Behavioral Patterns

Template Method

Strategy

Behavioral Patterns

- Describe interactions between objects to achieve a complex behavior
- Divide responsibilities across collaborating objects
- Flexible and extensible
- Enable changing behavior at runtime by switching objects

Behavioral Patterns

- Chain of responsibility
  - Request delegated to the responsible service provider
- Command
  - Request as first-class object
- Iterator
  - Aggregate elements are accessed sequentially
- Interpreter
  - Language interpreter for a small grammar
- Template Method
  - Algorithm with some steps supplied by a derived class
- Strategy
  - Abstraction for selecting one of many algorithms
- Mediator
  - coordinates interactions between its colleagues
- Memento
  - Snapshot captures and restores object states privately
- Observer
  - Dependents update automatically when a subject changes
- State
  - Object whose behavior depends on its state
- Visitor
  - Operations applied to elements of a heterogeneous object structure

Behavioral Patterns

- Chain of responsibility
  - Request delegated to the responsible service provider
- Command
  - Request as first-class object
- Iterator
  - Aggregate elements are accessed sequentially
- Interpreter
  - Language interpreter for a small grammar
- Template Method
  - **Algorithm** with some steps supplied by a derived class
- Strategy
  - Abstraction for selecting one of many **algorithms**
- Mediator
  - coordinates interactions between its colleagues
- Memento
  - Snapshot captures and restores object states privately
- Observer
  - Dependents update automatically when a subject changes
- State
  - Object whose behavior depends on its state
- Visitor
  - Operations applied to elements of a heterogeneous object structure

Algorithms sharing steps

- Similar steps but specific details are different
- e.g. retrieve OS name from a system spec

Read the JSON file	Read the XML file
Parse the JSON file into a JSON object	parse XML file into Document object
Get “OS Version” data	Get the element with “OS Version” tag

- Algorithms with common structure are common

```
JSONParser parser = new JSONParser();
try {
    Object obj = parser.parse(new FileReader("system_specifications.json"));
    JSONObject jsonObject = (JSONObject) obj;
    String osVersion = (String) jsonObject.get("OS Version");
    System.out.println("The operating system version is: " + osVersion);
} catch (Exception e) {
    e.printStackTrace();
}
```

Read the JSON file	Read the XML file
Parse the JSON file into a JSON object	parse XML file into Document object
Get “OS Version” data	Get the element with “OS Version” tag

Refactored – click here

```
try {
    File inputFile = new File("system_specifications.xml");
    DocumentBuilderFactory dbFactory = DocumentBuilderFactory.newInstance();
    DocumentBuilder dBuilder = dbFactory.newDocumentBuilder();
    Document doc = dBuilder.parse(inputFile);
    doc.getDocumentElement().normalize();
    NodeList nList = doc.getElementsByTagName("system");
    for (int temp = 0; temp < nList.getLength(); temp++) {
        Node nNode = nList.item(temp);
        if (nNode.getNodeType() == Node.ELEMENT_NODE) {
            Element eElement = (Element) nNode;
            System.out.println("The operating system version is: "
                               +
eElement.getElementsByTagName("OSVersion").item(0).getTextContent());
        }
    }
} catch (Exception e) {
    e.printStackTrace();
```

Algorithms sharing steps

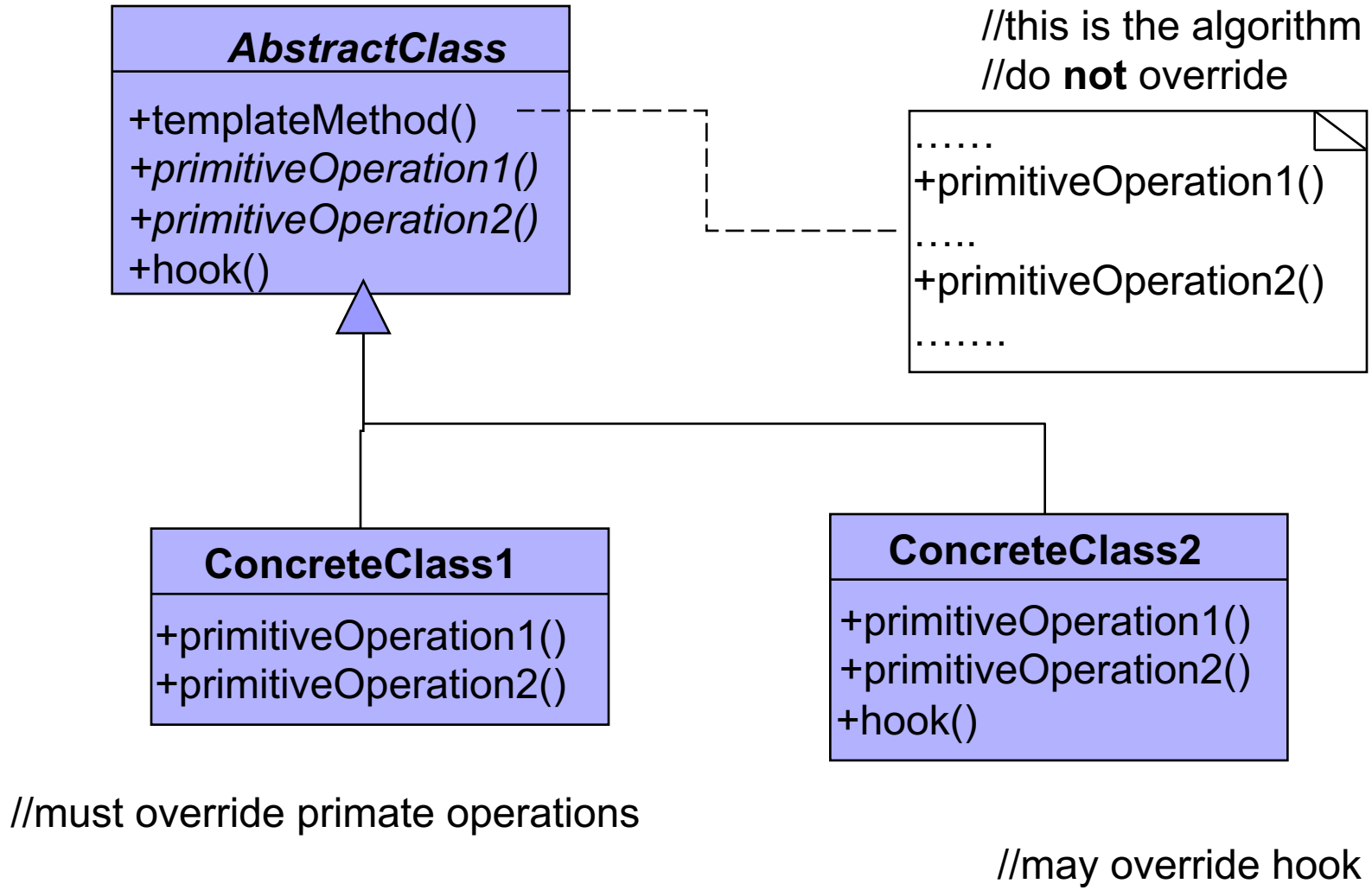
- We are going to think about the steps of an algorithm

  alg1:

  1) doA;
  2) doB;
  3) doC;

- Set the algorithm (order of steps) in stone
- Leave out the details of the steps to be defined by someone else
  - alg1 does not know how to carry out doA, someone else will define it for me

Structure –Template Method

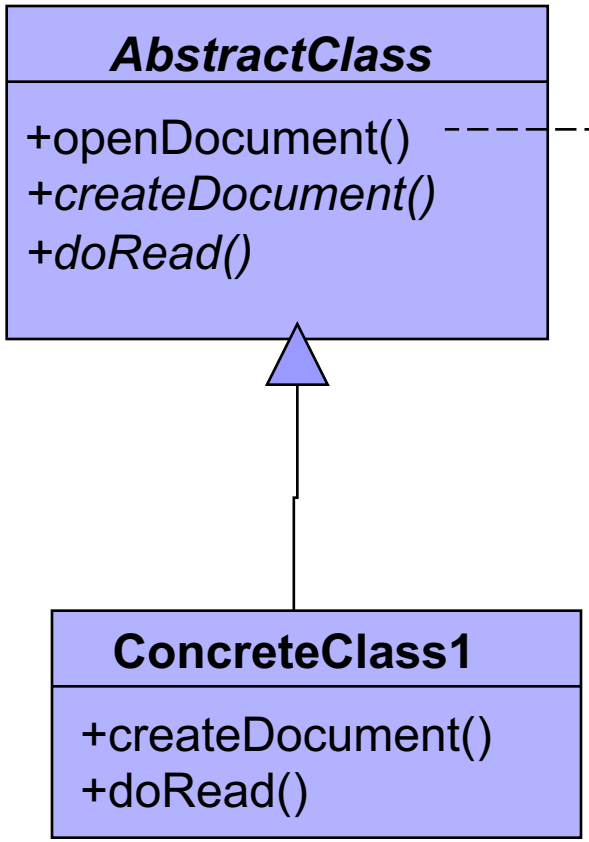


Example:Template Method

```
public void OpenDocument (String name) {
    if (!CanOpenDocument(name)) { return; }
    Document doc = createDocument();
    if (doc != null) {
      docs.AddDocument(doc);
      AboutToOpenDocument(doc);
      doc.Open();
      doc.doRead();
    }
}
```

- The OpenDocument() method is a *Template Method*
- The template method **fixes** the order of operations, but allows Application subclasses to **vary** those steps as needed
  - How to read the document and what kind of document to create is left to the subclass.

Template Method




```
if (!CanOpenDocument(name)) {
  return; }
Document doc = createDocument();
if (doc != null) {
     docs.AddDocument(doc);
      AboutToOpenDocument(doc);
      doc.Open();
      doc.doRead();
}
```

Template Method

- **Intent**:
  - Define the *skeleton* of an algorithm in an operation, deferring some steps to subclasses
  - TM lets subclasses redefine certain steps of an algorithm *without* changing the algorithm's structure.
- Removes code duplication!

- Motivation
  - want to specify the order of operations that a method uses, but allow subclasses to provide their own implementations of some of these operations

This is a class pattern, so we use …

Participants

- Template method ---Cannot be overridden
  - defines the algorithm skeleton
  - May call concrete operations, primitive operations and hooks
- Primitive operations –must be overridden
  - The operations in the template that must be implemented by the subclasses
  - The diversity occurs here
- Hooks –may be overridden
  - Superclass defines the default behavior
    - usually empty method body
  - Subclass may override it

Exercise 1: print documents

```
class PlainTextDocument{…
 public void printPage (Page
     page){
   System.out.println(page.title());
   System.out.println(page.body());
   System.out.println(page.date());
    }
…}

class HtmlTextDocument{
 public void printPage (Page
     page){
  printHtmlTextHeader();
  System.out.println(page.body());
```

- The order of tasks in printPage operations are the same
- Make printPage() a template method in a superclass
- Allow PlainTextDocument and HtmlTextDocument to provide their unique implementations of abstract methods to print the header and footer

Exercise 1:Template Method

```
public abstract class TextDocument {...
    public final void printPage (Page page) {     //final -- the algorithm is
     fixed
       printTextHeader(page);
       printTextBody(page);
      printTextFooter();
    }
   public abstract void printTextHeader();
   public final void printTextBody(Page page) {
          System.out.println(page.body()); }
   public abstract void printTextFooter();
}

public class PlainTextDocument extends TextDocument {...
   public void printTextHeader (Page page) {
          System.out.println(page.title()); }
   public void printTextFooter () {}
}
```

All we have to do is provide the proper implementations of the primitive operations

Applicability: Template Method

- To implement the **invariant** parts of an algorithm **once** and leave it up to subclasses to implement the behavior that can **vary**
  - particularly important in class libraries, because they are the means for factoring out common behavior in library classes
  - New programmer cannot mess up the invariant
  - This is how we customize frameworks
    - Extend a class in the framework
    - Override some methods for customization

Applicability: Template Method

- To implement the **invariant** parts of an algorithm **once** and leave it up to subclasses to implement the behavior that can **vary**

- To localize common behavior among subclasses and place it in a common class to **avoid code duplication.**
  - Classic example of ”code refactoring”
  - the general workflow of the algorithm is implemented once in the abstract class's template method,
  - and necessary variations are implemented in the subclasses.

Applicability: Template Method

- To implement the **invariant** parts of an algorithm **once** and leave it up to subclasses to implement the behavior that can **vary**
- To localize common behavior among subclasses and place it in a common class to **avoid code duplication.**
  - Classic example of ”code refactoring”
- To control how subclasses extend superclass operations.
  - You can define a template method that calls "hook" operations at specific points, hence permitting extensions only at those points.

Example: Hook method

```
class Travel{
  public:
       void itenary(int days){
            addTransport();
            planDay(1);
            addHotel();
            planDay(days-1);
       }
      virtual void addHotel(){}
      virtual void addTransport()=0;
      virtual void planDay(int
day)=0;
    //…
}
```

- AirTravelPackage subclass must override `addTransport` and `planDay`.
- Adding a hotel is **optional**
- Extension to the workflow by only adding the hotel and it will work only after the first day.

Hollywood principle

- Template method uses **inverted control** structure.
  - from the superclass point of view:

    “Don't call us, we'll call you”.

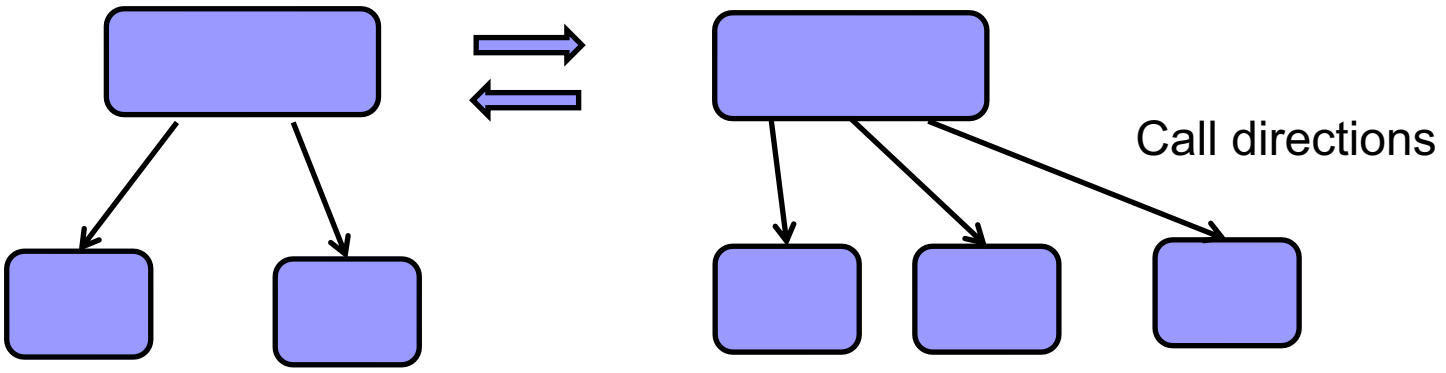
- Instead of calling the methods of base class inside the subclass methods,
  - **No** super.doA()
- the methods of subclass are called in the template method from superclass.

Hollywood principle

- “Don’t call us, we’ll call you”
- Seen this in frameworks and libraries
  - The framework carries out a computation/algorithm
  - Some parts are not specified. Your job is to implement those parts.
  - Your code does not call the framework functions, but the framework calls you
  - Opposite of using library functions

Hollywood Principle

- Reduces coupling between lower and upper level object
  - Lower level object do not call upper levels unless they are called
  - upper level objects call each other and lower levels



Consequences -Pros

- Reduces code duplication
- Enables customization of an algorithm in the subclasses
- Decouple lower and upper level objects
- Controls the extension of the workflow (hooks)
  - controls the point(s) at which specialization is permitted
- *Cannot* change the algorithm drastically

Consequences -Cons

- *Cannot* change the algorithm drastically
- LSP Risk: May violate Liskov substitution principle
  - If a subclass is used where a base class is expected, it may behave poorly due to its implementation of primitive operations.
- Too many Primitive Operations
  - make subclassing tedious and error-prone: Maintenance
  - Subclasses may be forced to implement methods they don't need.
- Beware of Inheritance
  - Changes in base class may unintentionally affect all subclasses.
- Hidden Dependencies
  - Subclasses must understand the order in TM

Example LSP violation

```
public class MaliciousXMLReader extends SysemSpecReader {
    // VIOLATION: This primitive operation breaks the implicit contract.
   public Object readFile() {
        // Instead of reading the file, it unexpectedly throws a low-level error.
         throw new IllegalStateException("File system access denied!");
    }

    // These methods must still be implemented, but will never be reached
public Object parseFile(Object fileObj) { return null; }
public String extractOSversion(Object parsedObj) { return null; }
}
```

Related patterns

- Template Method vs Factory Method
  - Factory method is a special primitive operation
- Template Method vs Strategy (next)
  - TM: Skeleton of algorithm where realization of some steps are deferred
    - One algorithm but implementing the step varies
  - Strategy: interchangeable algorithms
    - Different algorithms for the same purpose

Best Practices

- Keep the number of primitive operations minimal and focused.
- Document the expected behavior and call order of hooks and primitives.
- Prefer composition if subclassing becomes too complex.
- Testing
  - Test the template method independently to verify the algorithm structure.
  - Use mock implementations of primitive operations to isolate behavior.

Known uses

- Libraries and frameworks (e.g. Swing)
- All non-abstract methods of `java.io.InputStream`, `java.io.OutputStream`, `java.io.Reader` and `java.io.Writer`.
- All non-abstract methods of `java.util.AbstractList`, `java.util.AbstractSet` and `java.util.AbstractMap`.

- Template Method can be recognized if you see a method in base class that calls a bunch of other methods that are either abstract or empty.

https://refactoring.guru/design-patterns/template-method/java/example

From java.io.InputStream source code:

```
public int read(byte[] b, int off, int len) throws IOException
{
  if (off < 0 || len < 0 || b.length - off < len)
    throw new IndexOutOfBoundsException();

  int i, ch;

  for (i = 0; i < len; ++i)
    try
  {
    if ((ch = read()) < 0)
      return i == 0 ? -1 : i;          // EOF
    b[off + i] = (byte) ch;
  }
    catch (IOException ex)
  {
    // Only reading the first byte should cause an IOException.
    if (i == 0)
      throw ex;
    return i;
  }

  return i;
}
```

```
public abstract int read() throws IOException;
```

MORE ALGORITHMS

Motivating Example

- Users log in with password. We need to save these passwords in a DB in an encrypted form.
- Currently we use 3 encryption algorithms.
  - Generate hash using one of them and then save in DB
- Problem: How to write setPassword to support both encryption?
  - Later I may embed new encryptions

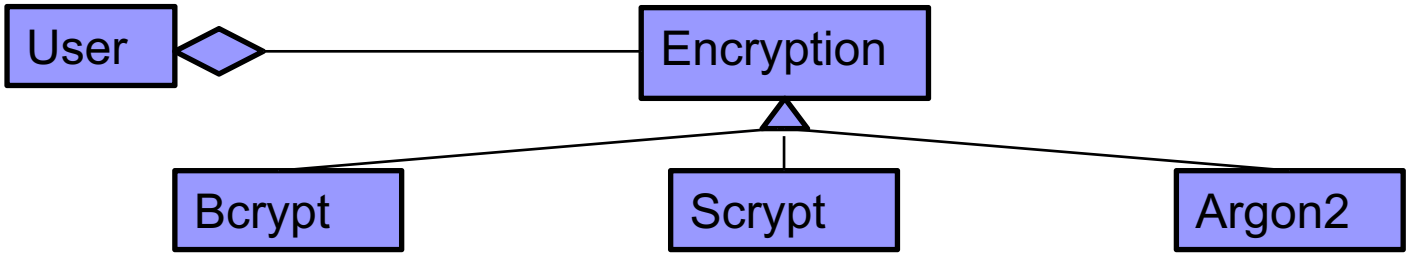
```
class User{
    public:  virtual bool setPassword(const String&
passwd);
                    virtual bool checkPassword(const String&
```

How to implement setPassword?

- Soln1: conditionals in setPassword and getPassword method
  - Switch and long ifs are not extensible
- Soln2: inheritance
  - Abstract User and Subclasses override setPassword and getPassword
    - Too many subclasses! One for each encryption
    - Should I use inheritance just to override 1 method?
- Soln3: encapsulate what varies
  - What is varying?

How to implement setPassword?

- What is varying? A function realization
  - Put it in a class -- a class for generateHash()
  - Choose the suitable function at runtime



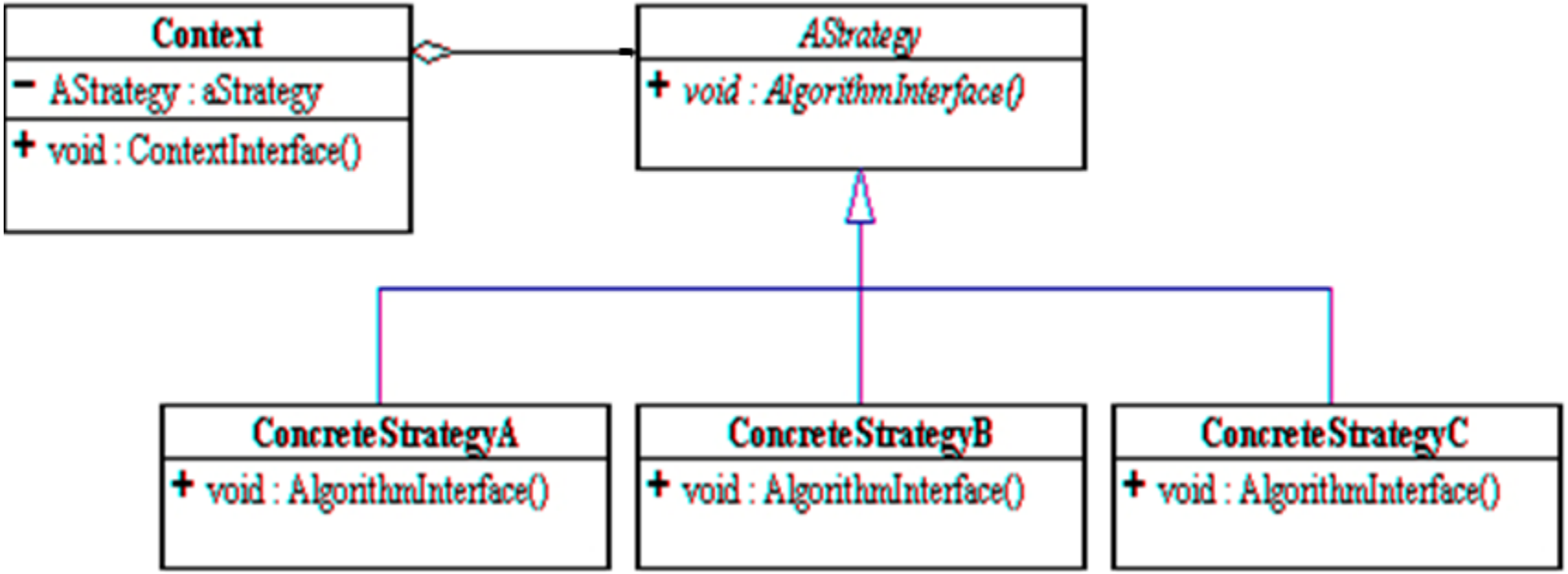
- Choose a suitable one from the **algorithm family** at runtime
  - Delegation instead of inheritance

Strategy

- **Intent**:
  - Define a family of algorithms, **encapsulate** each one, and make them interchangeable.
  - Strategy lets the algorithm vary independently from clients that use it.
- A.k.a. Policy

Strategy -Structure

- **Participants?**



setPassword with Strategy

Participants?

```
class User{
 public:
    virtual void setPassword(const String& passwd);
    virtual bool checkPassword(const String&
passwd);
     User(Encryption* e); ~User()=default;
private:     Encryption* strategy;
….}
void User::setPassword(const String& pass){
          checkStrength(pass);
          save(strategy ->generateHash(pass));
}
User::User(Encryption* e):strategy(e){…}
```

Strategy - Collaborations

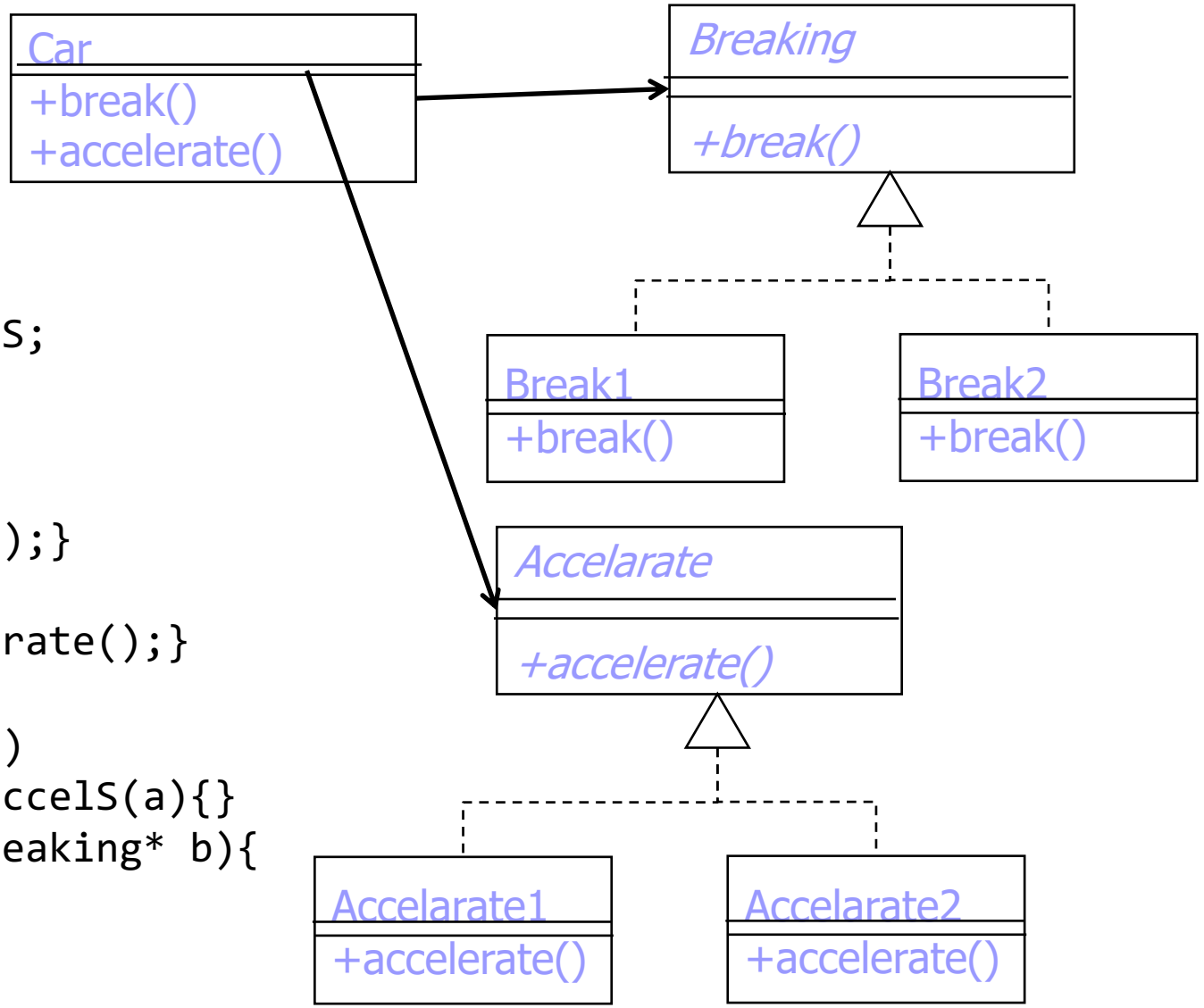
- The context object receives requests from the client and delegates them to the strategy object.
  - Usually, the ConcreteStrategy is created by the client and passed to the context. From this point the clients interacts only with the context.
- The Context objects contains a reference to the ConcreteStrategy 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, additional objects can be defined to pass data from context object to strategy.

Recall the Car example 2nd week

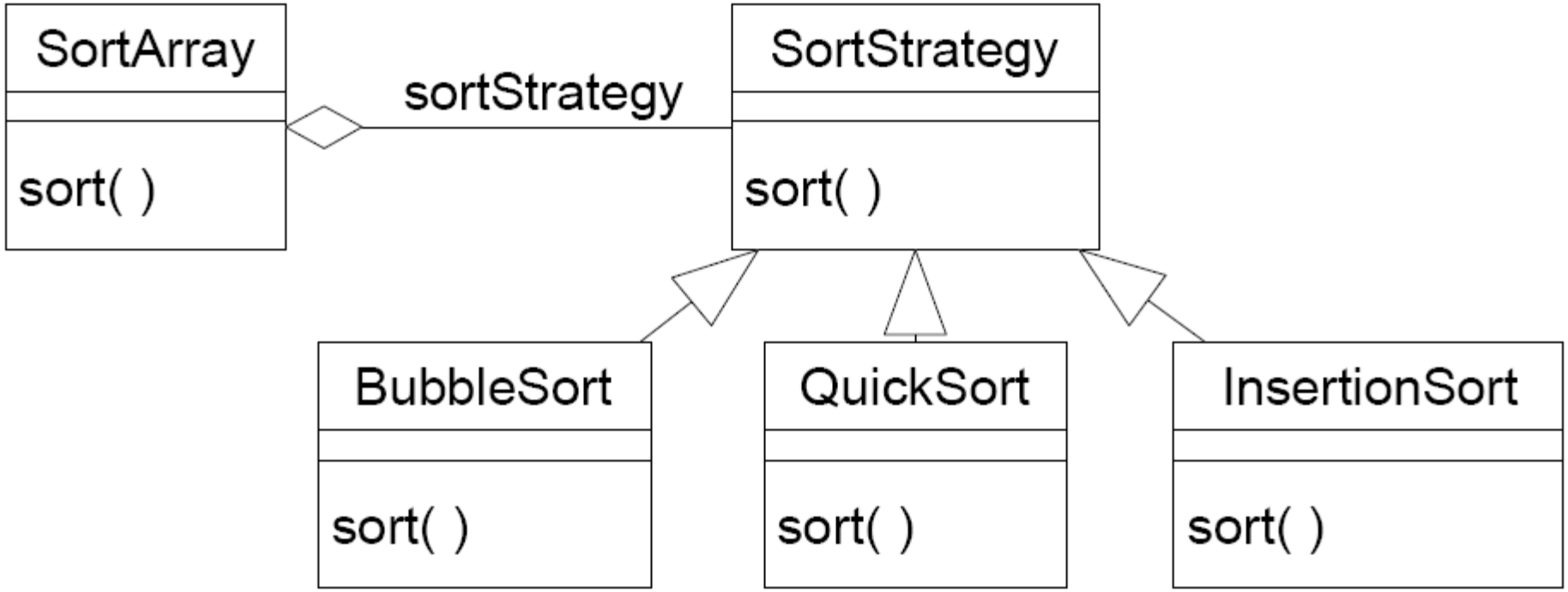
- Car class and its 2 operations/behaviors:
  - Brake and accelerate
- These behaviors change frequently between models, so implement these behaviors in subclasses: overriding
  - For each new model, override
    - Beware: Code duplication across models
    - The work of managing these behaviors increases greatly as the number of models increases

Strategy Solution

```
class Car{
private:
Breaking* breakS;
Accelerate( accelS;
…
public:
break(){
   breakS->break();}
accelerate(){
   accelS->accelerate();}
Car(Breaking* b,
    Accelerate* a)
    :breakS(b), accelS(a){}
void setBreakS(Breaking* b){
    breakS=b;}
…
}
```



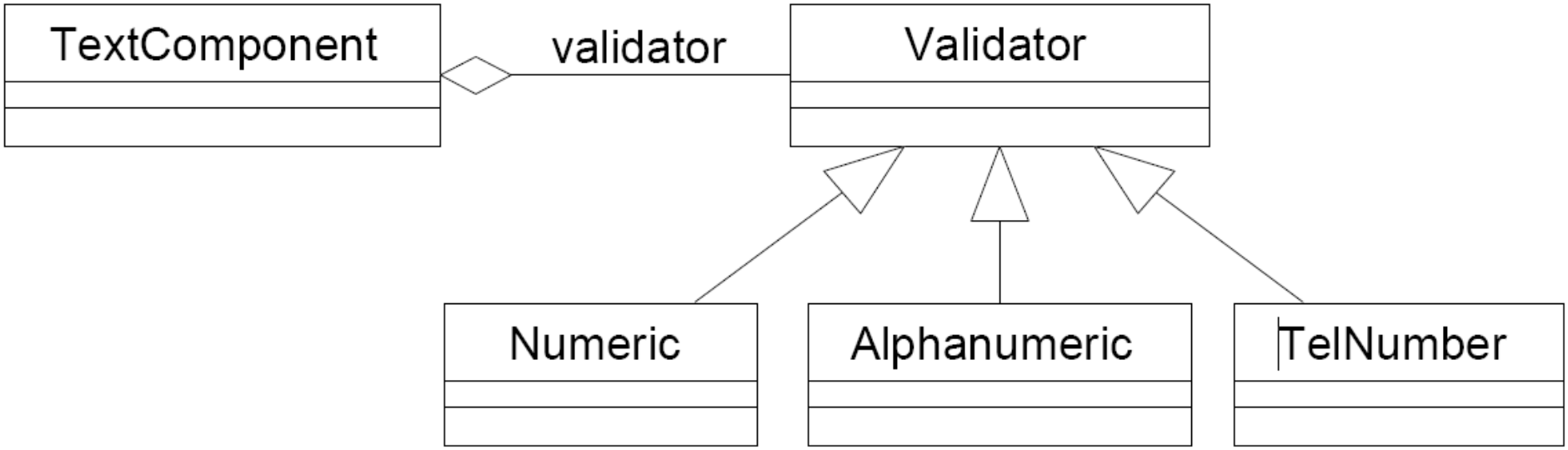
Example 3: Strategy



Application where the sorting algorithm is chosen at runtime

Example 4

- A GUI text component object wants to decide at runtime what strategy it should use to validate user input.
- Many different validation strategies are possible: numeric fields, alphanumeric fields, telephone-number fields, etc.

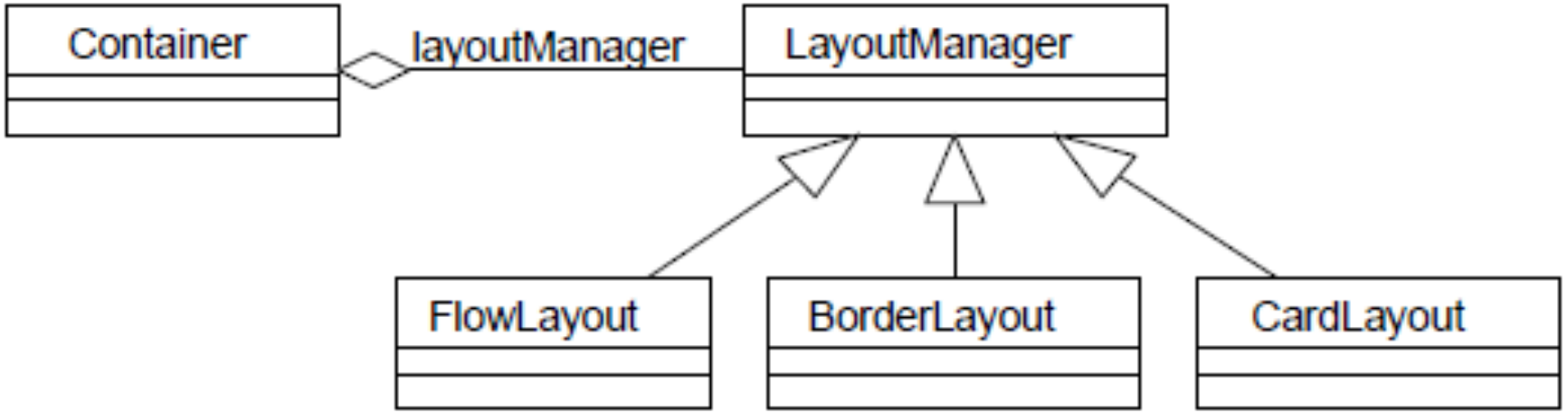


Applicability -Strategy

- Many related classes differ only in behavior
  - you can reduce these several objects to one class that uses several Strategies.
- Need different variants of an algorithm
  - switch from one algorithm to another during runtime
- Avoid exposing complex, algorithm specific data structures
  - E.g. intermediate data structure used for compression
- A class defines many behaviors and chooses one with conditionals

Example 5 – javax.swing

- 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.




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

Example 6

- There are numerous border types
  - Line, titled, edged,…
- Each visual component draws itself with different bordering (assume they are not decorated)
- All they differ is border drawing algorithm
  - There is a family of border drawing algorithm

Example

```
class JComponent{….
 protected void paintBorder(Graphics g) {
  switch(getBorderType()) {
   case LINE_BORDER:    paintLineBorder(g); break;
   case ETCHED_BORDER: paintEtchedBorder(g); break;
   case TITLED_BORDER: paintTitledBorder(g); break;
   ...}
}
```

// The actual implementation of the JComponent.paintBorder() method

```
protected void paintBorder(Graphics g) {
  Border border = getBorder();
   if (border != null) {
     border.paintBorder(this,g,0,0,getWidth(),
  getHeight()); }
}
```

Border Object has the drawing algorithm not the JComponent

Strategy – Consequences-1

- Eliminates conditional statements
  - When you have several behaviors together in one class, you'll use conditionals
  - With Strategies you won't need to check for anything, since whatever the current strategy is just executes without asking questions
- Alternative to subclassing context object to achieve different behaviors
  - With Strategies all you need to do is switch the context's strategy and it will immediately change how it behaves.

Strategy – Consequences-2

- Change the algorithm on the fly
- We can introduce new strategies without having to change the context. (OCP)
- Strategies can be shared
- Increases number of objects and communication overhead btw strategy and context
  - See the implementation issues-2

Implementation issues-1

- Lambda functions instead of explicit strategy classes
  - Cons: strategies will not be shared
  - Pro: less number of classes

```
class Car {
public:
    std::function<void()> strategy;
    Car(std::function<void()> strategy) : strategy(strategy) {}
    void applyBrakes() {  strategy(); }
};
int main() {
    Car sportsCar( { std::cout << "Applying ABS brakes...\n"; });
    sportsCar.applyBrakes();
}
```

Lambda

```
interface BrakingStrategy {  void applyBrakes();}
class Car {
   private BrakingStrategy strategy;
    public Car(BrakingStrategy strategy) {  this.strategy = strategy;   }
    public void applyBrakes(){    strategy.applyBrakes();    }
}
public static void main(String[] args) {
    Car sportsCar = new Car(() ->
                                System.out.println("Applying ABS brakes..."));
   sportsCar.applyBrakes();
}
//body of lambda implements a single abstract method
```

Implementation issues -2

- Data from Context to Strategy
  - Example: sorting strategy needs data to sort
  1. Pass data as parameter
     - Not all strategies may need the same data
  2. Pass the context reference
     - Strategy would query context what data it needs
     - Context's interface with getters
     - Strategy is coupled with context
  - There is no best

Implementation issues-3

- Default Strategy in the Context
  - Benefit: Clients don't have to deal with Strategy objects at all *unless* they don't like the default behavior
- Make stateless Strategies
  - When Strategies are stateless, they can be shared
    - e.g. 3 users share Bcrypt for hashing their password
  - No need to clutter the memory with same objects
  - Strategies make good flyweights

Implementation issues-4

- Using C++ templates : Cannot change strategy

```
template <class AStrategy>  class Context {
        void operation() { theStrategy.DoAlgorithm(); }
    private:
        AStrategy theStrategy;
};
//The class is then configured with a Strategy class when it's
instantiated:
class MyStrategy {
        public: void DoAlgorithm();
};
Context<MyStrategy> aContext;
```

Strategy: Key notes

- Family of algorithms
  - One strategy class per algorithm
- Make them interchangeable
  - Just change the current strategy object and the algorithm changes
- Client of the context can use different algorithms easily
- Alternative for subclassing

Question: why not delete?

```
class User{
 public:
     virtual void setPassword(const String& passwd);
     virtual bool checkPassword(const String&
passwd);
      User(Encryption* e); ~User()=default;
private:      Encryption* strategy;
….}
void User::setPassword(const String& pass){
          checkStrength(pass);
          save(strategy ->generateHash(pass));
}
User::User(Encryption* e):strategy(e){…}
```

Strategies are shared: flyweight

Strategy -Related patterns -1

- Strategies can be **flyweights**
- **Template M** vs **Strategy**
  - **Template Method** fixes skeleton of algorithm, variation in steps
  - **Strategy** changes the algorithm completely
    - Change mergesort to bublesort
- **Decorator** vs Strategy
  - Both alternative to inheritance
  - Decorator adds functionality on top -wrapper
  - Strategy replaces functionality

Strategy -Related patterns -2

- **Bridge** and Strategy have the same Class diagram, but intent is different
  - strategy is related with the behavior and bridge is for structure.
  - the coupling between the context and strategies is tighter than the coupling between the abstraction and implementation in the bridge pattern.
- State pattern (next) vs Strategy
- Command (later)

Strategy –Known Use

- java.util.Comparator with compare() method is a strategy used by many, e.g. sort method of Collections