廣東技術師範大學
Guangdong Polytechnic Normal University

《系统分析与设计》课程

Instructor: Wen Jianfeng
Email: wjfgdin@qq.com

系统分析与设计

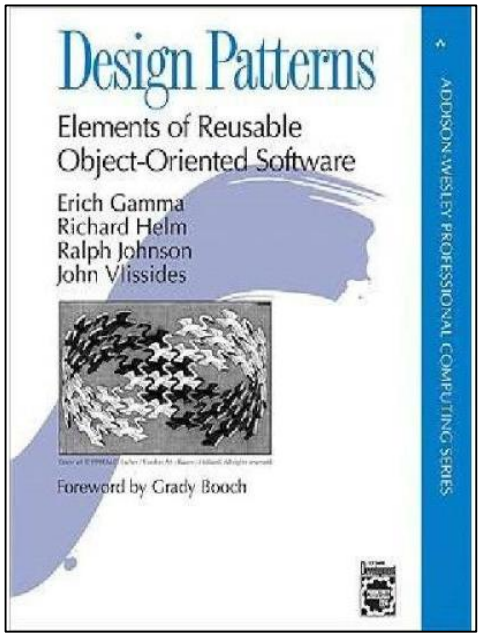
第5讲：设计模式-创建型

提纲

- 设计模式概述
- 创建型设计模式
  - 单例模式
  - 工厂模式
    - 简单工厂
    - 工厂方法
    - 抽象工厂

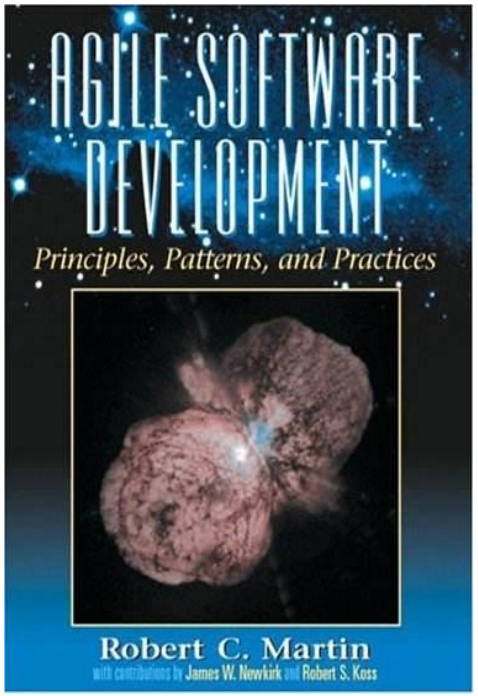
设计模式参考书一

- 书名：Design-Patterns： Elements of Reusable Object-Oriented Software (1995)
  【设计模式：可复用面向对象软件的基础】
  - 作者：Erich Gamma、Richard Helm、Ralph Johnson 、John Vlissides；也称为：GoF（Gang of Four）
  - 出版社：Addision Wesley



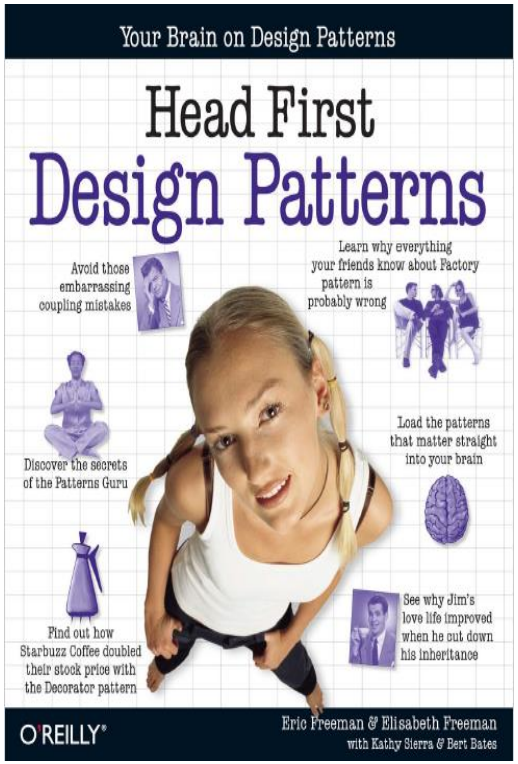
设计模式参考书二

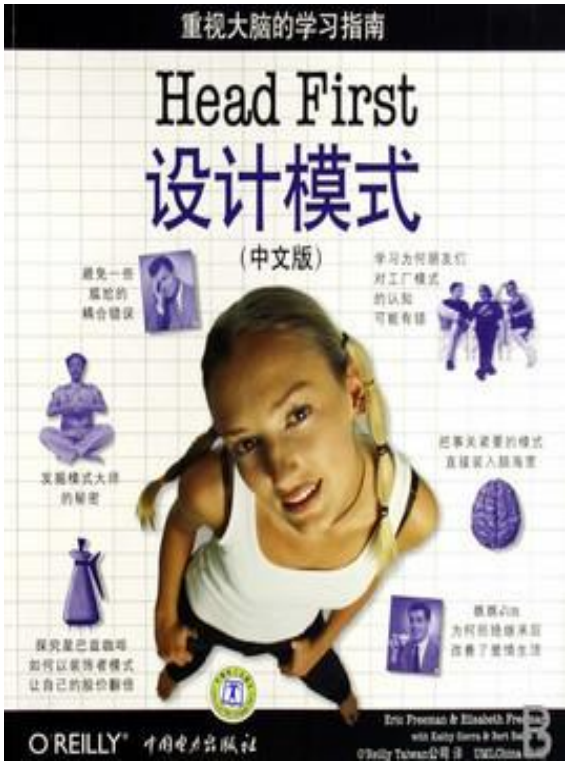
- 书名：Agile Software Development: Principles, Patterns, and Practices (2002)
  【敏捷软件开发：原则、模式与实践】
  - 作者：Robert C. Martin
  - 出版社：Pearson



设计模式参考书三

- 书名：Head First Design Patterns (2004)
  【Head First 设计模式】
  - 作者：Eric Freeman, Elisabeth Freeman
  - 出版社：O'Reilly





设计模式参考资源

- 设计模式之美
  - 极客时间，王争

- 一些有用链接
  - Awesome Software and Architecture Design Patterns
    - https://github.com/DovAmir/awesome-design-patterns
  - Java Design Pattern
    - https://java-design-patterns.com/patterns/
  - Design patterns implemented in Java
    - https://github.com/iluwatar/java-design-patterns
  - Source Making
    - https://sourcemaking.com/

设计模式参考资源

- 菜鸟教程-设计模式
  - https://www.runoob.com/design-pattern/design-pattern-tutorial.html
- 图说设计模式
  - https://design-patterns.readthedocs.io/zh_CN/latest/index.html
- Design Patterns for Humans
  - https://github.com/kamranahmedse/design-patterns-for-humans

- 老师的课程示例代码所在的仓库地址
  - https://gitee.com/wenjianfeng/sad-student

设计模式概述

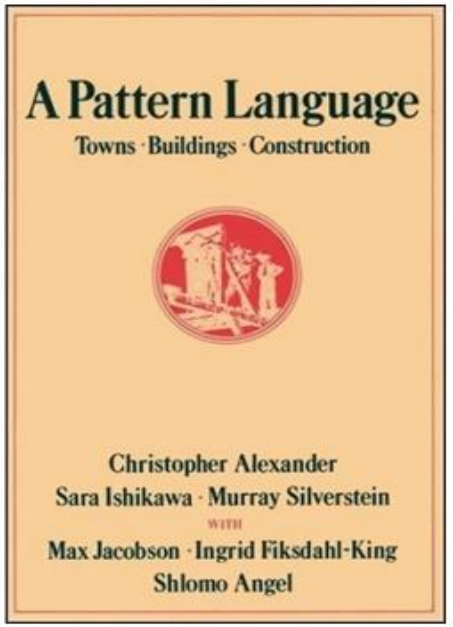
模式起源

- 模式起源于建筑业
- 模式（Pattern）之父：Christopher Alexander
  - A Pattern Language: Towns, Buildings, Construction, 1977

Christopher Alexander
(1936 --)

Architect

Harvard University (Ph.D)
MIT, UC Berkeley Professor



模式概念

- A pattern is a solution to a problem in a context
  （模式是在特定环境中解决问题的一种方案）

- 模式
  - Context（模式可适用的前提条件）
  - Theme或Problem（在特定条件下要解决的目标问题）
  - Solution（对目标问题的解决方案）

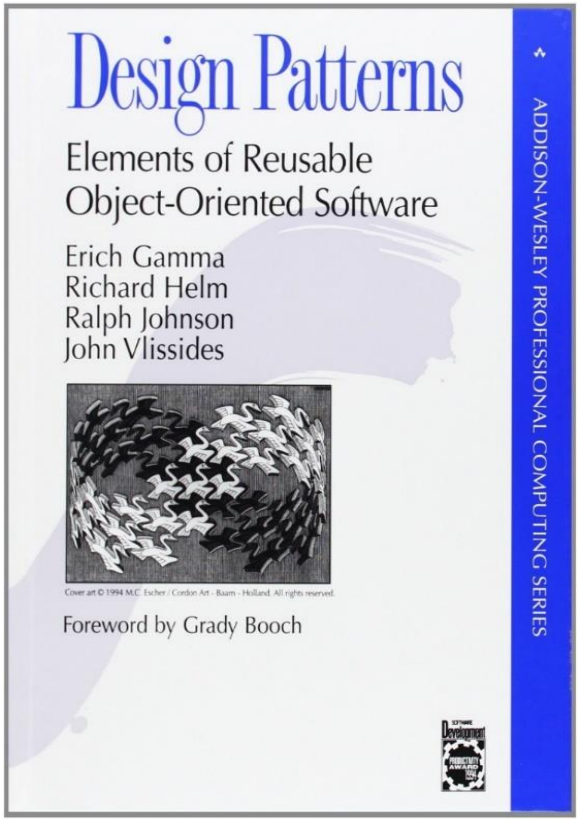
  - 每个模式都描述了一个在我们的环境中不断出现的问题，然后描述了该问题的解决方案的核心，通过这种方式，我们可以无数次地重用那些已有的解决方案，无需再重复相同的工作

软件模式

- 软件模式
  - 1990年，软件工程界开始关注Christopher Alexander等在建筑领域的模式的思想
    - 软件模式包括：架构模式、分析模式、设计模式、并发模式、过程模式
    - 更多关于软件设计模式的介绍：https://en.wikipedia.org/wiki/Software_design_pattern
  - 最早将模式的思想引入软件工程方法学的是“四人帮”（Gang of Four，GoF）
    - 即Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides四位著名软件工程学者
  - GoF在1995年归纳发表了23种在软件开发中使用频率较高的设计模式，旨在用模式来统一沟通面向对象方法在分析、设计和实现间的鸿沟
    - Design patterns : elements of reusable object-oriented software

Gang of Four, GoF

- GoF



Gang of Four, GoF

- Erich Gamma

  - Born 1961~ , Zurich, Switzerland
  - PhD, University of Zurich
  - Developed JUnit (with Kent Beck)
  - Led the design of Eclipse JDT (Java Development Tools)
  - Join Microsoft Switzerland in 2011, Visual Studio Code

Gang of Four, GoF

- Richard Helm

  - PhD, University of Melbourne
  - Partner & Managing Director, Boston Consulting Group, Sydney

Gang of Four, GoF

- Ralph Johnson

  - PhD, Computer Science, Cornell University
  - Research Associate Professor in the Department of Computer Science at the University of Illinois at Urbana-Champaign

- John Vlissides

  - Born 1961, Died 2005
  - PhD, Stanford University

设计模式的定义

- 设计模式的定义
  - 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

- 设计模式的基本要素
  - 模式名称（Pattern name）
  - 问题（Problem）
  - 解决方案（Solution）
  - 效果（Consequences）
  - 实例代码（Sample Code）
  - 相关设计模式（Related Patterns）

设计模式的分类

- 设计模式的分类
  - 根据其目的（模式是用来做什么的）可分为创建型（Creational），结构型（Structural）和行为型（Behavioral）三种：
    - 创建型模式主要用于创建对象
    - 结构型模式主要用于处理类或对象的组合
    - 行为型模式主要用于描述对类或对象怎样交互和怎样分配职责
  - 根据范围，即模式主要是用于处理类之间关系还是处理对象之间的关系，可分为类模式和对象模式两种：
    - 类模式处理类和子类之间的关系，这些关系通过继承建立，在编译时刻就被确定下来，是属于静态的
    - 对象模式处理对象间的关系，这些关系在运行时刻变化，更具动态性

设计模式的分类

		Purpose 目的		
		Creational 创建型（5种）	**Structural** 结构型（7种）	**Behavioral** 行为型（11种）
Scope 范围	**Class** 类	Factory Method [工厂方法]	Adapter (class) [适配器（类）]	Interpreter [解释器] Template Method [模板方法]
	Object 对象	Abstract Factory [抽象工厂] Builder [建造者 / 生成器] Prototype [原型] Singleton [单例]	Adapter (object) [适配器（对象）] Bridge [桥接] Composite [组合] Decorator [装饰器] Façade [门面 / 外观] Flyweight [享元] Proxy [代理]	Chain of Responsibility [责任链 / 职责链] Command [命令] Iterator [迭代器] Mediator [中介者] Memento [备忘录] Observer [观察者] State [状态] Strategy [策略] Visitor [访问者]

创建型设计模式（Creational）

- Factory Method（工厂方法）
  - Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.

    定义一个用于创建对象的接口，让子类决定将哪一个类实例化。工厂方法使一个类的实例化延迟到其子类

- Abstract Factory（抽象工厂）
  - Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

    提供一个创建一系列相关或相互依赖对象的接口，而无需指定它们具体的类

创建型设计模式（Creational）

- Builder（建造者 / 生成器）
  - Separate the construction of a complex object from its representation so that the same construction process can create different representations.

    将一个复杂对象的创建与它的表示分离，使得同样的创建过程可以创建不同的表示

- Prototype（原型）
  - 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，并通过拷贝这个原型来创建新的对象

创建型设计模式（Creational）

- Singleton（单例）
  - Ensure a class only has one instance, and provide a global point of access to it.
    保证一个类仅有一个实例，并提供一个访问它的全局访问点

结构型设计模式（Structural）

- Adapter（适配器）
  - Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.

    将一个类的接口转换成客户希望的另外一个接口。适配器使得原本由于接口不兼容而不能一起工作的那些类可以一起工作

- Bridge（桥接）
  - Decouple an abstraction from its implementation so that the two can vary independently.

    将抽象部分与它的实现部分分离，使它们都可以独立地变化

结构型设计模式（Structural）

- Composite（组合）
  - Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

    将对象组合成树形结构以表示“部分-整体”的层次结构，使得客户对单个对象和复合对象的使用具有一致性

- Decorator（装饰器）
  - Attach additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

    动态地给一个对象添加一些额外的职责。就扩展功能而言，装饰模式比生成子类方式更为灵活

结构型设计模式（Structural）

- Facade（门面 / 外观）
  - Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.

    为子系统中的一组接口提供一个统一的接口。门面模式定义了一个高层接口，这个接口使得这一子系统更加容易使用

- Flyweight（享元）
  - Use sharing to support large numbers of fine-grained objects efficiently.

    运用共享技术有效地支持大量细粒度的对象

结构型设计模式（Structural）

- Proxy（代理）
  - Provide a surrogate or placeholder for another object to control access to it.
    为其他对象提供一个代理以控制对这个对象的访问

行为型设计模式（Behavioral）

- Interpreter（解释器）
  - Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.

    给定一个语言，定义文法表示，定义解释器。该解释器使用该表示来解释语言中的句子
- Template Method（模板方法）
  - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.

    定义一个操作中的算法骨架，将一些步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法中某些特定步骤

行为型设计模式（Behavioral）

- Chain of Responsibility（责任链 / 职责链）
  - Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.

    通过给予多个对象一个机会处理请求，使请求的发送者和接收者解耦。将接收请求的对象链接起来，并使请求沿着这个链条传递下去直至有一个对象处理它
- Command（命令）
  - Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

    将一个请求封装为一个对象，使得可以用不同的请求对客户进行参数化，对请求排队或记录请求日志，支持撤销操作

行为型设计模式（Behavioral）

- Iterator（迭代器）
  - Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

    提供一种方法顺序访问一个集合对象中各个元素，而无需暴露该对象的内部表示
- Mediator（中介者）
  - Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently.

    用一个中介对象来封装一系列的对象交互。中介者使各对象不需要显式地相互引用（松散耦合），而且可以独立地改变它们之间的交互

行为型设计模式（Behavioral）

- Memento（备忘录）
  - Without violating encapsulation, capture and externalize an object's internal state so that the object can be restored to this state later.

    在不破坏封装性的前提下，捕获一个对象的内部状态，并在该对象之外保存这个状态，便于将对象恢复到保存的状态

- Observer（观察者）
  - Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

    定义对象间一对多的依赖关系，当一个对象状态发生变化时，所以依赖于它的对象都将得到通知并自动刷新

行为型设计模式（Behavioral）

- State（状态）
  - Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.

    允许一个对象在其内部状态改变时改变它的行为。对象看起来似乎修改了它所属的类

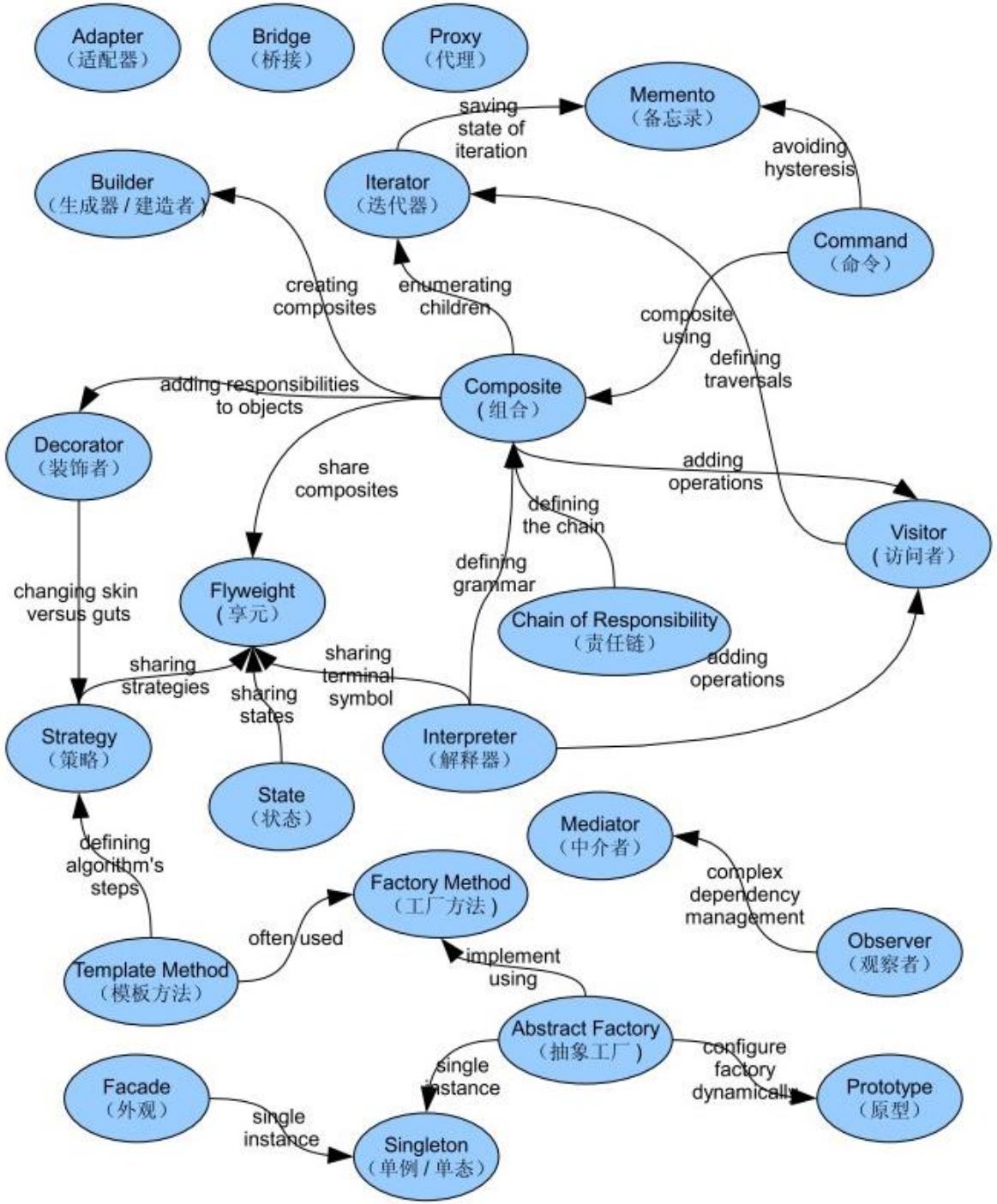
- Strategy（策略）
  - Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.

    定义一系列的算法，对其进行封装，使其可以互相替换。策略模式使得算法的变化可以独立于使用它的客户

行为型设计模式（Behavioral）

- Visitor（访问者）
  - Represent an operation to be performed on the elements of an object structure. Visitor lets you define a new operation without changing the classes of the elements on which it operates.

    表示一个作用于某对象结构中的各元素的操作。访问者模式使得可以在不改变各元素的类的前提下定义作用于这些元素的新操作


Adapter
（适配器）
Bridge
（桥接）
Proxy
（代理）
Memento
（备忘录）
saving
state of
iteration
avoiding
hysteresis
Builder
（生成器 / 建造者）
Iterator
（迭代器）
Command
（命令）
creating
composites
enumerating
children
composite
using
defining
traversals
adding responsibilities
to objects
Composite
（组合）
Decorator
（装饰者）
share
composites
adding
operations
defining
the chain
Visitor
（访问者）
changing skin
versus guts
Flyweight
（享元）
defining
grammar
Chain of Responsibility
（责任链）
adding
operations
sharing
strategies
sharing
terminal
symbol
sharing
states
Strategy
（策略）
Interpreter
（解释器）
State
（状态）
Mediator
（中介者）
defining
algorithm's
steps
Factory Method
（工厂方法）
complex
dependency
management
Observer
（观察者）
often used
Template Method
（模板方法）
implement
using
Abstract Factory
（抽象工厂）
configure
factory
dynamically
single
instance
Facade
（外观）
Prototype
（原型）
single
instance
Singleton
（单例 / 单态）

创建型设计模式
Creational

创建型设计模式

- Creational patterns are focused towards how to instantiate an object or group of related objects.
  - 创建型设计模式专注于如何实例化一个对象或一组相关的对象

创建型设计模式

- In software engineering, creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.
  - 创建型设计模式处理对象的创建机制，尝试以适合应用情形的方式创建对象。传统的对象创建方式可能会导致设计问题或增加设计的复杂性，创建型设计模式通过某种控制对象的创建方式来解决此问题

		Purpose 目的		
		Creational 创建型（5种）	**Structural** 结构型（7种）	**Behavioral** 行为型（11种）
Scope 范围	**Class** 类	**Factory Method** [工厂方法]	Adapter (class) [适配器（类）]	Interpreter [解释器] Template Method [模板方法]
	Object 对象	**Abstract Factory** [抽象工厂] **Builder** [建造者 / 生成器] **Prototype** [原型] **Singleton** [单例]	Adapter (object) [适配器（对象）] Bridge [桥接] Composite [组合] Decorator [装饰器] Façade [门面 / 外观] Flyweight [享元] Proxy [代理]	Chain of Responsibility [责任链 / 职责链] Command [命令] Iterator [迭代器] Mediator [中介者] Memento [备忘录] Observer [观察者] State [状态] Strategy [策略] Visitor [访问者]

单例模式
Singleton

现实世界的例子

- 单例模式—现实世界的例子
  - There can only be one president of a country at a time. The same president has to be brought to action, whenever duty calls. President here is singleton.

    同一时刻一个国家只能有一位总统。每当有任务需要总统处理时，都是这位总统出面处理。在这里，总统便是“单例”。

单例模式 – Intent【意图】

- Ensure a class only has one instance, and provide a global point of access to it.

  保证一个类仅有一个实例，并提供一个访问它的全局访问点

单例模式 – Motivation【动机】

- It's important for some classes to have exactly one instance.

  有些类有时只需要唯一的实例

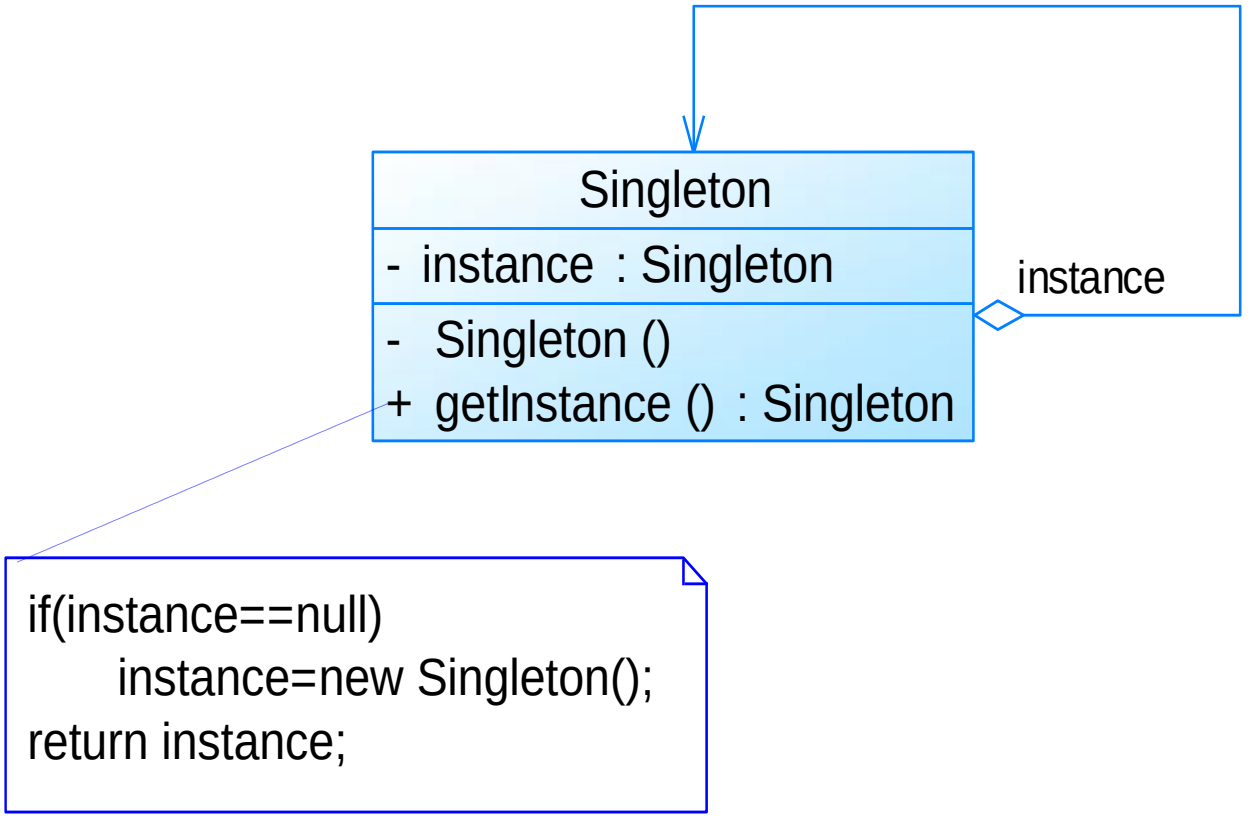
  - 如：数据库连接类、配置信息类、唯一递增ID号码生成器类 …
  - 如：线程池、对话框、日志 …

- How do we ensure that a class has only one instance and that the instance is easily accessible?

  如何保证一个类只有一个实例并且这个实例易于被访问呢？

  - 让类自身负责保存它的唯一实例
  - 保证没有其他实例被创建(通过监视新对象创建的请求)
  - 提供一个访问该实例的方法

单例模式 – Structure【结构】



单例模式 – Implementation【实现】

```
public class Singleton {
  //静态私有成员变量
   private static Singleton instance;

  // … … 其他成员变量

   private Singleton() { //私有构造方法
   }

  //静态公有方法，返回唯一实例
   public static Singleton
   getInstance() {
        if (instance == null)
            instance = new Singleton();
        return instance;
   }

  // … … 其他方法
}
```

- 单例类拥有一个私有构造方法，确保用户无法通过new关键字直接实例化它
- 包含一个静态私有成员变量与静态公有的方法，该方法负责检验实例的存在性并实例化自己，然后存储在成员变量中

单例模式 – 多线程的情形

- 解决方案一：同步getInstance()方法
  - 虽然，可以确保多线程情形下实例的唯一性
  - 但是，在getInstance()方法上同步是昂贵且浪费的，因为只有第一次调用才有必要同步，后续并无同步的必要

```
public class Singleton {
   private static Singleton instance;
   private Singleton() {
   }

  // 同一时间只允许一个线程进入该方法
   public static synchronized Singleton getInstance() {
        if (instance == null)
            instance = new Singleton();
        return instance;
   }
}
```

单例模式 – 多线程的情形

- 解决方案二：Eagerly Created Instance
  - 在类装载时创建实例，能确保是线程安全的

```
public class Singleton {
    // 饿汉式创建实例
    private static Singleton instance = new Singleton();

    private Singleton() {
    }

    // 实例已创建，直接返回实例
    public static Singleton getInstance() {
        return instance;
    }
}
```

单例模式 – 多线程的情形

- 解决方案三：Double-checked Locking
  - 先检测是否存在实例，如果没有才进入同步块

```
public class Singleton {
   private volatile static Singleton instance;
   private Singleton() {
   }

   public static Singleton getInstance() {
      // 只在第一次调用时才进入同步块
      if (instance == null) {
        synchronized (Singleton.class) {
          if (instance == null)
            instance = new Singleton();
        }
      }
      return instance;
   }
}
```

单例模式 – 举例

- 举例：实现一个往文件中打印日志的类

```
public class Logger {
    private FileWriter writer; // FileWriter是对象级线程安全的
    private static final Logger instance = new Logger();

    private Logger() {
        File file = new File("/Users/wangzheng/log.txt");
        writer = new FileWriter(file, true); // true表示追加写入
    }

    public static Logger getInstance() {
        return instance;
    }

    public void log(String message) {
        writer.write(message);
    }
}
```

单例模式 – 举例

```
// Logger类的应用示例：
public class UserController {
    public void login(String username, String password) {
        // ...省略业务逻辑代码...
        Logger.getInstance().log(username + " logined!");
    }
}
```

```
public class OrderController {
    public void create(OrderVo order) {
        // ...省略业务逻辑代码...
        Logger.getInstance().log("Created a order: " +
   order.toString());
    }
}
```

单例模式 – 优缺点

- 单例模式的优点
  - 单例模式在创建后在内存中只存在一个实例，节约了内存开支和实例化时的性能开支，特别是需要重复使用一个创建开销比较大的类时，比起实例不断地销毁和重新实例化，单例能节约更多资源
  - 单例模式可以解决对资源的多重占用，比如写文件操作时，因为只有一个实例，可以避免对一个文件进行同时操作
  - 单例模式只使用一个实例，也可以减小垃圾回收机制 GC 的压力

单例模式 – 优缺点

- 单例模式的缺点
  - 单例模式对扩展不友好，一般不容易扩展，因为单例模式一般自行实例化，没有接口
  - 与单一职责原则冲突，一个类应该只关心内部逻辑，而不关心外面怎么样来实例化

单例模式 –JDK中的应用

- Java中的例子
  - Runtime类的getRuntime()方法

static Runtime	getRuntime() 返回与当前 Java 应用程序相关的运行时对象。

```
/** Don't let anyone else instantiate this class */
private Runtime() {}
```

  - Runtime可用于执行外部命令，如：
    - Process proc = Runtime.getRuntime().exec("notepad.exe")

工厂模式
Factory

重温依赖倒置原则

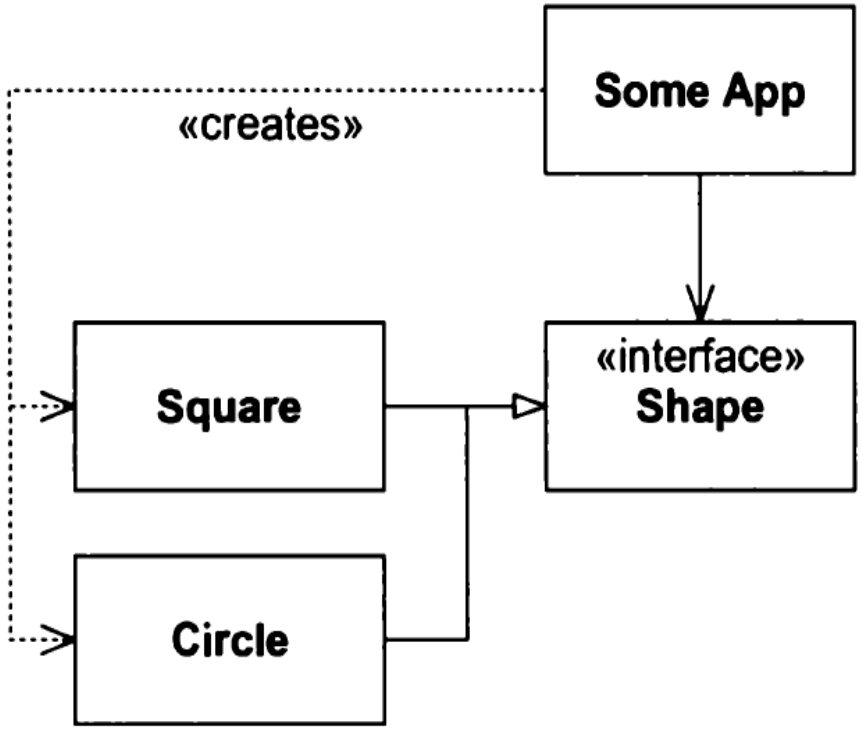
- 依赖倒置原则（Dependency-Inversion Principle）告诉我们应该依赖抽象类，而避免依赖具体类，特别是对于那些易发生变化的类

- 当然，如果所依赖的类是不易变化的，则依赖它也没有任何问题，如：new String(“abc”)是非常安全的，因为String类不太可能发生变化

- 工厂模式允许我们仅依赖抽象接口去创建具体类的实例

重温依赖倒置原则

- 违反DIP的设计
  - SomeApp创建Square和Circle的实例，因而不得不依赖于这两个具体类



  - 用工厂模式解决此类问题

工厂模式

- 工厂模式可分为三种类型
  - 简单工厂【Simple Factory】
  - 工厂方法【Factory Method】
  - 抽象工厂【Abstract Factory】
  - 注：简单工厂模式并不属于 GoF 的23个经典设计模式，在 GoF 的《设计模式》一书中，它将简单工厂模式看作是工厂方法模式的一种特例

简单工厂模式
Simple Factory

现实世界的例子

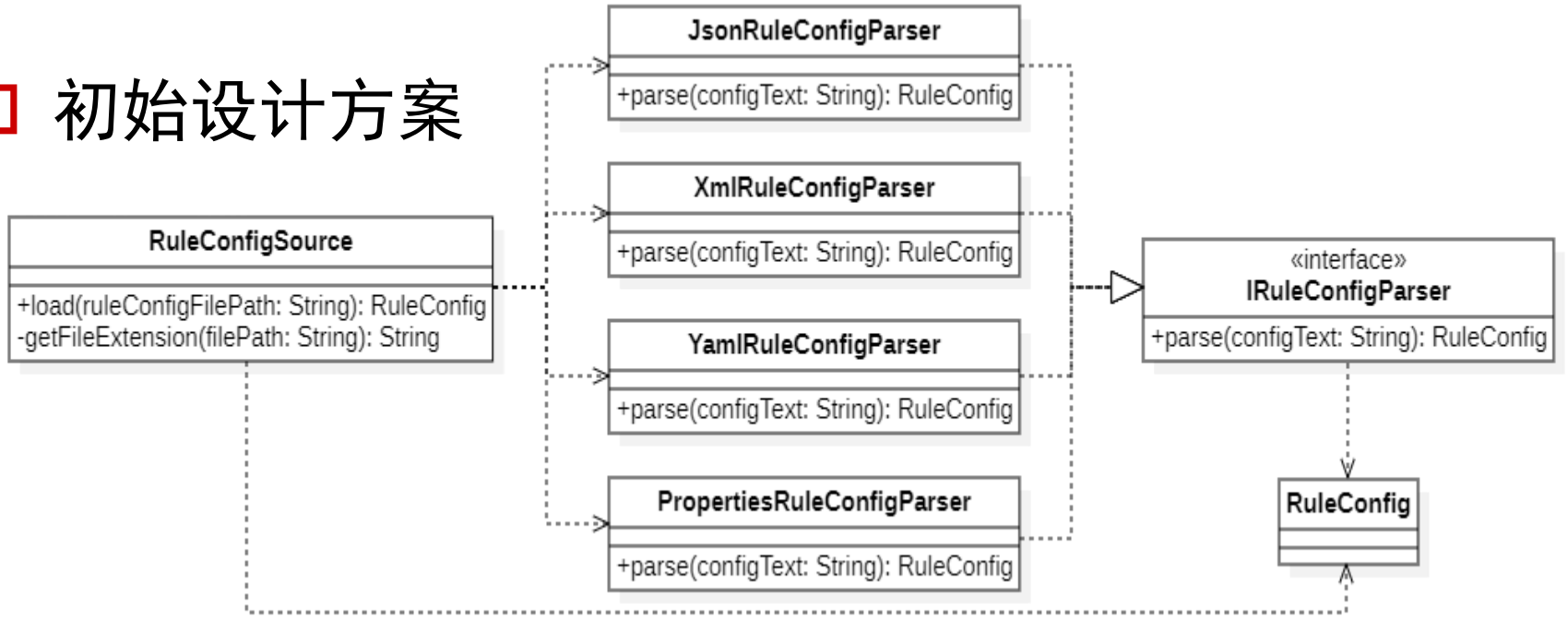
- 简单工厂模式--现实世界的例子
  - Consider, you are building a house and you need doors. You can either put on your carpenter clothes, bring some wood, glue, nails and all the tools required to build the door and start building it in your house or you can simply call the factory and get the built door delivered to you so that you don't need to learn anything about the door making or to deal with the mess that comes with making it.

    考虑一下，你正在建房子，你需要门。你可以穿上你的木匠衣服，带上一些木头，胶水，钉子和建造门所需的所有工具，然后开始在你的房子里建造它；或者你可以简单地打电话给工厂并把已做好的门送到你这里，因此你不需要学习关于制作门的任何知识或处理制作它时所带来的混乱。

简单工厂模式 – Motivation【动机】

- 通过一个案例引入
  - 根据配置文件的后缀（json、xml、yaml、properties），选择不同的解析器（JsonRuleConfigParser、XmlRuleConfigParser……），将存储在文件中的配置解析成内存对象 RuleConfig

- 初始设计方案



简单工厂模式 – Motivation【动机】

```
public class RuleConfigSource {
    public RuleConfig load(String ruleConfigFilePath) {
        String ruleConfigFileExtension =
   getFileExtension(ruleConfigFilePath); // 获取扩展名（略）
        IRuleConfigParser parser = null;

        if ("json".equalsIgnoreCase(ruleConfigFileExtension)) {
            parser = new JsonRuleConfigParser();
        } else if ("xml".equalsIgnoreCase(ruleConfigFileExtension)) {
            parser = new XmlRuleConfigParser();
        } else if ("yaml".equalsIgnoreCase(ruleConfigFileExtension)) {
            parser = new YamlRuleConfigParser();
        } else if ("properties".equalsIgnoreCase(ruleConfigFileExtension)) {
            parser = new PropertiesRuleConfigParser();
        }

        String configText = "";
        // ... 从ruleConfigFilePath文件中读取配置文本到configText中（略）
        RuleConfig ruleConfig = parser.parse(configText);
        return ruleConfig;
    }
}
```

简单工厂模式 – Motivation【动机】

为了让代码逻辑更加清晰，可读性更好，可以将功能独立的代码块封装成函数

按照这个设计思路，我们也可以将代码中涉及 parser 创建的部分逻辑剥离出来，抽象成 createParser() 函数。

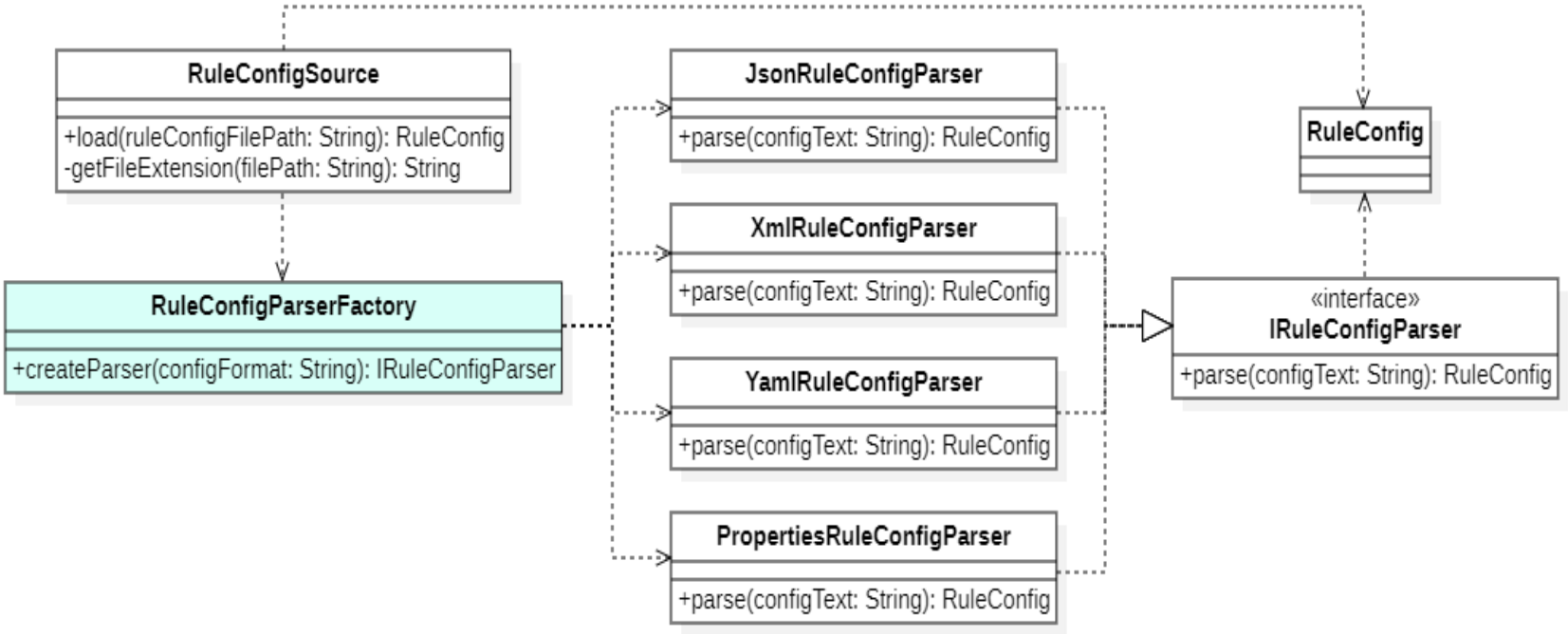
```
private IRuleConfigParser createParser(String configFormat) {
   IRuleConfigParser parser = null;
   if ("json".equalsIgnoreCase(configFormat)) {
     parser = new JsonRuleConfigParser();
   } else if ("xml".equalsIgnoreCase(configFormat)) {
     parser = new XmlRuleConfigParser();
   } else if ("yaml".equalsIgnoreCase(configFormat)) {
     parser = new YamlRuleConfigParser();
   } else if ("properties".equalsIgnoreCase(configFormat)) {
     parser = new PropertiesRuleConfigParser();
   }
   return parser;
 }
```

简单工厂模式 – Motivation【动机】

- 上述方案存在的问题
  - RuleConfigSource类中包含很多“if…else…”代码块
  - RuleConfigSource类的职责过重，违反了“单一职责原则”
  - 当需要解析新的文件类型时，必须修改RuleConfigSource类，违反了“开闭原则”

简单工厂模式 – 重构后

- 上述案例的重构
  - 为了让类的职责更加单一、代码更加清晰，可以进一步将 createParser() 函数剥离到一个独立的类 RuleConfigParserFactory 中，让这个类只负责对象的创建。而这个类就是简单工厂类



简单工厂模式 – 重构后

```
public RuleConfig load(String ruleConfigFilePath) {
   String ruleConfigFileExtension =
       getFileExtension(ruleConfigFilePath); // 获取扩展名（略）

   IRuleConfigParser parser =
    RuleConfigParserFactory.createParser(ruleConfigFileExtension);

   String configText = "";
   // ... 从ruleConfigFilePath文件中读取配置文本到configText中（略）
   RuleConfig ruleConfig = parser.parse(configText);
   return ruleConfig;
 }
```

简单工厂模式 – 重构后

```
public class RuleConfigParserFactory {

  public static IRuleConfigParser createParser(String configFormat) {
        IRuleConfigParser parser = null;
        if ("json".equalsIgnoreCase(configFormat)) {
            parser = new JsonRuleConfigParser();
        } else if ("xml".equalsIgnoreCase(configFormat)) {
            parser = new XmlRuleConfigParser();
        } else if ("yaml".equalsIgnoreCase(configFormat)) {
            parser = new YamlRuleConfigParser();
        } else if ("properties".equalsIgnoreCase(configFormat)) {
            parser = new PropertiesRuleConfigParser();
        }
        return parser;
  }
}
```

简单工厂模式 – 分析

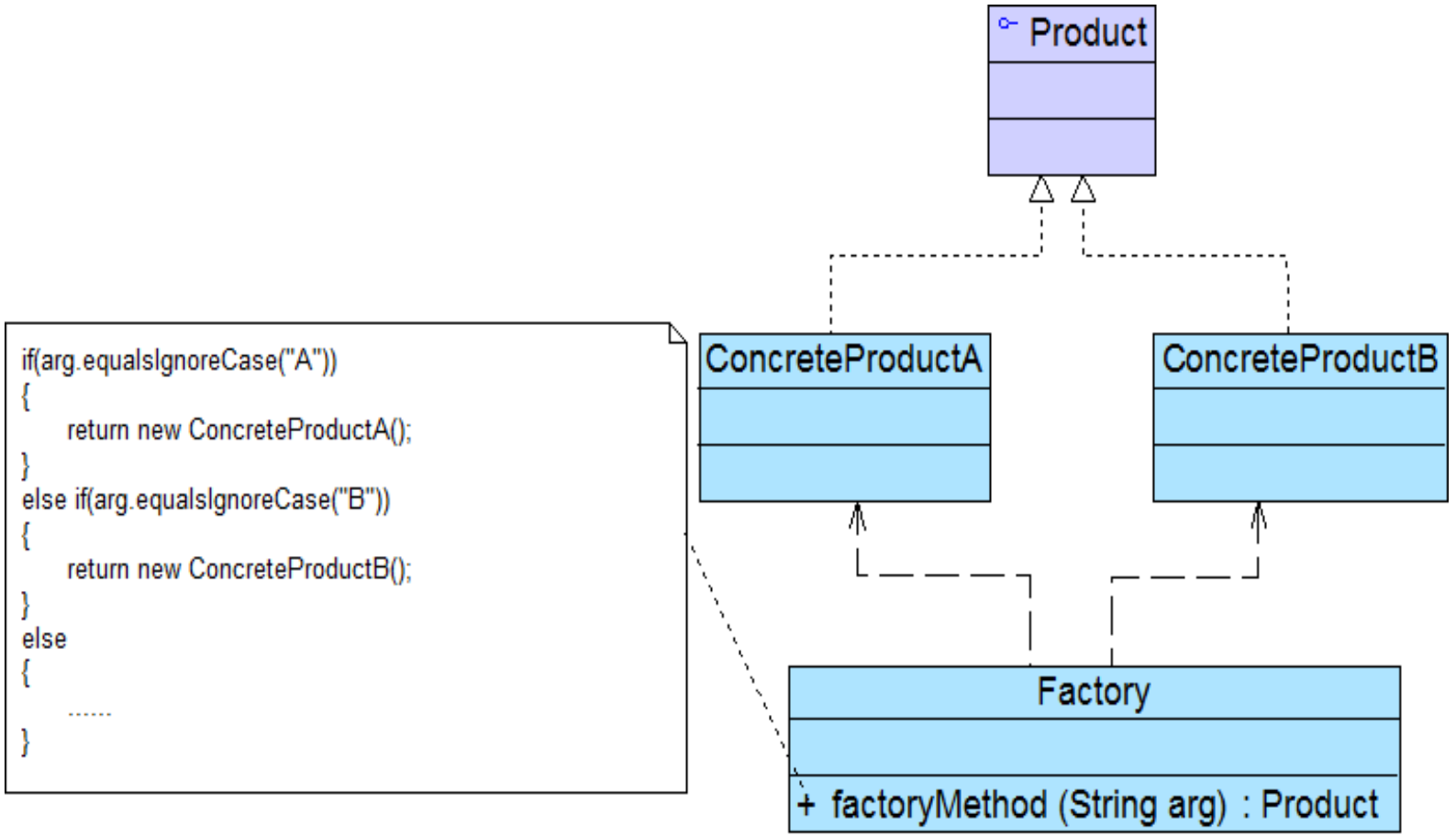
- 上例中，如果要添加新的 parser，要改动 RuleConfigParserFactory 的代码，这是不是违反开闭原则呢？
  - 实际上，如果不是需要频繁地添加新的 parser，只是偶尔修改一下 RuleConfigParserFactory 代码，稍微不符合开闭原则，也是完全可以接受的
- 上例中，有一组 if 分支判断逻辑，是不是应该用多态或其他设计模式来替代呢？
  - 实际上，如果 if 分支并不是很多，代码中有 if 分支也是完全可以接受的
- 总结
  - 尽管简单工厂模式违背开闭原则，但权衡扩展性和可读性，在大多数情况下是没有问题的

简单工厂模式 – Definition【定义】

- 简单工厂模式（Simple Factory Pattern），又称为静态工厂方法（Static Factory Method）模式
  - 创建产品对象的工厂方法也可以声明为静态的，即可以不实例化工厂类
- 在简单工厂模式中，可以根据参数的不同返回不同类的实例
- 简单工厂模式专门定义一个类来负责创建其他类的实例，被创建的实例通常都具有共同的父类

简单工厂模式 – Structure【结构】

- 简单工厂模式的类图

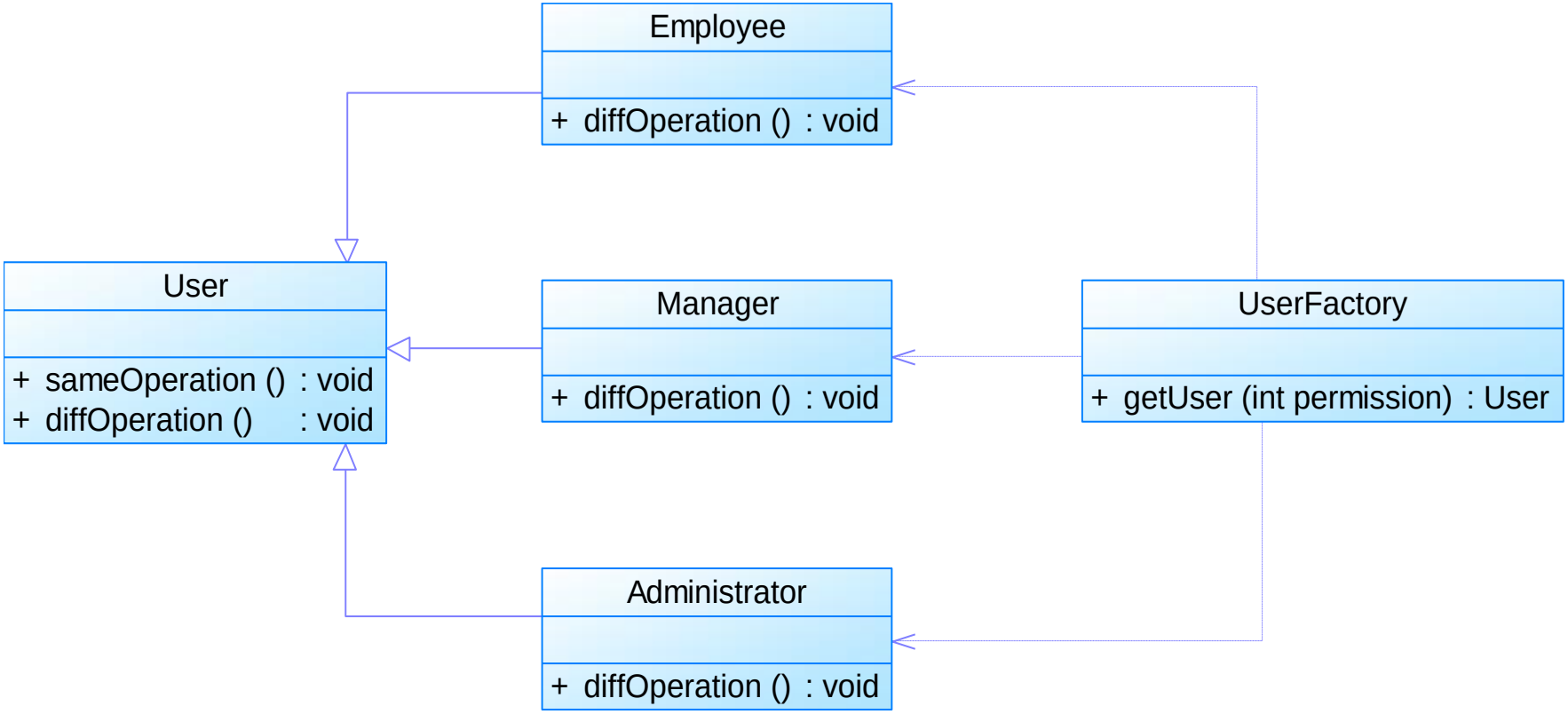


简单工厂模式 – 举例

- 例子：系统的权限管理
  - 系统根据用户在登录时输入的账号和密码以及在数据库中存储的账号和密码是否一致来进行身份验证
  - 如果验证通过，则取出存储在数据库中的用户权限等级（以整数形式存储），根据不同的权限等级创建不同等级的用户对象，不同等级的用户对象拥有不同的操作权限

简单工厂模式 – 举例

- 设计如图：



简单工厂模式 – 优缺点

- 简单工厂模式的优点
  - 工厂类含有必要的判断逻辑，可以决定在什么时候创建哪一个产品类的实例，客户可以免除直接创建产品对象的责任，而仅仅“消费”产品；简单工厂模式通过这种做法实现了对责任的分割，它提供了专门的工厂类用于创建对象
  - 客户无须知道所创建的具体产品类的类名，只需要知道具体产品类所对应的参数即可，对于一些复杂的类名，通过简单工厂模式可以减少使用者的记忆量
  - 通过引入配置文件，可以在不修改任何客户代码的情况下更换和增加新的具体产品类，在一定程度上提高了系统的灵活性

简单工厂模式 – 优缺点

- 简单工厂模式的缺点
  - 由于工厂类集中了所有产品创建逻辑，一旦不能正常工作，整个系统都要受到影响
  - 使用简单工厂模式将会增加系统中类的个数，在一定程度上增加了系统的复杂度和理解难度
  - 系统扩展困难，一旦添加新产品就不得不修改工厂逻辑，在产品类型较多时，有可能造成工厂逻辑过于复杂，不利于系统的扩展和维护

简单工厂模式 – 适用情形

- 工厂类负责创建的对象比较少
  - 由于创建的对象较少，不会造成工厂方法中的业务逻辑太过复杂
- 客户只知道传入工厂类的参数，对于如何创建对象不关心
  - 客户既不需要关心创建细节，甚至连类名都不需要记住，只需要知道类型所对应的参数

应用

- 在JDK类库中广泛使用了简单工厂模式，如工具类java.text.DateFormat，它用于格式化一个本地日期或者时间。

```
public final static DateFormat getDateInstance();

public final static DateFormat getDateInstance(int style);

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

工厂方法模式
Factory Method

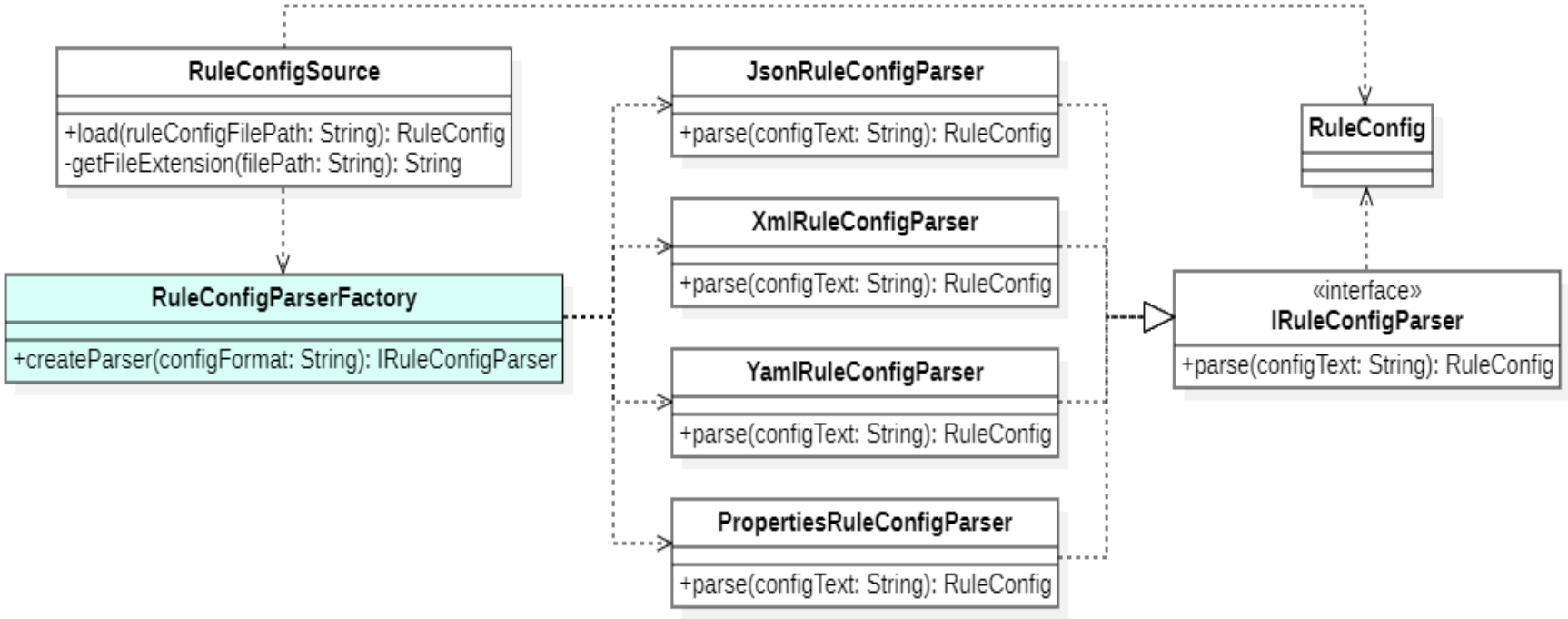
现实世界的例子

- 工厂方法模式—现实世界的例子
  - Consider the case of a hiring manager. It is impossible for one person to interview for each of the positions. Based on the job opening, she has to decide and delegate the interview steps to different people.

    考虑招聘的场景。一个招聘人员不可能去面试每个职位的应聘者，一般是由招聘职位所属部门的人来负责面试。所以，可根据招聘职位不同，指派不同的人去负责面试应聘者。

工厂方法模式 – Motivation【动机】

- 回到前面配置文件解析的例子
  - 这里显示的是简单工厂的设计方案
  - 大量的 if… else if … 代码会导致维护和测试难度增大，如果要把简单工厂中的if … else if … 去掉，可以用多态处理



工厂方法模式 – Motivation【动机】

```
public class RuleConfigParserFactory {

  public static IRuleConfigParser createParser(String configFormat) {
        IRuleConfigParser parser = null;
        if ("json".equalsIgnoreCase(configFormat)) {
            parser = new JsonRuleConfigParser();
        } else if ("xml".equalsIgnoreCase(configFormat)) {
            parser = new XmlRuleConfigParser();
        } else if ("yaml".equalsIgnoreCase(configFormat)) {
            parser = new YamlRuleConfigParser();
        } else if ("properties".equalsIgnoreCase(configFormat)) {
            parser = new PropertiesRuleConfigParser();
        }
        return parser;
  }
}
```

工厂方法模式 – 重构后

- 将简单工厂模式中的 if … else if … 语句进行重构后，得到的设计便是工厂方法模式
  - 这样当我们新增一种 parser 的时候，只需要新增一个实现了 IRuleConfigParserFactory 接口的 Factory 类即可

工厂方法模式 – 重构后

- 其实，if … else if … 语句是转移到RuleConfigSource类中的load()方法中去了。

```
public interface IRuleConfigParserFactory {
    IRuleConfigParser createParser();
}
```

```
public class JsonRuleConfigParserFactory implements
   IRuleConfigParserFactory {
    @Override
    public IRuleConfigParser createParser() {
        return new JsonRuleConfigParser();
    }
}
```

工厂方法模式 – 重构后

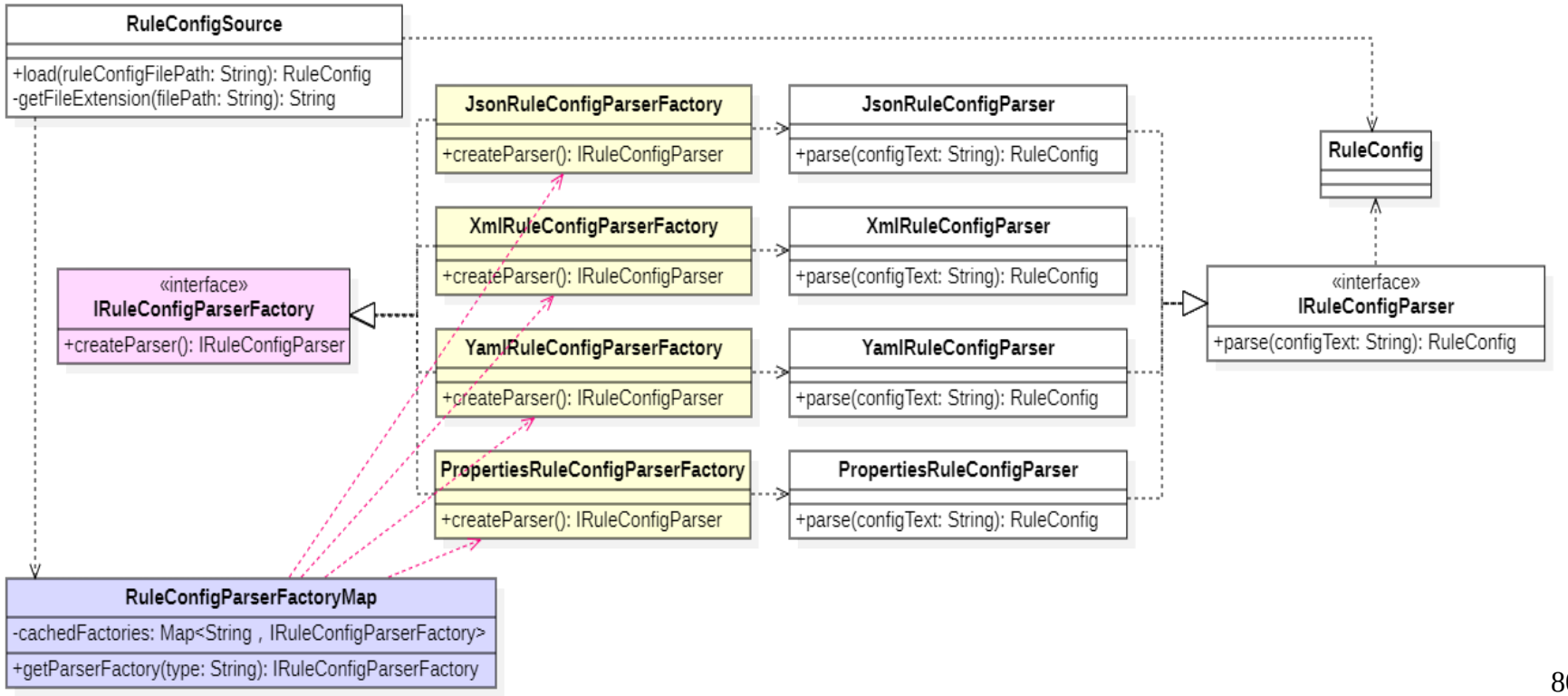
```
public class RuleConfigSource {
    public RuleConfig load(String ruleConfigFilePath) {
        String ruleConfigFileExtension =
                    getFileExtension(ruleConfigFilePath);

        IRuleConfigParserFactory parserFactory = null;
        if ("json".equalsIgnoreCase(ruleConfigFileExtension)) {
          parserFactory = new JsonRuleConfigParserFactory();
        } else if ("xml".equalsIgnoreCase(ruleConfigFileExtension)) {
          parserFactory = new XmlRuleConfigParserFactory();
        } else if … …
```

- 如果需要，也可以进一步重构...见下页

工厂方法模式 – 进一步重构后

- 可以为工厂类再创建一个简单工厂，也就是工厂的工厂，用来创建工厂类对象
  - RuleConfigParserFactoryMap 类是创建工厂对象的工厂类，getParserFactory() 返回的是缓存好的单例工厂对象



工厂方法模式 – 进一步重构后

```
//因为工厂类只包含方法，不包含成员变量，完全可以复用，不需要每次都创建新的工厂类对象
public class RuleConfigParserFactoryMap { //工厂的工厂
  private static final Map<String, IRuleConfigParserFactory> cachedFactories =
                       new HashMap<>();

  static {
    cachedFactories.put("json", new JsonRuleConfigParserFactory());
    cachedFactories.put("xml", new XmlRuleConfigParserFactory());
    cachedFactories.put("yaml", new YamlRuleConfigParserFactory());
    cachedFactories.put("properties", new PropertiesRuleConfigParserFactory());
  }

  public static IRuleConfigParserFactory getParserFactory(String type) {
    IRuleConfigParserFactory parserFactory =
                                    cachedFactories.get(type.toLowerCase());
    return parserFactory;
  }
}
```

工厂方法模式 – 进一步重构后

```
public class RuleConfigSource {
  public RuleConfig load(String ruleConfigFilePath) {
    String ruleConfigFileExtension = getFileExtension(ruleConfigFilePath);

    IRuleConfigParserFactory parserFactory =
        RuleConfigParserFactoryMap.getParserFactory(ruleConfigFileExtension);

    IRuleConfigParser parser = parserFactory.createParser();

    ... ...
}
```

工厂方法模式 – Intent【意图】

- Define an interface for creating an object, but let subclasses decide which class to instantiate.

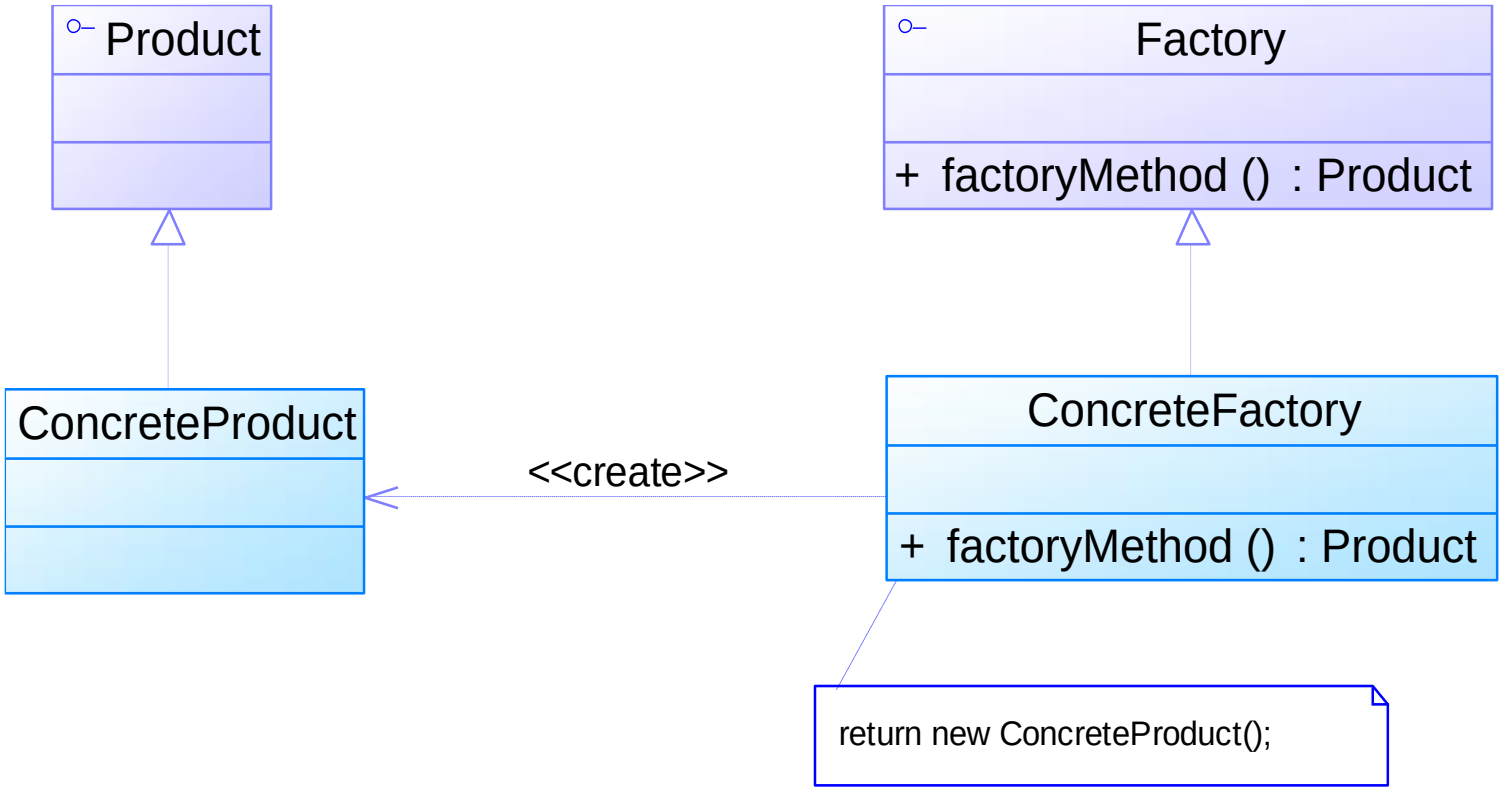
  定义一个用于创建对象的接口，让子类决定实例化哪一个类

- Factory Method lets a class defer instantiation to subclasses.

  Factory Method使一个类的实例化延迟到其子类

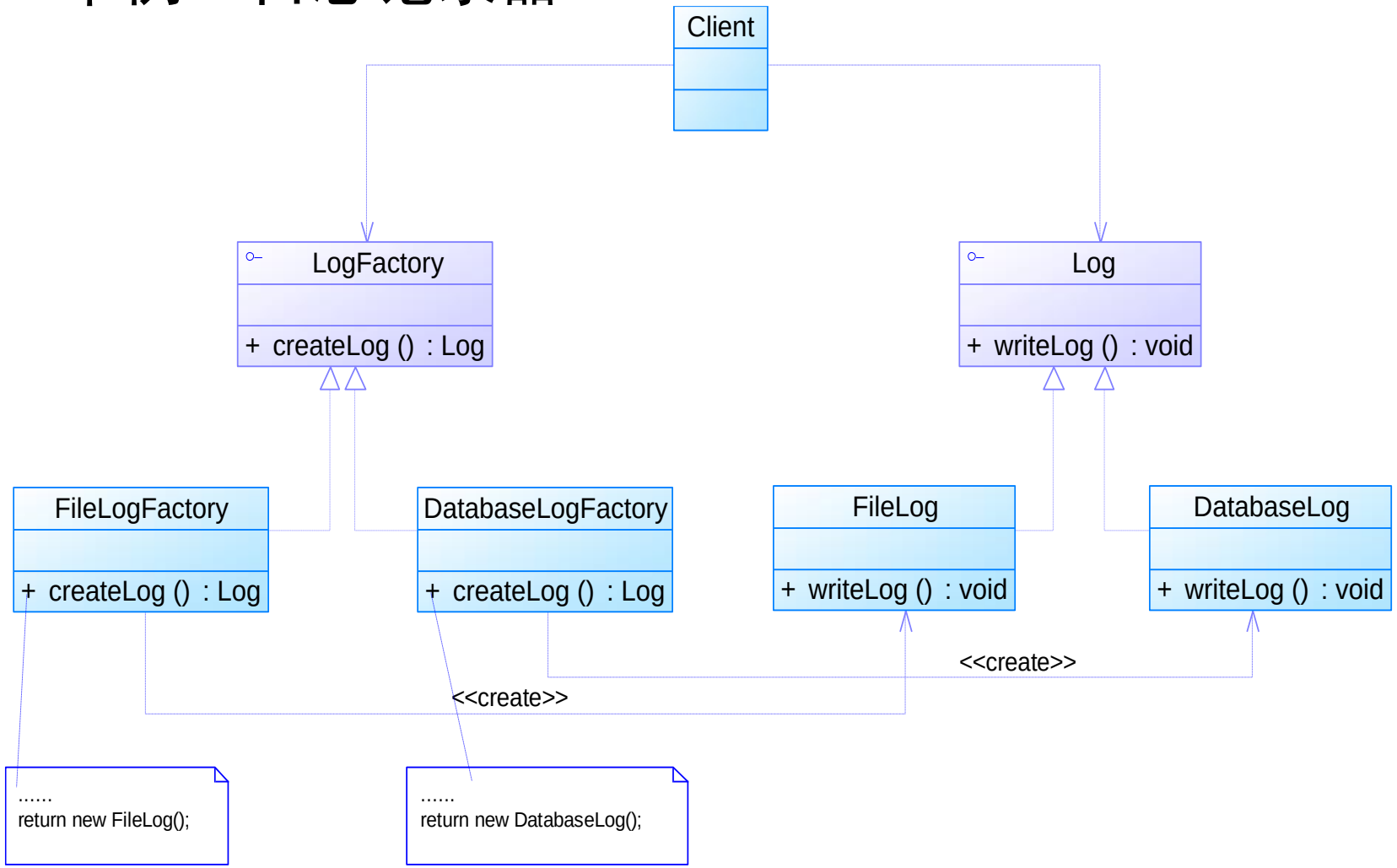
- 在工厂方法模式中，工厂父类/接口负责定义创建产品对象的公共接口，而工厂子类则负责生成具体的产品对象，这样做的目的是将产品类的实例化操作延迟到工厂子类中完成，即通过工厂子类来确定究竟应该实例化哪一个具体产品类

工厂方法模式 – Structure【结构】



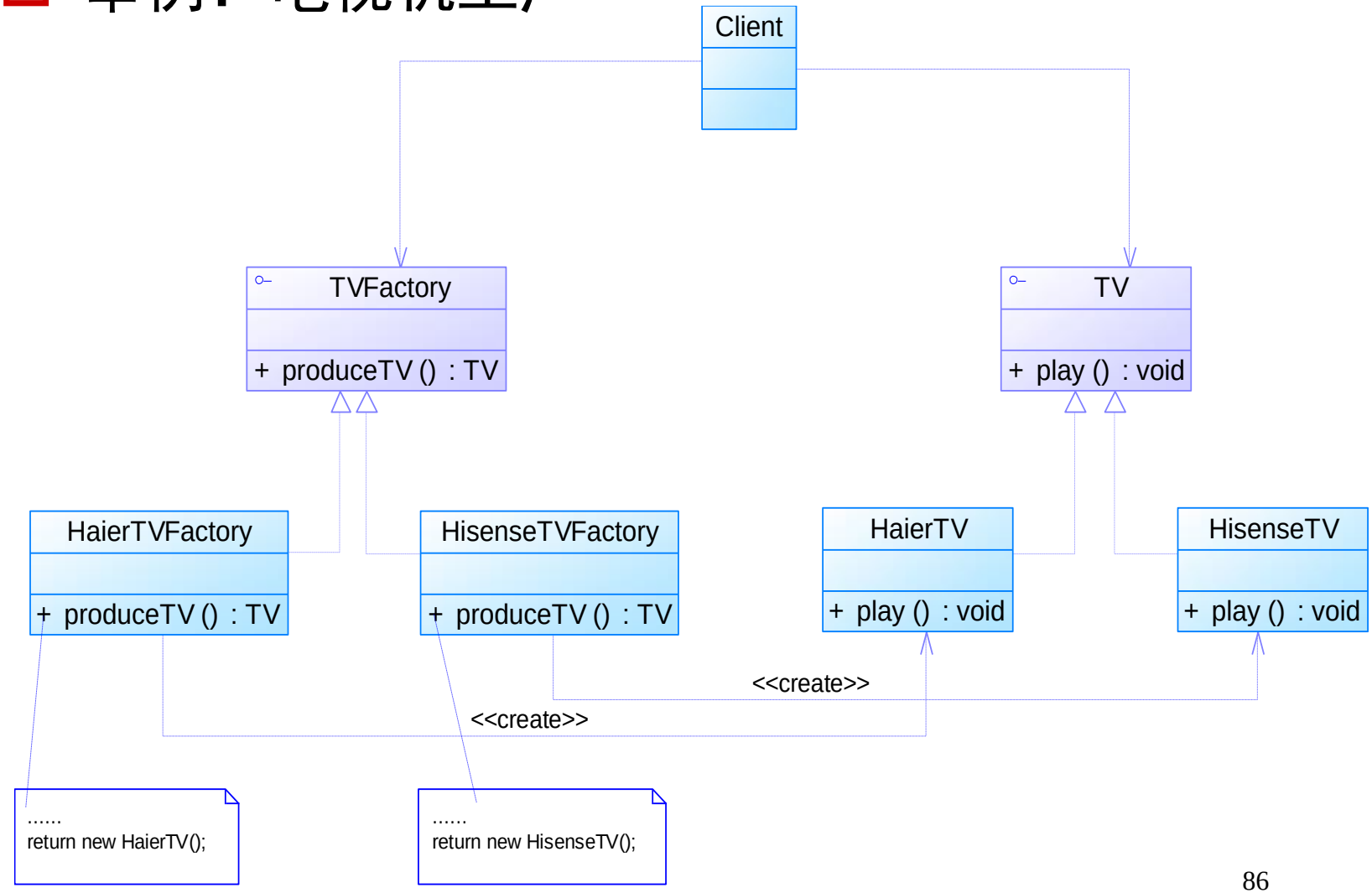
工厂方法模式 – 举例一

- 举例：日志记录器



工厂方法模式 – 举例二

- 举例：电视机工厂



工厂方法模式 – 优缺点

- 工厂方法模式的优点
  - 在工厂方法模式中，工厂方法用来创建客户所需要的产品，同时还向客户隐藏了哪种具体产品类将被实例化这一细节，用户只需要关心所需产品对应的工厂，无须关心创建细节，甚至无须知道具体产品类的类名
  - 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确定创建何种产品对象，而如何创建这个对象的细节则完全封装在具体工厂内部
  - 在系统中加入新产品时，无须修改抽象工厂和抽象产品提供的接口，无须修改客户端，也无须修改其他的具体工厂和具体产品，而只要添加一个具体工厂和具体产品就可以了。这样，系统的可扩展性也就变得非常好，完全符合“开闭原则”

工厂方法模式 – 优缺点

- 工厂方法模式的缺点
  - 在添加新产品时，需要编写新的具体产品类，而且还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，有更多的类需要编译和运行，会给系统带来一些额外的开销
  - 由于考虑到系统的可扩展性，需要引入抽象层，在客户代码中均使用抽象层进行定义，增加了系统的抽象性和理解难度

工厂方法模式 –【适用情形】

- A class can't anticipate the class of objects it must create.

  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工厂类创建；客户需要知道创建具体产品的工厂类

工厂方法模式 –【适用情形】

- A class wants its subclasses to specify the objects it creates.

  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，而由其子类来确定具体要创建的对象，利用面向对象的多态性和Liskov替换原则，在程序运行时，子类对象将覆盖父类对象，从而使得系统更容易扩展

工厂方法模式 – JDK中的应用

- Java中的例子：
  - 集合框架（java.util.Collection接口）
    - Collection接口中的iterator()方法

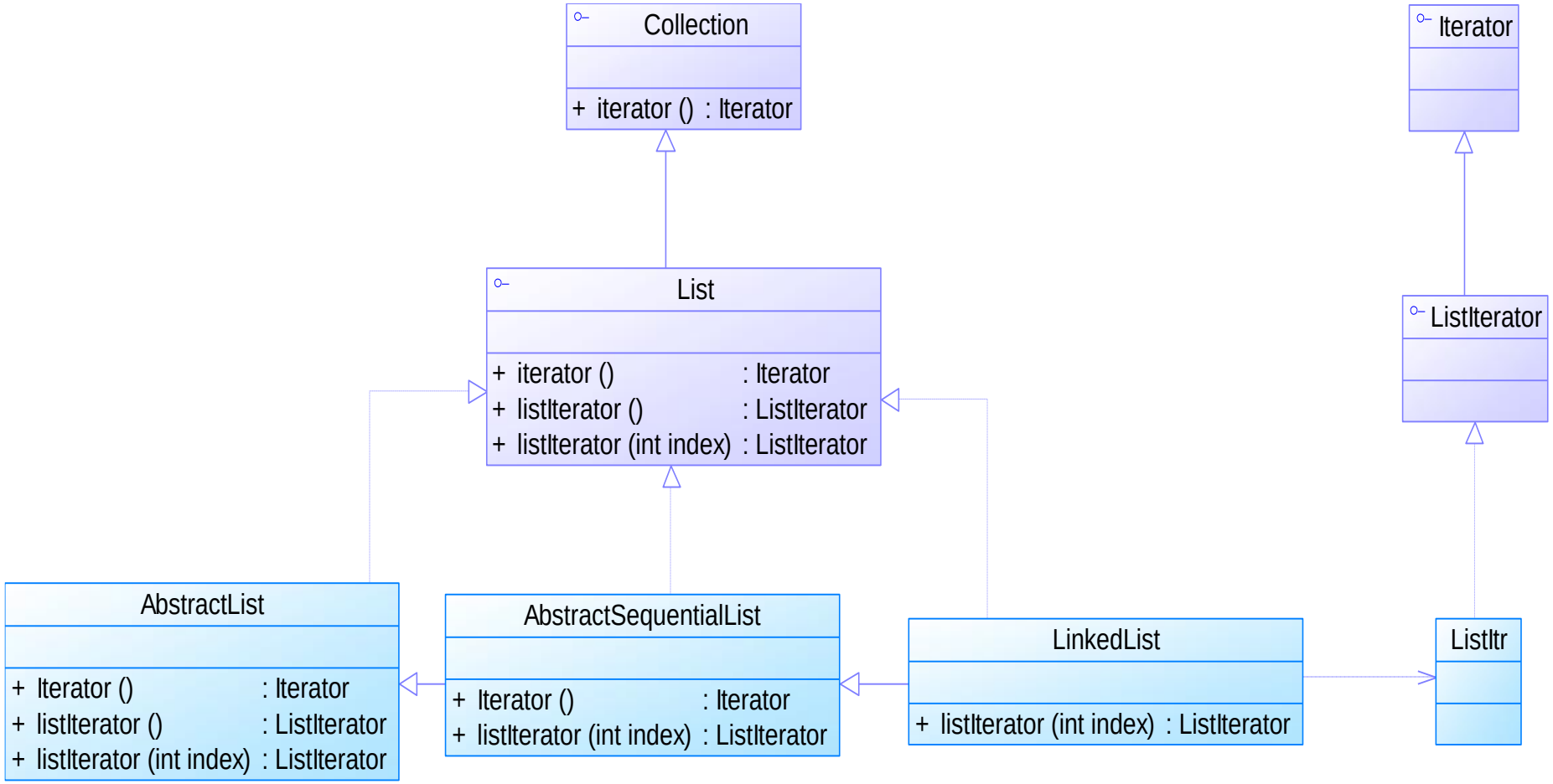
Iterator<E> iterator()

返回在此 collection 的元素上进行迭代的迭代器。关于元素返回的顺序没有任何保证（除非此 collection 是某个能提供保证顺序的类实例）。

      - 一个具体的Java集合对象会通过这个iterator()方法返回一个具体的Iterator对象，这个iterator()就是工厂方法

    - List接口中的iterator()方法、listIterator()方法
      - Iterator()、listIterator()是List的两个工厂方法

工厂方法模式 – JDK中的应用



抽象工厂模式
Abstract Factory

现实世界的例子

- 抽象工厂模式—现实世界的例子
  - Extending our door example from Simple Factory. Based on your needs you might get a wooden door from a wooden door shop, iron door from an iron shop or a PVC door from the relevant shop. Plus you might need a guy with different kind of specialities to fit the door, for example a carpenter for wooden door, welder for iron door etc. As you can see there is a dependency between the doors now, wooden door needs carpenter, iron door needs a welder etc.

    从 Simple Factory 扩展前面的制作门的例子。根据您的需求，您可以从木门店获得木门，铁门店获得铁门或PVC门店获得PVC门。另外，你可能还需要一个有不同技能的师傅来安装门，例如木门的木匠，铁门的焊接工等。你可以看到存在的依赖关系，木门需要木匠，铁门需要焊工等

抽象工厂模式 – Intent【意图】

- Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
  提供一个创建一系列相关或相互依赖对象的接口，而无须指定它们具体的类

  - 注：相对于简单工厂模式和工厂方法模式，抽象工厂模式用得少些

抽象工厂模式 – Motivation【动机】

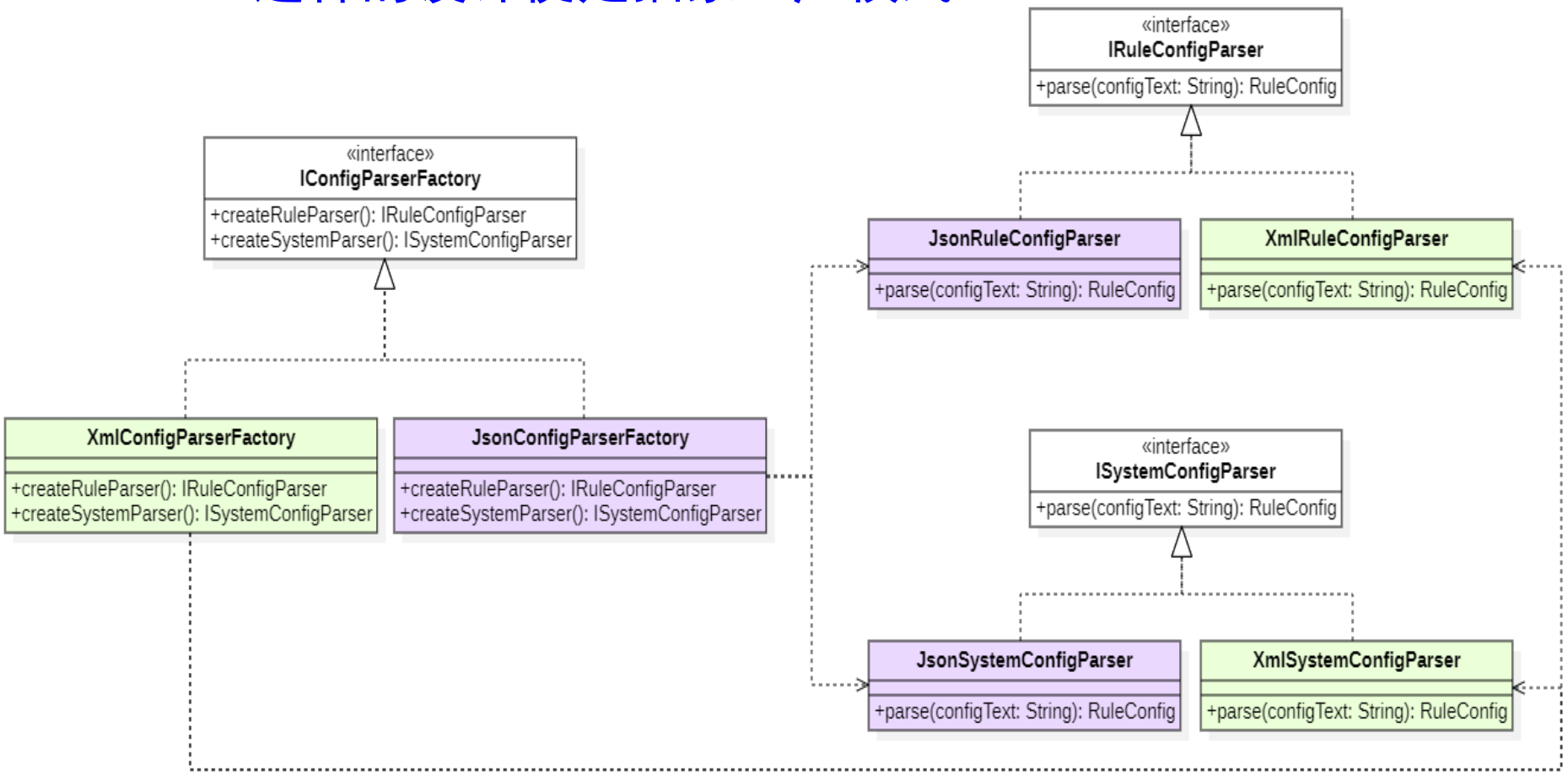
- 例如：在配置文件解析的例子中，之前是按照文件的格式分类为 json、xml、yaml、properties 4个解析器产品。现在如果解析的对象除了规则配置文件类别外，还有系统配置文件类别，那么就会有8个解析器产品如下：
  - 针对规则配置的解析器：基于接口IRuleConfigParser
    - JsonRuleConfigParser，XmlRuleConfigParser
    - YamlRuleConfigParser，PropertiesRuleConfigParser
  - 针对系统配置的解析器：基于接口ISystemConfigParser
    - JsonSystemConfigParser
    - XmlSystemConfigParser
    - YamlSystemConfigParser
    - PropertiesSystemConfigParser

如果采用工厂方法模式，那么针对每个产品均需定义一个工厂类，就会有8个工厂类。如果将来还有其他分类，则工厂类将成倍增加，不好维护

如何解决？

抽象工厂模式 – Motivation【动机】

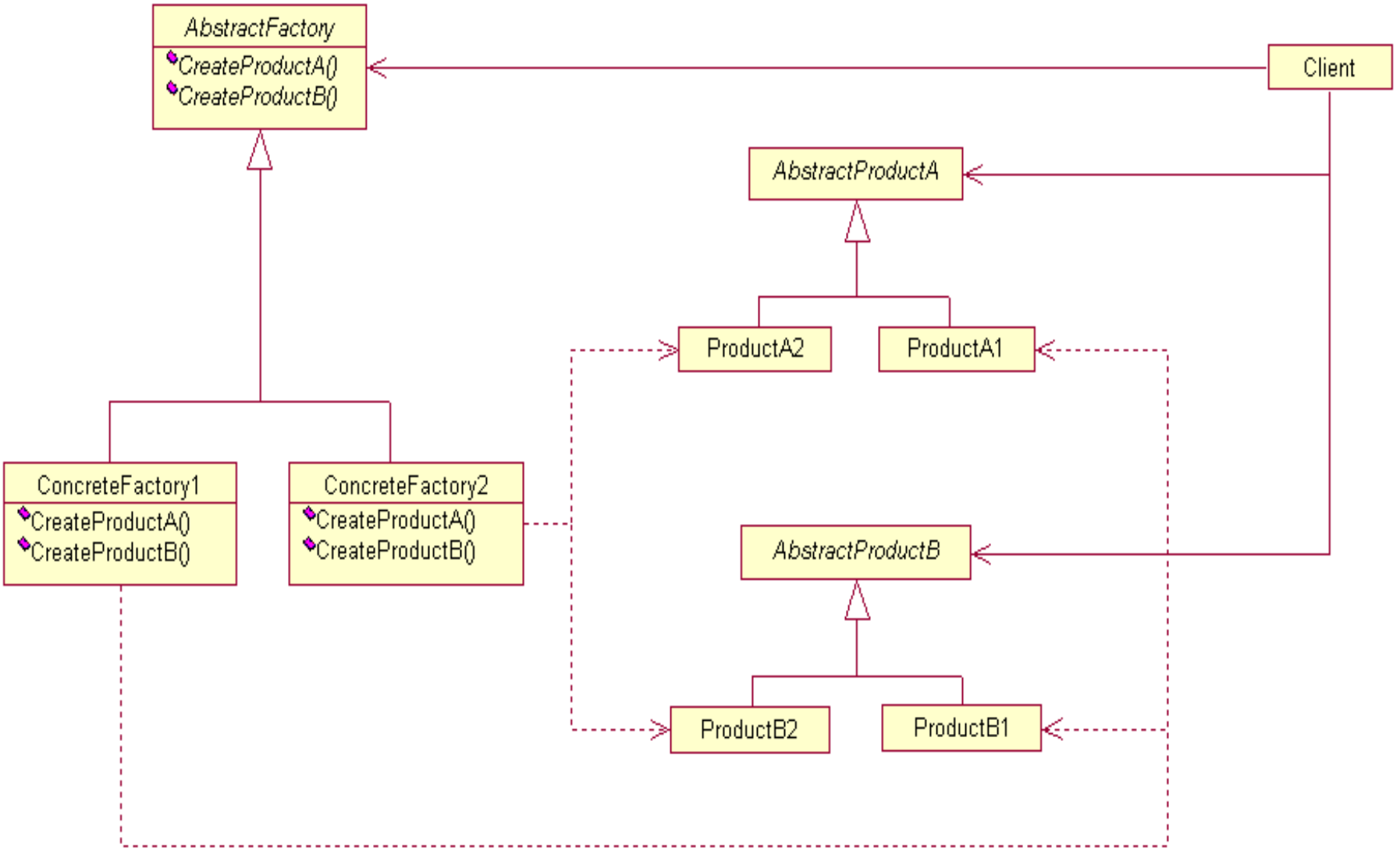
- 可不可以让一个工厂生产多个产品呢？
  - 这样的设计便是抽象工厂模式



抽象工厂模式 – Motivation【动机】

- 抽象工厂模式与工厂方法模式的区别
  - 工厂方法模式针对的是一个产品等级结构，而抽象工厂模式则需要面对多个产品等级结构
  - 当一个工厂等级结构可以创建出分属于不同产品等级结构的一个产品族中的所有对象时，抽象工厂模式比工厂方法模式更为简单、有效率

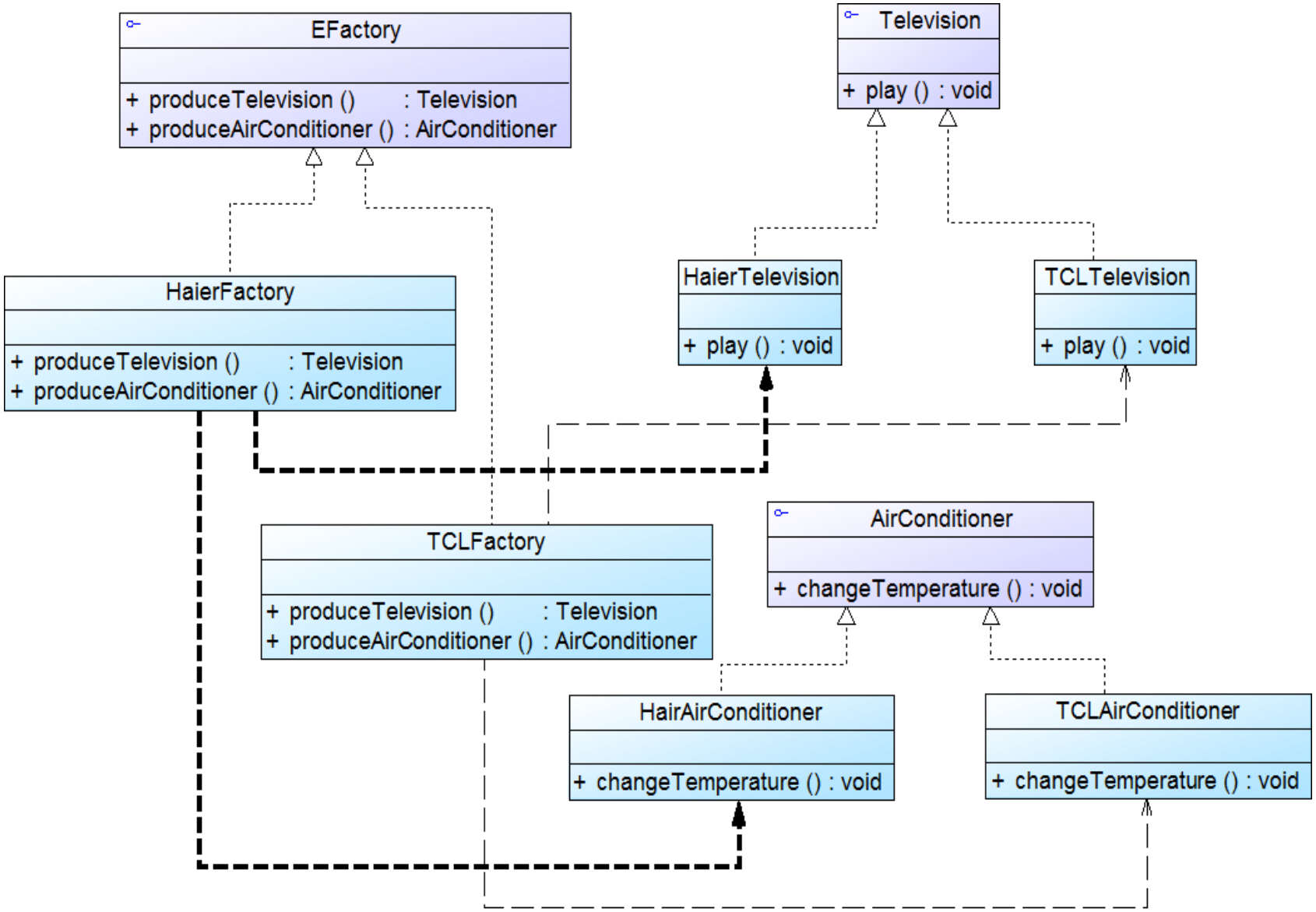
抽象工厂模式 – Structure【结构】



抽象工厂模式 – 举例

- 举例：电器工厂
  - 一个电器工厂可以产生多种类型的电器，如海尔工厂可以生产海尔电视机、海尔空调等，TCL工厂可以生产TCL电视机、TCL空调等
  - 相同品牌的电器构成一个产品族，而相同类型的电器构成了一个产品等级结构

抽象工厂模式 – 举例


EFactory
+ produceTelevision () : Television
+ produceAirConditioner () : AirConditioner
Television
+ play () : void
HaierFactory
+ produceTelevision () : Television
+ produceAirConditioner () : AirConditioner
HaierTelevision
+ play () : void
TCLTelevision
+ play () : void
TCLFactory
+ produceTelevision () : Television
+ produceAirConditioner () : AirConditioner
AirConditioner
+ changeTemperature () : void
HairAirConditioner
+ changeTemperature () : void
TCLAirConditioner
+ changeTemperature () : void

抽象工厂模式 – 优缺点

- 抽象工厂模式的优点
  - 抽象工厂模式隔离了具体类的生成，使得客户并不需要知道什么被创建
  - 更换一个具体工厂变得相对容易。所有的具体工厂都实现了抽象工厂中定义的那些公共接口，因此只需改变具体工厂的实例，就可以在某种程度上改变软件系统的行为
  - 当一个产品族中的多个对象被设计成一起工作时，它能够保证客户始终只使用同一个产品族中的对象
  - 增加新的具体工厂和产品族很方便，无须修改已有系统，符合“开闭原则”

抽象工厂模式 – 优缺点

- 抽象工厂模式的缺点
  - 在添加新的产品对象时，难以扩展抽象工厂来生产新种类的产品
    - 这是因为在抽象工厂角色中规定了所有可能被创建的产品集合，要支持新种类的产品就意味着要对该接口进行扩展，而这将涉及到对抽象工厂角色及其所有子类的修改，显然会带来较大的不便

抽象工厂模式 – 【适用情形】

- 在以下情况下可以使用抽象工厂模式
  - 一个系统不应当依赖于产品类实例如何被创建、组合和表达的细节，这对于所有类型的工厂模式都是重要的
  - 系统中有多于一个的产品族，而每次只使用其中某一产品族
  - 属于同一个产品族的产品将在一起使用，这一约束必须在系统的设计中体现出来
  - 系统提供一个产品类的库，所有的产品以同样的接口出现，从而使客户不依赖于具体实现

Thank you!