私塾在线《设计模式综合项目实战》

——跟着CC学设计系列精品教程

本阶段课程概览

- 设计并实现配置管理模块

一：配置管理模块的详细功能

二：配置管理模块的功能边界

三：配置管理模块对外的接口

包括：对外的数据接口、对外的程序接口

四：配置管理模块的内部实现

包括：简单工厂模式、单例模式、桥接模式、解释器模式、组合模式、备忘录模式、原型模式、生成器模式、策略模式 的综合应用

配置管理模块的详细功能

- 获取用户配置的数据

  配置的方式很多，要求除了框架自身提供的配置方式外，还要能支持用户自定义的配置方式。比如：框架本身提供默认的配置方式为xml配置，如果用户想使用数据库来配置，那么框架必须能够支持用户自定义一种数据库配置的方式 ，并能够很容易的加入到系统中进行使用。

- 缓存用户配置的数据

  同一份配置数据，在运行期间会多次使用，但是获取用户配置数据的动作就只需要一次就可以了，获取过后，就缓存下来，重复使用了。

- 对外提供接口

  让其他模块通过这些接口来获取他们需要的数据

配置管理模块的功能边界

- 配置管理模块的功能边界

1：配置管理模块不关心被访问的数据是怎么来的，它只是按照访问方式去获取这些数据，当然不同的数据格式有不同的访问方式。

2：虽然需要支持自定义配置方式，但是需要配置的数据是一定的，只是配置的格式不同，访问这些配置数据的方式不同，但最后殊途同归，都想配置管理模块提供相应的数据。因此这个需要配置的数据项是要统一起来的。

3：配置管理模块不关心获取到的数据是什么，也不关心这些数据的含义，更不关心这些数据怎么用，它只是负责获取这些数据并保存起来。

4：配置管理模块不关心谁来使用这些数据，也不关心外部获取这些数据后如何使用，它只是负责提供这些数据而已。

配置管理模块的对外接口

- 对外数据接口设计

  目前设计的配置分成三类，虽然配置的方式不固定，但是对于核心框架而言，有一些数据是必需要配置的，所以还是需要设计对外的数据接口。

  为了简单，这里以xml为例子来说明定义的数据结构。但是请注意，x-gen框架并不局限于使用xml作为配置的方式，同样你可以选用properties文件、数据库等其他的配置方式。

  虽然配置方式不同，需要配置的数据却是一样的，所以，你可以把这里xml描述的配置数据改成任何你希望使用的配置方式相对应的数据结构。

- 第一类是核心框架运行需要的数据

  以xml配置为例，默认取名为GenConf.xml，在每次使用的时候根据需要修改或配置其内容。当然，为了示例简单，就不去做dtd或者schema了。

配置管理模块的对外接口

```
<?xml version="1.0" encoding="UTF-8"?>
<GenConf>
    <NeedGens>
        <NeedGen id="UserGenConf" provider="XmlModuleGenConf" themeId="simple">
            <Params>
                <Param id="fileName">UserGenConf.xml</Param>
            </Params>
        </NeedGen>
    </NeedGens>
    <Themes>
        <Theme id="simple">cn/javass/themes/simple</Theme>
    </Themes>
  <Constants>
        <Constant id="prePackage" value="cn.javass"></Constant>
        <Constant id="projectName" value="framework"></Constant>
    </Constants>
</GenConf>
```

配置管理模块的对外接口

n 节点说明如下：

（1）<GenConf>是根节点

（2）<NeedGens> ：用来注册需要generate的模块的配置，里面可以注册很多个需要generate的模块

（3）<NeedGen> ：需要generate的一个模块，属性含义如下：

Id：需要generate的模块的标识名称或编号，必须唯一

provider：用来获取该模块配置数据的程序的标识，该标识对应的实现类定义，在该模块使用的theme里面配置

themeId：该模块生成的时候具体使用的theme的标识

（4）<Params>：参数，通常是根据不同的provider，提供不同的参数配置，每个参数配置都包括参数标识和具体的参数值

（5）<Param>：具体一个参数的配置，值就是参数的值，属性如下：

Id：参数的标识名称或编号，必须唯一

（6）<Themes>：用来注册generate所需要的外部theme，可以配置多个<Theme>

（7）<Theme>：具体的描述一个theme，theme存放的具体位置，可以配置一个相对路径，配置到theme存放的根文件夹即可，属性如下：

Id：theme的标识名称或编号，必须唯一

（8）<Constants>：常量定义，可以配置多个常量

（9）<Constant>：单个常量定义，值就是常量的值，属性为：

id ：常量的标识名称或编号，必须唯一

配置管理模块的对外接口

- 第二类是用户需要生成的模块的配置数据

比如：用户想要生成一个模块内的增删改查的源代码，里面有每个具体功能的配置，而每个具体功能就是一个源代码文件。

这个也需要在每次使用的时候根据需要来配置，并注册到核心框架运行配置里面去，每次生成主要配置的就是这类数据。示例如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<ModuleGenConf id="UserConf">
    <NeedGenTypes>
        <NeedGenType id="GenBusinessEbi">
            <NeedGenOutType id="File"></NeedGenOutType>
        </NeedGenType>
    </NeedGenTypes>
    <ExtendConfs>
        <ExtendConf id="moduleName" isSingle="true">user</ExtendConf>
    </ExtendConfs>
</ModuleGenConf>
```

配置管理模块的对外接口

n 节点说明如下：

（1）<ModuleGenConf>：每个模块配置的根节点，属性如下：
id：模块的标识，不重复即可

（2）<NeedGenTypes>：用来配置需要生成的功能，可以配置多个

（3）<NeedGenType>：配置一个本模块需要生成的功能，属性如下：
id：需要生成的功能的标识，这个标识在theme的配置中定义

（4）<NeedGenOutType>：配置模块的某个功能生成完成后，输出的类型，一个功能可以有多种输出。属性如下：
id：具体输出的标识，这个标识在theme的配置中定义

（5）<ExtendsConfs>：本模块需要配置的，自行扩展的数据，可以配置多个数据。

（6）<ExtendConf>：描述一条自行扩展的数据，属性如下：
id：自定义数据的标识
isSingle：数据是单个值还是一个数组，true表示是单个值，false表示是数组，如果是数组的话，多个值之间用逗号分开

配置管理模块的对外接口

- 第三类就是外部主题的配置数据

在制作主题的时候就配置好，里面有这套主题需要的外部数据，和预定义好的功能配置，在每次使用的时候一般不需要配置或修改。比如：缺省为xml配置，取名为ThemeConf.xml。

外部主题目前不支持自定义，也就是采用默认的xml方式，而且文件名称也固定是ThemeConf.xml，其实要实现支持自定义也很简单，只是没有必要，那样会无谓的增加复杂度。

示例如下：

配置管理模块的对外接口

```
<?xml version="1.0" encoding="UTF-8"?>
<Theme id="simple">
  <GenTypes>
    <GenType id="GenBusinessEbi" type="cn.javass.themes.simple.actions.GenBusinessEbiAction">
      <Params>
        <Param id="relativePath">business.ebi</Param>
        <Param id="mbPathFile">business/Ebi.txt</Param>
        <Param id="preGenFileName"></Param>
        <Param id="afterGenFileName">Ebi.java</Param>
      </Params>
    </GenType>
  </GenTypes>
  <GenOutTypes>
    <GenOutType id="Console" type="cn.javass.xgen.output.types.OutputToConsole"></GenOutType>
    <GenOutType id="File" type="cn.javass.xgen.output.types.OutputToFile"></GenOutType>
  </GenOutTypes>
  <Providers>
    <Provider id="XmlModuleGenConf"
    type="cn.javass.xgen.genconf.implementors.xmlimpl.ModuleGenConfXmlImpl"></Provider>
  </Providers>
</Theme>
```

配置管理模块的对外接口

（1）<Theme>：主题配置的根节点，属性如下：
id：主题的标识，必须唯一，在GenConf注册theme的id就是这个值
（2）<GenTypes>：主题提供的可输出的功能，可配置多个
（3）<GenType>：描述一个主题可输出的具体功能，属性如下：
id：功能的标识，必须唯一，在ModuleGenConf里面配置每个模块的NeedGenType的id就是这个值
type：真正实现输出功能的类
（4）Params：配置每个输出类需要的参数，下面可以配置多个param
（5）param：具体的每个参数的配置，值就是参数的值，属性为：
id：参数的标识，在一个params必须唯一
（6）GenOutTypes：主题提供的输出类型，可以配置多个
（7）GenOutType：一个具体的输出类型，属性如下：
id：功能的标识，必须唯一，在ModuleGenConf里面配置每个模块的NeedGenOutType的id就是这个值
type：真正实现输出类型的类
（6）Providers：主题提供的读取配置文件的类型，可以配置多个
（9）Provider：一个具体的读取配置文件的类型，属性如下：
id：功能的标识，必须唯一，在GenConf里面配置NeedGen的provider就是这个值
type：真正实现具体的读取配置文件的类

配置管理模块的对外接口

- 对外程序接口设计

  为了让外部获取配置管理模块内的数据，提供相应的API（应用程序接口），也就是GenConfEbi

配置管理模块的内部实现

- 实现的起点

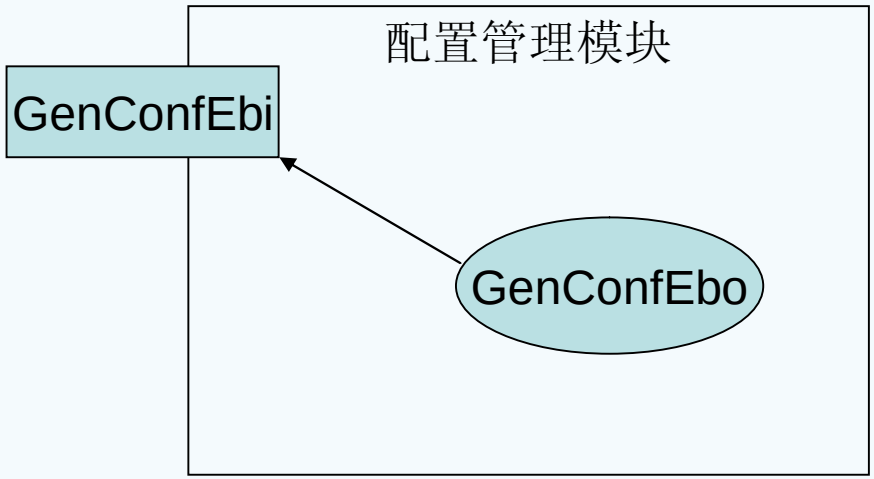
为了让大家更好的理解配置管理模块的内部实现架构，因此先以一个最简单的实现结构为起点，采用重构的方式，逐步把相关的设计模式应用进来，从简单到复杂，从而让大家更好的看到如何选择要使用的设计模式、如何实际应用设计模式以及如何让多种设计模式协同工作。

1：先就来看看实现配置管理的起点，首先根据对外提供的数据结构定义，制作出相应的数据model来。

2：针对前面定义的API，提供一个最基本的实现，只需要满足最基本的功能就可以了，需要实现读取配置文件的功能，然后要有缓存配置数据的功能，最后就是实现API中要求的功能。

配置管理模块的内部实现

- 此时配置管理模块的结构示意如图



加入简单工厂

- 面临的问题

  观察上面的实现，向模块外部提供了接口，可是外部根本不知道模块内部的具体实现，那么模块外部如何来获取一个实现接口的实现对象呢？

- 用简单工厂来解决

  简单工厂是解决上述问题的一个合理方案。那么先一起来回顾一下简单工厂的一些基础知识，然后再来看如何应用它来解决上面的问题。

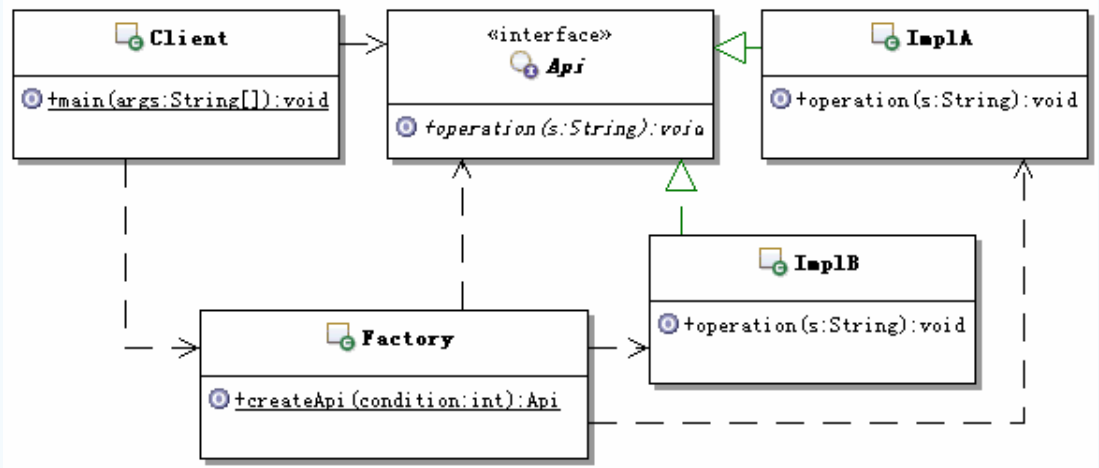
- 简单工厂基础回顾

初识简单工厂

- 定义

  提供一个创建对象实例的功能，而无须关心其具体实现。被创建实例的类型可以是接口、抽象类，也可以是具体的类。

- 结构和说明



Api：定义客户所需要的功能接口
Impl：具体实现Api的实现类，可能会有多个
Factory：工厂，选择合适的实现类来创建Api接口对象
Client：客户端，通过Factory去获取Api接口对象，然后面向Api接口编程

简单工厂的知识要点

- 简单工厂的知识要点

1：简单工厂位于对外提供接口的模块内

2：简单工作的主要功能就是用来创建对象实例，被创建的对象可以是接口、抽象类或是普通的类

3：简单工厂可以实现成为单例，也可以实现成静态工厂

4：简单工厂的内部实现，主要是做“选择合适的实现”，实现是已经做好的，简单工厂只是来选择使用即可

5：简单工厂在进行选择的时候，需要的参数可以从客户端传入、配置文件、或者是运行期程序某个运行结果等

6：如果使用反射+配置文件的方式，可以写出通用的简单工厂

思考简单工厂

- 简单工厂的本质

  简单工厂的本质是：**选择实现**

- 何时选用简单工厂

1：如果想要完全封装隔离具体实现，让外部只能通过接口来操作封装体，那么可以选用简单工厂，让客户端通过工厂来获取相应的接口，而无需关心具体实现

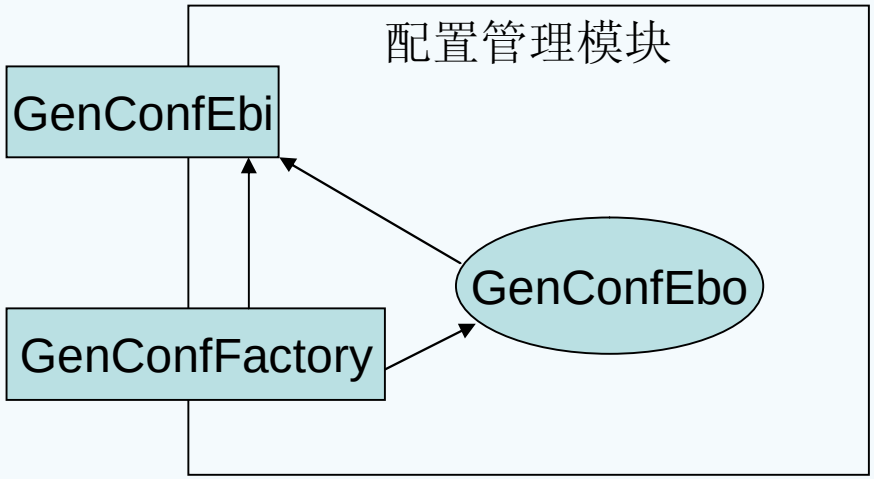
2：如果想要把对外创建对象的职责集中管理和控制，可以选用简单工厂，一个简单工厂可以创建很多的、不相关的对象，可以把对外创建对象的职责集中到一个简单工厂来，从而实现集中管理和控制

应用简单工厂

- 使用简单工厂来解决问题的思路

  简单工厂解决这个问题的思路就是，在配置管理模块里面添加一个类，在这个类里面实现一个方法，让这个方法来创建一个接口对象并返回然后把这个类提供给客户端，让客户端通过调用这个类的方法来获取接口对象。

- 此时配置管理模块的结构示意如图



加入单例模式

- 面临的问题

  看看上面的基本实现，会发现一些问题：

1：如果GenConfEbo被创建多次的话，那么就会重复获取配置数据，浪费程序运行时间；

2：并且每个GenConfEbo的实例都会缓存这些数据，浪费内存空间。

3：同一个类里面，既有实现GenConfEbi要求的对外功能，又有内部实现需要的获取配置数据和缓存数据的功能，从类的设计上来说，这个类的职责太不单一了，应该分离一部分职责出去。

因此这种实现肯定是不好的，那么怎么解决呢？

- 用单例模式来解决

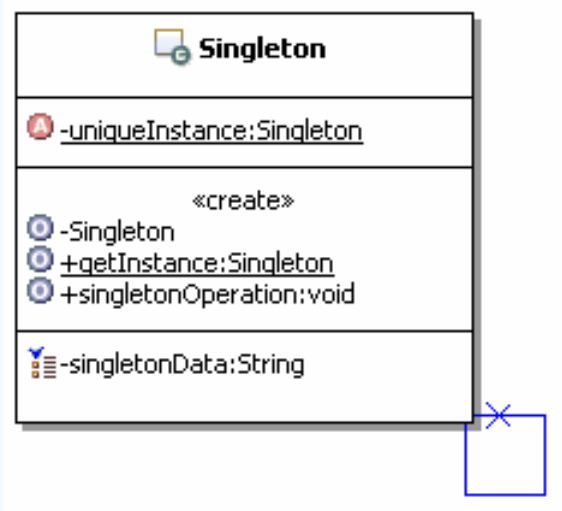
- 单例模式基础回顾

初识单例模式

- 定义

  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。

- 结构和说明



Singleton：

负责创建Singleton类自己的唯一实例，并提供一个getInstance的方法，让外部来访问这个类的唯一实例。

单例模式的知识要点

- 单例模式的知识要点

1：单例模式是用来保证一个类在运行期间只会被创建一个类实例的

2：单例模式还提供了一个全局唯一访问这个类实例的访问点，通常就是那个getInstance方法

3：单例模式只关心类实例的创建问题，并不关心具体的业务功能。

4：单例模式是一个虚拟机范围内单例，不适用于集群等环境

5：单例模式的实现有很多种，除了常见的懒汉式和饿汉式外，还有利用缓存来实现、双重检查加锁的实现、Lazy initialization holder class模式、以及枚举的实现方式等

6：使用单例模式的时候要注意它的线程安全性

7：单例模式体现了延迟加载、缓存等常见的设计思想

8：单例模式是可以很容易的扩展到多实例控制的

思考单例模式

- 单例模式的本质

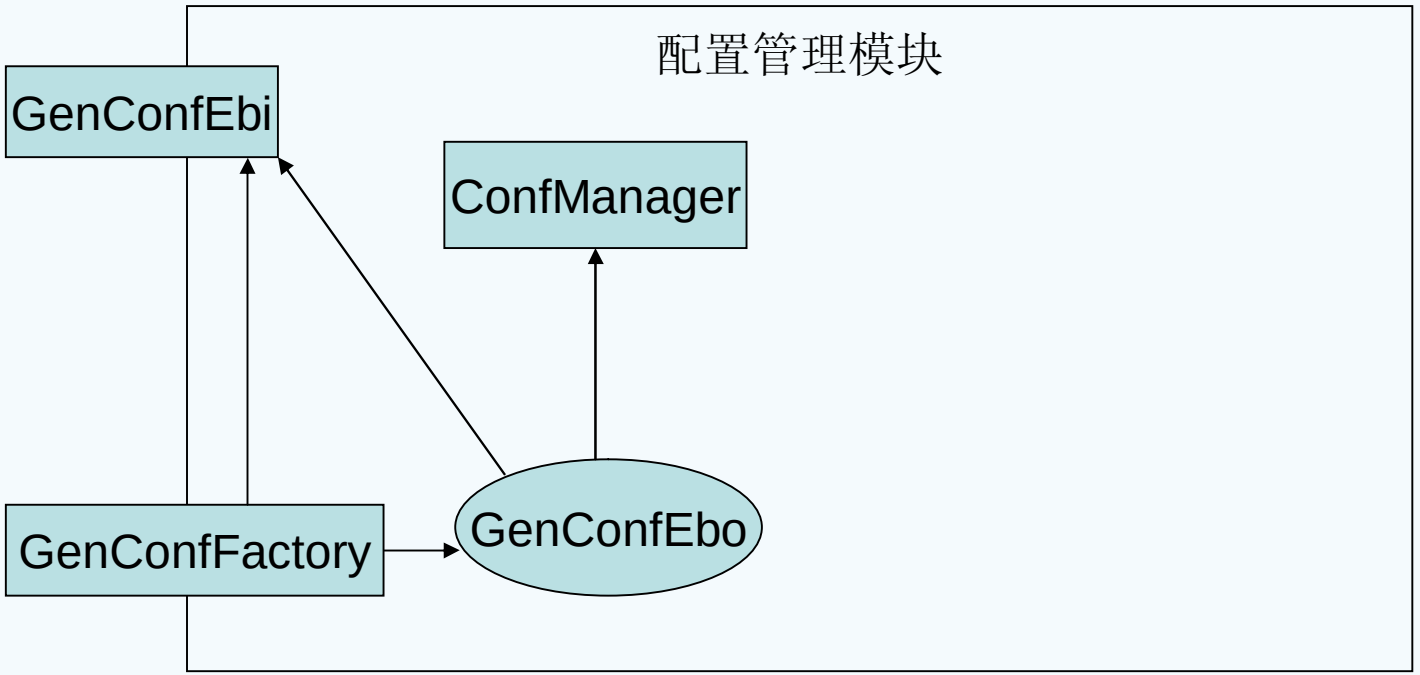
  单例模式的本质是：**控制实例数目**

- 何时选用单例模式

  当需要控制一个类的实例只能有一个，而且客户只能从一个全局访问点访问它时，可以选用单例模式，这些功能恰好是单例模式要解决的问题

应用单例模式

- 此时配置管理模块的结构示意如图



加入桥接模式

- 面临的问题

  按照功能要求，配置数据的来源是多方面，比如：xml、properties、txt、DB等等，这也就意味着需要有不同的获取数据的实现来对应这些不同的数据来源。

  另外一个方面，对于模块外部的应用而言，他们不关心配置数据是如何来的，他们只关心需要使用的数据，而且在某些需要的情况下，他们可能会要求更多的、不同的数据，他们会认为“只要他们要数据，配置管理模块就应该提供这些数据”。换句话说，这些数据的多少、数据的组合结构是可能发生变化的。

  这就出现了两个纬度的变化，一个是获取配置数据这边需要不断扩展，另一个是配置数据所构成的数据模型这边也需要变化和扩展，怎么办呢？

- 用桥接模式来解决

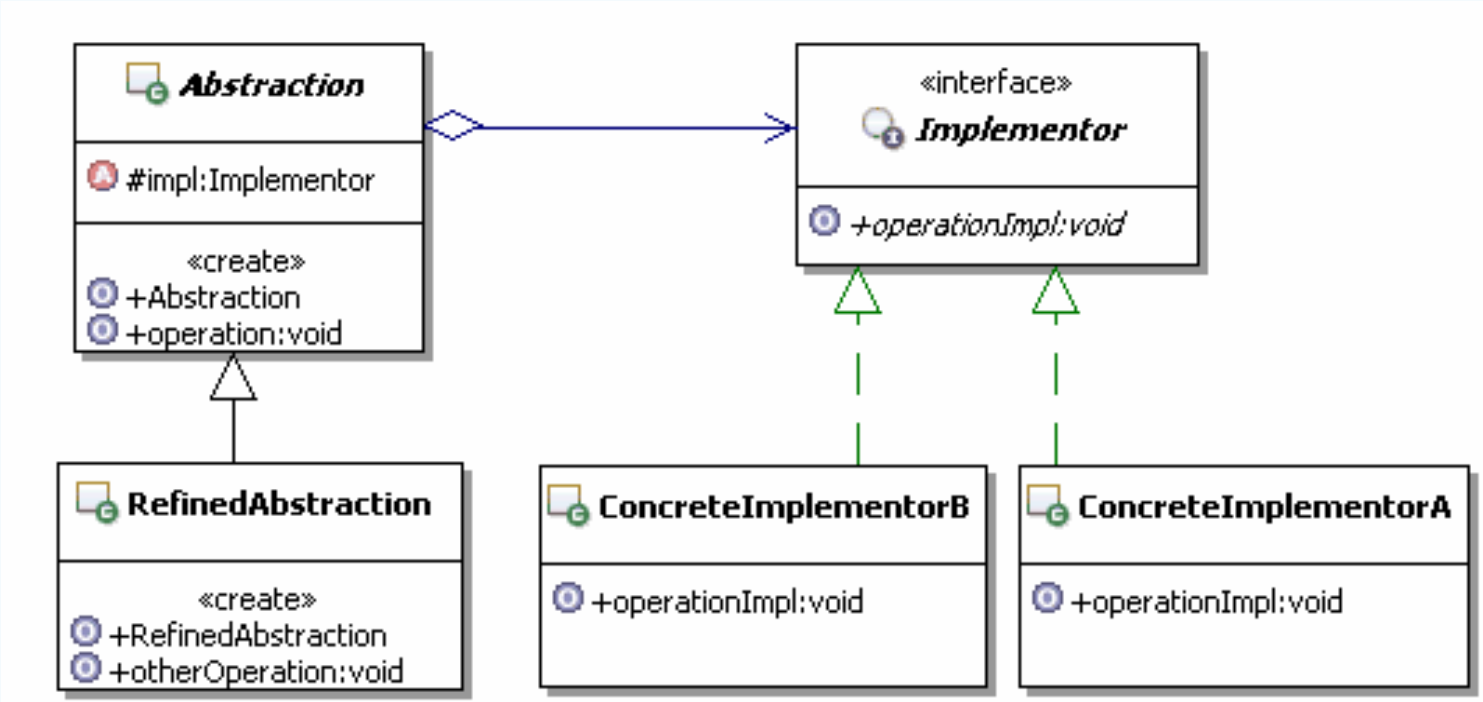
- 桥接模式基础回顾

初识桥接模式

- 定义

  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。

- 结构和说明



桥接模式的知识要点

- 桥接模式的知识要点

1：桥接是在被分离了的抽象部分和实现部分之间来搭桥，桥接在程序上就体现成了在抽象部分拥有实现部分的接口对象，维护桥接就是维护这个关系

2：桥接模式的意图：使得抽象和实现可以独立变化，都可以分别扩充。

3：桥接模式可以实现运行时动态组合具体的真实实现，从而达到动态变换功能的目的

4：桥接模式适应于两个纬度的变化，而继承适用于一个纬度的变化

5：使用桥接模式的时候，要注意谁来创建Implementor的对象，并把它设置到抽象部分的对象里面去。

6：从某个角度来讲，桥接模式就是对“面向抽象编程”这个设计原则的扩展。

7：桥接模式是可以连续组合使用的，一个桥接模式的实现部分，可以作为下一个桥接模式的抽象部分

思考桥接模式

- 桥接模式的本质是：**分离抽象和实现**
- 何时选用桥接模式

1：如果你不希望在抽象和实现部分采用固定的绑定关系，可以采用桥接模式，来把抽象和实现部分分开，然后在程序运行期间来动态的设置抽象部分需要用到的具体的实现，还可以动态切换具体的实现

2：如果出现抽象部分和实现部分都应该可以扩展的情况，可以采用桥接模式，让抽象部分和实现部分可以独立的变化，从而可以灵活的进行单独扩展，而不是搅在一起，扩展一边会影响到另一边 。

3：如果希望实现部分的修改，不会对客户产生影响，可以采用桥接模式，客户是面向抽象的接口在运行，实现部分的修改，可以独立于抽象部分，也就不会对客户产生影响了，也可以说对客户是透明的

4：如果采用继承的实现方案，会导致产生很多子类，对于这种情况，可以考虑采用桥接模式，分析功能变化的原因，看看是否能分离成不同的纬度，然后通过桥接模式来分离它们，从而减少子类的数目

应用桥接模式

- 使用桥接模式来解决问题的思路

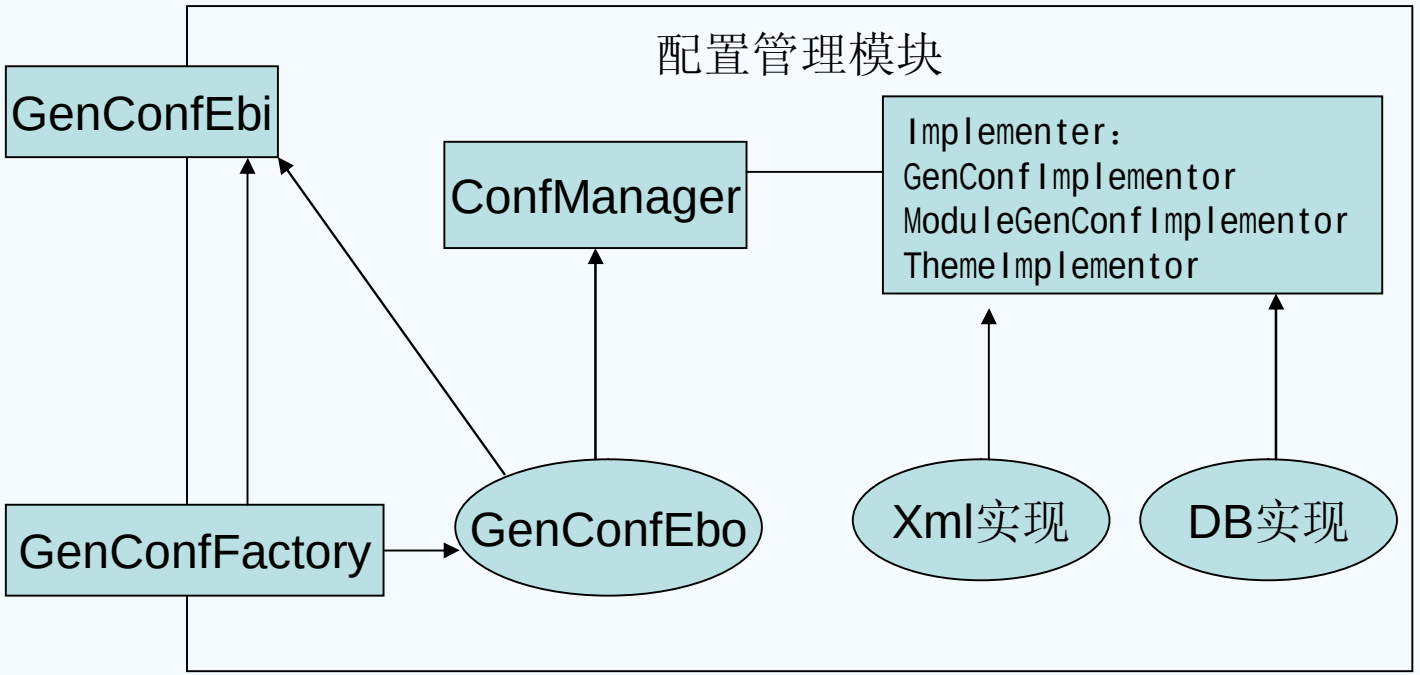
可以把获取配置数据这边，设计成为桥接模式的实现部分，而数据模型这边设计成为桥接模式的抽象部分，而且数据模型这边确实需要使用具体实现部分来获取数据，简直就是标准的桥接模式的应用。

而ConfManager就相当于桥接的抽象部分的顶层实现，而GenConfEbo就相当于使用基本的抽象部分的扩展部分，只不过这里采用的是对象组合的方式，而不是标准桥接模式中的对象继承的方式。

具体实现那边，需要先定义出实现的接口来，然后不同的配置方式对应不同的实现。

应用桥接模式

- 此时配置管理模块的结构示意如图



加入解释器模式

- 面临的问题

  自己解析xml，本来也没有什么特别困难的，但是问题就在于，如果xml文件的格式发生了变化，那么读取配置文件的程序就需要做出相应的变更，严重的时候，几乎相当于完全重写程序。

  那么怎么解决当xml的结构发生改变过后，能够很方便的获取相应元素、或者是属性的值，而不用再去修改解析xml的程序呢？

- 用解释器模式来解决

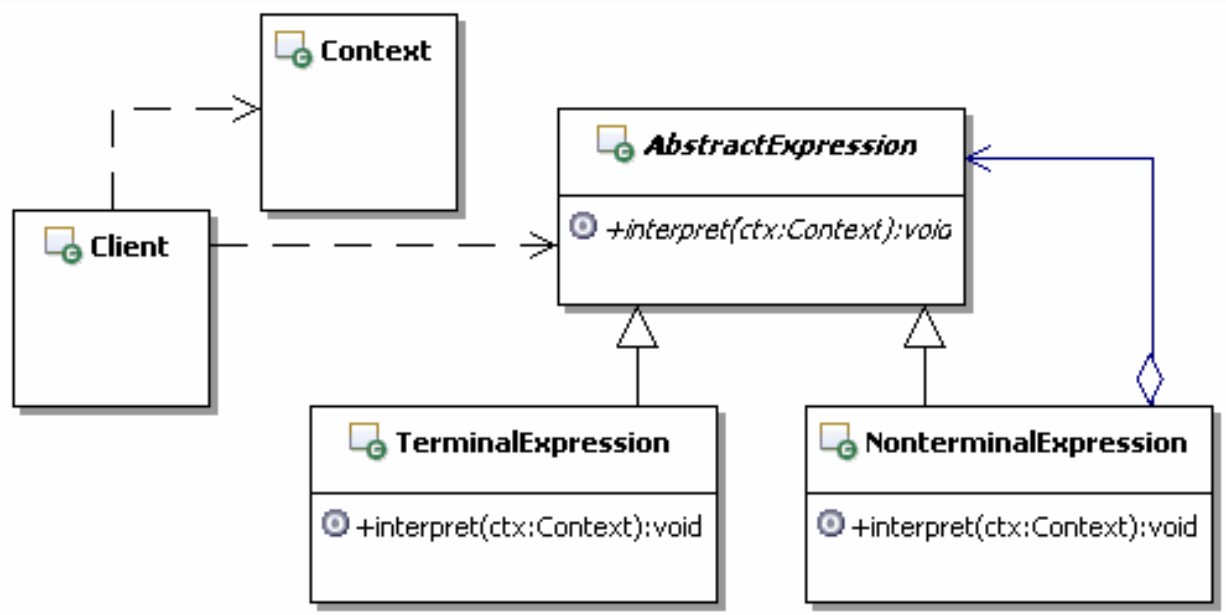
- 解释器模式基础回顾

初识解释器模式

- 定义

  给定一个语言，定义它的文法的一种表示，并定义一个解释器，这个解释器使用该表示来解释语言中的句子。

- 结构和说明



解释器模式的知识要点

- 解释器模式的知识要点

1：解释器模式使用解释器对象来表示和处理相应的语法规则，一般一个解释器处理一条语法规则。

2：解释器模式没有定义谁来构建抽象语法树，把这个工作交给了客户端处理

3：使用解释器模式的时候，应该先定义好相应的语法规则，并根据规则制定解释器的语法树；然后客户端在调用解释器进行解释操作的时候，需要自行构建符合语法规则要求的语法树；而解释器模式只是负责解释并执行。

4：从实质上看，解释器模式的思路仍然是分离、封装、简化，跟很多模式是一样的。

思考解释器模式

- 解释器模式的本质

  解释器模式的本质是：**分离实现，解释执行**

- 何时选用解释器模式

1：当有一个语言需要解释执行，并且可以将该语言中的句子表示为一个抽象语法树的时候，可以考虑使用解释器模式。

在使用解释器模式的时候，还有两个特点需要考虑，一个是语法相对应该比较简单，太复杂的语法不合适使用解释器模式；另一个是效率要求不是很高，对效率要求很高的情况下，不适合使用解释器模式。

应用解释器模式

- 使用解释器模式来解决问题的思路

要解决通用解析xml的问题，第一步：需要先设计一个简单的表达式语言，在客户端调用解析程序的时候，传入用这个表达式语言描述的一个表达式，然后把这个表达式通过解析器的解析，形成一个抽象的语法树。

第二步：解析完成后，自动调用解释器来解释抽象语法树，并执行每个节点所对应的功能，从而完成通用的xml解析。

这样一来，每次当xml结构发生了更改，也就是在客户端调用的时候，传入不同的表达式即可，整个解析xml过程的代码都不需要再修改了。

应用解释器模式

- 约定简单的语法规则如下，为了通用，用root表示根元素，a、b、c、d等来代表元素，一个简单的xml如下：

```
<?xml version="1.0" encoding="UTF-8"?>
<root id="rootId">
  <a>
    <b>
      <c name="testC">12345</c>
      <d id="1">d1</d>
      <d id="2">d2</d>
      <d id="3">d3</d>
      <d id="4">d4</d>
    </b>
    <e id="e1">
      <f>f1</f>
    </e>
    <e id="e2">
      <f>f2</f>
    </e>
  </a>
</root>
```

应用解释器模式

- 约定简单的语法规则如下：

1：获取单个元素的值：从根元素开始，一直到想要获取值的元素，元素中间用“/”分隔，根元素前不加“/”。比如表达式“root/a/b/c”就表示获取根元素下、a元素下、b元素下的c元素的值

2：获取单个元素的属性的值：要获取值的属性一定是表达式的最后一个元素的属性，在最后一个元素后面添加“.”然后再加上属性的名称。比如表达式“root/a/b/c.name”就表示获取根元素下、a元素下、b元素下、c元素的name属性的值

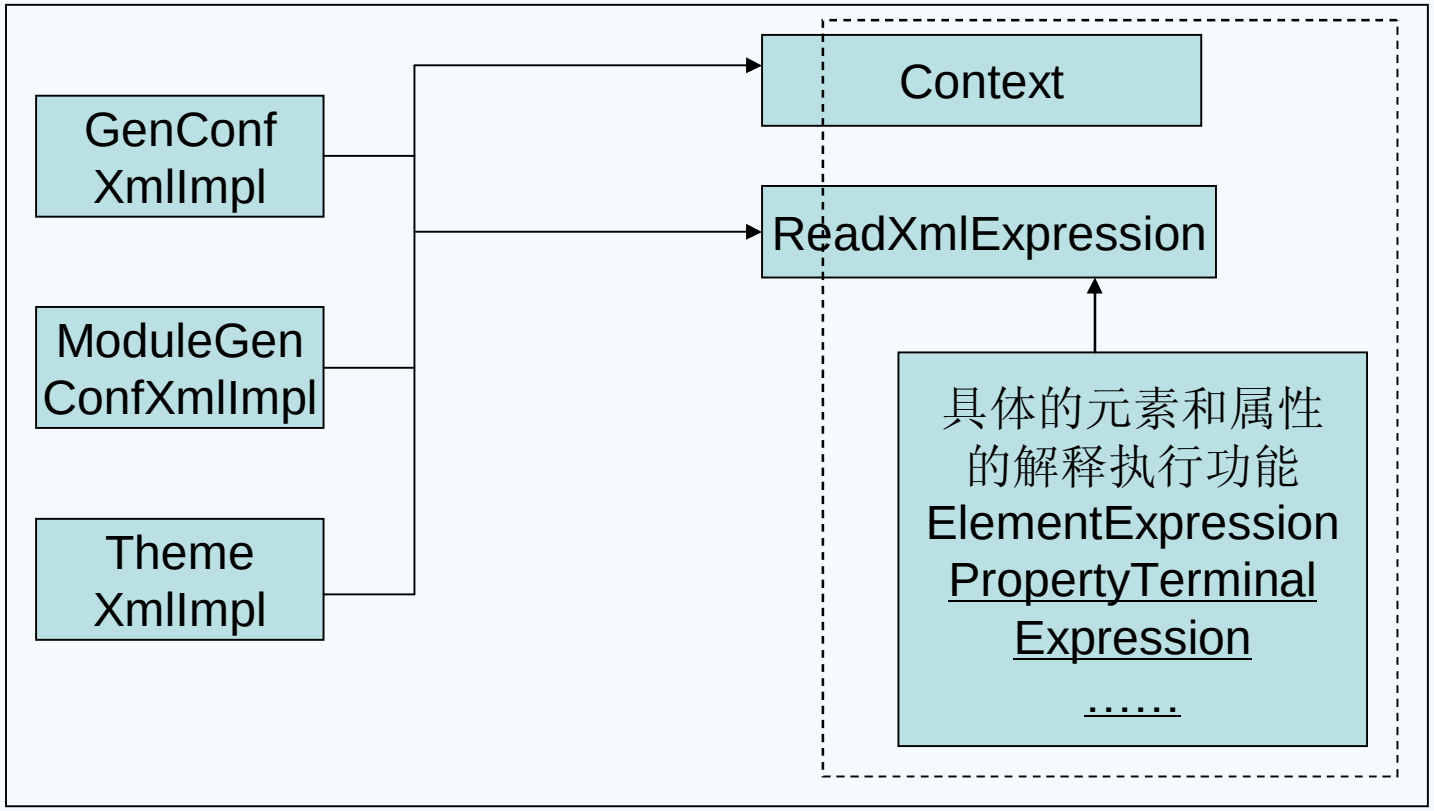
3：获取相同元素名称的值，当然是多个：要获取值的元素一定是表达式的最后一个元素，在最后一个元素后面添加“$”。比如表达式“root/a/b/d$”就表示获取根元素下、a元素下、b元素下的多个d元素的值的集合

4：获取相同元素名称的属性的值，当然也是多个：要获取属性值的元素一定是表达式的最后一个元素，在最后一个元素后面添加“$”，然后在后面添加“.”然后再加上属性的名称，在属性名称后面也添加“$”。比如表达式“root/a/b/d$.id$”就表示获取根元素下、a元素下、b元素下的多个d元素的id属性的值的集合

5：如果要获取某个需要区分的元素下面的值，那么对于判断使用这样的语法：[属性名=值]，属性名和值都不需要加引号，比如：要想获取id=”e1”的e元素下面的f元素的值，就是用这样的表达式：root/a/e$[id=e1]/f。

应用解释器模式

- 此时配置管理模块中读取XML部分的结构示意如图



加入组合模式

- 面临的问题

  分析前面解释器模式的实现，会发现对于客户端而言，并不想要去区分到底是非终结符对象还是终结符对象，只是想要以一个统一的方式来请求解析。

  该怎么解决这个问题呢？

- 用组合模式来解决

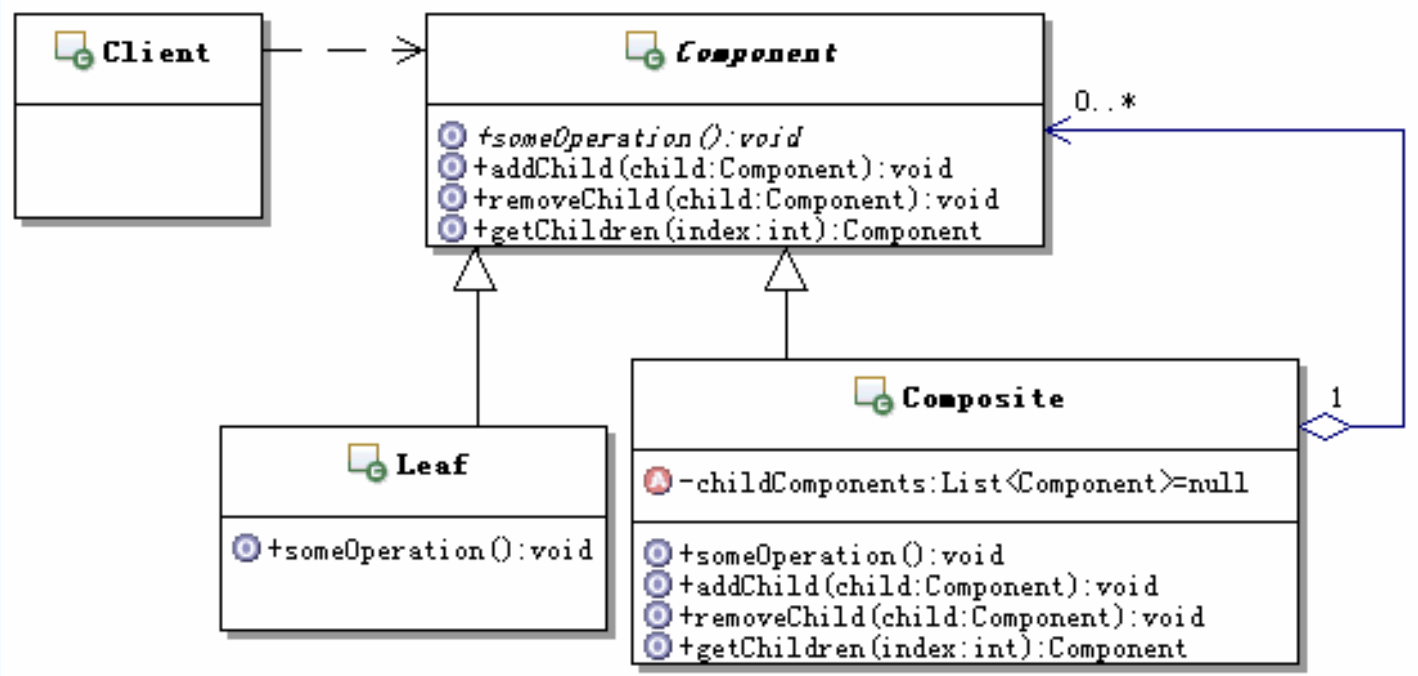
- 组合模式基础回顾

初识组合模式

- 定义

  将对象组合成树形结构以表示“部分-整体”的层次结构。组合模式使得用户对单个对象和组合对象的使用具有一致性 。

- 结构和说明



组合模式的知识要点

- 组合模式的知识要点

1：组合模式的目的是：让客户端不再区分操作的是组合对象还是叶子对象，而是以一个统一的方式来操作。

2：组合模式设计的关键之处，是设计一个抽象的组件类，让它可以代表组合对象和叶子对象。

3：组合模式通常会构成对象树，理论上可以包含无数个层次

4：通常需要以对象递归的方式来访问对象树种的对象

5：使用组合模式的时候，应该更关注于透明性而非安全性

6：必要的时候，要考虑父子组件的引用关系，并避免环状引用，当然有意这样设计的除外

思考组合模式

- 组合模式的本质

  组合模式的本质是：**统一叶子对象和组合对象**

- 何时选用组合模式

1：如果你想表示对象的部分-整体层次结构，可以选用组合模式，把整体和部分的操作统一起来，使得层次结构实现更简单，从外部来使用这个层次结构也简单。

2：如果你希望统一的使用组合结构中的所有对象，可以选用组合模式，这正是组合模式提供的主要功能

应用组合模式

- 使用组合模式来解决问题的思路

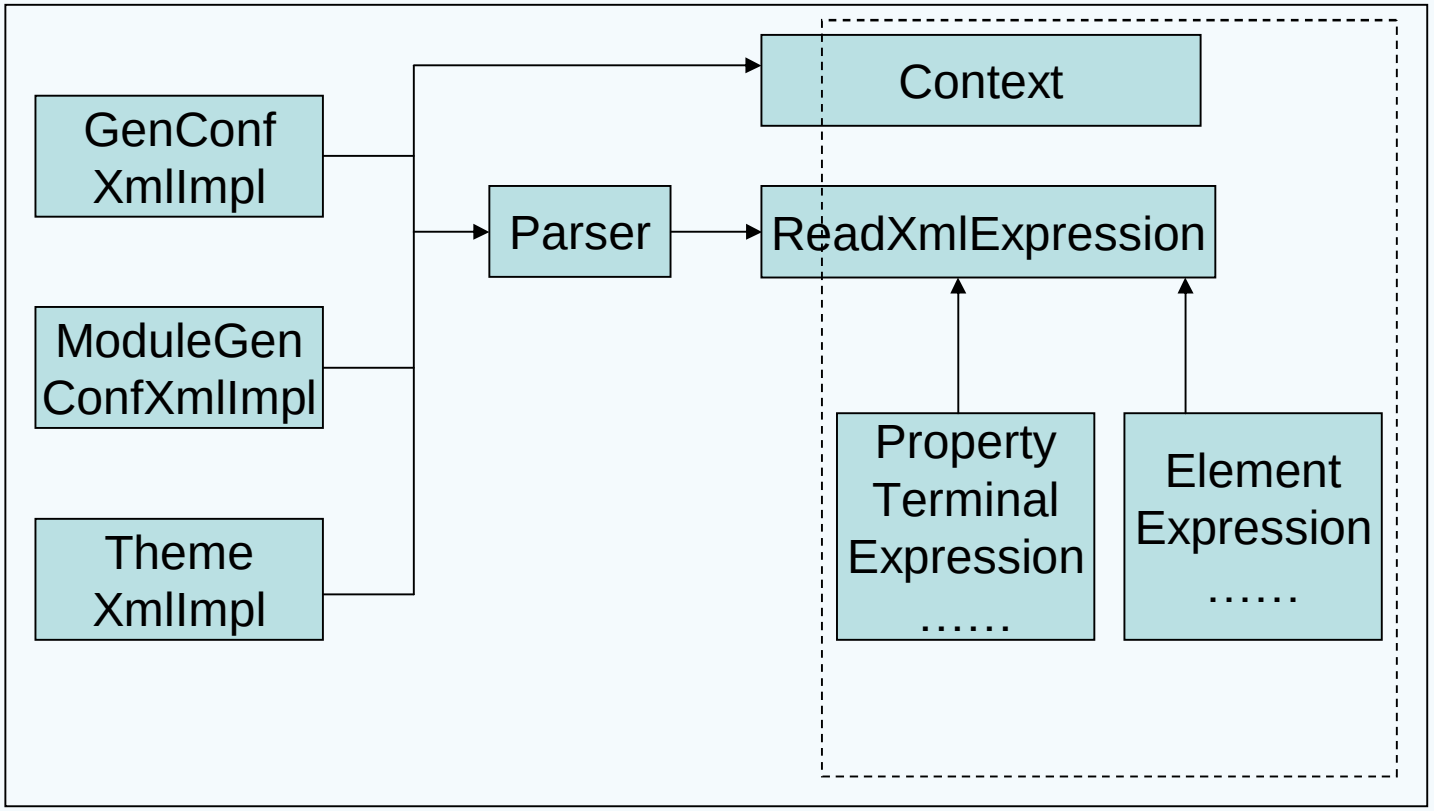
事实上，解释器模式本身就已经应用了组合模式来设计了，比如上面的类实现中，ReadXmlExpression就相当于组合模式的Component，而ElementExpression、ElementsExpression就相当于树枝对象，而ElementsTerminalExpression、ElementTerminalExpression、PropertyTerminalExpression、PropertysTerminalExpression就相当于树叶对象。

对于解释器模式而言，只是负责按照抽象语法树进行解析，可是抽象语法树怎么来呢？就是谁来负责组合这棵树呢？

那就需要我们自己来写一个解析器对象了，这个解析器对象就相当于解释器模式的客户端，在这个对象里面去组合抽象语法树。

应用组合模式

- 此时配置管理模块中读取XML部分的结构示意如图



加入备忘录模式

- 面临的问题

  在根据字符串来构建对应的抽象语法树的时候，有很多字符串前面都是一样的，这样一来，每次重复创建太浪费时间了，最好是相同字符串对应的树形对象就不用创建了，直接分析后面不同的，然后把新的部分添加到已有的这棵树里面就可以了。

  那么该怎么样来处理呢？

- 用备忘录模式来解决

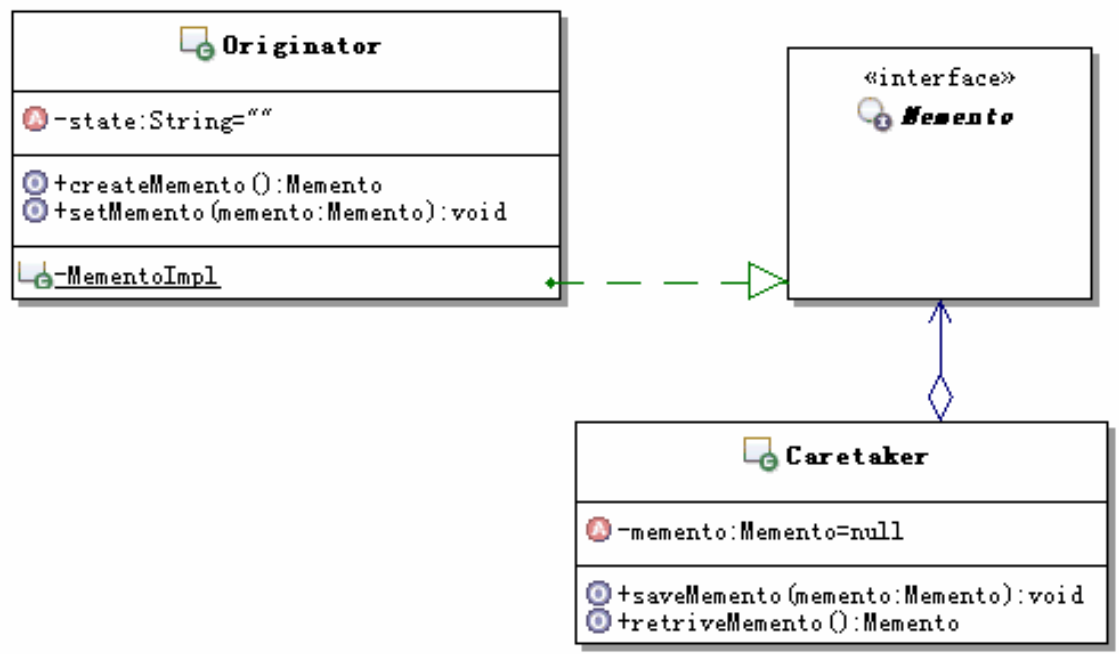
- 备忘录模式基础回顾

初识备忘录模式

- 定义

  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。这样以后就可将该对象恢复到原先保存的状态 。

- 结构和说明



备忘录模式的知识要点

- 备忘录模式的知识要点

1：备忘录模式的功能，首先是在不破坏封装性的前提下，捕获一个对象的内部状态。但是要记住，备忘的目的是为了在后面需要的时候进行恢复，把对象的状态恢复到备忘录所保存的状态，这才是备忘录真正的目的。

2：备忘录对象通常都会实现窄接口，一般不考虑宽接口，而且通常会实现成为原发器对象的一个私有的静态内部类

3：管理者对象可要可不要，管理者对象只是存放备忘录对象，并不能操作备忘录对象

4：使用备忘录模式，潜在的代价就是比较耗费内存

5：备忘录模式通常会结合原型模式来使用

6：备忘录的存储方式可以多样，除了内存，还可以离线存储

思考备忘录模式

- 备忘录模式的本质

  备忘录模式的本质是：**保存和恢复内部状态**

- 何时选用备忘录模式

1：如果必须保存一个对象在某一个时刻的全部或者部分状态，这样在以后需要的时候，可以把该对象恢复到先前的状态。可以使用备忘录模式，使用备忘录对象来封装和保存需要保存的内部状态，然后把备忘录对象保存到管理者对象里面，在需要的时候，再从管理者对象里面获取备忘录对象，来恢复对象的状态

2：如果需要保存一个对象的内部状态，但是如果用接口来让其它对象直接得到这些需要保存的状态，将会暴露对象的实现细节并破坏对象的封装性。可以使用备忘录模式，把备忘录对象实现成为原发器对象的内部类，而且还是私有的，从而保证只有原发器对象才能访问该备忘录对象。这样既保存了需要保存的状态，又不会暴露原发器对象的内部实现细节。

应用备忘录模式

- 使用备忘录模式来解决问题的思路

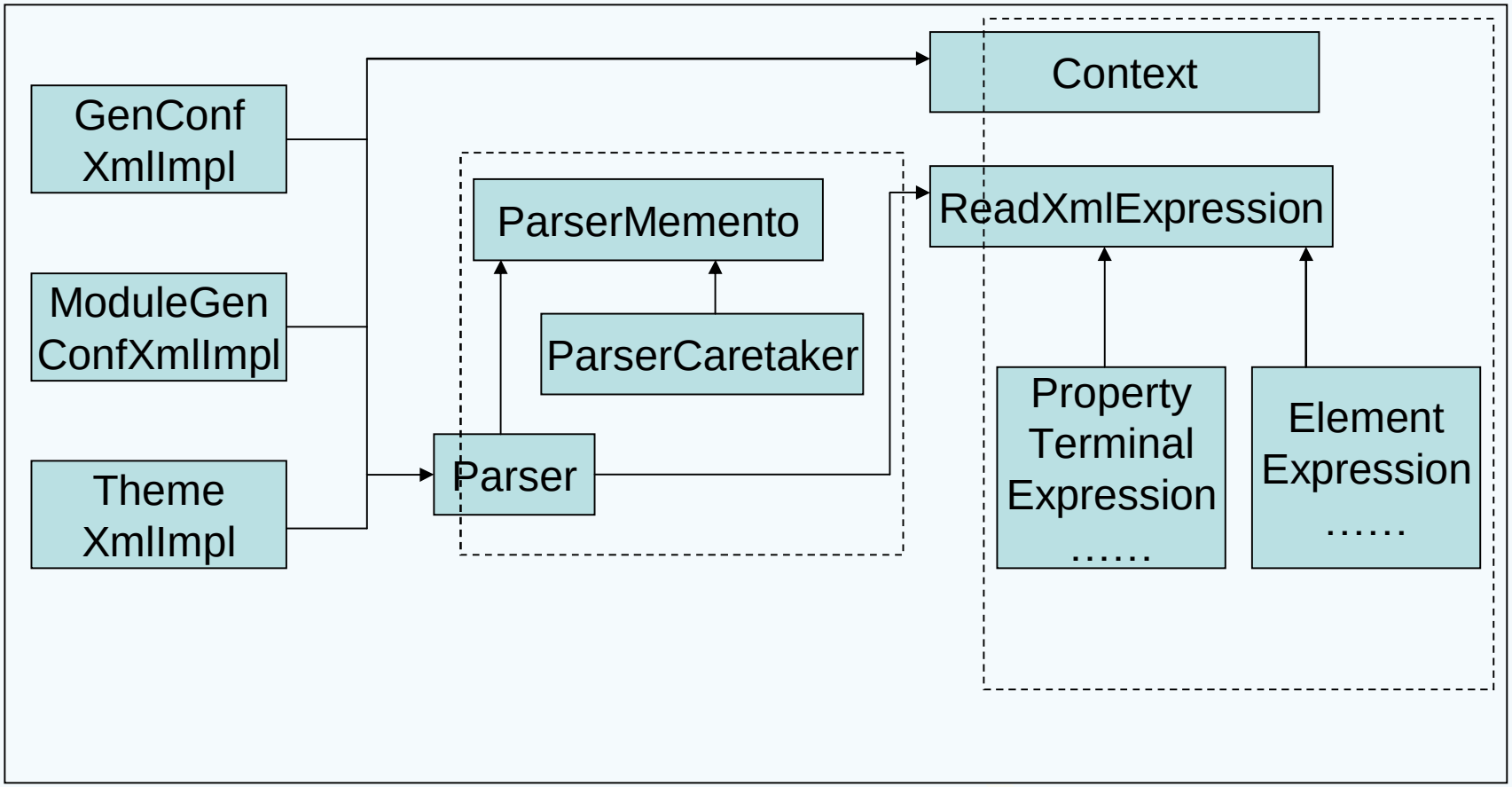
每次在拿到一个字符串过后，先跟已有的记录进行比较，找出最大的已经解析过的字符串，那么这个长度的字符串是不需要再解析的，只需要把后面没有解析的字符串拿出来进行解析，然后把解析的对象添加到已经解析好的这个对象后头就可以了。

这就需要每次在解析的时候，每次解析完成，就向备忘录里面添加一条记录，而每次进入的时候，根据最长能匹配的字符串，从备忘录里面获取到相应的对象，这就不用解析了。

当然，备忘录对象还是需要实现一个窄接口，也需要实现一个备忘录管理者。这里跟标准的备忘录模式有一个小小的区别，这里是在解析器对象里面直接去备忘录管理者里面获取备忘录对象，而不是从客户端来传递，这是为了让客户端操作更方便，其实由客户端传递也是可以的。

应用备忘录模式

- 此时配置管理模块中读取XML部分的结构示意如图



加入原型模式

- 面临的问题

  当我们把备忘录模式加入过后，就需要从备忘录里面获取字符串对应的对象，也需要在解析过后，把当前解析好的对象设置到备忘录里面，这里就出现了一个新的问题？那就是这些对象如果直接设置到备忘录里面，那么大家都是指向同一个内存空间，那么当解析中操作这些对象的时候，就会影响到备忘录中的对象。

  那么该怎么解决这个问题呢？

- 用原型模式来解决

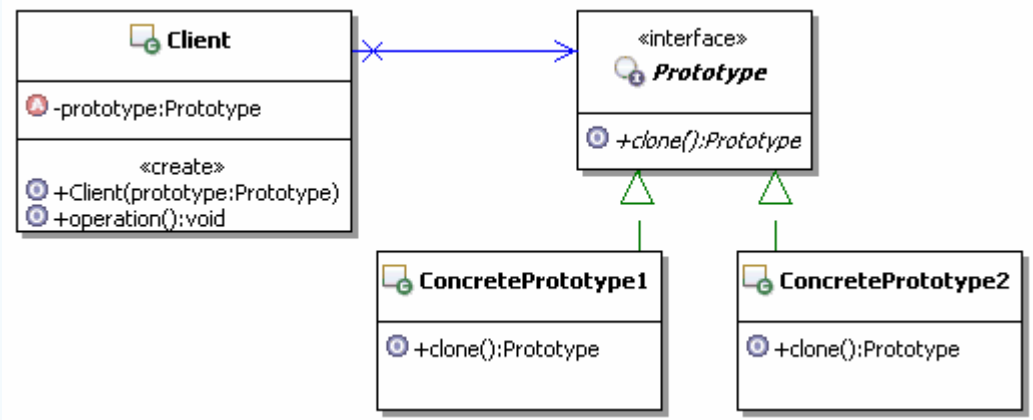
- 原型模式基础回顾

初识原型模式

- 定义

  用原型实例指定创建对象的种类，并通过拷贝这些原型创建新的对象。

- 结构和说明



Prototype：声明一个克隆自身的接口，用来约束想要克隆自己的类，要求它们都要实现这里定义的克隆方法。

ConcretePrototype：实现Prototype接口的类，这些类真正实现克隆自身的功能

Client：使用原型的客户端，首先要获取到原型实例对象，然后通过原型实例克隆自身来创建新的对象实例。

原型模式的知识要点

- 原型模式的知识要点

1：原型模式的主要功能就是：通过克隆来创建新的对象实例。一般来讲，新创建出来的实例的数据是和原型实例一样的。

2：原型模式的克隆方法和new操作最明显的不同就在于：new一个对象实例，一般属性是没有值的，或者是只有默认值；如果是克隆得到的一个实例，通常属性是有值的，属性的值就是原型对象实例在克隆的时候，原型对象实例的属性的值。

3：Java中已经有了clone方法的基本实现，可以直接使用，但是只是实现了浅度克隆，如果是深度克隆，还是需要额外代码实现

4：要考虑浅度克隆和深度克隆的问题，要想深度克隆成功，必须要整个克隆所涉及的对象都要正确实现克隆方法，如果其中有一个没有正确实现克隆，那么就会导致克隆失败。

5：原型模式的重心在创建新的对象实例，克隆是实现的手段。

思考原型模式

- 原型模式的本质

  原型模式的本质是：**克隆生成对象**

- 何时选用原型模式

1：如果一个系统想要独立于它想要使用的对象时，可以使用原型模式，让系统只面向接口编程，在系统需要新的对象的时候，可以通过克隆原型来得到

2：如果需要实例化的类是在运行时刻动态指定时，可以使用原型模式，通过克隆原型来得到需要的实例

应用原型模式

- 使用原型模式来解决问题的思路

  分析上面的问题，发现是需要把这些对象的数据设置到备忘录里面，因此一个简单的方法就是使用原型模式，通过克隆这些对象，这样既得到一个新的对象，而值又完全是一样的，这样一来，把这个新创建的对象设置到备忘录里面，大家就互不干扰了。

  同样的道理，要从备忘录里面取出这些对象的时候，也不是直接返回备忘录里面的对象，而是重新克隆一个对象，然后返回这个新的对象，从而保证备忘数据和解析操作的数据是相互独立的。

加入生成器模式

- 面临的问题

  现在已经实现了通过一个字符串，就能够自动创建抽象语法树，然后通过解释器去解释执行，最终获取相应的数据的功能了。

  而且这个功能是通用的，语法也是比较简洁的，基本可以满足我们的功能要求了。但是新的问题又出现了，那就是要解析一个xml，需要拼接很多很多的字符串，而这些字符串的拼接过程是十分类似的，只是最终拼接后的结果不一样，那么有没有什么好方法，来让这些字符串的拼接过程变得简单、统一呢？

- 用生成器模式来解决

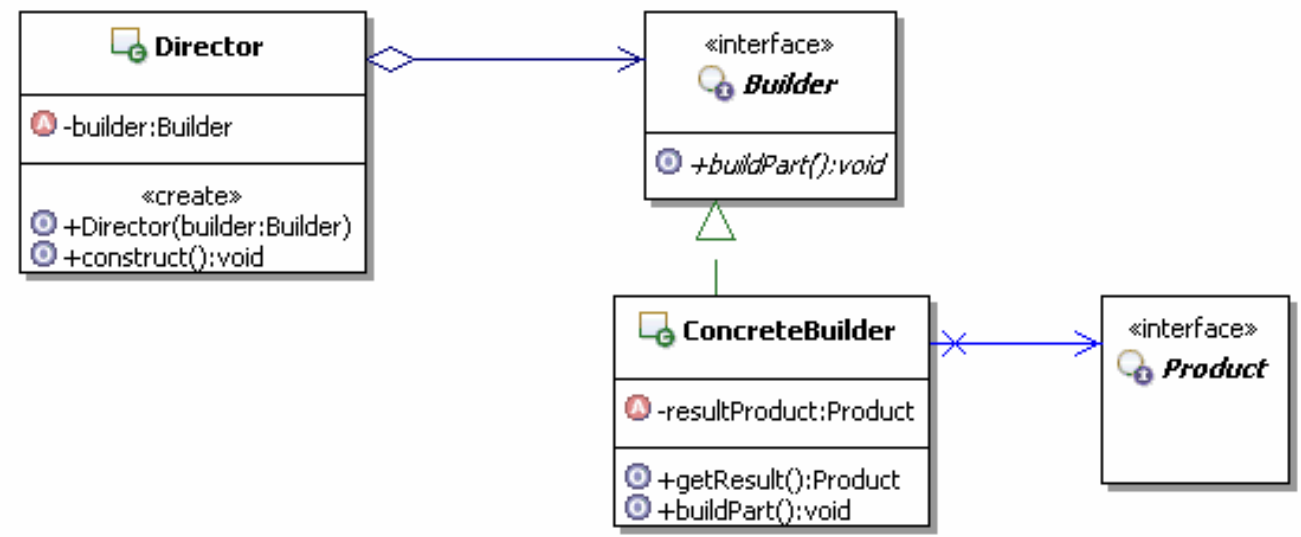
- 生成器模式基础回顾

初识生成器模式

- 定义

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

- 结构和说明



生成器模式的知识要点

n 生成器模式的知识要点

1：生成器模式重在解决一步一步构造复杂对象的问题，更为重要的是，这个构建的过程是统一的，固定不变的，变化的部分放到生成器部分了。

2：生成器模式的重心在于分离构建算法和具体的构造实现，从而使得构建算法可以重用，具体的构造实现可以很方便的扩展和切换，从而可以灵活的组合来构造出不同的产品对象。

3：生成器模式一般存在两个部分，一个部分是部件构造和产品装配，另一个部分是整体构建的算法。

4：生成器模式有一种退化的情况，就是让客户端和Director融合起来，让客户端直接去操作Builder，就好像是指导者自己想要给自己构建产品一样。

5：生成器模式里面，指导者承担的是整体构建算法部分，是相对不变的部分。

6：指导者分离出去的变化部分，就到了生成器那边，每个生成器对象都可以有两部分功能，一个是创建部件对象，一个是组装部件。

7：在标准的生成器模式里面，在Builder实现里面会提供一个返回装配好的产品的方法，在Builder接口上是没有的。

8：在标准的生成器模式里面，一般是不需要对产品定义抽象接口。

思考生成器模式

- 生成器模式的本质

  生成器模式的本质是：**分离整体构建算法和部件构造**

- 何时选用生成器模式

1：如果创建对象的算法，应该独立于该对象的组成部分以及它们的装配方式时

2：如果同一个构建过程有着不同的表示时

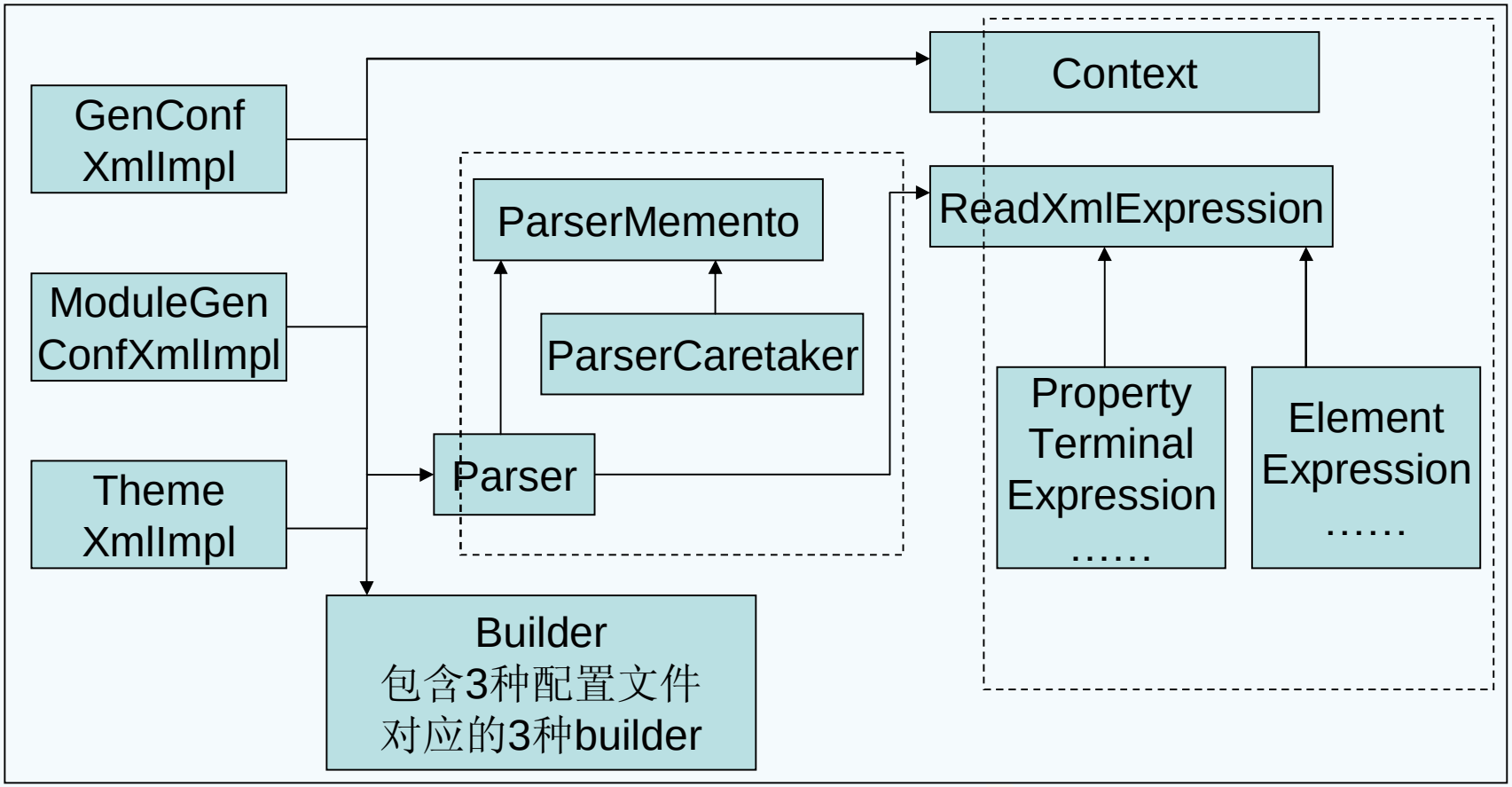
应用生成器模式

- 使用生成器模式来解决问题的思路

  每个字符串都是从根节点开始，依次拼接到最后需要取值的地方，因此可以把这个拼接的过程统一起来，做成生成器，而调用这些生成器来构建最终产品的客户端，这个时候就充当了生成器模式中的指导者。

应用生成器模式

- 此时配置管理模块中读取XML部分的结构示意如图



加入策略模式

n 面临的问题

现在字符串也可以通过Builder模式来拼接了，看起来一切很棒。又有一个新的问题出现了，那就是配置的数据里面有动态的内容，需要动态解析。有些朋友会说，哪里有动态的内容呢？

由于在配置ExtendConf的值时，会发现经常需要在某一个配置里面，需要使用已经配过的一个值，这个值可能直接来之本模块其它的ExtendConf的值，也可能来自GenConf中配置的数据。因而设计了表达动态引用的语法，大致如下：

1：直接引用本模块配置的其它ExtendConf的数据，有两种方式，如果只是简单引用某一个配置的值，那么就直接写成${引用的ExtendConf的id}，示例如下：

```
<ExtendConfs>
  <ExtendConf id="moduleName" isSingle="true">user</ExtendConf>
  <ExtendConf id="modulePackge"
   isSingle="true">cn.javass.${moduleName}</ExtendConf>
</ExtendConfs>
```

加入策略模式

2：直接引用本模块配置的其它ExtendConf的数据，还有两种方式，x-gen支持Beanshell的脚本，并自动传入gm代表GenConfModel，mapEcms代表当前模块配置的mapExtends，那么就可以写成${mapEcms.get(“引用的ExtendConf的id”);}，示例如下：

```
<ExtendConfs>
  <ExtendConf id="moduleName" isSingle="true">user</ExtendConf>
  <ExtendConf id="modulePackge"
    isSingle="true">cn.javass.${ mapEcms.get(“moduleName”);}</ExtendConf>
</ExtendConfs>
```

当然，Beanshell脚本更多的知识，请参见Beanshell随机文档。总之现在要解析包含动态内容的配置，而且有两种基本的解析方式，一种是直接替换当前模块内的配置数据，另外一种是用Beanshell来运行配置的表达式，以获得最后的结果。那么在具体运行时，我们该到底使用哪一种方式来实现动态解析呢？

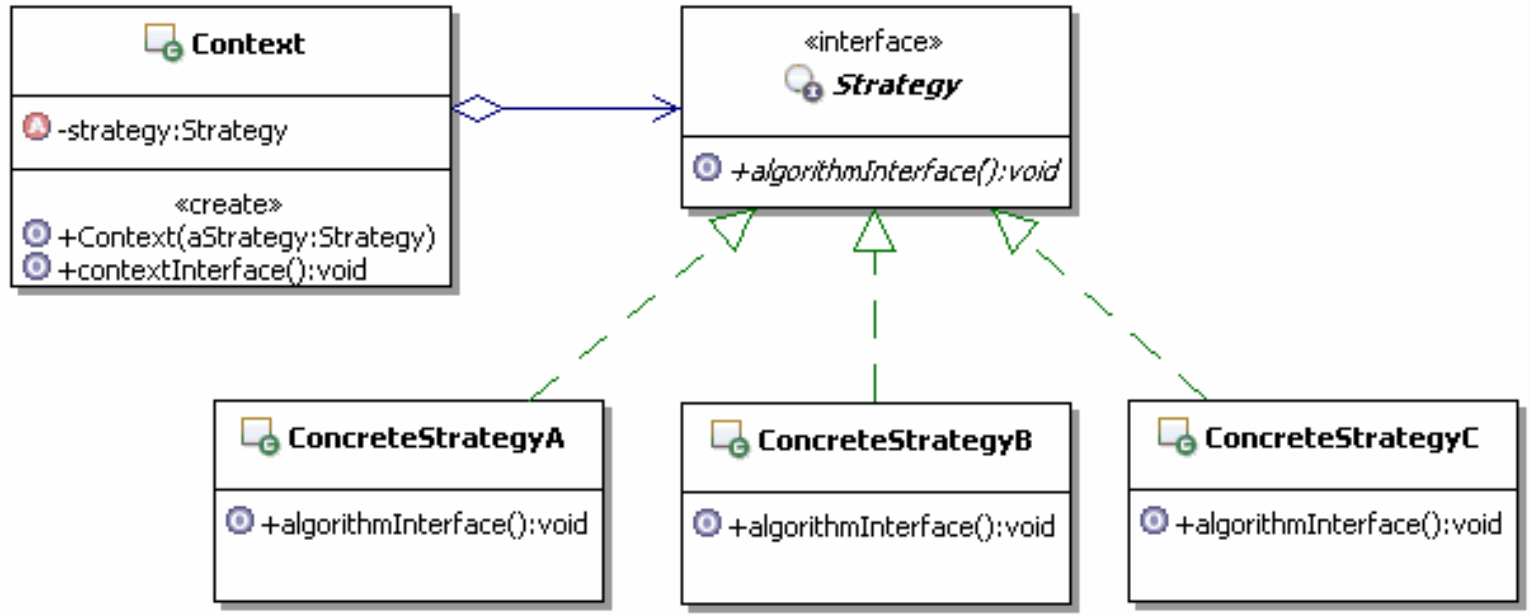
- 用策略模式来解决
- 策略模式基础回顾

初识策略模式

- 定义

  定义一系列的算法，把它们一个个封装起来，并且使它们可相互替换。本模式使得算法可独立于使用它的客户而变化 。

- 结构和说明



策略模式的知识要点

n 策略模式的知识要点

1：策略模式的功能是把具体的算法实现，从具体的业务处理里面独立出来，把它们实现成为单独的算法类，从而形成一系列的算法，并让这些算法可以相互替换。

2：策略模式的重心不是如何来实现算法，而是如何组织、调用这些算法，从而让程序结构更灵活、具有更好的维护性和扩展性。

3：策略模式一个很大的特点就是各个策略算法的平等性。所有的策略算法在实现上也是相互独立的，相互之间是没有依赖的。所以可以这样描述这一系列策略算法：策略算法是相同行为的不同实现。

4：使用策略模式的时候，要注意谁来选择策略，可以是客户端，也可以在上下文里面。而跟策略模式类似的状态模式，一般是不会让客户端来选择状态的，状态是内部行为，这是一个很重要的区别。

5：策略模式在每一个时刻只能使用一个具体的策略实现对象，虽然可以动态的在不同的策略实现中切换，但是同时只能使用一个。

6：上下文在策略模式里面有特殊的地位，上下文使用具体的策略实现对象，反过来，策略实现对象也可以从上下文获取所需要的数据。甚至在某些情况下，策略实现对象还可以回调上下文的方法来实现一定的功能，这种使用场景下，上下文变相充当了多个策略算法实现的公共接口，在上下文定义的方法可以当做是所有或者是部分策略算法使用的公共功能。

思考策略模式

- 策略模式的本质

  策略模式的本质是：**分离算法，选择实现**

- 何时选用策略模式

1：出现有许多相关的类，仅仅是行为有差别的情况，可以使用策略模式来使用多个行为中的一个来配置一个类的方法，实现算法动态切换

2：出现同一个算法，有很多不同的实现的情况，可以使用策略模式来把这些“不同的实现”实现成为一个算法的类层次

3：需要封装算法中，与算法相关的数据的情况，可以使用策略模式来避免暴露这些跟算法相关的数据结构

4：出现抽象一个定义了很多行为的类，并且是通过多个if-else语句来选择这些行为的情况，可以使用策略模式来代替这些条件语句

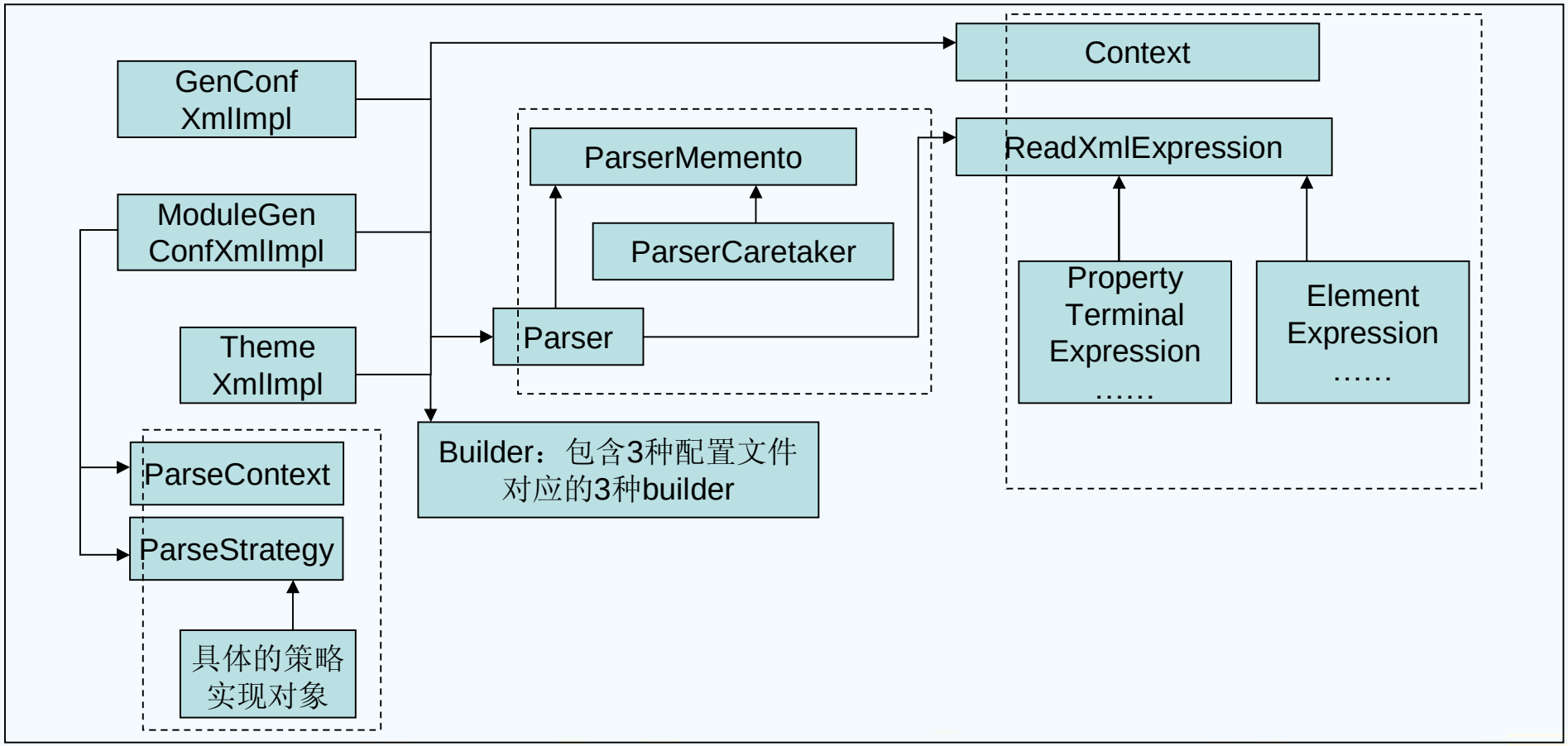
应用策略模式

- 使用策略模式来解决问题的思路

  很明显，上面描述的场景是策略模式适用的场景，两种解析动态内容的方式就是两种算法，而要判断到底该使用哪一种方式来实现动态解析，就可以在策略模式的上下文中进行就可以了。

应用策略模式

- 此时配置管理模块读取XML部分的结构示意如图



恭喜！恭喜！

恭喜！恭喜！

学到这里，你已经完成了配置管理模块的实现，并综合应用了9种以上的设计模式！