## CS-273 (section 1) Fall 2019 Exam 1 Practical (50 points)

This exam involves two exercises. In the first, you will use UML to design a simple, object oriented application. In the second, you will implement your application according to your designs in C++. You may use any of the following resources for this part of the exam: your textbook, the course slides, and your notes. Do not use the web! **This exam will be due on Monday, 9/30.**

You should make your UML in a Word document or Adobe PDF and call it either DESIGN.docx or DESIGN.pdf. Your UML should be thoroughly legible. Do not hand draw a picture and scan it in. Use a drawing tool to create the boxes and arrows according to standard UML rules.

You should submit your Exam1 Practical via Whitgit. Matt will create the repository for you by the morning of 9/30. Your project should correspond to your UML, should match the general specifications given in this exam instructions, and should be thoroughly commented. Your comments should include at least a comment header in every source file that gives your name and a short description of the relation of the file to the larger project. Your project should be organized in a modular way, with implementation separated from definition (i.e. there are separate \*.h and \*.cpp files). You should use inclusion guard.

You may call your project by any name you choose. However, please make sure it is submitted via Whitgit as described above so I can find it.

### Task 1: UML and Design

Imagine you are contracted to make a toy example that ultimately will be part of a simulator. Your toy should involve one, purely virtual base class and two derived classes, at least one of which, in turn, uses composition to include a third object not derived from the virtual base class. For example, if you were to be simulating vehicles on a road, perhaps your UML might look like so:

**Reins**

**Chariot**

**Motorcycle**

**Vehicle**

Your base class should specify that there will be at least one pure virtual method named **describe\_me()**, and the derived classes should define this method such that it displays an appropriate message to the screen for the object in question. Additionally, the object that is contained (in the example above, the reins) should have a property, e.g. color, or size, etc, and the describe\_me() of the containing object should display that property. E.g., for the example, above, Motorcycle’s describe\_me() might just display “VROOOM!” to the screen, but Chariot’s describe\_me() could display “Behold the mighty chariot, driven by horses with reins of \_\_\_\_\_\_\_\_\_!”, where the blank gets replaced by a description of the reins.

Finally, you should use an enumerated type to represent the property of the contained object, and the contained object should have a get\_property() method that “translates” that property into a string and returns it for the container class’s use. So, for example, Reins could be fleshed out like so:

**Reins**

**- Color color**

**+ string get\_color()**

**Hint**: when you get to implementation, you can use a switch() statement to return an appropriate string. In that case, Reins.h might look like:

enum Color { WHITE, BLACK};

class Reins {

private:

Color color;

public:

Reins() : color(BLACK) {}

Reins(Color c) : color(c) {}

std::string getColor();

};

And Reins.cpp:

std::string Reins::getColor()

{

std::string ret\_string;

switch (color) {

case(WHITE) :

ret\_string = "white";

break;

case(BLACK) :

ret\_string = "black";

break;

default:

ret\_string = "Err";

}

return ret\_string;

}

The general idea is that when the describe\_me() of the container class is called, it will use the get\_property method of the contained object to help it generate an appropriate description.

A couple more instructions:

You may go above and beyond the above specifications, but you may not do less.

Do not simply copy my Motorcycle and Chariot example. Make up your own.

### Task 2: Implementation

Using the loose specifications and hints given above, implement your toy in well-commented, modular C++ code, and test it using an appropriate main().