McGill University
COMP 303 – Software Development
Midterm Examination 1 – Fall 2014

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	/20
2	/20
3	/20
4	/25
5	/15
Total (/100)	

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.

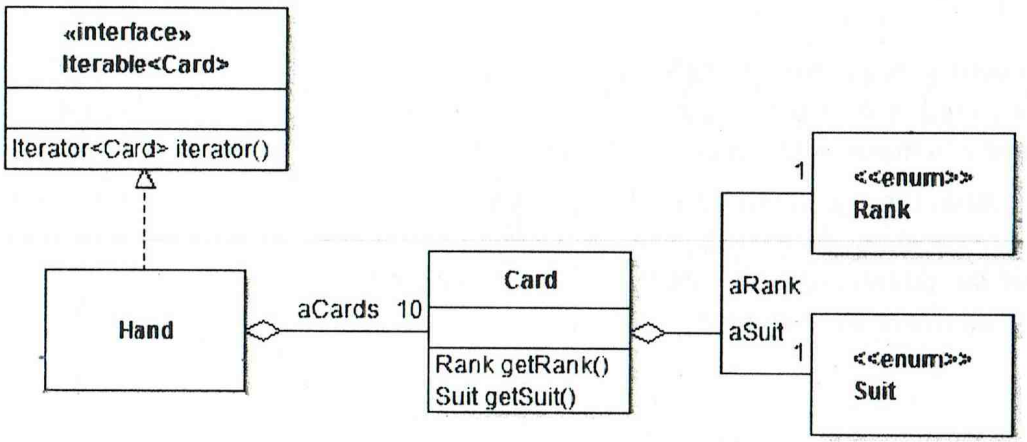
Question 1 [20 marks]

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.

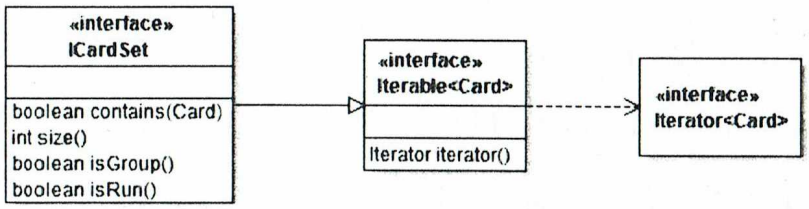
To answer this question, complete the UML diagram with all relevant elements, and write the code of the method or methods that implement the actual comparison. Your solution should include, among others, the strategy design pattern and a factory method.



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









}
```

Question 2 [20 marks]

Part of your project required you to implement method `automatch()` in class `Hand`. The following code and diagram is a *simplified version* of some of the relevant structures. Read it carefully.

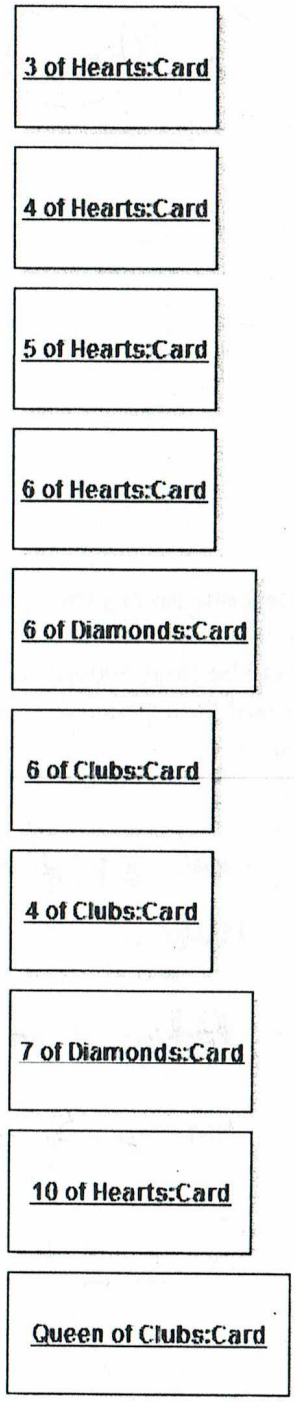



```
/**
 * Models a hand of 10 cards. The hand is not necessarily sorted.
 * The hand is a set: adding the same card twice will not add duplicates
 * of the card. */
public class Hand
{
	/** Creates a new, empty hand. */
	public Hand() // your implementation

	/** Adds pCard to the list of unmatched cards...
	  * @pre !isComplete()
	  */
	public void add( Card pCard ) // some implementation

	/** @return True if the hand is complete. */
	public boolean isComplete() // some implementation

	/** Removes all the cards from the hand. */
	public void clear() // some implementation

	/** @return A copy of the set of matched sets */
	public Set<ICardSet> getMatchedSets() // some vimplementation

	/** @return A copy of the set of unmatched cards. */
	public Set<Card> getUnmatchedCards() // some implementation

	/** @return The number of cards in the hand. */
	public int size() // some implementation

	/** Calculates the matching of cards into groups and runs that
	  * results in the lowest amount of points for unmatched cards.
	  * @pre isComplete() */
	public void autoMatch() // some implementation

}
```

a) Draw a UML state diagram that models the important abstract states of an object or class `Hand`. Consider only state-changing methods in the list above as transition (the actual class has a few more, don't worry about them).

b) Describe (in English or pseudo-code) a strategy for implementing `automatch()` so that it produces the best possible matching. Your description does not need to get into the details of the code, but it should be clear enough to allow a competent Java programmer to implement it. Be sure to name any relevant data structure used.

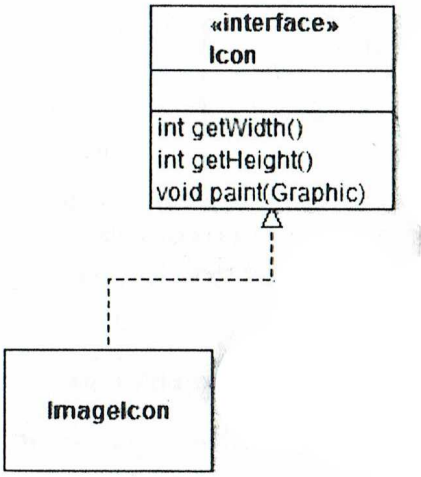
c) Consider that we call `automatch()` on a hand object representing the hand containing the cards below. Complete the object diagram to show the state of the `Hand` object after the call to `automatch()`. Make sure to include all relevant elements on your diagram. For convenience this question uses the name of the object as a shorthand for its value. In your diagram, use the standard notation seen in class for representing object values. Assume `automatch()` produces optimal matches.



Question 3 [20 marks]

Design a `CompositeIcon` class that can contain multiple icons. Note that a standard application of the COMPOSITE design pattern will result in the composed icons being painted on top of each other. Solve this problem with a `ShiftedIcon` decorator that will support drawing an icon as shifted by (parametric) x and y values.

a) Complete this UML class diagram to show your solution. Make sure you list all the methods (including constructors) that will be necessary to make this work.

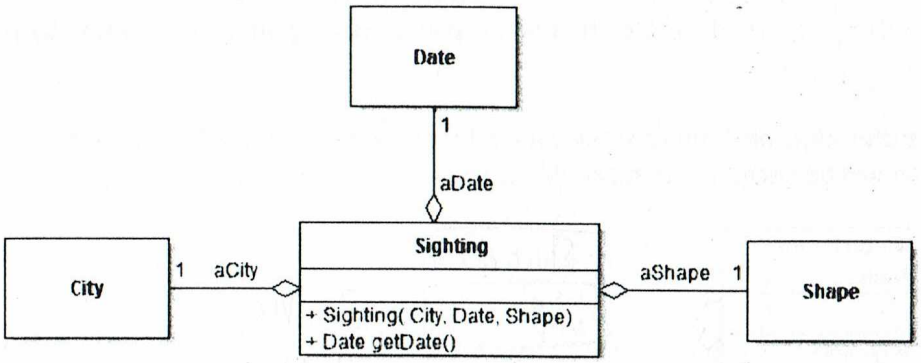


b) Write the code that will create an icon from two non-overlapping `ImageIcon`s, and assign it to a variable named `myIcon`. The code must reflect that the design required for a) is implemented.

b) Draw a UML sequence diagram showing what happens after a call to `paint` on the variable `myIcon` initialized as in the answer to b).

Question 4 [25 marks]

The following diagram shows part of the implementation of a software system used to keep track of Unidentified Flying Object (UFO) sightings in Canada.



a) The `Shape` model element represents the UFO shape: saucer, rocket, drone, etc. This element could be implemented as an enumerated type, or as an interface with concrete implementing classes. What are the respective main advantages of each approach? Fill both sides of the table using clear language and specific terminology.

Major advantage of using enums	Major advantage of using polymorphic types

b) Assume class `Date` is mutable. How will you ensure proper encapsulation of `Date` objects in class `Sighting`? Write the code that solves this issue (you don't need to implement anything that is not relevant to this question). Note the field names on the UML diagram: they map to the implementation of class `Sighting`.

```
public class Sighting
{

}
```

c) We wish to ensure that there is only ever one object of class `City` that represents a specific city. For example, there should only ever be one object to represent Montreal, even if multiple UFO sightings are recorded in Montreal. Show the implementation of class `City` that realizes this requirement. Use an appropriate design pattern.

```
public class City
{
```

d) Given the requirement in c), should instances of class `City` be immutable? Why or why not?

e) Given the requirement in c), do we need to override `Object.equals()` in class City? Why or why not?

Question 5 [15 marks]

Here is the partial implementation of a class Day representing some day in the history of the universe:

```
public class Day implements Comparable<Day>
{
	private int aYear;  private int aMonth; private int aDate;

	/** Constructs a day with a given year, month, and day…
	 * of the Julian/Gregorian calendar. */
	public Day(int pYear, int pMonth, int pDate)
	{ /* simple field assignments */ }

	public int compareTo(Day other)
	{
		if( aYear > other.aYear ) return 1;
		if( aYear < other.aYear ) return -1;
		if( aMonth > other.aMonth ) return 1;
		if( aMonth < other.aMonth ) return -1;
		return aDate - other.aDate;
	}
	// more code irrelevant to this question
}
```

a) Complete this test class to implement one JUnit test method that will execute at least each statement of `Day.compare` once. Your test should comprise all three components of a unit test. In your test, only use `Day` objects that correspond to the following dates: 1-Jan-2014, 1-Jan-2013, 1-Feb-2014, 2-Jan-2014.

```
public class TestDay {









}
```

b) Complete the code of a method that takes a string denoting the fully-qualified name of a class as argument, and returns the number of JUnit *test methods* in this class. If you don't remember the exact names of the required API assume the obvious. Feel free to add in-line comments for clarity.

```
/** returns the number of JUnit test methods in pClassName */
public int numberOfTestMethods(String pClassName)
{



}
```

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."