華中科技大學

课 程 实 验 报 告

课程名称： 软件工程设计实验

专业班级：软件 2203 班

学　　号：U202217238

姓　　名：刘杰

指导教师：刘小峰

报告日期：2024 年 4 月 1 日

软件学院

目　录

1　重构实验

1.1 实验目的

（1）理解重构在软件开发中的作用

（2）熟悉常见的代码坏味道和重构方法

1.2 实验内容及要求

（1）阅读：Martin Fowler 《重构-改善既有代码的设计》

（2）掌握你认为最常见的 6 种代码坏味道及其重构方法

（3）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；反对拷贝别人的重构代码例子

1.3 实验内容报告

一．神秘命名

（1）坏味道的代码

```
#include<stdio.h>
void func(char* a);
int main()
{
    func("Hello World!");
   return 0;
}
void func(char* a)
{
    printf("%s", a);
}
```

（2）什么坏味道

该代码的坏味道是“神秘命名”。“神秘命名”指的是变量、函数、类等的命名

不清晰、直观，不能准确地反映其用途或含义。这样的命名方式会导致其他开发者难以理解代码，增加阅读和维护的难度。

（3）代码来源

这部分代码来自于大一学习 C 语言时第一次认识函数定义，声明，调用时写的。

（4）重构方法

1. 使用有意义的名称：变量、函数、类等应该使用具有描述性的名称，能够清晰地表达其用途或含义。
2. 避免使用缩写和简写：除非在广泛接受和使用的缩写或简写情况下，否则应该避免使用缩写和简写，以免增加理解难度。
3. 使用统一的命名规范：在团队中应该建立统一的命名规范，以确保代码风格的一致性和可读性。

（5）重构代码

```
#include<stdio.h>
void printString(char* string);
int main()
{
    printString("Hello World!");
   return 0;
}
void printString(char* string)
{
    printf("%s", string);

}
```

二．重复代码

（1）坏味道的代码

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
```

```
struct exist {
    int val;
    int target;
};
int func(int n, int a, int b, int c, struct exist* list);
int main(int argc, char* argv[])
{
    // 请在此输入您的代码
    int n = 0 , a = 0, b = 0, c = 0;
    scanf("%d\n%d %d %d", &n, &a, &b, &c);
    struct exist* list = malloc(sizeof(struct exist) * n);
    for (int i = 0; i < n; i++) {
        list[i].target = -1;
    }
    printf("%d", func(n, a, b, c, list) % 1000000007);
    return 0;
}
int func(int n, int a, int b, int c, struct exist* list) {
    if (n == 0)return 1;
    if ((fmin(fmin(a, b), c)) > n)return 0;
    int result = 0, resulta = 0, resultb = 0, resultc = 0;
    if (n - a >= 0) {
        if (list[n - a].target == -1) {
            resulta = func(n - a, a, b, c, list);
            list[n - a].val = resulta;
            list[n - a].target = 0;
        }
        else resulta = list[n - a].val;
    }
    if (n - b >= 0) {
        if (list[n - b].target == -1) {
            resultb = func(n - b, a, b, c, list);
            list[n - b].val = resultb;
            list[n - b].target = 0;
        }
        else resultb = list[n - b].val;
    }
    if (n - c >= 0) {
        if (list[n - c].target == -1) {
            resultc = func(n - c, a, b, c, list);
            list[n - c].val = resultc;
            list[n - c].target = 0;
        }
        else resultc = list[n - c].val;
```

```
    }
    result = resulta + resultb + resultc;
    return result;
}
```

（2）什么坏味道

这个代码的坏味道是“重复代码”，重复代码指的是在多个地方出现相同或相似的代码段落，这会导致代码冗余、难以维护，并且增加出错的概率。重复代码会导致出现代码维护困难，代码体积增大，代码质量降低等问题。

（3）代码来源

该段代码来自于我参加的第 15 届蓝侨杯第三期模拟赛题单中的“台阶方案问题”

（4）重构方法

1. 提取方法（Extract Method）：将重复的代码段落提取到一个独立的方法中，并在需要的地方调用该方法。

2. 提取类（Extract Class）：如果重复代码涉及多个方法或属性，可以考虑将其提取到一个新的类中，实现更好的封装和重用。

3. 使用继承或接口：通过继承或接口的方式，将公共代码放在父类或接口中，子类或实现类可以共享这些代码。

（5）重构代码

```
#define _CRT_SECURE_NO_WARNINGS
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
struct exist {
    int val;
    int target;
};
int func(int n, int a, int b, int c, struct exist* list);
int recursion(int n,int num,int a,int b,int c,struct exist* list);
int main(int argc, char* argv[])
```

```
{
    // 请在此输入您的代码
    int n = 0 , a = 0, b = 0, c = 0;
    scanf("%d\n%d %d %d", &n, &a, &b, &c);
    struct exist* list = malloc(sizeof(struct exist) * n);
    for (int i = 0; i < n; i++) {
        list[i].target = -1;
    }
    printf("%d", func(n, a, b, c, list) % 1000000007);
    return 0;
}
int func(int n, int a, int b, int c, struct exist* list) {
    if (n == 0)return 1;
    if ((fmin(fmin(a, b), c)) > n)return 0;
    int result = 0, resulta = 0, resultb = 0, resultc = 0;
    resulta = recursion(n,a,a,b,c,list);
    resultb = recursion(n,b,a,b,c,list);
    resultc = recursion(n,c,a,b,c,list);
    result = resulta + resultb + resultc;
    return result;
}

int recursion(int n,int num,int a,int b,int c,struct exist* list)
{
if (n - num >= 0) {
        if (list[n - num].target == -1) {
            result = func(n - num, a, b, c, list);
            list[n - num].val = result;
            list[n - num].target = 0;
        }
        else result = list[n - num].val;
    }
    return result;
}
```

三．无注释代码

（1）坏味道的代码

```
import numpy as np

from matplotlib import pyplot as plt
```

```
train = np.loadtxt('D:\AIStudy\click.csv',delimiter = ',',dtype
= 'int',skiprows = 1)
train_x = train[:,0]
train_y = train[:,1]
'''plt.plot(train_x,train_y,'o')
plt.show()'''
theta0 = np.random.rand()
theta1 = np.random.rand()
print(theta0,theta1)


def func_predict(x):
    return theta0 + theta1 * x


def func_target(x,y):
    return 0.5 * np.sum((y - func_predict(x)) ** 2)


mu = train_x.mean()
sigma = train_x.std()


def standardize(x):
    return (x - mu) / sigma


train_z = standardize(train_x)
'''plt.plot(train_z,train_y,'o')
plt.show()'''
```

```
ETA = 1e-3
diff = 1
count = 0
error = func_target(train_z,train_y)
while diff > 1e-2:
    temp0 = theta0 - ETA * np.sum(func_predict(train_z) - train_y)
    temp1 = theta1 - ETA * np.sum((func_predict(train_z) - train_y)
* train_z)
    theta0 = temp0
    theta1 = temp1
    current_error = func_target(train_z,train_y)
    diff = error - current_error
    error = current_error
    count += 1
    print(f'第{count}次，theta0 = {theta0},theta1 = {theta1},差值
= {diff}')
x = np.linspace(-3,3,2)
# print(x)
plt.plot(train_z,train_y,'o')
plt.plot(x,func_predict(x))
plt.show()
print(func_predict(standardize(100)))
print(func_predict(standardize(200)))
print(func_predict(standardize(300)))
```

（2）什么坏味道

这段代码的坏味道是没有注释。对于一段长代码来说，没有注释会导致代码的可读性，可靠性以及可重用性大大降低，对于其他程序员来说，可能会很难理解这段代码以至于无法正确进行复用，在真正的软件开发过程中，会严重影响开发进

度。

（3）代码来源

这段代码来自于我第一次实现机器学习当中的线性回归算法时所编写

（4）重构方法

对代码当中重要的函数，变量，代码段增添相关必要性注释。

（5）重构代码

```
# 导入相关依赖包
import numpy as np
from matplotlib import pyplot as plt
# 得到训练数据
train = np.loadtxt('D:\AIStudy\click.csv',delimiter = ',',dtype
= 'int',skiprows = 1)
train_x = train[:,0]
train_y = train[:,1]
'''plt.plot(train_x,train_y,'o')
plt.show()'''
# 初始化参数
theta0 = np.random.rand()
theta1 = np.random.rand()
print(theta0,theta1)



# 预测函数
def func_predict(x):
    return theta0 + theta1 * x
```

```
# 目标函数
def func_target(x,y):
    return 0.5 * np.sum((y - func_predict(x)) ** 2)


# 得到训练数据的中位数和标准差
mu = train_x.mean()
sigma = train_x.std()


# 标准化数据函数
def standardize(x):
    return (x - mu) / sigma


train_z = standardize(train_x)
'''plt.plot(train_z,train_y,'o')
plt.show()'''
# 学习率
ETA = 1e-3
# 误差
diff = 1
# 学习次数
count = 0
error = func_target(train_z,train_y)
# 循环学习，更新参数
while diff > 1e-2:
    temp0 = theta0 - ETA * np.sum(func_predict(train_z) - train_y)
```

```
    temp1 = theta1 - ETA * np.sum((func_predict(train_z) - train_y)
* train_z)
    theta0 = temp0
    theta1 = temp1
    current_error = func_target(train_z,train_y)
    diff = error - current_error
    error = current_error
    count += 1
    print(f'第{count}次，theta0 = {theta0},theta1 = {theta1},差值
= {diff}')
x = np.linspace(-3,3,2)
# print(x)
# 画图
plt.plot(train_z,train_y,'o')
plt.plot(x,func_predict(x))
plt.show()
# 检验预测函数的正确性
print(func_predict(standardize(100)))
print(func_predict(standardize(200)))
print(func_predict(standardize(300)))
```

四．函数参数列表过长

（1）坏味道的代码

```
import numpy as np
theta0 = np.random.rand()
theta1 = np.random.rand()
theta2 = np.random.rand()
theta3 = np.random.rand()
```

```
theta4 = np.random.rand()
theta5 = np.random.rand()
theta6 = np.random.rand()
theta7 = np.random.rand()
theta8 = np.random.rand()
theta9 = np.random.rand()
theta10 = np.random.rand()
def
func_predict(x,theta0,theta1,theta2,theta3,theta4,theta5,theta
6,theta7,theta8,theta9,theta10):
    # 预测函数形式为因变量的 10 次多项式
    return theta0 + \
        theta1 * x + \
        theta2 * x ** 2 +\
        theta3 * x ** 3 + \
        theta4 * x ** 4 + \
        theta5 * x ** 5 + \
        theta6 * x ** 6 + \
        theta7 * x ** 7 + \
        theta8 * x ** 8 + \
        theta9 * x ** 9 + \
        theta10 * x ** 10
```

（2）什么坏味道

此代码的坏味道是函数的参数列表过长，过长参数列表会导致代码的可读性和可维护性下降。当一个函数需要接受大量的输入参数时，这通常意味着这个函数承担了过多的责任。根据单一职责原则，一个函数应该只做一件事情，并且只接受必要的参数。过长的参数列表可能导致参数之间的耦合度增加，使得函数之间的依赖关系变得复杂。这会增加代码维护的难度，并且可能导致代码中的错误和缺

陷。

（3）代码来源

该段代码节选自我在实现机器学习中正则化方法避免高次多项式预测函数对训练数据过拟合的情况发生的代码。

（4）重构方法

1. 将参数封装成对象：如果多个函数有同样的几个参数，可以考虑将这些参数封装成一个对象，并将这个对象作为参数传递给函数。这样可以减少参数的数量，并提高代码的可读性和可维护性。
2. 使用构建器模式：对于具有大量参数的类，可以考虑使用构建器模式来创建对象。这种模式允许将对象的构建过程分解为一系列步骤，每个步骤只设置对象的一个或几个属性。这样可以避免在构造函数中使用过长的参数列表。
3. 使用依赖注入：通过依赖注入的方式，可以将函数的依赖项作为参数传递给它。这样可以减少函数自身的复杂性，并使得函数之间的依赖关系更加清晰。

（5）重构代码

```
theta = np.random.rand(x.shape[1]) # 此处 x.shape[1]为 11
def func_predict(x,theta):
    return np.dot(x,theta)
```

五．过长函数

（1）坏味道的代码

```
#include<stdio.h>
#include<math.h>
int time[2][7] = { {2,7,9,3,40,8,3},{4,8,5,6,4,5,6} };
int changetime[2][4] = { {2,3,1,3},{2,1,2,2} };
struct answer {
```

```
    int time;
    int from;
}result[2][5],time_exit;
int main()
{
    result[0][0] = { 9,1 };
    result[1][0] = { 12,2 };
    for(int i = 1;i < 5;i++)
    {
        if(result[0][i - 1].time > result[1][i - 1].time +
changetime[1][i - 1])
        {
            result[0][i].time = result[1][i - 1].time +
changetime[1][i - 1] + time[0][i + 1];
            result[0][i].from = 2;
        }else
        {
            result[0][i].time = result[0][i - 1].time + time[0][i +
1];
            result[0][i].from = 1;
        }
        if (result[0][i - 1].time + changetime[0][i - 1] >
result[1][i - 1].time)
        {
            result[1][i].time = result[1][i - 1].time + time[1][i +
1];
            result[1][i].from = 2;
        }
        else
        {
            result[1][i].time = result[0][i - 1].time +
changetime[0][i - 1] + time[1][i + 1];
            result[1][i].from = 1;
        }
    }
    if (result[0][4].time + time[0][6] < result[1][4].time +
time[1][6])
    {
        time_exit.time = result[0][4].time + time[0][6];
        time_exit.from = 1;
    }
    else
    {
        time_exit.time = result[1][4].time + time[1][6];
```

```
    time_exit.from = 2;
}
printf(" 最短时间为%d\n", time_exit.time);
printf("对应的排程方案:\n");
for (int i = 0; i < 5; i++)
{
    static int flag = 0;
    if (i == 0)
    {
        if (time_exit.from == 1)
        {
            printf("1<---");
            flag = 1;
            continue;
        }
        else
        {
            printf("2<---");
            flag = 2;
            continue;
        }
    }
    if (i == 4)
    {
        if(result[flag - 1][1].from == 1)
        {
            printf("1");
            continue;
        }
    else
    {
        printf("2");
        continue;
    }
    }
    if (result[flag - 1][5 - i].from == 1)
    {
        printf("1<---");
        flag = 1;
        continue;
    }
    else
    {
        printf("2<---");
```

```
            flag = 2;
            continue;
        }
    }
    return 0;
}
```

（2）什么坏味道

过长函数坏味道是指一个函数或方法中包含过多的代码行和逻辑判断，使得函数的阅读和理解变得困难。这样的函数往往具有复杂的控制流和难以追踪的状态变化，增加了代码维护的难度。

（3）代码来源

这段代码来自于我刚刚上机完的算法实验 2 上第一题作业排程问题的代码

（4）重构方法

1. 提炼函数：将过长函数中的一部分逻辑提炼出来，形成一个新的函数。这样可以使原函数更加简洁，同时也提高了代码的可读性和可维护性。
2. 分解函数：如果函数内部存在多个逻辑分支或条件判断，可以考虑将其分解为多个小函数，每个小函数只负责一个具体的逻辑分支或条件判断。
3. 移动函数：如果过长函数中的某些逻辑与其他函数或类更为紧密相关，可以考虑将这些逻辑移动到更合适的位置，以减少函数的长度和复杂度。

（5）重构后的代码

```
#include<stdio.h>
#include<math.h>
int shortestpath();
void print_shortestpathsolution();
int time[2][7] = { {2,7,9,3,40,8,3},{4,8,5,6,4,5,6} };
int changetime[2][4] = { {2,3,1,3},{2,1,2,2} };
```

```
struct answer {
   int time;
   int from;
}result[2][5],time_exit;
int main()
{
   int result_time = shortestpath();
   printf(" 最短时间为%d\n",result_time );
   print_shortestpathsolution();
   return 0;
}
int shortestpath()
{
   result[0][0] = { 9,1 };
   result[1][0] = { 12,2 };
   for (int i = 1; i < 5; i++)
   {
      if (result[0][i - 1].time > result[1][i - 1].time +
changetime[1][i - 1])
      {
         result[0][i].time = result[1][i - 1].time +
changetime[1][i - 1] + time[0][i + 1];
         result[0][i].from = 2;
      }
      else
      {
         result[0][i].time = result[0][i - 1].time + time[0][i +
1];
         result[0][i].from = 1;
      }
```

```
      if (result[0][i - 1].time + changetime[0][i - 1] > result[1][i - 1].time)
      {
         result[1][i].time = result[1][i - 1].time + time[1][i + 1];
         result[1][i].from = 2;
      }
      else
      {
         result[1][i].time = result[0][i - 1].time + changetime[0][i - 1] + time[1][i + 1];
         result[1][i].from = 1;
      }
   }
   if (result[0][4].time + time[0][6] < result[1][4].time + time[1][6])
   {
      time_exit.time = result[0][4].time + time[0][6];
      time_exit.from = 1;
   }
   else
   {
      time_exit.time = result[1][4].time + time[1][6];
      time_exit.from = 2;
   }
   return time_exit.time;
}
void print_shortestpathsolution()
{
```

```
printf("对应的排程方案:\n");
for (int i = 0; i < 5; i++)
{
   static int flag = 0;
   if (i == 0)
   {
      if (time_exit.from == 1)
      {
         printf("1<---");
         flag = 1;
         continue;
      }
      else
      {
         printf("2<---");
         flag = 2;
         continue;
      }
   }
   if (i == 4)
   {
      if (result[flag - 1][1].from == 1)
      {
         printf("1");
         continue;
      }
      else
      {
         printf("2");
         continue;
```

```
            }
        }
        if (result[flag - 1][5 - i].from == 1)
        {
            printf("1<---");
            flag = 1;
            continue;
        }
        else
        {
            printf("2<---");
            flag = 2;
            continue;
        }
    }
}
```

（六）发散式变化

（1）坏味道代码

```
public class Rectangle{
    private double length;
    private double width;
    public Rectangle(double length,double width){
        this.length = length;
        this.width = width;
    }
    public Rectangle(){
```

```
    }
    public double Getlength(){
        return this.length;
    }
    public double Getwidth(){
        return this.width;
    }
    public void Setlength(double length){
        this.length = length;
    }
    public void Setwidth(double width){
        this.width = width;
    }
    public double getarea(){
        return this.length * this.width;
    }
}
public class Square{
    private double side;
    public Square(double side){
        this.side = side;
    }
    public Square(){

    }
    public double Getside(){
        return this.side;
    }
    public void Setside(double side){
        this.side = side;
```

```
    }
    public double getarea(){
        return side * side;
    }
}
```

（2）什么坏味道

这段代码的坏味道是发散式变化，主要指的是一个类因为不同类型的原因经常发生变化。也就是说，这个类内部包含了多种不同的职责，使得每当其中任何一个职责发生变化时，都需要修改这个类。

（3）代码来源

这段代码来自于面向对象课程当中使用 Java 来实现相关图形面积计算时所写

（4）重构方法

1.提取类：将类中因为不同原因而变化的部分提取出来，形成新的类。这样，每个类只负责一个职责，减少了类之间的耦合度。
2.使用接口或抽象类：如果多个类因为相同的原因而发生变化，可以考虑使用接口或抽象类来定义这些变化的共同行为。这样，当这些行为发生变化时，只需要修改接口或抽象类，而不需要修改所有实现类。

（5）重构后的代码

```
public class Rectangle{
    private double length;
    private double width;
    public Rectangle(double length,double width){
        this.length = length;
```

```
        this.width = width;
    }
    public Rectangle(){
        super();
    }
    public double Getlength(){
        return this.length;
    }
    public double Getwidth(){
        return this.width;
    }
    public void Setlength(double length){
        this.length = length;
    }
    public void Setwidth(double width){
        this.width = width;
    }
    public double getarea(){
        return this.length * this.width;
    }
}
public class Square extends Rectangle{
    public Square(double side){
        super(side,side);
    }
    public Square(){
        super();
    }
    @Override
        public void Setlength(double length) {
```

```
            super.Setlength(length);
            super.Setwidth(length);
        }

        @Override
    public void Setwidth(double width) {
            super.Setlength(width);
            super.Setwidth(width);
        }
}
```

1.4 实验小结

1.通过本次实验，我深刻理解了重构在软件开发中的作用。通过对代码的重构，可以极大地提高代码在软件开发中的可读性、可维护性、性能和可扩展性。

2.在重构的过程当中，我熟悉了在代码中常见的六种坏味道，并且掌握了其对应的相关重构方法。