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

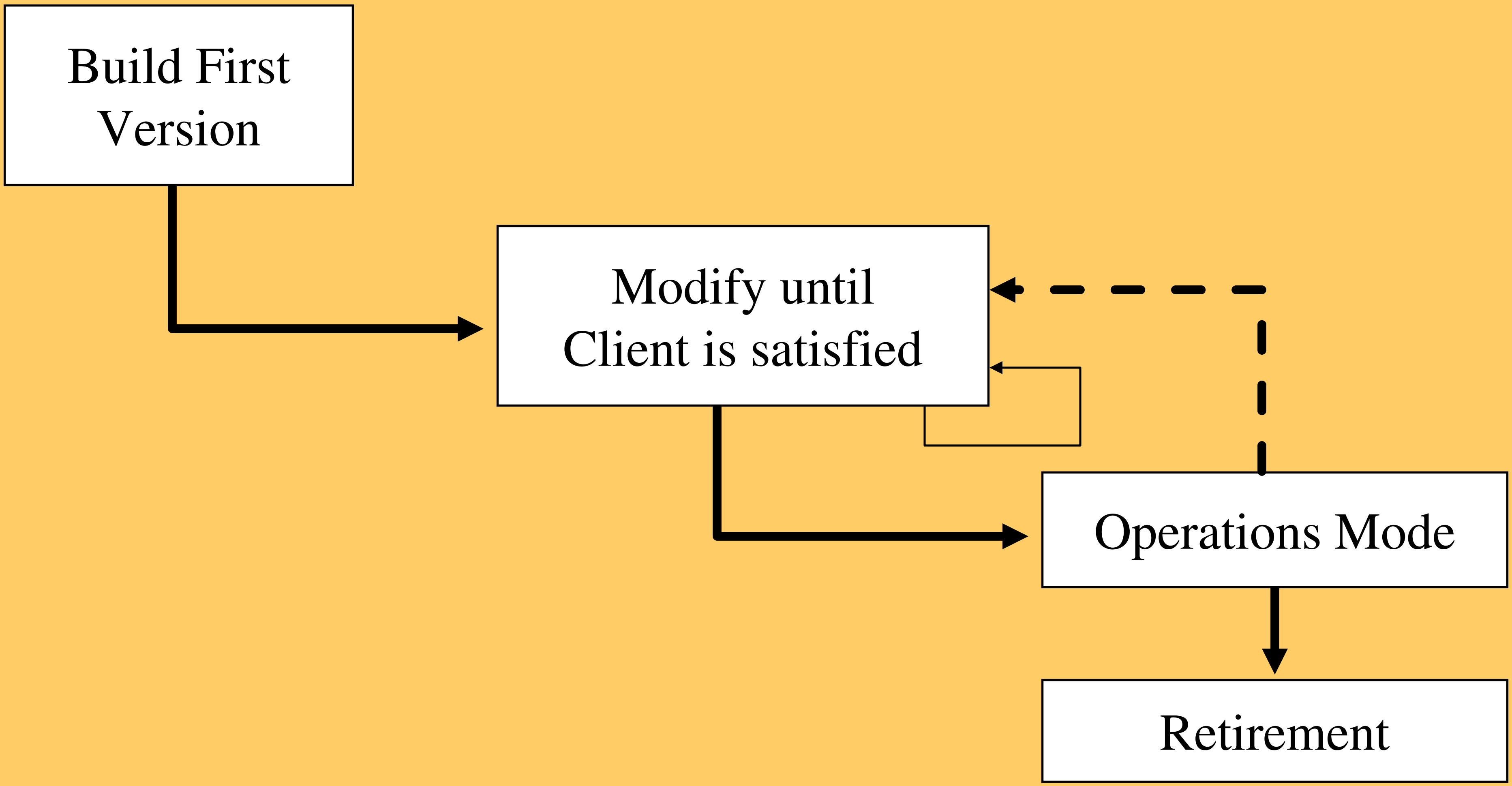
Processes vs. Practices

- 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

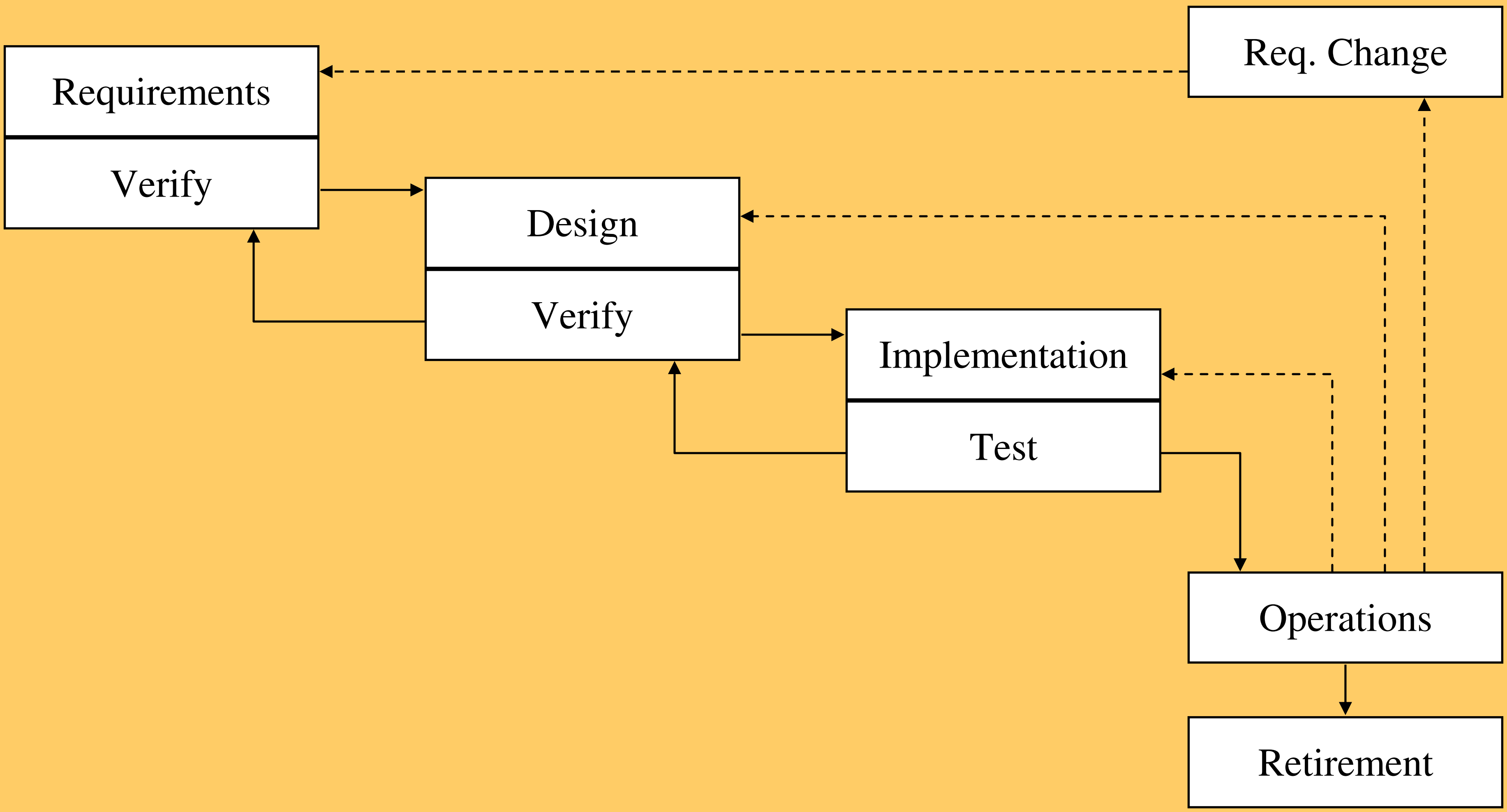
Phases of a Software Lifecycle

- 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

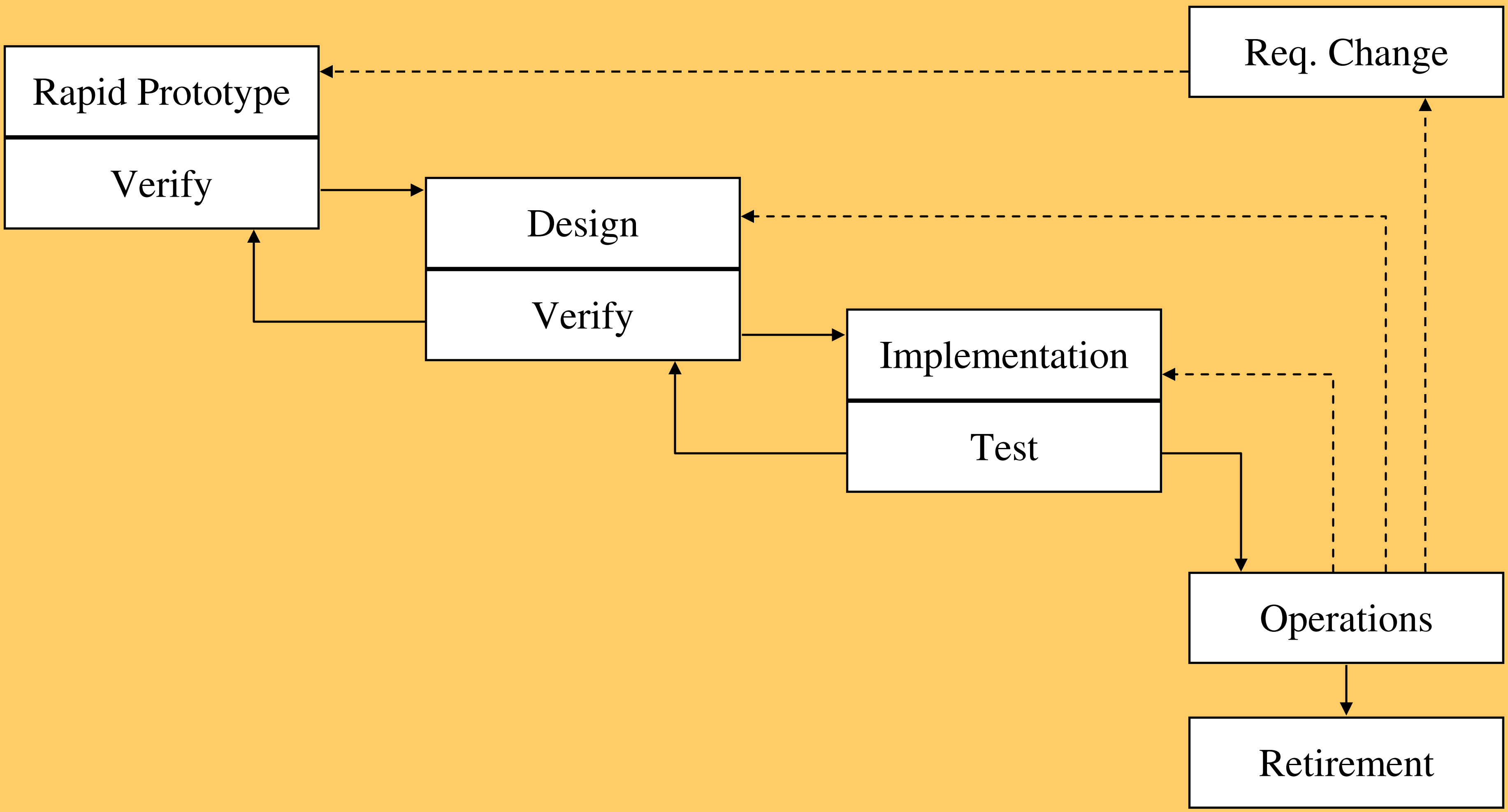


Waterfall[1]

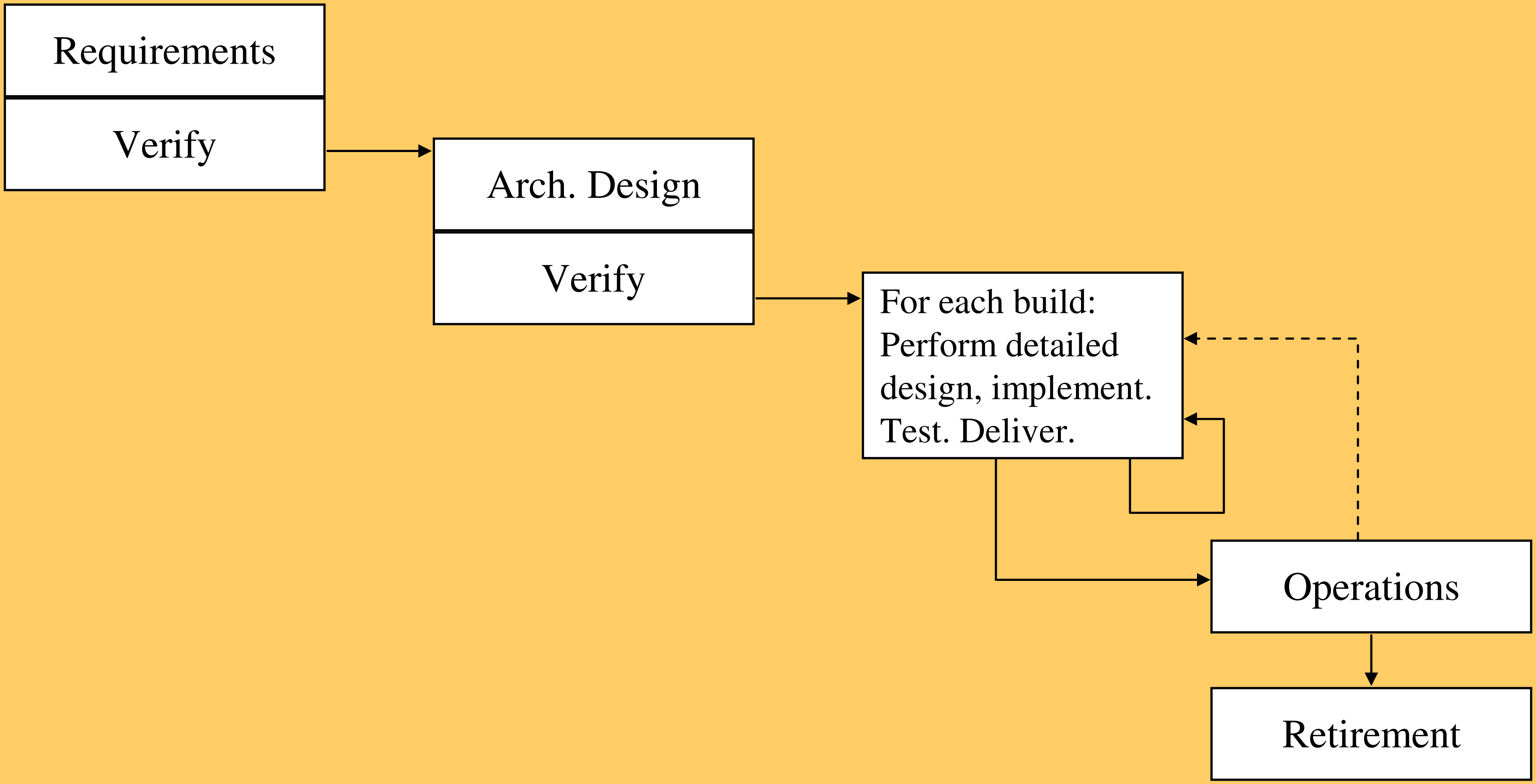


1. Winston Royce, "Modeling the Development of Large Software Systems", Westcon 1970

Rapid Prototyping



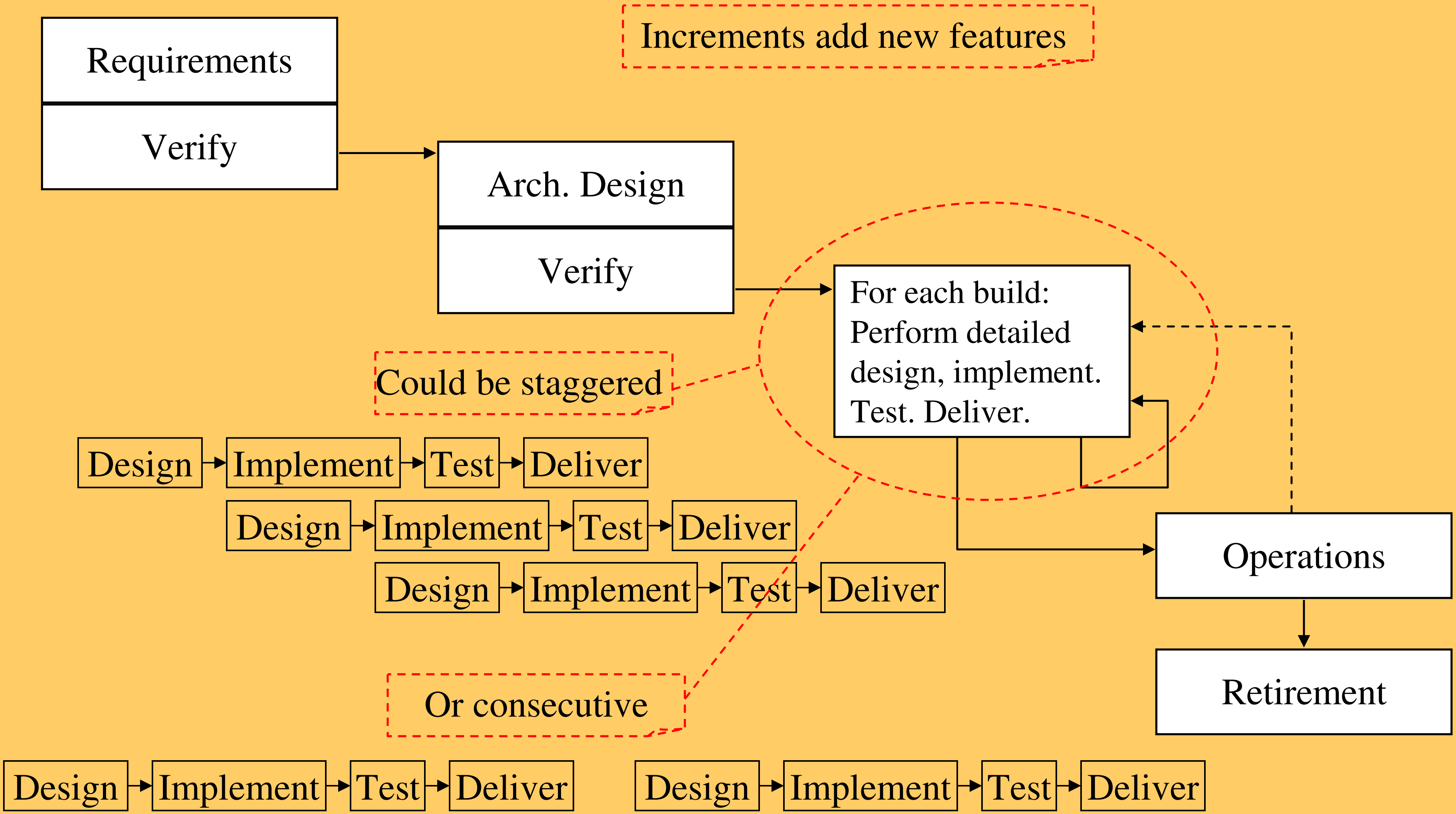
Incremental[1] and Iterative[2]



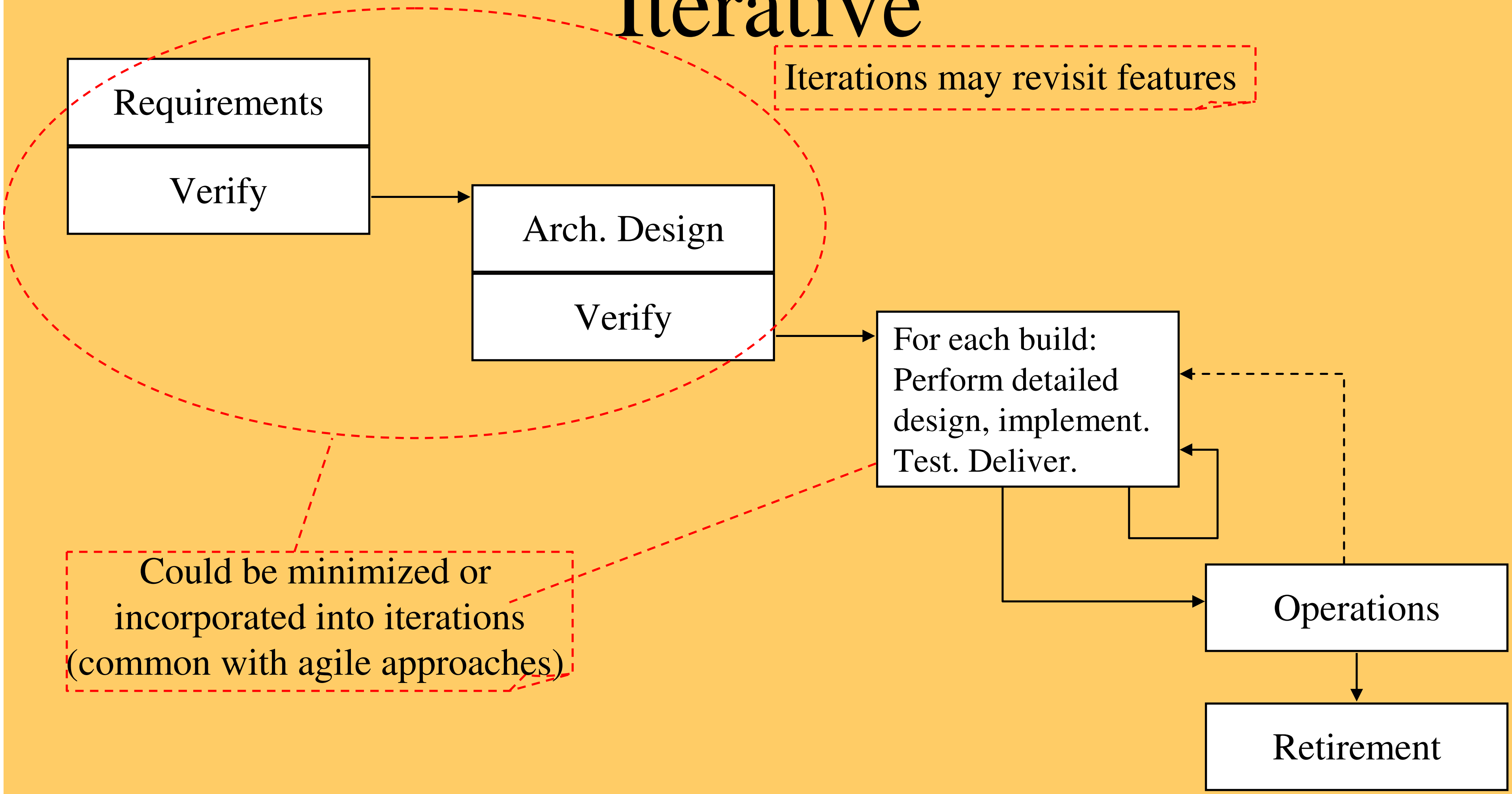
1. Harlan Mills, "Cleanroom Software Engineering", IEEE Software, 1987

2. Victor Basili and Joe Turner, "Iterative Enhancement", IEEE Trans. Software Eng., 1975

Incremental

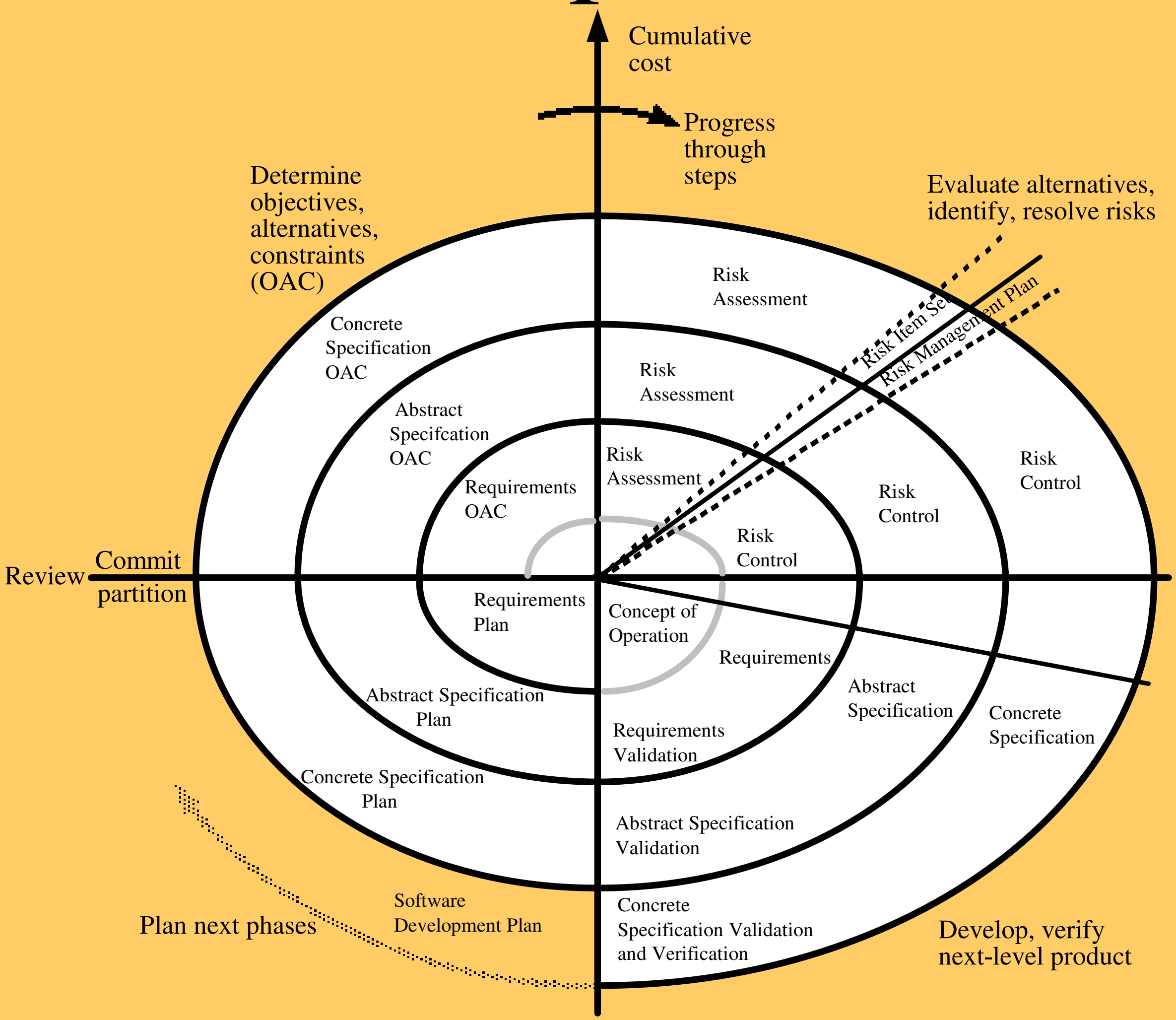


Iterative



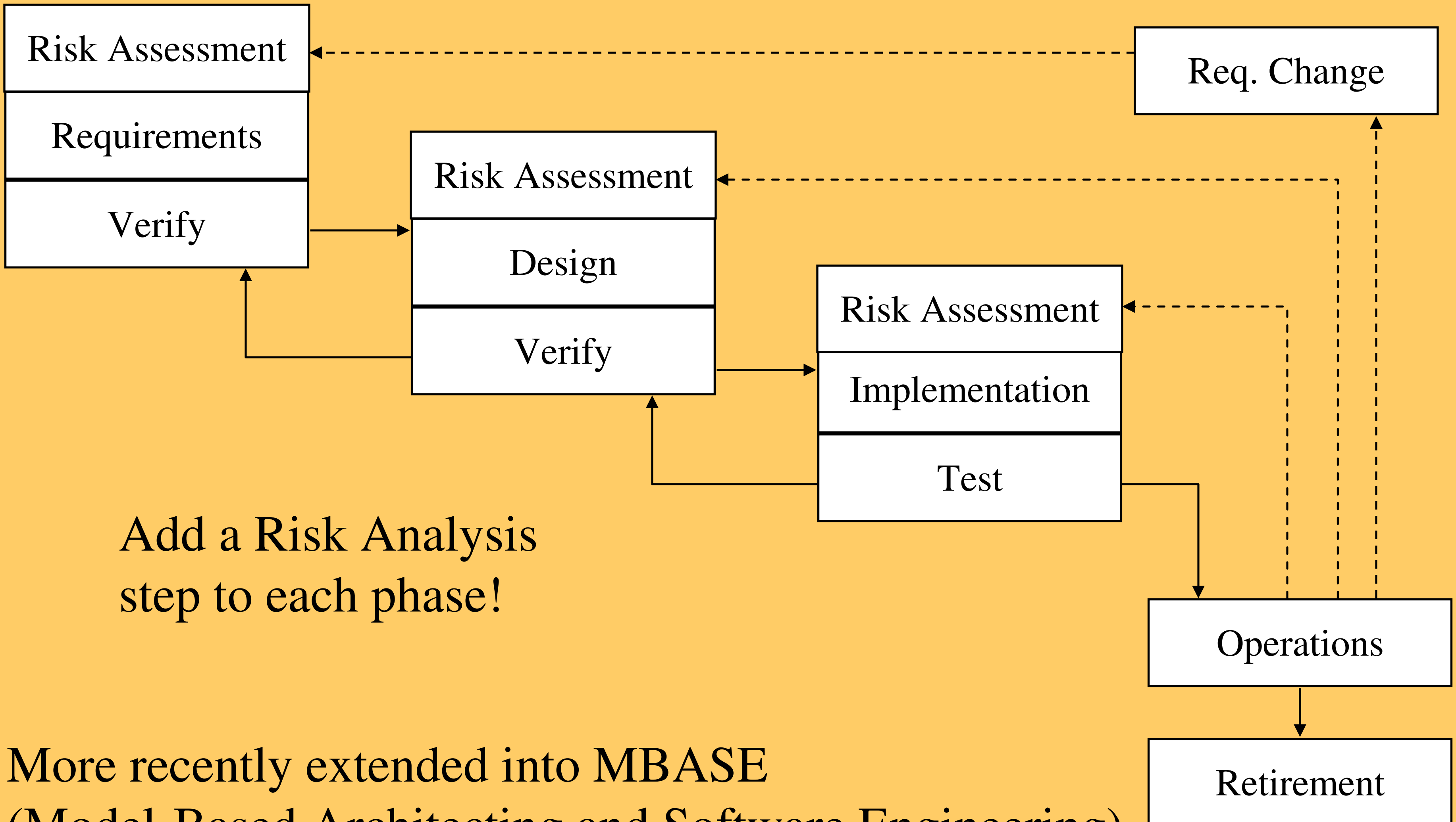
Iterations may revisit features
Requirements
Verify
Arch. Design
Verify
For each build: Perform detailed design, implement. Test. Deliver.
Operations
Retirement
Could be minimized or incorporated into iterations (common with agile approaches)

The Spiral Model[1]



1. Barry Boehm, "A Spiral model of Software Development and Enhancement", SPSE 1985

(Extremely) Simplified Spiral Model



Add a Risk Analysis
step to each phase!

More recently extended into MBASE
(Model-Based Architecting and Software Engineering)

TDD in Software Development Lifecycle

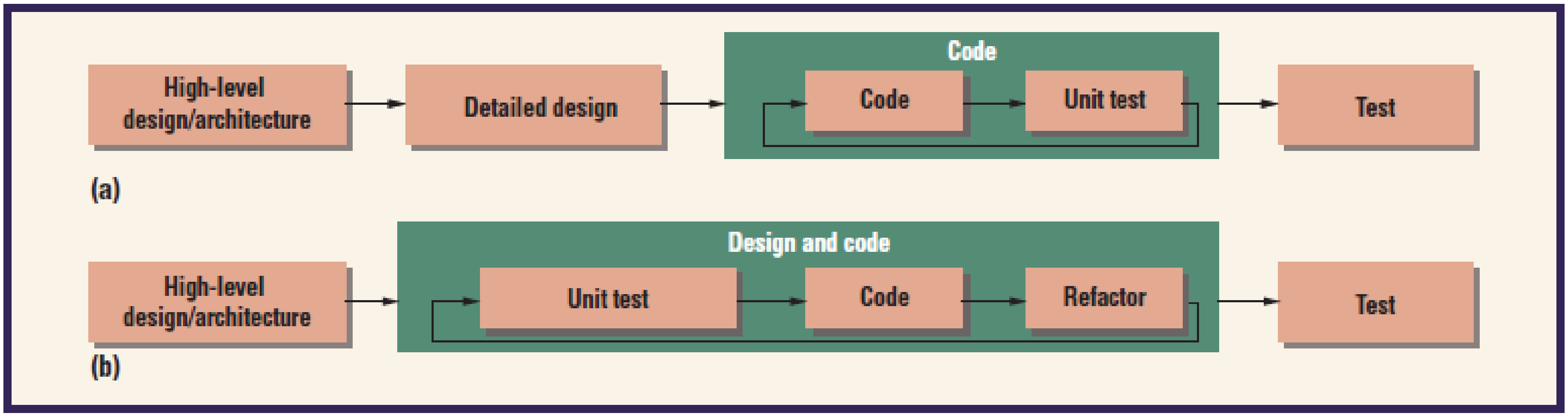
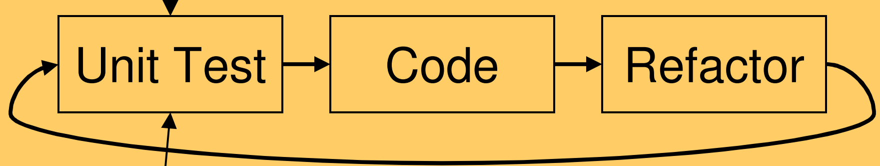


Figure 1. Development flow: (a) traditional test-last and (b) test-driven development/test-first flow.

What is Test-Driven Development?

- TDD is a design (and testing) approach
  … 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!

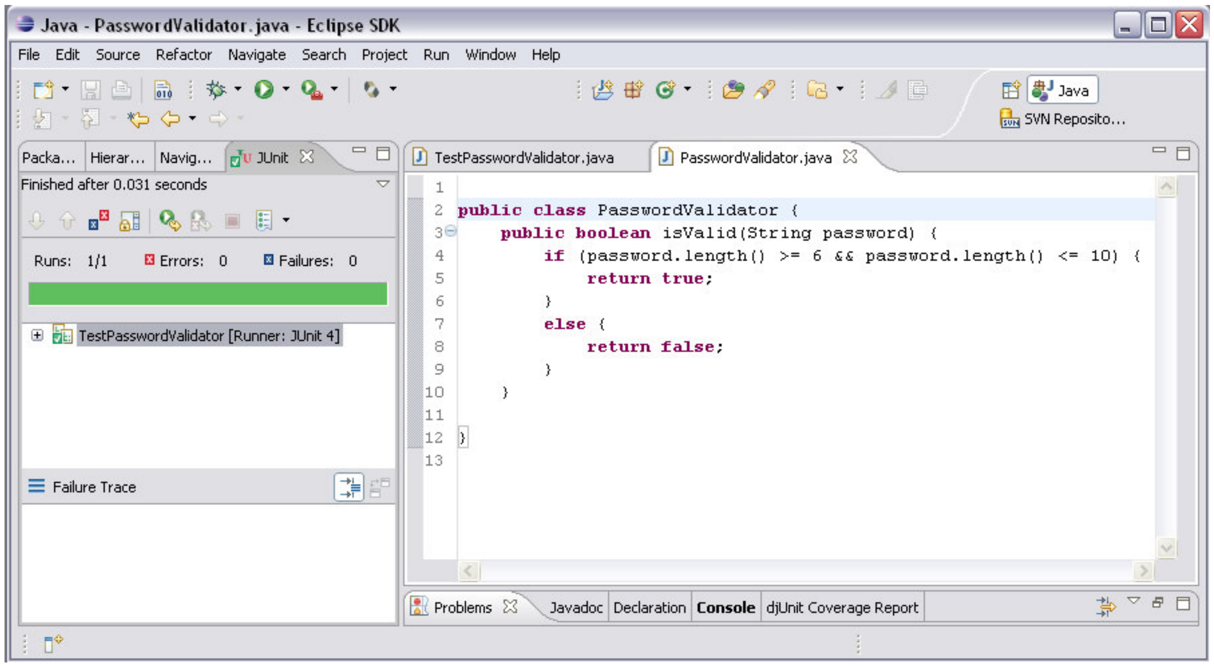
TDD Example: Write a test

```
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"));
  }
}
```

Design decisions:
  class name, constructor,
  method name, parameters and return type

TDD Example: Write the code

```
public class PasswordValidator {
  public boolean isValid(String password) {
    if (password.length() >= 6 && password.length() <= 10) {
      return true;
    }
    else {
      return false;
    }
  }
}
```



TDD Example: Refactor

```
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?

TDD Example: Refactor the test

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

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

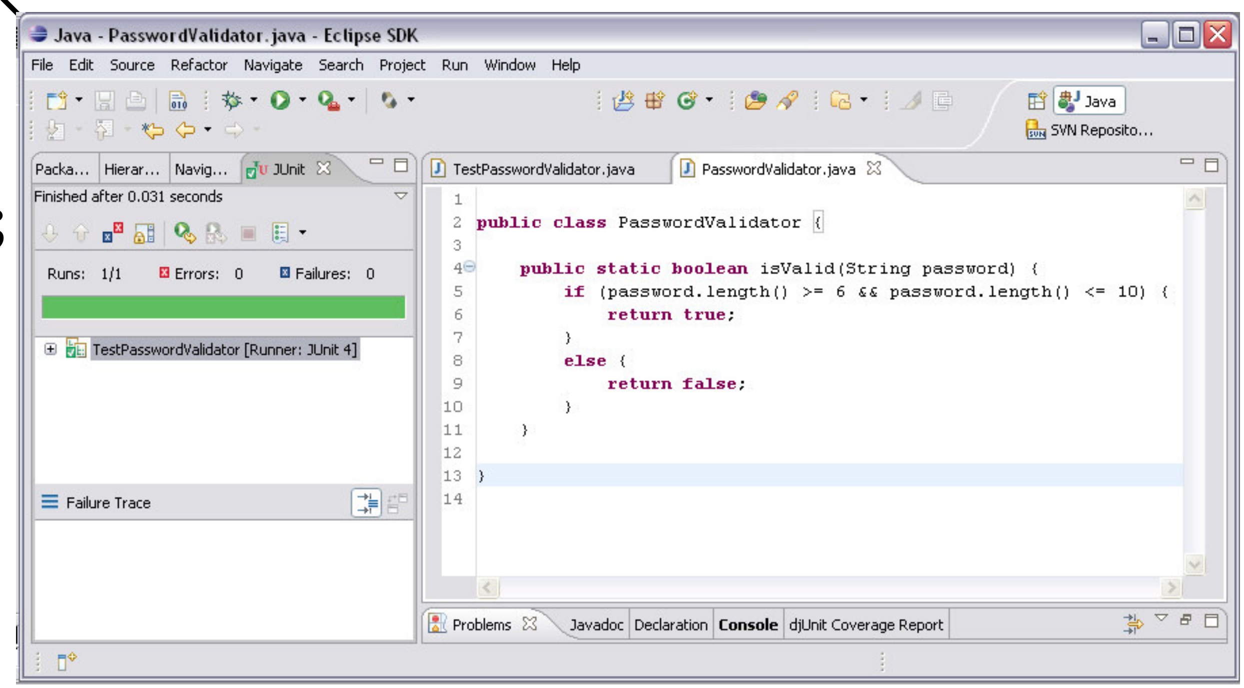
Design decision: static method

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

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



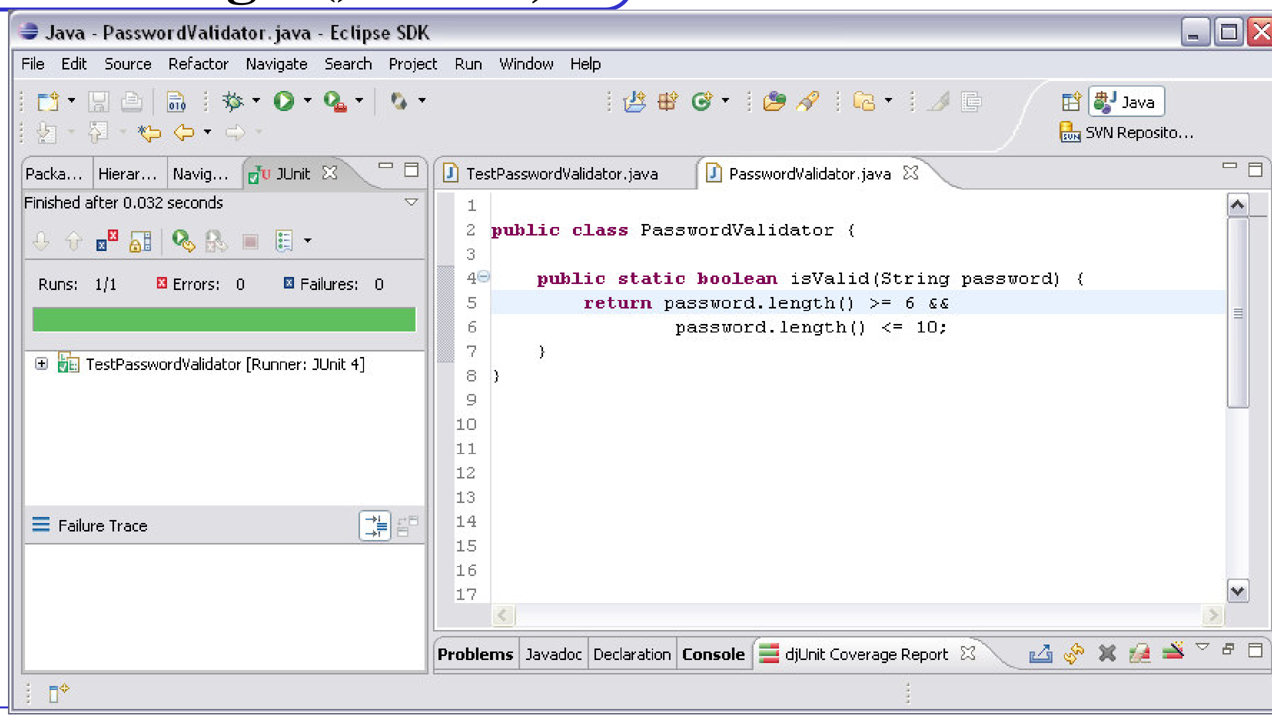
TDD Example: Refactor the code

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

Can we simplify this?

TDD Example: Refactoring #1

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



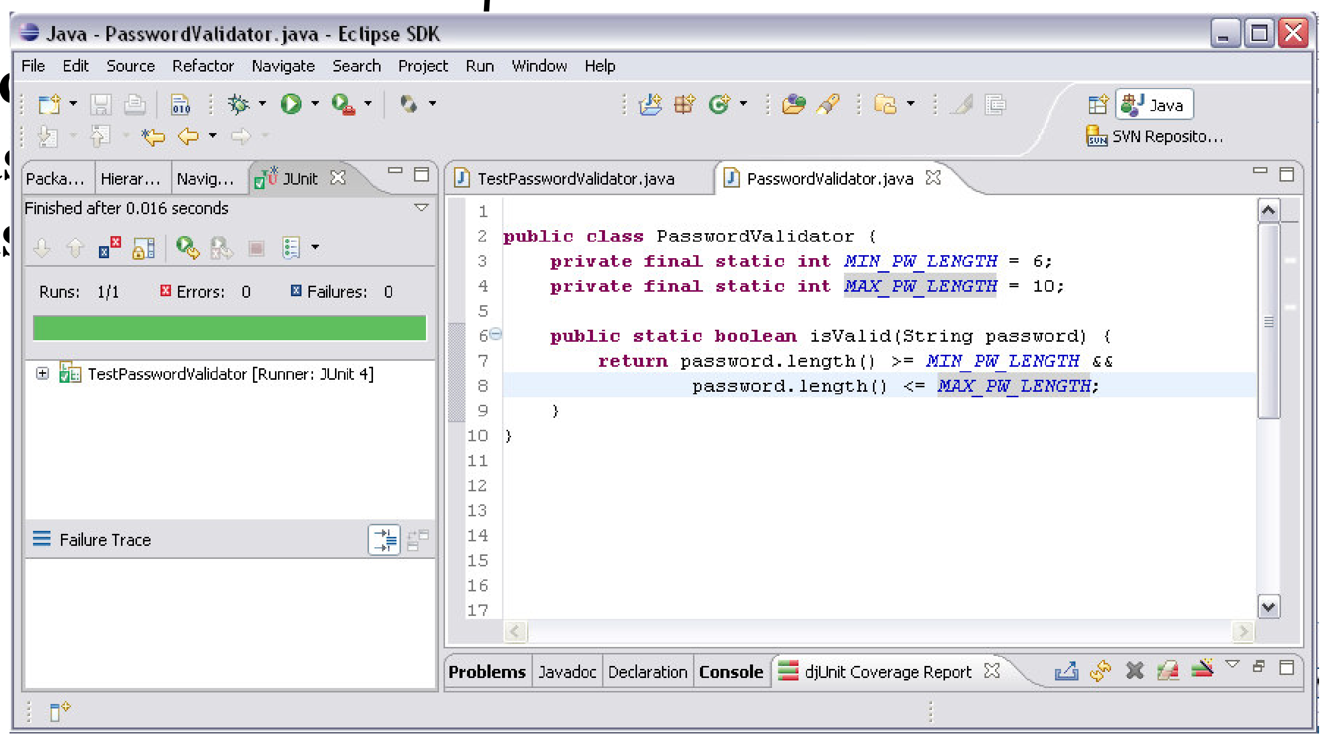
TDD Example: Refactoring #1

```
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

TDD Example: Refactoring #2

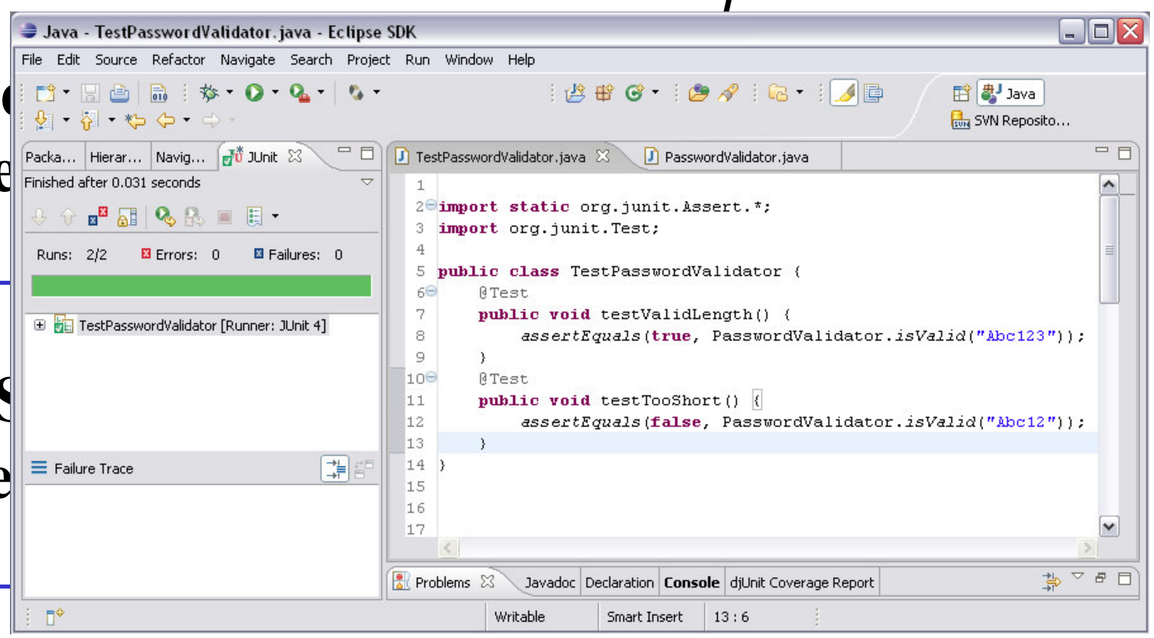
```
public class PasswordValidator {
    private final static int MIN_PW_LENGTH = 6;
    private final static int MAX_PW_LENGTH = 10;

    public stati
        return pa
                pa
    }
}
```



TDD Example: Write another test

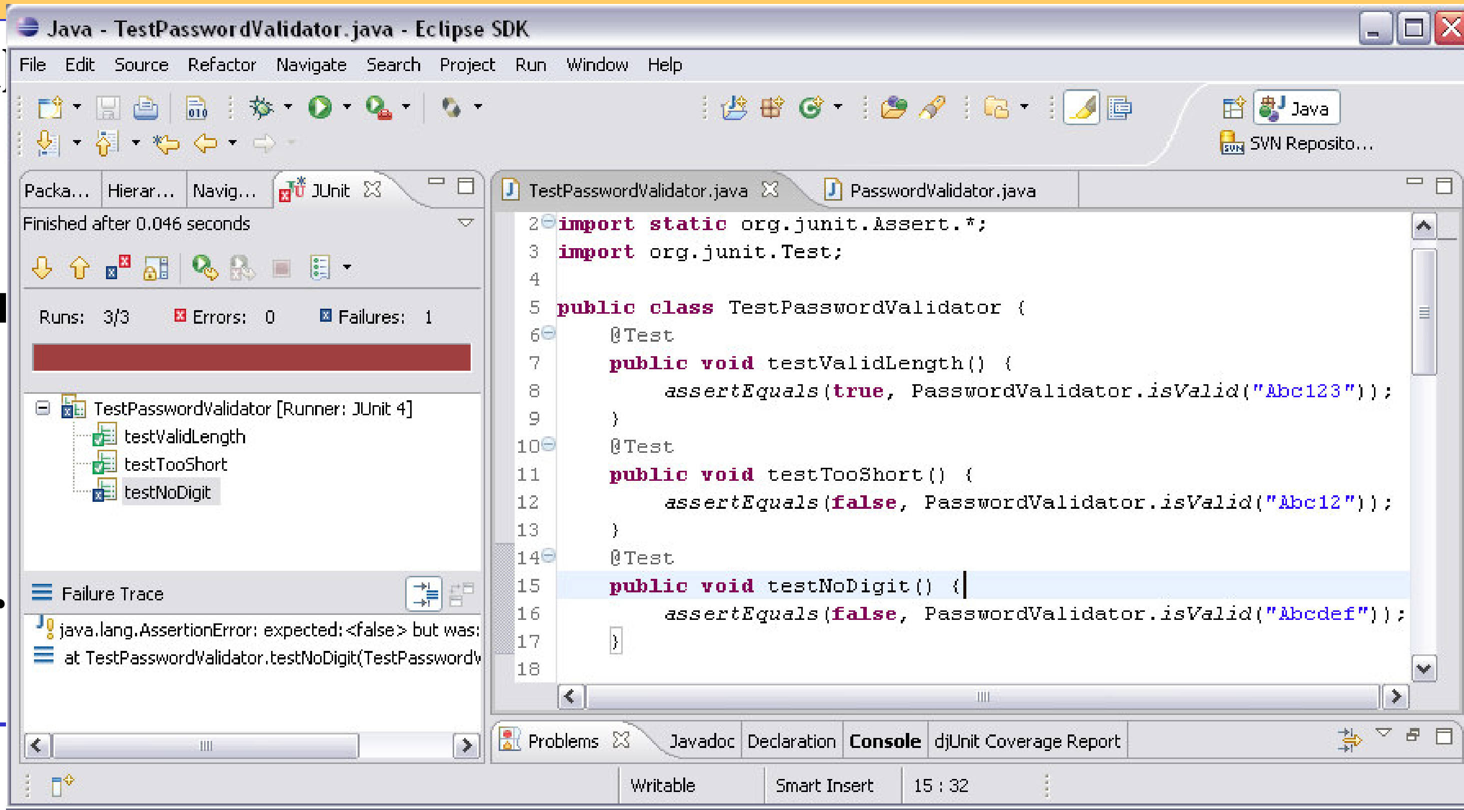
```
import static org.junit.Assert.*;
import org.junit.Test;

public class TestPasswordValidator {
  @Test
  public void testVali
     assertEquals(true
  }
  @Test
  public void testTooS
     assertEquals(false
  }
}
```

No design decisions; just unit testing



TDD Example: Write another test

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



TDD Example: Make the test pass

```
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;

Check for a digit

public class PasswordValidator {

privat

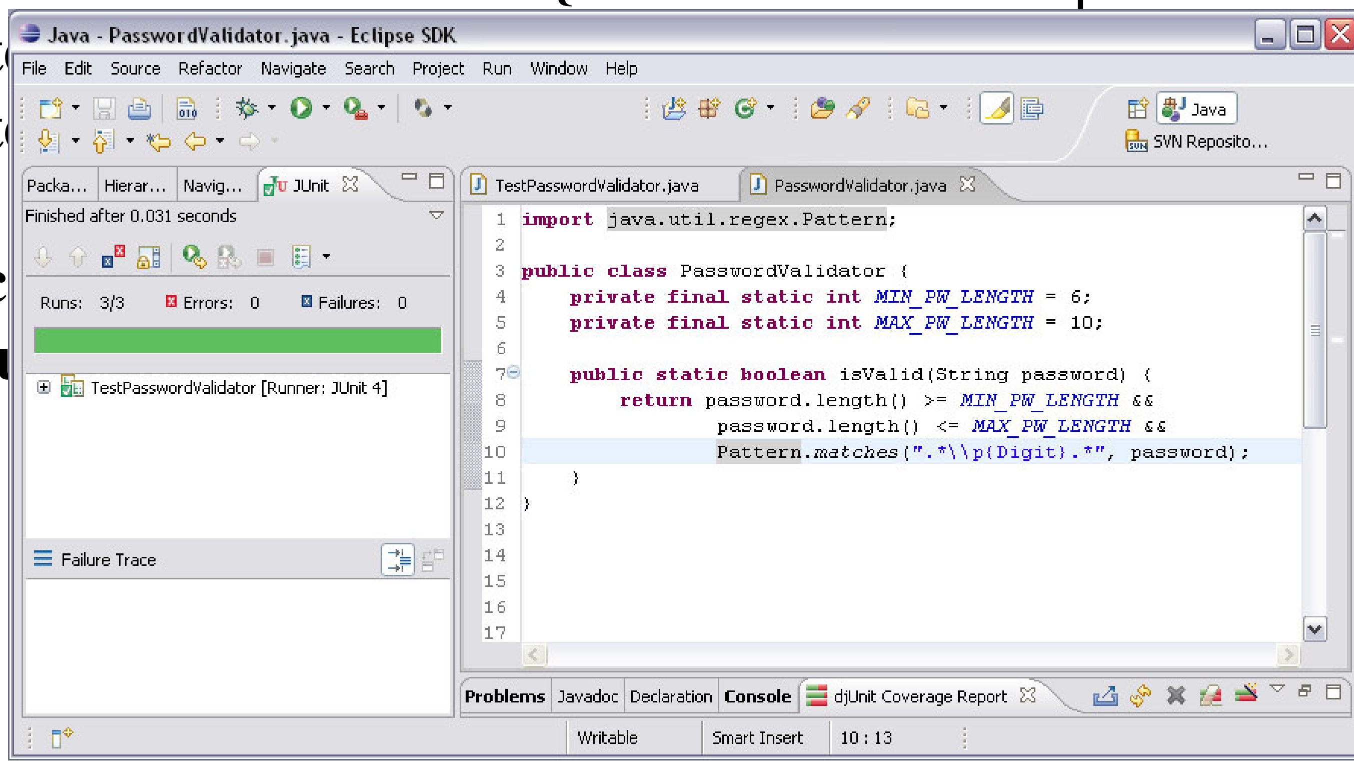
privat

public

retu

}

}



TDD Example: Refactor

```
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

TDD Example: Done for now

```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;
  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

- **Short introduction**[1]
  - 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.

1. http://www.agilealliance.org/programs/roadmaps/Roadmap/tdd/tdd_index.htm

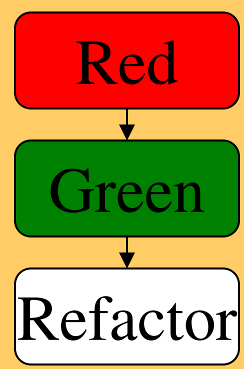
Test-Driven Development

- **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

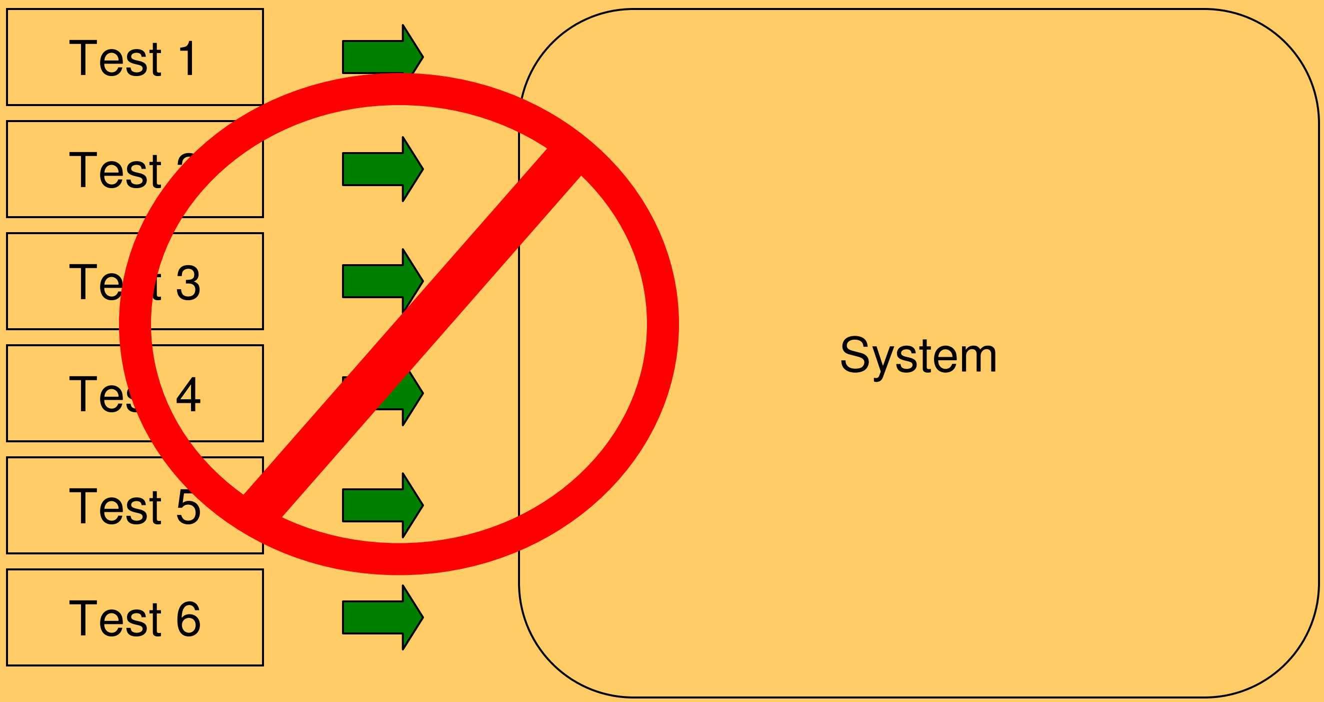
- Unit Testing ← TDD focuses here
  - Testing individual units (typically methods)
  - White/Clear-box testing performed by original programmer
- Integration and Functional Testing ← and may help here
  - Testing interactions of units and testing use cases
- Regression Testing ← and here
  - Testing 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 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

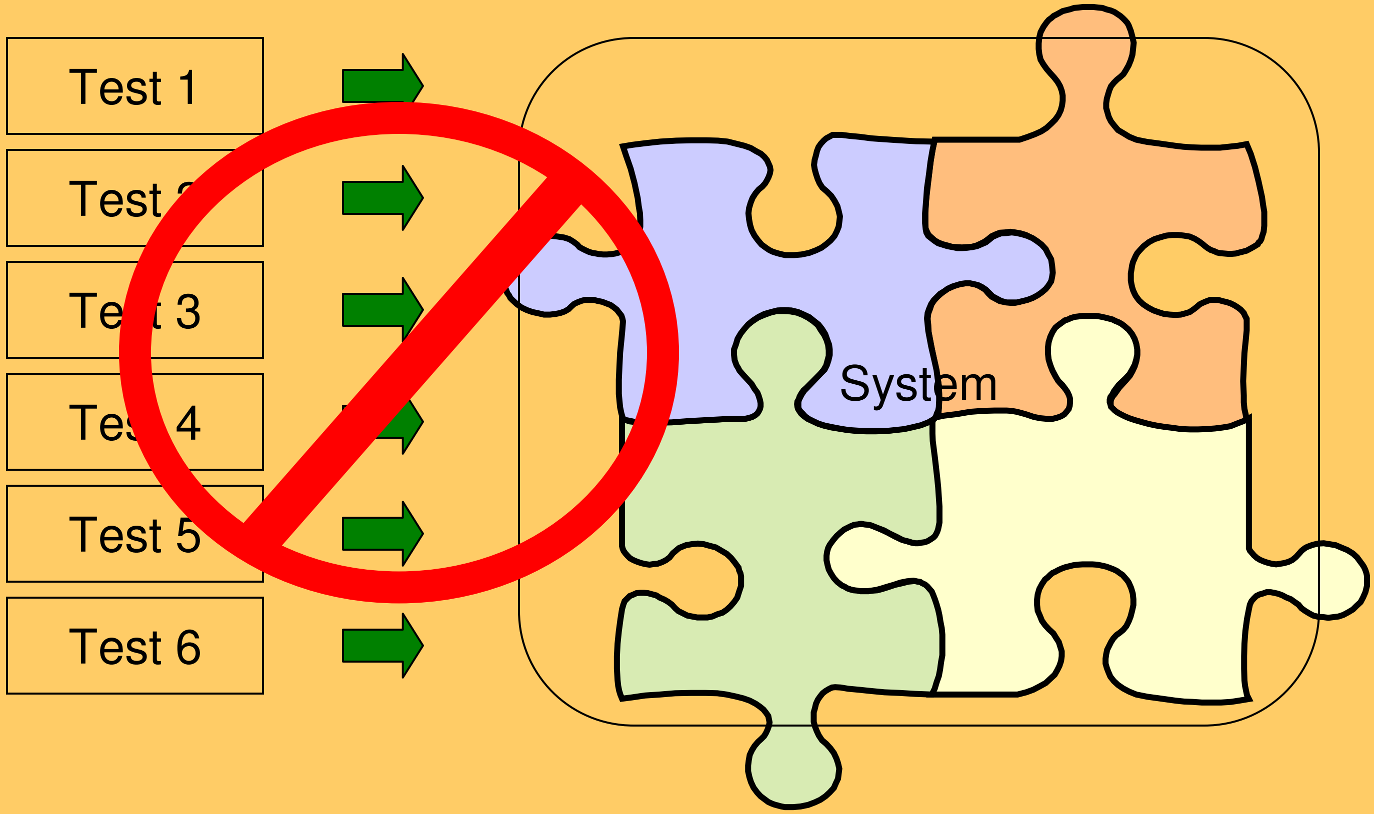
TDD Misconception #1

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



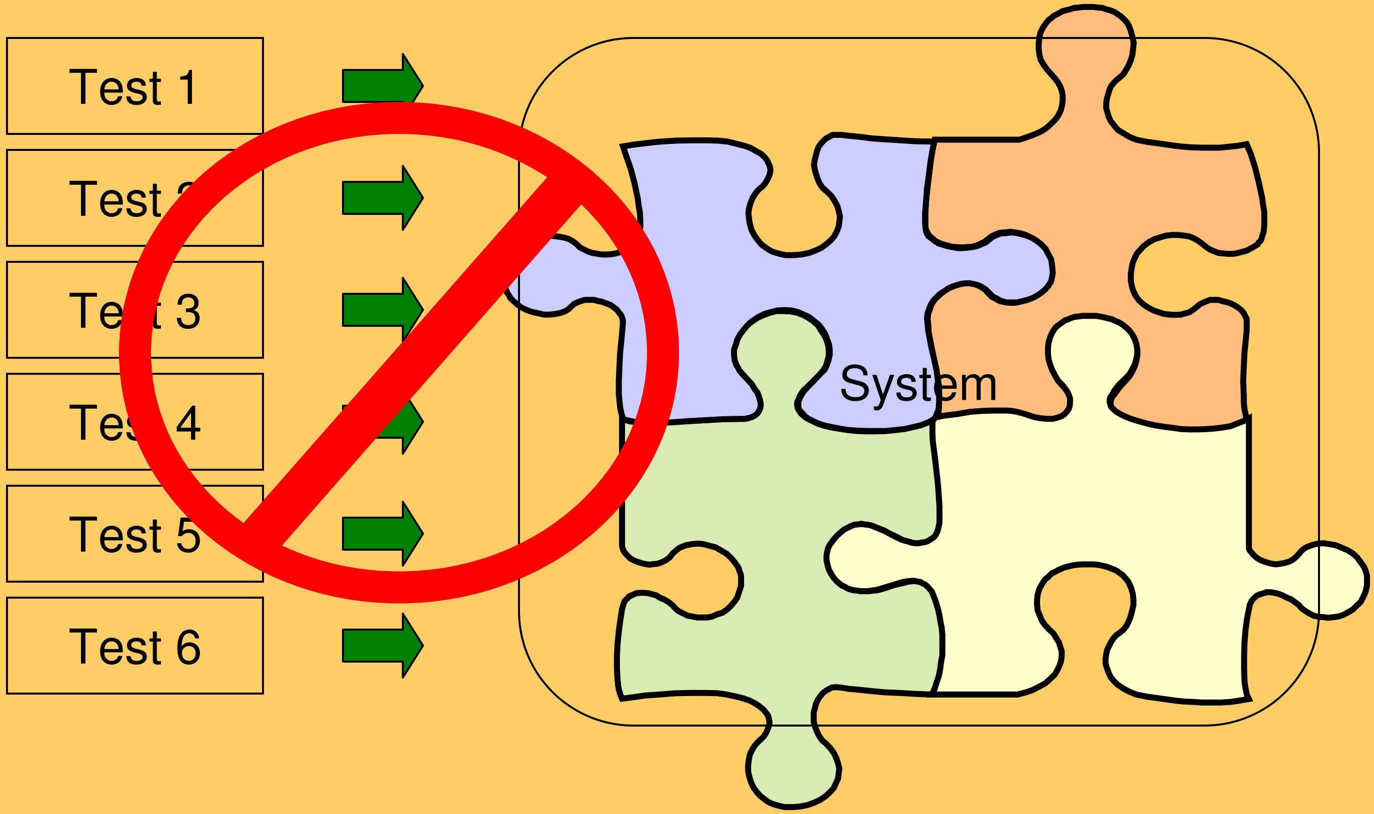
TDD Misconception #2

- TDD does not mean "write some of the tests, then build a system that passes the tests"



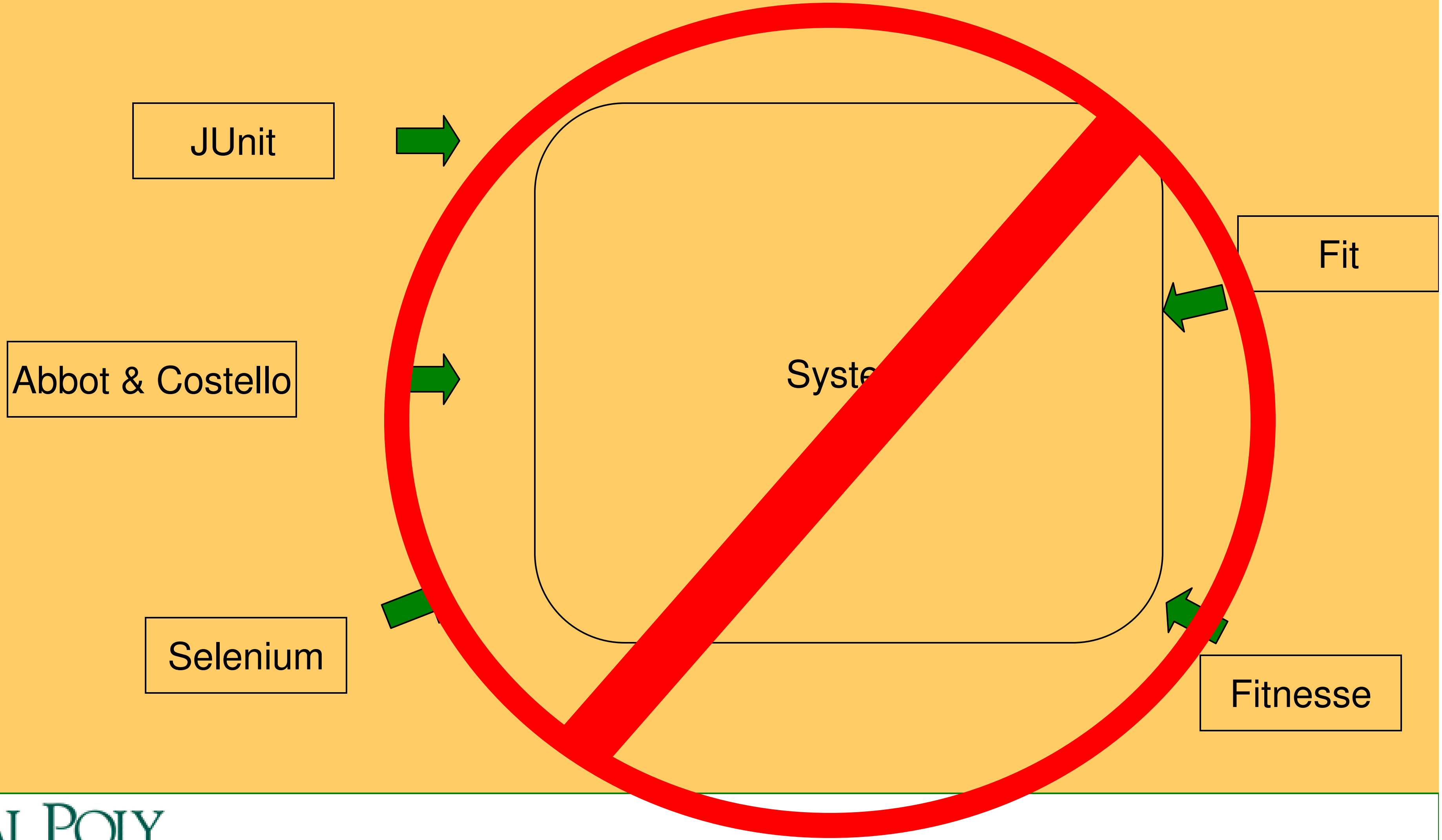
TDD Misconception #3

- TDD does not mean "write some of the code, then test it before going on"



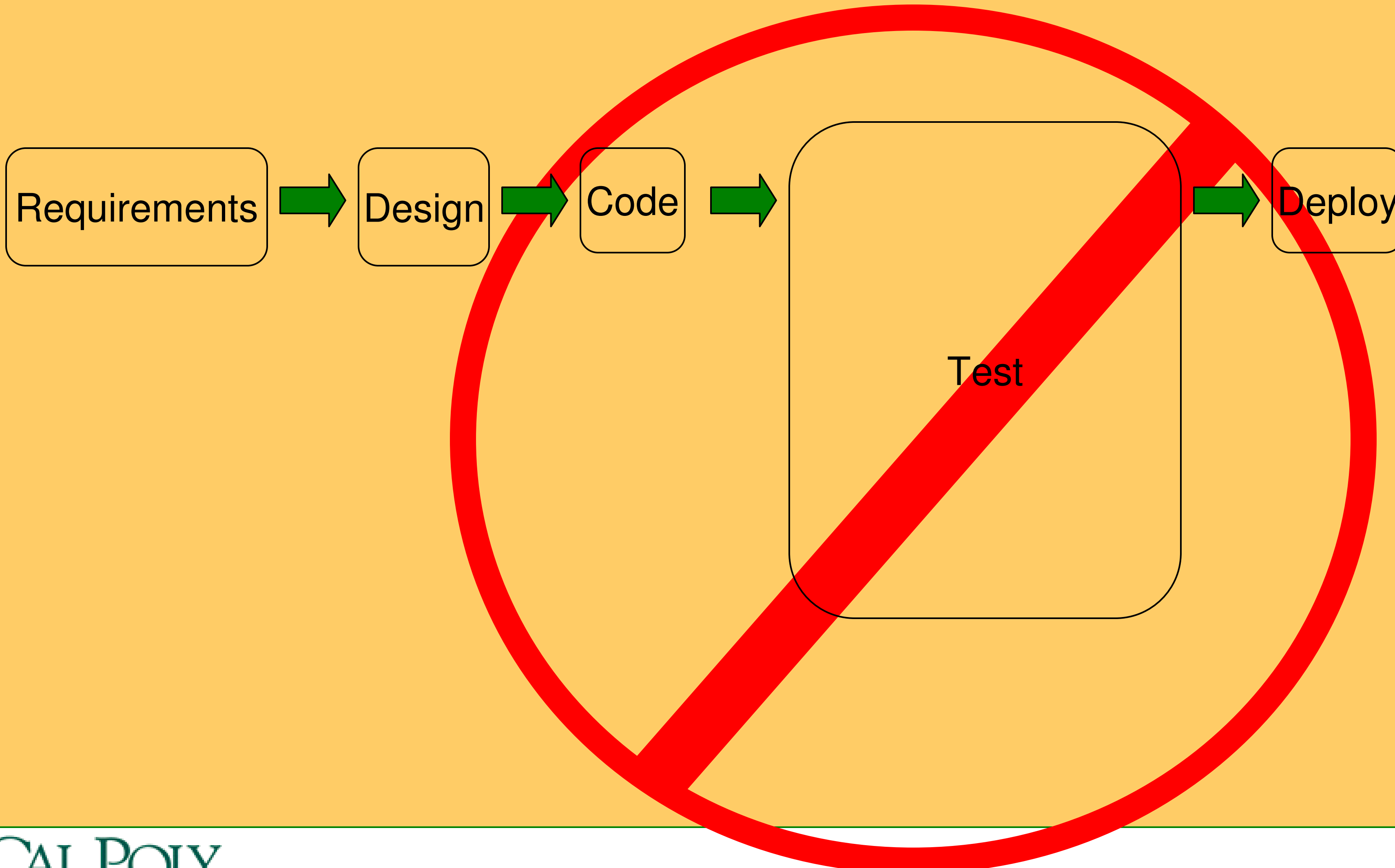
TDD Misconception #4

- 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*

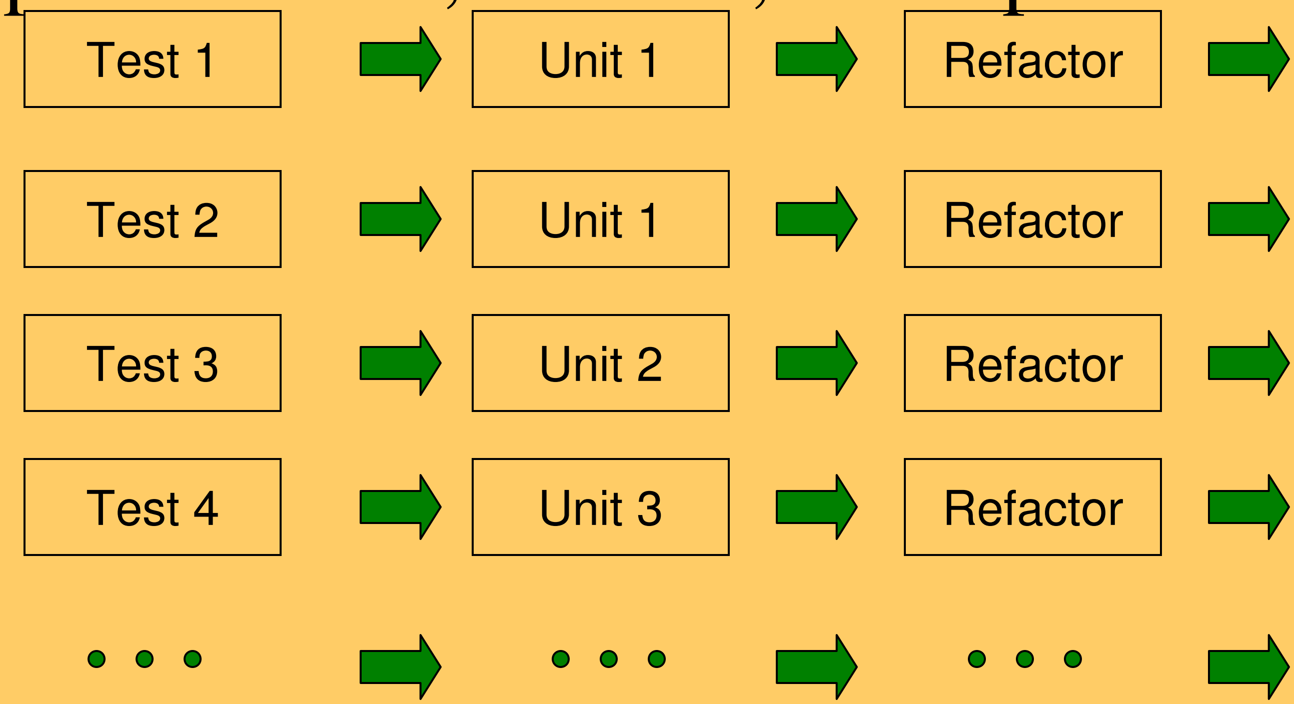
  (like pair programming, code reviews, and stand-up meetings)

  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

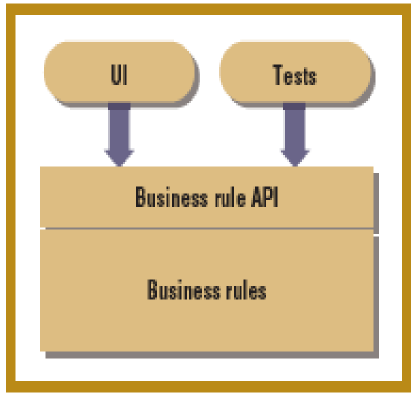


Figure 1. A testable system includes a test bus that can access the API independent of the UI.