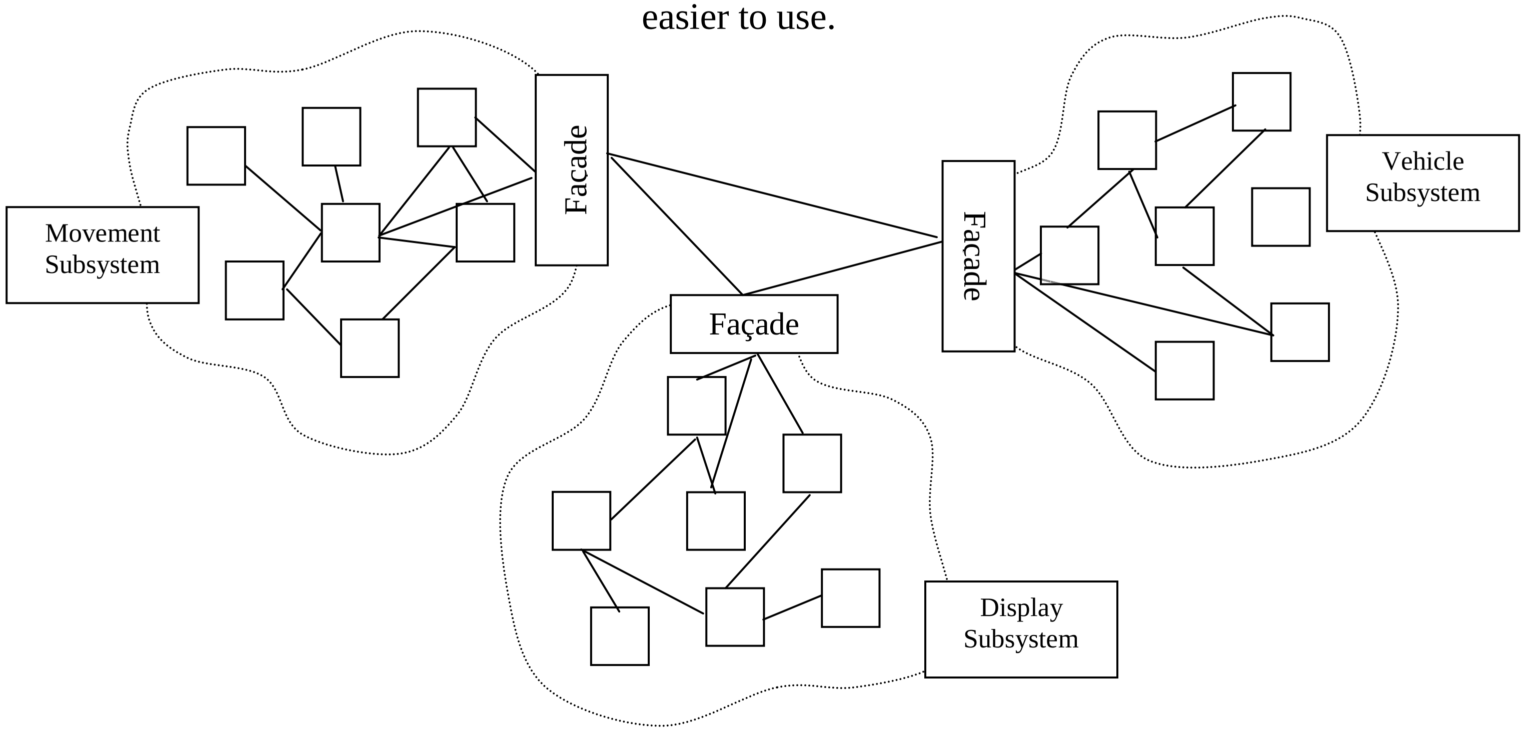
Design Patterns

The Data Transfer Object Design Pattern

The Issue:

Recall the Façade design pattern:

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.



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.

The Issue:

This technique works as the following example illustrates:

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);
```

The Issue:

However…

There is a problem that can arise when the subsystem that the Façade is designed to work with is made up of data-rich or method-rich classes (classes that contain many data attributes or many methods).

In this situation, the Façade's interface can grow large, containing a proliferation of accessors (get's), modifiers (set's), etc. needed to support the needs of the subsystem components.

In the previous example, we saw an example of a getSpeed & setSpeed method of the Vehicle subsystem Façade. If the classes behind that Façade contained many data attributes and publically exposed methods, then the Façade would likely have to support the ability to get and set those data attributes.

This can result in a Façade whose interface looks like:

```
public Point3D getDestination();
public void setDestination(Point3D aPoint);
public void setDestination(Locatable l);
public void clearDestination();
public double getSpeed();
public void setSpeed(double speedIn);
public double getMaxSpeed();
public void setMaxSpeed(double maxIn);
public boolean atDestination();
public void destroy();
public void drawItem();
public void move(double time);
```

```
public Point3D getLocation();
public void setLocation(Point3D aPoint);
public double distance(Locatable aPoint);
public double distance(Point3D aPoint);
public Locatable closest(ArrayList points);
public void setIdentifier(String id);
public String getIdentifier();
public void setDescription(String desc);
public String getDescription();
public void update(double time);
public double calculateDirection();
public void arrived();
```

This interface does not look very high-level, and it is doubtful that it makes the subsystem easier to use.

This is contrary to the goals for the Façade pattern.

Data Transfer Object
Design Pattern

The Solution:

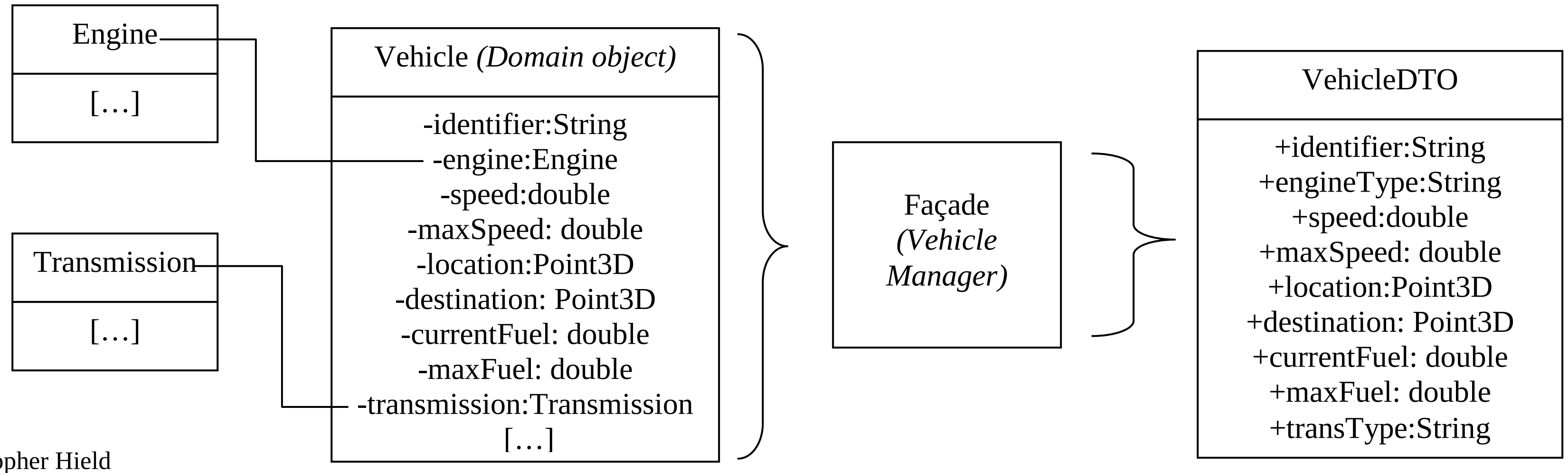
The Data Transfer Object Pattern

Using the Data Transfer Object design pattern we can avoid these problems.

Rather than force a Façade class to maintain a large interface to support domain objects composed of a large number of data attributes, in the Data Transfer Object (DTO) pattern the Façade encapsulates the key data of a domain object in a simple class (the Data Transfer Object) and returns the DTO object (not individual data elements) to clients.

DTO's are simple data objects made up of public data attributes containing copies of key data of the domain object.

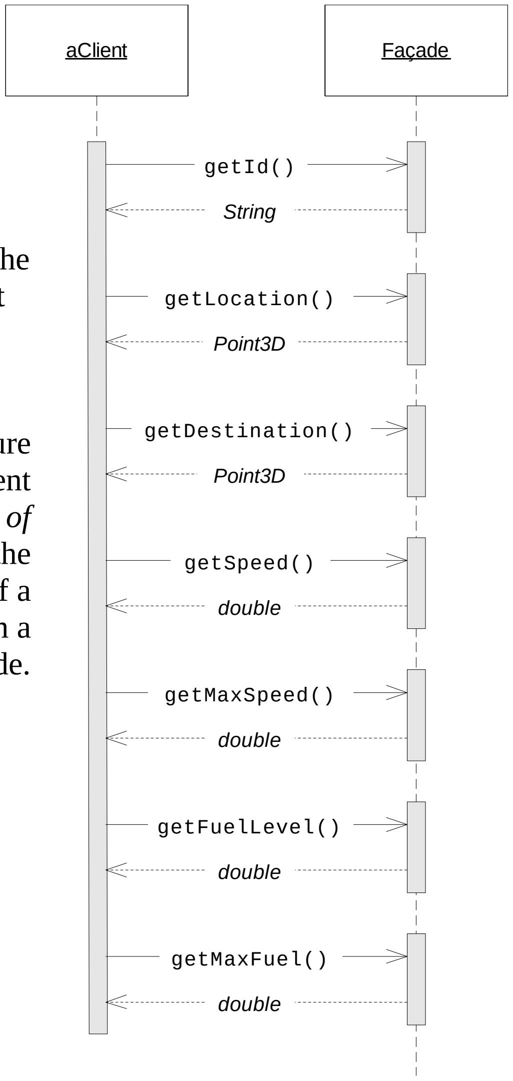
In this way, one call to the Façade is needed to get all the data that the Façade is willing to give for a domain object. Without using this pattern, many calls might be needed to access the key data of a domain object needed to perform a particular function.

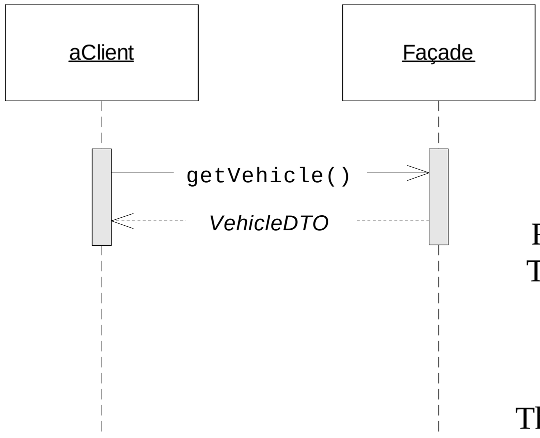


The Façade Pattern:

Accessing data via Façade *without using* the Data Transfer Object design pattern:

The following figure shows how a client makes a *sequence of calls* to retrieve the various elements of a Vehicle object from a Façade.





Accessing data via Façade *using* the Data Transfer Object design pattern:

This figure shows how a client makes a *single call* to retrieve all the various elements of a Vehicle object via DTO from a Façade.

The Façade Pattern:

Data Transfer Object Creation

The Data Transfer Objects can be created by the Façade itself, or by some other “Assembler” object (this is called the Transfer Object Assembler Pattern).

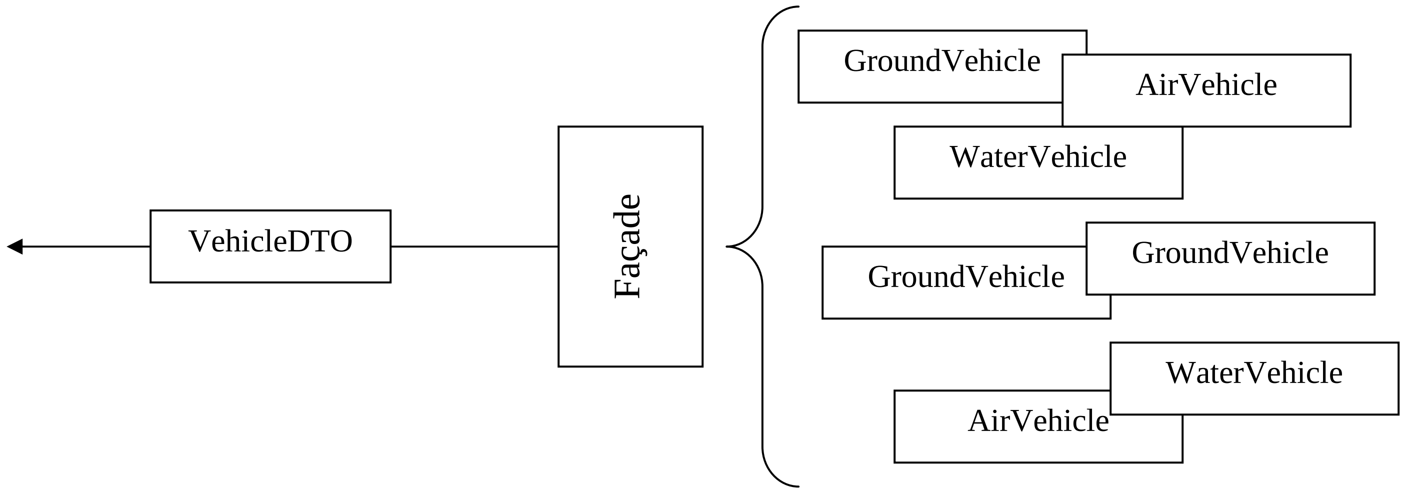
Example: Data Transfer Object Creation Within the Vehicle Façade

```
public class VehicleManager
{
     […]

     protected ArrayList<Vehicle> vehicleList = new ArrayList<Vehicle>();

     […]

     public VehicleDTO getVehicle (int i)
     {
          Vehicle v = vehicleList.get(i);
          return new VehicleDTO (v.getIdentifier(), v.getLocation());
     }

     […]
}
```

The Façade Pattern:

Data Transfer Object Creation

The Data Transfer Objects can be created by the Façade itself, or by some other “Assembler” object (this is called the Transfer Object Assembler Pattern).



New classes (like “SpaceVehicle”) can be added to those that the Façade maintains without affecting client code.

Client code never needs to know what objects exist behind the Façade.

Data Transfer Object

Design Pattern

Summary:

- The Data Transfer Object (DTO) pattern is used in conjunction with the “Façade” pattern to encapsulate the key data of a domain object in a simple class (the Data Transfer Object). The Façade then returns the DTO object (not individual data elements) to clients.

- DTO’s are simple data objects made up of public data attributes containing copies of key data of the domain object.

- The Data Transfer Object (DTO) pattern allows clients to make a single call to retrieve all the various elements of a domain object via DTO from a Façade.

- The use of Data Transfer Objects supports Object-Oriented Abstraction.

- New classes can be added to those that the Façade maintains without affecting client code.