Programare II
Programare Orientată Obiect

Curs anterior

- Stanard Template Library
  - Containere asociative
  - Algoritmi
  - Clasa string

Curs Curent

- Procesul de dezvoltare software
  - OOA
  - UML

- Modelul de programare orientată pe obiect
  - Introducere
  - Concepte
  - Structurarea obiectelor, atributelor şi serviciilor
  - Principii OOD - S.O.L.I.D.

PROCESUL DE DEZVOLTARE SOFTWARE

- Ciclu de viață
  1. Planificare inițială
  2. Analiza specificațiilor (cerințelor)
  3. Design și dezvoltarea inițiala
  4. Codarea și implementarea specificațiilor
  5. Testare manuală și automată - testare și integrare
  6. Evaluarea pe partea clientului
  7. Release
  8. Suport

MODELE DE DEZVOLTARE SOFTWARE

- Waterfall
  - cel mai vechi, bine ştiut, diferiţi paşi (cerinţe, analiză, design, implementare, testare, etc) care sunt urmaţi în ordine
- V-model
  - introdus de administraţia germană; 2 fluxuri: specificaţii (analiza cererilor, specificaţii funcţionale, specificaţii de proiectare) şi testare (instalare, operaţional, performanţe) + implementare
- Iterativ
  - dezvoltarea de software în iteraţii mici; ajută la identificarea de probleme în azele incipiente ale procesului de dezvoltare:
- Cleanroom
  - combină metode formale în fazele de definire a specificațiilor şi design cu implementare incrementală şi testare statistică
- Metode formale
  - abordări matematice pentru a rezolva problemele din specificaţii, specificaţii şi nivelele de design (ex. B+method, Petri nets, AISE= Rigorous Approach to Industrial Software Engineering , VDM = Vienna Development Method)

MODELE DE DEZVOLTARE SOFTWARE

- Iterativ: dezvoltarea de software în iteraţii mici; ajută la identificarea de probleme în azele incipiente ale procesului de dezvoltare
  - Spiral: combină modelele de prototipare cu cel waterfall
  - Rational Unified Process (RUP)
  - Agile
  - Extreme Programming (XP)
  - Scrum
  - Kanban

ANALIZA ORIENTATĂ OBIECT

- Definiţie: Scopul OOA (Object Oriented Analysis) este de a modela domeniul problemei, problema pe care dorim să o rezolvăm prin dezvoltarea unui sistem orientat obiect.
- Poate fi aplicată oricărui proces de modelare a dezvoltării software
- Intrări:
  - Cerinţele (problema)
  - Specificațiile (pot include diagrame de utilizare (use case) sau alte diagrame)
- Ieşirile:
  - Modelul conceptual
  - Cazurile de utilizare
  - Orice altă documentaţie

ANALIZA ORIENTATĂ OBIECT

- OOA nu se ocupă de detaliile de implementare (structura bazei de date, modelul de persistenţă etc), acestea sunt decise în pasul OOD
- Notaţii grafice
  - Coad, Yourdon, Rumbaugh, Booch, Firesmith, Embley, Kurtz, Etc
- Unified Modeling Language (www.uml.org) (UML) – standardul pentru OOA
- Taskuri realizate în timpul analizei OOA
  - Găsirea obiectelor
  - Găsirea relaţiilor dintre obiecte
  - Definirea cazurilor de utilizare
  - Definire UI

UNIFIED MODELING LANGUAGE

- Diagrame comportamentale: diagrame care descriu comportamentul sistemului sau procesului de business.
  - Activitate
  - State machine
  - Use case
  - Interaction
- Diagrame de interacţiune: o submulţime de diagrame comportamentale care subliniază comportamentul sistemului
  - Communication
  - Interaction overview
  - Sequence
  - Timing diagrams
- Diagrame structurale: depistează elementele specificaţiilor care ar nu sunt transparente în acest moment
  - Class
  - Composite structure
  - Component
  - Deployment
  - Object
  - Package diagrams

OBJECT ORIENTED DESIGN

- Definiţie
  - Object-oriented design (OOD) este o disciplină care defineşte obiectele şi modul în care ele interacţionează ca să rezolve probleme care au identificate şi documentate în timpul analizei orientate obiect

- Face tranziţia de la proiectarea produsului la dezvoltarea software (software arhitecture -> software development)

- Intrările pentru OOD
  - Intrarile OOA (Object Oriented Arhitecture) (modelul conceptual, diagrame de utilizare, documentaţie UI, alte documente)

- Ieşirile (deriverables/output) pentru OOD
  - Diagrame de clasă
    - Descriu structura stemului prin clase, cu atributele lor şi relaţiile dintre clase
  - Diagrame de secvenţă
    - Schimbul de mesaje dintre anumite obiecte şi ordinea în care se realizează

CONCEPTE OOD

- Paşi OOD
  - Definirea obiectelor: identificarea atributului, comportamentului, serviciilor expuse de obiecte
  - Crearea de diagrame din modelul conceptual
  - Definirea framework-ului aplicaţiei:framework-ul aplicaţiei se referă la un set de biblioteci sau clase folosit pentru a crea structura unei aplicaţii pentru un sistem de operare. Folosirea unui framework reduce timpul necesar scrieri de cod dezvoltatorului aplicaţiei prin refolosirea funcţionalităţi standard.
  - Identificarea obiectelor/datelor persistente: identificarea obiectelor care trebuie persistate. Dacă sunt folosite baze de date relaţionare acest pas este echivalent cu maparea obiect relaţie
  - Identificarea, definirea obiectelor remote
  - Evaluarea limbajelor OO şi alegerea celui mai potrivit
  - Evaluarea proiectării OO
  - Definirea strategiilor de testare: testare unitară, testare non-regresivă, teste de integrare, etc.

- Cum se realizează
  - Pe baza experienţei, „bunului simţ", folosind principiile OOD şi şabloanele de proiectare

STRUCTURAREA OBIECTELOR

- Generalizare-specializarea ierarhiilor (is-a) – folosirea moştenirii pentru a grupa atribute şi comportament comun regăsit la obiecte
  - Reuniunea tuturor specializărilor acoperă generalizările descrise?
  - Se exclud reciproc specializările?
  - Exemplu: formă, elipsă, punct
  - Moştenirea multiplă
    - Tendinţa de a complica lucrurile
    - Pot apărea conflicte între atribute similare, moştenite de la clasele de bază
    - Trebuie folosită cu grijă
    - Abordări gen Java: o singură reprezentare şi implementarea mai multor comportamente

- Ierarhii parte-întreg(has-a)
  - Exemplu: persoana are 1 corp, 0 sau 2 braţe, 1 cap, etc. o linie poligonală conţine 2..N puncte
  - Întregul nu moşteneşte comportament de la părţi => moştenirea nu este aplicabilă

ATRIBUTELE OBIECTELOR

- Găsirea atributelor
  - Folosirea persoanei întâi (personificare)
  - Analizarea problemei, chestionarea clientului

- Plasarea atributelor într-o ierarhie de clase – care clasă din ierarhie este mai potrivită pentru a conţine atributul?

- Definirea domeniului atributelor, ex. care sunt valorile valide pe care un atribut le poate lua

- Relaţiile dintre obiecte sunt implementate ca atribute, aceste făcând parte din starea obiectelor

- În această fază, ierarhiile de clase sunt revăzute

- Exemplu: un punct are 2 coordonate, denumite X şi Y, care pot lua doar valori pozitive

COMPORTAMENTUL OBIECTELOR

- Definiţie:
  - Comportamentul descrie activitatea unui obiect. Un serviciu defineşte relaţiile cu alte componente ale modelului

- Reprezentat prin diagrame UML de comportament şi interacţiune

- Posibilă identificare: stările obiectelor şi apoi explorarea sensului conexiunilor

- Au fost definite toate stările? Sunt toate stările tangibile? În fiecare stare, obiectul are comportamentul aşteptat

SERVICIILE OBIECTELOR

- Determinarea serviciilor necesare (=funcţii membre) pe baza tipurilor lor
  - Servicii implicite: crearea de noi instanţe (constructori), destructori, metode de get/set etc (metode care nu sunt incluse în diagrame de obicei)
  - Servicii asociate cu conectarea mesajelor: identificarea mesajelor trimise obiectelor în stări anterioare şi adăugarea de servicii care să le trateze; pot fi sugerate de diagramele de comportament
  - Servicii asociate cu relaţiile obiectelor: activează/deactivează relaţiile dintre obiecte (relaţiile sunt identificate în faza de proiectare OOA) (ex. Poligoanele conţin Puncte => adăugarea/ştergerea/modificarea de puncte obiectelor poligon)
  - Servicii asociate cu atributele: protejează anumite atribute, modifică un atribut doar împreună cu alt atribut, sincronizare în timp real etc.

SERVICIILE OBIECTELOR

- Mesajele sunt schimbate între obiecte pentru a executa servicii complete
- Reprezentare grafică
  - Ca funcţii membre în diagramele de clase
  - Conectori ai mesajelor în diferite diagrame de interacţiuni (colaborare/secvenţă, comunicare, interacţiune etc)
- Implementate ca şi funcţii membru publice

PRINCIPII OOD

- Definiţie
  - Un principiu de proiectare este un principiu sau utilitar de bază care poate fi aplicat pentru proiectarea sau scrierea de cod mai uşor de întreţinut, flexibil sau extensibil

- Principiile OOD - SOLID
  - OCP - Open-closed principle
  - DRY - Dependency Inversion Principle
  - SRP - Single-responsiblity principle
  - LSP - Liskov substitution principle
  - ISG - Interface segregation principle

OPEN-CLOSE PRINCIPLE (OCP)

- OCP
  - clasele ar trebui să fie deschise pentru extensii dar închise pentru modificare
- Permite schimbări fără a modifica codul existent
- Folosirea moştenirii pentru a extinde/schimba codul funcţional existent şi a nu se atinge de codul care funcţionează
- Poate fi implementat şi prin intermediul compoziţiei

OPEN-CLOSE PRINCIPLE (OCP)

NOK

```
class Shape {
    int type;
    void drawPolygon () { /* ... */ }
    void drawPoint () { /* ... */ }
  public:
    void draw();
};

void Shape::draw() {
  switch(type) {
    case POLYGON:
         drawPolygon (); break;
    case POINT:
         drawPoint (); break;
 }
}
```

OK

```
class Shape {
  public:
    virtual void draw() = 0;
};
class Polygon : public Shape {
  public:
    void draw();
};
class Point : public Shape {
  public:
    void draw();
};

void Polygon::draw() { /* ...*/ }
void Point::draw() { /* ... */ }
```

DON'T REPEAT YOURSELF (DRY)

- DRY
  - evitarea codului duplicat prin abstractizarea lucrurilor comune şi plasarea acestor lucruri într-o singură locaţie
- Fără cod duplicat => O cerinţă într-un singur loc
- Acest principiu poate şi ar trebui aplicat oriunde (ex. analiză – să nu se duplice cerinţe sau features)
- Codul este mai uşor de întreţinut şi mai sigur deoarece realizăm modificările doar într-u loc

DON'T REPEAT YOURSELF (DRY)

NOK

```
String::String(const char* pch) {
    if(pch!=NULL) {
        str = new char[(sz=strlen(pch))+1];
        strcpy(str, pch);
    } else {
        str = NULL;
        sz = 0;
 } }
void String::set(const char* pch) {
    if(str!=NULL) delete [] str;
    if(pch!=NULL) {
        str = new char[(sz=strlen(pch))+1];
        strcpy(str, pch);
    } else {
        str = NULL;
        sz = 0;
 } }
```

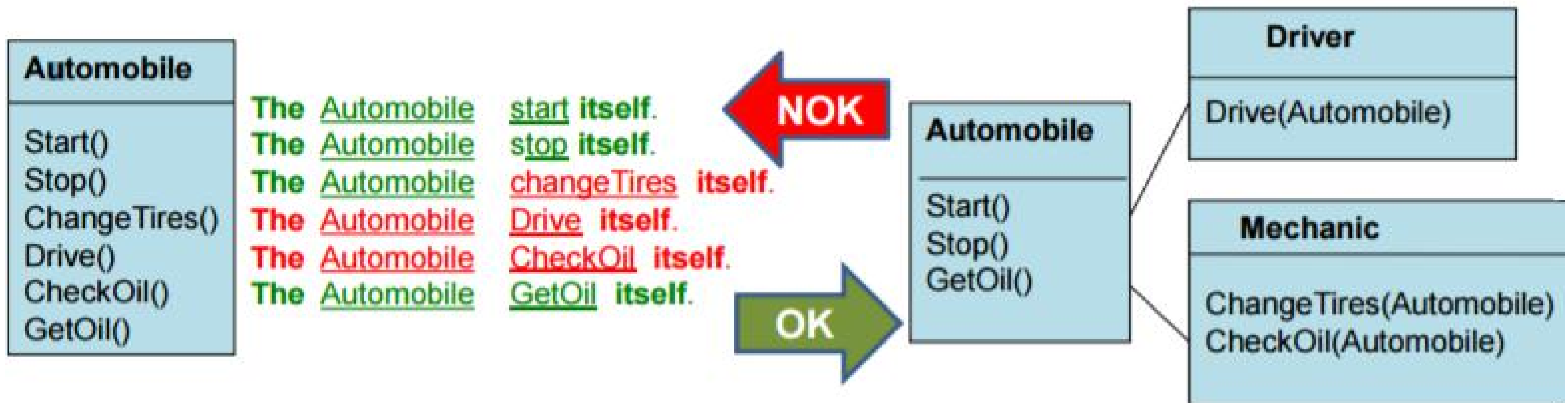
OK

```
/*private*/ void String::init(const char* pch) {
    if(pch!=NULL) {
        str = new char[(sz=strlen(pch))+1];
        strcpy(str, pch);
    } else {
        str = NULL;
        sz = 0;
    }
}
String::String(const char* pch) {
    init(pch);
}
void String::set(const char* pch) {
    if(str!=NULL) delete [] str;
    init(pch)
}
```

SINGLE RESPONSABILITY PRINCIPLE (SRP)

- SRP – fiecare obiect ar trebui să aibă o singură responsabilitate, şi toate serviciile ar trebui să se focuseze spre a retransmite acea responsabilitate
- Doar un singur motiv pentru a modifica ceva
- Codul este mai simplu şi uşor de întreţinut
- Exemplu: containerele şi iteratorii (containerele gestionează obiecte, iteratorii le traversează)
- Cum să nu suportăm responsabilităţi multiple? Formând propoziţii care se termină cu cuvântul *itself*

SINGLE RESPONSABILITY PRINCIPLE (SRP)



LISKOV SUBSTITUTION PRINCIPLE

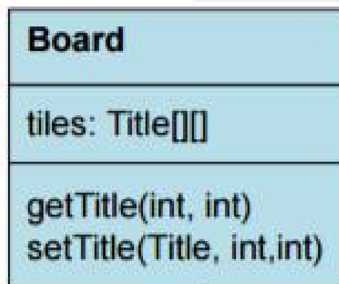
- LSP
  - subtipuri trebuie să fie potrivite pentru clasele lor de bază

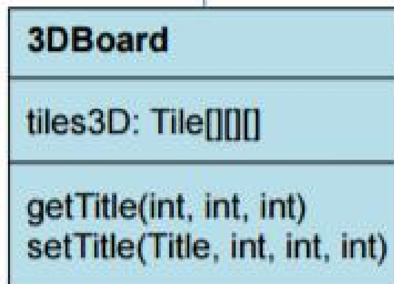
- Ierahii de clase bine proiectate

- Subclasele trebuie să fie potrivite pentru clasele de bază fără a ne gândi că ceva nu este bine

LISKOV SUBSTITUTION PRINCIPLE

NOK






```
void f() {
Board* board = new 3DBoard; // ok!
// doesn't make sense for a 3D board
board -> getTile (1,7);   }
```

OK

```
Tile 3DBoard::getTile (int x, int y, int z) {
        return boards[x].getTile (y, z);
}
```

Interface segregation principle

- ISP
  - Un client nu ar trebui să fie obligat să implementeze o interfață pe care nu o folosește sau un client nu ar trebui să depindă de metode pe care nu le folosesc.

Interface segregation principle

NOK

```
class ShapeInterface {
  public:
    virtual double area() = 0;
    virtual double volume() = 0;
};

class Square: ShapeInterface {
   public:
    double area() { /*calcul arie */}
    double volum() {/*nu are sens*/}
};

class Cuboid: ShapeInterface {
   public:
    double area() { /*calcul arie cub*/}
    double volum() {/*calcul volum cub*/}
};
```

OK

```
class ShapeInterface {
  public:
    virtual double area() = 0;
};

class SolidShapeInterface {
  public:
    virtual double volume() = 0;
};

class Square: ShapeInterface {
   public:
    double area() { /*calcul arie */}
};

class Cuboid: ShapeInterface, SolidShapeInterface  {
   public:
    double area() { /*calcul arie cub*/}
    double volum() {/*calcul volum cub*/}
};
```

MOȘTENIREA VS.COMPOZIȚIE

Moştenirea (white-box)	Compoziţia obiectelor (black-box)
Vizibiliatea	Refolosirea
Static (compile time)	Dinamic (se poate modifica la runtime prin instanţiere)
Uşor de folosit şi înţeles	Greu de înţeles
Violează principiul încapsulării	Nu violează principiul încapsulării
Probleme de refolosire	Menţine fiecare clasă încapsulată şi focusată pe o anumită sarcină
Ierarhi de clase mari; puţine obiecte	Ierarhi de clase mici; mai multe obiecte