MSO
Design Patterns: Strategy

Hans Philippi

September 17, 2018

Case study: e-commerce

- Suppose we need to write software for an online retailer (Amazon, Bol, . . .)
- We know that different countries have different address formats, tax rules, or shipping costs
- Even if we do not yet know *which* countries we will ship to, or *how* these issues will be implemented exactly, we *do* know where to expect variation

General architecture

- Introduce a *controller* object that processes sales requests
- It delegates the handling of sales to a SalesOrder class:
  - Allows for filling out the order with a GUI
  - Handles tax calculation
  - Processes the order
  - Prints a receipt
- Probably with the help of other classes

Anticipating variation

- How can I handle new taxation rules?
- There are a few alternatives you may already be familiar with:
  - Copy-and-paste
  - Switches or ifs
  - Inheritance (possibly nested)
- But these techniques all have drawbacks . . .

Copy-and-paste

We don't have to discuss this, do we?

Conditionals

```
public int computeTax(country, price)
  switch(country)
  {
    case "NL":
      return price * 0.21;
      break;
    case "UK":
      // Insert code here
      break;
  }
```

This works fine – how extensible is it?

Conditionals

```
public int computeShipping(country)
  switch(country)
  {
    case "NL":
      // Insert code here
      break;
    case "UK":
      // Insert code here
      break;
  }
```

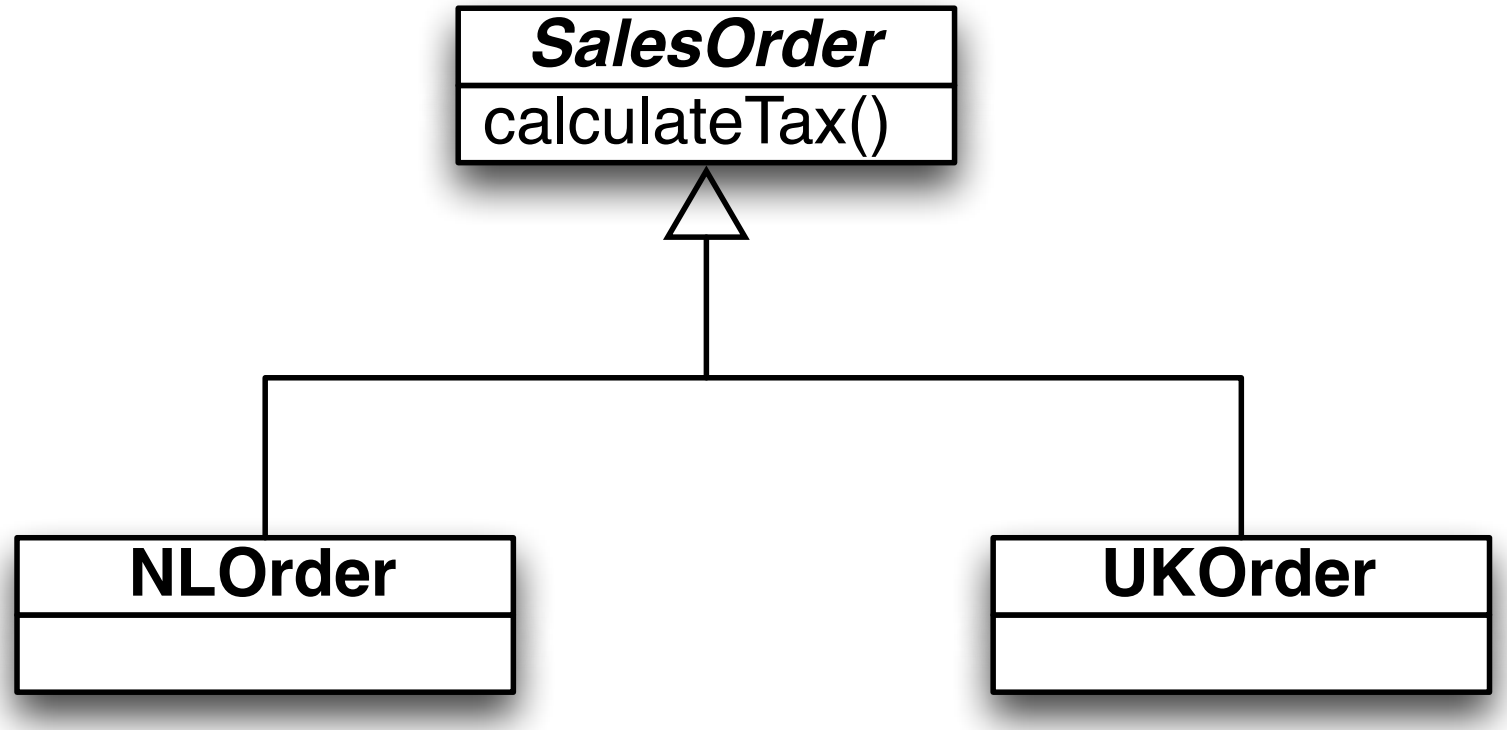
Conditionals

```
public int convertCurrency(country)
  switch(country)
  {
    case "NL":
      // Insert code here
      break;
    case "UK":
      // Insert code here
      break;
  }
```

Handling change

- What happens if I want to start shipping to Germany?
- Suddenly, I need to change lots of pieces of code, spread over different files: lack of *cohesion*!

Inheritance



Inheritance: a critical evaluation

- Introducing this class hierarchy enables us to vary the tax calculation in every subclass
- But what about handling other variations:
  - Different languages (like in Belgium)
  - Different date formats
  - Different shipping costs
  - Different address formats
  - ...
- How can I share code between subclasses?

Many modifications may require us to revisit the entire hierarchy

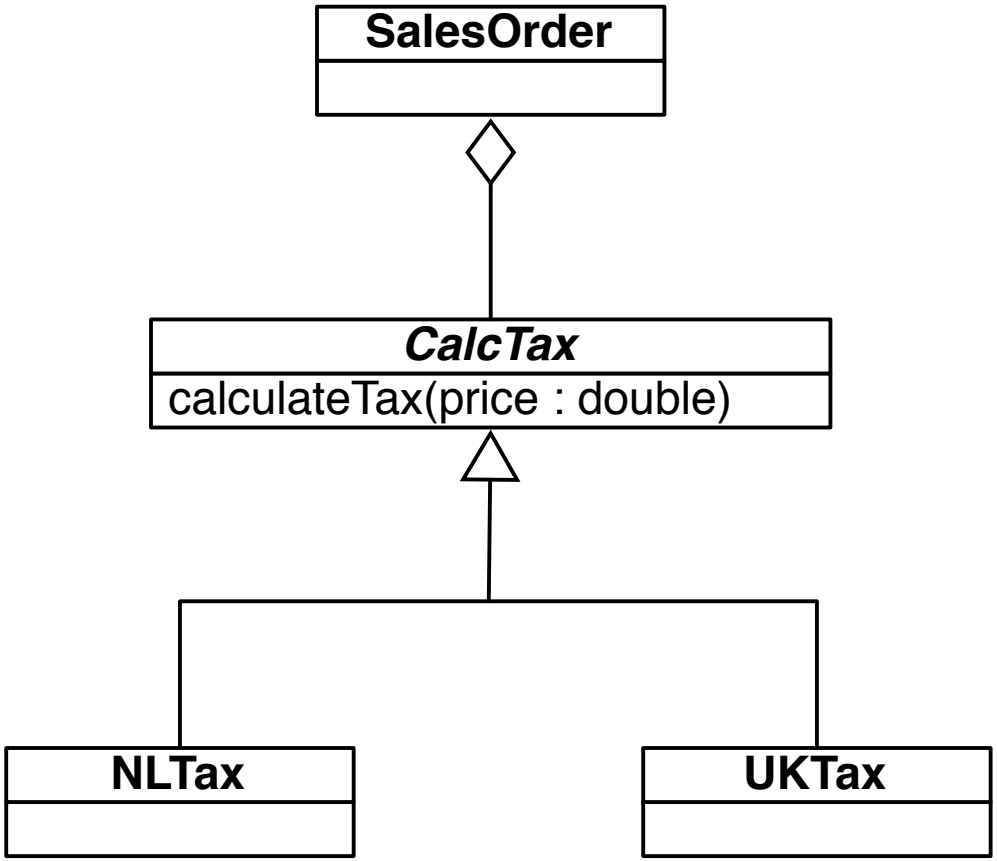
A better approach...

1. Find what varies and encapsulate it in a class of its own
2. Contain this class in the original class

More concretely

1. The tax calculation across different countries is different, so we need to introduce a separate class, CalcTax
2. Every SalesOrder should have a CalcTax

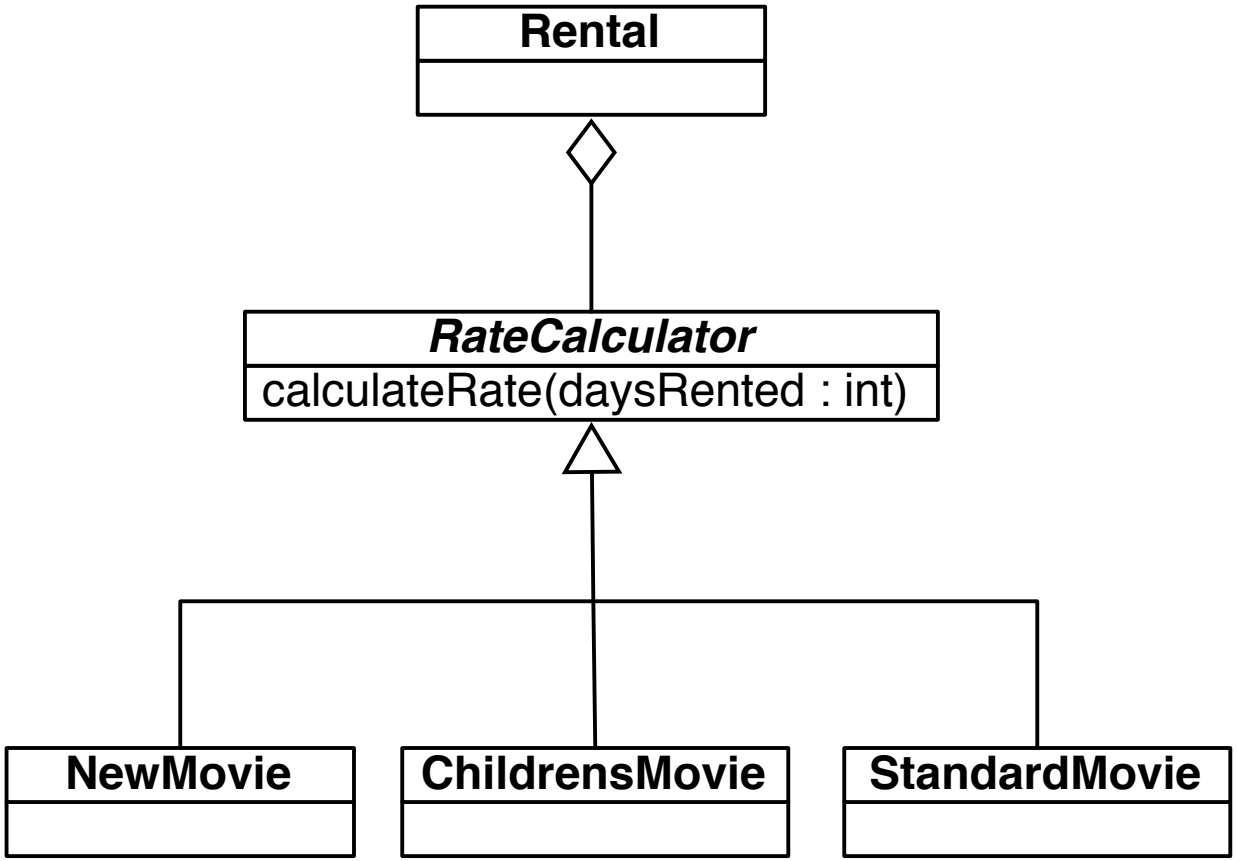
More concretely



Another example

1. Different kinds of movies have different rental rates, so we need to introduce a separate class, RateCalculator
2. Every Rental should have a RateCalculator

Another example



Or in code

```
public class Rental
{
  private rateCalculator RateCalculator;
  ...
}
```

Another example

```
abstract class RateCalculator
{
  public abstract int calculateRate(days int)
}
```

Another example

```
class NewMovieRateCalculator : RateCalculator
{
  public override int calculateRate(days int)
  {
    // New movies cost 5 euros per day to rent
    return 5 * daysRented;
  }
}
```

And similar implementations for other price classes

Why is this better than inheritance?

- Cohesion is stronger
  - There is one place where all rate calculation happens
- Easier to shift responsibility
  - Users of the SalesOrder or Movie class do not need to decide how to compute taxes or rental prices
  - When a New movie is no longer New, future rentals will assign the right RateCalculator

Is inheritance bad?

- This solution still uses inheritance (NewMovieCalculator is a subclass of RateCalculator)
- But it uses inheritance very carefully, to capture one single kind of variation

The Strategy Pattern

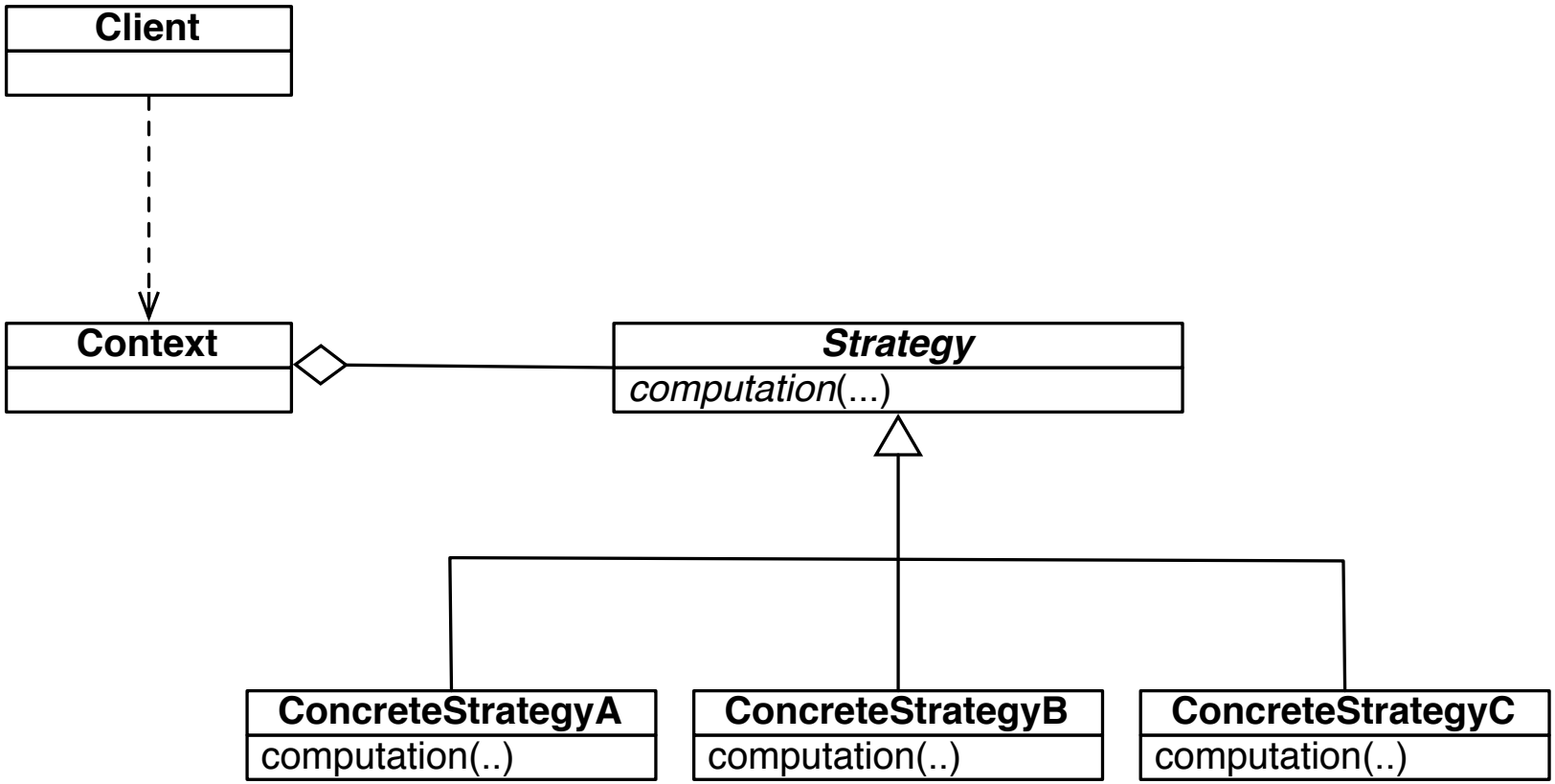
> *Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from the clients that use it.*

The Gang of Four, describing the Strategy pattern's intent

Strategy Pattern

- **Intent:** Use different business rules or algorithms depending on the context
- **Problem:** The selection of a computation that needs to be performed depends on the client making the request or the data being acted on
- **Solution:** Separate the *selection* of the algorithm from the *implementation* of the algorithm. Allows for the selection to be made in context.
- **Participants:**
  - The abstract Strategy class specifies the different algorithms
  - ConcreteStrategies implement such algorithms
  - Context uses a specific ConcreteStrategy
- **Consequences:**
  - Defines a family of algorithms
  - Switches/conditionals can be minimised
  - You invoke all algorithms in the same way

Strategy: Implementation



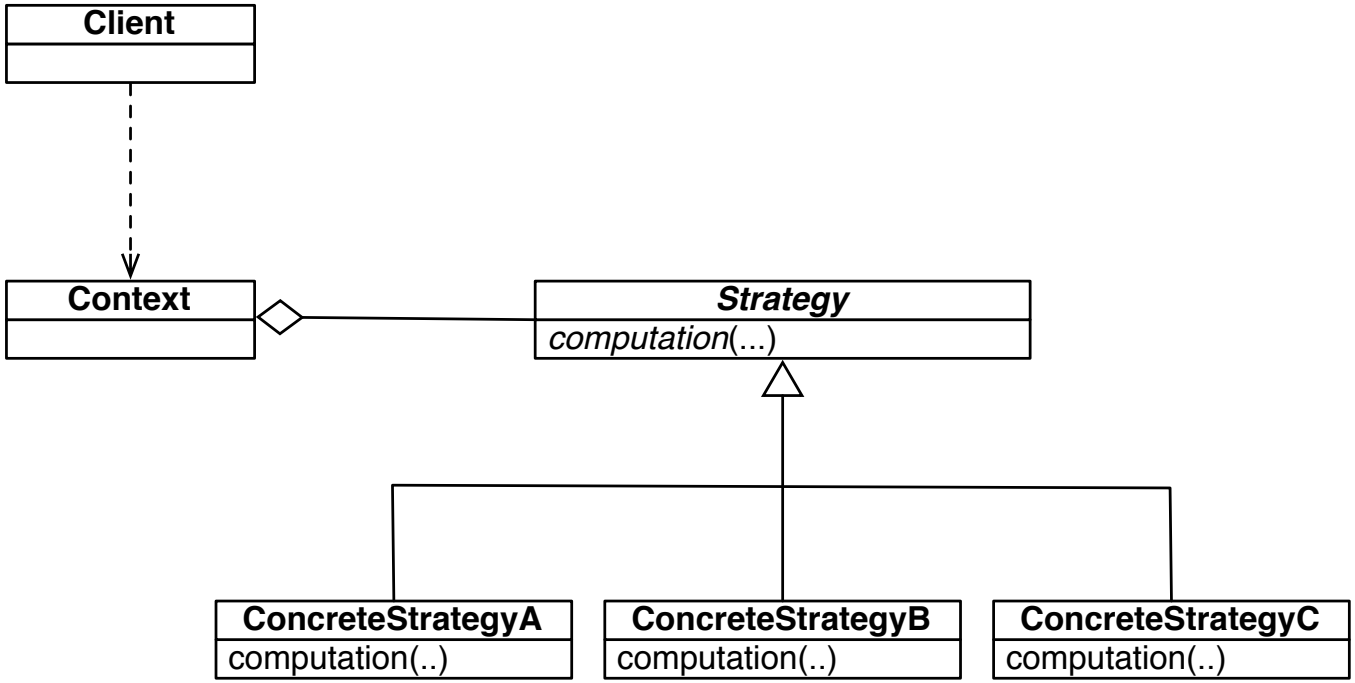
Notes on using the Strategy pattern

- The computation in the ConcreteStrategy classes may need extra information
- It is the responsibility of the Context to gather the required information and pass it to the computations, for instance by parameters or configuration data
- Applying the Strategy class makes it *easier* to test code
- You can write unit tests for individual ConcreteStrategy objects, without having to worry about the rest of the context

Example

- We will finish with a more elaborate example
- Our class SalesOrder deals with diversity in countries
- We apply a strategy approach to Taxes
- We apply a strategy approach to Currencies
- We apply a strategy approach to Language issues
- Each of these three abstract classes covers a one level hierarchy of variation
- So there are three objects aggregated in the class SalesOrder

Strategy x 3



So Context (= SalesOrder) will contain three Strategy objects, each with a one level subclass hierarchy

Example: coding of the choice in SalesOrder

- Of course, we have to choose the country where the order comes from ...
- ... but we do it only once: in the class that has the responsibility to choose: SalesOrder

```
switch(country)
    case ‘‘NL’’:
        myTax = new NLTax();
        myCurr = new Euro();
        myLanguage = new Dutch();
        break;
```

Example: coding of the choice in SalesOrder

- Notice the possibility of sharing Currencies and Languages!

```
switch(country)
...
    case ‘‘IRL’’:
        myTax = new IRLTax();
        myCurr = new Euro();
        myLanguage = new English();
        break;

    case ‘‘UK’’:
        myTax = new UKTax();
        myCurr = new Pound();
        myLanguage = new English();
        break;
```

Example: coding of the choice in SalesOrder

- When dealing with Taxes, Currencies and Languages, only references to properties and methods in the abstract classes are used!

```
...
double tax =
    myTax.TaxCalculator(totalAmount);
...
double amountInLocalCurrency =
    myCurr.CurrencyCalculator(amount);
...
myLanguage.printInvoiceHeader();
```

Epilog

- We have introduced a more advanced pattern: Strategy
- It's main goal is to separate usage of calculations from definition of calculations
- It applies abstract classes