面向对象程序设计基础

(OOP)

黄民烈

aihuang@tsinghua.edu.cn

http://coai.cs.tsinghua.edu.cn/hml

课程团队：刘知远 姚海龙 黄民烈

设计模式

- 在日常的开发任务中，采用精心设计的程序架构可以极大方便日常的变动与修改，从而降低维护的代价
- 设计模式（Design Pattern）则是在长时间的实践之中，开发人员总结出的优秀架构与解决方案。经典的设计模式，都是经过相当长的一段时间的试验和错误总结而成的
- 学习设计模式将有助于经验不足的开发人员在实际开发中，灵活地运用面向对象特性，并能够快速构建不同场景下的程序框架，写出优质代码

设计模式

- 《Design Patterns - Elements of Reusable Object-Oriented Software》首次提到了软件开发中设计模式的概念
  - 遵循面向对象设计原则
  - 对接口编程而不是对实现编程（即提高代码复用，抽象通用接口）
  - 优先使用对象组合而不是继承（即降低模型复杂程度，对功能尽可能划分）
- 设计模式也被划分为三大类
  - 行为型模式（Behavioral Patterns）
  - 结构型模式（Structural Patterns）
  - 创建型模式（Creational Patterns）

注：本学期课程不涉及创建型模式

设计模式

- 行为型模式（Behavioral Patterns）
  - 关注对象行为功能上的抽象，从而提升对象在行为功能上的可拓展性，能以最少的代码变动完成功能的增减
- 结构型模式（Structural Patterns）
  - 关注对象之间结构关系上的抽象，从而提升对象结构的可维护性、代码的健壮性，能在结构层面上尽可能的解耦合
- 创建型模式（Creational Patterns）
  - 将对象的创建与使用进行划分，从而规避复杂对象创建带来的资源消耗，能以简短的代码完成对象的高效创建

本讲内容提要

- 设计模式：行为型模式
- 12.1 模板方法（Template Method）模式
- 12.2 策略（Strategy）模式
- 12.3 迭代器（Iterator）模式

一个例子：负载监视器

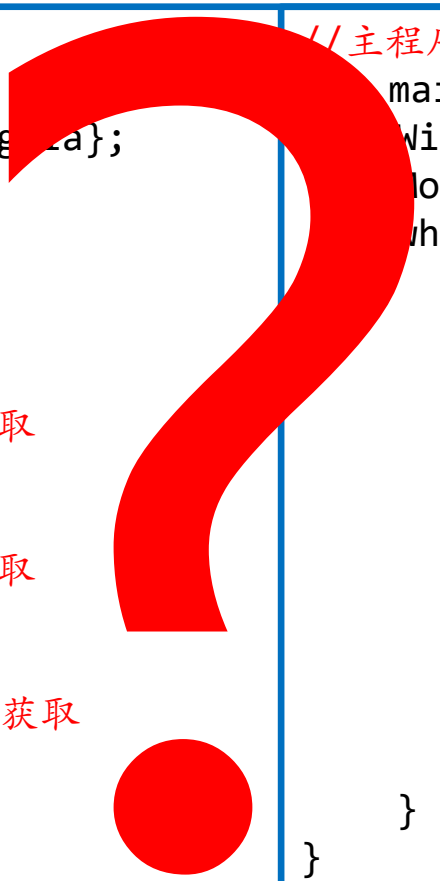
- 监视计算节点的负载状态（如CPU占用率）
- 以CPU占用率的监视为例，不同条件下（例如不同种类不同版本的OS）获得CPU占用率的方法不同
- 怎样在一个程序中实现对这些不同条件的适应呢？

简单枚举

```
class Monitor {
public:
    void getLoad();
    void getTotalMemory();
    void getUsedMemory();
    void getNetworkLatency();
    Monitor(Display *display);
    virtual ~Monitor();
    void show();
private:
//用以存储信息的成员变量
    float load, latency;
    long totalMemory, usedMemory;
    Display* m_display;
};
//组合一个Display接口来进行结果展示
void Monitor::show() {
    m_display -> show(load, totalMemory,
                      usedMemory, latency);
}
```

简单枚举

```
//规定所有的系统类型
enum MonitorType
        {Win32, Win64, Ganglia};
MonitorType type = Ganglia;
…
//获取负载信息的实现
void Monitor::getLoad() {
    switch (type) {
        //Win32版本的信息获取
        case Win32:
            load = …;
        //Win64版本的信息获取
        case Win64:
            load = …;
        //Ganglia版本的信息获取
        case Ganglia:
            load = …;
    }
}
…
```

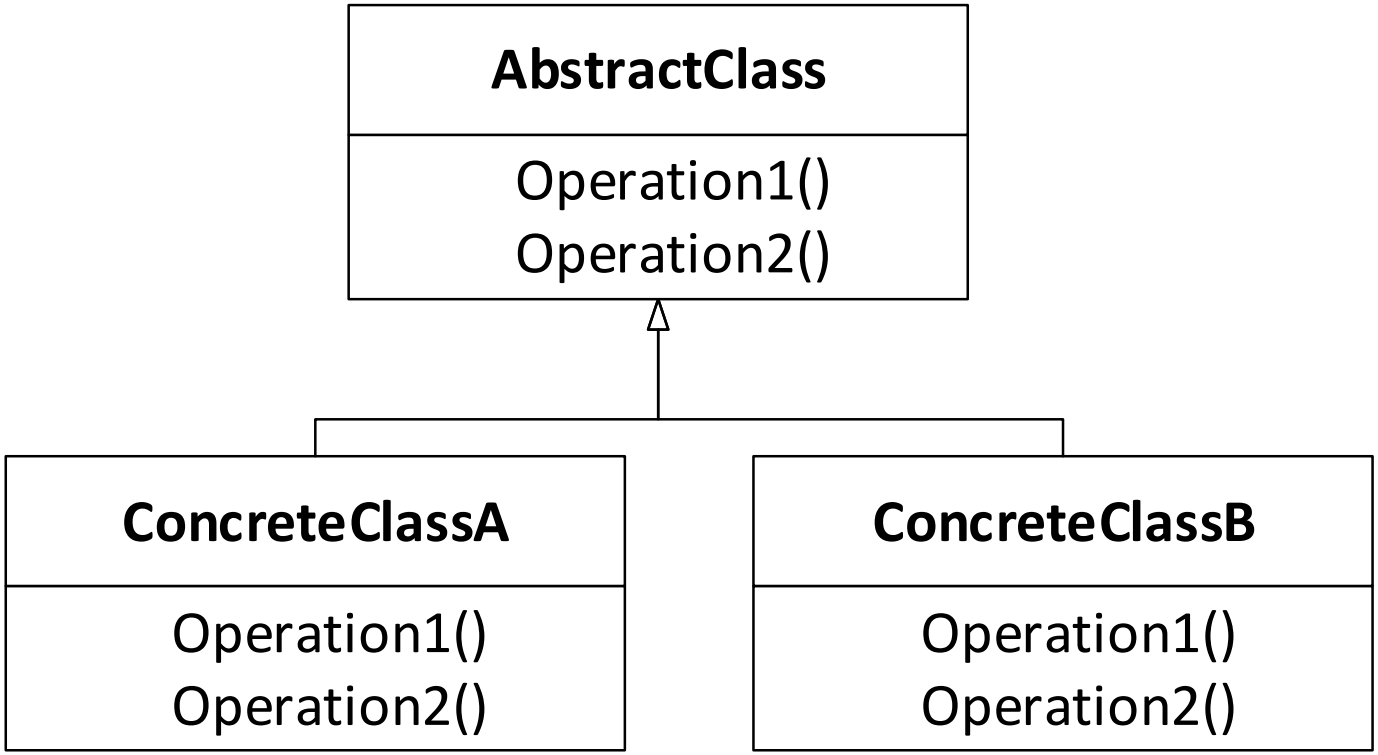


```
//主程序
    main(int argc, char *argv[]) {
    WindowsDisplay display;
    Monitor monitor(&display);
    while (running()) {
        //获取负载信息
        monitor.getLoad();
        //获取内存大小信息
        monitor.getTotalMemory();
        //获取内存使用信息
        monitor.getUsedMemory();
        //获取网络延迟信息
        monitor.getNetworkLatency();
        //信息输出
        monitor.show();
        sleep(1000);
    }
}
```

模板方法
Template Method

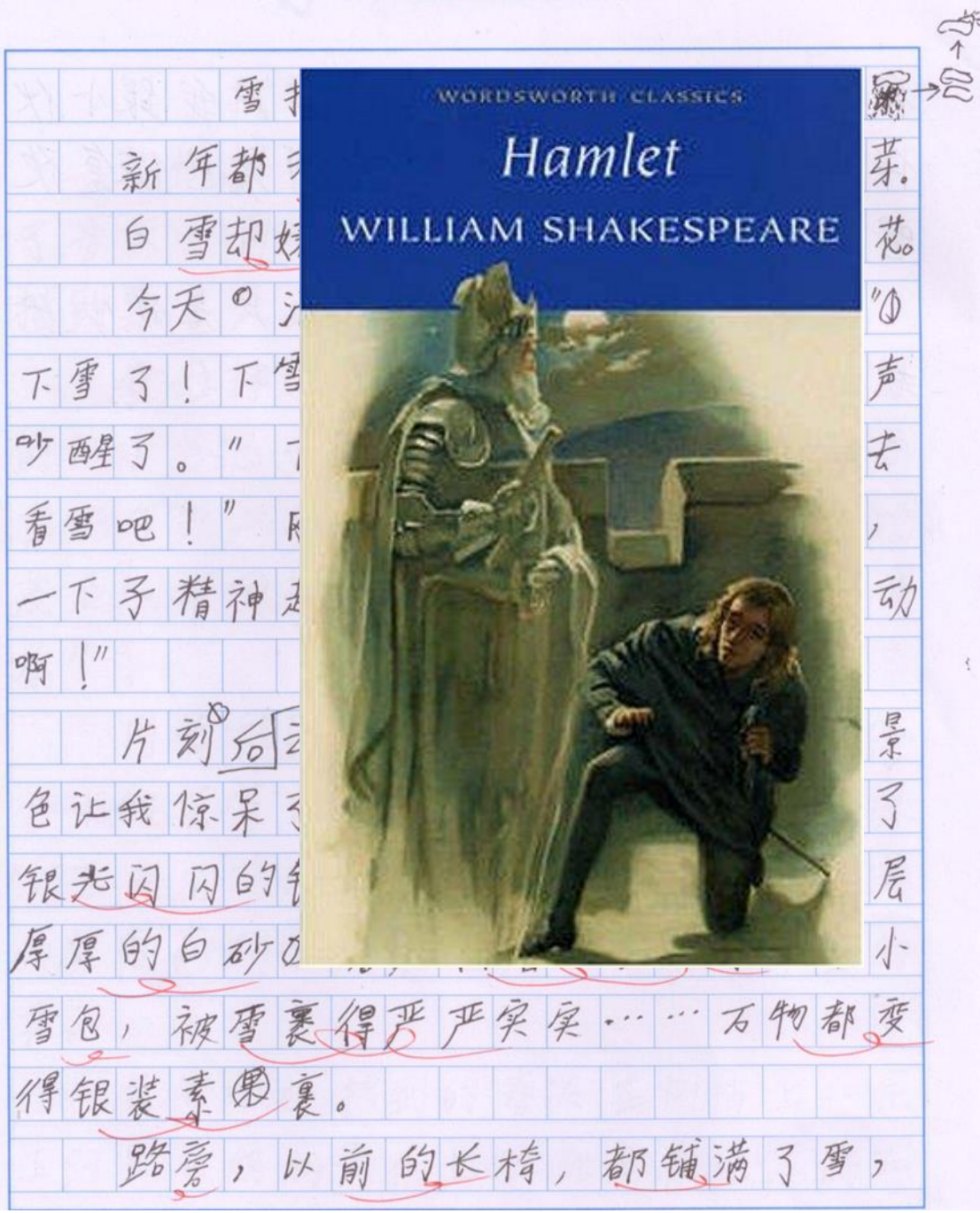
模板方法

- 在接口的一个方法中定义算法的骨架
- 将一些步骤的实现延迟到子类中
- 使得子类可以在不改变算法结构的情况下，重新定义算法中的某些步骤。



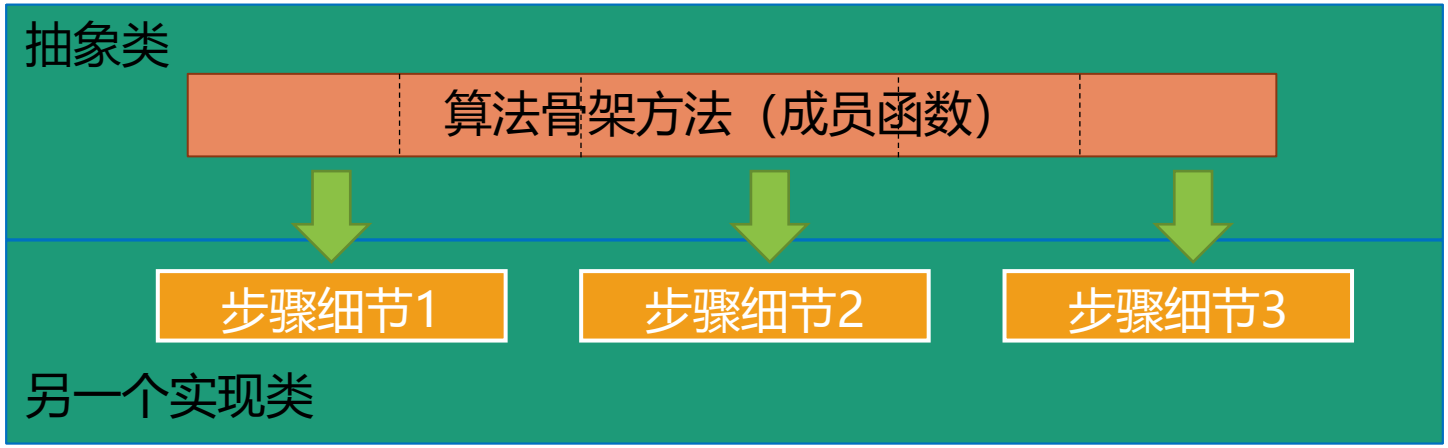
模板方法

- 还记得我们小学的时候是怎么写作文的吗?
- 我们学习一些经典的行文结构，然后在不同的题目下分别组织语言。
- 其实不光小学生作文可以这么写，程序设计也可以如此。

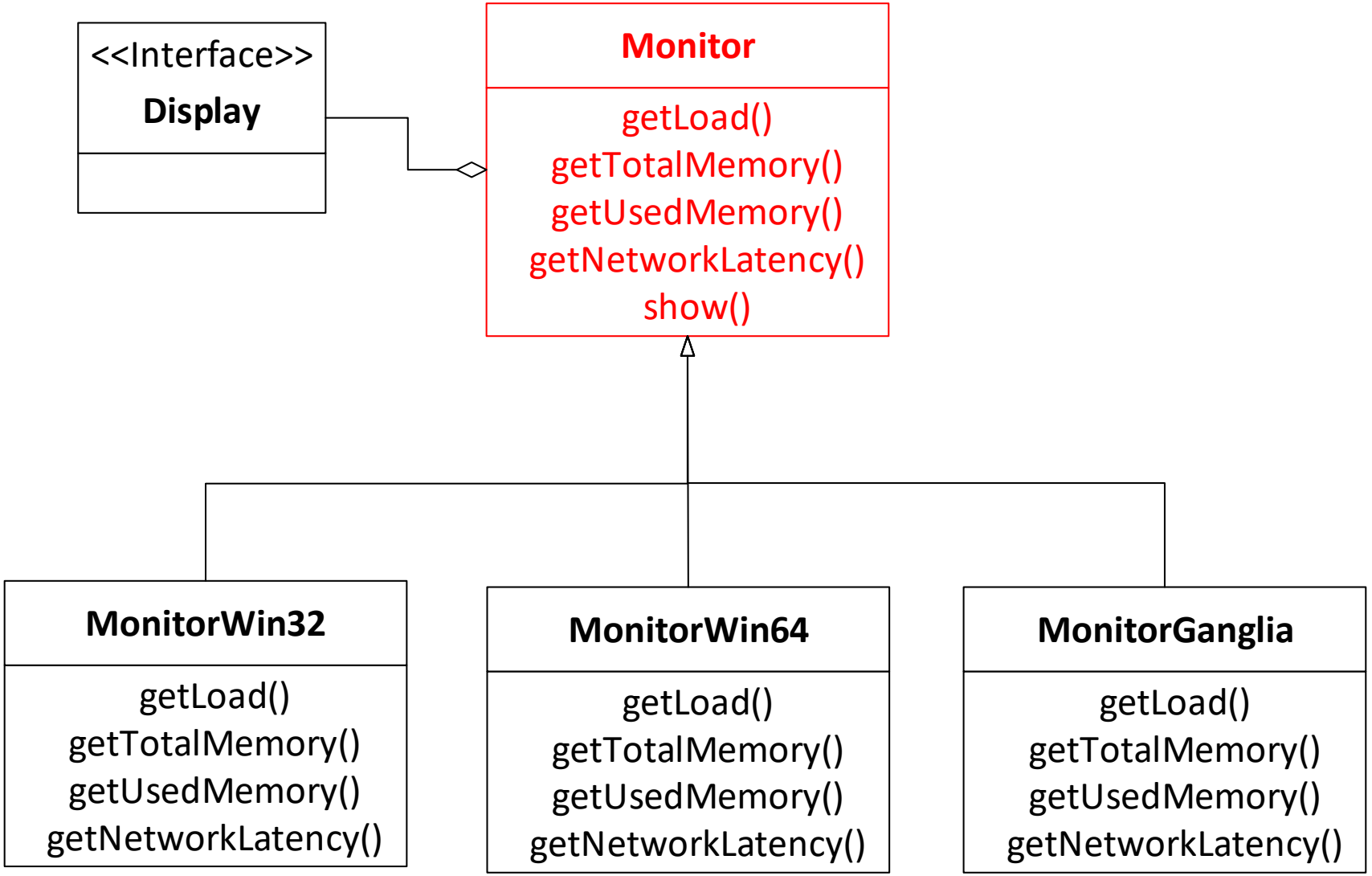


模板方法

- 抽象类（父类）定义算法的骨架
- 算法的细节由实现类（子类）负责实现
- 在使用时，调用抽象类的算法骨架方法，再由这个方法来根据需要调用具体类的实现细节
- 当拓展一个新的实现类时，重新继承与实现即可，无需对已有的实现类进行修改



实现Monitor

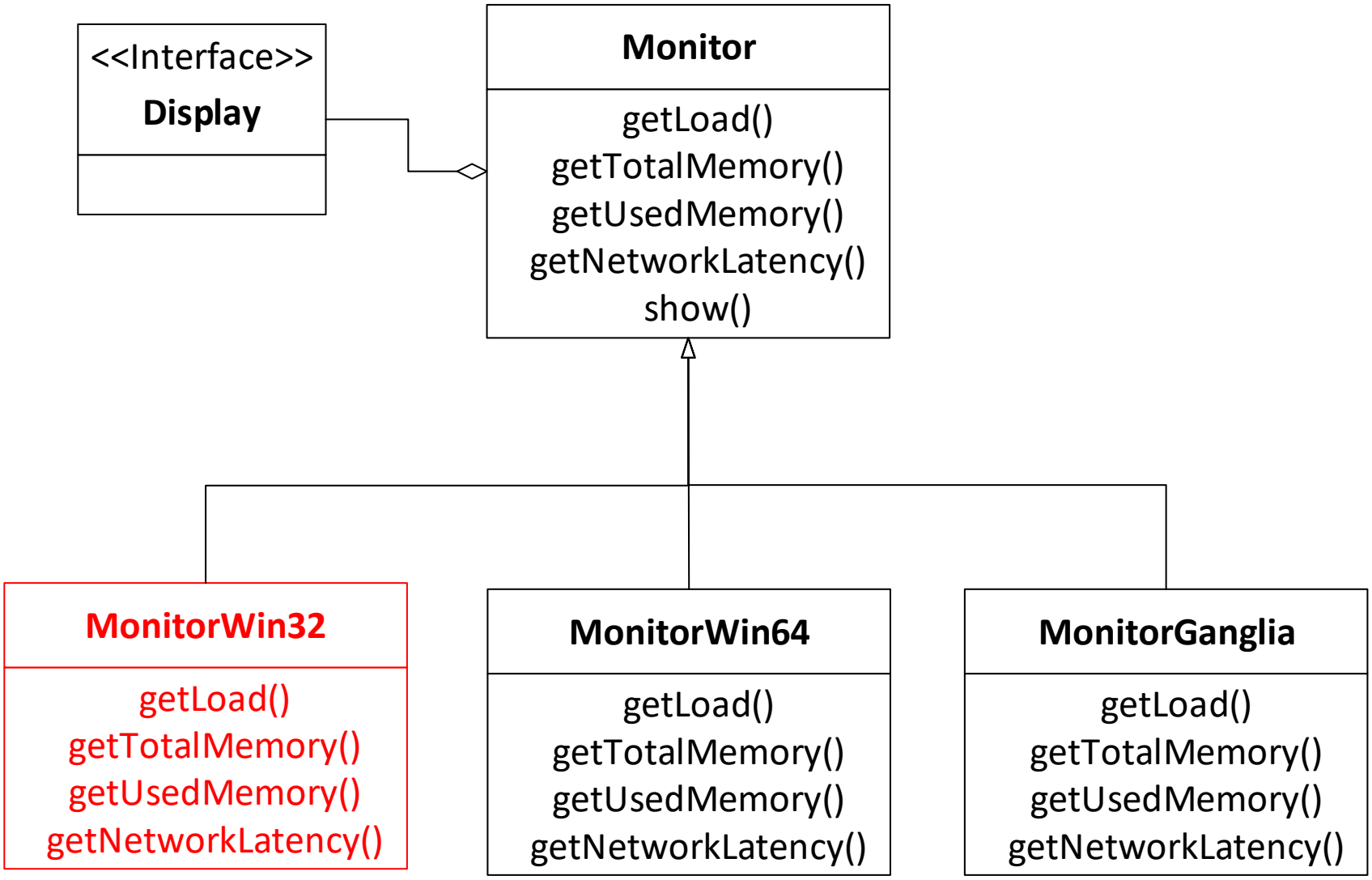

<<Interface>>
Display
Monitor
getLoad()
getTotalMemory()
getUsedMemory()
getNetworkLatency()
show()
MonitorWin32
getLoad()
getTotalMemory()
getUsedMemory()
getNetworkLatency()
MonitorWin64
getLoad()
getTotalMemory()
getUsedMemory()
getNetworkLatency()
MonitorGanglia
getLoad()
getTotalMemory()
getUsedMemory()
getNetworkLatency()

代码实现

```
class Monitor {
public:
    virtual void getLoad() = 0;
    virtual void getTotalMemory() = 0;
    virtual void getUsedMemory() = 0;
    virtual void getNetworkLatency() = 0;
    Monitor(Display *display);
    virtual ~Monitor();
    void show();
protected:
    //用以存储信息的成员变量
    float load, latency;
    long totalMemory, usedMemory;
    Display* m_display;
};

//组合一个Display接口来进行结果展示
void Monitor::show() {
    m_display -> show(load, totalMemory,
                      usedMemory, latency);
}
```

纯虚函数无需实现

实现MonitorWin32



代码实现

```
//通过具体实现抽象的模板来完成Win32系统下的监控器实现
class MonitorWin32::public Monitor {
public:
    void getLoad();
    void getTotalMemory();
    void getUsedMemory();
    void getNetworkLatency();
};
//Win32的getLoad()的具体实现
void MonitorWin32::getLoad() {
    …
    load = …;
    …
}
//Win32的getTotalMemory()的具体实现
void MonitorWin32::getTotalMemory() {
    …
    totalMemory = …;
    …
}
…
```

代码实现

抽象类指针指向实现类对象
通过指针实现多态

```
int main(int argc, char *argv[]) {
    WindowsDisplay display;
    //创建MonitorWin32模式的监控器，并用基类指针来调用方法
    Monitor* monitor = new MonitorWin32(&display);
    while (running()) {
        monitor->getLoad();              //获取负载信息
        monitor->getTotalMemory();       //获取内存大小信息
        monitor->getUsedMemory();        //获取内存使用信息
        monitor->getNetworkLatency(); //获取网络延迟信息
        monitor->show();                 //信息输出
        sleep(1000);
    }
    //释放
    delete monitor;
}
```

会根据monitor指针实际指向的类型
进行函数调用

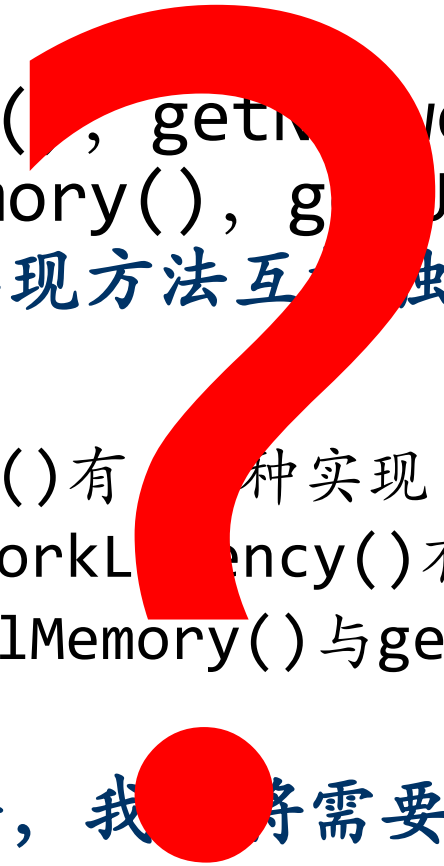
针对接口编程

- 模板方法其实就是一种针对接口编程的设计
- 通过抽象出“抽象概念”，设计出描述这个抽象概念的抽象类，或称为“接口类”，这个类有一系列的（纯）虚函数，描述了这个类的“接口”
- 对这个接口类进行继承并实现这些（纯）虚函数，从而形成这个抽象概念的“实现类”——实现可以有很多种
- 在使用这个概念的时候，我们使用接口类来引用这个概念，而不直接使用实现类，从而避免实现类的改变造成整个程序的大规模变化

开放封闭原则

- **模板方法很好的体现了开放封闭原则**
  - 对扩展开放，有新需求或变化时，可以方便地现有代码进行扩展，而无需整体变动
  - 对修改封闭，新的扩展类一旦设计完成，可以独立完成其工作，同样不需要整体变动
- **开放封闭原则的核心就是在结构层面上解耦，对抽象进行编程，而不对具体编程**
  - 抽象结构是简单与稳定的
  - 具体实现是复杂与多变的

需求变化

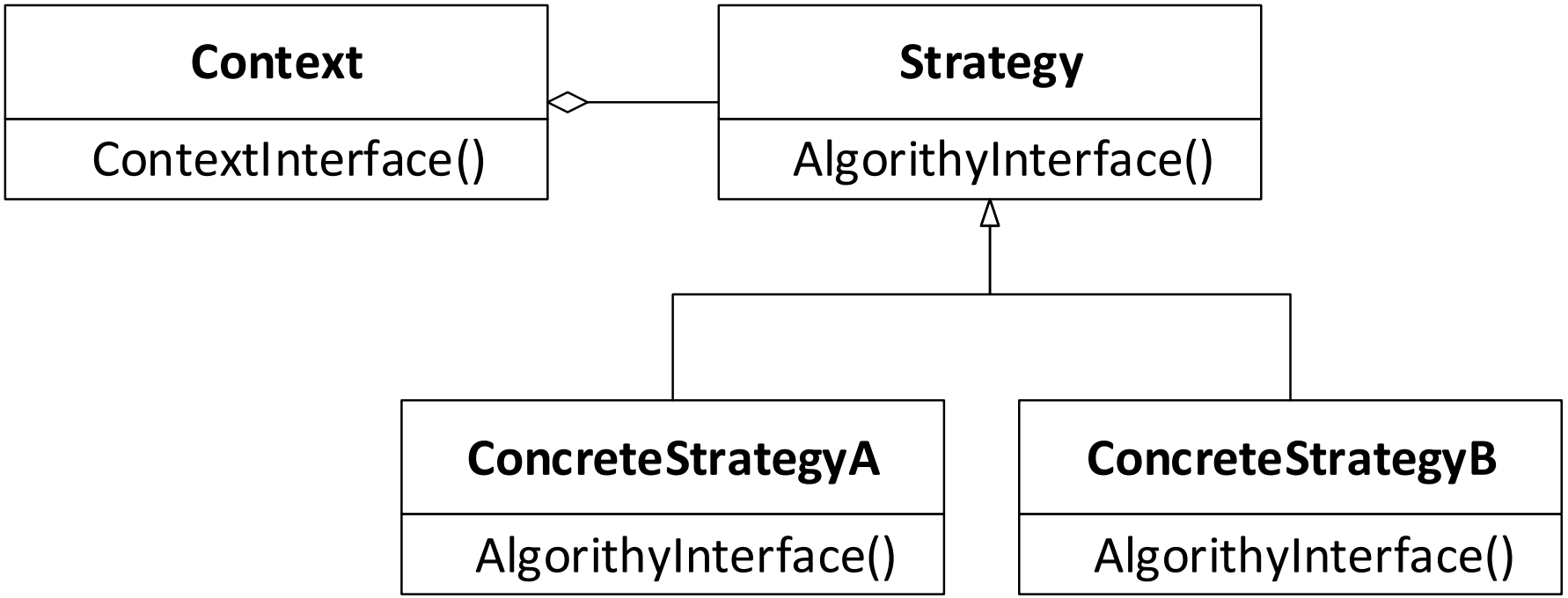


- 如果getLoad()，getNetworkLatency()，getTotalMemory()，getUsedMemory()这几个函数接口的实现方法互相独立
- 假设
  - getLoad()有 n 种实现
  - getNetworkLatency()有 m 种实现
  - getTotalMemory()与getUsedMemory()有 k 种实现
- 使用模板方法，我们将需要实现 n*m*k 个子类
- 这样充斥大量冗余的实现方式是不可取的

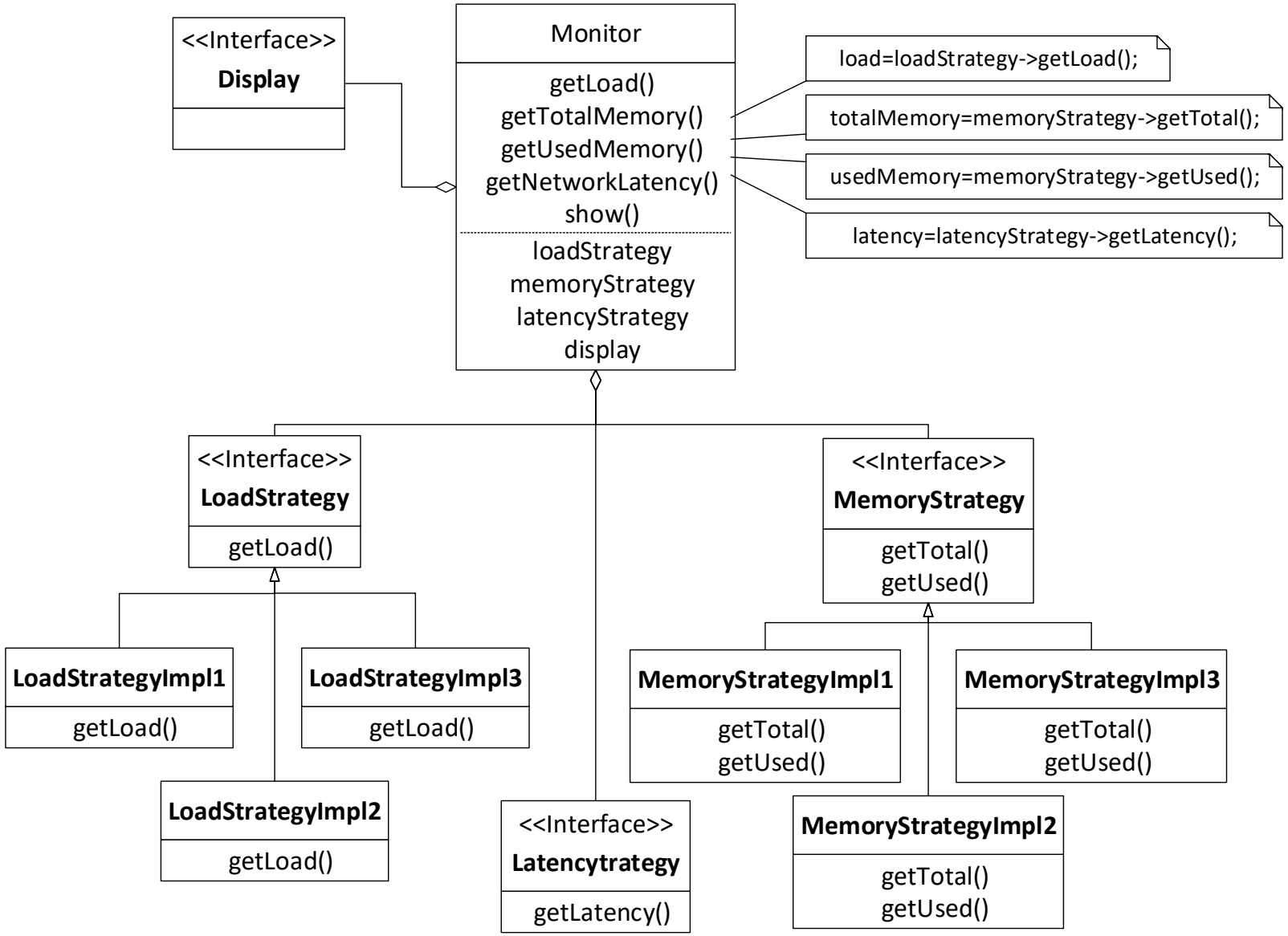
策略模式
Strategy

策略（Strategy）模式

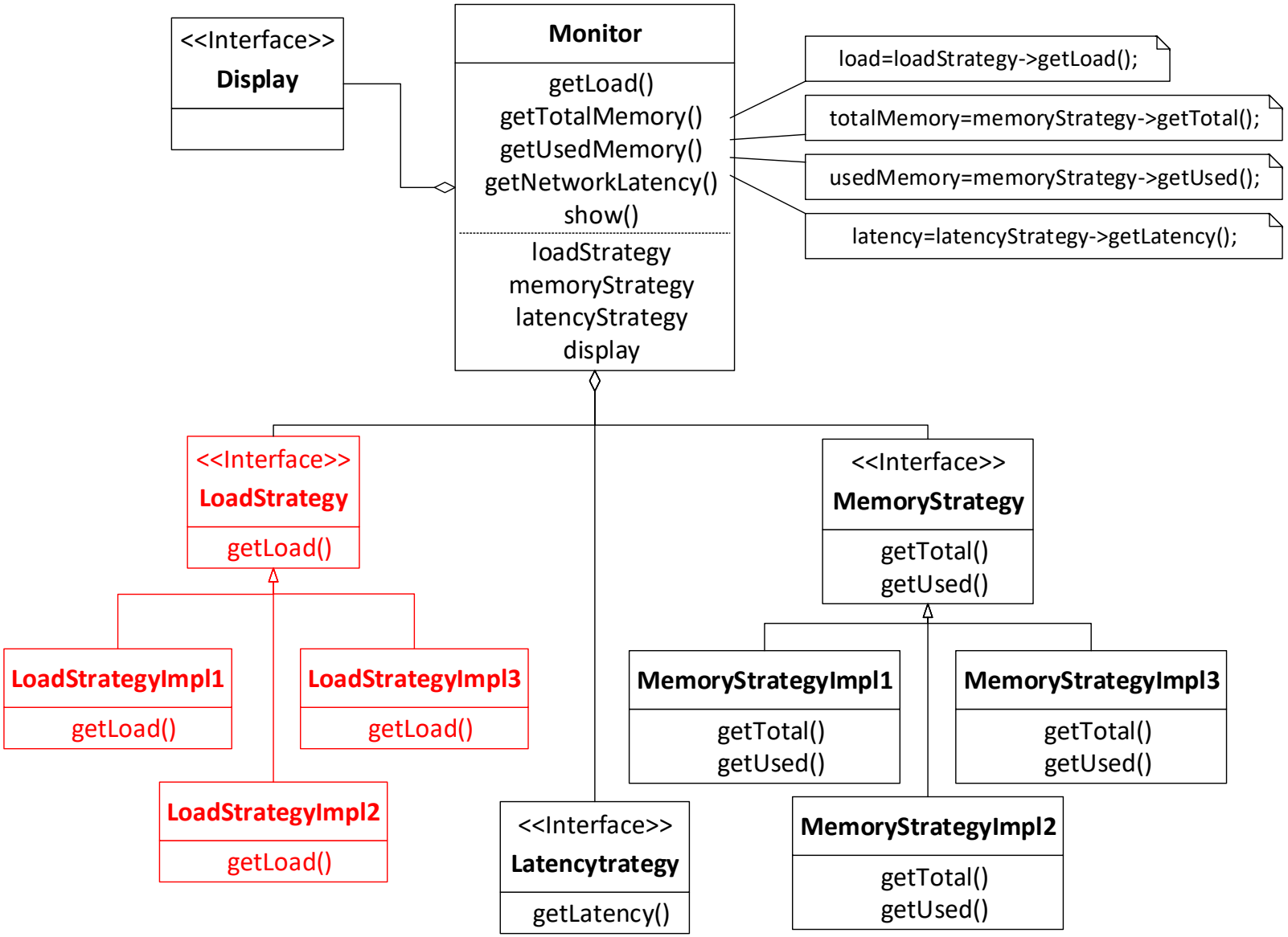
- 定义一系列算法并加以封装，使得这些算法可以互相替换



具体化到我们的问题


<<Interface>>
Display
Monitor
getLoad()
getTotalMemory()
getUsedMemory()
getNetworkLatency()
show()
loadStrategy
memoryStrategy
latencyStrategy
display
load=loadStrategy->getLoad();
totalMemory=memoryStrategy->getTotal();
usedMemory=memoryStrategy->getUsed();
latency=latencyStrategy->getLatency();
<<Interface>>
LoadStrategy
getLoad()
<<Interface>>
MemoryStrategy
getTotal()
getUsed()
LoadStrategyImpl1
getLoad()
LoadStrategyImpl3
getLoad()
MemoryStrategyImpl1
getTotal()
getUsed()
MemoryStrategyImpl3
getTotal()
getUsed()
LoadStrategyImpl2
getLoad()
<<Interface>>
Latencytrategy
getLatency()
MemoryStrategyImpl2
getTotal()
getUsed()

实现LoadStrategy



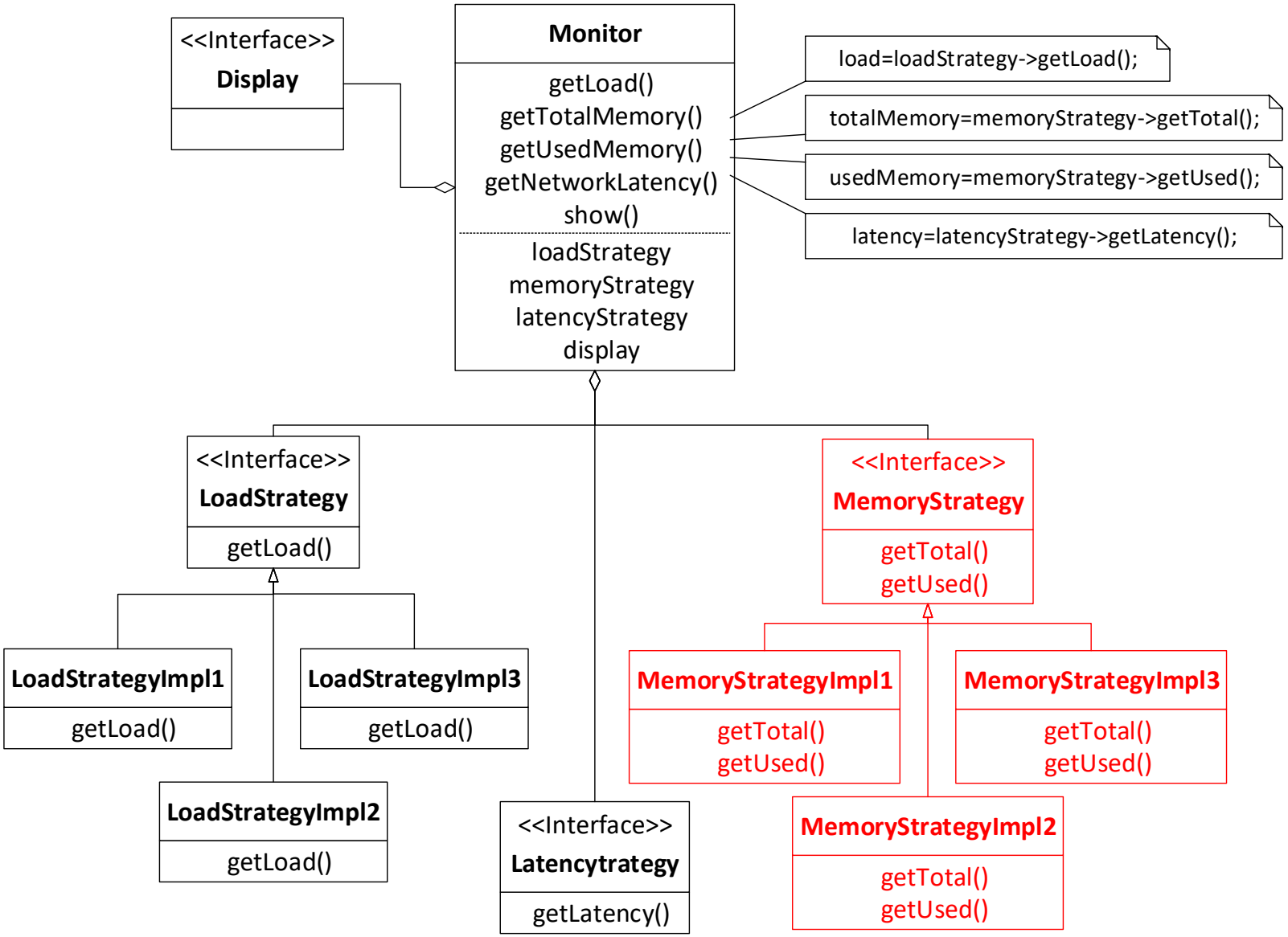
代码实现

所谓strategy模式就是抽象分类的方法不同而已

```
//负载策略基类
class LoadStrategy {
public:
    virtual float getLoad() = 0;
};
//负载算法一具体实现
class LoadStrategyImpl1 : public LoadStrategy {
public:
    float getLoad() { //获取负载数值
        …
        return load;
    }
};
//负载算法二具体实现
class LoadStrategyImpl2 : public LoadStrategy {
public:
    float getLoad() { //获取负载数值
        …
        return load;
    }
};
```

strategy**基类**里面是纯虚函数

实现MemoryStrategy

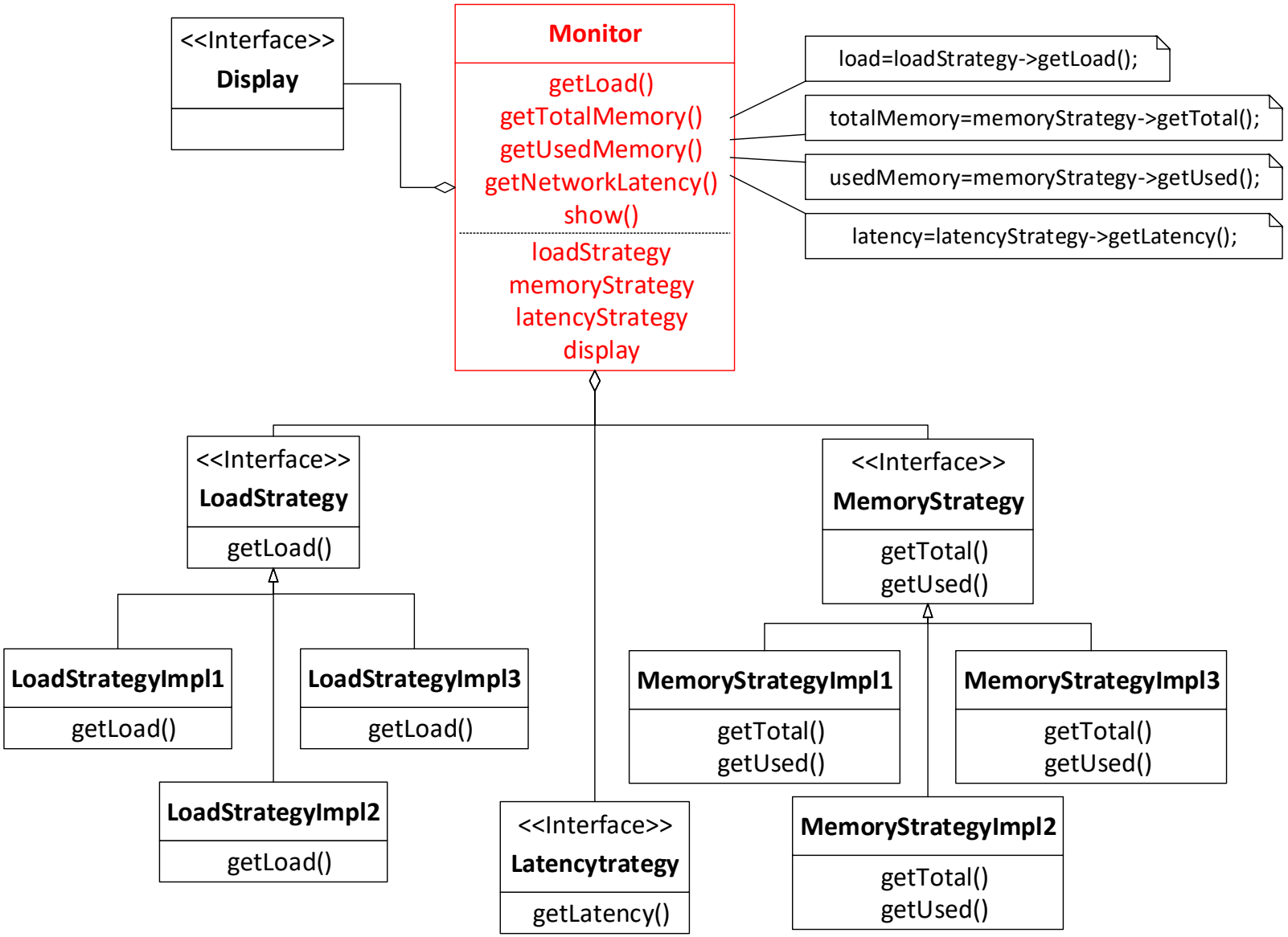


代码实现

```
//内存策略基类
class MemoryStrategy {
public:
    virtual long getTotal() = 0;
    virtual long getUsed() = 0;
};
//内存算法一具体实现
class MemoryStrategyImpl1 : public MemoryStrategy {
public:
    long getTotal() { //获取内存信息
        …
        return total;
    }
    long getUsed() { //获取已用内存数值
        …
        return used;
    }
};
//内存算法二具体实现
class MemoryStrategyImpl2 : public MemoryStrategy {
    …
}
```

strategy在memory里面使用的情况

实现Monitor


<<Interface>>
Display
Monitor
getLoad()
getTotalMemory()
getUsedMemory()
getNetworkLatency()
show()
loadStrategy
memoryStrategy
latencyStrategy
display
load=loadStrategy->getLoad();
totalMemory=memoryStrategy->getTotal();
usedMemory=memoryStrategy->getUsed();
latency=latencyStrategy->getLatency();
<<Interface>>
LoadStrategy
getLoad()
<<Interface>>
MemoryStrategy
getTotal()
getUsed()
LoadStrategyImpl1
getLoad()
LoadStrategyImpl3
getLoad()
MemoryStrategyImpl1
getTotal()
getUsed()
MemoryStrategyImpl3
getTotal()
getUsed()
LoadStrategyImpl2
getLoad()
<<Interface>>
Latencytrategy
getLatency()
MemoryStrategyImpl2
getTotal()
getUsed()

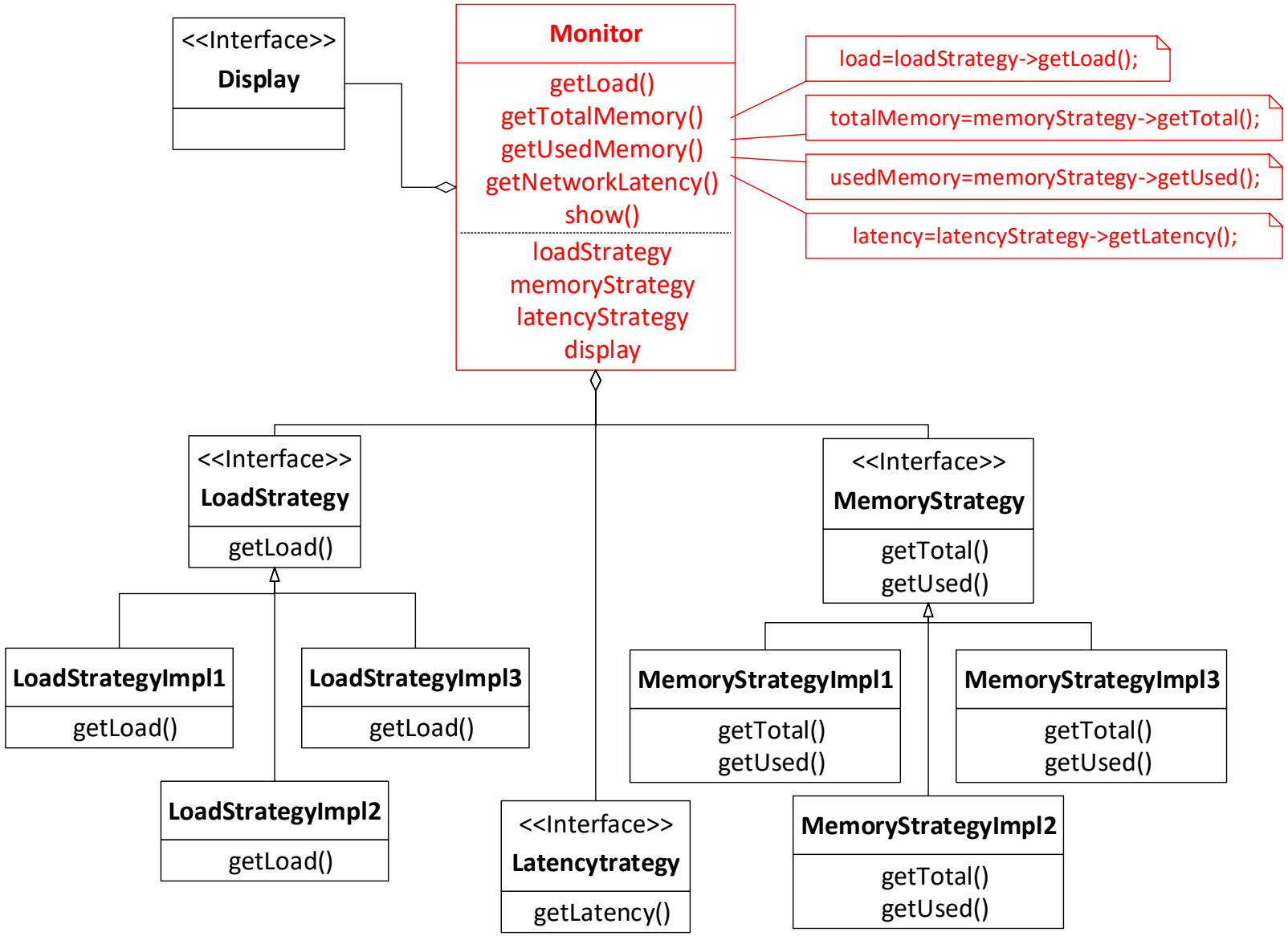
代码实现

```
class Monitor { //监控器类
public:
    //监控器就是各个策略类的组合
    Monitor(LoadStrategy *loadStrategy,
          MemoryStrategy *memStrategy,
          LatencyStrategy *latencyStrategy
          Display *display);
    void getLoad();
    void getTotalMemory();
    void getUsedMemory();
    void getNetworkLatency();
    void show();
private:
    //获取各类不同信息的策略类
    LoadStrategy *m_loadStrategy;
    MemoryStrategy *m_memStrategy;
    LatencyStrategy *m_latencyStrategy;
    //用以存储信息的成员变量
    float load, latency;
    long totalMemory, usedMemory;
    Display *m_display;
};
```

代码实现

```
//构造函数初始化所有的策略和参数
Monitor::Monitor(LoadStrategy *loadStrategy,
                 MemoryStrategy *memStrategy,
                 LatencyStrategy *latencyStrategy
                 Display *display) :
                     m_loadStrategy(loadStrategy),
                     m_memStrategy(memStrategy),
                     m_latencyStrategy(latencyStrategy),
                     m_display(display),
                     load(0.0), latency(0.0),
                     totalMemory(0), usedMemory(0) {}

//统一的输出接口输出不同策略类获得的系统信息
void Monitor::show() {
    m_display -> show(   load,
                         totalMemory,
                         usedMemory,
                         latency);
}
```

实现Monitor


<<Interface>>
Display
Monitor
getLoad()
getTotalMemory()
getUsedMemory()
getNetworkLatency()
show()
loadStrategy
memoryStrategy
latencyStrategy
display
load=loadStrategy->getLoad();
totalMemory=memoryStrategy->getTotal();
usedMemory=memoryStrategy->getUsed();
latency=latencyStrategy->getLatency();
<<Interface>>
LoadStrategy
getLoad()
<<Interface>>
MemoryStrategy
getTotal()
getUsed()
LoadStrategyImpl1
getLoad()
LoadStrategyImpl3
getLoad()
MemoryStrategyImpl1
getTotal()
getUsed()
MemoryStrategyImpl3
getTotal()
getUsed()
LoadStrategyImpl2
getLoad()
<<Interface>>
Latencytrategy
getLatency()
MemoryStrategyImpl2
getTotal()
getUsed()

代码实现

```
//统一的接口来获取负载
void Monitor::getLoad() {
  load = m_loadStrategy -> getLoad();
}
//统一的接口来获取总内存
void Monitor::getTotalMemory() {
  totalMemory = m_memStrategy -> getTotal();
}
//统一的接口来获取已用内存
void Monitor::getUsedMemory() {
  usedMemory = m_memStrategy -> getUsed();
}
//统一的接口来获取网络延迟信息
void Monitor::getNetworkLatency() {
  latency = m_latencyStrategy -> getLatency();
}
```

代码实现

```
int main(int argc, char *argv[]){
    //为每个策略的选择具体的实现算法，并创建监控器类
    GangliaLoadStrategy loadStrategy;
    WinMemoryStrategy   memoryStrategy;
    PingLatencyStrategy latencyStrategy;
    WindowsDisplay      display;
    //具体构建过程是将每个策略的具体算法类传入构造函数
    Monitor monitor( &loadStrategy,
                     &memoryStrategy,
                     &latencyStrategy,
                     &display);
    while (running()) {
        //统一的接口获取系统信息
        monitor.getLoad();
        monitor.getTotalMemory();
        monitor.getUsedMemory();
        monitor.getNetworkLatency();
        //统一的接口输出系统信息
        monitor.show();
        sleep(1000);
    }
}
```

调用过程

```
int main(int argc, char *argv[]) {
    …
    monitor -> getLoad();
}
```

统一的策略调用接口

```
void Monitor::getLoad() {
    load = m_loadStrategy -> getLoad();
}
```

m_loadStrategy在初始化时实现

```
float LoadStrategyImpl1::getLoad()
{
    …
}
```

```
float LoadStrategyImpl2::getLoad()
{
    …
}
```

现在的类数量

- 假设
  - getLoad()有 n 种实现
  - getNetworkLatency()有 m 种实现
  - getTotalMemory()与getUsedMemory()有 k 种实现
- 我们需要实现
  - 1个Monitor类
  - 3个抽象策略类（接口）
  - n+m+k个策略实现类（实现）
- 策略模式极大的降低了代码冗余，(n+m+k+3+1) vs (n*m*k+1)

单一责任原则

- 策略模式很好的体现了单一责任原则
  - 一个类（接口）只负责一项职责
  - 不要存在多于一个导致类变更的原因
- 如果一个类承担的职责过多，职责之间的耦合度会很大
  - 职责的变化可能会削弱或者抑制这个类完成其他职责的能力
  - 多变的场景会使得整体程序的设计遭受破坏，维护难度增大
- 单一职责原则的核心就是在功能层面上解耦

模板方法VS策略模式

所谓的“实现类”就是有具体功能的子类

- 当我们需要实现一个新的getTotalMemory()+getUsedMemory()

模板方法	策略模式
• 针对所有getLoad()、getNetworkLatency()的组合，都要实现一组新的子类 • 需要实现新子类（实现类）：n*m 个	• 无需修改LoadStrategy和LatencyStrategy，只要实现一个新的MemoryStrategy实现类即可 • 需要实现新子类（实现类）：1 个

模板方法VS策略

模板方法

- 定义算法的骨架，而将具体实现步骤延迟到子类中。
- 子类可以不改变算法的结构即可重定义该算法的某些特定步骤
- 优先继承行为，重视功能的抽象与归纳
- 优点：
  - 基类高度抽象统一，逻辑简洁明了
  - 子类之间关联不紧密时易于简单快速实现
  - 封装性好，实现类内部不会对外暴露
- 弊端：
  - 接口同时负责所有的功能（算法）
  - 任何算法的修改都导致整个实现类的变化定义一系列的算法，把它们一个个封装起来,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

模板方法VS策略

策略模式

- 定义一系列的算法，把它们一个个封装起来,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
- 优先组合行为，重视功能的划分与组合
- 优点：
  - 每个策略只负责一个功能，易于拓展。
  - 算法的修改被限制在单个策略类的变化中，任何算法的修改对整体不造成影响
- 弊端：
  - 在功能较多的情况下结构复杂
  - 策略组合时对外暴露，封装性相对较差

要传入strategy组合作为参数

模板方法VS策略

- 模板方法和策略模式都是解决算法多样性对代码结构冲击的问题。业务相对简单时，策略模式和模板方法几乎等效。

- 模式方法更加侧重于逻辑复杂但结构稳定的场景，尤其是其中的某些步骤（部分功能）变化剧烈且没有相互关联。

- 策略模式则适用于算法（功能）本身灵活多变的场景，且多种算法之间需要协同工作。

再从一个简单的实例开始

- 实例：对考试结果进行统计分析(及格率)

```
int main(int argc, char *argv[]) {

    float scores[STUDENT_COUNT];
    int passed = 0;

    for (int i = 0; i != STUDENT_COUNT; i++) {
        if (scores[i] >= 60)
            passed ++;
    }

    cout << "passing rate = "
       << (float)passed / STUDENT_COUNT << endl;

    return EXIT_SUCCESS;
}
```

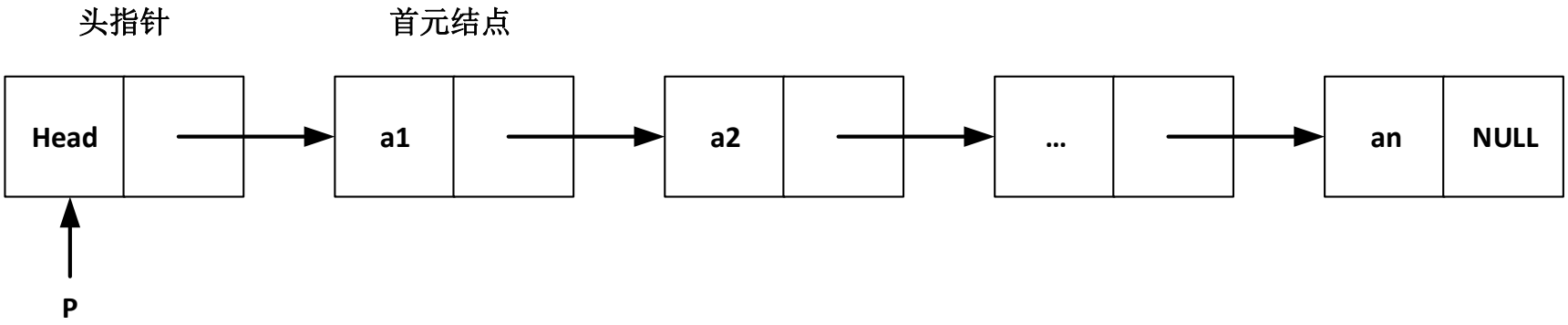
责任分解

- 把“分析”单独作为一个功能

```
void analyze(float *scores, int student_count) {
    int passed = 0;

    for (int i = 0; i != student_count; i++) {
        if (scores[i] >= 60)
            passed ++;
    }

    cout << "passing rate = "
            << (float)passed / student_count << endl;
}
```

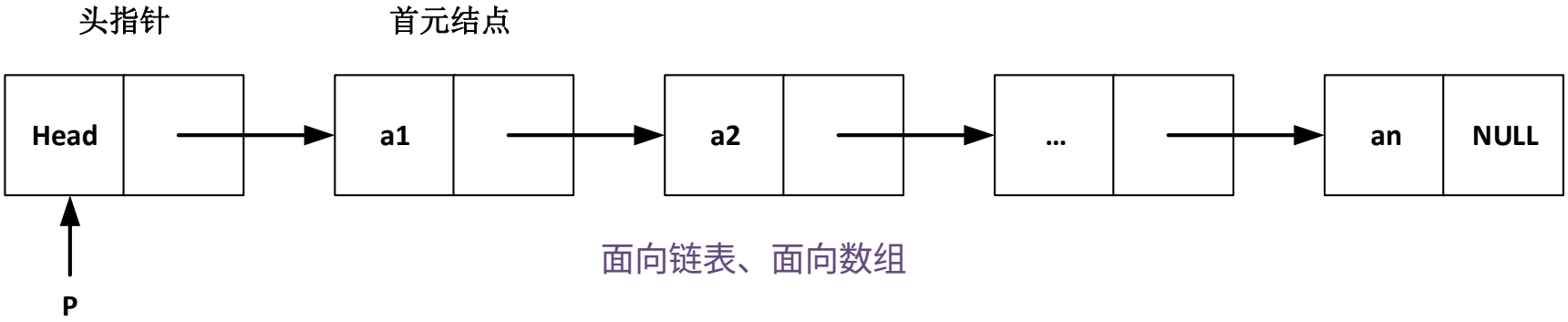
链表替代数组

- 如果成绩是用单向非循环链表取代数组进行存储?




```
struct Student {
    float score;
    Student* next;
};

…

Student *head;
```

链表替代数组



面向链表、面向数组

```
void analyze(Student *head) {
    int passed = 0, count = 0;

    for (Student *p = head; p != NULL; p = p -> next) {
        if (p -> score >= 60)
            passed ++;
        count ++;
    }

    cout << "passing rate = "
         << (float)passed / count << endl;
}
```

“遍历”

- 如何实现与底层数据结构无关的统一算法接口？
- 变与不变
  - 需要遍历所有学生的成绩，即算法是不变的
  - 不希望绑定在某种存储方式，即底层数据结构是变的
- 分离“变”（存储）与“不变”（访问）
  - 把数据“访问”设计为一个接口
  - 针对不同的“存储”完成这个接口的不同实现

```
//数组下标遍历
for (int i = 0; i != STUDENT_COUNT; i++) {
  //access item by i
}
```

```
//指针遍历
for (Student *p = scores; p != NULL; p = p -> next) {
  //access item by p
}
```

迭代器模式
Iterator

迭代器模式

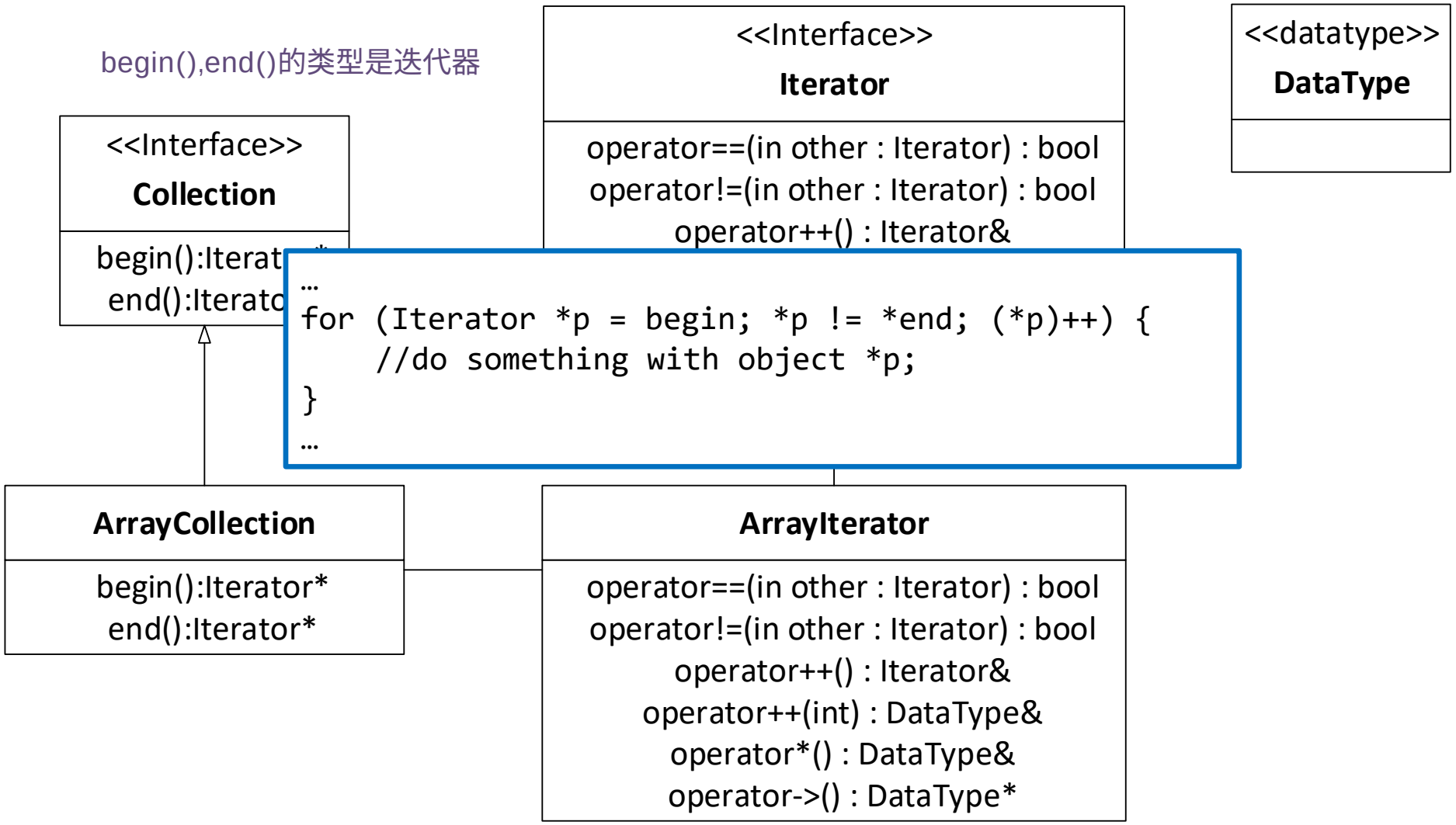
- 提供一种方法顺序访问一个聚合对象中各个元素
- 又不需暴露该对象的内部表示——与对象的内部数据结构形式无关（数组还是链表）
- 具体实现相当于用模板方法构建迭代器和数据存储基类，为每种单独的数据结构都实现其独有的迭代器和存储类
- 但对于上层算法，算法的执行只依赖于抽象的迭代器接口，而无需关注最底层的具体数据结构

统一数组、链表、vector

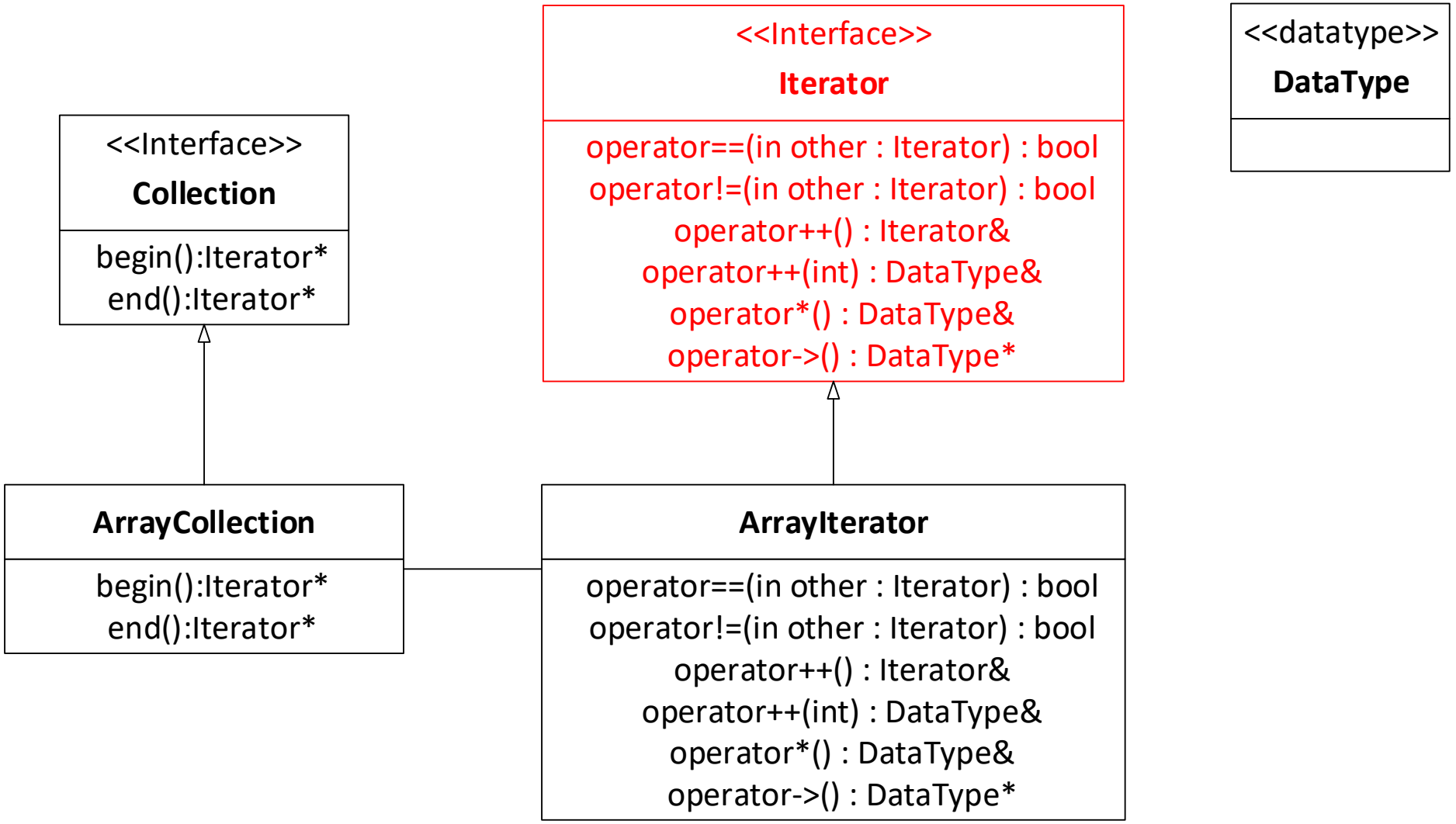
迭代器模式

->[]是取对象
*[]是取值

begin(),end()的类型是迭代器



实现Iterator基类



迭代器

- 把数据“访问”设计为一个统一接口，形成迭代器
- 这个迭代器可以套接在任意的数据结构上

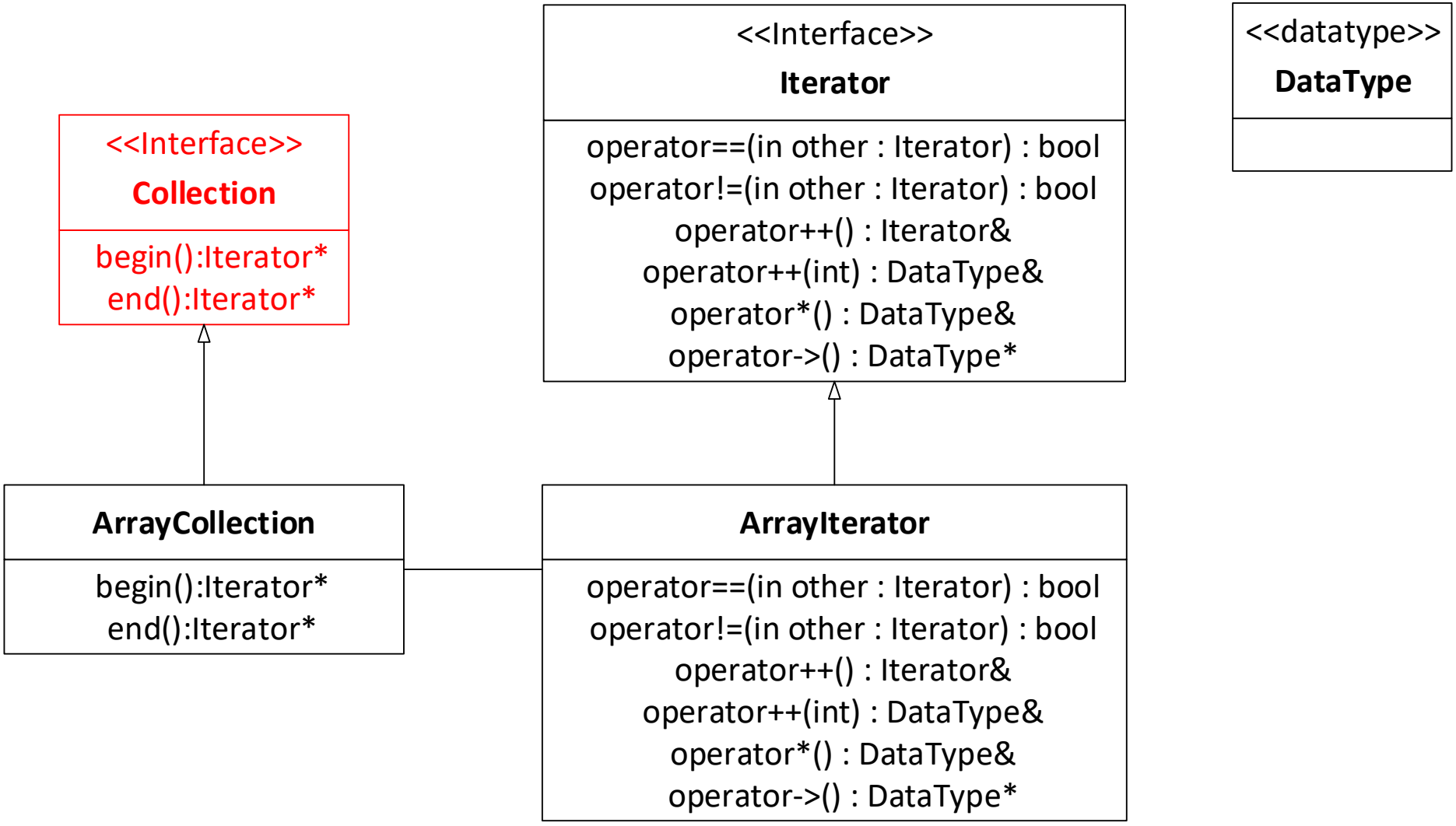
```
//迭代器基类
class Iterator {
public:
    virtual ~Iterator() { }
    virtual Iterator& operator++() = 0;
    virtual float& operator++(int) = 0;
    virtual float& operator*() = 0;
    virtual float* operator->() = 0;
    virtual bool operator!=(const Iterator &other) const = 0;
    bool operator==(const Iterator &other) const {
        return !(*this != other);
    }
};
```

使用迭代器

- 用“迭代器”作为参数传递，参与上层算法构建
- 这样算法构建就可以不依赖于底层的数据结构

```
void analyze(Iterator* begin, Iterator* end) {
    int passed = 0, count = 0;

    for (Iterator* p = begin; *p != *end; (*p)++) {
        if (**p >= 60)
            passed ++;
        count ++;
    }

    cout << "passing rate =" << (float)passed / count << endl;
}
```

实现Collection基类



实现Collection基类

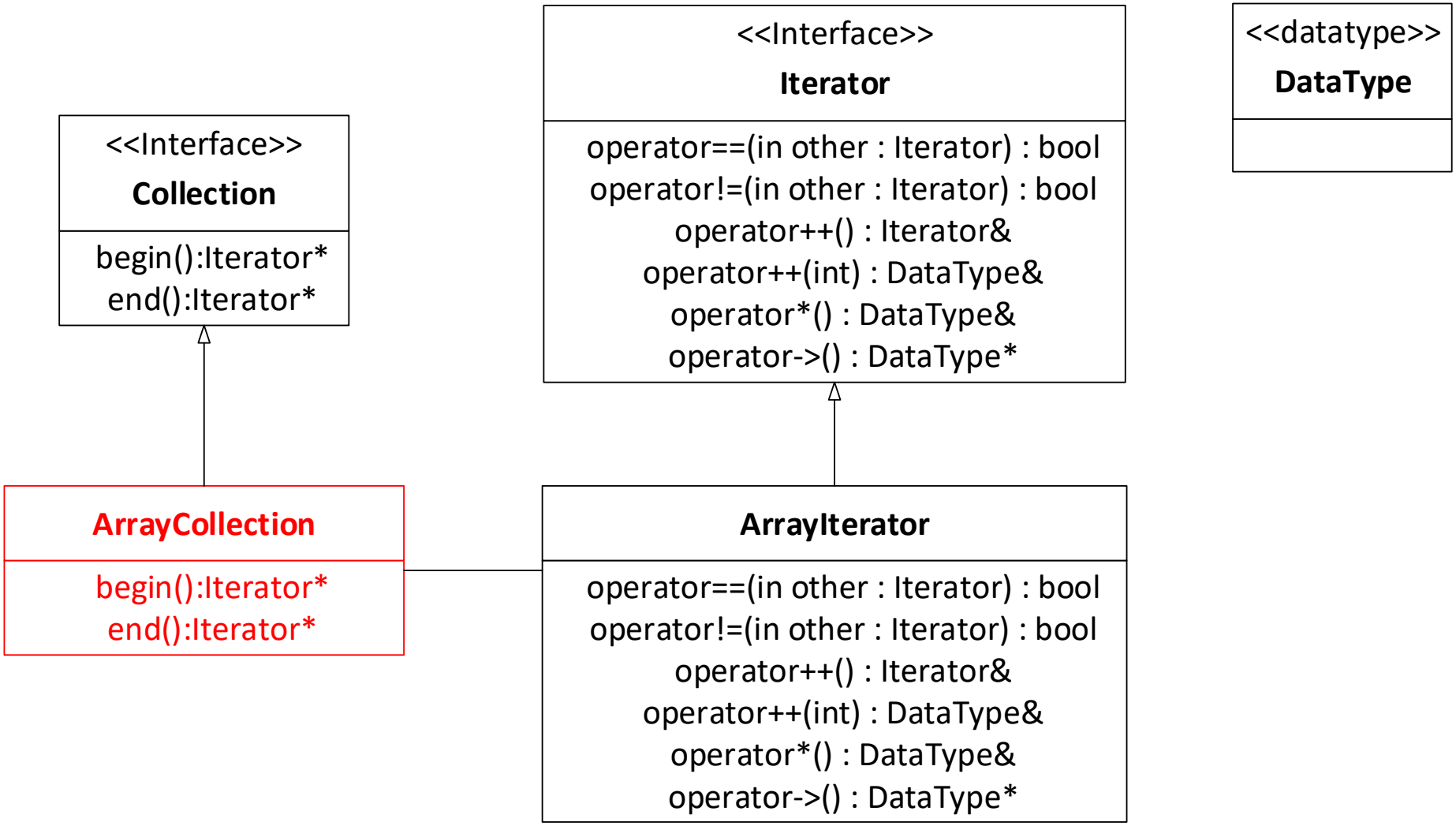
- 定义数据的存储结构基类Collection
- 需要给“存储”对象一个约束
  - 能够返回代表“头”和“尾”的迭代器
  - 使用“左闭右开区间”，即[begin, end)

```
class Collection {
public:
    virtual ~Collection() { }
    virtual Iterator* begin() const = 0;
    virtual Iterator* end() const = 0;
    virtual int size() = 0;
};
```

const类型表示返回值不能更改？？

（回去复习！！！！

实现基于数组的Collection



实现基于数组的Collection

```
class ArrayCollection : public Collection { //底层为数组的存储结构类
    friend class ArrayIterator; //friend可以使得配套的迭代器类可以访问数据
    float* _data;
    int _size;
public:
    ArrayCollection() : _size(10){_data = new float[_size]; }
    ArrayCollection(int size, float* data) : _size(size) {
        _data = new float[_size];          //开辟数组空间用以存储数据
        for (int i = 0; i < size; i++)
            *(_data+i) = *(data+i);
    }
    ~ArrayCollection() { delete[] _data; }
    int size() { return _size; }
    Iterator* begin() const;
    Iterator* end() const;
};
Iterator* ArrayCollection::begin() const {    //头迭代器，并放入相应数据
    return new ArrayIterator(_data, 0);
}
Iterator* ArrayCollection::end() const {      //尾迭代器，并放入相应数据
    return new ArrayIterator(_data, _size);
}
```

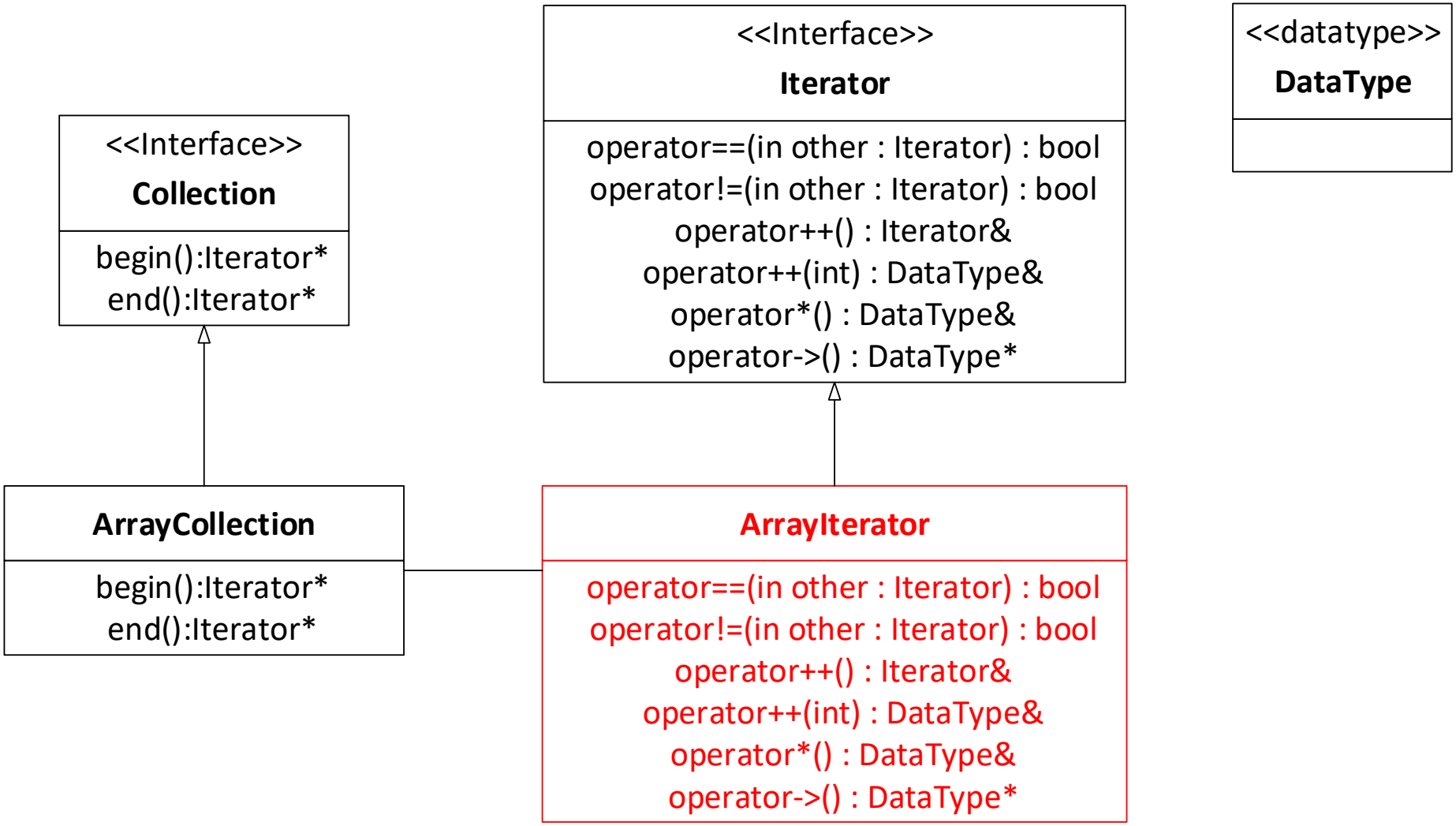
声明friend

默认大小

初始化一个名叫ArrayIterator的对象，0是其位置

_size是其位置

实现基于数组的Iterator



实现基于数组的Iterator

```
//继承自迭代器基类并配套ArrayCollection使用的迭代器
class ArrayIterator : public Iterator {
    float *_data;    //ArrayCollection的数据
    int _index;      //数据访问到的下标
public:
    ArrayIterator(float* data, int index) :
        _data(data), _index(index) { }
    ArrayIterator(const ArrayIterator& other) :
        _data(other._data), _index(other._index) { }
    ~ArrayIterator() { }
    Iterator& operator++();
    float& operator++(int);
    float& operator*();
    float* operator->();
    bool operator!=(const Iterator &other) const;
};
```

Iterator对Collection的数据访问

```
//迭代器各种内容的实现
Iterator& ArrayIterator::operator++() {
    _index++; return *this;
}
//因为是数组，所以直接将空间指针位置+1即可，可以思考下这里为什么返回float&，而不是Iterator
float& ArrayIterator::operator++(int) {
    _index++;
    return _data[_index - 1];
}
//对data的内存位置取值
float& ArrayIterator::operator*() {
    return *(_data + _index);
}
float* ArrayIterator::operator->() {
    return (_data + _index);
}
//判断是不是指向内存的同一位置
bool ArrayIterator::operator!=(const Iterator &other) const {
    return (_data != ((ArrayIterator*)(&other))->_data ||
            _index != ((ArrayIterator*)(&other))->_index);
}
```

返回的是++后的iterator

返回的是++前的iterator,但是不能临时构造一个iterator返回，因此这里返回的是float&,直接返回的是内容，但因为取了引用，对其修改也可以修改原数据（这只是一种实现方式，在STL中实现方式不同）

返回对应位置的内容

返回对应位置的指针

是同一个位置，当且仅当index和data（i.e.数组的头地址）一样

测试迭代器模式

为了实现继承多态，参数必须是引用or**指针**，这就是为啥这里是Iterator*

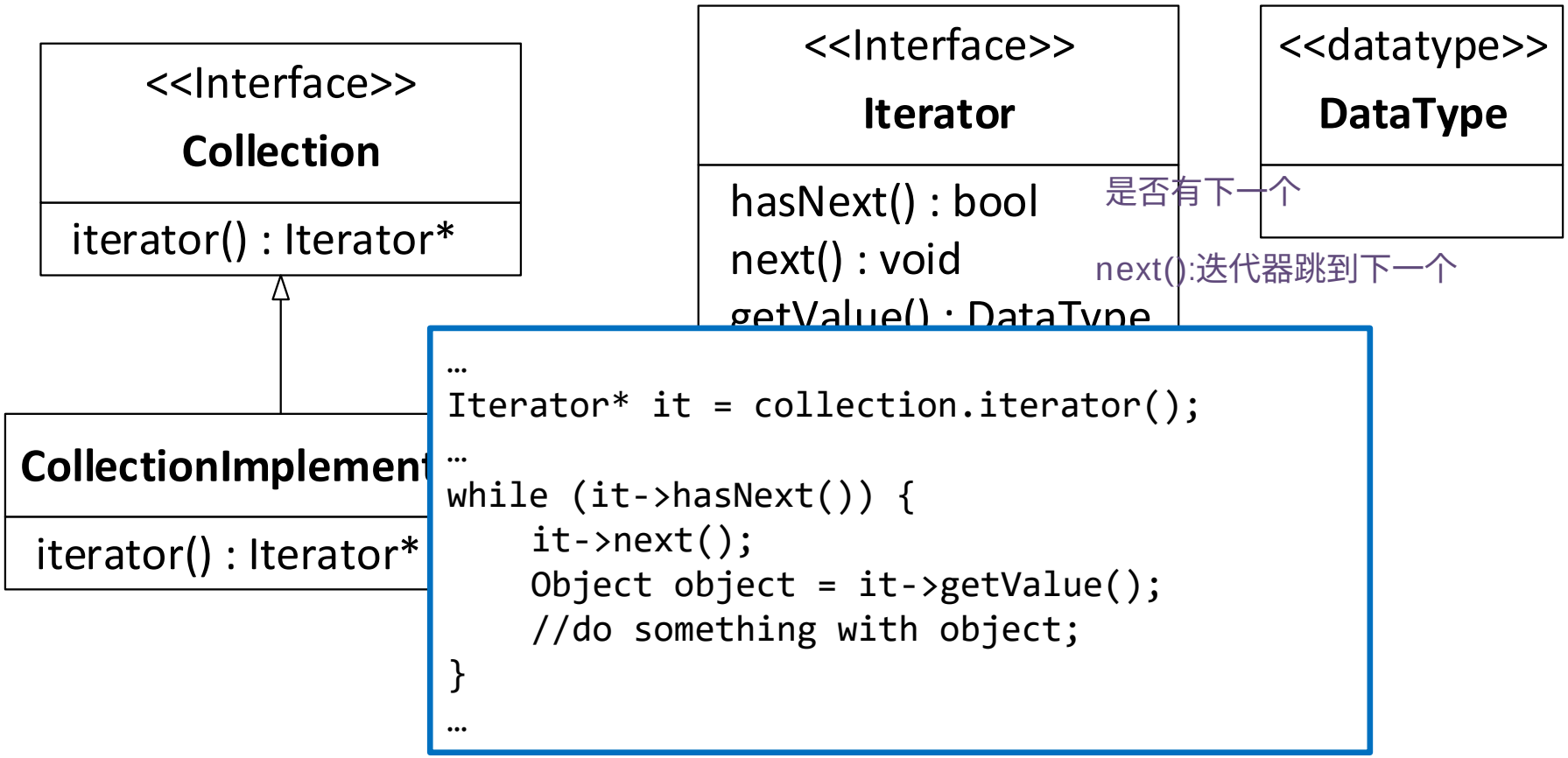
```
void analyze(Iterator* begin, Iterator* end) {
    int passed = 0, count = 0;
    for (Iterator* p = begin; *p != *end; (*p)++) {
        if (**p >= 60)
            passed ++;
        count ++;
    }
    cout << "passing rate =" << (float)passed / count << endl;
}


int main() {

    float scores[]={90, 20, 40, 40, 30, 60, 70, 30, 90, 100};
    Collection *collection = new ArrayCollection(10, scores);

    analyze(collection -> begin(), collection -> end());

    return 0;

}
```

collection就是把各种类型（list，vector,...)统一起来的一种类型

输出： passing rate = 0.5

另一种常见的迭代器模式

这种实现常用在while循环中



STL中的迭代器

■STL中的迭代器模式

```
template<class Iterator>
void analyze(Iterator begin, Iterator end) {
    int passed = 0, count = 0;
    for (Iterator p = begin; p != end; p++) {
        if (*p >= 60)
            passed ++;
        count ++;
    }
    cout << "passing rate =" << (float)passed / count << endl;
}

int main()
{
    std::vector<float> scores
            {90, 20, 40, 40, 30, 60, 70, 30, 90, 100};
    analyze(scores.begin(), scores.end());
    return 0;
}
```

STL已经帮你重载了++

STL中的迭代器

- STL中的迭代器模式
  - 之前介绍中的迭代器模式使用继承来实现 必须通过指针或引用实现
  - STL中迭代器模式使用模板来实现
  - 两种不同的多态实现方式

基于继承

```
class Collection {
public:
    virtual Iterator* begin() const = 0;
    virtual Iterator* end() const = 0;
};
```

返回迭代器基类指针

基于模板

```
template <class T>
class vector {
public:
    vector::iterator<T> begin() const;
    vector::iterator<T> end() const;
};
```

返回迭代器对象

STL中的迭代器

```
void analyze(Iterator* begin, Iterator* end) {
    int passed = 0, count = 0;
    for (Iterator* p = begin; *p != *end; (*p)++) {
        if (**p >= 60) passed ++;
        count ++;
    }
    cout << "passing rate =" << (float)passed / count << endl;
}
```

基于继承的迭代器模式

```
template<class Iterator>
void analyze(Iterator begin, Iterator end) {
    int passed = 0, count = 0;
    for (Iterator p = begin; p != end; p++) {
        if (*p >= 60) passed ++;
        count ++;
    }
    cout << "passing rate =" << (float)passed / count << endl;
}
```

基于模板的迭代器模式

STL中的迭代器

- 迭代器模式：模板 vs. 继承
  - 目标相同：将算法构建与底层数据结构解耦
  - 区别
    - 继承：
      - 算法中需要使用迭代器的基类指针
    - 模板：
      - 更加简洁，算法可以使用迭代器对象
      - 对每一种迭代器类型都会生成相应代码，使编译速度变慢、可执行文件变大

使用STL迭代器实现循环

- 使用迭代器进行循环：
  - for (auto i : container)

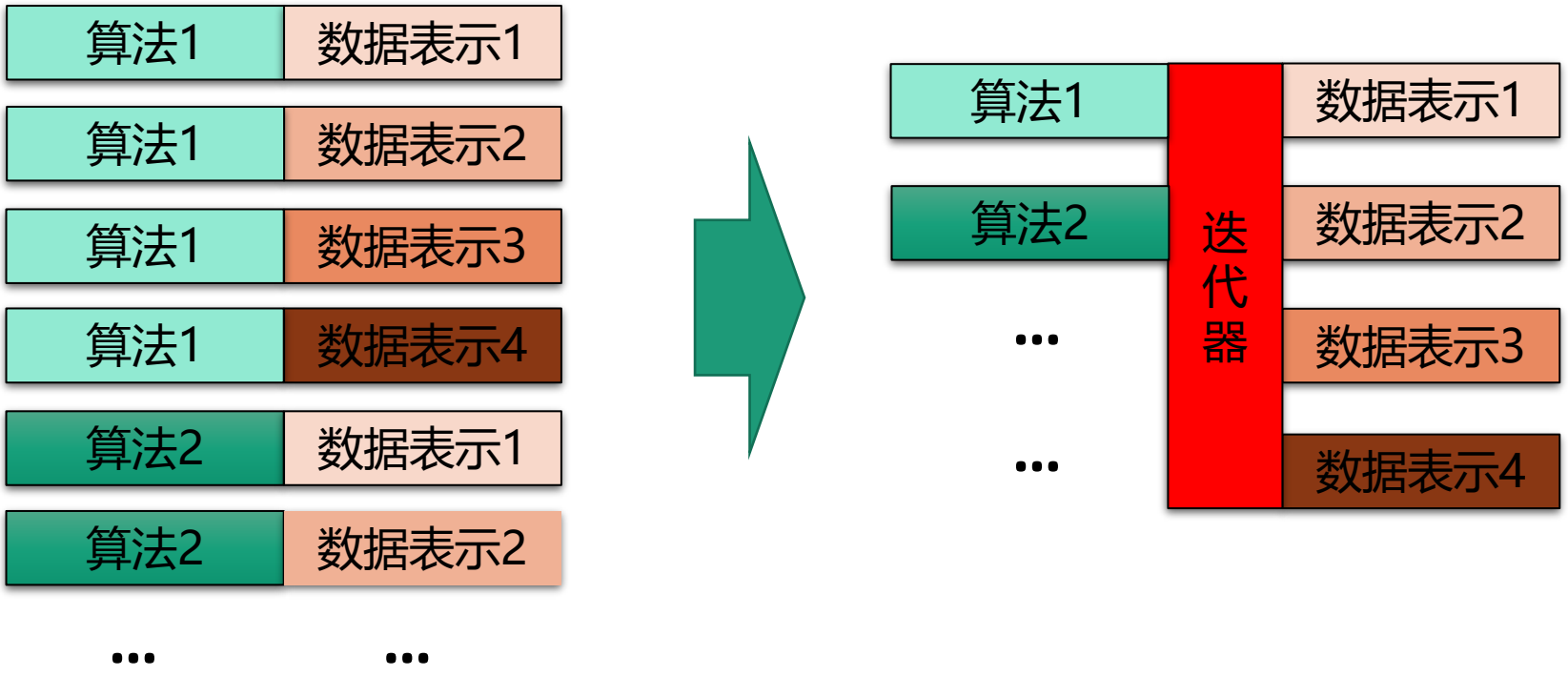
```
for (auto i : container)
{
    ...
}
```

```
for (auto i=container.begin();
    i != container.end(); ++i)
{
    ...
}
```

```
int main(){
    vector<int> container{1,2,3};
    for (auto a : container){
        cout << a << endl;
    }
    return 0;
}
```

总结

- 迭代器模式实现了算法和数据存储的隔离
- 规避了为每一个算法和数据存储的组合均进行代码实现的巨大工作量



STL

- C++的STL中提供了大量的数据容器
- 这些容器数据结构不同，数据访问操作类似，如遍历、最大值、最小值……
- STL繁多的数据结构均采用了类似的设计架构来抽象访问接口
- 推荐阅读STL的具体实现代码

本节课

- 行为型设计模式关心对象之间的行为功能抽象，核心在于抽象行为功能中不变的成分，具体实现行为功能中变的成分，保证以尽可能少的代码改动完成功能的增减
  - 模板方法归纳了一系列类的通用功能，在基类中将功能的接口固定，在子类中具体实现流程细节，使得新类的增加不对已有类产生影响
  - 策略模式抽象了功能的选择与组合，隔离不同的功能使得相互之间不受影响，可以灵活支持算法或策略的变动
  - 迭代器模式抽象了数据访问方法，可以访问对象的元素但却不暴露底层实现，隔离具体算法与数据结构

本节课

- 课后阅读
  - https://www.liaoxuefeng.com/wiki/1252599548343744/1281319453589538

结束