重构草案的具体格式：后续在对每一种重构方式都进行说明的时候均采用此格式。

|  |  |
| --- | --- |
| 名称 | 建造一个重构词汇表 |
| 概要 | 简单介绍一下重构手法的适用场景以及它所做的事情 |
| 动机 | 为什么需要这个重构以及什么情况下不能使用这个重构 |
| 做法 | 简明扼要的一步一步介绍此重构的运作方式 |
| 范例 | 以一个十分简单的例子说明重构的运作方式 |

1. Compose Method

|  |  |
| --- | --- |
| 名称 | Extract Method |
| 概要 | 将这一段代码放进一个独立函数，并让函数名称解释函数的用途 |
| 动机 | 一个过长的函数或者需要一段注释才能让人理解用途的代码，就会将这段长长的代码放进一个独立函数中。 |
| 做法 | 创造一个新函数，根据这个函数的意图对其进行命名（以它“做什么”来命名，而不是以它“怎么做”命名。 |
| 范例  （重构前） | void printOwing(double amount) {  printBanner();  System.out.println(“name” + name);  System.out.println(“amount” + amount);  } |
| 范例  （重构后） | void printOwing(double amount) {  printBanner();  printDetail(amount);  }  void printDetail(double amount) {  System.out.println(“name” + name);  System.out.println(“amount” + amount);  } |

|  |  |
| --- | --- |
| 名称 | Inline Method |
| 概要 | 一个函数的本体与名称同样清楚易懂，在函数调用点插入函数本身，然后移除该函数。 |
| 动机 | 在遇到函数内部代码与函数名称同样清晰易读的情况下，你应该直接使用函数本体，而不是使用一个短小的函数去封装。 |
| 做法 | 1）检查函数，确定其不具备多态性；  2）找出该函数的所有被调用点；  3）将这个函数的所有调用点更换成函数本体；  4）编译以及测试； |
| 范例  （重构前） | int getRatings() {  return (moreThanFiveLateDeliveries()) ? 2 : 1;  }  boolean moreThanFiveLateDeliveries() {  return numberOfLateDeliveries > 5;  } |
| 范例  （重构后） | int getRatings() {  return (numberOfLateDeliveries > 5) ? 2 : 1;  }  ~~boolean moreThanFiveLateDeliveries() {~~  ~~return numberOfLateDeliveries > 5;~~  ~~}~~ |

|  |  |
| --- | --- |
| 名称 | Inline Temp |
| 概要 | 将所有对该变量的引用动作，替换成对它赋值的那个表达式自身 |
| 动机 | 当你发现某个临时变量被赋予某个函数调用的返回值，如果它的存在会妨碍其他重构手法时，就应当将其进行重构。 |
| 做法 | 1）检查给临时变量赋值的语句，确保等号右边的表达式没有副作用；  2）如果该临时变量未被声明为final，那就将其声明为final，然后编译；  3）找到该临时变量的所有引用点，将其替换成“为临时变量”赋值的表达式；  4）修改完后编译，测试，并删除该临时变量的声明和赋值语句； |
| 范例  （重构前） | double basePrice = anOrder.basePrice();  return (basePrice > 1000); |
| 范例  （重构后） | return (anOrder.basePrice() > 1000); |

|  |  |
| --- | --- |
| 名称 | Replace Temp with Query |
| 概要 | 将临时变量保存的表达式提炼到一个独立函数中，将这个临时变量的所有引用点替换为对新函数的调用。此后，新函数就可以被其他函数所使用。 |
| 动机 | 临时变量只在所属函数内使用，因此对外来说时不可见的。为了使得其他函数能够见到同属的一份信息，可以将临时变量替换为一个查询。 |
| 做法 | 1）找出只被赋值一次的临时变量，将其声明为final并进行编译；  2）将“对该临时变量赋值”的语句等号右侧部分提炼到一个独立函数中； |
| 范例  （重构前） | double basePrice = quantity \* itemPrice;  if (basePrice > 1000) {  return basePrice \* 0.95;  } else {  return basePrice \* 0.98;  } |
| 范例  （重构后） | if (basePrice > 1000) {  basePrice() \* 0.95;  } else {  basePrice() \* 0.98;  }  double basePrice() {  return quantity \* itemPrice;  } |

|  |  |
| --- | --- |
| 名称 | Introduce Explaining Variable |
| 概要 | 将复杂表达式（或其中的一部分）的结果放进一个临时变量，以此变量名称来解释表达式的用途 |
| 动机 | 表达式可能非常复杂难以阅读，临时变量可以帮助将表达式分解为比较容易管理和理解的形式。 |
| 做法 | 1）声明一个final变量，将待分解的复杂表达式中的一部分动作的运算结果赋值给它；  2）将表达式中“运算结果”的这一部分替换为上述临时变量；  3）编译，测试并重复上述步骤； |
| 范例  （重构前） | if ((platform.toUpperCase().indexOf("MAC") > -1) &&  (browser.toUpperCase().indexOf("IE") > -1) &&  wasInitialized() && resize > 0) {  // do something  } |
| 范例  （重构后） | final boolean isMacOS = platform.toUpperCase().indexOf("MAC") > -1;  final boolean isIEBrowser = browser.toUpperCase().indexOf("IE") > -1;  final boolean wasResized = resize > 0;  if (isMacOS && isIEBrowser && wasInitialized() && wasResized) {\  // do something  } |

|  |  |
| --- | --- |
| 名称 | Split Temporary Variable |
| 概要 | 针对每一次赋值，创造一个独立、对应的临时变量 |
| 动机 | 某些对临时变量的用途可能导致临时变量多次赋值，例如循环变量会随着循环的每次进行而改变；结果收集变量会将整个函数的运算构成某个数值后再收集起来。多次被赋值的临时变量承担了多种责任，就应当被替换分解为多个临时变量，每个变量值承担一个责任。 |
| 做法 | 1）再待分解临时变量的声明以及第一次被赋值处修改其名称，并将新的临时变量声明为final；  2）以该临时变量的第二次赋值动作为界，修改此前对该临时变量的所有引用点，让它们引用新的临时变量；  3）编译，测试并重复前述步骤； |
| 范例  （重构前） | double temp = 2 \* (height + width);  System.out.println(temp);  temp = height \* width;  System.out.println(temp); |
| 范例  （重构后） | final double perimeter = 2 \* (height + width);  System.out.println(perimeter);  final double area = height \* width;  System.out.println(area); |

|  |  |
| --- | --- |
| 名称 | Remove Assignments to Parameters |
| 概要 | 用一个临时变量取代该参数的位置 |
| 动机 | 如果代码的语义是按引用传递的，就在调用端检查调用后是否使用了这个参数。也要检查有多少个按引用传递的参数被赋值后又被使用。尽量使用一个return 来返回一个值。如果返回的值比较多，可以将返回的数据封装成一个对象，或者是干脆为每一个返回值设置一个独立函数 |
| 做法 | 1）建立一个临时变量，把待处理的参数值赋予它；  2）以“对参数赋值”为界，将其后所有对此参数的引用点，全部替换成“对此临时变量的引用”；  3）修改赋值语句，使其改为对新建之临时变量赋值； |
| 范例  （重构前） | Int discount(int inputVal, int quantity, int yearToDate){  If (intputVal > 50) inputVal -= 2;  } |
| 范例  （重构后） | Int discount(int inputVal, int quantity, int yearToDate){  Int result = inputVal  If (result > 50) result -= 2;  } |

|  |  |
| --- | --- |
| 名称 | Replace Method with Method Object |
| 概要 | 将这个函数放进一个单独对象中，如此一来局部变量就成了对象内的字段。然后你可以在同一个对象中将这个大型函数分解成多个小型函数 |
| 动机 | 局部变量会增加函数分解的难度，通常情况下我们可以使用Replace Temp with Query消除局部变量，但是在该方法失效的情况下，我们就应当使用当前的手法抽取函数对象。Replace Method with Method Object会将所有局部变量都变成函数对象的字段。然后就可以对这个对象使用Extract Method创造出新的函数，从而将原本大型的函数拆解成短小函数。 |
| 做法 | 1）建立一个新类，根据待处理函数的用途，为这个类命名；  2）在新类中建立一个final字段，用以保存原先大型函数所在的对象。我们将这个字段称为“源对象”。同时，针对原函数的每个临时变量和每个参数，在新类中建立一个对应的字段保存之；  3）在新类中建立一个构造函数，接收源对象以及原函数的所有参数作为参数；  4）在新类中建立一个compute函数，并将原函数中的代码复制到compute函数中，如果需要调用源对象的任何函数，可以通过源对象字段调用；  5）编译； |
| 范例  （重构前） | Class Account {  int gamma(int inputVal, int quantity, int yearToDate) {  int importantValue1 = (inputVal \* quantity) + delta();  int importantValue2 = (inputVal \* yearToDate) + 100;  if (yearToDate - importantValue1 > 100) {  importantValue2 -= 20;  }  int importantValue3 = importantValue2 \* 70;  return importantValue3 - 2 \* importantValue1;  }  } |
| 范例  （重构后） | Class Account {  int gamma(int inputVal, int quantity, int yearToDate) {  return new Gamma(this, inputVal, quantity, yearToDate).compute();  }  }  Class Gamma {  // Step 1 & Step 2  private final Account account;  private final int inputVal;  private final int quantity;  private final int yearToDate;  private final int importantValue1;  private final int importantValue2;  private final int importantValue3;  // Step 2:  public Gamma(Account accountArg, int inputValArg, int quantityArg, int yearToDateArg) {  account = accountArg;  inputVal = inputValArg;  quantity = quantityArg;  yearToDate = yearToDateArg;  }  // Step 3 & Step 4  public int compute() {  int importantValue1 = (inputVal \* quantity) + delta();  int importantValue2 = (inputVal \* yearToDate) + 100;  if (yearToDate - importantValue1 > 100) {  importantValue2 -= 20;  }  int importantValue3 = importantValue2 \* 70;  return importantValue3 - 2 \* importantValue1;  }  } |

|  |  |
| --- | --- |
| 名称 | Substitute Algorithm |
| 概要 | 将函数本体替换成另一个算法 |
| 动机 | 如果做一件事情可以有更清晰的方式，就应该使用更清晰的方式代替复杂的方式。当然在替换前你需要将复杂的大块代码替换成简单的小块，再进行替换。 |
| 做法 | 1）准备好替换用的算法；  2）针对现有测试，执行上述新算法，如果结果与原本相同，重构结束；  3）如果测试结果不同于原先，在测试和调试的过程中，以旧算法作为比较参照标准。 |
| 范例  （重构前） | String foundPerson(String[] people) {  for (int i = 0; i < people.length; i++) {  if (people[i].equals("Don")) {  return "Don";  }  if (people[i].equals("John")) {  return "John";  }  if (people[i].equals("Kent")) {  return "Kent";  }  }  return "";  } |
| 范例  （重构后） | String foundPerson(String[] people) {  List candidates = Arrays.asList(new String[] {"Don", "John", "Kent"});  for (int i = 0; i < people.length; i++) {  if (candidates.contains(people[i])) {  return people[i];  }  }  return "";  } |