Design Patterns

The Façade Design Pattern

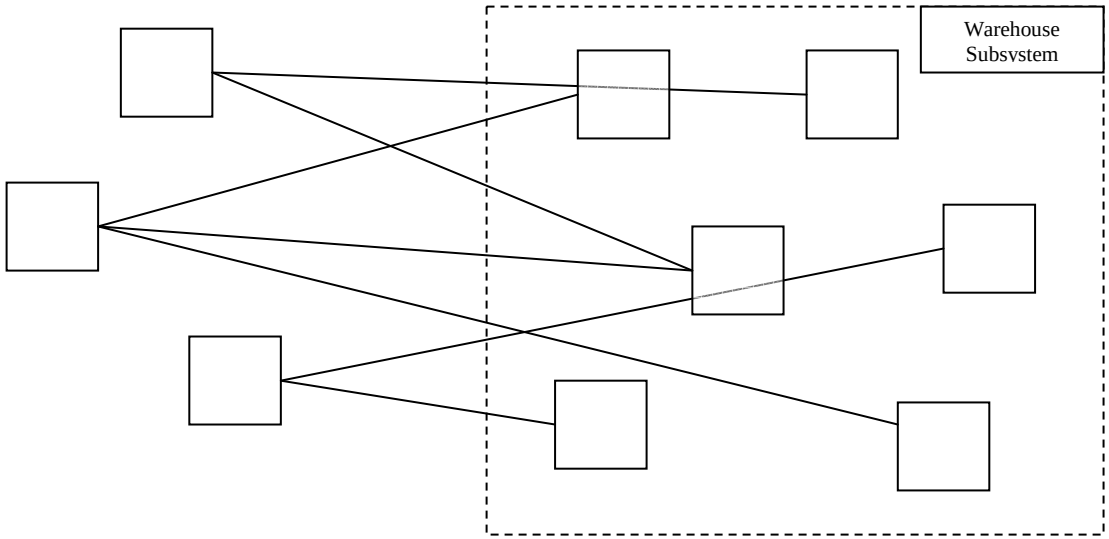
Façade
Design Pattern

The Issue:

Though a common situation, it is generally considered a bad practice to integrate the details of one class into another. This violates the basic principles of abstraction and information hiding.

If some class is coded to the publicly exposed methods of some other class (i.e., it's interface) – then these classes become tightly coupled to each other.

For example, if classes in some application (referred to as "client" classes) are coded directly to the details of a "Warehouse" class in some warehouse subsystem, then those classes become tightly coupled to the "Warehouse" class.



The Issue (cont):

As a result, changes to the interface or structure of the “Warehouse” class could force the client code to change.

Additionally, if classes were added to the Warehouse subsystem (i.e., Depot, Factory, etc.), then since the client code is coded specifically to Warehouse objects, the code in those client classes would require changes to make use of these additional classes.

Furthermore, directly accessing individual domain objects (and invoking methods of their interface) can potentially result in incorrect behavior or create an inconsistent application state

For example, assume that ordered items should be selected from the Warehouse with the most items in stock. References to the individual Warehouse objects would allow client code to invoke methods that would result in selecting items from a Warehouse regardless of the number of items in stock.

As you can see, this situation results in significant problems:

- Tightly coupled classes
- Violations of abstraction, encapsulation & information hiding

Façade

Design Pattern

The Solution:

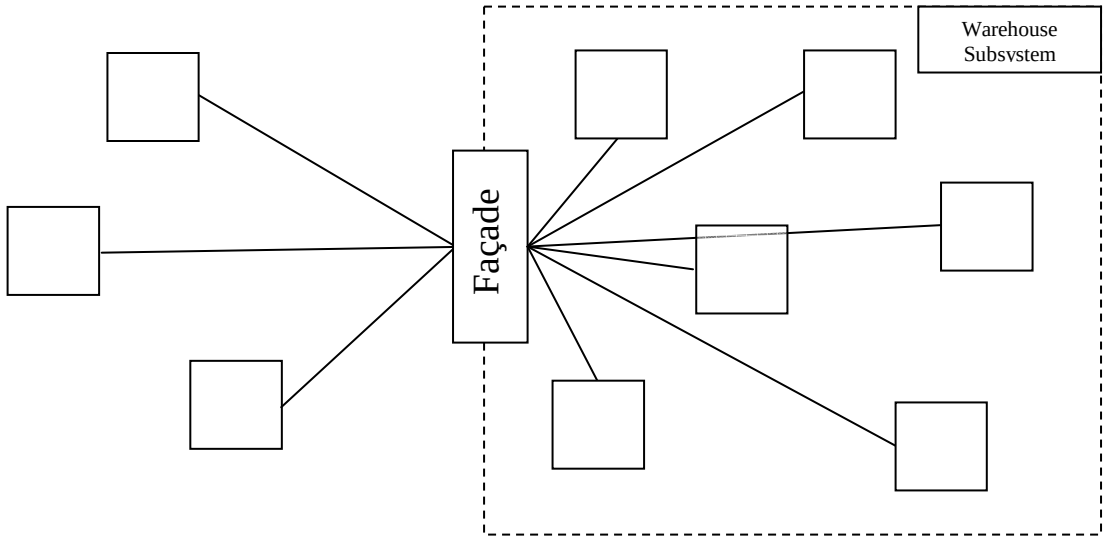
The Façade Pattern

Using the Façade design pattern we can avoid these problems.

The Façade design pattern specifies the creation of a class with a unified interface to a set of interfaces (publicly exposed methods) in a subsystem. The Façade pattern defines a higher-level interface that makes the subsystem easier to use.

By using the Façade pattern, we provide a simple interface to a complex subsystem. This interface is good enough for most clients; more sophisticated clients can look beyond the facade.

By using the Façade pattern, we also decouple the classes of a subsystem from client classes and other subsystems, thereby promoting subsystem independence and portability.



Façade
Design Pattern

The Façade Pattern:

The Principle of Least Knowledge and the Law of Demeter

These concepts are design guidelines for developing software, particularly object-oriented programs. The guideline was invented at Northeastern University, Boston in the fall of 1987, and can be succinctly summarized as:
"Only talk to your immediate friends"

The fundamental notion is that a given object should assume as little as possible about the structure or properties of anything else (including its subcomponents).

The Façade pattern is an example of the Principle of Least Knowledge and the Law of Demeter at work.

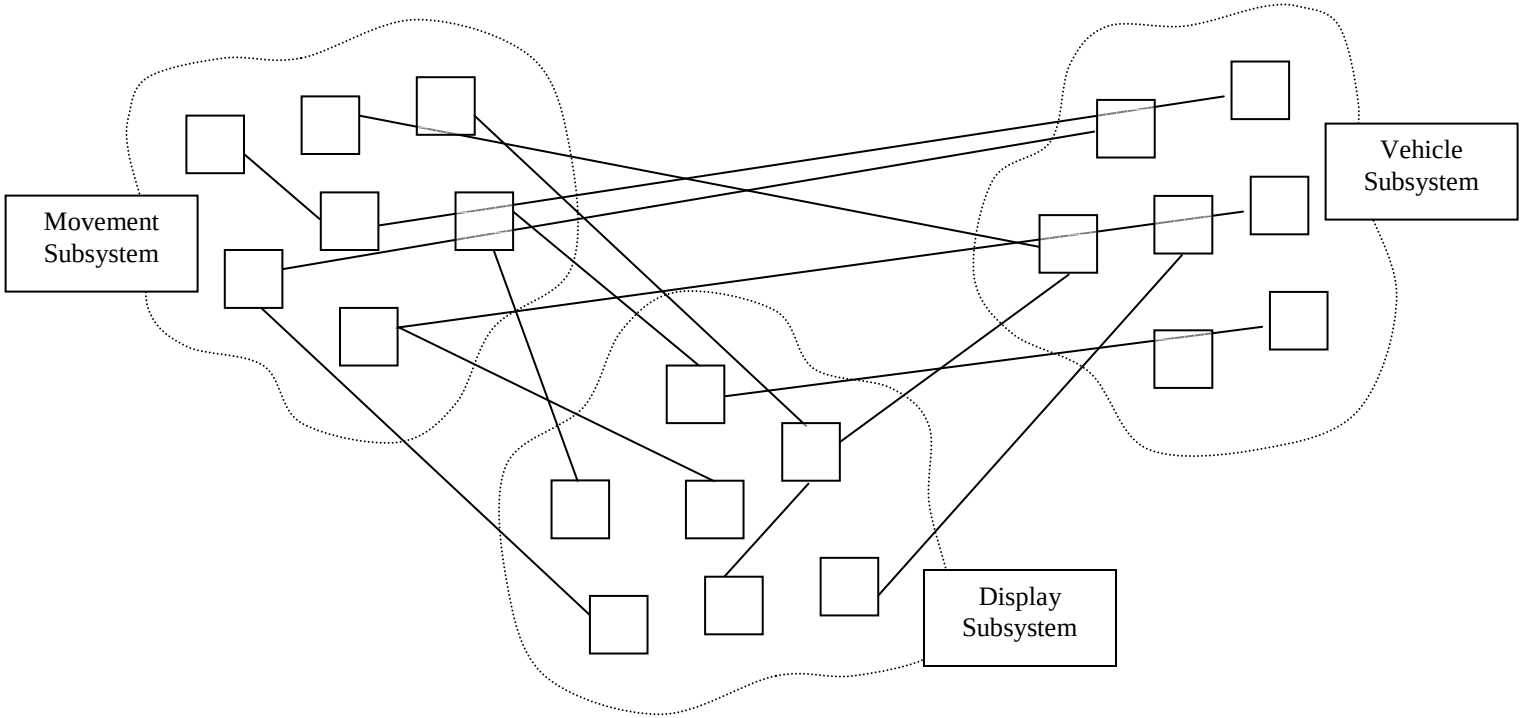
In the Façade pattern, several classes are hidden behind one API (i.e., interface).

For example, if you are working with a set of classes that represent vehicles, a common design practice is to control all access to these objects through a “Façade” (a class). This Façade will prevent clients from obtaining a reference to the objects that represent the vehicles themselves. This is the Session Façade pattern.

Façade
Design Pattern

The Façade Pattern:

Example: Vehicle Application – Without the Façade Pattern

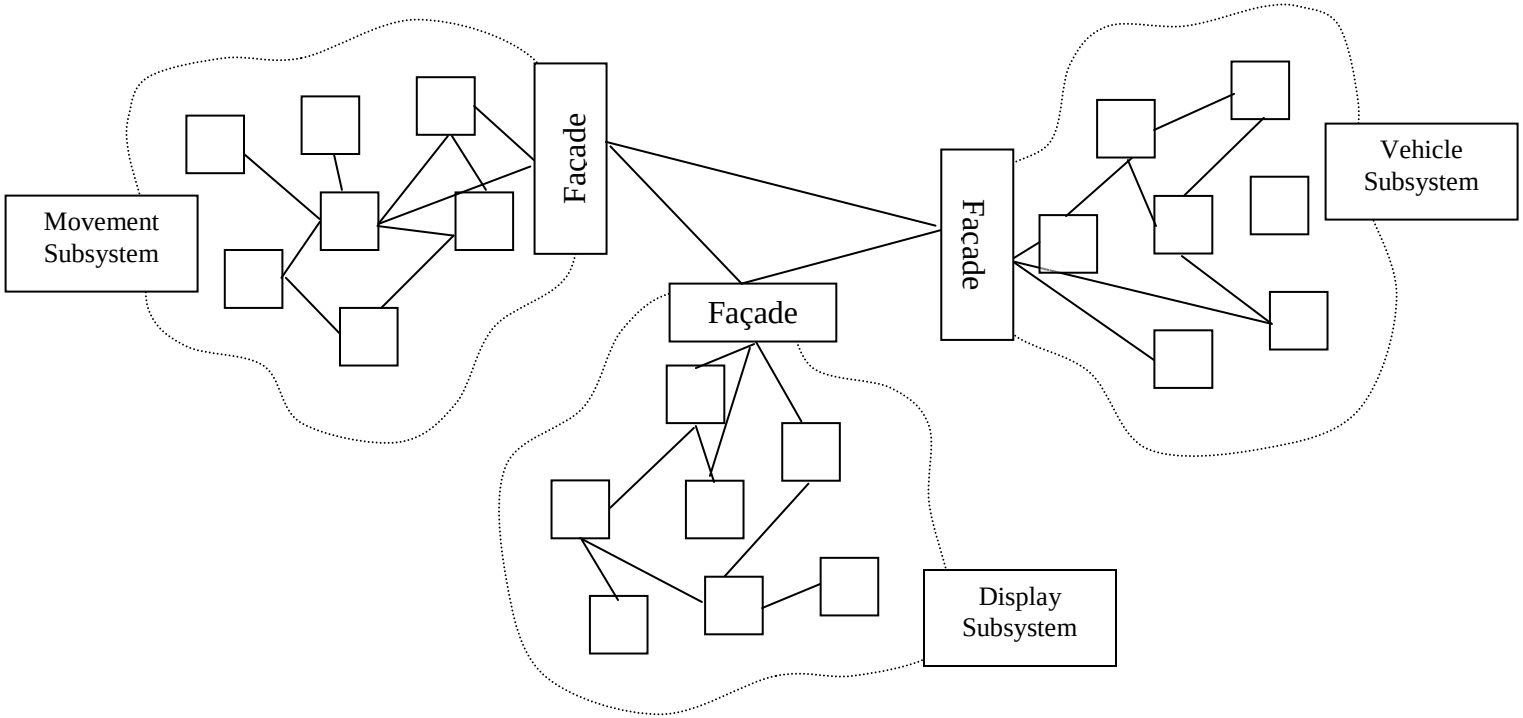


These components and their associated objects are tightly coupled. Each subsystem and their component objects know about the contents of the other subsystems. Attempting to reuse one of these components would be nearly impossible without significantly modifying the existing code. Note that contrary to the Principle of Least Knowledge, the classes in the subsystems above are *not* just communicating with their "immediate friends".

Façade
Design Pattern

The Façade Pattern:

Example: Vehicle Application – With the Façade Pattern



These components are loosely coupled. Each subsystem exists independent of the other subsystems. These subsystems can be easily reused without modifying any existing code. Note as Principle of Least Knowledge dictates, the classes in the subsystems above *are* just communicating with their "immediate friends".

Façade
Design Pattern

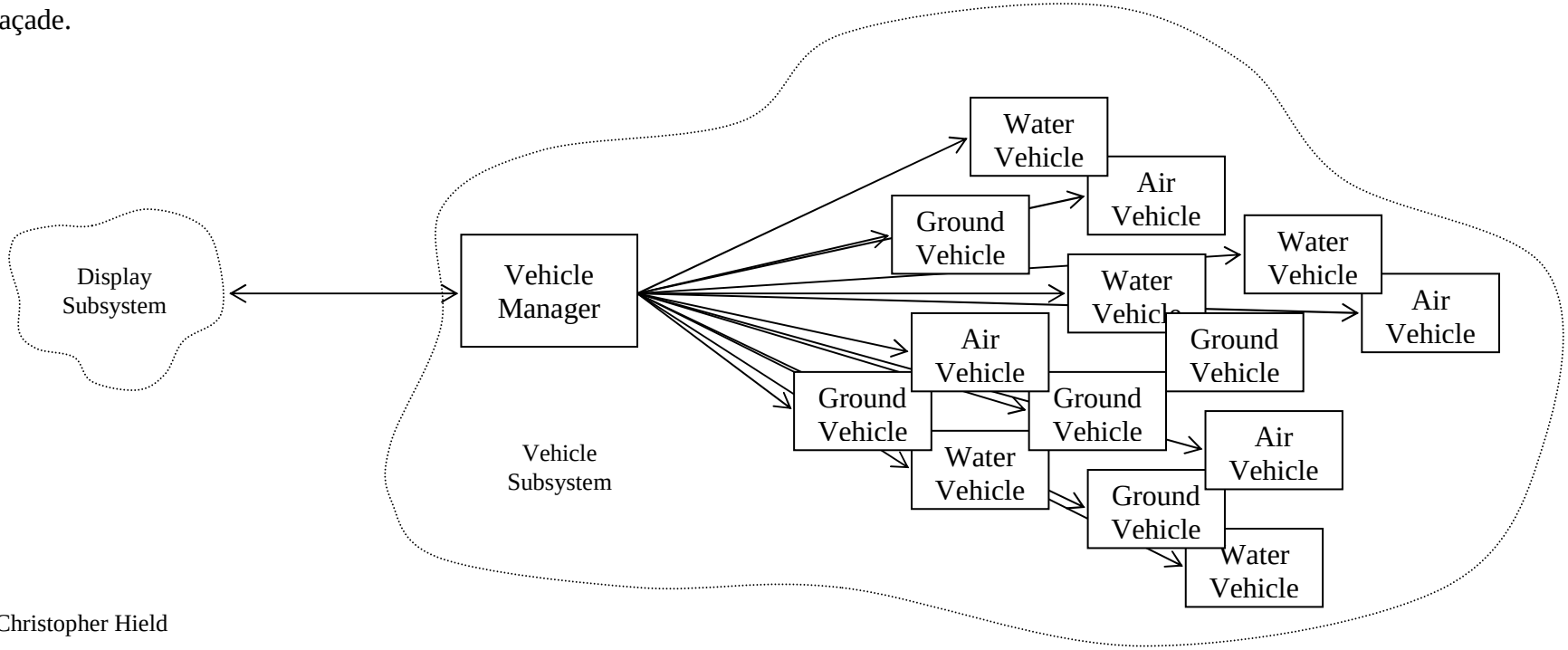
The Façade Pattern:

Example: Vehicle Application – With the Façade Pattern

The Display subsystem below is loosely coupled to the Vehicle subsystem. Changes within the Vehicle subsystem do not impact the Display subsystem. Classes can be added, modified, or removed within the Vehicle subsystem.

Additionally, the objects within the Vehicle subsystem are separated and protected from access by client code. Client code cannot obtain a reference to the domain objects behind the Vehicle Manager (behind the Façade).

Requests for information or modification go from client code through the Vehicle Manager (the Façade) to the objects behind the Façade.



Façade
Design Pattern

The Façade Pattern:

Singleton Façades

Classes implementing the Façade pattern are often designed as Singletons. Since Façades are designed to "hide" the contents of a subsystem or component, it does not always make sense to have more then one instance of a point of entry to the subsystem. By designing a Façade as a Singleton, we are insured that only one Façade object can be created.

Manage by Interface, not by Class

Recall the previous example that used a Vehicle Manager Façade class to manage Vehicle objects of all sorts. This is only possible if the Façade class refers to the objects it is associated with by *interface* type.

If a Façade class refers to the objects it is associated with by *class* type, then the Façade can only ever refer to objects of that class type (or it's one of it's subclasses). This can be limiting. If additional classes need to be referenced/managed by the Façade, code changes will likely be needed.

If the Façade class refers to the objects that it is associated with by *interface* type, then that Façade can refer to any object of any class type as long as the class (or one of it's super-classes) implements the appropriate interface.
New classes can be referenced/managed by the Façade with no code changes as long as the new class implements the appropriate interface.

Façade
Design Pattern

The Façade Pattern:

Façades Usage

Assume I want to get the speed of a certain vehicle, and the vehicle objects are hidden behind a Façade (VehicleManager).

Remember that a Façade simplifies the interface to the objects that it protects. This means that we should expect the VehicleManager class to contain a simple, useful and concise set of methods designed to work the various vehicle objects that it refers to.

Now assume that the VehicleManager Façade class has a method called “getSpeed” that takes a String vehicle identifier as it’s parameter. In this method, the VehicleManager will find the vehicle with the same identifier as the vehicle identifier passed in, and will then return it’s speed.

```
double theSpeed = VehicleManager.getInstance().getSpeed(“ABC123”);
```

Note that the above assumes that the VehicleManager Façade is implemented as a Singleton

Assume a similar method exists to *set* the speed:

```
double theSpeed = VehicleManager.getInstance().setSpeed(“ABC123”, 55.0);
```

Façade

Design Pattern

Summary:

- The Façade pattern hides the implementation of subsystems from clients, making subsystem easier to use.
- The Façade pattern promotes weak/loose coupling between the subsystem and its clients. This allows you to change the classes that comprise the subsystem without affecting the clients.
- The Façade pattern reduces compilation dependencies in large software systems.
- The Façade pattern simplifies porting systems to other platforms, because it's less likely that building one subsystem requires building all others.
- Note that Façade does not add any functionality, it simplifies interfaces
- The Façade provides developers with a common interface to the subsystem, leading to more uniform code.
- The Façade isolates your system and provides a layer of protection from complexities in your subsystems as they evolve. This protection makes it easier to replace one subsystem with another because the dependencies are isolated.
- Façades support the Principle of Least Knowledge and the Law of Demeter. These concepts can be succinctly summarized as: "Only talk to your immediate friends"