Introduction: Motivation for Design Patterns

- Design pattern defined:
  
  A solution to a problem in a context. (p 579)

  - The context is the situation in which the pattern applies
    - The situation should be recurring
  
  - The problem is the goal to be achieved in the context
    - This includes any constraints that apply
  
  - The solution is the pattern
    - A general design that can be applied to problems in this context
    - It achieves the goal while satisfying the constraints

- Advantages of design patterns

  1. Provides a shared vocabulary
  2. Promotes thinking at a higher level of abstraction (i.e., above the object level)
Strategy Pattern: Basic Inheritance

- Basic inheritance implemented in terms of superclasses (often abstract) and subclasses

  - Consider

  ```
  Superclass
  
  commonBehavior1()
  commonBehavior2()
  commonBehavior3()
  commonToSome1()
  commonToSome2()
  
  SubClass1
  
  specificBehavior1a()
  specificBehavior1b()
  
  SubClass2
  
  specificBehavior2a()
  specificBehavior2b()
  commonToSome1()
  ```

- *commonBehavior* methods are behaviors shared by all subclasses
- *commonToMost* methods are behaviors shared by *most* subclasses
  - They are overridden in subclasses for which the default behavior does not apply
- This can lead to problems

Strategy Pattern: Problems with Inheritance

1. When a new subclass is added, it is possible that it does not share all of the behaviors common to the original set of subclasses
   • These exceptions will need to be overridden in the newly added subclass

2. If the superclass is modified (e.g., a new behavior is introduced)
   (a) All subclasses that are exceptions to this behavior will need to be modified
   (b) If the behavior is erroneously thought to be common (no overriding in subclasses), subclasses may exhibit incorrect behavior

3. Changes to runtime behavior are also difficult
• An alternative to inheritance is to use an interface
  – Create an interface for every behavior that is not shared by all
  – Only those subclasses that exhibit such a behavior will implement that interface
  – Consider

- This introduces new problems
Strategy Pattern: Problems with Interfaces

1. Precludes code reuse
2. Duplication of code
3. Lots of work when a behavior is modified
Strategy Pattern: Design Principle 1

- Design Principle 1 (p9)

  *Id the aspects of your app that vary and separate them from what stays the same.*

  - This is accomplished by encapsulating the parts of the code that vary
  - They can then be modified or extended without affecting the fixed aspects of the code
  - We would also like to be able to choose a particular type of behavior at runtime
• Design Principle 2 (p11)

*Program to an interface, not an implementation.*

– *Interface* does not necessarily refer to an interface as implemented in Java.
– Rather, it means a class from which others may inherit, usually an abstract class or Java interface.
– The principle could be rephrased as "*Program to a supertype*”
  * I.e., variables should be declared to be a superclass type, and not a concrete type
  * This allows the use of polymorphism
– Example:
  * Programming to an implementation:

```java
SpecificClass1 object1 = new SpecificClass1();
object1.performBehavior1();
```

* Programming to an interface:

```java
//SpecificClass1 inherits from GeneralClass1
GeneralClass1 object1 = new SpecificClass1();

//performGeneralBehavior1() is defined in GeneralClass1
//and over-ridden in each subclass
object1.performGeneralBehavior1();
```

• In this case, since *object1* is declared to be of type *GeneralClass1* but is an object of type *SpecificClass1*, the call
  ```java
  object1.performGeneralBehavior1();
  ```
  will use polymorphism to correctly access the implementation specific to objects of type *SpecificClass1*
Strategy Pattern: Design Principle 2 (2)

• Overview of process:
  – Implement a set of classes for each behavior not common to all subclasses
  – Each behavior will consist of a
    1. Superclass that represents the generic behavior, and
    2. Subclasses, each of which implements a specific version of that behavior
  – Each set of specific behaviors will inherit from the generic behavior of that type
    * The generic behaviors represent the interfaces
    * Each specific behavior implements one of these interfaces
  – To implement:
    1. Create the interfaces
      * In each, include a single method for that behavior \((\text{intended to be overridden})\)

\[
\begin{align*}
\text{public interface BehaviorType1} & \{ \\
& \quad \text{public void commonToSome1();} \\
& \}\ \\
\text{public interface BehaviorType2} & \{ \\
& \quad \text{public void commonToSome2();} \\
& \}\ \\
\end{align*}
\]

...
Strategy Pattern: Design Principle 2 (3)

2. Create a subclass for each specific behavior

* Each overrides the method of its superclass, providing a concrete implementation for that specific version of the behavior

```java
public class Type1Version1 implements BehaviorType1 {
    public void commonToSome1() {
        // actual implementation specific to this version
    }
}

public class Type1Version2 implements BehaviorType1 {
    public void commonToSome1() {
        // actual implementation specific to this version
    }
}

// This makes the behaviors independent of the classes that exhibit them
```
• Class diagram for above example using Design Principle 2:
Strategy Pattern: Design Principle 3

• Design Principle 3 (p23)

  Favor composition over inheritance.

  – I.e., favor HASA over ISA constructions
  – This approach promotes
    1. Encapsulation of families of algorithms
    2. Ability to change behavior at run time

• This principle can be incorporated into our design
Strategy Pattern: Coordinating Classes with the Interface

- Making the superclass and subclasses work with the behavior interface

1. In the superclass
   (a) Add an instance variable to the superclass for each generic behavior that isn’t shared by all subclasses

```java
public abstract Superclass {
    BehaviorType1 type1Behavior;
    BehaviorType2 type2Behavior;

    public Superclass () {
    }

    // Behaviors specific to each subclass - overridden in each
    public abstract someBehavior () {
        // implementation
    }
    ...

    // Shared behaviors
    public xxx commonBehavior1 () {
        // implementation
    }

    public xxx commonBehavior2 () {
        // implementation
    }
    ...
}
```
(b) Add a method for each generic behavior (named `performXXX`) which simply calls the single method of the generic behavior’s implementation.

```java
public abstract Superclass {
    BehaviorType1 type1Behavior;
    BehaviorType2 type2Behavior;

    public Superclass () {
    }

    //Behaviors specific to each subclass - overridden in each
    public abstract xxx someBehavior () {
        //implementation
    }

    //Generic behaviors
    public void performBehaviorType1() {
        type1Behavior.commonToSome1();
    }

    public void performBehaviorType2() {
        type2Behavior.commonToSome2();
    }

    //Shared behaviors
    public xxx commonBehavior1 () {
        //implementation
    }

    public xxx commonBehavior2 () {
        //implementation
    }
}
```
2. In the subclasses
   (a) Instantiate an inherited instance variable for each specific behavior that it exhibits
   (b) Eliminate inheritance from the behavior interface
public class Subclass3 extends Superclass{

    public Subclass3() {
        type1Behavior = new Type1Version1;
        type2Behavior = new Type2Version1;
    }

    public xxx someBehavior () {
        \\implementation
    }

    ...

3. Client code

    public class Client {
        public static void main(String args[]) {
            Superclass object1 = new Subclass3();

            object1.performType1Behavior();
            object1.performType2Behavior();
        }
    }
Strategy Pattern: Changing Behavior Dynamically

- Behavior can be changed at runtime (as opposed being fixed at compilation) by

  1. Adding parameterized set methods to the superclass
     - The parameters represent a particular behavior
     - Client can then specify a particular version of a behavior at any time by sending a message to the subclass

     ```java
     public abstract Superclass {
         BehaviorType1 type1Behavior;
         BehaviorType2 type2Behavior;
         ...

         public void setBehaviorType1(BehaviorType1 behaviorType) {
             type1Behavior = behaviorType;
         }

         public void setBehaviorType2(BehaviorType1 behaviorType) {
             type2Behavior = behaviorType;
         }
         ...
     }
     
     public class Client {
         public static void main(String args[]) {
             Superclass object1 = new Subclass3();

             object1.setBehaviorType1(new Type1Version1)
             object1.performType1Behavior();
             object1.setBehaviorType1(new Type1Version2)
             object1.performType1Behavior();
         }
     }
     ```
Strategy Pattern: The Pattern

- Strategy pattern (p 24)
  
  Defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

- The Strategy pattern incorporates design principles 1, 2, and 3.