GN3 Software Developer Best Practice Guide 4.0

Last updated:	07-09-2012
Activity:	SA4
Dissemination Level:	PU (Public)
Document Code:	GN3-09-186v6
Authors:	Branko Marovic (AMRES), Marcin Wrzos (PSNC), Marek Lewandowski (PSNC), Antoine Delvaux (DANTE), Tihana Žuljevic (CARNET), Candido Rodriguez (RedIRIS), Rade Martinovic (AMRES), Spass Kostov (BREN), Ognjen Blagojevic (AMRES), Waldemar Zurowski (DANTE), Paweł Kędziora (PSNC), Ian Barker (DANTE), Gina Kramer (DANTE)

Table of Contents

Table of Figures

Table of Tables

1 Introduction

The GN3 Software Developer Guide provides guidelines and recommendations on how to collaborate within GN3 and use the tools involved in the development lifecycle (these are the tools that were identified by SA4 Task1 [DS4.3.1]). This includes:

- Setting up an integrated development environment and share workspace.
- Using the version control system.
- Building and releasing software.
- Using coding standards and best practices.
- Documenting software.

This guide should be used by all roles involved in GN3 software development (e.g. developers, managers, technical authors etc.).

2 The Software Development Infrastructure

In GN3, multiple international software development teams will work together to create and enhance the software functionality required in GN3. To maximise both inter-team and intra-team efficiency and effectiveness, a software development infrastructure has been implemented to provide the required tools and supports collaboration [DS4.3.1]

Recommended access is via Secure Shell with authentication using public keys. To gain access to an SVN repository, a nominated coordinator should contact the GÉANT Development Software Infrastructure Service Desk [SD] or log an issue online [SWDSD]. The preferred way of contact is to log an issue.

2.1 Workspace Usage Recommendations

The workspace is the environment in which source files are edited, software components are built, tested and debugged. The workplace usually integrates version control, build, and release management tools, which all comprise the software configuration management (SCM) system. Best practices related to workspace usage include [PERF]:

- **Do not share workspaces.**
  A workspace should have a single purpose, such as an edit/build/test area for a single developer, or a build/test/release area for a product release task. Sharing workspaces confuses people, and compromises the software configuration system's ability to track activity by user or task.

- **Do not work outside of managed workspaces.**
  The SCM system can only track work in progress when it takes place within managed workspaces. SCM systems generally use workspaces to facilitate some of the communication among developers working on related tasks.

- **Retain control over your workspace.**
  It is not recommended to create a volatile workspace where the actual files are replaced by a view into somebody else's workspace (for example by using symbolic links), with intention to simplify your work. As the result, the changes of the workspace contents are caused by external events beyond your control, and the very purpose of the workspace as a place that integrates your activities and tools is broken.

- **Synchronise your workspace often.**

  Frequent updating of workspace content and merging of changes made by others improves communication and integration of changes performed by team members. The workspace update procedure must therefore be simple and short. Otherwise developers will be tempted to avoid synchronisation.

- **Publish (commit) your work often.**

  Finished changes should be checked in as soon as they are ready and the required verifications are passed. This makes the work and workspace changes available to other team members and allows them to synchronise with your changes. Time-consuming pre-publishing validation procedures must be avoided. Therefore, the verification should be limited to the local build and basic tests.

2.1.1 Setting up a Development Environment

Many Integrated Development Environment (IDE) settings depend on user preferences or project requirements. The settings described in this section cover some of the key recommendations made in Section 5, Coding Guidelines. The Eclipse IDE comes with a pre-defined profile for Sun's Java Code Conventions, which only needs to be slightly modified:

- For spacing, it is mandatory to use only spaces (not tabs).
- The maximum line width can be set to 100 or 120 instead of 80 (this is optional). As wide screens are getting popular, larger text widths are also not uncommon.
- You can change the indentation width using the number of spaces defined for the project (typically 2–4 characters).

To change the settings:

1. In Eclipse, click **Window -> Preferences**.
2. Navigate to **Java -> Code Style -> Formatter**.
3. Click **New**.
4. Set the profile name to "Java Conventions [modified]".
5. From the **Initialize settings** drop-down list, select **Java Conventions [built-in]**.
6. Ensure that the checkbox for opening the edit dialog is checked.
7. Click **OK**.
8. Click **Edit**.
9. Select the **Indentation** tab.
10. Under **General Settings -> Tab Policy**, select **Spaces only** from the drop-down list.
11. Select the **Line Wrapping** tab.
12. Enter a value for the maximum line width.
13. Click **Apply**, and rename the code profile when you are prompted.
14. Click **OK**.

Any code that you now write will automatically be formatted using the conventions you have set. To reformat old code with your conventions, select the code you want to format and press Ctrl+Shift+F.

2.1.2 Sharing Workspace Configuration

Maven [MAVEN] allows the creation of artifacts that act as project skeletons. The GN3 SA4, Task 3 Software Development Infrastructure provides a Maven repository to store such artifacts and make them publicly available. These and other related workspace elements can be shared through the provided SVN repository. It is up to the software projects to define which parts of configuration should be shared and kept in version-control software or managed through Maven skeletons. The infrastructure can also provide a parent Maven project for all other GN3 Maven-using projects to share common Maven configuration settings.

Recommended access is via Secure Shell with authentication using public keys. To gain access to an SVN repository, a nominated coordinator should contact the GÉANT Development Software Infrastructure Service Desk [SD] or log an issue online [SWDSD]. The preferred way of contact is to log an issue.

3 Version Control Management

Version Control Management is the management of changes to any type of file (for example, source code or configuration files). This is crucial in team collaboration, where a number of team members change the same file. If a change is made to a file, it is identified by a number or a letter code (revision number). Each revision number is associated with a timestamp and the person who made the change. Individual revisions can be compared, merged or restored [RVN]. By locking or merging changes, a Version Control Software (VCS) serialises performed changes.

Version Control Management is crucial in the development lifecycle because it makes it possible to develop different branches of the software simultaneously. Later these independent branches can be merged. Since bugs and other issues usually relate to specific versions, it is very important to be able to retrieve, compile and run different versions of the software. Most revision control systems can use delta compression which store only changes between successive revisions.

In GN3 Subversion (SVN [SVN]) is used for Version Control Management (this tool has already been successfully used in GN2).

When using the version control system, it is recommended that you:

- Only commit code that compiles and passes unit tests.
- Make sure that different commits relate to different business logic (different functionality, modules, etc – different commits).
- Comment each of your commits to SVN (e.g. with an issue number).

To minimise conflicts (simultaneous modifications of the same part of source code), each module (source code package, class or file) should be assigned to a single developer only. However, some conflicts will occur when different branches (versions) of the developed module are merged. In this case, conflicts should be solved by the developer who owns the conflicted module.

Merging operations should be performed separately from regular development. At the point of merge, the source code of the main development and merged branches is frozen, and commits that do not solve existing conflicts are forbidden. The sequence of merging is important. First, independent modules are merged, then modules that depend on the merged modules. Synchronisation and cooperation between developers is crucial, as only one developer can solve existing conflicts between two developments at any time. This can be enforced, for example, by using a ticketing system (where only the owner of the ticked is permitted to commit).

The best practices for code versioning are:

- Establish trunk and branch sets for development (feature) and maintenance branches, as well as read-only tags, as detailed in Section 3.3 Recommended SVN Layout for Individual Projects. Codelines within each of these groups should have the same policy.
- Define a policy for each codeline group. Establish clear rules for documenting, reviewing, and testing changes. Allow fair use and permissible modification by using differentiated policies among codelines.
- Define a policy manager. This person can decide, document and advise developers on policy changes or exceptions when the policy is inapplicable or ambiguous. However, with clear and elaborated policies in place, this should rarely happen.
- Give each feature branch an owner. This person is ultimately responsible for the branch's maintenance, updates, merges, and final destiny.
- Branch only when necessary. Do not branch for trivial changes. Before branching, consider the branch maintenance costs: builds, checks, updates against the trunk, periodic and final merges.
- Do not establish independent copies of the code. Apart from check-outs, do not make editable or reference copies outside of your VCS, nor check in the copies of existing files as new ones. Use branches instead.
- Branch on incompatible code changing and versioning policy. For example, besides establishing new branches upon release, a new branch in an appropriate branch group should be set up whenever a line of development that requires less frequent commits or different sanity checks or visibility is started.
- Branch late. It is better to branch after the originating branch has reached maturity, so that further massive changes are not expected. This will reduce the branch maintenance costs caused by later propagation of changes. For example, a release branch is started after the exhaustive testing of the trunk is completed and most bugs fixed. A feature branch should start from the stable code in order to minimise merging and interference of earlier bugs with the new development.
- Branch instead of freeze. Branching decouples different aspects of development. For example, the developers of long-term branches and release maintainers are not constrained by freezes and subsequent testing of the trunk or the release candidate.
- Synchronise branches often. Regular merging of a branch with the main development is important, since it makes the final merge into the trunk much easier by keeping two codelines better aligned and reducing the number of potential conflicts.
- Merge as soon as the branch is complete. If possible, merge before the original codeline creates a new wave of changes.
- Merging should be done by the best prepared person. The owner of the target files and the person who made the changes are likely do a better merge than someone else.
- Merge branches, freeze code and produce releases by following and maintaining a release plan and respecting your release cycle schedule.

3.1 Additional Methods for Managing Change

Versioning and branching are not the only means for managing evolution and providing modular software development. There are often situations where newly added features are optional because some users or application scenarios do not need them, or because they are provided for testing or evaluation purposes. Such features can be managed by using configuration and dynamic branching within the code or plugins. In other words, branching of code versions, which is static from the point of its editing and execution, can be complemented with more dynamic means like configuration, conditional execution, or the use of alternative or pluggable code (where different classes or modules implement a common interface).

For example, availability, visibility or use of some feature may be managed by a configuration switch. However, this also requires the checks of the switch value to be added to the code, together with access to this switch or the whole configuration. This approach does not substitute version control branching, as it does not support simultaneous maintenance of several versions of the code. Also, inserted configuration switch checks, passing of parts of configuration context in additional arguments, and conditional branches spread all over the code greatly affect code readability and maintainability. Addition or modification of even a small feature can affect several layers of software architecture, of which not all support optionality or conditional execution. For example, a database row either exists or does not, and can be of only one type. Finally, if the change is decided to be permanently incorporated into production code, all code related to the checking of the switch needs to be removed manually. Therefore, when some non-trivial change or option (which is expected to become a part of the main codeline) is checked, it is better to establish a feature branch.

Merging code branches is an opportunity for double-checking the changes made in a branch, how they match with the trunk, and whether they may have unintentional consequences on the main codeline. The moment of merging also provides the opportunity to decide whether to bracket the new feature with a configuration option. If some new feature is expected to be provided as an option, it is better to prepare for this before branching. The impact of the change can be localised by refactoring the code in the trunk before the branch is created. This is usually done by additional decomposition and decoupling through change of class responsibilities, where all necessary hooks are provided through the new or extended interface implemented by delegated classes. After this change, the trunk code may rely on the original code that is moved to a new class created for that purpose, while the new feature can be independently developed in a separate set of classes and source files. Which class or plugin will be actually used is determined from the configuration. Such refactoring can reduce the need for branching, or, if it is still necessary, greatly simplify merging.

3.2 Organising Software into Repositories

When a complex or distributed software project is put into a source code repository, the question that may arise is whether to use a single or several repositories (using several VCS servers or several ‘software projects’ on a single VCS). The basic recommendations are the following:

- The geographical distribution of the development team does not matter.

- The personal responsibilities of individual developers for some part of development are not a sufficient excuse to split the development into several projects.

- If a group of developers is permanently dealing with some part of code that other developers are not familiar with (nor need to be), and if this part may become a separate product or development project, this part of the code is a good candidate for factoring out into a separate VCS project.

Productising (besides marketability and saleability, which are not goals in all projects) is also related to establishing separate builds and packaging the software as a separate application, tool, service, component or library. To have its own life, this offspring must have its own user population, clear set of distinguishing and relatively complete features, and limited and well-defined external interfaces. Having separate repositories also facilitates the establishment of different policies, including access control.

If the above is not the case, the questioned component of development should be kept as an integral part of the bigger project. Having a single code repository allows a flatter and integral view in the future, preserving the opportunity for easier code sharing, major refactoring, change of purpose, or modification of personal or organisational assignments. Once a project is split, there is usually no easy way back. Once the codebase is fragmented and some parts of code become inaccessible, other developers lose the sense of control over them and even an opportunity to establish a personal attachment.

Moreover, enforcing some access control is not a sufficient reason for splitting a software project. Limiting access to some folders or making them read-only for some users can be achieved within a single Subversion project by using path-based authorisation and the `authz-db` configuration parameter [SVNAUTH].

Subversion also provides the svn:externals feature, which allows logical inserting of a part of an external SVN repository from within the project. However, this is acceptable only for non-editable access, because actual folders bound by svn:externals will probably reference some specific release or snapshot, and people who are editing a project using this feature may otherwise be tempted to treat the external code then the same way as the one belonging to their own project. There is an obvious danger here, and this is the reason why check-in by default does not traverse into folders attached by using svn:externals. However, the user can still check in individual contents within svn:externals, and thus modify something that is supposed to be immutable.

Even if svn:externals points to the trunk (not a specific revision), this makes it easier for outsiders to modify the external trunk, which may break the policies set by those primarily responsible for the external source. And, in opposite direction, using an external trunk could be a problem for developers using the code primarily managed by somebody else, as they will be affected by all changes within the trunk, which is rarely desired. So it is better to implement the mutual references primarily through the use packaged modules provided by other projects, using a Maven repository which should provide binary, source code and documentation for each particular release. The access to source code within another SVN via svn:externals should be used only when it is really necessary. For example, if a tool used for debugging does not support access to a packaged source, if this source is not included in the package being debugged, or if the source code of a rapidly evolving project needs to be made accessible to the developers of another project.

3.3 Recommended SVN Layout for Individual Projects

The following SVN layout is recommended for the file repository:

- **/trunk**

  Contains the main body of development. New projects should be started here. It is recommended that /trunk is always kept in a state in which it compiles and passes regression tests. This is done by performing only small changes within it and by quick merges of significant changes, which are first completed within the branches.

- **/branches**

  A branch is a special line of development of the source tree that starts with a specific copy of code ('svn copy'). It is used to manage work that may be later merged back into the main trunk or originating branch. Using branches helps preserve the integrity of the trunk. Each branch is created as a separate folder within **/branches**. Unless the branch is by its nature limited to some part of the source, it is preferred that it represents a full snapshot of the whole **/trunk** (or originating branch). This provides an integral view of the branch, as well as a complete set of files to work with.

  It is important to remember that multiple branches are developed and maintained simultaneously. The branch is maintained over time by porting some changes from **/trunk** to **/branches**, so each one imposes additional maintenance costs. A branch is one of the following:

  - An editable snapshot of a stable software version (which is bookmarked in **/tags**). A branch is created for a stable release. This stable version is supported in parallel with development of the new releases (new functionality that is contained in the trunk). Having support provided through branches allows bugs to be fixed and new maintenance releases to be produced without having to release the newest and possibly immature features. For example, if version 2.0 is under development, a stable branch from the 1.5 tag is created. This version is a base for the 1.5.1 release which contains bug fixes reported for 1.5. The release is prepared in **/trunk**. Before the release, the source code in the trunk is frozen and tests are performed.

    Once the tests are passed, a Maven plugin is used (or other standard procedure performed) to prepare and tag the release. A new branch is created as a tag copy, but later may be associated with new revision tags during maintenance.

  - A feature branch (e.g. /branches/NewFunkyInvoicing/branches/HopelessBug9876Extermination). A major modification, potentially destabilising change or anything that may not become mature before the next release should be separated in a branch.

    Feature branches must be properly described by using clear names that will remain associative and meaningful throughout the project lifespan. These names should describe a feature or component but not the author or timeline.

    With a large number of destabilising changes of which some are in isolation from **/trunk**, feature branches should either have a limited lifespan (for example, if used for communication or experimental purposes) or be fully synchronised with **/trunk** in regular intervals (weekly or even daily). This is done by merging the differences that occurred within the base (trunk or originating branch) into the branch. First, update and commit the local copy of the branch. Then perform ‘svn merge’ against the trunk on it. Finally, after possible corrections and successful local build and basic tests, commit the synchronised branch with a comment like "Merged latest trunk changes to <BranchName>".

- **/tags**

  Used for bookmarking or taking reference (non-editable) snapshots. Tags are used to mark specific releases or phases of development, in order to be able to retrieve and inspect a past release exactly as it was. Unlike branches, tags do not impose any maintenance costs, so they should be used liberally in order to track the release history.

  - The release tag is packaged and released for production (e.g. /tags/(Release-)1.0.1, /tags/(Release-)1.0.2) from a version maintained within corresponding /branches/(Stable-)1.0, while the main development continues in /trunk.
  - Tags should be created automatically by a Maven release plugin. This approach will provide correct versions dependencies in multi-projects.

    Currently, on the GÉANT SVN server, only the release manager (the person who performs releases) can create tags. Developers are only allowed to commit.

In SVN, the revision number is global for a whole repository. It allows consistent repository content to be checked out from a selected revision number (revision version or date). It should also be noted that the creation of branches and tags does not make a full copy of the files in SVN. Instead, this produces a lightweight reference to specific path and revision, which does not consume any significant storage within the source code repository.

For more details on the usage of tags and branches, see [SVNUSE].

3.4 SVN Access

Developers will access SVN via Secure Shell (SSH) only, using keys. Authentication and authorisation is based on the developers’ SSH key. Assigning a developer to a project’s SVN repository will be delegated to the appropriate person who is in charge of the project.

Access for the general public will be determined once the GÉANT Consortium decides on a software licensing policy.

3.4.1 SSH Keys

A number of online resources provide information on how to generate a good SSH key and use it. See [SSH] for some examples. Once you have your public SSH key, you need to mail it to the SVN administrator [SD] and ask your project coordinator to contact the SVN administrator (by creating a request in GN3 JIRA [SWDSD] or by emailing the Service Desk [SD]) to grant you appropriate access rights. An up-to-date list of coordinators is available on the Intranet [COOR].

It is good to remember that an SSH key usually identifies a user or a role rather than a machine or a user on a machine. Therefore, if you use different computers to perform your work, you should use the same key on all those computers. You should also note that it is considered a good security practice to protect your key with a pass phrase. You should also keep a safe and secure backup copy of your private key; this can be as valuable as the key to your office.

3.4.2 Per-Activity Repository

The GN3 subversion server provides access to the different repositories it holds. There is usually one repository per activity or task. The URL you need to use to get read and write access to your repository has the following format:

```
svn+ssh://svn@svn.geant.net/GEANT/ACTIVITY_NAME/REPOSITORY_NAME
```

You will be provided with the actual repository name, once it has been created by the GÉANT Development Infrastructure Service Desk. It is important to note that the SSH user that must be used to connect is ‘svn’. If your public key has been added to the SVN server, you will gain access.

3.4.3 Browsing an SVN Repository

The GÉANT Development Infrastructure provides web browser access to created repositories via Atlassian Fisheye [FISH]. The actual location of the repository in Fisheye will be provided, once it has been created by the Service Desk. The access to Fisheye is protected by your GÉANT Intranet user name and password, however, the ‘GEANT\’ prefix must not be used.

4 Building, Integrating and Releasing Software

Software needs to be built, integrated and released in a way that is consistent across the GN3 project. The need for automating software packaging, building and/or publishing process has been recognised from the earliest days of software development. In modern times when requirements and the software itself are rapidly changing, the need for automated management of this process is greatly increased. Many software teams also integrate testing in their build automation, and use automated publishing of software to test/production environments. Also, it is not uncommon to find software setups where a commit by a developer triggers the entire process of building and testing of application. Therefore, only a consistent approach can ensure that good quality software is produced every time a release is issued.

4.1 Building Software

Building software is the process in which the source code is converted into usable software that can be run on a computer. Where complex software is concerned, a proper build management strategy combined with appropriate tools is crucial. Automated building of the source code makes it possible to put a previously created strategy into practice, so human errors can be excluded. It implies the act of scripting (automating) a wide variety of tasks that software developers do in their daily activities including:

- Compiling source code into binary code.
- Packaging binary code.
- Running tests.
- Deploying and creating documentation and release notes (less frequently).

The following guidelines for the build process extend the workspace usage best practices that are given in Section 2.1, Workspace Usage Recommendations.

- Automate builds.

  Once started, the build should be performed without human intervention, by applying the build tools to the source code. Tool configurations and build as well as check/test scripts should be treated as source code. With this automation in place, the same input files should produce the same build results.

- Commit all changes of code and build the tool immediately after successful build.

  This is important if you want to be able to reproduce the build result.

- Separate intermediate objects and final build results from sources.

  This protects the source code and VCS from being polluted by built objects. It also allows build results to be more easily identified and cleared, saved, tested, or deployed.

- Use common build tools.

  This increases productivity, simplifies usage and maintenance of workspaces, testing, release, and fosters reproducibility.

- Build often.

  Frequent, end-to-end builds with 'sanity' included reveal integration problems, and produce up-to date built objects that can easily be reused.

The last recommendation is so important that it resulted in a practice called continuous integration, which further extends this list of best practices (see *Using Continuous Integration on page 15*).

To make builds less prone to problems, it is recommended for a release manager to define and maintain a build schedule as a part of the release lifecycle. However, the software builds may be scheduled even more frequently, like nightly builds, or within continuous integration, where they may be run several times per day. This process should be used by applying a build or continuous integration automation tool or, for nightly builds, using scripting and 'cron' or similar tool, which automatically enforces the whole process.

Since a common Maven [MAVEN] repository is provided as a part of the GN3 SA4 T3 Software Development Infrastructure, along with a reference project skeletons defining common configuration settings, all projects in which this is possible are suggested to use Maven as a build management tool. For projects that currently use Ant, there is option to migrate to Maven by applying Maven's AntRun plugin [ANTRUN].

Even with continuous integration, the automatic nightly builds provide an opportunity to run a more complete and resource-intensive set of tests, since much longer build and test times can be afforded. Besides helping to keep the software code coherent and checked for integration, these builds also provide the QA team with the latest executable on which to run the tests.

However, in order for nightly builds to make sense, it is crucial for the release manager to be notified of the build outcome, usually by email. Since software builds produce significant results, the information contained in these notifications can be stored in a build history database. This database could also be extended with some additional details and be used for continuous integration and release builds, thus including [REL-MGM]:

- Build number.
- Build date.
- Build version.
- Source code label or tag used.
- Overnight build (Y/N).

- QA tested (Y/N).
- QA test results (Pass/Fail).
- Location of full logs.

4.1.1 Using a GN3 Maven Repository

A Maven repository dedicated to GN3 is available for all projects to use and share their packages. This repository is built on top of a regular file system and served by a web server, so it can easily be reached by all.

Using SSH key authentication, Maven maintainers should use the Maven deploy target to send their package to the GN3 Maven repository [REPO1]. The deploy URL is configured in the pom.xml file and SSH credentials are given in the settings.xml as described in [DEPL].

The GN3 Maven Repository is available at [REPO1] for downloading libraries and resources built by another GÉANT project. It is reachable at [REPO2] for uploading libraries and resources built by your project. For snapshots you should use [REPO3] and [REPO4], respectively.

Any user who wants to access the Maven repository must send their SSH public key to the Maven sysadmin by getting contacting the registered coordinator for the project in question (an up-to-date list of coordinators is available at [COOR]). See *SSH Keys* on page 10 for some best practices.

To ensure a common Maven repository structure across GÉANT, all GÉANT projects should prefix the groupId attribute with 'net.geant.' The project name should then be concatenated to have a groupId like 'net.geant.projetname'. Posting packages outside the /repo/net/geant/ directory will not be allowed by the server.

More information on rules and best practices to use when writing your pom.xml file and deploying to the GÉANT Maven repository can be found at [POM].

4.1.2 Using an Internal Maven Repository

If needed, an internal repository can be implemented for the team/company using both plain file system and operating systems privileges, or dedicated software such as Sonatype Nexus [NEXUS] or JFrog Artifactory [ARTIFACTORY]. Both of the mentioned products are free software, and both have a substantial user community [CMP-NEX-AR].

Implementing an internal repository using dedicated software has several advantages:

- Automatic checking for existing artifacts or updates on public Maven repositories.
- The option to have separated repositories for different needs (releases, snapshots, proxies and so on), yet integrated in one single interface.
- A quick search of all artifacts in all repositories, using extensive search capabilities (by group/artifact/version, hash, keyword, class name and so on).

- A web-based user interface.
- Detailed control of user privileges using categories such as targets, privileges, users, roles or a combination of those.

4.2 Using Continuous Integration

Continuous integration is the process of building and testing code on a frequent basis. Continuous integration servers constantly monitor source code repositories, and as soon as new changes or commits are detected, they initiate a new build cycle. The modified module (and modules that are dependent on it) are compiled and tested [CIMF].

One of the most significant benefits of continuous integration is the reduction of risk. Detailed information about the integration process is provided (during regular, scheduled tests and deploys), so that bugs are found early on and can easily be removed, since they do not accumulate. Continuous integration is useful for both users and developers as frequent deploys provide developers with quick feedback and users with quick development of new functionality.

When setting up continuous integration, it is good practice to [CI]:

- Have a single source repository that is shared among developers.
- Automate the build. Build every time a change is committed, or at least every day.
- Make the build self-testing by including automated tests that can check the larger part of code.
- Commit every day. To optimise integration, developers need to communicate the changes they have made to the other developers every day.
- Ensure that every commit builds the main codeline on an integration machine (the trunk must stay in a healthy state).
- Keep the build fast (quick feedback is desirable).
- Test in a clone of the production environment (the same software, hardware, configuration, etc., or with virtualisation).
- Make it easy for anyone to get the latest executable (the more testers or users, the faster and more comprehensive feedback).
- Ensure that your changes are clear and legible, so that everyone can follow what is happening.
- Automate deployments (use scripts that allow you to deploy the application easily).

4.3 Release and Deployment Management

ITIL (Information Technology Infrastructure Library) defines release and deployment management as a proactive technical support focused on the planning, preparation and delivery of new services. This definition can be extended to all types of products, including software. This proactive technical support activity should ensure that releases are reliably planned, scheduled, prepared, built, packaged, and successfully deployed to test and live environments, in a repeatable and controlled manner.

The importance of this release cycle management increases with the size and complexity of the software project, particularly the number of supported active instances. If adequately performed, release and deployment management ensures that the software release manager knows [REL-MGM]:

- What went into a release.
- Where it went.
- Why it went there.
- How to deal with it reported bugs.

A release is a stable, uniquely identified and properly packaged version of a product intended for use in a production environment. In other words, it is a production build of the system that is made available to end users. Its earlier version that has not yet been verified and corrected through a set of tests defined for the release is called a 'release candidate'. Sometimes, besides performing tests internally, a release candidate is also published to a limited group of end users, but usually without giving any commitment of support.

In addition to putting the runtime software in its final form, release management includes the deployment or publishing process and the update of related metadata tracking a given version of a software application, for example by publishing build history database release notes. The primary purpose of a release is to make the application available to its end users.

A software release package is a single file or a logical container of several files. In ITIL there are three release types – Full Release (standalone distribution), Delta (patch, or incremental/partial release), and Packaged Release. The latest is intended for complex settings, when the release contains several release units that need to be deployed to a number of configuration items.

A release distribution package should include metadata such as the version number, category (e.g. ITIL: Major, Minor and Emergency – this is usually implied by version numbering scheme), release type, and release notes describing performed changes (new features, enhancements, bugs fixes) as well as addressed and known (open) issues. They may also include a subset of build attributes described in *Building Software* on page 12.

A comprehensive and well-organised approach to release and deployment ensures a transparent and safe introduction of new functionality. Transparency results from the regularity of releases and the repeatability of all procedures. Safety is provided by the separation of functionality delivered in following releases: it is possible to download and deploy a historical version of the software that lacks functionality, and therefore potential bugs.

4.3.1 On Releases and Versions

The term 'revision' is commonly used to name or distinguish closely related software iterations. Most of contents or features remain unchanged or compatible between these iterations.

'Version' is used for a formal name or VCS tag of software that comprises a specific set of features and bug fixes. This name is intentionally invented (not automatically generated), as the given release is made available to the users (software release), or is otherwise significant.

Version marks, or release tags, usually have two or three segments describing major and minor release number, for example ‘3.6’, ‘1.2.0rc’ (see *The Package Naming Scheme* on page 27. To achieve a one-to-one relationship with a specific revision, the third segment is often added to the version number. It can be a real (VCS) revision or ‘build’ number, or may be reset to 0 or 1 with each version, when it loses a direct relationship with a particular revision within the version control system, although this composite version mark should be used as an alias or release tag for a specific (SVN) revision. However, on version control systems that produce revision numbers on a per-file, not per-commit basis, this version mark tags a set of files, each with its own revision number. When this additional information is provided, it is recommended to use the ever-increasing build number, as this information can be automatically generated, and easily associated with a particular revision (regardless of the revision numbering by VCS) and set of performed test.

Since documents produced using word processors are rarely kept in a VCS system, which would produce revision numbers or keep track of version tags, revisions of textual documents are often called versions. The corresponding version numbers are usually included in file names, document titles and other document properties. Such documents are usually kept on a file system under different file names or on document management systems. These arrangements rarely support comparison of revisions or insertion of revision numbers, as VCS systems do.

To summarise, a Version Control Management tool automatically creates revision numbers, while humans are responsible for creating version numbers including their last (optional), revision part. In order to avoid possible ambiguities, it is recommended to use the term revision for automatically generated numbers, and version for numeric tags generated by humans, which are, after being built and packaged, released and provided to the users.

4.3.2 Release Procedure

To provide effective release and deployment management, a release manager (RM) and a deploy manager (DM) should be appointed. The RM should be provided with unlimited access to the development and testing infrastructure (e.g. database and application server credentials) and a means of contacting all involved software developers and end users (e.g. mailing lists).

A release should cover the following aspects:

- **Release Candidate** (RC):
  a. The RM decides that the implementation phase is finished. They notify the developers and remind them to update their local repositories. The RM decides to freeze the source code. Only fixes to bugs found during tests can be committed.
  b. The RM tags the source code.
  c. The RM prepares the RC, installs it on a test server and notifies testers of the beginning of the tests.

- **RC testing** – strictly related to problem management:
  a. A series of test procedures that are predefined for an RC is performed, issues reported, and fixed, (if possible).

b. The reporter of each requested task tests the solution and closes the issue if it is fixed. If a reporter is also responsible for the implementation of the solution (a developer requests that they themself should solve an issue), the lead developer should assign another person to test and close the issue.
c. If tests do not confirm that a solution has been implemented and an appropriate modification cannot be delivered immediately, the issue should be marked as to be repaired in the next release. If a fatal defect is discovered, the attempt to move to the release stage must be aborted and a new RC scheduled instead.
d. Developers (or technical writers) provide documentation updates.
e. Static analysis of the source code (e.g. FindBug reports). It is usually better to simply report non-fatal issues detected by the static analysis, even trivial ones, than to fix them. Hasty changes that are made in attempt to address some minor problems may actually destabilise the code. Additional work related to the results of the static analysis has to be taken into account when the work schedule for next release is prepared.

- **Release** (R):
  a. The RM tags the source code.
  b. The RM publishes installers and updates the project pages.
  c. The RM notifies users of the new release and publishes release notes.
  d. The RM notifies developers that source code is not frozen any more.
  e. The RM prepares the release and updates all existing demonstration instances of the system. This provides potential users with the most up-to-date version of the product.
  f. The Project Manager (PM) verifies the development plan for the next release.

A deployment should cover following aspects:

- **Test deployment**:

  The DM deploys the system in a test environment. The test environment and operational environment should be as similar as possible (software, hardware and content should be considered).

  a. If the test deployment fails, the cause of the failure should be found and support should be requested from the development team.
  b. If the test deployment succeeds, a detailed deployment plan for an operational environment should be prepared (e.g. the time and date of the operation should be agreed, so end users are affected as little as possible).

- **Deployment in an operational environment**:

  The actual release procedure can also be formalised by defining a schedule for the regular release cycle. With an agile lifecycle, this could be defined via a Gantt chart similar to the one in Table 4.1. Other release management process examples are available at [REL-M-PROC].

Activity/Day	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Release and Deployment	■					■									
Release fixes	■	■	■	▒	▒	■	▒	▒	▒	▒	▒	▒	▒	▒	▒
Revision plan update										■					
Branch merging										■	■	▒	▒		
Acceptance tests												■	■	■	■
Documentation updates and checks										■	■	■	■	■	■
Development	■	■	■	■	■	■	■	■	■	▒	▒	▒	▒	■	■
Internal QA review				■					■					■	

Table 4.1: An example of a release-cycle schedule

The release management process documentation should explain any aspects of the release cycle schedule that are not obvious. For example, in the given schedule it is assumed that the first deployment is run as a release candidate on a single or few selected not-critical but representative instances, in order to perform quick and agile beta testing. The second deployment is a verified release, rolled out to all production instances, including the one the RC was distributed to. This two-stage rollout should be used if:

- An adequate user community exists (i.e. if there are many software instances).
- There are some ‘guinea pig’ users willing to tolerate possible usability problems.
- It is unacceptable to propagate such problems to all instances (either in terms of support or reputation).

If these conditions do not apply, it is more effective to deploy a single revision (fixes excluded) within each release cycle.

To be able to react promptly, it is recommended to plan these rollouts for the first half of a working week. If a fatal defect is discovered in any of these two releases, the release from the previous cycle is restored. Otherwise, the current release updated with bug fixes is only redeployed if critical problems occur. Note that anticipated early-life support of the first few releases and releases after major changes may be labour-intensive and take longer than just a few days. The general release cycle schedule may be adapted accordingly, but an attempt should be made not to break the usual scheme.

Release fixes are made on both the main development (SVN trunk) and the release branch, while the development of new features, major refactoring, and the solving of bugs that require longer work are done within trunk or separate branches. This means that the live non-release branches in agile development are always in alpha mode.

When the revision plan is updated, it is determined which feature branches will be merged with the trunk for the release. If needed, the plan for the next and later releases is also updated. This is also a good moment to decide whether to start a new branch that deals with a particular problem or resource-consuming task. However, development tasks which are not questionable can start any time, they do not need to wait for the next plan update. Where possible, it is recommended to start a new branch and schedule the work on it for the period the originating branch is relatively stable (for example, after a release is published).

4.3.3 Releasing a Product

Once all development, testing and building activities are completed, a polished and tested product is created. This product is then released and distributed. There are different options how this process can be managed. All projects should weight these options and try to find a balance between wanting to have a broad distribution and being able to provide support for multiple package types and OS.

Once a new build is finished, tested by a test team and packaged, it is announced and distributed. Notification and distribution can be managed using different approaches depending on the project's policies. It can be announced via web sites, different advertisements or mailing lists of current and prospective users.

General availability is the point where the software has been made available to the general users either via a website or physical media. In most cases an internet site is used, from which the program itself can also automatically download the new version. This is usually done via update notifications from within the software and with the user's permission. At this stage the software product is said to be live. A live release is considered to be stable, relatively bug-free and suitable for wide distribution. It may also be signed to allow end users to verify that their copy has not been modified since the release.

The release of a new software version and its maintenance along with the previous versions lead to increased support efforts that need to be managed and planned. There are two approaches to reducing the need to support a wide range of versions:

- Users can be stimulated to adopt the new version by announcing its new features, bug fixes, or other advantages over the previous version.

- Based on the general lifecycle of a software project, the adoption of a new version can be forced by retiring the previous version and ending its support. However, retirements must be announced in good time. This is also referred to as the "end-of-life" of a particular software version.

Some good software release management practices are [SRM-BP]:

- Use English.

  If even the slightest chance exists that your code will be used and maintained in a multilingual context, all names should be in English. Use correct and consistent spelling.

- Define regular, targeted release dates.

  The releases should be planned, managed and delivered regularly. If a number of releases are being developed in parallel, create and communicate a high-level release milestone plan.

- Always have a tested back-out plan.

  For releases that are being deployed to live environments there should always be a tested back-out plan. The acceptance testing and rollout scenario should be part of the workflow defined for each release package.

- Have a documented release policy.

  The software release management processes and policies should be clearly defined and documented. This should include the definition of the types of release you will manage, their workflows, the default calendars, as well as roles, responsibilities and artefacts.

- Construct deployment units as early as possible.

  For each application build it is better to construct deployment units with the potential to be installed (e.g. a J2EE .EAR file or a Windows .MSI package). Although these deployment units might not be released beyond the development team, they allow the build, packaging and installation process to be proven early and at each phase.

- Use an independent team to construct all external releases.

  Small development teams might be able to carry out the building and construction of the deployment units. However, for any medium to large size software projects it is desirable to have a separate release team.

- All deployments should be performed by a team independent of the development team.

  Developers should not be allowed to transition deployment units into a live environment, as a potential conflict of interests may exist. To ensure auditability and traceability, it is better to have a separate team deploy the release to live (and acceptance testing).

- Test the deployment process at least once before deploying to live.

  The deployment process should be tested at least once before any release is put into a live environment. This is normally done by having an acceptance test environment that mimics the live environment and is controlled in the same way.

- Automate as much as possible by using integrated tools for configuration, change management and deployment management.

  Software release management can be a repetitive and error-prone manual process, therefore as much as possible should be automated.

- Use a software configuration management process and tool to support the development of multiple releases in parallel.

  If developing, building and deploying multiple releases in parallel, it is critical to have a software configuration management tool that supports parallel development.

- Link all release documentation and scripts to your deployment unit.

  All related hardware and software required to support a release should be identified from the deployment unit. The release package is a good container for managing these relationships. It can either relate all this together in documentation form or reference entries in a database. Such a database should identify for each product that is built and released:

  - The required hardware.
  - Supporting third-party software.
  - Installation and configuration instructions.

  In ITIL such a database is usually called CMDB (Configuration Management Database).

4.4 Software Packaging and Publishing

The following major packaging schemes can be used:

- **Source** (GZip, BZip, ZIP)
    - Compressed archive of source files, build tools (e.g. autoconf helpers, makefiles) and documentation.
    - Designed to build on UNIX operating environments.
    - Generic and universally accepted, is usually the base of other packaging systems.
    - Does not lend itself to single click installation – targeted at the intermediate user.
- **RPM** (Redhat Package Management)
    - Compressed package that includes all files (source, binaries, documentation) and specialised installation instructions (e.g. where to install, permissions to give the new files, etc.).
    - Targeted at RedHat-based OS (e.g. RHEL, CentOS, Fedora, Scientific Linux, etc.).
    - Created via a single specification file; all instructions are entered in this one location.
    - Different OS offerings have different environments; building and releasing for each target system is recommended.
    - Used in package management tools such as YUM or Up2date, which manage dependencies.
- **DEB** (Debian Package Management)
    - Similar to RPM, but for Debian based operating systems (e.g. Debian, Ubuntu, Knoppix, etc.).
    - Normally paired with the APT system for managing dependencies.
- **JAR** (JAVA Archive) and WAR (Web Application Archives)
    - Packaging similar to RPM, DEB, and DMG but specifically for Java software.
    - Integrates with other Java tools such as ANT for makefile-like behaviour.
    - WAR archives are directly deployed in Java enabled web servers (Tomcat, WebSphere, etc.).
    - Only makes sense for Java software releases.
- **CPAN** (Comprehensive Perl Archive Network)
    - Source packaged Perl source code and documentation, constructed using Perl tools.
    - Distributed through a network of server mirrors.
    - Installable via simple command line application, CPAN, that manages package dependencies.
    - Requires additional packaging investment beyond source.

- **Ports** (BSD Family)
  - A collection of software sources arranged in a hierarchy for the BSD family of operating systems. Members of the hierarchy are constructed to rely on other packages when a dependency is required. This allows an easy cascade of activity when something needs to be installed.
  - Same requirements as Source for the most part; additional work to port the software release into this environment (e.g. special makefiles, etc.) is necessary.

- **DMG** (Mac OS)
  - Similar to RPM and DEB, but for the Mac OS family of operating systems.
  - File is encoded similar to a drive (e.g. block encoded) and is mounted and installed as such.
  - Results in drag & drop or easy wizard installation.
  - Format is very specific to this OS, and in general the OS is capable of using other formats like DarwinPorts (BSD) or source packages.
  - Windows packaging schemes

4.4.1 Java Packaging

Java applications should be packaged in JAR, WAR or EAR archives. The files within these archives can be extracted by using the JDK “jar” command or an archive management tool (since it actually uses ZIP archive format).

- **Java Archive or JAR** (with .jar file extension):

  This file type is generally used to distribute Java applications or libraries in the form of classes, associated metadata and resources (text, images, etc.). JAR archives are used for standalone and client applications, but also for some modules used by Java EE applications. Java classes and resources should be stored in subdirectories of the archive’s root which are named following the usual package naming hierarchy. The JAR archive also contains a manifest file, located in the path /META-INF/MANIFEST.MF. The entries in the manifest file determine how the JAR file can be used. The JAR archive can also include a Class-Path entry with a list of absolute or relative paths to other JAR files to be loaded with the JAR. If the content of JAR files is digitally signed, the signature information becomes part of the manifest file.

- **Enterprise Java Bean** (EJB):

  A JAR module (.jar) with a META-INF directory containing an EJB deployment descriptor in /META-INF/ejb-jar.xml. If there is an EJB-based web service, this META-INF directory also contains a webservices.xml file, proprietary deployment descriptors, a JAX-RPC mapping file, and a wsdl directory for the web service’s WSDL files.

- **EJB JAR client module**:

  Contains the class files that a client program needs to access a local or remote EJB implementation.

- **Web application archive or WAR** (.war):

  A JAR file used to distribute a collection of servlets, Java classes, JavaServer Pages, XML files, tag libraries, static Web pages, and images. It can be digitally signed to confirm that the code is trusted. The mandatory file /WEB-INF/web.xml defines the structure of the web application, including mapping of servlets, initialisation parameters and other web application or component settings. The WEB-INF folder may also contain vendor-specific runtime deployment descriptors. Java classes should be stored in the /WEB-INF/classes directory. The additional code can be placed in JAR files within /WEB-INF/lib, which is a good option for packaging EJB JAR modules or other libraries that are required by the web application. A WAR module may also contain a web service, if WEB-INF contains a webservices.xml file, a JAX-RPC mapping file, classes, lib, and a wsdl subfolder with <serviceName>.wsdl files.

- **Resource adapter RAR module** (.rar):

  An extension implementing the J2EE Connector Architecture, that is, a system-level software driver that a Java application uses to connect to an enterprise information system.

- **Enterprise archive or EAR** (.ear):

  This assembles one or more modules of the above listed types into a single archive, so that the deployment of all the components of a complex application is simultaneous. An EAR is a standard JAR file with a /META-INF directory which must also contain an application.xml file that describes the application and contained modules. The same directory may also contain additional files with application-server-specific deployment descriptors.

  Dependencies among modules within an EAR are expressed by naming other modules in the MANIFEST.MF Class-Path of the depending module. Another option is to package a JAR with a common component into an independent extension JAR, and then reference it from manifest files in JAR files within application EAR files [EAR]:

```
app1.ear:
   META-INF/application.xml
   ejb1.jar:
      META-INF/MANIFEST.MF:
         Extension-List: util
         util-Extension-Name: com/example/util
         util-Extension-Specification-Version: 1.4
      META-INF/ejb-jar.xml

util.jar:
   META-INF/MANIFEST.MF:
      Extension-Name: com/example/util
      Specification-Title: example.com's util package
      Specification-Version: 1.4
      Specification-Vendor: example.com
      Implementation-Version: build96
```

With both of the above packaging, each module will be loaded using a separate classloader, which provides a better separation between them. The variant with extension JAR files is particularly handy with a library that is used by several applications or when individual archives are frequently and separately deployed. However, this approach may be a source of problems in a production setting due to absence or an invalid version of the extension archive. On the other hand, placing library and EJB JAR files into the /WEB-INF/lib of a web application WAR encapsulates the project into a monolithic package, which is more suitable for self-contained projects. Maven assumes this usage by default.

The creation of these archives should be an automated part of the build process. With Maven, packaging of archives is performed by configuring ear, ejb, jar, rar, war, and shade plugins operating during the “package" phase of Maven’s lifecycle. For example, which one of WAR packaging approaches will be used is configured via the <packagingIncludes> or <packagingExcludes> elements of the war plugin. There are also some relationships between configurations of various plugins. If some JAR files are packaged separately from a depending WAR, but within the single EAR, the WAR dependencies must be explicitly written in EAR’s pom.xml file. However, if JAR files are to be packaged within a WAR, it is sufficient to define a compile-scoped WAR’s dependency on these JAR modules.

4.4.2 Java EE Packaging

Some best practices for packaging Java EE module are as follows [EE]:

- Since most application containers allow applications to be deployed in exploded format, it is tempting to use such a deployment. However, after the development phase this should be avoided. The direct modification of exploded files must also be avoided.
- JAR files that are normally supplied by the system or application server classpath should not be included in the EAR. Besides increasing the packaging size and deployment time, this can cause class-loading issues when these archives are upgraded by the administrators.
- Project-specific JAR files should not be placed on the system or application server classpath, as this breaks the regular application maintenance procedure.
- If your project consists of several applications that share some common JAR files, these modules should be packaged separately. If this is not the case, you should package your applications in a single WAR or EAR file.
- The setting of packaging descriptors like those describing inter-module dependencies should be automated using Maven or similar tools. These descriptors should never be managed manually.
- Source files, uncompiled JSP pages, and build and test artifacts (such as build.xml or pom.xml files) should be removed from all deployment packages. This reduces packaging size, guarantees there are no compile time errors, improves the initial response time, and eliminates unnecessary exposure that may lead to security issues. However, delegation of requests for JSP pages to the compiled servlet classes file will only work with appropriate servlet mapping in the web.xml file.

- All JAR files used to run the application outside of a Java EE container (for example for testing) must be removed from deployment packages. In a production deployment their presence is likely to break the application.

- If an application uses large or independently managed static web content, it should be removed from the EAR archive. If this content is not to be to be edited, it may be distributed in a separate JAR file that will be exploded by the application server. If this is not the case, the static content may be placed on the file system outside the application server, which requires additional work on content mapping on both the web and application server. However, moderate static content that is closely related to the application should be packaged with it.

- The application configuration should be stored in the applications deployment descriptors outside the application archive. If other mutable or instance-specific configuration files are also required, they must be consistently named and located. Having them on a system or application server classpath requires consistent configuration of all servers, so is it a good idea to package these files in separate JAR archives.

- Archive names and their versioning attributes comply with the established standards and conventions (e.g. the ones defined in Section 4.4.5, The Package Naming Scheme). However, web application and service modules within EAR files should be named after their desired context root. Using this convention for web-accessible modules instead of explicitly setting their <context-root> makes it easier to access and track them later on.

4.4.3 Using Java Web Start

The best mechanism for installation and management of moderately sized desktop Java applications is Java Web Start [WS]. It allows users to download and easily activate Java applications from the web. This technology is a good solution for deploying and updating desktop applications, and also supports application signing.

Java Web Start applications are packaged into JAR archives and described in JNLP XML files. If a user clicks on a link pointing to a JNLP file, the browser launches Java Web Start, which automatically downloads, caches, and runs the application. Upon later runs Java Web Start checks for the latest application version and if needed, updates it on the fly, eliminating complicated installation or upgrade procedures. If the user does not have the required Java Runtime Environment version installed, Java Web Start will download and install it.

Applications launched with Java Web Start are, by default, run in a restricted sandbox, with limited access to local files and system properties, and without the ability to run native libraries. Network communication is allowed only with the host from which the application is downloaded. Some of these restrictions can be overcome by the use of JNLP API which provides access to the system printer, clipboard, and even local files via user-controlled dialogs. There is also cookie-like persistence for storing application state and ability to create desktop and start menu shortcuts and associate the application with the specified mime-type and file extensions. The first time the user tries to use one of these features, they are asked to grant the appropriate permission.

If the application needs unrestricted access to local files and the network, it needs to be delivered in a set of digitally signed JAR files [JAVASIG]. The first time a user launches such an application, Java Web Start opens a security dialog to prompt the user to accept the digital certificate. Subsequent invocations will not show this dialog.

Developers should be aware that the usage of Java Web Start requires a proper MIME type setting for JNLP files on the Web server. Also, there are several significant bugs in earlier version of Java Web Start [JWSbugs]. These are related to the updating of JNLP components, the uninstalling of JNLP extensions, the starting of cached applications while the originating server is offline, the relative codebase paths in JNLP files, and desktop shortcuts. All but the last bug were fixed in Java 6 SE update 18 [JBUGS]. The JNLP update from a desktop shortcut bug was fixed in Java SE7 update 2 [JWSdesktop].

4.4.4 RPM, DEB, Ports Packaging

If distributed software is to be installed on an OS using a proper packaging system such as RPM, Deb, Ports and others, the chosen packaging system should be used as much as possible. This makes the system administrator's job easier, and generally contributes to making the software more usable.

The following are useful references to help you build your packages:

- **RPM**
  - RPM packaging best practices [RPM1].
  - Reference documentation [RPM2].
- **DEB**
  - How to build a DEB package in 10 minutes [DEB1].
  - Reference documentation [DEB2].
- **Ports**
  - Quick how to [Ports1].
  - Reference documentation [Ports2].

4.4.5 The Package Naming Scheme

For project visibility, to strengthen and contribute to the GÉANT image, and to give the user a good experience, all GÉANT released software should use a common naming scheme. The criteria for defining a GÉANT package name are:

- The name should be uniquely identifiable.
- The name should include the product (software) version number. No other version number should be present except for meaningful dependencies (java6, perl5.8, etc.).

- GEANT2, GEANT3, GEANT, GÉANT, GN or any other conceivable variation will not be used in the package name.
- The name should respect specific OS distributions naming policies.

4.4.5.1 *Name Format*

Every name should contain 5 elements, separated by a dash (-) and an underscore (_):

For Source:	$name = $project-$softname_$version-$revision.tar.gz
For RPM:	$name = $project-$softname-$version-$revision.$arch.rpm
For DEB:	$name = $project-$softname_$version-$revision_$arch.deb
For JAR or WAR:	$name = $project-$softname_$version-$revision.(jar|war)

The first two parts ($project and $softname) can be made up of different words separated by "-" (dash). Due to the issues with packaging tools from Debian and its derivatives, it is not acceptable to separate name parts with an underscore ("_"). To satisfy the various naming conventions for Linux distributions, only lower case should be used for all name parts.

The name elements should be constructed as follows:

$project

The name of the task or project in which the product is being developed (this must be unique across all GN3 projects). For example, 'perfsonar', 'cnis', 'autobahn'.

$softname

A product/software-specific name that identifies the following:

- Programming language used: 'java', 'perl', other or nothing at all.
- Product: when a project produces different products (piece of software), a list of all products inside the project (GÉANT activity or task) has to be agreed within the project.

For example, 'bwctl-mp'.

$version

The version of each product has to be managed by the team who is developing and releasing the product/software. This may also be described with a composite code describing major and minor release, for example '2.1'.

$revision

The number of the package for this specific version. Care should be taken with RC to avoid upgrade issues when the final release is out. RC numbers should be prefixed with “0rc”, so that, when the final release is ready, a “1” can be used as revision number, enabling a seamless upgrade.

$arch

The architecture for which the package was built. This is “noarch” (for RPM) or “all” (for DEB) if the package is not specific to any architecture. ; refer to the specific list of architectures valid for the distribution you are building your package for. This one is not used for JAR and WAR as those packaging schemes are architecture independent.

For specific naming convention for DEB and RPM, see [DEB] and [RPM].

4.4.6 Producing Release Notes

The recommended way of automatically producing release notes (build – test – package – publish) is using custom scripts using a scripting tool (for example Ant or Linux shell script language).

Specialised software for automating the building and publishing process can also be used (e.g. Apache’s Maven), but this may be problematic.

- VCS systems provide tools to extract information from source control management software, but it is questionable how useful the created change logs are for end users. Information on which source file was changed and how is irrelevant to end users. Comments entered in VCS can be more meaningful, provided the developers are disciplined about entering meaningful, useful, user-oriented, and accurate comments for each code change they perform. A possible approach could be to especially tag all user-oriented parts of commit comments that should be passed to the release notes or user documentation.

- While extracting issue summaries from ticketing/bug tracking system could be useful, there is no readily available tool that could perform this task.

Therefore, the release manager should either manually query the bug tracking system or build and use a custom script.

The release manager should combine the following in the release notes:

- Details from the latest release plan.
- VCS commit comments.
- Details of tickets solved since the previous release.

However, the final tone and level of detail depends on the target user community (i.e. whether the end users can and want to access the bug tracker and source code repository, and if they are encouraged to contribute to the source code or debugging process).

It is not possible to automatically generate descriptions for release notifications to end users (at least not without manual editing). However, using scripts, manually edited text can automatically be inserted into software documentation provided with software packaging, publicly available revision histories, and release notifications.

4.4.7 Transformation of Local Data

With a well-organised release and deployment management process in place, it is possible to identify the persistent system data (e.g. relational database software configurations) that needs to be updated during the deployment of a new release. Such changes typically include the modification of database schemes or the modification of configuration file content or format.

Although the identification of necessary changes may rely on the release plan and tracking of commit comments and change tickets, it is good practice to define a comprehensive scheme for the placement and naming of scripts that are responsible for the transformation of data between software revisions and branches. These could also include scripts for reverting changes or restoring original data, although a backup might be sufficient for this. The scripts produced in this process can then be easily collected and included in distribution packages for execution on deployed instances. It is good practice to place these scripts in a dedicated project folder and to name them according to the version code of the target release:

- **Version code** – Appropriate release tag, for example, 3.2.1.
- **Script execution order** – Order within one version, for example, 10, 20, 30...; preserve some numbers for possible insertion.
- **Script name** – Short descriptive text.

The scripts should be automatically executed during the upgrade installation process. To execute only the scripts appropriate for the transition from the old to the new software version, appropriate logic can be included in the installation program or script. The database update utility should have access to version metadata of a previous version or database table, thus preventing a script from being run twice on the same database. Such a helper table (e.g. run_script) should contain columns that describe the target release code, script name and start date/time. A corresponding row should be added after the execution of each script immediately before commit.

There are situations where the system downtime should be minimised. Some database modifications, like index creation or deletion, the addition of nullable columns or table partitioning can sometimes be performed while the application is running. If database availability is of utmost importance, the update scripts should be grouped into three categories:

- **pre** – Run while the old version of the software is still used.
- **main** – Require the application to be stopped and all application clients to be disconnected.
- **post** – Run after the update and restart of the application, which now executes the new code.

If this complex scheme is used, the script names should also include the group code, which should be used to filter scripts before executing them in the appropriate order. The update utility should then also verify that all suitable scripts from the previous group of each release have been started before executing the current group. This is important since each group is executed in a separate run of the utility (a potential source of problems). Also, this scheme runs only well if the system is regularly updated with each new release. If this is not the case, either all intermediary releases should be installed, causing several downtimes, or pre and post scripts of intermediate versions should be also run during a single downtime.

The update scripts should first be tested on a developer‘s local database (in this case it is better not to update the helper table). In this scenario, it is also convenient to maintain a special rollback script for undoing the changes a current update performed, as these may be altered before they are accepted. This also allows changes to be integrated that other developers have made in parallel. Once all changes committed by other developers are in place in the developer’s local project and database, the new update script should be added to the script folder in the trunk together with the code commit. The same applies to the branches. They may have their own set of update scripts for the alteration of the database layout. These should be kept compatible with updates gathered during periodic branch synchronisations, and only be put into the main script folder when the branch is merged back into the trunk.

With all described procedures in place, the new release package should include all scripts from the trunk that whose version code is more recent than the code of the oldest supported release.

As part of the development process, it is good practice to combine update scripts with database backups. Producing a usable development database can be frustrating, so it is a good idea to save snapshots of populated databases for all live branches and supported releases. However, the number of database backups can significantly increase if the users’ initial database content is also kept (or both starter and populated databases for various types of customers).

With proper naming of database dumps, this approach will help developers in switching to particular release or branch. To keep the number of snapshots under control, the shared backups should be kept separate from temporary images that may be used during the testing or debugging of some complex features. The naming scheme should clearly define whether a particular backup is a starter or fully populated database, which type of setting it represents, and to which release or branch it is related.

4.4.8 Including Documentation in Software Distributions

When a distribution package is released, it is good practice to include the following:

- **Licence**

  The license should provide information about the terms and conditions of use for the distributed software (see the *GN3 Software Architecture Strategy* guide for details on licensing [STRAT]).

- **Release notes**

  The release notes that have been produced for the release (see the *Documentation Best Practice* guide [DOCBP]).

- **Support details**

  Information about how users of the software distribution can access support (for example, an email address, a URL to a forum, etc.).

- **Appropriate guides**

  Guides that help using the software distribution (for example, installation, administration or configuration guides). These can be provided as documents, online help systems, web pages, or in other formats, as appropriate (see *Documentation Best Practice* [DOCBP]). Inclusion of end-user and developer guides (e.g. on software's public APIs) is optional, and may depend on their size, format, integration with the software and whether their distribution along with the software would be beneficial to users and supporters.

- **Project information**

  Information about the project team (for example, information about the organisations and developers involved in the project).

4.4.9 Update Notifications and Automatic Updates

It is good practice to include functionality for automatic update notifications in software. This is usually done by communicating over the Internet with the servers that host the software for which an update is available. While it can be the user's decision to accept an update, there are also software products which automatically update themselves. From a software developer's point of view the automatic update notification is primarily appropriate for well tested desktop or client software with regular, but not too frequent updates. There are many benefits but there are also risks in this process. In addition to the quicker delivery of general benefits provided by the update process, auto-updates help to:

- Streamline the codebase in use which facilitates earlier retirement of old versions of the software.
- Reduce the need to maintain an end-user database.
- Provide better insight into usage demographics, including frequency of use, host platforms, and geographical distribution of users.

The risks of automatically updating software include the following:

- Fast propagation of the latest version may also spread undesired features or serious bugs. Auto-update thus emphasises testing and quality control aspects of release process.
- A buggy update program can introduce unintended consequences.
- Introduction of malicious programs through the auto-update mechanism.
- Possible security holes within the proprietary auto-update solutions. A third-party could intercept the requests for updates and send a malicious program instead.
- Unexpected problems when interacting with OS dependant software management utilities.
- Privacy concerns of the auto update concepts.

The management of the automatic updates and upgrades may need to deal with several versions simultaneously installed on the target host. It should also be considered whether the update process should be able to give a choice to leave the old version. There may be different reasons for this, including incompatibilities between the software versions, user comparison of the versions, etc.

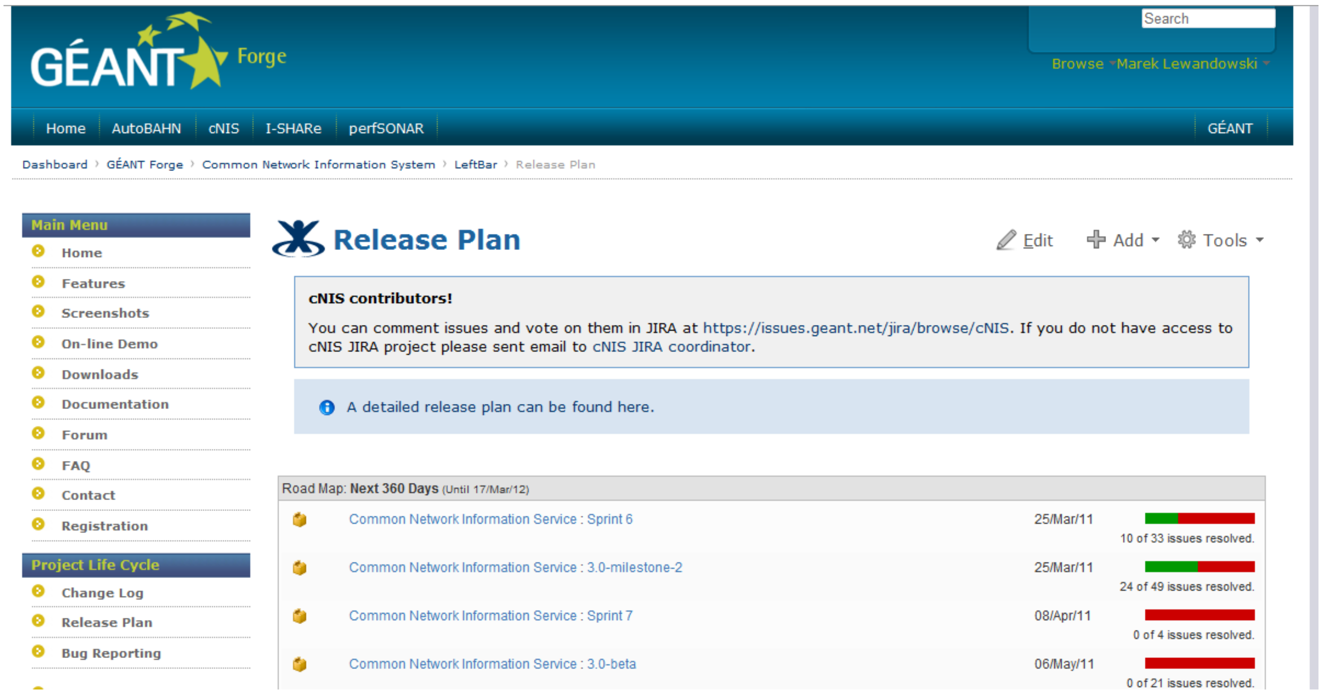
Including authentication and legitimating procedures for updates is considered a good practice that ensures the security of automatic updates. Using an authenticated connection to the update server is imperative. Signing updates with a digital signature does not prevent hostile code from being downloaded, but it does prevent the code from being run.

4.5 Using the GN3 Software Development Infrastructure

This chapter provides an example of how to apply the infrastructure provided by SA4 Task 3.

4.5.1 GN3 Confluence Wiki

The Confluence Wiki is available online [Forge]. Developers find this wiki-style tool powerful and easy to use because of its strong configurability, numerous available extensions and high integration with other Atlassian products that are used in the GN3 development environment. The GN3 Confluence Wiki is the recommended solution for distributed collaboration as a comprehensive template for project pages and a user documentation repository.



cNIS contributors!

You can comment issues and vote on them in JIRA at https://issues.geant.net/jira/browse/cNIS. If you do not have access to cNIS JIRA project please sent email to cNIS JIRA coordinator.

A detailed release plan can be found here.

Road Map: **Next 360 Days** (Until 17/Mar/12)

Common Network Information Service : Sprint 6	25/Mar/11	10 of 33 issues resolved.
Common Network Information Service : 3.0-milestone-2	25/Mar/11	24 of 49 issues resolved.
Common Network Information Service : Sprint 7	08/Apr/11	0 of 4 issues resolved.
Common Network Information Service : 3.0-beta	06/May/11	0 of 21 issues resolved.

Figure 4.1: GN3 Confluence Wiki example

4.5.2 GN3 Crowd

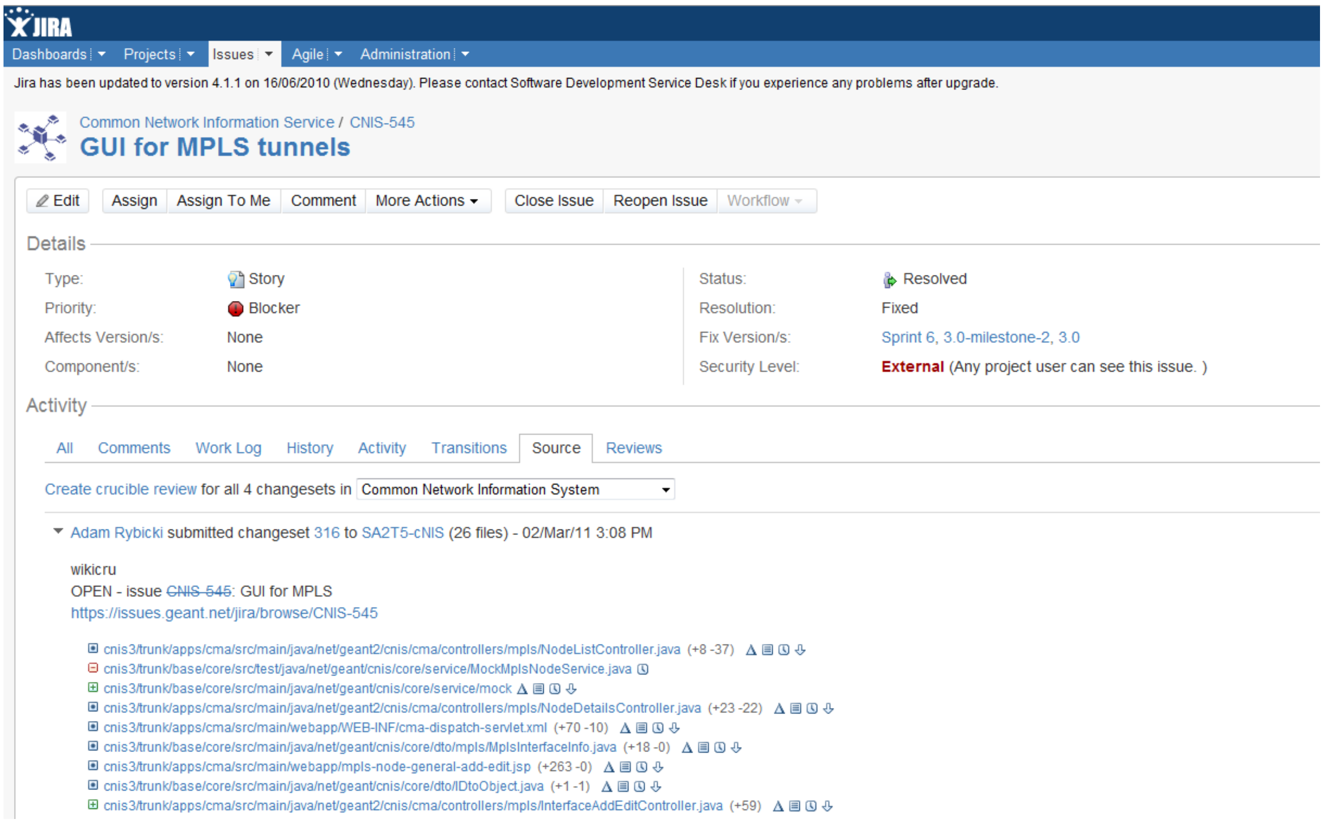
The GN3 Crowd instance is available online [Crowd]. The primary aim of this tool is to provide common identities for other tools that constitute the SA4 T3 infrastructure. Crowd can also be used as an authentication and authorisation mechanism for developed services. The following lines, for example, present a part of the I-SHARe Spring framework configuration file:

```
<bean id="authenticationManager"
class="org.springframework.security.providers.ProviderManager">
<property name="providers">
<list>
<ref bean="crowdAuthenticationProvider" />
</list>
</property>
</bean>
```

The `CrowdAuthenticationProvider` class is provided by AAI, which is a library developed by SA2 T5. More information on AAI library can be found in the documentation [AAI].

4.5.3 Fisheye with Crucible

GN3 Fisheye and Crucible are integrated with GN3 JIRA. Anyone who works on a particular issue may verify which project's files were affected by related commits. Fisheye also allows you to list all commits and JIRA issues related to a particular file.



Jira has been updated to version 4.1.1 on 16/06/2010 (Wednesday). Please contact Software Development Service Desk if you experience any problems after upgrade.

Common Network Information Service / CNIS-545

GUI for MPLS tunnels

Details

Type:	Story	Status:	Resolved
Priority:	Blocker	Resolution:	Fixed
Affects Version/s:	None	Fix Version/s:	Sprint 6, 3.0-milestone-2, 3.0
Component/s:	None	Security Level:	**External** (Any project user can see this issue.)

Activity

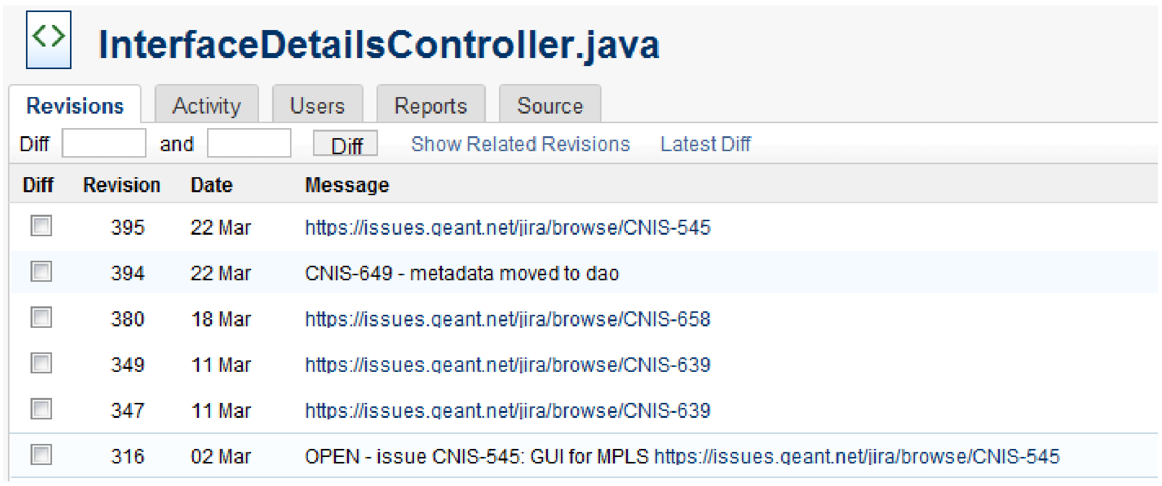
Create crucible review for all 4 changesets in Common Network Information System

Adam Rybicki submitted changeset 316 to SA2T5-cNIS (26 files) - 02/Mar/11 3:08 PM

wikicru
OPEN - issue ~~CNIS-545~~: GUI for MPLS
https://issues.geant.net/jira/browse/CNIS-545

```
cnis3/trunk/apps/cma/src/main/java/net/geant2/cnis/cma/controllers/mpls/NodeListController.java (+8 -37)
cnis3/trunk/base/core/src/test/java/net/geant/cnis/core/service/MockMplsNodeService.java
cnis3/trunk/base/core/src/main/java/net/geant/cnis/core/service/mock
cnis3/trunk/apps/cma/src/main/java/net/geant2/cnis/cma/controllers/mpls/NodeDetailsController.java (+23 -22)
cnis3/trunk/apps/cma/src/main/webapp/WEB-INF/cma-dispatch-servlet.xml (+70 -10)
cnis3/trunk/base/core/src/main/java/net/geant/cnis/core/dto/mpls/MplsInterfaceInfo.java (+18 -0)
cnis3/trunk/apps/cma/src/main/webapp/mpls-node-general-add-edit.jsp (+263 -0)
cnis3/trunk/base/core/src/main/java/net/geant/cnis/core/dto/IDtoObject.java (+1 -1)
cnis3/trunk/apps/cma/src/main/java/net/geant2/cnis/cma/controllers/mpls/InterfaceAddEditController.java (+59)
```

Figure 4.2: List of files affected by commits related to displayed issue


InterfaceDetailsController.java

Revisions | Activity | Users | Reports | Source

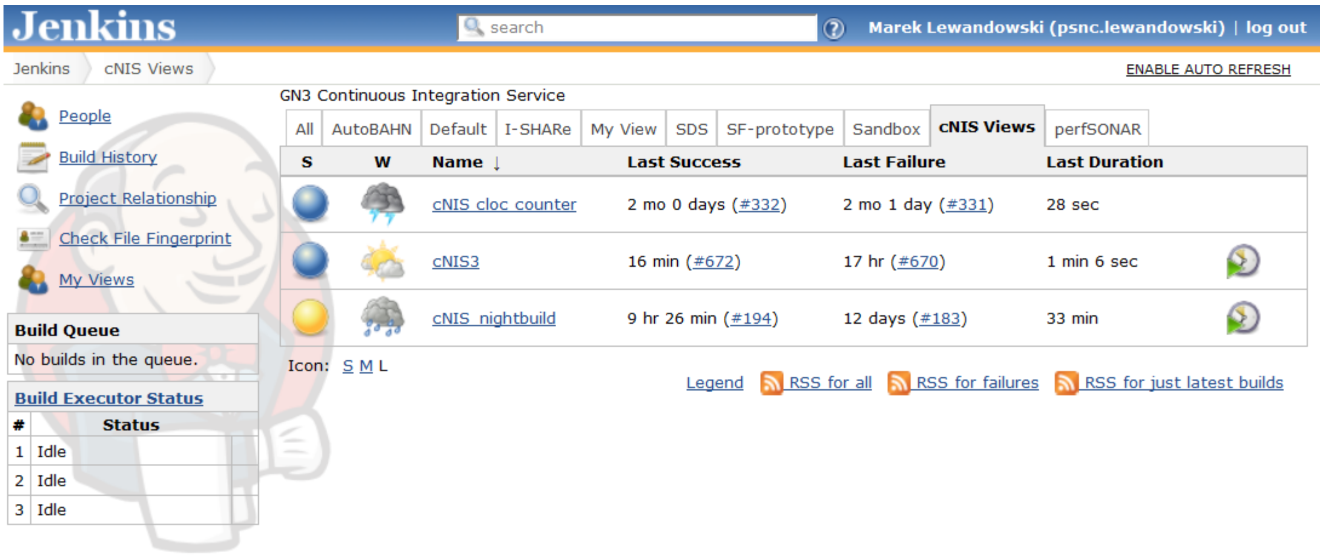
Diff ____ and ____ Diff | Show Related Revisions | Latest Diff

Diff	Revision	Date	Message
☐	395	22 Mar	https://issues.geant.net/jira/browse/CNIS-545
☐	394	22 Mar	CNIS-649 - metadata moved to dao
☐	380	18 Mar	https://issues.geant.net/jira/browse/CNIS-658
☐	349	11 Mar	https://issues.geant.net/jira/browse/CNIS-639
☐	347	11 Mar	https://issues.geant.net/jira/browse/CNIS-639
☐	316	02 Mar	OPEN - issue CNIS-545: GUI for MPLS https://issues.geant.net/jira/browse/CNIS-545

Figure 4.3: Commits related to InterfaceDetailsController.java file

4.5.4 GN3 Jenkins

This tool is useful not only for developers, but also for project managers, since it provides aggregated information on current project status and progress of work. Jenkins allows verification of the quality of the development, since it executes test cases and publishes statistics of failures and successes. The Jenkins Continuous Integration server provides results of static analysis of the source code. All time-consuming operations are executed in a background, providing Hudson users with list of warnings and vulnerabilities.



Build Queue
No builds in the queue.

Build Executor Status

#	Status
1	Idle
2	Idle
3	Idle

GN3 Continuous Integration Service

All | AutoBAHN | Default | I-SHARe | My View | SDS | SF-prototype | Sandbox | cNIS Views | perfSONAR

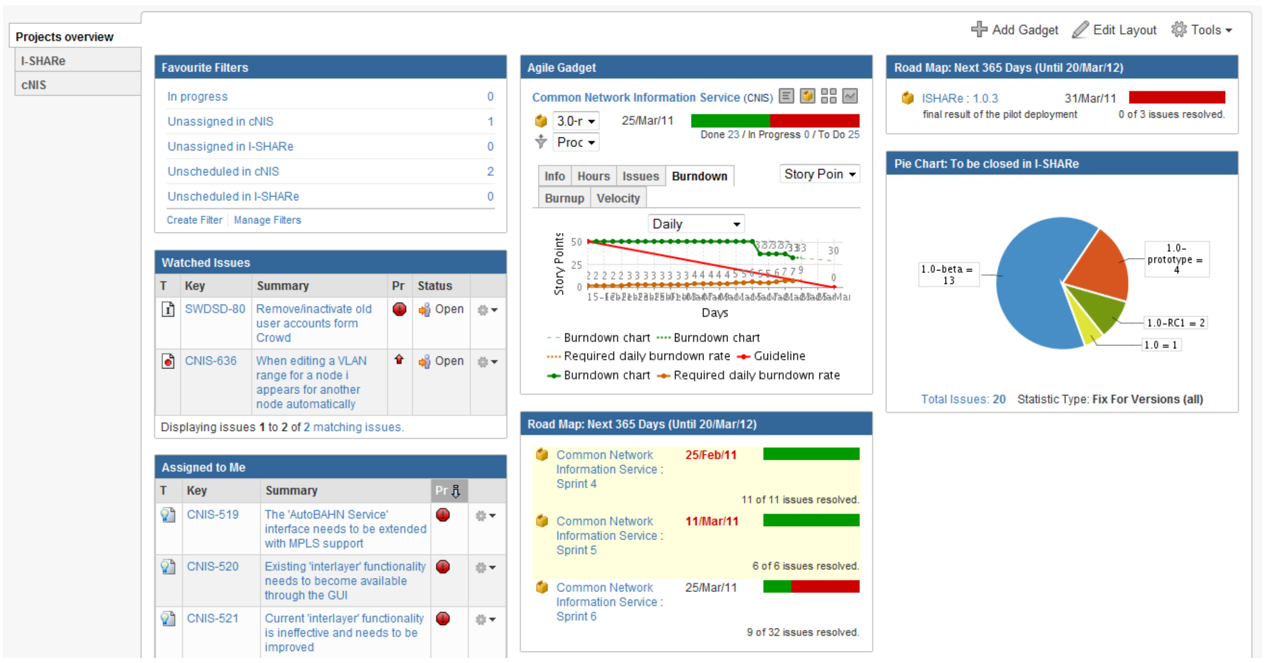
S	W	Name ↓	Last Success	Last Failure	Last Duration
●		cNIS cloc counter	2 mo 0 days (#332)	2 mo 1 day (#331)	28 sec
●		cNIS3	16 min (#672)	17 hr (#670)	1 min 6 sec
●		cNIS_nightbuild	9 hr 26 min (#194)	12 days (#183)	33 min

Figure 4.4: Main view of Jenkins interface – list of projects and status

4.5.5 GN3 JIRA

JIRA software is used to coordinate day-to-day implementation work. This is one of the most important tools used by developers and project managers. While the first group provides information about progress on the development of requested issues, the latter creates and maintains road maps and development plans. Since JIRA is a collaboration platform, results of all activities are immediately visible to other users.

JIRA, if extended by GreenHopper [GRH], significantly improves development with SCRUM methodology. This tool is highly configurable (with dashboards and gadgets), easily to integrate with other Atlassian products (e.g. Fisheye) and Subversion.



Favourite Filters

In progress	0
Unassigned in cNIS	1
Unassigned in I-SHARe	0
Unscheduled in cNIS	2
Unscheduled in I-SHARe	0

Create Filter | Manage Filters

Watched Issues

T	Key	Summary	Pr	Status	
	SWDSD-80	Remove/inactivate old user accounts form Crowd	●	Open	
	CNIS-636	When editing a VLAN range for a node i appears for another node automatically		Open	

Displaying issues 1 to 2 of 2 matching issues.

Assigned to Me

T	Key	Summary	Pr	
	CNIS-519	The 'AutoBAHN Service' interface needs to be extended with MPLS support	●	
	CNIS-520	Existing 'interlayer' functionality needs to become available through the GUI	●	
	CNIS-521	Current 'interlayer' functionality is ineffective and needs to be improved	●	

Agile Gadget

Common Network Information Service (CNIS)

3.0-r 25/Mar/11 Done 23 / In Progress 0 / To Do 25

Proc

Road Map: Next 365 Days (Until 20/Mar/12)

Common Network Information Service : Sprint 4 — 25/Feb/11 — 11 of 11 issues resolved.

Common Network Information Service : Sprint 5 — 11/Mar/11 — 6 of 6 issues resolved.

Common Network Information Service : Sprint 6 — 25/Mar/11 — 9 of 32 issues resolved.

Road Map: Next 365 Days (Until 20/Mar/12)

ISHARe : 1.0.3 — 31/Mar/11
final result of the pilot deployment — 0 of 3 issues resolved.

Pie Chart: To be closed in I-SHARe

Total Issues: 20 Statistic Type: Fix For Versions (all)

Figure 4.5: GN3 JIRA – cNIS lead developer personalized dashboard

4.5.6 GN3 Maven Repository

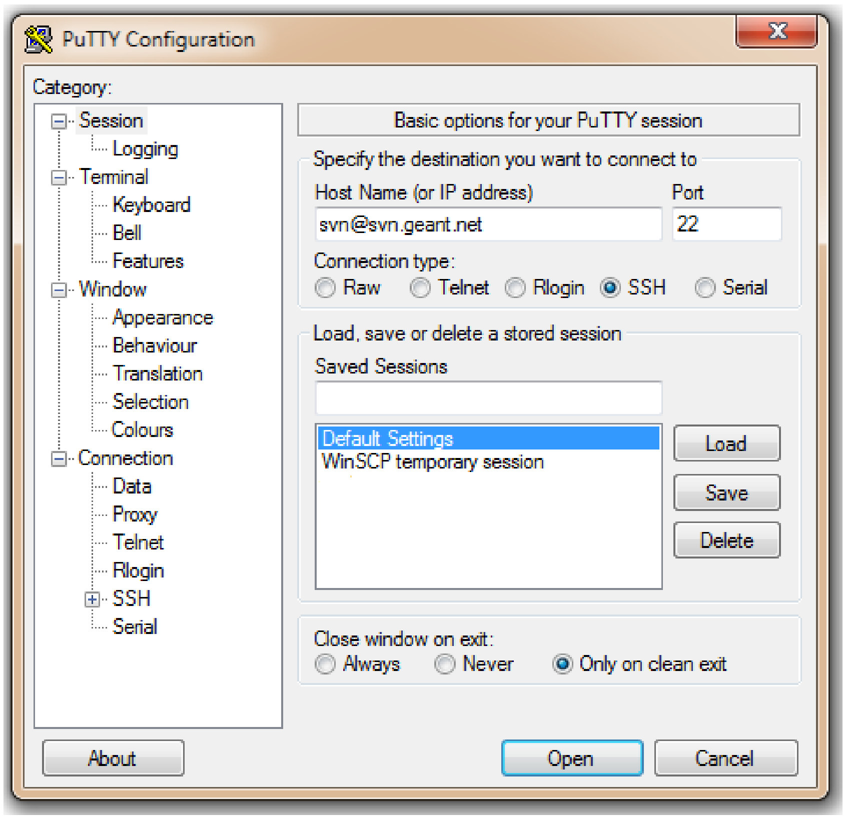
This tool is used to publish binaries among the GN3 community. “Binaries” term means particular software modules (libraries) and final applications.

All additional files (e.g. graphics, readme files) that are required by distributed binaries and potential users of these may also be propagated with the GN3 Maven repository.

4.5.7 GN3 Subversion

As both cNIS and I-SHARe development teams commit source code changes several times a day, using a version control tool is necessary.

MS Windows users may simplify their work with Subversion using graphical client – TortoiseSVN. Although the usage of this tool is intuitive, its initial configuration is a little tricky. To enable SSH support (which in fact is obligatory when using GN3 Subversion), TortoiseSVN needs an appropriate environment configuration to be able to use the Putty tool to create an SSH communication tunnel. TortoiseSVN requires Putty connections to be defined in a certain order: entry related to GN3 Subversion must be a default Putty's connection.



Specify the destination you want to connect to

Host Name (or IP address): svn@svn.geant.net

Port: 22

Connection type: Raw Telnet Rlogin [✓ SSH] Serial

Saved Sessions: Default Settings, WinSCP temporary session

Close window on exit: Always Never [✓ Only on clean exit]

Figure 4.6: Default connection (GN3 Subversion) defined in Putty

5 Coding Guidelines

The three key priorities in the coding process are:

- Keep your code simple.

  Good code can easily be understood, maintained and reused by others.

- Keep the implementation simple.

  Smart tricks, hacks or optimisations should always be avoided, unless there are specific functional or interaction related requirements for them.

- Code performance needs to be acceptable.

The code conventions and recommendations in this chapter aim to define some common rules to ensure that code is easy to read and understand, regardless of its original author. This helps code authors to formalise code concepts and visualise code structure (and therefore introduce fewer bugs). It also makes it easier to review and maintain code.

Conventions may be ignored where this enhances readability, and where stable legacy code exists that does not comply with some convention. Where small modifications need to be made to existing code, it is better to try to be consistent with the surrounding code.

The following sections describe some coding-related recommendations. They provide an overview on naming, commenting, formatting and other code conventions to be used, as well as recommendations for avoiding typical code errors. Adhering to these recommendations generally results in better code quality and improved readability and maintainability.

For recommendations on the reuse of existing code, refer to the GN3 Software Architecture Strategy Guide [STRAT]. For further information on code conventions and standards, see [CODECONV].

5.1 Naming

Using proper identifiers and names for entities (for example, files, software packages, classes, methods, variables, etc.) is essential in coding. Proper naming increases readability, minimises the need for comments and reduces omissions (for example, if identifiers end with unit names for contained quantities).

The following recommendations ensure consistent naming:

- Use English.

  If even the slightest chance exists that your code will be used and maintained in a multilingual context, all names should be in English. Use correct and consistent spelling.

- Use meaningful identifiers.

  Use terminology that applies to the domain. For example, if your users refer to their clients as customers, use the term 'customer', not 'client'.

  The more public and more globally accessible entities are, the more precise their names need to be. The same applies to obscure entities or entities that have already caused a bug by somebody misinterpreting them.

  A single word may not be as meaningful, or specific, as multiple words. Generally speaking, longer and more descriptive identifiers are better than short ones. While it is better to keep variable names below 15 characters, you should not hesitate to create names that are 20-30 characters long, if this aids clarity.

  Shorter identifiers are easier to type, but with auto-completion and refactoring features of all modern IDEs, this is not an excuse. Identifiers should be consistent, well thought-out, and descriptive. Avoid names like ID1, number, status, flag and similar, as well as names that are too generic or used to describe the type of an object or variable instead of its real purpose or role. The only exception to this are iterators (which have names like i, j, k, etc.), and heavily used local variables of very limited scope (for example, variables used in a short block or arguments of a small method). Similarly, c is often used for characters.

  Where long names are used, they need to be composed intelligently, to avoid visual clutter, redundancy and uninformative repetition. Avoid using names that are very similar. Even with shorter names, be cautious with the letters 'O', 'I' (capital), 'l' (lowercase 'L') and digits '0' and '1'. While the compiler will warn you if you use an uninitialised identifier, it will happily accept another compatible variable that is (due to a typo) used by mistake. Use plurals sparsely and consistently (e.g. for collections).

  Identifiers should not contain extremely wide or mutable contextual details that have a significant chance of becoming obsolete. For example, if a project or product name is likely to change, it should not be hardcoded in the software, unless it specifically refers to a specific product or implementation. Also, identifiers should not be overloaded with collateral semantics that are outside of the developers' focus. However, the product or project name may be used where this is necessary to specify some otherwise excessively generic name. For example, 'geanttools' is better than 'tools' to describe utilities that are shared by multiple systems in GÉANT projects.

- Comply with the rules of the programming language or runtime environment.

  Capitalise and separate words in names according to the rules that have been defined for the programming language or runtime environment you are using. This also applies to the names of source code files, packages and distribution packages. Where possible, use a consistent set of naming conventions across all languages used. Some core rules that have been popularised by Java, but are now used in many other languages, are:

  ○ Class, interface and enumeration names – use mixed case with the first letter in upper case and the first letter of each internal word capitalised (e.g. UserTracker).

  ○ Variable, class member and method names – use mixed case with the first letter in lower case and the first letter of each internal word capitalised (e.g. currentOrigin, simplifyGeometry()).

  ○ Constants and enumeration values – use capital letters only and separate words with underscores (ORIGIN_X)

- Avoid abbreviations and acronyms.

  Abbreviation and acronyms often reflect personal preferences or short-term context and are often difficult to decipher for others (Triangulation is better than Tri). The only exceptions are established abbreviations and acronyms that are well known within the scope of the domain.

  Be consistent about using particular abbreviations, and restrict yourself to using only a small number of them. Producing abbreviations by removing vowels makes reading difficult. In a bad abbreviation like MseEvtHdlr (instead of MouseEventHandler), savings of only a few letters eliminates a great deal of readability. However, using abbreviations may be justified where a name is extremely long, as names that are too long can be as hard to read as bad abbreviations.

  Generally, standard acronyms are more readable in small letters or if only the first letter is capitalised (SqlConnection, HttpRequest, mimeType). However, acronyms like HTML, URL, SOAP, XML and similar are most frequently written in capitals, except when starting variable names (htmlHeader).

- Avoid repetition.

  If some detail is already implied by its immediate context, do not repeat it. There is no need to describe the involved participants or constraints in a method name if they are obvious from the name of the class or parameter types. For example, Thread.start() is better than Thread.startThread().

  Setter and getter methods are an exception to this. You should use setTopic(Topic topic) instead of set(Topic topic), however, open(Topic topic) is acceptable. Similarly, you should not explicate the type of a variable in its name, since this is specified in its definition, checked by the compiler, and easily accessible in all IDEs.

  Types used to implement some variables often change. This applies in particular in object-oriented programming, but is also true where primitive types are concerned (it is, for example, frequently necessary to promote a short integer into a long). Another good example is the naming of methods. The type of the return object should be avoided in the method name. getHandler() is better than getEventHandler(). This is particularly true where several classes implement the same interface, returning different result implementations from the method.

- Describe the variable's purpose rather than its type.

  Notations which require the encoding of types or scopes to be included in identifiers (like the once popular Hungarian notation) are obsolete in an environment of modern languages and development tools. Variable names should describe its purpose rather than how it is represented.

  Where a single instance of a class is used in some context, the same name as its type (but with the first letter lowercase) may be used, for example, setTopic(Topic topic). This makes it easy to deduce the type given a variable name and reduce complexity by reducing the number of terms and names used. Where multiple variables have the same or a similar type, their roles should be described in their names. The variables can often be named by combining role and type, if the role itself is not descriptive enough. For example Point startPoint, endPoint.

  Similarly, ending variable and function names with names of measurement units may be useful in avoiding mistakes in arithmetic expressions when various units are used in the code.

- Use nouns for class, interface and enumeration names.

  Name classes based on their inheritance pattern, with the base class to the right of the name, and drop unnecessary parts. For example, EventHandler is the base for UIEventHandler which is inherited by MouseEventHandler (which is better than MouseUIEventHandler).

- Use verbs or verb phrases for method names.

  Name functions after what they return and procedures (void methods) after what they do. Define and consistently use a clear viewpoint in naming (regard verbs and names as dependent on participant perspective – names containing 'send', 'receiver', 'recipient', 'upload', 'put'). Use a single viewpoint for the whole model (one internal to the system, not an external actor or service), and use an object/service-bound viewpoint for object/service methods and related interfaces.

- Use getter/setter conventions.

  It is good practice to use get (is for Boolean) and set prefixes for getter and setter methods (this is a Java practice).

- Avoid negated Boolean variable names.

  You should not use negated Boolean variable names (for example, isNotSet).

- Use consistent verb pairs.

  For example, use add/remove for set management without destruction of objects, create/destroy for instantiation and deletion, insert/delete for operations on ordered containers, and so on (get/set, start/stop, increment/decrement, begin/end, open/close, show/hide, suspend/resume, first/last, next/previous, min/max, etc.).

If the development framework that is used imposes some conventions that are not consistent with the general naming rules (for example, generating identifiers of automatically produced classes by adding a suffix to existing class names, like _listener), this should be accepted without an attempt to refactor all such names.

If the naming convention for some reason seems to produce an odd result, it is a strong indication that the name is badly chosen.

5.2 Comments

Comments should be used to provide an overview of the code and additional information that is not readily available in the code itself. They should contain only information that is relevant to reading and understanding the program. For example, information about how the corresponding package is built or in what directory it resides should not be included as a comment. You do not need to explain every line of your program. If your program flows logically, a comment is probably not needed. If the line of code you are commenting is indented then make sure your comment matches the indentation.

Avoid any comments that are likely to become outdated as the code evolves. If a comment no longer applies, you should modify or remove it. Avoid replicating information that is obvious from the code. Redundant comments easily get out of date. The frequency of comments sometimes reflects poor quality of code. Rather than commenting everything, consider if you could rewrite the code to make it clearer.

In the context of GN3, you must comment in English. Use as few comments as possible where logic can be expressed programmatically, but make sure to include comments for Javadocs and functional descriptions of a higher level.

If the functionality of a block or method is complex (a long block or sequence of procedural instructions), use detailed comments to explain it. You should also add detailed comments before any segment of code that should be improved or optimised, explaining what you want to achieve (i.e. for planned but not yet implemented functionality). The code itself should document how this is done.

Use TODO in comments to mark places where updates or corrections are required. You should also include details of what updates or correction are required. This ensures that your comment is clear both to you (e.g. if you return to it after a period of time) and others. It is also good practice to include the author and date.

Similarly, when a method or class is retired using Javadoc @deprecated, you should always specify the time of retirement, explain what replaces the retired functionality, and by when the replacement should be made. This helps customers who use this functionality to manage the replacement.

A few rules of thumb are:

- All class files should contain formally required comments, as copyright header or author name.
- All classes, interfaces and, public and non-trivial internal methods should contain a descriptive Javadoc.
- All methods should contain comments that specify input parameters, side effects of a method to its parameters, and possible return values.
- All unobvious procedural code segments should contain descriptions of intentional and, if needed, implementation concerns or constrains.
- All complex algorithms should be thoroughly commented.
- Comment all variables that are not self-describing.
- Static variables should describe why they are declared static.
- Code that has been optimised or modified to ‘work around’ an issue should be thoroughly commented, so as to avoid confusion and re-introduction of bugs.

- Code that has been ‘commented out’ should be explained or removed.
- Code that needs to be reworked should have a TODO comment and a clear explanation of what needs to be done.
- If in doubt about the need for commenting, first try to refactor the code in order to make it clearer. If still in doubt, comment to clarify.

5.3 Code Conventions

Code conventions are programming guidelines that focus on the physical structure and appearance of a program, rather than its logic. They make the code easier to read, understand, and maintain. Code conventions usually include:

- Naming conventions for objects, variables, and procedures.
- Standardised formats for labelling and commenting code.
- Guidelines for spacing, formatting, and indenting.

5.3.1 Java

The descriptions in this section are based on Sun's Java Code Conventions (JCC) (see [JCC]).

5.3.1.1 *Naming Conventions*

Note: For language-neutral naming conventions see *Naming* on page 39.

- Names that represent packages should be lower case.
- Private class variables may be required to have an underscore suffix (this can be decided within a project).
- The name of the class is implicit, and should be avoided in a method name.
- JFC (Java Swing) variables should be suffixed with the element type.
- Default interface implementations can be prefixed with Default.
- Singleton classes should return their sole instance using the getInstance method.
- Classes that create instances on behalf of others (factories) can do this using the new(ClassName) method.

5.3.1.2 *Files*

- Java source files should have the extension .java.
- Classes should be declared in individual files with the file name matching the class name. Secondary private classes can be declared as inner classes and reside in the file of the class they belong to.

- File content must not exceed the defined width (80, 100, 120 or some other value).
- Special characters like tab and page break must be avoided. This differs from Java code conventions for indentation [JCC4].
- Split lines must be made obvious using proper indentation.

5.3.1.3 *Statements*

- Package and Import Statements:
    - The package statement must be the first statement of the file. Every class should belong to a specific package.
    - The import statements must follow the package statement. Import statements should be sorted with the most fundamental packages first, and grouped with associated packages together and one blank line between groups.
    - Imported classes should always be listed explicitly.
    - Prohibit '*' in import statements. This gives you more control at the expense of more import statements, and IDEs easily organise imports. This differs from Java code conventions for package and support statements [JCC312].

- Classes and Interfaces:

  Class and interface declarations should be organised in the following order:

  1. Class/Interface documentation.
  2. Class or interface statement.
  3. Class (static) variables in the order public, protected, package (no access modifier), private.
  4. Instance variables in the order public, protected, package (no access modifier), private.
  5. Constructors.
  6. Methods (no specific order).

- Methods:
    - Keep methods short.
    - Limit them to a single purpose and level of abstraction.
    - Method modifiers should be given in the following order:

      ```
      <access> static abstract synchronized <unusual> final native
      ```

    - The access modifier (if present) must be the first modifier.

- Types**:**
    - Type conversions must always be done explicitly. Never rely on implicit type conversion.
    - Array specifiers should be attached to the type not the variable (e.g. `Person[] persons`).

- Variables:
  - Variables should be initialised where they are declared and they should be declared in the smallest scope possible.
  - Variables must never have dual meaning.
  - Class variables should never be declared public.
  - Arrays should be declared with their brackets next to the type.
  - Variables should be kept alive for as short a time as possible.

- Loops:
  - Only loop control statements must be included in the for() construction.
  - Loop variables should be initialised immediately before the loop.
  - The statements break and continue should be used moderately in loops. Be careful to keep the obvious program flow.

- Conditionals:
  - Complex conditional expressions must be avoided. Use temporary Boolean variables instead.
  - In an if statement, the nominal case should be put in the if part and the exception in the else part.
  - The conditional should be put on a separate line.
  - Avoid executable statements in conditionals.

- Miscellaneous:
  - Avoid using magic numbers in the code. Instead declare numbers other than 0 and 1 as named constants.
  - Floating point constants should always be written with decimal point and at least one decimal.
  - Floating point constants should always be written with a digit before the decimal point.
  - Static variables or methods must always be referred to through the class name and never through an instance variable.

5.3.1.4 *Layout and Comments*

Layout**:**

- The basic indentation should be 2 or 4 white spaces (consistent with existing project conventions).

- Block layout should have the following format:

```
while (!done) {
    doSomething();
    done = moreToDo();
}
```

- Class and interface declarations should have the following format:

```
class Rectangle extends Shape
        implements Cloneable, Serializable {
    ...
}
```

- Method definitions should have the following format:

```
Public void someMethod()
        throws SomeException {
    ...
}
```

- The `if-else` class of statements should have the following format:

```
if (condition) {
    statements;
}

if (condition) {
    statements;
} else {
    statements;
}

if (condition) {
    statements;
} else if (condition) {
    statements;
} else {
    statements;
}
```

- The `for` statement should have the following format:

```
for (initialization; condition; update) {
    statements;
}
```

- The `while` statement should have the following format:

```
while (condition) {
    statements;
}
```

- The `do-while` statement should have the following format:

```
do {
    statements;
} while (condition);
```

- The `switch` statement should have the following format:

```
switch (condition) {
case ABC:
    statements;
    // Fall through
case DEF:
    statements;
    break;
default:
    statements;
    break;
}
```

- A `try-catch` statement should have the following format:

```
try {
    statements;
} catch (Exception exception) {
    statements;
}

try {
    statements;
} catch (Exception exception) {
    statements;
} finally {
    statements;
}
```

- Single statements contained in `if-else`, `for`, `while`, `do-while`, and `try-catch` should be enclosed in blocks:

```
if (condition) {
    statement;
}
while (condition) {
    statement;
}
for (initialization; condition; update) {
    statement;
}
```

White spaces:

- Operators should be surrounded by a space character, except for unary operators and operator '.'. (e.g. `dog.bark(), a++ b--, -c`).
- Java reserved words should be followed by a space.
- Commas should be followed by a space.
- Colons should be surrounded by a space.
- Semicolons in for statements should be followed by a space.
- Logical units within a block should be separated by one blank line.
- Methods should be separated by at least one blank line.
- Statements should be aligned wherever this enhances readability.

Comments:

- All source files should begin with a c-style comment that lists the class name, version information, date, and copyright notice (Use the three-line form of the copyright with a proper license notice). This differs from Java code conventions for beginning comments [JCC311].

  ```
  /*
  * Classname
  *
  * Version information
  *
  * Date
  *
  * Copyright notice
  */
  ```

  Note that the usage of keyword expansion and the keywords $Revision$, $Author$, $Date$ and similar may lead to problems with binary files and in the comparison of files that contain them. Furthermore, Subversion fixed-width keywords may cause corruption of UTF-8 characters (e.g. in personal names), since the width is measured in bytes. Therefore, keyword expansion is discouraged, except if it has already been a widely accepted practice in the project.

- The '/*-' form of comment is not allowed. The special form with '-' is recognised by the indent utility. This differs from JCC section 5.1.1.
- Do not use '//' for commenting out blocks of code. Instead use version control software to allow recovery of deleted code. This differs from Java code conventions for end-of-line comments [JCC514].

Javadoc comments:

- All public classes and public and protected methods within public classes should be documented using the Java documentation (Javadoc) conventions [JAVADOC].

- Javadoc comments should have the following format:

```
/**
* Return lateral location of the specified position.
* If the position is unset, NaN is returned.
*
* @param x X coordinate of position.
* @param y Y coordinate of position.
* @param zone Zone of position.
* @return Lateral location.
* @throws IllegalArgumentException If zone is <= 0.
*/
public double computeLocation(double x, double y, int zone)
        throws IllegalArgumentException {
    ...
}
```

- There should be one white space after the comment identifier.
- Comments should be indented relative to their position in the code.
- The declaration of anonymous collection variables should be followed by a comment stating the common type of the elements of the collection.

In addition to the above conventions, the general JCC programming practices [JCC10] may be useful. Also note the following recommendations:

- The left-hand comparison style with constants or expressions to the left in any comparison is inspired by C comparison (`1 == a`) that protects code in compile time from a mistake like if (`1 = a`). With comparison like if (`"integrate".equals(str)`), if `str` is null, the expression will return false instead of raising a null-pointer exception.
- `if (isOk)` is better than `if (true==isOk)` or `if (isOk==true)`.

See also [JAVAGEN].

5.3.2 C

Although there are similarities in Java and C syntax, there are also differences (namespaces, method declarations), so it is not possible to use exactly the same set of conventions for Java and C. Conventions for the C language exist [C], but are not as accepted as Java conventions.

In most cases, C code is legacy code that needs some maintenance and very little development. New code should therefore conform to the already accepted legacy code style, and keep the readability of the project as a whole. It is wrong to partially introduce new code conventions.

5.3.3 C++

The following C++ code conventions are widely accepted:

- 25 lines 80 columns.

  If you are writing code that is longer than 80 columns or 25 lines, you should consider splitting it up into functions. This helps to keep the functions short and makes them easier to understand.

- Whitespace and indentation.

  Every block of code and every definition should follow a consistent indention style. This usually means everything within { `and` } but also applies to one-line code blocks. Use a fixed number of spaces for indentation. Recommendations vary (2, 3, 4, 8 are all common). If you use tabs for indention you have to be aware that editors and printers may deal with and expand tabs differently. The K&R standard recommends an indentation size of 4 spaces [KR].

- Placement of braces.

  Compound statements are subject to different coding styles that recommend different placements of opening and closing braces (`{ and }`). Some recommend putting the opening brace on the line with the statement, at the end (K&R). Others recommend putting these on a line by itself, but not indented (ANSI C++). GNU recommends putting braces on a line by itself, and indenting them `half-way`. You should choose a brace-placement style and stick to it.

- Comments.

  Comments are made in block form or as single lines.

  - Single-line comments (informally, C++ style), start with // and continue until the end of the line. If the last character in a comment line is a \ the comment will continue in the next line.
  - Multi-line comments (informally, C style), start with /* and end with */.

- Identifiers.

  C++ identifiers must start with a letter or an underscore character (_). This can be followed by a series of letters, underscores or digits. None of the C++ keywords can be used as identifier. Identifiers with successive underscores are reserved for use in the header files or by the compiler for special purposes (e.g. name mangling). This leaves a lot of freedom in naming but you should stick to the following rules:

  - Hungarian notation is considered outdated, since it is prone to errors and requires some effort to maintain without any real gains in today's IDEs.
  - Avoid leading underscores. They are reserved for the compiler or internal variables of a library, and can make your code less portable and more difficult to maintain.
  - Reusing existing names: Do not use the names of standard library functions and objects for your identifiers. These names are considered reserved words and using them in unexpected ways can cause errors.
  - Always use sensible, unabbreviated, correctly-spelled, meaningful names for identifiers. Use English (GN3 is an international project, and C++ and most libraries use English) and avoid short cryptic names. This makes it easier to read and type a name without having to look it up.

- An identifier's name should indicate its purpose. For example, `foobar` is probably not a good name for a variable that stores a person's age. Identifier names should be descriptive, so n would not be a good name for a global variable that represents the number of employees. However, you should try to find a name that is not too long. Therefore, this rule can be relaxed for variables that are used in a small scope or context. Many programmers prefer short variables (such as `i`) as loop iterators.
- Capitalisation: Conventionally, variable names start with a lower case character. In identifiers which contain multiple natural language words, underscores or capital letters are used to delimit the words (e.g. `num_chars` (K&R style) or `numChars` (Java style)). It is recommended that you pick one notation and stick to it within one project.
- Constants: Use uppercase when naming `#defines`, constant variables, `enum` constants and macros, and use underscore separators. This makes it very clear that the value cannot be altered (and for macros, indicates that you are using a construct that requires care).
- Use descriptive names for functions and member functions that indicate their purpose. Since usually functions and member functions perform actions, the good names usually contain a mix of verbs and nouns (e.g. `CheckForErrors()` rather than `ErrorCheck()`, and `dump_data_to_file()` rather than `data_file()`).
- Balance your use of keywords: If you use fewer keywords, you may have to type less, but the code may be less immediately readable. If you use many keywords, your code may be clearer and fewer errors may occur.
- Pointer declaration: Due to historical reasons some programmers refer to a specific use as:

  ```
  // C codestyle
  int *z;
  // C++ codestyle
  int* z;
  ```

  The second variation is preferred by C++ programmers and helps identifying a C programmer or legacy code.

For further information, see [C++].

5.3.4 Source Code Formatters

Source code formatters (also known as source beautifiers) accept program sources and generate equivalent sources that are nicely formatted, according to the sources' language syntax. This includes indentation, a normalised case for identifiers, etc.

Source code formatters improve the readability of source code and ensure adherence to code conventions. Properly formatted code is much easier to follow, understand and maintain. Source code formatters also make it easier for developers to follow in-house code conventions.

Formatting is not only applied to class sources, but also to HTML, XML, CSS and other resource files such as properties, CVS comments, etc. Formatting is, for example, applied to the following source points (according to Java reference standards):

- Package and import statements.
- Class and interface declarations.
- Beginning, implementation and documentation (JavaDoc) comments.
- Declaration, initialisation and placement of variables.
- Method declaration.
- Formatting of method invocations, expressions, control statements and constructs (`return`, `if`, `if-else`, `for`, `while`, `do-while`, `switch`, `try-catch`, etc.).
- Usage and placements of braces.
- Usage of blank lines and blank spaces to improve readability (for example separation of class and interface members).
- Naming conventions and indentation, including line length and the way the lines are wrapped.
- Some programming practices such as referencing class (static) variables and methods, variable assignments, usage of parentheses and usage of special comments (such as XXX, FIXME, TODO).

Modern IDEs (e.g. Eclipse and NetBeans) provide extensive out-of-the-box support for code formatting. Formatting templates in such IDEs can be changed according to developer preferences or corporate/team conventions. There are also many external free and commercial tools that aid source code formatting (e.g. Jindent, Code Chameleon, Semantic Designs' Source Code Formatter, Polystyle, SourceFormatX, etc.).

5.4 Software Parameterisation and Configuration

When writing code it is best to not assume or hardcode any parameters or settings that may need to be changed at a later stage. Wherever possible you should define reasonable defaults that apply to the majority of cases, adopting a 'configuration by exception' approach in which the users only need to supply values if they want to override the default ones. This applies both for software interfaces and for configurations. It is also good practice to make service implementations fully configurable without a need for a rebuild.

The configuration must contain all key settings of the system. These parameters must be factored out from the code, even if they appear to be constant at the time. Note that some frameworks require specific parameters to be located at predefined places. For example, a JDBC connection should be set using the <connection-url> element within the corresponding *-ds.xml file which is (through use of the same JNDI name) referenced in the web.xml and ejb-jar.xml (with EJB) files, and/or some other application server-dependent file.

Static parts of the configuration that are related to deployments set up by integrators or administrators who perform software installations, should be located in a configuration file. Depending on the platform used, this may be a file containing key-value pairs or an XML hierarchy. Some good places for storing configuration data are:

- Java .properties files (accessible via java.util.ResourceBundle class).
- For web applications: <context-param> in the web.xml file (available through javax.servlet.ServletContext).
- For JSF-based applications: <managed-property> of <managed-bean> within the faces-config.xml file.

If the loading of configuration parameters, resources, or files is not implemented by the underlying framework, but by the software, it should be performed only once, by implementing a Singleton pattern or a static block of a class.

Where different configuration versions for various deployment instances are handled, clear methodology should be used for version control. The configuration of different instances may, for example, be kept in separate branches. Also, a procedure for the using the file comparison tool (e.g. “diff” command) should be defined (see *Version Control Management* on page 5).

Using a consistent structure for all types of configuration file makes it easier to compare them. It is good practice to precede each parameter with a comment describing its scope, purpose, default and possible values. This comment should be present even if the parameter is not explicitly set. Placeholder comments with different example values that can be uncommented to set the value can also be provided. An example for this can be found in the Apache web server configuration file:

```
# Listen: Allows you to bind Apache to specific IP addresses and/or
# ports, instead of the default. See also the <VirtualHost>
# directive.
#
# Change this to Listen on specific IP addresses as shown below to
# prevent Apache from glomming onto all bound IP addresses (0.0.0.0)
#
#Listen 12.34.56.78:80
Listen 80
```

See *Internationalisation and Localisation* on page 62 for internalisation- and localisation-related aspects of the configuration.

The parts of the system configuration that can be performed by application users should preferably be stored in an application's persistent storage (e.g. a relational database system like RDBMS), not in configuration files.

5.5 Handling Classifiers

Whenever classifier data is time-dependent, the attributes `validForm` and `validUntil` or similar should be used to represent the beginning and end of the time span in which the classifier's value is valid. These attributes should be dates or timestamps, as appropriate. Null values in the attributes mean that they are unlimited (e.g. null value for `validUntil` means that the record is still valid and the time of deprecation is still unknown). Records outside those boundaries are considered invalid (or better, deprecated) for current use, and should only be used for historic purposes. The attributes defining the validity interval should be used even when a history of classifier changes is implemented by connecting values with their predecessors or successors.

Where the classifier subset is needed, it should be fetched using appropriate attributes in the classifier data. If only one record should be selected, it is wrong to use the primary key in the code, configuration file or even configuration table, especially if the primary key is auto-generated. It is better to introduce the numeric attribute appropriate for the selection, using an enumeration or the named `static final int` constant. The primary key obtained by using a corresponding query can be cached in the Singleton class, also using the `static final int` declaration.

Numeric attributes can also be introduced if a request for the non-standard order of the classifier data is issued (when the natural attributes of the classifier are not sufficient).

5.5.1 General Rules for Classifier Maintenance

- The general rules for maintaining classifiers are:
  - A complete classifier is defined using two documents:
  - A document that describes the model of classification data (data model).
- A document that contains the classification data itself (classifier).
- Each record in a classifier must have unique identification code, called identifier.
- The format of the identifier (numeric, alphanumeric, and similar), as well as maximum length or minimum and maximum value, must be clearly defined in the data model.
- The format, maximum length or minimum and maximum value of any other data in the classifier must be clearly defined in the data model.
- The identifier represents the identity of the record.
- The identifier of a publicly available record is not allowed to be changed (see [QA]).
- It is not permitted to have two records that contain identical data but have different identifiers.

- Every single record must be uniquely identifiable by a human, without knowledge of identifiers. This identification can be done using one or several record fields (attributes).

- The classifier and data model must be published in a form that is appropriate for automatic processing (e.g. XML format for the classifier, and XSD format for the data model), and accessible through the Internet (e.g. via Web service).

5.5.2 Rules for Time-Dependent Classifier Maintenance

The rules for maintaining time-dependent classifiers are:

- If the data in the classifier is time-dependent, every record must have a field indicating if the record is archived or not, to indicate the validity of the record. This field can be implemented using one of the following:
  - An indicator (archived / not archived).
  - An expiration date.
  - An expiration date and time (if better precision is needed).
  - Some other, unambiguous, representation.

  For instance, it is possible to adopt the rule that an empty expiration date means that the record is valid, and that an entered expiration date means that the record is archived and should not be used for new data.

- Once a record is published, it is not allowed to delete it from the classifier. It can only be archived by setting an expiration date, indicator or similar.

- If a name has changed (e.g. people's first or last name, an organisation name, or similar), the rule is that if the identity stays the same, the identifier must also be unchanged (see [QA]).

- Where needed, a classifier must include archived data that is necessary for the classification of historic records.

- If a new data model or new classification is adopted, the users of the classifier must be informed about the changes.

- If a new data model or classification is adopted, the time frame and migration procedures from the old to the new classifier must be defined.

5.5.3 Rules for Classifier Maintenance with Numeric Non-Disclosing Identifiers

The rules for maintaining classifiers with numeric non-disclosing identifiers are:

- The identifier must be numeric.
- The identifier must be a non-disclosing random number that does not reveal any connection between the entity (e.g. school) and its attributes (e.g. name, city, district, ownership).

5.6 Error and Exception Handling

This section uses Java as example language, but also applies to any other language that supports exceptions.

To make testing and debugging easier, you should write code without masking errors (by fixing them). For example, you should not:

- Correct arguments.
- Create missing objects.
- Ambiguously catch or ignorantly 'eat' exceptions.
- Use the switch statement without a default case.
- Use the `if/else` if statement without enclosing else.

If invalid arguments, unsupported operations or failures are encountered, `null` or similar value information should not be returned to indicate an 'error' status. Such attempts are likely to either result in an error later on that is harder to trace, or in a `NullPointerException` in a different place. Therefore, it is better to generate an appropriate exception.

You should trust that existing code meets contractually defined requirements. If you have doubts about that, you should verify your concerns without negative impact on performance in production use. Include key tests in the code using the assert statement, and unit tests. Logging (see *Java Logging* on page 57*)* can also be helpful.

Using exceptions can dramatically simplify code structure and the handling of irregular situations. Code that potentially raises an exception should not be surrounded by ambiguous `try-catch` blocks catching all the exceptions. If an exception is caught, an appropriate message should be recovered or logged, preferably using `exception.printStackTrace(PrintWriter).` Catch block should not both log and throw another exception, because this doubles the log. Return from the method from inside catch block is only acceptable if the recovery was successful, and the outcome is regular. In this is the case, it is not necessary to log the exception. If this is not the case, it is better to propagate the exception further.

Generated exceptions should be of an appropriate type (user-defined, if necessary), and should be declared in the `throws` clause. A method should only throw more than one exception if multiple irregular situations must be handled in different ways. Exception replacement, where the original exception is lost or only the exception message (`exception.getMessage()`) is saved, should not be used, since original stack trace is lost that way. It is better to include the original exception in the created one as the cause. For the same reason new exceptions should not be generated in final blocks. `InterruptedException` generated by threads should not be silently ignored. Also, do not rely on `e.getMessage()` to detect the cause of a problem, as it may make the code fragile and dependent on an underlying implementation of exceptions along the stack.

While working with EJB exceptions, appropriate rollback behaviour should be defined using an `ApplicationException` annotation, after which the exceptions can be raised in the code as needed.

For detailed examples and exception-related guidelines, see [EXCEPT].

5.7 Java Logging

All applications need to log exceptions and important application events. Using Java `System.out` and `System.err` is only sufficient for very simple applications. In all other cases a logging toolkit must be used. There are several available alternatives, including Apache Log4J [Log4J], LOGBack [LOGB], the JRE implementation of the Java Logging API – `java.util.logging`, also known as j.u.l [JUL], and logging wrappers like Apache Commons Logging [ACL] and SLF4J (Simple Logging Facade for Java) [SLF4J].

When logging, related data should be logged in a single message. This simplifies debugging and the analysis of the logs in a multi-threaded environment. Logging frameworks provide some useful features that are not available with plain streaming into a file:

- Infrastructure and interfaces for logging.
- Various settings for log storage (e.g. how long logs are kept, in which files logs are held, how much space logs may occupy).
- Formatting of logging messages.

Besides messages and exceptions, logging frameworks can be configured to log:

- The date and time of an event.
- Specific message levels.
- The identifier of a thread generating the event. (This is useful when tracing an application that is running in a multithreaded environment).
- Thread-bound end user or client identities (username, IP address, hostname, security principal, or http session ID). This requires some custom coding (e.g. using Log4J Nested Diagnostics Context or Log4J and SLF4J Mapped Diagnostic Context, which facilitate distinguishing of interleaved log output from different sources).

- Source code location details, like class, method, and file names and line number of the code invoking the logging. The latest two require debug information in the compiled code (by using Java compiler '-g' flag).

5.7.1 Logging Message Levels

The logged messages are split into several levels, typically:

- **FATAL** (only in Log4J and Apache Commons Logging).

  Serious errors that are likely to lead to premature application termination. These errors indicate or cause system instability or unavailability of a key resource: e.g. lack of memory, lost database connections, deadlocks that halt the system. A message of this level should be immediately visible on a status console. In SLF4J and Java Logging API, this category has been dropped on the grounds that the logging framework is not the place to decide whether an application should be aborted.

- **ERROR** (SEVERE in Java Logging API).

  Errors in the regular operation of the system which prevent normal program execution but are likely to allow the application to continue running. Examples are unexpected conditions, exceptions, or runtime errors that can be handled or recovered from. A message of this level should be immediately visible on a status console. Besides being logged, these errors should also be reported in a manner that is understandable to end users.

- **WARN** (or WARNING in Java Logging API).

  Should be used to record transient problems or undesirable, unexpected or potentially harmful situations that are not necessarily errors. A message of this level is most likely to be shown on a status console. Problems that affect end users and should influence their behaviour should be also reported to them. However, use of deprecated APIs or poor use of API may be just logged.

- **INFO**.

  Describes interesting runtime events, like startup, shutdown and receipt of user request. These messages should inform on the progress of the application at coarse-grained level during its normal operation. Provided information is likely help in interpretation of problems that may occur latter. A message of this level is most likely to be shown on a status console. Therefore, these messages should be generated conservatively and make sense to both end users and system administrators. They should not be used to log debug information, trace program execution or log frequently repeating events.

- **CONFIG** (only in Java Logging API).

  Messages that describe static configuration information that may help debug problems associated with particular configurations. These messages are normally only logged.

- **DEBUG** (FINE and FINER in Java Logging API).

  These messages are produced by code added to debug an application by tracing its flow or internal data. These fine-grained messages are only logged. The logging of these messages should only be active while an error is being searched for or examined.

- **TRACE** (FINEST in Java Logging API).

  Used to log the most detailed informational events that occur on application execution. This level provides even more verbose information than the previous level.

It is important to appropriately and consistently use the available logging levels. Instead of DEBUG and TRACE, Java Logging API uses FINE, FINER, and FINEST, which provides even finer tuning of logging granularity by decreasing importance and increasing volume of the output. In the list above, these three levels are classified according to SLF4J mapping.

5.7.2 Logging Framework Selection

If your project already successfully uses some logging framework, you should stick to it.

For smaller projects with strict footprint limitations, the Java Logging API is useful, as it is already available with Java runtime. However, it lacks some useful features provided by Log4J and SLF4J, e.g. NDC and MDC diagnostic contexts, support for usual external log destinations (syslog, MS Windows event log, JMS channel), comprehensive file rotation by date or size, etc.

Log4J is currently the most popular logging framework and it scales well with the size of software project. LOGBack is intended to be a successor to Log4J, In comparison to its predecessor, it provides additional functionality, e.g. automatic reloading of configuration files, extended filtering options and automated compression of archived log files.

If your development needs to integrate internal logging of used components, or is a library or component intended to be used with various setups with different logging implementations, it is wise to use SLF4J. It should be noted that Apache Commons Logging faces some serious criticisms [CRITIC], which led to the creation of SLF4J. SLF4J allows dynamic binding with available underlying loaders, works smoothly with applications that have multiple class loaders, and can bridge the loggers used by other project components.

SLF4J may be also considered for simpler projects, as it allows more efficient passing over disabled messages, as its format of logging methods allows skipping string concatenation and conversion of objects into strings.

5.7.3 Logging Recommendations

Regardless of which logging framework you use, the following guidelines should always be observed

- Adhere to levels.

  Decide on a scheme that assigns each log message to a particular level (e.g. FATAL, ERROR, DEBUG etc.) and use this convention consistently. See *Logging Message Levels* on page 58 for a generally applicable scheme.

- Direct messages to different loggers according to purpose.

  For example, security- or accounting-related messages may be directed to separate loggers.

- Configure appropriate logging levels.

  For example, do not forget to disable debug and trace messages in production environments.

- Log relevant and meaningful information only.

  Logging everything makes code less readable, makes relevant information harder to find, and can hurt performance. While extensive logging can be useful to provide a better insight and help with diagnostics, performance, security, and privacy concerns should be considered.

- Avoid logging at frequently or iteratively executed spots.

  This can greatly increase log size and hurt performance. On modern platforms, the invocation of a logging method lasts below 100ns if the message is logged, or below 10ns if not. However, do not ignore the performance penalty induced in both cases by preparation of message arguments.

- Wrap time-consuming log messages in conditional statements.

  If some form of object retrieval or preparation is needed to produce a message, it is justified to wrap the logging related code in an if (`logger.isDebugEnabled()`) or similar statement. Note that the syntax of SLF4J logging methods postpones string creation and concatenation, so this conditional check can be often avoided with simple passing of objects as arguments, which increases code readability.

- Avoid multi-message logs.

  Enclose each logging item in a single logging message, with all necessary context information that is not already provided at higher levels. Splitting a message into multiple logging statements makes later filtering and analysis more difficult.

- Be careful with copy and paste.

  This can easily lead to the use of an inappropriate logger, or (less likely) logging level.

- Think twice before writing your logging wrapper.

  It is quite easy to write an inefficient wrapper that decreases logging performance (even for inactive statements). For example, by using `Logger.getLogger` on every message.

- Escalate logging if a problem is detected in a production environment.

  If a problem in runtime environment is detected, logging can be augmented by inclusion of debug and trace messages. Various logging frameworks provide different types of support for doing this in runtime. The simplest solution in Log4J is to use `DOMConfigurator.configureAndWatch` method in order to detect changes in the log configuration file without restarting the application. A more complex alternative suitable for elaborated management arrangements is to use JMX (Java Management Extensions) API to manage application settings, including logging level. You can also write a logging wrapper that automatically changes logging level when an error is logged. However, this should be done only on closely monitored system instances.

- Be careful with logging sensitive data.

  Logging of passwords (even for unsuccessful logins) creates a serious security risk. Also, logging of email addresses, phone numbers, or other sensitive personal data may threat user’s privacy. If logging such information is necessary, it should be performed using a security-specific logger, which may store logged messages in a secure way.

- Log errors in an informative way and properly present them to end users.

  Error logs should include exceptions and their stack traces, and where appropriate, thread and user identifiers. Related informative, warning, and configuration log messages can also provide useful contextual information. Also make sure that the cause of the error does not prevent it from being logged. It is also a good idea to log a (near) unique error ID and show it on the error page or problem reporting form presented to end users. This facilitates later matching of reported problems with logs.

- Rotate log files.

  Log files can grow very large and difficult for browsing and finding of useful information. Therefore, do not forget to configure logging to periodically archive old messages by rotating log files when they reach some size, or on daily or weekly base, depending on logging intensity. In order to limit storage consumption, log files older than one week or month should be removed.

These recommendations are a good base for a project-specific logging policy. However, the details on usage of logging levels, purpose of available loggers, and possible security and privacy concerns, should be defined at the project level, communicated to the developers, and enforced.

For additional useful recommendations, see [LogInf].

5.8 Handling Temporal Data

When the database and related coding is designed, it is crucial to detect temporal components and what historical information is needed for an object (for example, to record the validity period) [TEMPORALOBJ]. This can have a significant impact on the multiplicity of relations and the formulation of database queries, which usually directly affect the code as well as software and user interfaces. Adding a second or third temporal column to a table greatly complicates the maintenance of an application. It is good practice to recognise a need for attributes like validity start, validity end, and timestamp in advance, to indicate when an item was changed or its validity recorded.

Data expiration time is an important concept for data management in many application areas. Applications can benefit from being expiration time-enabled by experiencing a lower transaction workload while at the same time being able to immediately free memory occupied by stale data.

One way of handling temporal data is to use an audit log. This can be either a file or a database table. Files are easier to write to but harder to read when they grow large. If information has to be more accessible, Effectivity can be used [EFFECTIV]. This marks an object with the time period for which it is considered valid. As you manipulate your objects, you can use this time period to get the right object at the right time. The problem with Effectivity is that it is obvious and explicit. Anyone who uses the object has to take its temporal aspect into account, which can complicate a model considerably. Therefore, if temporal issues are pervasive, it is often sensible to hide them when you do not need them, and make them more convenient to work with when you do.

If you have just a few object properties that need easily accessible temporal information, you can use a Temporal Property on those objects [TEMPORALPROP]. This allows you to remove the obvious Effectivity date, and instead use what looks like a regular property, but with an accessor that takes a date as an argument. This allows you to query the value of this property on a specific date.

The usage of a 'temporal database' covers the built-in time aspects that include 'valid-time' and 'transaction-time' attributes [TEMPORALDB]. Valid time denotes the time period during which a fact is true and transactional time is the time period during which a fact is stored in the database. These attributes go together to form bi-temporal data.

Any application that may be used in multiple time zones should internally represent time and timestamps by using UTC time. Simultaneous use of date, time, and timestamp types often causes problems due to their different temporal resolution. With JDBC it is recommended to use `ResultSet.getTimestamp()`, rather than `ResultSet.getDate()`. With EJB it is best to use the `@Temporal` annotation to specify a Java type into which some temporal attribute is converted (e.g. by putting `@Temporal(TemporalType.TIMESTAMP)` before private `java.util.Date dateFrom;`). Furthermore, when using the EJB `Query.setParameter()`, you should use an additional parameter of `TemporalType` to specify whether the provided value represents date, time, or timestamp.

A common mistake is related to getting temporal data from the user in 3-layer and n-layer applications. When handling time input, time information sent through the form should be interpreted according to the time zone provided within the form (usually as hidden field), the user's preferences, or the time zone of the server hosting the application. If the current time is the default, it is good practice to explicitly set it in the input field when the form is created. You should never rely on the local client time (e.g. by using JavaScript), and avoid using the time provided by the database server.

The best ways to allow users to enter 'now' are to either provide them with a button that gets the current time from the server (e.g. by using AJAX) and sets a special value that is interpreted by the server upon receiving the form content, or to populate the input with the time that is provided by the server during form instantiation. In situations where time constrains are mandatory (e.g. searches), an empty value in 'From (time)' input may be interpreted as the current time minus, for example, 10 minutes, while in 'To (time)' it would represent the current time. If a user interface's design permits both date and time to be specified within the same input, it is recommended to interpret the value with the current date only as 'now', while other dates without time should revert to 00:00, except if the input is used to represent the end of a period, in which case it should be interpreted as 24:00.

5.9 Internationalisation and Localisation

Internationalisation and localisation are means of adapting computer software for different languages and regional differences. Internationalisation is the process of designing a software application so that it can be adapted to various languages and regions without changes to its engineering. Localisation is the process of adapting internationalised software for a specific region or language by adding locale-specific components and translating text.

Focal points of internationalisation and localisation efforts include:

- **Language.**
  - Computer-encoded text:
    - Alphabets/scripts. Most recent systems use the Unicode standard to solve many of the character encoding problems.
    - Different numeral systems.
    - Writing direction. The direction of text varies between languages. For example, most European languages read from left to right, but languages like, for example, Persian, Hebrew and Arabic read from right to left.
    - Spelling variants for different countries where the same language is spoken. For example, 'localization' (en-US, en-CA, en-GB-oed) versus 'localisation' (en-GB, en-AU).
    - Text processing language differences. For example, the concept of capitalisation which exists in some scripts and not in others, different text sorting rules, etc.
  - Input.
  - Enablement of keyboard shortcuts on any keyboard layout.
  - Graphical representations of text (printed materials, online images containing text).
  - Spoken (Audio).
  - Subtitling of film and video.

- **Culture.**
  - Images and colours. Comprehensibility and cultural appropriateness must be considered.
  - Names and titles.
  - Government assigned numbers (such as the Social Security number in the US, National Insurance number in the UK, etc.) and passports.
  - Telephone numbers, addresses and international postal codes.
  - Currency (symbols, positions of currency markers).
  - Weights and measures.
  - Paper sizes.

- **Writing Conventions.**
  - Date/time format, including the use of different calendars.
  - Time zones (UTC in internationalised environments).
  - Number formatting (decimal points, positioning of separators, character used as separator).

- **Any other aspect of the product or service that is subject to regulatory compliance.**

The distinction between internationalisation and localisation is subtle but important. Internationalisation is the adaptation of products for potential use virtually everywhere, while localisation is the addition of special features for use in a specific locale. Internationalisation is done once per product, while localisation is done once for each combination of product and locale. The processes are complementary, and must be combined to produce a system that works globally. Subjects unique to localisation include:

- Language translation.
- National varieties of languages.
- Special support for certain languages such as East Asian languages.
- Local customs, content, and symbols.
- Sorting order.
- Aesthetics, cultural values and social context.

For further information, see [LOCAL].

5.9.1 String Localisation

The current prevailing practice is for applications to place text in resource strings which are loaded during program execution as needed. These strings, stored in resource files, are relatively easy to translate. Programs are often built to reference resource libraries depending on the selected locale data.

An application that supports multiple languages would therefore be designed to select the appropriate language resource file at runtime. Resource files are translated into the required languages. This method tends to be application-specific and vendor-specific. The code required to manage date entry verification and many other locale-sensitive data types also must support differing locale requirements. Modern development systems and operating systems include sophisticated libraries for international support of these types.

Some guidelines that should be followed when creating resource strings are:

- Resource strings should be identified with descriptive English identifiers or strings.
- In case of long phrases or sentences a shorter description should be used as identifier.
- Even with one word resources, a longer description may be required to provide an appropriate translation for a resource with multiple meanings and uses in English.
- Consistency between resource identifiers, their values, and usages in code should be carefully maintained.
- Already existing resources should be used whenever appropriate to reduce the amount of translation and to improve linguistic consistency.
- Developers should avoid construction of phrases from separate fragments, since these cannot be localised well.
- Punctuation marks should be included in resource values. However, to reduce duplication of similar resources, the colon character should not be used after label texts.
- The insertion of mark-ups (e.g. <br>) tags into language resources is strongly discouraged.

Maintaining parallel versions of texts throughout the life of the product is difficult. For instance, if the meaning of a message displayed to the user is modified, all of the translated versions must be changed. This, in turn, results in a longer development cycle.

If internationalisation or localisation is provided though configuration files that store the text strings which are displayed in the user interface, then different version of these resources should be stored in separate resource/configuration files, and named consistently using file names with appropriate language (ISO 639), country (ISO 3166) and locale codes. The coding guidelines for parameterisation apply to this (see *Software Parameterisation and Configuration* on page 52). However, the described naming of resource files permits their simultaneous existence in a single version of code, which is not necessarily the case with alternate configuration files.

Comments preceding resource names and values can provide contextual information that can greatly improve translation quality by reducing ambiguities. The challenge of maintaining diverse resource files can be addressed using consistent naming conventions for resources and implementing or using a merging tool. One software library that aids this is GNU `gettext`. Another solution is to implement a tool that would ideally provide the following features:

- Layout, comments and order of resources are copied from the master file.
- Resource values are copied from the localised file.
- Resources that are undefined in the localised file are marked with special comments to indicate where missing resources need to be set.
- Values that are present in the master file are inserted as specially marked comments before actual values from the localised file to make it easier to check equivalent values.
- Resources that only exist in the localised file are copied to the end of the merged file, so renamed resources can be reused and obsolete entries can be identified.
- Duplicate resource keys are checked and reported. This is generally not an error, but may indicate an undesired redundancy.

For example, if the reference file contains:

```
#Our resources
Text = Text
Language = Language
```

and the modification file contains:

```
Language = Idioma
#this resource is obsolete
Tongue = Lengüeta
```

then the merged file looks like this:

```
#Our resources
#REF: Text = Text
    # Text =
#REF: Language = Language
Language = Idioma

#### Extraneous resources
Tongue = Lengüeta
```

Access to already existing translations to other languages may also help to reduce ambiguities that arise when translating to a new language. However, it is better to use conclusions obtained from existing translations to permanently resolve such issues by renaming, forking, or commenting textual resources.

Many localisation issues (e.g. writing direction, text sorting) require more profound changes in the software than text translation (OpenOffice.Org, for example, achieves this with compilation switches).

To anticipate any application design problems that may arise during localisation, a testing practice called pseudolocalisation [PSEUDOLOC] can be used to pinpoint the strings and their usages within the user interface that are not prepared for localisation. The idea behind this approach is to generate localised versions of original texts that are clearly distinguishable, about twice as long as the original strings, recognisable by testers and developers and which use long words and special characters used in target languages. Such a string can be generated by a small program that processes the resource files, transforming, for instance, ‘Print Preview’ into something like ‘[LßЧђЖšĐž Print Preview]’.

Another solution is to use an automatic translation tool to coarsely translate the original string resource values into the target language. The benefit of this approach is that it provides the specific features of the target language. The original text can be included in results to make testing easier for the QA team (thus, ‘Print Preview’ could become ‘Print Preview#Vista previa de impression’).

Since this practice may be used early in the software development cycle, it is possible to check localisation support without really translating the texts appearing in an early version of the user interface. Since these texts may easily change, pseudolocalisation also reduces unnecessary translation.

5.10 Security

Security is crucial in the software development process. Exploiting the vulnerabilities of insecure software can lead to the corruption of sensitive data. It can also lead to compromising the operating system that the software is running on.

It is recommended to use the SD3 software development strategy (a strategy created and used for building software at the Microsoft Corporation [SD3]) to ensure that your application is:

- Secure by design.

  Security features are blended into the application at the design level. Developers follow secure coding best practices and implement security features in their applications according to the design.

- Secure in deployment.

  The application is maintained through updates and security patches. The maintenance process should be simple and (preferably) automated. The application should allow backups of itself and its data.

- Secure by default.

  If an application is installed with its default settings, it is secure. All optional features of the application should be disabled by default. If a feature is required by a user, it has to be enabled as a user's choice (either during the installation phase or while the application is running).

It is important to deal with potential vulnerabilities at the design level. Removing vulnerabilities at a later stage of the software development process (coding, testing, shipping) entails significantly increased costs.

The overall overview of security is provided in the SA2 T4 Security Practices Cookbook on the GÉANT intranet [SC]. There is also the GN3 SA2 T4 Security Expertise Consultancy service through which the experts provide consultancy in areas like network, host, application, and monitoring security in accordance with the procedures and rules specified at [SCserv].

5.10.1 Common Application Security Vulnerabilities

According to the Open Web Application Security Project (OWASP) see [OWASP]) the top 10 application security vulnerabilities are:

- Invalidated input.
- Broken access control (restrictions on what authenticated users are allowed to do are not properly enforced).
- Broken authentication and session management (account credentials and session tokens are not properly protected).
- Cross-Site Scripting (XSS) flaws (the web application can be used as a mechanism to attack to an end user's browser).
- Buffer overflows.
- Injection flaws (e.g. SQL injection, command injection).
- Improper error handling.
- Insecure storage.
- Denial of service.
- Insecure configuration management.

Most of the vulnerabilities listed above can be dismissed using coding best practices (see *Best Practices and Recommendations for Security-Related Coding* on page 69). Other vulnerabilities (such as insecure configuration management) depend on the production environment configuration and should be dealt with using appropriate best practices (see [PCMAG] and [SECURITY] for sample web server configuration best practices). For more information on common security vulnerabilities, see [VULNERABILITIES]. For more information on role-based access controls, see [ROLES].

5.10.2 Security Threat Modelling

Threat modelling is a process that identifies, analyses and documents software vulnerabilities. The threat modelling steps are:

1. Identify assets.

   Determine which elements of the application require protection.

2. Create an architecture overview.

   Create diagrams (e.g. UML diagrams) and description of the used technologies.

3. Decompose the application.

   Identify the entry and exit points, trust boundaries, data flow descriptions, privileges (user and application), etc.

4. Identify the threats.

   Describe possible ways to compromise the application. It is recommended to use various tools to describe possible attacks, e.g. attack trees, vulnerability databases and the STRIDE model [STRIDE].

   STRIDE is an approach that helps to classify computer software threats. STRIDE stands for:

   - Spoofing identity: An attack method where adversaries falsely represent themselves as valid users. For example, the attacker obtains system administrator credentials, logs in as system administrator, gains access to system data and executes further attacks.
   - Tampering with data: An attack method where an adversary alters or deletes data within the system. For example, an adversary gains access to the system database and alters records that contain product prices.
   - Repudiation: An attack method where an adversary attacks a system without detection or evidence that the attack occurred. An example would be an adversary who performs a ‘tampering with data’ attack without leaving any trail indicating that the data had been compromised.
   - Information disclosure: An attack method where an adversary gains access to data outside their trust level. Such data may include system information that may facilitate further attacks.
   - Denial of service: An attack method where an adversary causes a system to be unavailable for valid user entities. For example, an adversary who executes a shutdown command to a file server.
   - Elevation of privilege: An attack method where the adversary's system trust level is raised, permitting additional attacks. For example, an adversary who enters a system as an anonymous user entity but is able to obtain the trust level of a system administrator.

5. Document the threats.

   Create threat documentation.

6. Rate the threats.

   Evaluate all threats in terms of damage potential, reproducibility, exploitability, affected users and discoverability.

For more information on threat modelling, see [THREAT].

5.10.3 Best Practices and Recommendations for Security-Related Coding

The following practices are recommended for security-related coding:

- Compile applications using safety switches.

  A number of compilers supplies developers with mechanisms that help avoid potential security vulnerabilities (e.g. Visual C++ compiler's '/GS' switch reduces the risk of buffer overruns in the application). It is recommended that you check whether compilers used in the project contain any mechanisms for improving security, and that you use them when you build applications for a production environment.

- Sanitise all input.

  Many attacks (XSS, SQL and command injections, buffer overflows) are based on improper input handling. It is recommended that you regard all incoming data as potentially dangerous to the function. Data from trusted sources (e.g. database) may have been forged (e.g. due to a successful attack on the database), and should therefore also be handled as insecure data.

  If input is expected to be a number (integer, double, etc.), you should check that:

  - It is in numeric format.
  - Its value is within the proper range.
  - Its value is within the expected range.

  If input is expected to be an SQL query parameter, a file name, etc., you should check that:

  - Its length is within the expected range.
  - It does not contain any forbidden characters.
  - Its value is within the expected range.
  - Prepared SQL statements are used.

  If input is expected to be a file, you should check that:

  - The file exists.
  - It is of the expected type and size.
  - It does not contain any hazardous data (e.g. pdf files containing malicious code).

If any check fails, the input validation process should be aborted, the validation process' status should be logged and an error message should be presented to the user.

It is recommended that you design each function to be immune to improper input (e.g. null values, improper data-type values, etc.).

- Handle array size.

  Some security vulnerabilities (e.g. buffer overflow, information leaks) occur when a function tries to read from or write to outside bounds of an array. It is recommended to check an array's boundaries before executing reading from or writing to a given cell in the array.

  Some programming languages support array size checking (e.g. Java). If array size checking is not supported by a programming language used in the development process, you should declare the array's size as a constant value. This value should be used to determine whether the given cell's address is within boundaries of the array.

- Use safe data structures.

  Programming languages often offer alternative data structures (e.g. Java – storing integer values in integer variables or Integer class objects, C++ – storing chains of chars in C-style strings, C-arrays, std::strings). Some of the data structures offer additional functions for avoiding security vulnerabilities. The Java Integer class, for example, has methods that determine the maximum allowed value of a variable (this prevents data overflow, if used properly), convert to other data types, etc.

- Use tested cryptographic mechanisms.

  According to OWASP, attempting to create non-standard and non-tested algorithms, using weak algorithms, or applying algorithms incorrectly will pose a high risk to data that is meant to be secure. For more information on pitfalls in cryptography, see [CRYPT].

- Handle errors properly.

  If a function can fail, it should be able to report errors. A detailed error report should be written to the application log file.

  All errors should be handled in such a way that the application remains stable. Errors that cause the application to crash, restart or abort are a potential Denial of Service issue. If a programming language supports error handling (e.g. Java try-catch-finally blocks) always handle the error (e.g. never leave a 'catch' block empty and always use free resources in the 'finally' block). Errors should not be visible to users. Instead an error message should inform the user that the application did not complete its task. Error reports, such as Java error stack traces, may help potential attackers to gain more knowledge about applications; therefore they should never be visible to end users.

- Encrypt sensitive data that is stored.

  Sensitive data that is stored (e.g. user passwords stored in a database) should be encrypted. Data stored in plain text is available to the attacker as soon as they gain control of the storage.

- Encrypt sensitive data that is transmitted.

  If any sensitive data is transmitted and there is a chance of eavesdropping on the communication, all communication should be encrypted. It is recommended to implement SSL/TLS sockets to provide secure communication.

- Use unpredictable identifier values.

  Predictable identifier values lead to a number of security breaches (e.g. information disclosures). If an adversary can predict the session identifier value, user profile id value, etc., they may forge credentials stored on their computer and be able to gain access to restricted data (they could, for example, forge a cookie file to gain access to another user's profile information).

  It is recommended to generate identifier values using a strong random algorithm. A web application's session identifier values should be valid for a limited period of time and bound to the user's IP address.

- Synchronise access to shared resources.

  Shared resources (e.g. variables, files) are a potential source of security vulnerabilities (e.g. race conditions [RACE]). If operations on a resource are not atomic, the application may be vulnerable to 'Time-of-check-to-time-of-use' bugs [TOCTTOU].

  To provide exclusive access to shared resources within one application, all operations on those resources should be atomic. An availability check on the resource and use of the resource should only be performed if it does not cause disruption.

- Log all user actions.

  To ensure accountability, it is recommended to write all user actions to a log file. Detailed logs also help finding security breaches (e.g. an unprivileged user modifying restricted data) and fixing malfunctions in applications (it is possible to track a user's actions that led to a malfunction).

- Always check the integrity of loaded libraries.

  If an application uses external libraries, it is recommended that you check their integrity before they are loaded. It is possible to replace a library's content with malicious code and attack the system with the content of the forged library. The best solution is to use signed libraries. If using signed libraries is not possible, it is recommended that you compare checksums of the libraries with the expected values.

- Do not use insecure technologies.

  Many attacks are not based on application vulnerabilities, but on vulnerabilities of the technologies used in the project (e.g. exploiting security vulnerabilities of the .NET 1.1 framework). It is recommended that you thoroughly check the security of technologies that are used in the application before using them in a production environment.

- Use the principle of least privilege.

  User, process or application should be given no more privilege than necessary to perform the given task. Privileges should be revoked as soon as the task is completed.

- Perform cyclic checks of application integrity.

  It is advised to perform cyclic integrity checks of running application. Checks can be automated (e.g. in Linux systems using Cron and TripWire). If an integrity check fails, it proves that the integrity of the application has been compromised and the application is no longer secure.

- Test for security.

  Details are provided in *Security-Oriented Testing* [QA].

6 Preserving and Transferring Knowledge and Skills

6.1 Knowledge Capture and Curation

The improvement and capture of software team members' knowledge about the developed system and related technologies should be achieved by organising the software engineering process in a way which facilitates communication and exchange between developers, allows for implicit/transparent/organic transfer of knowledge between colleagues, and at the same, time stimulates information and knowledge gathering and codification by the means of development support tools. The collected information must be searchable and traceable through time, so that it could be retrieved when needed and easily verified for being relevant and up to date. In order to achieve this, the standard tools that used in software development and support (starting from version control, via issue and build tracking, to publishing and deployment of artifacts) must be selected and used so that relevant data is captured and accessible.

The established internal policies and conventions, as well as examples of lead developers, should encourage all developers to enter the relevant information into available supporting systems, and keep it up to date. In fact, all notes associated with tasks, issues, code commits, builds, and configuration changes should be carefully and clearly written, e.g. on the internal developers' workspace of the GN3 Confluence Wiki (see *GN3 Confluence Wiki* on page 33).

Furthermore, the available internal and public documentation and knowledge base accumulated within software engineering tools should be able to adequately serve as reference material for new project members or maintainers, imparting to them enough information and understanding about the project implementation and design and implementation decisions and their known ramifications, so that they are able to understand what is being expected from them, how it relates with already present code features, and how to deal with the source code.

Finally, periodic intentional and controlled switching of assignments and roles of developers within the teams should be exercised. This can help to spread knowledge and widen expertise within the teams, as well as help to avoid boredom. It is important to organise these changes in a way which relates the new assignment of a person with something s/he is already partially familiar with, but also gives the access to the key information and other persons familiar with the non-overlapping area. By doing this, the developers will be encouraged to extend their knowledge of a greater part of the project than they are involved in. This, in turn, allows greater flexibility in the deployment of resources, and can help to alleviate the difficulties of knowledge and skills infill which may arise when a developer leaves mid-project.

6.2 Preparing New Developers

When new developers are hired, they should be informed about the organisation's mission, structure, rules, conventions and best practices. They must be also educated about the context of the project and work that has preceded them. The team leader (or assigned mentor) should make sure that the introduction is conducted in the right pace: not too slow (to avoid boredom and time wasting) and not too fast (to avoid newcomer's confusion and misunderstanding of the core principles).The new employee needs to develop knowledge and skills in within the areas of:

- Team culture, conventions and procedures.
- Problem domain or business process.
- Technologies s/he will work with.
- Relevant elements of the existing implementation s/he will use or develop further.
- Areas of expertise of other team members in the code base.

Introducing new developers late into the project's development should be avoided, as it may only result in increased costs and decreased productivity of the team [Brooks's law]. At the point when the delays become apparent (for instance, if the project is already late), there may be no alternative but to either accept the delay and arrange a relaxed schedule, or reduce the scope of the delivery. Of course, the cost of additional training and increased internal communication may be countered by adding people to the project earlier (while the code base is smaller and expertise less profiled), or by adding only very good programmers or domain experts, who may invigorate the team without too much overhead.

If the newcomer is to join a project that is already running, s/he should be integrated as soon as possible. One of the best ways to do this is to use a top-down approach, where the person is being familiarised with the generalities of the developed system and gradually shifts towards details. The exemplary sequence of steps the introduction process can be as follows:

1. Key characteristics of the software – what is it and what is it not supposed to do.
2. External and user interfaces (black box view).
3. User stories and / or requirements.
4. Issue reporting and handling guidelines, timeframe and procedures.
5. Responsibilities associated with roles and their mapping onto team members.
6. User acceptance tests – the person may also work as tester for some time.
7. Significant known issues.
8. Internal core principles of the software (architecture, frameworks, development tools, coding conventions).
9. Internal structure of the software (modules and dependencies between them).
10. Implementation – algorithms, utilities, code and release management procedures and policies, etc.

It is important to provide the newcomer with the assistance of a mentor who will answer his or her questions or address these to other and more experienced members of the development team. Once the new member is introduced to the team he or she can work on the tasks resulting from the approved development plan. It is difficult to provide any time frame for the introduction process, since it is directly related to the newcomer's background, project size and maturity, and the availability and quality of documentation.

The novice's questions that emerge during this process may also be seen as an opportunity to improve the documentation and knowledge base. The tutoring and assistance may have significant impact on colleagues who act as tutors, since they gain importance in the process and became more aware of their own skills and significance of their work. Tutoring is also an opportunity for developers to systematise their own knowledge and review and (re)apply the practices preached to the newcomer. This internal evaluation allows tutors to realign to the norm, but also protects from proliferation of bad habits.

6.3 Knowledge Update Briefings

Increasing knowledge and improving skills is crucial and should become an integral part of project. Generally speaking, training briefing attendees gain not only knowledge related to new technologies and methodologies ('hard skills'), but are able to exchange their experience on high level, that goes beyond issues related to day-to-day work. One cannot underestimate 'soft skills' that are gained in trainings. Interpersonal relationships developed during these educational events have a great impact on team spirit and cooperation between teams' and groups' members. There, the attendees can find inspiring examples from other projects and role models, and also learn about analogous or comparable experiences.

For the purpose of this document we decided to distinguish two types of briefings:

1. Seminars (performed weekly or once per fortnight).
2. Working-day meetings (performed once a month).

Although these trainings greatly differ (from both organisational and methodological point of view), they complement each other. The following table summarises and compares all aforementioned briefing types.

	Seminar	Working-day meeting
Duration	1–2 hours	½ day
Frequency	Once a week	Once a month
Number of attendees	~ 5 (team)	~ 15 (couple of teams)
Cost	No cost at all	Low cost (coach, lunch)
Expected result	Update on current issues that co-employees are working on. Refresh of the skills and knowledge related to those current issues.	General theoretical knowledge and some practice skills related to new framework, methodology or a tool.

Seminars are short meetings performed once a week in a common space at the workplace. During these meetings, team members present some new techniques they recently familiarised with, or achievements and challenges, related to their current work. Although the meeting is usually conducted without agenda, attendees should agree on the form of discussion and questions and answers session (after presentation or by interrupting the presenter). The meeting should be completed within two hours. Using laptops and smart phones is not allowed in order to focus attendees' attention on the main topic of the meeting. The presenter should prepare the presentation and accompanying materials, which usually requires at least one day of preparatory work.

Working-day-long meetings are usually performed once a month. Dependently on the purpose of the meeting it can be held in a common space at the working place or outside the company and can be led by member of the team or external trainer. In order to make sure that all training's topics are covered and to avoid unwanted delays, a formal agenda with expected timings should be defined and followed. This type of training can consist of theoretical and practical sessions where attendees complete hands-on exercises. One working day is a convenient period to provide wide range of material, for example:

- Development framework.
- Development methodology.
- Management procedures.
- Installation, configuration and use of new tool.
- Assumptions and utilisation of new communication channel.
- New workflow.

The meeting, including wrap-up, should be completed within a working day. Lunch or refreshment should be provided during the meeting. Using laptops and smart phones is not allowed (excluding breaks) in order to focus attendees' attention on the main topic of the meeting.

References

[AAI]	https://forge.geant.net/forge/display/AAI/Documentation
[ACL]	http://commons.apache.org/logging/
[ANTRUN]	http://maven.apache.org/plugins/maven-antrun-plugin/
[ARTIFACTORY]	http://www.jfrog.org/products.php
[Brooks's law]	http://en.wikipedia.org/wiki/Brooks's_law
[C]	Ganssle Group's Firmware Development Standard: http://www.ganssle.com/fsm.pdf Netrino Embedded C Coding Standard: http://www.netrino.com/Coding-Standard Micrium C Coding Standard: http://micrium.com/newmicrium/uploads/file/appnotes/an2000.pdf
[C++]	http://en.wikibooks.org/wiki/C%2B%2B_Programming/Code_Style http://www.state-machine.com/doc/AN_QL_Coding_Standard.pdf http://geosoft.no/development/cppstyle.html http://google-styleguide.googlecode.com/svn/trunk/cppguide.xml
[CI]	http://en.wikipedia.org/wiki/Continuous_Integration
[CIMF]	http://martinfowler.com/articles/continuousIntegration.html
[CMP-NEX-AR]	http://stackoverflow.com/questions/364775/should-we-use-nexus-or-artifactory-for-a-maven-repo
[CODECONV]	http://java.sun.com/docs/codeconv/ http://www.ambysoft.com/downloads/javaCodingStandards.pdf http://www.python.org/dev/peps/pep-0008/
[COOR]	https://intranet.geant.net/sites/Services/SA4/T3/pages/home.aspx#Coordinators
[CRITIC]	http://articles.qos.ch/thinkAgain.html
[Crowd]	https://devel.geant.net/crowd
[CRYPT]	Schneier B., Security Pitfalls in Cryptography: http://www.schneier.com/essay-028.html
[DBD]	http://www.fabforce.net/dbdesigner4/
[DEB]	http://www.debian.org/doc/debian-policy/ch-controlfields.html#s-f-Package
[DEB1]	http://ubuntuforums.org/showthread.php?t=51003
[DEB2]	http://www.debian.org/doc/manuals/maint-guide/index.en.html
[DEPL]	http://maven.apache.org/guides/getting-started/index.html#How_do_I_deploy_my_jar_in_my_remote_repository
[DOCBP]	The most recent version of the GN3 Document Best Practice Guide can be accessed from the GÉANT Intranet at: https://intranet.geant.net/sites/Services/SA4/T1/Documents/Forms/AllItems.aspx

[DS4.3.1]	W. Zurowski, G. Kramer, "DS4.3.1 Specification of Software Development Infrastructure" https://intranet.geant.net/sites/Services/SA4/Deliverables/DS431/Documents/GN3-09-099-DS4-3-1_Specification_of_Software_Development_Infrastructure.pdf
[EAR]	http://java.sun.com/j2ee/verified/packaging.html
[EE]	http://www.oracle.com/technology/pub/articles/dev2arch/2008/01/packaging-best-practices.html
[EFFECTIV]	http://www.martinfowler.com/ap2/effectivity.html
[EXCEPT]	http://today.java.net/pub/a/today/2006/04/06/exception-handling-antipatterns.html
[FISH]	https://svn.geant.net/fisheye
[Forge]	https://forge.geant.net/forge/display/cNIS/Home
[GN3-09-040]	http://wiki.geant.net/pub/NA1/DocumentsMain/GN3-09-040_Writing_Process_for_GN3.pdf
[GNU]	http://www.gnu.org/software/gettext
[GRH]	http://www.atlassian.com/software/greenhopper/
[INTDOC]	http://docbook.org/
	http://www.xmlmind.com/xmleditor/
	http://en.wikipedia.org/wiki/Comparison_of_documentation_generators
	http://www.stack.nl/~dimitri/doxygen/
	http://www.xs4all.nl/~rfsber/Robo/robodoc.html
	http://www.naturaldocs.org/
[JAVADOC]	http://java.sun.com/j2se/javadoc/writingdoccomments/
[JAVAGEN]	http://geosoft.no/development/javastyle.html#General%20Recommendations
	Bloch J., Effective Java: Programming Language Guide, Addison-Wesley, 2001
[JAVASIG]	http://java.sun.com/docs/books/tutorial/security/toolsign/index.html
[JBUGS]	http://java.sun.com/javase/6/webnotes/6u18.html
	http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6921609
[JCC]	http://java.sun.com/docs/codeconv/
[JCC311]	http://java.sun.com/docs/codeconv/html/CodeConventions.doc2.html#3441
[JCC312]	http://java.sun.com/docs/codeconv/html/CodeConventions.doc2.html#277
[JCC4]	http://java.sun.com/docs/codeconv/html/CodeConventions.doc3.html#262
[JCC514]	http://java.sun.com/docs/codeconv/html/CodeConventions.doc4.html#286
[JCC10]	http://java.sun.com/docs/codeconv/html/CodeConventions.doc9.html#529
[JUL]	http://java.sun.com/javase/6/docs/technotes/guides/logging/
[JWSbugs]	http://scalemania.wordpress.com/2010/01/14/when-will-java-web-start-be-production-quality/
[JWSdesktop]	http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=7063209
[KR]	Kernighan, B.W.; D.M. Ritchie, The C Programming Language (1st ed.), Prentice Hall, 1978
[LOCAL]	http://en.wikipedia.org/wiki/Software_localization
[LOGB]	http://logback.qos.ch/
[Log4J]	http://logging.apache.org/log4j/
[LogInf]	http://www.ibm.com/developerworks/java/library/j-logging/
[MAVEN]	http://maven.apache.org/
	http://maven.apache.org/plugins/index.html
	http://mojo.codehaus.org/plugins.html
[NEXUS]	http://nexus.sonatype.org/
[OWASP]	http://searchsoftwarequality.techtarget.com/sDefinition/0,,sid92_gci1192885,00.html
[PARCH]	http://www.sqlpower.ca/page/architect
[PCMAG]	http://www.pcmag.com/article2/0,2817,11525,00.asp
[PERF]	http://www.perforce.com/perforce/papers/bestpractices.html

[POM]	http://maven.apache.org/pom.html
[Ports1]	http://undeadly.org/cgi?action=article&sid=20080318060000
[Ports2]	http://www.freebsd.org/doc/en_US.ISO8859-1/books/porters-handbook/index.html
[PSEUDOLOC]	http://en.wikipedia.org/wiki/Pseudolocalization
[QA]	The most recent version of the GN3 Quality Assurance Guide can be accessed from the GÉANT Intranet at: https://intranet.geant.net/sites/Services/SA4/T1/Documents/Forms/AllItems.aspx
[RACE]	Bishop M., Dilger M., Checking for Race Conditions in File Accesses: http://nob.cs.ucdavis.edu/bishop/papers/1996-compsys/racecond.pdf
[REL-MGM]	http://en.wikibooks.org/wiki/Release_Management
[REL-M-PROC]	http://en.wikipedia.org/wiki/Release_Management
[REPO1]	http://maven.geant.net/repo
[REPO2]	scpexe://maven@maven.geant.net/repo
[REPO3]	http://maven.geant.net/snapshots-repo
[REPO4]	scpexe://maven@maven.geant.net/snapshots-repo
[ROLES]	Ferraiolo D., Kuhn R., Role-Based Access Controls, http://hissa.ncsl.nist.gov/rbac/paper/rbac1.html
[RPM]	http://www.rpm.org/max-rpm/ch-rpm-file-format.html
[RPM1]	http://wiki.centos.org/Newsletter/0904#head-1109c83c6d8b7562f4fc54090a607d269e01e115
[RPM2]	http://www.rpm.org/wiki/Docs
[RVN]	http://en.wikipedia.org/wiki/Revision_control
[SC]	GN3 SA2 T4 Security Practices Cookbook: https://intranet.geant.net/sites/Services/SA2/Documents/Forms/AllItems.aspx
[SCserv]	GN3 SA2 T4 Security Expertise Consultancy: https://intranet.geant.net/sites/Services/SA2/Documents/SEC_CONS_EXPL_4_211009_VC.doc
[SD]	Service Desk email address: gn3-swinfra-sd@geant.net
[SD3]	http://blogs.msdn.com/stronglinks/archive/2007/10/24/secure-simply-by-sd3-way.aspx
[SECURITY]	http://httpd.apache.org/docs/1.3/misc/security_tips.html
[SLF4J]	http://www.slf4j.org/
[SRM-BP]	Software Release Management Best Practices http://www.buildmeister.com/articles/software_release_management_best_practices
[SSH]	http://hkn.eecs.berkeley.edu/~dhsu/ssh_public_key_howto.html http://the.earth.li/~sgtatham/putty/0.60/htmldoc/Chapter8.html
[STRAT]	The most recent version of the GN3 Software Architecture Strategy Guide can be accessed from the GÉANT Intranet at: https://intranet.geant.net/sites/Services/SA4/T1/Documents/Forms/AllItems.aspx
[STRIDE]	http://msdn.microsoft.com/en-us/library/ms954176.aspx Hernan S., Lambert S., Ostwald T., Shostack A., Uncover Security Design Flaws Using The STRIDE Approach: http://msdn.microsoft.com/en-us/magazine/cc163519.aspx
[SVN]	http://subversion.tigris.org/ http://en.wikipedia.org/wiki/Subversion_(software)
[SVNUSE]	Version Control with Subversion: Chapter 4. Branching and Merging: http://svnbook.red-bean.com/en/1.5/svn.branchmerge.html

[SVNAUTH]	Version Control with Subversion: Path-Based Authorization: http://svnbook.red-bean.com/en/1.5/svn.serverconfig.pathbasedauthz.html
[SWDSD]	https://issues.geant.net/jira/browse/SWDSD
[TEMPORALDB]	http://en.wikipedia.org/wiki/Temporal_database
[TEMPORALOBJ]	http://www.martinfowler.com/ap2/temporalObject.html
[TEMPORALPROP]	http://www.martinfowler.com/ap2/temporalProperty.html
[THREAT]	Swiderski F., Snyder W., Threat Modeling, Microsoft Press, 2004, ISBN-10: 0735619913, ISBN-13: 978-0735619913
[TOCTTOU]	http://en.wikipedia.org/wiki/Time-of-check-to-time-of-use
[VULNERABILITIES]	Christey S., Unforgivable Vulnerabilities: http://cve.mitre.org/docs/docs-2007/unforgivable.pdf
[WS]	http://www.oracle.com/technetwork/java/javase/overview-137531.html

Glossary

AJAX	Asynchronous JavaScript And XML
AL	Activity Leader
ANSI	American National Standards Institute
APT	Advanced Package Tool
API	Application Programming Interface
BSD	Berkeley Software Distribution
CMDB	Configuration Management Database
CPAN	Comprehensive Perl Archive Network
CSS	Cascading Style Sheets
CVS	Concurrent Versions System
DDL	Data Definition Language
DEB	Debian Package Management
DM	Deploy Manager
EC	European Commission
EER	Enhanced Entity-Relationship
EJB	Enterprise JavaBeans
FAQ	Frequently Asked Question
GUI	Graphical User Interface
HTML	Hyper Text Markup Language
IDE	Integrated Development Environment
IP	Internet Protocol
ISO	International Organization for Standardization
ITIL	Information Technology Infrastructure Library
JAR	JAVA Archive
JAX-RPC	Java API for XML-based RPC (Remote Procedure Call)
JCC	Sun's Java Code Conventions
JDBC	Java Database Connectivity
JDK	Java Development Kit
JFC	Java Foundation Classes
JMS	Java Message Service
JMX	Java Management Extensions
JNDI	Java Naming and Directory Interface
JNLP	Java Network Launching Protocol

JRE	Java Runtime Environment
JSF	JavaServer Faces
K&R	Kernighan and Ritchie
MDC	Mapped Diagnostic Context
MIME	Multipurpose Internet Mail Extensions
MS	Microsoft
NDC	Nested Diagnostic Context
OS	Operating System
OWASP	Open Web Application Security Project
PDF	Portable Document Format
PO	Project Office
PR	Public Relations
RC	Release Candidate
RDBMS	Relational Database Management System
RM	Release Manager
RPM	Redhat Package Management
RTF	Rich Text Format
SA	Service Activity
SCM	Software Configuration Management
SOAP	Simple Object Access Protocol
SQL	Structured Query Language
SSH	Secure Shell
SSL	Secure Sockets Layer
SVN	Subversion
T	Task
TA	Technical Author
TL	Task Leader
TLS	Transport Layer Security
UML	Unified Modeling Language
URL	Uniform Resource Locator
UTC	Coordinated Universal Time
XHTML	Extensible Hypertext Markup Language
XLST	XML Stylesheets
XML	Extensible Markup Language
XSD	XML Schema Definition
XSL	Extensible Stylesheet Language
XSL-FO	Extensible Stylesheet Language Formatting Objects
XSLT	Extensible Stylesheet Language Transformations
XSS	Cross-Site Scripting
VCS	Version Control Management
WAR	Web Application Archives
WSDL	Web Services Description Language
WYSIWYG	What You See Is What You Get