Bringing British Education to You

www.nccedu.com

Analysis, Design and Implementation

Topic 10:

Redesign and Implementation

V1.0

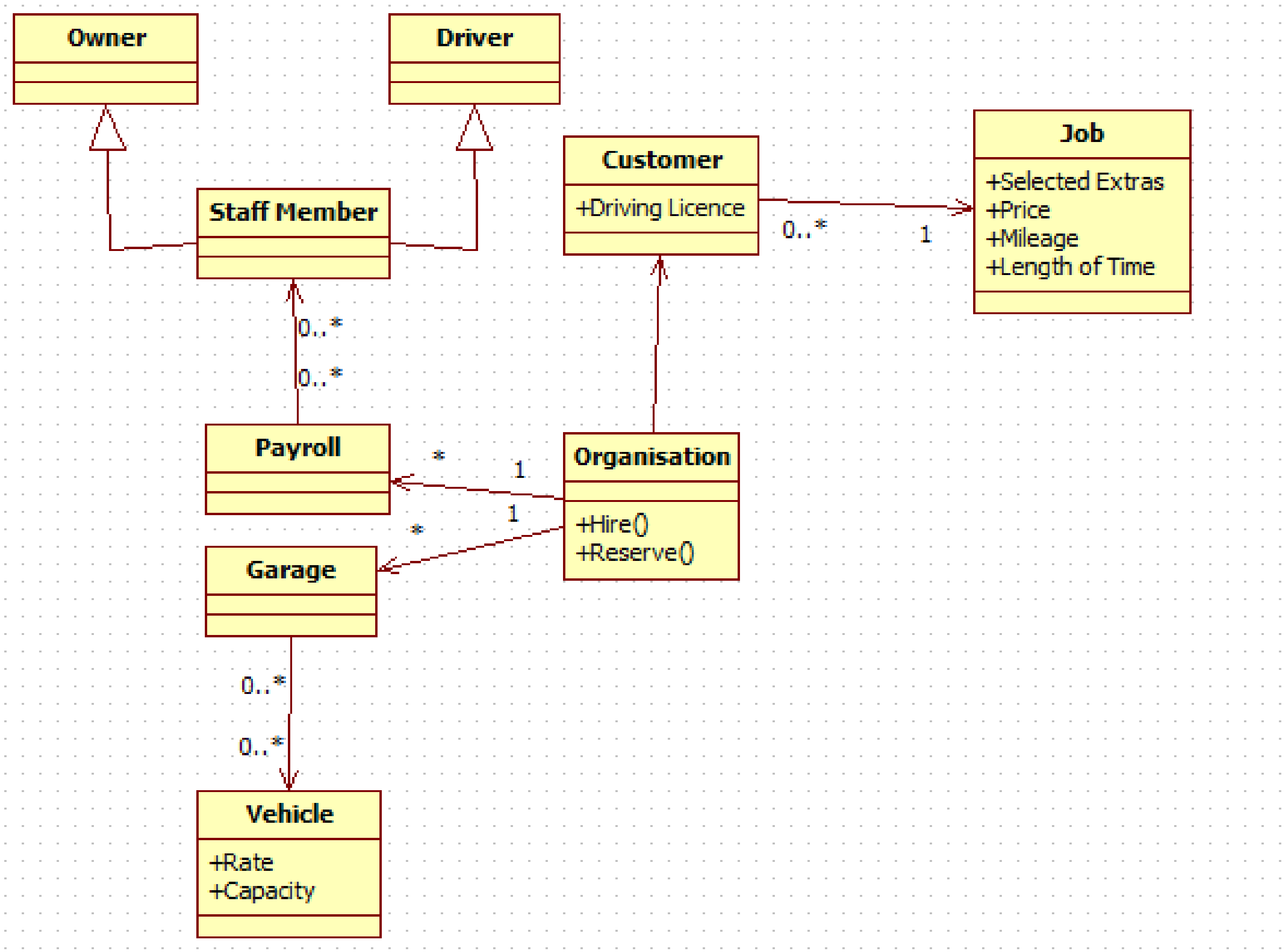
Introduction

- In Topic six, we worked through a design case study for a vehicle management service.
- In this topic, we are going to look at issues of implementation that go with the scenario.
- We have a number of new tools in our toolkit. These are our design patterns.
- We should examine each of the things our system will have to do, and identify if we need to adjust our design to accommodate.

Refactoring

- ***Refactoring*** is the process of improving things that already exist.
  - We'll talk more about this in the next topic.
- We want to refactor our design so that it is as well engineered as it possibly can be.
  - This is part of the iterative nature of analysis and design.
- This process falls a little between design and implementation.
  - We need to know about our implementation context.

Our Design so Far – Classes



Assessing the Design - 1

- Our first step is to look at where we can refactor our class diagram in light of what we now know about high quality software.
  - Assess for coupling and cohesion
  - Apply design patterns in light of requirements
- Our system is data coupled for the most part, but not heavily so.
  - That's good, but it perhaps it could be better.
  - It will require some redesign.

Assessing the Design - 2

- Although we do not have methods and attributes defined, we can be reasonably certain cohesion is high.
  - Each class has a narrowly defined responsibility.
  - The existence of classes like Payroll and Garage show that there is a proper separation between ‘representing a unit’ and ‘representing the collection of units’
- We may want to reconsider the class diagram in light of designing for software components.

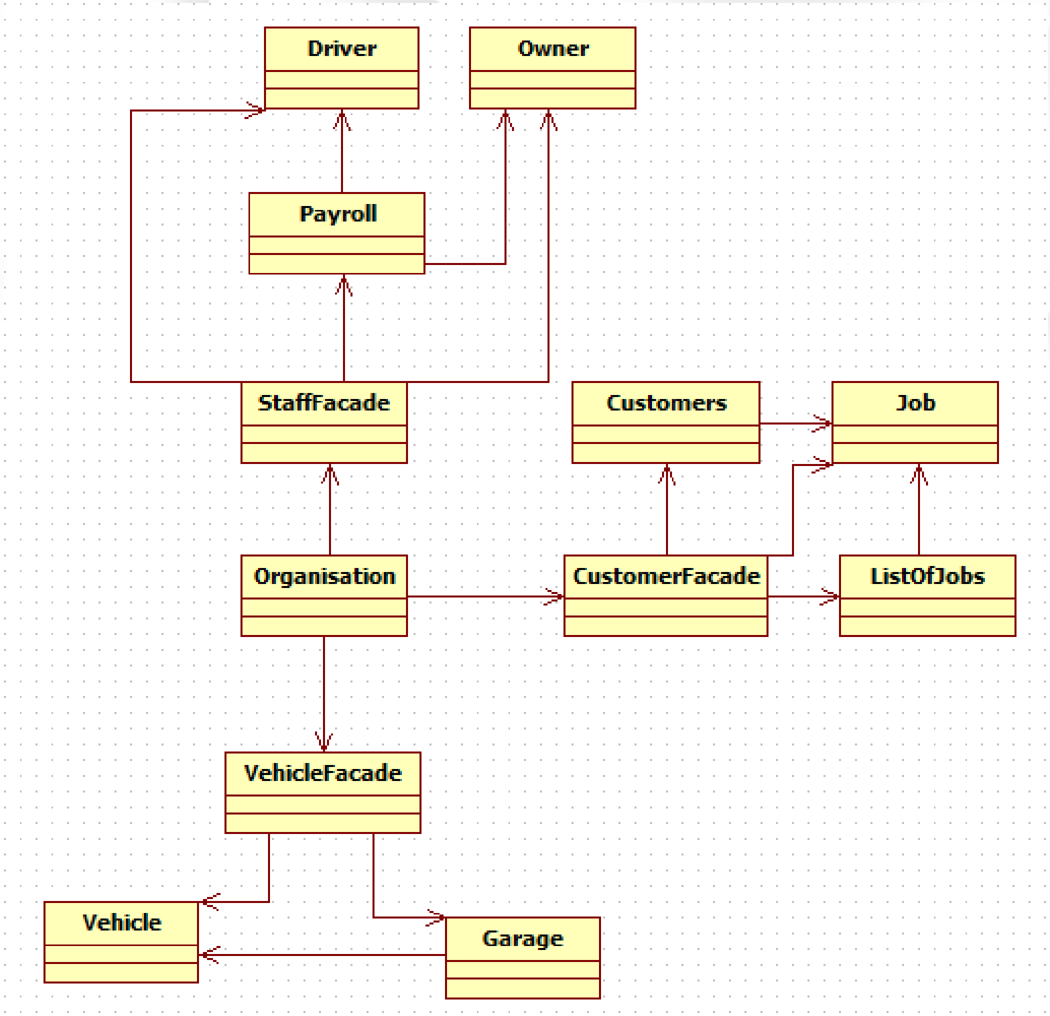
Redesigning

- Redesigning is not a scientific process.
  - There is no ***right*** answer, although there are plenty of ***wrong*** answers.
- Opinions will vary on how to approach a particular redesign.
  - Everything involves trade-offs.
- Even choosing to use a design pattern is a trade-off.
  - Extra flexibility versus an increased class count and all that is associated.

Component Design - 1

- Software component design would require us to break this system up into three parts:
  - Vehicle Management
  - Staff management
  - Customer management
- Each of these would be linked into the Organisation class.
- We could usefully use a facade here to implement our black box. Is this good design?

Component Design



- Component design here introduces three new classes, and a large amount of additional coupling. Our three new classes have low cohesion.
- Not appropriate for this project.

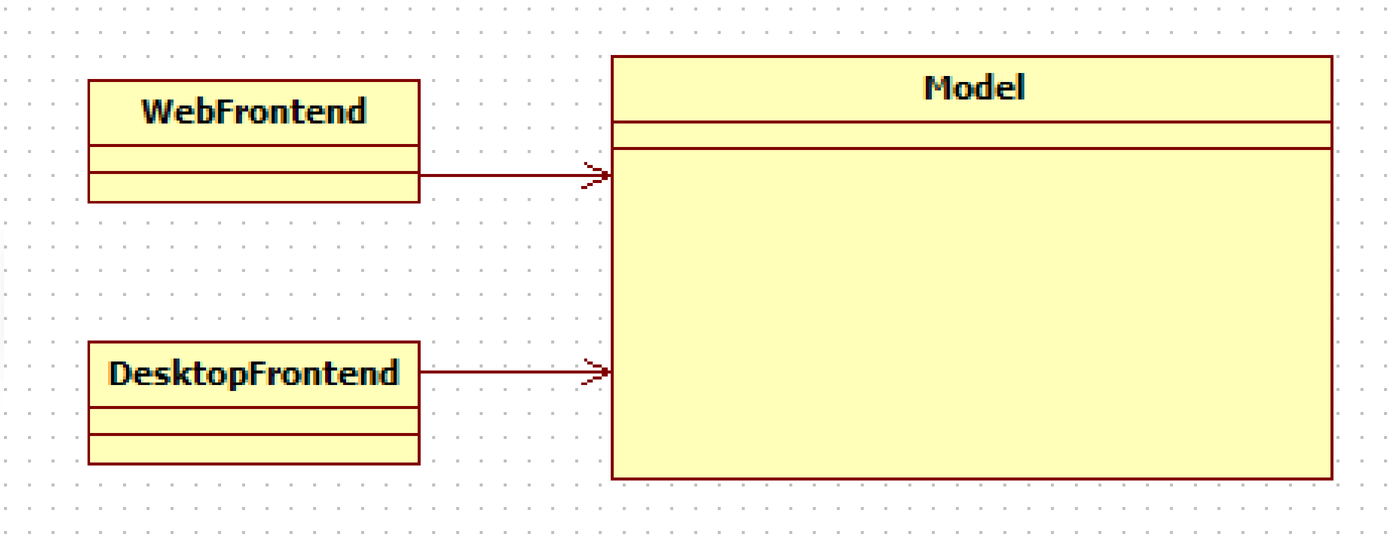
Design Patterns

- Component design introduces more problems than it solves in this example.
  - It comes into its own when discussing much larger projects.
- What about our design patterns?
- Are any appropriate here?
- Starting from our original design, we can start to look at the functionality we have identified and determine where they are appropriate.

The Model View Controller - 1

- The MVC architecture is one that we should always be looking to use.
- In our case, all we have at the moment is our model. However, we have also been told we must implement front-ends in both desktop and web form.
  - Thus, we need to expand our system a bit to include this.
- These new classes will be two separate view/controller classes.
  - They do the same thing, just in different ways.

The Model View Controller - 2

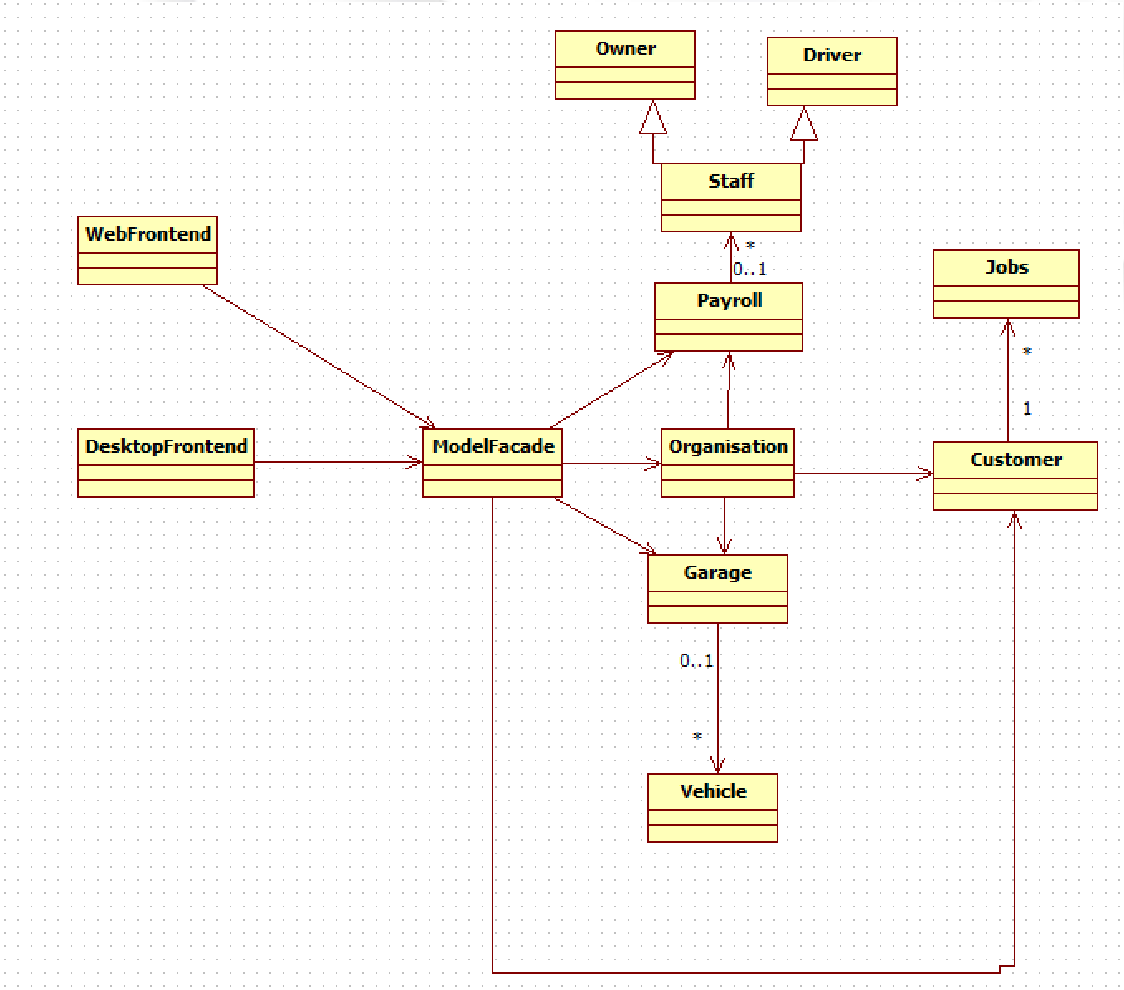


- How do the View/Controller classes interact with the model?
- We'd want to expose a facade from our model to permit this.

The Facade

- When creating a black box component, we must hide implementation details.
  - Otherwise, parts of the system become structurally dependent.
- We can do this in our model behind a facade.
  - Note that while the Organisation class ties together all of our system, it’s not a facade.
  - The roles performed by the classes are different.
    - Organisation ties things together.
    - The facade simplifies the API and creates an interface.

Modified Design

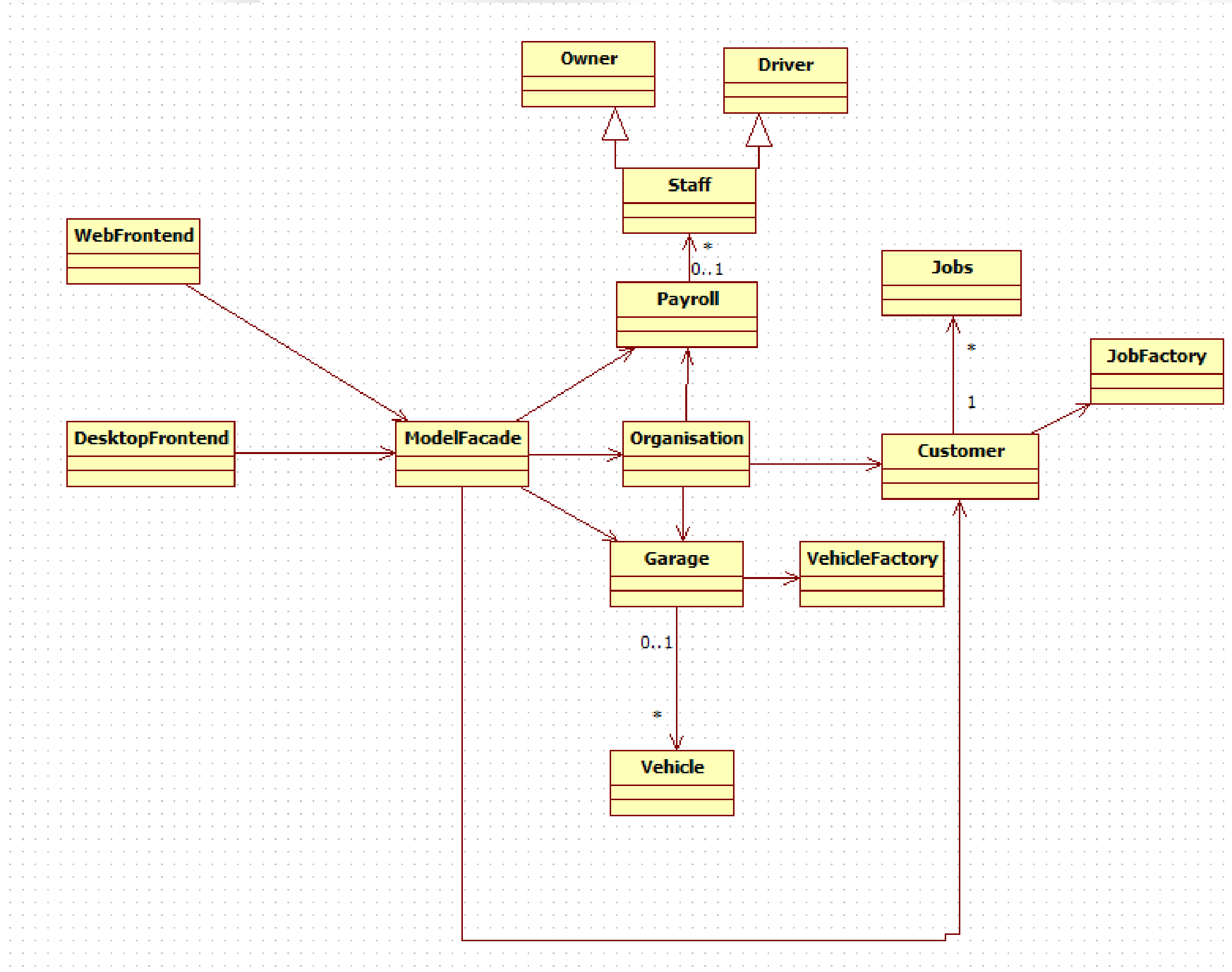


- We gain our facade as an entry point and exposed interface to our model.
- It’s a highly coupled class with low cohesion, but that is the cost we must pay.

The Factory Design Pattern - 1

- We are presumably going to be creating a number of jobs as we go along.
  - We perhaps want to create a job factory that:
    - Creates the job based on the data we are given
    - Assigns it to the customer
- Likewise for vehicles:
  - Creates vehicle objects using the details we give them.
- We should consider a factory whenever we are creating many instances of an object with complex configuration.

The Factory Design Pattern - 2



- There is no need for our factories to be complex.
- We can have them as classes with static factory methods if we desire.

The Strategy Pattern

- We don’t have a lot of need here for implementing a strategy pattern.
  - Not all patterns are useful everywhere.
- We ***could*** use a strategy pattern to create a flexible link between the logic for hiring a vehicle and hiring a driver.
  - We must consider what we would really gain from this versus the cost.
- It is perhaps not suitable in this project.

The Flyweight Pattern

- Most of the objects we create have context to go with their state.
  - There is a difference between Vehicle 1 and Vehicle 2, in that they will be assigned to different drivers and jobs.
- Flyweight objects are useful only if they are identical in all respects and free of context.
- The flyweight has no appropriate role in our project. Thus, we don’t include it.

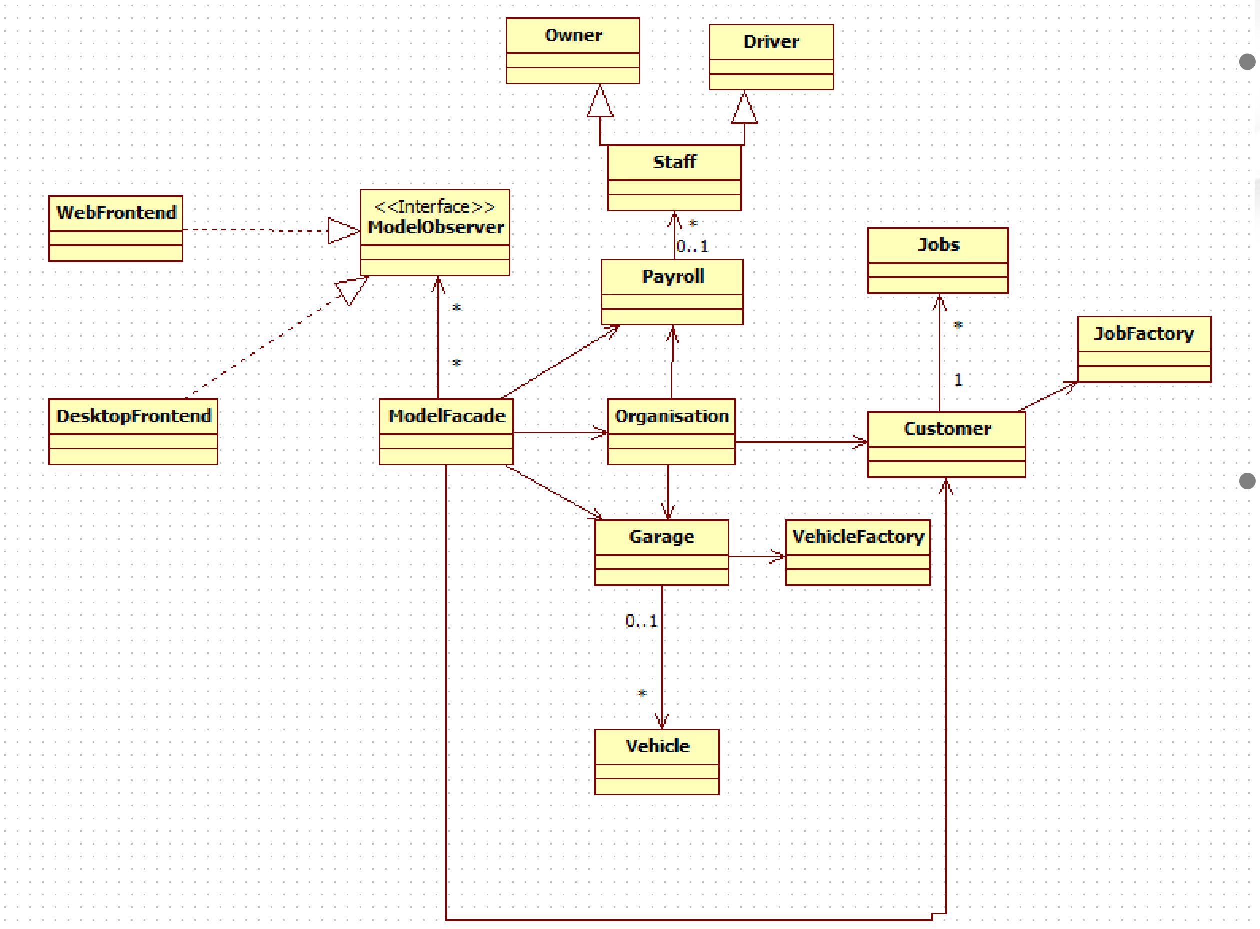
The Observer Pattern - 1

- The Observer pattern has a role in most programs.
  - It lets us implement callback coupling.

- However, the benefit gained from this is often not worth the cost of increased object and class complexity.

- We can profitably implement this as the primary mechanism for communication between our model and the facade however, and we should do this to remove structural dependencies.

The Observer Pattern - 2



- Here we add in an interface class, and have the front-end classes implement it.
- In this way, we have obtained the loosest coupling between the model and the view/controller.

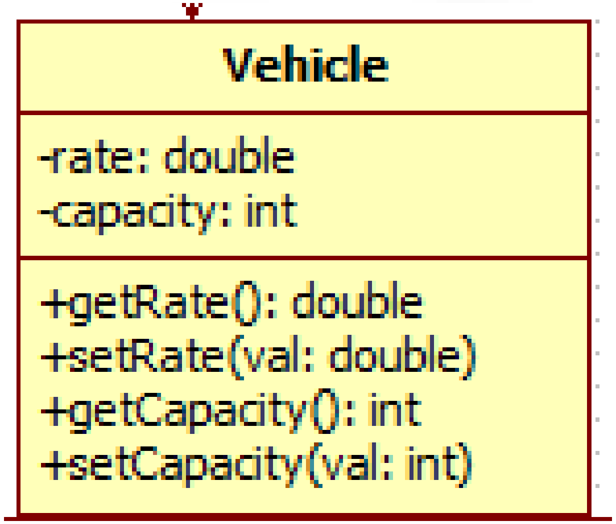
Implementation - 1

- We now have a class diagram that we can convert into code.
  - We know what role each of the classes are going to play.
  - We know where we are using design patterns to their best effect.
- Converting an activity diagram into code is the same process as turning pseudocode into code.
  - We have discussed this process already.
- Our class diagram is a little more complex.

Implementation - 2

- Our class diagram omits attributes and operations.
  - We have already discussed how these should be handled.
  - Fleshing out the diagram with these is left as an exercise for students.
- Our first step in implementing a class diagram is to sketch out the classes in code.
- We start with classes that have no dependencies, so that we can compile as we go along.

Vehicle Implementation



Vehicle

-rate: double
-capacity: int

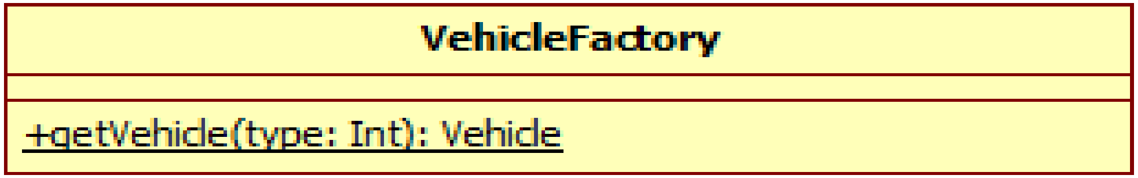
+getRate(): double
+setRate(val: double)
+getCapacity(): int
+setCapacity(val: int)

```
public class Vehicle {
    private double rate;
    private int capacity;

    public void setRate(double val) {
        rate = val;
    }

    public double getRate() {
        return rate;
    }

    public void setCapacity (int val) {
        capacity = val;
    }

    public int getCapacity() {
        return capacity;
    }
}
```

Implementation - 3

- Implementing a base class like this allows us to then implement dependent classes, such as the VehicleFactory.
- Our factory is going to take in the parts of the vehicle that must be configured, and then spit out a configured object.
  - This will be done as a static method so as to avoid the need to instantiate an object.

Implementation of Factory – 1



```
public class VehicleFactory {
    private static final int TYPE_TRANSIT = 0;
    private static final int TYPE_COMBO = 1;
    private static final int TYPE_BOX = 2;

    public static Vehicle getVehicle (int type) {
        return null;
    }
}
```

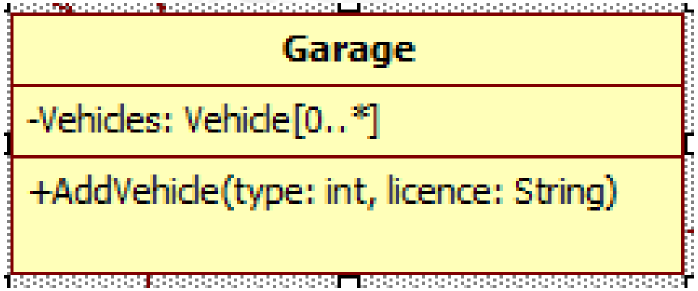
Implementation of Factory – 2

```
public static Vehicle getVehicle (int type) {
       double rate = 0.0;
       int capacity = 0;
       Vehicle v = new Vehicle();

       switch (type) {
           case TYPE_TRANSIT:
               rate = 2.0;
               capacity = 2000;
           break;
           case TYPE_COMBO:
               rate = 1.5;
               capacity = 1000;
           break;
           case TYPE_BOX:
               rate = 3.0;
               capacity = 5000;
           break;
           default:
               return null;
       }
       v.setRate (rate);
       v.setCapacity (capacity);
       return v;
   }
```

Implementation - 4

- Then, we can implement the class that requires the existence of our factory – the Garage
- We need to decide on how the Garage is going to store vehicles.
    - We’ll use a HashMap for the this.
- Our HashMap will store vehicle objects by licence plate.
    - This give us an easy way to query specific vehicles.

Garage Implementation



Garage

-Vehicles: Vehicle[0..*]

+AddVehicle(type: int, licence: String)

```
public class Garage {

    HashMap<String,Vehicle> myVehicles;

    public Garage() {
        myVehicles = new
HashMap<String,Vehicle>();
    }

    public void addVehicle (String licence, int
type) {
        Vehicle v = VehicleFactory.getVehicle
(type);

        myVehicles.put (licence, v);
    }

}
```

Implementation - 5

- Implementation progresses like so:
  - Create base classes
    - Implement their logic.
  - Create dependent classes
    - Link them to the base classes.
- You do not need to implement all functionality at once. You can approach the development incrementally.
  - We still need to implement functionality for removing vehicles, for example.

Conclusion

- Analysis and Design is an iterative process.
  - We need to revisit our designs as we learn more about how to implement things.
  - We need to revisit our analysis as we reveal problems with our design.
- Design patterns can be useful, but not in all situations.
- We must always be mindful of the cost of the benefits they give us.

Topic 10 – Redesign and Implementation

Any Questions?

Bringing British
Education to You
www.nccedu.com