What is Test-Driven Development?

- TDD is a design (and testing) approach involving short, rapid iterations of Refactor Unit Test Code. Unit tests are automated, which forces programmer to consider use of a method before implementation of the method.
TDD Example: Requirements

• Ensure that passwords meet the following criteria:
  – Between 6 and 10 characters long
  – Contain at least one digit
  – Contain at least one upper case letter
import static org.junit.Assert.*; import org.junit.Test;

class TestPasswordValidator {
    @Test
    public void testValidLength() {
        PasswordValidator pv = new PasswordValidator();
        assertEquals(true, pv.isValid("Abc123"));
    }
}

Needed for JUnit

This is the teeth of the test

Cannot even run test yet because PasswordValidator doesn’t exist!
import static org.junit.Assert.*;
import org.junit.Test;

public class TestPasswordValidator {
    @Test
    public void testValidLength() {
        PasswordValidator pv = new PasswordValidator();
        assertEquals(true, pv.isValid("Abc123"));
    }
}

Design decisions:
- class name, constructor,
- method name, parameters and return type
public class PasswordValidator {
    public boolean isValid(String password) {
        if (password.length() >= 6 && password.length() <= 10) {
            return true;
        } else {
            return false;
        }
    }
}
import static org.junit.Assert.*;
import org.junit.Test;

public class TestPasswordValidator {
    @Test
    public void testValidLength() {
        PasswordValidator pv = new PasswordValidator();
        assertEquals(true, pv.isValid("Abc123"));
    }
}

Do we really need an instance of PasswordValidator?
import static org.junit.Assert.*;
import org.junit.Test;

public class TestPasswordValidator {
    @Test
    public void testValidLength() {
        assertEquals(true, PasswordValidator.isValid("Abc123");
    }
}
What is Refactoring?

• Changing the *structure* of the code without changing its *behavior*
  – Example refactorings:
    • Rename
    • Extract method/extract interface
    • Inline
    • Pull up/Push down

• Some IDE’s (e.g. Eclipse) include automated refactorings
public class PasswordValidator {
    public static boolean isValid(String password) {
        if (password.length() >= 6 && password.length() <= 10) {
            return true;
        } else {
            return false;
        }
    }
}
public class PasswordValidator {
    public static boolean isValid(String password) {
        if (password.length() >= 6 && password.length() <= 10) {
            return true;
        } else {
            return false;
        }
    }
}
public class PasswordValidator {
    public static boolean isValid(String password) {
        return password.length() >= 6 &&
               password.length() <= 10;
    }
}
TDD Example: Refactoring #1

```java
public class PasswordValidator {
    public static boolean isValid(String password) {
        return password.length() >= 6 &&
                password.length() <= 10;
    }
}
```

“Magic numbers” (i.e. literal constants that are buried in code) can be dangerous
public class PasswordValidator {
    private final static int MIN_PW_LENGTH = 6;
    private final static int MAX_PW_LENGTH = 10;

    public static boolean isValid(String password) {
        return password.length() >= MIN_PW_LENGTH && password.length() <= MAX_PW_LENGTH;
    }
}

Refactoring #2: extract constant
import static org.junit.Assert.*; import org.junit.Test;

public class TestPasswordValidator {
    @Test
    public void testValidLength() {
        assertEquals(true, PasswordValidator.isValid("Abc123"));
    }

    @Test
    public void testTooShort() {
        assertEquals(false, PasswordValidator.isValid("Abc12"));
    }
}

TDD Example: Write another test

No design decisions; just unit testing
public class TestPasswordValidator {

    @Test
    public void testValidLength() {
        assertEquals(true, PasswordValidator.isValid("Abc123"));
    }

    @Test
    public void testTooShort() {
        assertEquals(false, PasswordValidator.isValid("Abc12"));
    }

    @Test
    public void testNoDigit() {
        assertEquals(false, PasswordValidator.isValid("Abcdef"));
    }

    @Test
    public void testNoDigit() {
        assertEquals(false, PasswordValidator.isValid("Abcdef"));
    }

}
public class PasswordValidator {
    private final static int MIN_PW_LENGTH = 6;
    private final static int MAX_PW_LENGTH = 10;

    public static boolean isValid(String password) {
        return password.length() >= MIN_PW_LENGTH &&
                password.length() <= MAX_PW_LENGTH;
    }
}
TDD Example: Make the test pass

```java
import java.util.regex.Pattern;

public class PasswordValidator {
    private final static int MIN_PW_LENGTH = 6;
    private final static int MAX_PW_LENGTH = 10;

    public static boolean isValid(String password) {
        return password.length() >= MIN_PW_LENGTH &&
            password.length() <= MAX_PW_LENGTH &&
            Pattern.matches(".*\p{Digit}.*", password);
    }
}
```

Check for a digit
import java.util.regex.Pattern;

public class PasswordValidator {

    private final static int MIN_PW_LENGTH = 6;
    private final static int MAX_PW_LENGTH = 10;

    public static boolean isValid(String password) {
        return password.length() >= MIN_PW_LENGTH &&
            password.length() <= MAX_PW_LENGTH &&
            Pattern.matches(".*\p{Digit}.*", password);
    }
}

Extract methods for readability
import java.util.regex.Pattern;
public class PasswordValidator {
    private final static int MIN_PW_LENGTH = 6;
    private final static int MAX_PW_LENGTH = 10;
    private static boolean isValidLength(String password) {
        return password.length() >= MIN_PW_LENGTH &&
        password.length() <= MAX_PW_LENGTH;
    }
    private static boolean containsDigit(String password) {
        return Pattern.matches(".*\p{Digit}.*", password);
    }
    public static boolean isValid(String password) {
        return isValidLength(password) &&
        containsDigit(password);
    }
}
TDD Example: Done for now
Test-Driven Development

- Test-driven development (TDD) is the craft of producing automated tests for production code, and using that process to drive design and programming. For every tiny bit of functionality in the production code, you first develop a test that specifies and validates what the code will do. You then produce exactly as much code as will enable that test to pass. Then you refactor (simplify and clarify) both the production code and the test code.

Test-Driven Development

• **Definition**¹
  – Test-driven Development (TDD) is a programming practice that instructs developers to write new code only if an automated test has failed, and to eliminate duplication. The goal of TDD is “clean code that works.”

1. “JUnit in Action” Massol and Husted.

• **The TDD Two-Step**²
  – Write a failing automatic test before writing new code
  – Eliminate duplication

• **The TDD Cycle**²
  – Write a test
  – Make it run
  – Make it right

2. “Test-Driven Development By Example” Beck.
Some Types of Testing

- **Unit Testing**
  - Testing individual units (typically methods)
  - White/Clear-box testing performed by original programmer
- **Integration and Functional Testing**
  - Testing interactions of units and testing use cases
  - TDD focuses here
- **Regression Testing**
  - Testing previously tested components after changes
  - and here
- **Stress/Load/Performance Testing**
  - How many transactions/users/events/… can the system handle?
- **Acceptance Testing**
  - Does the system do what the customer wants?
TDD Misconceptions

• There are many misconceptions about TDD
• They probably stem from the fact that the first word in TDD is “Test”
• TDD is **not about testing**, TDD is about **design**
  – Automated tests are just a nice side effect
TDD Misconception #1

- TDD does not mean “write all the tests, then build a system that passes the tests”
TDD Misconception #2

- TDD does not mean “write some of the tests, then build a system that passes the tests”
TDD Misconception #3

• TDD does not mean “write some of the code, then test it before going on”
TDD Misconception #4

- TDD does not mean “do automated testing”
TDD Misconception #5

• TDD does not mean “do lots of testing”
TDD Misconception #6

• TDD does not mean “the TDD process”
• TDD is a *practice*
  (like pair programming, code reviews, and stand-up meetings)
not a *process*
  (like waterfall, Scrum, XP, TSP)
TDD Clarified

• TDD means “write one test, write code to pass that test, refactor, and repeat”