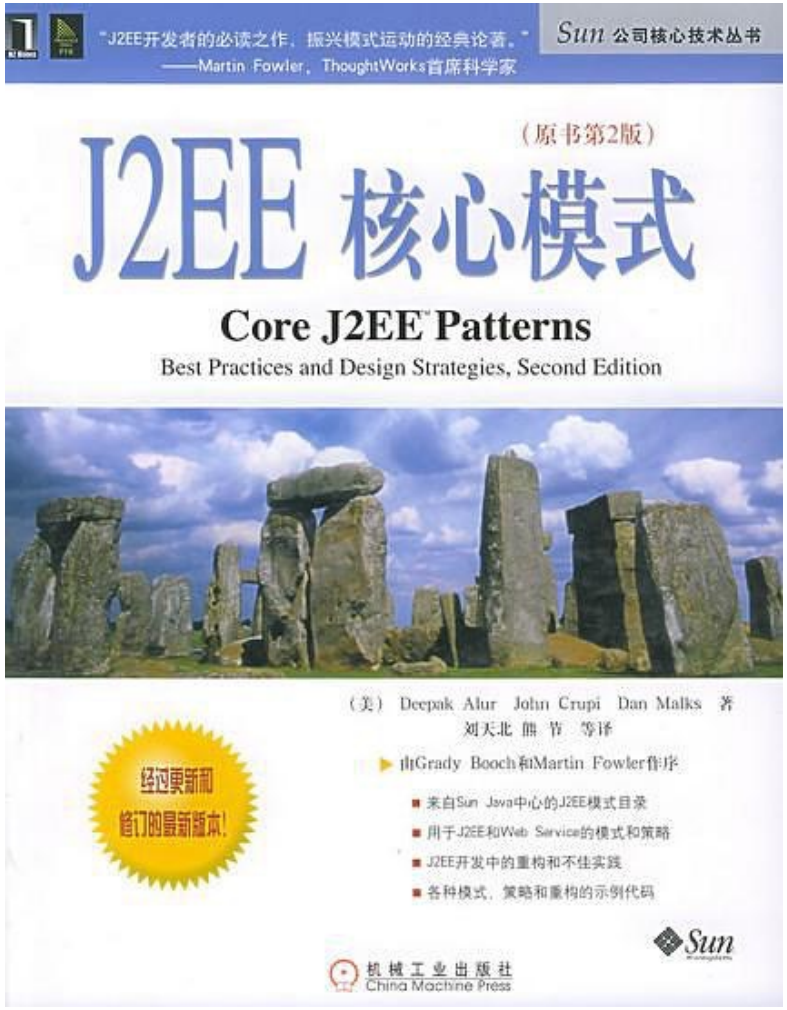
南京大学

J2EE核心模式简介

参考资料

- J2EE核心模式



什么是J2EE

- J2EE是一种用来开发分布式企业软件应用系统的平台。
- Java语言从创生之日起，就获得了广泛接纳，经历了巨大的发展。越来越多的技术都成为了Java平台的一部分，为了适应不同的需要也开发出了很多全新的API和标准，最终，Sun公司联合了多家业界巨头，在开放的Java社区组织名义下，把所有与企业开发相关的标准、API整合起来，构成了J2EE平台。

J2EE优势

对于企业，J2EE平台有很多优势

- J2EE为企业级运算的许多领域（比如数据库连接、企业业务组件、面向消息的中间件、Web相关组件、通信协议以及互操作性）设立了标准
- J2EE促进了人们基于开放的标准开发软件；如此构建的系统实现，出自名门，安全稳固，因此J2EE构成了一种可靠的技术投资
- J2EE是一种标准的开发平台，基于此开发的软件组件能够在不同厂商中相互移植，避免被一家厂商锁定

J2EE优势

- 在软件开发过程中采用J2EE能够缩短开发周期，使产品尽快投放市场，这是因为，系统的很多底层架构和基础部分都已经由产品厂商按照J2EE规范标准实现出来了。因此大多数IT企业可以不再开发中间件，集中精力构建符合自己商业需要的应用
- J2EE提高了程序员的生产力，因为对于Java程序员们，相对来说很容易就能学会基于Java语言的J2EE技术。所有企业软件开发能够在J2EE平台上、利用Java语言完成
- J2EE增进了现存各种异构系统之间的互操作性

J2EE中的模式

- 模式的诞生与定义
  - ✓ **Alexander给出了关于模式的经典定义：**
    - **每个模式都描述了一个在我们的环境中不断出现的问题，然后描述了该问题的解决方案的核心，通过这种方式，人们可以无数次地重用那些已有的解决方案，无须再重复相同的工作**

模式是在特定环境下人们解决某类重复出现问题的一套成功或有效的解决方案。

A pattern is a successful or efficient solution to a recurring problem within a context.

J2EE中的模式

- J2EE模式分类（21个）
  - 表现层（8个）
    - 拦截过滤器、前端控制器、Context对象、应用控制器、视图助手、复合视图、服务到工作者、分配器视图
  - 业务层（9个）
    - 业务代表、服务定位器、会话门面、应用服务、业务对象、复合实体、传输对象、传输对象组装器、值列表处理器
  - 集成层（4个）
    - 数据访问对象、服务激活器、业务领域存储、WebService中转

表示层-拦截过滤器模式

拦截过滤器模式(Intercepting Filter Pattern)

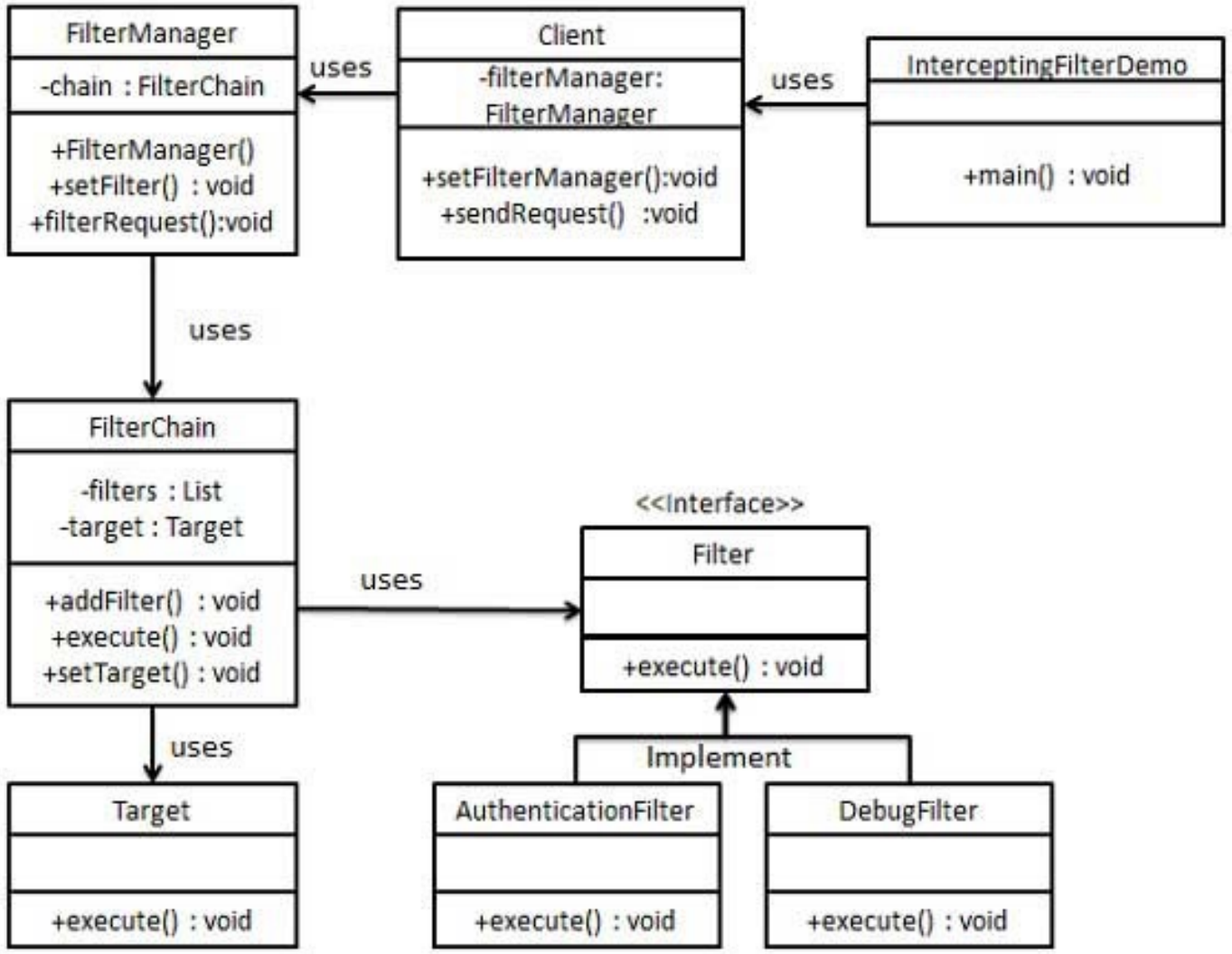
- 问题：
  - 要在请求被处理之前、之后拦截并操作一个请求和它的响应，即请求的预处理和后处理。预处理与后处理包括：
    - 客户端是否有一个有效的会话？
    - 请求的目录路径是否违反了限制条件？
    - 系统是否支持客户端的浏览器类型？
    - 客户端是用哪一种编码方式发送数据的？
    - 请求数据流是否加密？是否压缩？

拦截过滤器模式

- 解决方案：
  - 使用拦截过滤器，作为一个可插拔式的过滤器，实现请求、响应的预处理和后处理。
  - 一个过滤器管理器，负责把各个处于松耦合关系的过滤器结合成一个链，并把控制依次委派给合适的过滤器。这样一来，不必改动现有代码就能够以各种方式加入、删除、合并这些过滤器。

拦截过滤器模式

- 模式结构：



参与者

- Client
  - 客户端把请求发送到FilterManager
- FilterManager
  - 过滤器管理器负责管理过滤器的处理过程。它用合适的过滤器、按正确的顺序创建FilterChain，并初始化处理过程
- FilterChain
  - 过滤器链是由多个独立过滤器构成的一个有序集合
- Filter
  - 代表被映射到一个目标资源的独立过滤器
- Target
  - 目标是客户端请求的资源

拦截过滤器模式-实现

- 抽象过滤器

```
public interface Filter {
        //声明抽象业务方法
        public void execute(String request);

}
```

拦截过滤器模式-实现

- 实体过滤器

```
public class AuthenticationFilter implements Filter {
   //声明具体业务方法
   public void execute(String request){
      System.out.println("Authenticating request: " + request);
   }
}
```

```
public class DebugFilter implements Filter {
   //声明具体业务方法
   public void execute(String request){
      System.out.println("request log: " + request);
   }
}
```

拦截过滤器模式-实现

- 目标Target

```
public class Target {
   //目标执行的方法
   public void execute(String request){
      System.out.println("Executing request: " + request);
   }
}
```

拦截过滤器模式-实现

- 过滤器链

```
public class FilterChain {
   private List<Filter> filters = new ArrayList<Filter>(); //过滤器链
   private Target target; //目标
   public void addFilter(Filter filter){
      filters.add(filter);
   }
   public void execute(String request){
      for (Filter filter : filters) {
         filter.execute(request);
      }
      target.execute(request);
   }
   public void setTarget(Target target){
      this.target = target;
   }
}
```

拦截过滤器模式-实现

- 过滤器链

```
public class FilterManager {
   FilterChain filterChain; //过滤器链定义

   public FilterManager(Target target){
      filterChain = new FilterChain(); //初始化
      filterChain.setTarget(target);
   }
   public void setFilter(Filter filter){
      filterChain.addFilter(filter); //设置过滤器链
   }

   public void filterRequest(String request){
      filterChain.execute(request); //调用过滤器链
   }
}
```

拦截过滤器模式-实现

- 过滤器链

```
public class Client {
   FilterManager filterManager; //过滤器管理器定义

   public void setFilterManager(FilterManager filterManager){
      this.filterManager = filterManager;
   }

   public void sendRequest(String request){
      filterManager.filterRequest(request); //调用过滤器管理器
   }
}
```

拦截过滤器模式-实现

- 过滤器链

```
public class InterceptingFilterDemo {

  public static void main(String[] args) {
    FilterManager filterManager = new FilterManager(new Target());
    filterManager.setFilter(new AuthenticationFilter());
    filterManager.setFilter(new DebugFilter());

    Client client = new Client();
    client.setFilterManager(filterManager);
    client.sendRequest("HOME");
  }
}
```

程序输出效果：

```
Authenticating request: HOME
request log: HOME
Executing request: HOME
```

拦截过滤器模式扩展

- 标准过滤器
- 定制过滤器-与装饰器模式(GoF)结合
- 基本过滤器-过滤器的共通功能可以封装在基本过滤器里，在所有过滤器里得到共享
- 模板过滤器-融合了模板方法(GoF)模式，基本过滤器规定了每个过滤器都要完成的大致步骤，过滤器子类确定具体怎么样完成这些步骤
- WebService消息处理
- 定制Soap过滤器
- JX-RPC过滤器

拦截过滤器模式

- 效果：
  - 利用松耦合的处理器实现控制集中化(允许按多个方式组合)
  - 增进了重用（可插拔式的拦截器）
  - 能够灵活的通过声明配置（不要对核心代码重新编译）
  - 不便于信息共享

前端控制器（Front Controller Pattern）

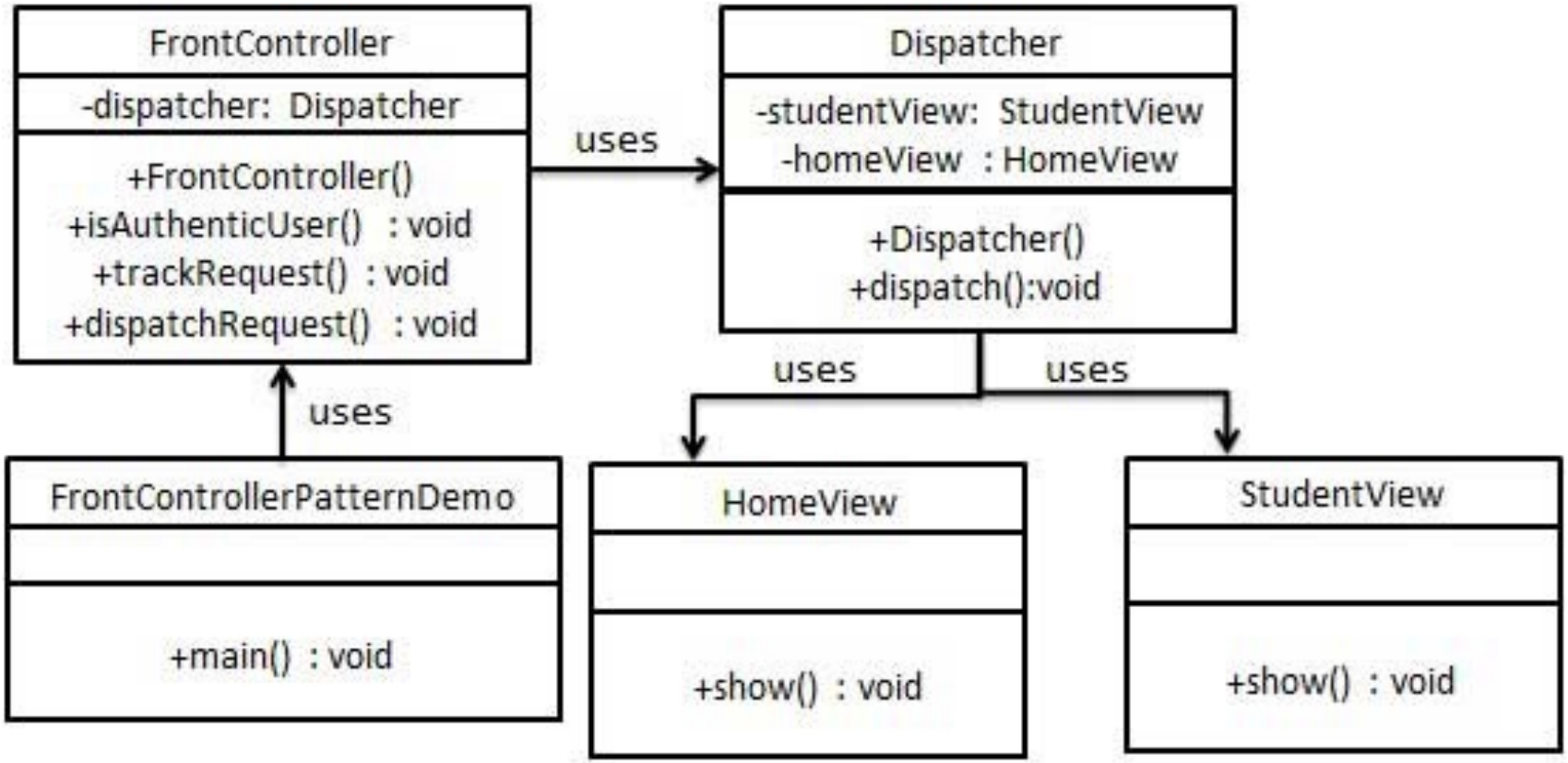
- 问题：
  - 想给表示层的请求处理安排一个集中的访问点。
  - 系统需要一个集中的访问点来处理请求，如果没有集中的访问点，多个请求之间共用控制代码就会在许多文件中重复出现。
  - 如果控制代码和视图创建代码混在一起，整个应用系统的模块化程度和内聚性都要下降。
  - 另外，如果控制代码多处散放，也不便于代码维护，只要代码一处有改动，就需要改动多个文件。

前端控制器

- 解决方案：
  - 使用一个前端控制器，作为最初的接触点，用来处理所有相关请求。前端控制器集中了控制逻辑，避免了逻辑的重复，完成了主要的请求处理操作。

前端控制器模式

- 模式结构：



参与者

- **前端控制器（Front Controller）**
  - 处理应用程序所有类型请求的单个处理程序，应用程序可以是基于 web 的应用程序，也可以是基于桌面的应用程序。
- **调度器（Dispatcher）**
  - 前端控制器可能使用一个调度器对象来调度请求到相应的具体处理程序。
- **视图（View）**
  - 视图是为请求而创建的对象。

前端控制器模式-实现

- 视图

```
public class HomeView {
   public void show(){
      System.out.println("Displaying Home Page");
   }
}
```

```
public class StudentView {
   public void show(){
      System.out.println("Displaying Student Page");
   }
}
```

前端控制器模式-实现

- 调度器

```
public class Dispatcher {
   private StudentView studentView;
   private HomeView homeView;
   public Dispatcher(){
      studentView = new StudentView();
      homeView = new HomeView();
   }

   public void dispatch(String request){
      if(request.equalsIgnoreCase("STUDENT")){
         studentView.show();
      }else{
         homeView.show();
      }
   }
}
```

前端控制器

- 前端控制器

```
public class FrontController {
   private Dispatcher dispatcher;
   public FrontController(){
      dispatcher = new Dispatcher();
   }
   private boolean isAuthenticUser(){
      System.out.println("User is authenticated successfully.");
      return true;
   }
   private void trackRequest(String request){
      System.out.println("Page requested: " + request);
   }
   public void dispatchRequest(String request){
      //记录每一个请求
      trackRequest(request);
      //对用户进行身份验证
      if(isAuthenticUser()){
         dispatcher.dispatch(request);
      }
   }
}
```

前端控制器模式-实现

- Demo代码

```
public class FrontControllerPatternDemo {
   public static void main(String[] args) {
      FrontController frontController = new FrontController();
      frontController.dispatchRequest("HOME");
      frontController.dispatchRequest("STUDENT");
   }
}
```

程序输出效果：

```
Page requested: HOME
User is authenticated successfully.
Displaying Home Page
Page requested: STUDENT
User is authenticated successfully.
Displaying Student Page
```

前端控制器模式扩展

- Servlet前端：将控制器实现为servlet
- JSP前端：不太合适，一般用Servlet前端
- 命令加控制器：命令模式(GoF)与前端控制器模式的结合
- 物理资源映射
- 逻辑资源映射
- 多路资源映射
- 控制器中的分配器：用于视图管理以及分配器功能重的场景
- 基类前端：实现一个控制器基类，其它的控制器扩展其实现

效果

- 集中控制
  - 集中了各个请求之间共同的处理控制逻辑。控制器负责把最初的请求委派给应用控制器，进行底层的业务处理以及视图生成操作
- 提高系统的可维护性
  - 便于监管控制流程
- 增进了重用
- 增加开发团队中的职责之间的划分

服务定位器（Service Locator Pattern）

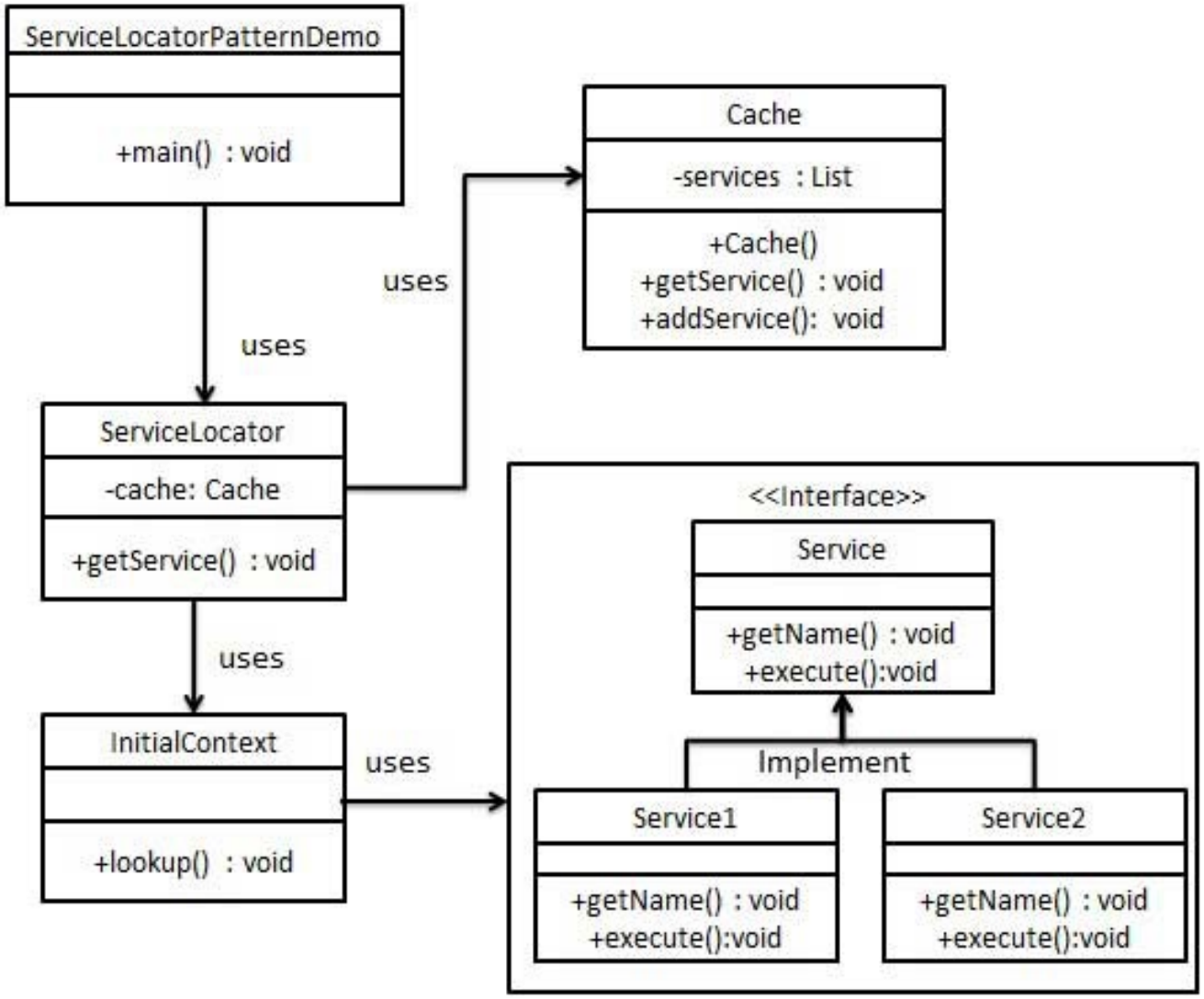
- 问题：
  - 需要以一种统一的、透明的方式定位业务组件和业务服务

服务定位器模式

- 解决方案
  - 使用服务定位器，实现、封装对服务、组件的寻址。服务定位器能够隐藏寻址机制的实现细节，封装这一机制对不同实现的依赖
  - 可以通过服务定位器提供唯一的控制点，并提供缓存机制，改善系统性能。
  - 通常整个应用系统中只需要一个服务定位器，不过两个的也不罕见，一个应用于表示层，另一个应用于业务层。
  - 降低了客户端对底层寻址的依赖。
  - 通常使用单例模式来实现。

服务定位器模式

- 模式结构



参与者

- **服务（Service）**
  - 实际处理请求的服务。对这种服务的引用可以在 JNDI 服务器中查找到。
- **Context / 初始的 Context**
  - JNDI Context 带有对要查找的服务的引用。
- **服务定位器（Service Locator）**
  - 服务定位器是通过 JNDI 查找和缓存服务来获取服务的单点接触。
- **缓存（Cache）**
  - 缓存存储服务的引用，以便复用它们。
- **客户端（Client）**
  - Client 是通过 ServiceLocator 调用服务的对象。

服务定位器模式-实现

- 服务接口

```
public interface Service {
   public String getName();
   public void execute();
}
```

服务定位器模式-实现

- 具体服务

```
public class Service1 implements Service {
   public void execute(){
      System.out.println("Executing Service1");
   }
public String getName() {
      return "Service1";
   }
}
```

```
public class Service2 implements Service {
   public void execute(){
      System.out.println("Executing Service2");
   }
public String getName() {
      return "Service2";
   }
}
```

服务定位器模式-实现

- InitialContext：带有对要查找的服务的引用

```
public class InitialContext {
   public Object lookup(String jndiName){
      if(jndiName.equalsIgnoreCase("SERVICE1")){
         System.out.println("Looking up and creating a new Service1 object");
         return new Service1();
      }else if (jndiName.equalsIgnoreCase("SERVICE2")){
         System.out.println("Looking up and creating a new Service2 object");
         return new Service2();
      }
      return null;
   }
}
```

服务定

- 缓存

```
public class Cache {
  private List<Service> services;
  public Cache(){
    services = new ArrayList<Service>();
  }
  public Service getService(String serviceName){
    for (Service service : services) {
      if(service.getName().equalsIgnoreCase(serviceName)){
        System.out.println("Returning cached  "+serviceName+" object");
        return service;
      }
    }
    return null;
  }
  public void addService(Service newService){
    boolean exists = false;
    for (Service service : services) {
      if(service.getName().equalsIgnoreCase(newService.getName())){
        exists = true;
      }
    }
    if(!exists){
      services.add(newService);
    }
  }
}
```

服务定位器模式-实现

- 服务定位器

```
public class ServiceLocator {
   private static Cache cache;
   static {
      cache = new Cache();
   }
   public static Service getService(String jndiName){
      Service service = cache.getService(jndiName);
      if(service != null){
         return service;
      }
      InitialContext context = new InitialContext();
      Service service1 = (Service)context.lookup(jndiName);
      cache.addService(service1);
      return service1;
   }
}
```

服务定位器模式-实现

- 服务定位器Demo

```
public class ServiceLocatorPatternDemo {
   public static void main(String[] args) {
      Service service = ServiceLocator.getService("Service1");
      service.execute();
      service = ServiceLocator.getService("Service2");
      service.execute();
      service = ServiceLocator.getService("Service1");
```

程序输出效果：

```
Looking up and creating a new Service1 object
Executing Service1
Looking up and creating a new Service2 object
Executing Service2
Returning cached  Service1 object
Executing Service1
Returning cached  Service2 object
Executing Service2
```

服务定位器模式扩展

- EJB服务定位器：EJB客户端可以使用服务定位器来完成EJB组件的寻址
- JDBC数据源服务定位器：实现对JDBC数据源的寻址功能
- JMS服务定位器：用于JMS客户端对多种不同的JMS组件寻址
- WebService定位器

效果

- 抽象了系统的复杂性
  - 服务定位器封装了服务寻址、创建过程的复杂性，对客户端影藏了这种复杂性
- 为客户端提供了统一的服务访问方式
  - 服务定位器提供了一种有用的、精确地接口，可供所有客户端使用
- 便于添加EJB业务组件
  - 添加新的组件不会影响客户端，JMS组件同理
- 提高了系统的网络性能
- 通过缓存提高了客户端性能

数据访问对象模式

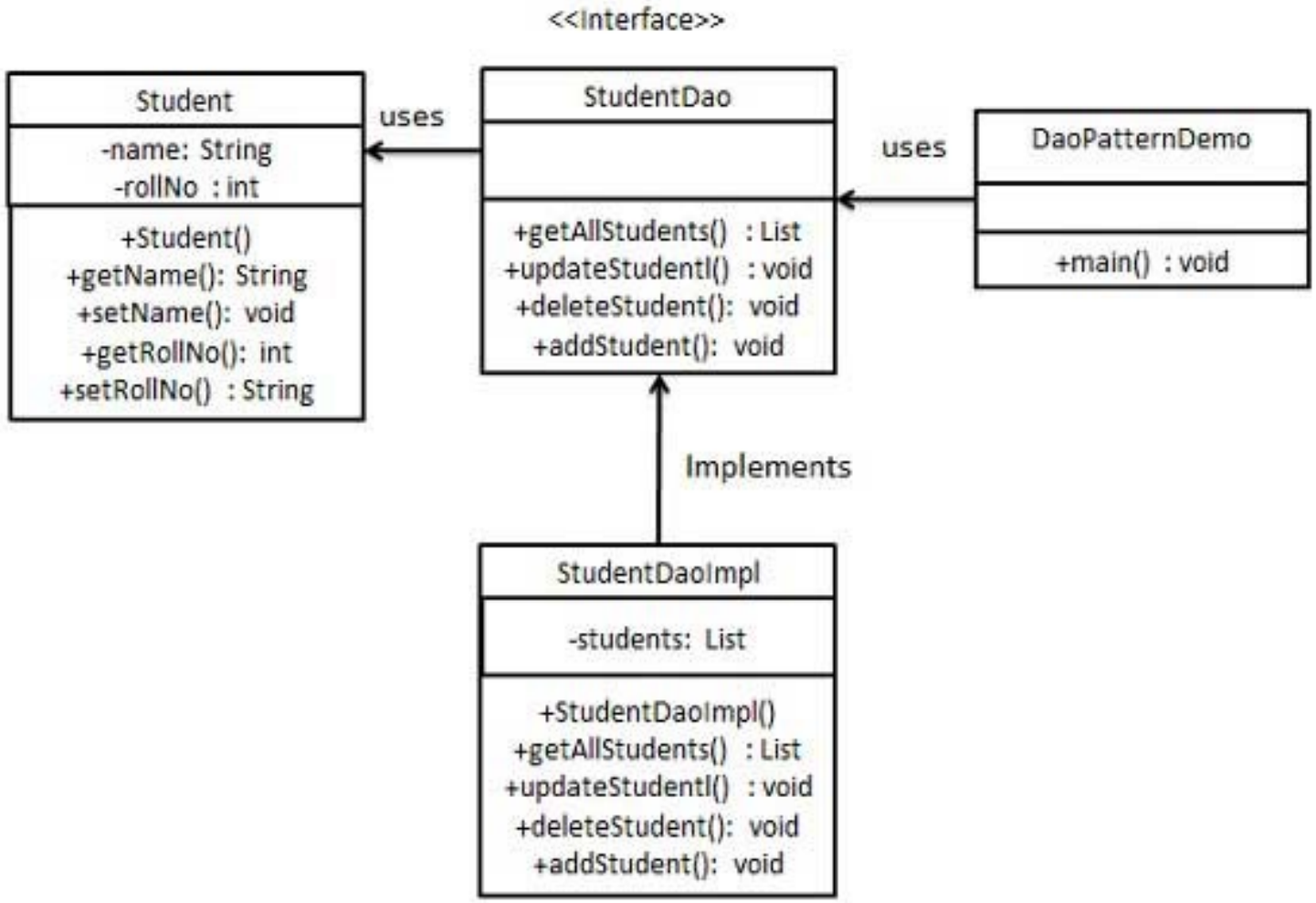
- 问题：
  - 需要将数据访问及操作的逻辑封装在一个单独的层次中

数据访问对象模式

- 解决方案
  - 使用数据访问对象提炼、封装对持久化存储介质的访问。数据访问对象负责管理与数据源的连接、并通过此连接获取、存储数据。
  - 数据访问对象(简称为DAO)实现了使用数据源所需的访问机制。不论使用哪种数据源，DAO总是向使用者提供了统一的API。

数据访问对象模式

- 模式结构



参与者

- **数据访问对象接口（Data Access Object Interface）**
  - 该接口定义了在一个模型对象上要执行的标准操作。
- **数据访问对象实体类（Data Access Object Concrete Class）**
  - 该类实现了上述的接口。该类负责从数据源获取数据，数据源可以是数据库，也可以是 xml，或者是其他的存储机制。
- **模型对象/数值对象（Model Object/Value Object）**
  - 该对象是简单的 POJO，包含了 get/set 方法来存储通过使用 DAO 类检索到的数据。

数据访问对象模式-实现

- 数值对象

```
public class Student {
   private String name;
   private int rollNo;
   Student(String name, int rollNo){
      this.name = name;
      this.rollNo = rollNo;
   }
   public String getName() {
      return name;
   }
   public void setName(String name) {
      this.name = name;
   }
   public int getRollNo() {
      return rollNo;
   }
   public void setRollNo(int rollNo) {
      this.rollNo = rollNo;
   }
}
```

数据访问对象模式-实现

- 数据访问对象接口

```
public interface StudentDao {
   public List<Student> getAllStudents();
   public Student getStudent(int rollNo);
   public void updateStudent(Student student);
   public void deleteStudent(Student student);
}
```

数据访问

- 实现了上述

```
public class StudentDaoImpl implements StudentDao {
   List<Student> students;
   public StudentDaoImpl(){
      students = new ArrayList<Student>();
      Student student1 = new Student("Robert",0);
      Student student2 = new Student("John",1);
      students.add(student1);      students.add(student2);
   }
   public void deleteStudent(Student student) {
      students.remove(student.getRollNo());
      System.out.println("Student: Roll No " + student.getRollNo()
         +", deleted from database");
   }
   public List<Student> getAllStudents() {
      return students;
   }
   public Student getStudent(int rollNo) {
      return students.get(rollNo);
   }
   public void updateStudent(Student student) {
      students.get(student.getRollNo()).setName(student.getName());
      System.out.println("Student: Roll No " + student.getRollNo()
         +", updated in the database");
   }
}
```

数据访问对象模式-实现

- 数据访问对象Demo

```
public class DaoPatternDemo {
   public static void main(String[] args) {
      StudentDao studentDao = new StudentDaoImpl();
      for (Student student : studentDao.getAllStudents()) {//输出所有的学生
         System.out.println("Student: [RollNo : "
            +student.getRollNo()+", Name : "+student.getName()+" ]");
      }
      Student student =studentDao.getAllStudents().get(0); //更新学生
      student.setName("Michael");
      studentDao.updateStudent(student);
```

程序输出效果：

```
Student: [RollNo : 0, Name : Robert ]
Student: [RollNo : 1, Name : John ]
Student: Roll No 0, updated in the database
Student: [RollNo : 0, Name : Michael ]
```

数据访问对象模式扩展

- 定制数据访问对象：自己编码数据访问层，可以使用数据访问对象模式，一般数据访问对象需要包含在数据库中创建、删除、更新、查找数据等的操作
- 数据访问对象工厂：借助抽象工厂和工厂方法，可以使数据访问对象的创建极具灵活性
- 传输对象集合：返回查询集合
- 缓存RowSet
- 只读RowSet

效果

- 用松耦合处理程序集中控制
- 对持久化数据的透明访问
- 为数据库数据结构提供面向对象的视图和封装
- 简化数据库移植
- 降低客户端代码的复杂度
- 将所有数据访问代码组织到一个独立的层次中
- 增加了一个层次：使用者与数据源之间
- 但需要设计整个类体系：如果使用工程策略，需要设计工厂与产品之间的类体系
- 面向对象设计增加了复杂度：使用RowSet对实现增加了复杂度