Ian Mutahi

CS3250

Classwork Notes 1/28/19-5/15/19

1.28.19

**Java Revision Control**

Keeps track of updates/changes (deltas) to software basis of any good software development, files are checked in and checked out from repo.

What to do: Every time you have something running check (commit) it! Also check it in

Commit = commits to the local repository

Push = pushing the local commits to the central (master) repo

Merge = deals with multiple files, merges them into one

History

1970’s, CCS with only one central computer

1980’s Multiple users have access- file locking (only one person can view/edit the file at a time)

Modern- Distributed and P2P now. Examples: Mercurial. Git (designed by Linus)

Using Git

Non-centralized, distributed version control.

1/31/19

**What is Travis CI?**

When something is pushed to your repo, Travis will automatically check for failures, test the code, and enable continuous deployment (CI). It essentially automates the pipe-line process for product deployment. (\*Note Jenkins is a similar tool to keep track of the local machine repo)

Tags

Typically used to “tag” version releases

2/4/19

Git flow

The flow of branching and pushing from branch to branch within the git system, The master branch supplies the develop (dev.) branch. Feature branches (which generally correspond 1:1 for user stories) are created from the dev. branch, worked on, then pushed into the dev. Branch again once completed. Once ready, the dev branch is pushed into the release branch, and a release is made.

Other common branches

Hotfix: used for major bug fixes (facetime), avoided unless absolutely needed because these are bug fixes that apply directly to the release.

Version Semantics

Typical layout is Major.Minor.Patch as in: 1.2.31

Typically, must use a public API (otherwise why even do it?)

Major version zero is typically denoted (0.y.z)

* 1. Denoted initial public AP

Key: ab32fda27736f954e0c0bcfdaf65535385718fab

**2.13.19 Sprint 2 notes**

javac test.java

javap -help

javap -v -c -l (verbose, code, linenumber)

xxd test.class Hex dump on bash

Delete HelloWorld from “project”

Class ClassFile():

Def \_\_init\_\_(self, filepath):

With open(filepath) as file:

Byes = file.read()

Magic\_number bytes[0:4]

SonarCloud:

As a developer I want to research Dynamic analysis tools that work with windows so that Luke doesn’t have to pull his hair out. PyLint and Radon can be used (needs to represent all the possible data for dynamic analysis- includes: Maintainability index would be nice, (McCabes complexity aka cyclomatic complexity), Linting of some sort (pylint works but should be automated), CodeCov (for coverage),

Radon notes:

* cc is the radon command to compute Cyclomatic Complexity
* -a tells radon to calculate the average complexity at the end. Note that the average is computed among the *shown* blocks. If you want the total average, among all the blocks, regardless of what is being shown, you should use --total-average.
* -nc tells radon to print only results with a complexity rank of C or worse. Other examples: -na (from A to F), or -nd (from D to F).
* The letter *in front of* the line numbers represents the type of the block (F means function, M method and Cclass).
* cc: compute Cyclomatic Complexity (add -s flag to show complexity)
* raw: compute raw metrics
* mi: compute Maintainability Index
* hal: compute Halstead complexity metrics

PyLintcommand line stuff: <https://pylint.readthedocs.io/en/latest/user_guide/run.html>

Radon command line stuff:

<https://radon.readthedocs.io/en/latest/commandline.html>

Order of operation: CodeCov, then class file build, then mocking, then research, then finally opcodes.

For presentation: Save it as a pdf for turning in!, cannot present a topic unless 90% coverage is DONE

Mocking Creating a “fake” object to run tests on. Basically creating an object that will answer any call in order to test its function

Python code: mock\_open(specify param) will tell you what happens when you open it a bunch!

Instead of running unit tests, read in file called cd (class file) with the following code

If ‘\_\_main\_\_’ == name:

Disassemble class file to see tools, methods, etc….: javap -c -v test.cl

2.18.19

Opcodes everything between 0-255 is a function, a byte that corresponds to a function

More Scrum stuff

2.20.19

Implement all the integer op-codes (all begin with I)

Into to Software Engineering (a high level view)

Basically discussed the complexity of current software system (beginning to be on oar with living creatures)

2.25.19

Opcodes come form file, operands go onto stack (in a nutshell)

Opcode = code for the operation as in a = 1 + 2 (the + is the operation, or opcode)

The operands ( 1, 2 come form the run-time stack) Input comes from the jvpm class file.

Example: ox60 is the opcode for integer add. SO for a = 1 + 2 the 1 and 2 would be on the runtime stack.

Icont\_1, Iconst\_2.

\*From Dr B: Rest of the semester looks like this, read the opcode from a class, interpret, then execute the operation.

\*Later for system Test, use something Like <jasmine> (which is an assembler for JVM, takes in jvm opcodes and spits out hex opcodes)

Lecture Notes

From Dr B, “Make sure you test things or bad things will happen, straight up”.

Software engineering terminology:

Program requirements:

Functional requirements: What a program needs to do, tends to be a yes/no situation (either does it or does not do it)

Non-functional requirements:

The manner in which the functional requirements are met, performance, how usable/maintainable, these tend to be rated on a scale (instead of yes/no)

2.28/ 3.4 First two design patterns from head first design patterns

First Design Pattern

Third design principle

Favor composition over inheritance. Favor has-a relationships over is-a relationships

The Observer Pattern

Defines a one-to-many dependency between objects so that when one object changes state, all of its dependents are notified and update automatically. It’s a software design pattern where the subject class has a list of dependants. You can subscribe and unsubscribe from the notifications that come from the subject. Mainly used to implement distributed event handling systems, in “event driven” software. Also, a key part in the familiar model-view-controller (MVC) architectural pattern. (where a model is a persistent data type like a database or a flat file)

Basically: The subject has a list of dependants, known as observers, which it can notify/update.

Example of python code: for each object in observers:

o. update()

Design Pattern 5

Strive for loosely-coupled designs between objects that interact. Objects have very little information about each other and have no shared state! Basically… Objects know very little about each other in an effort to improve inheritance and maintainability. AKA avoiding the ripple effect (“whack-a-mole effect”). Public variables, setter methods are a great example of WHAT NOT TO DO because they essential become public data which anyone can change (and then break).

3.6.19 More design Patterns

The decorator pattern (structural pattern)

Designs should be open for extensions, but closed for modification. Basically, because you want to be able to count on it when you need it. Should have the same super type as the objects they decorate. Can pass a wrapped object anywhere the original could be passed (because super types are the same!). Objects can be decorated at runtime.

\*This is one of the reasons composition is favored over inheritance. Functionality can simply be added at run time.

Example of Java decoration

Line number read is decorated Bufferedreader which is inherited from read which is inherited from object

The factory Pattern (the first creator design pattern we’ve looked at)

Handles the details of object creation (basically why make a new object if there is one already made that works for you)

Client doesn’t care what TYPE is used, just functionality.

3.11.19

Sprint 4 notes:

When you get to invoke virtual (native method-call) just go and fetch it off

Now factory pattern continued:

Factories don’t have to be abstract, they can have default constructors, and then call down additional functions through abstractions.

Dependency Inversion principle

Depend upon abstractions not concrete classes. Summed up: we don’t care about the concrete implementation, only the functionality (who cares what type of list you write, you just need a list functionality). Therefore, use abstraction so when API and codes changes, code is less affected because of more abstraction instead of concrete typing.

* No variable should hold a reference to a concrete class
* No class should derive from a concrete class
* No method should override an implemented method of any of its base classes.

Dependency Injection

A technique whereby one object (or static method) supplies the dependencies of another object (and then you “inject” the new parts that you need) Basically a class or client has MOST of what you want, and then you pass an object that provides it with the injection of the increased functionality.

Inversion of control

Using frameworks instead of libraries\*\*\* (dependency pulls down code from reusable libraries)

Abstract vs Non-abstract factories

Factory: Creation through inheritance, creates objects of a single type

Abstract Factory: Creation thru composition, instantiates via new and passed, creates families of related objects via factories

Design pattern 05, 4.1

CH 5, singleton

Need only one of: Thread pools, caches, dialog boxes, preferences, logging, device drivers, I/O

BUT there are times ewe might not need a singleton EVERY time we run a program. EXAMPLE: static methods: there are only one of them.

Summed up: ensures a class only has one instance.

Threading

Synchronize getInstance,

Volatile

Read and writes happen to main memory (not form individual CPU caches)

Atomic

Any write to a volatile variable establishes a happens-before relationship with subsequent reads of that same variable. This means that changes to a volatile variable are always visible to other threads.

Read and writes are atomic for reference variables and for most primitive variables (except long and double variables)

Reads and writes are atomic for all variables declared volatile (including long and double variables)

CH6, Command pattern

A behavioral design pattern….

Essentially, we have a caller which will call the ‘execute’ which has a concreteCommand (essentially an interface) which refers to the receiver (which performs the action requested)

Adapter Pattern

We have an interface, we love it, we write an API to it, essentially what we do is adapt it to a new use.

Object adapters use composition, class adapters use inheritance. Essentially, you have a bunch of client code that you don’t want to change so instead you use an adapter to translate.

Façade

Provides a unified interface to a set of interfaces in a subsystem. Façade defines a higher-level interface that makes the subsystem easier to use. \*Not a one-one relationship because it is being applied to many different classes or methods. Facade is simply giving us a high-level abstraction.

Difference between adapter and façade

Adapter alters an interface to make it usable, façade makes a complicated interface easier to use.

Principle of least knowledge

Basically, don’t concern yourself with every detail, strive for loosely coupled designs. AKA the “Law of Demeter” which states: Methods may talk to (their own object, objects passed as parameters, objects they instantiate, instance variables) .

Some pattern (hooks?)

Hooks (ppl can ‘hook’ their own implementation in)

Can define concrete methods that do nothing unless subclass overrides them. Use abstract when subclass must implement, hooks when optional. AKA the Hollywood principle (don’t call us we’ll call u). Lowlevel- hooks into system. Highlevel- calls at the appropriate time.

EXAMPLE: java’s Array.sort calls the compareTo() method

Ch9 Iterators and composites

Iterator pattern

Provides a way to access the elements of an aggregate object sequentially without exposing the underlying representation. This place the task of traversal on the iterator object, not on the aggregate, which simplifies the aggregate interface and implementation, and places the responsibility where it should be.

Design Principle

A class should have only one reason to change (single responsibility)

This helps with HIGH COHESION (all methods related to purpose)

Composite Pattern

Allows you to compose objects into tree structures to represent part/whole hierarchies. Composite allows clients to treat individual objects and compositions of objects uniformly. We can apply the same operations over both composites and induvial objects. Can ignore differences between the two (think recursion!)

Example: Given a part or a whole (in this case animals, say 3d rendering)

Animals, mammals, cats:

All breath, move, etc….

Objects in a scene:

All have texture, placement, etc…

4.17.19

The State Pattern

The combination of the value of all variables in an object (in your program)

State machines are used all the time (NFA’s DFA’s)

Automata- Combinatorial logic, FSM’s (finite state machines), Pushdown (FSM with stack), Turing Machines.

More examples of machines with a state: Wending machines, elevator, locks, traffic signals, etc…

FSM’s are generally limited to the amount of memory (determining the number of states) it has.

State Pattern Allows an object to change its behavior when its internal state changes. The object will appear to change its class. Very similar to strategy pattern in the sense that strategy is an alternative to sub-classing as it uses composition instead of inheritance (avoids class explosion). State is an alternative to having lots of conditionals.

Effective Java CH 2.

Consider Static Factory Methods instead of Constructors

One advantage is they have names whereas constructors do not, and one has to differentiate via parameters. This can be confusing and lead to many errors.

A class can only have constructor with a given name. Don’t change order of constructor parameters to differentiate. Makes a lot more sense from a readability point of view (as in reading the code).

Static factory methods don’t have to create a new object. (constructors always do). Maybe there is an object already created that works! This helps with immutable classes and pre-constructed instances.

This allows you to return a subtype. java.util.collections contains all static methods that work on many types. Polymorphism (like being able to make a sort that can handle all different types), addAll, binarySearch, disjoint, frequency, min, max, sort, shuffle ,reverse.Type returned can be non-public, can vary implementation.

Returned class need not exist at the time the class is written. Allows run-time specification. JDBC (Java database connector) is an example. JDBC consults a configuration file you create, to specify the database in the config file, and it instantiates the correct object AT RUN TIME.

Service Provider Framework

Service interface (runs a query against a database or writes a value against a database)

Provider registration

Service access

DISADVANTAGES OF STATIC FACTORY METHODS INSTEAD OF CONSTRUCTORS

Classes without public or protected constructors cannot be sub-classed. Not called out in Javadoc. Popular Java static factory names: value Of, getInstance (most often this one), newInstance, getType.

Consider a constructor when faced with many constructor parameters

If a class has many fields that need initializing, constructors have long lists of parameters. To solve this, constructors are often chained/telescoped.

Example in Java: where constructor basically calls anther constructor with the ‘this’

Public class NutritionFacts {

Private final int servingSize;

Private final int servings;

Public NutritionFacts(int servingsSize, int servings){

this(servingSize = xxxxx, servings, xxxxx);

}

Builder example: This is calling the Builder class which has a sperate constructor to Build to object, this helps builds this more complex object with many parameters.

Public static void main(String[] args) {

NutritionFacts cocaCola = new NutritionFacts.Builder(240, 8).calories(100).sodium(35).carbohydrates(27).build();

Enforcing non-instantiability with Private constructor

Just have a private no-args constructor, this way the class cannot be subclassed and there WILL BE ONLY ONE.

So… prefer primitives to boxed (automatically converted) primitives. Be careful of unintended autoboxing!!!!! As in having java autobox from capital ‘L’ Long to regular long.

Obey the General Contract when Overriding the .equals method

Sometimes, you don’t need to when: All objects are unique (such as threads)

When you simply don’t need it (as with random number generators). If the class is private or package private, and you know you don’t need it (won’t be calling it). Or if the superclass .equals method works well, such as sets, lists, and maps getting form AbstractList, etc.\

Java project with built in library for equals and getter methods. <<https://projectlombok.org/>>

When to implement

When logical equality (.equals) is different from simple object identity (==)

\*Note from Dr B.- .equals asked whether a thing is in the same state, aka equivalence. == test whether they are the same object.

Must Implement an Equivalence Relation

Must be reflexive: x.equals(x) must return true

Must be symmetric: x.equals(y) must be true if and only if y.equals(x) also returns true

Consistency

Do not write .equals methods that depends on unreliable resources (like a network being available or a file being open). Example: Java’s URL equals method relies on IP address comparison (which can change or you can simply not be connected to the network).

Recipe for writing a good .equal method

Check for object == this

Use instanceOf to check for correct type

Cast argument to correct type

Test == for all significant fields (except for Float.compare, Double.compare, and Arrays.equals)

Also override hashCode

Use @Override

To sum up

There is no way to extend an instantiable class and add a value while preserving the equals contact. You can safely ass value to a subclass of an abstract class.

Always override hashCode when you Override .equals

When invoked on the same object, and the object hasn’t changed to affect equals, always return the same integer. (Does not have to be the same integer from runtime to runtime)

If two objects are equal, both hashCodes must be the same.

If they are not equal, it is not required to produce distinct, hashCodes. (if not, hashtable performance can be affected. “return 42” is legal, but horrible and hackish).

Creating a hashCode

Set result to = 17

For all the instance fields in your object or class:

If Boolean, c = f ? 1 : 0

If byte, char, short, or int, c = (int) f

If long, c = (int) (f^(>>> 32))

If float, c – Float.floatToIntBits(f)

If double, c = (int) (Double.doubleToLongBits(f) ^ (Double.doubleToLongbIts(f) >>> 32))

If reference, call hashCode to object

If array ….

Then update result = 31 \* result + c

Exclude any redundant field (which you really shouldn’t have nay anyways) and then Ignore any fields ignored by equals.

Always Override toString method

Makes class much more pleasant to use. When practical, toString method should return all interesting information in object. One has to choose the format returned (choose some useful stuff like making sure variables equal each other, AKA return the instance variables)

Consider implementing comparable in your objects

Similar to equals, but provides ordering information (useful for sorting). Is generic, and is useful ( like in Arrays.sort() )

Returns comparison between two objects (-1 if first less than second, 0 if equals to, 1 if greater than)

This basically requires you to implement the compareTo method in your object.

More effective Java notes

Use function objects to represent strategies

Favor static member classes over non-static member classes

For next class ( presentation) include total hours for the semester, and the standup will be an entire semester retrospective.

Push all notes and evidence of your research to GitHub (commit based evidence).

Dark Web Presentation

Tor Technology

The onion router is the way they send packets, funded and researched via navy special operations (US Naval Research Lab). Utilized a special Firefox browser (Tor browser now) and a sophisticated anonymizing network. Essentially circumvents a nation of organizations ability to packet filter (exclude or track data)

Private Internet Access VPN recommended by Dr B. (Does VPN use Tor technology?)

Hidden Services

When the web server itself is hidden within the Tor network (cannot access normally)

Surface vs Deep vs Dark web

Surface is what you see, google, amazon, etc.…

Deep Web = medical records, academic databases, etc. (stuff you need password to log it to, google wont search it)

Dark Web, need a special browser to view, no readily available indexing services (can’t search it). Uses for illegal trafficking, military, etc.…

Resources

Darkwebnews.com

Deepdotweb.com

Zqktlwi4fecvo6ri.onion

Blockchain Presentation

Essentially a public ledger or list of transactions. Each new block contains a cryptographic hash code of the previous block, a timestamp, and transaction data.