CPSC 210

Sample Midterm #2 Exam Questions

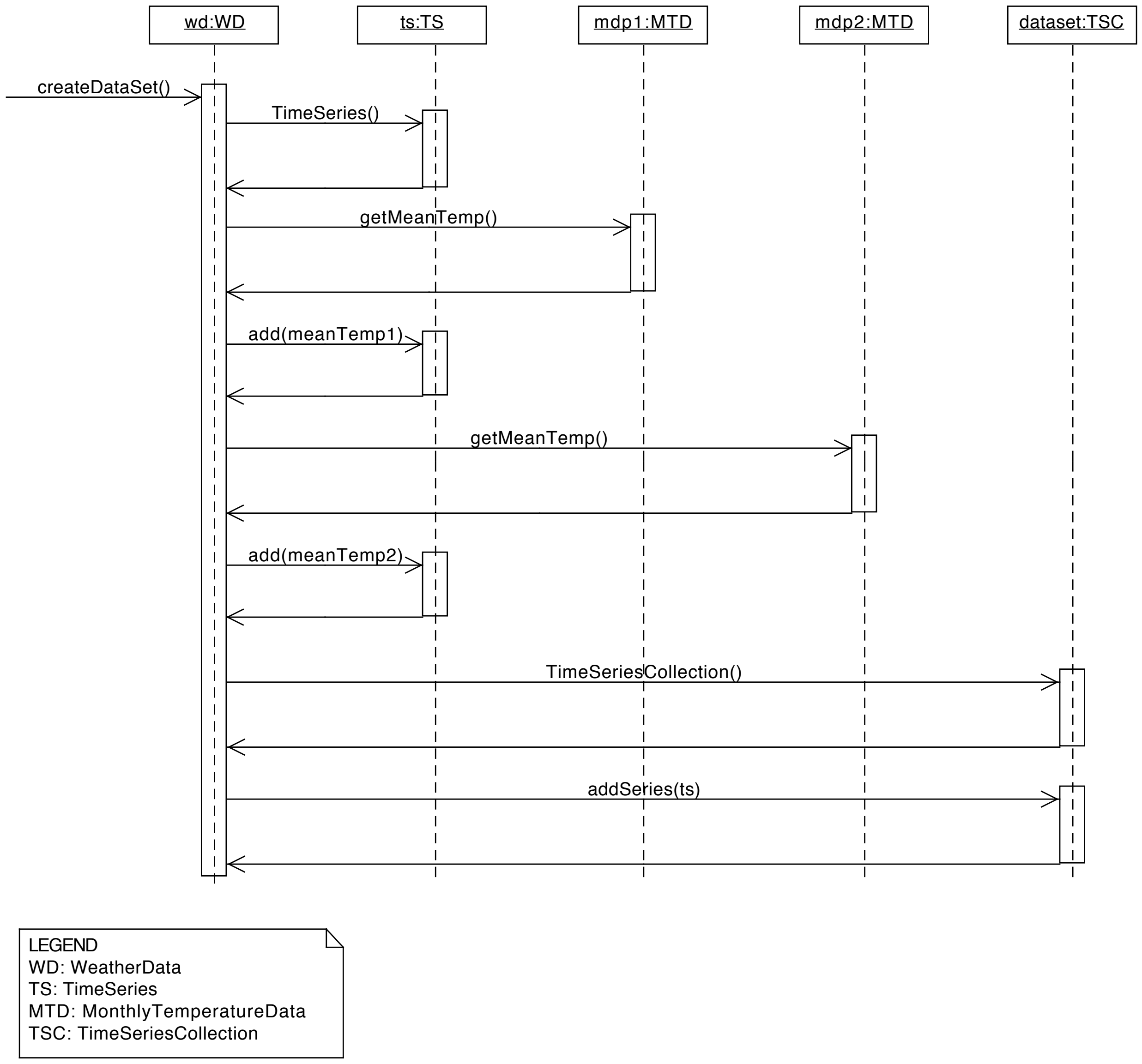
Note: the questions in this document do not constitute an actual midterm. However, many of the questions are taken from actual midterm exams and are therefore representative of the kinds of questions that could be asked. Please keep in mind that this set of questions does not exhaust all the possibilities and therefore should not be used as your primary source of study material.

You will need to check out the following systems from the lectures repository:

`EmailManager`

Question 1. UML Sequence Diagram

Consider the UML Sequence Diagram given below. **Sketch** the code implementing the method `createDataset()` on a `WeatherData` object that is described by this diagram. You may need to assume where there are likely return values from calls to methods. Handle these return values in the code you sketch. You can add comments to explain your code. We will **not** be grading for correct Java syntax.



Note: we include parameters for method calls on this sequence diagram to provide sufficient detail for you to build the code from this model. When drawing your own sequence diagrams on exams, do not include parameters unless you are told to do so, or you need to distinguish between overloaded methods.

Note: Question 2 refers to the `EMailManager` project checked out of the repository.

Question 2: Control & Data Models

Draw a sequence diagram for the method `AddressBook.addGroup` within the package `ca.ubc.cs.cpsc210.addressbook`. If you have to loop over a collection, you must assume that there are exactly two objects in that collection. Include calls to *all* methods in the `EMailManager` project *except* constructors. You must also include the first level of calls to the Java library, if any. Be sure to include a legend if you abbreviate class names.

<u>Question 3. Reading Code with Exception Handling.</u>
Consider the following *partial* class implementations. In addition to the methods shown below, you can assume that each class has appropriate constructors.

```
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 IntelliJ 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 4: Designing Robust Classes

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 5. Designing Robust Classes

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:

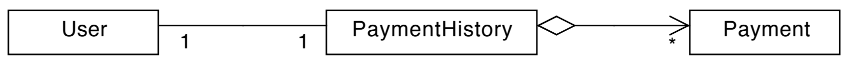
```
public House(); // constructs a new House object
public boolean isGasTurnedOff(); // true if gas is off,
                                 // false otherwise
public boolean isFurnaceInstalled(); // true if house has had
       // the furnace installed, false otherwise
public void setFurnaceInstalled(boolean 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 6. Implementing an Object-Oriented Design

Consider the UML class diagram shown below:



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. Further assume that the payments must be stored in the order in which they were added and that the `PaymentHistory` must not contain duplicate `Payment` objects. Assume that the constructor of the `User` class creates the corresponding `PaymentHistory` object.

Question 7. Object-Oriented Design - Modelling

Draw a UML class diagram to represent the design of all the types in the `addressBook` and `email` packages of the `EMailManager` system checked out from the lectures repository.

Question 8. Object-Oriented Design - Modelling

a) Draw a UML class diagram to represent the design of all the types in the `model` package of the `SnakeStarter` project checked out of the labs repository for Lab 1. It is recommended that you check out a new copy of this code and assume that the `Food` class has been implemented according to the given specification.

b) Draw a UML sequence diagram to model the call to `Snake.move`. Include the first level of calls to the Java library. Model only the code found in the *first* case in any `switch` statement that you encounter.