Paradigme de Programare - Laborator 4

Utilizarea principiilor SOLID în Kotlin

Principiile S.O.L.I.D

Pentru o înțelegere mai ușoară a principiilor SOLID, vezi:

- The SOLID principles in pictures

Principiul Responsabilității unice (Single responsibility principle)

O clasă trebuie să aibă o singură rațiune pentru a se schimba. Cu alte cuvinte, clasele complicate trebuie divizate în clase mai mici care au responsabilități explicite.

Principiul Închis/Deschis (Open/closed principle)

Entitățile software trebuie să fie deschise pentru extindere, dar închise pentru modificare. Acest lucru înseamnă că fiecare entitate software trebuie scrisă astfel încât să poată fi extinsă fără a necesita modificări explicite în interiorul lor.

Principiul substituției Liskov (Liskov substitution principle)

Clasele derivate trebuie să fie substituibile pentru clasele lor de bază. Adică funcțiile care folosesc referințe la clase de bază, trebuie să poată manipula într-un mod transparent instanțele claselor derivate din acestea. Așadar clasele derivate trebuie doar să adauge funcționalități noi la clasele de bază, nu să le înlocuiască pe cele existente.

Principiul separării Interfețelor (Interface segregation principle)

Mai multe interfețe specifice sunt mai bune decât o interfață generală. Nici un client nu trebuie forțat să depindă de metode pe care nu le folosește. Numărul de membri din interfață care este vizibil pentru clasele dependente trebuie minimizat. Clasele mari vor implementa mai multe interfețe mai mici care grupează funcțiile după maniera lor de utilizare.

Principiul Dependenței inverse (Dependency inversion principle)

Modulele de nivel arhitectural superior nu trebuie să depindă de cele de nivel inferior. Ambele trebuie să depindă de abstracții care, la rândul lor nu trebuie să depindă de detalii (implementări concrete). Practic detaliile depind de abstracții, nu invers. Dacă această dependență nu este vizibilă în faza de proiectare, atunci ea se construiește.

Alte principii de programare

- KISS (Keep It Simple, Stupid!)
- DRY (Don't Repeat Yourself!)
- YAGNI (You Ain't Gonna Need It!)
- Separarea responsabilităților (Separation of Concerns) - de exemplu MVC într-o aplicație Web.
- Legea lui Demeter - reducerea cuplării și îmbunătățirea încapsulării

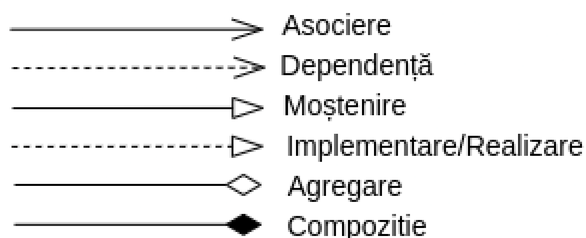
UML (Unified Modeling Language)

Pentru o imagine de ansamblu asupra diagramelor UML, vezi:

- UML 2.5
- UML Cheatsheet

Orice săgeată are:

- Baza săgeții - Clasa luată în considerare, sursa (subiectul 1)
- Vârful săgeții - se referă la cine este în legătură cu subiectul 1, de cine depinde ierarhic, “target” (subiectul 2).
- În cazul vârfului romb, clasa unde se află vârful conține un potențial tablou (array) de elemente de tipul clasei din partea săgeții de la capătul opus.

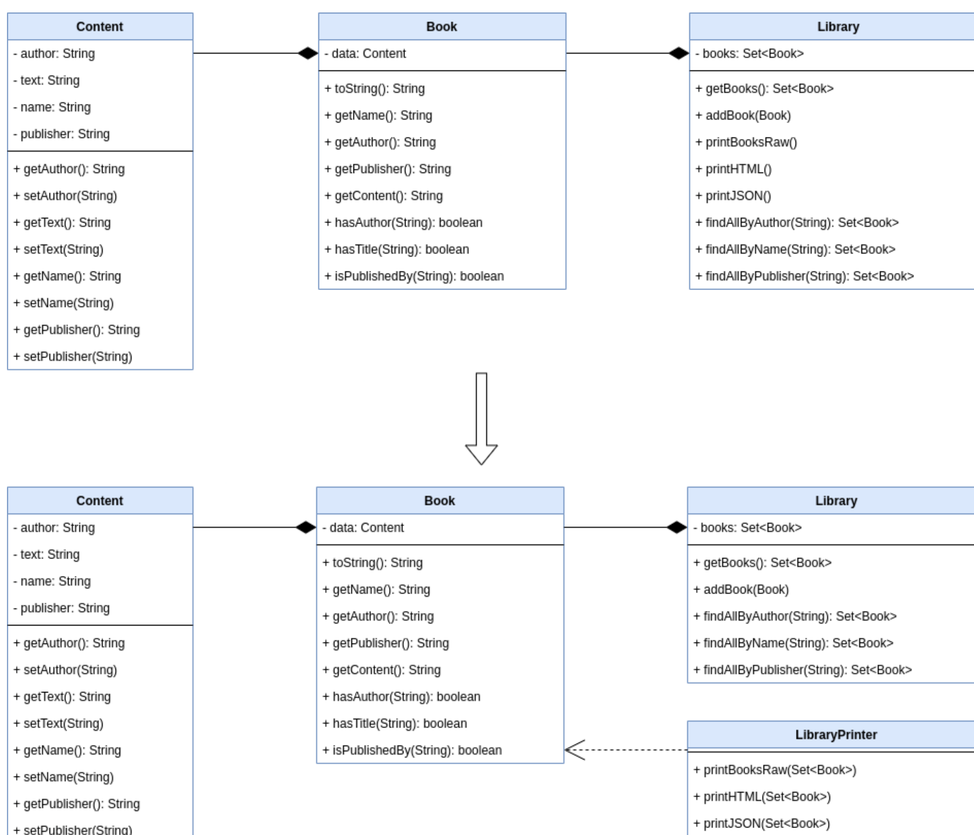


- **Asociere** - relație de asociere între două clase (clasa sursă folosește membri din țintă (target)) sub formă de câmp / atribut / proprietate (terminologia diferă).
- **Dependență** - relație de dependență între două clase (clasa sursă folosește clasa din țintă ca un parametru sau ca o variabilă în interiorul funcțiilor).
- **Moștenire** - moștenire (clasa sursă derivează clasa țintă, adăugându-i noi funcționalități).
- **Implementare (Realizare)** - Implementarea unei interfețe (clasa sursă implementează interfața din țintă)
- **Agregare** - implică o relație în care o clasă A conține una sau mai multe instanțe ale clasei B, iar ștergerea instanței clasei A **nu** duce la ștergerea instanțelor clasei B (Exemplu: o sală de curs și studenții care participă la curs).
- **Compoziție** - spre deosebire de agregare, ștergerea instanței clasei A duce la ștergerea tuturor instanțelor clasei B (Exemplu: Apartament, cameră -> dacă se șterge instanța de apartament, camerele nu pot exista fără acesta și se șterg).

Obs.: Problemele din acest laborator nu necesită support GraalVM

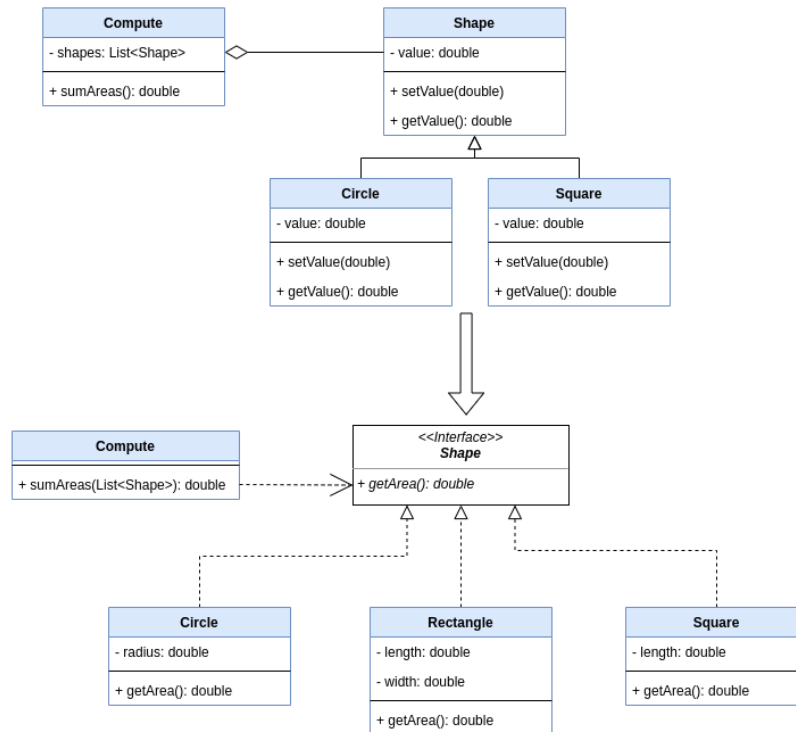
Exemplul 1 - Principiul Responsabilității Unice

În prima diagramă se observă că în clasa Library se regăsesc pe lângă funcții care operează pe lista de cărți, alte trei funcții de serializare: printHTML, printJSON, printBooksRaw. Acestea sunt specifice unui obiect serializer.



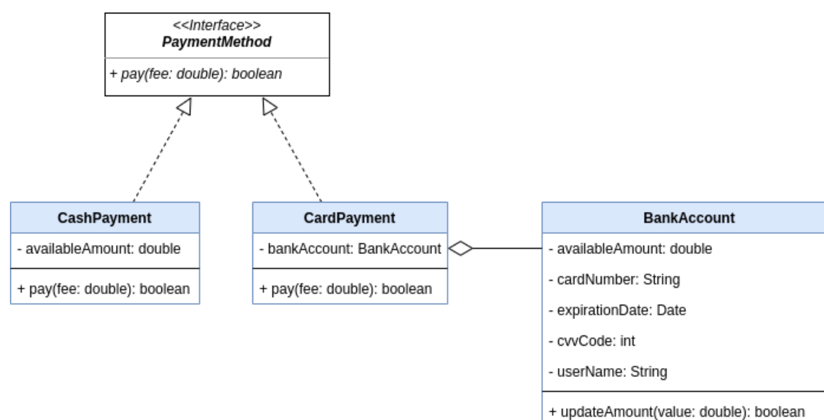
Exemplul 2 - Principiul Închis/Deschis

În prima diagramă pentru funcția sumAreas este nevoie de un switch case care să gestioneze tipul instanței de Shape astfel încât să calculeze aria cu formula specifică. Totuși, această abordare nu permite adăugarea unei noi forme fără modificarea codului deja existent. O soluție posibilă ar fi ca toate clasele derivate să implementeze o funcție specifică de calcul a ariei.



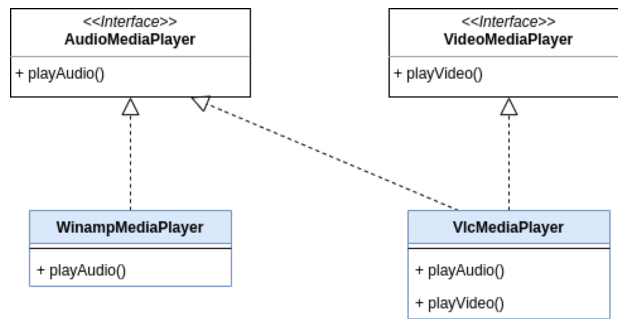
Exemplul 3 - Principiul Substituției Liskov

Se observă în diagrama de mai jos că cele două implementări concrete rezolvă aceeași problemă prin modalități diferite, păstrând funcționalitatea din clasa de bază (sau în cazul de față, implementarea funcțiilor din interfață), adică pot fi substituite tipului de bază (interfeței / clasei moștenite).



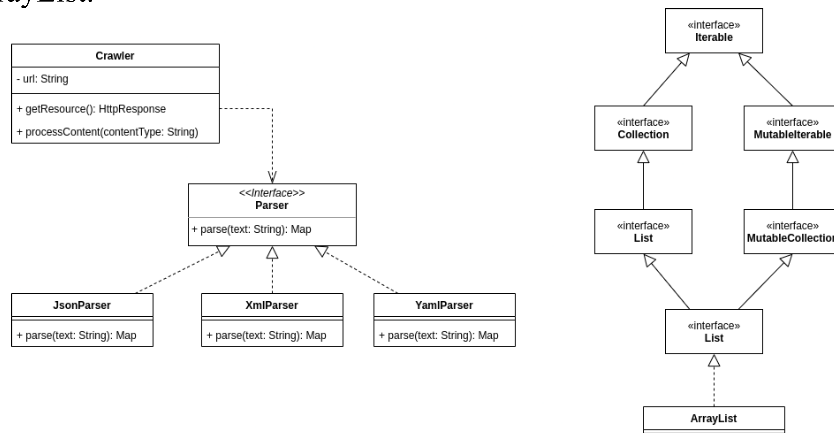
Exemplul 4 - Principiul Separării Interfețelor

Se remarcă în diagrama de mai jos nevoia de a avea mai multe interfețe specifice în locul uneia generale. Spre exemplu, dacă ar fi fost doar o interfață care să grupeze cele două funcționalități, clasa *WinampMediaPlayer* ar fi trebuit să implementeze și funcția *playVideo*.



Exemplul 5 - Principiul Dependenței Inverse

În prima diagramă se observă că acel Crawler nu instanțiază direct un parser în interiorul funcției *processContent*, ci se folosește de o abstractizare (interfața care e implementată de fiecare parser). Similar în cazul colecțiilor utilizate frecvent în orice limbaj: există multe abstractizări, după cum se vede în cea de-a doua diagramă, iar implementarea propriu zisă a unei liste este un ArrayList.



Aplicații şi teme

Aplicații de laborator:

1. Să se implementeze a doua diagrama UML de la exemplul 1 astfel încât să se respecte principiul separării interfețelor și principiul dependenței inverse (vezi clasa LibraryPrinter).
2. Respectând principiul închis/deschis, să se modifice exemplul 1 astfel încât să se adauge atributul *price*. Totodată, se va păstra funcționalitatea inițială (clasele Library și LibraryPrinter), prin utilizarea principiului dependenței inverse.

Teme pe acasă:

1. Utilizând modulul khttp și/sau Jsoup, să se implementeze crawler-ul Web din prima diagramă de clase de la exemplul 5. Fiecare parser va trebui să implementeze o metodă specifică tipului de răspuns (JSON / XML / YAML).
2. Să se proiecteze (să se deseneze diagrama UML de *clase*, precum și cea de *use case*) și să se implementeze respectând principiile SOLID, o aplicație pentru cumpărarea biletelor la cinema, plecând de la exemplul 3. Pentru desenarea diagramei, se poate utiliza https://app.diagrams.net/ (meniul UML).
3. Să se proiecteze (diagrama UML de clase) și să se implementeze respectând principiile SOLID, o aplicație de notițe. Acestea pot fi salvate pe disk în fișiere individuale. Printr-un meniu de tip CLI, aplicația va permite la pornirea acesteia:
   - să se afișeze lista de notițe
   - să se încarce o anumită notiță
   - să se creeze o nouă notiță
   - să se șteargă o anumită notiță.

Așadar aplicația trebuie să conțină minim trei clase: notița propriu zisă (autor, data, ora, conținut, etc), user-ul, manager-ul de notițe.