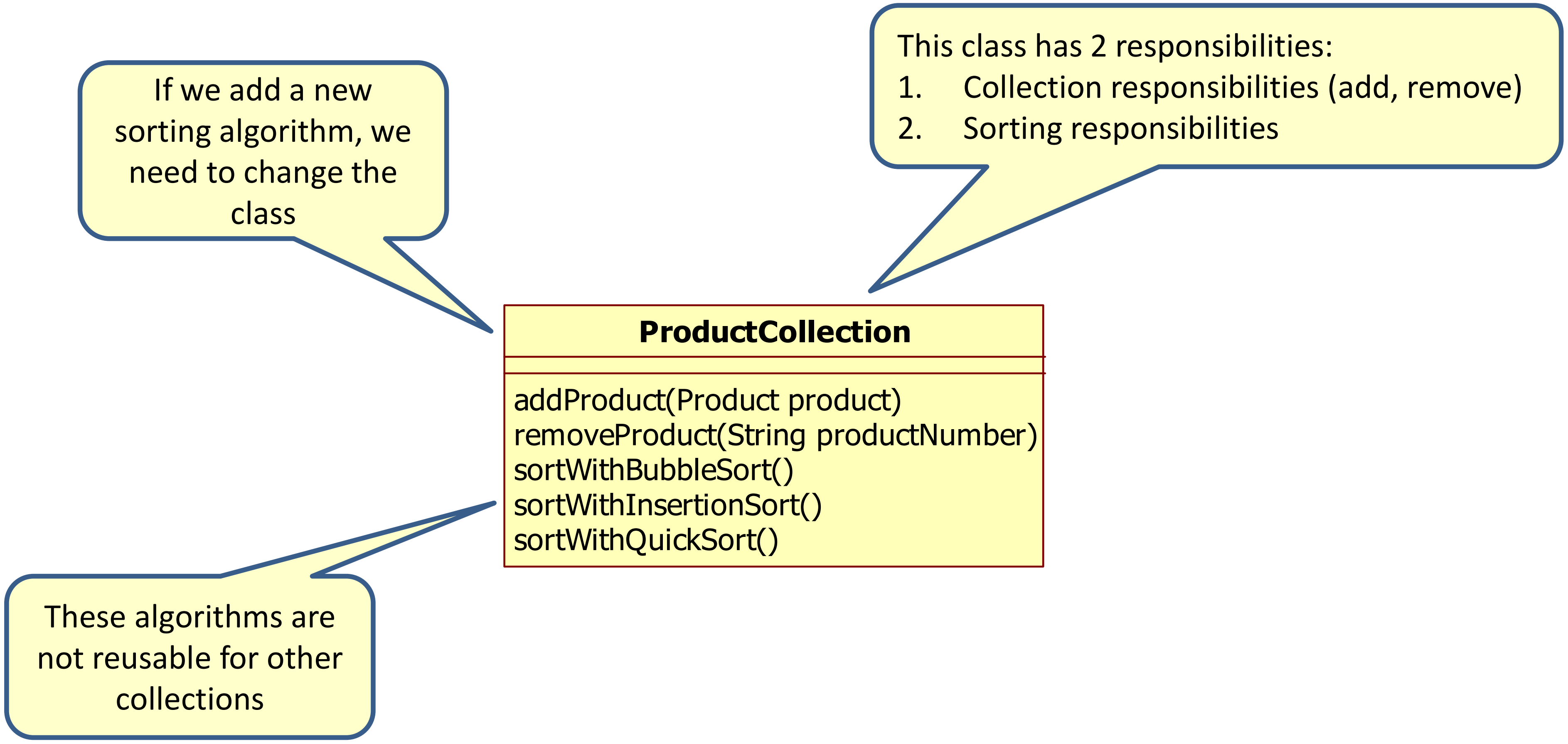
LESSON 3
STRATEGY PATTERN
TEMPLATE METHOD PATTERN

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


If we add a new sorting algorithm, we need to change the class
This class has 2 responsibilities:
1. Collection responsibilities (add, remove)
2. Sorting responsibilities
ProductCollection
addProduct(Product product)
removeProduct(String productNumber)
sortWithBubbleSort()
sortWithInsertionSort()
sortWithQuickSort()
These algorithms are not reusable for other collections

ProductCollection

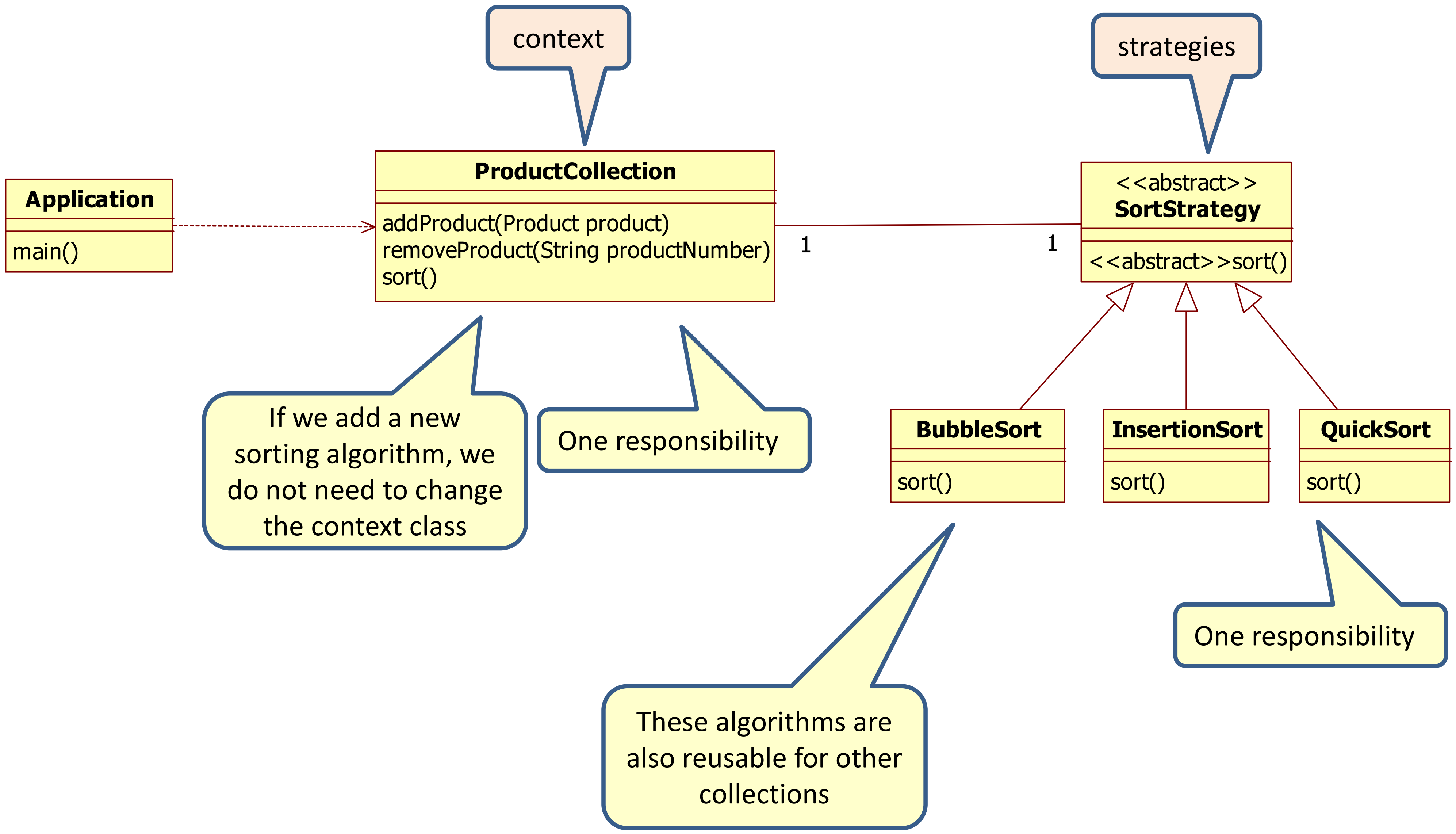
```java
public class ProductCollection {
  private List<Product> products = new ArrayList<Product>();

  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");
  }
}
```

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

Application

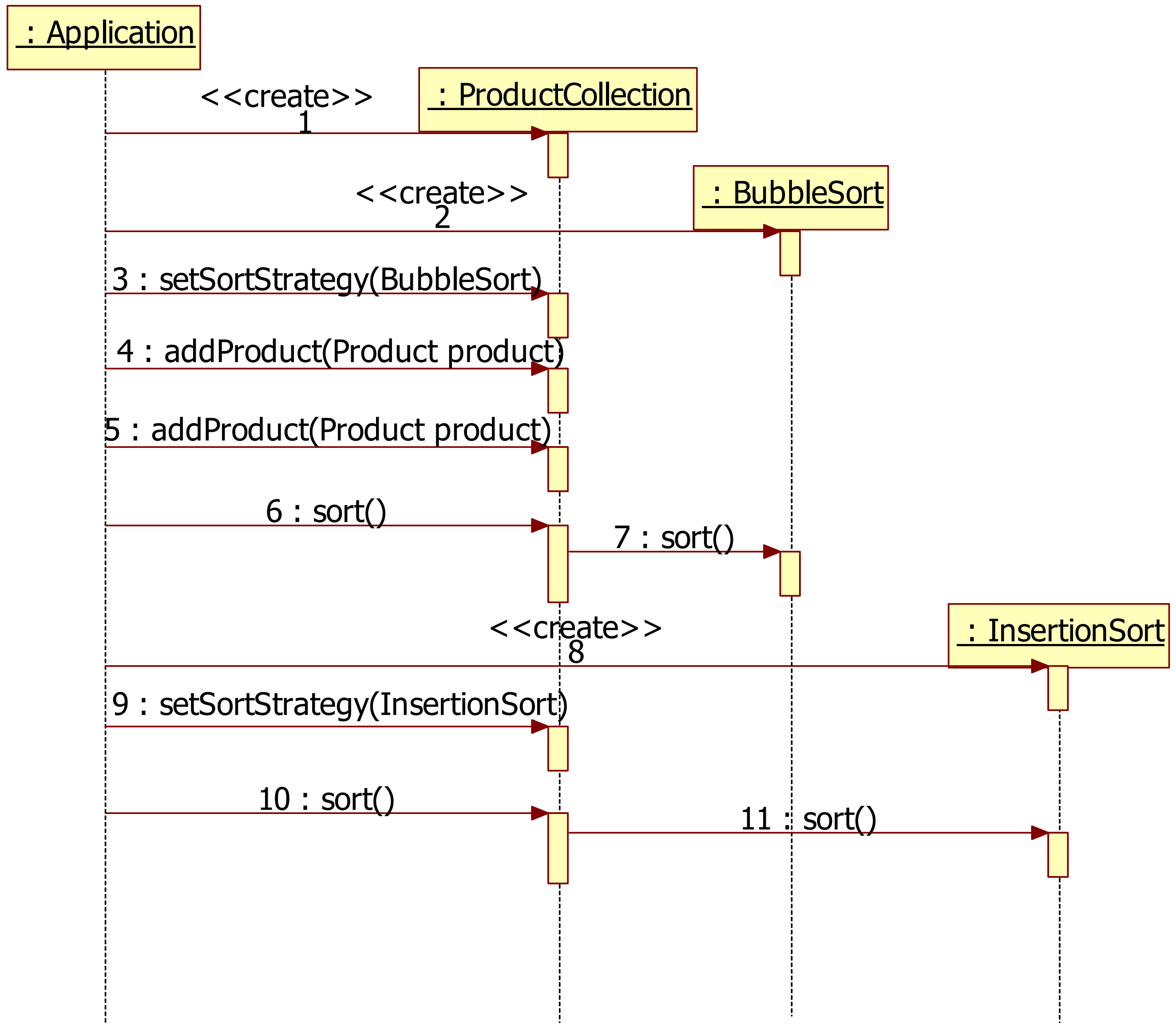
```
public class Application {

  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



context
strategies
Application
main()
ProductCollection
addProduct(Product product)
removeProduct(String productNumber)
sort()
1
1
<<abstract>>
SortStrategy
<<abstract>>sort()
BubbleSort
sort()
InsertionSort
sort()
QuickSort
sort()
If we add a new sorting algorithm, we do not need to change the context class
One responsibility
These algorithms are also reusable for other collections
One responsibility

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

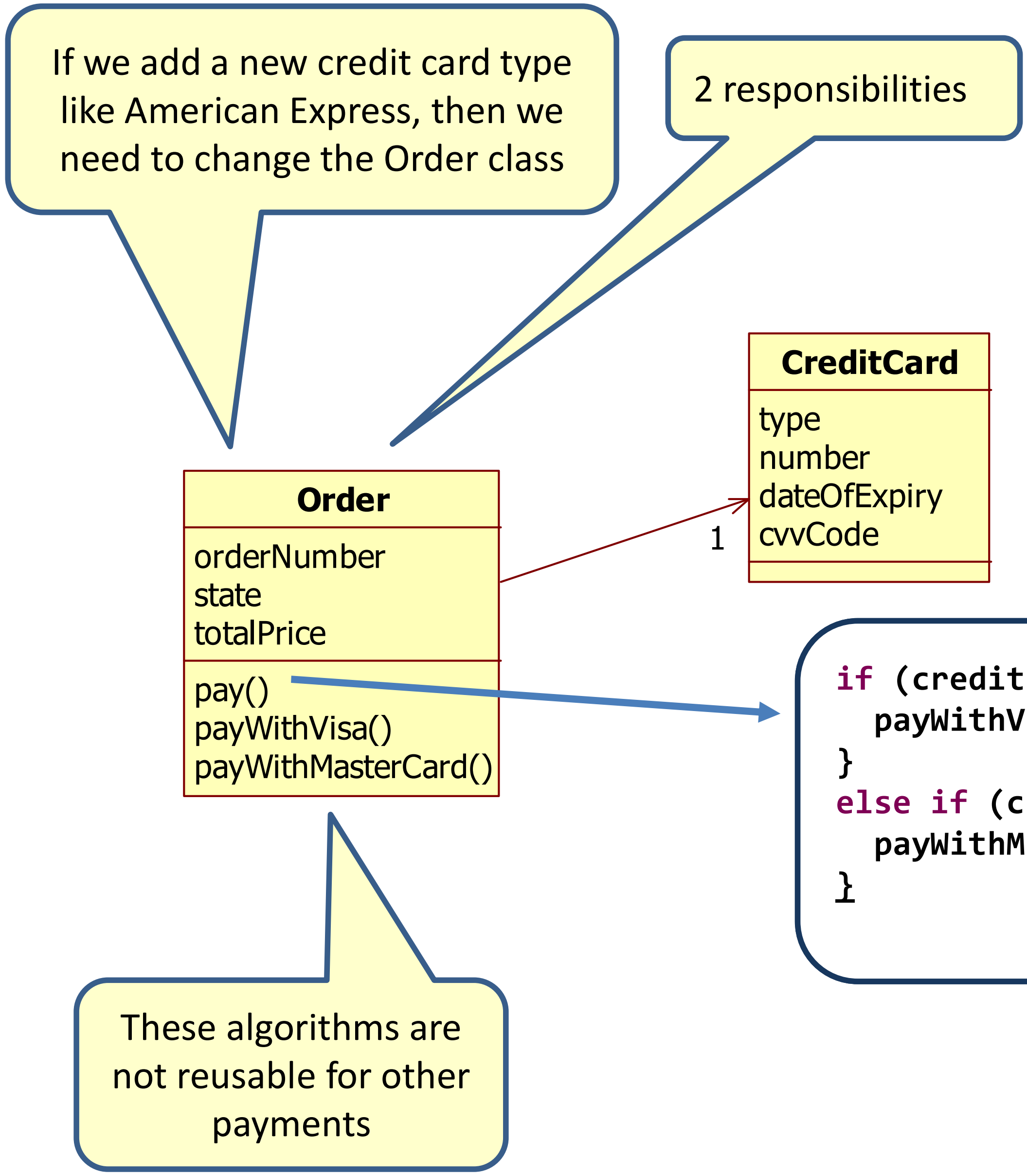
```java
public class ProductCollection {
  private List<Product> products = new ArrayList<Product>();
  private SortStrategy sortStrategy;

  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;
  }
}
```

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

Application

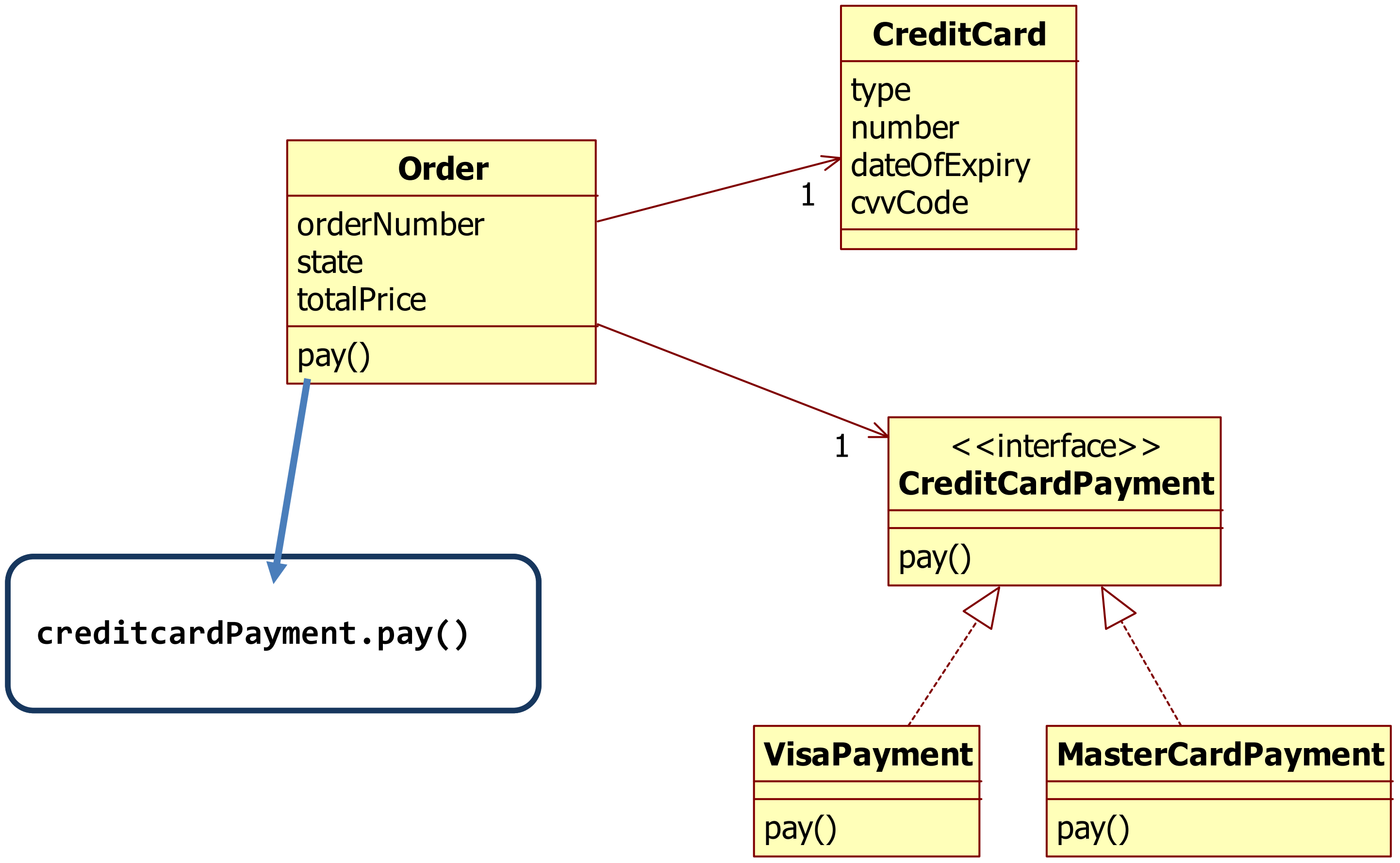
```
public class Application {

  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

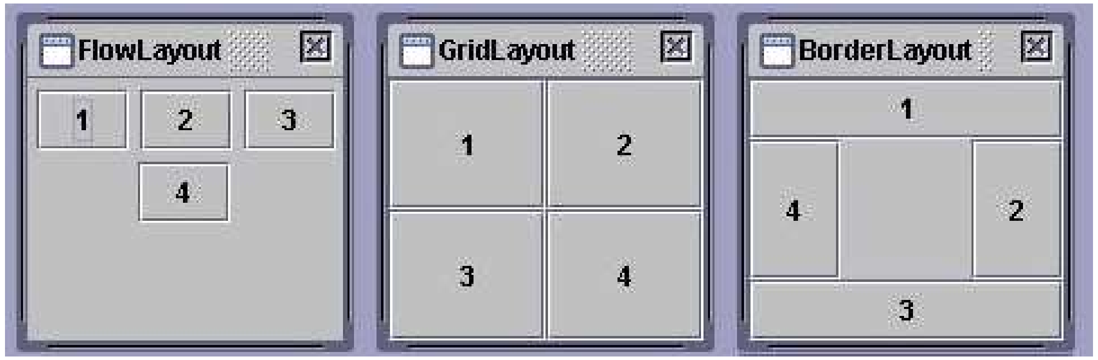

If we add a new credit card type like American Express, then we need to change the Order class
2 responsibilities
CreditCard
type
number
dateOfExpiry
cvvCode
1
Order
orderNumber
state
totalPrice
pay()
payWithVisa()
payWithMasterCard()
These algorithms are not reusable for other payments

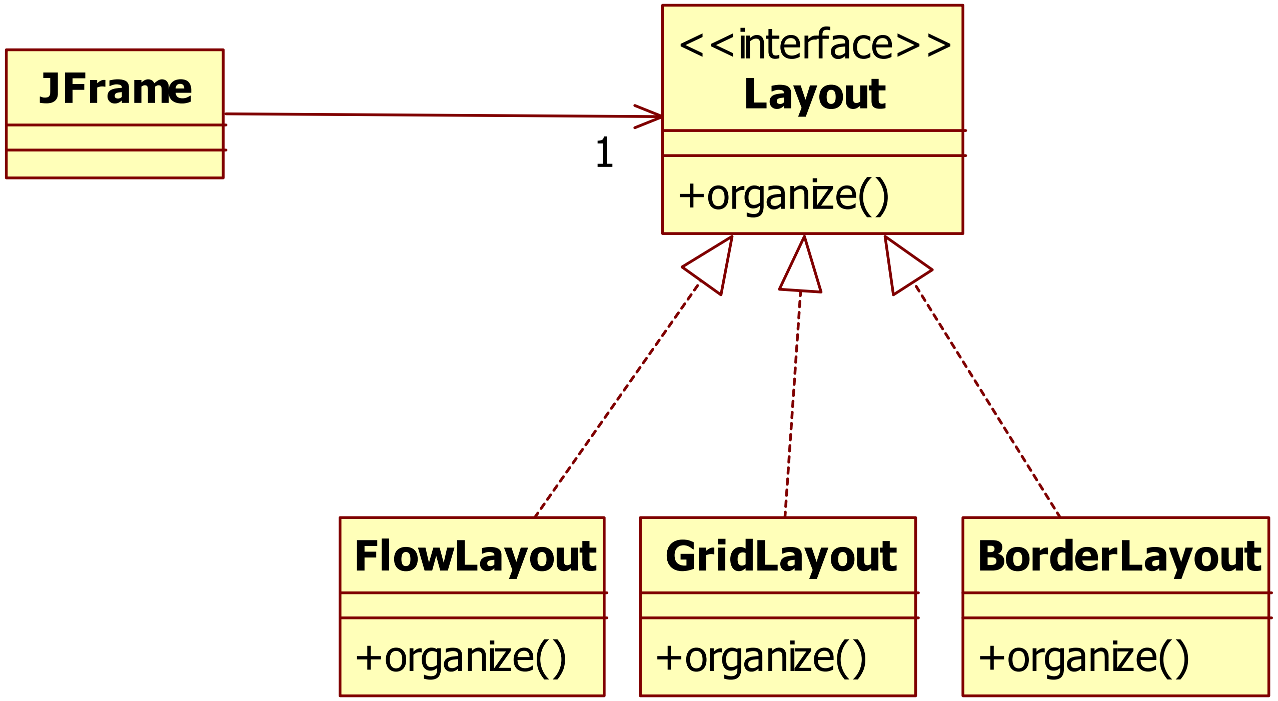
```
if (creditcard.getType().equals("Visa")) {
  payWithVisa();
}
else if (creditcard.getType().equals("Mastercard")) {
  payWithMasterCard();
}
```

Order with strategy

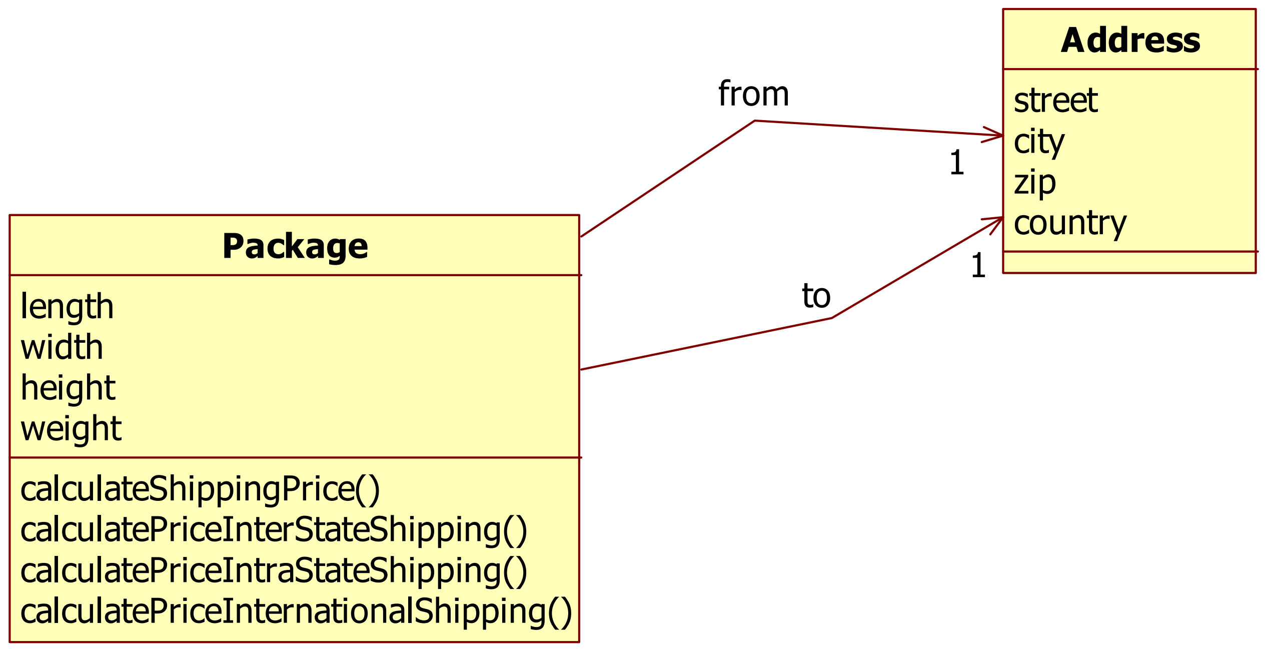


Strategy pattern example

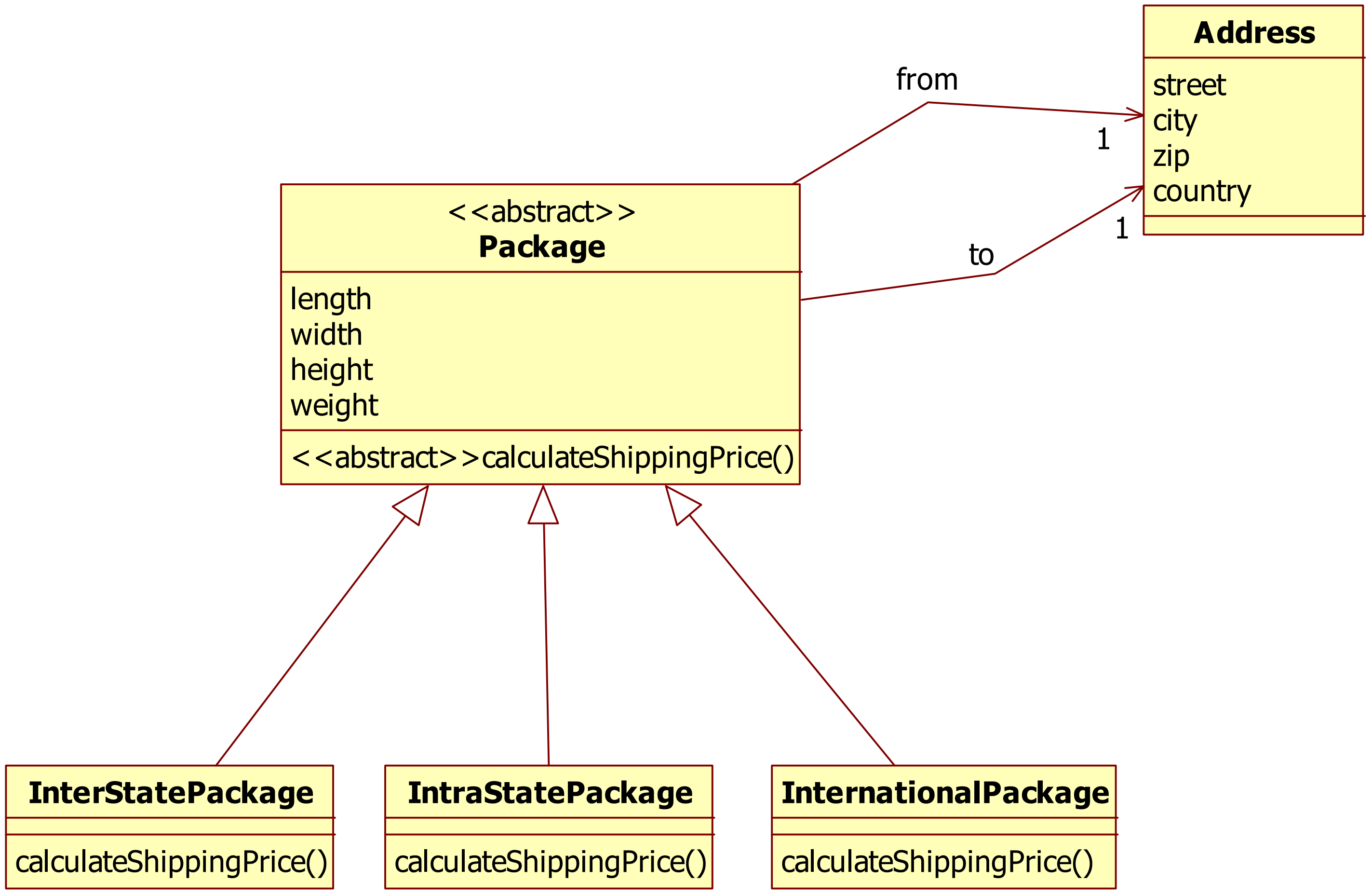




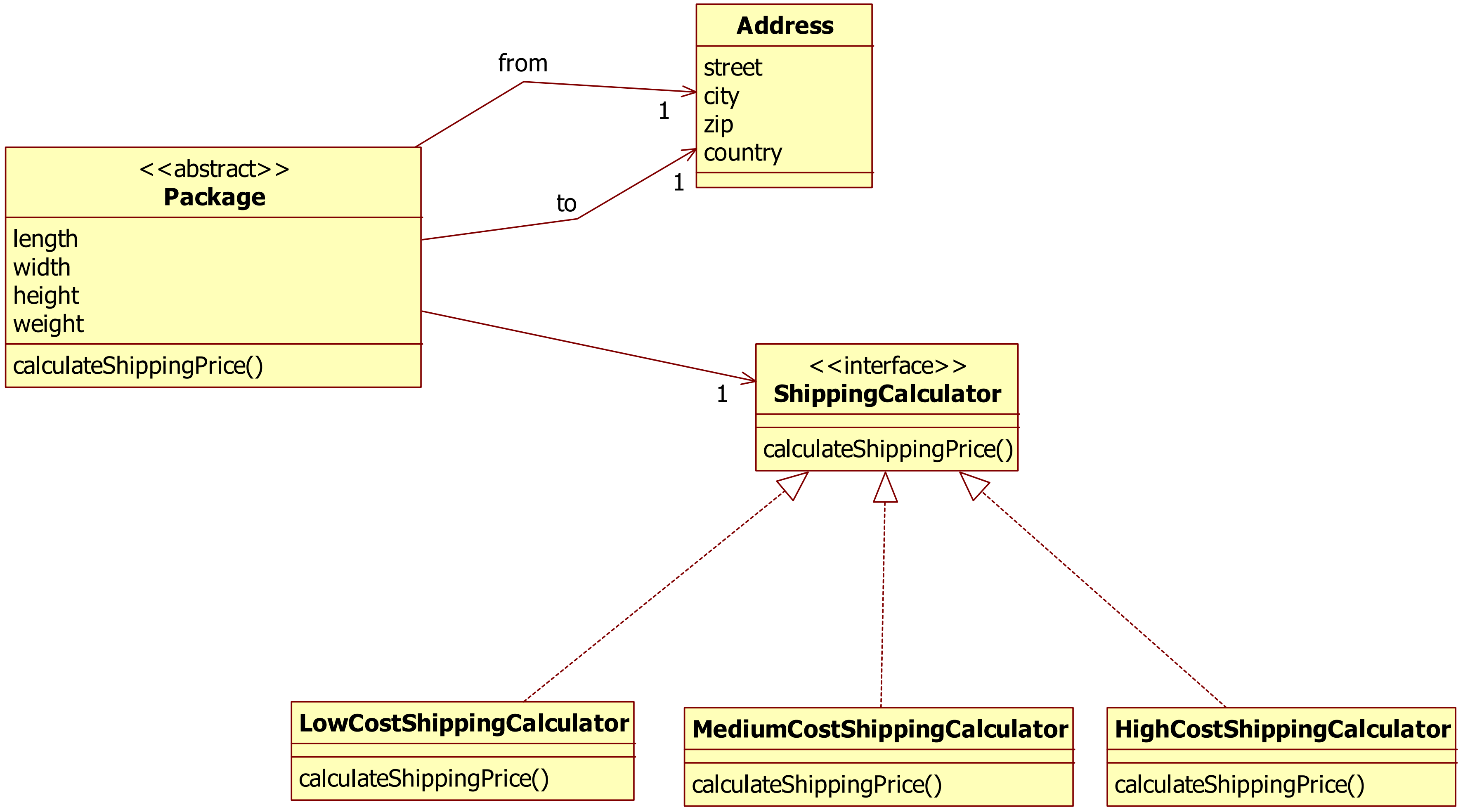
Calculate shipping price


Address
street
city
zip
country
from
1
to
1
Package
length
width
height
weight
calculateShippingPrice()
calculatePriceInterStateShipping()
calculatePriceIntraStateShipping()
calculatePriceInternationalShipping()

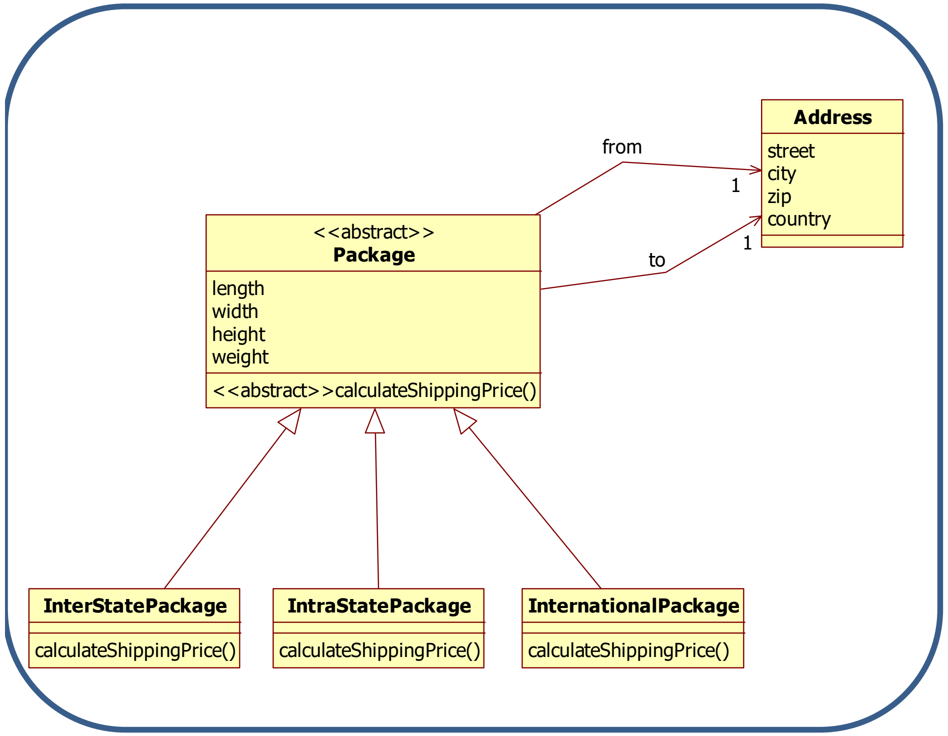
Solution 1: inheritance

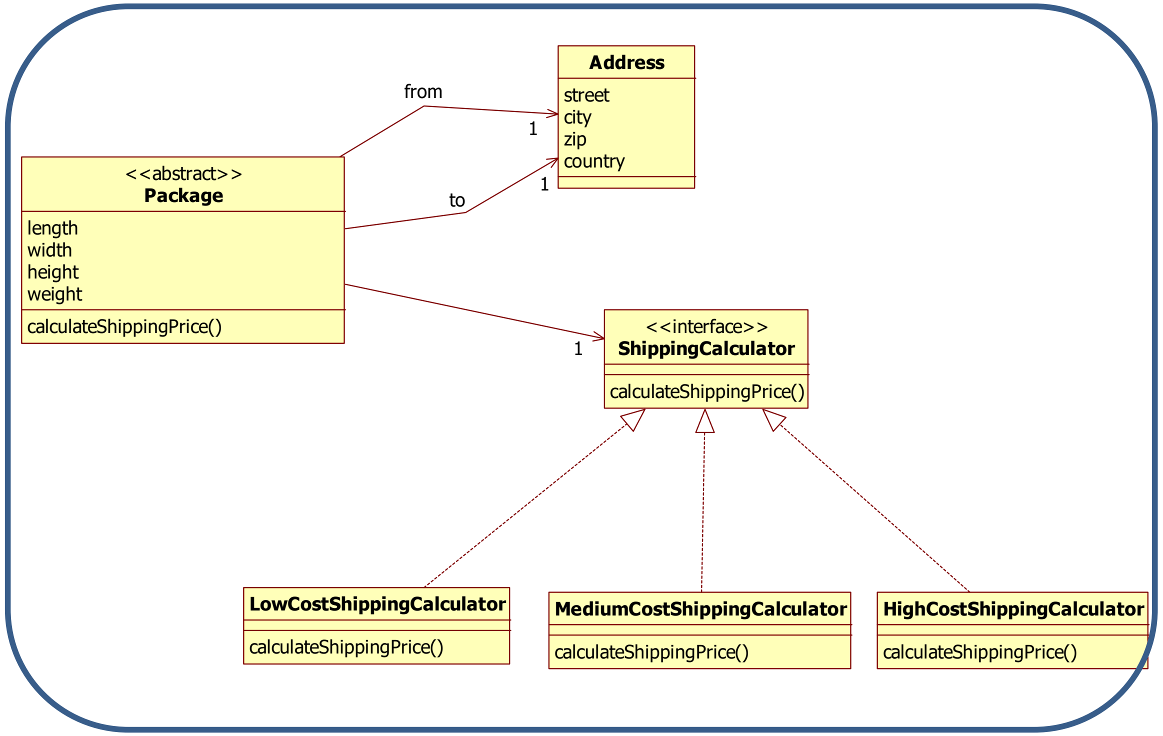


Solution 2: strategy

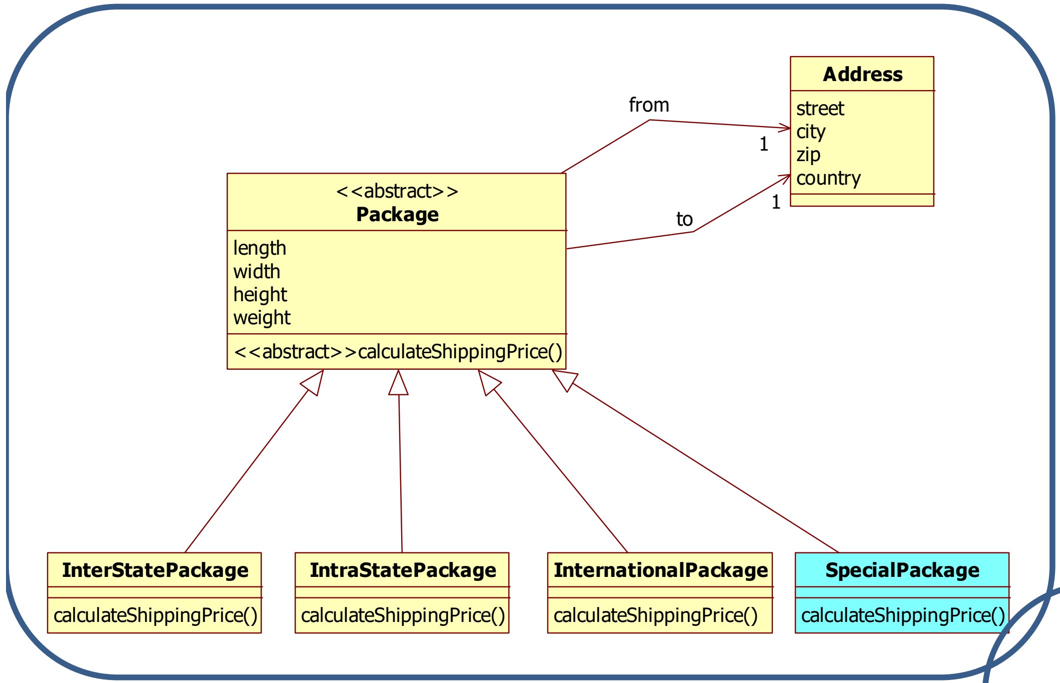


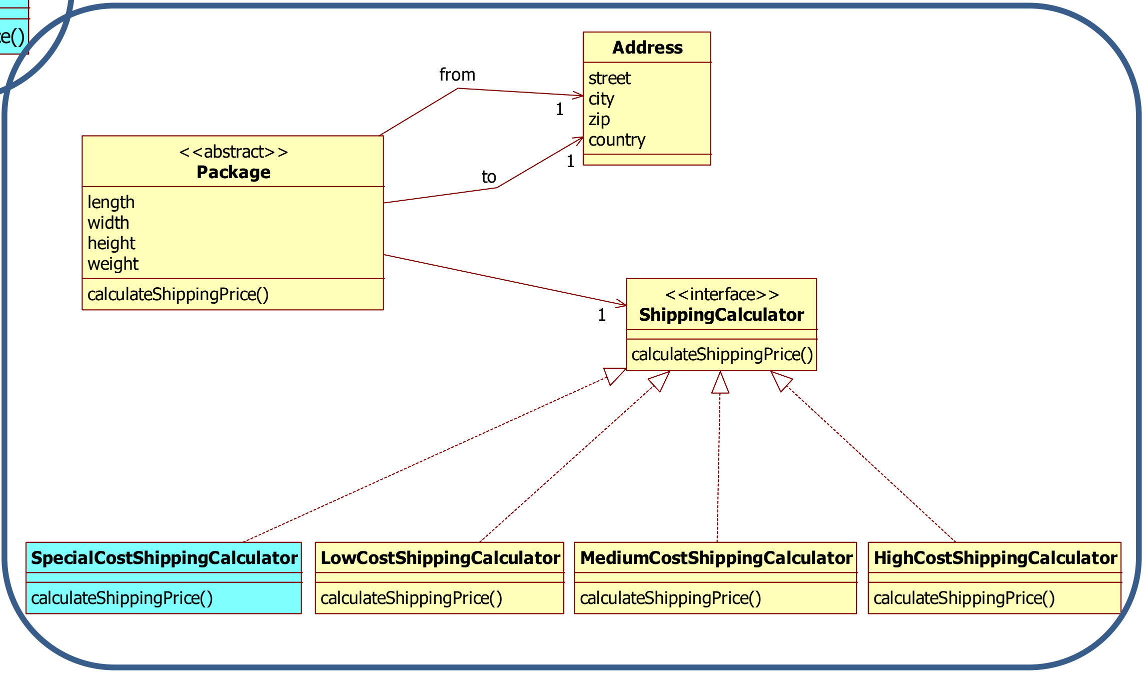
What are the differences?



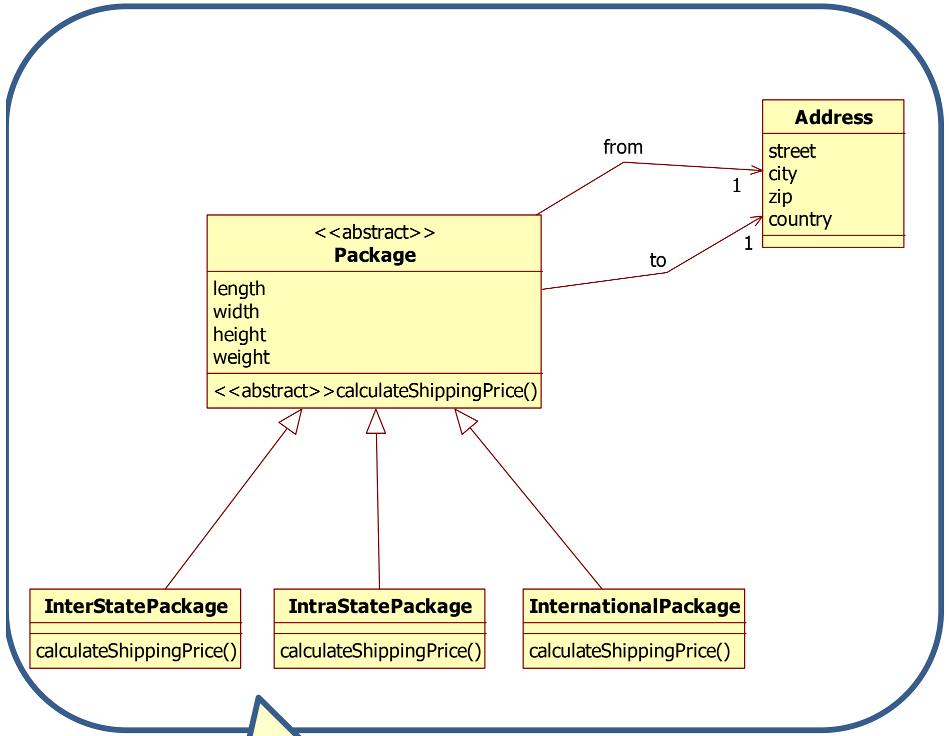


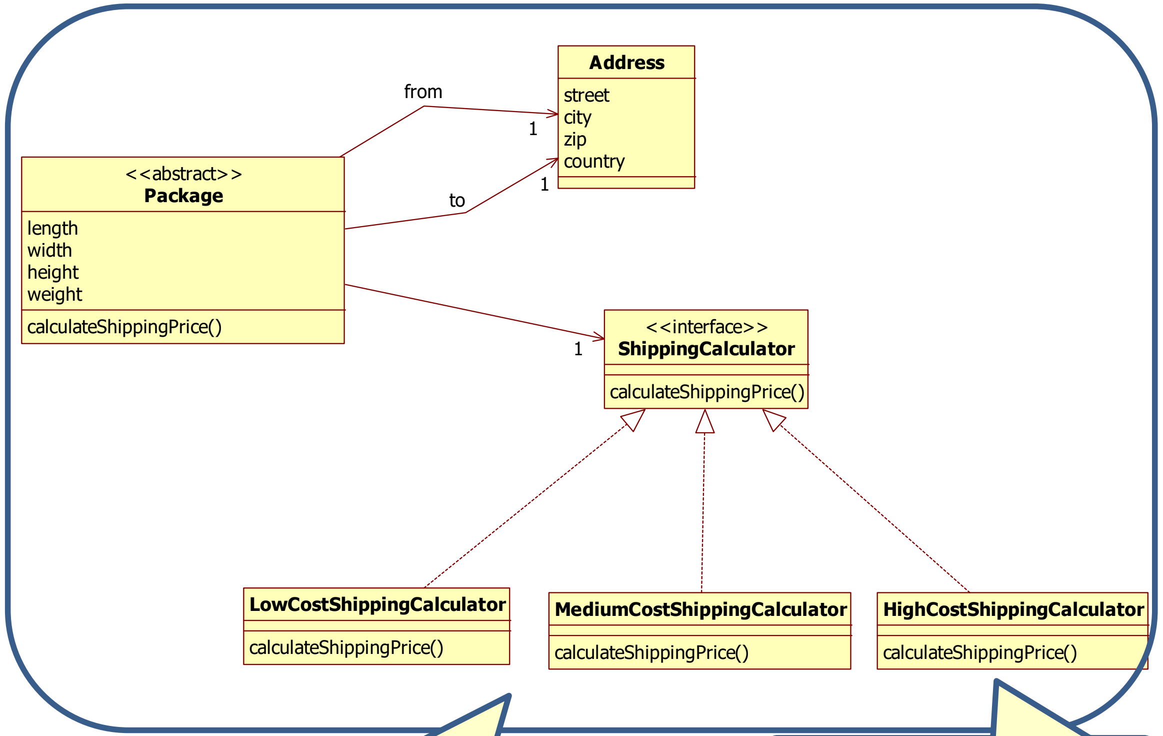
Add a new kind of shipping





The real difference



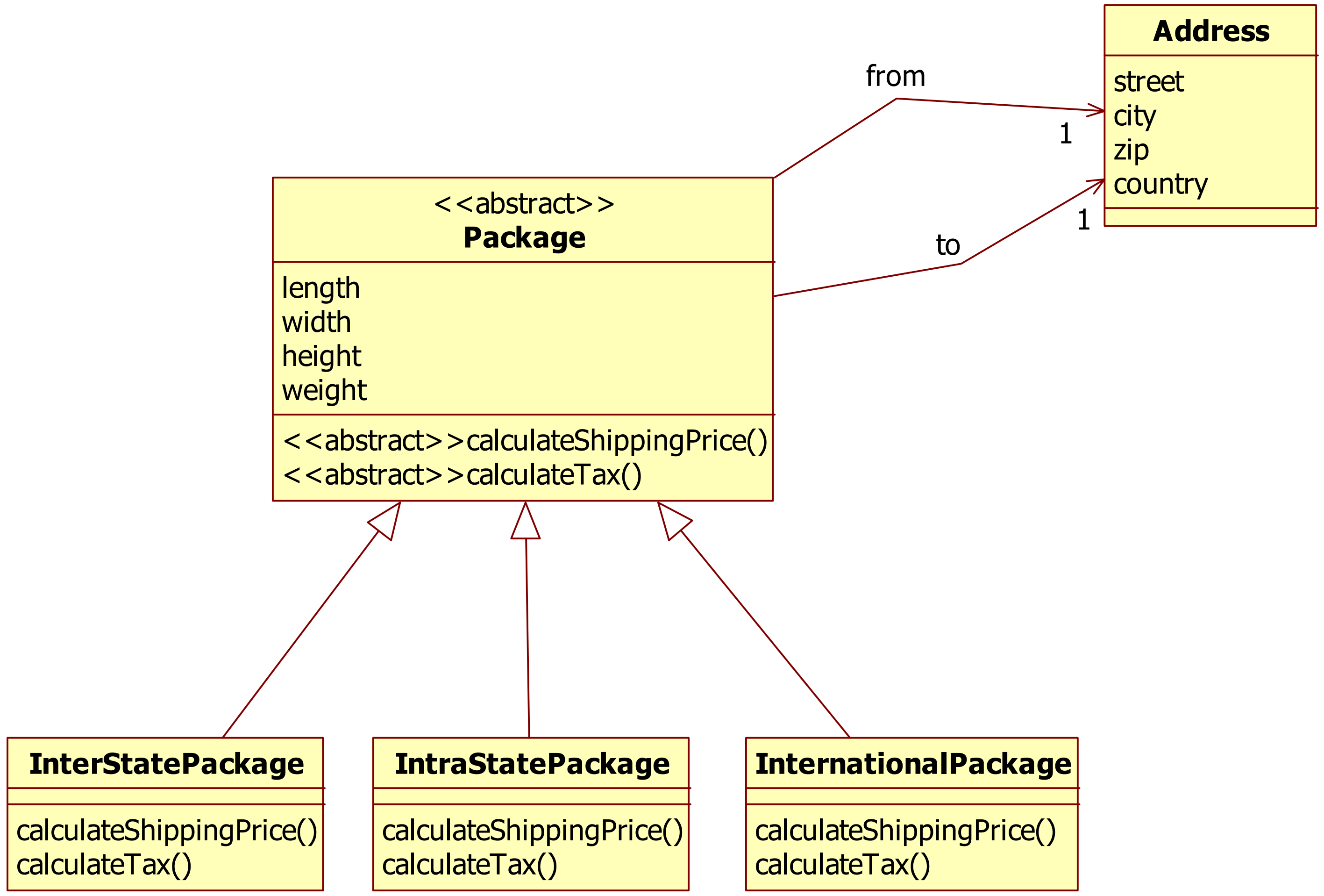


reusable algorithms

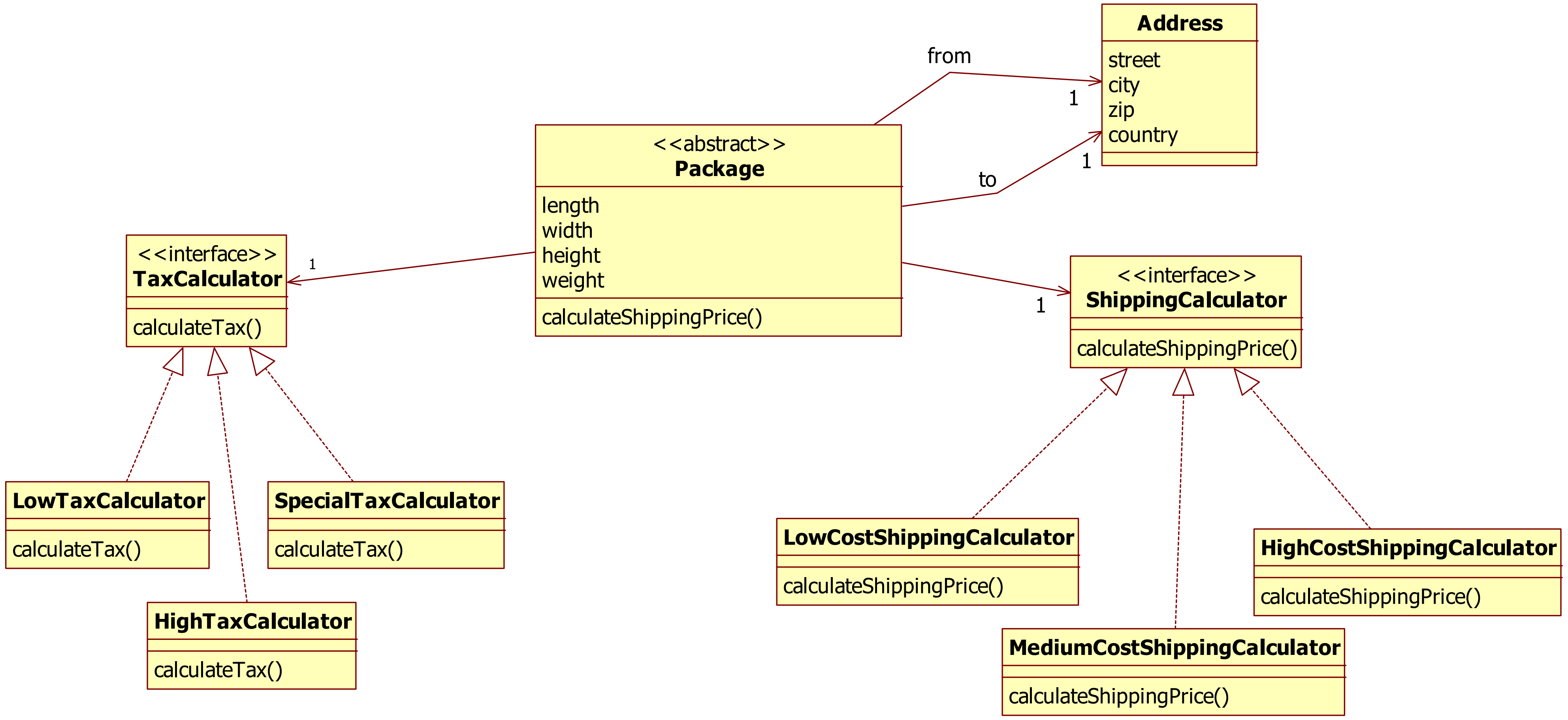
If we use the same algorithm for inter and intra state packages, we have to copy & paste the algorithm

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

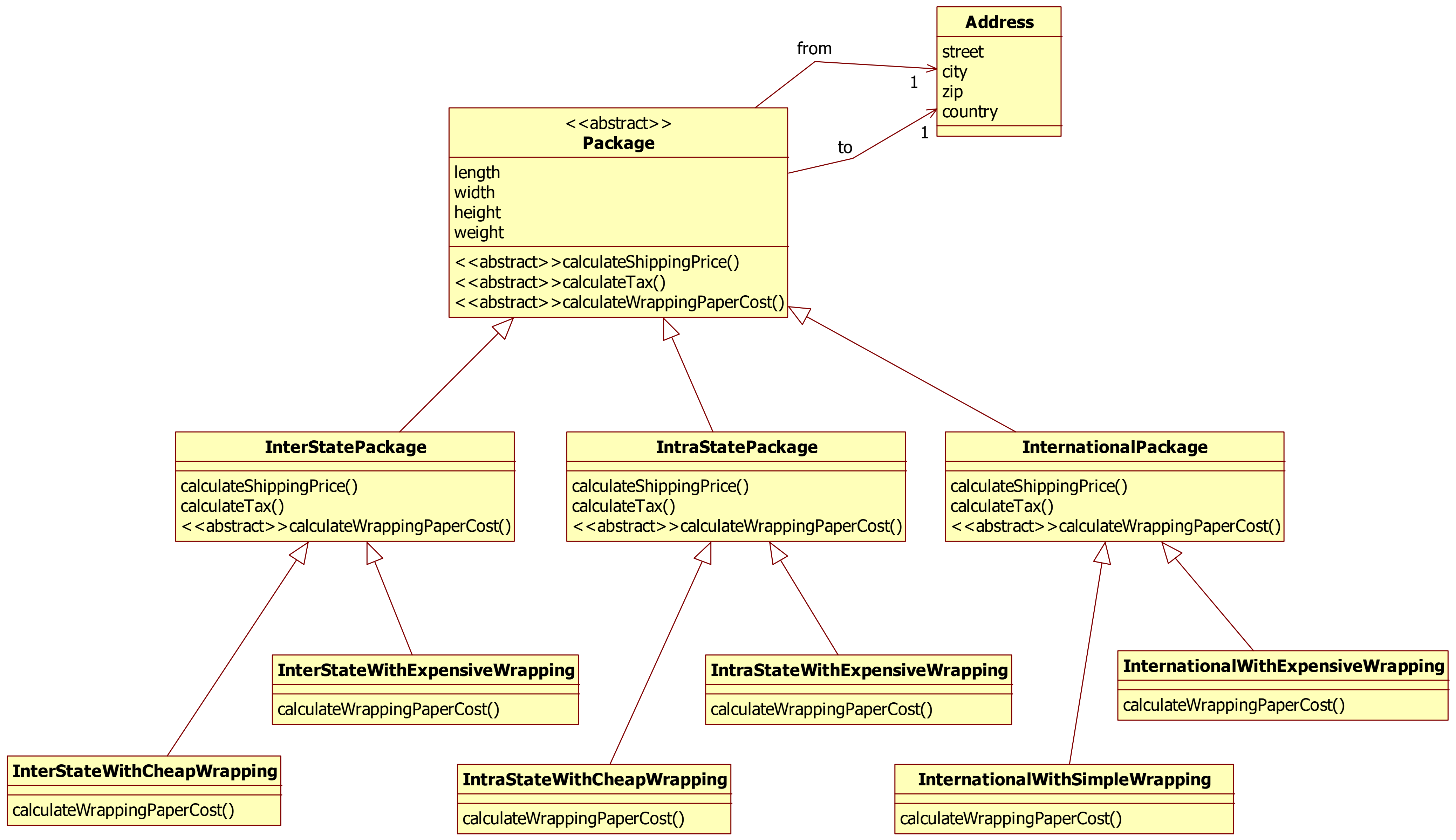
Different ways to calculate tax with inheritance



Different ways to calculate tax with strategy

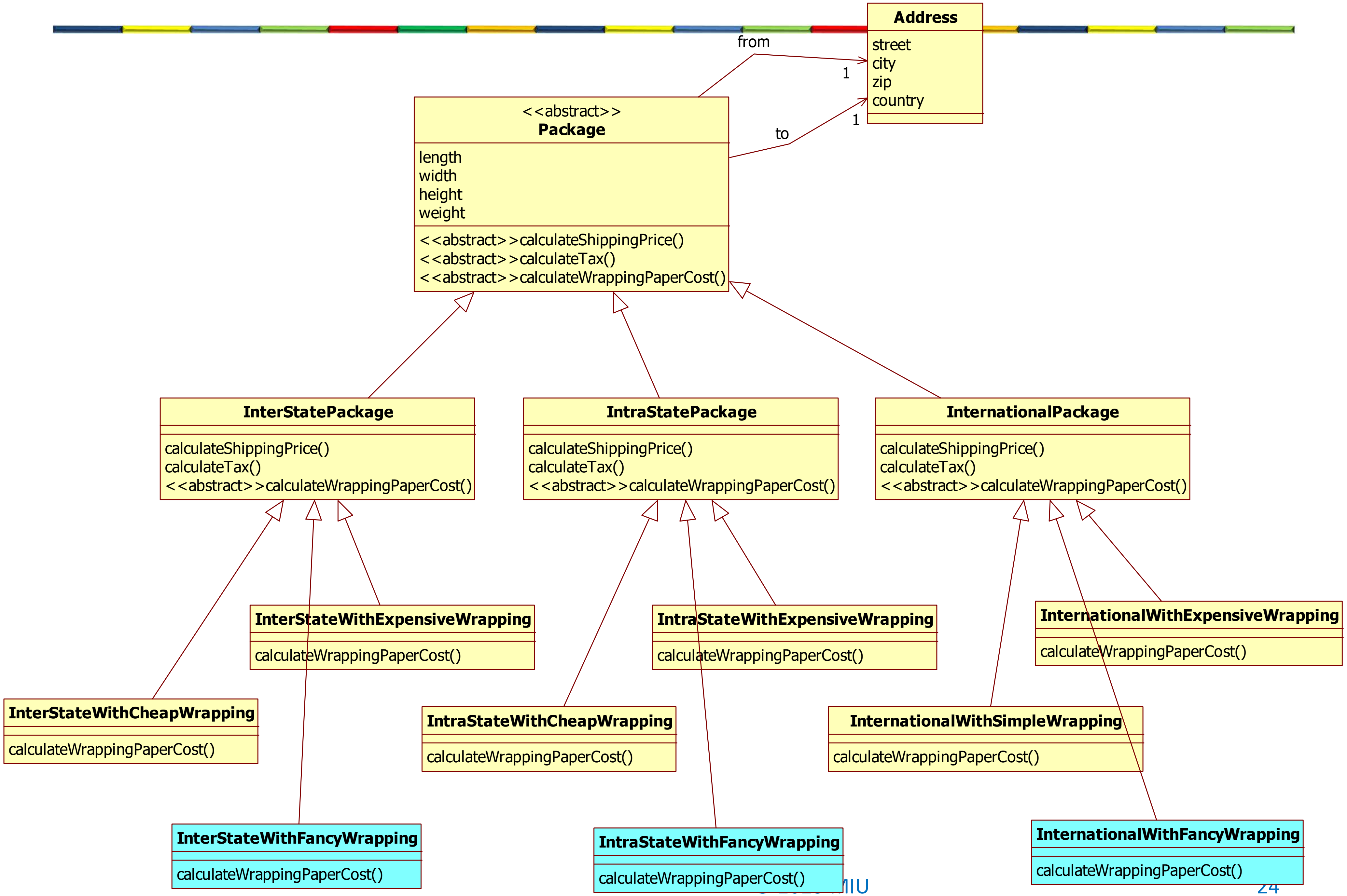


Giftwrap possibilities with inheritance

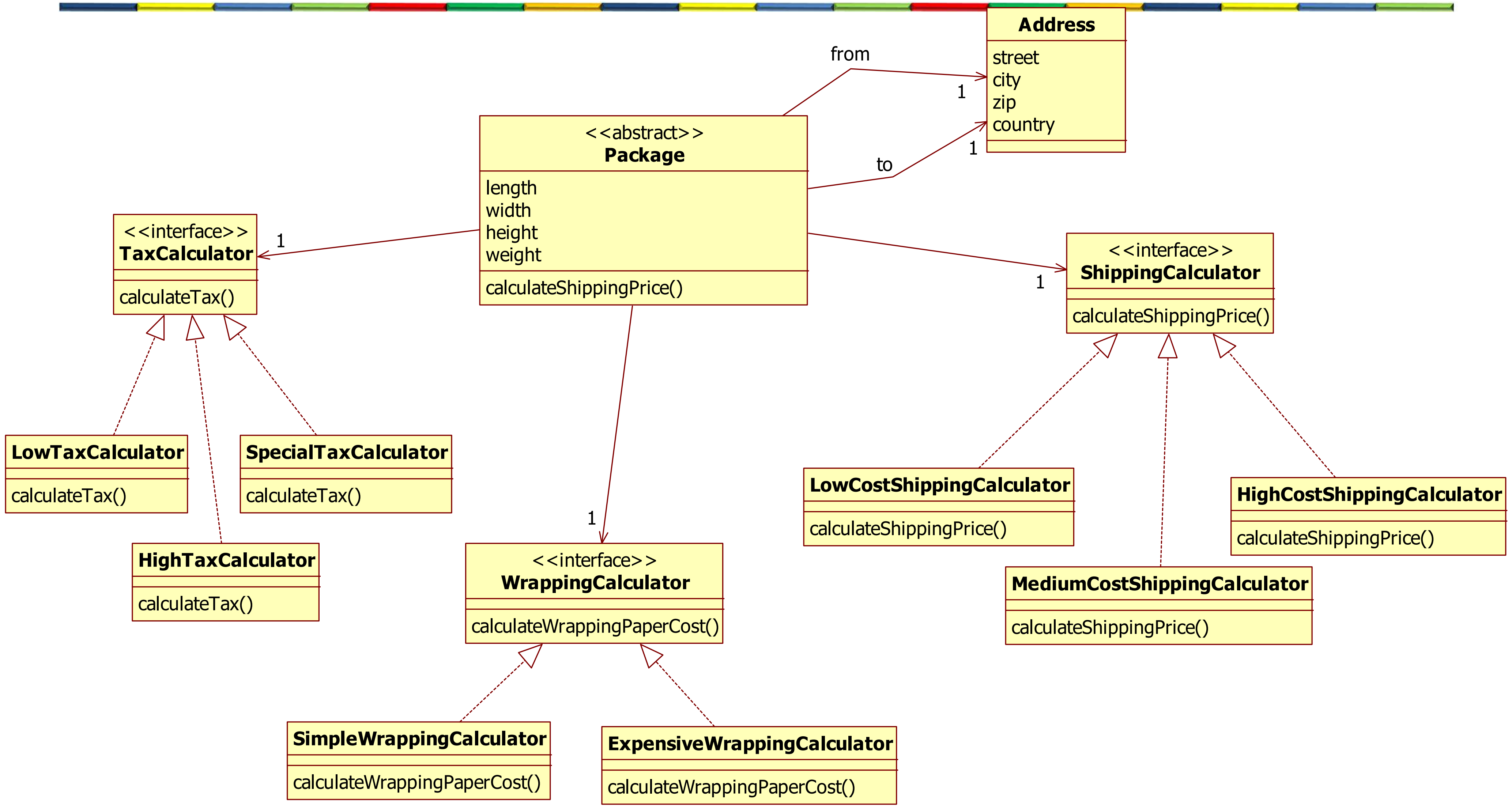


Address
street
city
zip
country
from
1
to
1
<<abstract>>
Package
length
width
height
weight
<<abstract>>calculateShippingPrice()
<<abstract>>calculateTax()
<<abstract>>calculateWrappingPaperCost()
InterStatePackage
calculateShippingPrice()
calculateTax()
<<abstract>>calculateWrappingPaperCost()
IntraStatePackage
calculateShippingPrice()
calculateTax()
<<abstract>>calculateWrappingPaperCost()
InternationalPackage
calculateShippingPrice()
calculateTax()
<<abstract>>calculateWrappingPaperCost()
InterStateWithExpensiveWrapping
calculateWrappingPaperCost()
IntraStateWithExpensiveWrapping
calculateWrappingPaperCost()
InternationalWithExpensiveWrapping
calculateWrappingPaperCost()
InterStateWithCheapWrapping
calculateWrappingPaperCost()
IntraStateWithCheapWrapping
calculateWrappingPaperCost()
InternationalWithSimpleWrapping
calculateWrappingPaperCost()

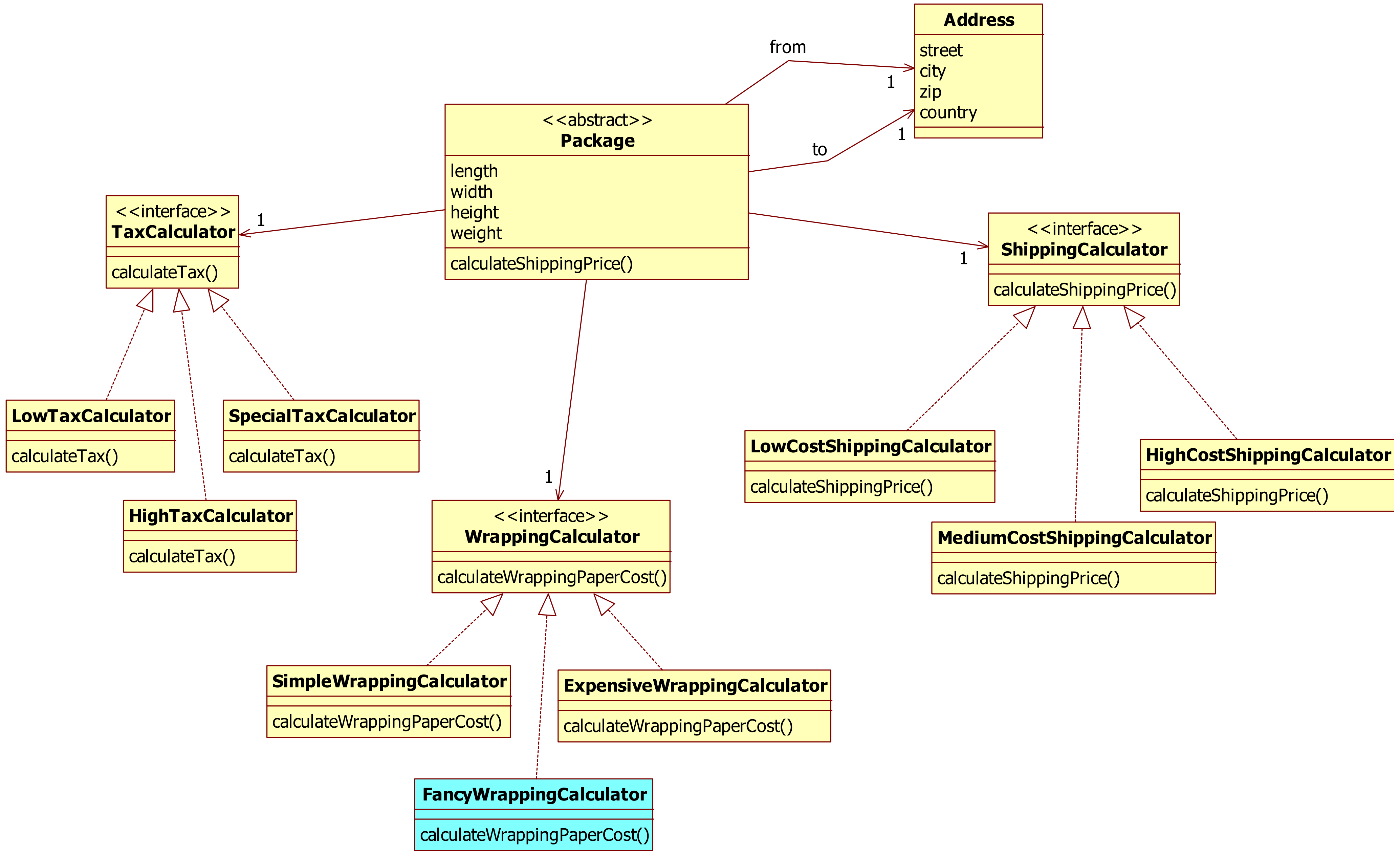
Let’s add Fancy gift wrapping


Address
street
city
zip
country
from
1
to
1
<<abstract>>
Package
length
width
height
weight
<<abstract>>calculateShippingPrice()
<<abstract>>calculateTax()
<<abstract>>calculateWrappingPaperCost()
InterStatePackage
calculateShippingPrice()
calculateTax()
<<abstract>>calculateWrappingPaperCost()
IntraStatePackage
calculateShippingPrice()
calculateTax()
<<abstract>>calculateWrappingPaperCost()
InternationalPackage
calculateShippingPrice()
calculateTax()
<<abstract>>calculateWrappingPaperCost()
InterStateWithExpensiveWrapping
calculateWrappingPaperCost()
IntraStateWithExpensiveWrapping
calculateWrappingPaperCost()
InternationalWithExpensiveWrapping
calculateWrappingPaperCost()
InterStateWithCheapWrapping
calculateWrappingPaperCost()
IntraStateWithCheapWrapping
calculateWrappingPaperCost()
InternationalWithSimpleWrapping
calculateWrappingPaperCost()
InterStateWithFancyWrapping
calculateWrappingPaperCost()
IntraStateWithFancyWrapping
calculateWrappingPaperCost()
InternationalWithFancyWrapping
calculateWrappingPaperCost()

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.

Issues

- Do all strategies share the same interface?
- Who creates the stategies?
- Do the strategies need a reference to the context?

Main point

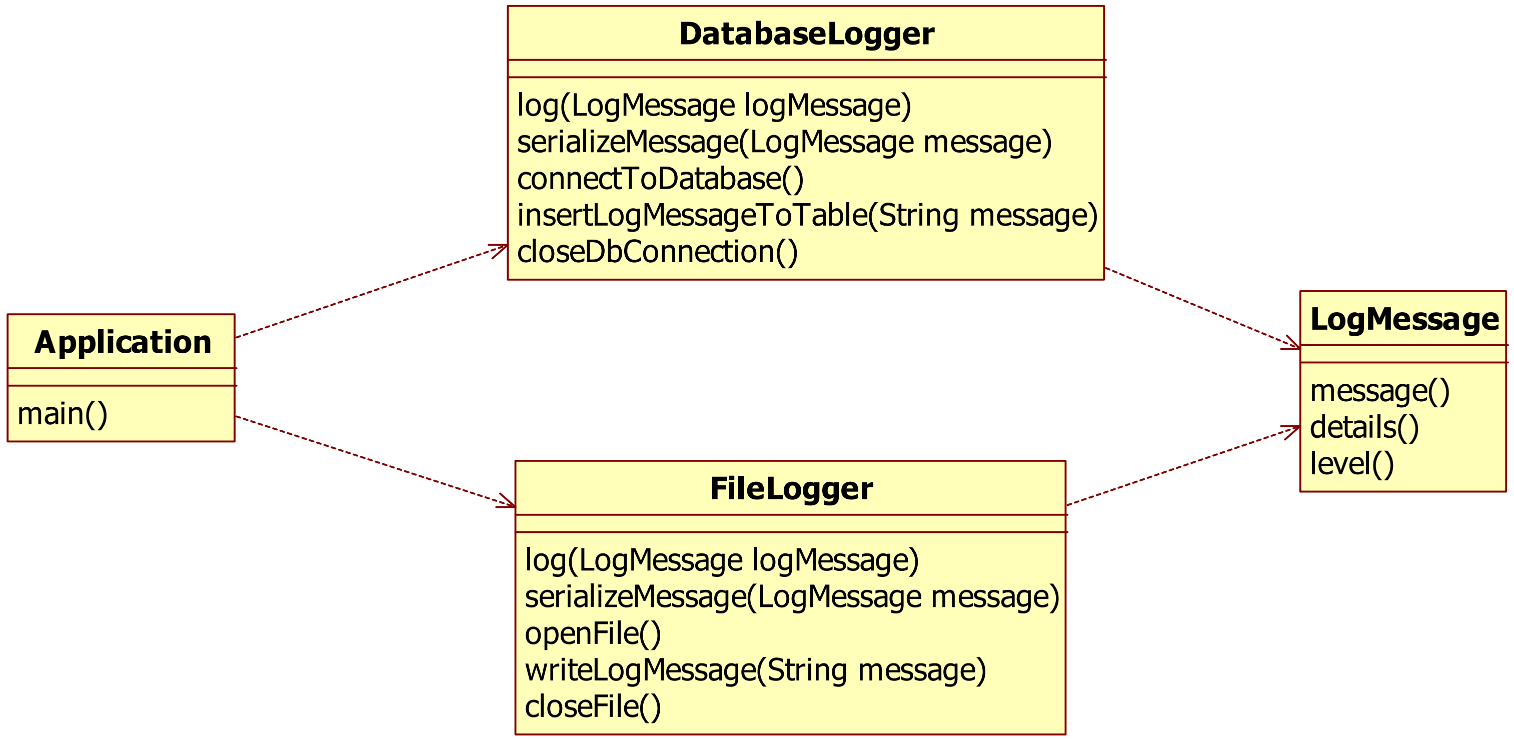
- With the strategy pattern, different algorithms are extracted from its context and encapsulated as strategy classes
- Nature always takes the path of least resistance

TEMPLATE METHOD PATTERN

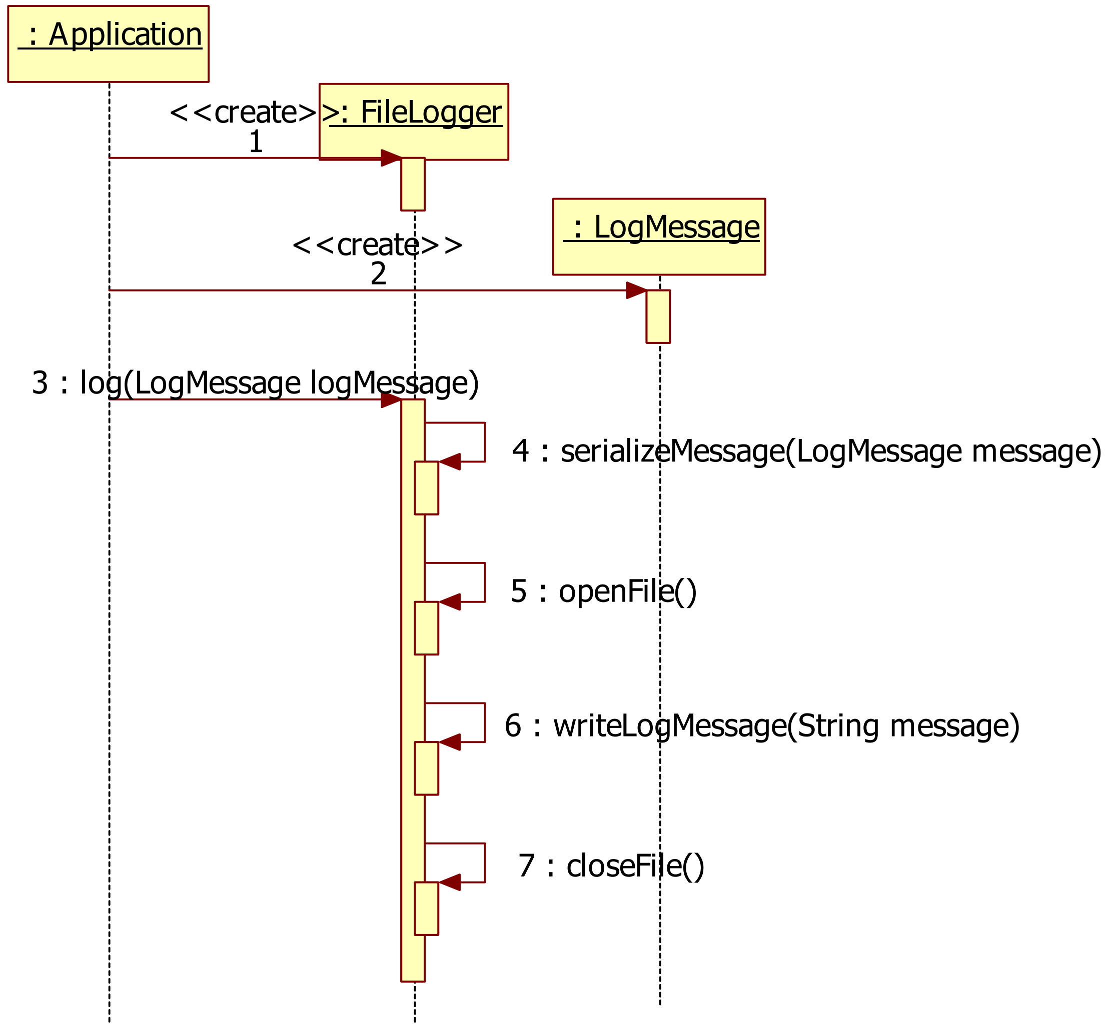
Template method

- The template method pattern defines the skeleton of an algorithm in the superclass but let subclasses override specific steps of the algorithm without changing its structure

Logging to different sources


DatabaseLogger
log(LogMessage logMessage)
serializeMessage(LogMessage message)
connectToDatabase()
insertLogMessageToTable(String message)
closeDbConnection()
Application
main()
LogMessage
message()
details()
level()
FileLogger
log(LogMessage logMessage)
serializeMessage(LogMessage message)
openFile()
writeLogMessage(String message)
closeFile()

Using the FileLogger



: Application
: FileLogger
: LogMessage
<<create>>
1
<<create>>
2
3 : log(LogMessage logMessage)
4 : serializeMessage(LogMessage message)
5 : openFile()
6 : writeLogMessage(String message)
7 : closeFile()

DatabaseLogger

```java
public class DatabaseLogger {
  public void log(LogMessage message) {
    String messageToLog = serializeMessage(message);
    connectToDatabase();
    insertLogMessageToTable(messageToLog);
    closeDbConnection();
  }
  private String serializeMessage(LogMessage message) {
    System.out.println("Serializing message");
    return message.toString();
  }
  private void connectToDatabase() {
    System.out.println("Connecting to Database.");
  }
  private void insertLogMessageToTable(String message) {
    System.out.println("Inserting Log Message to DB table : " + message);
  }
  private void closeDbConnection() {
    System.out.println("Closing DB connection.");
  }
}
```

FileLogger

```
public class FileLogger{
  public void log(LogMessage message) {
    String messageToLog = serializeMessage(message);
    openFile();
    writeLogMessage(messageToLog);
    closeFile();
  }
  private String serializeMessage(LogMessage message) {
    System.out.println("Serializing message");
    return message.toString();
  }
  private void openFile() {
    System.out.println("Opening File.");
  }
  private void writeLogMessage(String message) {
    System.out.println("Appending Log message to file : " + message);
  }
  private void closeFile() {
    System.out.println("Close File.");
  }
}
```

LogMessage

```java
public class LogMessage {
  enum LogLevel {
    WARNING,
    INFO,
    ERROR
  }

  private String message;
  private String details;
  private LogLevel level;

  public LogMessage(String message, String details, LogLevel level) {
    this.message = message;
    this.details = details;
    this.level = level;
  }

  @Override
  public String toString() {
    return "LogMessage "+LocalDate.now()+" - "+LocalTime.now()+" [message=" +
            message + ", details=" + details + ", level=" + level + "]";
  }
}
```

Application

```
public class Application {

  public static void main(String[] args) {
    FileLogger fileLogger = new FileLogger();
    LogMessage message = new LogMessage("cannot send email", "smpt server
                smtp.acme.com cannot be reached", LogLevel.ERROR);
    fileLogger.log(message);

    System.out.println("------------------------");

    DatabaseLogger databaseLogger = new DatabaseLogger();
    LogMessage message2 = new LogMessage("subject is empty", "this email has no
               subject, emails should have a subject", LogLevel.INFO);
    databaseLogger.log(message2);
  }
}
```

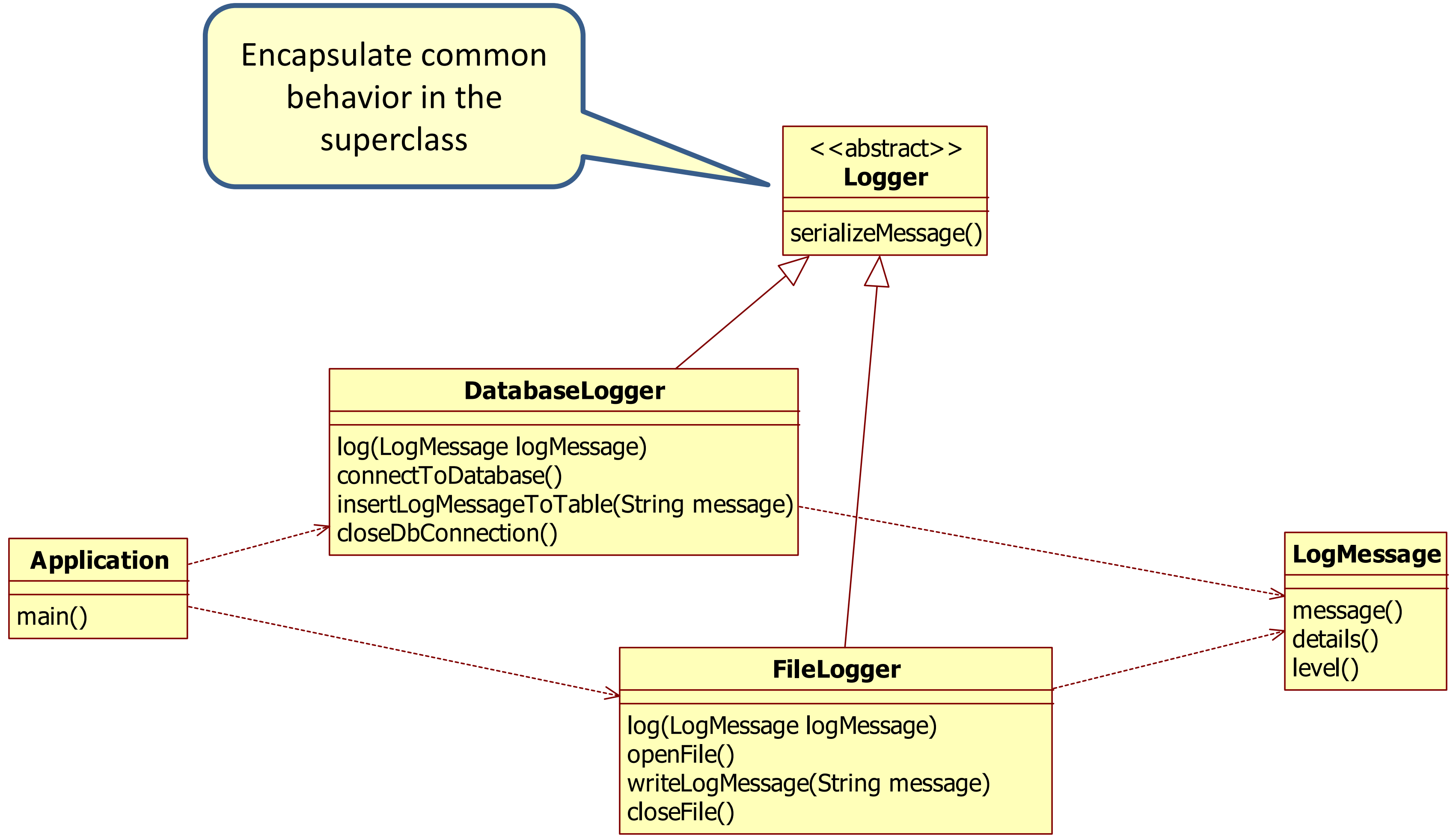
Common behavior

```java
public class FileLogger{
  public void log(LogMessage message) {...}

  private String serializeMessage(LogMessage message) {
    System.out.println("Serializing message");
    return message.toString();
  }
  private void openFile() {...}
  private void writeLogMessage(String message) {...}
  private void closeFile() {...}
}
```

Serializing the LogMessage is the same for all loggers

```java
public class DatabaseLogger {
  public void log(LogMessage message) {...}

  private String serializeMessage(LogMessage message) {
    System.out.println("Serializing message");
    return message.toString();
  }
  private void connectToDatabase() {...}
  private void insertLogMessageToTable(String message) {...}
  private void closeDbConnection() {...}
}
```

Inheritance



DatabaseLogger

```java
public abstract class Logger {
    protected String serializeMessage(LogMessage message) {
        System.out.println("Serializing message");
        return message.toString();
    }
}
```

```java
public class DatabaseLogger extends Logger {
  public void log(LogMessage message) {
    String messageToLog = serializeMessage(message);
    connectToDatabase();
    insertLogMessageToTable(messageToLog);
    closeDbConnection();
  }
  private void connectToDatabase() {
    System.out.println("Connecting to Database.");
  }
  private void insertLogMessageToTable(String message) {
    System.out.println("Inserting Log Message to DB table : " + message);
  }
  private void closeDbConnection() {
    System.out.println("Closing DB connection.");
  }
}
```

FileLogger

```java
public abstract class Logger {
    protected String serializeMessage(LogMessage message) {
        System.out.println("Serializing message");
        return message.toString();
    }
}
```

```java
public class FileLogger extends Logger {
  public void log(LogMessage message) {
    String messageToLog = serializeMessage(message);
    openFile();
    writeLogMessage(messageToLog);
    closeFile();
  }
  private void openFile() {
    System.out.println("Opening File.");
  }
  private void writeLogMessage(String message) {
    System.out.println("Appending Log message to file : " + message);
  }
  private void closeFile() {
    System.out.println("Close File.");
  }
}
```

A common algorithm

```java
public class DatabaseLogger extends Logger {
  public void log(LogMessage message) {
    String messageToLog = serializeMessage(message);
    connectToDatabase();
    insertLogMessageToTable(messageToLog);
    closeDbConnection();
  }
  ...
}
```

Open/connect to repository

Write log message to repository

Close repository

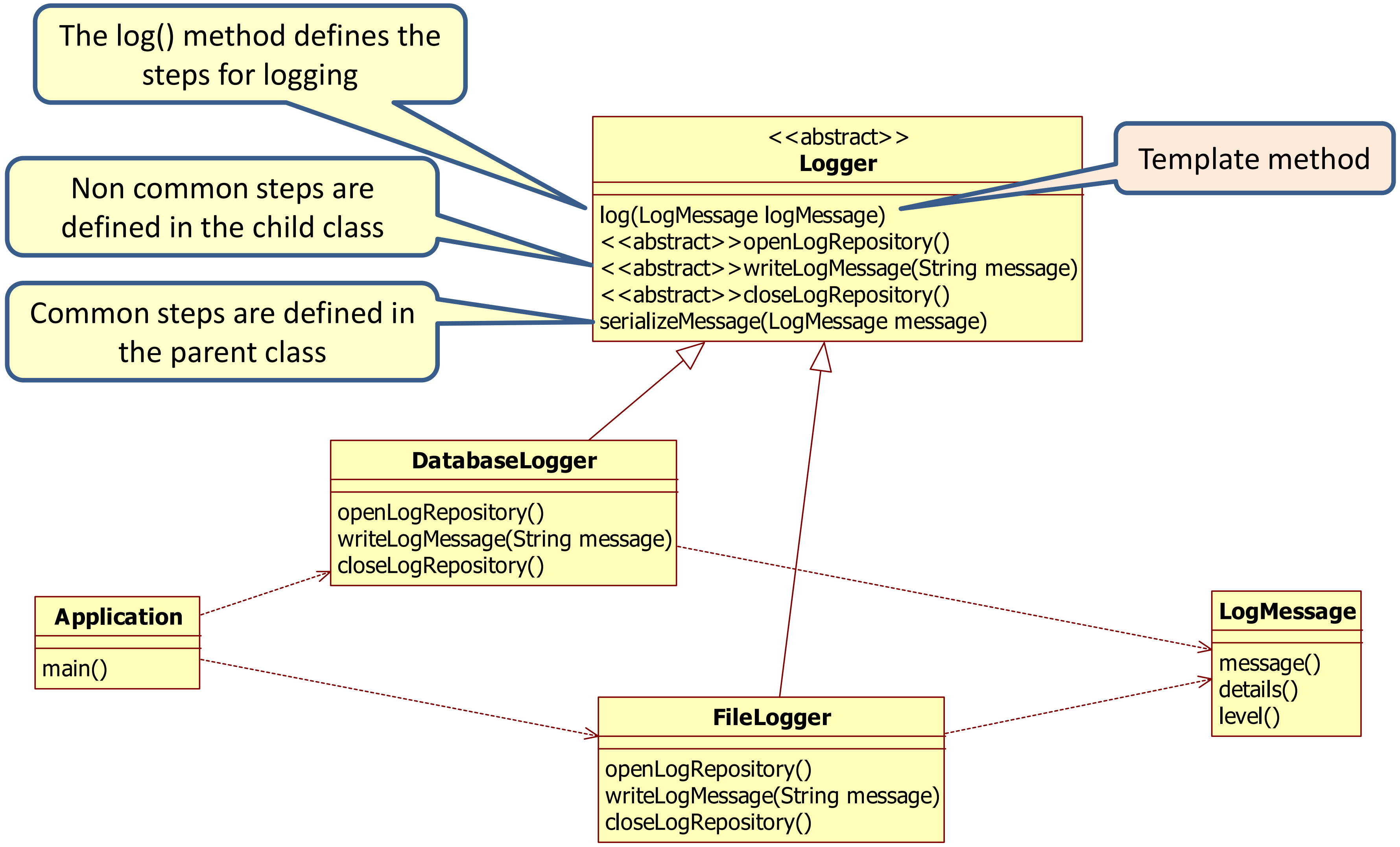
```java
public class FileLogger extends Logger {
  public void log(LogMessage message) {
    String messageToLog = serializeMessage(message);
    openFile();
    writeLogMessage(messageToLog);
    closeFile();
  }
  ...
}
```

Open/connect to repository

Write log message to repository

Close repository

Template method pattern


The log() method defines the steps for logging
Non common steps are defined in the child class
Common steps are defined in the parent class
Template method
<<abstract>>
Logger
log(LogMessage logMessage)
<<abstract>>openLogRepository()
<<abstract>>writeLogMessage(String message)
<<abstract>>closeLogRepository()
serializeMessage(LogMessage message)
DatabaseLogger
openLogRepository()
writeLogMessage(String message)
closeLogRepository()
FileLogger
openLogRepository()
writeLogMessage(String message)
closeLogRepository()
Application
main()
LogMessage
message()
details()
level()

Logger

```
public abstract class Logger {
  protected void log(LogMessage message) {
    String messageToLog = serializeMessage(message);
    openLogRepository();
    writeLogMessage(messageToLog);
    closeLogRepository();
  }
  protected abstract void openLogRepository() ;
  protected abstract void writeLogMessage(String message);
  protected abstract void closeLogRepository();

  protected String serializeMessage(LogMessage message) {
    System.out.println("Serializing message");
    return message.toString();
  }

}
```

Template method

The log() method defines the steps for logging

Non common steps are defined in the child class

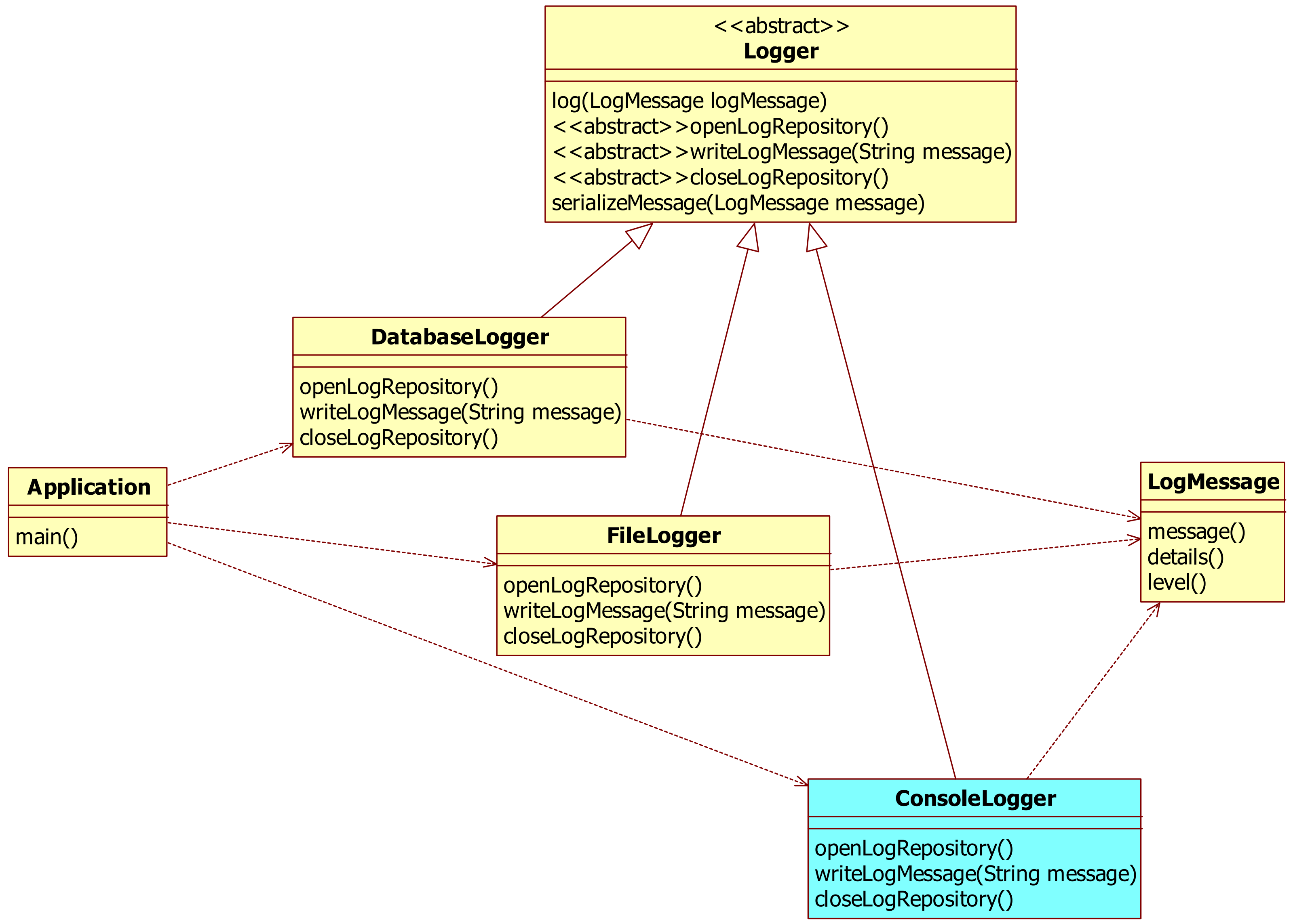
Common steps are defined in the parent class

Concrete loggers

```java
public class DatabaseLogger extends Logger{

  protected void openLogRepository() {
    System.out.println("Connecting to Database.");
  }
  protected void writeLogMessage(String message) {
    System.out.println("Inserting Log Message to DB table : " + message);
  }
  protected void closeLogRepository() {
    System.out.println("Closing DB connection.");
  }
}
```

```java
public class FileLogger extends Logger{

  protected void openLogRepository() {
    System.out.println("Opening File.");
  }
  protected void writeLogMessage(String message) {
    System.out.println("Appending Log message to file : " + message);
  }
  protected void closeLogRepository() {
    System.out.println("Close File.");
  }
}
```

Let’s add a ConsoleLogger



<<abstract>>
Logger
log(LogMessage logMessage)
<<abstract>>openLogRepository()
<<abstract>>writeLogMessage(String message)
<<abstract>>closeLogRepository()
serializeMessage(LogMessage message)
DatabaseLogger
openLogRepository()
writeLogMessage(String message)
closeLogRepository()
Application
main()
FileLogger
openLogRepository()
writeLogMessage(String message)
closeLogRepository()
LogMessage
message()
details()
level()
ConsoleLogger
openLogRepository()
writeLogMessage(String message)
closeLogRepository()

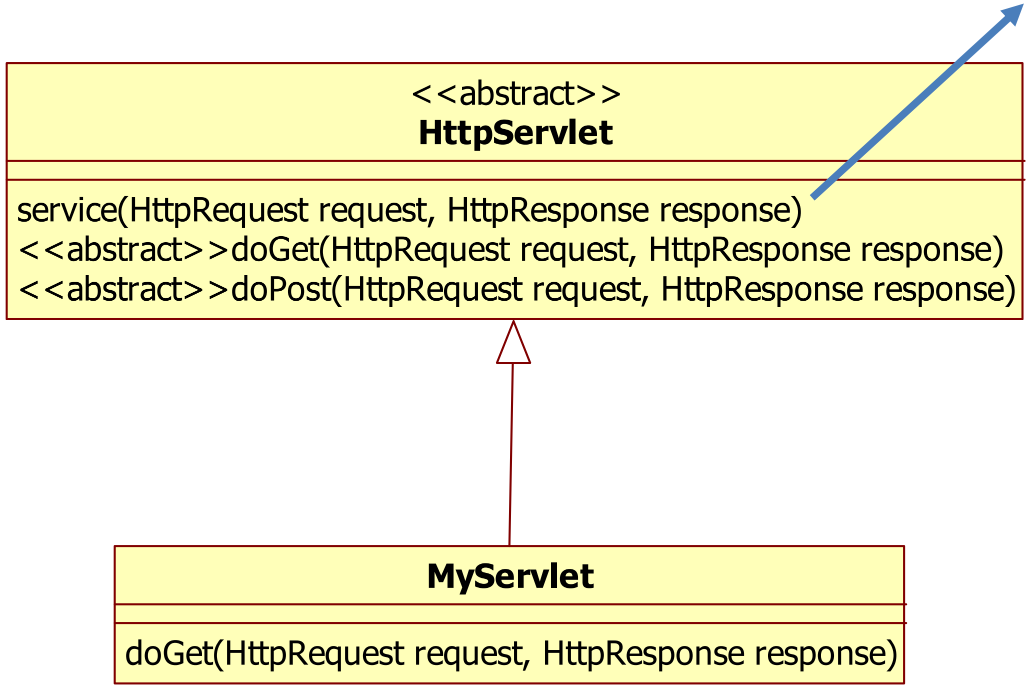
ConsoleLogger

```java
public class ConsoleLogger extends Logger {

  protected void openLogRepository() {}
  protected void writeLogMessage(String message) {
   System.out.println("Console logger: "+message);
  }
  protected void closeLogRepository() {}
}
```

```java
ConsoleLogger consoleLogger = new ConsoleLogger();
LogMessage message3 = new LogMessage("this email has multiple receivers", "try
       the mailmerge functionality", LogLevel.WARNING);
consoleLogger.log(message3);
```

Example of template method

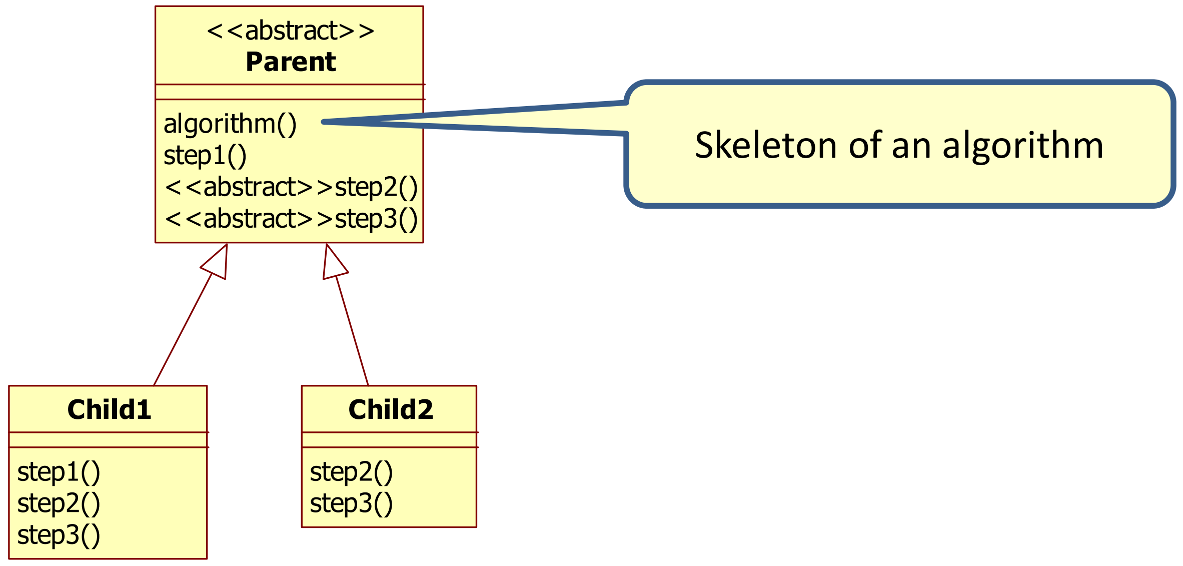
```
if (request.getType().equals("GET"))
  doGet(request, response);
else if (request.getType().equals("POST"))
  doPost(request, response);
```



<<abstract>>
HttpServlet
service(HttpRequest request, HttpResponse response)
<<abstract>>doGet(HttpRequest request, HttpResponse response)
<<abstract>>doPost(HttpRequest request, HttpResponse response)
MyServlet
doGet(HttpRequest request, HttpResponse response)

Template method pattern

- What problem does it solve?
  - Whenever you have an algorithm with different steps that is used in different situations, then define this algorithm in one place (parent class) and let child classes implement the concrete steps.



Main point

- The template method defines an algorithm in the parent class and the different steps can be implemented in the child class(es)
- Every human being has free will to decide how to live your life within the boundaries of the laws of nature.