**UNIVERSITATEA „ALEXANDRU IOAN CUZA” DIN IAŞI**

**FACULTATEA DE ECONOMIE ȘI ADMINISTRAREA AFACERILOR**

**SPECIALIZAREA** **INFORMATICĂ ECONOMICĂ**

**Construirea de aplicații în Python folosind UML**

**Profesor Coordonator**, **Student,**

**Prof. Dumitriu Florin Cojocariu Daniel**

**IULIE 2023**

**CUPRINS**

Cuprins

[INTRODUCERE 3](#_Toc138720792)

[CAPITOLUL 1. Caracteristicile limbajului Python 4](#_Toc138720793)

[1.1. Principalele concepte Python 4](#_Toc138720794)

[1.1.1. Sintaxa limbajului Pyton 5](#_Toc138720795)

[1.1.2. Tipuri de date și variabile 5](#_Toc138720796)

[1.1.3. Biblioteci în Python 7](#_Toc138720797)

[1.1.4. Comunitatea Python 7](#_Toc138720798)

[1.2. Conceptele unei aplicații în Python 8](#_Toc138720799)

[1.2.1 Integrated Development Environment (IDE) 8](#_Toc138720800)

[1.2.2 Structura și organizarea codului 9](#_Toc138720801)

[1.2.3 Performanță și optimizare 10](#_Toc138720802)

[1.2.4 Impactul bibliotecilor într-o aplicație Python 11](#_Toc138720803)

[1.3. Avantajele și dezavantajele folosirii Python ca limbaj de programare 12](#_Toc138720804)

[1.3.1. Avantajele folosirii Python 12](#_Toc138720805)

[1.3.2. Dezavantajele folosirii Python 16](#_Toc138720806)

[CAPITOLUL 2. Utilizarea UML în dezvoltarea de software 17](#_Toc138720807)

[2.1. Diagrame UML 17](#_Toc138720808)

[2.2. Elemente de bază UML 20](#_Toc138720809)

[2.3. Diagrama de clasă 21](#_Toc138720810)

[2.3.1. Concepte generale ale unei clase 21](#_Toc138720811)

[2.3.2. Relațiile unei diagrame de clase 22](#_Toc138720812)

[2.4. Diagramă de secvență 24](#_Toc138720813)

[2.4.1. Utilizarea diagramelor de secvențe în aplicații Python. 27](#_Toc138720814)

[2.5. Diagrama cazurilor de utilizare 27](#_Toc138720815)

[2.5.1. Elementele unei diagrame use case 27](#_Toc138720816)

[2.6. Instrumente și biblioteci UML pentru Python. 30](#_Toc138720817)

[CAPITOLUL 3. Dezvoltarea aplicației STANDARD ENTERPRISE PLANNER (SEBP) 31](#_Toc138720818)

[3.1. Prezentarea firmei 31](#_Toc138720819)

[3.2. Prezentarea aplicației Python. 32](#_Toc138720820)

[3.3. Proiectarea aplicației. 32](#_Toc138720821)

[3.3.1. Modelarea funcționalităților aplicației folosind diagrama cazurilor de utilizare 33](#_Toc138720822)

[3.3.2. Modelarea diagramei de clase 34](#_Toc138720823)

[3.3.3. Modelarea funcționalităților aplicației folosind diagrama cazurilor de utilizare 35](#_Toc138720824)

[3.4. Structura aplicației. 39](#_Toc138720825)

[3.5. Procesul de implementare a aplicației 40](#_Toc138720826)

[3.5.1. Crearea interfeței utilizator folosind biblioteca PyQT5. 40](#_Toc138720827)

[3.5.2. Crearea bazei de date folosind biblioteca SQLAlchemy. 41](#_Toc138720828)

[3.5.3. Dezvoltarea aplicației SEBP și relația dintre user interface cu baza de date. 43](#_Toc138720829)

[CONCLUZII 47](#_Toc138720830)

[Bibliografie 48](#_Toc138720831)

# INTRODUCERE

Tema acestei lucrări se fereră la dezvoltarea unei aplicații în limbajul de programare folosind diagrame UML și cum pot fi utilizate aceste diagrame pentru a proiecta un sistem complex. Construirea unei aplicații este un proces complex și necesită o proiectare abstractă folosind UML pentru a evita erorile umane care pot apărea în dezvoltarea unei aplicații. În prezent, majoritea companiilor folosesc aplicații proiectate în diferite limbaje de programare și folosind anumite strategii de implementare. Python este unul dintre cele mai folosite limbaje de programare deoarece oferă un mediu de dezvoltare în care pot fi utilizate diferite instrumente care sporesc timpul de crearea a unei aplicații.

Lucrarea aceasta este structurată în trei capitole, în care sunt explicate conceptele de bază a limbajului de programare Python și a diagramelor UML iar în final prezentarea aplicației STANDARD ENTERPISE BUSINESS PLANNER care este un prototip de aplicație de tip ERP dezvoltată folosind anumite instrumente și tehnici de proiectare.

În primul capitol voi discuta despre limbajul de programare Python, cenceptele acestuia, sintaxa de cod și diferite instrumente folosite pentru a îmbunătății mediul de dezvoltare în cadrul unei aplicații.

Pe lângă acest limbaj de programare voi discuta despre diferite diagrame UML și rolul acestora în proiectarea unei aplicații în capitolul al doilea. Totodată, cum pot fi utilizate acestea pentru a modela un sistem și mări productivitatea unui proiect.

Ultimul capitol constă în prezentarea aplicației STANDARD ENTERPISE BUSINESS PLANNER sau pe scurt SEBP, un prototip de ERP unde sunt implementate difierite interfețe grafice și prezentarea modului în care utilizatorul interacționează cu acestea. Scopul aplicației este de a descrie cum sunt folosite anumite instrumente din bibliotecile disponibile în Python, utilizarea diferitelor concepte de bază a limbajului, structuri de date, metode, clase, subclase și combinarea acestora pentru a obține rezultate dorite.

Pe parcursul prezentării aplicației voi arăta anumite părți din cod, de exemplu crearea claselor pentru baza de date folosind biblioteca SQLAlchemy pentru crearea bazei de date și prezentarea instrumentului Qt Designer din biblioteca PyQT5 care se ocupă cu crearea unui design de bază al interfeței utilizator unde este utilizat ulterior și îmbunătățit cu funcționalități specifice ale unei aplicații de tip ERP.

# CAPITOLUL 1. Caracteristicile limbajului Python

## Principalele concepte Python

Limbajul de programare Python a apărut pentru prima dată în anul 1991 dezvoltat de către Guido van Rossum ca urmare a unei dorințe de a crea un limbaj cu o sintaxă simplă, foarte ușor de citit și accesibil fiecărui programator, developer sau oricărei persoane de rând.[[1]](#footnote-1)

Una dintre caracteristicile principale ale limbajului Python este munca unei singure persoane, față de alte limbaje noi, care sunt create de companii cu un număr semnificativ de angajați. Desigur, meritul dezvoltării acestui limbaj nu aparține doar lui Guido van Rossum, care nu a creat și dezvoltat singur acest limbaj, limbajul în sine a evoluat în decursul a trei decenii, evoluție datorată developerilor, testerilor și utilizatorilor, unde majoritatea nu făceau parte din domeniul IT. Toate aceste persoane cu un statut anonim au contribuit la îmbunătățirea limbajului, însă startul de bază al limbajului se datorează lui Guido Van Rossum, fiind prima persoană care a dezvoltat prima versiune de Python.[[2]](#footnote-2)

Python este folosit de un număr foarte mare de persoane care utilizează dispozitive care folosesc indirect acest limbaj de programare, chiar dacă sunt conștienți sau nu de existența acestuia. În momentul actual sunt sute de milioane de linii de cod scrise care oferă un număr nelimitat de oportunități pentru a dezvolta o aplicație, un script sau algoritm.

Un utilizator cu experiență minimă în domeniul IT, poate să învețe într-un timp foarte scurt Python deoarece documentația limbajului este foarte ușor de înțeles, iar aceasta este dezvoltată și actualizată la zi cu ultimele funcționalități.

Un limbaj de programare, indiferent care este acesta, are ca obiectiv comunicarea cu un dispozitiv de exemplu: calculator personal, smartphone, tablete, echipamente de rețea, sisteme complexe cum ar fi cele de navigație, sisteme industriale etc. Un limbaj, precum Python ne oferă o scurtătură pentru a comunica cu aceste dispozitive, deoarece nu necesită cunoștințe avansate pentru a crea o aplicație, un algoritm sau un sistem în cadrul acestui limbaj. Un exemplu de limbaj mai complex și dificil ar fi C++, care prezintă mult mai multe dificultăți în scrierea unui program sau algoritm deoarece acesta comunică la un nivel avansat cu calculatorul sau alte dispozitive. Pentru a crea o aplicație pe un sistem de operare Windows, Linux, sau macOS folosind C++, persoana care dezvoltă aplicația necesită cunoștințe avansate despre acel sistem de operare: cum funcționează acesta, culegerea de informații, procesarea acestora, și transmiterea unor instrucțiuni care diferă de la un sistem de operare la altul. Python a automatizat aceste procese complicate, indiferent de dispozitivul pentru care este creată aplicația sau orice alt sistem, algoritm etc. În continuare voi discuta despre conceptele care stau la baza limbajului Python.

### Sintaxa limbajului Pyton

Ca limbaj de programare, Python are o sintaxă foarte simplă, minimalistă, nu necesită un număr mare de rânduri sau linii de cod pentru a crea o instrucțiune concisă. Față de alte limbaje de programare, nu este necesară stabilirea tipului variabilei la creare, cum ar fi Integer, String, Dict, Set, List, etc. Atunci când se creează o variabilă, se introduce numele acesteia urmată de un egal și valoarea variabilei respective de exemplu: “**număr = 105**”. În exemplul anterior am atribuit numele “**număr**” variabilei și după am atribuit o valoare care este “**1**”. Se poate vedea cu ochiul liber că variabila “**număr**” este de tip Integer, însă nu a fost nevoie să specificăm acest lucru deoarece acest limbaj își poate da seama ce tip de variabilă este în funcție de valoarea atribuită. În cazuri speciale sau în funcție de preferințele programatorului se poate specifica tipul variabilei la crearea acesteia folosind o funcție de tip constructor care se folosește atunci cand se atribuie o valoare unei variabile: “**număr = int(105)**”, “**preț = float(7.2)**”, “**culoare = str(albastru)**”. În final această metoda nu se practică atunci când se creează o variabilă deoarece nu este nevoie pentru că variabila va avea un tip de date care va fi determinat automat.

Funcțiile sau metodele se definesc la fel de simplu, se introduce keyword-ul sau sintaxa **def** care reprezintă definirea unei funcții înainte de numele funcției urmat de două paranteze în care se află argumentele acesteia, de exemplu:

“**def funcție():**”. După fiecare funcție se introduc două puncte pentru a specifica urmarea blocului de cod care o să se afle în acea funcție. Este nevoie de minim o linie de cod ca metoda respectivă sa funcționeze, în anumite cazuri, se poate introduce doar termenul „**pass**” pentru a avea o funcție care nu procesează sau transmite instrucțiuni.

### Tipuri de date și variabile

În limbajul de programare Python, ca în fiecare alt limbaj sunt puse la dispoziție toate tipurile de date esențiale acestea fiind: numeric (int, float, complex), text (str), secvență (list, range, tuple), mapping (dict), set (set, frozenset), Boolean (bool), Binary (bytes, bytearray, memoryview) și în final None Type.

Pentru toate aceste tipuri de date nu este nevoie de specificarea tipului la creare la fel cum am explicat anterior. Însă avem noi tipuri de date cum ar fi cele de secvență. Atunci când se crează o variabilă de tip listă aceasta trebuie să conțină două paranteze pătratice, iar între aceste paranteze valorile respective de exemplu:

“**numere = [1,2,3,4]**”, o listă are elemente care sunt ordonate, pot fi editate și permit valori egale între ele. Tuple este o listă de variabile care sunt ordonate și nu pot fi schimbate. Pentru a crea o variabilă de tip tuple în loc de paranteze pătrate ca la listă, se folosesc două paranteze normale **„(1,2,3,4)”.** Set este o listă de elemente care sunt neordonate, nu pot fi schimbate și nu sunt indexate. Într-un set se poate adăuga și pot fi șterse valori din listă.

Clasele în Pyhon se definesc cu ajutorul keyword-ului **class** urmat de numele clasei, în final două paranteze, de exemplu: “**class Fructe()**”. După ce a fost creată clasa, se folosește funcția implementată “def \_\_init\_\_()” pentru a inițializa parametrii care o să fie folosiți. Față de alte limbaje de programare cum ar fi Java, nu este nevoie de implementarea unor funcții de tip „set” și “get” pentru a putea accesa și edita parametrii respectivi deoarece fiecare parametru poate fi accesat direct și editat prin scrierea clasei și a parametrului respectiv. Dacă se folosește egal atunci se va edita valoarea parametrului respectiv.

Un exemplu de clasa creată în Python ar arăta în felul următor:

“class Fructe():

def \_\_init\_\_(nume, culoare):

self.nume = nume

self.culoare = culoare”

După cum se vede avem termenul **self** în exemplul anterior care face referire la instanța curentă a clasei și ne oferă acces la parametrul respectiv. În cazul în care se creează alte funcții sau metode în această clasă, fiecare variabilă care nu este la nivel de funcție, adică în codul de bloc al funcției și la nivel de clasă, cum ar fi parametrii definiți în metoda “def \_\_init\_\_()”, variabila poate fi accesată folosind “self.” urmat de numele variabilei. Datorită acestei funcționalități implementate în acest limbaj de programare se evită scrierea unor funcții de tip „set” și “get”.

Sub clasele se definesc la fel ca și clasele, însă între paranteze se adaugă o clasă existentă care va avea rol de părinte pentru clasa respectivă. De exemplu: class B(A). Clasa B este o sub clasă a clasei A. În final structura de creare a unei sub clase este exact aceeași ca la crearea unei clase cu avantajul accesării parametrilor variabilelor din clasa părinte. Un exemplu de folosire a unei sub clase ar arăta în felul următor: o sub clasă de tip tabel care are ca părinte clasa tabel, însă în sub clasa se va modifica funcționalitatea de sortare a tabelului. În acest caz particular, clasa părinte care este un tabel și conține mai multe funcții, una dintre ele fiind funcția de sortare. Pentru a nu crea o clasa de tip tabel și a rescrie toate funcțiile respective, se creează o sub clasă a clasei tabel, care va primi toată structura clasei părinte inclusiv parametrii și funcțiile sale. În sub clasă se va crea din nou funcția de sortare care are același nume cu funcția existentă și se vor implementa modificările dorite, , cum ar fi sortarea numerică. În final avem o clasă părinte care are o funcție de sortare alfabetică și o altă clasă copil care are funcția de sortare modificată pentru a sorta numeric rândurile.[[3]](#footnote-3)

### Biblioteci în Python

O bibliotecă în Python este o colecție de fișiere care conțin blocuri de cod. O bibliotecă are ca obiectiv refolosirea unor funcționalități, clase și funcții pentru a ușura munca unui programator[[4]](#footnote-4). Având în vedere că limbajul de programare a evoluat foarte repede datorită unor număr mare de programatori, în mare parte anonimi care au contribuit fiecare la dezvoltarea limbajului, au fost create biblioteci separate care sunt folosite doar în cazuri particulare, cum ar fi Matplotlib care este o librărie pentru crearea, animarea și interacționarea cu grafice. Această librărie este folosită și instalată în principal doar de către programatorii care se întâlnesc cu grafice în munca lor de zi cu zi.

Există numeroase librării care pot fi instalate, acestea sunt open-source, pot fi modificate, și folosite de către oricine.

### Comunitatea Python

După 30 de ani de evoluție a limbajului Python, acesta a dezvoltat o comunitate foarte mare de programatori, unde fiecare își postează întrebările, nelămuririle legate de acest limbaj de programare. Fiind o comunitate foarte populară există un număr mare de tutoriale, exemple de aplicații și scripturi care ar putea fi accesate de alți programatori care doresc să învețe acest limbaj, având la dispoziție nu doar documentația limbajului, dar și o cantitate suficientă de informații accesibile în format online.

Un alt avantaj având o comunitate mare este progresul continuu cu care se dezvoltă un limbaj de programare, de rezolvarea imediată a bug-urilor existente, până la implementarea unor funcționalități care sunt dorite a fi implementate de către comunitate.

## Conceptele unei aplicații în Python

Pentru crearea unei aplicații trebuie respectate mai multe concepte: organizarea codului, utilizarea corectă a structurilor de date, documentația trebuie să fie concisă și pe înțelesul tuturor indiferent dacă aceasta este scrisă pentru un client, un programator sau un tester. Alte două concepte importante sunt testarea și perfomanța aplicației. Înainte de crearea unei aplicații în Python, este esențială programarea din timp a obiectivelor și cerințelor aplicației, stabilirea funcționalităților, și a modului prin care aceasta va interacționa cu utilizatorul și ce rezultate se așteptă în final. În sub capitolele ce urmează voi discuta în detaliu aceste aspecte.

### Integrated Development Environment (IDE)

Mediul de dezvoltare nu este în sine un concept însă are o influență mare când vine vorba despre crearea unei aplicații. Un IDE ( Integrated  Development Environment ) este un program dedicat dezvoltării software (John M. Zelle , 2003)[[5]](#footnote-5). După cum se poate vedea în numele acestuia, oferă diverse instrumente care îmbunătățesc experiența dezvoltării a unei aplicații. Aceste instrumente pot fi de exemplu o interfață ajutătoare pentru editarea codului, evidențierea greșelilor de sintaxă, completarea automată a keyword-urilor. Pe lângă aceste instrumente, folosirea unui IDE oferă posibilitatea de a folosi funcționalitatea debug pentru a testa și a remedia într-un timp foarte scurt erorile din aplicație.

Nu este necesară folosirea unui IDE pentru a crea o aplicație, este de ajuns să fie folosit doar Python REPL (Read-Eval-Print-Loop). Acesta este o interfață, o linie de comandă (CLI) interactivă care permite introducerea și executarea unor instrucțiuni Python. Însă această metodă de programare nu este practică sau facilă deoarece se pot executa doar instrucțiuni simple, codul nu poate fi organizat și reutilizat ulterior.

Majoritatea programatorilor folosesc un IDE, indiferent de limbajul de programare de exemplu Eclipse pentru Java, Visual Studio Code pentru C, Pycharm pentru Python, fiecare având avantajele și dezavantajele sale.

În limbajul de programare Python, cel mai utilizat IDE este Pycharm deoarece oferă suport integrat pentru gestionarea proiectelor, permițând programatorilor să dezvolte aplicații în mod eficient. Acest IDE vine cu funcționalitatea debug integrată, ce oferă un instrument în plus la depanarea unei aplicații și rezvolvarea într-un timp foarte scurt a erorilor apărute pe parcursul dezvoltării acesteia. Pycharm oferă și un sistem integrat de control a versiunilor. Un astfel de exeplu ar fi Git (GitHub), care permite programatorilor să gestioneze mai ușor diferite versiuni ale aplicației sau ale codului sursă și colaborarea eficientă în echipă. Cu toate aceste funcționalități avansate, Pycharm contribuie la dezvoltarea unei aplicații în mod eficient, fiind un IDE ce oferă utilizatorilor un mediu prietenos, productiv și intuitiv.

### Structura și organizarea codului

Structura și organizarea codului reprezintă aspecte fundamentale în dezvoltarea unei aplicații. O structură bine definită și o organizare eficientă facilitează înțelegerea, întreținerea și extensibilitatea codului.

Pentru a obține o structură clară, este important ca fiecare secvență de cod să fie scrisă în fișiere diferite cum ar fi module sau pachete. Modulele reprezintă fișiere separate care conțin funcționalități individuale și ar trebui să aibă un singur scop pentru fiecare modul în parte. În plus fiecare modul trebuie să aibă o denumire clară care reflectă scopul acestuia.

Organizarea codului în pachete este importantă atunci când aplicația este complexă și necesită o structură avansată. După ce o structură a proiectului a fost stabilită, se va ține cont și de denumirile de clase, variabile sau metode. Acestea trebuie sa aibă un nume sugestiv pentru a înțelege rolul acestora.

Comentariile și documentația au de asemenea un rol important în organizarea codului deoarece comentariile oferă o explicație a codului respectiv. Documentația este esențială pentru ca proiectul să fie înțeles de către utilizator, dar și de programatorii care o să contribuie în viitor la dezvoltarea aplicației.

Nu în ultimul rând, o structură și o organizare bine definită contribuie la eficiența și flexibilitatea dezvoltării. Prin separarea funcționalităților în fișiere diferite se poate reduce reutilizarea codului, modificarea acestuia fără a influența alte funcționalități din proiect. De asemenea o structură bine definită va avea un impact mare legat de crearea unor teste pentru fiecare modul în mod individual și permite adăugarea sau modificarea unor funcționalități noi cu un impact minim asupra aplicației.

### Performanță și optimizare

Într-un proiect complex este importantă performanța aplicației. Dacă aplicația gestionează un volum mare de date aceasta trebuie să proceseze datele respective cât se poate de eficient. Aceste aspecte sunt cruciale când vine vorba despre viteza de rulare, consumul de resurse și experiența utilizatorilor. Pentru a îmbunătăți performanța unei aplicații în Python trebuie urmați mai mulți pași[[6]](#footnote-6)[[7]](#footnote-7):

1. Optimizarea algoritmilor

Algoritmii trebuie analizați și trebuie identificat dacă există posibilitatea utilizării acestora într-o manieră mai eficientă sau dacă este necesară îmbunătățirea codului. Structura proiectului poate să aibă o influență în modul în care un algoritm rulează. De exemplu dacă proiectul nu este structurat corect, un algoritm sau o funcționalitate are de suferit nu doar din punct de vedere al performanței, dar este și mai greu de îmbunătățit deoarece este mult mai greu de înțeles rolul acestuia, rezultatul așteptat sau dependențele sale cu alte funcționalități din proiect.

1. Analizarea aplicației

Se identifică zonele critice ale aplicației care necesită îmbunătățiri, se măsoară timpul de execuție, memoria utilizată și utilizarea altor metode relevante de a se identifica probleme de performanță.

1. Gestionarea eficientă a memoriei

Se asigură dacă aplicația utilizează resursele de memorie într-un mod eficient. Se identifică dacă există scurgeri de memorie, utilizarea unor algoritmi ineficient poate duce la o situație asemănătoare.

1. Testare și benchmarking

Se testează aplicația în mod regulat și se compară performanța cu un test anterior pentru a se identifica anumite probleme din proiect. În general se creează scripturi de testare pentru a testa algoritmi și rezultatul obținut pentru a verifica dacă aceștia rulează într-un mod eficient și dacă necesită o îmbunătățire. Un alt avantaj al creării unor scripturi de teste este verificarea constantă a algoritmilor după parcurgerea unor etape din proiect pentru a verifica dacă anumite funcționalități nu sunt afectate de funcționalitățile noi adăugate.

### Impactul bibliotecilor într-o aplicație Python

Într-o aplicație Python se utilizează doar bibliotecile esențiale care sunt necesare pentru finalizarea și rularea programului respectiv. O bibliotecă este o colecție de module care conțin funcții, clase cu scopul de a fi utilizate de către programatori și de a economisi timp în dezvoltarea unei aplicații. Un exemplu ar fi biblioteca PyQT5 care oferă funcționalități complete ale framework-ului Qt, care este unul dintre cele mai utilizate tool-uri pentru aplicațiile grafice (GUI). Această bibliotecă permite dezvoltarea interfețelor grafice, utilizând o gamă de instrumente precum butoane, frame-uri, tabele etc. O altă caracteristică a acestei biblioteci este suportul pentru a crea un design într-un mod eficient a unei interfețe utilizând teme prestabilite care pot fi personalizate. Pentru a folosi această bibliotecă este necesară instalarea acesteia și utilizarea anumitor clase și funcții care sunt puse la dispoziția programatorului pentru a economisi timp la dezvoltarea unei aplicații[[8]](#footnote-8).

**Avantajele folosirii bibliotecilor în Python sunt:**

* Dezvoltare eficientă a aplicației, folosind o bibliotecă se reduce efortul de dezvoltare, fără a fi nevoie de a construi totul de la 0.
* Comunitate activă ceea ce înseamnă că există multe resurse, documentații și suport pentru utilizarea bibliotecilor în Python.
* Bibliotecile oferă funcționalități dezvoltate de programatori și testate, permițând dezvoltatorilor să reutilizeze codul în loc de dezvoltarea acestuia, astfel dezvoltarea unei aplicații este mai eficientă și mai rapidă.

**Dezavantajele folosirii bibliotecilor în Python sunt:**

* Timpul de învățare. O bibliotecă necesită consultația documentației pentru a înțelege conținutul acesteia, rolul funcționalităților și modul în care sunt folosite.
* Folosirea bibliotecilor poate la necesitatea mai multor dependențe, eventual vor apărea probleme de compatibilitate dacă nu sunt respectate anumite reguli.
* Flexibilitate redusă. Deși o bibliotecă oferă funcționalități preinstalate, acestea pot fi limitate în anumite aspect sau pot avea constrângeri în funcționarea lor.
* Performanța poate să aibă de suferit atunci când se utilizează anumite biblioteci în mod incorect.

## Avantajele și dezavantajele folosirii Python ca limbaj de programare

În contextul actual există o gamă variată de limbaje de programare care pot oferi diferite soluții la problemele care pot apărea la realizarea unei aplicații. Printre aceste limbaje de programare, se pot numără Java, Kotlin, PHP, C#, Swift, Ruby, C, C++, Scala, Matlab și multe altele. Am discutat în subcapitolele anterioare despre principalele concepte Python, care vor fi regăsite și în cadrul acestui subcapitol, deoarece voi discuta în mod detaliat despre avantajele și dezavantajele care apar odată cu utilizarea limbajului Python în conceperea unei aplicații.

Python s-a remarcat și a devenit unul dintre cele mai populare limbaje de programare. El include o sintaxă simplă, codare puțină față de alte limbaje care sunt utilizate la dezvoltarea aplicațiilor. Testarea este mult mai ușoară și atenția cade direct asupra procesului de programare. Vine cu librării incluse pentru orice arie de la statistică până la dezvoltare[[9]](#footnote-9). Printre caracteristicile importante ale limbajului Python se enumeră:

* Suport funcțional și structurat pentru metodele de programare, cât și OOP.
* Poate fi folosit ca un limbaj de scriptare sau poate fi compilat in bytecode pentru a putea construi aplicații de mari dimensiuni
* Oferă tipuri de date dinamice de nivel înalt și oferă suport asupra verificărilor de tip dinamic
* Oferă suport automat asupra garbage collection.
* Poate fi integrat cu ușurință cu C, C++, COM, ActiveX, CORBA și Java[[10]](#footnote-10).

### 1.3.1. Avantajele folosirii Python

Am discutat la nivel general caracteristicile importante ale limbajului Python, dar mai departe voi discuta despre principalele avantaje pentru care acesta este atât de popular pe piața IT și de ce este acesta ales în dezvoltarea aplicațiilor.

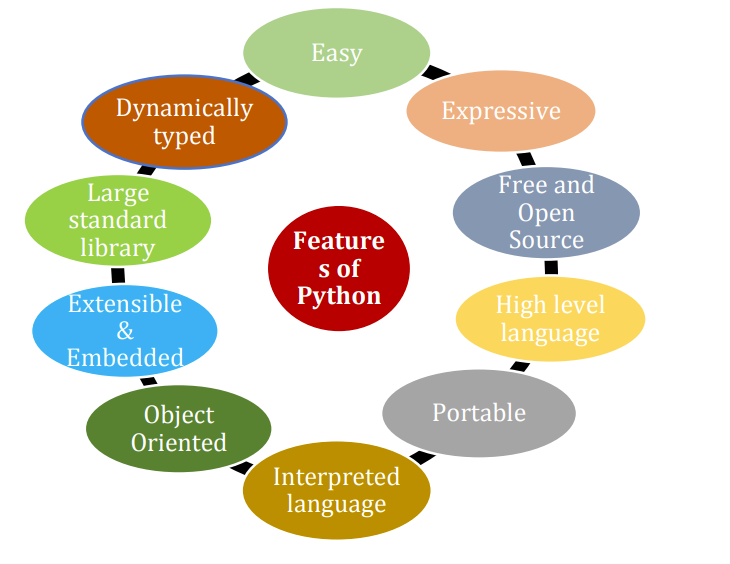


Figure 1. International Research Journal of Engineering and Technology

Sursa: *International Research Journal of Engineering and Technology (IRJET), Comparative Analysis of Python and Java for Beginners*.

Conform figurii nr 1. Sunt enumerate toate avantajele relevante pentru alegerea limbajului Python ca și limbaj principal de programare în dezvoltarea unei aplicații. Aceste avantaje sunt:

1. Ușor de utilizat

Pyhon este ușor codat și citit în comparație cu alte limbaje de programare precum Java, C, C++. Sintaxa Python poate fi studiată de oricine în timpul unei perioade scurte de timp.

1. Expresiv

Poate executa funcții complicate cu puține linii de cod în comparație cu celelalte limbaje de programare. De exemplu, în Pyhon pentru a scrie un script care sa printeze “Hello World” este nevoie doar de o linie de cod: print(‘Hello World’), în schimb pentru a scrie același script în Java este nevoie de un număr mai mare de linii de cod:

Class HelloWorld {

public static void main(String[] args {

System.out.println(“Hello World”);

}

}

1. Gratuit și Open Source

Este un limbaj open source și gratuit. Comunitatea de utilizatori poate asista și poate contribui activ la îmbunătățirea limbajului. Codul sursă poate fi descărcat, modificat, utilizat și distribuit în cadrul comunității.

1. High-level language

Este un high-level language, utilizatorul nu trebuie să țină minte arhitectura și nici nu se confruntă cu probleme legate de managementul memoriei.

1. Portabil

Același program scris în Python poate fi executat în diferite platforme de operare precum Windows, Linux, Unix.

1. Interpreted Language

Codul nu necesita compilare, codurile sunt executate linie cu linie, ceea ce face metoda de debugging mult mai ușoară, dar este încetinit mult mai mult comparativ cu Java.

1. Orientat Obiect

Python susține apropierea orientată pe obiect care ajută programatorul să scrie cod reutilizabil pentru a face o aplicația mult mai simplă.

1. Extensible and embedded

Proprietatea de extindere a Python permite codului să fie scris și compilat în alte limbaje de programare precum C sau C++. Acest cod nu se pierde, deoarece poate fi utilizat în continuare în Python.

1. Librărie Standard Mare

Librăria standard din Python oferă accesul la o gamă largă de module și funcții. Programatorul nu trebuie să scrie cod, acesta poate doar să-l importe.

1. Dynammicaly Typed

Nu este necesară să fie specificat tipul de date al unei variabile când aceasta este declarată. Tipul variabilei respective este decisă în timpul rulării.

1. Artificial Intelligence and Machine Learning

AI și Machine Learning este un subiect dezbătut pe larg în această perioadă și este un topic de interes pentru viitor. Python are un caracter stabil, securizat, flexibil cu o varietate largă de instrumente. ChatGBT permite integrarea chiar în limbajul Python pentru a utiliza diferite funcții precum funcția de traducere a textelor. De asemenea, unele librării Python și frameworks sunt utilizate în AI: SciPy, Pandas, Seaborn, Keras, TensorFlow, Scikit-learn, NLTK, Pytorch, Accord.Net.

1. Desktop GUI applications

Python este utilizat și pentru aplicațiile desktop. Diferite framework-uri și instrumente GUI ajută la crearea funcțională a aplicațiilor desktop.

1. Web scrapping applications

Python poate fi utilizat pentru a extrage o cantitate mare de date dintr-un website pentru a putea crea listări de job-uri, comparări de preț.

1. Data Sciente and Data Visualization

Este preferat pentru analiză și pentru vizualizarea pachetelor mari de date. Este asociat cu mijloace de analizare și de descifrare a datelor complicate de către data scientits. Pachete precum NumPy, Pandas, Sci-Kit sunt larg utilizate[[11]](#footnote-11).

### 1.3.2. Dezavantajele folosirii Python

Am vorbit despre avantaje, dar nu putem uita dezavantajele care pot apărea atunci când utilizăm Python. Printre principalele dezavantaje putem enumera.

* Este mult mai lent în execuție în comparație cu alte limbaje
* Nu este un mediu favorabil pentru dezvoltarea pe mobil
* Nu este preferat pentru sarcini de memorie mare
* Poate aduce erori legate de run time, care duc către erori de design
* Nu este atât de folosit în companiile mari și în afaceri datorită limitărilor de acces asupra nivelurilor din baza de date față de JDBC (Java DataBase Connectivitiy) și ODBC (Open DataBase Connectivity)
* Global Interpreter Lock al Python permite executarea unei singure thread la un moment.
* Simplicitatea acestuia apare ca un dezavantaj când programatorul dorește să se mute către alte limbaje precum Java, care au o structură strictă[[12]](#footnote-12).

# CAPITOLUL 2. Utilizarea UML în dezvoltarea de software

Unified Modeling Language sau pe scurt UML este un limbaj de modelare vizual pentru proiectarea sistemelor software. Acesta nu este un limbaj de programare însă există aplicații software care pot genera blocuri de cod în numeroase limbaje de programare cum ar fi Java, C++, C# și Python. UML a apărut în anul 1990 având ca scop modelarea grafică în domeniul dezvoltării software. Conceptul UML derivă din împreunarea a trei metode de modelare, acestea fiind OMT (Object Modeling Technique), OOSE (Object Oriented Software Engineering) și OOA (Object-Oriented Analysis). Fuzionarea acestor trei metode au avut ca rezultat apariția limbajului UML.[[13]](#footnote-13)

În ziua de azi, UML este folosit pentru modelarea sistemelor informaționale pentru a dezvolta aplicații software și nu în ultimul rând generarea automată de cod.

## 2.1. Diagrame UML

Diagramele sunt instrumente care permit reprezentarea și modelarea unui sistem software cum ar fi comportamentul, structura, funcționalitatea dintre componentele sale. Aceste diagrame oferă un nivel de abstractizare care facilitează analiza unui sistem, oferind o perspectivă vizuală asupra acestuia.[[14]](#footnote-14)

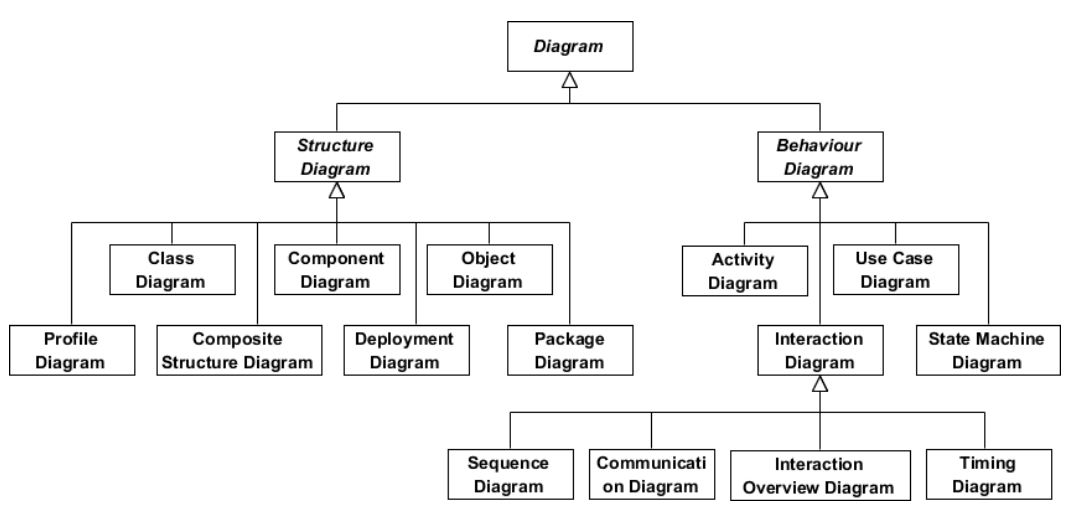


Figure 2. Ierarhia diagramelor UML

Sursa: *Teilans, Artis & Kleins, Arnis & Merkuryev, Yuri & Grinbergs, Andris. (2008). Design of UML models and their simulation using ARENA. WSEAS Transactions on Computers Research. 3.*

După cum se poate vedea în Figura 2. Ierarhia diagramelor UML, există o ierarhie legată de nivelul tipurilor de diagrame și legăturile dintre acestea. Diagrama din vârful ierarhiei care este cunoscută sub numele de diagramă de pachete (Package Diagram) și oferă o vizualizare generală asupra structurii și a organizării celorlalte diagrame. Evident, pentru crearea unui sistem nu este necesar ca acesta să conțină toate digramele, de exemplu este de ajuns să fie folosită diagrama de use case și cea de activitate pentru un sistem de analiză deoarece diagrama de use case are ca rol identificarea și definirea interacțiunilor între utilizator și sistemul de analiză oferind o perspectivă globală asupra comportamentului sistemului, pe de altă parte, diagrama de activitate oferă o înțelegere mai ușoară asupra fluxului de lucru și ilustrează acțiunile care sunt executate în cadrul sistemului.

În cadrul diagramelor UML, un concept foarte important este abstractizarea care permite evidențierea aspectelor esențiale și semnificative ale unui sistem, ignorând anumite detalii irelevante. Se pot evidenția anumite aspect cheie și relații fundamentale fără a se concentra asupra tuturor detaliilor ale implementării. [[15]](#footnote-15)

Alte concept importante privind diagramele UML sunt:

1. Simplicitate

O diagramă UML trebuie să fie ușor de înțeles, evitând utilizarea detaliilor irelevante. Principalul obiect este de a comunica concepte într-un mod concis și de a facilita colaborarea între persoanele care participă la dezvoltare a proiectului

1. Reutilizare

Prin proiectarea diagramelor UML într-un mod modular, se pot dezvolta componente reutilizabile și care pot fi integrate în diferite sisteme.

1. Comunicare

Diagrama UML este un instrument care oferă o comunicare eficientă între participanții proiectului permițând o înțelegere comuna a sistemului.

1. Documentare

Diagramele UML servesc ca instrumente de documentare care prezintă informații despre sistem cum ar fi arhitectura și comportamentul acestuia.

1. Analiză și proiectare

Legat de analiza și proiectarea unui sistem, folosind UML se permite proiectarea sistemelor complexe într-un mod sistematic, oferind un cadru vizual pentru a identifica și rezolva probleme de proiectare, precum și pentru a identifica dependențele dintre componentele sistemului.

1. Testare și validare

Utilizând diagrame UML se poate testa și valida în mod eficient un sistem. Se definesc scenarii de testare pentru a verifica comportamentul sistemului în diferite situații particulare.[[16]](#footnote-16)[[17]](#footnote-17)

După cum se poate vedea în Figura 2. Ierarhia diagramelor UML, există un număr mare de diagrame fiecare având un rol specific. În funcție de sistemul proiectat se folosesc anumite diagrame. În contextul unei aplicații ERP (Enterprise Resource Planning), sunt cinci diagrame care se folosesc în general la proiectare: Use Case, Activity, State, Class și Sequence.

Fiecare dintre cele cinci diagrame sunt esențiale în proiectarea unui ERP, diagrama Use Case este folosită pentru a evidenția funcționalitățile aplicației din perspectiva utilizatorului, diagrama de clasă descrie structura sistemului și modul în care interacționează clasele sau obiectele între ele, dar și cum sunt organizate în cadrul sistemului. În continuare voi prezenta clase folosite în aplicația creată în studiul de caz de [la Capitolul 3](#_CAPITOLUL_3._STUDIU).

### 2.2. Elemente de bază UML

Unified Modeling Language oferă o modalitate de a comunica și reprezenta anumite aspecte ale unui sistem software. Pentru a implementa acest limbaj se utilizează diverse elemente specializate, având fiecare un scop în contextul diagramelor UML. Aceste elemente pot fi impărțite pe categorii și sunt următoarele:

1. Class

Clasa este un concept general, care definește o clasă și caracteristicile sale. O clasă conține atribute, metode, relații și constructori. Atributele reprezintă variabile de instanță care descriu o stare a obiectelor și definesc diferite tipuri de variabile de exemplu șiruri de caractere sau booleane. Relația este un termen specific unei clase care permite reprezentarea unei legături dintre clase folosind diferite metode cum ar fi asocieri, agregări, moștenire și compoziții.

În final clasa poate avea metode care ilustrează comportamentul sau acțiunile unor obiecte. Metodele pot accesa sau manipula atributele unei clase, dar și să interacționeze cu alte obiecte.

1. Object

Un obiect este un concept specific sau o instanță a unei clase care are caracteristici definite de clasa din care face parte. Fiecare obiect are un atribut de valoare care ilustrează starea sau proprietatea acestuia. La fel ca în cazul claselor, un obiect poate să aibă o relație cu altul.

1. Relații

O relație se regăsește în clase, dar și în obiecte având ca rol ilustrarea unei conexiuni între diferite clase sau obiecte. Sunt mai multe tipuri de relații având fiecare un rol specific:

* Asocierea este o legătura între doua sau mai multe elemente, această relație poate să aibă un singur sens, de la o clasă la alta sau două sensuri.
* Agregarea este o relație între doua elemente în care unul dintre elemente este un întreg iar celălalt element o componentă. Această relație este folosită atunci când o clasa conține mai multe componente care formează clasa respectivă.
* Compozit este o relație folosită pentru a elimina elementele legate de o anumită clasa atunci când clasa respectivă este distrusă.
* Moștenirea indică faptul că clasa copil va moșteni toate proprietățile clasei părinte.
* Dependencies (dependință) este o relație în care un element depinde de altul.[[18]](#footnote-18)

Toate aceste elemente sunt fundamentale în limbajul UML fiecare având un rol unic, împreună formează o modalitate de vizualizare și înțelegere a sistemului, modul în care funcționează și interacționează cu utilizatorul.

### 2.3. Diagrama de clasă

Diagrama de clase este foarte populară în contextul modelării unui sistem, aceasta arată proprietățile și relațiile dintre obiecte. O diagramă de clase este concepută din mai multe elemente: clase, obiecte și relații

### 2.3.1. Concepte generale ale unei clase

O diagramă de clase conține entități din sistem și fiecare entitate are un număr de atribute și metode. Din punct de vedere vizual o entitate de tip clasă este reprezentată sub forma unui chenar împărțit în trei secțiuni, prima secțiune conține numele clasei, a doua secțiune atributele clasei iar în ultima secțiune se află metodele.

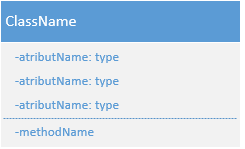


Figure 3. Exemplu clasă

După cum se poate vedea în Figura 3. Exemplu clasă, clasa are trei atribute, fiecare având un tip specific și o metodă. Nu există o limită de atribute sau metode într-o clasă. Atributele sunt concepte care descriu proprietatea unei variabile care face parte din clasa respectivă. Un atribut este conceput din trei elemente distincte: vizibilitate, denumire și tipul acestuia. Vizibilitatea unui atribut indică dacă este public sau privat cu ajutorul semnului plus respectiv minus. Există patru tipuri de vizibilitate a unui atribut acestea fiind: public, private, protected, package sau default. Pentru ca un atribut să fie public acesta trebuie sa aibă un plus în fața denumirii acestuia, atributul este accesibil din orice altă clasă, se poate accesa direct sau prin alte metode publice. La fel ca în cazul atributului public, cel privat este reprezentat de un minus și acesta este accesibil doar în cadrul clasei, inclusiv clasele derivate. Acesta nu poate fi accesat direct din afara clasei însă poate fi modificat cu ajutorul metodelor publice ale clasei. Atributul protejat este asemănător cu cel privat însă poate fi accesat și modificat din alte clase derivate, unde atributul privat nu poate fi modificat. Dacă nu se află niciun semn, implicit atributul este public.

Metodele sunt funcții în cadrul clasei în care sunt menționate și sunt utilizate pentru a realiza operații asupra atributelor din clasa respectivă. În general aceste metode sunt utilizate pentru a seta sau obține valorile atributelor, dar există metode specifice unei clase care realizează anumite operații specifice. La fel ca în cazul atributelor o metodă poate fi publică, privată sau protejată. Metodele sunt utile pentru că permit definirea unei interfețe clare, coerente și ajută la organizarea codului.[[19]](#footnote-19) [[20]](#footnote-20)

### 2.3.2. Relațiile unei diagrame de clase

Anterior am discutat despre elementele unei diagrame de clase și ce atribute conține fiecare. O diagramă de clase conține mai multe entități sau obiecte care sunt legate fiecare prin diferite relații. Folosind diagrame de clase putem reprezenta vizual un sistem și permite înțelegerea rapidă, clară a modului în care clasele din sistem interacționează între ele. În cadrul unui proiect în care există un grup de persoane care participă la dezvoltarea unui sistem, folosirea diagramei de clase ajută la comunicare și la colaborarea cu persoane terțe cum ar fi clienți, manageri de proiect etc. Din punct de vedere al proiectării, utilizarea diagramelor permite o planificare sistematică ce oferă o bază solidă în luare deciziilor și definirea interfețelor reducând riscul de erori, îmbunătățind calitatea proiectului. Cum am explicat anterior, diagramele oferă o perspectiva de ansamblu asupra sistemului, ca rezultat procesul de mentenanță și modificare a proiectului este eficient luând în calcul impactul modificărilor asupra claselor și a relațiilor din sistem.[[21]](#footnote-21)

Relația de asociere este o legătura între două sau mai multe clase, aceasta este reprezentată printr-o linie clară care conectează doua clase. În cazuri particulare pe linia respectivă există o denumire, dar nu este obligatoriu.

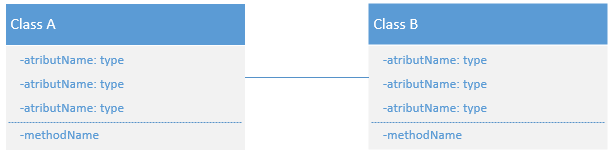


Figure 4. Exemplu asociere între clasa A și clasa B

Agregarea este o relație care indică faptul că o clasă este compusă din mai multe componente, clasa care conține elementele este denumită clasă de bază, iar cele care fac parte din clasa de bază se numesc componente. Pentru a defini o relație de tip agregare se leagă două clase printr-o linie unde la capătul acesteia este un diamant.

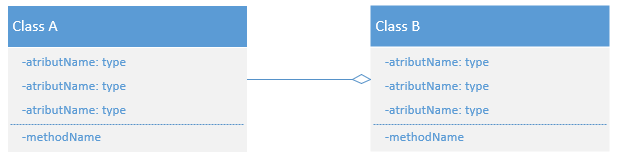


Figure 5. Exemplu relație de agregare între clasa A și clasa B

În Figura 5. Exemplu relație de agregare între clasa A și clasa B avem un exemplu de relație de agregare între clasa A și clasa B, unde clasa B este o componentă a clasei A sau în alți termeni, clasa B este o clasă agregată pentru clasa A. Utilizarea relațiilor de agregare oferă o flexibilitate asupra gestionării independentă a claselor agregate.

Moștenirea este o relație între două clase sau mai multe unde o clasă părinte poate să aibă mai mulți copii care moștenesc atributele clasei părinte. Pentru a crea o astfel de relație se folosește de o săgeată cu un singur sens. [[22]](#footnote-22)

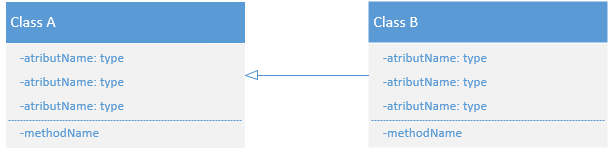


Figure 6 Exemplu relație de moștenire între clasa A și clasa B.

Această relație este utilă deoarece la fel ca la relațiile interioare, sistemul poate fi structurat într-un mod sistematic și reduce complexitatea acestuia.

Relația de dependență este asemănătoare cu cea de moștenire când vine vorba despre săgeată, aceasta fiind o săgeată cu capăt deschis iar linia este punctată. Această relație poate să apară atunci o clasă primește ca parametru un obiect al altei clase cea ce va genera o dependență dintre cele două entități.

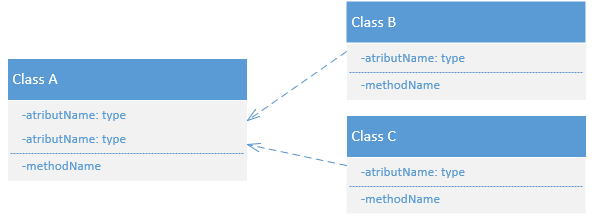


Figure 7. Exemplu de dependență între clasele B, C cu A.

### 2.4. Diagramă de secvență

În UML sunt diferite diagrame de interacțiune, cea mai populară fiind diagrama de secvență unde această diagramă explică în scenarii specifice interacțiunea dintre obiecte în ciclul de viață al acestora. În această diagramă există elemente specifice având fiecare un rol unic.

1. Obiect. Un obiect un diagrama de secvență reprezintă o clasă sau o entitate care implică cum va interacționa cu sistemul în ciclul său de viață.
2. Timpul necesar de completare a unui proces este reprezentat printr-un dreptunghi.
3. Actor. În general un actor este o entitate care interacționează cu sistem, această entitate poate să fie internă sau externă. Având ca reprezentare vizuală un element care seamănă cu o persoană.

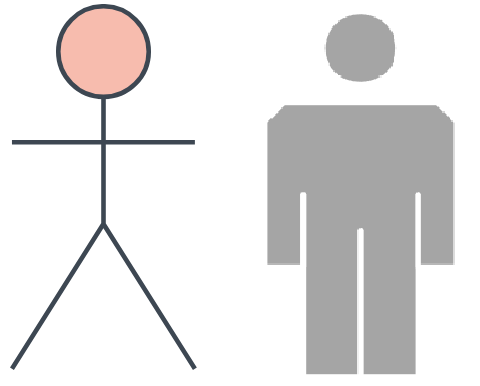


Figure 8. Două exemple de reprezentare vizuală a unui actor.

Sursă. *Lucidchart*, https://www.lucidchart.com/pages/uml-sequence-diagram

1. Mesaje. Săgețile sunt utilizate pentru a reprezenta mesaje sau interacțiuni între obiecte. Există două tipuri de săgeți care sunt utilizate în general într-o diagramă de secvențe: mesaj și returnare. Săgețile de tip mesaj reprezintă mesajele transmise între obiecte. Sunt reprezentate vizual printr-o săgeata și la utilizarea acestora indică o invocare a unei metode sau acțiuni de către un obiect. Fiecare săgeată are un punct de pornire și unul final. Săgeata de tip returnare, aceasta este reprezentată vizual printr-o săgeata punctată, este utilizată pentru a indica returnarea unui rezultat de la obiectul destinatar înapoi la obiectul sursă. Săgeata de returnare are sens invers față de cele de tip mesaj. Folosirea acestor săgeți permit o reprezentare vizuală eficientă a interacțiunilor dintre obiecte.

În final săgeata de tip mesaj poate fi de două feluri: sincronizată și nesincronizată. Mesajele sincronizate au o săgeată cu capăt plin, iar cele nesincronizate au un capăt gol. Se poate vedea clar în figura de mai jos.

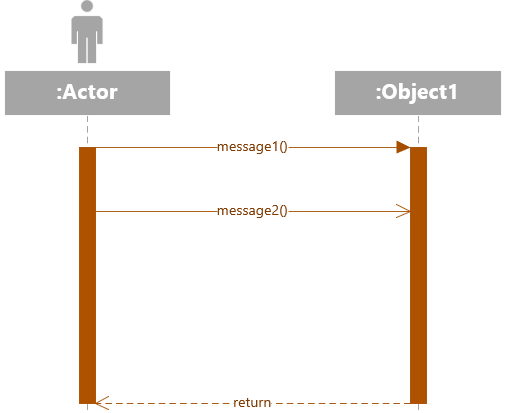


Figure 9 Exemplu de creare a unei diagrame de secvență.

Însă, deoarece figura 9 este o diagramă foarte simplă, în figura 10 se poate vedea o diagramă complexă pentru un ATM.

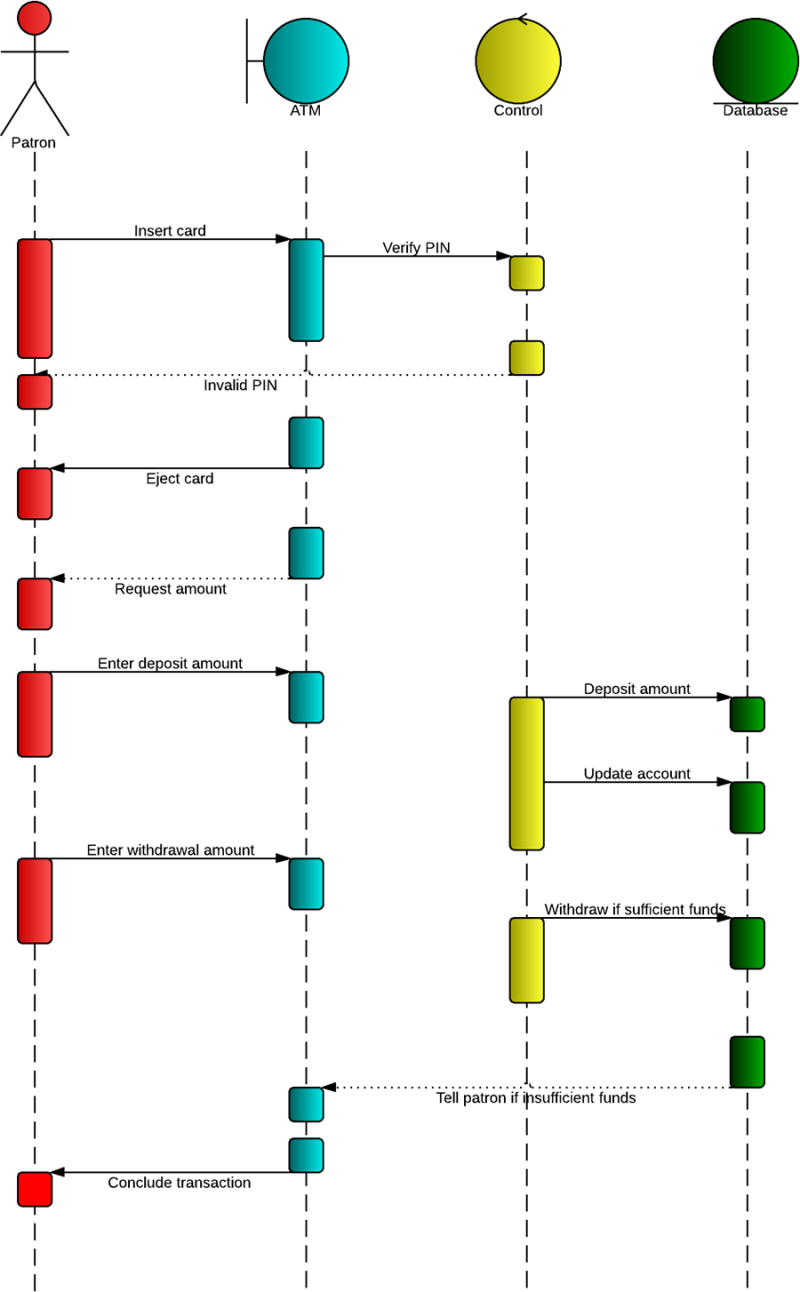


Figure 10. Exemplu diagramă de secvență pentru un ATM.

Sursă: <https://www.lucidchart.com/pages/uml-sequence-diagram> - **Sequence diagram for ATM systems**

### 2.4.1. Utilizarea diagramelor de secvențe în aplicații Python.

Există multe beneficii pentru utilizarea unei diagrame de secvență într-o aplicație Python. În primul rând utilizarea acestei diagrame ajută la înțelegerea aplicației, a fluxului de execuție în care obiectele interacționează între ele. Un alt argument este testarea și validarea logică pe care diagramele de secvență le oferă, se pot evidenția anumite probleme sau interacțiuni ilogice care au loc în sistem și având la dispoziție toate interacțiunile se poate găsi o soluție într-un timp scurt.

Diagrama de secvență poate fi utilizată și ca documentație pentru dezvoltatori și echipa de testare unde nu este nevoie de înțelegerea în mod concis a blocurilor de cod scrie în limbajul de programare Python atât timp cât diagrama este corectă și conține toate interacțiunile dintre componente și actori.[[23]](#footnote-23)

## 2.5. Diagrama cazurilor de utilizare

Utilizarea unei diagrame de tip use case, la fel ca în cazul celor două diagrame prezentate anterior, principalul obiect este înțelegerea interacțiunilor interne sau externe cu utilizatori, alte sisteme sau dispozitive. Această diagramă poate fi utilizată pentru a reprezenta obiectivele de viitor ale sistemului legat de interacțiunea cu utilizatorii. Un alt scop îl mai are definirea și organizarea funcționalităților din sistem într-o manieră concisă[[24]](#footnote-24). Cu toate acestea sunt și dezavantaje în utilizarea acestei diagrame, poate deveni foarte complexă în cazul unui sistem și există limitări legat de exprimarea detaliată a logici și algoritmilor specifici aplicației.

Crearea unei diagrame de use case este asemănătoare ca cea de secvență, în această diagramă avem actori, acțiuni și obiective.

### 2.5.1. Elementele unei diagrame use case

În diagrama use case sunt trei elemente de bază care sunt folosite pentru proiectare:

1. Cazuri de utilizare (Use cases). Este un element cu o formă ovală care reprezintă diferite cazuri utilizate având un text în mijlocul formei.
2. Actori. Actori sunt reprezentați printr-o formă asemănătoare cu cea din Figura 8. Două exemple de reprezentare vizuală a unui actor unde aceștia sunt utilizatori care interacționează cu sistemul. Rolul acestora este de a iniția și de a participa la cazurile de utilizare ale aplicației sau pot fi destinatari și să primească un anumit rezultat.
3. Asociere. Cazurile și actori sunt asociați prin anumite legături. O asociere directă este reprezentată printr-o linie continuă sau completă dintre actor și a cazului de utilizare având o interacțiune directă. Asocierea indirectă este reprezentată printr-o linie punctată care semnifică o interacțiune indirectă sau intermediată prin alte componente. Această asociere indirectă are două utilizări, anume de includere și de extindere unde relația de includere indică faptul că cazul respectiv este esențial fluxului de execuție unde se adaugă funcționalități suplimentare. Cea de extindere reprezintă un caz de utilizare care poate extinde sau completa alt caz, iar acestea sunt opționale. O asociere indirectă de tip inclus este reprezentată printr-o linie punctată și având ca mesaj pe linia respectivă “<<include>>”, iar pentru linia de tip extinde se notează la fel ca cea de tip inclus însă mesajul este diferit “<<extended>>”.[[25]](#footnote-25)

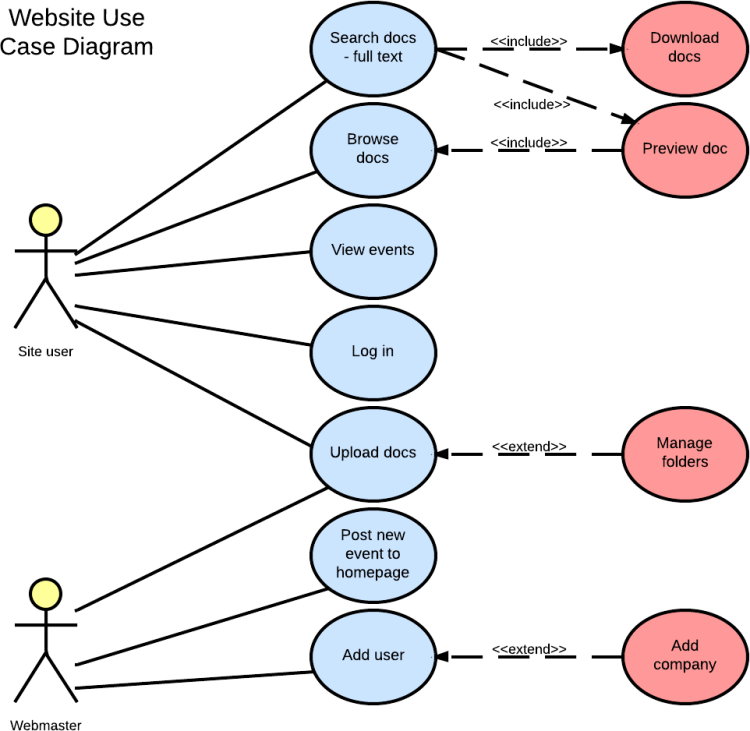


Figure 11. Exemplu de diagramă use case.

Sursa: [*https://www.lucidchart.com/pages/uml-use-case-diagram*](https://www.lucidchart.com/pages/uml-use-case-diagram) *-* ***Use case diagram components***

După cum se poate vedea în Figura 11. Exemplu de diagramă use case este o diagramă de use case unde sunt reprezentate interacțiunile și dependențele dintre actori și cazurile de utilizare. Se poate vedea clar că **Site user** interacționează cu cinci cazuri, iar **webmaster** interacționează doar cu trei.. Cazuril download docs si preview doc sunt cazuri incluse, se poate vedea prin asocierea dintre ele având pe linie textul “<<include>>”, iar cazurile manage folder și company sunt cazuri extinse.

## 2.6. Instrumente și biblioteci UML pentru Python.

Utilizarea de instrumente sau biblioteci UML în Python oferă mijloace prin care se pot crea sau gestiona diagrame UML care facilitează procesul de dezvoltare a unei aplicații în Python. Generarea automată a diagramelor din codul sursă Python este cel mai mare avantaj pe care îl au bibliotecile respective pentru că nu mai este necesar de a proiecta diagrame manual și se pot identifica probleme în aplicație într-un mod eficient în cazul în care rezultatele obținute nu sunt corecte. Însă acesta nu este singurul avantaj, se pot genera blocuri de cod folosind diagrame UML urmat de testarea sistemului sau adăugarea unor noi funcționalități.

Există numeroase biblioteci UML disponibile în Python.

1. PyNsource. Permite generarea automată diagramelor UML din codul sursă Python. Folosind această librărie se pot genera diagrame de clase, secvență, state și activitate.
2. PyUML. Este o librărie open source foarte simplă care oferă generare de cod folosind doar diagrama de clase.
3. Epydoc. Generează documentație UML din codul sursă din python.
4. Lumpy. Este o librărie complexă ce generează doar diagrame de obiecte și clase din codul sursă Python. Principala funcționalitate a acestei librării este analiza codului sursă și de a genera o reprezentare vizuală a importurilor între module. Acest lucru este util pentru a înțelege în ansamblu structura aplicației.
5. Pyreverse. Această librărie face parte din setul de instrumente pyling pentru analizarea și verificarea codului Python. Permite și generarea diagramelor UML, diagrame de clase și obiecte.
6. Pywebuml. Permite generarea automată a diagramelor UML din codul sursă special pentru aplicații web. Această librărie analizează structura proiectului și extrage informațiile necesare despre obiecte, clase, atribute și metode. Diagramele care pot fi generate folosind biblioteca Pywebuml sunt diagramele de clase, pachete, dependență, stare, activitate, iar in final se pot genera și diagrame personalizate specifice aplicației.
7. Visual Paradigm. Este o librărie de proiectare a diagramelor UML în diferite limbaje de programe inclusiv Python. Se pot genera diagrame din cod sursă dar și viceversa. Diagramele care pot fi generate din codul sursă sunt diagramele de clase, pachete, secvență și activitate. Folosind această librărie se poate genera cod sursă pentru Pytthon doar din două diagrame, cea de clase și activitate.[[26]](#footnote-26)

# CAPITOLUL 3. Dezvoltarea aplicației STANDARD ENTERPRISE PLANNER (SEBP)

În acest capitol voi discuta despre proiectarea unei aplicații de tip ERP în Python. Scopul acestui studiu de caz este de a dezvolta un sistem software care să satisfacă nevoile și cerințele de bază ale unei afaceri folosind limbajul de programare Python și a diagramelor UML. Scopul aplicației este de a integra și automatiza resursele unei întreprinderi de exemplu gestionarea clienților, vânzărilor și a altor resurse. Unul dintre principalele avantaje ale acestei aplicații este eliminarea greșelilor umane și îmbunătățirea productivității de muncă.

## 3.1. Prezentarea firmei

Aplicația despre care vom discuta și pe care o vom implementa în cadrul capitolului 3. STUDIU DE CAZ – CONSTRUIREA APLICAȚIEI PYTHON – STANDARD ENTERPRISE BUSINESS PLANNER (SEBP) a pornit de la nevoia companiei Romchim Protect S.A de a-și automatiza procesul de afaceri. Având sediul social în județul Bacău, sat Filipești, Romchim Protect S.A este o societate pe acțiuni care a fost înființată în anul 1998. Această companie are ca principal obiect de activitate producerea vopselelor lavabile și a ignifugaților pentru construcții din material lemnos.

Aflată în continuă dezvoltare pe piața Românească, atât activitatea de producție cât și cifra de afaceri a companiei a crescut semnificat de la an la an, ajungând la procesul de diversificare a gamei de produse. Astfel, compania este specializată în producerea unei palete mult mai largi de produse chimice precum agenți de degivrare pentru aeronave, soluții de combatere a mucegaiului, soluții dezinfectante, etc. Aceste produse sunt comercializate unor clienți mari de pe piața românească precum Dedeman și principalele aeroporturi din țară. Pe lângă acești clienți se pot adăuga și persoane fizice cât și clienți mulți mai mici.

În ciuda gamei diversificate de produse, compania este încă una de mici dimensiuni din punct de vedere al angajaților, având 35 angajați în 2023, dar din punct de vedere al cifrei de afaceri este o firmă medie. [[27]](#footnote-27)

Până în prezent, compania a utilizat soluții software precum Excel și Saga pentru a stoca datele relevante în format electronic, cât și utilizarea de documente în format fizic. Procesul business a devenit tot mai greu pentru companie, neputând prelua în timp util diverse documente și neputând păstra relația cu clienții și furnizorii în mod direct. Pandemia și războiul din Ucraina au reprezentat în acest context încă un factor prin care compania a pierdut contactul cu mare parte din furnizorii de materie primă, fapt ce o împinge tot mai mult spre dorința implementării unei aplicații interne pentru gestionarea produselor, lanțului de aprovizionare și logistic cât și gestionarea relației cu clienții săi. Pe lângă aceste motive, dorința de automatizare a proceselor reprezintă și o întărire a poziției pe piața din România.

## 3.2. Prezentarea aplicației Python.

Aplicația Standard Enterprise Business Planner sau pe scurt SEBP este un sistem software de tip ERP care are trei module funcționale: vânzări, clienți și produse. Prin această aplicație prezint un prototip de ERP în care sunt gestionate anumite resurse, mai exact cele trei module prezentate anterior. Acest instrument software este proiectat cu ajutorul diagramelor UML (diagramă de clasă, secvență și use case), și în cadrul creării aplicației se folosește biblioteca SQLAlchemy pentru a putea realiza o conexiune cu baza de date . O altă librărie folosită este PyQT5 care ajută la crearea unei interfețe utilizator[[28]](#footnote-28). Această librărie oferă și un instrument de crearea unei interfețe din punct de vedere vizual asemănător cu cea din limbajul de programare C# în Microsoft Visual Studio. În final după ce s-a modelat schița unui design se salvează într-un fișier cu extensia “.ui” care este un fișier de interfață utilizator. Acest fișier este ulterior convertit în limbaj Python folosind o comandă în terminal.

## 3.3. Proiectarea aplicației.

Primul pas pentru proiectarea sistemului este identificarea cerințelor aplicației, aceasta fiind crearea a trei module pentru gestionarea vânzărilor, produselor și a clienților. Următorul pas este modelarea diagramelor UML, se modelează diagrama de use case pentru a se identifica funcționalitățile aplicației și cum interacționează utilizatorul cu acestea. Următoarea diagramă folosită este cea de clase și are ca scop stabilirea legăturilor și relațiilor între clase. În final ultima diagramă folosită, cea de secvențe care descrie interacțiunile între obiectele aplicației în fluxul de viață ala acestora.

## 3.3.1. Modelarea funcționalităților aplicației folosind diagrama cazurilor de utilizare

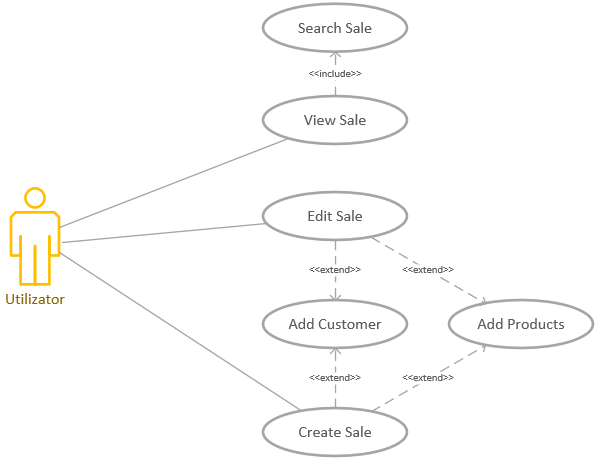


Figure 12 Diagramă de use case pentru interfața vânzări.

Această diagramă ilustrează cazul particular în care utilizatorul se află în meniul de vizualizare a unei vânzări și are mai multe alegeri pe care le poate face. Poate să creeze o vânzare nouă sau să vizualizeze una deja existentă. Daca vizualizează una deja există poate să o editeze, salveze sau anuleze modificările făcute unde este inclus automat cazul de închidere a vânzării. Dacă utilizatorul alege să creeze o vânzare procesul este asemănător, se salvează sau anulează modificările făcute.

O altă diagramă de tip use case pentru interfața client este foarte asemănătoare cu cea de vânzare unde utilizatorul are opțiunea de a crea sau edita un client, în final să salveze modificările respective sau să le anuleze.

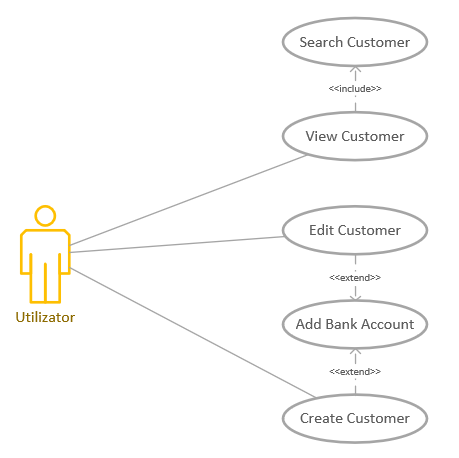


Figure 13 Diagramă use case pentru interfața customer.

## 3.3.2. Modelarea diagramei de clase

Modelarea diagramei de clase constă în cinci clase diferite, Customer, Adress, BankAccount, Sales și Product, iar în final o clasă intermediară dintre Sale și Product care conține un număr de produse care se află într-o vânzare. Între toate aceste clase există diferite tipuri de relații unde se pot vedea în figura următoare.

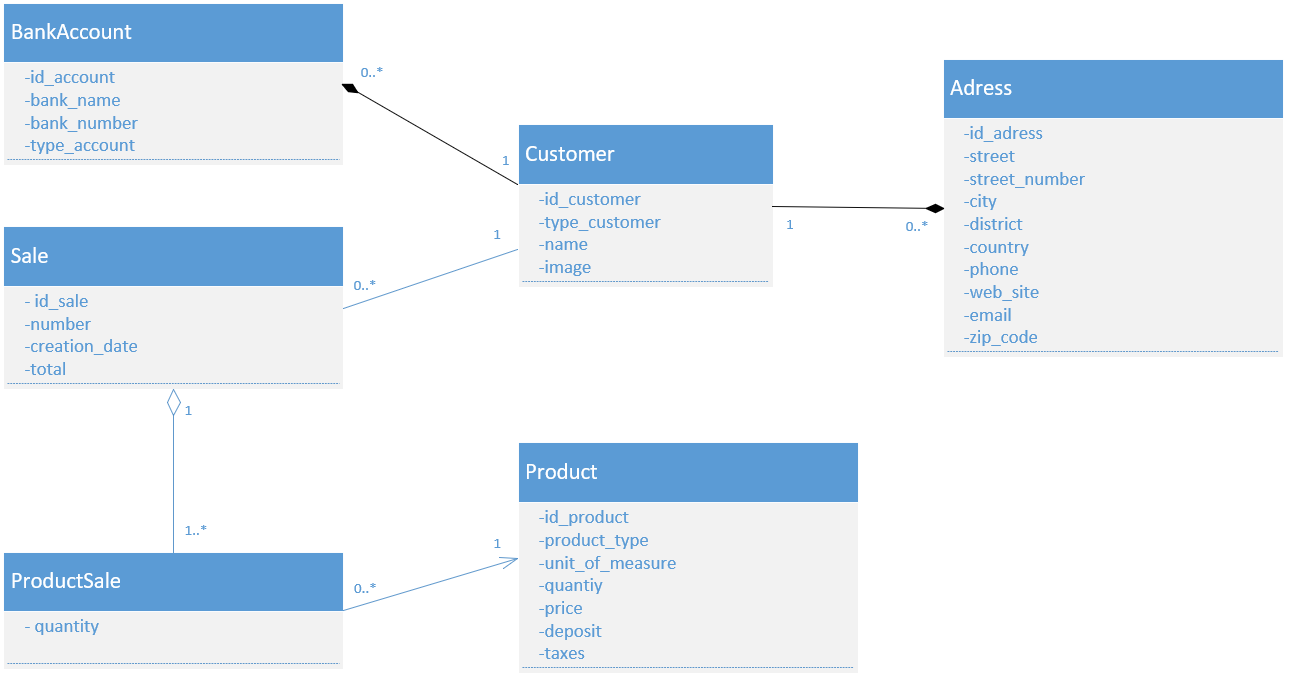


Figure 14. Diagramă de clase - SEBP

Între clasa Customer și cele două clase BankAccount și Adress există o relație de compoziție unde clasa Customer este compusă din cele două clase. Între Sale și Customer există o relație de tip asociație unde un customer poate avea mai multe vânzări sau nici una. În final Clasa ProductSale este o clasă intermediară dintre Sale și Product care conține pentru fiecare vânzare toate produsele respective și cantitatea produsului. Prin această diagramă avem o vizualizare clară a claselor și cum interacționează acestea.

## 3.3.3. Modelarea funcționalităților aplicației folosind diagrama cazurilor de utilizare

Această diagramă oferă o înțelegere clară a interacțiunii dintre obiectele aplicației și comportamentul pe durata ciclului de viață ale acestora. Se începe prin identificarea actorilor, în cazul nostru este utilizatorul care folosește aplicația, identificarea obiectelor care fac parte din acest scenariu și detalierea interacțiunilor dintre acestea. În continuare voi explica diagrama de secvență pentru interfața de vânzări și avem următoarele obiecte[[29]](#footnote-29):

1. Utilizator. Utilizatorul este o persoană fizică care interacționează cu formularul respectiv elementele grafice din acesta.
2. Formular. Acest obiect reprezintă interfața grafică în care utilizatorul vizualizează sau editează anumite date.
3. SalesForm() este o clasă în Python care conține toate funcționalitățile formularului cum ar fi elemente grafice și metode.
4. Repository este directorul care conține toate metodele care accesează baza de date.
5. Session este clasa care facilitează interacțiune cu baza de date.
6. DB sau Data Base este baza de date în sine și locul în care sunt salvate toate datele din aplicație.[[30]](#footnote-30)

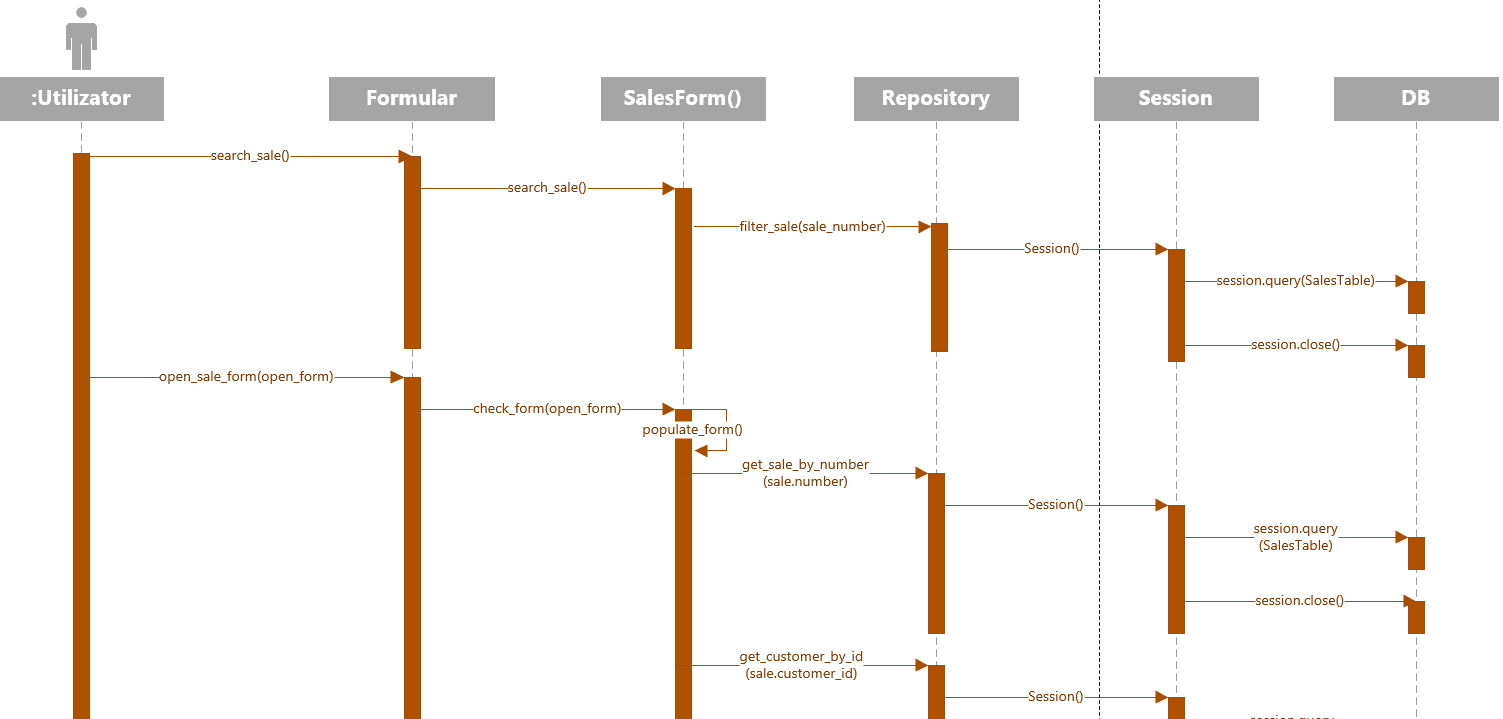


Figure 15 Diagramă de secvență – SEBP.

După cum se poate vedea în figura 13, utilizatorul interacționează cu formularul în mai multe moduri, primul mod este folosirea funcționalității **search\_sale()** din **Formular**, iar la rândul său este accesată o altă metodă rezultând o secvență. Un alt mod de a interacționa cu formularul este apăsarea butonului **create** sau selectarea unei vânzări din interfață. Indiferent de alegere se verifică cu metoda **check\_form()** daca a ales să creeze o vânzare sau selectează una deja existentă. În cazul nostru daca selectează o vânzare existentă, din clasa **SaleForm()** sunt apelate mai multe metode în secvență. Primele trei funcții au ca scop accesarea bazei de date și primirea datelor despre vânzarea selectată, produsele care se află în vânzarea respectivă și clientul care este asociat acesteia. Această situație se poate vedea în figura 15 de mai jos.

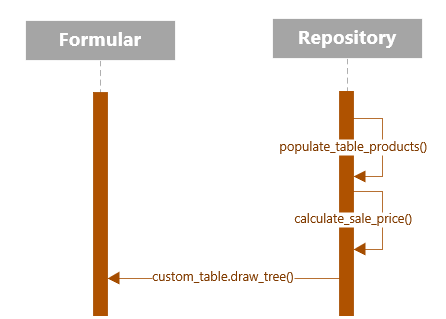


Figure 16. Diagramă de secvență - SEBP.

În final după ce toate datele au fost colectate, se populează tabelul cu toate produsele care fac parte din vânzarea selectată folosind metoda **populate\_table\_products(),** se invocă următoarea metodă **calculate\_sale\_price()** care calculează prețul total al vânzării în funcție de produsele care se află în vânzare. Este invocată o ultimă metodă **custom\_table.draw\_tree()** care va afișa tabelul cu produsele respective în interfața de vânzare.

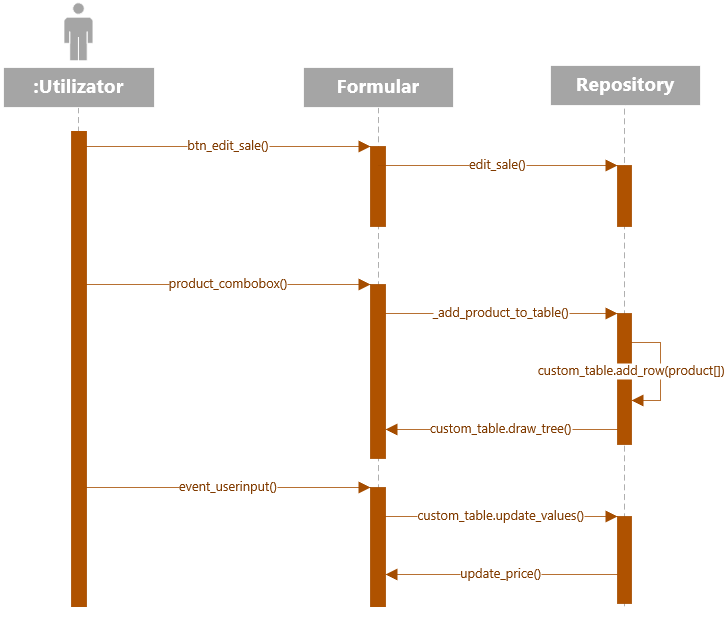


Figure 17. Diagramă de secvență – editare vânzare - SEBP

În figura 16, pentru referință prima linie verticală arată durata de viață a unui obiect, în cazul nostru fiind utilizatorul, a doua linie Formularul și a treia clasa SalesForm().

În continuare, dacă utilizatorul apasă butonul de editare a unei vânzări metoda **btn\_edit\_sale()** va fi invocată care la rândul ei în Formular, se va invoca o altă metodă **edit\_sale()** care va modifica formularul să fie editabil. Ca urmare utilizatorul poate adăuga produse în vânzare alegând un produs din **product\_combobox()** sau poate să editeze cantitatea unui produs apăsând click stânga pe cantitatea acestuia în tabel, care accesează funcția **event\_userinput()**. Se poate vedea clar din diagrama de secvențe că aceasta ajută la înțelegerea unui sistem software și la stabilirea unui plan de proiectare a unei aplicații.

## 3.4. Structura aplicației.

Sistemul SEBP are o arhitectură simplă care conține cinci dosare: classes, model, repository, images și views. Fiecare dosar la rândul său poate conține fișiere sau alte dosare de exemplu **custom\_classes** care face parte din **model.**

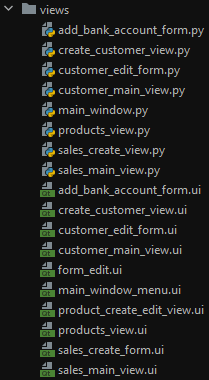
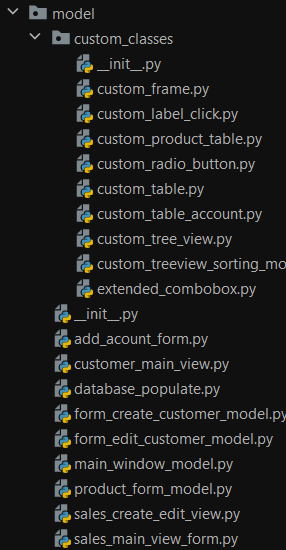
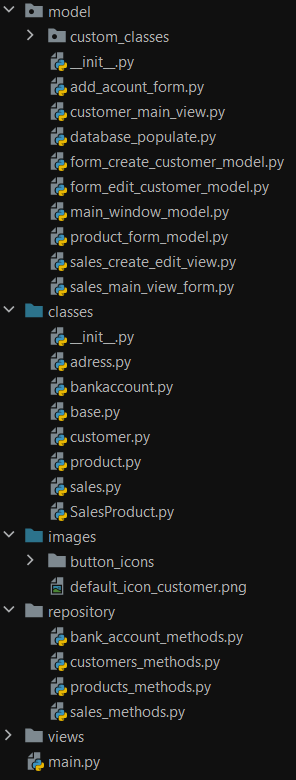


Figure 18. Exemplu de structură completă a aplicației SEBP.

## 3.5. Procesul de implementare a aplicației

În figura 16 se pot vedea toate fișierele care fac parte din aplicație, în continuare voi vorbi despre cum am implementat funcționalitățile necesare ca acest sistem software să funcționeze conform diagramelor UML modelate anterior.

### 3.5.1. Crearea interfeței utilizator folosind biblioteca PyQT5.

Primul pas este crearea unei interfețe utilizator folosind un instrument din biblioteca **PyQT5** care se numește **Qt Designer**. Acest instrument are ca rol crearea unei interfețe utilizator asemănător cu cea a unei interfețe din Windows Forms App în Microsoft Visual Studio. Pentru a crea o interfață se aleg anumite widget-uri și prin drag-and-drop se așază într-un chenar care va reprezenta interfață din aplicație. La sfârșit se salvează modificările făcute într-un fișier de tip user interface care are extensia “.ui” acest fișier este folosit ulterior pentru a genera automat cod în Python, respectiv blocuri de cod specific librăriei PyQT5.[[31]](#footnote-31)

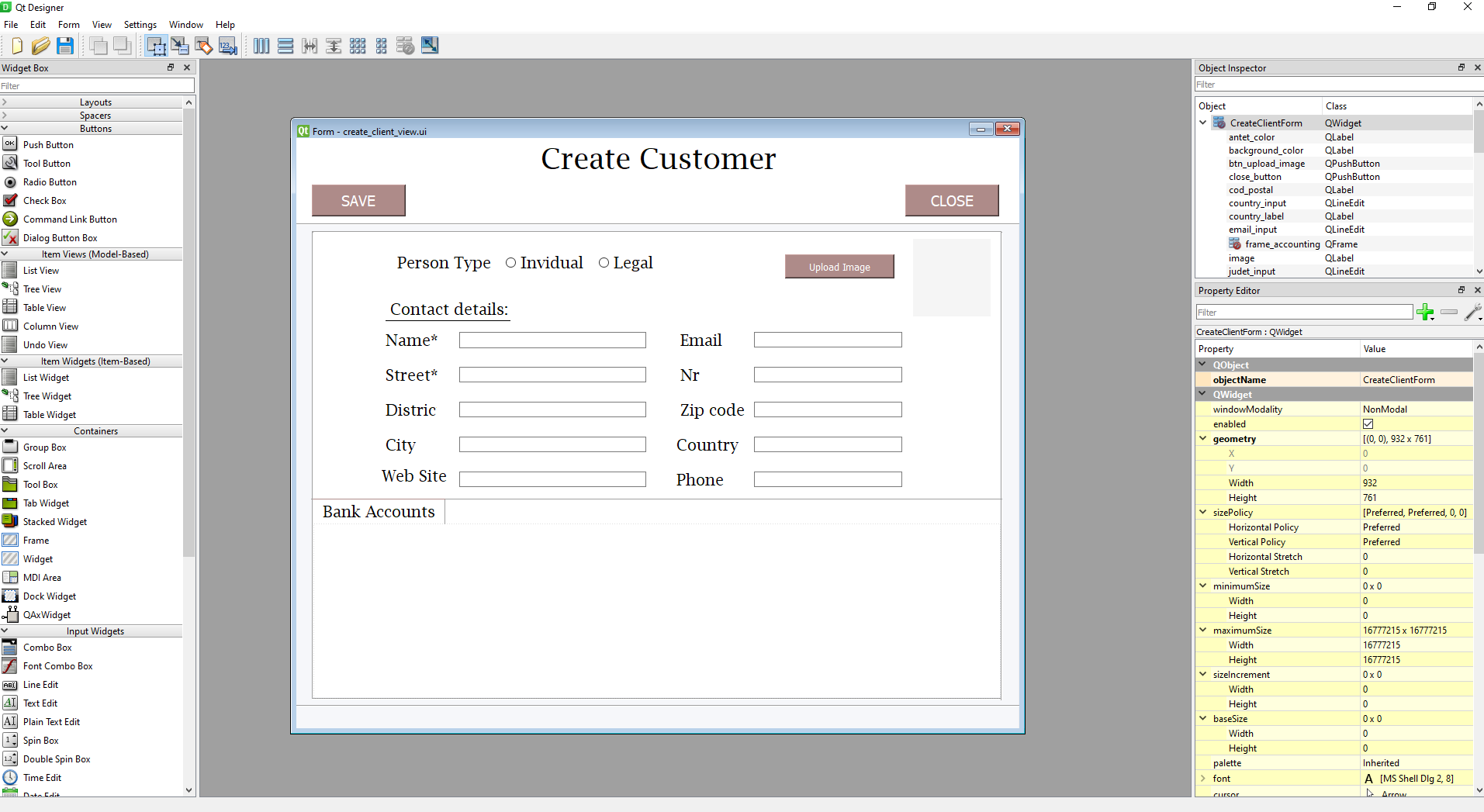


Figure 19. Exemplu de modelare a unei interfețe pentru crearea unui client - SEBP.

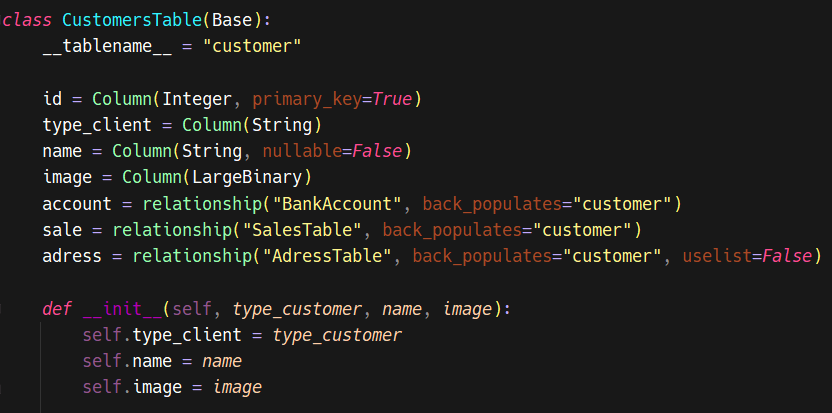
Se folosesc numeroase **widget-uri** sau elemente pentru a modela această interfață, unele dintre fiind: push button, label, frame etc. După ce interfața a fost modelată folosind acest instrument, fișierul **create\_client\_view.ui** este salvat in directorul **views** din aplicație. Ca urmare se folosește o comandă în terminal, mai exact “pyuic5 -x create\_client\_view.ui -o create\_client\_view.py” pentru a genera cod automat în limbajul de programare Python. Acest pas a fost necesar pentru că librăria PyQT5 poate transforma din fișiere de tip user interface unde acest fișier este scris în xml sau mai bine zis generat de către instrumentul Qt Designer într-un fișier Python. Pe baza modificărilor făcute în modelarea interfeței după generarea codului vom obține un fișier care conține o clasă **Ui\_Form(object)** care este utilizată pentru a configura sau inițializa interfața creată anterior. Este important de menționat că nu trebuie făcute modificări în acest fișier sau in clasa respectivă deoarece este un fișier generat automat, ca rezultat al unei noi generări de cod toate modificările făcute manual în clasa respectivă vor fi pierdute. O soluție pentru această situație este crearea unei clase copil care se folosește de clasa UI\_Form(object) pentru a moșteni toate atributele clasei respective. Ca urmare se pot face modificări ulterioare ale interfeței cum ar fi implementarea unor funcționalități deoarece clasa copil funcționează independent de clasa părinte în cazul în care aceasta este modificată. Însă trebuie menționat că dacă există modificări majore în designul aplicației de exemplu este șters un buton unde în clasa copil există o funcție care are ca referință la butonul respectiv vor apărea erori în aplicație.

### 3.5.2. Crearea bazei de date folosind biblioteca SQLAlchemy.

Deoarece limbajul de programare Python nu are instrumente legat de crearea sau accesarea unei baze de date, am folosit biblioteca SQLAlchemy pentru a crea o bază de date ținând cont de diagrama UML de clase, voi explica cum am creat clasa CustomersTable() și rolul

acesteia.

Figure 20. Exemplu crearea unei clase de tip table în Python



Pentru a crea o clasă de tip table care va fi recunoscută ca o tabelă la rularea scriptului de generare a bazei de date se creează o clasă care extinde clasa **“Base”** cu scopul de a defini o schemă a unei clase pentru o bază de date[[32]](#footnote-32). Primul atribut al clasei **CustomersTable()** este **“\_\_table\_\_”** care va reprezenta numele tabelei urmat de toate atributele din clasa respectivă cum ar fi id, type\_client, name, image etc. În final se definește metoda **“\_\_init\_\_”** care este o metodă pentru a inițializa atributele clasei atunci când aceasta este creată. Metoda aceasta mai are care rol setarea automată a funcționalității de set și get, de exemplu față de limbajul de programare Java nu este nevoie de setat manual pentru fiecare atribut o metodă de setare a atributului și una de a primi atributul respectiv. În Python prin definirea metodei **“\_\_init\_\_”** se pot accesa variabilele respective și se pot seta anumite valori pentru acestea. În final după ce toate clasele au fost create se folosește metoda **Base.metadata.create\_all(engine)**[[33]](#footnote-33) pentru a genera baza de date în funcție de clasele create anterior. Pentru a popula tabelele din baza de date se poate crea un script **populate\_database()** care conține instanțe ale claselor create anterior de exemplu CustomersTable() urmat de adăugarea parametrilor necesari pentru a inițializa clasa.

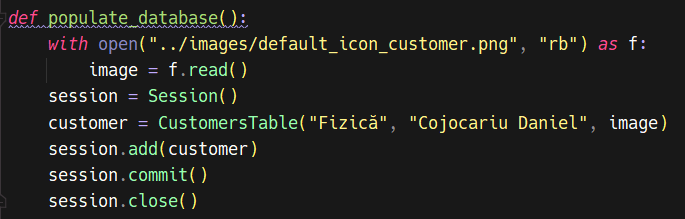


Figure 21. Exemplu de metodă pentru creare a unui client și adăugarea în baza de date

În figura 19 am definit o metodă **populate\_database()** în care se inițializează o variabilă care conține o imagine de tip default pentru iconița client-ului care va fi afișată în interfață, se deschide o nouă sesiune, se creează o instanță a clasei **CustomersTable()** care primește trei parametri, tipul clientului, numele acestuia și imaginea respectivă. Se pot inițializa mai multe instanțe ale unei clase și toate pot fi adăugate în baza de date. Acesta nu este singurul mod de populare și se poate folosi orice instrument care poate accesa o bază de date de exemplu PostgreSQL sau MySQL și folosirea unui script sql sau folosirea funcției de import a datelor utilizând un fișier excel.

### 3.5.3. Dezvoltarea aplicației SEBP și relația dintre user interface cu baza de date.

Am explicat anterior cum se creează o interfață utilizator folosind biblioteca **PyQT5** și baza de date cu biblioteca **SQLAlchemy**. În continuare voi explica procesul de dezvoltare a aplicației și relația dintre user interface cu baza de date. După ce a fost stabilită schema bazei de date conform diagramei și s-a creat interfețele necesare, în continuare se vor crea clasele copil pentru fiecare interfață în parte de exemplu pentru interfața principală Sales. Se creează clasa **SalesForm()** și va primi ca parametru **QtWidgets.Qwidget** care reprezintă un widget din biblioteca **PyQT5**, acest widget este unul general și are ca rol moștenirea tuturor elementelor din clasa părinte. Se inițializează metoda **“\_\_ini\_\_”** pentru a inițializa parametrii clasei. În final se creează o variabilă self.ui care primește ca valoare clasa **Ui\_Form()**. Pe scurt se va moșteni toate atributele clasei **Ui\_Form()** în variabila **self.ui** care din punct de vedere teoretic acum va deveni modul de accesare a oricărui atribut din clasa părinte, adică vor putea fi accesate toate elementele din interfață modelate cu ajutorul instrumentului Qt Designer.

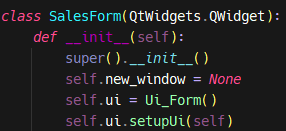


Figure 22. Exemplu creare a clasei SalesForm

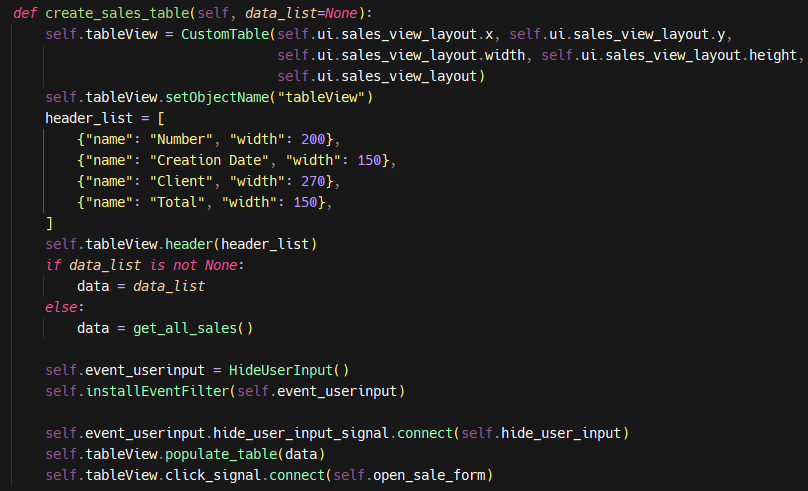


Figure 23. Definire metodă create\_sales\_table

După ce a fost inițializată clasa se vor adăuga funcționalitățile necesare unde utilizatorul poate interacționa cu aplicația. Un exemplu este inițializarea unui tabel care va fi populat cu toate vânzările din baza de date unde am prezentat un model în figura 22..

Se începe prin instanțierea clasei **CustomTable()** care primește ca parametri date despre poziție și mărimea acestuia. Se creează o listă de dicționare care reprezintă coloanele tabelului și lățimea acestora. Folosind o interogare **if else** se verifică dacă metoda actuală a primit ca parametru o listă de vânzări sau nu. În cazul în care nu a primit ca parametru o listă de vânzări adică acesta este un tabel nou se va invoca metoda **get\_all\_sales()**, metodă accesată din directorul Repository cu scopul de a primi toate vânzările din baza de date. În directorul Repository, se află toate metodele care accesează baza de date. În final se accesează metoda de populare a tabelului **populate\_table(data)** unde parametrul **data** este lista de vânzări primită anterior din baza de date.

Un alt exemplu de implementare a unei funcționalități este funcționalitatea de filtrare a vânzărilor prin campul sales\_filter.

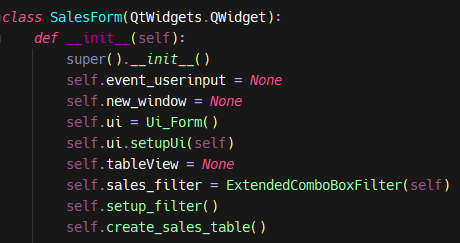


Figure 24. Exemplu adăugare funcționalitate în clasa SalesForm()

În figura 22, a fost creat câmpul **self.sales\_filter** care este de tip **ExtendedComboBoxFilter().** Aceasta este o clasă custom combobox care are funcționalitatea de a filtra datele în funcție de textul introdus de către utilizator. După ce a fost inițializată variabila **sales\_filter** se va crea o metodă care implementează funcționalitatea de search a vânzărilor din tabel și nu doar a elementelor care se află în combobox.



Figure 25. Configurarea și crearea metodei de filtrare

În figura 23, metoda **setup\_filter()** configurează elementul de filtrare în cazul nostru este un combobox care primește ca atribuite datele de poziționare a unui element existent. Folosind metoda **text\_changed** din clasa combobox se face o legătura cu funcția **search\_sale(text)** care primește ca parametru textul introdus de către utilizator. La fiecare modificare a textului este invocată funcția **search\_sale()** care la rândul ei invocă o metodă din directorul Repository care accesează baza de date și se va face un query în care vor fi filtrate vânzările în funcție de textul primit ca parametru. Ca rezultat, se face actualizează tabelul folosind metoda **update\_table(data)** având ca parametru o listă de vânzări filtrare. În figura următoare se poate vedea că rezultatul vânzărilor din combobox și tabel corespund cu textul introdus în inputul de la search sale.

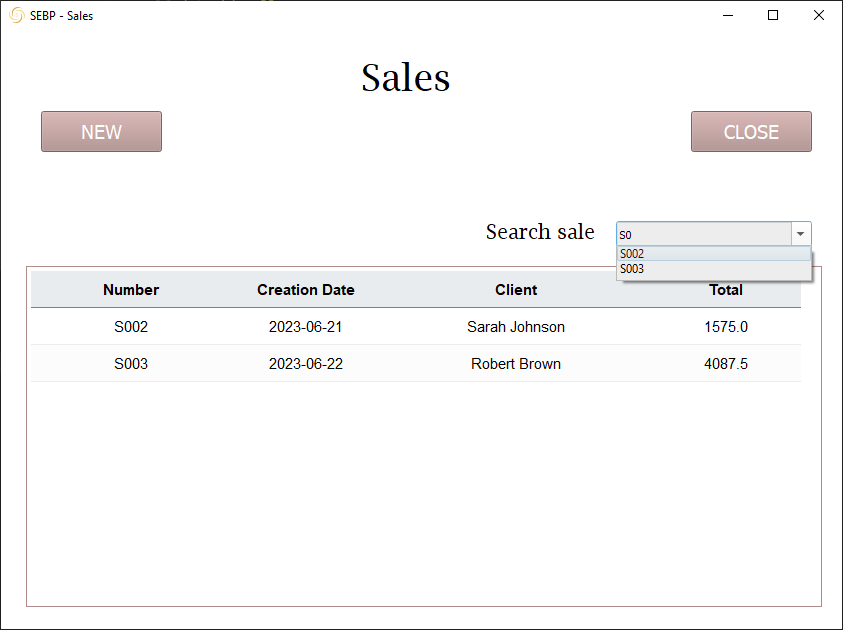


Figure 26 Exemplu de utilizare a funcționalității search - SEBP

# CONCLUZII

Scopul acestei teme a fost crearea unei aplicații în Python și folosirea diagramelor UML pentru proiectarea acesteia. Pe baza capitolului trei putem deduce că utilizarea diferitelor tipuri de diagrame pot îmbunătății timpul de dezvoltare a unui sistem. Folosind diagrama de clase am putut implementa eficient baza de date, diagrama de use case pentru a înțelege cum va interacționa utilizatorul cu user interface și în final diagrama de secvență care oferă o vizualizare abstractă a funcționalităților din aplicație și cum interacționează acestea. Obiectivul acestei lucrări a fost atins și am obținut o aplicație funcțională, un prototip ERP cu diferite funcționalități specifice.

În momentul actual, Python este unul dintre cele mai populare limbaje de programare fiind utilizat de către mari companii cum ar fi Google, Shopify, Netflix, Dropbox, Instagram și multe altele. Motivul pentru care Python este unul dintre cele mai utilizate limbaje de programare este flexibilitatea și simplicitatea acestuia. Este utilizat în diferite domenii precum algoritmi, inteligență artificială, baze de date, statistică și aplicații web. În viitor acest limbaj va fi la fel de popular ca in ziua de astăzi deoarece este intr-o continuă dezvoltare în toate domeniile sale, cel mai explorat fiind dezvoltarea unei inteligențe artificale însă conceptul de construire a unei aplicații va rămăne relevant.

Din punct de vedere al limitelor, Pyhon este limitat în primul rând din cauza că este un limbaj de tip interpretat unde există anumite restricții față de alte limbaje compilate, cum ar fi C. Există și limite din punct de vedere hardware deoarece nu comunică la un nivel avansat cu acesta.

Opinia mea personală legat de acest limbaj este una pozitivă, nu necesită cunoștințe avansate pentru a crea o aplicație, dezvoltarea acesteia se poate face într-un timp relativ scurt, iar în final este ușor de utilizat alte limbaje de programare împreună cu Python în funcție de cerințele unei companii. Este important ca pe viitor, companiile să utilizeze diagramele UML pentru a proiecta o aplicație și a înțelege din timp care sunt pași de implementeare a acesteia.

# Bibliografie

Cărți și volume:

1. Alhir, S. S. (2003). Learning UML. Taiwan: O'Reilly.
2. Arlow, J., Neustadt, I. (2005). UML 2 and the Unified Process: Practical Object-Oriented Analysis and Design. Regatul Unit: Pearson Education.
3. Beazley, D. M. (2001). Python Essential Reference. Regatul Unit: New Riders.
4. Chun, W. (2006). Core Python Programming. Statele Unite ale Americii: Pearson Education.
5. Dennis, A., Wixom, B. H., Tegarden, D. (2015). Systems Analysis and Design: An Object-Oriented Approach with UML. Regatul Unit: Wiley.
6. Fowler, M. (2018). UML Distilled: A Brief Guide to the Standard Object Modeling Language. Regatul Unit: Pearson Education.
7. Larman, C. (2004). Applying UML and Patterns. Germania: Prentice Hall PTR.
8. Miles, R., Hamilton, K. (2006). Learning UML 2.0. Elveţia: O'Reilly Media.

Site-uri Web:

*SQLAlchemy 2.0 Documentation,* <https://docs.sqlalchemy.org/en/20/core/metadata.html>

*Qt Documentation*, <https://doc.qt.io/qt-5.15>

*SQL 2.0 Documentation, Describe Database with MetaData,* [*https://docs.sqlalchemy.org/en/20/core/metadata.html#sqlalchemy.schema.MetaData.create\_all*](https://docs.sqlalchemy.org/en/20/core/metadata.html#sqlalchemy.schema.MetaData.create_all)

*SQL 2.0 Documentation, ORM Mapped Class Overview,* [*https://docs.sqlalchemy.org/en/20/orm/mapping\_styles.html#mapped-class-essential-components*](https://docs.sqlalchemy.org/en/20/orm/mapping_styles.html#mapped-class-essential-components)

Grady Booch*, Curated list of UML tools – 2022 edition*, <https://modeling-languages.com/uml-tools/>

*UML Use Case Diagram*, <https://www.lucidchart.com/pages/uml-use-case-diagram>

*UML Sequence Diagram*, <https://www.lucidchart.com/pages/uml-sequence-diagram>

*Rational Software Architect Standard Edition, UML models and diagrams,* <https://www.ibm.com/docs/en/rsas/7.5.0?topic=models-uml-diagrams>

*What is Class Diagram*, <https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-class-diagram/>

*What is Unified Modeling Language,* <https://www.lucidchart.com/pages/what-is-UML-unified-modeling-language>

*What are Important Advantages and Disadvantages Of Python*, <https://www.edureka.co/blog/advantages-and-disadvantages-of-python/#Disadvantagesofpython>

FrankHenigman, *PythonSpeed, Performance Tips,* <https://wiki.python.org/moin/PythonSpeed/PerformanceTips>

Saurabh Barot, *Python Optimization*, <https://aglowiditsolutions.com/blog/python-optimization/>

Tutort Academy*, Python: Understanding Python Libraries*, <https://medium.com/@-TutortAcademy/python-understanding-python-libraries-bb20976922c1>

*Documentation, History and License*, <https://docs.python.org/3/license.html>

*Python Features*, <https://www.geeksforgeeks.org/python-features/>

1. History of the software - <https://docs.python.org/3/license.html> [↑](#footnote-ref-1)
2. Python Features - https://www.geeksforgeeks.org/python-features/ [↑](#footnote-ref-2)
3. Hetland, M. L. (2008). Beginning Python: From Novice to Professional. Olanda: Apress. [↑](#footnote-ref-3)
4. Python: Understanding python libraries, https://medium.com/@-TutortAcademy/python-understanding-python-libraries-bb20976922c1 [↑](#footnote-ref-4)
5. Zelle, J. M. (2004). Python Programming: An Introduction to Computer Science. Statele Unite ale Americii: Franklin, Beedle. [↑](#footnote-ref-5)
6. https://wiki.python.org/moin/PythonSpeed/PerformanceTips [↑](#footnote-ref-6)
7. Python Optimization: Performance, https://aglowiditsolutions.com/blog/python-optimization/ [↑](#footnote-ref-7)
8. Hellmann, D. (2017). The Python 3 Standard Library by Example. Regatul Unit: Addison-Wesley. [↑](#footnote-ref-8)
9. The Use of Python in the field of Artifical Intelligence International Conference on Information Technology and Development of Education – ITRO 2020 October, 2020. Zrenjanin, Republic of Serbia [↑](#footnote-ref-9)
10. Artifical Intelligence with Python, Tutorials Point (I), Pvt. Ltd., 2016 [↑](#footnote-ref-10)
11. International Research Journal of Engineering and Technology (IRJET), Python – The Fastest Growing Programming Language K. R. Srinath Associate Professor, Department of Computer Science, Pragati Mahavidyalaya Degree and PG college, Hanuman Tekdi, Koti, Hyderabad, Telangana, India ---------------------------------------------------------------------\*\*\*------------------------------ [↑](#footnote-ref-11)
12. What are Important Advantages and Disadvantages Of Python?

    , https://www.edureka.co/blog/advantages-and-disadvantages-of-python/#Disadvantagesofpython [↑](#footnote-ref-12)
13. https://www.lucidchart.com/pages/what-is-UML-unified-modeling-language, ***UML and its role in object-oriented modeling and design*** [↑](#footnote-ref-13)
14. Fowler, M. (2018). UML Distilled: A Brief Guide to the Standard Object Modeling Language. Regatul Unit: Pearson Education. [↑](#footnote-ref-14)
15. **What is Unified Modeling Language**, https://www.lucidchart.com/pages/what-is-UML-unified-modeling-language [↑](#footnote-ref-15)
16. Fowler, M. (2018). UML Distilled: A Brief Guide to the Standard Object Modeling Language. Regatul Unit: Pearson Education. [↑](#footnote-ref-16)
17. UML models and diagrams, https://www.ibm.com/docs/en/rsas/7.5.0?topic=models-uml-diagrams [↑](#footnote-ref-17)
18. https://www.lucidchart.com/pages/what-is-UML-unified-modeling-language, *Object-oriented concepts in UML* [↑](#footnote-ref-18)
19. https://www.visual-paradigm.com/guide/uml-unified-modeling-language/what-is-class-diagram/ [↑](#footnote-ref-19)
20. UML models and diagrams ,https://www.ibm.com/docs/en/rsas/7.5.0?topic=models-uml-diagrams [↑](#footnote-ref-20)
21. Alhir, S. S. (2003). Learning UML. Taiwan: O'Reilly. [↑](#footnote-ref-21)
22. Alhir, S. S. (2003). Learning UML. Taiwan: O'Reilly. [↑](#footnote-ref-22)
23. Fowler, M. (2018). UML Distilled: A Brief Guide to the Standard Object Modeling [↑](#footnote-ref-23)
24. https://www.lucidchart.com/pages/uml-use-case-diagram [↑](#footnote-ref-24)
25. https://www.lucidchart.com/pages/uml-use-case-diagram [↑](#footnote-ref-25)
26. https://modeling-languages.com/uml-tools/ [↑](#footnote-ref-26)
27. \*\*\*, <https://www.romchimprotect.ro/> [↑](#footnote-ref-27)
28. https://doc.qt.io/qt-5.15/ [↑](#footnote-ref-28)
29. Alhir, S. S. (2003). Learning UML. Taiwan: O'Reilly. [↑](#footnote-ref-29)
30. https://www.lucidchart.com/pages/uml-sequence-diagram. [↑](#footnote-ref-30)
31. https://www.riverbankcomputing.com/static/Docs/PyQt5/designer.html [↑](#footnote-ref-31)
32. https://docs.sqlalchemy.org/en/20/core/metadata.html [↑](#footnote-ref-32)
33. https://docs.sqlalchemy.org/en/20/core/metadata.html#sqlalchemy.schema.MetaData.create\_all [↑](#footnote-ref-33)