**Стратегия** — это поведенческий паттерн проектирования, который определяет семейство схожих алгоритмов и помещает каждый из них в собственный класс, после чего алгоритмы можно взаимозаменять прямо во время исполнения программы.

Несколько жизненных примеров использования паттерна "Стратегия":

1. Платежные системы. В интернет-магазинах пользователи могут выбирать разные способы оплаты, такие как кредитные карты, PayPal или криптовалюты. Каждый способ реализует общий интерфейс, позволяя легко добавлять новые методы оплаты без изменения основного кода.

2. Сортировка данных. В приложениях, работающих с большими объемами данных, могут использоваться различные алгоритмы сортировки в зависимости от размера и структуры данных. Выбор алгоритма может производиться динамически на основе характеристик данных.

3. Навигационные приложения. В приложениях для навигации могут использоваться разные стратегии маршрутизации, такие как "самый быстрый путь", "самый короткий путь" или "безопасный путь". Пользователь может выбирать, какую стратегию использовать в зависимости от своих предпочтений.

4. Системы рекомендаций. В приложениях, предлагающих товары или контент (например, Netflix), могут использоваться разные стратегии рекомендаций: на основе предпочтений пользователя, популярности или анализа поведения.

Выше сказанное можно выделить, как «проблемы». А общее для всех «решение» выглядит следующим образом:

Вместо того, чтобы изначальный класс сам выполнял тот или иной алгоритм, он будет играть роль контекста, ссылаясь на одну из стратегий и делегируя ей выполнение работы. Чтобы сменить алгоритм, вам будет достаточно подставить в контекст другой объект-стратегию.

Важно, чтобы все стратегии имели общий интерфейс. Используя этот интерфейс, контекст будет независимым от конкретных классов стратегий. С другой стороны, вы сможете изменять и добавлять новые виды алгоритмов, не трогая код контекста. А как мы знаем, Любое изменение, будь то исправление багов или добавление нового алгоритма, затрагивает основной класс. Это повышает риск сделать ошибку, случайно задев остальной работающий код.

Структура паттерна (без привязки к какому-либо языку программирования):

1. Контекст хранит ссылку на объект конкретной стратегии, работая с ним через общий интерфейс стратегий.
2. Стратегия определяет интерфейс, общий для всех вариаций алгоритма. Контекст использует этот интерфейс для вызова алгоритма. Для контекста неважно, какая именно вариация алгоритма будет выбрана, так как все они имеют одинаковый интерфейс.
3. Конкретные стратегии реализуют различные вариации алгоритма.
4. Во время выполнения программы контекст получает вызовы от клиента и делегирует их объекту конкретной стратегии.
5. Клиент должен создать объект конкретной стратегии и передать его в конструктор контекста. Кроме этого, клиент должен иметь возможность заменить стратегию на лету, используя сеттер. Благодаря этому, контекст не будет знать о том, какая именно стратегия сейчас выбрана.

Шаги реализации:

1. Определите алгоритм, который подвержен частым изменениям. Также подойдёт алгоритм, имеющий несколько вариаций, которые выбираются во время выполнения программы.
2. Создайте интерфейс стратегий, описывающий этот алгоритм. Он должен быть общим для всех вариантов алгоритма.
3. Поместите вариации алгоритма в собственные классы, которые реализуют этот интерфейс.
4. В классе контекста создайте поле для хранения ссылки на текущий объект-стратегию, а также метод для её изменения. Убедитесь в том, что контекст работает с этим объектом только через общий интерфейс стратегий.
5. Клиенты контекста должны подавать в него соответствующий объект-стратегию, когда хотят, чтобы контекст вёл себя определённым образом.

Отношения с другими паттернами:

1. [Состояние](https://refactoring.guru/ru/design-patterns/state) можно рассматривать как надстройку над [Стратегией](https://refactoring.guru/ru/design-patterns/strategy). Оба паттерна используют композицию, чтобы менять поведение основного объекта, делегируя работу вложенным объектам-помощникам. Однако в *Стратегии* эти объекты не знают друг о друге и никак не связаны. В *Состоянии* сами конкретные состояния могут переключать контекст.
2. [Команда](https://refactoring.guru/ru/design-patterns/command) и [Стратегия](https://refactoring.guru/ru/design-patterns/strategy) похожи по духу, но отличаются масштабом и применением:
   1. Команду используют, чтобы превратить любые разнородные действия в объекты. Параметры операции превращаются в поля объекта. Этот объект теперь можно логировать, хранить в истории для отмены, передавать во внешние сервисы и так далее.
   2. С другой стороны, Стратегия описывает разные способы произвести одно и то же действие, позволяя взаимозаменять эти способы в каком-то объекте контекста.

Паттерн "Стратегия" позволяет создавать гибкие и расширяемые системы, которые легко адаптируются к изменяющимся требованиям и обстоятельствам. Это делает его особенно полезным в различных областях, где требуется возможность выбора между несколькими алгоритмами или подходами.