# Review

## Lecture1

- All teams followed the same **software process**: (academic) **XP (Extreme Programming)**
- What is not S.E
  - **Not just software programming**
  - **Not just a process**
- What is S.E
  - The application of a **systematic**, **disciplined**, **quantifiable** approach to the **development**, **operation**, and **maintenance** of software.
- How to start working on open source
  - Before starting to work on issue
    - Build the project
    - Run the test cases
    - Read contribution guideline
    - Sign the (Contributor License Agreement) CLA
  - Working on issue
    - Start from "good first issue" or issue that the developer said that "it is easy"
    - Read the issue, ask question, tell developer “Hi, I would interested to contribute. Can I work on this issue?”
    - Add a test (if unavailable)
    - Discuss with developer about your planned solution if the developer replied

## Lecture2

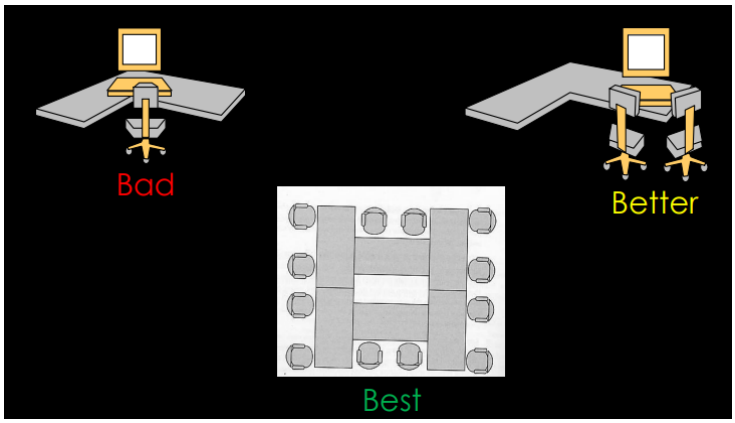
- Basic Git workflow
  - **Modify** files
    - If it was **changed** since it was checked out but has **not been staged**, it is **modified**
  - **Stage** files
    - If it’s **modified** but has been added to the **staging area**, it is **staged**
  - Do a **Commit**
    - If a particular version of a file is in the **git directory**, it’s considered **committed**
- Good reasons to branch
  - Fixing bugs in customer version
  - Experimental version
  - Political fights
- Bad reasons to branch
  - Support different hardware platform
  - Support different customer
- SCM according to the SEI
  - **Identification**

    - **Control**
    - **Status accounting**
    - **Audit and review**
- Various tests
    - **Smoke test**
    - **Unit test**
    - **Regression test**
- SCM Software Configuration Management Summary
    - Four aspects
        - **Change control**
        - **Version control**
        - **Building**
        - **Releasing**
    - Supported by tools
    - Requires expertise and oversight
    - More important on large projects

# Lecture3

- Waterfall process
    - Requirements
    - Design
    - Implementation
    - Verification
        - Integration
        - Testing
    - Maintenance
- Extreme Programming XP
    - Radically different from the rigid waterfall process
        - Replace it with a **collaborative** and **iterative** design process
    - Main ideas
        - **Don't write much documentation**
            - Working code and tests are the main written product
        - Implement features one by one
        - Release code frequently
        - Work closely with the customer
        - Communicate a lot with team members
- Creator of XP: KENT BACK
- XP: Some Key Practice
    - **Planning game** for requirements
    - **Test-driven development** for design and testing
    - **Refactoring** for design
    - **Pair programming** for development

    - **Continuous integration** for integration
- XP Simplicity rules
    - Runs all the tests
    - Minimizes coupling
    - Maximizes cohesion
    - Says everything “Once and only once”
- Pair programming: A simple, straightforward concept. Two programmers work **side-by-side** at **one** computer, continuously collaborating on the **same** design, algorithm, code, and test. It allows two people to produce a **higher quality** of code than that produced by the summation of their solitary efforts.

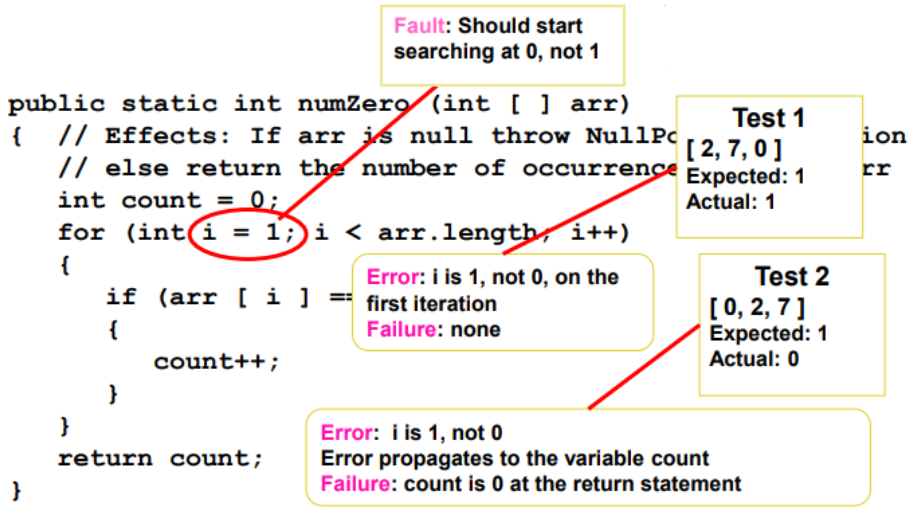



- Why does pair programming work?
    - Pair Pressure
    - Pair Negotiation
    - Pair Courage
    - Pair Reviews
    - Pair Debugging
    - Pair-Learning
- User Story
    - A feature **customers want** in the software
    - Written by **customers** (through communication with developers), and **not by developers**
- Concepts
    - **Story point**: unit of measure for expressing the overall size of a user story, feature, or other piece of work. The **raw value of a story point is unimportant**. What matters are the **relative values**
    - **Ideal time**: the amount of time “something” takes **when stripped of all peripheral activities**
    - **Elapsed time**: the amount of time that passes on the **clock** to do “something”
    - **Velocity**: measure of a team’s rate of progress
        - Velocity is calculated by **summing the number of story points assigned to each user story** that the team **completed** during the operation
- XP works best when
    - Educated **customer on site**
    - Small team
    - People who like to talk
    - All in one room (including customer)
    - Changing requirements
- Working Software
    - All software **has automated (unit) tests**

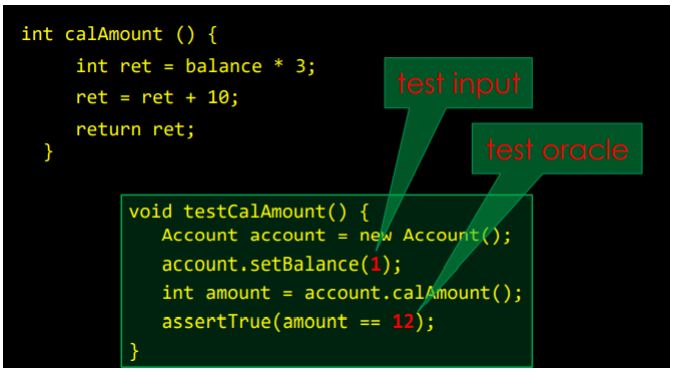
- **All tests pass**, all the time
  - **Never check in broken code**

# Lecture4

- Terminology
  - **Mistake**: Programmer makes a mistake
  - **Fault(defect, bug)**: Appears in the program
    - 医生试图诊断**根本原因**，即疾病
    - 代码中具体哪个位置写错了（问题的根源）
  - **Failure**: Program failure occurs during execution (program behaves unexpectedly)
    - 病人给医生一份**症状清单**
    - 在程序中，如果输出结果与预期不符合（外观上），即有 Failures
    - 当一个系统不能执行所要求的功能时，即为Failure
  - **Error**: Difference between computed, observed, or measured value or condition and true, specified, or theoretically correct value or condition
    - 医生可能会寻找**内部异常情况**(高血压、心律不齐、血流中有细菌)
    - 在程序中，如何由 Fault 导致出现了某些 Error（尽管可能不会导致 Failures）
    - 计算、观察或测量值或条件，与真实、规定或理论上正确的值或条件之间的差异




- Test input & Test oracle




- JUnit
  - Used to write and run repeatable **automated tests**
  - A structure for writing **test drivers**

- JUnit **features** include
  - **Assertions** for testing expected results
  - **Sharing common test data** among tests
  - **Test suites** for easily organizing and running tests
  - **Test runners**, both graphical and textual
- Can be used as **stand alone** Java programs (from command line) or **from an IDE** such as IntelliJ or Eclipse
- Need to use methods of `junit.framework.assert` class
  - `javadoc` gives a complete description of its capabilities
- Each test method checks a condition (**assertion**) and reports to the test runner whether the test **succeeded** or **failed**
  - No assertion = no junit test
- All of the methods **return void**

- JUnit Test
  -
```java
import org.junit.Test;
import static org.junit.Assert.*;
public class calcTest{
    private Calc calc;
    @Test
    public void testAdd(){
        calc = new Calc ();
        // Expected result     // The test
        assertEquals((long) 5, calc.add(2,3));
    }
}
```

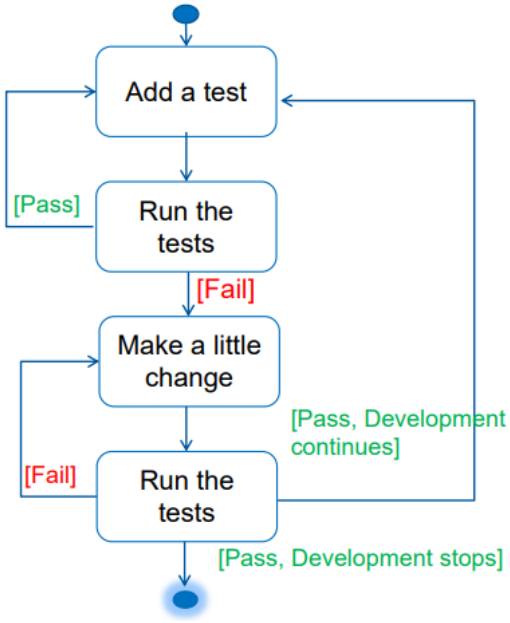
- JUNIT Test Fixtures
  - A test fixture is the **state** of the test
    - Objects and variables used by more than one test
    - Initializations (prefix values)
      - They should be initialized in a `@Before` method
    - Reset values (postfix values)
      - Can be deallocated or reset in an `@After` method
  - Objects in fixtures declared as **instance variables**
- Assertion patterns
  - **State Testing Patterns**
    - **Final State Assertion**
      - Most Common Pattern: Arrange. Act. Assert.
    - **Guard Assertion**
      - Assert Both Before and After The Action (Precondition Testing)
    - **Delta Assertion**
      - Verify a Relative Change to the State
    - **Custom Assertion**

        - Encodes Complex Verification Rules
    - **Interaction Assertions**
        - Verify Expected Interactions
        - Heavily used in Mocking tools
- JUnit Theories
    - Contract Model
        - **Assume**
        - **Act**
        - **Assert**
    - ```
      @Theory public void removeThenAddDoesNotChangeSet(
          Set<String> set, String string) { // Parameters!
          assumeTrue(set.contains(string)) ; // Assume
          Set<String> copy = new HashSet<String>(set); // Act
          copy.remove(string);
          copy.add(string);
          assertTrue (set.equals(copy)); // Assert
          // System.out.println("Instantiated test: " + set + ", " + string);
      }
      ```
    - Where does data come from
        - All combinations of values from `@DataPoint` annotations  where assume clause is true
        - Note: `@DataPoint` format is an array.

      ```
      @DataPoints
      public static String[] string = {"ant", "bat", "cat"};
      @DataPoints
      public static Set[] sets = {
          new HashSet(Arrays.asList("ant", "bat")),
          new HashSet(Arrays.asList("bat", "cat", "dog", "elk")),
          new HashSet(Arrays.asList("Snap", "Crackle", "Pop"))
      };
      ```

- Kent Beck's rules
    - Never write a single line of code unless you have a failing automated test
    - Eliminate duplication
      ```

- Steps in Test Driven Development (TDD)




  - Quickly add a test
  - Run all tests and see the new one fail
  - Make a little change to code
  - Run all tests and see them all succeed
  - Refactor to remove  duplication
- Test First Scenario
  - **Write test** for the newly added functionality
  - **Run the tests**: your team gets to see the failing ones
  - **Make them Pass**: Add code to make the failing tests pass
  - **Add more tests**, and run the new set of tests to see the failing ones
    - Newly added components or helper methods
    - Checking for corner cases
  - **Make them Pass**: Add more code to make the added tests pass
- Which test is correct?

**Example 1**

```
@Test
public void popTest() {
 MyStack s = new MyStack ();
  s.push (314);
  assertEquals (314, s.pop ());
}
@Test
public void sizeTest() {
 MyStack s = new MyStack ();
 s.push (2);
 assertEquals (1, s.size ());
}
```

**Example 2**

```
MyStack s = new MyStack ();
@Test
public void popTest() {
 s.push (314);
 assertEquals (314, s.pop ());
}

@Test
public void sizeTest() {
 s.push (2);
 assertEquals (1, s.size ());
}
```

**Example 1**

```
@Test
public void sizeTest() {
  MyStack s = new MyStack ();
  assertEquals (0, s.size ());
  s.push (2);
  assertEquals (1, s.size ());
}
```

**Example 2**

```
@Test
public void emptyTest() {
  MyStack s = new MyStack ();
  assertEquals (0, s.size ());
}

@Test
public void sizeTest() {
  MyStack s = new MyStack ();
  s.push (2);
  assertEquals (1, s.size ());
}
```

**Example 1**

```
@Test
public void sizeTest() {
 MyStack s = new MyStack ();
 assertEquals (s.size (),0);
}
```

**Example 2**

```
@Test
public void emptyTest() {
  MyStack s = new MyStack ();
  assertEquals (0, s.size ());
}
```

**Example 1**

```
@Test
public void sizeTest() {
 MyStack s = new MyStack ();
 s.push("Hello");
 assertEquals ("Hello", s.pop());
}
```

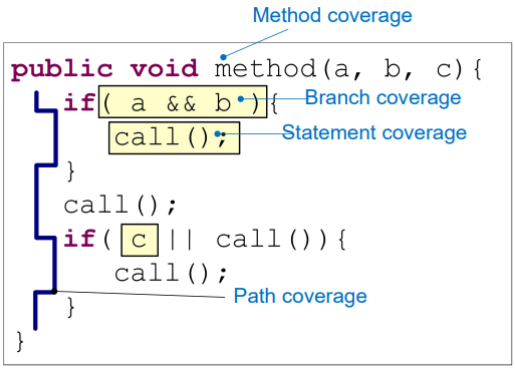
**Example 2**

```
@Test
public void emptyTest() {
 MyStack s = new MyStack ();
 s.push("Hello");
 assertTrue (s.pop()=="Hello");
 }
```

# Lecture5

- Code Coverage
  - A measure used to describe **the degree to which the source code of a program is executed** when a **particular test suite runs**
  - Code Coverage is classified as **White box texting**
  - Benefits

        - **Identify untested part of codebase**
        - **Improve the Quality by improved test coverage**
        - **Identify testing gaps or missing tests**
        - **Identify the redundant/dead Code**
    - Coverage Criteria




        - Instructions Coverage
        - Statements Coverage
        - Branch Coverage
        - Method Coverage
        - Class Coverage
    - Code Coverage Analysis Process
        - Writing test cases and execute them
        - Finding areas of code not covered using Code Coverage Tool
        - Creating additional tests for identified gaps to increase test coverage
        - Determining a quantitative measure of code coverage
- Code Coverage Tools
    - Cobertura
    - Atlassian Clover
    - DevPartner
    - JTest
    - Bullseye for C++
    - Sonar
    - Kalistic
- open-source test generation tools
    - Randoop
    - Evosuite
- build system
    - Ant
    - Maven
    - Gradle
- Maven
    - Provide ways to help with managing
        - Builds
        - Documentation
        - Reporting
        - Dependencies
        - SCMs
        - Releases

- Distribution

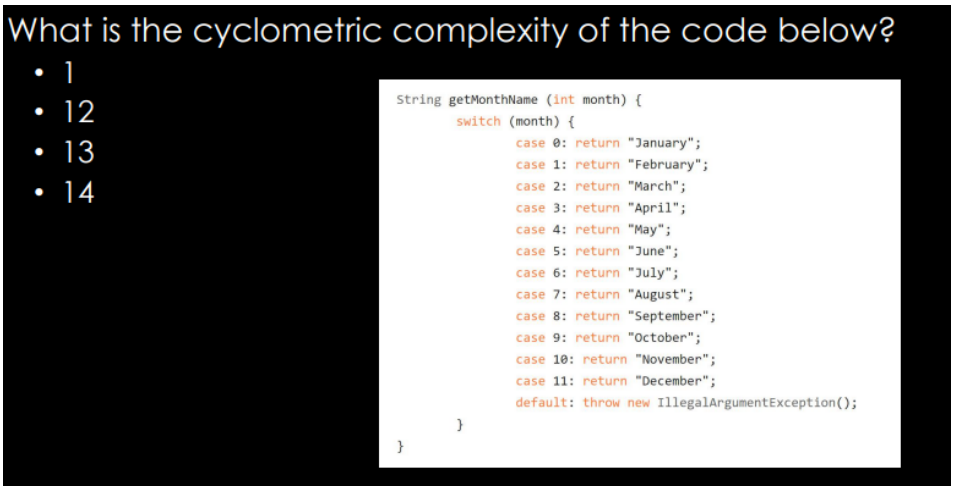
```
// removing all the files and directories generated by Maven during its build?
mvn clean

// command for running tests in your project
mvn test
```

- Maven Phase
  - Common default lifecycle phases
    - validate
    - compile
    - test
    - package
    - integration-test
    - verify
    - intall
    - deploy
  - beyond the default list above
    - clean
    - site

# Lecture6

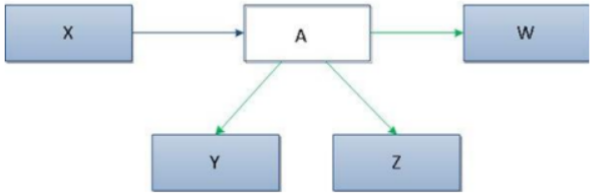
- Measure size of system
  - Lines of code
  - Complexity
    - **Cyclomatic complexity**
      - Cyclomatic complexity is the **number of independent paths** through the procedure




      14
      - Σ(if, elseif, else, case, default, for, while) + 1
    - **Function points**
    - **Coupling and Cohesion**

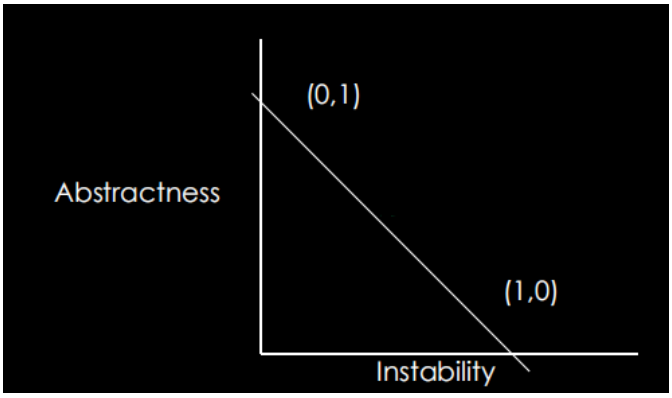
- Dhama Coupling Metric

```
Module coupling = 1 / ( number of input parameters +
                        number of output parameters +
                        number of global variables used +
                        number of modules called +
                        number of modules calling
                        )
```

- Afferent coupling(CA)
- Efferent coupling(CE)
- Instability = Ce / (Ca + Ce)
  - 

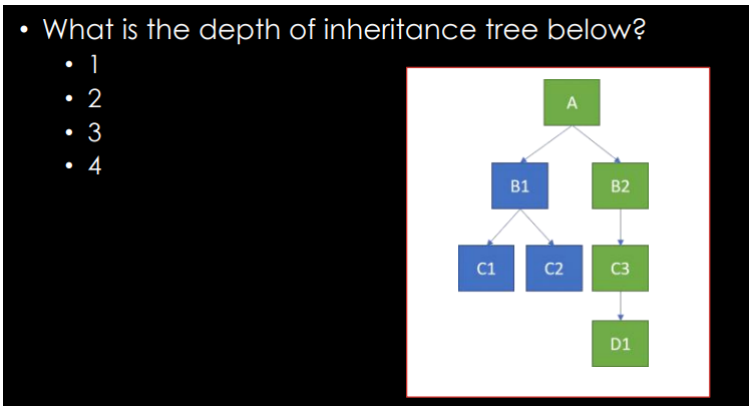
  - Ce = 3 Ca = 1, Instability = 3/4
- 很多的 Ce > 20 和不是很多的 Ca（值 I 接近1）是不稳定的

  很多的 Ca 和不是很多的 Ce（值 I 接近 0）意味着组件的稿稳定性，因为他们的责任更大，所以更难修改
- Abstractness = number of abstract classes in module / number of classes in module




- **Technical OO Metrics**
  - **Weighted Methods Per Class (WMC)**
    - the sum of the complexities of the methods in the class
      - 1 (number of methods)
      - Lines of code
      - Number of method calls
      - Cyclomatic complexity
  - **Depth of Inheritance Tree (DIT)**

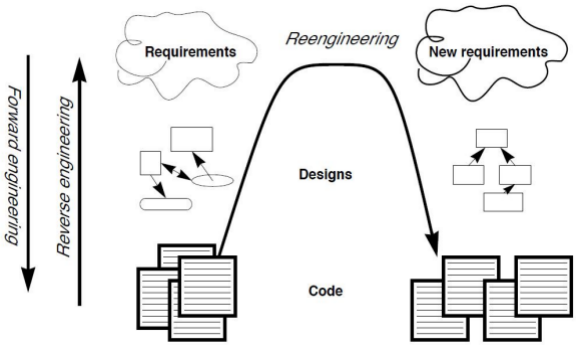
        - Maximum length from a class to the root of the tree




    - **Number of Children (NOC)**
        - Number of immediate subclasses
    - **Coupling between Object Classes (CBO)**
        - Number of other classes to which a class is coupled
        - Class A is coupled to class B if there is a method in A that invokes a method of B
    - **Response for a Class (RFC)**
        - Number of methods in a class or called by a class
    - **Lack of Cohesion in Methods (LCOM)**
        - Number of pairs of methods that don't share instance variables minus number of pairs of methods that share instance variables
        - $Num_{\text{don't share instance}} = Num_{\text{share instance}}$

# Lecture7

- Terminology



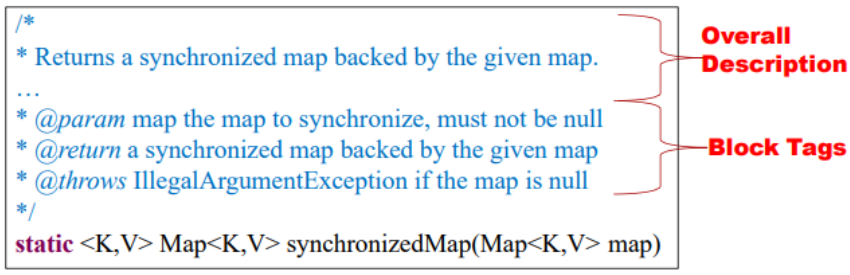

    - **Forward engineering**
        - From requirements to design to code
    - **Reverse engineering**
        - From code to design, maybe to requirement
    - **Reengineering**
        - From old code to new code via some design
- Format of a Patterns
    - Name
    - Intent
    - Problems
    - Solution
        - steps to apply the pattern

    - Tradeoffs
        - Pros
        - Cons
        - Difficulties
    - Rationale
    - Known Uses
    - Related Patterns
    - What Next
- Metrics for Reverse Engineering
    - Coupling
    - Frequent changes
        - % of git changes that affect a module (class)
    - Bugs
        - Code coverage

# Lecture8

- What is testing
    - Dynamic Analysis
    - Static Analysis
- Tools for Static Analysis
    - Checkstyle
    - PMD
    - FindBugs
- Defensive Programming
    - Making sure that **no warning produced by Static Analysis** is a form of defensive programming
    - Defensive programming is a form of defensive design intended to **ensure the continuing function of a piece of software under unforeseen circumstances**
    - Risky programming construct
        - Pointers
        - Dynamic memory allocation
        - Floating-point numbers
        - Parallelism
        - Recursion
        - Interrupts
    - Defensive Programming Examples
        - Use boolean variable not integer
        - Test i <= n not i = = n (greater range)
        - Assertion checking (e.g., validate parameters)
        - Build debugging code into program with a switch to display values at interface
        - Error checking codes in data (e.g., checksum or hash)
    - Security in the Software Development Process
        - Agents and Component
        - Barriers

        - Authentication & Authorization
        - Encryption
- JavaDoc
    - Example




- @param - Parameter name, Description
- @return - Description
- @throws - Exception name, Condition under which the exception is thrown

    - ```
      /**
      * This is where the text starts. The asterisk lines
      * up with the first asterisk above; there is a space
      * after each asterisk. The first sentence is the most
      * important: it becomes the “summary.”
      *
      * @param x Describe the first parameter (don’t say its type).
      * @param y Describe the first parameter (don’t say its type).
      * @return Tell what value is being returned (don’t say its type).
      */
      public String myMethod(int x, int y) { // p lines up with the / in /*
      ```

    - javadoc comments begin with `/**` and end with `*/`
    - Javadoc comments must be immediately before
        - a class (plain, inner, abstract, or enum)
        - an interface
        - a constructor
        - a method
        - a field(instance or static)
    - Who the javadoc is for
        - Javadoc comments **should** be written for programmers who **want to use your code**
        - Javadoc comments **should not** be written for
            - Programmers who need to debug, maintain, or upgrade the code
            - People who just want to use the program
        - Javadoc comments **should** describe exactly **how to use** the class, method, constructor
        - Javadoc comments should not describe the **internal 1workings of the class** or method (unless it affects the user in some way)
    -
      ```

- Use the standard ordering for javadoc tags
  - In class and interface descriptions, use:

    @author *your name*
    @version *a version number or date*
    - ***Use the @author tag in your assignments for all top-level classes!!!***
    - These tags are only used for classes and interfaces
  - In method descriptions, use:

    @param ***p*** *A description of parameter p.*
    @return *A description of the value returned (not used if the method returns void).*
    @exception *e* *Describe any thrown exception.*

    Do *not* mention the type in @param and @return tags--javadoc will do this (and get it right)

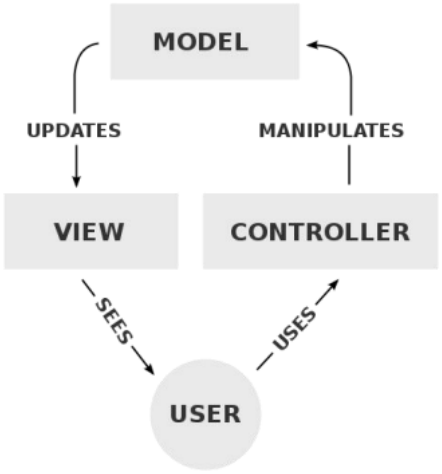
  - **Do not mention the type** in `@param` and `@return` tags-- javadoc will do this (and get it right)
  - 
    - The *first sentence* should summarize the purpose of the element
    - For methods, omit the subject and write in the third-person narrative form
      - Good: Finds the first blank in the string.
      - Not as good: Find the first blank in the string.
      - Bad: This method finds the first blank in the string.
      - Worse: Method findBlank(String s) finds the first blank in the string.
    - Use the word this rather than "the" when referring to instances of the current class (for example, Multiplies this fraction...)
    - Do not add parentheses to a method or constructor name unless you want to specify a particular signature
    - Keep comments up to date!
  - Test code inconsistency `@tCommaent`

# Lecture9

- Types of Software Reuse
  - Application System Reuse
  - Component Reuse
  - Function Reuse
- Benefits of Reuse
  - Increased Reliability
  - Reduced Process Risk
  - Effective Use of Specialists
  - Standards Compliance
  - Accelerated Development
- Reuse Problem
  - Increased maintenance costs
  - Lack of tool support
  - Pervasiveness of the "not invented here" syndrome
  - Need to create and maintain a component library
  - Finding and adapting reusable components
- **CSBE Component-Based Software Engineering**
  - CBSE is an approach to software development that **relies on reuse**
- Component Abstractions

    - **Functional Abstractions**
    - **Casual Groupings**
    - **Data Abstractions**
    - **Cluster Abstractions**
    - **System Abstraction**
- Requirements that can be addressed with available components
    - component qualification
    - component adaptation
    - component composition
    - component update
- **Commercial Off-the-Shelf Software (COTS): COTS systems are usually complete applications library the off an applications programming interface (API)**
- Component-Based Development
    - Analysis
    - Architectural design
        - component qualification
        - component adaptation
        - component decomposition
    - Component engineering
    - Testing
    - Iterative component update
- Component reusability should strive to
    - reflect stable domain abstractions
    - hide state representations
    - be independent (low coupling)
    - propagate exceptions via the component interface
- Component Adaptation Techniques
    - **White-box Wrapping**
    - **Grey-box Wrapping**
    - **Black-box Wrapping**
- Kinds of abstraction in S.E
    - **Procedural abstraction**
        - Naming a sequence of instructions
        - Parameterizing a procedure
    - **Data abstraction**
        - Naming a collection of data
        - Data type defined by a set of procedures
    - **Control abstraction**
        - W/o specifying all register/binary-level steps
    - **Performance abstraction**
        - O(N)

- MVC

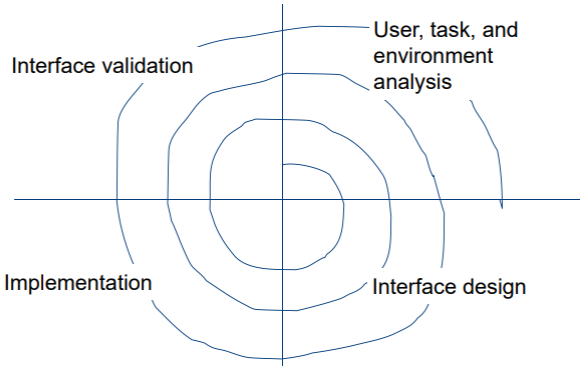



    - View: Little logic; just display info
        - JSPs
    - Model:  Logic related to persistent storage
        - DB system
            - each DB entity maps to a single DAO and a single Bean
        - Beans: placeholders for data related to an entity
            - minimal functionality (only store data
        - DAOs(Database Access Objects): java objects that interact with the DB
    - Controller
        - Handle all logic
        - Everything between Action classes and DAO classes
- Module
    - **Coupling**:  Measure of interconnection **among** modules
    - **Cohesion**:  Measure of interconnection **within** a module
    - **Information hiding**:  Each module should hide a design decision from others
- How to get modularity
    - Reuse a design with good modularity
    - Think about design decisions – hide them
    - Reduce coupling and increase cohesion
    - Eliminate duplication
    - Reduce impact
- Enhancing Reliability
    - Name generalization
    - Operation generalization
    - Exception generalization
    - Component certification
- Component Composition Infrastructure Elements
    - Data exchange model
    - Automation
    - Structured storage
    - Underlying object model
- Framework Classes
    - System infrastructure framework

    - Middleware integration frameworks
    - Enterprise

# Lecture10

- 10 rules of Good UI Design
    - Make Everything the User Needs Readily Accessible
    - Be Consistent
    - Be Clear
    - Give Feedback
    - Use Recognition, Not Recall
    - Choose How People Will Interact First
    - Follow Design Standards
    - Elemental Hierarchy Matters
    - Keep Things Simple
    - Keep Your Users Free & In Control
- Back to Joel: Golden Rules
    - Let the user be in control
    - Reduce the user's memory load
    - Be consistent
- UI Design Process




- Implementation concerns
    - Simplicity
    - Safety
    - Use standard libraries/toolkits
    - Separate UI from application

# Lecture11

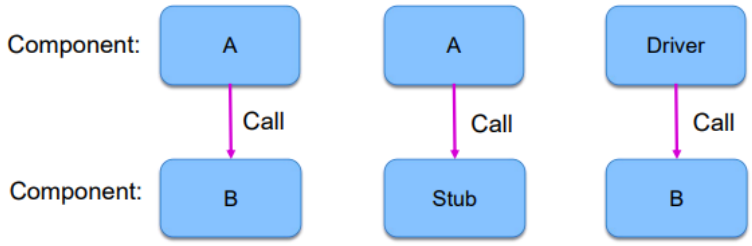
- DevOps = Development + IT Operation
- Continuous Integration
    - a software development practice where members of a team **integrate** their work **frequently**, usually each person integrates **at least daily** - leading to multiple integrations per day. Each integration is verified by an **automated build** (including **test**) to detect integration errors as quickly as possible
    - 10 Principles of Continuous Integration
        - Maintain a code repository – version control

        - Automate the build
        - Make your build self-testing
        - Everyone commits to mainline every day
        - Every commit should build mainline on an integration machine  Keep the build fast
        - Test in a clone of the production environment
        - Make it easy for anyone to get the latest executable
        - Everyone can see what's happening
        - Automate deployment
    - Examples of CI Server

      First CI Server: *CruiseControl*

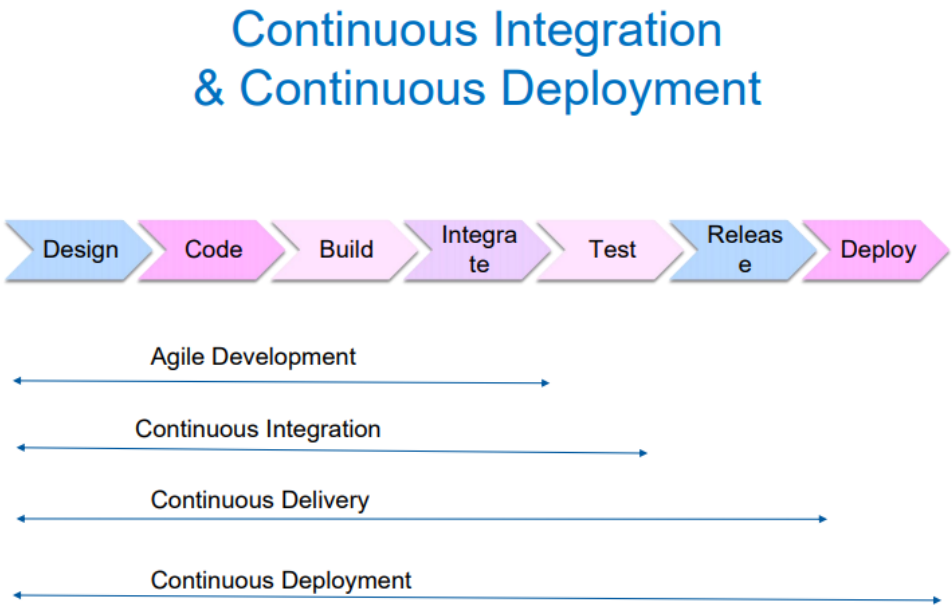
      Jenkins

      Bamboo

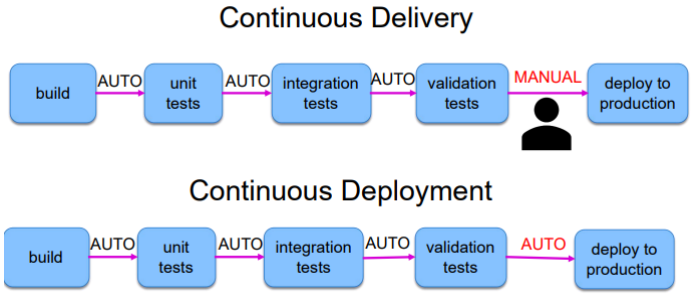
      Travis CI
    - Levels of Software Testing
        - Acceptance Testing
        - System Testing
        - Integration Testing
            - Top-down integration
            - Bottom-up integration
            - "Sandwich" integration
        - Unit Testing
- Stub & Driver
    - Both are used to replace the missing software and simulate the interface between component
    - 

    - Stub: Dummy function gets called by another function
    - Driver: Dummy function to call another function

- Concepts




  - Continuous Delivery
    - The essence of my philosophy to software delivery is to build software so that it is always in a **state** where it could **be put into production**
    - Continuous Delivery = CI(Continuous Integration) + **automated test** suite
    - The key is automated testing
  - Continuous Deployment




  - Continuous Deployment = Continuous Delivery + Automatic Deployment
- How to fix regression bug
  - What has changed?
  - Which changes are wrong?
  - Fix the incorrect changes
  - Check if regression bugs are fixed
- Flaky Test
  - Test which could fail or pass for the same code

# Lecture12

- Security dimension
  - Confidentiality
  - Integrity
  - Availability
- Security levels
  - Infrastructure security

    - Application security
    - Operational security
- Threat type
    - Interception threats
    - Interruption threats
    - Modification threats
    - Fabrication threats