Using Mutation Testing to Weed Out Fake Unit Tests

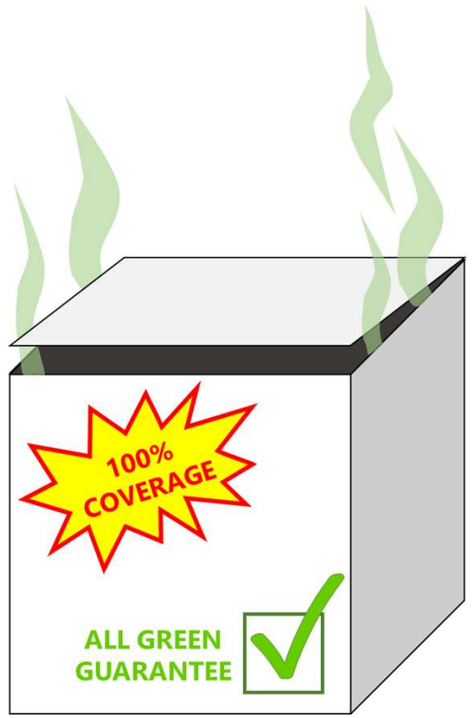
Code Kata

What Is A Code Kata?

- A code kata is an exercise in programming which helps programmers hone their skills through practice
- The idea is inspired by the Japanese concept of kata in the martial arts
  - Just like in the martial arts, you can repeat a kata multiple times to make improvements to your solutions
- A code kata is usually set up as a series of unit tests, which fail
- Your task is to write code to make them pass

What Is This Kata About?

- Low quality unit tests may be hiding behind passing tests and high coverage numbers
- This may be giving a false sense of security and introducing major risks
- With this kata we will learn
  - How to identify common unit test problems using mutation testing and test smells
  - How to fix these problems
- A small twist on a traditional kata
  - Our tests initially pass but they really shouldn't



Before We Start…

Deep Dive into Unit Testing

Unit Tests – Necessary, But Not Sufficient…

Category	Purpose	Who	Tools
Unit	Validate a unit of behavior at the low (code) level, focusing on a small part of the system (e.g., a method)	Dev	JUnit
Acceptance	Validates that the business logic is implemented as specified for a given scenario	Dev, User	FitNesse
Mutation	Ensure quality of unit and acceptance tests	Dev	PIT
Integration	Detect issues in interactions between modules of the system	Tech Ops, Dev	
User Acceptance	Certify by the users that the system as a whole is operating as expected	Dev, User	
Production Mirror	Test system under load identical to production	Tech Ops	
Chaos Engineering	Test system resiliency by failure injection into infrastructure (service processes, networks, clients, etc.)	Tech Ops	Chaos Monkey
Breakpoint	Determine the maximum amount of load that the system can support	Tech Ops	The Grinder

Unit Tests: Best Practices

- Automated – require no human involvement to determine the outcome
- Focused – each test method tests one scenario
- Complete – test the edge cases, try to cover all **meaningfully different** scenarios
- Well named – test method name describes the scenario being tested
- Fast – the relevant tests execute in a few seconds or faster
- Independent – no external dependencies, no dependencies on other tests

Unit Tests: Smells

- No assertions
- Irrelevant assertions
- Use of Mocks
- Expected results are calculated rather than explicitly specified
- Test code reuse (that is test logic reuse, test utilities are good)
- Test data reuse
- "Flickering" tests (tests with nondeterministic behavior)
- Interdependencies between tests (e.g., execution order)
- Long running tests
- `@Ignore`'d or commented out tests

Sidebar: Test Driven Development

1. Write a test	• Take the user's perspective: "What is the API that would make my job the easiest?" • Work in tiny increments
2. Make it pass	• Do whatever it takes • Duplication? Fine! Hardcoding the expected result? Fine!
3. Refactor	• Remove duplication

Repeat 1-3 for all meaningfully different scenarios

4. Profit

- Almost all code is tested
- You know when to stop
- User friendly interfaces
- Just enough abstraction
- Just enough code
- Develop with confidence

Maintain Discipline: do not cut corners, else end up with code and tests of “usual” quality

Measuring Unit Test Quality

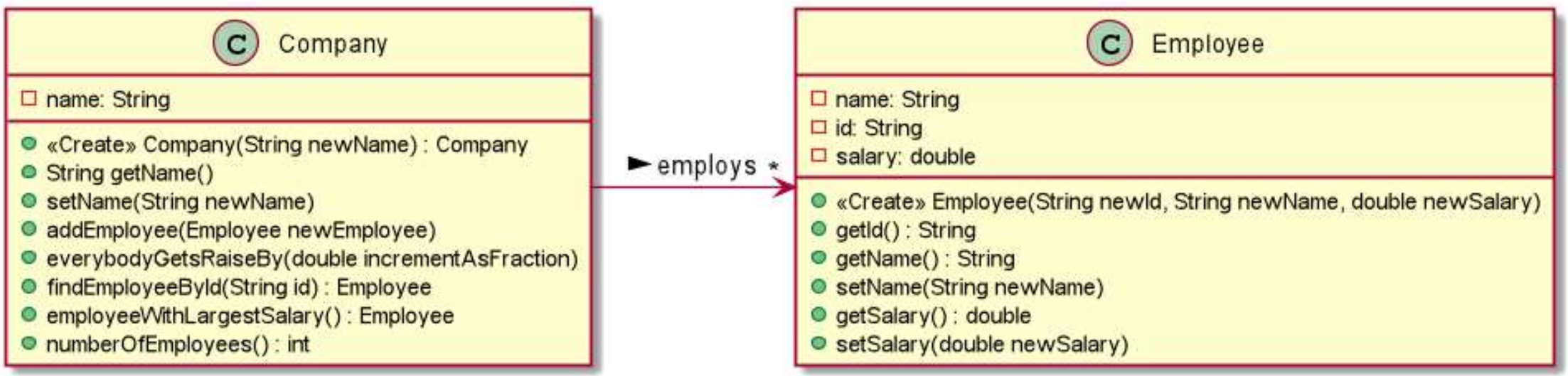
- Test coverage
  - %-ge of LOC, methods, classes covered by tests
  - Does not guarantee the covered code is actually tested
- Does identify the code that is definitely not tested
- So having a coverage target in not entirely pointless, but…
- …don't optimize just for coverage
- How do we make sure that the tests actually test?

Mutation Testing

- A way to validate the quality of unit tests
- Start with all tests passing
- Introduce changes in the code and observe the behavior of the unit tests
- Two possible outcomes:
  - Some of the unit tests will start failing → good
  - All the tests will keep on passing → bad
- The latter scenario means that the unit tests do not really validate outcomes of the code under test
- Caveats: performance and false positives

Enough Talk! Let's Do The Kata!

Kata Domain


Company
name: String
«Create» Company(String newName) : Company
String getName()
setName(String newName)
addEmployee(Employee newEmployee)
everybodyGetsRaiseBy(double incrementAsFraction)
findEmployeeById(String id) : Employee
employeeWithLargestSalary() : Employee
numberOfEmployees() : int
employs *
Employee
name: String
id: String
salary: double
«Create» Employee(String newId, String newName, double newSalary)
getId() : String
getName() : String
setName(String newName)
getSalary() : double
setSalary(double newSalary)

Steps

- Run all the unit tests in the `mtk.domain.CompanyTest` class
- Run and inspect the output of `mtk.CompanyRunner`
  - Anything does not look right?
- Enable the `pitest` profile and run the `test` maven lifecycle phase
  - Uh-oh. What is all this red?
  - Mutants that have survived – not caught by the unit tests

- For each test
  - Identify and fix the test smells → the test will start failing
  - Fix the business logic → the test should pass
- Re-run mutation test coverage
  - Any mutants survived? If so, try to understand what test behavior could catch that mutant.

Reading Mutation Code Coverage Report

- The code in red is the code that had surviving mutations applied to it
- The report will not tell you exactly what is wrong with your tests
- The report will tell you that the outcome of the mutated code is not being tested – or asserted properly
- Use the report as the pointer to unit tests needing fixing – do not analyze it too much
- We will pick one example of a PIT Test failure and trace it back to code to show how it broke
- Mind the false positives
  - Due to aggressive mutations:
    - DEFAULTS should be conservative enough
    - ALL is probably too aggressive
  - Due to the business logic under test
    - Certain combinations of code and mutations may result in code that is still valid

Links

This kata

https://github.com/vmzakharov/mutate-test-kata

PIT Mutation Testing

Home: http://pitest.org

GitHub: https://github.com/hcoles/pitest