Ultimate Architecture Enforcement

Custom checks enforced at code-commit time

Paulo Merson

Federal Court of Accounts (TCU)
Brasília, Brazil
pmerson@acm.org

Abstract

Creating a software architecture is a critical task in the development of software systems. However, the architecture discussed and carefully created is often not entirely followed in the implementation. Unless the architecture is communicated effectively to all developers, divergence between the intended architecture (created by the architect) and the actual architecture (found in the source code) tends to gradually increase. Static analysis tools, which are often used to check coding conventions and best practices, can help. However, the common use of static analysis tools for architecture enforcement has two limitations. One is the fact that design rules specific to a software architecture are not known and hence not enforced by the tool. The other limitation is more of a practical issue: static analysis tools are often integrated to the IDE or to a continuous integration environment; they report violations but the developers may choose to ignore them. This paper reports a successful experience where we addressed these two limitations for a large codebase comprising over 50 Java applications. Using a free open source tool called checkstyle and its Java API, we implemented custom checks for design constraints specified by the architecture of our software systems. In addition, we created a script that executes automatically on the Subversion software configuration management server *prior* to any code commit operation. This script runs the custom checks and denies the commit operation in case a violation is found. When that happens, the developer gets a clear error message explaining the problem. The architecture team is also notified and can proactively contact the developer to address any lack of understanding of the architecture. This experience report provides technical details of our architecture enforcement approach and recommendations to employ this or similar solutions more effectively.

Categories and Subject Descriptors D.2.11 Software Architectures; D.2.9 Management: software quality assurance.

Keywords software architecture; architecture conformance; architecture enforcement; static analysis; Java; checkstyle.

SPLASH'13, October 26–31, 2013, Indianapolis, Indiana, USA.

ACM 978-1-4503-1995-9/13/10…$15.00.

1. Introduction

Creating a software architecture is a critical task in the development of software systems because the structures in the design will dictate whether the system will exhibit good modifiability, performance, interoperability, and other qualities. Software architectures are created every day in IT departments and software companies around the world based on previous experience of the architects as well as knowledge codified as architecture patterns, design patterns, and tactics. These architectures are often discussed and reviewed with the stakeholders. Incrementally, the software architecture is handed to developers who translate the design diagrams into code. Once in place, the code becomes the main artifact of a software project ("code is king!").

In software projects of reasonable size, several people do write code. Often times, these people are geographically distributed. It's also common for new developers to join software projects when the software system is partially implemented and/or rolled out. The newcomers are readily assigned features to implement, or bugs to fix. Many times they write code before they understand the overall architecture. Developers of large software systems are like the XV century explorers who tried to "connect", say, Portugal to India, but didn't have the big picture (of Earth!) and ended up finding unexpected pathways, often not optimal.

When software development and maintenance involves several developers and spans months or years, a common phenomenon can happen: the *actual* architecture found in the source code diverges from the *intended* architecture, which was the diligent work of the architects. This problem, often introduced involuntarily by developers who write code non conformant to the architecture, is influenced by different factors:

- whether the architecture documentation is effectively communicated to developers;
- turnover among developers, since newcomers may be required to write code before they have a good understanding of the architecture;
- pressure to quickly fix bugs and deliver new features, which leads developers to take shortcuts in the code disregarding the architecture;
- size of the system (the larger the codebase, the more likely developers don't see the big picture);
- presence of sub-teams of developers, possibly outsourced; and
- degree of accountability for creating code that violates the design constraints.

The disconnect between the code and the intended architecture may also occur due to changes in the architecture itself. Long after the implementation is created, the intended architecture may be modified to tend to new requirements and technology innovations. This evolution of the architecture is natural and welcomed. However, in many cases refactoring the old code in order to comply with the new design is too expensive. The once-compliant old code is left as is and now it doesn't conform to the (new) architecture anymore.

When the discrepancy between the architecture and the source code grows uncontrolled, problems arise. First off, maintainability is impaired. The introduction of code dependencies (shortcuts) not permitted by the architecture makes the code brittle, hard to understand and to change. But consequences can go beyond that. Design decisions in the intended architecture aimed at achieving certain qualities, such as reliability, security, modifiability, performance, portability, and interoperability. If the code departs from that architecture, these qualities can be negatively affected.

This paper describes our experience creating an approach based on static analysis to watchfully check that source code is only added to the codebase if it follows the design rules and other constraints defined in the software architecture. Section 2 will discuss how to address the architecture conformance challenge in general. Section 3 briefly describes the environment where this experience took place. Section 4 is a quick introduction to the checkstyle tool and how to create custom checks using the checkstyle Java API. Section 5 gives an overview of the types of custom checks created so far in our organization. Section 6 describes how we made architecture conformance an automatically enforced requirement for source code added to our subversion repository. Section 7 describes the results and lessons learned in this experience. Section 8 provides some conclusions and final thoughts, including a discussion of the limitations of the approach.

2. How to avoid code and architecture disparity

To keep the source code compliant with the software architecture over time, there are two important things to be done: one is to properly communicate the architecture to the developers, and the other is to actively check that the source code follows the intended design.

2.1 Communicating the architecture to stakeholders

The architecture of a software system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both [2]. Describing and communicating the software architecture to all stakeholders is an encumbering task. It involves: documenting the different structures as multiple views; recording information needed by the stakeholders about the software elements and their relations; describing the relevant software interfaces; creating structural design diagrams and complementing them with behavior documentation, such as UML sequence diagrams and state-machine diagrams; recording the rationale for the design decisions; and keeping the documentation up-to-date.

Creating effective software architecture documentation is important not only to avoid that developers create code that doesn't follow the architecture. The architecture documentation is the-blueprints for creating the code, is the primary indicator of the quality attributes of the system (e.g., performance, availability, modifiability), and guides incremental development plans, allocation of tasks, procurement of software and hardware. However, the focus of this paper is not on documenting and communicating the architecture to developers—for that we refer the reader to the work of Paul Clements and colleagues [2]. This paper will explore automated architecture conformance analysis, introduced next.

2.2 Automated architecture conformance analysis

One can manually review the code to check that it conforms to the intended architecture. Indeed, architecture conformance can be one of the goals of code reviews. However, manual reviews are time consuming. Also, the reviewer is not always well versed in the architecture, and the architect is not always available to participate in code reviews. Thus, architecture conformance is more often verified using automated tools.

The tools commonly used in industry for architecture enforcement employ static program analysis. Examples include: Checkstyle, FindBugs, Fortify, JDepend, Lattix, Lint, NDepend, PMD, Sonar, and Understand [12]. Some tools give you a reverse-engineered depiction of the actual architecture found in the code, so that you can visually compare it with the intended architecture. In other tools you can even indicate that a dependency between modules A and B is not allowed, and the tool will let you know whenever that dependency rule is violated.

Adding static analysis to continuous integration is important to ensure the quality of the code remains at a good level, or at least to ensure it is not deteriorating. However, the common practice of continuous conformance verification has two limitations.

One limitation of static analysis as performed in many organizations is that it is restricted to the checks available out-of-the-box in the static analysis tools. Although the tools usually give you the ability to turn on/off, configure thresholds, and change the severity of each check, these built-in checks are always generic. They are oblivious to the architecture of your software system; they are unaware of the many design constraints, infrastructure services, and idiosyncrasies specific to your software projects.

The other limitation is the fact that violations may find their way into the codebase despite the finger pointing of the tools. Sometimes the number of violations creeps up to hundreds or thousands; sometimes there is no accountability for violations added to the codebase; sometimes management does not want to allocate time and effort to fix the violations ("after all, addressing these violations doesn't represent progress towards delivering functionality to the customers"). As a result, violations tend to be ignored. This problem can be attenuated when the static analysis tool offers a classification of the violations. In such cases, the top priority or critical violations are not acceptable, but the lower priority ones may remain unheeded.

The approach presented in this paper solves these two limitations.

3. Context of our development organization

The experience described in this paper took place at the IT department of the Brazilian Federal Court of Accounts (TCU). TCU has over 50 Java EE applications and several shared libraries developed in house. The Java codebase consists of approximately 2.2 million physical LOC spread across 15 thousand source files.

The common denominator platform for all these applications is a cluster of JBoss EAP application servers and an Oracle database. Yet, some applications access different small data repositories and interact with other software systems, both internal and external to the organization.

To a great extent, these applications are built using the same frameworks and libraries, have nearly the same quality attribute requirements, offer similar data-centric functionality, and share the same runtime environment. Because of this similarity, reference architectures [4] have been established over the years. Current reference architectures prescribe a layered architecture, MVC

and other architectural patterns, and constrained use of a few specific frameworks. Any application may specialize and deviate from the reference architecture if necessary, but otherwise design constraints are prescribed to all Java applications.

The development team consists of 57 full-time employees. Their development skills and experience vary tremendously, since the recruiting process does not require previous experience with Java or any specific technology. Part of the development team, there are also 24 interns working part-time. The internship program takes computer science undergraduate students for a period of six months to two years. The development work force will soon grow, since a contract is about to be signed for outsourcing some of the software development effort.

3.1 Architecture conformance undertaking

The small group within the software development organization responsible for the reference architectures has always strived to see the architecture being followed in the implementation of the various applications. This group has spent significant effort to document the reference software architectures and to communicate them to all developers, including new hires and interns, through presentations and one-on-one coaching.

In the past, this group resorted to source code lexical searches based on regular expressions and code reviews to spot architecture conformance problems. Success was very limited. In 2011, we set out to create a mechanism based on static analysis tools to enforce our own architecture constraints and coding guidelines. We evaluated four different static analysis tools for Java and chose checkstyle for its Java-based API for creating custom checks. This choice has showed to produce effective results at a low cost. The remaining of this paper describes some technical details, results of this experience, and lessons learned.

4. Checkstyle API

Checkstyle is a free open-source static analysis tool for Java [5]. Like other tools, out of the box it can analyze the source code for coding conventions and numerous programming best practices. Unlike most tools though, Checkstyle offers a Java API that allows the implementation of custom code analyses.

A *check* is a Java class that is invoked when Checkstyle is parsing a Java file. The check is given that file's AST and can inspect each token and look for constructs that represent a violation of some sort. Using the Checkstyle API, we can create *custom checks*. For example, let's say your architecture uses data access objects (DAO) [6] to access the database. Suppose in your architecture, a class could be recognized as a DAO class by the "Dao" prefix—other common options would be extending an abstract Dao class, or using a specific annotation, such as `@Dao` or `@Repository`. Now suppose your architecture dictates that only code in the "service" layer can use DAO classes, and classes in the service layer belong to a package namespace com.mycompany.service.*. **Figure 1** shows the code for the custom check that enforces that rule.

Checkstyle uses the Visitor design pattern [7] to invoke the checks while traversing a Java program AST. Each check is a visitor that has a single entry point method called `visitToken`. This method takes as argument the AST where the root is the token just parsed in the Java program. The `visitToken` method typically has an if-else-if or switch-case construct to identify what type of token it received and execute the part of the analysis logic that applies to that type of token. As an optimization, a check can declare which types of tokens it's interested in. The example in **Figure 1** is only interested in package definitions (`TokenTypes.PACKAGE_DEF`) and references to identifiers (`TokenTypes.IDENT`), so checkstyle only calls `visitToken` on this check when it parses tokens of these types.

```
/**
 * Classes prefixed by Dao can't be used
 * outside com.mycompany.mysystem.service.*
 */
public classCheckNonServiceUsesDao extends Check {
  private boolean inServiceLayer;
  @Override
  public int[] getDefaultTokens() {
    return new int[] {TokenTypes.PACKAGE_DEF, TokenTypes.IDENT};
  }
  @Override
  public void visitToken(DetailAST aAST) {
    if (aAST.getType() == TokenTypes.PACKAGE_DEF) {
      inServiceLayer = false;
      String packageName = fullyQualifiedPackage(aAST);
      if (packageName != null &&
          packageName.startsWith("com.mycompany.mysystem.service")) {
        inServiceLayer = true;
      }
    } else if (aAST.getType() == TokenTypes.IDENT && !inServiceLayer) {
      if (aAST.getText().startsWith("Dao")) {
        log(aAST.getLineNo(),
          "Classes outside the service layer can't call Dao classes");
      }
    }
  }
}
```

Figure 1. Example of a checkstyle custom check.

The checkstyle API provides several methods used in the implementation of custom checks, including:

- `findFirstToken`: for the current AST node, returns the first child (in pre-order) of a given token type.
- `getNextSibling`, `getPreviousSibling`, `getFirstChild`: allow navigation around the AST.
- `branchCointains`: returns true if the AST rooted at the current node contains a node of a given token type.

We have enhanced the API by creating a few other methods, which have been submitted as contributions to the checkstyle sourceforge project [5]. Two of the API methods we created are:

- `findFirstAstOfType`: returns the first node (in pre-order) within an AST of a given token type. Different from `findFirstToken`, this method searches the root node and the entire tree, not only the direct children.
- `findAllAstsOfType`: returns the list of all nodes of a given token type found within a given AST traversed in pre-order.

A key benefit of checkstyle over other approaches is that the custom analysis is implemented using Java—there is no need to learn another language or syntax for specifying an analysis rule. Another benefit is that checkstyle gives you the flexibility to inspect any syntactic element of a Java program, even comments. Other approaches used for architecture conformance analysis, such as AOP [8], are limited to inspecting class definitions, method definitions, method calls, and other specific points in a Java program.

5. Custom checks at TCU

We have created 40 checkstyle custom checks so far. The checks are divided into three categories: architecture conformance, coding guidelines, security constraints.

5.1 Architecture conformance checks

We have created 19 custom checks that verify design constraints defined in the reference architectures. These checks enforce: the layered architecture; specific class inheritance or interface realization that is required from certain types of modules; proper placement of business logic, data access logic, UI logic; proper use of infrastructure and "util" software elements; naming of certain types of modules as defined in the architecture; and disallowed dependencies in general.

The custom check in **Figure 1** is an example of architecture conformance check that can help to enforce a layered architecture, such as the one seen in **Figure 2**. That custom check along with similar checks can make sure usage dependencies that violate the design (shown by dashed lines in **Figure 2**) are not created. We have written several custom checks like that.

Another example of architecture conformance check is related to class inheritance. For instance, in our reference architecture, a design constraint dictates that stateless session beans [1] must be a subclass of a given abstract class. This abstract super class transparently introduces exception handling, auditing, and transaction management services in all subclasses. Thus, we created a custom check that identifies that the analyzed class is a stateless session bean and makes sure that that class is a subclass of the aforementioned abstract class.

5.2 Coding guideline checks

We created other 12 custom checks that enforce coding guidelines that are specific to our organization. These guidelines span various aspects of Java programming, including: exception handling, resource release to avoid resource leaks, proper placement of JUnit tests, thread programming traps.

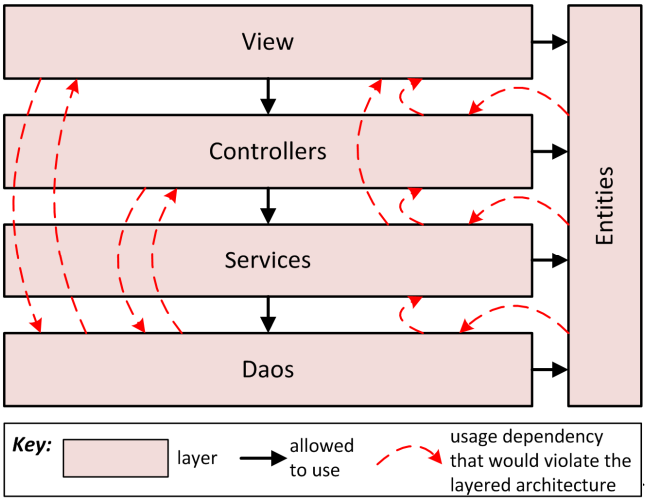

Figure 2. Layered architecture showing dependencies that could be enforced by architecture conformance checks.

An example of coding guideline that we enforce using a custom check is that threads cannot work on previously created transactional objects. In our reference architecture, we identify by inheritance, Java annotation or package namespace all the objects that carry a transactional context. Some applications employ the "introduce concurrency" tactic [3] by spawning threads for background processing. We created a custom check that makes sure the constructor of a thread does not receive as a parameter an object that holds transactional context. This check prevents two threads from operating on the same transaction and causing a runtime error.

5.3 Application security checks

A couple of years ago we had the opportunity to analyze part of our codebase using a static analysis tool called Fortify, which specializes in detecting security vulnerabilities in application programs. Although Fortify is very useful to improve application security, it has the limitations discussed in Section 2.2—the security checks the tool applies are generic and unaware of specific elements in our architecture. So we decided to implement custom checks that deal with security vulnerabilities. Some are generic and others target elements of our home-grown software infrastructure that are security sensitive.

We developed 9 custom checks that enforce security constraints in Java applications. These checks prevent vulnerabilities such as: SQL injection in JDBC and Hibernate programming, execution of external programs on Java web applications, hard-coded passwords, and security critical classes or methods that can be subclassed or overridden.

6. Next step: prevent new violations

Section 2.2 discusses two limitations of the common use of static analysis tools to automate conformance checks. The second limitation is that violations reported by automated checks can be ignored by developers, and unfortunately often they are. We solved this limitation in our organization with a mechanism that runs the checks when the developer attempts to commit a source

file to the code repository. This mechanism denies the operation if there are violations.

A Subversion (svn) hook is a program that can be configured in the subversion repository [13]. The hook is automatically invoked when there is a commit operation. We created a *pre-commit hook* that runs our checkstyle custom checks on the Java source files that are the subject of the commit operation. If any of the checks detects a violation, the commit operation fails and the user gets an error message indicating the source file, line number, and a description of the problem.

The svn pre-commit hook does not allow developers to ignore violations. The violations are either fixed or out of the codebase.

6.1 Notification of violations

As soon as the svn hook was in place, developers began to get error messages upon svn commit attempts. We realized we needed to keep track of these failed code commit attempts, not only to have a sense of the number of attempts, but more important to explain the enforced rule and indicate a solution to the developers.

We configured the svn hook to send an email to the architecture team whenever a developer tries to commit source code that violates the architecture. This email message allows the architecture team to pro-actively contact the developer and help address the issue. **Figure 3** shows the overall flow that takes place when a code commit attempt violates a custom check. (The diagram is a simplification, since the loop is executed for each custom check, not only one.)

7. Results and lessons learned

There are two general approaches that architecture teams follow to try to enforce the architecture. One is to act as the "architecture police" to make sure developers are following the published architecture. The other is to mentor and work closely with developers to make sure they understand and naturally follow the architecture. It may look like automated checks that bar source-code commit attempts because of violations fall within the first approach. In our experience, it's the opposite. The email notifications mentioned in Section 6.1 are the answer. They expose lack of understanding of the architecture to which we can respond immediately. We contact the developer to further explain why and how does his/her code violate the architecture. These email messages have been incredibly effective in raising awareness of the architecture and coding best practices.

Today the custom checks that execute prior to any svn commit operation quite frequently deny the introduction of violations in our Java codebase. More precisely, in a period of 12 months more than 400 commit attempts resulted in an error message to the code committer due to a custom check violation. Based on the specific violation in these attempts, the architecture team has sent approximately 40 email messages to developers with further clarification; in other instances a quick chat clarified matters. A couple of times this personal follow-up with the code committer ended up in revisions to the custom checks. However, the vast majority of cases resulted in the code being fixed by the developer.

The experience has been successful. At this point, we can quickly develop a custom check when a new design constraint is devised. There were some hurdles we had to overcome that could have been avoided. Following are some recommendations based on our experience.

7.1 Recommendations

In these two years working with custom checks, we adopted some practices to make the most out of the checks. Probably the most important guideline is to adopt an architecturally-evident coding style [9], which is translated into the first three recommendations in the list.

1. Use naming conventions that define different prefix or suffix for different types of classes and interfaces. This idea is quite

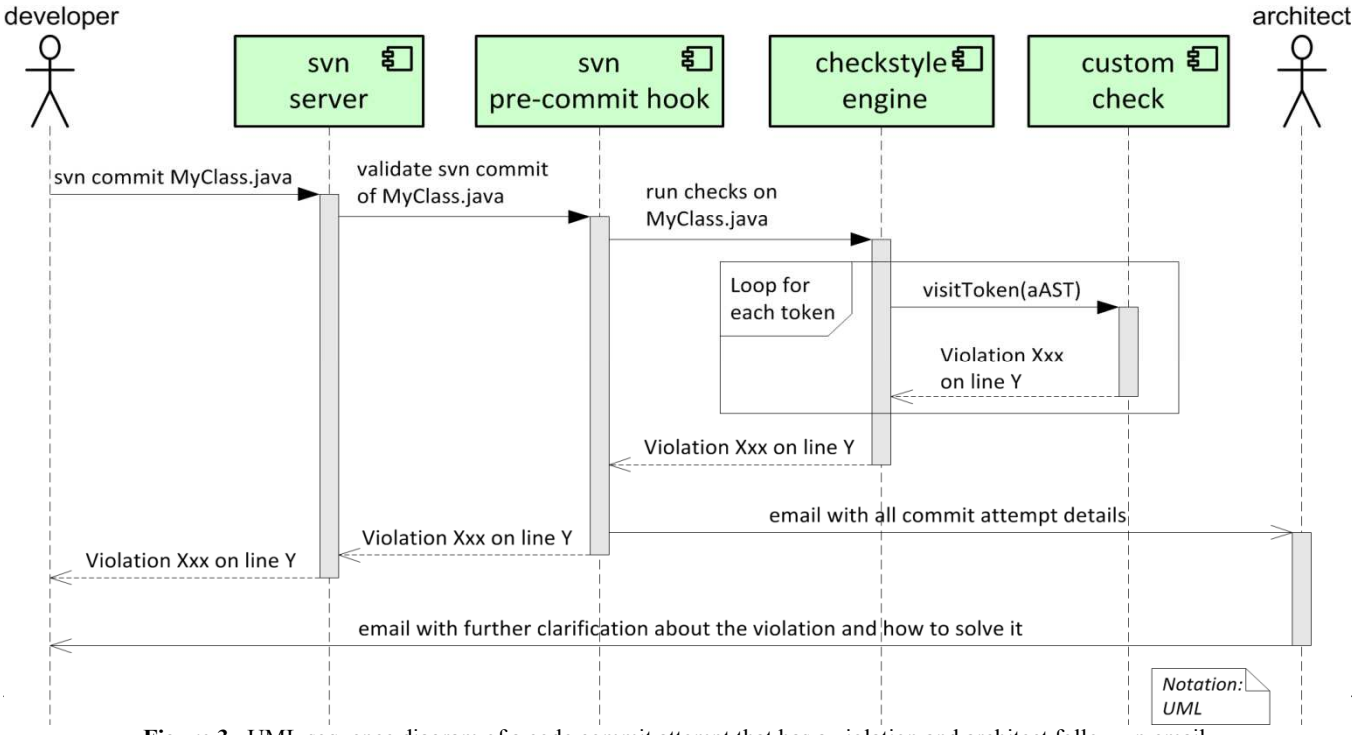

Figure 3. UML sequence diagram of a code commit attempt that has a violation and architect follow up email.

common in Java and other development platforms. For example, exception classes in Java by convention have suffix "Exception", the controller classes in applications crated with the Spring MVC framework usually use suffix "Controller".

2. When a naming convention that establishes a prefix or suffix for classes is not adequate for some reason, define a convention for the package namespace. For example, if not all classes in the presentation layer can be identified by strict name prefixes or suffixes, we could define that all of them should be in a package namespace that ends in "presentation".
3. If a naming convention is not adequate for packages either, define Java annotations that qualify the classes. For example, we could identify the classes that represent data entities mapped to relational database tables because they necessarily have the `@Entity` annotation.
4. Once established, divulge the existence of code verification at commit time to all developers, and then on to any newcomers. Indicate the benefits and try to get their buy-in. Make sure to open space for suggestions. Some of the checks we created were suggested by application developers.
5. Create and nourish the unit tests for your custom checks, and enhance them with new code idioms found along the way.
6. Create a mechanism for easily configuring exceptions. It should give you the ability to configure projects, packages or classes over which the custom checks should not be run.
7. Keep track of the number of violations found for each check. When you first implement it, write down the date and the total number of violations. If you are able to fix all violations, write down another checkpoint with the date that total number came down to zero. If you were not able to fix all violations, whenever you run the check over the entire codebase, take note of the number of violations, so you can assess if that number is under control. If you use the Sonar platform, you can use the time machine mechanism [10].
8. Make sure software architects understand the potential of the custom checks even if they don't understand the technical details. Then try to establish the following mindset among the architects: whenever you make a design decision that is reflected in the Java code, ask yourself whether that decision can be enforced via a custom check, and what conventions should be defined to make the custom check feasible. For example, suppose some business classes in your application need to support simple "undo" on operations that change state. The architect decides to use the Memento design pattern [7] to implement undo. Keeping in mind that a custom check can enforce the proper use of Memento, the architect just needs to define a means to identify those business classes. So, an extra decision could be that business classes that require undo must have annotation `@UndoEnabled`.
9. In addition to custom checks that can be executed at code commit time, enable the static analysis tool built-in checks in the developers IDE and/or in the continuous integration environment.

7.2 Process followed for each check

We have created custom checks that analyze the code for various kinds of rules. In common, all checks went through the same work process, which is described below and summarized in **Figure 4**:

1. The first step is to envision the rule and express it in terms of syntactic elements in the Java code.
2. Next the corresponding custom check is implemented using the checkstyle Java API.
3. The custom check is then executed against the entire codebase to generate an html report showing all violations. Sometimes there's only a handful, sometimes there are thousands of them. In this step we often find false positives that take us back to steps 1 and 2, that is, we refine the rule statement and its implementation as a custom check.
4. The most laborious step is to manually fix the violations in the codebase. In fact, this step involves a go/no-go decision with respect to fixing the violations. In some situations, that task is not feasible. The ideal situation is when we can take the time to fix the violations and move on to the next step.
5. Once the number of code violations for a particular custom check is down to zero (or a small number of violations in legacy code), the check is enabled in the svn pre-commit hook. As described in Section 6, the pre-commit hook will ensure that new violations will not be added to the codebase from then on.
6. Even when we can't fix all the violations, the check can still be enabled in the pre-commit hook. In this case, we either enable it only for "svn add" operations (new code) or we adapt the check to ignore the few modules with violations (they're considered exceptions to the rule).

The downside of a no-go decision for fixing the violations is step 4 is that the rule cannot be added to the pre-commit hook. Otherwise, developers will not be able to make any changes to classes that contain violations. Violations that stay in the codebase tend to creep up due to copy and paste programming [11]. To minimize this propagation of violations, our pre-commit hook treats *svn add* and *svn update* operations differently. Thus, rules that were not fixed in the overall codebase are still enforced for svn add operations. However, many developers quickly found out a trick to bypass the svn add conformance check. I will let it to the reader to find out what that trick is.

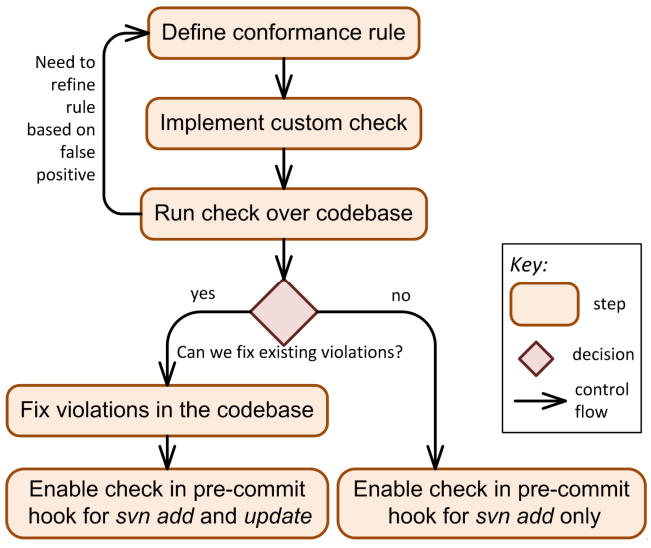

Figure 4. Process followed for defining, implementing and enabling a custom check.

8. Conclusions

The codebase of an active software project is like a shapeless, living entity that is subject to frequent, localized modifications and additions. Lack of conformance between the as-designed architecture and the as-built architecture can easily happen and indeed is a problem faced by many software development teams. For many years, I've studied and tried out various solutions to this problem. This paper described the first solution I see that is simple to implement, scales up to all codebase, allows for continuous verification, and is powerful and flexible. At TCU we have tackled the problem with static analysis of the code. To verify some design constraints in the code, a simple lexical analysis would suffice. For other constraints, contextual information (semantic information) is needed. We have created custom checks using the checkstyle tool, which offers a Java API that uses the Visitor design pattern and gives us the ability to create AST-based verifications with contextual information.

Checkstyle checks also have limitations. The most inconvenient one is that a check logic only sees the tokens within the source file being analyzed. A check can't verify, for example, that a given class is a grandchild of a specific class—it can only see the immediate superclass because of the `extends` keyword present in the type declaration. (One can use introspection to overcome this particular limitation, but that alternative incurs other limitations.) Checkstyle in particular is limited to parsing Java artifacts, so custom checks cannot verify design and implementation constraints on xhtml, jsp, wsdl, xsd, or any other non Java artifacts. Another limitation of checkstyle checks and static analysis in general is that they cannot detect patterns of interaction that rely on polymorphism, late binding, or introspection. For dynamic interactions, we would need a profiling tool, code coverage tool, or worse, code instrumentation. However, dynamic analysis is not suitable for continuous verification because the environment where the verification takes place can be set up to read source files (static analysis), but typically does not have all the infrastructure pieces required to *execute* the programs.

The pairing of checkstyle checks with the pre-commit svn hook was an important addition to the solution. The pre-commit hook not only curbs violations in the source code repository, but also (discreetly) names the developers who need further clarification about architecture or implementation rules.

In addition to architecture enforcement, we have successfully used checkstyle checks to enforce proper exception handling, use of design patterns, security guidelines for application development, and other good Java development practices. I invite the reader who works with Java development to try out the approach described in this paper.

Acknowledgments

I want to thank Jefferson da Silva and Frederico Ferreira for invaluable help in exploring the checkstyle API, implementing many different custom checks, and—the hardest task—fixing violations in the code by the thousands. I'm also grateful to Marcelo Pacote and Fabiana Ruas for the continuous support to this project.

References

[1] *Enterprise JavaBeans 3.1, Final Release*. Sun Microsystems, November 2009.

[2] Clements, P., F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, and J. Stafford. *Documenting Software Architectures: Views and Beyond, Second Edition*. Addison-Wesley, 2010.

[3] Bass, L., P. Clements, R. Kazman. *Software Architecture in Practice, Third Edition*. Addison-Wesley, 2013.

[4] Garland, J., and R. Anthony. *Large-Scale Software Architecture: A Practical Guide Using UML*. John Wiley & Sons, 2003.

[5] Checkstyle project: http://checkstyle.sourceforge.net/

[6] Fowler, M. *Patterns of Enterprise Application Architecture*. Addison-Wesley, November 2002.

[7] Gamma, E., R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, November 1994.

[8] Merson, P. *Using Aspect-Oriented Programming to Enforce Architecture*. Software Engineering Institute, CMU/SEI-2007-TN-019, September 2007.

[9] Fairbanks, G. *Just Enough Software Architecture*. Marshall & Brainerd, 2010.

[10] Mallet, F. "Sonar Time Machine: replaying the past". Blog post available at www.sonarsource.org/sonar-time-machine-replaying-the-past. January 2009.

[11] "Copy and Paste Programming". Available at: http://c2.com/cgi/wiki?CopyAndPasteProgramming

[12] "List of tools for static code analysis". Available at: http://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis.

[13] Collins-Sussman, B., B. Fitzpatrick, and C. Pilato. "Version Control with Subversion". Available at: http://svnbook.red-bean.com/en/1.7/svn.reposadmin.create.html#svn.reposadmin.create.hooks.