

A photograph of three people in a meeting. On the left, a woman with glasses is looking at a whiteboard. In the center, a woman with short blonde hair is looking at the whiteboard. On the right, a man in a striped shirt is looking at the whiteboard. The whiteboard is covered with many sticky notes and has a box labeled 'Key Objectives' at the top. The image has a blue and purple color overlay.

Refactorizare

Alecsandru Florin Soare

ThoughtWorks®

A woman with dark hair is standing in a dimly lit room, looking upwards with a surprised expression. The room is decorated with numerous strings of colorful, glowing Christmas lights (red, green, blue, and white) that create a bokeh effect in the background. The lighting is warm and festive.

REFACTORING LEGACY CODE

CE ESTE ȘI LA CE E BUNĂ REFACTORIZAREA?

Refactorizarea codului sau „code refactoring” este procesul de modificare a unei secvențe de program fără a-i schimba funcționalitatea externă:

- Uneori prin refactorizare se înțelege simpla reorganizare a codului însă...
- Deseori prin refactorizare funcționalitatea internă e schimbată prin optimizarea codului existent sau prin folosirea unor algoritmi mai buni;
- Martin Fowler definește refactorizarea ca fiind procesul de schimbare a unui sistem software în urma căruia:
 - Comportamentul extern al codului nu este alterat;
 - Structura internă a codului este îmbunătățită;

CUM APARE NEVOIA DE REFACTORIZARE?

- În orice sistem software, inițial codul acestuia arată bine;
- În timp apar tot felul de cerințe:
 - de modificare (parțială sau totală) a unor funcționalități existente;
 - de adăugare de funcționalități noi;

în urma cărora codul se „atrofiază”, devenind greu de urmărit, buggy sau pur și simplu haotic (devine incoerent la nivel de design, de code style, etc.)

CÂND REFACTORIZEZ?

- Ori de câte ori trebuie adăugată o nouă funcționalitate iar regiunea de cod care urmează a fi modificată se dovedește a fi ori prea „fragilă”, ori foarte greu de citit și/sau urmărit. De exemplu: Atunci când o modificare repetată într-o clasă implică o serie de modificări mici într-un alt grup de clase;
- Ori de câte ori trebuie fixat un defect mai complex, înainte de fixarea propriu-zisă;
- În urma unui proces de code review;
- În urma unui proces mai îndelungat de „code development”. În totdeauna procesul de dezvoltare va trebui alternat cu cel de refactorizare;
- Atunci când codul începe să prindă „miros”;

A background image showing two individuals in full-body protective hazmat suits. They are holding their noses with their hands, suggesting a strong, unpleasant odor. The image is overlaid with a semi-transparent gradient that transitions from a reddish-pink on the left to a blueish-purple on the right.

*Bad Code Smells are Symptoms
of poor design or implementation choices.*

Martin Fowler

CODE „SMELLS” – LA NIVEL DE APLICATIE

- Duplicated code: Secțiuni de cod (clase/metode/etc.) sunt duplicate;
- Contrived complexity: Datorită design-ului existent ești forțat să folosești patternuri complexe (deși s-ar putea mult mai simplu);
- Shotgun surgery antipattern – e nevoie de o modificare în cascadă atunci când faci o modificare în cod. Exemplu: ca să modifichi o clasă, ești forțat să mai modifichi alte N clase;
- Uncontrolled side effects: erori ce apar la runtime și care nu sunt prinse de unit-teste;
- Speculative Generality – o ierarhie (de clase/interfețe) proiectată pentru un viitor incert și nefolosite încă;

CODE „SMELLS” – LA NIVEL DE CLASE

- Clasele sunt foarte lungi (God object anti-pattern);
- Feature Envy – o clasă folosește excesiv funcționalități aflate într-o altă clasă;
- Lazy class – clase care nu fac mai nimic;
- Inappropriate Intimacy – clase care partajează zone private și unde încapsularea poate deveni practic imposibilă;
- Excessive use of literals;
- Refused bequest: prin suprascrierea unor metode se încalcă contractul din clasa de bază;
- Downcasting: type cast folosit incorect și care are drept efect distrugerea abstractizării;
- Cyclomatic complexity: prea multe ramificații și bucle într-o singură metodă;
- Data clump: un același set de variabile sunt folosite grupat în diferite părți ale clasei;
- Message chains (chain pattern), Middle man (wrapper/facade pattern)

CODE „SMELLS” – LA NIVEL DE METODE

- Too many parameters: Lista de argumente/parametrii ai metodelor e foarte mare;
- Long method: metode mult prea mari
- Blocuri switch sau de if ... else if ... în locul polimorfismului;
- Excessively long/short identifiers
- Excessive return of data: o metodă care întoarce mai mult decât este nevoie;
- Excessively long line of code (or God Line): O linie excesiv de lungă care e foarte dificil de citit, înțeles și de depanat. Exemplu:

```
new XYZ(s).doSomething(buildParam1(x), buildParam2(x), buildParam3(x), a  
+ Math.sin(x)*Math.tan(x*y + z)).doAnythingElse().build().sendRequest();
```

LA CE E BUNĂ REFACTORIZAREA?

- Refactorizarea vizează în principal îmbunătățirea structurală a unui codul existent – cod care a fost testat și validat anterior - având ca obiective finale:
- Reducerea complexității prin:
 - eliminarea codului duplicat sau „umflat” artificial;
 - agregarea funcționalităților similare într-un singur loc;
- Creșterea lizibilității și implicit a mentenabilității codului;
- Modificarea internă a acestuia în vederea extinderii codului cu noi opțiuni (capacitatea codului de a fi extensibil);
- Deseori ajută la descoperirea unor defecte „ascunse”;

CE **NU** ESTE REFACTORIZAREA

- Refactorizarea codului nu este tot una cu procesul de bug-fixing
- În cazul defectelor majore și/sau complicate (defecte provenite din etapele de specificare sau design), procesul de fixare a acestora poate implica una sau mai multe etape de refactorizare a codului existent înainte de fixarea propriu-zisă a acestora;
- Totdeauna trebuie avut în vedere faptul că:
 - refactorizarea se aplică pe codul care funcționează corect;
 - fixarea se aplică pe codul care nu funcționează corect.

REFACTORIZAREA CODULUI != REWRITE/REWORK

Rewrite - înseamnă reimplementarea unei largi porțiuni de cod sau a întregii aplicații fără utilizarea codului sursă inițial. Numai un caz particular de rewrite, cel de re-write parțial, poate fi considerat ca fiind produsul unui proces de refactorizare și doar atunci când funcționalitatea externă rămâne neschimbată;

Rework - Decizia de „rework from scratch” sau „total rewriting” se ia în situații excepționale, atunci când:

- se dorește extinderea și/sau optimizarea unui sistem software și;
- codul existent nu poate oferi acest lucru (chiar dacă s-ar încerca refactorizarea lui) datorită deciziilor de design inițiale (vezi arhitecturi software și design patterns) sau al tehnologiilor folosite (limbaje de programare, sisteme de operare, biblioteci, etc.);

CÂND **NU** REFACTORIZEZ?

- Atunci când codul este ori prea vechi, ori prea fragil ori prea complex pentru acest proces. În aceste condiții se recomandă un proces de rewrite sau rework;
- Atunci când ești în preajma unui deadline. Refactorizarea în sine este un proces care necesită în primul rând timp și care implică o re-parcursare (la o scală mai mică) a tuturor etapelor de dezvoltare a unui sistem software (de la etapa de design și până la cea de implementare a codului de urmează a fi refactorizat);

CE RISC ATUNCI CÂND **NU** REFACTORIZEZ?

- O proiectare și o implementare defectuoasă necesită la rândul său mai mult cod atunci când sunt adăugate funcționalități noi;
- Codul duplicat este o realitate în industrie. Deși se estimează că acesta este cuprins între 8 – 10% din codul din producție acesta ajunge să fie în realitate mult mai mare pentru anumite instrumente software;

REZULTATELE REFACTORIZĂRII CODULUI

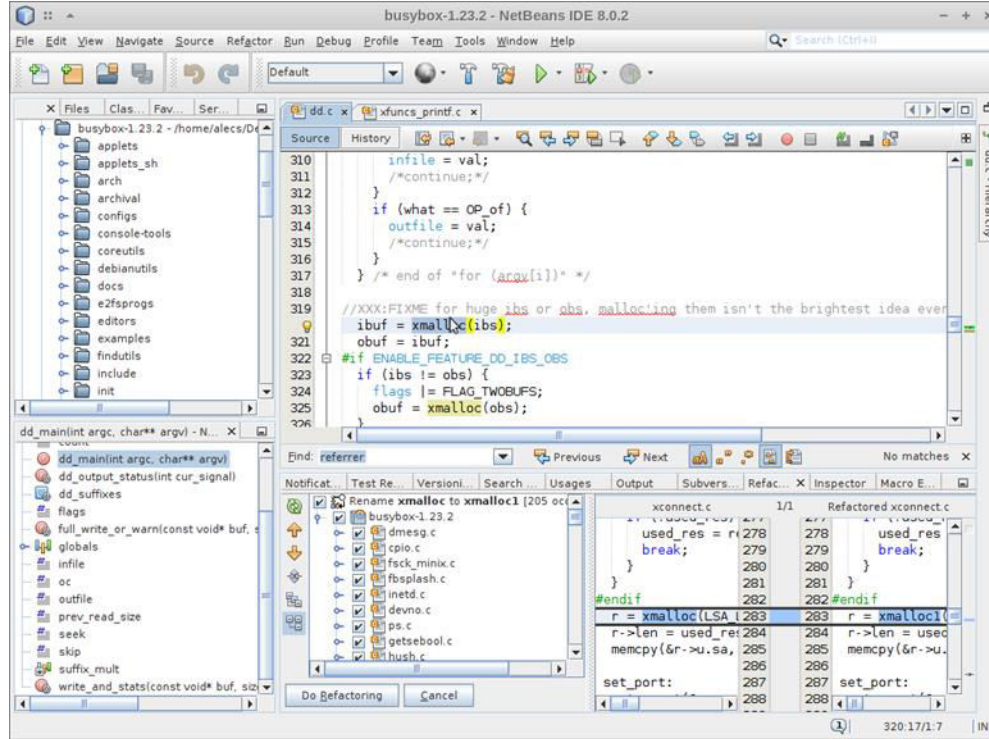
- Refactorizarea poate produce un cod diferit de la caz la caz, în funcție de obiectivul urmărit. De exemplu dacă obiectivul urmărit este:
- Reducerea complexității => poate produce mai multe componente simple;
- Creșterea lizibilității și implicit a mentenabilității codului => o serie de componente simple pot fi grupate în expresii/funcții/clase mai complexe, dar mai elegante, care pot fi utilizate mai ușor în diferite părți ale aplicației;
- Modificarea internă a acestuia în vederea extinderii codului cu noi opțiuni => apar noi nivele de abstractizate situate deasupra nivelelor deja existente (de tipul interfețelor sau claselor de bază);
- Prin urmare, înainte de a realiza o refactorizare va trebui să îmi aleg modalitatea/tehnica în care urmează să refactorizez codul.

TEHNICI DE REFACTORIZARE A CODULUI

Tehnici pentru îmbunătățirea denumirii și localizării codului:

- Mutarea definiției unei variabile, constante, funcție, etc. într-un fișier (sau într-o clasă) care ilustrează mai bine apartenența funcțională de aceasta;
- Redenumirea unei variabile, constante, funcție, etc. într-un fișier (sau într-o clasă) care să reflecte mai bine scopul/utilitatea acesteia;
- Pull Up/Push Down – cazuri particulare de mutare folosite în OOP pentru a evidenția deplasarea unor membrii în cadrul ierarhiei de clase (mutare într-o super clasă respectiv mutare într-o sub clasă).

NETBEANS: RENAME METHOD



ANDROID STUDIO: PULL UP

(GIF ANIMAT)

```
@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.my_activity);

    Button button = (Button) findViewById(R.id.my_button);
    button.setText("Click Me!");

    TextView textview = (TextView) findViewById(R.id.my_textview);
    textview.setText("some text");
}

@Override
protected void onStart() {
    super.onStart();
}

@Override
protected void onResume() {
    super.onResume();
    start();
}

@Override
protected void onPause() {
    super.onPause();
    stop();
}

@Override
protected void onStop() {
```

TEHNICI DE REFACTORIZARE A CODULUI

Tehnici pentru „spargerea” codului în secțiuni logice distincte:

- Componentizarea sau spargerea codului în unități semantice reutilizabile reprezentate de interfețe care sunt mai clare, mai bine definite și mai simplu de utilizat;
- **Extract class** – mutarea unei părți dintr-o clasă într-o clasă nouă. Nevoia apare atunci când există cod duplicat și se dorește a se crea o clasă de bază sau atunci când o clasă este prea mare și ea abstractizează funcționalități diferite (în acest caz putem crea clase de sine stătătoare sau inline);
- Opusul operației de mai sus se numește **collapse hierarchy** și se folosește când:
 - avem o ierarhie de clase cu subclase aproape goale;
 - avem un abuz de clase inline care nu fac mai nimic;
- **Extract field/constant/method** – extragerea unei valori (sau a unei secțiuni de cod) care se repetă într-o nouă variabilă sau funcție. Se elimină astfel codul duplicat.

ANDROID STUDIO: EXTRACT CONSTANT

(GIF ANIMAT)



```
package com.example.test;

import ...

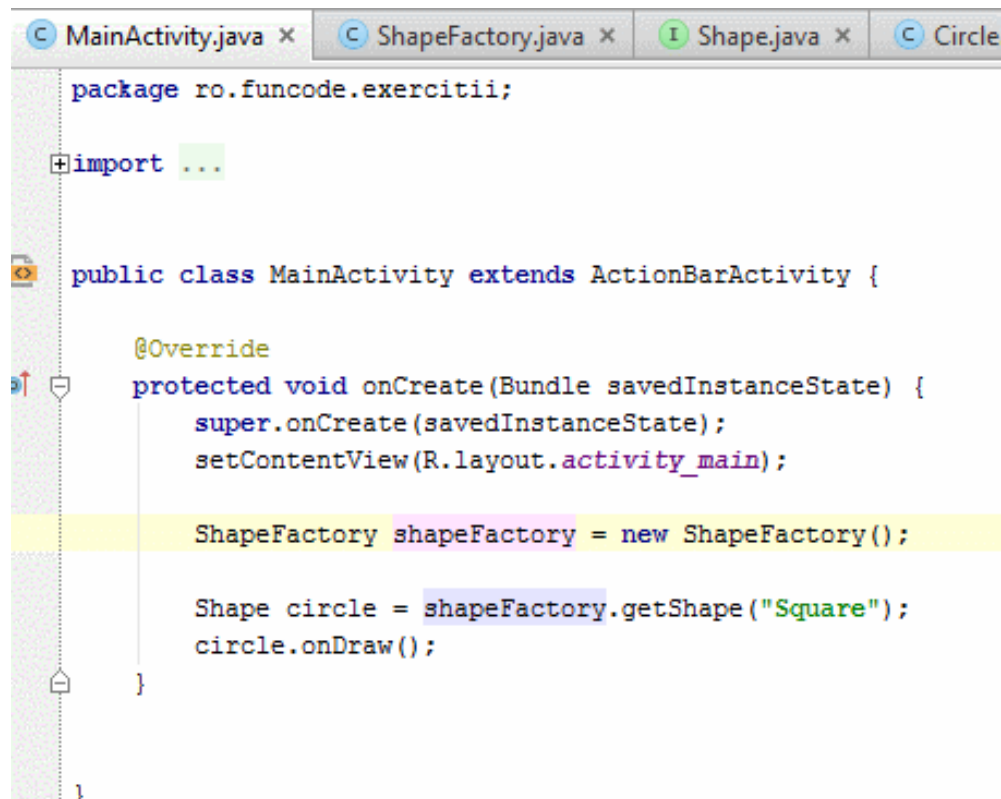
public class ActivityB extends Activity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_);

        String name = "Test";
        if (name.equals("Test")) {
            Toast.makeText(this, "Equal", Toast.LENGTH_SHORT).show();
        }
    }
}
```

ANDROID STUDIO: EXTRACT FIELD

(GIF ANIMAT)



```
package ro.funcode.exercitii;

import ...

public class MainActivity extends ActionBarActivity {

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        ShapeFactory shapeFactory = new ShapeFactory();

        Shape circle = shapeFactory.getShape("Square");
        circle.onDraw();
    }
}
```

ANDROID STUDIO: EXTRACT INTERFACE

(GIF ANIMAT)

```
package com.donnfelker.solid;

import java.util.ArrayList;
import java.util.List;

public class DataLayer {

    public int insert(Expense expense) {
        // go to db and insert
        return 1; // return the value of the id
    }

    public Expense find(int id) {
        Expense expense = null; // find an expense in the db/etc
        return expense;
    }

    public List<Expense> findAll() {
        return new ArrayList<Expense>(); // find all expenses in the db/etc
    }
}
```

RUBYMINE: MOVE METHODS -> NEW CLASS

(GIF ANIMAT)

```
extract_service.rb
1  class User
2    after_create :finalize_user_creation
3
4    def finalize_user_creation
5      tweet_new_user_joined_the_app
6      accept_invitation_and_welcome_user
7    end
8
9    def tweet_new_user_joined_the_app
10     # Twitter code here
11   end
12
13   def accept_invitation_and_welcome_user
14     accept_user_invitation
15     send_welcome_email
16     clear_user_invitation_token
17   end
18
19   def accept_user_invitation
20     self.accept_invitation = true
21     self.invitation_accepted_at = Time.zone.now
22   end
23
24   def send_welcome_email
25     UserMailer.welcome_user(self).deliver_later
26   end
27
28   def clear_user_invitation_token
29     self.invitation_token = nil
30     save
31   end
32 end
```

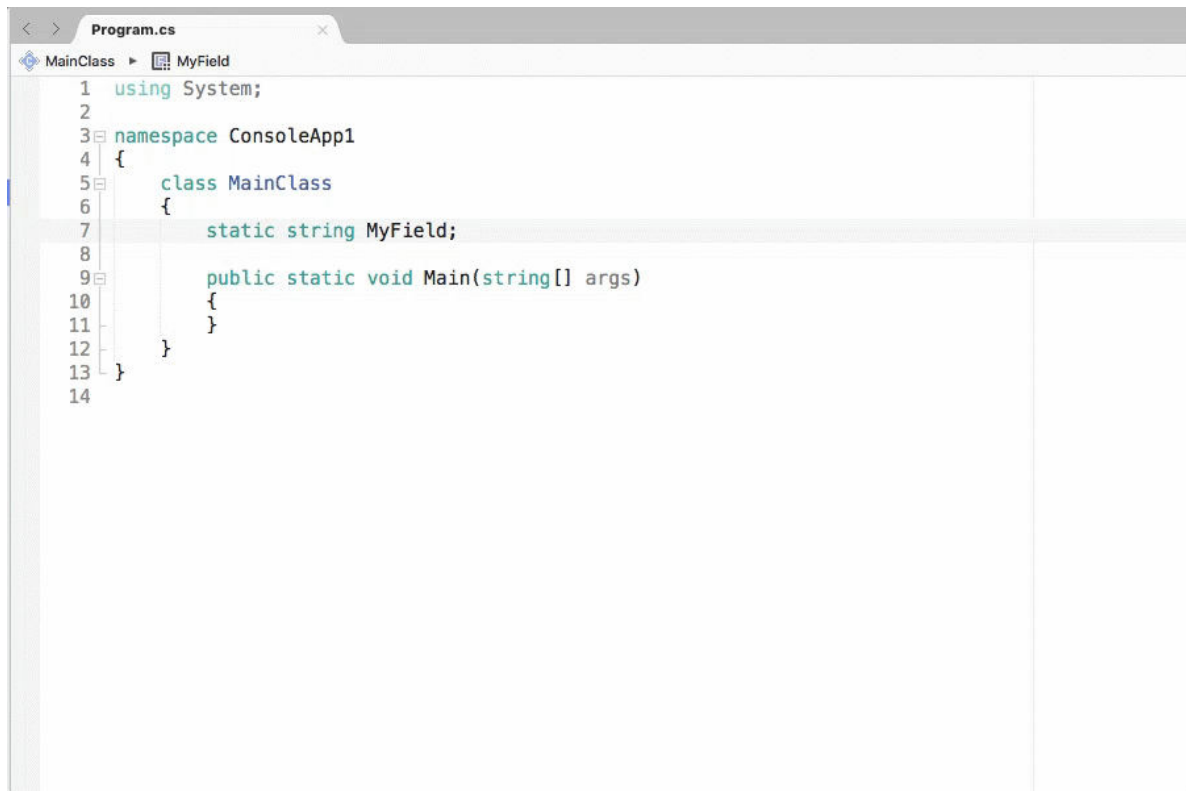
TEHNICI DE REFACTORIZARE A CODULUI

Tehnici care permit creșterea gradului de abstractizare a codului:

- **Encapsulate Field** – așa cum îi spune și numele permite încapsularea unui câmp și forțează utilizatorul să folosească accesorii de tip getter și setter în locul accesării directe;
- Gruparea unor parametrii care se repetă într-o clasă. De ex. parametrii an, luna, zi, ora, minut, secundă într-o instanță de tip Date;
- **Generalize type** – permite, în anumite condiții schimbarea tipului de date al unui câmp cu un tip de date cu un grad de generalizare mai ridicat (de exemplu: de la List la Collection);
 - Transformarea blocurilor condiționale în metode (aplicarea polimorfismului);
 - Transformarea constructorilor în Factory sau Builder methods (vezi design patterns);
 - Etc.

XAMARIN: ENCAPSULATE FIELD

(GIF ANIMAT)



```
1 using System;
2
3 namespace ConsoleApp1
4 {
5     class MainClass
6     {
7         static string MyField;
8
9         public static void Main(string[] args)
10        {
11        }
12    }
13 }
14
```

INTELLIJ: REPLACE CONSTRUCTOR WITH FACTORY METHOD

Înainte:

```
// File Class.java
public class Class {
    public Class(String s) { ... }
}

// File AnotherClass.java
public class AnotherClass {
    public void method() {
        Class aClass = new
        Class("string");
    }
}
```

După:

```
// File Class.java
public class Class {
    public Class(String s) { ... }
    public static createClass(String s) { return
    new Class(s); }
}

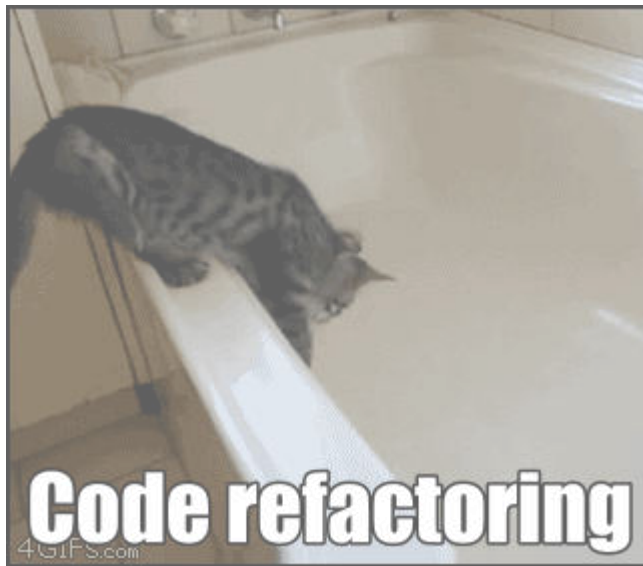
// File AnotherClass.java
public class AnotherClass {
    public void method() {
        Class aClass =
        Class.createClass("string");
    }
}
```



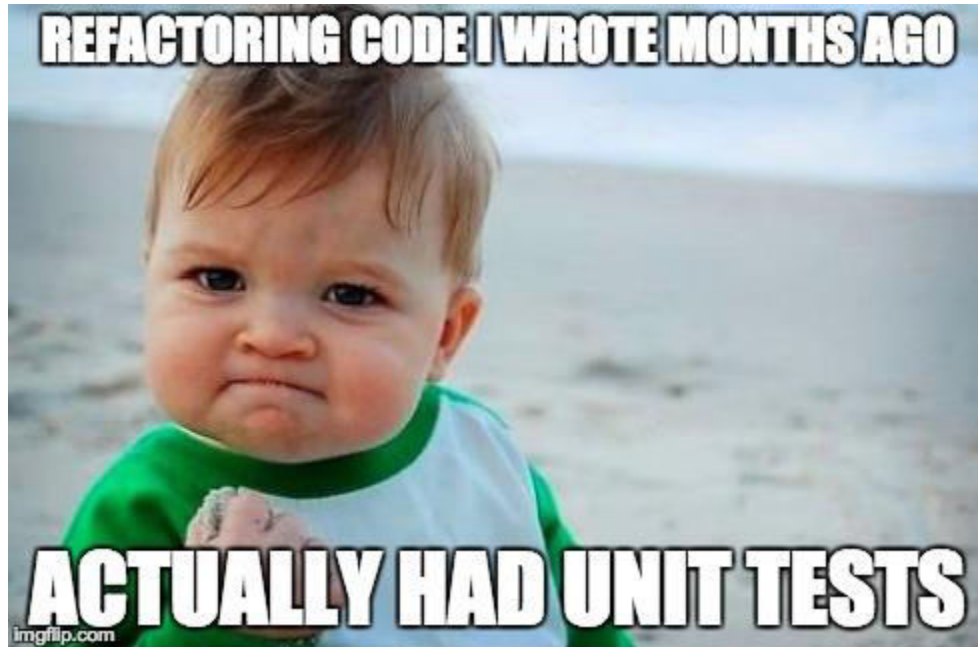
BUNE PRACTICI PENTRU REFACTORIZARE

- Restaurează codul inițial atunci când refactorizarea eșuează (folosind un version control system)
- Creează-ți un scenariu de testare (sau mai bine, o întreagă suită de teste) înainte de a realiza prima operație de refactorizare
- Refactorizează în pași cât mai mici
- Testează modificările după fiecare refactorizare
- Refactorizează codul automat (folosind un IDE) și nu refactoriza manual decât în situații excepționale
- Nu combina în același pas refactorizarea cu bug-fixing-ul
- Nu combina în același pas refactorizarea cu adăugarea/extinderea funcționalității codului

RESTAUREAZĂ CODUL INIȚIAL ATUNCI CÂND
REFACTORIZAREA EȘUEAZĂ (FOLOSIND UN VCS)



CREEAZĂ-ȚI O SUITĂ DE TESTE ÎNAINTE DE A
REALIZA PRIMA OPERAȚIE DE REFACTORIZARE



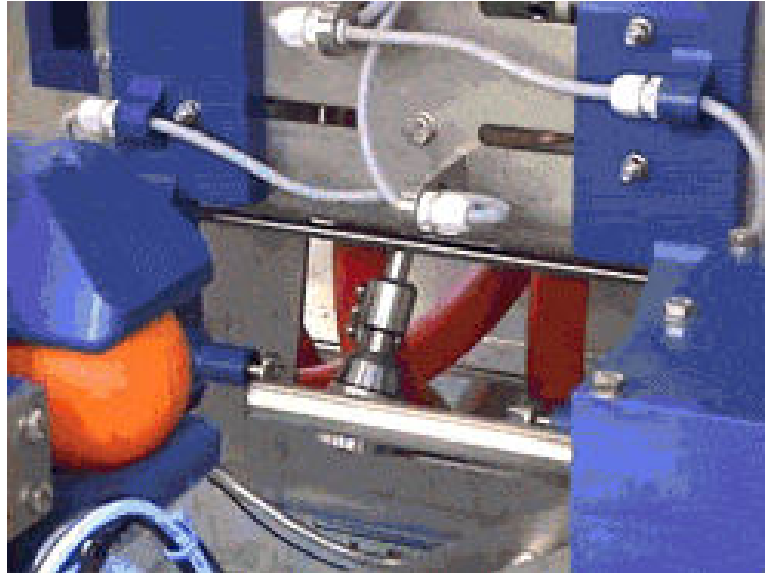
REFACTORIZEAZĂ ÎN PAȘI CÂT MAI MICI



TESTEAZĂ MODIFICĂRILE DUPĂ FIECARE REFACTORIZARE



REFACTORIZEAZĂ CODUL AUTOMAT (FOLOSIND UN IDE) ȘI NU REFACTORIZA MANUAL DECÂT ÎN SITUAȚII EXCEPȚIONALE



NU COMBINA REFACTORIZAREA CU BUG-FIXING-UL ȘI/SAU CU EXTINDEREA FUNCȚIONALITĂȚII



*Any fool can write code that a computer can understand.
Good programmers write code that human can understand.*

Martin Fowler

<https://www.refactoring.com/index.html>