(FP7 614100)

D7.1 Test and integration plan

2014-06-30 – Version 1.0

Published by the IMPReSS Consortium

Dissemination Level: Public

Project co-funded by the European Commission within the 7th Framework Programme and the Conselho Nacional de Desenvolvimento Científico e Tecnológico
Objective ICT-2013.10.2 EU-Brazil research and development Cooperation

Document control page

Document file: D7.1_Test_And_Integration_Plan
Document version: 1.0
Document owner: Peter Rosengren (CNET)

Work package: WP7 – IDE Framework for Model-driven development
Task: T7.1 – Test and Integration planning
Deliverable type: R

Document status: x approved by the document owner for internal review
x approved for submission to the EC

Document history:

Version	Author(s)	Date	Summary of changes made
0.1	Peeter Kool (CNET)	2014-04-01	Initial ToC
0.5	Peeter Kool, Matts Ahlsén (CNET)	2014-04-28	Initial Content
0.8	Peter Rosengren (CNET)	2014-05-15	Added testbeds, unit testing
0.9	Peeter (CNET)	2014-06-15	Editing and spell check, versions for internal review
1.0	Peter Rosengren, Peeter Kool (CNET)	2014-06-30	Final version submitted to the European Commission

Internal review history:

Reviewed by	Date	Summary of comments
Enrico Ferrera (ISMB)	2014-06-30	Accepted with minor comments.
Markus Taumberger (VTT)	2014-06-30	Accepted with minor comments.

Legal Notice

The information in this document is subject to change without notice.

The Members of the IMPReSS Consortium make no warranty of any kind with regard to this document, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. The Members of the IMPReSS Consortium shall not be held liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Possible inaccuracies of information are under the responsibility of the project. This report reflects solely the views of its authors. The European Commission is not liable for any use that may be made of the information contained therein.

Index:

1. Executive summary 4

2. Introduction 5

2.1 Purpose and context of this deliverable 5

2.2 Scope of this deliverable 5

3. Test Strategy 6

3.1 Perspectives on software quality 6

3.1.1 Quality models 6

3.2 Testing Approach 7

3.3 Testing Levels 7

4. Unit Testing 8

5. Integration Testing 9

6. Internet of Things Testing 10

6.1 Discovery Interoperability 10

6.1.1 Device, Service and Resource Availability 10

6.1.2 IoT Service Discovery 10

6.2 IoT Service Interoperability 11

6.2.1 Web Service Interoperability testing 11

6.2.2 REST service interoperability 12

6.2.3 UPnP service interoperability 12

6.3 IoT Eventing Interoperability 12

6.4 IoT Exception and Error Handling 14

6.5 IoT Data Security and Privacy Testing 14

7. System Testing 15

8. Software Test Plan 16

8.1.1 Unit Testing 16

8.1.2 Integration testing 18

8.1.3 Internet of Things Testing 19

8.1.4 System Testing 20

9. Integration Plans 21

9.1 Integration Strategies 21

9.1.1 One-Step Integration 21

9.1.2 Incremental Integration 21

9.1.3 Problems with Integration 22

9.1.4 Fundamental Integration Issues 23

9.2 Source code and Configuration management 23

9.2.1 Source Code Management with Subversion 24

9.2.2 General Setup of Subversion for the IMPReSS Project 24

9.2.3 Integration Plan for the IMPReSS software 25

10. Conclusions 28

11. References 29

1. Executive summary

This deliverable defines the basis for testing of platform and applications developed in the IMPReSS project. It is based on the current development plan and architecture. Testing will be undertaken continuously during the project and will be done both for individual components as well as for the integrated platform, the IMPReSS Architecture. The testing of the developed components and the integrated software will be done with regards to a number of quality attributes, including performance, stability and resilience.

The choice of testing approach is dependent on the chosen development approach. The development approach in IMPReSS is iterative and incremental, comprising several complete cycles of analysis, development, and validation in which components from different partners are frequently integrated. It is increasingly clear that the approach to testing in such conditions should preferably be "agile" or "lean" which in turn leads to that IMPReSS should use automatic testing whenever possible.

Testing is typically considered to take place on five different levels:

1. Unit testing – Testing at the lowest level using a coverage tool. A unit is a piece of code that does not call any subroutines or functions developed in the project.
2. Integration testing – Testing of interfaces between components to ensure that they are compatible.
3. Internet of Things testing - Testing of how the components conforms to its intended Internet of Things network. For instance this covers conformances to the different protocols for discovery and service invocation used in a particular IoT setting.
4. System testing – Testing of the entire software system.
5. Validation – Evaluation at the end of development to ensure requirements fulfilment.

The IMPReSS platform will be developed in different environments as well as on different platforms (Java and .NET), with an open SOA based architecture. Web and REST services interoperability is thus essential for seamless integration of individual components. However, as unit testing frameworks do not support web and REST services level tests, a specific testing approach for services will be used including WSDL file testing and Web Service Interoperability (WS-I) testing tools.

System testing on the complete IMPReSS Architecture will be performed using “test beds”. A test bed provides a stable test environment where reproducible tests can be made. The Eventing test bed should be used to verify the performance of the Architecture at large. Moreover, more limited test beds interfacing Wireless Sensor and Actuactor Networks (WSANs) and other Application level resources will be used to ensure that all relevant data generated can be handled in by the Resource Abstration Layer and the Service proxies without congestion.

In line with the agile approach to software development, integration should be performed incrementally and continuously. By use of Source Code Management System such as Subversion, it will be ensured that all developers have access to the latest versions of the code base.

2. Introduction

The goal of WP7 – IDE Framework for Model-driven development – is to ensure that the IMPReSS platform is well integrated, easy to use, directly transferable to product development, and therefore increasing impact of the project in the commercial world. This work package will provide a common framework, guidelines, rules for managing the development of various development tools, and configuration components. This will ensure that the various tools and configuration components are developed consistently and easier to be integrated on one platform. This work package will also conduct the integration of all infrastructure components of the IMPRESS platform and conduct integration testing against use cases defined in the WP2.

2.1 Purpose and context of this deliverable

This deliverable defines the basis for testing of platform and applications developed in the IMPReSS project. It is based on the current development plan and architecture. It should be noted that since IMPReSS is using an iterative and incremental development approach, the test plan may change as a result of evolutions in the project.

Testing is an activity performed for evaluating product quality, and for improving it, by identifying defects and problems (Bourque and Dupuis, 2004). Generally, this may involve all stakeholders (i.e. everyone who has an interest in the IMPReSS platform and its applications: users and administrators but also the companies which use the platform as development base for adding value to the services.

The work package that this deliverable belongs to is concerned, however, with technical testing only, i.e., (developer) user validation is out of the scope of this document. For dealing with this and in the quest for finding an appropriate framework capable at providing the suitable criteria and justification we approach IMPReSS testing with a lightweight/agile combination of testing practices and quality frameworks.

As second part of the deliverable, we define the plan for the integration of the platform and its components developed in the IMPReSS project. Again, since IMPReSS is using an iterative and incremental software development procedure, the integration plan may change as a result of evolutions in the project. In general, the integration serves for the composition and concatenation of different applications. In IMPReSS, parts of the platform which are developed in the technical work packages shall be integrated into one platform.

2.2 Scope of this deliverable

Even though some work packages have individual test plans, in this document we present an integrated test plan for all work packages. This is because we are going to develop one integrated software platform where the outcomes of the particular work packages interact with each other in order to have a single system in the end. We give perspectives on software quality and discuss the testing strategy in chapter 3. Chapter 4-7 then discuss what to test . In chapter 4 we describe our approach for Unit Testing while chapter 5 covers Integration testing. Chapter 6 then describes our approach to test our developed components as part of an Internet of Things ecosystem/network. System testing is described in chapter 7. In chapter 8 we then describe how to do the testing presenting our software test plan. Finally our integration plans are presented in chapter 9.

3. Test Strategy

3.1 Perspectives on software quality

A conception of software quality is a necessary background of testing. Several views on software quality exist including (Kitchenham and Pfleeger, 1996):

- Transcendental view: quality can be recognized but not defined; its presence may be characterized as "the quality without a name"
- User view: quality is fitness for purpose in particular from the viewpoint of users
- Manufacturing view: quality is performance to specification
- Product view: quality is tied to inherent characteristics of the product
- Value-based view: quality is the amount a customer will pay for the product.

The technical quality concept adopted by IMPReSS has a user as well as a manufacturing view component.

3.1.1 Quality models

Given specific views on software quality, IMPReSS developers still need a comprehensive model for the early evaluation of software quality. Again, there are many models of software product quality that define software quality attributes. Three often used models are discussed here as examples. McCall's model of software quality (McCall et.al, 1977) incorporates 11 criteria encompassing product operation, product revision, and product transition. Boehm's model (Boehm, 1989) is based on a wider range of characteristics and incorporates 19 criteria. Criteria in these models are not independent; they interact with each other and often cause conflict, especially when software providers try to incorporate them into the software development process. ISO 9126 incorporates six quality goals, each goal having a large number of attributes.

The criteria and goals defined in each of these models are listed in Table 1. Note that the ISO Model includes a number of criteria under its goal of maintainability.

Criteria/Goals	McCall, 1977	Boehm, 1978	ISO 9126, 2001
Correctness	X	X	maintainability
Reliability	X	X	X
Integrity	X	X	
Usability	X	X	X
Efficiency	X	X	X
Maintainability	X	X	X
Testability	X		Maintainability
Interoperability	X		
Flexibility	X	X	
Reusability	X	X	
Portability	X	X	X
Clarity		X	
Modifiability		X	Maintainability
Documentation		X	
Resilience		X	
Understandability		X	
Validity		X	maintainability
Functionality			X
Generality		X	
Economy		X	

Table 1: Software Quality Models

Of these, the ISO 9126 model (ISO 9126-1, 2001) is the most recently updated and comprehensive in terms of coverage of quality views.

3.2 Testing Approach

In general, there are numerous dimensions of a testing approach and a continuum of values within each dimension (McGregor and Sykes, 2001):

- When will testing be performed? Will testing be done every day, as components are developed, or when all components are put together?
- Who performs the testing? Are developers responsible or are independent testers responsible?
- Which pieces will be tested? Will everything, a sample, or nothing be tested?
- How will testing be performed? Will testers have knowledge of only the specification of the component under test (blackbox testing) or also knowledge of implementation (whitebox testing)?
- How much testing is adequate? Will no testing be done or will exhaustive testing be done?

The choice of approach is dependent on the chosen development approach. The development approach in IMPReSS is iterative and incremental, comprising several complete cycles of analysis, development, and validation in which components from different partners are frequently integrated.

It is increasingly clear that the approach to testing in such conditions should preferably be "agile" or "lean" (Beck, 2000). This means that IMPReSS should use automatic testing whenever possible. In addition “adequate” testing needs to be seen in the context of what is currently needed of the platform. For demonstrators, e.g., the primary objective is to demonstrate partial functionality. Thus, focus is here not on doing a complete acceptance test (or even tests conforming to statement coverage), but rather on integration testing.

3.3 Testing Levels

Testing is typically considered to take place on five different levels (Beizer, 1990):

6. Unit testing – Testing at the lowest level using a coverage tool. A unit is a piece of code that does not call any subroutines or functions developed in the project.
7. Integration testing – Testing of interfaces between components to ensure that they are compatible.
8. Internet of Things TestingTesting of how the components conforms to its intended Internet of Things network. For instance this covers conformances to the different protocols for discovery and service invocation used in a particular IoT setting.
9. System testing – Testing of the entire software system.
10. Validation – Evaluation at the end of development to ensure requirements fulfilment.

In the following we will describe how each of these maps to IMPReSS. Level 1) to 4) are internal (i.e., tests are performed by IMPReSS participants) and 5) is external (i.e., partly performed by means of user applications). The general sequence is 1) -> 2) -> 3) -> 4->5) but the iterative approach of IMPReSS means that there will be interleaving, backtracking etc. of these activities.

4. Unit Testing

A "unit" in IMPReSS is a closed functional part. Unit tests must be automated and be written using a unit testing framework. In the case of Java this is JUnit (JUnit, 2011). In the case of .NET this is NUnit (NUnit, 2011).

Upon completion of an increment of a unit, the following must be considered

- Code checked into the Subversion repository must not break the build process
- Prior to committing new versions of a unit to the Subversion repository there must be reasonable automated, functional unit tests. We do not prescribe any specific coverage criteria for blackbox or whitebox tests.

To enhance quality of the software it is recommended to create a new unit test for each detected bug if this was caused not by an error in the programming but in a misunderstanding of a requirement. After fixing the bug the commit statement has to contain the bug number so that an automated tracking of bug fixes can be performed.

5. Integration Testing

Integration testing takes place whenever multiple units need to work together as one component for instance as part of a software manager. We do incremental integration (as opposed to integrating all units at once) but do not prescribe a specific approach such as bottom-up or top-down integration (Beizer, 1990).

The IMPReSS platform will be developed in different environments as well as on different platforms (Java and .NET), with an open SOA based architecture. Web service interoperability is thus essential for seamless integration of individual components. However, as unit testing frameworks do not support web services level tests, a specific testing approach for services will be used.

6. Internet of Things Testing

The aim of the IMPReSS project is to provide a Systems Development Platform which enables rapid and cost effective development of mixed criticality complex systems involving Internet of Things and Services (IoTS) and at the same time facilitates the interplay with users and external systems. Most of the components developed in the project are intended to be used in an IoT scenario meaning that they cannot make assumptions about how they are being used and by which applications. Therefore they need also to be tested from an IoT perspective and to conform to established IoT standards and principles.

IoT Testing includes the following aspects:

- IoT Discovery Interoperability
  - IoT Ecosystem protocol conformance
- IoT Service Interoperability
  - Web Service Interoperability
  - REST Service Interoperability
  - UPnP Service Interoperability
- IoT Eventing Interoperability
  - Event Format conformance
  - Event Reporting Behaviour
- IoT Exception and Error Handling
- IoT Data Security and Privacy Testing

6.1 Discovery Interoperability

Internet of Things applications are by nature discovery-driven, they need to discover which "Things" are available on the network and how to communicate with the "Things". Discovery involves two main functionalities:

- Discovery of the existence of a "thing"
- Discovery of which IoT services are offered by the "thing"

6.1.1 Device, Service and Resource Availability

An important aspect of any IoT-based application is that devices, sensors and resources in general are stand-alone, self-contained and loosely coupled components. This means that before they can be integrated and used by an application they need to be discovered by the application. For any object (device, sensor, resource et c) that is intended to be used in this way we need to verify that it complies with the discovery mechanisms defined for the specific IoT platform. In the IMPReSS project this is done in task 3.2 "Resource and Service Discovery".

The conformance testing will be done using the LinkSmart DAC Browser which discovers any well-behaved LinkSmart IoT-enabled object.

6.1.2 IoT Service Discovery

The second aspect of discovery is that objects (such as devices and sensors) need to be able to communicate/report which IoT Services they support. This mechanism also needs to conform to the specific IoT platform service discovery mechanisms.

The conformance testing will be done using the LinkSmart DAC Browser which discovers any well-behaved LinkSmart IoT-enabled object.

6.2 IoT Service Interoperability

IoT devices, sensors and resources are controlled and used by invoking the different services they offer. Three types of service protocols needs to be verified:

- Web Services
- REST Services
- UPnP Services

6.2.1 Web Service Interoperability testing

The Web Service interoperability tests to be made on the IMPReSS platform can be divided into two main groups:

- Testing done on WSDL files between components during development and integration. As the IMPReSS platform is based on two different environments (Java and .NET) there may be initial interoperability problems with regard to the WSDL files and service invocations.
- More formal testing of components using the Web Services Interoperability Organization (WS-I) tools. These test both design time interoperability (based on a WSDL file) and run-time interoperability (whether the web services responds according to WS-I at run-time).

The Web Services Interoperability Organization (WS-I) has developed testing tools that evaluate Web services conformance to Profiles. These tools test Web service implementations using a non-intrusive, black box approach. The tools focus is on the interaction between a Web service and user applications.

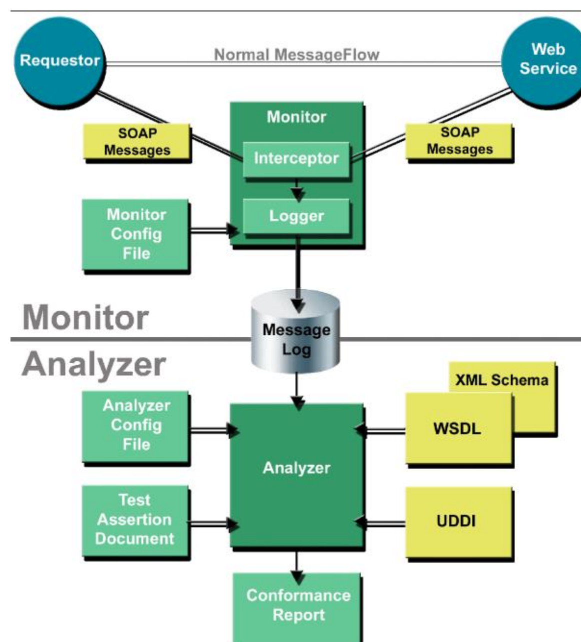

Figure 1: Web Service Testing Tools Architecture

The testing infrastructure is comprised of the Monitor and the Analyzer and a variety of supporting files (see Figure 1):

- Monitor - This is both a message capture and logging tool. The Interceptor captures the messages and the Logger reformats them and stores them for later analysis in the message log. The monitor is implemented using a "man in the middle" approach to intercept and record messages.

- Analyzer – This is an analysis tool that verifies the conformance of Web Services artefacts to Profiles. For example, it analyses the messages sent to and from a Web service, after these have been stored in the message log by the Monitor.
- Configuration Files – These are XML files used to control the execution of the Testing Tools: Monitor Configuration File – controls the execution of the monitor
    - Analyzer Configuration File – controls the execution of the analyser
    - Test Assertion Document – defines the test assertions that will be processed by the analyser
    - Other files or data artefacts will be accessed, which are not part of the test framework, but dependent on the Web Service to be tested:
- Web Service artefacts – these inputs to the Analyzer are target material for testing, and will be reported on:
    - Message Log – contains the monitoring trace of messages captured at transport level.
    - WSDL definitions - contains the definitions related to the Web Service
    - UDDI entries - contains references to Web Service definitions, as well as bindings.
- Generated Files – These are XML files produced by Testing Tools, that are specific to the Web Service being tested: Message Log – (also a "Web Service artefact")
- Conformance Report – contains the complete conformance analysis from the specified inputs.

The WS-I defines a number of assertions that are checked and reported in the Conformance Report. A sample WS-I assertion is shown below,

Assertion: BP2416

Result	**Passed**
Assertion Description	Every QName in the WSDL document that is not referring to a schema component is either using the target namespace of this WSDL or the target namespace of a directly imported WSDL component.

To find explanations for all assertions, see (WS-I, 2011).

6.2.2 REST service interoperability

A popular architecture style is the RESTful based approach. In such an approach a resource based model is employed. An IoT Resource is described in terms of its properties. The state and behaviour of the resource can be monitored through simple http GET, POST and PUT commands.

REST service interoperability will be tested using standard web browser extension tools such as RESTClient to verify conformance to the defined resource model. It will also be tested for the compliance with the defined XML and JSON output.

6.2.3 UPnP service interoperability

Universal Plug and Play is a standard for discovery and service invocation in local area networks. In case UPnP services have been used they will be tested using the LinkSmart DAC Browser which discovers any well-behaved LinkSmart IoT-enabled object and can invoke its UPnP services.

6.3 IoT Eventing Interoperability

Eventing interoperability is important since most IoT-based systems are based upon events as the main means for distributing different measurements and observations to the consumers of data. Eventing interoperability involves several aspects that need to be tested:

- **Event format:** The events generated by devices and components in the system should follow the agreed schema for the events. This will be done by intercepting events and checking them against the agreed schema.
- **Event Meta content:** This means to test that meta information contained in the events hasthe correct information, for instance, that the device id in the message is correct. Other parts to check are the format of the Timestamp of the event. This will be tested by message interception and validated by checking the values provided.
- **Event payload content:** This means that actual payload of the message is correctly formatted. The values supplied have to match the indicated data type and also match the indicated unit. This will be tested by message interception and validated by checking the values provided.
- **Event frequency:** In an event driven IoT based application is important that the event providers do not create redundant events. For instance a thermometer device should only report changes to the temperature when issuing the event instead of creating an event for each sampling of the sensor. The main rationale behind this is to avoid event flooding of event consumers as well as increasing the overall scalability of the system. For some devices this is not possible and in those cases special event management is needed or one has to rely on pulling the data from the device rather than on eventing. This will be tested by using the Event Trace and Debug Tool (see description below) that creates a the database of the events transmitted to an event manager. This database will be analysed to validate that the event producer does not create redundant events.

The Event Trace and debug tool provides the capabilities of eavesdropping on all event communication at a LinkSmart event manager. This tool is not part of the LinkSmart event manager, instead it as run side by side with the LinkSmart Event Manager and can be turned off/on completely independently. This tool will be used for all the eavesdropping needed to be done to test the IoT Eventing Interoperability.

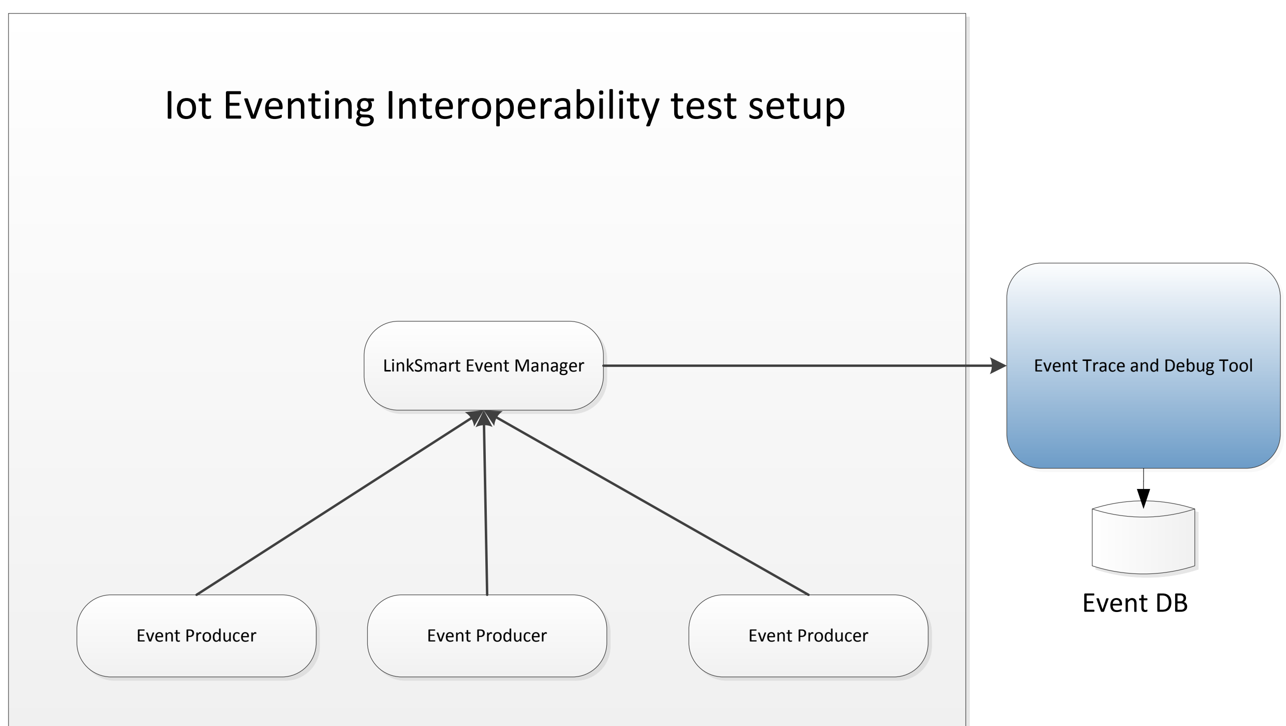

Figure 1: IoT Eventing Interoperability test setup

In the IoT Eventing test setup, see Figure 1 above, The Event Trace and Debug Tool acts as a normal event consumer from the LinkSmart Event Manager but it listens to all events that pass through the Event Manager without any filtering and stores them in a database. Because of this it is essential that the Event Trace and Debug Tool is well behaved and does not introduce problems by listening i.e., it should be transparent for the system and the system behaviour should not change when the ETDT is used.

The test setup can include a number of event producers or a single producer depending on the type of the test as well as the type of the device/resource considered.

6.4 IoT Exception and Error Handling

Interaction with the physical world is unreliable. Therefore software artefacts representing the physical things need to be sensitive to malfunctions and errors. They need to communicate to other objects when they experience an exception or enter into an error state.

Exception and error handling will be tested using physical tests like simply shutting down or unplug devices from their power and observe the corresponding IoT software behaviour.

6.5 IoT Data Security and Privacy Testing

Devices, sensors, resources and applications exchange data that might be more or less sensitive from a security and privacy perspective. We need to ensure the IoT objects that have received data over a secure channel don´t store or propagate this data in an unsecure manner.

7. System Testing

On a system level, use case/high level functional requirement and non-functional/quality requirements will be tested. These tests will be automated whenever possible. An IMPReSS system level test concerns complete distributed Internet of Things applications composed from many cooperating nodes (aka the "IMPReSS architecture").

In order to test quality criteria on the complete IMPReSS Architecture, "test bed" techniques will be used. A test bed provides a stable test environment where reproducible tests can be made. The aim with this testing is to verify the architecture in terms of opportunistic networking performance and resilience. The IMPReSS test beds will be designed for this limited scope based on early proof of concept software avoiding the need for major additional development resources.

8. Software Test Plan

The test plan describes how to do the testing regarding unit, integration, IoT and system testing. Activities, frequency, and responsibilities are as summarized in Table 2.

Level of Testing	Activities	Frequency of Testing	Responsible Party
Unit	Select test cases Write automated test cases	Test creation as units are ready Automated tests run continuously	Unit developer
Integration	Select test cases Manage unit dependencies Write automated test cases	Automated tests run continuously	Increment developers
Internet of Things	Discovery tests WS-I and WSDL test REST Test UPnP Test Event testing Exception testing	Discovery, each iteration using DAC Browser WS-I, each iteration WSDL, Continuously Event, each iteration Exception, each iteration	Technical management
System	Select test cases from requirements Prepare test beds and test data Run test beds	Manual test at end of increment Automated tests run continuously At each iteration	Technical management

Table 2: Project Test Plan

8.1.1 Unit Testing

We recommend that the developers use unit tests as applicable in all IMPReSS developed software and to use the template in Figure 3 below to document the different unit tests as well as their limitations.

```
/**
 * <h1>Test of <Unit></h1>
 *
 * $Date$
 *
 * <h2>Objectives</h2>
 * <Objectives for unit and test>
 * <h2>Strategy</h2>
 * <Strategy for test>

 * <h2>Status</h2>
 * <Status of test>
 *
 * <h2>Responsible</h2>
 * <Unit and test responsible>
 *
 */
```

Figure 2: *Unit test report template*

An example of usage of this template is shown below:

```
/**
 * <h1>Test of EventManager</h1>
 *
 * $Date: 2007-06-19 16:43:03 $
 *
 * <h2>Objectives</h2>
 * This manager implements topic-based publish/subscribe. Testing objectives
are:
 * <ol>
 * <li>Test implementation of topic matching
 * <li>Test remote call to EventManager
 * <li>Interoperability testing (JAX-WS, Apache Axis, and .NET)
 * </ol>
 *
 * <h2>Strategy</h2>
 * Whitebox automated test for 1) and 2). 3) is manual

 * <h2>Status</h2>
 * Partially done. Interoperability testing remains to be done
 *
 * <h2>Responsible</h2>
 * @author Klaus Marius Hansen, klaus.m.hansen@daimi.au.dk
 *
 */
```

Figure 3: Unit test report example

With JavaDoc this results in the output shown below.

Test of EventManager

$Date: 2007-06-19 16:43:03 $

Objectives

This manager implements topic-based publish/subscribe. Testing objectives are:

1. Test implementation of topic matching
2. Test remote call to EventManager
3. Interoperability testing (JAX-WS, Apache Axis, and .NET)

Strategy

Whitebox automated test for 1) and 2). 3) is manual

Status

Partially done. Interoperability testing remains to be done

Responsible

Author:

Klaus Marius Hansen, klaus.m.hansen@daimi.au.dk

Figure 4: Output generated from unit test report example

8.1.2 Integration testing

A best practice in integration and system testing is that the developers who wrote parts of the system/integrated unit should not write and perform the tests. Given that IMPReSS is a large, complex project it will almost always be the case that an integrator did not write part of the units being integrated and thus the integrator is permitted to write and perform the tests.

Integration using web services poses some special problems when testing since it is not easy to use unit testing since all services need to be deployed in order to test the interfaces.

One possible solution is to have a central test environment where all managers and other components of the IMPReSS architecture are deployed. The drawback of this solution is that maintaining a centrally deployed test server might be quite time consuming especially when we are talking about multiple deployment environments such as.Net,.Net CF, Java and JavaME.

Another easier solution is that each partner that develops web services would maintain one or more test servers with their web services deployed which could be used for testing purposes. The main drawback of this solution is that there can be inconsistencies in-between the deployed manager and the one in the SVN. But it would still enable us to use unit testing of individual managers WS communication to other managers.

Some guidelines can be noted to ease the integration tests:

- Each manager with a WS interface should have its WSDL file available directly in the SVN together with any referenced schemas/WSDL files. It should not be necessary to deploy the manager in order to get hold of the WSDL file. The endpoint defined in the WSDL file should point to the partners test server where the service is deployed.
- There is also a need to have all endpoints used by clients available in a configuration file. For.NET components this is done automatically and to either web.config or <executable_name>.config depending on the project type. Java clients must add these manually to a config file.
- Each Web Service should be tested using the WS-I test tools in order to make sure that the service is invokable using all standard WS clients.

8.1.3 Internet of Things Testing

The exception is the proposed "test beds" described below which will be used for verifying the different quality attributes with regards to the IMPReSS architecture.

Currently we foresee three different test beds:

- Discovery test bed
- Service interface test bed
- Eventing test bed
- Exception test bed

Discovery Test bed

The discovery test bed consist of a continuously running "Discovery Manager" and a LinkSmart DAC Browser which will detect and display new IoT Devices, sensors and resources. If a new object is not discovered properly using the defined protocols within a defined time frame it will be reported as not working properly.

The discovery test bed will be divided into two parts. The first is focused on the physical device discovery such as Bluetooth discovery. The second part is focused on the application discovery, to check that the discovered physical devices are also presented in the network in a proper way to be discovered by the applications needing them.

Service Interface test bed

The main objective of Service Interface test bed is to verify the performance and architecture of the wireless sensor network interface in IMPReSS. The following quality attributes will be addressed in this test bed:

- Connectivity
- Throughput

This test bed will be used to measure the possible throughput to Device Gateways. The objective is that the Device Gateways should outperform the sensor networks in terms of throughput, i.e. the bottlenecks should be in the sensor networks and not in the Device Gateway.

This test bed should be able to use a relatively simple setup, basically only requiring a single Device Gateway and wireless sensor network.

Eventing test bed

The main objective of this test bed is to verify the Eventing architecture and software to be able to measure the following quality attributes:

1. Delay tolerance
2. Throughput
3. Resilience

The aim is to use this test bed to refine strategies for opportunistic networking including store and forward techniques and their impact on throughput etc. The setup of this test bed will basically be limited to three nodes:

- Device Gateway Node. This is the node where the Resource Abstraction Interface resides.
- IMPReSS Data Fusion Gateway. This node is acting as an intermediary data integrator, it is only needed in case of high demands of throughput where processing needs to be distributed over several nodes.
- Relevant parts of the Central IMPReSS node, i.e. Network Manager, Event Manager and Event DB.

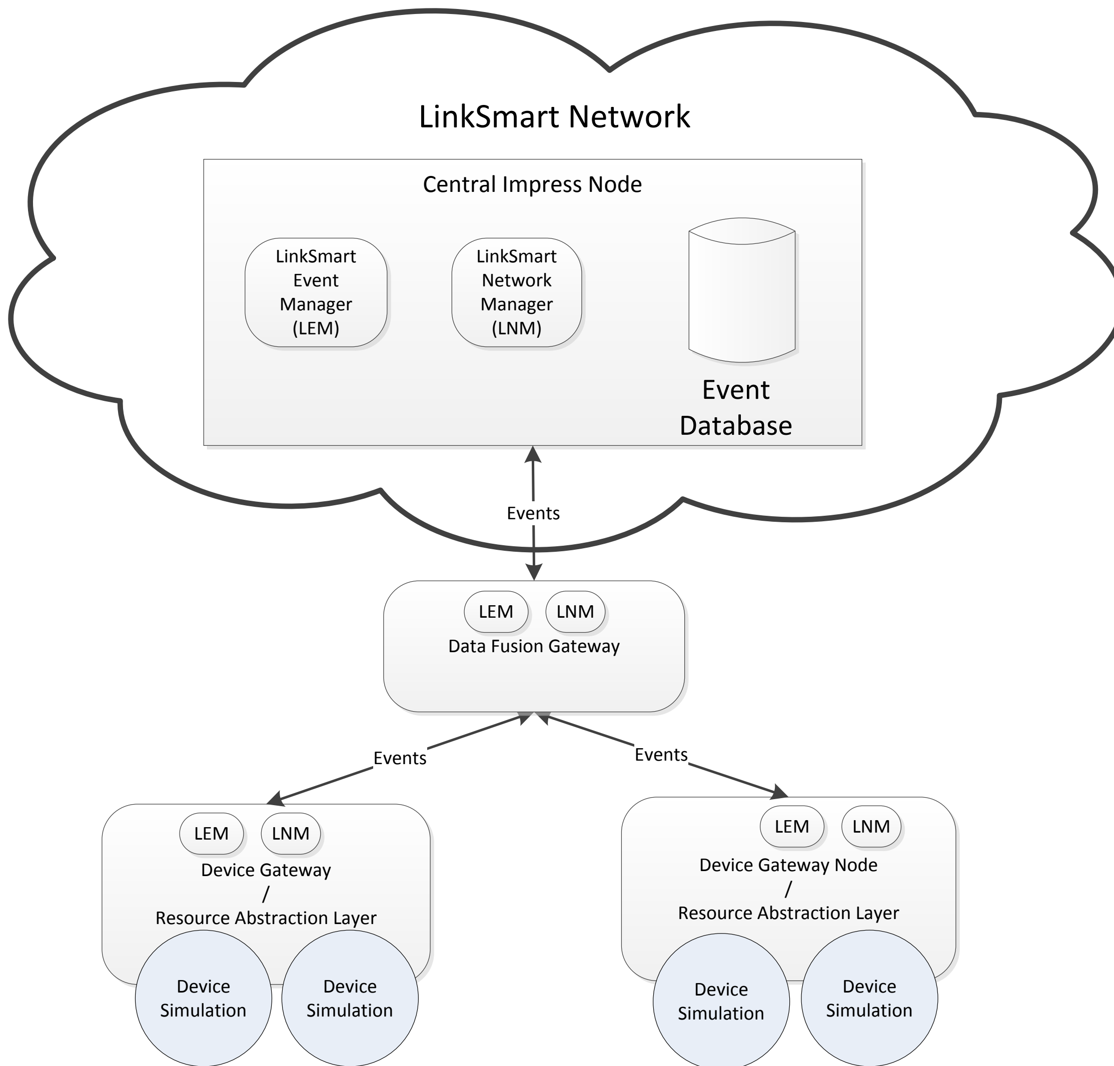

Figure 5: The Eventing test bed.

In order to run this test bed we need to create a number of virtual devices that will be creating events. These virtual devices will be quite trivial and will consume limited development resources.

Exception Test bed

The Exception test bed consists of a set of physical devices connected using the LinkSmart/IMPRESS platform. When new IMPReSS software is deployed, these devices will be randomly unplugged to generate exceptions and errors and the different objects behavior will be observed.

8.1.4 System Testing

System testing will be performed by end user applications developed in WP8 and the deployment of these.

9. Integration Plans

This section describes the plans for the integration of the software and hardware into the overall IMPReSS system platform. Section 4.1 describes the overall integration strategy, while Section 4.2 describes the software source code management as part of the continuous integration cycle.

Definition: Integration

Integration is the process of combining software components, hardware components, or both into an overall system. (IEEE Std 610.12, 1990).

9.1 Integration Strategies

The integration of IMPReSS software will be performed continuously and incrementally. IMPReSS's continuous integration approach splits up into a cycle of eight steps described in detail in this section.

The integration plan bases on the current software architecture description of IMPReSS and therefore, the plan must evolve along with the software architecture description. However, the approach and overall plan will remain the same.

After the components of a system are developed and tested individually, they can be joined together. In the course of the integration, these components build a complete and functioning software system. The plan of component assembly can be found within the software architecture, because the architecture determines the components, the interfaces between these components and the external interfaces. This "meta-model" covered by the architecture provides levels of integration, that are typically arranged in a hierarchy.

We will now describe different possible integration strategies. As long as the system is not completely integrated, the result of each integration step is denoted as "partially integrated system". In general, a partially integrated system is not executable, and therefore, an integration test requires test drivers and placeholder (stubs). A test driver provides the partially integrated system with test input, and a placeholder acts in the place of a component that is required by the partially integrated system, but that is not integrated, so far. A placeholder either delivers constant values or simulates the behaviour of the replaced component more or less realistic. Depending on the chosen integration strategy, the number of required test drivers and placeholders increases. The development of intelligent placeholders that simulate the behaviour of the missing component is more complex than the development of test drivers. The integration strategy needs to take this aspect into account.

9.1.1 One-Step Integration

In principle, it is possible to integrate all (or many) components in one single step. This approach is also called "big-bang-integration". Test drivers and placeholders are unnecessary, because after the integration the system is complete and testable without any additional aid.

Nevertheless, integration in one step is marginally considered in practice. If each component exhibits errors with a certain probability and inconsistencies introduce additional errors, the testers need to work with a system that is barely executable. Uncovering errors becomes complex, since these errors are spread in a large system, which is barely known to the testers. Even if the developers themselves participate in the testing, they only possess knowledge of their components. Most software development approaches prefer an incremental integration as described in the next paragraph.

9.1.2 Incremental Integration

The incremental integration is performed on a case-by-case basis. Depending on whether the integration starts with the main class or with the basic components, the integration is called "top-down" or "bottom-up".

Top-Down Integration

The top-down integration follows the hierarchical structure of the architecture. If the architecture defines a layered structure, the main class and the components hosted by the upper layer are integrated first. Thereafter, the components on the next underlying layer are integrated and so forth. The advantage of this strategy lies in the fast availability of an executable system. On the other hand, the disadvantage is that potentially many placeholders need to be constructed. In order to make the system truly executable, these place holders need to offer a certain level of functionality and accordingly their construction is complex.

Bottom-Up Integration

With the bottom-up integration those components are integrated first, that do not rely on services of other components. Thereafter, components are added that use these basic components. In the last step, the main class is attached. If possible, the "use"-relationship between the components should be followed strictly backwards: one component is integrated after all required components are available in the partly integrated system. Solely cyclic dependencies require placeholders or all cyclic dependencies need to be integrated at once. The benefit of this type of integration is that none or only few placeholders are required. Thus, the effort for an integration test decreases, because only one new test driver is necessary for each step. The testing of the complete system becomes possible not until the last integration step.

Continuous Integration

A different approach arises from the Extreme Programming paradigm. In order to rapidly build an executable system and to enable the developers to equally work together on different parts of the system, new and modified components need to be continuously integrated in a simple and fast manner. As soon as a new component is completed, it is integrated and tested. The integration takes place within a separated environment – on a dedicated integration machine – which is detached from the development environment.

The continuous integration approach implies frequent integration and testing of work results typically once a day (daily builds). Only if the test does not reveal any errors, the new code remains on the integration machine. Otherwise, the code must be removed and the prior state of the system is reconstructed.

This type integration requires a close cooperation of all developers. Because the integration happens frequently, the individual extensions and modifications need to be small. Otherwise, each single integration step takes too long and the integration becomes a bottleneck. The short work cycles require use of automated methods for source code management, configuration and testing.

9.1.3 Problems with Integration

In an ideal world the integration constitutes a simple task: the developers realise their individual components independently from each other. Because they precisely follow the specification and the syntactical level fits anyway through strict type checking, all parts of the system fit together as it has been planned. The effort of integration is only marginal. In practice, two reasons are opposed to a trouble-free integration:

- Usually the specification is incomplete and spongy. Thus, the specification is interpreted differently and sometimes misunderstood by the developers. The result is a set of components that do not fit together.
- Usually a software system includes third party components. In many cases information about these components is entirely insufficient. In addition, such components are not available on time.

First after the integration, inconsistencies and contradictions become visible. If integration is performed at a late point in time, the analysis and correction of defects or the substitution of corrupt components might be expensive and happens in a "panic mode". Modern programming languages and methods allow for a minimisation of such problems. Strict type checking guarantees the

syntactic compatibility of the components, but semantic compatibility can only be achieved through accurate planning and specification of the parts before they are implemented.

9.1.4 Fundamental Integration Issues

The following rules can help to avoid common errors and weaknesses of the integration:

Plan the Integration

Only if the integration is planned explicitly, a reasonable and realisable sequence of integration steps arises. An integration plan is a prerequisite for building the system with as small effort as possible.

Integrate Early and Often

One approach of integration constitutes in launching the integration before the implementation started. The designed components are realised as placeholders that only match the design with regard to interfaces, and during the implementation they are filled with code. In this way, an executable system is achieved at an early stage.

Capture the Total Effort of the Integration Precisely

A shallow or non-existent integration plan leads to massive problems that put a burden on the budget and schedule of the entire project. Even if an integration plan exists, the planned resources and time frame is underestimated in many cases.

Coordinate the Integration with the Project Organization and Development Process

Particularly large systems that are developed in a distributed manner require the architecture to consider the organization of the development process. This in turn influences the integration strategy. The planning of the integration involves the consideration whether the system is developed incrementally or as a whole.

Identify and Reduce the Risks of the Integration

In particular the integration of third party components requires a plan of how to deal with low-quality delivery or delivery behind schedule. Possibly other, more elementary components can be integrated instead, or only a leaner version of the system is completed at first. The approach to such problems needs to be prepared in advance.

9.2 Source code and Configuration management

An important part of modern software development is Software Configuration Management. Configuration Management is often considered to have originated in the frame of software development in recent years and that it merely consists of a tool for versioning of source files. In software development the core of configuration management is made up of a version control system. Source code (or all source documents of the final product, respectively) is considered a system that spans time and space. Files and directories (which, of course, can contain files) form the space, while their evolution during development forms time. A version control system serves the purpose of moving through this space (Spinellis, 2003).

Configuration management contains version control, and extends it by providing additional methods of project management (Weischedel and Versteegen, 2002). Software configuration consists of software configuration items (SCI) which can be organised in three categories and which exhibit several types of versions. The following activities constitute configuration management (Bruegge and Dutoit, 2000):

- identification of SCIs and their versions through unique identifiers
- control of changes through developers, management or a control instance,
- accounting of the state of individual components,
- auditing of selected versions (which are scheduled for a release) by a quality control team.

The Institute of Electrical and Electronics Engineers (IEEE) sees configuration management as a discipline that uses observation and control on a technical as well as on an administrative level. Software configuration management deals with the governing of complex software systems (Westfechtel and Conradi, 2003). In IMPReSS we base the software configuration management on Subversion which addresses many of the problems listed in (Ommering, 2003; Bruegge and Dutoit, 2000 and which is described in the following subsection.

9.2.1 Source Code Management with Subversion

It is common practice to immediately commit every change to a revision control system, no matter how small it is. The rationale behind is that other developers should always work with the latest version of the code base. For source code management we have set up a Subversion (SVN) repository (http://subversion.tigris.org/) where all artefacts of the development process will be stored and organised. Subversion, like many other version control systems, manages different versions of documents by calculating editing operations between these and then only saving these operations. This approach usually leads to very good compression results.

We use Subversion because it is a modern revision control system. It has been designed to resolve many of the short-comings of widely-used Concurrent Versions System (CVS), while still being free software. Subversion attempts to fix most of the noticeable flaws of the CVS and is very powerful, usable and flexible. New to Subversion is the way how it internally treats directories and documents: directories (which are also documents) only contain links to specific revisions of other documents. Thus, when a source file is copied, Subversion will not copy all the data, but rather create a new directory entry referencing the same object. Hence, it is possible to copy the entire development branch at very low cost.

Access to the Subversion code repository is provided through secure SSL connections using the GForge project hosting server. Thus, this repository allows the geographically dispersed group of consortium members to collaborate and share their source code. It tracks changes to source code and allows a versioning of source code files. Additionally, Subversion remembers every change ever made to the files and directories so that earlier versions can be recovered in case that something was accidentally deleted.

9.2.2 General Setup of Subversion for the IMPReSS Project

The Subversion repository created for the IMPReSS project comprises three subdirectories:

- branches, i.e. for short time branches
- tags, i.e. a full copy of the source code for releases
- trunk, i.e. for the working source code as the current development version

Trunk is going to be the development branch where we will be storing our most recent version of the code. The different releases of the code will be stored in the Branches directory. For example, if a developer wants to create the Release-Version-1.1.0 of the code then she creates the directory in the Branches section and copies the most recent code stored in the trunk section to the release directory.

The Release branch will be used to correct the errors in the code during the alpha/beta testing phase. All the commits at this stage will take place in the Branch directory. When the developers are done with this they will copy this code to the tag directory and will name it as release version 1.1.0. Tag is defined to be nothing more than a copy of the repository at any given moment. Although there is not much difference between the branch and tag, the tag code is never committed and so it will act as the final release of any given version (i.e. public release).

The common structure of the software directory of the SVN trunk currently comprises the following subdirectories:

- bin containing all dlls
- src containing all Java and.Net source files

- lib containing all library files
- include containing all include files
- project containing all Visual Studio Project Files
- docs containing all documentation corresponding to this release.

Version control is generally independent of the operating system and programming language used. The standard management program "svn" for command line usage is available for different platforms, including Mac, Linux and Windows. Subversion integrates well with programming languages that have C-like syntax and source file schemas. It is important to note that a revision repository should contain all files necessary to build the software system, but should not contain any files that can be derived from other files in an automated fashion. Such redundancy only pollutes the repository and is considered bad practice (Richardson and Gwaltney, 2005).

There exist differences in the integration of SVN support into the integrated development environment regarding the programming languages Java and.Net, which are addressed by the following paragraphs.

Subversion and Java

Besides the management of Java sources using the command line tool (which, of course, can be used for revision control of almost all kinds of software), integration of Subversion into Java IDEs is quite advanced. Most IDEs natively support Subversion. This is partly due to open source Subversion libraries, but also due to the fact that Subversion is following the success of CVS, especially with regard to open source software.

The integrated development environment IntelliJ IDEA supports Subversion out of the box. The integrated development environment Eclipse does not directly support Subversion, but a Subversion library is available. Eclipse plugins are easily installed with its plugins manager.

Subversion and .Net

Microsoft has its own configuration management tool Visual SourceSafe in its portfolio, and hence, support for other revision control software cannot be expected natively. However, there exist other free and commercial plugins such as AnkhSVN and VisualSVN, which are available for Visual Studio and thus can be used for Subversion configuration management of the.Net developments in IMPReSS.

In addition, the command line tool TortoiseSVN allows for a configuration management of any directory in the file system. Furthermore, TortoiseSVN hooks into the Windows Explorer and allows for a more graphic-based checking in, updating and committing of changes.

9.2.3 Integration Plan for the IMPReSS software

The integration process of the IMPReSS software components is driven by principles of agile software development and will apply continuous integration to a large extent. The resulting overall IMPReSS system will be integrated incrementally. Continuous Integration increases the awareness of problems by involved developers by reporting back problems in a very short time frame after these problems occurred. The error resolving takes less time and efforts compared to failure detection in later stages of the development.

But because of the distributed fashion of the development we foresee a need for actual physical integration meetings as well. So for each development iteration there will be one or more integration meetings.

Considering the Integration Plan a Living Document

The plan is based on the current software architecture description of IMPReSS. The software architecture is the result of an ongoing and iterative process of user involvement, requirement analysis, and user evaluation. This process is not yet finished and therefore, the integration plan bases on a snapshot of the current interplay of the IMPReSS components.

The integration plan must evolve along with the software architecture description and potential changes of the architecture (e.g. towards a more loosely coupled approach) that might have impact on the integration process need to be considered in an updated integration plan. However, the approach and overall plan will remain the same in principle.

Continuous Integration Process

A well-designed continuous integration process encapsulates other sub-practices such as continuous builds, continuous unit testing, continuous deployment, continuous functional testing, continuous notifications, and continuous reporting. All of these are designed to shorten the time between the problem discovery and the problem solution. The IMPReSS integration process forms a continuous cycle as shown below. The following paragraphs describe the steps of this cycle in more detail.

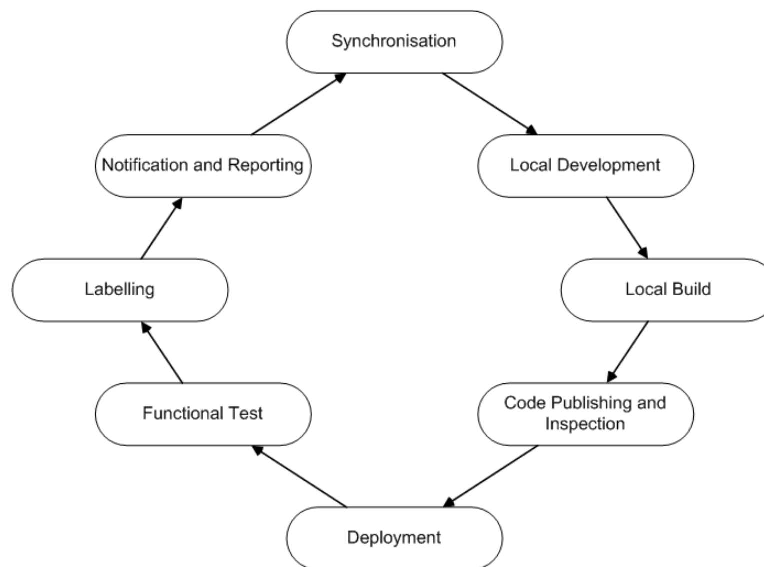

Figure 6: The Continuous Integration Process

Synchronization

The first step of the continuous integration process constitutes in the synchronisation of the files on the local developer machine. The developer is checking out a working copy of the source code repository from the mainline (or trunk) of the revision control system. Thus, the developer obtains a copy of the currently integrated source onto her local development machine. This local copy can now be modified so that it implements the feature required by the overall software system.

Local Development

During this step, the developers alternate between adding new tests and functionality and refactoring the code to improve its internal consistency and clarity. They realise and implement the technical specification of their manager as a software component. The refactoring of their source code often means modifying without changing its external behaviour; the developers are cleaning up their source code.

Local Build

The developer who works on her software component and who changes the system makes every effort to ensure that her feature does not introduce a bug into the overall system: Once she is done (and usually at various points while she is working) she carries out an automated build on her development machine. This takes the source code she is working with, compiles and links it into an executable program, and runs automated or manual tests. These tests may comprise functional, security, load and performance tests. Only if the entire code builds correctly and runs the tests without errors the overall build is considered to be good. The unit can be functionally accepted.

Code Publishing and Inspection

After the local build succeeded the developer can publish her local changes of the source code. This is usually done through committing the local sources to the central software revision system. From there the source code can be inspected by executing static or dynamic analysis checkers that address the coverage of the source code, possible duplication of code and dependencies among the respective software modules.

Deployment

Whenever a new revision of a software unit is made available and committed to the repository, the automated continuous integration system is notified. The continuous integration tool waits for a short time and then performs a clean check-out of the most recent version of the software system from the repository. Thus, the overall software system is deployed on a dedicated integration machine. Additionally, remote deployments into target environments can be done, e.g. from an integration machine to staging environment (for web containers and database servers).

Functional Test

After the check-out is complete the build process is triggered, which starts with the compilation and packaging or linking of the middleware. Any syntax errors or interface incompatibilities introduced with a new revision will usually fail the continuous integration cycle at this stage and the system will notify the submitter of the entire team of the failure. Once the problems are fixed the continuous integration server will recompile.

Thereafter, a regression test using the unit tests (as mentioned earlier in this document) and specific integration tests is performed based on the mainline code. Although unit tests do not guarantee that the software does not contain errors, a lot of errors are ruled out. Especially if unit testing is used to prevent the re-introduction of already known bugs. It is therefore important that build files for an automatic build system be maintained.

Labelling

After all functional tests successfully pass and the last overall build succeeds, the middleware is entirely integrated and changes of the code are accepted. The reason for this is that there is always a chance of missing something on the local machine or that the repository was not properly updated. Only when the committed changes build successfully on the integration machine the job of the developer is done. Then, the continuous integration server will label the source code with an automatically generated label.

Notification and Reporting

In case the build or unit testing fails, the development team will automatically be notified by the continuous integration server. This notification will be done via email. This ensures that errors are reported back before the programmer forgets how to undo her errors. It also serves to prevent sloppy code check-ins as mistakes will immediately become visible to every person of the development team. Hence, the developer risks getting a bad reputation and the development team can be quite sure that every version of the software in the repository compiles without errors and that it is unit tested. Additionally, the continuous integration server will generate a report of the integration cycle in regular intervals.

10. Conclusions

This deliverable has presented the technical testing approach and plan as well as integration strategies for the IMPReSS project. The plan is based on the current software architecture description of the IMPReSS platform and the plan must evolve along with the software architecture description. The approach and overall plan will however remain the same. The overall integration strategy has been discussed, and emphasis has been put on the continuous integration model used, required good measures for software code management.

Both testing and integration strategies are much influenced by agile software development processes (e.g. XP, Extreme Programming) and IMPReSS will utilize automatic test and integration frameworks. It is however envisaged that due to the loosely coupled nature of the final IMPReSS system where applications work with a dynamically changing set of devices and network resources, also incremental integration will be applied.

11. References

(Beck, 2000) Beck, K. (2000): Extreme Programming Explained: Embrace Change. Addison-Wesley

(Beizer, 1990) Beizer, B. (1990): Software Testing Techniques. International Thomson Press, second ed.

(Boehm, 1989) Boehm, B. (1989): Software Risk Management. IEEE Computer Society Press, CA

(Bourque and Dupuis, 2004) Bourque, P. and Dupuis, R. (2004): Guide to the Software Engineering Body of Knowledge, IEEE Press

(Bruegge and Dutoit, 2000) Bruegge, B.; Dutoit, A.H. (2000): Object-Oriented Software Engineering: Conquering Complex and Changing Systems. Prentice Hall.

(IEEE Std 610.12, 1990) IEEE Std 610.12 (1990): IEEE Standard Glossary of Software Engineering Terminology. IEEE Standards Association.

(ISO 9126-1 2001). Software Engineering. Product Quality. Part 1: Quality model. Technical Report ISO/IEC 9126-1:2001(E), ISO/IEC.

(JUnit, 2011) JUnit. http://www.junit.org. Accessed 2011-04-20

(Kitchenham and Pfleeger, 1996) Kitchenham, B. and Pfleeger, S.L. (1996): Software Quality: The Elusive Target. IEEE Software 13(1): 12-21.

(McCall et.al, 1977) McCall, JA, Richards, PK, and Walters, GF (1977): Factors in Software Quality. General Electric, Co., Rep. GE-TIS-77 CIS 02.

(McGregor and Sykes, 2001) McGregor, J.D., and Sykes, D.A. (2001): A Practical Guide to Testing Object-Oriented Software, Addison-Wesley.

(NUnit, 2011) NUnit. http://www.nunit.org. Accessed 2011-04-20

(Ommering, 2003) Ommering, R. v. (2003): Configuration Management in Component Based Product Populations. Westfechtel, B. (ed.): Software Configuration Management, Springer.

(Richardson and Gwaltney, 2005) Richardson, J.; Gwaltney, W.A. (2005): Ship It!: A Practical Guide to Successful Software Projects. The Pragmatic Bookshelf.

(Spinellis, 2003) Spinellis, D. (2003): Code Reading – The Open Source Perspective. p. 528, Addison Wesley, ISBN: 0201799405.

(Weischedel and Versteegen, 2002) Weischedel, G.; Versteegen, G. (2002): Konfigurationsmanagement. Springer.

(Westfechtel and Conradi, 2003) Westfechtel, B.; Conradi, R. (2003): Software Architecture and Software Configuration Management. Westfechtel, B.; Hoek, A. v. d. (eds.); Software Configuration Management. Springer

(WS-I, 2011) The Web Service Interoperability organization. http://www.ws-i.org. Accessed 2011-04-20