# Software Testing Evolution

Roberto A. Bittencourt

Based on Mens & Demeyer's book, Chapter 8

# Some questions about software testing evolution

1. How does a system's test suite influence the program comprehension process of a software engineer trying to understand a given system? What are the possible side effects with regard to evolving the software system?
2. Are there typical code characteristics that indicate which test code resists evolution? And if so, how can we help alleviate these, so called, test smells?

# Some questions about software testing evolution

3. Given that production code evolves through e.g. refactorings — behavior preserving changes —, what is the influence of these refactorings on the associated test code? Does that test code need to be refactored as well or can it remain in place unadapted? And what will happen to its role as safety net against errors?
4. Can we use metrics to understand the relation between test code and production code? In particular, can object-oriented metrics on the production code be used to predict key properties of the test code?

# Program Comprehension and Agile Methods

# Program Understanding

- Program understanding (comprehension) is the **task** of building **mental models** of an underlying software system at various abstraction levels, ranging from models of the code itself to ones of the underlying application domain, for software maintenance, evolution, and re-engineering purposes
- Estimates put the total cost of the understanding phase at **50% of the total effort**
- **An extensive test suite can stimulate the program comprehension process**, especially in the light of continuously evolving software

# Unit Testing and XP

- Unit testing is at the heart of XP
  - Tests are small, take a white box view on the code, and include a check on the correctness of the results obtained, comparing actual results with expected ones
  - Tests are an explicit part of the code, they are put under revision control, and all tests are shared by thedevelopment team
- Testing is typically done using a testing framework such as JUnit
- The XP process encourages writing a test class for every class in the system
  - Particularly, to use tests for documentation purposes

# Comprehension Benefits

- XP's testing policy encourages programmers to explain their code using test cases
- All tests must run 100% at all times, ensuring documentation via unit tests is kept up-to-date
- Adding unit tests provides a repeatable program comprehension strategy
- A comprehensive set of unit tests reduces the comprehension space when modifying source code
- Systematic unit testing helps build team confidence

# Comprehension Risks

- XP increases the code base and this code needs to be maintained as well
- Automated tests may lead to lack of knowledge during initial program comprehension
- If the general trend is not to write documentation, some important decisions may be left undocumented
- Some types of code are inherently hard to test
  - e.g., user interfaces and database code

# Test Smells and Refactorings

# Test Smells and Refactorings

- Test code has a distinct set of smells from production code
  - dealing with the ways in which test cases are organized
  - how they are implemented
  - how they interact with each other
- Improving test code involves a mixture of:
  - applying refactorings specialized to test code improvements
  - a set of additional refactorings
    - modification of test classes
    - the way of grouping test cases

# Test Code Smells (1)

- **Smell 1**: *Mystery Guest*
  - When a test uses external resources, such as a file containing test data, the test is no longer self contained.
- **Smell 2**: *Resource Optimism*
  - Test code that makes optimistic assumptions about the existence (or absence) and state of external resources can cause non-deterministic behavior in test outcomes.
- **Smell 3**: *Test Run War*
  - Such wars arise when the tests run fine as long as you are the only one testing but fail when more programmers run them.

# Test Code Smells (2)

- **Smell 4**: *General Fixture*
  - The setUp fixture is too general and different tests only access part of it.
- **Smell 5**: *Eager Test*
  - When a test method checks several methods of the object to be tested.
- **Smell 6**: *Lazy Test*
  - Several test methods check the same method using the same fixture.
- **Smell 7**: *Assertion Roulette*
  - Having a number of assertions in a single test method that do not have a distinct explanation.

# Test Code Smells (3)

- **Smell 8**: *Indirect Testing*
  - A test class contains methods that perform tests on objects other than the one being tested.
- **Smell 9**: *For Testers Only*
  - When a production class contains methods that are only used by test methods.
- **Smell 10**: *Sensitive Equality*
  - Comparing with toString() instead of comparing values with real equality checks
- **Smell 11**: *Test Code Duplication*
  - Test code may contain undesirable duplication

# Test Refactorings (1)

- **Refactoring 1**: *Inline Resource*
  - To remove the dependency between a test method and some external resource, we incorporate that resource in the test code.
- **Refactoring 2**: *Setup External Resource*
  - If it is necessary for a test to rely on external resources, such as directories, databases, or files, make sure the test that uses them explicitly creates or allocates these resourcesbefore testing, and releases them when done.

# Test Refactorings (2)

- **Refactoring 3**: *Make Resource Unique*
  - Use unique identifiers for all resources that are allocated, e.g. by including a time-stamp or the test name
- **Refactoring 4**: *Reduce Data*
  - Minimize the data that is setup in fixtures to the bare essentials.

# Test Refactorings (3)

- **Refactoring 5**: *Add Assertion Explanation*
  - Assertions in the JUnit framework have an optional first argument to give an explanatory message to the user when the assertion fails. Use this.
- **Refactoring 6**: *Introduce Equality Method*
  - If an object structure needs to be checked for equality in tests, add an implementation for the "equals" method for the object's class.

# How Production Code Refactoring Can Invalidate Its Safety Net

# Types of Refactoring and some examples

- **Composite**: series of smaller refactorings
- **Compatible**: does not change the original interface
  - Extract Class
  - Change Bidirectional Association to Unidirectional
  - Replace Data Value with Object
- **Backwards Compatible**: changes the original interface by extending it
  - Replace Inheritance with Delegation
  - Extract Method
  - Extract Superclass

# Types of Refactoring and some examples

- **Make Backwards Compatible**: changes the original interface, but can adapt the new interface to the old one
  - Remove Parameter
  - Rename Method
  - Move Method
- **Incompatible**: changes the original interface and is not backwards compatible
  - Extract Subclass
  - Inline Method
  - Move Field

# Working Example: Video Store

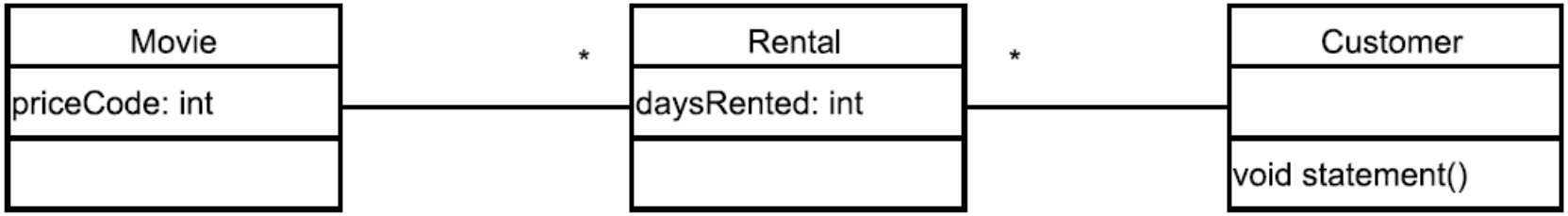



**Fig. 8.1.** Classes before refactoring

# Working Example: Video Store

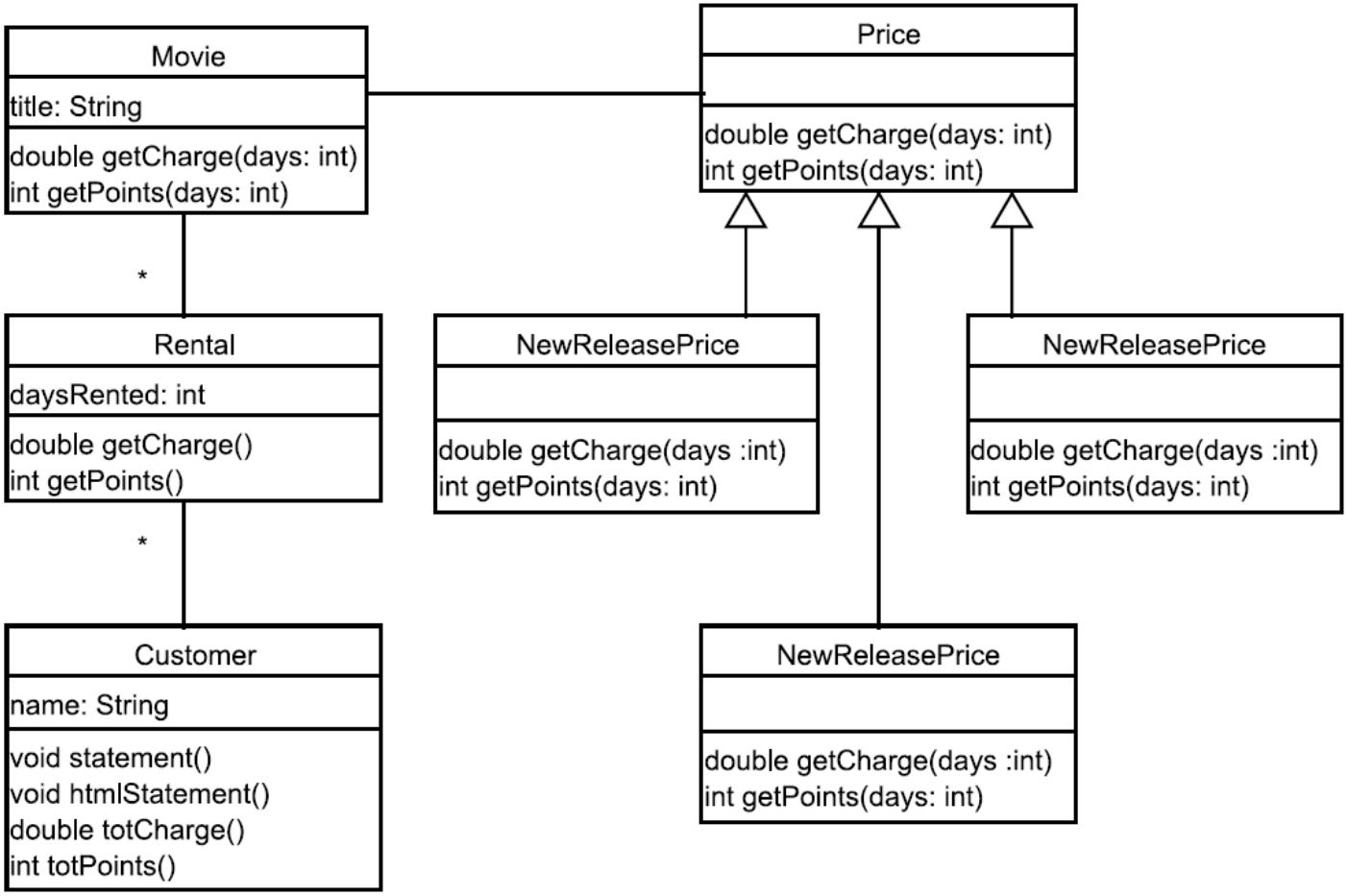



**Fig. 8.2.** Class structure after refactoring

# Working Example: Video Store

```
Movie m1 = new Movie("m1",Movie.CHILDRENS);
Movie m2 = new Movie("m2", Movie.REGULAR);
Movie m3 = new Movie("m3", Movie.NEW_RELEASE);
Rental r1 = new Rental(m1, 5);
Rental r2 = new Rental(m2, 7);
Rental r3 = new Rental(m3, 1);
Customer c1 = new Customer("c1");
Customer c2 = new Customer("c2");

public void setUp() {
  c1.addRental(r1);
  c1.addRental(r2);
  c2.addRental(r3);
}

public void testStatement1() {
  String  expected =
    "Rental Record for c1\n" +
    "\tm1\t4.5\n" +
    "\tm2\t9.5\n" +
    "Amount owed is 14.0\n" +
    "You earned 2 frequent renter points";
  assertEquals(expected, c1.statement());
}
```

# Test-Driven Refactoring

- Use existing test cases to determine code-level refactorings
  - study *test* code find improvements to the *production* code
  - uses both code smells and test smells
- Natural consequence of test-driven design
- Test refactoring may "lose" test cases
  - Measure coverage before and after refactoring
  - Use mutation testing

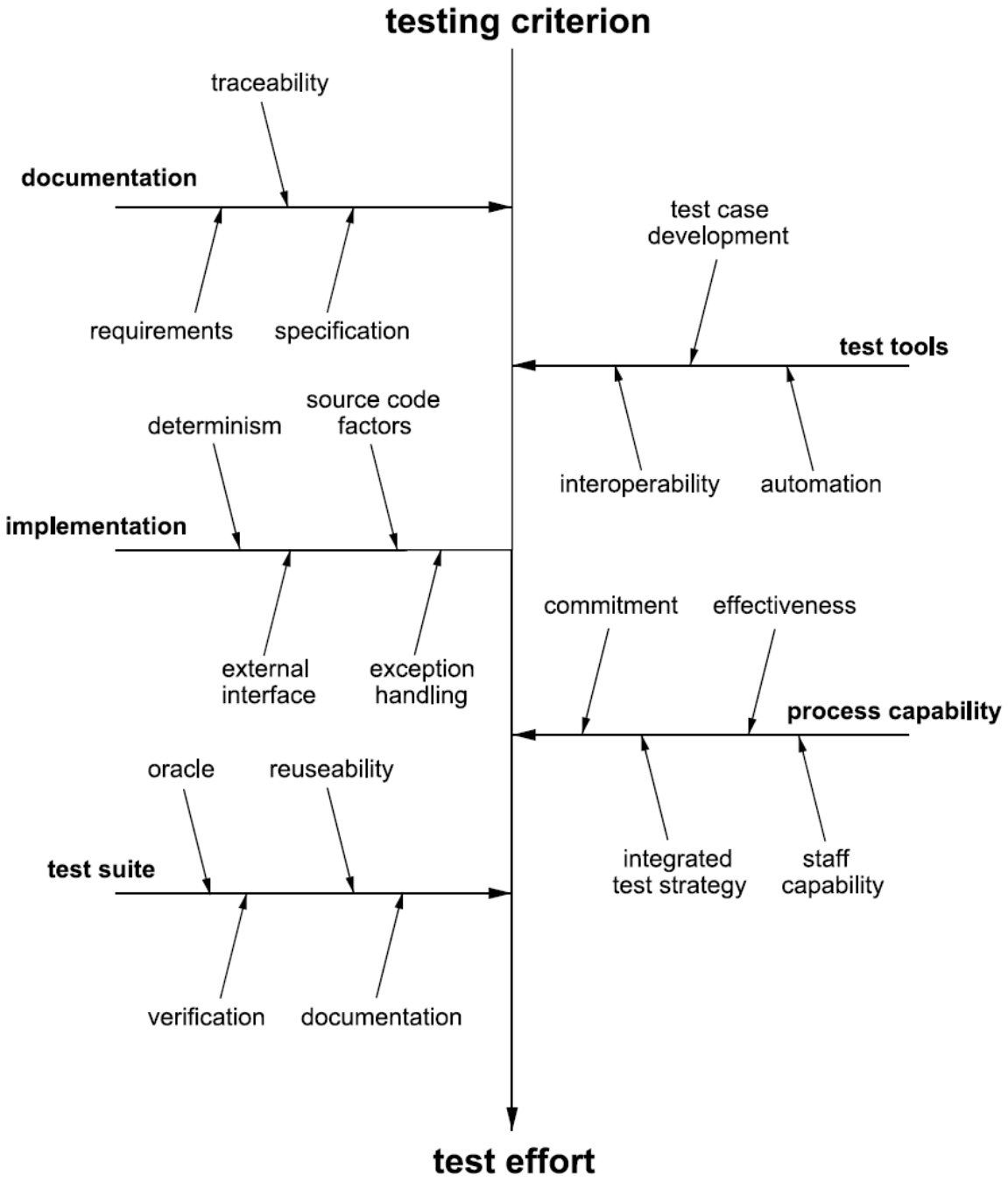
# Refactoring Test Code

1. Detect smells in the code or test code that need to be fixed. In test-driven refactoring, the test set is the starting point for finding such smells.
2. Identify candidate refactorings addressing the smell.
3. Ensure that all existing tests run.
4. Apply the selected refactoring to the code. Provide a backwards compatible interface if the refactoring falls in category D. Only change the associated test classes when the refactoring falls in category E.
5. Ensure that all existing tests run. Consider applying mutation testing to assess the coverage of the test cases.
6. Apply the testing counterpart of the selected refactoring.
7. Ensure that the modified tests still run. Check that the coverage has not changed.
8. Extend the test cases now that the underlying code has become easier to test.
9. Ensure the new tests run.

# Measuring Code and Test Code

# Testability

- "Attributes of software that bear on the effort needed to validate the software product"



# Research Questions

- **Question 1:** Are the values of the object-oriented metrics for a class associated with the required testing effort for that class?

- **Question 2:** Are the values of the object-oriented metrics for a class associated with the size of the corresponding test suite?

# Metrics suite

**Table 8.5.** Metrics suite used for assessing testability of a class *c*

| Metric | Description |
|---|---|
| DIT | Depth of inheritance tree |
| FOUT | Fan out, nr of classes used by *c* |
| LCOM | Lack of cohesion in methods—which measures how fields are used in methods |
| LOCC | Lines of code per class |
| NOC | Number of children |
| NOF | Number of fields |
| NOM | Number of methods |
| RFC | Response for class—Methods in *c* plus the number of methods invoked by *c*. |
| WMC | Weighted methods per class—sum of McCabe's cyclomatic complexity number of all methods. |

# Experimental Results

- Traditional object-oriented source code metrics applied to production code can indicate the effort needed for developing unit tests.

**Table 8.6.** Correlation values and confidence levels found for Ant

| $r_s$ | dLOCC | dNOTC |
|---|---|---|
| DIT | -.0456 | -.201 |
| **FOUT** | .465 | .307 |
| **LCOM** | .437 | .382 |
| **LOCC** | .500 | .325 |
| NOC | .0537 | -.0262 |
| **NOF** | .455 | .294 |
| **NOM** | .532 | .369 |
| **RFC** | .526 | .341 |
| **WMC** | .531 | .348 |

| $p$ | dLOCC | dNOTC |
|---|---|---|
| DIT | .634 | .0344 |
| **FOUT** | $< .01$ | $< .01$ |
| **LCOM** | $< .01$ | $< .01$ |
| **LOCC** | $< .01$ | $< .01$ |
| NOC | .575 | .785 |
| **NOF** | $< .01$ | $< .01$ |
| **NOM** | $< .01$ | $< .01$ |
| **RFC** | $< .01$ | $< .01$ |
| **WMC** | $< .01$ | $< .01$ |

# Conclusion

# Summary

- An extensive test suite can stimulate the program comprehension process, especially in the light of continuously evolving software.
- Test code has a distinct set of smells, dealing with the ways in which test cases are organized, how they are implemented, and how theyinteract with each other.
- Improving test code involves a mixture of applying production code refactorings specialized to test code improvements, as well as a set of additional refactorings, involving the modification of test classes and the way of grouping test cases.

# Summary

- The refactorings as proposed by Fowler can be classified based on the type of change they make to the code, and therefore on the possible change they require in the test code.
- In parallel to test-driven design, *test-driven refactoring* can improve the design of production code by focusing on the desired way of organizing test code to drive refactoring of production code
- Traditional object-oriented source code metrics applied to production code can indicate the effort needed for developing unit tests.