

<https://xkcd.com/1700/>

Unit Testing and Test-Driven Development

CSCI 2134: Software Development

Agenda

- Lecture Content
 - Creating unit tests
 - Basis testing
 - Path testing
 - Requirements testing
 - Test Driven Development
- Brightspace Quiz
- Readings:
 - This Lecture: Chapter 22
 - Next Lecture: Chapter 22

Creating Unit Tests

- A unit tests is targeted at small pieces of code written by a single programmer:
 - Methods
 - Classes
- Unit test are used to ensure that:
 - **Each method works:** given the parameters, and a known state of the object on which the method is called, the method does what is expected
 - **The class works:** a sequence of operations on an object of the class does what is expected



Examples of Functionality Tested by Unit Tests

Methods

- Example 1: *Queue.size()*
The method returns the size of the queue
- Example 2: *Matrix.clone()*
The method returns a copy of the matrix
- Note: Sometimes it's hard to distinguish between method unit tests and class unit tests (that's ok)

Classes

- Example 1: A *Queue* class
 - A sequence of items is dequeued in the same order they were enqueued.
 - Number of items in the queue is equal to number of items enqueued minus number of items dequeued
- Example 2: A *Matrix* class
 - Operations performed on a Matrix object are not lost

Unit Test Deliverables



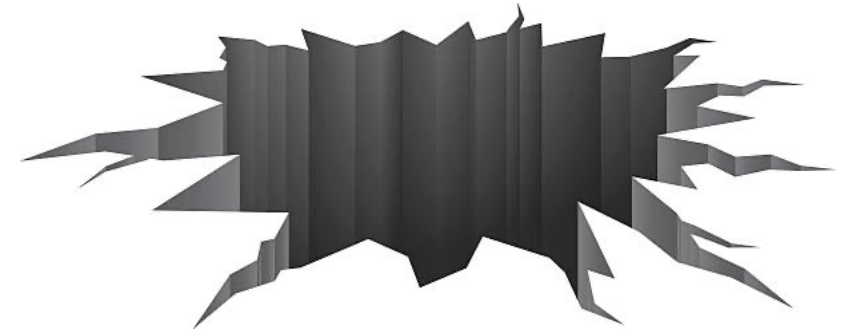
- What are you being asked to do?
 - Create tests for each method that you write
 - Create tests for each class that you write
- Questions:
 - How many tests do we need to create?
 - How long should the tests be?
 - How exhaustive should the tests be?
 - Should they be large tests or small tests?
 - How do we know when we have enough tests?
- All these questions ask the same thing: **What makes a good test suite?**

What makes a good test suite?

Good test suites:

- Good (high) coverage
 - Tests all (most) parts of the code we are testing
 - Types of Coverage
 - Functions / methods: Every method gets called by a test
 - Statements: Every statement gets executed by a test
 - Branches / Loop (if / switch / for / while statements): Every branch gets taken
 - Path coverage: Every path through the code is performed
- Smallest number of tests necessary
 - Minimize the overlap
 - Tests take time to write, run, and verify
- Smaller tests are easier to understand and debug
- Include all boundary conditions and exceptional cases
- Can be hard to write

Testing Pitfalls



- Create “clean” tests
 - Wrong mindset: developers want to verify that code works instead of thinking how to break the code
- Optimistic view of coverage
 - Common to assume 95% coverage, when the true figure is between 30% - 80%)
- Oversimplification
 - Assume the statement coverage is sufficient
 - Do not do component or integration testing because such tests are harder to design and implement

Nice example: https://miro.medium.com/max/480/1*4-T6VVnULaszi9ydHQ7Sfw.gif

Need a structured approach to avoiding these pitfalls

A Structured Approach to Writing Unit Tests

Task: Enumerate and identify all parts of our code we want to test

- White Box Approaches

- Control-flow

- Structured basis testing (exercise each statement)
 - Path testing (exercise each path)

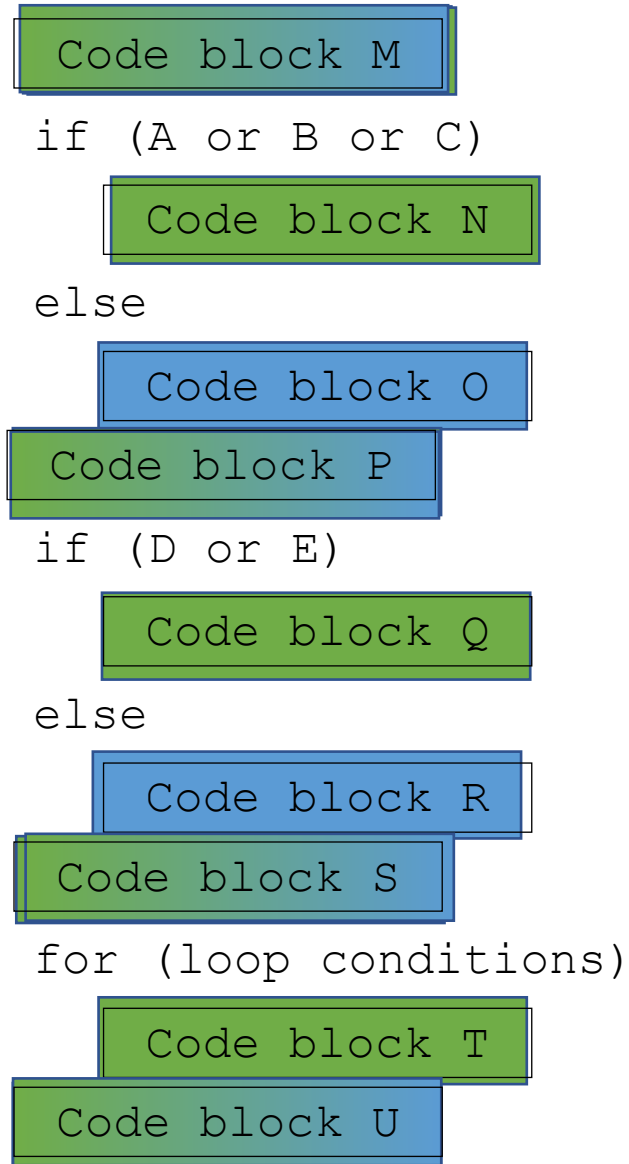
- Data-flow

- Black Box Approaches

- Requirements testing



How many tests do we need? Naive

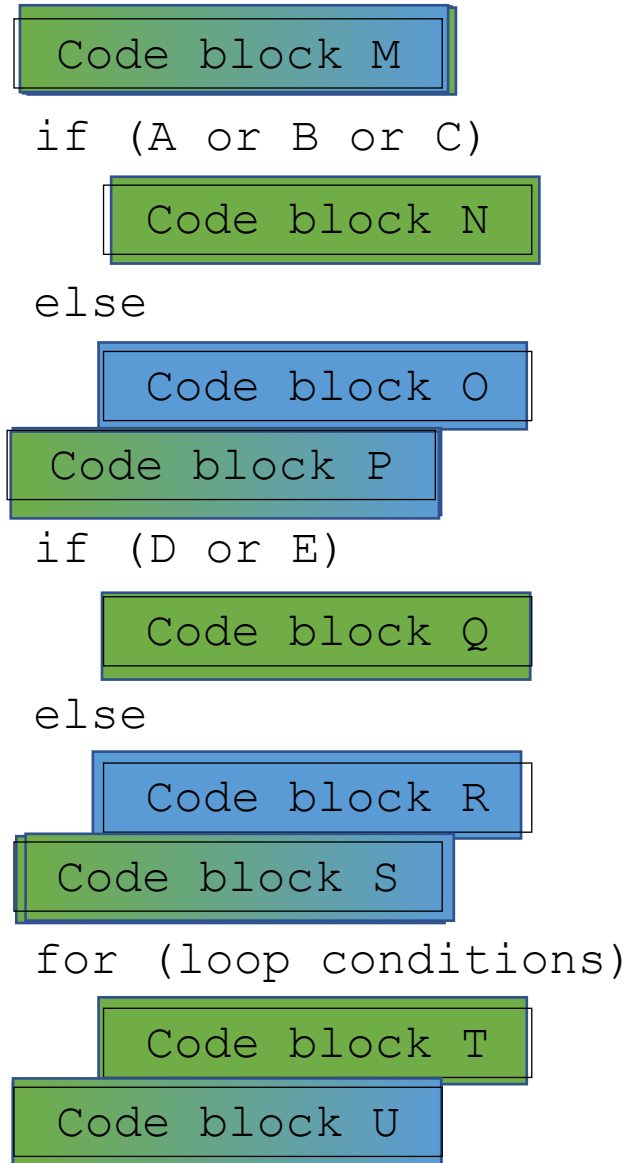


- First test case code:
 - code block M
 - code block N
 - code block P
 - code block Q
 - code block S
 - code block T
 - code block U
- Second test case code:
 - code block M
 - code block O
 - code block P
 - code block R
 - code block S
 - code block U
- Two test cases exercise every statement of code
- Assume Code blocks have no branches or loops
- Doesn't test extra conditions

Structured Basis Testing

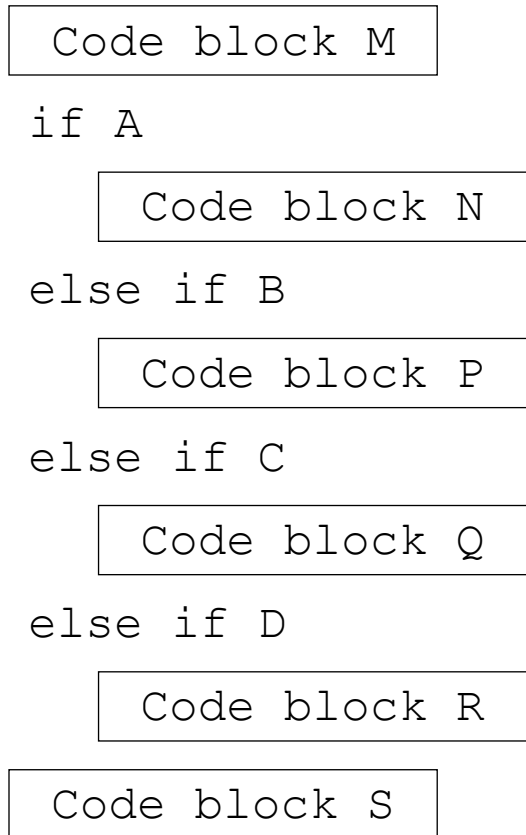
- Idea:
 - For each method create tests that test each part of the code in the method
 - Each part of the code needs to be tested once
- Algorithm
 - Start with one test case at start of method
 - Add a test case for each
 - Loop
 - If statement
 - Part of a Boolean condition
 - Every code branch
- Observation: Smaller methods need fewer test cases

Structured Basis Testing: How many tests do we need?



- One:
 - code block M
 - code block O
 - code block P
 - code block R
 - code block S
 - code block U
- Two:
 - code block M
 - code block A->N
 - code block P
 - code block R
 - code block S
 - code block U
- Three:
 - code block M
 - code block B->N
 - code block P
 - code block R
 - code block S
 - code block U
- Four:
 - code block M
 - code block C->N
 - code block P
 - code block R
 - code block S
 - code block U
- Five:
 - code block M
 - code block O
 - code block P
 - code block D->Q
 - code block S
 - code block U
- Six:
 - code block M
 - code block O
 - code block P
 - code block E->Q
 - code block S
 - code block U
- Seven:
 - code block M
 - code block O
 - code block P
 - code block R
 - code block T
 - code block U

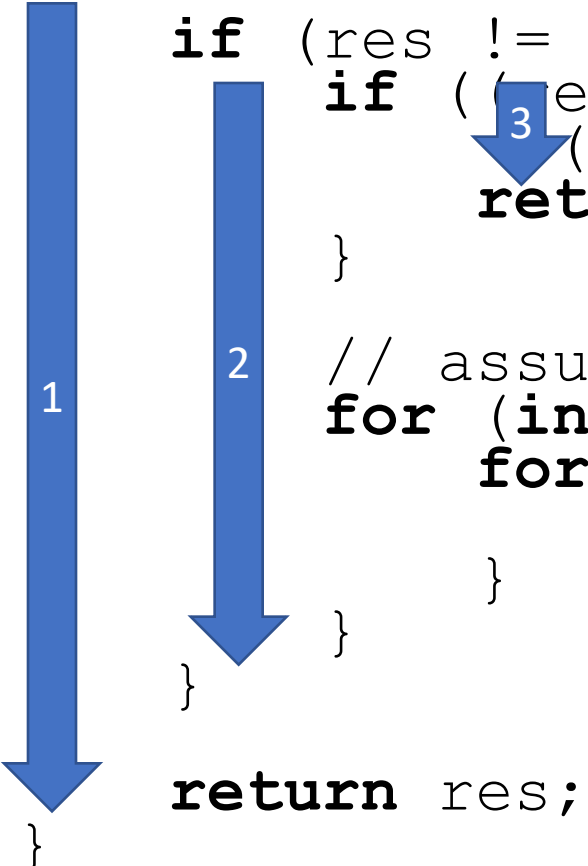
How about now?



- Test Case 1
 - Code Block M
 - **Code Block N**
 - Code Block S
- Test Case 2
 - Code Block M
 - **Code Block P**
 - Code Block S
- Test Case 3
 - Code Block M
 - **Code Block Q**
 - Code Block S
- Test Case 4
 - Code Block M
 - **Code Block R**
 - Code Block S
- Test Case 5
 - Code Block M
 - Code Block S

Example: How many tests do we need?

```
public Matrix multiplyWithScalar(double s, Matrix res) {  
    if (res != null) {  
        if (res.getHeight() != height) ||  
            (res.getWidth() != width) {  
            return null;  
        }  
        // assume height, width > 0  
        for (int i = 0; i < height; i++) {  
            for (int j = 0; j < width; j++) {  
                res.matrix[i][j] = s * matrix[i][j];  
            }  
        }  
    }  
    return res;  
}
```



Test Cases:

1. `res == null`; expected return: null
2. `res.height == height, res.width == width`, expected return `s*this`
3. `res != null, res.height != height`, expected return null
4. `res.height == height, res.width != width`, expected return null

Path Testing

- **Idea:** Instead of just ensuring that each statement is executed, we want to ensure that all execution paths are taken.

- **Why?**

- Same statement may do different things depending on the path!

- **Example:**

```
if (temp < 0) {  
    day = null;  
}  
if (time > 60) {  
    day.getCoffee();  
}
```

- Depending on the structured basis test, some bugs can be missed!

- How to avoid? Check all four cases

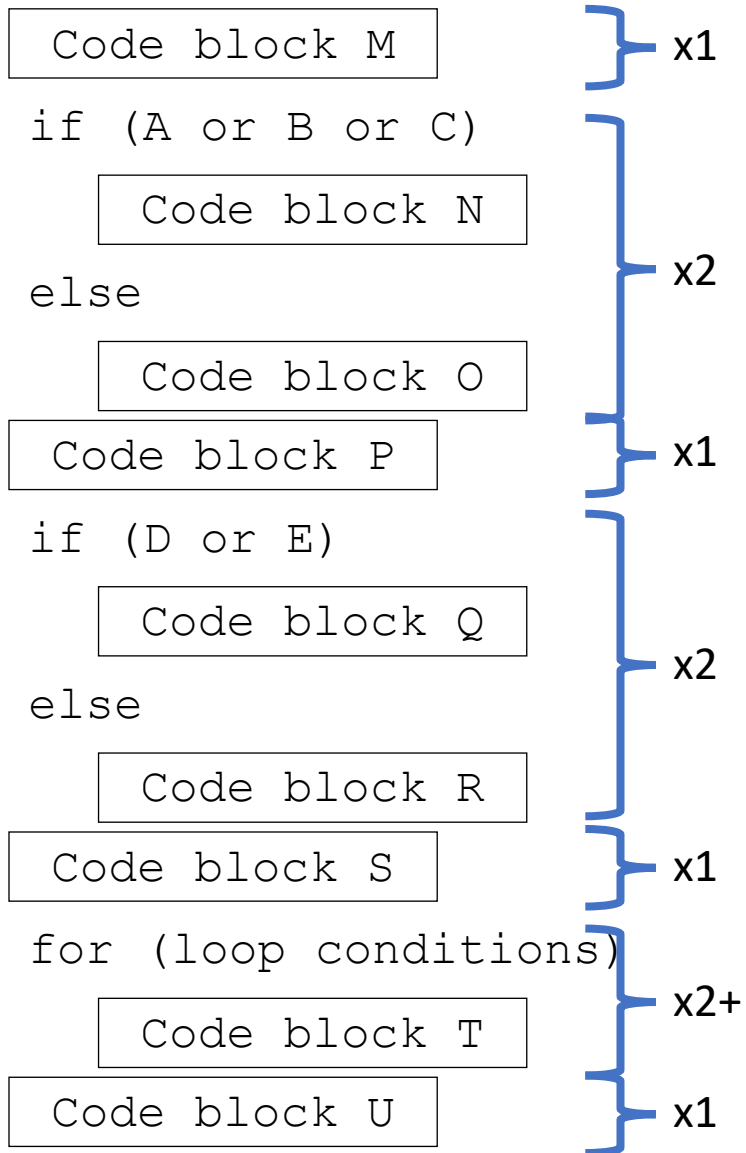
- temp < 0, time <= 60
 - temp >= 0, time <= 60
 - temp < 0, time > 60
 - temp >= 0, time > 60

- **Key Idea:**

- In **structured basis testing** we **sum** the number of test cases
 - In **path testing** we **multiply** the number of test cases

Will this statement work?

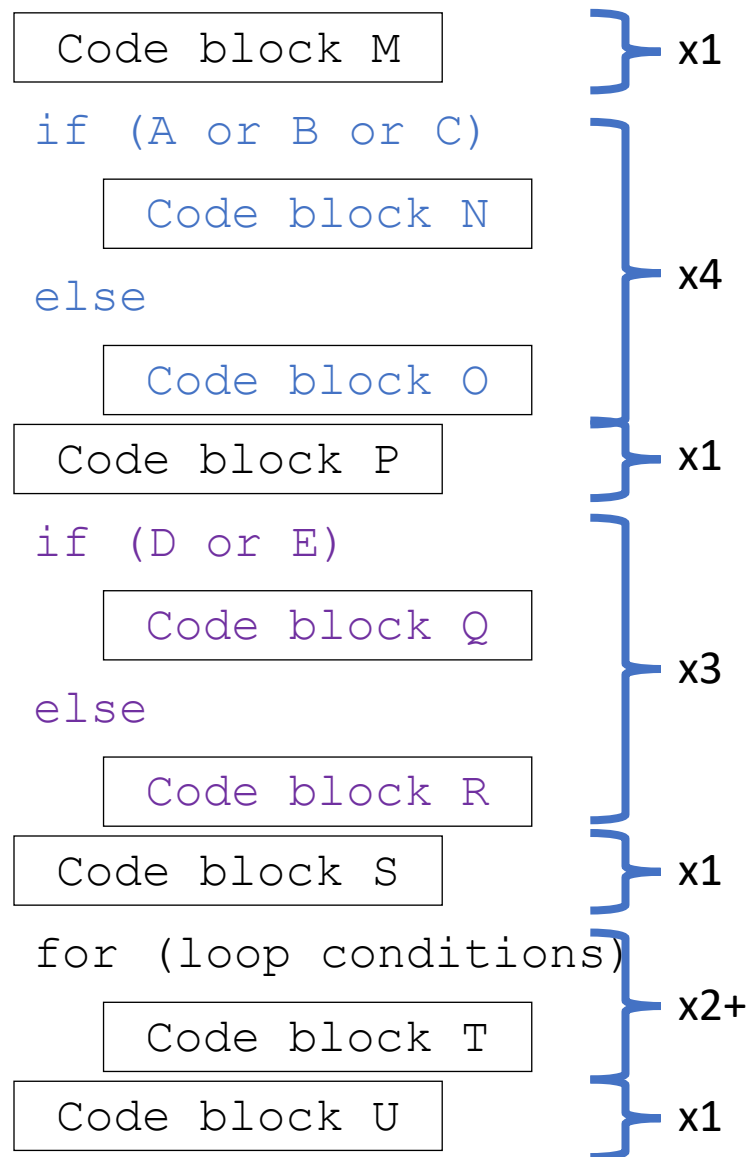
How many tests do we need for path testing of just code blocks?



• Paths

- M, N, P, Q, S, U
- M, N, P, Q, S, T, U
- M, N, P, R, S, U
- M, N, P, R, S, T, U
- M, O, P, Q, S, U
- M, O, P, Q, S, T, U
- M, O, P, R, S, U
- M, O, P, R, S, T, U
- Total # of paths = $1 \times 2 \times 1 \times 2 \times 2(+)\times 1 = 8(+)$
- Why 2+ for the loop?
Depends on whether the number of iterations matters

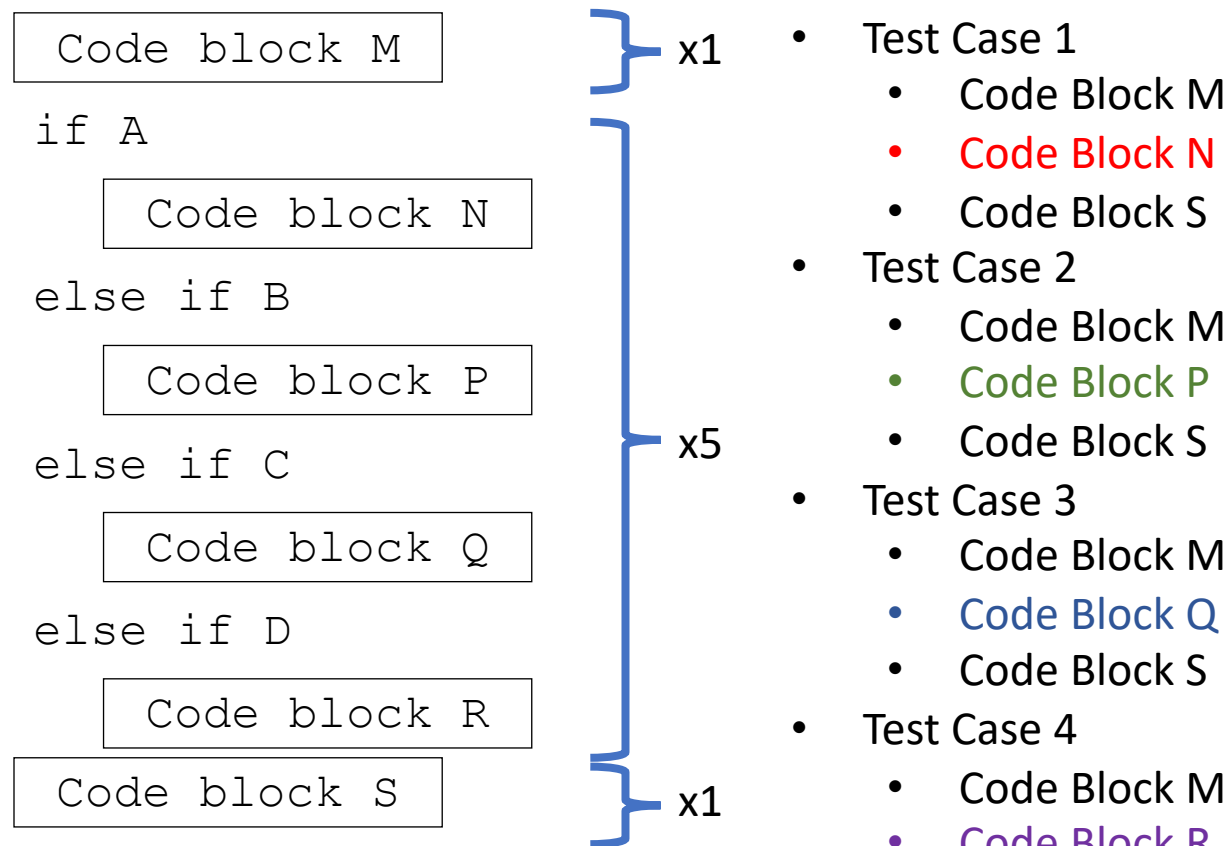
How many tests do we need for path testing?



• Paths

- M, Nx3, P, Qx2, S, U
- M, Nx3, P, Qx2, S, T, U
- M, Nx3, P, R, S, U
- M, Nx3, P, R, S, T, U
- M, O, P, Qx2, S, U
- M, O, P, Qx2, S, T, U
- M, O, P, R, S, U
- M, O, P, R, S, T, U
- Total # of paths = $1 \times 4 \times 1 \times 3 \times 2(+)$ $\times 1 = 24(+)$
- Why 2+ for the loop?
Depends on whether the number of iterations matters

How about this for path testing?



- Test Case 1
 - Code Block M
 - **Code Block N**
 - Code Block S
- Test Case 2
 - Code Block M
 - **Code Block P**
 - Code Block S
- Test Case 3
 - Code Block M
 - **Code Block Q**
 - Code Block S
- Test Case 4
 - Code Block M
 - **Code Block R**
 - Code Block S
- Test Case 5
 - Code Block M
 - Code Block S

Q: Why 5 and not 4?

A: There is an implicit **else**

Path and Condition Testing

- **Idea:** We want to ensure that all conditions are exercised as well.
- **Why?**
 - Same path may do different things depending on the condition!

- **Example:**

```
if (temp < 0) {  
    day = null;  
}  
if (time > 60 || day != null) {  
    day.getCoffee();  
}
```

- Even more cases
 - temp < 0, time <= 60, day != null
 - temp >= 0, time <= 60, day != null
 - temp < 0, time > 60, day != null
 - temp >= 0, time > 60, day != null
 - temp < 0, time <= 60, day == null
 - temp >= 0, time <= 60, day == null
 - temp < 0, time > 60, day == null
 - temp >= 0, time > 60, day == null
- **Key Idea:** Each condition is considered a separate path

Motivation for Requirements/Specification Testing

Tends to be preferred... why?

- Can be used with Test Driven Development
- Avoids implementation-dependent biases
- Supports standard design principles (SOLID)
- Forces the developer to understand the requirements
- Same tests can be reused for different implementations



Blackbox: Requirements/Specification Testing



- **Idea:** Use the specification to determine
 - Error / Exceptional cases
 - Boundary cases
 - Common cases
- Create tests for these cases
- Specifications typically take the form of:
 - Expected inputs or parameters
Typically described using @param tag
 - Processing
Typically described using @description tag
 - Expected outputs
Typically described using the @return tag
- Using this information to derive test cases

```
/**
 * @description: Multiply the
 * current matrix with matrix b
 * and store the results in
 * res. If matrices are not
 * allocated or correctly sized,
 * return null.
 * @param: Matrix b, matrix to be
 * multiplied with the
 * current matrix
 * @param; Matrix res, destination
 * matrix where the result
 * is placed.
 * @return: returns res matrix
 * for programming convenience.
 */
```

Creating Test Cases Using Requirements or Specification

Process: Create 1+ test cases for

- Each error condition
 - E.g., returns null or throws exception
 - Typically based on parameters and return values
 - Ignore undefined behavior
- Each boundary condition
 - E.g., minimum, maximum, and threshold values
 - Typically based on method description
- Common / typical cases
 - Can be very subjective
 - Anything that is not a error or boundary

Example: Postage Problem

```

/* @description: This method takes a
 * destination country (Canada or US)
 * and the weight of a standard envelope
 * and returns the needed postage. If
 * the weight is not positive or the
 * country is invalid, -1 is returned.
 *
 * @param: int weight, in grams
 * @param: String country, either "US"
 *         or "CAN"
 * @return: int postage in cents or -1
 *         if params are invalid
 */

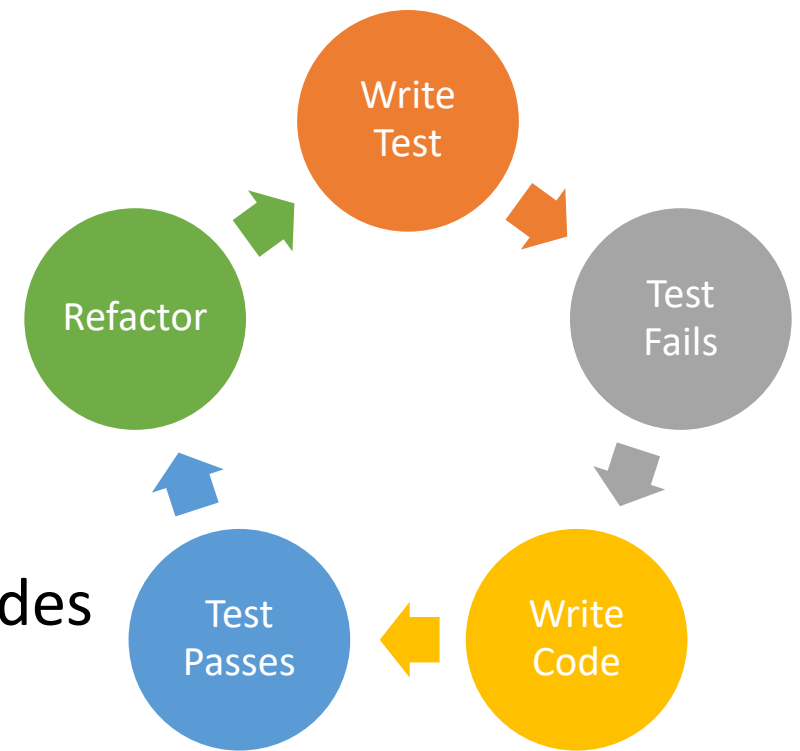
```

Weight	Canada	US
Up to 30g	\$0.85	\$1.20
Over 30g and up to 50g	\$1.20	\$1.80
Up to 100g	\$1.80	\$2.95
Over 100g and up to 200g	\$2.95	\$5.15
Over 200g and up to 300g	\$4.10	\$10.30
Over 300g and up to 400g	\$4.70	\$10.30
Over 400g and up to 500g	\$5.05	\$10.30

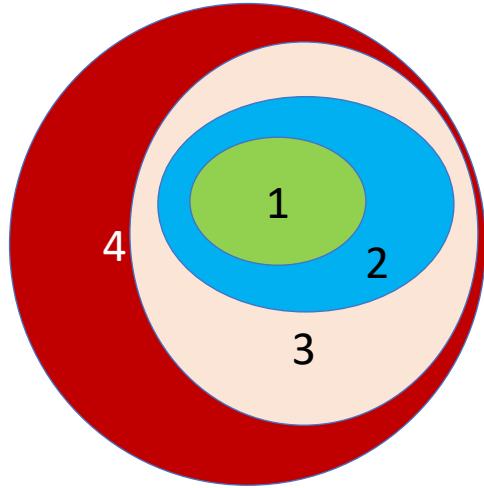
- Error cases:
 - weight < 1 (OR < 0)
 - country != "US" and country != "CAN"
 - country == null (**is this an error case?**)
- Boundary conditions
 - weight == 1, country == "US" or "CAN"
 - weight == 30, country == "US" or "CAN"
 - weight == 31, country == "US" or "CAN"
 - weight == 50, country == "US" or "CAN"
 - weight == 51, country == "US" or "CAN"
 - ... **total of 1 + 7 x 2 x 2 = 29 cases**
- Common cases:
 - One in each category for US and for CAN (7 x 2 = 14 cases)

Test Driven Development (TDD)

- Test-driven development places testing before, during, and after coding
Kent Beck, *“Test-Driven Development: By Example”*, (2003).
- An incremental approach to writing code that provides for continual testing
- Idea:
 - Create requirements-based tests prior to implementation
 - Test as you implement
 - Do not proceed until all tests pass
 - Repeat
- Fundamentally, the tests you create drive the implementation instead of the other way around



Select-Write-Test-Commit-Repeat



- Select test set 1
Write code to cover test set 1
Commit code
- Select test set 2
Write code to cover test set 1, 2
Commit code
- Select test set 3
Write code to cover test set 1, 2, 3
Commit code
- Select test set 4
Write code to cover test set 1, 2, 3, 4
Commit code

TDD Process



- While some requirement remains incomplete
 - Select a requirement (or portion of requirement) to code
 - Write test cases for the requirement
 - Include the test cases in the automated testing
 - Write just enough code to get the test cases to pass
 - Test and debug the new code
 - Do regression testing
 - Do not proceed until all tests are passing
- Note: JUnit is ideal for doing this

Characteristics of TDD

- Write the minimal amount of code to pass the tests
 - Assumes that you have a good set of test cases for the requirement
 - Ensures that you aren't spending time on code that is not meeting a requirement
- Tries to keep you from overdesigning solutions

Overdesign often comes from thinking ahead of all the functionality that you might want to do sometime in the future and so try to code up now.
- The iterative element may have you re-do some code later on when earlier design choices impede your work

That's ok and is expected.
- Code added for a test may expand on functionality from before

May need to update or fix code for tests that worked before and that now fail.

TDD Example using *Matrix* class

- **Test Set 1**

- `Matrix(int m, int n)`
- `int getHeight()`
- `int getWidth()`

- **Test Set 2**

- `getElem(int i, int j)`
- `setElem(int i, int j, double v)`

- **Test Set 3**

- `Matrix(Matrix mtx)`
- `boolean equals(Matrix m)`

- **Test Set 4**

- `Matrix(Scanner s)`
- `void write(PrintStream out)`

```
public class Matrix {  
    Matrix(int m, int n)  
    Matrix(Matrix mtx)  
    Matrix(Scanner s)  
  
    boolean equals(Matrix a)  
  
    void negate()  
    Matrix add(Matrix b, Matrix res)  
    Matrix multiplyWithScalar(double s, Matrix res)  
    Matrix multiplyWithMatrix(Matrix b, Matrix res)  
  
    void write(PrintStream out)  
  
    double getElem(int i, int j)  
    void setElem(int i, int j, double v)  
    int getHeight()  
    int getWidth()  
}
```

TDD Example using *Matrix* class (cont)

- Test Set 5

- void negate()
- Matrix
multiplyWithScalar(double s,
Matrix res)

- Test Set 6

- Matrix add(Matrix b, Matrix res)

- Test Set 7

- Matrix
multiplyWithMatrix(Matrix b,
Matrix res)

- Q: How do we know what to test?

- A: The specification for each method

```
public class Matrix {  
    Matrix(int m, int n)  
    Matrix(Matrix mtx)  
    Matrix(Scanner s)  
  
    boolean equals(Matrix a)  
  
    void negate()  
    Matrix add(Matrix b, Matrix res)  
    Matrix multiplyWithScalar(double s, Matrix res)  
    Matrix multiplyWithMatrix(Matrix b, Matrix res)  
  
    void write(PrintStream out)  
  
    double getElem(int i, int j)  
    void setElem(int i, int j, double v)  
    int getHeight()  
    int getWidth()  
}
```

TDD Advantages and Disadvantages

Advantages

- Always have some set of working code
- Minimizes the areas to look for to find bugs
 - Should be in the code recently added
- Finds defects early
 - Coding and testing are interleaved
 - Reduces costs of development
- Plans for code coverage
- Always have test cases around for regression testing

Disadvantages

- Difficult to do for a code base that has already started development
 - Legacy code
- Can be carried to too much of an extreme
- Need to maintain the unit tests
- Some need to redesign or refactor code

Key Points

- Unit tests focus on small pieces code (methods and classes) written by the programmer
- Unit tests should high coverage, low overlap, small, focused, and include all boundary conditions
- Structured approaches to unit tests includes basis testing, path testing, and requirements testing
- Test-driven development (TDD) is a process that integrates testing into the implementation phase and facilitates continuous testing
- In TDD tests are developed based on requirements and then the code is written to pass the tests

Image References

Retrieved January 8, 2020

- <http://pengetouristboard.co.uk/vote-best-takeaway-se20/>
- <https://cdn.iconscout.com/icon/premium/png-512-thumb/output-file-1127198.png>
- <https://media.istockphoto.com/vectors/hole-vector-id472384703?k=6&m=472384703&s=612x612&w=0&h=xpsOpX89QEGXfOi hX4em0RoXBmls1X86 b68zDHoul8=>
- <https://pbs.twimg.com/media/EAzW4gGXkAUFnhp.png>
- <https://encrypted-tbn0.gstatic.com/images?q=tbn:ANd9GcQzCrCKJP3PbvrPTwpspoKgkNCwRr9PZbT7tE1FTj-PKteSL9Kb&s>
- https://miro.medium.com/max/800/1*wUsUWiM0o5H-CVNnnD-QUg.png

Retrieved September 30, 2020

- <http://oppressive-silence.com/comic/oh-no-the-robots>