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. Move responsibility between source object and target object

|  |  |
| --- | --- |
| 名称 | Move Method |
| 概要 | 在该函数最常引用的类中建立一个有着类似行为的新函数。将旧函数变成一个单纯地委托函数，或者是将旧函数完全移除 |
| 动机 | 一个类有太多行为或者两个类之间高度耦合。通过这种重构手段来降低系统复杂度，对于一个使用另一个对象的次数比使用自己所驻对象次数还要多的函数。 |
| 做法 | 1）检查源类中被源函数所使用的一切特性（包括字段和函数），考虑其是否应该被搬移；  2）检查源类的子类和超类，看看是否还有该函数的其他声明；  3）在目标中使用该新函数；  4）将源函数的代码复制到目标函数中，调整后者，使其在新家中正常运行；  5）决定如何从源函数正确引用目标对象；  6）修改源函数，使之成为一个纯委托函数；  7）决定是否删除源函数，或者将其作为一个委托函数保留下来；  8）如果要移除源函数，请将源类中所有对源函数的调用替换成对目标函数的调用； |
| 范例  （重构前） | class Account {  private AccountType type;  private int daysOverdrawn;    **// Moving overdraftCharge into AccountType Class**  double overdraftCharge(){  if (type.isPremium()){  double result = 10;  if (daysOverdrawn > 7) {  result += (daysOverdrawn - 7) \* 0.85;  return result;  }  } else {  return daysOverdrawn \* 1.75;  }  }  double bankCharge(){  double result = 4.5;  if (daysOverdrawn > 0)  result += overdraftCharge();  return result;  }  } |
| 范例  （重构后） | **// Moving overdraftCharge() into AccountType, Preserve original.**  class AccountType {  double overdraftCharge(){  if (type.isPremium()){  double result = 10;  if (daysOverdrawn > 7) {  result += (daysOverdrawn - 7) \* 0.85;  return result;  }  } else {  return daysOverdrawn \* 1.75;  }  }  }  // Adjust original call for original  class Account {  … …  double overdraftCharge(){  return type.overdraftCharge(daysOverdrawn);  }  double bankCharge(){  double result = 4.5;  if (daysOverdrawn > 0)  **result += overdraftCharge();**  // result += type.overdraftCharge(daysOverdrawn);  return result;  }  } |

|  |  |
| --- | --- |
| 名称 | Move Field |
| 概要 | 在目标类新建一个字段，修改源字段的所有用户，令它们改用新字段 |
| 动机 | 对于一个字段，在其所驻类之外的另一个类中有更多函数使用了它，就需要考虑搬移该字段，所谓“使用”可能是通过设值/取值函数间接进行 |
| 做法 | 1）如果字段访问级别是public，使用Encapsulate Field将其封装起来，并进行编译测试；  2）在目标类中建立源字段相同的字段，并同时建立相应的设值/取值函数；  3）编译目标类，决定如何在源对象中引用目标对象；  4）删除源字段，将所有对源字段的引用替换为对某个目标函数的调用； |
| 范例  （重构前） | **// I want to move interestRate field into AccountType**  class Account {  private AccountType type;  private double interestRate;  double interestForAmountDays(double amount, int days) {  return interestRate \* amount \* days / 365;  }  } |
| 范例  （重构后） | class Account {  private AccountType type;  double interestForAmountDays(double amount, int days) {  return type.getInterestRate() \* amount \* days / 365;  }  }  class AccountType {  private double interestRate;  void setInterestRate(double arg) {  interestRate = arg;  }  double getInterestRate() {  return interestRate;  }  } |

|  |  |
| --- | --- |
| 名称 | Extract Class（改善并发程序常用的技术） |
| 概要 | 建立一个新类，将相关的字段和函数从旧类搬移到新类 |
| 动机 | 一个类在开发过程中职责膨胀而导致代码混乱，从而难以理解；另外一种情况就是开发后期出现子类化的方式，如果子类化只影响类的部分特性或者某些特性需要以一种方式来子类化，就需要分解原来的类。 |
| 做法 | 1）决定如何分解类所负的责任；  2）建立一个新类，用以表现从旧类中分离出来的责任；  3）建立“从旧类访问新类”的连接关系；  4）对于想要搬移的字段，运用Move Field方法搬移之，并编译测试；  5）使用Move Method将必要函数搬移到新类，先搬移较低层函数（被其他函数调用多于调用其他函数），在搬移较高层的函数；  6）每次搬移之后，编译测试；  7）决定是否公开新类。如果需要公开就决定让它成为引用对象还是不可变对象。 |
| 范例  （重构前） | class Person {  public String getName() {  return name;  }  // Spearate behavior of telephone  public String getTelephoneNumber() {  return ("(" + officeAreaCode + ")" + officeNumber);  }  public getOfficeAreaCode() {  return officeAreaCode;  }  public void setOfficeAreaCode(String arg) {  officeAreaCode = arg;  }  public String getOfficeNumber() {  return officeNumber;  }  public void setOfficeNumber(String arg) {  officeNumber = arg;  }  private String name;  private String officeAreaCode;  private String officeNumber;  } |
| 范例  （重构后） | class Person {  public String getName() {  return name;  }  public String getTelephoneNumber() {  return officeTelephone.getTelephoneNumber();  }  TelephoneNumber getOfficeTelephone() {  return officeTelephone;  }    private String name;  private TelephoneNumber officeTelephone = new TelephoneNumber();  }  class TelephoneNumber {  public String getTelephoneNumber() {  return ("(" + getOfficeAreaCode() + ")" + officeNumber);  }  public getOfficeAreaCode() {  return officeAreaCode;  }  public void setOfficeAreaCode(String arg) {  officeAreaCode = arg;  }  public String getOfficeNumber() {  return officeNumber;  }  public void setOfficeNumber(String arg) {  officeNumber = arg;  }  private String officeAreaCode;  private String officeNumber;  } |

|  |  |
| --- | --- |
| 名称 | Inline Class |
| 概要 | 将这个类的所有特性搬移到另一个类中，然后移除原类 |
| 动机 | 如果一个类不再承担足够的责任，不再有单独存在的理由。就需要将这个不再承担责任的类塞进另外一个类。 |
| 做法 | 1）在目标类身上声明源类的public协议，并将其中的所有函数委托至原函数  2）修改所有源类引用点，改而引用目标类；  3）编译测试；  4）运用Move Method和Move Field，将源类的特性全部搬移至目标类；  5）为源类举行一个简单地“丧礼”； |
| 范例  （重构前） | class Person {  public String getName() {  return name;  }  public String getTelephoneNumber(){  return officeTelephone.getTelephoneNumber();  }  public TelephoneNumber getOfficeTelephone() {  return officeTelephone;  }  private String name;  private TelephoneNumber officeTelephone = new TelephoneNumber();  }  class TelephoneNumber {  public String getTelephoneNumber() {  return ("(" + areaCode + ")" + number);  }  public String getAreaCode() {  return areaCode;  }  public void setAreaCode(String arg) {  areaCode = arg;  }  public String getNumber() {  return number;  }  public void setNumber(String number) {  number = arg;  }  private String number;  private String areaCode;  } |
| 范例  （重构后） | class Person {  public String getName() {  return name;  }  public String getTelephoneNumber(){  return officeTelephone.getTelephoneNumber();  }  public TelephoneNumber getOfficeTelephone() {  return officeTelephone;  }  public String getAreaCode() {  return officeTelephone.getAreaCode();  }  public void setAreaCode(String arg) {  officeTelephone.setAreaCode(arg);  }  public String getNumber() {  return officeTelephone.getNumber();  }  public void setNumber(String number) {  officeTelephone..setNumber(number);  }  private String name;  private TelephoneNumber officeTelephone = new TelephoneNumber();  }  class TelephoneNumber {  public String getTelephoneNumber() {  return ("(" + areaCode + ")" + number);  }  public String getAreaCode() {  return areaCode;  }  public void setAreaCode(String arg) {  areaCode = arg;  }  public String getNumber() {  return number;  }  public void setNumber(String number) {  number = arg;  }  private String number;  private String areaCode;  } |

|  |  |
| --- | --- |
| 名称 | Hide Delegate |
| 概要 | 在服务类上建立客户所需的函数，用以隐藏委托关系。 |
| 动机 | 如果某个客户先通过对象的字段得到另一个对象，然后调用后者的函数，那么客户就必须知道这一层委托关系。万一委托关系发生变化，客户也得跟着变化。你可以在服务对象上放置一个简单的委托函数，将委托函数隐藏起来，从而去除依赖。 |
| 做法 | 1）对于每一个委托关系中的函数，在服务对象端建立一个简单的委托函数；  2）调整客户，令它只调用服务对象提供的函数；  3）每次调整后，编译并测试；  4）如果将来不再有认可客户需要取用委托类，便可移除服务对象中的相关访问函数； |
| 范例  （重构前） | class Person {  Department department;  public Department getDepartment() {  return department;  }  public void setDepartment(Department department) {  this.department = department;  }  }  class Department {  private String chargeCode;  private Person manager;  public Department(Person manager) {  this.manager = manager;  }  public Person getManager() {  return manager;  }  } |
| 范例  （重构后） | class Person {  Department department;  public void setDepartment(Department department) {  this.department = department;  }  public Person getManager() {  return department.getManager();  }  }  class Department {  private String chargeCode;  private Person manager;  public Department(Person manager) {  this.manager = manager;  }  public Person getManager() {  return manager;  }  } |

|  |  |
| --- | --- |
| 名称 | Remove Middle Man |
| 概要 | 让客户直接调用受委托类 |
| 动机 | 封装过度导致服务类已经完全变为一个“中间人”，此时就应该让客户直接调用受托类，随着系统的变化，合适的隐藏程度也应得到改变。 |
| 做法 | 1）建立一个函数，用以获得受托对象；  2）对于每一个委托函数，在服务类中删除该函数，并让需要调用该函数的客户转为调用受托对象；  3）处理每个委托函数后，编译测试； |
| 范例  （重构前） | class Person {  Department department;  public void setDepartment(Department department) {  this.department = department;  }  public Person getManager() {  return department.getManager();  }  }  class Department {  private String chargeCode;  private Person manager;  public Department(Person manager) {  this.manager = manager;  }  public Person getManager() {  return manager;  }  } |
| 范例  （重构后） | class Person {  Department department;  public void setDepartment(Department department) {  this.department = department;  }  public Person getManager() {  return department.getManager();  }  public Department getDepartment() {  return department;  }  }  class Department {  private String chargeCode;  private Person manager;  public Department(Person manager) {  this.manager = manager;  }  public Person getManager() {  return manager;  }  } |

|  |  |
| --- | --- |
| 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 "";  } |