Strategy Pattern

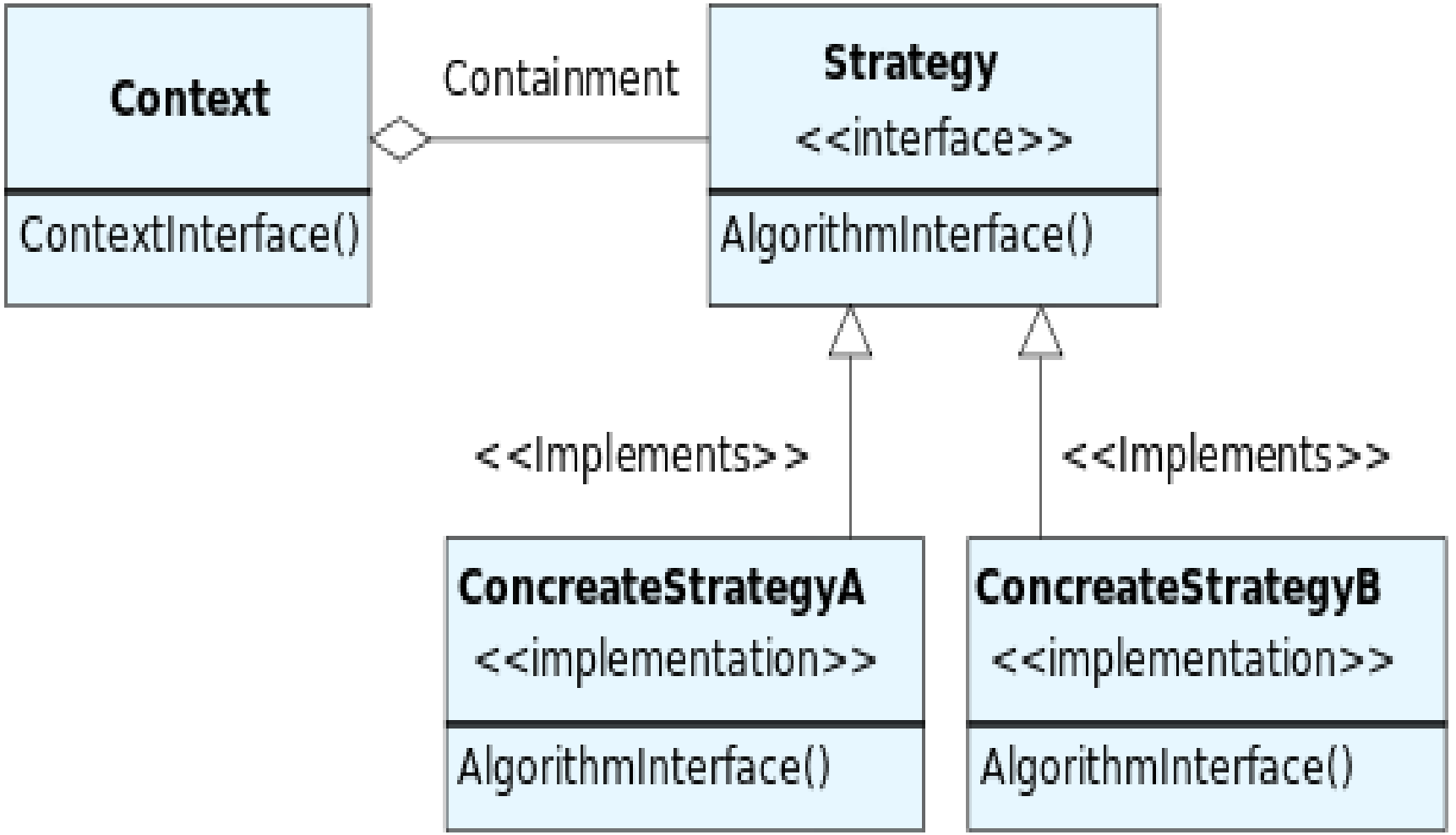
From Head First: Defines a family of algorithms, encapsulates each one, and makes them interchangeable. It lets the algorithms vary independently from the clients that use it.

From Wikipedia: A pattern whereby algorithms can be selected at runtime.

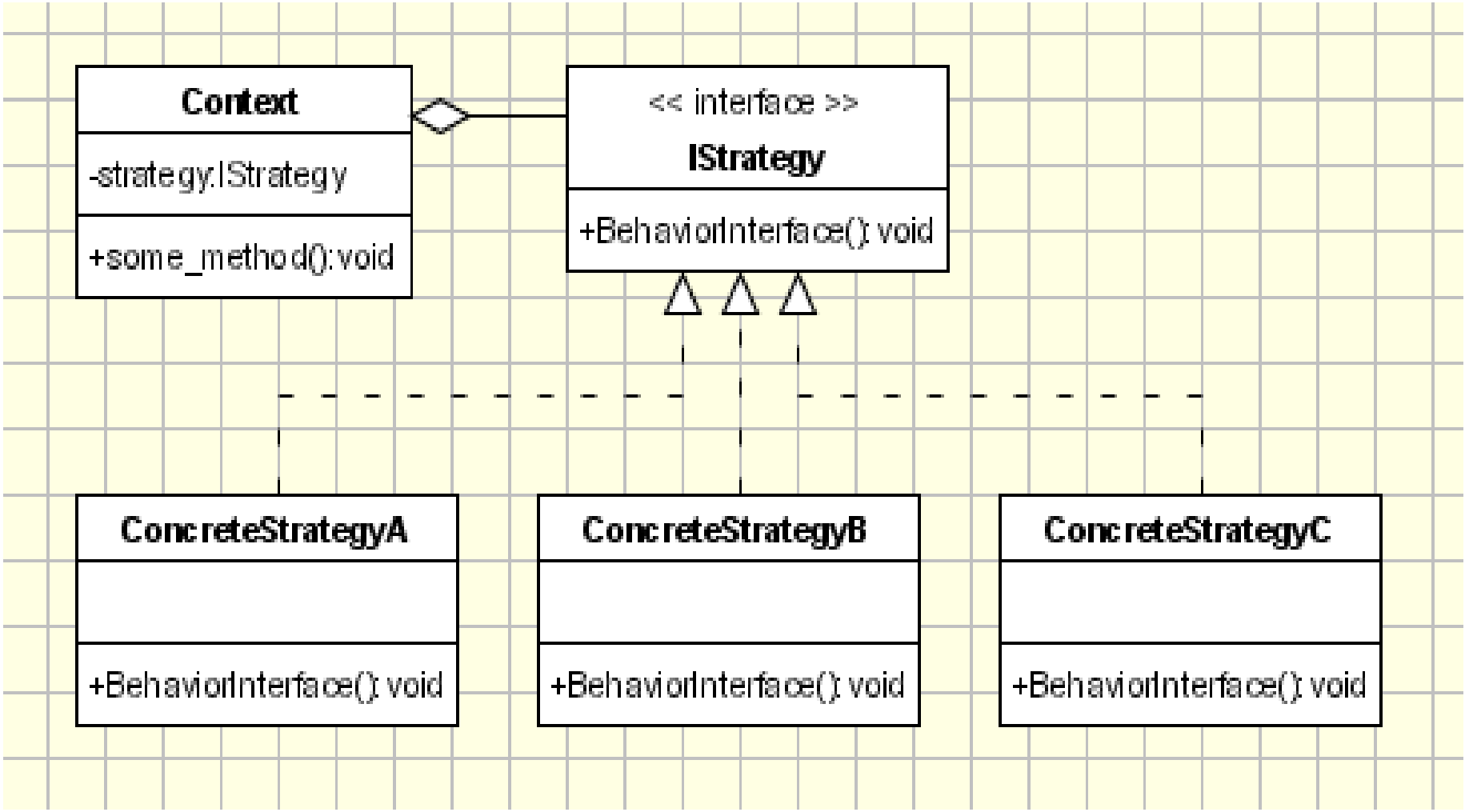
Motivation: There are common situations when classes differ only in their behavior. For this cases is a good idea to isolate the algorithms in separate classes in order to have the ability to select different algorithms at runtime.

It is a behavioral pattern in that it allows you to change the behavior of an algorithm dynamically.

Strategy UML



A second UML diagram



Second UML Diagram Explained

•Strategy - defines an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.

•ConcreteStrategy - each concrete strategy implements an algorithm.

•Context
- contains a reference to a strategy object.
- may define an interface that lets strategy accessing its data.
- The Context objects contains a reference to the ConcreteStrategy that should be used. When an operation is required then the algorithm is run from the strategy object. The Context is not aware of the strategy implementation. If necessary, addition objects can be defined to pass data from context object to strategy.

•The context object receives requests from the client and delegates them to the strategy object. Usually the ConcreteStartegy is created by the client and passed to the context. From this point the clients interacts only with the context.

Strategy Pattern 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 run-time, 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.

Code Example

```
// The classes that implement a concrete strategy should
implement this.
// The Context class uses this to call the concrete strategy.
interface Strategy {
    int execute(int a, int b);
}

// Implements the algorithm using the strategy interface
class ConcreteStrategyAdd implements Strategy {

    public int execute(int a, int b) {
        System.out.println("Called ConcreteStrategyAdd's execute()");
        return a + b;  // Do an addition with a and b
    }
}
```

Code Example Continued

```
class ConcreteStrategySubtract implements Strategy {

    public int execute(int a, int b) {
        System.out.println("Called ConcreteStrategySubtract's execute()");
        return a - b;  // Do a subtraction with a and b
    }
}

class ConcreteStrategyMultiply implements Strategy {

    public int execute(int a, int b) {
        System.out.println("Called ConcreteStrategyMultiply's execute()");
        return a * b;   // Do a multiplication with a and b
    }
}
```

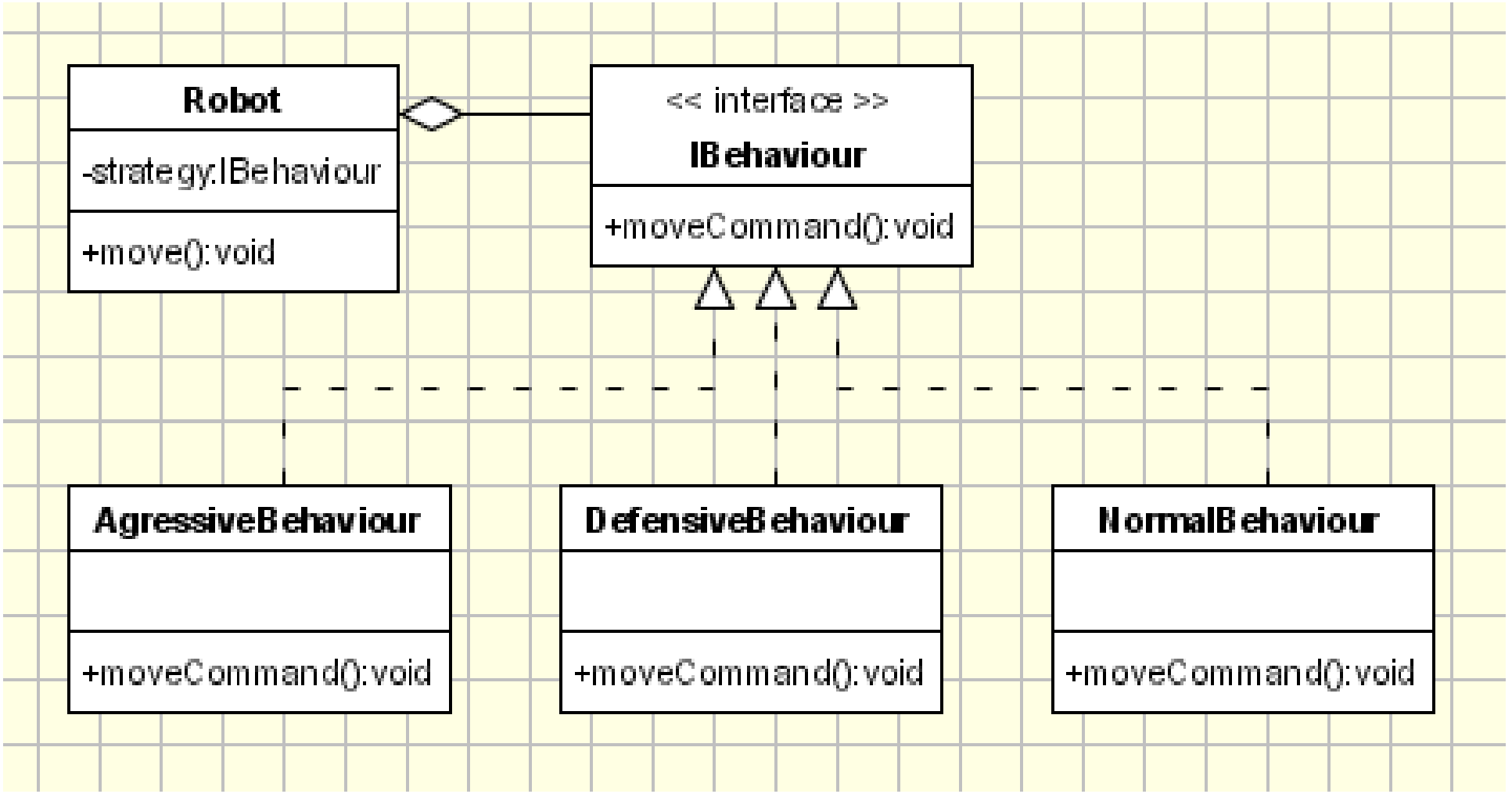
Code Example Continued

```
// Configured with a ConcreteStrategy object and maintains a
reference to a Strategy object
class Context {

    private Strategy strategy;

    // Constructor
    public Context(Strategy strategy) {
        this.strategy = strategy;
    }

    public int executeStrategy(int a, int b) {
        return strategy.execute(a, b);
    }
}
```

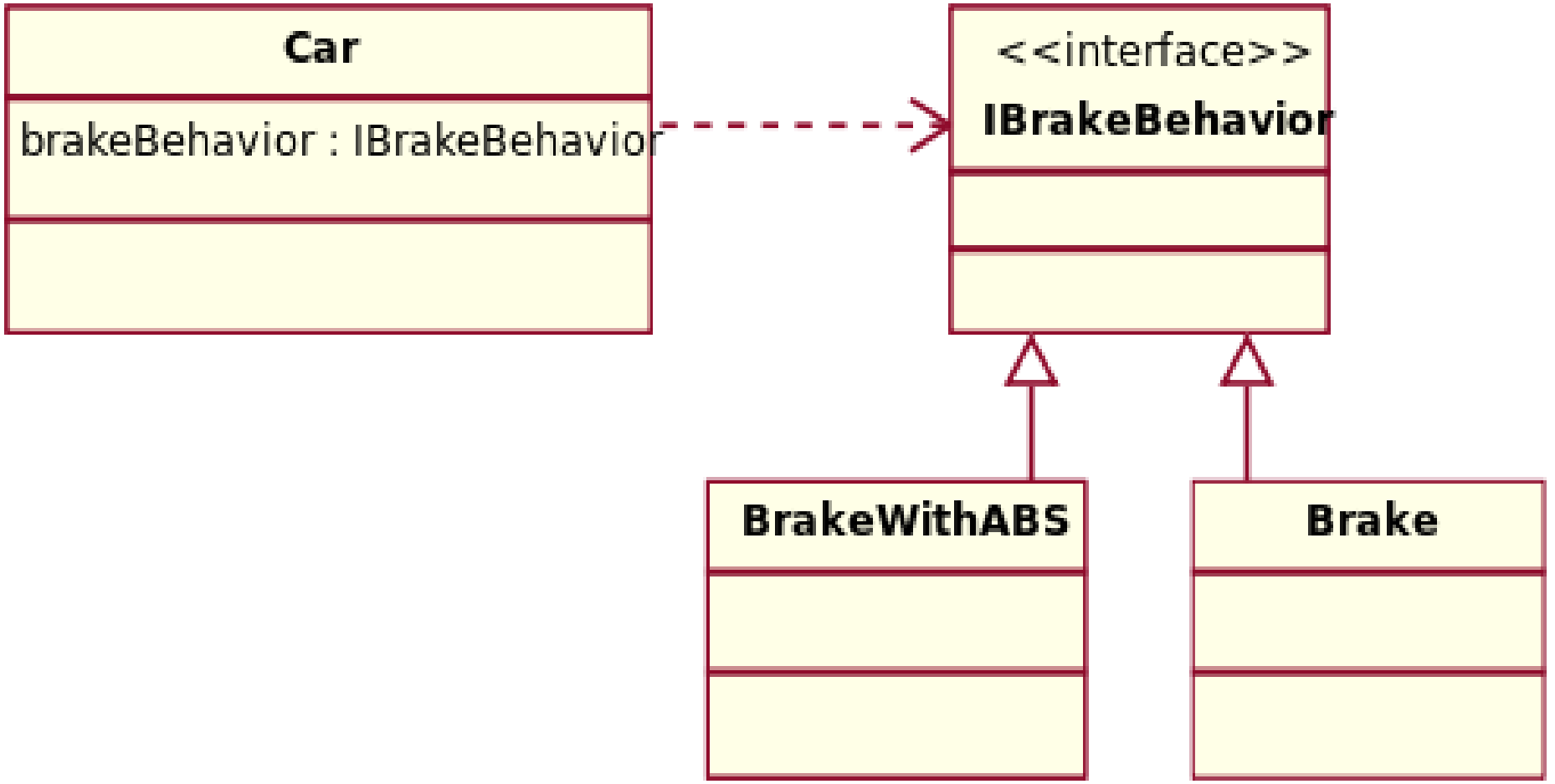
Code Example Finish

```
// Test application
class StrategyExample {

    public static void main(String[] args) {

        Context context;

        // Three contexts following different strategies
        context = new Context(new ConcreteStrategyAdd());
        int resultA = context.executeStrategy(3,4);

        context = new Context(new ConcreteStrategySubtract());
        int resultB = context.executeStrategy(3,4);

        context = new Context(new ConcreteStrategyMultiply());
        int resultC = context.executeStrategy(3,4);
    }
}
```

Strategy Scenario in UML



Second Strategy Scenario in UML



Strategy Pattern and Design Principles

- 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.

OO and Pattern Bullet Points

- Good OO designs are reusable, extensible, and maintainable
- 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