# 一.意图导向编程

## 1.1 基本思想

“意图导向编程”,Programming by Intention,也称目的导向编程/自顶向下编程.

其基本思想是：每一个问题都可以分解成一系列的功能性步骤，在写代码的过程中，会按照顺序有意识的去完成每一个步骤；而意图导向编程则是先假设每一个步骤都有一个理想的方法来完成，而不关注每个步骤的具体实现，在这种情况下，需要关心的是每个方法的输入参数，返回值以及什么样的名字最符合它的含义。

例如，创建一个服务，它接受一个业务交易，然后提交，具体的需求如下：

* 交易信息开始于一串标准的ASCII字符串。
* 这个信息字符串必须转换成一个字符串的数组，数组存放的值是此次交易用到的领域语言(domain language)中所包含的词汇元素(token)。
* 每一个词汇元素必须标准化(第一个字母大写，其余字母小写，空格和非字符数字的符号都要删掉)。
* 超过150个词汇元素的交易，应该采用不同于小型交易的方式(不同的算法)来提交，以提高效率。
* 如果提交成功，API方法返回true，否则返回false。

基于“意图导向编程”的思想，我们假设有一个类，类中有一个API实现上面的服务：

public class Transaction{  
 public Boolean commit(String command){  
 Boolean result = true;  
 String[] tokens = tokenize(command);  
 normalizeTokens(tokens);  
 if(isALargeTransaction(tokens)){  
 result = processLargeTransaction(tokens);  
 }else{  
 result = processSmallTransaction(tokens);  
 }  
  
 return result;  
 }  
}

commit()方法是程序API，用于提供服务，而其他方法(tokenize()、normalizeTokens()、isALargeTransaction()、processLargeTransaction()、processSmallTransaction())都不属于这个对象API，仅仅是实现过程中的功能性步骤，称之为“辅助方法”(helper methods),暂时可以将他们视为私有方法。

通过这样的编码方式，可以将精力集中在如何分解最终目标，以及那些全局性的问题上。

并且这种实现方式，与直接把所有代码写到一个很长的方法里相比并没有增加工作量，不同点在于思考的方式和编码的顺序(先理清整体流程，再深入细节)。

## 1.2 优点

如果遵循意图导向编程，那么代码将会:

* 更加内聚(职责单一)。
* 更加可读和清晰
* 更易于调试
* 更易于重构和优化
* 更易于单元测试
* 更易于维护
* 创建的方法更容易从一个类移动到另一个类
* 更容易应用设计模式

### 1.2.1 方法的内聚性

代码的质量标准之一就是内聚性。以类为例，每个类都应该根据职责来定义，并且应该只有一个职责。类内部包括方法、状态以及与其他对象之间的关系，如果各个方面都紧密相关，并且围绕着这个类的唯一职责，则说这个类的内聚性很强。

如果一个方法只实现整体职责中一个单独的功能点，则说这个方法的内聚性很强。

人的思维方式是单线程的，当进行“多任务”的时候，实际上是在多个任务之间快速切换而已，人们仍旧习惯于一次只思考一件事情。意图导向编程正是利用这一事实，用思维链条单一性的特定去创建同样具备单一性的内聚方法。

### 1.2.2 可读性和表达性

通过阅读最初的实例代码：

public class Transaction{  
 public Boolean commit(String command){  
 Boolean result = true;  
 String[] tokens = tokenize(command);  
 normalizeTokens(tokens);  
 if(isALargeTransaction(tokens)){  
 result = processLargeTransaction(tokens);  
 }else{  
 result = processSmallTransaction(tokens);  
 }  
  
 return result;  
 }  
}

可以发现该服务的实现流程是：获取到一个指令，然后对指令进行分词，再把分词后得到的指令标准化，判断需要进行交易处理的类型，根据判断结果来决定进行大型事务还是小型事务的处理，最后返回结果。

因为上面的代码只涉及到“做什么”，而不是具体的“如何做”，这种情况下，不需要注释也能读懂代码的基本逻辑，这得益于规范的命名和步骤的清晰界定。

考虑下面的实现方式:

public class Transaction{  
 public Boolean commit(String command){  
 Boolean result = true;  
  
 // tokenize the string  
 some code here  
 some more code here  
 even some more code here that sets tokens  
  
 // normalize the tokens  
 some code here that normalize tokens  
 some more code here that normalize tokens  
 even some more code here that normalize tokens  
  
 // see if you have a large transaction  
 code that determines if you have a large transaction  
 set lt = true if you do  
 if(lt){  
 // process large transaction  
 some code here to process large transaction  
 some more code here to process large transaction  
 }else{  
 // process small transaction  
 some code here to process small transaction  
 some more code here to process small transaction  
 }  
  
 return result;  
 }  
}

上面的实现方式是将所有逻辑写在一个大方法中，并且加了详尽的注释，但与意图导向编程的实现方式相比，注释显得很没有必要，并且代码太多，给人的心理无形中造成一种压力。

### 1.2.3 调试

在程序的代码错误修复过程中，寻找错误所在才是最花时间的。在遵循意图导向编程时，由于一个方法只做一件事，这个时候，如果出现错误，则可试试下面的办法：

* 通读一遍整个方法，看看所有事情是怎么运作的
* 对无法正常工作的部分，检查辅助代码的细节有什么问题

相比于费力的查阅一大段复杂的代码，这种调试方法发现代码错误的速度要快很多。

### 1.2.4 重构和增强

重构系统：保持系统行为不变的情况下，更改它的结构。

增强系统：增加或修改系统的行为以符合新的需求。

重构通常认为是“清理”系统中写得糟糕的代码，重构的一个基本实现方式是：把一部分代码从一个巨大的方法中抽取出来，放到一个属于它自己的新方法中，而在原来代码中的那个位置直接调用这个新方法。

由于原来方法的一部分临时变量也需要迁移到新方法中，所以需要多个步骤才能完成一个函数的提炼。

如果采用意图导向编程，一开始就是辅助方法了，只需要把公用的辅助方法迁移到其他类即可。这样的重构是很快的(复制-粘贴)。

当系统实现后，有新需求加进来了，如：与第三方程序交互时，由于第三方程序的原因，不再支持某些旧版词汇，这个时候需要更新一个词汇元素，如：

public class Transaction{  
 public Boolean commit(String command){  
 Boolean result = true;  
 String[] tokens = tokenize(command);  
 normalizeTokens(tokens);

updateTokens(tokens);  
 if(isALargeTransaction(tokens)){  
 result = processLargeTransaction(tokens);  
 }else{  
 result = processSmallTransaction(tokens);  
 }  
  
 return result;  
 }  
}

有新需求加进来的时候，只需要在API方法的实现流程中增加updateTokens()方法，其他都不需要修改到，把影响降到了最低。

如果修改了标准化的算法，则更改normalizeTokens()方法即可，其他都无需改动。

在修改的过程中，代码能很快定位。

### 1.2.5 单元测试

设计的基本建议：使用服务的客户端，在设计时应该遵照的是服务的接口定义，而不是服务的具体实现。

在上面的实现中，辅助方法被定义成了私有方法，是为了不想与外部对象发生关联，但这种情况下就不利于方法的单元测试。

我们只能对commit()方法进行单元测试，即测试服务的整体行为。此时测试情况比较复杂，会有很多种因素导致测试失败。

可以有如下解决办法:

* 如果辅助方法只是实现单个服务的一部分，则没必要单独测试辅助方法，测试这个服务流程即可。
* 如果某些辅助方法是能被其他服务使用到的，则需要将该辅助方法单独到其他的类中，并且定义成公有的方法，则对原来辅助方法的调用就变成了对新类方法的调用，并且新类的公有方法是能进行单元测试的。

### 1.2.6 可迁移的方法

为了提高类的内聚性，需要把这个类不应该有的方法迁移到其他类中，这样可以让这个类所关注的东西减少。

意图导向编程创建的方法只完成一个功能，这样避免了迁移方法是经常遇到的问题：包含不能迁走的部分。

当一个方法只做一件事时，要么全部迁移，要么不迁移。

方法迁移难，还可能由于它直接关联到了类中的状态变量，在使用意图导向编程时，习惯于将参数传递到辅助方法，然后获取一个返回结果，而不是让方法直接使用对象的状态。

### 1.2.7 易于修改和扩展

从之前的重构和增强可看成，当增加需求时，只需要在流程中增加对应的辅助方法；当需要修改需求时，只需要修改对应的辅助方法。这种修改和扩展容易定位并且不影响其他代码。

### 1.2.8 在代码中发现模式

上面的例子中，如果有两个不同的交易类型，流程步骤一样(分词、标准化、更新、处理)，但每一步的实现方式不一样。使用意图导向编程时，处理每个辅助方法具体实现不一样，commit()方法是一样的，这个时候，可以很容易的应用模板方法模式。