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. Re-Composite Data

|  |  |
| --- | --- |
| 名称 | Self Encapsulate Field |
| 概要 | 为这个字段建立设值/取值函数，并且只以这些函数来访问字段 |
| 动机 | 使用访问变量的好处在于子类可以通过覆写一个函数而改变获取数据的途径，并且支持灵活的数据管理方式。最好的地方在于代码便于阅读。 |
| 做法 | 1）为带封装的字段建立设值/取值函数；  2）找出该字段的所有引用点，将它们全部改为调用取值/设值函数；  3）声明字段为private，复查引用点后编译测试； |
| 范例  （重构前） | class IntRange {  private int low;  private int high;  boolean includes(int arg) {  return arg >= low && arg <= high;  }  } |
| 范例  （重构后） | class IntRange {  private int low;  private int high;  boolean includes(int arg) {  return arg >= getLow() && arg <= getHigh();  }  public int getLow() {  return low;  }  public int getHigh() {  return high;  }  } |

|  |  |
| --- | --- |
| 名称 | Replace Data Value with Object |
| 概要 | 有一个数据项，需要与其他数据的行为一起使用才有意义，将数据项变为对象； |
| 动机 | 初期代码开发会使用基本类型，随着代码演进出现越来越多的特殊行为。业务中许多基本字段构成的对象是业务最小不可分割的实体。因此需要将多个字段封装成一个对象； |
| 做法 | 1）为待替换值新建一个类，在其中声明一个final字段，其类型和源类型中的待替换数值类型一样；  2）在新类中加入这个字段的取值函数，再加上一个接受此字段为参数的构造函数；  3）将源类中待替换数值字段的类型改为前面新建的类；  4）修改源类中该字段的取值函数，令它们调用新类的取值函数；  5）修改源类中待替换字段的设值函数，令它为新类创建一个实例； |
| 范例  （重构前） | class Order {  public Order(String customer) {  this.customer = customer;  }  public String getCustomer() {  return customer;  }  public void setCustomer(String customer) {  this.customer = customer;  }  private String customer;  } |
| 范例  （重构后） | class Order {  public Order(String customer) {  this.customer = customer;  }  public String getCustomer() {  return customer.getName();  }  public void setCustomer(String customer) {  customer = new Customer(customer);  }  private Customer customer;  }  class Customer {  public Customer(String name) {  this.name = name;  }  public String getName() {  return name;  }  private final String name;  } |

|  |  |
| --- | --- |
| 名称 | Change Value to Reference |
| 概要 | 从一个类衍生出许多彼此相等的实例，将它们替换为同一个对象，将这个值对象变为引用对象。 |
| 动机 | 有时候从一个简单地值对象开始，其中保存了一些不可修改的数据，而后希望给这个对象加入一些可修改的数据，并确保对任何一个对象的修改都能影响到所有引用此对象的地方。这时候就需要将这个对象变为一个引用对象。 |
| 做法 | 1）使用Replace Constructor with Factory Method方法；  2）编译测试；  3）决定使用什么对象负责提供访问新对象的途径；  4）决定这些引用对象应该预先创建好，或者应该是动态创建；  5）修改工厂函数，令它们返回引用对象，并编译测试； |
| 范例  （重构前） | class Customer {  public Customer(String name) {  this.name = name;  }  public String getName() {  return name;  }  private final String name;  }  class Order {  public Order(String customerName) {  Customer = new Customer(customerName);  }  public void setCustomer(String customerName) {  customer = new Customer(customerName);  }  public String getCustomerName() {  return customer.getName();  }  private Customer customer;  public static int numberOfOrdersFor(Collection orders, String customer) {  int result = 0;  Iterator iter = orders.iterator();  while (iterator.hasNext()) {  Order each = (Order)iter.next();  if (each.getCustomerName().equals(customer))  result ++;  }  return result;  }  } |
| 范例  （重构后） | class Customer {  private Customer(String name) {  this.name = name;  }  public static Customer getNamed(String name) {  return (Customer)\_instances.get(name);  }  public String getName() {  return name;  }  static void loadCustomers() {  new Customer("Lemon Car Hire").store();  new Customer("Associated Coffee Machines").store();  new Customer("Bilston Gasworks").store();  }  private void store() {  \_instances.put(this.getName(), this);  }  private final String name;  private static Dictionary \_instances = new Hashtable<>();  }  class Order {  public Order(String customerName) {  customer = Customer.create(customerName);  }  public String getCustomerName() {  return customer.getName();  }  private Customer customer;  public static int numberOfOrdersFor(Collection orders, String customer) {  int result = 0;  Iterator iter = orders.iterator();  while (iterator.hasNext()) {  Order each = (Order)iter.next();  if (each.getCustomerName().equals(customer))  result ++;  }  return result;  }  } |

|  |  |
| --- | --- |
| 名称 | Change Reference to Value |
| 概要 | 存在一个很小且不可变的引用对象，将其变为一个值对象 |
| 动机 | 如果引用对象变得难以使用，也许就应该更改为值对象。值对象存在一个特性，其为不可变的。无论何时只要调用同一对象的同一个查询函数，都应该得到同样的结果 |
| 做法 | 1）检查重构目标是否为不可变对象，或是否可以修改为不可变对象；  2）建立equals()和hashCode()方法；  3）编译测试；  4）考虑是否可以删除工厂函数，并将构造函数声明为public; |
| 范例  （重构前） | class Currency {  private String code;  public String getCode() {  return code;  }  private Currency(String code) {  this.code = code;  }  }  // 重构前的结果：  new Current("USD").equals(new Currency("USD")) --> return false; |
| 范例  （重构后） | class Currency {  private String code;  public String getCode() {  return code;  }  private Currency(String code) {  this.code = code;  }  // equals  public boolean equals(Object arg) {  if (!(arg instanceof Currency)) {  return false;  }  Currency other = (Current)arg;  return (code.equals(other.code));  }  // hashCode  public int hashCode() {  return code.hashCode();  }  }  // 重构后的结果  new Current("USD").equals(new Currency("USD")) --> return true; |

|  |  |
| --- | --- |
| 名称 | Replace Array with Object |
| 概要 | 以对象替换数组，对于数组中的每个元素，以一个字段来表示； |
| 动机 | 数组只适用于“以某种顺序容纳一组相似对象”。一个数组容纳多种不同对象，容易给用户带来麻烦。 |
| 做法 | 1）新建一个类表示数组所拥有的信息，并在其中以一个public字段保存原先数组；  2）修改数组的所有用户，让它们改用新类的实例；  3）编译测试；  4）逐一为数组元素添加设值/取值函数。根据元素用途为这些访问函数命名，修改原先代码，让其通过设值取值函数访问；  5）编译测试通过后，将所有设值字段改为private； |
| 范例  （重构前） | String[] row = new String[3];  row[0] = "Livepool";  row[1] = "15"; |
| 范例  （重构后） | Performance row = new Performance();  row.setName("Livepool");  row.setWin(15); |

|  |  |
| --- | --- |
| 名称 | Duplicate Observed Data |
| 概要 | 将GUI控件中的领域数据复制到领域对象。建立一个Observer模式，用以同步领域对象和GUI对象内的重复数据。 |
| 动机 | 同一份数据既可能内嵌于GUI，也可能保存领域模型中，如果业务逻辑被嵌入到用户界面中，就需要将其分离出来。对于行为只需要搬移，但是对于数据，还需要进行复制，并提供相应的同步机制。 |
| 做法 | 1）修改展现类，使其成为领域类的Observer；  2）针对GUI类的领域数据，使用Self Encapsulate Field手段；  3）编译测试；  4）在事件处理函数中设置设值函数，直接更新GUI组件；  5）在领域类中定义数据及其相关访问函数；  6）修改展现类的访问函数，将它们的操作对象改为领域对象；  7）修改Observer的update()，使其从相应的领域对象中将所需数据复制给GUI组件； |
| 范例  （重构前） | 暂无 |
| 范例  （重构后） | 暂无 |

|  |  |
| --- | --- |
| 名称 | Change unidirectional Association to Bidirectional |
| 概要 | 两个类都需要使用对方的特性，但其间只有一条单向连接，添加一个反向指针，并使修改函数能够同时更新两条连接。 |
| 动机 | 同“概要”所述 |
| 做法 | 1）在被引用类中增加一个字段，用以保存反向指针；  2）决定由哪一个类——引用端还是被引用端——控制关联关系；  3）在被控制端建立一个辅助函数，其命名应该清楚所指出它的有限用途；  4）如果既有的修改函数在控制端，让它负责更新反向指针。  5）如果既有的修改函数再被控端，就在控制端建立一个控制函数，并让既有的修改函数调用这个新建的控制函数。 |
| 范例  （重构前） | class Order {  Customer getCustomer() {  return customer;  }  void setCustomer(Customer arg) {  customer = arg;  }  Customer customer;  }  class Customer {  private Set orders = new HashSet();  } |
| 范例  （重构后） | class Order {  Customer getCustomer() {  return customer;  }  void setCustomer(Customer arg) {  if (customer != null) {  customer.friendOrders().remove(this);  customer = arg;  }  if (customer != null) {  customer.friendOrders().add(this);  }  }  void addCustomer(Customer arg) {  arg.friendOrders().add(this);  }  Customer customer;  }  class Customer {  private Set orders = new HashSet();  Set friendOrders() {  return orders;  }  void addOrder(Order order) {  order.setCustomer(this);  }  } |

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