ICT PROJECT 258109
Monitoring Control for Remote Software Maintenance

<table>
<thead>
<tr>
<th>Project number:</th>
<th>258109</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Acronym:</td>
<td>FastFix</td>
</tr>
<tr>
<td>Project Title:</td>
<td>Monitoring Control for Remote Software Maintenance</td>
</tr>
<tr>
<td>Call identifier:</td>
<td>FP7-ICT-2009-5</td>
</tr>
<tr>
<td>Instrument:</td>
<td>STREP</td>
</tr>
<tr>
<td>Thematic Priority:</td>
<td>Internet of Services, Software, Virtualization</td>
</tr>
<tr>
<td>Start date of the project:</td>
<td>June 1, 2010</td>
</tr>
<tr>
<td>Duration:</td>
<td>30 months</td>
</tr>
</tbody>
</table>

D2.2: Integration Plan and Technical Project Guidelines

Lead contractor: TUM
Dennis Pagano, Nitesh Narayan, Amel Mahmuzić, Alex Waldmann, Sergio Zamarripa López, Walid Maalej, Daniel de los Reyes, João Coelho García

Author(s):            
Submission date:      | March 2011       |
Dissemination level:  | Restricted       |

Project co-founded by the European Commission under the Seventh Framework Programme
© S2, TUM, LERO, INESC ID, TXT, PRODEVELOP
Abstract: This document summarizes the project standards of FastFix. It defines the development and integration processes as well as quality assurance activities. We describe a continuous integration approach that reduces integration risks and removes dependencies between releases and integration activities. This report also describes the project infrastructure and tools to be used within FastFix. We plan automatic nightly builds, which minimize integration risks and support the continuous development approach. Additionally we describe concise standards and guidelines to facilitate tool usage and standardize developed artifacts in FastFix.
Contents

1 Introduction

2 Processes

2.1 Development and Integration Process

2.1.1 Continuous Integration

2.1.2 Development Roles

2.1.3 Architecture Design and Interface Specification

2.1.4 Integration Testing

2.1.5 Integration and Release Plan

2.2 Quality Assurance Process

2.2.1 Static Code Analysis

2.2.2 Code Review

2.2.3 Unit Testing

2.2.4 Regression Testing

2.2.5 Bug and Issue Tracking

2.2.6 End-User Feedback Integration

3 Infrastructure

3.1 Development Platform

3.1.1 Java

3.1.2 Eclipse

3.1.3 Software Configuration Management

3.2 Integration Platform

3.2.1 Automated Build System

3.2.2 Continuous Integration System

3.2.3 FastFix Automated Build Workflow

3.3 Quality Assurance Tools

3.3.1 Static Code Analysis Tool

3.3.2 Coding Standards Tool

3.3.3 Issue Tracking Tool

3.3.4 Quality Management Tool
4 Guidelines

4.1 Development Guidelines ............................................. 33
  4.1.1 Design Patterns ................................................. 33
  4.1.2 Code Comments .................................................. 34
  4.1.3 Style Checks .................................................... 34
  4.1.4 Javadoc Warnings ................................................. 34
  4.1.5 Code Templates .................................................. 35

4.2 Integration Guidelines ................................................. 35
  4.2.1 Development .................................................... 35
  4.2.2 Interface Specification .......................................... 36
  4.2.3 Service Oriented Architecture .................................. 38
  4.2.4 Dependency Management ........................................ 39

4.3 Security Guidelines .................................................. 39
List of Figures

2.1 Developer and Architect Roles in FastFix. .......................... 11
2.2 Architecture Design and Interface Specification Activities in FastFix. 12
2.3 Architecture and Interface Refinement Activities in FastFix. .......................... 12
2.4 External and internal releases (example process). .......................... 13
2.5 Integration and Release Activities in FastFix. .......................... 14
2.6 Refinement of Continuous Integration Means. .......................... 15
2.7 Bug and issue tracking in FastFix (adopted from [5]). .......................... 21

3.2 Maven output showing a successful build of the Server Project.  ................... 25
3.1 FastFix project overview on Huson. .......................... 26
3.3 FindBugs Bug Explorer. .......................... 28
3.4 Checkstyle Violations View. .......................... 28
3.5 Sonar Home. .......................... 29
3.6 Sonar Dashboard. .......................... 30
3.7 Sonar Time Machine. .......................... 31
3.8 Sonar Violations Drilldown. .......................... 31
3.9 Sonar Violation Detail. .......................... 32

4.1 IPatchService provider and consumer. .......................... 37
4.2 OSGi component structure of ApplicationBridge and PatchGenerator. .......................... 37
4.3 Implementation of OSGi service provision in PatchGenerator. .......................... 37
4.4 Implementation of OSGi service consumption in ApplicationBridge. .......................... 38
4.5 OWASP Application Security Verification Standard [13]. .......................... 40
1 Introduction

FastFix is developed by a distributed team of developers. To facilitate distributed development we base work on several processes that have proven to be beneficial. FastFix developers follow these processes sharing a common tool infrastructure. Concise standards and guidelines should finally facilitate tool usage and standardize developed artifacts.

This document introduces standards that define how FastFix will be developed. Apart from activities that ensure a high quality, we describe in particular the development and integration process used in FastFix. Integration is a crucial part in software development that needs to be planned and synchronized with the release plan. FastFix will use a continuous integration approach that reduces integration risks and removes dependencies between releases and integration activities. Integration is performed continuously resulting in daily mini releases.

The FastFix release plan is described in [8]. This document does not further detail or narrow down the release plan, but complements it by specifying how and when the integration of the different subsystems should take place. Further, the scope of FastFix, i.e. the functionality the FastFix system should provide, is not part of this document but defined in [14]. The security requirements described in [1] provide the starting point for security guidelines and activities described in this document.

Section 2 introduces the development and integration as well as quality assurance processes for FastFix. Activities in these processes require a common tool infrastructure, which we will introduce in Section 3. Last, in Section 4 we describe concise standards and guidelines to facilitate tool usage and standardize developed artifacts.
2 Processes

This section introduces two major processes, the development and integration process as well as the quality assurance process. Both processes describe activities suited for and supporting distributed development. While quality assurance activities focus on guaranteeing high quality development results, development and integration activities provide means for a continuous development approach that reduces risks and facilitates build and release management.

In the remainder of this section we introduce both processes by summarizing the according activities. The two subsequent sections describe how these processes are realized in terms of infrastructure and guidelines.

2.1 Development and Integration Process

FastFix is an aggregation of different specialized subsystems (components) that need to cooperate via specified interfaces to provide the desired functionality of the FastFix system as a whole. System integration is the process of combining (integrating) all subsystems into one system. It ensures that all subsystems work properly together and the system as a whole provides the desired functionalities.

The integration activity itself depends strongly on the system architecture and in particular on the public interfaces of the subsystems. Integration activities include the specification of adequate interfaces and influence architecture design decisions. Further, integration includes the identification and development of missing pieces that eventually enable the cooperation of the different subsystems. As a result, system integration adds value to the system by enabling capabilities that become possible only because of inter-subsystem interactions.

FastFix uses continuous integration and automated builds to reduce the risk of integration failures at a late stage of the project. To ensure that the subsystems work together as desired, integration tests are developed and performed automatically in the course of the continuous integration activities.

In the following section we briefly introduce continuous integration and summarize the continuous integration approach of FastFix. In Section 2.1.2 we describe different developer roles in FastFix with respect to integration. The following two sections summarize
architecture design and interface specification of the different components (Section 2.1.3) and introduce integration tests (Section 2.1.4). Finally, in Section 2.1.5 we describe the integration and release plan for the FastFix project.

2.1.1 Continuous Integration

FastFix will use a continuous integration [4] approach to combine subsystems and ensure interoperability. Continuous integration implies the continuous application of the integration activities, in particular bringing together all subsystems and testing their cooperation. The approach aims at continuous quality control of the developed system. It improves software quality and reduces the time to deliver by altering the traditional practice of applying quality control after completing all development. A continuous integration process is realized using special project infrastructure as well as special development and collaboration practices. In the remainder of this section we introduce these infrastructural and procedural aspects and summarize practices that have proven to be beneficial.

2.1.1.1 Infrastructural Enablers

Software Configuration Management systems (cf. Section 3.1.3) enable developers to work distributedly together in a structured way, while providing a central place to host the entire system under development. Automated build systems (cf. Section 3.2.1) facilitate build tasks and ensure that the developed software integrates properly by taking care of dependencies between subsystems as well as dependencies to third-party components. Continuous integration systems (cf. Section 3.2.2) eventually automate several integration tasks by connecting to the source code repository to acquire the latest version, using automated build systems to build the system, and performing integration tests after the builds to ensure the integration quality. These systems further allow the scheduling of integration tasks as well as the delivery of the built results, i.e. the compiled system so that a running version can be obtained easily.

2.1.1.2 Development Process Enablers

In a project using continuous integration, developers should carry out development activities in a way that supports the continuous integration approach. In order to perform changes to the current revision of the system, a developer takes a copy of the current code base on which to work. As other developers submit changed code to the code repository, this working copy gradually ceases to reflect the repository code. Before developers submit their changes to the repository they must therefore first update their copy to reflect the changes in the repository since they took their copy. If more than one developer changed the same source code section, this leads to a conflict that has to be resolved. Conflict
resolution often has to be done manually. The more changes the repository contains, the more probable is a conflict, and the more work developers must perform before being able to safely submit their own changes. Therefore it is a effective practice to commit changes frequently. We describe integration guidelines in Section 4.2.

2.1.1.3 Key Success Factors

The following practices are considered to be key success factors to continuous integration in software projects [6]. For each practice we summarize how it is supported and ensured in FastFix.

- **Single source repository.** The FastFix project is hosted on a single SVN repository\(^1\). More information can be found in Section 3.1.3.

- **Build automation.** The FastFix infrastructure includes an automated build system which we summarize in Section 3.2.1.

- **Test automation.** After building the current revision of the system, the continuous integration system used in FastFix (cf. Section 3.2.2) automatically performs unit and integration tests.

- **Regular commits to baseline.** Regular commits can significantly reduce the number of conflicting changes. FastFix developers obey this practice as stated in the integration guidelines in Section 4.2.1. In addition, nightly builds will be performed on the FastFix continuous integration system (cf. Section 3.2.2).

- **Build of commits to baseline.** The system should automatically build the current working version after a commit to baseline in order to verify that they integrate correctly. Continuous integration is sometimes considered as a completely automated process. The automation can be realized by a integration server that listens for changes in the version control system and automatically performs builds afterwards. The advantage of this practice is that integration errors are spotted early during development. A disadvantage though is that developers tend to hesitate to commit their source code, since they want to be sure that build errors do not occur. FastFix uses a continuous integration system that provides facilities to automatically build each commit to baseline (see Section 3.2.2). Developers’ commit frequencies will be monitored regularly in order to spot hesitation. In the latter case automated builds could be reduced to nightly builds.

- **Rapid builds.** Builds should be completed rapidly. This ensures that integration problems are identified quickly and reduces developers’ hesitation to commit their

\(^1\)https://repository.fastfixproject.eu/svn/fastfix/Software
changes. The FastFix infrastructure uses an efficient dependency management and build system (cf. Section 3.2.1). Builds are completely automated what reduces build time to a minimum.

- **Tests in production environment.** Test environments may hide failures in real systems deployed in the production environment, since the production environment may differ from the test environment in a significant way. However, building a production environment replica is cost and time intensive, and the production environment in FastFix does not provide all aspects needed for easily testing the functionality of FastFix. Instead, the test environment should be built to be a scalable version of the actual production environment to both alleviate costs while maintaining technology stack composition and nuances. In the FastFix project, the test environment comprises early versions of the trial applications to be maintained. This ensures that the tested technology is representative and the environment may be established with low costs.

- **Easy access to newest release.** Providing builds to stakeholders and testers reduces work when re-developing a feature that does not meet requirements. Early testing additionally reduces chances that errors remain undiscovered until deployment. The FastFix continuous integration system (see Section 3.2.2) will automatically provide latest builds to testers making them available on a dedicated website.

- **Visible build results.** Build results should be available to the team. If a build failed it should be easily spottable who made the breaking change and where the change occurs. FastFix uses a continuous integration system (see Section 3.2.2) that brings all this information together. Additionally build results can e.g. be pushed to a mailing list in order to notify relevant team members.

- **Automated deployment.** Automatically deploying the application to a live test server goes one step beyond providing executables to testers. It further reduces the overhead needed before being able to test the system. In the FastFix project, the built system will be automatically deployed into an OSGi testing environment.

### 2.1.2 Development Roles

As shown in Figure 2.1, we distinguish between two different types of FastFix developers. First, the actual developers who check out source code from the repository, change and integrate code, and commit their changes to the repository. These developers perform development tasks that do not influence the overall architecture, do not change the public interface specification, and do not influence the dependency structure of FastFix components. The second role, the architect, takes care about architectural decisions. In addition
to the described development tasks, the architect may change FastFix architecture, as well as the public interfaces of the components.

The reasons for this distinction are threefold. First, architectural decisions in a composed system like FastFix influence not only the component where a change is made, but also components that depend on the changed interfaces. Second, since the overall functionality can only be guaranteed if its components provide all services that are needed, an architectural change has always to be considered in a system-wide context. The developer, however, does not need to have the big picture at hand. The third reason regards the FastFix infrastructure and will be discussed in Section 3.2.1. In short, architects need a more sophisticated infrastructure than developers.

### 2.1.3 Architecture Design and Interface Specification

FastFix is a system that consists of different components that have to work together to provide the overall FastFix functionality. These components are developed by several distributed teams of developers. The conceptual FastFix architecture and an interface specification (or service description) of the components guide this distributed development process. Both, architecture and interface specification are designed and refined by the FastFix architecture taskforce, which embodies the architect role described in Section 2.1.2.

As shown in Figure 2.2 the architecture taskforce first designs the initial conceptual architecture of FastFix in the form of components and their dependencies based on the FastFix requirements. This architecture provides the starting point for the initial specification of component interfaces. The architecture taskforce then defines these component interfaces. The resulting conceptual architecture as well as the interface specification are
further refined continuously throughout the development.

As described in Section 2.1.2, only FastFix architects will be able to change the architecture and interface specification during development. Figure 2.3 summarizes how developers and architects work together to refine architecture and interface specification.

The focus of interface specification is to communicate clearly and precisely about public details of a component. Interfaces describe what is publicly visible and how a component is going to be used. In object-oriented software development, the concept of encapsulation demands the decoupling of the interface specification from its implementation in order to make requesting applications independent from internal modifications [11].

### 2.1.4 Integration Testing

Components will be integrated into larger subsystems, and finally into the system as a whole. It is still likely, however, that these components contain faults, since test stubs and drivers that are typically used in unit tests provide only approximately the behavior of the simulated components. Further, invalid assumptions when calling components’
interfaces lead to faults that may not be detected by unit tests. Integration testing detects faults that have not been detected during unit testing by focusing on small groups of components. Usually several components are integrated and tested, and when no new faults are revealed, additional components are added to the test group. The order in which components are integrated and tested can influence the total effort of integration tests. Resources needed for the integration testing activity may be reduced by sensibly integrating and ordering the tested components.

In practice there are two major integration test approaches [3]. Horizontal integration testing strategies that integrate components according to layers, and vertical integration testing strategies that integrate components according to functions. FastFix will use a vertical integration testing strategy for two major reasons. First, vertical integration testing strategies focus on early integration [3]. Second, the FastFix architecture is planned according to functional units (functional subsystem decomposition). Developers mainly work within the functionally decomposed subsystems, what requires integration tests against other functional units.

2.1.5 Integration and Release Plan

The preliminary FastFix release plan is described in [8]. It includes six official, external releases: five intermediate releases and one final release. External releases represent important development milestones and provide means to manage the overall progress of the project. Each release milestone describes what functionality should be implemented to what degree at a certain point in time.

Additionally to these official external releases there will be several internal releases. Internal releases are planned flexibly and used mainly to structure the development efforts and to provide fine grained development guidelines to component service providers. The internal releases provide means to manage the integration of system components by defining what sub-functionality has to be finished at a certain point in time. Currently the FastFix release roadmap contains five internal releases between the first two official releases, which amount to one internal release per month. Internal releases between sub-
sequent external releases will of course be planned and refined in the course of the project. Figure 2.4 illustrates external and internal releases in an example process.

As shown in Figure 2.5, the FastFix release manager role continuously elaborates and refines the release plan. After each release the plan for the next two releases is refined in detail using open issues and action items collected in the issue tracker. Developers use the refined release plan to develop FastFix features according to the priority and due date in the release plan. Integration takes place continuously, resulting in nightly builds and daily “mini” releases which reduce integration risks. When a milestone is reached, the release management team announces the according external release. Additionally, continuous integration means are refined throughout the whole development cycle by the architecture taskforce. In FastFix release and integration management activities are seamlessly combined.

As described before, FastFix uses a continuous integration approach (cf. Section 2.1.1) to combine the different subsystems and ensure their interoperability. The continuous integration task in FastFix will start at the end of May 2011 (month 12 of the project) with the first intermediate release called “Munich”. Integration itself will then take place continuously until the end of the project and the final release using the means described in the remaining sections of this document.
Initial Integration Milestone “Munich” - M12

The “Munich” milestone is defined to coincide with the first release of FastFix at the end of May 2011 and will provide.

- Public interfaces of complete architecture defined as UML & javadoc as well as their implementation in the form of stubs
- Initial iteration of integration tests
- Component dependencies (OSGi)

Continuous Integration Tasks

During the development of the project the following integration tasks will be carried out continuously by the FastFix architecture taskforce.

- Refine public interfaces
- Refine and iterate over integration tests
- Test and resolve component dependencies

Figure 2.6 summarizes how the architect refines continuous integration means throughout the whole development cycle. Actually combining components and performing integration tests is not shown as part of these tasks since these activities are carried out automatically to a great extent.
D2.2: Integration Plan and Technical Project Guidelines

2.2 Quality Assurance Process

Software developers perform several activities to ensure high software quality in terms of faults and compliance with specified behavior. These activities happen at different stages of the development process. Some of them are continuous and others are recurring.

This section summarizes the quality assurance activities performed in the development process of the FastFix system.

2.2.1 Static Code Analysis

Static code analysis is a methodology of error detection in program code, analyzing the places where potential errors may occur. In other words, the static analysis tool detects in the program code the places containing or likely to contain errors. Such code sections are to be considered by the programmer and he can decide whether to modify this program section or not. Static analysis can be both of general purpose and specialized for searching certain error classes.

The main advantage of static code analyzers use lies in the possibility of considerable cost saving of defects elimination in a program. The earlier an error is determined, the lower is the cost of its correction. Thus, according to McConnell [10], correction of an error at the testing stage is ten times more expensive than its correction at the construction (coding) stage.

Static code analysis can be used during several stages of the development process:

- **Code edition**: First, static code analysis must be provided during code editing, with the use of the current IDE (Eclipse in FastFix) in combination with a bug finder like the FindBugs plugin for Eclipse. Hence, the quality assurance process will include the installation and effective utilization of this plugin in all of the programmer computers.

- **Code sharing time**: In other words, at commit time, we can perform a pre-commit check in the source revision control. This will only be a syntax check, consisting of ensuring that the syntax of xml, scripts and properties files is valid. At this stage, the quality assurance process cannot provide a more complex analysis, since it takes time and it will slow down the team.

- **Automatic checks**: On a regular basis as one of the continuous integration tasks checks will be executed. To this end, a static code analysis tool (see Section 3.3.4) will be used by the continuous integration system (see Section 3.2.2) to perform regular checks within the course of the nightly system builds. It will also be possible to launch these checks on demand.
2.2.2 Code Review

Not all errors can be found using static analysis. Nowadays there are still many defects that may only be identified during code reviews. Code review is a process where one member of a team reviews code written by another member. It gets team members involved in different parts of the system than the one they had been working on. This helps to reduce redundancy and keeps the team informed about what everyone is doing. Some of the many immediate benefits of code reviewing in practice are:

- Consistent coding standards. By reviewing other developers code coding standards can be enforced.
- Mentoring. Code review can be used as a tool to help trainees and newcomers to boost their development skills and reduce rampup duration.
- Quality code. The code review process helps to identify defects early in the development process.
- Wide project knowledge. Developers performing a code review can review code in the system that is yet unknown to them. This practice helps to get a wider sight of the project.
- Structured code. Because developers know that someone will review their code, they will document it properly and format it well.

In the FastFix development process, code will be reviewed and refactored regularly by developers that did not take part in its creation. Further, pair programming will be performed if possible in particular on source code that is critical with respect to security (cf. Section 4.3).

2.2.3 Unit Testing

Unit testing allows to determine if individual units of source code are fit for use. FastFix will provide a way to create tests for the smallest testable parts of the application, written by the programmers of each one of FastFix components. These tests will then be performed automatically by the continuous integration system (see Section 3.2.2).

Among the benefits unit testing can provide, it is worth mentioning that it simplifies integration and it facilitates change, which is also connected with the spirit of regression testing, since it makes sure that the code in a module still works correctly.

For Unit Tests purposes, FastFix will use JUnit, as well as for the mentioned regression testing.
Unit test suites must be fully automated to be useful. Automation of unit tests can be provided with the integration of Apache Maven build tool (described in section 3.2.1) such that the unit tests can be automatically run for each important build. In addition to the execution of unit tests for each build, they can also be executed on demand with the command “mvn test”, and it also provides a way to disable the unit tests from running with the “maven.test.skip” property. The entire unit test suite should be executed before every code check in. Hence, each developer should be responsible for ensuring that all new and existing tests runs successfully upon code check in.

The following guidelines will be ensured:

- Name tests properly, using the test<What> naming convention (e.g. testAddListener()).
- Keep unit test small and fast.
- Unit tests should be fully automated and non-interactive. If the test require interaction, they cannot be automated, and if the results require manual inspection, they are not proper unit tests.
- Fix failing tests immediately.
- Keep tests independent, ensuring testing robustness and simplifying its maintenance.
- Cover boundary cases: Make sure the parameters boundary cases are covered.
- Test each feature once and only once, keeping the amount of test code as low as possible.
- Prepare test code for failures. The failure of a single test must not bring down the entire test suite execution.
- Write tests to reproduce bugs

2.2.4 Regression Testing

Any time a modification happens in an implementation within a program, regression testing should be done. This is done by rerunning existing tests against the modified code to determine whether the changes break anything that worked prior to the change and by writing new tests where necessary. In other words, regressions occur whenever software functionality that was previously working correctly, stops working as intended, since it is quite natural to have a big amount of changes in the source code during software
development. Adequate coverage is a primary concern when conducting regression tests. Strategies and best practices core to the whole process must include:

1. Test fixed bugs in every aspect. This affects e.g. the integration of functionalities.

2. Side effects of fixes. The bug itself might be fixed but the fix might invoke new problems.

3. Write a regression test for each bug fixed.

4. If two or more tests are similar, determine which is less effective and get rid of it.

5. Identify and archive core sets of test cases.

6. More focus on functional issues, then design.

7. Trace the effects of the changes on performance.

8. Perform automated regression testing on a regular basis, especially the ones related to functional testing.

9. Regression testing should be performed like a chain, starting from the unit level, involving adaptation & rerunning of the unit tests after sensing any change to the unit. The chain of regression tests continues further across integration testing, system testing, user acceptance testing and across all operational phases under the SDLC.

JUnit\(^2\) is a unit and regression testing framework for Java, one of a family of unit and regression testing frameworks collectively known as xUnit. It will be integrated with the FastFix continuous integration system to automate unit and regression tests.

2.2.5 Bug and Issue Tracking

Bug and issue trackers manage and maintain lists of bugs respective issues. Software engineers use issue trackers to create, update, and resolve project issues. Common issue trackers further provide the possibility to maintain a knowledge base including information that may help to resolve issues. In FastFix bug and issue tracking will support amongst others the following activities. First, project participants may document and maintain bugs. Second, they may use issues to file action items or tasks. Issue trackers have proven to be effective lightweight task management systems.

We will use Trac (cf. Section 3.3.3) as bug and issue tracker in FastFix. Bugs and issues are filed as so-called tickets, containing the following information attributes.

\(^2\)www.junit.org
• Reporter — The author of the ticket. The person who is reporting the ticket. It is important that the ticket defines what is exactly the problem or the request.

• Type — The nature of the ticket (for example, defect or enhancement request)

• Component — The project module or subsystem this ticket concerns.

• Version — Version of the project that this ticket pertains to.

• Keywords — Useful for searching and report generation.

• Priority — The importance of this issue, ranging from trivial to blocker.

• Milestone — When this issue should be resolved at the latest.

• Assigned to/Owner — Principal person responsible for handling the issue.

• Cc — A comma-separated list of other users or E-Mail addresses to notify. Note that this does not imply responsibility or any other policy.

• Resolution — Reason for why a ticket was closed. One of fixed, invalid, wontfix, duplicate, worksforme.

• Status — What is the current status? One of new, assigned, closed, reopened.

• Summary — A brief description summarizing the problem or issue.

• Description — The body of the ticket. A good description should be specific, descriptive and to the point.

In addition to the main attributes of a ticket, we define quality criteria for bug reports. To be helpful, a bug report needs the following aspects. First, the steps to reproduce the issue, second, a description of what the reported expected to see, and third, what the reporter saw instead. In addition to that, we will follow other guidelines that can be useful to benefit from the issue tracker to a larger extent:

• The person who opened the bug should close it, since only the person who saw it can really be sure that it is fixed.

• The steps to be followed to reproduce the bug should be kept to the minimal.

• Always keep in mind that the possible ways to resolve a bug are: fixed, won’t fix, duplicate or worksforme.

• The version is important, every build of the software must have a build ID, and it should be referenced in the version attribute, so the developer can identify the version of the source code that introduced/had the bug.
• Always use the issue tracker, don’t accept bug reports by any other method. If the issue is reported by email, it should be put in the bug database.

Figure 2.7 shows the ticket workflow in FastFix.

Figure 2.7: Bug and issue tracking in FastFix (adopted from [5]).

2.2.6 End-User Feedback Integration

The software maintainers point of view, as FastFix end-users, must be taken into account. Starting from the requirements acquisition, several interviews have been done in the project, in order to understand how software maintenance engineers are working, as well as to identify the main problems of current software maintenance practices.

Collecting end-user feedback is also a must for the FastFix evaluation metrics mentioned in [8], which describes that in the end of the project, the results will be evaluated with new interviews with software maintainers, in order to measure the usability and the benefits of the solution. Hence, these interviews will be done before the final release of the project, resulting in opened tickets in the Trac system, created as enhancement requests.
3 Infrastructure

This section describes the tools and platforms used to develop the FastFix system. Infrastructure decisions are based on consortium agreements and are triggered by common development standards as well as in particular the quality assurance and integration processes described in Section 2.

3.1 Development Platform

This section defines a common development platform for FastFix. FastFix will be developed in Java and using Eclipse. Developers will use a software configuration management system to perform work collaboratively.

3.1.1 Java

The main components of FastFix are developed in Java. Java is an object-oriented programming language, designed to provide a simple object model for developers and to abstract from the hardware and system specific problems. Java programs run in a safe sand-boxed environment called the Java Virtual Machine (JVM). JVM keeps track of used objects, memory and code execution. Java is known for being operating system independent. The JVM runs on the top of many operating systems and provides an abstraction from the lower level system calls. Since a version mismatch of Java can introduce problems of missing or unsupported libraries, it is a good programing practice in a collaborative development paradigm to have a consistent version of java. FastFix will base on version 1.6.

3.1.2 Eclipse

Eclipse is a free and open source software development environment that includes an integrated development environment and a plug-in system. Eclipse comprises the Rich Client Platform (RCP) for developing general purpose applications. It is based on the Eclipse core platform and the OSGi framework “Equinox”. OSGi allows developers to manage several applications as bundles inside the running Eclipse platform dynamically.
Eclipse further comprises the Rich AJAX Platform (RAP) that allows developing AJAX-enabled rich internet applications. RAP is very similar to RCP. FastFix development will be based on Eclipse Helios (version 3.6) which is the current version. However, since new Eclipse versions are launched every year, FastFix will perform compatibility checks on new versions and use them if required by FastFix developers.

### 3.1.3 Software Configuration Management

Software configuration management systems provide means for distributed teams to work collaboratively together on shared documents. Within FastFix, development will be carried out using Apache Subversion (SVN). SVN is a tool for source code versioning. It comprises a centralized repository for collaboration over source code artifacts. Further it allows to manage different development lines in so called branches. SVN provides online versioning of directories, files and metadata as well as bindings to a wide variety of textual documents including source code files. Other functionalities offered by SVN include conflict resolution and merging. FastFix developers will use the Subversive SVN client, which is available as an Eclipse plugin\(^1\).

### 3.2 Integration Platform

This section covers two main infrastructure aspects that are crucial to integration. First, automated build systems that leverage system builds by taking care of dependencies and automating builds. Second, continuous integration systems that support the continuous integration approach introduced in Section 2.1.1.

#### 3.2.1 Automated Build System

FastFix uses the Apache Maven automated build system. Apache Maven is a project management and comprehension tool. Based on the concept of a project object model (POM), it can manage a project’s build, reporting and documentation from a central place. Maven’s primary goal is to allow the developers to comprehend the complete state of a development effort in the shortest period of time. In order to attain this goal Maven considers various concerns including:

- Making the build process easy
- Providing a uniform build system
- Providing quality project information

\(^1\)http://www.eclipse.org/subversive/
• Providing guidelines for best practices development
• Allowing transparent migration to new features

Automatic dependency management with Maven influences the local development environment requirements for FastFix developers and architects as follows.

Developer

Developers are only allowed to change code without adding dependencies to other FastFix components or third-party components. Since they do not change dependencies and component interfaces, no additional tools are required on their local development environments to build the system. Developers thus can build the system using the standard Eclipse build mechanism.

Architect

Architects on the other hand need to change component interfaces and may change dependencies among bundles. To build and commit their changes, architects need to have the following systems installed in their local development environment.

1. Maven,
2. M2ECLIPSE\(^2\) - Plugin for Eclipse that allows to build using Maven directly from Eclipse.

Only architects who meet these requirements may actually commit their code changes. They also have to make sure that all bundles are built successfully locally with Maven in order to prevent errors on the build server.

A basic tutorial on creating OSGi projects with Maven can be found at: http://wso2.org/library/tutorials/develop-osgi-bundles-using-maven-bundle-plugin.

3.2.2 Continuous Integration System

For continuous integration tasks, FastFix uses the Hudson build Server\(^3\). Hudson monitors executions of repeated jobs, such as building a software project or jobs run by cron. Among those things, Hudson focuses on the following two jobs:

1. *Building and testing software projects continuously*. In a nutshell, Hudson provides a flexible continuous integration system, making it easy for developers to integrate changes to the project, and making it easy for users to obtain a fresh build. The automated, continuous build increases the productivity.

---

\(^2\)http://m2eclipse.sonatype.org/index.html

\(^3\)http://hudson-ci.org
2. Monitoring executions of externally-run jobs. This includes jobs such as cron jobs and procmail jobs, and even jobs that are run on a remote machine. With cron, developers only receive e-mails that capture job outputs. It is up to the developers to look at them diligently and notice when something broke. Hudson on the other hand keeps those outputs in one place and makes it easy for developer to notice when something is wrong. In FastFix, the architecture taskforce will receive mails when automated builds fail.

3.2.3 FastFix Automated Build Workflow

This section describes how to build the FastFix project with Maven and Hudson. As a prerequisite step, the source code should be checked out from the FastFix project repository\(^4\) using Eclipse or any other SVN tool by selecting all directories from the trunk folder. After committing a change, FastFix code can be built with the FastFix Hudson build Server\(^5\). FastFix comprises several bundles that can be built separately from each other. If changes were made in the FastFix server bundle, this bundle should be built. A project can be built by clicking "Build Now" in the main menu (cf. Figure 3.1). Alternatively, builds are performed nightly or on every commit (see Section 2.1.1).

The project build status can be seen in the Hudson user interface. The last build may always be accessed from the build history menu. To see more details about success or failure of the build, one can press on the “Console” link, where the according Maven log is shown. Figure 3.2 depicts the Maven output of a successful build.

![Maven output showing a successful build of the Server Project.](image)

After a build, the resulting packages have to be downloaded into a testing environment. All downloaded bundles must be saved inside the same folder. Because the bundles are OSGi bundles, the OSGi Execution Environment is necessary to run them. OSGi Execution Environment can be found at the Eclipse project page. To run the FastFix

\(^4\)https://repository.fastfixproject.eu/svn/fastfix/Software/

\(^5\)http://hudson.fastfixproject.eu/hudson/
Server Bundle, OSGi Environment must be started by running the following line from the console:

```
java -jar org.eclipse.osgi_3.6.0.v20100517.jar -console
```

After OSGi is started, all downloaded plugins must be installed inside of the OSGi environment. By typing the following line

```
install file:{DEVELOPER_PROJECT_FOLDER_PATH}\eu.fastfix.server.eventcorrelator_1.0-SNAPSHOT.jar
```

where DEVELOPER_PROJECT_FOLDER_PATH is the absolute path to downloaded plugins, a plugin is installed. For other two plugins, same step should be done. After all plugins are installed, type “ss” and press enter. All FastFix plugins must have status "INSTALLED".

To run a bundle, only the according bundle must be started by typing:

```
"start XX"
```
Here XX refers to the ID of the bundle to run. This bundle may then call other bundles automatically because of dependencies. The output should be like this:

Starting FastFix - Server Fault Replicator Correlator...
--- Calling Event Correlator ----
Server Event Correlator starting....
Event Correlator started!
This is new line, V.4
Starting FastFix - Server Event Correlator...
FastFix Server Event Correlator started!
--- Calling of Event Correlator ended----
FastFix Server Fault Replicator started!
Test new build

3.3 Quality Assurance Tools

Quality assurance tools provide means to test and control code and thus system quality. This section covers tools that are triggered by the quality assurance activities in Section 2.2.

3.3.1 Static Code Analysis Tool

FastFix will use FindBugs to analyze source code for defects. FindBugs is a program to find bugs in Java programs [9]. It looks for instances of “bug patterns”, code instances that are likely to be errors. Bug patterns are code idioms that are often errors. Findbugs incorporate automatic detectors for a variety of bug patterns found in Java programs. These bug pattern detectors helps to find serious bugs in several widely used Java applications and libraries. It is well known that even well tested code written by experts contains a surprising number of obvious bugs. Java (and similar languages) have many language features and APIs which are prone to misuse. Simple automatic techniques can be effective at countering the impact of both ordinary mistakes and misunderstood language features. Figure 3.3 shows the graphical user interface of FindBugs. A complete manual can be found at http://findbugs.sourceforge.net/manual/.

3.3.2 Coding Standards Tool

Checkstyle is a development tool that helps programmers to write Java code that adheres to a defined coding standard. It automates the process of checking Java code to assist in maintaining common code structure and improve readability. Some recommendations from checkstyle simply define a standard way to layout code or to name classes or variables,
3.3.3 Issue Tracking Tool

FastFix will use Trac to track issues and feature requests. Trac is an enhanced wiki and issue tracking system for software development projects. Trac allows hyperlinking information between a bug database, revision control and wiki content. It also serves as a web interface to the FastFix revision control system. The main features of Trac include:

- Project management (roadmap, milestones, etc.)
- Ticket system (bug tracking, tasks, etc.)
- Timeline of all recent activity
- Wiki system
● Customized reporting
● Web interface

The Trac Roadmap View can be used to map the bug tracker to the project milestones. It allows to add a description to milestones describing main objectives, for example. In addition, tickets targeted for a milestone are aggregated, and the ratio between active and resolved tickets is displayed as a milestone progress bar. It is possible to further customise the ticket grouping and have multiple ticket statuses shown on the progress bar.

The roadmap can be filtered to show or hide completed milestones and milestones with no due date. In the case that both show completed milestones and hide milestones with no due date are selected, completed milestones with no due date will be shown.

3.3.4 Quality Management Tool

Sonar is an open platform that helps analyzing and visualizing code quality in Java projects. It is the result of the addition of several open source tools (Checkstyle, PMD, FindBugs, Clover, Cobertura), becoming a static code analysis tool by gathering and presenting in an unified way an interesting amount of metrics. It provides a dashboard, trend reports, and drill downs to help visualize the state of a software project’s code quality, allowing to represent in a detailed way the weaknesses of analyzed projects.

On the start page (Figure 3.5) Sonar shows a list with all the projects that are using Sonar along with a summary of the metrics of the project.

![Figure 3.5: Sonar Home.](image-url)
By clicking on the project name, a more detailed dashboard is shown, where some metrics like rule compliance, code coverage, lines of code, and comments in code are displayed. The dashboard is shown in Figure 3.6.

Each of the items of the dashboard are links to a more detailed analysis. From the “Libraries” menu item, Sonar shows information about transitive dependencies of components and the version and scope associated with the compiling process.

Sonar not only settles for static reports, but it also offers data about the evolution of projects over time. Specifically, the section “Time Machine” (Figure 3.7) reproduces the performance metrics during recent months or years. Historical data is not restricted to code coverage, rules or complexity, but it allows temporal analysis over 30 different metrics.
Sonar is a powerful tool in terms of quality management of development projects, which allows to perform comprehensive analysis in an easy way, and which visualizes the results of this analysis as a whole.
Sonar will be used to perform checks on a regular basis as one of the continuous integration tasks. To this end it will be included into the Hudson infrastructure to be automatically used in nightly builds. The performed checks include the execution of the "Sonar way with Findbugs" quality profile, which contains 480 rules, including FindBugs, Checkstyle and PMD checks. It is also be possible to launch these checks on demand by clicking on the "Run" button in the Hudson tool. The automatic checks are the most powerful, since they provide an enhanceable project, allowing to know the trends and the current state of the project regarding code’s quality.

As can be seen in Figure 3.8 after Sonar processing the “Violations drilldown” section shows the result of the automatic checks, based on three well-known tools: pmd, cpd and FindBugs. Looking further into this section, we can navigate under each category up to the class where the check has raised the violation and view the code with a detailed explanation of the violation (Figure 3.9).
4 Guidelines

As per one of the open dictionary available online, guidelines are “a rule or set of rules giving guidance on how to behave in a situation.” These rules are identified and refined over time in order to benefit the people and processes in their endeavors. The same is true for software development process, where immense complexity and uncertainty spread across various phases of the development process. This explicitly enforces the use of well defined guidelines to minimize the risk from getting propagated from one phase to another.

4.1 Development Guidelines

Development guidelines provide a basic set of rules to enforce consistent and standardized coding practices. These guidelines are even more vital in a distributed software development project with teams at geographically separated locations. FastFix uses Java coding conventions [12] provided by Sun, since these are a known standard and the consortium members have previous experience with this standard. Additional guidelines that we describe in the following will refine these conventions.

The guidelines in this section assure code quality and complement the process defined in Section 2.2. They mostly refer to the tools described in Section 3.3.

4.1.1 Design Patterns

Within FastFix, developers will use design patterns where applicable. Design patterns [7] are commonly defined as time-tested solutions to recurring design problems. The term refers to both the description of a solution that you can read, and an instance of that solution as used to solve a particular problem. Design patterns have several benefits:

1. Provide solution to issues in software development using a proven solution.
2. Design patterns make communication between designers more efficient.
3. Facilitate program comprehension.
4.1.2 Code Comments

Comments are lines of codes that help to explain and describe the actions of a certain block of code. Thus comments can describe things that cannot otherwise be clearly expressed in the source language. Eventually they ease comprehension by collecting information in a single place. FastFix developers will comment crucial parts in the source code to help other developers understand their code (see also Section 2.2.2).

In spite of numerous benefits of having properly commented source code, comments can be misleading if not used properly. Thus a few points worth consideration while writing comments are:

1. Comments can get out of sync with the code if people change the code without updating the comments. Thus, a rule of thumb in FastFix should be that comments should always changed together with code.

2. Good comments are hard to write and time consuming, but pay off in long term.

3. Adding comments can be counter-productive if the information provided by them is not relevant to the part of code where they are provided. As a rule of thumb, inline comments should describe the next line of code.

4.1.3 Style Checks

FastFix developers use Checkstyle to maintain a common code structure and improve readability. As seen in Section 3.3.2, Checkstyle defines standard ways to layout code or to name classes or variables and makes code more reliable and better performing. Checkstyle rules will be tested during continuous integration (see Section 3.3.4). However, FastFix developers should try to write conforming code from the beginning and use the Checkstyle Eclipse client to avoid late identification of style flaws.

4.1.4 Javadoc Warnings

Eclipse provides built in support to configure javadoc warnings. These warnings are project specific and can be configured in order to enforce the developers to write informative comments which eventually can be part of javadoc. Eclipse provides a mechanism to automatically show warnings if javadoc comments are not inserted at certain code locations. FastFix will use this mechanism if the constructor of a class does not have a documentation, or a method declaration is not documented in javadoc. Further use cases will be explored during development.
D2.2: Integration Plan and Technical Project Guidelines

4.1.5 Code Templates

FastFix will use Eclipse code templates that will be elaborated throughout the project. The Code Template functionality provided in Eclipse goes beyond simple snippets. Templates allow the quick generation of commonly used code, as well as easy customization. This section outlines some of the more beneficial features of templates in Eclipse.

1. Integration templates are directly integrated directly into the Eclipse workflow. Templates are available when performing such tasks as creating a new document or working in the editor panel. In a way, they combine the functionality of new document templates and snippet functionality.

2. Variables is one of most useful feature of templates in Eclipse. This feature the customization within the automation. Each plug-in that includes template functionality can provide its own set of predefined variables. These variables can insert things such as the current date and time up to the name of the method or class that houses the template. The types and how many predefined variables there are vary from plug-in to plug-in, but also the ability to create custom variables. These custom variables act as placeholders in template for which the values can be provided when the template is inserted.

3. Finally, besides the previously mentioned conveniences, there are a couple of other benefits of templates worth noting: consistency and validity. So, if developers use a template, they know that they are writing the same code in the same way each time they use it. This consistency gives code enhanced predictability.

4.2 Integration Guidelines

In order to facilitate system integration, developers should obey several guidelines. As discussed in Section 2.1.2, FastFix developers can have two different roles. Developers write code and implement public interfaces, whereas architects design and refine the architecture and interface specification. In terms of integration, these two roles perform different activities and need to obey different integration guidelines. Section 4.2.1 briefly summarizes how developers have to work in order to support the FastFix continuous integration environment. The remaining sections provide integration guidelines for FastFix architects.

4.2.1 Development

As described in Section 2.1.1, the success of a continuous integration approach depends on a suitable integration process and the appropriate infrastructure as well as integration
guidelines for the developers. Section 3.2.3 comprises an according technical description that should support developers during development and integration activities. This section covers integration guidelines for developers.

First, developers should regularly commit to the baseline (cf. key success factors for continuous integration in Section 2.1.1.3). Regular commits can significantly reduce the number of conflicting changes. Changes of relatively short time periods (usually about one week) already provoke conflicts that are very difficult to resolve, if not even impossible. Conflicts that comprise only small parts of the systems trigger developers’ communication about the changes they are making. Common practice is to commit all code changes at least once a day, or once per feature or component, and in addition performing a nightly build. FastFix developers should obey this practice.

Second, developers should develop according to development and security guidelines, and try to perform according tests (see e.g. Section 4.1.3) before committing to the source code repository to speed up integration. Although the continuous integration system performs checks about the compliance to the given guidelines, multiple refactoring commits can be avoided by testing and checking already in the working copy.

Third, developers are not allowed to change the FastFix architecture or interface specification. Further, they may not add new dependencies to other FastFix components or third-party components (import). These tasks should be performed by FastFix architects. Developers should create an according ticket in the issue tracker instead.

4.2.2 Interface Specification

As seen in Section 2.1.3, object-oriented software concepts demand the decoupling of interface specification from its implementation in order to make requesting applications independent from internal modifications [11]. In Java it is not possible to inherit multiple classes but to implement multiple interfaces. It is very convenient to change the implementation of an existing class and add the implementation of one more interface rather than changing a complete class hierarchy.

FastFix architects will provide public interfaces for all components that are to be used by other subsystems. Internal implementations should be capsuled in a separate place and implement these published interfaces.

In the following we describe exemplarily how component interfaces are specified in Eclipse with OSGi. In our example, an application has to be patched by the Application-Bridge component. The ApplicationBridge itself does not create patches, but requests them from the PatchGenerator component. As shown in Figure 4.1, the PatchGenerator component provides the interface IPatchService for generating a patch. The IPatchService itself contains a method +getPatch() that actually generates and returns a patch.
The ApplicationBridge acts as service consumer and uses the interface IPatchService.

Figure 4.1: IPatchService provider and consumer.

Figure 4.2: OSGi component structure of ApplicationBridge and PatchGenerator.

The ApplicationBridge acts as service consumer and uses the interface IPatchService.

Figure 4.2 shows the according structure of both components realized in OSGi. The PatchGenerator component acts as service provider, the according package org.fastfix.patchGenerator contains the interface IPatchService.java. The package itself is exported and thus the interface IPatchService becomes visible to other components. The class PatchImpl.java is the concrete and private implementation of the interface IPatchService. The package org.fastfix.patchGenerator.internal should therefore not be exported. The ApplicationBridge component acts as service consumer and uses the IPatchService interface. Figures 4.3 and 4.4 depict the concrete realization of provision and consumption of the IPatchService service in OSGi.

FastFix architects will also use the Facade pattern [15] if possible. The purpose of the Facade pattern is to provide a unified interface to a set of interfaces in a component.

```java
public void start(BundleContext context) throws Exception {
    super.start(context);
    plugin = this;
    System.out.println(context.getBundle().getHeaders().get(Contents.BUNDLE_NAME) + " starting...");
    IPatchService patchService = new PatchImpl();
    context.registerService(IPatchService.class.getName(), patchService, new Hashtable());
    System.out.println("Service registered: PatchService");
}
```

Figure 4.3: Implementation of OSGi service provision in PatchGenerator.
Facade defines a higher-level interface that makes the component easier to use. Some of the many benefits of using facade pattern can be listed as:

- Hides the implementation of a component from its clients, thus making it easier to be used.
- Promotes weak coupling between a component and its clients. Thus allowing to change a component without affecting its clients.
- Reduces compilation dependencies in large and complex systems.
- Eases porting systems to other platforms.
- It does not add any functionality but simplifies interfaces and improves reuse.

### 4.2.3 Service Oriented Architecture

In order to understand Service Oriented Architecture (SOA) the initial step is understanding software architecture. Software architecture is influenced by a collection of decisions about a system to meet the requirements of various stakeholders. Architecture includes the components, their properties and attributes, and their interactions and behavior. Depending on the size of a project, its architecture is presented on different levels of abstraction. In many projects the architectural design document provides a common understanding between project partners and a carefully planned process to realize the solution for a given real world problem.

FastFix architects will target a SOA approach. SOA is a special type of architecture that provides a collection of services. Each service is a well defined, self-contained function which does not depend on the context or state of other services. Services communicate with each other using a simple “publish, find, use” approach. Where a service provider “publishes” its service in a service directory where a service user can “find” it and then “use” it. Some of the key characteristics and principles in SOA [2] are:

- Loose Coupling
• Location Transparency

• Protocol Independence

FastFix will be built based on the OSGi framework, which facilitates the design of SOA applications.

4.2.4 Dependency Management

Dependency management is a major concern in plug-in based software development. Dependencies should be managed in a proper way to avoid circular dependencies between plugins. Further introducing a dependency on a software library supplied by external supplier or any open source project requires careful planning and investigation. License issues, support and maintenance are some of the factors of concern. Additionally several external libraries depends on other libraries thus eventually increasing the dependency, which in turn has severe consequences on software runtime. FastFix uses Maven (see Section 3.2.1) to automatically manage dependencies. Developers are not allowed to introduce new libraries or change dependencies. Architects should check dependencies in Maven when introducing new external libraries and avoid adding the same library multiple times. Further they should ensure that third-party libraries conform to the FastFix license standard.

4.3 Security Guidelines

Every software development project must keep security in mind, and it must fit like any other task and, in some cases where privacy is a key factor (and FastFix is a good example), it must be present to a larger extent.

The security guidelines to be followed in this project are based on the need to ensure that security is included in the Software Development Life Cycle (SDLC), since it is proven that trying to fix security code issues in later steps of the development life cycle, will produce higher cost that in earlier levels.

The main ideas of FastFix security guidelines should be provided by the spirit of the Open Web Application Security Project (OWASP) Application Security Verification Standard (ASVS). OWASP is an open-source community project that includes corporations, educational organizations, and individuals from around the world. This community works to create freely-available articles, methodologies, documentation, tools, and technologies. ASVS is an international standard, although created as a way to normalize

\[1\text{http://www.owasp.org}\]

\[2\text{http://www.owasp.org/index.php/OWASP\_Application\_Security\_Verification\_Standard\_ASVS}\]
the range in the coverage when it comes to perform web application security assessments. Most of its concepts can be exploited to define security guidelines in development projects even if they are not web applications.

Figure 4.5: OWASP Application Security Verification Standard [13].

The procedure is similar to the one presented in Figure 4.5, with the purpose of including security support within the SDLC. First, as described in [1], we have defined the security requirements in the scope of the project. Next, in the design phase, these security requirements will be closely watched to ensure that all design decisions taken in the project are fulfilling the mentioned requirements. All the risks and threats will also be identified. Once the implementation starts, security audits will be performed to discover vulnerabilities using black box testing or ethical hacking. The running system will be tested remotely, without using knowledge of the inner workings of the application itself, to find security vulnerabilities. Security testing will be done following the recommendations of a recognized open standard as the OSSTMM. The test will be done using automated penetration methods and tools as well as manual audits.

Best practices and possible errors in code will be assured mainly with the static code analysis, but regarding security, the source code must be reviewed to verify that the proper security controls are present and that they work as intended. Code review is the most effective technique for identifying security flaws. As explained in the OWASP Code
Review Guide, security code review is a method of assuring secure application developers are following secure development techniques. Tools can be used to perform this task but they always need human verification, since tools do not understand context, although they are good at assessing large amounts of code and pointing out possible issues, which must be verified to determine if each of them is a real issue.

While manual code reviews can find security flaws in code, they suffer from two problems. Manual code reviews are slow, covering 100-200 lines per hour on average. Also, there are hundreds of security flaws to look for in code. Source code analysis tools can search a program for hundreds of different security flaws at once, at a rate far greater than any human can review code. However, these tools don’t eliminate the need for a human reviewer, as they produce both false positive and false negative results. As part of this task, automated source code static analysis will be done for each of the developed modules before each integration.

In order to organize the security code review guidelines, we will focus on the most critical security flaws that can be found during a code review:

1. Input validation: It must include protection against cross site scripting, SQL-XPATH-LDAP injection, cross site request forgery, buffer overflow and format bug.

2. Source code design: The first stage where security must be taken into account is source code design. In this category, we must survey security check families like: redundant code, insecure method scope and unused external references.

3. Information leakage and improper error handling: It contains security check families regarding how source code is managing errors, exceptions, logging and sensitive information. Examples of these checks are: Unhandled exceptions or insecure logging.

4. Direct object reference: This category refers to the attacker’s capability to interact with application internals, like direct reference to database data, filesystem or memory.

5. Resource usage: It is related to all the unsafe ways a source code can request operating system managed resources. Most of them are prone to result in some kind of DoS (Denial of Service), like memory leak, unsafe process creation or insecure file deletion or modifying.

6. API usage: It refers to APIs provided by the system or the framework in use that can be used in a malicious way. Hence, we will watch for insecure database calls, HTTP session handling or strings manipulation.

7. Best practices violation: This category gathers together all security violations that don’t fit in previous categories. Most of them contain warning source code best
practices, like unsafe variable initialization or missing comments and source code documentation.

8. Weak Session Management: Some examples are: not invalidating session upon an error occurring, not checking for valid sessions upon HTTP request or not issuing a new session upon successful authentication.

9. Using HTTP GET query strings: Passing sensitive information using a query string and HTTP GET, since no protection can be provided against it, not even SSL.
Bibliography


