

Test Driven Development, Unit Testing in Software Development

CSc3350

Dr. William Greg Johnson
Department of Computer Science
Georgia State University

Test Driven Development

- Test-driven development (TDD) is an approach to program development in which your inter-leaving testing and code development.
- Tests are written (for failure) before code and ‘passing’ the tests is the critical driver of development.
- You develop code incrementally, along with a test for that increment. You don’t move on to the next increment until the code that you have developed passes its test.
- TDD was introduced as part of agile methods such as Extreme Programming. However, it can also be used in plan-driven development processes.

TDD process activities

- Start by identifying the increment of functionality that is required. This should normally be small and coded in a few lines.
- Write a test for this functionality and implement this as an automated test.
- Run the test, along with all other tests that have been implemented. Initially, you have not implemented the functionality so the new test will fail.
- Implement the functionality and re-run the test.
- Once all tests run successfully, you move on to implementing the next chunk of functionality.

Benefits of TDD

- Code coverage
 - Every code segment that you write has at least one associated test, so all code written has at least one test.
- Regression testing
 - A regression test suite is developed incrementally as a program is developed.
- Simplified debugging
 - When a test fails, it should be obvious where the problem lies. The newly written code needs to be checked and modified.
- System documentation
 - The tests themselves are a form of documentation that describe what the code should be doing.

TDD

Presentation from:

CS 3250 Software

Testing

Author: Dr. Upsorn Praphamontripong,
Associate Professor, Teaching Track
University of Virginia, Engineering

<https://engineering.virginia.edu/faculty/upsorn-praphamontripong>

Intro to TDD

What is TDD?

- Software development process that relies on the repetition of a very short development cycle
- Not a software testing technique; makes use of software testing technique in software development process

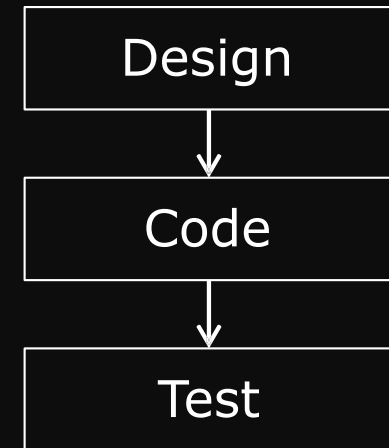
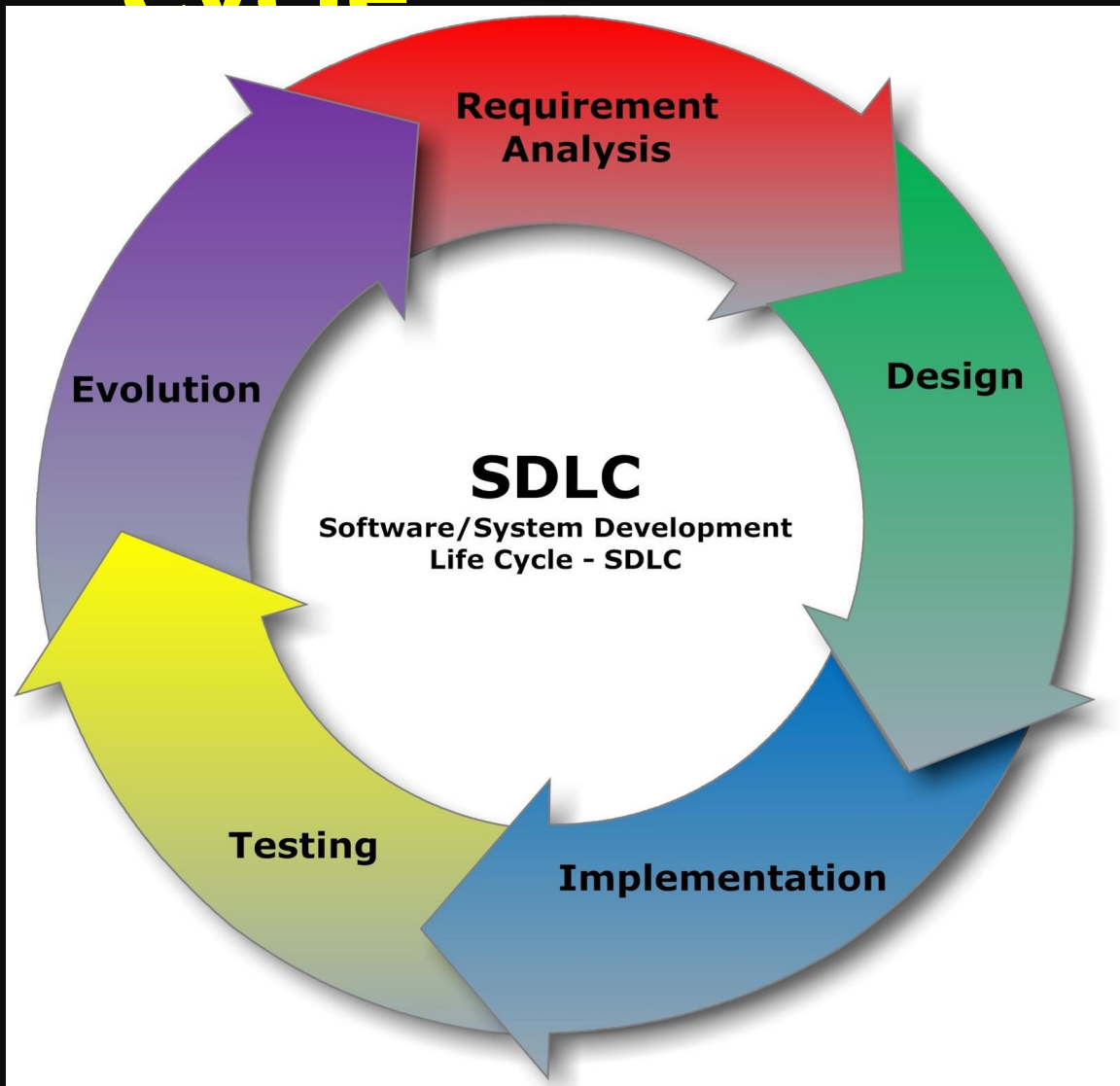
Why TDD?

- Prerequisite for many other practices (e.g., continuous delivery)
- Support better design, well-written code, faster time-to-market, up-to-date documentation, solid test coverage

Drawback

- Requires time and a lot of practice

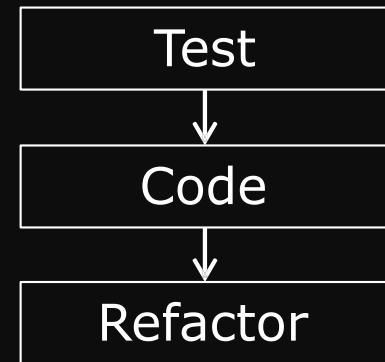
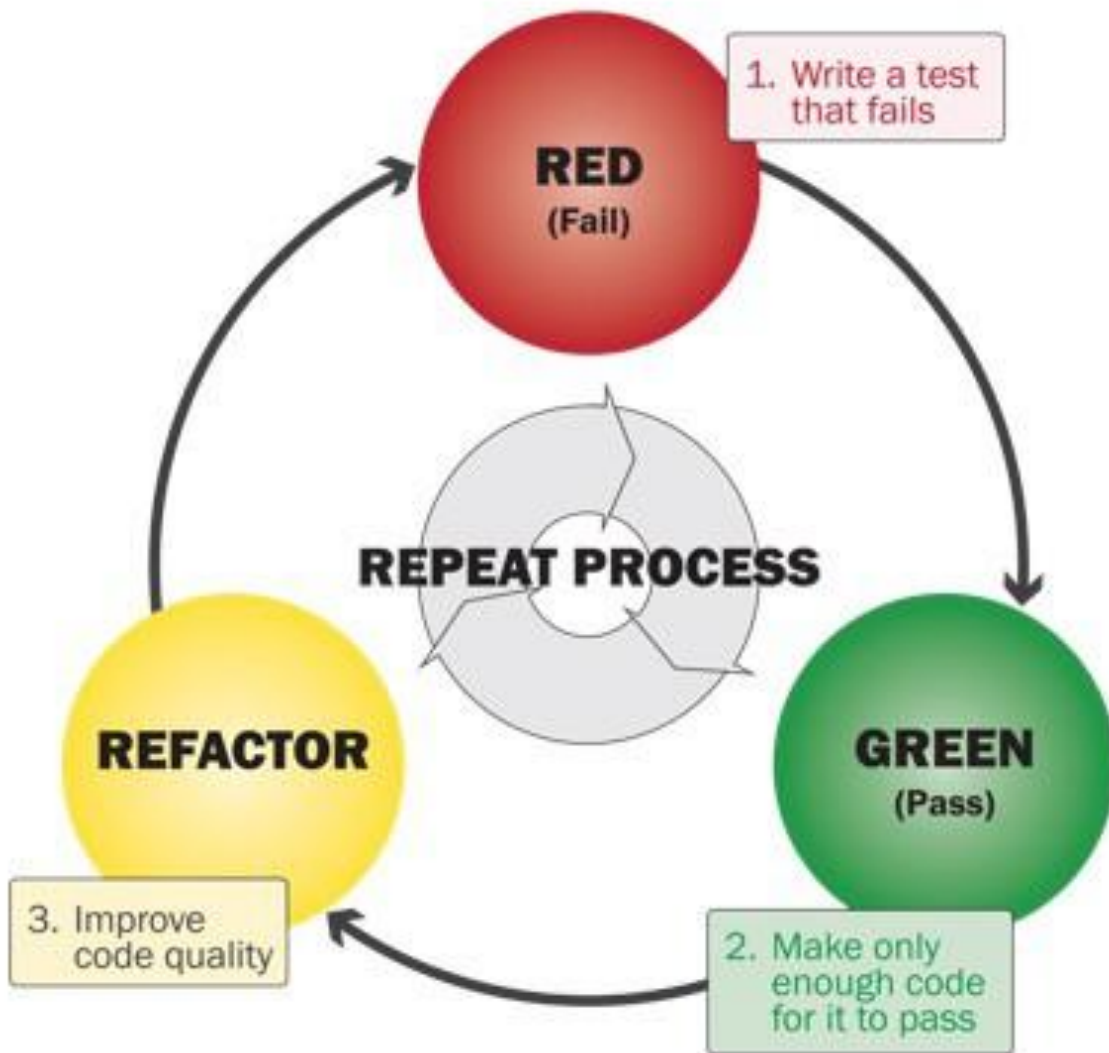
Traditional Development Cycle



Big design up front
Design – not evolution

[image by Cliffydaw - Own work, CC BY-SA 3.0,
<https://commons.wikimedia.org/w/index.php?curid=19054763>

TDD: Red-Green-Refactor Process



Write **tests** before the actual **implementation**

Only write "**just enough**" code to fix a failing test
Deliver "**for now**" items

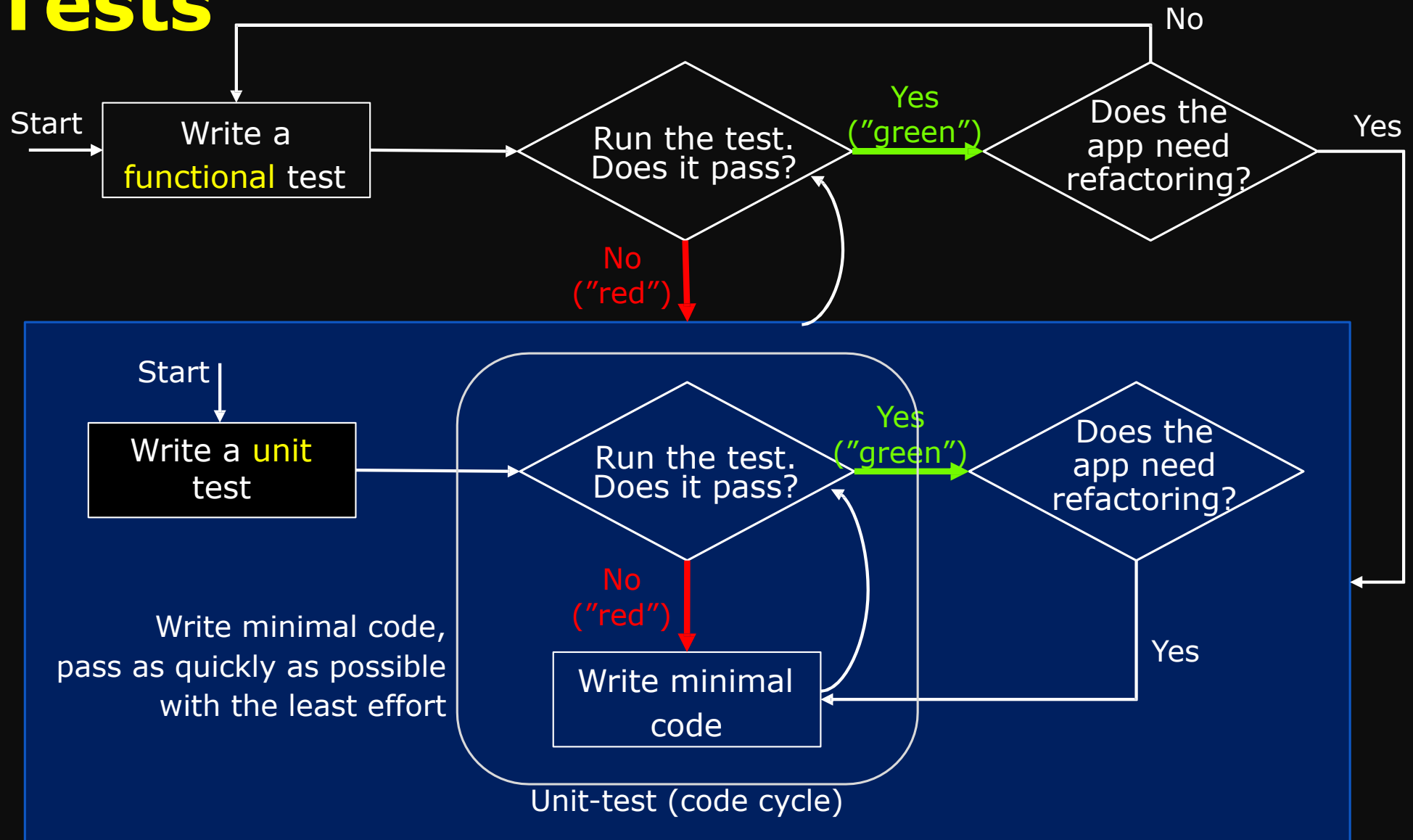
Speed is key

Not big design up front

Design – **evolve** based on feedback from real usage

[Image from <https://realpython.com/django-1-6-test-driven-development/>]

TDD with Functional and Unit Tests



[Based on Percival, "Test-Driven Development with Python", Figure

Overview of Process

1. From user story to requirements to tests
2. Choosing the first test
3. Breadth-first, depth-first
4. Let's not forget to refactor
5. Adding a bit of error handling
6. Loose ends on the test list
7. Repeat

Test first – make it run – make it better

Example: Requirements

- Imagine we are implementing a subsystem for the corporate email application.
- This subsystem is responsible for providing **mail-template** functionality so that the CEO's assistant can send all sorts of important, personalized emails to all personnel with a couple of mouse-clicks.
- How would tests drive the development of this subsystem?

1. From User Story to Requirements to Tests

The first step in TDD is writing a failing test, we need to figure out what desired behavior we'd like to test for

- Decomposing requirements
 - Template system as **tasks** – “things we need to **do**”
 - When completed, lead to satisfying the original requirements
 - Template system as **tests** – “things we need to **verify**”
 - When passing, lead to the requirements being satisfied

Example: Tasks vs. Tests

Imagine you are implementing a subsystem for an email application

Template system as **tasks**

- Write a regular expression for identifying variables from the template
- Implement a template parser that uses the regular expression
- Implement a template engine that provides a public API and uses the template parser internally
- ...

Idea of what we should do, easy to lose sight of the ultimate goal – **not represent progress** of the produced software

Template system as **tests**

- Template without any variables renders as is
- Template with one variable is rendered with the variable replaced with its value
- Template with multiple variables is rendered with the appropriate placeholders replaced by the associated values
- ...

Idea of what should be done – **connect to capabilities** of the produced software

[Koskela, p. 46]

What Are Good Tests Made

Of?

- Tests are generally better than tasks for guiding our work, but does it matter what kind of tests we write?
 - Sure it does!
- Two properties of a good test
 - A good test is **atomic**
 - Keeps things small and focused
 - A good test is **isolated**
 - Doesn't depend on other tests

2. Choosing the First

Test

Restrict focus, do not worry about the whole system

- Before coming up with an initial list of tests, **define** a set of **requirements** for the subsystem under test
- Example requirements:
 - System replaces variable placeholders like `{{firstname}}` with values provided at runtime
 - Attempt to send a template with undefined variables raises errors
 - System ignores variables that are not in the template
- Example corresponding tests:
 - Evaluating template "Hello, `{{name}}`" with value "Reader" "Hello, Reader"
 - Evaluating "`{{greeting}}`, `{{name}}`" with "Hi" and "Reader" result in "Hi, Reader"
 - Evaluating "Hello, `{{name}}`" with "name" undefined raises `MissingValueError`

Requirements



Tests

Writing The First Failing Test

- We got a list of tests that tell us exactly when the requirements have been fulfilled. Now, we start working through the list, making them pass one by one
- Consider the following test
 - Evaluating template "Hello, \${name}" with value "Reader" results in "Hello, Reader"
- Now, let's create a JUnit test

Example

Step 1: Creating a **skeleton** for our tests

```
import static org.junit.jupiter.api.Assertions.*; import
org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach; import
org.junit.jupiter.api.BeforeAll; import
org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class mail_TestTemplate
{
}
```

Note: this example uses Junit 5

Example

Step 2: Adding a test method

```
import static org.junit.jupiter.api.Assertions.*; import
org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach; import
org.junit.jupiter.api.BeforeAll; import
org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class mail_TestTemplate
{
    @Test
    public void oneVariable()
    {

    }
}
```

Example

Step 3: Writing the actual test

```
import static org.junit.jupiter.api.Assertions.*; import
org.junit.jupiter.api.AfterAll;
import org.junit.jupiter.api.AfterEach; import
org.junit.jupiter.api.BeforeAll; import
org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;

public class mail_TestTemplate
{
    @Test
    public void oneVariable()
    {
        mailTemplate template = new mailTemplate("Hello, ${name}");
        template.set("name", "Reader");
        assertEquals("Hello, Reader", template.evaluate());
    }
}
```

Example

Now, the compiler points out that there is no such constructor for `mailTemplate` that takes a `String` as a parameter

Step 4: **Satisfying the compiler** by adding empty methods and constructors

```
public class mailTemplate
{
    public mailTemplate(String templateText)
    {
    }

    public void set(String variable, String value)
    {
    }

    public String evaluate()
    {
        return null;
    }
}
```

Example

Step 5: Running test

- Yes, the test fails – not surprisingly, because we haven't implemented the methods yet
- Benefit: to check that the **test is executed**, not the test result

The red phase of the TDD cycle

What we have now tell us when we are done with this particular task

“when the test passes, the code does what we expect it to do”

Exempl

Step 6: Making the first test pass

- **Passing as quickly as possible and with minimal effort** – it's fine to use a hard-coded return statement at this point

```
public class mailTemplate
{
    public mailTemplate(String templateText)
    {
    }

    public void set(String variable, String value)
    {
    }

    public String evaluate()
    {
        return "Hello, Reader";
    }
}
```

The green phase of the TDD cycle

2 dimensions to move forward:

- Variable
- Template text

Example

Step 7: Writing another test

How to make
the test pass

```
public class mail_TestTemplate
{
    @Test
    public void oneVariable()
    {
        mailTemplate template = new mailTemplate("Hello, ${name}")
        template.set("name", "Reader"); assertEquals("Hello, Reader",
        template.evaluate());
    }

    @Test
    public void differentValue()
    {
        mailTemplate template = new mailTemplate("Hi, ${name}");
        template.set("name", "someone else");
        assertEquals("Hi, someone else", template.evaluate());
    }
}
```

Forcing out the
hard-coded

with another test

The hard-coded
evaluate method
in the
mailTemplate
class will no
longer pass this
test

Example

Step 8: **Revising code** (to make the second test pass by storing and returning the set value)

```
public class mailTemplate
{
    private String variableValue;
    public mailTemplate(String templateText)
    {

    }

    public void set(String variable, String value)
    {
        this.variableValue = value;
    }

    public String evaluate()
    {
        return "Hello, " + variableValue;
    }
}
```

Our test passes again with minimal effort

Our test isn't good enough yet because of the hard-coded part

To improve the test's quality, follow three dimensions to push our code: **variable**, **value**, **template**

Example

Step 9: Revising test

```
public class mail_TestTemplate
{
    @Test
    public void oneVariable()
    {
        mailTemplate template = new mailTemplate("Hello, ${name}")
        template.set("name", "Reader"); assertEquals("Hello, Reader",
        template.evaluate());
    }
}
```

```
@Test
public void differentTemplate() throw Exception
{
    mailTemplate template = new mailTemplate("Hi, ${name}");
    template.set("name", "someone else");
    assertEquals("Hi, someone else", template.evaluate());
}
```

Rename test
to match what
we're doing

Squeeze out
more hard
coding

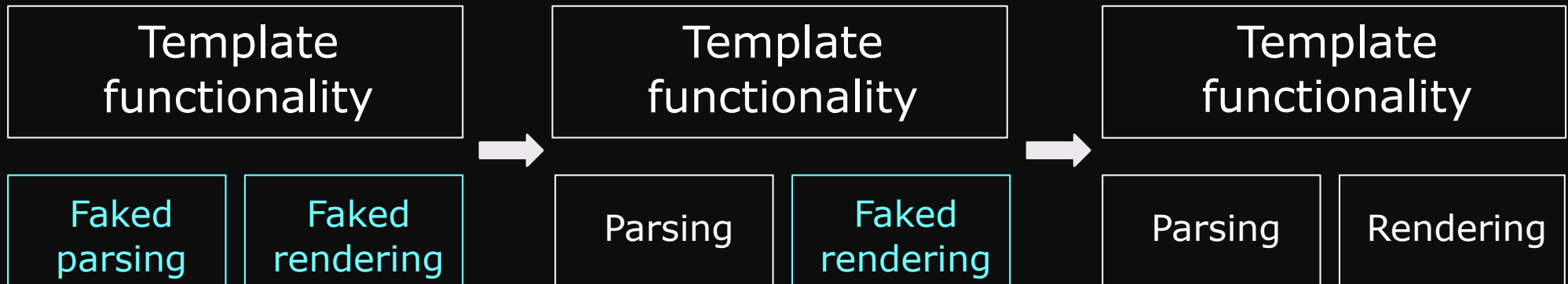
Hard-coded return from the production code won't work anymore

3. Breadth-First, Depth-First

- What to do with a “hard” red phase?
 - Issue is “What to fake” vs. “What to build”
- “Faking” is an accepted part of TDD
 - That is, “deferring a design decision”

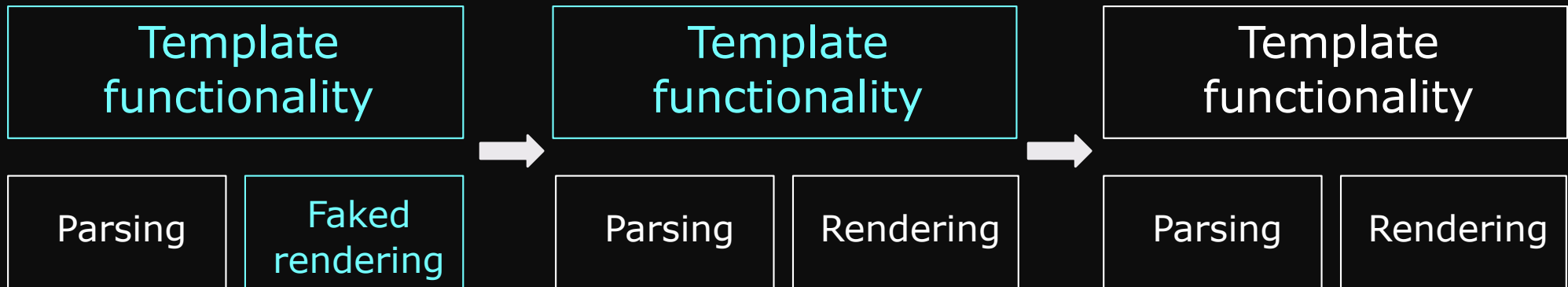
Breadth-First

- Implement the **higher-level** functionality first by faking the required lower-level functionality



Depth-First

- Implement the **lower-level** functionality first and only compose the higher-level functionality once all the ingredients are present



Back to Our Example

- Assume we are dealing with “Hello, $\text{\$}\{name\}$ ”
- We can fake the lower-level functionality
- Do breath-first

Faking Details a Little Longer

Handling variables as variables

```
public class mailTemplate
{
    private String variableValue; private
    String templateText;

    public mailTemplate(String templateText)
    {
        this.templateText = templateText;
    }

    public void set(String variable, String value)
    {
        this.variableValue = value;
    }

    public String evaluate()
    {
        return templateText.replaceAll("\\$\\{name\\}", variableValue);
    }
}
```

Store the variable value and the template text somewhere

Make evaluate() replace the placeholder with the value

Proceed with the TDD Cycle

- Run the tests
- All tests are passing
- Now, add more test to squeeze out the fake stuff

The green phase of the TDD cycle

Squeezing Out The Fake Stuff

Writing test for **multiple variables** on a template

```
@Test
public void multipleVariables() throws Exception
{
    mailTemplate template = new mailTemplate("${one}, ${two}, ${three}"); template.set("one",
    "1");
    template.set("two", "2");
    template.set("three", "3");
    assertEquals("1, 2, 3", template.evaluate());
}
```

The red phase

This test fails

To get the test
passing as quickly
as possible, do the
search-and-replace
implementation


```
import java.util.Map; import
java.util.HashMap; import
java.util.Map.Entry;
```

```
public class mailTemplate
{
```

```
private Map<String, String> variables; private String
templateText;
```

```
public mailTemplate(String templateText)
```

```
{
```

```
this.variables = new HashMap<String, String>();
this.templateText = templateText;
```

```
}
```

```
public void set(String name, String value)
```

```
{
```

```
this.variables.put(name, value);
```

```
}
```

```
public String evaluate()
```

```
{
```

```
String result = templateText;
```

```
for (Entry<String, String> entry : variables.entrySet())
```

```
{
```

```
String regex = "\\$\\{" + entry.getKey() + "\\}"; result =
result.replaceAll(regex, entry.getValue());
```

```
}
```

```
return result;
```

```
} }
```

Solution to Multiple Variables

Store variable
values in
HashMap

Loop through
variables

Replacing
each variable
with its value

Run tests again,
Nothing's
broken!

Special Test Case

Evaluating template "Hello, \${name}" with values "Hi" and "Reader" for variables "doesnotexist" and "name", results in the string "Hello, Reader"

```
@Test
public void unknownVariablesAreIgnored() throws Exception
{
    mailTemplate template = new mailTemplate("Hello, ${name}");
    template.set("doesnotexist", "whatever");
    template.set("name", "Reader");
    assertEquals("Hello, Reader", template.evaluate());
}
```

If we set variables that don't exist in the template text, the variables are ignored by the mailTemplate class

This test passes without any changes to the mailTemplate class

Why Red Then

Green

- We intentionally fail the test at first just to see that
 - Our test execution catches the failure
 - We are really executing the newly added test
 - Then proceed to implement the test and see the bar turn green again

4. Let's Not Forget To Refactor

- Refactor: changing internal structure (of the current code) without changing its external behavior
- At this point, it might seem that we didn't add any code and there is nothing to refactor

Refactoring applies to **code** and **test code**

- Though we didn't add any production code, we added test code, and that is code – just like any other
 - We don't want to let our test code rot and get us into serious trouble later
- What could we do about our test code?
 - Identify any **potential refactoring**
 - **Decide** which of them we'll carry out

Example : Test Class (So Far)

Can you spot
anything to
refactor?

```
public class mail_TestTemplate
{
    @Test
    public void oneVariable()
    {
        mailTemplate template = new mailTemplate("Hello, ${name}");
        template.set("name", "Reader"); assertEquals("Hello, Reader",
        template.evaluate());
    }

    @Test
    public void differentTemplate() throw Exception
    {
        mailTemplate template = new mailTemplate("Hi, ${name}");
        template.set("name", "someone else");
        assertEquals("Hi, someone else", template.evaluate());
    }

    @Test
    public void multipleVariables() throws Exception
    {
        mailTemplate template = new mailTemplate("${one},${two},${three}"); template.set("one",
        "1");
        template.set("two", "2");
        template.set("three", "3");
        assertEquals("1, 2, 3", template.evaluate());
    }

    @Test
    public void unknownVariablesAreIgnored() throws Exception
    {
        mailTemplate template = new mailTemplate("Hello, ${name}");
        template.set("doesnotexist", "whatever"); template.set("name",
        "Reader"); assertEquals("Hello, Reader", template.evaluate());
    }
}
```

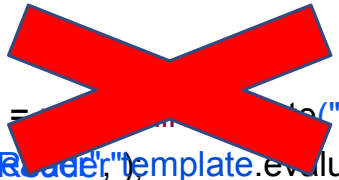
Potential Refactoring in Test Code

- All tests are using a `mailTemplate` object
 - Solution: extract it into an **instance variable** rather than declare it over and over again, use **fixtures**
- The `evaluate()` method is called several times as an argument to `assertEquals`
 - Solution: write a **method** that calls the `evaluate()` method
- The `mailTemplate` class is instantiated with the same template text in two places
 - Solution: remove the **duplicate** by using **fixtures** (with some unified values)

Remove redundant tests

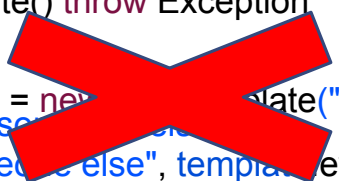
Revisi t Current Test Class

```
public class mail_TestTemplate
{
    @Test
    public void oneVariable()
    {
        mailTemplate template = new mailTemplate("Hello, ${name}");
        assertEquals("Hello, Reader", template.evaluate());
    }
}
```



Let's consider duplication between these tests

```
@Test
public void differentTemplate() throws Exception
{
    mailTemplate template = new mailTemplate("Hi, ${n
template.set("name", "some
assertEquals("Hi, some else", template.evaluate
}
```



multipleVariables() covers
oneVariable() and
differentTemplate()
-- thus, get rid of them

```
@Test
public void multipleVariables() throws Exception
{
    mailTemplate template = new mailTemplate("${one},${two},${three}");
    template.set("one", "1");
    template.set("two", "2");
    template.set("three", "3");
    assertEquals("1, 2, 3", template.evaluate());
}
```

unknownVariablesAreIgnored() can use
the same template text as
multipleVariables()

```
@Test
public void unknownVariablesAreIgnored() throws Exception
{
    mailTemplate template = new mailTemplate("Hello, ${name}");
    template.set("doesnotexist", "whatever"); template.set("name", "Reader");
    assertEquals("Hello, Reader", template.evaluate());
}
```

Refactor d Test Code

```
public class mail_TestTemplate  
{
```

```
    private mailTemplate template;
```

```
    @BeforeEach
```

```
    public void setUp() throws Exception  
    {
```

```
        template = new mailTemplate("${one}, ${two}, ${three}");  
        template.set("one", "1");  
        template.set("two", "2");  
        template.set("three", "3");
```

```
    }
```

```
    @Test
```

```
    public void multipleVariables() throws Exception  
    {
```

```
        assertTemplateEvaluatesTo("1, 2, 3");
```

```
    }
```

```
    @Test
```

```
    public void unknownVariablesAreIgnored() throws Exception  
    {
```

```
        template.set("doesnotexist", "whatever");
```

```
        private void assertTemplateEvaluatesTo(String expected)  
        {
```

```
            assertEquals(expected, template.evaluate());
```

```
        }
```

```
}
```

Common fixture for
all tests

Simple,
focused test

Helper method

Now, let's add more
functionality ... add
more tests

5. Adding a Bit of Error Handling

Add exception test, using try/catch block with fail()

```
@Test
public void missingValueRaisesException() throws Exception
{
    try {
        new mailTemplate("${foo}").evaluate();
        fail("evaluate() should throw an exception if " + "a variable was left
            without a value!");
    } catch (MissingValueException expected) { }
}
```

```
// in mailTemplate class
public class MissingValueException extends RuntimeException
{
    // this is all we need for now
}
```

Adding a Bit of Error Handling (2)

Add exception test, using `Assertions.assertThrows()`

```
@Test
public void missingValueRaisesException() throws Exception
{
    Assertions.assertThrows(RuntimeException.class, () -> { new
        mailTemplate("${foo}").evaluate();
    });
}
```

```
// in mailTemplate class
public class MissingValueException extends RuntimeException
{
    // this is all we need for now
}
```

Except test – either try/catch with `fail()` or

`Assertions.assertThrows()` fails.

That means, we have to somehow check the missing variables.

Let's make the test pass

How to get to the green phase as quickly as possible?

Writing Code To Make The Test Pass

- How do we know inside evaluate, whether some of the variables specified in the template text are without a value?
- Checking for **remaining variables** after the search-and-replace

```
public String evaluate()
{
    String result = templateText;
    for (Entry<String, String> entry : variables.entrySet())
    {
        String regex = "\\$" + entry.getKey() + "\\>"; result =
        result.replaceAll(regex, entry.getValue());
    }

    if (result.matches(".*\\$\\{.+\\}.*")) throw new
        MissingValueException();

    return result;
}
```

Does it look like we left a variable in there?

Refactoring Toward Small Methods

- `evaluate()` is doing too many different things
 - Replacing variables with values, checking for missing values
- **Extracting** the check for missing variables into its own method

```
public String evaluate()
{
    String result = templateText;
    for (Entry<String, String> entry : variables.entrySet())
    {
        String regex = "\\$\\{" + entry.getKey() + "\\}"; result =
        result.replaceAll(regex, entry.getValue());
    }
    checkForMissingValues(result); return
    result;
}

private void checkForMissingValues(String result)
{
    if (result.matches(".*\\$\\{.+\\}.*")) throw new
    MissingValueException();
}
```

Get rid of a whole
if-block from
`evaluate()`

Much better.
Is there still more
to do?

More Refactoring

- `evaluate()` is still doing two things:
 - Replacing variables with values
 - Checking for missing values
- **Extracting method refactoring**
 - To create simple, single, clear purpose methods

Run tests again,
Nothing's broken!

```
public String evaluate()
{
    String result = replaceVariables();
    checkForMissingValues(result); return
    result;
}

private String replaceVariables()
{
    String result = templateText;
    for (Entry<String, String> entry : variables.entrySet())
    {
        String regex = "\\$\\{" + entry.getKey() + "\\}"; result =
        result.replaceAll(regex, entry.getValue());
    }
    return result;
}

private void checkForMissingValues(String result)
{
    if (result.matches(".*\\$\\{.+\\}.*")) throw new
        MissingValueException();
}
```

Adding Diagnostics to Exceptions

```
@Test
public void missingValueRaisesException() throws Exception
{
    try {
        new mailTemplate("${foo}").evaluate(); fail("evaluate() should
        throw an exception if " +
            "a variable was left without a value!");
    } catch (MissingValueException expected) {
        assertEquals("No value for ${foo}", expected.getMessage());
    }
}
```

```
// in mailTemplate class
import java.util.regex.Pattern; import
java.util.regex.Matcher;
...
private void checkForMissingValues(String result)
{
    Matcher m = Pattern.compile(".*\\$\\{.+\\}.*").matcher(result);
    if (m.find())
        throw new MissingValueException("No value for " + m.group());
}

public class MissingValueException extends RuntimeException
{
    public MissingValueException(String msg)
    {
        super(msg);
    }
}
```

6. Loose Ends On The Test

List

Testing for performance

```
public class mail_TestTemplate
{
    // Omitted the setUp() for creating a 100-word template with 20 variables
    // and populating it with approximately 15-character values

    @Test
    public void templateWith100WordsAnd20Variables() throws Exception
    {
        long expected = 200L;
        long time = System.currentTimeMillis();
        template.evaluate();
        time = System.currentTimeMillis() - time;
        assertTrue(time <= expected,
            "Rendering the template took " + time + " ms " + "while the target
            was " + expected + " ms" );
    }
}
```

Test That Dooms Current Implementation

Write test that verifies whether the code's current behavior are correct

```
@Test
public void variablesGetProcessedJustOnce() throws Exception
{
    template.set("one", "${one}");
    template.set("two", "${three}");
    template.set("three", "${two}"); assertTemplateEvaluatesTo("${one},
    ${three}, ${two}");
}
```

Note:

- Most TDD tests focus on “happy paths” and often miss
 - Confused-user paths
 - Creative-user paths
 - Malicious-user paths

Summar

Y

- TDD
 - **Test**: write a test
 - **Code**: write code to make the test pass
 - **Refactor**: find the best possible design for what we have, relying on the existing tests to keep us from breaking things while we're at it
- Encourages good design, produces testable code, and keeps us away from over-engineering our system because of flawed assumptions
- When applying TDD, remember to consider both "**happy paths**" and "**non-happy paths**"

CS 3250 Software Testing

Spring 2023

Author: Dr. Upsorn Praphamontripong,
Associate Professor, Teaching Track
University of Virginia, Engineering

<https://engineering.virginia.edu/faculty/upsorn-praphamontripong>

**Test Driven Development, Unit Testing in
Software Development**

CSc3350
Dr. William Greg Johnson
Department of Computer Science
Georgia State University