Continuous Inspection

A Pattern for Keeping your Code Healthy and Aligned to the Architecture

Paulo Merson[1], Joseph Yoder[2],

Eduardo Guerra[3], Ademar Aguiar[4]

[1]Federal Court of Accounts (TCU), Brasilia, Brazil

[2]The Refactory, Inc.

[3]National Institute for Space Research (INPE) - Brazil

[4]FEUP.

pmerson@acm.org, joe@refactory.com,
guerraem@gmail.com, ademar.aguiar@fe.up.pt

Abstract. *Agile software development methodologies are primarily an iterative and incremental development process usually done in short sprints allowing the requirements and software to evolve based upon core business needs. This is done through a close collaboration between self-organizing and cross-functional teams. Larger systems can evolve with business needs over many months and years including enterprise architecture. When this happens, it is important to make sure the core principles of the architecture are maintained or else the system can evolve to something that can be hard to maintain. This paper examines the Continuous Inspection pattern that can help insure that as the systems evolves using an agile development process, new code can evolve to still map well to the expected architecture.*

Categories and Subject Descriptors
D.1.5 [**Programming Techniques**]: Object-oriented Programming; D.2.2 **[Design Tools and Techniques**]: Object-oriented design methods; D.2.11 **[Software Architectures]**: Patterns

General Terms
Architecture, Design

Keywords
Continuous Integration, Code Analysis, Static Analysis, Architecture Conformance

Introduction

Agile software development is an iterative and incremental development process, where the software evolves and adapts to changing requirements through self-organizing and cross-functional teams. Most agile processes embrace quick responses to change through feedback loops and short sprints with flexible planning and incremental delivery. Quite often the software development and maintenance effort involves many programmers and can span many months or years. When this happens, a common phenomenon can take place: the architecture and source code can evolve to something that can be hard to maintain and evolve. Continuous inspection helps with the evolution of a system by spotting problems early in the process. It helps to ensure that new code complies with the intended architecture and other design restrictions and coding guidelines in place.

This paper describes the continuous inspection pattern, which consists of best practices to preserve the quality of your source code and its alignment to the architecture in an agile environment. The intent of this paper is to document the main pattern that captures the overall practice of continuous inspection. There are two key aspects of continuous inspection—inspection moment and inspection type—as presented in Figure 1. Each aspect gives you alternatives that are discussed throughout the paper. Specialized patterns inside the Continuous Inspection domain stem from combinations of the alternatives for these two aspects. These Continuous Inspection specialized patterns represent recurrent solutions that can be used to address more specific needs and should be explored in future work.

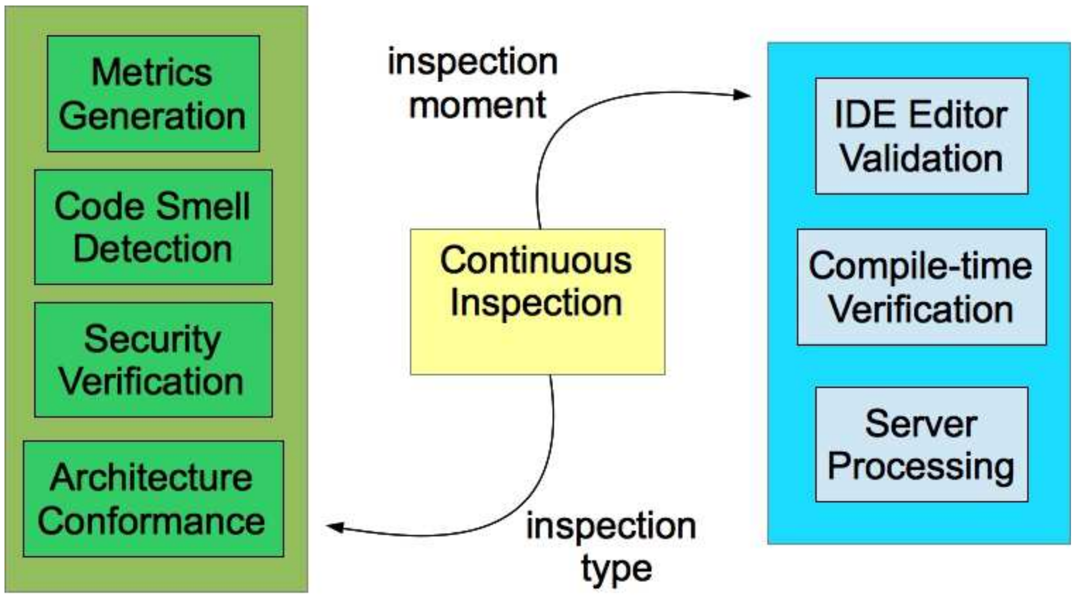


Figure 1 – Continuous Inspection aspects that engender specialized patterns

Context

A core agile development principle is that the team should respond to change. Changing the software with agility relies heavily on the level of modifiability of the code. That is why agile development uses techniques to continuously improve the software quality. Continuous care for the quality of implementation artefacts is important to reduce the impact of the risk of changing code, allowing incremental evolution in short iterations. Some important characteristics of agile software development that allow us to change code more easily and safely are:

- The presence of automated tests to allow changes to be performed safely;
- Good modularization to avoid changes to have broad impact; and
- Clean code [Mar08] that is easy to understand and hence easier to maintain and evolve.

During the iterations, the code grows and becomes hard to manage. It can be hard to know if an architectural rule was violated or if the code needs refactoring in some place. If things get out of control, the code structure can deteriorate and generate significant technical debt.

When the software development and maintenance effort involves several programmers and/or several teams, and spans months or years, a common phenomenon takes place: the source code exhibits an actual architecture that gradually diverges from the intended[1] architecture created to address the functional and quality attribute requirements of the application. The reasons for that may include:

- The intended architecture is not properly communicated to everybody who’s writing code.
- There’s high turnover in the team, and newcomers are not familiar with the architecture.
- Developers are assigned maintenance tasks and take shortcuts in the code, disregarding the now-forgotten intended architecture.
- Developers of large systems focus on small tasks, and don’t see the big picture of the system.
- The initial architecture should be evolved because of new requirements or technology innovations.

Problem

How to detect architecture and code problems as soon as possible?

Forces

- ***Awareness***: When developers focus on introducing new features, it is common to ignore existing design decisions. This is especially true if a developer that is making the change is not aware of the details of the architecture (new person).
- ***Detectability***: In a large project, sometimes it is hard to evaluate code quality and if the architecture is being followed throughout the source code. The code may grow to a very large code base using different technologies, thus making it difficult to evaluate and detect violations in the architecture.
- ***Early Diagnosis***: problems in the architecture are easier to be addressed when they are detected early. One reason for that is the fact that architecture issues that transpire to the source code tend to propagate due to copy and paste programming. However, it is hard to identify an architecture problem before the implementation is

[1] We use “intended architecture” to refer to the design specifications carefully created based on previous experience of the designers as well as knowledge codified as architecture and design patterns. The intended architecture also evolves and changes to accommodate new requirements, design patterns, technologies and frameworks. This evolution of the architecture is natural and welcomed.

already in place. In fact, some architecture issues that affect throughput and reliability are often only spotted once the application is rolled out.

- ***Software Qualities***: On the one hand, we would like to ensure that the quality attributes (e.g., modifiability, performance, portability) resulting from design decisions in the architecture are preserved in the implementation; on the other hand, as new requirements come in, it is important to adapt the software to implement them with agility. Many teams focus on addressing new functional requirements and often overlook software qualities.
- ***Architecture Evolution***: the original architecture may need to change because of new requirements (e.g., ‘the application shall provide a native Android front-end’) or adoption of technology innovations (e.g., adopting the Spring Security Framework for web authentication/authorization). How can we safely evolve the architecture keeping things working and staying focused on the top priority requirements? It is important that as the system evolves, that the architecture evolves as well.
- ***Technical Debt***: as changes are introduced into the system, it is normal to have some technical debt. If the debt goes unchecked and is never paid back, the software can become muddy [FY00] and the actual architecture of the existing implementation may not exhibit the desired modularity, interoperability, portability, and other qualities. In particular, any maintenance becomes a costly and risky task.
- ***Inspection Cost***: To manually inspect code and detect problems, specifically with the architecture, can be very difficult and time consuming. This kind of inspection requires an experienced person with good knowledge of the architecture and coding guidelines, and can require a lot of time. It is useful if these can be automated, however automated tools often require an experience person to describe the important rules for the architecture.
- ***Time***: Many times lack of code quality or problems on the application architecture are identified, but the team does not have time or is not given time to stop adding features to correct them. Agile teams in general focus on features based upon the backlog items. If time is not allocated to refactor the code and to keep things clean, the system can erode to something muddy [FY00].

Solution

Use available automated tools to continuously inspect code, generate a report on the overall code health, and point out if any violation was detected. These tools can be execute locally on the developer’s machine alone and by having the system communicate with a continuous integration server that builds the code at specific time intervals, or upon each code commit. This solution does require describing the intended architecture as part of the process where an expert is maintaining the rules for the desired architecture. Figure 2 represents the pattern main idea.

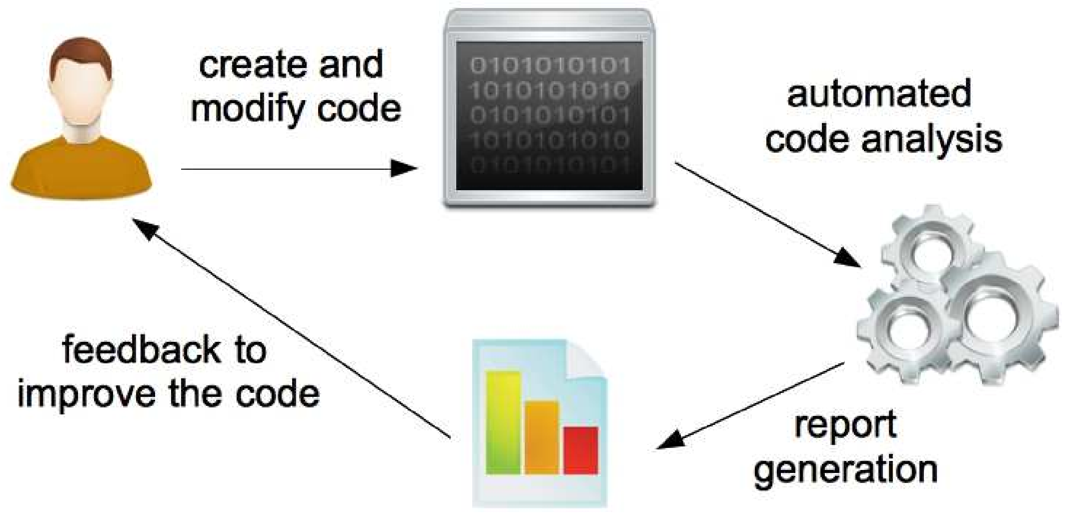


Figure 2 – Continuous Inspection pattern representation

There are some alternatives about how the tool should handle the violation found. The less invasive approach is to create a report and make it available to the entire team. Another alternative is to generate warnings that are presented each time the project is build. Finally, when the violation is severe, it can generate a build error.

Some teams adopts a zero-warning policy, that is, all violations should be fixed for the code to be considered complete. If warnings are accepted, they tend to grow in number and eventually become ignored by the team. Such policy may sound too strict or unattainable. In practice, the different verifications can be classified in categories (e.g., show-stopper, critical, major, minor), so the zero-warning policy can be adopted for the most important categories only.

The continuous integration server has access to the application source code and all the resources necessary to build it. As part of the build, some tools that perform static analysis are invoked to evaluate the quality and other aspects of the source code. Following are some types of tools that can be executed at this point:

- Metrics: One of the most common analysis tool is the one that extract metrics from the source code. This metrics can also be compared to statistical thresholds that indicate if the measured value is low, average or high. The average value per package or module can help to identify areas or layers that require more attention. This kind of analysis is usually presented on reports accessible to all in the team. Examples of metrics include:
    - Number of violations per category and ratio per KLOC.
    - Tests coverage for either or both unit and integration tests.
    - Size measures: LOC, number of classes, methods, etc.
    - Number of cyclic dependencies.
    - Cyclomatic complexity.
    - Technical debt.
- Code Smells Detection: The combination of some issues in a class can reveal some design disharmonies, also known as code smells [Fow99]. Usually

automated code smells detection is based on rules executed through static code analysis and based on parameterized metric values. For example, one can configure the number of lines of code (or number of instructions) that indicates the presence of the *long method* smell. The result of this type of analysis is usually presented in a report or is displayed as warnings during the build phase.

- Application security checks: some tools specialize in detecting security vulnerabilities in application programs. Examples include vulnerability to SQL injection and cross-site scripting, hard-coded passwords, overridden security critical methods.
- Architectural conformance: here the tool inspects the source code to make sure specific design rules are followed. These design rules typically include the allowed dependencies in the layered architecture—for example, a module in the business logic layer cannot depend on a module in the presentation layer.

The ideal scenario is to have these tools integrated in the developer IDE as well as with the build server. But when the integration is not implemented, it is still possible to use the tools for Continuous Inspection. When that happens, they should be executed continuously to give continuous feedback for the development team. Drawbacks of this approach are that the tool execution can consume time from team members and, since its execution is not integrated, it can be forgotten when time is pressing.

The continuous inspection pattern comprises the application of the four kinds of tools and analyses listed above. These tools can be employed in isolation or together and each category of verification they perform can be seen as a pattern on its own.

How It Works

The following outlines the basic steps to adopt the continuous inspection pattern:

1. ***Tool selection.*** Evaluate and select one or more tools that can perform static analysis on your code base. Evaluation criteria include:
   a. the programming and scripting language(s) used in your software projects versus the ones supported by the tool;
   b. whether the tool provides an API for developing customized verifications;
   c. integration with your IDE;
   d. integration with your continuous integration server; and a
   e. set of built-in verifications, which should include items that are deemed more relevant for your software project.
2. ***Verification selection.*** Most tools come with a variety of built-in verifications and metrics. You need to evaluate what built-in verifications in the selected tool(s) are applicable to your software project. Criteria may include:
   a. Modifiability requirements. If you are experiencing problems with maintenance and evolution of existing projects, you should enable verifications that pinpoint duplicated code, cyclic dependencies, overly complex and/or large classes and methods, and others that will help you to improve modularity and modifiability of the code.
   b. Security requirements. There are verifications that can spot security-related issues, such as SQL injection vulnerability, hard-coded credentials, and overridden security methods.

c. Team skills.
d. Quality of existing code base. It is likely that a more strict selection of verifications can be enabled for newer projects. For older code, a strict set of verifications may generate so many violations that they might tend to be ignored. In any case, it is desirable that the same selection of verifications is applied to all projects that are under the same governance system and quality scrutiny.
e. Development is/is not outsourced. The development contractor and the contracting organizations may want to negotiate what types of violations will not be accepted in delivered code.
f. Development team is/is not collocated.
g. Greenfield development. An existing code base will often exhibit several violations and the cost to fix them may be too high. Greenfield development opens space to enable all verifications considered important and enforce them from the get-go—a zero-warning policy is more easily adopted in such scenarios.
h. Feasibility to fix violations. Verifications that generate violations in the existing code that realistically will not be fixed in the foreseeable future due to lack of resources or other reasons should not be enabled.

3. ***Tool setup.*** The selected tools with the selected verifications enabled need to be configured on a continuous integration server. Ideally the tools should run as often as possible, say every 30 minutes. But in practice, if the code base is sizeable the static analysis can easily take tens of minutes. In that case, you may want to run the checks once every night.

4. ***Accountability for violations.*** It is important that developers are accountable for violations and that fixing them become part of the development process. As mentioned before, a zero-warning policy should be in place, at least for the most critical categories of violations. Checkpoints for fixing violations should be defined. For example, at the end of each sprint or at each code release, all violations should be fixed.

5. ***Customized verifications.*** The continuous inspection can go one step further and create customized verifications using the API available in the selected tools [Mer13]. This is a non-trivial step since it typically requires expressing an architecture decision or programming rule in syntactic terms of the target programming language. Albeit challenging to create, customized verifications represent an enormous enhancement to the different aspects of code health inspection. Built-in verifications are by-design generic. On the other hand, customized verifications can be specific to a software project or organization and hence more powerful. They can deal with module, layer, and method names that are specific to a project; they can accommodate known exceptions to a given rule; they are aware of home-made libraries and wrappers.

6. ***Enforcement at commit time.*** As an ultimate barrier to avoid code violations to enter the codebase, you can configure the automated verifications to be executed upon each source code commit operation [Mer13]. Of course, you need a versioning system that provides a mechanism to run user-defined scripts that can have access to the files in a code commit operation. This kind of solution will deny the code commit operation and give the user a clear error message pointing out the specific reason. Thus, even if a developer ignores the violations displayed on the IDE and on the continuous integration dashboard and reports, these

violations won’t make it into the codebase because of the enforcement at commit time. Clearly, only violations that are critical and/or applicable to all source files in the code repository should be enabled to run on this solution.

Consequences

The following outlines trade-offs of the continuous inspection pattern:

- ✓ Some problems in the code can be detected soon, enabling their correction or a planning to address them on the next iterations.
- ✓ The team is always aware of how is the code quality, and that gives them confidence in the software solution that they are developing.
- ✓ Team members can learn with some problems detected by the tools. Indeed, the tools typically display user messages that have a good rational explanation for all verifications.
- ✓ Developers are encouraged to create good quality code, since they know that it is valued by the team. Teams tend to strive to achieve ever better quality indicators, especially if those indicators are made visible to different teams on a dashboard for various projects.
- ✓ Project managers, contracting organizations, and governance agents in general can use the metrics generated by the tools as key performance indicators to evaluate the quality of the code being delivered by the development team.
- ✓ Reports generated by (widely used and trusted) tools help to make the claim before project managers and product owners to allocate effort to perform the code refactorings required to fix violations.
- ✗ An overconfidence in the tools can make the team relaxed in other aspects of the software that are not addressed by them.
- ✗ The setup of these tools and their integration with build tools and IDEs are usually time consuming and should be included in the beginning of the project.
- ✗ If the static analysis tools are not properly configured and unnecessary verifications are enabled, there can be too many violations being reported. When continuous inspection reports or the IDE shows thousands of warnings and many of them are considered non-issues, the developer tends to ignore the entirety of the warnings.
- ✗ If the motivation and benefits of continuous inspection is not made clear to all developers, some of them may feel and complain that verifications are an annoyance that disturb and constrain his/her work and creative process.
- ✗ Often times development teams install and begin to use tools that calculate metrics on the software code, but they don’t take the time to study, explore, and configure the various metrics. For example, some tools calculate an overarching “quality index” that combines a ratio of code violations with test coverage and a measure of complexity. Some metrics are derived from other basic metrics using weighted equations. The default values used in the calculation as well as the weight of each factor may not be appropriate for a given software project, and the metrics can be skewed or misleading. For example, some tools calculate technical debt and one of the outputs is the dollar amount required to pay the debt. If the average salary of the developer was not configured in the tool, the debt amount may not be realistic.

Related Patterns

Jez Humble and Dave Farley have a blog post about continuous delivery patterns [Hum13]. Although their text talks about patterns in continuous integration, it doesn't explicitly mention code inspection or code analysis.
Paul Duvall author of "Continuous Integration" [Duv07] wrote a piece titled "Continuous Integration - Patterns and Antipatterns" [Duv10]. Duvall suggests 23 patterns that go beyond building and running tests continuously and dicuss a broad set of activities related to continuous integration, such as deployment and IDE integration. One of the patterns is "Continuous Inspection", which is described as "Run automated code analysis to find common problems. Have these tools run as part of continuous integration or periodic builds". He mentions the use of checkstyle as part of an ant build script.
Much of SonarQube documentation and also "SonarQube in Action" [Cam13] talk about "continuous inspection", which is described as a process that consists of code analysis and reporting and should be part of the development lifecycle. However, continuous inspection is not described as a pattern in these publications.

Known Uses

As an example of setup for continuous inspection, consider an IT organization that develops enterprise applications using Java technologies. They have a few million lines of Java code spread across over 50 applications, and counting. The various development teams count up to just over a hundred people, who include full-time staff, interns, and contractors.

The continuous inspection pattern was applied using the following tools:

- Static analysis tools:
  - SonarQube
  - PMD
  - Checkstyle
  - FindBugs
- Coverage:
  - JaCoCo
- Continuous integration and build:
  - Jenkins
  - Maven
- Versioning system:
  - Subversion
- IDE:
  - Eclipse with plug-ins to run the code analysis tools

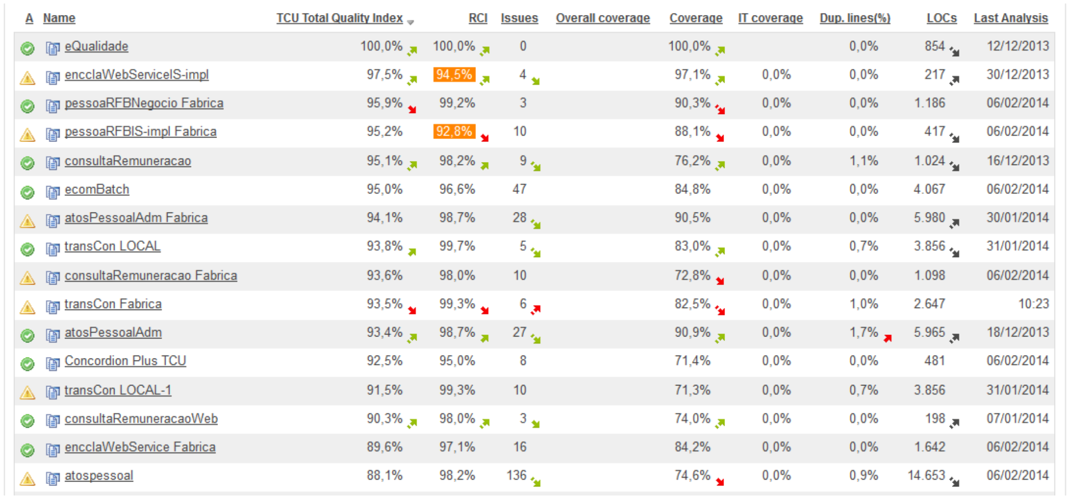
The built-in verifications provided by the static analysis tools were carefully selected and configured. A formula for a customized overall quality index was devised based on the different metrics and emphasizing the aspects that most directly affect maintainability, which is considered a key quality attribute for that organization.

Developers can execute the various tools on their IDE or even using the command line version of maven. Jobs were configured on the Jenkins continuous integration server to execute the code analyses and generate the metrics. Results of analyses are uploaded to the SonarQube server, which is at the same time a central repository for analyses and a web server that offers a web user interface to browse the metrics and violations.

Customized verifications were developed using the checkstyle Java API. They were packaged as a plug-in to the SonarQube server to enhance built-in verifications provided by that platform. In addition, some of these “custom checks” were deemed “must have” rules and were configured to run on the Subversion server upon any Java file commit, so that the entire commit operation is denied if any custom check detects a violation.

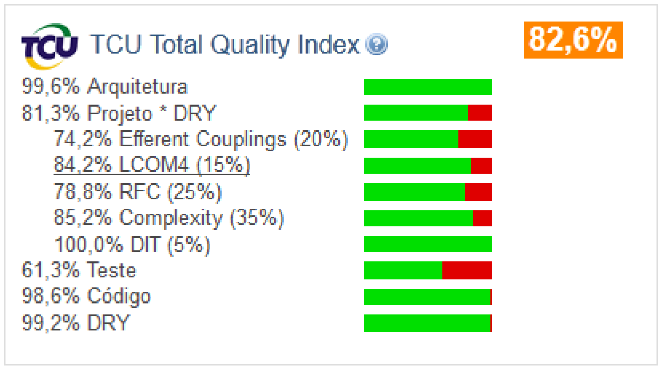
Jenkins has a dashboard view to show the situation of the various jobs. Likewise, SonarQube provides a global dashboard showing metrics for various projects (see snapshot in). The user can click on a given project to see a project specific dashboard. For this IT organization, the most prominent metrics and indicators on a project’s dashboard are: the customized quality index mentioned above (see snapshot in Figure 4), technical debt (Figure 5), unit and integration tests coverage (Figure 6), code duplication (Figure 7), and size measures (Figure 8). The SonarQube dashboard also shows a graph with the evolution of some metrics over time (Figure 9).

These dashboards are physically on display on a handful of big screen TVs spread across the software development department. They are also visible by any developer or manager on the web browser. Many of the metrics contain hyperlinks to detailed information about the calculation and annotated source code. For example, one can click on the measure under “Duplications” (Figure 7) to navigate and display the source files, and see where duplicate code is.



A	Name	TCU Total Quality Index	RCI	Issues	Overall coverage	Coverage	IT coverage	Dup. lines(%)	LOCs	Last Analysis
✓	eQualidade	100,0%	100,0%	0		100,0%		0,0%	854	12/12/2013
⚠	encclaWebServiceIS-impl	97,5%	94,5%	4		97,1%	0,0%	0,0%	217	30/12/2013
✓	pessoaRFBNegocio Fabrica	95,9%	99,2%	3		90,3%	0,0%	0,0%	1.186	06/02/2014
⚠	pessoaRFBIS-impl Fabrica	95,2%	92,8%	10		88,1%	0,0%	0,0%	417	06/02/2014
✓	consultaRemuneracao	95,1%	98,2%	9		76,2%	0,0%	1,1%	1.024	16/12/2013
✓	ecomBatch	95,0%	96,6%	47		84,8%	0,0%	0,0%	4.067	06/02/2014
⚠	atosPessoalAdm Fabrica	94,1%	98,7%	28		90,5%	0,0%	0,0%	5.980	30/01/2014
✓	transCon LOCAL	93,8%	99,7%	5		83,0%	0,0%	0,7%	3.856	31/01/2014
⚠	consultaRemuneracao Fabrica	93,6%	98,0%	10		72,8%	0,0%	0,0%	1.098	06/02/2014
⚠	transCon Fabrica	93,5%	99,3%	6		82,5%	0,0%	1,0%	2.647	10:23
✓	atosPessoalAdm	93,4%	98,7%	27		90,9%	0,0%	1,7%	5.965	18/12/2013
✓	Concordion Plus TCU	92,5%	95,0%	8		71,4%	0,0%	0,0%	481	06/02/2014
⚠	transCon LOCAL-1	91,5%	99,3%	10		71,3%	0,0%	0,7%	3.856	31/01/2014
✓	consultaRemuneracaoWeb	90,3%	98,0%	3		74,0%	0,0%	0,0%	198	07/01/2014
✓	encclaWebService Fabrica	89,6%	97,1%	16		84,2%	0,0%	0,0%	1.642	06/02/2014
⚠	atospessoal	88,1%	98,2%	136		74,6%	0,0%	0,9%	14.653	06/02/2014

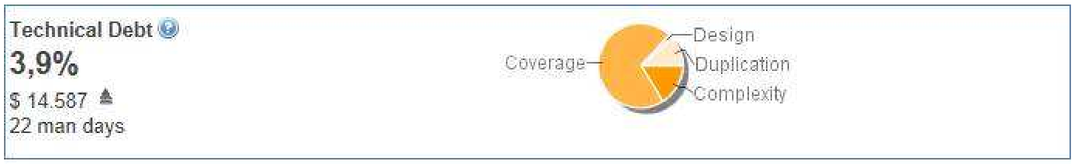
Figure 3 – Snapshot of SonarQube global dashboard showing various projects and a summary of metrics



TCU Total Quality Index 82,6%

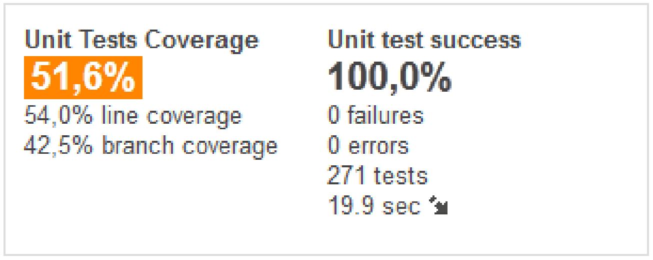
99,6% Arquitetura
81,3% Projeto * DRY
74,2% Efferent Couplings (20%)
84,2% LCOM4 (15%)
78,8% RFC (25%)
85,2% Complexity (35%)
100,0% DIT (5%)
61,3% Teste
98,6% Código
99,2% DRY

Figure 4 – Snapshot of SonarQube dashboard showing a customized quality index for a project



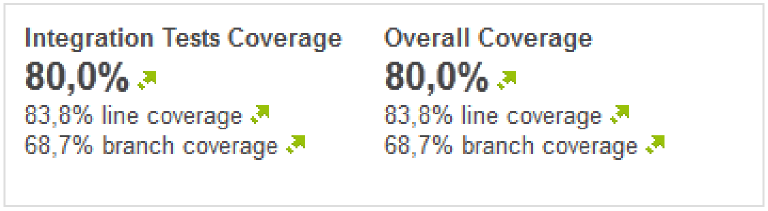
Technical Debt
3,9%
$ 14.587
22 man days

Figure 5 – Snapshot of SonarQube dashboard showing the technical debt for a project



Unit Tests Coverage
51,6%
54,0% line coverage
42,5% branch coverage

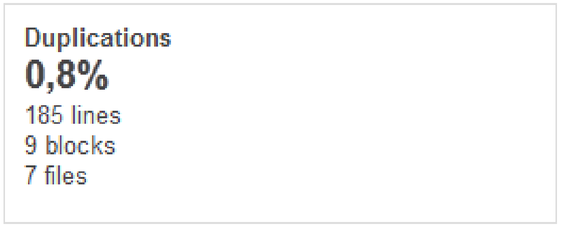
Unit test success
100,0%
0 failures
0 errors
271 tests
19.9 sec



Integration Tests Coverage
80,0%
83,8% line coverage
68,7% branch coverage

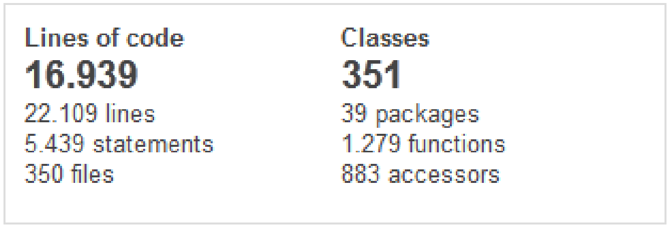
Overall Coverage
80,0%
83,8% line coverage
68,7% branch coverage

Figure 6 – Snapshot of SonarQube dashboard showing the unite test coverage (above) and integration test coverage (below) for a project



Duplications
0,8%
185 lines
9 blocks
7 files

Figure 7 – Snapshot of SonarQube dashboard showing size metrics for a project



Lines of code
16.939
22.109 lines
5.439 statements
350 files

Classes
351
39 packages
1.279 functions
883 accessors

Figure 8 – Snapshot of SonarQube dashboard showing the technical debt for a project

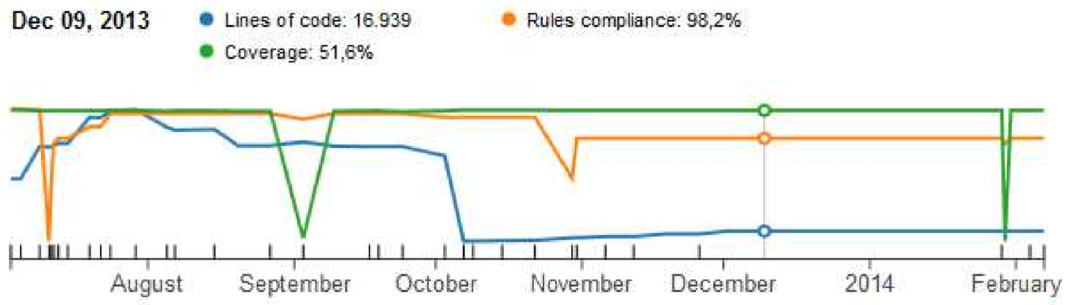


Figure 9 – Interactive graph in the SonarQube dashboard showing historical information for three different metrics.

Acknowledgements

We thank our shepherd Kazunori Sakamoto for his valuable comments and feedback during the AsianPLoP 2013 Shepherding process. We also thank our 2013 AsianPLoP Writers Workshop Group, YYY, for their valuable comments.

References

[Cam13] Campbell, G., Papapetrou, P. *SonarQube in Action*. Manning, November 2013.

[Duv07] Duval, P., Matyas, S., Glover, A. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, July 2007.

[Duv10] Duval, Paul. *Continuous Integration: Patterns and Anti-Patterns*. DZone, 2010. Available at http://refcardz.dzone.com/refcardz/continuous-integration.

[Fow99] Fowler, M., Refactoring: Improving the Design of Existing Code. Addison-Wesley. 1999.

[FPR01] Fontura, M., Pree, W., Rump, B. *The UML Profile for Framework Architectures*. Addison-Wesley. 2001.

[FY00] Foote, B., Yoder J. W. 2000. *Big Ball of Mud*, Fourth Conference on Patterns Languages of Programs (PLoP '97/EuroPLoP '97) Monticello, Illinois, September 1997. Pattern Languages of Programs Design 4 edited by Neil Harrison, Brian Foote, and Hans Rohnert. Addison Wesley, 2000.

[GHJ+95] Gamma, E., R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object Oriented Software*. Addison-Wesley. 1995.

[Hum13] Humble, J., Farley, D. *Continuous Delivery – Patterns*. Available at http://continuousdelivery.com/patterns/.

[KJ04] Kircher, M.; P. Jain. *Pattern Oriented Software Architecture, Volume 3: Patterns for Resource Management*. Wiley & Sons. 2004.

[Mar02] Martin, R. *Agile Software Development: Principles, Patterns, and Practices*. Prentice Hall, 2002.

[Mar08] Martin, R. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

[Mer13] Merson, P. *Ultimate architecture enforcement: custom checks enforced at code-commit time*. Proceedings of the 2013 Conference on Systems, programming, & applications: software for humanity (SPLASH'13). Indianapolis, October 2013.