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 constructing your own. The midterm exam will not ask you to understand a solution that has been presented to you – it *will* ask you to come up with your own.

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.

Given that negative payment numbers appear to be generated only for `Cash` type payments, we generate the following hypotheses:

(1) payment numbers are generated incorrectly when the type is `Cash` payment
(2) payment numbers are printed incorrectly when the type is `Cash` payment
(Extra credit.) What is actually causing the error in the output shown above?

The second hypothesis above is correct. The error is in the `toString` method of the `PaymentRecord` class:

```
if (typeOfPayment.equals("Cash"))
    repesentationAsString =
        repesentationAsString.concat(paymentNumber-100 + ", ");
```

should be:

```
if (typeOfPayment.equals("Cash"))
    repesentationAsString =
        repesentationAsString.concat(paymentNumber + ", ");
```

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*.

```
@Test
public void testUseFreeDrinkNoneAvailable() {
      assertFalse(card.useFreeDrink());
}
@Test
public void testUseFreeDrinkWhenAvailable() {
      // add credits to purchase enough drinks to earn a free one
      card.topUp(CoffeeCard.BEANS_PER_FREE_DRINK);

      // buy enough drinks to earn a free one
      for (int i = 0; i < CoffeeCard.BEANS_PER_FREE_DRINK - 1; i++) {
            assertTrue(card.purchaseDrink(1));
      }

      assertEquals(1, card.getFreeDrinks());   // not strictly needed
      assertTrue(card.useFreeDrink());
      assertEquals(0, card.getFreeDrinks());
}
```

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*.

testOneParamConstructor – bug is in the test

```
public void testOneParamConstructor() {
      Contact c = new Contact("Joey");
      assertEquals("Joey", c.getName());
}
```

testTwoParamConstructor - bug is in the constructor

```
public Contact(String name, String eMailAddress) {
      super(name);
            this.eMailAddress = eMailAddress;
}
```

testGetAddressList – bug is in the `getAddressList` method

```
public List<String> getAddressList() {
      List<String> al = new LinkedList<String();
      al.add(eMailAddress);
      return al;
}
```

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.

```
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?

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 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
```

```
// 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 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.

```
// 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) Testing

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 7: Type Hierarchies and Substitutability

Given the classes `Monitor` and `MegaMonitor`:

```
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?

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

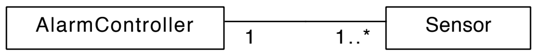
- the precondition on **`MegaMonitor.getHorizontalResolution`** is the same as that on **`Monitor.getHorizontalResolution`**
- the postcondition on **`MegaMonitor.getHorizontalResolution`** is stronger than that on **`Monitor.getHorizontalResolution`** because a smaller set of values is produced by the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

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

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

Question 8. Interpreting UML Class Diagrams

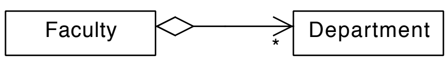
For each of the UML class diagrams shown below, indicate the name used to describe the relationship (e.g., an inheritance relationship) between the classes (or instances of those classes) and describe *in point form* what the diagram communicates about the relationship.



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

Point form description of what diagram communicates about relationship:

- 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



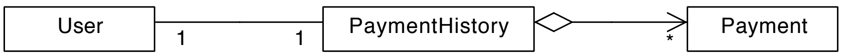
ii. Name used to describe this relationship: **aggregation**

Point form description of what diagram communicates about relationship:

- **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 9. Implementing an Object-Oriented Design.

a) 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, and that the `PaymentHistory` must not contain duplicate `Payment` objects. Assume that the constructor of the `User` class creates the corresponding `PaymentHistory` object.

```
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) 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?

`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`.