Mutation Testing

Dr. John H Robb, PMP, IEEE SEMC
UTA Computer Science and Engineering

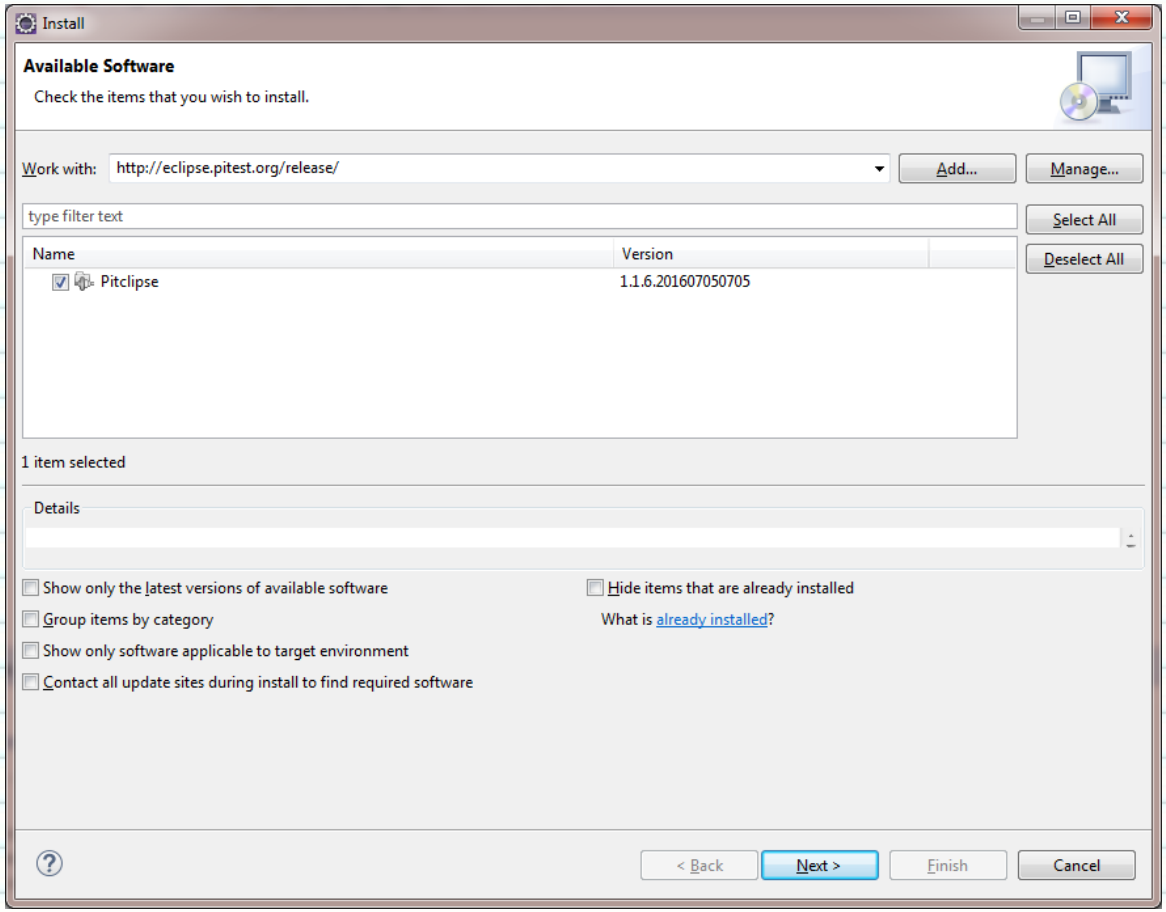
Increasing the Quality of the Test Product

- Why do we want to determine test quality?
  - To assess the quality of the tests - to gauge how good the software product is
  - To use these assessments to help construct more adequate tests
  - To produce a suite of valid tests which can be used on real programs
  - To continually improve the software product

- Methods to determine test adequacy
  - Error Seeding
  - Quality of Test training, tools, and procedures
  - Analysis of test artifacts and personnel
  - Mutation testing

Installing PIT (PITClipse)

- In Eclipse select -> Help -> Install New Software…
- In the “Work with:” text box paste the following:
- http://eclipse.pitest.org/release/



Installing PIT (PITClipse) - cont.

- Check the box next to Pitclipse
- Select Next>
- You may have to restart Eclipse

Error Seeding

- Error seeding works by adding a number of known faults to a software program for the purpose of measuring the rate of detection and removal.
- This allows for estimation of defect detection rates and total number of defects.
- It also allows for an estimation of the number of latent defects in the software.
- Fault seeding: wide variety of faults
  - Erroneous input from the user
  - Wrong data types or values
  - Programming errors
  - Wrong reading of sensors
- This approach assumes that the ability to detect faults is uniform across the software program and that detection of the different kinds of faults is uniform
- This approach is not widely used in practice

Test Coverage Reports

- Test tools provide many different measures of test coverage:
  - Statement
  - Decision
  - Condition
  - Condition/Decision
  - MCDC
  - Some also provide coverage analysis of boundary values

- Achieving high coverage measures can provide confidence in the test suite – does this mean that the tests are good?

- Probably want to combine this with standard test measures such as:
  - Defect density (defects detected per K SLOC)
  - Numbers of escaped defects and severity of defects
  - Software Reliability (up time, etc)

Analysis of Test Artifacts and Personnel

- Product level-assessments
  - Like software development artifacts, test artifacts also undergo technical reviews – this can be used to assure the quality of the test product being developed – checklists can include the same kind of items that mutation tools check for to ensure an “in-activity” capture
  - Many times independent reviewers are brought in to sample select test cases to ensure the quality of each from an independent stand-point
  - Deep dives can also be performed by test management - to look at randomly selected test cases – just to ensure that tests are in fact up to standards

- Process level-assessments
  - Random interviews of testers has also been effective - where testers are asked questions about their understanding and use of processes - the idea that

Quality of Test Training, Tools, and Procedures

- Test tools and processes are excellent ways to increase the capability of test personnel
- Use of tools can help to increase the efficiency of the test force and help increase defect detection - do need to be careful of tool proliferation - this is a call to use the right tools and number of tools
- Processes can help to introduce better methods, to improve existing methods, and train new personnel. They also provide a culture and focus within the test team and organization.
- Process maturity only provides an indirect linkage to product quality
  - Product quality may be quite high and rely more on personnel skills than documented processes
  - Typically, organizations that have process documentation and standards operate at a higher maturity level than those without

Mutation Testing

- Mutation testing is based on two hypotheses
  1. The competent programmer hypothesis - that most defects introduced by experienced programmers are due to small syntactic errors.
  2. The coupling effect. The coupling effect asserts that simple defects can cascade or couple to form other defects.
- The result of applying a mutation operator to the software is called a mutant. If the test suite is able to detect the change (i.e. one or more of its test cases fails), then the mutant is said to be killed.
- A first order mutant is performed by selecting a set of mutation operators and then applying them to the relevant source program one at a time for each component of the source code.
- A second order mutant is performed by applying two mutations at once:

  P: i=j+k; P': j=i+k, where the variable replacement order has been applied twice

Mutation Testing (cont.)

- Mutant equivalence
  - The surviving mutants that cannot be killed, are called Equivalent Mutants.
  - Although syntactically different, these mutants are indistinguishable through test.
  - Therefore, they have to be checked ‘by hand’ for applicability - if valid the tests are updated (or created) to fail the mutant - thereby increasing the quality of the test suite.
  - As a last resort the ratio of killed mutants to total mutants is considered, higher ratios increase confidence in the test suite.

Selecting Mutation Operators

- Typically, only first order mutants are generated:
  1. to lower the cost of testing
  2. most higher order mutants are killed by tests passable with respect to first order mutants

- A set of mutant operators is designed for each programming language

- The design of mutation operators is based on guidelines and experience. Different groups might arrive at a different set of mutation operators for the same programming language.

- We judge whether or not that a set of mutation operators is adequate by choosing the set that will generate the most errors over a set of erroneous programs

Categories of Mutant Operators

- Amman and Offutt
  - Eleven categories of mutation operators
    - arithmetic operators: A = {+, -, *, /, %}
    - relational operators: R = {<, <=, ==, ≠, >, >=}
    - logical connectives: L = {&,|,^,!}
    - Others? Application dependent, e.g.
      - String operators
      - Trigonometric functions
      - Statistics functions
- Mathur
  - Variable replacement
  - Relational operator replacement
  - Off-by-1
  - Replacement by 0
  - Arithmetic operator replacement

Why We Need Mutation Testing

- Getting quality technical reviews of actual test data is very difficult because only the author is familiar enough with the function under test to develop (or evaluate) good test cases.

- After looking at data of 10,000s of technical reviews - we tend to find disproportionally fewer defects per line of text in reviews of test artifacts
  - Many of the comments are about style and in-line comments - they do NOT improve the ability of the test or tester to find defects

- Mutation testing has the ability to provide an improvement to the individual test cases and to the tester.

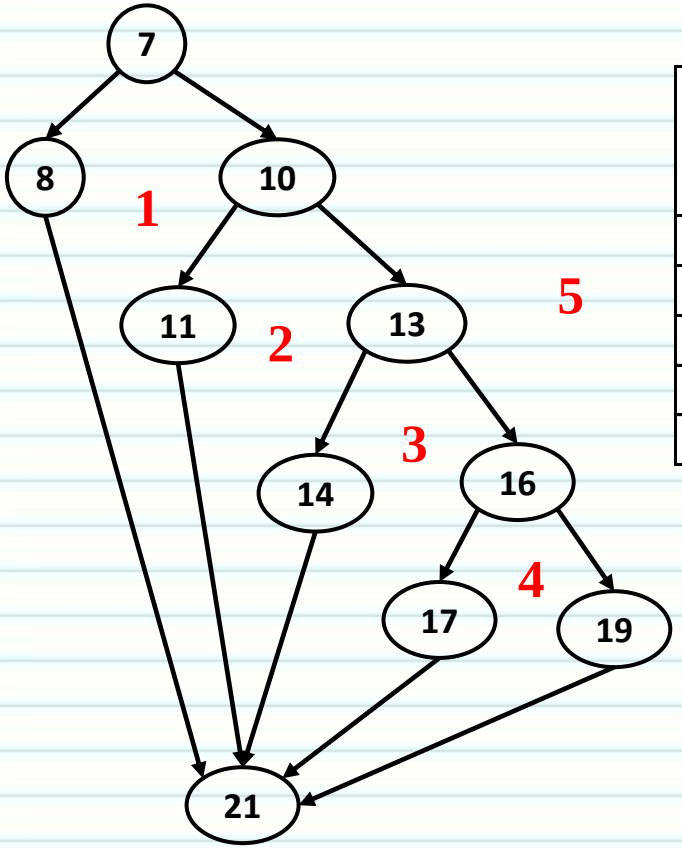
A Simple Example

```
1 public class Problem3Class {
2
3   public String getTemp(int temp) {
4       String feels;
5
6       if (temp<=32)
7           feels="Freezing";
8       else {
9           if (temp<40)
10              feels="Very Cold";
11          else {
12              if (temp<50)
13                  feels="Cold";
14              else {
15                  if (temp<80)
16                      feels="Nice";
17                  else
18                      feels="Warm";
19              }}}
20      return feels;
21  }
22 }
```

Draw the reduced CFG, develop the basis paths and identify test cases.

Do we need to add more test cases?

A Simple Example (cont.)

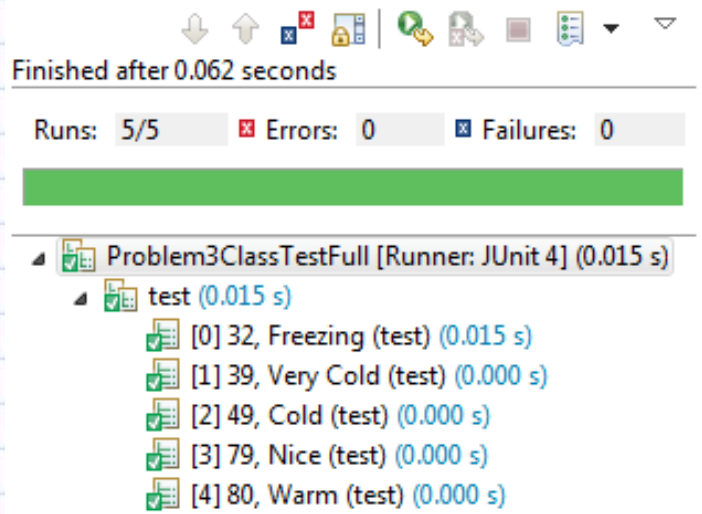


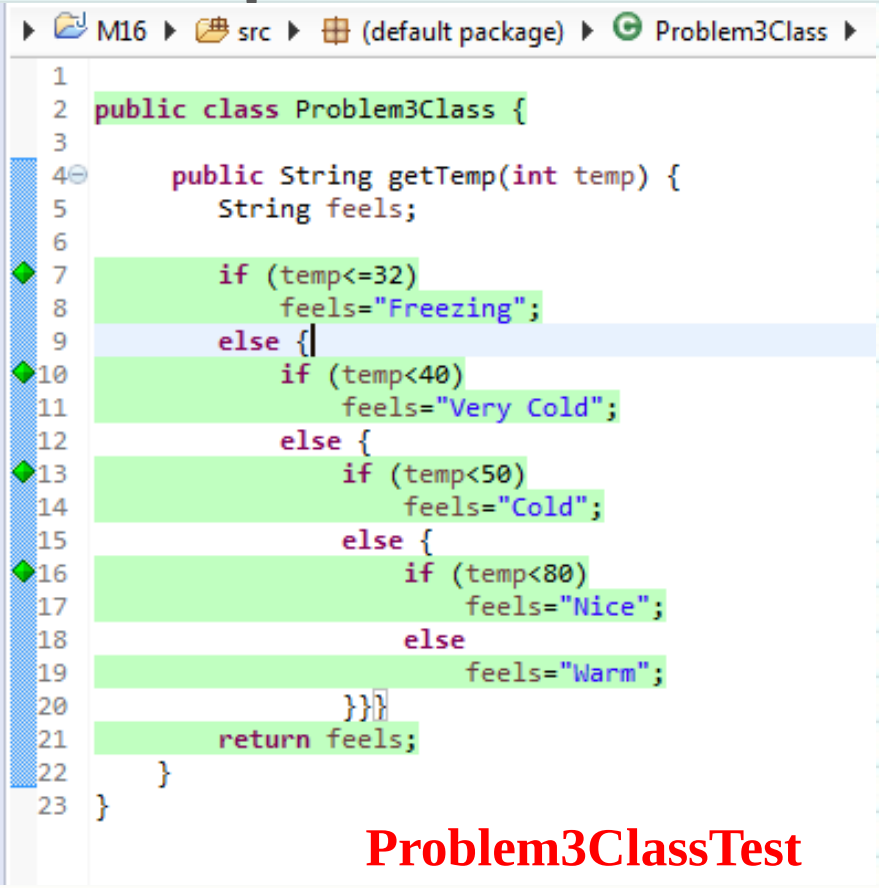
Test Case Number	Inputs temp	Expected result (String)	Basis Path Tested
1	32	"Freezing"	7,8,21
2	39	"Very Cold"	7,10,11,21
3	49	"Cold"	7,10,13,14,21
4	79	"Nice"	7,10,13,16,17,31
5	80	"Warm"	7,10,13,16,19,21

Draw the reduced CFG, develop the basis paths and identify test cases.

Do we need to add more test cases?

A Simple Example (cont.)





Problem3ClassTest

1. **With these 5 test cases we achieved full decision and statement coverage. Why?**
2. **JaCoCo happily reports us as testing everything - but we know better - what is missing?**

A Simple Example (cont.)

Problem3Class.java

Mutations

	1. changed conditional boundary → KILLED
	2. Substituted 32 with 33 → SURVIVED
7	3. negated conditional → KILLED
	4. removed conditional - replaced comparison check with false → KILLED
	5. removed conditional - replaced comparison check with true → KILLED
	1. changed conditional boundary → SURVIVED
	2. Substituted 40 with 41 → SURVIVED
10	3. negated conditional → KILLED
	4. removed conditional - replaced comparison check with false → KILLED
	5. removed conditional - replaced comparison check with true → KILLED
	1. changed conditional boundary → SURVIVED
	2. Substituted 50 with 51 → SURVIVED
13	3. negated conditional → KILLED
	4. removed conditional - replaced comparison check with false → KILLED
	5. removed conditional - replaced comparison check with true → KILLED
	1. changed conditional boundary → KILLED
	2. Substituted 80 with 81 → KILLED
16	3. negated conditional → KILLED
	4. removed conditional - replaced comparison check with false → KILLED
	5. removed conditional - replaced comparison check with true → KILLED
21	1. mutated return of Object value for Problem3Class::getTemp to (if (x != null) null else throw new RuntimeException) → KILLED

What does PIT say about our testing?

What happens if we run all 8 tests?

A Simple Example (cont.)

Problem3Class.java

Mutations

Problem3ClassTestFull

```
1. changed conditional boundary → KILLED
2. Substituted 32 with 33 → KILLED
7 3. negated conditional → KILLED
4. removed conditional - replaced comparison check with false → KILLED
5. removed conditional - replaced comparison check with true → KILLED

1. changed conditional boundary → KILLED
2. Substituted 40 with 41 → KILLED
10 3. negated conditional → KILLED
4. removed conditional - replaced comparison check with false → KILLED
5. removed conditional - replaced comparison check with true → KILLED

1. changed conditional boundary → KILLED
2. Substituted 50 with 51 → KILLED
13 3. negated conditional → KILLED
4. removed conditional - replaced comparison check with false → KILLED
5. removed conditional - replaced comparison check with true → KILLED

1. changed conditional boundary → KILLED
2. Substituted 80 with 81 → KILLED
16 3. negated conditional → KILLED
4. removed conditional - replaced comparison check with false → KILLED
5. removed conditional - replaced comparison check with true → KILLED

21 1. mutated return of Object value for Problem3Class::getTemp to ( if (x != null) null else throw new RuntimeException ) → KILLED
```

- **PIT reports that we have a good set of tests - there are no source code modifications that can be made without our tests detecting them - we are checking everything we can.**
- **We are getting very good feedback on the technical quality of our tests!**
- **Are we providing full coverage?**

Checking Just ECPs

- Remember that we had some authors that said that beginners check BVs and more experienced testers check in the middle of ECPs.
- Let's see what PIT has to say about that. Here are the ECPs

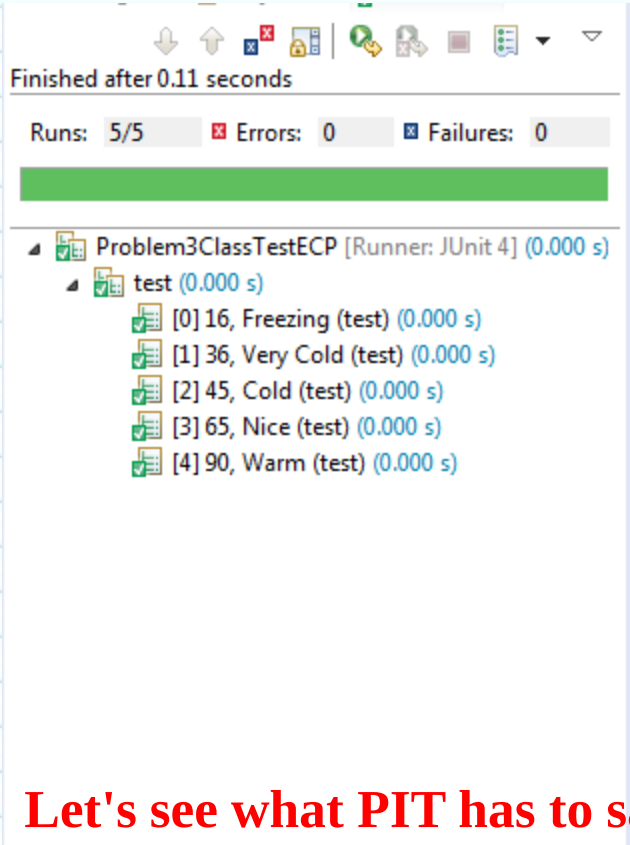
- We'll use these test cases.

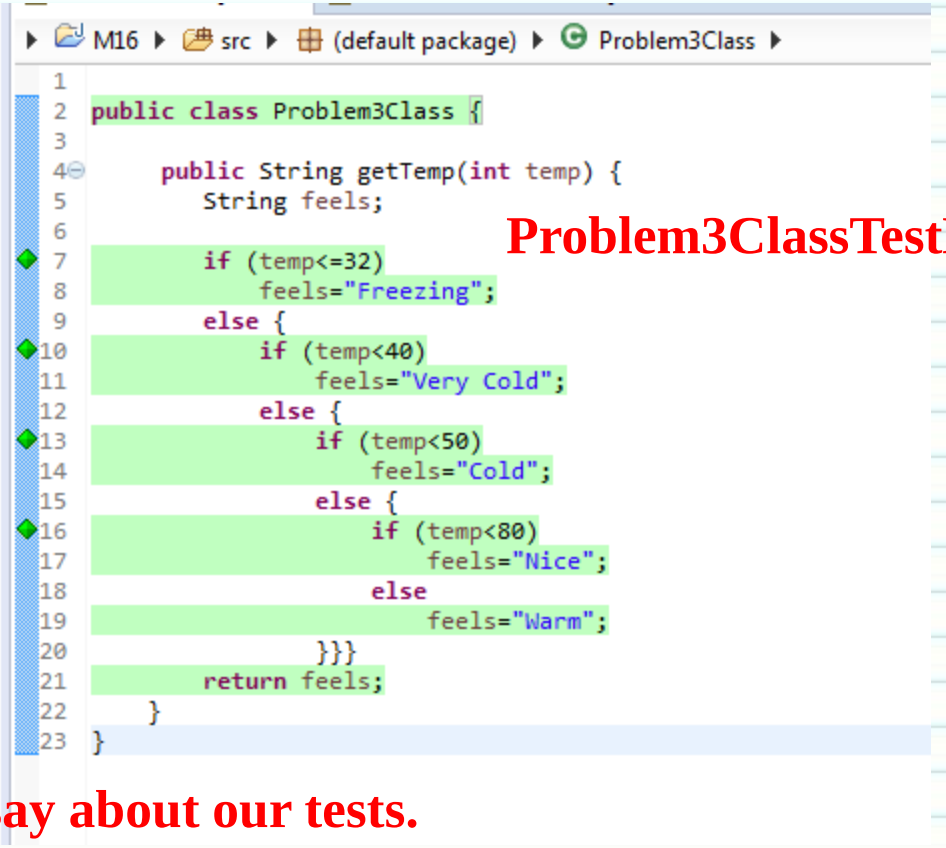
Test Case Number	Inputs	Expected Outputs	Basis Path Tested
	temp	result (String)	
1	16	"Freezing"	7,8,21
2	36	"Very Cold"	7,10,11,21
3	45	"Cold"	7,10,13,14,21
4	65	"Nice"	7,10,13,16,17,31
5	90	"Warm"	7,10,13,16,19,21

Checking Just ECPs (cont.)

- Do we achieve Basis Path coverage from these tests? Why?

- Do we achieve JaCoCo coverage from these tests? Why?





Problem3ClassTestECP

Let's see what PIT has to say about our tests.

Checking Just ECPs (cont.)

Problem3Class.java

Problem3ClassTestECP

Mutations

	1. changed conditional boundary → SURVIVED
	2. Substituted 32 with 33 → SURVIVED
7	3. negated conditional → KILLED
	4. removed conditional - replaced comparison check with false → KILLED
	5. removed conditional - replaced comparison check with true → KILLED
	1. changed conditional boundary → SURVIVED
	2. Substituted 40 with 41 → SURVIVED
10	3. negated conditional → KILLED
	4. removed conditional - replaced comparison check with false → KILLED
	5. removed conditional - replaced comparison check with true → KILLED
	1. changed conditional boundary → SURVIVED
	2. Substituted 50 with 51 → SURVIVED
13	3. negated conditional → KILLED
	4. removed conditional - replaced comparison check with false → KILLED
	5. removed conditional - replaced comparison check with true → KILLED
	1. changed conditional boundary → SURVIVED
	2. Substituted 80 with 81 → SURVIVED
16	3. negated conditional → KILLED
	4. removed conditional - replaced comparison check with false → KILLED
	5. removed conditional - replaced comparison check with true → KILLED
21	1. mutated return of Object value for Problem3Class::getTemp to (if (x != null) null else throw new RuntimeException) → KILLED

PIT says that we are not checking several BVs!

PIT verifies that this kind of testing is poor practice. Why - what is the most probable kind of defect in this code?

PIT Testing

- We will look at a few more examples of PIT on problems that we have done to assess our testing approach.

- From these we can see that PIT is extremely helpful in finding problem test cases and in improving the tester.

Back to the Gregorian Calendar

- This example is from the book "How We Test Software at Microsoft"

- From the Gregorian calendar example, the dates of 10/5/-10/14/1582 were excluded from the calendar
- The book writes this as:

```
if (year == 1582 && month ==10 && !(day<5) && !(day > 14))
    return true;
else
    return false;
```

Is there a better way to write the last two conditions?

```
if (year == 1582 && month ==10 && day>=5 && day <= 14)
```

Back to the Gregorian Calendar (cont.)

- This is the solution from the book - what is wrong with this solution?

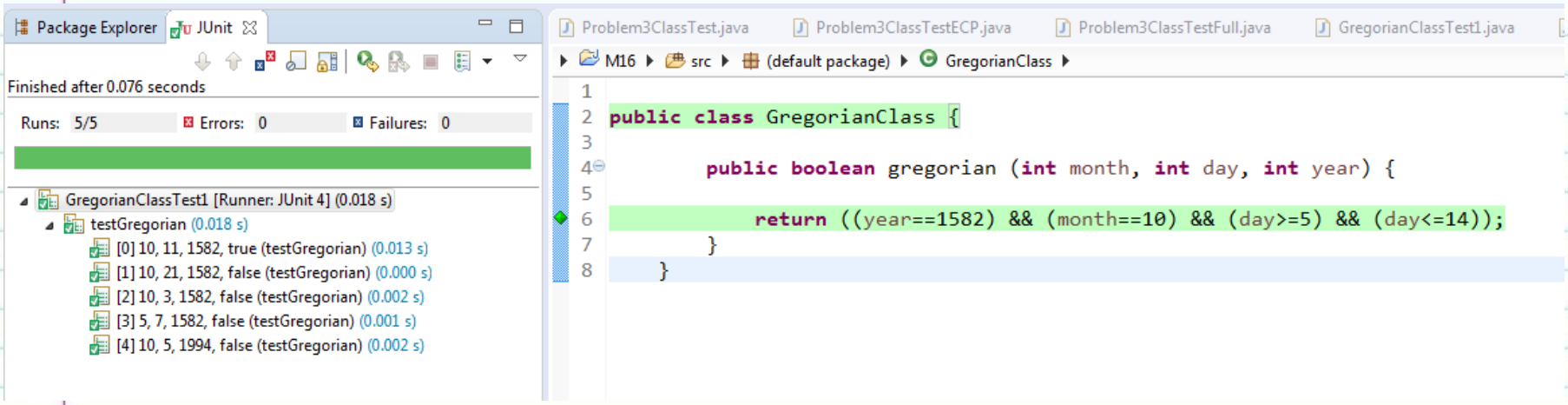
Table 6-1. Truth Table for IsInvalidGregorianCalendarDate Function

Tests	Parameters			Conditional clauses				Expected result
	Month	Day	Year	Year	Month	!(day < 5)	!(day > 14)	
1	10	11	1582	True	True	True	True	True
2	10	21	1582	True	True	True	False	False
3	10	3	1582	True	True	False		False
4	5	7	1582	True	False			False
5	10	5	1994	False				False

- This turns out to be a really good exercise. If we use MC/DC how many test cases is this - how many conditions are in the expression?

```
if (year == 1582 && month ==10 && day>=5 && day <= 14)
```

Back to the Gregorian Calendar (cont.)



All 5 tests get full coverage from JaCoCo - remember there is nothing wrong with JaCoCo - it is the leading Java code coverage on the market

This is where we have to understand what is being assessed and what is <u>not</u>

Back to the Gregorian Calendar (cont.)

- PIT finds fours faults with these tests

GregorianClass.java

GregorianClassTest1

Mutations

6

1. changed conditional boundary → SURVIVED
2. changed conditional boundary → SURVIVED
3. Substituted 1582 with 1583 → KILLED
4. Substituted 10 with 11 → KILLED
5. Substituted 5 with 6 → SURVIVED
6. Substituted 14 with 15 → SURVIVED
7. Substituted 1 with 0 → KILLED
8. Substituted 0 with 1 → KILLED
9. negated conditional → KILLED
10. negated conditional → KILLED
11. negated conditional → KILLED
12. negated conditional → KILLED
13. removed conditional - replaced equality check with false → KILLED
14. removed conditional - replaced equality check with false → KILLED
15. removed conditional - replaced equality check with true → KILLED
16. removed conditional - replaced equality check with true → KILLED
17. removed conditional - replaced comparison check with false → KILLED
18. removed conditional - replaced comparison check with false → KILLED
19. removed conditional - replaced comparison check with true → KILLED
20. removed conditional - replaced comparison check with true → KILLED
21. replaced return of integer sized value with (x == 0 ? 1 : 0) → KILLED
22. replaced return of integer sized value with (x == 0 ? 1 : 0) → KILLED

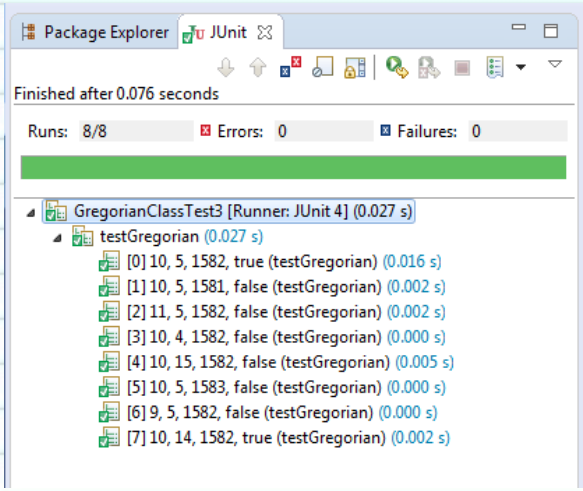
Active mutators

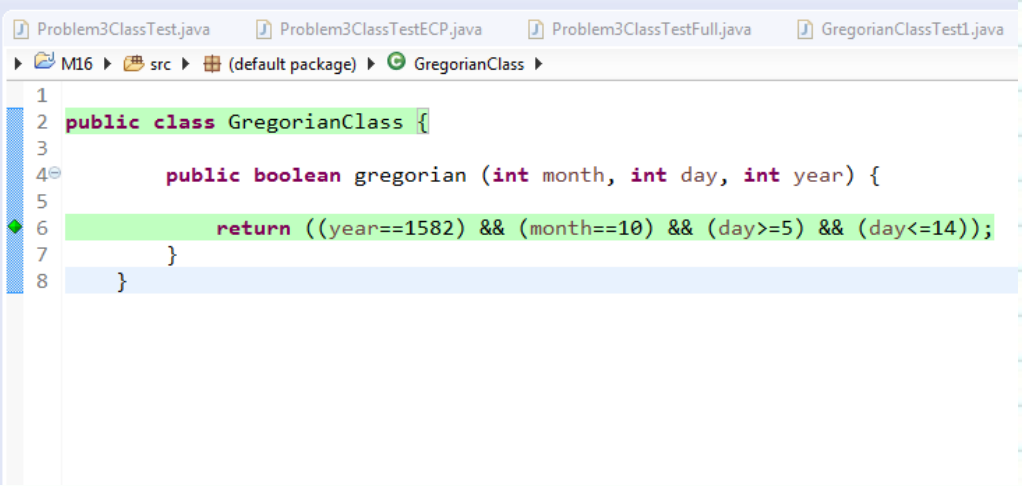
Back to the Gregorian Calendar (cont.)

- Let's run the tests we developed in class

	Inputs			Output
Test Case	Year	Month	Day	Return
1	1582	10	5	TRUE
2	1581	10	5	FALSE
3	1582	11	5	FALSE
4	1582	10	4	FALSE
5	1582	10	15	FALSE
6	1583	10	5	FALSE
7	1582	9	5	FALSE
8	1582	10	14	TRUE

GregorianClassTest3





Back to the Gregorian Calendar (cont.)

- Let's run the tests we developed in class

	Inputs			Output
Test Case	Year	Month	Day	Return
1	1582	10	5	TRUE
2	1581	10	5	FALSE
3	1582	11	5	FALSE
4	1582	10	4	FALSE
5	1582	10	15	FALSE
6	1583	10	5	FALSE
7	1582	9	5	FALSE
8	1582	10	14	TRUE

- Again PIT likes our tests :)

GregorianClassTest3

GregorianClass.java

Mutations

```
1. changed conditional boundary → KILLED
2. changed conditional boundary → KILLED
3. Substituted 1582 with 1583 → KILLED
4. Substituted 10 with 11 → KILLED
5. Substituted 5 with 6 → KILLED
6. Substituted 14 with 15 → KILLED
7. Substituted 1 with 0 → KILLED
8. Substituted 0 with 1 → KILLED
9. negated conditional → KILLED
10. negated conditional → KILLED
11. negated conditional → KILLED
12. negated conditional → KILLED
13. removed conditional - replaced equality check with false → KILLED
14. removed conditional - replaced equality check with false → KILLED
15. removed conditional - replaced equality check with true → KILLED
16. removed conditional - replaced equality check with true → KILLED
17. removed conditional - replaced comparison check with false → KILLED
18. removed conditional - replaced comparison check with false → KILLED
19. removed conditional - replaced comparison check with true → KILLED
20. removed conditional - replaced comparison check with true → KILLED
21. replaced return of integer sized value with (x == 0 ? 1 : 0) → KILLED
22. replaced return of integer sized value with (x == 0 ? 1 : 0) → KILLED
```

6

Active mutators

Logical Operators

- Let's go back to the expression ab + cd that we used to demonstrate the extra coverage that Minimal-MUMCUT provides
- Recall for this expression there were 40 single-faults identified

Fault Class	Implementation (mutant)	Fault Class	Implementation (mutant)
ENF	$\overline{ab + cd}$	LRF	$cb + cd, \bar{c}b + cd, db + cd, \bar{d}b + cd,$ $ac + cd, a\bar{c} + cd, ad + cd, a\bar{d} + cd,$ $ab + ad, ab + \bar{a}d, ab + bd, ab + \bar{b}d,$ $ab + ca, ab + c\bar{a}, ab + cb, ab + c\bar{b}$
TNF	$\overline{ab} + cd, ab + \overline{cd}$		
LNF	$\overline{a}b + cd, a\overline{b} + cd, ab + \overline{c}d, ab + c\overline{d}$		
ORF[+]	$abcd$		
ORF[·]	$a + b + cd, ab + c + d$	LIF	$abc + cd, ab\bar{c} + cd, abd + cd,$ $ab\bar{d} + cd, ab + acd, ab + \bar{a}cd,$ $ab + bcd, ab + \bar{b}cd$
TOF	cd, ab		
LOF	$b + cd, a + cd, ab + d, ab + c$		

  - For this expression MC/DC provided 95 percent coverage with the following test cases: TTFT, TTTF, FTTT, TFTT, FTFT
  - Minimal MUMCUT gives - MCDC + TFTF (one additional term)
    - Let's use PIT to identify any possible issues

PIT Evaluation of Logical Expressions

- Lets evaluate a + b + c without short-circuiting - recall that we could get a single test case to pass with all coverage in JaCoCo (use FFF)

logicalExpressionClass2Test.java

```
2
3  public class logicalExpressionClass2 {
4
5      public boolean returnInput(boolean conditiona, boolean conditionb, boolean conditionc) {
6
7              return (conditiona | conditionb | conditionc);
8      }
9  }
10
```

logicalExpressionClass2.java

```
1  package Code;
2
3  public class logicalExpressionClass2 {
4
5          public boolean returnInput(boolean conditiona, boolean conditionb, boolean conditionc) {
6
7 3                  return (conditiona | conditionb | conditionc);
8          }
9  }
```

Mutations

```
1. Replaced bitwise OR with AND → SURVIVED
7 2. Replaced bitwise OR with AND → SURVIVED
3. replaced return of integer sized value with (x == 0 ? 1 : 0) → KILLED
```

PIT is not fooled

Active mutators

PIT Evaluation of Logical Expressions

- Lets evaluate a + b + c without short-circuiting - let's use c/d coverage of this expression (JaCoCo is still green)

logicalExpressionClass2Test1.java

```
2
3  public class logicalExpressionClass2 {
4
5      public boolean returnInput(boolean conditiona, boolean conditionb, boolean conditionc) {
6
7              return (conditiona | conditionb | conditionc);
8      }
9  }
10
```

logicalExpressionClass2.java

```
1  package Code;
2
3  public class logicalExpressionClass2 {
4
5          public boolean returnInput(boolean conditiona, boolean conditionb, boolean conditionc) {
6
7 3                  return (conditiona | conditionb | conditionc);
8          }
9  }
```

Mutations

```
1. Replaced bitwise OR with AND → SURVIVED
7 2. Replaced bitwise OR with AND → SURVIVED
3. replaced return of integer sized value with (x == 0 ? 1 : 0) → KILLED
```

PIT is not fooled

Active mutators

PIT Evaluation of Logical Expressions

- Lets evaluate a + b + c with short-circuiting - we'll use MCDC tests for this

```
1
2
3  public class logicalExpressionClass2 {
4
5      public boolean returnInput(boolean conditiona, boolean conditionb, boolean conditionc) {
6
7          return (conditiona || conditionb || conditionc);
8      }
9  }
10
```

logicalExpressionClass2Test2.java

logicalExpressionClass2.java

Mutations

PIT likes MCDC

1. Substituted 0 with 1 → KILLED
2. Substituted 1 with 0 → KILLED
3. negated conditional → KILLED
4. negated conditional → KILLED
5. negated conditional → KILLED
6. removed conditional - replaced equality check with false → KILLED
7. removed conditional - replaced equality check with false → KILLED
8. removed conditional - replaced equality check with false → KILLED
9. removed conditional - replaced equality check with true → KILLED
10. removed conditional - replaced equality check with true → KILLED
11. removed conditional - replaced equality check with true → KILLED
12. replaced return of integer sized value with (x == 0 ? 1 : 0) → KILLED
13. replaced return of integer sized value with (x == 0 ? 1 : 0) → KILLED

Active mutators

Looping Expressions and PIT

- In the first lecture we examined the expression (this is with the bug fixed):

```
public class NumZeroClass {

    public int numZero (int [ ] arr)
    {  // Effects: Return the number of occurrences of 0 in arr
       int count = 0;
       for (int i = 0; i < arr.length; i++)
          if (arr [ i ] == 0)
             count++;

       return count;
    }
}
```

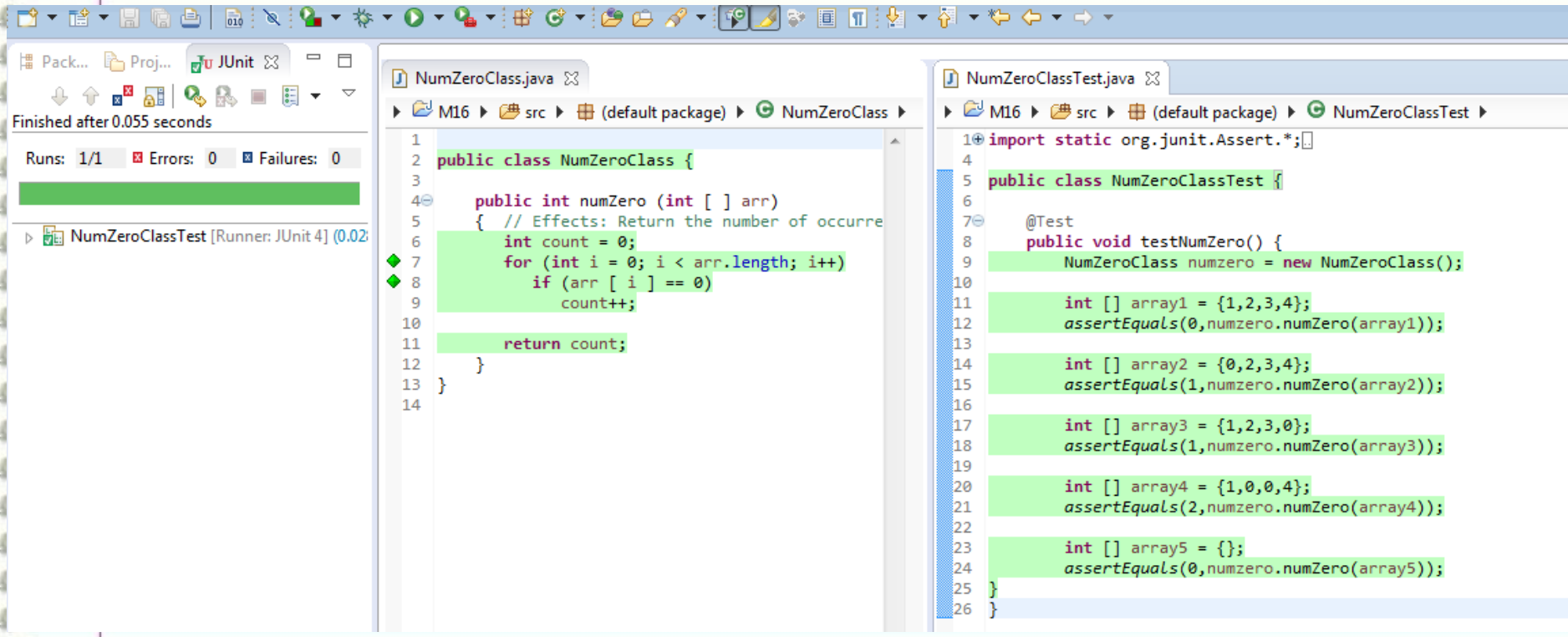
Looping Expressions and PIT (cont.)

- We examined four semantic test cases (and add another for null expressions):

```
@Test
public void testNumZero() {
        NumZeroClass numzero = new NumZeroClass();
        int [] array1 = {1,2,3,4};
        assertEquals(0,numzero.numZero(array1));
        int [] array2 = {0,2,3,4};
        assertEquals(1,numzero.numZero(array2));
        int [] array3 = {1,2,3,0};
        assertEquals(1,numzero.numZero(array3));
        int [] array4 = {1,0,0,4};
        assertEquals(2,numzero.numZero(array4));
        int [] array5 = {};
        assertEquals(0,numzero.numZero(array5));}}
```

Looping Expressions and PIT (cont.)

- JUnit passes and we get complete JaCoCo coverage



Looping Expressions and PIT (cont.)

- PIT provides the following analysis of these tests

NumZeroClass.java

Mutations

7	1. changed conditional boundary → KILLED 2. Changed increment from 1 to -1 → KILLED 3. negated conditional → KILLED
8	1. negated conditional → KILLED 2. removed conditional → KILLED
9	1. Changed increment from 1 to -1 → KILLED
11	1. replaced return of integer sized value with (x == 0 ? 1 : 0) → KILLED

Active mutators

- REMOVE_CONDITIONALS_EQUAL_ELSE_MUTATOR
- EXPERIMENTAL_SWITCH_MUTATOR
- INCREMENTS_MUTATOR
- VOID_METHOD_CALL_MUTATOR
- RETURN_VALS_MUTATOR
- MATH_MUTATOR
- NEGATE_CONDITIONALS_MUTATOR
- INVERT_NEGS_MUTATOR
- CONDITIONALS_BOUNDARY_MUTATOR

- PIT can give us some assessment of the quality our test cases

PIT and Switch Statements

```
1  public class Day {
2        public enum dayName {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};
3        public enum dayType {Weekday,TGIF,Weekend};
4     }
```

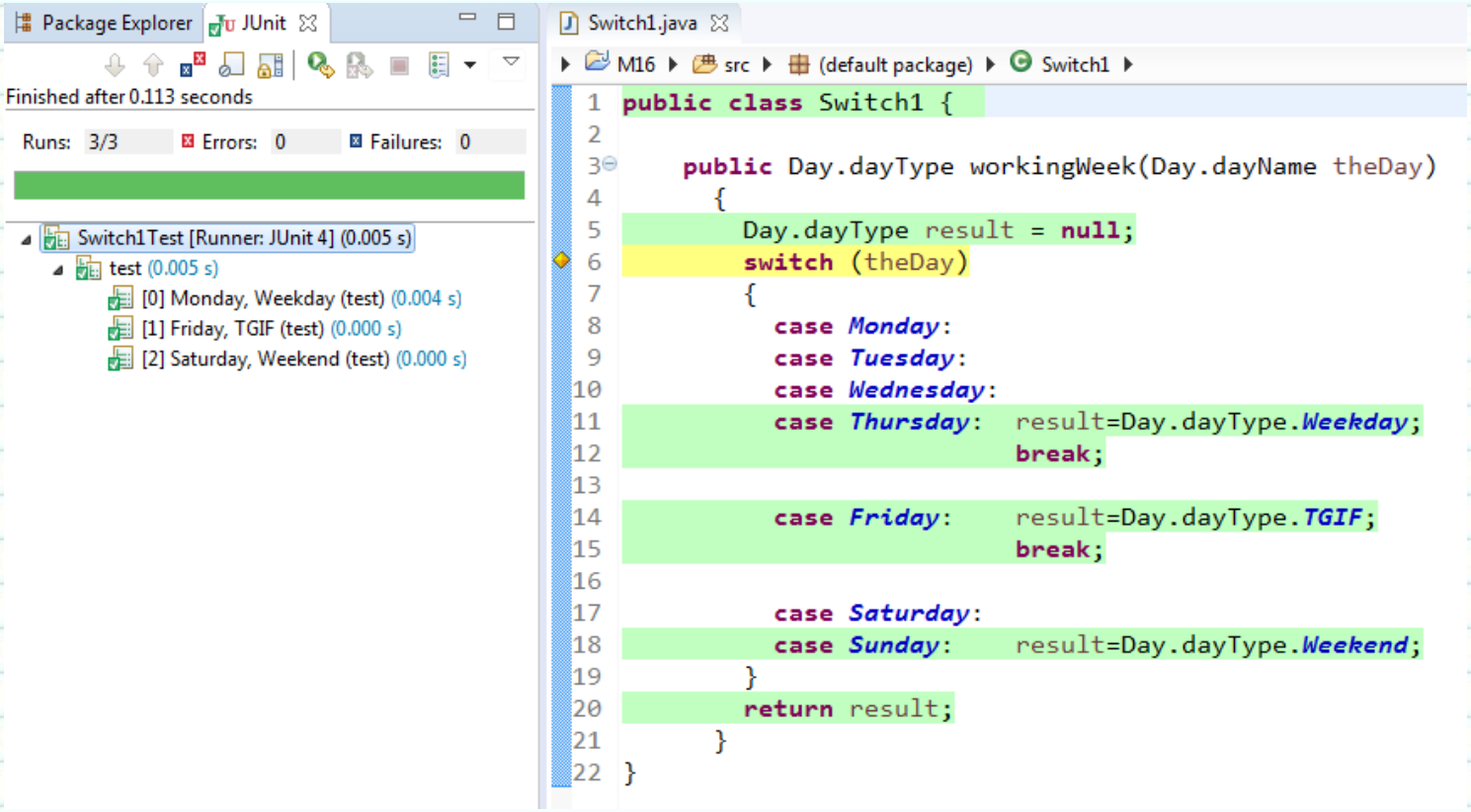
```
public class Switch1 {

    public Day.dayType workingWeek(Day.dayName theDay)
      {
        Day.dayType result = null;
        switch (theDay)
        {
          case Monday:
          case Tuesday:
          case Wednesday:
          case Thursday:   result=Day.dayType.Weekday;
                           break;

          case Friday:     result=Day.dayType.TGIF;
                           break;

          case Saturday:
          case Sunday:     result=Day.dayType.Weekend;
        }
        return result;
      }
}
```

```
19     @SuppressWarnings("unused")
20     private static final Object[] parametersForSwitch1Test () {
21         return $(
22 //                  Parameters are: (1,2)
23 //                          1=day Name, 2=expected Day type
24 //              Test case 1
25                 $(Day.dayName.Monday, Day.dayType.Weekday),
26 //              Test case 2
27                 $(Day.dayName.Friday, Day.dayType.TGIF),
28 //              Test case 3
29                 $(Day.dayName.Saturday, Day.dayType.Weekend)
30         );
31     }
32
33     @Test
34     @Parameters(method="parametersForSwitch1Test")
35     public void test(Day.dayName dName, Day.dayType expDType) {
36         assertEquals(expDType,sw1.workingWeek(dName));
37     }
38 }
```

We're only testing each unique case in the Switch statement of an enumeration value

PIT and Switch Statements (cont.)

- As expected we don't get full coverage because of the JBC default check



PIT and Switch Statements (cont.)

- Let's run PIT on the test with the three tests
  (code is without the default in the Switch)

Switch1.java

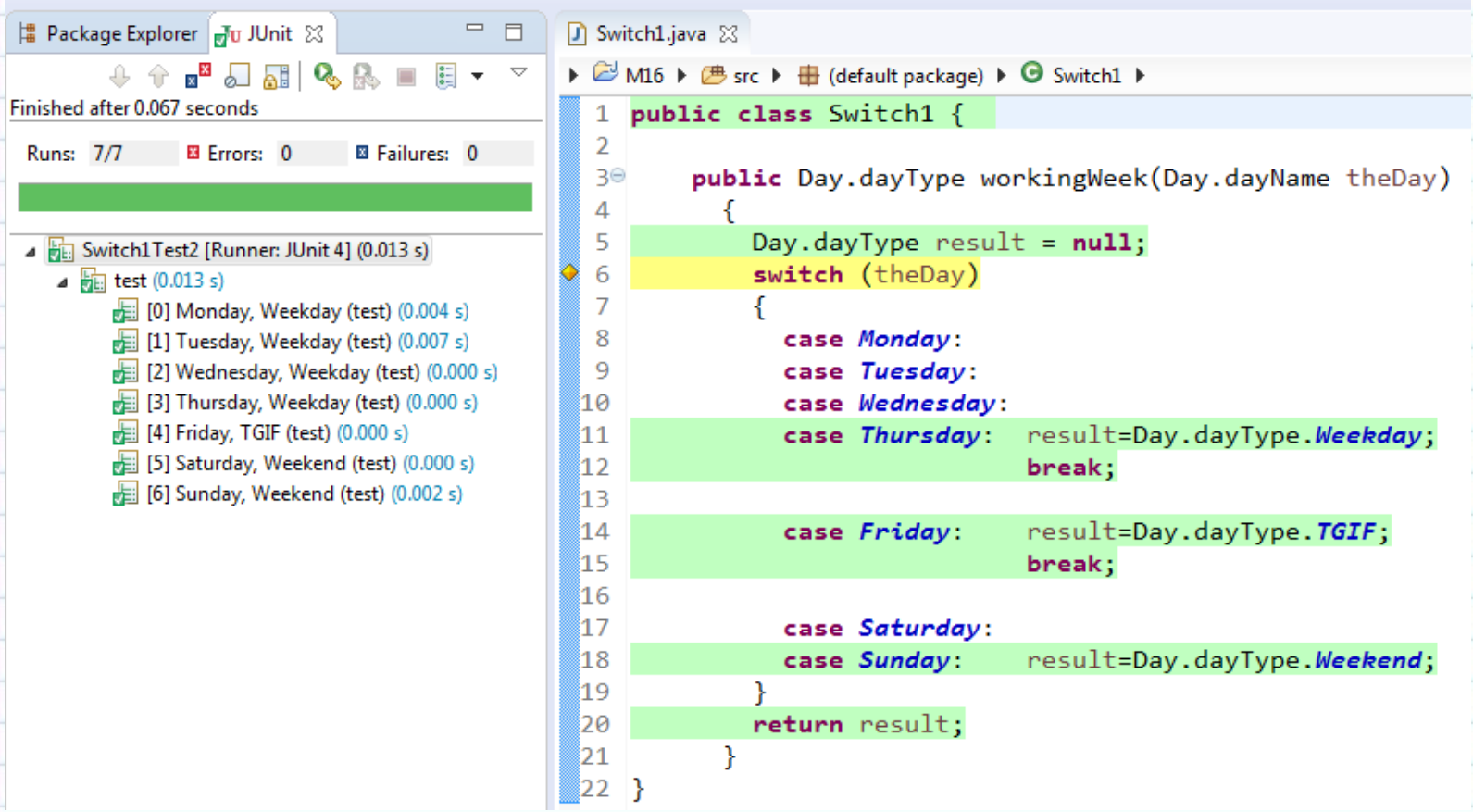
Mutations

```
1. removed call to Switch1::$SWITCH_TABLE$Day$dayName → KILLED
2. removed call to Day$dayName::ordinal → KILLED
3. RemoveSwitch 0 mutation → KILLED
4. RemoveSwitch 1 mutation → SURVIVED
5. RemoveSwitch 2 mutation → SURVIVED
6. RemoveSwitch 3 mutation → SURVIVED
7. RemoveSwitch 4 mutation → KILLED
8. RemoveSwitch 5 mutation → KILLED
9. RemoveSwitch 6 mutation → SURVIVED
10. Switch mutation → KILLED
1. mutated return of Object value for Switch1::workingWeek to ( if (x != null) null else throw new RuntimeException ) → KILLED
```

- PIT detects the four missing tests

PIT and Switch Statements (cont.)

- Running all 7 tests still doesn't get us coverage of the JBC generated



PIT and Switch Statements (cont.)

- Let's run PIT on the test with all 7 tests we get full coverage (code is without the default in the Switch)

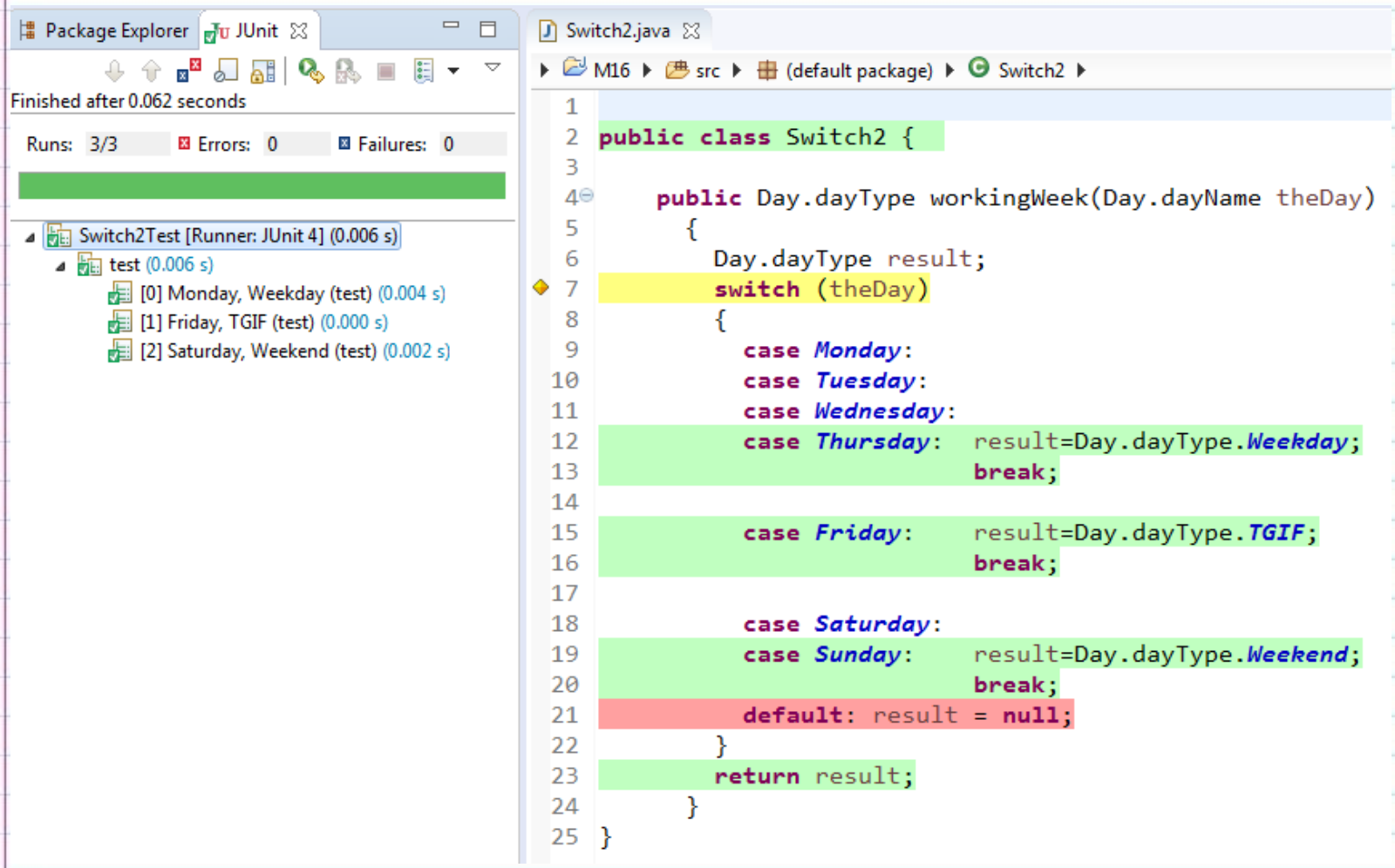
Switch1.java

Mutations

```
1. removed call to Switch1::$SWITCH_TABLE$Day$dayName → KILLED
2. removed call to Day$dayName::ordinal → KILLED
3. RemoveSwitch 0 mutation → KILLED
4. RemoveSwitch 1 mutation → KILLED
5. RemoveSwitch 2 mutation → KILLED
6. RemoveSwitch 3 mutation → KILLED
7. RemoveSwitch 4 mutation → KILLED
8. RemoveSwitch 5 mutation → KILLED
9. RemoveSwitch 6 mutation → KILLED
10. Switch mutation → KILLED
1. mutated return of Object value for Switch1::workingWeek to ( if (x != null) null else throw new RuntimeException ) → KILLED
```

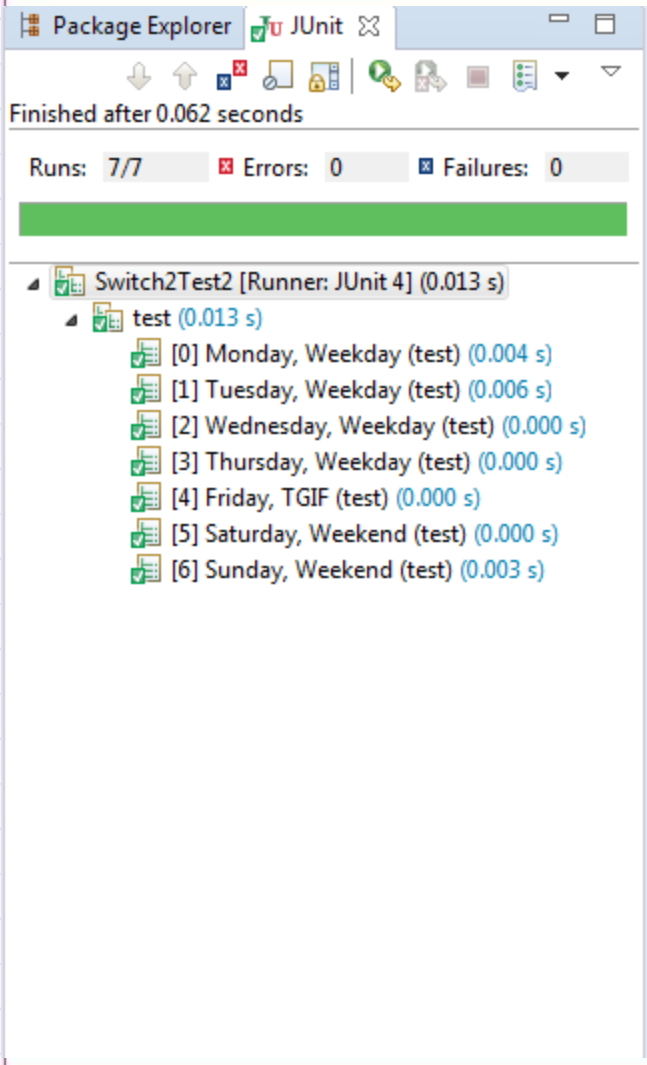
PIT and Switch Statements (cont.)

- If we add a default to the source code and run just three tests we see the



PIT and Switch Statements (cont.)

- If we run all seven tests we see the same result because we can't cover



Switch2.java

M16 ▸ src ▸ (default package) ▸ Switch2

```
1
2  public class Switch2 {
3
4      public Day.dayType workingWeek(Day.dayName theDay)
5        {
6          Day.dayType result;
7          switch (theDay)
8          {
9            case Monday:
10           case Tuesday:
11           case Wednesday:
12           case Thursday:   result=Day.dayType.Weekday;
13                            break;
14
15           case Friday:     result=Day.dayType.TGIF;
16                            break;
17
18           case Saturday:
19           case Sunday:     result=Day.dayType.Weekend;
20                            break;
21           default: result = null;
22         }
23         return result;
24       }
25 }
```

PIT and Switch Statements (cont.)

- Let's run PIT on the test with all 7 tests we get full coverage (code is **with** the default in the Switch)

Switch2.java

Mutations

```
1. removed call to Switch2::$SWITCH_TABLE$Day$dayName → KILLED
2. removed call to Day$dayName::ordinal → KILLED
3. RemoveSwitch 0 mutation → KILLED
4. RemoveSwitch 1 mutation → KILLED
5. RemoveSwitch 2 mutation → KILLED
6. RemoveSwitch 3 mutation → KILLED
7. RemoveSwitch 4 mutation → KILLED
8. RemoveSwitch 5 mutation → KILLED
9. RemoveSwitch 6 mutation → KILLED
10. Switch mutation → KILLED
```

7

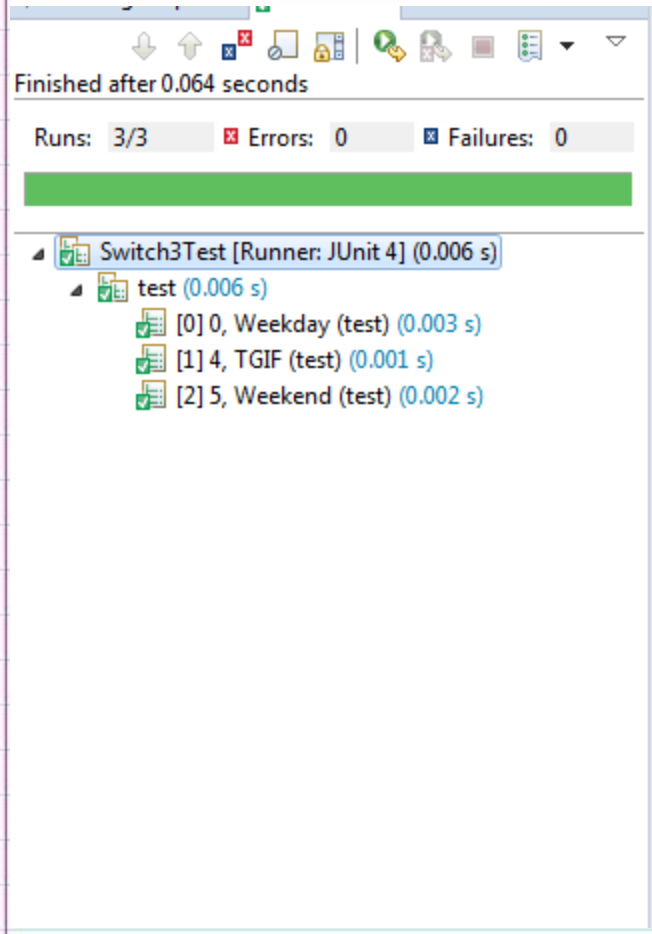
```
1. mutated return of Object value for Switch2::workingWeek to ( if (x != null) null else throw new RuntimeException ) → KILLED
```

23

- PIT knows we can't get the default value in the Switch source

PIT and Switch Statements (cont.)

- Now let's look at an integer based Switch (without a default) running three test cases

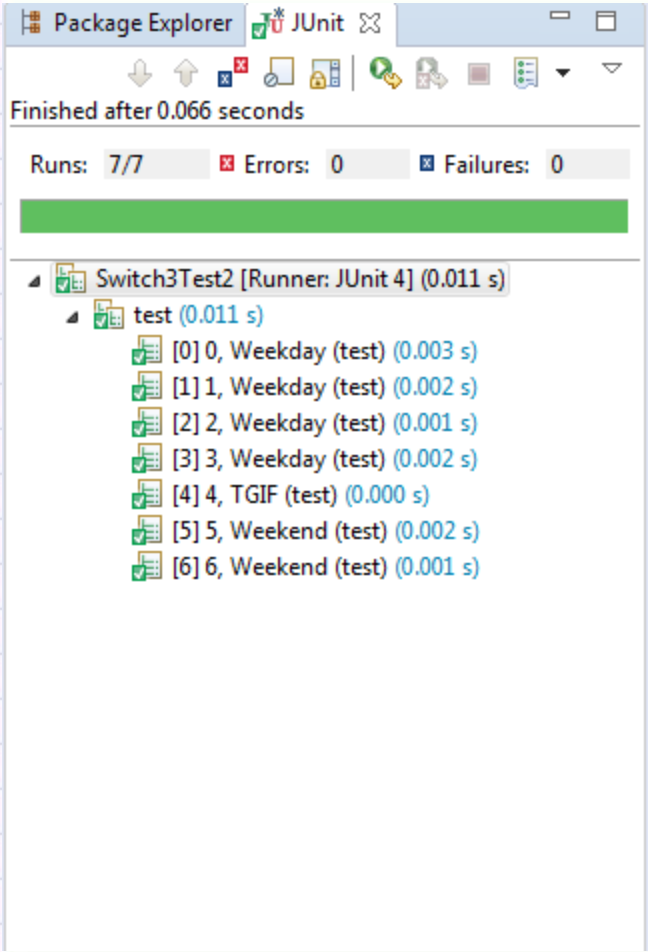


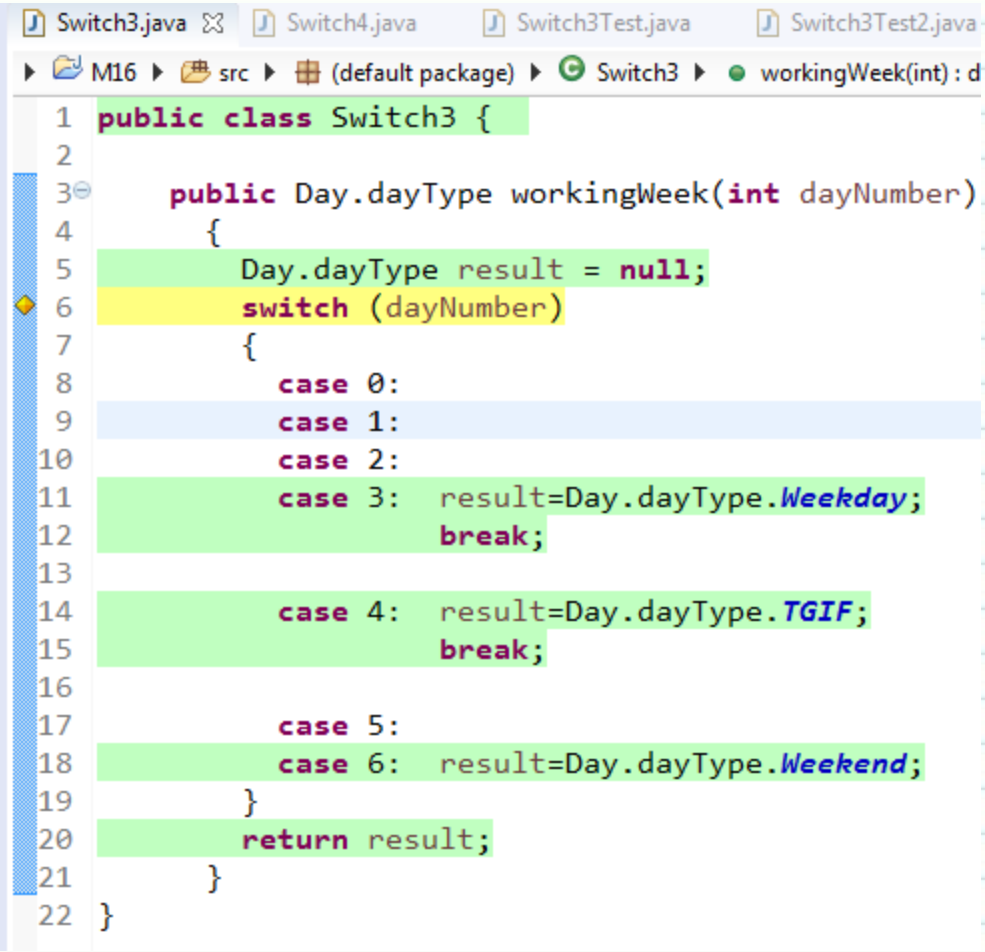
M16 ▸ src ▸ (default package) ▸ Switch3 ▸

```
1  public class Switch3 {
2
3      public Day.dayType workingWeek(int dayNumber)
4        {
5          Day.dayType result = null;
6          switch (dayNumber)
7          {
8            case 0:
9            case 1:
10           case 2:
11           case 3:  result=Day.dayType.Weekday;
12                    break;
13
14           case 4:  result=Day.dayType.TGIF;
15                    break;
16
17           case 5:
18           case 6:  result=Day.dayType.Weekend;
19         }
20         return result;
21       }
22 }
```

PIT and Switch Statements (cont.)

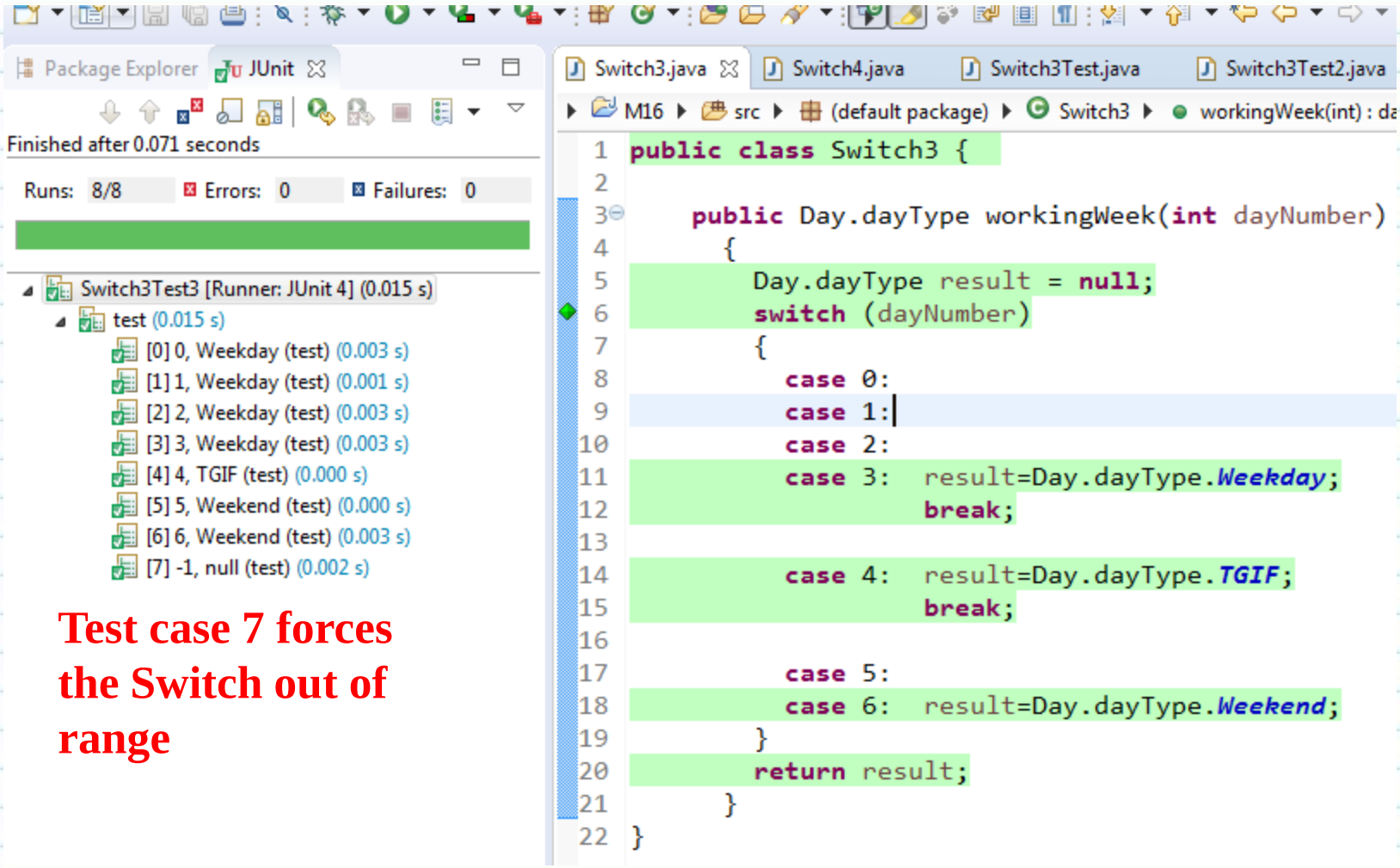
- Now let's look at an integer based Switch (without a default) running seven test cases





PIT and Switch Statements (cont.)

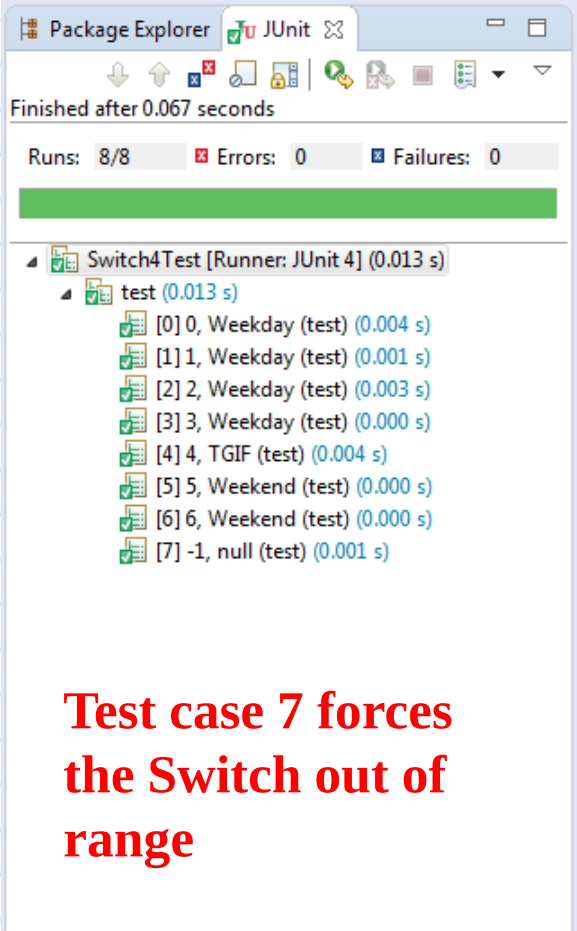
- Now let's look at an integer based Switch (without a default) running **eight** test cases - notice we get full coverage



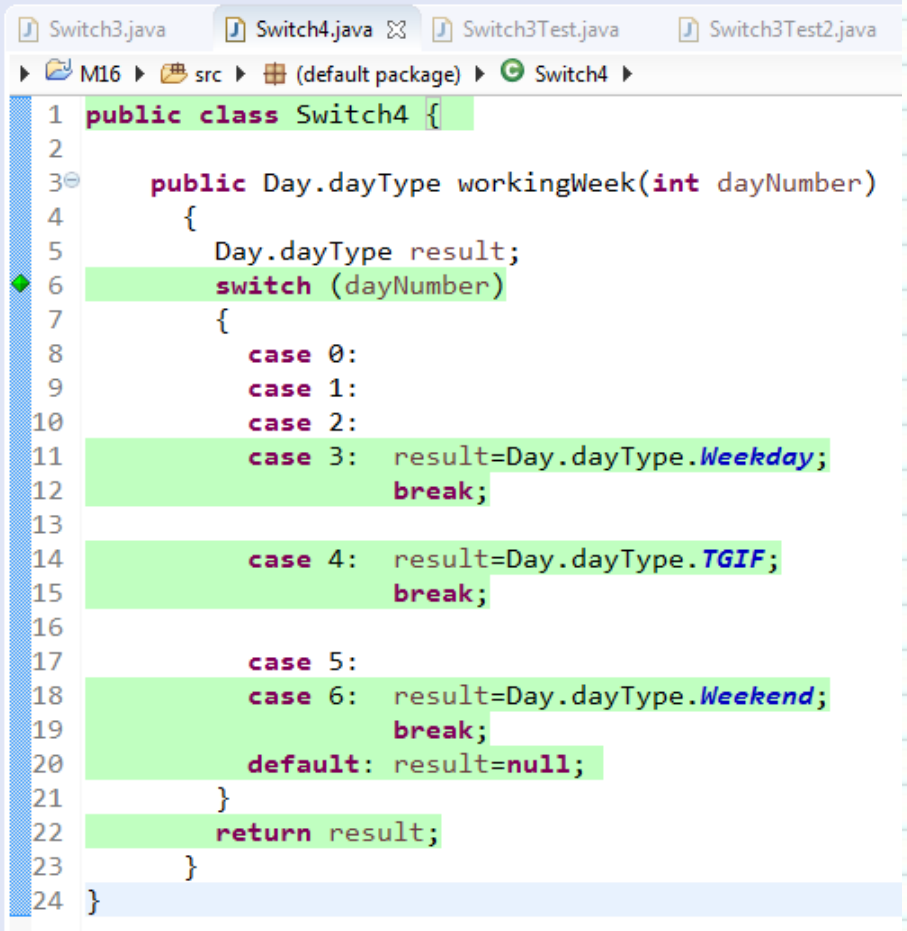
Test case 7 forces the Switch out of range

PIT and Switch Statements (cont.)

- Now let's look at an integer based Switch (**with** a default) running **eight** test cases - notice we get full coverage



Test case 7 forces the Switch out of range



PIT and Switch Statements (cont.)

- Let's look at the PIT results of the previous test

Switch4.java

Mutations

6
1. RemoveSwitch 0 mutation → KILLED
2. RemoveSwitch 1 mutation → KILLED
3. RemoveSwitch 2 mutation → KILLED
4. RemoveSwitch 3 mutation → KILLED
5. RemoveSwitch 4 mutation → KILLED
6. RemoveSwitch 5 mutation → KILLED
7. RemoveSwitch 6 mutation → KILLED
8. Switch mutation → KILLED

22
1. mutated return of Object value for Switch4::workingWeek to (if (x != null) null else throw new RuntimeException) → KILLED

Mathematical Expressions and PIT

Mathematical Expressions and PIT (cont.)

```
3 public class MathExpressionClass {
4   private double y;
5
6   public double calcY (double x) {
7       if (x<-2.0)
8           this.y=0.0;
9       else
10          if (x<0.0)
11              this.y=4*x+8.0;
12          else
13              if (x<4.0)
14                  this.y=-x*x+2*x+8.0;
15              else
16                  if (x<6.0)
17                      this.y=-4.0*x+16;
18                  else
19                      this.y=-8.0;
20      return this.y;
21  }

    public void setY(double y) {
        this.y = y;
    }
}
```

This is the Java code for the previous graph

Mathematical Expressions and PIT (cont.)

```
//          Test case 1
            $(  -0.1,       0.0),
//          Test case 2
            $(   2.0,       7.0),
//          Test case 3
            $(   6.0,       7.0),
//          Test case 4
            $(   8.0,       1.0),
//          Test case 5
            $(   9.0,       1.0),
//          Test case 6
            $(   9.1,       0.0),
//          Test case 7
            $(  -2.0,       0.0),
//          Test case 8
            $(   0.0,       0.0),
//          Test case 9
            $(   2.1,       6.61),
//          Test case 10
            $(   6.1,       6.7),
//          Test case 11
            $(   8.1,       1.0),
//          Test case 12
            $(  10.0,       0.0),
//          Test case 13
            $(   1.0,       3.5),
//          Test case 14
            $(   4.0,       3.0),
//          Test case 15
            $(   5.5,       5.25),
//          Test case 16
            $(   7.0,       4.0)
```

We ended up with 16 test cases when testing BP + ECP/BV + non-uniform regions

Mathematical Expressions and PIT (cont.)

	1. changed conditional boundary → SURVIVED
7	2. Substituted -2.0 with 1.0 → KILLED 3. negated conditional → KILLED 4. removed conditional - replaced comparison check with false → KILLED 5. removed conditional - replaced comparison check with true → KILLED
8	1. Substituted 0.0 with 1.0 → KILLED 2. Removed assignment to member variable y → KILLED
	1. changed conditional boundary → SURVIVED 2. Substituted 0.0 with 1.0 → SURVIVED
10	3. negated conditional → KILLED 4. removed conditional - replaced comparison check with false → KILLED 5. removed conditional - replaced comparison check with true → KILLED
11	1. Substituted 4.0 with 1.0 → KILLED 2. Substituted 8.0 with 1.0 → KILLED 3. Replaced double multiplication with division → KILLED 4. Replaced double addition with subtraction → KILLED 5. Removed assignment to member variable y → KILLED
	1. changed conditional boundary → SURVIVED 2. Substituted 4.0 with 1.0 → KILLED
13	3. negated conditional → KILLED 4. removed conditional - replaced comparison check with false → KILLED 5. removed conditional - replaced comparison check with true → KILLED
14	1. Substituted 2.0 with 1.0 → KILLED 2. Substituted 8.0 with 1.0 → KILLED 3. removed negation → KILLED 4. Replaced double multiplication with division → KILLED 5. Replaced double multiplication with division → KILLED 6. Replaced double addition with subtraction → KILLED 7. Replaced double addition with subtraction → KILLED 8. Removed assignment to member variable y → KILLED
	1. changed conditional boundary → SURVIVED 2. Substituted 6.0 with 1.0 → KILLED
16	3. negated conditional → KILLED 4. removed conditional - replaced comparison check with false → KILLED 5. removed conditional - replaced comparison check with true → SURVIVED
17	1. Substituted -4.0 with 1.0 → KILLED 2. Substituted 16.0 with 1.0 → KILLED 3. Replaced double multiplication with division → KILLED 4. Replaced double addition with subtraction → KILLED 5. Removed assignment to member variable y → KILLED

PIT detects these surviving mutations

Mathematical Expressions and PIT (cont.)

19	1. Substituted -8.0 with 1.0 → KILLED 2. Removed assignment to member variable y → KILLED
20	1. replaced return of double value with -(x + 1) for MathExpressionClass::calcY → KILLED
24	1. Removed assignment to member variable y → SURVIVED

PIT detects these surviving mutations

Why are we getting these surviving mutations?

Mathematical Expressions and PIT (cont.)

19	1. Substituted -8.0 with 1.0 → KILLED 2. Removed assignment to member variable y → KILLED
20	1. replaced return of double value with -(x + 1) for MathExpressionClass::calcY → KILLED
24	1. Removed assignment to member variable y → SURVIVED

PIT detects these surviving mutations

Why are we getting these surviving mutations? Because these are on contiguous boundaries - changing the expression

"if (x <-2.0)" to "if (x <=-2.0)"

has the same answer of y=0.

Mutation Test Tool Support

- Mutation testing requires tool support as it may generate a large number of mutants
- Support for Java mutant testing
  - javalanche - bytecode-based
  - jester - source-code based
  - judy - source-code based
  - jumble - bytecode-based
  - major - compiler-integrated mutation testing framework
  - muJava - includes class-level operators
  - mutator - source based commercial mutation analyzer
  - pit - bytecode-based
- Java has considerably more tools for Mutation testing than any other language

Fuzzing - A Special Case of Mutation Testing

- Fuzzing is a special case of typically automated testing where invalid, random, unexpected data is input to a software program. The program is then monitored for exceptions which can include crashes, failing assertions or potential memory leaks.
- It is commonly used to test for security problems in software.
- Fuzzing can also be performed by mutating data inputs to the software program.