The University of British Columbia

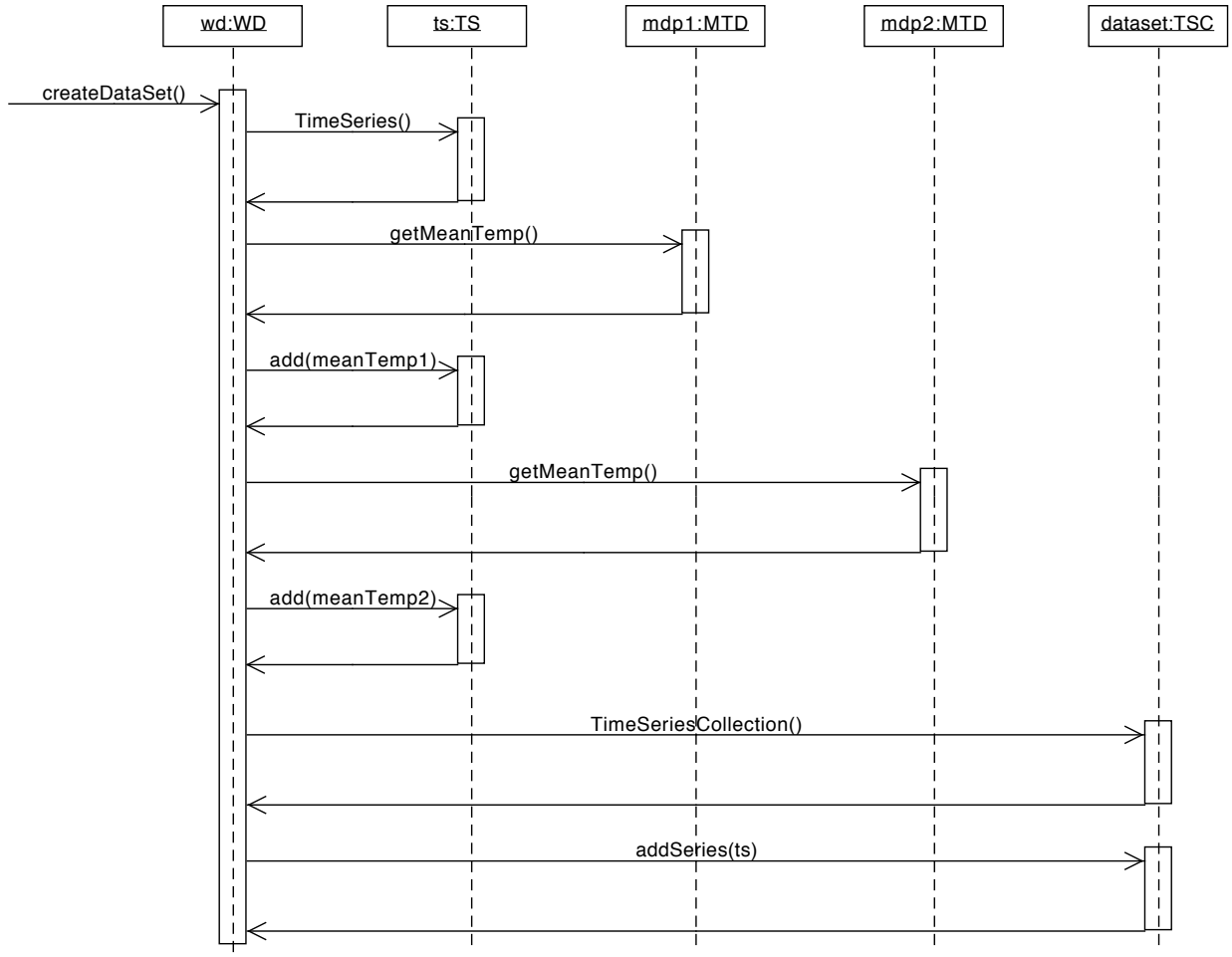
CPSC 210

Sample Midterm Exam Questions (SOLUTION)

Please don't look at these solutions until you have put significant effort into coming up with your own. The midterm exam will not ask you to understand a solution that has been presented to you. You need to practice doing what the exam will ask you to do – construct your own solution!

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.

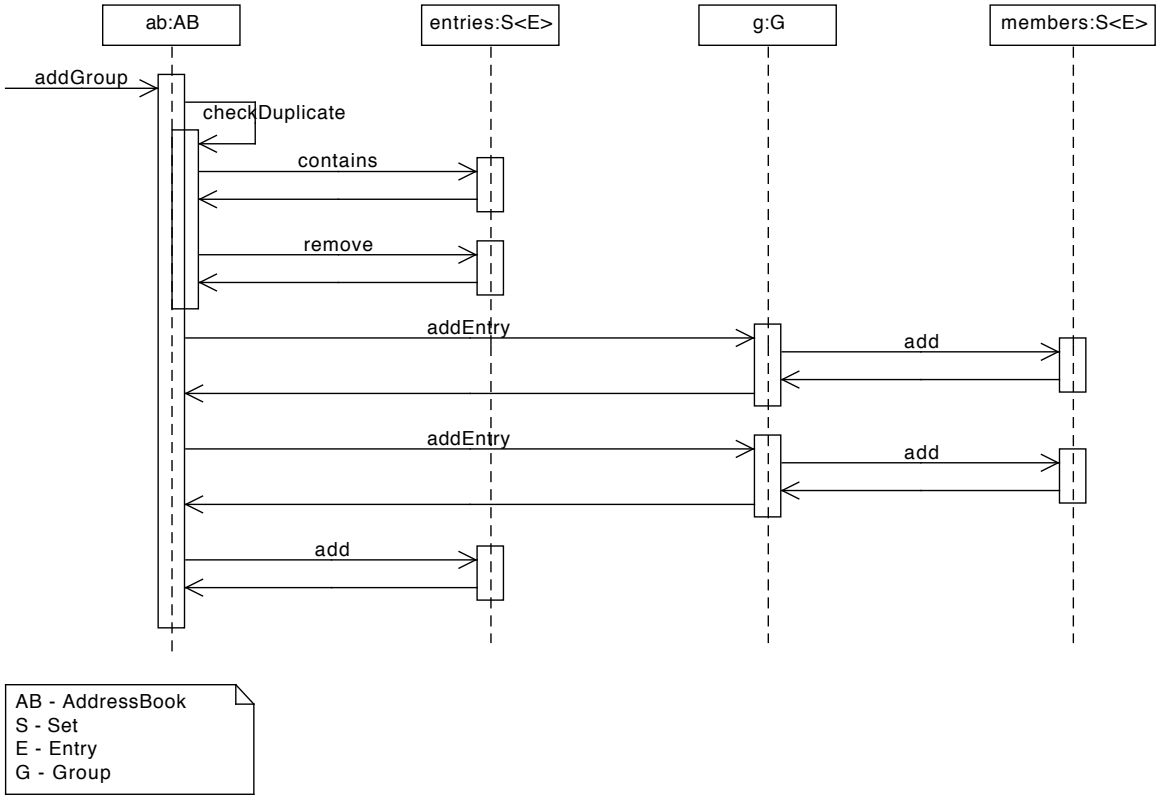


LEGEND
WD: WeatherData
TS: TimeSeries
MTD: MonthlyTemperatureData
TSC: TimeSeriesCollection

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

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.



Question 3. 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 4: 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 5. 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();
    }

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

    @Test (expected = FurnaceInstalledException.class)
    public void testInstallFurnaceTwiceWithGasOn()
                throws FurnaceInstalledException, GasOnException {
        aHouse.setFurnaceInstalled(true);
        aHouse.turnGasOnorOff(true);
        aHouse.installNewFurnace();
    }
}
```

Question 6. Implementing an Object-Oriented Design

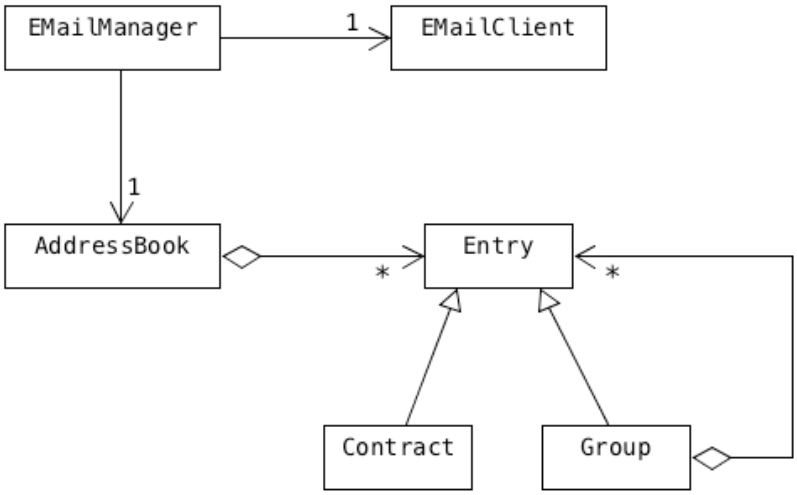
```
public class PaymentHistory {
   private List<Payment> payments;
   private User user;

   public PaymentHistory(User user) {
      this.user = user;
      payments = new LinkedList<>();  // or ArrayList
   }

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

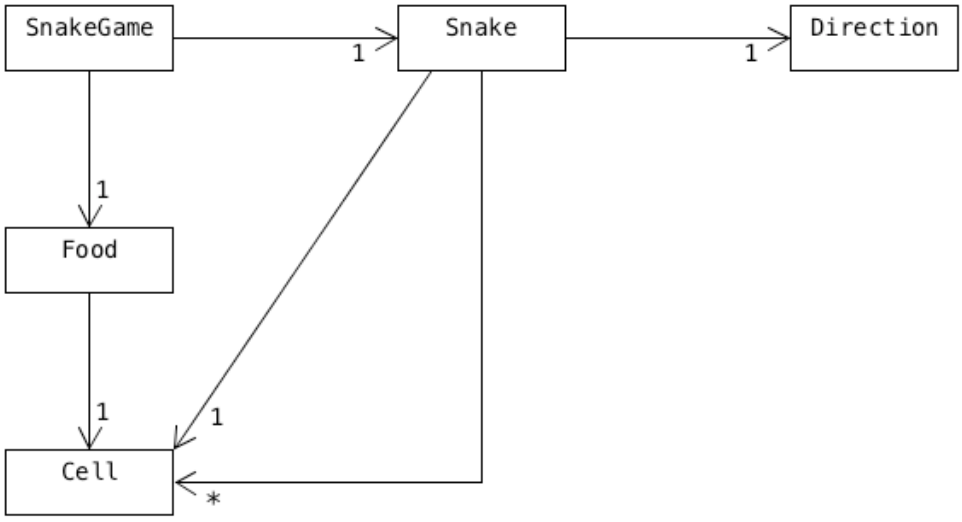
Note: this solution assumes that the user associated with a `PaymentHistory` object cannot be changed after the `PaymentHistory` object has been constructed.

Question 7. Object-Oriented Design - Modelling

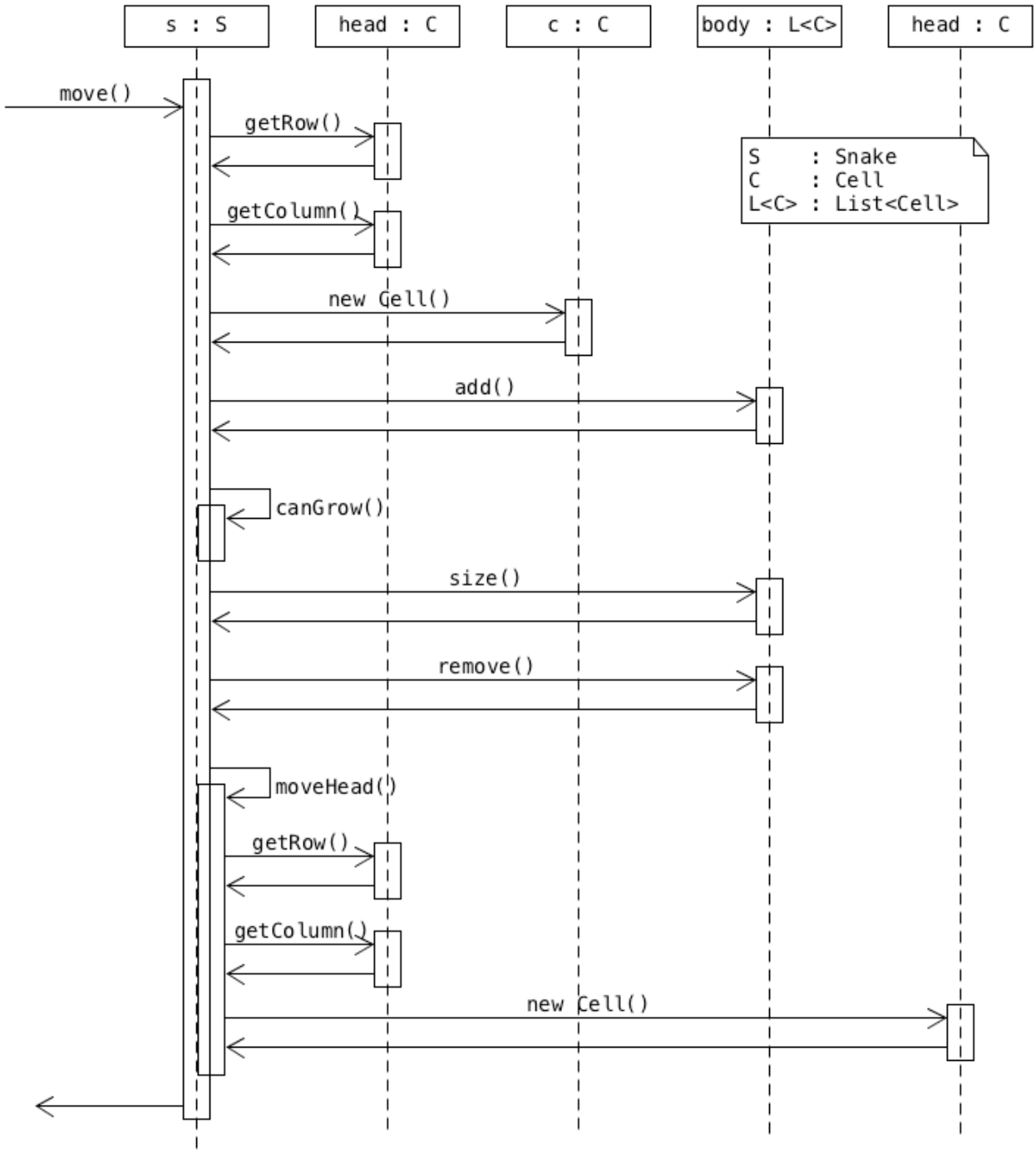


Question 8. Object-Oriented Design - Modelling

a)



b)


s : S
head : C
c : C
body : L<C>
head : C
move()
getRow()
getColumn()
new Cell()
add()
canGrow()
size()
remove()
moveHead()
getRow()
getColumn()
new Cell()
S : Snake
C : Cell
L<C> : List<Cell>