《软件工程理论与实践》

上机 1 实验报告

院系	软件学院
专业班级	软件工程 2102 班
姓名	
学号	
指导老师	

目 录

1 实验目的、内容和要求

1.1 实验名称

- 重构实验

1.2 实验目的

- 理解重构在软件开发中的作用
- 熟悉常见的代码环味道和重构方法

1.3 实验内容和要求

- 阅读：Martin Fowler 《重构-改善既有代码的设计》
- 掌握你认为最常见的 8 种代码坏味道及其重构方法
- 从你过去写过的代码或 Github 等开源代码库上寻找这 8 种坏味道，并对代码进行重构；反对拷贝别人重构例子。

2 代码重构

2.1 重构 1：神秘命名（Mysterious Name）

2.1.1 坏味道代码

```
def func(a, b):
    c = a + b
    return c
```

这段代码来自我初学 python 时写的 HelloWorld.py 文件，代码定义了一个函数，这个函数的输入是两个数 a, b；输出是 a + b。

2.1.2 坏味道说明

“神秘命名”是一种常见的坏味道，它指的是在代码中使用没有明确含义的变量、函数、类、模块等命名。这些命名往往缺乏清晰的语义和上下文，让其他开发人员难以理解代码的意图和目的。这可能导致代码可读性和可维护性的下降，并且增加了调试和重构代码的难度。

神秘命名的危害在于它会导致代码难以理解和维护。当其他开发人员需要在代码中添加新的功能或者修改代码时，他们可能需要花费更多的时间来理解这些命名的含义和作用。这可能导致开发时间的延长和错误的引入，从而降低了代码质量和开发效率。

2.1.3 重构方法

为了解决神秘命名的问题，可以采取以下重构方法：

1. 更改命名：将神秘的命名更改为更有意义和描述性的名称，使代码更易于理解和维护。
2. 引入注释：在代码中添加注释可以帮助其他开发人员理解变量、函数、类等的含义和用途，特别是在一些情况下，无法通过名称清楚地表达其意图时。

3. 优化命名规范：制定良好的命名规范并将其应用于整个代码库可以减少神秘命名的发生。在规范中，定义变量、函数、类、模块等的命名规则，并尽可能遵循这些规则。

2.1.4 重构后代码

```python
def add(a, b): # a, b represent the two addend number
    result = a + b
    return result
```

2.2 重构 2：重复代码（Duplicated Code）

2.2.1 坏味道代码

```javascript
// cow movement
if(cow.direction == 0)
{
    if(map[cow.x][cow.y - 1] == '*')
        cow.direction = (cow.direction + 1 ) % 4;
    else
    {
        cow.y--;
    }
}
else if(cow.direction == 1)
{
    if(map[cow.x + 1][cow.y] == '*')
        cow.direction = (cow.direction + 1 ) % 4;
    else
    {
        cow.x++;
    }
}
else if(cow.direction == 2)
{
    if(map[cow.x][cow.y + 1] == '*')
        cow.direction = (cow.direction + 1 ) % 4;
    else
    {
        cow.y++;
    }
}
else if(cow.direction == 3)
{
    if(map[cow.x - 1][cow.y] == '*')
        cow.direction = (cow.direction + 1 ) % 4;
    else
```

```
    {
        cow.x--;
    }
}
// farmer movement
if(framer.direction == 0)
{
    if(map[framer.x][framer.y - 1] == '*')
        framer.direction = (framer.direction + 1 ) % 4;
    else
    {
        framer.y--;
    }
}
else if(framer.direction == 1)
{
    if(map[framer.x + 1][framer.y] == '*')
        framer.direction = (framer.direction + 1 ) % 4;
    else
    {
        framer.x++;
    }
}
else if(framer.direction == 2)
{
    if(map[framer.x][framer.y + 1] == '*')
        framer.direction = (framer.direction + 1 ) % 4;
    else
    {
        framer.y++;
    }
}
else if(framer.direction == 3)
{
    if(map[framer.x - 1][framer.y] == '*')
        framer.direction = (framer.direction + 1 ) % 4;
    else
    {
        framer.x--;
    }
}
```

代码来源：我自己写的一道算法题 USACO2.4 两只塔姆沃斯牛 The Tamworth Two，这段代码的作用是控制农夫和牛在地图中方向的移动。

2.2.2 坏味道说明

“重复代码”是一种常见的坏味道，它指的是代码中存在多个相同或非常相似的代码片段。这些重复的代码可能存在于同一个文件、不同的文件或不同的代码库中，但它们执行的功能相同或者非常相似。

重复代码的危害在于它会导致代码冗余和维护困难。如果存在多个相同或相似的代码片段，每次需要修改功能时，必须修改所有重复的代码。这会增加代码的维护难度，并且可能导致错误的引入。此外，重复的代码还会占用更多的内存和磁盘空间，从而导致代码库变得更加庞大和不易维护。

2.2.3 重构方法

为了解决重复代码的问题，我们可以采取以下重构方法：

1. 提取方法：将重复的代码段提取到一个独立的方法中，并在需要时调用该方法。这可以减少重复代码并提高代码的可重用性。
2. 抽象公共方法：如果有多个代码段具有相同的结构，可以将它们抽象为一个通用方法，并在需要时使用。这可以减少重复代码的数量并提高代码的可维护性。

2.2.4 重构后代码

```
void move(object& obj)
{
    if(obj.direction == 0)
    {
        if(map[obj.x][obj.y - 1] == '*')
            obj.direction = (obj.direction + 1 ) % 4;
        else
        {
            obj.y--;
        }
    }
    else if(obj.direction == 1)
    {
        if(map[obj.x + 1][obj.y] == '*')
            obj.direction = (obj.direction + 1 ) % 4;
        else
        {
            obj.x++;
        }
```

```
    }
    else if(obj.direction == 2)
    {
        if(map[obj.x][obj.y + 1] == '*')
            obj.direction = (obj.direction + 1 ) % 4;
        else
        {
            obj.y++;
        }
    }
    else if(obj.direction == 3)
    {
        if(map[obj.x - 1][obj.y] == '*')
            obj.direction = (obj.direction + 1 ) % 4;
        else
        {
            obj.x--;
        }
    }
}

int main()
{
    ...
    move(cow);
    move(framer);
    ...
    return 0;
}
```

2.3 重构 3：过长函数（Long Function）

2.3.1 坏味道代码

```
#include <iostream>
#include <cstdio>

using namespace std;

#define re register
#define maxn (2023)

int max(int x, int y)
{
    return x > y ? x : y;
}
int a[maxn], b[maxn], ans[maxn << 1];
```

```
int main()
{
    string s1, s2;
    cin >> s1 >> s2;
    for (re int i = 0; i < s1.length(); i++)
        a[i] = s1[s1.length() - 1 - i] - '0';
    for (re int i = 0; i < s2.length(); i++)
        b[i] = s2[s2.length() - 1 - i] - '0';
    for (re int i = 0; i < s1.length(); i++)
    {
        for (re int j = 0; j < s2.length(); j++)
        {
            ans[i + j] += a[i] * b[j];
        }
    }
    for (re int i = 0; i <= (maxn << 1); i++)
    {
        if (ans[i] >= 10)
        {
            ans[i + 1] += ans[i] / 10;
            ans[i] %= 10;
        }
    }
    int tmp = maxn << 1;
    while (ans[tmp] == 0 && tmp > 0) tmp--;
    if(tmp == 0)
    {
        cout << 0;
        return 0;
    }
    for(re int i = tmp; i >= 0 ; i--)
        cout << ans[i];
    return 0;
}
```

这段代码是我利用 C 语言实现的高精度乘法，可以对两个10^{20} 的超级大整数相乘，采用数组模拟乘法的相乘与进位，思路是消耗空间换取时间。

2.3.2 坏味道说明

过长函数是指代码中某个函数过于冗长复杂，超过应有的长度限制，使得代码难以阅读、理解和维护。这种坏味道的存在会导致代码质量下降、可读性差、出错率高等问题，并且难以重用或调试。比如这段代码中一连串的 for 代码根本让人不知所云。

2.3.3 重构方法

对于过长函数的重构，考虑以下思路：

1. 拆分函数：将一个函数按照不同的职责或功能进行拆分，形成多个小函数，每个小函数只负责一项具体的工作，这样可以降低单个函数的复杂度和长度。
2. 提取方法：将函数中的某些独立操作提取为新的方法，以减少代码重复和提高重用性。
3. 优化参数列表：如果函数参数列表过长，可以考虑将其中的相关参数放置在同一个对象中，以简化函数的参数列表并提高代码可读性。
4. 使用注释：对于一些长函数，可以使用注释来标识代码的不同执行分支或处理步骤，以提高代码可读性。

2.3.4 重构后代码

```
#include <iostream>
#include <cstdio>

using namespace std;

#define re register
#define maxn (2023)

int max(int x, int y)
{
    return x > y ? x : y;
}
int a[maxn], b[maxn], ans[maxn << 1];
string s1, s2;

void multiple(int *a, int *b) // 采用数组模拟乘法的相乘与进位
{
    for (re int i = 0; i < s1.length(); i++)
    {
        for (re int j = 0; j < s2.length(); j++)
        {
            ans[i + j] += a[i] * b[j];
        }
    }
    for (re int i = 0; i <= (maxn << 1); i++)
    {
```

```
        if (ans[i] >= 10)
        {
            ans[i + 1] += ans[i] / 10;
            ans[i] %= 10;
        }
    }
}

void print() // 从高位到低位进行输出
{
    int tmp = maxn << 1;
    while (ans[tmp] == 0 && tmp > 0) tmp--;
    if(tmp == 0)
    {
        cout << 0;
        return 0;
    }
    for(re int i = tmp; i >= 0 ; i--)
        cout << ans[i];
}

int main()
{
    cin >> s1 >> s2;
    for (re int i = 0; i < s1.length(); i++)
        a[i] = s1[s1.length() - 1 - i] - '0';
    for (re int i = 0; i < s2.length(); i++)
        b[i] = s2[s2.length() - 1 - i] - '0';
    multiple(a, b);
    print();
    return 0;
}
```

2.4 重构 4：过长参数列表（Long Parameter List）

2.4.1 坏味道代码

```
def calculate_score(name, age, gender, height, weight, math_score, engl
ish_score, chinese_score):
    score = (math_score + english_score + chinese_score) / 3
    if gender == "male":
        if age > 20 and height > 175 and weight < 80:
            score += 10
    else:
        if age > 18 and height > 165 and weight < 60:
            score += 10
    return score
```

该代码来自我在学习 python 的面向对象的概念是完成的学生成绩管理系统，这段代码的作用是根据学生的性别，身高，体重，成绩等数据计算学生的分数并进行返回。

2.4.2 坏味道说明

“过长参数列表”是指函数或方法的参数数量过多或者参数类型过于复杂，导致函数声明或调用代码难以阅读和理解。这种坏味道可能导致代码的可维护性和可读性降低，同时也会增加代码的复杂度和错误的引入。

过长参数列表的危害在于它会导致代码难以理解和维护。当其他开发人员需要在代码中添加新的功能或者修改代码时，他们可能需要花费更多的时间来理解参数的作用和顺序。这可能导致开发时间的延长和错误的引入，从而降低了代码质量和开发效率。

2.4.3 重构方法

为了解决过长参数列表的问题，我们可以采取以下重构方法：

1. 重构为对象：将参数封装成一个对象，并将该对象作为参数传递给函数。这样可以减少函数的参数数量，提高代码的可读性和可维护性。
2. 使用默认值：对于一些不是必须的参数，可以设置默认值来避免在调用函数时传递参数。
3. 重构为多个函数：如果一个函数的参数过多，可以考虑将其拆分成多个较小的函数，每个函数只需要少量的参数。
4. 重构为参数对象：将多个参数封装成一个参数对象，并在函数声明中只传递一个参数对象。这样可以减少函数的参数数量，提高代码的可读性和可维护性。

通过上述重构方法，我们可以减少函数的参数数量，提高代码的可读性和可维护性，避免过长参数列表带来的问题。

2.4.4 重构后代码

```
class Student:
    def __init__(self, name, age, gender, height, weight, math_score, e
nglish_score, chinese_score):
        self.name = name
        self.age = age
        self.gender = gender
        self.height = height
        self.weight = weight
        self.math_score = math_score
        self.english_score = english_score
        self.chinese_score = chinese_score

    def calculate_score(self):
        score = (self.math_score + self.english_score + self.chinese_sc
ore) / 3
        if self.gender == "male":
            if self.age > 20 and self.height > 175 and self.weight < 80:
                score += 10
        else:
            if self.age > 18 and self.height > 165 and self.weight < 60:
                score += 10
        return score
```

2.5 重构 5：全局数据（Global Data）

2.5.1 坏味道代码

```
#include <iostream>
#include <cstdio>

using namespace std;

#define re register

int max(int x, int y){return x > y ? x : y;}

int a[505], b[505], ans[505];
int main()
{
    string s1, s2;
    int maxn;
    cin >> s1 >> s2;
    for(re int i = 0; i < s1.length(); i++)
        a[i] = s1[s1.length() - 1 - i] - '0';
    for(re int i = 0; i < s2.length(); i++)
        b[i] = s2[s2.length() - 1 - i] - '0';
```

```
    maxn = max(s1.length(), s2.length());
    // cout << maxn << endl;
    for (re int i = 0; i <= maxn; i++)
    {
        ans[i] += a[i] + b[i];
        if(ans[i] >= 10)
        {
            ans[i] -= 10;
            ans[i+1]++;
        }
    }
    if(ans[maxn] != 0) cout << ans[maxn];
    for(re int i = maxn - 1; i >= 0; i--)
        cout << ans[i];
    return 0;
}
```

2.5.2 坏味道说明

全局数据是指在整个程序中可被访问的数据，它们可以是全局变量、静态变量或常量等。这种坏味道的存在会导致代码的耦合度高、可维护性差、扩展性低等问题。

全局数据的危害包括：

1. 导致代码依赖复杂：由于全局数据可以被整个程序的任何部分引用和修改，因此当多个模块之间共享同一个全局数据时，代码的依赖关系变得非常复杂，使得代码难以理解和维护。
2. 难以进行单元测试：全局数据的存在会影响到模块的独立性，使得模块的单元测试变得困难，需要考虑全局数据的状态和影响范围。
3. 安全性问题：全局数据容易被不同模块同时访问和修改，这可能导致数据的竞争条件和安全漏洞。

2.5.3 重构方法

针对全局数据的坏味道，考虑以下重构方法：将全局数据转换为局部数据：将全局变量转化为函数参数或返回值，将静态变量转化为函数内的局部变量，这样可以减少对全局数据的依赖，提高代码的独立性和可维护性。

2.5.4 重构后代码

```
#include <iostream>
#include <cstdio>

using namespace std;

#define re register

int max(int x, int y){return x > y ? x : y;}


int *add(int *a, int *b)

// 执行按位加法，采用传递地址的方式避免使用全局变量
{
    for (re int i = 0; i <= maxn; i++)
    {
        ans[i] += a[i] + b[i];
        if(ans[i] >= 10)
        {
            ans[i] -= 10;
            ans[i+1]++;
        }
    }
}

int main()
{
    int a[505], b[505], ans[505];
    string s1, s2;
    int maxn;
    cin >> s1 >> s2;
    for(re int i = 0; i < s1.length(); i++)
        a[i] = s1[s1.length() - 1 - i] - '0';
    for(re int i = 0; i < s2.length(); i++)
        b[i] = s2[s2.length() - 1 - i] - '0';
    maxn = max(s1.length(), s2.length());
    // cout << maxn << endl;
    ans = add(a, b);
    if(ans[maxn] != 0) cout << ans[maxn];
    for(re int i = maxn - 1; i >= 0; i--)
        cout << ans[i];
    return 0;
}
```

2.6 重构 6：可变数据（Mutable Data）

2.6.1 坏味道代码

```
#include <iostream>
#include <cstdio>

using namespace std;

long long b, p, k;

long long qpow(long long a, long long b)
{
    long long ans = 1;
    while (b)
    {
        if (b & 1)
            ans = (ans * a) % k;
        a = (a * a) % k;
        b >>= 1;
    }
    ans %= k;
    return ans;
}

int main()
{
    scanf("%lld%lld%lld", &b, &p, &k);
    printf("%lld^%lld mod %lld=%lld\n", b, p, k, qpow(b, p));
    return 0;
}
```

这段代码是我自己写的快速幂，通过二分的思想将求取a^b的算法从 O(n)的复杂度降低至 O(log n)，在处理需要大量幂运算的程序中大大提高了程序的运行效率。

2.6.2 坏味道说明

可变数据是指在函数中修改了参数的值，导致代码的可读性差、维护成本高、易出现意外错误。重构的方法是尽可能避免修改参数的值，可以采用复制、封装成类等方式避免修改参数的值。

2.6.3 重构方法

要消除可变数据的坏味道，可以使用以下重构方法：将可变数据的修改范围限制在单个函数内部。通过定义局部变量，可以确保变量的生命周期仅限于该函数，并且不会泄露到程序的其他部分。

2.6.4 重构后代码

```cpp
#include <iostream>
#include <cstdio>

using namespace std;

long long b, p, k;

long long qpow(long long a, long long b)
{
    long long ans = 1;
    long long tmp = b; // 避免对参数进行直接修改
    while (b)
    {
        if (b & 1)
            ans = (ans * a) % k;
        a = (a * a) % k;
        b >>= 1;
    }
    ans %= k;
    return ans;
}

int main()
{
    scanf("%lld%lld%lld", &b, &p, &k);
    printf("%lld^%lld mod %lld=%lld\n", b, p, k, qpow(b, p));
    return 0;
}
```

2.7 重构 7：发散式变化（Divergent Change）

2.7.1 坏味道代码

```python
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height
```

```
    def get_area(self):
        return self.width * self.height


class Square:
    def __init__(self, side):
        self.side = side

    def get_area(self):
        return self.side ** 2
```

这段代码是我在学习 python 面向对象——继承的时候写的代码，代码定义了长方形类和正方形类，其中 get_area 函数用来求对应对象的面积。

2.7.2 坏味道说明

发散式变化是指修改一个类需要修改多个地方的代码，导致代码的可维护性差、扩展性差。这种坏味道的存在会导致代码的耦合度高、可维护性差、扩展性低等问题。

2.7.3 重构方法

对于发散式变化的重构，可以考虑以下思路：

1. 应用面向对象设计原则：例如单一职责原则（SRP）和开放封闭原则（OCP），确保代码的职责分明，易于扩展和维护。
2. 重构共享代码：将不同职责之间共享的代码提取出来，形成通用的类或模块，以降低代码的冗余度。

2.7.4 重构后代码

```
class Rectangle:
    def __init__(self, width, height):
        self.width = width
        self.height = height

    def get_area(self):
        return self.width * self.height


class Square(Rectangle):
    def __init__(self, side):
        super().__init__(side, side)霰弹式修改
```

2.8 重构 8：数据泥团（Data Clumps）

2.8.1 坏味道代码

```
int calculate_total_cost(int num_items, float price_per_item, float tax
_rate)
{
    float subtotal = num_items * price_per_item;
    float tax_amount = subtotal * tax_rate;
    float total_cost = subtotal + tax_amount;
    printf("Total cost: %.2f", total_cost);
}
```

2.8.2 坏味道说明

数据泥团指的是一组数据在代码中反复出现，这些数据之间存在某种联系或者依赖关系，但是却没有被单独封装起来。这样的代码会导致代码重复和可维护性下降。 在上述代码中，num_items、price_per_item、tax_rate、subtotal、tax_amount 等数据反复出现，这些数据之间存在联系，但是没有被封装起来，导致代码重复，难以维护。

2.8.3 重构方法

重构思路是将这些数据封装起来，形成一个独立的数据结构。这个数据结构可以是一个类、一个结构体、一个数组等等，具体形式取决于具体场景。 对于上述代码，我们可以将 num_items、price_per_item、tax_rate 封装成一个结构体，这样，我们就将相关的数据封装成一个 Order 结构体，并将计算 subtotal、tax_amount 的逻辑封装到了相应的函数中，代码变得更清晰、易于维护。

2.8.4 重构后代码

```
typedef struct {
    int num_items;
    float price_per_item;
    float tax_rate;
} Order;

float calculate_subtotal(Order order)
{
    return order.num_items * order.price_per_item;
}
```

```
float calculate_tax_amount(Order order)
{
    return calculate_subtotal(order) * order.tax_rate;
}

float calculate_total_cost(Order order)
{
    float subtotal = calculate_subtotal(order);
    float tax_amount = calculate_tax_amount(order);
    float total_cost = subtotal + tax_amount;
    printf("Total cost: %.2f", total_cost);
}
```

3 实验总结

通过这次重构实验，我深刻理解了重构在软件开发中的重要性。重构不仅可以改善代码的设计，提高代码的可读性和可维护性，还可以使代码更加清晰、简洁、易于扩展和修改。在重构过程中，我们需要熟悉常见的代码坏味道和重构方法，只有在深入理解和掌握了这些概念和技巧之后，才能正确地进行代码重构。

在本次实验中，我阅读了 Martin Fowler 的《重构-改善既有代码的设计》，并掌握了 8 种常见的代码坏味道及其重构方法。对于每一种坏味道，我都学会了相应的重构方法，例如，对于长方法，可以采取函数分解、函数组合以及提炼函数的方式进行重构；对于大类，可以采取提炼子类、提炼接口、提炼模块、提炼聚集以及提炼超类的方式进行重构。

在完成实验要求时，我选择了从自己过去写过的代码和 Github 等开源代码库中寻找这 8 种常见的坏味道，并对代码进行重构。在实践中，我深刻感受到了代码重构的必要性和难度。在重构过程中，需要仔细分析每一行代码以及代码之间的关系，考虑如何改进代码的结构和逻辑。同时还需要避免破坏代码的原有功能。

总之，通过这次实验，我对代码重构有了更深入的理解和认识。我认识到，在日后的编程工作中，我需要注意以下几个方面：首先，我的代码应该尽可能地简洁、可读、易于维护；其次，我应该时刻关注代码中存在的坏味道，及时采取相应的重构方法优化代码；最后，我需要不断学习和掌握新的技术和工具，以便更加高效和优雅地完成编程任务。