GN3 Quality Assurance Best Practice Guide 4.0

Last updated: 07-09-2012
Activity: SA4
Dissemination Level: PU (Public)
Document Code: GN3-09-184v6

Authors: Branko Marović (AMRES), Marcin Wrzos (PSNC), Marek Lewandowski (PSNC), Andrzej Bobak (PSNC), Tomasz Krysztofiak (PSNC), Rade Martinović (AMRES), Spass Kostov (BREN), Szymon Kupinski (PSNC), Stephan Kraft (DFN), Ann Harding (SWITCH), Gerard Frankowski (PSNC), Ognjen Blagojević (AMRES), Tihana Žuljević (CARnet), Jelena Đurić (AMRES), Paweł Kędziora (PSNC)

GÉANT

Table of Contents

1 Introduction 1

2 Change Management 3

3 Configuration Management 5

4 Testing 7

4.1 Test Levels 9

4.2 Unit Testing 10

4.3 Integration Testing 11

4.4 System Testing 12

4.4.1 Functional Testing 12

4.4.2 Non-Functional Testing 13

4.4.3 Security-Oriented Testing 16

4.5 Principles of Software Testing 17

4.5.1 What to Test For 17

4.5.2 How to Test 18

4.5.3 Testing Improvement from Bug Fixing 20

4.6 Tools Used in Testing 20

4.7 Testing of Web Applications 21

4.7.1 Performance and Stress Testing 21

4.7.2 Automated Functional/Regression Testing 25

4.7.3 Web Services Testing 28

4.7.4 Tools 30

5 Code Analysis 32

5.1 Internal Code Reviews 32

5.1.1 Review Setup 33

5.1.2 Carrying Out a Review 34

5.1.3 Handling Review Results 37

5.2 Automated Code Quality Checking 40

5.2.1 Static Analysis 41

5.2.2 Impact of Automated Code Reviews on Team and Developer 42

5.2.3 Recommended Configurations for Maven and Jenkins 43

5.2.4 Static Code Analysis with Macxim 47

6 Performance Improvement and Optimisation 50
6.1 Code Optimisation 50
6.1.1 Define Goals and Metrics 52
6.1.2 Choose an Appropriate Algorithm 53
6.1.3 Design and Use Efficient Structures 53
6.1.4 Ensure Data Density and Locality 54
6.1.5 Select the Appropriate Array Access Pattern 54
6.1.6 Use Compiler Optimisations 54
6.1.7 Use Profile Feedback 55
6.1.8 Computational Performance Optimisation in Java 55
6.1.9 Code Parallelisation 56
6.2 Relational Database Optimisation 64
6.3 Optimisation of JPA/EJB Usage 68

7 Problem Management 71
7.1 Acceptance Criteria Based Software Evaluation 72

8 Bug Tracking 77
8.1 Bug Reporting 77
8.1.1 Bug Report Principles 78
8.1.2 Preliminary Steps 78
8.1.3 Tips and Guidelines for Completing a Bug Report 78
8.2 Dealing with Bugs 80
8.2.1 Bug Tracking Software 81
8.2.2 Bug Tracking Guidelines 82
8.3 Monitoring the Issue Handling Process 83

9 Using QA Testbed Infrastructure 86
9.1 QA Testbed Procedures 86
9.2 Booking QA Testbed Resources 87
9.3 QA Testbed Documentation 88

10 Closing SW Projects 89
10.2 Project Closeout Reports 91
10.2.1 Project Closeout Transition Checklist 91
10.2.2 Project Closeout Report 92
10.2.3 Post Implementation Report 93
10.3 Documenting Lessons Learned 93

References 95

Glossary 98

Table of Figures

Figure 3.1: Number of active threads over time 22

Figure 3.2: Throughput (black) and average response time (blue) in ms 23

Figure 3.3: Throughput (blue), average response time (green) and errors (red) of system recovering from the intense, but short, workload (black) 23

Figure 3.4: Throughput (blue), average response time (green) and errors (red) of a system handling the constant workload (black) 24

Figure 3.5: Throughput (blue), average response time (green) and errors (red) in a system that cannot cope with the workload (black) 25

Figure 5.1: Example of Jenkins build result page 45

Figure 5.2: Example of CheckStyle warning description on Jenkins 46

Figure 5.3: Example of Jenkins CheckStyle code viewer 46

Figure 5.4: Lock ordering deadlock 62

Figure 7.1: Cumulative created (red) vs resolved (green) issues in JIRA 84

Figure 8.1: QA Testbed resource requesting workflow 87

Table of Tables

Table 6.1: Acceptance thresholds 76

1 Introduction

Software Quality Assurance (QA) controls the software engineering processes and ensures quality. It does this by auditing the quality management system in which the software system is created. Software QA encompasses the entire development process, which includes processes such as:

- Software design.
- Coding.
- Documentation.
- Source code control (revision control).
- gCode reviews.
- Testing.
- Change management.
- Configuration management.
- Release management.

Software QA controls the quality of software products by reviewing code and documents, and by testing software. The main test types are unit, integration, and system tests.

To be able to conduct software QA, the following practices must be in place during all stages of software development (requirements definition, design, review):

- The QA team must work with the application developers to ensure software QA is applied during the requirement definition phase, when the following are identified:
  - Needs.
  - Architecture.
  - Documentation requirements.
  - Software specifications.
- Software best practice recommendations should be applied during the design phase, especially in the iteration transition. Standard tools and procedures should also be used.
- Software QA should be an integrated aspect of software testing.
- A review process should be designed to support code reviews and audit reports should be issued.
- An environment for load testing should be designed.
- Process improvement will include ongoing gathering and evaluation of feedback.

Quality assurance is needed to ensure high standards, consistency and make sure common errors are avoided. Such errors are widely recognised and classified as code smells [SMELL] and anti-patterns [ANTIPATTERN]. Common code smells include: duplicate code, large methods or classes, inappropriate intimacy between classes, duplicate methods and so on. Anti-patterns have much wider scope, covering project organisation, methodology, management, analysis, software design, and programming.

Anti-patterns differ from usual bad habits or practices by appearing to be beneficial and including documented solutions that are well proven and can easily be repeated. The concept of anti-patterns has become so popular that even the most frequent errors specific to some programming languages such as Java [JAVAANTIP]) were codified as anti-patterns. While original design patterns [PATTERN] are relevant during design and coding, anti-patterns can be used to prevent bad practices in software development and help detect and address problematic code during code reviews.

Quality assurance tasks are often viewed as boring and time consuming by developers, who do not always see the direct benefits of following good practices, which are mostly visible in long-term project maintenance. Based on experience gained during the previous GN2 project, the project's managers set up a task responsible for a compliance audit of best-practice take-up. Task 2 of Activity SA4 provides reports for the management on an annual basis, to provide an overview of what has been done in each project with regards to software quality. The report will contain results of periodic reviews, recommendations and observed improvement or decline of quality of software development process.

It is not possible for the QA team to deeply understand and review each project. Therefore, the evaluation of the quality of projects will be based on information provided by each of the development teams. SA4 Task 2 reviews are based on prepared documentation and quality reports, therefore, the QA team uses tools to support any software quality assurance tasks.

In some cases, developers may not understand some of the quality processes or may not know how to implement them in their project. Therefore, SA4 T2 offers support for every team interested in putting them into practice.

2 Change Management

Change management is a process of requesting, defining, planning, implementing and evaluating changes to a system. The ITIL (Information Technology Infrastructure Library) defines this process as the standardisation of methods and procedures that are used for efficient handling of changes. A well-defined and comprehensive change policy provides safe and effective introduction and implementation of new or modified requirements. Such a policy should specify the following change process activities:

- Change request creation.
- Technical review.
- Assessment of importance and effort of change.
- Authorisation.
- Implementation.
- Testing.
- Releasing and deployment.
- Change review and closing.
- Remediation – handling of failed changes.

Requests for changes are submitted in writing and entered into a requests or issue-tracking system. They can be grouped into several general categories:

- Changes related to the project roadmap.
- New requirements requested by the client.
- Reported bugs.
- Comments from support.
- Developer proposals.
- Minor features.

If the impact of a change on usage or operation is not obvious, it should also be described. Feature requests should not be added or features changed without review and approval by both development team and users.

The technical review discusses the aspects and consequences of requested changes, including an analysis of related errors' root causes and an implementation proposal. Dependencies with other features, including user and operational documentation, training materials and the operating environment also need to be described.

During the assessment and evaluation of the change's importance and effort needed, the lead developer decides whether the request is accepted or rejected based on user input and the technical review. In case of the latter, the reason for rejection should be explained. If the request is accepted, appropriate priority should be assigned. The more important the issue, the faster the solution must be delivered. However, the cost of the change also has to be evaluated. For a more detailed discussion of defect severities and prioritisation is provided, see *Acceptance Criteria Based Software Evaluation* on page 72. Change feasibility and its impact on cost and schedule can be expressed by the priority given to individual requests. This represents overall importance and how much effort and resources dealing with the request would require.

Cost depends not only on the immediate effects of the issue and its anticipated solution, but also on the number of changes in dependent modules. A change does not only affect the cost and time of the development, but often also extends the system’s functionality, which can affect expected milestones, demo scenarios and deliverables. If new functionality has been requested, the potential gain should be calculated. If the gain does not balance the cost and the change significance is minor or trivial, it can be rejected. If a major change request is accepted, the implementation plan should be reviewed.

Change authorisation includes a dependency check, planning, scheduling, grouping of changes and defining of timeframe. Although the person who is operationally responsible for this is the project or release manager, it is advisable to establish a change management board or Change Advisory Board (ITIL: CAB) that comprises all key stakeholders responsible for authorising major changes.

The implementation and delivery of changes is strictly related to release and deployment management. Flexible scheduling of the changes is possible, if a project is managed using an agile methodology. Not only do these methodologies allow for the immediate scheduling of requested changes, they also enable their smooth delivery, verification and testing. OpenUP, one of the most popular agile methodologies, assumes frequent (e.g. monthly) releases. A new wave of change requests usually occurs after a version of the software is released. Again, these requests are analysed, and an appropriate priority assigned. Issues that are marked as critical should be solved as soon as possible, as part of software release maintenance. Other requests can be assigned to later releases. Any time an issue is closed (and a version with the implemented solution is released), the requester should test the resolution and verify if expectations have been met.

At the highest operational level, change management is expressed through creating, maintaining, updating, and communicating the project roadmap. It should contain milestones defining future versions, along with orientation dates and planned key features. The major changes should be placed on a larger timescale, while the more detailed content only needs to be specified for the next few releases. The moment reserved in the release cycle schedule for updating the revision plan is the most appropriate moment to update the project roadmap. Therefore, these two live documents may be merged into one.

The above described normal change management process should be used in common situations and for the most of changes. However, an alternative procedure can be also preauthorised and applied to emergency changes, which may be caused by some issues, most likely some of those marked as critical.

To be effective, change management requires comprehensive configuration management, which allows easy tracking and accessing of changes. Its effectiveness can be evaluated using Key Performance Indicators (KPIs) associated with ITIL Service Transition [KPI].

3 Configuration Management

According to the IEEE Std-729-1983 glossary, Configuration Management (CM), is the process of identifying and defining the items in the system, controlling the change of these items throughout their life cycle, recording and reporting the status of items and change requests, and verifying the completeness and correctness of items (see also IEEE Std-610.12-1990 [IEEE610]. According to ITIL, configuration management is the discipline of identifying, tracking, and controlling the various components of the IT environment in order to enable decision making, and it is implemented through the deployment of a Configuration Management System (CMS).

Software Configuration Management (SCM) is the aspect of CM that is primarily relevant for software developers, but should also track hardware and the software environment of software configuration items. More specifically, it is a set of procedures for tracking and documenting software throughout its lifecycle to ensure that all changes are recorded and that the current state of the software is known and reproducible. There are also some related standards, such as the IEEE 828 Standard (2005) for Software Configuration Management Plans, CMMI (Capability Maturity Model Integration), and others.

It is important to stay in control of configuration items and the ways they change. Manual configuration management often proves to be problematic – it is better to use technology to help discover, record, and maintain configuration information. Implementing configuration management enables change, incident, problem, and release management. Configuration management is not the most obvious or easiest function to implement in controlling the software development and maintenance environment. However, its integrating perspective is a very powerful tool for ensuring proper control of available resources. It allows understanding the resources and how they relate to one another, thus permitting informed decisions to be made. It can be implemented by deploying and using a comprehensive CMS, but may also start with simple tracking of configuration changes in a versioned document, that is then gradually replaced with a coordinated set of tools controlling changes to different elements of the IT environment. It is also good practice to maintain a change log section in functional specifications and other important documents.

SCM is more specific, as it targets the management of software development projects in which multiple developers work on the same code simultaneously, targeting multiple platforms and supporting multiple software versions. It deals with the management of complex software systems that integrate separate components (such as databases, application servers, and participating services). It also deals with software-related assets and infrastructure, which includes configuring software instances at distinct locations, and is, therefore, closely related to change and release management. Elements tracked through SCM include: code versions, internals documentation and manuals, plans, schedules, state of hardware, network, and software settings.

A comprehensive CMS must enable the team to reproduce a specific instance or its earlier configuration when needed. This provides baseline data for reviewing proposed changes and making effective change management decisions.

SCM is implemented by combining tools and techniques that control the software development process, including software configuration and usage. These tools address the problems of maintenance and simultaneous updates of shared data, which are related to the concurrent work of developers on the same codebase, parallel actions made by infrastructure and service administrators, and release and deployment processes.

While SCM specifically deals with the propagation of changes in common code (for example, libraries), including bug fixes and configuration changes to all affected supported versions of the software, other closely related configuration management activities deal with the management of the local assets used by the software. This includes various aspects of the execution environment, such as: local software configuration, hardware, installed operating system, middleware, security and network infrastructure and their configurations. Where the dividing line for the software development team's responsibilities is drawn depends on the role in the operation of the service provided by the software. If the software is not productised, if few of its instances are in use, or if the customers do not want to take control of the environment in which the software is running, it is highly likely that a software development team has to include the latter aspects into its CMS.

In order to deal with these problems, the CMS used should provide sufficient control and information for all elements of software configuration (version, functional status, dependencies and other relationships) by supporting software and configuration changes, tracking changes and notifying all affected by these changes. SCM therefore revolves around four key aspects [KEY]:

- Version control.
- Build management.
- Release management.
- Process control.

The first three are aspects of process control and described in the Software Developer Guide [SDG]. This guide maps the processes defined for the software development team to their actual implementation with CMS tools.

A more detailed overview of the SCM process, related activities and tasks can be found in ITIL literature on Service Asset and Configuration Management [ITIL Service Transition] and at [CMJournal].

There are also various SCM best practices [SCMBP]. However, all of them are already present in corresponding [SDG] recommendations. Two key recommendations related to all aspects of SCM are to:

- Name an owner for each process, policy, document, resource, product, component, package, branch, and task.

- Describe implemented policies and procedures in 'live' documents that are readily available and subject to continuous update, as required.

GÉANT

4 Testing

The purpose of testing is to verify that major design items meet specific functional, performance and reliability requirements. A primary purpose for software testing is to detect software failures, so that defects may be uncovered and corrected. However, testing cannot establish if a product functions properly under all conditions, it can only establish that it does not function properly under specific conditions.

Software testing often includes examining and executing code in different environments and conditions. Different aspects of the code are also examined to ascertain if it does what it is supposed to do and what it needs to do. Organisation of testing may be separated from the development team in order to assure independent quality checks, but developers should be included in some parts of software testing. There are various roles for testing team members. Information derived from software testing may be used to correct the process by which software is developed.

Software testing methods are traditionally divided into 'black box' testing and 'white box' testing, with some additional approaches that are in between the two. These approaches are used to describe the point of view taken by a test engineer when designing test cases.

- **Black box testing** treats the software as a black box – without any knowledge of internal implementation. Black box testing methods include:
  - Equivalence partitioning.
  - Boundary value analysis.
  - All-pairs testing.
  - Fuzz testing.
  - Model-based testing.
  - Traceability matrix.
  - Exploratory testing.

  Specification-based testing aims to test the functionality of software according to the applicable requirements. The tester inputs data to the test object, but only sees the test object's data output. This level of testing usually requires thorough test cases to be provided to the tester, who can then simply verify that for a given input, the output value (or behaviour), is or is not the same as the expected value specified in the test case. The test items should be developed from the assertion in reference documentation (for example, requirements, specifications and user documentation), tester's vision of the concepts behind and also pinpoint possible ambiguities or documentation contradictions. The traceability matrix should be used to tie assertions to actual tests.

- **White box testing** is when the tester has access to the internal data structures and algorithms including the code that implements them. The following types of white box testing exist:
    - API testing – testing of the application using public and private APIs.
    - Code coverage – creating tests to satisfy some criteria of code coverage (for example, the test designer can create tests to cause all statements in the program to be executed at least once). Execution of these tests should also verify that all covered code paths behave properly.
    - Fault injection methods – introducing faults, in order to test fault/error handling code paths.
    - Mutation testing methods – intentional adding of bugs in order to evaluate the ability of the test cases to detect these bugs.
    - Static testing – includes all static analyses and reviews, inspections and walkthroughs.
    - Code completeness evaluation – checking if all necessary components are implemented and in included, thus verifying that all stubs are replaced by fully functional code.

  White box testing methods can also be used to evaluate the completeness of a test suite created with black box testing methods. This allows the software team to examine parts of a system that are rarely tested, and ensures that the most important parts of code have been tested. Two common forms of code coverage are:

    - Function coverage, which reports on executed functions.
    - Statement coverage, which reports on the number of lines that were executed to complete the test.

  Both function coverage and statement coverage return code coverage metric, measured as a percentage.

- **Grey box testing** involves access to internal data structures and algorithms to design the test cases, by testing at the user, or black-box level. Manipulating input data and usage of output do not qualify as grey box tests, because the input and output are clearly outside of the black box, which is a part of the system being tested. This often happens during integration testing between two modules of code written by two different developers, where only the interfaces are exposed for test. However, modifying a data repository does qualify as grey box, as the user would not normally be able to change the data outside the system being tested. Grey box testing may also include reverse engineering to determine, e.g. boundary values or error messages.

Since various authors describe and classify many available aspects and types of testing in a number ways, this chapter, also tries to provide some systematisation of testing that it is aligned with [EGEE].

4.1 Test Levels

Software engineering distinguishes between several types of tests at different levels of granularity:

- Unit testing

  Tests that determine whether an individual unit of the source code (e.g. an individual method) works correctly. The aim of testing while developing is to ensure that the produced code is reliable at the most basic levels. This type of testing, commonly called unit testing, is not concerned with user requirements, but with the expected behaviour of the software: "Is the software doing what it is expected to?" Testing helps developers prove that it does, which is why this type of testing is done by developers, for developers. It is not the same as functional testing, which is done from the user's point of view.

- Integration testing

  These tests determine whether individual modules (or a combination of several modules) work correctly. Integration tests are performed after unit tests and before system tests. This form of testing is based on a 'building block' approach, in which verified assemblages are added to a verified base, which is then used to support the integration testing of further assemblages.

- System testing (commonly incorrectly aliased with 'Functional testing')

  The purpose of system testing is to determine if the system meets its specified requirements and is therefore ready for release. Unit and integration tests cannot find all possible types of bug, so it is necessary to also test the system as a whole.

It should be noted that there is no clear consensus on the nomenclature of testing. Within the developer and testing community, the distinction between system and functional testing is often unclear, since system testing should, in general, examine achieved functionality. System tests are performed on a fully integrated system, so they are suitable for implementation through functional testing. Functional testing, however, can be performed at any level, for example, to check interpretation and fulfilment of interfaces in integration tests, or overall behaviour of some code segment in unit tests. Functional tests simply check if a feature at a given level is working.

Functional testing can be categorised as a form of black box testing, which means that the tester does not have to know anything about the internal structure or logic of the system that is being examined. Although these tests do not require knowledge of system design or implementation, they may be performed more efficiently with knowledge of inner structure and functioning of the system.

A small additional nomenclature is provided in *System Testing* on page 12, which lists high-level functional tests but also non-functional tests related to performance, load and conformance that are primarily performed at that level. For further information, see [TEST].

4.2 Unit Testing

The aim of testing while developing is to ensure that the code produced is reliable at the most basic levels. As previously stated, unit testing is not concerned with user requirements, but with the expected behaviour of the software.

Unit tests verify if an individual unit of the source code works correctly. All unit tests should be created before the code that is to be tested (test-driven development). While it would be ideal to test all methods (public and private), in actual practice, unit tests of the API are sufficient (according to the Pareto principle 20% of source code causes 80% of errors [PARETO]).

All unit tests should be run after each unit is committed to the repository, so the developer can be sure that his/her modifications do not affect other system modules.

Some recommendations should be followed when developing unit tests:

- Key features, complex implementations and public interfaces must be tested.
- When a bug is found, there should be a test that verifies if it has been solved correctly.
- Trivial methods (e.g. simple setters), do not need to be tested.
- Constant values can be used just for test purposes, as they are easily modifiable and identifiable. However, hardcoded values should not be used when checking test results. Such tests are likely to fail, due to minor code changes. Instead, the code should provide a constant (or method) that could be used in the test.
- Data used for tests should be as detailed as possible (similar to data used in an operational environment).
- The dependence on specific test data in a database should be limited.
- If an existing test fails due to a legitimate change, it should be modified or substituted by a more robust test.
- Tests should be optimised. The more time tests require, the more likely it is that the developer will avoid or interrupt testing.

Unit tests can be created manually or using dedicated solutions that automate the generation of the tests. The resulting tests should check both business logic and user interface.

4.3 Integration Testing

Integration testing is any type of software testing that seeks to uncover collisions of individual software modules with each other. Such integration defects can arise if new modules are developed in separate branches and integrated into the main project.

Integration testing groups and examines how software modules that are usually developed and unit tested separately perform together, to verify that functional, performance and reliability requirements are met. Similarly to system tests, integration tests are a primarily a black box method, which means that the tester does not have to know anything about the internal structure or mechanisms of the tested application.

Integration tests are performed after unit and before system tests. There are three approaches to sequencing of integration tests:

- **Top-down:** The highest level components are tested first.
- **Bottom-up:** The lowest level components are tested first.
- **Big bang:** Multiple modules are tested together. This method is generally similar to system testing. A disadvantage is that potential bugs are found at a very late stage.

The overall strategy is to employ the building block approach, in which verified assemblages are added to a verified base, which is then used to support the integration testing of further assemblages.

Like unit testing, functional tests should be written and incorporated early on in the development process. Frameworks are also available to support the writing of functional test code.

Simulated usage of shared data areas and inter-process communication is tested, and individual subsystems are exercised through their input interface. Test cases are constructed to test that all components within assemblages interact correctly, for example across procedure calls or process activations, and this is done after testing individual modules, i.e. unit testing.

Including integration testing in nightly builds allows incremental handling of problems detected by integration tests. If an integration of several, already-available modules is to be performed, the continuous integration approach described may not be viable. In such a situation, it is suggested to plan and perform incremental integration, in which individual modules would be sequentially added one by one, augmenting by one the size of tested aggregate in each step. The planed sequence should maximise detection and coverage of integration problems detected by tests added in each step, but also minimise the number and complexity of stubs needed in each partial integration.

Within the GN3 context, integration tests could be implemented in a similar way to unit tests. Using the same tools would not only facilitate the development of the tests, but also their execution in an operational environment. The main difference is in scope. Integration tests verify the communication and cooperation level between different modules, while unit tests focus on particular methods and algorithms whose scope is limited to the single module. Developers implementing tests can use the same framework for both unit and integration tests.

For complex and multi-module systems, integration tests can be time and resource consuming. In this case executing all tests on developer workstations seems to be inefficient. The ideal approach should take into consideration continuous integration procedures. Integration tests performed on a continuous integration server can be run after each commit or launched periodically (e.g. once a day). This would provide better software quality, since developers promptly receive information about the influence of new or changed functionality on the whole system.

4.4 System Testing

System testing of software is conducted on a complete, integrated system to evaluate its compliance with its specified requirements [WP]. System testing falls within the scope of black box testing, and as such, does not require insider knowledge of the inner design of the code or logic.

Addressed software requirements can be coarsely classified into functional, non-functional and security-related areas.

4.4.1 Functional Testing

Functional testing aims to test if the functional requirements of software specifications are met. There are numerous procedures that could be applied to test functioning of the software. These procedures should test the system as a whole by using its standard access interfaces, in order to simulate real-life scenarios.

- Installation or implementation testing

  Installation testing focuses on installing and setting up new software successfully. The test process may involve the full or partial upgrade of the software or install/uninstall procedures. Testing should include:

  - Integration of the tools in the environment.
  - Tool interoperability.
  - Tool installation and usage.

  Installation and upgrades of the software usually fall under configuration management.

- Smoke testing or Sanity testing

  Testing of the proposed software release candidate to ensure that it is stable enough to enter more detailed testing. The purpose is to determine whether or not the application is so badly broken that testing functionality in a more detailed way is unnecessary, saving time and resources.

  Many companies run sanity tests and unit tests on an automated build as part of their development process.

- GUI testing

  Graphical User Interface (GUI) software testing is the process of testing a product that uses a graphical user interface, to ensure it meets specifications.

There is no definite or easy way to perform automated tests on software GUI. The techniques are constantly evolving and some of them include using capture/playback systems, algorithms that analyse screenshots of the software GUI, genetic algorithms to decrease the number of test cases, using event-flow graphs or even installing system drivers, so that commands or events can be sent to the software from another program.

- Usability testing

  Usability testing evaluates a product by testing it on users. It focuses on measuring a product's capacity to meet its intended purpose from an end-user’s point of view. The aim of this kind of testing is to observe people using the product to discover errors and areas of improvement. Usability testing usually involves systematic observation under controlled conditions to determine how well people can use the product.

- Portability testing

  Tests determine a product’s portability across system platforms and environments (e.g. different CPU, operating system, or third-party library). Results are expressed in terms of the time required to move the software and complete data conversion and documentation updates.

- Localisation and internationalisation testing

  Localisation testing checks how well a build has been localised into a particular target language. Some of the items of interest in localisation testing include: date formats, measurements, spelling rules, sorting rules, keyboard locales and text filters, besides obvious items such as user interface labels and documentation.

  One of the techniques to test the quality of the software localisation is pseudolocalisation, described in detail in the Software Developer Guide [SDG].

4.4.2 Non-Functional Testing

Non-functional testing refers to aspects of the software that may not be related to a specific function or user action, such as scalability or security. Non-functional testing answers questions such as "How many people can log in at once?" or "How easy is it to hack this software?"

- Performance testing

  Performance testing determines how fast a specific aspect of a system performs under a particular workload. The goal is to evaluate whether the integrated system meets the performance requirements and to identify performance bottlenecks. Some aspects that performance tests can cover are scalability, reliability and resource usage.

  - Volume testing: Volume testing refers to testing a software application with a certain amount of data. In generic terms, this amount can be the size of a database or it could also be the size of an interface file that is the subject of volume testing.

    For example, if the goal is to test the application with a specific database size, the application’s database should be expanded to that size and the application's performance then tested. Another example would be if there is a requirement to interact with an interface file (such as binary file or

XML file); this interaction could be reading or writing on to or from the file. A sample file of the desired size should be created and the application's functionality then be tested with that file.

  - Load and stress testing: Describes the functional testing of a system under various loads to identify at what point the system degrades or fails in a given test environment. For example, testing the performance of the system by simulating multiple users simultaneously accessing it. Stress testing occurs when the load placed on the system is raised beyond normal usage patterns in order to test the system's response at unusually high or peak loads.

- Error handling testing

  This type of testing is usually concerned with run-time errors that take place during the execution of a program and are usually caused by adverse system parameters or invalid input data. An example is the lack of sufficient memory to run an application or a memory conflict with another program. On the Internet, run-time errors can be caused by electrical noise, various forms of malware or an exceptionally heavy demand on a server.

  Testing of handling disastrous environment events, crashes and unreliable infrastructure is related to Resilience testing. Testing the handling of erroneous input is related to Fuzz testing (see below).

- Scalability testing

  Scalability testing is the testing of a software application for measuring its capability to scale up or scale out – be it the user load supported, the number of transactions, the data volume, etc.

  A system whose performance improves after adding hardware proportionally to the capacity added is said to be a scalable system. An algorithm, design, networking protocol, program, or other system is said to scale if it is suitably efficient and practical when applied to large situations (e.g. a large input data set or large number of participating nodes in the case of a distributed system). If the design fails when the quantity increases then it is said that the design does not scale.

- Resilience testing

  Also referred to as stability, recovery, availability or failover testing. Resilience testing tests how well an application is able to recover from crashes, hardware failures and other similar problems.

  For example, one such test is to suddenly restart the computer while an application is running and afterwards check the validity of the application's data integrity. Another example would be to unplug the connecting cable while an application is receiving data from a network and analyse the application's ability to continue receiving data after the network connectivity has been restored.

- Accessibility testing

  For an in-depth look at accessibility and accessibility testing, see [TG].

  Accessibility testing is the technique of making sure that a product is accessibility-compliant by measuring how easily users with disabilities can use a component or system.

  Typical accessibility problems can be classified into visual impairments (such as blindness, low or restricted vision, or colour blindness), motor skills disablements (such as the inability to use a keyboard or mouse, or to make precisely controlled movements), hearing impairments and cognitive ability disablements (such as reading difficulties, or memory loss).

Typical accessibility test cases make sure that all functions are available via the keyboard (without mouse use), that information is visible when the display setting is changed to a high-contrast mode, that screen reading tools can read all the text available and every picture/image have corresponding alternate text associated with it, etc.

- Compatibility testing

  Tests that determine whether the application (system) is compatible with the rest of the computer environment (hardware, network, other software, etc.). These include capability with various processor types, compatibility with different peripherals (printer, DVD drive, etc.), compatibility with other installed software, etc.

  A related type of testing, browser compatibility testing, is performed using different web browsers or different versions of web browser to ensure that users have the same visual experience, irrespective of the browsers through which they view the web application, and the application behaves and responds the consistently across different browsers.

- Conformance testing

  This includes tests that determine whether a system meets a specified standard. Conformance testing is often performed by an external organisation, sometimes the standards body itself, to provide better guarantees of compliance. Products tested in such a manner are then advertised as being certified by that external organisation as complying with the standard. Compilers, for instance, are extensively tested to determine whether they meet the recognised standard for that language.

- Fuzz testing

  Fuzz testing or 'fuzzing' is a software testing technique that inputs invalid, unexpected, or random data to a program. If the program fails (for example, by crashing or failing built-in code assertions), the defects can be noted.

  File formats and network protocols are the most common targets of fuzz testing. Interesting inputs also include environment variables, keyboard and mouse events, and sequences of API calls.

  As a practical matter, developers need to reproduce errors in order to fix them. For this reason, almost all fuzz testing makes a record of the data it produces, usually before applying it to the software, so that if the software fails dramatically, the test data is preserved. If the fuzz stream is pseudo-random number generated, it may be easier to store the seed value to reproduce the fuzz attempt.

  Fuzz testing can demonstrate the presence of bugs in a program. It does not prove that no such bugs exist. It is not a substitute for exhaustive testing or formal methods; however, it can only provide a random sample of the system's behaviour, and in many cases, passing a fuzz test may only demonstrate that a piece of software handles exceptions without crashing, rather than behaving correctly. Thus, fuzz testing can only be regarded as a bug-finding tool, not an assurance of quality.

  A related concept is mutation testing, which involves making minor modifications to a programs' source code or byte code. If the test suite is run and all tests pass, the test suite is deemed lacking because the defect in the code was not found. The purpose of Mutation testing is to help the tester develop effective tests or locate weaknesses in the test data used for the program or in sections of the code that are seldom or never accessed during execution. Fuzzing can be considered as applying the mutation concept to the messages or data exchanged inside communication interfaces.

4.4.3 Security-Oriented Testing

The following paragraphs are sourced from Guideline on Network Security Testing – Recommendations of the National Institute of Standards and Technology [GNST] and Software Security Testing [SST]:

Security testing is a type of testing that determines whether the system protects data and maintains its intended functionality. It is essential for software that processes confidential data to prevent system intrusion by hackers. The primary reason for testing the security of an operational system is to identify potential vulnerabilities and subsequently repair them.

Security evaluation activities include, but are not limited to, confidentiality, integrity, non-repudiation, risk assessment, certification and accreditation (C&A), system audits, and security testing at appropriate periods during a system's life cycle. These activities are geared toward ensuring that the system is being developed and operated in accordance with an organisation's security policy.

There are different types of security testing, for example: network scanning, vulnerability scanning, password cracking, log review, integrity checkers, virus detection, war dialling, war driving and penetration testing. Some testing techniques are predominantly manual, requiring an individual to initiate and conduct the test. Other tests are highly automated and require less human involvement. Regardless of the type of testing, staff that set up and conduct security testing should have significant security and networking knowledge, including significant expertise in the areas of network security, firewalls, intrusion detection systems, operating systems, programming and networking protocols (such as TCP/IP).

Often, several of these testing techniques are combined to gain a more comprehensive assessment of the overall network security. For example, penetration testing usually includes network scanning and vulnerability scanning to identify vulnerable hosts and services that may be targeted for later penetration. Some vulnerability scanners incorporate password cracking. None of these tests by themselves will provide a complete picture of the network or its security.

After running any tests, certain procedures should be followed, including documenting the test results, informing system owners of the results, and ensuring that vulnerabilities are patched or mitigated.

The OWASP (Open Web Application Security Project) is the community project dedicated to improving security of the application software. Their Code Review Guide [OWASP Code Review] and Testing Guide [OWASP Testing Project] cover many of the practices of testing and reviewing code for security issues in great detail.

4.4.3.1 *Automated Tools in Security Testing*

In the field of security testing, more so than with other types of testing, the use of automated tools must be applied carefully, and with measure. Application security assessment software, while useful as a first pass to find obvious anomalies is generally immature and ineffective at in-depth assessments, and at providing adequate test coverage. Manual testing is an absolute must in order to properly assess all or most of the possible security problems and risks. Still, these tools are certainly part of a well-balanced application security program. Used wisely, they can support your overall processes to produce more secure code.

Within security testing, only limited coverage is possible when black box testing or penetration testing, without having entire source code and all the interfaces at hand. The combination of black-box with grey-box techniques is often recommended, where some security problems within the code can be instantly recognised and verified, for instance, using 'magic' numbers (arbitrary values placed in code without explanation or possibility of change through configuration) or leaving 'back-door' access to the application for test purposes.

Some key principles when dealing with security issues are no different from those dealing with functionality and/or load issues of the application, although the impact of security issues can cause significant damage to the users of the application. Testing early and often is always a good way to prevent most bugs from appearing in production. A security bug is no different from a functional or performance-based bug in this regard. A key step in making this possible is to educate the development and QA organisations about common security issues and the ways to detect and prevent them.

New threats arise constantly and developers must be aware of threats that affect the software they are developing. Education in security testing also helps developers acquire the appropriate mindset to test an application from an attacker's perspective. For testers, understanding the subject at hand may seem dull, but in security, attention to detail is essential. That is why the investment in education on security issues will eventually pay off in more secure software.

4.5 Principles of Software Testing

The following points apply to all types of software testing.

4.5.1 What to Test For

- **Correct results**: Check for simple validation of results. If the code ran correctly, this type of test should show it. As an example check: if the code should return the largest number from a list, does it?

- **Cross-check**: Sometimes there is more than one way to solve a problem. One method may be implemented, but another algorithm doing the same thing could be used to cross-check the results of the chosen method. Similarly, when changing how something is implemented, perhaps to improve performance or standards compliance, the old method could be used as a test to verify the new one.

- **Boundary conditions**: It is important to understand the boundary conditions for the tested code, how they are handled, and what happens if they are violated. This will require constant revision as more is learned about how the code is used, but there are some general guidelines:
    - Conformance – Does the input conform to an expected format?
    - Ordering – Is the input in an appropriate order? Is an order expected?
    - Range – Do the values make sense? Are they within reasonable minimum and maximum values?
    - Reference – Does the code reference anything external it does not control?
    - Existence – Does the input value exist?
    - Cardinality – Are there enough values?

- ○ Time – Do things happen when they should and in the right order?

  In each of these cases, it is important to test to see what happens if the conditions are problematic, for example, if the code expects three items as input and instead gets two or four.

- **Error conditions**: This is a difficult, but important, part of testing, and although it is tempting to leave error condition testing until the final product is tested in 'real-world' conditions, there are tests that can be done during development that will make this process easier. As a developer, it is important to think about the type of real-world errors that could impact the unit of code you are working on and write tests to try to simulate them. This could include things like running out of memory/disk space, high system load, network availability errors, etc.

- **Performance characteristics**: This is not testing performance itself, but understanding the performance characteristics of the code and knowing when they may not be stable or scalable. For example, if something proves that it can work with a small sample set how does this perform if the set is a hundred or a thousand times bigger?

- **Insufficient user data filtering**: This is especially associated with security. Basically, all security vulnerabilities are due to insufficient (or a lack of) filtering of the data that are passed to the program by its users. External data not only includes the parameters passed from the command line, the URL or a web form. It also includes all sources of data not defined in the program source code, e.g. environment variables, X.509 certificate contents, configuration files, and even the contents of network packets or responses from services like DNS.

  Testing for insufficient user data operates on a slightly higher level than other tests. The whole working application should be taken into account and, malicious data should be passed onto it to enable inspection of the program's behaviour: Does it crash? Does it stop responding? Does it display unexpected data? As these tests are difficult to automate (their specifics depend on the format and meaning of the data), they should be performed by security specialists. Nonetheless, tests should be carried out for the following general malicious patterns:

  - Very large strings where a short text value is expected.
  - Empty strings where any value is expected.
  - Large numbers.
  - Numbers that could cause an overflow (e.g. MAX_INT + 1).
  - Negative numbers where 0 or a positive number is expected.
  - Strings containing meta characters (especially in web applications).

4.5.2 How to Test

Here are some principles for implementing good tests:

- **Identify defects early**: The sooner the issue is identified, the easier it is to fix and the smaller the costs. For this reason, specification-based tests should be defined as soon as specifications are available. To

purge inconsistencies in requirements, unit tests should be written before the actual implementation, and all tests should be performed as early as possible.

- **Automate**: Testing should be done often throughout the coding process – several times a day, when possible. Automated tests included into night builds are also common practice. This means that the tests should not be a lot of work to run. Manual intervention steps should be avoided, and instead the appropriate inputs simulated in the test code. In addition, code that is checked in, even though it previously passed tests, should have the tests automatically run against it at regular intervals. Doing this manually would be time consuming, so automating the running of the tests, and the determining if the test passed or failed and the reporting on tests is strongly recommended.

- **Be thorough**: Ideally, the test should test everything that is likely to break. It would not be practical to test every line of code and every possible exception but as a minimum, attention should be paid to the areas where the code is most likely to contain bugs – the boundary cases. Free code-coverage tools are available to determine how much of the code has been tested for some environments and could be worth using.

- **Be repeatable**: Developers want to be able to run and rerun tests, and get the same results. This means that testing at this stage should not rely on anything in the external environment that is not under developer control, so that if a problem is detected, it is reasonable certain that it really is a bug. If the code requires an external input, it should be simulated it with a mock object.

- **Be independent**: It is important when testing to keep the environment as simple as possible. Try to test only one thing at a time so that if a test fails the location of the bug should be obvious. The majority of tests that should not rely on other tests or on being run in a particular order – this would simplify the identification of causes and limit dependencies that are hard to trace.

- **Be professional**: The produced tests cases and code are as important as the implementation code, so they should be treated professionally, using best practice to produce them. The test targets should be seen as the means to test the tests. Tests should be sanity checked by deliberately introducing bugs from time to time, and the tests improved as they are used.

- **Be proactive**: If possible, try to perform ad hoc tests. It is recommended that these are run by a person who is not involved in daily development or testing. Do not restrict a tester, as their creativity will result in new test scenarios and cases that will find new bugs or confirm functionality to be working. This approach is called 'exploratory testing'.

- **Be suspicious**: Whenever new functionality is developed or some bugs are fixed, you should perform tests to verify that the new source code does not introduce new, unknown, bugs. Regression testing [REGTEST] ensures that new bugs have not been introduced into already working parts of the software. This is achieved by regular testing of all important features. If any test fails that passed before the new functionality was introduced, it means that regression has been found. Most of the tests needed for pre-existing functionalities should already exist (as these features are established), but new developments should be tested by specifically designed new tests or by modifying existing tests. Regression testing should be as effortless as possible, but must also provide adequate coverage. If two or more tests target the same feature, only the most efficient one should be kept. Quick, indicative tests should be

included in the build procedure and run locally, while the additional verification can be provided by using a continuous integration tool.

- **Rely on the expertise of others**: Include project time for independent security tests of a module. These should be executed by a cooperating security specialist who is experienced in security testing and is unbiased (a person who has implemented something should never be the only tester). After a module reaches its final functionality, it should undergo security tests.

4.5.3 Testing Improvement from Bug Fixing

No method of testing can find all bugs. However, the report of a bug 'in the wild' provides an opportunity to improve the tests – it indicates a gap in the test environment. As more of these holes are patched, fewer bugs should escape into the production code. These bugs may be addressed by performing the following steps:

1. Identify the bug.
2. Open a bug file in the bug tracking tool.
3. Write a test that fails to prove the bug exists.
4. Fix the code.
5. Rerun all tests until they pass.
6. Consider if the same problem could exist elsewhere in the code and set a test trap for it.
7. Close the bug file.

4.6 Tools Used in Testing

Regression testing and extensive functional testing cannot be completed in a reasonable amount of time and effort, with optimal use of resources without help of the tools that automate this process. There are many tools available that adopt different methodologies and provide a vast array of features:

- Testing the web applications in browsers.
- Testing browser emulations without GUI.
- Offering recording of testers' actions in browsers and/or declarative stating what and how should be tested.
- Employing multiple and different versions of various browsers at the same time.
- Offering support for integration of different kinds of server technologies with popular build and CI tools.

The extensive list of open source testing tools can be found at the Open Source Testing site [Open Source Testing].

The automated tools are a perfect match for regression testing, as they will find bugs in code as fast as possible with minimal effort, however, automated functional testing cannot reproduce the logic behind human thinking and evaluate the overall satisfaction of a product. It can and should be used to test repetitive procedures or controls within a web application. The expectations of the test automation should be kept real

and attainable. Tools do not make software bug-free. They help scale the process and help enforce policy. Although the appeal of these tools to find potential issues is great, and running the tools is time efficient, each one of the potential problems takes time to investigate and verify.

The extensive list of open source testing tools can be found here [Open Source Testing].

4.7 Testing of Web Applications

4.7.1 Performance and Stress Testing

Software performance tests challenge performance, stability and reliability of software application under the particular workload. These tests help in finding code or components that need to be improved:

- **Load tests** investigate system behaviour on normal or large, but expected and attainable, number of users. Load testing enables the estimation of response times, throughput rates, resource utilisation, and application and system bottlenecks. Such testing also improves overall configuration and tuning of the system.

- **Stress tests** are performed in order to verify system reliability under unexpected workload and point at synchronization issues, race conditions and leaks of resources as memory, connections, or OS resources. The workload is usually increased until a breaking point is reached.

This section is based on the experience in testing GÉANT AutoBAHN software [AutoBAHN] using Apache JMeter [Apache JMeter] performance testing tool. It is an open source desktop application, primarily used for testing functionalities and performance of web applications. However, the most of the following text is relevant for any other testing tool. If JMeter is not the right testing tool for the target system, a more appropriate alternative may be easily available. A good starting point to find alternatives is [Open Source Testing].Testing starts with selection of a good test scenario (use case) that will generate usable results and point at the issues that should be checked or improved in a tested application. The best candidate scenarios cover performance-critical points of the application or typical usages that will, in the production environment and with real data, generate the most of the load.

After this, the expected workload should be defined (number of users executing the scenario, time between starting of new threads, 'think' time between steps in test scenario, etc.). When the test is run, each user is simulated by a separate thread, using distinct session information. JMeter workload parameters include: the total number of threads, an amount of time to create all threads (ramp-up period), and the number of times needed to execute the test by each thread (the loop count). Each thread generates multiple requests according to the test scenario, waiting for a response after each and 'thinking' after receiving it. Threads may run concurrently or sequentially. Concurrent threads access the system at the same time (in parallel), while simultaneous threads come one after another. In order to achieve concurrency, the time between thread starts should be shorter than time needed for each thread to complete the test case, otherwise, the system will always have the workload produced by a single thread, as if users were arriving one after other. The actual number of concurrent threads may be also be affected by varying system response times, which may result in

the accumulation of unfinished threads awaiting a response from the tested system. Figure 5.2 gives an example of situation in which the chosen scenario and workload resulted in a significant number of concurrent users.

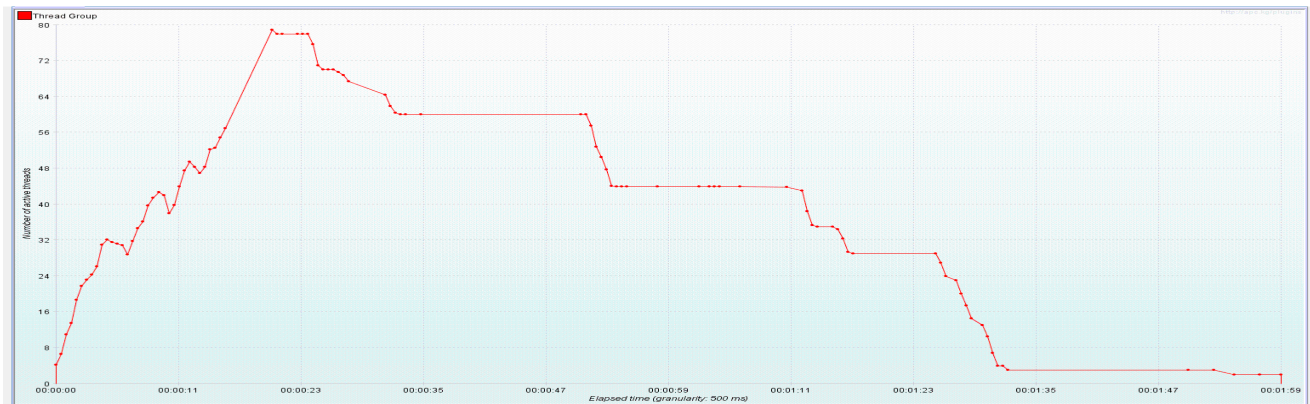
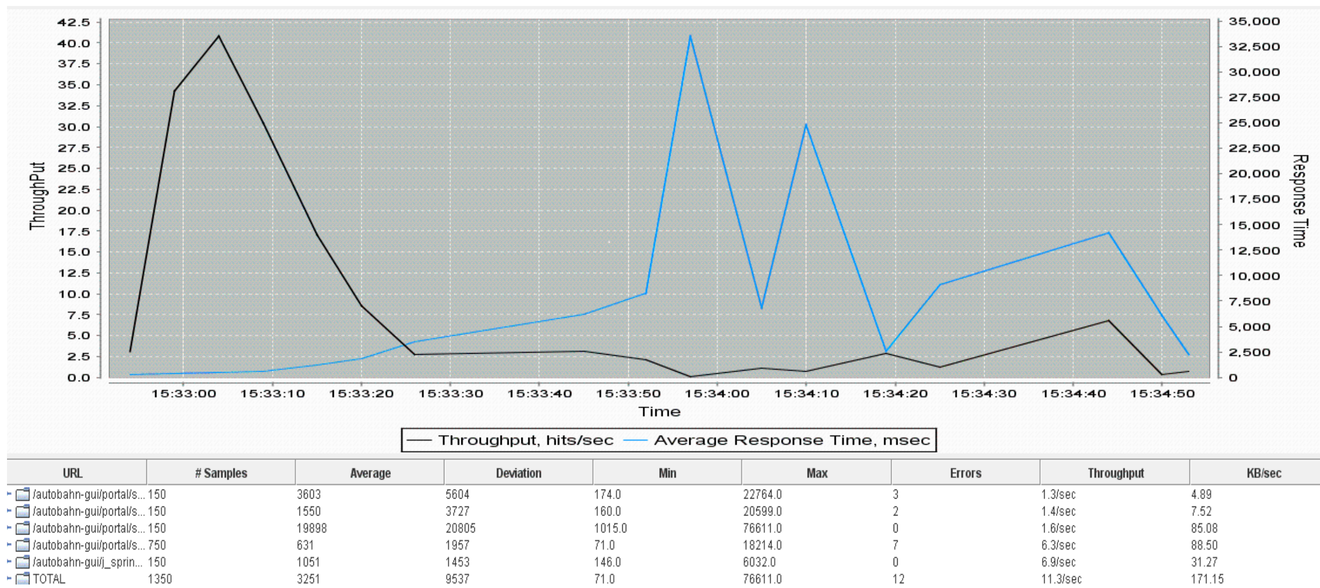


Figure 4.1: Number of active threads over time

Before starting the test, the tester should also identify metrics, namely the expected test result, including the observed parameters and their acceptable ranges. In order to produce a more realistic user load and avoid the effects of data caching, it is good practice to vary the values of request parameters across the threads, as otherwise, the test results may be significantly distorted. In order to achieve this, some tools have the ability to read the values of user inputs from a file and use them to form requests. Varying of inputs is particularly important in search, read and delete scenarios, while it may be less critical when new items are being created.

When the test tool is run, it will simulate the workload, starting with the threads that execute the test case and record the behaviour of the target system. The recorded quantitative test results then need to be analysed. These results are: time spent for each step in the process, request latencies, response times, number of requests per second, number of concurrent threads over time, etc. A result graph for throughput and average response time is shown in Figure 4.2.



URL	# Samples	Average	Deviation	Min	Max	Errors	Throughput	KB/sec
/autobahn-gui/portal/s...	150	3603	5604	174.0	22764.0	3	1.3/sec	4.89
/autobahn-gui/portal/s...	150	1550	3727	160.0	20599.0	2	1.4/sec	7.52
/autobahn-gui/portal/s...	150	19898	20805	1015.0	76611.0	0	1.6/sec	85.08
/autobahn-gui/portal/s...	750	631	1957	71.0	18214.0	7	6.3/sec	88.50
/autobahn-gui/j_sprin...	150	1051	1453	146.0	6032.0	0	6.9/sec	31.27
TOTAL	1350	3251	9537	71.0	76611.0	12	11.3/sec	171.15

Figure 4.2: Throughput (black) and average response time (blue) in ms

Usually, one test run with single workload settings does not provide a sufficient amount of data, and more conclusive data can be obtained by varying parameters defining the workload. In consecutive runs, the tester should adjust parameters in a way that will show the application behaviour under different workloads, particularly when trying to validate:

- Typical surge workload generated in a shorter period of time, which the system should handle.
- Reasonable constant workload during a longer period of time, which the system should sustain.

It is also interesting to identify the maximum values for these workloads. While this is quite easy for surge load, which can be gradually increased, it is more difficult to determine which time period is long enough to represent the ‘constant’ workload.

Below are some typical forms of graphs showing the throughput, average response time, and error count.

Increased Workload that the System can Handle During Short Period

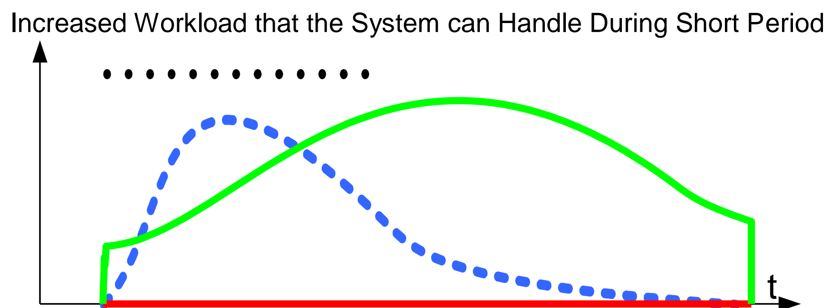


Figure 4.3: Throughput (blue), average response time (green) and errors (red) of system recovering from the intense, but short, workload (black)

- **Throughput** (blue) increases rather quickly after the requests start to arrive, but it decreases as the system is loaded with the requests. This decrease slows down or may even reverse once the new requests stop arriving. It eventually falls to zero as the residual requests are handled.

- **Average response time** (green) gradually grows as new requests arrive. Then, as the system clears the load and without the arrival of new requests, the time starts to decline.

- **Errors** (failed or rejected requests, red) are not present, or are proportional to throughput (shifted for the corresponding response delay). Therefore, they are not caused by failures produced by the overload of requests.

4.7.1.1 *Constant Workload that the System can Sustain*

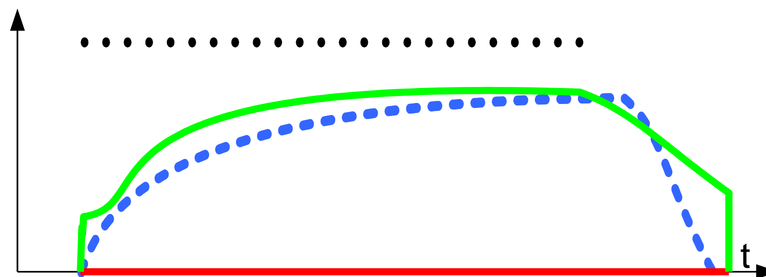


Figure 4.4: Throughput (blue), average response time (green) and errors (red) of a system handling the constant workload (black)

- **Throughput** (blue) grows, but at some point it becomes relatively constant. After the end of request arrivals, throughput starts to decline.

- **Average response time** (green) also grows, until it becomes relatively constant. It starts to decline after requests stop to arrive.

- **Errors** (red) behave as in the previous case shown in Figure 4.3.

4.7.1.2 *Increased Workload that Cannot be Processed by the System*

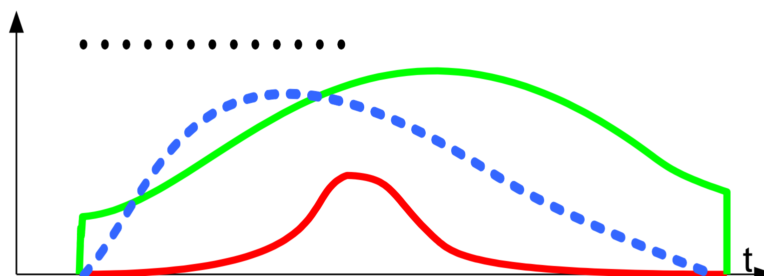


Figure 4.5: Throughput (blue), average response time (green) and errors (red) in a system that cannot cope with the workload (black)

- **Throughput** (blue) increases, but it then starts to decline due to overload (resulting in e.g. virtual memory trashing) and errors that are produced.
- **Average response time** (green) increases and remains high as long as new requests arrive and errors are generated.
- **Errors** (red) start to appear when some critical resource (some buffer, connection pool, memory, etc.) is exhausted or some problems related to the internal design start to emerge (memory leakage, deadlocks, race conditions, etc.). This is a successful stress test where produced errors that are logged can help in identifying weak points of the application.

While it is quite easy to verify if the system can withstand specific load during a limited period of time, it is much more difficult to determine a steady load that can be sustained by the system. In order to test this, the sample charts should be used as guides. The load should be increased by increasing the frequency of requests (or duration of time in which the request are generated), until the results change from Figure 4.3 to something similar to Figure 4.5. Once the limit is reached, the frequency should be decreased and duration increased, until the result becomes similar to Figure 4.4.

Additional JMeter plug-ins, Statistical Aggregate Report and additional metrics may also be useful [JMeter Plug-ins]

4.7.2 Automated Functional/Regression Testing

4.7.2.1 *Testing Considerations*

Before involvement with an automated testing tool, the decision of whether automation should be used has to be made. If the timing is critical, it is better to perform the testing manually since setting up the tool and writing a usable test takes a considerable amount of time. Also, if the layout or internal design of the application is expected to change dramatically in the near future, all automated tests would need to be rewritten anyway. In such circumstances, it is better to postpone the automation of the existing scenarios for manual tests. But this

effort should still be made at the first appropriate moment, since it will start to pay off with the first subsequent developers' commit. Initially, the automated tests should base on scenarios of existing manual tests, so their applicability for automation is also a factor contributing to the decision on whether to automate testing.

It is important not to go 'overboard' with the automation of the testing process. Automation cannot replace human testing and a person's intelligence and ability to notice things that are out of the ordinary or irregularities covered by the test. Also, it is important to find the balance between not covering all the reasonable cases for testing, and covering so much that immense time and effort was spent on writing tests, possibly covering some trivial or less-important parts of the application.

The strength of manual testing also has the potential to become a disadvantage, as manual testing relies on humans to verify the validity of functionality within an application. Sometimes it is not easy for humans to distinguish between correct and incorrect application behaviour. Also, an automated test can complete the test and navigate to the desired place or state of the application in much less time than any human could. Optimal usage of resources and time should be key considerations when writing automated tests.

The most critical tests should be run first. This will uncover the most serious errors before more resources are committed to tests that cover less critical parts of the application.

Tests should be added continuously as applications grow and change. It is important to never stop trying to find new bugs. Although Test-Driven Development (TDD) (described in the *Service Architecture Best Practice* document) could be employed in the development of web applications, this approach is not often used in practice.

The subject of the tests is, in itself, a big area. Static content, such as the existence of links, images, footers with company information, etc. may or may not be of interest for automated testing, depending on whether the content of the page is expected to change or not. In some cases, it is more reasonable to perform a single manual test for these features while committing time and effort elsewhere.

Dynamic content and AJAX-capable web pages can prove difficult to test since, in most cases, it is not easy to specify page elements that should be clicked or interacted with or page element that should be checked. For example, some pages generate content in a way that an element's name and/or ID is not always unique, or content that should be checked may not be readily present on the page if AJAX is used. In the case of former, the tester can instruct the tool to find the element by its relation to other elements on the page or by its position in the page. In the case of latter, a busy waiting loop could be employed to check whether an AJAX request was completed, and then the element that is the result of the AJAX request could be checked.

In a function test, some user input is required before checking for some type of result. Function tests can be expanded into testing user stories consisting of several linked function tests, which are the most important and the most complex types of tests.

Data-Driven Testing (DDT) is name given to the approach of reusing the same test multiple times with varying data. As with testing of the other application layers, it makes perfect sense to reuse a test and run it with diverse data, for example, testing authorization with correct and incorrect username and password combinations, or entering various page numbers in an application's "jump to page" section.

These and many more testing considerations are explained in more detail in the following articles:

- http://seleniumhq.org/docs/06_test_design_considerations.html – General recommendations and some specific to Selenium)
- http://softarc.blogspot.com/2008/06/automated-regression-testing-why-what.html – Best Practices section within the document
- http://www.wilsonmar.com/1autotst.htm – Good general recommendations although with emphasis on desktop applications
- http://www.onestoptesting.com/testing-download/download/Functional%20Test%20Automation.pdf – Good general recommendations although with emphasis on desktop applications
- http://www.softwaretestinghelp.com/automated-regression-testing-challenges-in-agile-testing-environment/ – Agile and automated testing.

4.7.2.2 *Tools for Automated Testing of Web Applications*

With a great number of automated tools available, it is not easy to choose the right one for the needs of project, and more broadly, for the organisation. Arguably, some of the most frequently cited and used tools are HtmlUnit, JWebUnit, JSFUnit, Celerity, Canoo WebTest, HTTP Test Tool and Selenium.

HtmlUnit

HtmlUnit is a web browser written in Java, which is able to load and parse HTML pages, including JavaScript, but it does not have a GUI display for user interaction, thus it is 'headless'. HtmlUnit is designed for automated testing as its primary purpose, and it should be used with testing environment, such as JUnit or TestNG. It is used by many other tools as a browser 'simulation'.

It is possible to write functional/regression tests with only HtmlUnit and JUnit or TestNG, but other tools facilitate test writing, for example, by providing features that make it easier to locate an element of interest, expressing conditions that should be checked, and offering greater integration with build and CI tools.

JWebUnit advertises itself as a high-level API that allows for more-rapid test creation than just using JUnit with HtmlUnit. JSFUnit focuses on JSF applications, running the tests inside the application container, which offers greater possibilities for testing managed beans, Faces Context and setting up complex data-driven scenarios. Celerity is a relatively new project that is a JRuby wrapper around HtmlUnit. Using JRuby syntax and simplified Celerity API makes writing tests an easier, less-technical task.

HTTP Test Tool

HTTP Test Tool, (httest), is a script-based tool for testing HTTP clients and servers on a protocol level. Since it is more focused on HTTP as protocol instead of Web pages, it has no JavaScript support, which makes the tool more appropriate for testing simple Web applications, HTTP clients and HTTP proxies.

Canoo WebTest

Canoo WebTest is a testing framework for web applications. The tests in Canoo can be expressed as standard Ant-build files, which makes Canoo a good candidate for integrating with projects that rely on Ant for building.

Another possibility for test writing is to use Groovy syntax and run completed Groovy scripts through Groovy's Ant Builder. Canoo can be good choice for teams that are used to Ant, as it can be adopted with little effort.

Selenium

Selenium is a set of tools with a different approach to supporting test automation. It is arguably the most popular test automation open source tool. It uses several approaches to testing web applications, and is capable of both performing the tests within actual browsers as well using HtmlUnit for headless testing. There are several pointers with recommendations on installing Firefox in headless mode, thus making automated testing in CI with Firefox more elegant. Selenium has good support for integration with build and CI tools such as Maven and Jenkins. There is a wealth of resources, documentation, users and knowledge base for Selenium. Selenium is already used for test automation of GÉANT Tools portal [GÉANT Tools]

The Selenium tool suite consists of Selenium IDE, Selenium 2 (a.k.a. Selenium Webdriver), Selenium 1 (a.k.a. Selenium RC) and Selenium Grid. Selenium IDE is a Firefox plugin providing interface for recording user actions in browser, which can be later used as functional/regression tests. Test are expressed in Selenium's domain specific language (Selenese), but can be exported as fragments of syntax of most popular programming languages (Java, C#, Python, Ruby). It is most often the starting point for test writers when writing tests.

Selenium 1 or Selenium RC (Remote Control) (officially deprecated in favour of Selenium 2) is a server that acts as a proxy for web pages that inject Selenium's JavaScript code to control the web page. This process, however, has proved to be somewhat slow in some instances.

Selenium 2 is a framework that sends commands to browsers on their native API level, which provides a much faster, flexible and maintainable way to write and execute tests. There are specially created 'drivers' for most popular browsers that are often maintained by browsers' developers themselves. All browser instances that will be used in tests have to be installed on the machine where the tests would run unless Selenium Grid is used via special "Selenium RC" driver.

Selenium Grid is a server that allows tests to use web browser instances running on remote machines. That server acts as a hub accessed via Selenium RC API, and tests written for both Selenium RC and 2 can work with Selenium Grid with no code change. The Hub allocates Selenium RC instances for each test, which can all reside on different machines, thus making the test faster and creating the possibility for concurrent testing of the web application with a great number of browsers, their versions and operating systems.

4.7.3 Web Services Testing

4.7.3.1 *Methods*

Many of the earlier stated principles that apply for functional, regression and load (performance and stress) testing of web applications also apply to web services, but with different weight and priority. Functional testing often serves as a verification of a properly programmed web service and is often performed as black-box testing, also called 'penetration testing'. Extending functional tests into scenarios by recreating an exact configuration that a customer requires is of great importance, with verifying proper functioning of web services.

On the other hand, the architecture of web services is essentially different than classic user applications or web applications. A web services-based architecture results in a distributed, loosely coupled system with dependent stakeholders. While this architecture holds great benefits in terms of scalability and extensibility of the system, it poses a great challenge for the tester. Potentially, the overall system may actually be derived from components that are constructed, tested and hosted by multiple stakeholders. As such, it is difficult to establish consistent code quality, availability and quality of service (QoS) across these entities. In an inter-enterprise Service Oriented Architecture (SOA) model, it is not enough to validate the functionalities of each component in the system, but it is also necessary to validate the efficacy of the relationships between these components.

This state of loosely coupled agents and distributed and componentised organisation is managed by proactively establishing Service Level Agreements (SLA). It is important to establish service levels, such as high-availability metrics, web-service response times and security policies upfront, during the requirements phase. Once established, these service levels should be proactively incorporated into test plans. Doing so is not only politically expedient when dealing with multiple stakeholders, but also less costly from a development standpoint. Eventually, both providers and consumers of the service, as well as programmers and testers, would benefit from these established guidelines. This promotes the possibility to perform component-level validation and benchmarking before integration into system to ensure they meet acceptable parameters.

Since SLAs rely on metrics such as response times, wait times (latency), number of transactions passed/failed, throughput (amount of data received by virtual users in amount of time), load size (number of concurrent virtual users), CPU and memory utilisation, performance and stress testing are essential for web services. Distributed development, large teams of developers and a desire for code to become more componentized may result in web services that are more susceptible to obscure bugs, extremely difficult to detect. Stress testing is an efficient method of detecting such code defects, but only if the stress systems are designed effectively.

Some resources that cover load, stress and performance testing of web services in great detail are:

- SOA Testing blog [SOA Testing]
- Syscon SOA web services journal [Syscon SOA]
- IBM's developerWorks site [developerWorks], with tips and tricks on stress testing web services.

Security testing should be of great importance to any team that is developing public web service that communicates sensitive private users' information. Both Section 4.4.3 of this document and Section 5.10 *Security* of the *Software Developer Best Practice Guide* [SDG] offer a good introduction into general considerations of secure coding and testing. The OWASP (Open Web Application Security Project) provides an extensive written guide that deals with many security intricacies in great detail, including entire chapter dedicated to web services [OWASP Testing Guide].Some of the concrete issues that are covered in the guide are:

- Testing WSDL – testing for accessing functionalities of a web service in non-intended way,
- XML Structural Testing – testing whether web service recovers from ill-formed XML requests,
- Content-level Testing – SQL Injection, XPath injection, Buffer Overflow, Command Injection,
- Naughty SOAP attachments – attachments that are too large and attachments with viruses
- Replay Testing – man-in-the-middle attack – by replaying XML requests gaining unauthorised access to web service.

4.7.4 Tools

There is a multitude of tools available for testing web services. Almost all of the tools mentioned below are created specifically for testing web services. These tools are aimed towards load and stress testing of the web applications and other HTTP-based testing tools, but can also be used in some manner to test web services. Of all listed tools, soapUI is the richest in features and area of application.

4.7.4.1 *WSDigger*

WSDigger is an open source GUI tool designed to automate black-box web services security testing (penetration testing). WSDigger functionalities can be extended with plugins, and comes packed with sample attack plug-ins for SQL injection, cross-site scripting and XPATH injection attacks. WSDigger is a standalone tool with limited reporting abilities. It is best used as a method to manually test web services against a set of known WS vulnerabilities.

4.7.4.2 *WebScarab*

OWASP WebScarab is a framework for analysing applications that communicate using the HTTP and HTTPS protocols. In its most common usage, WebScarab operates as an intercepting proxy, allowing for reviewing and modifying HTTP requests and responses between the applications that communicate using HTTP or HTTPS.

WebScarab functionality is implemented by a number of plugins, mainly aimed at the security functionality. Some of the more interesting plugins are Beanshell (execution of arbitrarily complex operations on requests and responses), Bandwidth simulator, session ID analysis, parameter fuzzer (automated substitution of parameter values that are likely to expose incomplete parameter validation, revealing vulnerabilities such as Cross Site Scripting (XSS) and SQL Injection, and a SOAP plugin.

4.7.4.3 *Pylot*

Pylot is an open source tool written in Python for testing performance and scalability of web services. It runs HTTP load tests, which are useful for capacity planning, benchmarking, analysis, and system tuning. Pylot generates concurrent load (HTTP Requests), verifies server responses, and produces reports with metrics. Tests suites are executed and monitored from a GUI or shell/console.

In Pylot tests, cases are expressed in XML files, where the requests (URL, method, body/payload, etc) and verifications are specified. Server responses can be verified by matching content to regular expressions and with HTTP status codes. Some level of integration of Pylot with other tools is possible, since it is able to run from command line but with limited value, since the reports produced by Pylot are aimed at manual test runs.

4.7.4.4 *soapUI*

soapUI is a free and open source, cross-platform Functional Testing solution. With an easy-to-use graphical interface, soapUI allows for easy and rapid creation and execution of automated functional, regression,

compliance, and load tests. In a single test environment, soapUI provides complete test coverage and supports all the standard protocols and technologies. soapUI is extensively used in practice and well documented with a wealth of third-party resources on the Internet.

The simplification of creating complex tests is provided by extensive user interface and tight integration with most popular Java IDEs (Eclipse, IntelliJ and NetBeans), although it is possible to write tests manually and execute them from the command line, soapUI has the ability to jump-start test case creation with just WSDL definitions (for SOAP style web services) or WADL definitions (for REST style services). However, a good knowledge of XPath and JSON is essential when writing extensive functional and security tests.

soapUI is a self-contained tool. Its workflow is more suited to stand-alone usage, with Maven and JUnit integration more of an afterthought. The biggest problem with JUnit usage is that soapUI assertions do not map to JUnit assertions. The only thing that can be asserted from the JUnit test case is whether the whole test finished successfully or not; which may be impractical, and diminishes usability of standard JUnit reports. soapUI's plugin for Maven, reportedly, has its own quirks, such as generating log files at unexpected locations and printing the console responses from failed tests, which may be completely unnecessary and can't be switched off.

MockServices in soapUI allows testers to mimic web services and create/run functional and load tests against them even before they are implemented. To some extent, it is also possible to test JMS (Java Messaging System), AMS (Adobes ActionScript Messaging Format used by Flash/Flex applications) and make simple test for web applications.

5 Code Analysis

5.1 Internal Code Reviews

Code reviews that are performed by peers are an important practice for improving the overall quality of software and the developers' skills. Such reviews have many advantages over testing:

- The collaboration between developers is increased as they develop a better understanding and appreciation of others code, while testing is quite often done by specialised personnel.
- Code readability and maintainability are improved, as code comprehensibility is one of prerequisites of a review.
- A higher percentage of defects are found when using reviews (up to twice as many as using testing).
- When an issue is detected by a review, the root cause of the problem is usually known. Defects found by testing require further analytical work.
- Reviews can be performed earlier in the development life cycle than testing.
- Reviews provide more precise and more immediate feedback than testing, and can suggest improvement opportunities to the code author.

The code review process comprises the following stages:

- Review setup.
- Review execution.
- Review evaluation.

These stages are described in the following sections.

For further information about code review, see [CODEREV].

5.1.1 Review Setup

The code reviews may be formal (as described by Fagan [FAGAN]) or lightweight. Well conducted lightweight reviews can produce similar results to formal reviews, with a significantly smaller time expense. Lightweight reviews are often conducted as part of the normal development process:

- Over-the-shoulder

  One developer looks over the author's shoulder as the latter goes through the code.

- Email pass-around

  When code is checked in, the source code management system automatically emails it to reviewers (who communicate using email or another electronic communication medium).

- Pair programming

  Two authors develop code together at the same workstation (as recommended in the Extreme Programming methodology [EXTREME]).

- Tool-assisted code review

  Authors and reviewers use specialised tools designed for peer code review.

A developer should not view a review as an annoying interruption from more interesting work. This would result in superficial reviews without much effect. While reviewing, developers become familiar with different ways of coding and better acquainted with the system they are reviewing. To gain benefits from code reviews, those performing it need to take a questioning and critical attitude. Maintaining this attitude throughout the long review is difficult, and it is helpful to perceive issue finding as a positive goal to preserving motivation.

Code reviews should not be used or seen as personal criticism but as something that facilitates learning and communication. The finding of defects and defects themselves must be viewed positively as an opportunity to improve the code. Finding and fixing a bug in a peer review in particular is a positive result that prevents the defect to appear during testing or before a customer or end user, and eliminates the greater cost of its addressing at a later stage. Therefore, code reviewing should become a means for team building, in which the author and reviewer work as a team, complementing each other.

Reviewers should focus on the purpose of the review. The primary purpose of a code review is to improve the code by finding defects, potential problems, or poorly-written sections. The reviewer should try to understand and challenge the assumptions and decisions behind code. This is especially important for finding errors of omission, as it is much harder to find a defect because of what was not written.

Typically, a review comprises 3 to 5 participants:

- The code author.
- Peers of the author (other developers).
- The architect of the system (if or while other developers are not familiar with the design).

- Possibly a participant unrelated to the analysed development or even entire project who does not know the system under review.

Regardless of what approach and team constitution are adopted, the reviewers should manage to understand the overall context, design, and actual code in the process. The review sessions should last between 60 and 90 minutes, covering between 200 and 400 lines of code at a time. At this rate, the reviews produce 70-90% yield of defects.

Two reviews should never be conducted on the same day. The typical lightweight practice (pair programming excluded) is to dedicate about two hours per week to the reviews in on regular basis, covering all code changes and additions made during the week. But during the catch up phase, reviews can be performed on a daily base.

The reviews can be conducted either through personal meetings, or, to reduce overhead, over-the-shoulder (among the peers), using desktop sharing tools, or even by participants working on their own and communicating with others via email. To instil a regular reviews practice, it is much better that to conduct the reviews in a regular working setting than in a meeting room.

It is highly recommended that the original author is required to prepare for the review by double-checking and annotating source code. The code should not be modified with review-specific comments in the code, but simply double-checked for quality, while the separate review annotations should guide the reviewer through the code, showing which files to look at first and defending the reason and methods behind code changes. The use of static code analysis tools may help in this process. Faced with coming reviews and encouraged to critically rethink and inspect their own code and explain changes, the author is likely to uncover many of the defects before the review begins. There is no need to be afraid that author comments make reviewers biased or complacent, they just make them more effective.

5.1.2 Carrying Out a Review

To carry out an effective review you must learn how the code is structured, what it is trying to accomplish, and where it falls short. It is difficult to do all this simultaneously while reading the code for the first time. Therefore, you should start with core components, base classes and complex areas. These may be followed by heavily modified code or code that is already associated with bugs. After that, you can select code for review on a use case or architectural layer basis. The list of candidates for a review can be maintained through regular issue tracking system, while the reviews can be logged through version control tagging.

An effective approach towards big reviews is to use multiple passes to review the code with the initial passes focused on learning about how the code works and later passes focused on detailed control of the code. The number of passes needed should depend on the reviewers' background, the size of the code under review, its complexity, and how thorough he review needs to be. These passes may be organised in the following order:

1. Problem overview

   Review the requirements or specification to understand what the code is supposed to do. What problem is it trying to address? What requirements is it implementing?

2. Solution overview

   Review the design document or specification to learn the overall solution to the problem. How are the requirements being addressed?

3. High-level organisation

   Review the code or detailed design document to understand how the code is organised and functions at a high level. What are the packages and the major classes within each? UML class diagrams showing class relationships and hierarchies are very useful for this pass, even as sketches. UML sequence or state diagrams can also be useful to highlight the main interactions or behaviour of the system.

4. Major class review

   Review the code for the most important classes as identified previously. While issues and problems can be found in earlier passes through the code, particularly at the design level, this pass marks the transition from learning about the code to critically analysing it for problems.

5. Multiple detailed review passes

   The previous passes will have given a reviewer a sufficient understanding of the code base to critically review the remainder of the code base.

At a start of a review session, the author may quickly explain what the most important aspects of the code and what it does.

Review checklists are useful for both authors and reviewers. Since it is hard to review something that's not there, omissions are the hardest defects to find. Checklists remind the reviewer or author to take the time to look for something that might be missing. It is also good practice for developers to produce their own personal checklist, as people tend to make the same mistakes. These typical errors can easily be determined by the reviewers, and the developer simply needs to compile them into a short checklist of common flaws, including items that tend to be forgotten. However, for such checklists to be effective, the number of items needs to be limited and should not go beyond about 7, since trying to remember too many issues to look for while reviewing does not work. The solution is to focus on a single goal for each detailed review pass through the code. Each goal can be as specific as an individual checklist item, but this level of detail is seldom necessary and leads to too many passes. Instead, use general goals corresponding to categories of issues.

For example, instead of a checklist item such as checking return codes of system calls, you could review error handling, ignoring the bulk of the logic and focussing within the pass on how errors might occur and how they are handled. Focussing on a single goal also improves the ability to find relevant problems. When focussing on a single goal such as error handling, the reviewer is more likely to focus on segments of code that deal with error handling (or lack of such code). The code is reviewed faster because irrelevant sections can be skipped and only relevant blocks or suspicious sections are carefully analysed. Changing the goal between review passes causes the perspective to shift, making it easier to identify a different set of issues in the same code base.

Some typical goals and corresponding checklist items used in reviews are:

- Functionality

- Does the code meet the functional requirements?
- Is the logic correct? Very useful help during this check are functional comments and descriptions in the code.
- Are there functional tests specified and performed?

- Class design and reusability
  - Does each class have low coupling and high cohesion?
  - Is the class too large or too complicated?
  - Is the code a candidate for reusability?
  - Is the code as generalised and abstract as it could be?
  - Should any feature or implementation be factored out into a separate class?
  - Does it do too little or is it unnecessary?
  - Is there an inappropriate intimacy between classes? Does a class use methods of another class excessively (putting aside legitimate usages in structural patterns like Adapter, Facade, Proxy, Strategy, Delegation, or Decorator)?
  - Is some artificial complexity introduced, for example, by forced usage of overly complicated design or design patterns where simpler design would suffice?

- Maintainability
  - Does the code make sense?
  - Is it readable and adequately commented for readability?
  - Are the comments necessary, accurate and up to date? Is the code over commented or overlaid by trivial, formal, useless, or obsolete comments?
  - Is duplication of code avoided?
  - Are any methods too long?
  - Are there test cases for code changes?
  - Does the code use expressions that are too complex, have side effects or quirks?
  - Are regular expressions in the code explained?

- Coding standards

  Does the code comply with the accepted coding idioms, standards and conventions? For coding standards, see the GN3 Software Developer Guide [SDG].

  - Are packages, classes, methods and fields given meaningful and consistent names?
  - Are some constants directly inserted into code? Are there unexplained numbers in algorithms (magic numbers anti-pattern) or literal strings in used in code for comparisons or as event types (magic strings)?

- Error handling
  - Does the code check for any error conditions that can occur?
  - Does the code correctly impose conditions for regular ‘expected’ values, for example null?

- Are method arguments tested by checking all possible invalid values? Are JUnit tests covering them?
  - Does the code use exception handling?
  - Does the code comply with the accepted exception handling conventions?
  - Does the code catch, handle, and where needed, log exceptions?
  - Does the code catch unnecessary general or runtime exceptions?

- Security
  - Does the code appear to pose a security problem?
  - Does the code require any special permissions to execute?
  - Does the code contain any vulnerabilities, such as format string exploits, race conditions, memory leaks and buffer overflows?
  - Is authentication and authorisation performed securely?
  - Are passwords encrypted and factored out of the source code?
  - Is communication with other systems secure?

- Unit tests
  - Are there automated unit tests providing adequate coverage of the code base?
  - Are these unit tests well-written?
  - Are there test cases for code changes?

For a detailed low level code review check list, see [CodeReview]

5.1.3 Handling Review Results

To report issues that have been found during the review back to the original developer, a review document can be created which lists all the found issues. For each issue the problematic section of code must be referenced, usually by supplying the file name and line number. This can be done by tagging the code with specially labelled comment, and using bug tracking tools. Produced tickets should be linked with the parent ticket that triggered the review (or associated with the review in another way. The status of the parent ticket should be changed (for example to "Reviewed"). Once all defects have been completed, the parent ticket should either be reopened for the review or marked as approved.

To reduce context switching and facilitate communication between the reviewer and original developer by anchoring the references to the code, it is god practice to record issues by adding them directly to the code base as comments, at the place where the issue occurs. Any other interventions on the source code during the review session are not allowed, as it should focus on detecting, not fixing bugs. Each review comment should be tagged with a special label, for example "REVIEW" ("TODO" followed by a reference to a review may be also used), so that the issues can be easily found by the original developer later. It is important that such comments are concise, yet understandable and neutral in tone to minimise any possible personal offence.

Once the review is completed, the code with the review comments can be checked back into the version control system and issues created within a tracking system can be made available to the original developers. While doing this, some of the review comments may be entirely replaced with generated issues, where suitable. Whatever is done, it should help to verify that defects found during the review are actually fixed. Defect fixing should be seen as a final part of the review process, as it may need discussion with the author and other reviewers. False positives, which turn out not to be defects, should not automatically be discarded. They are often the result of poorly documented or organised code. If a reader is confused, it is possible that the code should be changed to avoid future confusion, even if this just means introducing better code comments.

If the used issue or bug tracking system supports defining custom categories for the description of subject areas, the following classes (or subclasses) can be used. If supported (for example with tags) and needed in a specific case, several classifiers may be associated with a single issue.

- Utility:
  - Utility – Simplicity of installation and configuration
  - Utility – Overall usability

    Including coverage of usages/use cases expected by end users
  - Utility – Performance
  - Utility – Scalability and substantially

    Adequate features and performance in an environment with many instances of clients and services
  - Utility – Integration between components and with other systems

- Architecture:
  - Architecture – Appropriate decomposition of components or modules
  - Architecture – Consistent application framework
  - Architecture – Application of standards

    Are applied framework and standards adequate and adequately used?
  - Architecture – Adequacy for requirements

    If not, what would best deliver the required characteristics?
  - Architecture – Design of individual modules
  - Architecture – Need to re-design and re-implement from scratch
  - Architecture – Improvement of interaction between components

    Are the interactions between components understood? Where is the room for improvement? Any recommended steps?
  - Architecture – Documentation

- Software interfaces:
  - Software interfaces – Completeness and adequacy for the purpose
  - Software interfaces – Sufficient simplicity
  - Software interfaces – Testability
  - Software interfaces – Documentation

- Implementation and source code:
  - Implementation and source code – Code structure and reusability
  - Implementation and source code – Algorithm efficiency

    Efficiency of original algorithms implemented in code
  - Implementation and source code – Code efficiency

    Use of the most adequate and efficient means provided by the programming environment
  - Implementation and source code – Concurrency
  - Implementation and source code – Clarity and readability

    Or excess complexity and bloat
  - Implementation and source code – Developer documentation
  - Implementation and source code – Understanding and handling of failures
  - Implementation and source code – Efficiency of resource usage

    Particularly if usage of system resources is within typical boundaries
  - Implementation and source code – Resource leaks
  - Implementation and source code – Bottlenecks

- Packaging/releasing:
  - Packaging/releasing – Management process
  - Packaging/releasing – Testing
  - Packaging/releasing – Better performing alternatives for required dependencies

    Are these alternatives mature and integrated into native OS packaging by their maintainers?
  - Packaging/releasing – Licensing issues

    And are there alternatives with less problematic licensing?
  - Packaging/releasing – Manageability

    To what extent can we avoid dependence on particular environment versions (Java or Perl) so deployment can track typical OS upgrade cycles?
  - Packaging/releasing –Security concerns raised by dependencies
  - Packaging/releasing – Package dependencies

    Are dependency hierarchies sensible and optimal?
  - Packaging/releasing – Documentation

The resolution of the issue requires a final verification by the reviewer. However, some well described issues of lesser importance or requiring major refactoring may be left for the future. Optionally, a review document may be created from the review comments added to the code and related issues.

With or without author preparation, code reviews stimulate developers to write better code and to review their own work carefully, because they know that others will be looking at their code. This even works with regular random checks that do not fully cover somebody's code.

The quantifiable goals and metrics of the code review process should be decided in advance. This makes it possible to measure and improve the effectiveness of the process. If there is an adequate baseline, it is better to measure some external metrics, (like the number of issues detected during testing or the number of support requests) than the number of detected defects. However, since there may be external interference and it may take quite some time for external metrics to show results, it is useful to watch internal process metrics measuring how many defects are found, where these problems lie, and how long developers are spending on reviews. The most common internal metrics for code review are inspection rate, defect rate, and density of defects in reviewed code. However, such metrics should be automatically gathered by a code review tool.

Metrics should be used to measure the process efficiency and process change effects. If they help a manager uncover an issue, it should be dealt with by addressing the group as a whole within the regular progress managing procedure.

The most difficult code is more prone to error and as such needs to be reviewed heavily. A large number of defects may be due to the complexity of the code rather than to the author's abilities. To maintain a consistent message that finding bugs is good, management should never use every bug found and fixed during reviews as a basis for negative performance review. And with good review practice in place, it makes no sense to hold the author responsible for missed bugs that are discovered later. This attitude should be promoted in public policy, to prevent possible violations.

5.2 Automated Code Quality Checking

Automated code quality checking can be used to check conformance to many of coding guidelines during commit and at build time, for each piece of code modified by any developer. There are cases where the most sensible approach is to ignore a rule. This is contrasted to testing programs, which checks a program's quality at run time. In some unlikely situations (for example, massive initialisations of data structures or in automatically generated code), the indiscriminate usage of some conventions, like those related to naming, indentation and spacing, may be counterproductive. If an automated tool is used to check the code quality, this should be kept in mind.

Code quality checking is usually performed through peer review. A manual review of the code can detect bugs early on, make code more easily readable, make improvements to the object architecture and organisation, etc. Code quality checking tools can augment this process. Code quality tools use a number of static code analysis techniques to check code quality. These tools can be incorporated into the build process and thus automated.

Code quality is usually concerned with the following code qualities:

- Correctness

  The code should perform according to specification. Many software defects can cause incorrect behaviour, such as wrong outputs or system crashes. Some of these faults may be hard to detect because they appear only under certain inputs or operating conditions.

- Security

Some code defects may not necessarily violate behavioural specifications, but may nevertheless open the system up to security vulnerabilities (for example, buffer overruns, not validated inputs or a 'dead' code that malicious party can use to gain backdoor access).

- Performance

  The code should make optimal use of the resources (memory, CPU time, database calls) at hand. Some code defects may indicate inefficiencies or issues that can result in degraded system performance over time (for example, resource leaks, unnecessary size checks, etc.).

- Maintainability

  Developers should be able to easily understand and change the source code and other parts of the software (documentation, data structures, etc.). The programming style adopted by a software development project can help to comply with good programming practices which improve the code readability, re-usability, and reduce the cost of development. Usually this is achieved by adhering to coding standards and best practices, such as naming policies, formatting etc. Some consider code as defective, if it is not written according to team standards or 'style', or has improper procedure or code naming that addresses several issues from a single procedure.

Some coding defects can be considered more or less serious in certain teams. Therefore, it is important for code analysis tools to have different levels of seriousness for reported issues. The ability to remove coding standards items that are unrealistic in some circumstances can greatly remove the noise and promote tool adoption.

Automatic coding quality checking should complement code reviews. In the development process, code review is mandatory because it effectively catches developers' logical errors and mistakes. However, developers nevertheless skip code reviews due to schedule constraints. Various company experiences and research show that some developers are easily distracted by style issues and focus on them during code reviews. Consequently, some developers view the code review process as being time-consuming with little benefit, and therefore skip it. This can be prevented through automatic coding quality checking before the review, which allows the review to focus on finding logical and serious errors. Moreover, there are many bugs or discrepancies that cannot be found with analysis tools as they are today. Also, tools that report 10% of false positives are considered accurate. Further development of these tools and techniques will lower that threshold, but project team cannot rely only on analysis tools and completely bypass manual review process.

5.2.1 Static Analysis

Static program analysis is the systematic examination of an abstraction of a program's state space, or a way of exploring the different methods a program might execute in an abstract way.

Static-analysis techniques infer information about software behaviour based on a static representation of the software. This contrasts with dynamic analysis techniques which gather information by observing software as it is running. Because code is analysed rather than executed, static analysis does not require test cases.

Some static analysis techniques are:

- Pattern checking.
- Dataflow analysis.
- Syntax tree analysis.
- Flow graphs control.
- Instrumentation.
- Alias analysis.
- Dependency analysis.
- Binary analysis.
- Automated debugging.
- Fault isolation.
- Symbolic evaluation.
- Model checking.

Static analysis can find 'mechanical' errors, which are defects that result from inconsistently following simple, mechanical design rules:

- Security vulnerabilities.
- Memory errors.
- Resource leaks.
- Violations of API or framework rules.
- Exceptions.
- Encapsulation violations.
- Race conditions.

All these errors are difficult to find with testing or inspection.

5.2.2 Impact of Automated Code Reviews on Team and Developer

If the reviews process is performed after the development process, mistakes are harder to manage. The bad code has already reached production and the damage is done. This can be avoided through centrally controlled build processes and compliance checks that are executed automatically during the software builds. Harmful code is never promoted in the first place, reducing the need for costly clean-up projects and uncomfortable employee performance discussions. Instead, developers are given immediate feedback and have an opportunity to learn from their mistakes, while the system continues to ensure that the organisation is protected from dangerous code even if it takes a developer a couple of build attempts to apply a new coding standard correctly.

An unexpected benefit of coding standards checking that has been reported is promotion of developer education. Beginners, and even some experienced developers, initially did not understand the meaning and

importance of some coding standards items. One such example is "Prefer initialisations to assignments," which is known as an efficient way to initialise member variables in a constructor in C++.

Developers who do not carefully follow standards can produce code with many violations, and are often reluctant to correct the violations. This is especially problematic with violations from identifier- or design-related rules, which are difficult to correct in later development phases as this can have a large, project-wide impact. It is recommended to check for coding standards conformance from the early coding phase, so that the violated code is corrected before it is propagated out of the module. Another reason for early adoption is that developers prefer to immediately verify that a violation correction has eliminated the problem.

One reason why the coding standards checking effort may fail is the lack of rule maintenance at the organisation level. After the initial rule set is established, rules should be customised continuously, because some rules might not apply to particular projects and development style might vary across development domains. A rule set should be maintained throughout each project's operation, as the project constraints evolve.

5.2.3 Recommended Configurations for Maven and Jenkins

This recommendation is based on experience with cNIS and I-SHARe tools development in Java technology in SA2 Task 5. According to best practice guidelines those products are developed with Maven project management tool. Also Jenkins server is used for continuous builds.

It is recommended to use at least two well-known tools, PMD and FindBugs, because both of them analyse the Java application in a different way. FindBugs is a Java byte-code scanner. PMD is a Java source scanner, which also enables code duplication finding in a project. It is also good practice to use Checkstyle, which checks if developers adhere to a coding standard.

The following tools are recommended:

- PMD [PMD]
- FindBugs [FindBugs]
- Checkstyle [Checkstyle]

All these tools allow configuring code checking rules. It is recommended to use FindBugs and PMD with the default configuration and at first pay attention to warnings with high and normal priority. It is also good practice to execute PMD with the CPD (Copy & Paste Detection) extension. The configuration of the FindBugs and PMD Maven plugins which were executed during code analysis is presented in lines 262 – 292 in an I-SHARe POM file [POM].

Checkstyle has very restricted rules by default, which generates hundreds of warnings. In the SA2 development process custom rule sets are used. The configuration file is available online [SA2conf]. To reuse this configuration in a modular project like I-SHARe, a common-tools sub-module was prepared. This module is packed to a jar file with XML configuration file inside it. During the Checkstyle execution this configuration has to be applied (see the Checkstyle Maven plugin configuration in the I-SHARe POM file (lines 293 – 314) [POM]). Inside the plugin description the dependency to I-SHARe-common-tools is defined. The configuration section

contains the entry `<configLocation>` which references the checkstyle_checks.xml file, which is provided by a jar file defined in the dependency.

Automated code analysis is a part of the build process. The execution of all used tools is defined in the Maven POM file in a separate profile called "CodeAnalysis" (see lines 257 – 318 [POM]). This approach improves the productivity of the development process, because code analysis is not executed during all builds executed by developers. Code analysis is by default performed on the Jenkins build server, which is extended by plugins for the PMD, FindBugs and Checkstyle analysis tools. Also, developers can execute it on theirs workstation if needed.

To execute the "CodeAnalysis" profile during the Maven install, the developer has to execute a command, e.g.:

```
maven -PCodeAnalysis clean install
```

On the continuous integration server this profile is executed by default. This allows the developer to check the result of each commit that starts the build process.

Jenkins plugins for code analysis tools allow to trend reports to be added to the main job page and summarise the results of each build code analysis. Jenkins automatically collects historical data, which can be used to generate ‘trend’ charts. Based on this data, information about new and fixed warnings in each build is presented to user. Figure 5.1 represents how the quality of the code was changed. This page shows that two new warnings were found by the Checkstyle tool and nine warnings were fixed.

Figure 5.1: Example of Jenkins build result page

Jenkins plugins also support developers in fixing warnings. Each tool contains guidelines about warnings it generates and informs developers how to fix it. Figure 5.2 shows detail information about warnings. You can click on a file name to see code of this class with highlighted lines of code and hints with warning descriptions (see Figure 5.3).

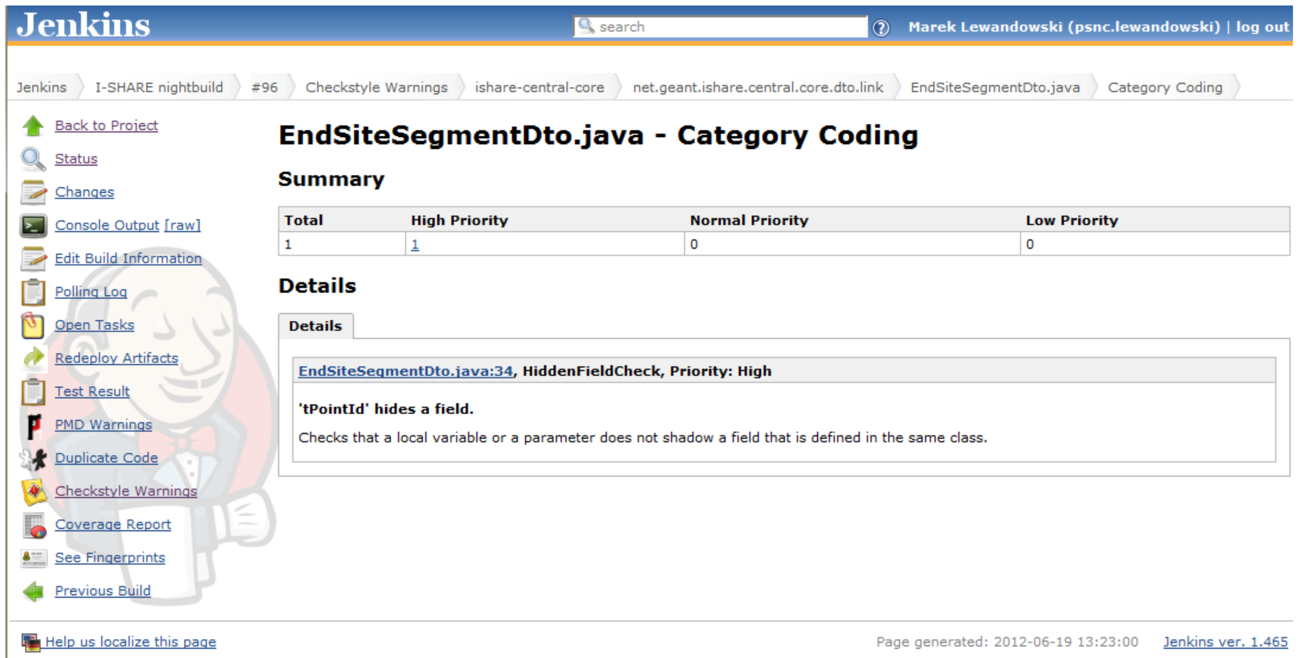


Figure 5.2: Example of CheckStyle warning description on Jenkins

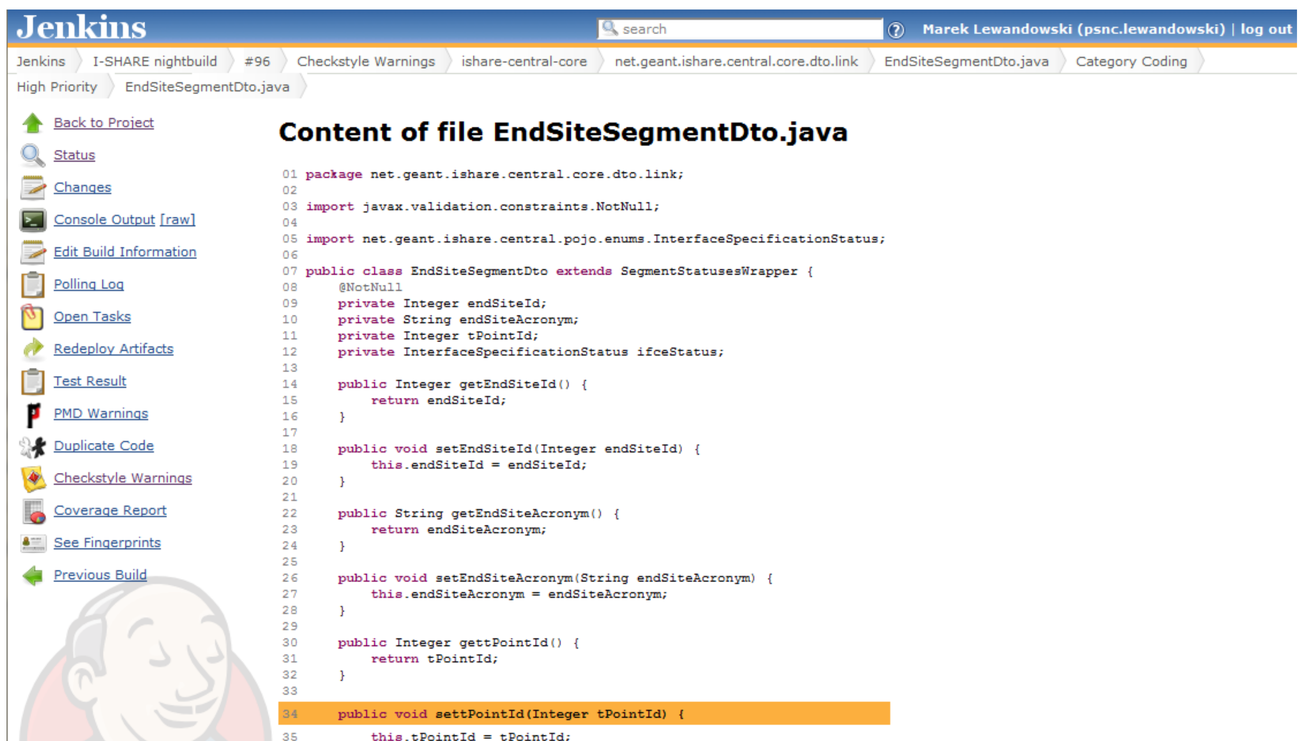


Figure 5.3: Example of Jenkins CheckStyle code viewer

Another advantage of using code analysis tools integrated with Jenkins is the good accessibility of code analysis results. Each participating project can check results without additional cost. The product leader or QA team can, for example, follow the code quality metrics.

The historical data Jenkins collects allows developers to consider recently introduced warnings. It is easier for the code author to maintain code quality warnings and fix them shortly after introducing them.

When using a code analysis tool, it is important not to pay attention to all warnings. Some tools can generate reports with hundreds of warnings and developers do not want to read or fix all of them. You have to select the warnings that have the highest priority from your application's point of view (e.g. in the performance category in FindBugs results). Also, it is recommended to prepare a custom configuration for tools where possible.

For more information see [SCan].

5.2.4 Static Code Analysis with Macxim

Macxim [Macxim] is open source software that performs a static code analysis of Java programs. It aims to scan Java source code and extract a simplified Abstract Syntax Tree (AST) representation of the source code, and eventually derive a number of source-code related metrics. This is done in two phases: source-code analysis and computation of software measures.

The Macxim tool stores all calculated measures in a repository implemented with a relational database (MySQL). The measures related to analysed projects can be viewed at three levels of granularity: application, package and class levels.

The project analysis can be both triggered manually and scheduled periodically through the Macxim web interface. The tool allows scanning projects kept in CVS or SVN repositories, however, no file-filtering options are available. The tool also allows the selection of a revision number, and the analysis can be scheduled periodically through the Macxim web interface. At present, however, it is not possible to integrate Macxim with Maven or Jenkins CI. The results of project analyses are stored in the repository of analysed software, which allows changes to be tracked in a software measures between analyses, as the development proceeds. However, each branch or a release of a given software project shall be analysed separately to allow for metrics comparison.

Macxim enables the comparison of the metrics of analysed projects, or their particular modules. The comparison covers all accessible measures. The side-by-side comparison can also be carried out for project modules, as long as they are analysed as separate projects.

5.2.4.1 *Source Code Measures*

Macxim is meant to support different kinds of measures for Java source code, where the majority of metrics is automatically calculated by the Macxim engine. The provided measures are divided into the groups related to code size, documentation, quality, modularity and Elementary Code Assessment (ECA) rules.

Code Size Metrics

The measures in this group are mostly related to the number of code lines, in particular, effective lines of code and number of classes, interfaces, packages and methods. The effective lines of code exclude blank lines, only braces lines and comment lines. In combination with the lines of comments, this forms the basis for deriving other measures.

Code size-related measures include also a number of simple averages derived from a number of classes, interfaces and a number of methods and attributes.

Code Documentation

Measures related to the documentation are based on the comment lines of code, which include both inline comments and only-comment lines. The measures calculated by Macxim also cover the TODO comments and missing Javadoc comments of methods, classes and interface definitions. As derived measures, average numbers of comment lines per class or interface are also included.

Code Quality

Code quality measures are meant to support the process of evaluating the maintainability and reliability of a software. In this set of measures, the Macxim tool includes metrics that are mainly related to code complexity, these include:

- **Cyclomatic Complexity** (McCabe Index) is a number of linearly independent paths in the flow of the unit of source code. The greater the complexity of methods reduces the testability of the code, which requires more effort to ensure good test coverage. Ultimately, this has an indirect influence on software maintainability costs.

- **Coupling between Objects** (CBO), a software measure that represents a number of other types a class or interface is coupled to. It is meant to count the number of unique reference types that occur through method calls, method parameters, return types, thrown exceptions and accessed fields. However, the sub-types and super-types of a given class are not included in the CBO calculation.

  High CBO values may indicate the complex interclass dependencies limiting the reusability of the code, lowering modularity and software encapsulation. The highly coupled classes are also more difficult to isolate and test.

- **Depth of Inheritance Tree** (DIT) is the maximum length of a path from a class to a root class in the inheritance structure of a system. DIT measures how many super-classes can affect a class.

  The increased DIT has an impact on maintainability and testability.

- **Lack of Cohesion in Methods** (LCOM) is viewed as a measure of how well the methods of the class co-operate to achieve the aims of the class. It measures the number of method pairs in a class that are independent and without cohesion, implied by using shared instance variables. It is the difference between the number of method pairs not using common instance variables and the number of method pairs using them.

- **Number of Children** (NOC) is the number of immediate subclasses (children) subordinated to a class (parent) in the class hierarchy. The NOC measures how many classes inherit directly methods or fields from a super-class. The high NOC can have a negative impact on software portability, since the classes with a large number of children are not easy to separate and replace. The children are dependent on a parent class by extending its functionalities and can, therefore, depend on certain functionalities the parent class provides. It may be difficult to find or design another class satisfying all these specific needs, in order to replace the parent class.

- **Response For a Class** (RFC) is the size of the response set of a class. The response set of a class is the set of all methods and constructors that can be invoked as a result of interaction with an object of the class. This set includes the methods in the class, inheritance hierarchy, and methods that can be invoked on other objects.

  If the RFC for a class is large, it means that there is high complexity. For example, if a method call on a class can result in a large number of different method calls on the target and other classes, it can be difficult to test the behaviour of the class and to debug problems, since comprehending class behaviour requires a deep understanding of the interactions that objects of the class can have with the rest of the system.

The aforementioned measures are calculated at application, package and class levels. Macxim also provides maximum, minimum, standard deviation, median and average values of those measures.

Code Modularity

In the set of measures related to code modularity, Macxim calculates metrics that describe if classes have defined methods and attributes, the percentage of interfaces is not implemented or the average number of interfaces implemented by a class.

Elementary Code Assessment (ECA) Rules

Macxim software also counts the appearance of violations of ECA rules, in a way similar to popular software tools such as PMD or Checkstyle. Altogether, it reports on 60 simple rules that every well-written code should satisfy. The rules are mainly related to code structure, used (or misused) constructs and patterns. ECA rules are calculated only at the application level.

6 Performance Improvement and Optimisation

6.1 Code Optimisation

Developers should have some basic performance considerations in mind from the very beginning of their implementation, but this should not make their design significantly more complicated. If a system does what it was designed for within acceptable performance boundaries, it is wise not to try to improve its performance, as such efforts may reduce readability or enlarge code, thus greatly complicating the original design, introducing bugs, and making the software much more difficult to maintain.

It is wise to keep the performance tuning for the end of each development cycle. With this in mind, Donald Knuth stated: “We should forget about small efficiencies, say about 97% of the time: premature optimisation is the root of all evil.” [Knuth]. Excessive early inclusion of performance considerations can result in design that is not clean and too complicated. A better approach is to design first, and then profile the resulting code to see which parts should be optimised. However, sometimes the programmer must balance design and optimisation goals. Some common sense coding-related practices should be adhered to:

- Select design with performance in mind, but do not try to optimise while coding.
- Avoid frequent communication by applying SOA-style decomposition and caching.
- Initialise and prepare data and structures in advance if they are to be used multiple times or if this can improve perceived performance.
- Factor common parts out of loops as long as this does not affect readability.
- Optimise later on, after identifying areas needing optimisation.
- Be aware what compiler is to be used and with what default optimisations.

Optimisation can be manual (by programmers) or automated (by compilers). The optimiser is a part of the compiler performing automated optimisation of code segments or programming idioms. Optimising the whole system is too complex for automated optimisers and, therefore, usually done by programmers. There are several aspects of performance (e.g. execution time, memory usage, disk space, bandwidth, power consumption) and optimisation generally focuses on one or two of them. Optimisation usually requires trade-offs as one factor is often optimised at the expense of others. The process of improving the performance of software by optimisation [OPTIMISE] should be started only after unacceptable consumption of available resources is detected by benchmarking and measurement of resources usage, or anticipated with appropriate certainty. Since the optimisation of code or execution platform may significantly transform and complicate the original design, it is important to understand the area and desired amount optimisation and direction of action. By adding code that is only used to improve performance, optimisation can often complicate programs or systems, making them harder to maintain and debug. Therefore, it is very useful to identify bottlenecks (or hot spots), i.e. critical parts of code consuming most resources and needing optimisation. This is done by using a profiler (performance analyser). This can help in determination of optimisation targets. In accordance with the Pareto principle, about 20% of code is responsible for 80% of resource usage (this is also sometimes referred to as 80/20 rule or even 90/10 rule, as actual numbers may vary).

Goals that are set for optimisation may greatly affect the approach to be used, since improvements of efficiency in terms of speed may often be at cost of memory or storage and vice versa. Sometimes performance bottlenecks can be caused by language limitations and critical parts can be rewritten in a different programming language closer to the underlying machine. Code optimisation techniques can also be categorised as platform-dependent and platform-independent. With current trends in hardware, the most frequent approaches involve calculation speedup with reduction of amount of data transferred to slow memory, storage, or over communication channel.

There are several levels of code optimisation [OPTIMISE]:

- The highest level is the design level. Optimising at the design level means choosing and implementing efficient algorithms. The choice of algorithm affects efficiency more than any other item of the design and this is also the most powerful optimisation.
- Next is the source code level. Avoiding poor quality code can improve performance.
- Some optimisations can be performed by optimising compilers - this is the compile level.
- At the lowest level is writing assembly code designed for particular hardware platforms and taking advantage of the full repertoire of machine instructions in order to produce the most efficient and compact code. Such optimisation can be performed by assembler programmers, or, at run time, by just-in-time compilers which exceed the capability of static compilers by dynamically adjusting parameters affecting generated machine code.

The approaches towards optimisation may be sorted according to growing complexity and incurred costs if performed at the end of the project:

- Modification of configuration parameters, like buffer or connection pool sizes.
- Application of compiler optimisation switches or opting for another compiler.
- Adaptation of the hardware to the application by adding additional memory, faster or larger storage, or faster or additional CPUs (if the implementation can benefit from these additional resources). Usually it is easiest to add memory, and this may result in a great improvement if the application can fit within the newly available memory.
- Performance of simple changes in code, like factoring out of parts of calculation (out of loops and only to branches in which they are needed), reordering of comparisons (putting the most selective ones first), usage of look up tables, caches, or buffers. Some of these changes can be performed automatically by using advanced compiler optimisations, others can be deducted from the insight into code, but some require a close knowledge of the problem being addressed and its specific patterns in data and common paths processing.
- Data compaction may be performed by either packing data structures or using compression algorithms.
- Change of data structures to speed up access.

- Substantial change or parallelisation of original algorithm or substitution with an alternative approach.

A well-chosen algorithm with appropriate complexity and data-holding structures can have a major impact on overall performance. Sometimes this choice results from balancing these factors [MAP]:

- Available programmer time.
- User sensitivity to application performance (some features are rarely used so the user might accept a performance that is not so good).
- The scale of the problem may be small enough to allow less efficient approaches to be used.

Therefore, it is more effective to use a brute force approach in parts of the code that do not require high performance, while other parts may require a more sophisticated approach and more effort from the programmer. Avoiding premature optimisation increases code readability and saves programmers from unnecessary effort, giving them time to spend on more significant work.

However, decisions made early in the design process potentially have a larger impact on the application's performance. Reverting early decisions may result in unacceptably high costs. Therefore, although general recommendations of agile development only require design and coding for immediate, clear needs, and although optimisation should only be done if it is needed, some performance-related decisions must be made in the design phase. These decisions are not actual optimisations, but may have great impact on future optimisations. They are mostly related to flexible design and include encapsulation of key algorithms, data and communication. These practices provide "performance by design".

There are also some good coding practices that may have a great impact on performance. For example, opening and closing of a file or database and HTTP connections should always be localised in a single method or at least class. This will prevent resource leakage (which also obstructs resource pooling), but also duplicate resource releases that may result in exceptions. This localisation also facilitates implementation of resource management and reuse. It is also wise to always check if the code uses the most efficient class when it handles specific resources (for instance by using buffered input / output classes and appropriate container classes).

6.1.1 Define Goals and Metrics

With current trends in design of computer hardware such as multicore processors and multiprocessor boards, the most effective way of improving overall performance is to increase the number of threads that concurrently work on a given task. This is the main reason for writing applications that use multiple threads. Another, more traditional reason, is to support separate, different interactions (e.g. user interaction from background processing or communication with external systems), handle concurrent users, and provide adequate responsiveness in each of these flows. Since the performance can be observed from the perspective of overall throughput or individual threads, there are two common metrics used to measure performance: items per time unit (e.g. transactions per second, jobs per hour etc.) and time per item (time to complete a single task). Three tasks help to define metrics and expectations early in the design:

- Clearly specified requirements can be used to select appropriate hardware and determine structure of the software.

- Having metrics defined and knowing what is expected enables developers to be sure that the system they deliver fulfils the criteria and makes it easy to declare the project a success or a failure.

- Defining metrics also includes specifying a test environment or conditions, and the expected inputs used for test cases.

6.1.2 Choose an Appropriate Algorithm

When selecting one of the candidate algorithms for a particular problem, it is important to evaluate their algorithmic complexity. This is a measure of how much computation a program will perform when using a particular algorithm. It measures the efficiency of the code but not the complexity necessary to implement an algorithm. The code with the greatest algorithmic complexity tends to dominate the runtime of the application. However, algorithms with lower algorithmic complexity must be enforced only in the most critical code segments in terms of processing time, as these algorithms are usually more difficult to implement and understand. Which code part is time-consuming also depends on how frequently it is used and on the typical values or sizes of its inputs.

The best algorithm for a single thread does not necessarily correspond to the best parallel algorithm. Also the best parallel algorithm may be slower in the serial case then the best serial algorithm.

The choice of algorithm can change the way the application behaves completely. In many cases it is possible to write the application in such a way that the choice of algorithm for critical parts of the code can be replaced at a later point, if this becomes critical to performance.

6.1.3 Design and Use Efficient Structures

Three attributes of application construction can be considered as ‘structure’:

- Build structure (how source code is distributed between source files).
- How source files are combined into libraries and applications.
- How data is organised in the application.

Each data structure will potentially be accessed millions of times so even a small gain in performance can be magnified by the number of accesses. In many programming languages there are libraries of code that implement different data management structures. For C++ there is the Standard Template Library with many efficient data structures. Java also provides various alternative container classes based on distinct data structures suitable for a range of usage scenarios. If the developer carefully encapsulates the use of data structures, they can later be replaced with others that are more suitable for actual use.

6.1.4 Ensure Data Density and Locality

When an application needs an item of data, it fetches it from memory and installs it in its cache because the cost of fetching data from the cache is significantly lower than fetching it from memory. Data that is frequently accessed will become resident in the cache and the application will spend less time waiting for the data to be retrieved from memory. The same caching principle is used to optimise access to other resources, like earlier constructed disposable objects and data structures, pooled connections, or data on permanent storage.

To make the use of caches effective, the application must make the current working set it operates with small enough to fit into the cache, or, where possible, appropriately extend the cache size. If an extension of the cache is not a viable option, the algorithm which allows the data to be divided into smaller chunks or blocks that can be processed sequentially, or, even better, in parallel, has a clear advantage.

Improvement in performance can also be achieved by taking care of the order in which variables are declared and laid out in memory. Placing variables that are often accessed together into a structure so that they reside on the same cache will lead to performance improvement.

6.1.5 Select the Appropriate Array Access Pattern

Striding through array elements is a common data access pattern. If an array can be arranged so that the subsequently accessed elements are adjacent, performance improves. Most compilers are able to interchange the loops and improve memory access patterns; however, in many situations it is necessary for the developer to restructure the code appropriately.

6.1.6 Use Compiler Optimisations

Modern compilers are capable of executing many optimisations on the original source code. There are two fundamental types of optimisation: elimination of work and restructuring of work. There is a long list of optimisations that can be performed by a compiler which all aim to either eliminate unnecessary actions or carry out actions in a more efficient way”. However, to use these optimisations, appropriate compiler switches must be set. Also different parts of code may require different compiler optimisations.

Some optimisations may result in a performance gain on one processor but result in performance loss on a different processor.

Using a compiler to do the optimisation has few benefits:

- The compiler will do the optimisation correctly (this may have exceptions with some compilers or the most ‘aggressive’ optimisations).
- The code remains manageable (if the algorithm changes, it will require only a minimal number of modifications of the source code, since it was unaffected by optimisation; the compiler will appropriately reapply its optimisations to the new source).
- The compiler will do the optimisation only if this results in a performance gain.

6.1.7 Use Profile Feedback

Profile feedback is supported by most compilers. It is a mechanism that enables the compiler to gather information about the runtime application behaviour. It is a three-step process. The first step is to build an instrumented version of the application to collect runtime metrics. The next step is to run the application on a typical data set, and the final step is to recompile the application using this profile information. The benefit of profile feedback depends on the application. Some will see no benefit while some may get a significant gain. One concern of using profile feedback is that it complicates the build process and increases its duration, so profile feedback should be used only on release builds. Another concern is that using profile feedback optimises application for one particular scenario while slowing down all other scenarios.

6.1.8 Computational Performance Optimisation in Java

When attempting to optimise Java code, it is important to remember that proper algorithms and data structures are the key elements in the computational performance of developed software. However, it is advisable to follow practical hints when developing Java code or refactoring existing code for better performance. Some of these are as follows:

- Use primitive types when conducting arithmetical computations. Avoid the use of wrapper classes (auto-boxing) unless it is necessary.
- Use Java `StringBuilder` instead of "+" concatenation in order to minimise the creation of temporary objects.
- Use buffers with I/O streams, readers and writers, or base your I/O on `java.nio` classes. Buffers should be at least 1-2 KB in size. It is also possible to get a better performance by using bulk operations instead of single byte-at-a-time reads.
- In web apps, cache repeatedly used pieces of markup by applying OSCache [OSCACHE] or Seam <s:cache>.
- Use Java API classes when they offer native performance, where appropriate (e.g. `System.arraycopy()`).
- Avoid using URLs as keys in `Map` collections, as on the key lookup action the URL object actually calls the URL address it wraps, which may result in a serious performance drawback.
- Be careful with computationally expensive constructs. A good example is a Java switch statement. The Java compiler has two instructions to compile the switch statement: `tableswitch` and `lookupswitch`. Favour `tableswitch` over `lookupswitch`; it is faster because the value does not need to be compared to every possible case. `Tableswitch` jumps to the default section if the value does not lie within the range or, if within the range, it calculates the correct offset and jumps to it. To verify what the switch statement is compiled to, you can use the `javap` <class file> command. To force the compiler to use `tableswitch`, the cases must be in a close range. It is possible to add extra

empty cases to populate the cases range and eventually make the compiler use `tableswitch` instruction.

The performance of developed software is particularly important if it deals with large collections of data. The use of primitive collections (and sometimes even arrays) may be useful to improve the overall performance. As making operations on large data sets even faster is a complex task, it is useful to take advantage of the existing solutions distributed as Open Source Software, which significantly exceed standard Java API. When considering the use of specialised Java libraries for handling large data sets, open source implementations such as fastutil, Apache Mahout Collections, GNU Trove or HPPC are worth looking at. These libraries provide, for example:

- Data structures with a small memory footprint, fast read access and data insertion, which also implement the `java.util` Collections API. Fast iterators are also included.
- Support for big data structures (e.g. large data collections, arrays of arrays), where the API provides bulk methods implemented with high-performance system calls.
- Useful I/O classes for binary and text files.
- Efficient full-text search engines for large document collections, supporting powerful indexing, expressive operators, distributed processing, multithreading and clustering.
- Low-level data structures for further performance tuning.

It is also worth mentioning, that the software binaries can be reduced in size by using Java code shrinkers (e.g., ProGuard). Such tools can also conduct code obfuscation, eliminating a lot of debugging information, hence further reducing the size of byte code and making the byte code much harder to reverse-engineer (which sometimes may be desirable). Depending on the source code and the version of JVM (Java Virtual Machine), code shrinkers can also perform many more operations regarded as code optimisations, such as:

- Removal of unnecessary field access, method calls, branches, `instanceof` tests.
- Merging identical code blocks.
- Making methods private, static and final when possible.
- Making classes static and final when possible.
- Performing peephole optimisations, e.g. replacing arithmetical operations with bitwise and bit shift operators.

However, when deciding which parts of code base require optimisation, it is advisable to use Java profilers (like JVisualVM) to find methods with time-consuming execution.

6.1.9 Code Parallelisation

The growing requirements users define against software systems force IT vendors towards more complex, often multiprocessor infrastructures. The direction of hardware development also affects programming

languages, which aim to exploit the advantages offered by the rapid evolution of hardware. One of key changes in recent years is a shift towards concurrent processing on several computing units.

As applications with complex architectures are common and widely used, threads introduce many advantages, reducing development and maintenance costs and improving performance. Threads allow complex workflows to be designed, asynchronous processing to be implemented, the responsiveness of GUI applications to be improved, resource utilisation in server applications to be managed better etc. Threads also play an important role in designing applications that run on multiprocessors, and help to optimise responsiveness of systems that run on single-processor machines. However, threads also introduce hazards to applications' algorithmic correctness, performance, liveness and responsiveness. When optimising the software by exploiting concurrent programming, the advantages and risks the concurrent programming introduces need to be kept in mind.

Darryl Gove in his book "Multicore Application Programming" [MAP] points out:

Work can be divided among multiple threads in many ways, but there are two main forms of parallelism: data parallelism and task parallelism. Data parallel applications have multiple threads performing the same operation on separate items of data. Task parallel applications have separate threads performing different operations on different items of data.

Parallelisation can be achieved by having a single application with multiple threads or a multi-process application made of multiple independent processes, some of them potentially having multiple threads. The typical patterns of parallel execution of are the following:

- Multiple independent tasks

  This approach is not so interesting from a parallelisation strategy since there is no communication between the components. This approach is not expected to lead to a performance change of individual tasks. This strategy is interesting because tasks are independent and performance should increase linearly with the number of cores, processors, or nodes executing threads, without any additional programming effort related in parallelisation.

- Multiple copies of the same task

  Way to do more work at the same time is to employ more copies of the same task. Each task will take the same time to complete but because multiple tasks are working in parallel, output of the system will increase. The performance of the system is increased in terms of items processed per time unit not in rate at which a single is completed. This pattern is a version of multiple independent tasks.

- Multiple loosely coupled tasks

  In this approach tasks are largely independent. They may occasionally communicate but that communication is asynchronous and the performance of the application primarily depends on the activity of this individual tasks.

- Single task split over multiple threads

  Typical scenario is distributing a loop's iterations over multiple threads so that each thread computes a discrete range of the iterations. Single unit of work is divided between threads so time taken for the work to complete should decrease proportionally to the number of threads executed in parallel. There is

a reduction in completion time and also an increase in throughput. However, there is usually a need to synchronize the threads at some points, which may reduce the parallelism.

- Using a pipeline of tasks to work on a single item

  A single unit of work is split into multiple stages and is passed from one stage to the next rather like an assembly line. The level of parallelism depends on the number of busy stages and duration of the longest one.

- Division of work into a client and a server

  One thread (the client) sends request to another (the server), and the other thread responds. This scenario provides a performance improvement because the client can continue to interact with the user while the server is performing the requested calculation.

- Splitting responsibility into a producer and a consumer

  This model is similar to both the pipeline and the client-server model. Producer is generating units of work and the consumer is taking those units and processing them. Again, this approach does not necessarily reduce the latency of the tasks but provides an improvement in throughput by allowing multiple tasks to work simultaneously. This model may provide a way of reducing the complexity of combining the output from multiple data producers.

- Combining parallelisation strategies

  In situations where one parallelisation strategy is not sufficient to solve the problem it is necessary to select a combination of approaches.

Parallelisation opportunities can be identified by trying to recognise one of the above patterns by performing the following steps:

1. Get a representative application runtime profile and identify the most time-consuming regions of code.

2. Examine code dependencies for these regions and determine whether they can be broken so that the code can be performed as multiple parallel tasks or as a loop over multiple parallel iterations. At this point it may also be worth to investigate if different algorithm would give code that could be more easily made parallel.

3. Estimate likely performance gains from identified parallelisation strategy and if it promises close to linear scaling with the number of threads, then it is probably a good approach. If not, it may be worth to broaden the scope of the analysis.

4. Broaden the scope of the analysis by considering the routine that calls the previously examined region of interest.

All parallel applications must have some way of communication between threads or processes. It is usually an implicit or explicit action of one thread which is sending data to another. For example, one thread might be sending signal to another that some data is ready, or a thread could place a message on a queue which would be received by another thread handling that queue. The support for messages sending between threads or

processes is provided by the operating system or runtime environment. All major programming languages have standardised support for threads, like Java threads or C++ Pthreads (POSIX Threads), but there are also specialised libraries that provide more abstract parallel programming paradigms. These are message passing, with several implementations of MPI (Message-Passing Interface) specification, and shared memory paradigm (which also stands behind the thread programming model), with implementations of OpenMP API. However, these libraries are usually applied in applications used in high performance and scientific computing, while everyday parallelisation usually employs threads or readily available mechanism for inter-process or inter-system communication.

6.1.9.1 *Synchronisation Primitives*

Darryl Gove in his book "Multicore Application Programming" [MAP] points out:

Synchronisation is used to coordinate the activity of multiple threads. It is common to use synchronisation primitives which are provided by operating system. In opposite to custom methods (which we can write), OS provides support for sharing the primitives between threads or processes.

- Mutexes and critical regions

  Mutually exclusive (mutex) lock is the simplest form of synchronisation. They can be placed around a data structure to ensure that it can be modified by only single thread at a time. If multiple threads attempt to get the same mutex at the same time, only one will succeed while others have to wait. This situation is called contended mutex. The region of code between the acquisition and release of a mutex lock is called a critical region. This code will be executed by only one thread at a time. Threads block (block - sent to sleep) if they try to get a mutex lock which is already held by another thread.

- Spin locks

  Spin lock is a mutex lock which doesn't sleep when trying to acquire the lock held by another thread. Advantage of spin locks is that they will get the lock as soon as they are released while mutex has to be woken up by the OS before getting the lock. Disadvantage is monopolising resources while spinning on a CPU, while mutex when sleeping frees CPU for another thread.

- Semaphores

  Semaphores are guarded counters used as a mechanism to impose limit of a resource (when there is a limit). Every time a resource is added or removed from a buffer, the number of available positions is increased or decreased.

- Readers – writer locks

  Data that is read-only does not need protection with some kind of lock, but sometimes it has to be updated. A readers-writer lock allows many threads to read the shared data but allows one thread to get a writer lock to modify the data.

- Barriers

  There is situation where a number of threads have to complete their work before any of them can start the next task. It is useful to have a barrier where all threads wait until all of them finish their tasks.

- Atomic operations and lock-free code

  Sometimes using synchronisation primitives can produce a high overhead particularly if they are implemented as calls into the OS rather than calls into a supporting library. This decreases the performance of the parallel application and can limit scalability. In some cases using of atomic operations or lock-free code can result in better functionality without this amount of overhead. Atomic operation is one that successfully completes or fails and never results in a 'bad' value.

6.1.9.2 *Concurrent Programming in Java*

The Java programming language provides a significant step towards developing concurrent applications, providing a number of mechanisms to improve the way applications work and to make it easier for programmers to develop concurrent systems. In fact, multithreading is already present in a large number of frameworks, application containers, server applications and in the JVM itself (e.g. garbage collector threads). The allegedly single-threaded Java programs are in reality multithreaded, with the concurrent nature being hidden by the Java platform. However, the strength of Java concurrency is not only multithreaded environments or frameworks, but most of all the coherent programming API Java for developing powerful multithreaded information systems.

The Java programming language defines a thread as a basic processing unit. It is represented by a `Thread` class. In fact, JVM allows an application to have multiple threads running concurrently. Threads have a well-defined API, including a publicly accessible `run()` method, which executes the defined computations and is run only once. When creating threads it is worth remembering that they can be created by subclassing the `Thread` class, but also by implementing a `Runnable` interface.

While threads can be started by calling their `start()` method, it is not advisable to use the `stop()` method the thread has in its API (the method is deprecated). It is good practice for developers to implement natural termination of a thread in the body of the `run()` method. It is also worth mentioning that threads cannot be run when they sleep, are blocked by the I/O operation or wait on the monitor. In fact, using threads that do not mutate the shared data is safe and quite easy. In other cases, the proper mechanisms for threads synchronisation are to be applied.

Java delivers mechanisms for multithreading at the language level. These mechanisms are centred on synchronisation, which is the concept of coordinating activities and shared data access among multiple threads. Java uses a mechanism of monitors. Monitors are associated with every Java object. Java's monitor supports two kinds of synchronisation: mutual exclusion and threads cooperation. Mutual exclusion is supported by locking via object locks, which allows threads to work independently on the shared data. Threads cooperation is supported by `wait()` and `notify()` methods of class Object, which enables threads to work together. For further detail on concurrent programming and threads in Java see [Java].

It is also worth mentioning, that JVM can sometimes notify a waiting thread without a reason. This is called a spurious wake-up. Therefore, it is advisable for threads to wait in loops, testing for the condition that should have caused the thread to be awakened.

6.1.9.3 *Concurrency-Related Issues*

This section focuses on the means used to provide parallel processing in multithreading, concurrency hazards, and introduces recommendations how to avoid them. Although general, it is illustrated with mechanisms available in Java.

Race conditions

A race condition occurs when the correctness of computation depends on the relative timing or the interleaving of multiple threads by the runtime. The most common race condition is based on a check-then-act pattern, which uses a potentially stale observation to perform a computation, while, in the meantime, the observation may become invalid. To avoid race conditions, the sequences of operations as check-then-act and read-modify-write must remain thread-safe. This can be ensured by atomic execution of such operations. Java delivers locking mechanism to guarantee that atomic operations (check-then-act and read-modify-write) are thread-safe and executed as compound actions, following the statement that "Operations A and B are atomic with respect to each other if, from the perspective of a thread executing A, when another thread executes B, either all of B has executed or none of it has. An atomic operation is one that is atomic with respect to all operations, including itself, that operate on the same state" [JCIP].

Data races

Data races are the most common programming errors found in parallel code. When multiple threads use the same data item and one or more threads are updating it, a data race occurs. Data races occur when the synchronisation is not used to coordinate multithreaded access to shared changeable (non-final in Java) field(s). Data races are more specific than race conditions, they happen when a variable is read by more than one thread and written by at least one, and read and write operations are not ordered by happen-before pattern. If the threads operating on the shared variable are not synchronised on the same lock, it can happen that the pattern of happen-before is not met.

Avoiding data races can be very simple although it can be very hard to identify them. However, there are tools for detecting them like the Helgrind tool, which is part of the Valgrind suite [Valgrind] for memory debugging, leak detection and profiling. Other tools which are able to detect potential data races are ThreadSanitizer, DRD, Intel Thread Checker, and Oracle Solaris Studio Thread Analyzer.

Make sure that only one thread can update the variable at a time. Placing synchronisation lock around all accesses to that variable and ensuring that before referencing the variable the thread must acquire the lock is the easiest way to do this.

Deadlocks

Sometimes using synchronisation primitives to manage sharing access to resources between threads can go wrong.

Java uses a locking mechanism to ensure thread safety. However, improper use of locking can result in lock-ordering deadlocks. Furthermore, the use of thread pools and semaphores to determine/limit resource consumption with no understanding of the activities being bound may lead to resource deadlocks. The following section outlines a number of causes of liveliness hazards and proposes possible approaches to prevent them.

First, there is the deadlock where two or more threads cannot make progress because the resources they need are held by another thread. A deadlock occurs when threads in a group of threads are waiting for a condition that can be caused only by another thread member of the group. An example of the simplest case of deadlock, is when the thread T1 is holding a resource L2 while waiting for resource L1 to become available. Meanwhile, T2 is holding resource L1 and is waiting for resource L2 to become available. T1 cannot make progress because it is waiting for L1, and T2 cannot make progress because it is waiting for L2. In this case the deadlock is encountered, as there is a cyclic locking dependency between given threads.

If a set of threads is deadlocked in the Java environment, they are out of operation. Depending on the roles they play, the Java application may stall completely, or a sub-system may stall leading to unavailability of the system. Some of the deadlock categories and situations that occur are presented below:

- **Lock ordering deadlocks**: This may occur if two threads acquire the same locks but in a different order (see Figure 6.1). The best way to avoid deadlocks is to ensure that threads always acquire the locks in the same order. If every thread tries to acquire the locks L1 and L2 in the same order, there will be no cyclic lock dependency, so the deadlock will not occur. Therefore, if thread T1 acquires the locks in the order L1 and then L2, it will stall while waiting for lock L1 without having first acquired lock L2. This would enable thread T2 to acquire L2 and then eventually release both locks, allowing thread T1 to make progress.

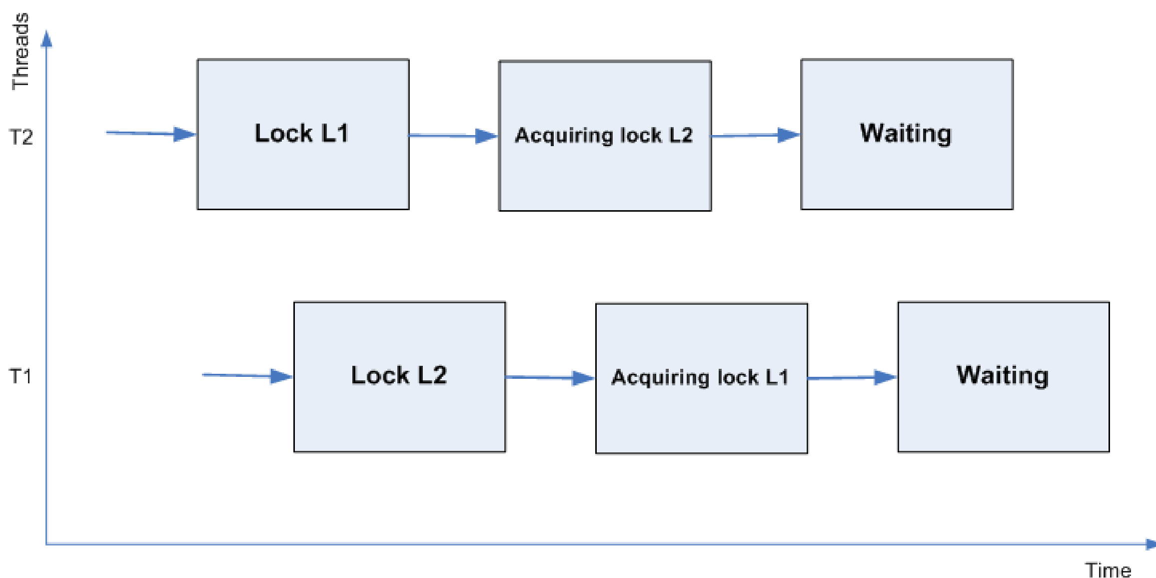


Figure 6.1: Lock ordering deadlock

The code inspection in terms of revising all execution paths is often not sufficient to apply the global ordering. A holistic view of the application is required to assess if ordering deadlocks may occur.

- **Dynamic lock ordering deadlocks**: These are ordering deadlocks where the developer has no control over lock acquisitions. This can occur if lock ordering depends on external inputs, e.g. in situations where the lock order is determined by the order of arguments passed to the method.

As the order of arguments cannot be controlled, it is advisable solution to enforce ordering on the lock acquisitions and keep this ordering throughout the application. `System.identifyHashCode()` can be used to induce the lock ordering. In infrequent situations where two objects have the same hash codes the third lock can be implemented to prevent the inconsistent lock ordering. If a given thread acquires the additional lock, then this single thread performs the acquisition of the remaining two locks. Using this approach in a consistent way improves the prevention of deadlocks. In simple cases where objects have unique and immutable identifiers, imposing the lock ordering can be implemented with their help, with no need for introducing the additional lock.

There are also other cases, where the deadlock may occur between different objects. This is much more difficult to spot. It may be resolved by detecting alien methods. These methods, when holding a lock from their own class, try to acquire the lock from another object. As alien methods are risky to use, the open calls approach is more advisable. This approach is to call a method with no locks held on its own class. Implementing the approach as an open calls paradigm makes it easier to analyse programs for possible deadlocks than programs that allow alien methods in their code base.

- **Resource deadlocks**: Threads may also get blocked while trying to acquire resources. The resources are often organised in pools, where a pool is guarded by a semaphore. However, the resource deadlock may occur if the thread requires access to a number of resources and the required resources are not always in the same order.

Avoiding Deadlocks

In multithreaded applications it is rare that a program acquires only a single lock and in such cases the lock ordering deadlock does not occur. However, as it is more likely that multiple locks are acquired, the lock ordering approach is an inevitable part of software design. It is good practice to avoid all possible deadlocks and the following activities facilitate this:

- Code base auditing for deadlock freedom: A first step in auditing source code for scenarios that lead to deadlocks is to identify the places where multiple locks are acquired. It is advised to avoid this where possible. Subsequently, the places where multiple locks are acquired should be investigated to check if common/consistent lock ordering is applied throughout the entire program. Minimising the set of code pieces where multiple locks are acquired and applying the open calls approach simplifies the analysis.

- Timed lock attempts: The timed `tryLock` Java Lock class method can be used as an example of an explicit lock. It gives the software developer the flexibility to define a timeout in case the thread waits "forever" for lock acquisition. This makes it possible to control situations where the thread waits. However, timed lock attempts also have a downside. If a thread that is acquiring a lock in such a fashion fails, the reason of failure may not be known. Nonetheless, timed attempt locking provides the chance to log the failure, release the lock, wait and attempt acquisition again. Timed lock attempts can only be used when multiple locks are acquired together. For alien and nested method calls it is not sufficient to just release the outer lock.

- Analysis of thread dumps: Approaches for avoiding deadlocks are to be considered and implemented by the software designer. However, JVM also supports the software developers in activities related to deadlocks tracking. Thread dumps can be useful while identifying when the deadlock(s) occurred. By providing thread dumps, JVM enables the programmer to see the actual stack trace of each running thread. The locking information is also included. It shows which locks were acquired and what locks are

attempted to be acquired. In the process of preparing the thread dump, JVM searches the wait-for-graph of each thread in order to detect cycles. The information included in a dump outlines e.g. what threads are involved and what locks. It is also worth mentioning that the thread dumps include information regarding locks acquired in the intrinsic way. When using explicit Lock class, the associated information is thread dump is less precise when in case of the use of intrinsic locking.

Liveness Hazards

- Starvation

  Starvation occurs when a thread's access to a shared resource is permanently rejected, so that the thread cannot progress. This is likely to happen if the code base has design faults. Using, for example, infinite constructs (e.g. forever loops) with an already acquired lock prevents other threads from obtaining the lock. Thread starvation may also occur if the thread priorities defined by the Java API are used. In the Java API, Thread class offers `setPriority()` and `yield()` methods, which are not advisable as the API defines thread priorities of a different range than operating systems and JVM may map API-defined priorities in a different way than the software developer expects (e.g. few API-defined priorities to the same number of OS level thread priorities). Applying thread priorities has the consequence that the application becomes platform-specific and is exposed to the risk of thread starvation.

- Missed signals

  A missed signal occurs when a thread waits for a condition which is already satisfied, although it did not succeed in checking the condition before. To avoid the condition wait, the condition predicate must be shielded by a lock. When a thread is waking up from the wait, the predicate condition must be verified again. It is recommended to perform such a check in a loop (waiting in a loop).

- Livelocks

  Livelocks occur if a thread is trying to execute an operation which always fails and therefore cannot progress. This happens when multiple cooperating threads change their state in response to others so that the execution of none of the threads can progress. A livelock traps threads in an unending loop of releasing and acquiring locks. Livelocks can be caused by code backing out of deadlocks. This hazard may be minimised by applying random latency between subsequent re-tries of the operation. A well-known use of this approach is collision detection in the Ethernet protocol, where random waits and back-offs are used to manage access to the communication medium.

6.2 Relational Database Optimisation

When accessing an RDBMS, due care must be taken to facilitate its efficient operation:

- Database behaviour is influenced by configuration parameters, although usually only few parameters impact performance. Default RDBMS settings are usually not ready for production use. Out-of-the-box settings are usually suitable for developer machines, not servers. Consider setting the appropriate amount of resource (memory, and other). Also, consider adjusting data integrity parameters appropriately for the application. Default settings are usually the highest possible which is not always

required. For instance, asynchronous commit with 200ms delay might be acceptable for some applications. Also for development purposes it is usually safe to turn off the storage synchronisation ("fsync") function, ensuring that all modified data is transferred to the storage.

- Adopt naming convention for database objects (tables, indexes, PK and FK constraints). It does not matter which one, what matters is that you apply it consistently. It is easy to identify PK and FK columns. Group related tables and codebooks with a name prefix. System and user codebooks should be differentiated. Use consistent data types, if the purpose is the same.

- Always use named parameters in the queries – do not concatenate user-entered values ("WHERE user.username="+username). Using named parameters allows RDBMS to cache queries and avoid unnecessary parsing, and also eliminates SQL injection vulnerability. For named parameters use JDBC class `PreparedStatement`, or EJB named queries.

- DB tuning is crucial to achieving good performance. Several people are involved in tuning: the designer – since the foundation of everything is good design; the developer – during testing but also often in production; the database admin monitors the work of the DBMS, notices bottlenecks and takes care of DB configuration; the system admin deals with OS and hardware performance.

- Consider using data grouping or denormalisation techniques to speed up reads, avoid JOINs or pre-calculated aggregated values (like SUM, MAX, MIN). If available, use materialised or indexed views. Otherwise think about denormalisation of logical data design by adding redundant columns, tables and indexes, as well as supporting constraints and triggers. Have in mind that these techniques increase write time.

- Weigh up horizontal portioning or sharding of large tables, especially if it is possible to establish such segmentation where typical queries could be answered by consulting a single partition or shard.

- Consider other acceleration techniques: archiving of obsolete data when possible, hardware upgrade, OS and database settings (startup parameters and DB objects physical distribution on files and disks, application of SSD, RAID, ASM – DB tuning in a broad sense), rescheduling batch jobs so that they do not collide, redundancy for load shedding (clustering, replication, separation of OLTP and OLAP database instances), changes in the business process.

- If possible, separate subcomponents on the file system that are accessed simultaneously to separate drives. For instance, it is usually wise to use one disk for data files and one for the write-ahead log (also called "redo log"). This will reduce storage seek time (i.e. hard disk drive head movement) and improve performance as well as the lifespan of the storage drive.

- Consider using SSD for the write-ahead log (WAL). WAL files are usually small but write intensive. SSDs are suitable for this.

- Use RAID intelligently. Usually Raid 0, 1, 1+0 are best suited for DB applications, although for some purposes other RAID configurations are also recommended.

- Use query- and performance-monitoring tools. Tuning workflow: find a problematic query, analyse and, optimise it. There are many GUI SQL profiling tools for all DBMS: Oracle Enterprise Manager, Toad, PL/SQL Developer, MS SQL Profiler. The alternative is to query the system tables and views that track the performance (which is a free DBMS feature).

- Perform tuning as much as possible in the test phase. If possible you should provide test data in realistic volumes. For a more realistic picture, a true distribution of column values is preferred as well as a simulation of a large number of concurrent users. In practice tuning in production is inevitable but the testing phase reduces workload and risk.

- Construct queries and configure RDMBS in a way that allows caching and reuse of execution plans and table and index data. When you send a request to execute a query, it checks if it has already been executed. If not, the SQL is analysed and objects located (tables and indexes, statistics on these objects). The query optimiser DBMS module plans the sequence and manner of access to objects. It also plans merging, filtering etc. which may be required to return the result, producing an execution plan. The plan is cached in the appropriate area of the DBMS server memory. After the query is executed results are returned as a stream. All data that operates in query execution must be in memory. These are blocks of tables and indexes that are cached and subsequently read from memory buffers or brought up to memory from disk.

- Observe used query optimisations and execution plans. Modern DBMS use the cost-based optimiser (CBO). The CBO tests permutations of sent queries and finds the permutation with the lowest cost. In the CBO cost estimation a number of parameters are involved. The number of permutations is approximately the product of the number of used tables and the number of WHERE predicates. Query transformations that CBO typically does are views to join transformation and subquery to join transformation.

- Be aware of the statistics that CBO uses. While searching for the optimal plan CBO takes into account the number of columns in tables, the number of distinct column values etc. The reason is to choose in which order tables will be merged (depending on the ratio of rows), or whether it is faster to access the table directly or through the index (for a small table direct access (full table scan) is often faster. In this decision CBO uses statistics for all objects that are periodically collected. For a good execution plan it is necessary that statistics are updated, which may need to be checked. Broken statistics might lead to a faulty query plan and significantly increased execution times.

- You can enforce plan stability: hints are given in the form of specific comments in the query text and they suggest to CBO a specific kind of behaviour: forcing merging algorithm, tables merging order, indexes usage, subqueries transformation etc.

- Data access recommendations: the retrieval speed largely depends on the number of table rows on which it is performed. If you have a query whose end result has a significantly smaller number of rows than the table on which it is executed, the unnecessary rows need to be eliminated as soon as possible (e.g. with JOIN or WHERE condition) For the same purpose, if OUTER joins are not mandatory they should be replaced with INNER joins.

- Limit the number of rows returned by the database, (e.g. by using `Query.setFirstResult()` and `setMaxResults()`) and use results paging.

- When accessing a database, avoid long lasting transactions that affect large amounts of data, as they greatly decrease database performance, result in increased connection pools and increase memory usage with object-relational mapping (as those provided with EJB or Hibernate).

- When performing batch processing of large datasets, commit after every single or few iterations.

- Use indexes on all primary keys, and on some of the fields used for sorting and searching. Keep in mind that indexes reduce read time, but increase write time. When writing indexes indicate the most restrictive column (high selectivity) first and then carry on in order of decreasing selectivity. Indexes speed up SELECT with ORDER BY clause (order of columns is essential), the WHERE clause (if it is partly or completely made up of indexed columns), the MIN and MAX over indexed column, the JOIN on indexed column and over columns with many different values. The quality of indexes depends on the high selectivity of the first column, a good column order, the table not being too small, the clustering factor and up-to-date index statistics.

- Subqueries: the subquery is executed before the query for that row; the most deeply nested subquery executes first. The transformation of subqueries to JOIN improves performance; subqueries in SELECT list are inconvenient because CBO does not optimise them. When manually transforming subqueries to JOIN watch out for duplication of rows if the subquery is not scalar. The WITH clause eliminates the use of several identical or similar subqueries.

- Follow SQL recommendations for improving performance:

  - SELECT COUNT (*) is faster than SELECT COUNT (1). SELECT COUNT (<indexed_col>) is even faster.

  - Use UNION ALL instead of UNION where possible.

  - Use NOT EXISTS instead of NOT IN – both process all rows of the subquery, but NOT IN sorts them.

  - Use UNION [ALL] to merge two queries instead of one with OR, as it allows the use of index.

  - LIKE with % suffix can use the index.

  - Use WHERE instead of HAVING.

  - In GROUP BY list the minimum number of columns needed for grouping.

  - Add to the WHERE clause some tuning conditions that do not change the semantics of the query, but allow some index to be used to isolate the partition (partition pruning) etc. It is good to mark them.

  - Specify only needed columns in the select list because of the size of the returned row and the chance that the result is returned only by accessing the index, by using index lookup only and without actually accessing the table.

  - For JOIN use the inner / left / right / full join syntax (this way the Cartesian product is avoided and queries are easily readable), use table nicknames (the origin of the column is clearer in conditions, avoided ambiguity errors and wrong bindings by the DBMS e.g. in the subquery WHERE clause)

  - Always close brackets to avoid errors in the combined OR and AND conditions: (CND1 OR CND2) AND (CND3 OR CND4), index cannot be applied if condition with indexed column is in one of OR's (e.q. (<IDX_COL> = :1) OR (<NON_IDX_COL> = :2))

  - Be careful with NULL. In conditions with =,<>,>: comparisons with NULL always return FALSE. Only 'NULL is NULL' returns TRUE. Arithmetic: 'NULL + 100' returns NULL. Aggregation results (COUNT always returns a row, COUNT does not count NULL occurrences, SUM of zero rows is NULL, SUM(...)OVER() throws "No data found" exception, SUM where all rows are NULL is NULL, SUM where at least one row is different from NULL is not NULL). The solution functions are: NVL, COALESCE, ISNULL, IFNULL.

  - Triggers: Use with caution and only if all other possibilities are exhausted.

- Good design can significantly speed up the execution of conditional structures and/or the execution of the query parts of both SQL and host application code. Also, bad design can slow down execution dramatically and jeopardise application code (heap, synchronisation etc.).

- Lower transaction isolation level when reading data that is rarely modified, as classifiers. Particularly if such results are not cached at applications level. To do this, use SET TRANSACTION ISOLATION LEVEL (SQL) or @TransactionAttribute (TransactionAttributeType.NOT_SUPPORTED) (EJB).

- Whenever possible provide filtering, locating, and sorting of objects and population of associations by using a database or persistency layer, not by explicit implementation within an application.

6.3 Optimisation of JPA/EJB Usage

JPA (Java Persistence API) and EJB 3.0 (Enterprise Java Beans), which encompasses JPA, are increasingly used to provide database access and object-relationship mapping (ORM) on Java SE (JPA) and EE (EJB) platforms. They are also often used in implementations of internal logic that is relatively independent from the presentation layer. EJB and JPA can greatly ease the development of applications that are dealing with complex data models by simplifying many common operations and reducing boilerplate code for handling individual data items. Their usage is also strongly promoted by their close integration within several popular development environments and frameworks, but they are quite inadequate for massive batch processing. Although EJB and JPA provide a high level of abstraction in database usage, various available options have a significant impact on performance. To make the right choice, the application's data model and data access

patterns need to be understood. Based on this knowledge, significant improvements can be achieved by selecting appropriate database access options [EJB].

- Use remote invocations of session beans only for coarse-grained calls.

  Software interactions based on remote calls should always be based on SOA-style decomposition and service paradigms. Otherwise, most of the time will be spent on invoking remote methods and passing objects over the network.

- Choose appropriate transaction management.

  With bean-managed transactions, only a part of a method can be executed within a transaction. With stateful session beans, it is even possible to start a transaction within one method and end it several method calls later. However, the developer must ensure transactional correctness of bean-managed transactions. A coarser control with container-managed transactions can be achieved by using session bean transaction attributes and separating transactional and non-transactional code into different business methods. You can move time-consuming I/O operations and database accesses implementing read-only "browse" scenarios into non-transactional methods or code segments. Also, splitting an action into several short transactions may reduce the cost of transaction isolation and serialisation and greatly increase parallelism. Nevertheless, the sequential code that uses several simultaneously opened transactions rarely produces the intended results.

  However, performing of a whole action with a single transaction may be safer. Also, pairing user-managed transactions with execution of servlets or JSP pages preserves the persistence context, which prevents problems with lazy fetching.

- Stateless session beans provide a better performance of single-interaction-based operations, but stateful session beans with an extended container-managed or application-managed entity manager provide faster multi-transaction conversations.

  Use stateless session beans to implement services and methods that are not significantly dependent on a previously established context. Stateless session beans typically use a transactional persistence context, which maintains the association with entity beans only within a single transaction. This results in a better performance if there is no significant need to reuse previously fetched entity beans. However, reusing entity beans that were retrieved earlier in a new transaction requires them to be fetched again by a new lookup or reattached to the persistence context as a side effect of the merge() operation. All this affects the performance and makes the code less intuitive.

  Stateful session beans support an extended persistence context which keeps the association between objects and the persistence context across transactions. This allows a simpler implementation of long multi-transaction user conversations. Although a similar effect can be achieved by using stateless session beans with application-managed entity managers, extended persistence contexts are simpler to use. [JAVAPER] provides a detailed discussion of available transaction and persistence context options.

  Finally, application-managed entity managers must be passed to the session beans and used in a thread-safe manner. They must be passed via method arguments or some thread-bound context, which requires significant code refactoring and makes this option unattractive for extensive use.

- Caching entity beans and query results can reduce the database load and traffic.

If your EJB 3.0 container supports entity caching, and there is no caching at the presentation level that already exploits most caching benefits, it is worth to cache frequently read but rarely changed entities and query results. However, the cached data is not refreshed upon external database updates, and there is no standard way to clear the cache. Implementations of EJB caches typically allow you to specify timeouts to cached entities and a maximum number of cached items [ENTB]. Cached items can be grouped into various regions. Depending on the implementation, entities to be cached and associated regions need to be specified by using either configuration files or Java annotations.

- Choose between optimistic locking, possibility of changes overwriting each other, or proprietary extensions.

  Optimistic locking, which is achieved by using the Version annotation, provides better performance than pessimistic locking if concurrent updates are rare or unlikely, even though it causes an additional read before the update. However, this is not the case in scenarios where several users concurrently update the same data, which results in a large number of rollbacks. These rollbacks are expensive, affect the user experience, and lead to later repetition of the same actions. Optimistic locking sometimes also requires explicit locks by using EntityManager.lock(). These locks, which are shared by all container's persistence contexts, are needed when the database does not guarantee repeatable read (READ lock), or when the entity is changed by its participation in relations which are not reflected in the table that corresponds to the entity (WRITE lock). READ locks also help improving the database performance by allowing lower isolation levels, with WRITE locks additionally enforcing the incrementing of version numbers.

  However, in EJB 3.0 there is no standard way to perform database-level pessimistic locking with JPA, although it can be done by using vendor-specific extensions or native SQL queries. EJB 3.1 and JPA2.0 support pessimistic locking by extending the EntityManager `find()` and `lock()` methods. [EM].

- Use a lazy or eager fetch strategy, depending on the pattern used to access to entity attributes and traversal to associated objects.

  Properly placed lazy fetching reduces the loading of unneeded objects, while eager eliminates the additional queries needed to later fetch the missing data. It is particularly important to use lazy fetching on rarely traversed relationships in order to eliminate massive loads of unnecessary objects. However, unloaded lazy attributes and relationships cannot be accessed in detached objects, so their usage requires a valid persistence context. It should also be noted that the fetch strategy has a strong impact on resulting database queries and thus significantly changes query execution plans, which may produce surprising results.

- Consider using JPA features that support direct access and control whenever the ultimate performance on a large database is needed.

  These features include bean-managed transactions and native queries that can execute fine-tailored and optimised SQL statements and even stored procedures. However, applying this practice to simple and non-critical queries and updates leads to portability problems.

7 Problem Management

ITIL defines problem management as a process that focuses on identifying and solving the causes of incidents to avoid adverse influence of these on the functionality and availability of the software.

It is important to remember that procedures defined in ITIL usually cannot be directly used in projects. ITIL defines management and development procedures on a very generic level that applies not only to software development, but also other technical projects (not necessarily IT). ITIL has been designed to relate not only to regular daily activities (like problem management, release management) but also to executive duties (e.g. strategy). ITIL differentiates between three types of possible problems:

- **Incident** – a fault or disruption which affects the normal operation of an IT service.
- **Problem** – a state identified from incidents that indicates an error in the IT infrastructure. A problem remains in this state until a cause is found.
- **Known Error** – a problem for which the cause is found and is often related to a fault with a configuration item (CI) or a number of CIs within the IT infrastructure.

Problem management approach can be both reactive and proactive. A reactive approach means solving problems in response to incidents, while a proactive approach identifies and solves potential incidents before they occur. A proactive approach assumes that potential failures and problems can be predicted on the basis of statistics and due to the experience and intuition of project managers and lead developers. They can minimise the risk of serious problems during the team building process (e.g. taking the personalities of particular developers into consideration), scheduling releases and selecting tools that will be in use.

The project manager, lead developer and release manager supervise development and deployment iterations. By collecting and analysing statistical data they can conclude which of these processes (planning, development, testing, releasing) are vulnerable to incidents and problems. On the basis of the knowledge derived from the historical data, they work out internal procedures (related to development, testing and releasing) to minimise re-occurrence of a particular issue.

The reactive approach to problem management is strictly related to the change management process, since the solution of the issue can result in a change of the system. In general, incidents can be split into following categories:

- Incidents related to implementation bugs.
- Incidents related to installation and configuration errors.

- Incidents related to lack of the documentation.
- Incidents related to unavailability of the service.

Sometimes an incident belongs to more than one of these categories (e.g. the lack of documentation results in installation or configuration errors).

User should report a problem to the point of contact (this can be a bug tracking system, like JIRA, or a mailing list). Anytime a problem is raised, it has to be assigned to one of the categories. Due to the change management and bug reporting procedures, an appropriate priority and developer responsible for delivering a solution should be assigned to each problem.

If a reported problem relates to the unavailability of a service (e.g. a failure of the database or application server) or incorrect configuration, no changes to the system are needed and change management procedures are not used. In this case, the point of contact redirects the issue to the support team. Depending on the structure and internal procedures of the support team, it can provide online (interactive help or remote monitoring of the application) or off-line support. The team helps to apply the correct configuration or contacts server administrators to bring the service back to use as soon as possible. The support team can ask the reporter of an incident to provide detailed information about the affected infrastructure (both hardware and software) and all required records of the system usage (e.g. log files, monitoring parameters, etc). Based on this information, rescue procedures are launched (e.g. system reboot, database recovery, tuning and configuration). Every serious failure should be wrapped up with a review of applied procedures, so the support team can improve its performance.

Problem management procedures assume that, besides providing a solution for the problem, the development team works out new development procedures (or refines and improves existing ones), and releases and documents the software. The feedback obtained results in a proactive approach to the problem and in minimising its occurrence in the future.

7.1 Acceptance Criteria Based Software Evaluation

Every new software system passes through several phases of maturity [OPF]. The most important milestones are test completion and the handover of the system or its improvements to end users. Controlling transition between phases improves the predictability and quality of IT developments, helping to plan and manage load and commitment during implementation and support.

From time to time (at least before the official release) acceptance tests should be performed. These tests do not focus on finding new bugs, but aim to confirm that the software's quality is adequate. There are alpha and beta acceptance tests. Alpha tests are performed by internal testers: software architects, analysts and developers. Although it is useful to provide these testers with coarse functional scenarios in terms of goals, it is important to let them perform a freestyle exploration so they can gain a fresher perspective of the software. After solving the critical bugs found during this stage, beta tests are performed. A beta version is delivered to the public. New bugs that are found during acceptance tests should be solved (or at least planned to be solved in one of following versions) before an official release.

There are many criteria that could mark the end of testing:

- Testing and release deadlines are reached.
- The test budget is spent.
- The bug rate falls below a certain level.
- The defect discovery and resolution rate have converged.
- The number of successfully passed test cases reaches a specific percentage.
- The test case coverage is met. The code, functionality and requirements coverage by tests reaches a specified level.
- Zero bug bounce ripples have settled.

Zero bug bounce marks the moment when the number of open defects, excluding those with the lowest significance, reaches 0. After that, some ripples in the chart depicting defects in time are still very likely, but the magnitude of these ripples normally tends to attenuate, indicating that that the application is becoming stable. It should be noted that the first bounces in the number of bugs can appear even before reaching zero. After the ripples are settled, the software is mature enough to start its further extension or major refactoring. The possibility of diminishing returns should also be considered, where the cost for addressing the remaining bugs increases while added value decreases. At some point it becomes more effective to accept the cost of the remaining defects than to work on them.

The safest practice is to establish a specific suite of black-box tests indicating an adequate quality for the software. When a software project reaches its final acceptance term or release date, its functionality and usability is verified by performing this suite, called acceptance test, which measures the severity of software defects and their impact on users or customers. The selected issues emerging from code reviews, regular tests and actual usage are consolidated by running a selected suite of tests under operational conditions or similar.

In commercial projects, the results of these tests are often verified through agreed and duly documented formal acceptance procedure. Passing acceptance test is often a crucial element of meeting contractual or regulatory obligations, but in the agile lifecycle it is a good practice to perform it before each release. It is important to clearly define in advance what is actually checked or measured by specific tests and what expected or acceptable results are. The audit procedure, tests included in the acceptance suite and pass criteria need to be developed and agreed early on in order to adequately capture the key utility and features, and minimise friction between users and developers. For a discussion of acceptance test case development, their specification, documentation, processing and closing, see [TexasUni].

Controlling software development and maintenance milestones through the described procedure provides a point of synchronisation for the project team, milestone deliverables, and user expectations. Application of this approach results in overall accountability of all involved stakeholders and the whole project.

The discrepancies detected by acceptance test are marked with different levels of severity. Virtually all common classification examples use four levels, although the terminology may vary [Level]. Furthermore, an additional level can be used to describe any issues that should be excluded from the acceptance test.

- **Critical** – the problem has an extreme impact on the system, effectively blocking its usage.

  - The system hangs or cannot be used at all; its key capability or feature is inaccessible; data (database, file) is lost or corrupted.
  - There is no available workaround or alternative; the system is unusable even after a restart.
  - The problem is seriously or critically impacting user operations.

- **Major** – the defect is severely restricting system usability.
  - The system can be still used but its functionality is significantly limited regarding usability and the performance of important user functions. Some important capabilities are lost, it does not work as documented or expected and produces incorrect, incomplete, or inconsistent results. Errors are not reported, key functionality or its limitations are badly documented and no guides are available for clarifications.
  - There is no easy workaround to make the feature or component functional again, but there is an alternative which allows results to be obtained and essential operations to be continued.
  - There is an essential impact on user operations which are disrupted but still assisted by the system.

- **Medium** – the defect does not prevent important functions of the system to be used.
  - There is a loss of non-critical functionality which may cause major inconvenience and loss of flexibility. There is a loss of some non-essential but still important capability. Error conditions are reported incorrectly. Important features are badly documented, but there are some alternative guidelines.
  - There is a known simple or easily reachable workaround which allows normal operations to be continued.
  - There is moderate impact on everyday user operations.

- **Minor** – the defect does not interfere with the use of the system and user operations.
  - Annoyance which does not affect functionality or capability in general usage. A problem with a minor functionality that is not crucial for basic functioning. The system is operational with a small inconvenience in usage. Requests for enhancement of working functionality or improvement in usability or flexibility. Non-conformance to a standard which does not affect functionality. Issues with appearance, consistency, aesthetics, terminology or spelling. Inadequate documentation or insufficient error messaging.
  - Desired results are easily obtained by working around the defect.
  - There is no impact on user operations, or the impact is negligible. The defect may be deferred or even ignored.

- **Not relevant** – not actually a defect, but there is some possible impact on the quality or perception of the system.
  - A remark whose resolution is irrelevant. Requests for enhancement or features that are outside the scope of the system and its target usage. Defects that cannot be solved or tested, for example, if the correct behaviour cannot be articulated.
  - These reports are not subject to acceptance evaluation, but may still provide some inputs for change management, release schedule or feature planning.

Both the defect's severity and the results of the analysis of its technical aspects (which should also include an estimate of resources and time needed to resolve it) are inputs for change management and release planning. If there is no first-line helpdesk, issues can be tracked in a bug tracking system. Priorities are often used to integrate the estimated severity and assessed feasibility, and to address costs. But even when issue reporting, support and testing are done using a bug tracking system, it is advisable to separate the reporting of severity and dealing with prioritisation. Independent tracking of bug severities and priorities allows more sophisticated workflows to be implemented and to overlay acceptance test tracking with bug tracking. This approach also allows responsibilities to be separated, so that different individuals may be responsible for evaluating defect impacts, deciding in which order solutions should be applied, and resolving assignments. While defect severity is related to the impact on functionality, the priority is associated with solution scheduling. To clarify this distinction, here are some illustrations of difference between severity and priority:

- High priority, high severity – All critical and most of major defects which have a great impact on user operation and compromise system usability.
- Low priority, high severity – Even crashing of the system can be assigned low priority if that happens extremely rarely and under some very specific circumstances.
- High priority, low severity – Assigning high priority to technically trivial and functionally irrelevant issues improves user satisfaction. Examples include GUI glitches, inappropriately displayed logos, spelling errors (unless they mislead users to incorrect actions).
- Low priority, low severity – There are some small defects that are usually ignored or stay unnoticed for a long time.

The use of severity levels becomes more complicated if they are not only used for internal acceptance evaluation, but also as discrepancy categories associated with user- or client-driven acceptance procedure, SLA, response time or resolution commitments. Since these categories describe impact on end users, they are defined in SLA documents or contracts and associated with strict deadlines. This leaves little options for free prioritisation by the support or development team and effectively merges severity and priority into one. The only option for controlling impact of severities on support and development teams is to differentiate response times according to support levels, SLA ranges, or system or product phases (alpha, beta, RC, production release, finalised).

If this is the case, it is advisable to further specify and agree on ranking procedure and criteria. It is recommended to specifically number all common defects in order to keep the users' view of the defects and expectations aligned with the above described severities. These system- and application-specific descriptions could be grouped into several areas, for example, usage preconditions and assumptions (like operating environment), ability to perform particular use cases, user interface, performance, documentation. Items within each of these areas could be associated with different acceptable severity ranges, but the vast majority of explicitly listed items are likely to be associated with medium and minor severities. After such descriptions are mutually agreed, the impact and severity classification and corresponding prioritisation will become simpler and less prone to debate.

After the acceptance suite elements are checked or defects classified, the number of discrepancies in each category can be used to verify whether the desired level of maturity is reached. The orientation values of acceptance thresholds are given in Table 7.1, but the actual numbers depend on the mission criticality, actual

phase, number of tests in the suite, and whether all active bugs are also counted. The actual numbers of unresolved defect thresholds should be agreed upon in advance by all relevant stakeholders.

Maturity Phase	Critical	Major	Medium	Minor
Alpha	0	≤ 4-5	N/A	N/A
Provisional Acceptance, Release Candidate, Beta	0	≤ 3-5	≤ 15-25	N/A
Release, Final Acceptance	0	0	≤ 3-4	≤ 5-8

Table 7.1: Acceptance thresholds

If threshold-based quality checks are to be performed, but the acceptance tests are not formalised, the number of bugs (classified by severity) can be used instead. No serious bugs should be passed to beta tests, since they should all be caught and repaired in previous testing and releasing iterations. In a perfect approach all bugs found during the alpha phase should be fixed before the beta version is released and corresponding tests are launched. Nevertheless, if some major defects are forwarded from the alpha, it is harder for the beta to pass the provisional acceptance. Similarly, all critical and major bugs found in beta tests should be fixed before a final version of the software is published. Bugs that cannot be fixed because of complexity or lack of time should be considered separately. Minor issues can be omitted, but defects issues (if present) must be fixed, postponing the final release, if necessary.

Errors or malfunctions should be documented in the acceptance protocol and corrected by further development, maintenance, and support activities. Upon completion of corrections, they are again subject to testing.

8 Bug Tracking

Bug tracking is a process of reporting and tracking the life cycle and progress of bugs and defects in computer systems from discovery to resolution. A software bug is defined as an error, mistake or fault in computer software that causes it to produce an unexpected or unintended result. Bug tracking allows developers to improve software design, make continual changes or upgrade products.

Bug tracking might be accomplished by asking users to report any difficulties via email or phone but in larger programs involving high number of programmers or end users such approach might be very ineffective. A bug tracking system is a software application that is designed to manage and maintain list of bugs in order to help quality assurance and programmers keep track of reported software bugs.

Bug tracking is crucial in the coding and testing phases of the software development process. However, bug tracking systems are not limited to tracking bugs, but are also used for general issue tracking, help desk situations, tracking feature requests, improvements and inquiries.

Bug tracking can improve communication and customer satisfaction, raise software quality, the productivity of the development team and reduce software release time, and therefore costs.

8.1 Bug Reporting

The following summarises Bugzilla's best practices for reporting bugs [BUGZ]. Although Bugzilla's naming conventions are used, the principles behind these best practices apply to any bug tracking/ticketing software.

Any person that is involved with the software in question, be it a developer, end user or a manager, should be able to report a bug found in software or to submit a feature request. All problems encountered in software should be reported in the bug tracking software, no matter how small they seem.

The purpose of a bug report is to let the developers see the problem with their own eyes. When a user notices the problem, s/he should report it as soon as s/he can in order to write a more effective bug report. Team members are likely to dismiss reports of problems that they cannot see or reproduce. Users should also be reminded to check if the bug is reproducible (and if it isn't, then this should also be indicated in the report).

A bug report should be regarded as a type of communication with the developing team behind the software, as well as a permanent record of a bug resolving process that could be viewed by anyone, if the management team makes its bug reports available to the public. The language used in a bug report should be formal and its

tone should be polite. The entire bug report should be read before it is submitted – the more precise and concise it is, the better the chances that the developer will understand the error or the requested enhancement.

The writer of a bug report should not propose his/her own solution, unless they are one of the developers.

8.1.1 Bug Report Principles

The following are some simple principles that if honoured can produce a better bug report:

- Be precise.
- Be clear – explain the bug, so others can reproduce it.
- One bug per report.
- No bug is too trivial to report – small bugs may hide bigger bugs.

8.1.2 Preliminary Steps

Before you report a bug you should:

- Reproduce your bug using a recent build of the software, to see whether it has already been fixed.
- Search the bug tracking software, to see whether your bug has already been reported. Duplicates waste time and confuse the tracking process. Always search the bug database first. Details about the bugs usually come from the database, (for end users, they are free of internal details relating to developers and testers). Known bugs are presented in the published version of the software, contained in a static document that is useful to end users, and are described with as much detail as is necessary. On this basis, it should be possible to determine whether the problem is already known, and review recommendations on how to avoid reporting new bugs if not necessary. If bug report already exists, more data can then be added to the report.

8.1.3 Tips and Guidelines for Completing a Bug Report

- **One bug per report** – Report each bug in a single report. Do not include multiple problems in one report. If more than one problem needs to be reported, separate reports should be filed to keep the issues distinct from each other. Multiple bugs can result in multiple resolutions, which complicate the bug reporting process when written as a single bug.

- **Bug classification** – Reports should be categorised according to problem type. Classification is software dependent, but bugs should be placed in a listed category, depending on the software that is used.

- **Bug summary** – A brief, but meaningful, one-sentence summary of the encountered problem. The summary should be as descriptive and specific as possible, but brief. The bug should be described in

approximately 10 words that will quickly and uniquely identify the report. The problem needs to be explained, not the solution.

- **The report should answer three basic questions** – What did you do? What happened? What did you expect to happen instead? A bug report should always contain the expected and observed results. Include what was expected to happen, as well as what actually did happen to avoid the developers thinking that the bug is not a real bug. When you report a bug be precise, write clearly and make sure report cannot be ambiguous and misinterpreted. Be brief, but include important details.

- **Steps to reproduce the problem** – this is very important. Developers need to be able to get to the problem in the shortest time possible. Steps to reproduce the bug are the most important part of any bug report. The bug is very likely to be fixed if a developer is able to reproduce it. lTry to reduce the steps to the minimum, most logical necessary to reproduce the problem, as this is extremely helpful for the programmer.

- **Environment –** Often the environmental details matter (platform, OS, version number, build number, browser you are using (for web applications) when reporting a bug.

- **Priority / severity** – Before setting the severity rating of the bug, analyse its impact. Severity is the level of impact or consequence of the bug to the end-user, organisation, service, etc. It can be low (cosmetic issue), medium (feature is malfunctioning), high (unexpected fatal errors) or urgent (the software will not run). The priority classification of a software error is based on the importance and urgency of resolving the error. The priority classification is as follows:

    - Low (bug can be resolved in a future major system revision or not be resolved at all).
    - Medium (bug should be repaired after serious bugs have been fixed).
    - High (bug should be resolved as soon as possible),
    - Immediate (bug should be resolved immediately).

  The severity of a bug may be low, but the priority is typically high. Frequency is the likelihood of the bug occurring or manifesting itself to the end-user, an organisation, service, etc. Given the above definitions, it is easy to see why the priority would best be defined as a result of understanding the combination of both severity and frequency of a bug.

- **Include background references** – If a bug is related to other bugs, this information should be included in the relevant bug reports. This will help everyone who reads the report to gain a better understanding of the issues.

- **Screenshots –** These are an essential part of the bug report and should be included as attachments. Bug reports need to be effective enough to enable developers to reproduce the problem, and screenshots are very useful for verification and clarification of the bug report. Screenshots should not be too heavy in terms of size (.jpg or .gif formats (not .bmp). Annotated screenshots are also helpful to pin-point problems. This will help the developers to easily locate the problem.

- **Logs** – These will help developers to analyse and debug the system. Logs should be included as attachments. Content will varies by product, but normally there is quite a lot of information that might be useful for debugging in a log file or core file.

- **Contact** – Information about the bug reporter, including contact details, should also be provided, ether by the submitter or the bug tracking system.

8.2 Dealing with Bugs

The bug status changes throughout its lifetime. Although the naming conventions of various bug tracker systems may differ the main stages of a bug's lifecycle can be distinguished as follows:

- New

  The bug is reported.

- Analysed

  The bug is analysed and prioritised according to impact and estimated effort.

- Assigned

  The person who is responsible for the bug resolution is chosen.

- In Progress

  The bug is being fixed.

- Resolved

  The bug is resolved by the assignee but it needs further testing to be closed.

- Verified

  The bug is tested, but not yet submitted to the production code.

- Closed

  The bug is resolved and tested and the solution is delivered in the next version of the software. The bug entry could be reopened if needed.

The actual bug lifecycle and related workflows in many bug tracking systems can be customised to suit the needs of a company, organisation or development team, although it is advised to keep the workflow as simple as possible. For example, an organisation that uses peer review of the software may incorporate a Review status that is placed between the Resolved and Verified status. The idea is to send the solution of the issue to peer developers who would review it before testing takes place. This has proved itself as the easiest way to find errors in the code early on, and to enforce a uniform style of coding.

When the bug is in the system, the project leader, product manager or another selected person analyses the bug entry, verifies its impact and severity and adds who is responsible for its resolution, the bug's priority and

which software version number the bug should be fixed for. The importance and order of bug resolution depends on the last two factors.

After the bug is assigned to a developer their responsibility is to evaluate the bug. They analyse the bug entry, determine if it can be reproduced and find the possible cause of the bug. During this stage the bug can be reassigned to another developer. It is recommended to assign the bug resolution to the person directly responsible for the required functionality. When the bug resolution requires redesigning software architecture or major changes to the code or functionality, it is useful to reschedule the bug fix to a later version (for example, next major release). The bug could also be rejected as invalid. However, before changing a bug's entry state to invalid, it should be carefully analysed if the bug is really invalid. If any clarifications are needed it should be instead be reassigned to the bug reporter. If a bug is rejected the cause should be provided. The main causes of bug rejections are:

- The bug cannot be reproduced (for example, because not enough information is provided).
- The bug entry is a duplicate (it already exists in the system).
- The bug is not a bug (the system works as expected).

As the correspondence about the bug report continues, the participants can also attach documents (a test case, patch, or screenshot of the problem) using bug tracking software.

After the bug is resolved and tested by the developer its status is changed to Fixed and it is assigned to QA or the bug reporter (depending of the software development model) for testing. If the system behaves as expected, the state of the bug entry can be changed to Verified. If not the bug is reassigned to the developer.

Once a bug entry's status is set to Verified, the fix can be incorporated in the source code and can be released with the new version of the software or separately as a source code patch or update to the previous versions of the software. It is the responsibility of the bug tracking coordinator to search for all identified fixed issues and new features that have a Verified status in order to produce and maintain a change log of the software. After the software has been delivered to customers, the coordinator changes the status of all Verified bugs and features to Closed.

8.2.1 Bug Tracking Software

The following summarises the information provided at [BTRACK].

Using a bug tracking software system is the optimal way to deal with bugs. Using a specialised system allows developers, for example, to search for all bug reports that are assigned to them, project leaders and coordinators can manage open bugs and either close them as invalid or duplicates, or assign them to developers, and users of software can use the bug tracking system as a knowledge base to see if their problem has been already resolved or is being worked on.

For these reasons it is important for the specialised bug tracking software to include the following features:

- Assigning unique IDs to each bug/issue report.

- Extensive customisability – an ability to add custom statuses, severities or priorities (numbers, letters or words) to the bug work cycle, and an ability to add custom fields to bug reports and similar (for example, classifiers like those listed in *Handling Review Results* on page 37).

- Ability to add attachments to bug reports.

- Ability to review the entire history of conversations and status changes within a bug report to easily determine who changed the status of the bug report, and when.

- Customisable reports – an ability to create reports based on any number of criteria, for example keywords in the bug summary, persons assigned, statuses, severities, priorities or a combination of all of these.

- Web interface – allowing the team or customers to access the tool, regardless of which web browser they are using.

- User-based permissions – ability to limit the access of some users or user groups to some features or some types of bug reports.

8.2.2 Bug Tracking Guidelines

Some of the following summarises information is provided at [BTBP].

To deal with bugs successfully and easily, the roles involved with bug resolving should follow these recommendations:

- Developers should exercise patience when dealing with bugs. It is better to assign a bug back to the reporter and ask for clarification than to dismiss a bug report. While no developer likes being told their code is buggy, they should not be defensive, but verify every bug report before dismissing it outright.

- Developers should take advantage of the bug tracking software and its specialised features. Using filtering, reporting and organisation features in the bug tracker will keep developers' focus on a specific set of issues.

- Developers who work on a solution of the bug reported should never close issues. Depending on the adopted and agreed bug resolving process, only people who opened issues can close them, or possibly managers/coordinators of the bug tracking process, and only after the solution has been tested and verified and, depending on the process, delivered to the customers.

- Project managers should resist the temptation to keep adding new fields and metadata to the bug tracking system. It should be remembered that the focus of the bug tracking system is not to store large amounts of information but to enable the team to fix issues.

- Keeping the workflow simple is essential to getting an issue from an opened state to a closed state in the fastest time possible.

- Developers and testers should be trusted to make the right decisions. Manager should avoid executing too much control. Setting up complicated permissions and repeated approval/review processes may slow down bug resolution.

- Tasks or issues should be kept small and achievable.

- Project managers should not abuse reporting features.

8.3 Monitoring the Issue Handling Process

Observing how issues, improvements, and new features are handled and followed within the issue tracking software can provide an important insight into the work of software development and support teams. Periodic reflection on the state of tracked issues and trends can reveal a lot about how the software process is managed. Tools such as JIRA offer rich reporting capabilities that make such deliberation possible. However, in order to gain the full benefits, it is necessary to integrate issue tracking with release management, maintenance, and development plans. Some basic areas that may be addressed by this analysis include:

- Actual usage of issue tracking system – if the development and support teams are actively using issue tracking, or are relying on other, informal communication channels.

- Trends in the numbers of new issues, and how these trends relate to software releases. Usually, the highest peaks in created issues relate to the integration of major additions or changes before key releases and after user catch-up and feedback.

- Whether the issue addressing process is converging, that is, if the issues are reported more quickly or solved.

It is quite useful to track these trends by charting the created vs. resolved issues on monthly basis. Another important chart shows the same data, but in incremental manner (Figure 8.1), which allows the visualisation of whether the gap between created and resolved issues is growing or shrinking. Although this gap should decrease with maturity and stability of software, it is quite common to have a group of residual issues that remain unsolved or ignored, causing the average age of unsolved tickets to grow linearly. This is caused by the presence of issues that remain unsolved for long periods, as they are related to unclear or difficult problems, or their addressing depends on some precondition. The ratio of such tickets among active ones can be estimated by dividing the increase of average age of unsolved tickets by duration of observed period, if the complexity and solving rate of newly reported tickets is constant.

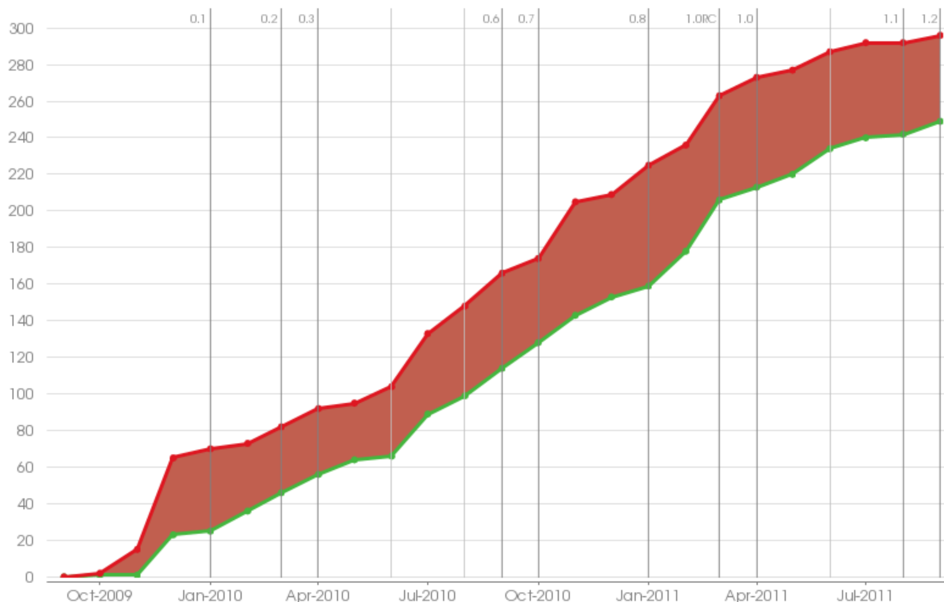


Figure 8.1: Cumulative created (red) vs resolved (green) issues in JIRA

On the other hand, the increase of average resolution time for issues solved during the period may mean that:

- New issues are becoming more complex,
- New issues are being resolved with less enthusiasm.
- Long-standing issues are finally being resolved.

The actual reason for increased resolution time may be distinguished by analysing the distribution of age of solved issues and looking into individual tickets. Solving a couple of issues that are a few years old can significantly affect the average age, while the lack of the dedication can be recognised by assessing the complexity of less-persistent tickets. A small population of resolved issues during monthly (or even trimestral) periods and lack or unreliability of corresponding initial effort estimates require per-case analysis. The final judgment must be made by observing the individual representative tickets.

In order to ensure convergence and clarity, long-standing issues should be continuously addressed or, if unclear or irrelevant, periodically reviewed and discarded or otherwise retired. If there are a significant number of such issues, the team should review them and close those that are irrelevant or practically solved, but without formal closure. After this, the developers should also become more agile in handling and closing of problems.

Another factor that may cause an increased number of issues and extension of their lifespan is the integration of agile process management with issue tracking through the use of tools such as the GreenHopper plugin for JIRA. This results in the creation of a number of new, long-living project backlog or feature wish-list items. It is therefore important to compare the age of issues in various categories. The charts showing the distribution of issue types and age of unresolved tickets on per-type basis are very useful for this analysis, as well as to validate procedures for individual issue types. Issues related to bugs and small improvements should be the shortest lived, while the lifespan of those associated with tests may depend on actual testing schedule. Issue

types such as General, Task, and Development (in JIRA/GreenHopper terms) are not sufficiently specific, and should be avoided. It is better to use the nomenclature of issue types which clearly associate them with responsibilities and specific phases in the software lifecycle. Alternatively, issues with longer life expectancy should be associated with appropriately high-level or long-term types, such as Epic, Story, and Research.

Another useful feature of JIRA is the ability to create issues that are subtasks of other issues. Although this feature should not be overused, it provides a significant opportunity to associate related tasks, as it facilitates planning and tracking of complex tasks in a larger view. The use of subtasks is also useful when resolving issues that trigger separate, but related work, such as updating of internal or user documentation. However, standard recurring sequences of subtasks should not be modelled by using this feature. For example, it is better to consider testing as a mandatory stage in issue solving workflow than to require creation of Test subtasks. Issues of test related types should be instead used to report missing unit or other automated tests or to request dedicated manual testing by QA personnel.

The need for such complex monitoring calls for setting up a clear policy and instructions covering issue handling workflow (with status changes) and classification of issue types.

9 Using QA Testbed Infrastructure

The main purpose of QA testbed infrastructure is to deliver a unified and scalable testing and demo environment to GN3 development teams. This testbed consists of a set of virtual machines serving particular teams, and a set of machines shared between teams and booked on a calendar basis. The virtual machines are supposed to be used primarily for testing purposes, including distributed testing scenarios, integration and system-level testing, as well as user acceptance testing, thus becoming a valid part of the GN3 software development life cycle. The virtual machines are dedicated to development teams, and those machines from a shared pool can be also used for non-functional testing (performance, stress testing).

The QA testbed environment is delivered and managed by SA4, Task 3. Information on the QA testbed infrastructure and its capabilities can be found on the SA4, Task 3 intranet pages, listed in Section 9.3.

9.1 QA Testbed Procedures

The process of requesting resources within the QA testbed infrastructure is managed via the GN3 Service Desk for software development Infrastructure. The service desk, based on GN3 JIRA, provides the GÉANT QA Testbed workflow (QATB), where the user can request virtual machines of described capacity. The VMs can be requested as:

- A fixed resource (fixed VM), permanently assigned to a development team.
- A temporary resource (testing VM), assigned to a development team in a requested time frame.

In particular, the Help Desk service is also supposed to allow for the following actions related to QA testbed infrastructure:

- Creating the virtual machine with well-defined resources, features and operating system (Ubuntu or RHEL).
- Running the VM in the requested timeframe.
- Suspending the VM outside of a requested timeframe.
- Closing or deleting a particular VM.
- Creating a snapshot of a particular VM in a particular state.

- Loading/reloading VM from a particular snapshot.
- Modifying resources of a particular VM.

The Help Desk is supposed to implement the workflow depicted in Figure 9.1 to handle QA testbed-related requests.

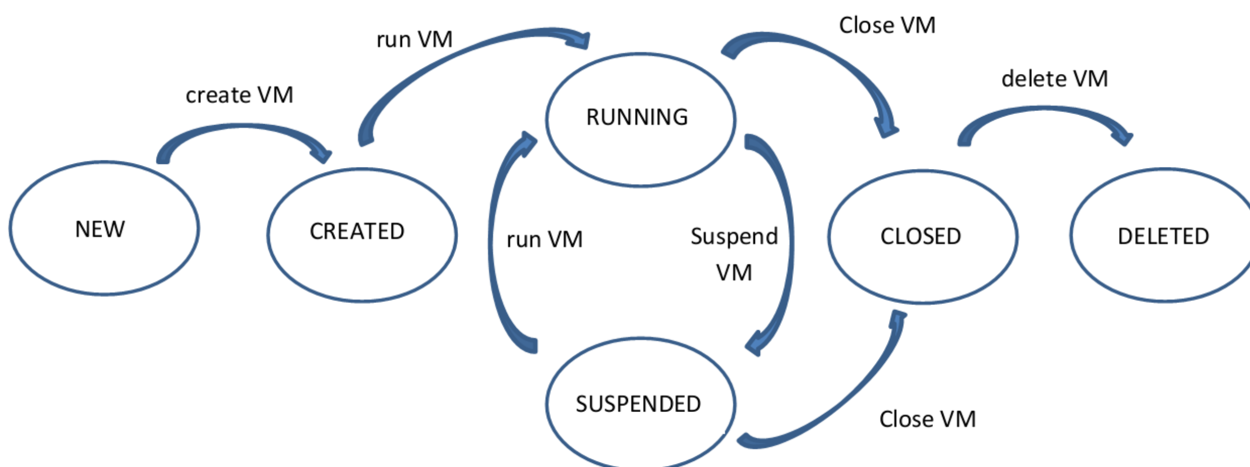


Figure 9.1: QA Testbed resource requesting workflow

While a single JIRA issue is related to a particular VM and all the actions related to the VM are linked with this issue, the issue sub-tasks are used to model secondary actions, such as creating a VM snapshot, (re)loading a VM with a new OS instance or given snapshot, rebooting VM or deleting a particular snapshot.

Further details on procedures, capabilities and environment limitations can be found on the SA4, Task 3 Intranet pages related to the QA testbed, described below in *QA Testbed Documentation.*

9.2 Booking QA Testbed Resources

Requesting QA testbed resources from the shared pool is based on availability, as tracked by the QA testbed calendar. The booking calendar shows the timeframes when VMs have assigned the testbed resources and can be up and running. For the end user, the calendar is read-only and plays an informative role. Only the testbed administrators are in position to introduce changes in the calendar. The time windows marked in the calendar reflect the requests that have been processed and approved, however, it does not show the requests are queued or have been rejected. The QA Testbed calendar is available at the QA Testbed Intranet pages.

9.3 QA Testbed Documentation

The main point of information for the QA testbed is the Intranet which directly links the user to the JIRA-based request system and QA testbed calendar. It can be found at the following locations:

- QA testbed Intranet pages, including the QA testbed calendar: https://intranet.geant.net/sites/Services/SA4/T3/Pages/Gn3QATB.aspx.
- QA testbed resource requesting system: https://issues.geant.net/jira/browse/QATB.

10 Closing SW Projects

The project closeout is the last phase of the software development. The closing phase can be initialised when the software recipient (user or customer) accepts the final deliverables of the project, and the supervisory board concludes that the project has met expected goals (if a project does not meet these goals, it can be terminated before completion). Until all stakeholders (project management, project supervisory board, user / customer) agree on these facts, the project closeout (which in fact focuses on administration and logistics) cannot begin. Once the agreement is achieved it must be revealed to all parties involved in the design, development, testing and usage of the software.

The project closeout must include the following steps:

1. Provision of the project deliverables to operations.
2. Redistribution of utilised resources (staff, facilities, equipment, premises).
3. Closure of financial accounts.
4. Archiving (completing and collecting if needed) of project records.
5. Documenting the project's success.
6. Documenting lessons learned.

Although the activities during the project closeout phase seem to be quite formalised, they may be important to other projects. While steps 2, 3 and 5 are related to an organisation's administrative, financial and marketing / PR departments, steps 4 and 6 may be beneficial for other development teams and for the whole community involved, even indirectly, with the project.

The closure of financial accounts and redistribution of utilised resources needs to be carried out as soon as possible. Financial closure (both external and internal) is an accounting process in which all expenditures are reconciled. When financial closure is completed, all expenditures have been paid as agreed in purchase orders and other contracts. Personnel that had been committed to the project needs to be returned to the resource pool immediately. This decreases the risk of other projects failing due to lack of resources. Before the personnel involved in the project is released or transferred, the project manager must ensure that the employees return all project materials and property. Generally, it is very important that resources are always fully utilised and report their efforts in a consistent way.

Other (non-human) resources used during development also need to be returned to the resource pool as quickly as possible. These include premises, cars, development infrastructure (servers, virtual machines etc.) and test infrastructure (both hardware and software). Only resources required to maintain the project results (e.g. available services, web pages, call centres) should continue to be used.

The closeout of all commercial and most scientific projects entails legal and financial liability. The closeout process is conducted to evaluate the acceptability of the project's deliverables (regarding completeness and quality). It also addresses each project's contract and involves administration activities, e.g. completing any open requests and issues, updating records etc. Incomplete requests need to be addressed, since these may be subject to litigation after the closure and entail financial consequences. Therefore, the closure procedures should be described in the contract's terms and conditions in detail.

Once all requirements are met and all necessary open issues agreed, the customer's representative should provide the project contractor with a formal written notice, stating that the contract has been completed. After the project's completion, the contractor is responsible for the correctness and availability of the project results to the extent agreed in other arrangements.

Project Success

Once a project has been successfully completed:

- It may become the basis for another project.
- Its results (like binaries, installers or source code) may be made available to users.
- It may turn into a service that is provided to users.

The first possibility is very common in research, where large projects very often consist of a number of smaller ones. Due to ordering limitation and time constraints, it is very important to provide deliverables on time. The delay of any stage can lead to serious problems for subsequent stages that are dependent on its results. This can lead to failure even if the original project is accepted as a success.

Once the development is completed, users may download available installers / binaries and launch their own instance of the software. The daily duties of the development team are usually limited to operating the project site and providing support to users. Depending on the contract, developers may be obliged to prepare regular improvements on the basis of user feedback. New versions containing improvements are usually published as updates and require noncomplex interactions in order to be installed.

Since a lot of software products are available as a service, users often do not have to launch and operate their own software instance, but consume the results provided by an external service. On the basis of the Service Level Agreement (SLA), the provider commits themself to operate the service and make it available and operational within the agreed timescale. Updates and new versions are provided with cooperation of the provider (maintainer), so that usually no additional actions are required from users.

Project Failure

It is rare that a contract is terminated early. Even if an early termination is the result of a mutual agreement, the appropriate termination clause in the contract should be applied. Procurement terms and conditions usually give the client the right to terminate the contract (or a part of the contract) at any time. However, the contract

may stipulate financial (or equivalent) compensation for any work that has been done and accepted (including preparation and initial costs).

Closeout decisions (e.g. agreements on incomplete requests and issues) should be taken based on negotiations between the involved stakeholders. If a settlement based on a negotiation is impossible to achieve, mediation and / or arbitration may be applied. Where these measures fail, litigation (although expensive and time consuming) seems to be the best option.

10.2 Project Closeout Reports

To provide the most valuable information, a series of reports can be delivered at the end of closeout phase. VITA (Virginia Information Technologies Agency [VITA]) recommends the project closeout to be finalised by three reports: Project Closeout Transition Checklist, Project Closeout Report and Post Implementation Report [VITA1].

10.2.1 Project Closeout Transition Checklist

A transition checklist verifies if a particular project phase has been completed, before the next phase begins. In the last project development phase, the Project Closeout Transition Checklist is supposed to verify if the project has met the acceptance criteria. This checklist contains a series of questions. Later on, a status and comments are assigned to each question. The status can be one of three values:

- Y – indicates that the item has been addressed and completed.
- N – indicates that the item has yet to be addressed for the process to be completed.
- N/A – indicates that the item has not been addressed and does not apply to this project.

The Project Closeout Transition Checklist should check the following:

- Have all deliverables (products, services, documents) been accepted?
- Has the project been evaluated against each goal established in the project plan?
- Has the actual cost of the project been registered and compared to the approved cost baseline?
    - Have all changes to the cost baseline been identified and their impact on the project documented?
- Have the actual milestone completion dates been compared to the approved schedule?
    - Have all changes to the schedule baseline been identified and their impact on the project documented?
- Have all approved changes to the project scope been identified, and their impact on the performance, cost and schedule baselines documented?
- Has operations management formally accepted responsibility for operating and maintaining the project deliverables (products, services)?
    - Has the documentation related to operation and maintenance been completed?
    - Has the training and knowledge transfer been completed?

- Have the resources used by the project been released / transferred to the resource pool?
- Has the project documentation been archived?
- Have the lessons learned been collected and archived?
- Has the Post Implementation review date been set?

The Project Closeout Transition Checklist should be signed by all involved parties (project manager, client / customer representative).

10.2.2 Project Closeout Report

The Project Closeout Report is prepared by the project manager. This report documents the completion of the closeout tasks, provides a historical summary of deliverables and identifies variances from the baseline plan. It also details the lessons learned. The project manager collects information from stakeholders, customers, design, development and testing teams. Based on their input, the lessons learned reports and the Project Closeout Transition Checklist, the successes and failures of the project are identified.

The Project Closeout Report should provide the following information:

- General Information – basic information identifying the project and its stakeholders.
- Project Deliverables – a list of all product, service or documentation deliverables including the acceptance date of each deliverable and the contingencies / conditions related to each deliverable
- Performance Baseline – a list of all performance goals (if any) established in the Project Performance Plan including measurement and performance test results.
- Cost Baseline – a list of all project expenditures (e.g. tools, staff, hardware, training), including their cost compared to actual spending. If there is a difference between planned and actual cost, an explanation must be provided.
- Schedule Baseline – a list of the Work Breakdown Structure (WBS) elements, including the start and end dates of these compared to the actual status. If there is a difference between planned and actual duration, an explanation must be provided.
- Scope – a list of all changes (and their impact) to the scope of the project.
- Operation and Maintenance – a description of the operation and maintenance plan for the project's deliverables. The cost of the maintenance should be provided, if possible.
- Project Resources – a list of the resources (with estimated turnover dates) used by the project.
- Project Documentation – a list of all available project documentation. The location and license information of each document must be provided.

- Lessons learned – a list of all reports on lessons learned (see *Documenting Lessons Learned* on page 95).

- Date of Post Implementation Report – the scope of the Post Implementation Report (see *Post Implementation Report* on page 95.

- Approval – the signature of the person who authenticates the content of the Project Closeout Report.

10.2.3 Post Implementation Report

The Post Implementation Report is a document that records the successes and failures of the project deliverables. This report is prepared by the project manager and indicates whether the project achieved a return on investment. The report should include:

- Verification that the deliverables solved any problems that the project proposal identified.
- An assessment of the project deliverables' impact:
  - Scientific impact.
  - Business impact.
  - Social impact.
  - Required operational changes in the client's environment.
- Project Performance Measures.
- Actual cost vs. projected cost.
- Client's satisfaction / acceptance with the delivered product.
- Return on Investment (RoI, if possible).

10.3 Documenting Lessons Learned

Lessons learned are the documentation of the experience gained during a project while working on and delivering solutions for real problems. Gathering the lessons learned is supposed to eliminate a reoccurrence of the same problems in future projects. The lessons learned during the project lifecycle should be collected (preferably during a meeting attended by all involved parties: designers, developers and managers of the project) and published as a set of reports (one report per lesson learned). Each report should be no longer that one page and contain the following sections:

- Heading:
  - Name of the project.
  - Meaningful name of the lesson.
  - Date.
  - Point of Contact for the particular lesson.

- Body:
  - Description of the problem.
  - Description of the causes and potential impact of the problem.
  - References (e.g. printed or online materials that describe this problem in detail).
  - List of actions that were taken and impact of these (also if the actions that were taken turned out to be incorrect).
  - References the problem.
  - Description of the causes and potential impact of the problem.
  - References (e.g. printed or online materials that describe this problem in detail).
  - List of actions that were taken and impact of these (also if the actions that were taken turned out to be incorrect).

References

[ALGEFF]	http://en.wikipedia.org/wiki/Algorithmic_efficiency (accessed September 2009)
[ANTIPATTERN]	http://en.wikipedia.org/wiki/Anti-pattern
[Apache JMeter]	http://jmeter.apache.org/
[Apache JMeter Plug-ins]	http://rubenlaguna.com/wp/better-jmeter-graphs/index.html/ http://code.google.com/p/jmeter-plugins/downloads
[AutoBAHN]	https://intranet.geant.net/sites/Services/SA4/Documents/AutoBAHN%20audit/AutoBAHN%20Audit%20Report.pdf
[BTBP]	http://hedgehoglab.com/about/blog/2009/01/04/bug-tracking-best-practice/
[BTRACK]	http://www.developer.com/java/other/article.php/3389021
[BUGZ]	http://landfill.bugzilla.org/bugzilla-tip/page.cgi?id=bug-writing.html http://www.bugzilla.org/docs/tip/en/html/bugreports.html#fillingbugs
[Checkstyle]	http://checkstyle.sourceforge.net/
[CMJournal]	http://www.cmcrossroads.com/cm-journal-articles/13144-isoiecieee-12207-and-15288-the-entry-level-standards-for-process-definition-part-2
[CODEREV]	http://en.wikipedia.org/wiki/Code_review http://www.basilv.com/psd/blog/2007/strategies-for-effective-code-reviews http://smartbear.com/docs/BestPracticesForPeerCodeReview.pdf http://smartbear.com/codecollab-code-review-book.php http://wiki.openlaszlo.org/Code_Review_Checklist http://openmrs.org/wiki/Code_Review_Overview
[CodeReview]	http://www.cs.drexel.edu/~spiros/teaching/SE320/slides/code-inspections.ppt
[developerWorks]	http://www.ibm.com/developerworks/webservices/library/ws-tip-strstest/index.html
[EGEE]	http://egee-jra1-testing.web.cern.ch/egee-jra1-testing/glossary.html
[EJB]	http://java.sun.com/developer/technicalArticles/ebeans/ejb_30/
[EM]	http://java.sun.com/developer/technicalArticles/JavaEE/JavaEE6Overview_Part3.html#pesslock
[ENOS]	http://enos.itcollege.ee/~jpoial/docs/tutorial/essential/threads/lifecycle.html
[ENTB]	http://www.redhat.com/docs/en-US/JBoss_Enterprise_Application_Platform/4.3.0.cp03/html/Server_Configuration_Guide/clustering-entity-30.html
[EXTREME]	http://www.extremeprogramming.org/rules/pair.html
[FAGAN]	Fagan, M.E., Design and Code inspections to reduce errors in program development, 1976, IBM Systems Journal, Vol. 15, No 3, Page 182-211
[FindBugs]	http://findbugs.sourceforge.net/
[GÉANT Tools]	http://tools.geant.net/portal/
[GNST]	http://csrc.nist.gov/publications/nistpubs/800-42/NIST-SP800-42.pdf

[IEEE610] http://www.idi.ntnu.no/grupper/su/publ/ese/ieee-se-glossary-610.12-1990.pdf

[ITIL Service Transition] *ITIL Service Transition Book*, http://www.itil.org.uk/st.htm

[JACAANTIP] http://www.odi.ch/prog/design/newbies.php

[Java] http://download.oracle.com/javase/tutorial/essential/concurrency/

[JAVAPER] M. Keith, M. Schincariol, "Pro EJB 3: Java Persistence API (Pro)", Apress, Berkeley, CA, USA, 2006

[JCIP] Goetz B., et al, “Java Concurrency in Practice”, Addison-Wesley 2006

[KEY] “Change and Configuration Management: Is Your SCM Up to the Job?” http://accurev.com/blog/2011/07/14/change-and-configuration-management-is-your-scm-up-to-the-job/

[Knuth] D. Knuth, Structured Programming with go to Statements, ACM Journal Computing Surveys, Vol 6, No. 4, Dec. 1974. p.268

[KPI] http://wiki.en.it-processmaps.com/index.php/ITIL_KPIs_Service_Transition

[Level] http://technet.microsoft.com/en-us/library/bb497053.aspx
https://cabig.nci.nih.gov/workspaces/CTMS/Templates/Test%20Plan%20Template_Baseline.doc
http://www.comforte.com/ecomaXL/get_blob.php?name=Service_Level_Overview

[Macxim] http://qualipso.dscpi.uninsubria.it/macxim

[MAP] D. Gove, Application Programming, Addison-Wesley 2010

[Open Source Testing] http://www.opensourcetesting.org/functional.php.

[OPF] http://www.opfro.org/index.html?Components/WorkProducts/ArchitectureSet/TechnologyReadinessAssessment/TechnologyReadinessAssessment.html~Contents
http://www.brightworksupport.com/White%20Papers%20%20Guides/MSF%20Processs%20Model.pdf

[OPTIMISE] http://en.wikipedia.org/wiki/Program_optimization (accessed September 2009)

[OSCACHE] http://www.opensymphony.com/oscache/

[OWASP Code Review] https://www.owasp.org/index.php/Category:OWASP_Code_Review_Project

[OWASP Testing Guide] https://www.owasp.org/images/5/56/OWASP_Testing_Guide_v3.pdf

[OWASP Testing Project] https://www.owasp.org/index.php/Category:OWASP_Testing_Project

[PARETO] http://en.wikipedia.org/wiki/Pareto_principle

[PATTERN] Fowler M., "Analysis Patterns: Reusable Object Models", Addison-Wesley 1997
http://en.wikipedia.org/wiki/Design_pattern_%28computer_science%29

[PMD] http://pmd.sourceforge.net/

[POM] https://intranet.geant.net/sites/Services/SA4/T1/Documents/ishare_pom.xml

[REGTEST] http://msdn.microsoft.com/en-us/library/aa292167%28VS.71%29.aspx

[SA2conf] https://intranet.geant.net/sites/Services/SA4/T1/Documents/ishare_checkstyle_checks.xml

[SCan] http://security.psnc.pl/files/szkolenia/gn3_training/day2/03_Source_code_analyzers.pdf

[SCMBP] http://www.perforce.com/perforce/papers/bestpractices.html

[SDG] The most recent version of the GN3 Software Developer Guide can be accessed from the GÉANT Intranet at:
https://intranet.geant.net/sites/Services/SA4/T1/Documents/Forms/AllItems.aspx

[SMELL] http://en.wikipedia.org/wiki/Code_smell

[SOA Testing] http://soa-testing.blogspot.com/

[SST] http://www.cigital.com/papers/download/bsi4-testing.pdf

[Syscon SOA] http://soa.sys-con.com/node/284564

[TEST] http://en.wikipedia.org/wiki/Software_testing

[TexasUni] http://media.govtech.net/GOVTECH_WEBSITE/EVENTS/PRESENTATION_DOCS/2007/GTC_EAST/P15%20Part%204-acceptance-test%20execution.pdf

[TG] http://www.testinggeek.com/index.php/testing-types/testing-purpose/106-accessibility-testing

[Valgrind] http://valgrind.org/

[VITA] http://www.vita.virginia.gov/

[VITA1] http://www.vita.virginia.gov/uploadedFiles/Library/CPMG-SEC5.pdf

[WP] http://en.wikipedia.org/wiki/System_testing

Glossary

API	Application Programming Interface
ASM	Automatic Storage Management
AST	Abstract Syntax Tree
C&A	Certification and Accreditation
CAB	Change Advisory Board
CBO	Coupling Between Objects (page 50)
CBO	Cost-Based Optimiser (page 68, 69)
CI	Configuration Item
CM	Configuration Management
CMMI	Capability Maturity Model Integration
CPU	Central Processing Unit
DDT	Data-Driven Testing
DIT	Depth of Inheritance Tree
DNS	Domain Name System
ECA	Elementary Code Assessment
EJB	Enterprise JavaBeans
GUI	Graphical User Interface
Httest	HTTP Test Tool
HTTP	Hypertext Transfer Protocol
IEEE	Institute of Electrical and Electronics Engineers
ITIL	Information Technology Infrastructure Library
IO	Input Output
IP	Internet Protocol
JDBC	Java Database Connectivity
JPA	Java Persistence API
JSP	JavaServer Pages
JVM	Java Virtual Machine
KB	Kilobyte
KPI	Key Performance Indicator
LCOM	Lack of Cohesion in Methods
MPI	Message-Passing Interface
NOC	Number of Children
OLAP	On Line Analytical Processing
OLTP	Online Transaction Processing
ORM	Object-Relational Mapping
OWASP	Open Web Application Security Project

QA	Quality Assurance
QATB	QA TestBed workflow
QoS	Quality of Service
RAID	Redundant Array of Independent/Inexpensive Disks
RDBMS	Relational Database Management System
RFC	Response for a Class
RoI	Return on Investment
SA	Service Activity
SCM	Software Configuration Management
SLA	Service Level Agreement
SOA	Service-Oriented Architecture
SSD	Solid State Drive
SQL	Structured Query Language
TCP	Transmission Control Protocol
TDD	Test-Driven Development
UML	Unified Modelling Language
VM	Virtual Machine
WAL	Write-ahead log