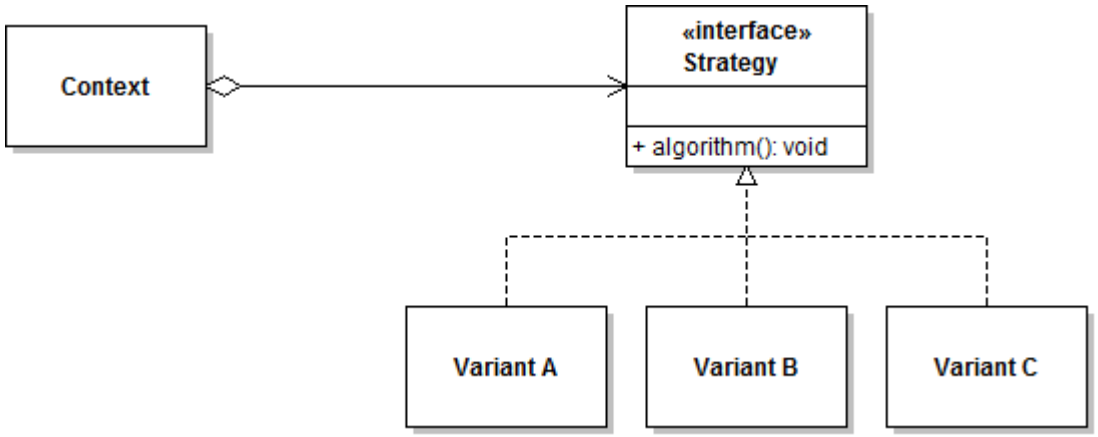
Strategy Pattern

- **Intent**
  - Define a family of algorithms
    - Encapsulate each one
    - Make algorithms interchangeable
    - Support extensibility with new algorithms
  - Allow to change algorithms independently of the clients that use them

- **Examples**
  - Swing: Look & Feel
  - AWT: Layout Manager
  - Collections: Order in sorted sets
  - JDraw: DrawGrid (may also be considered as state)

Strategy Pattern: Structure

- **Structure**



  - *Context* uses a concrete *Strategy* instance
  - The interface of Strategy has to be powerful enough so that all (existing and future) algorithms can be supported
  - *Context* can define methods which allow the *Strategy* to access the context state

Strategy Pattern: Example

- **Motivation**
  - A security application allows for encrypting and decrypting data before they are sent over the net
    - => SecureChannel
  - The user can choose the encryption/decryption algorithm (e.g., AES, Blowfish, and IDEA) at run time
  - How should the security application be designed with respect to offering a choice of encryption/decryption algorithms?

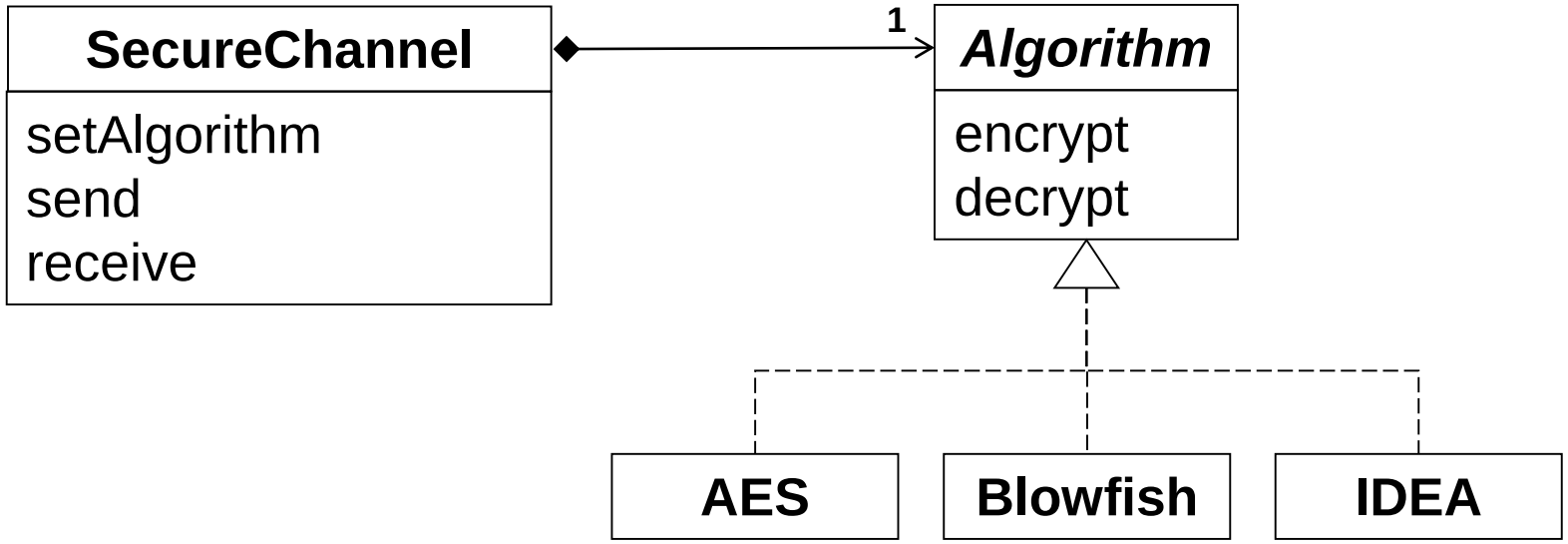
Strategy Pattern: Example

```
public class SecureChannel {
   public enum Algorithm { AES, Blowfish, IDEA; }
   private Algorithm algorithm;
   public void setAlgorithm(Algorithm algorithm) {
       this.algorithm = algorithm;
   }

   public void send(byte[] key, int[] plain) {
      switch (algorithm) {
         case AES:      encrypted = ...; break;
         case Blowfish: encrypted = ...; break;
         case IDEA:     encrypted = ...; break;
         default:       assert false; break;
      }
      write(encrypted);
   }
   public int[] receive(byte[] key) {
      ...
   }
}
```

Strategy Pattern: Example

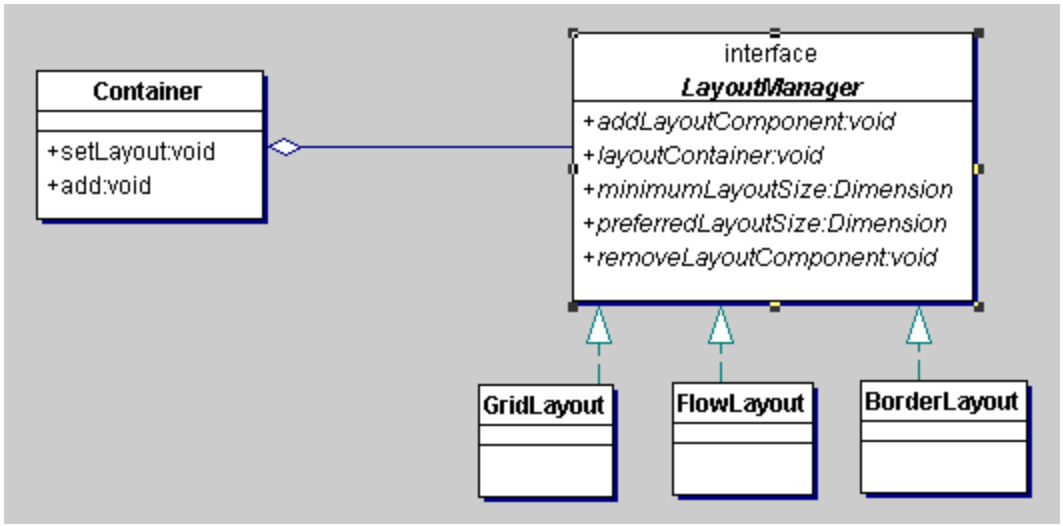
```
public class SecureChannel {

   public interface Algorithm {
      public int[] encrypt(byte[] key, int[] plain);
      public int[] decrypt(byte[] key, int[] encrypted);
   }

   private Algorithm algorithm;
   public void setAlgorithm(Algorithm algorithm) {
      if (algorithm == null) throw new IllegalArgumentException();
      this.algorithm = algorithm;
   }

   public void send(byte[] key, int[] plain) {
      write(algorithm.encrypt(key, plain));
   }
   public int[] receive(byte[] key) {
      return algorithm.decrypt(key, read());
   }
}
```

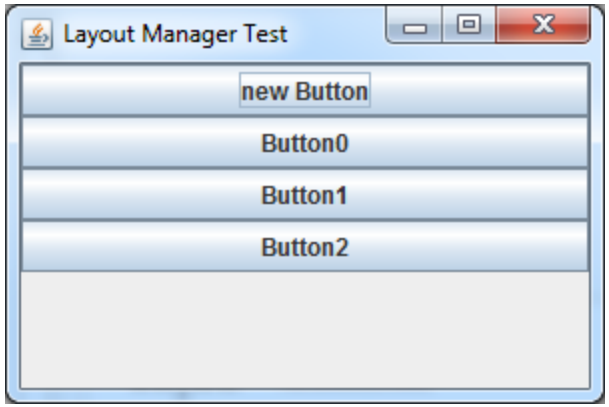
Strategy Pattern: Example



Strategy Pattern: Example

- **LayoutManager:**





Strategy Pattern: Example

```
public interface LayoutManager {
   void addLayoutComponent(String name, Component comp);
   void removeLayoutComponent(Component comp);

   Dimension preferredLayoutSize(Container parent);
   Dimension minimumLayoutSize(Container parent);

   /**
    * Lays out the specified container.
    * @param parent the container to be laid out
    */
   void layoutContainer(Container parent);
}
```

Strategy Pattern: Example

```
import java.awt.*;

public class ListLayout implements LayoutManager {
   // Strategy:
   // respect preferred height of components
   // set width of components to container's width

   public void addLayoutComponent(String name, Component comp) { }
   public void removeLayoutComponent(Component comp) { }

   public Dimension minimumLayoutSize(Container parent) {
      return parent.getSize();
   }

   public Dimension preferredLayoutSize(Container parent) {
      return parent.getSize();
   }
```

Strategy Pattern: Example

```
   public void layoutContainer(Container parent) {
      Insets insets = parent.getInsets();
      int x = insets.left;
      int y = insets.top;
      int w = parent.getSize().width – (insets.left+insets.right);

      int numberOfComponents = parent.getComponentCount();
      for (int i = 0; i < numberOfComponents; i++) {
         Component c = parent.getComponent(i);
         if (c != null && c.isVisible()) {
            c.setBounds(x, y, w, c.getPreferredSize().height);
         }
         y += c.getPreferredSize().height;
      }
   }
}
```

Strategy Pattern: Preconditions

- **When can we speak of a Strategy Pattern?**
  - A context class must want to use different variants of an algorithm
  - The context can also deal with new implementations of the algorithm
  - There must be an *interface type* that is an abstraction for the algorithm
  - Concrete strategy classes must implement the strategy interface type
  - The context class uses the strategy object to invoke the algorithm
  - A client supplies an object of a concrete strategy class to the context

- **When do we not speak of a Strategy Pattern?**
  - Only one algorithm which depends on parameters
    - E.g. filename for internationalization
    - Size of a Grid: 5/10/20 pixels

Strategy Pattern: Applicability

- **Extensibility** ***Design / Architecture***
  - You need different variants of an algorithm
  - Finding a simple and powerful enough interface is challenging
- **Separation** ***Refactoring***
  - 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
- **Combination** ***Refactoring***
  - Many related classes differ only in their behavior.

    => extract common behavior as context
- **Use** ***Implementation***
  - You implement a given strategy interface

Strategy vs. State

- **Strategy**
  - Represents an algorithm, interface typically defines a "compute" method
  - Strategy is typically set only once
  - Strategy is chosen externally (`setStrategy`) or by the context (depending on parameters)
  - Strategy is typically not aware of other concrete strategies
  - Usually only one public method (and additional private methods)
  - Strategy may contain algorithm specific state
- **State**
  - Defines state-specific behavior, i.e. the behavior contained in a state object is specific to the state of the associated context
  - State changes are typical at run-time (impl. of a state machine)
  - State is set externally (`setState`) or by the state itself (`setNext-State`), choice usually dependent on state of context object
  - A concrete state may be aware of other concrete states (=>transition)
  - Usually several public methods for the state-specific behavior
  - State usually contains no state but accesses state in context

Null Object Pattern

- **Motivation**
  - How can “no strategy” be represented?
    - `setLayout(null)` => absolute positioning is possible
  - Context code is contaminated with statements of type
    - `if(strategy != null) { … }`
- **Intent**
  - The intent of a Null Object is to encapsulate the absence of an object by providing a substitutable alternative that offers suitable default *do nothing* behavior
- **Solution**
  - Provide a Strategy implementation for the null-case (empty methods)
- **Consequences**
  - No special case *null (null*-handling is abstracted away from the client)
  - Null-Strategy could be used as base class for other implementations

Null Object Pattern: NullLayout

```
public class NullLayout implements LayoutManager {
   public void addLayoutComponent(String name, Component comp) { }
   public void removeLayoutComponent(Component comp) { }

   public Dimension minimumLayoutSize(Container parent) {
      return parent.getSize();
   }
   public Dimension preferredLayoutSize(Container parent) {
      return parent.getSize();
   }
   public void layoutContainer(Container parent) { }
}
```