Enabling the business-based
Internet of Things and Services

(FP7 257852)

D9.2.3 Annual Integration and Quality Assurance Report 3

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 control page

Document file: D9.2.3 Annual Integration and Quality Assurance Report 3 v.1.0.docx
Document version: 1.00
Document owner: Karl Catewicz (FIT)

Work package: WP9 – Platform integration and deployment
Task: T9.2 – Quality Management, develop monitoring of source code quality and usage of metrics
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
0.02	Karl Catewicz (FIT)	2013-07-30	First version with results from Sonar
0.03	Karl Catewicz (FIT)	2013-08-02	Chapter 3.2. introduced. Additional fixes
0.04	Davide Conzon (ISMB)	2013-08-26	Addition of the 3.4.2.1 section
0.05	Karl Catewicz (FIT)	2012-08-27	Summary & conclusion
0.051	Peter Kool (CNET)	2012-08-27	Test Beds chapter
0.06	Karl Catewicz (FIT)	2012-09-04	Internal review related changes
1.00	Karl Catewicz (FIT)	2012-09-04	Ready for submission

Internal review history:

Reviewed by	Date	Summary of comments
Paolo Brizzi	2013-08-28	Approved with minor comments
Jan Hreno	2013-08-30	Approved with minor comments

Index:

1. Executive summary

The "Annual Integration and Quality Assurance Report" deliverable repeatedly reports on the progress made in integration and quality assurance in ebbits from a technical perspective. The report is connected with subtask T9.2 with the aim of improving maintainability and overall quality of software developed in ebbits. 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 current report follows the path initialized in the D9.2.1. Source code monitoring methods discussed and established in previous deliverable were consequently used in D9.2.2 and in the current report.The current document focuses mostly on the status of current code base (sec. 3) and high level architecture (sec. 4)

After a short motivational introduction, section 3 reports on the current status of quality and integration in ebbits. Main part of it covers tests done with help of Sonar. Improvement of the total java based code base was measured in almost all fields like rule compliance, API documentation and code duplications. Code complexity slightly increased. The total java code base including both core- and ebbits specific linksmart modules was significantly refactored, resulting in a lower techical debt for the team.

To summarize the Java code base status: siginificant progress was made regarding the quality. The total "health" of C# code base is also good. It is comparable to the results from D9.2.2. Section 4 presents an overview of the integration status in ebbits. It describes modules, sub-systems and systems of ebbits. Section 5 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. Report of the Current Situation of Quality in ebbits

3.1 Introduction

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). Furthermore, this section is updated regularly with current information obtained through quality assurance tools like Java Sonar.

Like the previous versions, the third version of this report relies on information from Sonar only.

3.2 Motivation of having an automatic deployment environment

In the last report, chapter 3.2 emphasized why the need for Unit and integration tests are an essential part of software integration.
In the current chapter the focus concentrates on a automatic development environment.
Unit and integration tests are an important part of the whole deployment chain. This chapter presents a few arguments about why it is good to use such an automatic deployment chain.

In many cases software development is a manual process. Everything is manually installed.This approach requires expert knowledge of a proprietary setup. The knowledge is intrinsic and noteasily accessible to new developers. The whole deployment chain has been developed to satisfy todays urgent demands. The chains starts with the editor and ends with a released product.

The insider knows, for example, how to setup 3rd party software to run manual tests. He also knows what should be inside the final release binary to make it run.
Everything is hand crafted, requiring a sort of esoteric knowledge, thus extremly error prone. Also, if some problems appear in later stages, sometimes nobody knows where the error is located.

(J. Humble and D. Farley, Continuous Delivery) outlines a few important reasons why automated deployment is a goal worth to achieve.

- In not automated deployments, errors occur every time a deployment is performed. What is sure: the errors will occur. The only question is, are those errors significant or not.
- Without automatic deployment the process is not repeatable or reliable. As result of this, time is wasted on debugging deployment errors.
- Documentation is needed for a manual deployment process. Maintenance of documentations is a complex and time-consuming task. It could happen that the documentation itself is incomplete or out-of-date and deployment scripts always up-to-date and document the process. The deployment script is a representation of the process itself thus it documents the whole process itself.
- Automated deployments encourage collaboration, because everything is inside the script. A documentation is a weaker form of process description, because it will be interpreted by a human. Different humans have different levels of knowledge. In actual case the documentation is just a aide-memoire for the person who is performing the deployment. The hidden assumptions are not visible to outsiders. In short: manual deployments depend on deployment experts. If such an expert is missing because of vacation or job change, you are in trouble.

- Manual deployments are boring and repetitive but still require significant degree of expertise. Conscious human beings suffer from such tasks. You can free your expensive, highly skilled, overworked staff to work on really important tasks.
- The only way to do an actual test of manual deployment is to do it. It's often time-consuming and expensive. On contrary automated deployments are cheap and easy to test. Even a child can run it.
- Manual deployment process is less auditable than an automated one. A script is auditable, a fuzzy manual process is not.

Regarding the ebbits code base, the fully automated deployment approach is not part of it. Nevertheless it is recommended from the integration and quality point of view.

3.3 Results from Java Sonar

3.3.1 Sonar setup environment

Similar to D9.2.2 Java Sonar was used in current integration report. Details about the tool can be found in the D9.2.1, chapter 3.1.5.

Analysis results were created with the Sonar version 3.1. Sonar was installed on a Debian 6.0.5 VM. Sonar requieres a database. Postgres is used here. Tests scripts were triggered by Sonar runner 1.3 (recommended by Sonar). For the report creation, additional plugins were installed:

- Technical Debt Plugin 1.2.1
- C# Plugin Ecosystem 1.3 (without ndeps-plugin)

Analysis of C# code was possible with same ecosystem as presented in D9.2.2.
The C# plugin ecosystem requieres a .NET SDK installed. A Windows 7 VM was setup to provide the required environment. Results from sonar-runner execution were send to the Sonar service installed on the Debian VM.

3.3.2 Java code analysis

The current section describes Sonar results of the ebbits java code base. Becaue of refactoring, module names changed as compared to D9.2.2. The following java modules were analysed:

- NetworkManager
- CryptoManager
- OntologyManager*
- ContextManager*
- TrustManager
- GrandMessageHandler
- EbbitsContextManager*
- LinkSmartManagerConfigurator
- PWAL*
- LinkSmartMiddlewareAPI
- LinkSmartWSProvider
- LinkSmartMiddlewareClients

(*) ebbits specific code

Over 50.000 lines of total java code were scanned by Sonar. Over 50% represents ebbits specific code. There are 319 files, 97 packages, 442 classes and 3539 methods. 22.65% of total code is commented. 65.35% of the API is commented. 27.25% of the code, are duplications.

The duplication rate in the LinkSmart code base is around 16%. This is about twice as much as in Jboss with 8.1% (Nemo,Jboss) or Apache 8.2% (Nemo,Apache). Still this is a big improvement. Compared to values from last report, the duplication rate dropped from 32%.

In the ebbits specific modules 88% of the duplications come from automatic code generators. This can be omitted in future by refactoring web service stacks (stubs & skeletons) into one package. In general, the duplication rate dropes slightly (3%). The biggest contributor of such automatic duplications remains eu.ebbits.pwal.impl.driver.pwaldriver_plc.support_libs.opc.OPCXML_DataAccessStub from the PWAL module. This module is today under a completely refactory activities, but, being a work in progress, it has not yet added to the main ebbits java code base. In order to partly fill this gap, the section 3.3.2.1 provides a code quality analysis of the current PWAL 2.0 development. The PWAL2.0 development fully addresses the code problems identified since the previouse relase of this deliverable, while the next one will contain a new and more exahustive analysis of the total code.

76.4% of total code base complies with static analysis rules. A 5% increase of rule compliance can be detected for the LinkSmart code base. Overall complexity of code is 3.5 per method. No big changes are detectable here.
The total technical debt is around 14,7% for LinkSmart and 24.5% for ebbits specific modules. This means a drop by 12% for LinkSmart and 2% for the ebbits specific modules.

As stated in previous deliverable, duplications create big chunk of the total technical debt. Clearing the technical debt from the total ebbits java based middleware would take estimated 397 man days.

Statistics for total LinkSmart middleware code analysis can be seen from Fig.1. As you can see from Fig.1, unit test coverage analysis is skipped due to the mentioned lack of unit tests.

For the analysis of code violations, the default "Sonar way" profile with 115 rules was chosen. Regarding the rule violations there are a few worth mentioning:

- Emtpy if Stmt (statement)
- Security – Array is stored directly
- Unconditional If Statement
- Avoid Catching Throwable
- Useless Operation On Immutable
- Equals Hash Code
- Equals Null

Developers should review them in the future. For exact definitions, please read the Sonar documentation.

When only ebbits related modules are taken into account, still 271 man days have to be invested to clear the debt (see Fig.2). Compared with the whole LinkSmart middleware, the ebbits specific modules have lower complexity, more duplications (PWAL module), slightly better documentation and rules compliance. The ebbits specific modules also lack unit tests.

The technical debt plugin masks the information about test coverage ("No information available on coverage"). 6 out of 17 LS modules have a test folders. Because they are hand crafted, sonar cannot analyse them. Further consolidation is needed to improve it. Only 1 of the ebbits specific modules has a test folder. Because of it the technical debt pie-chart does not show a coverage slice. Nonetheless the missing test coverage remains the biggest debt for the whole project. All other factors like duplications, rule violations, comments or complexity contribute much less to the overall debt.

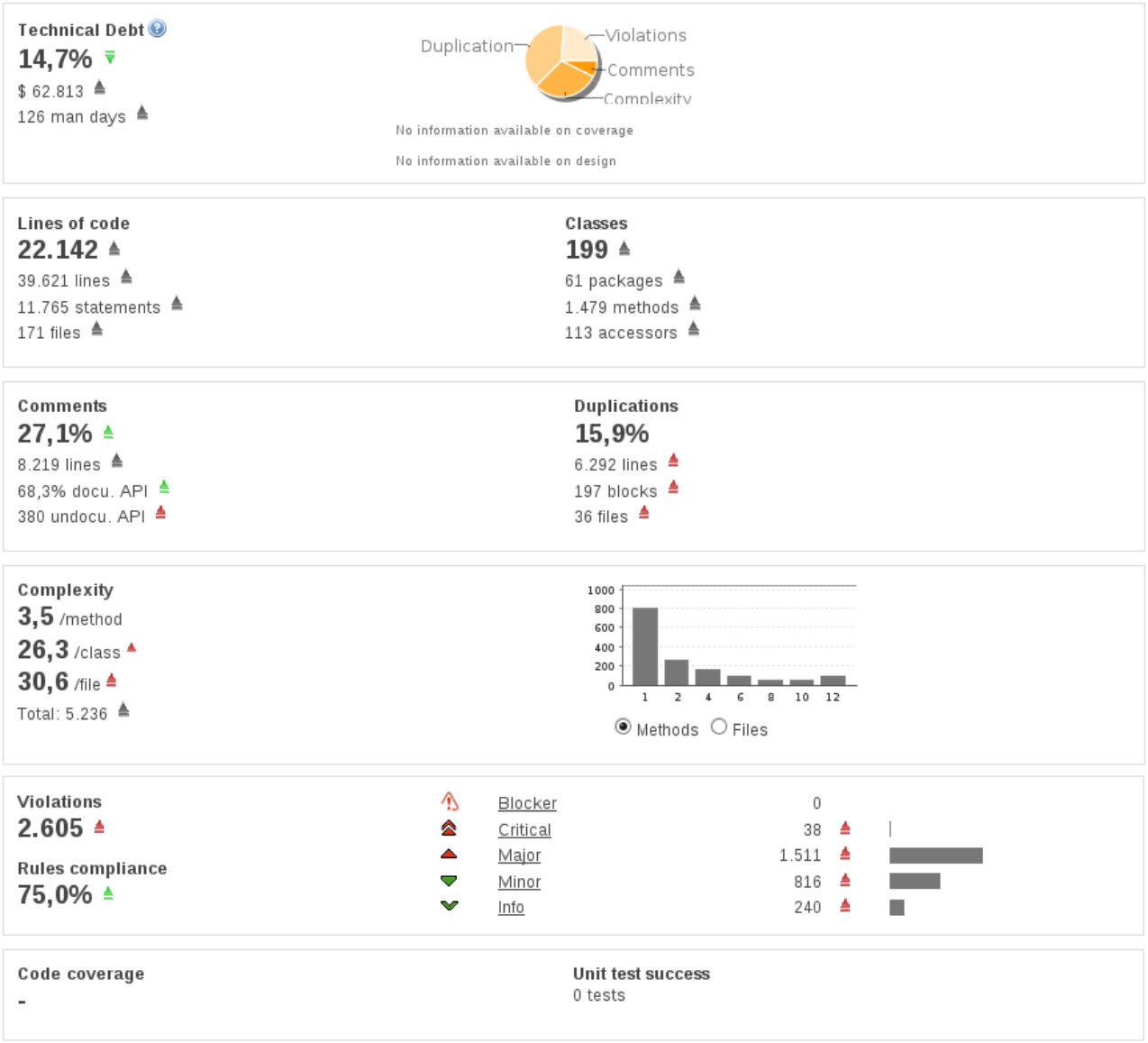

Fig. 1 Sonar results for LinkSmart code base

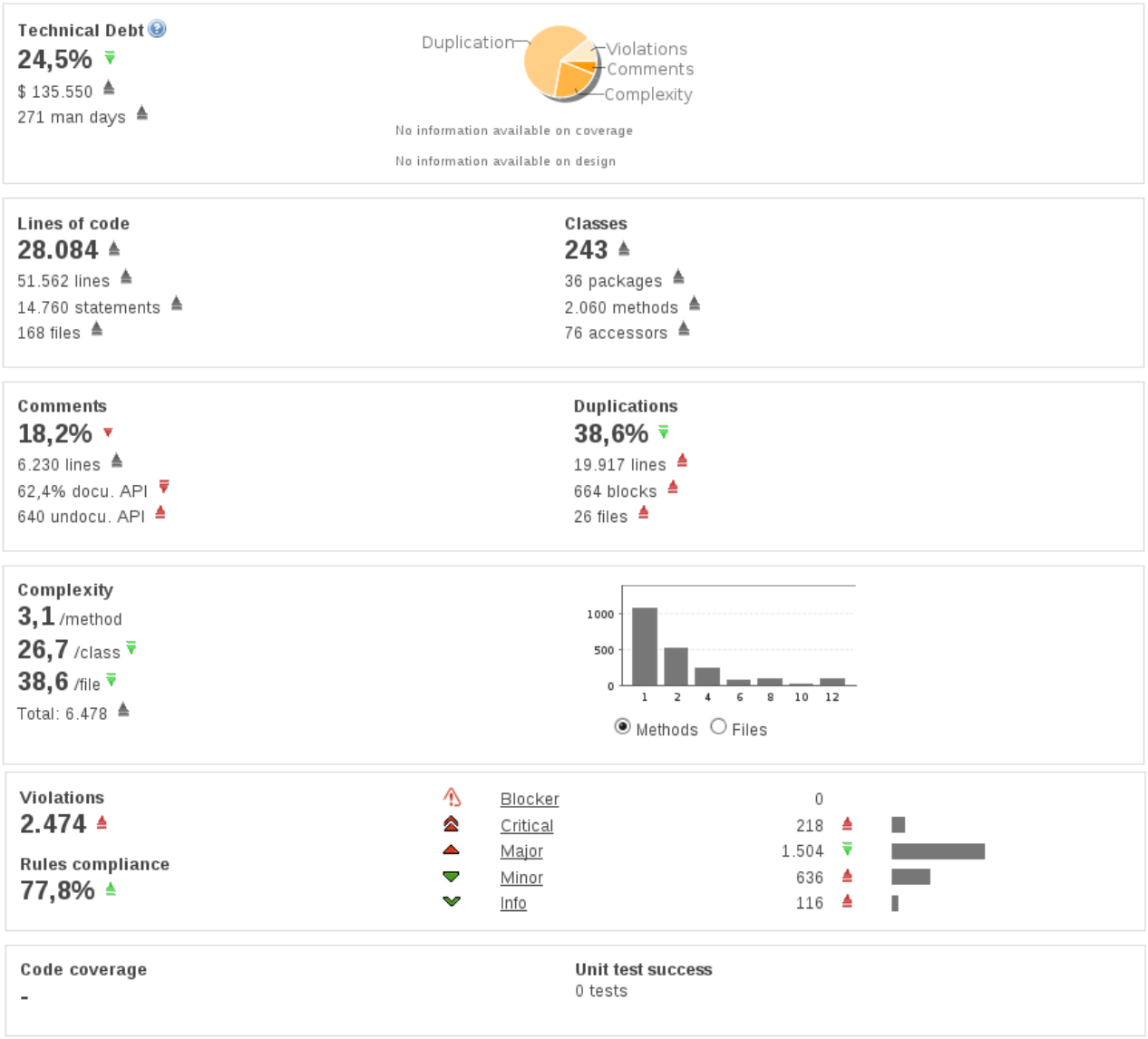

Fig. 2 Sonar results for LinkSmart Ebbits specific java code base

Compared to previous deliverable (D9.2.2), one can see a improvement.
Rules compliance enhanced from 72% to 75% in LinkSmart code base. 75% to 77,8% for the ebbits specific modules.

The API documentation stays at approx. the same level. Duplication rate droped significantly for the LinkSmart code base. The code complexity for the total code base remains similar compared to the last report. Also the total number of lines was reduced by around 3000 for the total code base.

Some special remarks about double checked in code are a small issue. This is not just a duplicated code but a code with even same package names. Following parts of code can be easily revisited and refactored.
One comes from the LS core :

Duplicate source for resource: eu.linksmart.network.identity.IdentityManagercomponents/LinkSmartMiddlewareAPI/src/eu/linksmart/network/identity/IdentityManager.java

And two from the ebbits specific modules:

Duplicate source for resource:….eventschema.ObjectFactory, on file: Context/ContextManagerEM/src…./eventschema/ObjectFactory.java only/Context/ContextManagerEM/src/it/polito/seempubs/eventschema/ObjectFactory.java
Duplicate source for resource: eu.linksmart.contextmanager.EntityManager, on file: Context/ContextManagerEM/src/eu/linksmart/contextmanager/EntityManager.java

The general impression of the java based code is good. Test coverage should be considered as the main goal for future development. Other quality characteristics either increased or stayed at the same level.

3.3.2.1 PWAL 2.0

In D9.2.2 there wasn't a detailed analysis of the PWAL, but it was indicated as one of the main sources of duplications, for the ebbits java code. In this section, there is a comparison of the new PWAL structure (PWAL 2.0) with the old one.
In the previous version of the PWAL all the code was contained in a single project, while the PWAL 2.0 is currently composed by 7 projects:

- PWAL.Framework (the PWAL core)
- PWALDriver.RobotController (driver for the Robot Controller)
- PWALDriver.OPCXML-DA (Driver for the PLC)
- PWALDriver.LLRPReader (Driver for the reader RFId compliant with LLRP)
- PWAL.Test (Test suite for the PWAL)
- PWALDriver.WSN (Driver for the WSN)
- PWALDriver.CoolingCircuit (Driver for the Cooling Circuit)

The first 4 projects are almost completed, instead the last 3 are still under development. For this reason, in this analysis only PWALFramework, PWALDriver.RobotController, PWALDriverOPCXML-DA and PWALDriver.LLRPReader are considered as part of PWAL 2.0.
In the following figures, the Sonar analysis for the two PWAL versions is presented. Figures 3 and 4 show the Sonar output of the PWAL 1.0 and 2.0 module analysis.

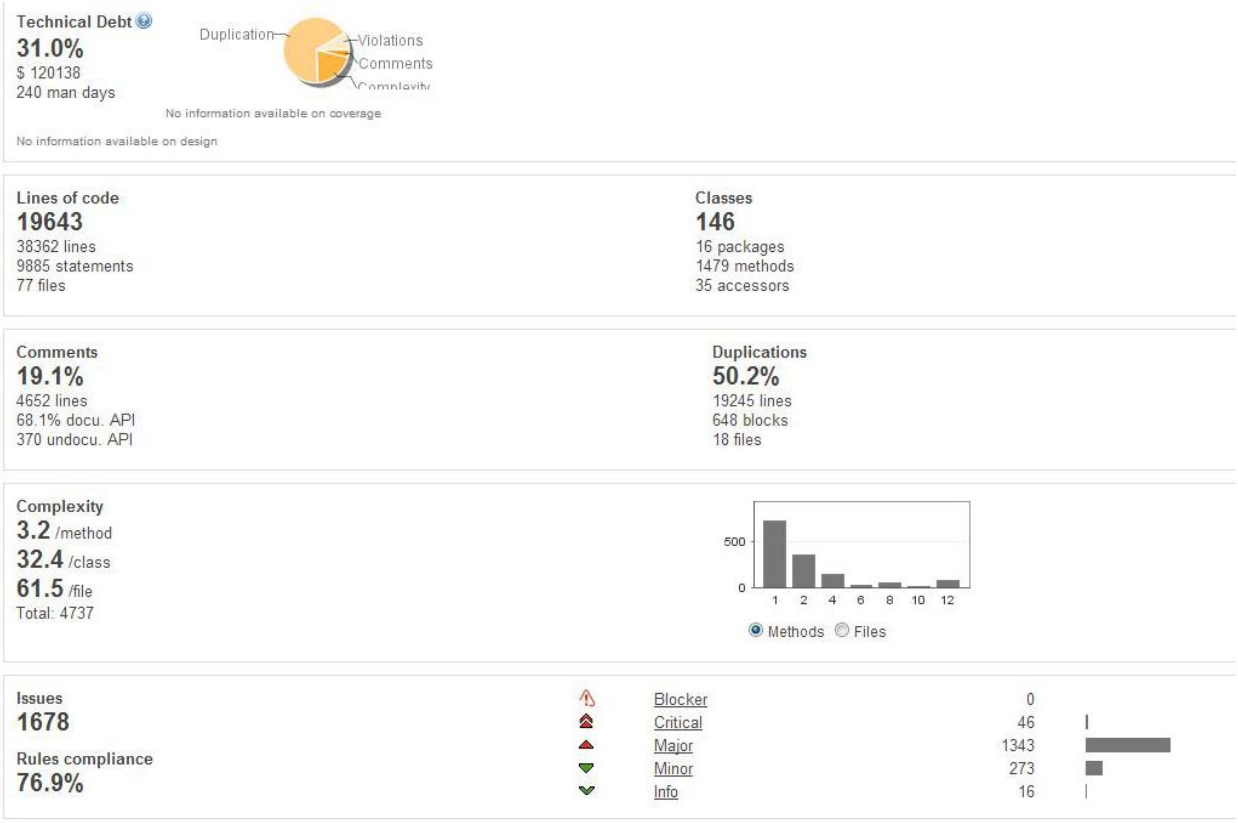

Fig. 3 Sonar Results for PWAL 1.0

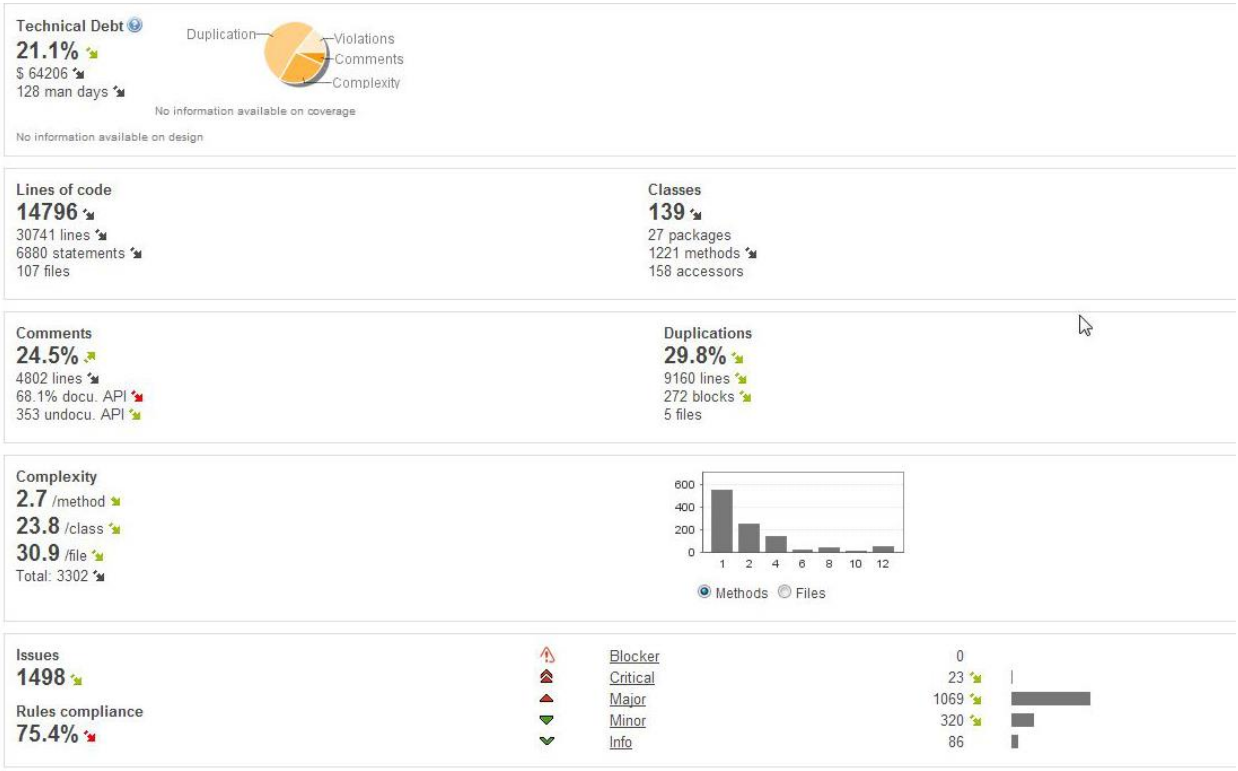

Fig. 4 Sonar Results for PWAL 2.0

Analysing the results obtained with Sonar, it is evident how the Technical Debt of PWAL 2.0 has been reduced, compared with the previous one, and also the duplication level has

significantly dropped. This has been possible because, as stated in D9.2.2, mainly the duplication problems were located in PWALDriver.OPCXML-DA. This OPC XML-DA driver had a high level of duplication (and consequently an high Technical Debt), mainly due the class auto-generated:

eu.ebbits.pwal.impl.driver.opcxmlda.stub.OPCXML_DataAccessStub

Among the approaches suggested to resolve the problems, in the section Lesson Learned of D9.2.2, the second one has been choice for this release: the code of DataAccessStub has been optimized, removing the part not useful for the PWAL Driver. Also if this approach has allowed to have a reduction of almost the 50% of the duplications in the PWAL code, to further reduce the value, in next releases, the different choices should be reviewed, to understand if there is a better solution (maybe not available now, i.e. a new standard stub).
The average rules compliance is a little decreased, but it will be simply improved in next releases, with a simple redesign and refactoring of the modules.
The average documentation of the API is about the same, but can be improved especially in the PWALDriver.RobotController, where the value is particularly low.
Also for PWAL 2.0, besides the refactoring and the completion of the components not yet finished, one of the main focuses of the new developments will be the test coverage, with the improvement of the test suite contained in PWAL.Test.

3.3.2.2 WSN Proxy

The WSN Proxy is available for quality analysis for the first time in this deliverable release (please refer to D8.5.1 and D2.5.3 for more details about modules functionality). This Java project includes the following modules:

- Data receiver and decoder from the WSN PWAL (for monitoring packets).
- Data encoder and transmitter to the WSN PWAL (for control packets).
- Data processing for both the manufacturing and the traceability scenarios (for differentiating the information required by each application).
- Multi-radio package (for the traceability scenario).
- Proxy implementation to connect with the LinkSmart environment (for remote monitoring and control).

The project was analysed using Sonar, and the analysis is shown in Fig. 5.
The technical debt value of 18.6% is mainly due to the violations to programming conventions, followed by complexity, duplication, and comments as illustrated in the chart. Detailed analysis is provided in the following. The low value of 9.1% for rules compliance is mainly due to, first, the excessive use of console messages (system println) instead of using a debugging library; and second, not following a standard naming convention for variable names. These issues as well as others with fewer occurrences will be addressed in the near future through the use of more appropriate libraries and refactoring of the code.

The project complexity is due to the relatively large number of involved classes. This is the result of that the project performs several independent functionalities, as listed above. Efforts will be made to address this issue by redesigning the code, while keeping the same functionalities intact. The duplication ratio of 5.3% is relatively low; it is caused by a few repeated routines for data reception and processing. This issue will be addressed by creating and invoking the proper methods.

The commented code metric has a relatively high value of 31.3%. This is caused by code blocks that were used during earlier stages of development then replaced and commented afterwards. This will be addressed by simply deleting the unused code blocks.
Finally, the documentation ratio has a value of 55.8%. Although the most functional and effective parts of the code are documented, this value is not as high as expected. This is because of two main reasons; first, a few recently developed methods are still to be documented ; second, some of the less functional methods were not documented such as the ones used for configuration, encoding/decoding, and numerical conversions. This issue will be

addressed in the near future by generating the missing comments and revising the code for other potential methods that may require documentation.

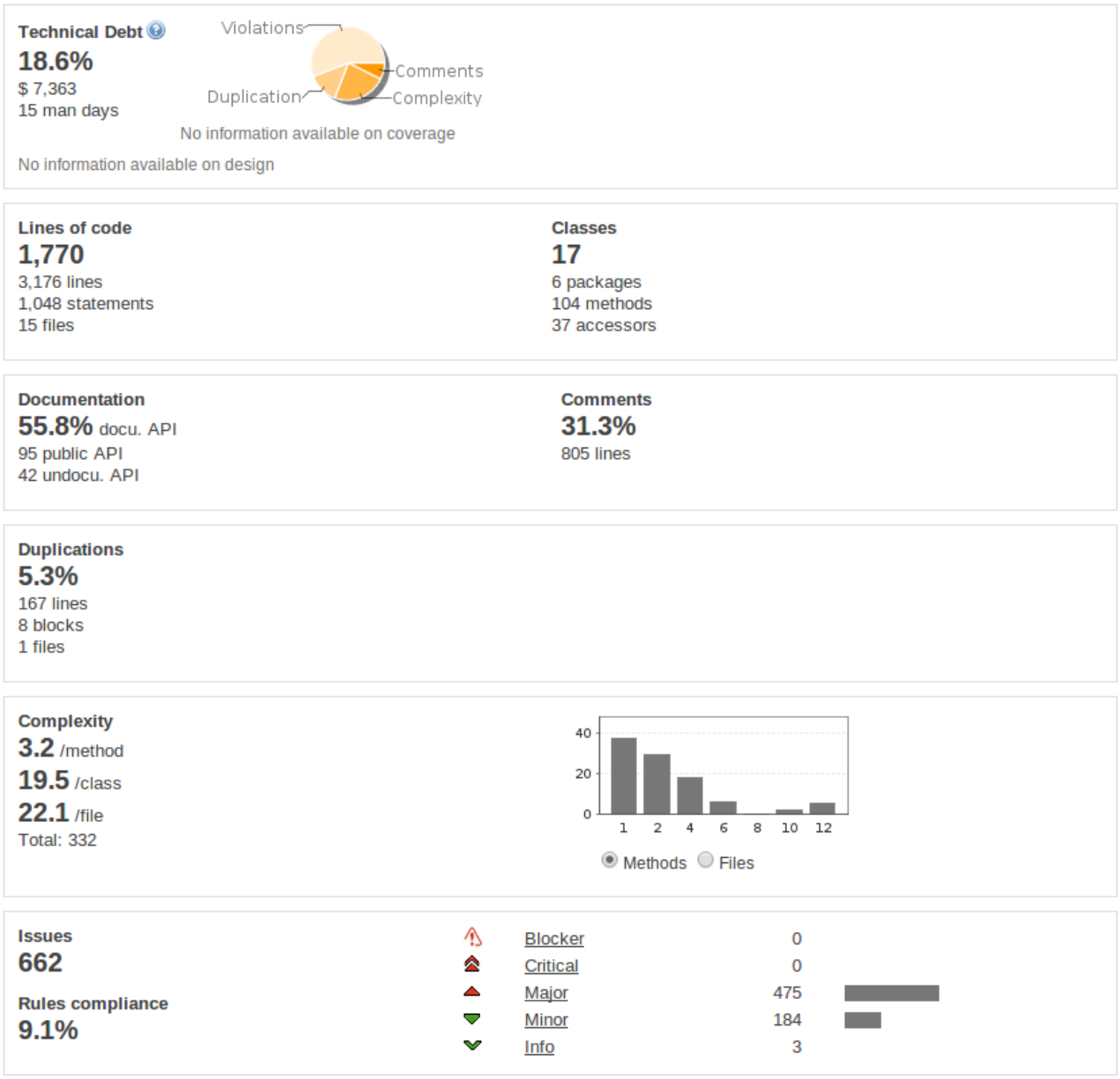

Fig. 5 Sonar results for the WSN Proxy project.

3.3.3 C# code analysis

Parts of the ebbits code base are written in C#. Since D9.2.2 reports can provide analysis of the C# code.

The ebbits repository contains three C# based modules, namely:

- EventManager
- EventProcessingAgent
- LMStationProxy
- ebbitsBRE

The total number of C# lines of code is around 7500. There are ~85 files, ~100 classes and 650 methods.

The total C# code is commented at rate of 20%. Although LMStationProxy comes with only 6%. API comment rates are at 40-50% (9% LMStationProxy) Compared to Java code, the duplication rates are low at around 0-13% . Static rule compliance ranges from 11 to 70%. The EventManager has the lowest compliance rate(11%).
With 1.3-2.4 per method, the complexity of the C# code is rather low.

The total technical debt can be estimated with 14-33%. It is estimated 51 man days are necessary to clean up the accumulated debt.
Also no unit tests are provided by the code base therefore test coverage analysis has to be skipped.

The following figures 6,7,8 and 9 depict Sonar statistics for all three C# modules. An accumulated analysis for all modules was not possible. The Sonar C# plugin requires one project file (.sln) per run, but there are three of them in the repository. A parent project can solve this shortcoming.

It can be seen that code coverage seems to be the biggest debt for all three modules. It is followed by rule violations and missing comments in one case (LMStationProxy).
For the analysis of code violations, the default "Sonar C# Way" profile with predefined 316 rules was chosen.

The code also contains some rules violations. Those violations should be reviewed in the next iteration. Here a few of them are listed:

- Use built in type alias
- File may only contain a single namespace
- Sections of code should not be "commented out"
- Fields must be private
- Interfaces names must begin with 'i'

Especialy EventManager and EventProcessingAgent contain lot of commented-out lines of code. Check (SonarQube,Commented) to learn about this issue.

In current report the C# code was analysed for the second time, thus the outcome can be compared with results achieved from D9.2.2 . The overall code increased by around 30%. The total technical debt grew linear to the code amount by around 28%. The complexity and duplication rates remain quite low with insignificant changes.
The comment rate stays a roughly the same level compared to the last report.
The rules compliance of EventManger droped from 18 to 11%. This happened mostly due the out commented code. Removal of the unused code is suggested to increase the code quality.
The general impression gives no spectacular changes of the #C code base. Besides the increased amount of code no big shifts are detectable.

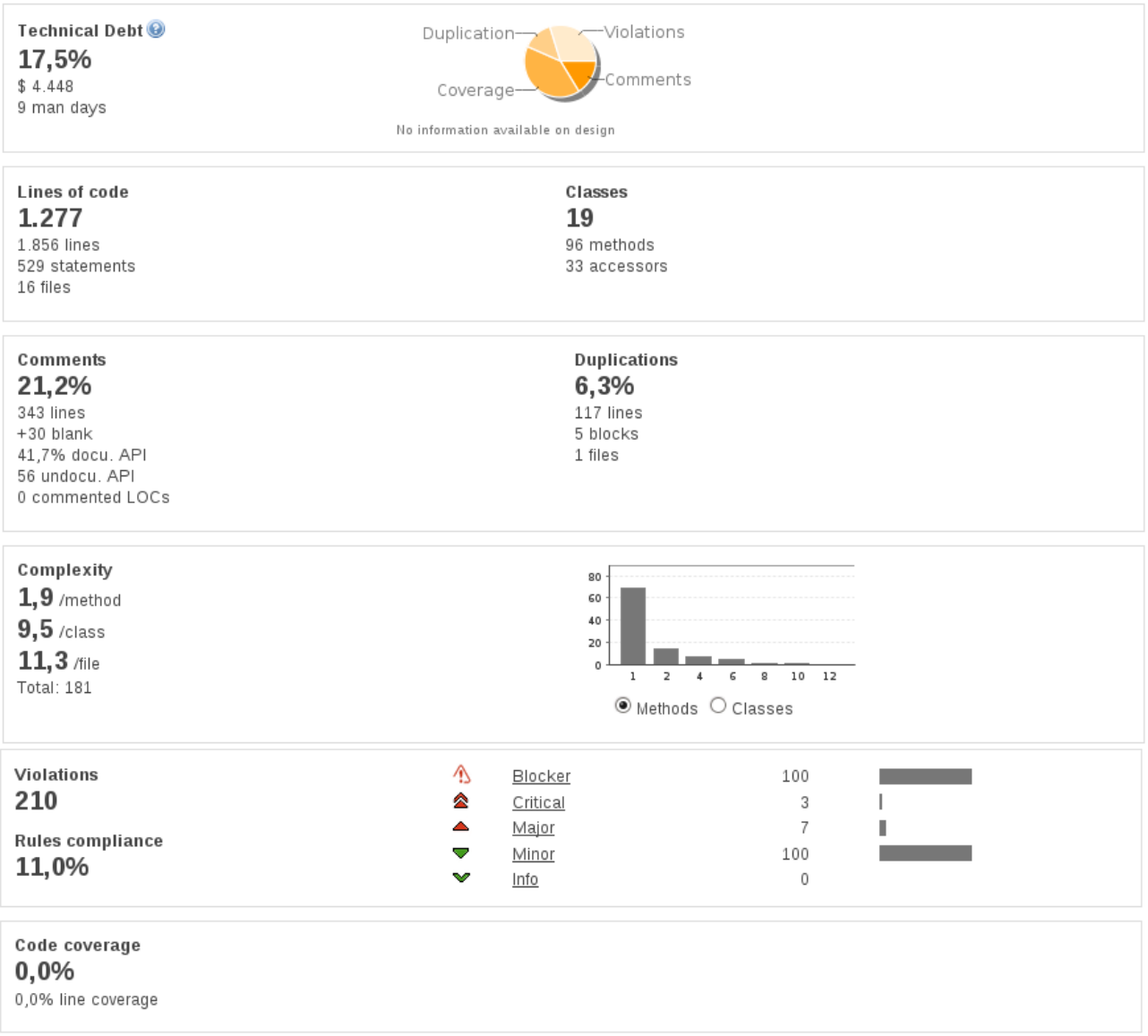

Fig. 6 Sonar results for EventManager C# module

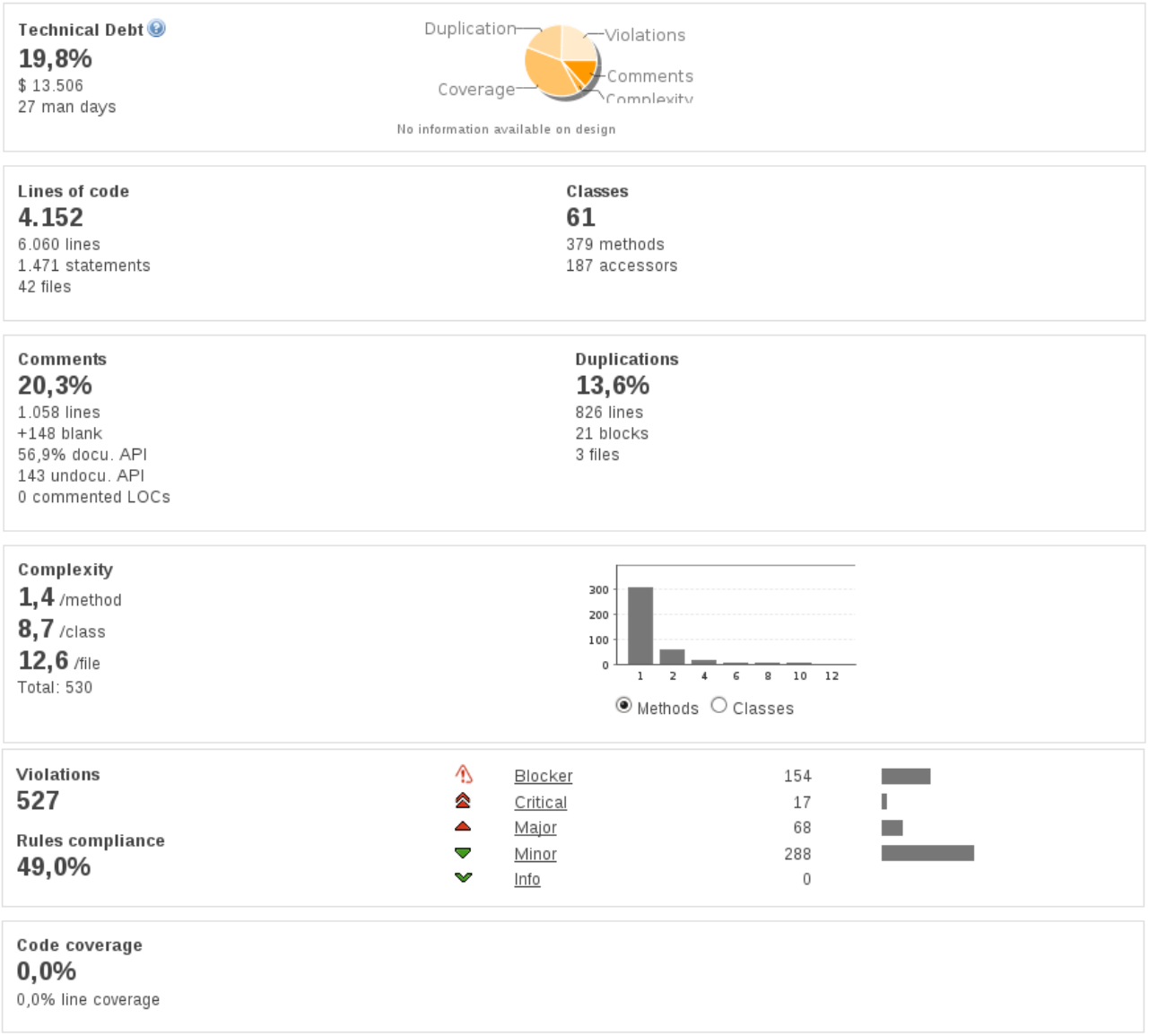

Fig. 7 Sonar results for EventProcessingAgent C# module

Fig. 8 Sonar results for LMStationProxy C# module

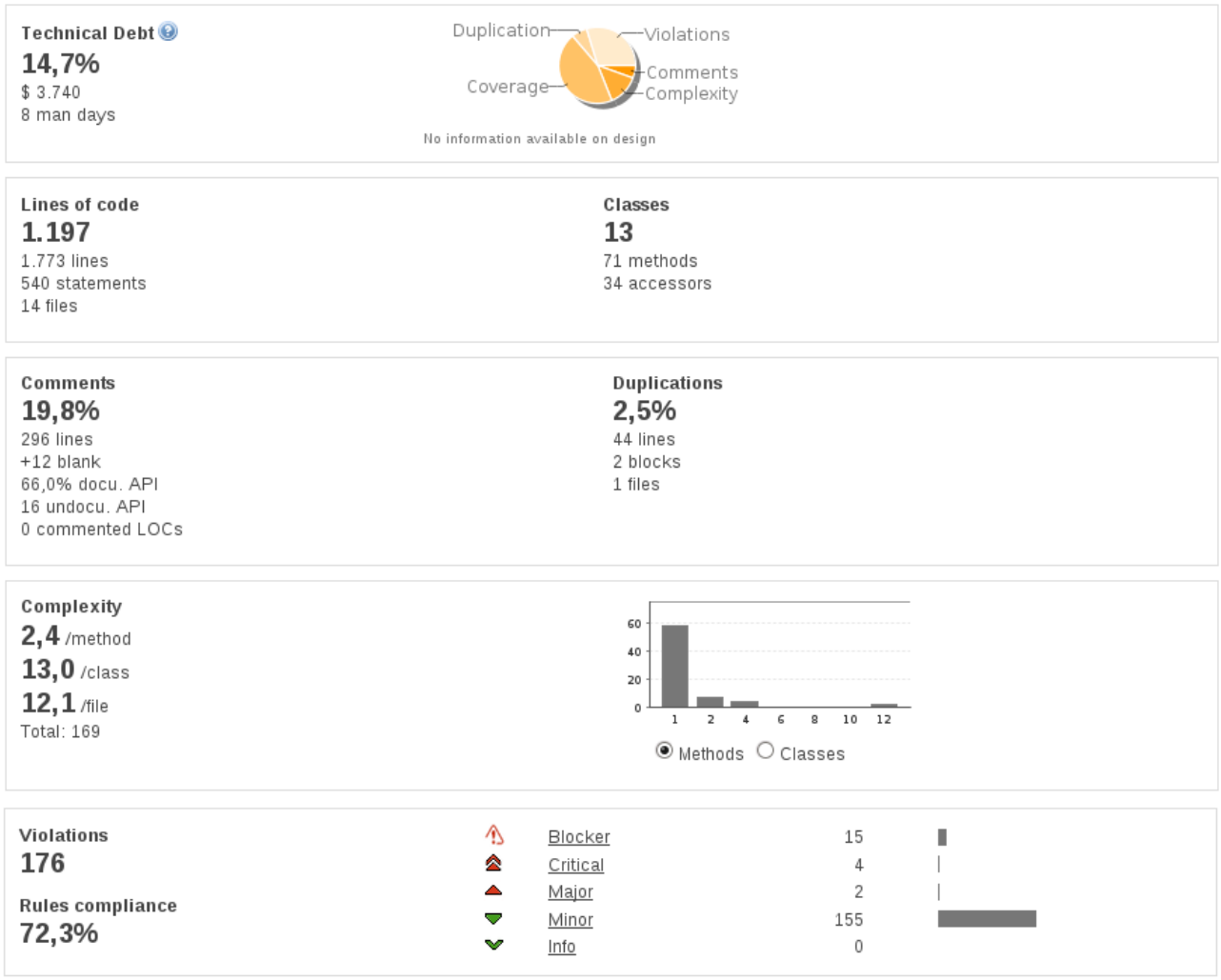

Fig. 9 Sonar results for ebbitsBRE C# module

3.4 Test Beds

As stated in D9.1 Test and integration plan we will create specific "test beds" for verifying architecture quality criteria's such as scalability etc. These "test beds" will not to be fully fledged test beds in the classical sense since this would be out of the scope of the project; instead they should focus on certain quality attributes.

In the previous iteration a test bed has been created for testing the ebbits eventing scalability and stability. The test bed consists of two main components see Figure 10 below:

- The Event Producer that creates events that either go to the LinkSmart Event Manager or directly to the ebbits Event Processing Agent (EPA).
- The Event Consumer that receives events from both the EPA and the Event Manager

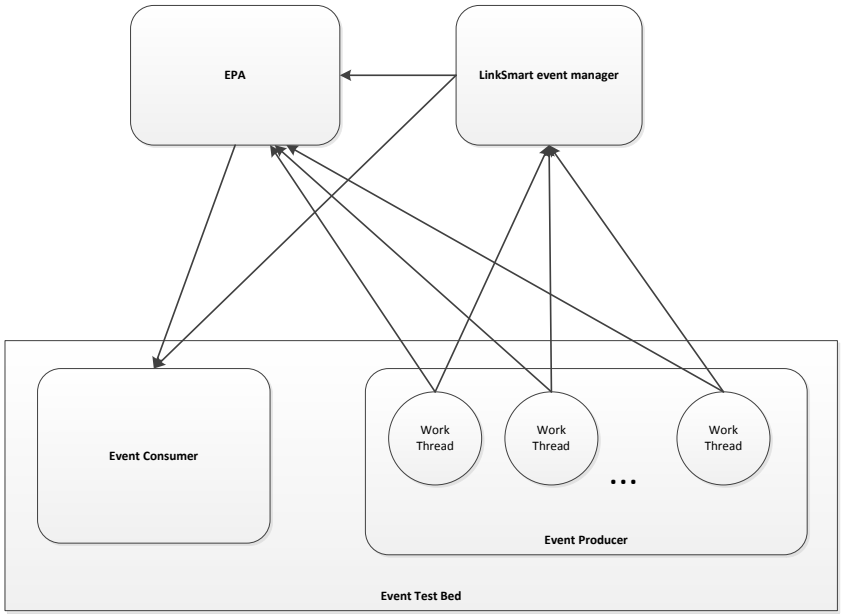

Fig. 10 Event Test bed

The main purpose of the test bed is to simulate very high event loads to be able to verify the behaviour of the ebbits eventing when it is under stress and to make sure that it does not loose events.

The basic functionality is quite simple:

1. A number of work threads that send events are started
2. At a given interval these will send a message to the LinkSmart Event Manager or to the EPA.
3. The contents of the message encode a serial # and the ID of work thread
4. The Event Consumer receives the outcome from the EPA
5. The event consumer checks the contents for the serial # and the ID to make sure that no events are missing
6. And it also measures the throughput, i.e. events per second.

The test bed is configurable with respect to the number of work threads that will be created, which events are sent.

> The intention is to evolve this test bed to a more general development support tool for testing the scalability of developed rules in the EPA so that is possible for the developer to make dry runs of event loads.

For this iteration an additional testbed has been created with the focus of manageing data transmisson in-between ebbits nodes using the EPA. The purpose of this testbed is to make sure that information can not be lost in transmission due to network problems or other malfunctions. The rationale behind this test is that we introduced the Thing concept that relies on this functionality and that, especially in any traceability application, it is very important that information can not get lost.

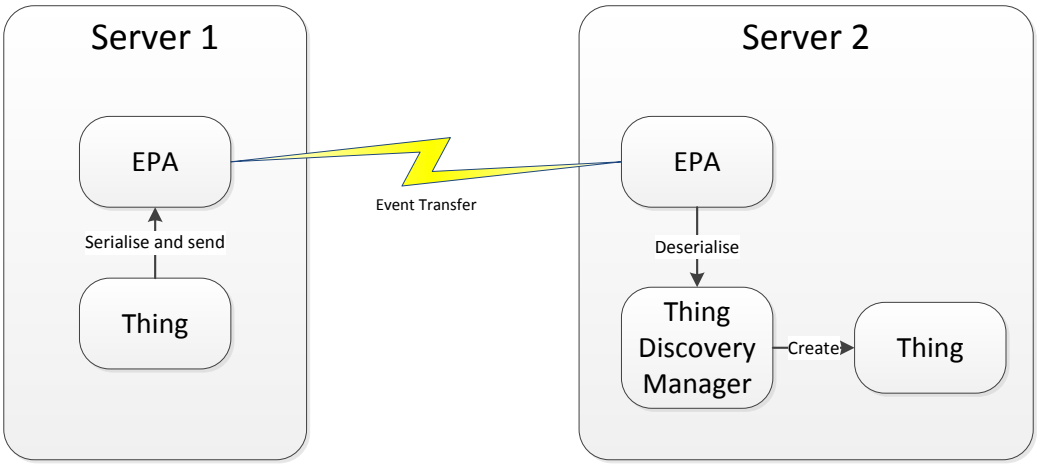

Fig. 11 Transmission

Figure 11 shows the setup of the testbed which comprises of two nodes where a message is sent from the EPA in server 1 to server 2. The basic setup is as follows:

1. A Thing serialises itself and sends the content to the local EPA
2. The EPA tries to transmit the serialised thing to the EPA in server 2
3. The communication link in-between the two servers is interrupted
    a. Either by making the server 2 unreachable
    b. Or that the EPA in server 2 is not running
    c. Or that we interrupt the transmission while it is sent (This is simulated by that the EPA in server 2 accepts a number of packets and tthen does not reply)
4. The network is established again, i.e. making it accept incoming calls again.
5. Check that the Thing Discovery Manager has deserialised the incoming data and created a new Thing at server 2 that contains the same data as the serialised one from server1

The aim of this setup is to make sure that the EPAs do not loose any information or messages sent in-between them and to verify that the EPA resend functionality works as intended.

4. Conclusion

This document is the second in a line of deliverables reporting about integration and quality assurance progress in ebbits. It also describes current test beds and modules.

The positive changes regarding the java code show on-going efforts towards better integration and quality of the code base. It also shows the significance of such reports.
The previously mentioned (D9.2.2) problems like insufficient commit activity and organic growth of the repository are less severe than last year.

The significant refactoring of the java code base improved the overall total debt of the code base. Lot of unused code was removed during last year. In total 1/3 of the linksmart code base was removed. Comments, documentation and also unit tests were added to the java code base, helping to improve the overall quality.
D9.2.1 highlited the duplication problem. Fortuantely it was tackled during the last year.
A reduction of 50% of the code duplication inside the PWAL module, increased the total health of the code base significantly.
The documentation is improving but the rate is rather low.

Contrary to the D9.2.2 the C# code base comparison with previous results was possible. The C# code base is in a good state. Compared to D9.2.2 no big changes can be highlighted. The main issue here are the low documentation rates of the code.

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.

5. References

(Nemo, Jboss) http://nemo.sonarsource.org/dashboard/index/176173

(Nemo, Apache) http://nemo.sonarsource.org/dashboard/index/Apache

(SonarQube, Commented) http://www.sonarqube.org/commented-out-code-eradication-with-sonar/

(J. Humble and D. Farley, Continuous Delivery)
Jez Humble and David Farley. 2010. Continuous Delivery: Reliable Software Releases Through Build, Test, and Deployment Automation (1st ed.). Addison-Wesley Professional.