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 | Introduce Foreign Method |
| 概要 | 在客户类中简历一个函数，并以第一参数形式传入一个服务类实例 |
| 动机 | 为本不属于待修改的服务类中添加我需要提供的职责，这样炒出服务类本身职责的函数我们称之为外加函数。外加函数要考虑通用性，如果是系统多处调用，可以做成公共模块。 |
| 做法 | 1）在客户类中建立一个函数，用来提供你需要的功能；  2）以服务类实例作为函数的第一参数；  3）将该函数注释为“外加函数”，应在服务类中实现； |
| 范例  （重构前） | Date newStart = new Date(previousEnd.getYear(), previousEnd.getMonth(), previousEnd.getDate() + 1); |
| 范例  （重构后） | Date newStart = nextDay(previousEnd);  private static Date nextDay(Date arg) {  return new Date(previousEnd.getYear(), previousEnd.getMonth(), previousEnd.getDate() + 1);  } |

|  |  |
| --- | --- |
| 名称 | Introduce Local Extension |
| 概要 | 建立一个新类，使它包含这些额外函数，让这个扩展品成为源类的子类或者包装类； |
| 动机 | 在对代码的更改过程中，如果源类中外加函数的调用超过两个则无法管理。因此使用一个包装类去管理这些外加函数，所有的更改都限制在扩展类中。 |
| 做法 | 1）建立一个扩展类，将它们作为原始类的子类或者包装类；  2）在扩展类中加入转型构造函数；  3）在扩展类中加入新特性；  4）根据需要，将源对象替换为扩展对象；  5）将针对原始类定义的所有外加函数搬移到扩展类中； |
| 范例  （重构前） | 以Java 1.0.1中的Date类为例； |
| 范例  （重构后） | class MfDateSub extends Date {  public MfDateSub(String dateString) {  super(dateString);  }  // 转型构造函数  public MfDateSub(Date arg) {  super(arg.getTime())  }  public MfDateSub nextDay(Date arg) {  // To-Do  return new Date(arg.getYear(), arg.getMonth(), arg.getDate() + 1);  }  public int dayOfYear() {  // To-Do  }  }  class MfDateWrap {  private Date original;  Date nextDay() {  return new Date(getYear(), getMonth(), getDate() + 1);  }  } |