CS304 Review

Lec01

Software processes:

- Agile
  - eXtreme Programming (XP), Scrum…
- Theoretical
  - Waterfall…
- Formal
  - Rational Unified Process (RUP), Cleanroom…
- Distributed, open-source
  - Bazaar…
- …

Default process: XP

- Roles
  - XP: Customer, Developer, Coach
  - Scrum: Pigs (product owner, dev team [3-9 ppl], Scrum master), Chicken (customers and executive management) (http://en.wikipedia.org/wiki/The_Chicken_and_the_Pig)
- Activities
  - XP: Write stories, planning game, test-first, pair programming, continuous integration, refactoring
- Work products
  - XP: User stories, tests, code

What is (not) S.E.?

- Not just software programming
  - Individual vs. team
- Not just a process
  - Field that studies several different processes
- IEEE 610: “The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.”

Lec02

Software configuration management (SCM)

- Four aspects
  - Change control
  - Version control
  - Building
  - Releasing
- Supported by tools
- Requires expertise and oversight
- More important on large projects

SVN: Apache Subversion

Create local copy: `svn checkout <address_to_remote> <name_of_local_dir>` `git checkout <branch_name>`

Commit local changes: `svn commit -m "msg"` `git commit -m "msg"`

Update local copy: `svn up` `git pull upstream master`

Telling svn/git about a new file to track: `svn add <file-name>` `git add <file-name>`

More commands:

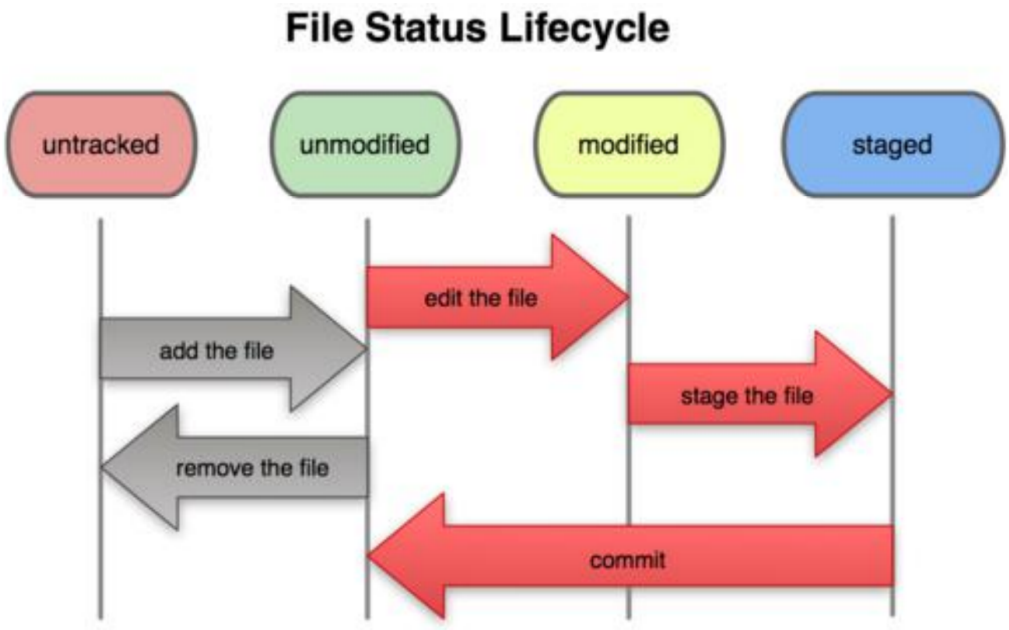
- **svn st:** shows the status of files in the current svn directory
- **svn rm:** removes a file from the set of tracked files (will be removed on the remote server as well)
- **svn mv:** moves a file from one directory to another (or renames if in same directory)
- **svn diff:** diff between two revisions, or diff a file to see uncommitted local changes.

SVN Directory Layout

- The general layout of an svn directory consists of 3 directories:
  - Trunk
    - most up to date current dev
  - Branches
    - releases, bug fixes, experimental
    - Do *not* branch to: support a different hardware, or support a different customer
  - Tags
    - Mark a state of the code (release for e.g.)

Git: Distributed, checksum(first 7 chars). Modify->Stage->Commit

Git file lifecycle



Building tool: make, ant, mvn, Gradle, ...

Software product

- Product = set of components/documents
  - Code
  - Test suites
  - Operation manuals (admins, end-users)
  - Requirements
  - Specifications
  - Design documentation
  - Plans/schedules
- Need to keep track of how you created a product
  - Rules for building the executable
  - Version of code
  - Version of libraries
  - The compiler
  - The operating system
- SCM tool should be able to keep track of all of these (and more)

SCM according to the SEI

- A discipline for controlling the evolution of software systems
- Has many aspects
  - Identification
  - Control
  - Status accounting
  - Audit and review

Various tests

- Smoke test
  - Ensure that the system still runs after you make a change
- Unit test
  - Ensure that a module is not broken after you make a change
- Regression test
  - Ensure that existing code doesn’t get worse as you make other improvements

Summary of SCM

- Four aspects
  - Change control
  - Version control
  - Building
  - Releasing
- Supported by tools
- Requires expertise and oversight
- More important on large projects

Lec03

WATERFALL PROCESS ACTIVITIES

- Requirements – what software should do
- Design – structure code into modules; architecture
- Implementation – hack code
- Integration – put modules together
- Testing – check if code works
- Maintenance – keep making changes

Often merged together as Verification

XP: SOME KEY PRACTICES

- Planning game for requirements
- Test-driven development for design and testing
- Refactoring for design
- Pair programming for development
- Continuous integration for integration

FORMAT OF A USER STORY

- Title: 2-3 words
- Acceptance test (unique identifier)
- Priority: 1-2-3 (1 most important)
- Story points (can mean #days of ideal development time, i.e., no distractions or working on other things)
- Description: 1-2 sentences (a single step towards achieving the goal)

Title: Enter Player Info		
Acceptance Test: enterPlayerInfo1	Priority: 1	Story Points: 1
Right after the game starts, the Player Information dialog will prompt the players to enter the number of players (between 2 and 8). Each player will then be prompted for their name, which may not be an empty string. If Cancel is pressed the game exits gracefully.		

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.
  - Related to how hard it is and how much of it there is
  - Not related to amount of time or the number of people
  - Unitless, but numerically-meaningful
- Ideal time: the amount of time "something" takes when stripped of all peripheral activities
  - Example: American football game = 60 minutes
- Elapsed time: the amount of time that passes on the clock to do "something"
  - Example: American football game = 3 hours
- Velocity: measure of a team's rate of progress

TERMINOLOGY: MISTAKE, FAULT/BUG, FAILURE, ERROR

Programmer makes a mistake

Running the test inputs …

Fault (defect, bug) appears in the program

Fault remains undetected during testing

Program failure occurs during execution (program behaves unexpectedly)

Error: difference between *computed, observed, or measured value or condition* and *true, specified, or theoretically correct value or condition*

A Concrete Example

Fault: Should start searching at 0, not 1

```
public static int numZero (int [ ] arr)
{  // Effects: If arr is null throw NullPo                ion
   // else return the number of occurrence               rr
   int count = 0;
   for (int i = 1; i < arr.length; i++)
   {
      if (arr [ i ] ==
      {
         count++;
      }
   }
   return count;
}
```

Test 1
[2, 7, 0]
Expected: 1
Actual: 1

Error: i is 1, not 0, on the first iteration
Failure: none

Test 2
[0, 2, 7]
Expected: 1
Actual: 0

Error: i is 1, not 0
Error propagates to the variable count
Failure: count is 0 at the return statement

JUNIT BASICS

- Open source (junit.org) Java testing framework 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
- JUnit is widely used in industry
- Can be used as stand alone Java programs (from command line) or from an IDE such as IntelliJ or Eclipse

ADVANCED TOPICS IN JUNIT

- Assertion patterns
  - How to decide if your test passes
  - State testing vs. interaction testing patterns
- Parameterized JUnit tests
  - How to describe and run very similar tests
- JUnit theories
  - Applying the contract model to testing
  - AAA model: Assume, Act, Assert
  - Very powerful approach
    - But also still a work in progress

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
  - Very Different Analysis Compared to State Testing
    - Resource: http://martinfowler.com/articles/mocksArentStubs.html

PARAMETERIZED TESTS

- Problem: Testing a function with similar values
  - How to avoid test code bloat?
- Simple example: Adding two numbers
  - Adding given pair of numbers is just like adding any other pair
  - You really only want to write one test
- Parameterized unit tests call constructor for each logical set of data values
  - Same tests are then run on each set of data values
  - List of data values identified with `@Parameters` annotation

JUNIT THEORIES

- These Are Unit Tests With Actual Parameters
  - So Far, We've Only Seen Parameterless Test Methods
- Contract Model: Assume, Act, Assert
  - Assumptions (Preconditions) Limit Values Appropriately
  - Action Performs Activity Under Scrutiny
  - Assertions (Postconditions) Check Result

```
@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);
}
```

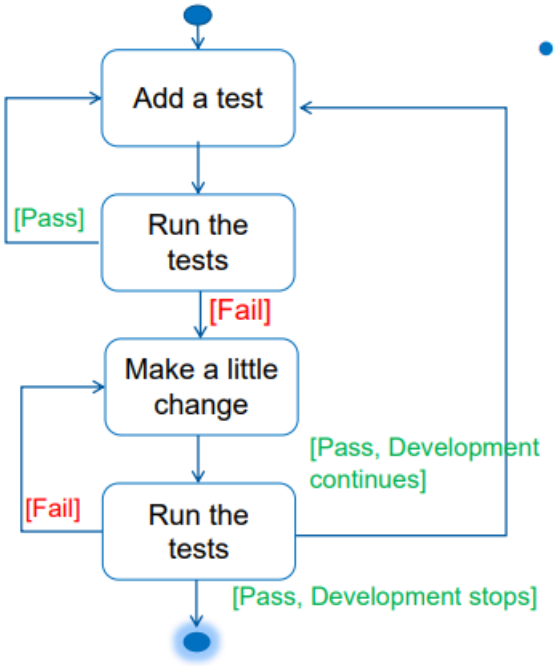
SUMMARY

- The only way to make testing efficient as well as effective is to automate as much as possible
- JUnit provides a very simple way to automate our unit tests
- It is no "silver bullet" however … it does not solve the hard problem of testing :

What test values to use ?

- **This is test design … the purpose of test criteria**

Steps in Test Driven Development (TDD)



- The iterative process
  - 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.

Lec05

Each test should be independent of each other

Any given behavior should be specified in one and only one test.

`assertEquals(expected, actual)`

Which example has better tests?

Example 1 is better!

➢ **Each test should be independent of each other**

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 ());
}
```

Which example has better tests?

Example 2 is better!

- **Any given behaviour should be specified in one and only one test.**

Example 1

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

Multiple assertions are bad because after one assertion fail, execution stops

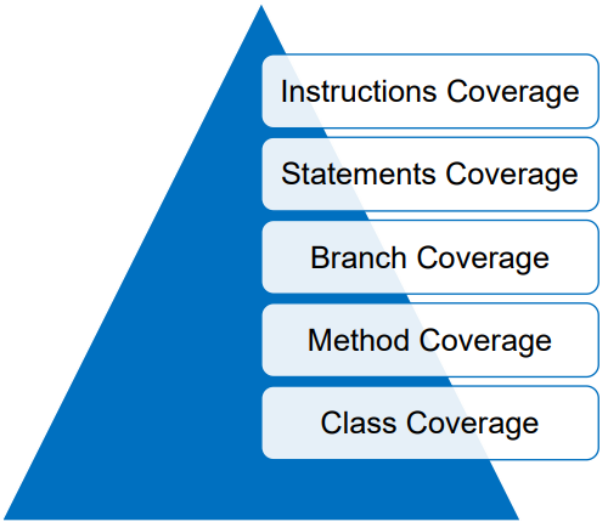
Example 2

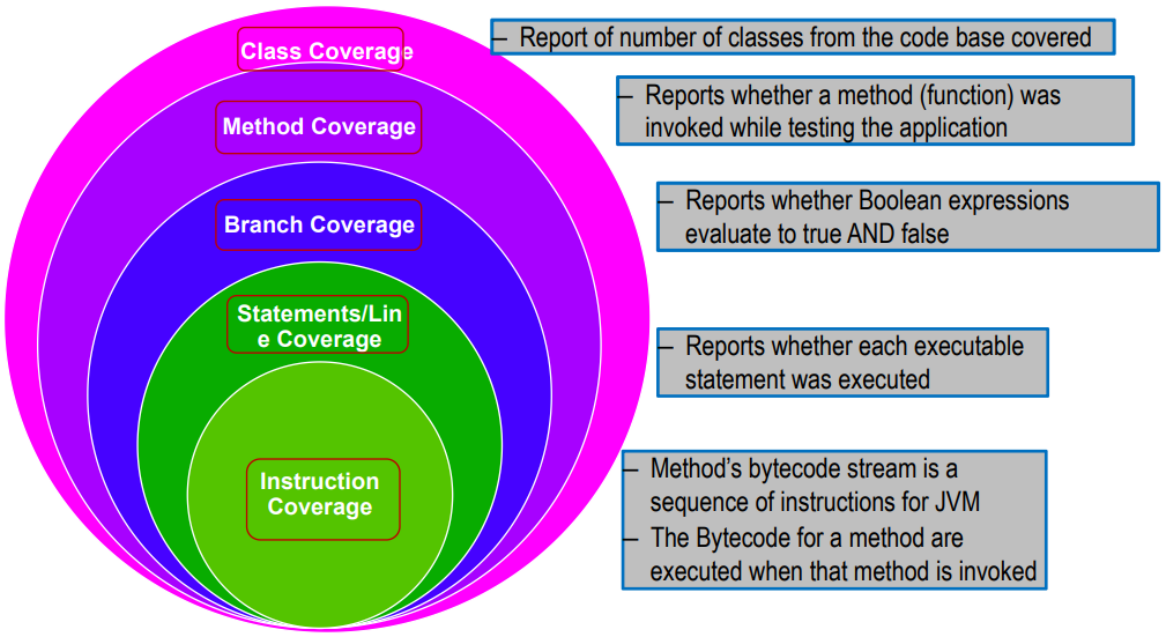
```
@Test
public void emptyTest() {
  MyStack s = new MyStack ();
  assertEquals (0, s.size ());
}

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

Code coverage:

Coverage Criteria

- To measure what percentage of code has been exercised by a test suite, one or more coverage criteria are used





$$Branch\ Coverage = \frac{Number\ of\ Executed\ Branches}{Total\ Number\ of\ Branches}$$

5 code coverage criteria

To measure the lines of code that are actually exercised by test runs, various criteria are taken into consideration. We have outlined below a few critical coverage criteria that companies use.

1. **Function Coverage** – The functions in the source code that are called and executed at least once.
2. **Statement Coverage** – The number of statements that have been successfully validated in the source code.
3. **Path Coverage** – The flows containing a sequence of controls and conditions that have worked well at least once.
4. **Branch or Decision Coverage** – The decision control structures (loops, for example) that have executed fine.
5. **Condition Coverage** – The Boolean expressions that are validated and that executes both TRUE and FALSE as per the test runs.

Jacoco: `java -jar- -javaagent:/jacocoagent.jar=destfile=/jacoco.exec`

Evosuite: EvoSuite uses evolutionary algorithm to generate and optimize whole test suites towards satisfying a coverage criterion.

Lec06

Phrases in Maven

Maven Phases

Common default lifecycle phases:

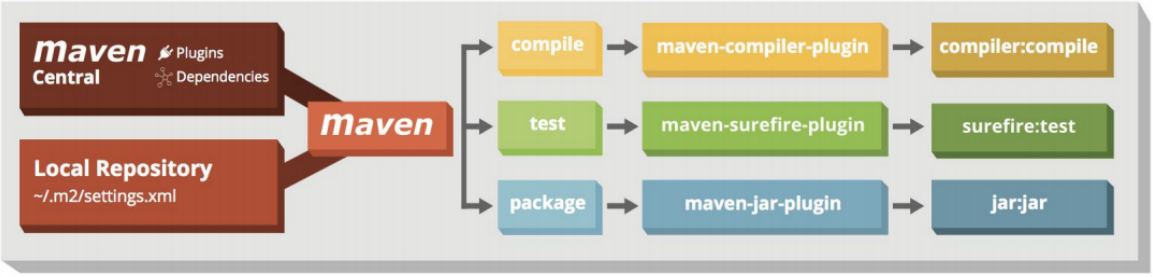
- **validate**: validate the project is correct and all necessary information is available
- **compile编译**: compile the source code of the project
- **Test测试**: test the compiled source code using a suitable unit testing framework. These tests should not require the code be packaged or deployed
- **Package打包**: take the compiled code and package it in its distributable format, such as a JAR.
- **integration-test**: process and deploy the package if necessary into an environment where integration tests can be run
- **verify**: run any checks to verify the package is valid and meets quality criteria
- **Install安装**: install the package into the local repository, for use as a dependency in other projects locally
- **deploy**: done in an integration or release environment, copies the final package to the remote repository for sharing with other developers and projects.

There are two other Maven lifecycles of note beyond the *default* list above. They are

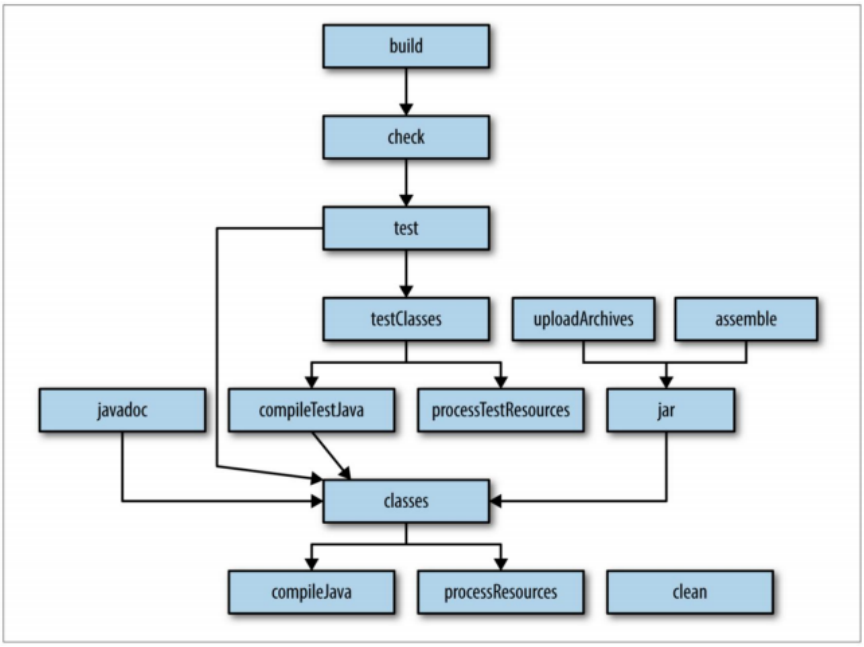
- **clean**: cleans up artifacts created by prior builds
- **site**: generates site documentation for this project

Example command: mvn test

Main commands in maven



Gradle: The build lifecycle



TECHNICAL METRICS

- Size of code
  - Number of files
  - Number of classes
  - Number of processes
- Complexity of code
  - Dependencies / Coupling / Cohesion
  - Depth of nesting
  - Cyclomatic complexity

MEASURING SIZE OF SYSTEM

- Lines of code (Source Lines of Code - SLOC)
- Number of classes, functions, files, etc.
- Function Points

CYCLOMATIC COMPLEXITY (1)

- A measure of logical complexity
- Tells how many tests are needed to execute every statement of program

=Number of branches (`if`, `while`, `for`) + 1

COUPLING AND COHESION (1)

- Coupling - dependences among modules
- Cohesion - dependences within modules
- Dependences
  - Call methods, refer to class, share variable
- Coupling - bad
- Cohesion - good

DHAMA'S 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
)

0.5 is low coupling, 0.001 is high coupling

MARTIN'S COUPLING METRIC

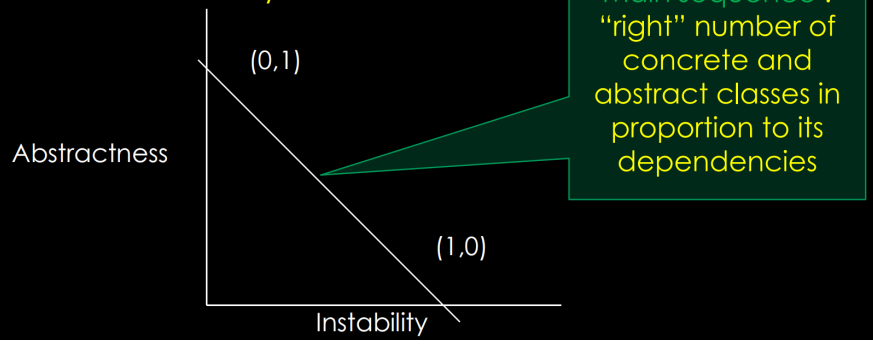
- Ca : Afferent coupling: the number of classes outside this module that depend on classes inside this module
- Ce : Efferent coupling: the number of classes inside this module that depend on classes outside this module

Instability = Ce / (Ca + Ce)

ABSTRACTNESS

Abstractness = $A = \frac{T_{abstract}}{T_{abstract} + T_{concrete}}$

(number of abstract classes in module / number of classes in module)



Lec07

LIST OF METRICS

- Weighted Methods Per Class (WMC)
- Depth of Inheritance Tree (DIT)
- Number of Children (NOC)
- Coupling between Object Classes (CBO)
- Response for a Class (RFC)
- Lack of Cohesion in Methods (LCOM)

SUMMARY: METRICS (IN GENERAL)

- Non-technical: about process
- Technical: about product
  - Size, complexity (cyclomatic, function points)
- How to use metrics
  - Prioritize work - compare modules in a system version, compare system versions over time
  - Measure programmer productivity
  - Make plans based on predicted effort

- Discovering design of an artifact
  - From lower level to higher level; for example:
    - Given binary, discover source code
    - Given code, discover specification & design rationale
- Layman: trying to understand how the system works
- When are you done?
  - Learn enough to:
    - Change it, or
    - Replace it, or
    - Write a book about it

SOME TERMINOLOGY

- Forward engineering
  - From requirements to design to code
- Reverse engineering
  - From code to design, maybe to requirements
- Reengineering
  - From old code to new code via some design

Lec09

REVERSE ENGINEERING ACTIVITIES

- Purpose
  - Learn the system from the system
- Activities
  - Read documentation
  - Talk to people
  - Look at the code
  - Work with the system
  - Write documentation

SUMMARY

- Reverse engineering is analyzing an existing system
- Instead of inventing design ideas, discover them
- Tools useful, but people are more important and hard work is essential

Testing:

- Dynamic analysis()
- static analysis(no dynamic execution, detect possible defects in an early stage).
  - Checkstyle: focus on java coding style and standards.

Checkstyle

- Focus on Java coding style and standards.
  - whitespace and indentation
  - variable names
  - Javadoc commenting
  - code complexity
    - number of statements per method
    - levels of nested ifs/loops
    - lines, methods, fields, etc. per class
  - proper usage
    - import statements
    - regular expressions
    - exceptions
    - I/O
    - thread usage, ...

  - PMD: scan source code and looks for potential problems

PMD

PMD scans Java source code and looks for potential problems like:

- Possible bugs - empty try/catch/finally/switch statements
- Dead code - unused local variables, parameters and private methods
- Suboptimal code - wasteful String/StringBuffer usage
- Overcomplicated expressions - unnecessary if statements, for loops that could be while loops
- Duplicate code - copied/pasted code means copied/pasted bugs

PMD Basic Rules

- **EmptyCatchBlock:** Empty Catch Block finds instances where an exception is caught, but nothing is done. In most circumstances, this swallows an exception which should either be acted on or reported.
- **EmptyIfStmt:** Empty If Statement finds instances where a condition is checked but nothing is done about it.
- **EmptyWhileStmt:** Empty While Statement finds all instances where a while statement does nothing. If it is a timing loop, then you should use Thread.sleep() for it; if it's a while loop that does a lot in the exit expression, rewrite it to make it clearer.
- **EmptyTryBlock:** Avoid empty try blocks - what's the point?
- **EmptyFinallyBlock:** Avoid empty finally blocks - these can be deleted.
- **EmptySwitchStatements:** Avoid empty switch statements.
- **JumbledIncrementer:** Avoid jumbled loop incrementers - it's usually a mistake, and it's confusing even if it's what's intended.
- **ForLoopShouldBeWhileLoop:** Some for loops can be simplified to while loops - this makes them more concise.

- Findbugs

What is FindBugs?

- Result of a research project at the University of Maryland
- Based on the concept of *bug patterns*. A bug pattern is a code idiom that is often an error.
  - Difficult language features
  - Misunderstood API methods
  - Misunderstood invariants when code is modified during maintenance
  - Garden variety mistakes: typos, use of the wrong boolean operator
- FindBugs uses *static analysis* to inspect Java bytecode for occurrences of bug patterns.
- Static analysis means that FindBugs can find bugs by simply inspecting a program's code: executing the program is not necessary.
- FindBugs works by analyzing Java bytecode (compiled class files), so you don't even need the program's source code to use it.
- FindBugs can report ***false warnings***, not indicate real errors. In practice, the rate of false warnings reported by FindBugs is less than 50%.

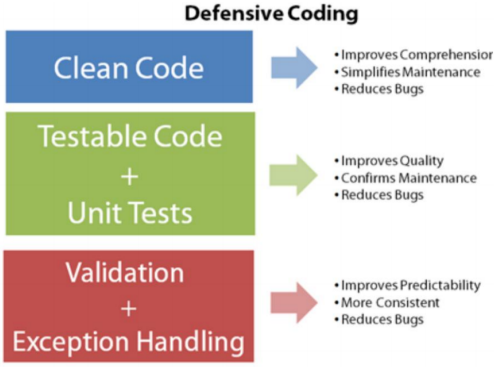
- Defensive Programming: no warning by Static Analysis

Lec10

Tools check for coding standard: PIT, CheckStyle, FindBugs, PMD

What is Defensive Programming?

- **Defensive programming** is a form of defensive design intended to ensure the continuing function of a piece of software under unforeseen circumstances.
- Often used when **high availability, safety or security** is needed.



Pic From: https://blogs.msmvps.com/deborahk/what-is-defensive-coding/

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 interfaces
- Error checking codes in data (e.g., checksum or hash)

Fault Tolerance

Basic Techniques:

- After error continue with next transaction (e.g., drop packet)
- Timers and timeout in networked systems
- User break options (e.g., force quit, cancel)
- Error correcting codes in data
- Bad block tables on disk drives
- Forward and backward pointers in databases

Report all errors for quality control

Backward Recovery:

- Record system state at specific events (**checkpoints**). After failure, recreate state at last checkpoint.
- **Backup** of files
- Combine checkpoints with system **log** (**audit trail** of transactions) that allows transactions from last checkpoint to be repeated automatically.
- ***Test the restore software!***

Techniques: Barriers

Place barriers that separate parts of a complex system:

- Isolate components, e.g., do not connect a computer to a network
- Firewalls
- Require authentication to access certain systems or parts of systems

Every barrier imposes restrictions on permitted uses of the system

Barriers are most effective when the system can be divided into subsystems with simple boundaries

Techniques: Authentication & Authorization

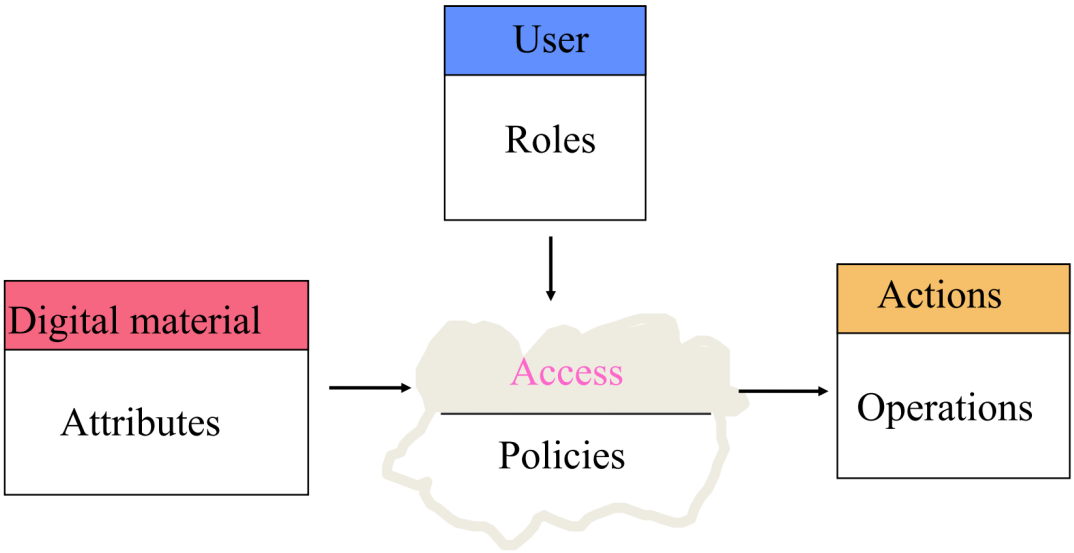
Authentication establishes the identity of an agent:

- What the agent knows (e.g., password)
- What the agent possess (e.g., smart card)
- Where does the agent have access to (e.g., controller)
- What are the physical properties of the agent (e.g., fingerprint)

Authorization establishes what an authenticated agent may do:

- Access control lists
- Group membership

Example: An Access Model for Digital Content



What are *Javadoc* Comments?

```
/*
* Returns a synchronized map backed by the given map.
...
* @param map the map to synchronize, must not be null
* @return a synchronized map backed by the given map
* @throws IllegalArgumentException if the map is null
*/
static <K,V> Map<K,V> synchronizedMap(Map<K,V> map)
```

Overall Description

Block Tags

- @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 /**

/*
```

```
@auther
@version
*/
    //TODO //FIXME //XXX
```

Javadoc placement

- javadoc comments begin with /** and end with */
  - In a javadoc comment, a * at the beginning of the line is not part of the comment text
- javadoc comments must be *immediately before:*
  - a class (plain, inner, abstract, or enum)
  - an interface
  - a constructor
  - a method
  - a field (instance or static)
- Anywhere else, javadoc comments will be *ignored!*
  - Plus, they look silly

Types of Software Reuse

- Application System Reuse
  - reusing an entire application by incorporation of one application inside another (COTS reuse)
  - development of application families (e.g. MS Office)
- Component Reuse
  - components (e.g. subsystems or single objects) of one application reused in another application
- Function Reuse
  - reusing software components that implement a single well-defined function

Benefits of Reuse

- Increased Reliability
  - components already exercised in working systems
- Reduced Process Risk
  - less uncertainty in development costs
- Effective Use of Specialists
  - reuse components instead of people
- Standards Compliance
  - embed standards in reusable components
- Accelerated Development
  - avoid custom development and speed up delivery

Component-Based Software Engineering

- CBSE is an approach to software development that relies on reuse
- CBSE emerged from the failure of object-oriented development to support reuse effectively
- Objects (classes) are too specific and too detailed to support design for reuse work
- Components are more abstract than classes and can be considered to be stand-alone service providers

Component Abstractions

- Functional Abstractions
  - component implements a single function (e.g. *ln*)
- Casual Groupings
  - component is part of a loosely related entities like declarations and functions
- Data Abstractions
  - abstract data types or objects
- Cluster Abstractions
  - component from group of cooperating objects
- System Abstraction
  - component is a self-contained system

Modularity summary

- Modularity is key to managing complexity
- Modularity is about managing dependencies
- Modularity should be based on hiding design decisions
- No perfect answer
  - Trade-offs
  - Iteration

Enhancing Reliability

- Name generalization
  - names modified to use domain independent language
- Operation generalization
  - operations added to provide extra functionality
  - domain specific operations may be removed
- Exception generalization
  - application specific exceptions removed
  - exception management added to increase robustness
- Component certification
  - component warranted correct and reliable for reuse

10 rules of Good UI Design

1. Make Everything the User Needs Readily Accessible
2. Be Consistent
3. Be Clear
4. Give Feedback
5. Use Recognition, Not Recall
6. Choose How People Will Interact First
7. Follow Design Standards
8. Elemental Hierarchy Matters
9. Keep Things Simple
10. Keep Your Users Free & In Control

Summary of Mobile UI Design

- Mobile UI design faces new challenges
  - Small screens
  - Fat fingers
  - Poor text entry
- Simplify
  - Follow design patterns
  - Use touch gestures where possible

Principle

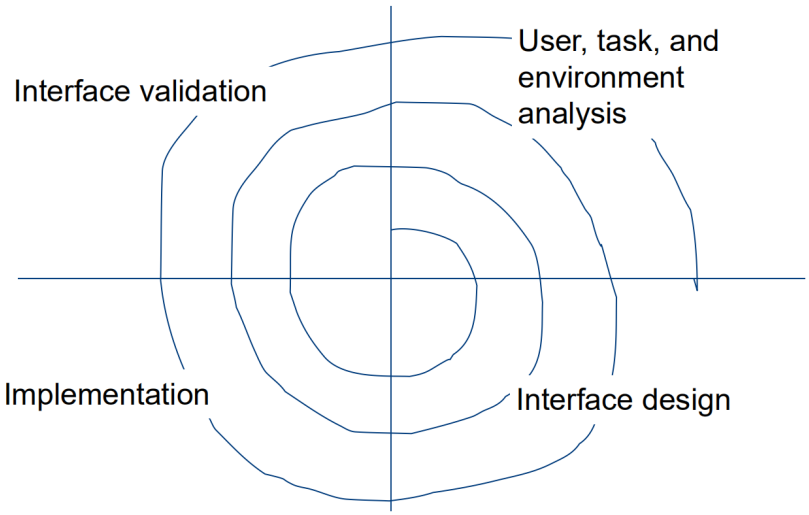
- UI design is more like film-making than bridge-building
  - About communication
  - Requires understanding audience
  - Requires specialized skills
  - Requires iteration

Lec12

Common techniques

- Menus with keyboard shortcuts
- Dialog boxes
- Tabs
- Toolbar

The UI Design Process



UI metrics

- Size of written specification
- Number of user tasks
- Number of actions per task
- Number of system states
- Number of help messages

CEIP data

- ☐Usage
  - How software is used
  - Examples: general feature usage, commands on Ribbon, actions taken in wizards, etc.
- ☐Reliability and performance
  - Whether software performs as expected
  - Examples: assertions for logical inconsistency, measuring execution speed, etc.
- ☐Hardware/software configuration
  - Providing context for data interpretation
  - Example: long document loading time only on machines with low RAM or a particular processor speed?

Implementation concerns

- ☐Simplicity
- ☐Safety
- ☐Use standard libraries/toolkits
- ☐Separate UI from application
  - Model-View-Controller (MVC)
  - Three-tier: presentation, application, data

1. Simplicity

- ☐Tradeoff between number of features and simplicity
- ☐Don't compromise usability for function
- ☐A well-designed interface fades into the background
- ☐Basic functions should be obvious
- ☐Advanced functions can be hidden

2. Safety

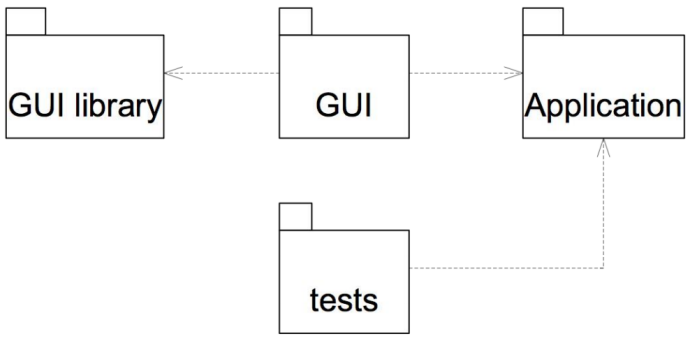
- Make actions predictable and reversible
- Each action does one thing
- Effects are visible
  - User should be able to tell whether operation has been performed
- Undo

3. Use standard libraries

- Don’t build your own!
  - If necessary, add to it, but try to use standard parts instead of building your own
- Provide familiar controls
- Provide consistency
- Reduce cost of implementation
- Library designers probably better UI designers than you are

4. Separate UI and application

- UI and application change independently
- UI and application built by different people



Benefits

- Write automatic tests for application objects, not for UI
- People who write HTML don't need to know how to program well
- Programmers don't need to be good UI designers

Summary

- UI design is hard
  - Must understand users
  - Must understand problems
  - Must understand technology
  - Must understand how to evaluate
- UI design is important
  - UI is what the users see
  - UI can "make it or break it" for software

Lec13

*"**Continuous Integration** is 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."*

Benefits of Continuous Integration:

- Reduce integration problems
- Allows team to develop cohesive software more rapidly

Martin Fowler

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

CI Server:

First CI Server: *CruiseControl*

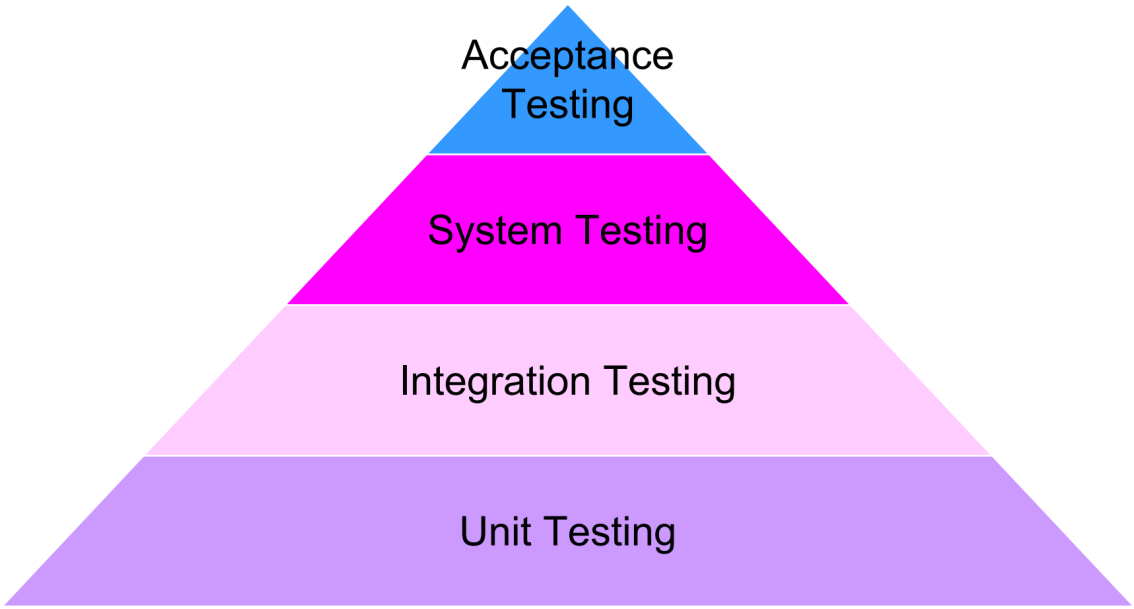
Jenkins

An extendable open source continuous integration server

Bamboo

Travis CI

Levels of Software Testing



Big-Bang Integration Testing, Incremental Integration Testing, Top-Down Integration, Bottom-Up Integration, "Sandwich" Integration

Continuous Integration != Continuous Delivery: CI != CD

- $Continous\ Delivery\ =\ CI\ +\ automated\ test\ suite$
- Not every change is a release
  - Manual trigger
  - Trigger on a key file (version)
  - Tag releases!

- *CD* – The key is *automated testing.*

Continuous Deployment

- $Continous\ Deployment\ =\ CD + Automatic\ Deployment$
- Every change that passes the automated tests is deployed to production automatically.
- Deployment Schedule:
  - Release when a feature is complete
  - Release every day
- $Continous\ Deployment\ =\ CD + Automatic\ Deployment$

Deployment strategies

Strategy 1: Zero-downtime deployment

1. Deploy version 1 of your service
2. Migrate your database to a new version
3. Deploy version 2 of your service in parallel to the version 1
4. If version 2 works fine, bring down version 1
5. Deployment Complete!

Strategy 2: Blue-green deployment

1. Maintain two copies of your production environment ("blue" and "green")
2. Route all traffic to the blue environment by mapping production URLs to it
3. Deploy and test any changes to the application in the green environment
4. "Flip the switch": Map URLs onto green & unmap them from blue.

Zero-downtime & Blue-green Deployment

- Advantage:
  - No outage/shut down
  - User can still use the application without downtime
- Disadvantages:
  - Needs to maintain 2 copies
  - Double the efforts required to support multiple copies
  - Migration of database may not be backward compatible

Safer Strategy: Shut down→Migrate → Deploy

Lec14

Security dimensions

✧ *Confidentiality*

- Information in a system may be disclosed or made accessible to people or programs that are not authorized to have access to that information.

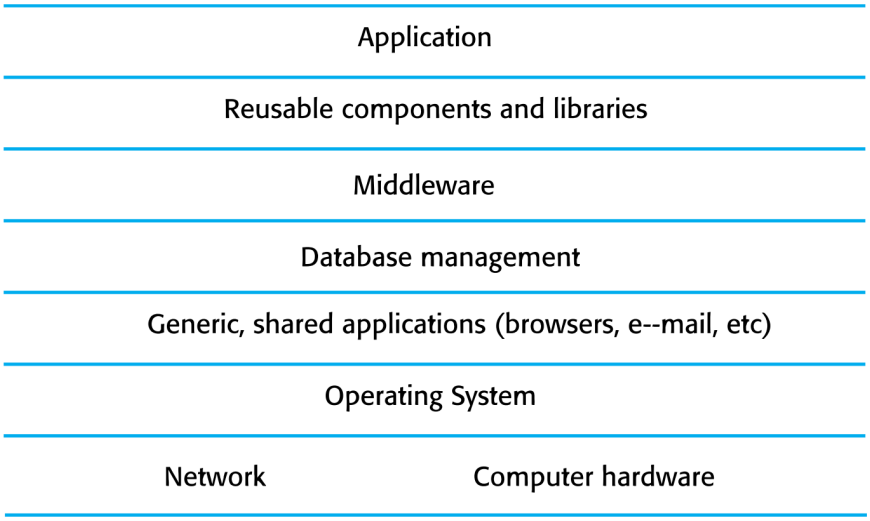
✧ *Integrity*

- Information in a system may be damaged or corrupted making it unusual or unreliable.

✧ *Availability*

- Access to a system or its data that is normally available may not be possible.

System layers where security may be compromised



Appendix

Extreme Programming(XP)

Software configuration management (SCM)

Version Control System(VCS)

Test Driven Development(TDD)

Weighted Methods Per Class (WMC)

Depth of Inheritance Tree (DIT)

Number of Children (NOC)

Coupling between Object Classes (CBO)

Response for a Class (RFC)

Lack of Cohesion in Methods (LCOM)

Component-Based Software Engineering(CBSE)

Commercial Off-the-Shelf Software (COTS)

Experience Improvement Program (CEIP)

Continuous Integration(CI)

Continuous Delivery(CD)

Regression Test Selection(RTS)