Martin 笔记

Contents

[一. Properties 4](#_Toc398036771)

[Fixed Properties 4](#_Toc398036772)

[Dynamic Properties 4](#_Toc398036773)

[Defined Dynamic Properties 5](#_Toc398036774)

[Typed Dynamic Properties 6](#_Toc398036775)

[Dynamic Properties with Multi-value 6](#_Toc398036776)

[二. Record the history of the changes —Temporal 7](#_Toc398036777)

[Effectivity 7](#_Toc398036778)

[Temporal Property 7](#_Toc398036779)

[Snapshot 7](#_Toc398036780)

[Dimensions of Time 8](#_Toc398036781)

[三. Quant 8](#_Toc398036782)

[Money 8](#_Toc398036783)

[四. 组织结构Organization Structures 10](#_Toc398036784)

[Hierarchy on Party 11](#_Toc398036785)

[Accountbility 11](#_Toc398036786)

[Knowledge Level 13](#_Toc398036787)

[Levelled Accountability Type 14](#_Toc398036788)

[派生属性 15](#_Toc398036789)

[五. 角色建模——Role 16](#_Toc398036790)

[Role Object 16](#_Toc398036791)

[六. Event 17](#_Toc398036792)

[Representing an Event 17](#_Toc398036793)

[Event Sourcing – 记录Event的历史，可回滚 18](#_Toc398036794)

[Agreement Dispatcher 21](#_Toc398036795)

[Parallel Model – Run alternative states 22](#_Toc398036796)

[Retroactive Model – Handle error input 23](#_Toc398036797)

[七. 筛选标准Specification 27](#_Toc398036798)

[八. Glossary 27](#_Toc398036799)

# Properties

http://martinfowler.com/apsupp/properties.pdf

Property is some information about an object that you can obtain by a query method.

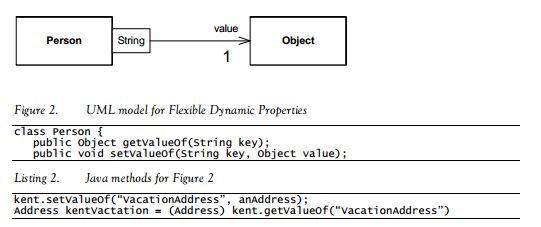
### Fixed Properties

最普通的Property，最经常使用。

优点：

* + 1. 接口清晰，直接使用PropertyName来获取值；
    2. 从Interface角度看，将Property的实现隔离，一个Property的值可以来自自己的存储、其他object、一组计算的结果，但使用Proeprty就可以不必知道究竟是来自于哪里
    3. 当Property改变时，所有依赖于这个Property的代码会在编译期就发现这种改变，从而随之改变；

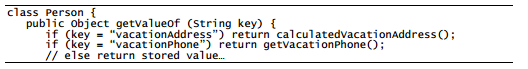
### Dynamic Properties



用一个<Key Value>来实现，PropertyName就是Key，值就是Value。所有使用Dynamic Proeprty的对象要支持Get和Set函数。

缺点：You lose the clear interface and all design time checking.

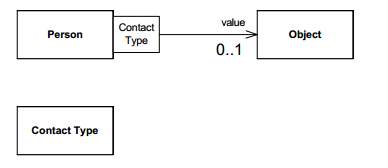
* + 1. 接口不清晰，不知道究竟有什么Property
    2. 如果client代码已经使用了一个Property，但这个property的Name随后改变了或者类型改变了，client代码无法察觉这种变化，将会导致运行错误；
    3. 使用KeyValue来存储，不适合必须通过计算来获取Property的场合。

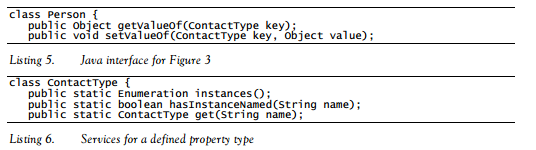


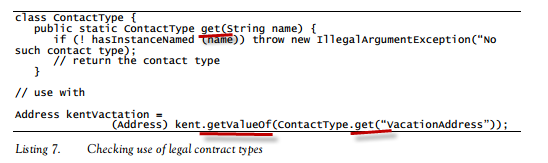
### Defined Dynamic Properties

Dynamic Property 加上在运行期检查Property是否存在的功能。实现方法是，不再用一个String来做Key，而是用一个具体的Class的对象来做Key，这个对象是***类型对象***，包括三个职责，定义所有可用的PropertyName，检查PropertyName是否存在，实例化一个类型对象。

注意对PropertyName是否存在的检查不是在获取Property值的时候完成的，而是在这之前，通过一个PropertyName获得一个ContactType对象时做的，如果PropertyName不合法那么就不能返回一个ContactType对象，因此必须是static函数，也就不用提之后用ContactType对象去获得对象值了。

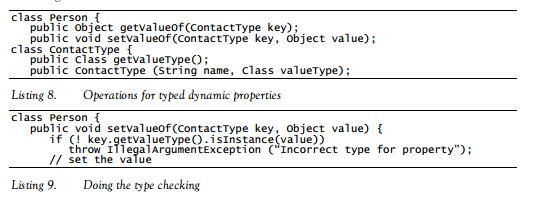






### Typed Dynamic Properties

**Defined Dynamic Properties** + Property的类型检查。对Property的类型检查是在设置属性值的时候做的，因此这个函数应该属于作为key的ContactType对象。

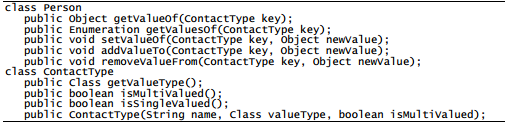


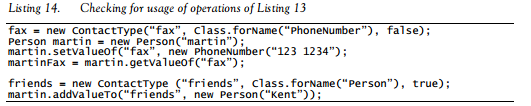
### Dynamic Properties with Multi-value

解决Property是个List的问题。

所有包含这种Property的对象必须能够获取值，设置值，还要能够获取列表，增加一项，去除一项。

作为Key的ContractType来判断是单值还是多值。



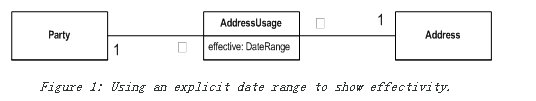


# Record the history of the changes —Temporal

<http://martinfowler.com/eaaDev/timeNarrative.html>

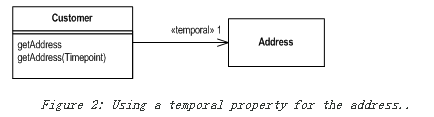
Not just do we want to know the state of the world, we want to know the state of the world six months ago. Even worse we may want to know what two months ago we thought the state of the world six months ago was.

### Effectivity



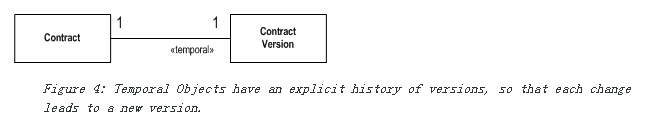
用一个DateRange类型的effective属性来表示是属于哪个时间段的Address

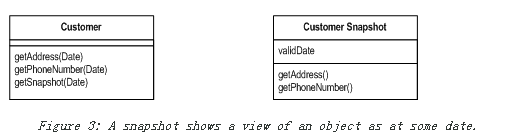
### Temporal Property



所有属性都有两个，一个与时间无关一个要传一个时间，返回的东西都一样，返回的对象本身不带时间。

### Snapshot





传一个Date进去后获得的是Customer的整个Sanpshot，属性一样但没有时间。

### Dimensions of Time

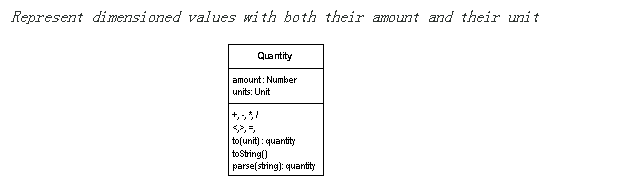
the first dimension as **actual time**: the time something happened. The second dimension is **record time**, the time we knew about it.

# Quant

<http://martinfowler.com/eaaDev/quantity.html>

表示量和单位的对象，支持加减乘除比较操作，支持单位换算和字符串处理。

注意，Quantity是Immutable的，即设了值之后无法改变，改变值只能新建一个对象。



### Money

class Money...

public class Money implements Comparable{

private BigInteger amount;

private Currency currency;

用BigInterger来存储值，从而提高计算精度，也可以用decimal.

.

class Money...

public double amount() {

return amount.doubleValue() / 100;

}

public Currency currency() {

return currency;

}

GetAmount的时候把存储值转成小数，计算的时候用存储值直接计算。

class Money...

public Money (double amount, Currency currency) {

this.amount = BigInteger.valueOf(Math.round (amount \* 100));

this.currency = currency;

}

public Money (long amount, Currency currency) {

this.amount = BigInteger.valueOf(amount \* 100);

this.currency = currency;

}

加减乘除操作都在存储值上直接操作，而不是转成double再做，这样精度不高。

class Money...

public Money add (Money arg) {

assertSameCurrencyAs(arg);

return new Money (amount.add(arg.amount), currency, true);

}

public Money subtract (Money arg) {

return this.add(arg.negate());

}

void assertSameCurrencyAs(Money arg) {

Assert.equals("money math mismatch", currency, arg.currency);

}

private Money (BigInteger amountInPennies, Currency currency, boolean privacyMarker) {

Assert.notNull(amountInPennies);

Assert.notNull(currency);

this.amount = amountInPennies;

this.currency = currency;

}

public Money negate() {

return new Money (amount.negate(), currency, true);

}

class Money...

public Money multiply (double arg) {

return new Money (amount() \* arg, currency);

}

做除法时不是用数学除法，而是把总的钱分成几堆，最后的余数部分从第一堆开始分配，也就是前面的几堆会多分到几分钱。

class Money...

public Money[] divide(int denominator) {

BigInteger bigDenominator = BigInteger.valueOf(denominator);

Money[] result = new Money[denominator];

BigInteger simpleResult = amount.divide(bigDenominator);

for (int i = 0; i < denominator ; i++) {

result[i] = new Money(simpleResult, currency, true);

}

int remainder = amount.subtract(simpleResult.multiply(bigDenominator)).intValue();

for (int i=0; i < remainder; i++) {

result[i] = result[i].add(new Money(BigInteger.valueOf(1), currency, true));

}

return result;

}

比较操作

class Money...

public int compareTo (Object arg) {

Money moneyArg = (Money) arg;

assertSameCurrencyAs(moneyArg);

return amount.compareTo(moneyArg.amount);

}

class Money...

public boolean greaterThan(Money arg) {

return (this.compareTo(arg) == 1);

}

public boolean lessThan(Money arg) {

return (this.compareTo(arg) == -1);

}

class Money...

public boolean equals(Object arg) {

if (!(arg instanceof Money)) return false;

Money other = (Money) arg;

return (currency.equals(other.currency) && (amount.equals(other.amount)));

}

class Money...

public int hashCode() {

return amount.hashCode();

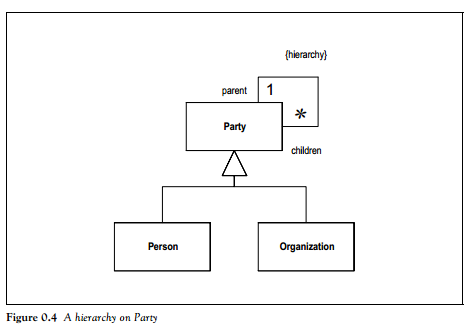
}

# 组织结构Organization Structures

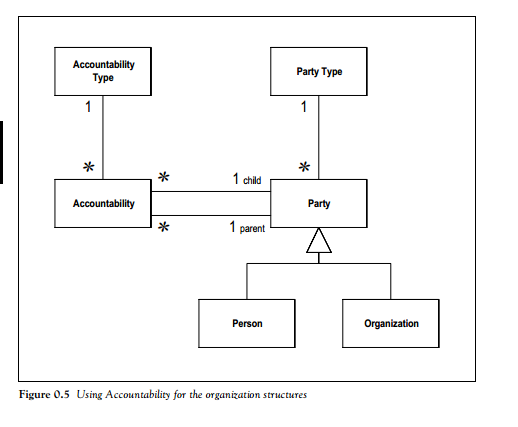
http://martinfowler.com/apsupp/accountability.pdf

所谓组织结构，主要描述的是组织由哪些对象组成，这些对象之间的所属关系（父子关系）。

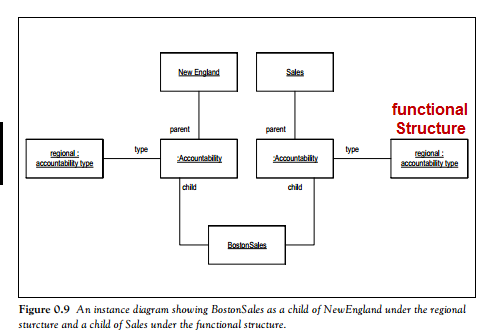
### Hierarchy on Party



### Accountbility

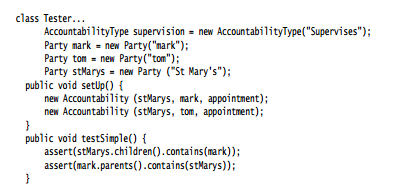


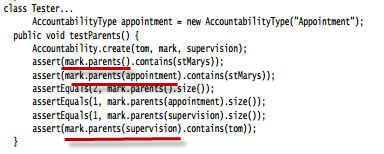
Accountbility本身表示两个Party之间的父子关系，它的用途是表明Party之间的所属关系不是固定的，从不同的AccountbilityType来看，Party之间的所属关系会变，比如从地域这个AccountbilityType来看，BostonSales属于New England，但从functional structure这个AccountbilityType来看，BostonSales属于Sales。



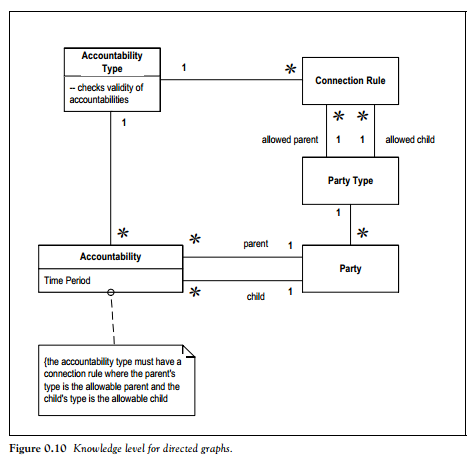
使用的案例如下：

定义了两个AccountabilityType，Supervises和Appointment。Party能知道自己所有的parent和child以及在某个AccountabilityType下自己的parents()和children。



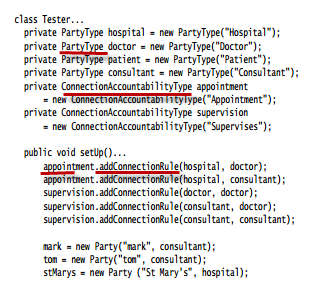


### Knowledge Level

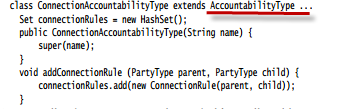


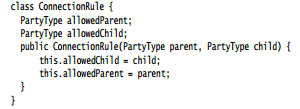
Knowledge Level是给父子关系添加规则，在某种Accountability下，某种Party能否和另一种Party建立父子关系。规则的添加是在创建关系之前建立好。

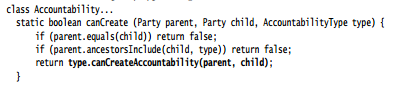
使用案例如下：



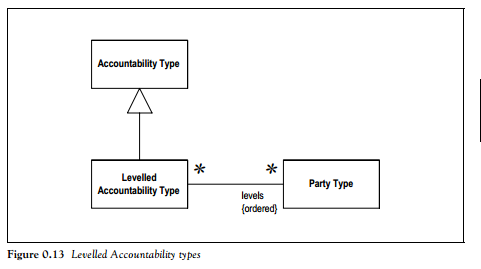
有两种AccountbilityType，appointment和Supervises。Appointment下，允许医院和医生、顾问之间建立关系，因此当代码要添加除此以外父子关系到appointment时，就会抛错。





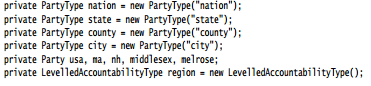


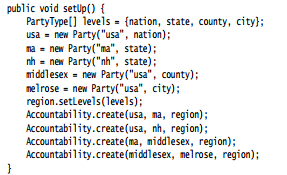
### Levelled Accountability Type

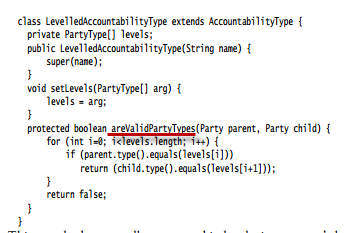


用来定义在某种AccountbilityType下，partyType之间的层级关系。比如在地域下，Party层级有nation,state county city四级，不能越级进行关系设置。

案例代码如下：

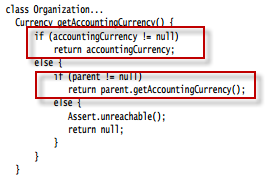






### 派生属性

派生属性是指子Party会默认使用父Party的属性值。子Party也可以使用自己的值。

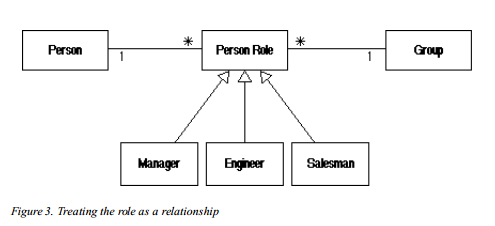


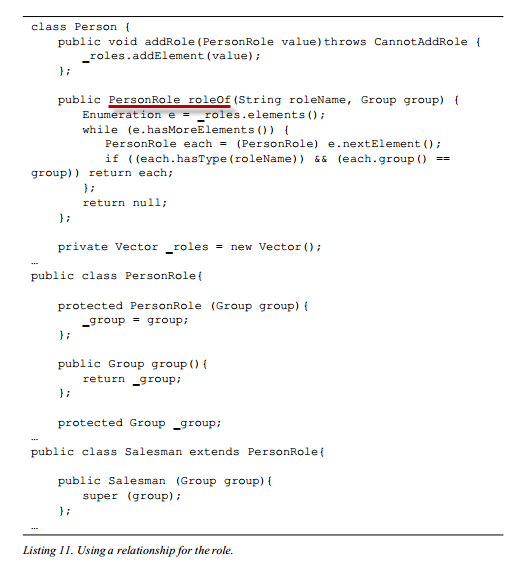
# 角色建模——Role

Role就是Party的Role，这样就和前一部分Accountability结合起来了。

### Role Object

有一个基类PersonRole，所有的其他Role都继承他，并且每个Role有自己的函数与属性。Person中包含了一个Role的List，是他所有承担的角色。这里的Group相当于Organization，意思是每个角色都属于一个组织，不同组织中的Role即使名字是一样的也是不一样的实体。





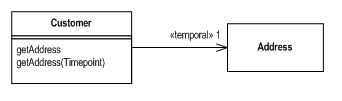
# Temporal Patterns

处理根据时间发生变化的情况。

### Temporal Property

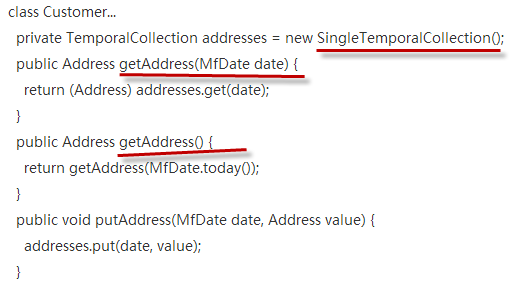
http://blog.csdn.net/lgy2001/article/details/6940664

**A property that changes over time**

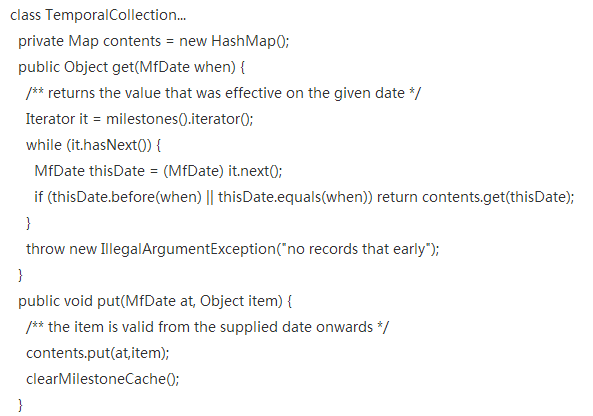


Temporal Property的关键在于怎么提供一个访问属性的接口，它表明属性是跟着时间变化的。

**实现方法Temporal Collection**



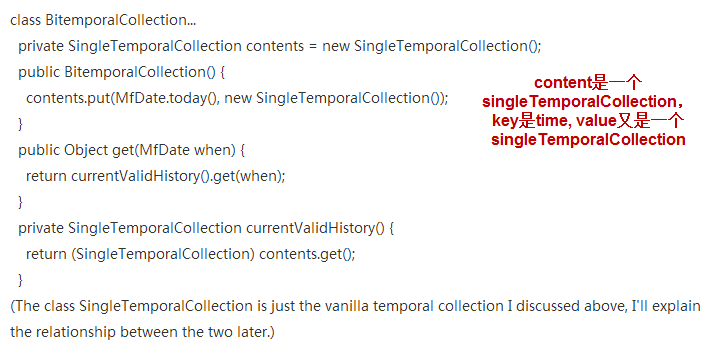
**SingleTemporalCollection**就是个MappingList，每个Period对应Value

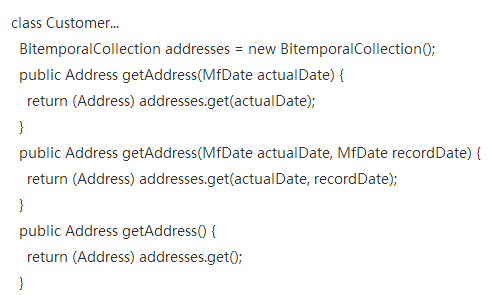


**Bi-temporal Property**

根据记录时间和发生时间查找值







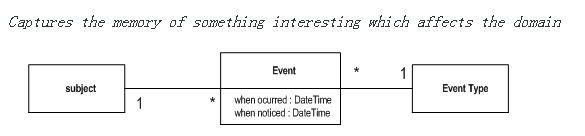
# Event

<http://martinfowler.com/eaaDev/EventNarrative.html>

know about the world in the past or explore the consequences of what might happen in the future

### Representing an Event

<http://martinfowler.com/eaaDev/DomainEvent.html> 主要在Domain Model里记录了这两个信息。



要记录两个信息

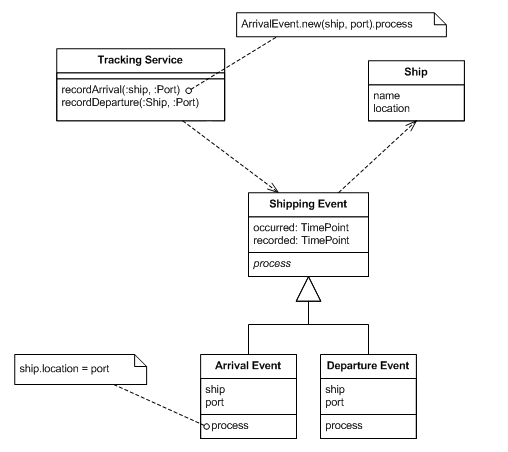
1、2个时间点，Event发生的时间，系统发觉Event的时间

2、Event里有两种data，source data与processing data，要加以区分。source data- an event's source data is immutable - it's what we know of what happened, and we can't easily change it；processing data - what we've done with the event

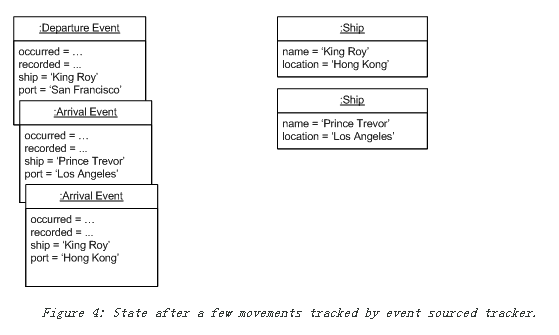
Event的作用是一种Component之间Collaborate的方式。比较常用的方式是request/response。

### Event Sourcing – 记录Event的历史，可回滚

http://martinfowler.com/eaaDev/EventSourcing.html

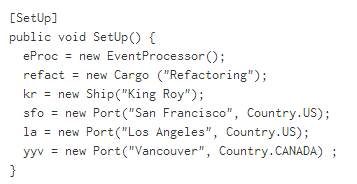


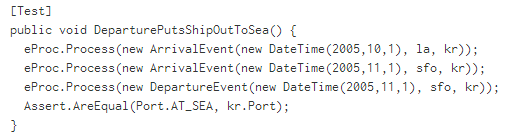
最基本的做法就是Ship对象的属性只记录最新的状态，每改变一次后都记录一个ShippingEvent。



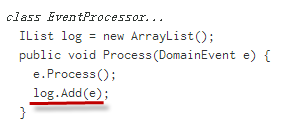
**使用案例**

所有的状态变化都是发生了时间new Event后，由EventProcessor来触发的。

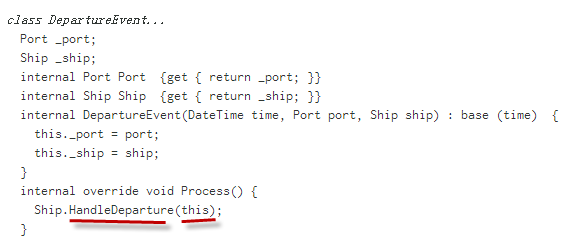




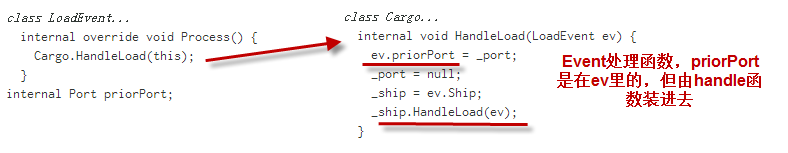
EventPrcessor本身也不处理Event，而是转发给Event的execute函数，它主要做的是记录下这个Event。

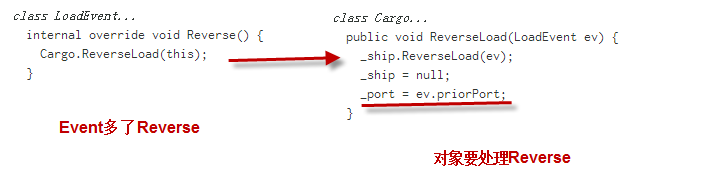


Event的process则把自己发送给所有要关注这个Event的对象，来实现Event的处理



**Reverse操作**。实现方法就是在每个Event对象里存储当前的状态，然后再进行Event处理，也就是处理函数负责在Event里记录当前的状态和Event的处理，Event负责提供保存当前状态的地方。同时Event除了process函数之外，还要提供Reverse函数，每个对象相应地还要提供Event的回滚处理。



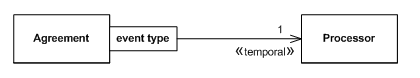


### Agreement Dispatcher

<http://martinfowler.com/eaaDev/AgreementDispatcher.html>

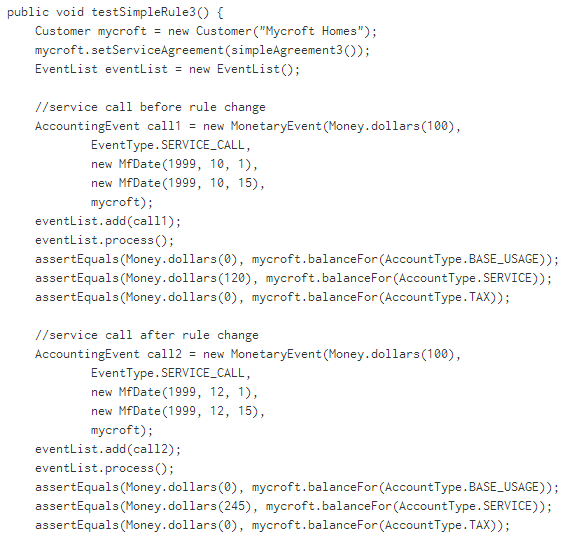
负责管理Event和它的Processor之间的对应关系。

You model each agreement as an object which has a method to process an event



**案例代码**

当发生一个Event时，就是用Agreement的process来处理event。Agreement的处理原理是获得这个event的Type，根据Type查找到EventProcessor，然后由它来完成event处理。



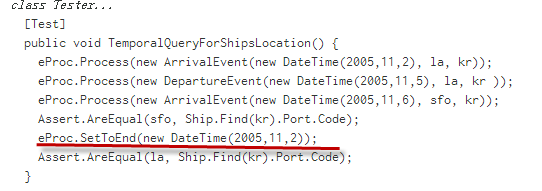


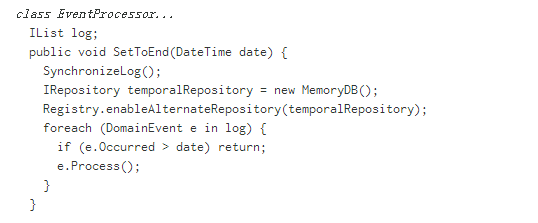
### Parallel Model – Run alternative states

http://martinfowler.com/eaaDev/ParallelModel.html

handling historic and **alternative** states。

实现方法是不同的Event保存在不同的Repository中，当跑假设替换Event时，切换Repository载入不同的Event。





上例中，原本所有的Event都存在DB中，然后从2005-11-2日开始跑一个替换状态，不再使用真实存储的Event来运行。在SetToEnd中，Repository被切换成了MemoryDB，DB中的Event运行到2005-11-2日为止，之后开始用替换的MemoryDB中的Event运行。

### Retroactive Model – Handle error input

http://martinfowler.com/eaaDev/RetroactiveEvent.html

Dealing with a retroactive event means that our current application state is incorrect in some way。

发生retroactive event意味着当前系统的状态已经出错了，所以它包括三个Parallel Model：

* + - Incorrect reality: 当前系统的状态，有错误，由系统当前的Events构成
    - Correct Branch： 执行完retroactive event后，系统将会是什么状态，由系统当前的Events + retroactive event构成
    - Corrected Reality：我们希望的系统的正确状态，由另一组Events构成。通常最后的结果和Correct Branch一样

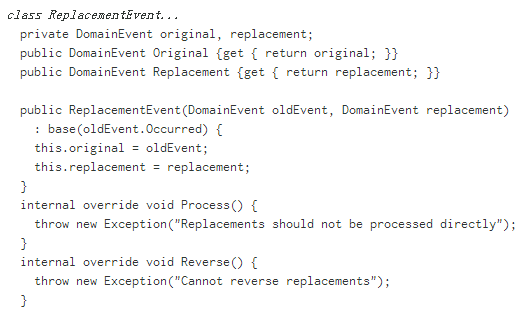
把他们看做三个Parallel Model就意味着每一个都有一系列Events，但是执行完所有Events之后结果不同，各自的Events之间又互不影响。第一步是找到incorrect reality和correct branch开始不同的地方，这个地方就是应该发生retroactive event的地方，成为**Branch Point**。

三种导致错误发生的Event及处理方法。

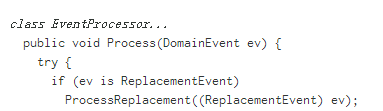
|  |  |  |
| --- | --- | --- |
| **名称** | **含义** | **操作** |
| **out-of-order event** | 收到一个迟到的event，此时已经处理了该迟到event之后的event | insert and process the retroactive event at the branch point. 把它加进去，重新执行 |
| **rejected event** | you now realize was false and should never have been processed | Ignore这个event，之后的重新执行 |
| **incorrect event** | you received incorrect information about the event，知道某个event是错的，并知道正确的是什么 | 1和2 的综合， reverse the original event and mark it as rejected， insert the retroactive event and process it |

**实现incorrect和reject**

对于incorrect和reject，可以实现一个ReplacementEvent，当出现reject时，replacement event设置成null，incorrect时则都赋值。



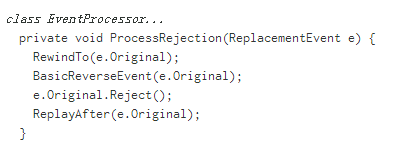
ReplacementEvent不能直接Process和Reverse，它的处理由EventProcessor来负责。



处理ReplacementEvent时，要分四种情况

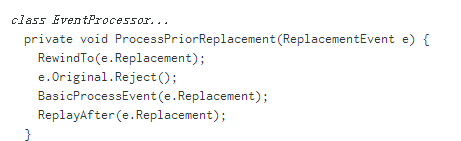
1. 如果original已经reject了，那么这个replacement就是异常
2. 如果replacement为null，就去reject

回滚到original，回滚original，reject original，从original之后重新运行



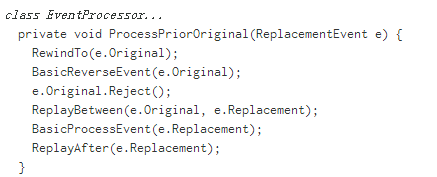
1. 如果replacement在original之前发生，

回滚到replacement，reject original, 执行replacement，执行repalce之后的events



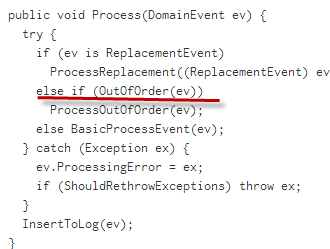
1. 如果replacement在original之后发生，

回滚到original，回滚original，reject 它，重新执行original与replecement之间的events，执行replacement，然后执行replacement之后的events

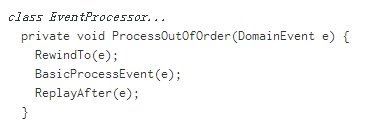


**实现out of order**

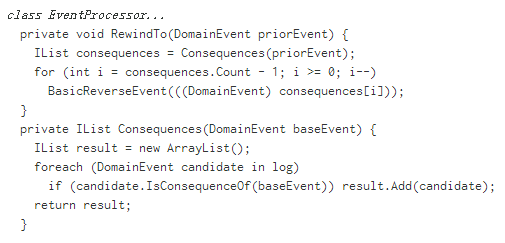
也由EventProcessor来执行。判断一个Event是不是out of order的方法就是拿log中的最后一个event与其比较，看它是否在最后一个event之后发生。



处理out of order的方法就是回滚到event，处理event，然后处理event之后的所有events

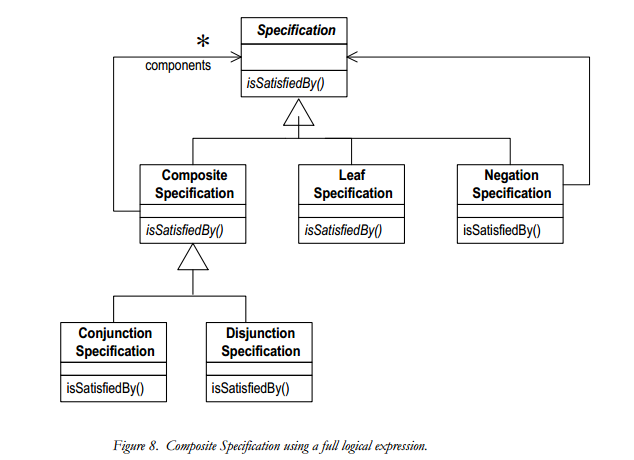


**实现EventProcessor**



# 筛选标准Specification

Need to select a subset of objects based on some criteria. 对criteria建模。



# Glossary

##### 类型对象

是指Martin Fowler的对象图中出现的各种xxxType对象，它的含义相当于面向对象编程语言中的class，那么一个对象依赖于一个class，就是这个对象需要这个class的信息的意思，对应到编程语言里就是这个对象会根据具体的class类型来进行不同的操作，实现起来就是子类override的意思。

从本质上看，依赖于类型对象就是看对象是属于什么类型的，然后做相应操作的一组if…else…，只不过在面向对象编程语言中这种if…else…由类的多态性来实现。

类型对象的函数对应到编程语言就是静态函数，类型对象的属性对应到编程语言就是静态变量。