# **CSCB07 –** Software Design

Fall 2016 Finals Notes – Abbas Attarwala

**Introduction to Subversion**

Version control: Tracks files over a period of time, tracks what happens by what developer. We know who wrote what, when, and why
Revision: Keeps track of changing states of files over merges and contributions of multiple developers
In SVN, there are two components: Client and Server
svn checkout [web address] -- username xy@gmail.com
svn add names.txt
svn commit –m "adding a new file" names.txt (or remove filename for all)
svn update
svn status
_ - no mods A – scheduled for addition D – scheduled for deletion M – modified ? – Not under vcontrol
C – conflict
svn log
CONFLICT RESOLUTION
svn revert
cp somefile.mine.java somefile.java
svn resolved
svn commit –m "I accepted my changes"

**Java Fundamentals and OOP**

| | | | | | |
|---|---|---|---|---|---|
| byte: | 8 | bit | signed | integer | |
| short: | 16 | bit | signed | integer | |
| long: | 64 | bit | signed | integer | |
| float: | 32 | bit | floating | point | values. |
| double: | 64 | bit | floating | point | values. |
| boolean | 1 | bit | | | |
| char: | 16 | bits | unsigned | integer. | |

Strings – are immutable, value cannot be changed e.g. str[4] = 'c';
Static fields – unchanging through different objects, belongs to a class not an object

Factory methods -> abstraction of constructors
e.g. instead of createComplexNumber, we make that private, and make createCartesianCN, and createDecimalCN public instead

**Inheritance, Composition, and CRC cards**

Memory has 2 parts in Java: Stack and Heap.
Stack is stored in RAM, variables and local data are stored here.
Heap is stored in the RAM, objects are allocated on the heap, and must be freed manually
e.g. Student Jack = new Student();
^Stack: Student ref ^Heap: Student Obj

Inheritance – We can create a general class that defines traits common to a set of related items (super class) to subclasses (is-a)
Extends keyword
Method overriding -> redefine the method from a super class
Method overloading -> when methods with the same name have different parameters

CRC Cards -> Class, Responsibility, Collaboration
Class
Superclasses
Subclasses
Responsibility
Some behavior for which this class is responsible for
Collaboration
Processes where several objects cooperate to do high-level tasks

**Software Design Principles**

Single Responsibility Principle -> Every class should have a single responsibility, entirely encapsulated by the class
toString() -> Always override, otherwise we get a pointer to object, e.g. object@163A91

Validation -> Testing specification, does it meet the user's needs, is it the right thing?
Building the right thing
Verification -> Does the software work as intended, to meet the specification?
Building the thing right

Waterfall Model –
Requirements
Determine what needs to be built
Formalize what needs to be built
Design

Figuring out how the requirements can be met
Implementation
Make it
Verification
Did we make it according to standards?
Maintenance
Fixing/Adding/Changing features

Advantages: Easy to use, good for small projects, cost effective
Disadvantages: Testing is limited, high amounts of risk, inflexible, and poor model for advanced OOP
Doesn't work when requirements change, or not sure what market needs

Iterative Development Model –
Code > Feedback cycle, until done. Requirements always evolve, so we need to always iterate and rework systems

Incremental Development Model –
Build as much as you need, never over-engineer or add flexibility until need arises

Spiral Model –
Determine objectives, alternatives, and constraints
Risk analysis and evaluation of alternatives
Execution of development
Planning for next phase

Agile Development Methods –
Build & Release continuously
User Stories -> describe situations in which the software must be used
Scrum -> Iterative, incremental methodology
Product Owner (customer)
Team (group)
Scrum master (maintains scrums, removes obstacles, facilitates communications)
Product Backlog (Total tasks)
Sprint Backlog (Tasks per sprint)
Daily Scrum Meetings (What was accomplished, what needs to be done, anything blocked?)

**Java Memory Model**

Stack -> every time a function is called, added to the TOP. It's a LIFO structure!
Heap -> Anytime, anywhere access. No order to remove/get

**Unit Testing**

Test Suites -> classes of tests targeting a specific class or function

```
import org.junit.*;
import static org.junit.Assert.*; // note static import
public class MyProgramTest{
          @Begin
          Void setup()
          @Test
          Void testMax()
          @After
          Void teardown()
}
```

Override the equals method -> Must make sure that is Reflexive (x.equals(x)), Symmetric (x.eq(y) ➔ y.eq(x)), Transitive (x.eq(y), y.eq(z) ➔ x.eq(z)) and Consistent, multiple invocations must return consistent results

**Polymorphism, Interfaces, Liskov, and Singleton**

Polymorphism -> Many forms. Base class that can be extended to make something more reasonable
IS-A relationship
Function binding -> Mapping function call to function implementation (static, compiletime vs dynamic runtime bindings)
Static uses type of reference (e.g. draw(obj) vs draw(car))
Dynamic uses class of object (e.g. car.draw() vs superclass's vehicle.draw())
Abstract Classes -> placeholder in a class hierarchy that represents a generic concept
Contains abstract methods, e.g. abstract public void move(); (no body)
Non-abstract children must define the body
Cannot be instantiated
Interfaces -> Something things can implement. Classes can only extend one, but can implement many (e.g. implement drawable)

```
Interface Pest{
        Void beannoying();
}
Class Housefly implements Pest extends NPC
```

Interfaces are a collection of constants and abstract methods (since all methods in interfaces are abstract, we don't need to state)

Singleton Design Pattern -> e.g. does a OS have multiple filesystems? NO! Its only one! We can use factory methods to define a constructor, so we can only have max=1

Liskov Substitution Principle -> Don't make abstractions not work. Anything that points to a base class should be able to use a base class. Subclasses must display the same behavior without changing the base classes' behavior
Instead, we can do Square HAS-A rectangle, rather than Square IS-A rectangle

**Exceptions and Collections**

Compile time errors -> Errors caught by the compiler: Mainly syntax, like missing semicolons or keywords
Runtime errors -> Errors caught when program is running: File cannot open, insufficient memory, index out of bounds,
Exceptions make it so that spaghetti code isn't common – e.g. not 7 lines of error detection
Exception IS-A Throwable
    Superclass Throwable, can be thrown by 'throw' in Java, returned from a method
    Can be caught by an exception handler in calling method

```
Public float getGoalsPerGame() throws NotEnoughGamesPlayed{
    Throw newNotEnoughGamesPlayed("Insufficient games", MIN_GAMES);
}
Class NotEnoughGamesPlayed extends Exception{
    Public notenoughgamesplayed(String message, int gamesneeded){
        Super(message);
        numberofGamesNeede=gamesNeeded;
    }
}
```

The Catch block of the try-catch handles the exceptions

Collections -> import java.util, has stuff like ArrayList, Set, etc.
    Set -> Iterator, to traverse through items in the set
        Iterator.hasnext(), Iterator.next(), Iterator.remove()
    Iterator e = teamSet.iterator();
    While(e.hasnext()) … etc.
Generics -> We can now define the object type of a Set or etc. E.g. Set<E>, or Set<String>

TreeSet -> Industrial level binary search trees
Comparator -> Interface for compare, have to declare the compare interface, can be generic

**Iterator and Builder Design Patterns**

Iterator Pattern allows traversal of elements of a collection without exposing underlying implementation
e.g. don't use array vs arraylist, instead just create an iterator

```
dinnerMenuIterator(){
    public dinnerMenuIterator(MenuItems[] items){}
    public Object next()
    public Boolean hasNext()
}
```

Nested classes -> Logically ordering classes that are only used in one place (e.g. DinnerMenuIterator)
    Static: Static nested classes  Nonstatic: Inner classes
    e.g. OuterClass.StaticNestedClass nestedOjb = new OuterClass.StaticNestedClass();
Builder Design Pattern: When factory/constructors have limitations due to lots of arguments
Bad → Create many constructors -> how will we know what food(230,30,30) what argument is which? Even all getters is bad!
Good → Use the builder design pattern NutritionFacts Cola = new NutritionFacts.Bulder(240, 8).calories(100).sodium(35)… etc

1. Make the constructor private
2. Declare a public static class Builder inner class
3. Have all relevant variables (e.g. serving size, serving, fat, etc.)
4. Make builders constructor the mandatory arguments
5. Do methods like fat(int val){ fat = val; return this; );
6. Public NutritionFacts build()
7. Set the constructor to private NutritionFacts(Builder b)
8. Set NutrituonFacts e.g. servingSize = b.servingSize();

Calling it
NutritionFacts Cola = new NutritionFacts.Builder(240, 8).calories(200).fat(20).build();

**Junit Testing Part II – Test Driven Development**

Developer writes test suites that fails, to outline an improvement or new function
Produces the MVP for the test to pass, the refactors to be acceptable standards
Stubs -> Code bodies that aren't anything yet – just to satisfy the necessity that the test has to have to run

Mock Objects -> objects that reflect the behavior of other objects, except are designed for unit tests (e.g. live page may update and invalidate UnitTests, but that doesn't mean code is wrong!) Instead, test with Mock Objects so that it works (validity)

**Refactoring**

Restructuring the code so its easier to maintain and modify
Anytime you can improve design, or see bad design in code
Make Change -> Unit Test -> Next
    Encapsulation
    Extract Functionality
    Extract Subclass
    Insert Exceptions

**Regular Expressions**

Pattern p = Pattern.compile("regex"); (regex can be any regex e.g. "[a-z]+")
Matcher m = p.matcher("This is a test string!);

REGEX:

“abc” exactly this sequence
“[abc]” any one of a,b,c
“[^abc]” not a,b,c
“[a-z]” any of a to z
“[a-zA-Z0-9]” any letter or digit
. any one character
\d digit
\D nondigit
\s whitespace
\S non whitespace
\w
\W non-word
\b word bountry
^beginning of line $end of line \B not a word boundry

? once or not at all
* 0 or more times
+ 1 or more times
X{n} exactly n times
X{n, } n or more times
X {n, m} between n and m number of times

---

m.matches() if everything matches
m.lookingAt() if pattern matches start of string
m.find() if pattern matches any part of string returns true until last occurrence, then goes back to beginning
m.start()/m.end() finds the index of the last character/first character matched +1
m.group(n) gets the nth capturing group
Capturing groups \n, refers to the nth outermost set of brackets in regex expression
([a-zA-Z])\1 will match the group preceeding it

**Publish Subscribe and Builder Design Patterns**

---

When publisher changes something, subscribers are notified to respond in whatever way it chooses
Publishers are Observable, Subscribers are Observers. These are interfaces!
Observable:
AddObserver()
DeleteObserver()
NotifyObserver()
HasChanged() – checks if object has changed
ChangeCleared() – clear the changes Boolean

```
class OperatingSystem extends Observable
{
        public void shutDown(){
                this.setChanged();
                String event = “I am about to shut down”;
                this.notifyObservers(event);
        }
}
Class WordFile extends Observer
{
        public void modifyContent(String newContent)
        {
                 content=content+newContent;
                 os.addObserver(this);
        }
        public void promptToSaveWordFile()
        {
                …
                os.deleteObserver(this);
        }
        public void update(Observable o, Object arg)
        {
                promptToSaveWordFile();
        }
}
```

Remember that OS should be a singleton!