Modele de proiectare a aplicațiilor de întreprindere

Cursul 1 - 2

Prezentare disciplină

- Tematica disciplinei
- Modalitatea de desfășurare
- Modalitatea de evaluare

Tematica disciplinei

- Principii de design OO, modele de design, anti-modele
- Modele de proiectare (creaționale, structurale și comportamentale)
- Modele pentru aplicații concurente
- Modele de design în programarea aplicațiilor Web
- Modele de design în programarea aplicațiilor de întreprindere

Modalitatea de desfășurare

- Cursuri și laboratoare
- Laboratoare
  - Java (Eclipse, InteliJIdea, Netbeans)
  - C# (Visual Studio)
  - Android - Java (Android Studio)
- Platforma
  - Google Classroom
  - http://online.ase.ro
- Fișa disciplinei
  - http://fisadisciplina.ase.ro/

Modalitatea de evaluare

- Seminar (50%)
  - Proiect în echipă
  - Prezentare la ultima întîlnire
  - Evaluare individuală
  - [illegible]
- Curs (50%)
  - Probă practică, la calculator
  - [illegible]

Sumar

- Aplicații de întreprindere
- Principii ale POO
- Modele de design
- Anti-modele de design
- Modele de proiectare (GoF)

Aplicații de întreprindere

Aplicații de întreprindere

- Destinate organizațiilor cu scopul de a asista componenta de business în rezolvarea diferitelor probleme specifice
- Caracteristici
  - Complexe
  - Scalabile
  - Distribuite
  - Bazate pe componente
- Deployment (Instalare)
  - Rețelele companiilor
  - Intranet
  - Internet

Provocări

- Cerințele aplicațiilor se modifică în timp
- Apar provocări și noi oportunități de afaceri
- Cerințele apărute în timpul dezvoltării pot conduce la modificarea sferei de cuprindere și cerințelor inițiale ale aplicației
- Aplicațiile sînt complexe și se dezvoltă în echipă

Cerințe

- Proiectarea aplicațiilor astfel încît acestea să poată modificate sau extinse cu ușurință în timp
- Proiectarea de componente individuale, independente, care pot fi dezvoltate și testate izolat
- Partiționarea aplicației în componente discrete și slab cuplate între ele, care pot fi integrate cu ușurință

Aplicații de întreprindere: cerințe de proiectare

Modelul de dezvoltare	• Echipa, procese, management, testare, livrabile.
Modelul de afaceri	• Obiective, resurse, timp, reguli
Modelul utilizatorilor	• UI, instruire, configurare
Modelul logic	• Structura aplicației, modelare obiectelor/datelor, definirea interfețelor
Modelul tehnologic	• Instrumente de dezvoltare, SGBD, platforme de instalare
Modelul fizic	• Arhitectura fizică a aplicației, distribuirea componentelor

Principii în dezvoltarea software

Restricții

- Programarea se face întotdeauna în contextul unor restricții
- Mentenabilitatea: întotdeauna o restricție importantă
- Eleganța codului: adaptarea ideală la restricții

Caracteristicile codului scris bine

- Lizibilitate
- Testabilitate
- Structurarea
- Dimensiune redusă, ușor de gestionat
- Funcționalitate simplă și specifică

Stiluri de scriere a codului

- https://google.github.io/styleguide/javaguide.html
- https://google.github.io/styleguide/cppguide.html
- https://source.android.com/source/code-style.html

- Suport
  - https://pmd.github.io/

Principii de dezvoltare software

- **DRY** – Don't repeat yourself
- **KISS** – Keep It Simple, Stupid
- **YAGNI** – You Aren't Gonna Need It

DRY – Don't repeat yourself

- " *Fiecare element de cunoaștere trebuie să aibă o reprezentare unică, lipsită de ambiguitate și autoritate în cadrul unui sistem*"
- Reducerea repetării informațiilor de orice natură
- Duplicarea în logică este eliminată prin abstractizare
- Duplicarea în proces este eliminată prin automatizare
- Împărțirea proiectului în componente care pot fi gestionate

KISS – Keep It Simple, Stupid

- Simplicitatea trebuie să fie un obiectiv în proiectarea entităților software
- Implementarea, cît mai simplă cu putință

YAGNI – You Aren't Gonna Need It

- Funcționalitățile trebuie adăugate doar atunci cînd este nevoie de ele
- Reducerea complexității prin reducerea numărului de module
- Simplicitate redusă pînă la neimplementarea de module
- 80% din timpul alocat unui proiect software este dedicat pentru 20% din funcționalități

Mecanisme de design orientat obiect

- Moștenire
- Compoziție
- Polimorfism
- Delegare
- Tipuri generice

Principii fundamentale de design (POO)

- Separă ceea ce variază
- Programează la interfață, nu la implementare
- Preferă compoziția moștenirii
- Scrie cod adaptabil la schimbare

Principii de design orientat obiect: SOLID

- **S**RP – Single Responsibility Principle
- **O**CP – Open Closed Principle
- **L**SP – Liskov Substitution Principle
- **I**SP – Interface Segregation Principle
- **D**IP – Dependency Inversion Principle

SRP – Single Responsibility Principle

- O clasă trebuie implementată pentru un singur scop (responsabilitate)
- Nu trebuie să existe mai mult de un motiv pentru a modifica o clasă
- Dacă există mai multe responsabilități, o clasă trebuie împărțită în mai multe clase
- Dacă ar fi mai multe responsabilități, modificarea uneia poate afecta cealaltă responsabilitate

Avantaje SRP

- Testabilitatea
- Cuplarea scăzută
- Reducerea complexității claselor

OCP – Open Open Closed Principle

- Orice clasă trebuie să fie
  - **deschisă pentru extindere** și
  - **închisă pentru modificare**
- Funcționalitățile noi sînt adăugate cu schimbări minimale ale codului existent

LSP – Liskov Substitution Principle

- Orice clasă derivată trebuie să poată substitui clasa de bază
- Clasele derivate nu trebuie să modifice fundamental funcționalitățile din clasele de bază
  - Pot apărea efecte nedorite în modulele existente
- Dacă o clasă de bază este înlocuită cu o clasă derivată, funcționalitatea programului nu trebuie să fie afectată

ISP – Interface Segregation Principle

- Clasele nu trebuie să implementeze interfețe pe care nu le utilizează
- Interfețele trebuie proiectate astfel încît să nu includă metode care vor fi utilizate doar într-un anumit context, pe lîngă metodele comune
- Este de preferat să fie definite mai multe interfețe

DIP – Dependency Inversion Principle

- Clasele de nivel înalt nu trebuie să depindă de clasele de nivel scăzut
  - Ambele trebuie să depindă de abstractizări
- Abstractizările nu trebuie să depindă de detalii
  - Detaliile vor depinde de abstractizări
- Proiectarea claselor
  - Nivel înalt
  - Nivel abstractizare
  - Nivel scăzut
- Soluția uzuală: *dependency injection*

Perspective asupra POO

- Design complex vs. YAGNI
- Worse is better vs. the right thing
  - simplitatea implementării vs. simplitatea interfeței
- Mentenabilitatea este cea mai importantă restricție asupra codului
- Codul mentenabil este puternic decuplat

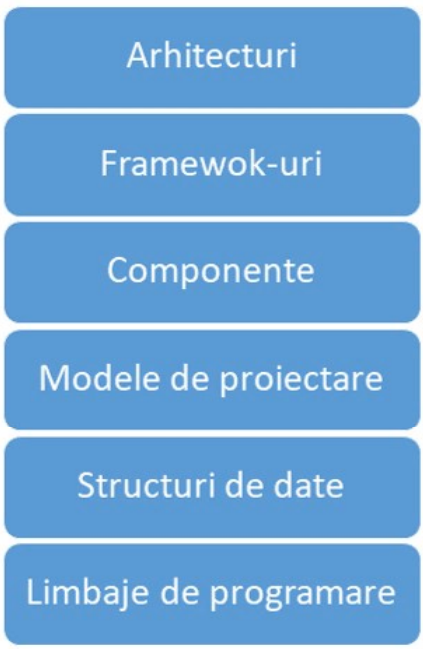
Pași fundamentali de design pentru POO (GoF)[1]

- Identificarea abstracțiilor corecte
  - Care sunt obiectele prin care se poate reprezenta universul de discuție al problemei?
- Ce granularitate trebuie să aibă obiectele selectate?
- Ce interfețe trebuie să aibă obiectele?
- Ce implementare trebuie să aibă obiectele?

[1]**GoF:** Gang of Four (Gamma, Helm, Johnson, Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, 1995)

Modele de proiectare

Contextul



Modele de proiectare

- Soluțiii reutilizabile pentru un anumit set de probleme
- Echipe separate de programatori vor ajunge la aceeași soluție sau soluții similare pe o perioadă suficient de mare de timp
- Soluția este elegantă în raport cu restricțiile definite
- Modelele nu sînt atît de complexe încît să devină framework-uri
- Modelele nu atît de simple încât să devină elemente de sintaxă a unui limbaj

Niveluri de abstractizare pentru modele de proiectare

- **Arhitectură**
  - Furnizează un set de subsisteme predefinite
  - Specifică responsabilitățile acestora
  - Include reguli și linii directoare pentru organizarea relațiilor dintre ele
  - Exemplu: organizarea aplicațiilor pe niveluri
- **Proiectare**
  - Oferă o schemă de rafinare a subsistemelor sau componentelor unui sistem software sau relațiile dintre acestea
  - Descrie o structură frecvent recurentă a componentelor care comunică, care rezolvă o problemă de proiectare generală într-un anumit context
  - Exemplu: MVC (Model View Controller)
- **Implementare**
  - Model de nivel scăzut specific unei anumite platforme
  - Descrie modul de implementare a anumitor aspecte ale componentelor sau a relațiilor dintre ele, utilizând caracteristicile unei platforme date
  - Exemplu: implementări specifice Java, .NET etc.

Perspective ale modelelor de proiectare

- Baze de date
  - Nivelul de asigurare a persistenței
- Aplicație
  - Aspectele executabile ale soluției
- Deployment (Instalare)
  - Asocierea componentelor aplicației la infrastructură
- Infrastructură
  - Include toate componentele hardware și de rețea necesare rulării soluției

Gruparea modelelor de proiectare pentru aplicații de întreprindere

Model de design vs. Framework

- Mecanisme pentru reutilizarea designului
- Modele de design
  - Nu sînt concrete
  - Independente de limbajul de programare
  - Elementele arhitecturale de bază pentru framework-uri
- Framework
  - Compilabile
  - Uzual conțin clase abstracte și interfețe
  - Pun la dispoziție infrastructura care permite extinderea și dezvoltarea soluțiilor proprii

Limitări ale modelelor de proiectare

- Pot îmbunătăți sau nu înțelegerea unui proiect sau a unei implementări.
- Pot reduce inteligibilitatea prin adăugarea de indirectări sau creșterea numărului de componente (clase și interfețe)
- Pot conduce la creșterea complexității
- Tendința de a forța utilizarea modelelor de proiectare
- Alegerea necorespunzătoare a unui model de proiectare

Managementul complexității codului sursă

- Utilizarea modelelor de proiectare poate conduce la reducerea complexității ciclomatice a codului sursă

$$CC = V - N + 2 \times C$$

  - V – numărul de vîrfuri
  - N – numărul de noduri
  - C – numărul de componente conexe

- Complexitatea este redusă prin furnizarea de abstractizări gata de utilizare

API (Application Programming Interface)

- O modalitate de comunicare pusă la dispoziție de diferite componente software:
  - Biblioteci software
  - Framework-uri
  - Sisteme de operare
  - Sisteme la distanță
  - Web
- Acces: Privat, partener și public
- Ascunderea informației

Modele de design vs mecanisme de limbaj

- Modelele de design extind capabilitățile anumitor limbaje de programare
- Modelele de design folosite frecvent într-un limbaj pot fi invizibile sau triviale într-un alt limbaj
- Exemplu: 16 din 23 modele de proiectare sînt invizibile sau mai ușor de utilizat în Lisp

Modele de design vs mecanisme de limbaj

- Anumite modele de proiectare au fost înlocuite cu facilități ale limbajelor de programare
  - Funcții generice
  - Expresii lambda
  - Funcții anonime
  - Module
  - Dicționare, tabele de dispersie
  - Delegați
- Modelele de design nu pot fi incluse în totalitate într-un limbaj de programare

Pași de aplicare a unui model de design

- Identificarea unei situații care reprezintă o problemă standard
- Identifcarea modelelor aplicabile
- Identificarea interacțiunilor dintre modele
- Aplicarea modelelor la situația dată

Anti-modele

Anti-modele

- **Nevalidarea intrărilor**
  - Nu se specifică și nu sînt puse în aplicare gestionarea eventualelor intrări invalide
- **Race Hazard**
  - Neobservarea consecințelor ordinii diferite a evenimentelor
- **Dependențe circulare**
  - Introducerea dependențelor reciproce directe sau indirecte între obiecte sau module software
- **Obiectul central**
  - Un obiect care are prea multe informații sau prea multă responsabilitate.
  - Includerea multor funcții într-o singură clasă.
  - Adesea codul pentru un model și un view sunt combinate în aceeași clasă
- **Interfețe utilizator supraîncărcate**
  - Efectuarea unei interfețe atât de puternică și complicată încât este greu de reutilizat sau implementat

Anti-modele

- **Numerele magice**:
  - Includerea de valori constante direct în cod, fără nici o explicație a semnificației acestora
- **Șiruri magice**:
  - Includerea șirurilor literale în cod, pentru comparații, tipuri de evenimente etc.
- **Programarea copy/paste**
  - Copierea și modificarea codul existent fără a crea mai multe soluții generice
  - Se folosesc aceleași secvențe de cod în mai multe locuri, cu mici modificări
  - Nu se respectă principiul DRY
- **Reinventarea roții**
  - Nu sunt folosite soluții existente și adecvate și, în schimb, să alege o soluție personalizată, care se comportă mult mai rău decât cea existentă
  - Se face totul prin noi înșine și se scrie totul de la zero

Anti-modele

- **Optimizare prematură**
  - Opus YAGNI
  - Efecte: reducerea lizibilității codului, depanarea și întreținerea devin mai greu de realizat și adăugarea unor părți inutile la codul scris
- **Prea multe dependențe**
  - Se utilizează prea multe biblioteci terțe care se bazează pe versiuni specifice ale altor biblioteci
  - Apar incompatibilități
- **Cod Spaghetti**
  - Cod greu de depanat sau modificat din cauza lipsei unei arhitecturi adecvate
- **Programarea prin permutări**
  - Încercarea de a găsi o soluție pentru o problemă prin experimentarea succesivă cu mici modificări, testarea și evaluarea acestora una câte una și, în final, implementarea celei care a funcționat la început
  - Nu se știe dacă soluția va funcționa în toate scenariile sau nu

Anti-modele

- **Lava Flow**
  - Codul care are componente redundante sau de calitate slabă, care par a fi parte integrantă a programului, dar nu fără a înțelege cu desăvârșire ce anume face sau cum influențează întreaga aplicație
  - Uzual, codul este preluat (scris de altcineva) sau proiectul se derulează prea rapid
- **Hard coding**
  - Încadrarea ipotezelor despre mediul unui sistem în implementarea acestuia
- **Soft Coding**
  - Anumite componente care ar trebui să fie în codul sursă sunt plasate în surse externe
- **Cargo cult programming**
  - Scrierea codului fără înțelegerea acestuia
  - Teama de modificare: poate nu mai rulează codul corect

Modele de proiectare (GoF)

Modele de proiectare

- Tipul
- Sfera de cuprindere
  - Nivel de clasă
  - Nivel de obiect
- Definire
  - Denumirea
  - Problema
    - Context, exemple
  - Scopul
  - Soluția
    - Structura
    - Implementarea
  - Consecințe
  - Variante, utilizări frecvente

Modele de proiectare (GoF)

Creaționale	Structurale	Comportamentale
• Factory Method • **Abstract Factory** • Builder • Prototype • Singleton	• Adapter (clasă)/Adapter (obiect) • **Bridge** • Composite • Decorator • Façade • Flyweight • Proxy	• ***Interpreter*** • Template • Chain of Responsibility • Command • ***Iterator*** • ***Mediator*** • Memento • Observer • State • Strategy • ***Visitor***

Referințe

- .NET Design Patterns, http://www.dofactory.com/net/design-patterns
- Data & Object Factory, *Gang of Four Software Design Patterns*, Companion document to Design Pattern Framework™ 4.5, 2017
- Data & Object Factory, *Patterns in Action 4.5*, A pattern reference application, Companion document to Design Pattern Framework™ 4.5, 2017
- Design Patterns | Object Oriented Design, http://www.oodesign.com/
- Design patterns implemented in Java, http://java-design-patterns.com/patterns/
- M. Fowler, D. Rice, M. Foemmel, E. Hieatt, R. Mee, R. Stafford, *Patterns of Enterprise Application Architecture*, Addison Wesley, 2002
- E. Freeman ș.a, *Head First Design Patterns*, O'Reilly, 2004
- J.D. Meier et al, Application Patterns, http://apparch.codeplex.com/wikipage?title =Application%20Patterns, 2009
- D. Schmidt, M. Stal, H. Rohnert and F. Buschmann, *Pattern-Oriented Software Architecture, Patterns for Concurrent and Networked Objects*, Volume 2, John Wiley & Sons, 2000
- A. Shivets, *Design Patterns Made Simple*, http://sourcemaking.com
- Stack Overflow, https://stackoverflow.com/
- D. Trowbridge et. al, *Enterprise Solution Patterns Using Microsoft .NET*, Version 2.0, Microsoft, 2003
- Enterprise-application-ecommerce software development , https://www.heminfo.com/enterprise-application.html