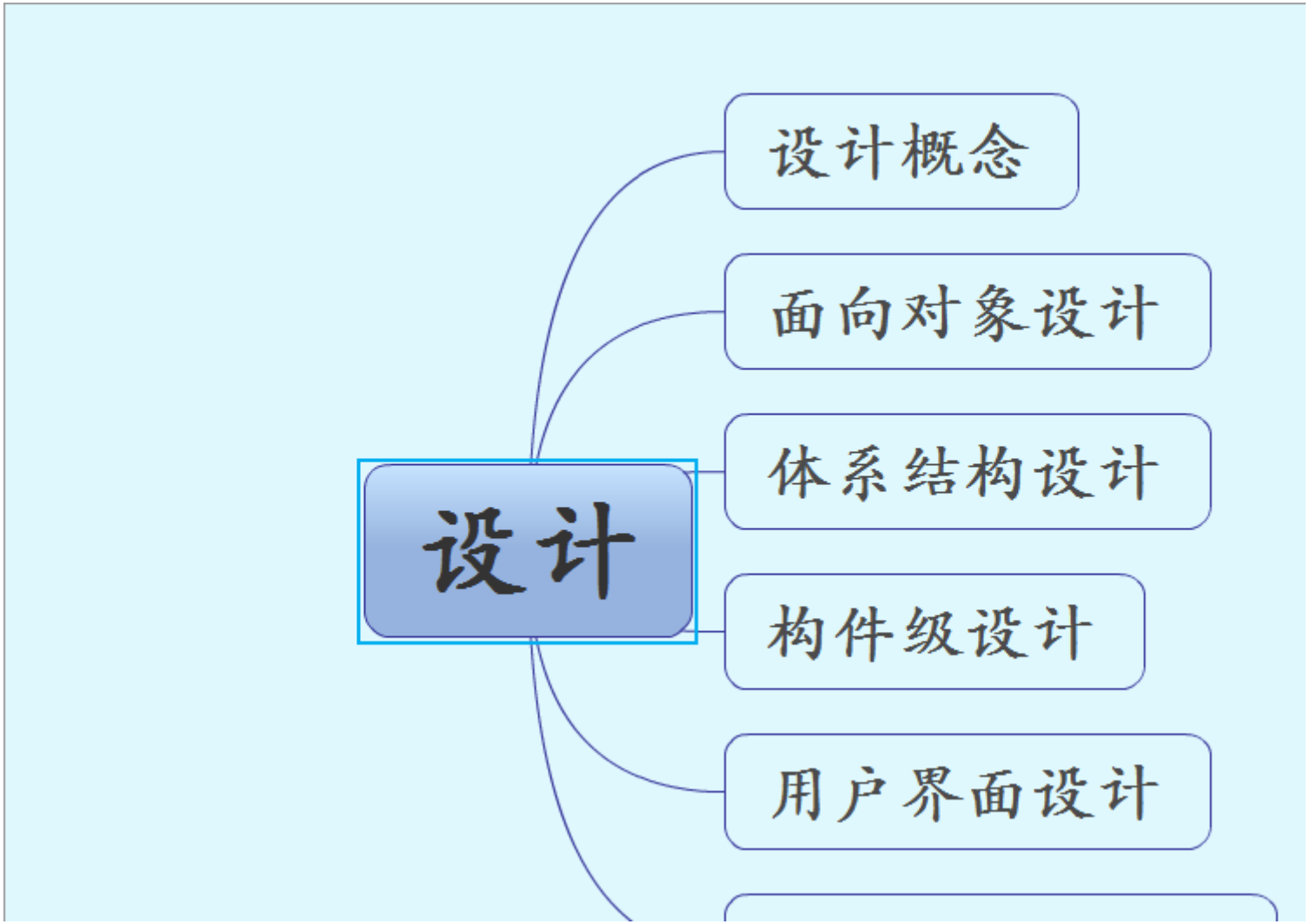
outline



概述

- 在设计过程中经常问这样一个问题

  “*I wonder if anyone has developed a solution for this?*”
- 对于清晰描述的一组问题，基于模式的设计通过查找一组已被证明有效的解决方案来创建新的应用系统。
- 起源于20世纪70年代
  - Alexander
  - 建筑设计模式

设计模式定义

- Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

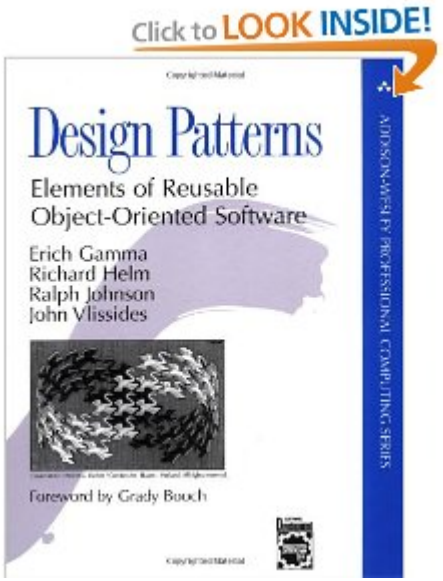
  ---------Chirstopher Alexander

- “每一个模式描述了一个在我们周围不断重复发生的问题，以及该问题的解决方案的核心。这样，你就能一次又一次地使用该方案而不必做重复劳动”

软件设计中的模式

- Erich Gamma、Richard Helm、Ralph Johnson、John Vlissides

  《Design Patterns—Elements of Reusable Software》
- Gang of Four (GOF) 23



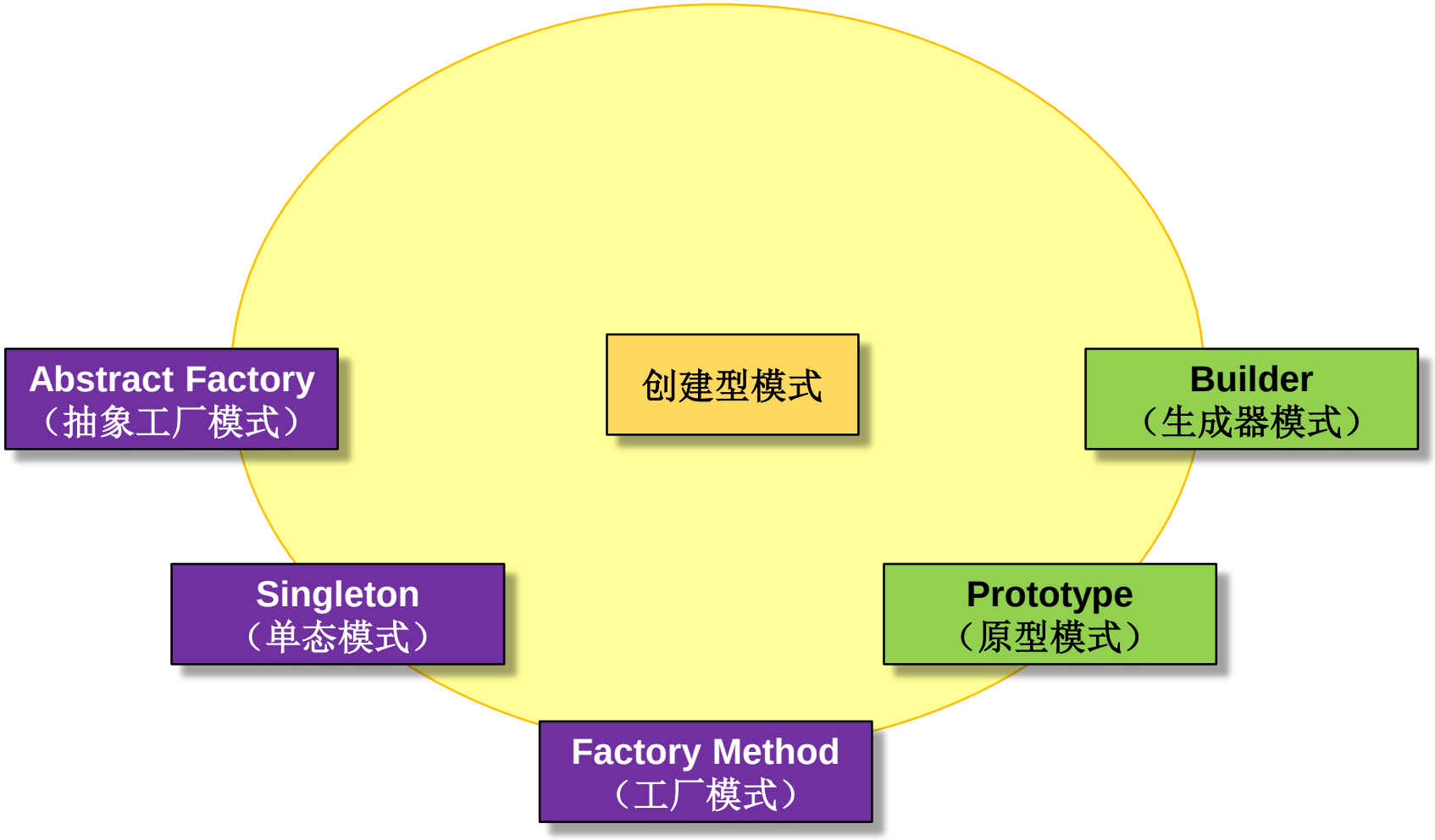
什么是设计模式

- 设计模式（Design pattern）是一套被反复使用、多数人知晓的、经过分类编目的、代码设计经验的总结。
- 使用设计模式是为了可重用代码、让代码更容易被他人理解、保证代码可靠性
- 整个设计模式贯穿一个原理：**面对接口编程，而不是面对实现**。目标原则是：**降低耦合,增强灵活性**。

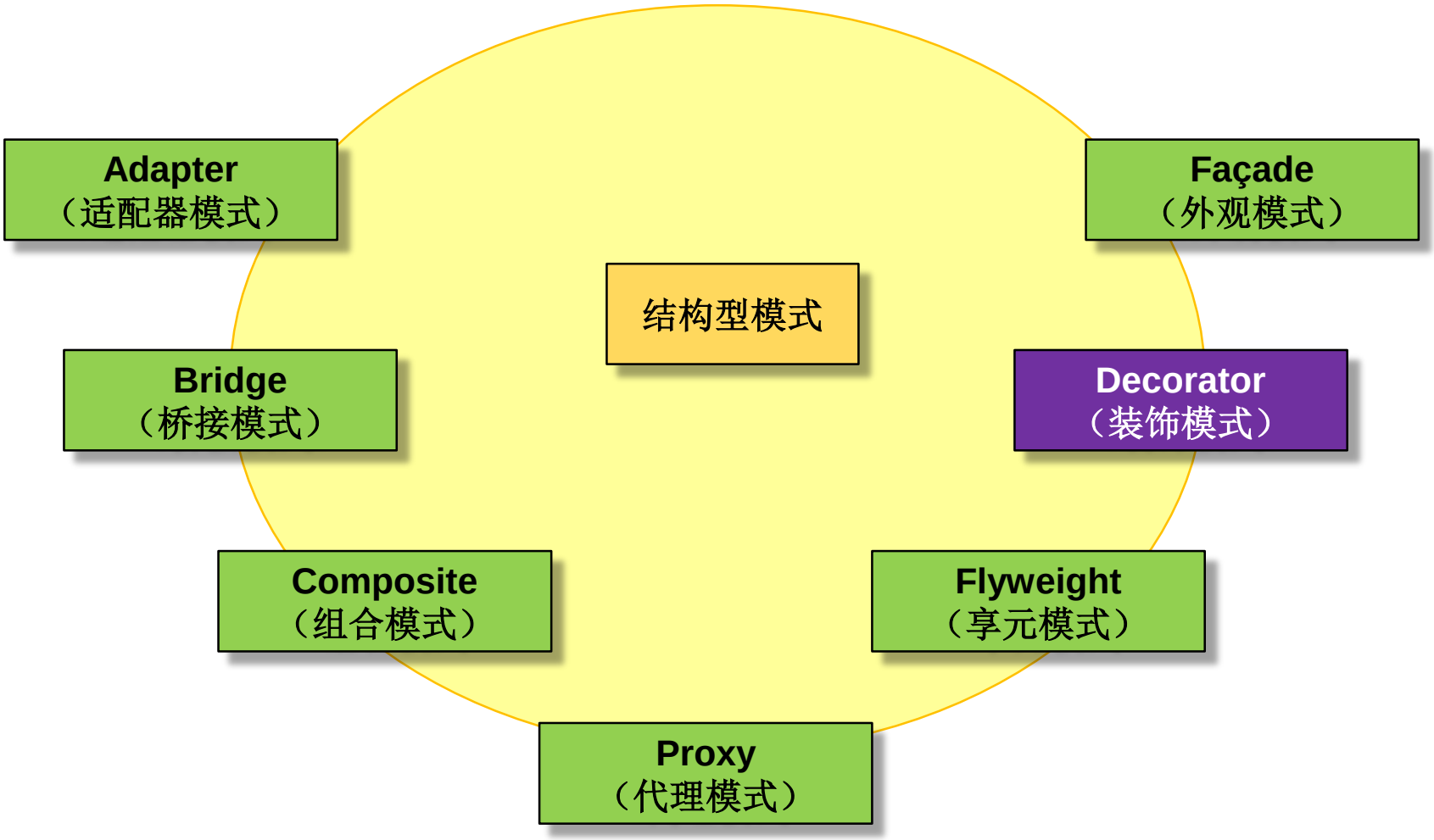
设计模式的分类

- 创建型模式（Creational Pattern）
  - 有关对象创建的模式
- 结构型模式（Structural Pattern）
  - 描述对象构造和组成的方式
- 行为型模式（Behavioral Pattern）
  - 描述一组对象交互的方式

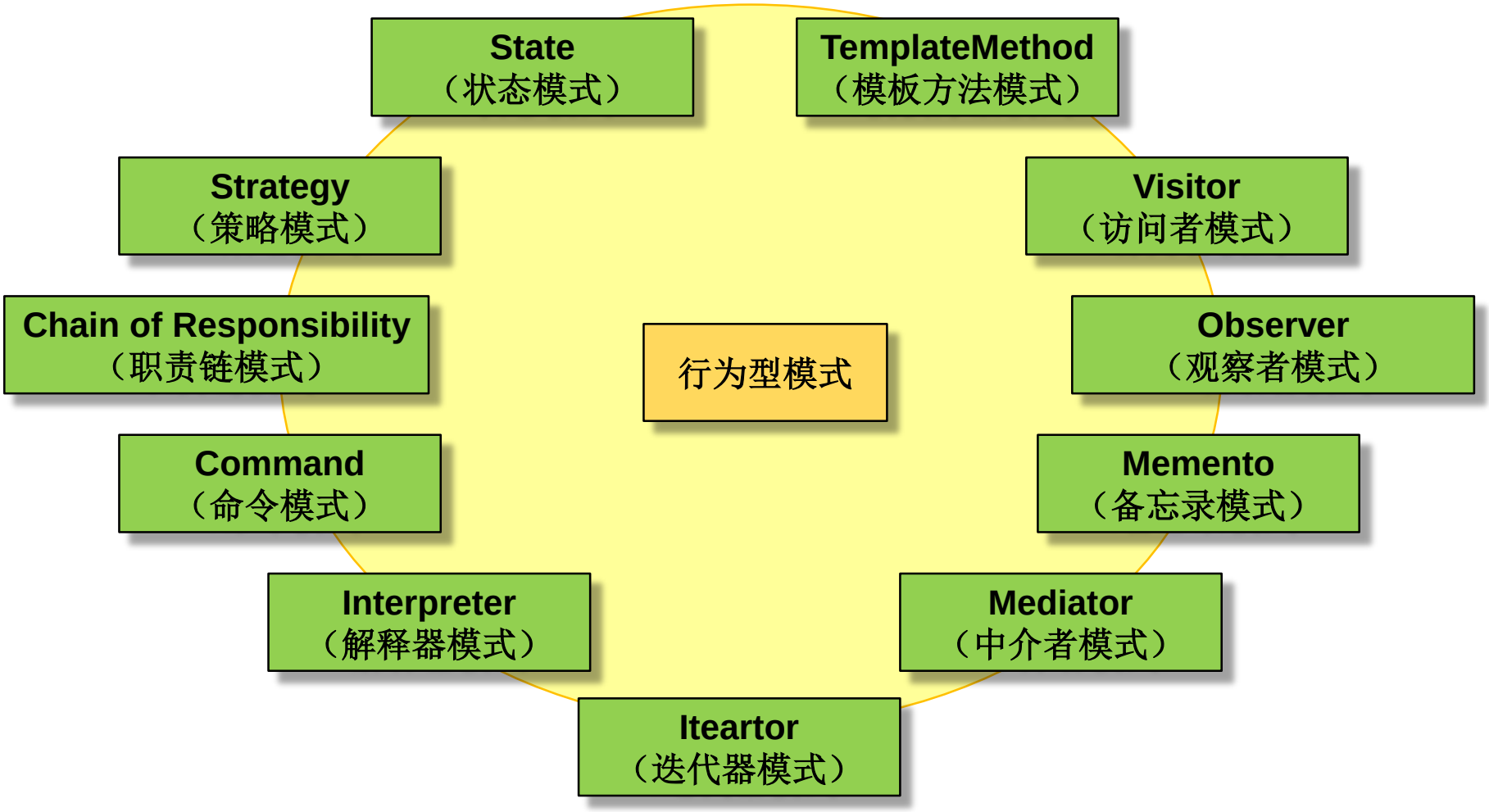
GOF 23


创建型模式
Abstract Factory
（抽象工厂模式）
Builder
（生成器模式）
Singleton
（单态模式）
Prototype
（原型模式）
Factory Method
（工厂模式）

GOF 23

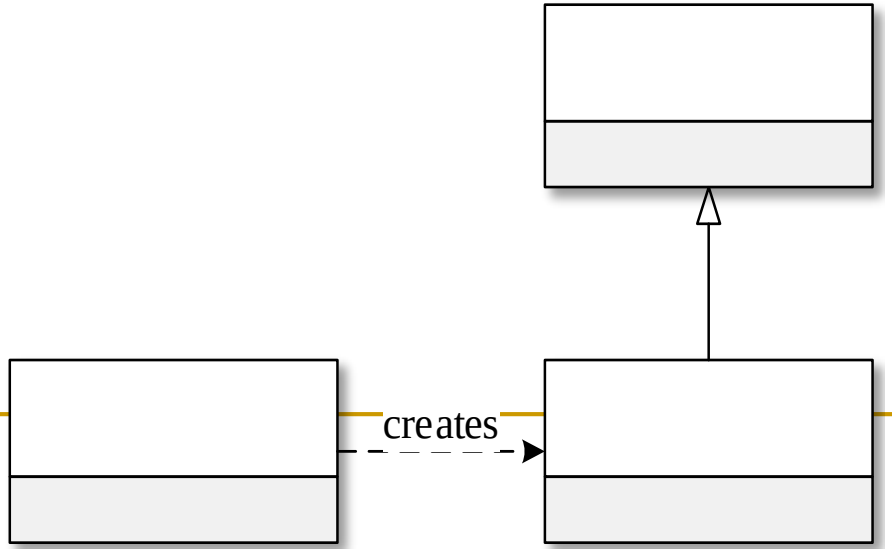

Adapter
（适配器模式）
Façade
（外观模式）
结构型模式
Bridge
（桥接模式）
Decorator
（装饰模式）
Composite
（组合模式）
Flyweight
（享元模式）
Proxy
（代理模式）

GOF 23


State
（状态模式）
TemplateMethod
（模板方法模式）
Strategy
（策略模式）
Visitor
（访问者模式）
Chain of Responsibility
（职责链模式）
行为型模式
Observer
（观察者模式）
Command
（命令模式）
Memento
（备忘录模式）
Interpreter
（解释器模式）
Mediator
（中介者模式）
Iteartor
（迭代器模式）

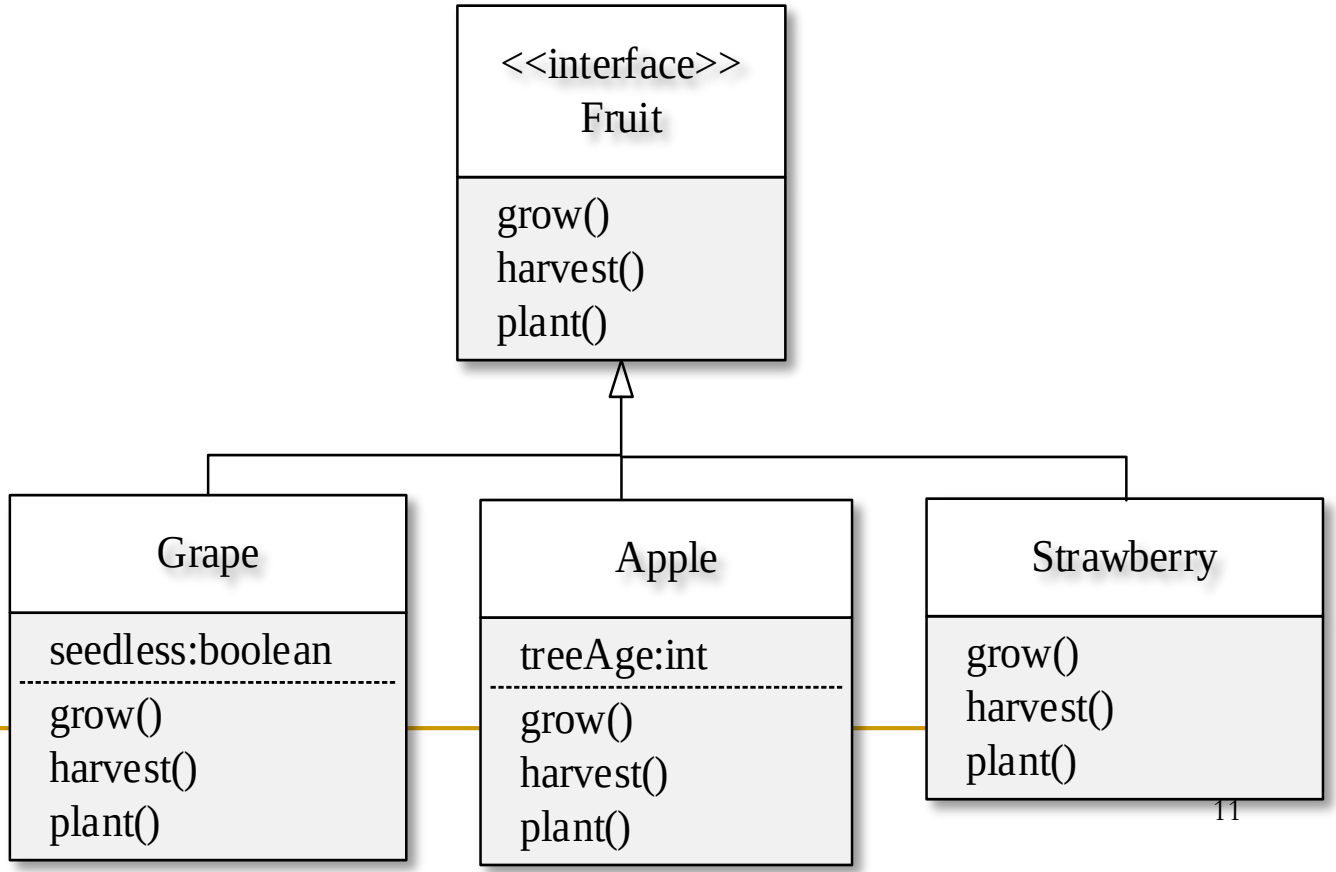
工厂模式

- 工厂模式专门负责将大量有共同接口的类实例化，工厂模式可以动态决定将哪一个类实例化。
- 工厂模式有以下几种形态：
  - **简单工厂(Simple Factory)模式：**又称静态工厂方法模式
  - **工厂方法(Factory Method)模式：**又称多态工厂模式
  - **抽象工厂(Abstract Factory)模式：**又称工具箱模式

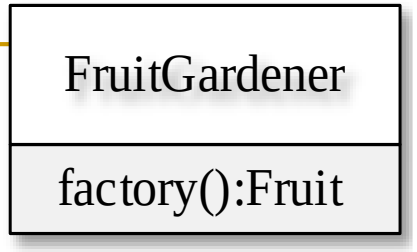


简单工厂模式的引进

- 有一个农场，专门出售各种水果，这个系统里有如下的水果：葡萄、草莓、苹果。水果与其他的植物有很大的不同，水果最终是可以采摘食用的。那么自然的方法是创建一个各种水果都适应的接口，以便和农场里的其他植物区分开。



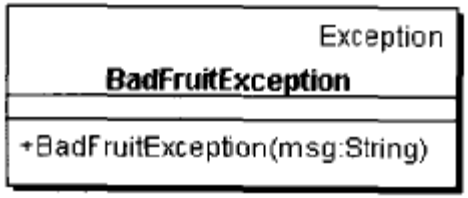
简单工厂模式的引进



- 园丁是农场的一部分，园丁类是FruitGardener，其有一个静态工厂方法。
- 园丁会根据客户的不同要求，创建出不同的水果。而接到不合法的要求，就会抛出异常。

```
public class FruitGardener
{
    /**
     *  静态工厂方法
     */
    public static Fruit factory(String which)
            throws BadFruitException
    {
        if (which.equalsIgnoreCase("apple"))
        {
            return new Apple();
        }
        else if (which.equalsIgnoreCase("strawberry"))
        {
            return new Strawberry();
        }
        else if (which.equalsIgnoreCase("grape"))
        {
            return new Grape();
        }
        else
        {
            throw new BadFruitException("Bad fruit request");
        }
    }
```

简单工厂模式的引进

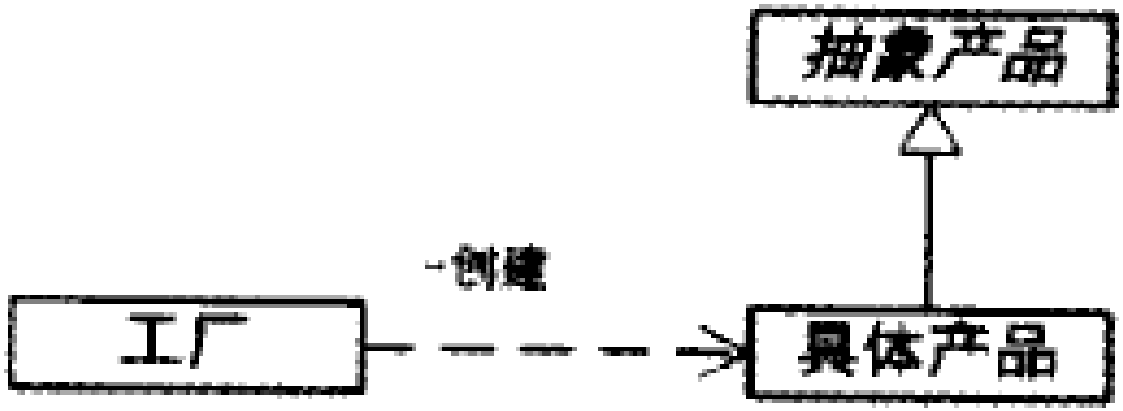


客户端代码：

```
public class BadFruitException extends Exception
{
    public BadFruitException(String msg)
    {
        super(msg);
    }
}
try
{
    FruitGardener.factory("grape");
    FruitGardener.factory("apple");
    FruitGardener.factory("strawberry");
        ...
}
catch(BadFruitException e)
{
    ...
}
```

简单工厂模式的结构

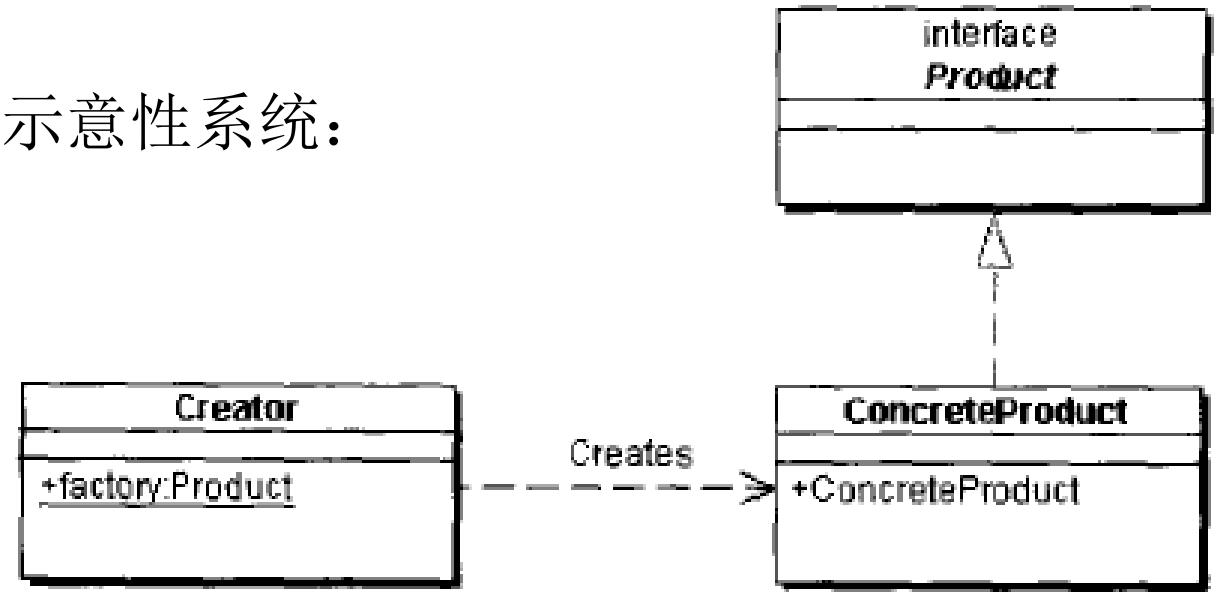
- 简单工厂模式是类的创建模式，一般性结构如下：



角色与结构

- 简单工厂模式就是由一个工厂类根据传入的参数来决定创建出那一种产品类的实例。

示意性系统：



角色与结构

- 简单工厂模式涉及到工厂角色、抽象产品角色和具体产品角色。
  - 工厂类(Creator) 角色：担任这个角色的是工厂模式的核心，含有与应用紧密相关的商业逻辑。工厂类在客户端的直接调用下创建产品对象，往往由一个具体的java类实现。
  - 抽象产品(Product)角色：担任这个角色的类是由工厂方法模式所创建的对象的基类，或他们拥有共同的接口。抽象产品可以由一个java接口或java抽象类实现。
  - 具体产品(Concrete Product)角色：工厂方法模式所创建的任何对象都是这个角色的实例，具体产品由一个java类来实现。

角色与结构—源代码

示意性系统源代码：

```
public class Creator
{
    /**
     *  静态工厂方法
     */
     public static Product factory()
     {
          return new ConcreteProduct();
     }
}
```

```
public interface Product
{
}
```

```
public class ConcreteProduct implements Product
{
     public ConcreteProduct(){}
}
```

简单工厂模式在Java中的应用

- java.text.DateFormat：格式化本地时间或日期

  public abstract class DateFormat extends Format

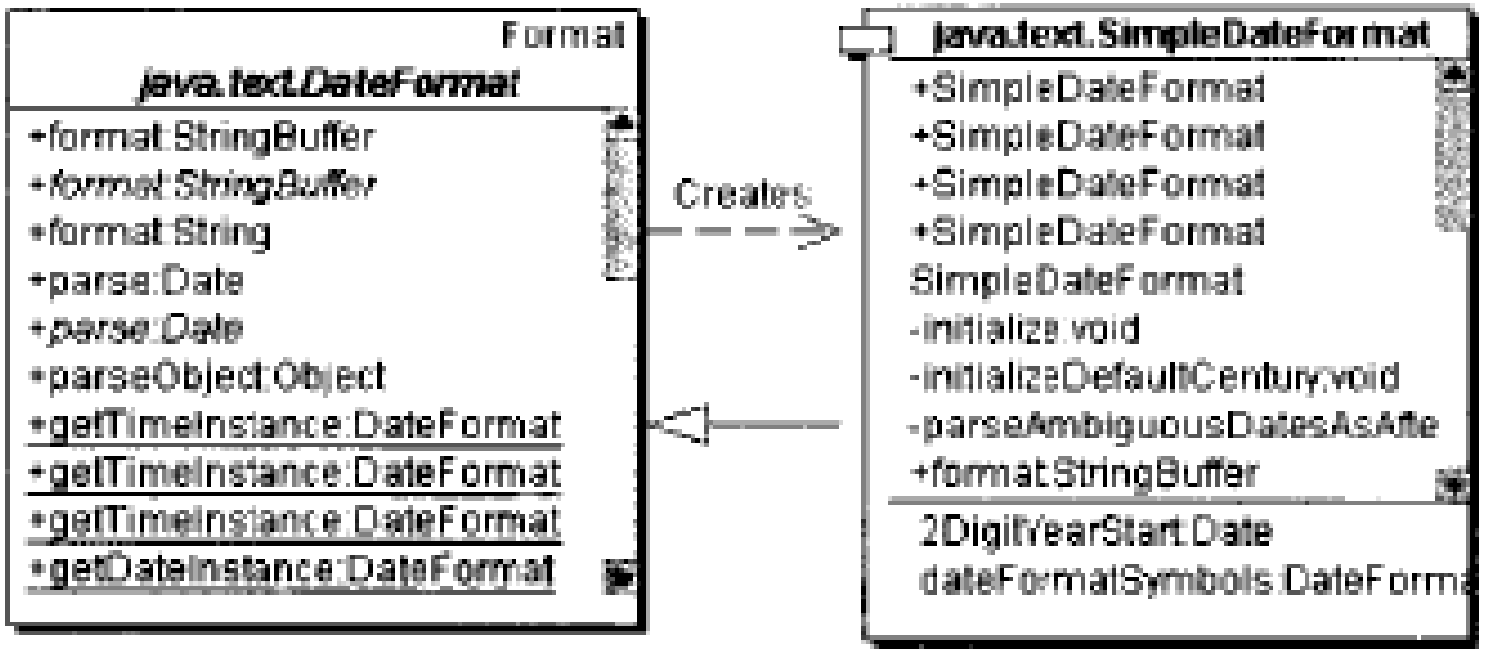
  三个重载的静态工厂方法：

```
public final static DateFormat getDateInstance();
public final static DateFormat getDateInstance(int style);
public final static DateFormat getDateInstance(int style, Locale locale);
```

简单工厂模式在Java中的应用

- getDateInstance()方法并没有调用DateFormat的构造子来提供自己的实例，他做了两件事：
  - 运用多态性：SimpleDateFormat是DateFormat的具体子类，该方法可以返回SimpleDateFormat的实例，并且仅将它声明为DateFormat类型。
  - 使用静态工厂方法：将具体子类实例化的工作隐藏起来，客户端不必考虑如何将具体子类实例化，抽象类DateFormat提供了它的合适的具体子类的实例。

类图


Format
java.text.DateFormat
+format:StringBuffer
+format:StringBuffer
+format:String
+parse:Date
+parse:Date
+parseObject:Object
+getTimeInstance:DateFormat
+getTimeInstance:DateFormat
+getTimeInstance:DateFormat
+getDateInstance:DateFormat
Creates
java.text.SimpleDateFormat
+SimpleDateFormat
+SimpleDateFormat
+SimpleDateFormat
+SimpleDateFormat
SimpleDateFormat
-initialize:void
-initializeDefaultCentury:void
-parseAmbiguousDatesAsAfte
+format:StringBuffer
2DigitYearStart:Date
dateFormatSymbols:DateForma

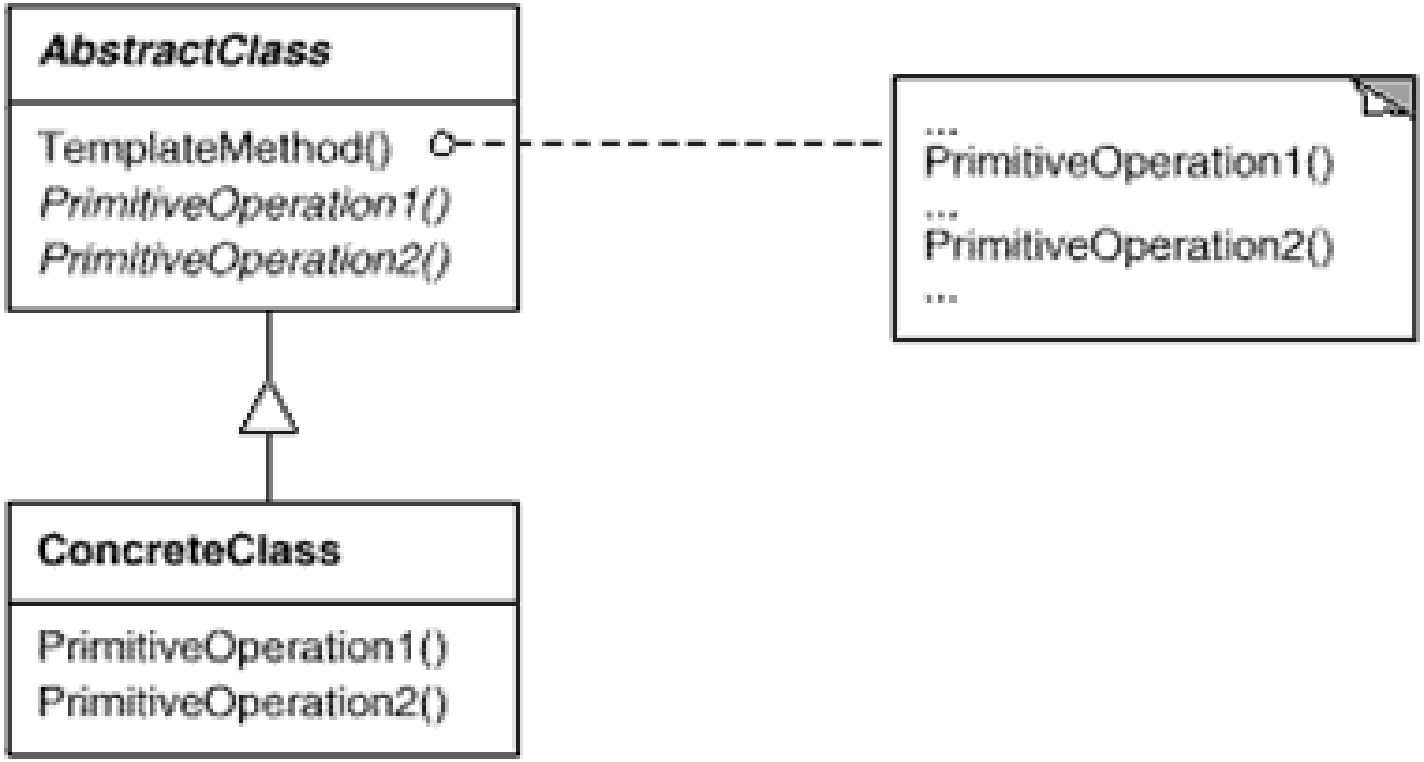
模版方法(Template Method)模式

- 定义一个操作中的算法的骨架，而将这些步骤延迟到子类中。Template Method模式使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。

——Gof

模版方法模式

- 抽象模版角色
  - 定义了一个或多个抽象操作（基本方法），让子类实现。
  - 定义了并实现了一个模版方法。
- 具体模版角色
  - 实现父类所定义的一个或多个抽象方法。



示意性源代码

抽象模版类：

```
abstract public class AbstractClass
{
    /**
     *    模版方法的声明和实现
     */
    public void TemplateMethod()
    {
        // 调用基本方法（由子类实现）
        doOperation1();
        // 调用基本方法（由子类实现）
        doOperation2();
        // 调用基本方法（已经实现）
        doOperation3();
    }

    /**
     *    基本方法的声明（由子类实现）
     */
    protected abstract void doOperation1();
    /**
     *    基本方法的声明（由子类实现）
     */
    protected abstract void doOperation2();
    /**
     *    基本方法（已经实现）
     */
    private final void doOperation3()
    {
        //do something
    }
}
```

示意性源代码

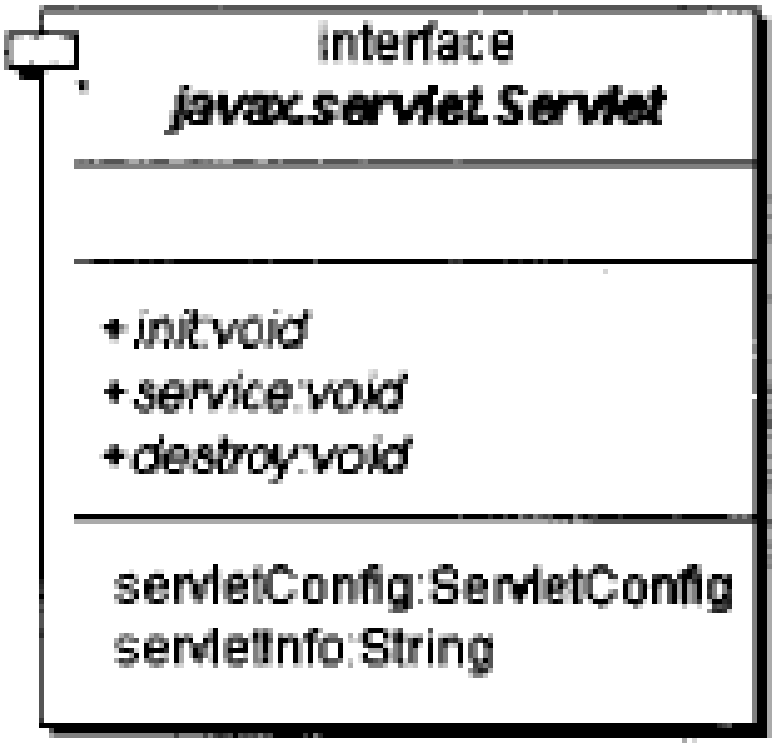
具体模版类：

```
public class ConcreteClass extends AbstractClass
{
    /**
     *  基本方法的实现
     */
    public void doOperation1()
    {
        System.out.println("doOperation1();");
    }
    /**
     *  基本方法的实现
     */
    public void doOperation2()
    {
        //像下面这样的调用不应当发生
        //doOperation3();
        System.out.println("doOperation2();");
    }
}
```

专题：Servlet中的模式

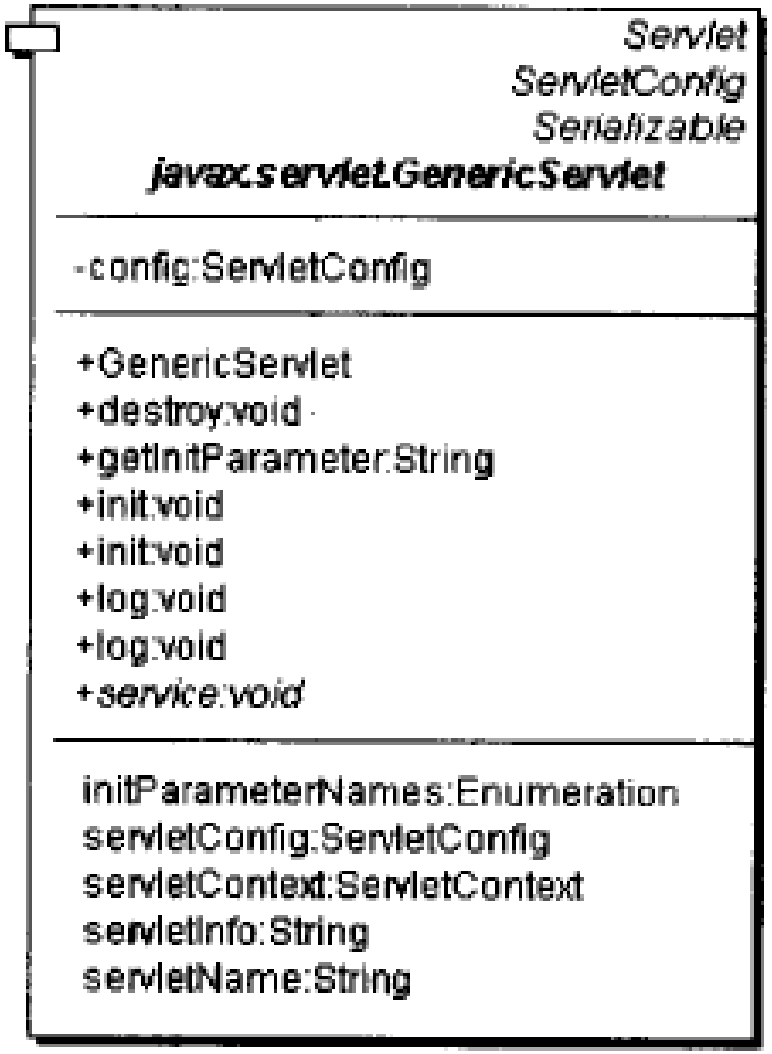
- Java Servlet API
  - javax.servlet：普通Servlet模型（TCP/IP）
  - javax.servlet.http：HTTP和HTTPS的Servlet模型
- 符合开闭原则

专题：Servlet中的模式



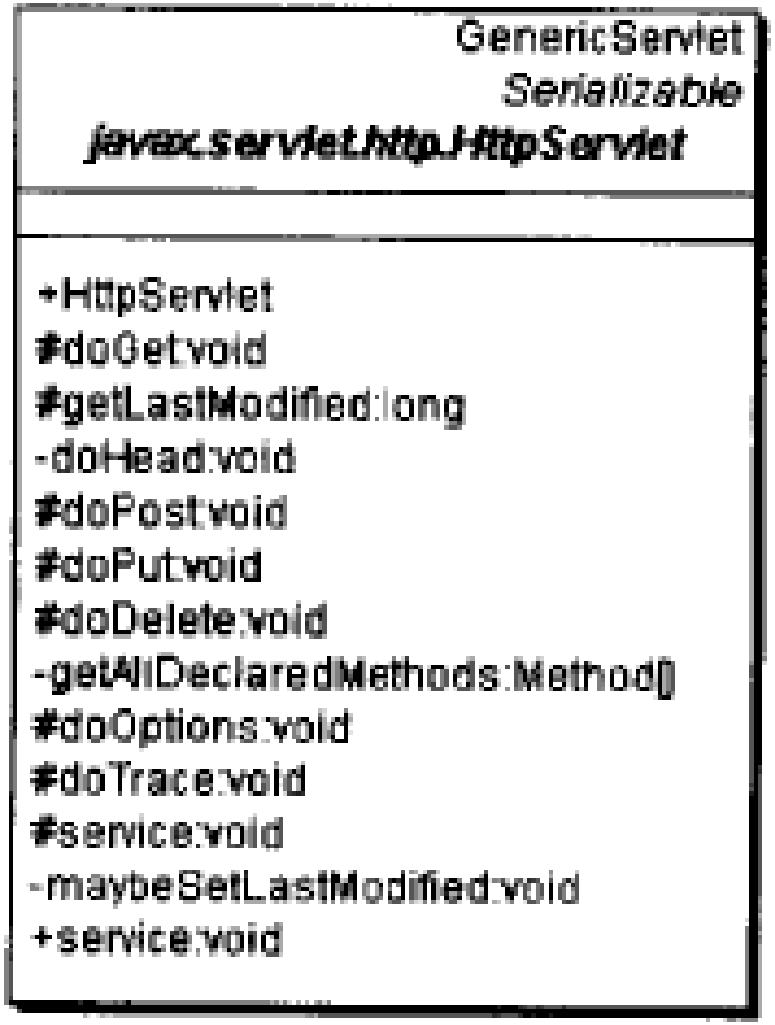
Servlet接口声明三个方法：init()、service()和destroy()方法决定了Servlet的生命周期。

专题：Servlet中的模式



- GenericServlet是一个抽象类，提供了Servlet接口的默认实现。
- 留下了一个抽象方法service()，任何具体的Servlet类均必须提供service方法。

专题：Servlet中的模式



- HttpServlet是一个抽象类，为service()方法和七个do方法都提供了默认实现。

模版方法模式在Servlet中的使用

- HttpServlet类提供了一个service()方法，这个方法调用七个do方法中的一个或几个，完成对客户端调用的响应。
- 这些do方法需要由HttpServlet的具体子类提供。
- **service()方法是一个模版方法**
- **七个do方法是基本方法**
- HttpServlet源码

HttpServlet源码

```
protected void service(HttpServletRequest req, HttpServletResponse resp)
                                        throws ServletException, IOException {

    String method = req.getMethod();

    if (method.equals(METHOD_GET)) {
       ……
       doGet(req, resp);
       ……

    } else if (method.equals(METHOD_HEAD)) {
        ……
        doHead(req, resp);

    } else if (method.equals(METHOD_POST)) {
        doPost(req, resp);

    } ……

}
```

例子

```
public class TestServlet extends HttpServlet
{
    /**
     *      当 HTTP 请求是 GET 时，此方法会被触发
     */
    public void doGet(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException
    {
            response.setContentType("text/html");
            PrintWriter out = response.getWriter();
            out.println("<h1>The http request is GET.</h1>");
            out.println("<h2>Now is " + new Date() + "</h2>");
            out.close();
    }
```

```
    /**
     *      当 HTTP 请求是 POST 时，此方法会被触发
     */
    public void doPost(HttpServletRequest request,
            HttpServletResponse response)
            throws ServletException, IOException
    {
            response.setContentType("text/html");
            PrintWriter out = response.getWriter();
            out.println("<h1>The http request is POST.</h1>");
            out.println("<h2>Now is " + new Date() + "</h2>");
            out.close();
    }
}
```