# Strategy

A behavioral pattern

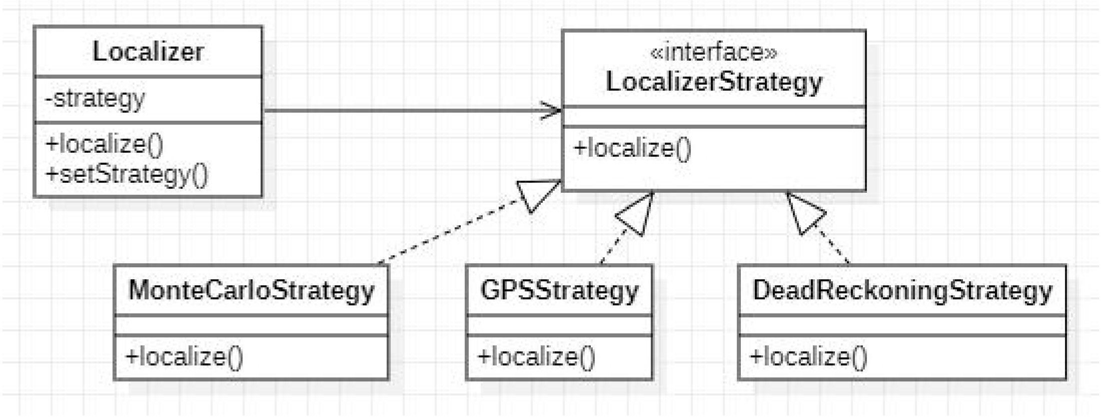
# Learning goals

1. Learn the idea, structure, and Java implementation of the Strategy design pattern.
2. Learn to apply the Strategy DP in your own programming.

# Idea of Strategy

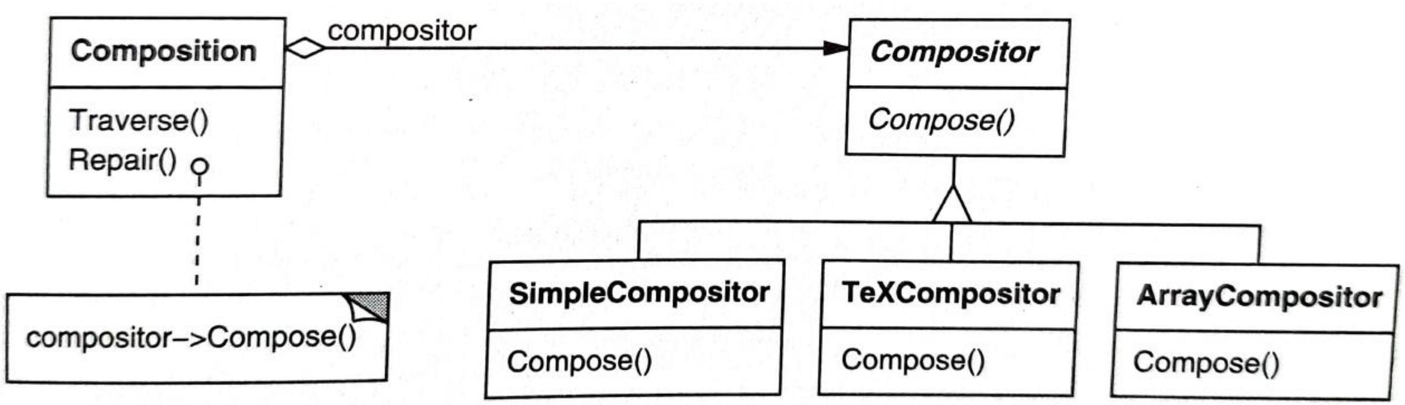
- Sometimes, an application may need to choose from several **alternative implementations** for observed behaviour of fixed type.
  - These implementations are called **algorithms**.
- Example: sorting an array (or list) can be done with BubbleSort, QuickSort, or MergeSort.
  - The observed behaviour is still the same: the structure becomes sorted.
- The Strategy DP provides a way to encapsulate the algorithms so that they are used via a simple ”front door”: a context object.
- The context knows the concrete strategy applied, but does not expose its implementation details.
- The selected strategy can be changed runtime, via the context.

# Example 1



- A robot or drone localizer may rely on three different algorithms:
  1. Map and sensor based localization (Monte Carlo)
  2. GPS localization
  3. Dead reckoning

# Example 2



- There are three strategies for managing line breaks in a file stream:
  1. manage one linebreak at a time (Simple strategy)
  2. optimize linebreaks over the paragraph in LaTeX style (TeX strategy)
  3. Put a fixed number of items on a row (Array strategy)

Image: Gamma et al., Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley Longman (1995), p. 315

# General structure

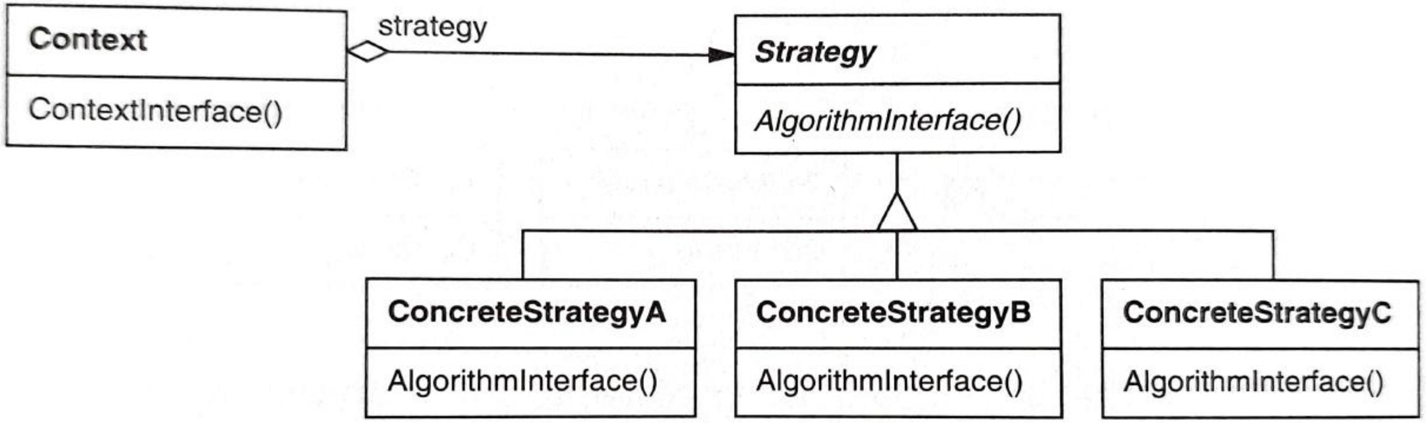


Image: Gamma et al., Design Patterns. Elements of Reusable Object-Oriented Software. Addison Wesley Longman (1995), p. 316

# Roles

- **Strategy**: declares a common interface for all available algorithms.
- **Concrete Strategy:** Implements the algorithm declared in the Strategy interface.
- **Context**: calls the Strategy interface. Maintains a reference to a Strategy object and is able to update that reference if the client so asks.

# Purpose of the Context class

- The Context is an intermediate class that **delegates** the method requests to the concrete strategy object.
- It provides a **uniform interface** for applying any of the strategies.
- The **client can stay agnostic** about the full set of available strategies.
- As the concrete strategies are not subclasses of the context, the Context **stays simple**.
- The **Context object stays the same** even if the concrete strategy changes runtime.

# Strategy vs. State

- The Strategy and State DPs look superficially the same. They have different use cases and characteristics, though:
- Usage:
  - **Strategy**: the algorithms may vary dynamically, but the observed behaviour still meets the same purpose
  - **State**: the observed behaviour usually changes.
- Context:
  - **Strategy**: The context is not bound to any of the algorithm implementations.
  - **State**: The context may be tightly coupled to states, and it can even manage the state changes.
- Dynamic Change:
  - **Strategy**: the Strategy change is initiated by the client.
  - **State**: a state is changed by the context or the states themselves.

# Practical issues

- The Strategy pattern makes it possible to avoid complex conditional statements in choosing a strategy.
  - That is, if/else or switch/state constructs
- The DP provides **a way to make alternative implementations** for a task, and switching between them is easy.
  - Example: one algorithm may be quick but require a lot of memory whereas another one can be slower but memory-efficient.
- Once the client has passed the chosen algorithm to the Context, the client uses the algorithm only by interacting with the Context.
- As a drawback, the clients must be aware of the available strategies.
  - Or, the strategy may be generated by a Factory class.
- The set of available methods in the Context class and the Strategy interface may vary dpending on the needs.
  - The context may need to pass data to the Strategy, and vice versa.
- Optionally, the Context may provide a **default implementation** that is implemented without a Strategy object.