



DELIVERABLE D5.2.3
FULL DESCRIPTION OF THE
CROSSGRID STANDARD OPERATIONAL
PROCEDURES AND SPECIFICATION
OF THE STRUCTURE OF
DELIVERABLES
FINAL VERSION

WP5

Document Filename:	CG5.2-D5.2.3-v1.0-CYF020-StandardOperatingProcedures.doc
Work package:	WP5
Partner(s):	CYFRONET
Lead Partner:	CYFRONET
Config ID:	CG5.2-D5.2.3-v1.0-CYF020-StandardOperatingProcedures
Document classification:	PUBLIC

Abstract: This document presents the standard operating procedures for CrossGrid repository access, problem reporting, change requests and release preparation, along with naming conventions and indispensable standardization requirements based on EU DataGrid (EDG) experience.





Delivery Slip

	Name	Partner	Date	Signature
From				
Verified by				
Approved by				

Document Log

Version	Date	Summary of changes	Author
1-0-DRAFT-A	02/07/2002	Definition of the preliminary structure	Marian Bubak, Marek Garbacz, Piotr Nowakowski
1-0-DRAFT-B	08/07/2002	Draft version	Piotr Nowakowski
1-0-DRAFT-C	14/07/2002	New section on naming conventions	Piotr Nowakowski
1-0-DRAFT-D	24/07/2002	SQL naming conventions	Marek Garbacz, Marcus Hardt, Piotr Nowakowski
1-0-DRAFT-E	03/08/2002	Inclusion of remarks and amendments by R. Pająk, R. Wismüller, M. Kupczyk and M. Holger	Piotr Nowakowski
0.5	05/08/2002	New chapter on document naming	Marios Dikaiakos, Piotr Nowakowski
0.6	06/08/2002	New sections on CVS, Bugzilla and Autobuild	Marcus Hardt, Piotr Nowakowski
0.7	07/08/2002	New section on script naming	Marek Garbacz
0.9	08/08/2002	Proofreading, corrections	Marian Bubak
1.0	08/08/2002	Final version for internal review	Piotr Nowakowski

CONTENTS

1. EXECUTIVE SUMMARY.....	6
2. LIST OF ACRONYMS USED.....	7
3. CENTRAL REPOSITORY.....	8
3.1. REPOSITORY DESIGN	8
3.2. HOW TO USE THE CVS SYSTEM AT FZK	8
3.2.1. CVS access procedure – a general outlook	8
3.2.2. Detailed access procedure.....	10
4. PROBLEM REPORTING AND CHANGE REQUEST MECHANISMS.....	15
4.1. BUGZILLA AS AN ISSUE TRACKING TOOL.....	15
4.2. ISSUE REPORTING AND HANDLING	15
5. THE AUTOBUILD TOOL.....	17
6. RELEASE PREPARATION PROCEDURE.....	18
7. CROSSGRID NAMING CONVENTIONS FOR SOFTWARE DEVELOPMENT.....	20
7.1. NAMING CONVENTIONS FOR CROSSGRID SOFTWARE DEVELOPMENT	20
7.1.1. Packages	20
7.1.2. Classes and Interfaces	21
7.1.3. Methods	22
7.1.4. Constants and Enum types.....	22
7.1.5. Variables.....	23
7.1.6. Special considerations for C++ programming.....	23
7.1.7. Special considerations for C programming.....	23
7.1.8. Special considerations for Perl programming.....	24
7.2. DIRECTORIES	24
7.3. FILES.....	25
7.3.1. Java files	25
7.3.2. C++ files.....	25
7.3.3. C files.....	25
7.3.4. Perl files.....	25
7.4. SQL NAMING CONVENTIONS.....	26
7.4.1. Tables and views.....	26
7.4.2. Fields	26
7.4.3. Stored procedures.....	26
7.4.4. Triggers.....	26
7.4.5. Indexes.....	27
7.4.6. General SQL coding conventions	27
7.5. LIST OF FILE EXTENSIONS.....	27
7.6. COMMENT CONVENTIONS	28
7.6.1. Documentation comments:	28
7.6.2. Implementation Comments (C and C++ style).....	29
8. DOCUMENT NAMING CONVENTIONS.....	30
8.1. DOCUMENT FILENAME AND CONFIG ID.....	30
8.2. EXAMPLES	30
9. OTHER CONVENTIONS.....	32
10. DOCUMENT TEMPLATES.....	33



11. REFERENCES..... 34

1. EXECUTIVE SUMMARY

This document lists the proposals for Standard Operating Procedures (SOPs) with regard to repository set-up, issue tracking and software release policy, along with naming conventions. The individual topics are addressed as follows:

- The central code repository is envisioned as a CVS server in Karlsruhe (mirrored at Cyfronet), which will store and dispense all code and code-related documentation developed within the Project, with special regard for preparing software releases.
- The main issue tracking tool to be used is Bugzilla (this will be integrated with the CVS server at Karlsruhe).
- A special Autobuild tool is being adapted for CrossGrid use. Code developers are asked to maintain compliance with the Autobuild standard.
- The software release policy will consist of the following steps:
 - Coordination meeting
 - WP follow-up meetings
 - Software release workplan coordination
 - Middleware integration
 - Acceptance tests
 - Release

Each step is outlined in more detail in Chapter 6.

- Naming conventions are presented for task, module, class, method and attribute names; also for file names and comments. The rule of thumb is to use Java-style names for both Java and C++ (including ANSI C), while comments should be compatible with the Javadoc tool. These principles have been adopted from the EDG naming conventions. The overall goal is to develop reliable software that can easily be updated to accommodate unforeseen changes throughout the lifetime of CrossGrid Project (hereafter called the Project). The software should be clear and understandable, not only to its original author, but also to other collaborators. A uniform and consistent set of naming conventions should prevent naming conflicts when large and complex programs are created, using code from many different sources. Good programming practices can significantly reduce the debugging phase of program development and ensure easy integration into the overall system.

This document constitutes an M6 CrossGrid deliverable, but may be subject to updates and amendments basing on further information obtained during the products' development and testing. All updates are performed by the CrossGrid Architecture Team upon request from other Project members and an evaluation thereof. The current version of this document will always be available on the CG Architecture Team website ([CGTAT]).

2. LIST OF ACRONYMS USED

ANSI	American National Standards Institute
CVS	Concurrent Versioning System
EDG	European DataGrid
HTTP	Hypertext Transport Protocol
ISO	International Standardization Organization
QA	Quality Assurance
RDBMS	Relational Database Management System
SSH	Secure Shell
WP	Work Package
XML	Extended Markup Language

3. CENTRAL REPOSITORY

In order to plan, coordinate and develop subsequent releases of the CrossGrid software, a centralized code repository system must be created and maintained. It is imperative that individual tasks adopt the use of code repositories as part of their daily routines. While it is permissible for some low-level work to be carried out *in situ* (rather than being disseminated at the repository level – see below), subject to the decision of the individual task leaders, all new releases and documentation must be documented and uploaded to the repository. The repository plays a particularly important role with relation to subsequent software releases (see chapter 6), as it provides a common basis for the integration of individual modules.

3.1. REPOSITORY DESIGN

The main CrossGrid repository for code and documentation will be located in Karlsruhe (further referred to as FZK). The central repository is going to be a CVS server, to which all individual tasks will upload the results of their work. It is expected, that the code developed within all Work Packages will yield to automatic documentation tools (such as JavaDoc and Doxygen) and be accompanied by further, manually-written documentation. Therefore, the central repository must also maintain documents compliant with the CVS standard.

3.2. HOW TO USE THE CVS SYSTEM AT FZK

As of August 7, 2002, FZK is in the process of setting up a GNU version of the Sourceforge system, called Savannah ([SAVANNAH]). A sample project environment is available for review at <http://gridportal.fzk.de/projects/testproject/>. Task leaders are strongly encouraged to familiarize themselves with the system, as they will be expected to begin using it once the coding phase of the Project begins. The system provides a single point of entry for all code-related documentation and the code itself.

3.2.1. CVS access procedure – a general outlook

The Web interface will initially be divided into subsections, hereafter called *projects*. Each project corresponds to a single CrossGrid task. Should any task leader decide that it would be beneficial to split his tasks into several subprojects, he may request that an additional project be created and placed under his supervision. The current maintainer and administrator of the SourceForge system, along with its interface, the CVS repository and the issue tracking mechanism is Mr. Marcus Hardt (feedback@gridportal.fzk.de).

In order to obtain access to the system, a programmer must first obtain a SourceForge account. This is done via an automated form, available on the above mentioned website. Each project must have its own administrator (this would normally be the task leader). The administrator has the right to create and remove user accounts and grant access privileges within his project.

Project names and their associated CVS directories will correspond to the default names defined as part of Chapter 7, *CrossGrid Naming Conventions for Software Development* (see section 7.1.1., *Packages*).

Once the account has been created, a mail must be sent to the repository administrator, so that he may set the proper access rights for repository checkout/checkin operations. An automation of this process is in the works, but is not expected before September (see next section). Once the account has been activated, the user may begin submitting code to the repository. The preferred setup of local CVS clients is explained in detail on the FZK site and should be followed closely. A more general SourceForge tutorial is available at <http://gridportal.fzk.de/tutorial>. Detailed access procedures for the central CVS repository at Karlsruhe are provided in the next section.

The FZK site also contains integrated interfaces for bug reporting (via Bugzilla – see next section), discussion forums within each project, download of documents and help files.

A more specific issue has arisen with regard to the WP3 repository, which is currently located in Poznań (PSNC). The documents contained therein are available via CVS-WEB at <http://wp3.crossgrid.org/>. It has therefore been decided, that FZK will mirror that repository. Checkouts can be obtained from either PSNC or FZK, and all updated documents can be checked back into the PSNC repository, whereupon they will be automatically copied to the main repository at FZK. This change has been signed off upon by the Architecture Team and is subject to special arrangements between the PSNC task leader and the FZK repository manager.

A decision is pending whether to locate the backup repository at Cyfronet or in Valencia. This will be of no consequence for individual programmers. The backup repository will not provide a direct access interface and is only meant as a safeguard in case of data loss at the main FZK site.

Figure 3.1 presents a sample structure of the repository tree.

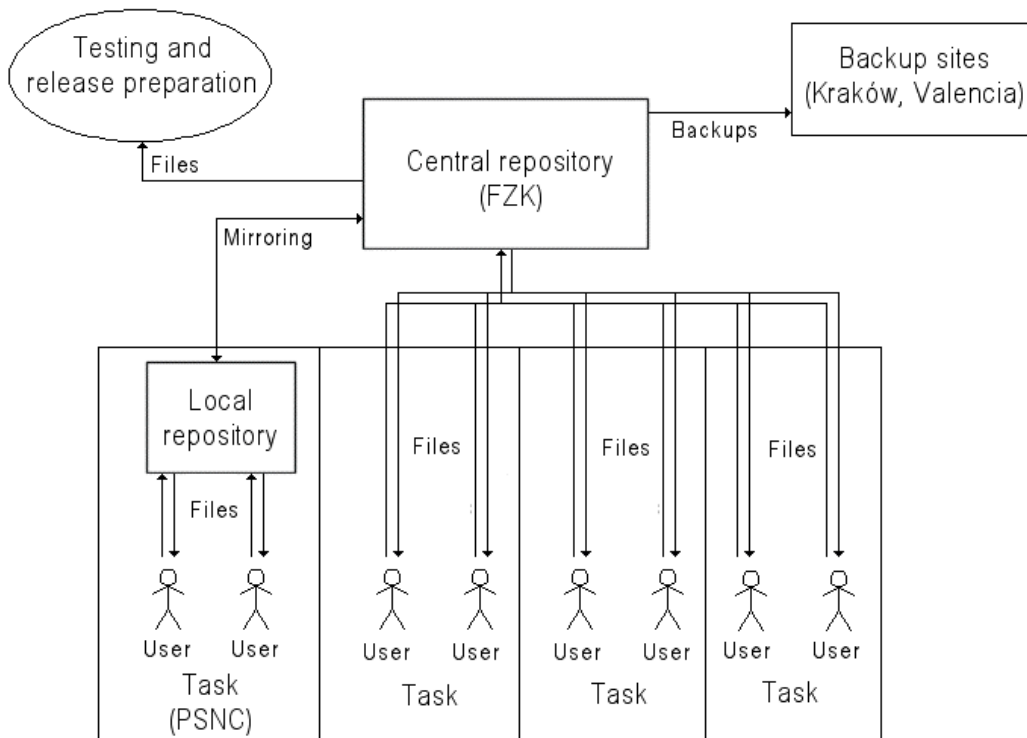


Figure 3.1: CrossGrid repositories

3.2.2. Detailed access procedure

The section below describes basic access principles as they relate to the CrossGrid CVS repository. Additional information can be found in the CVS Tutorial ([CVSTUT]), CVS Overview ([CVSOV]), on the CVS Plugin site ([CVSPLUG]) and on the graphics tools site ([CVSGRAPH]).

3.2.2.1. Prerequisites

The instructions provided here are based on the following assumptions:

- you are using Linux (might work on other platforms, but not tested),
- you have a CVS client installed,
- you have an ssh2 client installed.

3.2.2.2. Obtaining an account

To access CVS in a read-write manner you need to have an account at the central CVS repository on <http://gridportal.fzk.de/>. If you require read-only access to the repository, you can skip to the next step.

- Create a public-private key pair for ssh2 and follow the instructions given:

```
#> ssh-keygen -t dsa
```

NOTE: It is not absolutely necessary to perform this step, but it makes everyday work with CVS a lot more comfortable

NOTE 2: This is a preliminary solution. The eventual goal is to use your CrossGrid certificate to grant access to CVS.

- Visit https://gridportal.fzk.de/new_account and fill out the form to request an account at Gridportal. You will get an email containing your credentials as soon as the account is created.

3.2.2.3. Accessing the CVS Repository

To access the repository you have to specify where the repository is located and how you want to access it. This is done through two environment variables. In **bash** you would issue the commands:

```
#> export CVSROOT=:ext:<username>@gridportal.fzk.de:/cvs/crossgrid
#> export CVS_RSH=ssh
```

Replace **username** with **anoncvs** if you want read-only access. Note that you always commit changes as that user who checked out the files.

As a typical developer, you will most likely want to use the basic five CVS file operations:

- checkout,
- add,
- commit,
- update,
- remove,
- tag (for WP managers).

These functions will soon be explained in a case study.

First, we want to obtain a working copy of a module (directory) in the repository:

```
#> cvs checkout <modulename>
```

NOTE: You can find out the name of the desired module by visiting <http://gridportal.fzk.de/cgi-bin/cvsweb.cgi> and selecting the appropriate **CVSROOT**.

If you are a testbed administrator, you should try this module:

```
#> cvs co cg-interim-inst
```

You now have the complete tree stored under **cg-interim-inst** on your local hard drive, plus a directory called **CVS** in every sub-directory.

3.2.2.4. Adding files to the repository

Assume, you have created a couple of new files in the subdirectory **cg-interim-inst/sites/cyfronet**. You would have to add these files using:

```
#> cvs add <file>, ..|<dir>, ..]      or
#> cvs add          no <files> or <dir> means "use all the files in the current workdir"
```

The response should look similar to:

```
cvs server: scheduling file 'test' for addition
cvs server: use 'cvs commit' to add this file permanently
```

That means you still have to commit the files. The directories don't need to be committed.

Committing is similar:

```
#> cvs commit <file>, ..  or
#> cvs ci          where ci is short for commit
```

You will now be prompted to enter a message for the files you added. This is a comment for everybody to understand what this file is about and should not exceed 10 lines.

If the committing fails (because of inappropriate access rights), you will now be shown an error message.

3.2.2.5. Changing files

When you have changed a file in you working copy you should send these changes back to the repository. This is also done with the **commit** command we have just seen:

```
#> cvs ci <file>, ..      or
#> cvs ci                  which selects all changed files
```

IMPORTANT NOTE: It may happen, that another developer changes the same file you do and commits it ahead of you. In that case **cvs commit** will not work for those files. You have to update your working copy of the repository beforehand (see below). This will merge all changes to the file. This works in most cases, but if you processed the same section of a file simultaneously with someone else, you will encounter a merge conflict. The affected portion of the file might look like this (as updated by CVS):

```
void main ()
{
<<<<<<< test
    int ii;
=====
    int p;
>>>>>>> 1.4
}
```

This means your copy (upper part) differs with respect to revision 1.4 in the given section. You can only commit a file to the repository once all such conflicts are resolved.

3.2.2.6. Updating local copy

When many developers work simultaneously, it is common for the files in the repository to change, new files to emerge and still others to become stale. To keep your working copy up to date you should regularly run the **update** command:

```
#> cvs update -dP      or
#> cvs up -dP
```

This updates the directory structure down from your current working directory. This includes the creation of new directories (**-d**) and the deletion of old ones (**-P**).

3.2.2.7. Removing files

If you wish to remove a file from the repository you first have to delete the file locally and then run the following command

```
#> cvs remove [<file>, ..|<dir>, ..]
```

This file will then be deleted in all the other developers' working copies when they run **cvs up -dP** (unless there are changes in the file).

Note: Files or directories will never really be deleted, because we still want to be able to retrieve old revisions of every file. They will be stored in the “Attic”.

3.2.2.8. Tagging releases

When a piece of software reaches a certain level, it is very useful to "tag" a release. This is a simple operation when the current working copy is at the level you want to tag. You simply call:

```
#> cvs tag <tag-name>
```

This adds the tag **<tag-name>** to the current releases of files.

NOTE: Be sure not to confuse *release* and *revision*! (The term *version* is not used in CVS terminology).

Tags are useful because if you want to check out the release you tagged earlier you can get it by typing:

```
#> cvs checkout -r <tag-name>
```

CrossGrid software should be tagged by task leaders, so as to avoid confusion.

4. PROBLEM REPORTING AND CHANGE REQUEST MECHANISMS

Since the Technical Annex calls for every change in software code to be well documented, it is imperative that we use some sort of issue tracking tool. This becomes especially important for preparing code releases, when requirements present in one software module may influence alterations in another module.

4.1. BUGZILLA AS AN ISSUE TRACKING TOOL

There currently exist many issue tracking tools; we have, however, decided to use the BugZilla tool ([BUGZILLA]), the advantages of which are described below:

- open-source software (no registration fee necessary),
- utilized by other Grid projects (i.e. DataGrid),
- integrated, product-based granular security schema,
- inter-bug dependencies and dependency graphing,
- advanced reporting capabilities,
- a robust, stable RDBMS back-end,
- extensive configurability,
- a very well-thought-out natural bug resolution protocol,
- e-mail, XML, console, and HTTP APIs,
- Integration with automated software configuration management systems, including Perforce and CVS (through the Bugzilla email interface and checkin/checkout scripts).

4.2. ISSUE REPORTING AND HANDLING

The BugZilla mechanism, described above, will be integrated with the main SourceForge back-end at FZK. A separate menu will be made available within each project for reporting issues related to that project. Each issue is automatically forwarded to the designated *autoresponder* (usually the task leader) for further processing. Each issue will, at the minimum, consist of:

- the name and e-mail address of the originator,
- the name and e-mail address of the responder,
- issue severity (on a scale of 1 to 8),
- a text description of the issue.

Any Project member can raise an issue, and so it is expected that the WP4 Integration Team will sign up as members of each project handled by the FZK system. In addition to issues which may arise during integration testing, all internal errors and change requests may also be submitted through the

BugZilla tool. The issue reporting interface may be further divided into sections for bug reporting, change requests, patch requests and the like – this is still being investigated but will not affect the functioning of the mechanism as a whole.

5. THE AUTOBUILD TOOL

Autobuild is a tool being developed by the European DataGrid (EDG) project and will be adapted for use in CrossGrid. It is an automated build device used on the server side to obtain uniform builds from submitted code across all CVS projects. It requires some conformance on the part of code developers, but its advantages are manifold:

- it allows for nightly rebuilds of the CrossGrid code,
- it reports results on a Web page,
- API documentation is generated from source code and published on the Web page,
- an autocheck procedure will perform basic functionality tests,
- it creates packages (RPM) for distribution and publishes them on the repository website.

The Autobuild tool will be incorporated into the CVS repository at FZK and accessed by members of WP4 to construct code releases. In order to comply with the standard, each package must contain a Makefile, which provides the following targets:

- **ALL**: Default target, build the package (create executable and libraries).
- **INSTALL**: Install the software (default location has to be **/opt/edg**).
- **DIST**: Create a source **tar.gz** archive using the naming convention described in Source Packages Subsection.
- **RPM**: Generate RPM packages.
- **APIDOC**: Generate developer documentation (using Doxygen for C/C++ or JavaDoc for Java). This document should be found in **doc/apidoc** directory.
- **USERDOC**: Generate user documentation. It should be found in **doc/userdoc** directory.
- **CHECK**: Some automatic checking method will be implemented, but several levels of checking may be needed that will not be addressed by a single make target. This item has yet to be developed further.

To make compliance easier for developers, a sample project will be stored in the main CVS archive (see <http://gridportal.fzk.de/cgi-bin/cvswweb.cgi/cg-example/?cvsroot=crossgrid>).

Developers should frequently visit the status website (examples: <http://datagrid.in2p3.fr/autobuild/>, <http://datagrid.in2p3.fr/autobuild/rh6.2/edg-se-libhandler.html>) so that they can see if their software compiles on the reference platform using the latest interfaces to other tasks.

Marcus Hardt (FZK) is currently responsible for installing and maintaining the Autobuild tool and can be contacted for further details (feedback@gridportal.fzk.de).

6. RELEASE PREPARATION PROCEDURE

The CrossGrid Technical Annex provides for three successive iterations, each of which must end with a release of the system. For technical reasons, though, it may be beneficial to increase the total number of releases, to account for and test minor changes introduced in the implementation process of each module. The Architecture Team proposes the following release plan (exact dates of each sub-release will have to be established, possibly through consultations within the Steering Group):

- *Coordination meeting*: involves a discussion of the previous release and develops a framework for a subsequent release.

The coordination meeting should be organized after each release and be open to all Project participants, but entail the participation of WP task leaders, CrossGrid steering group and representatives of the Technical Architecture Team and WP4 Integration Team.

The end product of this meeting should be an evaluation of the release's suitability and quality, along with a prioritized list of issues concerning the next release, including: all software and documentation modifications with regard to the previous release and versions of external software (i.e. Globus Toolkit, DataGrid middleware, compilers etc.) to be used for the next release.

In preparation for the coordination meetings it is expected that each task will hold internal meetings to develop a common stance and deliver a cohesive presentation.

- *WP follow-up meetings*: individual tasks discuss the issues related to the next release and develop specific workplans.

Each task is expected to hold an internal meeting in order to develop its own workplan with regard to the next release, taking into account the results of the coordination meeting. The follow-up meetings should involve designations of responsibilities for individual participants.

A separate meeting should be held by the Architecture Team to gauge the influence of the coordination meeting on CrossGrid architecture. If significant changes are foreseen, an updated version of the architecture document must be developed.

- *Software release workplan coordination*: The CrossGrid Steering Group should develop a plan for the next release, upon discussion with individual WP leaders. The plan should include a list of all third-party software units which need to be installed on testbed sites in order to satisfy the upcoming release's requirements.
- *Middleware integration*: applications and middleware tasks upload the latest (internally tested) versions of their software to the central repository (Karlsruhe), where they are accessed by WP4.4 (Verification and QA) for testing. Integration issues are reported and fixed through the use of mechanisms described in chapter 4.

The WP4 Integration Team integrates the pertinent modules with external software units and oversees the basic integration tests as per the software release test plan described in WP4 deliverables.

- *Acceptance testing*: an integrated CrossGrid release is installed on Testbed sites for acceptance testing. The acceptance tests are deemed to have been passed when each scenario presented by the application groups is executed correctly.
- *Release*: The Integration Team holds a meeting to describe the release and indicate changes introduced since the last release. The release is accompanied by relevant documentation, developed by individual WPs and by the Integration Team.

The types of tests and the applicable procedures are further described in the WP4 deliverable D4.1 (appendix 4).[TEST]

Figure 6.1 presents the release procedure diagram.

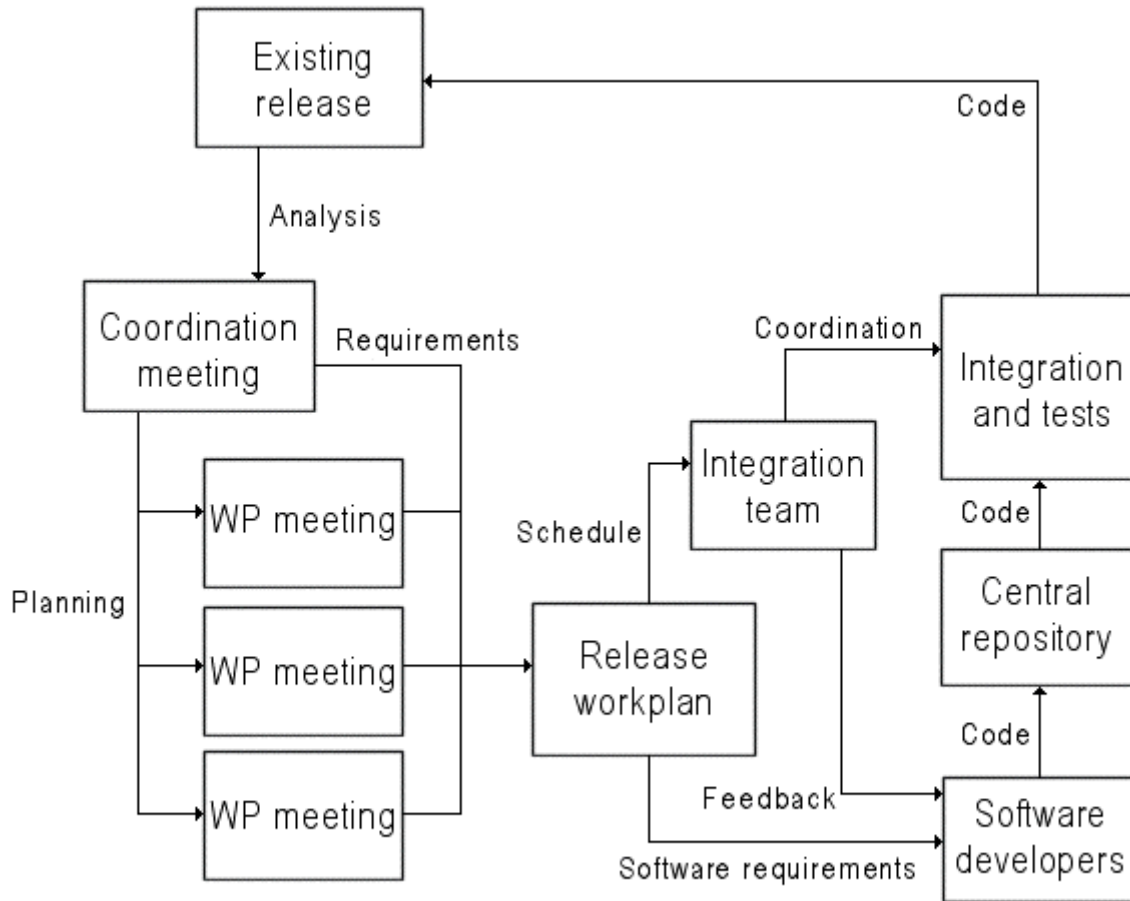


Figure 6.1: CrossGrid software release preparation procedure

7. CROSSGRID NAMING CONVENTIONS FOR SOFTWARE DEVELOPMENT

The use of naming conventions for labeling individual code components is essential in distributed projects, such as the CrossGrid Project. A uniform set of conventions enables participants to understand one another and makes it easier to isolate bugs and maintain final applications, along with middleware.

Since most of the CrossGrid code will be generated either in C/C++ or in Java, it is on these languages that the standardization effort must be concentrated. It so happens, that there are well-entrenched industry standards for writing Java code, while related C/C++ standards are, at best, haphazard. Therefore, it is proposed that CrossGrid use the Java naming policy for class, attribute and function names throughout the entire Project (this includes code written in C/C++).

Since we intend to subject our code to automated documentation tools (namely Javadoc), it is also common sense to adopt a Java-related convention for comments.

Exceptions to these rules, most notably associated with code which has been developed prior to the commencement of the Project, along with all imported code will be considered on a per-case basis.

Note: the naming conventions presented in this chapter have been largely inherited from DataGrid, where they are said to be performing adequately. We feel that it is common sense to adopt a working solution instead of creating one from scratch, but praise must be returned where it's due (see [DGNC]). For additional discussion of naming conventions see [CCON] and [BRAUDE].

7.1. NAMING CONVENTIONS FOR CROSSGRID SOFTWARE DEVELOPMENT

This section contains general naming rules (derived from Java, but applicable to other high-level programming languages; most notably C and C++). A list of dissimilarities between this general set of rules and specific rules for C and C++ is provided in sections 7.2.5 and 7.2.6.

7.1.1. Packages

The Java-derived convention is to prefix the package name with a reversed DNS name that is unique and registered. The prefix is always written in all-lowercase ASCII letters and should be according to the standard, one of the top-level domain names: currently com, edu, gov, mil, net, org, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981 ([ISO]).

For CrossGrid, the registered DNS name is eu-crossgrid.org then the package prefix is **org.crossgrid**.

Subsequent components of the package name define the scope of the package:

```
org.crossgrid.wp<wp_number>.<task_name>.<component>.<sub-  
component>.(...)
```

Table 7.1 contains the list of individual task names.

WP1.1 (biomedical application)	biomed
WP1.2 (flooding crisis team support application)	flood
WP1.3 (distributed data analysis in HEP)	hep
WP1.4 (weather and air pollution forecasting)	meteo
WP2.2 (MPI verification)	verif
WP2.3 (metrics and benchmarks)	bench
WP2.4 (performance monitoring)	perf
WP3.1 (Grid portals)	portals
WP3.2 (scheduling agents)	scheduling
WP3.3 (monitoring infrastructure)	moninfr
WP3.4 (selection/migration policies)	expert
WP3.5 (integration, testing and refinement)	integration

Table 7.1: Task names in CrossGrid

The components and subcomponents are named by the individual tasks.

Example: The biomed package of WP1 should be named thus:

org.crossgrid.wp1.biomed

Each package and component also corresponds to a separate directory on disk (see Chapter 3, *Central Repository*).

Every package must have a **README** or **README.html** file. This is where people will look first for basic information about the package. The file must contain the name of the package coordinator, a brief description of the purpose of the package, a brief list of classes and their functionality and a description of external dependencies (if any exist).

7.1.2. Classes and Interfaces

Class names should be nouns, in mixed case with the first letter of each internal word capitalized. No underscores are normally used in class names. Use singulars (i.e. **Window** instead of **Windows**), unless the purpose of the class is to collect objects (i.e. **Clients**).

Try to keep your class names simple and descriptive. Use whole words – avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or

html). For example, if there is a class for a visualization window provided by the biomed package, its full name should be written as

```
org.crossgrid.wp1.biomed.VisualizationWindow
```

(Please note that the sample classes and methods presented in this document bear no resemblance to reality and merely serve demonstrative purposes.)

Blanket exceptions may be permitted at the discretion of the programmer – for instance, **AAAAAAAuto** would be better expressed as **AAA_AA_A_Auto** instead of slavishly following the convention.

7.1.3. Methods

Methods should be verbs in mixed case with the first letter in lowercase and the first letter of each internal word capitalized.

Example:

```
org.crossgrid.wp1.biomed.VisualizationWindow.addSimulation()
```

The prefixes **get**, **set** and **is** should be used for accessor methods, as in **getName()**, **setName()** and **isBox()** (which returns a boolean value). The alternative syntax **get_name()**, **set_name()** and **is_box()** is permissible for C++ machine-generated code.

Use a consistent standard for operation. Single blank lines are useful for separating code sections within methods; a consistent standard is to use double blank lines between methods.

Within methods, the following standards should be considered:

- perform only one operation per line,
- try to keep methods to a single screen,
- use parentheses within expressions to clarify their meaning, even if they are redundant.

7.1.4. Constants and Enum types

According to Sun, the names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores ("_"). (ANSI constants should be avoided, for ease of debugging.)

Example:

```
static final int MIN_WIDTH = 4;
```

In C and C++ constants are declared differently but this naming convention is largely adopted for pre-processor constants. We suggest using it for C++ global constants and for all enum types (the capitalization of enum elements, however uncommon, still makes sense, because they are really just constants).

7.1.5. Variables

Variable-level naming conventions are left to task leaders as nowadays most if not all class attributes are declared private and manipulated through the use of accessor methods (see section 7.1.3).

Do NOT use environmental variables to pass external data into your code. Proliferation of “magic variables” results in code which is very difficult to understand and maintain. Use constants or command-line parameters.

7.1.6. Special considerations for C++ programming

In C++ we don’t have the notion of packages. The same logical functionality is provided by namespaces so it is natural to translate Java packages and components into C++ namespaces. The concept of classes is the same as in Java; Interfaces are a special kind of class in C++ (pure virtual classes).

7.1.7. Special considerations for C programming

Unfortunately C has neither packages nor namespaces nor classes so we propose to adopt the widely used convention of separating the package and component names with underscores. This results in rather long method names like in Globus. Since there are no classes the class is the same concept as a component for methods (functions). The corresponding data structure should still use the same naming convention as a class in Java and C++.

Table 7.2 presents samples of fully qualified method names in Java, C++ and C.

LANGUAGE	EXAMPLE	NOTES
Java	org.crossgrid.wp1.biomed.VisualizationWindow.addSimulation()	Follows the convention of starting with a reverse domain
C++	org::crossgrid::wp1::biomed::VisualizationWindow.addSimulation()	Use namespaces rather than prefixes
C	org_crossgrid_wp1_biomed_VisualizationWindow_addSimulation()	Pure C, so no namespaces

Table 7.2: Class names in Java, C++ and C

7.1.8. Special considerations for Perl programming

We advise following Java conventions wherever possible. In order to decrease the possibility of accidentally introducing a new variable because of a typo, we advise always using strict type checking. For the sake of readability try to use English: readable global variables are much preferred to perlesque gibberish, i.e.:

- **\$PROGRAM_NAME** instead of **\$0**,
- **\$ARG** instead of **\$_**,
- **\$PROCESS_ID** instead of **\$\$**.

Perl is not type-safe and this often causes confusion and errors. Use a limited prefix notation for such common basic types as arrays, hashes and file handles:

- If an array is referred to, precede the reference with the **a** prefix, e.g. **aAtoms**, **aChains**.
- If a hash is referred to, precede the reference with the **h** prefix, e.g. **hNames2Places**, **hChains**.
- If a FileHandle object is referred to, precede the reference with the **fh** prefix, e.g. **fhIn**, **fhOut**, **fhPdb**.
- If either an input stream or an output stream is referred to, precede the references with the **ist** and **ost** prefixes accordingly.

In Perl, the notion of modules allows grouping particular pieces of code together, to be reused by others. We advise naming modules in a Java-like way, e.g. any Perl module(s) for Task 1.1 should be named **org::crossgrid::wp1::biomed**.

7.2. DIRECTORIES

In Java, each package and component corresponds to a directory as well. We propose to adopt this rule to organize the directory structure in the CrossGrid Project.

```
<reverse domain name>/wp<wp_number>/<task name>/<component>/<sub-component>/(...)
```

In our example:

```
org/crossgrid/wp1/biomed/
```

7.3. FILES

7.3.1. Java files

Each Java source file contains a single public class or interface. When private classes and interfaces are associated with a public class, you can put them in the same source file as the public class. The public class should be the first class or interface in the file.

Java source files have the following ordering:

- beginning comments, disclaimers;
- package and Import statements;
- class and interface declarations.

7.3.2. C++ files

We propose to adopt the Java file convention for C++ include and source files, i.e. to have a separate file for each public class (including closely associated classes). Include statements should specify the relative component path to avoid potential conflicts between modules.

Example:

```
#include "crossgrid/wpl/biomed/VisualizationWindow.h"
```

The ordering is the following:

- beginning comments, disclaimers;
- include statements;
- class declarations (header files) and implementations (source files).

7.3.3. C files

The C file conventions do not differ from the C++ file conventions except that the concept of a class is a logical concept for C. If the functions and data elements have been identified that would correspond to a class in the Java or C++ sense, then they belong in the same header and source files.

7.3.4. Perl files

We propose to adopt the Java file convention for Perl module files, i.e. to have separate file for each Perl module, including module public functions together with any other functions/variables needed by the module to operate. When importing a module using **use** clauses, a fully qualified name of the module has to be used, e. g. **use "org::crossgrid::wpl::meteo"**.

7.4. SQL NAMING CONVENTIONS

Several applications developed for use with CrossGrid will contain databases written and accessed through SQL. Since there are numerous SQL naming conventions, none of which is intrinsically wrong, we believe that a minimum set of common standards should be enforced within CrossGrid.

7.4.1. Tables and views

Table and view names should be in lowercase, but capitalized. Use plurals (i.e. **Customers** instead of **Customer**). Do not use underscores (**StoredSessions**, not **Stored_Sessions**). Avoid nondescriptive abbreviations and acronyms. The name of each table should be readily understandable to all its users. Never use words which relate to the physical characteristics of the database (such as **Table**, **File**, **Records** etc.) The names of intermediary tables in *m:n* relationships should be simple concatenations of their base table names (such as **CustomersOrders**). Avoid names which promote misinterpretation of the tables' contents.

Correlation names in queries may be used as needed and are not subject to naming conventions.

7.4.2. Fields

Individual field names are expressed in singulars (**ClientAddress**, not **ClientAddresses**). Use lowercase, but capitalize. Avoid abbreviations and acronyms, save for the most obvious ones (i.e. **DocumentURL**). Prefix field names with a singularized version of their parent table's name. Use a proper relational database design methodology; avoid multivalued, multipart and calculated fields. If a table contains an artificial primary key, it should be named *<singularized table name>ID* (for instance, **ClientID**)

7.4.3. Stored procedures

The names of stored procedures should always reflect the actions they perform (i.e. **GetClientNames** or **AuthorizeUser**). Avoid abbreviations and acronyms.

7.4.4. Triggers

Triggers cannot function independently of tables, and so the name of each trigger will begin with the name of its parent table. That name should be followed with an underscore (**_**), the name of the operation with which the trigger is associated and the string **_trigger**, as in: **Clients_insert_trigger**, **Clients_update_trigger** and **Clients_delete_trigger**.

7.4.5. Indexes

Like triggers, indexes always depend on their parent tables. Therefore, the name of the index should begin with a table name, followed by an underscore, the name of the indexed column (if more than one column is used, all column names should be concatenated) and the string **_index**, as in:

- **Clients_ClientName_index**
- **Clients_ClientExtensionClientHomePhone_index**

7.4.6. General SQL coding conventions

Write generic SQL keywords in UPPERCASE. Split long queries into multiple lines along clause divisions, as in:

```
SELECT CustomerName, CustomerAddress, CustomerHomePhone, CreditCardNumber
FROM Customers, CreditCards
WHERE Customers.CustomerID=CreditCards.CustomerID
ORDER BY CustomerName
```

Use indents for subqueries.

7.5. LIST OF FILE EXTENSIONS

Table 7.3 contains a list of permissible filename extensions for various programming languages used in the Project.

EXTENSION	DESCRIPTION
.java	Java source file
.class	Java bytecode
.cc (.cpp)	C++ source files
.c	C source files
.h	C/C++ header files
.inl	C++ inline function files
.idl (.odl)	Interface description language
.pl	Perl source files
.pm	Perl module files
.sql	SQL DDL/DML command file

Table 7.3: File extensions

7.6. COMMENT CONVENTIONS

Programs can have two kinds of comments:

- Documentation comments (known as "doc comments") are delimited by `/**...*/`. Doc comments can be extracted to html files using the Javadoc tool.
- Implementation comments are those found in C/C++, which are delimited by `/*...*/`, and `//` (C++ only).

The following paragraph describes the types of comments and suggested uses for them.

7.6.1. Documentation comments:

Doc comments are meant to describe the specification of the code, from an implementation-free perspective, to be read by developers who might not necessarily have the source code at hand.

Use documentation comments immediately before declarations of interfaces, class, member functions and fields to document them.

Documentation comments are processed by Javadoc to create external documentation for class. For C++ code, Doxygen should be used as it understands the Javadoc conventions (Doxygen also supports its own style of comments, but that should be avoided in CrossGrid).

Doc comments describe classes, interfaces, constructors, methods, and fields. Each doc comment is set inside the comment delimiters `/**...*/`, with one comment per class, interface, or member. This comment should appear just before the declaration:

```
/**
 * The Example class provides ...
 * (...)
 */
public class Example { ...
```

For further details, see *How to Write Doc Comments for Javadoc* [JAVADOC]. Automatic html documentation creation tools for C and C++ also use the Javadoc standard [DOXYGEN].

Document methods with a description of the following [BRAUDE]:

- what the method does,
- why it does what it does,
- what parameters must be passed (use `@param` tags for Javadoc),
- what exceptions are thrown (use `@exception` tags for Javadoc),

- reason for choice of visibility (**private** etc.),
- ways in which instance variables (i.e. attributes) are changed,
- known bugs,
- test description, stating whether the method has been tested and in what manner (locations of test scripts would be useful),
- history of changes,
- example of use,
- special considerations for threaded and synchronized methods.

APIs should be documented through Javadoc, as per their descriptions provided in design documents (M6 deliverables).

7.6.2. Implementation Comments (C and C++ style)

C and C++ comments can be used to document implementation specifics that are not of concern to the external user. As usual, it is recommended to comment on the implementation as much as possible.

8. DOCUMENT NAMING CONVENTIONS

This section describes the document naming rules. These rules are to be applied in the CrossGrid Project for developer-generated documentation (i.e. deliverables, minutes, presentations etc.) A set of document naming conventions did exist before (and was used – with limited success – for naming M3 deliverables). Unfortunately, there *two* separate proposals in circulation and neither was clearly defined, resulting in chaos. Therefore we propose a new standard, which – we hope – will eliminate all ambiguities and be widely adopted within CrossGrid (this standard will be mirrored in the CrossGrid QA Plan).

8.1. DOCUMENT FILENAME AND CONFIG ID

Below is the standard grammar for naming CrossGrid documents:

<Document name> = <project name><WP/task ID>--<document type>-<version>-<lead partner ID>-<description>.<extension>

<Config ID> = <project name>-<WP/task ID>-<document type>-<version>-<lead partner ID>-<description>

<Project name> = CG

<WP/task ID> = <WP number>.<task number> | <WP number>.<task number>.<subtask number> */* Requested by several task leaders */*

<Lead Partner ID> = <boldface partner code><document number>

<boldface partner code> = CYF | ICM | INP | INS | UVA | SAS | LINZ | FZK | UST | TUM | PSNC | UCY | DAT | TCD | CSIC | UAB | USC | DEMO | AUT | LIP | ALGO

<document number> = <three-digit document counter value, specific to a particular partner>

<Document type> = D<deliverable ID> */*deliverable*/* | Min | Pub | Pres | Rep | Tem | Other

<Deliverable ID> = <WP number>.<task number>.<deliverable number>

<Version> = v<major revision number>.<minor revision number>

<Description> = <Short text describing the nature of the document (do NOT use spaces!)>

<Extension> = doc | tex | txt | ppt | xls | <other extension>

Please note that the **.doc** filename extension presented on the title page of a document does not necessarily need to change to **.pdf** if a conversion takes place. **.pdf** files should be created only for release purposes, since editing them is cumbersome and requires specialized software.

8.2. EXAMPLES

Here are some application examples of the document naming grammar presented above:

CG Standard Operating Procedures (Deliverable D5.2.3)

Document name:

CG5.2-D5.2.3-v1.0-CYF020-StandardOperatingProcedures.doc

Config ID:

CG5.2-D5.2.3-v1.0-CYF020-StandardOperatingProcedures

CG Steering Group meeting minutes (session 18)

Document name:

CG5.1-Min-v1.0-CYF218-SteeringGroup18.doc

Config ID:

CG5.2-Min-v1.0-CYF218-SteeringGroup18

M6 Two-part financial report

Document names:

CG5.1-Rep-v1.0-CYF302-M6FinancialReport.doc

CG5.1-Rep-v1.0-CYF302-M6FinancialReport.xls

WP3 M6 Design documents (Deliverable D3.2, with each task submitting one or more design documents)

Document names:

CG3.1-D3.2-v1.0-PSNC020-PortalsDesign.doc

CG3.2-D3.2-v1.0-UAB020-SchedulingAgentsDesign.doc

CG3.3-D3.2-v1.0-TCD019-DesignSummary.doc

CG3.3.1-D3.2-v1.0-TCD020-SantaGDesign.doc

CG3.3.2-D3.2-v1.0-CYF021-OCMGDesign.doc

CG3.3.3-D3.2-v1.0-CYF022-JiroDesign.doc

CG3.3.4-D3.2-v1.0-ICM007-ICMSystemDesign.doc

(etc.)

Please note that the document counter values may vary (the values provided above serve only as examples)! For Deliverable D3.2 and other design documents the new naming conventions will come into force when the final versions are mailed to Cyfronet for dissemination (following reviews).

9. OTHER CONVENTIONS

Being a European project, CrossGrid utilizes the standard MKS unit convention (Meter-Kilogram-Second). Units should always appear in documentation (it is acceptable to provide unit names in column headers of a table instead of adding them to each field). The datetime standard is GMT, with 0 corresponding to 00:00 GMT 1 January 1970.

10. DOCUMENT TEMPLATES

CrossGrid utilizes a standardized document template for all documents written in MS Word for Windows. The default template is available on the Architecture Team website (http://kinga.cyf-kr.edu.pl/~tat/template_docs.html). This template has been in use since the commencement of the Project.

11. REFERENCES

- [BRAUDE] Eric J. Braude, *Software Engineering: An Object-Oriented Perspective* (John Wiley & Sons Inc., 2001)
- [BUGZILLA] The Bugzilla Project (<http://www.bugzilla.org/>)
- [CCON] Coding Conventions for C++ and Java (<http://www.macadamian.com/codingconventions.htm>)
- [CGTAT] The CrossGrid Architecture Team website (<http://kinga.cyf-kr.edu.pl/~tat>)
- [CVSOV] CVS Detailed Overview (http://www.loria.fr/~molli/cvs/doc/cvs_toc.html)
- [CVSGRAPH] Graphical User Interfaces for CVS (<http://cvsgui.org/>)
- [CVSPLUG] The CVS Plugin for EMACS (http://www.loria.fr/~molli/cvs/pcl-cvs/pcl-cvs_toc.html)
- [CVSTUT] The CVS Tutorial (http://www.loria.fr/~molli/cvs/cvs-tut/cvs_tutorial_toc.html)
- [DGNC] The DataGrid Naming Conventions draft (available at http://edmsoraweb.cern.ch:8001/cedar/doc.info?document_id=328838)
- [DOXYGEN] The Doxygen website (<http://www.doxygen.org/>)
- [ISO] ISO Standard 3166 (<http://www.din.de/gremien/nas/nabd/iso3166ma/>)
- [JAVADOC] The Javadoc website (<http://java.sun.com/javadoc>)
- [SAVANNAH] The Savannah Project website (<http://savannah.gnu.org/>)
- [SF] Sourceforge.net (<http://sourceforge.net/>)
- [TEST] CrossGrid WP4.1 appendix 4 (available at <http://www.eu-crossgrid.org/M3deliverables.htm>)