Алгоритмы и вариативность: паттерны Стратегия, Шаблонный метод и Состояние

Автор: Мокобиа Джейн Чидима,

группа 24.Б83

1. Введение: вариативность алгоритмов

В программной инженерии вариативность означает возможность изменять части алгоритма без необходимости переписывать всю систему.
Управление вариативностью делает код более гибким, повторно используемым и удобным для сопровождения.

Чтобы эффективно управлять такими случаями, используют три классических паттерна проектирования:

1. Стратегия — позволяет менять алгоритм во время выполнения.
2. Шаблонный метод — фиксирует общую структуру, но позволяет изменять шаги.
3. Состояние — изменяет поведение объекта в зависимости от его внутреннего состояния.

2. Паттерн Стратегия

Определение:
Паттерн *Стратегия* определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми.
Он позволяет изменять алгоритм независимо от объекта, который его использует.

Когда применять:

- Когда есть несколько способов выполнить одну и ту же задачу.
- Когда нужно избавиться от длинных цепочек условий if/elif.

Преимущества:

- Легко добавлять новые стратегии.
- Уменьшает дублирование кода.
- Соответствует принципу открытости/закрытости.

Недостатки:

- Увеличивает количество классов и объектов.
- Клиент должен знать, какую стратегию выбрать.

```
from abc import ABC, abstractmethod
from typing import Iterable

class Strategy(ABC):

    @abstractmethod
    def execute(self, data: Iterable[int]) -> list[int]:
        pass

class SortAscending(Strategy):
    def execute(self, data: Iterable[int]) -> list[int]:
        return sorted(data)

class SortDescending(Strategy):
    def execute(self, data: Iterable[int]) -> list[int]:
        return sorted(data, reverse=True)

class UniqueThenSort(Strategy):
    def execute(self, data: Iterable[int]) -> list[int]:
        return sorted(set(data))

class DataProcessor:

    def __init__(self, strategy: Strategy):
        self._strategy = strategy

    @property
    def strategy(self) -> Strategy:
        return self._strategy

    @strategy.setter
    def strategy(self, strategy: Strategy):
        self._strategy = strategy

    def process(self, data: Iterable[int]) -> list[int]:
        print(f"[Context] using strategy:
{self._strategy.__class__.__name__}")
        return self._strategy.execute(data)


data = [5, 3, 5, 1, 2, 9, 1]

ctx = DataProcessor(SortAscending())
print(ctx.process(data))
```

```
ctx.strategy = SortDescending()
print(ctx.process(data))

ctx.strategy = UniqueThenSort()
print(ctx.process(data))
```

```
[Context] using strategy: SortAscending
[1, 1, 2, 3, 5, 5, 9]
[Context] using strategy: SortDescending
[9, 5, 5, 3, 2, 1, 1]
[Context] using strategy: UniqueThenSort
[1, 2, 3, 5, 9]
```

Класс DataProcessor делегирует выполнение алгоритма объекту Strategy.
Мы можем в любой момент заменить стратегию сортировки (`SortAscending`, SortDescending, `UniqueThenSort`) без изменения логики контекста.
Таким образом, Стратегия позволяет легко менять поведение программы во время выполнения.

in short:
Паттерн *Стратегия* отделяет алгоритм от контекста, делая выбор и замену алгоритмов гибкими и безопасными.

3. Паттерн Шаблонный метод

Определение:
Паттерн *Шаблонный метод* определяет общий порядок выполнения алгоритма в базовом классе
и позволяет подклассам переопределять отдельные шаги, не изменяя саму структуру.

Когда применять:

- Когда несколько классов выполняют один и тот же процесс, но с небольшими отличиями.
- Когда важно сохранить общий порядок шагов.
- Когда некоторые шаги должны выполняться всегда (например, проверка данных).

Преимущества:

- Гарантирует единый порядок действий.
- Сокращает дублирование кода.
- Позволяет переопределять только необходимые шаги.

Недостатки:

- Основан на наследовании, что снижает гибкость.
- Нельзя менять шаги во время выполнения.

```python
from abc import ABC, abstractmethod
from time import perf_counter

class DataPipeline(ABC):

    def run(self, raw: str) -> list[str]:
        t0 = perf_counter()
        data = self.load(raw)
        data = self.clean(data)
        tokens = self.tokenize(data)
        result = self.postprocess(tokens)
        t1 = perf_counter()
        print(f"[Pipeline] completed in  {t1 - t0:.6f} sec")
        return result


    @abstractmethod
    def load(self, raw: str) -> str: ...
    def clean(self, s: str) -> str:
        return s.strip()
    @abstractmethod
    def tokenize(self, s: str) -> list[str]: ...
    def postprocess(self, tokens: list[str]) -> list[str]:
        return tokens

class CsvPipeline(DataPipeline):
    def load(self, raw: str) -> str:
        return raw.replace(",", ";")
    def tokenize(self, s: str) -> list[str]:
        return [t for t in s.split(";") if t]

class TextPipeline(DataPipeline):
    def load(self, raw: str) -> str:
        return raw.lower()
    def clean(self, s: str) -> str:
        return " ".join(s.split())
    def tokenize(self, s: str) -> list[str]:
        return s.split(" ")
    def postprocess(self, tokens: list[str]) -> list[str]:
        return sorted(set(tokens))


raw_csv = "A,B,,C"
raw_txt = "  The   quick   brown fox  jumps  "

print("CSV:", CsvPipeline().run(raw_csv))
print("Текст:", TextPipeline().run(raw_txt))
```

```
[Pipeline] completed in  0.000011 sec
CSV: ['A', 'B', 'C']
[Pipeline] completed in  0.000021 sec
Текст: ['brown', 'fox', 'jumps', 'quick', 'the']
```

Метод run() задаёт общий порядок шагов обработки данных, а подклассы переопределяют отдельные этапы.
CsvPipeline заменяет запятые и делит строку по ;, а TextPipeline очищает пробелы, переводит текст в нижний регистр и убирает повторы.
Благодаря этому сохраняется единая структура, но детали можно адаптировать под нужный формат.

in short: Паттерн *Шаблонный метод* обеспечивает общую структуру алгоритма и позволяет изменять отдельные шаги в подклассах.

4. Паттерн Состояние

Определение:
Паттерн *Состояние* позволяет объекту изменять своё поведение при изменении внутреннего состояния.
С точки зрения клиента, объект будто меняет свой класс.

Когда применять:

- Когда поведение объекта зависит от текущего состояния.
- Когда нужно избавиться от длинных конструкций if/else.
- Когда объект проходит последовательность стадий (конечный автомат).

Преимущества:

- Код становится чище и понятнее.
- Легко добавлять новые состояния.
- Поведение каждого состояния изолировано в отдельном классе.

Недостатки:

- Увеличивается количество классов.
- Переходы между состояниями нужно проектировать аккуратно.

```
from abc import ABC, abstractmethod

class Player:

    def __init__(self) -> None:
```

```
        self.state: State = Stopped(self)
        self.volume = 5

    def set_state(self, state: 'State') -> None:
        print(f"[Player] change of state:
{self.state.__class__.__name__} -> {state.__class__.__name__}")
        self.state = state

    # Действия пользователя
    def press_play_pause(self) -> None:
        self.state.on_play_pause()
    def press_stop(self) -> None:
        self.state.on_stop()
    def volume_up(self) -> None:
        self.state.on_volume(+1)
    def volume_down(self) -> None:
        self.state.on_volume(-1)

class State(ABC):
    def __init__(self, player: Player) -> None:
        self.player = player

    @abstractmethod
    def on_play_pause(self) -> None: ...
    @abstractmethod
    def on_stop(self) -> None: ...
    def on_volume(self, delta: int) -> None:
        self.player.volume = max(0, min(10, self.player.volume +
delta))
        print(f"[Volume] {self.player.volume}")

class Playing(State):
    def on_play_pause(self) -> None:
        print("Pause")
        self.player.set_state(Paused(self.player))
    def on_stop(self) -> None:
        print("stop")
        self.player.set_state(Stopped(self.player))

class Paused(State):
    def on_play_pause(self) -> None:
        print("continue")
        self.player.set_state(Playing(self.player))
    def on_stop(self) -> None:
        print("stop")
        self.player.set_state(Stopped(self.player))

class Stopped(State):
    def on_play_pause(self) -> None:
        print("start")
```

```
        self.player.set_state(Playing(self.player))
    def on_stop(self) -> None:
        print("already stopped")


p = Player()
p.press_play_pause()  # Stopped -> Playing
p.volume_up()
p.press_play_pause()  # Playing -> Paused
p.press_stop()        # Paused -> Stopped
```

```
start
[Player] change of state: Stopped -> Playing
[Volume] 6
Pause
[Player] change of state: Playing -> Paused
stop
[Player] change of state: Paused -> Stopped
```

Каждое состояние (`Playing`, Paused, `Stopped`) реализует собственное поведение.
При переходе из одного состояния в другое объект Player автоматически меняет реакцию на действия пользователя.
Это исключает использование громоздких условных операторов.

in short
Паттерн *Состояние* позволяет динамически изменять поведение объекта, делая код логичным и структурированным.

5. Сравнение и выводы

Характеристика	Стратегия	Шаблонный метод	Состояние
Что изменяется	Весь алгоритм	Отдельные шаги	Поведение по состоянию
Тип связи	Композиция	Наследование	Композиция
Изменения во время работы	Да	Нет	Да

Общий вывод:

- Стратегия — для подмены алгоритмов.
- Шаблонный метод — для единой структуры с изменяемыми шагами.
- Состояние — для логики, зависящей от режима работы объекта.

Все три паттерна помогают управлять вариативностью алгоритмов и повышают гибкость программных систем.