Principles of Software Construction: Objects, Design, and Concurrency

Test case design

Christian Kästner **Vincent Hellendoorn**

Carnegie Mellon University
School of Computer Science

institute for SOFTWARE RESEARCH

Last Week

- Unit testing: small, simple, per-method tests
- Specification vs. Structural testing

Little Quiz

https://forms.gle/Am48bu6avqLh7ytb9

Note on Precondition Testing

question @175

24 views

Actions

HW2 - Testing constructor for RepeatingCardOrganizer

How should I test the constructor for RepeatingCardOrganizer?

The javadoc mentions that `repetitions` must be positive, but it doesn't explicitly say that an exception / error will be thrown (like AssertionError) if that is violated.

```
/**
     * Creates a RepeatingCardSorter instance.
     *
     * @param repetitions The number of repetitions to require of each card. Must be positive.
     */
    public RepeatingCardOrganizer(int repetitions) {
        assert repetitions >= 1;
        this.repetitions = repetitions;
    }
```

run code snippet Visit 'Manage Class' to disable runnable code snippets

I understand that we shouldn't assume anything not stated (an exception / error will be thrown). But if we don't do that, the behavior of the RepeatingCardOrganizer will be undefined if we pass an invalid value.

How should we deal with that?

Today

- Structural Testing Strategies
  - Statement, branch, path coverage; limitations
- Writing testable code & good tests
- Specification Testing Strategies
  - Boundary value analysis, combinatorial testing, decision tables
- Bit of both

Structural Testing: a closer look

Takes into account the internal mechanism of a system (IEEE, 1990).

- Approaches include tracing data and control flow through a program

Case Study

Assume various Wallets

```
public interface Wallet {

   boolean pay(int cost);

   int getValue();

}
```

DebitWallet.pay()

What should we test in this code?

```
public boolean pay(int cost) {
    if (cost <= this.money) {
        this.money -= cost;
        return true;
    }
    return false;
}
```

DebitWallet.pay()

```
public boolean pay(int cost) {
    if (cost <= this.money) {
        this.money -= cost;
        return true;
    }
    return false;
}

new DebitWallet(100).pay(10);
```

DebitWallet.pay()

```
public boolean pay(int cost) {
    if (cost <= this.money) {
        this.money -= cost;
        return true;
    }
    return false;
}

new DebitWallet(0).pay(10);
```

CreditWallet.pay()

How about now?

```
public boolean pay(int cost, boolean useCredit) {
    if (useCredit) {
        if (this.credit + cost <= this.maxCredit) {
            this.credit += cost;
            return true;
        }
    }
    if (cost <= this.cash) {
        this.cash -= cost;
        return true;
    }
    return false;
}
```

CreditWallet.pay()

```
public boolean pay(int cost, boolean useCredit) {
   if (useCredit) {
       if (enoughCredit) {
           return true;
       }
   }
   if (enoughCash) {
       return true;
   }
   return false;
}
```

Exercise: think about as many test scenarios as you can

CreditWallet.pay()

```
public boolean pay(int cost, boolean useCredit) {
    if (useCredit) {
        if (enoughCredit) {
            return true;
        }
    }
    if (enoughCash) {
        return true;
    }
    return false;
}
```

Test case	useCredit	Enough Credit	Enough Cash	Result	Coverage
1	T	T	-	Pass	??

CreditWallet.pay()

```
public boolean pay(int cost, boolean useCredit) {
    if (useCredit) {
        if (enoughCredit) {
            return true;
        }
    }
    if (enoughCash) {
        return true;
    }
    return false;
}
```

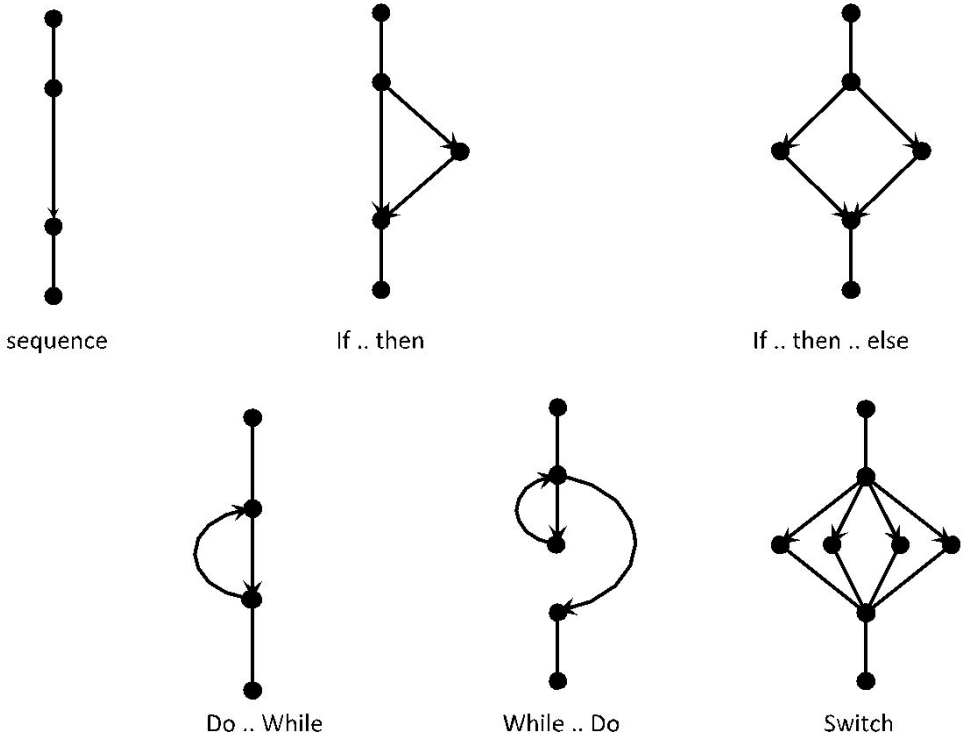
Test case	useCredit	Enough Credit	Enough Cash	Result	Coverage
1	T	T	-	Pass	--
2	F	-	T	Pass	--
3	F	-	F	Fails	Statement

Coverage

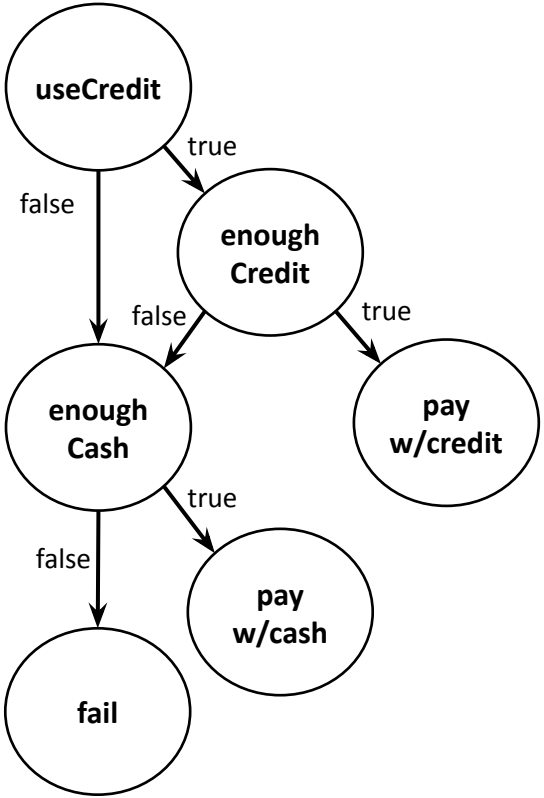
We have tested every statement; are we done?
Depends on desired **coverage**:

- Provide at least one test for distinct types of behavior
- Typically on control flow paths through the program
- Statement, branch, basis paths, MC/DC

Structures in Code

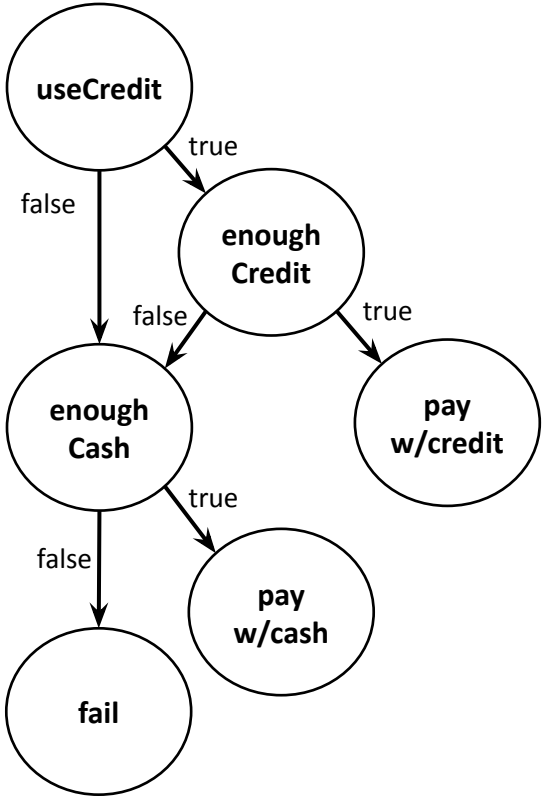


Control-Flow of CreditCard.pay()



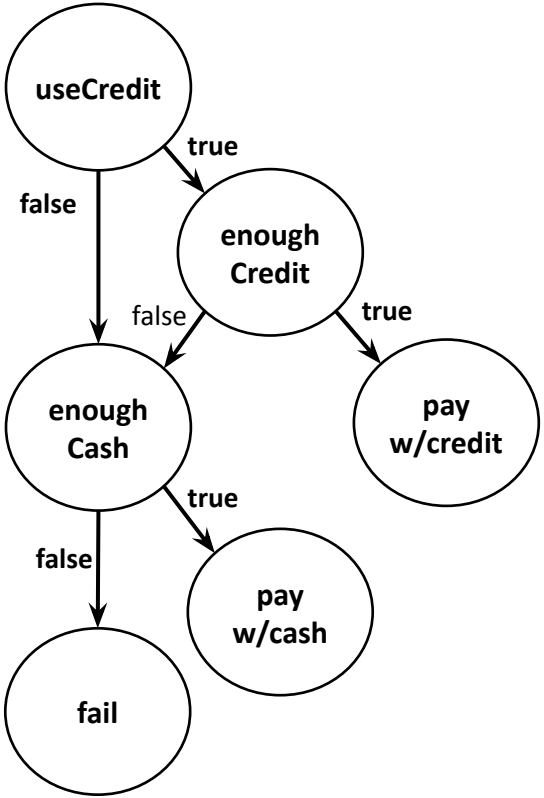
Control-Flow of CreditCard.pay()

Test case	useCredit	Enough Credit	Enough Cash	Result	Coverage
1	T	T	-	Pass	--
2	F	-	T	Pass	--
3	F	-	F	Fails	Statement



Control-Flow of CreditCard.pay()

Test case	useCredit	Enough Credit	Enough Cash	Result	Coverage
1	T	T	-	Pass	--
2	F	-	T	Pass	--
3	F	-	F	Fails	Statement



CreditWallet.pay()

```
public boolean pay(int cost, boolean useCredit) {
    if (useCredit) {
        if (enoughCredit) {
            return true;
        }
    }
    if (enoughCash) {
        return true;
    }
    return false;
}
```

Test case	useCredit	Enough Credit	Enough Cash	Result	Coverage	
1	T	T	-	Pass	--	
2	F	-	T	Pass	--	
3	F	-	F	Fails	Statement	
4	T	F	T	Pass	Branch	

Path Coverage

We have seen every condition … what else is missing?

Path Coverage

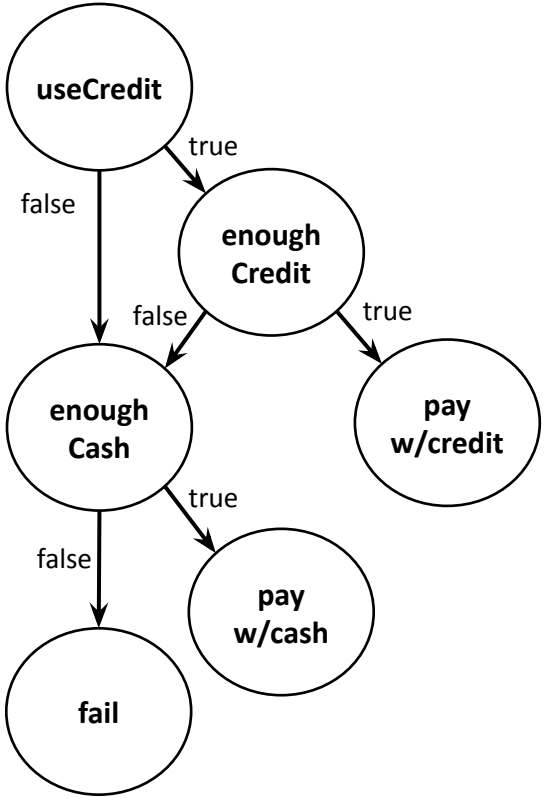
We have seen every condition … but not every path.

- 3 conditions, each with two values = 8 permutations
- Some permutations are impossible
- Still one *path* left

Control-Flow of CreditCard.pay()

Paths:

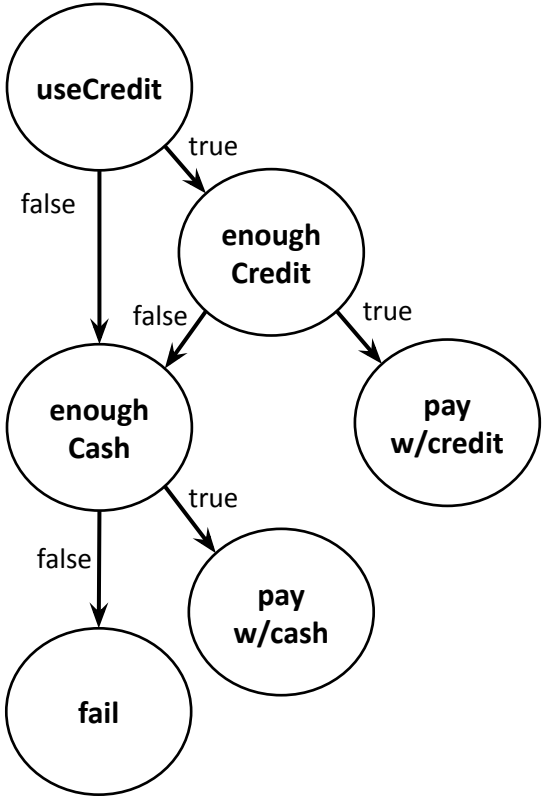
- {true, true}: pay w/credit
- {false, true}: pay w/cash
- {false, false}: fail



Control-Flow of CreditCard.pay()

Paths:

- {true, true}: pay w/credit
- {false, true}: pay w/cash
- {false, false}: fail
- {true, false, true}: pay w/cash after failing credit
- {true, false, false}: try credit, but fail, **and** no cash



CreditWallet.pay()

```
public boolean pay(int cost, boolean useCredit) {
    if (useCredit) {
        if (enoughCredit) {
            return true;
        }
    }
    if (enoughCash) {
        return true;
    }
    return false;
}
```

Test case	useCredit	Enough Credit	Enough Cash	Result	Coverage
1	T	T	-	Pass	--
2	F	-	T	Pass	--
3	F	-	F	Fails	Statement
4	T	F	T	Pass	Branch
5	T	F	F	Fails	(Basis) paths

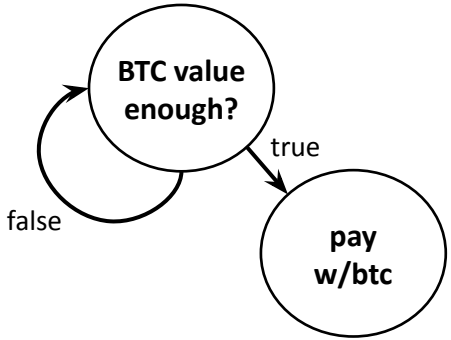
BitCoinWallet.pay()

```
public boolean pay(int cost) {
    int currValue;
    while ((currValue = getValue()) < cost) {
        // Just wait.
    }
    this.btc -= cost / currValue;
    return true;
}

public int getValue() {
    return (int)
      (this.btc * Math.pow(2, 20*Math.random()));
}
```

Control-flow of BitCoinWallet.pay()

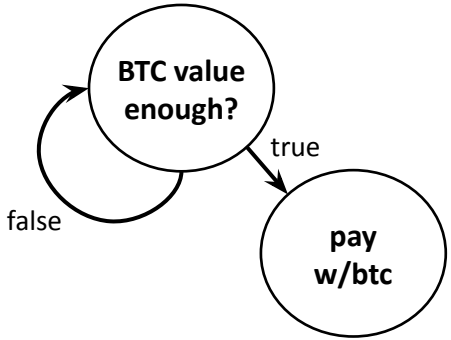
What are all the paths?



Control-flow of BitCoinWallet.pay()

What are all the paths?

- {true}
- {false, true}
- {false, false, true}
- {false, false, false, true}
- ...

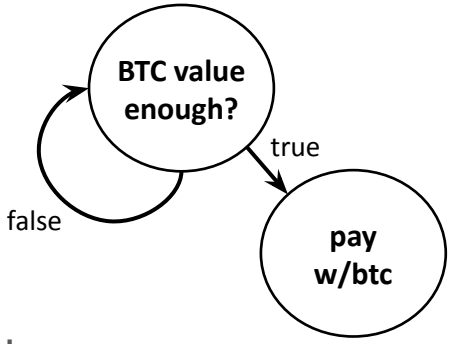


Control-flow of BitCoinWallet.pay()

Perfect “general” path coverage is elusive

But “adequate” coverage criteria exist:

- Basis paths: each path must cover one new *edge*
  - {true} and {false, true} are sufficient
  - As is just {false, true}
- Loop adequacy: iterate each loop zero, one, and 2+ times

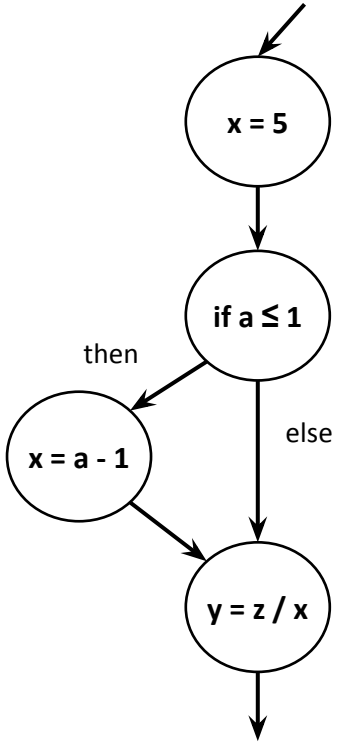


More Coverage

Many more criteria exist:

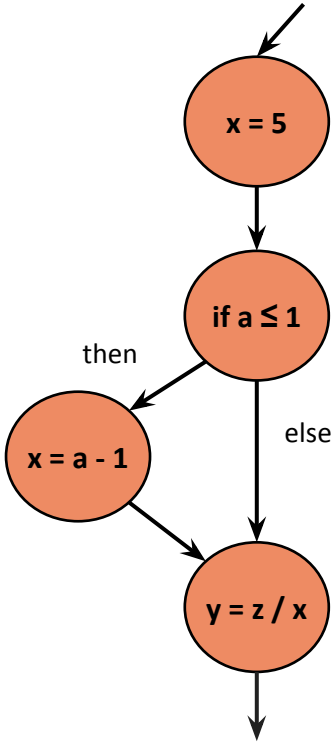
- For branches with multiple conditions
  - Modified Condition/Decision Coverage is quite popular
- For loops
  - Boundary Interior Testing
- Branch coverage is by far the most common

Coverage and Quality



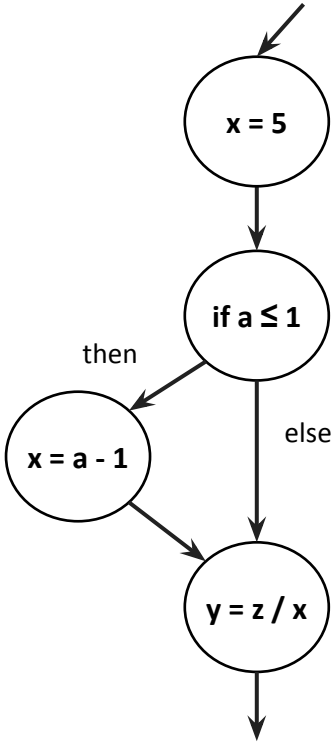
Question 1: Is there a defect?

Coverage and Quality



Question 2: Can we achieve 100% **statement** coverage and miss the defect?

Coverage and Quality



Question 3: Can we achieve 100% **branch** coverage and miss the defect?

Outline

- Structural Testing Strategies
- **Writing testable code & good tests**
- Specification Testing Strategies

Writing Testable Code

What is the problem with this?

```
public boolean hasHeader(String path) throws IOException {
   List<String> lines = Files.readAllLines(Path.of(path));
   return !lines.get(0).isEmpty()
}

// complete control-flow coverage!
hasHeader("cards.csv") // true
```

Writing Testable Code

What is the problem with this?

```
public boolean hasHeader(String path) throws IOException {
   List<String> lines = Files.readAllLines(Path.of(path));
   return !lines.get(0).isEmpty()
}

// to achieve a 'false' output:
try {
   Path tempFile = Files.createTempFile(null, null);
   Files.write(tempFile,"\n".getBytes(StandardCharsets.UTF_8));
   hasHeader(tempFile.toFile().getAbsolutePath()); // false
} catch (IOException e) {
   e.printStackTrace();
}
```

Writing Testable Code

Exercise: rewrite to make this easier

- And: what would you test?

```
public boolean hasHeader(String path) throws IOException {
   List<String> lines = Files.readAllLines(Path.of(path));
   return !lines.get(0).isEmpty()
}
```

Writing Testable Code

Aim to write easily testable code

- Which is almost by definition more modular

```
public List<String> getLines(String path) throws IOException {
   return Files.readAllLines(Path.of(path));
}

public boolean hasHeader(List<String> lines) {
   return !lines.get(0).isEmpty()
}

// Test:
// - hasHeader with empty, non-empty first line
// - getLines (if you must) with null, real path
```

Writing Testable Code

What is the problem with this?

```
public String[] getHeaderParts(List<String> lines) {
   if (!lines.isEmpty()) {
       String header = lines.get(0);
       if (header.contains(",")) {
           return header.split(",");
       } else {
           return new String[0];
       }
   } else {
       return null;
   }
}
```

Writing Testable Code

Split functionality into easily testable **units**

```
public String getFirstLine(List<String> lines) {
    if (!lines.isEmpty()) {
        return lines.get(0);
    } else {
        return null;
    }
}

public String[] getHeaderParts(String header) {
    if (header.contains(",")) {
        return header.split(",");
    } else {
        return new String[0];
    }
}
```

Clean Testing

What is the problem with this?

```
public String[] getHeaderParts(String header) {
   if (header.contains(",")) {
       return header.split(",");
   } else {
       return null;
   }
}

@Test
public void testGetHeaderParts() {
   for (String header : List.of("line", "", "one,two")) {
      String[] parts = getHeaderParts(line);
      if (header.contains(",")) assertNull(parts);
      else assertEqual(header.split(","), parts.length);
   }
}
```

Clean Testing

Keep tests simple, small

```
public String[] getHeaderParts(String header) {
   if (header.contains(",")) {
       return header.split(",");
   } else {
       return null;
   }
}

@Test
public void testGetHeaderPartsNoComma() {
   String[] parts = getHeaderParts("line");
   assertNull(parts);
}

@Test
...
```

Testing Best Practices

Coverage is useful, but no substitute for your insight

- Cannot capture all paths
  - Especially beyond “unit”
  - Write testable code
- You may be testing buggy code
  - (add regression tests)
- Aim for at least branch coverage
  - And think through scenarios that demand more

Outline

- Structural Testing Strategies
- Writing testable code & good tests
- **Specification Testing Strategies**

Back to Specification Testing

What would you test differently in this situation?

- Previously identified five paths through the code. Are there still?
- Should we test anything new?

```
/** Pays with credit if useCredit is set and enough
  * credit is available; otherwise, pays with cash if
  * enough cash is available; otherwise, returns false.
  */
public boolean pay(int cost, boolean useCredit);
```

Back to Specification Testing

What would you test differently in this situation?

- “if useCredit is set and enough credit is available”:
  - Test both true, either/both false
- “pays with cash if enough cash is available; otherwise”:
  - Test true, false
- Could to this with three test cases

```
/** Pays with credit if useCredit is set and enough
  * credit is available; otherwise, pays with cash if
  * enough cash is available; otherwise, returns false.
  */
public boolean pay(int cost, boolean useCredit);
```

Specification Testing

We need a *strategy* to identify plausible mistakes

Specification Testing

We need a *strategy* to identify plausible mistakes

- Random: avoids bias, but inefficient
  - Yet potentially *very* valuable, because automatable
  - Not for today

Boundary Value Testing

We need a *strategy* to identify plausible mistakes

- Boundary Value Testing: errors often occur at boundary conditions
  - E.g.:

```
/** Returns true and subtracts cost if enough
  * money is available, false otherwise.
  */
public boolean pay(int cost) {
   if (cost < this.money) {
       this.money -= cost;
       return true;
   }
   return false;
}
```

Boundary Value Testing

We need a *strategy* to identify plausible mistakes

- Boundary Value Testing: errors often occur at boundary conditions
  - Identify equivalence partitions: regions where behavior should be the same
    - `cost <= money: true, cost > money: false`
    - Boundary value: `cost == money`

```
/** Returns true and subtracts cost if enough
  * money is available, false otherwise.
  */
public boolean pay(int cost) {
   if (cost < this.money) {
       this.money -= cost;
       return true;
   }
   return false;
}
```

Boundary Value Testing

We need a *strategy* to identify plausible mistakes

- Boundary Value Testing: errors often occur at boundary conditions
  - Select: a nominal/normal case, a boundary value, and an abnormal case
  - Useful for few *categories* of behavior (e.g., null/not-null) per value
- Test: `cost < credit`, `cost == credit`, `cost > credit`, `cost < cash`, `cost == cash`, `cost > cash`

```
/** Pays with credit if useCredit is set and enough
  * credit is available; otherwise, pays with cash if
  * enough cash is available; otherwise, returns false.
  */
public boolean pay(int cost, boolean useCredit);
```

Combinatorial Testing

We need a *strategy* to identify plausible mistakes

- Combinatorial Testing: focus on tuples of boundary values
  - Captures bugs in **interactions** between risky inputs
  - Rarely need to test pairs of “invalid” values (cost too high for credit & cash)

```
/** Pays with credit if useCredit is set and enough
  * credit is available; otherwise, pays with cash if
  * enough cash is available; otherwise, returns false.
  */
public boolean pay(int cost, boolean useCredit);
```

Combinatorial Testing

We need a *strategy* to identify plausible mistakes

- Combinatorial Testing: focus on tuples of boundary values
  - Captures bugs in **interactions** between risky inputs
  - Rarely need to test pairs of “invalid” values (cost too high for credit & cash)
- Include: {cost > credit && cost == cash}
- Maybe: {cost < credit && cost == cash}

```
/** Pays with credit if useCredit is set and enough
  * credit is available; otherwise, pays with cash if
  * enough cash is available; otherwise, returns false.
  */
public boolean pay(int cost, boolean useCredit);
```

Decision Tables

We need a *strategy* to identify plausible mistakes

- Decision Tables
  - You've seen one already
  - Enumerate condition options
    - Leave out impossibles
    - Identify "don't-matter" values
  - Useful for redundant input domains

Test case	useCredit	Enough Credit	Enough Cash	Result
1	T	T	-	Pass
2	F	-	T	Pass
3	F	-	F	Fails
4	T	F	T	Pass
5	T	F	F	Fails

Specification Tests

So what is the right granularity?

- It depends
- We are still aiming for coverage
  - Just of specifications, and their innumerable implementations
  - BVA (& its cousins), decision tables tend to provide good coverage

Structural Testing vs. Specification Testing

You will ***typically have both*** code & (prose) specification

- Test specification, but know that it can be underspecified
- Test implementation, but not to the point that it cannot change
- Use testing strategies that leverage both
  - There is a fair bit of overlap; e.g., BVA yields useful branch coverage

Further Testing Strategies

Many more aspects, some later in this course:

- Stubbing/Mocking, to avoid testing dependencies
- Integration testing: scenarios that span units
- Beyond correctness: performance, security

Summary

Testing comprehensively is hard

- Tailor to your task: specification vs. structural testing
  - Do not assume unstated specifications for part 2; spend your energy wisely in part 3
- Pick a strategy, or a few
  - Be systematic; defend your decisions
- Tomorrow's recitation covers:
  - Unit test best practices
  - Test organization
  - Running tests, coverage; Travis setup

Bonus: Coding like the tour the france

```
public boolean foo() {
    try {
        synchronized () {
            if () {
            } else {
            }
            for () {
                if () {
                    if () {
                        if () {
                            if ()
                            {
                                if () {
                                    for () {
                                    }
                                }
                            }
                        } else {
                            if () {
                                for () {
                                    if () {
                                    } else {
                                    }
                                    if () {
                                    } else {
                                        if () {
                                        }
                                    }
                                    if () {
                                        if () {
                                            if () {
                                                for () {
                                                }
                                            }
                                        }
                                    } else {
                                    }
                                }
                            } else {
                            }
                        }
                    }
                }
            }
        }
```

https://thedailywtf.com/articles/coding-like-the-tour-de-france