# 3.10.Strategy

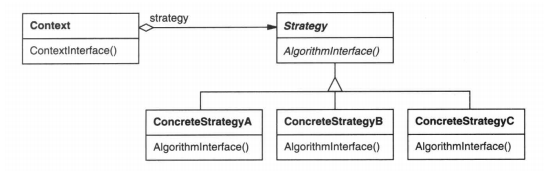
## 3.10.1设计模式简介：

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

Strategy模式适用于以下场景：

1. 许多相关的类仅仅是行为有异。strategy提供了一种用多个行为中的一个行为来配置一个类的方法。
2. 需要使用一个算法的不同变体。例如，你可能会定义一些反映不同的空间/时间权衡的算法。当这些变体实现为一个算法的类层次时,可以使用strategy模式。
3. 算法使用客户不应该知道的数据。可使用策略模式以避免暴露复杂的、与算法相关的数据结构。
4. 一个类定义了多种行为,并且这些行为在这个类的操作中以多个条件语句的形式出现。将相关的条件分支移入它们各自的Strategy类中以代替这些条件语句。

Strategy模式的结构如下：



上图结构中有Strategy，ConcreteStrategy和Context三个参与者。Strategy定义所有支持的算法的公共接口; Context以Strategy接口实现某具体算法;Context用一个ConcreteStrategy对象来配置,维护一个对Strategy对象的引用, 可定义一个接口来让Strategy访问它的数据。

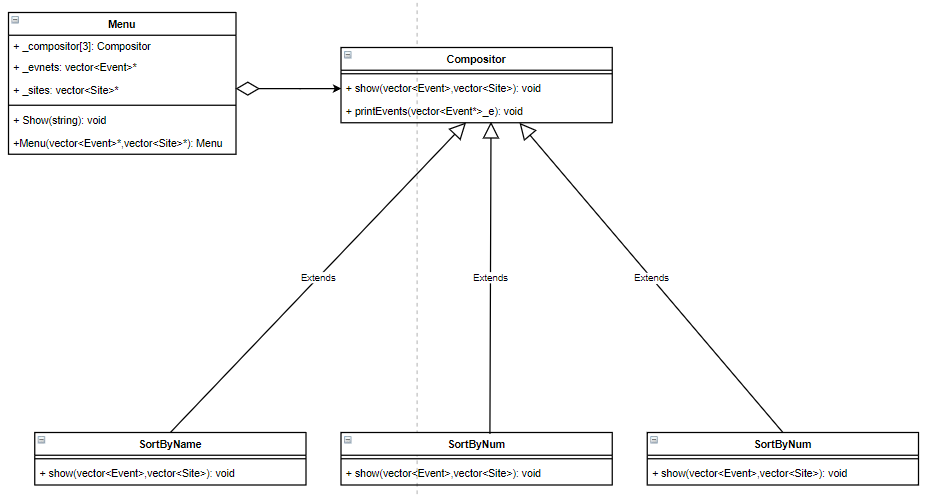
## 3.10.2在本项目中的应用：

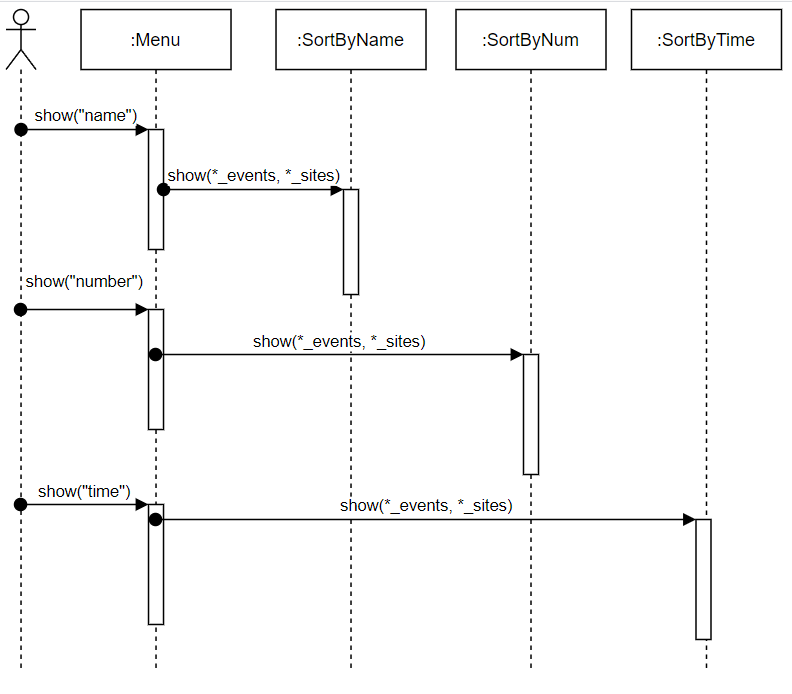
在本项目中，需要用不同的方式展示场地信息——通过场地名字首字母排序，通过场地编号顺序，通过在场地上举办活动的时间顺序来展示场地。这个时候，就需要使用Strategy设计模式，将不同展示方式算法封装起来，方便其他类对展示算法进行调用，当需要增加新的展示算法时，可以通过增加ConcreteStrategy实现，不用修改已有代码；当需要修改展示算法时，只要修改ConcreteStrategy，不需要改动其他代码。

## 3.10.3 api描述：

|  |  |  |
| --- | --- | --- |
| 类名 | 函数声明 | 作用 |
| Compositor | virtual void **show**(**vector**<**Event**>,**vector**<**Site**>) = 0; | 作为纯虚函数，提供一个可以被子类型改写的接口 |
| Compositor | void **printEvents**(**vector**<**Event**\*>\_); | 按照时间顺序打印在某一个场地所举办的所有赛事信息的函数 |
| SortByName | virtual void **show**(**vector**<**Event**>,**vector**<**Site**>); | 对Compositor中的纯虚函数进行实现，将场地按照字母表顺序打印出来 |
| SortByNum | virtual void **show**(**vector**<**Event**>,**vector**<**Site**>); | 对Compositor中的纯虚函数进行实现，将场地按照场地编号的顺序打印出来 |
| SortByTime | virtual void **show**(**vector**<**Event**>,**vector**<**Site**>); | 对Compositor中的纯虚函数进行实现，将场地按照举办赛事的时间顺序打印出来（打印出各赛事所举办的场地） |
| Menu | void **Show**(**string**); | 该函数根据提示信息（传入的string参数）来决定展示赛事的方式。如果传入“name”, 则调用SortByName来展示；如果传入“number”, 则调用SortByTime来展示；如果传入“time”, 则调用SortByTime来展示。 |

## 3.10.4类图和时序图：





# 3.11.Composite

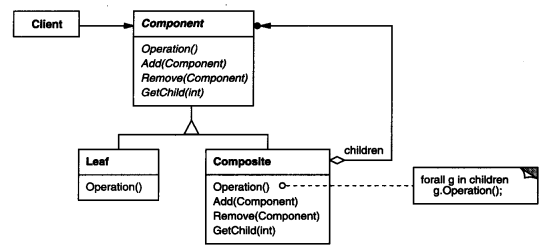
## 3.11.1设计模式简介：

该设计模式的意图是将对象组合成树形结构以表示“部分-整体”的层次结构。Composite使得用户对单个对象和组合对象的使用具有一致性。

Composite模式适用于以下场景：

1. 你想表示对象的部分-整体层次结构。
2. 你希望用户忽略组合对象与单个对象的不同，用户将统一地使用组合结构中的所有对象。

Composite模式的结构如下：



上图结构中有Component，Leaf，Composite和Client四个参与者。Component为组合中的对象声明接口，在适当的情况下，实现所有类共有接口的缺省行为，声明一个接口用于访问和管理Component的子组件，也可以在递归结构中定义一个接口，用于访问一个父部件，并在合适的情况下实现它；Leaf在组合中表示叶节点对象，叶节点没有子节点；Composite定义有子部件的那些部件的行为，存储子部件，在Composite接口中实现与子部件有关的操作；Client通过Component接口操纵组合部件的对象。

## 3.11.2在本项目中的应用：

在本项目中，需要展示所有项目的信息，而所有项目具有一定的层级。比如所有的奥运项目包括跑步、游泳、羽毛球等项目，而跑步、游泳中又包含更小的项目，比如男子100m游泳，女子1000m赛跑等等。可以将项目分为复合项目（CompositeEvents）和原子项目（SingleEvents），要统一地对复合项目和原子项目进行增加、删除、展示等操作，并体现层级关系，Composite是最好的选择。

## 3.11.3 api描述：

|  |  |  |
| --- | --- | --- |
| 类名 | 函数声明 | 作用 |
| Events | virtual void **add**(**Events**\*); | 提供一个增加子赛事的接口，主要用于复合类重载 |
| Events | virtual void **remove**(**Events**\*,**Events** \*); | 提供一个删除子赛事的接口，主要用于复合类重载 |
| Events | virtual int **getType**() = 0; | 获取该类的类型，1代表复合比赛，2代表单个比赛，基类设置为纯虚函数 |
| Events | virtual **LinkList**<**Events**\*>\* **getChildren**(); | 获得子赛事，主要用于复合类重载 |
| Events | **string** **getName**(); | 获得比赛名称 |
| Events | virtual void **show**() = 0; | 提供一个展示函数接口，主要用于子类重载 |
| CompositeEvents | virtual void **add**(**Events**\*); | 重载基类的增加赛事函数 |
| CompositeEvents | virtual int **getType**(); | 返回1，代表这是复合赛事类 |
| CompositeEvents | virtual void **remove**(**Events**\*,**Events** \*); | 重载基类的删除赛事函数，递归地将某一分支的赛事删除干净 |
| CompositeEvents | virtual **LinkList**<**Events**\*>\* **getChildren**(); | 重载基类的获得子赛事函数，获得指向children数据的指针，主要用于remove操作 |
| CompositeEvents | virtual void **show**(); | 重载基类的展示赛事函数，递归地展示子类所有赛事 |
| SingleEvents | virtual void **show**(); | 重载基类的展示赛事函数，展示比赛时间和比赛地点 |
| SingleEvents | virtual int **getType**(); | 返回1，代表这是原子赛事类 |

## 3.11.4类图和时序图：

