Laboratorul 13: Paradigma calculului funcțional în Python
Laboratorul 12: Paradigma calculului funcțional în Kotlin
Laboratorul 11: Utilizarea bibliotecilor Python pentru paralelism
Laboratorul 10: Corutine - suport nativ pentru paralelism în Kotlin
Laboratorul 9: Design patterns în Python
Laboratorul 8: Design patterns în Kotlin
Laboratorul 7: Colecții și generice în Kotlin
Laboratorul 6: Utilizarea POO în Python
Laboratorul 4: Utilizarea principiilor SOLID în Kotlin

Laboratorul 13: Paradigma calculului funcțional în Python

-Introducere-

Limbajele de programare suportă descompunerea problemelor în mai multe moduri diferite:

Majoritatea limbajelor de programare sunt procedurale: programele reprezintă liste de instrucțiuni care spun computerului ce să facă cu intrarea programului. C, Pascal și chiar Unix sunt limbaje procedurale.

În limbajele declarative, descrieți problema ce necesită rezolvare iar implementarea arată cum să efectuați calculul în mod eficient. SQL este limbajul declarativ cel mai cunoscut; o interogare SQL descrie setul de date pe care doriți să îl recuperați, iar motorul SQL decide dacă să scaneze tabele sau să utilizeze indexuri, care subclauzele trebuie efectuate mai întâi etc.

Programele orientate pe obiecte manipulează colecțiile de obiecte. Obiectele au stare internă și suportă metode care interogează sau modifică într-un fel această stare internă. Smalltalk și Java sunt limbaje orientate pe obiecte. C++ și Python sunt limbaje care acceptă programarea orientată pe obiecte, dar nu forțează utilizarea funcțiilor orientate pe obiecte.

Programarea funcțională descompune o problemă într-un set de funcții. În mod ideal, funcțiile doar iau intrări și produc rezultate și nu au nicio stare internă care să afecteze rezultatul pentru o intrare dată. Limbaje funcționale cunoscute includ familia ML (Standard ML, OCaml și alte variante) și Haskell.

Într-un program funcțional, intrarea trece printr-un set de funcții. Fiecare funcție acționează pe intrarea sa și produce o ieșire. Stilul funcțional descurajează funcțiile cu efecte secundare care modifică starea internă sau fac alte modificări care nu sunt vizibile în valoarea returnată de funcție. Funcțiile care nu au efecte secundare sunt denumite pur funcționale. Evitarea efectelor secundare înseamnă a nu utiliza structuri de date care se actualizează pe măsură ce un program rulează; ieșirea fiecărei funcții trebuie să depindă numai de intrarea sa.

Există avantaje teoretice și practice ale stilului funcțional: probabilitate formală, modularitate, compozabilitate, ușurință de depanare și testare.

Probabilitate formală

Un avantaj teoretic este că e mai ușor să dovedești matematic că un program funcțional este corect.

Modularitate

Un beneficiu mai practic al programării funcționale este că te obligă să desparți problema în bucăți mai mici. Ca urmare, programele sunt mai modulare. Este mai ușor să specifici și să scrii o funcție mică care face un lucru decât o funcție mare care realizează o transformare complicată. Funcțiile mici sunt, de asemenea, mai ușor de citit și de verificat când vine vorba de erori.

Compozabilitatea

Pe măsură ce lucrați la un program în stil funcțional, veți scrie o serie de funcții cu intrări și ieșiri diferite. Unele dintre aceste funcții vor fi inevitabil specializate pentru anumită aplicație, dar altele pot fi utile pentru o mare varietate de programe.

Ușurință de depanare și testare

Testarea și depanarea unui program în stil funcțional este mai ușoară.

-Iteratori-

Un iterator este un obiect care reprezintă un flux de date; acest obiect returnează valoarea unui element la un moment dat.

-Expresii generatoare și liste de comprehensiune-

Două operații comune care acționează pe ieșirea unui iterator sunt 1) efectuarea unei operații pentru fiecare element, 2) selectarea unui subset de elemente care îndeplinesc o anumită condiție. Listele de comprehensiune și expresiile generatoare sunt notări concise pentru astfel de operațiuni. Cu o listă de comprehensiune obțineți înapoi o listă Python. Expresiile generatoare returnează un iterator care calculează valorile după cum este necesar, nefiind necesar să materializeze toate valorile simultan. Expresiile generatoare sunt înconjurate de paranteze rotunde („()”) și listele de comprehensiune sunt înconjurate de paranteze pătrate („[]”).

-Generator-

Generatoarele sunt o clasă specială de funcții care simplifică sarcina de a scrie iteratorii. Funcțiile obișnuite calculează o valoare și o returnează, dar generatoarele returnează un iterator care returnează un flux de valori.

-Modulul itertools-

Modulul itertools conține o serie de iteratori utilizate și funcții pentru combinarea mai multor iteratori. Funcțiile modulului se încadrează în câteva clase largi:

-Funcții care creează un iterator nou pe baza unui iterator existent.
-Funcții pentru tratarea elementelor unui iterator ca argumente funcționale.
-Funcții pentru selectarea unor porțiuni a ieșirii unui iterator.
-O funcție pentru gruparea ieșirilor unui iterator.

-Modulul functools-

Modulul funcools din Python 2.5 conține câteva funcții de ordin superior. O funcție de ordin superior ia una sau mai multe funcții ca intrare și returnează o nouă funcție.

-Modulul operator-

Conține un set de funcții corespunzătoare operatorilor din Python. Unele dintre funcțiile din acest modul sunt:

-Operații matematice: add(), sub(), mul(), floordiv(), abs(), …
-Operații logice: not_(), truth().
-Operații în timp: and_(), or_(), invert().
-Comparații: eq(), ne(), lt(), le(), gt(), and ge().
-Identitate obiect: is_(), is_not().

-Funcții mici și expresia lambda-

Când scrieți programe în stil funcțional, veți avea nevoie adesea de mici funcții care să acționeze ca predicat sau care să combine elemente într-un fel. Un mod de a scrie funcții mici este să folosiți expresia lambda. lambda preia o serie de parametri și o expresie care combină acești parametri și creează o funcție anonimă care returnează valoarea expresiei.

Laboratorul 12: Paradigma calculului funcțional în Kotlin

Programarea funcțională este o paradigmă în care accentul este pus pe transformarea datelor cu expresii (ideal, aceste expresii ar trebui să nu aibă side effects).

Funcția de ordin superior 1 este o funcție care primește funcții ca parametri sau returnează o funcție.Funcțiile lambda pot fi utilizate ca parametri în alte funcții. Parametrul it ar trebui folosit doar în cazul în care este clar ce tip de date se folosește. De asemenea, dacă se dorește ca o funcție clasică să fie trimisă ca parametru, se poate utiliza double colon (::).

Într-un program, când o funcție modifică orice obiect/date din afara scopului propriu, acest lucru se numește efect secundar. O funcție care modifică o proprietate statică sau globală, modifică un argument, generează o excepție, scrie output-ul la consolă/în fișier sau apelează o altă funcție, are un efect secundar.

O funcție pură este o funcție a cărei rezultat returnat este complet dependent de parametrii acesteia. Pentru fiecare apel al unei funcții pure cu același parametru, aceasta va produce mereu același rezultat.De asemenea, o funcție pură NUtrebuie să cauzeze efecte secundare sau să apeleze alte funcții.

Funcțiile extensie permit modificarea tipurilor existente cu funcții noi. Sintaxa pentru adăugarea unei funcții extensie la un tip existent: NumeClasă.funcțieExtensie(listăParametri).

Un functor este un tip care definește o modalitate de a transforma (transform/map) conținutul lui.

Delegatul Delegates.notNull permite continuarea programului fără inițializarea proprietății notNullStr. Dacă acea variabilă este utilizată înainte de a fi inițializată, va genera o excepție. Funcția lateinit este o variantă mai simplă pentru a obține același comportament. Observație: Delegates.notNullși lateinit funcționează numai pentru proprietăți declarate cu var.

Spre deosebire de lateinit și Delegates.notNull, la funcția lazy trebuie specificată variabila de inițializare în momentul declarării, avantajul fiind că inițializarea nu va fi executată până când variabila este folosită.

Funcția Delegates.observable are nevoie de doi parametri pentru a crea delegatul: valoarea inițială a proprietății și o funcție lambda care să fie executată de fiecare dată când se schimbă valoarea proprietății. Funcția lambda primește 3 parametri:

-o instanță deKProperty<out R>(o proprietate -valsauvar)

-valoarea veche a proprietății

-valoarea nou asignată

Dreptul de veto permite o verificare logică la fiecare asignare a unei proprietăți, unde se poate decide dacă se continuă cu asignarea sau nu.

Laboratorul 11: Utilizarea bibliotecilor Python pentru paralelism

-Introducere-

Execuția paralelă reprezintă executarea a două sau mai multe programe simultan de către un singur computer.

-multiprocessing-

multiprocesarea este un pachet care suporta inițializarea proceselor prin utilizarea subproceselor. Pachetul oferă concurență locală și la distanță. În funcție de platformă, multiprocesarea acceptă trei moduri de a începe un proces. Aceste metode de pornire sunt:

spawn

Procesul părinte începe un nou proces. Procesul copil va moșteni doar acele resurse necesare pentru a rula metoda run(). În special, descriptorii și handler-ele de fișiere inutile din

procesul părinte nu vor fi moștenite. Începerea unui proces folosind această metodă este destul de lentă în comparație cu utilizarea fork sau forkserver.

fork

Procesul părinte folosește os.fork(). Procesul copilului, atunci când începe, este efectiv identic cu procesul părintelui. Toate resursele părintelui sunt moștenite de procesul copilului.

forkserver

Când programul pornește și selectează metoda de pornire a unui forkserver, se începe un proces de server. De atunci, ori de câte ori este nevoie de un nou proces, procesul părinte se conectează la server și solicită ca acesta să inițializeze un nou proces. Procesul unui forkserver este single threaded, deci este sigur pentru acesta să utilizeze os.fork(). Nu sunt moștenite resurse inutile.

multiprocesarea acceptă două tipuri de canale de comunicare între procese: cozi și conducte(pipe-uri). Pentru comunicare mesajelor se poate utiliza Pipe() (pentru o conexiune între două procese) sau o coadă (care permite mai mulți producători și consumatori).

Atunci când efectuați o execuție paralelă, este de obicei cel mai bine să evitați utilizarea stării partajate pe cât posibil. Acest lucru este valabil în special atunci când se utilizează mai multe procese. Cu toate acestea, dacă aveți nevoie cu adevărat de a utiliza unele date partajate, atunci multiprocesarea oferă câteva modalități de a face acest lucru: memoria partajată sau procesul server.

Clasa Pool reprezintă o serie de procese de lucrători.

-asyncio-

asyncio este o bibliotecă pentru a scrie cod paralel folosind sintaxa async/wait. asyncio oferă un set de API-uri de nivel înalt care:

-rulează coroutine Python concomitent și au control deplin asupra execuției lor;
-execută IO și IPC de rețea;
-controlează subprocese;
-distribuie sarcini prin cozi;
-sincronizează codul concurent.

-threading-

Un fir este un flux de execuție separat.

Clasa Thread reprezintă o activitate care este rulată într-un fir de control separat. Există două moduri de a specifica activitatea: transmițând un obiect care poate fi apelat către constructor sau suprasolicitând metoda run() într-o subclasă. Nici o altă metodă (cu excepția constructorului) nu ar trebui să fie suprascrisă într-o subclasă.

Un lock este o primitivă de sincronizare care nu este deținută de un anumit fir atunci când este locked. Un lock se poate afla într-una din cele două stări, „lock” sau „unlock”. Este creat în starea unlocked. Are două metode de bază, acquire() și release().

O rlock este o primitivă de sincronizare care poate fi dobândită de mai multe ori de

către același fir.

O variabilă de condiție este întotdeauna asociată cu un fel de lock; aceasta poate fi transmisă sau una va fi creată implicit. Transmiterea unei intrări este utilă când mai multe variabile de condiție trebuie să partajeze același lock. Lock-ul face parte din obiectul condiției: nu trebuie să îl urmăriți separat.

Un semafor este una dintre cele mai vechi primitive de sincronizare și gestionează un contor intern care este decrementat de fiecare apel acquire() și incrementat de fiecare apel release().

Un eveniment este unul dintre cele mai simple mecanisme de comunicare între thread-uri: un thread semnalează un eveniment și alte thread-uri îl așteaptă.

-concurrent.futures-

Modulul concurrent.futures oferă o interfață la nivel înalt pentru executarea asincronă a apelurilor.

Execuția asincronă poate fi efectuată cu fire de execuție, folosind ThreadPoolExecutor sau procese separate, folosind ProcessPoolExecutor. Ambele implementează aceeași interfață, definită de clasa abstractă Executor.

ThreadPoolExecutor este o subclasă Executor care folosește un grup de fire pentru a executa apeluri în mod asincron.Deadlock-uri pot apărea atunci când apelul asociat cu un Future așteaptă rezultatele unui alt Future.

Clasa ProcessPoolExecutor este o subclasă Executor care folosește un grup de procese pentru a executa apeluri în mod asincron. Apelarea metodelor Executor sau Future dintr-un apel trimis la un ProcessPoolExecutor va avea ca rezultat un impas.

Clasa Future încapsulează execuția asincronă a unui apelabil. Instanțele viitoare sunt create de Executor.submit().

-subprocess-

Modulul subprocesului vă permite să generați noi procese, să vă conectați la conductele de intrare/ieșire/eroare și să obțineți codurile lor de return. Abordarea recomandată pentru invocarea subproceselor este utilizarea funcției run() pentru toate cazurile de utilizare pe care le poate gestiona.

Laboratorul 10: Corutine - suport nativ pentru paralelism în Kotlin

-Funcții recursive-

Kotlin suportă un stil de programare funcțională cunoscut ca recursivitatea coadă (tail recursion). Aceasta permite unor algoritmi care în mod normal ar fi fost scrișI cu bucle să fie scrișI cu o funcție recursivă, dar fără riscul de „stack overflow”. Spre deosebire de recursivitatea normală în care

toate apelurile recursive sunt efecutate la început și apoi se calculează rezultatul din valorile returnate la urmă, în recursivitatea coadă calculele sunt executate primele, apoi apelurile recursive (apelul recursiv trimite rezultatul pasului curentcătre următorul apel recursiv).

-Coroutine-

În esență, coroutinele sunt fire light-weight, ceea ce înseamnă că durata de viață a noii coroutine este limitată doar de durata de viață a întregii aplicații. Întârzierea (delay) este o funcție specială de suspendare, care nu blochează un fir, ci suspendă coroutina și poate fi utilizată doar dintr-o coroutină.

Dacă vorbim despre concurență structurată, în loc să lansăm coroutine în GlobalScope, la fel cum facem de obicei cu firele de execuție (firele sunt întotdeauna globale), putem lansa coroutine în domeniul specific al operației pe care o efectuăm. Este posibil să vă declarați propriul domeniu de aplicare folosind constructorul coroutineScope. Creează un scop și nu se termină până când nu se finalizează toți copiii lansați.

runBlocking și coroutineScope pot arăta similare, deoarece ambele așteaptă corpul lor și toți copiii să se finalizeze. Principala diferență este că metoda runBlocking pune firul curent pe așteptare, în timp ce coroutineScope doar suspendă, eliberând firul de bază pentru alte utilizări.

Coroutinele sunt o formă de procesare secvențială: doar una se execută la un moment dat. Firele de execuție sunt (cel puțin conceptual) o formă de procesare simultană: mai multe fire pot fi executate la un moment dat.

Laboratorul 9: Design patterns în Python

-Decorator-

Modelul Decorator permite utilizatorului să adauge funcționalități noi la un obiect existent, fără a-i modifica structura. Acest tip de model de design se încadrează în modelul structural, deoarece acest model acționează ca un înveliș pentru clasa existentă.

Acest model creează o clasă decorator, care „înfășoară” clasa originală și oferă o funcționalitate suplimentară, păstrând intactă semnătura metodelor clasei.

Motivul unui model de decorator este să atașeze dinamic responsabilități suplimentare unui obiect.

-Iterator-

Modelul de design iterator se încadrează în categoria de modele de design comportamental. Dezvoltatorii întâlnesc modelul iterator în aproape fiecare limbaj de programare. Acest model este utilizat astfel încât ajută la accesarea elementelor unei colecții (clasă) în mod secvențial, fără a înțelege designul stratului de bază.

-Generator-

Generatoarele reprezintă o modalitate simplă de a crea iteratori, toate cerințele anterioare fiind gestionateîn mod automat de generator.

Definirea unui generator presupune utilizarea cuvântului cheie yield în loc de return.
Acesta trebuie să conțină cel puțin un yield (poate conține mai multe alte yield-uri, return-uri).

Diferența între return și yield:
return - încheie complet execuția funcției
yield - pune pauză funcției, salvând stările și continuând mai târziu de unde a rămas.

Avantajele utilizării unui generator în loc de iterator:
-ușor de implementat
-eficient din punct de vedere al memoriei
-permite reprezentarea unui stream infinit
-generatoarele pot fi folosite pentru a realiza un pipeline cu o serie de operații.

-Fabrică-

Modelul fabrică intră în categoria design pattern-urilor de creație. Oferă unul dintre cele mai bune moduri de a crea un obiect. În modelul fabrică, obiectele sunt create fără a expune logica clientului și se referă la obiectul nou creat folosind o interfață comună.

Modelele fabrică sunt implementate în Python prin metoda fabricii. Tipul de obiect utilizat în metoda fabrică este determinat de șirul care este transmis prin metodă.

-Pipeline-

În programare, o conductă, cunoscută și sub denumirea de conductă de date, este un set de elemente de prelucrare a datelor conectate în serie, unde ieșirea unui element este intrarea următorului. Elementele unei conducte sunt adesea executate în paralel sau în porțiuni de timp.

-Lanț de responsabilități-

Modelul lanțului de responsabilități este utilizat pentru a realiza cuplarea liberă în software unde o solicitare specificată de client este transmisă printr-un lanț de obiecte incluse în acesta. Ajută la construirea unui lanț de obiecte. Cererea intră de la un capăt și se mută de la un obiect la altul.

Acest model permite unui obiect să trimită o comandă fără să știe ce obiect va gestiona cererea.

-Comandă-

Modelul comandă adaugă un nivel de abstractizare între acțiuni și include un obiect, care invocă aceste acțiuni.În acest model de proiectare, clientul creează un obiect de comandă care include o listă de comenzi care trebuie executate. Obiectul de comandă creat implementează o interfață specifică.

-Automat finit de stări-

Metoda stării este un model de design comportamental care permite unui obiect să-și schimbe comportamentul atunci când apare o schimbare în starea sa internă. Ajută la implementarea stării ca o clasă derivată a interfeței cu modelul de stare. Dacă trebuie să schimbăm comportamentul unui obiect în funcție de starea lui, putem avea o variabilă de stare în obiectul respectiv și putem folosi blocul de condiții if-else pentru a efectua diferite acțiunibazate pe stare. Poate fi denumită ca automat de stări orientat pe obiecte. Implementează tranzițiile de stare invocând metode din superclasa modelului.

-Proxy-

Metoda Proxy este un model de design structural care vă permite să furnizați înlocuirea unui alt obiect. Aici, folosim diferite clase pentru a reprezenta funcționalitățile altei clase. Partea cea mai importantă este că aici creăm un obiect care are o funcționalitate originală pe care să o ofere lumii exterioare.

-Cache-

În calcul, un cache este un strat de stocare a datelor de mare viteză care stochează un subset de date, de natură tranzitorie, astfel încât cererile viitoare pentru aceste date sunt furnizate mai repede decât este posibil accesând locația de stocare primară a datelor. Caching-ul vă permite să reutilizați eficient datele preluate anterior sau calculate.

-Strategie-

Modelul strategie este un tip de model comportamental. Principalul obiectiv al modelului strategie este de a permite clientului să aleagă dintre diferiți algoritmi sau proceduri pentru a finaliza sarcina specificată. Diferiți algoritmi pot fi schimbați înăuntru și înafară fără complicații pentru sarcina menționată.

Acest model poate fi utilizat pentru a îmbunătăți flexibilitatea atunci când sunt accesate resurse externe.

Laboratorul 8: Design patterns în Kotlin

-Stare-

Scopul modelului: Permite unui obiect să își modifice comportamentul atunci când starea sa internă se schimbă. Obiectul va părea că își schimbă clasa.

Aplicabilitate: Modelul stare se utilizează în următoarele cazuri:

-Comportamentul unui obiect depinde de starea sa și trebuie să-și schimbe comportamentul în timpul execuției, în funcție de starea respectivă;

-Operațiile au declarații condiționale compuse, mari, care depind de starea obiectului.

Consecințe:
-Localizează un comportament specific stării și comportamentul partițiilor pentru diferite stări;
-Face tranzițiile între stări explicite;
-Obiectele stării pot fi partajate.

-Compus-

Modelul compus este un model de proiectare a compartimentării și descrie un grup de obiecte care este tratat la fel ca o singură instanță a aceluiași tip de obiect. Intenția unui compozit este de a „compune” obiecte în structuri de arbori pentru a reprezenta ierarhii parțiale. Vă permite să aveți o structură de arbore și să cereți fiecărui nod din structura copacului să efectueze o sarcină.

-Fabrică abstractă-

Modelul fabrică abstractă funcționează în jurul unei super-fabrici care creează alte fabrici. Această fabrică mai este numită și fabrică de fabrici. Acest tip de model de design se încadrează în modelul creațional, deoarece oferă una dintre cele mai bune metode de a crea un obiect.

În modelul Abstract Factory, o interfață este responsabilă pentru crearea unei fabrici de obiecte conexe, fără a specifica explicit clasele lor. Fiecare fabrică generată poate oferi obiectele conform modelului Factory.

-Observator-

Modelul de observare este un model de proiectare software în care un obiect, numit subiect, menține o listă a dependenților săi, numiți observatori și îi notifică automat cu privire la orice modificări de stare, de obicei apelând la una dintre metodele lorModelul observator se încadrează în categoria modelului comportamental.

-Memento-

Modelul memento este un model de design comportamental. Modelul memento este folosit pentru a restabili starea unui obiect la o stare anterioară. Pe măsură ce aplicația dvs. progresează, poate doriți să salvați punctele de control din aplicație și să le restabiliți mai târziu la aceste puncte de control.

-Pod-

Modelul pod este utilizat atunci când trebuie să decuplăm o abstractizare de la punerea în aplicare a acesteia, astfel încât cele două să poată varia independent. Acest tip de model de design se încadrează în modelul structural,deoarece decuplează clasa de implementare și clasa abstractă, oferind o structură de punte între ele.

Acest model implică o interfață care acționează ca o punte care face ca funcționalitatea claselor concrete să fie independentă de clasele implementatoare de interfață. Ambele tipuri de clase pot fi modificate structural, fără a se afecta reciproc.

-Constructor-

Modelul constructor construiește un obiect complex folosind obiecte simple și utilizând o abordare pas cu pas. Acest tip de model de design se încadrează în modelul creațional, deoarece oferă una dintre cele mai bune metode
de a crea un obiect.

O clasă Builder construiește final obiectul pas cu pas. Acest constructor este independent de alte obiecte.

-Prototip-

Modelul prototip se referă la crearea unui obiect duplicat, ținând cont de performanță. Acest tip de model de design se încadrează în modelul creațional, deoarece oferă una dintre cele mai bune metode de a crea un obiect.

Acest model implică implementarea unei interfețe prototip care să creeze o clonă a obiectului curent. Acest model este utilizat atunci când crearea obiectului este direct costisitoare. De exemplu, un obiect trebuie creat după o operațiune de bază de date costisitoare. Putem memora în cache obiectul, îi returnăm clona la următoarea solicitare și actualizăm baza de date, după cum este necesar, reducând astfel apelurile la baza de date.

-Fațadă-

Modelul fațadă ascunde complexitățile sistemului și oferă o interfață pentru client care poate accesa sistemul. Acest tip de model de design se încadrează în structura structurală, deoarece adaugă o interfață la sistemul existentpentru a-și ascunde complexitățile.

Acest model implică o singură clasă care furnizează metode simplificate solicitate de către client și apeluri delegați la metodele claselor de sistem existente.

-Mediator-

Modelul mediator este utilizat pentru a reduce complexitatea comunicării între mai multe obiecte sau clase. Acest model oferă o clasă mediator care gestionează în mod normal toate comunicațiile dintre diferite clase și sprijină întreținerea ușoară a codului prin cuplaj liber. Modelul mediator se încadrează în categoria modelului comportamental.

-Adaptor-

Modelul adaptor funcționează ca o punte de legătură între două interfețe incompatibile. Acest tip de model de design se încadrează în modelul structural, deoarece acest model combină capacitatea a două interfețe independente.

Acest model implică o singură clasă responsabilă de unirea funcționalităților interfețelor independente sau incompatibile.

-Singleton(Bonus)-

Modelul singleton este unul dintre cele mai simple modele de design în Java. Acest tip de model de design se încadrează în modelul creațional, deoarece acest model oferă una dintre cele mai bune metode de a crea un obiect.

Acest model implică o singură clasă care este responsabilă de crearea unui obiect, asigurându-se în același timp că este creat doar un singur obiect. Această clasă oferă o modalitate de a accesa singurul său obiect care poate fi accesat direct fără a fi necesar să instaureze obiectul clasei.

-Vizitator(Bonus)-

Modelul vizitator este unul dintre modelele de design comportamental. Este utilizat atunci când trebuie să executăm o operație pe un grup de obiecte similare. Cu ajutorul modelului vizitator, putem muta logica operațională de la obiecte la o altă clasă.

Laboratorul 7: Colecții și generice în Kotlin

-Colecții-

Kotlin face distincție între colecțiile mutabile și cele imutabile. O colecție mutabilă poate fi actualizată pe loc prin adăugarea, ștergerea sau înlocuirea unui element. O colecție imutabilă, deși oferă aceleași operații (adăugare, ștergere, înlocuire) prin funcțiile operator, va crea o nouă colecție, lăsând-o pe cea veche neschimbată.

O **secvență** este un tip iterabil1 pe care se pot efectua operații fără crearea de colecțiiintermediare inutile, prin executarea tuturor operațiilor aplicabile pe fiecare element înainte detrecerea la următorul.

Secvențele sunt lenețe (lazy), funcțiile intermediare corespunzătoare pentru procesareasecvențelor nu fac calcule, ci returnează o nouă secvență ce o decorează pe cea anterioară cu onouă operație. Toate calculele sunt evaluate în timpul operației terminale.

Procesarea secvențelor este în general mai rapidă decât procesarea directă a colecțiilorunde există mai mult de un pas de procesare.
ATENȚIE: Există cazuri în care secvențele nu sunt recomandate. Spre exemplu, în cazulsortării prin apelul funcției*sorted*,întrucât este necesară parcurgerea întregii colecții.

În ceea ce privește stream-urile din Java, acestea sunt tot „lenețe”, fiind colectate înpasul final de procesare. De asemenea, stream-urile Java sunt mult mai eficiente pentruprocesarea colecțiilor decât funcțiile de procesare corespunzătoare din Kotlin.

Diferențe între stream-urile Java și secvențele Kotlin:
secvențele Kotlin au mult mai multe funcții de procesare (fiind definite ca funcțiiextensie)
Stream-urile Java pot fi pornite în mod paralel, utilizând o funcție *parallel.*

Se recomandă utilizarea stream-urilor Java doar pentru procesări computaționale grele,unde se poate beneficia de modul paralel.

-Generice-

Programarea generică (sau genericele) este o tehnică prin care funcțiile pot fi scrise cutipuri care nu sunt specificate în momentul scrierii și sunt folosite mai târziu cu diferite tipuri dedate.

-Polimorfism mărginit-

Funcțiile care sunt generice pentru orice tip sunt utile, dar cumva limitate. Adesea, va finevoie de scrierea unor funcții care sunt generice pentru unele tipuri care au o caracteristicăcomună. Spre exemplu, definirea unei funcții care returnează minimul dintre două valori, pentruoricare valori ce suportă noțiunea de comparare.

Pentru a forța valorile să aibă acea noțiune de comparare, trebuie restricționate tipurilegenerice la cele care suportă funcțiile care trebuie invocate. Cu alte cuvinte, mărginim funcțiapolimorfică (generică), acest lucru fiind numit polimorfism mărginit.

-Mărginiri superioare-

Kotlin suportă un tip de mărginire a polimorfismului, mai precis mărginirea superioară.Din denumire, se remarcă că tipurile generice sunt restricționate la cele care sunt subclase alemărginirii.

Comparable este un tip din biblioteca standardcare definește metoda compareTo cereturnează o valoare mai mică decât 0 dacă primul element este mai mic, mai mare decât 0 dacăal doilea element este mai mic, egală cu 0 dacă elementele sunt egale.

-Mărginiri multiple-

Toate mărginirile superioare sunt scrise ca și clauze where și formează ouniune de mărginire superioară.

-Varianța-

Kotlin oferă:

-varianța la momentul declarării (declaration-site variance)

-proiecțiile de tip

-proiecțiile stea (utilizarea unui argument tip necunoscut)

Laboratorul 6: Utilizarea POO în Python

-Concepte de baza-

Abstractizare: este procesul de grupare de proprietăți/caracteristici esențiale ale unui obiect ignorând detaliile nenecesare.

Încapsulare: constă în separarea aspectelor externe ale unui obiect care sunt accesibile altor obiecte de aspectele interne ale obiectului care sunt ascunse celorlalte obiecte.

Moștenire: este relația dintre clase în care o clasă moștenește structura și comportamentul (funcțiile) definite în una sau mai multe clase.

Polimorfism: descrie conceptul prin care obiecte de diferite tipuri pot fi accesate prin intermediul aceleași interfețe.

Laboratorul 4: Utilizarea principiilor SOLID în Kotlin

-Principiile S.O.L.I.D-

Principiul Responsabilității unice (Single) - O clasă trebuie să aibă o singură rațiune pentru a se schimba.

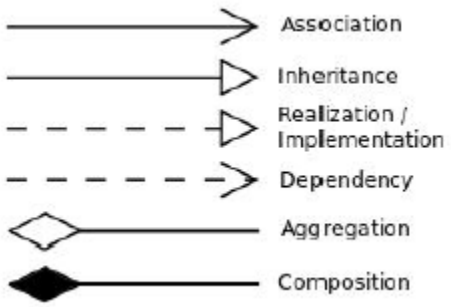
Principiul Închis/Deschis (Open/Closed) - Entitățile software trebuie să fie deschise pentru extindere dar închise pentru modificare.

Principiul substituției Liskov - Copii claselor nu au voie să încalce definițiile de tip din clasa părinte. Adică funcțiile care folosesc referințe la clase de bază, trebuie să poată manipula într-un mod transparent instanțele claselor derivate din acestea.

Principiul separării Interfețelor - 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 - 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. Practic detaliile depind de abstracții, nu invers. Dacă aceasta dependință nu este vizibilă în faza de proiectare atunci ea se construiește.

-Elemente UML-



-Association - relație de asociere între două clase (clasa sursă folosește membri din țintă (target)).

-Dependency - relație de dependență între două clase (clasa sursă folosește clasa din țintă ca un parametru sau ca o variabilă în interiorul funcțiilor).

-Inheritance - moștenire (clasa sursă derivează clasa țintă, adăugându-i noi funcționalități).

-Realization - Implementarea unei interfețe (clasa sursă implementează interfața din țintă)

-Aggregation - agregare (clasa sursă folosește de mai multe ori clasa din target)

-Composition - compoziție (clasa sursă folosește de mai multe ori clasa din țintă și în același timp o derivă (prin moștenire). Un exemplu pentru acest caz este structura arborescentă a unui GUI, unde fiecare element grafic conține un tablou de alte elemente grafice pe care îl poate deriva prin moștenire)