The University of British Columbia
CPSC 210

Sample Midterm Exam Questions

IMPORTANT NOTE:

This does not constitute a sample midterm. These are practice questions, selected from questions that cover the range of topics to be covered on the midterm. These questions should not be considered comprehensive, or be your only source for review.

Question 1. Debugging. (Refer to project: PaymentSystem)

If you run the Main class as a Java application, the output will include the following:

```
Payment[ num=15, type=PalPay, amt=0.724302501394058, txNum=15]
Payment[ num=16, type=PalPay, amt=1.2554252514453499, txNum=16]
Payment[ num=-83, type=Cash, amt=0.0]
Payment[ num=-82, type=Cash, amt=0.3682269387159234]
```

Note that the last two lines of this output have a negative payment number, which is illegal according to the specification of the PaymentRecord data abstraction. Generate two hypotheses about what might be causing this error.

Question 2. Data Abstraction (refer to project: KafeCompany):

The `ca.ubc.cs.cpsc210.kafe.CoffeeCard` class in the `KafeCompany` project contains a partial specification for a data type that represents a loyalty card for the Kafe company. A coffee card can be loaded with credits that can be used to purchase drinks at Kafe stores. Every time a drink is purchased, a bean is added to the card. For every 9 beans earned, a free drink is added to the card. To be purchased, some drinks require more credits than others. However, only one bean is earned per drink purchase, regardless of the number of credits required to purchase the drink.

Study the provided code for the `CoffeeCard` class carefully before continuing.

Design jUnit tests for the `CoffeeCard.useFreeDrink` method – be sure to study the specification for this method carefully. Don't worry if your Java syntax isn't perfect but note that it may help you to examine the tests provided in the `CoffeeCardTests` class. You must assume that the method `CoffeeCardTests.runBefore` runs before each of your tests. If you are unsure about your syntax, include comments to explain what you are trying to do. Note that there's more space for your answer to this question on the following page. You may use Eclipse to develop your solution but you must make a copy of your work onto this exam paper *before the end of the exam*.

Question 3: Debugging

The class `ca.ubc.cs.cpsc210.tests.ContactTests` contains three unit tests for the `Contact` class in the `ca.ubc.cs.cpsc210.addressbook` package. Run these tests and notice that all of them fail. Note that each test identifies a single software bug in the code. **For each test**:
- write the name of the test
- indicate how would fix the software bug identified by that test by writing a correct implementation of the method that contains the bug. Note that it is not necessary to copy the method's documentation (comment statements).

Note that the problem might be with the test rather than the method it is testing. In this case, you should re-write the test. You may use Eclipse to develop your solution but you must make a copy of your work onto this exam paper *before the end of the exam*.

Question 4. Reading Code with Exception Handling.
Consider the following *partial* class implementations. In addition to the methods shown below, you can assume that each class has appropriate constructors.

```java
public class ClassA {

    public void methodA() throws WindException, RainException
    {
        if (conditionOne())
            throw new WindException();
        if (conditionTwo())
            throw new RainException();
        System.out.println("Done method A");
    }

    private boolean conditionOne() {
        return ???;
    }

    private boolean conditionTwo() {
        return ???;
    }
}

public class ClassB {

    public void methodB() throws RainException {
        ClassA myA = new ClassA();
        try {
            myA.methodA();
            System.out.println("Just back from method A");
        } catch (WindException e) {
            System.out.println("Caught WindException in method B");
        } finally {
            System.out.println("Finally in B");
        }

        System.out.println("Now we're done with B");
    }
}

public class ClassC {

    public void methodC() {
        ClassB myB = new ClassB();
        try {
            myB.methodB();
        } catch (RainException e) {
            System.out.println("Caught RainException in method C");
        }
    }
}
```

You may not use Eclipse in any way for this question. Consider the following statements:

```
ClassC myC = new ClassC();
myC.methodC();                    // (***)
```

i) Assuming that methods `conditionOne()` and `conditionTwo()` in `ClassA` both return `false`, what is printed on the screen when the statement marked with (***) at the top of this page executes?

ii) Assuming that method `conditionOne()` returns true and method `conditionTwo()` returns false, what is printed on the screen when the statement marked with (***) at the top of this page executes?

iii) Assuming that method `conditionOne()` returns false and method `conditionTwo()` returns true, what is printed on the screen when the statement marked with (***) at the top of this page executes?

iv) Assuming that methods `conditionOne()` and `conditionTwo()` in `ClassA` both return `true`, what is printed on the screen when the statement marked with (***) at the top of this page executes?

Question 5: Designing Robust Classes (A)

Suppose the cook method of a `Microwave` class has the following specification:

```
// Cook
// Requires: !isDoorOpen()
// Modifies: this
// Effects: microwave is cooking
public void cook() {
    cooking = true;
}
```

Assume that the `Microwave` class has a field of type `boolean` named `cooking`. Redesign the method so that it is more robust. Note that a solution that has the `cook` method silently return (i.e., do nothing) if the door is open is not acceptable. Write a jUnit test class to fully test your redesigned method.   Further assume that the `Microwave` class has the following methods:

```
public boolean isDoorOpen(); // true if door is open,
                             // false otherwise
public boolean isCooking();  // true if microwave is cooking,
                             // false otherwise
public void openDoor();      // opens door and stops cooking
```

Question 6. Designing Robust Classes (B)

Suppose the `installNewFurnace()` method of a `House` class has the following specification:

```
// installNewFurnace
// REQUIRES: !isFurnaceInstalled() and isGasTurnedOff()
// MODIFIES: this
// EFFECTS: records that the furnace has been installed
public void installNewFurnace() {
    furnaceInstalled = true;
}
```

Assume that the `House` class has a field of type `boolean` named `furnaceInstalled`. Further assume that the `House` class has the following methods:

```
public House(); // constructs a new House object
public boolean isGasTurnedOff(); // true if gas is off,
                                 // false otherwise
public boolean isFurnaceInstalled(); // true if house has had
      // the furnace installed, false otherwise
public void setFurnaceInstalled(boolean installed); // if installed
      // is true, furnace has been installed, otherwise furnace
      // not installed
public void turnGasOnOrOff(Boolean onOnOff);  // turns natural
      // gas on if onOrOff is true, turns gas off otherwise
```

a) **Robustness**

   Redesign the method so that it is more robust. Note that a solution that has the
   `installNewFurnace()` method silently return (i.e., do nothing) if the natural gas is on is not
   acceptable. A solution that silently installs a second furnace is also not acceptable.

b) **Testing**

   Write a JUnit test class to fully test your redesigned method.

Question 7: Type Hierarchies and Substitutability
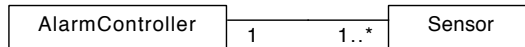
Given the classes `Monitor` and `MegaMonitor`:

```java
public class Monitor {

    /*
     * EFFECTS: returns the horizontal resolution as a number >= 1024
     */
    public int getHorizontalResolution() {
        // …
    }
}

public class MegaMonitor extends Monitor {

    /*
     * EFFECTS: returns the horizontal resolution as a number >= 4096
     */
    public int getHorizontalResolution() {
        // …
    }
}
```

Where relevant, you must explain your answer to the questions below in the context of the Liskov Substitution Principle.

a) Can you use a `MegaMonitor` object as a substitute for a `Monitor` object?

b) Can you use a `Monitor` object as a substitute for a `MegaMonitor` object?

Question 8. Interpreting UML Class Diagrams

For each of the UML class diagrams shown below, indicate the name used to describe the relationship (e.g., an inheritance relationship) between the classes (or instances of those classes) and describe *in point form* what the diagram communicates about the relationship.

```
┌──────────────────┐                    ┌──────────┐
│ AlarmController  │────────────────────│  Sensor  │
└──────────────────┘  1          1..*   └──────────┘
```

i.   Name used to describe this relationship: _____

     Point form description of what the above diagram communicates about relationship:

```
┌──────────┐              ┌──────────────┐
│ Faculty  │◇─────────────│  Department  │
└──────────┘          *   └──────────────┘
```
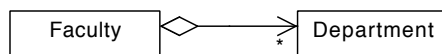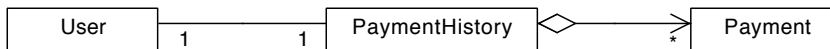
ii. Name used to describe this relationship: _____

     Point form description of what the above diagram communicates about relationship:

Question 9. Implementing an Object-Oriented Design.

a)   Consider the UML class diagram shown below:

```
┌────────┐              ┌──────────────────┐              ┌───────────┐
│  User  │──────────────│  PaymentHistory  │◇─────────────│  Payment  │
└────────┘  1        1  └──────────────────┘          *   └───────────┘
```

It represents the design for a small part of an online storage system. Users have to pay for the service and a history of payments is maintained in the system. Write the code for the `PaymentHistory` class. You must include fields and methods that are necessary to support relationships between the other classes shown on the UML diagram but it is not necessary to include any others. Assume that you can add a `Payment` to the `PaymentHistory` but a `Payment` cannot be removed, and that the `PaymentHistory` must not contain duplicate `Payment` objects. Assume that the constructor of the `User` class creates the corresponding `PaymentHistory` object.

b) Choose a suitable data structure from the Java Collections Framework (JCF) for each of the following problems:

i. You are writing a system to manage a hockey pool. For each participant in the pool, you must be able to track a team of players. What data structure will you use to represent the team of players? Why?

ii. You are writing a system to model line-ups at the bank. Each teller has their own line-up. What data structure will you use to store all the people in line at all of the tellers?