



SOFTWARE ENGINEERING TOOLS
PROPOSAL OF TOOLS AND APPROACHES TO BE USED

WP5.1

Document Filename: **CG-5.1-DES-008-SoftToolsDesc.doc**
Work package: **WP5.1**
Partner(s): **CYFRONET**
Lead Partner: **CYFRONET**
Config ID: **CG-5.1-DES-0008-1-0-FINAL**
Document classification: **PUBLIC**

Abstract:



Delivery Slip

	Name	Partner	Date	Signature
From				
Verified by				
Approved by				

Document Log

Version	Date	Summary of changes	Author
1-0-DRAFT-A	15/03/2002	Draft version	CrossGrid Architecture Team (M. Bubak, K. Zając, M. Malawski, M. Garbacz)
1-0-DRAFT-B	21/05/2002	Proofreading	Marian Bubak, Katarzyna Zając, Maciej Malawski, Mariusz Garbacz, Piotr Nowakowski

CONTENTS

1. INTRODUCTION.....	4
2. REQUIREMENTS MANAGEMENT.....	5
2.1. CHARACTERISTICS OF REQUIREMENTS	5
2.2. TOOLS FOR REQUIREMENTS MANAGEMENT.....	5
2.3. AVAILABLE TOOLS.....	6
3. ISSUES TRACKING.....	7
3.1. BUGS VS. ISSUES	7
3.2. EFFECTIVE BUG TRACKING PROCESS.....	7
3.3. WHAT SHOULD WE EXPECT FROM A BUG-TRACKING TOOL.....	8
3.4. PRIORITY AND SEVERITY OF AN ISSUE	9
3.5. EXAMPLES OF TOOLS AVAILABLE	9
4. SOFTWARE DESIGN.....	11
4.1. TYPES OF UML DIAGRAMS	11
4.2. AVAILABLE TOOLS.....	12
5. CONFIGURATION MANAGEMENT.....	13
6. SOFTWARE TESTING.....	14
7. REFERENCES.....	15

1. INTRODUCTION

This document contains an overview of the software engineering tools than can potentially be used in the CrossGrid project. Some general characteristics of particular SE activities are given along with suggested approaches and methodologies. For each type of tool the expected features are laid out.

Obviously the content of this document is based on up-to-date knowledge about detailed project activities and plans. Since these are going to be modified/aligned, some suggestions will be modified as well. This will particularly concern such domains as software testing. Some other activities like issue tracking are common to many different types of projects, so they can be described in more detail even now.

All suggestions/proposals concerning the tools and technologies that are going to be used are very welcome.

2. REQUIREMENTS MANAGEMENT

2.1. CHARACTERISTICS OF REQUIREMENTS

It is an obvious fact that the requirements gathering and analysis phase is one of most important project activities. To build something, we must first understand what that “something” is going to be. The process of understanding and documenting this “something” is called “requirements analysis” [Braude]. Requirements generally express what an application is meant to do; they do not usually try to express how to accomplish these functions. On the other hand, a general requirement often translates into several more detailed requirement(s), so there are exceptions to the general rule that requirements should avoid specifying how something must be done. In the case of the CrossGrid project a significant part of the requirements will probably be defined on a low level of abstraction: they will describe particular mechanisms that have to be used, libraries, etc.

It is obvious that requirements must be written down. In the CrossGrid project we intend to use the IEEE 830-1998 Standard Template [Std 830] to accomplish this. According to the documented practice, each specified requirement should have the following characteristics [Braude]:

Each requirement must be:

- Expressed properly
- Made easily accessible
- Numbered
- Accompanied by a test that verifies it
- Provided for in the design
- Accounted for by code
- Tested in isolation
- Tested in concert with other requirements
- Validated by testing after the application has been built

2.2. TOOLS FOR REQUIREMENTS MANAGEMENT

Specialized tools can make capturing and managing requirements easier; e.g., by sorting, prioritising, assigning, and tracking them. One benefit of such tools is that we know who is working on what requirement at what time. The tools are also a means for project leaders to determine what the status of particular requirements is, who is working on them, etc. Another important functionality offered by those tools is requirements traceability, i.e. the ability to identify related requirements. It is particularly important for any changes that are made in the requirements: the impact analysis of each change must base on information about related requirements.

Using automated documentation generation tools (like Doxygen or Javadoc) facilitates automatic incorporation of references from specified classes, modules, functions etc. into the appropriate requirements (that is, the sources for specified code parts). This allows us to avoid duplicating requirement descriptions. The concept presented using the following example (see below) is described in [Braude].

```
/**  
<a href="ReqAnal\# EngagingForeignCharacter">  
    Engagement Requirement 1  
    ("Engaging a foreign character")  
</a>  
.... other comments ...  
*/
```

```
The purpose of this method is stated in the SRS.  
The purpose is not repeated in the source code.
```

```
Public engageForeignCharacter (...)  
{  
    ...  
}
```

For small projects, Web-based spreadsheets can be used, as long as they describe particular requirements together with their states. However, in the case of the CrossGrid project this approach would be inefficient because of administrative overhead needed to maintain and update the file(s).

2.3. AVAILABLE TOOLS

There are many requirements tools available, although most of them seem to be commercial in nature. Some examples are:

1. Rational RequisitePro
2. DOORS
3. RTM

3. ISSUES TRACKING

3.1. BUGS VS. ISSUES

The term “issue” has a much broader meaning than just a bug found in a software product. Not all “issues” are *defects*, which are more commonly known as "bugs." Some issues will be *feature requests*, *enhancement requests*, *patch submissions*, or even *tasks*. For this reason, we feel the term "issue" is more appropriate than "bug". People often say “bug” when they should really be referring to an issue. A centralized database of open issues makes it possible for people involved in the project to avoid duplicating one another’s work and to - possibly - help out or provide feedback.

3.2. EFFECTIVE BUG TRACKING PROCESS

Bugs are part of every product development process. How can bugs found during product development (and afterwards) be tracked? Bugs that are found but not properly tracked might slip through and be discovered by the customers. It is essential to maintain a single, centralized database of all issues in order to eliminate the possibility of duplication of work on a specified issue. To prevent bugs from slipping through and to avoid unnecessary work duplication, the testing and development team should work together to eliminate any bugs, possibly using a bug-tracking tool. The Bug Life Cycle Model (see Figure 1) explains how to efficiently use a bug tracking tool:

1. New bugs, enhancements and features (a.k.a. *Issues*) are submitted to the bug tracking tool by the testing team or product manager.
2. The product manager or team leader assigns a priority and severity to each new issue and defers the issue to a specific programmer.
3. The programmers fix the issues that have been assigned to them. The fixed issues’ status is changed to “Fixed” in the bug tracking tool.
4. R&D releases a new internal version with the new features and fixed issues.
5. The testing team checks whether all issues that have been marked as fixed are really fixed.
6. The testing team closes the fixed issues in the bug tracking tool. New bugs are submitted and the process is repeated from step 1.

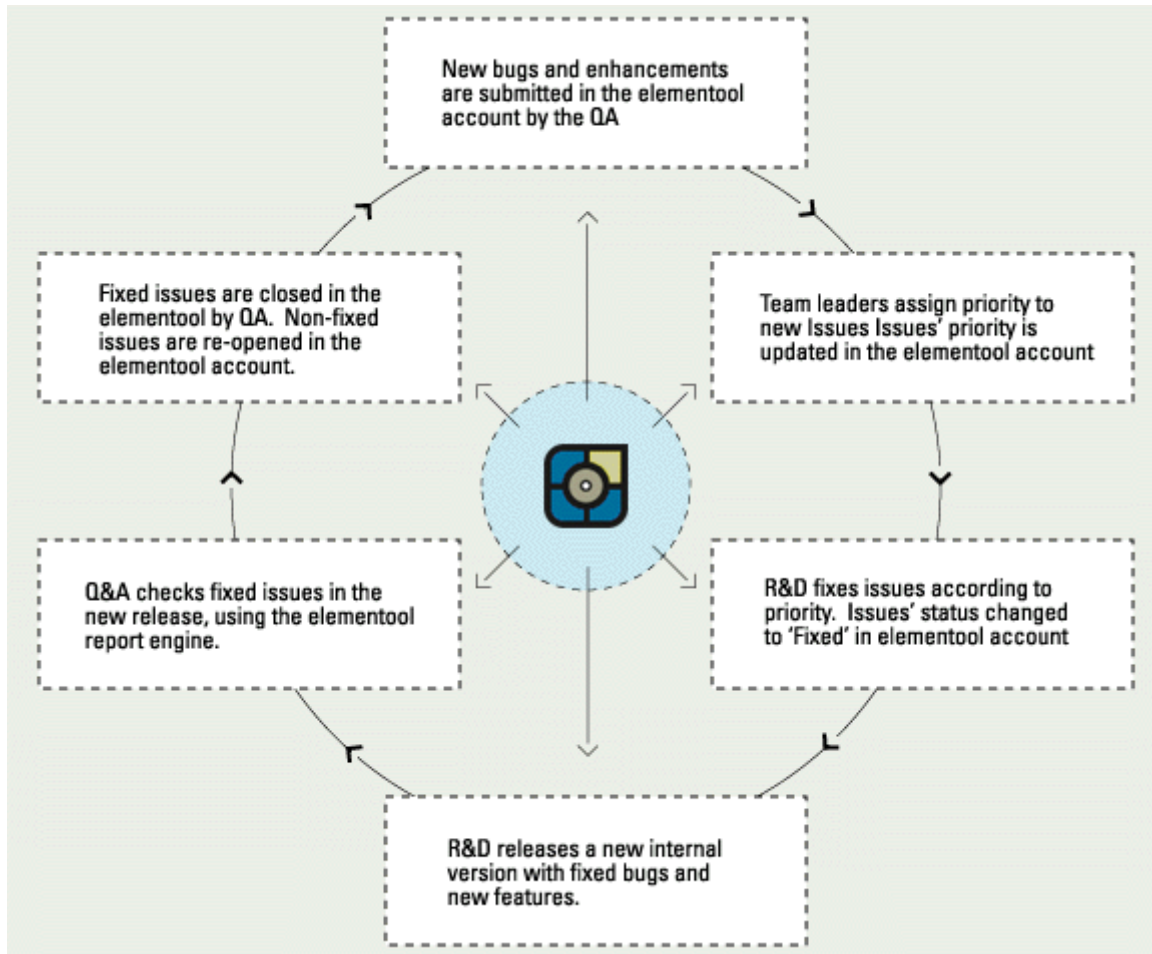


Figure 1. Lifecycle of a bug. Source: www.elementool.com

3.3. WHAT SHOULD WE EXPECT FROM A BUG TRACKING TOOL

Several important benefits are expected from a bug tracking tool:

- 1.E-mail notifications – whenever an issue is assigned to a group member. It would be very useful if a bug tracking tool could send an automatic email notification informing the programmer that a new issue has been assigned to him/her.
- 2.File attachments – the ability to attach screenshots, text files etc. that provide the other team members with a clear description (and examples) about the problem that has been reported. Complete and clear descriptions of a problems reduces the overhead involved in communication between the originator of a bug and the person assigned to resolve it.
- 3.History Trail – automatically tracks and displays all the changes made in a specific issue from the moment it is submitted throughout its whole lifecycle.
- 4.Full customisation – It is important that the bug tracking tool should enable group members to fully customize the different forms used for submitting issues, according to the project's special needs.

5. Powerful reports – a report engine must enable group members to search the issue list for different words and phrases, create focused reports (limited to the issues of interest to particular programmers/users - using AND/OR/NOT), set fields that should be included in the Report Query and set fields that should be part of the Issue Report.

6. Submitting issues directly from a website – this option enables customers or team members to submit issues to the bug-tracking tool using a form that is placed on a website, without the need to login to the tool.

7. Web-based – enables access to the tool and issue list from different locations using only a browser.

3.4. PRIORITY AND SEVERITY OF AN ISSUE

There are two primary attributes that are assigned to every issue: priority and severity. They help in determining which issues are the most urgent and therefore should be resolved first.

Priority: A priority classification of a software error is based on the importance and urgency of resolving the error. The priority classification is as follows:

- **Immediate** – The bug should be resolved immediately.
- **High** - This bug should be resolved as soon as possible in the normal course of development activity, before the software is released.
- **Medium** – This bug should be repaired after serious bugs have been fixed.
- **Low** – It can be resolved in a future major system revision or not be resolved at all.

Severity: A severity classification of a software error is based on the degree of the error's impact on the operation of the system. The severity classification is as follows:

- **Critical** – The bug causes a failure of the complete software system, subsystem or a program within the system.
- **High** - The bug does not cause a failure, but causes the system to produce incorrect, incomplete or inconsistent results or it impairs the system's usability.
- **Medium** – The bug does not cause a failure, does not impair usability, and does not interfere with the smooth work of the system and related programs.
- **Low** – The bug is an aesthetic issue, an enhancement or a result of non-conformance to a standard.

3.5. EXAMPLES OF TOOLS AVAILABLE

Here are some examples of available issue tracking tools:

1. www.zope.org
2. OpenTrack (www.accurev.com). Open source
3. IssueZilla - IssueZilla, a proprietary tool, is based on Mozilla's open-source Bugzilla, but has been generalized by CollabNet to handle all kinds of issues, not just code-based issues. On openoffice.org,

The supported issue types now include: DEFECT, ENHANCEMENT, FEATURE, TASK and PATCH.

4. Bugzilla (bugzilla.mozilla.org) – open source. It is continuously being improved. Its webpage says: Bugzilla has matured immensely, and now boasts many advanced features. These include:

- 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-understood and well-thought-out natural bug resolution protocol
- Email, XML, console, and HTTP APIs
- Available integration with automated software configuration management systems, including Perforce and CVS (through the Bugzilla email interface and checkin/checkout scripts)

5. PVCS Tracker (www.merant.com)

4. SOFTWARE DESIGN

We propose that UML (Unified Modelling Language) be adopted as a standard in defining and drafting project designs. UML has numerous advantages, some of which are listed below (by G. Booch):

- It is an open standard.
- It supports the entire software development lifecycle.
- It supports diverse applications areas.
- It is based on experience and needs of the user community.
- It is supported by many tools.

The following UML characteristics make it useful:

- UML supports system modelling based on the object model.
- UML notation can be used for specification, construction, visualization and documentation of software artifacts.
- UML is designed to be readable/understandable for both humans and machines.
- UML notation lays a direct bridge between the conceptual model and executable software.

4.1. TYPES OF UML DIAGRAMS

There are several different types of UML diagrams, comprising several classes.

Structure diagrams

- *Class Diagrams* - [static] class structures (classes, interfaces, relationships, etc.)
- *Object Diagrams* - [static] snapshots of class instances

Behavioral diagrams

- *Use Case Diagrams* - modelling of system requirements
- *Sequence Diagram* - [dynamic] time-ordered system interactions
- *Collaboration Diagram* - [dynamic] interaction in the system based on structural organization of the objects sending and receiving messages
- *Statechart Diagrams* - [dynamic] states, transitions, events, and activities
- *Activity Diagrams* - [dynamic] a special case of Statechart Diagrams, addressing the flow of control in the system

Implementation diagrams

- *Component Diagram* - [static] dependencies between components
- *Deployment Diagram* - [static] deployment view of the system

4.2. AVAILABLE TOOLS

Just like with requirements management tools, there are many commercial products available for UML modelling. Most of them have the ability to generate code from designs and *vice versa*: draw a design based on existing code. The latter can be used for documenting the code.

Two examples are: Rational Rose and Together Control Center. In order to ascertain the price range for such tools, we present the current pricing sheet for Together Control Center:

Actual prices are as follows (the annual mandatory support fees are given in parentheses):

Together Solo

Node-locked license USD 3.495 (+ 699)

Floating license USD 5.592 (+ 1.119)

Together ControlCenter

Node-locked license USD 5.995 (+ 1.199)

Floating license USD 9.592 (+ 1.919)

All prices are net prices independent of any operating system.

Premium Support includes priority product support by phone and e-mail, and all updates and upgrades of the product.

There also exist open source products, such as ArgoUML (argouml.tigris.org). ArgoUML seems to be a reasonable choice, because of the following product properties (repeated after product developers; further verification pending):

- Supports open standards: XMI, SVG and PGML
- Created in 100% Java, thus platform-independent
- Open Source allows to extend or customize it.
- [...] ArgoUML is based directly on the UML 1.3 specification

5. CONFIGURATION MANAGEMENT

Configuration Management is undoubtedly one of the most important software engineering activities which have to be performed. It is particularly necessary when working in a distributed environment, such as the CrossGrid.

The general functionality offered by CM tools is broadly known so we won't describe it here. Suffice to say that the suggested tool (CVS) allows branch-based development.

During the WP3 kick-off meeting held on 29th-30th Jan 2002 our colleagues from the Poznań Supercomputing Center presented a "Proposal of using the Control Version System for the CrossGrid project". CVS seems to be a reasonable choice because of its availability and a rich feature range, rivalling that of many commercial products. Another advantage is the presence of experienced CVS specialists (Poznań Supercomputing Center, others...) within the CrossGrid project.

It is important to have a hierarchical structure of repositories. According to the CVS documentation [CVS], with CVS version 1.10, a single command cannot recurse into directories from different repositories. Still, the development versions of CVS allow one to check out code from multiple servers into his/her working directory.

Detailed procedures concerning access to the repositories as well as retrieving particular versions of the software and distributing the product releases have to be established.

6. SOFTWARE TESTING

It is quite rare for off-the-shelf software to be directly used in the testing process. Typically, companies prefer to create their own testing workbenches using a combination of purchased and locally implemented tools. Testing workbenches are therefore invariably open systems, which evolve to suit the needs of the system being tested [Somerville].

Generally, the functionality implemented by the testing tools can be divided as follows [Braude]:

1. *Recording and playback of mouse and keyboard action.* Testing a user interface using this method can be inefficient because even small changes within a GUI element can derail the recorded sequences of actions.
2. *Running tests scripts repeatedly.* It saves time which would otherwise be needed for performing the same tests with varying parameters
3. *Recording of test results.*
4. *Recording of time usage.* This can be very useful for performance analysis.
5. *Managing regression testing.* A class of tools called file comparators can be used for this. Test results of sequential runs of the same test are recorded and then compared in order to determine potential changes.
6. *Generating of test reports.*
7. *Generating of test data.* Data for testing can be selected from a database or generated randomly using patterns.
8. *Measuring of memory usage.* This can be helpful in detecting possible memory leaks.
9. *Managing of test cases.* This involves keeping track of test data, expected results, program facilities tested and so on.

The most reasonable approach is to wait until particular product requirements are specified. Once this happens, the appropriate test cases will have to be developed. Based on test cases we will be able to decide what kind of test tools will be the most appropriate for project needs.

Any policies concerning unit and integration testing will have to be established in advance. For example, it should be stated that specified components are tested as units prior to being included in integrated subsystems, which will then undergo further tests.

7. REFERENCES

[Somerville] – Ian Somerville, “Software Engineering, Fifth Edition”. Addison-Wesley, 1995, ISBN 0-201-42765-6

[Braude] – Eric J. Braude, “Software Engineering An Object-Oriented Perspective”. John Wiley & Sons, 2001, ISBN 0-471-32208-3

[Std 830] – IEEE Standard for Software Requirements Specification, Std. no. 830-1998.

[CVS] – Cederqvist et al, “Version Management with CVS”. www.cvshome.org