



Design Patterns

宋 杰

Song Jie

东北大学 软件学院

Software College, Northeastern
University



15. Template Method Pattern

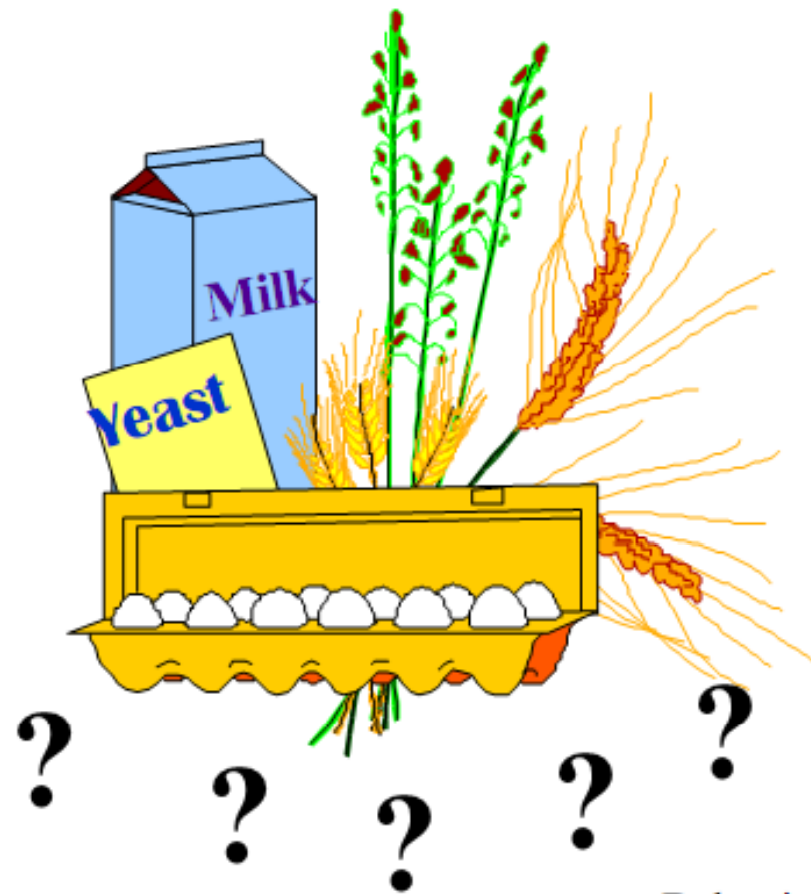


Intent

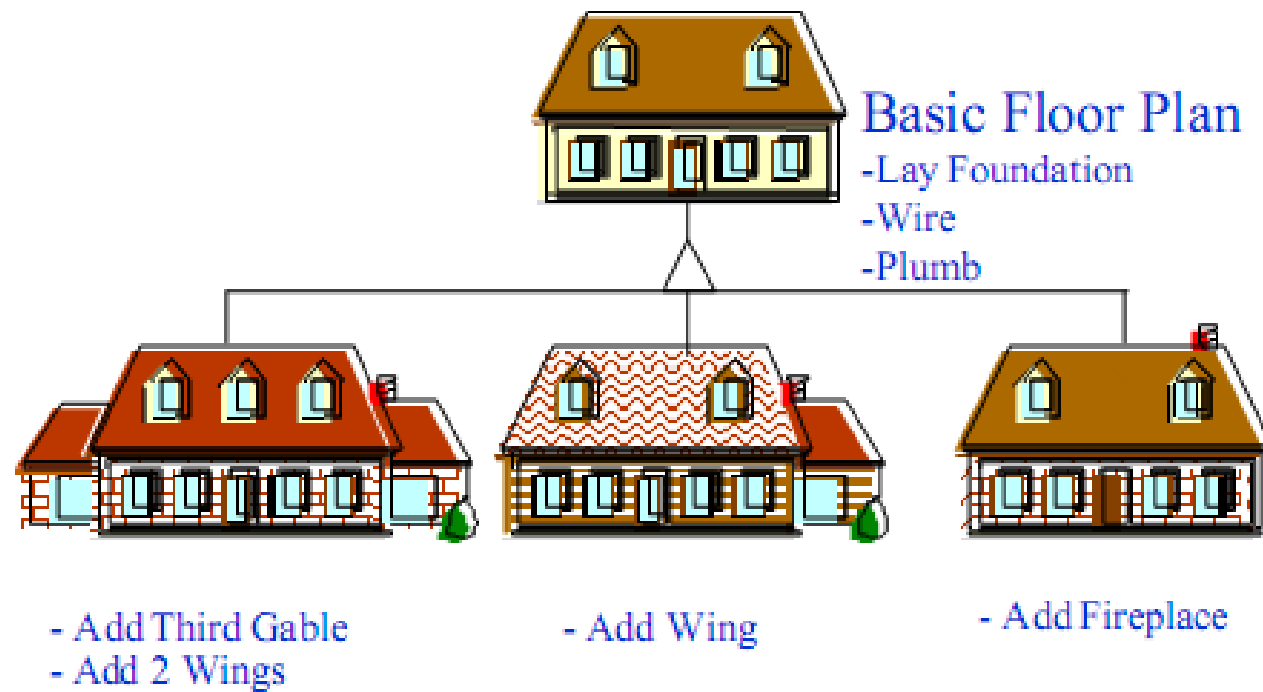
- Define the **skeleton** of an algorithm in an operation, deferring **some steps to subclasses**. Template Method lets subclasses **redefine certain steps of an algorithm without changing the algorithm's structure**.
 - **Java 实现**: 准备一个抽象类，将部分逻辑以具体方法以及具体构造子的形式实现，然后声明一些抽象方法来迫使子类实现剩余的逻辑。不同的子类可以以不同的方式实现这些抽象方法，从而对剩余的逻辑有不同的实现。
-

Example

- Once a basic bread recipe (烹饪法) is developed, additional steps, such as adding cinnamon(肉桂), raisins(葡萄干), nuts, peppers(胡椒粉), cheese, etc. can be used to create different types of bread.



Example



Variations added to Template Floor Plan


Example

```
abstract class Sorter {  
    // template method  
    public final void sort(List<Object> target) {  
        int length = target.size();  
        for (int i = 0; i < length; i++) {  
            Object former = target.get(i);  
            for (int j = i + 1; j < length; j++) {  
                Object later = target.get(j);  
                // call the primitive operations  
                if (isIncreasedSort() != isFormerLessThanLater(former, later)) {  
                    // switch elements  
                    target.set(i, later);  
                    target.set(j, former);  
                    former = later;  
                }  
            }  
        }  
    }  
  
    protected abstract boolean isIncreasedSort();  
    protected abstract boolean isFormerLessThanLater(Object former, Object later);  
}
```

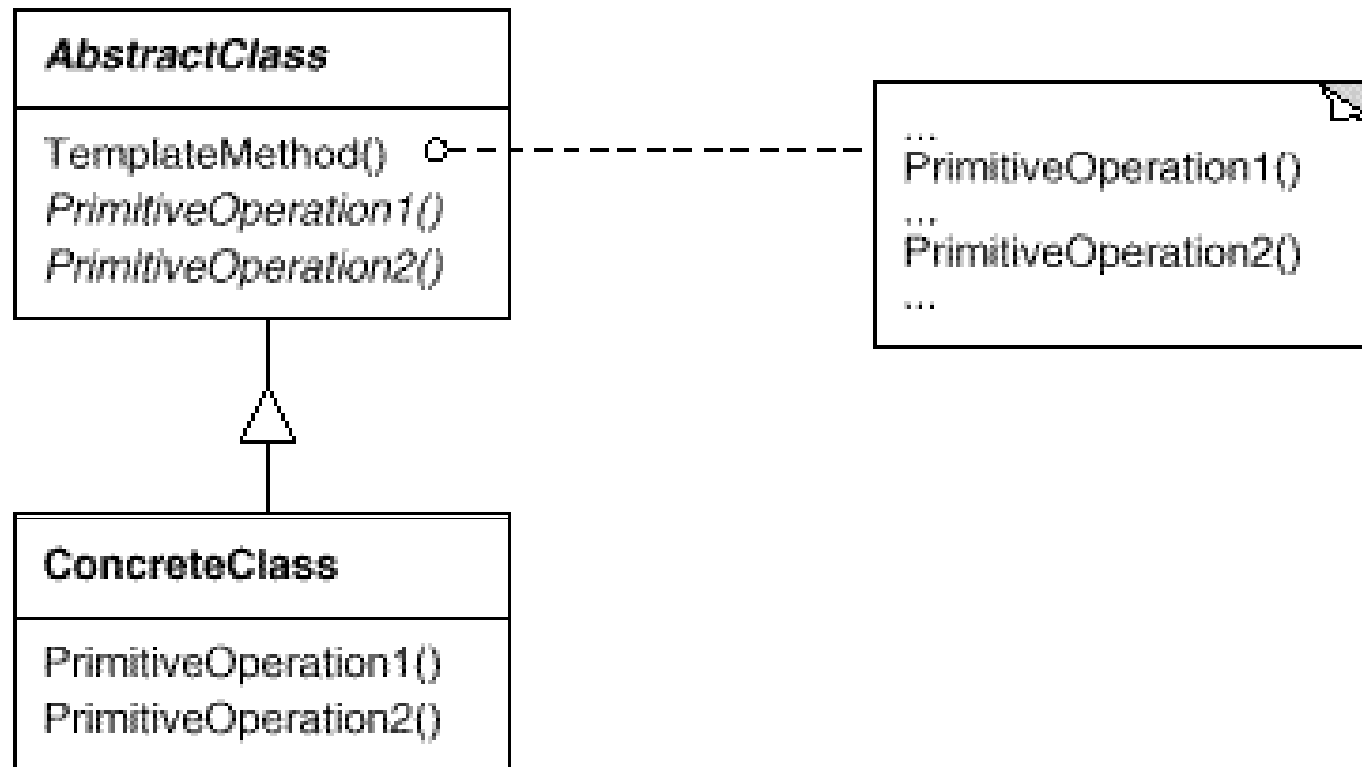


Example

```
class TextSorter extends Sorter {  
    protected boolean isIncreasedSort() {  
        return true;  
    }  
  
    protected boolean isFormerLessThanLater(Object former, Object later) {  
        return former.toString().compareTo(later.toString()) < 0;  
    }  
}
```



Structure





Participants

■ AbstractClass

- Defines abstract **primitive operations** that concrete subclasses define to implement steps of an algorithm.
- Implements a **template method** defining the skeleton of an algorithm. The **template method** calls primitive operations as well as operations defined in **AbstractClass**.

■ ConcreteClass

- Implements the primitive operations to carry out subclass-specific steps of the algorithm.
-



Consequences

- Template methods are a fundamental technique for **code reuse**.
 - Template methods are particularly important in class libraries, because they are the means for factoring out (分解) common behavior in library classes.
-



Hollywood principle

- Template methods lead to an inverted control structure (DIP) that's sometimes referred to as "the Hollywood principle," that is
 - "Don't call us, we'll call you".
 - This refers to how a parent class calls the operations of a subclass and not the other way around.
-



Applicability

- To implement the invariant parts of an algorithm once and leave it up to subclasses to implement the behavior that can vary.
 - Refactoring to generalize.
 - When common behavior among subclasses should be factored and localized in a common class to avoid code duplication.
 - Control subclasses extensions.
 - You can define a template method that calls "hook" operations at specific points, thereby permitting extensions only at those points.
-



Implementation 1: Naming conventions

- Identify the operations that should be overridden by adding a prefix to their names.
 - For example, the prefixes template method names with "do-": "doCreateDocument()", "doRead()"
-



Implementation 2: Using access control

■ In Java

- The **primitive operations** can be declared as **protected** and **abstract** method;
 - The **template method** can be declared as **final** method.
-



Implementation 3: Minimizing primitive operations

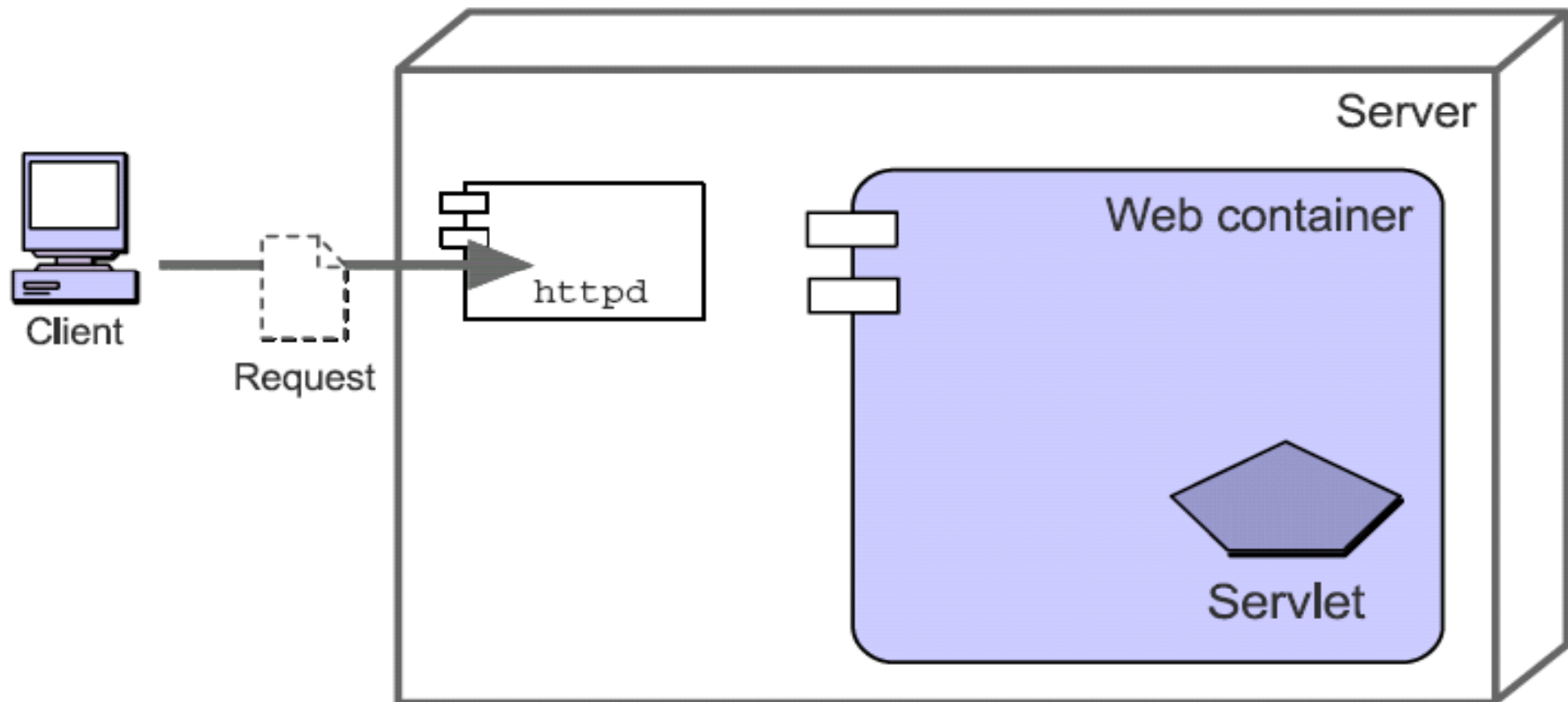
- An important goal in designing template methods is to minimize the number of primitive operations that a subclass must override to flesh out the algorithm.
 - The more operations that need overriding, the more tedious things get for clients.
-



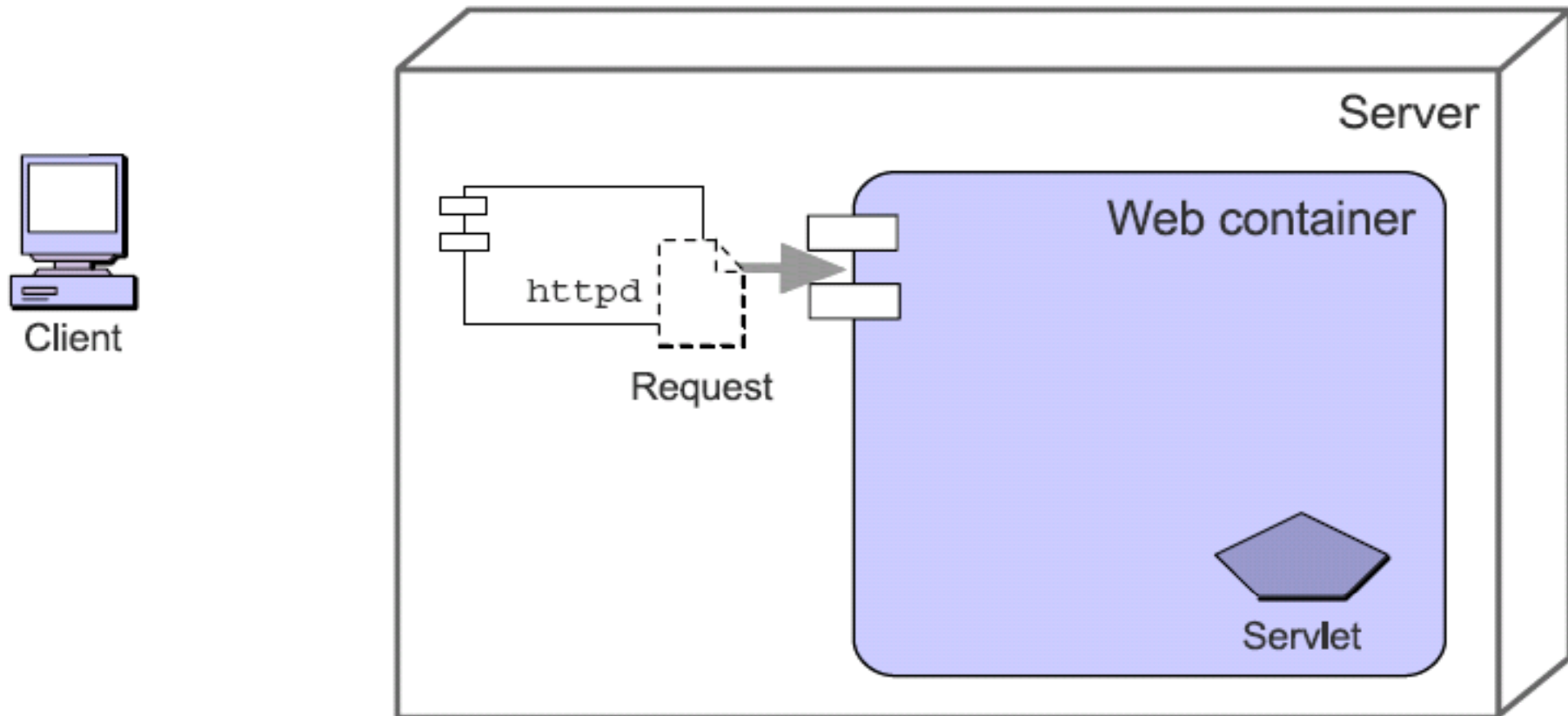
Example: `HttpServlet` in JavaEE

- `HttpServlet` use `Template Method Pattern` a lot;
 - The subclass of `HttpServlet` is used to process the http request in different according to it request method (type).
-

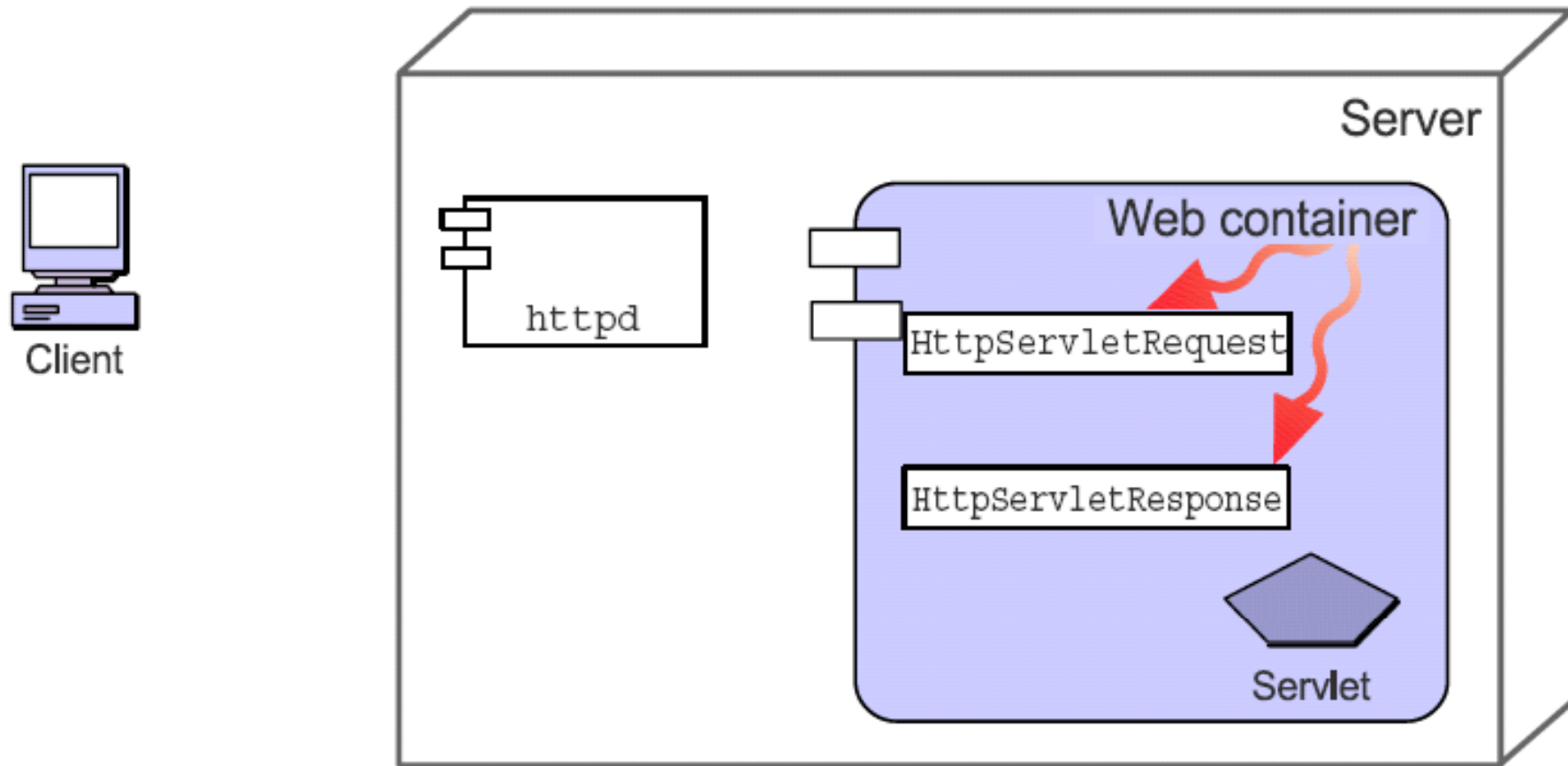
Step 1: Client send the request



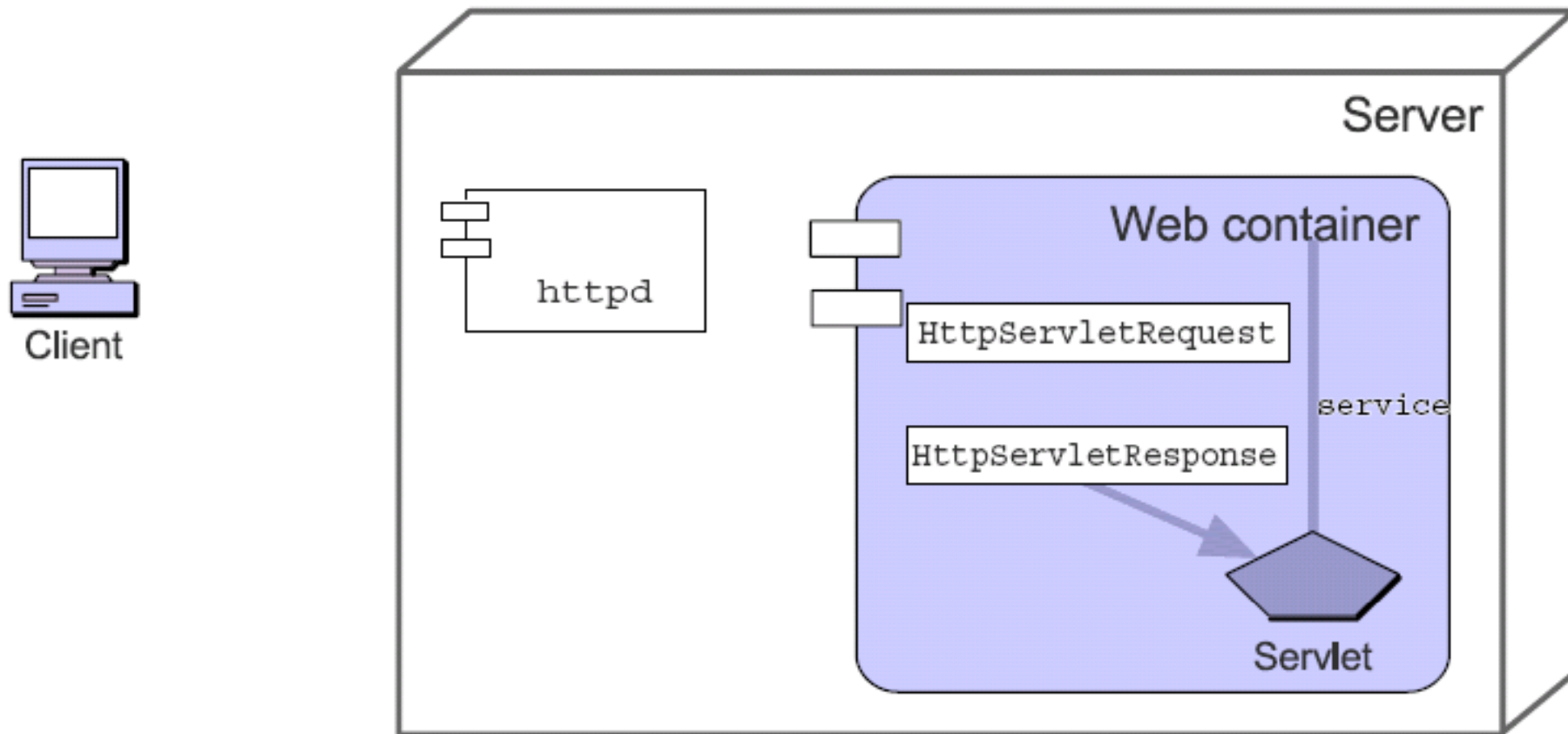
Step 2: Web server send the request to the Web container



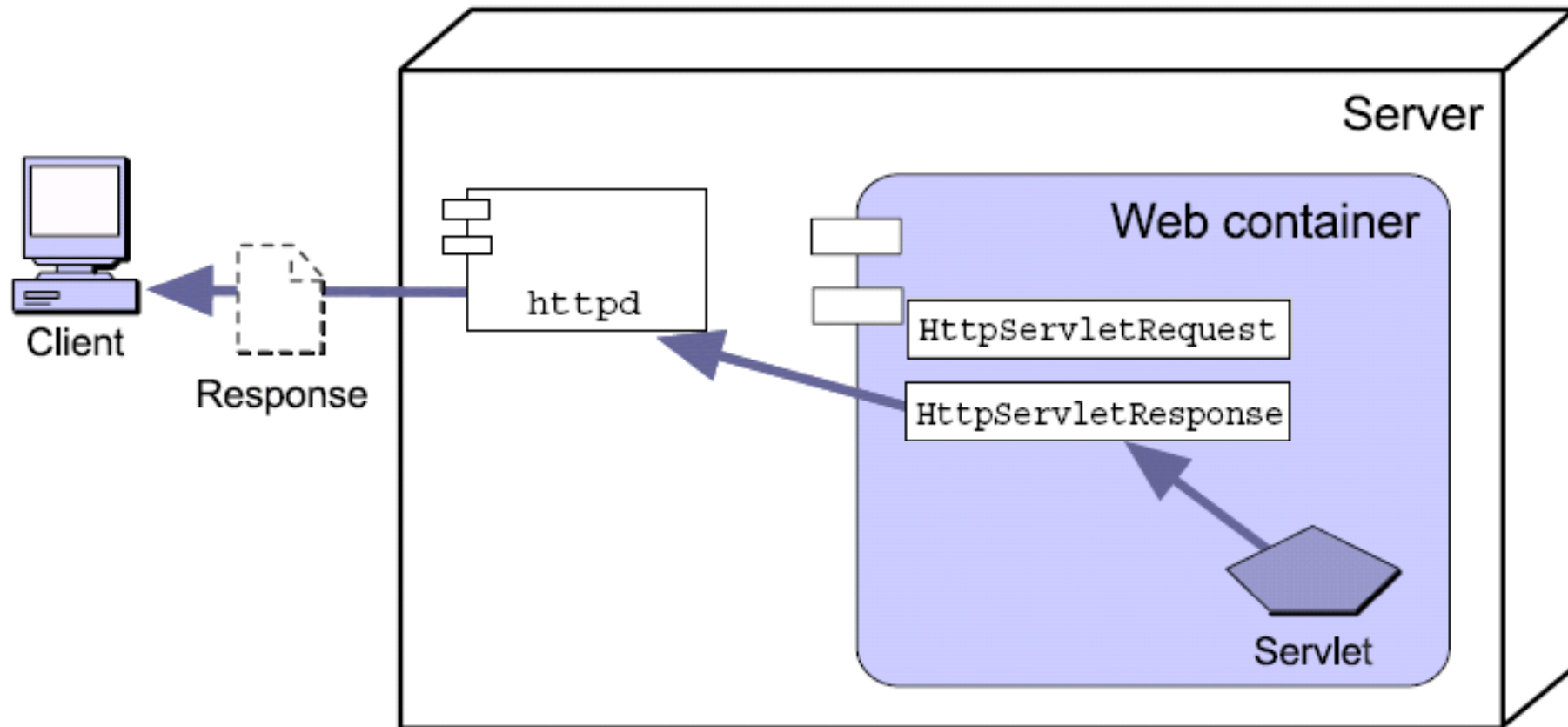
Step 3: Web container initializes the request and response object



Step 4: Web container invokes the Servlet according to the request URL



Step 5: Web container return the response, which is modified by the target Servlet, to the client



HttpServlet

javax.servlet.http

<<interface>>
HttpServletRequest

<<interface>>
HttpServletResponse

Request

Response

HttpServlet

service
doGet
doPost

The service method dispatches the call to either the doGet or doPost method based on the HTTP method of the request.

MyServlet

doGet
doPost

Your servlet class should override either the doGet or the doPost method based on the expected HTTP request method.



doXxx method in HttpServlet

- There are many types of request.
 - In service method, the types of request is determined, and corresponding *doXxx* method is invoked
 - *doGet*
 - *doPost*
 - *doPut*
 - *doDelete*
 - *doOptions*
 - *doTrace*
-

```

if (method.equals(METHOD_GET)) {
    long lastModified = getLastModified(req);
    if (lastModified == -1) {
        // servlet doesn't support if-modified-since, no reason
        // to go through further expensive logic
        doGet(req, resp);
    } else {
        long ifModifiedSince = req.getDateHeader(HEADER_IFMODSINCE);
        if (ifModifiedSince < (lastModified / 1000 * 1000)) {
            // If the servlet mod time is later, call doGet()
            // Round down to the nearest second for a proper compare
            // A ifModifiedSince of -1 will always be less
            maybeSetLastModified(resp, lastModified);
            doGet(req, resp);
        } else {
            resp.setStatus(HttpServletResponse.SC_NOT_MODIFIED);
        }
    }
}

} else if (method.equals(METHOD_HEAD)) {
    long lastModified = getLastModified(req);
    maybeSetLastModified(resp, lastModified);
    doHead(req, resp);
} else if (method.equals(METHOD_POST)) {
    doPost(req, resp);
} else if (method.equals(METHOD_PUT)) {
    doPut(req, resp);
} else if (method.equals(METHOD_DELETE)) {
    doDelete(req, resp);
} else if (method.equals(METHOD_OPTIONS)) {
    doOptions(req, resp);
} else if (method.equals(METHOD_TRACE)) {
    doTrace(req, resp);
} else {

```




Extension 1: Method in template method pattern

- Template methods
 - Primitive methods
 - Concrete methods;
 - Abstract methods;
 - Hook methods
 - Which provide default behavior that subclasses can extend if necessary.
 - A hook operation often does nothing by default.
-



Extension 2: Refactor in template method pattern

- Defining a class according to its **behaviors**, not **states**. That is, the implementation of an class should firstly base on its behaviors instead of its states
 - An exception is the value object;
 - Using **abstract** state instead of **concrete state** for implementations.
 - Using **indirect reference** instead of **direct reference**. That is, if a behavior involves a state of an object. Using access method instead of using property directly.
-



Extension 2: Refactor in template method pattern

- **Layering the operations.** The behaviors of an class should be distributed in a group of kernel methods (primitive methods) , thus they can be replaced easily in the subclass。
 - **Postponing the implementation of the state in the subclass.** Do not declare the properties in the abstraction (try to using interface and abstract class to present the abstraction, instead of concrete class)。
 - In the abstraction, if the states are defined, **using abstract accessed methods** to access the states, and let subclass implements them.
-



Let's go to next...