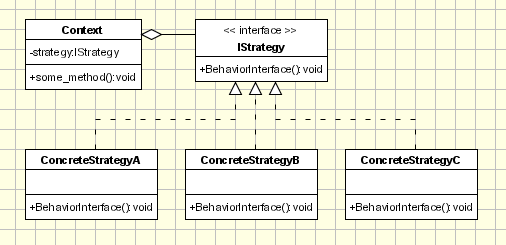
# Strategy

**Definition**: Defines a family of algorithms, encapsulates each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients who use it.

**Strategy**  lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.





## Relations with Other Patterns

* [**Bridge**](https://refactoring.guru/design-patterns/bridge), [**State**](https://refactoring.guru/design-patterns/state), [**Strategy**](https://refactoring.guru/design-patterns/strategy) (and to some degree [**Adapter**](https://refactoring.guru/design-patterns/adapter)) have very similar structures. Indeed, all of these patterns are based on composition, which is delegating work to other objects. However, they all solve different problems. A pattern isn’t just a recipe for structuring your code in a specific way. It can also communicate to other developers the problem the pattern solves.
* [**Command**](https://refactoring.guru/design-patterns/command) and [**Strategy**](https://refactoring.guru/design-patterns/strategy) may look similar because you can use both to parameterize an object with some action. However, they have very different intents.
  + You can use Command to convert any operation into an object. The operation’s parameters become fields of that object. The conversion lets you defer execution of the operation, queue it, store the history of commands, send commands to remote services, etc.
  + On the other hand, Strategy usually describes different ways of doing the same thing, letting you swap these algorithms within a single context class.
* [**Decorator**](https://refactoring.guru/design-patterns/decorator) lets you change the skin of an object, while [**Strategy**](https://refactoring.guru/design-patterns/strategy) lets you change the guts.
* [**Template Method**](https://refactoring.guru/design-patterns/template-method) is based on inheritance: it lets you alter parts of an algorithm by extending those parts in subclasses. [**Strategy**](https://refactoring.guru/design-patterns/strategy) is based on composition: you can alter parts of the object’s behavior by supplying it with different strategies that correspond to that behavior. Template Method works at the class level, so it’s static. Strategy works on the object level, letting you switch behaviors at runtime.
* [**State**](https://refactoring.guru/design-patterns/state) can be considered as an extension of [**Strategy**](https://refactoring.guru/design-patterns/strategy). Both patterns are based on composition: they change the behavior of the context by delegating some work to helper objects. Strategy makes these objects completely independent and unaware of each other. However, State doesn’t restrict dependencies between concrete states, letting them alter the state of the context at will.