ALMANAC

RELIABLE SMART SECURE
INTERNET OF THINGS FOR SMART CITIES

(FP7 609081)

D7.1 Test and Integration Plan

Date 30th May 2014 – Version 1.0

Published by the Almanac Consortium

Dissemination Level: Public

Project co-funded by the European Commission within the 7th Framework Programme
Objective ICT-2013.1.4: A reliable, smart and secure Internet of Things for Smart Cities

Document control page

Document file:	D7.1TestAndIntegrationPlan.docx
Document version:	1.0
Document owner:	Mark Vinkovits (FIT)
Work package:	WP7 – Platform Integration
Task:	T7.2 – Development and Integration Planning and Support
Deliverable type:	R
Document status:	☒ approved by the document owner for internal review ☒ approved for submission to the EC

Document history:

Version	Author(s)	Date	Summary of changes made
0.1	Mark Vinkovits (FIT)	2014-04-07	Initial structure and content
0.2	Mark Vinkovits (FIT)	2014-05-12	Input to ch. 4
0.3	Matts Ahlsen, Mathias Axling, Peeter Kool (CNET)	2014-05-21	System testing: event processing
0.4	Maria Teresa Delgado, Riccardo Tomasi (ISMB)	2014-05-21	System testing
0.5	Mark Vinkovits (FIT)	2015-05-26	Merging of contributions
0.6	Maria Teresa Delgado (ISMB), Mark Vinkovits (FIT)	2015-05-29	Addressing review comments
1.0	Mark Vinkovits (FIT)	2015-05-30	Final version submitted to the European Commission

Internal review history:

Reviewed by	Date	Summary of comments
Anders Skovbo Christensen	2014-05-28	Accepted with comments
Alexandre Alapetite	2014-08-29	Accepted with comments

Index:

1. Executive summary

This document describes the test and integration strategy for the software components developed in ALMANAC. It is based on the architecture and the development plan.

A test plan depends partly on the chosen development approach. In ALMANAC this approach is iterative and incremental, comprising several complete cycles of analysis, development, and validation in which components from different partners will be frequently integrated. To ensure the quality of the code tests are performed continuously throughout the project. These will include the testing of single components, integrated components and the integrated platform. Automated testing will be used whenever possible and the tests will be executed regularly.

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

1. Unit testing – Testing at the lowest level using a coverage tool. A unit is the smallest testable part of an application.
2. Integration testing – Testing of interfaces between components to ensure compatibility.
3. System testing – Testing of entire software systems
4. Validation – Evaluation at the end of development to ensure requirements fulfilment.

Validation will be performed based on user applications as developed in WP8 and is not further described in this document.

The ALMANAC software platform will be developed in different environments as well as on different platforms (Java and .NET), based on a service-oriented architecture. The communication is either done via OSGi (declarative services, blueprint) or via Web Services. For integration testing of OSGi bundles a combination of Maven, PAX Exam, and a Plug-in Development Environment (PDE) can be used. Software builds on the continuous integration server can be stored in a repository.

Traditional unit testing frameworks do not support web service tests. There are two different approaches which can be used instead in ALMANAC.

- Testing done on WSDL files between components during development and integration
- More formal testing of components using the Web Services Interoperability Organization (WS-I) tools

System testing on the integrated platform will be done using a reference server, including reference gateways. This system provides a stable test environment where reproducible tests can be made.

In line with the agile approach to software development, integration should be performed incrementally and continuously following an integration process. By using a Source Code Management System such as Git, it will be ensured that all developers have access to the latest versions of the code base. The use of a continuous integration server will ensure that the components are built after each change in the repository and that the automated tests are executed.

2. Introduction

2.1 Purpose, context and scope of this deliverable

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

Bourque and Dupuis state that 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, including developers, operators, or prosumers.

The ALMANAC work package 7 "Platform Integration", which this deliverable belongs to, is however concerned with technical testing only: user validation is thus out of scope of this document and will be done in work package 8 "Applications Definition, Development and Evaluation". For dealing with this and in the quest for finding an appropriate framework capable at providing the suitable criteria and justification we approach ALMANAC 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 ALMANAC project. In general, the integration serves for the composition and concatenation of different applications. In ALMANAC, parts of the platform which are developed in the technical work packages shall be integrated into one platform.

2.2 Background

The ALMANAC Smart City Platform (SCP) collects, aggregates, and analyses real-time or near real-time data from appliances, sensors and actuators, smart meters, etc. deployed to implement Smart City processes via an independent, pervasive data communication network. ALMANAC aims at achieving pervasiveness by defining a short range capillary radio network providing local Machine-to-Machine (M2M) connectivity to smart things and enabling their active involvement in Smart City processes. The SCP allows decision support and implements intelligent control of the devices through the capillary networks with a M2M management platforms, as well as management of local installations.

The technological work in connection with the development of the ALMANAC Smart City platform will be highly influenced by requirements generated in the City of Turin. Its path to become "Smart City" started 2 years ago, when the City Council took the decision to take part in the initiative of the European Commission "*Covenant of Mayors*" and – as one of the first Italian cities – engaged itself to elaborate an *Action Plan for Energy* in order to reduce its CO_2 emissions more than 20% by 2020.

Three specific applications (waste management, water supply and citizens' engagement) have been selected for proof-of-concept implementation and evaluation in the ALMANAC platform. These applications are deemed to be sufficiently representative for a large number of applications, as will be visible from the use case descriptions. Given the challenging objectives, we have aimed for a set of 1) cross application domain use cases that 2) consist of a large amount of heterogeneous devices and 3) generate large amounts of data.

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 ALMANAC includes both the user and the manufacturing view.

Given specific views on software quality, ALMANAC developers still need a comprehensive model for the early assessment and evaluation of software quality. There are many models of software product quality that define software quality attributes. Three commonly 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. The criteria and characteristics defined in each of these models are listed in Table 1. Note that the ISO Model subsumes a number of them under its characteristic of "maintainability".

Table 1 Software Quality Models

Criteria/Characteristics	McCall, 1977	Boehm, 1978	ISO/IEC 25010:2011
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	

Out of these models the ISO 25010 model (ISO 25010, 2011) is the most recent and comprehensive one and will therefore be followed in the quality assurance and validation process of the ALMANAC platform.

The standard distinguishes three gradually dependent dimensions of the software product quality:

1. Internal quality refers to the "internal", developer view on static aspects of the system, i.e. the architecture and implementation of the system.
2. External quality refers to the externally observed behaviour and functioning of the system when executed and in a simulated/controlled environment ("developers' environment").
3. Quality in use refers to user's perception of the system quality while achieving goals in a particular environment/context of use.

The internal and external quality is modelled in terms of 8 characteristics and numerous subcharacteristics as depicted in Figure 1.

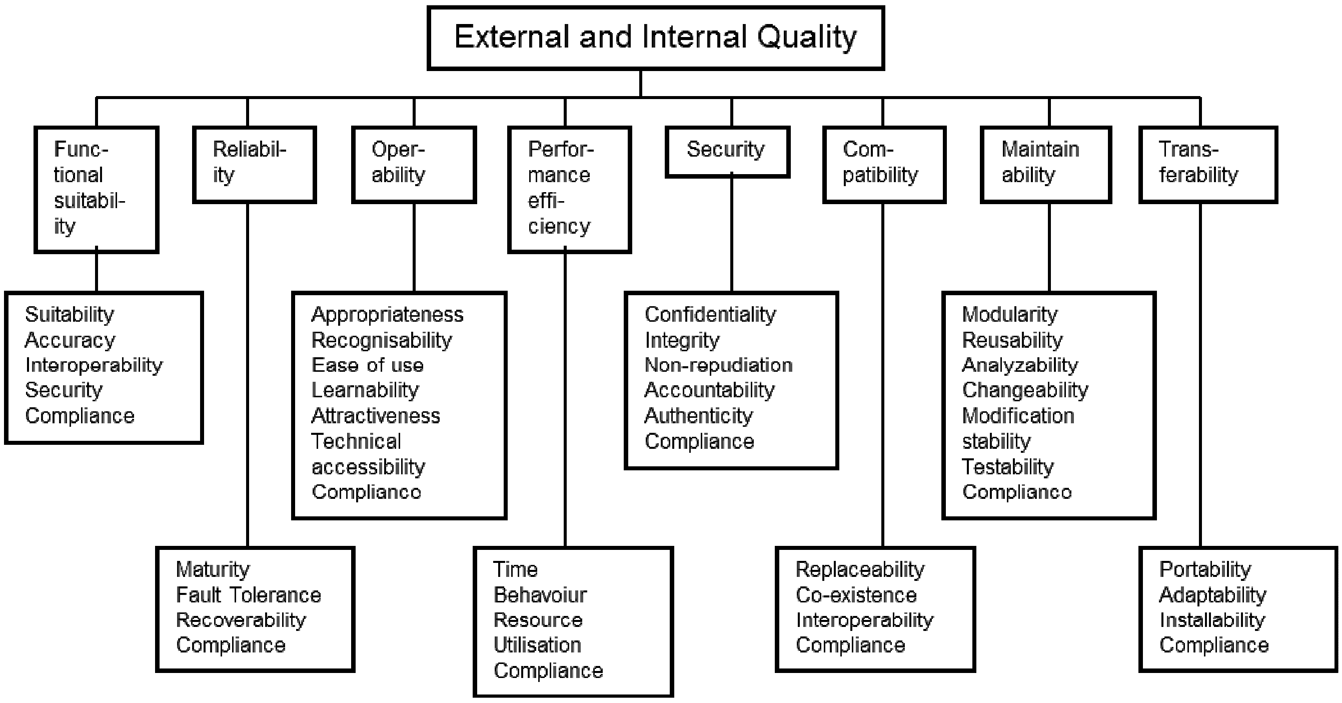

External and Internal Quality
Func-tional suitabil-ity
Reliabil-ity
Oper-ability
Perfor-mance effi-ciency
Security
Com-patibility
Maintain ability
Trans-ferability
Suitability Accuracy Interoperability Security Compliance
Appropriateness Recognisability Ease of use Learnability Attractiveness Technical accessibility Complianco
Confidentiality Integrity Non-repudiation Accountability Authenticity Compliance
Modularity Reusability Analyzability Changeability Modification stability Testability Complianco
Maturity Fault Tolerance Recoverability Compliance
Time Behavoiur Resource Utilisation Compliance
Replaceability Co-existence Interoperability Compliance
Portability Adaptability Installability Compliance

Figure 1 ISO/IEC 25010 Quality model for external and internal quality

Table 2 Quality characteristics of a software system

Quality characteristic	Description
Functionality	Functional coverage that meets stated and implied needs when the software is used under specified conditions.
Reliability	Maintenance of a specified level of performance when used under specified conditions.
Operability	Capability of the system to be comprehensible, operable and attractive to the user, when used under specified conditions.
Performance Efficiency	Appropriate performance, relative to the amount of resources used, under stated conditions.
Security	Proper protection of assets in their various aspects.
Compatibility	Aspect of the system to be compliant to standards, providing interoperable interfaces, and being replaceable with other standard compliant components.
Maintainability	Capability of the software product to be modified (corrected, improved or adapted to changes in environment and specifications).
Transferability	Capability of the system to be transferred from one environment (development) to another (production).

These characteristics provide a valuable, high-level model for quality assessment of a software system; the models will be covered throughout the various stages of the ALMANAC testing and quality assurance process prior to end-user deployments.

Complementary to considering the quality of system itself the quality in use refers to system's effective usage and perception by the end user. It refers to the overall capability of the system to

Table 3 Quality in use characteristics of a software system

Quality characteristic	Description
Effectiveness	System's capability to enable users to accurately and completely achieve specified goals in a particular context of use.
Productivity	System's capability to enable users to expend appropriate amounts of resources (time, effort, money etc.) in relation to the effectiveness achieved in a specified context of use.
Safety	System's capability to limit risk of harm to people, business, environment etc. to an acceptable level.
Satisfaction	System's capability to satisfy users in a specified context of use.
Usability	The extent to which a product can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use

enable specified users to achieve goals with regards to its quality characteristics: effectiveness, productivity, safety and satisfaction.

The quality in use conveys user's perspective on the system quality. It is measured by results of its concrete usage in context, rather than by properties of the software itself, and its evaluation will therefore involve participation of developers working with the ALMANAC platform.

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 ALMANAC 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 ALMANAC 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 four different levels (Beizer, 1990):

1. Unit testing – Testing at the lowest level using a coverage tool. A unit is the smallest testable part of an application.
2. Integration testing – Testing of interfaces between components to ensure that they are compatible.
3. System testing – Testing of entire software systems
4. Validation – Evaluation at the end of development to ensure requirements fulfilment.

Validation will be performed based on user applications as developed in WP8 and is not further described in this document.

In the following chapters the testing approach for ALMANAC is described. Unit testing, integration testing and system testing is done internally by the developers. Validation is done with the help of the users on Fur.

Software should first be unit tested, integration tests should follow and finally system testing should be performed. As ALMANAC has an iterative approach, there might be interleaving or backtracking of these testing activities.

3.3.1 Unit Testing

A "unit" in ALMANAC 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, 2013). In the case of .NET this is NUnit (NUnit, 2013).

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

- Code checked into the Git repository must not break the build process
- Prior to committing new versions of a unit to the Git repository there must be reasonable automated, functional unit tests. There will be no required test coverage criteria, but we advise a test-coverage of 50 percent or higher.

To enhance quality of the software it is recommended to create a new unit test for each detected bug if there was not one already. After fixing the bug the commit statement has to contain the bug number so that an automated tracking of bug fixes can be performed.

3.3.2 Integration Testing

Integration testing takes place whenever multiple units need to work together. 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 ALMANAC platform will be developed in different environments as well as on different platforms (Java and .NET), based on a service-oriented architecture. The communication is either done via OSGi (declarative services, blueprint) or via Web Services.

Unit testing frameworks do not support properly web service or OSGi-oriented tests. Concerning web-service tests, there are two different approaches which can be used instead in ALMANAC.

- Testing done on WSDL files between components during development and integration. As the 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).
- Concerning OSGI test, specific testing frameworks can be used to dynamically orchestrate the composition of different test configurations and run specific tests.

Table 4 Software test plan

Level of testing	Activities	Frequency of testing	Responsible
Unit	• Select test cases • Write automated test cases	• Test creation while developing component • Automated tests run continuously when component is built on the Continuous Integration Server	Component/ Unit developer
Integration	• Select test cases • Manage unit dependencies (Maven) • Write automated test cases • Prepare non-automated test cases	• Automated tests run continuously when components are built on the Continuous Integration Server • Manual tests, at least each iteration • WS-I, at least each iteration • WSDL continuously (whenever changes occur)	Component Developer whose component is calling another component
System	• Select test cases according to the requirements • Prepare test beds and test data (if necessary) • Run test beds	• Automated tests run continuously when components are built on the Continuous Integration Server • Manual test before installing a new prototype	Prototype Deliverable responsible

3.3.3 System Testing

The main purpose of System testing will be to verify practically that the overall system (i.e. the overall ALMANAC SCP) includes all the technical capabilities needed to support the foreseen scenarios. For this reason, system test can be considered as a step to assess the minimum required ALMANAC technical capabilities, and this is needed before ALMANAC Evaluation activities can take place.

Due to this link, on a system level, the indicators to be used as a starting point for system testing will be derived from existing requirements and innovation features, in form of use cases/high-level functional requirement and non-functional/quality requirements.

System testing will be performed after integration testing, and will be performed in a horizontal manner assuming that integration will take care of the vertical interoperability across the different levels of the system. These tests will be automated whenever possible.

The system testing process will be driven according to the approach described in Table 5.

Table 5 System testing plan

Step #	***Description***	***Responsible***
1.	Construction of the initial test list from the available "application oriented" use cases and "technical" uses cases derived by innovations	T7.3 Leader, supported by WP2 Leader, Technical Manager and Innovation Manager
2.	Definition of the indicators of success for each system test case (e.g. green light/ red light or quantitative indicators in case of variable parameters e.g. supported workload, number of devices, bandwidth, etc.)	T7.3 Leader, dividing the workload among all participating partners
3.	(optional) definition of the system configuration needed for each system test case (only needed in cases where the full system configuration is not possible for some reason)	T7.3 Leader, dividing the workload among all participating partners

4.	Implementation and execution of the system test (in automatic mode whereas possible – so tests can be repeated on demand)	T7.3 Leader, dividing the workload among all participating partners

3.4 Software Test Plan

The test plan regards unit, integration and system testing. Activities, frequency, and responsibilities are as summarized in Table 4. The responsible for system testing is not responsible to write all tests and execute them. The responsible should ensure though that system tests are being executed.

3.4.1 Testing the ALMANAC Platform

Generally, the optimal configuration of software testing strategies in complex, modular projects heavily depend on the specific selected eco-system i.e. the set of languages, tools, frameworks and policies used for the development, the management and the deployment of the solutions.

The following picture provides a graphical overview of the eco-system adopted for ALMANAC, together with a high-level matching with the different testing strategies described in the following.

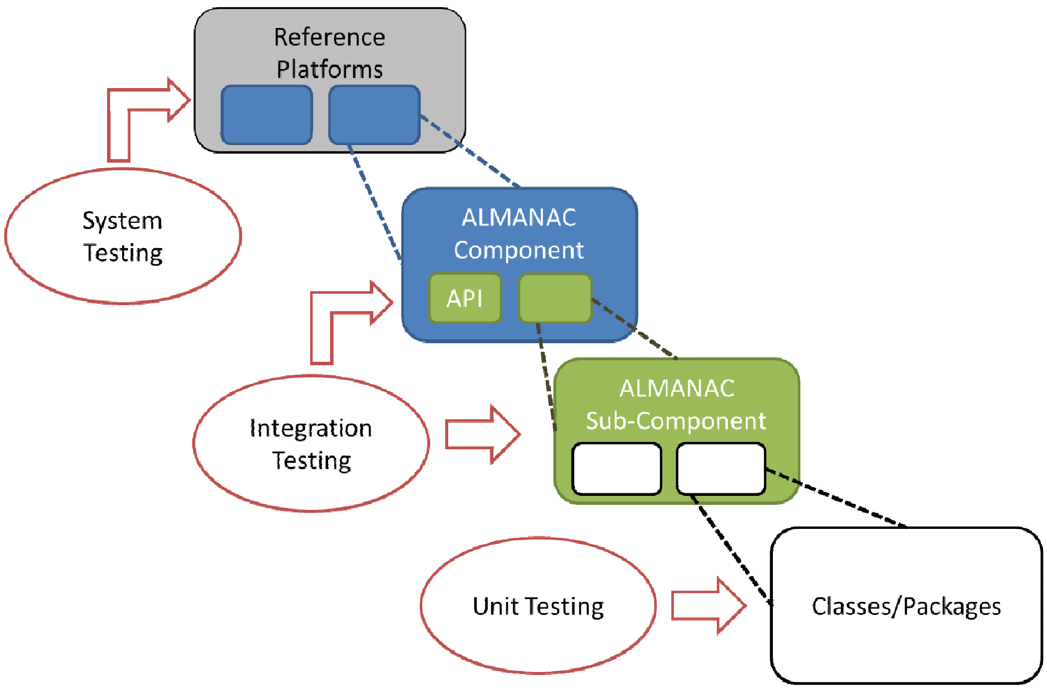


Figure 2 The ALMANAC SW eco-system

The reference platform, consisting of gateways and servers, typically hosts one or more ALMANAC components. ALMANAC components are defined in the D3.1.x series of deliverables and represent the highest level functional unit of the ALMANAC system (e.g. Virtualization Layer, Data Management, etc.). In the ALMANAC eco-system components are realized by means of Maven multi-module systems[1] which host a set of sub-components. Each component has a special API module that centralized all its main APIs and interfaces of sub-components, including the ones which are realized in non-Java technologies.

A component is typically composed by a set of sub-components which are implemented as OSGi bundles. ALMANAC OSGi bundles are developed in pom-first fashion. A component template for

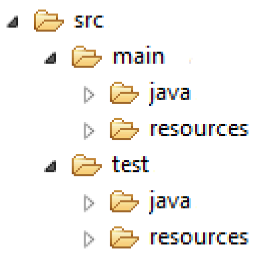
[1] http://maven.apache.org/guides/mini/guide-multiple-modules.html

developing bundles according to this work-flow has been developed by the ALMANAC project and is maintained on the software repository.

Within the ALMANAC work-flow, integration testing thus covers two levels: on the edge interfaces between components (e.g. between SCRAL and Virtualization Layer) and between different sub-components of each component (e.g between the Policy Enforcement Point and Policy Decision Point inside the Policy Framework). Finally, specific implementations of sub-components (i.e. Java classes organized as packages) are tested by Unit testing strategies. Unit testing is thus managed inside each sub-component, i.e. a test unit will be defined for each of the main classes hosted by a sub-component (e.g. a SCRALSerialDeviceDriver class inside the SCRAL component).

3.4.2 Unit Testing

Unit tests should be included in the bundle they belong to. In Java, the folder structure should comply with the structure commonly used in Maven:



If a class eu.almanac.virtualization.Discovery is to be tested, the class name of the test should be eu.almanac.virtualization.DiscoveryTest.

The test class itself and each test method should contain JavaDoc comments about what is tested with which parameters. If it is possible, there should be as well tests which provoke exceptions/ errors.

3.4.3 Integration Testing

Integration testing of ALMANAC components heavily leverages the previously mentioned pom-first work-flow based on Maven and Eclipse PDE (e.g. for automatically checking-out, building and composing components in different configurations to test different aspects).

In ALMANAC Integration testing covers two different aspects: integration between components and internal integration between sub-components.

While integration testing could be performed manually, in a highly modular system such as ALMANAC automated integration testing is expected to be more productive. For such reason, a set of integration tests for both component-to-component and inside-component tests will be developed using frameworks such as PAX-Exam[2].

It is important to mention that the approach selected for managing dependencies and control dependency injection impacts heavily on how these test can be developed. The integration tests will leverage OSGi declarative services (i.e. in terms of components and/or blueprint configurations) as main interface for testing, including checks on WSDL and WS-I whenever appropriate.

Dependencies will be managed leveraging the configuration/build support provided by Maven + Pax-Exam.

3.4.4 System Testing

ALMANAC system level testing will analyse and asses communication between sensors and actuators and gateways, communication among capillary networks, and the communication between gateways and the ALMANAC platform itself.

[2] https://ops4j1.jira.com/wiki/display/PAXEXAM3/Pax+Exam

Given that a consistent evaluation activity is foreseen by the ALMANAC project, testing the system in conditions similar to the realistic operating scenario is essential. Within the ALMANAC project, system testing will be performed on specific configurations of components, hosted by actual platforms (i.e. servers, gateways, sensors) aiming to emulate both waste and water transport networks from the Turin city, running configurations similar to the actual deployments, in order to evaluate long-term stability and behaviour under stress conditions and a critical traffic load.

The figure below outlines the infrastructure to be established for system testing.

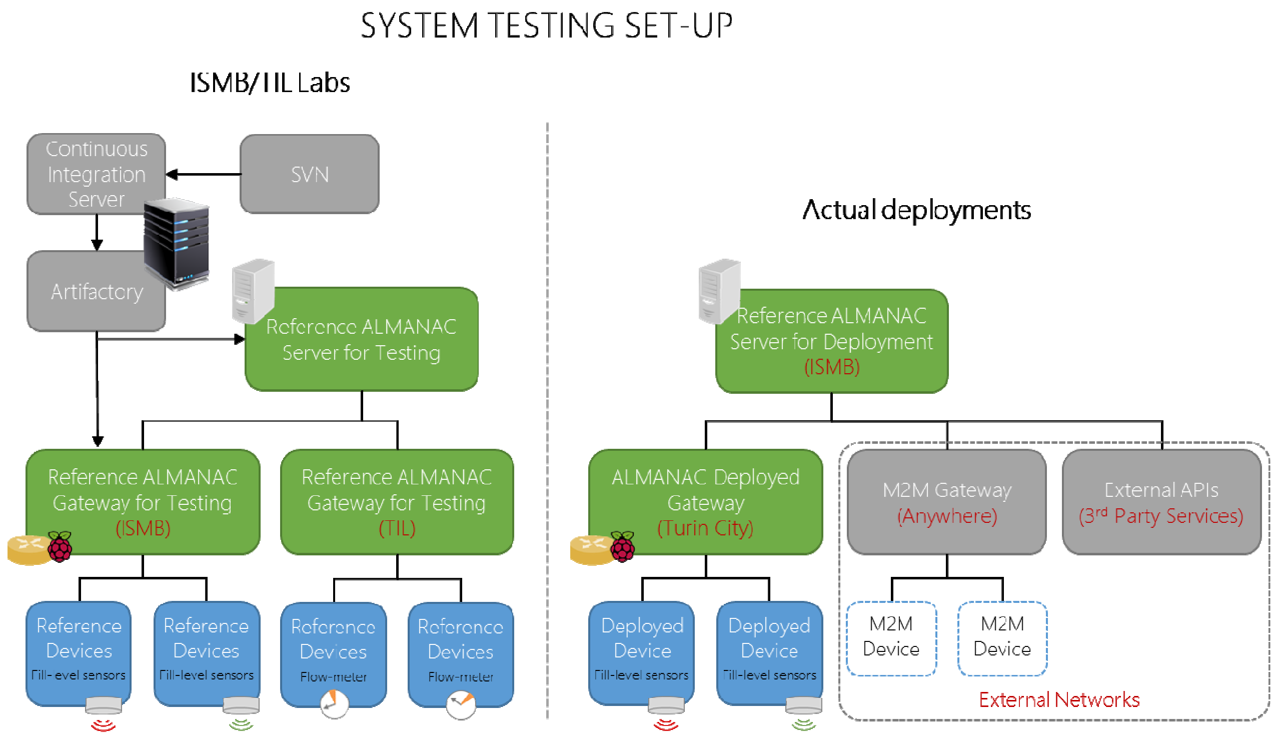

SYSTEM TESTING SET-UP
ISMB/TIL Labs
Continuous Integration Server
SVN
Artifactory
Reference ALMANAC Server for Testing
Reference ALMANAC Gateway for Testing (ISMB)
Reference ALMANAC Gateway for Testing (TIL)
Reference Devices Fill-level sensors
Reference Devices Fill-level sensors
Reference Devices Flow-meter
Reference Devices Flow-meter
Actual deployments
Reference ALMANAC Server for Deployment (ISMB)
ALMANAC Deployed Gateway (Turin City)
M2M Gateway (Anywhere)
External APIs (3rd Party Services)
Deployed Device Fill-level sensors
Deployed Device Fill-level sensors
M2M Device
M2M Device
External Networks

Figure 3 System testing set-up overview

ALMANAC project does not foresee the deployment of dedicated networks to support Smart City applications. Instead, ALMANAC will maximise the re-use of pre-existing communication network infrastructure to develop an adaptable and scalable smart city platform.

ALMANAC is accessible from any system directly joining the middleware domain (i.e. as a LinkSmart entity) or by any type of internet-oriented application (e.g. mobile applications, web applications, etc.) through Open Cloud-based APIs. Figure 3 provides an overview of the system testing approach foreseen for the ALMANAC platform. A reference server and a set of reference gateways will be installed both in ISMB laboratories and Telecom Italia Labs.

During the first phase of system testing (held in ISMB and TIL premises), a small-scale set of reference devices (e.g. fill-level sensors and water flow meters) will be used to feed data into the reference Gateways, mostly to assess the stability of the developed platform and detect in advance possible performance bottlenecks and stability issues. In a second phase, the installation of a wireless sensor network in a determined area of the city of Turin will be exploited to test more specific aspects related to the waste management and resemble a more realistic large-scale deployment. Both locations will be used to collect data useful to assess performance and scalability of the ALMANAC platform.

The deployment of ALMANAC components and 3rd party components will be automated in order to keep them synchronized with the component repository. For this purpose, we will leverage on the Artifact repository metadata to continuously identify version updates, resolve and deploy current component versions and dependencies (e.g. newly included components). Up-to-date versions of the components will be verified periodically, while running in normal operating conditions on the reference servers and gateways, to assess stability and performance of the overall system.

Testing Scalability aspects

Since ALMANAC is dealing with a large quantity of heterogeneous devices generating data, a key aspect to be tested is how the different segment of the platform scale under normal and “stress” conditions.

In order to test scalability on the field, ALMANAC components will be instrumented with the ability to generate synthetic up-link and down-link traffic in controlled/repeatable conditions, using software probes to verify the stability and performance of the platform under test. Such testing features will also be available on devices and systems deployed on the field.

In order to start scalability testing before the actual deployment, hybrid simulations will also be leveraged. A physical test-bed, as the one shown previously in Figure 3 will be deployed on the premises of ISMB and TIL, capturing detailed network transactions occurring during the operation of real communication devices under “stress-test” conditions.

To complement the pre-deployment evaluation with an assessment of network conditions on scales which cannot be physically emulated (i.e. the full city scale), ALMANAC will leverage network-scale simulations with increasing size and complexity, feeding data in the real platform.

3.4.5 Event Processing Testing

The ALMANAC platform and its applications will to a large extent be dependent on reliable and scalable event processing. The event management functionality will be based on underlying generic components of the LinkSmart middleware in combination with third party tools, adapted to the Smart City domain. The event model follows the well-established Publish-Subscribe pattern as implemented by the LinkSmart Event Manager.

In the initial ALMANAC architecture[3], the data from IoT Sensors or IoT Devices may be requested by a large number of applications and Semantic IoT Resources. The data also has to be transferred to long-term storage. To minimize the number of connections, and make the load on the IoT Device smaller in the case of a huge number of subscribers, the LinkSmart Event Manager will be extended with the necessary functionality. This enables multiple Event Managers to co-operate and use load-balancing to service publishers and subscription partitioning for publishing.

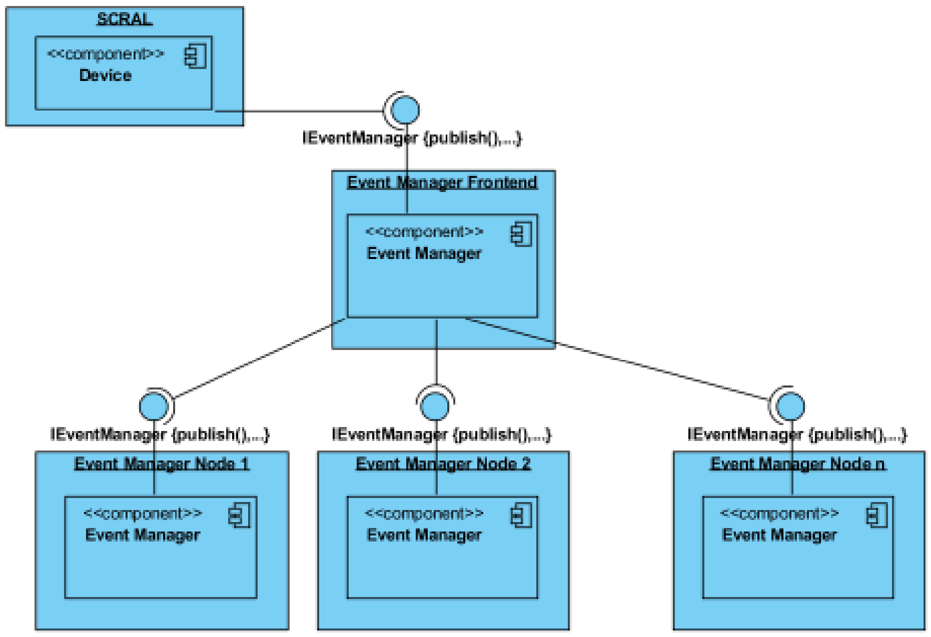


Figure 4 Event Manager nodes distribution

[3] Deliverable D3.1

Event Scalability Issues

A number of situations may affect the scalability of the event processing network, e.g.,

- A large number of producers are publishing events in parallel
- High frequency of events from one single event producer
- High number of event consumers.
    - If there is a high number of event consumers, each event may match several subscriptions and will have to be sent to many event consumers.
- High number of unresponsive event consumers.
    - If there is a high number of event consumers, and several of them do not respond, Event Managers will wait for a timeout for each of these.
- Subscriber loss of events due to processing overload
- Lost Subscriptions due broken event rules
- Size of event content (payload)
- Storage overload due to size of event contents
- Subscribers consume events slower than they are produced
- Event package lost in transmission

These situations will be further refined and extended based on the application requirements on functionality as well as non-functional requirements on the ALMANAC platform. A number of Quality of Service attributes will be defined, with threshold levels under different conditions, e.g., event-loss given frequency and number of producers and consumers. Test cases are defined based on the QoS levels. The same tests will have to be performed with the components running on the same local network and with the components running on different Network manager nodes.

Existing third party event processing environments and tools will also be used in ALMANAC. Some of these provide test bench tools (e.g., the Esper Test Bench).

Event Trace and Debug Tool (ETDT)

The Event Trace and debug tool provides the capabilities of eavesdropping on all event communication at an event manager. This tool run side by side with the LinkSmart Event Manager and can be turned off/on completely independently, see Figure 5.

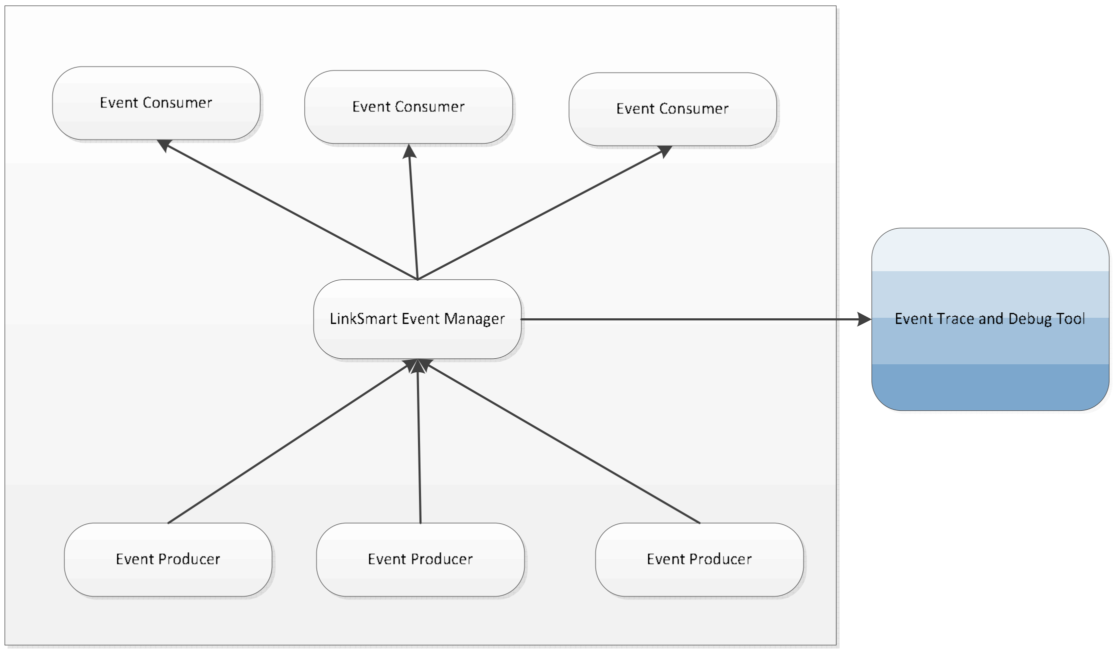


Figure 5 LinkSmart and the Event Trace and Debug Tool

The 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. 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 inner architecture of ETDT reflects this by trying to be as efficient as possible. The LinkSmart Event Manager can handle a large number of events per seconds, and therefore the ETDT must also be fast in its processing of events in order not to unnecessarily load the Event Manager.

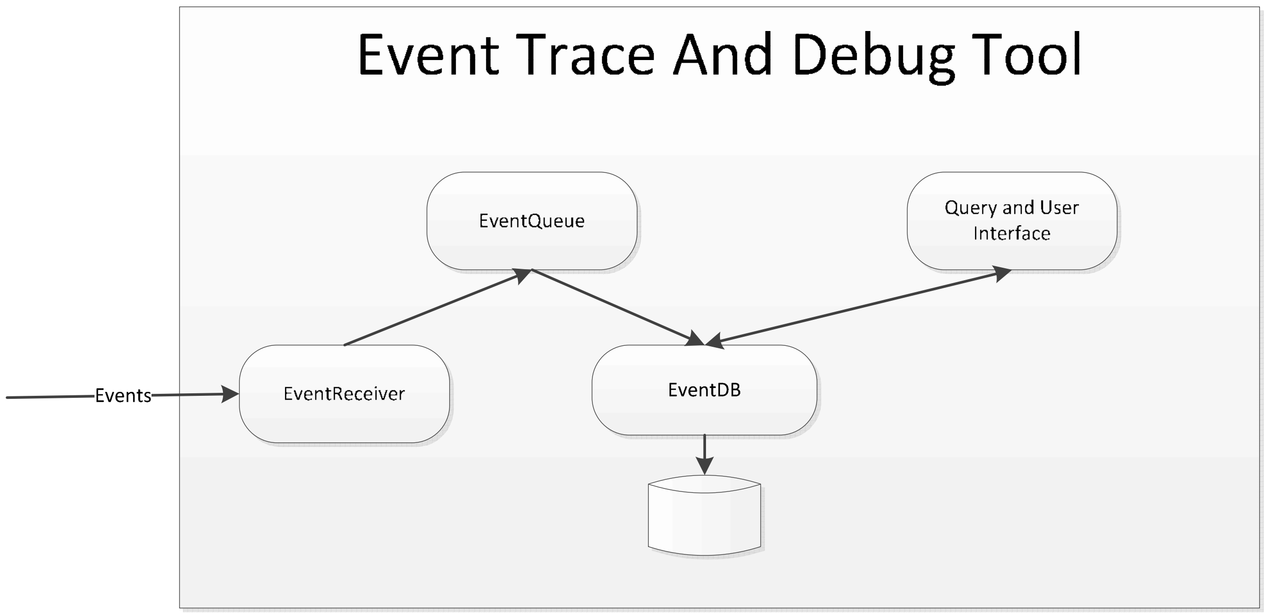


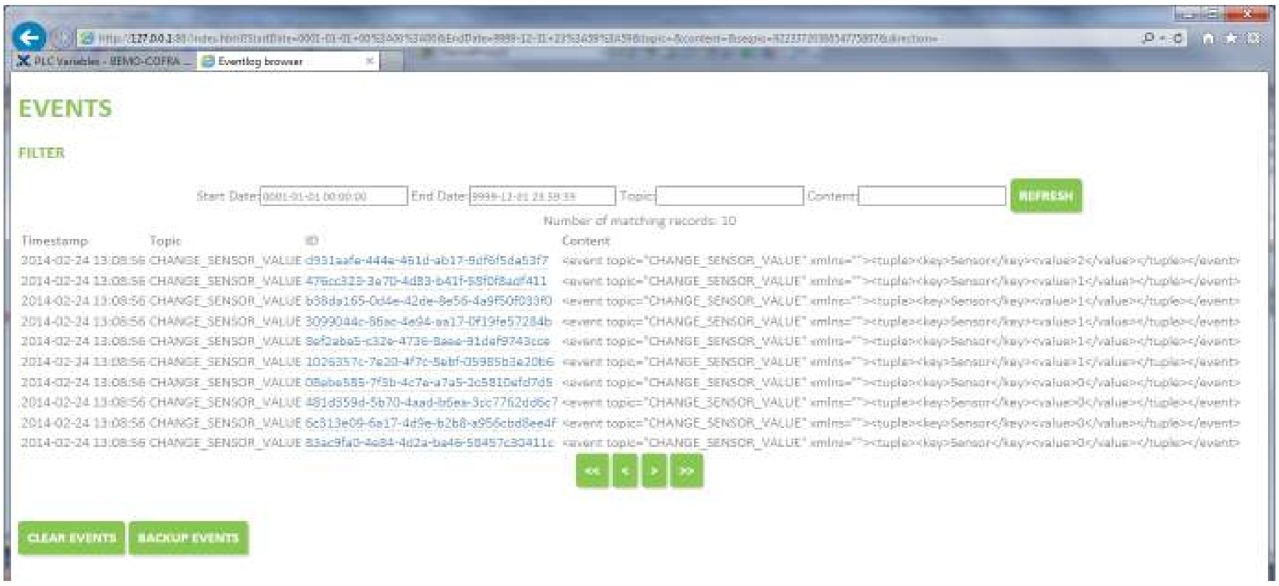
Figure 6 Inner architecture of the ETDT

The main components in the ETDT are:

- **Event Receiver:** Listens to the events. As soon as an event arrives it will queue in the EventQueue without any other processing.

- **EventQueue:** Acts as a buffer in between the persistent store and incoming events. It uses MSMQ as mechanism guarantying that no events are lost.
- **EventDB:** Manages the persistent store of events and provides interfaces to query the stored events database. The default implementation uses SQLite for storage, but this can be changed to in-memory databases etc.
- **Query and User Interface:** Is the actual consumer of the stored event data.

The ETDT provides a web based user interface that can be used with any ordinary web browser, see Figure 7.


EVENTS

FILTER

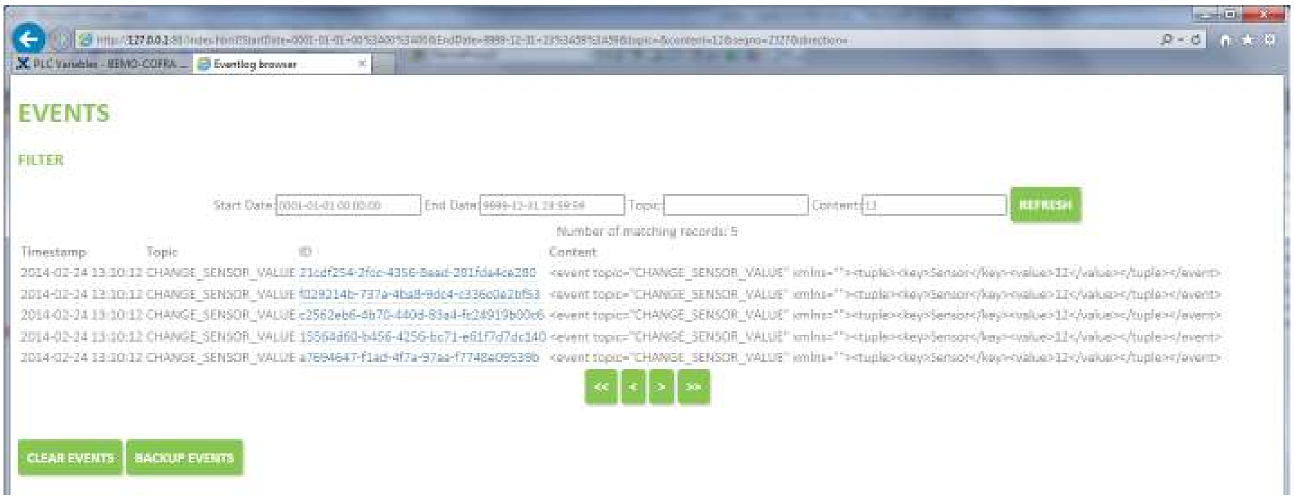
Start Date: 0001-01-01 00:00:00 End Date: 9999-12-31 23:59:59 Topic: Content:

Number of matching records: 10

Timestamp	Topic	ID	Content
2014-02-24 13:08:56	CHANGE_SENSOR_VALUE	d931aafe-444e-461d-ab17-5df6f5da53f7	<event topic="CHANGE_SENSOR_VALUE" xmlns=""><tuple><key>Sensor</key><value>2</value></tuple></event>
2014-02-24 13:08:56	CHANGE_SENSOR_VALUE	476cc323-3e70-4d83-b41f-58f0f8adf411	<event topic="CHANGE_SENSOR_VALUE" xmlns=""><tuple><key>Sensor</key><value>1</value></tuple></event>
2014-02-24 13:08:56	CHANGE_SENSOR_VALUE	b38da165-0d4e-42de-8e56-4a9f50f033f0	<event topic="CHANGE_SENSOR_VALUE" xmlns=""><tuple><key>Sensor</key><value>1</value></tuple></event>
2014-02-24 13:08:56	CHANGE_SENSOR_VALUE	3099044c-86ac-4e94-aa17-0f19fe57284b	<event topic="CHANGE_SENSOR_VALUE" xmlns=""><tuple><key>Sensor</key><value>1</value></tuple></event>
2014-02-24 13:08:56	CHANGE_SENSOR_VALUE	8ef2aba5-c32e-4736-8aea-81def9743cce	<event topic="CHANGE_SENSOR_VALUE" xmlns=""><tuple><key>Sensor</key><value>1</value></tuple></event>
2014-02-24 13:08:56	CHANGE_SENSOR_VALUE	1026357c-7e20-4f7c-8ebf-05985b3e20b6	<event topic="CHANGE_SENSOR_VALUE" xmlns=""><tuple><key>Sensor</key><value>1</value></tuple></event>
2014-02-24 13:08:56	CHANGE_SENSOR_VALUE	08abe585-7f3b-4c7e-a7a5-2c5810efd7d5	<event topic="CHANGE_SENSOR_VALUE" xmlns=""><tuple><key>Sensor</key><value>0</value></tuple></event>
2014-02-24 13:08:56	CHANGE_SENSOR_VALUE	481d359d-5b70-4aad-b6ea-3cc7762dd6c7	<event topic="CHANGE_SENSOR_VALUE" xmlns=""><tuple><key>Sensor</key><value>0</value></tuple></event>
2014-02-24 13:08:56	CHANGE_SENSOR_VALUE	6c313e09-6e17-4d9e-b2b8-a956cbd8ee4f	<event topic="CHANGE_SENSOR_VALUE" xmlns=""><tuple><key>Sensor</key><value>0</value></tuple></event>
2014-02-24 13:08:56	CHANGE_SENSOR_VALUE	83ac9fa0-4e84-4d2a-ba46-58457c30411c	<event topic="CHANGE_SENSOR_VALUE" xmlns=""><tuple><key>Sensor</key><value>0</value></tuple></event>

Figure 7: ETDT web based user interface

The tool presents the events always with the newest events first and always in the exact order they have arrived to the tool. The page also contains filters that one can apply for selecting events. An example of this is shown in Figure 8.


EVENTS

FILTER

Start Date: 0001-01-01 00:00:00 End Date: 9999-12-31 23:59:59 Topic: Content: 12

Number of matching records: 5

Timestamp	Topic	ID	Content
2014-02-24 13:10:12	CHANGE_SENSOR_VALUE	21cdf254-2fcc-4356-8aad-281fda4ca280	<event topic="CHANGE_SENSOR_VALUE" xmlns=""><tuple><key>Sensor</key><value>12</value></tuple></event>
2014-02-24 13:10:12	CHANGE_SENSOR_VALUE	f029214b-737a-4ba8-9dc4-c336c0e2bf53	<event topic="CHANGE_SENSOR_VALUE" xmlns=""><tuple><key>Sensor</key><value>12</value></tuple></event>
2014-02-24 13:10:12	CHANGE_SENSOR_VALUE	c2562eb6-4b70-440d-83a4-fc24919b00e6	<event topic="CHANGE_SENSOR_VALUE" xmlns=""><tuple><key>Sensor</key><value>12</value></tuple></event>
2014-02-24 13:10:12	CHANGE_SENSOR_VALUE	15564d60-b456-4256-bc71-e61f7d7dc140	<event topic="CHANGE_SENSOR_VALUE" xmlns=""><tuple><key>Sensor</key><value>12</value></tuple></event>
2014-02-24 13:10:12	CHANGE_SENSOR_VALUE	a7694647-f1ad-4f7a-97aa-f7748e09539b	<event topic="CHANGE_SENSOR_VALUE" xmlns=""><tuple><key>Sensor</key><value>12</value></tuple></event>

Figure 8: Example of a filter

4. Integration Plans

This section describes the plans for the integration of the software and hardware into the ALMANAC platform. Starting with an administrative perspective the overall integration and testing time line is presented in section 4.1. The reminder of the chapter refers to its realization. Section 4.2 describes the software source code management as part of the continuous integration cycle. Section 4.3 describes the build process and artefact management and finally section 4.4 summarizes the overall integration strategy for ALMANAC.

4.1 Integration and Testing Timeline

The ALMANAC project not only develops a distributed platform but is itself subject to a distributed development process making "integration" a central concern. In alignment with the Description of Work a time line has been scheduled that ensures a timely testing and system integration of the ALMANAC components. The platform prototypes will be committed by releases of the prototype deliverables ID8.3, ID8.5, ID8.7 in M12, M24, M36. An exemplary time line is shown in Figure 9.

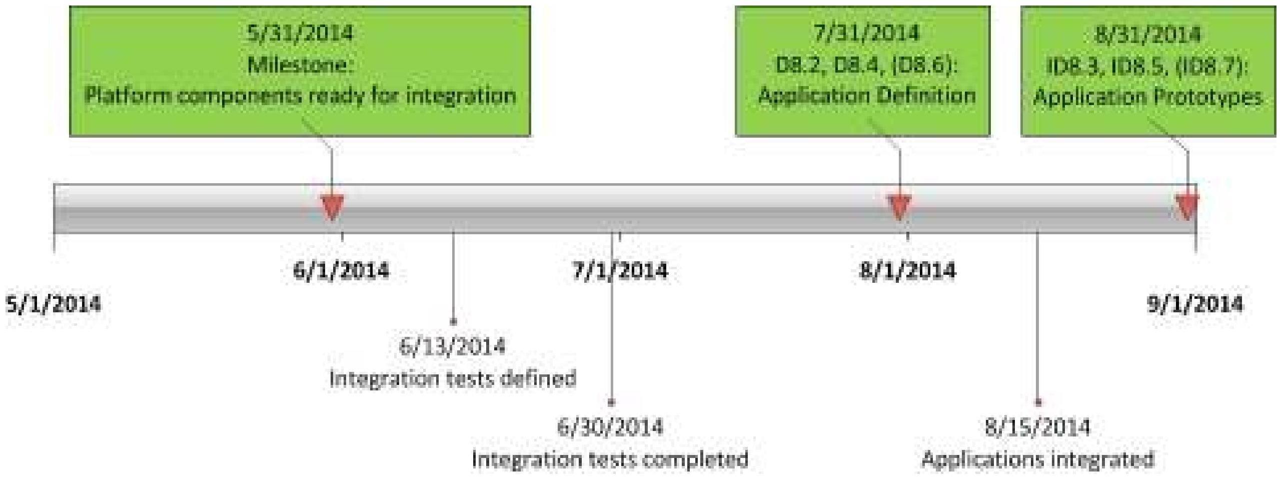


Figure 9 Exemplary Integration Timeline

4.2 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 ALMANAC we base the software configuration management on Git which addresses many of the problems listed in (Ommering, 2003) and (Bruegge and Dutoit, 2000) and which is described in the following subsection.

4.2.1 Source Code Management with Git

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 Git[4] repository where all artefacts of the development process will be stored and organised. Git, 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 Git as it is a leading edge technology for distributed revision control. It eases the creation of experimental branches, simplifies merge operations, and allows for a clearer, feature based development over distributed teams. Additionally, due to its wide acceptance, Git may foster the transition of the platform to smart city application developers, as it is a tool these developers are probably most familiar with.

Access to the Git code repository is provided through secure SSL connections. 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, Git remembers every change made to the files and directories so that earlier versions can be recovered in case that something was accidentally deleted.

Version control is generally independent of the operating system and programming language used. The standard management program "git" for command line usage is available for different platforms, including Mac, Linux and Windows. Git 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).

4.2.2 General Setup of the Git Repository for the ALMANAC Project

The Git repository created for the ALMANAC project is available under the URL https://subv-ext.fit.fraunhofer.de/scm/git/almanac and it comprises three parts:

- Branches, i.e. for experimental feature development;
- Tags, i.e. a full copy of the source code for milestones and prototypes
- Master, i.e. for the working source code as the current development version.

Master will be the integrated and stable development branch, where we will be storing a version of the platform that includes tested and integrated functionality. Branches can be used for each individual feature under development, before they get merged into the trunk. 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 master and a tag, the tag code is never committed to, and so it will act as a final release of any given version.

The common structure of the software directory of the Git master shortly comprises the following subdirectories:

[4] http://git-scm.com/

- Applications (ALEXANDRA): This folder is to contain the applications that are running on top of the ALMANAC platform.
- Devices (ISMB): This folder is to contain any code specific to a particular device. This includes arduino code, specific proxies, drivers, etc.
- Miscellaneous (FIT): This folder is to contain only data which is not going to be part of the final ALMANAC outcomes.
- Platform (FIT): This folder is to contain only components of the ALMANAC platform as a middleware and its cloud-based components.
- Templates (FIT): This folder is to contain templates for bundles, devices, etc.
- Tools (CNET): This folder is to contain developer tools, SDKs, DDKs, etc. If they cannot be reasonably detached from their respective component, they should be placed into the platform folder.

Names of components in the Git should be consistent to the latest stable architecture definition. A description of the sub-components is included in the README.txt file for each component.

All public interfaces which need to be shared across components or sub-components are included in API bundles of each component, namely: ScralAPI, VirtualizationAPI, DataManagementAPI, etc.. API bundles are coordinated by the component responsible.

Packages should start with eu.almanac followed by the bundle name. In cases where this is not possible, the other developers should be informed.

Unit tests will be included in the bundle they belong to. They will be placed in an extra folder called "test".

Integration tests which can test the compatibility of several components or bundles will be placed in separate bundles.

4.3 Build Process and Artefact Management

Out of the various tools facilitating the build process tasks (e.g. dependency management, compilation of source code, testing, packaging and provision of binary artefacts) we selected Maven for our Java modules, because of its maturity, popularity and rich software support (IDE integration, repository support).

Maven imposes some simple conventions on project organization. It defines a complete build life cycle[5], split into a sequence of dependent build phases (e.g. "compile", "test", "install"). For additional tasks, plugins can be integrated. The build life-cycle is configured by a central build configuration file (pom.xml) allowing for interception and proprietary handling of every phase. For example the "maven-scr-plugin"[6] allows for creation of standard OSGi component descriptors by evaluating annotations put on class members (in difference to creating the XML descriptors manually).

Maven not only eases the reference and resolution of dependencies at different build stages (compile- , test- or runtime), it also provides means of publishing and sharing code libraries (plain Java or OSGI bundles) via Maven repositories. When combined with a continuous integration server, a Maven repository can be used to persist and publish the results of the automated build process. These artefacts can be searched and referenced by their metadata (artefact group, name, version and packaging format). Popular repositories offer different metadata formats or "layouts" (such as Maven, P2 and OBR) in order to support different resolution strategies and environments.

The figure bellow depicts the build process adopted in ALMANAC. Developers use (internal and external) libraries (artefacts) to develop code. This is committed to the version control server

[5] http://maven.apache.org/ref/3.0.5/maven-core/lifecycles.html#default_Lifecycle
[6] http://felix.apache.org/documentation/subprojects/apache-felix-maven-scr-plugin/scr-annotations.html

triggering an automatic build on the continuous integration server. The resultant build artefacts are finally deployed into the repository and used in turn by a variety of clients.

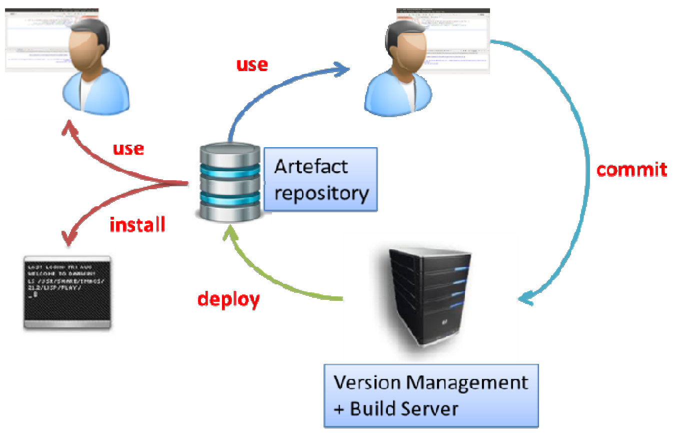


Figure 10 Build process and artefact management

4.4 Integration Strategies

The integration of ALMANAC software will be performed continuously and incrementally. The project'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 ALMANAC 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.

4.4.1 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 generally only possess detailed knowledge of their own components. Most software development approaches prefer an incremental integration as described in the next paragraph.

4.4.2 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. 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 (e.g. onc3e a day or each time something is checked into the repository), 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.

4.4.3 Integration Issues

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, three 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.
- Different hardware sensors will be deployed which constitute specific third party software. This places further demands on the integration process.

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.

4.4.4 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.

4.5 Integration process for the ALMANAC software

The integration process in ALMANAC will make use of continuous integration wherever appropriate, which means that the components will be integrated incrementally.

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

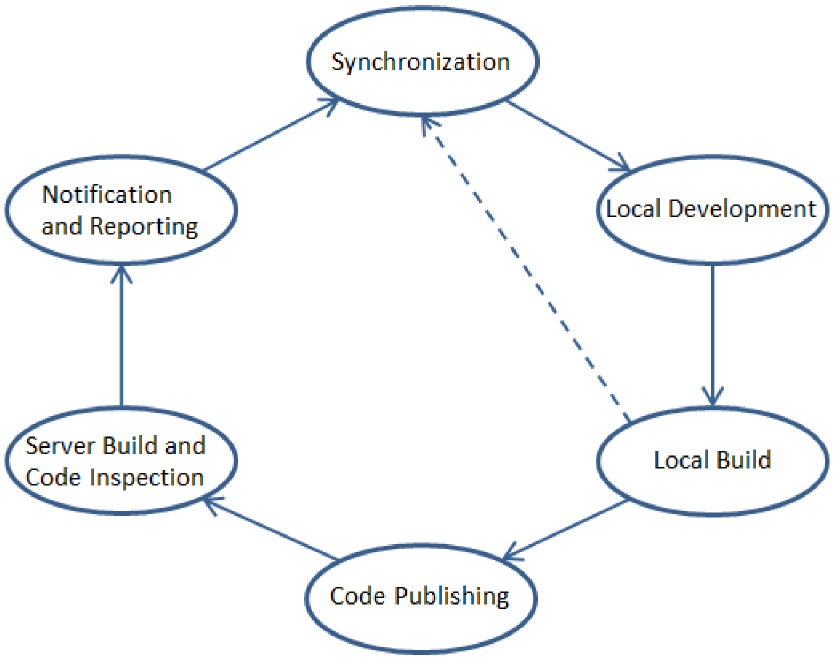


Figure 11 The Continuous Integration Process

4.5.1 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 his local development machine. This local copy is branched, and can now be modified so that it implements the feature required by the overall software system.

4.5.2 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.

In case that there are already local changes in a file which the developer wants to keep, but there are also changes in repository, the developer has to merge the changes.

4.5.3 Local Build

The developer who works on a software component and who changes the system makes every effort to ensure that the new feature does not introduce a bug into the overall system: once this has been done (and usually at various points while during the development) the developer triggers an automated build on his development machine. This takes the currently developed source code, 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 valid. The unit can be functionally accepted.

4.5.4 Code Publishing

After the local build succeeded the developer can publish the local changes of the source code. This is done by committing the local sources to the central software revision system. In case there are already committed changes to the source code in the software revision system, the source code has to be synchronized again before it can be published.

4.5.5 Server Build and Code Inspection

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 server performs a check-out of the most recent version of the software system from the repository. The code is built on the integration server and the automated tests are executed.

With certain plugins the continuous integration server can also perform static or dynamic analysis checks that address the test coverage of the source code, possible duplication of code and dependencies among classes.

4.5.6 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 what exactly he has done. It also serves to prevent sloppy code check-ins as mistakes will immediately become visible not only to the responsible developer.

Additionally, the continuous integration server can generate tests reports and reports about source code quality.

5. 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.
(ISO 25010, 2011)	Systems and Software Engineering. Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and software quality models. Technical Report ISO/IEC 25010:2011(E), ISO/IEC.
(JUnit, 2013)	JUnit. http://www.junit.org. Accessed 2013-06-17
(Kitchenham and Pfleeger, 1996)	Kitchenham, S. L., Pfleeger (1996): Software Quality: The Elusive Target. IEEE Software
(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, 2013)	NUnit. http://www.nunit.org. Accessed 2013-06-17
(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.
(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
(Weischedel and Versteegen, 2002)	Weischedel, G.; Versteegen, G. (2002): Konfigurationsmanagement. Springer.