Name:

Midterm

CS 121

Software Engineering

Fall 2022

October 19, 2022

Instructions

This exam contains 9 pages, including this one. Make sure you have all the pages. Write your name on the top of this page before starting the exam.

Write your answers on the exam sheets. If you finish at least 15 minutes early, bring your exam to the front when you are finished; otherwise, wait until the end of the exam to turn it in. Please be as quiet as possible.

If you have a question, raise your hand. If you feel an exam question assumes something that is not written, write it down on your exam sheet. Barring some unforeseen error on the exam, however, you shouldn't need to do this at all, so be careful when making assumptions.

Question	Score	Max
1		20
2		30
3		29
4		21
Total		100

Question 1. Short Answer (20 points).

a. (5 points) Briefly explain the difference between a *public field* and a *private field.* Give one reason it might be a good idea to make fields private.

> **Answer:** Public fields can be read or written from any class. Private fields can only be read or written by the class they are inside of. Making a field private ensures the state of an object cannot be unexpectedly changed from outside of its class.

b. (5 points) Briefly explain the difference between how == and the `equals` method are typically used in Java.

> **Answer:** The == operator is physical equality, which compared objects using their address in memory. In contrast, `equals` is a method of `Object` that can be overridden in a subclass. Typically, `equals` is used for a deep equality check (e.g., comparing values of objects' fields rather than just their addresses in memory).

c. (5 points) Parnas's paper is titled, "On the criteria to be used in decomposing systems into modules." What criterion does Parnas recommend for decomposing systems into modules? Give two reasons why that is a good criterion to use in identifying modules.

> **Answer:** Parnas recommends using *information hiding* to decompose systems into modules. Information hiding helps make it easier to change an implementation without changing the client; to divide work more effectively across teams; and to make code more understandable.

d. (5 points) List two general advantages and one general disadvantage of using design patterns. If it's helpful you can refer to specific patterns but you don't need to.

> **Answer:** Advantages: Better information hiding; better modularity; reduces coupling across modules; helps ensure informal coding rules are enforced.
>
> Disadvantages: Adds levels of indirection, which may decrease performance; code is more complex, making it harder to understand;

Question 2. Graphs with Edge Weights (30 points). In this problem, you will implement a few methods from an extended version of Project 1 in which each graph edge has a *weight*. For example, the graph $\mathsf{a} \rightarrow^{12} \mathsf{b} \rightarrow^{5} \mathsf{c}$ has an edge from a to b with weight 12, and an edge from b to c with weight 5. Graphs with edge weights will be implemented using the following data structure:

```
private Map<String, List<Pair<String, Integer>>> nodes = new HashMap<>();
```

For example, the graph above could be constructed with the calls:

```
    nodes.put("a", List.of(new Pair("b", 12))); nodes.put("b", List.of(new Pair("c", 5)));
```

Here is the definition of Pair:

```
class Pair<T, U> {
    T left; U right; Pair(T left, U right) { this.left = left; this.right = right; }
    T left() { return left; }                    U right() { return right; }
    void setLeft(T left) { this.left = left; }   void setRight(U right) { this.right = right; }
}
```

a. (5 points) Write a method **boolean** addNode(String n) that adds a new node to the graph. This method returns **true** if the node was not previously in the graph, and **false** if it was previously in the graph.

```
boolean addNode(String n) {
```

Answer:

```
boolean addNode(String n) {
    if (!nodes.hasKey(n)) { nodes.put(n, new LinkedList<>()); return true; }
    return false;
}
```

b. (7 points) Write a method **int** edgeWeight(String n1, String n2) that returns the weight of the edge from n1 to n2. If there is no such edge, the method throws an exception (any exception).

```
int edgeWeight(String n1, String n2) {
```

Answer:

```
int edgeWeight(String n1, String n2) {
    for (Pair<String, Integer> p : nodes.get(n1)) {
        if (p.left().equals(n2)) { return p.right(); }
    }
    throw new NoSuchEdgeException(); // made up exception
}
```

c. (8 points) Write a method **void** addEdge(String n1, String n2, **int** w) that adds an edge from n1 to n2 with weight w. If there is no existing edge from n1 to n2, this method adds it. If there is already an edge from n1 to n2 in the graph, this method **replaces** the weight of that edge with w. The method adds nodes n1 and n2 if necessary.

```
void addEdge(String n1, String n2, int w) {
```

Answer:

```
boolean addEdge(String n1, String n2, int w) {
    addNode(n1); addNode(n2);
    List<Pair<String, Integer>> succs = nodes.get(n1);
    for (Pair<String, Integer> p : succs) {
        if p.left().equals(n2) { p.setRight(w); return; }
    }
    succs.add(new Pair<>(n2, w));
}
```

d. (10 points) Write a method **int** pathWeight(List<String> p) that takes a list of nodes n1, n2, n3, ... and returns the sum of the edge weights between each successive pair of nodes, i.e., the weight of the edge from n1 to n2 plus the weight of the edge from n2 to n3 etc. If any edge is not present in the graph, this method should raise an exception (any exception). The weight of the empty path is 0.

```
int pathWeight(List<String> p) {
```

Answer:

```
int pathWeight(List<String> p) {
    int w = 0;
    if (p.size() == 0) { return 0; }
    String prev = p.get(0);
    for (int i = 1; i < p.size(); i++) {
        String next = p.get(i);
        w += edgeWeight(prev, next);
        prev = next;
    }
    return w;
}
```

Question 3. Design Patterns (29 points). Consider the following interface for a factory object for creating Widgets:

```
interface Factory { Widget create(); }
```

In this problem, you will implement **class** PooledFactory, which **implements** Factory, but maintains a *pool* of widgets, which are kept ready for use rather than being allocated and deallocated one at a time. More specifically:

- Constructor PooledFactory(Factory f, **int** max) creates a factory that delegates to f to create new Widgets and maintains a pool of at most max Widgets.
- PooledFacory#create() either:
  1. If a Widget is available in the pool, it is removed from the pool and returned.
  2. If no Widgets are available in the pool, it throws EmptyPool, an unchecked exception (assume this exception exists).
- **void** PooledFactory#free(Widget w) adds w back to the pool. If w was not originally from the pool, it throws UnknownWidget, an unchecked exception (assume this exception exists).
- **int** PooledFactory#size() returns the maximum size of the pool.
- **int** PooledFactory#avail() returns the number of Widgets that are available in the pool.
- For full credit, Widget creation should be *lazy*, meaning Widgets should not be created in advance, but only when they are needed.

You may find the following methods useful. You can also use any other standard library classes and methods.

```
LinkedList#isEmpty() // return true if and only if the list is empty
LinkedList#size()    // return the size of the list
LinkedList#add(E x) // add x to the end of the list
LinkedList#remove(int i) // remove element at position i of the list and return it
```

a. (4 points) Aside from the factory pattern, what other design pattern does PooledFactory implement?

Answer: It implements the proxy pattern.

b. (25 points) Implement PooledFactory following the specification above.

```
public class PooledFactory implements Factory {
```

Answer:

```
class PooledFactory implements Factory {
    private Factory f;
    private int max;
    private List<Widget> avail = new LinkedList<>();
    private List<Widget> taken = new LinkedList<>();
    PooledFactory(Factory f, int max) { this.f = f; this.max = max; }
    public Widget create() {
        if (avail.isEmpty()) {
            if (taken.size() == max) { throw new EmptyPool(); }
            Widget w = f.create();
            taken.add(w);
            return w;
        }
        Widget w = avail.remove(0);
        taken.add(w);
        return w;
    }
    void free(Widget w) {
        if (!taken.contains(w)) { throw new UnknownWidget(); }
        taken.remove(w);
        avail.add(w);
    }
    int size() { return max; }
    int avail() { return max - taken.size(); }
}
```

Question 4. Find the Problem (21 points). Each of the following code snippets violates some design principle or coding style we discussed in class. In each case, briefly explain the problem and what you would do to fix it.

a. (7 points)

```
public class Singleton {
    private Singleton theInstance;
    public Singleton getSingleton() {
        if (theInstance == null) { theInstance = new Singleton(); }
        return theInstance;
}   }
```

> **Answer:** There should only ever be one instance of a singleton. However, the constructor of Singleton has package scope, and thus some client code could construct additional instances. The constructor should be made **private**. Also theInstance and getSingleton need to be static.

b. (7 points)

```
boolean hasNode(String n) {
    if (nodes.containsKey(n)) { return true; }
    else { return false; }
}
```

> **Answer:** This code is longer and more complicated that it needs to be. The method body can simply be written **return** (node.containsKey(n)).

c. (7 points)

```
public class Shape {
    double radius;
    double width, height;
    Shape(double radius) { this.radius = radius; }
    Shape(double width, double height) { this.width = width; this.height = height; }
    public double circArea() { return 3.14 * radius * radius; }
    public double rectArea() { return width * height; }
}
```

Answer: This class exhibits poor cohesion, because it mixes code together that should be separated. This code is especially poor because there's no easy way to tell if a given Shape should be a circle of a rectangle. It would be much better to split this into two classes, e.g., Circle and Rectangle, that both implement a common interface or extend a common superclass.