華中科技大學

课 程 实 验 报 告

课程名称： 软件工程理论

专业班级： 软件 2202 班

学　　号： U202217216

姓　　名： 郑德凯

报告日期： 2024/4/1

软件学院

目录

一、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 6 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 6 种坏味道，并对代码进行重构；反对拷贝别人重构例子。

二、代码重构

2.1 重构 1：神秘命名（Mysterious Name）

2.1.1 坏味道代码

```
n,m=map(int,input().split())
a=[[-1 for i in range(31)]for k in range(31)]
for i in range(31):
```

```
    a[i][1]=0
    a[i][2]=2
a[3][3]=2
def fun(n,m):
    if a[n][m]!=-1:
        return a[n][m]
    else:
        if a[n][m]==-1:
            a[n][m]=4*fun(n,m-2)
            return a[n][m]

print(fun(n,m))
```

代码来源：这个代码是我在写洛谷 p1057 题时写下的代码，其中定义的函数 fun(n,m) 没有描述其具体意义

2.1.2 坏味道说明

"神秘命名" 指的是在代码中使用了难以理解或者没有意义的命名，这使得其他开发者很难理解代码的含义和目的。好的命名应该清晰、具有描述性，并且能够准确地表达变量、函数、类等元素的作用和用途。

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：

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。

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。

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

```
n,m=map(int,input().split())
a=[[-1 for i in range(31)]for k in range(31)]
for i in range(31):
    a[i][1]=0
    a[i][2]=2
a[3][3]=2
def recursion(n,m)
    if a[n][m]!=-1:
        return a[n][m]
    else:
        if a[n][m]==-1:
            a[n][m]=4*recursion (n,m-2)
            return a[n][m]

print(recursion (n,m))
```

2.2 重构 2：重复代码（Duplicated Code）

2.2.1 坏味道代码

```
temp=input()
a=[]
while temp!='EOF':
    c=[]
    for i in range(len(temp)):
        if temp[i]!='<':
            c.append(temp[i])
        elif len(c)!=0:
                c.pop()
    a.append(c)
```

```
    temp=input()
b=[]
temp1=input()
while temp1!='EOF':
    d=[]
    for i in range(len(temp1)):
        if temp1[i]!='<':
            d.append(temp1[i])
        elif len(d)!=0:
            d.pop()
    b.append(d)
    temp1=input()
n=int(input())
ans=0
for k in range(len(a)):
    x=min(len(a[k]),len(b[k]))
    for i in (range(x)):
        if b[k][i]==a[k][i]:
           ans+=1
if n!=0:
    print(round((ans*60)/n))
else:
    print(0)
```

代码来源：本人在洛谷 p5587 题的源码，代码中我两个地方都有相似的逻辑，用来处理输入字符串 temp 和 temp1。代码有所重复

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，可以采取以下重构方法：

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。

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

```
def process_input():
    result = []
    while True:
        temp = input()
        if temp == 'EOF':
            break
        c = []
        for char in temp:
            if char != '<':
                c.append(char)
            elif c:
                c.pop()
        result.append(c)
    return result
a = process_input()
b = process_input()
n = int(input())
ans = 0
for k in range(len(a)):
```

```
        x = min(len(a[k]), len(b[k]))
        for i in range(x):
            if b[k][i] == a[k][i]:
                ans += 1
if n != 0:
        print(round((ans * 60) / n))
else:
        print(0)
```

2.3 重构 3：过长函数（Long Function）

2.3.1 坏味道代码

```
n,na,nb=map(int,input().split())
a=input().split()
b=input().split()
a=a*(n//na+1)
a=a[:n]
b=b*(n//nb+1)
b=b[:n]
def fun(a,b):
        if a==b:
            return 0
        if a==0:
            if b==1:
                return -1
            elif b==2:
                return 1
            elif b==3:
                return 1
            elif b==4:
```

```
            return -1
    elif a==1:
        if b==0:
            return 1
        elif b==2:
            return -1
        elif b==3:
            return 1
        elif b==4:
            return -1
    elif a==2:
        if b==0:
            return -1
        elif b==1:
            return 1
        elif b==3:
            return -1
        elif b==4:
            return 1
    elif a==3:
        if b==0:
            return -1
        elif b==1:
            return -1
        elif b==2:
            return 1
        elif b==4:
            return 1
    elif a==4:
```

```
        if b==0:
            return 1
        elif b==1:
            return 1
        elif b==2:
            return -1
        elif b==4:
            return -1



ans_a=0
ans_b=0
for i in range(n):
    if fun(int(a[i]),int(b[i]))==1:
        ans_a+=1
    elif fun(int(a[i]),int(b[i]))==0:
        continue
    else:
        ans_b+=1
print(ans_a,ans_b)
```

代码来源:我在洛谷 p1387 题中所写的代码，其中我的 fun 函数这段代码中一连串的 elif 代码，降低了可读性

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。

2.3.3 重构方法

对于过长函数的重构，考虑以下思路：

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。

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。

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。

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

```
def compare(a, b):
    rules = {0: {1: -1, 2: 1, 3: 1, 4: -1},
             1: {0: 1, 2: -1, 3: 1, 4: -1},
             2: {0: -1, 1: 1, 3: -1, 4: 1},
             3: {0: -1, 1: -1, 2: 1, 4: 1},
             4: {0: 1, 1: 1, 2: -1, 3: -1}}
    if a == b:
        return 0
    return rules[a][b]


n, na, nb = map(int, input().split())
a = list(map(int, input().split())) * (n // na + 1)
a = a[:n]
b = list(map(int, input().split())) * (n // nb + 1)
b = b[:n]
ans_a = sum(1 for a_val, b_val in zip(a, b) if compare(a_val, b_val) == 1)
ans_b = sum(1 for a_val, b_val in zip(a, b) if compare(a_val, b_val) == -1)
print(ans_a, ans_b)
```

2.4 重构 4: 过长参数列表（Long Parameter List）

2.4.1 坏味道代码

```
n=int(input())
a=[0 for i in range(n+1)]
b=[0 for i in range(100)]
c=[0 for i in range(100)]
d=[0 for i in range(100)]
total=0
def printf():
    global total
    if total<3:
        print(*(a[1:]))
        ##print(' '.join(str(x) for x in a[1:]))
    total+=1
def dfs(i,a,b,c,d):
    if i>n:
        printf()
        return 1
    else:
        for j in range(1,n+1):
            if b[j]==0 and c[i+j]==0 and d[j-i+n]==0:
                a[i]=j
                b[j]=1
                c[i+j]=1
                d[j-i+n]=1
                dfs(i+1,a,b,c,d)
                b[j]=0
```

```
                c[i+j]=0
                d[j-i+n]=0
dfs(1,a,b,c,d)
print(total)
```

代码来源：我对于八皇后问题的代码，其中 dfs 函数参数列表过长了

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：

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。

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。

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。

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 Parameters:
    def __init__(self, n):
        self.n = n
        self.a = [0] * (n + 1)
```

```
        self.b = [0] * 100
        self.c = [0] * 100
        self.d = [0] * 100
        self.total = 0


def printf(params):
    if params.total < 3:
        print(*params.a[1:])
    params.total += 1


def dfs(i, params):
    if i > params.n:
        printf(params)
        return
    else:
        for j in range(1, params.n + 1):
            if params.b[j] == 0 and params.c[i + j] == 0 and params.d[j - i + params.n] == 0:
                params.a[i] = j
                params.b[j] = 1
                params.c[i + j] = 1
                params.d[j - i + params.n] = 1
                dfs(i + 1, params)
                params.b[j] = 0
                params.c[i + j] = 0
                params.d[j - i + params.n] = 0


n = int(input())
params = Parameters(n)
dfs(1, params)
```

print(params.total)

2.5 重构 5: 全局数据（Global Data）

2.5.1 坏味道代码

```
n,m=map(int,input().split())

a=[[0 for i in range(n+1)]for k in range(n+1)]

b=[0 for i in range(n+1)]

p=[0 for i in range(n+1)]

for i in range(m):

x,y=map(int,input().split())

a[x][y]=1

a[y][x]=1

u,v=map(int,input().split())

cont=0

def dfs(i,a,b):

global v

global cont

global p

if i==v:

cont+=1

for k in range(1,n+1):

if b[k]==1:

p[k]+=1

return 1

else:

for j in range(1,n+1):

if a[i][j]==1 and j!=i and b[j]==0:

b[j]=1

dfs(j,a,b)
```

```
b[j]=0

ans=0

dfs(u,a,b)

print(p)

for i in range(1,n+1):

if p[i]==cont:

ans+=1

print(ans-1)
```

代码来源：本人所写洛谷 p8604 题的代码，其中 v,cont,p 都被我用来作全局变量

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：

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。

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。

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

```
n, m = map(int, input().split())

a = [[0 for i in range(n + 1)] for k in range(n + 1)]

b = [0 for i in range(n + 1)]

p = [0 for i in range(n + 1)]
```

```
for i in range(m):
    x,y=map(int,input().split())
    a[x][y]=1
    a[y][x]=1


def dfs(i, a, b, cont, p, v):
    if i == v:
        cont[0] += 1
        for k in range(1, n + 1):
            if b[k] == 1:
                p[k] += 1
        return 1
    else:
        for j in range(1, n + 1):
            if a[i][j] == 1 and j != i and b[j] == 0:
                b[j] = 1
                dfs(j, a, b, cont, p, v)
                b[j] = 0


u, v = map(int, input().split())
cont = [0]  # 使用列表来模拟可变类型的参数
dfs(u, a, b, cont, p, v)
print(p)
ans = 0
for i in range(1, n + 1):
    if p[i] == cont[0]:
        ans += 1
print(ans - 1)
```

2.6 可变数据

2.6.1 坏味道代码

```
#include <iostream>

using namespace std;

typedef long long ll;

ll x, y, p;

ll quickpow() {

    ll m = 1;

    while (y > 0) {

        if (y % 2 == 1) {

            m = (m * x) % p;

        }

        x = (x * x) % p;

        y /= 2;

    }

    return m;

}

int main() {

    cin >> x >> y >> p;

    ll result = quickpow();

    cout << x << "^" << y << " mod " << p << " = " << result << endl;

    return 0;

}
```

代码来源：是我初学 c++时写的快速幂模板代码，其中 quickpow 函数直接修改了外部变量的值

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

```
#include <iostream>

using namespace std;

typedef long long ll;

ll x, y, p;

ll quickpow(x,y,p)

    ll m = 1;

    ll n = x

    while (y > 0) {

        if (y % 2 == 1) {

            m = (m * n )%p

        }

        n=(n*n)%p

        y /= 2;

    }

    return m;

}

int main() {

    cin >> x >> y >> p;

    ll result = quickpow(x,y,p)

    cout << x << "^" << y << " mod " << p << " = " << result << endl;

    return 0;

}
```

三、实验总结

通过本次实验，我对重构技术有了更深入的了解和实践，提升了自己的编程能力和代码质量意识。

在本次实验中，我阅读了《重构 – 改善既有代码的设计》一书，深刻认识到重构对于提高代码质量、可维护性和可读性的重要性。通过将重构技术应用到实际代码中，我意识到重构不仅仅是一种技术手段，更是一种提高开发效率和代码质量的重要方法。

完成实验时，我深入学习了书中描述的常见代码坏味道，包括过长函数、过长参数列表、重复代码、神秘命名等，并掌握了相应的重构方法。这些知识使我能够更准确地识别代码中存在的问题，并有针对性地进行重构优化。因此，在未来的编程实践中，我将继续运用所学的重构技术，不断优化和改进自己的代码，提高软件开发效率和质量。

总的来说，本次重构实践为我提供了宝贵的学习机会，让我深刻体会到了重构对于代码质量和开发效率的重要性，也使我对自己的编程技能和软件工程能力有了更深层次的认识和提升。