设计模式分析报告

2014 软工 S3-1 组

学号	姓名
3100000210	许是程
3110102942	谢晨威
3110000281	魏铭
3110300112	金龙湖
3100000212	张麟

摘要

本文主要介绍了工厂方法模式(Factory Method Pattern)和原型模式(Prototype Pattern)近年来的研究进展；作为证券账户系统开发团队，我们经过讨论认为工厂方法和原型这两种设计模式比较适合证券账户系统；分析了本小组所负责的证券账户子系统的系统架构；然后重点分析了我们预备采用的工厂方法模型的概念、特点，以及与本系统的契合处；最后探查了原型模式的相关属性以及其支持我们子系统的地方。

1.文献回顾

查阅了包涵 IEEE、ACMSIGSOFT 和 PATTERNS2011 等学术会议和期刊的 7 篇论文，并结合经典著作《设计模式》和网上资料，总结出来关于这两种设计模式近期的一些研究进展。

Gamma 等人的经典著作《设计模式》[1]所提出的 23 种设计模式为模块化的实践提供了很大程度上的改善，其中主要是对模式的实现进行了模块化。这直接反应于各个实现是在代码层面上分开的，从而使得项目工程当中各个交互的类的代码级别的依存关系能够去除。这些模式（例如原型模式）控制了访问特定对象的权限。所有这些模式都对客户端提供了批量方法，而且拥有一个基于需求创建的策略。其他模式（例如工厂方法模式）在结构上是相似的，继承被用于区分不同但是有关联的实现。因为这已经在面向对象中实现，所以这些方法并没有提供更多可重用的实现。

工厂方法模式的意图是定义一个用于创建对象的接口，让子类决定实例化哪一个类。工厂方法使一个类的实例化延迟到其子类。框架使用抽象类定义和维护对象之间的关系。这些对象的创建通常也由框架负责。首先，良好的封装性，代码结构清晰。一个对象创建是有条件约束的，如一个调用者需要一个具体的产品对象，只要知道这个产品的类名（或约束字符串）就可以了，不用知道创建对象的艰辛过程，减少模块间的耦合。其次，工厂方法模式的扩展性非常优秀。在增

加产品类的情况下，只要适当地修改具体的工厂类或扩展一个工厂类，就可以完成“拥抱变化”。再次，屏蔽产品类。这一特点非常重要，产品类的实现如何变化，调用者都不需要关心，它只需要关心产品的接口，只要接口保持不表，系统中的上层模块就不要发生变化，因为产品类的实例化工作是由工厂类负责，一个产品对象具体由哪一个产品生成是由工厂类决定的。在数据库开发中，大家应该能够深刻体会到工厂方法模式的好处：如果使用 JDBC 连接数据库，数据库从 MySql 切换到 Oracle，需要改动地方就是切换一下驱动名称（前提条件是 SQL 语句是标准语句），其他的都不需要修改，这是工厂方法模式灵活性的一个直接案例。最后，工厂方法模式是典型的解耦框架。高层模块值需要知道产品的抽象类，其他的实现类都不用关心，符合迪米特原则，我不需要的就不要去交流；也符合依赖倒转原则，只依赖产品类的抽象；当然也符合里氏替换原则，使用产品子类替换产品父类也没问题。

原型模式的简单程度是仅次于单例模式和迭代器模式，正是由于简单，使用的场景才非常的多，其定义如下：Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype. 用原型实例指定创建对象的种类，并且通过拷贝这些原型创建新的对象。原型模式的核心是一个克隆方法，通过该方法进行对象的拷贝，Java 提供了一个 Cloneable 接口来标示这个对象是可拷贝的，方法是覆盖 clone()方法。原型模式的优点：性能优良——原型模式是在内存二进制流的拷贝，要比直接新建一个对象性能好很多，特别是要在一个循环体内产生大量的对象时，原型模式可以更好的体现其优点；逃避构造函数的约束——这既是它的优点也是缺点，直接在内存中拷贝，构造函数是不会执行的（见“原型模式的注意事项”），优点就是减少了约束，缺点也是减少了约束，双刃剑，需要大家在实际应用时考虑。原型模式的使用场景：资源优化场景，类初始化需要消化非常多的资源，这个资源包括数据、硬件资源等；性能和安全要求的场景，通过新建产生一个对象需要非常繁琐的数据准备或访问权限，则可以使用原型模式；一个对象多个修改者的场景，一个对象需要提供给其他对象访问，而且各个调用者可能都需要修改其值时，可以考虑使用原型模式拷贝多个对象供调用者使用。在实际项目中，原型模式很少单独出现，一般是和工厂方法模式一起出现，通过克隆的方法创建一个对象，然后由工厂方法提供给

调用者。

Hannemann 等人[2]在此基础上探讨了设计模式的横切（crosscutting）。程序经常表现出这样的行为：不能自然地适应于一个程序模块、甚至几个密切关联的程序模块。方面社区把这种行为描述为横切，因为它正好切到指定编程模型的职责分界上。例如，在面向对象编程中，模块化的自然单位是类，而横切关注点就是跨多个类的关注点。这样，如果一个设计模式的行为要分解到三个不同的类，那么就说这个设计模式横切了这些类。横切会造成代码散布（相关代码不能和其他相关代码本地化到一起）和代码混乱（相关代码和非相关代码挨在一起）。这种散布和混乱造成很难对系统进行推断。但好处是能够对某些模式实现代码级重用，除了节省的实现工作之外，代码级重用还允许模式代码和文档进行更紧密的耦合。

下面是工厂方法模式的两个应用实例分析。

Ellis 等人[3]对比了工厂模式（以及相关的工厂方法模式和抽象工厂模式）和构造函数在 API 的设计中的表现。通过 Notepad Email Task 等实验，发现使用工厂模式在很多时候对设计是有很大程度的不利的，特别是时间有很大程度增长。因此在这种情况下，使用构造函数或子类化是更好的设计选择。之所以产生这样的结果是因为工厂模式中，工厂本身就是一个抽象类。因此警惕我们，设计模式虽然提供了很大的便利，但也使得一些原本更容易的事情变得复杂。

Parsana 等人[4]通过在数据库连接中使用工厂模式（以及相关的工厂方法模式和抽象工厂模式），我们可以得到弱耦合而非紧耦合，从而在应用程序中隐藏具体类。工厂模式使得应用程序与类的家族实现分离，并且提供一个简单的方法来使得少量改动代码就可以实现类的扩展。当直接创建对象时，在类中很难用功能扩展后的对象来替换它们。而如果使用工厂模式创建一个系列的对象时，就能轻易替换。使用工厂模式保持了项目的完整性，并且大大减少了开发消耗。

1.6 参考文献

[1] Gamma, Erich, et al. Design patterns: elements of reusable object-oriented software. Pearson Education, 1994.

[2] Hannemann, Jan, and Gregor Kiczales. "Design pattern implementation in Java

and AspectJ." *ACM Sigplan Notices*. Vol. 37. No. 11. ACM, 2002.

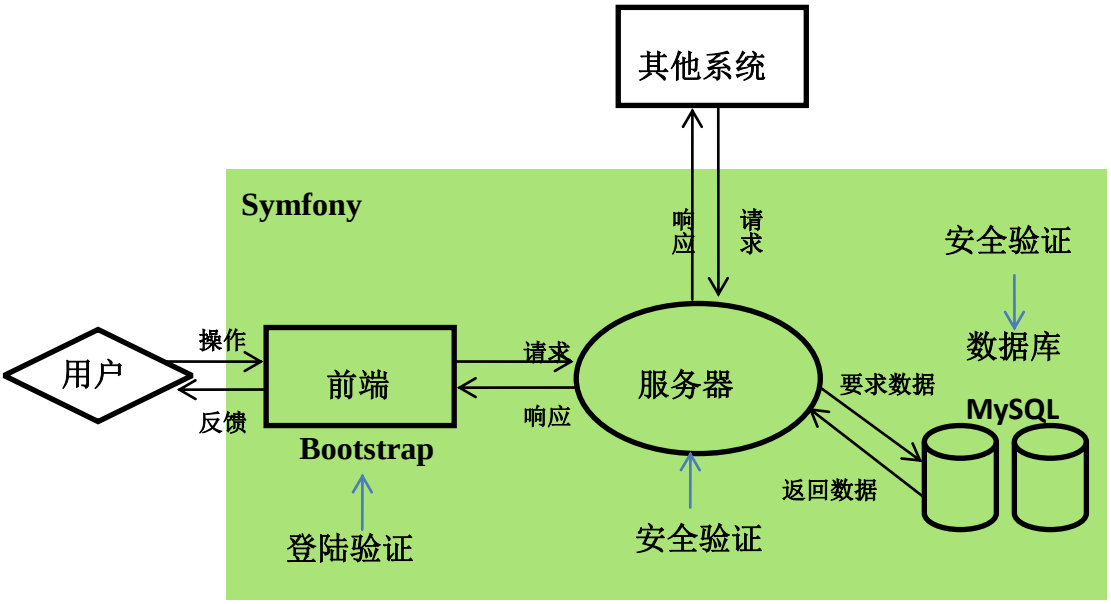
[3] Ellis, Brian, Jeffrey Stylos, and Brad Myers. "The factory pattern in API design: A usability evaluation." Proceedings of the 29th international conference on Software Engineering. IEEE Computer Society, 2007.

[4] Parsana, Mukesh D., Jayesh N. Rathod, and Jaladhi D. Joshi. "Using Factory Design Pattern for Database Connection and Daos (Data Access Objects) With Struts Framework."

2 系统架构

根据证券管理系统的实际需求可知，整个系统实际上是一个网页系统。经过组内多次讨论和查阅相关资料后，确定系统由前端、服务器、数据库组成，相互关系如图所示：

前端界面负责与用户进行交互工作，负责将用户的操作转化为请求，然后发送给服务器。服务器根据收到的请求，向数据库发送数据请求，要求数据库提供对应的数据。数据库收到服务器的数据请求后，取出对应数据，然后回发给服务器。服务器再将数据封装，传递到前端显示，最终将信息反馈给用户。同时，本系统是整个证券系统的子系统，因此其他系统也可能需要调用本数据库中的内容。对于这种情况，其他系统在需要调用数据库内容时候，也发送请求给服务器。服务器取得对应数据后，将数据发送给其他系统。整个过程中的数据交换使用 JSON 格式，与其他系统统一，以便于相互通信。



整体架构关系图

系统使用的主要组件如下：

1） 主体框架：Symfony2

2） 前端框架：Bootstrap

3） 数据库：MySQL

4） 数据交换格式：JSON

2.1 主体框架 Symfony2

整个证券账户子系统实际是一个网页系统，故可以参考常用的网页开发架构，以提高开发效率。根据需求，系统需要稳定、可靠，并具有较好的安全性。综合考虑各种因素后，我们组决定采用 Symfony2 架构。

Symfony2 是一款高性能的 PHP 网页开发框架，可以帮助开发者快速、高效的开发出符合要求的网页。整个框架功能强大，性能优秀，被业界广为采用，具有良好的口碑，是公认的较为成熟的网页开发框架。因此，证券管理子系统采用该框架来开发是完全合适和可行的。

2.2 前端框架 Bootstrap

前端负责与用户进行交互，是服务器和用户之间的桥梁。用户通过前端，直观的了解所需要的信息，并进行相关操作；服务器通过前端，获取用户的需求，并将对应的数据通过前端显示给用户。证券管理子系统的定位是一个工作系统，不需要花哨的图形界面，而需要简洁、清晰的页面结构。

Bootstrap 是一款优秀的前端框架，它包含许多 UI 组件可供开发者使用。该 UI 的设计风格简洁、优雅，符合系统的需求。借助于 Bootstrap 丰富的组件库，系统可以使用简单的代码编写出优秀的前端界面。

2.3 数据库 MySQL

证券管理子系统的数据库负责存储账户信息、管理员信息等。这些信息数据

量很大，且十分重要，所以系统需要采用比较成熟和高性能的数据库来存储。本系统采用 MySQL 作为系统的数据库。

MySQL 是目前运用的最广的数据库解决方案，具有良好的稳定性和兼容性。对于数据管理子系统来说，MySQL 完全满足系统需求，并易于开发。同时由于系统还需要保证用户数据的安全性，相比于其他的一些数据库，开源的 MySQL 更加透明和具备可定制性。

2.4 数据交换格式 JSON

证券管理子系统共有两类数据交换，分别是系统内的数据交换和与其他系统的数据交换。为了保持整个大系统的数据格式统一，系统也采用 JSON 格式封装数据。

2.5 架构特性

2.5.1 安全性

证券账户子系统要求系统具有很高的安全性。系统的数据库中存放着用户的账户信息，这些信息极其重要，只能被系统的工作人员等具有资质的人访问和修改。一旦这些数据泄露或被篡改，将会造成极大的损失和影响。因此，针对安全性，整个架构有如下的保障措施：

1） 登陆验证

为了确保系统的安全，在系统中设计了登陆验证功能，用于验证访问系统者的权限。如果访问者没有权限，则无法进入系统。

2） 多层安全验证，拒绝非法操作

系统会多层级验证请求的合法性，具体验证内容为请求的内容和发送请求者的权限。一旦任意一层验证发现请求不合法，就拒绝请求。

2.5.2 可靠性

证券账户子系统要求系统具有很好的可靠性。系统必须保持运作正常，并正确处理信息数据。根据中央交易系统的要求，可靠性通过如下机制保证：

1） 中央交易系统询问资金账户子系统和证券账户子系统，确认交易人的资金正常且账户有权进行账户交易。对于中央交易系统的询问，证券账户子系统和资金子系统会返回一个 token，告知中央交易系统交易人是否符合交易条件。

2） 如果交易人符合交易条件，中央交易系统进行交易，同时发送请求，通知资金子系统和证券账户子系统更新交易人数据；如果交易人不符合交易条件，则取消交易。

只有按照上面的流程，证券账户子系统才认为是一次合法的证券交易，会提供或修改对应数据，否则将不予回应。

3.主要设计模式

设计模式分为三种类型，创建型模式，结构型模式和行为型模式，三种类型共包括 23 种具体的设计模式，通过对我们证券账户业务子系统的特点和架构的分析，我们决定选择使用工厂方法模式作为我们的设计模式，以下将详细介绍工厂方法模式以及该模式适用于证券账户子系统的原因。

3.1.工厂方法模式

工厂方法模式是简单工厂模式的衍生，该模式属于类创建型模式，在工厂方法模式中，工厂父类负责定义创建产品对象，而工厂子类则负责生产具体的产品对象，这样做的目的是将产品类的实例化操作延迟到工厂子类中完成，即通过工厂子类来确定究竟应该实例化哪一个具体的产品。

3.1.1.工厂方法模式的优点

1）多态性设计。

基于工厂角色和产品角色的多态性设计是工厂方法模式的关键，它能够使工厂可以自主决定创建何种产品对象，而如何创建这个对象的细节完全封装在具体工厂的内部。

2）优秀扩展性。

在增加产品类的情况下，无需修改抽象工厂和抽象产品提供的接口，无需修改客户端，也无需修改其他的具体工厂和具体产品，只需要添加一个具体工厂和具体产品就可以，从而可以使得系统的扩展性变得很好。

3）屏蔽产品类。

在工厂方法模式中，工厂方法用来创建客户所需要的产品，如果调用者需要一个具体的产品对象，只要知道这个产品的类名就可以了，不用知道创建对象的过程，调用者只需要关心产品的接口，只要接口保持不表，系统中的上层模块就不要发生变化，因为产品类的实例化工作是由工厂类负责，一个产品对象具体由哪一个产品生成是由工厂类决定的，这同时也使得我们的代码结构清晰，有很好地封装性。

4）符合多个软件设计原则。

除了之前所说的优秀扩展性符合开闭原则外，工厂方法模式符合符合迪米特原则，高层模块值需要知道产品的抽象类，其他的实现类都不用关心，我不需要的就不要去交流；也符合依赖倒转原则，该模式只依赖产品类的抽象；也符合里氏替换原则，因为该模式中我们可以使用产品子类替换产品父类。

3.1.2.工厂方法模式的缺点

1）额外开销。

在添加新产品时，需要编写新的具体产品类，而且需要提供与之对应的具体工厂类，系统中类的个数将成对增加，在一定程度上增加了系统的复杂度。

2）系统抽象性。

由于考虑到系统的可扩展性，工厂方法模式将引入抽象层，这会增加系统的抽象性和理解难度。

3.1.3.工厂方法模式的使用场景

1）一个类不知道它所需要的对象的类。

在工厂方法模式中，客户端不需要知道具体产品类的类名，只需要知道所对应的工厂即可，具体的产品对象由具体的工厂类创建，客户端需要知道创建具体产品的工厂类。

2）一个类通过其子类来指定创建哪个对象。

在工厂方法模式中，对于抽象工厂类只需要提供一个创建产品的接口，而由其子类确定具体要创建的对象，在程序运行的时，子类对象将覆盖父类对象，从而使得系统更容易扩展。

3）动态指定创建对象。

客户端在使用子类对象时无需关心是哪一个工厂子类创建的产品，需要时再动态指定。

3.2.采用工厂方法模式的原因

结合上文分析的工厂方法模式的定义，优点，缺点以及使用场景，并分析证券账户业务子系统的特性，以下将详细分析采用工厂方法模式的原因。

3.2.1.符合工厂方法模式的适用场景

证券账户业务子系统的设计主要包括数据库设计，界面设计和 API 设计，主要是 API 的设计。该子系统的设计符合工厂方法模式的使用场景，主要原因有一下两点。首先为了保证和其他子系统完好的对接，我们要使我们的子系统具有良好的扩展性，与工厂方法模式的第一个使用场景相符。其次，对于我们的实际的操作，由于我们要对两种类型的账户进行操作，即个人账户和法人账户，而他们都属于账户这个父类，因此在实际操作的时候，我们将动态确定具体由哪一个子类完成该功能，并且将覆盖父类，这与工厂模式的第二个和第三个使用场景相符。可见该子系统符合工厂方法模式的使用场景。最后，其他子模块在调用我们的 API 接口时，并不需要知道我们具体是使用哪个工厂进行设计的，只需要知道对应的接口名字就可以。

3.2.2.有效利用工厂方法模式的优点

1）多态性设计。

对于 API 而言，我们如何创建对应的接口都是由我们内部决定的，而如何创建的细节也都封装在具体的工厂内部。

2）优秀扩展性。

虽然我们当前需要提供的证券账户的 API 的接口，界面以及数据库是固定的，但是我们不能保证之后整个系统是否会对证券账户提出新的需求，因此我们需要保证我们的子系统是有很好的扩展性，才能保证当增加新的需求时能够快速地改进我们的子系统。

3）屏蔽产品类。

由于我们是为其他子系统提供有证券账户的 API 接口，因此，其他子系统在使用我们的接口，即产品时，不需要知道我们的具体实现方法，他们只需要知道接口的名称，并进行相应的调用即可。并且，但我们需要改进我们内部的具体实现方法时，其他子系统并不需要担心他们的系统会无法正常执行，因为只要保证接口名正确，就能够正确使用我们提供的接口。

4）符合多个软件设计原则。

整个证券系统除了包含我们这个证券账户子系统，还包含其他多个子系统，因此我们需要尽可能地符合软件设计的原则，才能保证在子系统合并的时候不出现差错。

3.2.3.可尽量避免工厂方法模式的缺点

1）额外开销。

虽然证券账户的操作比较频繁，也有可能会要求增加新的需求，但是我们认为，新的需求增加的数量并不会太多，因为在需求分析中已经基本囊括证券账户的操作的类型，因此我们认为整个子系统的额外开销并不会太大。

2）系统抽象性。

由于我们实现的是内部的系统，因此系统的复杂性只会增加编码的难度和实现的复杂性，而我们整个系统需要实现的则是证券账户操作的简单性，因此牺牲内部系统的复杂性来满足外部系统的方便是必须的。

4. 其他支持设计模式

除了上一节介绍的工厂方法模式，我们认为我们的子系统设计还有可能会用到原型模式，以下将介绍原型模式以及该模式可能支持证券账户子系统的原因。

4.1.原型模式

原型模式是一种创建型设计模式，原型模式允许一个对象再创建另外一个可定制的对象，根本无需知道任何如何创建的细节,其工作原理是通过将一个原型

对象传给那个要发动创建的对象，这个要发动创建的对象通过请求原型对象拷贝它们自己来实施创建。

4.1.1.原型模式的优点

1） 提高系统性能

使用原型模式创建对象比直接新建一个对象在性能上要好很多。

2） 简化对象的创建

在原型模式中创建对象就像在编辑文档时复制粘贴一样十分简单方便，会大大减少接口类的编写和子类的构造。

3） 一个对象多个修改者的场景

一个对象需要提供给其他对象访问，而且各个调用者可能都需要修改其值时，可以考虑使用原型模式拷贝多个对象供调用者使用。

4） 动态指定实例化的类

原型模式可以再运行时刻增加和删除产品，可以改变值或结构以指定新对象，从而用类动态配置应用。

4.1.2.原型模式的缺点

在原型模式中，每一个类都必须配备一个克隆方法，因此在实际的操作中，会产生大量的克隆方法。并且这个克隆方法需要对类的功能进行检测，这对于全新的类来说较容易，但对已有的类进行改造时将不是件容易的事情。

4.1.3.原型模式的使用场景

1） 资源优化场景

类初始化需要消化非常多的资源，这个资源包括数据、硬件资源等，在这种情况下我们可以使用原型模式。

2） 性能和安全要求的场景

通过新建产生一个对象需要非常繁琐的数据准备或访问权限，则可以使用原型模式。

3） 一个对象多个修改者的场景

一个对象需要提供给其他对象访问，而且各个调用者可能都需要修改其值时，可以考虑使用原型模式拷贝多个对象供调用者使用。

4） 与工厂方法模式合用

实际项目中，原型模式很少单独出现，一般是和工厂方法模式一起出现，通过克隆的方法创建一个对象，然后由工厂方法提供给调用者。

4.2.原型模式可能将支持我们系统的原因

结合上文分析的原型模式的定义、优点、缺点以及使用场景，并分析证券账户业务子系统的特性，以下将详细分析原型模式可能支持我们系统的原因。

4.2.1.符合原型模式的适用场景

我们所设计的证券账户子系统符合原型模式的适用场景，主要原因有一下几点。首先之前已经介绍了，我们的主要设计模式是工厂方法模式，而原型模式是一种常与工厂方法模式合用的设计模式。其次，我们在新建一个对象的时候经常会访问数据库，从性能和安全性来考虑，也十分符合原型模式的适用场景。

4.2.2.有效弥补了工厂方法模式的不足

在之前工厂方法模式一节中，我们可以看到，虽然工厂方法模式功能强大，也比较符合我们的设计场景，但是工厂方法模式也有他不可避免的缺点，即工厂方法模式在创建子对象时要增加一个类层次，而原型模式不需要，这可以大大简化我们创建子类的复杂度。

4.2.3.为什么不是我们的主要设计模式

首先原型模式并不是一种经常单独存在的设计模式，它常常与工厂设计模式共同出现，其次，我们系统里有两种账户：个人账户和法人账户。 所以两个不同的账户不能属于同一个“账户”类，原型方法不允许新对象拥有与父对象不同的方法。所以这样的话，不能动态确定子类的功能。换言之，对随时变化的要求不能随时应付。这样的话，最后整个股票系统整合的时候，出现问题的话，很难

对付。那时候的代价会很大，基本上是重新安排 API 接口。这样的话该类的多态性和扩展性是基本上很低。所以在这个点上原型模式不太适合。因此，我们将原型模式作为我们的辅助设计模式而不是主要设计模式。