**McGill University**
**COMP 303 – Software Development**
**Midterm Examination 2 – 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 | /35 |
| 2 | /25 |
| 3 | /20 |
| 4 | /20 |
| **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 [35 marks]**

Produce a partial design and implementation for a bike-sharing system called *Bixee*. With Bixee, `Users` can borrow and return `Bikes` (one at the time maximum) from different `Stations` located in various places in Montreal. `User`, `Bike`, and `Station` objects all have a unique identification number (integer), and are *unique* and *immutable*.

An instance of type `Fleet` collects and manages information with the following operations:

- `borrowBike(User, Bike)`: A user borrows the bike. Information that the bike is out needs to be stored somehow. There's no need to specify from which station the bike is borrowed because the `Fleet` object should have this information.
- `returnBike(User, Bike, Station)`: A user returns the bike to the specified station.
- `removeBike(Bike)`: A bike is removed from its station by the Bixee company, for example to move it somewhere else or to repair it.
- `restoreBike(Bike, Station)`: A bike that was previously removed is restored to the fleet by putting it back to the specified station.

For convenience, you can define additional private helper methods. A good example would be `getStation(Bike)` that returns the station where a bike is located.

Whenever one of the four operations (borrow, return, remove, or restore) is performed, a number of objects should be notified, including:

- A `UserHistory` object that incrementally builds a record of all trips *for a given user*. For simplicity you can store trip information as a list of string messages describing the relevant information (for example "borrowed bike 26 from station 1"). There can be as many instances of `UserHistory` are there are users.
- A `BikeAvailabilityViewer` object that lists the bikes currently available, with their station.

Your solution should realize the following requirements:

1. There should only ever be a single instance of class `Fleet`, and this instance should be globally available.
2. It should use the Observer design pattern with a strict *push* data-flow technique.
3. Clients of the `Fleet` object should *not be responsible* for triggering notifications to observers.
4. The design needs to cleanly handle the fact that not all observers will be interested in all events (for example, `UserHistory` objects do not need to know about bikes being removed and restored).

Assume the system is initialized with all the bikes, users, and stations required, and that users or stations are never added or removed from the system. Do not concern yourself with billing or other unmentioned features. Do not concern yourself with input validation: you can assume that all requested operations will be valid (for example, that users will only try to borrow a bike if they do not have one out already).

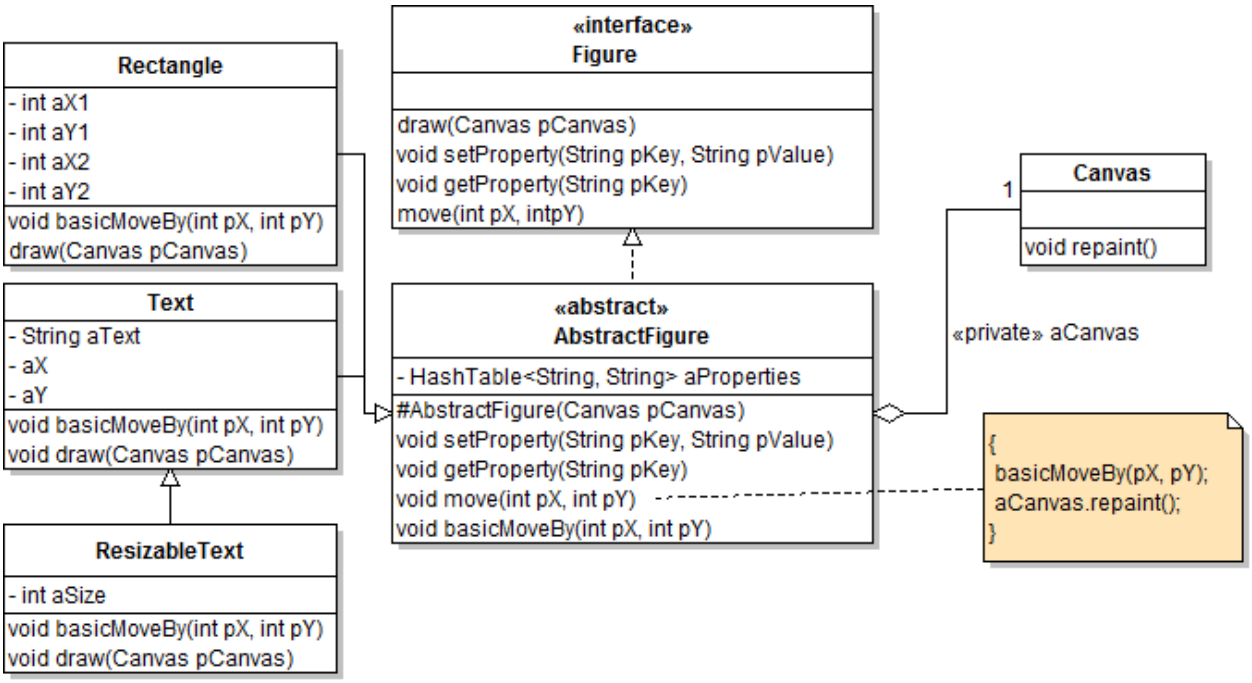
a) Draw a UML class diagram that shows all required solution elements in sufficient details. Make sure your diagram clarifies all four requirements above. Make sure to include method parameters and access modifiers. At the bottom of the page, write out, in Java source code, the definition of fields in the `Fleet` class that show how the aggregation of `User`, `Bike`, and `Station` objects is realized in practice. Use appropriate data structures.

b) Write the code of the `borrowBike` method. The code needs to be consistent with your answer to a). If your code calls helper methods, add that code as well. If you designed your aggregation structures properly, you should not need much more than 12 lines of code or so.

c) Write the code of the `UserHistory` class. The code needs to be consistent with your diagram.

**Question 2 [25 marks]**

Answer the questions based on the following design for a graph-drawing program. Method `move` is a Template Method that ensures that operations to move any kind of figure always result in a call to `aCanvas.repaint()`. Clients of `Figure` objects should only move figures by calling `move(x,y)`. The diagram is purposefully omitting some information. Assume the classes would be typically used by code in packages different from the one the classes are defined in.




a) For each method below, circle the most appropriate qualifier(s) given the above design. Not selecting any modifier is also an option.

| | |
|---|---|
| Figure.move(int pX, int pY) | public protected private static final abstract |
| AbstractFigure.move(int pX, int pY) | public protected private static final abstract |
| AbstractFigure.basicMoveBy(int pX, int pY) | public protected private static final abstract |
| Rectangle.draw(Canvas pCanvas) | public protected private static final abstract |
| Rectangle.basicMoveBy(int pX, int pY) | public protected private static final abstract |
| Canvas.repaint() | public protected private static final abstract |

b) Clients will need to be able to instantiate different kinds of figures. Write the code for the constructor of classes `Text` and the constructor for class `ResizableText`. Include all appropriate modifiers and parameters.

c) When the first line of method `AbstractFigure.move` is executed, a version of `basicMoveBy` is selected. What mechanism(s) is/are involved in the selection of `basicMoveBy`? Circle all that apply:

[overriding] [overloading] [dynamic dispatch] [static dispatch] [meta-dispatch]

[runtime error] [compiler error] [exception handling] [reflection] [instantiation]

d) In this system, assume the behavior for moving a resizable text is the same as the one required for moving a general Text (just move the position). In a first version of the system, the following code for `ResizableText.basicMoveBy` is supplied:

```
<assume the correct modifier(s) is/are used> void basicMoveBy(int pX, int pY)
{
	super.basicMoveBy(pX, pY);
}
```

Is this code? (circle the answer and complete the question below as requested):

d.1 Incorrect? How do you fix the bug?

d.2 Correct but not efficient? How do you improve it?

d.3 Correct and efficient? Briefly justify.

e) One of the properties supported is “color”. For example, `aFigure.setProperty(“color”,”red”)` would set the color of a figure to red. However, the color property is not supported for `Text` figures. To deal with this limitation, a developer redefines `setProperty` in `Text` and supplies the following implementation:

```
public void setProperty(String pKey, String pValue)
{
   if( pKey.equals("color"))
   {
      // IllegalArgumentException is an unchecked exception type
      throw new IllegalArgumentException("Color not supported for Text figures");
   }
   super.setProperty(pKey, pValue);
}
```

Is this a valid design decision? *Briefly* justify your answer with specific arguments and terminology.

f) Assume that implementations of method `basicMoveBy` can throw unchecked instances of `ArithmeticException` or `NullPointerExceptions`, and if this occurs we want to propagate the exception instance up the call chain. How can you ensure that the canvas always gets repainted in the template method, even in the presence of exceptions? Re-write the code of `AbstractFigure.move` to realize this requirement:

g) Assume the compiler’s null analysis is enabled and `Figure.getProperty` is annotated as follows:

```
@Nullable String getProperty( String pKey); // @Nullable means the value can be null
```

We have the following client code. What will be the result of the null analysis? Indicate the type of warning message(s) and corresponding line(s), or “No warning” if everything is fine.

```
01 private static final void someClientOperation(@NonNull Figure pFigure)
02 {
03     if( !pFigure.getProperty("border").equals("none") )
04     { pFigure.setProperty("default-border", "line"); }
05 }
```

**Question 3 [20 marks]**

This question requires you to describe the solution that you have completed as required for Milestone 2 of the project. Your grade will be based on *1)* how clearly you describe your existing solution, and *2)* that your description is evidence of actual working software.

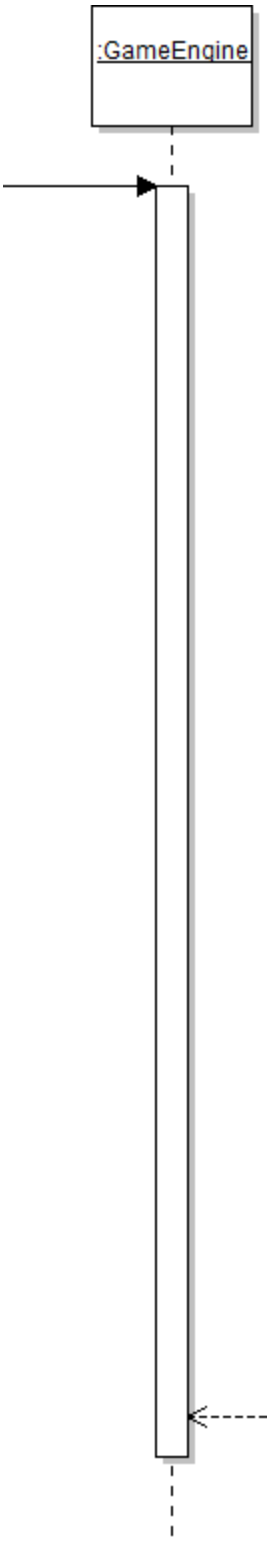
Assume that a game has progressed as follows:

```
00 New Game Initialized
01 Deck shuffled and cards distributed
02 Human:    [4H, 5D, 6S, 7H, 8S, TS, JH, JS, QD, KC]
03 Computer: [AH, 2C, 2H, 4C, 5H, 6C, 9C, 9S, TC, TD]
04 Discard Top: TWO of SPADES
05 Human starts
06 Human selects from the discard pile
07 Human discards: KING of CLUBS
08 Human's hand: [2S, 4H, 5D, 6S, 7H, 8S, TS, JH, JS, QD] (72 points)
09 Computer draws from deck: EIGHT of HEARTS
10 Computer discards: NINE of SPADES
11 Computer's hand: [AH, 2C, 2H, 4C, 5H, 6C, 8H, 9C, TC, TD] (57 points)
12 Human draws from deck: NINE of DIAMONDS
```

a) What is the next legal move?

b) In Milestone 2, you had to “isolate the decision-making behavior a computer player object should have in an interface”. Describe this interface here by listing its method and providing short comments if necessary. Make sure to include sufficient information to show how *implementers of your interface obtain the data they need to perform their function, and what that data is*. If this information is not apparent from the method parameters, provide this information in a small diagram and/or comments.

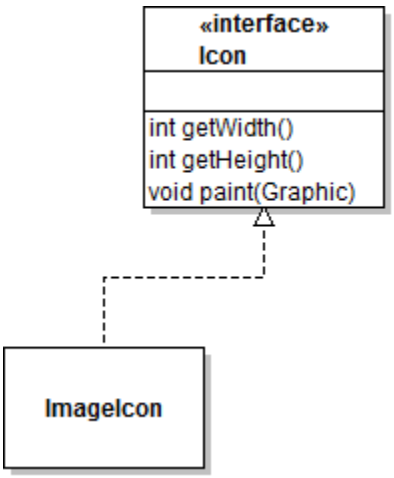
c) Create a UML sequence diagram that describes how your GameEngine works by completing the scenario that starts at step 13 in the transcript above. Write the name of the message called to the GameEngine. Include all interactions until control returns from the GameEngine. Make sure to include information that shows how each relevant game move is realized, including interactions with the AI interface described in b) and logging object. Do not overload your diagram with low-level logic details.



**Question 4 [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. *Structure you design so shifting occurs in the composite icon: clients should not be aware of the necessity to shift icons*.

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) Assume that some client code creates a composite icon from two `ImageIcons`, and assign it to a variable named `myIcon`. What constructor and/or methods *in your design* will need to be called for this? Write the client code and, below, *the code of the methods in your design that implement the required behavior*.