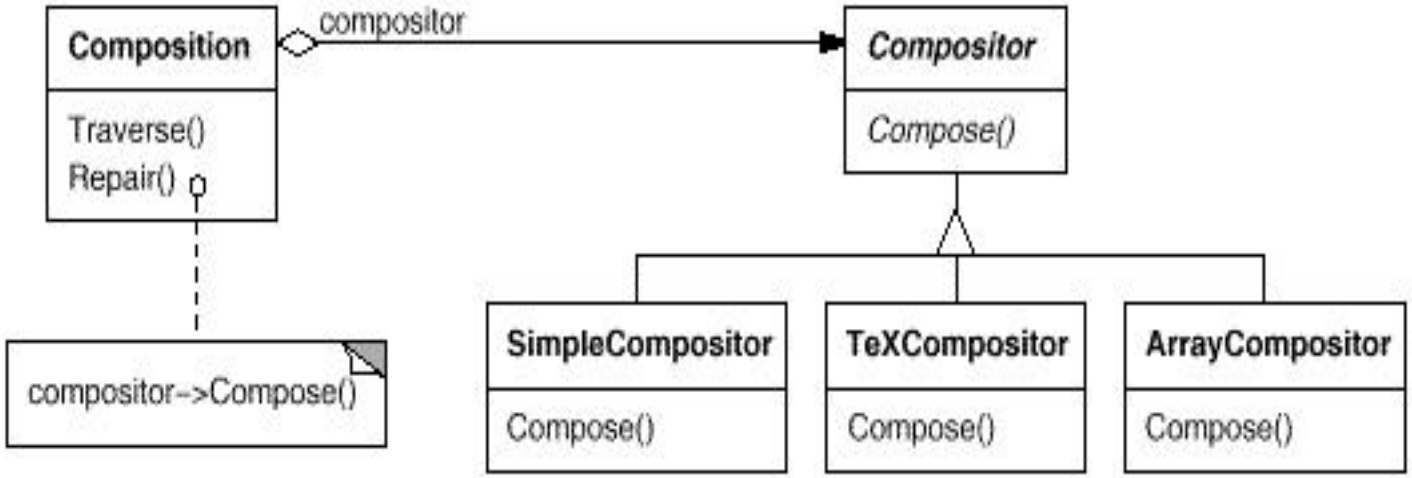
The Strategy Pattern

- Intent
  - ⇨ Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
- Motivation



The Strategy Pattern

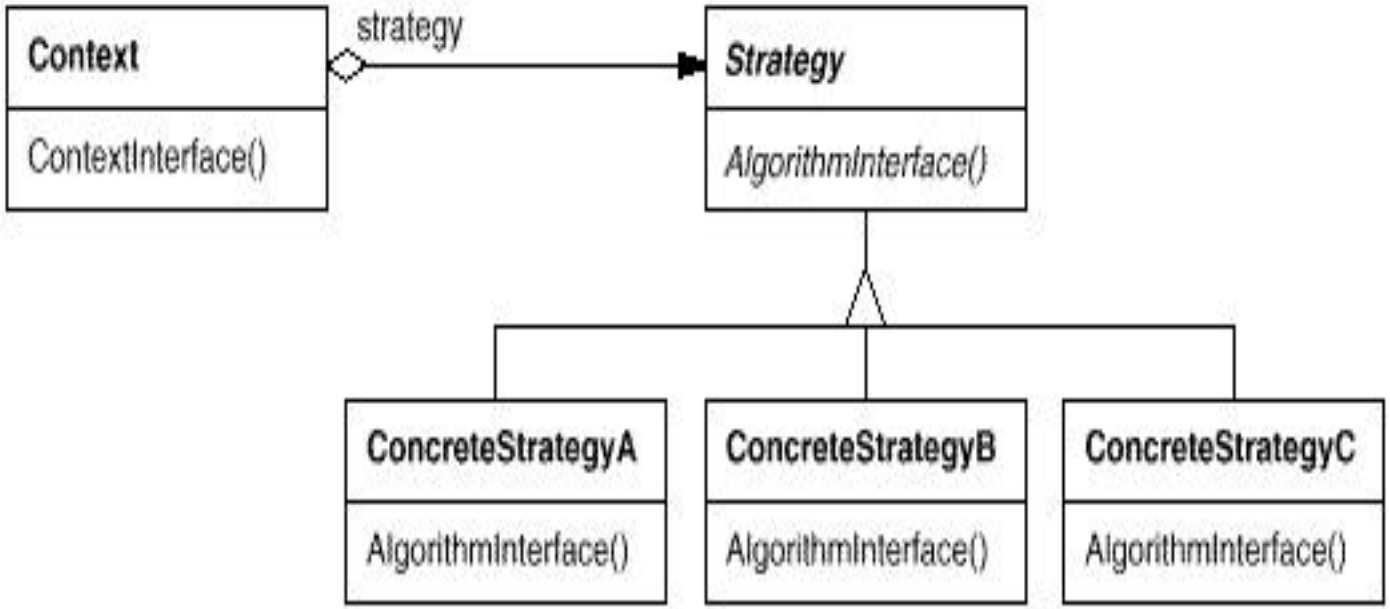
- Applicability

  Use the Strategy pattern whenever:

  - Many related classes differ only in their behavior
  - You need different variants of an algorithm
  - 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.

The Strategy Pattern

- Structure

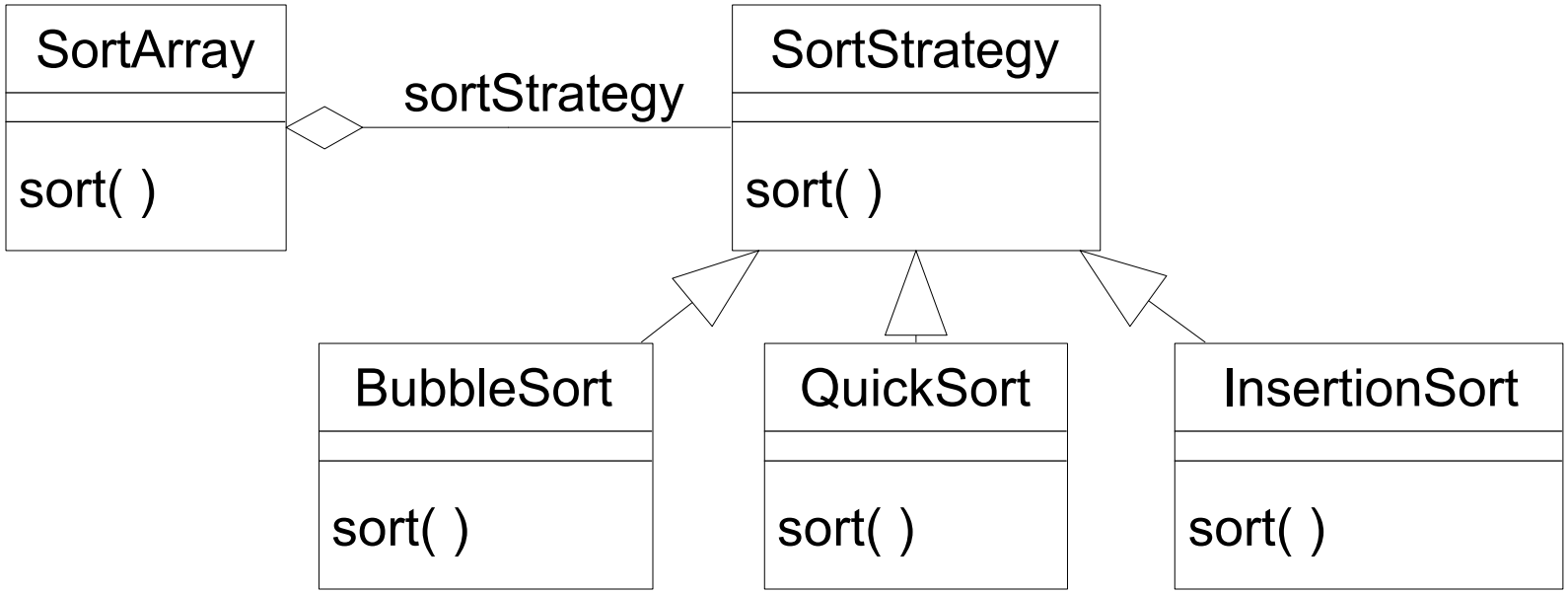


The Strategy Pattern

- Consequences
  - Benefits
    - Provides an alternative to subclassing the Context class to get a variety of algorithms or behaviors
    - Eliminates large conditional statements
    - Provides a choice of implementations for the same behavior
  - Liabilities
    - Increases the number of objects
    - All algorithms must use the same Strategy interface

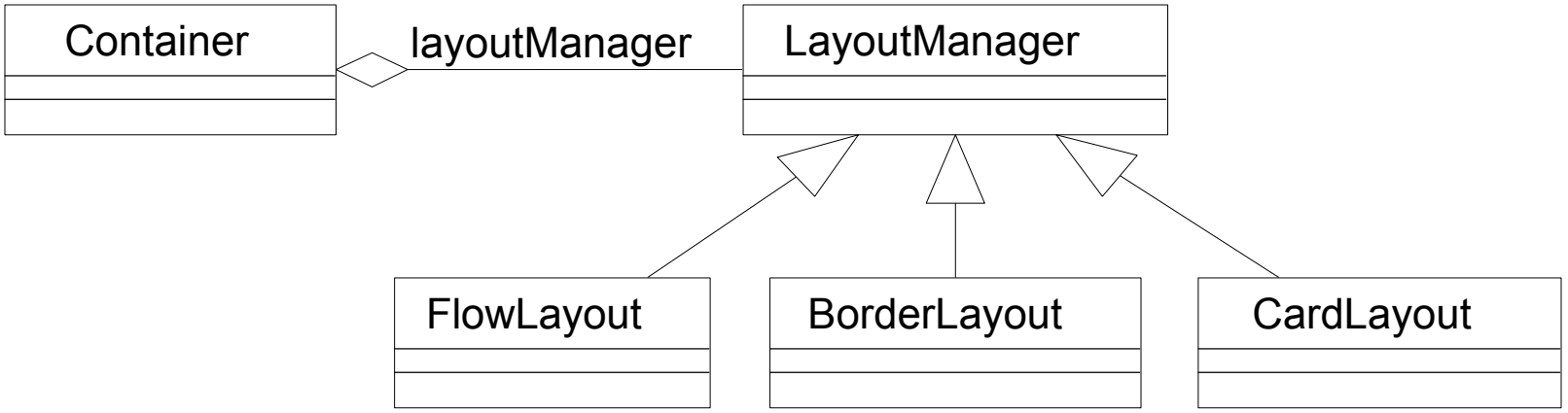
Strategy Pattern Example 1

- Situation: A class wants to decide at run-time what algorithm it should use to sort an array. Many different sort algorithms are already available.
- Solution: Encapsulate the different sort algorithms using the Strategy pattern!



Strategy Pattern Example 2

- Situation: A GUI container object wants to decide at run-time what strategy it should use to layout the GUI components it contains. Many different layout strategies are already available.
- Solution: Encapsulate the different layout strategies using the Strategy pattern!
- Hey! This is what the Java AWT does with its LayoutManagers!



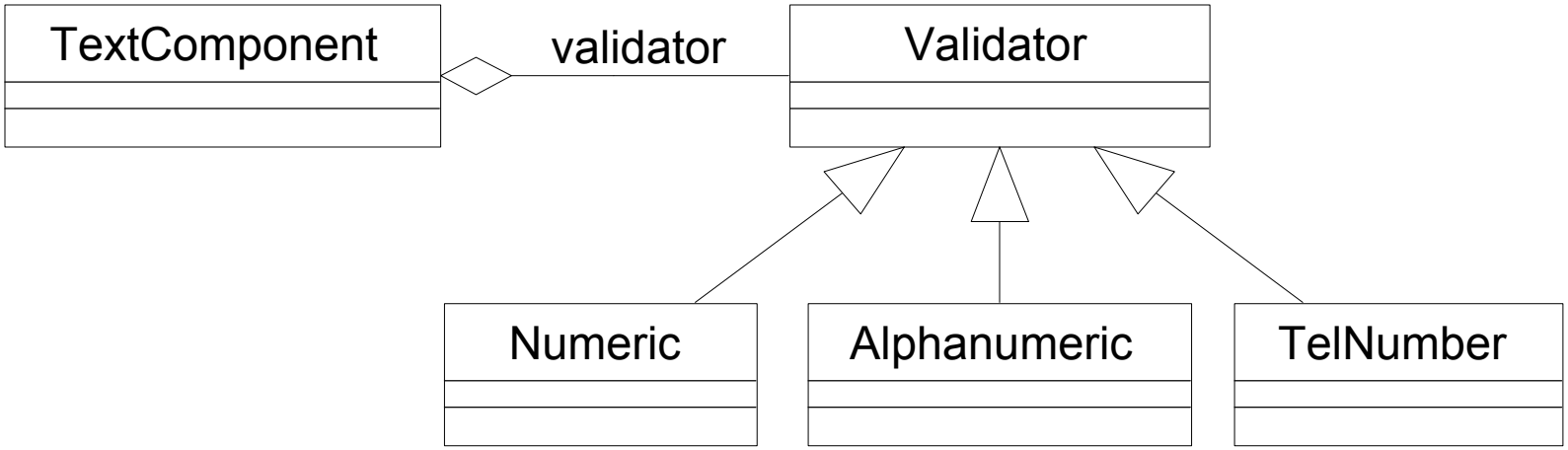
Strategy Pattern Example 2 (Continued)

- Some client code:

```
Frame f = new Frame();
f.setLayout(new FlowLayout());
f.add(new Button("Press"));
```

Strategy Pattern Example 3

- Situation: A GUI text component object wants to decide at run-time what strategy it should use to validate user input. Many different validation strategies are possible: numeric fields, alphanumeric fields, telephone-number fields, etc.
- Solution: Encapsulate the different input validation strategies using the Strategy pattern!



Strategy Pattern Example 3 (Continued)

- This is the technique used by the Java Swing GUI text components. Every text component has a reference to a document model which provides the required user input validation strategy.

The Null Object Pattern

- Sometimes the Context may not want to use the strategy provided by its contained Strategy object. That is, the Context wants a "do-nothing" strategy.
- One way to do this is to have the Context assign a null reference to its contained Strategy object. In this case, the Context must always check for this null value:

```
if (strategy != null)
   strategy.doOperation();
```

The Null Object Pattern

- Another way to accomplish this is to actually have a “do-nothing” strategy class which implements all the required operations of a Strategy object, but these operations do nothing. Now clients do not have to distinguish between strategy objects which actually do something useful and those that do nothing.
- Using a “do-nothing” object for this purpose is known as the *Null Object Pattern*

The Strategy Pattern

- Note the similarities between the State and Strategy patterns! The difference is one of intent.
  - ⇨ A State object encapsulates a state-dependent behavior (and possibly state transitions)
  - ⇨ A Strategy object encapsulates an algorithm
- And they are both examples of Composition with Delegation!