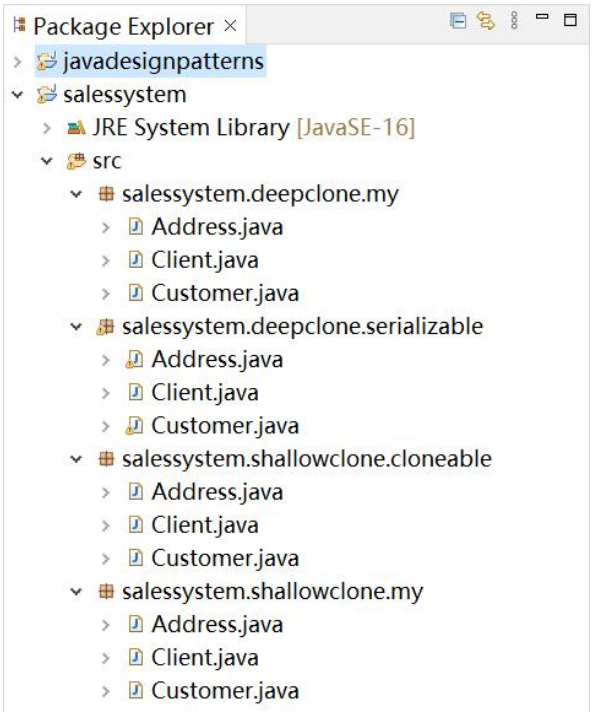
- shallowclone.cloneable 和 deepclone.serializable 为使用 Java 中的 Cloneable 接口以及 Serializable 接口的实现，.my 为不采用 java 中的接口的实现：



- 浅拷贝两种实现使用相同的客户端测试代码，为：

```
public class Client {
    public static void main(String args[]) {
        Customer cus_previous, cus_new;
        cus_previous = new Customer();
        Address address = new Address();
        cus_previous.setAddress(address);
        cus_new = cus_previous.clone();
        System.out.println("Customer是否相同？ " + (cus_previous == cus_new));
        System.out.println("Address是否相同？ " + (cus_previous.getAddress() ==
    }
}
```

- 深拷贝两种实现使用相同的客户端测试代码，为：

```
public class Client {
    public static void main(String args[]) {
        Customer cus_previous, cus_new = null;
        cus_previous = new Customer();
        Address address = new Address();
        cus_previous.setAddress(address);
        try {
            cus_new = cus_previous.deepClone();
        }
        catch(Exception e) {
            System.err.println("克隆失败！");
        }
        System.out.println("Customer是否相同？ " + (cus_previous == cus_new));
        System.out.println("Address是否相同？ " + (cus_previous.getAddress() ==
    }
}
```

1. 使用 Java 中的 Cloneable 接口以及 Serializable 接口实现浅拷贝和深拷贝。
浅拷贝：仿照 java design patterns 中提供的示例为 Customer 提供 clone()

```
public Customer clone() {
    Object obj = null;
    try {
        obj = super.clone();
        return (Customer)obj;
    }
    catch(CloneNotSupportedException e) {
        System.out.println("不支持复制！");
        return null;
    }
}
```

深拷贝：仿照 java design patterns 中示例为 Customer 提供 deepClone()

```
public Customer deepClone() throws IOException, ClassNotFoundException, Opt
    ByteArrayOutputStream bao=new ByteArrayOutputStream();
    ObjectOutputStream oos=new ObjectOutputStream(bao);
    oos.writeObject(this);
    ByteArrayInputStream bis=new ByteArrayInputStream(bao.toByteArray());
    ObjectInputStream ois=new ObjectInputStream(bis);
    return (Customer)ois.readObject();
}
```

2. 不采用 Java 中的接口实现深拷贝以及浅拷贝的原型模式。
浅拷贝：new 一个 Customer 对象并对成员逐个调用 setter 函数，将每个成员变量都 set 为 this 中对应项（此处 setxx 均用 this.xx = xx 实现）

```
public Customer clone() {
    Customer clone = new Customer();
    clone.setName(this.name);
    clone.setAge(this.age);
    clone.setGender(this.gender);
    clone.setAddress(this.address);
    return clone;
}
```

深拷贝：其他同浅拷贝，setAddress 的参数改为 this.address.deepClone()，另外为 Address 添加 deepClone 函数，即需要手动对 Address 也进行深拷贝。

```
public Customer deepClone() {
    Customer clone = new Customer();
    clone.setName(this.name);
    clone.setAge(this.age);
    clone.setGender(this.gender);
    clone.setAddress(this.address.deepClone());
    return clone;
}

public Address deepClone() {
    Address clone = new Address();
    clone.setStreet1(this.street1);
    clone.setCity(this.city);
    clone.setProvince(this.province);
    clone.setCountry(this.country);
    clone.setPostCode(this.postcode);
    return clone;
}
```

Address 中的成员均按与浅拷贝相同的方式进行 set 即可。

- 运行结果：对两种实现的浅拷贝均有：

```
Customer是否相同？ false
Address是否相同？ true
```

对两种实现的深拷贝均有：

```
Customer是否相同？ false
Address是否相同？ false
```

- 思考：对于浅克隆（Shallow Clone），当原型对象被复制时，只复制它本身和其中包含的值类型的成员变量，而引用类型的成员变量并没有复制。即原型对象的成员变量不分值和地址类型，一律直接按照值的方式进行了复制。而对于深克隆（Deep Clone），除了对象本身被复制外，对象所包含的所有成员变量也将被复制。其中值直接复制，引用则递归地对引用对象进行复制。

  那么，int 是值自然不必说，String 和 Boolean 是值还是引用？这些成员在进行深拷贝时还可以使用 xx = this.xx 和 this.xx = xx 的直接赋值方式吗？或者是像 Address 类一样需要另外提供 deepClone()函数？观察使用 Java 接口实现的拷贝中 Name（String）、Age（int）、Gender（Boolean）以及 Address 成员 City，发现在深浅拷贝中它们均相同，只有 Address 在深拷贝中不同。

```
Customer是否相同？ false
Name是否相同？ true
Age是否相同？ true
Gender是否相同？ true
Address是否相同？ true
City是否相同？ true
```

```
Customer是否相同？ false
Name是否相同？ true
Age是否相同？ true
Gender是否相同？ true
Address是否相同？ false
City是否相同？ true
```

对不使用 Java 接口实现的拷贝进行输出，结果仍然相同。于是得出结论，在自己实现的深拷贝中继续对 String 和 Boolean 使用直接赋值是可以的。