CS 525 - ASD

Advanced Software Development

MS.CS Program

Department of Computer Science

Rene de Jong, MsC.

Maharishi University
OF MANAGEMENT

CS 525 - ASD

Advanced Software Development

Maharishi University
OF MANAGEMENT

Lesson 2

L1: ASD Introduction
L2: Strategy, Template method
L3: Observer pattern
L4: Composite pattern, iterator pattern
L5: Command pattern
L6: State pattern
L7: Chain Of Responsibility pattern

Midterm

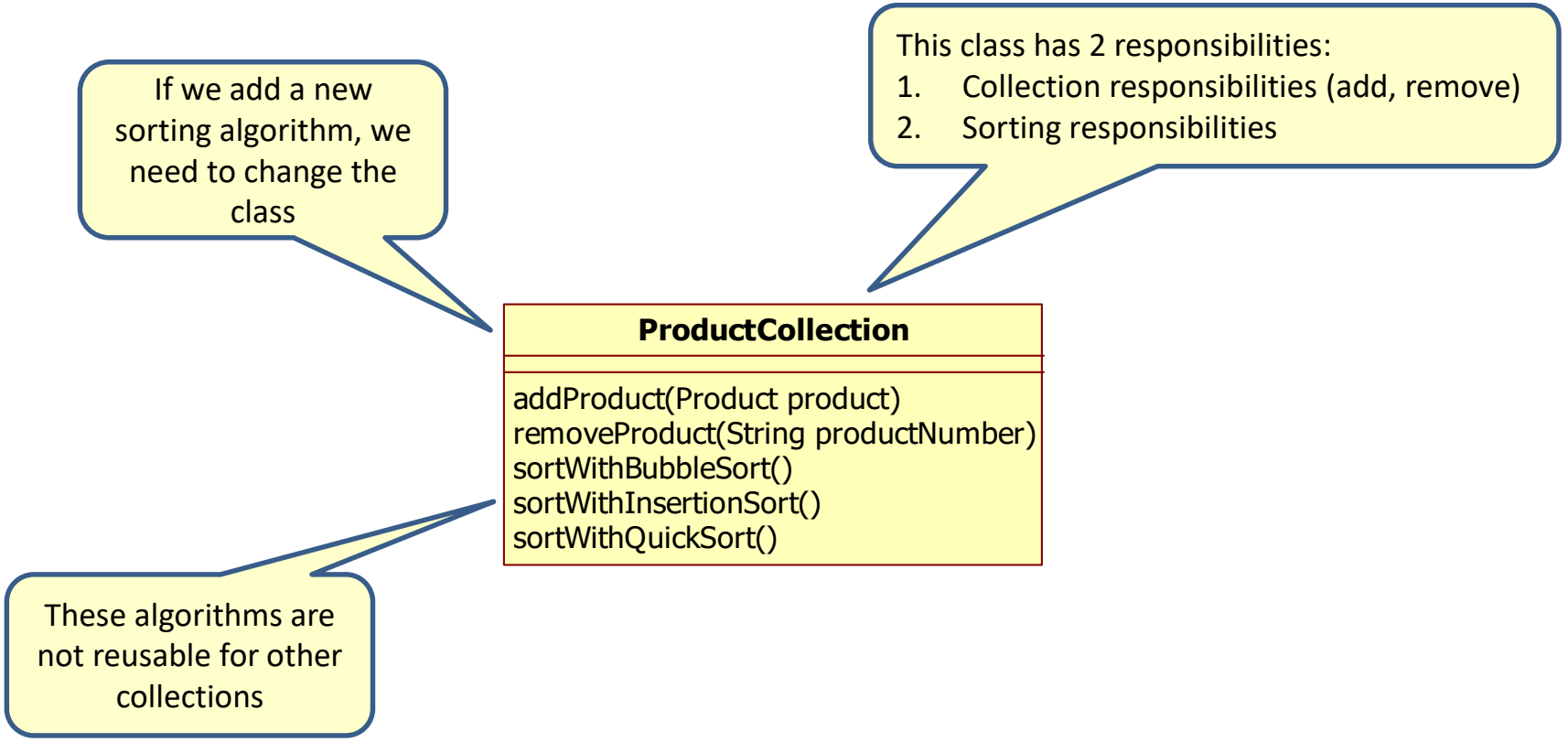
L8: Proxy, Adapter, Mediator
L9: Factory, Builder, Decorator, Singleton
L10: Framework design
L11: Framework implementation
L12: Framework example: Spring framework
L13: Framework example: Spring framework

Final

Strategy pattern

- The strategy pattern extracts algorithms (strategies) from a certain class (context class) and makes a different class for every single algorithm. This gives the following advantages
  - We can easily add new algorithms without changing the context class
  - The strategies are better reusable

Sorting a collection



ProductCollection

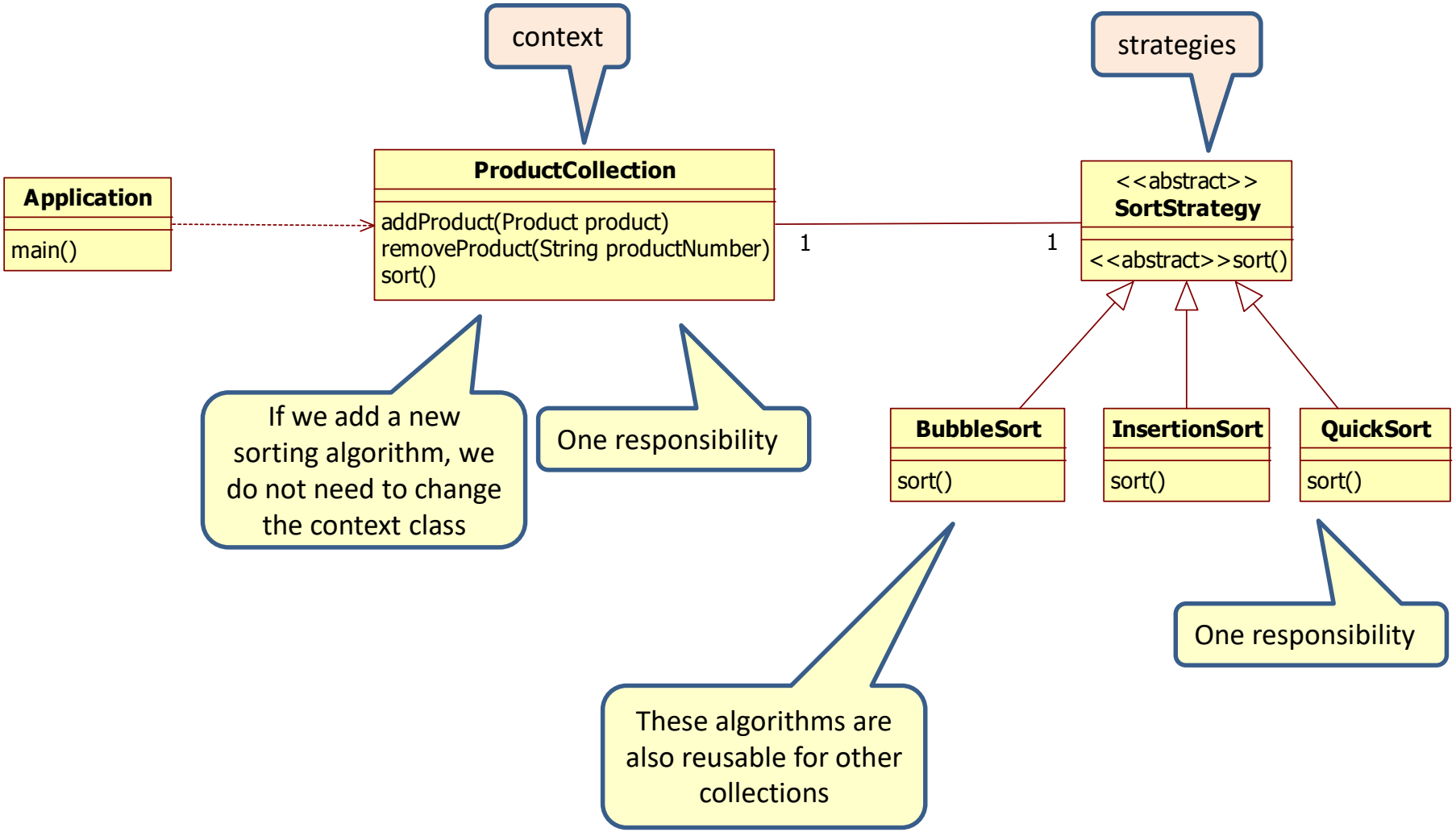
```
public class ProductCollection {
  private List<Product> products = new ArrayList<Product>();

  public void addproduct(Product product) {
    products.add(product);
  }

  public boolean removeProduct(String productNumber) {
    Iterator<Product> iterator = products.iterator();
    while (iterator.hasNext()) {
      if (iterator.next().getProductNumber().contentEquals(productNumber)) {
        iterator.remove();
        return true;
      }
    }
    return false;
  }

  public void bubbleSort() {
    System.out.println("peform bubblesort");
  }
  public void insertionSort() {
    System.out.println("peform insertionsort");
  }
  public void quickSort() {
    System.out.println("peform quicksort");
  }
}
```

```
public class Product {
  private String productNumber;
  private String name;
  ...
}
```

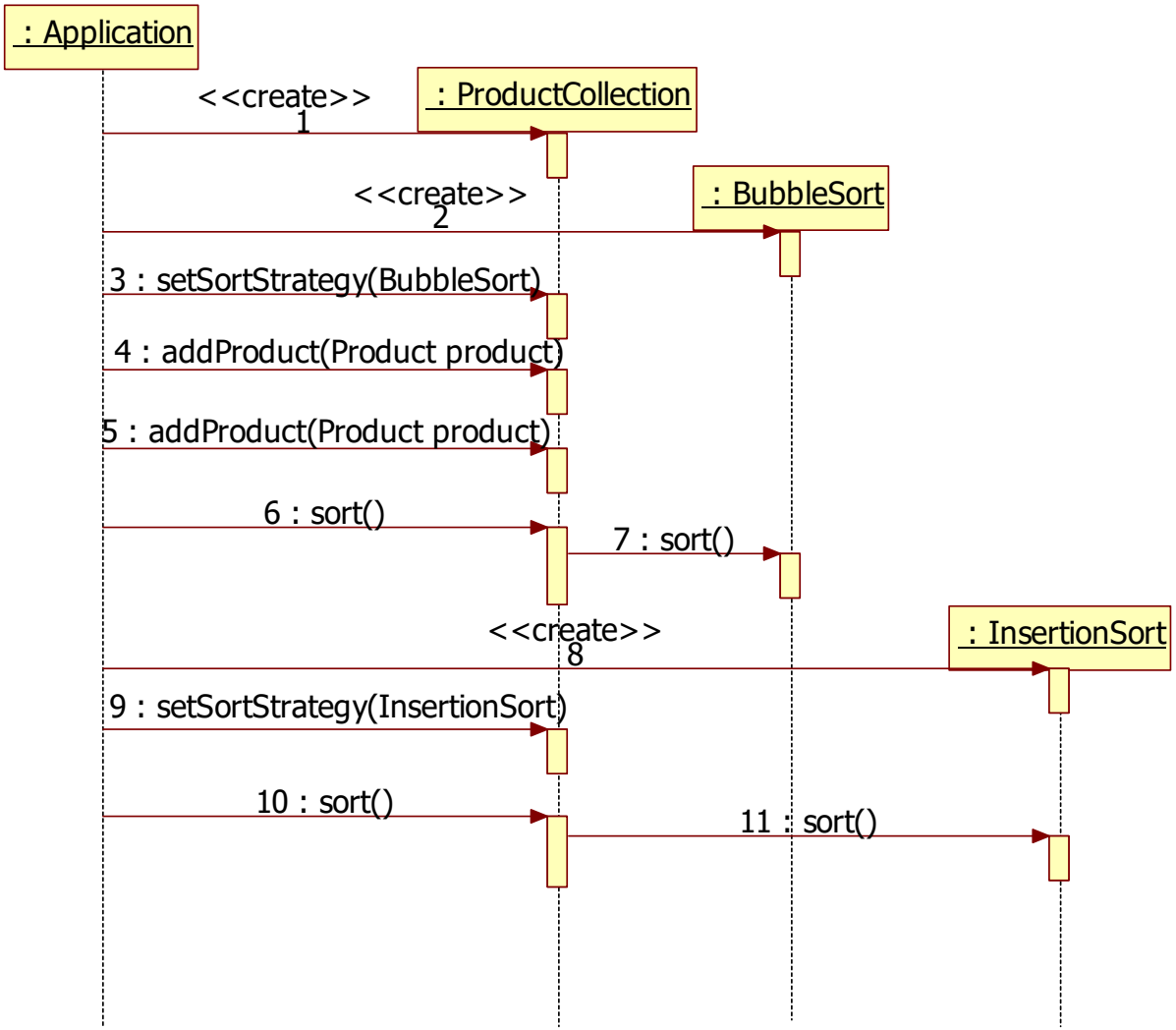
Application

```
public class Application {

  public static void main(String[] args) {
    ProductCollection productCollection = new ProductCollection();
    productCollection.addproduct(new Product("A23", "Iphone 10"));
    productCollection.addproduct(new Product("A28", "Iphone 11"));
    productCollection.bubbleSort();
    productCollection.insertionSort();
  }
}
```

Apply the strategy pattern



Apply the strategy pattern



The strategies

```
public abstract class SortStrategy {
  private ProductCollection productCollection;

  public SortStrategy(ProductCollection productCollection) {
    this.productCollection = productCollection;
  }

  abstract void sort();
}
```

```
public class BubbleSort extends SortStrategy{
  public BubbleSort(ProductCollection productCollection) {
    super(productCollection);
  }

  @Override
  void sort() {
    System.out.println("peform bubblesort");
  }
}
```

The strategies

```
public class InsertionSort extends SortStrategy{
  public InsertionSort(ProductCollection productCollection) {
    super(productCollection);
  }

  @Override
  void sort() {
    System.out.println("peform insertionsort");
  }
}
```

```
public class QuickSort extends SortStrategy{
  public QuickSort(ProductCollection productCollection) {
    super(productCollection);
  }

  @Override
  void sort() {
    System.out.println("peform quicksort");
  }
}
```

ProductCollection

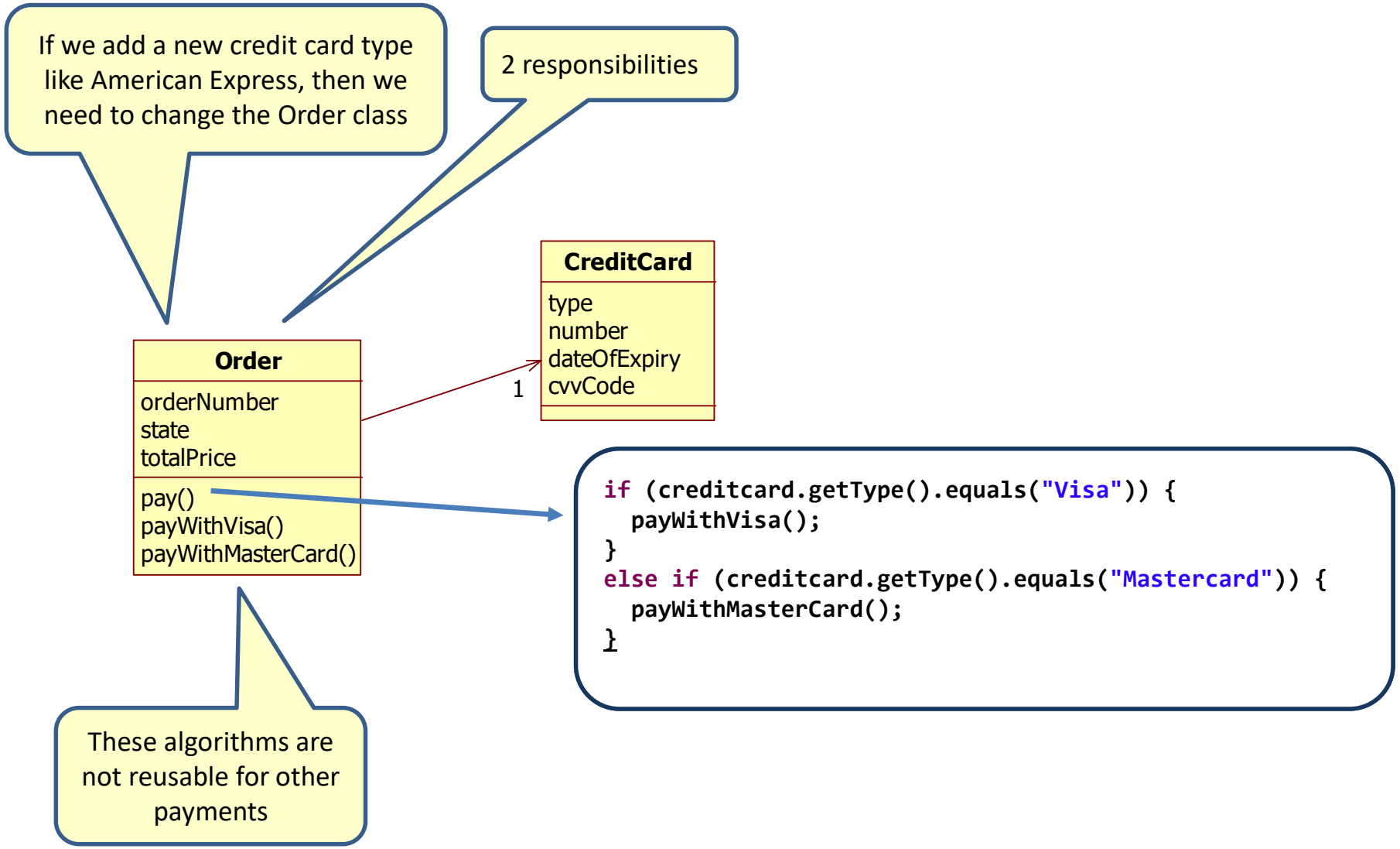
```
public class ProductCollection {
  private List<Product> products = new ArrayList<Product>();
  private SortStrategy sortStrategy;

  public void addproduct(Product product) {
    products.add(product);
  }

  public boolean removeProduct(String productNumber) {
    Iterator<Product> iterator = products.iterator();
    while (iterator.hasNext()) {
      if (iterator.next().getProductNumber().contentEquals(productNumber)) {
        iterator.remove();
        return true;
      }
    }
    return false;
  }

  public void sort() {
    sortStrategy.sort();
  }

  public void setSortStrategy(SortStrategy sortStrategy) {
    this.sortStrategy=sortStrategy;
  }
}
```

```
public class Product {
  private String productNumber;
  private String name;
  ...
}
```

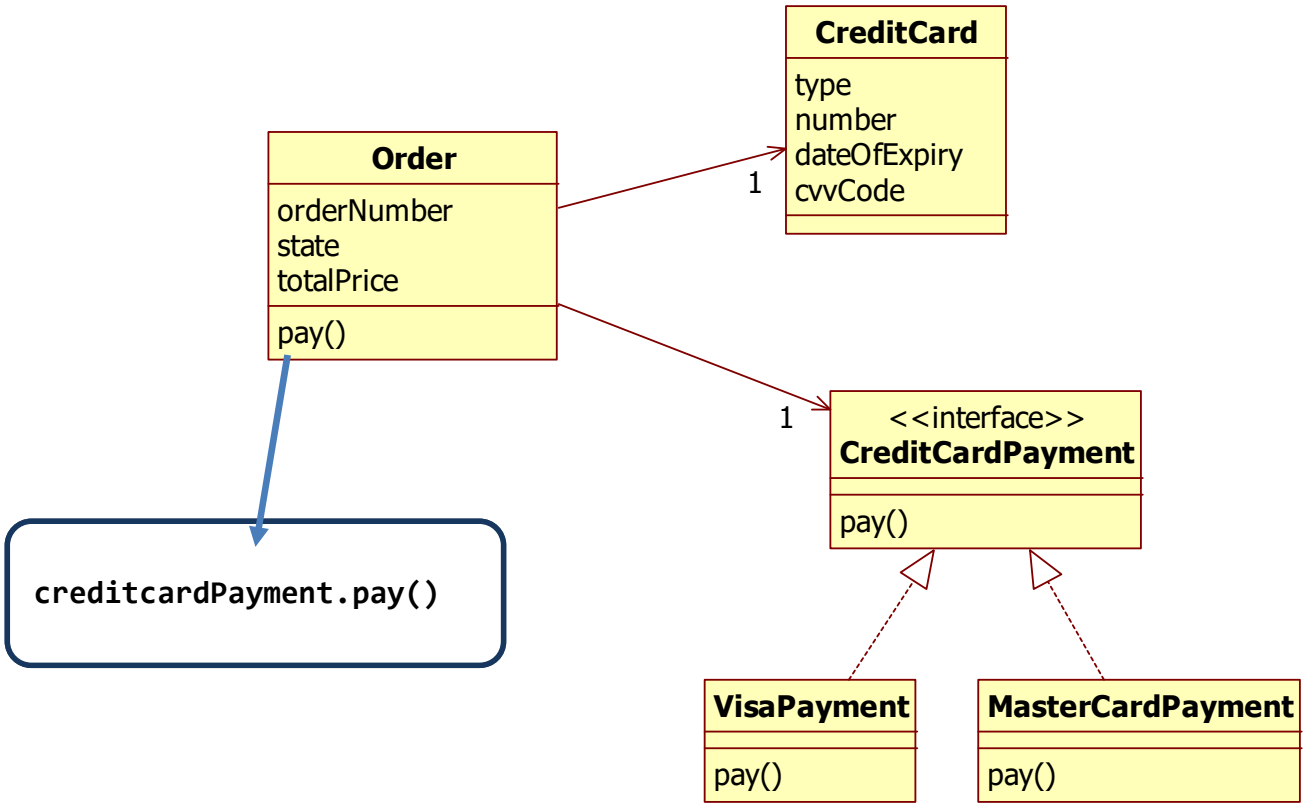
Application

```
public class Application {

  public static void main(String[] args) {
    ProductCollection productCollection = new ProductCollection();
    SortStrategy sortStrategy = new BubbleSort(productCollection);
    productCollection.setSortStrategy(sortStrategy);

    productCollection.addproduct(new Product("A23", "Iphone 10"));
    productCollection.addproduct(new Product("A28", "Iphone 11"));
    productCollection.sort();

    SortStrategy newsortStrategy = new InsertionSort(productCollection);
    productCollection.setSortStrategy(newsortStrategy);
    productCollection.sort();
  }
}
```

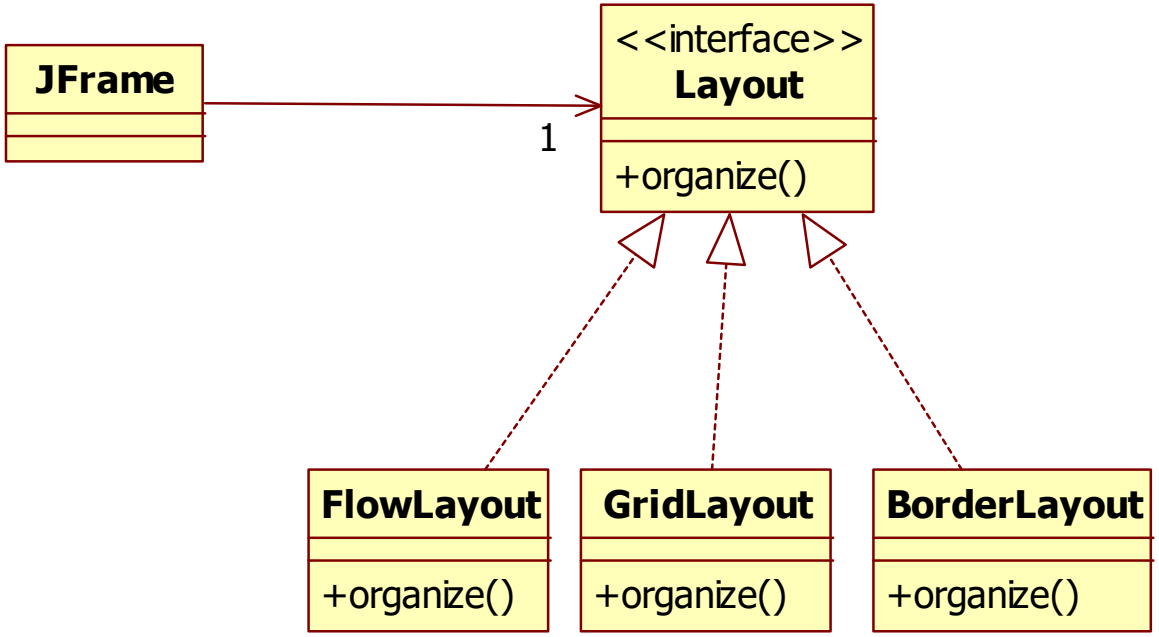
Order without strategy



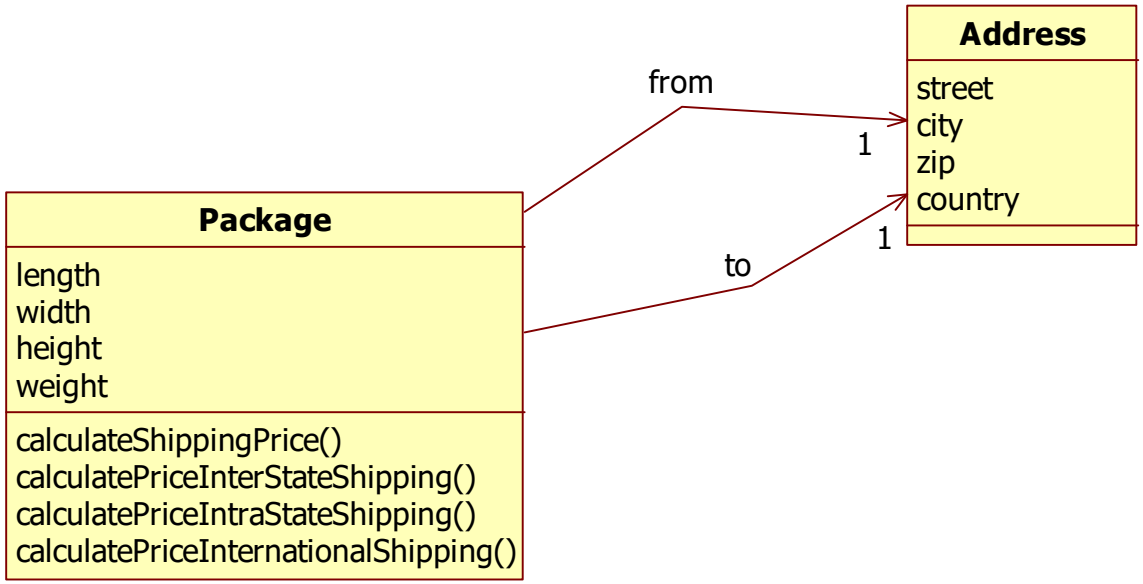
Order with strategy



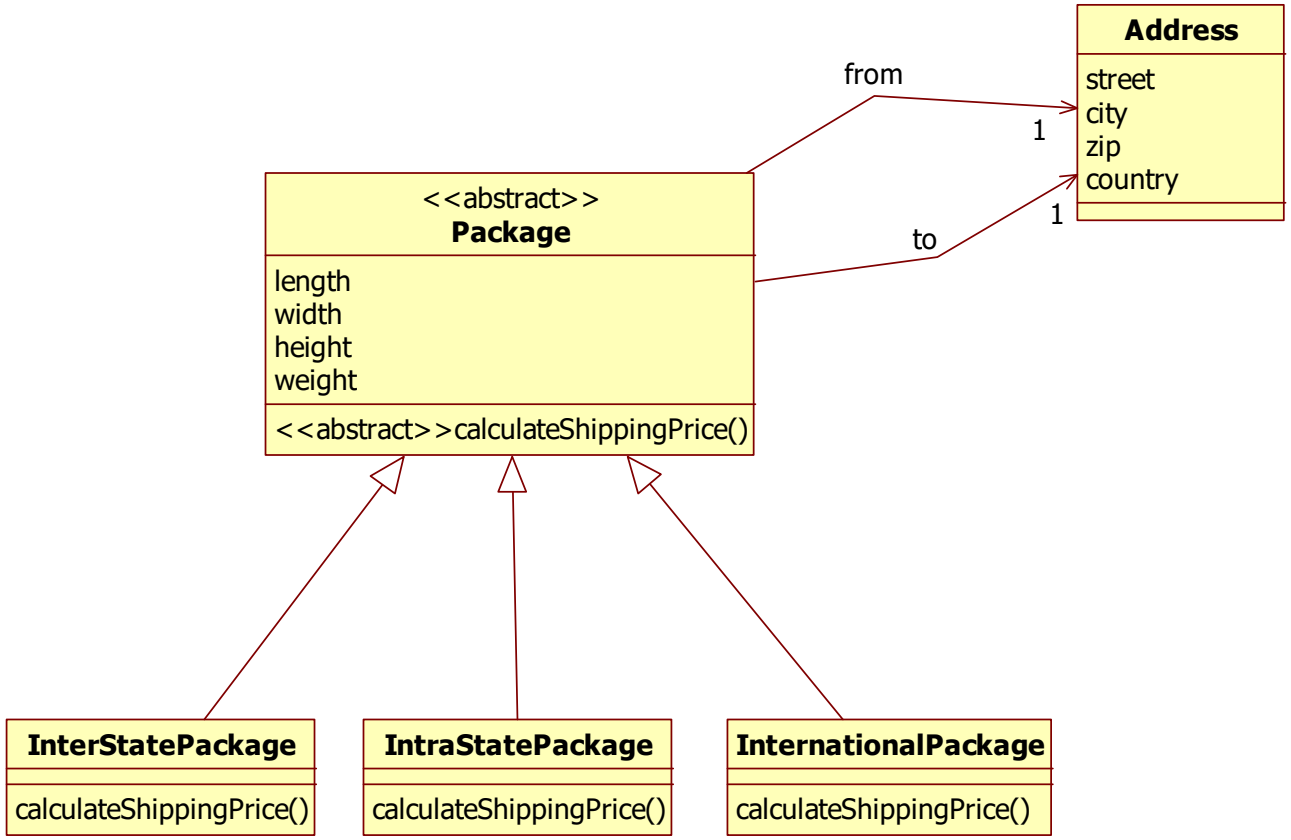
Strategy pattern example



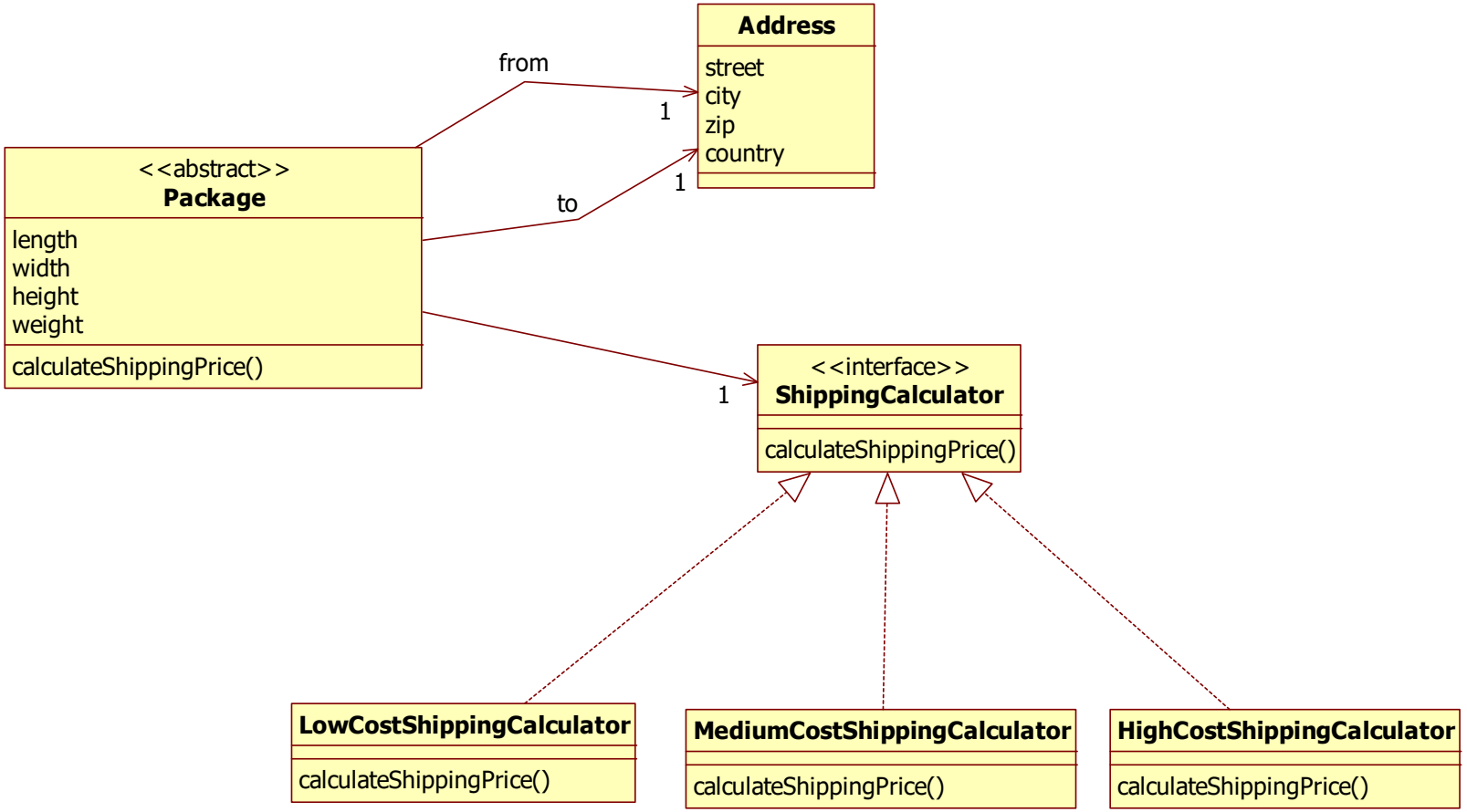
Calculate shipping price



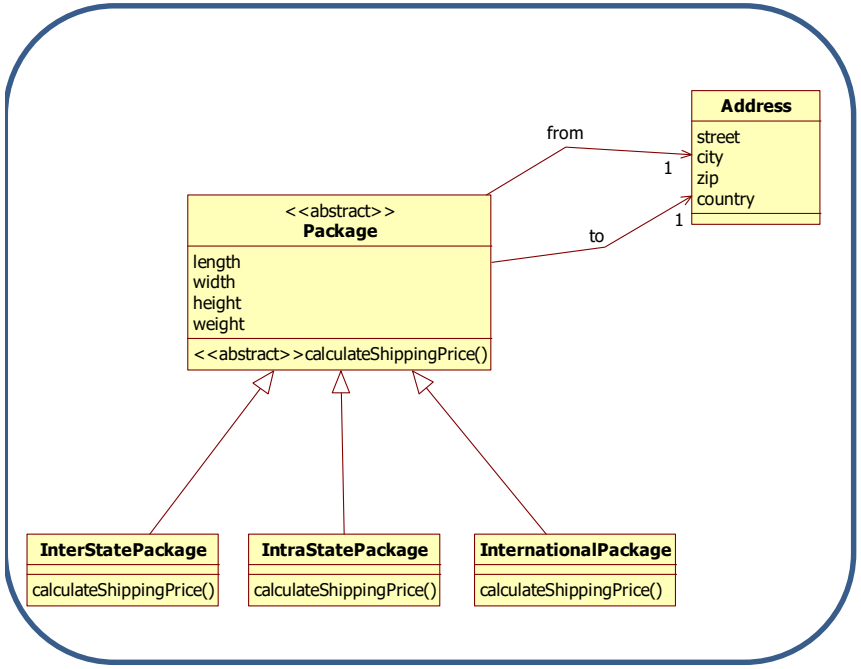
Solution 1: inheritance

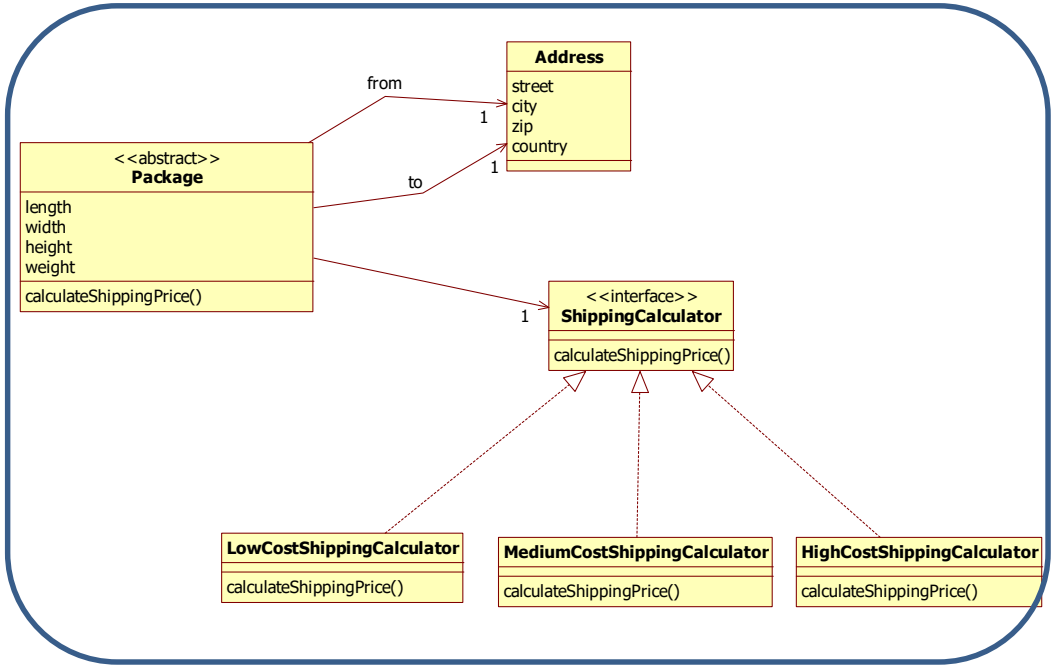


Solution 2: strategy

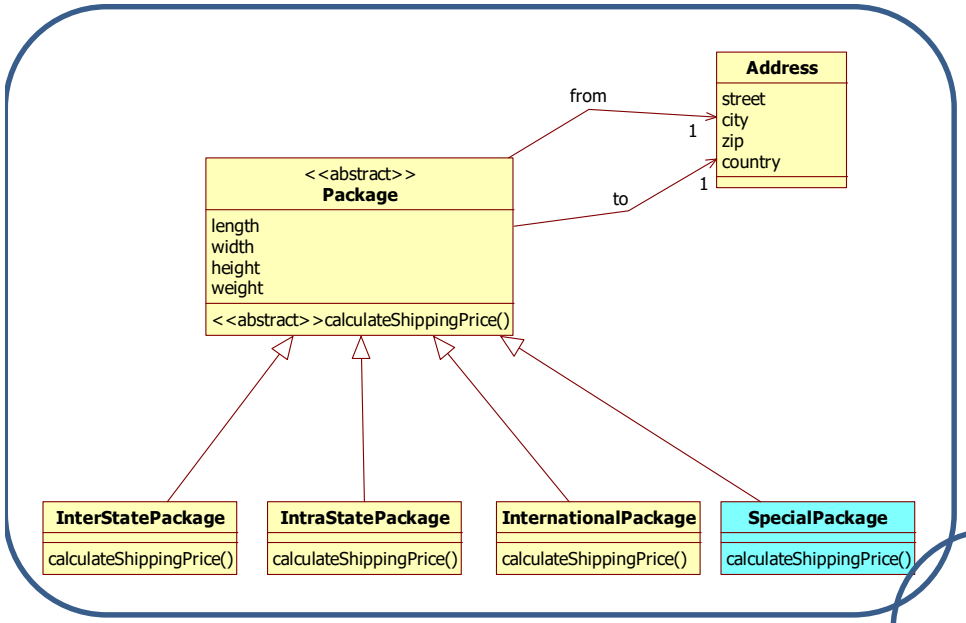


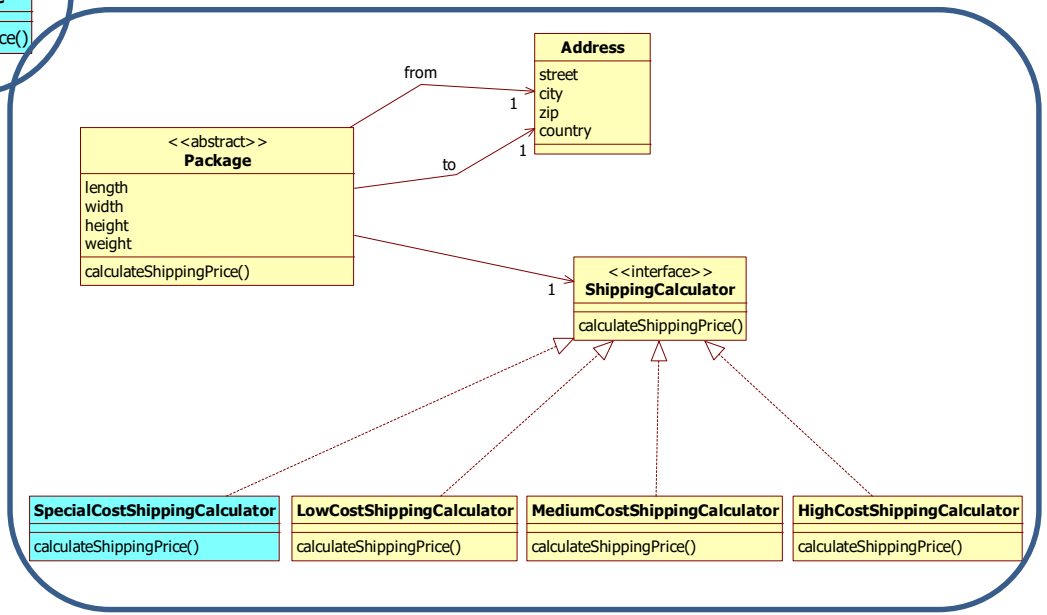
What are the differences?



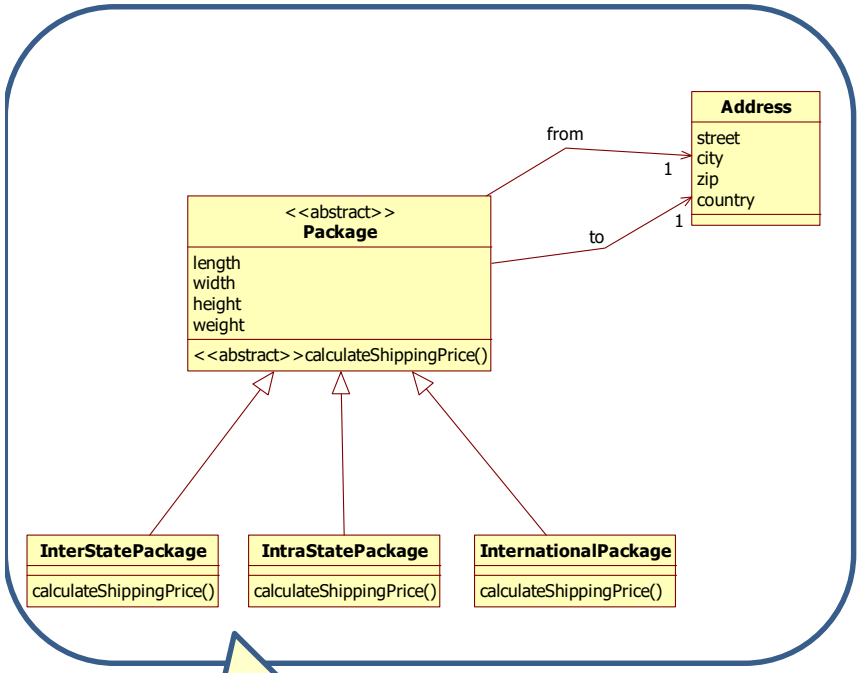


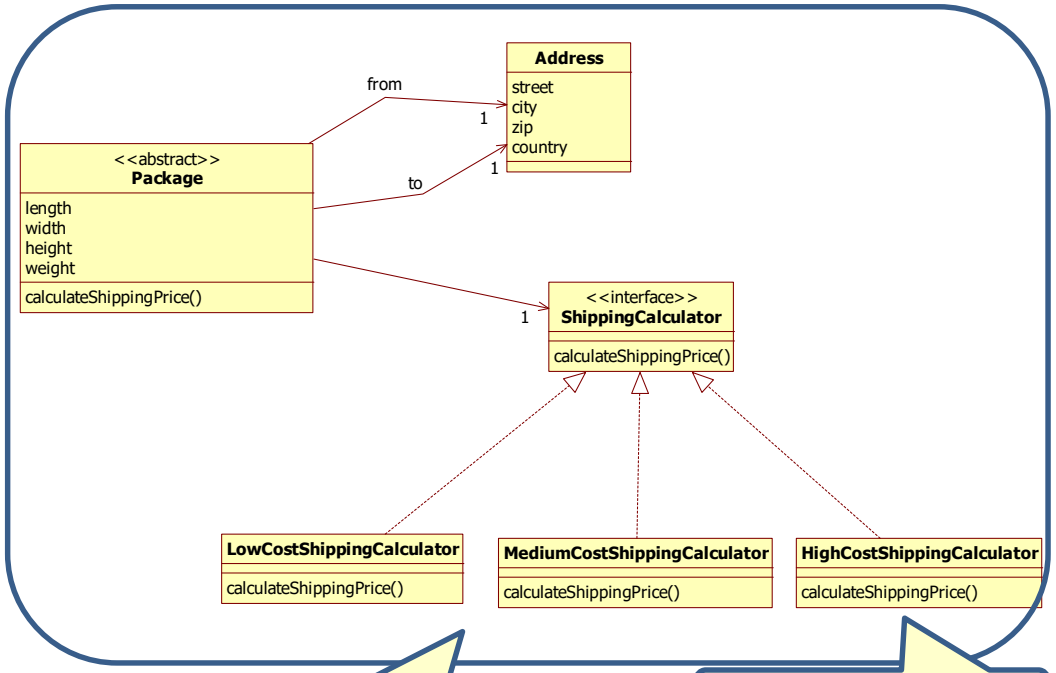
Add a new kind of shipping





The real difference



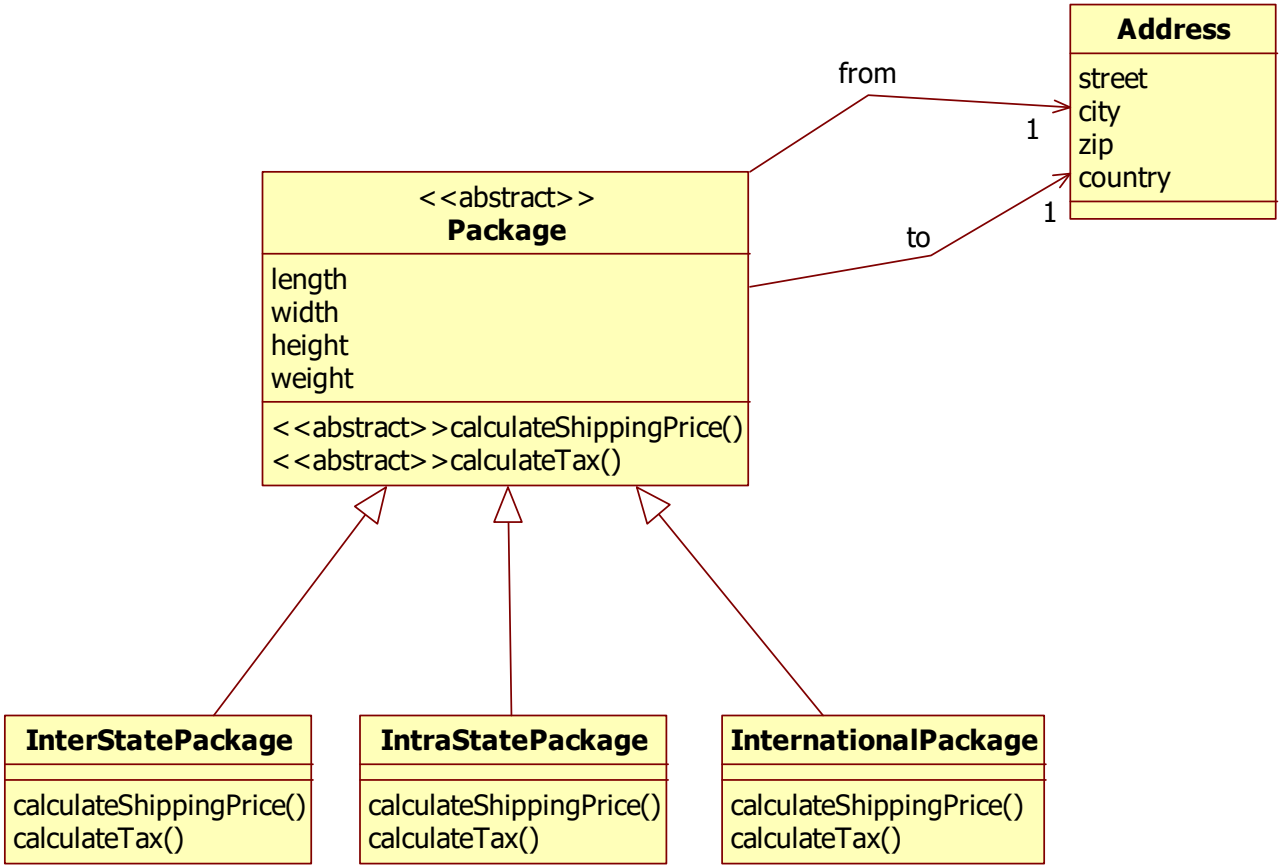


If we use the same algorithm for inter and intra state packages, we have to copy & paste the algorithm

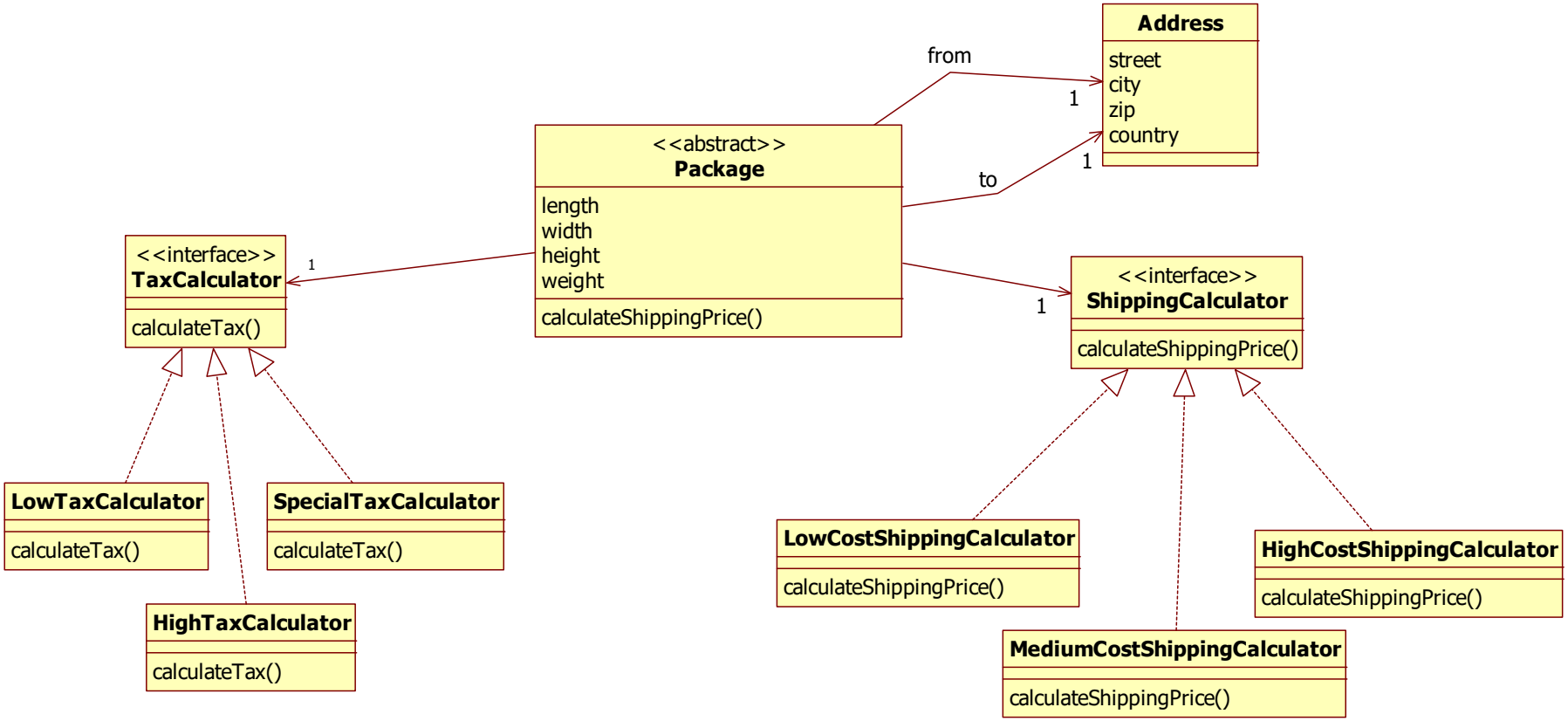
If we use the same algorithm for inter and intra state packages, we can reuse the same ShippingCalculator

reusable algorithms

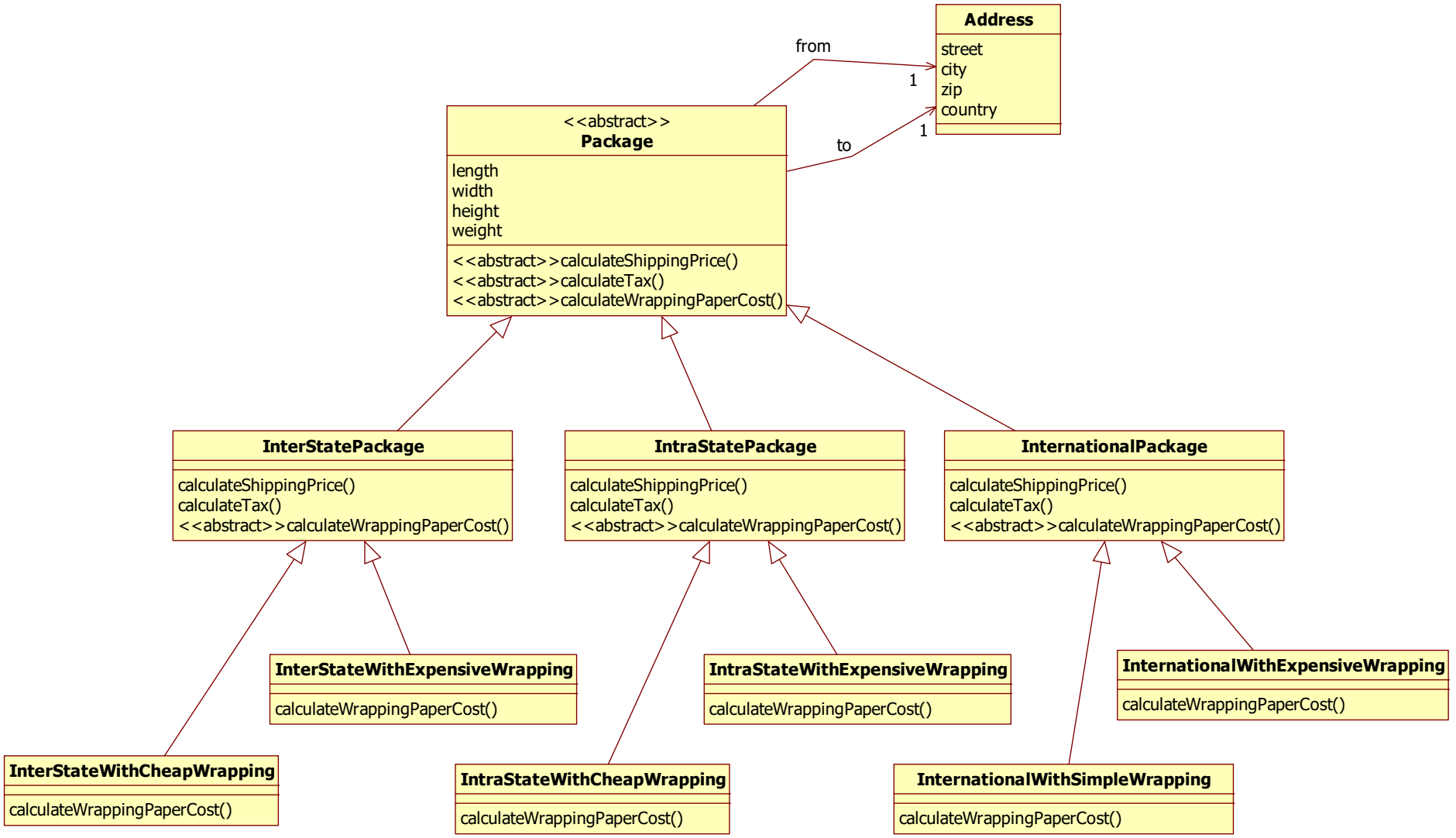
Different ways to calculate tax with inheritance



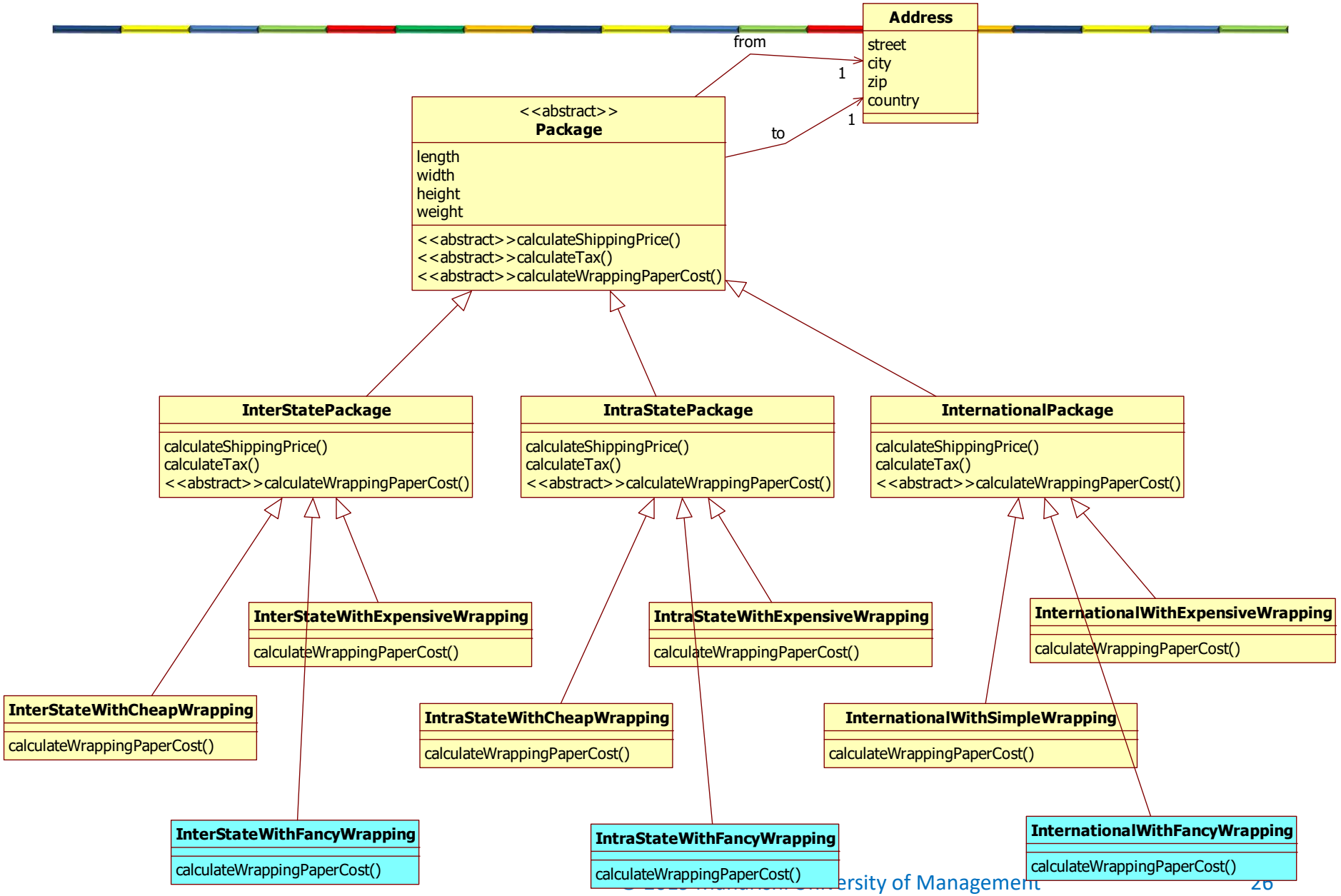
Different ways to calculate tax with strategy



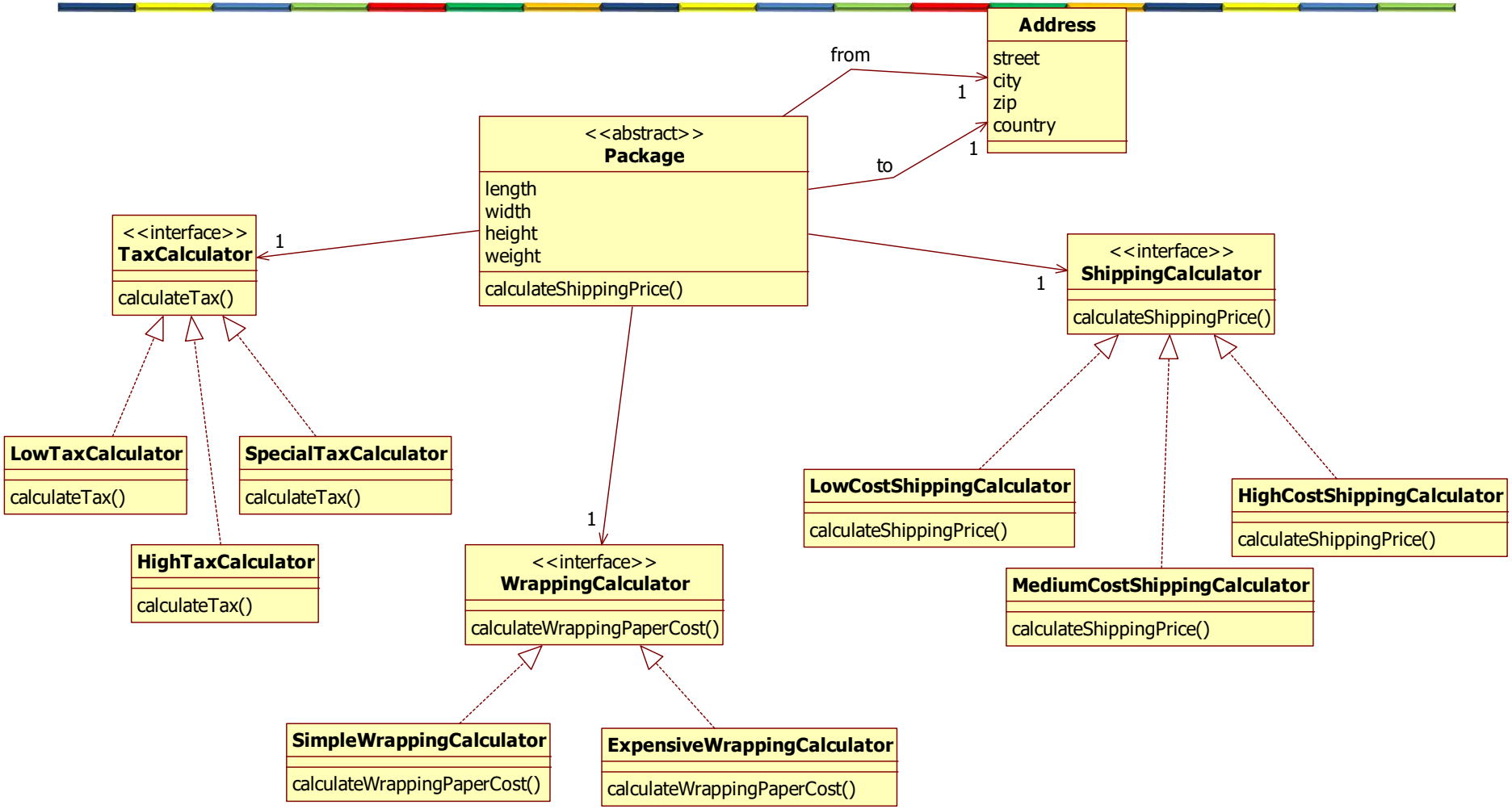
Giftwrap possibilities with inheritance



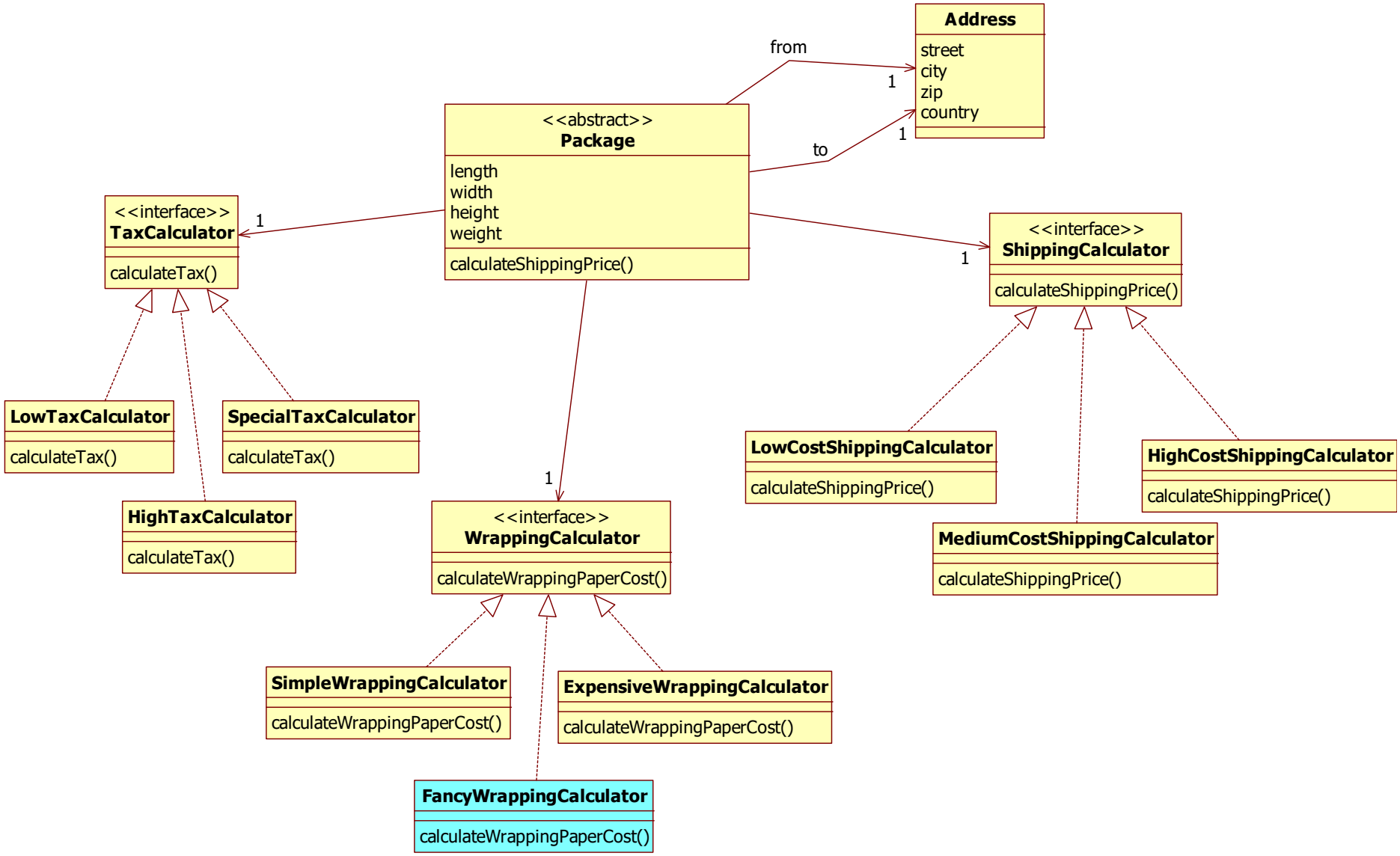
Let’s add Fancy gift wrapping



Giftwrap possibilities with strategy



Let’s add Fancy gift wrapping



Example of strategy pattern

```
public class Application {
  public static void main(String[] args) {
    List<String> fruits = Arrays.asList(
        "watermelon",
        "apple",
        "pear");

    Collections.sort(fruits, new AlphabeticalComparator());
    // will print [apple, pear, watermelon]
    System.out.println(fruits);

    Collections.sort(fruits, new ByLengthComparator());
    // will print [pear, apple, watermelon]
    System.out.println(fruits);
  }
}
```

```
public class AlphabeticalComparator implements Comparator<String> {
  @Override
  public int compare(String o1, String o2) {
    return o1.compareTo(o2);
  }
}
```

```
public class ByLengthComparator implements Comparator<String> {
  @Override
  public int compare(String o1, String o2) {
    return Integer.compare(o1.length(), o2.length());
  }
}
```

Strategy pattern

- What problem does it solve?
  - The Strategy pattern provides a way to define a family of algorithms, encapsulate each one as an object, and make them interchangeable.
  - Whenever you want to choose the algorithm to use at runtime.

Main point

- With the strategy pattern, different algorithms are extracted from its context and encapsulated as strategy classes
- The unified field is the field of all possibilities.