Principles of Software Construction: Objects, Design, and Concurrency

Test case design

Bogdan Vasilescu   Jonathan Aldrich

Carnegie Mellon University
School of Computer Science

institute for
SOFTWARE
RESEARCH

S3D

Administrative

- Canvas submissions
  - "Submit all code of your solution to GitHub and submit a link to your final commit here on Canvas in the form *https://github.com/CMU-17-214/<reponame>/commit/<commitid>*"
- Homework 2 is due next week: testing
  - Lots of useful stuff in recitation on Wednesday

S3D

Quiz Time

Under "Quizzes → Lecture 5 Quiz" on Canvas.

Last Week

- Exception handling
- Unit testing: small, simple, per-method tests
- Started talking about specifications

S3D

Today

- Specifications
- Specification vs. Structural testing
- Testing Strategies
  - Structural Testing: Statement, branch, path coverage; limitations
  - Specification Testing: Boundary value analysis, combinatorial testing, decision tables
- Writing testable code & good tests

Specifications and testing are closely related

Q: What exactly do you test when given a method?

- What it claims to do: specification testing – the contract (last week)
- What it does: structural testing – coverage

S3D

What is a contract?

- Agreement between an object and its user
  - What object provides, and user can count on
- Includes:
  - Method signature (type specifications)
  - Functionality and correctness expectations
  - Sometimes: performance expectations
- **What** the method does, not **how** it does it
  - **Interface** (API), not **implementation**
- "Focus on concepts rather than operations"

S3D

Method contract details

- Defines method's and caller's responsibilities
- Analogy: legal contract
  - If you pay me this amount on this schedule…
  - I will build a room with the following detailed spec
  - Some contracts have remedies for nonperformance
- Method contract structure
  - Preconditions: what method requires for correct operation
  - Postconditions: what method establishes on completion
  - Exceptional behavior: what it does if precondition violated
- Defines correctness of implementation

S3D

How to Encode Specifications?

Formal frameworks exist, to capture pre- and post-conditions

- **E.g.,** `requires arr != null`
- Useful for formal verification
- But rarely used in general-purpose application software
  - Takes a lot of effort, and doesn't scale well

S3D

How to Encode Specifications?

Most common: prose specification.

```
class Algorithms {
    /**
     * This method finds the
     * shortest distance between two
     * vertices. It returns -1 if
     * the two nodes are not
     * connected. */
    int shortestDistance(…) {…}
}
```

Recall the earlier example?
(Probably too unstructured)

How to Encode Specifications?

Most common: prose specification.

Document:

- Every parameter
- Return value
- Every exception (checked and unchecked)
- What the method does, including
  - Primary purpose
  - Any side effects
  - Any thread safety issues
  - Any performance issues

S3D

How to Encode Specifications?

Most common: prose specification.

Document:

- Every parameter
- Return value
- Every exception (checked and unchecked)
- What the method does, including
  - Primary purpose
  - Any side effects
  - Any thread safety issues
  - Any performance issues

Do **not** document implementation details

- Known as overspecification

S3D

Docstring Specification

```java
class RepeatingCardOrganizer {
  ...



  public boolean isComplete(CardStatus card) {
    return card.getResults().stream()
      .filter(isSuccess -> isSuccess)
      .count() >= this.repetitions;
  }
}
```

S3D

Docstring Specification

```java
class RepeatingCardOrganizer {
  ...
  /**
   * Checks if the provided card has been answered correctly the required
number of times.
   * @param card The {@link CardStatus} object to check.
   * @return {@code true} if this card has been answered correctly at least
{@code this.repetitions} times.
   */
  public boolean isComplete(CardStatus card) {
    return card.getResults().stream()
      .filter(isSuccess -> isSuccess)
      .count() >= this.repetitions;
  }
}
```

S3D

Docstring Specification

```java
class RepeatingCardOrganizer {
  ...
  /**
   * Checks if the provided card has been answered correctly the required
number of times.
   * @param card The {@link CardStatus} object to check.
   * @return {@code true} if this card has been answered correctly at least
{@code this.repetitions} times.
   */
  public boolean isComplete(CardStatus card) {
    // IGNORE THIS WHEN SPECIFICATION TESTING!
  }
}
```

15 S3D

Docstring Specification

```java
/**
 * Checks if the provided card has been answered correctly the required
number of times.
 * @param card The {@link CardStatus} object to check.
 * @return {@code true} if this card has been answered correctly at least
{@code this.repetitions} times.
 */
public boolean isComplete(CardStatus card);

// What is specified?
```

S3D

Docstring Specification

```java
/**
 * Checks if the provided card has been answered correctly the required
number of times.
 * @param card The {@link CardStatus} object to check.
 * @return {@code true} if this card has been answered correctly at least
{@code this.repetitions} times.
 */
public boolean isComplete(CardStatus card);

// What is specified?
// - What the method does (but not how)
```

17
S3D

Docstring Specification

```java
/**
 * Checks if the provided card has been answered correctly the required
number of times.
 * @param card The {@link CardStatus} object to check.
 * @return {@code true} if this card has been answered correctly at least
{@code this.repetitions} times.
 */
public boolean isComplete(CardStatus card);

// What is specified?
// - What the method does (but not how)
// - Parameter type (no constraints)
```

S3D

Docstring Specification

```java
/**
 * Checks if the provided card has been answered correctly the required
number of times.
 * @param card The {@link CardStatus} object to check.
 * @return {@code true} if this card has been answered correctly at least
{@code this.repetitions} times.
 */
public boolean isComplete(CardStatus card);

// What is specified?
// - What the method does (but not how)
// - Parameter type (no constraints)
// - Return constraints: "at least" this.repetitions correct answers
```

S3D

Docstring Specification

```java
/**
 * Checks if the provided card has been answered correctly the required
number of times.
 * @param card The {@link CardStatus} object to check.
 * @return {@code true} if this card has been answered correctly at least
{@code this.repetitions} times.
 */
public boolean isComplete(CardStatus card);

// What is specified?
// - Parameter type (no constraints)
// - Return constraints: "at least" this.repetitions correct answers
// So what do we test?
```

Docstring Specification

```java
/**
 * Checks if the provided card has been answered correctly the required
number of times.
 * @param card The {@link CardStatus} object to check.
 * @return {@code true} if this card has been answered correctly at least
{@code this.repetitions} times.
 */
public boolean isComplete(CardStatus card);


@Test
public void testIsCompleteSingleSuccess() {
  CardRepeater repeater = new RepeatingCardOrganizer(1); // Single repetition
  CardStatus cs = new CardStatus(new FlashCard("", ""));
  cs.recordResult(true); // Single Success
  assert???(repeater.isComplete(cs));
}
```

Docstring Specification

```java
    /**
     * Checks if the provided card has been answered correctly the required
number of times.
     * @param card The {@link CardStatus} object to check.
     * @return {@code true} if this card has been answered correctly at least
{@code this.repetitions} times.
     */
    public boolean isComplete(CardStatus card);


@Test
public void testIsCompleteSingleSuccess() {
    CardRepeater repeater = new RepeatingCardOrganizer(1); // Single repetition
    CardStatus cs = new CardStatus(new FlashCard("", ""));
    cs.recordResult(true); // Single Success
    assertTrue(repeater.isComplete(cs));
}
```

S3D

Docstring Specification

```java
/**
 * Checks if the provided card has been answered correctly the required
number of times.
 * @param card The {@link CardStatus} object to check.
 * @return {@code true} if this card has been answered correctly at least
{@code this.repetitions} times.
 */
public boolean isComplete(CardStatus card);

@Test
public void testIsNotCompleteSingleFailure() {
  CardRepeater repeater = new RepeatingCardOrganizer(1); // Single repetition
  CardStatus cs = new CardStatus(new FlashCard("", ""));
  cs.recordResult(false); // Single failure
  assertFalse(repeater.isComplete(cs));
}
```

S3D

Docstring Specification

```
  /**
   * Checks if the provided card has been answered correctly the required
number of times.
   * @param card The {@link CardStatus} object to check.
   * @return {@code true} if this card has been answered correctly at least
{@code this.repetitions} times.
   */
  public boolean isComplete(CardStatus card);
```

We've now run this twice.
Are we done testing?

S3D

Specification vs. Structural Testing

```java
/**
 * Checks if the provided card has been answered correctly the required
number of times.
 * @param card The {@link CardStatus} object to check.
 * @return {@code true} if this card has been answered correctly at least
{@code this.repetitions} times.
 */
public boolean isComplete(CardStatus card) {
  return card.getSuccesses.get(0);  // <-- Bad, but passes both tests
}
```

Specification vs. Structural Testing

You can test for different objectives:
- Specification-based testing: test solely the specification
  - Ignores implementation, use inputs/outputs only
  - Cover all specified behavior
  - <u>Do not rely on code</u>; consider corner-cases
    - Think like an attacker

- Structural Testing: consider implementation
  - Optimize for various kinds of code coverage
    - Line, Statement, Data-flow, etc.
  - By some definitions, we are done. Full line coverage, branch coverage.
    - Which is rarely enough

S3D

Specification vs. Structural Testing

You can test for different objectives:

- Structural Testing: consider implementation
  - Optimize for various kinds of code coverage
    - Line, Statement, Data-flow, etc.
  - By some definitions, we are done. Full line coverage, branch coverage.
    - Which is rarely enough

S3D

Specification vs. Structural Testing

You can test for different objectives:

- Structural Testing: consider implementation
  - Optimize for various kinds of code coverage
    - Line, Statement, Data-flow, etc.
  - By some definitions, we are done. Full line coverage, branch coverage.
    - Which is rarely enough
- Specification-based testing: test solely the specification
  - Ignores implementation, use inputs/outputs only
  - Cover all specified behavior
  - Do not rely on code; consider corner-cases
    - Think like an attacker

Outlook

Homework 2 is all about testing

- Specification-testing the FlashCard system
- Some structural testing as well

So is recitation tomorrow!

S3D

Summary

- Being explicit about program behavior is ideal
  - Helps you detect bugs
  - Forces handling of special cases -- a key source of bugs
  - Increases transparency of your program's interface
- Specification comes in multiple forms
  - Explicit contracts, formal or informal
  - Compile-time signals, e.g. through exceptions
  - Testing helps clarify, often improve specifications
    - TDD takes this to the extreme
    - You rarely know your code until you test it

S3D

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

S3D

Case Study

Assume various Wallets

```java
public interface Wallet {

    boolean pay(int cost);

    int getValue();

}
```

S3D

DebitWallet.pay()

What should we test in this code?

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

S3D

DebitWallet.pay()

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

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

S3D

DebitWallet.pay()

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

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

S3D

CreditWallet.pay()

How about now?

```java
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;
}
```

S3D

CreditWallet.pay()

```java
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()

```java
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	--

S3D

CreditWallet.pay()

```java
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
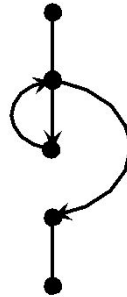
S3D

Structures in Code
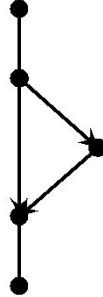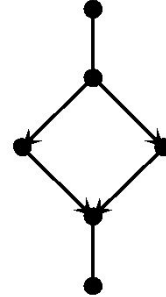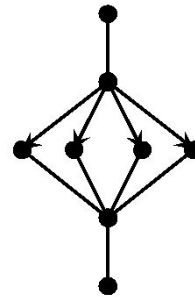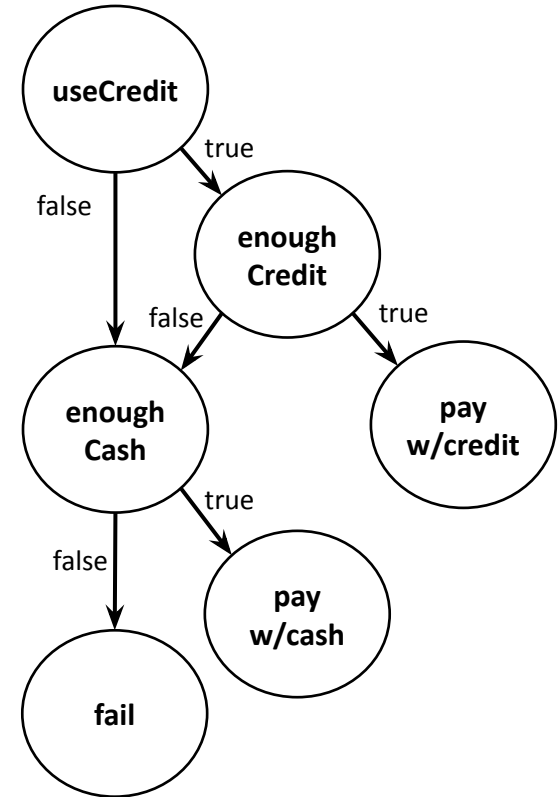

sequence   If .. then   If .. then .. else
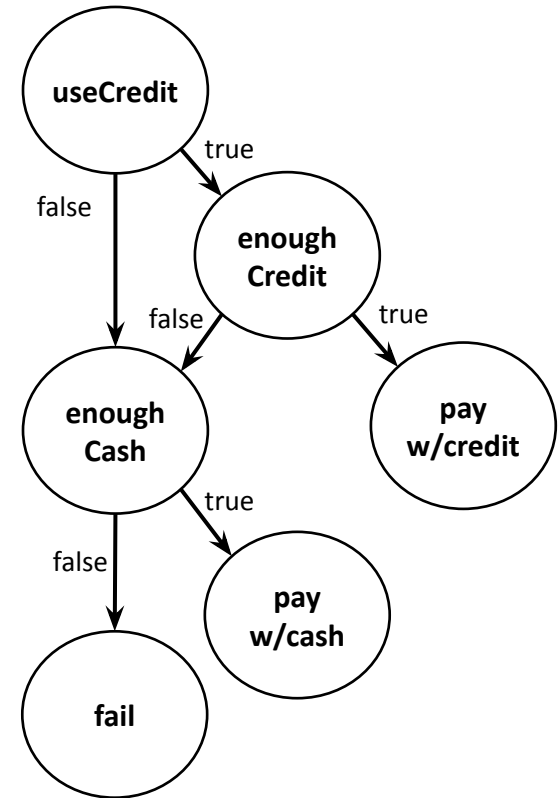
Do .. While   While .. Do   Switch

Control-Flow of CreditCard.pay()

```java
public boolean pay(int cost, boolean useCredit) {
    if (useCredit) {
        if (enoughCredit) {
            return true;
        }
    }
    if (enoughCash) {
        return true;
    }
    return false;
}
```
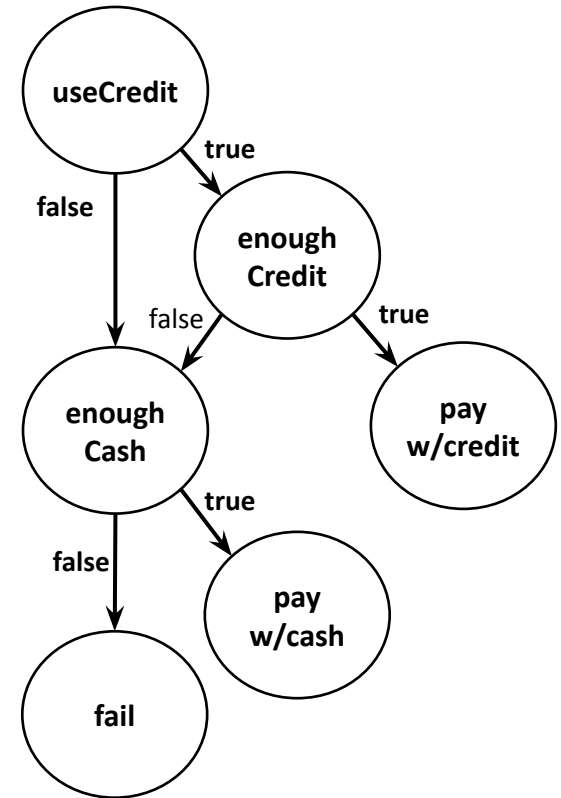
Control-Flow of CreditCard.pay()

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

S3D

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()

```java
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?

S3D

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

S3D

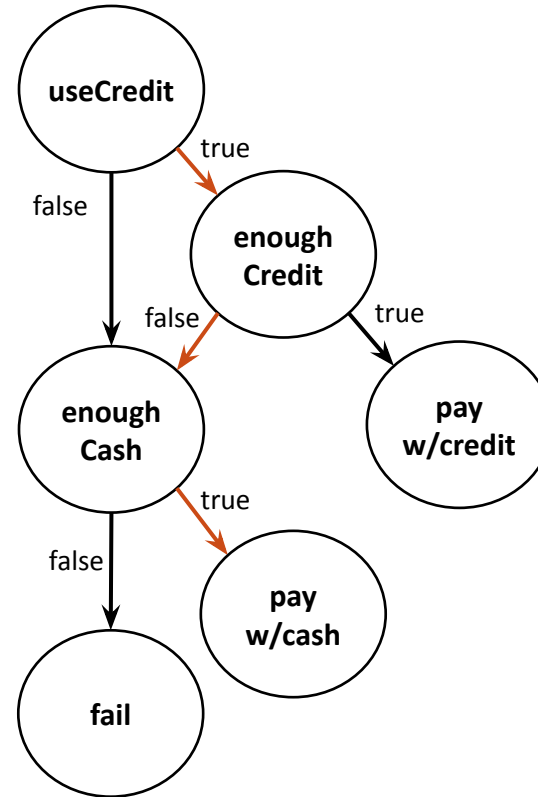# Control-Flow of CreditCard.pay()

Paths:

- {true, true}: pay w/credit
- {false, true}: pay w/cash
- {false, false}: fail
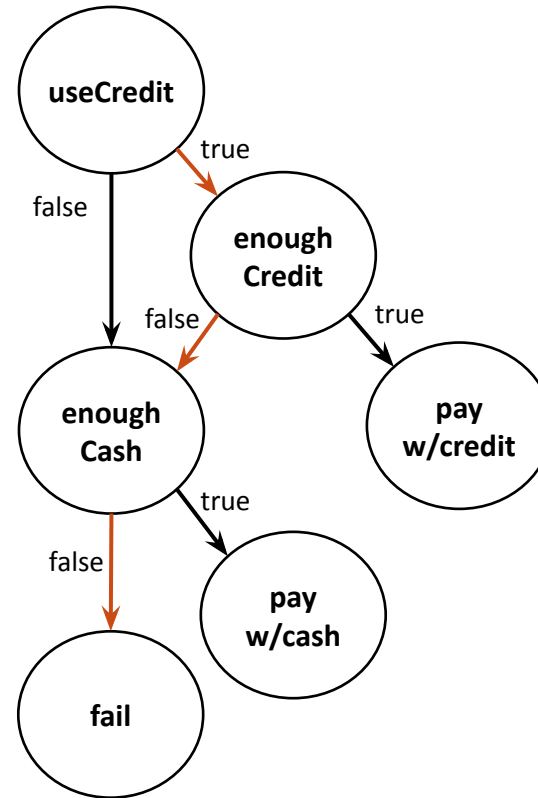
S3D

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

S3D
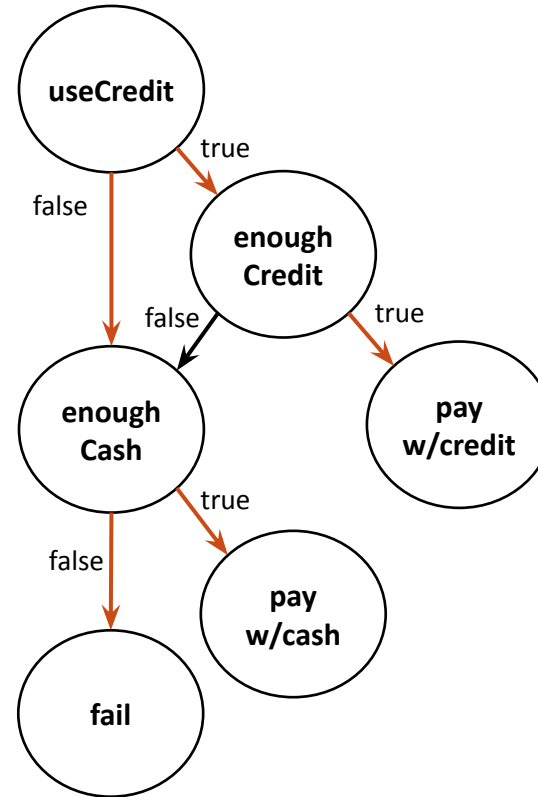
Control-Flow of CreditCard.pay()

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()

```java
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()

```java
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()));
}
```
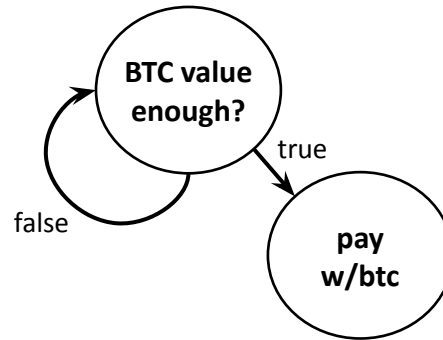
Control-flow of BitCoinWallet.pay()

What are all the paths?
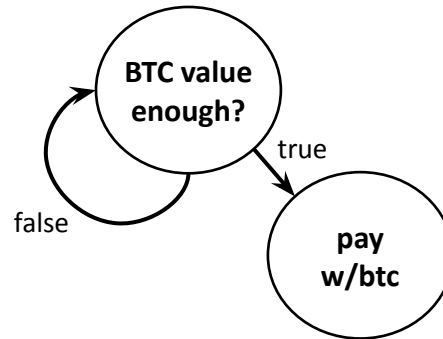
Control-flow of BitCoinWallet.pay()

What are all the paths?

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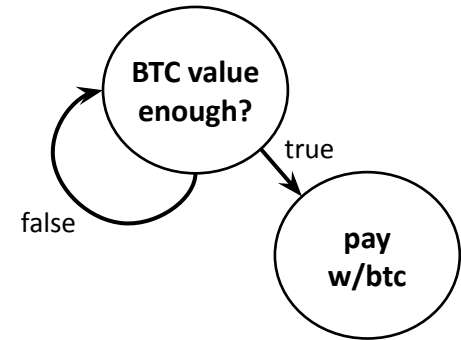- {false, false, false, true}
- ...

S3D

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
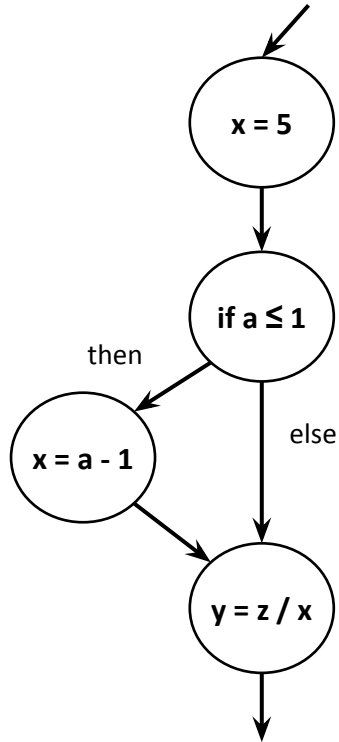

BTC value enough?

true

false

pay w/btc

More Coverage

Many more criteria exist:

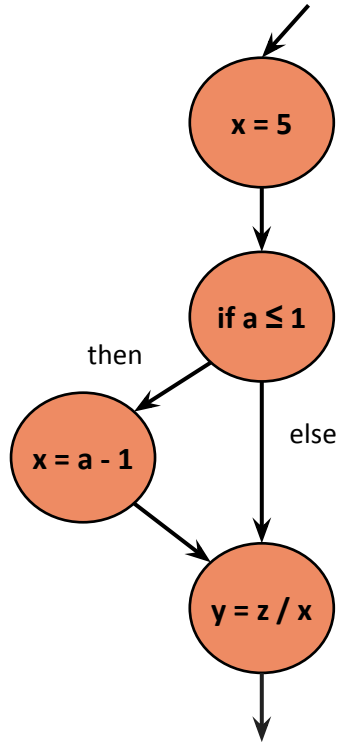- For branches with multiple conditions
  - Modified Condition/Decision Coverage is quite popular
- For loops
  - Boundary Interior Testing
- Branch coverage is by far the most common

Coverage and Quality

x = 5

if a ≤ 1

then

x = a - 1

else

y = z / x

Question 1: Is there a defect?
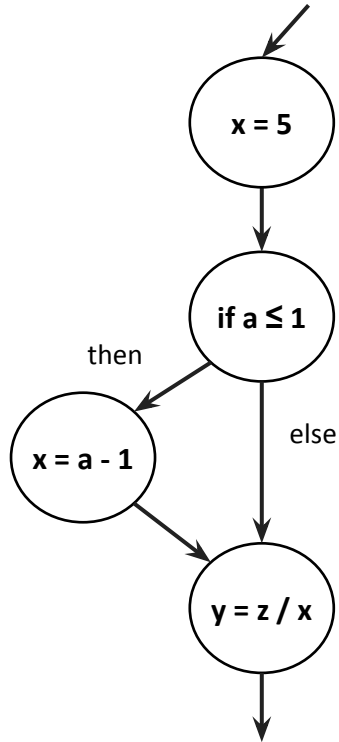
S3D

Coverage and Quality

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

Coverage and Quality

x = 5

if a ≤ 1

then

x = a - 1

else

y = z / x

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

S3D

Outline

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

S3D

Writing Testable Code

What is the problem with this?

```java
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
```

S3D

Writing Testable Code

What is the problem with this?

```java
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 without having a test input file:
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();
}
```

S3D

Writing Testable Code

Exercise: rewrite to make this easier

- And: what would you test?

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

S3D

Writing Testable Code

Aim to write easily testable code

- Which is almost by definition more modular

```java
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
```

S3D

Writing Testable Code

What is the problem with this?

```java
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;
    }
}
```

S3D

Writing Testable Code

Split functionality into easily testable **units**

```java
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];
    }
}
```

S3D

Clean Testing

What is the problem with this?

```java
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);
    }
}
```

S3D

Clean Testing

Keep tests simple, small

```java
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
...
```

S3D

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

S3D

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 {
              }
            }
          }
        }
      }
    }
  }
}
```

S3D

Outline

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

S3D

Back to Specification Testing

What would you test differently in this situation?

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

```java
/** 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);
```

S3D

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 as few as three test cases

```java
/** 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);
```

S3D

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

S3D

Boundary Value Testing

We need a *strategy* to identify plausible mistakes

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

```java
/** 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`

```java
/** 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;
}
```

S3D

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`

```java
/** 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)

```java
/** 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}

```java
/** 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);
```

S3D

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

S3D

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 <u>useful</u> branch coverage

Further Testing Strategies

Many more aspects, some later in this course:

- Stubbing/Mocking, to avoid testing dependencies
  - We'll loop back to this
- Integration testing: scenarios that span units
  - With unit testing one should not test for an expected <u>usage</u> scenario
    - e.g., in HW2: that everything gets called from Main
  - This lets one make some simplifying assumptions
    - e.g., that every card is seen equally often
- Beyond correctness: performance, security

Summary

Testing comprehensively is hard

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