# CSCI 3081W Workshop 4 - Abstract Factory and Façade

Spring 2025

## Submission:

After you complete your code, save it as `workshop4.cc` or `workshop4.cpp` and submit it on Canvas under the Workshop 4 assignment. Workshop 4 is due Thursday, March 20th at 11:59 pm.

**If you physically attended the workshop, and if you worked with someone in this workshop, please leave a comment which contains their full name as it appears in Canvas in your Canvas submission comments section. The max group size is three people.**

**If you weren't present in today's workshop, this is an individual assignment.**

**No AI use is allowed on this assignment. Use this instead: Design Patterns**

## Goal:

The goal of this workshop is to practice implementing design patterns in C++. You will gain experience with the Abstract Factory, Facade, and (optionally) Singleton patterns, reinforcing concepts such as encapsulation, modular design, and object-oriented programming. Additionally, this workshop will help you understand how to structure complex systems, manage object creation dynamically, and apply software design principles to improve maintainability and scalability.

## Prerequisites:

Be able to SSH into a lab machine via VSCode or your terminal, and make sure your quota is below 1.5GB.
https://github.umn.edu/umn-csci-3081w-s25/FAQ
~or~
Know how to use vole, and make sure your vole quota is below 1.5GB.
(Else, contact csehelp@umn.edu )
https://cse.umn.edu/cseit/self-help-guides/virtual-online-linux-environment-vole

# Creating, compiling, and running:

**You will write all of your code in a single .cc/.cpp document (probably 100+ lines). I know that usually you would have a set of .cc and .h files, but it's really hard to grade if you do that.**

**How to Compile and Run:**

Compile: `g++ workshop4.cc`
Run: `./a.out`

Compile: `g++ workshop4.cc -o workshop4`
Run: `./workshop4`

**Frequently recompile and test your program as you develop!**

# Objective:

In this workshop, you will implement a Car Sales system using multiple design patterns, including Abstract Factory, Facade, and Singleton (Extra Credit). By completing this workshop, you will practice class inheritance, polymorphism, object creation through factories, encapsulation via a facade, and centralized state management using a singleton.

**Reminder: no AI**

# Setting up the Object Inheritance Hierarchy

1. Create an abstract class called `IVehicle`
   This class should have:
   - A pure virtual function `showDetails()` to display the vehicle's name.
   - A virtual destructor
2. Create three more abstract classes: `ICar`, `ITruck`, and `ISUV` which all inherit from `IVehicle`.
3. Create nine concrete classes: `HondaCar`, `HondaTruck`, `HondaSUV`, `ToyotaCar`, `ToyotaTruck`, `ToyotaSUV`, `SubaruCar`, `SubaruTruck`, and `SubaruSUV`.
   - Each of the classes should inherit from one of the abstract classes in step 2.
   - Each of the classes should override the `showDetails` function and print something
     - i.e. `void HondaCar::showDetails() override { std::cout << "Honda Car\n"; }`

# Implementing the Abstract Factory Pattern

1. Create an abstract factory class called `IVehicleFactory`.
   There are 3 types of vehicle (car, truck, SUV) which means we need 3 pure virtual functions:
   - `createCar()` which returns `ICar*`
   - …
2. Implement concrete factory classes for each brand:
   - `HondaFactory`, `ToyotaFactory`, `SubaruFactory`
   - Each class should realize `IVehicleFactory` and implement the creation methods to return the correct vehicle types.

# Implementing the Façade Pattern

1. Create a class called VehicleStore to serve as the façade.
2. Inside this class, create a public method called run() which will:
   a. Display menu prompting the user to choose a make.
   b. Create the corresponding factory based on user input.
   c. Prompt the user for which vehicle they would like to purchase.
   d. Continue to prompt the user until they exit (use a while loop)

# Implementing the Main Function

1. In main, create an instance of the VehicleStore.
2. Call the run() function to start the program.

```
int main() {
    VehicleStore* vs = new VehicleStore();
    vs->run();
    delete vs;
}
```

# Extra Credit: Implementing the Singleton Pattern (Vehicle Sales Log)

Think about how you might design a centralized system to keep track of all vehicle sales created during the program's execution.

This system should:

- Ensure there is only one instance managing this information.
- Keep a record of created vehicles (use a map)
- Display the record of information on program termination.
- Be integrated into the existing workflow without altering the core functionality of the factory or façade.

Submission requirements: If you attempt the extra credit, please comment "extra credit" in the Canvas submission comments.

**Reminder: no AI**