



# Clean Code\*

\* sau de ce e mai important felul în care scriem cod decât ceea ce scriem

Catalin Boja, Bogdan Iancu, Alin Zamfiroiu

# Despre ce vom discuta

- De ce clean code?
- Principii
- Convenții de nume
- Clean Code în practică
- Scurt dicționar
- Instrumente
- Bonus



# De ce Clean Code?

"Any fool can write code that a computer can understand. Good programmers write code that humans can understand."

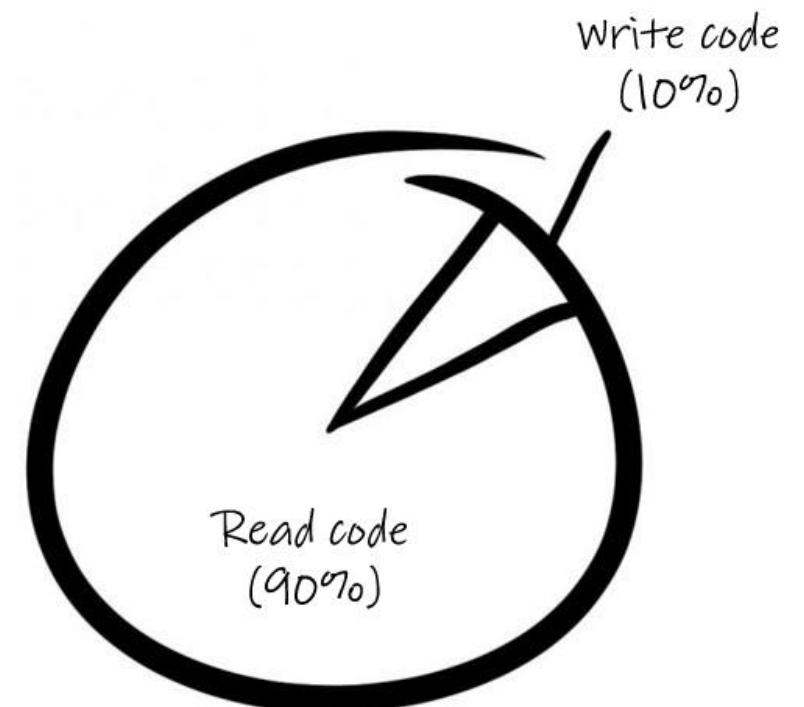


Martin Fowler

# De ce Clean Code?

- Programarea nu constă în a spune computerului ce să facă
- Programarea constă în a spune altui om ce vrem să facă un computer

Things programmers do at work

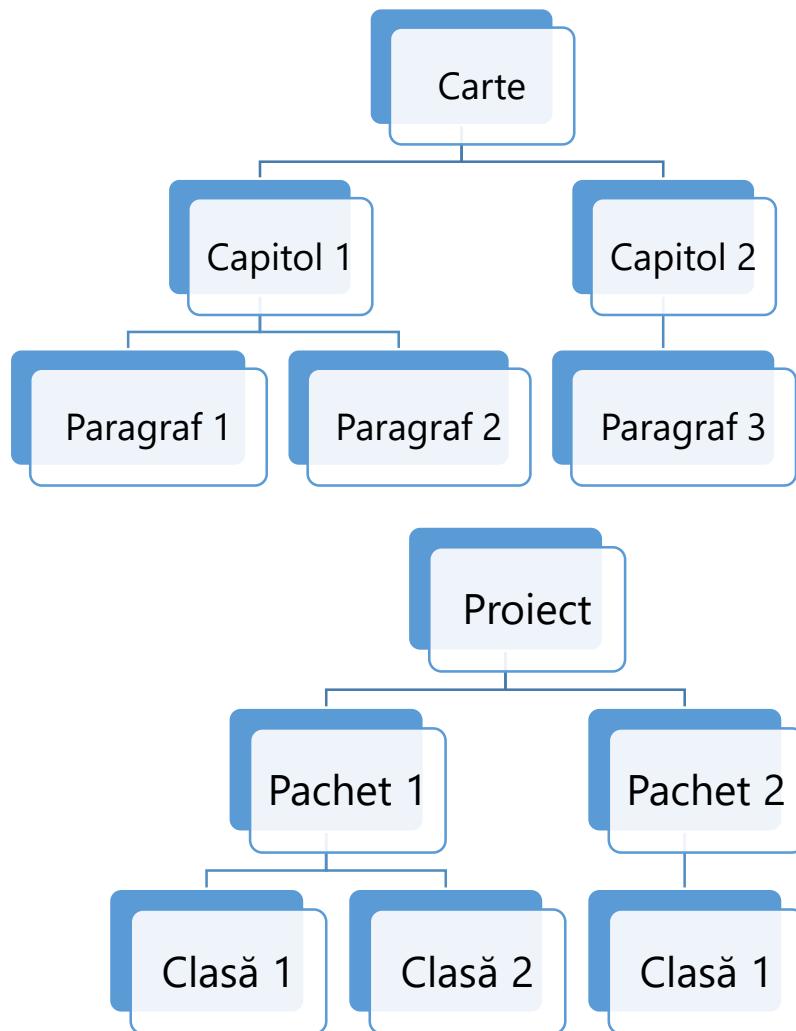


# De ce Clean Code?

- Din păcate câteodată acel „alt om” suntem chiar noi
- Nu avem timp să fim lenesi
- Altfel putem ajunge un substantiv
- Până la urmă suntem autori

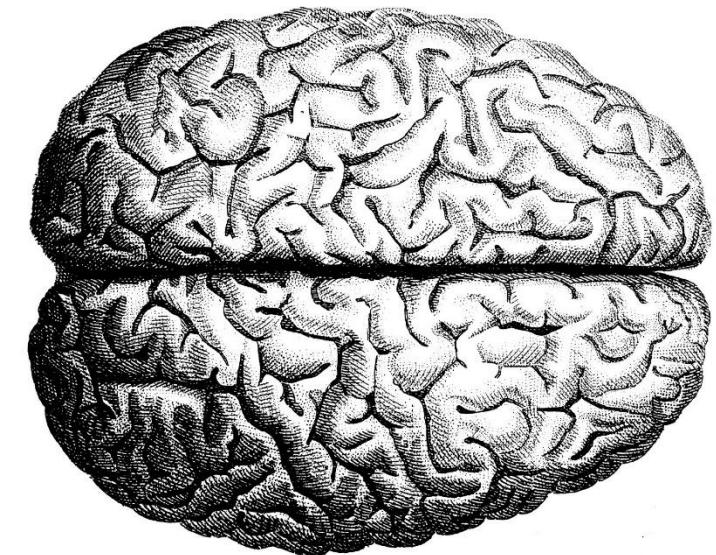


# Programator = Scriitor



# Încă ceva

- Când citim cod creierul nostru joacă rol de compilator
- Conform studiilor oamenii pot reține simultan doar 7 elemente ( $\pm 2$ ) în memorie
- Rubber Duck Programming (Debugging)



# Ce înseamnă Clean Code

**EASY**

&

**Nice**

**Clean Code**



# Ce înseamnă Clean Code

- Codul trebuie sa fie ușor de citit
  - Codul trebuie sa fie ușor de înțeles
  - Codul trebuie să fie ușor de modificat
- ... de către oricine

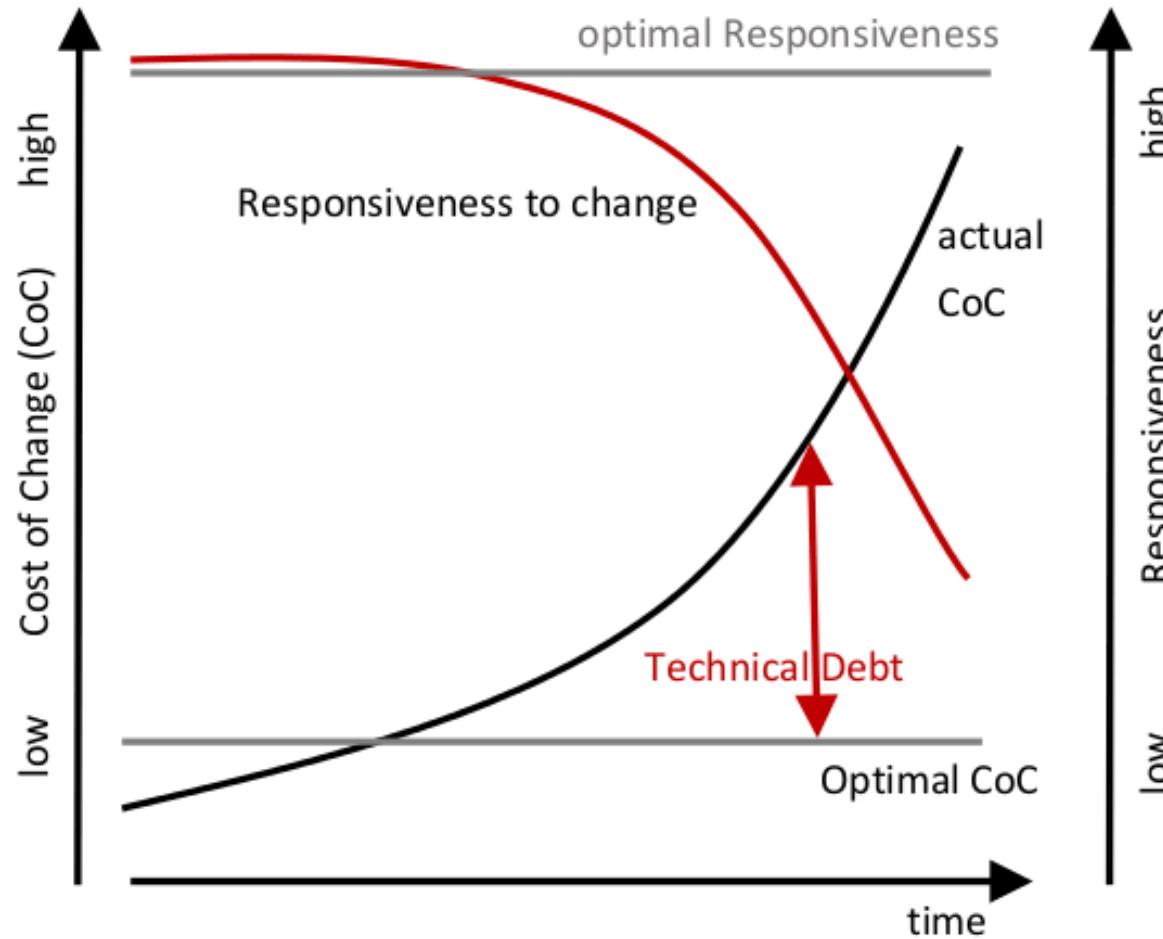
```
$super-dark-beige: #383229;  
$dark-beige: #534a3d;  
$medium-dark-beige: #847968;  
$medium-beige: #ada290;  
$medium-light-beige: #e6e4e1;  
$light-beige: #f6f5f2;  
$super-light-beige: #f8f8f8;  
$medium-light-grey: #c4bfbe;  
$light-grey: #d5d5d5;  
$super-light-grey: #edded;
```



```
def search  
  @per_page = 50  
  if @query = search_query  
    @foo = FooBar.full_text_search( @query,  
      :page => params[:page],  
      :order => "created_at DESC",  
      :per_page => @per_page)
```

MG  
15

# Avantaje



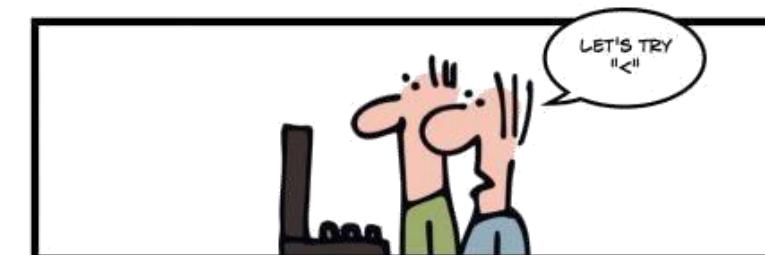
# Ce înseamnă Good Code

CLEAN Code

=

GOOD Code

GOOD CODERS...



# Ce înseamnă Bad Code

- Greu de citit și înțeles
- Induce în eroare
- Se strică atunci când îl modifici
- Are dependințe în multe module externe – *glass breaking code*
- Strâns legat (tight coupled) de alte secvențe de cod



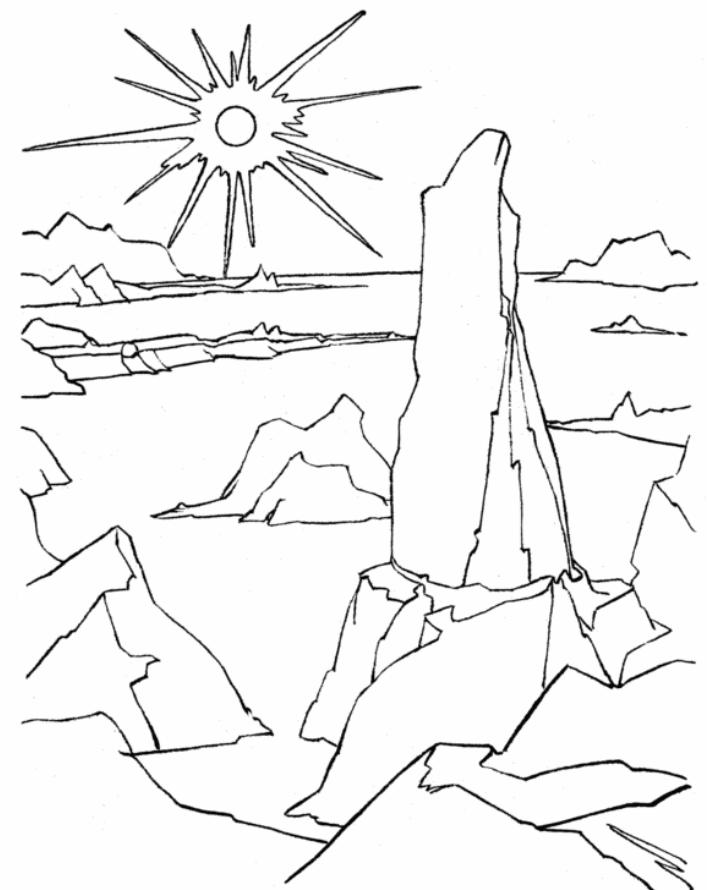
# Principii

- DRY
- KISS
- YAGNI
- SOLID



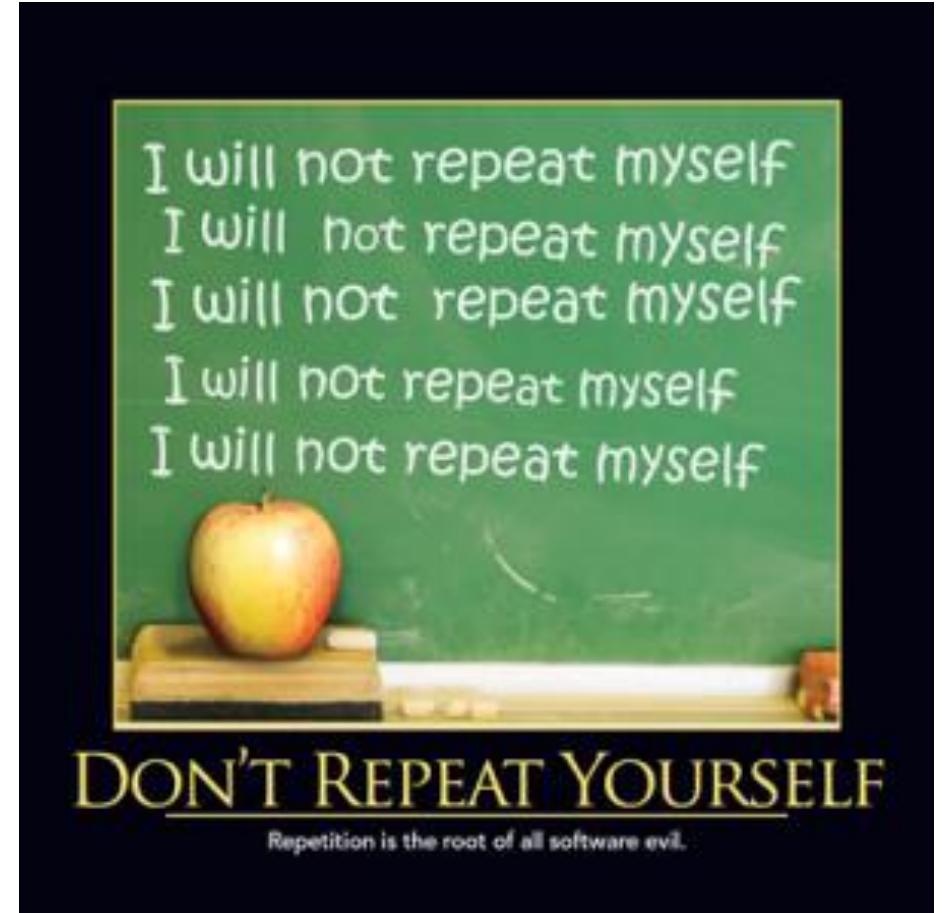
# D.R.Y.

- **Don't Repeat Yourself**
- Aplicabil ori de câte ori dăm Copy/Paste unei bucăți de cod



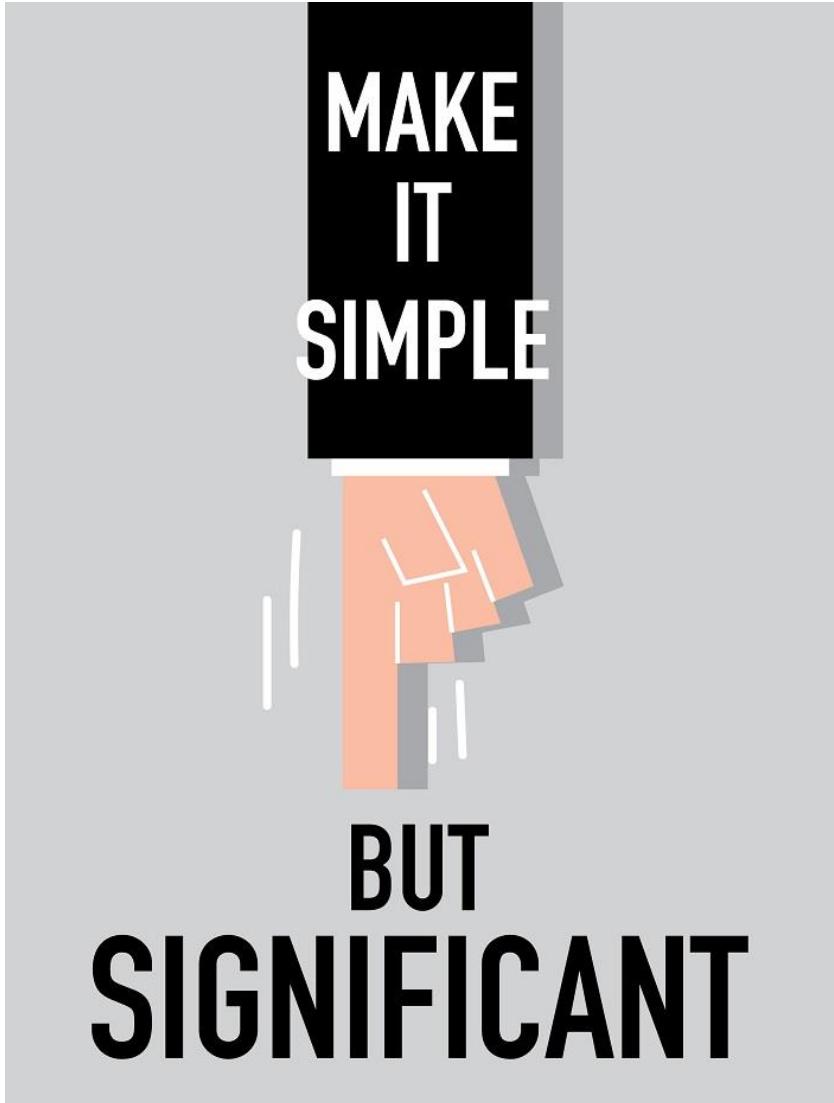
# D.R.Y.

- Sau de fiecare dată când fără să ne dăm seama scriem două metode care fac același lucru.



# K.I.S.S.

- Keep It Simple and Stupid
- Ori de câte ori vrem ca o metodă să facă de toate

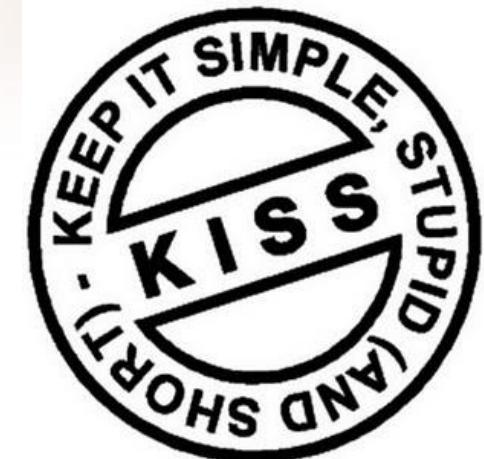


# K.I.S.S.

- **Beneficiile KISS:**
- Permite rezolvarea rapidă de probleme.
- Permite rezolvarea unor probleme complexe, într-o manieră simplă.
- Permite realizarea de produse complexe, ușor de întreținut.



BENEFITS



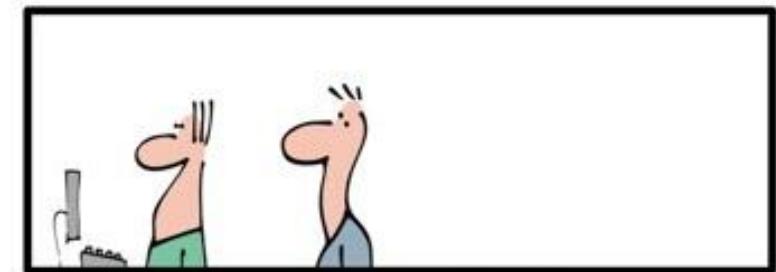
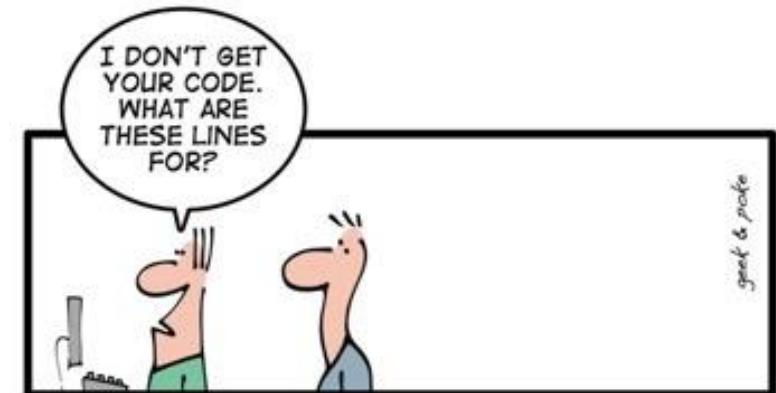
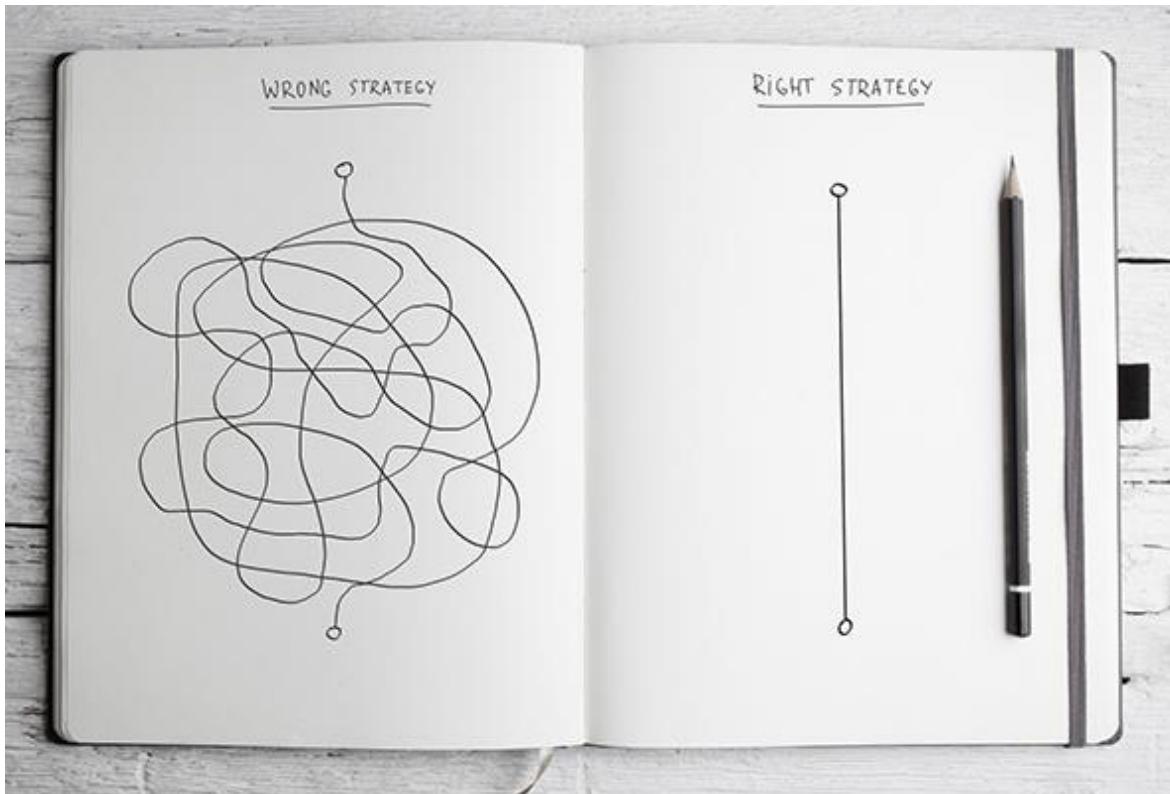
# K.I.S.S.

- **Beneficiile KISS:**
- Codul scris în această manieră este mult mai flexibil;
- Codul este mult mai ușor de extins și modificat dacă apar noi cerințe.



# K.I.S.S.

- **Dezavantaje**



THE ART OF PROGRAMMING - PART 2: KISS

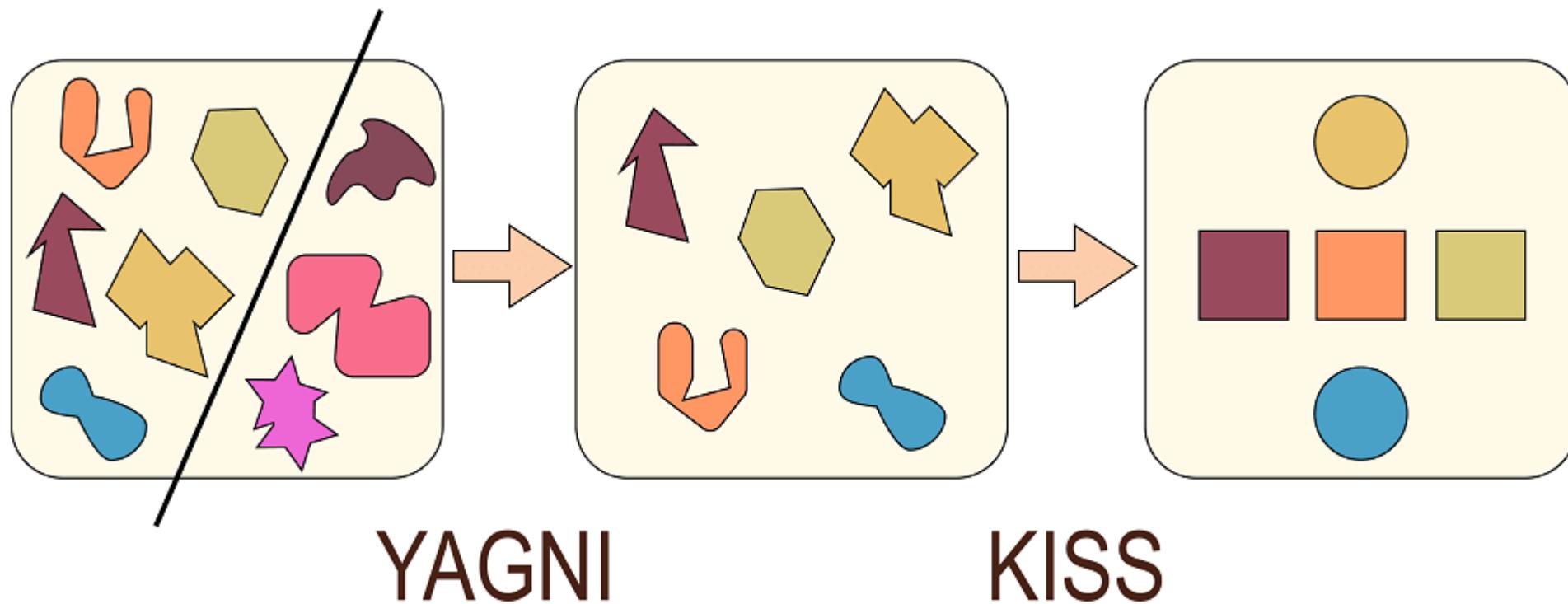
# Y.A.G.N.I.

- You Ain't Gonna Need It
- Nu scriem metode ce nu sunt necesare încă (poate nu vor fi necesare niciodată)
- Oarecum derivat din KISS



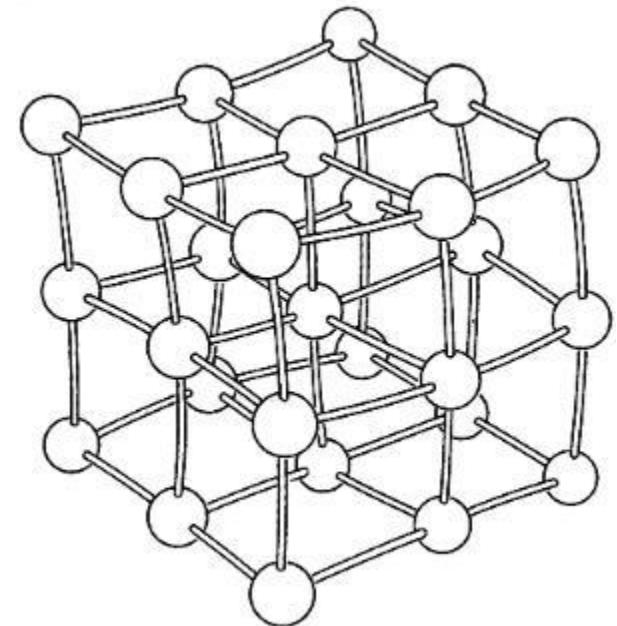
# Y.A.G.N.I.

- Oarecum derivat din KISS



# S.O.L.I.D.

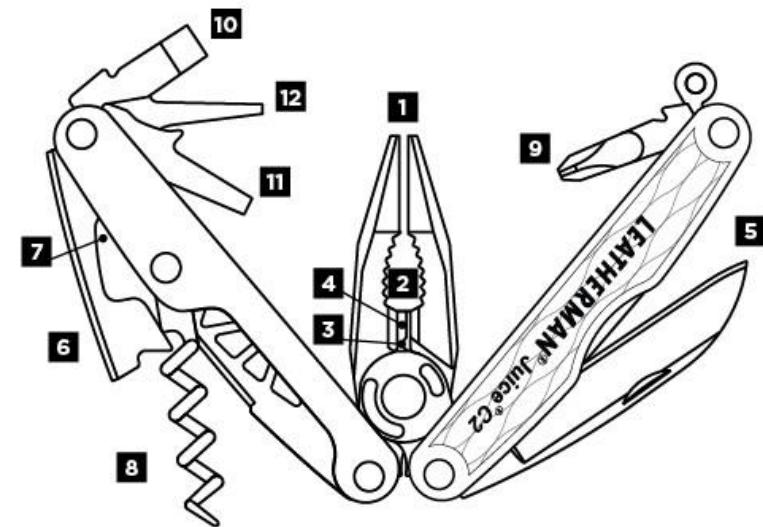
- Single responsibility (SRP)
- Open-closed (OCP)
- Liskov substitution (LSP)
- Interface segregation (ISP)
- Dependency inversion



[https://en.wikipedia.org/wiki/SOLID\\_\(object-oriented\\_design\)](https://en.wikipedia.org/wiki/SOLID_(object-oriented_design))

# Single Responsibility Principle

- O clasă trebuie să aibă întotdeauna o singură responsabilitate și numai una
- În caz contrar orice schimbare de specificații va duce la inutilitatea ei și rescrierea întregului cod
- *A class should have only one reason to change*  
(Robert C. Martin - Agile Software Development, Principles, Patterns, and Practices)



# Single Responsibility Principle

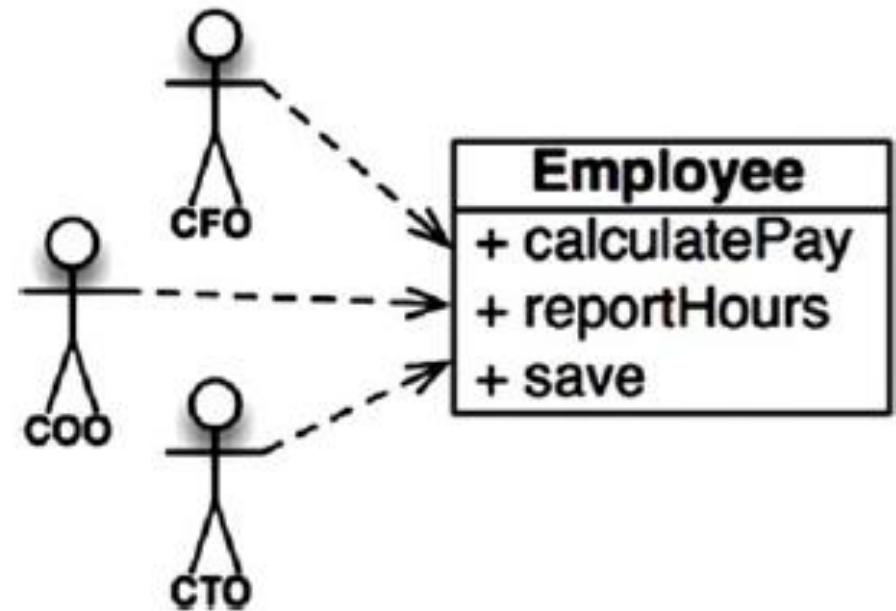
```
class Student{  
    void platesteTaxa(){ }  
    void sustineExamenPOO(){ }  
    void salvareBazaDate(){ }  
}
```

- O clasă despre un student
- Depinde de modificări din 3 zone diferite
  - contabilitate
  - academic
  - departament IT



# Single Responsibility Principle

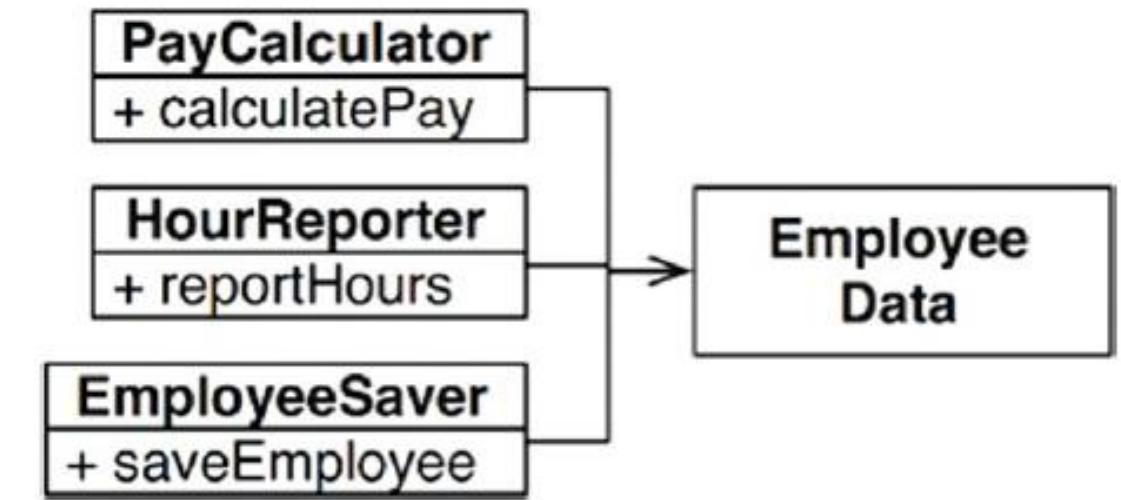
- O clasă angajat depinde de trei actori. Deci, poate fi modificată de oricare dintre cei trei actori:
  - Director Financiar;
  - Director General;
  - Directorul tehnic.



Robert C. Martin

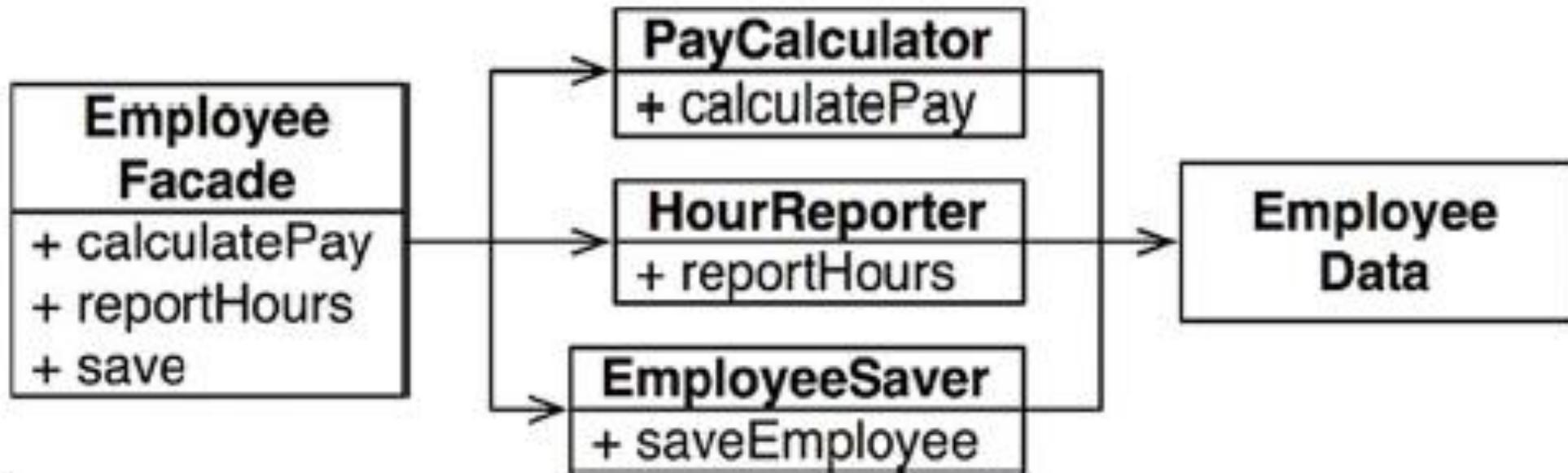
# Single Responsibility Principle

- Soluția este împărțirea clasei în trei clase, astfel încât fiecare clasă să răspundă unui actor.
- Acum apare însă o altă problemă. Care este aceasta?



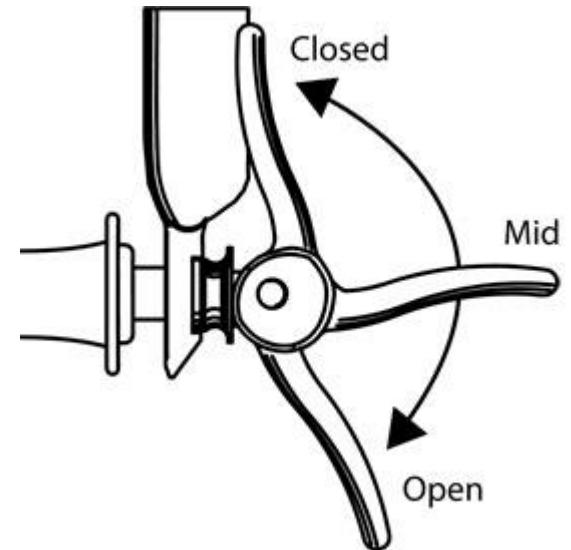
Robert C. Martin

# Single Responsibility Principle



# Open-Close Principle

- Clasele trebuie să fie deschise (open) pentru extensii
- Dar totuși închise (closed) pentru modificări



[https://en.wikipedia.org/wiki/Open/closed\\_principle](https://en.wikipedia.org/wiki/Open/closed_principle)

# Liskov Substitution Principle

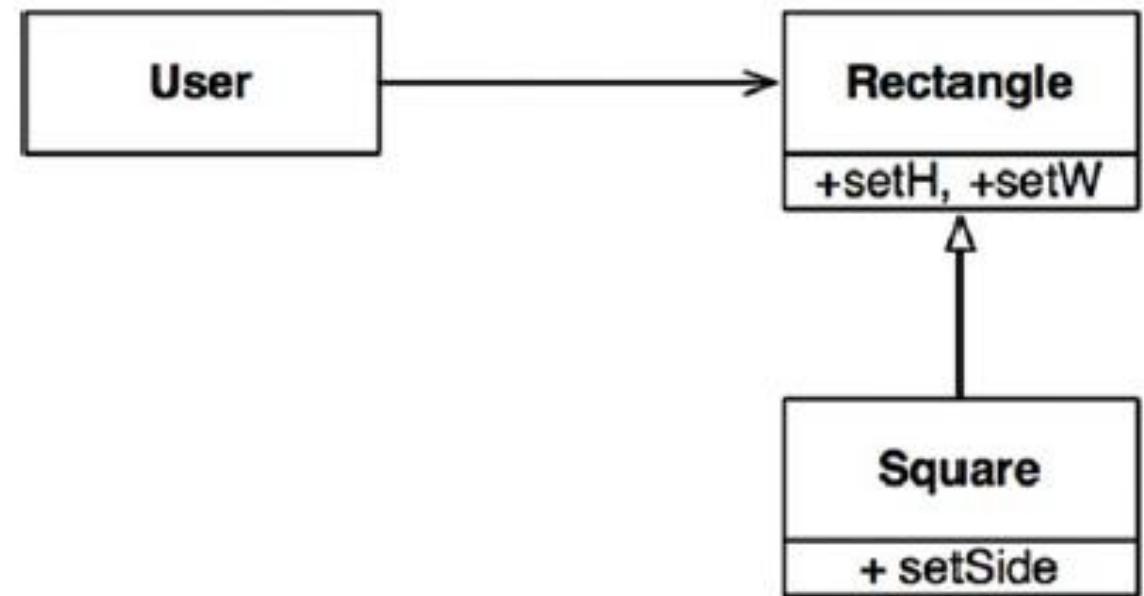
- Obiectele pot fi înlocuite oricând cu instanțe ale claselor derivate fără ca acest lucru să afecteze funcționalitatea
- Întâlnită și sub denumirea de „Design by Contract”



[https://en.wikipedia.org/wiki/Liskov\\_substitution\\_principle](https://en.wikipedia.org/wiki/Liskov_substitution_principle)

# Liskov Substitution Principle

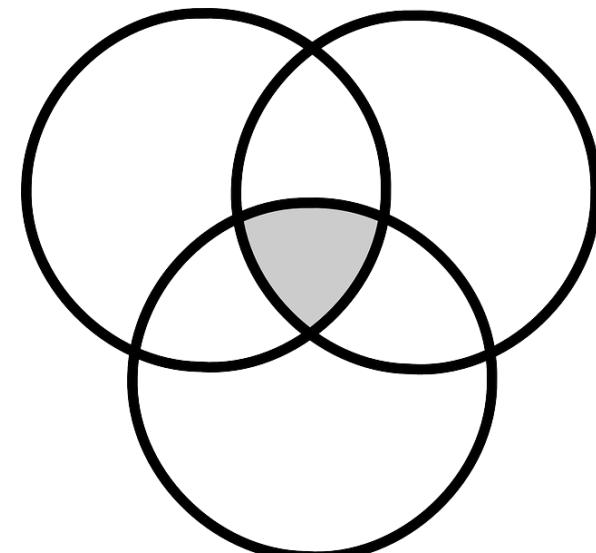
O încălcare a acestui principiu conduce la adoptarea unor mecanisme suplimentare de rezolvare a problemei și de utilizarea adecvată a claselor.



Robert C. Martin

# Interface Segregation Principle

- Mai multe interfețe specializabile sunt oricând de preferat unei singure interfețe generale
- Nu riscăm astfel ca prin modificarea „contractului” unui client să modificăm și contractele altor clienți
- Obiectele nu trebuie obligate să implementeze metode care nu sunt utile

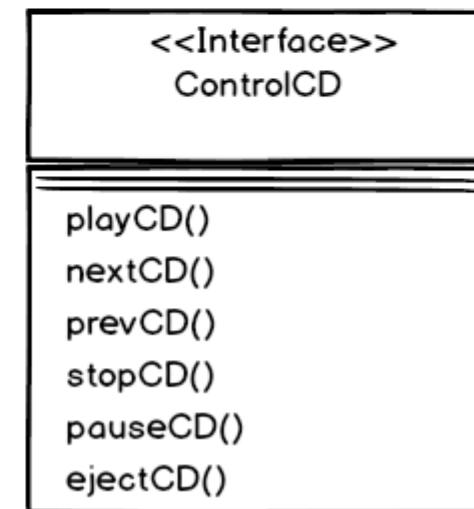
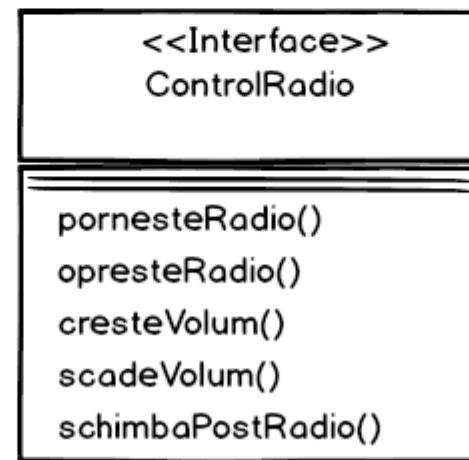
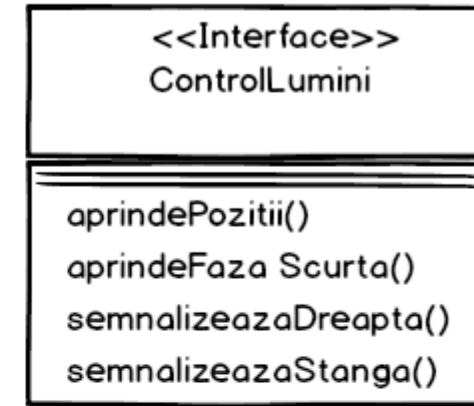
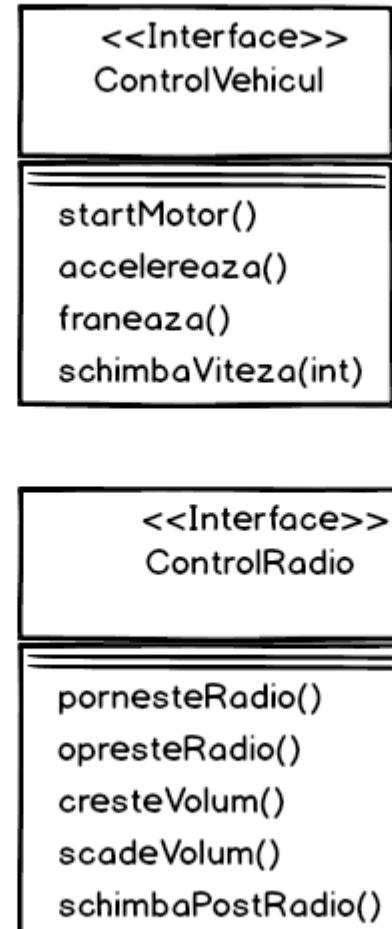


[https://en.wikipedia.org/wiki/Interface\\_segregation\\_principle](https://en.wikipedia.org/wiki/Interface_segregation_principle)

# Interface Segregation Principle



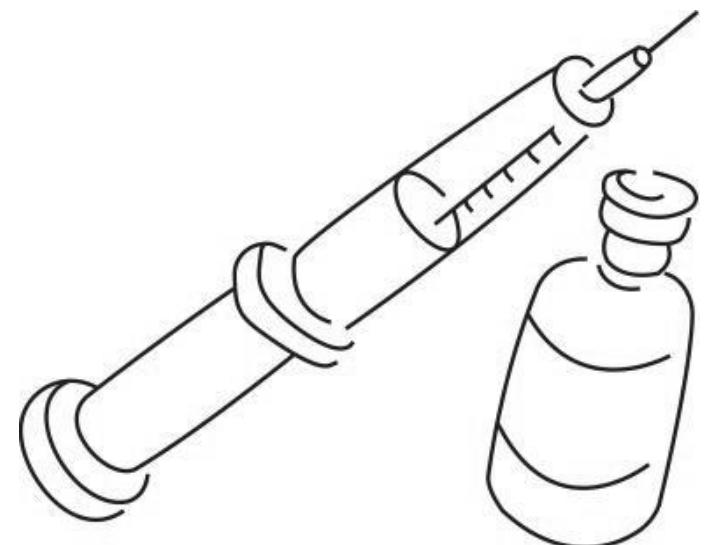
VS



# Dependency Inversion Principle

*Program to interfaces, not implementations*

*Depend on abstractions. Do not depend on concrete classes*



[https://en.wikipedia.org/wiki/Dependency\\_inversion\\_principle](https://en.wikipedia.org/wiki/Dependency_inversion_principle)

# Ce părere aveți despre codul de mai jos?

```
public static int Calculeaza() {  
    int x = 5;  
    ArrayList l = new ArrayList();  
    l.add(x);  
    int y = 10;  
    l.add(y);  
    l.add(15);  
    int m = 0;  
    for(Object k : l) {  
        m+= (int)k;  
    }  
    return m;  
}
```



# Convenții de nume

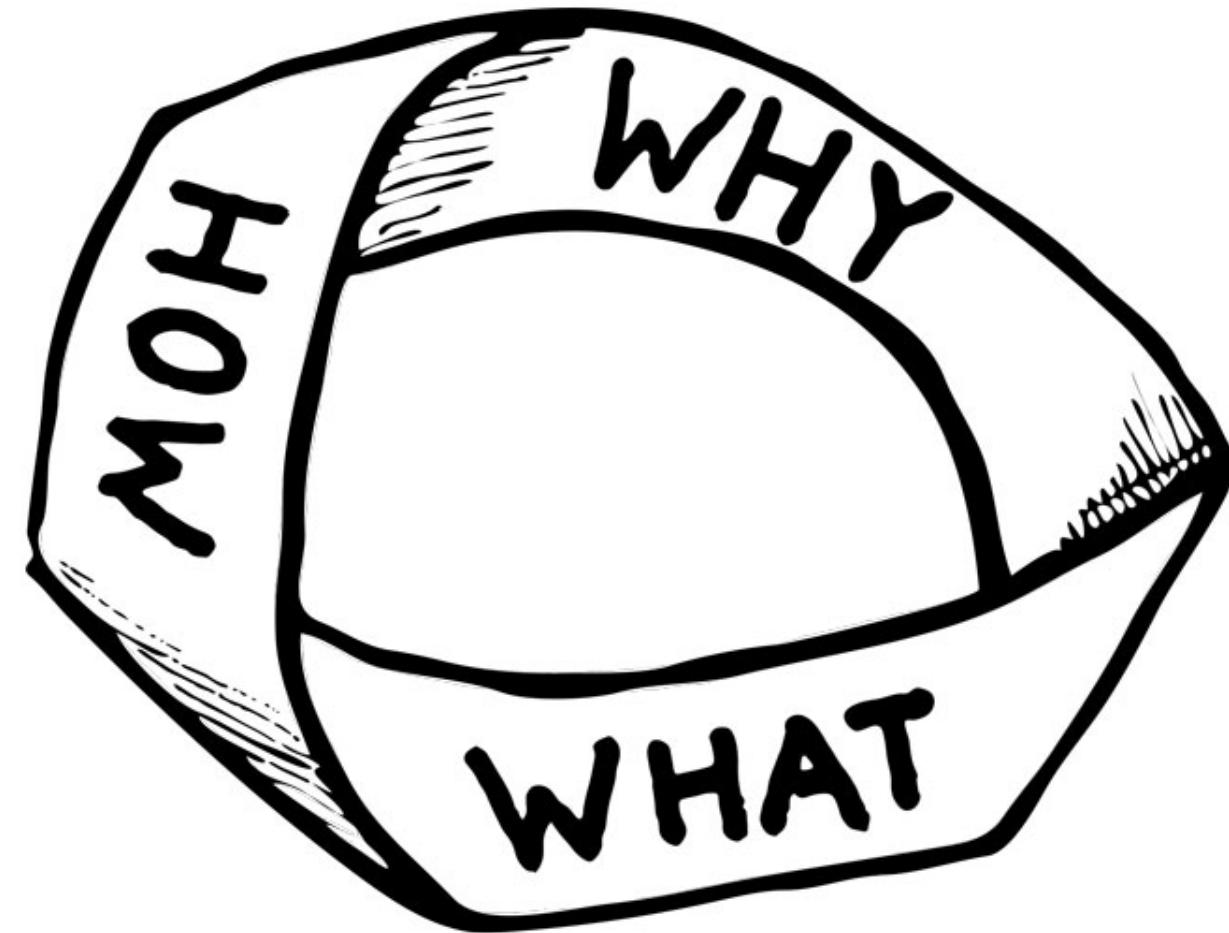
- UpperCamelCase
- lowerCamelCase
- System Hungarian Notation
- Apps Hungarian Notation



# Convenții de nume

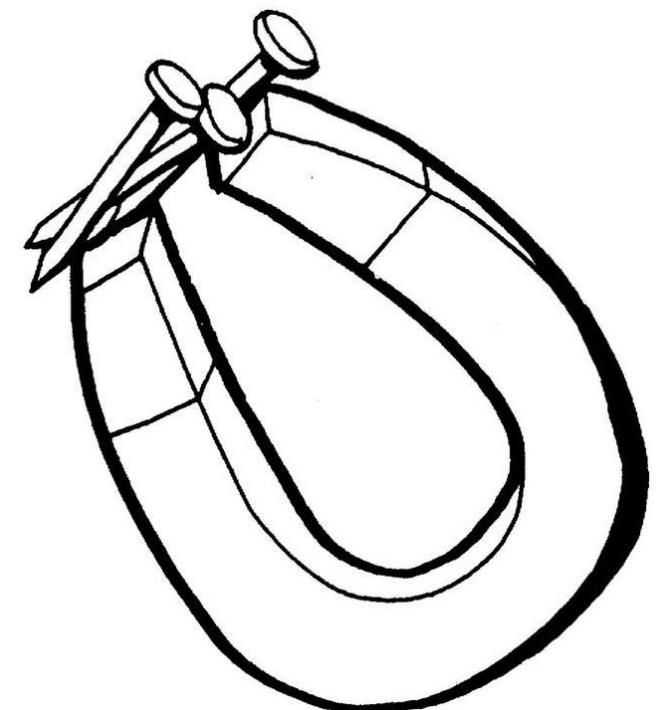
Un nume trebuie construit, astfel încât să răspundă la următoarele întrebări:

- De ce există? (**WHY?**)
- Ce face? (**WHAT?**)
- Cum este folosit? (**HOW?**)



# Convenții de nume - Clase

- Atenție la denumirile alese
- Numele prost alese sunt un magnet pentru programatorii leneși
- Compuze dintr-un substantiv specific, fără prefixe și sufixe inutile
- Nu trebuie să uităm de Single Responsibility Principle



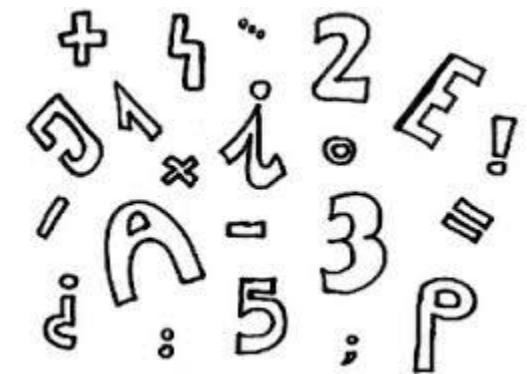
# Convenții de nume - Metode

- Trebuie să descrie clar ce fac
- Unde denumirile prost alese nu pot fi evitate (sunt generate automat de mediu) e indicat ca în interiorul lor să fie doar apeluri de alte metode
- Dacă denumirea unei metode conține o conjuncție („și”, „sau”, „ori”) cel mai probabil vorbim de două metode
- Nu abr den met



# Convenții de nume - Variabile

- Nu e indicat ca variabilele de tip siruri de caractere să conțină cod din alte limbaje (SQL, CSS)
- Variabilele booleane trebuie să sune ca o întrebare la care se poate răspunde cu adevărat/fals  
*boolean isTerminated = false;*
- Când există variabile complementare, numele trebuie să fie simetrice



# Reguli de scriere a codului sursă

```
protected String nume; protected String prenume;protected int varsta;
```

Declararea unei variabile  
pe fiecare linie.

```
public class Persoana {  
    protected String nume;  
    protected String prenume;  
    protected int varsta;
```

# Reguli de scriere a codului sursă

Blocurile de cod încep cu  
{ și se termină cu }.

```
public double getMedie() {  
    return medie;  
}
```

Chiar dacă avem o  
singură instrucțiune.

# Reguli de scriere a codului sursă

Blocurile cu instrucțiuni  
sunt marcate și prin  
identare.

```
function register()
{
    if (!empty($_POST)) {
        $msg = '';
        if ($_POST['user_name']) {
            if ($_POST['user_password_new']) {
                if ($_POST['user_password_repeat']) {
                    if (strlen($_POST['user_password_new']) === $_POST['user_password_repeat']) {
                        if (strlen($_POST['user_name']) < 65 || strlen($_POST['user_name']) > 1) {
                            if (preg_match('/^([a-zA-Z0-9]+)$/', $_POST['user_name'])) {
                                $user = read_user($_POST['user_name']);
                                if (!isset($user['user_name'])) {
                                    if ($_POST['user_email']) {
                                        if (strlen($_POST['user_email']) < 65) {
                                            if (filter_var($_POST['user_email'], FILTER_VALIDATE_EMAIL)) {
                                                create_user();
                                                $_SESSION['msg'] = 'You are now registered so please login';
                                                header('Location: ' . $_SERVER['PHP_SELF']);
                                                exit();
                                            } else $msg = 'You must provide a valid email address';
                                        } else $msg = 'Email must be less than 64 characters';
                                    } else $msg = 'Email cannot be empty';
                                } else $msg = 'Username already exists';
                            } else $msg = 'Username must be only a-z, A-Z, 0-9';
                        } else $msg = 'Username must be between 2 and 64 characters';
                    } else $msg = 'Password must be at least 6 characters';
                } else $msg = 'Passwords do not match';
            } else $msg = 'Empty Password';
        } else $msg = 'Empty Username';
        $_SESSION['msg'] = $msg;
    }
    return register_form();
}
```



# Reguli de scriere a codului sursă

Între headerul funcției și  
acolada de deschidere a  
blocului funcției se pune  
un spațiu

```
public void setMedie(double medie) {  
    this.medie = medie;  
}
```

# Reguli de scriere a codului sursă

Acolada de închidere a unui corp de instrucțiuni este singură pe linie.

Excepție fac situațiile când avem *if-else* sau *try-catch*.

```
public void setVarsta(int varsta) {  
    if (varsta > 0) {  
        this.varsta = varsta;  
    } else {  
        this.varsta = 1;  
    }  
}
```

# Reguli de scriere a codului sursă

Metodele sunt separate  
printr-o singură linie goală.

```
public double getMedie() {  
    return medie;  
}  
  
public void setMedie(double medie) {  
    this.medie = medie;  
}
```

# Reguli de scriere a codului sursă

Parametrii sunt separați prin virgulă și spațiu.

```
public Persoana(String nume, String prenume, int varsta) {
```

# Reguli de scriere a codului sursă

Operatorii sunt separați de  
operanzi printr-un spațiu.

```
return nume + " " + prenume;
```

Excepția de la această regulă  
fac operatorii unari.

# Reguli de Clean Code în structuri condiționale

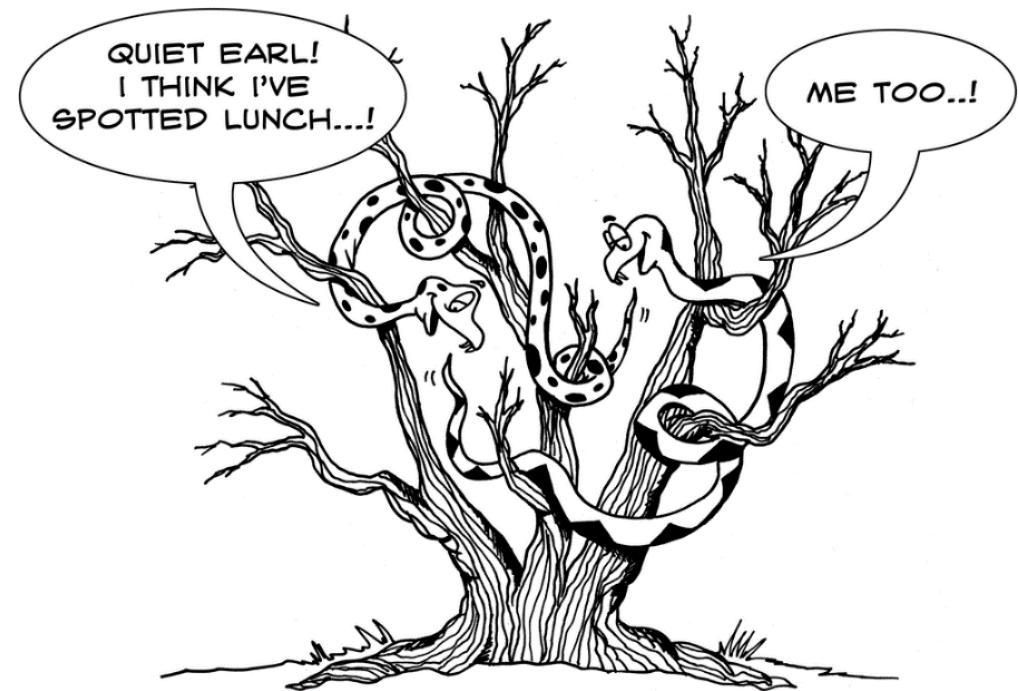
- Evitați comparațiile cu true și false

```
if(raspunsCorect == true) {  
    // ...  
}
```

- Variabilele booleane pot fi instantiată direct

```
boolean raspunsCorect;  
  
if(punctaj == 0) {  
    raspunsCorect = false;  
}  
else {  
    raspunsCorect = true;  
}
```

- Nu fiți negativiști!

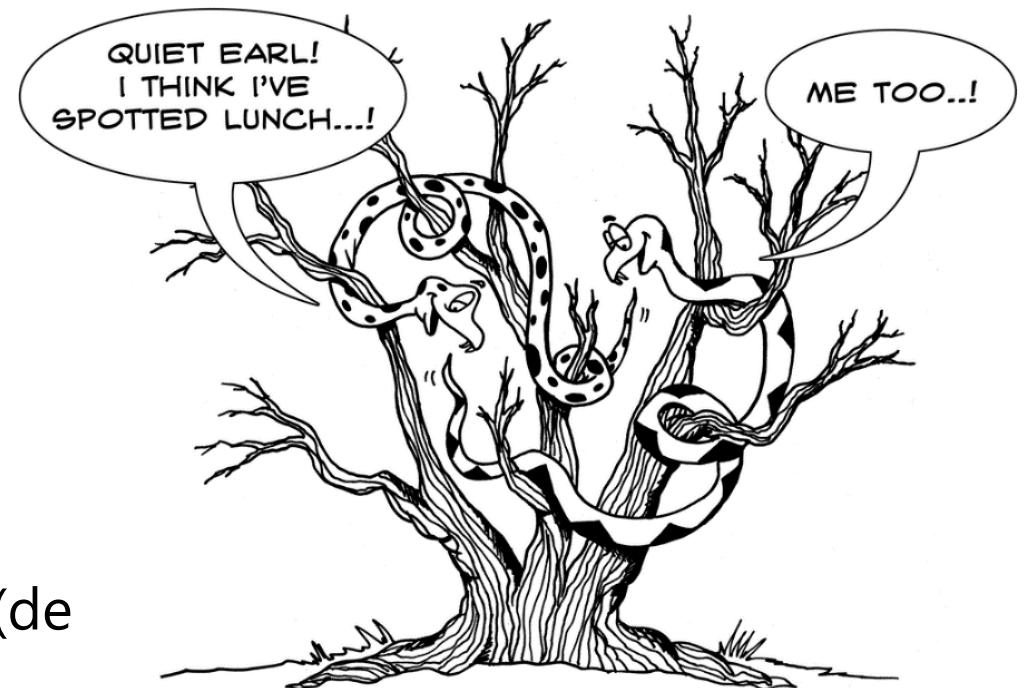


# Reguli de Clean Code în structuri condiționale

- Folosiți operatorul ternar ori de câte ori este posibil

```
int max = a > b ? a : b;
```

- Nu comparați direct cu stringuri, folosiți enum pentru situații de genul
- Constantele trebuie identificate și denumite (de obicei la începutul claselor)



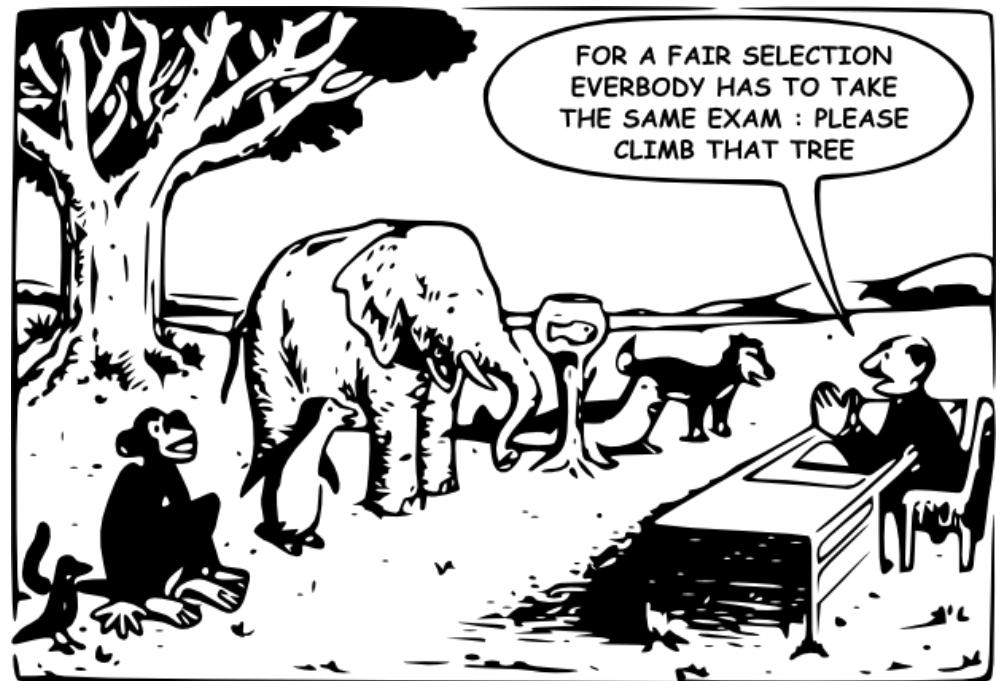
# Reguli de Clean Code în structuri condiționale

- Ori de câte ori condițiile devin prea mari sunt indicate variabilele intermediare
- De obicei folosirea *enum* denotă un design greșit al claselor
- Multe constante indică o nevoie de înglobare a lor într-o tabelă din baza de date



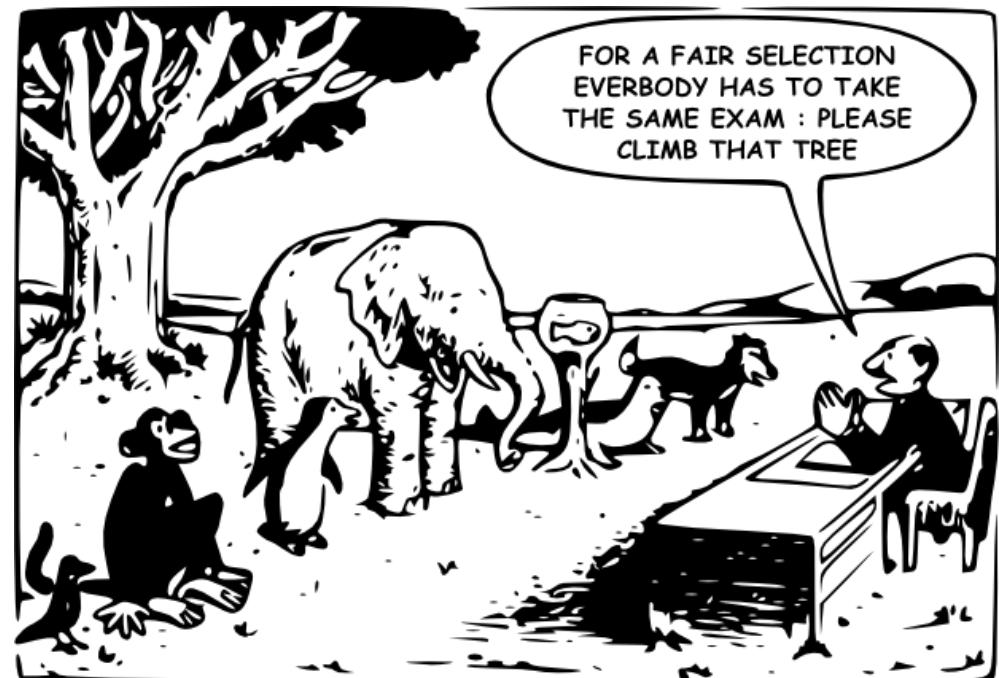
# Reguli de Clean Code în metode

- Orice metodă e indicat să aibă cel mult trei niveluri de structuri imbricate (arrow code)
- Întotdeauna se va încerca ieșirea din funcție cât mai repede posibil (prin return sau excepție)
- Variabilele vor fi declarate cât mai aproape de utilizarea lor
- Încercați pe cât posibil folosirea *this* și stabilirea unei convenții de nume pentru parametrii constructorului



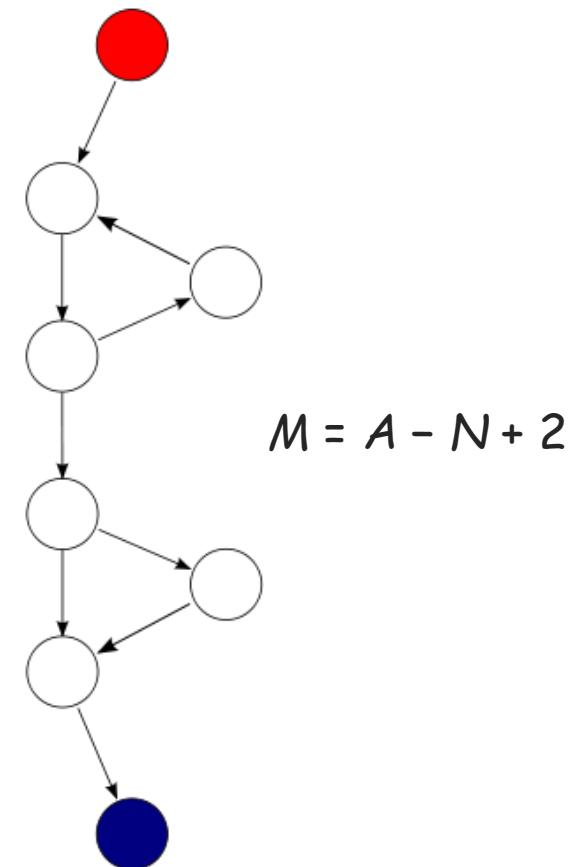
# Reguli de Clean Code în metode

- Evitați metodele cu mai mult de doi parametri
- Evitați metodele foarte lungi (peste 20 de linii de cod) – *one screen rule*
- Complexitatea trebuie să fie invers proporțională cu numărul de linii de cod
- Atenție la ordinea în care tratați exceptiile



# Reguli de Clean Code în metode

- Verificați complexitatea ciclometrică a metodelor
- Metodele simple au complexitate = 1
- Structurile de tip *if* și *switch* cresc complexitatea
- Valoarea determină numărul de teste

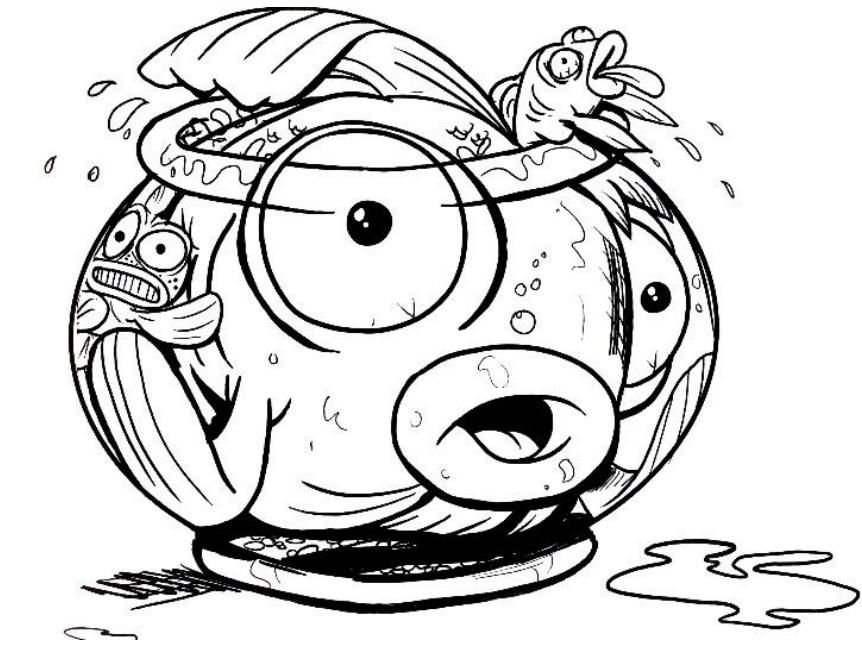
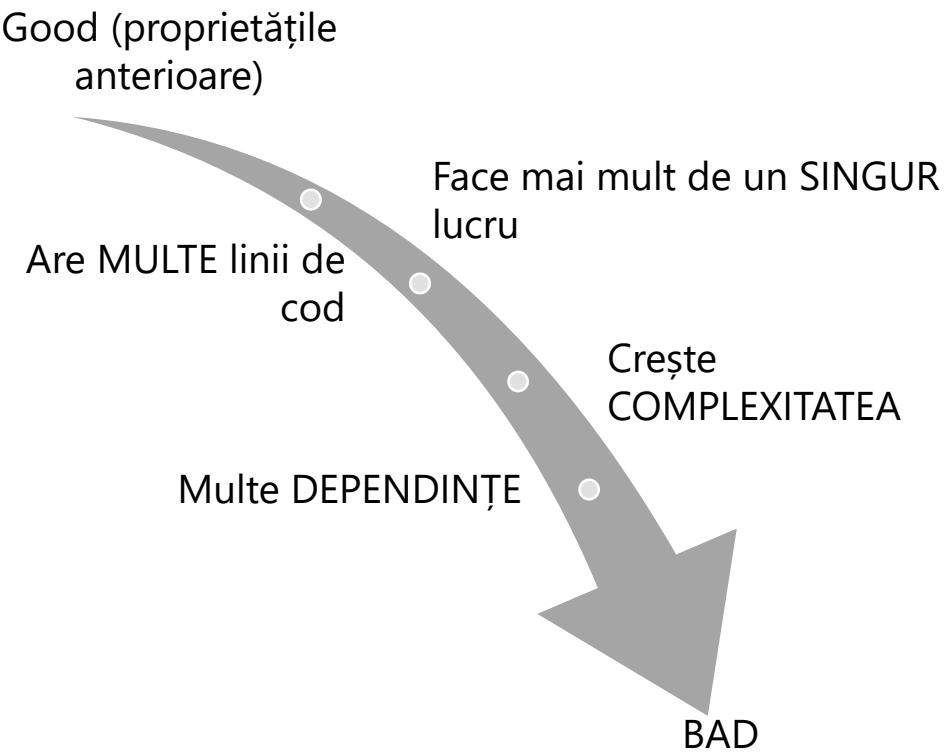


# Reguli SIMPLE de Clean Code pentru metode

- Single responsibility - SRD
- Keep It Simple & Stupid - KISS
- Deleagă prin pointeri/interfețe
- Folosește interfețe

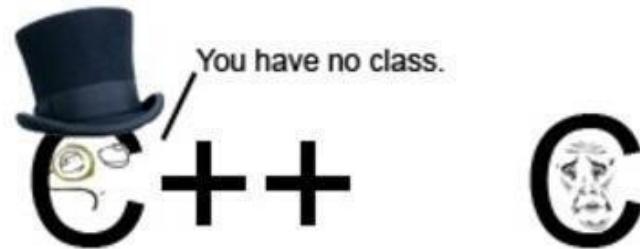


# Metode GOOD vs BAD



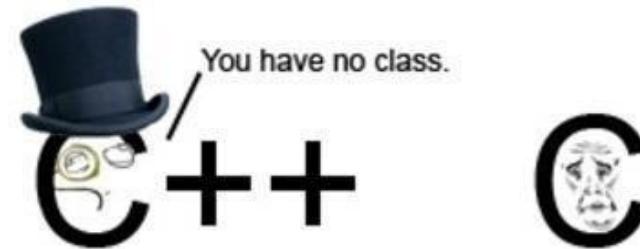
# Reguli de Clean Code în clase

- Toate metodele dintr-o clasă trebuie să aibă legătură cu acea clasă
- Evitați folosirea claselor generale și mutați prelucrările respective ca metode statice în clasele aferente
- Evitați primitivele ca parametri și folosiți obiecte (clase Wrapper in Java) ori de câte ori acest lucru este posibil



# Reguli de Clean Code în clase

- Atenție la primitive și în cazul prelucrărilor  
în mai multe fire de execuție
- Folosiți fișiere de resurse pentru șirurile de  
caracter din GUI
- Clasele ce conlucrează vor fi așezate una  
lângă alta pe cât posibil
- Folosiți-vă de *design patterns* acolo unde  
situația o cere



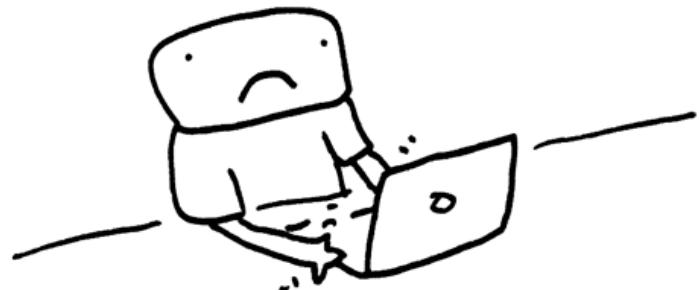
# Reguli de Clean code în comentarii

- De cele mai multe ori acestea nu își au deloc locul
- Codul bine scris este auto-explicativ
- Nu folosiți comentarii pentru a vă cere scuze

//When I wrote this, only God and I understood what I was doing

//Now, God only knows

You're lucky I don't  
hit "Submit" on most  
of the comments I write.

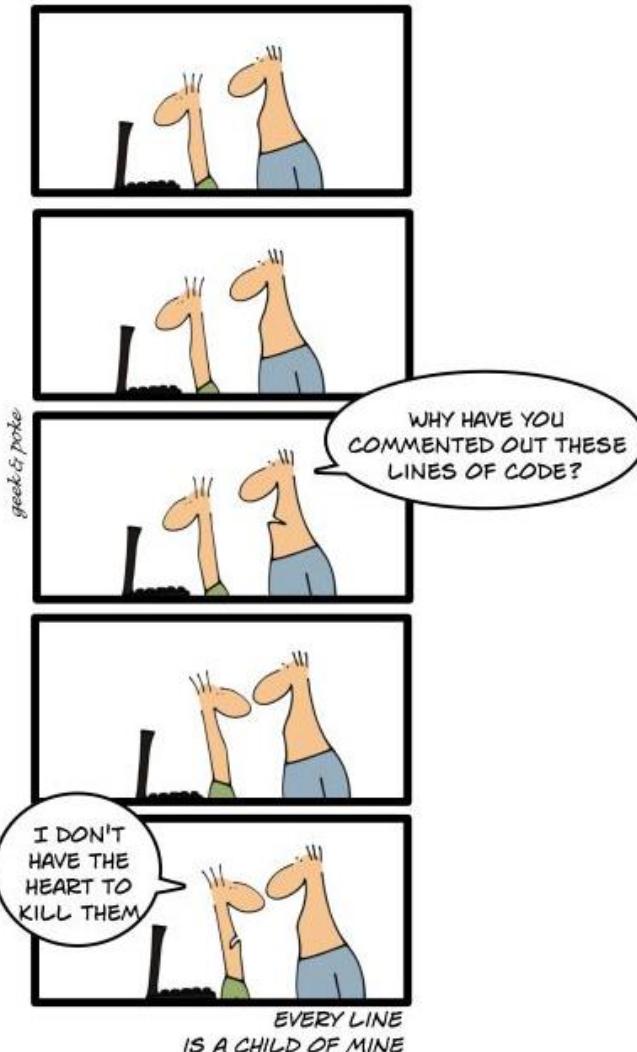


# Reguli de Clean code în comentarii

```
8 // Dear programmer:  
9 // When I wrote this code, only god and  
10 // I knew how it worked.  
11 // Now, only god knows it!  
12 //  
13 // Therefore, if you are trying to optimize  
14 // this routine and it fails (most surely),  
15 // please increase this counter as a  
16 // warning for the next person:  
17 //  
18 // total_hours_wasted_here = 254  
19 //  
20
```

# Reguli de Clean code în comentarii

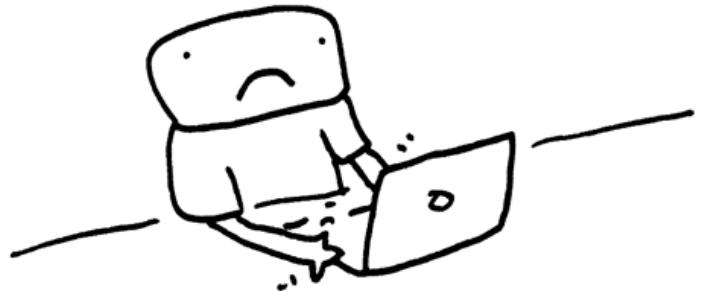
- Nu comentați codul nefolosit – devine *zombie*
- Există soluții de versionare pentru recuperarea codului modificat
- Atunci când simțiți nevoia de a folosi comentarii pentru a face o metodă lizibilă, cel mai probabil acea funcție trebuie separată în două funcții



# Reguli de Clean code în comentarii

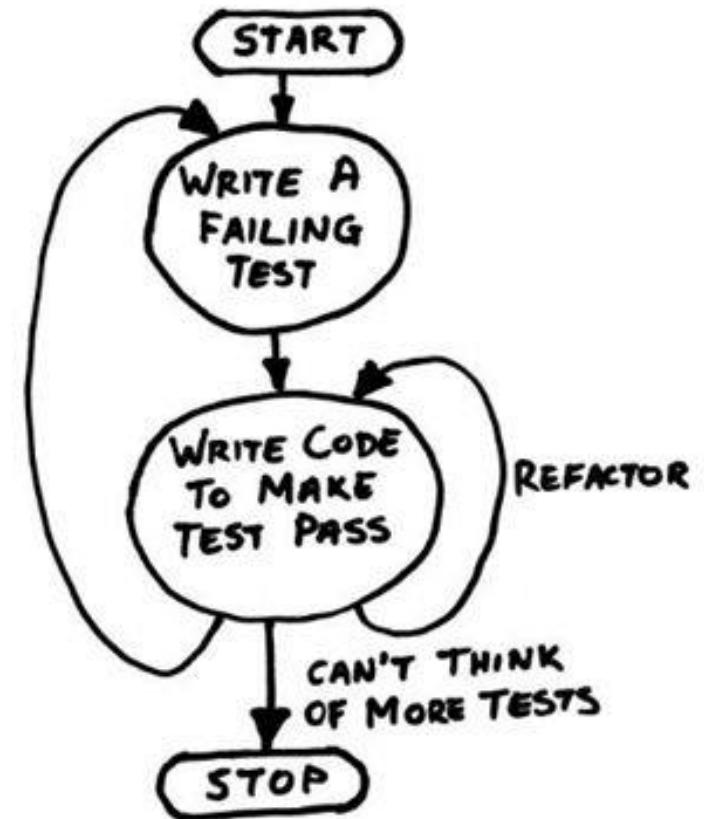
- Evitați blocurile de comentarii introductive
- Toate detaliile de acolo se vor găsi în soluția de versionare
- Sunt indicate doar pentru
  - biblioteci ce vor fi refolosite de alți programatori (doc comments) -  
<http://www.oracle.com/technetwork/articles/java/index-137868.html>
  - TODO comments

You're lucky I don't hit "Submit" on most of the comments I write.



# Scurt dictionar

- **Test Driven Development (TDD)** – Dezvoltare bazată pe cazuri de utilizare
- **Refactoring** – rescrierea codului într-o manieră ce se adaptează mai bine noilor specificații
- **Automatic Testing (Unit Testing)** – Testarea automată a codului pe baza unor cazuri de utilizare. Foarte utilă în refactoring pentru că putem verifica dacă am păstrat toate funcționalitățile sau nu (regression)
- **Code review** – Procedură întâlnită în special în AGILE (XP, SCRUM) ce presupune ca orice bucată de cod scrisă să fie revizuită și de un alt programator
- **Pair programming** – Tehnică specifică AGILE prin care programatorii lucrează pe perechi pentru task-uri complexe, pentru a învăța sau pentru a evita code review



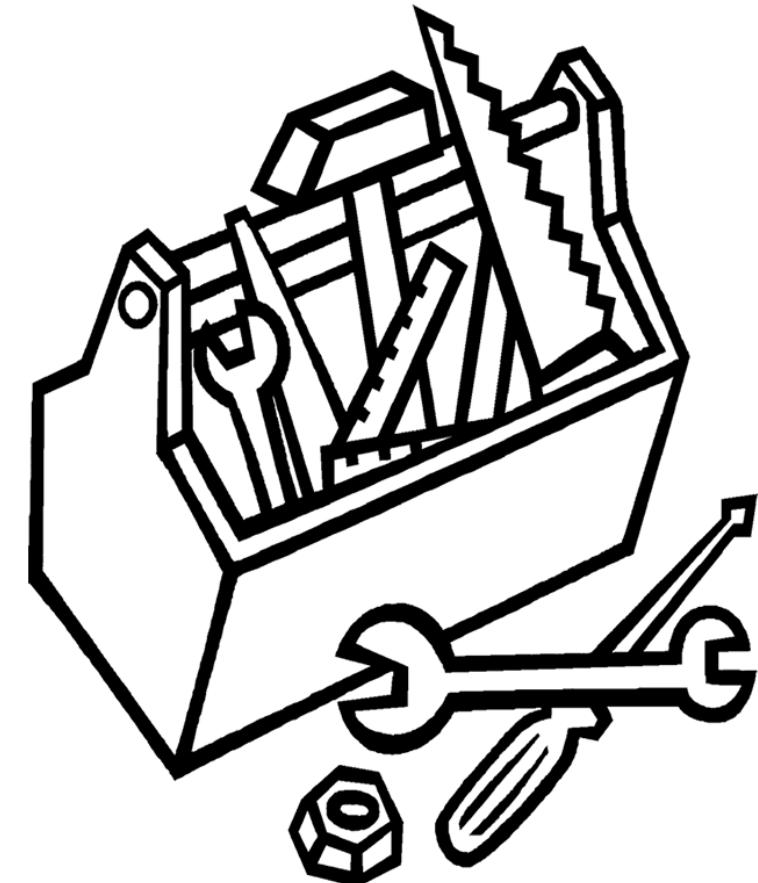
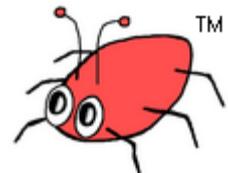
# Instrumente



IntelliJIDEA

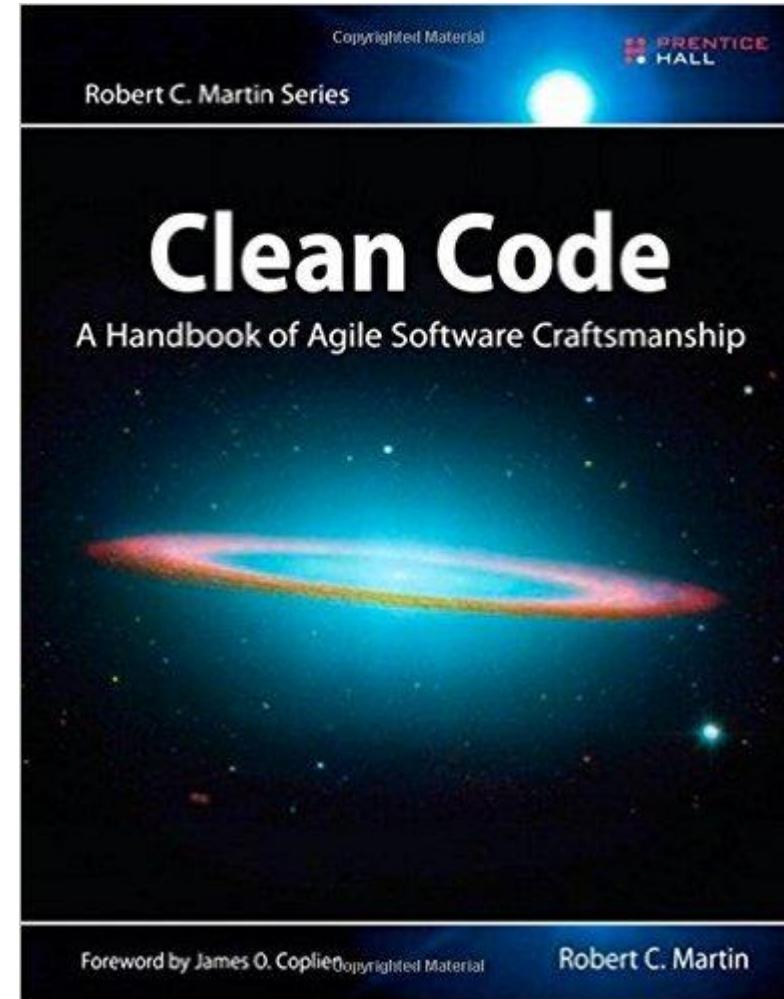


ReSharper



De citit

Robert C. Martin (Uncle Bob) – *Clean  
Code: A Handbook of Agile Software  
Craftsmanship*



# Bonus

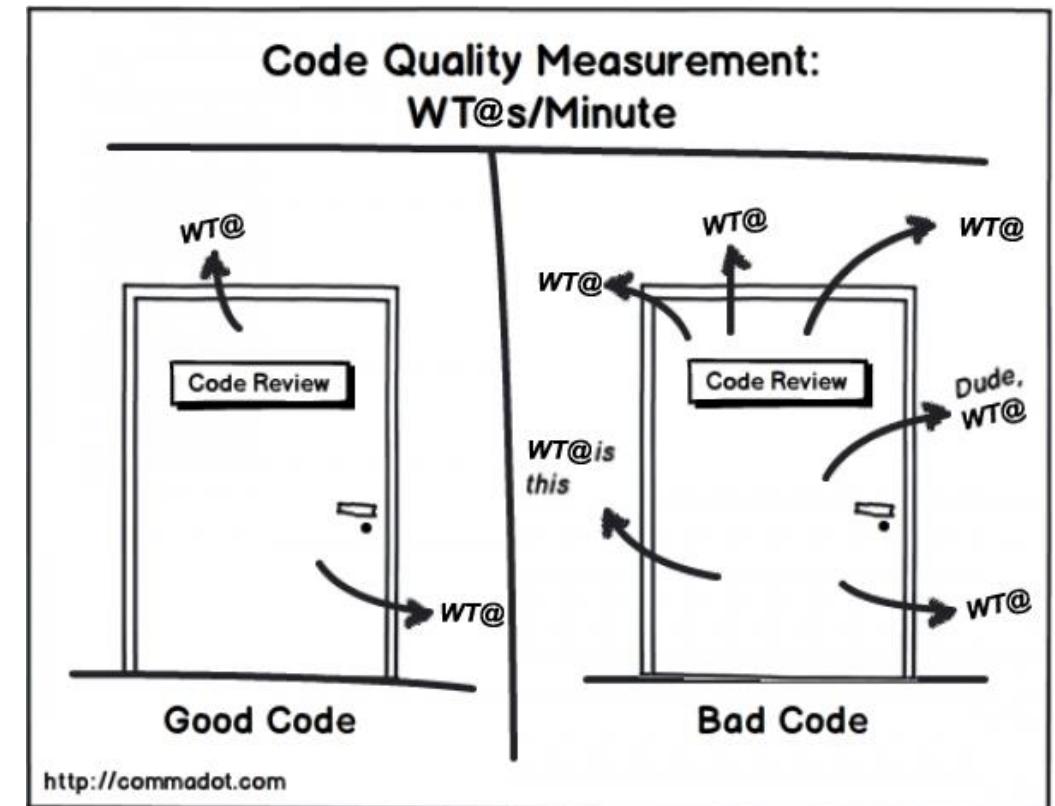
- The *Broken Window Principle*: clădirile cu ferestre sparte sunt mult mai vulnerabile la vandalism, care va duce la mai multe ferestre sparte;
- The *Boy Scout Rule*: lasați codul puțin mai curat decât l-ați găsit.
- Resurse suplimentare:
  1. Robert C. Martin (Uncle Bob) – Clean Code: A Handbook of Agile Software Craftsmanship
  2. Clean Code: Writing Code for Humans – Pluralsight series
  3. Design Principles and Design Patterns”, Robert C. Martin



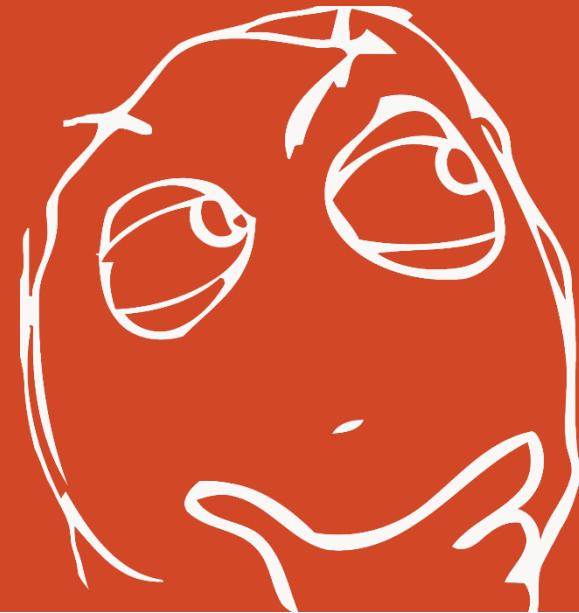
Încă ceva

- *"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live."*

Martin Golding



Întrebări?



Vă mulțumesc!

# Unit Testing – Assertions

ALIN ZAMFIROIU

# Ce este testarea?

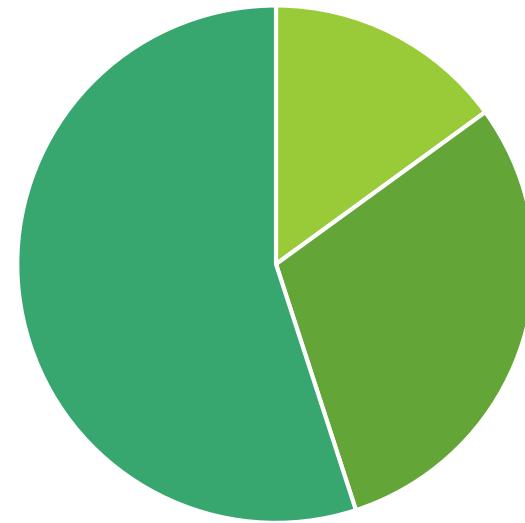
- ▶ Procesul de căutare a erorilor și al defectelor?
- ▶ Este utilizata pentru a semnala prezența defectelor, dar nu garantează absența acestora. - **Dijkstra**

# Termeni specifci

- ▶ Esec
- ▶ Defect
- ▶ Exceptie
- ▶ Problemă
- ▶ Eroare
- ▶ Incident
- ▶ Anomalie
- ▶ Inconsistență
- ▶ Aparență
- ▶ Neajuns
- ▶ Bug

# Cauzele erorilor software

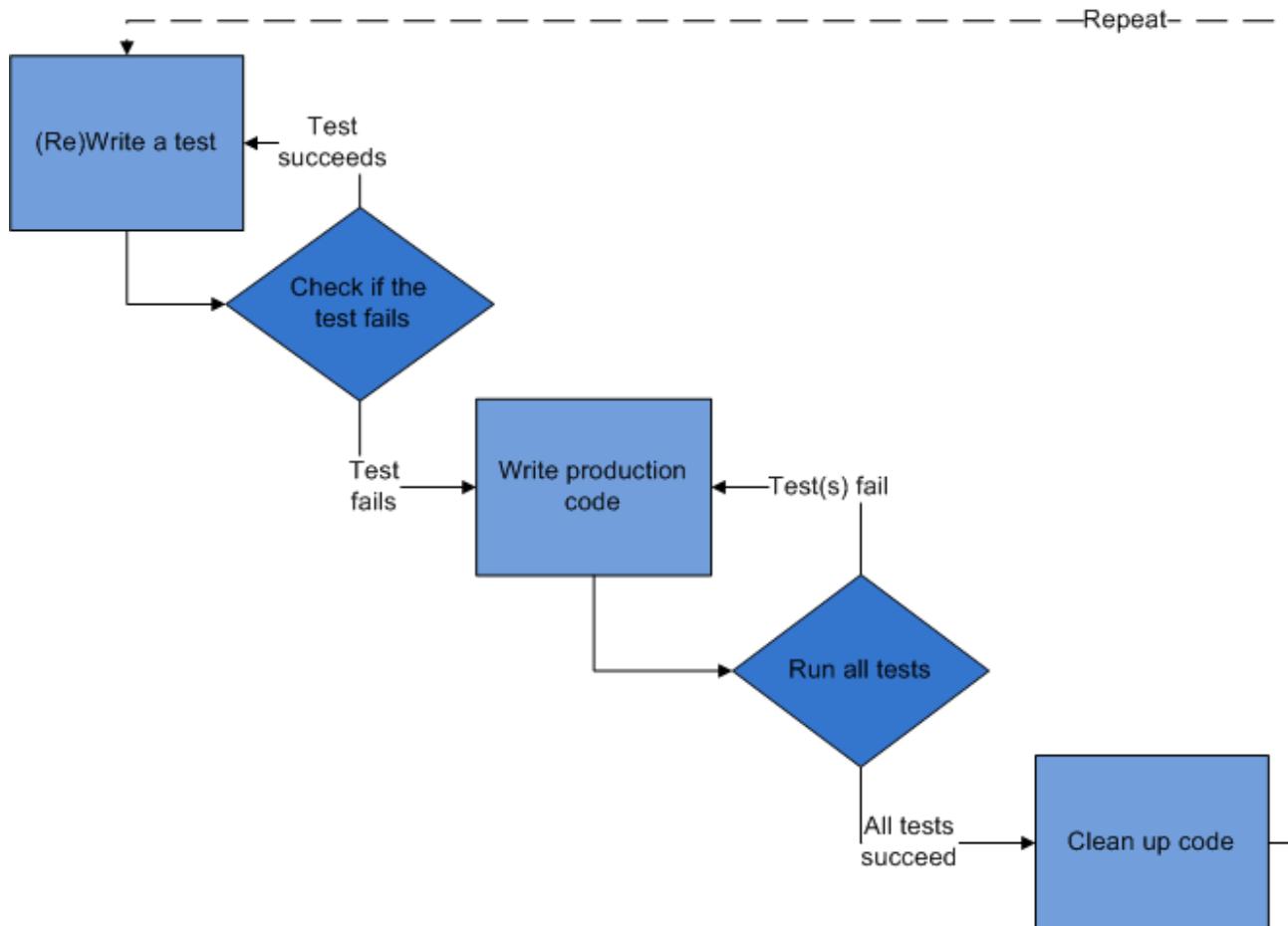
- ▶ Erori de programare – 15%
- ▶ Erori de proiectare – 30%
- ▶ Erori de specificații – 55%



# Ce este Unit Testing?

- ▶ O cale de testare a codului, de către programatori încă din etapa de dezvoltare a produsului software.
- ▶ **UNIT TEST** – secvență de cod folosită pentru testarea unei unități bine definite din codul aplicației software. De obicei unitatea este reprezentată de o metodă.
- ▶ Testarea unității se realizează într-un context bine definit în specificațiile de testare.

# Test Driven Development

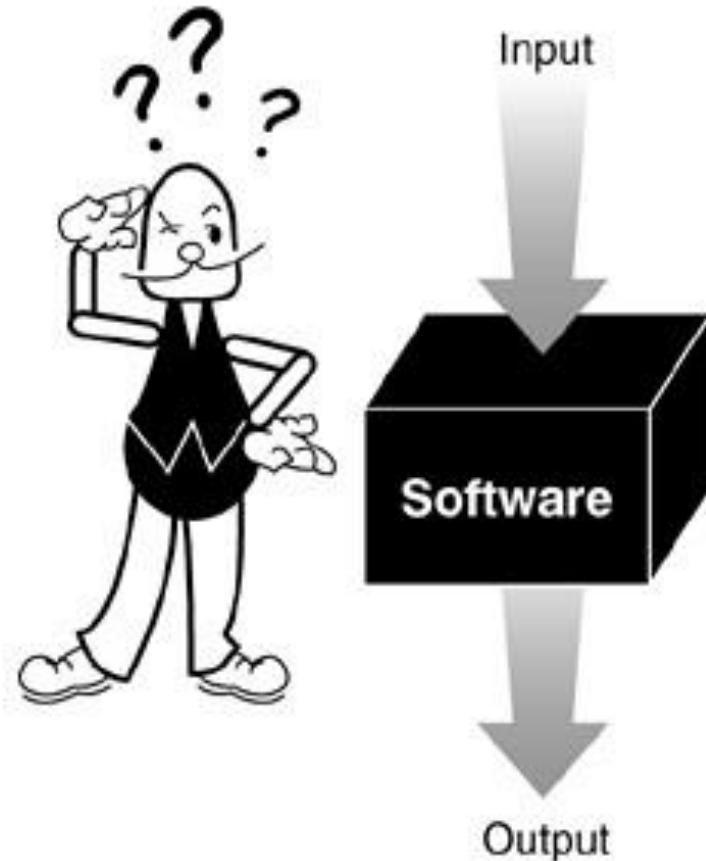


- ▶ Dezvoltarea pe baza testelor.
- ▶ Este exact modul de gândire al oamenilor pentru realizarea metodelor.

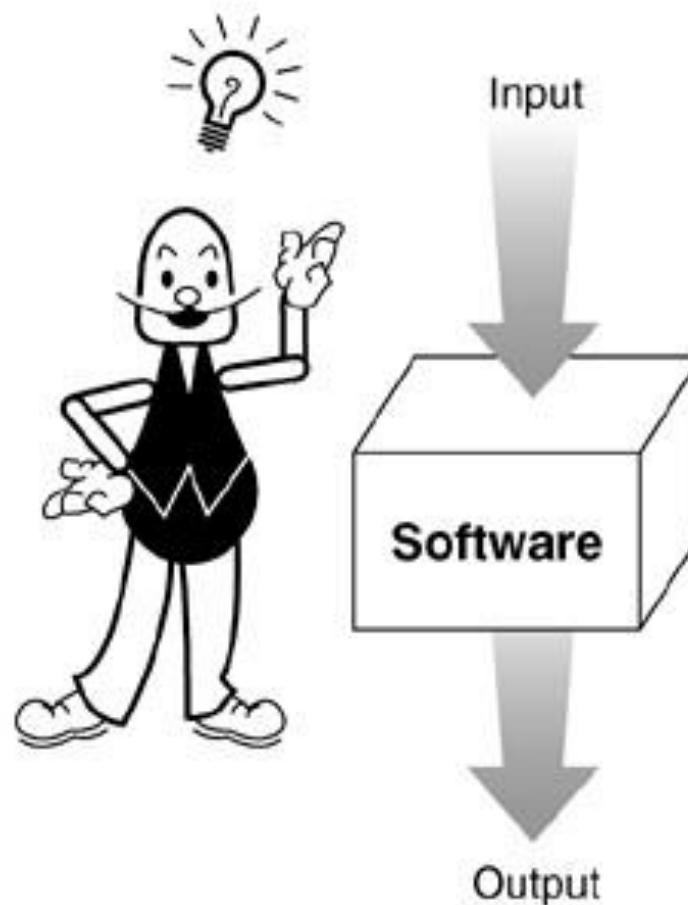
# Tipuri de testare



# WhiteBox sau BlackBox testing



**Black-Box Testing**



**White-Box Testing**

# Testarea BlackBox

- ▶ Testarea **BlackBox** sau testarea comportamentală, este o metodă utilizată pentru a testa aplicația software de către persoane care nu cunosc arhitectura internă a aplicației testate.
- ▶ Testerul cunoaște doar datele de intrare și datele de ieșire ale aplicației.

# Avantajele testării BlackBox

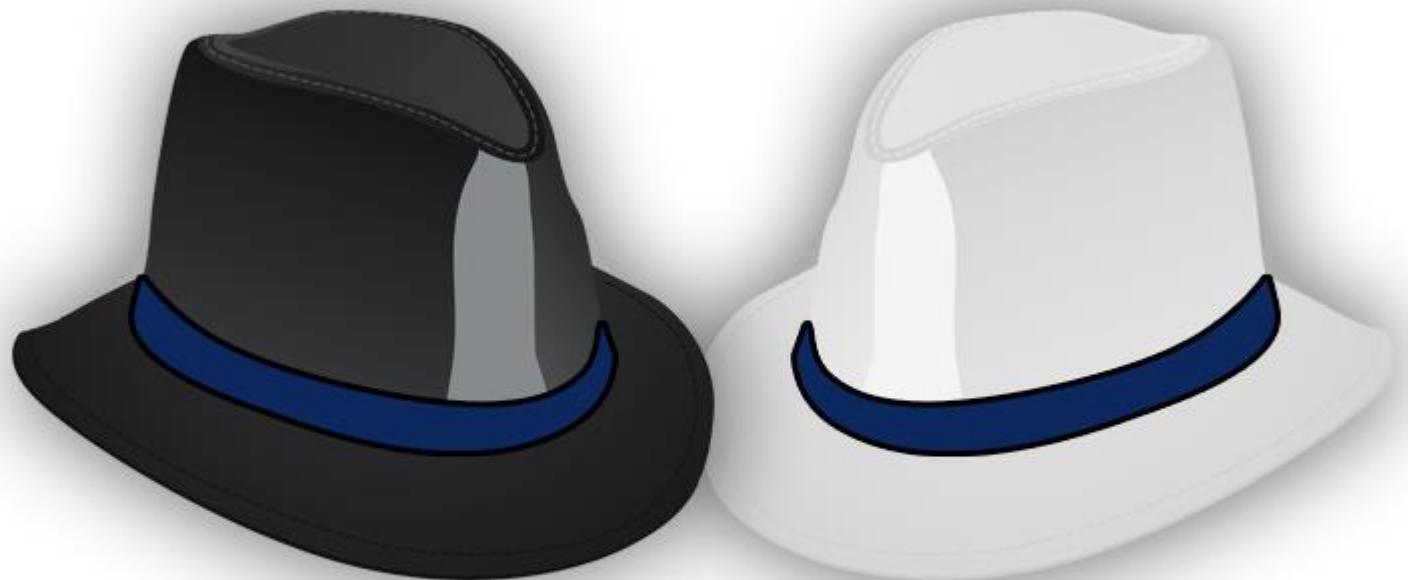
- ▶ Testele sunt realizate din punctul de vedere al utilizatorului.
- ▶ Testerul nu trebuie să știe programare, limbajul folosit pentru dezvoltare sau structura de cod a aplicației.
- ▶ Testele sunt efectuate independent de dezvoltatorii și au o perspectivă obiectivă.

# Dezavantajele testării BlackBox

- ▶ Cazurile de testare sunt dificil de proiectat, deoarece testerul nu are caietul de sarcini al aplicației;
- ▶ Testele vor avea un număr mic de intrări;
- ▶ Testele pot fi redundante cu alte teste realizate de dezvoltator.

# Testarea WhiteBox

- ▶ Este o metodă de testare folosită de către dezvoltatori sau de testeri care cunosc structura internă a aplicației testate.



# Testarea WhiteBox

- ▶ Testarea WhiteBox mai este cunoscută și sub formele:
  - ▶ Clear Box Testing;
  - ▶ Open Box Testing;
  - ▶ Glass Box Testing;
  - ▶ Transparent Box Testing;
  - ▶ Code-Based Testing;
  - ▶ Structural Testing.

# Avantajele testării WhiteBox

- ▶ Testarea poate fi începută într-o etapă anterioră punerii în funcțiune. Nu trebuie să se aștepte realizarea interfeței pentru a fi realizată testarea.
- ▶ Testarea este mai profundată, cu posibilitatea de a acoperi cele mai multe posibilități.

# Dezavantajele testării WhiteBox

- ▶ Din moment ce testele pot fi foarte complexe, sunt necesare resurse de înaltă calificare, cu o cunoaștere aprofundată a programării și a punerii în aplicare.
- ▶ Întreținerea codului de testare poate fi o povară în cazul în care punerea în aplicare se schimbă foarte des.

# Motive să folosești teste unitare

- ▶ Ușor de scris
- ▶ Testele pot fi scrise ad-hoc atunci când ai nevoie de ele
- ▶ Deși sunt simple, pe baza lor se pot defini colecții de teste – Test Suites
- ▶ Pot fi rulate automat de fiecare dată când e nevoie (write once, use many times)

# Motive să folosești teste unitare

- ▶ Există multe framework-uri și instrumente ce simplifică procesul de scriere și rulare
- ▶ Reduc timpul pierdut pentru debugging și pentru găsirea bug-urilor
- ▶ Reduc numărul de bug-uri în codul livrat sau integrat
- ▶ Crește rata bug-urilor identificate în faza de scrierea a codului

# Motive să nu folosești teste unitare

- ▶ De ce trebuie să îmi testez codul ?
- ▶ Codul scris de mine este corect !!!
- ▶ Nu am timp de teste. Trebuie să implementez funcționalități, nu teste.
- ▶ Nu este trecut în specificații că trebuie să facem teste.

# Framework-uri folosite pentru testare unitară

Framework	Programming language / scripting
<b>JUnit</b>	Java
<b>PHPUnit</b>	PHP
<b>PyUnit</b>	Python
<b>CPPUnit</b>	C++
<b>VBUnit</b>	Visual Basic
<b>DUnit</b>	Delphi
<b>cfcUnit</b>	ColdFusion
<b>HTMLUnit</b>	HTML și JavaScript
<b>Jsunit</b>	JavaScript
<b>dotUnit</b>	.NET
<b>NUnit</b>	C#, ASP.NET
<b>Ruby</b>	Ruby
<b>XMLUnit</b>	XML
<b>ASPUnit</b>	ASP
<b>xUnit</b>	C#

# JUnit

- ▶ Este un framework ce permite realizarea și rularea de teste pentru diferite metode din cadrul proiectelor dezvoltate.
- ▶ Cel mai folosit framework pentru testarea unitară a codului scris în JAVA.
- ▶ Reprezintă o adaptare de la xUnit.

# Junit - istoric

- ▶ **Kent Beck** a dezvoltat in anii 90 primul instrument de testare automata, **xUnit**, pentru Smalltalk
- ▶ Beck si Gamma (Gang of Four) au dezvoltat **JUnit** in timpul unui zbor de la Zurich la Washington, D.C.
- ▶ **JUnit** a devenit instrumentul standard pentru procesele de dezvoltare de tip **TDD** - Test-Driven Development in Java
- ▶ **JUnit** este componenta standard in multiple IDE-uri de Java (Eclipse, BlueJ, Jbuilder, DrJava, IntelliJ)
- ▶ Instrumentele de tip **Xunit** au fost dezvoltate si pentru alte limbaje (Perl, C++, Python, Visual Basic, C#, ...)

# JUnit

- ▶ JUnit funcționează conform a două design patterns: **Composite** și **Command**.
- ▶ O clasă TestCase reprezintă un obiect Command iar o clasă TestSuite este compusă din mai multe instanțe TestCase sau TestSuite.

# Concepte JUnit

- ▶ **Fixture** – set de obiecte utilizate în test
- ▶ **Test Case** – clasă ce definește setul de obiecte (fixture) pentru a rula mai multe teste
- ▶ **Setup** – o metodă/etapă de definire a setului de obiecte utilizate (fixture), înainte de testare.
- ▶ **Teardown** – o metodă/etapă de distrugere a obiectelor (fixture) după terminarea testelor
- ▶ **Test Suite** – colecție de cazuri de testare (test cases)
- ▶ **Test Runner** – instrument de rulare a testelor (test suite) și de afișare a rezultatelor

# JUnit- Assertions

**assertEquals**(expected, actual)

**assertEquals**(expected, actual, delta)

**assertSame**(expected, actual)

**assertNotSame**(expected, actual)

**assertNull**(object)

**assertNotNull**(object)

**assertTrue**(condition)

**assertFalse**(condition)

**fail**(message)

**assertEquals**(message, expected, actual)

**assertEquals**(message, expected, actual, delta)

**assertSame**(message, expected, actual)

**assertNotSame**(message, expected, actual)

**assertNull**(message, object)

**assertNotNull**(message, object)

**assertTrue**(message, condition)

**assertFalse**(message, condition)

**fail**(message)

# JUnit și NUnit

## JUnit

assertEquals

assertEquals

assertSame

assertNotSame

assertNull

assertNotNull

assertTrue

assertFalse

## NUnit

Assert.AreEqual

Assert.AreNotEqual

Assert.AreSame

Assert.AreNotSame

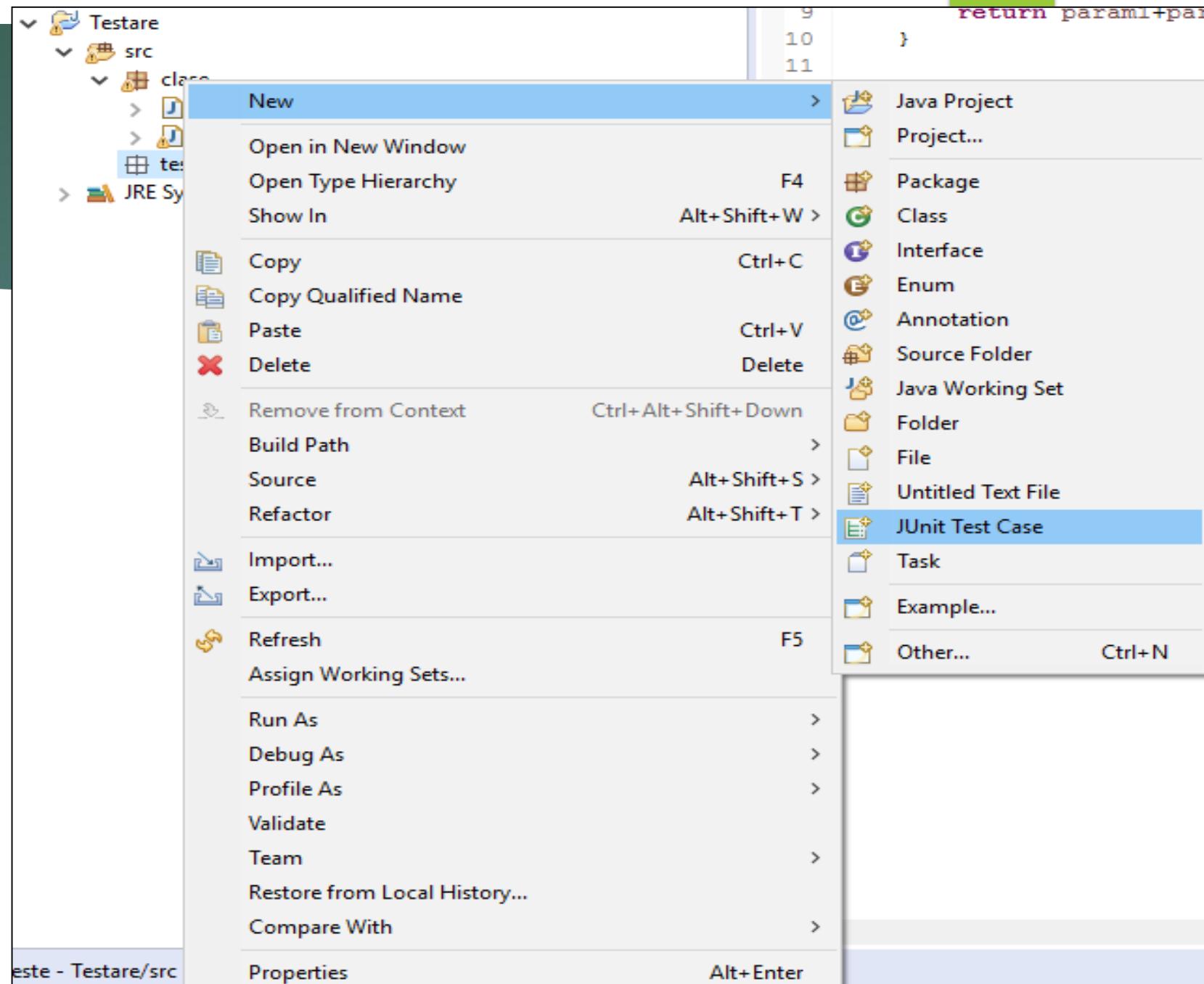
Assert.IsNull

Assert.IsNotNull

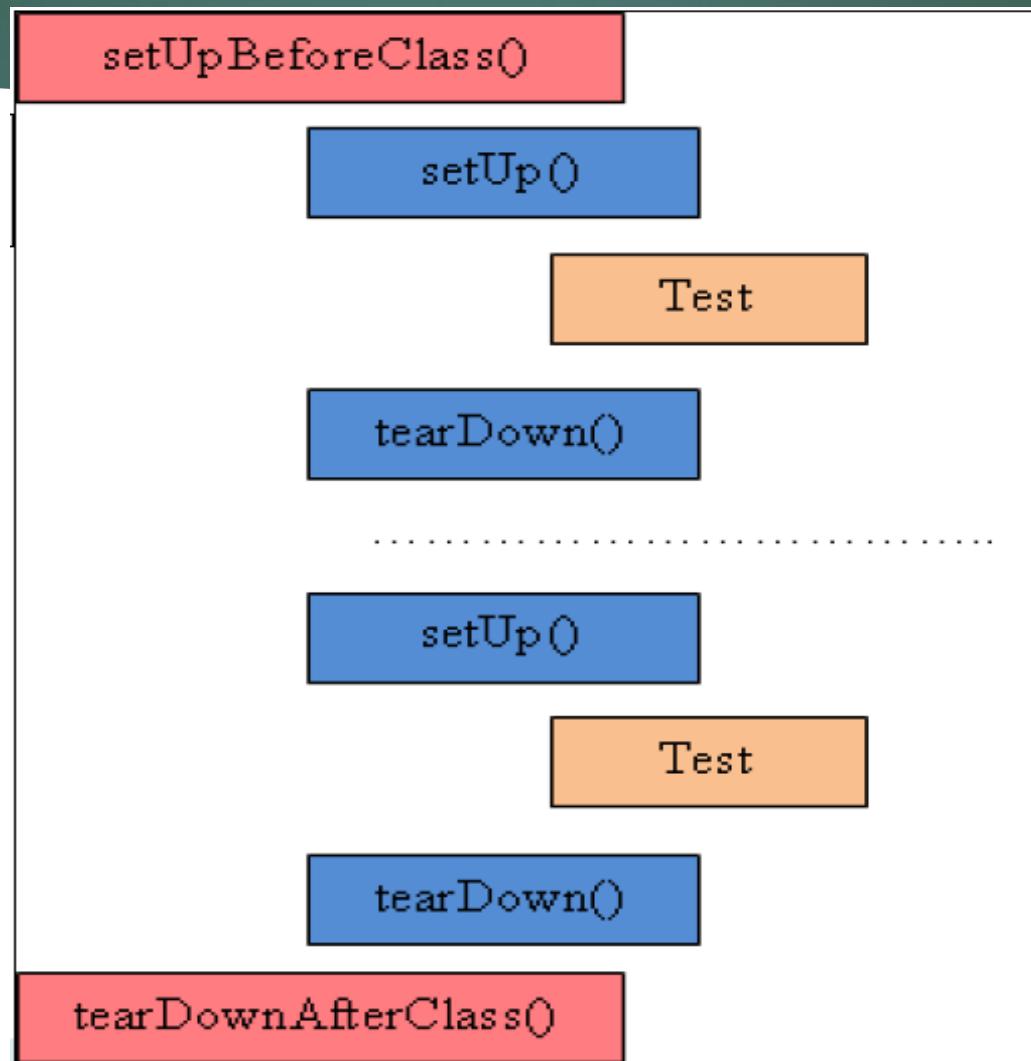
Assert.IsTrue

Assert.IsFalse

# Test Case



# Unit testing - Junit skeleton



# Unit testing - Junit skeleton

Which method stubs would you like to create?

- setUpBeforeClass()     tearDownAfterClass()
- setUp()                 tearDown()
- constructor

setUpBeforeClass

setUp

Test1

tearDown

setUp

Test2

tearDown

setUp

Test3

tearDown

tearDownBeforeClass

# JUnit3 și JUnit4 – diferențe

- ▶ JUnit 3 are nevoie de o versiune de JDK mai nouă decât JDK 1.2 în timp ce JUnit 4 are nevoie de o versiune mai nouă decât JDK 5;
- ▶ în JUnit 3 clasele de test trebuie să fie derivate din clasa TestCase, iar în JUnit 4 nu este necesară moștenirea clasei TestCase;
- ▶ în JUnit3 numele metodelor de test este construit după formatul *testAAA*; astfel toate metodele de test conțin în numele acestora cuvântul *test* în JUnit4 numele metodele nu este important; însă metodele care sunt rulate ca teste au adnotarea *@Test*;
- ▶ în JUnit 4 este folosită adnotarea *timeout* pentru verificarea finalizării testului într-un interval de timp precizat: *@Test(timeout=1000)*; valoarea pentru timeout este setată în milisecunde;
- ▶ pentru ca un test să nu fie rulat în JUnit3 trebuie să fie comentat sau modificat numele, astfel încât să nu respecte formatul *testAAA*; în JUnit 4 testul care nu se dorește a fi rulat primește adnotarea *@Ignore* sau se sterge adnotarea *@Test*.

# Unit testing - Junit skeleton

- ▶ Adnotările utilizate în JUnit4 pentru metodele automate din skeleton:
  - ▶ @BeforeClass pentru metoda `setUpBeforeClass()`;
  - ▶ @AfterClass pentru metoda `tearDownAfterClass()`;
  - ▶ @Before pentru metoda `setUp()`;
  - ▶ @After pentru metoda `tearDown()`.
- ▶ În JUnit3 neexistând adnotări numele metodelor `setUp()`, respectiv, `tearDown()` sunt obligatorii.

# JUnit5

- ▶ Adnotările pentru structura unui test s-au schimbat

JUnit4	JUnit5 - Jupiter
<b>@BeforeClass</b>	<b>@BeforeAll</b>
<b>@AfterClass</b>	<b>@AfterAll</b>
<b>@Before</b>	<b>@BeforeEach</b>
<b>@After</b>	<b>@AfterEach</b>

# What next?

- ▶ What to test?



# Unit Testing: Right-BICEP

ALIN ZAMFIROIU

# Recapitulare

- ▶ Testing
- ▶ Unit testing
- ▶ Junit
- ▶ Test
- ▶ TestCase
- ▶ TestSuite
- ▶ Assertion
- ▶ Skeleton

# What to test?



**What could possibly  
go wrong?**

# Right-BICEP

- ▶ Acum știm cum să facem teste, mai trebuie să știm ce trebuie să testăm.
- ▶ Persoanele cu experiență știu direct ce să testeze, din propria experiență.
- ▶ Persoanele cu mai puțină experiență au nevoie de direcții de urmat sau de principii care să îi îndrumă în realizarea acestor teste.
- ▶ Cel mai cunoscut principiu de testare este Right-BICEP

# Right-BICEP

- ▶ **R**IGHT – dacă rezultatele furnizate de către metodă sunt corecte;
- ▶ **B** – trebuie verificate toate limitele (**Boundary**) și dacă în cazul acestor limite rezultatele furnizate de metoda testată sunt de asemenea corecte;
- ▶ **I** – trebuie verificate relațiile inverse (**Inverse**);

# Right-BICEP

- ▶ **C** – trebuie verificată corectitudinea printr-o verificare încrucișată (**Cross-Check**), folosind metode de calcul asemănătoare, testate și validate de către o comunitate mare de programatori;
- ▶ **E** – trebuie simulată și forțată obținerea erorilor (**Errors**) pentru verificarea comportamentului metodei în cazul anumitor erori;
- ▶ **P** – trebuie verificată păstrarea performanței (**Performance**) între limitele acceptanței pentru produsul software final.

# Unit testing - Right

- ▶ De fiecare dată când testăm o metodă, primul lucru care ar trebui verificat este că această metodă oferă rezultatele corecte.
- ▶ De aceea prima direcție este de a verifica corectitudinea rezultatelor.
- ▶ Această verificare se face în conformitate cu specificațiile proiectului dezvoltat.

# Unit testing - Boundary

- ▶ Problemele apar de obicei la “margini”, deci trebuie să fim atenți să testăm metodele pentru limitele intervalelor.
- ▶ Pentru fiecare metodă, trebuie să determinăm intervalul în care pot fi valorile parametrilor de intrare, precum și intervalul de rezultate furnizat de metodă.

# Unit testing - Boundary

- ▶ Odată ce aceste limite au fost determinate, se efectuează teste exact pentru aceste valori.
- ▶ Testele Boundary nu presupun testarea valorilor din afara acestor valori, ci verificarea corectitudinii acestor valori - valori limită.
- ▶ Există de obicei limite inferioare și limite superioare. Testele se fac pentru ambele situații.

# Unit testing - Boundary

- ▶ Pentru a identifica mai ușor limitele extreme, putem să utilizăm principiul CORECT :
  - ▶ C – Conformance;
  - ▶ O – Ordering;
  - ▶ R – Range;
  - ▶ R – References;
  - ▶ E – Existence;
  - ▶ C – Cardinality;
  - ▶ T – Time.

# Unit testing - Inverse relationship

- ▶ Anumite metode pot fi testate prin aplicarea regulii inverse: pornind de la rezultat, trebuie să se ajungă la aceeași intrare de la care a început inițial.
- ▶ Nu se aplică pentru toate metodele. De obicei se aplică metodelor matematice.
- ▶ De asemenea, pentru bazele de date se poate verifica dacă a fost efectuată o inserare prin operația inversă: select.

# Unit testing - Cross-Check

- ▶ Pentru fiecare metodă, putem încerca să o testăm utilizând altă metodă.
- ▶ De obicei, există mai multe modalități de a rezolva o problemă. Astfel, se poate utiliza o altă metodă pentru rezolvarea problemei pentru verificarea / testarea metodei nou implementate.

# Unit testing - Cross-Check

- ▶ Această situație este posibilă atunci când metoda implementată a fost concepută pentru a crește productivitatea sau dacă metoda veche consumă prea multe resurse.
- ▶ Testarea metodei noi se face prin metoda veche, chiar dacă aceasta consumă mai multe resurse.

# Unit testing - Error conditions

- ▶ Probabil cel mai urât scenariu pentru o aplicație este să crape. De aceea, atunci când testăm fiecare metodă unitar, trebuie să testăm și situațiile în care aplicația ar putea să crape.
- ▶ Dacă am studiat limitele extreme pentru valorile de intrare sau valori rezultate, testarea pentru furnizarea erorilor ar trebui să utilizeze valori în afara acestor intervale.

# Unit testing - Error conditions

- ▶ Testarea de forțare a erorilor se face pentru toate metodele. Toate metodele au cel puțin o situație în care vor oferi erori. Testarea se face pentru aceste situații și se verifică dacă metoda tratează acel caz și aruncă sau oferă o excepție.

# Unit testing - Performance

- ▶ Pentru diferite metode, este posibil să se testeze cât de bine funcționează metoda respectivă.
- ▶ Pe lângă testarea corectitudinii rezultatelor metodelor, este foarte important să se verifice performanța procesării.

# Unit testing - Performance

- ▶ Verificarea performanței se face atât din punctul de vedere al resurselor consumate, cât și din punctul de vedere al timpului necesar pentru obținerea rezultatelor.
- ▶ Testarea performanței este efectuată atunci când input-ul sau rezultatul unei metode este reprezentat de o listă sau de un număr foarte mare de elemente, iar aceste valori pot crește foarte mult.
- ▶ Pentru JUnit4 pentru a testa timpul în care rulează o anumită metodă, este folosită următoarea adnotare : `@Test(timeout=100)`

# CORRECT

- ▶ **C** – Conformitatea formatului (**Conformance**);
- ▶ **O** – Ordinea (**Order**);
- ▶ **R** – Intervalul (**Range**);
- ▶ **R** – Referințe externe (**References**);
- ▶ **E** – Existența obiectelor sau a rezultatelor (**Existence**);
- ▶ **C** – Cardinalitatea rezultatelor (**Cardinality**)
- ▶ **T** – Timpul (**Time**).

# CORRECT

- ▶ Fiecare sub-principiu are o întrebare care ar trebui să fie în mintea testerului.
- ▶ Acest principiu este folosit și pentru a stabili condițiile limită pentru testele de Boundary din Right-BICEP.

# Conformance

- ▶ Este, de asemenea, cunoscut sub numele de:
  - ▶ Type testing
  - ▶ Compliance testing
  - ▶ Conformity assessment



# Conformance

- ▶ Se aplică în numeroase domenii în care ceva ar trebui să îndeplinească anumite standarde specifice.
- ▶ De obicei, pentru orice intrare și pentru orice ieșire, trebuie să se verifice conformitatea cu un format sau cu un standard.

# Conformance

- ▶ Testele pot fi efectuate pentru a verifica ce se întâmplă dacă datele de intrare nu sunt conforme cu formatul sau pentru a vedea dacă rezultatul obținut este conform cu formatul specific proiectului.

# Ordering

- ▶ Testele de ordine sunt specifice listelor, dar nu numai.
- ▶ În cazul listelor, trebuie să verificăm dacă ordinea articolelor este cea dorită.
- ▶ De asemenea, putem testa comportamentul metodei dacă primește anumiți parametri într-o altă ordine sau o listă de elemente într-o ordine diferită de cea așteptată.

# Range

- ▶ Pentru valorile de **intrare** și de **ieșire**, sunt setate anumite intervale. Aceste intervale trebuie verificate.
- ▶ Pentru anumite metode sunt stabilite mai multe intervale. Acest lucru va fi testat pentru toate aceste intervale.

# Range

- ▶ Toate funcțiile care au un index trebuie să fie testate pentru interval, deoarece acel index are un domeniu bine stabilit.
- ▶ De obicei, este necesar să verificați :
  - ▶ Valorile initiale și finale pentru index au **aceeași valoare**;
  - ▶ **Primul** element este mai mare sau mai mic decât **ultimul** element;
  - ▶ Ce se întâmplă dacă indicele este **negativ**;
  - ▶ Ce se întâmplă dacă indicele este mai mare decât **limita superioară**;
  - ▶ Numărul de articole nu este același cu cel pe care îl doriți - **dimensiunea**;
  - ▶ etc.

# Reference

- ▶ Anumite metode depind de lucrurile externe sau de obiectele externe acestor metode. Aceste elemente trebuie verificate și controlate.
- ▶ Exemple:
  - ▶ O aplicație web necesită **conectarea** utilizatorului;
  - ▶ O **extragere** din stivă funcționează dacă există elemente în stivă;
  - ▶ etc.

# Reference

- ▶ Aceste elemente sunt numite **precondiții** sau **condiții preliminare**.
- ▶ Condiții preliminare pentru ca metoda să funcționeze în mod normal.
- ▶ Aceste teste sunt efectuate folosind dubluri de test (stub, fake, dummy, mock).

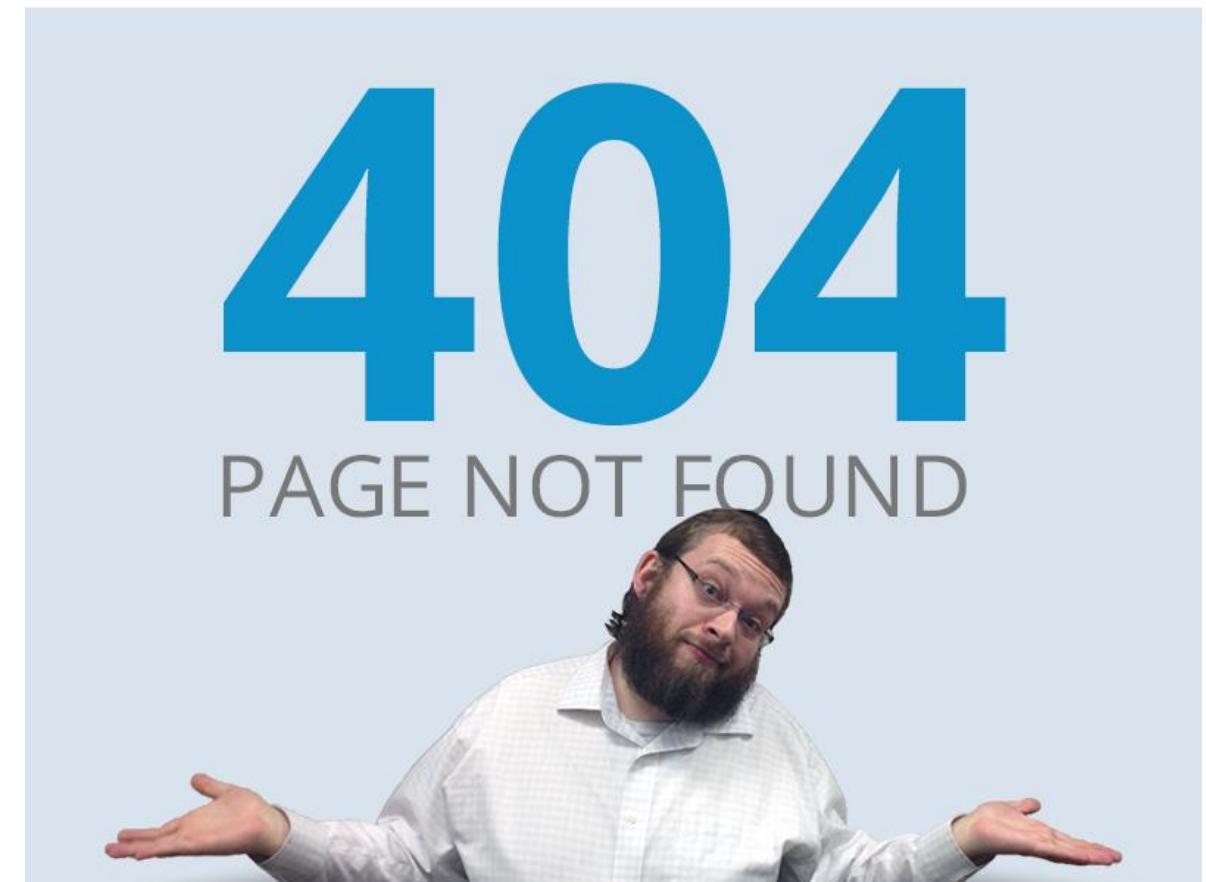
# Existence

► “**does some given thing exist?**”

- ▶ Trebuie să ne întrebăm ce se întâmplă cu metoda dacă un parametru nu există, dacă este nul sau dacă este 0.
- ▶ De asemenea, pentru sistemele software care funcționează cu fișiere sau cu conexiune la internet, este necesar să se verifice existența acestor fișiere sau disponibilitatea conexiunii la internet. În caz contrar, aplicația nu trebuie să dea eroare, ci trebuie să se comporte normal cu avertizarea utilizatorului de problema întâmpinată.

# Existence

- ▶ Make sure your method can stand up to nothing.
- ▶ Este asemănătoare cu condiția de eroare din **Right-BICEP**.



# Cardinality

## ► 0-1-n Rule

- ▶ Este similar cu testele de existență (**Existence**) și testele privind intervalul (**Range**).
- ▶ Trebuie să verificăm dacă metoda/lista/colecția are 0 elemente, 1 element sau elemente n.
- ▶ Dacă funcționează pentru 2, 3 sau 4 elemente, se consideră că va funcționa pentru mai multe elemente, însă nu trebuie să uităm de testul de **Boundary** superior.

# Time

- ▶ Este similar cu testul de performanță din Right-BICEP.
- ▶ De asemenea, poate fi testat dacă şablonul de apeluri este respectat.  
Similar cu design pattern-ul **Template**.
- ▶ De exemplu, pentru a apela metoda *logout()*, trebuie mai întâi să apelăm metoda de conectare().

# CORRECT - Questions

- ▶ **C** – Conformitatea formatului (**Conformance**);
- ▶ **O** – Ordinea (**Order**);
- ▶ **R** – Intervalul (**Range**);
- ▶ **R** – Referințe externe (**References**);
- ▶ **E** – Existența obiectelor sau a rezultatelor (**Existence**);
- ▶ **C** – Cardinalitatea rezultatelor (**Cardinality**)
- ▶ **T** – Timpul (**Time**).

# F.I.R.S.T

- ▶ “ Pentru ca testele unitare să fie utile și eficiente pentru echipa de programare, trebuie să vă amintiți să le faceți FIRST.”



# F.I.R.S.T

- ▶ **Fast**
- ▶ **Isolated/Independent**
- ▶ **Repeatable**
- ▶ **Self-Validating**
- ▶ **Timely**

# Fast

- ▶ Testul dezvoltat ar trebui să fie rapid, deoarece dacă avem prea multe teste, nu trebuie să aşteptăm prea mult timp când le executăm.



# Isolated

- ▶ **Single responsibility (SOLID)**
- ▶ “Each unit test should have a single reason to fail.”

# Isolated

- ▶ Atunci când un test eșuează, dezvoltatorul nu trebuie să facă debug pentru a identifica ce este greșit și unde este problema.
- ▶ Testul ar trebui să fie izolat și să spună exact unde este problema și ce problemă există.

# Repeatable

- ▶ Rezultatele obținute ar trebui să fie identice indiferent de numărul rulări ale acestor teste.
- ▶ Testele ar trebui să se desfășoare în mod repetat, fără alte intervenții.

# Self-Validating

- ▶ Încrederea în testele implementate.
- ▶ În cazul în care testele trec, dezvoltatorul ar trebui să aibă mare încredere că codul este corect și fără erori.
- ▶ Dacă un test nu reușește să treacă, dezvoltatorul trebuie să aibă încredere în faptul că metoda trebuie îmbunătățită ci nu să considere ca testul este greșit.

# Timely

- ▶ Când trebuie să punem în aplicare testele pentru metoda noastră?
- ▶ Când considerăm că am făcut toate testele?

# EclEmma – Code Coverage

**EclEmma Java Code Coverage 2.3.3**



EclEmma is a free Java code coverage tool for Eclipse, available under the Eclipse Public License. It brings code coverage analysis directly into the Eclipse...

[more info](#)

by [Mountainminds GmbH & Co. KG](#), [EPL](#)  
[quality metrics](#) [code coverage](#) [fileExtension](#) [exec](#)

 380     Installs: 516K (10,346 last month)    Installed

# EclEmma – Code Coverage

Element	Coverage	Covered Instructions	Missed Instructions	Total Instructions
➤ Seminar11Testare	 83.6 %	353	69	422
➤ src	 83.6 %	353	69	422
➤ testing	 89.8 %	237	27	264
➤ clase	 82.3 %	116	25	141
➤ Persoana.java	 81.8 %	90	20	110
➤ PersoanaMock.java	 62.5 %	5	3	8
➤ Familie.java	 91.3 %	21	2	23
➤ program	 0.0 %	0	17	17

# DUBLURI DE TESTARE

ALIN ZAMFIROIU

# Recapitulare

- ▶ Testing
- ▶ Unit testing
- ▶ Junit
- ▶ Test
- ▶ TestCase
- ▶ TestSuite
- ▶ Assertion
- ▶ Right-BICEP
- ▶ CORRECT

# DUBLURI DE TESTARE

- ▶ În testarea automată este obișnuită folosirea obiectelor care arată și se comportă ca echivalentele lor de producție, dar sunt de fapt simplificate. Acest lucru reduce complexitatea, permite verificarea codului independent de restul sistemului și, uneori, este chiar necesar să se efectueze teste de auto-validation. Un termen generic folosit pentru aceste obiecte este **dublură de testare**.
- ▶ O dublură de testare este pur și simplu un alt obiect care se potrivește cu interfața colaboratorului necesar și poate fi trecut în locul său. Există mai multe tipuri de dubluri de testare.

# DUBLURI DE TESTARE

- ▶ **Dummy object** – un obiect care respectă interfața dar metodele nu fac nimic sau null.
- ▶ **Stub** – Spre deosebire de Dummies, metodele dintr-un Stub vor întoarce răspunsuri conserve / hardcodate.
- ▶ **Spy** – este un Stub care gestionează și contorizează numărul de apeluri.
- ▶ **Fake** – este un obiect care se comportă asemănător cu unul real, dar are o versiune simplificată.
- ▶ **Mock** – diferit de toate celelalte.

# Obiecte Dummy

- ▶ **Dummy object** – un obiect care respectă interfața, dar metodele nu fac nimic sau returnează 0 sau null.
- ▶ Când trebuie să folosim obiectul real, de fapt folosim un obiect dummy.
- ▶ Aceste duble sunt folosite atunci când nu trebuie să apelăm metodele din acel obiect. Pentru că nu fac nimic.

# Obiecte Dummy

```
@Test
public void test() {
    Companie company=new Companie("Company", new PersoanaDummy(), 0);

    List<IPersoana> lista=new ArrayList<>();
    lista.add(new PersoanaDummy());
    lista.add(new PersoanaDummy());
    lista.add(new PersoanaDummy());
    company.setSalariati(lista);

    assertEquals(3, company.getNumarSlariati());
}
```

# Stub

- ▶ **Stub** – metodele de la un Stub vor returna răspunsuri conserveate / hardcodate.

```
public int getVarsta() {  
    return 33;  
}
```

- ▶ În acest fel, putem folosi aceste obiecte cu apeluri reale.

# Stub

```
@Test  
public void test_verificareLegalitate() {  
    IPersoana persoana=new PersoanaStub("Nume Prenume", "43");  
    Companie c=new Companie("Companie", persoana, 1000);  
    assertTrue(c.verificareLegalitate());  
}
```

# Fake

- ▶ **Fake** – este un obiect care se comportă ca unul real, dar are o versiune simplificată.

```
@Override  
public int getVarsta() {  
    return valoareGetVarsta;  
}
```

- ▶ De obicei, pentru un fake, putem stabili ce valoare ar trebui să se întoarcă. Nu va fi o valoare hardocdată.

# Spy

- ▶ **Spy** – este un Stub sau Fake care gestionează și numărul de apeluri realizate pentru metodele acestor obiecte.

```
public int getVarsta() {  
    numberGetVarsta++;  
    return 33;  
}
```

```
public int getVarsta() {  
    numberGetVarsta++;  
    return valoareGetVarsta;  
}
```

# Mock

- ▶ **Mock** – diferit de toate celealte, dar funcționează similar.

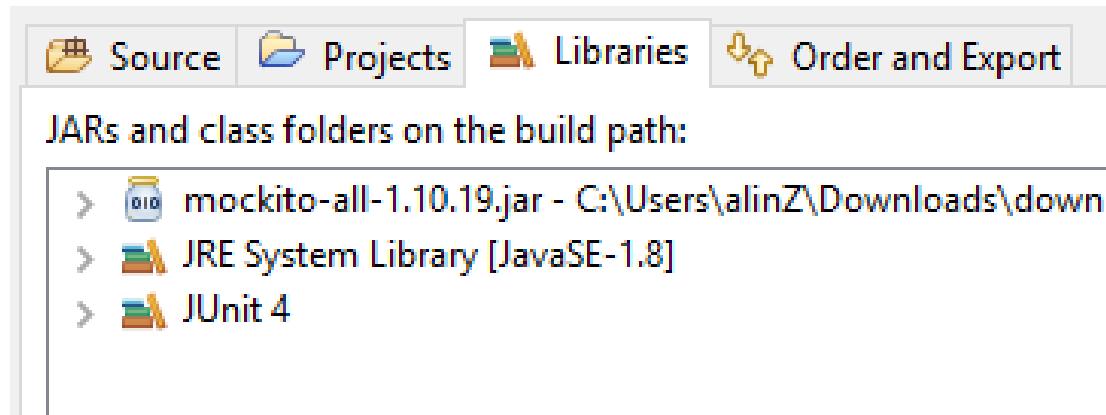


# Mock testing

- ▶ Mock testing – Este utilizat atunci când dorim ca metoda testată să nu fie influențată de referințe externe.
- ▶ Mock object (obiect mock-uit) – un obiect care simulează comportamentul unui obiect real, însă într-un mod controlat.
- ▶ Cele mai folosite framework-uri sunt **Mockito**, **EasyMock**, etc

# Mockito

- ▶ Se descarcă jar-ul și se adaugă la proiectul curent.



- ▶ Dacă nu îl găsiți pe google:

<https://mvnrepository.com/artifact/org.mockito/mockito-all/1.10.19>

# Mockito

- ▶ Se creează un obiect mock, pe baza unei clase reale:

```
Persoana sot=mock(Persoana.class);
```

- ▶ Se setează comportamentul pentru metodele dorite:

```
when(sot.getVarsta()).thenReturn(23);
```

```
doReturn(3).when(sot).getVarsta();
```

# Mockito

- ▶ Metode:
  - ▶ doReturn()
  - ▶ doAnswer()
  - ▶ when()
  - ▶ thenReturn()
  - ▶ thenAnswer()
  - ▶ thenThrow()
  - ▶ doThrow()
  - ▶ doCallRealMethod()

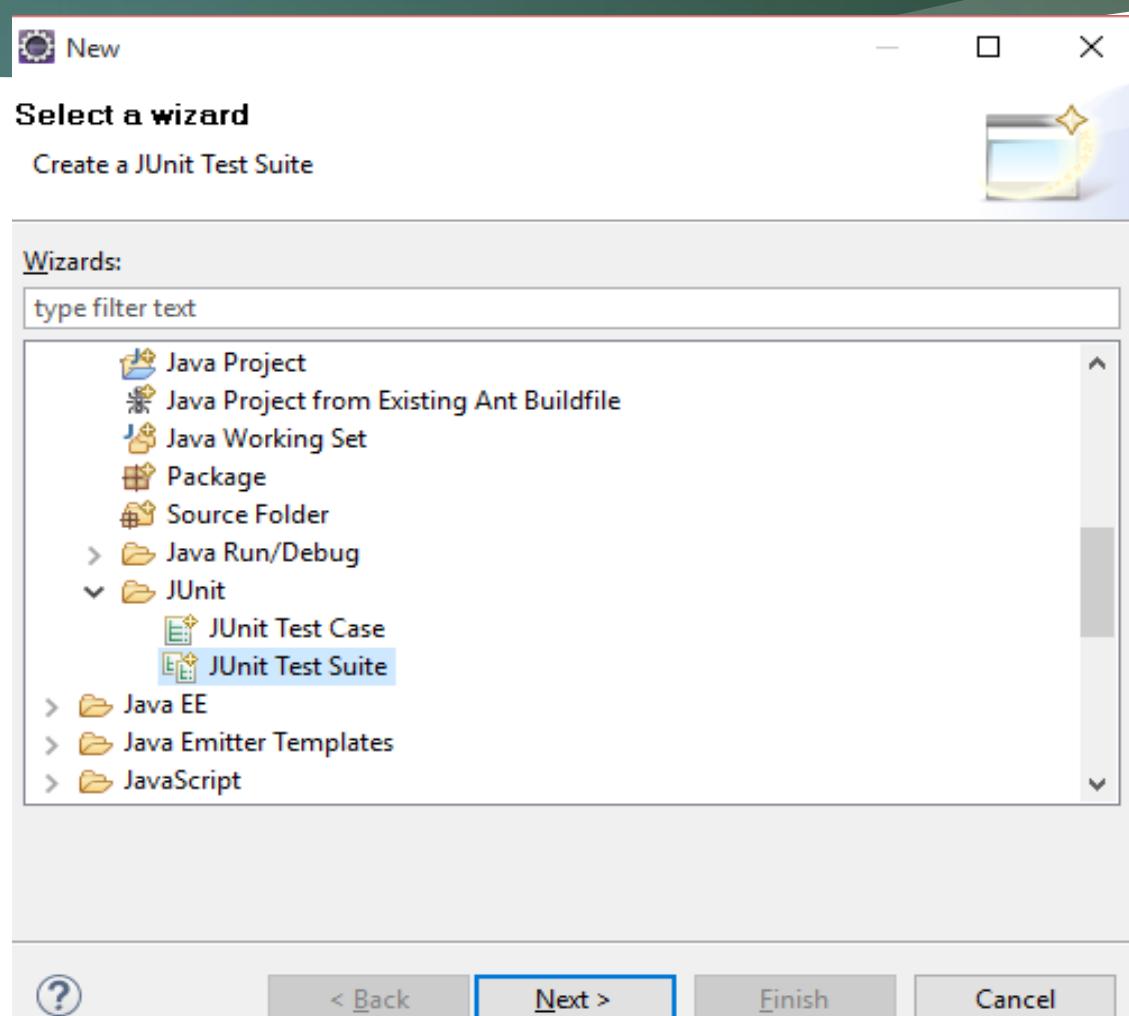
# Mockito

```
@Test  
public void test_verificareLegalitate_mockito() {  
    Persoana p=mock(Persoana.class);  
    when(p.getVarsta()).thenReturn(21);  
    Companie c=new Companie("Companie", p, 1000);  
    assertTrue(c.verificareLegalitate());  
}
```

# Utilizarea fișierelor pentru datele de test

- ▶ Se va realiza un singur TestCase cu metoda de assert apelată într-un loop.
- ▶ Pentru fiecare set de date de test din fișier se apelează metoda assert.
- ▶ Fișierul, sau fluxul este considerat un fixture și de aceea trebuie deschis în setUp și închis în tearDown.

# Unit testing – TestSuite general



# Custom TestSuite – JUnit 3

- ▶ TestCase-ul creat trebuie să extindă clasa TestCase:
  - ▶ 

```
public class UtilsTest extends TestCase{
```
- ▶ Se implementează constructorul cu un parametru de tipul String:

```
public UtilsTest(String method)  
{  
    super(method);  
}
```

# Custom TestSuite – JUnit 3

```
public static void main(String[] args) {  
    TestSuite suite=new TestSuite();  
    suite.addTest(new UtileTest("test_correct_division"));  
    suite.addTest(new UtileTest("test_correct_sum"));  
    TestRunner.run(suite);  
}
```

# Custom TestSuite JUnit 4

- ▶ Se implementează o clasă sau o interfață CustomSuite.
- ▶ Pentru fiecare test dorit să facă parte din acea suită se adaugă anotarea:
  - ▶ `@Category(CustomSuite.class)`

# Custom TestSuite – JUnit 4

- ▶ Pentru crearea unei suite custom , sunt incluse toate categoriile dorite.
- ▶ În cazul de față se include o singură categorie de teste. Testele din această categorie se regăsesc în două TestCase-uri **CompanyTest** și **PersonTest**:

```
@RunWith(Categories.class)
@includeCategory(CustomSuite.class)
@SuiteClasses({ CompanyTest.class, PersonTest.class })
public class NewSuiteTests {
}
```

# JUnit5



# JUnit5

- ▶ Parametrul optional este pe ultima pozitie.

```
@Test  
@Ignore  
public void testCuMesaj() {  
    Persoana persoanal = new Persoana("Nume Prenume", "1900807381167");  
    assertTrue(persoanal.checkCNP(), "CNP incorrect");  
}
```

# JUnit5

- ▶ Adnotările pentru structura unui test s-au schimbat

JUnit4	JUnit5 - Jupiter
<b>@BeforeClass</b>	<b>@BeforeAll</b>
<b>@AfterClass</b>	<b>@AfterAll</b>
<b>@Before</b>	<b>@BeforeEach</b>
<b>@After</b>	<b>@AfterEach</b>

# JUnit5

- ▶ **assertThrows** – pentru testarea condițiilor de eroare.

```
@Test
public void test_checkCNP_conditii_de_eroare() {
    Persoana persoanal = new Persoana("Nume Prenume", "2iili13144434");
    assertThrows(Exception.class, new Executable() {
        @Override
        public void execute() throws Throwable {
            assertTrue(persoanal.checkCNP());
        }
    });
}
```

# JUnit5

- ▶ assertTimeout – pentru testarea timpului de rulare

```
@Test
public void test_performanta() {
    Persoana persoana = new Persoana("Nume Prenume", "2971023404186");
    assertTimeout(Duration.ofMillis(10), new Executable() {
        @Override
        public void execute() throws Throwable {
            assertEquals("F", persoana.getSex());
        }
    });
}
```

# JUnit5

- ▶ @Tag – pentru suitele customize

```
@Test  
@Tag("Performance")  
@Tag("Fast")
```

```
@Test  
@Tag("Error")  
@Tag("Slow")
```

# Integration tests and Unit tests



via reddit.com/r/programmerhumor

# Integration tests and Unit tests



# The end!

