Final Exam Review

CS 4354
Summer II 2016

Jill Seaman

Final Exam

- Thursday, August 11, 11AM-1:30PM
- Closed book, closed notes, clean desk
- Content (Comprehensive):
  - ✦Textbook: Chapters 1, 2, 3.4-5, 4.1-5, 5.1-7, 6.1
  - ✦GRASP, JUnit + Refactoring Lectures
- Weighted more towards content from second half of the course
- 35% of your final grade
- I recommend using a pencil (and eraser)
- I will bring extra paper and stapler, in case they are needed.

Exam Format

- 120 points total
  - ✦60 pts: 30 Multiple choice questions (scantron), may include:
    - Tracing code (what is the output)
    - Reading diagrams (what does it mean, is it a good design)
  - ✦60 pts: Coding and Design:
    - Writing Use Cases, Writing CRC cards
    - Drawing UML diagrams (class, sequence, state)
    - Writing programs/classes/code/JUnit tests in Java
    - Applying Design Patterns
- Each question will indicate how many points it is worth (out of 120)

Java: Introduction

- Compilation, execution (byte code)
- Features
  - ✦Object-oriented, inheritance, polymorphism, garbage collection
  - ✦Exception handling, concurrency, Persistence, platform independence
- Primitive types, control flow, operators, assignment (like C++)
- Classes, fields, methods
- Objects are references (pointers underneath)
- Parameter passing (pass by value, but objects can be mutated)
- Constructors, this
- Packages, directories, import statement
- Java library, API

Java: Introduction

- String, toString
- ArrayList
- arrays

- Javadoc, how to document the elements of a program

- Access specifiers: public, private, protected, [package]

Java: Input/Output

- Input using a Scanner
- Output using System.out.println()
- Wrapper classes (Integer, Float, Double, etc)
- Formatting using the DecimalFormatter and/or String.format

- Object serialization
  - ✦ObjectInputStream, ObjectOutputStream
  - ✦readObject, writeObject
  - ✦Understand how it works (when to cast)
  - ✦Don't memorize the exceptions

Java: Inheritance

- Interfaces
  - ✦Using, Defining and implementing Interfaces
  - ✦Sorting: implementing Comparable<T> or Comparator<T>
- Inheritance
  - ✦class hierarchy: superclass, subclass, (extends keyword)
  - ✦overriding methods
  - ✦constructors
- Polymorphism
  - ✦upcasting, polymorphic functions, dynamic binding
- Abstract methods and classes

Java: Collections and Exceptions

- Collections
  - ✦LinkedList<T>
  - ✦Iterator<T> (next(), hasNext(), remove())
  - ✦iterator() method

- Exceptions
  - ✦Semantics (how exceptions are thrown/caught) and syntax
  - ✦Catch or specify requirement (how to satisfy)
  - ✦Runtime exceptions
  - ✦Create your own exception classes (and instances)

Ch 2: The Object-Oriented Design Process

- Analysis, Design, Implementation
- Objects and Classes
- Identifying Classes and Responsibilities
- Identifying Relationships
  - ✦Dependency
  - ✦Aggregation
  - ✦Inheritance
- Use Cases
  - ✦Actor
  - ✦textual descriptions (set of steps), with variations
  - ✦single interaction between actor and system.

Ch 2: The Object-Oriented Design Process

- CRC cards
  - ✦Classes, Responsibilities, Collaborators
  - ✦Index cards describing each class
  - ✦Walkthrough use cases to generate/develop the cards
- Class Diagrams
  - ✦Classes, attributes, operations, associations
  - ✦unidirectional, bidirectional associations
  - ✦Dependency, Aggregation (or association), Inheritance
  - ✦Multiplicity {1, 0..1, 0..n, 1..n}
  - ✦one-to-one, one-to-many, many-to-many

Ch 2: The Object-Oriented Design Process

- Sequence Diagrams
  - ✦Describes interactions between objects
  - ✦Objects, lifelines, activation boxes
  - ✦Messages from one object to another (must be methods on the receiving object), messages run in sequence top to bottom.
- State Machine Diagrams
  - ✦States an object can go through in response to external events,
  - ✦State is a node
  - ✦Transition is a directed edge labeled with the event that causes it
- For all of the types of diagrams:
  - ✦Be able to draw simple diagrams, like for Assignment 3
  - ✦Be able to read (understand, interpret) diagrams.

Ch 3: Class Design

- The Importance of Encapsulation
  - ✦Sharing Mutable References unintentionally
  - ✦Separating Accessors and Mutators
  - ✦Side effects
  - ✦Sharing Mutable References intentionally (Law of Demeter)
- Analyzing the Quality of an Interface
  - ✦cohesion
  - ✦completeness
  - ✦convenience
  - ✦clarity
  - ✦consistency
- See Assignment 4

GRASP: Assigning Responsibilities to Objects

- GRASP
  - ✦Deciding which classes should perform which operations
  - ✦Information Expert: That which has the information does the work
  - ✦Creator: Assign class B the responsibility to create instances of class A if:
    - B aggregates A objects.
    - B contains A objects.
    - B records instances of A objects. OR
    - B has the initializing data that will be passed to A
  - ✦Low Coupling: Keep the amount of dependency on other classes low.
  - ✦High Cohesion: Keep the tasks of a class focused and related to each other.
  - ✦Controller: Use a controller class to separate the UI from the entity objects.
- See Assignment 4

Ch 5: Design Patterns

- Concepts
  - ✦Delegation
- Design Patterns
  - ✦ Iterator Pattern
  - ✦ Adapter Pattern
  - ✦ Strategy Pattern
  - ✦ Observer Pattern
  - ✦Composite Pattern
  - ✦Decorator Pattern
  - ✦Facade Pattern
- Be familiar with the class diagrams.
- Be able to apply them.
- See Assignment 5

JUnit

- JUnit
  - ✦Open source framework for writing and running unit tests
  - ✦Provides automation
  - ✦Annotations: @Test, @Before, @After, @Ignore
  - ✦Assert Methods: fail, assertTrue, assertFalse, assertEquals, assertNull
  - ✦Separation of testing code from production code
  - ✦Testing methods, classes, and collaborating classes.
  - ✦Be able to write simple test cases, using the Annotations and Assert methods as we did in Assignment 6.

Refactoring

- Disciplined technique for changing a software system: altering its internal structure without changing its external behavior
- What are the benefits, when is it applied?
- Know the refactorings listed in the lecture:
  - ✦Rename, Extract Method, Encapsulate Field, Move Method/Field
  - ✦Pull Up Method/Field, Push Down Method/Field, Extract Superclass
  - ✦Replace conditional with polymorphism, Remove Parameter
- Know the “bad smells” and how to fix them.
- Be able to apply (or recognize) simple refactorings “by hand”
- See Assignment 7 (the lab)

Sample Questions: Multiple Choice

- You are designing a datatype to store a mathematical expression composed of binary operations (plus, minus, times, divide) and numbers, such as: (3+4)/(6*2). These expressions can be drawn as a tree with the operations as the node values and the numbers as the leaves. What design pattern should you use for this?

  (a) adapter (b) composite (c) facade (d) iterator

- Which of the following is NOT potentially an example of refactoring :
  (a) Push down field.
  (b) Moving a class to a different package.
  (c) Adding a new feature requested by a customer.
  (d) Making your code implement the Facade pattern (without changing the external behavior of the system).

Sample Questions: Class Design

- Rewrite (part of) the following code, so that it does not violate encapsulation (information hiding):

```
public class Time implements Cloneable {
   private int hours, minutes;
   public Time (int h, int m) {
      hours = h; minutes = m;
   }
   public String toString {
      return Integer.toString(hours)+":"+Integer.toString(minutes);
   void addMinute() {
     if (minutes==59) {
        minutes = 0; hours++;
     } else
        minutes++;
   }
   //implements the clone() method here
}
public class TravelClock {
   private Time currentTime, alarm;
   private int temperature;
   Time getCurrentTime() { return currentTime; }
}
```

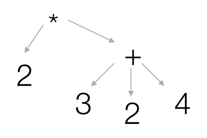
Sample Questions: JUnit

- Write a JUnit test for the Time class to test the addMinute method:

```
public class Time {
   private int hours, minutes;
   public Time (int h, int m) {
      hours = h; minutes = m;
   }
   public String toString {
      return Integer.toString(hours)+":"+Integer.toString(minutes);
   void addMinute() {
     if (minutes==59) {
        minutes = 0; hours++;
     } else
        minutes++;
   }
}
```

Sample Questions: Design Patterns

- Use the COMPOSITE pattern to implement arithmetic expressions involving addition and multiplication of numbers. Add an eval() function to evaluate the expression.



$2 * (3 + 2 + 4)$

- this example evaluates to 18.

Sample Questions: Java programming

- Draw a class diagram showing the structure of data about employees of a given company. The employees attributes include name, street address, city, state, zip, and an id number. Full-time employees have an annual salary. Part time employees have an hourly pay rate. Departments have names and a group of employees, but each employee can be in only one department. Employees work on one or more projects, which also have names. Projects may have multiple employees assigned to them. Include attributes, associations, and multiplicity, in your diagram.
- Implement in Java the Employee class structure described above. The Employee class should have a polymorphic function called weeklyPay. For Full time employees, their weekly pay is their salary divided by 52. For part time employees their salary is their hourly pay rate time 40. In another class called Driver, define a main function that creates an array or ArrayList of Employees of two full time and one part time employees, then iterates over the list and outputs the name and weekly pay for each employee.