McGill University
COMP 303 – Software Development
Midterm Examination 1 – Winter 2016

Name: _____

Student Number: _____

Signature: _____

You have **80 minutes** to write this examination.

Do not start until you are informed to.

Do not continue after you are informed to stop.

Your answers must be **concise**, **clear**, and **precise**. Long-winded, vague, and/or unclear answers will not get full marks.

No notes, books, or any type of electronic equipment is allowed.

Please **write legibly**. No marks can be attributed to undecipherable answers.

If you believe the requirements stated in a question are ambiguous, you are required to state the ambiguity, state the assumption that you are going to make and then proceed to answer the question.

The appendix at the end of this booklet contains a partial selection of API documentation. If you require an API element that is not mentioned but do not remember its exact name, simply make up the closest approximation you can.

Good luck!

Question	Mark
1	/10
2	/16
3	/12
4	/12
Total (/50)	

Academic Integrity

McGill University values academic integrity. Therefore all students must understand the meaning and consequences of cheating, plagiarism and other academic offenses under the Code of Student Conduct and Disciplinary Procedures.

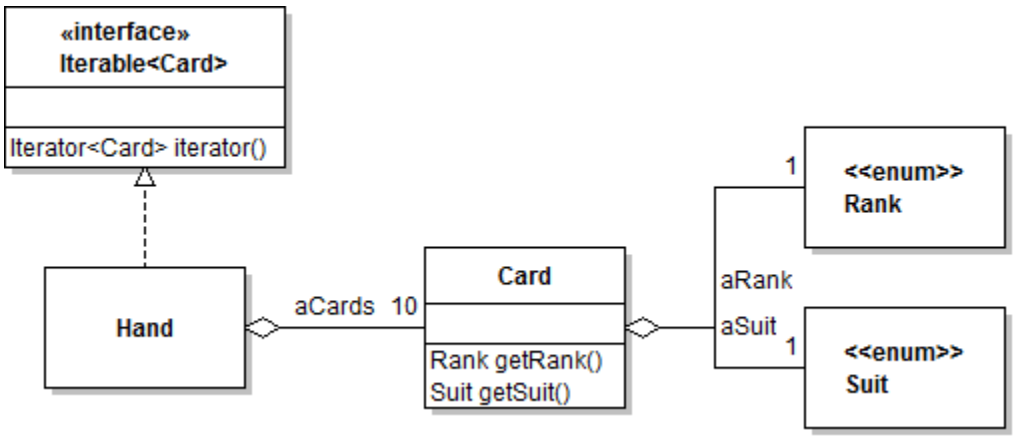
Question 1 [10 marks]

Objects of class Hand aggregate exactly 10 objects of class `Card`. Implement the mechanism necessary to support sorting hands using the `Arrays.sort` functionality of the JDK.

```
Hand[] hands = // Correctly initialized array of 8 complete hands of 10 cards.
Arrays.sort... // There are different possible ways of completing this call.
```

The required behavior for comparing hands is that hands should be ordered in terms of number of cards of a certain rank. Clients should be able to compare hands by number of aces, or number kings, or number of fours, etc. For example, if the client chooses to compare hands by number of aces, a hand with one ace should come *before* a hand with two aces. If two hands have the same number of aces, they should be considered equal and their order does not matter. The same logic applies to any rank.

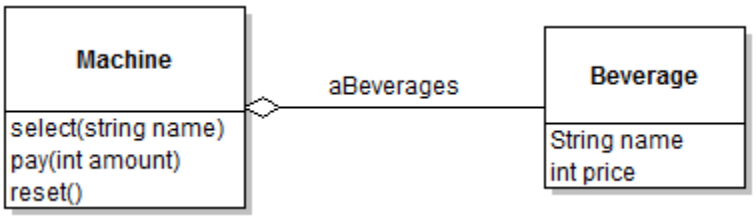
To answer this question, complete the UML diagram with all relevant elements, and write the code of the method or methods that implement the actual comparison. Your solution should include, among others, the STRATEGY design pattern and an "object factory" method.



```
public class Hand
{
	private Card[] aCards = new Card[10];
	// Assume the cards are initialized somehow.
	// You don't have to provide the code to do this.









}
```

Question 2 [16 marks]

You are designing the software for a vending machine able to dispense beverages in exchange for coins. The following class diagram shows a partial solution.



a) If you implement this design in a way that makes *instances* of class `Beverage` play the role of flyweight objects in the FLYWEIGHT design pattern, which of the following characteristic(s) will they *certainly* have (assuming a correct implementation). Circle all that apply:

immutable unique well-encapsulated shared static tested

b) Write the code necessary to make `Beverage` instances flyweight objects by realizing the FLYWEIGHT design pattern in a way such that all the required implementation is localized in class `Beverage`. Assume that the price of an item can be somewhat obtained through the static call `Catalog.getPrice(String name)`.

```
public class Beverage
{
      private final String aName;  // assume unique in the software
      private final int aPrice;
```

c) The machine works as follows:

- **`select(name)`** selects a beverage. The selection **times out** after 30 seconds of inactivity.
- **`pay(amount)`** adds a number of cents into the machine in one atomic operation. This amount may be inferior, equal, or superior to the price of the selection. It is **not possible** to pay an amount if there is no currently selected drink. If the amount paid is inferior to the price of the selected beverage, a balance is accumulated. If the amount is superior or equal to the price, the beverage is returned, the selection is erased, and any change remaining is returned. Once a balance exists, it is **not possible** to change the selection.
- **`reset()`** erases the selection and returns any balance accumulated.

Draw a *state diagram* that provides an overview of all the necessary abstract states for an instance of the `Machine` class. Include all legal transitions along with their condition(s) and action(s), if applicable, and omit any invalid transition. Assume the machine is initially in a state with no selection and no balance.

Question 3 [12 marks]

You are in charge of testing the following code:

```
public static int countAces(List<Card> pCards)
{
   int total = 0;
   for( Card card : pCards )
   {
      if( card.getRank() == Rank.ACE )
      {
         total++;
      }
   }
   return total;
}
```

a) Transform (re-write) the code into a more basic form that does not use the "for-all" loop construct and makes its use of the ITERATOR pattern more explicit in the code.

b) Draw a control flow graph of the (transformed) method. Hint: the start node should have the name of the method ("`countAces`") and there is a single end node. Label nodes (basic blocks) with all statements in the block (you can abbreviate as long as there is no ambiguity)

c) Write the code of a *JUnit test method* that tests the above method and achieves branch coverage. Assume the tested method is available as `Util.countAces(List<Card>)` . Include all necessary constructs. Assumes the `Card` class has a constructor `Card(Rank,Suit)` where `Rank` and `Suit` are enums with appropriate names. Assume all necessary JUnit elements are properly imported.

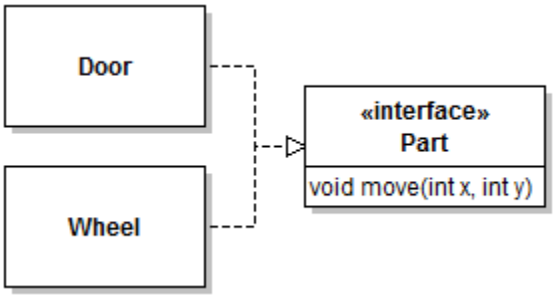
Question 4 [12 marks]

In a software system used for the computer-assisted design (CAD) of cars, it is possible for users to manipulate and draw car parts (instances of the `Part` class). You need to extend the design so that it is also possible to manipulate `Assembly` instances. Assemblies should be a collection of parts that can be treated as a `Part` by the CAD system.

a) Which design pattern will best support the above requirement? Circle one.

COMMAND COMPOSITE DECORATOR FLYWEIGHT ITERATOR SINGLETON STRATEGY

b) Add the `Assembly` class to the class diagram below, as well as any additional model element (nodes, edges, attributes, methods, etc.) that is required to fully support the requirement above.



c) Complete the declaration of the `Assembly` class and add the code for any field(s) and method(s) that correspond to necessary elements in your UML Class Diagram.

```
public class Assembly
```

Appendix – Selected API Documentation

method void Arrays.sort(Object[] a)

Sorts the specified array of objects into ascending order, according to the natural ordering of its elements. All elements in the array must implement the Comparable interface. Furthermore, all elements in the array must be *mutually comparable* (that is, e1.compareTo(e2) must not throw a ClassCastException for any elements e1 and e2 in the array).

method void Arrays.sort(T[] a, Comparator<? super T> c)

Sorts the specified array of objects according to the order induced by the specified comparator. All elements in the array must be *mutually comparable* by the specified comparator (that is, c.compare(e1, e2) must not throw a ClassCastException for any elements e1 and e2 in the array).

interface Comparable<T>

This interface imposes a total ordering on the objects of each class that implements it. This ordering is referred to as the class's *natural ordering*, and the class's compareTo method is referred to as its *natural comparison method*. The natural ordering for a class C is said to be *consistent with equals* if and only if e1.compareTo(e2) == 0 has the same boolean value as e1.equals(e2) for every e1 and e2 of class C. Note that null is not an instance of any class, and e.compareTo(null) should throw a NullPointerException even though e.equals(null) returns false.

method int Comparable<T>.compareTo(T o)

Compares this object with the specified object for order. Returns a negative integer, zero, or a positive integer as this object is less than, equal to, or greater than the specified object.

interface Comparator<T>

A comparison function, which imposes a *total ordering* on some collection of objects. Comparators can be passed to a sort method (such as Collections.sort or Arrays.sort) to allow precise control over the sort order. Comparators can also be used to control the order of certain data structures (such as sorted sets or sorted maps), or to provide an ordering for collections of objects that don't have a natural ordering. The ordering imposed by a comparator c on a set of elements S is said to be *consistent with equals* if and only if c.compare(e1, e2)==0 has the same boolean value as e1.equals(e2) for every e1 and e2 in S.

method int Comparator<T>. compare(T o1, T o2)

Compares its two arguments for order. Returns a negative integer, zero, or a positive integer as the first argument is less than, equal to, or greater than the second. In the foregoing description, the notation sgn(*expression*) designates the mathematical *signum* function, which is defined to return one of -1, 0, or 1 according to whether the value of *expression* is negative, zero or positive. The implementor must ensure that sgn(compare(x, y)) == -sgn(compare(y, x)) for all x and y. (This implies that compare(x, y) must throw an exception if and only if compare(y, x) throws an exception.) The implementor must also ensure that the relation is transitive: ((compare(x, y)>0) && (compare(y, z)>0)) implies compare(x, z)>0. Finally, the implementor must ensure that compare(x, y)==0 implies that sgn(compare(x, z))==sgn(compare(y, z)) for all z. It is generally the case, but *not* strictly required that (compare(x, y)==0) == (x.equals(y)). Generally speaking, any comparator that violates this condition should clearly indicate this fact. The recommended language is "Note: this comparator imposes orderings that are inconsistent with equals."