CptS 487
Software Design and Architecture

Lesson 25

Design Patterns 9:

Adapter & Strategy

WASHINGTON STATE UNIVERSITY
World Class. Face to Face.

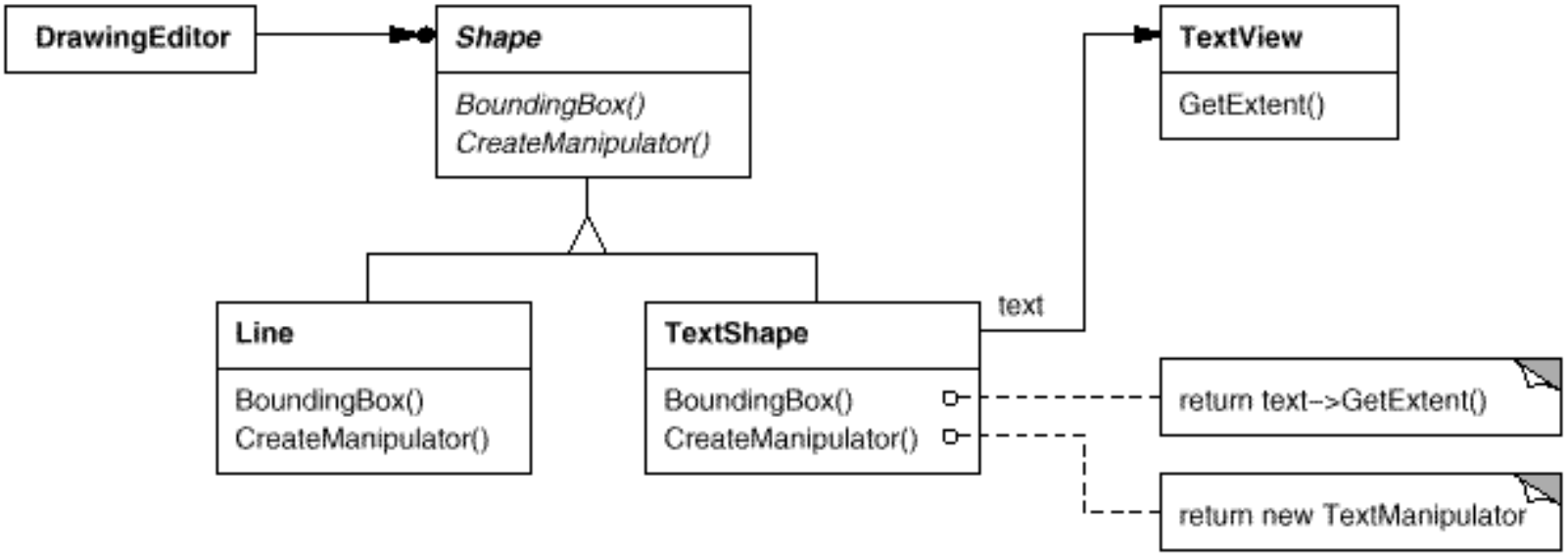
Instructors:

1. Adapter (Class, Object structural pattern)

- Intent
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- Also Known As
  - Wrapper
- Motivation
  - Sometimes a toolkit or class library can not be used because its interface is incompatible with the interface required by an application
  - We can not change the library interface, since we may not have its source code
  - Even if we did have the source code, we probably should not change the library for each domain-specific application

1. Adapter

- Motivation
  - Example



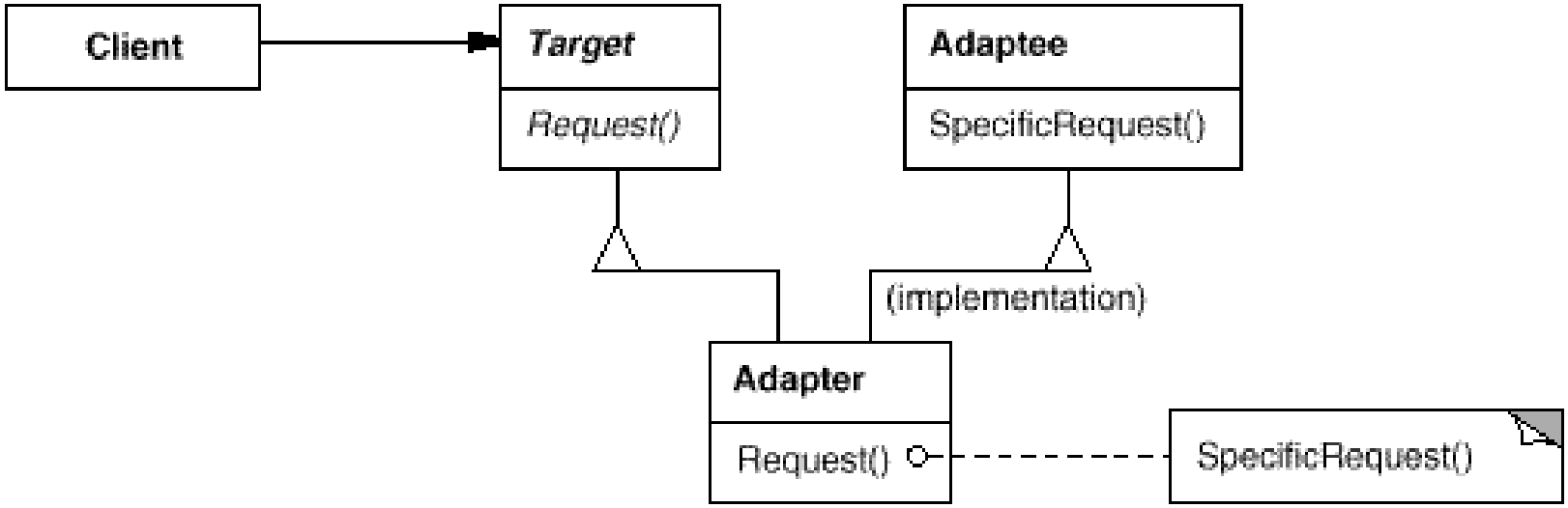
—TextShape *adapts* the TextView interface to Shape's.

1. Adapter

- Structure
  - Adapter can adapt the Adaptee interface to Target's in two ways:
    1. By inheriting Target's interface and Adaptee's implementation
       ⇒ Multiple inheritance
    2. By inheriting Target's interface and by composing an Adaptee instance within the Adapter

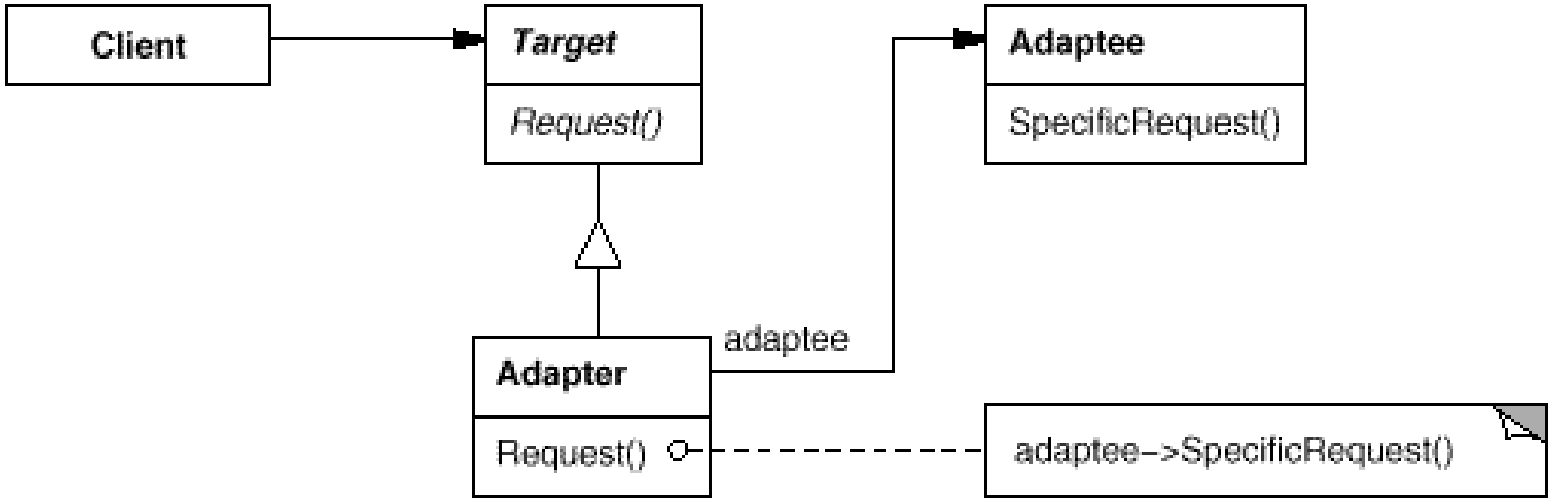
1. Adapter

- Structure
  1. A class adaptor uses multiple inheritance to adapt one interface to another



1. Adapter

- Structure
  2. An object adapter relies on object composition

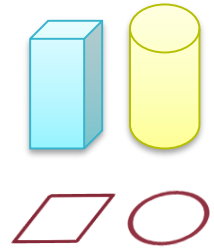


1. Adapter

- Applicability
  - Use the Adapter pattern when
    - You want to use an existing class, and its interface does not match the one you need
    - You want to create a reusable class that cooperates with unrelated classes with incompatible interfaces

- Implementation Issues
  - How much adapting should be done?
    - Simple interface conversion that just changes operation names and order of arguments
    - Totally different set of operations
  - Does the adapter provide two-way transparency?
    - A two-way adapter supports both the Target and the Adaptee interface. It allows an adapted object (Adapter) to appear as an Adaptee object or a Target

Adapter Pattern Example 1



- The classic **round pegs** and **square pegs**!

- Here's the SquarePeg class:

```
/* The SquarePeg class.*/
public class SquarePeg {
 public void insert(String str) {
   System.out.println("SquarePeg insert(): " + str);
 }
}
```

This is the Target class.

- And the RoundPeg class:

```
/*The RoundPeg class*/
public class RoundPeg {
 public void insertIntoHole(String msg) {
   System.out.println("RoundPeg insertIntoHole(): " + msg);
 }
}
```

This is the Adaptee class.

Adapter Pattern Example 1

- If a client only understands the SquarePeg interface for inserting pegs using the `insert()` method, how can it insert round pegs?
  - Using a peg adapter!
- Here is the PegAdapter class as an object adapter:

Inherits SquarePeg interface

```
public class PegAdapter extends SquarePeg {
    private RoundPeg roundPeg;
    public PegAdapter(RoundPeg peg) {this.roundPeg = peg;}
    public void insert(String str)
     {roundPeg.insertIntoHole(str);}
}
```

Delegates the insert method to RoundPeg

This is the Adapter class.
It adapts a RoundPeg to a SquarePeg.
Its interface is that of a SquarePeg.

Adapter Pattern Example 1

- Typical client program:

```
// Test program for Pegs (client).
public class TestPegs {
    public static void main(String args[]) {
        // Create a square peg.
        SquarePeg squarePeg = new SquarePeg();
        // Do an insert using the square peg.
        squarePeg.insert("Inserting square peg...");
```

We'd like to do an insert using the round peg. But this client only understands the `insert()` method of pegs, not a `insertIntoHole()` method.

⇒The solution: create an adapter that adapts a square peg to a round peg!

```
        RoundPeg roundPeg = new RoundPeg();
        PegAdapter adapter = new PegAdapter(roundPeg);
        adapter.insert("Inserting round peg...");
    }
}
```

- Client program output:

```
SquarePeg insert(): Inserting square peg...
RoundPeg insertIntoHole(): Inserting round peg...
```

Adapter Pattern Example 1

- How to implement PegAdapter as a class adapter using multiple inheritance
  - We can do this in C++, but we can't do this in Java
  - In Java we can have our adapter class implement two different Java interfaces!

```
 /* The PegAdapter as a class adapter (in C++). */
class PegAdapter : public SquarePeg, private  RoundPeg{
   public:
      PegAdapter() {}
      void insert(char *str) {insertIntoHole(str);}
}
```

- The first inheritance (SquarePeg) inherits the interface
  - Inherits publicly
- The second inheritance inherits the implementation
  - Inherits privately

Adapter Pattern Example 2

- Notice in Example 1 that the PegAdapter adapts a RoundPeg to a SquarePeg. The interface for PegAdapter is that of a SquarePeg.
- What if we want to have an adapter that acts as a SquarePeg or a RoundPeg? Such an adapter is called a two-way adapter.
- One way to implement two-way adapters is to use multiple inheritance.
  - We can do this in C++, but we can't do this in Java
  - In Java we can have our adapter class implement two different Java interfaces!

Adapter Pattern Example 2

- The new two-way PegAdapter in C++:

```
 /* The PegAdapter class (in C++). This is the two-way
   adapter class. */

class PegAdapter : public SquarePeg, public RoundPeg{

    public:
    PegAdapter() {}
    void insert(char *str) {insertIntoHole(str);}
    void insertIntoHole(char *msg){ insert(msg);}

}
```

Adapter Pattern Example 2

- The new two-way PegAdapter in C++:
  - In Java, our adapter class can implement two different Java interfaces!
- Here are the interfaces for round and square pegs:

```
/* The IRoundPeg interface. */
 public interface IRoundPeg {
    public void insertIntoHole(String msg);
 }

 /* The ISquarePeg interface. */
 public interface ISquarePeg {
    public void insert(String str);
 }
```

Adapter Pattern Example 2

- Here are the new RoundPeg and SquarePeg classes. These are essentially the same as before except they now implement the appropriate interface:

```
// The RoundPeg class.
 public class RoundPeg implements IRoundPeg {
    public void insertIntoHole(String msg) {
       System.out.println("RoundPeg insertIntoHole(): " + msg);
    }
 }

 // The SquarePeg class.
 public class SquarePeg implements ISquarePeg {
    public void insert(String str) {
       System.out.println("SquarePeg insert(): " + str);
    }
 }
```

Adapter Pattern Example 2

- And here is the new PegAdapter:

```
/* The PegAdapter class.
 * This is the two-way adapter class. */

 public class PegAdapter implements ISquarePeg, IRoundPeg {

     private RoundPeg roundPeg;
     private SquarePeg squarePeg;

     public PegAdapter(RoundPeg peg)
      {this.roundPeg = peg;}
     public PegAdapter(SquarePeg peg)
      {this.squarePeg = peg;}

     public void insert(String str)
      {roundPeg.insertIntoHole(str);}
     public void insertIntoHole(String msg)
      {squarePeg.insert(msg);}
 }
```

Adapter Pattern Example 2

- A client that uses the two-way adapter:

```
// Test program for Pegs.
public class TestPegs {
  public static void main(String args[]) {

    // Create some pegs.
    RoundPeg roundPeg = new RoundPeg();
    SquarePeg squarePeg = new SquarePeg();

    // Do an insert using the square peg.
    squarePeg.insert("Inserting square peg...");

    // Create a two-way adapter and do an insert with it.
    ISquarePeg roundToSquare = new PegAdapter(roundPeg);
    roundToSquare.insert("Inserting round peg...");

  //-----------------
    // Do an insert using the round peg.
    roundPeg.insertIntoHole("Inserting round peg...");

    // Create a two-way adapter and do an insert with it.
    IRoundPeg squareToRound = new PegAdapter(squarePeg);
    squareToRound.insertIntoHole("Inserting square peg...");
  }
}
```

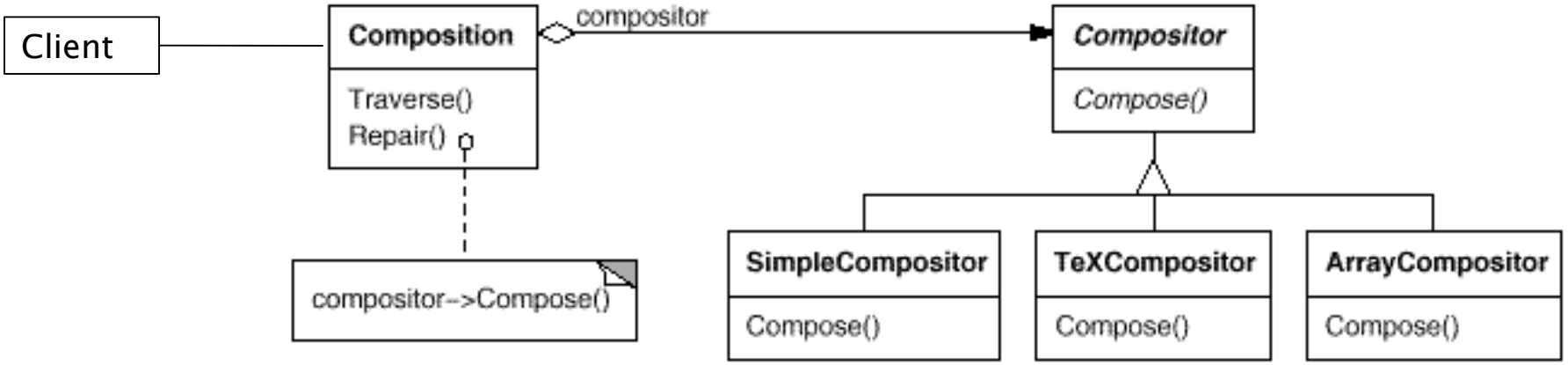
Adapter Pattern Example 2

- Client program output:

```
SquarePeg insert(): Inserting square peg...
RoundPeg insertIntoHole(): Inserting round peg...
RoundPeg insertIntoHole(): Inserting round peg...
SquarePeg insert(): Inserting square peg...
```

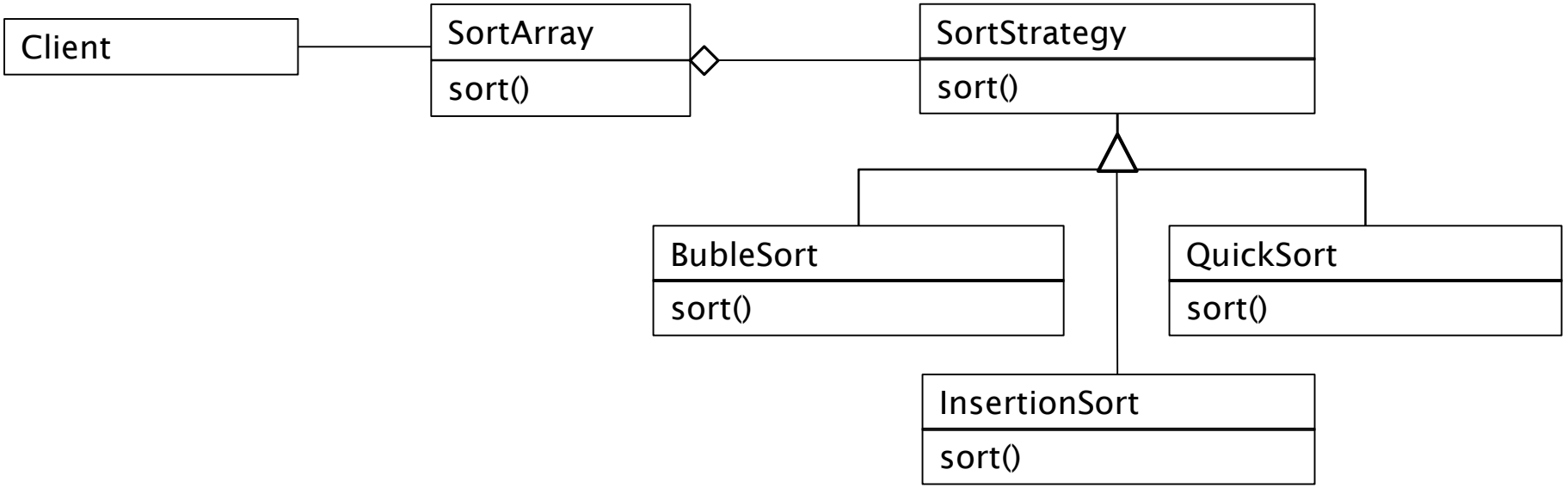
2. Strategy (Object behavioral pattern)

- Intent
  - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Also Known As
  - Policy
- Motivation



Strategy Example

- Situation: A class wants to decide at run-time what algorithm it should use to sort an array. Many different sort algorithms are already available.
- Solution: Encapsulate the different sort algorithms using the Strategy pattern!



2. Strategy

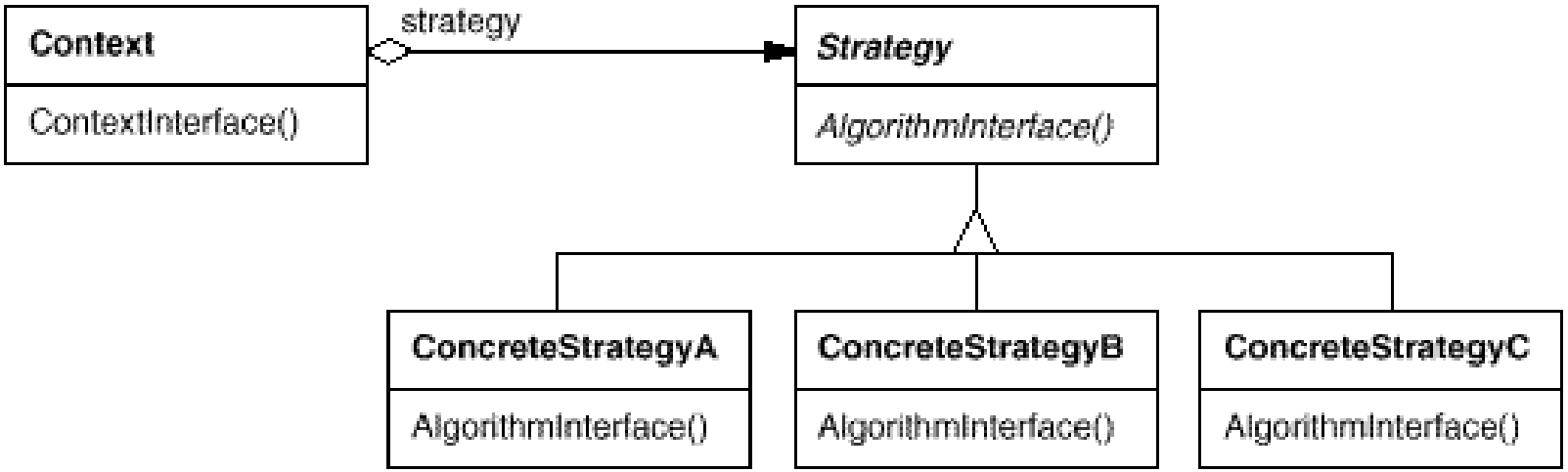
- Applicability

Use the Strategy pattern whenever:

- Many related classes differ only in their behavior
- You need different variants of an algorithm
- An algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
- A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

2. Strategy

- Structure



Strategy Example

- Situation: The rules for calculating tax and currency change according to country. The client wants to change the choice of country at runtime. All rules would be customized according to the country of choice.
- Solution: Encapsulate the different calculation strategies using the Strategy pattern!

Strategy Example

- Implementing different strategies with if-else/switch statements

```
// Handle Tax
switch (myNation) {
    case US:
        // US Tax rules here
        break;
    case Canada:
        // Canadian Tax rules here
        break;
}

// Handle Currency
switch (myNation) {
    case US:
        // US Currency rules here
        break;
    case Canada:
        // Canadian Currency rules
        // here
        break;
}
```

- What happens when there are more variations?
- For example, suppose I need to add Germany to the list of countries and also add language as a result.

- Example is taken from “Design Patterns Explained” by M. Shalloway

Strategy Example

- New code looks like this:

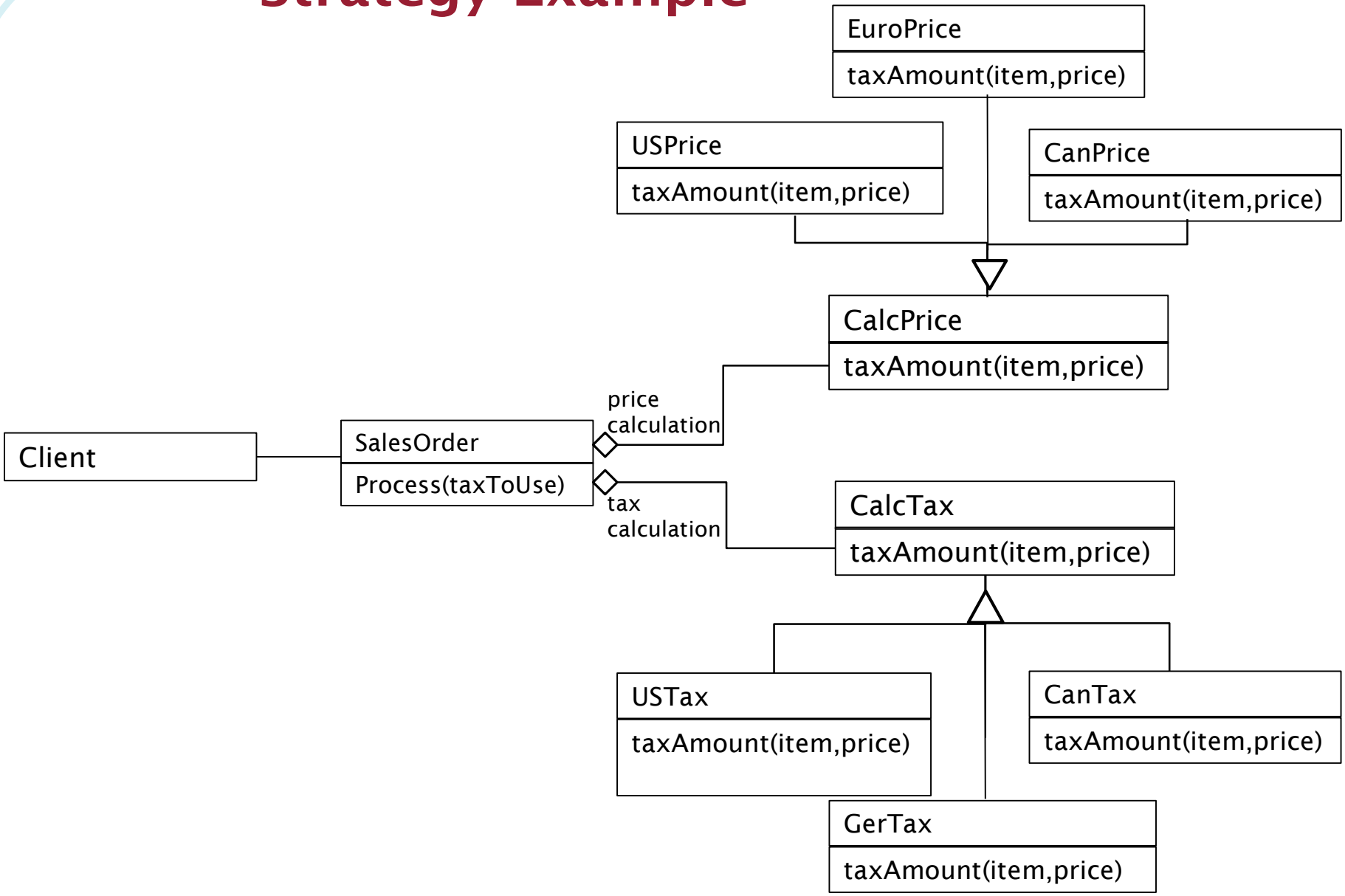
```
// Handle Tax
switch (myNation) {
    case US:
        // US Tax rules here
        break;
    case Canada:
        // Canadian Tax rules here
        break;
    case Germany:
        // Germany Tax rules here
        break;
}

// Handle Currency
switch (myNation) {
    case US:
        // US Currency rules here
        break;
    case Canada:
        // Canadian Currency rules
        // here
        break;
    case Germany:
        // Euro Currency rules here
        break;
}
```

```
// Handle Language
switch (myNation) {
    case US:
    case Canada:
        // use English
        break;
    case Germany:
        // use German
        break;
}
```

- What if we need to start adding variations within a case. Suddenly, things get more complex.
- For example, add French in Quebec.

Strategy Example



Strategy Example

- Solution: Apply the strategy pattern

```
public abstract class CalcTax {
    abstract public double taxAmount(long itemSold,double price);
}

public class CanTax extends CalcTax {

    public void CanTax();

    public double taxAmount(long itemSold, double price) {
       /*calculate tax according to the rules in Canada return it here*/
        return 0.0;
    }
}
public class USTax extends CalcTax {

    public void USTax();

    public double taxAmount(long itemSold,double price) {
       /*calculate tax according to the rules in US return it here*/
        return 0.0;
    }
}
```

- Strategy class
- Concrete strategy class
- Concrete strategy class

Strategy Example

- Context class

```
public class SalesOrder {

   public void process (CalcTax taxToUse) {
         double price= 0;
         // given the tax object calculate the tax
         double tax = taxToUse.taxAmount(itemNumber,price);
     }
}
```

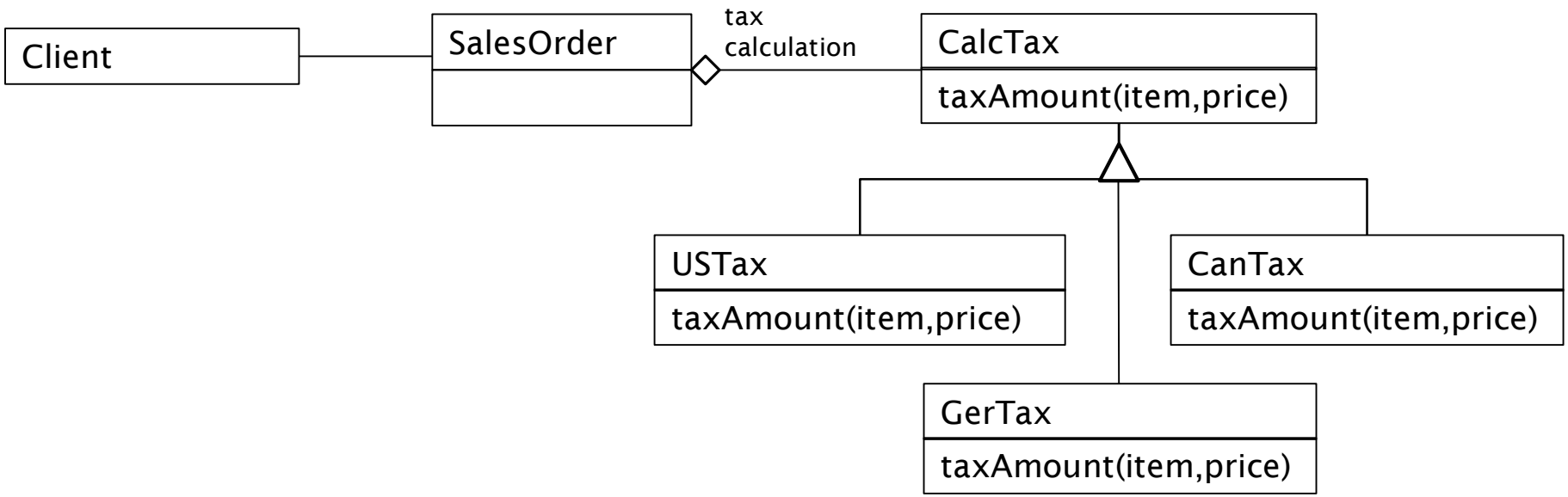
- Client

```
  /*Get the tax rules based on country you are in, i.e., USTax*/
  CalcTax myTax = new USTax();

  SalesOrder mySO= new SalesOrder();
  mySO.process(myTax);
```

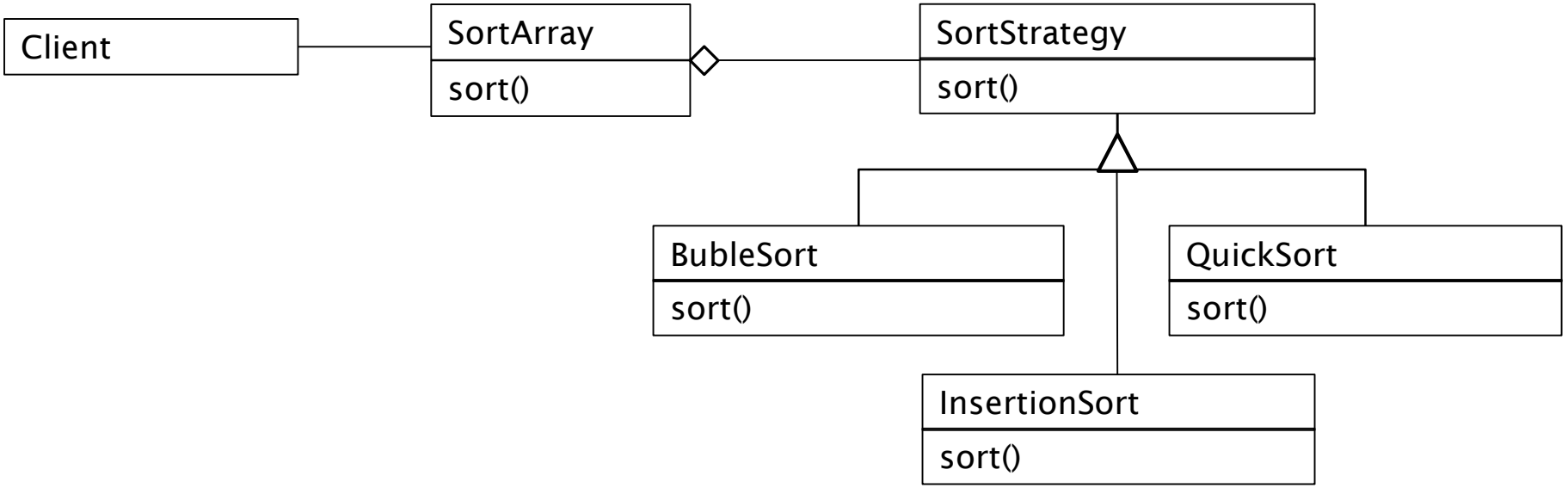
Strategy Example

- Encapsulate the different calculation strategies using the Strategy pattern!



Strategy Example -2

- Situation: A class wants to decide at run-time what algorithm it should use to sort an array. Many different sort algorithms are already available.
- Solution: Encapsulate the different sort algorithms using the Strategy pattern!



2. Strategy

Consequences

- Benefits
  - Provides an alternative to subclassing the Context class to get a variety of algorithms or behaviors
  - Eliminates large conditional statements
  - Provides a choice of implementations for the same behavior
- Liabilities
  - Increases the number of objects
  - All algorithms must use the same Strategy interface

Question:

- Strategy vs. Bridge?