Capability Maturity Model (CMM)

• CMM is not a software lifecycle model ...
  – Strategy for improving the software development process regardless of the process “model” followed
    • Basic premise: the use of new software methods alone will not improve productivity and quality, because software management is, in part, the cause of problems
    – CMM assists organizations in providing the infrastructure required for achieving a disciplined and mature process ($$)

• Includes
  – technical aspects of software production
  – managerial aspects of software production
Capability Maturity Model (continued)

- Five maturity levels
  - 1. initial – ad hoc process
  - 2. repeatable process – basic project management
  - 3. defined process – process modeling and definition
  - 4. managed process – process measurement
  - 5. optimizing process – process control and dynamic improvement

- To move from one stage to the next, the SEI provides a series of questionnaires and conducts process assessments that highlight current shortcomings.
Software Lifecycles and Software Process

- Software lifecycle basics
- Software lifecycles
  - build-and-fix
  - waterfall
  - rapid prototype
  - incremental and iterative
  - spiral
- Process improvement
  - CMM & ISO9000
Processes vs. Practices

• Software processes provide a framework for how to build a software product
  – E.g. Waterfall, XP, Scrum, TSP

• Software practices are activities used when building software
  – E.g. code reviews, test-driven development, pair programming, daily communication
Phases of a Software Lifecycle

• Standard Phases
  – Requirements Analysis & Specification
  – Design
  – Implementation and Integration
  – Operation and Maintenance
  – Change in Requirements
  – Testing throughout

• Phases promote manageability and provide organization
Build-and-Fix

Build First Version

Modify until Client is satisfied

Operations Mode

Retirement
Waterfall\textsuperscript{1}

Rapid Prototyping

Rapid Prototype
  Verify

Design
  Verify

Implementation
  Test

Operations
  Retirement

Req. Change
For each build: Perform detailed design, implement, test, deliver.

For each build: Perform detailed design, implement, test, deliver.

Increments add new features

- Requirements
  - Verify
  - Arch. Design
    - Verify
    - For each build: Perform detailed design, implement, test, deliver.
  - Design → Implement → Test → Deliver
    - Could be staggered
    - Or consecutive
  - Design → Implement → Test → Deliver
    - Design → Implement → Test → Deliver
  - Design → Implement → Test → Deliver
  - Operations
    - Retirement

CAL POLY
For each build: Perform detailed design, implement. Test. Deliver.

Could be minimized or incorporated into iterations (common with agile approaches)

Operations

Retirement

Iterative

Requirements

Verify

Arch. Design

Verify

Iterations may revisit features
The Spiral Model

(Extremely) Simplified Spiral Model

Risk Assessment
- Requirements
  - Verify

Risk Assessment
- Design
  - Verify

Risk Assessment
- Implementation
  - Test

Req. Change

Operations

Retirement

Add a Risk Analysis step to each phase!

More recently extended into MBASE
(Model-Based Architecting and Software Engineering)
TDD in Software Development Lifecycle

Figure 1. Development flow: (a) traditional test-last and (b) test-driven development/test-first flow.
What is Test-Driven Development?

- TDD is a design (and testing) approach involving short, rapid iterations of
  - Unit tests are automated
  - 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
TDD Example: Write a test

```
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");
    }
}
```

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"));
    }
}

TDD Example: Write a test

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"));
    }
}

Design decision:
static method
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
TDD Example: Refactor the code

```java
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;
    }
}

Refactoring #1: collapse conditional
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.

---

CAL POLY
TDD Example: Refactoring #2

```java
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;
    }
}
```
import static org.junit.Assert.*; import org.junit.Test;

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

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

No design decisions; just unit testing

TDD Example: Write another test
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"));
    }
}

TDD Example: Write another test
TDD Example: Make the test pass

```java
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

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);
    }
}
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

- **Regression Testing**
  - Testing previously tested components after changes

- **Stress/Load/Performance Testing**
  - How many transactions/users/events/… can the system handle?

- **Acceptance Testing**
  - Does the system do what the customer wants?

TDD focuses here

and may help here

and here
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”
Test Bus Discussion

1. What is a test bus?
   - Built-in test access

Figure 1. A testable system includes a test bus that can access the API independent of the UI.