The Strategy Design Pattern

This pattern is one of the most commonly used patterns in Object Oriented Programming

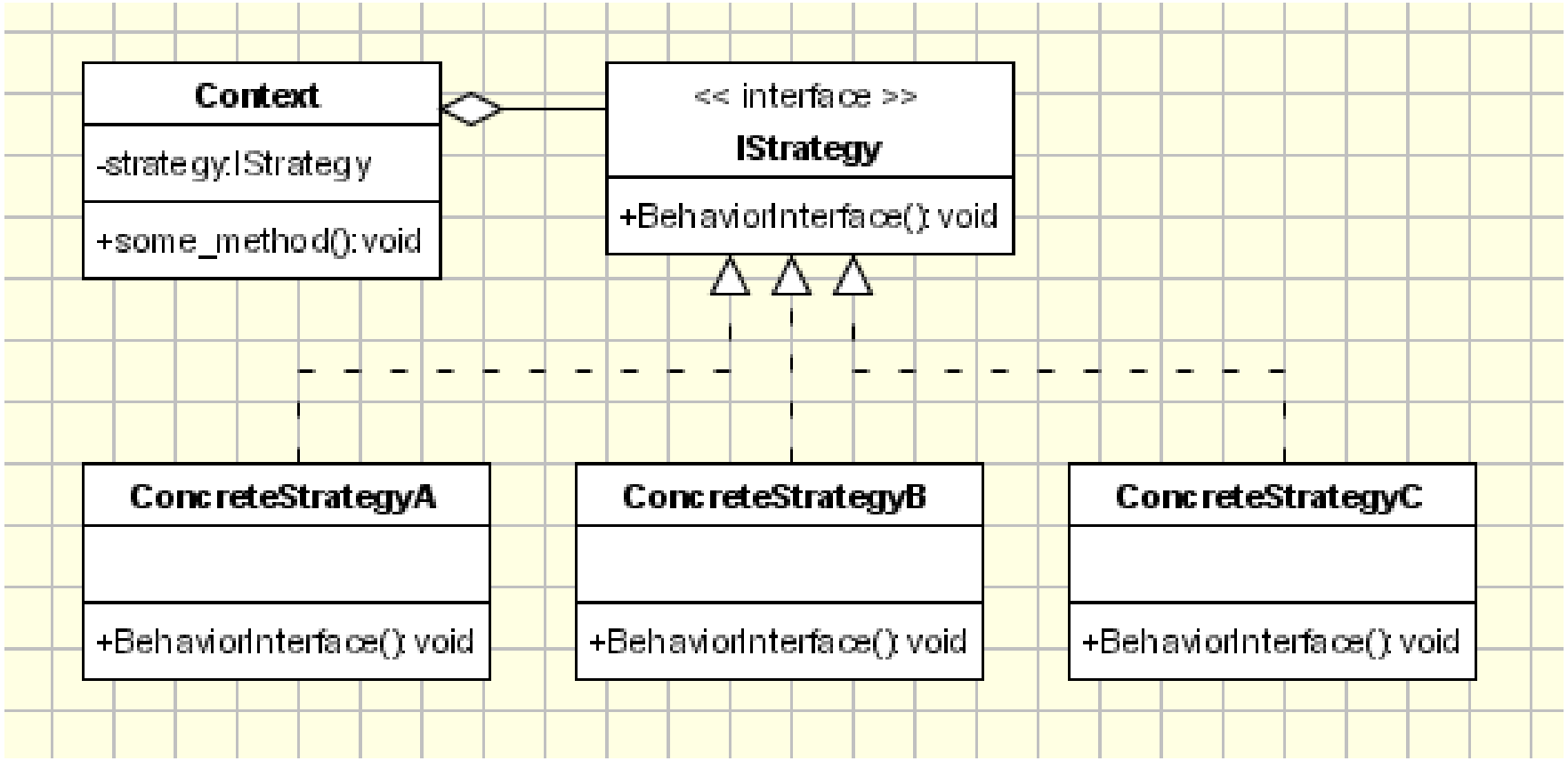
Definition

- Defines a family of algorithms, encapsulates each one, and makes them interchangeable. It lets the algorithms vary independently from the clients that use it.
- From Wikipedia: A pattern whereby algorithms can be selected at runtime.
  - https://en.wikipedia.org/wiki/Strategy_pattern
- It is a behavioral pattern in that it allows you to change the behavior of an algorithm dynamically.

Motivation

- There are common situations when classes differ only in their behavior
- For these cases it is a good idea to isolate the algorithms in separate classes in order to have the ability to select different algorithms at runtime
- Flexibility to choose a specific version of an algorithm based on the current things happening in your program is incredibly useful

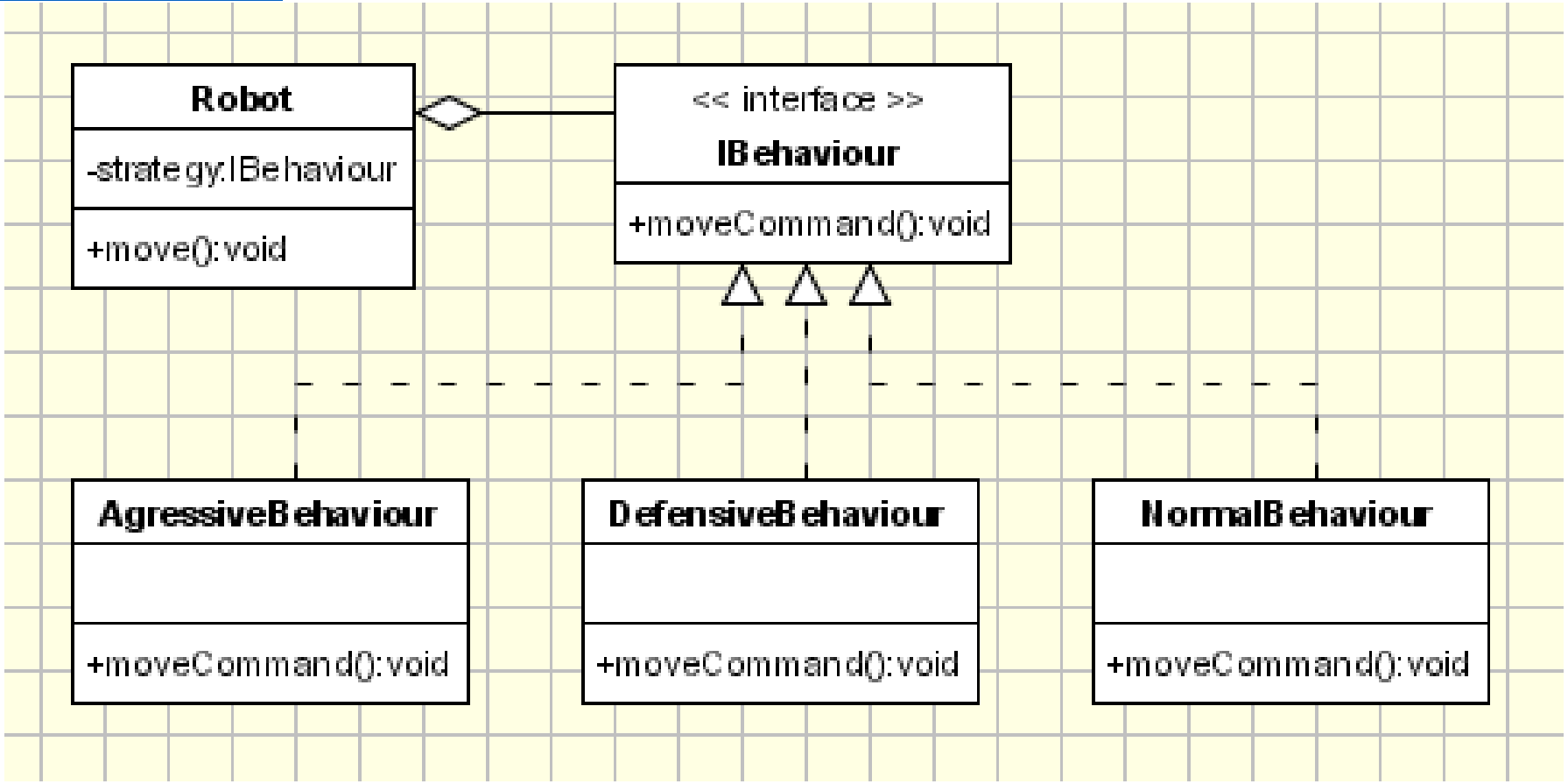
Strategy UML (https://www.oodesign.com/strategy-pattern.html)



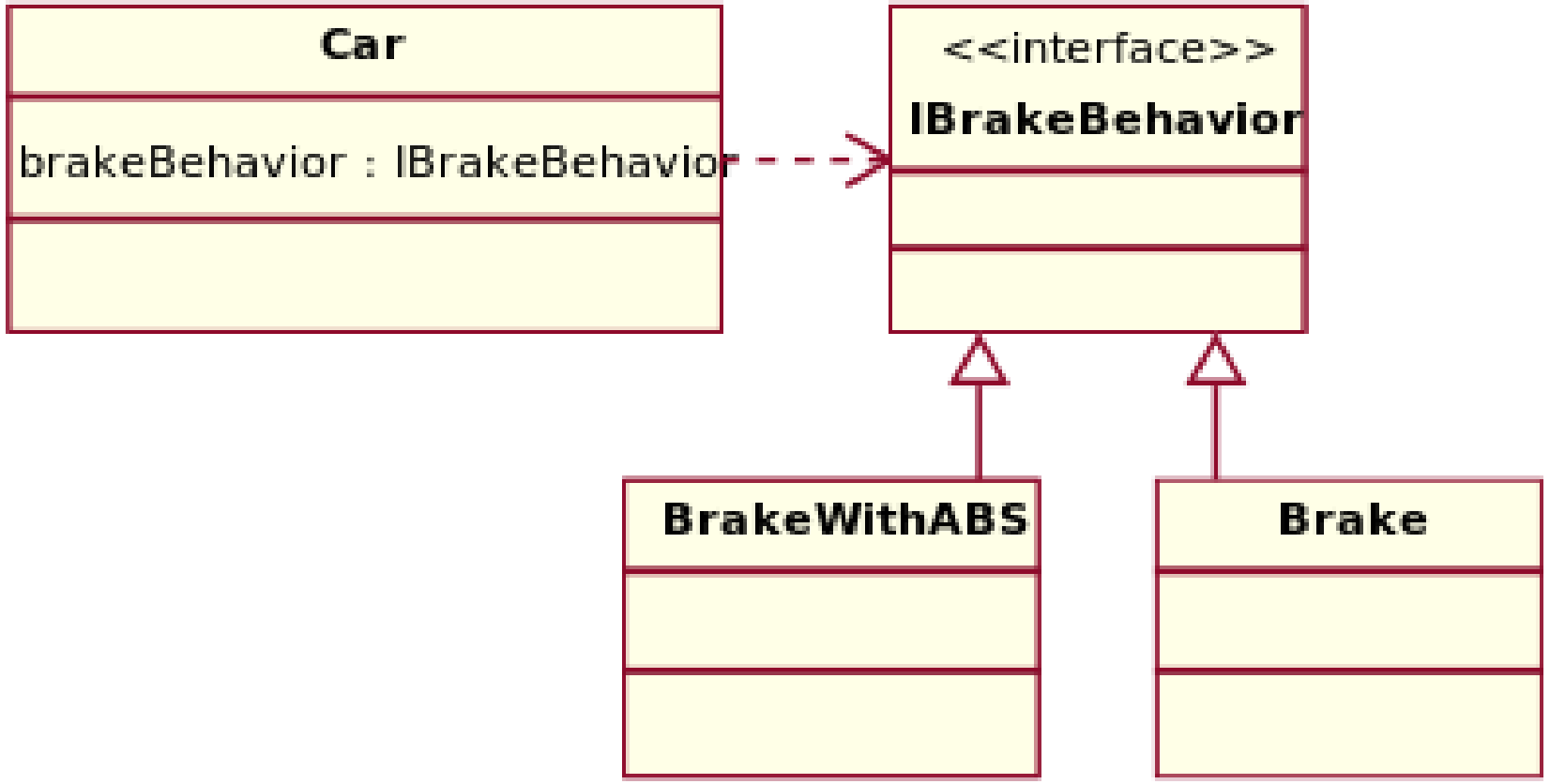
Example

- A class that performs validation on incoming data may use a strategy pattern to select a validation algorithm based on the type of data, the source of the data, user choice, and/or other discriminating factors
- These factors are not known for each case until runtime, and may require radically different validation to be performed
- The validation strategies, encapsulated separately from the validating object, may be used by other validating objects in different areas of the system (or even different systems) without code duplication

UML Specific Example (https://www.oodesign.com/strategy-pattern.html)



Another UML Example (https://en.wikipedia.org/wiki/Strategy_pattern)



Strategy Pattern and OO DesignPrinciples

- Strategy follows these principles
  - Classes should be open for extension but closed for modification (Open/Closed Principle)
  - Favor composition/delegation/aggregation over inheritance
  - Depend on abstractions. Do not depend on concrete classes.
  - Encapsulate what varies

Strategy uses aggregation instead of inheritance

- Behaviors are defined as separate interfaces and specific classes that implement these interfaces
- This allows better decoupling between the behavior and the class that uses the behavior. The behavior can change without breaking the classes that use it, and classes can switch behaviors by changing the specific implementation used without requiring any significant code changes.

Good OO and Design Pattern Bullet Points

- Good OO designs are reusable, extensible, and maintainable (these are part of non-functional requirements for any software)
- Patterns are proven OO experience
- Patterns are NOT code, they are general solutions to design problems
- Most patterns address issues of change in software
- Most patterns allow some part of a system to vary independently of all other parts. We should take what varies in a system and encapsulate it if possible.

Python Example

```
class Robot:
    def __init__(self, func=None):
        if func:
            self.execute = func

    def move(self):
        print(“Robot Moving Normally")
```

Python Example

```
def moveAggressively():
    print(“Robot Moving Aggressively")

def moveDefensively():
    print(“Robot Moving Defensively")
```

Python Example

```
if __name__ == "__main__":
    robot = Robot()
    robot1 = Robot(moveAggressively)
    robot2 = Robot(moveDefensively)

    robot.execute()
    robot1.execute()
    robot2.execute()
```