Strategy Pattern

Goal:

Exchangeability of algorithms, unification of different algorithms under a common interface.
Examples:

- Algorithm for the layouting of pictures in a photo album
- Algorithm for a train timetable lookup (fastest, cheapest, fewest changes, etc.)
- Algorithm for solving a system of linear equations (diagonal dominant, stiff, etc.)
- Algorithm for computing the next move in a chess engine

Motivation:

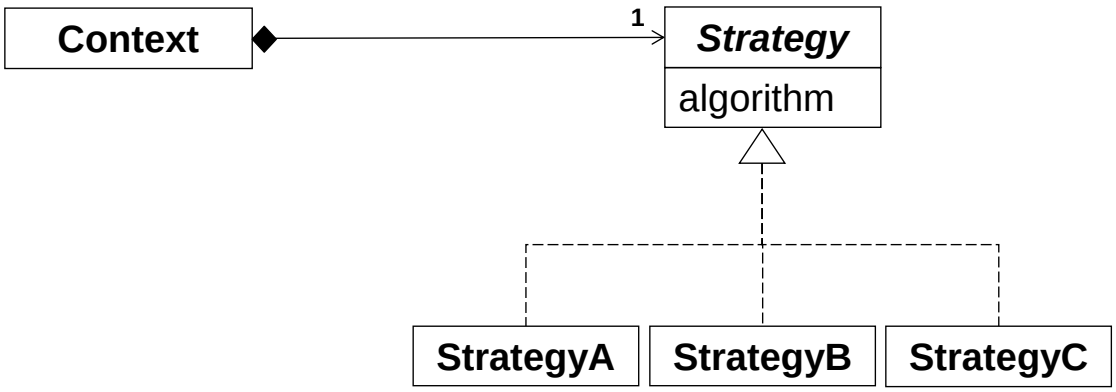
The algorithm to be executed is often stored in a flag defined in the context. This leads to if/switch statements as the following code snipped shows:

```
if (option == 'A') {
   /* Code for option A */
} else if (option == 'B') {
   /* Code for option B */
}
```

Disadvantages:

- No simple extensibility: If a new algorithm is to be added, then many places must be adapted (violation of the “open-closed” principle)!
- The different variants are not properly encapsulated; changes of one variant (e.g. different assumptions about the preconditions) may have effects on other variants (this issue will not be solved by the strategy pattern, but the dependencies will be made more explicit).

Structure:



- The context object uses a concrete strategy instance (this strategy instance is either passed to the context over the constructor or the context chooses a suitable strategy based on the current situation – in this case the context object would store several references).
- The interface of the strategy must be powerful enough to serve all imaginable algorithms (i.e. information needed by future algorithms must be provided over the interface).
- The context might define an interface over which the strategy can access additional data from the context. This interface can then either passed to the strategy whenever the algorithm is invoked, or such a reference could also be passed to the constructor of the strategy and stored in the strategy (then this strategy could only serve one context).

Example:

```
public class Date {
	int day, month, year;
	private final PrintDate p; // algorithm which defines how to format a date

	public Date(PrintDate p) { this.p = p; }

	public void print() {
		p.print(this);
	}

	public static void main(String[] args) throws Exception {
		PrintDate printer = (PrintDate) Class.forName(args[0])
					.getDeclaredConstructor().newInstance();
		Date d = new Date(printer);

		LocalDate today = LocalDate.now();
		d.day = today.getDayOfMonth();
		d.month = today.getMonthValue();
		d.year = today.getYear();

		d.print();
	}
}

public interface PrintDate {
	void print(Date d);
}

public class StdPrintDate implements PrintDate {
	@Override
	public void print(Date d) {
		System.out.println("Date: " + d.day + "." + d.month + "." + d.year);
	}
}

public class USPrintDate implements PrintDate {
	@Override
	public void print(Date d) {
		System.out.println("Date: " + d.month + "/" + d.day + "/" + d.year);
	}
}
```

Remarks:

- In the above example, the context is passed as a parameter to the strategy, i.e. the strategy has access to all information provided by the context
- Concrete implementations can be used by different contexts if they do not store context specific state (=> such strategies are called stateless).
- The strategy pattern is an alternative to subclassing, it also allows to specialize the behavior of the context class.
- Under certain conditions the interface might be very comprehensive (i.e. may have many parameters) in order to support all existing and future algorithms. Many concrete strategies then ignore many parameters.
- In order to prevent bulky interfaces (methods with countless parameters) a generic interface might be used with parameters of type Object or String, but this weakens type security.

Applications:

- As an architect / software designer you plan right from the start to use the strategy pattern. Then the challenge is to define a suitable interface which is as simple a possible but nevertheless as powerful as needed in order to serve all existing and imaginable strategies.
- Refactoring: as a programmer you dislike the if-else-cascade-code-smell. These can be eliminated with the strategy pattern. Here again: the definition of the interface might be challenging, in particular if not all strategies / states are foreseeable.
- Implementation of a given Strategy: The most frequent application of the strategy pattern is to implement a particular strategy interface defined by a software library (e.g. layout manager of the AWT/Swing framework).
- Application is advisable if:
    - Combination: Several classes only differ in some parts, but they contain behavior in common => extract this commonality as a context and define the differing parts in strategies.
    - Separation: If a class contains code in different branches of conditional (if / switch) statements which all switch on the same parameter, then all related branches should be moved into a concrete strategy.
    - Extensibility: If you have to support different variants of an algorithm (and if you want to add additional algorithms later to the context).

State versus Strategy:

The State and the Strategy pattern are very similar and it might be difficult to decide whether a particular application is a Strategy or a State pattern. Typically the following characteristics hold (although deviations from these rules are possible and even typical):

Strategy	State
• Represents an algorithm, interface typically contains a "compute" method	• Defines state-specific behavior, i.e. the behavior contained in a state object is specific to state of the associated context
• Strategy is typically set only once	• State changes are typical at run-time (whenever the state of the state machine changes)
• Strategy is chosen externally (`setStrategy`) or by the context (depending on parameters)	• State is set externally (`setState`) or by the state itself (`setNextState`), choice usually dependent on state of context object
• Strategy is typically not aware of other concrete strategies	• A concrete state may be aware of other concrete states (=>transition)
• Usually only one public method (and additional private methods)	• Usually several public methods for the state-specific behavior
• Strategy may contain algorithm specific state	• State usually contains no state but accesses state in context