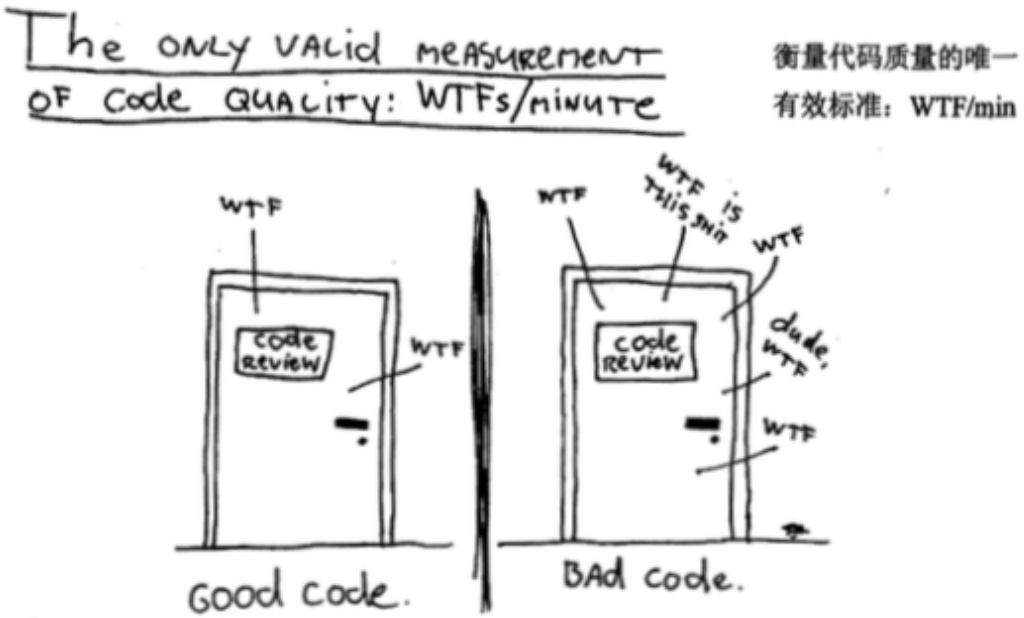
代码之道-整洁的代码

by David

谨记:代码是写给人看的


The ONLY VALid MEASUREMENT OF Code QUALITY: WTFs/minute
衡量代码质量的唯一有效标准：WTF/min
WTF
code review
WTF
Good code.
WTF
WTF is this shit
WTF
code review
dude, WTF
WTF
BAd code.

推荐阅读

- 《代码整洁之道》
- 《编写可读代码的艺术》
- 《重构:改善既有代码的设计》

0 代码就像我们的孩子

0.1 代码永存、混乱当道

- 混乱的制造者
  - 规范的缺失
  - 程序员作茧自缚
- 混乱的代价
  - 新人如坠云雾
  - 添加新特性,需要披荆斩棘
  - 修改BUG,如同大海捞针

0.2 我们的态度

- 保持代码的整洁
- Make It Better and Better ...
- 改变"只要代码能跑就不要去动它的"想法

0.3 Clean Code

KISS:风格统一、代码逻辑简洁明了,直截了当!

让重复代码见鬼去吧!

1. 取个好名字

1.1 原则:名副其实

- 选名字是件严肃的事情,选个好名字很重要
- 如果名字需要注释来补充,那就不是个好名字

```
1  int t = currentTime.elapse(e); // 消逝的时间,以毫秒计
2  ...
3  if (t > timeout_value)
4  {
5     Zebra::logger->debug("---一次循环用时 %u 毫秒-----", t);
6  }
```

```
1  int elapsed_ms = currentTime.elapse(e);
2  ...
3  if (elapsed_ms > timeout_value)
4  {
5     Zebra::logger->debug("-----一次循环用时 %u 毫秒---", elapsed_ms);
6  }
```

1.2 原则:避免误导

- 必须避免留下掩藏代码本意的错误线索
- 避免使用与本意相悖的词
- 提防使用不同之处较小的名称
- 拼写前后不一致就是误导

```
1  std::vector<int> account_list; // _list就是一个误导，accounts会更好
2
3  bool sendToZoneServer(); // 和下面的函数差别很小
4  bool sendToZoneServers(); // sendToAllZoneServers会好点
```

1.3 原则:做有意义的区分

- 代码是写给人看的,仅仅是满足编译器的要求,就会引起混乱
- 以数字系列命名(a1,a2,...),纯属误导
- 无意义的废话: a, an, the, Info, Data

```
1  void copy(char a1[], char a2[]) {
2    for (size_t i = 0; a1[i] != ''; i++)
3      a2[i] = a1[i];
4  }
```

```
1  void copy(char source[], char dest[]) {
2    for (size_t i = 0; source[i] != ''; i++)
3      dest[i] = source[i];
4  }
```

1.4 原则:使用可读的名字

- 避免过度使用缩写
- 可读的名字交流方便
- BAD: class XLQY, class FCNV, class LTQManager

1.5 原则:使用可搜索的名字

- 避免使用Magic Number
- 避免使用单字母,或出现频率极高的短字母组合(注意度的把握)

```
1  if (obj->base->id == 4661) // 4661是啥玩意？
2  {
3     usetype = Cmd::XXXXXXX;
4  }
5
6  int e; // 怎么查找？
7  XXXX:iterator it; // 变量作用的范围比较大的时候,也不见得是个好名字
```

```
1
2  #define OJBECT_FEEDBACK_CARD 4661
3
4  if (OJBECT_FEEDBACK_CARD == obj->base->id)
5  {
6     usetype = Cmd::XXXXXXX;
7  }
```

1.6 原则:避免使用编码

- 匈牙利标记法
  - Windows API时代留下的玩意
  - wdXX, dwXXX, strXXX
  - 变量类型变换导致名不副实
- 成员前缀
  - m_name, m_xxx
  - 基本上都无视,为何要多次一举
- 接口和实现
  - IXXX, I-接口修饰前缀
  - CXXX, C-类修饰前缀
  - 这些修饰多数时候都是废话

1.7 原则:名字尽量来自解决方案领域或问题领域

- 使用解决方案领域名称
  - 都出自CS,术语、算法名、模式名、数学术语尽管用。
  - 如AccountVisitor,Visitor模式实现的Account类
- 使用问题领域的名称
  - 我们代码里面多数都是这些名称,不明白找策划问问

1.8 原则:适当使用有意义的语境

- 良好命名的类、函数、名称空间来放置名称,给读者提供语境
- 只有两三个变量,给名称前加前缀
- 事不过三,变量超过三个考虑封装成概念,添加struct或class

```
1  // 看着整齐?使用方便?
2  DWORD love_ensure_type_;  //当前的爱情保险类型
3  DWORD love_ensure_ret_; //购买爱情保险回应标示
4  DWORD love_ensure_total_; //现在已经盖章数目
5  DWORD love_ensure_..._;  //...
6  DWORD love_ensure_..._;  //...
```

我们的原则:

- 名副其实
- 避免误导
- 做有意义的区分
- 使用可读的名字
- 使用可搜索的名字
- 避免使用编码
- 名字尽量来自问题/解决方案领域
- 适当使用有意义的语境

我们的规范:

- 文件名
  - 首字母大写,多个词组合起来
  - 如: SceneUser.h Sept.h
- 类名/名称空间名
  - 首字母大写，多个词组合起来
  - 使用名词或名词词组
  - 避免使用C前缀，如:CSept
  - 如: SceneUser SeptWar

我们的规范:

- 函数名
  - 首字母小写
  - 使用动词或动词词组
  - 避免使用孤立的全局函数,可以封装在类或名称空间里面
  - get, set, is前缀的使用
  - 如: fuckYou(), levelup()
- 变量名
  - 全部字母小写,多个词以下划线分隔
  - 私有成员变量加后缀"_",公有变量不用
  - 避免使用孤立的全局变量,可以封装在类或名称空间里面
  - 如: quest_id, questid_

我们应该这样做:

- 写下任何一行代码的时候，心里都要想着自己的代码是给别人看的
  - 为函数、变量、类取个好名字，遵循规范和原则
  - 见到不符合规范和原则的名字，毫不留情的干掉它

2. 函数:过程的抽象

2.1 原则:取个描述性的名字

- 取个一眼就看出函数本意的名字很重要
- 长而具有描述性的名称，要比短而让人费解的好
- 使用动词或动词+名词短语

2.2 原则:保持参数列表的简洁

- 无参数最好，其次一元，再次二元，三元尽量避免
- 尽量避免标识参数
- 使用参数对象
- 参数列表
- 避免输出和输入混用，无法避免则输出在左，输入在右

```
1  bool isBossNpc();
2  void summonNpc(int id);
3  void summonNpc(int id, int type);
4  void summonNpc(int id, int state, int type); // 还能记得参数顺序吗？
5
6  void showCurrentEffect(int state, bool show); // Bad!!!
7  void showCurrentEffect(int state); // Good!!
8  void hideCurrentEffect(int state); // 新加个函数也没多难吧？
9
10 bool needWeapon(DWORD skillid, BYTE& failtype); // Bad!!!
```

2.3 原则:保持函数短小

- 现状：有些函数我们得按米来度量
- 第一规则：要短小
- 第二规则：还要更短小
- 一屏之地，一览无余

2.4 原则:只做一件事

- 函数应该只做一件事，做好这件事
- 只做这一件事

2.5 原则:每个函数位于同一抽象层级

- 要确保函数只做一件事，函数中的语句都要在同一个抽象层级上
- 自顶下下读代码

2.6 原则:无副作用

- 谎言，往往名不副实

2.7 原则:操作和检查要分离

- 要么是做点什么，要么回答点什么，但二者不可兼得
- 混合使用—副作用的肇事者

2.8 原则:使用异常来代替返回错误码

- 操作函数返回错误码轻微违法了操作与检查的隔离原则
- 用异常在某些情况下会更好点
- 抽离try-cacth
- 错误处理也是一件事情，也应该封装为函数

```
1  bool RedisClient::connect(const std::string& host, uint16_t port)
2  {
3          this->host = host;
4          this->port = port;
5          this->close();
6
7          try
8          {
9                  redis_cli = new redis::client(host, port);
10                  return true;
11          }
12          catch (redis::redis_error& e)
13          {
14                  redis_cli = NULL;
15                  std::cerr << "error:" << e.what() << std::endl;
16                  return false;
17          }
18
19          return false;
20  }
```

2.9 原则:减少重复代码

重复是一些邪恶的根源！！！

2.10 原则:避免丑陋不堪的switch-case

- 天生要做N件事情的货色
- 多次出现就要考虑用多态进行重构

```
1  bool saveBinary(type, data) {
2     switch (type) {
3       case TYPE_OBJECT:
4             ....
5           break;
6       case TYPE_SKILL:
7             ...
8           break;
9       ....
10    }
11 }
12 bool needSaveBinary(type) {
13    switch (type) {
14      case TYPE_OBJECT:
15          return true;
16      case TYPE_SKILL:
17            ...
18          break;
19      ....
20    }
21 }
```

```
1
2  class BinaryMember
3  {
4    BinaryMember* createByType(type){
5     switch (type) {
6       case TYPE_OBJECT:
7            return new ObjectBinaryMember;
8       case TYPE_SKILL:
9            return new SkillBinaryMember;
10       ....
11    }
12
13    virtual bool save(data);
14    virtual bool needSave(data);
15 };
16
17 class ObjectBinaryMember : public BinaryMember
18 {
19    bool save(data){
20        ....
21    }
22    bool needSave(data){
23        ....
24    }
25 };
```

我们的原则:

- 使用描述性的名字
- 保持参数列表的整洁和清晰
- 短小
- 只做一件事，并把它做好
- 每个函数位于同一抽象层
- 无副作用
- 操作和检查要分离
- 使用异常来代替返回错误码
- 避免丑陋不堪的switch-case
- 避免重复

我们的规范:

- 命名：动词/动词词组，首字母小写
- 参数：无特殊原因3个以内,输出在左，输入在右
- 函数体：一屏之地，一览无余

我们应该这样做:

- 添加新函数
  - 刚下手时违反规范和原则没关系
  - 开发过程中逐步打磨
  - 保证提交后的代码是整洁的即可
- 重构已有函数，见到一个消灭一个
  - 冗长而复杂
  - 有太多缩进和嵌套循环
  - 参数列表过长
  - 名字随意取
  - 重复了三次以上

3.数据的抽象:结构体VS.对象

3.1 数据抽象

- 结构体(struct实例):
  - 暴露数据实现
- 对象(class实例):
  - 隐藏数据实现
  - 暴露行为接口

```
1  struct Point {
2      int x;
3      int y;
4  };
```

```
1
2  class Point {
3
4  public:
5       int getX();
6       int getY();
7       void setByCartesion(int x, int y);
8
9       int getR();
10       int getTheta();
11       void setByPolor(int r, int theta);
12  // God knows the following.. who cares?
13  private:
14       int x_;
15       int y_;
16  };
17
```

3.2 结构体 VS. 对象

- 反对称性
  - 结构体:容易添加新函数,缺难于修改数据
  - 对象:容易添加数据,缺难于添加新函数
- 思考:
  - Shape增加一个计算周长的函数?
  - Shape增加一种新类型?

```
1  struct Shape
2  {
3     int type;
4     union {
5        Point  rect;
6        double radius;
7     };
8  };
9
10  double caclArea(Shape* shape)
11  {
12    switch (shape->type)
13    {
14        case RECT:
15          return shape->rect.width*shape->rect.height;
16        case CIRCLE:
17          return shape->radius*shape->radius*3.14;
18    }
19  }
20
```

```
1
2  class Shape {
3  public:
4     virtual double caclArea();
5  };
6
7  class Squre : public Shape {
8  public:
9     double caclArea(){ return width_ * height_;}
10  private:
11     double width_;
12     double height_:
13  };
14
15  class Circle : public Shape {
16  public:
17     double caclArea(){ return 3.14*radius_* radius_;}
18  private:
19     double radius_;
20  };
21
```

3.3 原则:结构体和对象的选择

- 结构体
  - 结构相对固定
  - 行为变化较大
  - 如:Point, Time, ...
- 对象
  - 行为相对固定
  - 经常会添加新类型
- 避免混合使用

3.4 原则:不要和陌生人说话

- 德墨忒尔律:模块不应了解它所操作的内部情形

- 类C的方法f只应该调用以下对象的方法:
  - C
  - 由f创建的对象
  - 作为参数传递给f的对象
  - 由C的实体变量持有的对象

- 目的
  - 信息隐藏
  - 避免修改时引起多处修改

3.5 原则:数据传输对象/POD

- POD:Plain Of Data
- 用于消息传递
- 用于存档

我们的原则:

- 结构体和对象的选择遵循数据和行为的变化
- 不要和陌生人说话
- POD的使用

4. 类

4.1 原则：统一的布局

布局原则：

1.static和typedef放在最前面

2.构造和析构

3.使用频率较高的通用函数

4.根据功能分段

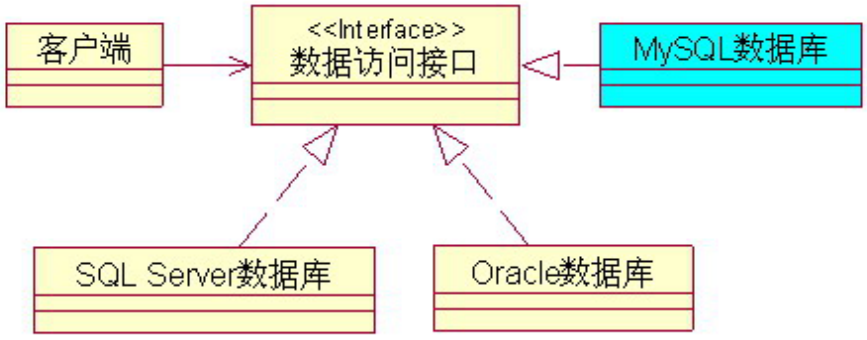
```
1  class Sept
2  {
3  public:
4      typedef std::map<int,int> ObjectMap;
5      const float PI = 3.14;
6
7      static int getCounter();
8      static int counter;
9
10      Sept();
11      ~Sept();
12
13      // 使用频率较高的通用函数
14      int getSeptMaster();
15      void setSeptMoney(int money);
16
17      // XXX功能
18      void doXXCmd();
19      void giveXXAward();
20
21      // YYY功能
22      void doYYCmd();
23      void giveYYAward();
24
25  protected:
26      ...
27  private:
28      int privateFunction();
29      int private2Function();
30
31      int member_;
32  };
```

4.2 原则：保持短小

- SRP：单一职责
  - 系统应该由许多短小的类组成
  - 每个小类封装一个职责
  - 只有一条修改的理由
- 高内聚
  - 类应该只有数量实体变量
  - 每个方法都应该操作一个或多个这种变量

4.3 原则：为修改做准备（面向对象五大原则）

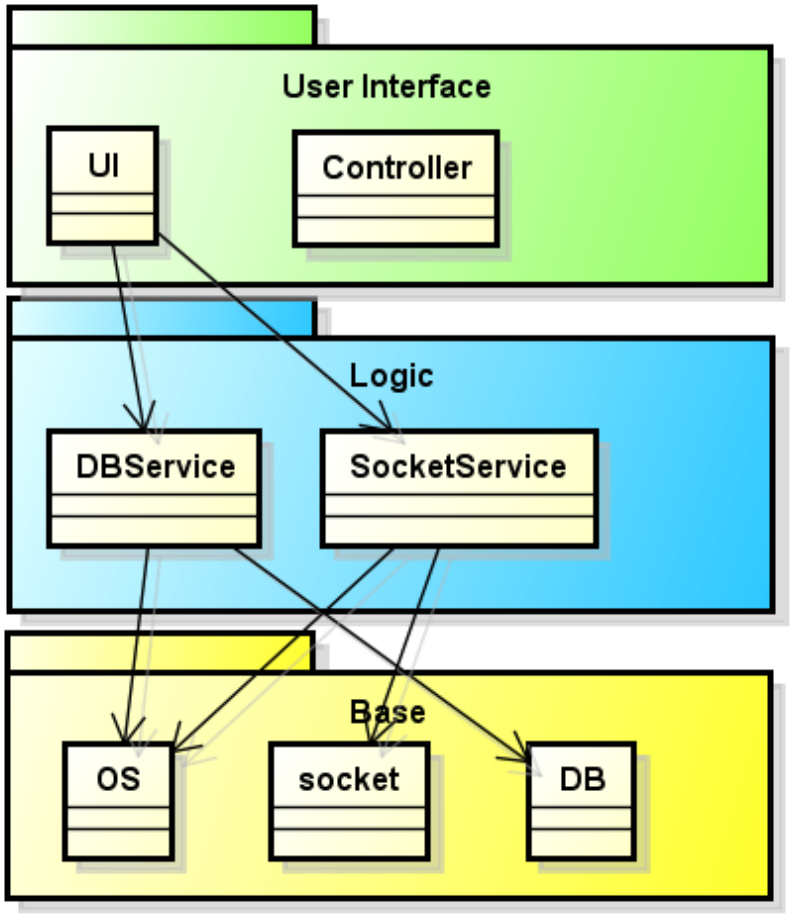
- 1. SRP：单一职责原则
  - Single Responsibility Principle
  - 一个类应该仅有一个引起它变化的原因
- 2. OCP：开放封闭原则
  - Open Close Principle
  - 为扩展开放、为修改关闭

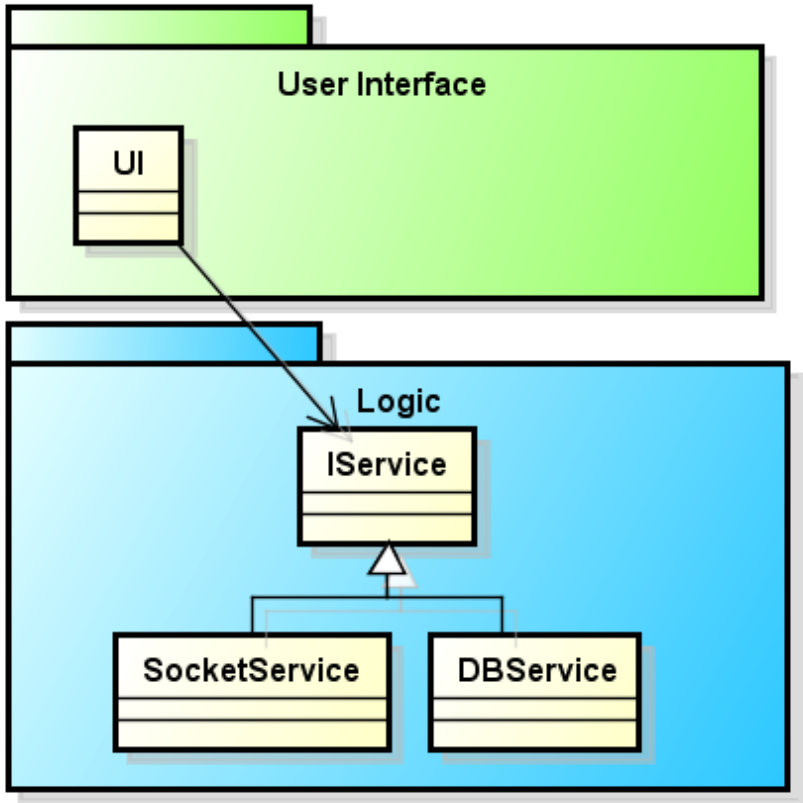


OCP图

4.3 原则:为修改做准备(面向对象五大原则)

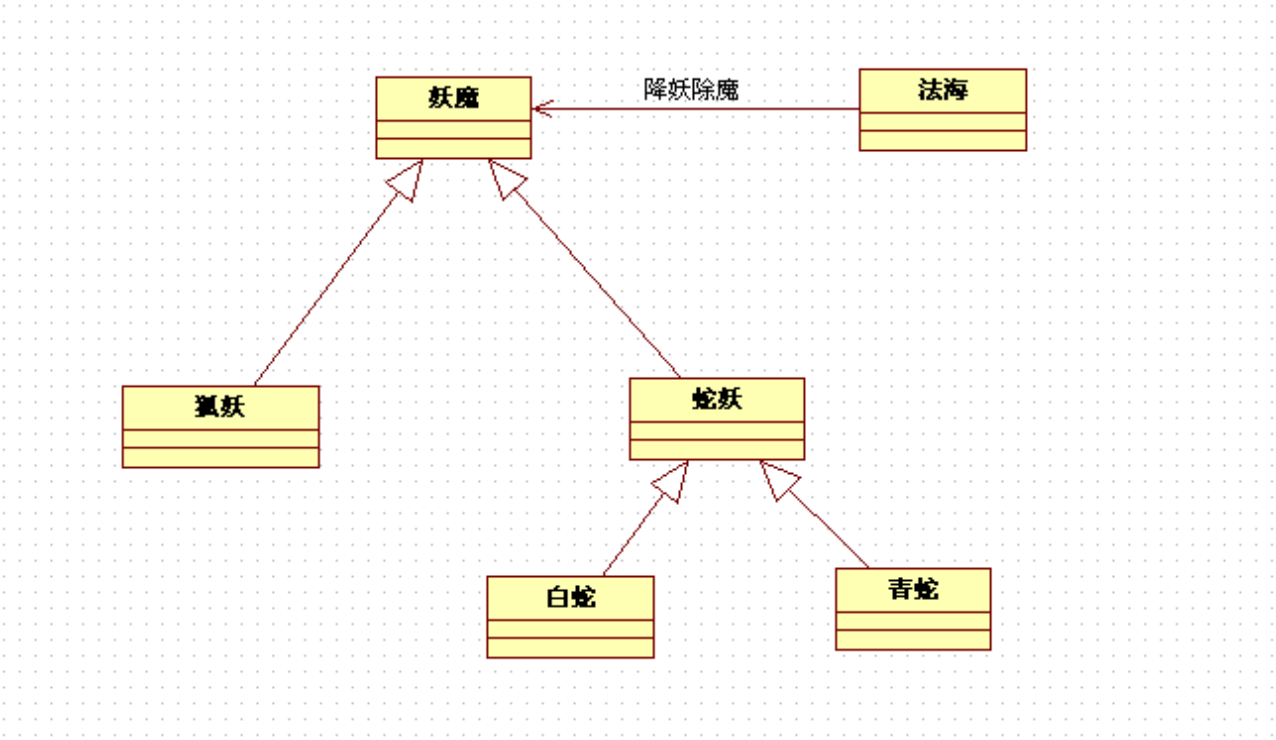
- 3. DIP:依赖倒置原则
  - Dependency Inverse Principle
  - 逆转高层依赖底层





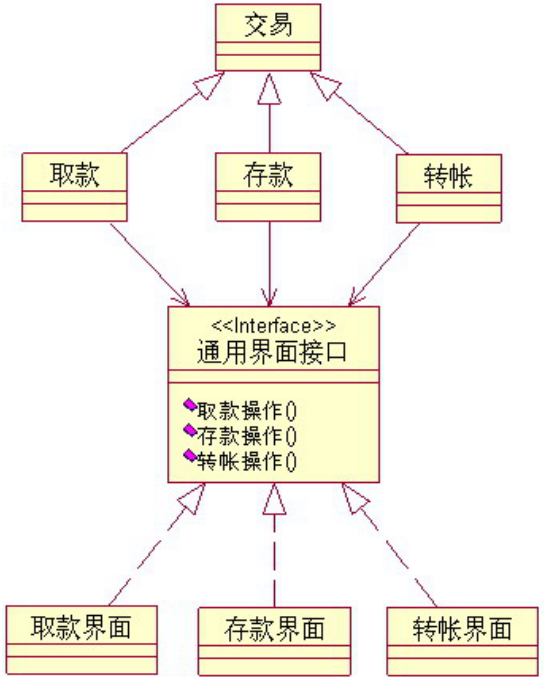
4.3 原则:为修改做准备(面向对象五大原则)

- 4. LSP:Liskov替换原则
  - Liskov Substitution Principle
  - 子类可以替换父类并且出现在父类能够出现的地方
  - 面向接口编程

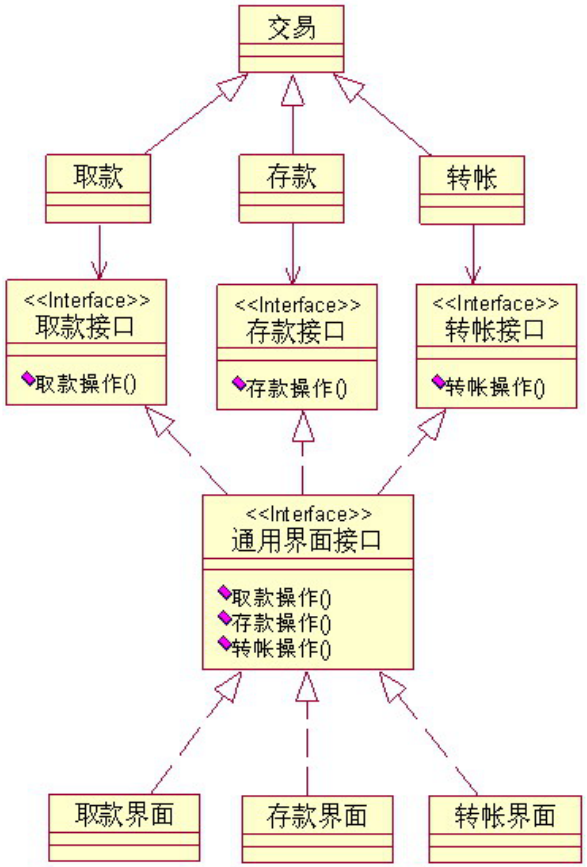


4.3 原则:为修改做准备(面向对象五大原则)

- 5. ISP:接口隔离原则
  - Interface Isolation Principle
  - 使用多个专门的接口比使用单个接口要好的多



ISP图1

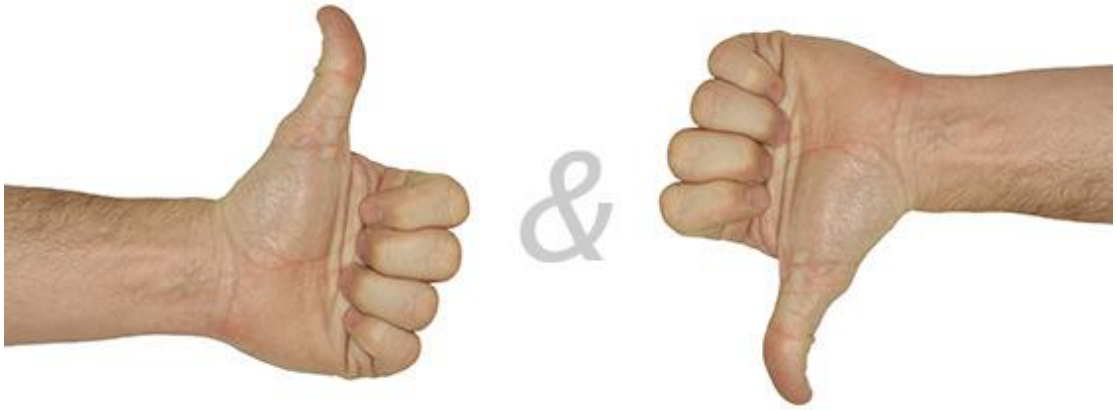


ISP图2

我们的原则：

- 统一布局
- 保持短小
- 面向对象五大原则：
  - SRP
  - OCP
  - DIP
  - LSP
  - ISP

5. 注释



5.1 原则:别给糟糕的代码加注释，重新写吧！

- 注释的“好”与“恶”
  - 什么也比不上放置良好的注释有用
  - 什么也不会比乱七八糟的注释更有本事搞乱一个模块
  - 什么也不会比陈旧、提供错误信息的注释更有破坏性
- 通常注释意味着什么
  - 失败的表达
  - 没有几个人修改代码的时候会去及时完善注释
  - 不能美化糟糕的代码，只会让人更恶心

5.2 原则:保留好的注释

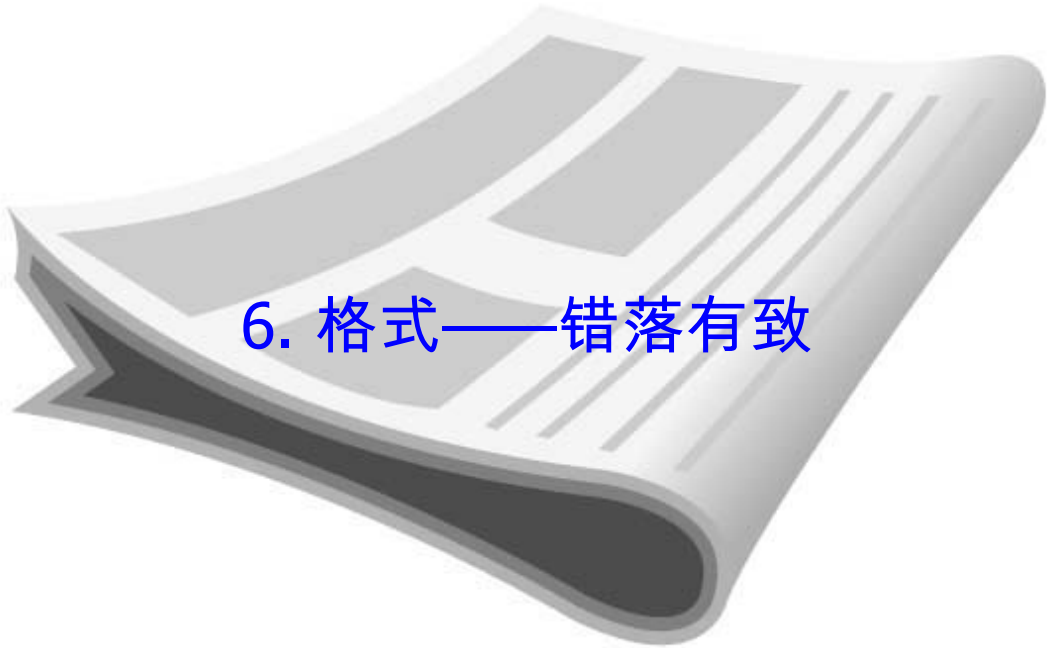
- 法律信息（CopyRight...）
- 对意图的解释（特别是一些复杂的逻辑）
- 警告（一些影响运行或者其它类似的信息）
- TODO注释（临时性的注释，将来要整理的代码）

5.3 原则:拒绝坏的注释

- 喃喃自语，废话连篇（不要把看代码的人当成超级小白）
- 误导性的注释（不要也罢，但不能出来害人啊！）
- 形式化的注释（完全就是浪费时间和生命）
- 括号后面的注释（意味你的函数太庞大了)
- 归属或签名（SVN知道你是谁，不要留太多名）
- 注释掉的代码（或许永远都不会用到，交给SVN吧）

我们的原则:

- 别给糟糕的代码加注释，重新写吧！
- 保留好的注释
- 拒绝坏的注释
- PS.每当写下一行注释的时候，都要思考下：
  - 是不是名字去的不好？
  - 是不是函数太过庞大了？算法太复杂了？
  - 是不是放错位置的了？
  - ...



6. 格式——错落有致

6.1 原则:像写文章一样写代码

- 人如其文，代码同此理
- 整洁一致的代码赏心悦目，宾至如归
- 格式不统一的代码，总是无法让人淡定
- 代码之美：能工作vs.可读性

6.2 垂直格式

- 垂直段落
  - 靠近：语义相近的靠近
  - 间隔：不相近的以空行隔开
- 垂直距离
  - 变量声明应该靠近使用的位置
  - 相关函数放在一起
  - 概念相关的语句放在一起
- 垂直顺序
  - 自顶而下
  - 遵循函数调用依赖的顺序：被调用的放在下面

6.3 横向格式

- 缩进：有效区分代码块和作用域
- 空格：挤在一起看着别扭
- 换行：一行尽量不要太长
- 垂直对齐：无所谓

我们的原则:

像写文章一样来些代码，讲究段落层次，错落有致！

Z.最后

问题：

- 你心目中，整洁代码是什么样的？

下一步:

- Shut the Fuck Up! Just Do It !!!
  - 规范
  - 自觉
  - 监督

That's All, Thanks !