Introducere în programare
2016 - 2017
6

http://profs.info.uaic.ro/~introp/

Cuprins

- Clean Code
- Variabile
- Functii
- Comentarii
- Formatare

Clean Code

- There will be code
  - **Nivelul de abstractizare** a codului va creste dar codul in sine nu va fi eliminat.
  - Va fi nevoie in continuare de **rigurozitate, precizie, formalitate si detaliere** in cod pentru a putea fi inteles si executat de masina (de calcul).
  - Intr-un final codul este de fapt limbajul prin care exprimam anumite necesitati/cerinte. Vor fi limbaje mai bune, utilitare pentru a forma cerintele in structuri de date cat mai formale, dar **nu scapam de precizie**.

Clean Code

- Bad code
  - Orice programator cu experienta a intalnit **“bad code”**. Prin care incerci sa navighezi, sa il intelegi, speri sa descoperi un indiciu dar tot ce gasesti e mai multa confuzie
  - Cel care l-a scris era grabit? Probabil…
  - Toti ne-am gasit in situatia in care am scris “bad code” dar din necesitate am decis ca “o dezordine functionala e mai bine decat nimic”. Si “o sa il curat mai tarziu”
  - LeBlanc’s law: ***Later equals never***

Clean Code

- Costul dezordinei
  - Orice “bad code” ajunge sa **incetineasca** procesul la un moment dat
  - Orice schimbare a codului duce la alte 2 probleme..
  - In timp ce problemele cresc productivitatea echipei incepe sa scada, asimptotic apropiindu-se de 0.

- Problema principala
  - Toti programatorii experimentati stiu ca “bad code” incetineste tot procesul dar in acelasi timp sunt presati sa scrie rau pentru a termina mai repede.
  - Pe scurt. Nu isi aloca timp pentru a putea merge repede
  - Adevaratii profesionisti stiu ca afirmatia este gresita. *Singurul* mod de a termina la timp este sa pastrezi codul cat mai curat posibil tot timpul.

Ce este Clean Code?

Sunt nenumarate definitii, fiecare programator isi dezvolta propria definitie

Bjarne Stroustrup, inventor of C++ and author of The C++ Programming Language "I like my code to be **elegant** and **efficient**. The logic should be **straightforward** to make it hard for bugs to hide, the dependencies minimal to **ease maintenance**, error handling complete according to an articulated strategy, and performance close to optimal so as not to tempt people to make the code messy with unprincipled optimizations. **Clean code does one thing well.**"

Grady Booch, author of Object Oriented Analysis and Design with Applications "Clean code is simple and direct. **Clean code reads like well-written prose.** Clean code never obscures the designer's intent but rather is full of crisp abstractions and straightforward lines of control."

"Big" Dave Thomas, founder of OTI, godfather of the Eclipse strategy "Clean code **can be read**, and **enhanced** by a developer **other than its original author**. It has unit and acceptance tests. It has meaningful names. It provides one way rather than many ways for doing one thing. It has **minimal dependencies**, which are explicitly defined, and provides a clear and minimal API. Code should be literate since depending on the language, not all necessary information can be expressed clearly in code alone."

Ce este Clean Code?

- Elegant
- Eficient
- Simplu
- Minimal
- La fel de simplu de citit ca o proza bine scrisa
- Alti programatori pot modifica si imbunatati codul usor
- Trateaza cazurile de exceptie
- Fara repetitie
- Expresivitate
- Abstractizari

Variabile

- Folosirea numelor ce sugereaza intentia (use intention-revealing names) si explica: (Exemplu bun CC 49-50)
  - De ce exista
  - Ce face
  - Cum este folosita
- Evita dezinformarea
  - E posibil prin nume sa sugerezi alte aplicatii ale variabilei
  - Scrierea conceptelor similare similar este o informatie, scrierea lor in moduri diferite este dezinformare
- Foloseste cuvinte usor de pronuntat
  - In discutii intre membrii din echipa e mai usor sa discuti despre “dataCurenta” decat “dCur” (va rog gasiti un exemplu mai bun)

Variabile

- Nume ce pot fi cautate usor
  - “MaxStudentClasses” este mai usor de cautat in 100k linii de cod decat “e”
- Evita memorarea variabilelor
  - Cei ce iti citesc codul nu ar trebui sa organizeze in mintea lor toate variabilele si intelesul lor pentru a-ti intelege codul
- Don’t be cute
  - Glumele se uita si daca denumesti o variabila “HolyHandGrenade” in loc de “DeleteItems” …
- Alege un cuvant pentru un concept
- Nu alege acelasi cuvant pentru doua cazuri diferite
- Adauga context numelor
- Nume ce definesc domeniul utilizarii
  - Cei ce citesc/folosesc codul tau sunt tot programatori. Mai bine folosesti “CoadaUtilizatori” decat “ListaUtilizatori” daca e folosita ca o coada

Variabile

- Problema alegerii unui nume bun este necesitatea unui fundal cultural comun
- Nu trebuie sa iti fie frica sa redenumesti variabila daca imbunatatesti numele
- Tine minte ca numele valoreaza mai mult pentru cititori decat pentru cel ce scrie codul. Citeste codul tau vechi de 6 luni si incearca sa intelegi unde trebuie sa ai grija pe viitor

Variabile - conventii de nume - de ce?

- Printr-o singura decizie globala in loc de multe locale ai mai mult timp de concentrare asupra altor parti mai importante

- Prin similaritatile dintre nume poti intelege mai usor ce reprezinta variabilele in diferite proiecte

- Ajuta la intelegerea rapida a codului. In loc sa inveti stilurile proprii de codat ale lui Ionel, Ana si Maria, poti lucra cu un cod cu denumiri consistente.

- Ajuta acolo unde limbajul are lipsuri. Poti folosi conventii pentru a simula diferite constante sau tipur lipsa de date

Variabile - cum sa nu

- Evita abrevieri sau nume derutante
- Evita nume cu intelesuri similare
- Evita variabile cu intelesuri diferite dar nume similare
- Evita nume ce suna similar (wrap & rap)
- Evita numere in numele variabilelor
- Evita cuvintele scrise gresit
- Evite cuvintele complicate ce pot fi scrise gresit usor
- Evita multiple limbi
- Evita nume similare cu numele rezervate
- Nu folosi nume ce au cu totul alt inteles fata de ce reprezinta variabila
- Evita nume ce contin caractere greu de citit

Variabile - Checklist

- Numele descrie in totalitate si cu acuratete ce reprezinta variabila?
- Numele este clar si nu trebuie sa ghicesti ce inseamna?
- Numele foloseste *Count* sau *Index* in loc de *Num*?
- Variabilele booleene au fost denumite clar pentru a fi evident cand au valorile true si false?
- Conventia pe care o folosesti este pe cat se poate de compatibila cu cea oficiala a limbajului?
- Numele sunt usor de citit?
- Folosesti abrevieri doar in cazul in care sunt clare?
- Numele sunt usor de pronuntat?
- Numele ce ar putea fi pronuntate sau citite gresit sunt evitate?

Variabile - Checklist - Ai evitat...

- … nume ce sunt vagi?
- … nume cu intelesuri similare?
- … nume ce sunt diferite doar printr-un singur caracter?
- … nume ce suna la fel?
- … nume ce folosesc cifre?
- … nume intentionat scrise gresit doat pentru a fi scurtate?
- … nume ce in general sunt scrise gresit in engleza?
- … nume ce sunt deja folosite in librariile standard?
- … nume total arbitrare?
- … caractere greu de citit?

Functii - de ce

- Reducerea complexitatii
  - O creem pentru a ascunde informatia si a nu ne mai gandi la cum e implementata ci doar ce face
  - Minimizarea codului
  - Modificari ulterioare mai usoare
  - Imbunatatirea corectitudinii
- Introducerea unui termen inteligibil ce descrie codul
  - Nume ce descrie clar ce face codul din implementarea functiei
- Evitarea codului duplicat

Functii - de ce

- Simplificarea structurilor conditionale complexe
  - In unele cazuri avem if-uri multiple sau complicate, e mai bine sa stim prin numele functiei ce fac decat sa incercam sa intelegem expresiile

- Imbunatatirea performantei
  - Optimizarea unei functii poate imbunatati performanta intregului sistem.

- Imbunatatirea portabilitatii
  - Folosirea functiilor izoleaza codul non-portabil.

Functii - operatii ce par prea simple pentru a alcatui o functie

- Unul dintre cele mai mari blocaje mintale in a crea o functie este rezistenta de a crea o functie pentru un scop prea simplu. Construirea unei functii pentru doar 3 linii de cod poate parea prea mult, dar experienta arata cat de utilitara poate fi o functie mica buna

Pseudocode Example of a Calculation

```
points = deviceUnits * ( POINTS_PER_INCH / DeviceUnitsPerInch() )
```

Pseudocode Example of a Calculation Converted to a Function

```
Function DeviceUnitsToPoints ( deviceUnits Integer ): Integer
   DeviceUnitsToPoints = deviceUnits *
      ( POINTS_PER_INCH / DeviceUnitsPerInch() )
End Function
```

Functii

Pseudocode Example of a Function Call to a Calculation Function

```
points = DeviceUnitsToPoints( deviceUnits )
```

- Aceasta linie este mult mai lizibila, chiar apropiindu-se de “self-documenting”

Functii - numarul de linii

- Raspunsul in continua dezbatere

- Teoretic, maximul este considerat un ecran de calculator
  - ~ 50-150 linii.
  - IBM a limitat odata functiile la 50 de linii
  - Lungimea medie ~ 10-20linii.

- Programele moderne tind sa aiba foarte multe functii foarte mici cu doar cateva functii mai mari

Functii - lungimea

Studii asupra lungimii functiilor:

- Un studiu (Basili and Perricone 1984) a concluzionat faptul ca numarul erorilor este invers proportional cu dimensiunea unei functii (pana in 200 linii)
- Un studiu din 1986 spune ca functiile mici (<= 32 linii) nu coreleaza cu rata de eroare si costul. Dovezile sugereaza faptul ca functiile mai mari (>65 linii) sunt mai ieftin de implementat
- Alt studiu 1991 a concluzionat ca functiile mici (<143 instructiuni) au cu 23% mai multe erori decat functiile mari dar sunt de 2.4 ori mai ieftine de rezolvat.
- Alt studiu a aflat ca numarul de schimbari necesare a unui cod este cel mai mic atunci cand functiile au intre 100 si 150 linii.

Functii

- O functie pentru **un singur lucru**
  - O functie ar trebui sa faca un singur lucru, ar trebui sa il faca bine, ar trebui sa il faca doar pe el.
- Problema este cum decizi ca face un “singur lucru”
- Iti dai seama ca o functie face mai mult decat un singur lucru atunci cand poti extrage o alta functie din ea.
- Pentru a ne asigura ca functiile noastre fac acelasi lucru trebuie sa ne asiguram ca sunt toate la acelasi nivel de abstractizare.

Functii - nume

- Descriu tot ce face functia
  - ComputeReportTotals() vs ComputeReportTotalsAndOpenOutputFile()
- A se evita verbe fara sens, vagi sau cu multe intelesuri
  - handleCalculation(), OutputUser(), ProcessInput(), DealWithOutput()
- A nu se diferentia functiile doar prin numere
  - Part1(), Part2(), Part3()
- Numele poate fi atat de lung cat este necesar
- Foloseste o descriere a valorii returnate
  - cos(), printer.isReady(), pen.CurrentColor()

Functii - nume

- Foloseste un verb urmat de un substantiv
  - PrintDocument(), CheckOrderInfo(), RepaginateDocument()
- Foloseste antonime precise
  - add/remove, begin/end, open/close, old/new
- Stabileste conventii pentru operatii comune
  - Operatii similare dar metode diferite de apel/utilizare: employee.id.Get(), dependent.GetId(), supervisor(), candidate.id()
- Fii consistent in numele functiilor

Functii - parametrii

- Foloseste ordinea input-modify-output
- Ia in considerare posibilitatea de a-ti crea propriile cuvinte cheie IN si OUT pentru a clarifica ce argumente sunt doar de intrare sau de iesire
- Daca mai multe functii folosesc parametrii similari, scriei intr-o ordine consistenta
- Foloseste toti parametrii functiei
- Pune parametrii pentru status sau eroare ultimii

Functii - parametrii

- Nu folosi parametrii ca si variabile intermediare

- Documenteaza asumptii asupra parametrilor
  - Daca sunt doar input sau output, daca sunt intr-un anumit interval, valori care nu ar trebui sa apara niciodata, unitati de masura

- Numarul parametrilor ar trebui sa fie pana in 4

Functii - Preprocesor

Exemple de baza:

```
// Redenumiri
#define True False
#define False True

// Exemplul 2
#define True rand()

// Exemplul 3
#define TWO_HUNDRED_POINT_1
200.1

// Exemplul 4
#define BASE_NAME “ana.txt”
```

```
// Traduceri
#define folosim using
#define spatiul namespace

folosim spatiul std;

// Traduceri hardcore
#define intreg int
#define inceput main
#define incepe {

intreg inceput()
incepe
...
```

Functii - Preprocesor

Functii de preprocesor

```
// MAX
#define max(a,b)
     (((a) > (b)) ? (a):(b))

// MIN ?

// Conditionali
#define DEBUG_MODE
#ifdef DEBUG_MODE
     cout << "DEBUG ON!";
#endif
```

Exemple diabolice:

```
// 1
#define RETURN(result) return (result);}
int myfunction1(args) {
       int x = 0; // do something
       RETURN(x)

// 2
#define tatdeauna while(true)

// 3
#define repeat while
#define UntilJesusChristReturns 1
Do
{ // CODE
}
repeat(UntilJesusChristReturns);
```

Functii - Check List - Probleme uzuale

- Motivul pentru crearea functiei este suficient?
- Exista parti din functie ce ar putea fi la randul lor functii?
- Numele functiei este puternic, clar, verb+subsantiv si/sau o descriere a valorii intoarse?
- Numele descrie tot ce face functia?
- Ai stabilit conventii pentru operatii comune?
- Functia face un singur lucru si bine?

Comentarii

- Comentariile nu sunt “bune”. In cel mai bun caz comentariile sunt un **rau necesar**.

- Utilizarea potrivita de comentarii este pentru a compensa pentru esecurile noastre in a ne exprima prin cod. Comentariile sunt intotdeauna esec.

- Cand te gasesti intr-o situatie in care este necesar un comentariu, gandeste-te bine si cauta o metoda sa rastorni situatia si sa te exprimi prin cod.

Comentarii

- Comentariile nu compenseaza pentru codul rau
  - Unul dintre cele mai intalnite motive pentru a scrie comentarii este codul rau
  - In loc sa petreci timp scriind comentarii care sa explice dezordinea din cod, petrece timp curatand codul

- Explica folosind cod
  - Exista cu siguranta situatii cand codul este o metoda putin practica de a explica functionalitatea, insa in majoritatea timpului este usor sa te exprimi folosind codul

Comentarii

- Ce ai prefera sa vezi? Asta:

```
// Check to see if the employee is eligible for full benefits
if ((employee.flags & HOURLY_FLAG) &&
     (employee.age > 65))
```

- Sau asta?

```
if (employee.isEligibleForFullBenefits())
```

- Dureaza cateva secunde sa iti explici majoritatea scopului in cod. In multe cazuri este o problema de crearea unei functii care spune acelasi lucru ca si comentariul pe care doreai sa il scrii.

Comentarii

Exemple amuzante:

// 1

```
// I don't know why I need this,

// but it stops the people being upside-down

x = -x;
```

// 2

```
//

// Dear maintainer:

//
```

// 1

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

//Now, God only knows



Exception up = new Exception("Something is really
wrong.");

throw up;  //ha ha
```

// 2

```
// somedev1 -  6/7/02 Adding temporary tracking of
```

Comentarii

- Comentariile bune
  - Unele comentarii sunt benefice sau necesare
  - Singurul comentariu cu adevarat bun este cel ce nu il scrii

- Comentarii rele
  - Majoritatea comentariilor sunt in aceasta categorie
  - Deobicei sunt scuze pentru cod scris prost sau justificare pentru decizii insuficiente

Comentarii

- Regula: Nu folosi comentariu unde poti folosi o functie sau o variabila
- Indicatori de pozitie
  - Cateodata programatorilor le place sa isi marcheze o anumita pozitie in fisierul sursa
  - `// Actiuni ////////////////////////////////////`
  - Sunt rare cazurile cand astfel de grupari/indicatori sunt folositori
- Comentarii pe acoladele de inchidere
  - Cateodata programatorii pun un comentariu special pe acoladele de inchidere
  - Daca te regasesti intr-o astfel de situatie incearca sa scurtezi functia

Comentarii

- Cod comentat `/*int x = 0*/`
  - Putine practici sunt la fel de odioase precum codul comentat
  - Daca trebuie sa scrii un comentariu, asigura-te ca descrie codul din proximitate

- Prototipul functiilor
  - Functiile scurte nu necesita o descriere stufoasa
  - Un nume bine ales pentru o functie scurta ce face un singur lucru este mai bine decat un comentariu pentru functie

Comentarii

- Concluzie
  - Cand simti nevoia sa introduci un nou comentariu este mai bine sa extragi o functie sau sa introduci o variabila

Formatare

- Formatarea verticala
  - Cat de lung ar trebui sa fie fisierul sursa?

- Vertical Openness Between Concepts
  - Aproape orice cod este citit de la stanga la dreapta si de jos in sus
  - Orice linie reprezinta o expresie sau o clauza, iar fiecare grup de linii reprezinta un gand complet.
  - Acele ganduri ar trebuie separate unele de altele prin linii goale.

Formatare

- Distanta pe verticala
  - Conceptele ce sunt intr-o stransa legatura ar trebui sa fie apropiate pe verticala
- Declararea de variabile
  - Variabilele ar trebui sa fie declarate cat se poate de aproape te utilizarea lor
  - Deoarece functiile sunt scurte variabilele locale ar trebui sa fie la inceputul fiecarei functii
- Functii dependente
  - Daca o functie apeleaza alta functie, ele ar trebui sa fie apropiate pe verticala, si functia apelanta sa fie deasupra functiei apelate
  - Asta ii da programului un flux natural

Formatare

- Identare
  - Pentru a face ierarhia domeniului vizibila, indentam liniile de cod in proportie cu pozitia lor in ierarhie
  - Instructiunile de la nivelul fisierului nu sunt indentate
  - Implementarea functiilor sunt la un nivel la dreapta fata de declararea functiei
  - Blocurile de cod sunt scrise cu un nivel la dreapta fata de blocul parinte, si asa mai departe

```
#define I int
void bs(I v[], I n) {
I k=0, i;
while (k==0)
{
    k=1;
          for (i=0; i<n-1; i++)
          if (v[i]>v[i+1])
    {
    k=0; I a = v[i];
    v[i]=v[i+1];
    v[i+1]=a;
    }
```

```
void bubbleSort(int sirNumere[], int lungimeSir)
{
    bool amModificat = true;
    while (amModificat)
    {
          amModificat = false;
          for (int indexSir=0; indexSir<lungimeSir-1; indexSir++)
               if (sirNumere[indexSir] > sirNumere[indexSir+1])
               {
                    amModificat = true;
                    int aux = sirNumere[indexSir];
                    sirNumere[indexSir] = sirNumere[indexSir+1];
                    sirNumere[indexSir+1] = aux;
```

Bibliografie

- **The Clean Coder:** A Code of Conduct for Professional Programmers **(**Robert C. Martin Series)

- **Clean Code**: A Handbook of Agile Software Craftsmanship (Robert C. Martin Series)