Homework：

1、　什么是透明装饰模式，什么是半透明装饰模式？请举例说明。

答：在设计模式中，装饰模式是一种结构型模式，它允许动态地添加或修改对象的行为。它通过创建一个包装对象，也称为装饰器，来包装原有对象。这种模式主要用于扩展功能，同时又不改变现有对象的结构。

对于透明装饰模式，它装饰后的对象与原始对象保持相同的接口。这使得客户端代码在使用装饰过的对象时，不需要知道对象被装饰的事实，从而确保装饰器的使用对客户端是完全透明的。

半透明装饰模式允许装饰器提供一些原始对象没有的方法或属性，这意味着装饰器对客户端不完全透明。客户端可以选择性地使用装饰器提供的额外方法或属性。缺点是无法多次进行装饰。

为了演示透明装饰模式和半透明装饰模式，我们可以设计一个网络请求处理程序，具体包含两个透明装饰器（日志记录和压缩）和两个半透明装饰器（JSON解析和身份验证）。

首先定义抽象构件类：

```
// Component (抽象构件)
10 个用法   6 个继承者
abstract class HttpRequestHandler {
    7 个用法   6 个实现
    public abstract String handleRequest(String request);
}
```

接着定义具体构建类：

```
// ConcreteComponent (具体构件类)
2 个用法
class BasicRequestHandler extends HttpRequestHandler {
    7 个用法
    @Override
    public String handleRequest(String request) {
        return "Handled Request Data: " + request;
    }
}
```

然后定义抽象装饰器类：

```
// Decorator (抽象装饰类)
4 个用法   4 个继承者
abstract class RequestHandlerDecorator extends HttpRequestHandler {
    2 个用法
    protected HttpRequestHandler wrappedHandler;
    4 个用法
    public RequestHandlerDecorator(HttpRequestHandler wrappedHandler) {
        this.wrappedHandler = wrappedHandler;
    }
    7 个用法   4 个重写
    @Override
    public String handleRequest(String request) {
        return wrappedHandler.handleRequest(request);
    }
}
```

分别定义透明装饰器和半透明装饰器：

```java
// 透明装饰类 - 添加日志记录
1 个用法
public class LoggingDecorator extends RequestHandlerDecorator {
    1 个用法
    public LoggingDecorator(HttpRequestHandler wrappedHandler) {
        super(wrappedHandler);
    }

    7 个用法
    @Override
    public String handleRequest(String request) {
        System.out.println("Logging Request: " + request);
        return super.handleRequest(request);
    }
}
```

```java
// 透明装饰类 - 数据压缩
2 个用法
public class CompressionDecorator extends RequestHandlerDecorator {
    1 个用法
    public CompressionDecorator(HttpRequestHandler wrappedHandler) {
        super(wrappedHandler);
    }

    7 个用法
    @Override
    public String handleRequest(String request) {
        String compressed = Base64.getEncoder().encodeToString(request.getBytes());
        return super.handleRequest(compressed);
    }
}
```

```java
// 半透明装饰类 - 身份验证
2 个用法
public class AuthenticationDecorator extends RequestHandlerDecorator {
    1 个用法
    public AuthenticationDecorator(HttpRequestHandler wrappedHandler) {
        super(wrappedHandler);
    }
    7 个用法
    @Override
    public String handleRequest(String request) {
        if (!authenticate(request)) {
            return "Authentication Failed: Access Denied";
        }
        return super.handleRequest(request);
    }
    1 个用法
    private boolean authenticate(String request) {
        return request.contains("auth_token");
    }
}
```

```java
import java.util.HashMap;
import java.util.Map;

// 半透明装饰类 - 解析JSON数据
2 个用法
public class JsonParsingDecorator extends RequestHandlerDecorator {
    3 个用法
    private Map<String, String> parsedData = new HashMap<>();

    1 个用法
    public JsonParsingDecorator(HttpRequestHandler wrappedHandler) {
        super(wrappedHandler);
    }

    7 个用法
    @Override
    public String handleRequest(String request) {
        parseJson(request);
        return super.handleRequest("JSON Processed: " + parsedData.toString());
    }

    1 个用法
    private void parseJson(String json) {
        json = json.replace( target: "{", replacement: "").replace( target: "}", replacement: "");
        String[] entries = json.split( regex: ",");
        for (String entry : entries) {
            String[] keyValue = entry.split( regex: ":");
            parsedData.put(keyValue[0].trim(), keyValue[1].trim());
        }
    }

    1 个用法
    public String getJsonField(String field) {
        return parsedData.getOrDefault(field, defaultValue: "Field not found");
    }
}
```

最后在 Main 类中使用构件和装饰器的组合：

```java
public class Main {
    0 个用法
    public static void main(String[] args) {
        HttpRequestHandler handler = new BasicRequestHandler();
        handler = new LoggingDecorator(handler);
        JsonParsingDecorator jsonHandler = new JsonParsingDecorator(handler);
        AuthenticationDecorator authHandler = new AuthenticationDecorator(jsonHandler);
        String response = authHandler.handleRequest("{\"auth_token\":\"12345\", \"message\":\"hello\"}");
        System.out.println("Final Response: " + response);
        System.out.println("JSON Field (message): " + jsonHandler.getJsonField("\"message\""));

        HttpRequestHandler handler1 = new BasicRequestHandler();
        CompressionDecorator compressHandler= new CompressionDecorator(handler1);
        String compress = compressHandler.handleRequest("{\"auth_token\":\"abc\", \"message\":\"123\"}");
        System.out.println("Compressed Response: " + compress);
    }
}
```

查看运行结果：

```
运行:  Main
F:\Java\bin\java.exe "-javaagent:F:\IntelliJ IDEA Community Edition 2022.3.2\lib\idea_rt.jar=10725
Logging Request: JSON Processed: {"auth_token"="12345", "message"="hello"}
Final Response: Handled Request Data: JSON Processed: {"auth_token"="12345", "message"="hello"}
JSON Field (message): "hello"
Compressed Response: Handled Request Data: eyJhdXRoX3Rva2VuIjoiYWJjIiwgIm1lc3NhZ2UiOiIxMjMifQ==

进程已结束,退出代码0
```

LoggingDecorator 是透明装饰的一个例子。它增加了日志记录功能，但对于

使用这个装饰的客户端来说，它不影响对原始 handleRequest 方法的调用。客户端无需知道请求是否被记录，只关心请求的处理。JsonParsingDecorator 是半透明装饰的例子。它不仅覆盖了 handleRequest 方法以添加 JSON 解析，还引入了一个新的方法 getJsonField,该方法不是 HttpRequestHandler 接口的一部分。

这样的设计使得装饰者可以根据需要灵活地增加或修改功能，同时对客户端的影响最小化。

2、 附录

1) HttpRequestHandler

```
abstract class HttpRequestHandler {
    public abstract String handleRequest(String request);
}
```

2) BasicRequestHandler

```
class BasicRequestHandler extends HttpRequestHandler {
    @Override
    public String handleRequest(String request) {
        return "Handled Request Data: " + request;
    }
}
```

3) RequestHandlerDecorator

```
abstract class RequestHandlerDecorator extends HttpRequestHandler {
    protected HttpRequestHandler wrappedHandler;
    public RequestHandlerDecorator(HttpRequestHandler wrappedHandler)
{
        this.wrappedHandler = wrappedHandler;
    }
    @Override
    public String handleRequest(String request) {
        return wrappedHandler.handleRequest(request);
    }
}
```

4) CompressionDecorator

```
import java.util.Base64;
public class CompressionDecorator extends RequestHandlerDecorator {
    public CompressionDecorator(HttpRequestHandler wrappedHandler) {
        super(wrappedHandler);
    }

    @Override
    public String handleRequest(String request) {
        String compressed =
```

```
Base64.getEncoder().encodeToString(request.getBytes());
        return super.handleRequest(compressed);
    }
}
```

5) LoggingDecorator

```
public class LoggingDecorator extends RequestHandlerDecorator {
    public LoggingDecorator(HttpRequestHandler wrappedHandler) {
        super(wrappedHandler);
    }

    @Override
    public String handleRequest(String request) {
        System.out.println("Logging Request: " + request);
        return super.handleRequest(request);
    }
}
```

6) AuthenticationDecorator

```
public class AuthenticationDecorator extends RequestHandlerDecorator {
    public AuthenticationDecorator(HttpRequestHandler wrappedHandler)
{
        super(wrappedHandler);
    }
    @Override
    public String handleRequest(String request) {
        if (!authenticate(request)) {
            return "Authentication Failed: Access Denied";
        }
        return super.handleRequest(request);
    }
    private boolean authenticate(String request) {
        return request.contains("auth_token");
    }
}
```

7) JsonParsingDecorator

```
import java.util.HashMap;
import java.util.Map;

public class JsonParsingDecorator extends RequestHandlerDecorator {
    private Map<String, String> parsedData = new HashMap<>();

    public JsonParsingDecorator(HttpRequestHandler wrappedHandler) {
```

```
        super(wrappedHandler);
    }

    @Override
    public String handleRequest(String request) {
        parseJson(request);
        return super.handleRequest("JSON Processed: " +
parsedData.toString());
    }

    private void parseJson(String json) {
        json = json.replace("{", "").replace("}", "");
        String[] entries = json.split(",");
        for (String entry : entries) {
            String[] keyValue = entry.split(":");
            parsedData.put(keyValue[0].trim(), keyValue[1].trim());
        }
    }

    public String getJsonField(String field) {
        return parsedData.getOrDefault(field, "Field not found");
    }
}
```

8) Main

```
public class Main {
    public static void main(String[] args) {
        HttpRequestHandler handler = new BasicRequestHandler();
        handler = new LoggingDecorator(handler);
        JsonParsingDecorator jsonHandler = new
JsonParsingDecorator(handler);
        AuthenticationDecorator authHandler = new
AuthenticationDecorator(jsonHandler);
        String response =
authHandler.handleRequest("{\"auth_token\":\"12345\",
\"message\":\"hello\"}");
        System.out.println("Final Response: " + response);
        System.out.println("JSON Field (message): " +
jsonHandler.getJsonField("\"message\""));

        HttpRequestHandler handler1 = new BasicRequestHandler();
        CompressionDecorator compressHandler= new
CompressionDecorator(handler1);
        String compress =
```

```
compressHandler.handleRequest("{\"auth_token\":\"abc\",
\"message\":\"123\"}");
        System.out.println("Compressed Response: " + compress);
    }
}
```