Software Patterns

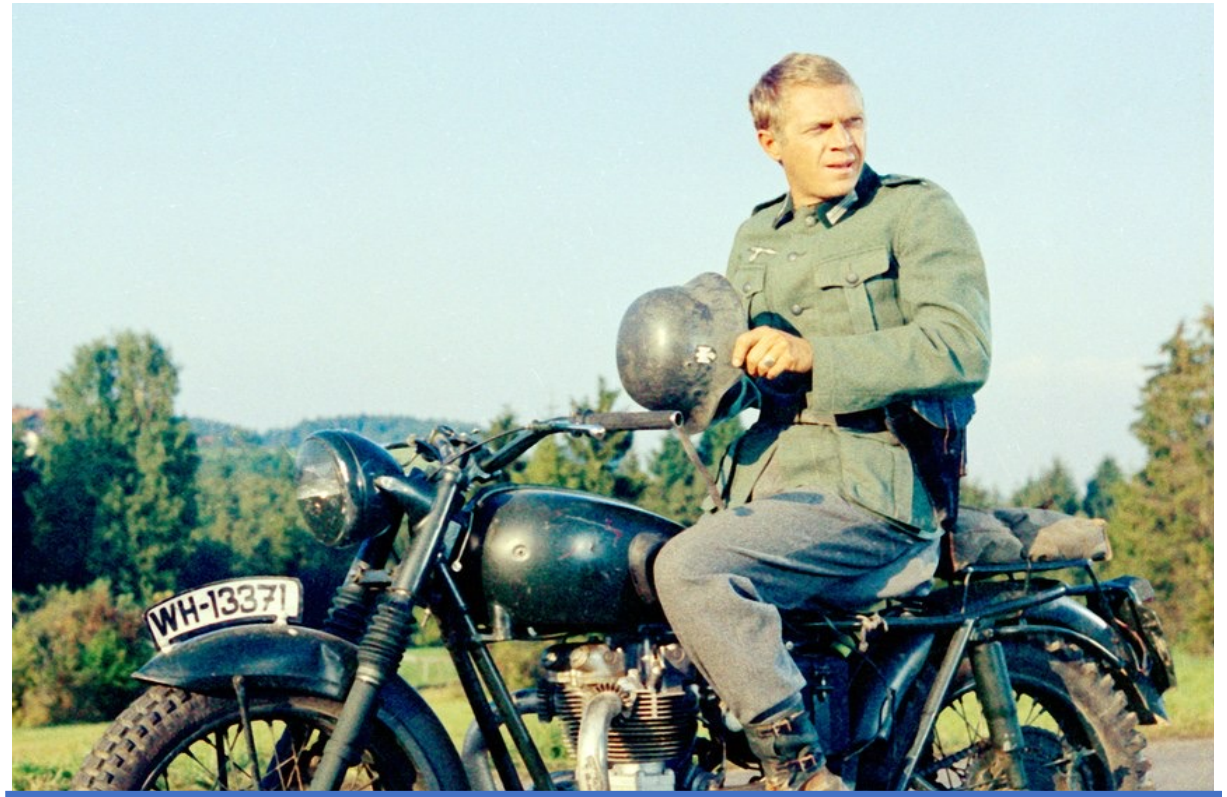
L09: Quality Patterns

Marlo Häring & Prof. W. Maalej (@maalejw)

Outline

1. Introduction
2. Smells and Refactoring
3. Continuous Quality Improvement

Pirsig's Definition of Quality



Robert Pirsig

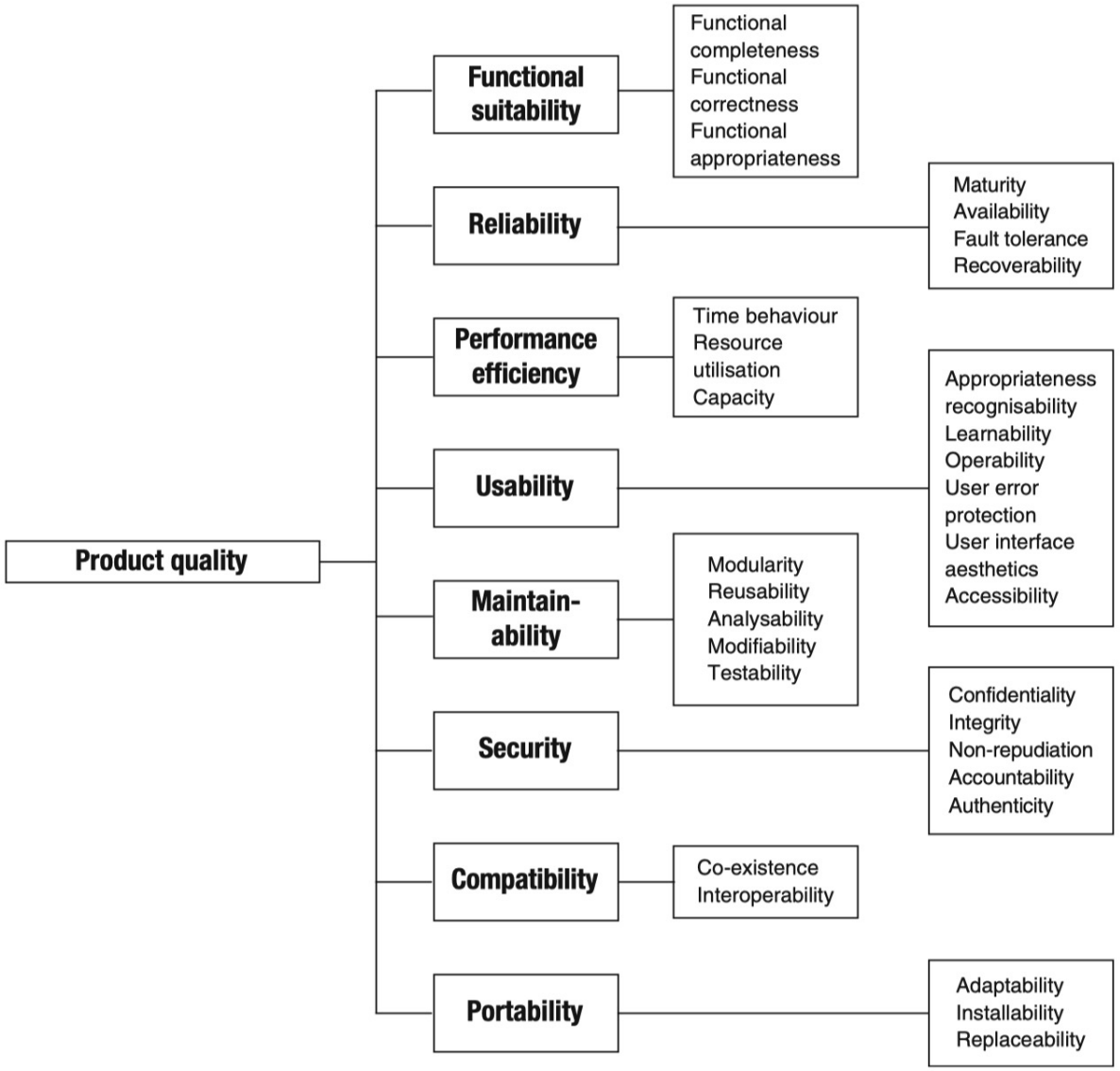
Quality is a characteristic of thought and statement that is recognized by a non-thinking process. Because definitions are a product of rigid, formal thinking, quality cannot be defined

... but even though quality cannot be defined, you know what quality is!

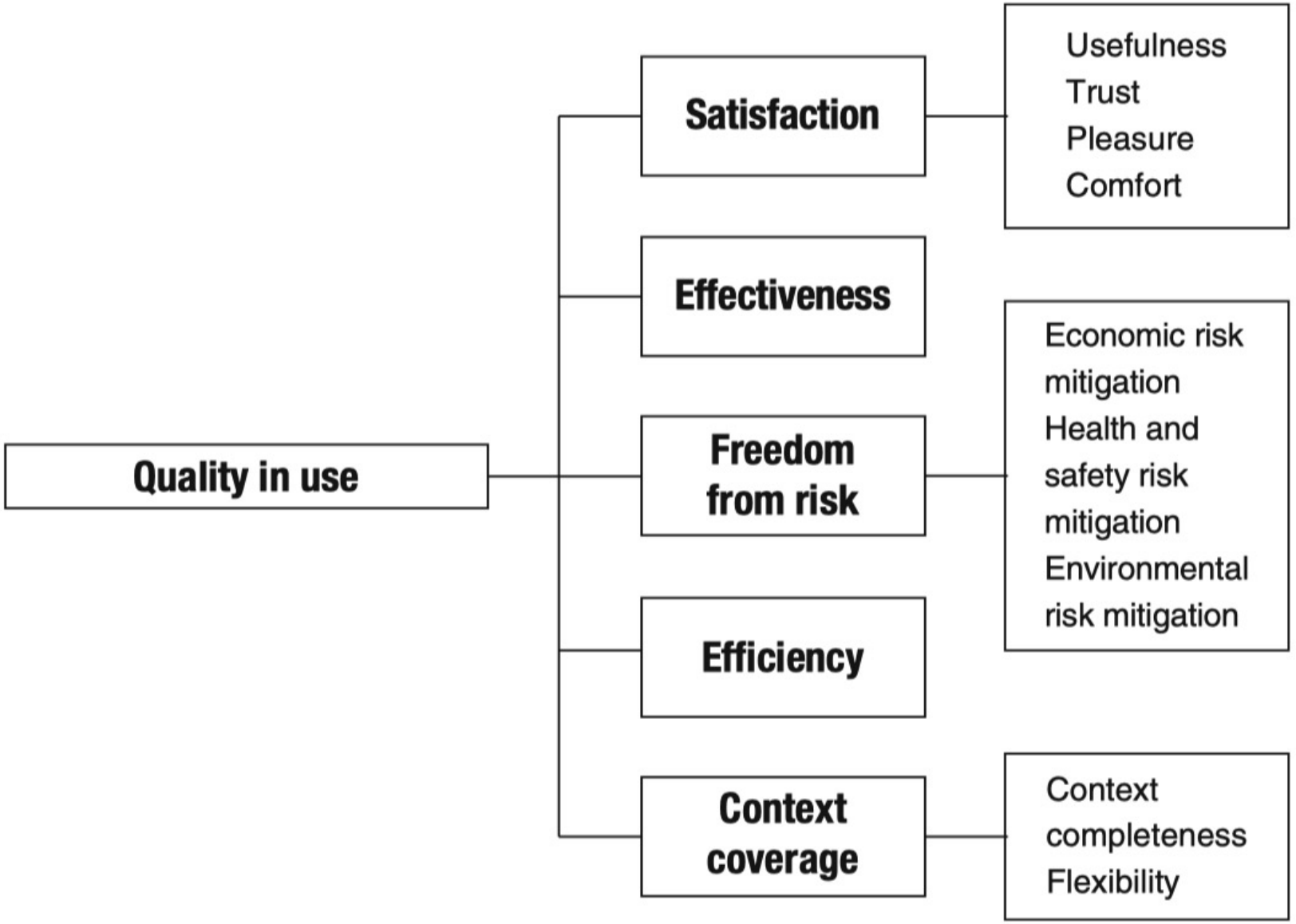
Different Views of Quality

- ISO 9001
- ISO/IEC 9126
- SQuaRE
- Industry specific

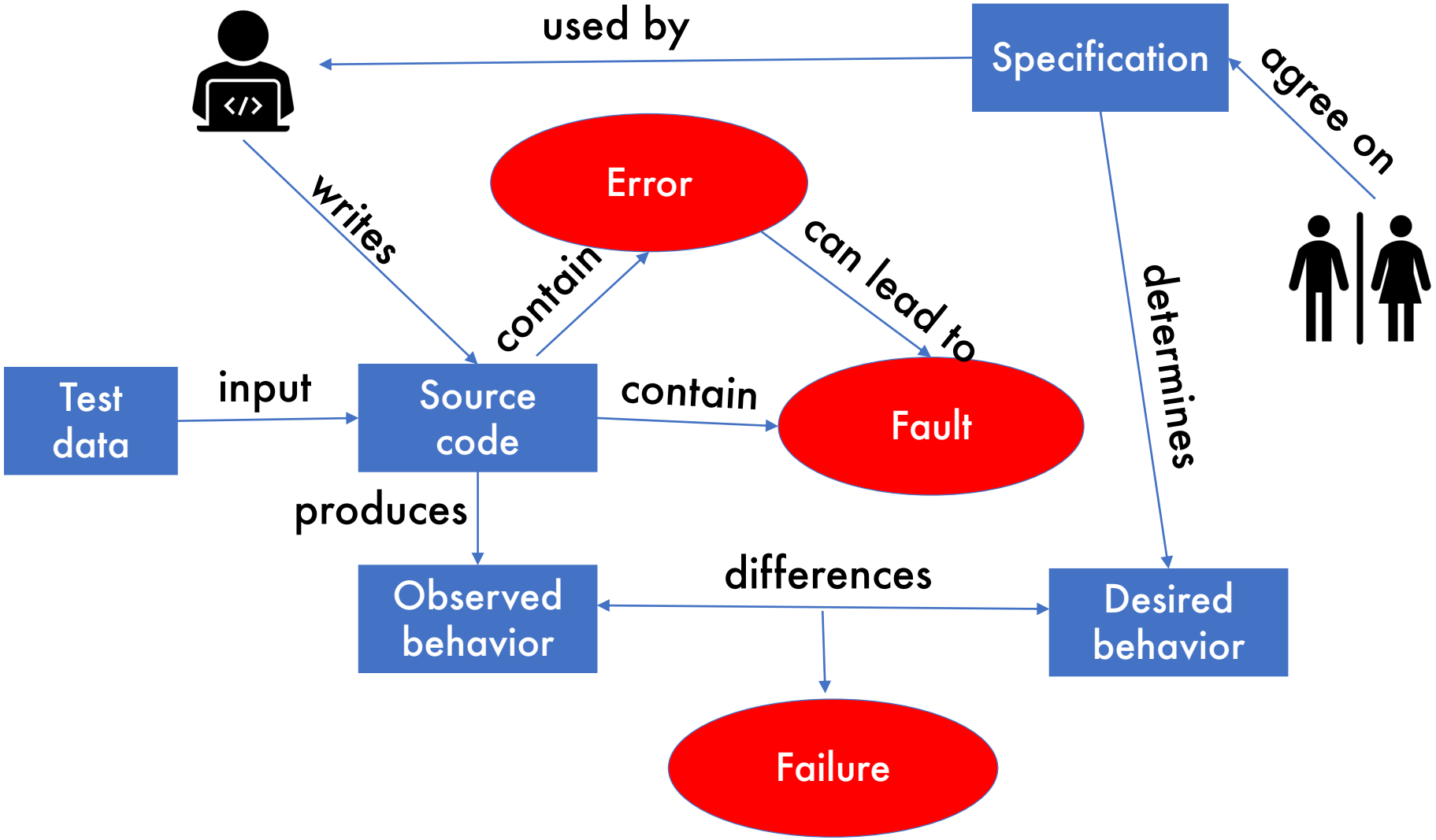
Product Quality (ISO 9126)



Quality in Use (SQuaRE)



Errors, Faults, and Failures

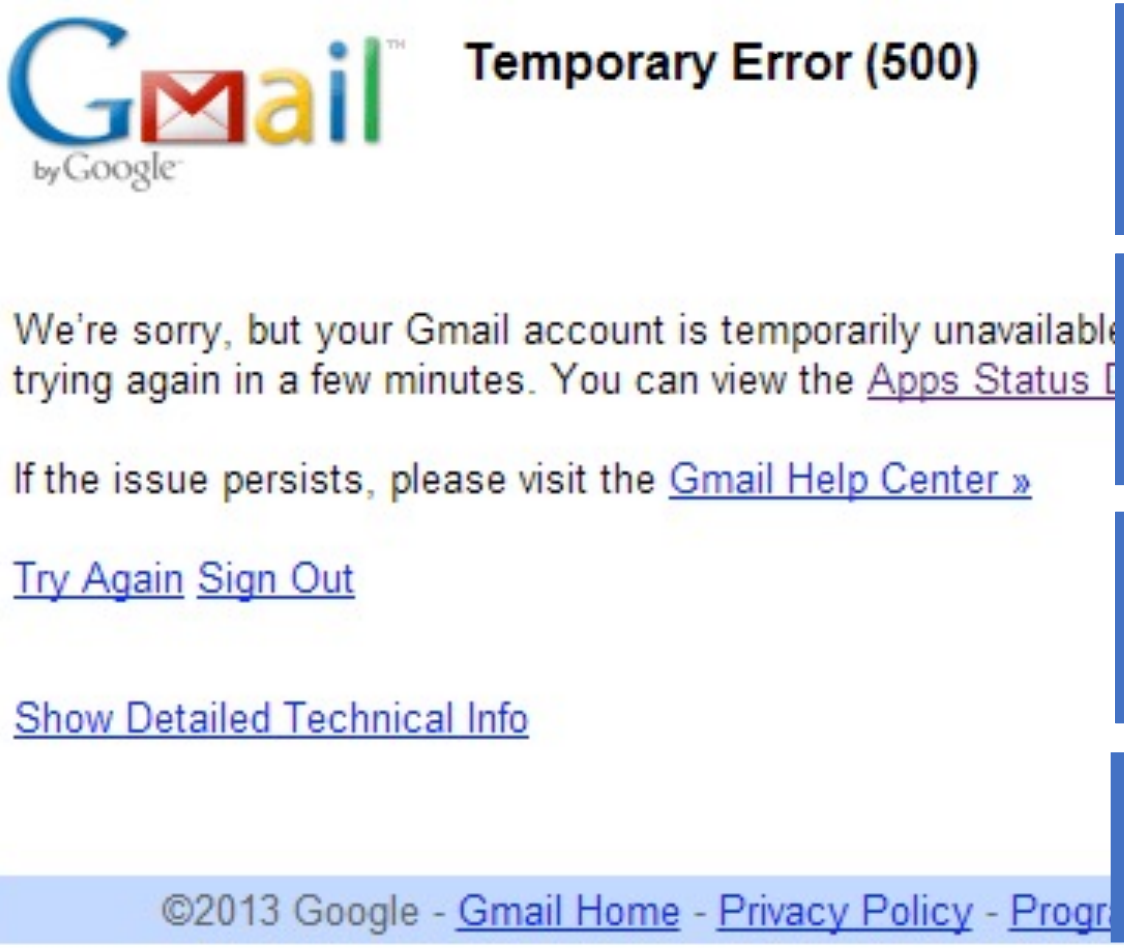


Errors, Faults, and Failures

- **Error** are made by developers (syntax, grammar mistake)
- **Fault** can cause improper functioning of the system
- **Failure** is a fault that becomes visible in certain situation

- **Incident** is a failure with bad repercussions on the business

Example



Incident: Most Google users who tried to log-in in Gmail found they were unable to access for approximately 25 minutes

Failure: An incorrect configuration was sent to live services over 15 minutes and user requests were ignored.

Fault: an internal system that generates configuration may generate an incorrect configuration.

Error: The software developer of the configuration generator introduced a bug in the program.

Other Failure Examples

Outline

1 Introduction

2 Smells and Refactoring

3 Continuous Quality Improvement

Bad Smells

" Bad smells (i.e., code and design smells) are poor solutions to recurring implementation and design problems "

-Naouel Moha et al.

Moha, N., Gueheneuc, Y. G., Duchien, L., & Le Meur, A. F. (2010). Decor: A method for the specification and detection of code and design smells. *IEEE Transactions on Software Engineering*, 36(1), 20-36.

Code Smells

- " Are surface indications that usually corresponds to a deeper problem in the system "
- -Martin Fowler

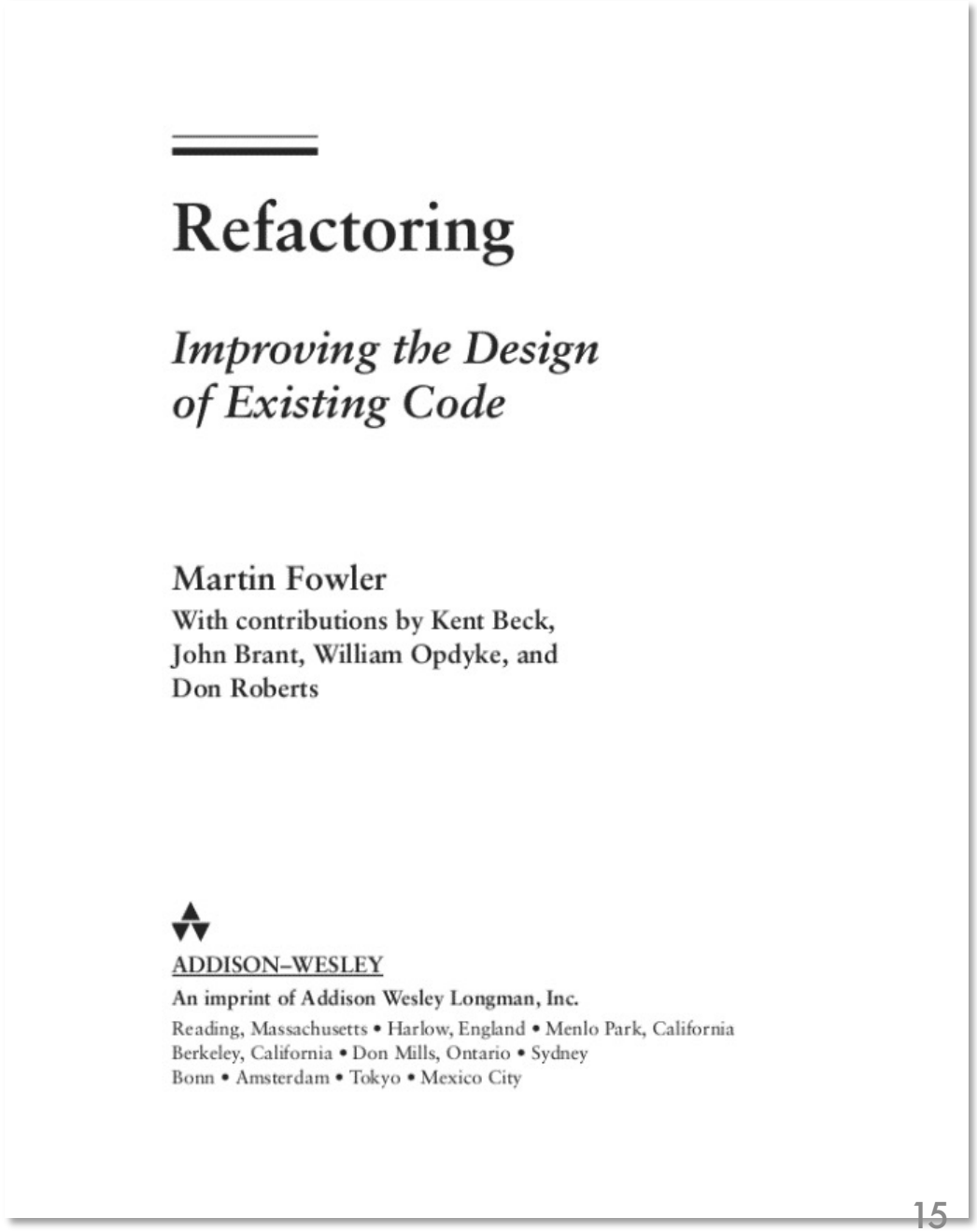
Characteristics of Code Smells

- Are low level design problem, not bugs (yet)
- Are something that is quick to identify
- Do not always indicate a problem
- Robert Martin and Kent Beck defined a catalogue with 22+ code smells

Refactoring

"Change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour"

-Martin Fowler



Refactoring Patterns Based on SOLID

- Single Responsability
- Open-closed
- Liskov substitution
- Interface segregation
- Dependency inversion

Single Responsability

- **Context**

I need to instantiate a class

- **Problem**

The new instance needs several other unrelated objects

```
1 class Email {
2   public Email(Person, Config, Content);
3   public void sendEmail();
4   public Message fetchEmail();
5   public HTMLBody createHTMLContent();
6
7 }
```

Single Responsability

- **Principle**

A unit serves a single principle and has one and only reason to change

- **Refactoring**

Create a new class and place the fields and methods responsible for the relevant functionality in it.

```
1 class Email {
2   public Email(Person, Config);
3   public void sendEmail();
4   public Message fetchEmail();
5 }
6
7 class Formatter {
8   public HTMLBody createHTMLContent();
9 }
```

Open-Closed

- **Context**

I want to add a new functionality to a class

- **Problem**

I have to edit the class every time

```
1  class GraphicEditor {
2      public void drawShape(Shape s) {
3          if (s.m_type==1) drawRectangle(s);
4          else if (s.m_type==2) drawCircle(s);
5      }
6      public void drawCircle(Circle r) {....}
7      public void drawRectangle(Rectangle r) {....}
8  }
9
10 class Shape {
11     int m_type;
12 }
13
14 class Rectangle extends Shape {
15     Rectangle() {
16         super.m_type=1;
17     }
18 }
19
20 class Circle extends Shape {
21     Circle() {
22         super.m_type=2;
23     }
24 }
```

Open-Closed

- **Principle**

An unit should be open for extension but closed for modification

- **Refactoring**

Create a hierarchy (e.g., *Replace Type With Subclass*) and replace conditional with polymorphism

```
1  class GraphicEditor {
2      public void drawShape(Shape s) {
3          s.draw();
4      }
5  }
6
7  interface Shape {
8      void draw();
9  }
10
11 class Rectangle implements Shape  {
12     public void draw() { // draw the rectangle}
13 }
14
15 class Circle implements Shape  {
16     public void draw() { // draw the circle}
17 }
```

Liskov Substitution

- **Context**

In the client code, I want to substitute the functionality offered by a class with its valid subclass

- **Problem**

The subclass does not behave as expected

```
1 class Rectangle {
2     private int m_width;
3     private int m_height;
4     public void setWidth(int width){m_width = width;}
5     public void setHeight(int height){m_height = height;}
6     public int getWidth(){return m_width;}
7     public int getHeight(){return m_height;}
8     public int getArea(){return m_width * m_height;}
9 }
10
11 class Square extends Rectangle {
12     public void setWidth(int width){
13         m_width = width;
14         m_height = width;
15     }
16 }
17     public void setHeight(int height){
18         m_width = height;
19         m_height = height;
20     }
21 }
22
23 class LSPTest {
24     private static Rectangle getNewRectangle(){
25         return new Square();
26 }
27
28 public static void main (String args[]){
29     Rectangle r = LSPTest.getNewRectangle();
30     r.setWidth(5);
31     r.setHeight(10);
32     System.out.println(r.getArea());
33     //The area is 100 instead of 50.
34     }
35 }
```

Liskov Substitution

- **Principle**

Functions that use references to base units must be able to use their derivation without knowing it

- **Refactoring**

Extract separate classes and remove inheritance

```
1 class Rectangle {
2     public void setHeight(int height)
3     public void setWidth(int width)
4     ...
5 }
6
7 class Square {
8     public void setWidth(int width)
9 }
```

Interface Segregation

- Context

In my class, I am implementing the required methods to conform to an interface

- Problem

I am implementing several methods that will not be used or are not necessary

```
1  interface IWorker {
2      public void work();
3      public void eat();
4  }
5
6  class Worker implements IWorker{
7      public void work() {// ....working}
8      public void eat() {// ..... eating in launch break}
9  }
10
11 class SuperWorker implements IWorker {
12     public void work() { //.... working much more}
13     public void eat() { //.... eating in launch break}
14 }
15
16 class Manager {
17     IWorker worker;
18     public void setWorker(IWorker w) {worker=w;}
19     public void manage() {
20         worker.work();
21     }
22 }
```

Interface Segregation

- **Principle**

Clients should not be forced to depend on abstractions they don't use

- **Refactoring**

Extract several *slim* interfaces

```java
interface IWorkable {
    public void work();
}

interface IFeedable{
    public void eat();
}

class Worker implements IWorkable, IFeedable{
    public void work() {// ....working}
    public void eat() {//.... eating in launch break}
}

class Robot implements IWorkable{
    public void work() {// ....working}
}

class SuperWorker implements IWorkable, IFeedable{
    public void work() {//.... working much more}
    public void eat() {//.... eating in launch break}
}
```

Dependency Inversion

- **Context**

I want to reuse a class in a similar context

- **Problem**

Reuse is not possible and I have to duplicate code

```
1  class Worker {
2      public void work() {// ....working}
3  }
4
5  class Manager {
6      Worker worker;
7      public void setWorker(Worker w) {worker = w;}
8      public void manage() {worker.work();}
9  }
10 class SuperWorker {
11     public void work() {//.... working much more}
12 }
```

Dependency Inversion

- **Principle**

Abstractions should not depend on details. Details should depend upon abstractions.

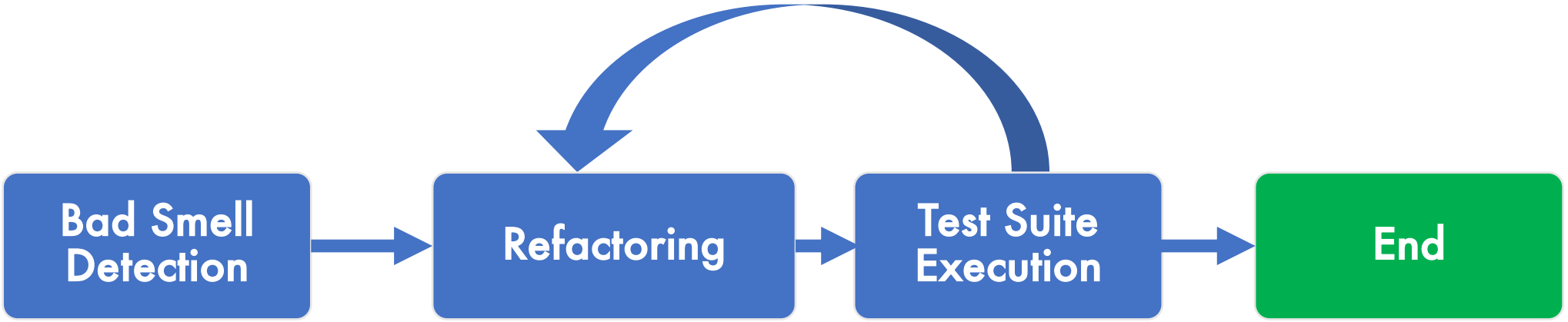
- **Refactoring**

Substitute object with interface.

Use Template Pattern.

```
1  interface IWorker {
2      public void work();
3  }
4
5  class Worker implements IWorker{
6      public void work() {// ....working}
7  }
8
9  class SuperWorker  implements IWorker{
10     public void work() {//.... working much more}
11 }
12
13 class Manager {
14     IWorker worker;
15     public void setWorker(IWorker w) {worker = w;}
16     public void manage() {worker.work();}
17 }
```

Refactoring Process



- Removing a bad smells can require more the one refactoring operation
- If you do not have a test suite, create it, then you can refactor the system

Outline

1 Introduction

2 Refactoring

3 Continuous Quality Improvement

Motivation for Continuous Integration

Context

- I am working on a complex code base together with other developers

Problem

- Avoid repetitive tasks without breaking existing code

Solution:

- ✓ Merging all developer working copies to a shared mainline (master) several times a day.
- ✓ Merging is verified by an automated process resulting in a build.

- Leverages:
  - (modern) SCM systems
  - Automated testing
  - Dedicated hardware
- Enables continuous deployment

"...but it works on my machine!?"

"What do you mean the tests are failing?"

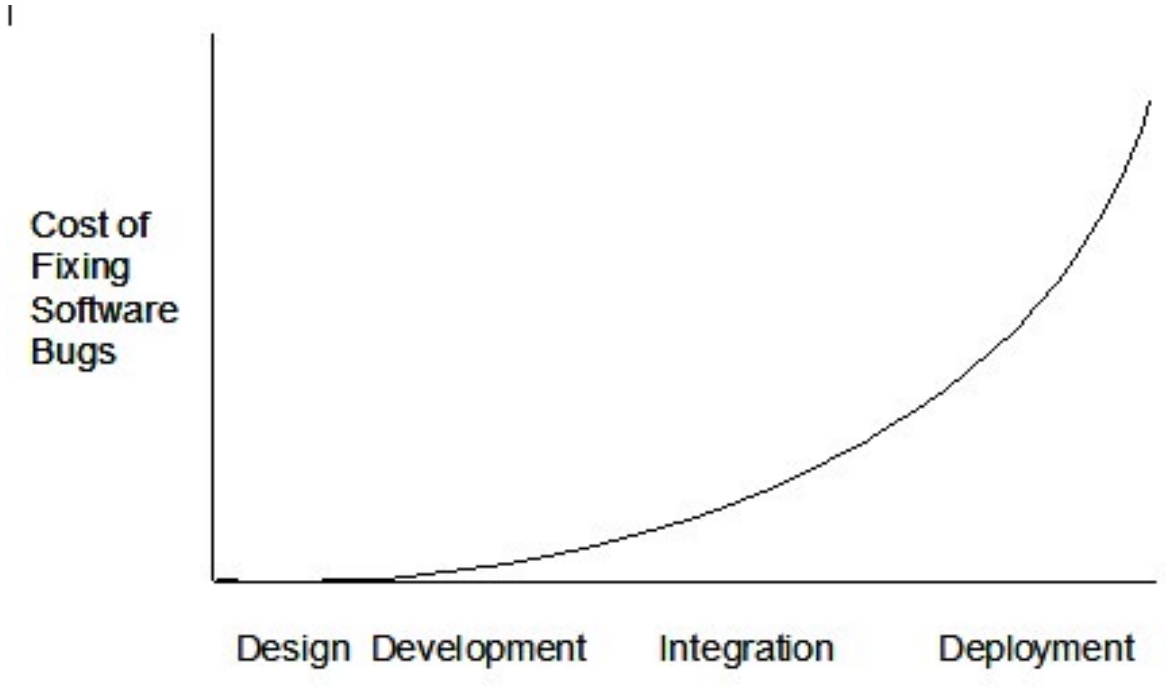
"Your changes to Foo.java are incompatible with mine? How do we merge?"

What's in the Build?

- **Build ≠ Source code compilation!**
- Compiled code
  - possibly for every target platform
- Results of test execution
- Database integration
  - automatic re-creation and seeding
- Code inspection
  - Code standards and best practice
- Documentation
  - Always current
  - Includes dev documentation (e.g., reports)

Continuous Integration Advantages

- Identify defect earlier & fix when it is still cheap
- Test run in parallel by each developer (unit-testing)
- Integration test run often



General Build Patterns

Problem

- When to build?

Solution

- Run a build with every change applied to the repository

General Build Patterns

Problem

- How granular is the change?

Solution

- Organize source code changes by task-oriented units of work and submit changes as a Task-Level Commit

General Build Patterns

Problem

- Such granularity produces a lot of builds, how to recognize them?

Solution

- Tag or Label the build with unique name so that you can refer to run the same build at another time.

Build Management Patterns

Problem

- How to efficiently manage a build?

Solution

- Automate all activities to build software from a source without manual configuration.

Build Management Patterns

Problem

- How to automate builds?

Solution

- Create build scripts that will be executed by an ad-hoc server so that software is built at every change

Build Quality Patterns

Problem

- How to establish a quality baseline for the build?

Solution

- Write automated tests for each code path, both success testing and failure testing.

Build Quality Patterns

Problem

- How to decide if the build is stable?

Solution

- Create smoke tests that can be used by CI servers, developers, QA, and testing as a pre-check to confirm the most important functionality.

Build Quality Patterns

Problem

- What to do when the build is not stable?

Solution

- Fail a build when a project rule is violated.
- Notify team members of problems such as low code coverage or the use of anti-patterns.

Measuring Software Product Quality

Code Level

- Object oriented
- Static code attributes
- Churn metrics

Business Level

- Cost/Schedule
- User satisfaction

Measuring Software Product Quality

Code-level metrics

- Number in failures (functional defects) of the released product

- Mean Time Between Failures

- Fault density

  Faults per SW size (Line of Code, Function Points)

Issues with Metrics

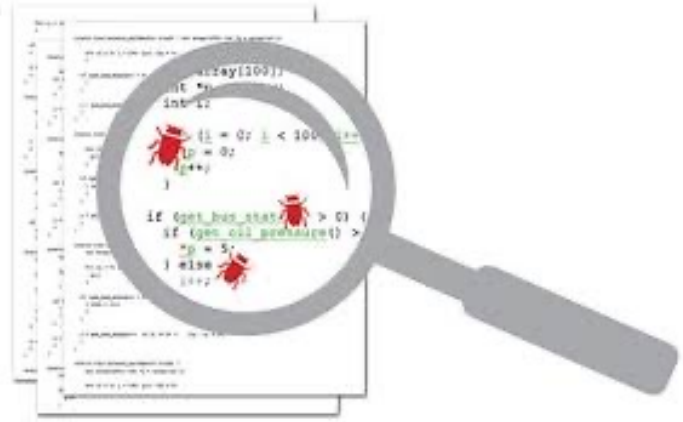
- Extreme caution is necessary when comparing two products
  - Definitions of LOC, defects, and time frames are not the same

- Time frame – L.O.P – Life of Product
  - Experience with operating systems shows that 95% of software defects are found within four years
  - Experience with application software shows two years

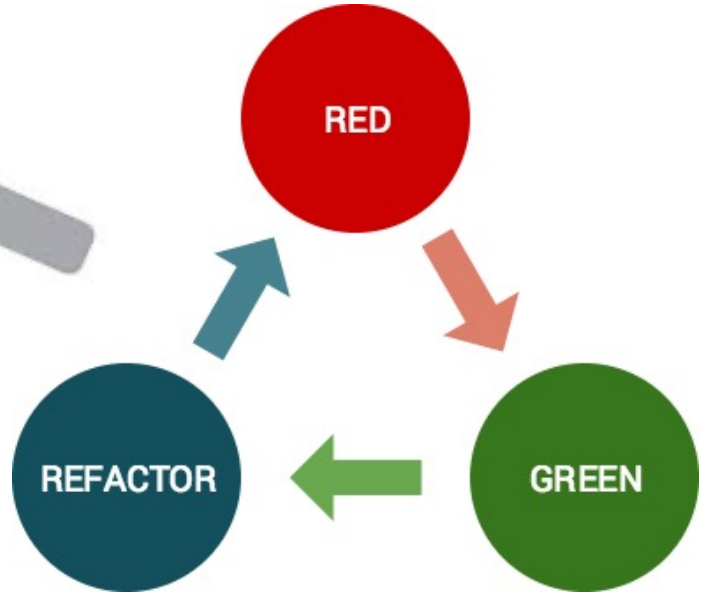
Another View of Software Product Quality: Static Analysis

- Checking a program for errors without executing it (it is NOT testing)
- A static code analyzer looks for patterns (e.g., rules) which *could* result in quality problems and vulnerabilities.
- Bug patterns arise due to:
  - Difficult language features
  - Misunderstood API methods
  - Misunderstood changes during maintenance
  - Typos
- Fast & Automated
- Relatively high (~50%) rate of *false warnings*

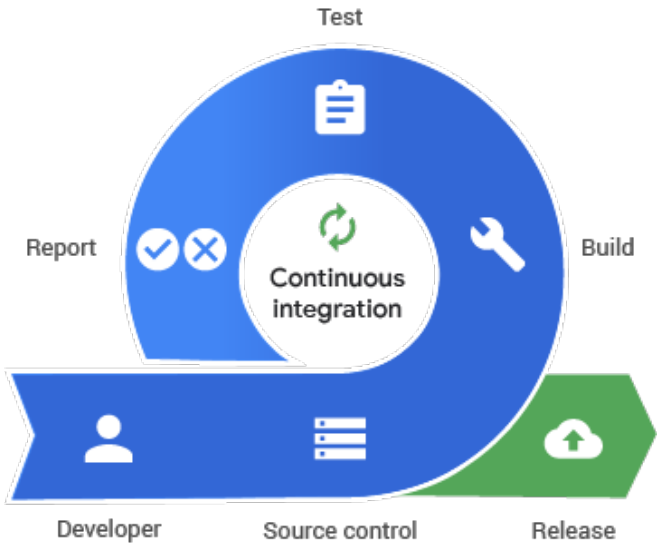
A Formula for Continuous Quality



Static Analysis



Automated Testing

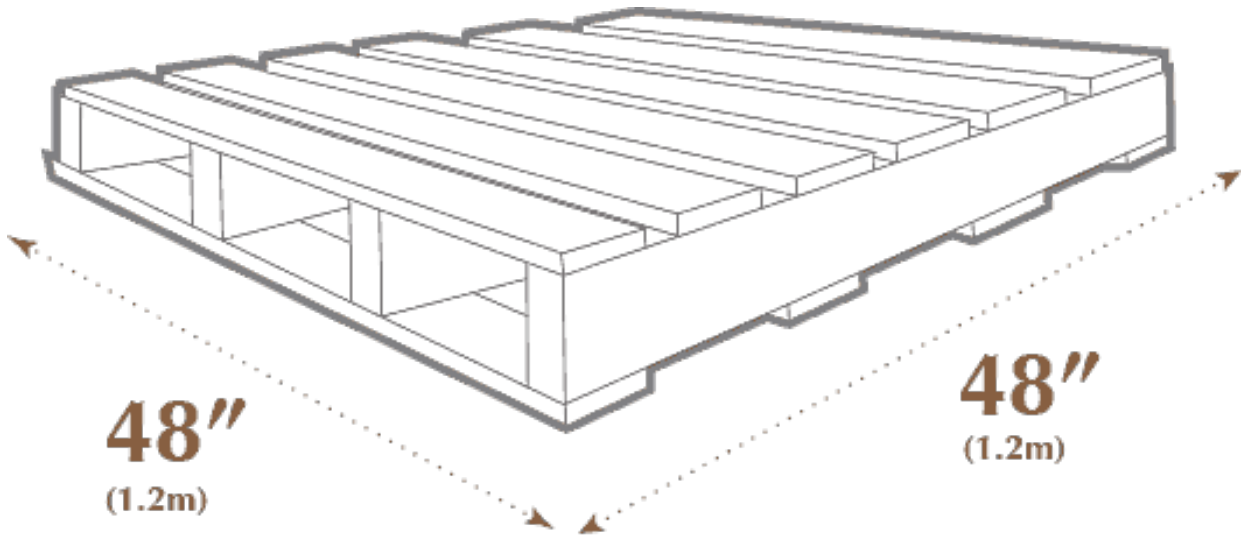


Continuous Integration

Continuous Quality (7 dimensions)

- Potential bugs
- Coding rules
- Duplications
- Complexity

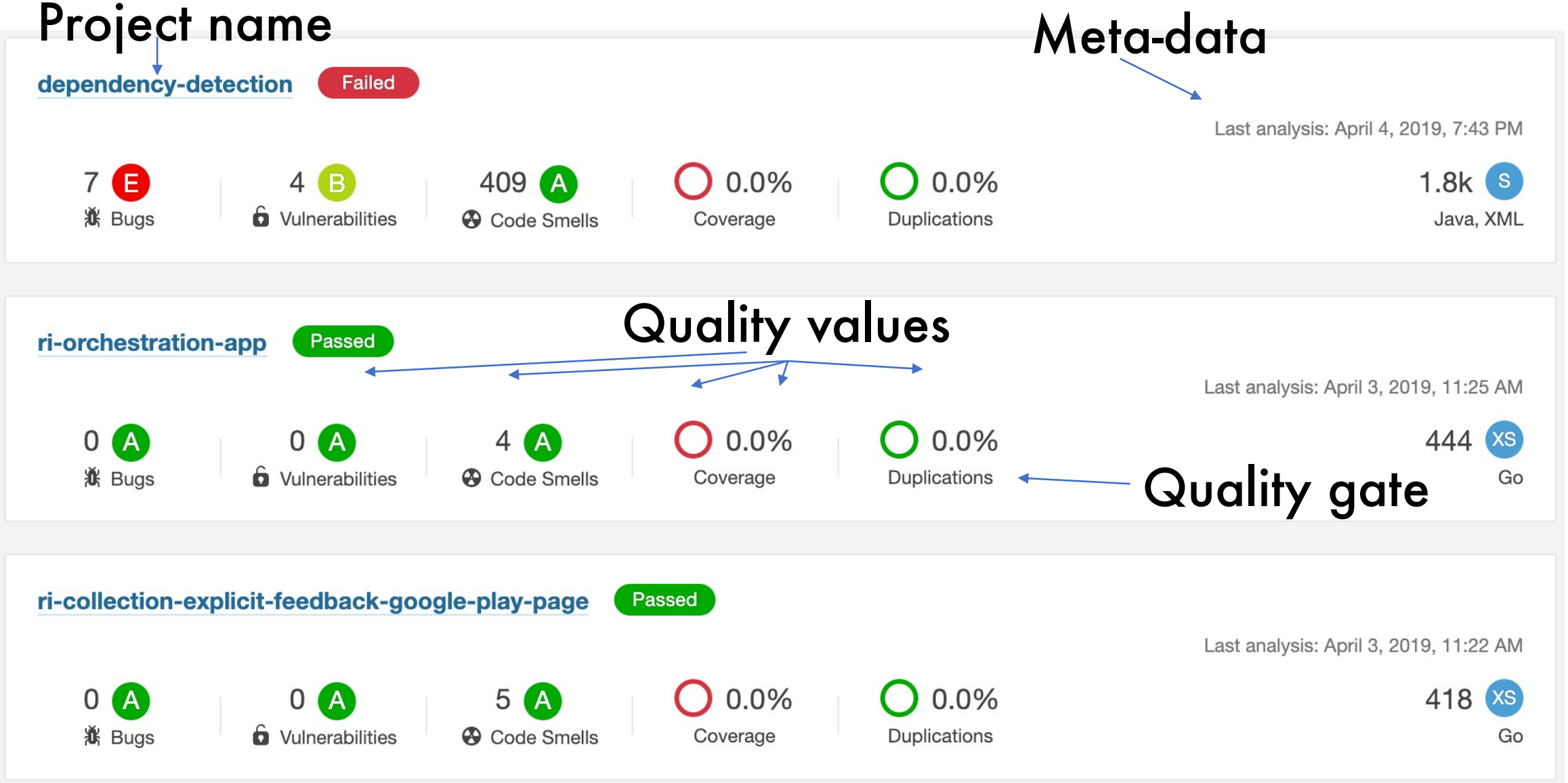
- Test coverage
- Architecture and design
- Comments



Continuous Quality Monitoring with SonarQube

- SonarQube analyzes code compliance against a set of rules.
  - If the code violates some rules, SonarQube adds the time needed to refactor it
- SonarQube also identifies a set of rules as bugs, claiming that they "represent something wrong in the code and will soon be reflected in a fault"
- They claim zero false positives from bugs
- SonarQube has been adopted by more than 85K organizations including nearly 15K public open-source projects

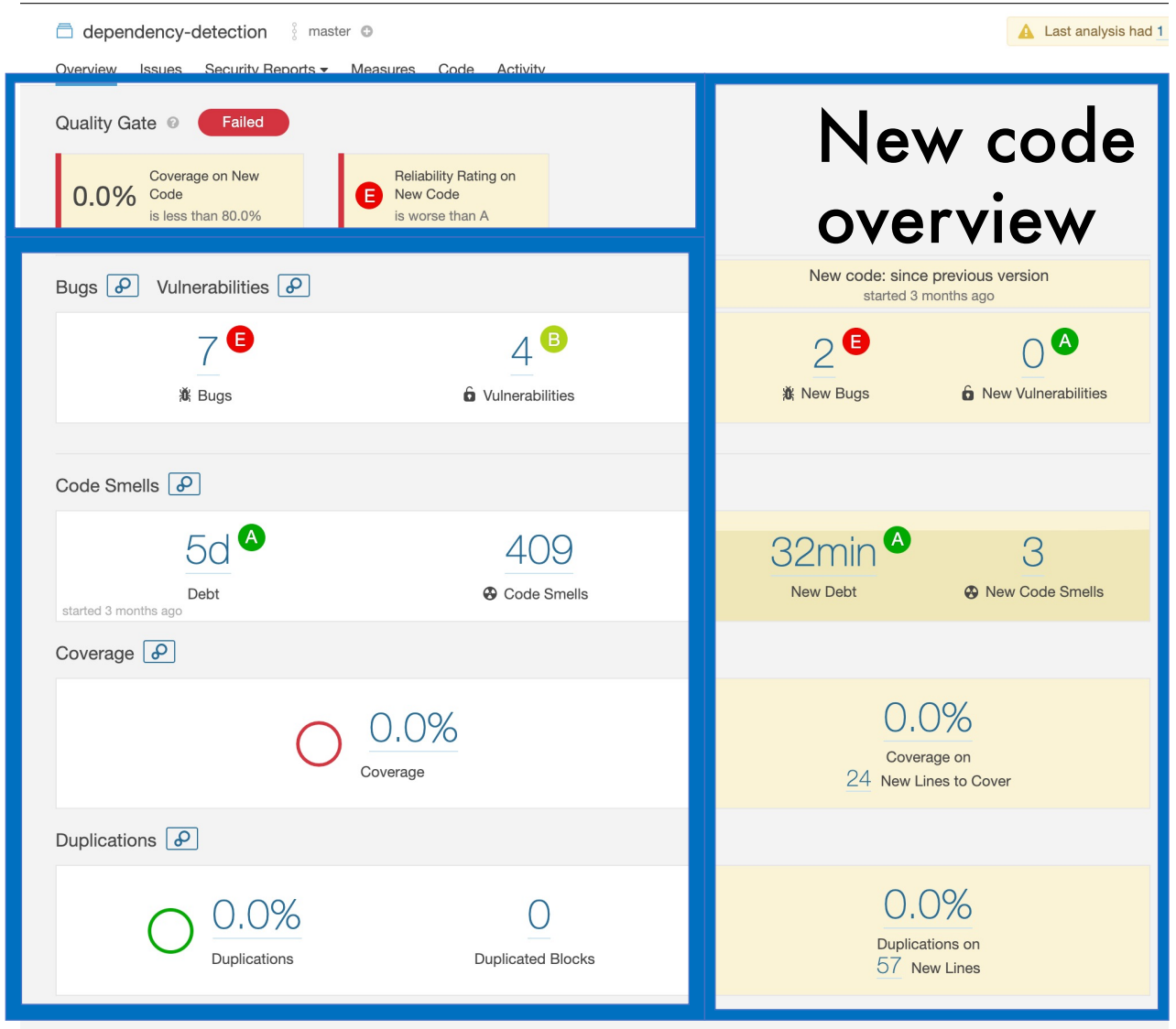
Projects Overview in SonarQube



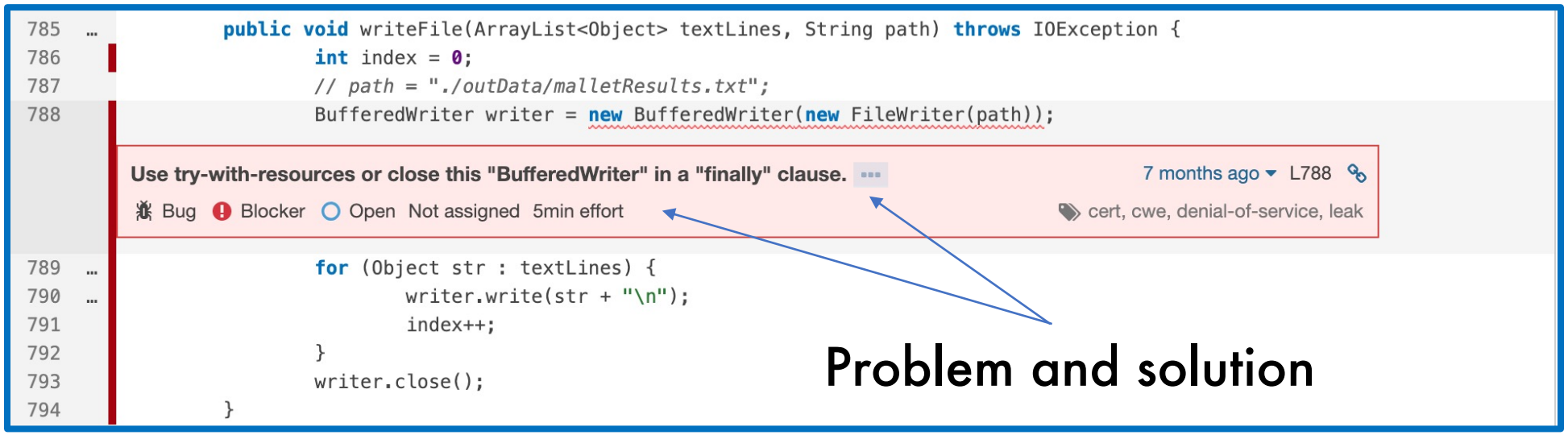
Project Details in SonarQube

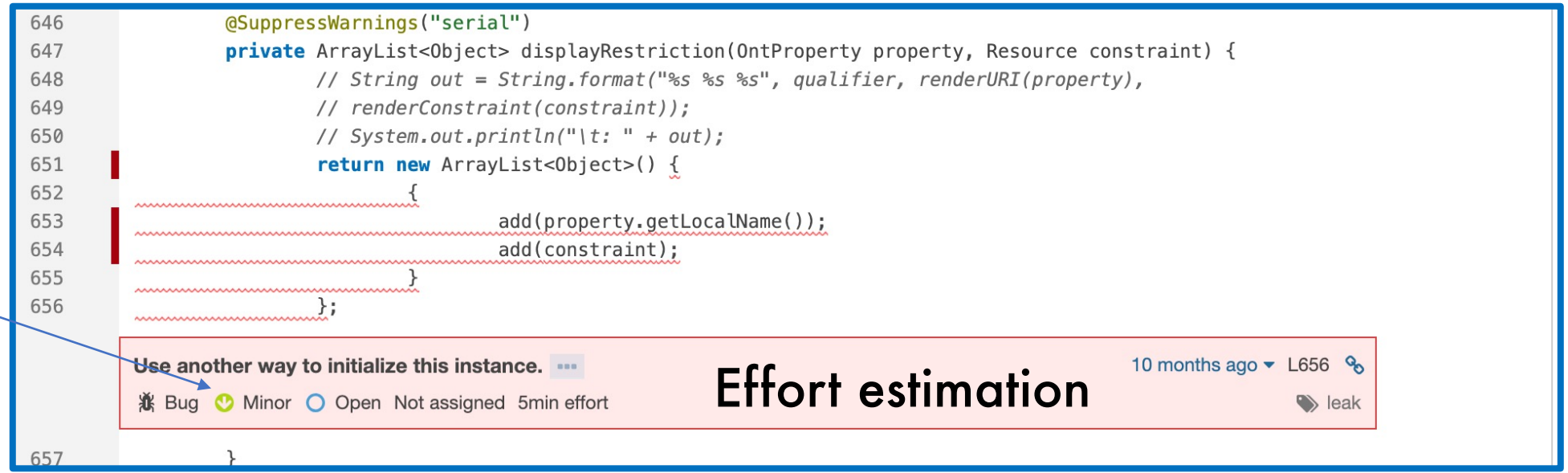
Failure reasons

Overall project view



Bugs in SonarQube





Severity

Bugs in SonarQube

```
785 …       public void writeFile(ArrayList<Object> textLines
786             int index = 0;
787             // path = "./outData/malletResults.txt";
788             BufferedWriter writer = new BufferedWrite
```

Use try-with-resources or close this "BufferedWriter" in a "finally"

Bug Blocker Open Not assigned 5min effort

```
789 …           for (Object str : textLines) {
790 …               writer.write(str + "\n");
791                 index++;
792             }
793             writer.close();
794         }
```

```
646         @SuppressWarnings("serial")
647         private ArrayList<Object> displayRestriction(OntP
648             // String out = String.format("%s %s %s",
649             // renderConstraint(constraint));
650             // System.out.println("\t: " + out);
651             return new ArrayList<Object>() {
652                     {
653                         add(property.getLocalName
654                         add(constraint);
655                     }
656                 };
```

Severity

Use another way to initialize this instance. …

Bug Minor Open Not assigned 5min effort

Effort es

Resources should be closed

squid:S2095

Bug Blocker Main sources cert, cwe, denial-of-service, leak

Available Since Feb 11, 2019 SonarAnalyzer (Java) Constant/issue: 5min

Connections, streams, files, and other classes that implement the `Closeable` interface or its super-interface, `AutoCloseable`, needs to be closed after use. Further, that `close` call must be made in a `finally` block otherwise an exception could keep the call from being made. Preferably, when class implements `AutoCloseable`, resource should be created using "try-with-resources" pattern and will be closed automatically.

Failure to properly close resources will result in a resource leak which could bring first the application and then perhaps the box it's on to their knees.

Noncompliant Code Example

```
private void readTheFile() throws IOException {
  Path path = Paths.get(this.fileName);
  BufferedReader reader = Files.newBufferedReader(path, this.charset);
  // ...
  reader.close();  // Noncompliant
  // ...
  Files.lines("input.txt").forEach(System.out::println); // Noncompliant: The stream
}

private void doSomething() {
  OutputStream stream = null;
  try {
    for (String property : propertyList) {
      stream = new FileOutputStream("myfile.txt");  // Noncompliant
      // ...
    }
  } catch (Exception e) {
    // ...
  } finally {
    stream.close();  // Multiple streams were opened. Only the last is closed.
  }
}
```

Compliant Solution

```
private void readTheFile(String fileName) throws IOException {
    Path path = Paths.get(fileName);
    try (BufferedReader reader = Files.newBufferedReader(path, StandardCharsets.UTF_
      reader.readLine();
      // ...
    }
    // ..
    try (Stream<String> input = Files.lines("input.txt"))  {
```

Vulnerabilities in SonarQube

```
351 …               try {
352 …                       if (sentenceDetector == null) {
353 …                               SentenceModel model = new SentenceModel(inputStream);
354
355                                 // Instantiating the SentenceDetectorME class
356 …                               sentenceDetector = new SentenceDetectorME(model);
357                         }
358 …                       // Detecting the sentence
359 …                       sentences = sentenceDetector.sentDetect(sentence);
360 …               } catch (IOException e) {
361                         e.printStackTrace();
```

Use a logger to log this exception. ▪▪▪ 10 months ago ▾ L361

Vulnerability Minor Open Not assigned 10min effort — cwe, error-handling

```
362             } finally {
363                     if (inputStream != null) {
364                             try {
365                                     inputStream.close();
366                             } catch (IOException e) {
367                                     e.printStackTrace();
```

Use a logger to log this exception. ▪▪▪ 10 months ago ▾ L367

Vulnerability Minor Open Not assigned 10min effort — cwe, error-handling

```
368                             }
369                     }
370             }
371 …           return sentences;
```

Code Smells in SonarQube

```
142 …         @PostMapping("/json/ontology/{projectId}/{synonymy}/{threshold}")
143 …         @ApiOperation(value = "Uploads JSON and Ontology files to detect dependencies", notes = "Uploads an ontology (in RDF/
144 …         @ApiResponses(value = { @ApiResponse(code = 0, message = "Non content: There is no content to submit."),
145                       @ApiResponse(code = 200, message = "OK: The request has succeeded."),
146 …                     @ApiResponse(code = 201, message = "Created: The request has been fulfilled and has resulted in one c
147                       @ApiResponse(code = 401, message = "Unauthorized: The request has not been applied because it lacks v
148                       @ApiResponse(code = 403, message = "Forbidden: The server understood the request but refuses to authc
149                       @ApiResponse(code = 404, message = "Not Found: The server could not find what was requested by the cl
150                       @ApiResponse(code = 500, message = "Internal Server Error. For more information see 'message' in the
151 …         public ResponseEntity<?> uploadJSONFile(
```

Remove usage of generic wildcard type. 10 months ago · L151

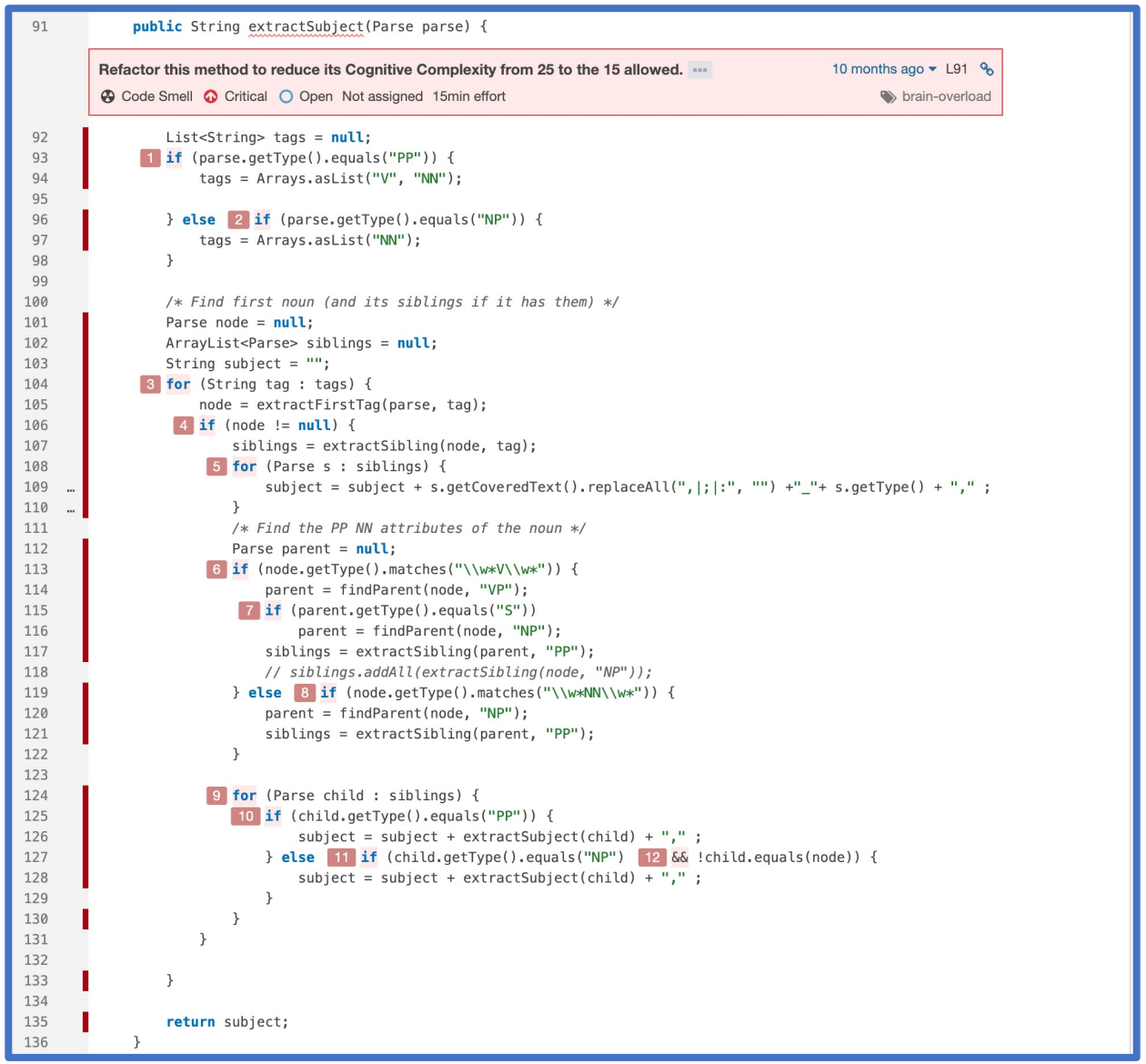
Code Smell · Critical · Open · Not assigned · 20min effort · pitfall

```
152 …                     @ApiParam(value = "The Ontology file to upload (RDF/XML lang.)", required = true) @RequestPart("ontol
153                       @ApiParam(value = "The JSON file to upload", required = true) @RequestPart("json") @Valid String json
154 …                     @ApiParam(value = "Id of the project where the requirements to analize are.", required = true) @PathV
155 …                     @ApiParam(value = "Apply Semantic Similarity (Synonymy) detcetion (Type: Boolean).", required = true)
156                       @ApiParam(value = "Threshold of semantic similarity to detect synonyms (included).", required = true)
157                       throws IOException, InterruptedException {
158 …
159 …               Instant start = Instant.now();
160 …               long stopTime;
```

Remove this unused "stopTime" local variable. 10 months ago · L160

Code Smell · Minor · Open · Not assigned · 5min effort · unused

Complexity in SonarQube




```
91      public String extractSubject(Parse parse) {
92          List<String> tags = null;
93      1   if (parse.getType().equals("PP")) {
94              tags = Arrays.asList("V", "NN");
95
96          } else  2  if (parse.getType().equals("NP")) {
97              tags = Arrays.asList("NN");
98          }
99
100         /* Find first noun (and its siblings if it has them) */
101         Parse node = null;
102         ArrayList<Parse> siblings = null;
103         String subject = "";
104     3   for (String tag : tags) {
105             node = extractFirstTag(parse, tag);
106         4   if (node != null) {
107                 siblings = extractSibling(node, tag);
108             5   for (Parse s : siblings) {
109                     subject = subject + s.getCoveredText().replaceAll(",|;|:", "") +"_"+ s.getType() + "," ;
110                 }
111                 /* Find the PP NN attributes of the noun */
112                 Parse parent = null;
113             6   if (node.getType().matches("\\w*V\\w*")) {
114                     parent = findParent(node, "VP");
115                 7   if (parent.getType().equals("S"))
116                         parent = findParent(node, "NP");
117                     siblings = extractSibling(parent, "PP");
118                     // siblings.addAll(extractSibling(node, "NP"));
119                 } else  8  if (node.getType().matches("\\w*NN\\w*")) {
120                     parent = findParent(node, "NP");
121                     siblings = extractSibling(parent, "PP");
122                 }
123
124             9   for (Parse child : siblings) {
125                 10  if (child.getType().equals("PP")) {
126                         subject = subject + extractSubject(child) + "," ;
127                     } else  11  if (child.getType().equals("NP")  12  && !child.equals(node)) {
128                         subject = subject + extractSubject(child) + "," ;
129                     }
130                 }
131             }
132
133         }
134
135         return subject;
136     }
```

Design Smells in SonarQube

CouplingBetweenObjects

Since: PMD 1.04

Priority: Medium (3)

This rule counts the number of unique attributes, local variables, and return types within an object. A number higher than the specified threshold can indicate a high degree of coupling.

This rule is defined by the following Java class: net.sourceforge.pmd.lang.java.rule.design.CouplingBetweenObjectsRule

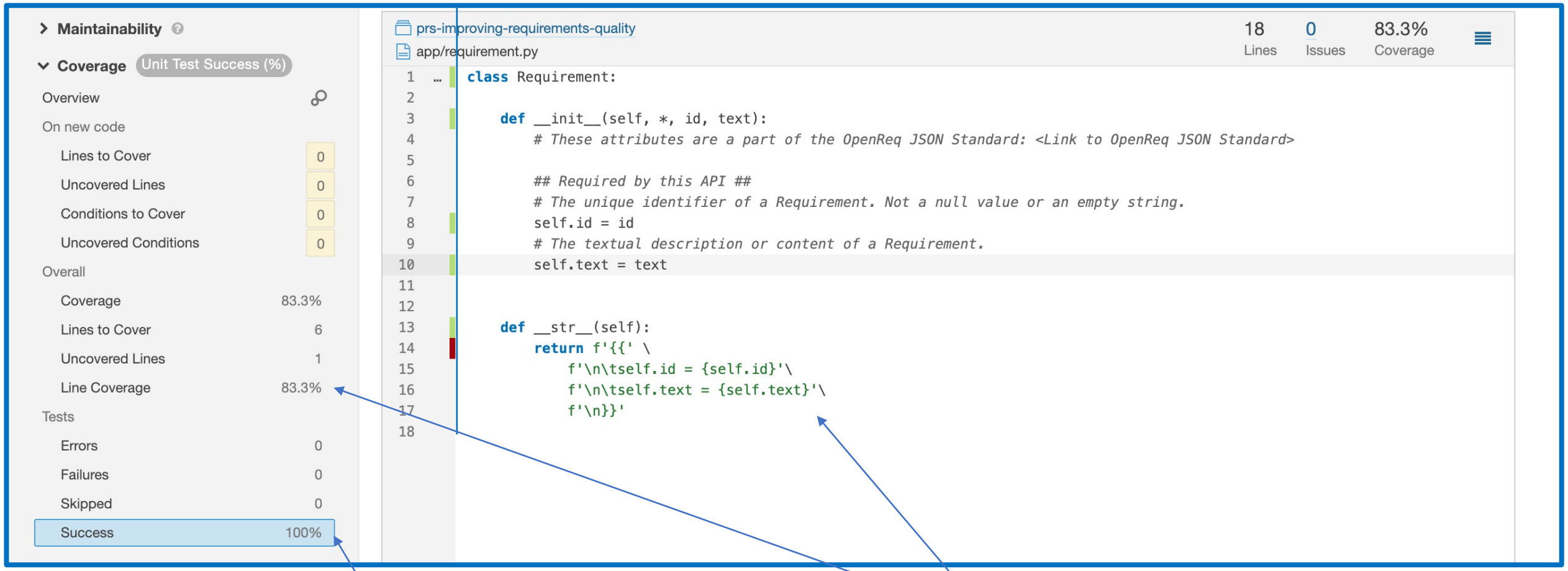
Example(s):

```
import com.Blah;
import org.Bar;
import org.Bardo;

public class Foo {
    private Blah var1;
    private Bar var2;

    //followed by many imports of unique objects
    void ObjectC doWork() {
        Bardo var55;
        ObjectA var44;
        ObjectZ var93;
        return something;
    }
}
```

This rule has the following properties:

Name	Default Value	Description	Multivalued
threshold	20	Unique type reporting threshold	no

Coverage in SonarQube



Test reports

Coverage report