The Strategy
Design Pattern

For details see Gamma et al. in “Design Patterns”
Supporting several kinds of external third-party services for calculating taxes.
Supporting several kinds of database connectors.
We want to be able to sort different kinds of values.
The Strategy Design Pattern

Intent & Example

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

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

General Structure

Define a family of algorithms, encapsulate each one, and make them interchangeable.
The Strategy Design Pattern

Strategy - An Alternative to Subclassing

• Subclassing Context mixes algorithm’s implementation with that of Context. Context harder to understand, maintain, extend.

• When using subclassing we can't vary the algorithm dynamically.

• Subclassing results in many related classes. They just differ in the algorithm or behavior they employ.

• Encapsulating the algorithm in Strategy...
  • lets you vary the algorithm independently of its context
  • makes it easier to switch, understand, reuse and extend the algorithm

If you would use subclassing instead of the Strategy Design Pattern.
Example - “The Strategy Pattern” in Java AWT/Swing

Client Code

```java
public class Container extends Component {
    ...
    /**
     * Sets the layout manager for this container.
     * @param mgr the specified layout manager
     */
    public void setLayout(LayoutManager mgr) {
        layoutMgr = mgr;
        invalidateIfValid();
    }

    /**
     * Causes this container to lay out its components. ...
     */
    public void doLayout() {
        LayoutManager layoutMgr = this.layoutMgr;
        if (layoutMgr != null) {
            layoutMgr.layoutContainer(this);
        }
    }
}
```

```java
java.awt.Container c = ...;
c.setLayout(new java.awt.BorderLayout());
```
The Strategy Design Pattern

When to use Strategy

• …many related classes differ only in their behavior rather than implementing different related abstractions
  Strategies allow to configure a class with one of many behaviors.
  
• …you need different variants of an algorithm
  Strategies can be used when variants of algorithms are implemented as a class hierarchy.

• …a class defines many behaviors that appear as multiple conditional statements in its operations
  Move *related conditional branches* into a strategy.
The Strategy Design Pattern
Things to Consider

• Clients must be aware of different strategies and how they differ, in order to select the appropriate one
• Clients might be exposed to implementation issues
• Use Strategy only when the behavior variation is relevant to clients
The Strategy Design Pattern

Things to Consider

• Optional Strategy objects
  • Context checks if it has a Strategy before accessing it...
    • If yes, Context uses it normally
    • If no, Context carries out default behavior
  • Benefit: clients don't have to deal with Strategy objects unless they don't like the default behavior
The Strategy Design Pattern

Things to Consider

• Increased number of (strategy) objects

• Sometimes can be reduced by **stateless strategies** that Contexts can share

• Any state is maintained by Context, passes it in for each request to the Strategy object (No / less coupling between Strategy implementations and Context.)

• Shared strategies should not maintain state across invocations (→ Services)
The Strategy Design Pattern - Implementation

The **Strategy** interface is shared by all Concrete Strategy classes whether the algorithms they implement are trivial or complex.

Some **ConcreteStrategies** won't use all the information passed to them (Simple ConcreteStrategies may use none of it.) (Context creates/initializes parameters that never get used.) If this is an issue use a tighter coupling between Strategy and Context; let Strategy know about Context.

**Communication Overhead**
The Strategy Design Pattern - Implementation

- Giving Strategy Visibility for the Context Information the Strategy needs; two possible strategies:

- **Pass the needed information as a parameter...**
  - Context and Strategy decoupled
  - Communication overhead
  - Algorithm can’t be adapted to specific needs of context

- **Context passes itself as a parameter or Strategy has a reference to its Context...**
  - Reduced communication overhead
  - Context must define a more elaborate interface to its data
  - Closer coupling of Strategy and Context
Using the strategy pattern, both - the template and the detailed implementations - depend on abstractions (interfaces).