Enabling the business-based
Internet of Things and Services

(FP7 257852)

D9.2.1 Annual Integration and Quality Assurance Report 1

Published by the ebbits Consortium

Dissemination Level: Public

Project co-funded by the European Commission within the 7th Framework Programme
Objective ICT-2009.1.3: Internet of Things and Enterprise environments

Document version: 1.0 Submission date: 2011-08-31

Document control page

Document file: D9.2.1 - Integration report.doc
Document version: 1.0
Document owner: Name (Organisation)

Work package: WP09 - Platform Integration and Deployment
Task: T9.2 – 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	Christian Prause, Alexander Schneider (FIT)	2011-07-27	Initial structure
0.2	Christian Prause (FIT)	2011-08-10	Contributions to several areas of the document, especially to state of the art, current status and GForge.
0.3	Peeter Kool, Matts Ahlsén (CNET)	2011-08-26	Updates to sec 3 and 4. Architecture and components. Lesson learned re. Configuration.
0.4	Pietro Cultrona (COMAU)	2011-08-30	Updates to lessons learned
1.0	Christian Prause (FIT)	2011-08-31	Final version submitted to the European Commission

Internal review history:

Reviewed by	Date	Summary of comments
Peter Kostelnik	2010-08-30	Approved with comments
Paolo Brizzi	2010-08-30	Approved with comments

Index:

1. Executive summary

The "Annual Integration and Quality Assurance Report" deliverable repeatedly reports on the progresses made in integration and quality assurance in ebbits from a technical perspective. The report is aimed at sub-tasks of tasks T9.2 ("Quality Management, develop monitoring of source code quality and usage of metrics") in order to ensure a lasting maintainability and fitness of the software developed in ebbits. With respect to this target, especially internal quality attributes of the code are important. The consortium agreed to use seasoned tools and techniques like iterative requirements engineering according to the Volere schema, software configuration management, automated build processes, unit testing, integration testing, web service testing, validation, continuous integration, coding rules, software metrics and code reviews. This report reviews the situation regarding adaptation of tools and processes by the consortium.

An essential component of quality management is quality control. The foundation of quality control is measuring. This report introduces software measurement from a theoretical perspective. After that the concept "quality" is discussed, and how quality influences the development cost of software. Next, the concept "technical debt" is introduced as an expression of a not-quite-right state of software artefacts. Technical debt temporarily speeds up development but interest on the technical debt leads to higher development costs in the long run. Furthermore, static analysis is discussed as a means of assessing the internal quality of software source code, and the Java Sonar platform is presented. Sonar includes current source code analysis toolkits, provides overviews like the value of the code base and the total technical debt. It also hints at the most serious problems in the code base and hints at starting points for improvement. This report also summarizes the iterative development approach of ebbits because a good and clear process methodology is important for creating quality software.

Section 4 reports on the current status of quality and integration in ebbits. It mentions two problems regarding the use of the Subversion repository as current practical problems. Also, the value of the Java code base has an assessed value of 500K Euros, including 19% technical debt.

Section 5 presents an overview of the integration status in ebbits. It describes modules, sub-systems and systems of ebbits.

Section 6 outlines the role of the integration infrastructure of ebbits, which is based on the web-based GForge software that provides an activity overview for creating awareness, mailing lists as communication support, Volere-based requirements management, a wiki for documentation purposes, software configuration management, and bug tracking.

Section 7 collects lessons learned with respect to integration and quality assurance.

Section 8 concludes this report.

2. Introduction

2.1 Purpose, context and scope of this deliverable

The "Annual Integration and Quality Assurance Report" deliverable repeatedly reports on the progresses made in integration and quality assurance in ebbits from a technical perspective. The report is aimed at sub-tasks of tasks T9.2 ("Quality Management, develop monitoring of source code quality and usage of metrics") in order to ensure a lasting maintainability and fitness of the software developed in ebbits. With respect to this target, especially internal quality attributes of the code (like Maintainability and Analysability) grow in importance. This task strives to strengthen the understanding of such quality attributes and, consequently, to improve the code quality.

Since the beginning of the project a considerable amount of efforts has been invested in planning and preparing integration and quality assurance. It was a common agreement to adhere to well-known software engineering methodologies to assure an acceptable quality of the software produced by the ebbits consortium. The consortium agreed to use seasoned tools and techniques like iterative requirements engineering according to the Volere schema, software configuration management, automated build processes, unit testing, integration testing, web service testing, validation, continuous integration, coding rules, software metrics and code reviews. The report aims to review the situation regarding adaptation of these tools and processes by the consortium.

3. State of the Art

3.1 Introduction to Measuring and Quality

3.1.1 Measuring

According to Fenton and Pfleeger (1997) measurement plays an important role in engineering and quality assurance. It is used to assess situations, track progress, evaluate effectiveness, and more. This section sketches the big picture of measurement, explains what typical problems are, and shows what implications this has for software engineering and quality management.

Starting from an obscure and esoteric specialty, software measurement has become essential to good software development. Unfortunately, measurement is not always acknowledged as being that essential for the realm of software development. So, there is a big gap between what could be measured and what actually is measured. A huge number of "measurement" definitions exist and some of them will be listed below and taken as a cue to investigate the problem.

Measurement is the process by which number or symbols are assigned to attributes of entities in the real world in such a way as to describe them according to clearly defined rules. In this definition an entity is a (virtual) object (e.g. a software program, or its code) — or an event (such as the testing phase of a software project). An entity has several attributes, which further describe the entity (e.g. its size). Measurement tries to quantify attributes, that is expressing its parameter value. This either happens in comparison to the same attribute of some other entity or according to some measurement scale (e.g. bytes). While comparative measurements are especially appealing to us humans — e.g. if something is larger than other things of its kind are on average, it is rather easy to say that this specific one is large — such quantifications are of lesser not scientific use and usually cannot be used for computations. What is needed in this situation is an exact measure on a precisely defined measurement scale.

In the field of programming, there is no commonly accepted clear definition for the process of measuring code or for what it consists of. Several limitations with measuring exist. In the following list we provide a non-exhaustive overview:

- The programming language a source file is written in, is certainly an attribute of that file. But can "Java" serve as a measure? Is a "Java" file more "Python" than a "C#" file? Hence, some attributes are not suitable as measures.
- Is there something like a "cleverness" index for algorithms? Can this be measured? Indicators for a "clever" algorithm could be its average case time and space complexity, the goodness of its approximations (if it does approximate at all) and so on. Multiple (measurable) attribute dimensions are projected down to a single number, but how this happens, the weighting, is a subjective matter or is dependent on the situation. The measuring instrument does not exactly measure what you want.
- The understandability of code is one of its quality criteria, but who or what decides whether code is readable or not? Asking different experts will probably yield similar results, but there will still be a lot of variance in their ratings. Also, if you ask experts of a different team that are used to apply different coding styles ratings will be quite different. The measuring device is not as reliable as it ought to be.
- If you use a profiler to determine processor utilization of a computer program or parts of it, you are likely to encounter minor inaccuracies due to program interruptions and stop timer imprecision. The measuring device is not as precise as desirable, but no reason to reject it altogether.
- Measuring the number of lines in a source file has different results depending, for example, on whether you do count comment lines or empty lines. The device must be handled correctly or results will differ. Although the device is perfectly accurate and used for the correct purpose, results are still different.

- Measurements are also expressed in different units and scales. File sizes could be expressed in megabytes instead of bytes, as this is more easily readable for humans, but it lacks precision. What scale is suitable depends on the purpose of the measure.
- After measuring values we can apply mathematical transformations on the values obtained. We hope that the newly obtained values bear additional information. However, we have to be careful what transformations we apply to the values. Some, like saying one method throws twice as many different exceptions as another, make no sense.

As mentioned in the last point, there are actually two ways of quantification: measurement and calculation.

Definition 1: Measurement is the direct quantification of an attribute, which is directly extracted by inspecting an the entity. Calculation means to mathematically transform and combine measurements or other calculations in order to extract further information in an indirect way.

The terms measurement and quantification will be used interchangeably, although — strictly speaking — one is rather a part of the other.

All quantification is done to first analyze entities and then to draw conclusions about them, which help us to understand what is going on. Knowing what is going on means to reduce the risk of unforeseen challenges. We are able to react to difficulties early on, before they can develop into major problems. This is an important aspect not only of software engineering, but also of many other scientific and engineering activities.

Many software projects fail to set measurable targets. Manufacturers promise that software will be user-friendly and reliable without specifying clearly and objectively what these terms mean. Managers fail to understand and quantify costs of software components correctly, which leads to excessive cost and frequent complaints from customers. But it is impossible to control costs without a clear understanding of where the costs come from. Often the quality of software products is not quantified or predicted during development. It is therefore impossible to tell potential customers what the quality (e.g. reliability in terms of likelihood of failure in a given period of use) of the product will be when it is finished.

Definition 2: Gilb's Principle of Fuzzy Targets: projects without clear goals will not achieve their goals clearly.

Without careful measurement and assessment if a novel tool or technology fits the development team — but instead relying on anecdotal evidence — projects fall prey to the extra efforts of trying to adapt to useless methodologies.

Measurement can also be used to determine the performance of teams or parts of teams. However, this is a completely different topic. Measurement can lead to top efficiency or have devastating effects on team performance.

3.1.2 Quality

Whenever there is a product (or a service) there is a quality associated with it. Quality plays an important role not only in the software industry but in all crafting or servicing. Consequently, it has been a research topic for ages, but nonetheless, currently there is no common agreement on the term. Geiger even postulates that the aim of unifying the term quality would be unreachable in the near future (Geiger, 1988). The root cause of the problem becomes visible when looking at the elementary differences of the five perspectives that exist about quality:

- there is an uncompromising and philosophical transcendent one,
- an economic and ingredient-centric product-based one,
- a subjective user-based one,
- a process-oriented manufacturing-based one, and
- a value-based one that considers cost and price (Garvin, 1984).

Depending on the different quality philosophy backgrounds, there are many different quality models. They try to decompose the vague term quality into quality characteristics to make it more understandable, and eventually measurable. Among the wide-spread models are those from Boehm, McCall, Dromey or the ISO. Generally, ISO's model is considered the most useful one due to the necessary international consensus (Al-Qutaish, 2010). The ISO 9126-1 (2001) identifies six characteristics: functionality, reliability, usability, efficiency, maintainability, portability. These characteristics are then further broken down into subcharacteristics (see Figure 1 ISO-9126 Quality ModelFigure 1), and attributes. Reaching high quality levels in all characteristics is desirable, of course. But besides other external project factors like cost and time that can impact quality goals, achieving high quality in a product often is not possible in practice. Sub-characteristics (and with them their superordinate characteristics) sometimes stand in conflict with each other. For example, adding code to improve the usability of a program (automation features, special cases, fancy GUI elements, etc.) will make the code larger and slower (Spinellis, 2006).

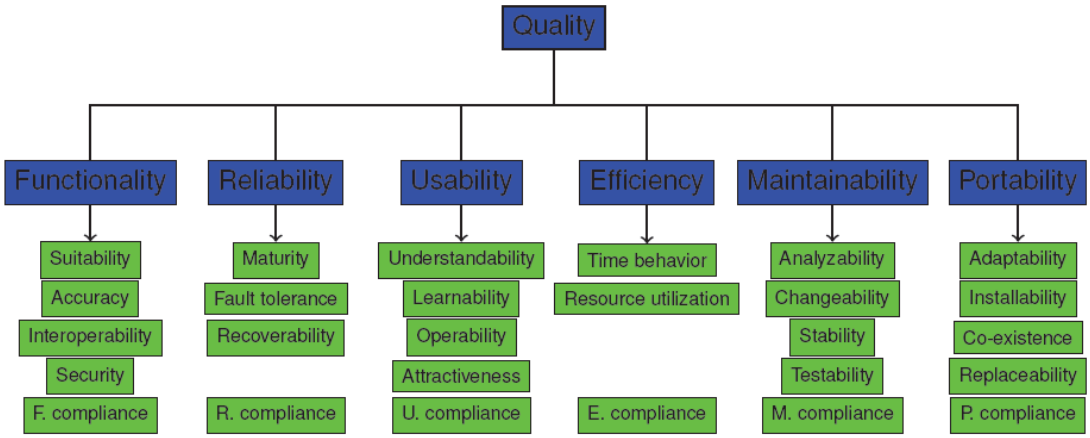

Figure 1 ISO-9126 Quality Model

In a prototype, especially internal quality criteria are important because prototypes are the basis of later products. Initially, there are no users that could be dissatisfied or get annoyed through e.g. usability problems with the software. No one expects full-fledged, flawless and efficient functioning from a prototype. Consequently, there is almost no risk and cost of software failures.

Internal quality problems, actually hinder the reuse of code and cause costs for those that later need to work with the code; be it just maintenance, or building into the code the other quality characteristics that are necessary in a product. The cost of quality is commonly expressed in the formula

$$C_{total} = C_{fail} + C_{prevent} + C_{appraise}$$

where C_{total} is the total cost of quality. It consists of the so-called positive and negative costs of quality. The negative cost C_{fail} results from problems of internal and external quality as in the examples above. The positive costs are the cost of prevention $C_{prevent}$ (e.g. metrics, process improvements, quality standards) and appraisals $C_{appraise}$ (e.g. review, testing, validation & verification) (Malik and Choudhary, 2008). Figure 2 shows the inverse relationship that exists between positive and negative costs: with an increasing investment in quality, the risk and fatality of software failures decreases (Galin, 2004). As the cost functions are non-linear, there is a minimum (or sweet spot) for the cost of quality.

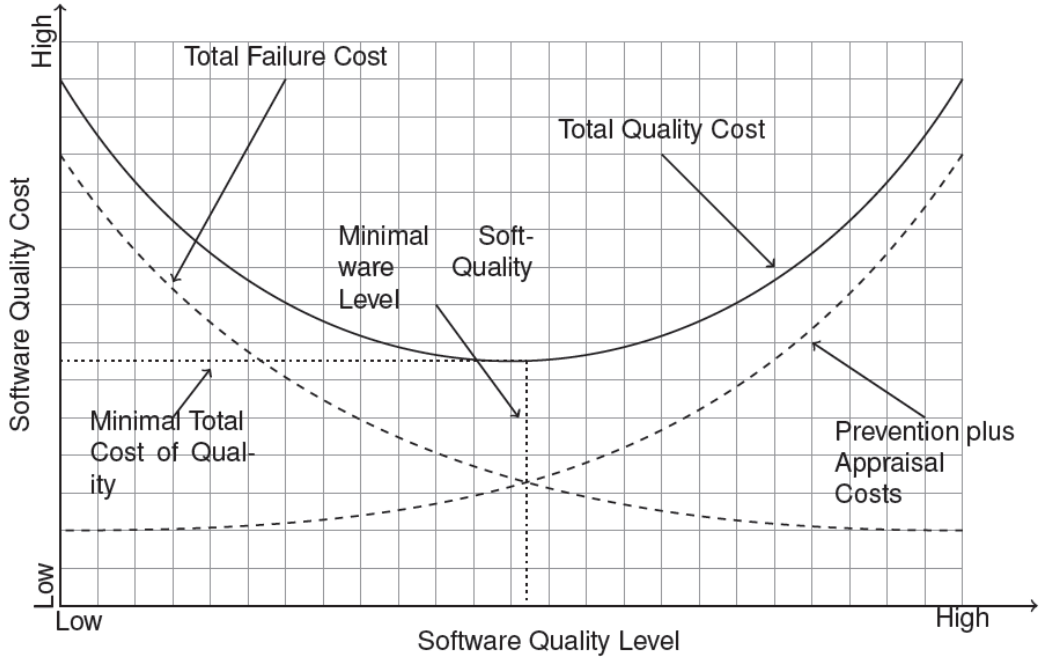

Figure 2 Sweet Spot of Quality Cost

This minimum is also what Crosby refers to when he says that quality is free. However, he continues: "Quality is free. It's not a gift, but it's free. What costs money are the unquality things — all the actions that involve not doing jobs right the first time." (Crosby, 1980) If quality is for free, and it reduces the development cost of software, then this is certainly where one wants to be no matter in what environment; even in one with a low cost of software failures.

3.1.3 Technical debt

Cunningham (1992) introduced technical debt as a way to speed development. In order to temporarily improve development speed, developers can go in debt by cutting down on development activities that do not implement new functionality (for example, doing documentation or creating unit tests). Technical debt, however, leads to a danger when that debt is not repaid promptly: all the time that is spent because software is not-quite-right, is regarded as interest on the debt. If there is too much technical debt in a project, it leads to extreme specialization of developers and brings entire development to a stand-still. The original debt term referred only to source code but has since been extended to other areas of the software process.

Technical debt characterizes the gap between the current state of a software's software process artifacts, and their hypothetical ideal state. Technical debt measures the cost for closing the gap between both states (Brown et al., 2010).

Fowler (2009) identifies two dimensions of technical debt:

- Reckless vs. prudent – whether the debt was incurred because the developers did not know better, and
- Deliberate vs. inadvertent – whether the debt was incurred intentionally.

3.1.4 Static Analysis and Tools

There are two kinds of analytic quality assurance of software. The first one is dynamic analysis, which is practically represented by various kinds of testing. This requires that the program source code is compiled into executable code, and then executed to observe if it is behaving as specified. The ebbits testing approach is described in deliverable D9.1; the choice of testing approach is

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

The second way to ensure software quality is static analysis, and its close relative software metrics. Static analysis means that the program is analyzed without being executed. Typical forms of static analysis are software metrics, various informal methods like code inspections and reviews, formal verification, but also the employment of specifically designed analysis tools (Floyd and Züllighoven, 1997).

Originating from pragmatic positions, software metrics measure different aspects of source code in order to serve as indicators for some quality aspects. The presumably oldest software measure is Source Lines of Code (SLOC). It has been shown to have a correlation with many software metrics. Ironically, however, it has limited value as a quality metric itself, but rather as a covariate for other metrics (Rosenberg, 1997). Generally, software metrics are a way to detect potential quality problems in software and code. They are a form of static code analysis and therefore complement testing as a means of software quality assurance.

Contrary to dynamic analysis that detects software errors through execution, static analysis can identify potential weak spots in a software without execution. Static analysis tools typically operate on the source code of programs, but there are also tools like Jlint[1], which analyse the compiled code of software. Weak spots that were identified by static analysis are then candidates for rework (or at least review) so that problems can be fixed before a costly software failure occurs. Unlike testing, however, static analysis can detect a wider range of problems for other quality characteristics. Besides revealing probable program malfunctions like infinite recursions, detectable failures include low test coverage (Alves and Visser, 2009) or coding style problems so that concerned code would be expensive to repair and adapt, or cannot be reused at all. Some organizations therefore have their revision control system configured to prevent checking-in of code that fails the static analysis (Ayewah et al., 2008).

If this is not the case then "defects" found in a file by static analysis can be interpreted as an indicator of missing quality. Counting the defects yields a quality measure. Of course, due to high complexity of the concept quality such an indicator is limited to certain aspects of quality in practice. For example, the usability of a software is difficult to assess automatically, i.e. without involving actual users. There are indicators of features in software that point at good or bad usability but a reliable usability evaluation will always require human user tests. The same applies to all the other characteristics of quality.

Code style checkers are part of the larger family of static analysis tools. These tools analyze programs — often by looking for specific patterns, but some also take into account the satisfiability problem (SAT)—to find defects before the program is actually tested. Static analysis programs either inspect the source code of a program or analyze its compiled form (Copeland, 2005).

3.1.5 Java Sonar

Sonar[2] is a web-based open source quality management platform, playing a crucial role in the effective construction of software systems by analysing and measuring source code quality, used and maintained by organizations according to their business objectives. Sonar addresses all so-called seven axes of software quality from both a practical and an academic viewpoint including architecture and design, comments, coding rules, potential bugs, complexity, unit tests and test coverage, and duplications. The Sonar website includes a community link and a blog providing a means of exchanging experience and information on every aspect of software quality, and the methods and tools used to measure and achieve it.

[1] http://artho.com/jlint/
[2] http://www.sonarsource.org/

By addressing all of the above mentioned aspects, Sonar succeeds in analyzing the explicit and implicit requirements a software should conform to: Rules, alerts, thresholds, exclusions, and settings can be configured through Sonar's web-based interface. Additionally, sonar leverages its internal database to not only allow to combine various metrics but also to mix them with historical measures. In this way, Sonar emphasizes three points:

- Software requirements are the basics from which quality is measured. A lack of compliance to requirements is a lack of quality. The section architecture and design in Sonar gives answer to this. In ebbits, requirements are not yet automatically mapped to Sonar reports. Instead, requirements are checked in the evaluation phases at the end of each iteration cycle.
- Specified standards define a set of development criteria that guide the software developers. If certain criteria are not followed, a lack of internal quality will result. The section rules compliance, violations, duplications, potential bugs, and complexity gives an answer to this point. The checks are based on well-known static analysis tools as Checkstyle[3] or FindBugs[4].
- A set of implicit requirements often goes unmentioned which nevertheless enhances the quality of the software program, like for example the ease of use, maintainability, efficiency, portability, and reliability. It needs to be noted that none of these factors are you either have it or you do not have it. Rather, they are characteristics that one seeks to maximize in one's software to optimize its quality. So, rather than answering whether a software product has factor x, Sonar instead answers the degree to which the software does comply according to efficiency, maintainability, portability, reliability, and usability.

Sonar is built according to a modular plugin architecture which allows source quality analysers to extend its functionalities like covering new programming languages: For example, additional languages that are covered through plug-ins are Flex, PHP, PL/SQL, Cobol, Java, Visual Basic 6, and .NET languages. But the plug-ins also add in new rules engines, or computing advanced metrics with additional powerful plugins.

- Quality Index — Calculates a global Quality Index based on coding rules, Style, Complexity and Coverage by unit tests.
- Technical debt — Calculates the technical debt on every component of projects with a breakdown by duplications, documentation, coverage, complexity. The plugin consists of four advanced measures:
  - The debt ratio
  - The cost to reimburse
  - The work to reimburse
  - The breakdown
- Total Quality — provides an overall measure of the quality of the project. It links code quality, design, architecture, and unit testing measurements.

[3] http://checkstyle.sourceforge.net/
[4] http://findbugs.sourceforge.net/

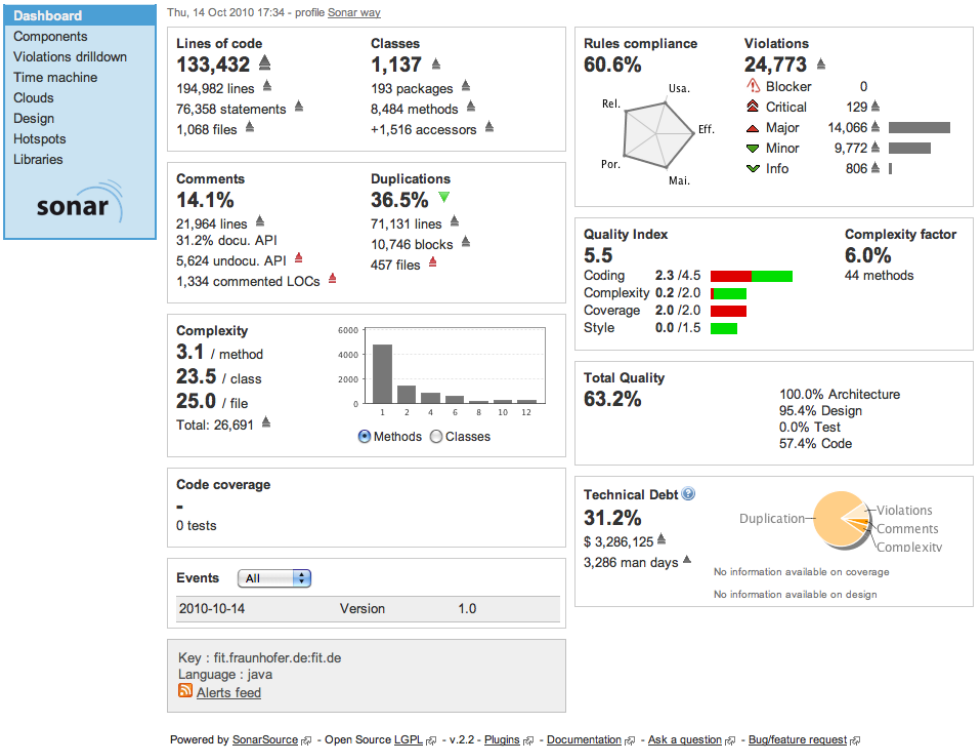

Figure 3 Java Sonar Dashboard

The Sonar Dashboard shown in Figure 3 gives an overview of the project's quality status based on several metrics. It contains basic counts like lines of code and the numbers of classes, packages, methods, etc. Next, there is the amount and length of comments, but also an overview of code duplications. Cyclomatic complexity for classes and methods is presented as well as compliance to coding rules.

Sonar is a single tool that recommends itself as the central measuring and quality surveilance tool. We make extensive use of it as a basis for this report. However, the output of Sonar can only be a starting point for quality and integration assessments. Human judgement is still necessary.

3.2 Iterative integration and development

The ebbits project has adopted an evolutionary requirement engineering, specification and design methodology, which includes a well-defined verification and validation process. The starting point of the iterative design process is a set of domain-specific vision scenarios delivering end-user visions of applications in the selected manufacturing domain.

The vision scenarios are used to derive detailed technical scenarios that will be discussed in focus groups with stakeholders. The result of this work in WP2 will be an initial set of requirements. The business and enterprise framework in WP3 results in sets of business and management rules, implementation of business eco-system analysis of ebbits applications and runs in parallel to the technical development, in order to use an enhanced optimization metric adding CO2 emissions and energy consumption to the traditional efficiency metrics (OEEE).
The architectural specification and models drives the research and development work in WP4, WP5. While WP4 develops ontologies and semantic components to be used by the Ontology Manager in the Service Layer, WP5 is more focused on components for distribution of intelligence for production optimization, which is the central part of the Application Layer functionality whereas WP6 develops components and knowledge base for integration with backend enterprise systems and physical world objects as part of the Application Layer. WP6's main task is to design and develop interfaces to the business management systems.
WP7 and WP8 are dedicated to development of the ebbits platform based on the models and components developed in the previous Work packages, but WP7 has the main task of develop event and Data Management structures for the Data Management Layer and the service execution

subsystem. WP8 develops network and communication infrastructure and semantic integration of physical world objects into the platform as part of the Network Management Layer.
At the end of each iterative cycle a prototype of the ebbits platform will be integrated and deployed in a user test bed. The integration is done in WP9 and use the models and components are developed in WP4-WP8. WP10 will develop the end-to-end application specified in the original scenarios and deploy the application in a field trial for validation by the users.
The validation results using fit criteria, together with Lessons Learned during the research and development work, will be fed back to the WP2 and drive the continued requirements reengineering work.The iterative process is visualised in **Fehler! Verweisquelle konnte nicht gefunden werden.**. Each cycle corresponds to one year of the four-year RTD project, with requirements being evaluated and the results fed back into the process together with the observations of developments in technology, market and regulatory standards reported in the related watch reports.

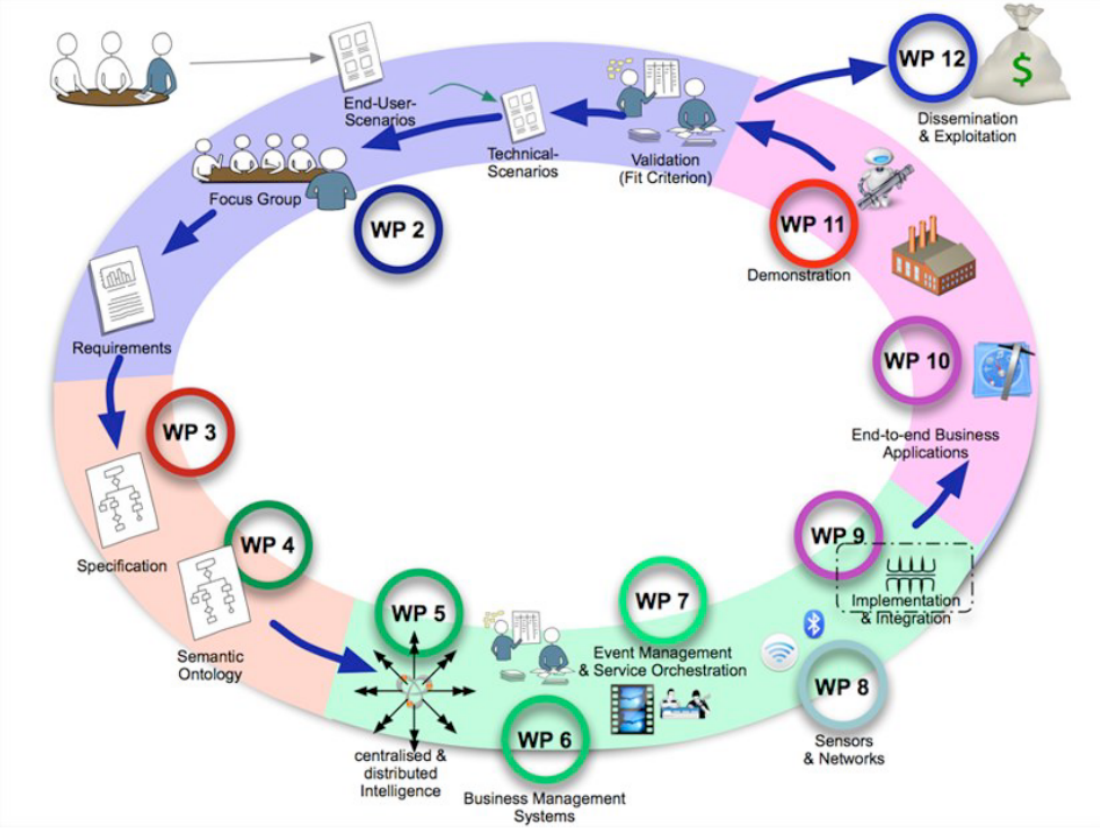

Figure 4 The iterative development cycle for ebbits

At the end of each annual cycle a prototype is planned with the specific purpose of illustrating the following aspects:

- End of year 1: First prototype of the ebbits platform serving as proof-of-concept
- End of year 2: Prototype II that will include the manufacturing optimisation scenario
- End of year 3: Prototype III incorporating the food traceability scenario and additional elements from the production optimisation scenario
- End of year 4: Fully operational prototype, combining all elements from the two domains into the final platform prototype and demonstrator

4. Report of the Current Situation of Quality in ebbits

This section reports on the current situation of integration and quality in ebbits. It reports what the presumably most critical quality and integration problems in ebbits are (see Section 4.1). Furthermore, this section is updated regularly with current information obtained through quality assurance tools like Java Sonar.

The first version of this report relies on information from Sonar only (see Section 4.2) but later versions will incorporate information from other sources, too. Other suitable tools that are identified will be reported about in the state of the art (Section **Fehler! Verweisquelle konnte nicht gefunden werden.**), and then – if appropriate – applied to enhance the results reported here.

4.1 Most critical current quality and integration problems

This section reports on the presumably most critical quality and integration problems identified in ebbits.

1. The most critical problem regarding the development and integration of the ebbits platform is the low use of the Subversion repository that can be improved. At the moment, not all partners actively contribute to the shared code base and commit regularly. We want to improve this situation by talking to partners, offering help, giving technology trainings, and outlining the importance of a shared code base for collaboration. Additionally, as not all code is available, quality indices, integration status, and code base statistics may not be fully accurate.
2. Another issue is that the structure in the Subversion has started to grow organically. The partners committing to the repository need agree on a shared, clear and simple structure. In fact, this issue has already been decided in the Test & Integration Plan. This matter will be brought up in the next integration meeting to make the intended structure more clear. D9.1 will then be updated accordingly.

4.2 Results from Java Sonar

It is not possible to analyse Java and C# code at the same time. Furthermore, Java Sonar is designed for use with Java. There is an extension for Sonar so that it can be used with C# and we are currently experimenting with it. However, this is the reason for why this report only treats the Java portions of ebbits at the moment.

4.2.1 Java Code

At the time of writing, the repository contained about 13KLOC, including about 10% lines of source code comments. There are 47 packages, 139 classes and 729 methods. 21% of the lines are duplicates (this is due to the automatic generation of source code for web services). According to Sonar with default settings, the Java code base is worth about half a million EUR.

Regarding an assessment of violations of the different quality characteristics from ISO-9126, the ebbits code base is 65% compliant to static analysis rules. The most violated quality characteristic is Reliability with about 700 violations. Second- and third-most neglected quality characteristics are Maintainability (660) and Usability (650). There are almost no violations regarding Efficiency and Portability.

The total technical debt accrued in ebbits is 19% of the value of the code base. Duplication is the severest problem and makes up for more than 50% of the total technical debt. Cleaning up the technical debt would take an estimated 136 person days.

Technical debt can become a problem when it becomes too much. Modifications to the source are usually considered to be more expensive, the higher the technical debt is. This phenomenon is

comparable to interest payments on financial debts. In the extreme case, too high technical debt can grind a software project to a halt, when all effort has to be spent on interest on technical debt.

5. Overview of Modules, Subsystems and Systems

As of the end of the first iteration at M12, the project has developed demonstrators for the manufacturing and traceability scenarios, based on a subset of the components of the complete ebbits platform.

The starting point for the demonstrator development is the initial architectural model of the foreseen ebbits platform, based on the first iterations requirements specification and on the Hydra middleware architecture. The motivation for the development of domain specific demonstrators is the need to validate the domain specific requirements and solutions, from which generic platform components can be derived. This work is thus performed in parallel to the design of the generic ebbits platform architecture, and with the development of software components from the individual work packages.

A note on terminology: The ebbits platform, refers to ebbits as a collection of coherent and reusable software components, which can be deployed in different configurations and combinations in order build the core parts of IoPTS-ERP applications. The ebbits middleware, refers to the implementation of the platform components as generic building blocks, and to their positioning in a software architecture.

The current ebbits prototype platform thus includes the set of common components from the two application demonstrators and a core set of middleware components being adapted from the Hydra/LinkSmart middleware. We emphasize that these component are in early versions and in some cases in the form of simulators and stubbs.

The figures below show the current (M11) set of component as they are deployed in the two demonstrators.

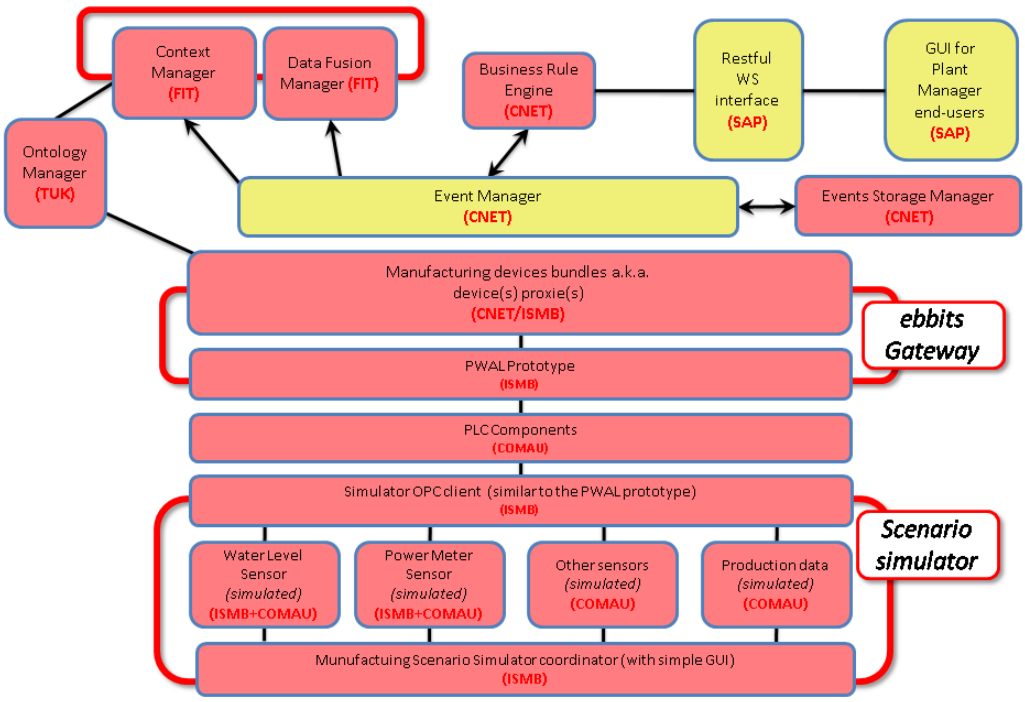

Figure X: Manufacturing scenario component architecture

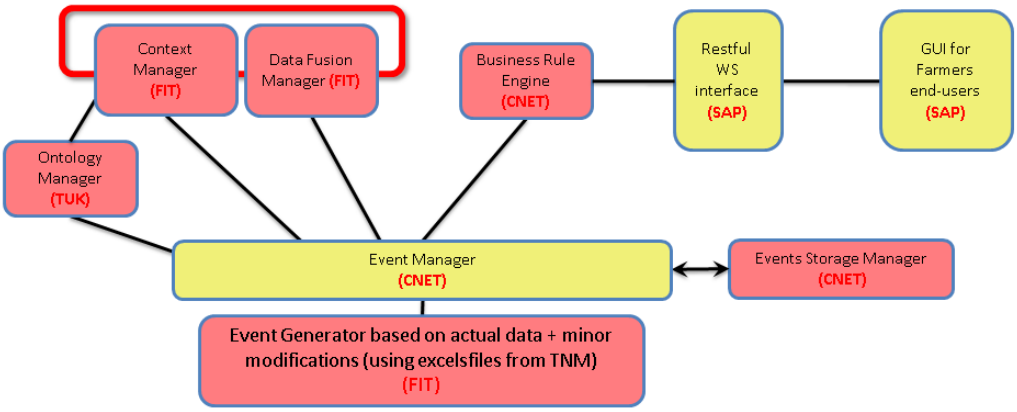

Figure Y: Traceability scenario component architeccture

The generic ebbits platform architecture is successively refined. In general, a system architecture can be modelled and depicted from several different views depending on the architectural qualities to be emphasized. A functional view focuses on the functional components, their interfaces and their inter-dependencies. Below is a functional view of the current (M11) ebbits architecture, depicted as a layered model, where the PWAL and physical devices represent lower layers, and applications and services the uppermost.

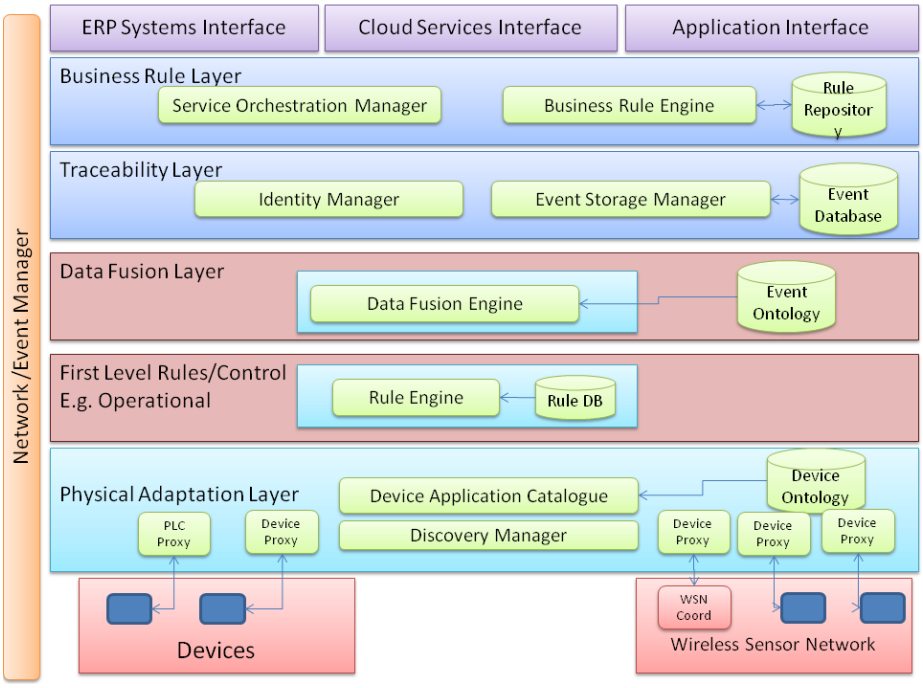

Figure z: Layered view of the ebbits platform architecture.

The operational integration work is based on a combination of meetings and workshops involving the technical work packages. Thus, Architecture Design meetings are followed by Technical Integration workshops, where on site integration and test of components is done, and subsequently fed back to the architecture design. The development work in the project is organised in four successive iterations. During the first development iteration (ending M12) work has focused on the basic test and integration infrastructure, the adaptation of the Hydra/LinkSmart middleware, and, on prototyping of the demonstrators. We anticipate the current architecture to evolve further in the coming iteration.

6. The Role of GForge for Integration and Quality

The ebbits consortium has decided to base its software development on the GForge software development platform. GForge provides a web-based development infrastructure that supports the most important phases of the software process with appropriate CASE tools: Activity overview for creating awareness, mailing lists as communication support, Volere-based requirements management, a wiki for documentation purposes, software configuration management, and bug tracking.

The tools for the different software processes are described in the following sections.

6.1 Awareness: Activity overview

Developing software is a highly collaborative process. However, collaboration requires coordination between several developers. Coordination is only possible if the developers are well aware of what is currently happening in the project. In co-located teams, awareness happens almost automatically in casual hallway chats and similar. But in distributed project, such awareness is much more difficult to create. GForge's activity overview (see Figure 4) aids in creating this awareness.

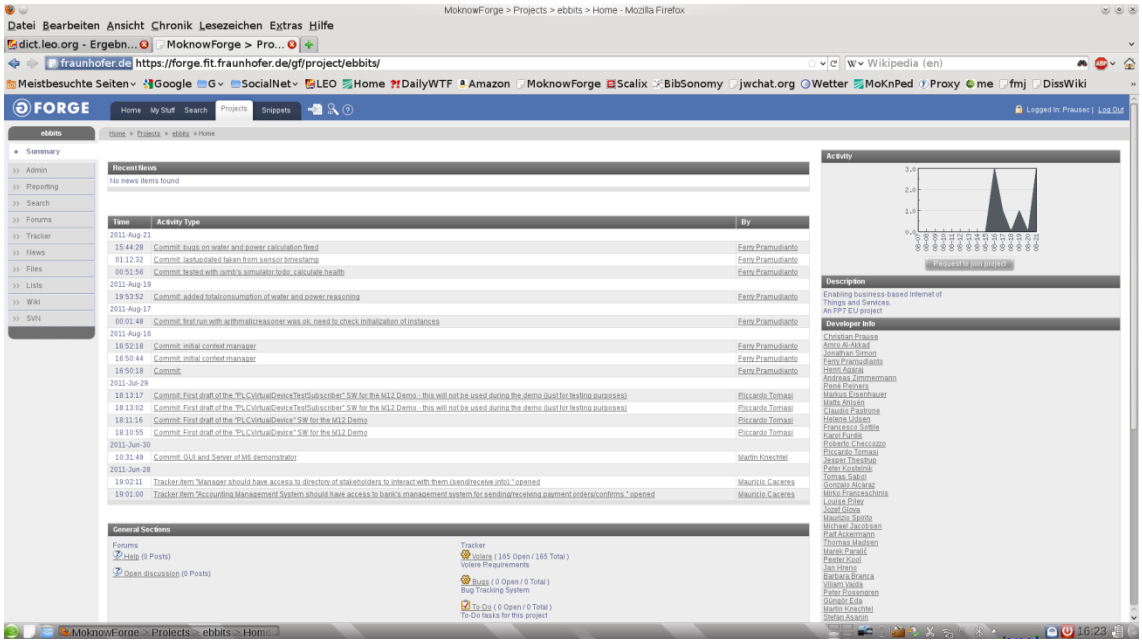

Figure 4 Recent Activity in GForge

6.2 Communication: Mailing lists

Mailing lists are an important way of communication in distributed projects. GForge respects this situation by providing mailing lists (see Figure 5). In addition to it, mailing list communication is archived automatically. Thus, messages are preserved and remain available for new persons entering the project later on, too.

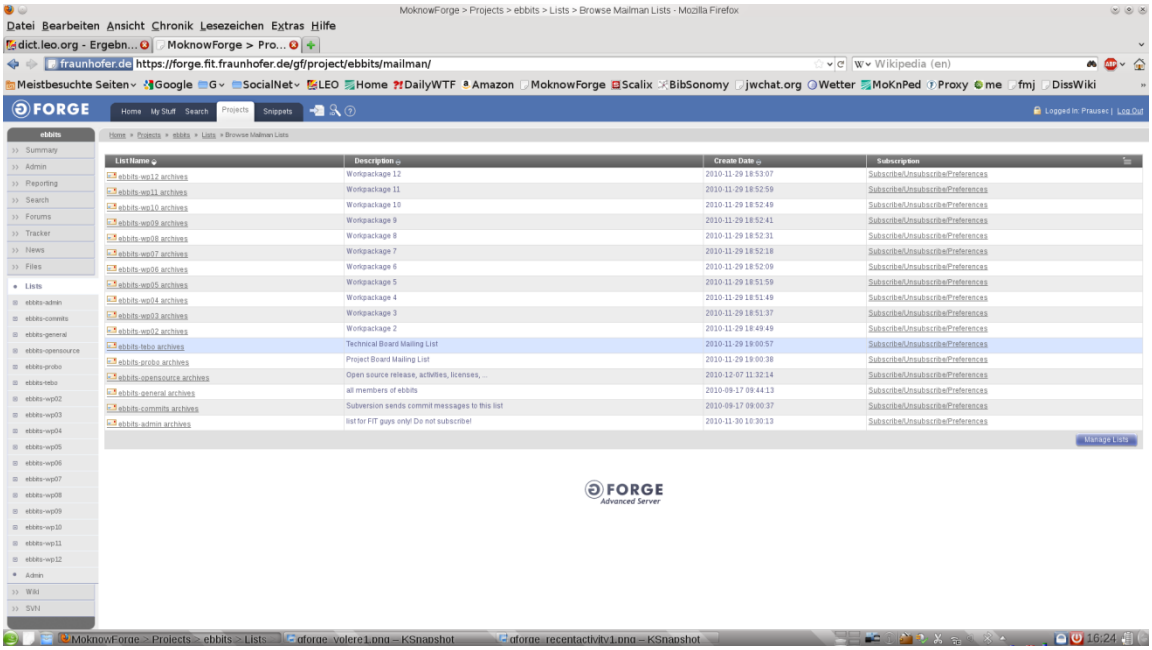

Figure 5 ebbits mailing lists in GForge

6.3 Volere requirements management

Ebbits manages its requirements according to the user-centered Volere requirements process to ensure that requirements adequately reflect user needs. The Volere process enables a high quality software with respect to external quality attributes like Usability. See Figure 6 for a screenshot of the Volere requirements overview.

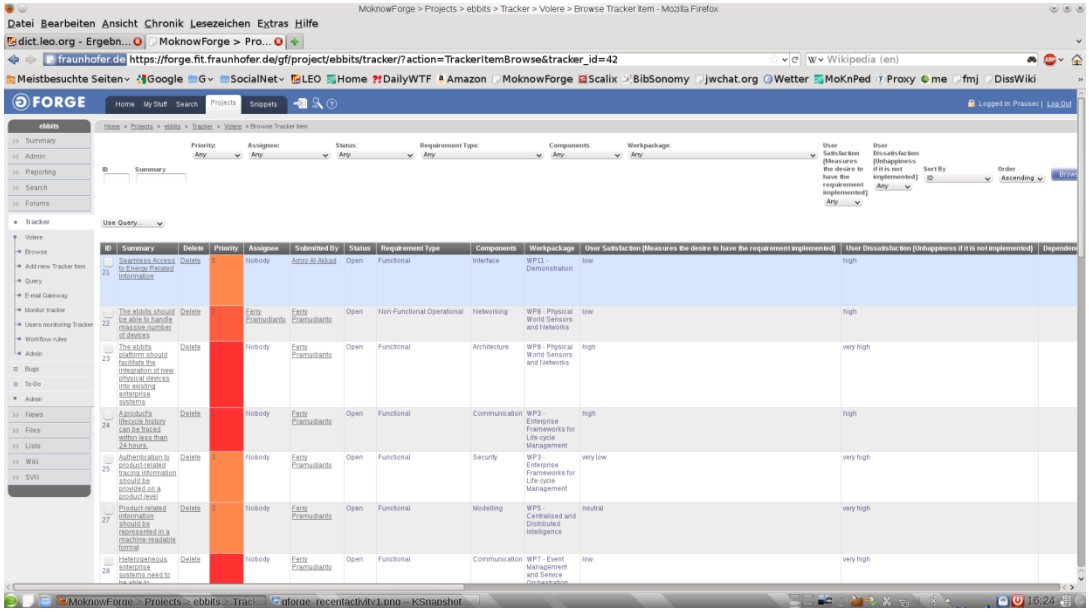

Figure 6 Overview of Volere requirements in GForge

6.4 Collaborative software documentation

Wikis provide fast and simple ways to collaboratively create text documents that remain easily editable. These features are especially important for documentation that needs to keep up with the changes of an evolving software. Figure 7 depicts access to the ebbits Wiki within GForge.

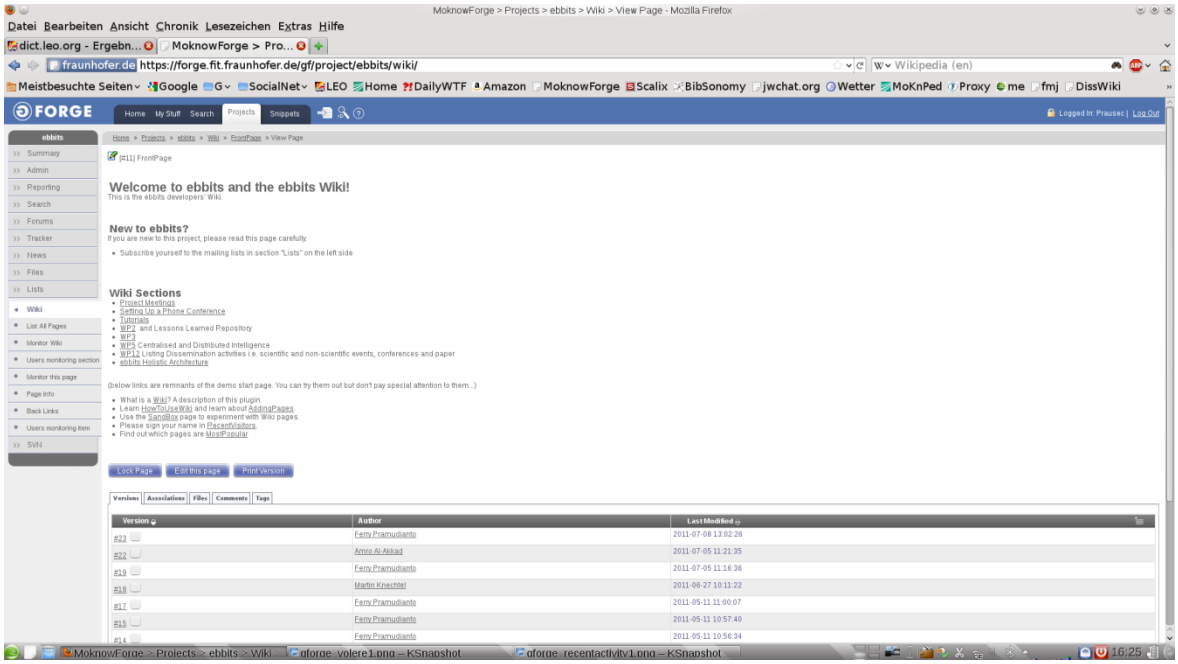

Figure 7 GForge integrated Wiki

6.5 Software configuration management

Software configuration management is considered the backbone of any engineering science. GForge includes the widely used Subversion revision control system. The Subversion repository enables all developers to always be able to access the most recent version of the software source code, a single point of access, and lets developers commit their changes to a well-defined place. At the same time, allowing access to older designs allows for gradual improvement and development of a product or prototype. A screenshot of the online repository browser of GForge can be seen in Figure 8.

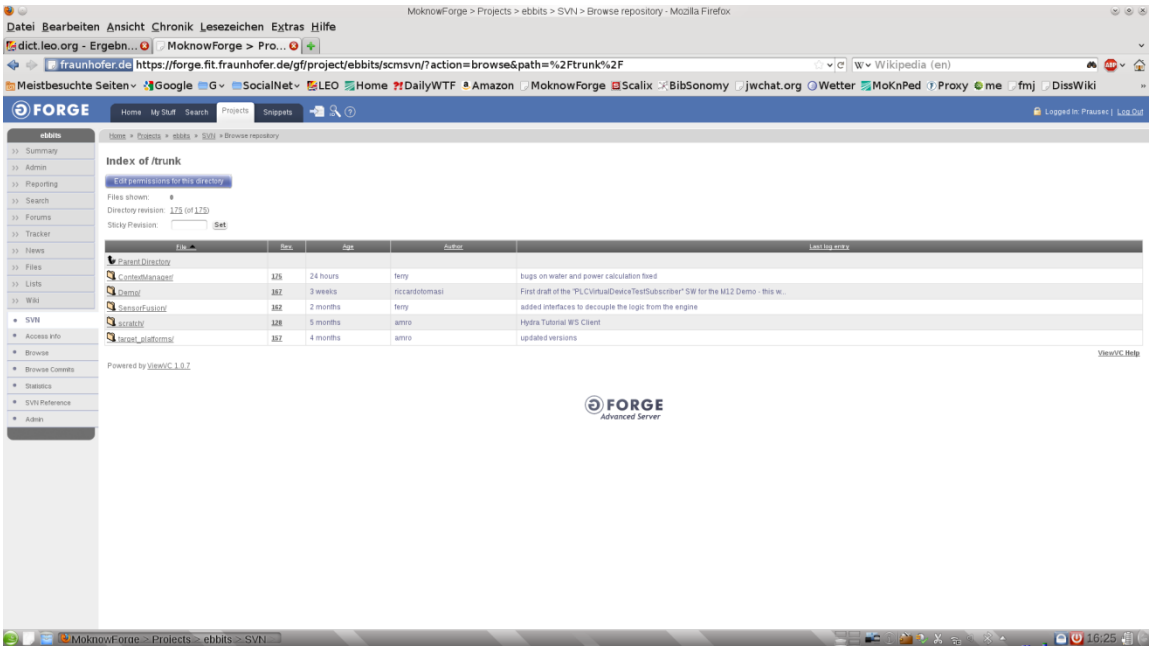

Figure 8 ebbits trunk in GForge Subversion repository

6.6 Bug tracking

Recent research shows that bug tracking is pivotal for the knowledge management in software project. Whenever a problem occurs, software developers often start their search for the cause of an

error in the bug database. Through its issue tracking system, GForge provides all the necessary bug tracking features.

7. Lessons Learned

SVN: AnkhSVN works very well in VS 2010. In the file system Turtoise works well as well.

SVN: AnkhSVN does not handle conflicts in project and solution files very well, needs to be manually fixed.

Configuration: Configuration of development environments is still time consuming, especially in the JAVA environment where one needs to download a number of tools such as ANT, IVY et c which each requires more or less configuration.

PLC Program: The software simulating the cycle of a production system has been developed on a virtual environment so not all real constraints have been taken into account. Thus, in order to ensure a proper operation in the real world several validation steps are required.

Implementation process: the design and implementation process of the ebbits embedded platform has been developed starting from the previous engineering experiences and is stable. To improve the existing process the application of checklists should be considered to ensure that all steps of the procedures have been applied.

Activities planning: the planning of all phases of the design, implementation and validation process is crucial to guarantee the achievement of the planned targets. Thus a tight planning with the relative follow up of all process steps are required in all the phases of the ebbits iterative process.

Sonar: the Sonar tool provides deep insights into the quality of a software project. Installation is not as trivial as expected, especially because Maven build files need to be provided (ebbits uses Ant). A major road-block is that Sonar is built for Java and is currently only able to analyse Java code. There exists a plug-in for C# code, too, but getting this to work is even less straight-forward than getting it to work with Java.

8. Conclusion

This report gave an introduction into theoretical foundations of software quality assurance with special focus on measurement as the basis of quality control. Additionally, the iterative development process was described, and GForge was introduced as the integrated support infrastructure for collaborative development activities in ebbits.

Two problems regarding the use of the Subversion repository were identified: insufficient commit activity and organic growth of the repository structure which is not aligned with the prescriptions from D9.1. These will be addressed in the next integration meeting by reminding partners of the importance good repository use.

In general, the project has made good progress. Although the project is in its first year and implementation could only start after other activities like requirements elicitation and architectural design are finished, the code base is already worth half a million Euros. The technical debt is at 19% (mostly due to copied code making up for 50% of the total technical debt), which is a bit high but certainly not a dangerous ratio.

Project integration is progressing well, and in accordance to the architecture of modules, sub-systems and systems presented here.

The lessons learned provide helpful hints for future integration activities and – especially with regard to quality assurance methods – might be valuable to other projects as well.

9. References

Use the following style for references.

(Al-Qutaish, 2010). Quality models in software engineering literature: An analytical and comparative study. Journal of American Science, 6(3):166–175.

(Alves & Visser, 2009). Static estimation of test coverage. In Ninth IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM09), pages 55–64, Los Alamitos, CA, USA. IEEE Computer Society.

(Ayewah et al., 2008). Using static analysis to find bugs. IEEE Software, 25:22–29.

(Brown et al., 2010) Managing technical debt in software-reliant systems. In FSE/SDP Workshop on Future of Software Engineering Research, FoSER, pages 47-52, New York, NY, USA. ACM

(Copeland, 2005). PMD applied. Centennial Books.

(Crosby, 1980). Quality is Free — The Art of Making Quality Certain. Mentor, New York.

(Cunningham, 1992) The wycash portfolio management system. In Addendum to the proceedings on Object-oriented programming systems, languages and applications conference, OOPSLA Addendum, pages 29-30, New York, NY, USA. ACM

(Fenton&Pfleeger, 1997) Software Metrics – A Rigorous & Practical Approach. International Thomas Computer Press, second edition.

(Floyd & Züllighoven, 1997) Informatik-Handbuch, chapter Softwaretechnik. Carl Hanser Verlag.

(Fowler, 2009) Technical debt quadrant (in bliki blog). Blog, online. http://www.martinfowler.com/bliki/TechnicalDebtQuadrant.html

(Galin, 2004). Software Quality Assurance: from Theory to Implementation. Pearson Education.

(Geiger, 1988). Begriffe. In Masing, W., editor, Handbuch der Qualitätssicherung. Carl Hanser Verlag.

(Malik & Choudhary, 2008). Software Quality — A Practitioner's Approach. Tata McGraw-Hill.

(Spinellis, 2006). Code Quality: The Open Source Perspective. Addison Wesley.

(Rosenberg, 1997). Some misconceptions about lines of code. In Fourth IEEE International Symposium on Software Metrics, pages 137–142, Albuquerque, USA. IEEE Computer Society.