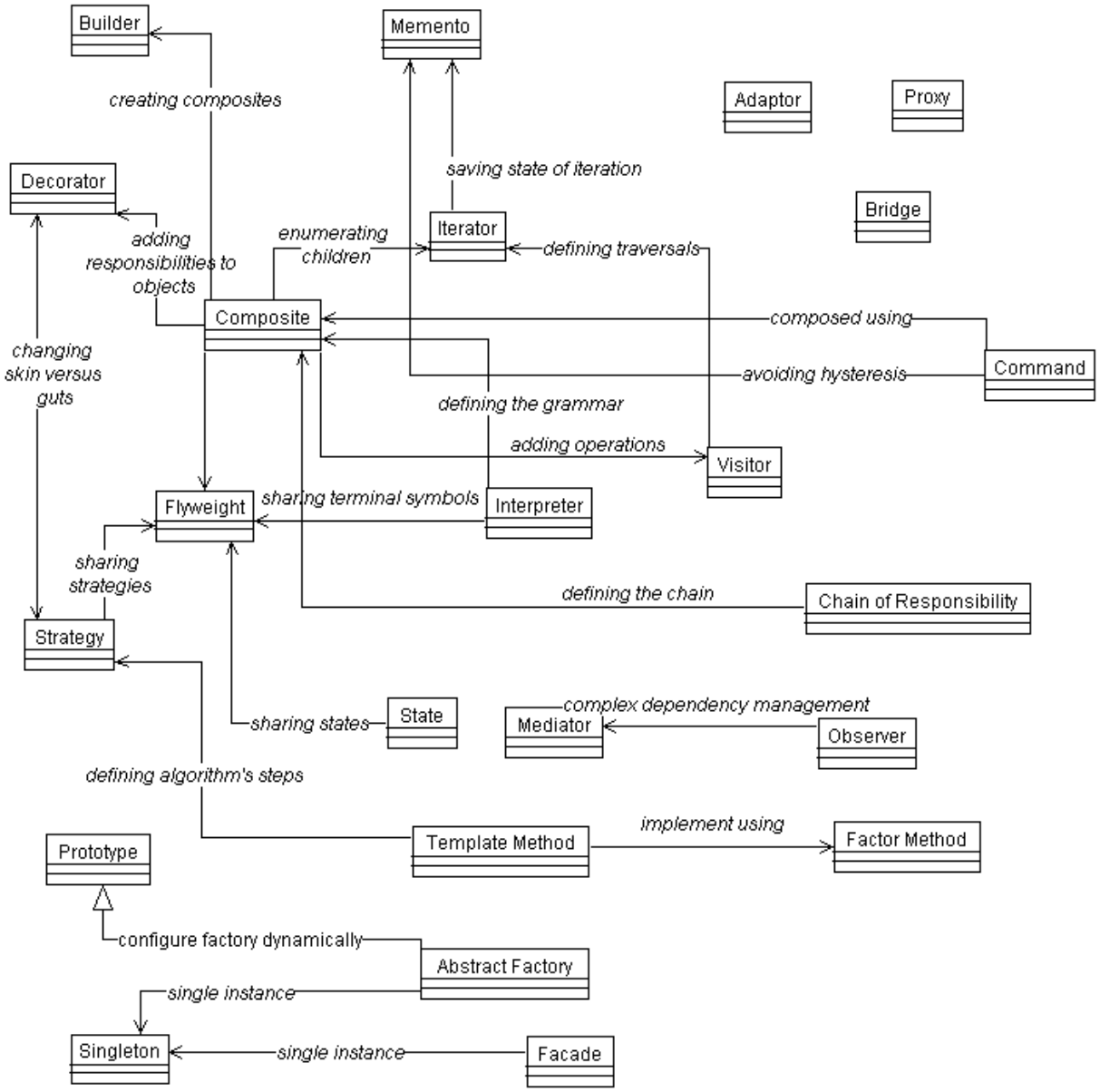
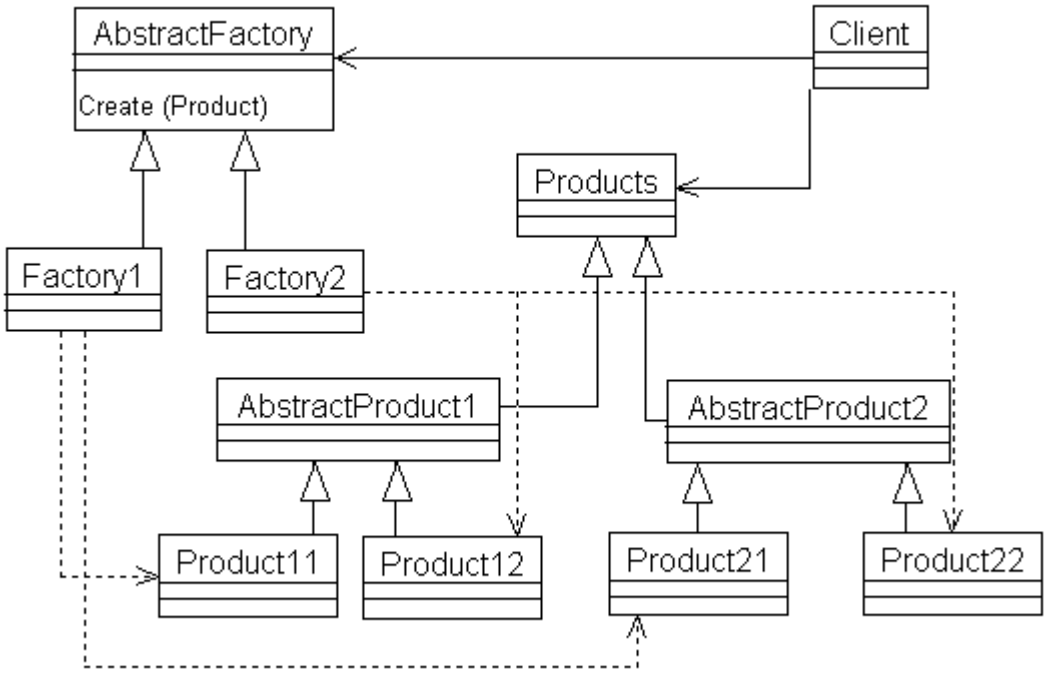
1. JAVA 设计模式



1.1. 创建型模式

1.1.1. Abstract Factory—抽象工厂模式

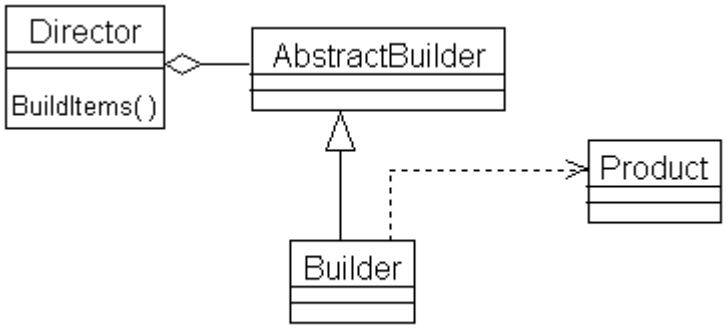
追 MM 少不了请吃饭了，麦当劳的鸡翅和肯德基的鸡翅都是 MM 爱吃的东西，虽然口味有所不同，但不管你带 MM 去麦当劳或肯德基，只管向服务员说“来四个鸡翅”就行了。麦当劳和肯德基就是生产鸡翅的 Factory



客户类和工厂类分开。消费者任何时候需要某种产品，只需向工厂请求即可。消费者无须修改就可以接纳新产品。缺点是当产品修改时，工厂类也要做相应的修改。如：如何创建及如何向客户端提供。

1.1.2. Builder—建造模式

MM 最爱听的就是“我爱你”这句话了，见到不同地方的 MM,要能够用她们的方言跟她说这句话哦，我有一个多种语言翻译机，上面每种语言都有一个按键，见到 MM 我只要按对应的键，它就能够用相应的语言说出“我爱你”这句话了，国外的 MM 也可以轻松搞掂，这就是我的“我爱你”builder。（这一定比美军在伊拉克用的翻译机好卖）

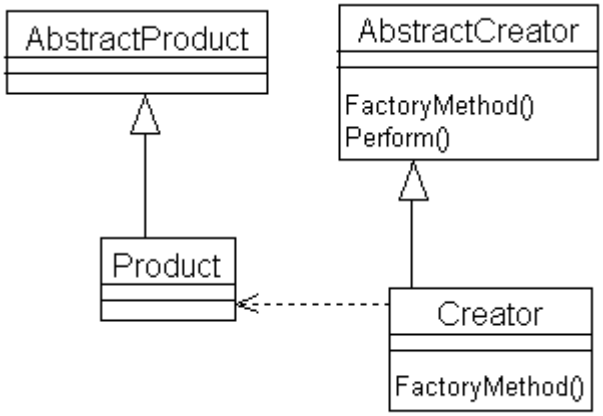


将产品的内部表象和产品的生成过程分割开来，从而使一个建造过程生成具有不同的内部表象的产品对象。建造模式使得产品内部表象可以独立的变化，客户不必知道产品内部组成的细节。建造模式可以强制实行一种分步骤进行的建造过程。

1.1.3. Factory Method—工厂方法模式

请 MM 去麦当劳吃汉堡，不同的 MM 有不同的口味，要每个都记住是一件烦人的事情，我一般采用 Factory Method 模式，带着 MM 到服务员那儿，说“要一个汉堡”，

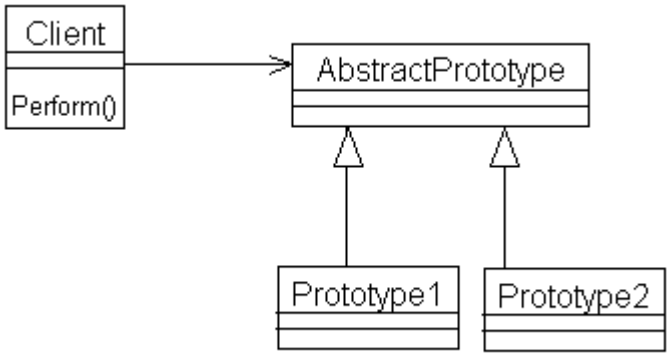
具体要什么样的汉堡呢，让 MM 直接跟服务员说就行了。



核心工厂类不再负责所有产品的创建，而是将具体创建的工作交给子类去做，成为一个抽象工厂角色，仅负责给出具体工厂类必须实现的接口，而不接触哪一个产品类应当被实例化这种细节。

1.1.4. Prototype—原始模型模式

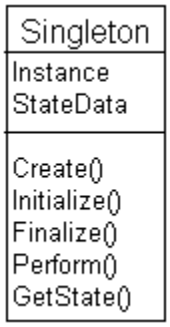
跟 MM 用 QQ 聊天，一定要说些深情的话语了，我搜集了好多肉麻的情话，需要时只要 copy 出来放到 QQ 里面就行了，这就是我的情话 prototype 了。（100 块钱一份，你要不要）



通过给出一个原型对象来指明所要创建的对象的类型，然后用复制这个原型对象的方法创建出更多同类型的对象。原始模型模式允许动态的增加或减少产品类，产品类不需要非得有任何事先确定的等级结构，原始模型模式适用于任何的等级结构。缺点是每一个类都必须配备一个克隆方法。

1.1.5. Singleton—单例模式

俺有 6 个漂亮的老婆，她们的老公都是我，我就是我们家里的老公 Sigleton，她们只要说道“老公”，都是指的同一个人，那就是我(刚才做了个梦啦，哪有这么好的事)

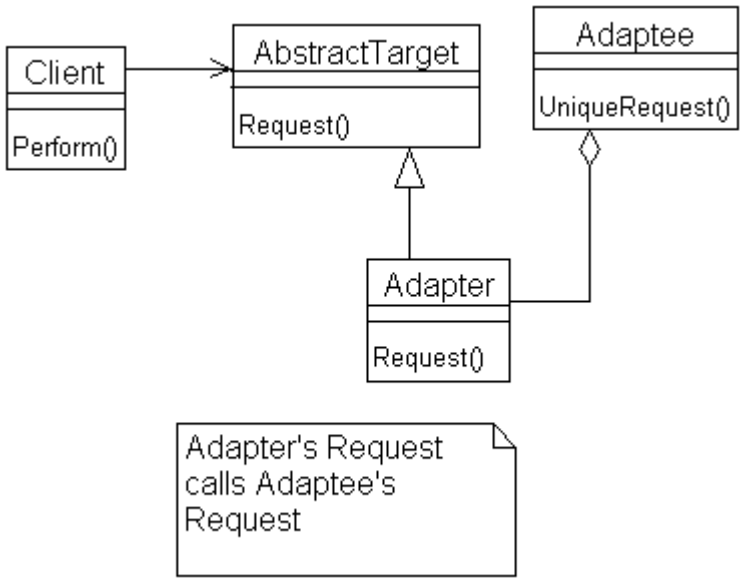


单例模式确保某一个类只有一个实例，而且自行实例化并向整个系统提供这个实例单例模式。单例模式只应在有真正的“单一实例”的需求时才可使用

1.2. 结构型模式

1.2.1. Adapter—适配器（变压器）模式

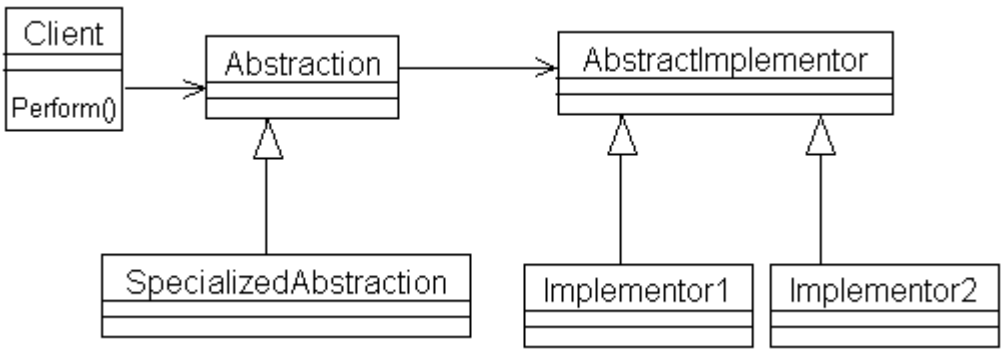
在朋友聚会上碰到了一个美女 Sarah，从香港来的，可我不会说粤语，她不会说普通话，只好求助于我的朋友 kent 了，他作为我和 Sarah 之间的 Adapter，让我和 Sarah 可以相互交谈了(也不知道他会不会耍我)



把一个类的接口变换成客户端所期待的另一种接口，从而使原本因接口原因不匹配而无法一起工作的两个类能够一起工作。适配类可以根据参数返还一个合适的实例给客户端。

1.2.2. Bridge—桥梁模式

早上碰到 MM，要说早上好，晚上碰到 MM，要说晚上好；碰到 MM 穿了件新衣服，要说你的衣服好漂亮哦，碰到 MM 新做的发型，要说你的头发好漂亮哦。不要问我“早上碰到 MM 新做了个发型怎么说”这种问题，自己用 BRIDGE 组合一下不就行了



将抽象化与实现化脱耦，使得二者可以独立的变化，也就是说将他们之间的强关联变成弱关联，也就是指在一个软件系统的抽象化和实现化之间使用组合/聚合关系而不是继承关系，从而使两者可以独立的变化。

1.2.3. Composite—合成模式

Mary 今天过生日。

“我过生日，你要送我一件礼物。”

“嗯，好吧，去商店，你自己挑。”

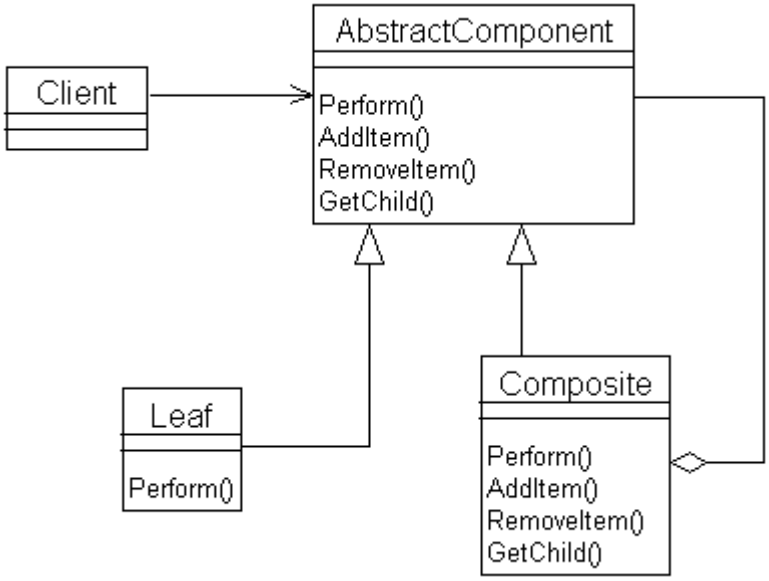
“这件 T 恤挺漂亮，买，这条裙子好看，买，这个包也不错，买。”

“喂，买了三件了呀，我只答应送一件礼物的哦。”

“什么呀，T 恤加裙子加包包，正好配成一套呀，小姐，麻烦你包起来。”

“……”,

MM 都会用 Composite 模式了，你会了没有？

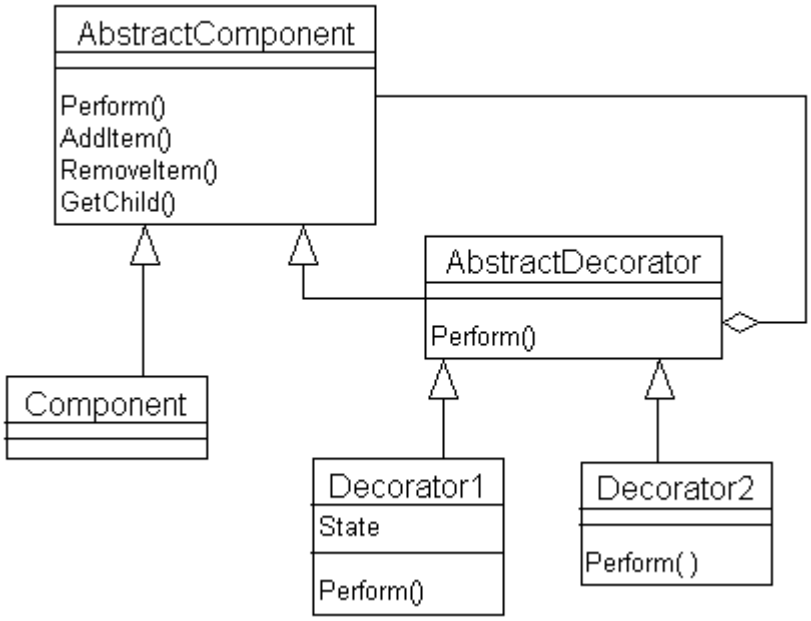


合成模式将对象组织到树结构中，可以用来描述整体与部分的关系。合成模式就是一个处理对象的树结构的模式。合成模式把部分与整体的关系用树结构表示出来。合成模式使得客户端把一个个单独的成分对象和由他们复合而成的合成对象同等看待。

1.2.4. Decorator—装饰模式

Mary 过完轮到 Sarly 过生日，还是不要叫她自己挑了，不然这个月伙食费肯定玩完，拿出我去年在华山顶上照的照片，在背面写上“最好的的礼物，就是爱你的 Fita”，再到街上礼品店买了个像框（卖礼品的 MM 也很漂亮哦），再找隔壁搞美术设计的 Mike

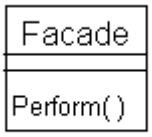
设计了一个漂亮的盒子装起来……，我们都是 Decorator，最终都在修饰我这个人呀，怎么样，看懂了吗？



装饰模式以对客户端透明的方式扩展对象的功能，是继承关系的一个替代方案，提供比继承更多的灵活性。动态给一个对象增加功能，这些功能可以再动态的撤消。增加由一些基本功能的排列组合而产生的非常大量的功能。

1.2.5. Facade—门面模式

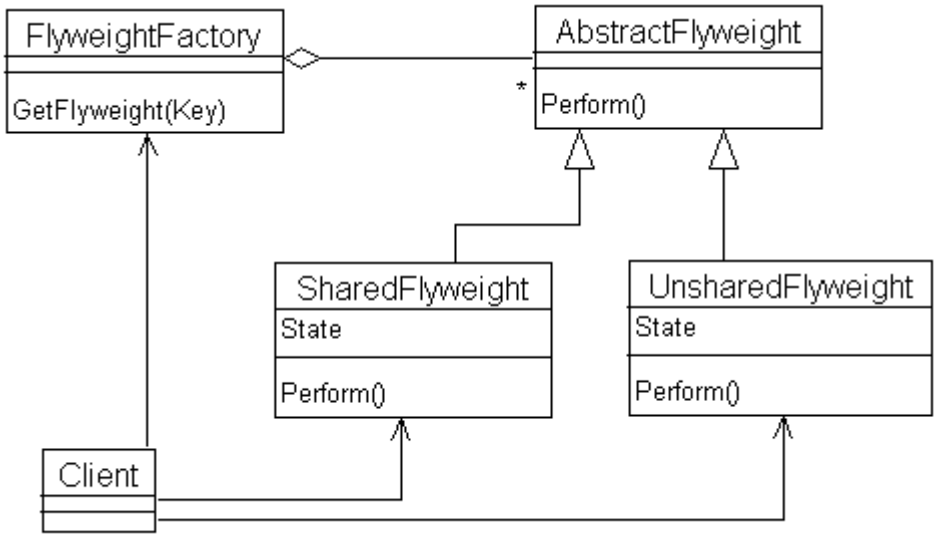
我有一个专业的 Nikon 相机，我就喜欢自己手动调光圈、快门，这样照出来的照片才专业，但 MM 可不懂这些，教了半天也不会。幸好相机有 Facade 设计模式，把相机调整到自动档，只要对准目标按快门就行了，一切由相机自动调整，这样 MM 也可以用这个相机给我拍张照片了。



外部与一个子系统的通信必须通过一个统一的门面对象进行。门面模式提供一个高层次的接口，使得子系统更易于使用。每一个子系统只有一个门面类，而且此门面类只有一个实例，也就是说它是一个单例模式。但整个系统可以有多个门面类。

1.2.6. Flyweight—享元模式

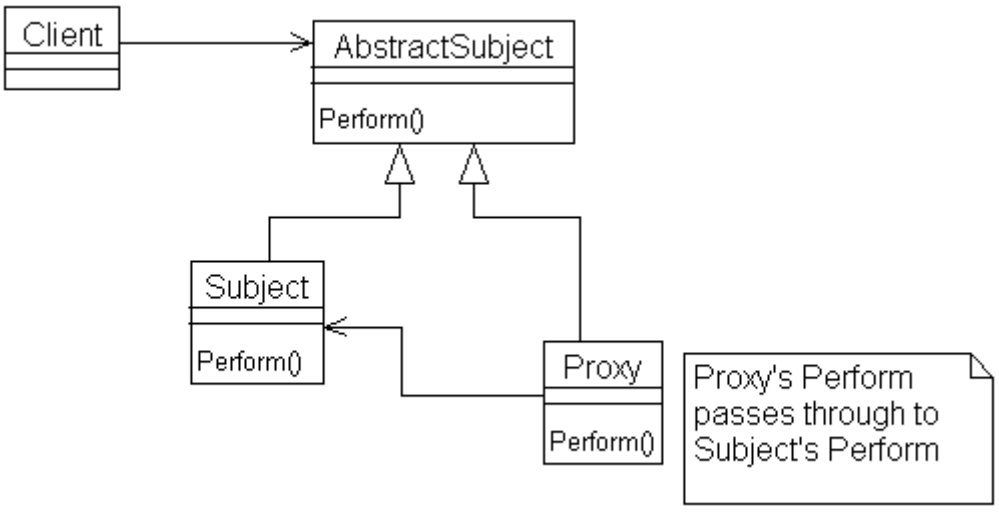
每天跟 MM 发短信，手指都累死了，最近买了个新手机，可以把一些常用的句子存在手机里，要用的时候，直接拿出来，在前面加上 MM 的名字就可以发送了，再不用一个字一个字敲了。共享的句子就是 Flyweight，MM 的名字就是提取出来的外部特征，根据上下文情况使用。



FLYWEIGHT 在拳击比赛中指最轻量级。享元模式以共享的方式高效的支持大量的细粒度对象。享元模式能做到共享的关键是区分内蕴状态和外蕴状态。内蕴状态存储在享元内部，不会随环境的改变而有所不同。外蕴状态是随环境的改变而改变的。外蕴状态不能影响内蕴状态，它们是相互独立的。将可以共享的状态和不可以共享的状态从常规类中区分开来，将不可以共享的状态从类里剔除出去。客户端不可以直接创建被共享的对象，而应当使用一个工厂对象负责创建被共享的对象。享元模式大幅度的降低内存中对象的数量。

1.2.7. Proxy—代理模式

跟 MM 在网上聊天，一开头总是“hi,你好”,“你从哪儿来呀？”“你多大了？”“身高多少呀？”这些话，真烦人，写个程序做为我的 Proxy 吧，凡是接收到这些话都设置好了自动的回答，接收到其他的话时再通知我回答，怎么样，酷吧。



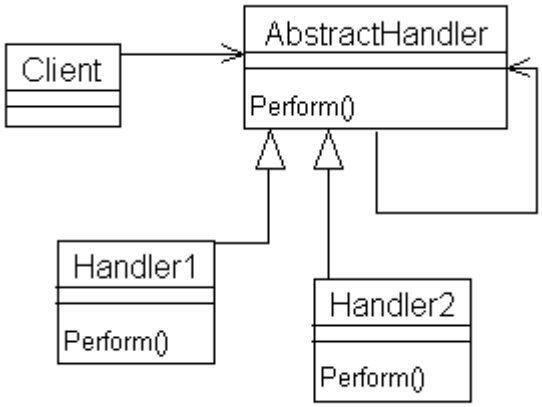
代理模式给某一个对象提供一个代理对象，并由代理对象控制对源对象的引用。代理就是一个人或一个机构代表另一个人或者一个机构采取行动。某些情况下，客户不想或者不能够直接引用一个对象，代理对象可以在客户和目标对象直接起到中介的作用。客户端分辨不出代理主题对象与真实主题对象。代理模式可以并不知道真正的被代理对象，而仅仅持有一个被代理对象的接口，这时候代理对象不能够创建被代理

对象，被代理对象必须有系统的其他角色代为创建并传入。

1.3. 行为型模式

1.3.1. Chain Of Responsibility—责任链模式

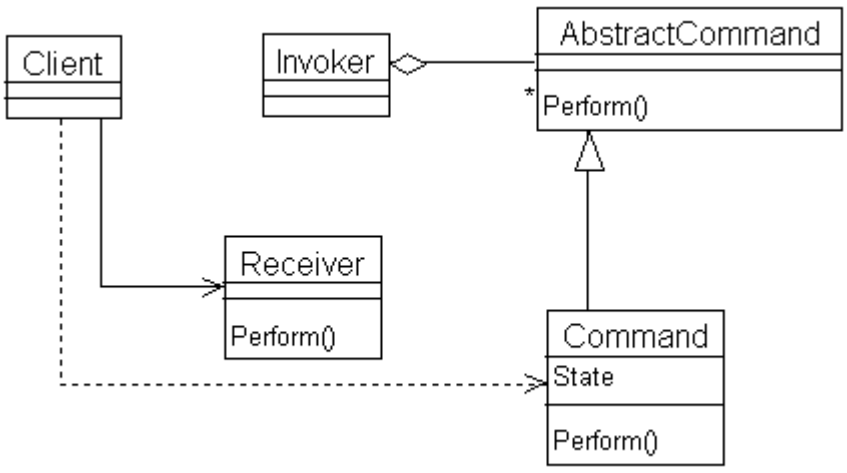
晚上去上英语课，为了好开溜坐到了最后一排，哇，前面坐了好几个漂亮的 MM 哎，找张纸条，写上“Hi,可以做我的女朋友吗？如果不愿意请向前传”，纸条就一个接一个的传上去了，糟糕，传到第一排的 MM 把纸条传给老师了，听说是个老处女呀，快跑!



在责任链模式中，很多对象由每一个对象对其下家的引用而接起来形成一条链。请求在这个链上传递，直到链上的某一个对象决定处理此请求。客户并不知道链上的哪一个对象最终处理这个请求，系统可以在不影响客户端的情况下动态的重新组织链和分配责任。处理者有两个选择：承担责任或者把责任推给下家。一个请求可以最终不被任何接收端对象所接受。

1.3.2. Command—命令模式

俺有一个 MM 家里管得特别严，没法见面，只好借助于她弟弟在我们俩之间传送信息，她对我有什么指示，就写一张纸条让她弟弟带给我。这不，她弟弟又传送过来一个 COMMAND，为了感谢他，我请他吃了碗杂酱面，哪知道他说：“我同时给我姐姐三个男朋友送 COMMAND，就数你最小气，才请我吃面。”，

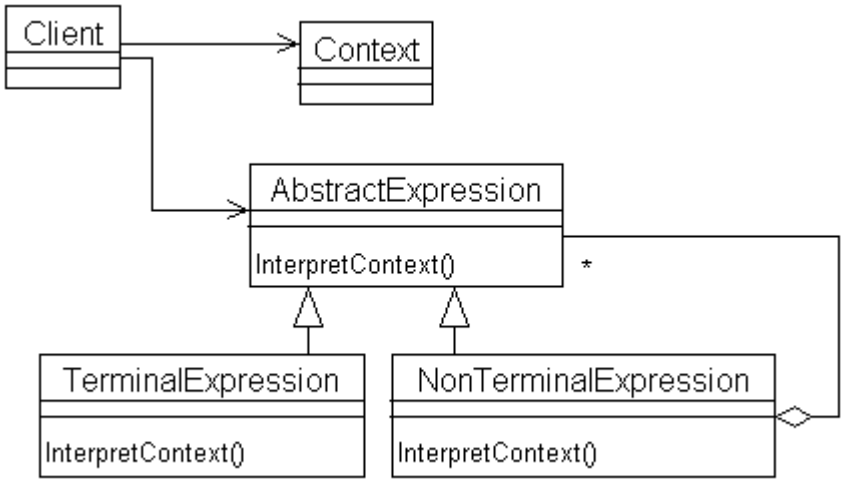


命令模式把一个请求或者操作封装到一个对象中。命令模式把发出命令的责任

和执行命令的责任分割开，委派给不同的对象。命令模式允许请求的一方和发送的一方独立开来，使得请求的一方不必知道接收请求的一方的接口，更不必知道请求是怎么被接收，以及操作是否执行，何时被执行以及是怎么被执行的。系统支持命令的撤消。

1.3.3. Interpreter—解释器模式

俺有一个《泡 MM 真经》，上面有各种泡 MM 的攻略，比如说去吃西餐的步骤、去看电影的方法等等，跟 MM 约会时，只要做一个 Interpreter，照着上面的脚本执行就可以了。



给定一个语言后，解释器模式可以定义出其文法的一种表示，并同时提供一个解释器。客户端可以使用这个解释器来解释这个语言中的句子。解释器模式将描述怎样在有了一个简单的文法后，使用模式设计解释这些语句。在解释器模式里面提到的语言是指任何解释器对象能够解释的任何组合。在解释器模式中需要定义一个代表文法的命令类的等级结构，也就是一系列的组合规则。每一个命令对象都有一个解释方法，代表对命令对象的解释。命令对象的等级结构中的对象的任何排列组合都是一个语言。

1.3.4. Iterator—迭代子模式

我爱上了 Mary，不顾一切的向她求婚。

Mary：“想要我跟你结婚，得答应我的条件”

我：“什么条件我都答应，你说吧”

Mary：“我看上了那个一克拉的钻石”

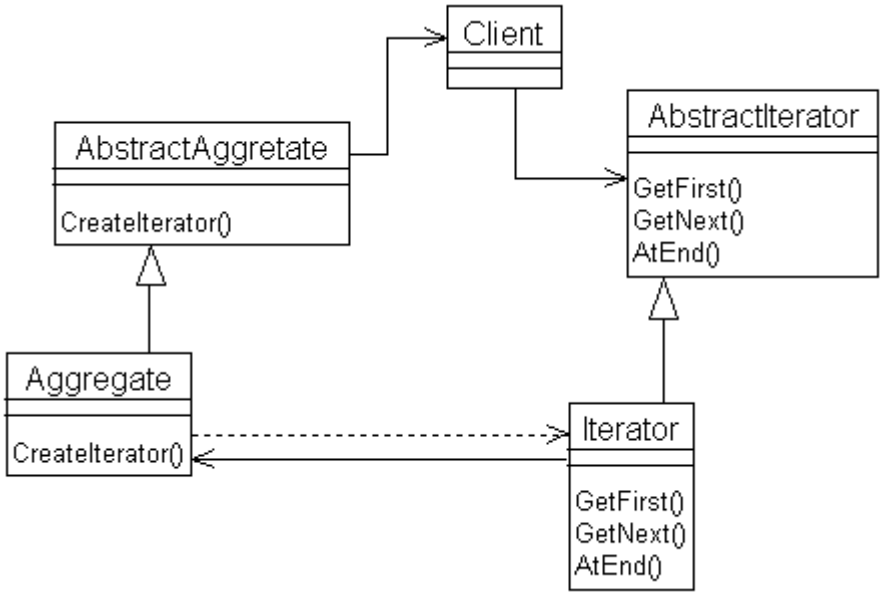
我：“我买，我买，还有吗？”

Mary：“我看上了湖边的那栋别墅”

我：“我买，我买，还有吗？”

Mary：“我看上那辆法拉利跑车”

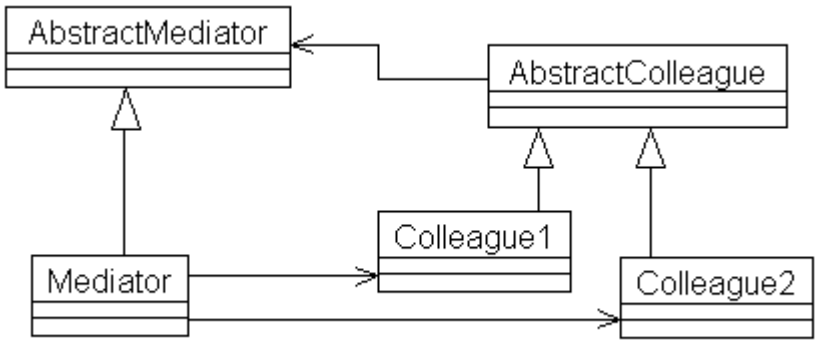
我脑袋嗡的一声，坐在椅子上，一咬牙：“我买，我买，还有吗？”



迭代子模式可以顺序访问一个聚集中的元素而不必暴露聚集的内部表象。多个对象聚在一起形成的总体称之为聚集，聚集对象是能够包容一组对象的容器对象。迭代子模式将迭代逻辑封装到一个独立的子对象中，从而与聚集本身隔开。迭代子模式简化了聚集的界面。每一个聚集对象都可以有一个或一个以上的迭代子对象，每一个迭代子的迭代状态可以是彼此独立的。迭代算法可以独立于聚集角色变化。

1.3.5. Mediator—调停者模式

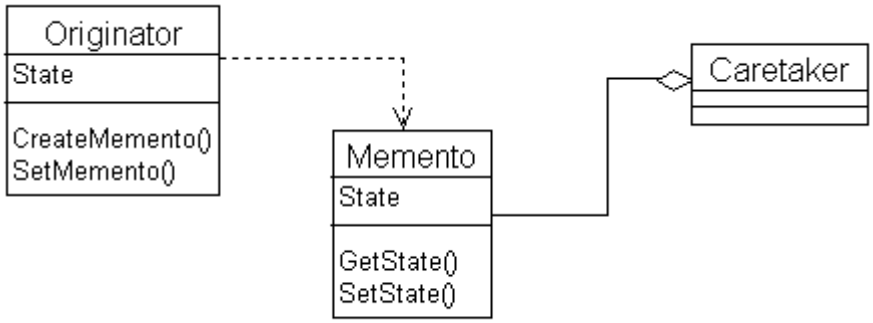
四个 MM 打麻将，相互之间谁应该给谁多少钱算不清楚了，幸亏当时我在旁边，按照各自的筹码数算钱，赚了钱的从我这里拿，赔了钱的也付给我，一切就 OK 啦，俺得到了四个 MM 的电话。



调停者模式包装了一系列对象相互作用的方式，使得这些对象不必相互明显作用。从而使他们可以松散偶合。当某些对象之间的作用发生改变时，不会立即影响其他的一些对象之间的作用。保证这些作用可以彼此独立的变化。调停者模式将多对多的相互作用转化为一对多的相互作用。调停者模式将对象的行为和协作抽象化，把对象在小尺度的行为上与其他对象的相互作用分开处理。

1.3.6. Memento—备忘录模式

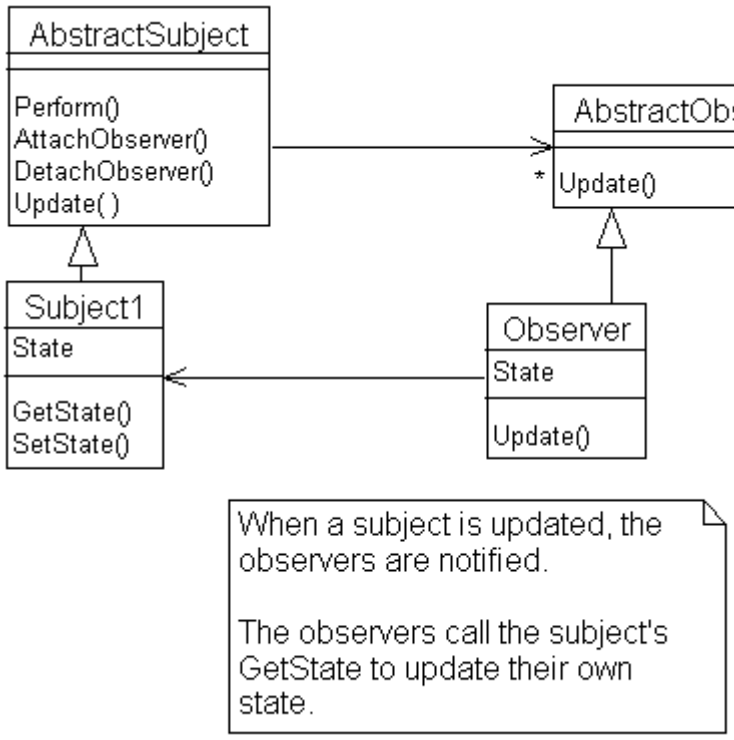
同时跟几个 MM 聊天时，一定要记清楚刚才跟 MM 说了些什么话，不然 MM 发现了会不高兴的哦，幸亏我有个备忘录，刚才与哪个 MM 说了什么话我都拷贝一份放到备忘录里面保存，这样可以随时察看以前的记录啦。



备忘录对象是一个用来存储另外一个对象内部状态的快照的对象。备忘录模式的用意是在不破坏封装的条件下，将一个对象的状态捉住，并外部化，存储起来，从而可以在将来合适的时候把这个对象还原到存储起来的状态。

1.3.7. Observer—观察者模式

想知道咱们公司最新 MM 情报吗？加入公司的 MM 情报邮件组就行了，tom 负责搜集情报，他发现的新情报不用一个一个通知我们，直接发布给邮件组，我们作为订阅者（观察者）就可以及时收到情报啦

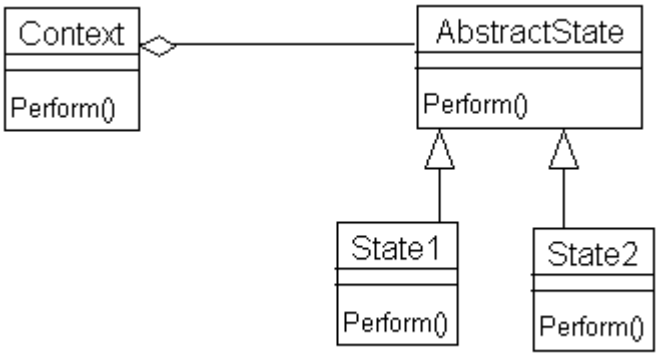


观察者模式定义了一种一队多的依赖关系，让多个观察者对象同时监听某一个主题对象。这个主题对象在状态上发生变化时，会通知所有观察者对象，使他们能够自动更新自己。

1.3.8. State—状态模式

跟 MM 交往时，一定要注意她的状态哦，在不同的状态时她的行为会有不同，比如你约她今天晚上去看电影，对你没兴趣的 MM 就会说“有事情啦”，对你不讨厌但还没喜欢上的 MM 就会说“好啊，不过可以带上我同事么？”，已经喜欢上你的 MM 就会说“几点钟？看完电影再去泡吧怎么样？”，当然你看电影过程中表现良好的话，

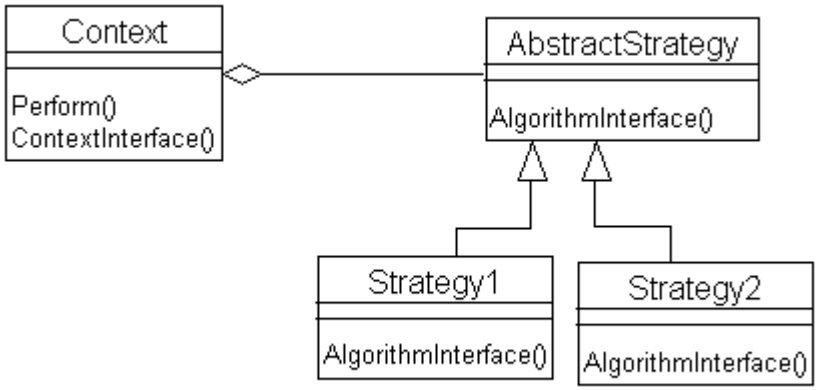
也可以把 MM 的状态从不讨厌不喜欢变成喜欢哦。



状态模式允许一个对象在其内部状态改变的时候改变行为。这个对象看上去象是改变了它的类一样。状态模式把所研究的对象的行为包装在不同的状态对象里，每一个状态对象都属于一个抽象状态类的一个子类。状态模式的意图是让一个对象在其内部状态改变的时候，其行为也随之改变。状态模式需要对每一个系统可能取得的状态创立一个状态类的子类。当系统的状态变化时，系统便改变所选的子类。

1.3.9. Strategy—策略模式

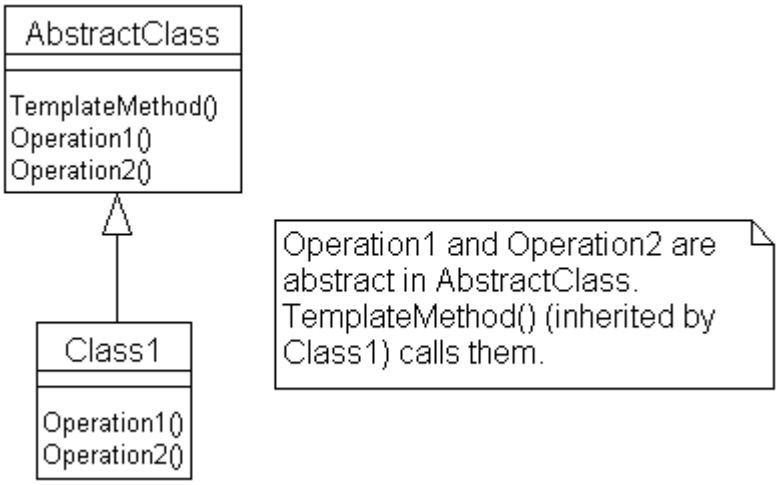
跟不同类型的 MM 约会，要用不同的策略，有的请电影比较好，有的则去吃小吃效果不错，有的去海边浪漫最合适，单目的都是为了得到 MM 的芳心，我的追 MM 锦囊中有好多 Strategy 哦。



策略模式针对一组算法，将每一个算法封装到具有共同接口的独立的类中，从而使得它们可以相互替换。策略模式使得算法可以在不影响到客户端的情况下发生变化。策略模式把行为和环境分开。环境类负责维持和查询行为类，各种算法在具体的策略类中提供。由于算法和环境独立开来，算法的增减，修改都不会影响到环境和客户端。

1.3.10. Template Method—模板方法模式

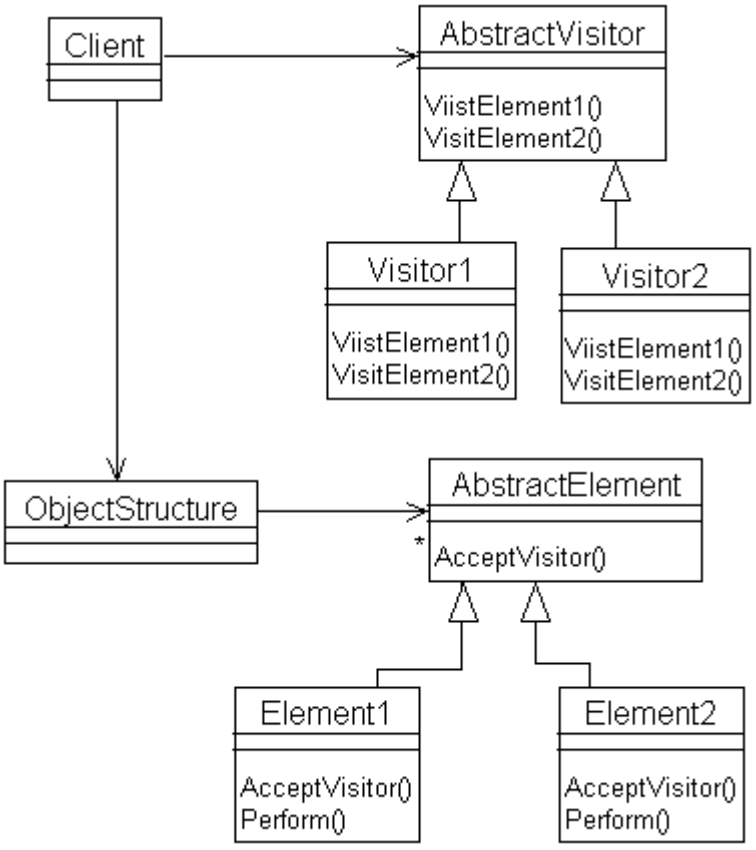
看过《如何说服女生上床》这部经典文章吗？女生从认识到上床的不变的步骤分为巧遇、打破僵局、展开追求、接吻、前戏、动手、爱抚、进去八大步骤(Template method)，但每个步骤针对不同的情况，都有不一样的做法，这就要看你随机应变啦(具体实现)；



模板方法模式准备一个抽象类，将部分逻辑以具体方法以及具体构造子的形式实现，然后声明一些抽象方法来迫使子类实现剩余的逻辑。不同的子类可以以不同的方式实现这些抽象方法，从而对剩余的逻辑有不同的实现。先制定一个顶级逻辑框架，而将逻辑的细节留给具体的子类去实现。

1.3.11. Visitor—访问者模式

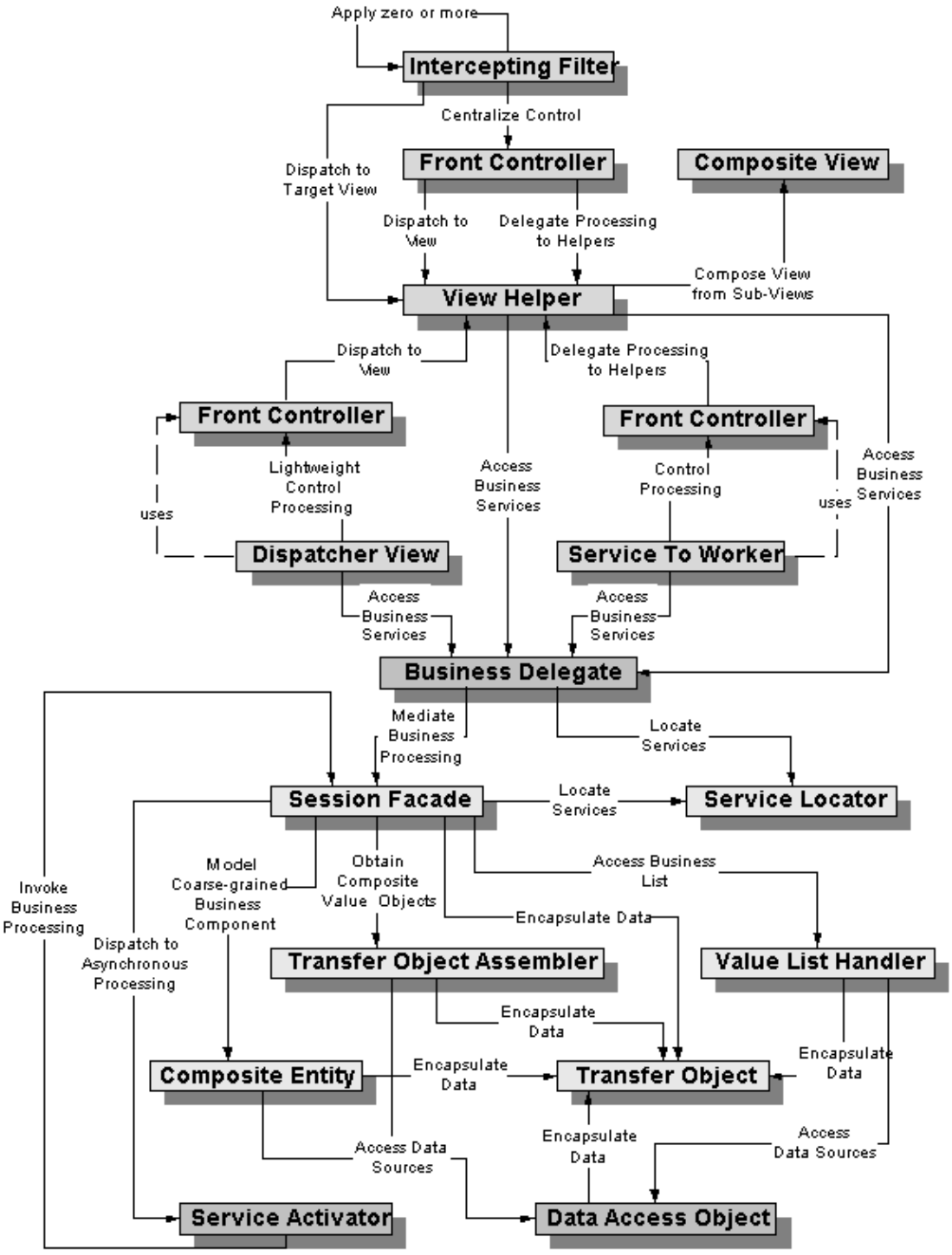
情人节到了，要给每个 MM 送一束鲜花和一张卡片，可是每个 MM 送的花都要针对她个人的特点，每张卡片也要根据个人的特点来挑，我一个人哪搞得清楚，还是找花店老板和礼品店老板做一下 Visitor，让花店老板根据 MM 的特点选一束花，让礼品店老板也根据每个人特点选一张卡，这样就轻松多了；



访问者模式的目的是封装一些施加于某种数据结构元素之上的操作。一旦这些

操作需要修改的话，接受这个操作的数据结构可以保持不变。访问者模式适用于数据结构相对未定的系统，它把数据结构和作用于结构上的操作之间的耦合解脱开，使得操作集合可以相对自由的演化。访问者模式使得增加新的操作变的很容易，就是增加一个新的访问者类。访问者模式将有关的行为集中到一个访问者对象中，而不是分散到一个个的节点类中。当使用访问者模式时，要将尽可能多的对象浏览逻辑放在访问者类中，而不是放到它的子类中。访问者模式可以跨过几个类的等级结构访问属于不同的等级结构的成员类。

2. J2EE 设计模式



2.1. 表示层模式

2.1.1. Intercepting Filter—拦截过滤器模式

Context

The presentation-tier request handling mechanism receives many different types of requests, which require varied types of processing. Some requests are simply forwarded to the appropriate handler component, while other requests must be modified, audited, or uncompressed before being further processed.

Problem

Preprocessing and post-processing of a client Web request and response are required.

When a request enters a Web application, it often must pass several entrance tests prior to the main processing stage. For example,

Has the client been authenticated?

Does the client have a valid session?

Is the client's IP address from a trusted network?

Does the request path violate any constraints?

What encoding does the client use to send the data?

Do we support the browser type of the client?

Some of these checks are tests, resulting in a yes or no answer that determines whether processing will continue. Other checks manipulate the incoming data stream into a form suitable for processing.

The classic solution consists of a series of conditional checks, with any failed check aborting the request. Nested if/else statements are a standard strategy, but this solution leads to code fragility and a copy-and-paste style of programming, because the flow of the filtering and the action of the filters is compiled into the application.

The key to solving this problem in a flexible and unobtrusive manner is to have a simple mechanism for adding and removing processing components, in which each component completes a specific filtering action.

Forces

Common processing, such as checking the data-encoding scheme or logging information about each request, completes per request.

Centralization of common logic is desired.

Services should be easy to add or remove unobtrusively without affecting existing components, so that they can be used in a variety of combinations, such as

Logging and authentication

Debugging and transformation of output for a specific client

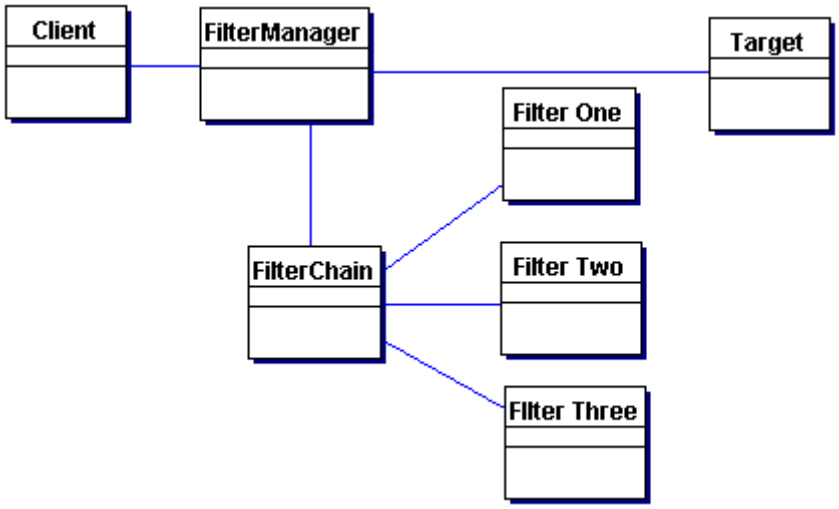
Uncompressing and converting encoding scheme of input

Solution

Create pluggable filters to process common services in a standard manner without requiring changes to core request processing code. The filters intercept incoming requests and outgoing responses, allowing preprocessing and post-processing. We are able to add and remove these filters unobtrusively, without requiring changes to our existing code.

We are able, in effect, to decorate our main processing with a variety of common services, such as security, logging, debugging, and so forth. These filters are components

that are independent of the main application code, and they may be added or removed declaratively. For example, a deployment configuration file may be modified to set up a chain of filters. The same configuration file might include a mapping of specific URLs to this filter chain. When a client requests a resource that matches this configured URL mapping, the filters in the chain are each processed in order before the requested target resource is invoked.



2.1.2. Front Controller—前端控制器模式

Context

The presentation-tier request handling mechanism must control and coordinate processing of each user across multiple requests. Such control mechanisms may be managed in either a centralized or decentralized manner.

Problem

The system requires a centralized access point for presentation-tier request handling to support the integration of system services, content retrieval, view management, and navigation. When the user accesses the view directly without going through a centralized mechanism, two problems may occur:

Each view is required to provide its own system services, often resulting in duplicate code.

View navigation is left to the views. This may result in commingled view content and view navigation.

Additionally, distributed control is more difficult to maintain, since changes will often need to be made in numerous places.

Forces

Common system services processing completes per request. For example, the security service completes authentication and authorization checks.

Logic that is best handled in one central location is instead replicated within numerous views.

Decision points exist with respect to the retrieval and manipulation of data.

Multiple views are used to respond to similar business requests.

A centralized point of contact for handling a request may be useful, for example, to control and log a user's progress through the site.

System services and view management logic are relatively sophisticated.

Solution

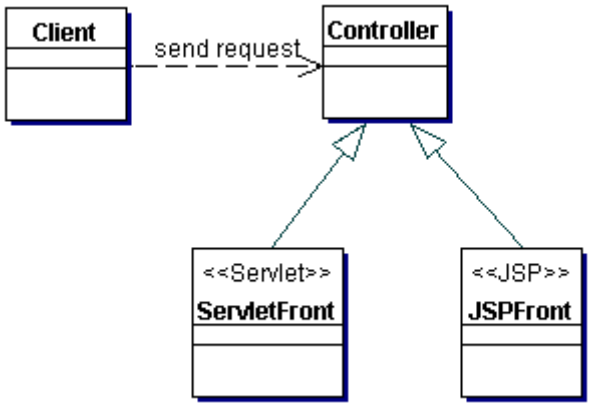
Use a controller as the initial point of contact for handling a request. The controller manages the handling of the request, including invoking security services such as authentication and authorization, delegating business processing, managing the choice of an appropriate view, handling errors, and managing the selection of content creation strategies.

The controller provides a centralized entry point that controls and manages Web request handling. By centralizing decision points and controls, the controller also helps reduce the amount of Java code, called *scriptlets*, embedded in the JavaServer Pages (JSP) page.

Centralizing control in the controller and reducing business logic in the view promotes code reuse across requests. It is a preferable approach to the alternative-embedding code in multiple views-because that approach may lead to a more error-prone, reuse-by-copy- and-paste environment.

Typically, a controller coordinates with a dispatcher component. Dispatchers are responsible for view management and navigation. Thus, a dispatcher chooses the next view for the user and vectors control to the resource. Dispatchers may be encapsulated within the controller directly or can be extracted into a separate component.

While the Front Controller pattern suggests centralizing the handling of all requests, it does not limit the number of handlers in the system, as does a Singleton. An application may use multiple controllers in a system, each mapping to a set of distinct services.



2.1.3. View Helper—视图助手模式

Context

The system creates presentation content, which requires processing of dynamic business data.

Problem

Presentation tier changes occur often and are difficult to develop and maintain when business data access logic and presentation formatting logic are interwoven. This makes the system less flexible, less reusable, and generally less resilient to change.

Intermingling the business and systems logic with the view processing reduces modularity and also provides a poor separation of roles among Web production and software development teams.

Forces

Business data assimilation requirements are nontrivial.

Embedding business logic in the view promotes a copy-and-paste type of reuse. This causes maintenance problems and bugs because a piece of logic is reused in the same or different view by simply duplicating it in the new location.

It is desirable to promote a clean separation of labor by having different individuals fulfill the roles of software developer and Web production team member.

One view is commonly used to respond to a particular business request.

Solution

A view contains formatting code, delegating its processing responsibilities to its helper classes, implemented as JavaBeans or custom tags. Helpers also store the view's intermediate data model and serve as business data adapters.

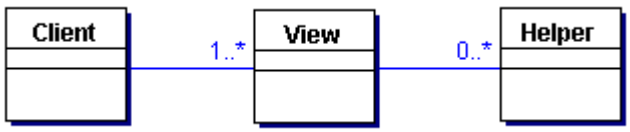
There are multiple strategies for implementing the view component. The JSP View Strategy suggests using a JSP as the view component. This is the preferred strategy, and it is the one most commonly used. The other principal strategy is the Servlet View Strategy, which utilizes a servlet as the view (see the section "Strategies" for more information).

Encapsulating business logic in a helper instead of a view makes our application more modular and facilitates component reuse. Multiple clients, such as controllers and views, may leverage the same helper to retrieve and adapt similar model state for presentation in multiple ways. The only way to reuse logic embedded in a view is by copying and pasting it elsewhere. Furthermore, copy-and-paste duplication makes a system harder to maintain, since the same bug potentially needs to be corrected in multiple places.

A signal that one may need to apply this pattern to existing code is when scriptlet code dominates the JSP view. The overriding goal when applying this pattern, then, is the partitioning of business logic outside of the view. While some logic is best encapsulated within helper objects, other logic is better placed in a centralized component that sits in front of the views and the helpers-this might include logic that is common across multiple requests, such as authentication checks or logging services, for example. Refer to the "Intercepting Filter" on page 4 and "Front Controller" on page 21 for more information on these issues.

If a separate controller is not employed in the architecture, or is not used to handle all requests, then the view component becomes the initial contact point for handling some requests. For certain requests, particularly those involving minimal processing, this scenario works fine. Typically, this situation occurs for pages that are based on static information, such as the first of a set of pages that will be served to a user to gather some information (see "Dispatcher View" on page 232). Additionally, this scenario occurs in some cases when a mechanism is employed to create composite pages (see "Composite View" on page 203).

The View Helper pattern focuses on recommending ways to partition your application responsibilities. For related discussions about issues dealing with directing client requests directly to a view, please refer to the section "Dispatcher View" on page 232.



2.1.4. Composite View—复合视图模式

Context

Sophisticated Web pages present content from numerous data sources, using multiple subviews that comprise a single display page. Additionally, a variety of individuals with different skill sets contribute to the development and maintenance of these Web pages.

Problem

Instead of providing a mechanism to combine modular, atomic portions of a view into a composite whole, pages are built by embedding formatting code directly within each view.

Modification to the layout of multiple views is difficult and error prone, due to the duplication of code.

Forces

Atomic portions of view content change frequently.

Multiple composite views use similar subviews, such as a customer inventory table. These atomic portions are decorated with different surrounding template text, or they appear in a different location within the page.

Layout changes are more difficult to manage and code harder to maintain when subviews are directly embedded and duplicated in multiple views.

Embedding frequently changing portions of template text directly into views also potentially affects the availability and administration of the system. The server may need to be restarted before clients see the modifications or updates to these template components.

Solution

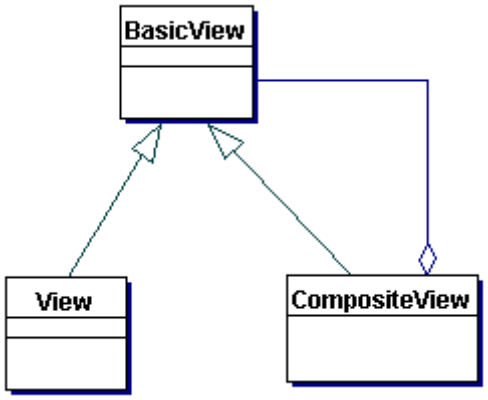
Use composite views that are composed of multiple atomic subviews. Each component of the template may be included dynamically into the whole and the layout of the page may be managed independently of the content.

This solution provides for the creation of a composite view based on the inclusion and substitution of modular dynamic and static template fragments. It promotes the reuse of atomic portions of the view by encouraging modular design. It is appropriate to use a composite view to generate pages containing display components that may be combined in a variety of ways. This scenario occurs, for example, with portal sites that include numerous independent subviews, such as news feeds, weather information, and stock quotes on a single page. The layout of the page is managed and modified independent of the subview content.

Another benefit of this pattern is that Web designers can prototype the layout of a site, plugging static content into each of the template regions. As site development progresses, the actual content is substituted for these placeholders.

This pattern is not without its drawbacks. There is a runtime overhead associated with it, a tradeoff for the increased flexibility that it provides. Also, the use of a more sophisticated layout mechanism brings with it some manageability and development issues, since there are more artifacts to maintain and a level of implementation indirection to

understand.



2.1.5. Service to Worker—工作者服务模式

Context

The system controls flow of execution and access to business data, from which it creates presentation content.

Note

The Service to Worker pattern, like the Dispatcher View pattern, describes a common combination of other patterns from the catalog. Both of these macro patterns describe the combination of a controller and dispatcher with views and helpers. While describing this common structure, they emphasize related but different usage patterns.

Problem

The problem is a combination of the problems solved by the Front Controller and View Helper patterns in the presentation tier. There is no centralized component for managing access control, content retrieval, or view management, and there is duplicate control code scattered throughout various views. Additionally, business logic and presentation formatting logic are intermingled within these views, making the system less flexible, less reusable, and generally less resilient to change.

Intermingling business logic with view processing also reduces modularity and provides a poor separation of roles among Web production and software development teams.

Forces

Authentication and authorization checks are completed per request.

Scriptlet code within views should be minimized.

Business logic should be encapsulated in components other than the view.

Control flow is relatively complex and based on values from dynamic content.

View management logic is relatively sophisticated, with multiple views potentially mapping to the same request.

Solution

Combine a controller and dispatcher with views and helpers (see "Front Controller" on page 172 and "View Helper" on page 186) to handle client requests and prepare a dynamic presentation as the response. Controllers delegate content retrieval to helpers,

which manage the population of the intermediate model for the view. A dispatcher is responsible for view management and navigation and can be encapsulated either within a controller or a separate component.

Service to Worker describes the combination of the Front Controller and View Helper patterns with a dispatcher component.

While this pattern and the Dispatcher View pattern describe a similar structure, the two patterns suggest a different division of labor among the components. In Service to Worker, the controller and the dispatcher have more responsibilities.

Since the Service to Worker and Dispatcher View patterns represent a common combination of other patterns from the catalog, each warrants its own name to promote efficient communication among developers. Unlike the Service to Worker pattern, the Dispatcher View pattern suggests deferring content retrieval to the time of view processing.

In the Dispatcher View pattern, the dispatcher typically plays a limited to moderate role in view management. In the Service to Worker pattern, the dispatcher typically plays a moderate to large role in view management.

A limited role for the dispatcher occurs when no outside resources are utilized in order to choose the view. The information encapsulated in the request is sufficient to determine the view to dispatch the request. For example,

http://some.server.com/servlet/Controller?next=login.jsp

The sole responsibility of the dispatcher component in this case is to dispatch to the view login.jsp.

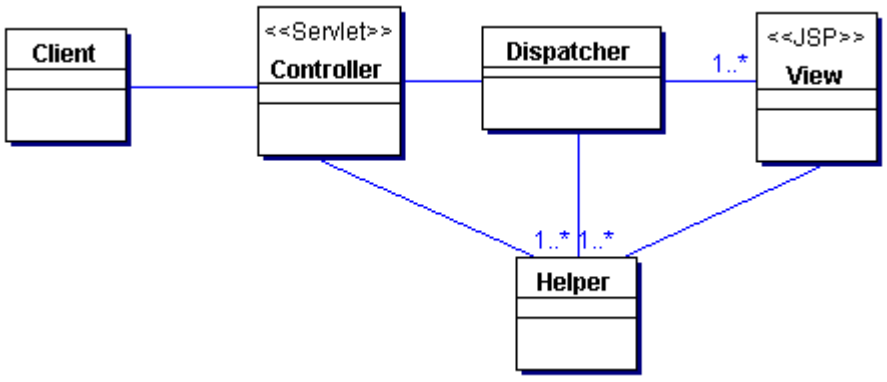
An example of the dispatcher playing a moderate role is the case where the client submits a request directly to a controller with a query parameter that describes an action to be completed:

http://some.server.com/servlet/Controller?action=login

The responsibility of the dispatcher component here is to translate the logical name login into the resource name of an appropriate view, such as login.jsp, and dispatch to that view. To accomplish this translation, the dispatcher may access resources such as an XML configuration file that specifies the appropriate view to display.

On the other hand, in the Service to Worker pattern, the dispatcher might be more sophisticated. The dispatcher may invoke a business service to determine the appropriate view to display.

The shared structure of Service to Worker and Dispatcher View consists of a controller working with a dispatcher, views, and helpers.



2.1.6. Dispatcher View—分发者视图模式

Context

System controls flow of execution and access to presentation processing, which is responsible for generating dynamic content.

Note

The Dispatcher View pattern, like the Service to Worker pattern, describes a common combination of other patterns from the catalog. Both of these macro patterns describe the combination of a controller and dispatcher with views and helpers. While describing this common structure, they emphasize related but different usage patterns.

Problem

The problem is a combination of the problems solved by the Front Controller and View Helper patterns in the presentation tier. There is no centralized component for managing access control, content retrieval or view management, and there is duplicate control code scattered throughout various views. Additionally, business logic and presentation formatting logic are intermingled within these views, making the system less flexible, less reusable, and generally less resilient to change.

Intermingling business logic with view processing also reduces modularity and provides a poor separation of roles among Web production and software development teams.

Forces

Authentication and authorization checks are completed per request.

Scriptlet code within views should be minimized.

Business logic should be encapsulated in components other than the view.

Control flow is relatively simple and is typically based on values encapsulated with the request.

View management logic is limited in complexity.

Solution

Combine a controller and dispatcher with views and helpers (see "Front Controller" on page 172 and "View Helper" on page 186) to handle client requests and prepare a dynamic presentation as the response. Controllers do not delegate content retrieval to helpers, because these activities are deferred to the time of view processing. A dispatcher is responsible for view management and navigation and can be encapsulated either within a controller, a view, or a separate component.

Dispatcher View describes the combination of the Front Controller and View Helper patterns with a dispatcher component. While this pattern and the Service to Worker pattern describe a similar structure, the two patterns suggest a different division of labor among the components. The controller and the dispatcher typically have limited responsibilities, as compared to the Service to Worker pattern, since the upfront processing and view management logic are basic. Furthermore, if centralized control of the underlying resources is considered unnecessary, then the controller is removed and the dispatcher may be moved into a view.

Since the Service to Worker and Dispatcher View patterns represent a common combination of other patterns from the catalog, each warrants its own name to promote

efficient communication among developers. Unlike the Service to Worker pattern, the Dispatcher View pattern suggests deferring content retrieval to the time of view processing.

In the Dispatcher View pattern, the dispatcher typically plays a limited to moderate role in view management. In the Service to Worker pattern, the dispatcher typically plays a moderate to large role in view management.

A limited role for the dispatcher occurs when no outside resources are utilized in order to choose the view. The information encapsulated in the request is sufficient to determine the view to dispatch the request. For example:

http://some.server.com/servlet/Controller?next=login.jsp

The sole responsibility of the dispatcher component in this case is to dispatch to the view login.jsp.

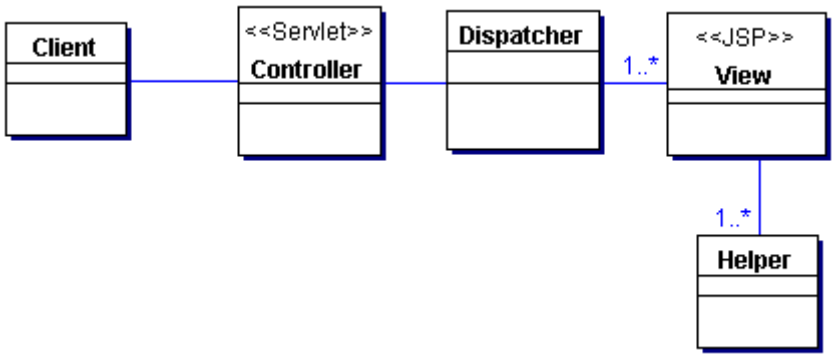
An example of the dispatcher playing a moderate role is the case where the client submits a request directly to a controller with a query parameter that describes an action to be completed:

http://some.server.com/servlet/Controller?action=login

The responsibility of the dispatcher component here is to translate the logical name login into the resource name of an appropriate view, such as login.jsp, and dispatch to that view. To accomplish this translation, the dispatcher may access resources such as an XML configuration file that specifies the appropriate view to display.

On the other hand, in the Service to Worker pattern, the dispatcher might be more sophisticated. The dispatcher may invoke a business service to determine the appropriate view to display.

The shared structure of these two patterns, as mentioned above, consists of a controller working with a dispatcher, views, and helpers.



2.2. 业务层模式

2.2.1. Business Delegate—业务委托模式

Context

A multi-tiered, distributed system requires remote method invocations to send and receive data across tiers. Clients are exposed to the complexity of dealing with distributed components.

Problem

Presentation-tier components interact directly with business services. This direct interaction exposes the underlying implementation details of the business service application program interface (API) to the presentation tier. As a result, the presentation-tier components are vulnerable to changes in the implementation of the business services: When the implementation of the business services change, the exposed implementation code in the presentation tier must change too.

Additionally, there may be a detrimental impact on network performance because presentation-tier components that use the business service API make too many invocations over the network. This happens when presentation-tier components use the underlying API directly, with no client-side caching mechanism or aggregating service.

Lastly, exposing the service APIs directly to the client forces the client to deal with the networking issues associated with the distributed nature of Enterprise JavaBeans (EJB) technology.

Forces

Presentation-tier clients need access to business services.

Different clients, such as devices, Web clients, and thick clients, need access to business service.

Business services APIs may change as business requirements evolve.

It is desirable to minimize coupling between presentation-tier clients and the business service, thus hiding the underlying implementation details of the service, such as lookup and access.

Clients may need to implement caching mechanisms for business service information.

It is desirable to reduce network traffic between client and business services.

Solution

Use a Business Delegate to reduce coupling between presentation-tier clients and business services. The Business Delegate hides the underlying implementation details of the business service, such as lookup and access details of the EJB architecture.

The Business Delegate acts as a client-side business abstraction; it provides an abstraction for, and thus hides, the implementation of the business services. Using a Business Delegate reduces the coupling between presentation-tier clients and the system's business services. Depending on the implementation strategy, the Business Delegate may shield clients from possible volatility in the implementation of the business service API. Potentially, this reduces the number of changes that must be made to the presentation-tier client code when the business service API or its underlying implementation changes.

However, interface methods in the Business Delegate may still require modification if the underlying business service API changes. Admittedly, though, it is more likely that changes will be made to the business service rather than to the Business Delegate.

Often, developers are skeptical when a design goal such as abstracting the business layer causes additional upfront work in return for future gains. However, using this pattern or its strategies results in only a small amount of additional upfront work and provides considerable benefits. The main benefit is hiding the details of the underlying service. For example, the client can become transparent to naming and lookup services. The Business Delegate also handles the exceptions from the business services, such as java.rmi.Remote exceptions, Java Messages Service (JMS) exceptions and so on. The Business Delegate may

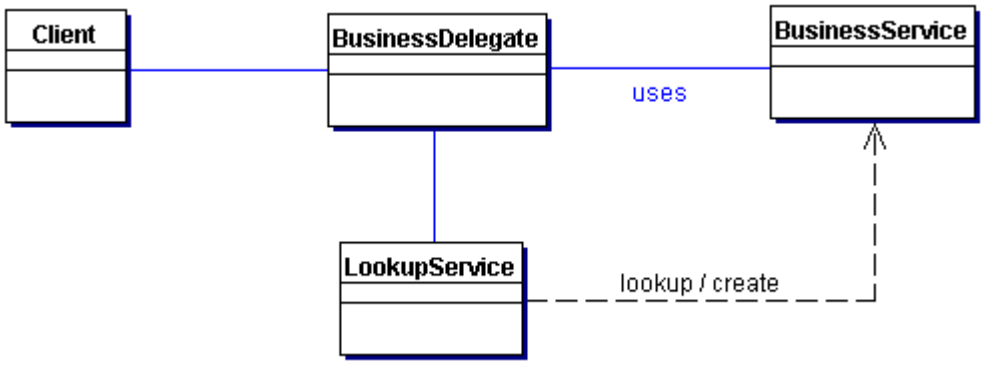
intercept such service level exceptions and generate application level exceptions instead. Application level exceptions are easier to handle by the clients, and may be user friendly. The Business Delegate may also transparently perform any retry or recovery operations necessary in the event of a service failure without exposing the client to the problem until it is determined that the problem is not resolvable. These gains present a compelling reason to use the pattern.

Another benefit is that the delegate may cache results and references to remote business services. Caching can significantly improve performance, because it limits unnecessary and potentially costly round trips over the network.

A Business Delegate uses a component called the Lookup Service. The Lookup Service is responsible for hiding the underlying implementation details of the business service lookup code. The Lookup Service may be written as part of the Delegate, but we recommend that it be implemented as a separate component, as outlined in the Service Locator pattern (See "Service Locator" on page 368.)

When the Business Delegate is used with a Session Facade, typically there is a one-to-one relationship between the two. This one-to-one relationship exists because logic that might have been encapsulated in a Business Delegate relating to its interaction with multiple business services (creating a one-to-many relationship) will often be factored back into a Session Facade.

Finally, it should be noted that this pattern could be used to reduce coupling between other tiers, not simply the presentation and the business tiers.



2.2.2. Transfer Object—传输对象模式

Context

Application clients need to exchange data with enterprise beans.

Problem

Java 2 Platform, Enterprise Edition (J2EE) applications implement server-side business components as session beans and entity beans. Some methods exposed by the business components return data to the client. Often, the client invokes a business object's get methods multiple times until it obtains all the attribute values.

Session beans represent the business services and are not shared between users. A session bean provides coarse-grained service methods when implemented per the Session Facade pattern.

Entity beans, on the other hand, are multiuser, transactional objects representing

persistent data. An entity bean exposes the values of attributes by providing an accessor method (also referred to as a *getter* or *get method*) for each attribute it wishes to expose.

Every method call made to the business service object, be it an entity bean or a session bean, is potentially remote. Thus, in an Enterprise JavaBeans (EJB) application such remote invocations use the network layer regardless of the proximity of the client to the bean, creating a network overhead. Enterprise bean method calls may permeate the network layers of the system even if the client and the EJB container holding the entity bean are both running in the same JVM, OS, or physical machine. Some vendors may implement mechanisms to reduce this overhead by using a more direct access approach and bypassing the network.

As the usage of these remote methods increases, application performance can significantly degrade. Therefore, using multiple calls to get methods that return single attribute values is inefficient for obtaining data values from an enterprise bean.

Forces

All access to an enterprise bean is performed via remote interfaces to the bean. Every call to an enterprise bean is potentially a remote method call with network overhead.

Typically, applications have a greater frequency of read transactions than update transactions. The client requires the data from the business tier for presentation, display, and other read-only types of processing. The client updates the data in the business tier much less frequently than it reads the data.

The client usually requires values for more than one attribute or dependent object from an enterprise bean. Thus, the client may invoke multiple remote calls to obtain the required data.

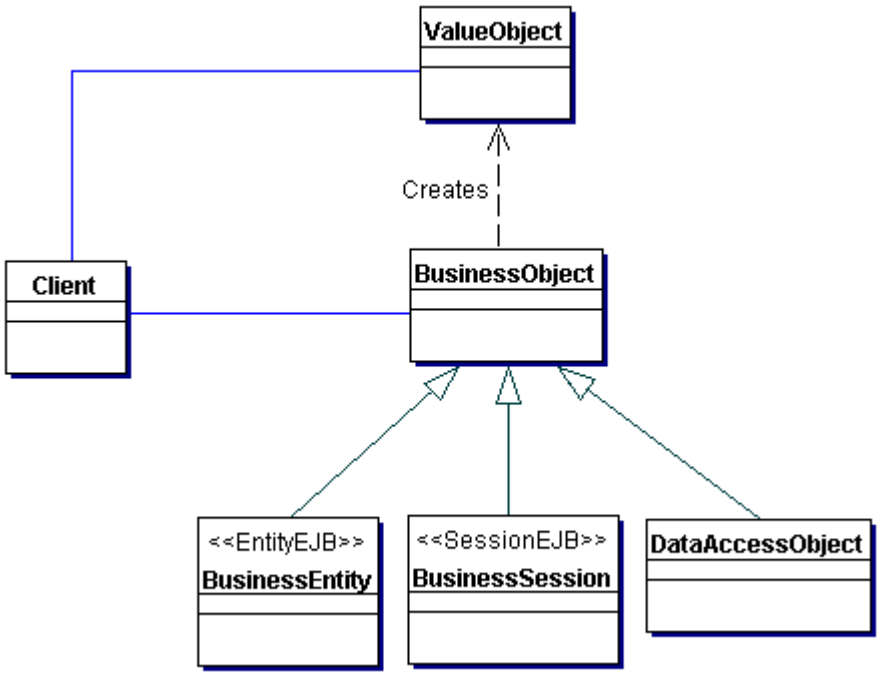
The number of calls made by the client to the enterprise bean impacts network performance. Chattier applications-those with increased traffic between client and server tiers-often degrade network performance.

Solution

Use a Transfer Object to encapsulate the business data. A single method call is used to send and retrieve the Transfer Object. When the client requests the enterprise bean for the business data, the enterprise bean can construct the Transfer Object, populate it with its attribute values, and pass it by value to the client.

Clients usually require more than one value from an enterprise bean. To reduce the number of remote calls and to avoid the associated overhead, it is best to use Transfer Objects to transport the data from the enterprise bean to its client.

When an enterprise bean uses a Transfer Object, the client makes a single remote method invocation to the enterprise bean to request the Transfer Object instead of numerous remote method calls to get individual attribute values. The enterprise bean then constructs a new Transfer Object instance, copies values into the object and returns it to the client. The client receives the Transfer Object and can then invoke accessor (or getter) methods on the Transfer Object to get the individual attribute values from the Transfer Object. Or, the implementation of the Transfer Object may be such that it makes all attributes public. Because the Transfer Object is passed by value to the client, all calls to the Transfer Object instance are local calls instead of remote method invocations.



2.2.3. Session Facade—会话门面模式

Context

Enterprise beans encapsulate business logic and business data and expose their interfaces, and thus the complexity of the distributed services, to the client tier.

Problem

In a multitiered Java 2 Platform, Enterprise Edition (J2EE) application environment, the following problems arise:

Tight coupling, which leads to direct dependence between clients and business objects;

Too many method invocations between client and server, leading to network performance problems;

Lack of a uniform client access strategy, exposing business objects to misuse.

A multitiered J2EE application has numerous server-side objects that are implemented as enterprise beans. In addition, some other arbitrary objects may provide services, data, or both. These objects are collectively referred to as business objects, since they encapsulate business data and business logic.

J2EE applications implement business objects that provide processing services as session beans. Coarse-grained business objects that represent an object view of persistent storage and are shared by multiple users are usually implemented as entity beans.

Application clients need access to business objects to fulfill their responsibilities and to meet user requirements. Clients can directly interact with these business objects because they expose their interfaces. When you expose business objects to the client, the client must understand and be responsible for the business data object relationships, and must be able to handle business process flow.

However, direct interaction between the client and the business objects leads to tight coupling between the two, and such tight coupling makes the client directly dependent on

the implementation of the business objects. Direct dependence means that the client must represent and implement the complex interactions regarding business object lookups and creations, and must manage the relationships between the participating business objects as well as understand the responsibility of transaction demarcation.

As client requirements increase, the complexity of interaction between various business objects increases. The client grows larger and more complex to fulfill these requirements. The client becomes very susceptible to changes in the business object layer; in addition, the client is unnecessarily exposed to the underlying complexity of the system.

Tight coupling between objects also results when objects manage their relationship within themselves. Often, it is not clear where the relationship is managed. This leads to complex relationships between business objects and rigidity in the application. Such lack of flexibility makes the application less manageable when changes are required.

When accessing the enterprise beans, clients interact with remote objects. Network performance problems may result if the client directly interacts with all the participating business objects. When invoking enterprise beans, every client invocation is potentially a remote method call. Each access to the business object is relatively fine-grained. As the number of participants increases in a scenario, the number of such remote method calls increases. As the number of remote method calls increases, the chattiness between the client and the server-side business objects increases. This may result in network performance degradation for the application, because the high volume of remote method calls increases the amount of interaction across the network layer.

A problem also arises when a client interacts directly with the business objects. Since the business objects are directly exposed to the clients, there is no unified strategy for accessing the business objects. Without such a uniform client access strategy, the business objects are exposed to clients and may reduce consistent usage.

Forces

Provide a simpler interface to the clients by hiding all the complex interactions between business components.

Reduce the number of business objects that are exposed to the client across the service layer over the network.

Hide from the client the underlying interactions and interdependencies between business components. This provides better manageability, centralization of interactions (responsibility), greater flexibility, and greater ability to cope with changes.

Provide a uniform coarse-grained service layer to separate business object implementation from business service abstraction.

Avoid exposing the underlying business objects directly to the client to keep tight coupling between the two tiers to a minimum.

Solution

Use a session bean as a facade to encapsulate the complexity of interactions between the business objects participating in a workflow. The Session Facade manages the business objects, and provides a uniform coarse-grained service access layer to clients.

The Session Facade abstracts the underlying business object interactions and provides a service layer that exposes only the required interfaces. Thus, it hides from the client's view the complex interactions between the participants. The Session Facade manages the

interactions between the business data and business service objects that participate in the workflow, and it encapsulates the business logic associated with the requirements. Thus, the session bean (representing the Session Facade) manages the relationships between business objects. The session bean also manages the life cycle of these participants by creating, locating (looking up), modifying, and deleting them as required by the workflow. In a complex application, the Session Facade may delegate this lifestyle management to a separate object. For example, to manage the lifestyle of participant session and entity beans, the Session Facade may delegate that work to a Service Locator object (see "Service Locator" on page 368).

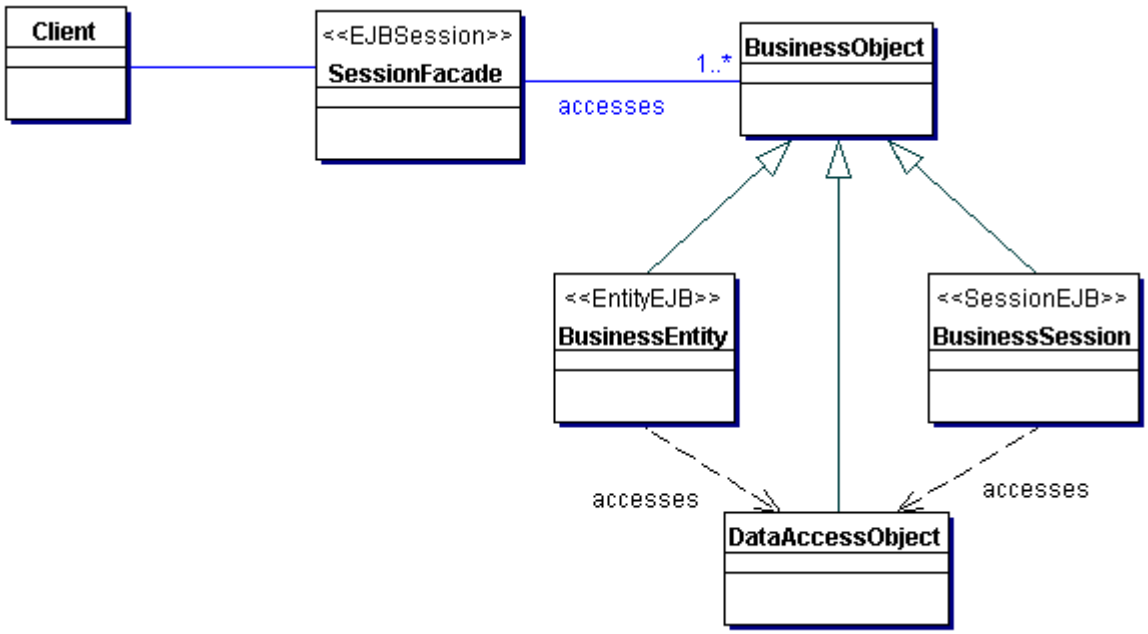
It is important to examine the relationship between business objects. Some relationships between business objects are transient, which means that the relationship is applicable to only that interaction or scenario. Other relationships may be more permanent. Transient relationships are best modeled as workflow in a facade, where the facade manages the relationships between the business objects. Permanent relationships between two business objects should be studied to determine which business object (if not both objects) maintains the relationship.

Use Cases and Session Facades

So, how do you identify the Session Facades through studying use cases? Mapping every use case to a Session Facade will result in too many Session Facades. This defeats the intention of having fewer coarse-grained session beans. Instead, as you derive the Session Facades during your modeling, look to consolidate them into fewer numbers of session beans based on some logical partitioning.

For example, for a banking application, you may group the interactions related to managing an account into a single facade. The use cases Create New Account, Change Account Information, View Account information, and so on all deal with the coarse-grained entity object Account. Creating a session bean facade for each use case is not recommended. Thus, the functions required to support these related use cases could be grouped into a single Session Facade called AccountSessionFacade.

In this case, the Session Facade will become a highly coarse-grained controller with high-level methods that can facilitate each interaction (that is, createNewAccount, changeAccount, getAccount). Therefore, we recommend that you design Session Facades to aggregate a group of the related interactions into a single Session Facade. This results in fewer Session Facades for the application, and leverages the benefits of the Session Facade pattern.



2.2.4. Composite Entity—复合实体模式

Context

Entity beans are not intended to represent every persistent object in the object model. Entity beans are better suited for coarse-grained persistent business objects.

Problem

In a Java 2 Platform, Enterprise Edition (J2EE) application, clients -- such as applications, JavaServer Pages (JSP) pages, servlets, JavaBeans components -- access entity beans via their remote interfaces. Thus, every client invocation potentially routes through network stubs and skeletons, even if the client and the enterprise bean are in the same JVM, OS, or machine. When entity beans are fine-grained objects, clients tend to invoke more individual entity bean methods, resulting in high network overhead.

Entity beans represent distributed persistent business objects. Whether developing or migrating an application to the J2EE platform, object granularity is very important when deciding what to implement as an entity bean. Entity beans should represent coarse-grained business objects, such as those that provide complex behavior beyond simply getting and setting field values. These coarse-grained objects typically have dependent objects. A dependent object is an object that has no real domain meaning when not associated with its coarse-grained parent.

A recurring problem is the direct mapping of the object model to an Enterprise JavaBeans (EJB) model (specifically entity beans). This creates a relationship between the entity bean objects without consideration of coarse-grained versus fine-grained (or dependent) objects. Determining what to make coarse-grained versus fine-grained is typically difficult and can best be done via modeling relationships in Unified Modeling Language (UML) models.

There are a number of areas impacted by the fine-grained entity bean design approach:

Entity Relationships - Directly mapping an object model to an EJB model does not

take into account the impact of relationships between the objects. The inter-object relationships are directly transformed into inter-entity bean relationships. As a result, an entity bean might contain or hold a remote reference to another entity bean. However, maintaining remote references to distributed objects involves different techniques and semantics than maintaining references to local objects. Besides increasing the complexity of the code, it reduces flexibility, because the entity bean must change if there are any changes in its relationships.

Also, there is no guarantee as to the validity of the entity bean references to other entity beans over time. Such references are established dynamically using the entity's home object and the primary key for that entity bean instance. This implies a high maintenance overhead of reference validity checking for each such entity-bean-to-entity-bean reference.

Manageability - Implementing fine-grained objects as entity beans results in a large number of entity beans in the system. An entity bean is defined using several classes. For each entity bean component, the developer must provide classes for the home interface, the remote interface, the bean implementation, and the primary key.

In addition, the container may generate classes to support the entity bean implementation. When the bean is created, these classes are realized as real objects in the container. In short, the container creates a number of objects to support each entity bean instance. Large numbers of entity beans result in more classes and code to maintain for the development team. It also results in a large number of objects in the container. This can negatively impact the application performance.

Network Performance - Fine-grained entity beans potentially have more inter-entity bean relationships. Entity beans are distributed objects. When one entity bean invokes a method on another entity bean, the call is potentially treated as a remote call by the container, even if both entity beans are in the same container or JVM. If the number of entity-bean-to-entity-bean relationships increases, then this decreases system scalability due to heavy network overhead.

Database Schema Dependency - When the entity beans are fine-grained, each entity bean instance usually represents a single row in a database. This is not a proper application of the entity bean design, since entity beans are more suitable for coarse-grained components. Fine-grained entity bean implementation typically is a direct representation of the underlying database schema in the entity bean design. When clients use these fine-grained entity beans, they are essentially operating at the row level in the database, since each entity bean is effectively a single row. Because the entity bean directly models a single database row, the clients become dependent on the database schema. When the schema changes, the entity bean definitions must change as well. Further, since the clients are operating at the same granularity, they must observe and react to this change. This schema dependency causes a loss of flexibility and increases the maintenance overhead whenever schema changes are required.

Object Granularity (Coarse-Grained versus Fine-Grained) - Object granularity impacts data transfer between the enterprise bean and the client. In most applications, clients typically need a larger chunk of data than one or two rows from a table. In such a case, implementing each of these fine-grained objects as an entity bean means that the client would have to manage the relationships between all these fine-grained objects. Depending

on the data requirements, the client might have to perform many lookups of a number of entity beans to obtain the required information.

Forces

Entity beans are best implemented as coarse-grained objects due to the high overhead associated with each entity bean. Each entity bean is implemented using several objects, such as EJB home object, remote object, bean implementation, and primary key, and each is managed by the container services.

Applications that directly map relational database schema to entity beans (where each row in a table is represented by an entity bean instance) tend to have a large number of fine-grained entity beans. It is desirable to keep the entity beans coarse-grained and reduce the number of entity beans in the application.

Direct mapping of object model to EJB model yields fine-grained entity beans. Fine-grained entity beans usually map to the database schema. This entity-to-database row mapping causes problems related to performance, manageability, security, and transaction handling. Relationships between tables are implemented as relationships between entity beans, which means that entity beans hold references to other entity beans to implement the fine-grained relationships. It is very expensive to manage inter-entity bean relationships, because these relationships must be established dynamically, using the entity home objects and the enterprise beans' primary keys.

Clients do not need to know the implementation of the database schema to use and support the entity beans. With fine-grained entity beans, the mapping is usually done so that each entity bean instance maps to a single row in the database. This fine-grained mapping creates a dependency between the client and the underlying database schema, since the clients deal with the fine-grained beans and they are essentially a direct representation of the underlying schema. This results in tight coupling between the database schema and entity beans. A change to the schema causes a corresponding change to the entity bean, and in addition requires a corresponding change to the clients.

There is an increase in chattiness of applications due to intercommunication among fine-grained entity beans. Excessive inter-entity bean communication often leads to a performance bottleneck. Every method call to the entity bean is made via the network layer, even if the caller is in the same address space as the called bean (that is, both the client, or caller entity bean, and the called entity bean are in the same container). While some container vendors optimize for this scenario, the developer cannot rely on this optimization in all containers.

Additional chattiness can be observed between the client and the entity beans because the client may have to communicate with many fine-grained entity beans to fulfill a requirement. It is desirable to reduce the communication between or among entity beans and to reduce the chattiness between the client and the entity bean layer.

Solution

Use Composite Entity to model, represent, and manage a set of interrelated persistent objects rather than representing them as individual fine-grained entity beans. A Composite Entity bean represents a graph of objects.

In order to understand this solution, let us first define what is meant by persistent objects and discuss their relationships.

A persistent object is an object that is stored in some type of data store. Multiple clients usually share persistent objects. Persistent objects can be classified into two types: coarse-grained objects and dependent objects.

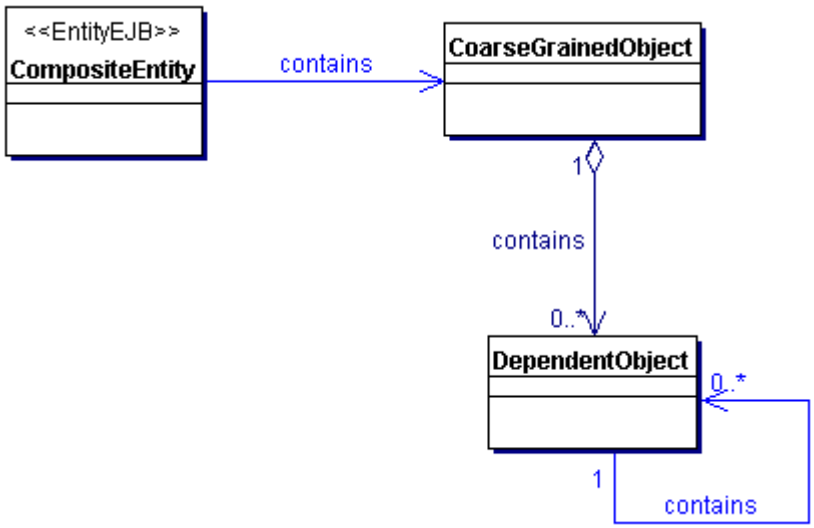
A coarse-grained object is self-sufficient. It has its own life cycle and manages its relationships to other objects. Each coarse-grained object may reference or contain one or more other objects. The coarse-grained object usually manages the lifecycles of these objects. Hence, these objects are called dependent objects. A dependent object can be a simple self-contained object or may in turn contain other dependent objects.

The life cycle of a dependent object is tightly coupled to the life cycle of the coarse-grained object. A client may only indirectly access a dependent object through the coarse-grained object. That is, dependent objects are not directly exposed to clients because their parent (coarse-grained) object manages them. Dependent objects cannot exist by themselves. Instead, they always need to have their coarse-grained (or parent) object to justify their existence.

Typically, you can view the relationship between a coarse-grained object and its dependent objects as a tree. The coarse-grained object is the root of the tree (the root node). Each dependent object can be a standalone dependent object (a leaf node) that is a child of the coarse-grained object. Or, the dependent object can have parent-child relationships with other dependent objects, in which case it is considered a branch node.

A Composite Entity bean can represent a coarse-grained object and all its related dependent objects. Aggregation combines interrelated persistent objects into a single entity bean, thus drastically reducing the number of entity beans required by the application. This leads to a highly coarse-grained entity bean that can better leverage the benefits of entity beans than can fine-grained entity beans.

Without the Composite Entity approach, there is a tendency to view each coarse-grained and dependent object as a separate entity bean, leading to a large number of entity beans.



2.2.5. Transfer Object Assembler—传输对象组装器模式

Context

In a Java 2 Platform, Enterprise Edition (J2EE) application, the server-side business components are implemented using session beans, entity beans, DAOs, and so forth. Application clients frequently need to access data that is composed from multiple objects.

Problem

Application clients typically require the data for the model or parts of the model to present to the user or to use for an intermediate processing step before providing some service. The application model is an abstraction of the business data and business logic implemented on the server side as business components. A model may be expressed as a collection of objects put together in a structured manner (tree or graph). In a J2EE application, the model is a distributed collection of objects such as session beans, entity beans, or DAOs and other objects. For a client to obtain the data for the model, such as to display to the user or to perform some processing, it must access individually each distributed object that defines the model. This approach has several drawbacks:

Because the client must access each distributed component individually, there is a tight coupling between the client and the distributed components of the model over the network

The client accesses the distributed components via the network layer, and this can lead to performance degradation if the model is complex with numerous distributed components. Network and client performance degradation occur when a number of distributed business components implement the application model and the client directly interacts with these components to obtain model data from that component. Each such access results in a remote method call that introduces network overhead and increases the chattiness between the client and the business tier.

The client must reconstruct the model after obtaining the model's parts from the distributed components. The client therefore needs to have the necessary business logic to construct the model. If the model construction is complex and numerous objects are involved in its definition, then there may be an additional performance overhead on the client due to the construction process. In addition, the client must contain the business logic to manage the relationships between the components, which results in a more complex, larger client. When the client constructs the application model, the construction happens on the client side. Complex model construction can result in a significant performance overhead on the client side for clients with limited resources.

Because the client is tightly coupled to the model, changes to the model require changes to the client. Furthermore, if there are different types of clients, it is more difficult to manage the changes across all client types. When there is tight coupling between the client and model implementation, which occurs when the client has direct knowledge of the model and manages the business component relationships, then changes to the model necessitate changes to the client. There is the further problem of code duplication for model access, which occurs when an application has many types of clients. This duplication makes client (code) management difficult when the model changes.

Forces

Separation of business logic is required between the client and the server-side components.

Because the model consists of distributed components, access to each component is associated with a network overhead. It is desirable to minimize the number of remote method calls over the network.

The client typically needs only to obtain the model to present it to the user. If the client must interact with multiple components to construct the model on the fly, the chattiness between the client and the application increases. Such chattiness may reduce the network performance.

Even if the client wants to perform an update, it usually updates only certain parts of the model and not the entire model.

Clients do not need to be aware of the intricacies and dependencies in the model implementation. It is desirable to have loose coupling between the clients and the business components that implement the application model.

Clients do not otherwise need to have the additional business logic required to construct the model from various business components.

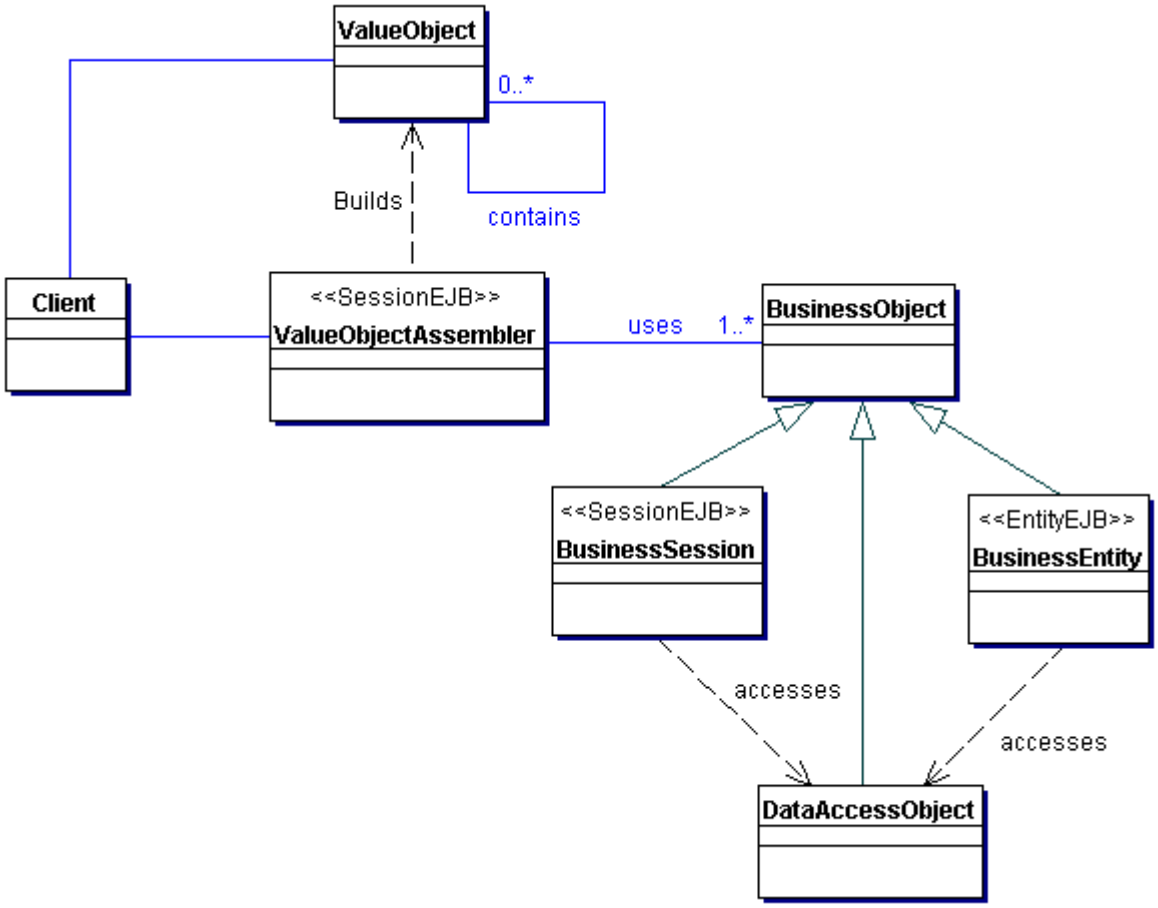
Solution

Use a Transfer Object Assembler to build the required model or submodel. The Transfer Object Assembler uses Transfer Objects to retrieve data from various business objects and other objects that define the model or part of the model.

The Transfer Object Assember constructs a composite Transfer Object that represents data from different business components. The Transfer Object caries the data for the model to the client in a single method call. Since the model data can be complex, it is recommended that this Transfer Object be immutable. That is, the client obtains such Transfer Objects with the sole purpose of using them for presentation and processing in a read-only manner. Clients are not allowed to make changes to the Transfer Objects.

When the client needs the model data, and if the model is represented by a single coarse-grained component (such as a Composite Entity), then the process of obtaining the model data is simple. The client simply requests the coarse-grained component for its composite Transfer Object. However, most real-world applications have a model composed of a combination of many coarse-grained and fine-grained components. In this case, the client must interact with numerous such business components to obtain all the data necessary to represent the model. The immediate drawbacks of this approach can be seen in that the clients become tightly coupled to the model implementation (model elements) and that the clients tend to make numerous remote method invocations to obtain the data from each individual component.

In some cases, a single coarse-grained component provides the model or parts of the model as a single Transfer Object (simple or composite). However, when multiple components represent the model, a single Transfer Object (simple or composite) may not represent the entire model. To represent the model, it is necessary to obtain Transfer Objects from various components and assemble them into a new composite Transfer Object. The server, not the client, should perform such "on-the-fly" construction of the model.



2.2.6. Value List Handler—值列表处理器模式

Context

The client requires a list of items from the service for presentation. The number of items in the list is unknown and can be quite large in many instances.

Problem

Most Java 2 Platform, Enterprise Edition (J2EE) applications have a search and query requirement to search and list certain data. In some cases, such a search and query operation could yield results that can be quite large. It is impractical to return the full result set when the client's requirements are to traverse the results, rather than process the complete set. Typically, a client uses the results of a query for read-only purposes, such as displaying the result list. Often, the client views only the first few matching records, and then may discard the remaining records and attempt a new query. The search activity often does not involve an immediate transaction on the matching objects. The practice of getting a list of values represented in entity beans by calling an ejbFind() method, which returns a collection of remote objects, and then calling each entity bean to get the value, is very network expensive and is considered a bad practice.

There are consequences associated with using Enterprise JavaBeans (EJB) finder methods that result in large results sets. Every container implementation has a certain amount of finder method overhead for creating a collection of EJBObject references. Finder method behavior performance varies, depending on a vendor's container implementation.

According to the EJB specification, a container may invoke ejbActivate() methods on entities found by a finder method. At a minimum, a finder method returns the primary keys of the matching entities, which the container returns to the client as a collection of EJBObject references. This behavior applies for all container implementations. Some container implementations may introduce additional finder method overhead by associating the entity bean instances to these EJBObject instances to give the client access to those entity beans. However, this is a poor use of resources if the client is not interested in accessing the bean or invoking its methods. This overhead can significantly impede application performance if the application includes queries that produce many matching results.

Forces

The application client needs an efficient query facility to avoid having to call the entity bean's ejbFind() method and invoking each remote object returned.

A server-tier caching mechanism is needed to serve clients that cannot receive and process the entire results set.

A query that is repeatedly executed on reasonably static data can be optimized to provide faster results. This depends on the application and on the implementation of this pattern.

EJB finder methods are not suitable for browsing entire tables in the database or for searching large result sets from a table.

Finder methods may have considerable overhead when used to find large numbers of result objects. The container may create a large number of infrastructure objects to facilitate the finders.

EJB finder methods are not suitable for caching results. The client may not be able to handle the entire result set in a single call. If so, the client may need server-side caching and navigation functions to traverse the result set.

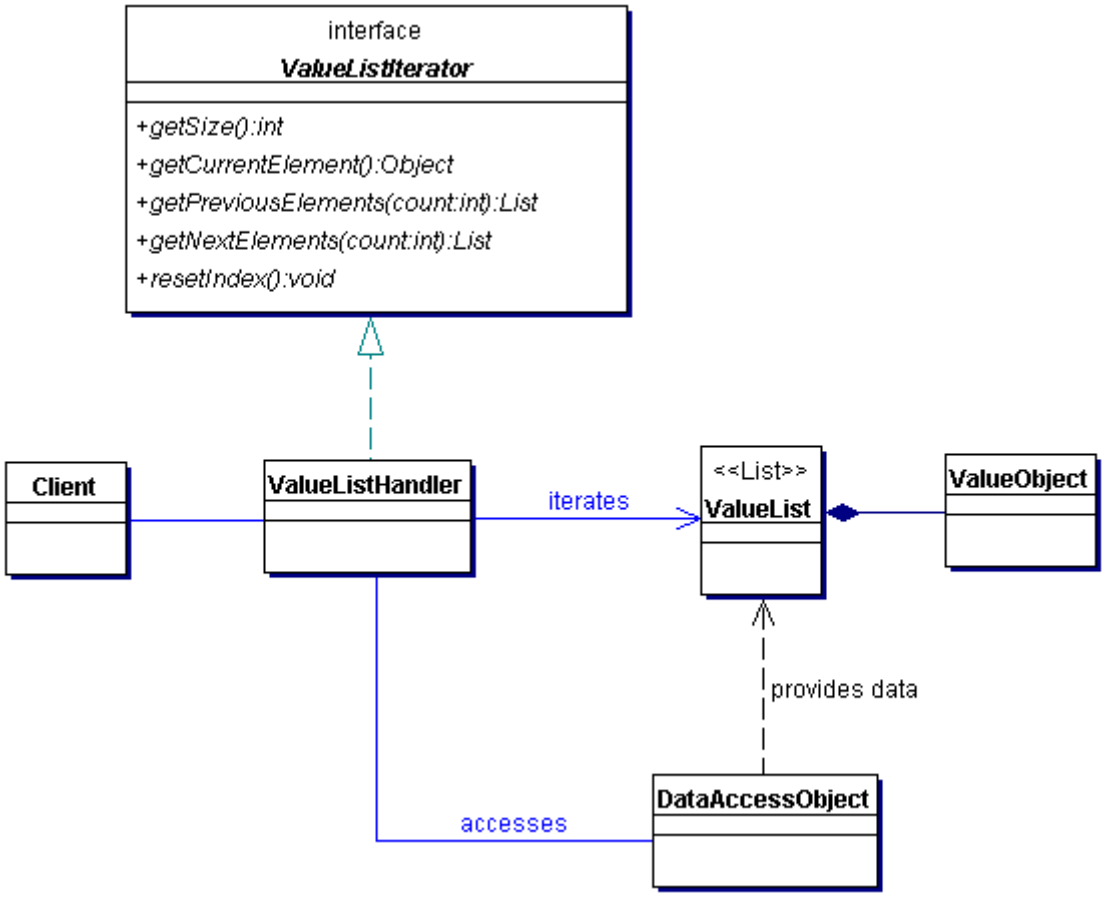
EJB finder methods have predetermined query constructs and offer minimum flexibility. The EJB specification 2.0 allows a query language, EJB QL, for container-managed entity beans. EJB QL makes it easier to write portable finders and offers greater flexibility for querying.

Client wants to scroll forward and backward within a result set.

Solution

Use a Value List Handler to control the search, cache the results, and provide the results to the client in a result set whose size and traversal meets the client's requirements.

This pattern creates a ValueListHandler to control query execution functionality and results caching. The ValueListHandler directly accesses a DAO that can execute the required query. The ValueListHandler stores the results obtained from the DAO as a collection of Transfer Objects. The client requests the ValueListHandler to provide the query results as needed. The ValueListHandler implements an Iterator pattern [GoF] to provide the solution.



2.2.7. Service Locator—服务定位器模式

Context

Service lookup and creation involves complex interfaces and network operations.

Problem

J2EE clients interact with service components, such as Enterprise JavaBeans (EJB) and Java Message Service (JMS) components, which provide business services and persistence capabilities. To interact with these components, clients must either locate the service component (referred to as a lookup operation) or create a new component. For instance, an EJB client must locate the enterprise bean's home object, which the client then uses either to find an object or to create or remove one or more enterprise beans. Similarly, a JMS client must first locate the JMS Connection Factory to obtain a JMS Connection or a JMS Session.

All Java 2 Platform, Enterprise Edition (J2EE) application clients use the JNDI common facility to look up and create EJB and JMS components. The JNDI API enables clients to obtain an initial context object that holds the component name to object bindings. The client begins by obtaining the initial context for a bean's home object. The initial context remains valid while the client session is valid. The client provides the JNDI registered name for the required object to obtain a reference to an administered object. In the context of an EJB application, a typical administered object is an enterprise bean's home object. For JMS applications, the administered object can be a JMS Connection Factory (for

a Topic or a Queue) or a JMS Destination (a Topic or a Queue).

So, locating a JNDI-administered service object is common to all clients that need to access that service object. That being the case, it is easy to see that many types of clients repeatedly use the JNDI service, and the JNDI code appears multiple times across these clients. This results in an unnecessary duplication of code in the clients that need to look up services.

Also, creating a JNDI initial context object and performing a lookup on an EJB home object utilizes significant resources. If multiple clients repeatedly require the same bean home object, such duplicate effort can negatively impact application performance.

Let us examine the lookup and creation process for various J2EE components.

The lookup and creation of enterprise beans relies upon the following:

A correct setup of the JNDI environment so that it connects to the naming and directory service used by the application. Setup entails providing the location of the naming service and the necessary authentication credentials to access that service.

The JNDI service can then provide the client with an initial context that acts as a placeholder for the component name-to-object bindings. The client requests this initial context to look up the EJBHome object for the required enterprise bean by providing the JNDI name for that EJBHome object.

Find the EJBHome object using the initial context's lookup mechanism.

After obtaining the EJBHome object, create, remove, or find the enterprise bean, using the EJBHome object's create, move, and find (for entity beans only).

The lookup and creation of JMS components (Topic, Queue, QueueConnection, QueueSession, TopicConnection, TopicSession, and so forth) involves the following steps. Note that in these steps, Topic refers to the publish/subscribe messaging model and Queue refers to the point-to-point messaging model.

Set up the JNDI environment to the naming service used by the application. Setup entails providing the location of the naming service and the necessary authentication credentials to access that service.

Obtain the initial context for the JMS service provider from the JNDI naming service.

Use the initial context to obtain a Topic or a Queue by supplying the JNDI name for the topic or the queue. Topic and Queue are JMSDestination objects.

Use the initial context to obtain a TopicConnectionFactory or a QueueConnectionFactory by supplying the JNDI name for the topic or queue connection factory.

Use the TopicConnectionFactory to obtain a TopicConnection or QueueConnectionFactory to obtain a QueueConnection.

Use the TopicConnection to obtain a TopicSession or a QueueConnection to obtain a QueueSession.

Use the TopicSession to obtain a TopicSubscriber or a TopicPublisher for the required Topic. Use the QueueSession to obtain a QueueReceiver or a QueueSender for the required Queue.

The process to look up and create components involves a vendor-supplied context factory implementation. This introduces vendor dependency in the application clients that need to use the JNDI lookup facility to locate the enterprise beans and JMS components,

such as topics, queues, and connection factory objects.

Forces

EJB clients need to use the JNDI API to look up EJBHome objects by using the enterprise bean's registered JNDI name.

JMS clients need to use JNDI API to look up JMS components by using the JNDI names registered for JMS components, such as connection factories, queues, and topics.

The context factory to use for the initial JNDI context creation is provided by the service provider vendor and is therefore vendor- dependent. The context factory is also dependent on the type of object being looked up. The context for JMS is different from the context for EJB, with different providers.

Lookup and creation of service components could be complex and may be used repeatedly in multiple clients in the application.

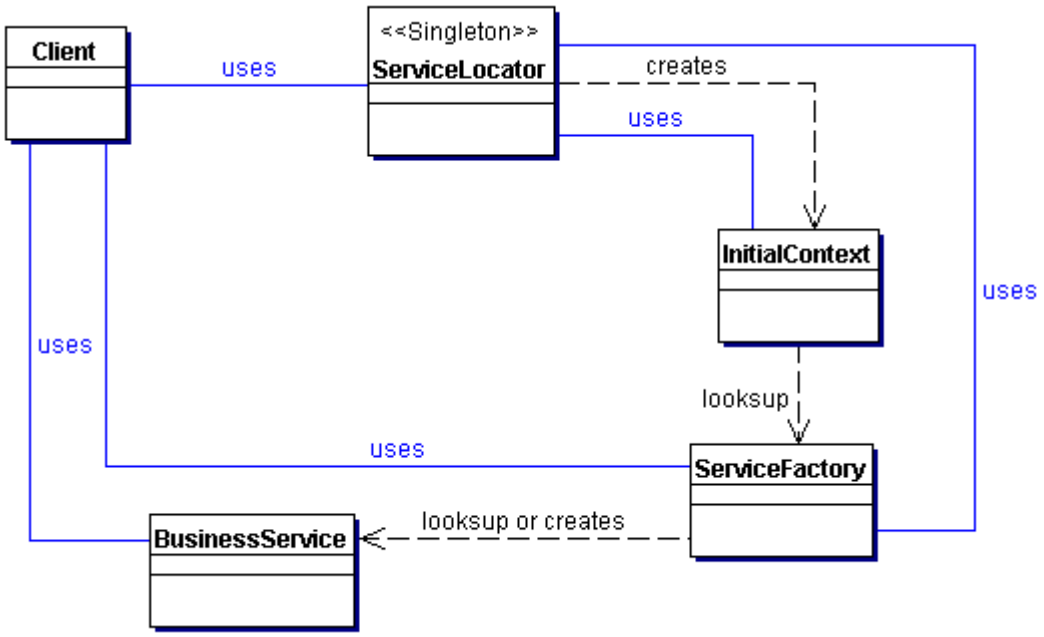
Initial context creation and service object lookups, if frequently required, can be resource-intensive and may impact application performance. This is especially true if the clients and the services are located in different tiers.

EJB clients may need to reestablish connection to a previously accessed enterprise bean instance, having only its Handle object.

Solution

Use a Service Locator object to abstract all JNDI usage and to hide the complexities of initial context creation, EJB home object lookup, and EJB object re-creation. Multiple clients can reuse the Service Locator object to reduce code complexity, provide a single point of control, and improve performance by providing a caching facility.

This pattern reduces the client complexity that results from the client's dependency on and need to perform lookup and creation processes, which are resource-intensive. To eliminate these problems, this pattern provides a mechanism to abstract all dependencies and network details into the Service Locator.



2.3. 集成层模式

2.3.1. Data Access Object—数据访问对象模式

Context

Access to data varies depending on the source of the data. Access to persistent storage, such as to a database, varies greatly depending on the type of storage (relational databases, object-oriented databases, flat files, and so forth) and the vendor implementation.

Problem

Many real-world Java 2 Platform, Enterprise Edition (J2EE) applications need to use persistent data at some point. For many applications, persistent storage is implemented with different mechanisms, and there are marked differences in the APIs used to access these different persistent storage mechanisms. Other applications may need to access data that resides on separate systems. For example, the data may reside in mainframe systems, Lightweight Directory Access Protocol (LDAP) repositories, and so forth. Another example is where data is provided by services through external systems such as business-to-business (B2B) integration systems, credit card bureau service, and so forth.

Typically, applications use shared distributed components such as entity beans to represent persistent data. An application is considered to employ bean-managed persistence (BMP) for its entity beans when these entity beans explicitly access the persistent storage-the entity bean includes code to directly access the persistent storage. An application with simpler requirements may forego using entity beans and instead use session beans or servlets to directly access the persistent storage to retrieve and modify the data. Or, the application could use entity beans with container-managed persistence, and thus let the container handle the transaction and persistent details.

Applications can use the JDBC API to access data residing in a relational database management system (RDBMS). The JDBC API enables standard access and manipulation of data in persistent storage, such as a relational database. The JDBC API enables J2EE applications to use SQL statements, which are the standard means for accessing RDBMS tables. However, even within an RDBMS environment, the actual syntax and format of the SQL statements may vary depending on the particular database product.

There is even greater variation with different types of persistent storage. Access mechanisms, supported APIs, and features vary between different types of persistent stores such as RDBMS, object-oriented databases, flat files, and so forth. Applications that need to access data from a legacy or disparate system (such as a mainframe, or B2B service) are often required to use APIs that may be proprietary. Such disparate data sources offer challenges to the application and can potentially create a direct dependency between application code and data access code. When business components-entity beans, session beans, and even presentation components like servlets and helper objects for JavaServer Pages (JSP) pages --need to access a data source, they can use the appropriate API to achieve connectivity and manipulate the data source. But including the connectivity and data access code within these components introduces a tight coupling between the components and the data source implementation. Such code dependencies in components make it difficult and tedious to migrate the application from one type of data source to another. When the data source changes, the components need to be changed to handle the

new type of data source.

Forces

Components such as bean-managed entity beans, session beans, servlets, and other objects like helpers for JSP pages need to retrieve and store information from persistent stores and other data sources like legacy systems, B2B, LDAP, and so forth.

Persistent storage APIs vary depending on the product vendor. Other data sources may have APIs that are nonstandard and/or proprietary. These APIs and their capabilities also vary depending on the type of storage-RDBMS, object-oriented database management system (OODBMS), XML documents, flat files, and so forth. There is a lack of uniform APIs to address the requirements to access such disparate systems.

Components typically use proprietary APIs to access external and/or legacy systems to retrieve and store data.

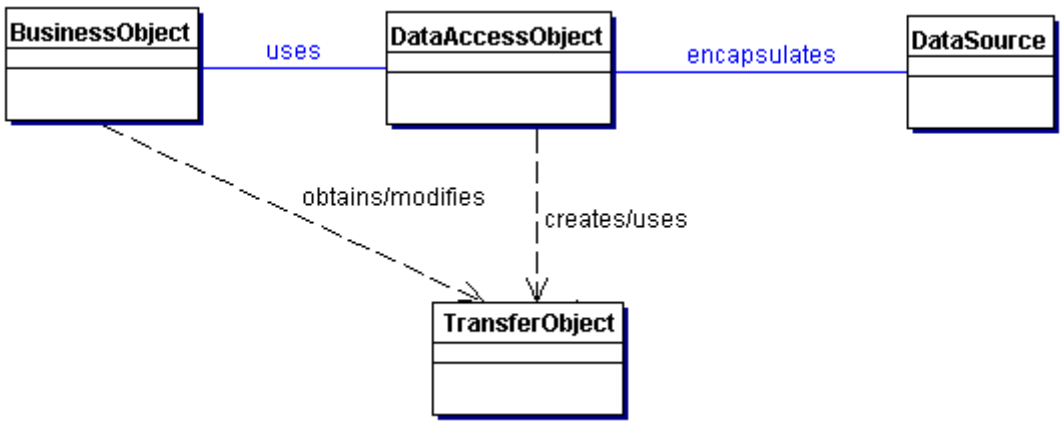
Portability of the components is directly affected when specific access mechanisms and APIs are included in the components.

Components need to be transparent to the actual persistent store or data source implementation to provide easy migration to different vendor products, different storage types, and different data source types.

Solution

Use a Data Access Object (DAO) to abstract and encapsulate all access to the data source. The DAO manages the connection with the data source to obtain and store data.

The DAO implements the access mechanism required to work with the data source. The data source could be a persistent store like an RDBMS, an external service like a B2B exchange, a repository like an LDAP database, or a business service accessed via CORBA Internet Inter-ORB Protocol (IIOP) or low-level sockets. The business component that relies on the DAO uses the simpler interface exposed by the DAO for its clients. The DAO completely hides the data source implementation details from its clients. Because the interface exposed by the DAO to clients does not change when the underlying data source implementation changes, this pattern allows the DAO to adapt to different storage schemes without affecting its clients or business components. Essentially, the DAO acts as an adapter between the component and the data source.



2.3.2. Service Activator—服务激发器模式

Context

Enterprise beans and other business services need a way to be activated asynchronously.

Problem

When a client needs to access an enterprise bean, it first looks up the bean's home object. The client requests the Enterprise JavaBeans (EJB) component's home to provide a remote reference to the required enterprise bean. The client then invokes business method calls on the remote reference to access the enterprise bean services. All these method calls, such as lookup and remote method calls, are synchronous. The client has to wait until these methods return.

Another factor to consider is the life cycle of an enterprise bean. The EJB specification permits the container to passivate an enterprise bean to secondary storage. As a result, the EJB container has no mechanism by which it can provide a process-like service to keep an enterprise bean constantly in an activated and ready state. Because the client must interact with the enterprise bean using the bean's remote interface, even if the bean is in an activated state in the container, the client still needs to obtain its remote interface via the lookup process and still interacts with the bean in a synchronous manner.

If an application needs synchronous processing for its server-side business components, then enterprise beans are an appropriate choice. Some application clients may require asynchronous processing for the server-side business objects because the clients do not need to wait or do not have the time to wait for the processing to complete. In cases where the application needs a form of asynchronous processing, enterprise beans do not offer this capability in implementations prior to the EJB 2.0 specification.

The EJB 2.0 specification provides integration by introducing message-driven bean, which is a special type of stateless session bean that offers asynchronous invocation capabilities. However, the new specification does not offer asynchronous invocation for other types of enterprise beans, such as stateful or entity beans.

In general, a business service such as a session or entity bean provides only synchronous processing and thus presents a challenge to implementing asynchronous processing.

Forces

Enterprise beans are exposed to their clients via their remote interfaces, which allow only synchronous access.

The container manages enterprise beans, allowing interactions only via the remote references. The EJB container does not allow direct access to the bean implementation and its methods. Thus, implementing the JMS message listener in an enterprise bean is not feasible, since this violates the EJB specification by permitting direct access to the bean implementation.

An application needs to provide a publish/subscribe or point-to-point messaging framework where clients can publish requests to enterprise beans for asynchronous processing.

Clients need asynchronous processing capabilities from the enterprise beans and other business components that can only provide synchronous access, so that the client can send a request for processing without waiting for the results.

Clients want to use the message-oriented middleware (MOM) interfaces offered by the Java Messaging Service (JMS). These interfaces are not integrated into EJB server products that are based on the pre-EJB 2.0 specification.

An application needs to provide daemon-like service so that an enterprise bean can be in a quiet mode until an event (or a message) triggers its activity.

Enterprise beans are subject to the container life cycle management, which includes passivation due to time-outs, inactivity and resource management. The client will have to invoke on an enterprise bean to activate it again.

The EJB 2.0 specification introduces a message-driven bean as a stateless session bean, but it is not possible to invoke other types of enterprise beans asynchronously.

Solution

Use a Service Activator to receive asynchronous client requests and messages. On receiving a message, the Service Activator locates and invokes the necessary business methods on the business service components to fulfill the request asynchronously.

The ServiceActivator is a JMS Listener and delegation service that requires implementing the JMS message listener-making it a JMS listener object that can listen to JMS messages. The ServiceActivator can be implemented as a standalone service. Clients act as the message generator, generating events based on their activity.

Any client that needs to asynchronously invoke a business service, such as an enterprise bean, may create and send a message to the Service Activator. The Service Activator receives the message and parses it to interpret the client request. Once the client's request is parsed or unmarshalled, the Service Activator identifies and locates the necessary business service component and invokes business methods to complete processing of the client's request asynchronously.

The Service Activator may optionally send an acknowledgement to the client after successfully completing the request processing. The Service Activator may also notify the client or other services on failure events if it fails to complete the asynchronous request processing.

The Service Activator may use the services of a Service Locator to locate a business component. See "Service Locator" on page 368.