

Refactoring

Доц. д-р Ася Стоянова-
Дойчева



Съдържание

- Какво е Refactoring?
- Причини за refactoring
- Стъпките на refactoring
- Ситуации в кода изискващи refactoring
- Ситуации, в които не се прави refactoring



Какво е refactoring?

- Refactoring: Промени във вътрешната структура на ОО програма, за да я направи по-лесна за разбиране и лесна за модифициране без да се променя нейното външно поведение.

M. Fowler

- Или “Промяна на кода на ОО софтуер след като вече е написан”



Много прост пример за refactoring😊

- Обединение на повтарещ се фрагмент в условието

Това:

```
if (isSpecialDeal()) {  
    total = price * 0.95;  
    send()  
} else {  
    total = price * 0.98;  
    send()  
}
```

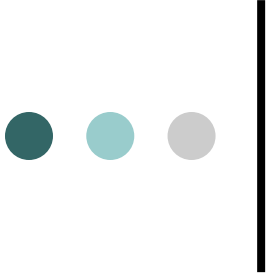
Става това:

```
if (isSpecialDeal()) {  
    total = price * 0.95;  
} else {  
    total = price * 0.98;  
}  
send();
```



Причини за refactoring

- подобрява архитектурата на приложението
- прави кода по-лесен за четене и разбиране
- помага при откриването на грешки и в подобряване на качеството на софтуера
- помага за по-бързо програмиране



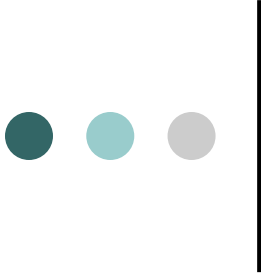
Кога трябва да правим refactoring?

- Когато добавяте функционалност
- Когато трябва да откриете грешка
- Когато искате да прегледате кода



Стъпки на refactoring

- Разучаване на проблема – възможно ли е да се приложи refactoring?
- Разучаване на кода на наследената система и извличане на архитектурата.
- Откриване на местата в архитектурата и съответно в кода изискващи refactoring.
- Създаване на тестови случаи и документиране на резултатите от тях преди прилагане на refactoring.
- Прилагане методите на refactoring – тестването съпровожда тази стъпка.
- Прилагане на пълни тестове след приключване на refactoring.



Къде да приложим refactoring?

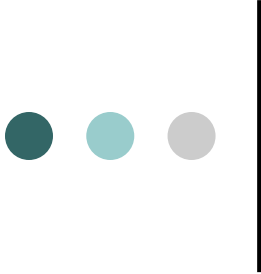
- Как да определим местата в кода, където да приложим refactoring?
 - Martin Fowler предлага да търсим “bad smells” в кода. Той предлага примери за такива лоши места в кода и дава решение за тях.
 - Kent Back който е основоположника на refactoring предлага да използваме “refactoring patterns”, описани в книгата на Fowler.



Ситуации в кода изискващи refactoring

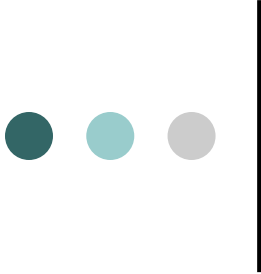
○ Повтарящ се код

- Ако се вижда една и съща структура код на повече от едно място, може да сте сигурни, че програмата ще стане по-добра, ако се намери начин да се обедини кода.



Ситуации в кода изискващи refactoring

- Дълги методи.
 - Ако имаме метод с много параметри и временни променливи. Ако в този случай се приложи *Extract Method* то много от тези параметри и променливи ще бъдат прехвърлени на новия метод.



Ситуации в кода изискващи refactoring

- **Голям клас**
- Когато един клас се опитва да прави прекалено много неща, той често има прекалено много атрибути. Когато един клас има прекалено много атрибути, дублиращият се код не остава по - назад. Може да се използва *Extract Class*, *Extract Subclass*. и др.



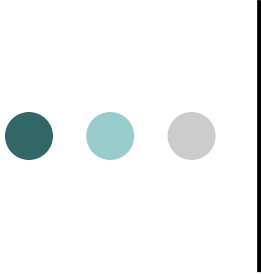
Ситуации в кода изискващи refactoring

- Дълъг списък от параметри.
 - Дългите списъци от параметри затрудняват разбирането, защото са непоследователни, несъобразени и трудни за използване, и защото когато се наложи внасянето на повече данни тези списъци се променят изцяло



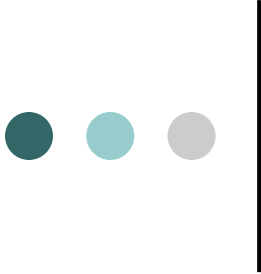
Ситуации в кода изискващи refactoring

- Switch Statement
 - Един от най-очевидните симптоми на обектно-ориентирания код е неговата нужда от switch (или case) statements. Проблемът с switch statements по същество е като дублиращия се код. Често един и същи switch statement е разхвърлян на различни места в програмата. Ако решите да въведете нова клауза в switch-а ще трябва да намерите всички срещания на този switch в програмата и да смените и тях. Обектно-ориентираната идея за полиморфизъм показва елегантен начин за справяне с този проблем.



Ситуации в кода изискващи refactoring

- Пасивни (бездействиящи) класове
 - Всеки клас, който се създава струва пари за да се поддържа. Клас, който не прави достатъчно за да се изплаща, трябва да бъде елиминиран. Много често това може да бъде клас, който е бил ефективен, но намален след приложен рефакторинг. Или може да бъде клас, който е бил създаден за промените, които са били планирани, но неосъществени. И при единия, и при другия случай, класът се отстранява от кода.



Ситуации в кода изискващи refactoring

- Алтернативни класове с различни интерфейси
 - Използва се *Rename Method* за всички методи, които правят едно и също нещо но имат различни сигнатури за това което правят.



Ситуации в кода изискващи refactoring

- Коментари
- Изненадващо е колко често гледаме към гъсто коментирания код и се уверяваме че коментарите са там, защото кодът е лош.

Коментарите ни показват всичките тези ситуации, които са лоши в кода. Първата работа е да се оправят тези места чрез рефакторинг. Когато сме привършили, често ще забелязваме че коментарите са излишни.



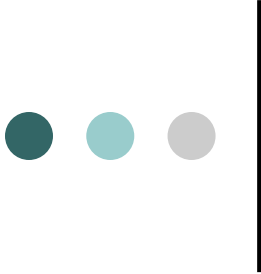
Refactoring patterns

- В книгата на М. Fowler има дефинирани 72 refactoring patterns. Някой от тях са:
 - Extract Method
 - Extract Subclass
 - Replace Temp with Query
 - Move Method
 - Replace Conditional with Polymorphism



Проблеми при refactoring

- Бази Данни – бизнес приложенията често са тясно свързани с бази от данни. Кода е лесен за промяна но логиката на БД не!
- Промяна на интерфейсите
- Промяна в архитектурата на приложението – “софтуерните инженери трябва да имат кураж да направят това 😊 ”



Ситуации в които не се прави refactoring

- Когато кода не работи
- Когато имаме точно определено време за рефакторинг т.е. имаме фиксиран срок. – “Един рефакторинг предизвиква друг....”



Пример

```
1      class Account {
2          float principal;
3          float rate;
4          int daysActive;
5          int accountType;
6
7          public static final int  STANDARD = 0;
8          public static final int  BUDGET = 1;
9          public static final int  PREMIUM = 2;
10         public static final int  PREMIUM_PLUS = 3;
11     }
12
13     float calculateFee(Account accounts[]) {
14         float totalFee = 0;
15         Account account;
16         for (int i = 0; i < accounts.length; i++) {
17             account = accounts[i];
18             if ( account.accountType == Account.PREMIUM ||
19                 account.accountType == Account.PREMIUM_PLUS ) {
20                 totalFee += .0125 * ( account.principal
21                                     * Math.exp( account.rate * (account.daysActive/365.25) )
22                                     - account.principal );
23             }
24         }
25         return totalFee;
26     }
```

Прилагане на Extract Method

```
1. class Account {
2.     float principal;
3.     float rate;
4.     int daysActive;
5.     int accountType;
6.
7.     public static final int STANDARD = 0;
8.     public static final int BUDGET = 1;
9.     public static final int PREMIUM = 2;
10.    public static final int PREMIUM_PLUS = 3;
11.
12.    float interestEarned() {
13.        return ( principal * (float) Math.exp( rate * (daysActive / 365.25) ) )
14.            - principal;
15.    }
16.}
17.
18.float calculateFee(Account accounts[]) {
19.    float totalFee = 0;
20.    Account account;
21.    for (int i = 0; i < accounts.length; i++) {
22.        account = accounts[i];
23.        if ( account.accountType == Account.PREMIUM ||
24.            account.accountType == Account.PREMIUM_PLUS )
25.            totalFee += .0125 * account.interestEarned();
26.    }
27.    return totalFee;
28.}
```



Replace Magic Number with Symbolic Constant

```
1. class Account {
2.     float principal;
3.     float rate;
4.     int daysActive;
5.     int accountType;
6.
7.     public static final int STANDARD = 0;
8.     public static final int BUDGET = 1;
9.     public static final int PREMIUM = 2;
10.    public static final int PREMIUM_PLUS = 3;
11.
12.    float interestEarned() {
13.        return ( principal * (float) Math.exp( rate * (daysActive / 365.25 ) ) )
14.            - principal;
15.    }
16.}
17.
18.float calculateFee(Account accounts[]) {
19.    float totalFee = 0;
20.    Account account;
21.    for (int i = 0; i < accounts.length; i++) {
22.        account = accounts[i];
23.        if ( account.accountType == Account.PREMIUM ||
24.            account.accountType == Account.PREMIUM_PLUS ) {
25.            totalFee += BROKER_FEE_PERCENT * account.interestEarned();
26.        }
27.    }
28.    return totalFee;
29.}
30.
31.static final double BROKER_FEE_PERCENT = 0.0125;
```



Decompose Conditional

```
1. class Account {
2.     float principal;
3.     float rate;
4.     int daysActive;
5.     int accountType;
6.
7.     public static final int STANDARD = 0;
8.     public static final int BUDGET = 1;
9.     public static final int PREMIUM = 2;
10.    public static final int PREMIUM_PLUS = 3;
11.
12.    float interestEarned() {
13.        return ( principal * (float) Math.exp( rate * (daysActive / 365.25 ) ) )
14.            - principal;
15.    }
16.
17.    public boolean isPremium() {
18.        if (accountType == Account.PREMIUM || accountType == Account.PREMIUM_PLUS)
19.            return true;
20.        else
21.            return false;
22.    }
23. }
24.
25. float calculateFee(Account accounts[]) {
26.     float totalFee = 0;
27.     Account account;
28.     for (int i = 0; i < accounts.length; i++) {
29.         account = accounts[i];
30.         if ( account.isPremium() )
31.             totalFee += BROKER_FEE_PERCENT * account.interestEarned();
32.     }
33.     return totalFee;
34. }
35.
36. static final double BROKER_FEE_PERCENT = 0.0125;
```



Introduce Explaining Variable

```
1. class Account {
2.     float principal;
3.     float rate;
4.     int daysActive;
5.     int accountType;
6.
7.     public static final int STANDARD = 0;
8.     public static final int BUDGET = 1;
9.     public static final int PREMIUM = 2;
10.    public static final int PREMIUM_PLUS = 3;
11.
12.    float interestEarned() {
13.        float years = daysActive / (float) 365.25;
14.        float compoundInterest = principal * (float) Math.exp( rate * years );
15.        return ( compoundInterest - principal );
16.    }
17.
18.    public boolean isPremium() {
19.        if (accountType == Account.PREMIUM || accountType == Account.PREMIUM_PLUS)
20.            return true;
21.        else
22.            return false;
23.    }
24. }
25.
26. float calculateFee(Account accounts[]) {
27.     float totalFee = 0;
28.     Account account;
29.     for (int i = 0; i < accounts.length; i++) {
30.         account = accounts[i];
31.         if ( account.isPremium() ) {
32.             totalFee += BROKER_FEE_PERCENT * account.interestEarned();
33.         }
34.     }
35.     return totalFee;
36. }
37.
38. static final double BROKER_FEE_PERCENT = 0.0125;
```