CPSC 210

Sample Midterm Exam Questions - Solutions

Question 1. Reading Code with Exception Handling.

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?

```
Done method A
Just back from method A
Finally in B
Now we're done with B
```

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?

```
Caught WindException in method B
Finally in B
Now we're done with B
```

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?

```
Finally in B
Caught RainException in method C
```

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?

```
Caught WindException in method B
Finally in B
Now we're done with B
```

Question 2: Designing Robust Classes

```
// Modifies: this
// Effects: if !isDoorOpen(), microwave is cooking;
// otherwise DoorException is thrown
public void cook() throws DoorException {
    if(!isDoorOpen())
        cooking = true;
    else
        throw new DoorException("Door is open!");
}

// unit tests
public class TestMicrowave {

    @Test
    public void testCookWithDoorClosed() {
        try {
            mw.cook();
            assertTrue(mw.isCooking());
        } catch(DoorException e) {
            fail("Door exception was thrown");
        }
    }

    @Test (expected = DoorException.class)
    public void testCookWithDoorOpen() throws DoorException {
        mw.openDoor();
        mw.cook();
        fail("Door exception should have been thrown");
    }
}
```

Question 3. Designing Robust Classes

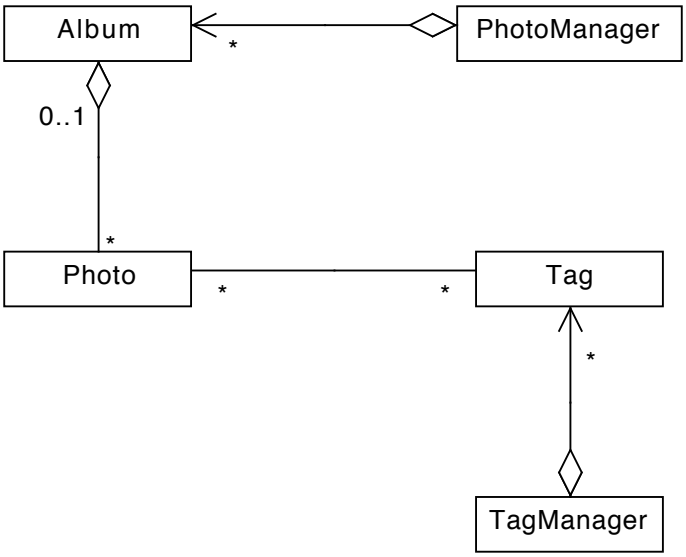
a) Redesign the method so that it is more robust. Note that a solution that has the `installNewFurnace()` method silently return (i.e., do nothing) if the natural gas is on is not acceptable. A solution that silently installs a second furnace is also not acceptable.

```
// MODIFIES: this
// EFFECTS: If a furnace has already been installed, throw a
//   FurnaceInstalledException.  If no furnace has been
//   installed and the gas is turned off, install the furnace.
//   If no furnace has been installed and the gas is turned on
//   throw a GasOnException.
public void installNewFurnace()
       throws FurnaceInstalledException, GasOnException {

   if (isFurnaceInstalled())
     throw new FurnaceInstalledException();

   if (isGasTurnedOff())
      furnaceInstalled = true;
   else
      throw new GasOnException();
}
```

b) Write a jUnit test class to fully test your redesigned method.

```
public class HouseTest {
   private House aHouse;

   @Before
    public void setUp() {
        aHouse = new House();
   }

   @Test
   public void testInstallFurnaceAllOK() {
        aHouse.setFurnaceInstalled(false);
        aHouse.turnGasOnorOff(false);
        try {
           aHouse.installNewFurnace();
           assertTrue(aHouse.isFurnaceInstalled());
        } catch (GasOnException e) {
           fail("Gas on exception thrown!");
        } catch (FurnaceInstalledException e) {
            fail("Furnace Installed Exception thrown!");
       }
   }



   @Test (expected = FurnaceInstalledException.class)
   public void testInstallFurnaceTwice()
               throws FurnaceInstalledException, GasOnException {
        aHouse.setFurnaceInstalled(false);
        aHouse.turnGasOnorOff(false);
        aHouse.installNewFurnace();
        assertTrue(aHouse.isFurnaceInstalled());
        aHouse.installNewFurnace();
        fail("FurnaceInstalledException should have been thrown'");
   }

   @Test (expected = GasOnException.class)
   public void testInstallFurnaceWithGasOn()
               throws FurnaceInstalledException, GasOnException {
       aHouse.setFurnaceInstalled(false);
       aHouse.turnGasOnorOff(true);
       aHouse.installNewFurnace();
       fail("GasOnException should have been thrown");
   }

   @Test (expected = FurnaceInstalledException.class)
   public void testInstallFurnaceTwiceWithGasOn()
               throws FurnaceInstalledException, GasOnException {
       aHouse.setFurnaceInstalled(true);
       aHouse.turnGasOnorOff(true);
       aHouse.installNewFurnace();
       fail("FurnaceInstallException should have been thrown");
   }
}
```

Question 4: Extracting a UML Diagram



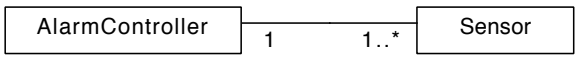
Question 5: Type Hierarchies and Substitutability

a) A **`MegaMonitor`** object can be used as a substitute for a **`Monitor`** object. The Liskov Substitution Principle holds as

- the precondition on **`MegaMonitor.getHorizontalResolution`** is the same as that on **`Monitor.getHorizontalResolution`**
- the postcondition on **`MegaMonitor.getHorizontalResolution`** is stronger than that on **`Monitor.getHorizontalResolution`** because a smaller set of values is produced by the overridden method in the subclass

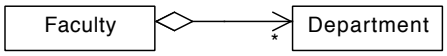
b) A **`Monitor`** object cannot be used as a substitute for a **`MegaMonitor`** object as **`Monitor`** is not a subclass of **`MegaMonitor`**.

Question 6. Interpreting UML Class Diagrams



i. Name used to describe this relationship: **bidirectional association**

Point form description of what diagram communicates about relationship between classes:
- each `AlarmController` object is associated with at least one `Sensor` object
- each `Sensor` object is associated with only one `AlarmController` object
- `AlarmController` can access ("knows about") services provided by the `Sensor` class and vice-versa



ii. Name used to describe this relationship: **aggregation**

Point form description of what diagram communicates about relationship between classes:

- each `Faculty` object is associated with many `Department` objects
- we consider `Faculty` to be the "whole" and `Department` objects to be the "parts"
- `Faculty` can access ("knows about") services provided by the `Department` class but *not* vice-versa

Question 7. Implementing an Object-Oriented Design.

a)

```
public class PaymentHistory {
   private Collection<Payment> payments;
   private User user;

   public PaymentHistory(User user) {
      this.user = user
      payments = new HashSet<Payment>();
   }

   public void addPayment( Payment p ) {
      payments.add(p);
   }
}
```

Note: it is also acceptable to declare the `payments` field to be of type `Set<Payment>`.
Note2: this solution assumes that the user associated with a `PaymentHistory` object cannot be changed after the `PaymentHistory` object has been constructed.

b)

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?

`HashSet<Player>`

We assume we have a `Player` class to represent a hockey player. We won't want to add a particular player to the team more than once and so we use a `Set` which does not allow for duplicate entries. `HashSet` is an implementation of the `Set` interface which provides efficient implementations of the `add`, `remove` and `contains` methods - all in `O(1)` time.

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?

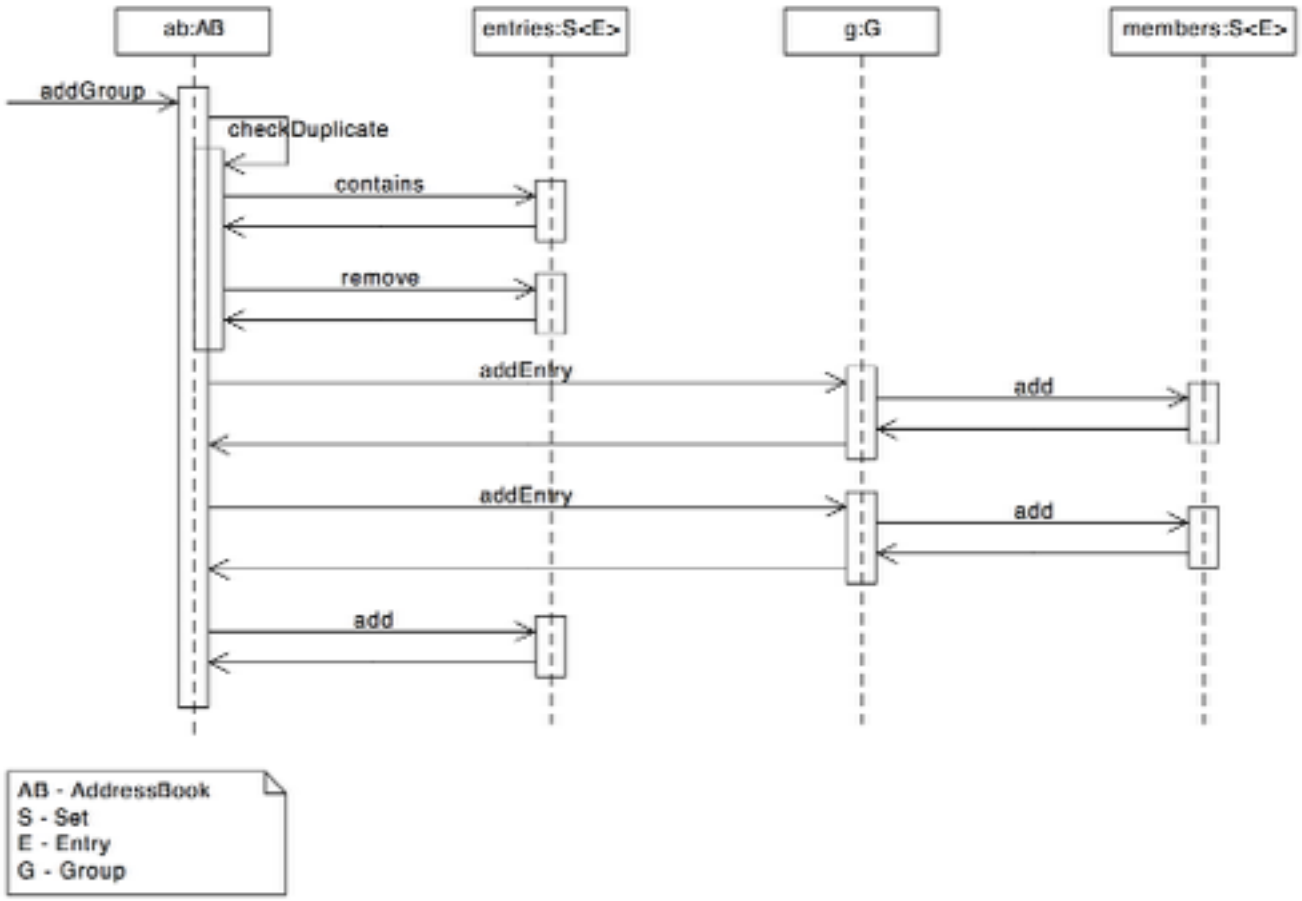
`ArrayList<LinkedList<Customer>>`

We assume that we have a `Customer` class to represent a customer at the bank. We represent each line-up using a `LinkedList` as `LinkedList` implements the `Queue` interface and can therefore be used to maintain customers in first-in, first-out (FIFO) order. Given that there is more than one teller (and therefore more than one line-up), we use an `ArrayList` to store each of the `LinkedLists`.

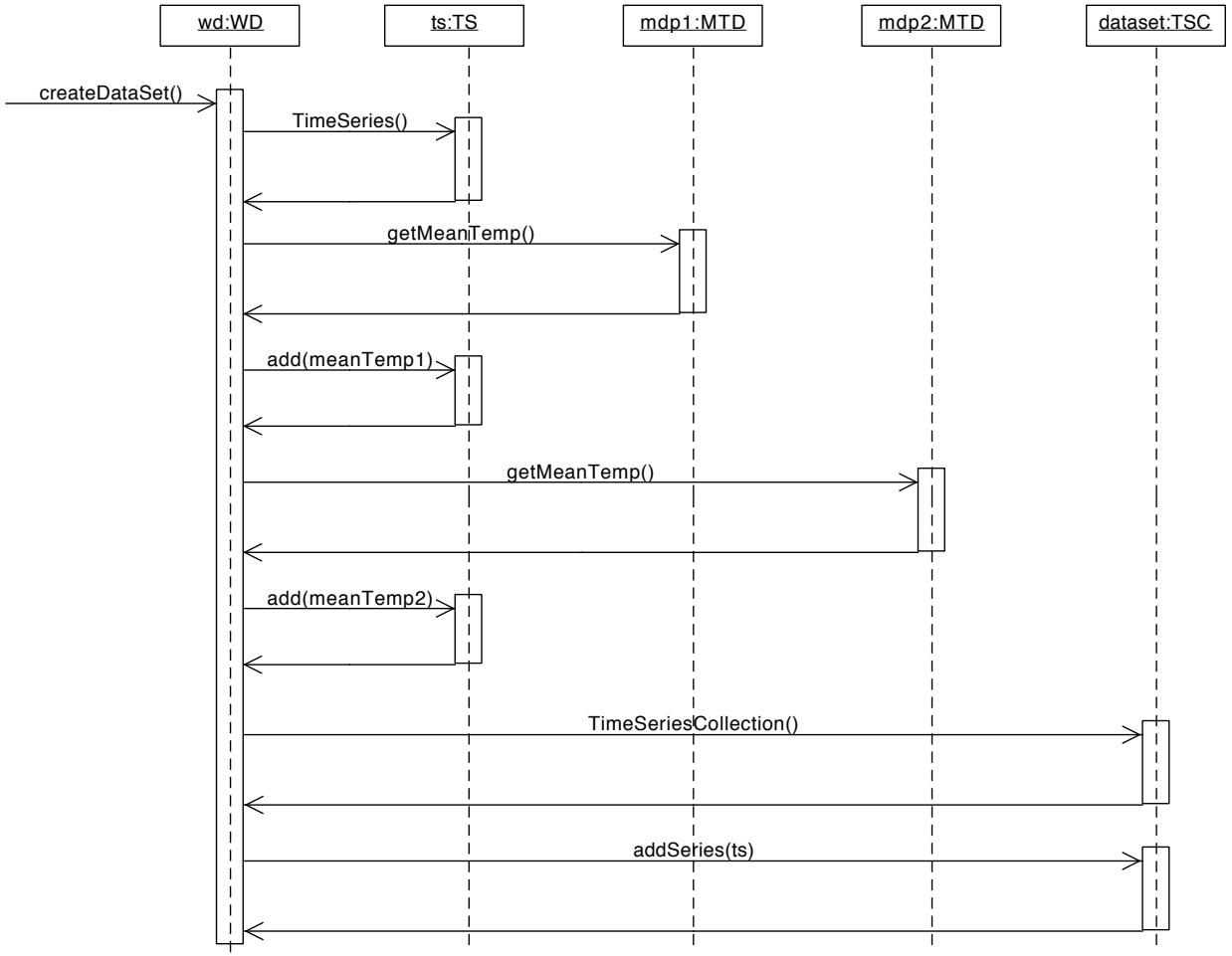
Question 8. Using the Java Collections Framework

See **`HRSystemComplete`** in lectures repository

Question 9: Extracting a UML Sequence Diagram



Question 10. Implementing a UML Sequence Diagram



LEGEND
WD: WeatherData
TS: TimeSeries
MTD: MonthlyTemperatureData
TSC: TimeSeriesCollection

```
createDataSet() {
  ts = new TimeSeries();
  meanTemp1 = mdp1.getMeanTemp();
  ts.add(meanTemp1);
  meanTemp2 = mdp2.getMeanTemp();
  ts.add(meanTemp2);
  dataset = new TimeSeriesCollection();
  dataset.addSeries(ts);
}
```