Design Patterns

The Strategy Design Pattern

The Concepts:

Interface

An interface is a named collection of method definitions (without implementations). A class that implements the interface agrees to implement all the methods defined in the interface, thereby agreeing to certain behavior.

You use an interface to define a protocol of behavior that can be implemented by any class anywhere in the class hierarchy. Interfaces are useful for the following:

- Capturing similarities among unrelated classes without artificially forcing a class relationship
- Declaring methods that one or more classes are expected to implement
- Revealing an object's programming interface without revealing its class
- Modeling multiple inheritance, a feature that some object-oriented languages support that allows a class to have more than one super-class

Strategy

Design Pattern

The Concepts:

Delegation

Delegation is a technique where an object outwardly expresses certain behavior, but in reality delegates (passes off) responsibility of providing that behavior to an associated object. This process is transparent to “client” code.

Delegation can be viewed as a relationship between objects where one object forwards certain method calls to another object, called its delegate. Delegation is a powerful design/reuse technique.

The primary advantage of delegation is run-time flexibility – the delegate can easily be changed at run-time.

Strategy

Design Pattern

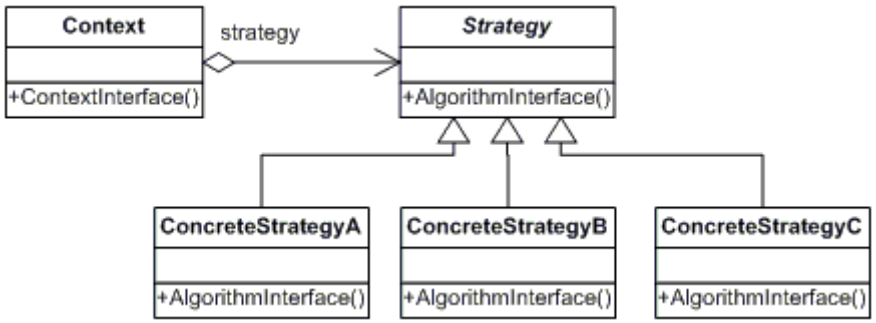
The Concepts:

The Strategy Pattern

Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

The Strategy pattern lets you build software as a loosely coupled collection of interchangeable parts. This is done by defining a family of algorithms, encapsulating each one, and then making them interchangeable.

The Strategy pattern lets the algorithm vary independently from clients that use it.



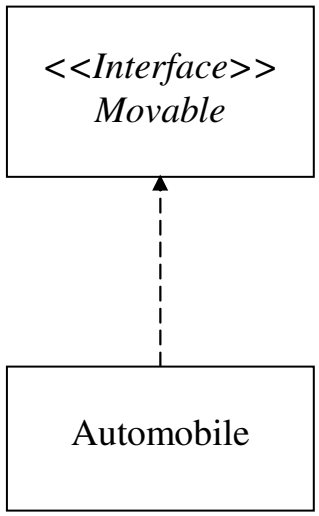
Strategy
Design Pattern

Background:

Issues With Interfaces (Java)

An interface in Java is much like an abstract class, but with no constructors, method bodies, or instance variables. An interface can be used to define a restricted view of a group of objects, or to specify a minimal set of features a group of objects is expected to have for some particular purpose.

To "use" an interface, we need a class that implements the interface. For a class to implement an interface, the class must provide a method body to all of the abstract methods defined in the interface.




```
public interface Movable
{

    public void move(int seconds);

}
```

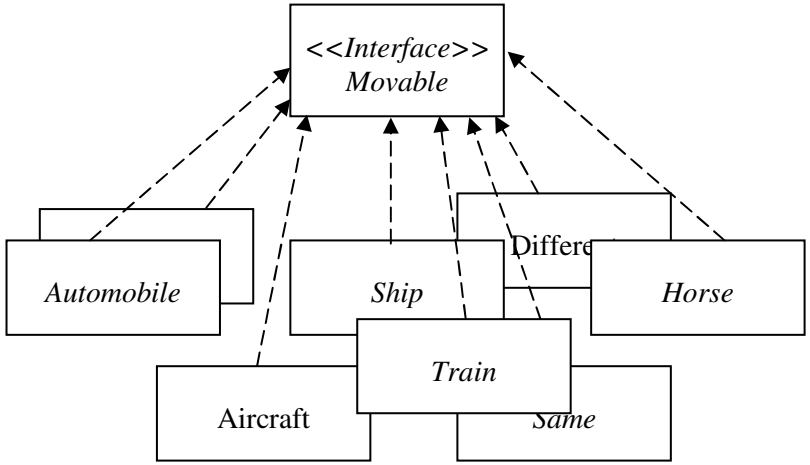
```
public class Automobile implements Movable
{
    private double speed;
    private Point location;

    public Automobile ()
    {…}

    public void move(int seconds)
    {
    // Automobile movement code goes here
    }
}
```

Strategy
Design Pattern

Background (cont.):

When one size does not fit all…

This simple interface implementation technique works well when most or all of the implementing classes require unique behavior implementations. However, when several of the implementing classes require identical implementations, this technique leads to code and data attribute duplication – a maintenance headache.




```
public class Automobile implements Movable
{
      private double speed;
      private Point location

      public Automobile ()
      {…}

      public void move(int seconds)
      {
      // Movement code goes here
      }
}
```

```
public class Aircraft implements Movable
{
      private double speed;
      private Point location

      public Aircraft ()
      {…}

      public void move(int seconds)
      {
      // Unique Aircraft-Related Movement
      // code goes here
      }
}
```

```
public class Train implements Movable
{
      private double speed;
      private Point location

      public Train ()
      {…}

      public void move(int seconds)
      {
      // Same Movement code duplicated here
      }
}
```

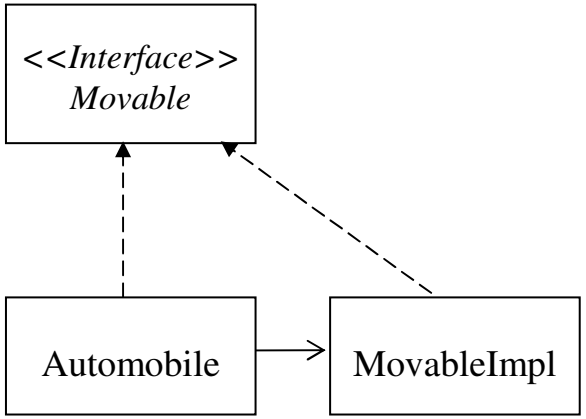
```
public class Ship implements Movable
{
      private double speed;
      private Point location

      public Ship ()
      {…}

      public void move(int seconds)
      {
      // Same Movement code duplicated here
      }
}
```

```
public class Horse implements Movable
{
      private double speed;
      private Point location

      public Horse ()
      {…}

      public void move(int seconds)
      {
      // Same Movement duplicated goes here
      }
}
```

Background (cont.):

Delegate the Details!

To solve the problem of duplicated code and data attribute definitions, the common behaviors and related data can be encapsulated in a class specifically designed to implement an interface (sometimes referred to as an "implementation" class or "Impl" class).




```
public interface Movable
{

    public void move(int seconds);

}
```

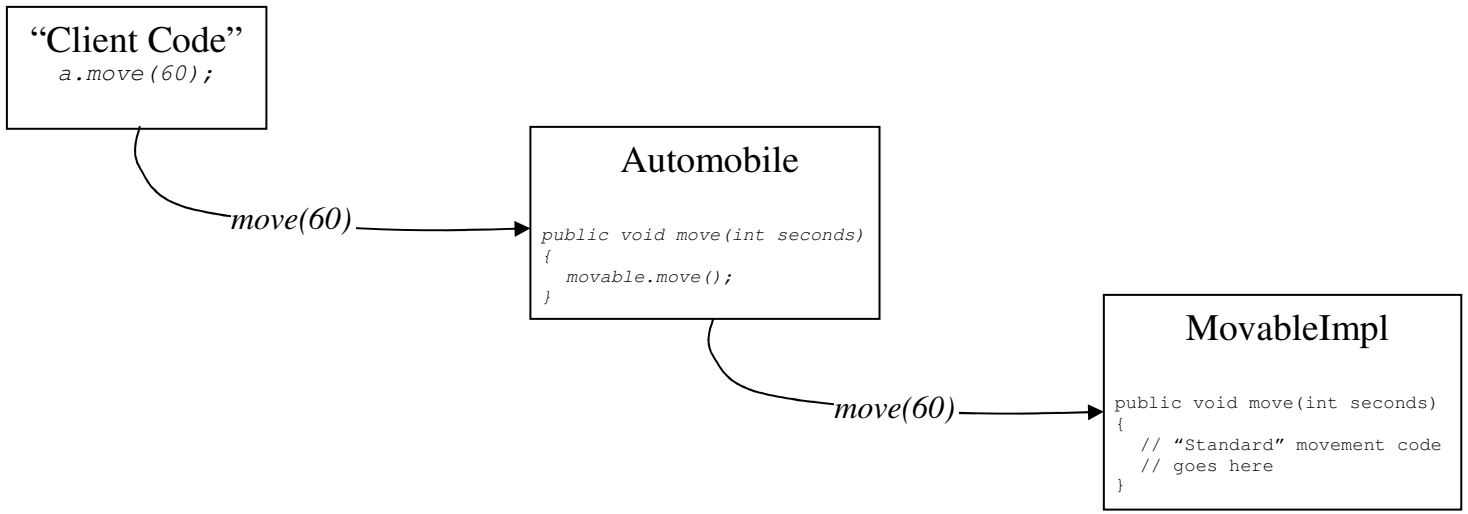
```
public class Automobile
                implements Movable
{
    private Movable movable;

    public Automobile ()
    {…}

    public void move(int seconds)
    {
        movable.move();
    }
}
```

```
public class MovableImpl
                implements Movable
{
    private double speed;
    private Point location;

    public MovableImpl ()
    {…}

    public void move(int seconds)
    {
    // "Standard" movement code goes here
    }
}
```

Background (cont.):

To delegate…

Classes that require the same implementation of some or all interface behaviors can now "delegate" the interface-defined behaviors to this Impl object.



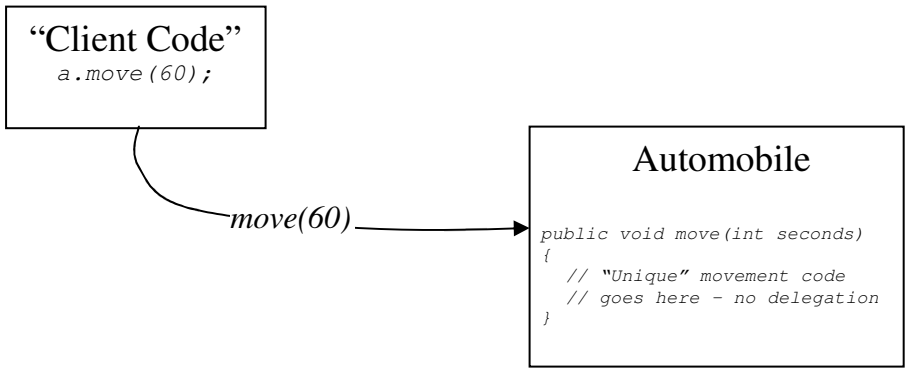
This delegation can be simple, as shown above, or more complex – spanning multiples classes, processes or applications.

Background (cont.):

…or not to delegate…

Classes that require *unique* implementations of some of the interface behaviors need not delegate the interface-defined behaviors to this Impl class. Instead, the implementing class can define a specific implementation of one or more behaviors locally.

Those with unique implementations of *all* interface behaviors need not own an instance of the Impl class at all.



Strategy
Design Pattern

The Problem:

…what a tangled web we weave…

Situations arise when the behavioral needs of a method can fundamentally change depending upon application conditions. Handling this situation in a single "Impl" class, this can result in complex, brittle code using "switch" or "if" statements to modify the behavior.

```
public class MovableImpl
               implements Movable
{
    private double speed;
    private Point location;

    private double ruralData;
    private double urbanData;
    private double offRoadData;

    public MovableImpl ()
    {…}

    public void move(int seconds)
    {
        switch(indicator)
        {
             case 0: // "Rural" movement algorithm
             {
             // Movement code goes here
             }
             case 1: // "Urban" movement algorithm
             {
             // Alternate 1 movement code goes here
             }
             case 2: // "Off Road" movement algorithm
             {
             // Alternate 2 movement code goes here
             }
             […]
        }
    }
}
```

This complex code requires modification whenever changes to any movement algorithms are needed (bug fixes, upgrades, refactoring, etc.) or when new movement algorithms are added.

Additionally, the various algorithms may have differing data needs, which can result in the Impl class carrying many data attributes that an individual algorithm never uses.

The Solution:

The Strategy Pattern

Using the Strategy design pattern we can avoid this problem.

As the Strategy pattern dictates, we encapsulate each of the identified algorithms in *separate* Impl classes, and make them interchangeable. The Strategy design pattern embodies two fundamental tenets of object-oriented (OO) design:

- Encapsulate the Concept that Varies
- Program to an Interface, Not an Implementation

Use the strategy pattern when:

- Many related classes differ only in their behavior.
- You need different variants of an algorithm.
- An algorithm uses data that client shouldn't know about.
- You need to vary a behavior's algorithm at run-time.

Strategy
Design Pattern

The Strategy Pattern:

Making use of the Strategy pattern will leave us with a loosely coupled collection of interchangeable, maintainable, and reusable parts, rather than a monolithic, tightly coupled system.

```
public class MovableRuralImpl
                implements Movable
{
    private double speed;
    private Point location;

    private double ruralData;

    public MovableRuralImpl ()
    {…}

    public void move(int seconds)
    {
    // Code for coarse-grained “rural”
    // movement algorithm goes here
    }
}
```

```
public class MovableUrbanImpl
                implements Movable
{
    private double speed;
    private Point location;

    private double urbanData;

    public MovableUrbanImpl ()
    {…}

    public void move(int seconds)
    {
    // Code for fine-grained “urban”
    // movement algorithm goes here
    }
}
```

```
public class MovableOffRoadImpl
                implements Movable
{
    private double speed;
    private Point location;

    private double offRoadData;

    public MovableOffRoadImpl ()
    {…}

    public void move(int seconds)
    {
    // Code for alternate “off-road”
    // movement algorithm goes here
    }
}
```

These classes can be further refactored to create a base class, “MovableBaseImpl”, that defines the data and behavior common to all 3 classes.

Often, “null” Impl classes are created as placeholders for the actual behaviors.

```
public class MovableNullImpl
                implements Movable
{
    public MovableImpl ()
    {}

    public void move(int seconds)
    {}
}
```

Strategy
Design Pattern

The Strategy Pattern:

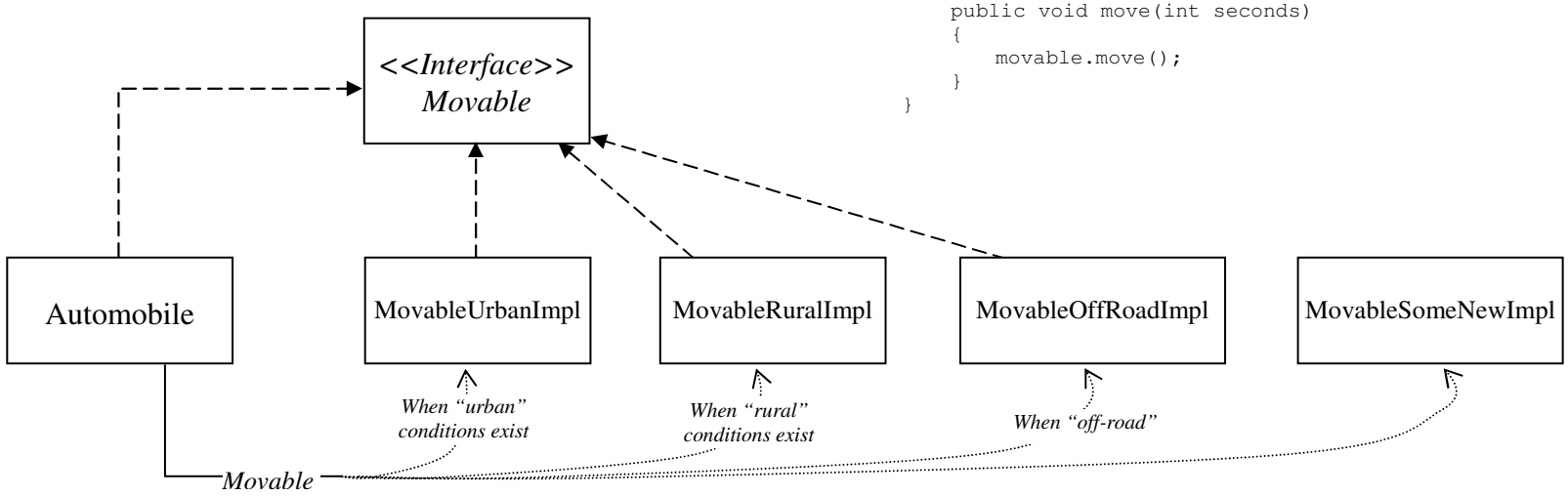
Do what I mean, not what I say!

The actual implementation object that is used can be changed dynamically during application execution depending upon conditions. Additionally, new implementations can be added to the application without affecting the existing code.

```
public class Automobile
        implements Movable
{
    private Movable movable;

    public Automobile ()
    {…}

    public void move(int seconds)
    {
        movable.move();
    }
}
```

Can point to any "Movable" object:

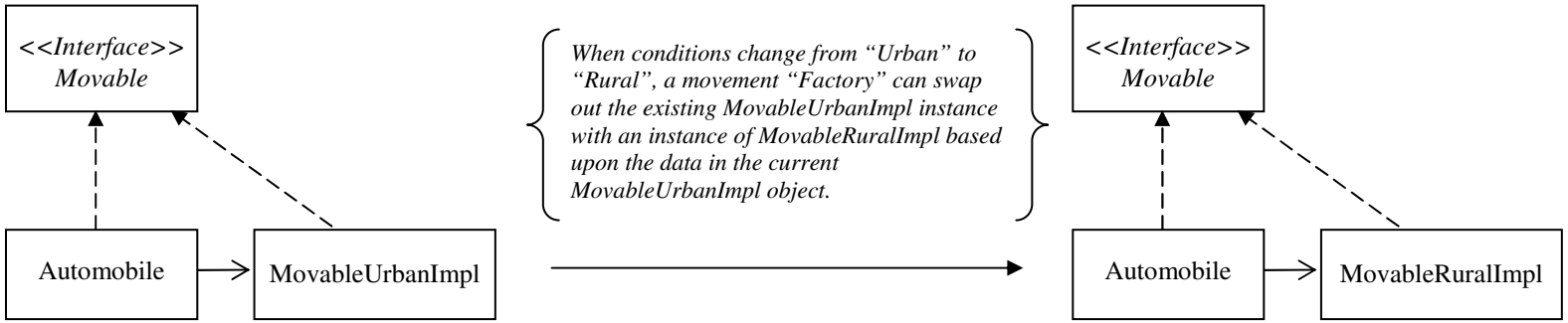
- *MovableUrbanImpl*
- *MovableRuralImpl*
- *MovableOffRoadImpl*
- *MovableSomeNewImpl*



Strategy
Design Pattern

The Strategy Pattern:

The implementation object that is used can be changed by the implementing class itself (a less optimal choice – we don't want the client to have to know what Impl's exist and when they should be used/changed) or by another class – such as a variant of the Factory Pattern – that can convert one Impl type to another when appropriate.



Strategy
Design Pattern

Summary:

… and they lived happily ever after!

- Java interfaces allow the specification of behaviors that a group of objects is expected to have for some particular purpose.
- Code and data attribute duplication can result when several classes require identical interface implementations.
- Creating “implementation” classes is a great way to encapsulate interface implementations used by many implementors. Classes can now “delegate” the interface-defined behaviors to the associated implementation object.
- Incorporating multiple algorithms into the same implementation class leads to complex, brittle code.
- Following the Strategy design pattern, multiple implementation classes can be created when a variety of algorithms are needed to handle interface behaviors.
- Implementation objects can be changed dynamically during application execution depending upon conditions.
- This results in a set of loosely coupled interchangeable, extensible, maintainable, and reusable parts