GIT:

-Steps involved in performing actions

-not git commands

-Purpose of branches

Allow you to make changes to files without altering the main copy by creating a “branch” of that file that is the same up to the branching point. Others can also branch and work on their own copies so nobody is stepping on the others’ toes.

-Pull Requests

Submit a proposal for your branch to be “pulled” into another branch (merged into it).

-Commits

A snapshot of the current project. Can be commented on. Records changes to the repository

-Staging

Step before commit process. Staged files are those that are included in the commit.

-Push/pull:

-Push:

Updates remote references along with associated objects. Interacts with the remote repository. Puts committed files in remote repo.

-Pull:

Process of fetching and merging. Gets updates from the remote and merges them to the local branch.

\*GIT-FLOW:

-

\*FEATURE-FLOW:

-

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*8

MOTIVATION:

\*DESIGN-PATTERNS:

Many problems follow noticeable patterns. Don’t reinvent the wheel. Communicate Solutions to common programming problems.

\*AUTOMATED-TESTING:

-Prevents bugs.

-prevent regression when code is modified.

-Can be used as a form of documentation of intent.

\*UML

Maintenance and enhancement cycle dominate cost/time. Errors can cause serious issues. Provides details for the system before development so you can find issues and weak points before committing to coding.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*8

KEY-OO-CONCEPTS:

-Visibility

Refers to the ability of being able to “see” things. The accessibility of Classes/objects/variables.

-Abstraction

Not knowing which object is doing what. Moving focus from details and concrete implementation, to types of things and available operations.

-Encapsulation

Hiding the implementation details. Hide them within the class. Class should be like black box. Providing only a method of interfacing with it.

-Modularity

Packages. Separating functionality into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of desired functionality.

-Polymorphism

Many forms. When the class or child class are interchangeable it is polymorphic.

UML-CLASS-DIAGRAMS:

\*VISIBILITY-NOTATION:

-public(+)

Feature is available to any class

-protected(#)

Feature is available in the class itself, all classes in its same package, and all its subclasses.

-package(~)

Feature is accessible to the class and all classes in the same package

-private(-)

Feature is only accessible within the class itself.

\*INTERFACE-VS-CLASS:

-no attribute section on interface

-need to explicitly state <<INTERFACE>>

\*RELATIONSHIPS:(TODO Add the images here )

-Aggregation

Open Diamond. Diamond on container.

-Composition

Black Diamond. Deals with using new keyword in class

-Navigability

Open arrow = one way. No reference to owner.

-Dependency

Dotted open arrow, a method signature uses it.

\*What is needed to show why a relationship exists:

-attributes

-methods

-explanation aggregation vs composition)

UML-STATIC-VS-DYNAMIC:

\*CLASS-VS-OBJECT:

-what is the difference

Object is a member/instance of a class. Objects have states.

-which aspects of UML modelling address each

Class diagram vs Sequence Diagram???????????????

-Difference in modeling:

-Static:

Class diagram

Structure of object model that will be created

-Dynamic:

The interaction of objects

The order of events and actions

-State

Control flow with states and transitions

-Sequence

The interaction of objects

Time ordering

When objects are created

Methods invoked, and order

Life of a function

\*\*\*Key Concept\*\*\*

-Why do we have each???

They all model different things

UML-SEQUENCE-DIAGRAMS

-Purpose

The interaction of objects

Time ordering

\*What needs to be shown:

-Objects:

-showing concrete class or interface is fine if interchangeable

-Call representations

-For return either solid or dotted line is fine

-Written on lines:

\*\*Does this mean what is written on the lines? as in the methods being passes?(slide 8ish)\*\*

UML-STATE-DIAGRAM:

-Why do we have them?

Depicts control using states and transitions

-Concept does not map to object model

-State vs trigger:

State: represents condition or situation that an object satisfies

Trigger: indicates relationship between objects on which an action leads to a change in state

-Transition trigger

-Nested state diagrams

JAVADOC:

\*\*\*Will not ask syntax for creating\*\*\*

-Fair questions--What should be documented?

-Packages

-Classes

Author

description

-Methods

Pre/post

Its function

Return

parameters

-Attributes

Should describe what it is

-Goal of description of each

JAVA/OO-IMPLEMENTATION CONCEPTS:

-Purpose of package

A package is a namespace that organizes a set of related classes and interfaces

\*MEANING OF MODIFIERS/ WHEN USED:

-Static

Field belongs to a class, not instance of a class.

You want to have variables that are common to all objects.

Static methods can only access class static attributes

-Final

-Will always contain the same value.

-Make objects non-transitive.

-Final classes cannot be specialized

-Final method cannot be overwritten or hidden by subclasses.

-Abstract

-Abstract classes cannot be instantiated, but can be specialized

-Abstract method is a method that is declared without an implementation (without braces, and followed by a semicolon)

- If a class includes abstract methods, then the class itself must be declared abstract

\*EXCEPTIONS:

-Acceptable use of Exceptions

-Exception handling for graceful exit/recovery in event of unexpected failure.

-Not meant to replace debugging

-Hierarchy of Exceptions:

-given description, which type of error / exception:

-RuntimeException vs Other Exceptions

-Difference between checked vs unchecked exceptions

-Unchecked:

Errors/Runtime-Exceptions

Thrown by JVM

Unit testing to eliminate

-Checked:

All other exceptions

Any function that does not explicitly handle a check exception must declare that it throws that exception

UNIT-TESTING:

\*\*\*Not be asked JUnit syntax

-Black box:

-Derive test cases based on specification of component alone.

-Tests can be created independently of implementation.

-Tests should include valid and invalid elements.

-Pre/post conditions

-White box:

-Derives test cases based on structure of code implementing the functionality

-Tests cannot be created prior to code completion

-only was to ensure robustness of code.

-Cover every statement, branch, condition

\*\*\*KEY-CONCEPT\*\*\*

-Difference between the two

-How do you come up with black box conditions

-How do you come up with white box conditions

\*Given description or code segment, identify test condition:

-table we did on board

-description of test condition needed/expected result:

1. a=c, c=true, expected result exception raised

2. a>1. c=true, expected result return true

DESIGN-PATTERNS:

-What is a design pattern?

Describe problem that occurs repeatedly and the solution to that problem.

Can be applied to all the situations even though surrounding implementation changes

\*CREATIONAL-PATTERNS:

-deal with process of creating objects

\*STRUCTURAL-PATTERNS:

-deal primarily with the static composition and structure of classes and objects

\*BEHAVIORAL-PATTERNS:

-deal primarily with dynamic interaction among classes and objects

-Know when you would use a particular design pattern and its benefits

-Be able to come up with an example application of that pattern that is not in the book or slides

-Explain why that is a good example

-Draw UML for your example

-How would you code that pattern?

-semi pseudo code:

-some key aspects

-abstract/final/interface/checked exceptions

REFACTORING:

-Not a design pattern; an idea of analyzing model/code to make it better

-Faster implementation of new components, Bug fixes made in one location, improve understanding of code

\*SPECIFIC-REFACTORING:

\*Extracting commonality:

-Within class

-Inheritance

If duplicate code appears across 2 different classes may be able to refactor by inheritance. Common code pulled into either shared parent class or new parent class created. Any variables used also pulled to parent class.

Created problems if the 2 classes are not related enough to justify inheriting. If either class has different parent, not possible.

-Delegation

Duplicate code pulled to different class and implementation of the common code is delegated to this class. Required original classes to have a reference to the delegated class. Non-visible variables need to be passed into helper class.

-why/when would you use it rather than inheritance???

Delegation can always be used and does not include inheritance restrictions. But reading is more difficult than inheritance.

\*\*\*Note: refactoring can result in design patterns\*\*\*

--EX.: Refactoring commonality on Cake/Pizza resulted in Template pattern--

CHAIN-OF-RESPONSIBILITY:

\*Intent: Avoid coupling class making request from class servicing:

-Reduced coupling

-Added flexibility

-But, receipt isn't guaranteed(i.e. not guaranteed class will service(might not be caught))

TEMPLATE:

-Common invariant functionality, concrete classes must provide key functionality via hooks

STRATEGY:

-Client coupled to interface, variants in interchangeable algorithms, client doesn't care about implementation

-Similar to template methods, but delegates implementation rather than relying on child implementation

SINGLETON:

-Want entire application to use a single instance of a class

-Unit testing dependent classes difficult due to global state

-Potential problems for parallel execution

\*\*\*\*\*\*\*SAMPLE-QUESTIONS\*\*\*\*\*\*\*

SHORT-ANSWER:

-What is the difference between encapsulation and modularity?

Encapsulation is about hiding things within the class. Modularity is about keeping related data in separate modules, no outside package can see.

-What is the purpose of a package comment in Javadoc?

UML:

-Create the UML class diagram for the following:

-Library has paperback and hardcover books; and employees

-Given a list of books an employee could be told to reshelf those books. An employee could either reshelf the books alphabetically by book title or alphabetically by author

-You should draw all relationships between the classes you define and the attribute/methods to support those relationships

ANSWER-THE-FOLLOWING

-What design patterns is the TestTaker an example of and why?

-what is the purpose of the takeTest methods

-what is the purpose of Written and Spoken if no attribures/methods in class diagram?

-What can you say about where each method Blind has is implemented?

-If Test and TestTaker have the same functionality for writeCharavteristics why wouldn't you extract this to a super class?

-What refactoring pattern is CharacteristicWriter an example of?

-Why is the CharacteristicsHolder interface needed?

-Based on visibility, what can you say about how the CharacteristicWriter was refactored?

DYNAMIC-UML:

-Assume that there is a class Main which has a method that kicks off the whole process.

1. Creates a test and test taker(pick any type you want)

2. Tells the test taker to take the test

3. Tells the test taker to write its characteristics

Draw the sequence diagram fot the full execution of Main's method

READING-SEQUENCE-DIAGRAMS

-Assume methodA is public, methodB is protected, and methodC is package; and all take no arguments and don't return anything. What would be the signature of each method in Java

-Describe what this sequence diagram is showing(i.e Why would it be needed)

DYNAMIC-UML

-Draw the state diagram for a student taking a computer based test that must be submitted at the end. Initial state computer is off, end state computer off. Note the test is presented a question at time; do not need to consider computer crashing; computer may be initially unplugged.

-Draw a sequence diagram for this process

TESTING

-Given the signature/interface to a function:

-Identify the black box test conditions

-Given a function description and its implementation:

-Identify the black box test conditions

-Identify any white box test conditions

-You will be required to specify what type a specific test is