Improve your tests with Mutation Testing

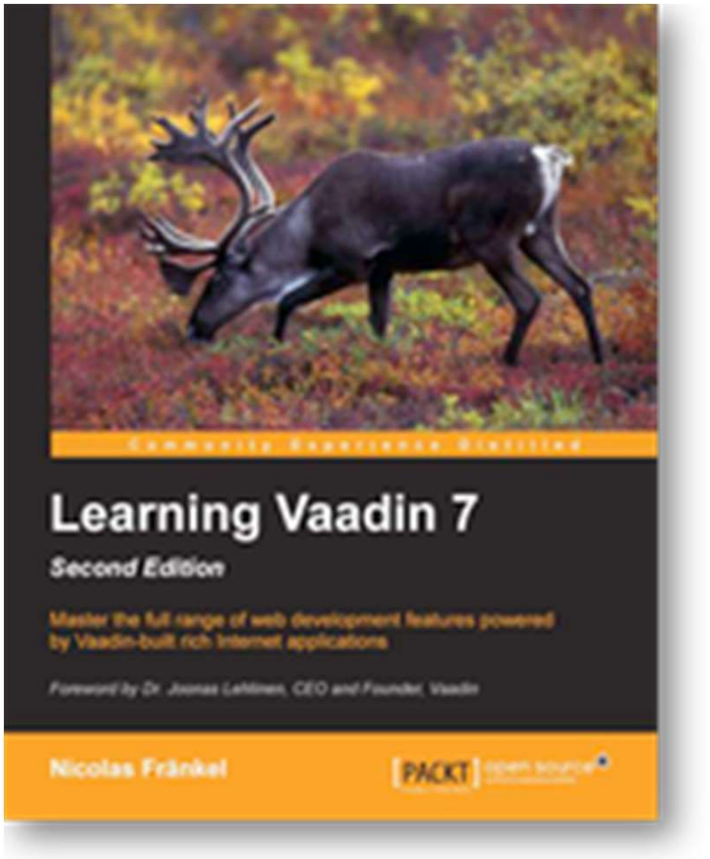
Nicolas Fränkel

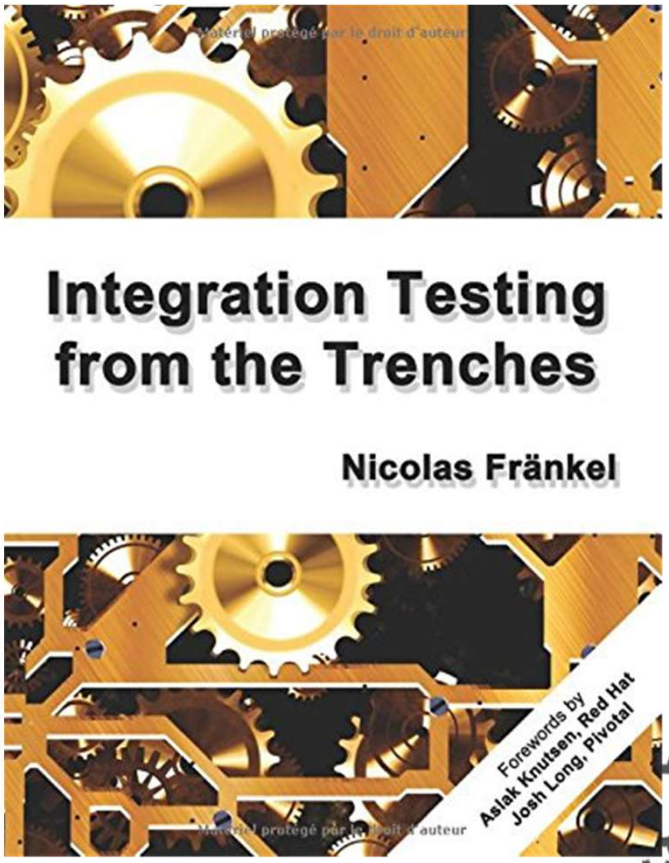
geecon

Me, Myself and I

- Developer & Architect
  - As Consultant
- Teacher/trainer
- Blogger
- Speaker
- Book Author

Shameless self-promotion





My job

hybris software
AN SAP COMPANY

Many kinds of testing

Unit Testing

Integration Testing

End-to-end Testing

Performance Testing

Penetration Testing

Exploratory Testing

etc.

Their only single goal

Ensure the Quality of the production code

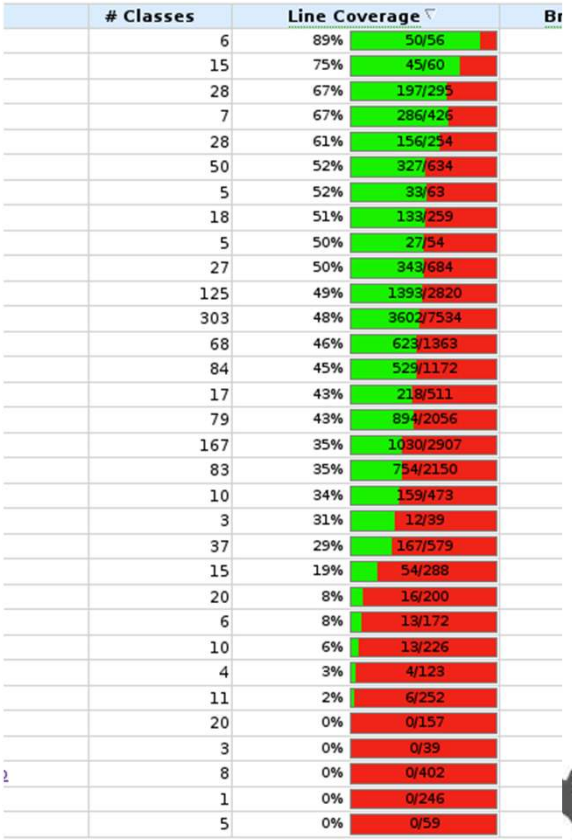
The problem

How to check the Quality of the testing code?

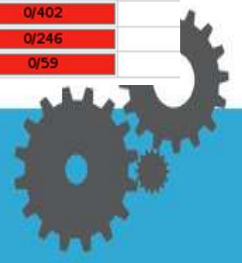
Code coverage

"Code coverage is a measure used to describe the degree to which the source code of a program is tested"

--Wikipedia
http://en.wikipedia.org/wiki/Code_coverage



# Classes	Line Coverage	
6	89%	50/56
15	75%	45/60
28	67%	197/295
7	67%	286/426
28	61%	156/254
50	52%	327/634
5	52%	33/63
18	51%	133/259
5	50%	27/54
27	50%	343/684
125	49%	1393/2820
303	48%	3602/7534
68	46%	623/1363
84	45%	529/1172
17	43%	218/511
79	43%	894/2056
167	35%	1030/2907
83	35%	754/2150
10	34%	159/473
3	31%	12/39
37	29%	167/579
15	19%	54/288
20	8%	16/200
6	8%	13/172
10	6%	13/226
4	3%	4/123
11	2%	6/252
20	0%	0/157
3	0%	0/39
8	0%	0/402
1	0%	0/246
5	0%	0/59



Measuring Code Coverage

Check whether a source code line is executed during a test

Or Branch Coverage

Computing Code Coverage

$$CC = \frac{L_{executed}}{L_{total}} * 100$$

CC: Code Coverage (in percent)

$L_{executed}$: Number of executed lines of code

L_{total}: Number of total lines of code

Java Tools for Code Coverage

JaCoCo

Clover

Cobertura

etc.

100% Code Coverage?

"Is 100% code coverage realistic? Of course it is. If you can write a line of code, you can write another that tests it."

Robert Martin (Uncle Bob)
https://twitter.com/unclebobmartin/status/559666205096673 28

Assert-less testing

```
@Test
public void add_should_add() {
    new Math().add(1, 1);
}
```

But, where is the assert?

As long as the Code Coverage is OK…

Code coverage as a measure of test quality

Any metric can be gamed!

Code coverage is a metric…

⇒ Code coverage can be gamed

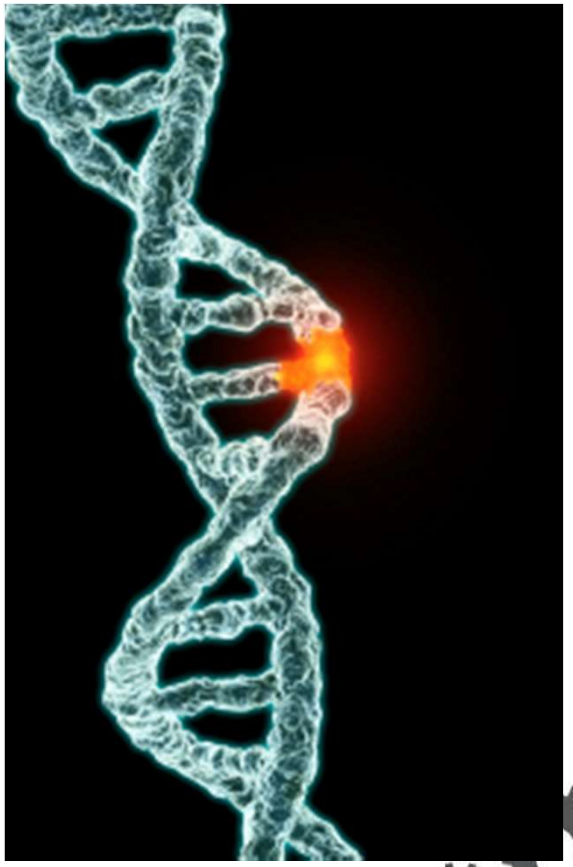
- On purpose
- Or by accident

Code coverage as a measure of test quality

Code Coverage lulls you into a false sense of security…

The problem still stands

- Code coverage cannot ensure test quality
  - Is there another way?
- Mutation Testing to the rescue!



The Cast

William Stryker

Original Source Code

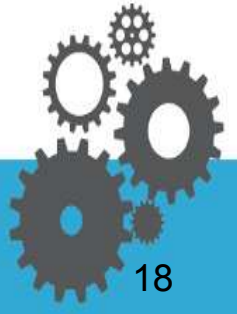
Jason Stryker

Modified Source Code

a.k.a "The Mutant"

```
public class Math {
  public int add(int i1, int i2) {
    return i1 + i2;
  }
}
```

```
public class Math {
  public int add(int i1, int i2) {
    return i1 - i2;
  }
}
```



Standard testing

Execute Test

Mutation testing

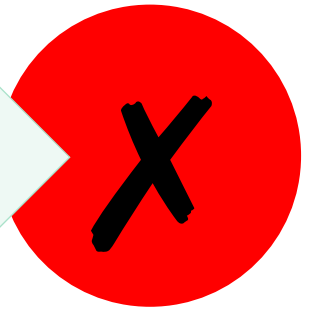
MUTATION

Execute SAME Test

?

Mutation testing

Execute SAME Test



Mutant Killed

Execute SAME Test

Mutant Survived

Killed or Surviving?

Surviving means changing the source code did not change the test result

It's bad!

Killed means changing the source code changed the test result

It's good

Test the code

```
public class Math {
  public int add(int i1, int i2) {
    return i1 + i2;
  }
}
```

Execute Test

```
@Test
public void add_should_add() {
    new Math().add(1, 1);
}
```

Surviving mutant

```
public class Math {
  public int add(int i1, int i2) {
    return i1 - i2;
  }
}
```

Execute SAME Test

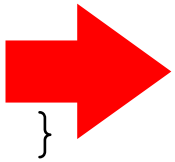
```
@Test
public void add_should_add() {
    new Math().add(1, 1);
}
```

Test the code

```
public class Math {
  public int add(int i1, int i2) {
    return i1 + i2;
  }
}
```

Execute Test

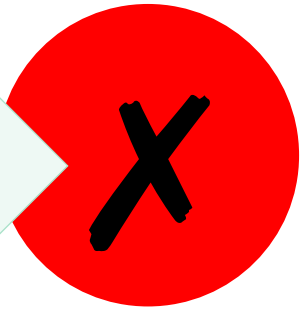
```
@Test
public void add_should_add() {
    int sum = new Math().add(1, 1);
    Assert.assertEquals(sum, 2);
}
```



Killed mutant

```
public class Math {
  public int add(int i1, int i2) {
    return i1 - i2;
  }
}
```

Execute SAME Test



```
@Test
public void add_should_add() {
    int sum = new Math().add(1, 1);
    Assert.assertEquals(sum, 2);
}
```

Mutation Testing in Java

PIT is a tool for Mutation testing

Available as

- Command-line tool
- Ant target
- Maven plugin

Mutators

Mutators are patterns applied to source code to produce mutations

PIT mutators sample

Name	Example source	Result
Conditionals Boundary	>	>=
Negate Conditionals	==	!=
Remove Conditionals	foo == bar	true
Math	+	-
Increments	foo++	foo--
Invert Negatives	-foo	foo
Inline Constant	static final FOO= 42	static final FOO = 43
Return Values	return true	return false
Void Method Call	System.out.println("foo")	
Non Void Method Call	long t = System.currentTimeMillis()	long t = 0
Constructor Call	Date d = new Date()	Date d = null;

Important mutators

Conditionals Boundary

Probably a potential serious bug smell

```
if (foo > bar)
```

Important mutators

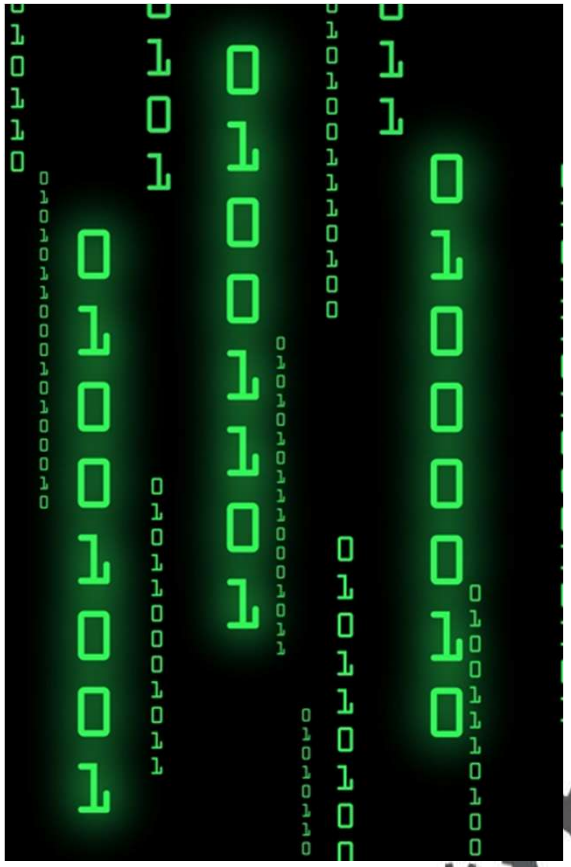
Void Method Call

```
Assert.checkNotNull()
connection.close()
```

Remember

It’s not because the IDE generates code safely that it will never change

- `equals()`
- `hashCode()`



False positives

Mutation Testing is not 100% bulletproof

Might return false positives

Be cautious!

Enough talk…

Simple PIT plugin call

```
mvn \
org.pitest:pitest-maven:mutationCoverage \
    -DtimestampedReports=false
```

Excluding some classes

```
mvn \
org.pitest:pitest-maven:mutationCoverage \
-DexcludedClasses=\
    org.joda.money.TestBigMoney,\
org.joda.money.TestCurrencyUnit,\
org.joda.money.TestMoney \
    -DtimestampedReports=false
```

Reports for Sonar

```
mvn \
org.pitest:pitest-maven:mutationCoverage \
-DexcludedClasses=\
    org.joda.money.TestBigMoney,\
org.joda.money.TestCurrencyUnit,\
org.joda.money.TestMoney \
-DoutputFormats=XML \
    -DtimestampedReports=false
```

Sending data to Sonar

```
mvn sonar:sonar \
-Dsonar.pitest.mode=reuseReport
```

Drawbacks

Slow

Sluggish

Crawling

Sulky

Lethargic

etc.

Metrics (kind of)

On *joda-money*

```
mvn clean test-compile
mvn surefire:test
```

Total time: 2.181 s

```
mvn pit-test...
```

Total time: 48.634 s

Why so slow?

- Analyze test code
- For each class under test
  - For each mutator
    - Create mutation
  - For each mutation
    - Run test
    - Analyze result
    - Aggregate results

Workarounds

This is not acceptable in a normal test run

But there are workarounds

Set mutators

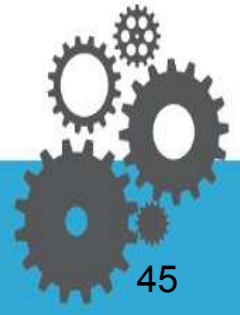
```
<configuration>
  <mutators>
    <mutator>
      CONSTRUCTOR_CALLS
    </mutator>
    <mutator>
      NON_VOID_METHOD_CALLS
    </mutator>
  </mutators>
</configuration>
```

Set target classes

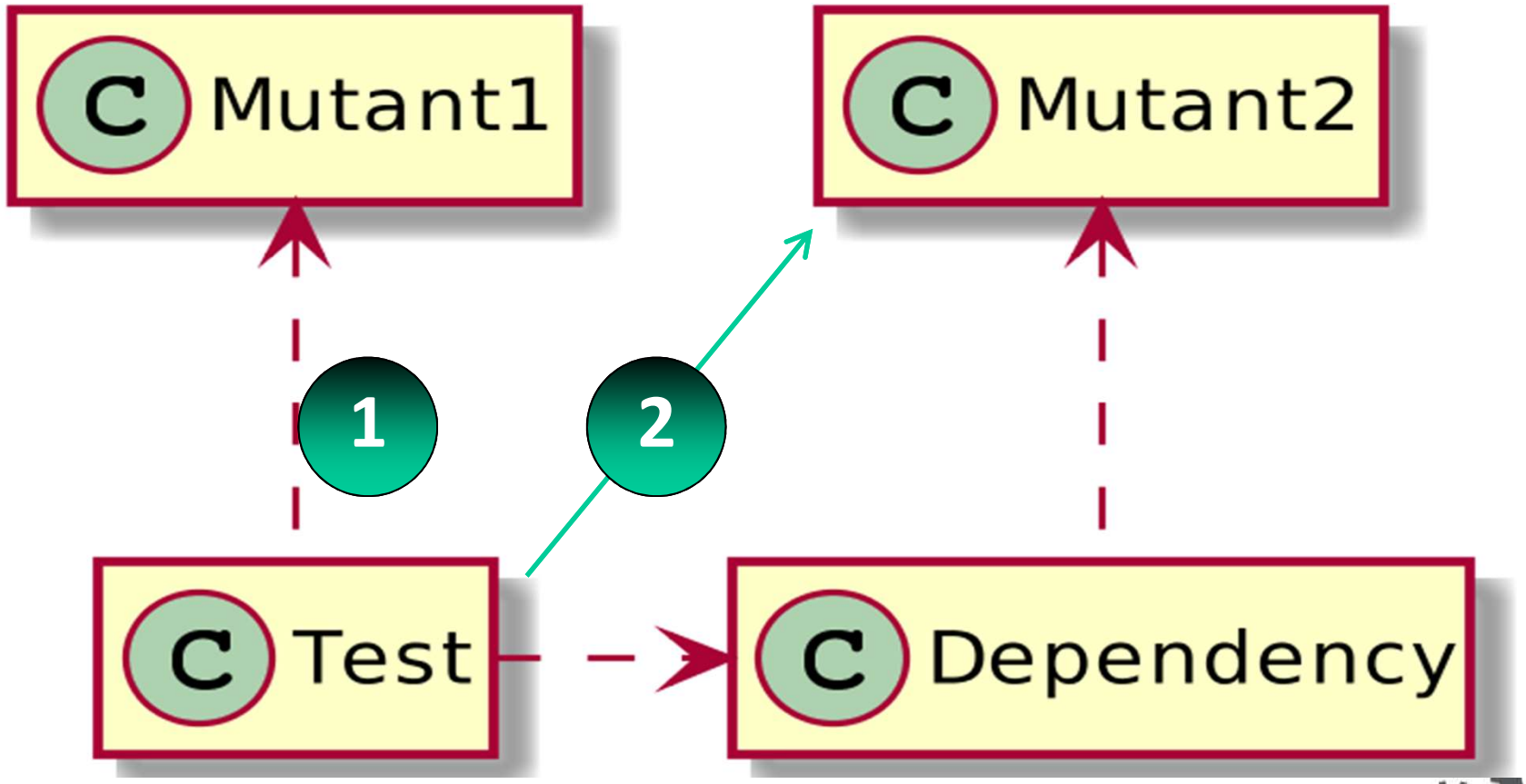
```
<configuration>
  <targetClasses>
    <param>ch.frankel.pit*</param>
  </targetClasses>
</configuration>
```

Set target tests

```
<configuration>
  <targetTests>
    <param>ch.frankel.pit*</param>
  </targetTests>
</configuration>
```



Dependency distance

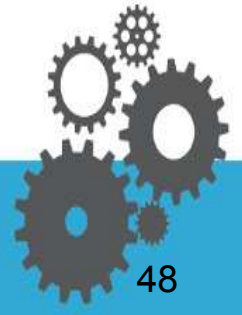


Limit dependency distance

```
<configuration>
  <maxDependencyDistance>
    4
  </maxDependencyDistance>
</configuration>
```

Limit number of mutations

```
<configuration>
  <maxMutationsPerClass>
    10
  </maxMutationsPerClass>
</configuration>
```



Use MutationFilter

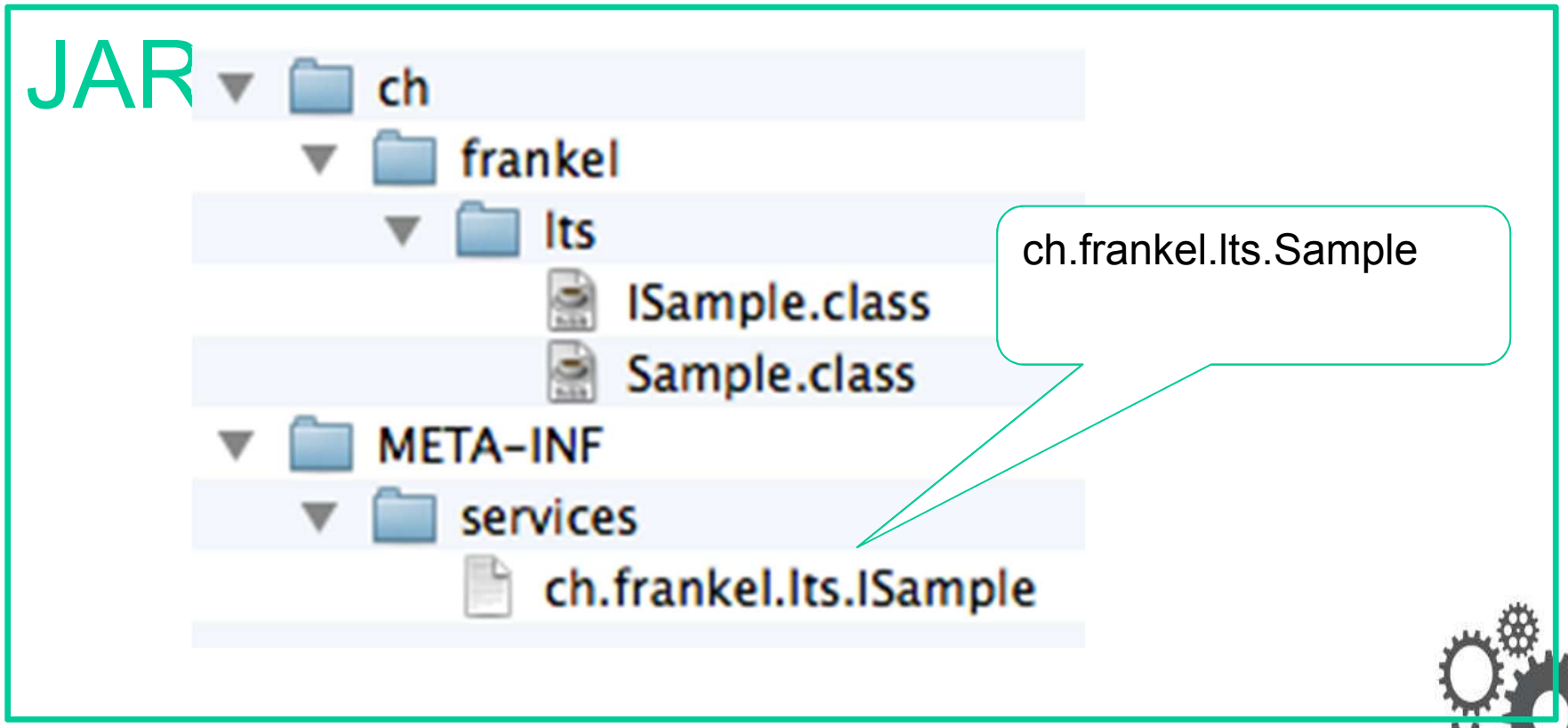
PIT extension points

- Must be packaged in JAR
- Have Implementation-Vendor and Implementation-Title in MANIFEST.MF that match PIT's
- Set on the classpath
- Use Java's Service Provider feature

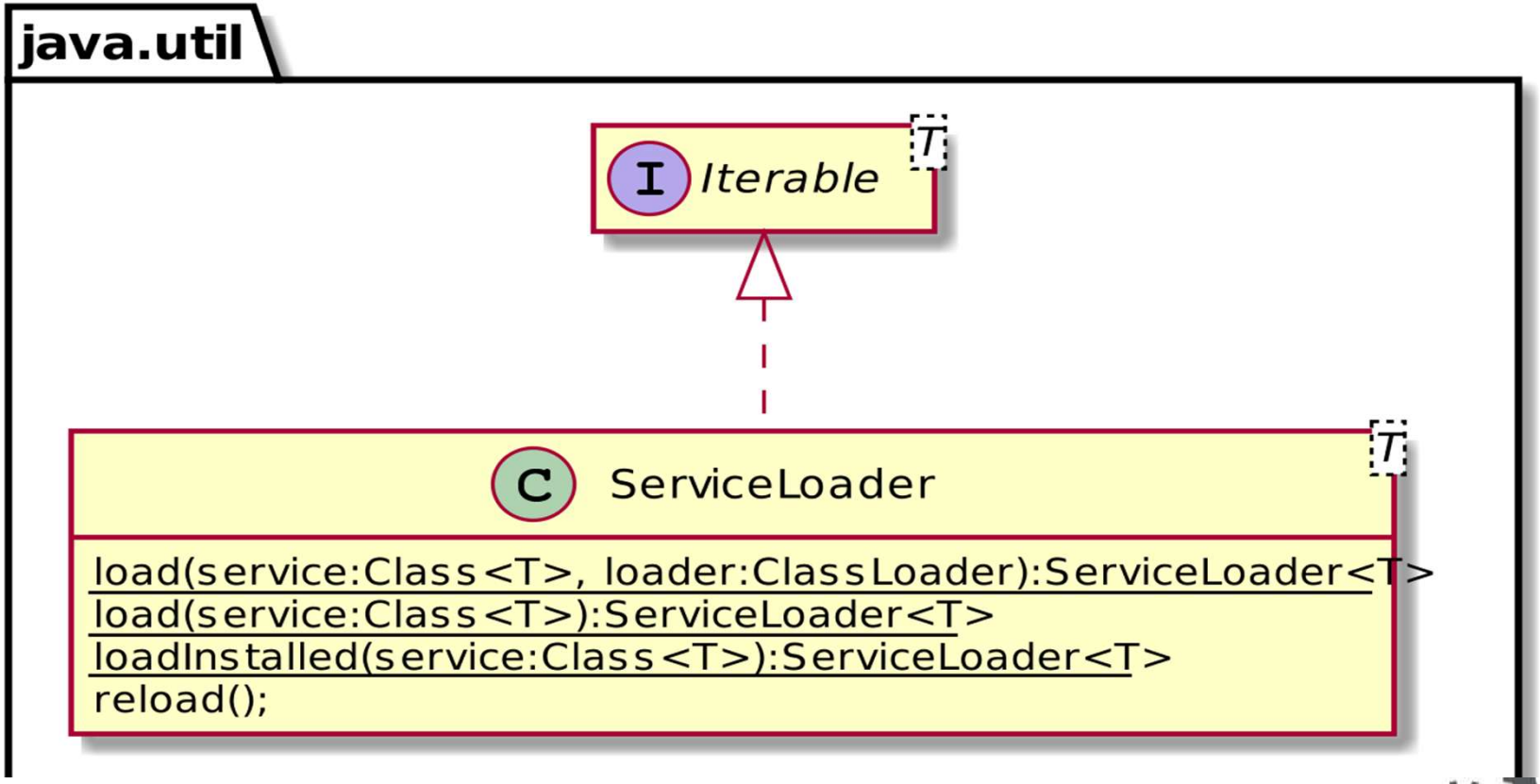
Service Provider

- Inversion of control
  - Since Java 1.3!
- Text file located in `META-INF/services`
- Interface
  - Name of the file
- Implementation class
  - Content of the file

Sample structure



Service Provider API



Service Provider sample

```
ServiceLoader<ISample> loaders =
    ServiceLoader.load(ISample.class);
for (ISample sample: loaders) {
    // Use the sample
}
```

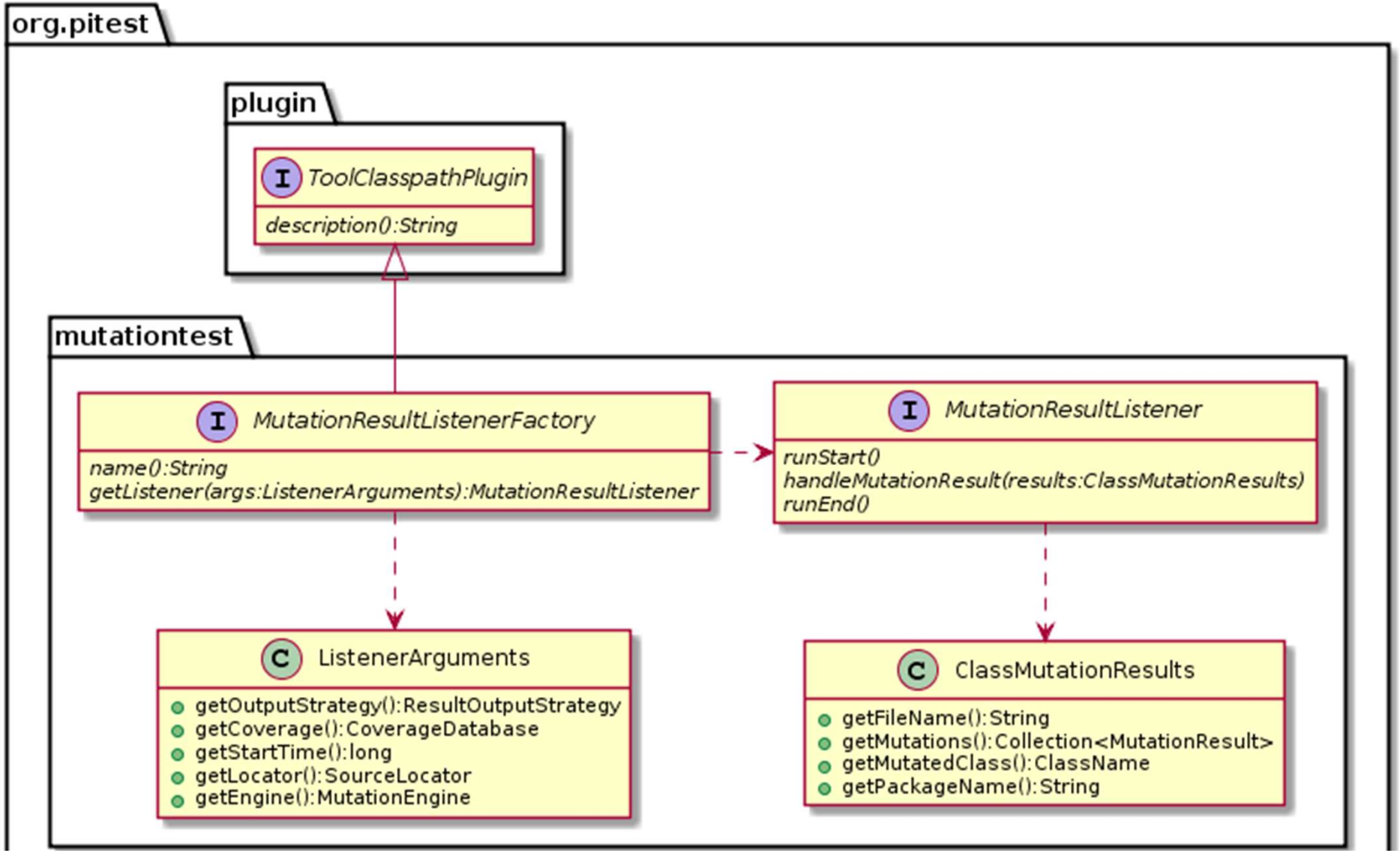
Output formats

Out-of-the-box

- HTML
- XML
- CSV

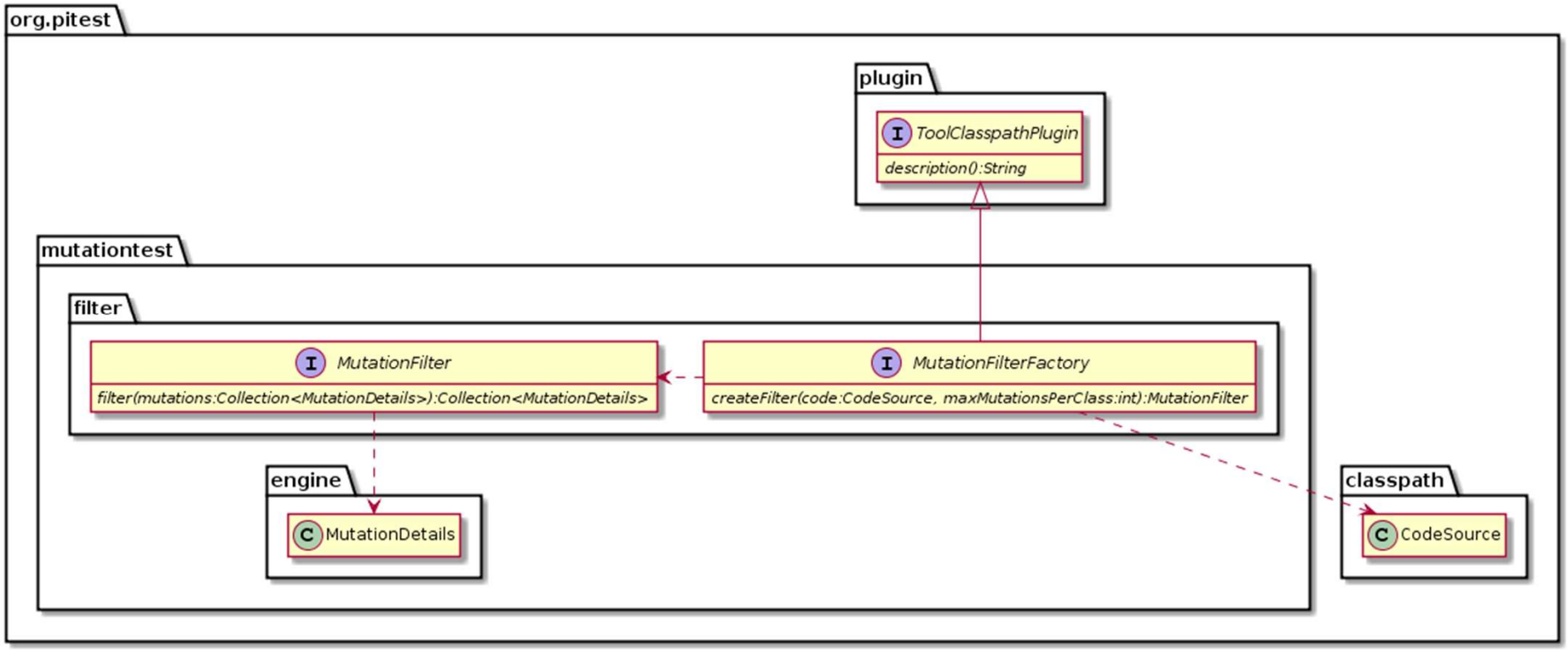
Output formats

```
<configuration>
  <outputFormats>
    <outputFormat>XML</outputFormat>
    <outputFormat>HTML</outputFormat>
  </outputFormats>
</configuration>
```


org.pitest
plugin
ToolClasspathPlugin
description():String
mutationtest
MutationResultListenerFactory
name():String
getListener(args:ListenerArguments):MutationResultListener
MutationResultListener
runStart()
handleMutationResult(results:ClassMutationResults)
runEnd()
ListenerArguments
getOutputStrategy():ResultOutputStrategy
getCoverage():CoverageDatabase
getStartTime():long
getLocator():SourceLocator
getEngine():MutationEngine
ClassMutationResults
getFileName():String
getMutations():Collection<MutationResult>
getMutatedClass():ClassName
getPackageName():String

Mutation Filter

Remove mutations from the list of available mutations for a class


org.pitest
plugin
ToolClasspathPlugin
description():String
mutationtest
filter
MutationFilter
filter(mutations:Collection<MutationDetails>):Collection<MutationDetails>
MutationFilterFactory
createFilter(code:CodeSource, maxMutationsPerClass:int):MutationFilter
engine
MutationDetails
classpath
CodeSource


Time for DEMO

Don’t bind to test phase!

```
<plugin>
  <groupId>org.pitest</groupId>
  <artifactId>pitest-maven</artifactId>
  <executions>
    <execution>
      <goals>
        <goal>mutationCoverage</goal>
      </goals>
      <phase>test</phase>
    </execution>
  </executions>
</plugin>
```

Use scmMutationCoverage

```
mvn \
org.pitest:pitest-maven:scmMutationCoverage \
-DtimestampedReports=false
```

Do use on Continuous Integration servers

```
mvn \
org.pitest:pitest-maven:mutationCoverage \
-DtimestampedReports=false
```

Is Mutation Testing the Silver Bullet?

Sorry, no!

It only

- Checks the relevance of your unit tests
- Points out potential bugs

What it doesn't do

Validate the assembled application

- Integration Testing

Check the performance

- Performance Testing

Look out for display bugs

- End-to-end testing

Etc.

Testing is about ROI

Don’t test to achieve 100% coverage

Test because it saves money in the long run

Prioritize:

- Business-critical code
- Complex code

Q&A

@nicolas_frankel

http://blog.frankel.ch/

https://leanpub.com/integrationtest/