A PATTERN-BASED CONTINUOUS INTEGRATION FRAMEWORK FOR DISTRIBUTED EGEE GRID MIDDLEWARE DEVELOPMENT

A. Di Meglio, J. Flammer, R. Harakaly, M. Zurek, CERN, Geneva, Switzerland
E. Ronchieri, INFN, Italy

Abstract

Software Configuration Management (SCM) Patterns and the Continuous Integration method are recent and powerful techniques to enforce a common software engineering process across large, heterogeneous, rapidly changing development projects where a rapid release lifecycle is required. In particular the Continuous Integration method allows tracking and addressing problems in the software components integration as early as possible in the release cycle. Since new incremental code builds are done several times per day, only small amounts of new code is built and integrated at relatively short intervals. Developers are immediately notified of arising problems and integrators can pinpoint configuration and build problems to the level of single files within any given software component. This paper presents the implementation and the initial results of the application of such techniques in the SCM and Integration of the EGEE Grid Middleware software [1]. The software is based on a Service Oriented Architecture model where services are developed in different programming languages by development groups in several European locations under stringent quality requirements. A number of basic SCM patterns, such as the Workspace, the Active Line, the Repository, are introduced and the Continuous Integration tools used in the project are presented with a discussion of the advantages and disadvantages of using the method.

PATTERNS AND THE EGEE PROJECT

The concepts of pattern and pattern language were originally developed in the ‘70s in the context of architectural design [2]. Patterns were later successfully applied to software in the ’90 in the context of OO design [3]. More recently the concepts have been applied to other areas of the software development cycle and in particular to architecture, software configuration management and quality assurance [4].

A pattern can be generically defined as “a proven solution to a known problem in a defined context”. A pattern language is a set of pattern that together defines the usage context for all collaborating patterns.

The EGEE gLite project [1, 5] is tasked to produce a set of quality middleware services for grid computing by re-engineering existing components and developing new functionality. It counts more than 80 developers in 8 European countries producing software in Java, C, C++, Perl and other languages for several different platforms, including Linux and Windows (see Table 1).

Table 1: Single Lines Of Code (SLOC) per language

Language	**SLOC**
Java	~122.500 (29%)
C++	~121.000 (28%)
C	~105.500 (25%)
Perl	~50.500 (12%)
Other	~20.500 (6%)

The requirements of the Software Configuration Management system and the Build system are necessarily complex. The systems must be platform independent, language independent and allow reusing existing software component and build tools.

The following sections describe how the implementation of a carefully chosen set of SCM patterns has helped achieving the goals.

CONFIGURATION MANAGEMENT

The first fundamental task in the software development management is to provide a consistent, reproducible environment where source code can be stored and versioned. Additionally, the source code must be available to all developers and several different configurations may have to coexist without conflicts between them and with the host build system.

The task can be achieved by application of the *workspace* and *repository* patterns. The workspace pattern and its derived companion, the *private workspace*, identify an area on a computer where all configuration management and build operations can be carried out. Each workspace is like a sand box containing everything needed to produce a version of the software including both the project source code and all its external dependencies. More workspaces can be created on a single computer and live together without conflicts even if they host completely different versions of the project software.

The repository pattern identifies a common, well-know place where source code and external dependencies are stored and versioned. The repository can itself have a derived pattern called private repository that act as a cache inside a private workspace.

The gLite configuration management system is based on the open source CVS (Concurrent Version System) program [6]. Although CVS is a powerful system with good support for concurrent remote development, it lacks built-in support for workspaces and storage of large binary packages (for example external dependencies for the project) is not recommended (slow access and retrieval). CVS has therefore been extended by introducing the concept of *Configuration Specification Files*. In addition a more generic concept of repository has been introduced in the gLite infrastructure.

A Configuration Specification File (CSF) is a description of all the component of the project in CVS together with a particular tag. The CSF itself is stored in CVS and tagged. Whenever a version of the project must be produced, the corresponding CSF can be checked out and processed to fill the local workspace with the consistent set of all components making up the project. Different CSF can be created to extract only portion of the entire project, such as subsystems or maintenance packages. In addition, the CSF can be augmented to include also the external dependencies taken from one of the available repositories.

The repository pattern has been generalized by applying it not only to the CVS repository, but also to other types of software repositories. In particular a repository hosted on a web server provides access to the versioned binary packages of the project external dependencies. In the same way the CPAN [7] repository for Perl modules has been included. The set of the CVS repository, the web-based repository and CPAN forms together a consistent and reproducible infrastructure for software storage and versioning.

QUALITY GUIDELINES

One of the major requirements of the gLite project is the enforcement of quality assurance guidelines. Quality spans from naming conventions to coding guidelines for all the languages used in the project (Java, C, C++ and Perl), from unit tests to test coverage and metrics collection and reporting.

Most of the quality guidelines have been enforced by introducing in the SCM system and the Build system a set of automated checks and default targets that are consistently applied to all components (CVS modules) of the project.

The source code is verified for compliance using a number of tools, such as Jalopy [8] for Java, CodeWizard [9] for C/C++ and PerlTidy [10] for Perl. Whenever a developer commits code to CVS, the rules are applied and the developers are sent an e-mail message with the result of the analysis. Commits can be allowed or prevented depending on the status of the project and the current policies. However, it is important in this case to keep the process and the actual checks light enough, so that they do provide a useful service and do not annoy the developers, tempting them not to commit new code for long period of time.

In the same way, during private code builds the code is checked for compliance, so that developers can immediately see what changes may be necessary.

Unit tests and coverage analysis are performed on all components during the builds using tools such as JUnit [11] and CppUnit [12] for the unit tests and Clover [13] and gCov [14] for the coverage analysis.

All quality checks reports together with a number of software metrics about code size and size stability, complexity, fragility and so on are collected during the build in a common directory for further publication to the gLite project server.

THE BUILD SYSTEM

As previously mentioned, the build system must satisfy a complex set of requirements about its platform and language independence, flexibility and extensibility. In addition, the system must be able to run in the same way on central build servers to produce reference builds and on the developers' computers for local integration tests.

To achieve such goals the *build* and *private build* patterns have been implemented using a platform-independent, Java-based build engine called Ant [15]. In addition, a set of extensions has been developed in Java or Ant xml scripts to abstract the more platform-specific tasks into generic platform-independent commands.

The core of the build system is made of a set of xml scripts and properties files for the Ant system stored in a module in CVS called the *system* module. In order to run the build it is enough to have Ant installed and check the system module out of the CVS repository. This approach allows performing easily the same build process on the servers and on individual development computers. In addition, with small exceptions, the entire system works on all platforms where Java is supported, making it easy to port the build system to other platforms and thus satisfying the platform independence requirement.

The general structure and behaviour of the project CVS modules have been specified in the project SCM Plan in order to enforce homogeneity across the entire project. In particular the directory structure and a number of default targets have been defined in the project SCM plan. All components are required to comply with this basic set of rules in various ways depending on the language and build tools used.

The agreed common build process and the build

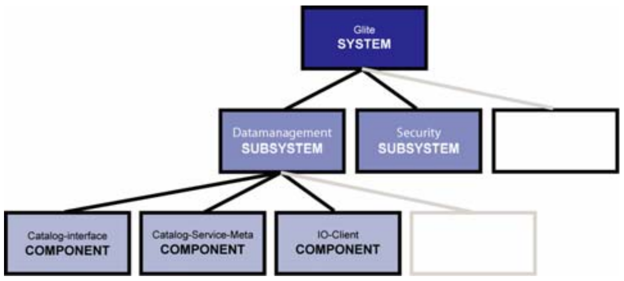


Figure 1: The system 3-tier structure

configurations are implemented in the files of the system module. All other modules are organised in a 3-tier hierarchical structure (system-subsystem-component) and import the common files, therefore sharing the same process and configurations (see Fig. 1).

In order to satisfy the language independence requirement and still be able to run a build of all the different components using the common configurations, the concept of *target wrappers* has been introduced. Each module in the project can be developed in any one of the supported languages using the appropriate build tool (Ant for Java, make and Autotools for C and C++, Perl makefile.PL, etc.). The target wrapper files exports the standard common targets defined by the SCM Plan, but have a different implementation of such targets depending on the underlying module language and build tool. For example, for Java the implementation is a native sequence of Ant tasks to perform initialization, quality assurance checks, compilation, unit testing, etc. For C++ on the other hand, the implementation may consist in a set of Ant commands that delegate the equivalent operation to autoconf, automake and make.

Special scripts have been provided to create and add to the hierarchy new subsystems and components, providing the users with preconfigured stubs of the required Ant xml files and easing the task of importing existing components into the system.

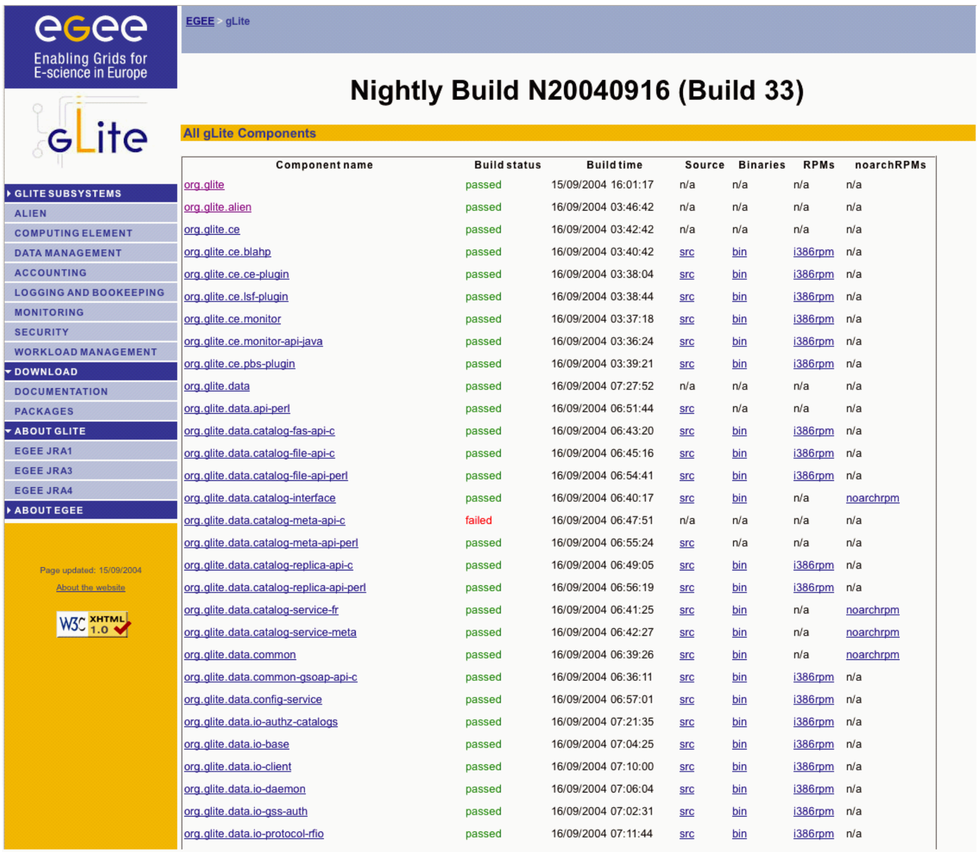
The system can easily be extended to other languages and build tools by providing a target wrapper file with the common interface and an appropriate implementation.

An additional pattern, the *stage area*, is used to facilitate dependency resolution within a workspace without the need for installing software or setting environment variables and allowing multiple workspaces to coexist on the same computer for parallel development or maintenance. All modules install their build products (binaries, libraries, etc) in a common area in the workspace and are configured to look for dependencies in this area during the build.

THE CONTINUOUS INTEGRATION LOOP

The *continuous integration loop* pattern is the core of the rapid development process of gLite. One of the major shortcomings of large distributed projects is the lack of integration until very late in the lifecycle and a final ‘big bang’ phase of frenetic troubleshooting. The continuous integration loop avoids this problem by continuously building, integrating and smoke testing all project components.

In the gLite system the software is built incrementally every 60 minutes. Whenever a problem is found, the developers who committed changes since the last



Nightly Build N20040916 (Build 33)

All gLite Components

Component name	Build status	Build time	Source	Binaries	RPMs	noarchRPMs
org.glite	passed	15/09/2004 16:01:17	n/a	n/a	n/a	n/a
org.glite.alien	passed	16/09/2004 03:46:42	n/a	n/a	n/a	n/a
org.glite.ce	passed	16/09/2004 03:42:42	n/a	n/a	n/a	n/a
org.glite.ce.blahp	passed	16/09/2004 03:40:42	src	bin	i386rpm	n/a
org.glite.ce.ce-plugin	passed	16/09/2004 03:38:04	src	bin	i386rpm	n/a
org.glite.ce.lsf-plugin	passed	16/09/2004 03:38:44	src	bin	i386rpm	n/a
org.glite.ce.monitor	passed	16/09/2004 03:37:18	src	bin	i386rpm	n/a
org.glite.ce.monitor-api-java	passed	16/09/2004 03:36:24	src	bin	i386rpm	n/a
org.glite.ce.pbs-plugin	passed	16/09/2004 03:39:21	src	bin	i386rpm	n/a
org.glite.data	passed	16/09/2004 07:27:52	n/a	n/a	n/a	n/a
org.glite.data.api-perl	passed	16/09/2004 06:51:44	src	n/a	n/a	n/a
org.glite.data.catalog-fas-api-c	passed	16/09/2004 06:43:20	src	bin	i386rpm	n/a
org.glite.data.catalog-file-api-c	passed	16/09/2004 06:45:16	src	bin	i386rpm	n/a
org.glite.data.catalog-file-api-perl	passed	16/09/2004 06:54:41	src	bin	i386rpm	n/a
org.glite.data.catalog-interface	passed	16/09/2004 06:40:17	src	bin	n/a	noarchrpm
org.glite.data.catalog-meta-api-c	failed	16/09/2004 06:47:51	n/a	n/a	n/a	n/a
org.glite.data.catalog-meta-api-perl	passed	16/09/2004 06:55:24	src	n/a	n/a	n/a
org.glite.data.catalog-replica-api-c	passed	16/09/2004 06:49:05	src	bin	i386rpm	n/a
org.glite.data.catalog-replica-api-perl	passed	16/09/2004 06:56:19	src	bin	i386rpm	n/a
org.glite.data.catalog-service-fr	passed	16/09/2004 06:41:25	src	bin	n/a	noarchrpm
org.glite.data.catalog-service-meta	passed	16/09/2004 06:42:27	src	bin	n/a	noarchrpm
org.glite.data.common	passed	16/09/2004 06:39:26	src	bin	n/a	noarchrpm
org.glite.data.common-gsoap-api-c	passed	16/09/2004 06:36:11	src	bin	i386rpm	n/a
org.glite.data.config-service	passed	16/09/2004 06:57:01	src	bin	i386rpm	n/a
org.glite.data.io-authz-catalogs	passed	16/09/2004 07:21:35	src	bin	i386rpm	n/a
org.glite.data.io-base	passed	16/09/2004 07:04:25	src	bin	i386rpm	n/a
org.glite.data.io-client	passed	16/09/2004 07:10:00	src	bin	i386rpm	n/a
org.glite.data.io-daemon	passed	16/09/2004 07:06:04	src	bin	i386rpm	n/a
org.glite.data.io-gss-auth	passed	16/09/2004 07:02:31	src	bin	i386rpm	n/a
org.glite.data.io-protocol-rfio	passed	16/09/2004 07:11:44	src	bin	i386rpm	n/a

Figure 1: CruiseControl build results andgLite package list

successful build are notified of the problem by means of an e-mail message containing a pointer to a web interface to the build reports.

All problems are therefore found within minutes of the introduction of a change and can be immediately fixed, keeping the entire code base healthy. Also in this case special care should be taken in setting reasonable notification policies. If the notifications are too frequent, developers may either be tempted not to commit their code frequently in fear of receiving too many build failure messages, or could ignore the messages altogether.

Additional nightly builds and weekly integration build are also run to produce tagged baselines of the code base.

The gLite loop is implemented using a product called CruiseControl [16]. CruiseControl is essentially a scheduling and reporting engine on top of Ant. It can be used to schedule any number of builds at regular intervals or at given times and to optionally collect and publish build artifacts and reports to a common web application running in Tomcat.

Although CruiseControl was mainly designed for Java software development, it was rather straightforward to adapt it to the gLite build system that already uses Ant for a variety of languages and build tools. In addition, it is a Java application and therefore satisfies the platform independency requirement of the project.

PACKAGING AND DISTRIBUTION

All components of the gLite project are required to be packaged in a variety of formats, namely source tar balls, binary tar balls, RPMs for RH Enterprise 3 and binary compatible distributions like Scientific Linux and CentOS and MSIs for Windows.

In order to maintain the consistency of the system and set correct dependencies between RPMs and MSIs sets, a packaging engine has been developed and integrated with the build system.

The packaging tasks are defined as one of the default mandatory targets for all modules. Source and binary tar balls are created for all components using the same procedure. RPMs and MSIs package creation is abstracted in a packaging target developed in Java as an extension of the standard Ant tasks.

The specification files for RPMs and MSIs are generated automatically by default by the packaging engine harvesting information from the build system configuration files. This information includes for example the list of dependencies between packages. If more specific tailoring is required, preinst, postinst, preun and postun scripts can be used. If necessary the entire spec file can be provided by the component developer to override the automatic spec file generation.

All packages are produced as part of the continuous integration builds and the nightly and weekly builds. The packages produced for nightly and weekly builds are then automatically copied to the gLite web site [5] for distribution together with the build and QA reports.

CONCLUSIONS

In this paper a pattern-based framework for continuous integration and quality assurance of a large distributed software development project has been presented.

After the first four months of practical applications both advantages and disadvantages have been noticed.

In particular the rate of the continuous integration loop has to be carefully balanced between the need for finding problems as early as possible and avoid flooding the developers with build error notifications. In addition, the enforcement of a software process and quality assurance procedures in the build system must be balanced against the need for flexibility and rapid prototyping to avoid killing the process itself.

However, the benefits of the application of the system are clearly emerging, most notably a better integration of diverse component on multiple platforms, faster problem tracking and resolution and a general improvement in the quality of the software compared to other similar projects in which the authors have taken part.

ACKNOWLEDGEMENTS

This work has been funded by the European Communities under contract INFSO-RI-508833. The paper has been written thanks to and on behalf of the gLite Extended Integration Team.

REFERENCES

[1] E. Laure, et al., Middleware for the Next Generation Grid Infrastructure, proceedings of the CHEP 2004 conference (2004)
[2] C. Alexander, S. Ishikawa, M. Silverstein, A Pattern Language: Towns, Building, Construction, Oxford University Press (1977)
[3] E. Gamma, et al., Design Patterns, Addison-Wesley Professional; 1st edition (January 15, 1995)
[4] S.P. Berczuk and B.Appleton, Software Configuration Management Patterns, Addison-Wesley Pub Co; 1st edition (November 4, 2002)
[5] http://www.glite.org
[6] https://www.cvshome.org
[7] http://www.cpan.org
[8] http://jalopy.sourceforge.net
[9] http://www.parasoft.com
[10] http://perltidy.sourceforge.net
[11] http://www.junit.org
[12] http://cppunit.sourceforge.net
[13] http://www.cenqua.com/clover
[14] http://gcc.gnu.org/onlinedocs/gcc/Gcov.html
[15] http://ant.apache.org
[16] http://cruisecontrol.sourceforge.net