Type your answers to the following questions in this Word Document. When you have completed the exam (or the exam is finished), upload your document into Moodle for the assignment entitled “Third Exam” and hand in your UML diagrams.

1. (Borrowed from FRS author; 20 points) Alphatown approaches us with a new requirement. They want to monitor the pay stations on a given parking lot for two purposes: (1) they want a digital sign at the entrance stating the number of available spots for cars at the parking lot, and (2) they want to monitor the total earnings of the parking lot. Below is shown an early prototype of such a system where four pay stations are monitored by two “monitor” applications.



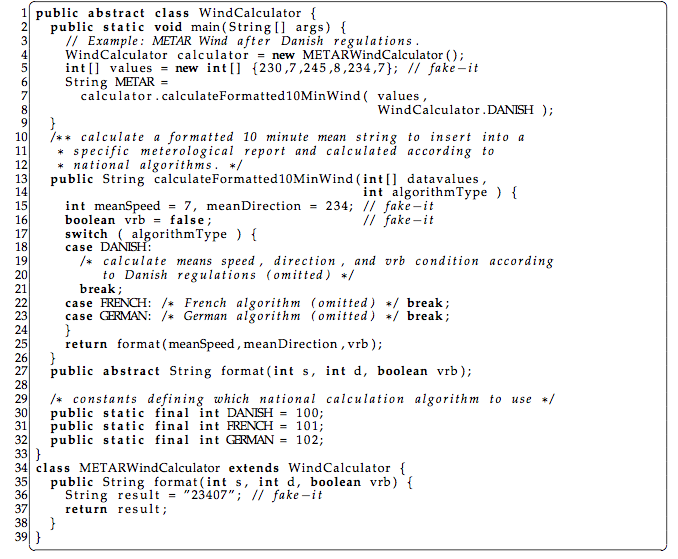
We realize that a monitor application can calculate the two properties (available spots and earnings) if it is informed of “number of minutes bought” and “amount of cents entered” in every *buy* transaction from every pay station in the parking lot (see p. 74 of the FRS textbook for the implementation of the *buy* method). We will assume that cars leave the parking lot when their time expires.

1. Describe the use of a design pattern that will solve this requirement in a flexible, reliable, and maintainable way. You should focus on the exchange of information between pay stations and monitor applications, not on the algorithm to calculate number of vacant slots.

* PayStation Status GUI invokes its main method, which then sets up the pay station status GUI and the PayStationGUI windows.

1. **On paper**, sketch a UML class diagram showing the relationship among the key interfaces and classes that are part of your design. Label the interfaces and classes with the respective roles they play in the design pattern.
2. **On paper**, sketch a UML sequence diagram showing the sequence of methods called to update the “Total Earning” display on the monitor station when a driver pushes the “Buy” button on one of the pay stations. Make sure you show how the amount inserted in the pay station is communicated to the monitor application and displayed.
3. (Borrowed from FRS author; 15 points) The pilots on in-bound and out-bound flights from an airport need precise information about the wind on the runway. One important information is the 10 minute average that describes speed, direction, and characteristics of the wind in the last 10 minutes. This information is coded in different types of “reports”, MET REPORT, METAR, and SYNOP, that each format the information in their own way. Furthermore the algorithms to calculate the 10 minute mean from observed values vary from one country to another. Reports and Countries are independent points of variability.

So far, we have sold a wind computation system to Denmark, France, and Germany, and have a design like the code shown below (Only METAR report variant is shown, and many data values are faked to reduce the code size.)



1. Use the terminology of principles for flexible design to analyze this code fragment with respect to benefits and liabilities, focusing on variability points.

* Lines 30 to 32 could have probably been changed to ENUMs. This conforms support for various regions/countries, and prevents errors from invalid values. What would happen if we called a switch on a -6 for DANISH algorithm type?
* There should be a default case on the switch statement. Could be just a default algorithm used for a continent or world depending on how big the program is.
* Depending on how in depth the algorithms are for each country, or if there are additional variables like a barometer, does the measurement for that change with the country or region of the world? Instead of a switch, there could be an algorithm object passed in. The algorithm itself could be an abstract class, with subclasses of DANISH, GERMAN, and FRENCH. Perhaps a state pattern might work if the program is limited to handling wind alone.
* If we are handling other variables besides wind, then DANISH, GERMAN, and FRENCH should be broken off into classes, with all of these classes implementing the appropriate interface, which in this case would be the algorithm. If we need a barometer, then we would need to make a barometer interface as well and consider using a factory pattern that would pass both interfaces through it.

1. Describe an alternative design that improves maintainability and flexibility.

* Instead of a switch, there could be an algorithm object passed in. The algorithm itself could be an abstract class, with subclasses of DANISH, GERMAN, and FRENCH. Perhaps a state pattern might work if the program is limited to handling wind alone.
* If we are handling other variables besides wind, then DANISH, GERMAN, and FRENCH should be broken off into classes, with all of these classes implementing the appropriate interface, which in this case would be the algorithm. If we need a barometer, then we would need to make a barometer interface as well and consider using a factory pattern that would pass both interfaces through it.

1. Sketch central refactorings of this code fragment in Java.
   1. Algorithm <interface> => FRENCH Class

GERMAN Class

DANISH Class

* 1. Public String calculateFormateFor10Wind(int[] datavalues, AlgorithmTypeFactory Algorithm)
     1. Switch (Algorithm)

Case (DANISH)

//do Danish stuff

Break;

Case (FRENCH)

//do French stuff

Break;

Case(GERMAN)

//do German stuff

Default (NULLObject)

Break;

1. For the second to last HotCiv iteration, you implemented the capability of recording a transcript of each game in the console (Ex. 36.30). Now we would like to draw an on/off switch for this capability in the top right-hand side of the HotCiv GUI, just above the shield that shows whose turn it is. We have two images for the switch, one in the *off* state and one in the *on* state:

Clicking on the switch should alternate between the two images and turn on/turn off the console transcript. (25 points)

1. Describe a design for adding this capability to your team’s HotCiv/MiniDraw project.

* Outline any refactoring needed to your team’s HotCiv/Minidraw iteration for adding this new capability.
* Identify any new classes and their responsibilities that are needed to implement your design.
* Label these classes with their respective roles from the Model-View-Controller pattern where appropriate.

1. **On paper**, draw a UML sequence diagram for your design that shows what methods are invoked when the user clicks on the *off* button to turn the console transcript off (you do not need to show methods invoked to update the GUI).
2. Write chunks of plausible Java code to show how you would implement this new capability; you can assume that the code you wrote for Exercise 36.30 is available for use in your project.