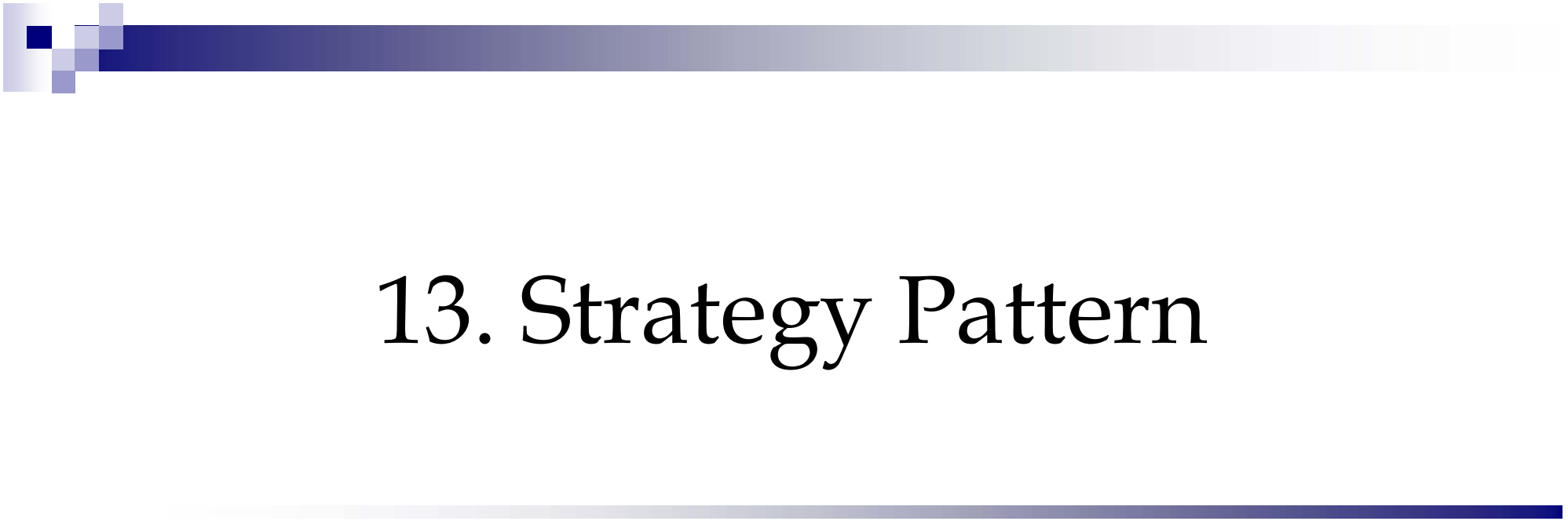
Design Patterns

宋 杰

Song Jie

东北大学 软件学院

Software College, Northeastern University



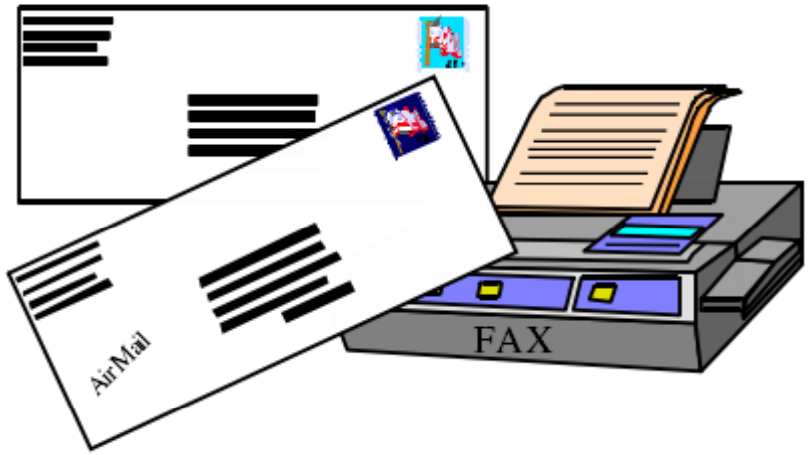
13. Strategy Pattern

Intent

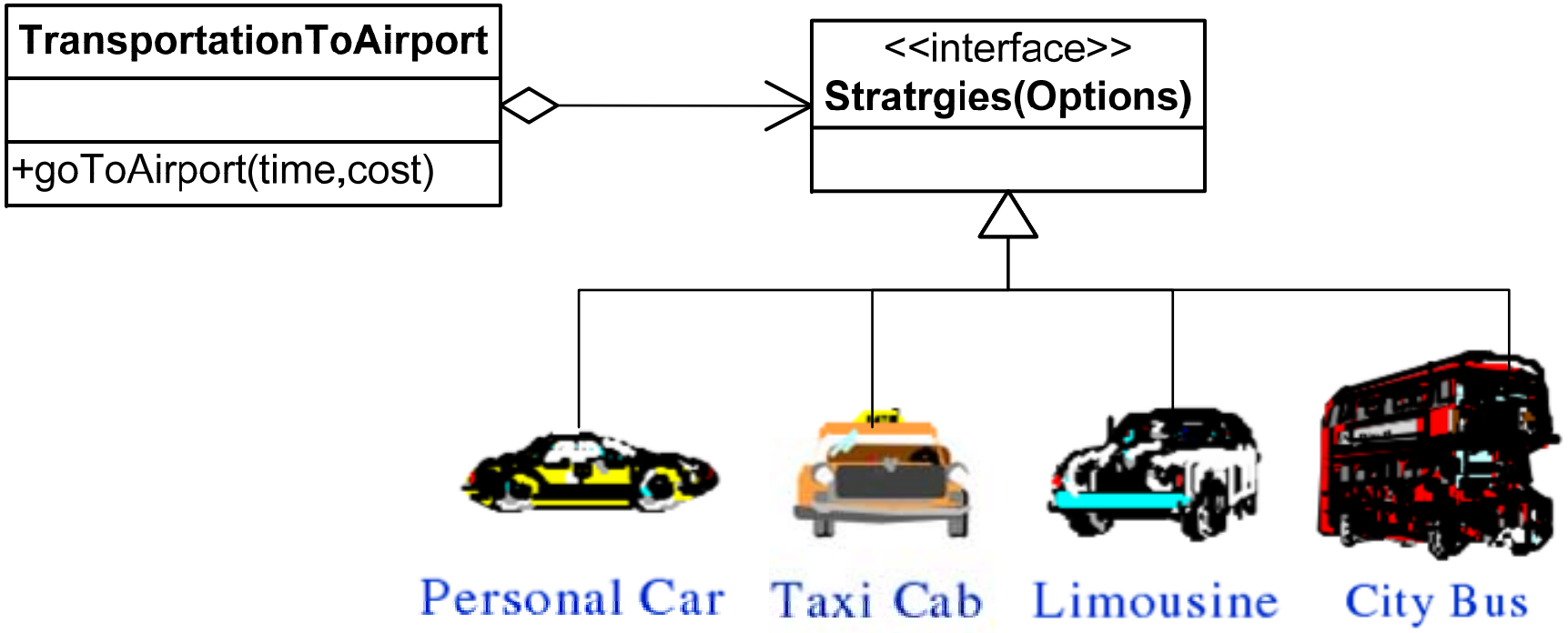
- Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- 针对一组算法，将每一个算法封装到具有共同接口的独立的类中，从而使得它们可以相互替换。策略模式使得算法可以在不影响到客户端的情况下发生变化。
- Pluggable Algorithms

Example

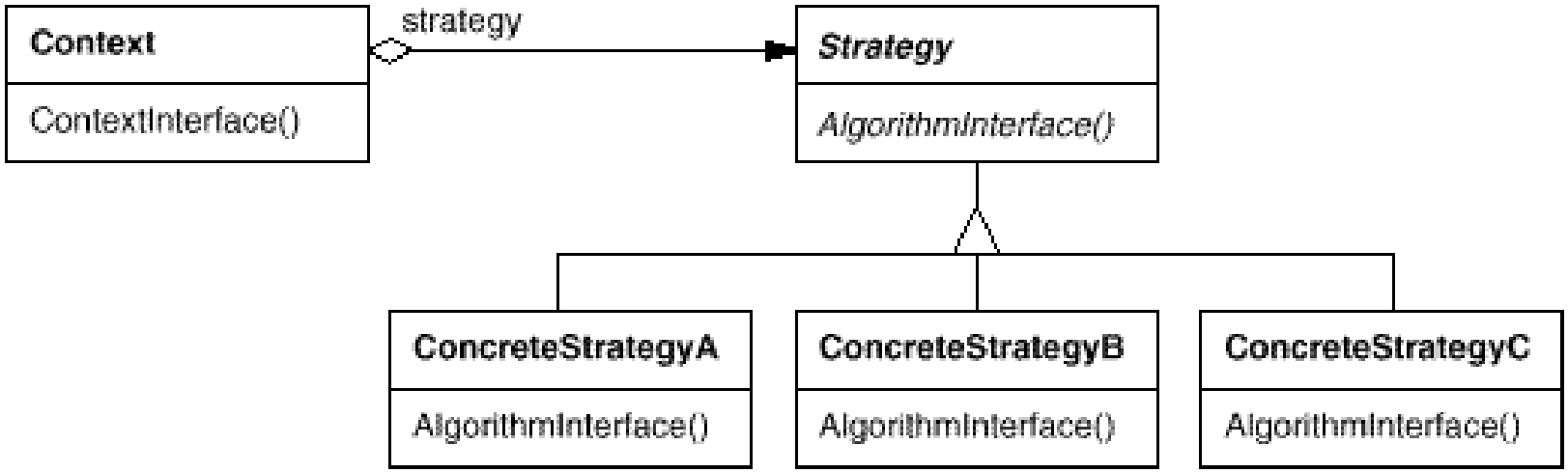
- Faxing, common mail, air-mail, and surface-mail all get a document from one place to another, but in different ways.



Example


TransportationToAirport
+goToAirport(time,cost)
<<interface>>
Stratrgies(Options)
Personal Car
Taxi Cab
Limousine
City Bus

Structure


Context
ContextInterface()
strategy
Strategy
AlgorithmInterface()
ConcreteStrategyA
AlgorithmInterface()
ConcreteStrategyB
AlgorithmInterface()
ConcreteStrategyC
AlgorithmInterface()

Participants

- Strategy: Declares an interface common to all supported algorithms. Context uses this interface to call the algorithm defined by a ConcreteStrategy.
- ConcreteStrategy: Implements the algorithm using the Strategy interface.
- Context is configured with a ConcreteStrategy object. maintains a reference to a Strategy object.
  - May define an interface that lets Strategy access its data.

Collaborations

- Strategy and Context interact to implement the chosen algorithm.
  - A context may pass all data required by the algorithm to the strategy when the algorithm is called.
  - Alternatively, the context can pass itself as an argument to strategy operations. That lets the strategy call back on the context as required.
- A Context forwards requests from its clients to its strategy.
  - There is often a family of ConcreteStrategy classes for a client to choose from.
  - Clients usually create and pass a ConcreteStrategy object to the context;
  - Clients interact with the context exclusively.

Consequences -benefits

- Families of related algorithms
  - Hierarchies of Strategy classes define a family of algorithms or behaviors for contexts to reuse.
- An alternative to subclassing.
  - Encapsulating the algorithm in separate Strategy classes lets you vary the algorithm independently of its context, making it easier to switch, understand, and extend. (CRP)
- Strategies eliminate conditional statements.
  - The Strategy pattern offers an alternative to conditional statements for selecting desired behavior.
- A choice of implementations.
  - Strategies can provide different implementations of the same behavior. The client can choose among strategies with different time and space trade-offs.

Consequences-drawbacks

- Clients must be aware of different Strategies.
  - Client must understand how Strategies differ before it can select the appropriate one.
- Communication overhead between Strategy and Context.
  - The Strategy is shared by all ConcreteStrategy classes whether they are simple or complex. Hence it's likely that some ConcreteStrategies won't use all the information passed by Context;
- Increased number of objects.
  - Shared strategies which not maintain state across invocations can reduce the number of objects. (Flyweight)

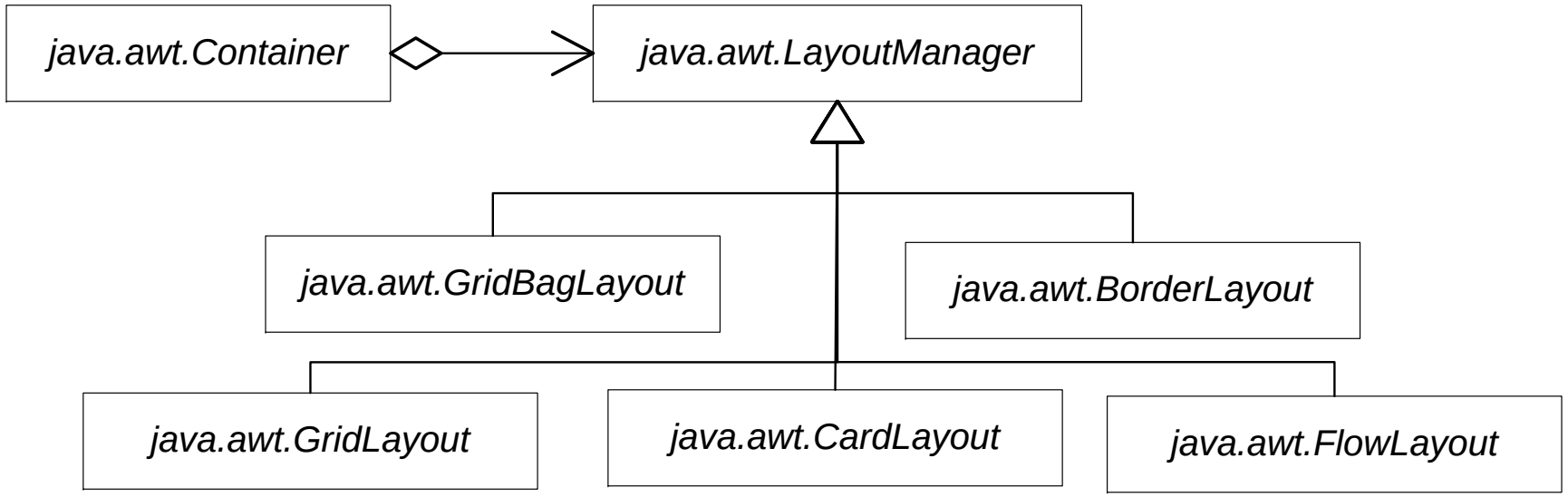
Applicability

- Many related classes differ only in their behavior. Strategies provide a way to configure a class with one of many behaviors.
- You need different variants of an algorithm. Strategies can be used when these variants are implemented as a class hierarchy of algorithms.
  - For example, you might define algorithms reflecting different space/time trade-offs.
- An algorithm uses data that clients shouldn't know about. Use the Strategy pattern to avoid exposing complex, algorithm-specific data structures.
- A class defines many behaviors, and these appear as multiple conditional statements in its operations. Instead of many conditionals, move related conditional branches into their own Strategy class.

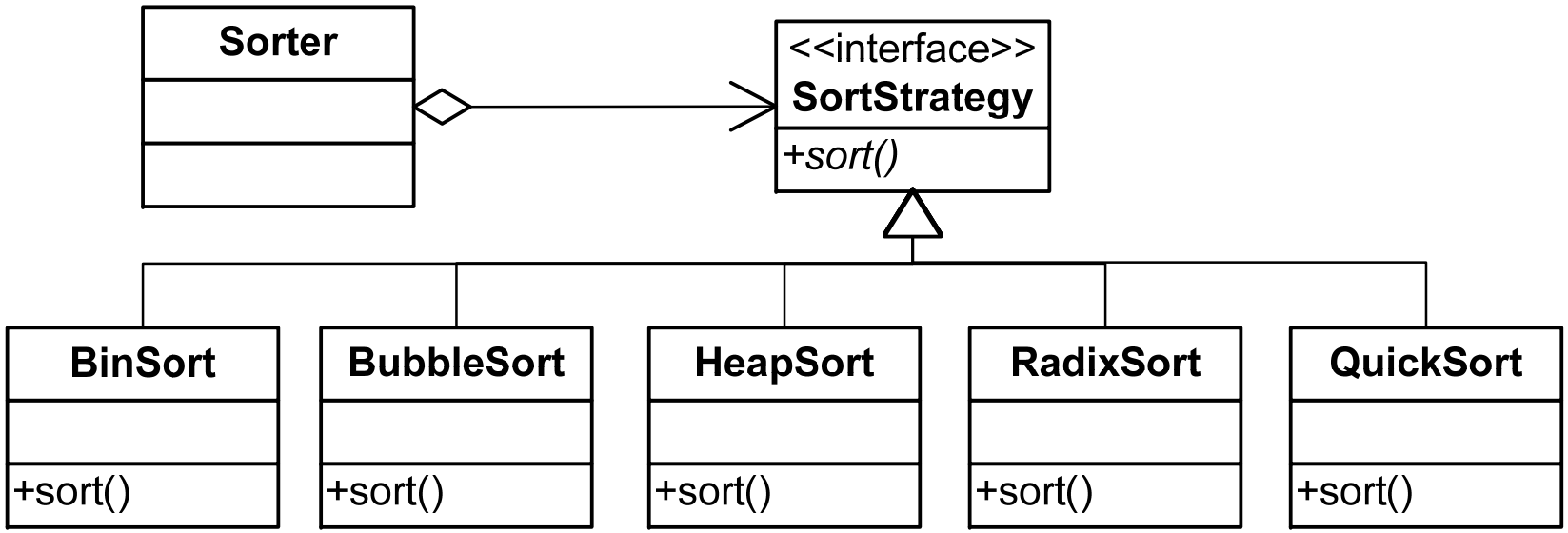
Example 1: Promotion

- Different commodities ,for example, books, have different discount
  - Computer books, 20%;
  - English books, 30%;
  - Children's books, 40%
  - Economic books, 0%;
  - Special book, 80%
- New discount approach may be introduced.

Example 2: LayoutManager in AWT



Example 3: Sorter System



Extension 1: Passing data between Context and Strategy

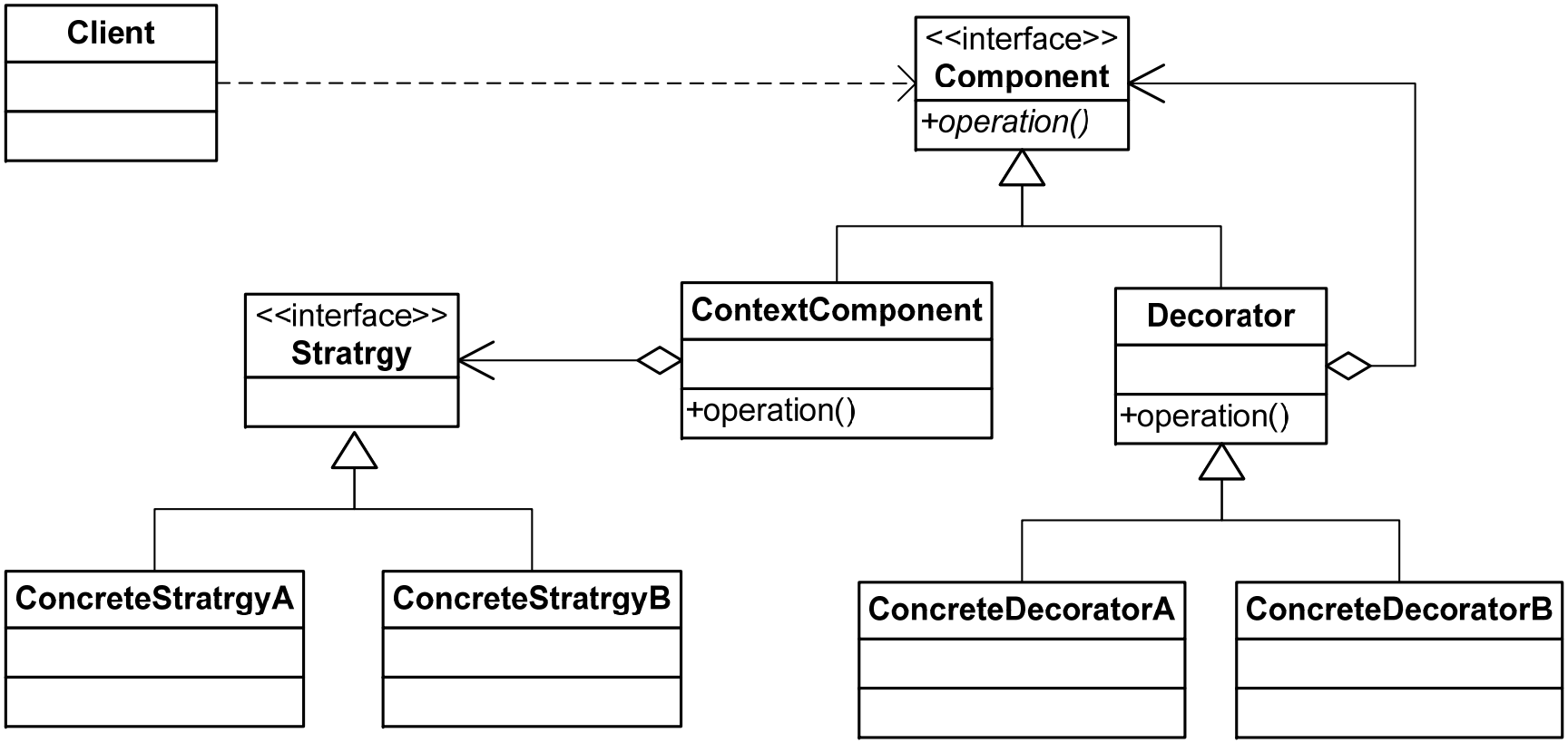
- The Strategy and Context must give a ConcreteStrategy efficient access to any data it needs from a context, and vice versa.
  - One approach is to have Context pass data in parameters to Strategy operations.
    - Simple approach
    - This keeps Strategy and Context decoupled.
    - Context might pass data the Strategy doesn't need.
  - A context pass itself as an argument, and the strategy requests data from the context explicitly.
    - Strategy can store a reference to its context, eliminating the need to pass anything at all.
    - Context must define a more elaborate interface to its data, which couples Strategy and Context more closely.

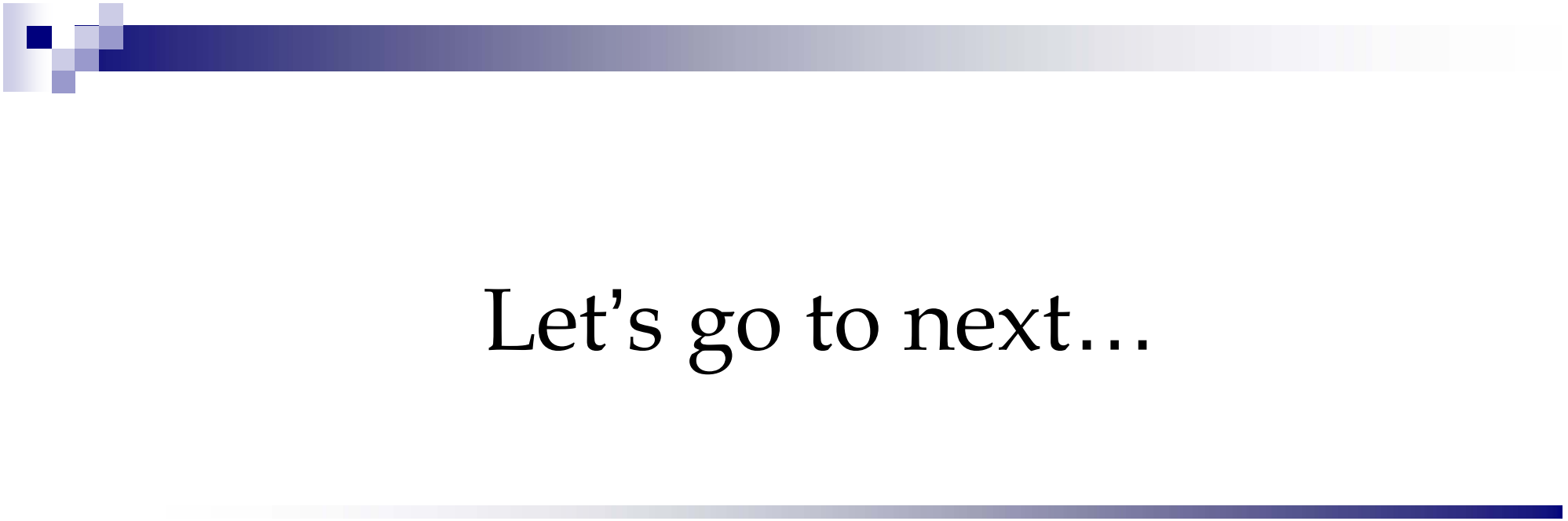
Extension 2: Making strategy objects optional

- The Context class may be simplified if it's meaningful not to have a Strategy object.
- Context checks to see if it has a Strategy object before accessing it.
  - If there is one, then Context uses it normally.
  - If there isn't a strategy, then Context carries out default behavior.
- Or it can be treated as an default Strategy is set up to the Context .

Think about it

- Changing the guts of an object versus changing its skin.
  - The Strategy pattern is a good example of a pattern for changing the guts.
  - The Decorator pattern is a good example of a pattern for changing the skin.
- Can we use Strategy and Decorator together ?


Client
<<interface>>
Component
+operation()
<<interface>>
Stratrgy
ContextComponent
+operation()
Decorator
+operation()
ConcreteStratrgyA
ConcreteStratrgyB
ConcreteDecoratorA
ConcreteDecoratorB



Let’s go to next…