Experiences in Coverage Testing of a Java Middleware

Mehdi Kessis
France Telecom R&D
MAPS/AMS laboratory, B.P. 98,
38243, Meylan, France
Mehdi.kessis@rd.francetelecom.com

Yves Ledru
Laboratoire LSR/IMAG
B.P. 72, 38402, St-Martin-d'Hères,
France
Yves.Ledru@imag.fr

Gérard Vandome
Bull Open Software Labs
1, rue de province, Echirolles, 38000,
Grenoble, France
Gerard.Vandome@bull.net

Abstract

This paper addresses the issues of test coverage analysis of J2EE servers. These middleware are nowadays at the core of the modern information technology's landscape. They provide enterprise applications with several non functional services such as security, persistence, transaction, messaging, etc. In several cases, J2EE servers play a critical role when applied to e-business or banking applications. Therefore, ensuring the quality of such software layers becomes an essential requirement. However, in industrial context, professional middleware software are highly complicated and have a huge size which makes their maintenance and quality management a big challenge for testers and quality managers. The aim of this paper is to present our test and coverage analysis case study with and the JOnAS J2EE server. The challenges of this work result from the size of the test suites and the size of the tested middleware (200.000 lines of code (LOC) for JOnAS)

M. Kessis, Y. Ledru, and G. Vandome. Experiences in Coverage Testing of a Java Middleware.
In Fifth Int. Workshop on Software Engineering and Middleware (SEM 2005), pages 39–45, Lisbon, September 2005. ACM Press. ISBN:1-59593-204-4

Foundations of Software Engineering
Proceedings of the 5th international workshop on Software engineering and middleware table
Lisbon, Portugal
SESSION: Testing and instrumentation
Pages: 39 - 45
Year of Publication: 2005
ISBN:1-59593-204-4
http://doi.acm.org/10.1145/1108473.1108483 "

Experiences in Coverage Testing of a Java Middleware

Mehdi Kessis
France Telecom R&D
MAPS/AMS laboratory, B.P. 98, 38243, Meylan, France
Mehdi.kessis@rd.francetelecom.com

Yves Ledru
Laboratoire LSR/IMAG
B.P. 72, 38402, St-Martin-d'Hères, France
Yves.Ledru@imag.fr

Gérard Vandome
Bull Open Software Labs
1, rue de province, Echirolles, 38000, Grenoble, France
Gerard.Vandome@bull.net

ABSTRACT

This paper addresses the issues of test coverage analysis of J2EE [22] servers. These middleware are nowadays at the core of the modern information technology's landscape. They provide enterprise applications with several non functional services such as security, persistence, transaction, messaging, etc. In several cases, J2EE servers play a critical role when applied to e-business or banking applications. Therefore, ensuring the quality of such software layers becomes an essential requirement. However, in industrial context, professional middleware software are highly complicated and have a huge size which makes their maintenance and quality management a big challenge for testers and quality managers. The aim of this paper is to present our test and coverage analysis case study with and the JOnAS [23] J2EE server. The challenges of this work result from the size of the test suites and the size of the tested middleware (200.000 lines of code (LOC) for JOnAS)

General Terms

Measurement, Documentation, Reliability, Verification.

Keywords

middleware, software engineering, J2EE, Code Coverage testing, large scale software development, JOnAS

1. INTRODUCTION

Coverage is a quality insurance metric which determines how thoroughly a test suite exercises a given program. It has been known in the software engineering community since the 1960s. In the early 1970s at IBM Rochester, Minnesota a hardware tool was developed to measure the operating systems statement and branch coverage [6]. At this time, coverage was considered complex to measure and was done with hardware tools. Today, coverage analyzers are usually user-friendly software. However, they are still not widely used in software industry.

Many works studied the coverage analysis from a theoretical point of view [5, 8, 9, 12, 15, 17, 20, 11]. However, rare are empirical studies of real industrial cases on this subject [24, 4, 7]. These works are usually judged expensive and time consuming [24]. Today, many works have proven the benefits of coverage analysis in software quality improvement [15, 18, 19, 20]. Some IBM studies studied the coverage of large scale applications in real case studies [4, 7]. The former focused on the fault distribution in large scale applications. The later proposed a coverage analysis approach based on views to monitor large scale application coverage.

The aim of this paper is to contribute to these works with a real case study of a Java middleware. The tested middleware is JOnAS, a J2EE server [23] of more than 200.000 LOC. The test suite that we used in this study is used by the JOnAS team to validate different versions of JOnAS server. It represents a real case of an industrial test suite. This test suite counts more than 2500 tests. We aim through this paper to share our experience in coverage analysis of such a large scale application with both middleware and software engineering communities.

The rest of the paper is organized as follows. In the first part of this paper we will study the code coverage feasibility in an industrial context. Then we will study the code coverage of large scale applications. After, we will introduce JOnAS J2EE server and its two test suites. Then we will present our test results and we will discuss them. Finally, in the last section we first interpret these results, and then draw some conclusions about the use of coverage in testing.

2. CODE COVERAGE

Usually, coverage analysis is used to provide quality manager with information about the portions of their code or specification which are played or not during tests. Two testing techniques are generally used; black box and white box approaches [12]. A similar distinction can be applied to coverage. In a black box approach, coverage is related to the requirements expressed on the application.

This paper addresses code coverage, which corresponds to a white box approach where the internal mechanisms of the application under test are seen by the tester. Code coverage identifies regions of code that are not adequately tested. It answers the following question: "*how much of the code and which pieces of code were exercised by played tests?*"

The answer to this question serves the following purposes: (a) to stop testing when a sufficient amount of code has been exercised [11] and (b) to monitor the quality of the tests.

The code coverage analysis process is generally divided into three tasks: code instrumentation, coverage data gathering, and coverage analysis.

- **Code instrumentation:** consists of inserting some additional code to compute coverage results. Instrumentation can be done at the source level in a separate pre-processing phase or at runtime by

instrumenting byte code (e.g., with JVMPI [1] for example).

- **Coverage data gathering:** consists of storing coverage data collected during test runtime.
- **Coverage data analysis:** consists of analysing the collected results and providing test strategy recommendations in order to reduce, to feed or to modify the relevant test suite.

In this section we will study the applicability of these three steps in industrial context and with large applications.

2.1 CODE COVERAGE OF LARGE SCALE APPLICATIONS

Despites many automation and integration efforts, code coverage analysis practices are not very popular in the industrial world. Their use is facing several challenges. First, market pressure shortens the development cycle of software. As a result, less importance and effort are dedicated to tests and quality insurance. The second challenge is the cost of the coverage analysis activity. In fact, code coverage appears as an additional expensive and non productive, task that we ask from developers and testers. It does not ensure immediate return on investment. Third, from a technical point of view, the instrumentation task is often complex to perform. Moreover, since the tests are applied to an instrumented version of the software, test engineers worry about the overhead introduced by the instrumentation, and the impact of the instrumentation on the behaviour of the program. In some cases, instrumentation even introduces new bugs. Most often, instrumentation slows down the test execution, and this overhead is in some cases unacceptable. Finally, analysing coverage data is a complex activity and often misused [8].

2.2 CODE COVERAGE FEASIBILITY IN INDUSTRIAL CONTEXT

Testing and coverage analysis of large applications is a particular case of using coverage in industrial context. This task is generally intimidating and time consuming task [7, 4]. To ensure sufficient requirement or code coverage, huge test suites are often written. In the case of application servers these tests are generally black box (functional) tests to test specification (conformance) or harness (robustness) tests which try to measure the server's performance. Measuring the scope of these tests is definitely needed to improve their quality. Although these test suites are constructed in a black box context, where their functional scope is evaluated, code coverage analysis gives a complementary view. This complementary view highlights those pieces of code that are not directly concerned with the functional requirements. This code can correspond to additional functionalities, defensive code to ensure robustness, or even dead code.

The discovery of "dead pieces of code" is a significant benefit of coverage analysers. These pieces of code are often forgotten pieces of code (automatically generated by some code generation tools, or deprecated pieces of code, etc) which are never executed. Such dead code increases the complexity of software maintenance and should be removed from the application.

[1]Java Profiling Interface URL: http://java.sun.com/j2se/1.4.2/docs/guide/jvmpi/jvmpi.html.

3. TESTING J2EE SERVERS

3.1 JONAS J2EE SERVER

JOnAS [22] is an open source implementation of the J2EE platform specification. It provides enterprise applications that it hosts with several non functional services (security, transaction, EJB, Web, naming, messaging, Web services, communication, etc). As Fig. 1 illustrates, applications are deployed in the server, then accessed through web browser clients, thin java clients or special client containers. Many services are external components to JOnAS server: Web container (Tomcat[2]), Web services implementation (Axis[3]), and JMS implementation (JORAM[4]). The total JOnAS project is composed of 1.000.000 LOC if we consider these external components. In this work we focused our analysis only on 200.000 LOC : Enterprise Java Beans (EJB) container, Client container, JMX, Web services, Resource Adapters (RA), Naming service, Security service, JDBC service. These correspond to the JOnAS source code regularly "built" by the JOnAS development team.

JOnAS supports several communication protocols (RMI-IIOP, JRRMP), the most popular database servers (Oracle, MySQL, HSQLDB, PostGreSQL) and various operating systems (Windows, Linux). This results in 16 possible configurations. To test adequately the JOnAS server, we have to test all the combinations of these parameters (Windows, Oracle, RMPI-IIOP, Linux Oracle, RMPI-IIOP; Linux, JRMP, HSQLDB,etc).

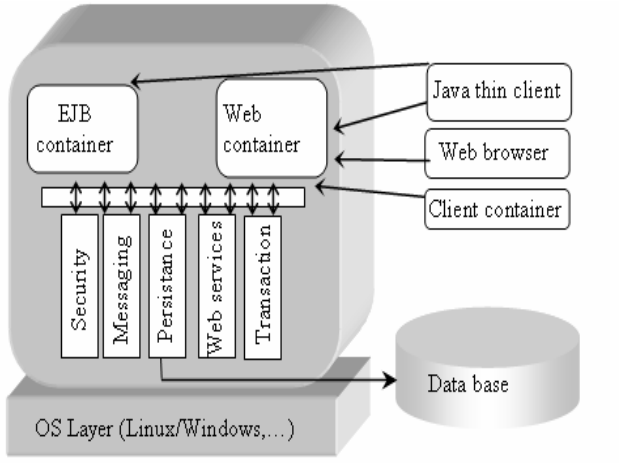

Fig. 1. JOnAS architecture

4. TEST COVERAGE OF JONAS SERVER

4.1 TEST SUITES ORGANIZATION

To evaluate JonAS's quality, we used a real world test suite named Jonas Test Suite (JTS). This test suite is composed of three types of tests: (a) functional tests, (b) structural tests and (c) performance tests.

[2]Apache Jakarta Tomcat URL: http://jakarta.apache.org/tomcat/

[3] Axis Web services URL:http://ws.apache.org/axis/

[4]Java (TM) Open Reliable Asynchronous MessagingURL: http://joram.objectweb.org/

JTS is developed by the JOnAS development team. It is regularly augmented by contributors and by final users bug reports. It is essentially a white box test suite. It counts 2.689 tests. These tests are evaluated for each combination of communication protocol, database server and operating system. Thus, the resulting size of the test suite is 43.024 tests.

Besides, JTS contains some integration tests and performance tests. Integration tests are used to validate the integration of external components (Tomcat, JORAM, Axis). Whereas, performance tests aim to evaluate the application performance in extreme conditions (e.g. high number of users over a long period of time).

4.2 Choice of the adequate coverage analyzer

pport. These automated support tools are called coverage analysers. A coverage analyser gathers data from an executing application. Several coverage analysis tools are proposed as commercial or open-source products. They propose to testers various coverage metrics (statement coverage, loop coverage, branch coverage, data flow coverage, etc). [9] lists about 101 coverage metrics. The most used metrics in industry are those supported by coverage analyzers. The most powerful metric is MC/DC [5, 14, 16]. It is used in aeronautic and critical systems certification. Despites the richness of the coverage metrics state of art, coverage analyzers for Java software do not offer a large choice [10].

There used to be two major problems with coverage analyzers; (a) instrumentation and (b) integration in the build project cycle [13]. The former problem is resolved with java technology tools. Source and binary instrumentations can be easily fully automated. To solve to the later problem, the new generation of coverage analysers provides many facilities to integrate the coverage analysis process in the automated build cycle. With such functionalities instrumentation, gathering coverage data and report generation are done in a batch mode. Besides, these tools generate coverage reports automatically which shows, for example, which probes were executed and how many times.

There are many commercial code coverage analyzers for Java programs. The most popular ones are OptimizeIt[5], Jprobe[6], DevPartnerStudio[7] and Jcover[8]. Most of these tools support basic coverage metrics; statement coverage, branch coverage, condition coverage. None of them propose advanced metrics like MC/DC. Besides, many open source and academic projects emerged such as Quilt[9], EMMA[10], InsECT[11], Hansel[12], JVMDI Code Coverage Analyzer[13], Jcoverage/GPL[14], and JBlanket[15]. They often try to bring some new features that are not supported by commercial tools. But most of them are still at experimentation stage and do not scale up to handle large applications and huge test suites that can run several hours or days (Tab.1).

Tab. 1 Open source Coverage analyzers state of art

Coverage analyzer	Line coverage	Condition coverage	Function coverage	Ant integration tasks	Reporting capabilities	Instrumentation approach	Other features
Quilt[1]	✓	×	×	✓	H	B/ S	Path coverage
EMMA	✓	×	✓	✓	H/X	B	Block coverage
NoUnit	✓	×	✓	×	H/X	B	-
InsECT	✓	✓	✓	×	G	B	profiling
Hansel	✓	×	×	✓	?	S	-
JVMDI Analyser	✓	✓	×	✓	?	B	-
GroboCode Coverage	✓	?	?	✓	H	B	-
Jcoverage	✓	✓	✓	✓	H/X/ G	B	-
Jblanket	✓	×	×	✓	?	B	-

Instrumentation approach:
B : Binary
S : Source code

Coverage report:
H : HTML
X : XML
G : GUI

Our coverage measurements were done with the Clover coverage analyzer[16]. Clover is a low cost code coverage tool for Java. It is freely licensed to Open Source and academic projects. The clover analyzer gathers large coverage information from large scale applications. Besides, it makes it possible to assemble coverage files from multiple runs.

With Clover the two major problems faced with coverage analyzers are resolved. First, it automates all coverage analysis steps through the Ant tool[17]. The instrumentation problem is automatically resolved by automating this process with Ant. Second, Clover allows an easy integration of the coverage

[5] http://www.borland.com/optimizeit/code_coverage/

[6] http:// www.quest.com/jprobe/

[7] http:// www.compuware.com/products/devpartenr/studio.htm

[8] http:// www.codework.com/JCover/product.html

[9] http://quilt.sourceforge.net/

[10] http://emma.sourceforge.net/

[11] http://insectj.sourceforge.net/

[12] http://hansel.sourceforge.net/

[13] http://jvmdicover.sourceforge.net/

[14] http://jcoverage.com/products/jcoverage-gpl.html

[15] http://csdl.ics.hawaii.edu/Tools/JBlanket/

[16] Clover Coverage tool, URL:// http://www.cenqua.com/clover/

[17] Ant: the most popular build tool for java programs [URL: http:// ant.apache.org/]

analysis process with the JOnAS build process. In fact, the JOnAS server build process is also based on the use of Ant. Thus, the coverage, build and test processes can all be done in batch mode (instrumentation, compilation, test, coverage gathering, reporting).

Clover analyzer supports method, statement and condition coverage. It computes a global coverage measure called "Total Percent Coverage" (TPC) based on these metrics. The TPC is calculated using this formula:

$$TPC = (C_T + C_F + C_s + MC) / (2 \times T_C + T_S + T_M)$$

C_T: conditionals evaluated true at least once
C_F: conditionals evaluated false at least once
C_S: statements covered
MC: methods entered
T_C: total number of conditionals
T_S: total number of statements
T_M: total number of methods

4.3 PUTTING IT INTO PRACTICE WITH JONAS

he JOnAS build and test processes is fully automated and done in batch mode. To integrate the coverage measurement process in this cycle it was necessary to make it automated to minimize the tester's efforts. The Clover analyzer tasks are fully automated via the Ant tool. The integration task of the coverage measure processes with the build project and test processes was relatively easy. (1) First the JOnAS sources are instrumented. (2) Then a binary version of the instrumented code is generated. (3) Third, tests are run. During test runtime, coverage data are gathered and stored (4). Finally coverage reports are generated based on stored coverage data (5). All coverage results were stored in XML files and were accessed through GUI and web browser. Fig.2 illustrates the coverage analysis process integration into the JOnAS build project.

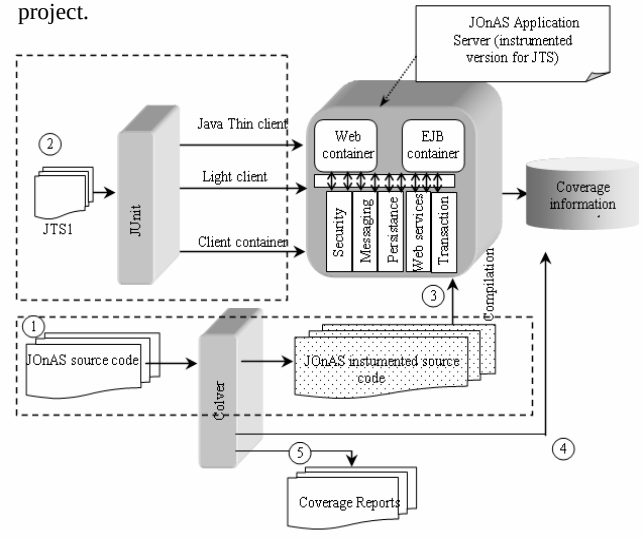

Fig. 2. Coverage process integrated into the build and test processes

5. COVERAGE MEASUREMENTS RESULTS

Following our measurements with JTS, we have noticed a 32.4 % of TPC (31.7% of conditions coverage, 32.4 % of statement coverage and 33.7% of methods coverage). Fig 3 illustrates the Clover main report of the total coverage rate that we reached.

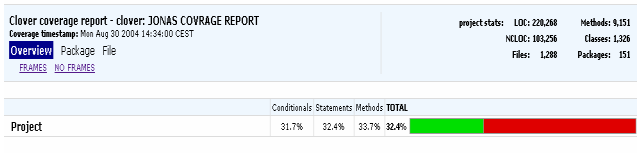

Fig. 3. Total Percent Coverage (TPC) for JTS

We then analysed the coverage distribution between JOnAS packages. The primary analysis led us to notice that coverage distribution is not balanced. Fig.4 illustrates the code coverage distribution over JOnAS server services. With JTS two services are covered at more than 50% (Web, Webservices), three at about 20 % (JDBC, security, naming) and only one package at less than 10% (Client container).

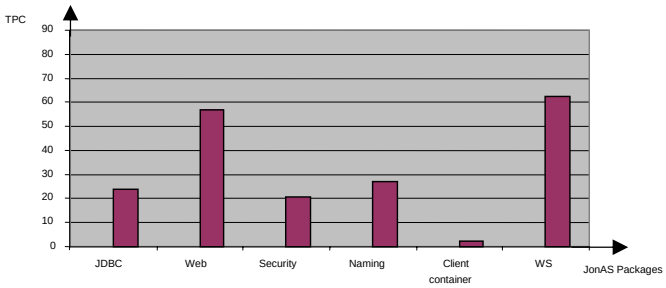

Fig. 4. Total Percent Coverage (TPC) for JTS

To understand the origin of these results we first started by looking at condition coverage and line coverage. Tab.2 illustrates the major results gathered following this analysis. According to these results, we noticed that conditions and line coverage are almost the same. We than adopted a second approach to analyse our results.

We focused on the coverage distribution. We noticed due to this second analysis that the coverage distribution is not balanced only between packages. It is also none balanced inside packages themselves.

Following to this constatation, we distinguished three sets of covered regions. The first set (Fig. 5) groups services fully or largely covered by JTS. Fig 4 illustrates the clover report for these parts of JOnAS. Green/light-grey charts correspond to covered packages and red/dark-grey ones correspond to non tested packages.

This region of code constitutes the core of the JOnAS server. It is essentially composed of core packages such as EJB, security or Web services. It includes the EJB container, the security service or the Web services.

Table 1.Line and condition coverage of JTS

J2EE services	Condition coverage	Line coverage
JDBC	25.93%	25.78%
Web	43.56%	42.95%
EJB	48.20%	48.65%
RA	27.71%	27.17%
Security	11.82%	11.76%
Naming	39.50%	37.13%
Client	0.43%	0.45%
JMX	27.93%	26.07%
WS	52.75%	52.76%

The second part of the report was composed of packages that were never tested. These packages are presented in Fig.6 with red charts and 0 % of code coverage. The non covered packages are composed of administration tool, EJB development assistance tools, etc. These components are not supported by the J2EE specification and are additional functionalities of JOnAS server.

Amongst non covered packages and classes we found some deprecated parts of code which were not and will never be used.

The third and the last part of the report contains packages partially tested (Fig.7). The partially covered packages contain non compliant code that means code which is not mentioned by the J2EEs specification and was less extensively tested than the core of the specification. They contain some JMX monitoring code, or some debug mode code, or not tested exceptions.

6. DISCUSSION

The originality of this work consists on combining three test techniques to evaluate middleware code complexity; (a) coverage testing, (b) black box testing and (c) white box testing. With white box testing we discovered that we do not cover some part of the code. Due to this measure we noticed that client container is not sufficiently evaluated with the JTS_1. Only 2.27% of the code of this package was tested. The second test technique we adopted was black box technique. Black box tests does not cover non compliant region of code. That means it does not concern parts of code that are not mentioned in the J2EE specification. Finally, the third technique (coverage testing) permitted to get a global and detailed view about the scope of our tests. The Fig.8 illustrates the distribution map of the JOnAS test suite code coverage.

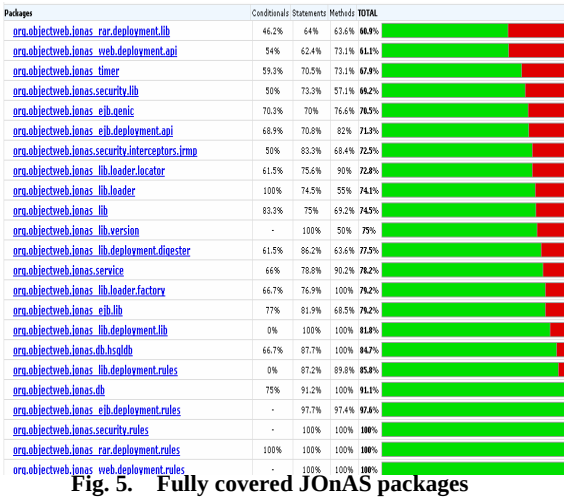

Fig. 5. Fully covered JOnAS packages

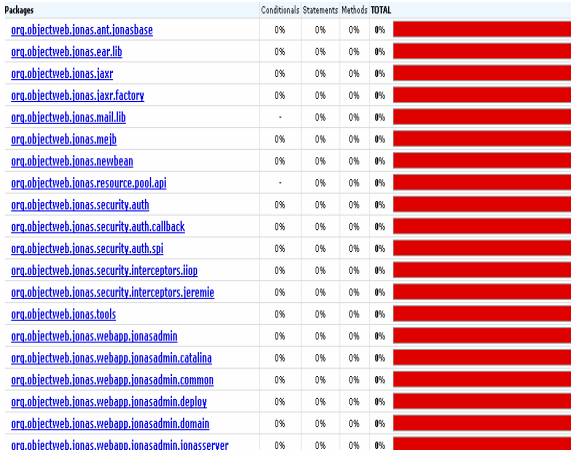

Fig. 6. Non covered JOnAS packages

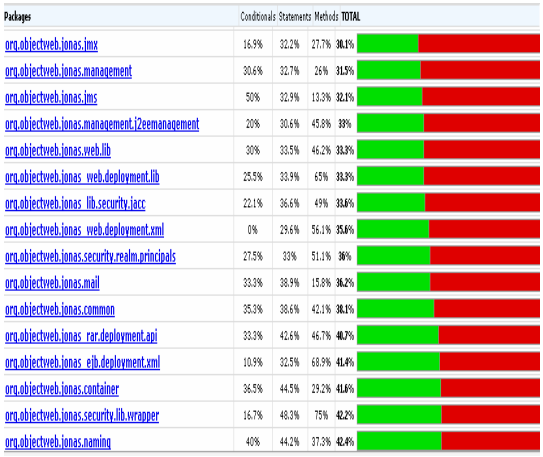

Fig. 7. Partially covered JOnAS packages

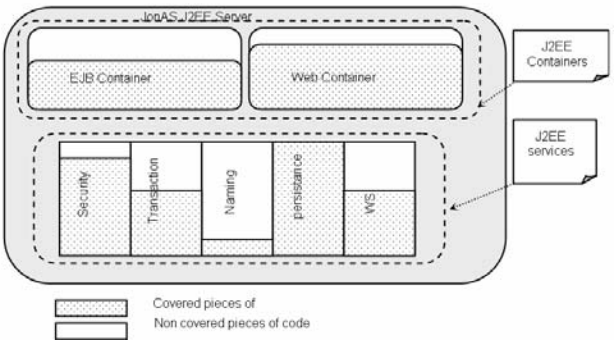

Fig. 8. Coverage distribution between JOnAS services

This map illustrates the coverage distribution of tests over JOnAS services. Each service is an implementation of a particular J2EE technology specification. In practice we can apply a selective coverage instrumentation and analysis to each of these services. That means that we instrument and we analyse the coverage of only the most critical packages and services. In our case these packages could be EJB container or security services. The selective instrumentation reduces the coverage testing effort expenses and permits us to save time and interpretation effort.

Following our study we noticed that reaching 100% of code coverage in large scale applications is nearly impossible even with the simplest coverage metric (line coverage). In fact, in practice, some parts of code concern debug mode, exception, monitoring possibility, require supplementary test efforts. A similar experience with testing java middleware called these areas "simple cases" [3]. Generally, developers judge these parts not worth to test. Testing such pieces of code can increase radically the cost of the maintenance phase and it is not recommended to adopt it as an initial goal.

7. RELATED WORKS

The principal learned lessons from [4] work are (a) there is a strong correlation between complexity and the module's size, (b) there is a positive correlation between coarse and detailed coverage and (c) the most important result is that approximatively 70 % of defects are coming from only 20 % of modules.

Asaf et al. [7] defines a coverage analysis approach based on views to monitor large scale application coverage. This work proposes a method for defining views onto the coverage data of cross-product functional coverage models. This approach defines coverage views based on selection, projection, and partition operations. The proposed method allows users to focus on certain aspects of the coverage data to extract relevant, useful information. These two studies recommend adopting a selective coverage analysis approach rather than a detailed one.

Cornett [21] studied the advantages and the weakness of use of the most common coverage measures with Java, C and C++ programs. It recommended combining using weaker measures for intermediate goal and stronger measures for release goal.

During our experiments, we noticed that in the case of middleware or large and complicated applications, the instrumentation is really a fastidious task. Installing correctly all test mechanism and coverage is often very intimidating and expensive. We think that combining advanced metrics and basics ones [21] and adopting a selective instrumentation and coverage measure politic [4, 7] would decrease radically the cost of this activity.

8. ONCLUSION AND FUTURE WORK

We had presented in this paper an empirical study of the usability of coverage analysis with a java middleware. Rare are papers that study large and real applications tests or coverage. In our paper, three test techniques were applied (black box/white box/ coverage testing) to study the quality and the usability of these approaches in an industrial context. The main lessons learned from our work are:

(1) Today Coverage measure can be easily integrated in automated build project process (Automated tools, user friendly interfaces, ease of integration in project's build process, low overhead, etc);
(2) Analysis task needs to be automated especially with large scale applications;
(3) Coverage measure of large scale applications can highlight non-covered code in the program, and can reveal the existing dead peace of code; In general this code can be deprecated peaces of code or the result of design mistakes;
(4) Coverage analyzer for Java programs are still not enough mature. There is no coverage analyzer for java programs that support advanced coverage metrics ;
(5) Middleware are complex program and as there is a strong correlation between code complexity and bugs [4], we recommend combining different test techniques to ensure a better quality.

After this study, we are still working on the improvement of the JOnAS server and the quality of its tests. The first improvement consists of eliminating the dead code from the JOnAS sources. Due to our white box test approach some supplementary tests will be written to cover not sufficiently tested regions. We learned from our experience that fixing as intermediate goal 100% code coverage, even if we use a basic coverage metrics, can impede developpement and test productivity. Besides, the complexity became more important when using advanced measures. [21] Explains well how much coverage testing become expensive and needs massive effort when testing Java, C and C++ programs with advanced metrics. Tse et al. [1] presented a non successful example of a coverage testing of a middleware with a full path metric. Developing research activity about new metrics adapted to the specific need of middleware testing could be a possible solution for this problem [2].

9. ACKNOWLEDGMENTS

We would like to thank INRIA Rhônes Alpes and ObjectWeb consortium, Bull Open Software Labs and the JOnAS team for their valuable comments and help during this study.

10. REFERENCES

[1] T.H Tse, S. S. Yau, W.K. Chan and H. Lu, "Testing Context-Sensitive Middleware-Based Software Applications", Proceedings of the 28th Annual International Computer Software and Applications

Conference (COMPSAC 2004), Los Alamitos, California (2004).

[2] S. Ghosh and A.P. Mathur, "Certification of Distributed Component Computing Middleware and Applications", Proceedings of the 4th CBSE workshop during ICSE 2001.

[3] B. Long and P. Strooper, "A Case Study in Testing Distributed Systems", Proceedings of the Third International Symposium on Distributed Objects and Applications, pp 20, 2001.

[4] Y. Woo Kim, "Efficient Use of Code Coverage in Large-Scale Software Development", IBM Center for Advanced Studies Conference, Proceedings of the 2003 Conference of the Centre for Advanced Studies on Collaborative research**,** Toronto, Ontario, Canada, pp: 145 – 155, 2003.

[5] J. Joseph Chilenski and Steven P. Miller, "Applicability of Modified Condition/Decision Coverage to Software Testing", Software Engineering Journal, Vol.9, N°5, pp: 193 – 200, Sept 1994.

[6] P. Piwowarski, M. Ohba and J. Caruso, "Coverage Measurement Experience During Function Test", Proceedings of the 15th International Conference on Software Engineering**,** Baltimore, Maryland, United States, pp: 287 – 301, 1993.

[7] S. Asaf, E. Marcus and A. Ziv, "Defining coverage views to improve functional coverage analysis", Proceedings of the 41st annual Conference on Design automation**,** San Diego, CA, USA, pp: 41 – 44, 2004.

[8] B. Marick, "How to miss use Code Coverage", 1997, available on the Web at URL: http://www.testing.com/writings/coverage.pdf.

[9] C. Kaner, "Software Negligence and Testing Coverage", Proceedings of STAR 96, 5th International Conference on Software Testing, Analysis, and Review, Software Quality Engineering, Orlando FL, 1996.

[10] C. Gaffney, C. Trefftz and P. Jorgensen, "Tools for coverage testing: necessary but not sufficient**",** Journal of Computing Sciences in Colleges, Volume 20, Issue 1, pp: 27 – 33, Oct 2004.

[11] S.R. Dalal and C.L. Mallows, "When should one stop testing software?", J. Amer. Sratistical Assoc, Vol 83, 1988 Sept, pp 872-879.

[12] G. J. Myers, "The Art of Software Testing", Wiley, John & Sons Edition, Mar 1979.

[13] B. Marick, J. Bach and C. Kaner, "A Manager's Guide to Evaluating Test Suites", International Software Quality week, San Francisco, CA, June, 2000.

[14] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson, "A Practical Tutorial on Modified Condition/Decision Coverage", Report NASA/TM-2001-210876, NASA, USA, May 2001.

[15] Y.K. Malaiya and J. Denton, "Estimating Defect Density Using Test Coverage", Rapport Technique CS-98-104, Colorado State University, 1998.

[16] K.J. Hayhurst, C.A. Dorsey, J. C. Knight, N.G. Leveson and G.F. McCormick, "Streamlining Software Aspects of Certification: Report on the SSAC Survey", Report NASA/TM-1999-209519, Aug 1999.

[17] B. Marick, "Experience with the Cost of Different Coverage Goals for Testing", 23rd Annual Pacific Northwest Software Quality Conference (PNSQC), Portland, Oregon, Oct 10-12, 2005.

[18] S. Brown, A. Mitchell and James F. Power, "A Coverage Analysis of Java Benchmark Suites", the IASTED International Conference on Software Engineering, Innsbruck, Austria, Feb 15 – 17, 2005.

[19] M.R. Lyu, J.R. Horgan and S. London, "A Coverage Analysis Tool for the Effectiveness of Software Testing", IEEE Transactions on Reliability, Vol. 43, N° 4, Dec 1994.

[20] D.S. Rosenblum, E.J. Weyuker, "Using Coverage Information to Predict the Cost-Effectiveness of Regression Testing Strategies", IEEE Transactions on Software Engineering, Vol.23, N°3, March 1997.

[21] S. Cornett "Code Coverage Analysis", available on the web at: URL: http://www.bullseye.com/coverage.html.

[22] Java 2 Platform, Enterprise Edition (J2EE™) 1.4. Specification, available on the web at: URL: http://java.sun.com/j2ee/.

[23] JOnAS: Java™ Open Application Server documentation, available on the web at: URL: http://jonas.objectweb.org/.

[24] T.J. Ostrand and E.J. Weyuker, "The distribution of faults in a large industrial software system", International Symposium on Software Testing and Analysis archive, Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis, Roma, Italy, pp: 55 – 64, 2002.