Ultimate Architecture Enforcement
Write Your Own Rules and Enforce Them Continuously

SATURN
May 2017
Paulo Merson
Agenda

- Architecture conformance
- Custom checks lab
- Sonarqube
- Custom checks at TCU
- Lessons learned
Exercise 0 – setup

Open www.dontpad.com/saturn17
Follow the steps for “Exercise 0”

Pre-requisites for all exercises:
- JDK 1.7+
- Java IDE of your choice
- maven
Consequences of lack of conformance

Lower maintainability, mainly because of undesired dependencies

• Code becomes brittle, hard to understand and change

Possible negative effect

• on reliability, portability, performance, interoperability, security, and other qualities
• caused by deviation from design decisions that addressed these quality requirements
Factors that influence architecture conformance

How effective the architecture documentation is
Turnover among developers
Haste to fix bugs or implement features
Size of the system
Distributed teams (outsourcing, offshoring)
Accountability for violating design constraints
How to avoid code and architecture disparity?

1) Communicate the architecture to developers
   - Create multiple views
   - Structural diagrams + behavior diagrams
   - Capture rationale

Not the focus of this tutorial
How to avoid code and architecture disparity?

2) Automate architecture conformance analysis
   • Often done with static analysis tools
Built-in checks and custom checks

Static analysis tools come with many built-in checks

• They are useful to spot bugs and improve your overall code quality
• But they’re oblivious to your design constraints and project-specific named layers and other elements

Custom checks (aka custom rules) can enforce your architecture

• Some static analysis tools offer APIs for that
Agenda

- Architecture conformance
- Custom checks lab
- Sonarqube
- Custom checks at TCU
- Lessons learned
Checkstyle

It is a free open source static code analysis tool
It offers a mature Java API for creating checks
Checks use the Visitor pattern
The API gives you access to the entire AST of a Java file
Exercise 1 – AST GUI

Follow the steps for “Exercise 1”  (www.dontpad.com/saturn17)

Pre-requisites for all exercises:
• JDK 1.7+
• Java IDE of your choice
• maven
Adventure Builder reference application

Old J2EE application created by Sun as an example of Web Services application
Over 250 source files
Approximately 14 KLOC (Java)
We have documented its architecture:
  • wiki.sei.cmu.edu/sad/index.php/The_Adventure_Builder_SAD
Checkstyle analyzer

Class `com.puppycrawl.tools.checkstyle.Main`
Executes checks configured in an xml file

Where’s the analyzed file in this diagram?
A usage dependency rule

Rule: *code in processmanager cannot use code in opc*

• How do we identify code that belongs to processmanager?
• How do we identify use of modules that belong to opc?
• What tokens do we need to visit?
Exercise 2 – CheckProcessManagerCallsOpc

There is an exception to the rule:

- *processmanager may use utils inside opc*

Follow the steps for “Exercise 2” on dontpad

**Homework assignment:**

1) Extend the check to consider **static imports**
2) Extend the check to find processmanager → opc references that **don’t use import**
Check for a mandatory generalization

Adventure Builder uses a web framework called “waf”
In “waf” a class that handles web page requests is an “html action”
So, “HTML action” is a special type of module in the architecture
Rule: *html action classes must extend HTMLActionSupport*
  - How do we identify an HTML action module?
An **architecturally-evident coding style** encourages you to embed hints in the source code that makes the architecture evident to a developer who reads the code.

Architecturally-evident coding style in practice (in Java)

- Naming conventions (e.g., “HTMLAction” suffix)
- Annotations (e.g., `@HtmlAction`)
- Package namespace (e.g., “*.actions.*”)
- Interfaces and abstract classes
- Comments
## Essential checkstyle API methods

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>getFirstChild()</code></td>
<td>Returns the first child token.</td>
</tr>
<tr>
<td><code>getNextSibling()</code></td>
<td>Returns the next sibling token.</td>
</tr>
<tr>
<td><code>getParent()</code></td>
<td>Returns the parent token or null if no such parent exists.</td>
</tr>
<tr>
<td><code>getPreviousSibling()</code></td>
<td>Returns the previous sibling or null if no such sibling exists.</td>
</tr>
<tr>
<td><code>findFirstToken(int type)</code></td>
<td>Returns the first child token that makes a specified type.</td>
</tr>
<tr>
<td><code>getFirstChild()</code></td>
<td>Returns the number of child nodes one level below this node.</td>
</tr>
<tr>
<td><code>getFirstChild(int type)</code></td>
<td>Returns the number of direct child tokens that have the specified type.</td>
</tr>
</tbody>
</table>

From: [http://checkstyle.sourceforge.net/apidocs](http://checkstyle.sourceforge.net/apidocs)
Extending the API

List<DetailAST> tutorial.checks.CustomCheck.findAllAstsOfType(DetailAST aAST, int type)
Recursively traverse an expression tree and return all ASTs matching a specific token type.

Returns:
list of DetailAST objects found; returns empty List if none is found.

DetailAST tutorial.checks.CustomCheck.findFirstAstOfType(DetailAST aAST, int type)
Recursively traverse a given AST and return the first AST node matching a specific token type within the
given AST. This method differs from DetailAST.findFirstToken(int) in that it searches for the
given type in the specified node itself, all children, and indirect descendants (the whole tree), whereas
DetailAST.findFirstToken(int) only searches the direct children.

Returns:
first DetailAST found or null if no node of the given type is found

see super class tutorial.checks.CustomCheck
Exercise 3 – CheckHtmlAction ExtendsHtmlActionSupport

There’s an exception to the rule:

• *extends not needed if html action class implements HTMAction*

Follow the steps for “Exercise 3” on dontpad

*Homework assignment:*

*Improve method extendsHtmlActionSupport to work when the superclass uses the fully qualified class name*
JUnit tests for custom checks

In general, custom checks require “integration tests” rather than simple “unit tests”

- Create an input file (.java) with violations at specific locations
- Run your check against the input file
- In your test assert the violation was reported (stdout)
- Also test non-violation situations when necessary
Exercise 4 – CheckHtmlAction
ExtendsHtmlActionSupportTest

Method testWhenHtmlActionHasImport tests classes that implements HTMLAction
It asserts the absence of violations
Follow the steps for “Exercise 4” on dontpad

Homework assignment:
Enable CheckProcessManagerCallsOpcTest and adapt it to test all situations in input test file:
“InputCheckProcessManagerCallsOpcTest.java”
Agenda

Architecture conformance
Custom checks lab
Sonarqube
Custom checks at TCU
Lessons learned
Sonarqube integration

Sonarqube is a pluggable platform for code quality analysis
Checkstyle custom checks can be packaged as a sonarqube plug-in
Sonarqube analyses will then show violations found by your checks
For each check you can configure

• Severity (blocker, critical, major, minor, info)
• Type (bug, vulnerability, code smell)
• Message to developer
Sonarqube Java API

Sonarqube also offers a Java API to create custom checks

*Should I use it instead of checkstyle to create my checks?*

- It depends on how much is your analysis tied to sonarqube

Recently, we released the [SonarQube Java 1.4 plugin ecosystem](https://blog.sonarsource.com/already-158-checkstyle-and-pmd-rules-deprecated-by-sonarqube-java-rules/), and as announced in [What’s coming up for SonarQube in 2013](https://blog.sonarsource.com/already-158-checkstyle-and-pmd-rules-deprecated-by-sonarqube-java-rules/), we’re working hard to progressively deprecate as many Checkstyle and PMD rules as possible, and write native replacements for them using the SonarQube Java rule engine. We’ll continue to support both tools for the foreseeable future, but the goal is to remove them from the default SonarQube Java plugin ecosystem package.
# Checkstyle API vs Sonarqube API

<table>
<thead>
<tr>
<th>Checkstyle API (checkstyle v6.19)</th>
<th>Sonarqube API (sonarqube v6.0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Setup for JUnit integration tests is complex</td>
<td>JUnit tests are easier to implement 😊</td>
</tr>
<tr>
<td>No SonarLint support</td>
<td>Checks can run on SonarLint (plugin for IDEs) 😊</td>
</tr>
<tr>
<td>checkstyle.gui.Main 😊</td>
<td>No GUI to visualize syntax tree</td>
</tr>
<tr>
<td>Single type for all tokens (DetailAST)</td>
<td>Multiple types for various kinds of tokens</td>
</tr>
<tr>
<td>Generic methods basically for navigating the AST</td>
<td>Convenience methods for each kind of token 😊</td>
</tr>
<tr>
<td>Full access to entire AST 😊</td>
<td>Can’t easily navigate entire tree from a given token</td>
</tr>
<tr>
<td>Java API is well documented 😊</td>
<td>Java API is not well documented</td>
</tr>
<tr>
<td>Can run via command line with different arguments 😊</td>
<td>Requires Sonarqube platform to run</td>
</tr>
<tr>
<td>XML report (convertible to html) 😊</td>
<td>No embedded reporting capability</td>
</tr>
<tr>
<td>Can analyze Java</td>
<td>Can analyze Java, JavaScript, COBOL, PHP, and RPG 😊</td>
</tr>
</tbody>
</table>
Agenda

Architecture conformance
Custom checks lab
Sonarqube
Custom checks at TCU
Lessons learned
Process to define, implement, and enable a custom check

1. Define conformance rule
2. Implement custom check
3. Run check over codebase
4. Fix violations in the codebase
5. Enable check in SonarQube analysis
6. Enable check in pre-commit hook
7. Can we fix existing violations?
   - Yes: Refine rule based on false positive
   - No: Is conformance mandatory?
      - Yes: Enable check in SonarQube analysis
      - No: Finish.
Custom checks at TCU

We’ve created 51 custom checks

• 27 checks for architecture conformance
• 14 checks for coding guidelines
• 10 checks for application security
Architecture conformance checks

Enforce

- the layered architecture
- placement of UI, business, data access logic
- mandated generalizations
- naming of software elements
- use of infrastructure/util modules
- design standards in reference architectures
Coding guideline checks

Check proper coding of

• Exception handling
• Resource release
• Thread programming
• JUnit tests
• REST and SOAP services
Application security checks

Prevent vulnerabilities, such as:

• SQL injection (JDBC and Hibernate)
• Execution of external programs in web apps
• Hard-coded passwords
• Subclassing/overriding security critical classes or methods
• Lack of authentication/authorization at entry points
Agenda

- Architecture conformance
- Custom checks lab
- Sonarqube
- Custom checks at TCU
- Lessons learned
Lessons learned (1)

Add custom checks to sonarqube analysis
And run analysis in your continuous integration
Lessons learned (2)

Adopt an architecturally-evident coding style

An *architecturally-evident coding style* encourages you to embed hints in the source code that makes the architecture evident to a developer who reads the code.
Lessons learned (3)

Let developers know about the custom checks and suggest changes and improvements
Let architects understand the potential of the custom checks
Lessons learned (4)

Be careful not to become the “architecture police”
Be the “architecture mentors”
Lessons learned (5)

Create a mechanism to easily trump a check in exceptional cases
Lessons learned (6)

Automate

• the way violations are reported to developers
• email with further clarification about a denied commit attempt
For more information

Email: pmerson@acm.org
Checkstyle: http://checkstyle.sourceforge.net/writingchecks.html
Sonarqube: docs.sonarqube.org/display/PLUG/Writing+Custom+Java+Rules+101


Exercise 0
========
1) Download checkstyle 6.19 (compatible with JDK 1.7+):
unzip the file

*** In the following, we'll assume checkstyle was downloaded to C:\dev\checkstyle-6.19\ ***

2) Download the application we'll use as "target code base":
if you have a github account then
   go to https://github.com/pmerson/adventurebuilder and fork the project
   git clone the project to your machine
else
   git clone https://github.com/pmerson/adventurebuilder.git
   OR
   download the zip file at https://github.com/pmerson/adventurebuilder

*** In the following, we'll assume the application was cloned to C:\saturn\adventurebuilder ***

3) Download the maven project that contains our "custom checks".
if you have a github account then
   go to https://github.com/awynne/adventurebuilder.git
   git clone the project to your machine
else
   git clone https://github.com/pmerson/customchecks.git
   OR
   Download the zip file at https://github.com/pmerson/customchecks

*** In the following, we'll assume this project was cloned to C:\saturn\customchecks ***
Exercise 1
========

1) Run the checkstyle AST GUI program:
   java -cp C:\dev\checkstyle-6.19\checkstyle-6.19-all.jar
        com.puppycrawl.tools.checkstyle.gui.Main

2) Click the Open File at the bottom and open:
   C:\saturn\adventurebuilder\components\waf\src\java\com\sun\j2ee\blueprints\waf\controller\web\ErrorMapping.java
Exercise 2
========

1) Make sure the declaration of CheckProcessManagerCallsOpc in custom_checks.xml is uncommented.

2) Build the customchecks project:
   cd C:\saturn\customchecks
   mvn clean package -U

3) Make sure the initial version of the custom check works fine:
   java -cp C:\dev\checkstyle-6.19\checkstyle-6.19-all.jar;C:\saturn\customchecks\target\custom-checks-1.0-SNAPSHOT.jar
   com.puppycrawl.tools.checkstyle.Main -c C:\saturn\customchecks\src\main\resources\custom_checks.xml
   C:\saturn\adventurebuilder\apps\opc\processmanager-ejb\src\java\com\sun\j2ee\blueprints\processmanager\transitions

4) The previous command ran the check on package "transitions" only. Now execute the check on the entire adventurebuilder project:
   java -cp C:\dev\checkstyle-6.19\checkstyle-6.19-all.jar;C:\saturn\customchecks\target\custom-checks-1.0-SNAPSHOT.jar
   com.puppycrawl.tools.checkstyle.Main -c C:\saturn\customchecks\src\main\resources\custom_checks.xml
   C:\saturn\adventurebuilder

5) Edit the custom check to allow calls to the utils package.
   If you have time, refactor the code in visitToken to move the logic for package definition to visitPackageDef, and the logic for
   import to visitImport.

6) Build the project and execute the custom check to make sure the violation is gone.
Exercise 3
========

1) Add (or uncomment) the new check to custom_checks.xml.

2) Build and run the new check to analyze the adventurebuilder project.
   
mvn clean package
   java -cp C:\dev\checkstyle-6.19\checkstyle-6.19-all.jar;C:\saturn\customchecks\target\custom-checks-1.0-SNAPSHOT.jar
   com.puppycrawl.tools.checkstyle.Main -c C:\saturn\customchecks\src\main\resources\custom_checks.xml C:\saturn\adventurebuilder

3) Rename file TO_BE_RENAMED.java to CatalogHTMLAction.java. Edit the file and uncomment the class definition for CatalogHTMLAction.

4) Run the check again. It should find a violation in CatalogHTMLAction.java.

4) Edit the custom check to also allow *HTMLAction classes that do not extend HTMLActionSupport but implement the HTMLAction interface.
   
   Hint 1: it's reasonable to think that any identifier with text "HTMLAction" found under the "implements" token AST indicates that the class implements the interface.
   
   Hint 2: remember not all classes implement an interface.

5) Rebuild the project and execute the custom check to make sure the violation is gone.
Exercise 4
========

1) Edit CheckHtmlActionExtendsHtmlActionSupportTest.java and enable the first test method: testWhenHtmlActionHasNoExtendOrImport().

2) Build the project and check that the test was executed successfully.

3) Enable the second test method: testWhenHtmlActionHasImport(). Build the project. The second test should fail.

3) Open file InputCheckHtmlActionExtendsHtmlActionSupportTest2.java inside C:saturn\customchecks\src\test\resources. Edit this input test file so that the test will no longer fail.

4) Build the project and check that both tests pass.
Backup slides
Two limitations of static analysis tools as commonly used

1. Many people just enable built-in checks, which
   - have configurable thresholds but are still generic
   - are oblivious to custom design constraints and project-specific named layers and other elements

2. Developers can ignore reported violations
   - There can be thousands of violations 😞
   - There’s no time to “clean the house” 😞
   - There may be no accountability for adding violations 😞
What if developers ignore violations?

**Problem:** Developers can ignore violations reported by custom checks (and other checks)
- There can be thousands of violations
- There’s no time to “clean the house”
- There may be no accountability for adding violations

**Solution:** make compliance mandatory by running checks at code commit
Pre-commit hooks

Upon code commit a *pre-commit hook*

- runs custom checks on each source file

If a custom check detects a violation

- the commit operation is denied
- developer sees a descriptive error message
- architecture team is notified of the failed commit
- developer gets email with further explanation
Failed code commit attempt
Some results

In the first 24 months

- 750+ commit attempts denied
- 135+ email messages to developers with further clarification on violations

Enhancements to checkstyle and PMD submitted to these projects

But the real hard work was

- Fixing ~12,000 violations in the codebase

And the real value was

- Increased code quality
- Improved awareness of the architecture and coding guidelines
Lessons learned (1)

Add to pre-commit hooks only the checks

• with zero violations in the target codebase
• that enforce mandatory rules