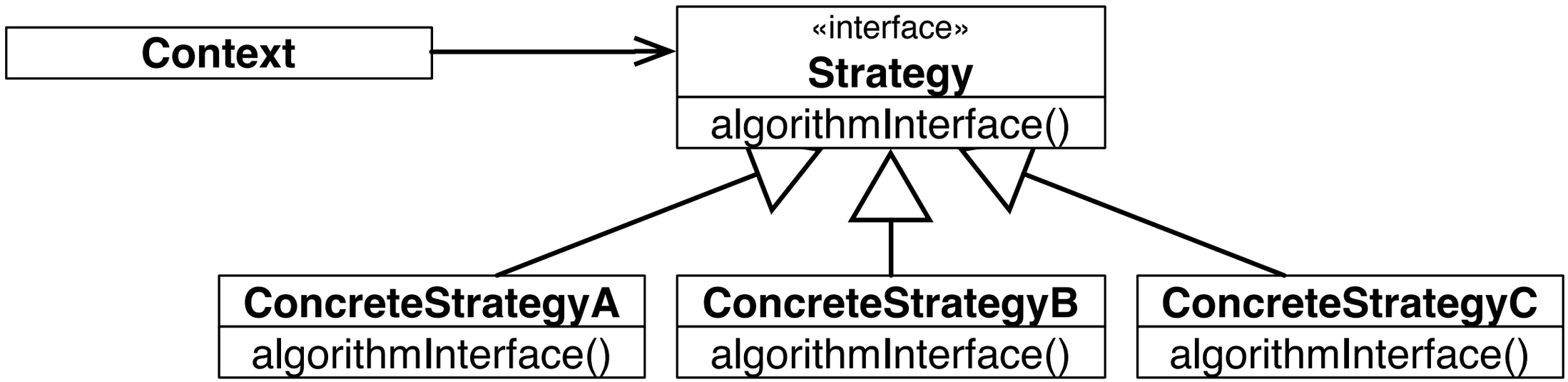
Winter Semester 16/17

Software Engineering Design & Construction

Dr. Michael Eichberg
Fachgebiet Softwaretechnik
Technische Universität Darmstadt

Strategy Pattern

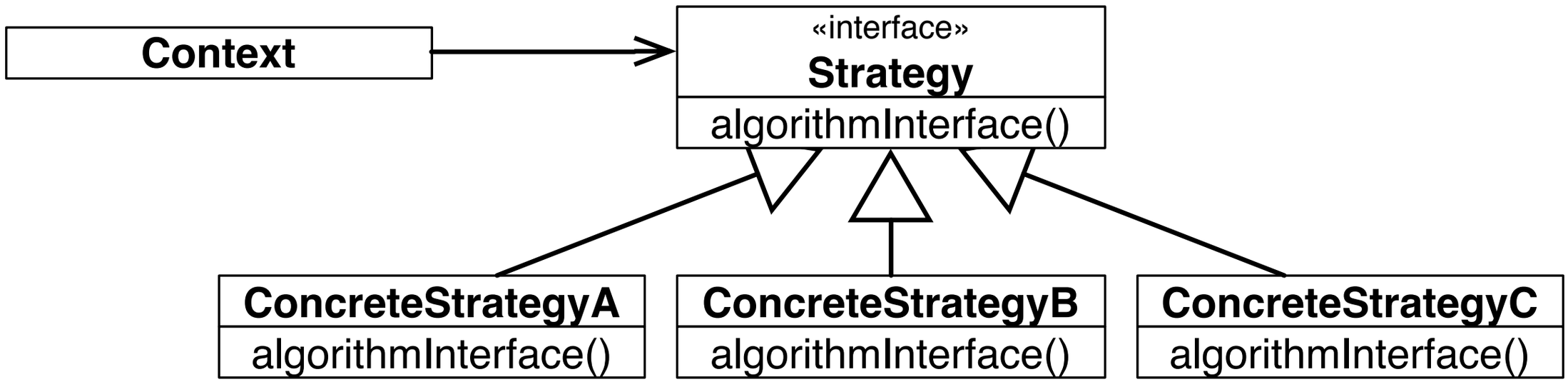
The Strategy Pattern in a Nutshell



Intent:

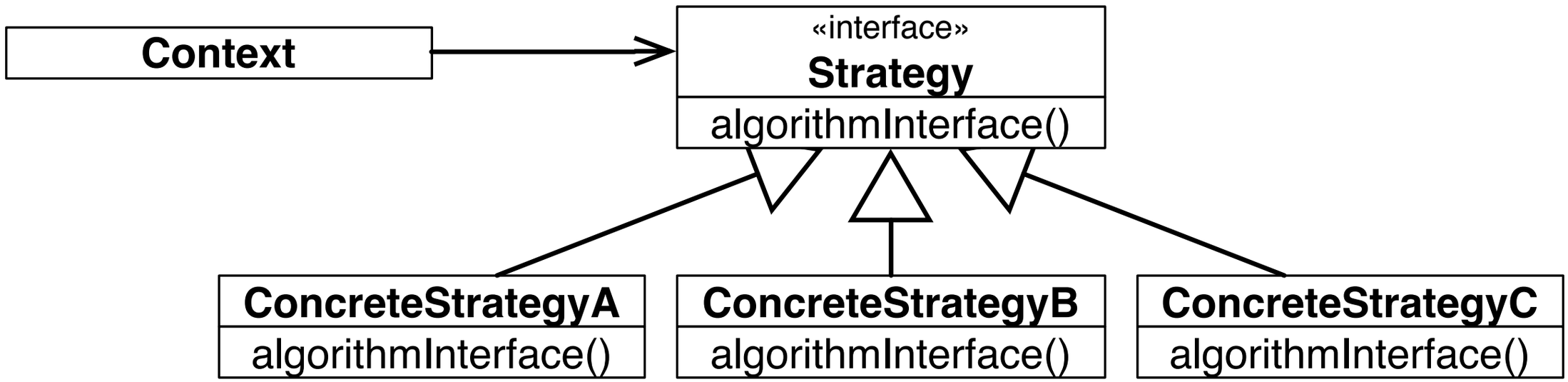
- Define a family of algorithms,
- Encapsulate each one,
- Make them interchangeable at runtime.

The Strategy Pattern in a Nutshell



Strategy lets the algorithm vary dynamically and independently from clients that use it.

When to Use the Strategy Pattern



- You need different variants of an algorithm.
- You need to select the variant of an algorithm dynamically.
- You need to be able to support the creation of new variants of the algorithm.

OCP

Strategy as an Alternative to Inheritance

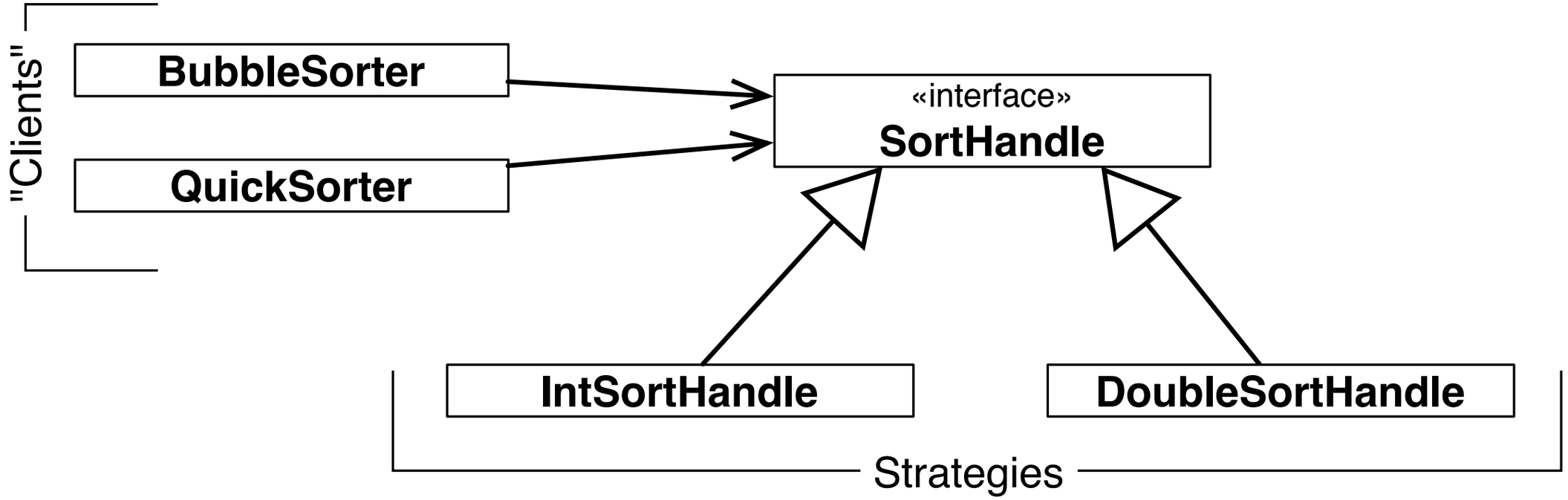
- The Strategy Pattern represents an alternative to modeling different algorithms (sub-behaviors) as subclasses of a usage Context.

- Inheritance mixes an algorithm‘s implementation with that of the Context. **The Context may become harder to understand, maintain, extend.**

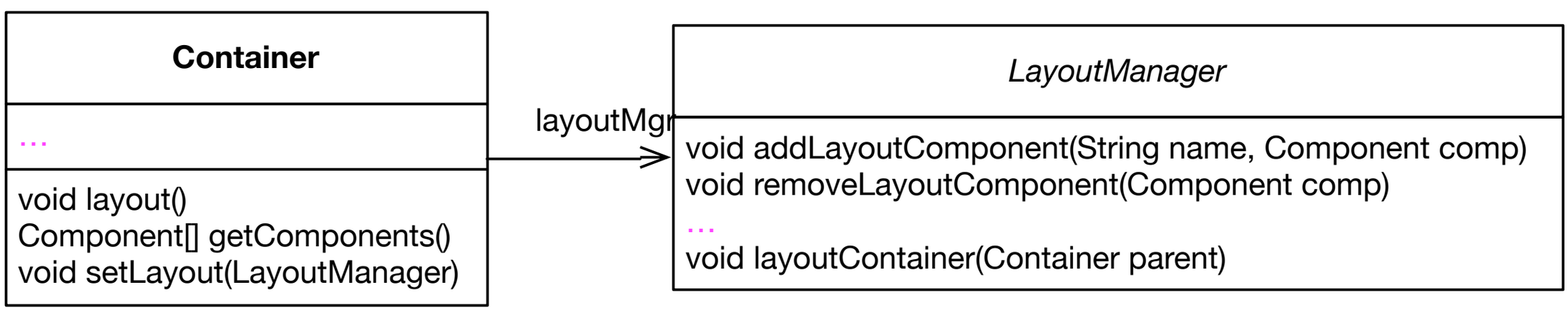
- Inheritance results in many related classes which only differ in the algorithm or behavior they employ.

- When using subclassing we cannot vary the algorithm dynamically.

Sorting Example with Strategy



Concrete Example: LayoutManager in Swing




```
class Container extends Component{
  LayoutManager layoutMgr;
  …
  public LayoutManager getLayout() {
    return layoutMgr;
  }

  public void layout() {
    layoutMgr.layoutContainer(this);
  }
  …
}
```

Functional Counterpart of Strategies

- One can look at the Strategy pattern as a style for emulating *first-class functions* available in functional programming languages.
- **Strategy objects** encapsulate sub-computations in first-class values that can be passed as parameters and returned as results of other computations (methods).

The Cost of the Strategy Pattern

There are trade-offs to be made to profit from the advantages of the Strategy pattern.

These trade-offs must be known and carefully considered when using the Strategy.

Footprint of Variations in Base Functionality

```
class Container extends Component{
  LayoutManager layoutMgr;

  public LayoutManager getLayout() {
    return layoutMgr;
  }

  public void layout() {
    layoutMgr.layoutContainer(this);
  }
  …
}
```

- The field `layoutMgr`
- Methods to manage strategy objects; e.g., `setLayout`
- Facade methods forwarding functionality to strategy, e.g., `layout`

Structural Variation is not Supported

- The Strategy interface must **fit the needs of all possible variations** of the outsourced feature.

- This may lead to bloated („One Size Fits All“) interfaces. The interfaces might be too complicated for some clients not interested in sophisticated variations of a feature.

- Careful anticipation of the needs of future variations is needed when designing the interface.

- Aggravates extensibility.

An Example „One Size Fits All“-Interface

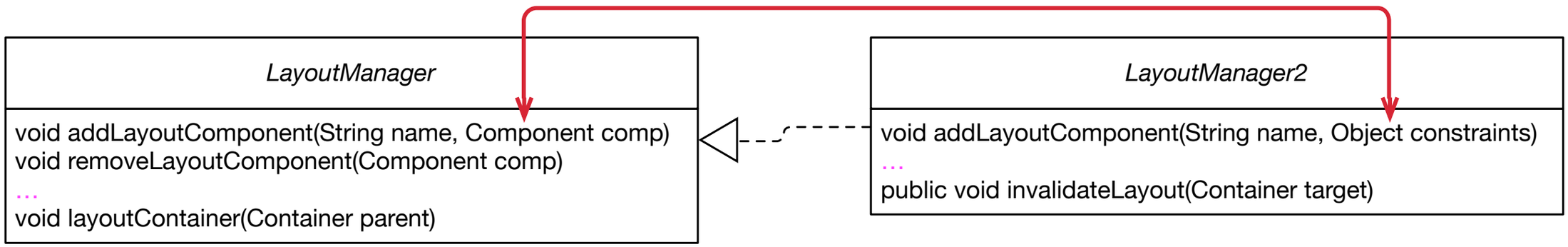
```
interface ListSelectionModel {
  int SINGLE_SELECTION = 0;
  int SINGLE_INTERVAL_SELECTION = 1;
  int MULTIPLE_INTERVAL_SELECTION = 2;

  /** …
   * In {@code SINGLE_SELECTION} selection mode,
   * this is equivalent to calling {@code setSelectionInterval},
   * and only the second index is used.
   * In {@code SINGLE_INTERVAL_SELECTION} selection mode,
   * this method behaves like {@code setSelectionInterval},
   * unless the given interval is immediately
   * adjacent to or overlaps the existing selection,
   * and can therefore be used to grow the selection.
   * …
   */
   void addSelectionInterval(int index0, int index1);

}
```

When the „One Size Fits All“-Interface Doesn't fit!

Example from Java Swing's `JComponent`




```
//javax.swing.JComponent - OpenJDK / 6-b14
1804  public float getAlignmentY() {
1805    float yAlign;
1806    if (layoutMgr instanceof LayoutManager2) {
1807      synchronized (getTreeLock()) {
1808        LayoutManager2 lm = (LayoutManager2) layoutMgr;
1809        yAlign = lm.getLayoutAlignmentY(this);
1810      }
1811    } else {
1812      yAlign = super.getAlignmentY();
1813    }
1814    return yAlign;
1815  }
```

Communication Overhead

- Some concrete strategies won't use all information passed to them.
  - Simple concrete strategies may use none of it.
  - Context creates/initializes parameters that never get used.
- If this is an issue, consider using a tighter coupling between `Strategy` and `Context`.
  Let Strategy know about Context.
  Two Ways of Strategy-Context Interaction:

1. Pass the needed information as a parameter.
   - Context and Strategy decoupled.
   - Interaction overhead.
   - Algorithm can’t be adapted to specific needs of context.
2. Context passes itself as a parameter or Strategy has a reference to its Context.
   - Reduced interaction overhead.
   - Context must define a more elaborate interface to its data.
   - Close(r) coupling of Strategy and Context.

Variations with Fixed Interface

Strategy objects are effective in modeling features of an object with dynamically `varying implementations but fixed interfaces`.

Increased Number of Objects

Potentially many strategy objects need to be instantiated.

To alleviate this problem you may use Stateless Strategies:

- The number of strategy objects can sometimes be reduced by stateless strategies that several Contexts can share.
- Any state is maintained by Context.
- Context passes the necessary state in each request to the Strategy object.
- (No / less coupling between Strategy implementations and Context.)
- Shared strategies should not maintain state across invocations.
- Such strategies are **Services**.

Composition of Multiple Variations

Strategy objects cannot be effectively used to model interdependent variations.

E.g., `JTable` uses Strategies related to cell rendering, which however may depend on other properties of the cell: `isSelected`, `isDropTarget`/`isDragSource`.

Such interdependencies between different variation dimensions cannot be properly modularized using strategy objects only.

Takeaway

The core of the Strategy Pattern is to model variability of object features by outsourcing the implementation of these features in “helper” (strategy) objects. Exploiting “implementation to interfaces” and subtype polymorphism for abstracting over variations of the outsourced feature.

The Strategy pattern addresses two problems of inheritance:

- Variations become reusable.
- Dynamic variations of features becomes possible.

Case Study: Ex- and Implicit Strategies

scala.collection.immutable.List

```
def sortWith(lt: (A, A) ⇒ Boolean): List[A]
```

Sorts this sequence according to a comparison function.

```
def sorted[B >: A](implicit ord: math.Ordering[B]): List[A]
```

Sorts this sequence according to an Ordering.

```
def sortBy[B](f: (A) ⇒ B)(implicit ord: math.Ordering[B]): List[A]
```

Sorts this Seq according to the Ordering which results from transforming an implicitly given Ordering with a transformation function.

Using `implicit` is very helpful if - in a given context - usually only one concrete Strategy object exists.

Filing the Design Space between Template and Strategy

Using mixin-composition and self-type annotations widens the design space.

```
trait Component

trait LayoutEngine { def layout(components: Array[Component]) }

trait BasicLayoutEngine extends LayoutEngine {
    def layout(components: Array[Component]) { /*Basic means nothing..*/ }
}
```

```
abstract class Container(private val components: Array[Component]) {
  this: LayoutEngine ⇒ // <= Self-type annotation
  def doLayout() { layout(components); }
}

object LayoutEngineDemo extends App {
    val c : Container = new Container(Array()) with BasicLayoutEngine
    //c.layout (won't compile, because C is only of type Container!)
    c.doLayout
    println(c)
}
```