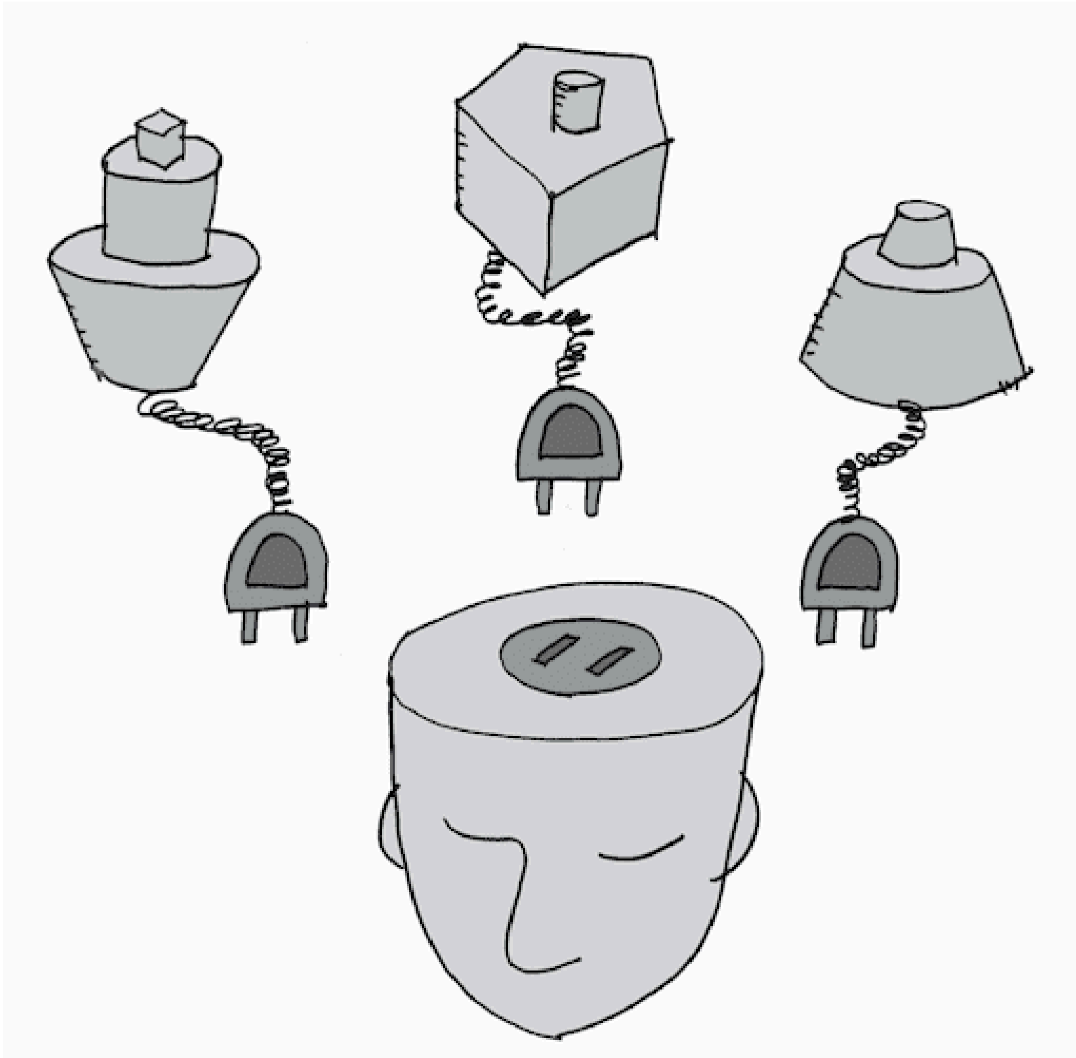
Strategy Pattern

Change Algorithm Dynamically



Strategy Pattern

When we need to go to **New York**, we can choose different **strategies**:

- **Fly** (expensive but fast)
- **Drive** (cheap but slow)
- **Train** (moderate cost and speed)

We can `dynamically` change the strategy based on circumstances.
The trip planner (context) remains the same, only the transportation strategy changes.

The Problem

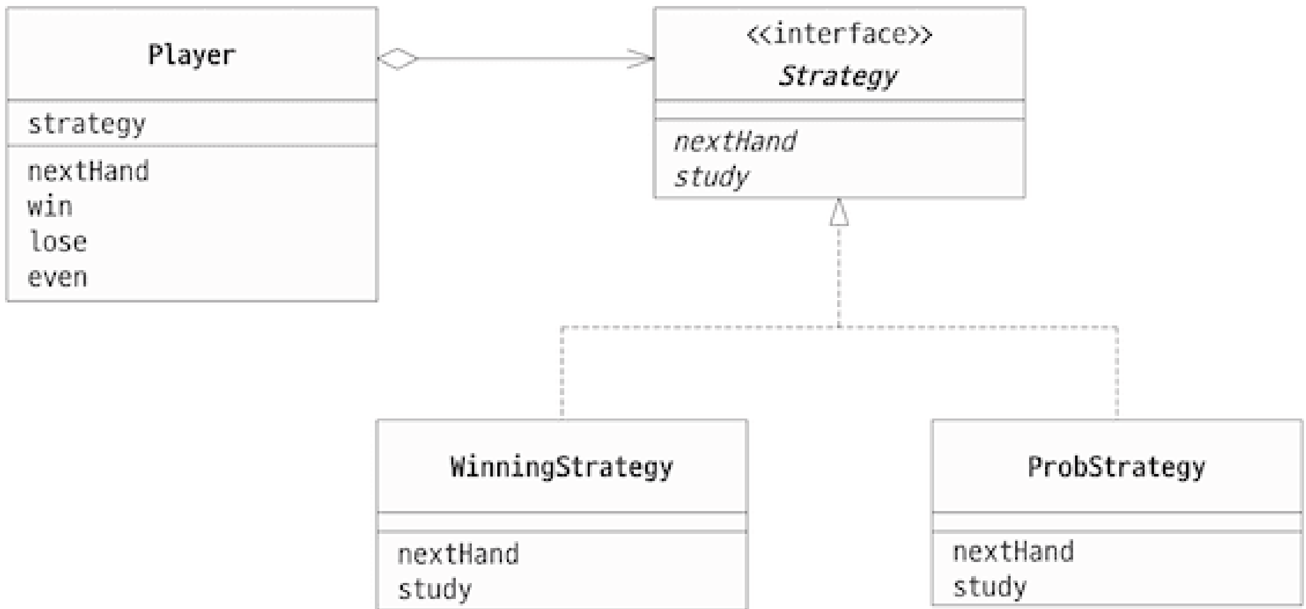
- We have **players** who need to choose <u>hands</u> in a Rock-Paper-Scissors game.
- Different **players** want to use different <u>strategies</u> for choosing hands.

The challenge: how to make **players** use different **algorithms** without changing the **Player** class itself?

The *Strategy* as the Solution

- We have an abstraction *Strategy* that defines how to choose the next hand.
- We do not need to `modify` the **Player** class, we only need to `provide` different **strategy implementations**.

The Solution (Design)



The Player owns a variety of strategies to win the game.

Step 1: Understand the Players

In this design, we have players:

- **Context** (Player)
- *Strategy* (abstract strategy interface)
  - **ConcreteStrategy** (WinningStrategy, ProbStrategy)
  - <u>Target</u> (Hand)

Step 2: The Player owns Strategies

- We need to see the `delegation` (the aggregation line) between **Context (Player)** and *Strategy Abstraction*.
- Notice that the **Context** `uses` the *Strategy* through aggregation.
    - It is as if we hire a **consultant** with different **expertise** for each project.

Step 3: Understand abstractions

- We have a *Strategy* that `defines` how to choose the next hand.
- In short, the **Player (Context)** delegates the algorithm to the *Strategy*.
  - The **Player (Context)** doesn't know about specific **algorithm implementations**.

Step 4: Understand concretion

- We have **WinningStrategy** and **ProbStrategy** that provide different algorithms for choosing hands.
    - **ConcreteStrategy** (WinningStrategy, ProbStrategy)
    - <u>Target</u> (Hand - what strategies operate on)

Code

- Main Method
- Context (Player)
- Strategy Classes

Main Method

```python
from base_classes import Player
from rock_strategy import AlwaysRockStrategy
from paper_strategy import AlwaysPaperStrategy
from random_strategy import RandomStrategy

def main():
    alice = Player("Alice", AlwaysRockStrategy())
    bob = Player("Bob", AlwaysPaperStrategy())
    charlie = Player("Charlie", RandomStrategy())

    players = [alice, bob, charlie]

    for round in range(3):
        print(f"\nRound {round + 1}:")
        for player in players:
            hand = player.play()
            print(f"  {player.name} plays {hand}")

    alice.set_strategy(RandomStrategy())
```

Step 1: Create context with different strategies

```
alice = Player("Alice", AlwaysRockStrategy())
bob = Player("Bob", AlwaysPaperStrategy())
charlie = Player("Charlie", RandomStrategy())
```

- Each **Player** (context) is configured with a different **strategy**.

Step 2: Use the context

```
players = [alice, bob, charlie]

for round in range(3):
    print(f"\nRound {round + 1}:")
    for player in players:
        hand = player.play()
        print(f"  {player.name} plays {hand}")
```

- Each **Player** uses the **algorithm**.
- This demonstrates `polymorphism` through strategy selection.

Abstarctions

ROCK/PAPER/SCISSORS as the Hand object:

```python
class Hand:
    """Simple hand representation"""
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return self.name


# Pre-made hands for easy use
ROCK = Hand("Rock")
PAPER = Hand("Paper")
SCISSORS = Hand("Scissors")
```

The Player:

```python
class Player:
    """Context class - uses strategy to choose hands"""
    def __init__(self, name, strategy):
        self.name = name
        self.strategy = strategy

    def play(self):
        """Ask strategy to choose a hand"""
        return self.strategy.choose_hand()

    def set_strategy(self, strategy):
        """Change strategy at runtime"""
        self.strategy = strategy

    def get_strategy_name(self):
        return self.strategy.__class__.__name__
```

The Strategy:

```
class Strategy:
    """Base strategy class"""
    def choose_hand(self):
        """Choose a hand - implemented by concrete strategies"""
        pass
```

Strategy Algorithms

PaperStrategy: Slways Paper

```python
from base_classes import Strategy, PAPER

class AlwaysPaperStrategy(Strategy):
    def choose_hand(self):
        return PAPER
```

RandomStrategy:

```python
class RandomStrategy(Strategy):
    def __init__(self):
        self.hands = [ROCK, PAPER, SCISSORS]
    def choose_hand(self):
        return random.choice(self.hands)
```

Discussion

Key Benefits

1. **Flexibility**: Easy to switch algorithms at runtime
2. **Extensibility**: Easy to add new strategies without changing context
3. **Isolation**: Algorithms are isolated in separate classes
4. **Testability**: Each strategy can be tested independently

Potential Drawbacks

- **Increased number of objects** - many strategy objects
- **Client complexity** - clients must understand different strategies
- **Communication overhead** - context and strategy must share data

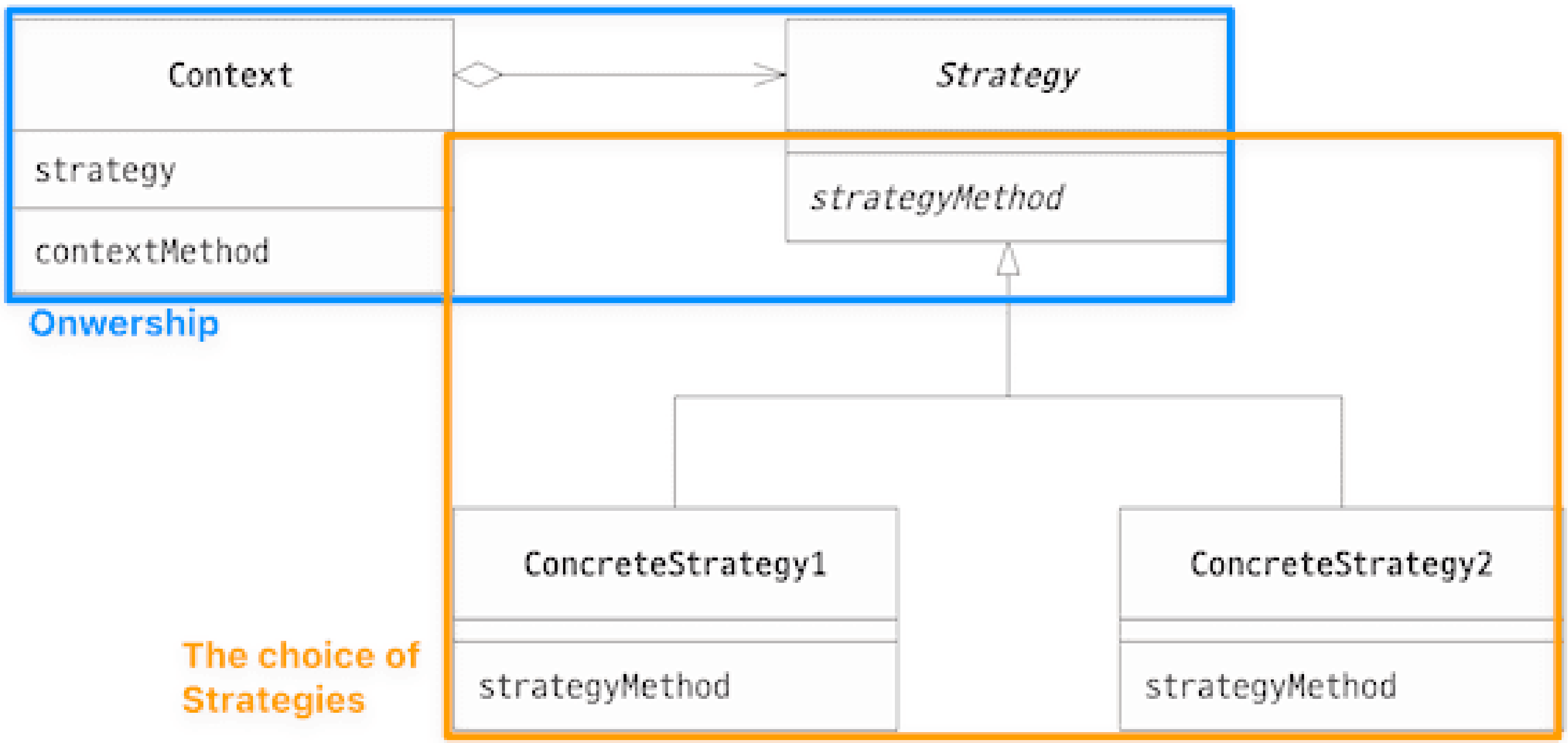
When to Use Strategy

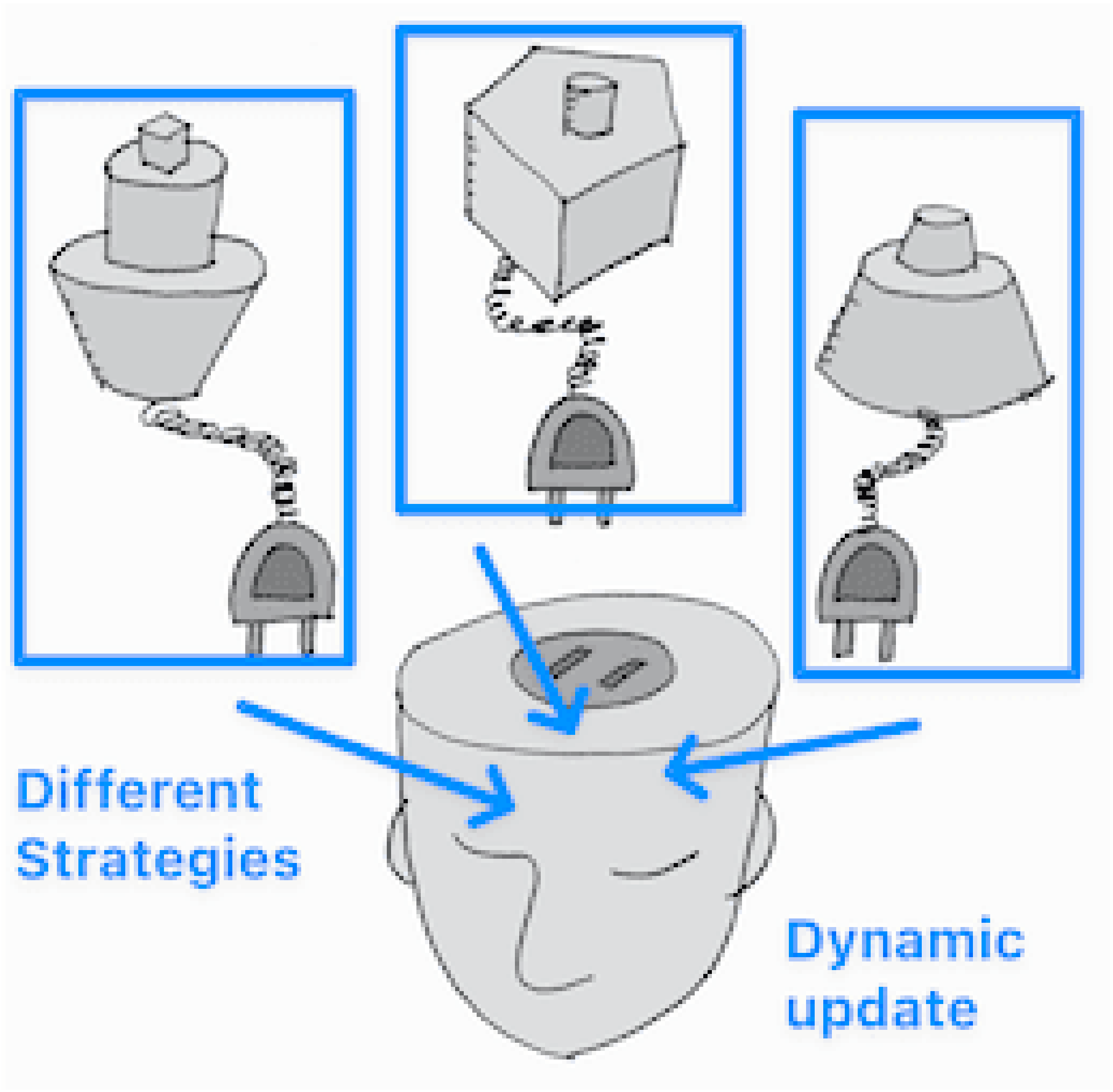
- When you have multiple ways to perform a task
- When you want to avoid conditional statements for algorithm selection
- When algorithms need to be swapped at runtime
- When you want to hide algorithm complexity from clients

Related Patterns

- **Bridge**: Strategy focuses on algorithms, Bridge separates interface from implementation
- **Template Method**: Strategy uses composition (aggregation), Template Method uses inheritance
- **Abstract Factory**: Can be used to create different families of strategies

UML


Context
strategy
contextMethod
Strategy
strategyMethod
Onwership
ConcreteStrategy1
strategyMethod
ConcreteStrategy2
strategyMethod
The choice of Strategies


Different
Strategies
Dynamic
update