Visual OETPN Simulator

PROIECT DE DIPLOMĂ

Autor: **Andreea-Ioana DRAGU**

Conducător științific: **SL. Dr. Ing. Octavian Cuibus**

|  |  |
| --- | --- |
| DECAN  **Prof.dr.ing. Liviu MICLEA** | Vizat,  DIRECTOR DEPARTAMENT AUTOMATICĂ  **Prof.dr.ing. Honoriu VĂLEAN** |

Autor: **Andreea DRAGU**

Visual OETPN Simulator

1. **Enunțul temei:**S-au proiectat două aplicații separate:

* O aplicație desktop (Visual OETPN Simulator – Desktop Application), pentru specificarea și simularea rețelelor OETPN (Object-Enhanced Timed Petri Nets);
* O aplicație web pentru descrierea aplicației desktop și pentru familiarizarea utilizatorului cu rețelele Petri și OETPN-uri.

1. **Conținutul proiectului:** *(enumerarea părților componente) Pagina de prezentare, Declarație privind autenticitatea proiectului, Sinteza proiectului, Cuprins, Titlul capitolului 1, Titlul capitolului 2,… Titlul capitolului n, Bibliografie, Anexe.*
2. **Locul documentării:** *Universitatea Tehnică din Cluj-Napoca*
3. **Consultanți:** *ing. Prenume Nume (dacă este cazul)*
4. **Data emiterii temei:** 18.10.2021
5. **Data predării:**

Semnătura autorului

Semnătura conducătorului științific

**Declaraţie pe proprie răspundere privind**

**autenticitatea proiectului de diplomă**

Subsemnatul(a)  **Andreea-Ioana DRAGU** , legitimat(ă) cu CI seria AR nr. 812184\_, CNP 2990416020073 ,

autorul lucrării:

Visual OETPN Simulator\_

elaborată în vederea susținerii examenului de finalizare a studiilor de licență la **Facultatea de Automatică și Calculatoare**, specializarea **Automatică și Informatică Aplicată,** din cadrul Universității Tehnice din Cluj-Napoca, sesiunea Iulie 2021 a anului universitar 2021-2022, declar pe proprie răspundere, că această lucrare este rezultatul propriei activități intelectuale, pe baza cercetărilor mele și pe baza informațiilor obținute din surse care au fost citate, în textul lucrării, și în bibliografie.

Declar, că această lucrare nu conține porțiuni plagiate, iar sursele bibliografice au fost folosite cu respectarea legislației române și a convențiilor internaționale privind drepturile de autor.

Declar, de asemenea, că această lucrare nu a mai fost prezentată în fața unei alte comisii de examen de licență.

În cazul constatării ulterioare a unor declarații false, voi suporta sancțiunile administrative, respectiv, *anularea examenului de licență*.

Data Andreea-Ioana DRAGU

(semnătura)

**SINTEZA**

proiectului de diplomă cu titlul:

Visual OETPN Simulator

Autor: **Andreea-Ioana DRAGU**

Conducător științific: **SL. Dr. Ing. Octavian Cuibus**

1. Cerințele temei:

2. Soluții alese:

3. Rezultate obținute:

4. Testări și verificări:

5. Contribuții personale:

6. Surse de documentare:

Semnătura autorului

Semnătura conducătorului științific

Cuprins

[1 Introducere 3](#_Toc106018361)

[1.1 Context general 3](#_Toc106018362)

[1.2 Obiective 4](#_Toc106018363)

[1.3 Specificații 4](#_Toc106018364)

[2 Studiu bibliografic 5](#_Toc106018365)

[2.1 Limbajul de programare Java 5](#_Toc106018366)

[2.1.1 Introducere în Java 5](#_Toc106018367)

[2.1.2 Noțiunea de clasă în Java 6](#_Toc106018368)

[2.1.3 Principiile programării orientate pe obiecte 8](#_Toc106018369)

[2.1.3.1 Extinderea claselor 8](#_Toc106018370)

[2.1.3.2 Abstractizarea 9](#_Toc106018371)

[2.1.3.3 Încapsularea 10](#_Toc106018372)

[2.1.3.4 Polimorfismul 11](#_Toc106018373)

[2.1.4 Excepțiile 12](#_Toc106018374)

[2.1.4.1 Prinderea și tratarea excepțiilor 12](#_Toc106018375)

[2.1.4.2 Aruncarea excepțiilor 13](#_Toc106018376)

[2.1.4.3 Avantajele tratării excepțiilor 13](#_Toc106018377)

[2.1.4.4 Ierarhia claselor ce descriu excepții 14](#_Toc106018378)

[2.1.5 Fluxuri de intrare și ieșire 14](#_Toc106018379)

[2.1.6 Bibliotecile Swing & AWT 16](#_Toc106018380)

[2.1.7 Fire de execuție 17](#_Toc106018381)

[2.1.7.1 Introducere & definiție 17](#_Toc106018382)

[2.1.7.2 Stările unui fir de execuție 18](#_Toc106018383)

[2.1.7.3 Crearea firelor de execuție 18](#_Toc106018384)

[2.1.7.4 Ciclul de viață al unui fir de execuție 19](#_Toc106018385)

[2.1.7.5 Setarea priorităților firelor de execuție 20](#_Toc106018386)

[2.1.7.6 Sincronizarea firelor de execuție 20](#_Toc106018387)

[2.2 Baze de date 21](#_Toc106018388)

[2.2.1 Introducere în conceptul de bază de date 22](#_Toc106018389)

[2.2.2 Baze de date relaționale 22](#_Toc106018390)

[2.2.3 Limbajul de interogare SQL 24](#_Toc106018391)

[2.2.3.1 Introducere în SQL 24](#_Toc106018392)

[2.2.3.2 Comenzile și sintaxa limbajului SQL 25](#_Toc106018393)

[2.2.3.3 Comenzi uzuale în limbajul SQL 26](#_Toc106018394)

[2.3 Crearea paginilor web 27](#_Toc106018395)

[2.3.1 HTML 28](#_Toc106018396)

[2.3.1.1 Noțiuni introductive 28](#_Toc106018397)

[2.3.1.2 Structura unei pagini Web 29](#_Toc106018398)

[2.3.1.3 Marcaje utilizate 30](#_Toc106018399)

[2.3.2 CSS 31](#_Toc106018400)

[2.3.2.1 Noțiuni introductive 31](#_Toc106018401)

[2.3.2.2 Sintaxa CSS 31](#_Toc106018402)

[2.3.2.3 Integrarea stilurilor CSS în documentul HTML 33](#_Toc106018403)

[2.3.2.4 Clasificarea stilurilor CSS 34](#_Toc106018404)

[2.3.3 JavaScript 35](#_Toc106018405)

[2.3.3.1 Noțiuni generale 35](#_Toc106018406)

[2.3.3.2 Integrarea script-urilor JavaScript în documentul HTML 35](#_Toc106018407)

[2.4 36](#_Toc106018408)

[3 Analiză, proiectare, implementare 37](#_Toc106018409)

[4 Concluzii 39](#_Toc106018410)

[4.1 Rezultate obținute 39](#_Toc106018411)

[4.2 Direcții de dezvoltare 39](#_Toc106018412)

[5 Bibliografie 41](#_Toc106018413)

# Introducere

## Context general

Sistemele reprezintă un ansamblu de elemente interdependente, între care se realizează acțiuni cu scopul îndeplinirii unui obiectiv sau a unei funcționalități comune. Pentru a determina comportarea sistemului, este necesar un studiu al proprietăților acestor elemente, dar și a acțiunilor pe care acestea le realizează. Prin realizarea acestui studiu al proprietăților componentelor, se poate face o analiză a modificării caracteristicilor sistemului în raport cu timpul: dacă una sau mai multe dintre respectivele caracteristici se modifică în timp, sistemul este dinamic.

Sistemele dinamice pot fi clasificate, după cum urmează:

* Sisteme cu parametri variabili în timp
* Sisteme invariante în timp
* Sisteme neliniare
* Sisteme liniare
* Sisteme cu stări continue
* Sisteme cu stări discrete
* Sisteme cu evoluție determinată de timp
* Sisteme cu evoluție determinată de evenimente
* Sisteme deterministe
* Sisteme stochastice

Categoria de sisteme care face obiectul acestei lucrări și pe care se va pune accentul în continuare, este categoria sistemelor cu evenimente discrete (SED), care diferă de celelalte categorii de sisteme prin următoarele caracteristici:

* Comportarea dinamică a sistemelor cu evenimente discrete este determinată de realizarea unor evenimente (event-driven) - evenimentele nu au durată;
* Acțiunile de control au loc la momente de timp bine precizate;
* Unele variabile ale sistemelor cu evenimente discrete sunt cuantificate;
* Modelarea și analiza sistemelor cu evenimente discrete se bazează pe ecuații diferențiale sau ecuații cu diferențe, deoarece evoluția acestor sisteme este descrisă de evenimente.

Pentru descrierea, analiza și modelarea sistemelor cu evenimente discrete, se utilizează diverse abordări, însă abordarea care realizează obiectul acestei lucrări este abordarea SED cu ajutorul rețelelor Petri de diferite tipuri.

Rețelele Petri modelează sistemele dinamice cu evenimente discrete (SDED), modelul descriind stările și evenimentele care duc la evoluția acestor sisteme. Starea unui sistem cu evenimente discrete este descrisă cu ajutorul unui set de variabile de stare, care reprezintă condițiile. Evenimentele care descriu evoluția sistemelor dinamice cu evenimente discrete nu au durată.

Pentru lucrarea de față, s-au folosit rețelele Petri temporizate și înzestrate de obiecte, mai exact modele OETPN (Object-Enhanced Timed Petri Nets). Modelele OETPN au câteva trăsături noi, dar conservă posibilitățile de verificare și descriere ale rețelelor Petri clasice, și pot descrie următoarele:

* Operații logice (AND, OR, NAND, NOR, etc);
* Excludere reciprocă / mutuală;
* Sincronizări;
* Utilizarea în comun a resurselor limitate;
* Blocaje (Deadlocks);
* Execuții fără sfârșit;
* Execuții condiționate;
* Concurență, etc.

Pentru a specifica, observa și simula evoluția și comportamentul sistemelor dinamice cu evenimente discrete utilizând rețelele Petri temporizate și înzestrate de obiecte (modele OETPN), s-a dezvoltat aplicația desktop Visual OETPN Simulator. Această aplicație are rolul de a proiecta, implementa și simula modele OETPN precum:

* Modelarea arcelor inhibitoare și a arcelor de resetare
* Modelarea operațiilor fără blocare
* Task-uri cu sau fără sincronizare
* Citirea fără blocaj, cu așteptare limitată
* Bucle / execuții fără sfârșit
* Deadlock (blocaj), evitarea blocării etc.

De asemenea, s-a dezvoltat o aplicație web, Visual OETPN Simulator – Web Application, care dispune de informații legate atât despre aplicația desktop, cât și despre rețele Petri și modele OETPN. Mai mult, Visual OETPN Simulator – Web Application are opțiunea de a descărca varianta desktop în pagina de Download.

## Obiective

Obiectivul acestei lucrări urmărește dezvoltarea unei aplicații desktop, numită Visual OETPN Simulator, care este capabilă să specifice, implementeze și să simuleze atât rețele Petri clasice, cât și rețele Petri temporizate, înzestrate de obiecte (Object-Enhanced Timed Petri Nets).

## Specificații

Pentru îndeplinirea obiectivelor enunțate la punctul anterior, dar și al realizării cu succes a întregii aplicații, s-au realizat următorii pași:

În specificațiile lucrării detaliați cerințele. Descrieți ce intenționați să obțineți. Vă puteți referi la funcțiile aplicației, interfață, nivele de performanță, structuri de date, elemente, securitate, fiabilitate, calitate, limitări, etc.

# Studiu bibliografic

Acest capitol are rolul de a prezenta cât mai clar informații despre limbajele de programare, tehnologiile și noțiunile teoretice utilizate pentru realizarea cu succes a aplicației de specificare și simulare a rețelelor Petri temporizate înzestrate de obiecte (Object-Enhanced Timed Petri Nets – OETPN), dar și a aplicației web care servește la prezentarea acestei aplicații. Respectivele informații au fost colectate cu scopul de a stabili caracteristicile pe care această aplicație le conține.

## Limbajul de programare Java

Pentru funcționalitatea interfeței grafice cu utilizatorul a aplicației desktop de specificare și simulare a rețelelor Petri temporizate înzestrate de obiecte, s-a utilizat limbajul de programare Java. În continuare, se vor prezenta toate informațiile și noțiunile din acest limbaj de programare, care au fost aplicate pentru realizarea scopului anterior.

În acest sens, se vor descrie următoarele:

* **Noțiunea de clasă:** fiecare componentă din interfața grafică cu utilizatorul (buton, fereastră, casetă de dialog etc.) reprezintă o clasă în Java;
* **Principiile programării orientate pe obiecte:** prin punerea lor în practică, codul devine mai facil, mai lizibil și mai corect din punct de vedere structural;
* **Excepțiile:** deoarece, în diferite puncte ale execuției programului este posibil ca acesta să se oprească în mod neașteptat, se va utiliza tratarea excepțiilor pentru a asigura o continuitate a rulării programului;
* **Fluxurile de intrare/ieșire:** deoarece avem de-a face cu scrierea și citirea în/din fișiere cu extensia .xml, este necesară utilizarea acestor fluxuri;
* **Bibliotecile Swing și AWT:** pentru realizarea design-ului interfeței grafice cu utilizatorul, s-au utilizat componente atât din biblioteca Swing, cât și din biblioteca AWT;
* **Firele de execuție:** deoarece avem nevoie de realizarea a mai multor activități în același timp, se vor folosi firele de execuție.

### Introducere în Java

Java reprezintă o tehnologie dezvoltată și lansată de compania Sun Microsystems în anul 1995. Java, ca tehnologie și limbaj de programare, a avut un impact remarcabil asupra comunității dezvoltatorilor software, remarcându-se și diferențiindu-se de alte limbaje de programare prin anumite calități:

* **Simplitate:** se renunță la utilizarea a mai multor caracteristici care influențează scrierea unui cod confuz, precum pointerii, supraîncărcarea operatorilor, moștenirea multiplă etc.;
* **Ușurință:** realizarea aplicațiilor de diferite complexități care utilizează interfețe grafice, fire de execuție, baze de date, programarea în rețea etc.;
* **Robustețe:** eliminarea surselor de erori ce apar frecvent în aplicații și, de asemenea, folosirea administrării automate a memoriei și colectarea obiectelor care nu mai sunt referite, operațiune ce elimină pierderile de memorie; acest administrator automat de memorie se mai numește și Garbage Collector (GC);
* **Orientare pe obiecte:** emiminarea completă a stilului de programare procedural (structural);
* **Securitate:** limbajul Java este un limbaj de programare care se remarcă prin securitatea de care dispune, furnizând anumite mecanisme de securitate a programelor, precum verificarea în mod dinamic a codului pentru detectarea posibilelor secvențe periculoase etc.;
* **Neutralitate arhitecturală:** comportamentul unei aplicații dezvoltate cu ajutorul limbajului de programare Java nu depinde de arhitectura fizică a mașinii pe care aceasta rulează;
* **Portabilitate:** limbajul de programare Java este un limbaj independent de platforma de lucru, ceea ce înseamnă că aceeași aplicație Java poate rula, fără modificări ulterioare, pe sisteme de operare diferite, precum MacOS, Windows și Linux;
* **Compilat și interpretat:** limbajul de programare Java este atât compilat (transformat de compilator într-un cod mașină, ce poate fi executat direct de către procesor), cât și interpretat (instrucțiunile sunt citite linie cu linie de un program numit interpretor și traduse în instrucțiuni mașină); faptul că limbajul de programare Java este, în același timp, compilat și interpretat, reprezintă o soluție eficientă pentru obținerea portabilității;
* **Performanță:** asigurarea performanțelor ridicate ale codului de octeți, chiar dacă este mai lent decât alte limbaje de programare;
* **Multithreading:** permiterea programării cu fire de execuție.

### Noțiunea de clasă în Java

În cazul limbajelor de programare orientate pe obiecte, clasa reprezintă entitatea de bază pentru acestea și, totodată, o modalitate de introducere a noi tipuri de date într-o aplicație Java. În limbajul de programare Java, este obligatorie definirea a cel puțin unei clase, care se va realiza după următoarea sintaxă generală:

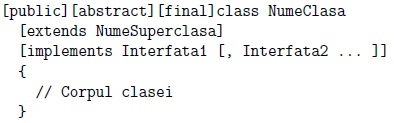


Figura 2.1.1. Sintaxa generală a unei clase în Java.

Observând figura de mai sus, putem defini următorii modificatori de acces ai unei clase, după cum urmează:

* **public:** o clasă declarată cu modificatorul de acces public poate fi accesată și folosită de orice altă clasă, indiferent de pachetul în care se regăsește;
* **abstract:** o clasă declarată cu modificatorul de acces abstract nu poate fi instanțiată, deoarece aceasta este utilizată doar în scopul de a crea un șablon pentru o serie de subclase ce o extind;
* **final:** o clasă declarată cu modificatorul de acces final marchează faptul că respectiva clasă nu poate fi extinsă (nu poate avea subclase).

În limbajul de programare Java, orice este asociat cu clase și obiecte, pe lângă atributele și metodele corespunzătoare. Spre exemplu, în lumea reală, un telefon este un obiect, care are atribute (culoare, model, marcă, anul lansării etc.), și metode (apelare, trimitere mesaj, încărcare etc.). Diagrama de clasă a clasei Telefon se regăsește în figura de mai jos.

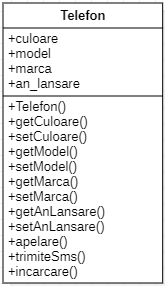


Figura 2.1.2. Diagrama de clasă a clasei Telefon, cu atributele, constructorul și metodele (funcțiile) aferente.

### Principiile programării orientate pe obiecte

În limbajul de programare Java, cele patru principii ale Programării Orientate pe Obiecte (Object-Oriented Programming – OOP) sunt:

* Extinderea
* Abstractizarea
* Încapsularea
* Polimorfismul

#### Extinderea claselor

Diferența pe care o posedă limbajul de programare Java, față de celelalte limbaje de programare, constă în faptul că Java permite doar moștenirea simplă. Acest lucru înseamnă că o subclasă poate avea doar o singură superclasă (poate moșteni / extinde doar o superclasă), însă este de reținut că o superclasă poate avea oricâte subclase. În figura de mai jos, se pot vedea toate tipurile posibile de extindere / moștenire a claselor în Java.

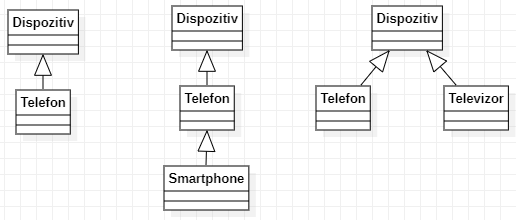


Figura 2.1.3.1.1. Tipuri de moștenire (simplă, multinivel, ierarhică).

Pentru ca o subclasă să extindă / moștenească o superclasă, se va folosi cuvântul cheie extends.

Toate clasele definite în Java pot fi văzute ca un arbore, rădăcina acestuia fiind clasa Object, reprezentând singura clasă care nu are părinte. De asemenea, clasa Object este extrem de importantă în modul de lucru cu obiecte și structuri de date în Java, dispunând de diferite metode, cele mai cunoscute fiind: toString(), hashCode(), equals(Object o), getClass() etc.

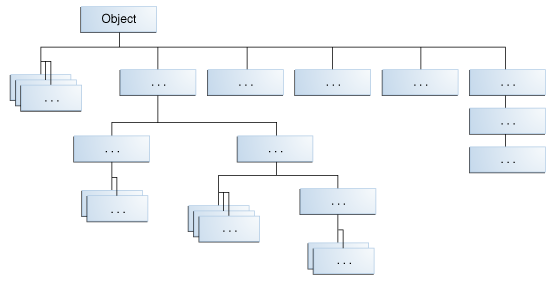


Figura 2.1.3.1.2. Clasa Object, împreună cu subclasele care o extind.

#### Abstractizarea

În Java, conceptul de abstractizare se referă la procesul de a lucra mai degrabă cu o idee, decât cu o implementare concretă. Prin abstractizare, se “ascund” detaliile complexe față de utilizator; cu alte cuvinte, utilizatorul va cunoaște doar funcționalitatea generală a entității în locul funcționalității interne a acesteia.

Avantajele abstractizării sunt descrise, după cum urmează:

* Reducerea complexității;
* Evitarea duplicării codului;
* Ușurință în mentenanță;
* Creșterea securității și a confidențialității.

Clasele abstracte reprezintă clasele ce nu pot fi instanțiate, adică obiectele de respectiva clasă nu pot fi create. O clasă abstractă (declarată cu modificatorul abstract) este o clasă care conține variabile de clasă, metode care nu sunt abstracte și metode abstracte.

Metodele abstracte dintr-o clasă abstractă reprezintă metode care nu au implementare, ci doar semnătură; implementarea metodei abstracte se va realiza de subclasele care extind / moștenesc superclasa abstractă.

Un exemplu concret de abstractizare se poate observa în figura de mai jos. Clasa Sensor este o superclasă abstractă, cu variabila de clasă location, cu modificatorul de acces private, metoda abstractă readValue() și metoda getLocation(). Subclasele care extind / moștenesc clasa abstractă Sensor sunt TemperatureSensor și LightSensor, fiecare dintre aceste clase având un obiect de clasă Controller (relație “has-a”). Mai mult, atât clasa TemperatureSensor, cât și clasa LightSensor trebuie să furnizeze o implementare pentru metoda abstractă readValue(), deoarece ambele clase extind clasa abstractă Sensor.

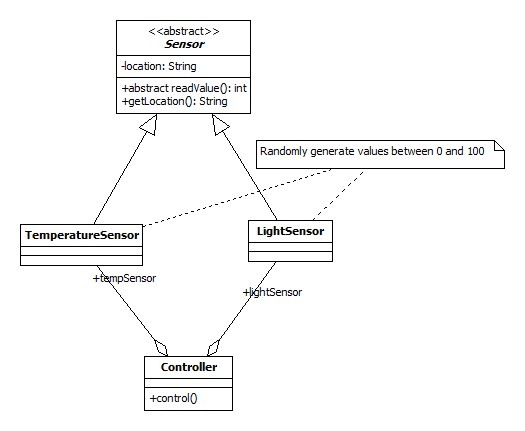


Figura 2.1.3.2.2. Superclasa abstractă Sensor, extinsă / moștenită de subclasele TemperatureSensor și LightSensor.

Pe lângă clasele abstracte, limbajul de programare Java oferă un alt tip de implementare a aplicațiilor prin utilizarea interfețelor, care duce conceptul de clasă abstractă la un alt nivel. Interfețele elimină orice implementare a metodelor, punându-se astfel în practică separarea modelului unui obiect de implementarea sa.

În Java, o interfață definește una sau mai multe metode, însă nu se specifică nicio implementare pentru aceasta; dacă o clasă implementează o interfață (cuvântul cheie interface și modificatorul implements), acea clasă trebuie, în mod obligatoriu, să ofere implementări pentru respectivele metode ale interfeței. Astfel, orice clasă care implementează o interfață se supune unui anumit comportament furnizat de interfața respectivă.

Un exemplu de interfață este prezentat în figura de mai jos.

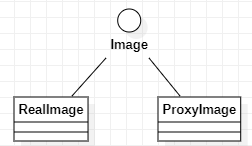


Figura 2.1.3.2.3. Interfața Image, implementată de clasele RealImage și ProxyImage.

#### Încapsularea

Încapsularea reprezintă unul din cele patru concepte fundamentale ale programării orientate pe obiecte, și reprezintă un mecanism de “împachetare” a variabilelor și a metodelor, făcând astfel aplicația să acționeze ca o singură unitate.

Procedeul de încapsulare mai poate fi văzut și ca ascunderea datelor, deoarece variabilele unei clase vor fi ascunse de alte clase (vor fi declarate cu modificatorul de acces private) și pot fi accesate doar prin metodele clasei lor curente (metode de tip getter și setter, declarate cu modificatorul de acces public, care returnează, respectiv setează variabilele clasei).

Un exemplu de încapsulare este prezentat în figura de mai jos. Se observă că variabilele clasei Encapsule sunt declarate cu modificatorul static private, pentru a nu putea fi accesate de alte clase. Pe lângă aceasta, metodele de tip getter și setter sunt declarate cu modificatorul de acces public, pentru a putea fi accesate de alte clase.

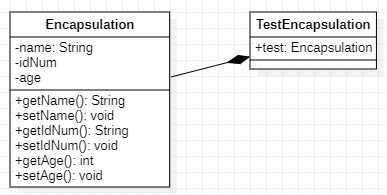


Figura 2.1.3.3. Exemplu de încapsulare

#### Polimorfismul

Polimorfismul, în Java, reprezintă abilitatea unui obiect de a lua mai multe forme, cu alte cuvinte, este abilitatea de a descoperi, în mod dinamic, tipul concret al unui obiect dintr-o ierarhie de tipuri. Acest principiu fundamental al programării orientate pe obiecte permite manipularea, într-un mod unitar, a colecțiilor de obiecte ce împart caracteristici comune, definite prin intermediul unei clase de bază.

Pentru o mai bună înțelegere a acestui concept, se va urmări exemplul din figura de mai jos. Se va construi o ierarhie de clase, având clasa de bază Bird și două clase derivate, Goose și Penguin. Clasa Bird conține metoda move(), care va fi suprascrisă de cele două clase derivate, în funcție de comportamentul dorit: pentru clasa Goose, se va afișa în consolă mesajul “The GOOSE is flying”, iar pentru clasa PENGUIN, se va afișa în consolă mesajul “The PENGUIN is swimming”.

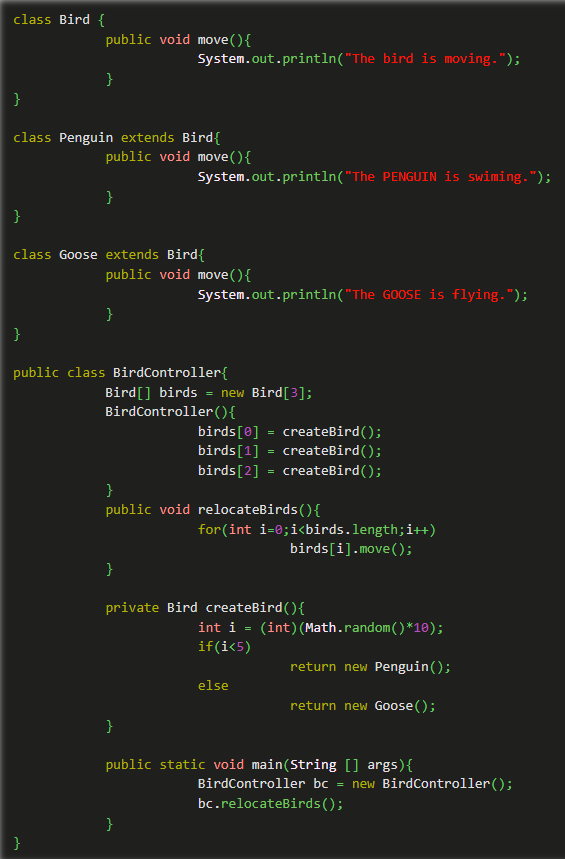


Figura 2.1.3.4. Exemplificarea polimorfismului cu ajutorul clasei Bird și a claselor derivate, Goose și Penguin.

### Excepțiile

Excepția reprezintă un eveniment care provoacă întreruperea unui program în timpul cursului normal al execuției acestuia. Acest tip de eveniment apare din diverse cauze și pot fi declanșate atât de erori ale echipamente hardware, cât și de erori ale codului programului.

În momentul în care se declanșează o excepție, va fi generat un obiect de tip excepție ce conține următoarele:

* Informații legate de excepția tocmai declanșată;
* Starea programului în momentul producerii excepției respective.

Crearea unui obiect de tip excepție se mai numește și aruncarea unei excepții, secvența de cod care tratează o anumită excepție se numește analizor de excepție, iar interceptarea și tratarea ei se numește prinderea excepției. Cu alte cuvinte, la apariția unei erori, se “aruncă” o excepție și, pentru a fi tratată, aceasta trebuie să fie “prinsă”.

O excepție poate fi prinsă și tratată, aruncată sau creată de utilizator.

#### Prinderea și tratarea excepțiilor

Pentru prinderea și tratarea excepțiilor, se va utiliza blocul de instrucțiuni try-catch-finally. Structura unui astfel de bloc se poate vedea în figura de mai jos. În blocul try, se va insera codul care poate produce excepții, în blocul catch se prind și se tratează excepțiile, iar în blocul finally se inserează codul redundant. De asemenea, se observă că se pot prinde mai multe excepții deodată, cu ajutorul blocurilor multiple catch.

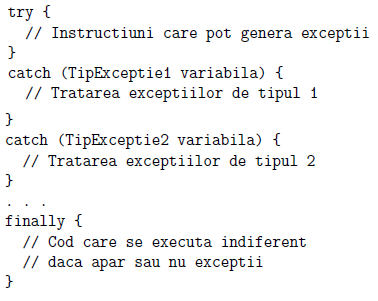


Figura 2.1.4.1. Blocul de instrucțiuni try-catch-finally.

#### Aruncarea excepțiilor

Dacă o metodă este predispusă să genereze excepții, însă nu își asumă responsabilitatea de a le trata, aceasta poate “arunca” aceste excepții către metodele care o apelează. Dacă nici aceste metode nu vor fi responsabile pentru tratarea acestor excepții, ele vor arunca mai departe excepțiile.

Aruncarea excepțiilor se va realiza prin specificarea cuvântului cheie throws, în declarația metodei, astfel:

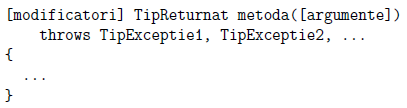


Figura 2.1.4.2. Aruncarea unei excepții

#### Avantajele tratării excepțiilor

În limbajul de programare Java, tratarea excepțiilor vine cu câteva avantaje:

* Posibilitatea de separare a codului pentru tratarea unei erori in codul în care ea poate să apară, face codul mai lizibil și mai ușor de înțeles;
* Posibilitatea propagării unei erori până la un analizor de excepții corespunzător, deoarece unele metode nu își asumă responsabilitatea de a trata excepția ce poate să apară;
* Posibilitatea grupării erorilor după tipul lor, în funcție de similaritățile pe care le împărtășesc.

#### Ierarhia claselor ce descriu excepții

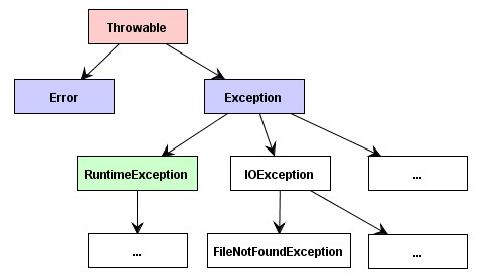


Figura 2.1.4.4. Ierarhia claselor care descriu diferite tipuri de excepții.

### Fluxuri de intrare și ieșire

Un flux de intrare/ieșire se referă la mecanismele cu ajutorul cărora programele Java pot realiza următoarele operațiuni:

* Cu ajutorul fluxurilor de intrare, datele sunt citite de la o destinație:
* Cu ajutorul fluxurilor de ieșire, datele sunt scrise/transmise către o destinație.

Cu alte cuvinte, un flux de intrare/ieșire reprezintă un canal de comunicație într-o singură direcție (unidirecțional), prin intermediul căruia datele pot fi citite sau scrise. Schematic, se poate observa funcționalitatea acestor fluxuri.

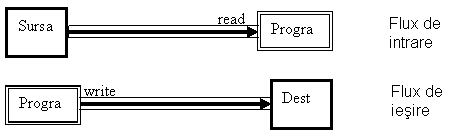


Figura 2.1.5.1. Funcționalitatea fluxurilor de intrare/ieșire.

În limbajul de programare Java, există două tipuri de fluxuri: orientate pe byte (folosite pentru scrierea și citirea de date binare) și orientate pe caracter (folosite pentru manipularea șirurilor de caractere).

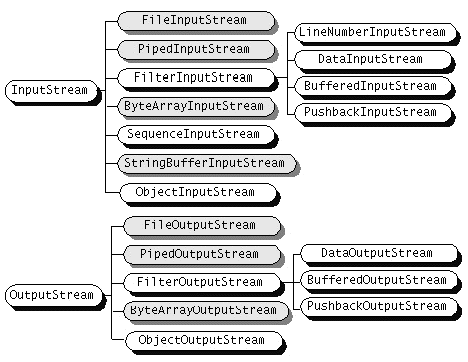


Figura 2.1.5.2. Fluxurile de intrare/ieșire orientate pe byte.

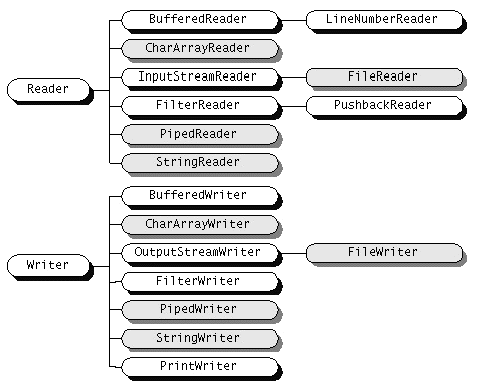


Figura 2.1.5.3. Fluxurile de intrare/ieșire orientate pe caracter.

### Bibliotecile Swing & AWT

Interfețele grafice cu utilizatorul (Graphic User Interfaces – GUI) reprezintă toate tipurile de comunicare vizuală dintre un program Java și utilizator. Mai exact, o interfață grafică reprezintă o fereastră ce conține elemente grafice (butoane, căsuțe de text, slidere, etichete etc.), care permit interacțiunea dintre aplicația Java și utilizator.

În momentul de față, pentru a realiza o interfață grafică, limbajul de programare Java pune la dispoziție două biblioteci:

* AWT (Abstract Window Toolkit): reprezintă API-ul pus la dispoziție începând cu primele versiuni de Java;
* Swing: bibliotecă ce face parte din Java Foundation Classes și care se bazează pe modelul AWT, însă se diferențiază de acesta prin adăugirea/înlocuirea unor componente.

Deoarece tehnologia Swing dispune de o paletă mai largă de componente și facilități în crearea interfețelor grafice cu utilizatorul, este recomandat să se utilizeze preponderent, însă este de reținut că AWT conține clase esențiale, reutilizate.

Pentru a realiza o interfață grafică cu utilizatorul, se țin cont de funcționalitate (definirea unor acțiuni care s vor executa în momentul în care utilizatorul interacționează cu componentele ferestrei) și de design (fereastră de afișare care conține diverse componente, menite să interacționeze cu utilizatorul).

Figura de mai jos prezintă câteva dintre cele mai importante clase din cadrul bibliotecilor AWT și Swing.

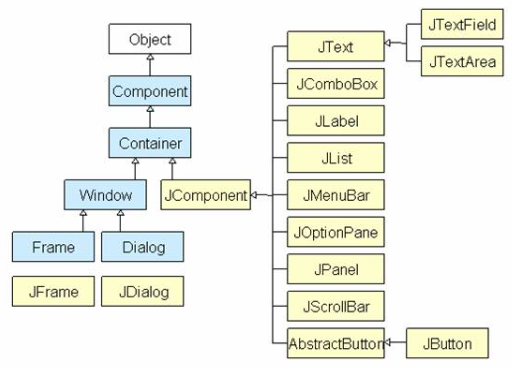


Figura 2.1.6. Clase din cadrul bibliotecilor AWT și Swing.

Majoritatea componentelor dintr-o interfață grafică cu utilizatorul sunt menite să îndeplinească diverse funcționalități și comportamente. Pentru a realiza acest lucru, se va stabili o comunicare între utilizator și aplicație prin intermediul unui mecanism bazat pe evenimente. Orice acțiune a utilizatorului exercitată asupra unei componente (apăsarea unui buton, mișcarea mouse-ului, apăsarea butonului stâng al mouse-ului) determină generarea unor eveniment de respectiva componentă.

Ca urmare a acțiunii utilizatorului asupra unei componente, evenimentul generat de către aceasta poate fi captat și gestionat în felul următor:

* Se va implementa o clasă de tip Listener, care va capta evenimentul generat de o componentă și, ulterior, tratat corespunzător prin intermediul unor metode;
* Se va înregistra respectiva clasă drept consumator al evenimentelor produse de componenta care le generează.

Câteva exemple de clase, capabile să capteze și să gestioneze evenimente generate de componente, sunt: ActionListener, AdjustmentListener, ComponentListener, ContainerListener, FocusListener, ItemListener, KeyListener, MouseListener, MouseMotionListener, TextListener, WindowListener etc.

Așadar, pentru crearea interfețelor grafice cu utilizatorul, este nevoie de utilizarea bibliotecilor Swing și AWT pentru plasarea componentelor în fereastră (design), iar pentru a produce un eveniment în momentul în care o componentă este acționată, se vor folosi clasele de tip Listener, împreună cu metodele aferente (funcționalitate).

### Fire de execuție

#### Introducere & definiție

În unele situații de dezvoltare a unei aplicații Java, este nevoie ca o aplicație să poată realiza, în același timp, mai multe activități; în acest caz, se vor folosi firele de execuție, care fac diferența dintre programarea secvențială (modelul clasic de program) și programarea concurentă.

Ca și definiție, un fir de execuție reprezintă o succesiune de instrucțiuni (secvențială) care se execută în cadrul unui proces. În figurile de mai jos, se pot observa două programe:

* Primul program (proces) cu un singur fir de execuție;
* Al doilea program (proces) cu două fire de execuție care se execută în paralel.

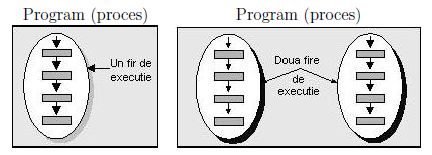


Figura 2.1.7.1. Programe (procese): a) Monofir b) Multifir

Orice program dezvoltat în Java poate avea oricâte fire de execuție, ceea ce înseamnă că, în cadrul unui proces, pot fi executate, în același timp, mai multe fire de execuție. Astfel, se pot îndeplini diferite sarcini independente ale programului, cum ar fi:

* Calcule matematice;
* Desenarea unor componente pentru o interfață grafică utilizator;
* Așteptarea eliberării unei resurse etc.

Uzual, firele de execuție își desfășoară activitatea în fundal, însă acest lucru nu este strict obligatoriu.

#### Stările unui fir de execuție

Un fir de execuție se poate afla în 4 stări, după cum urmează:

* **New:** firul de execuție a fost creat, însă nu a fost startat;
* **Runnable:** firul de execuție se află în starea în care poate fi rulat, în momentul în care procesorul este disponibil;
* **Dead:** firul de execuție și-a finalizat execuția (ieșire din metoda run(); de reținut că metoda stop() a fost depreciată și nu se recomandă folosirea sa);
* **Blocked:** firul de execuție se află în starea blocat și nu poate fi rulat, cu toate că procesorul este disponibil.

În schema de mai jos se pot observa toate stările posibile ale unui fir de execuție.

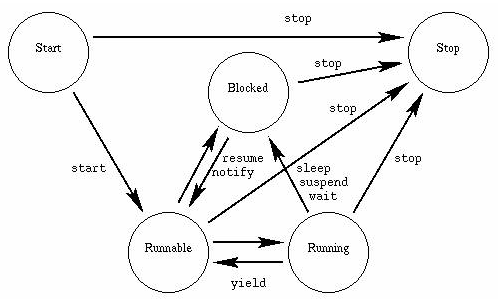


Figura 2.1.7.2. Stările posibile ale unui fir de execuție, împreună cu metodele aferente.

#### Crearea firelor de execuție

Pentru a crea un fir de execuție, se va proceda în aceeași manieră ca și în cazul creării unui obiect, deoarece un fir reprezintă o instanță a unei clase. Astfel, firele de execuție definite de o anumită clasă vor conține același cod și aceeașă secvență de instrucțiuni.

Acestea fiind spuse, o clasă care definește fire de execuție poate fi creată în două moduri:

* Prin extinderea clasei Thread;
* Prin implementarea interfeței Runnable.

În figura 2.1.7.3.1, respectiv 2.1.7.3.2, se pot observa modurile de creare a unui fir de execuție cu ajutorul clasei Thread, respectiv a interfeței Runnable.

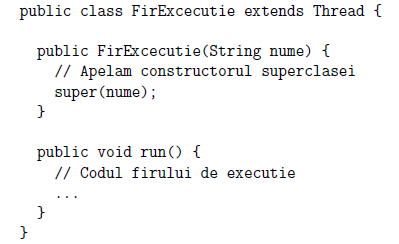


Figura 2.1.7.3.1. Crearea unui fir de execuție prin extinderea clasei Thread.

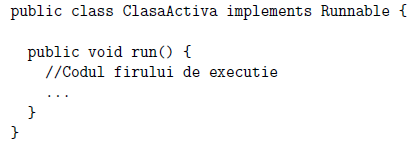


Figura 2.1.7.3.2. Crearea unui fir de execuție prin implementarea interfeței Runnable.

#### Ciclul de viață al unui fir de execuție

Așa cum este de așteptat, fiecare fir de execuție definit în cazul unui program are propriul său ciclu de viață: este creat, lansat și, după un timp, este oprit. Acest lucru se poate vedea, grafic, în figura de mai jos, în care un fir este creat (instanțiat), după care este lansat în execuție și, la un moment dat, este terminat (terminarea metodei run()).

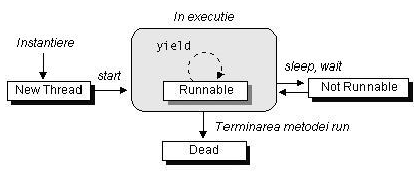


Figura 2.1.7.4. Ciclul de viață al unui fir de execuție

Trebuie acordată o deosebită atenție modului în care un fir de execuție se oprește. De exemplu, în cadrul clasei Thread, se găsește metoda stop() care a fost depreciată (nu este recomandat să fie folosită). Prin urmare, se evită utilizarea metodei stop(), iar în locul acesteia se va folosi metoda run(), în care se vor implementa niște blocuri condiționale pentru a decide în ce condiții se va opri firul de execuție.

Chiar dacă varianta utilizării metodei run() în loc de stop() este recomandată, există cazuri în care metoda run() determină blocarea unui fir și imposibilitatea de a verifica vreo condiție. Pentru oprirea forțată a firului blocat, se va utiliza metoda interrupt(), însă aceasta va genera o excepție de tipul InterruptedException, care va trebui prinsă și tratată.

#### Setarea priorităților firelor de execuție

Planificarea reprezintă procesul de execuție, într-o anumită ordine stabilită, a firelor de execuție pe un număr limitat de procesoare. Pentru a stabili ordinea execuției firelor, se vor seta prioritățile aferente în momentul în care firele de execuție sunt create.

Prioritatea unui fir de execuție stabilește importanța acestuia și transmite această informație planificatorului de execuție a firelor. Asta înseamnă că, dacă unul sau mai multe fire de execuție sunt blocate în timp ce își așteaptă execuția, planificatorul de execuție al firelor îl va alege pe cel cu prioritatea cea mai mare.

Câteva metode utile pentru lucrul cu fire de execuție și, implicit, cu prioritățile acestora sunt, după cum urmează:

* **getPriority():** obținerea priorității unui fir de execuție;
* **setPriority():** setarea priorității unui fir de execuție.
* **join():** determinarea unui fir de execuție să aștepte terminarea unui altul;

La nivelul sistemului de operare, lucrul cu fire de execuție este realizabil în două modele:

* **Modelul cooperativ:** firele de execuție decid când să elibereze procesorul (dezavantajul acestui model este acapararea procesorului de către unele fire, nepermițând execuția altor fire până la finalizarea celor curente);
* **Modelul preemptiv:** firele de execuție pot fi întrerupte oricând, după o perioadă de rulare, după care sunt reluate după ce și celelalte fire în execuție au avut acces la procesor (dezavantajul acestui model este nevoia de a sincroniza accesul firelor la resursele comune).

#### Sincronizarea firelor de execuție

Atunci când se lucrează cu fire de execuție, un aspect foarte important de care trebuie ținut cont este sincronizarea acestora, cu alte cuvinte, modul în care acestea accesează resursele comune.

Pentru a elabora noțiunea de sincronizare a două fire de execuție, se va lua în considerare situația firelor consumator/producător, în care un fir de execuție introduce date într-un buffer, iar celălalt fir de execuție citește datele puse de primul fir în buffer-ul respectiv. Este de la sine înțeles că cele două fire de execuție nu trebuie să acceseze în același timp buffer-ul, ci trebuie să aibă un acces sincronizat la acesta.

Pentru a nu se ajunge în situația în care două sau mai multe fire de execuție încearcă să acceseze aceeași resursă, se vor utiliza metode sau blocuri sincronizate. Acestea au rolul de a preveni coliziunea și de a păstra o ordine în execuția firelor.

O metodă (funcție) este sincronizată atunci când, înaintea tipului returnat, se află cuvântul cheie synchronized. O metodă sincronizată a unui obiect poate fi accesată doar de un singur fir de execuție la un moment dat; dacă se apelează o metodă sincronizată, monitorul este achiziționat de firul care a accesat metoda, prin urmare nicio altă metodă sincronizată nu va mai putea fi apelată cât timp metoda curentă este în execuție.

În figurile de mai jos sunt prezentate exemple de metode (funcții) și blocuri sincronizate.



Figura 2.1.7.6.1. Metode (funcții) sincronizate.



Figura 2.1.7.6.2. Bloc sincronizat.

Situațiile de interblocaje (deadlocks) reprezintă situații în care blocurile de sincronizare sunt folosite în mod greșit, ceea ce duce la blocarea firelor (se așteaptă unul după celălalt pentru eliberarea unui monitor). Pentru a bloca, respectiv debloca un fir de execuție, se folosesc metodele wait() și notify(), care manipulează monitoarele obiectelor.

De reținut este faptul că metodele wait() și notify() se pot apela doar din interiorul unor metode sau blocuri sincronizate, ceea ce înseamnă că pot fi apelate doar de către deținătorul monitorului obiectului.

## Baze de date

Deoarece utilizatorii au nevoie de un cont pentru a utiliza Visual OETPN Simulator – Desktop Application, vom avea nevoie de o bază de date care să stocheze toți utilizatorii înregistrați pentru a folosi aplicația. În continuare, se vor prezenta toate informațiile și noțiunile care au fost aplicate pentru realizarea scopului descris anterior.

În acest sens, se vor descrie următoarele:

* **Noțiunea de bază de date:** pentru a stoca diverse informații legate de aplicație, se va folosi o bază de date cu un anumit număr de tabele;
* **Baze de date relaționale:** pentru a reprezenta datele din tabele într-un mod simplu și intuitiv, se vor folosi bazele de date relaționale;
* **Limbajul de interogare SQL:** pentru elaborarea interogărilor pentru baza noastră de date, respectiv pentru a manipula datele necesare din tabele, se va utiliza limbajul SQL.

### Introducere în conceptul de bază de date

O bază de date reprezintă o colecție organizată și structurată de informații și date, care sunt stocate electronic într-un computer. Mai mult, o bază de date este, de obicei, controlată și monitorizată de un sistem de gestiune al bazelor de date (SGBD). Astfel, datele controlate și SGBD-ul care controlează aceste date, împreună cu aplicațiile care sunt asociate cu acestea, sunt văzute ca și un ansamblu (sistem) de baze de date.

Pentru a realiza, în mod eficient, interogarea și procesarea datelor din cele mai comune tipuri de baze de date, se utilizează o serie de tabele (cu rânduri și coloane). Prin utilizarea acestor tabele, datele vor putea fi accesate, gestionate, modificate, controlate, organizate și actualizate cu ușurință.

Tipurile de baze de date se împart în mai multe categorii, însă este de reținut că fiecare bază de date este utilizată în funcție de scopul pe care trebuie să îl îndeplinească. Prin urmare, avem:

* Baze de date relaționale;
* Baze de date orientate pe obiect;
* Baze de date distribuite;
* Baze de date open-source;
* Baze de date în cloud;
* Baze de date multimodel etc.

Pentru a îndeplini scopul acestei lucrări, s-au utilizat bazele de date relaționale. De asemenea, pentru scrierea și interogarea datelor dintr-o bază de date, se utilizează limbajul de interogare SQL.

### Baze de date relaționale

O bază de date relațională reprezintă un tip de baze de date, care permite stocarea și accesul la diverse puncte de date care sunt legate între ele. Bazele de date relaționale se bazează pe modelul relațional, care este un mod simplu și intuitiv de a reprezenta datele în tabele.

De regulă, o bază de date este personalizată în așa fel încât să se potrivească cu o anumită aplicație. Nu există două aplicații identice, prin urmare nici bazele lor de date nu vor fi identice. Fiecare creare de bază de date, indiferent de aplicația pentru care servește, trebuie să urmărească următorii pași:

1. **Definirea scopului bazei de date:** colectarea cerințelor și definirea obiectivelor bazei de date;
2. **Colectarea datelor, organizarea în tabele și stabilirea cheilor primare:** odată stabilit scopul bazei de date, se vor colecta datele care trebuiesc stocate în aceasta și, de asemenea, trebuiesc stabilite cheile primare (pentru identificarea unică a fiecărui rând din tabel);
3. **Crearea relațiilor dintre tabele:** aspectul crucial pentru definirea unei baze de date este identificarea relațiilor dintre tabele, care se împart în mai multe tipuri: one-to-many, many-to-many și one-to-one.

Câteva exemple de relații între tabele se pot observa în figurile de mai jos.

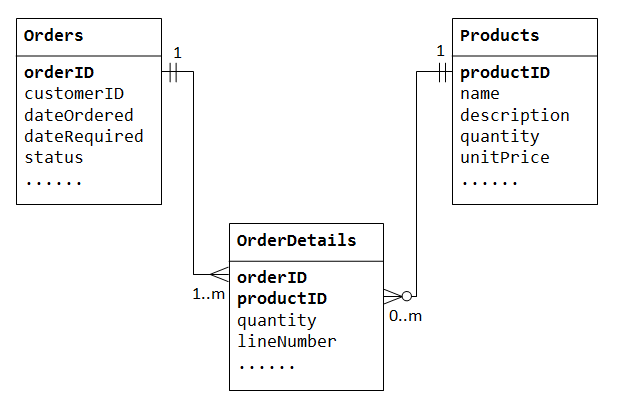


Figura 2.2.2.1. Relație many-to-many între tabele.

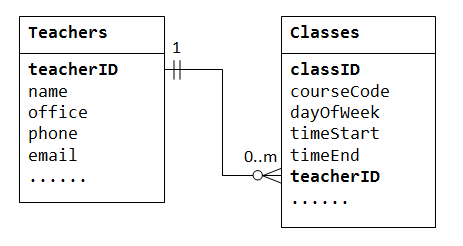


Figura 2.2.2.2. Relație one-to-many între tabele.

1. **Rafinarea și normalizarea design-ului:**

**Rafinarea** **design-ului** poate consta în diferite acțiuni, precum adăugarea mai multor tabele în structura bazei de date, crearea unui nou tabel pentru datele opționale utilizând relații de tip one-to-one, împărțirea unui tabel mai mare în două tabele mai mici etc.

Prin **normalizarea design-ului**, se verifică dacă baza de date este corectă și optimă din punct de vedere structural. Există mai multe reguli de normalizare, printre care se enumeră:

* Prima formă normală (First normal form – 1NF): un tabel este 1NF dacă fiecare celulă din tabel conține o singură valoare, și nu o listă de valori;
* A doua formă normală (Second normal form – 2NF): un tabel este 2NF dacă este 1NF și dacă fiecare coloană care nu este cheie primară este complet dependentă de cheia primară;
* A treia formă normală (Third normal form – 3NF): un tabel este 3NF dacă este 2NF și dacă fiecare coloană care nu este cheie primară este independentă una față de cealaltă.
* Forma normală superioară (Higher normal form): un tabel este de formă normală superioară dacă este descris de forma normală Boyce/Codd, a patra formă normală (Fourth normal form) sau de a cincea formă normală (Fifth normal form).

1. **Aplicarea regulilor de integritate:**

Pentru a verifica integritatea design-ului, se vor folosi următoarele reguli de integritate:

* Regula integrității entității (Entity Integrity rule): cheia primară nu poate fi NULL;
* Regula integrității referențiale (Referential Integrity rule): fiecare valoare a cheilor străine trebuie să coincidă cu o cheie primară din tabelul referențiat (tabelul părinte);
* Regula logicii de afaceri.

### Limbajul de interogare SQL

#### Introducere în SQL

Limbajul de interogare SQL (Structured Query Language – SQL) reprezintă un limbaj de programare standardizat, utilizat pentru gestionarea bazelor de date relaționale și pentru efectuarea diverselor operații asupra datelor din tabelele bazei de date. Acest limbaj standardizat de programare este utilizat atât de administratorii bazelor de date, cât și de dezvoltatori care elaborează script-uri de integrare a datelor sau analiști de date care doresc configurarea și rularea interogărilor.

Acest limbaj de programare standardizat se folosește în următoarele scopuri:

* Modificarea structurilor tabelelor și a indecșilor bazei de date;
* Adăugarea, actualizarea și ștergerea unor rânduri din tabele;
* Preluarea subseturilor de informații din cadrul sistemelor de management al bazelor de date (SGBD-uri); aceste informații pot fi folosite pentru a procesa tranzacții, aplicații de analiză și multe alte aplicații care au nevoie de o comunicare cu o bază de date.

În limbajul de programare standardizat SQL, interogările iau forma unor comenzi scrise sub forma unor instrucțiuni și agregate în programe care permit utilizatorilor să adauge, modifice și preia date din tabelele bazei de date.

#### Comenzile și sintaxa limbajului SQL

SQL este un limbaj de programare conceput pentru accesarea, modificarea și extragerea informațiilor din bazele de date relaționale. Ca orice alt limbaj de programare, SQL furnizează comenzi și dispune de o anumită sintaxă pentru crearea interogărilor pentru baza de date.

Comenzile în SQL se clasifică în anumite categorii, după cum urmează:

* **Limbajul de definire a datelor (Data Definition Language – DDL)** este folosit pentru a defini tabele de date.
* **Limbajul manipulării de date (Data Manipulation Language – DML)** este folosit pentru a manipula datele din tabelele existente prin adăugarea, modificarea sau ștergerea datelor.
* **Limbajul de interogare a datelor (Data Query Language - DQL)** constă într-o singură comandă (SELECT), utilizată pentru a obține date specifice din tabele.
* **Limbajul de control al datelor (Data Control Language – DCL)** este folosit pentru a acorda sau a retrage drepturi.
* **Limbajul de control al tranzacțiilor (Transaction Control Language – TCL)** este folosit pentru a schimba starea unor date.

O ierarhie a categoriilor descrise mai sus se poate observa în figura de mai jos.

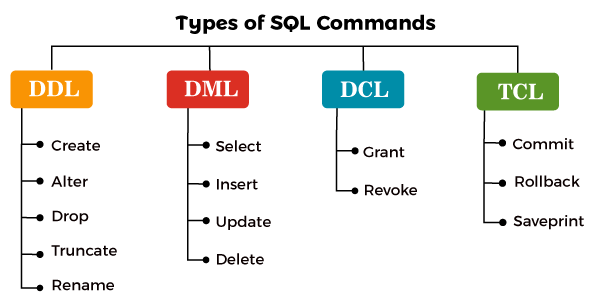


Figura 2.2.3.2.1. Tipurile de comenzi în limbajul de programare SQL.

Ca și sintaxă, limbajul de programare SQL se aseamănă cu celelalte limbaje de programare. În acest sens, se observă figura de mai jos, în care se selectează, din tabelul customers al unei baze de date, toate înregistrările.

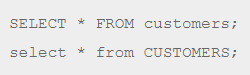


Figura 2.2.3.2.1. Exemplu de sintaxă uzuală a limbajului de programare SQL.

De reținut este că, în SQL, o comandă este încheiată doar după ce, la finalul acesteia, avem punct și virgulă. Mai mult, comenzile în SQL sunt insensibile la majuscule (case-insensitive), ceea ce înseamnă că acestea pot fi scrise cu litere mari, cu litere mici sau o combinație între acestea. Însă, o bună practică în scrierea comenzilor este de a scrie cuvintele cheie cu litere mari, iar numele coloanelor sau tabelelor utilizate cu litere mici.

#### Comenzi uzuale în limbajul SQL

Cele mai utilizate comenzi în limbajul de programare, indiferent de categoria din care fac parte, sunt:

* **SQL SELECT**, utilizată pentru a prelua o parte sau chiar toate datele dintr-un tabel;

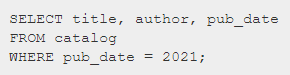


Figura 2.2.3.3.1. Exemplu de utilizare a comenzii SELECT.

* **SQL CREATE**, utilizată pentru a crea o nouă bază de date sau un tabel;



Figura 2.2.3.3.2. Exemplu de utilizare a comenzii CREATE pentru crearea unei baze de date.

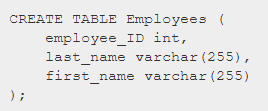


Figura 2.2.3.3.3. Exemplu de utilizare a comenzii CREATE pentru crearea unui tabel.

* **SQL UPDATE**, utilizată pentru a efectua modificări asupra rândurilor sau coloanelor dintr-un tabel.

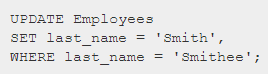


Figura 2.2.3.3.4. Exemplu de utilizare a comenzii UPDATE.

* **SQL DELETE**, utilizată pentru ștergerea unor rânduri dintr-un tabel.



Figura 2.2.3.3.5. Exemplu de utilizare a comenzii DELETE.

Toate aceste patru operații (**CREATE, SELECT, UPDATE, DELETE**) reprezintă operațiile de bază pentru lucrul cu baze de date, prescurtate cu acronimul **CRUD**.

## Crearea paginilor web

Pentru punerea la dispoziție a anumitor informații utile despre scopul, funcționalitățile și modul de utilizare a aplicației Visual OETPN Simulator – Desktop Application, dar și pentru furnizarea diferitelor resurse de documentație, s-a dezvoltat o pagină web (Visual OETPN Simulator – Web Application) cu ajutorul unor limbaje și tehnologii web. Mai mult, pagina Web dispune de o secțiune de descărcare a aplicației desktop (fișierul cu extensia .jar).

În continuare, se vor prezenta toate informațiile și noțiunile care au fost aplicate pentru realizarea scopului descris anterior. În acest sens, se vor descrie următoarele:

* **HyperText Markup Language (HTML):** deoarece pentru crearea unei pagini Web avem nevoie, mai întâi de toate, de stabilirea structurii acesteia, se va folosi limbajul HTML;
* **Cascading Style Sheet (CSS):** într-o pagină Web, se va folosi CSS pentru a stiliza un document HTML, dar și pentru a stabili modul în care elementele vor fi afișate;
* **JavaScript (JS):** pentru a crea conținut Web interactiv și dinamic, se va folosi limbajul de programare (scripting) JavaScript.

### HTML

#### Noțiuni introductive

HyperText Markup Language (prescurtat, HTML) reprezintă limbajul standard pentru crearea paginilor Web. De asemenea, HTML este un limbaj de marcare (NU un limbaj de programare), menit să descrie structura unei pagini Web și, totodată, ce elemente se regăsesc în aceasta.

Cu ajutorul limbajul de marcare HTML, împreună cu limbajul Cascade Style Sheet (CSS) și limbajul de scripting JavaScript (JS), se pot realiza pagini Web dinamice, cu interacțiuni între elemente sau cu utilizatorul.

Mai mult de atât, HTML permite:

* Publicarea unor documente care conțin text, tabele, imagini, liste etc.;
* Preluarea unor informații online cu ajutorul hyperlink-urilor;
* Proiectarea unor formulare pentru pagini care au ca scop efectuarea unor tranzacții, căutarea de informații, comandarea unor produse etc.;
* Includerea foilor de calcul, clipuri video, clipuri audio și alte aplicații.

În general, un fișier HTML conține următoarele marcaje (tag-uri):

|  |  |
| --- | --- |
| **Marcaj** | **Scop** |
| <!DOCTYPE html> | Indică browser-ului faptul că interpretarea textului este HTML5 |
| <html> | Marcajul rădăcină al unui document HTML |
| <head> | Conține metadată despre documentul HTML |
| <title> | Specificarea titlului pentru documentul HTML |
| <body> | Definește corpul documentului, reprezintă un container pentru toate componentele vizibile în pagină |

Tabelul 2.3.1.1.1. Marcajele de bază pentru o pagină Web.

Un simplu exemplu de pagină Web se poate observa în tabelul de mai jos. Prima coloană a tabelului descrie sursa, iar a doua coloană din tabel prezintă rezultatul (cum este pagina Web afișată în browser).

|  |  |
| --- | --- |
| **Sursă** | **Rezultat** |
|  |  |

Tabelul 2.3.1.1.2. Exemplu pagină Web

În sursa de mai sus, se setează titlul paginii cu ajutorul marcajului (tag-ului) <title>, iar corpul paginii se setează cu marcajul (tag-ul) <body>. Se observă că numele marcajelor sunt intuitive, făcând astfel sursa ușor de înțeles și de citit.

Pentru a verifica faptul că titlul și corpul paginii au fost adăugate, se va salva sursa ca și fișier cu extensia .html, după care se va deschide într-un browser. Rezultatul va fi cel din coloana dreaptă a tabelului 2.3.1.1.2.

#### Structura unei pagini Web

Pentru crearea unei pagini Web, trebuie să se țină cont de o anumită structură pentru ca elementele din respectiva pagină să fie afișate și poziționate corect. În figura de mai jos, se poate vedea o astfel de structură a unei pagini HTML, în care vom avea marcajele de bază (<html>, <head>, <title> și <body>) și alte marcaje care descriu modul în care este afișat textul în pagină.

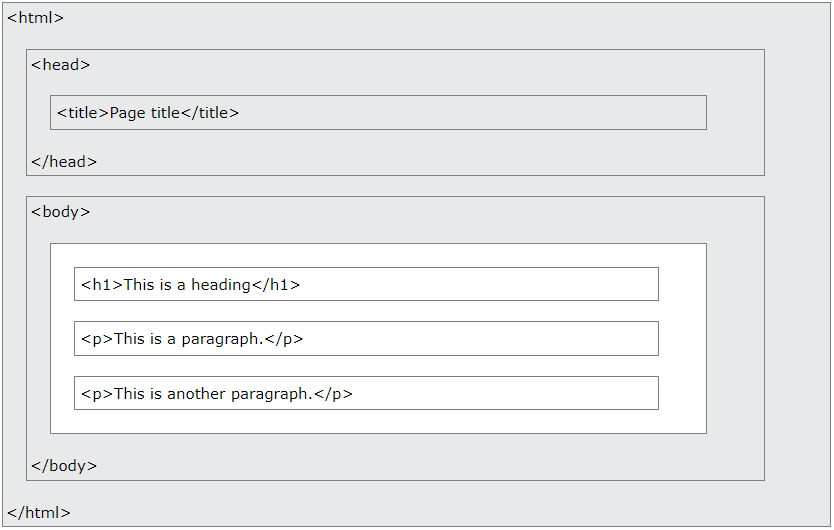


Figura 2.3.1.2. Structura unei pagini Web.

#### Marcaje utilizate

În HTML, se utilizează diferite marcaje (tag-uri) pentru a defini anumite componente din pagina Web. Un tabel cu câteva dintre marcajele cel mai des folosite se regăsește mai jos, împreună cu rolurile pe care le servesc.

|  |  |
| --- | --- |
| **Marcaj** | **Scop** |
| <!--> | Comentariu |
| <h1> la <h6> | Subtitluri |
| <p> | Paragraf |
| <br> | Linie nouă |
| <a> | Ancoră |
| <b> | Text îngroșat |
| <i> | Text italic |
| <u> | Subliniere |
| <button> | Buton de tip push-button |
| <div> | Secțiune în document |
| <form> | Formular |
| <img> | Imagine |
| <input> | Zonă de introducere |
| <li> | Element de listă |
| <link> | Referință către o sursă |
| <ol> | Listă ordonată |
| <ul> | Listă neordonată |
| <script> | Definirea de scripturi client |
| <section> | Definirea unei secțiuni sub formă de bloc |
| <span> | Secțiune de document de tip inline |
| <strong> | Text îngroșat cu rol de formatare logică |
| <style> | Definiție de stil |
| <sub> | Indice inferior |
| <sup> | Indice superior |
| <table> | Tabel |
| <video> | Element video |

Tabelul 2.3.1.3. Cele mai folosite marcaje în HTML.

În general, marcajele HTML au fost gândite pentru a defini conținutul unei pagini Web prin specificarea componentelor acesteia (antete, subsoluri, butoane, paragrafe etc.), însă modul de reprezentare al documentului, per total, urma să fie stabilit în funcție de browser-ul care deschidea documentul.

Deoarece, pe parcursul timpului, s-au adăugat noi marcaje HTML de font și atribute de culoare, crearea paginilor Web în care conținutul să fie separat de prezentare devenise din ce în ce mai greu. Astfel, World Wide Web Consortium (W3C) a creat limbajul CSS, pentru definirea acestor stiluri.

### CSS

#### Noțiuni introductive

Cascading Style Sheets (prescurtat, CSS), reprezintă limbajul standard pentru definirea stilurilor unei pagini Web. De asemenea, CSS este un limbaj menit să descrie modul în care elementele HTML sunt afișate pe ecran și care este capabil de a controla aspectul a mai multor pagini Web simultan.

Există trei moduri de a specifica un stil pentru o pagină Web:

* În cadrul unui singur element CSS;
* În interiorul elementului <head> al unei pagini HTML;
* În cadrul unui fișier CSS (se pot încărca mai multe fișiere de acest tip într-o pagină HTML).

Toate aceste stiluri menționate mai sus se vor cascada într-un stil virtual, după regulile de mai jos. Prioritățile acestora sunt în ordine crescătoare, după cum urmează:

* Stiluri implicite ale browser-ului;
* Stiluri specificate în fișiere CSS externe;
* Stiluri interne ale paginii HTML, specificate în marcajul <head>;
* Stiluri inline, specificate în interiorul elementului HTML.

#### Sintaxa CSS

Sintaxa definirii stilurilor unei pagini Web sau element HTML este formată din trei părți:

* **Selectorul:** este, de obicei, elementul HTML pentru care se face stilizarea;
* **Proprietatea:** este atributul care se va specifica pentru elementul HTML;
* **Valoarea:** este valoarea priorității specificate.

Proprietatea și valoarea sunt separate (despărțite) prin “:” și sunt înconjurate de acolade.

Un exemplu de definire a unui stil este cel din figura de mai jos. Selectorul este subtitlul (heading) h1, la care i se vor aplica proprietatea de culoare („color”) cu valoarea “blue” și proprietatea de dimensiunea fontului (“font-size”) cu valoarea “12px”.

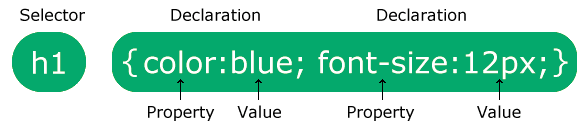


Figura 2.3.2.2.1. Definirea stilurilor

Mai mult, în cazul în care două sau mai multe elemente HTML dintr-o pagină Web trebuie să aibă aceleași stiluri, selectorii acestora se pot grupa ca în exemplul de mai jos. Se observă că selectorii sunt grupați și separați prin virgulă.



Figura 2.3.2.2.2. Gruparea selectorilor

Pentru a defini mai multe stiluri pentru același element HTML, se utilizează clasele CSS. Un astfel de exemplu se observă în figura de mai jos: pentru elementul HTML <div>, se vor defini trei stiluri: div.center, div.left și div.right, care va stiliza întregul conținut din element corespunzător clasei care se va utiliza.



Figura 2.3.2.2.3. Definirea claselor CSS

Este de reținut faptul că, pentru definirea unei clase CSS, nu este obligatorie specificarea numelui elementului HTML în definiția clasei CSS. Dacă se va omite numele din definiția clasei CSS, stilul definit în respectiva clasă se poate aplica oricărui element HTML.

Mai mult, se pot defini stiluri pentru elementele HTML prin specificarea ID-ului acestora, împreună cu prefixul #. Un exemplu concret pentru această metodă de aplicare a stilurilor se poate observa în figura de mai jos. Primul stil se va aplica elementului care are ID-ul “front”, iar al doilea stil se va aplica doar unui paragraf care are ID-ul “front”.

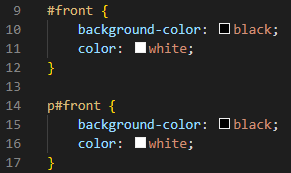


Figura 2.3.2.2.4. Definirea stilurilor elementelor HTML în funcție de ID.

#### Integrarea stilurilor CSS în documentul HTML

1. **Fișierele de stiluri externe:**

Folosirea stilurilor externe se referă la integrarea unui fișier de stiluri (fișier cu extensia .css) în secțiunea <head> a unui document HTML. Prin abordarea acestei metode, se pot implementa stiluri pe mai multe pagini Web cu un singur fișier CSS.

Un exemplu de utilizare a stilurilor externe se poate observa în figura de mai jos. Fișierul CSS “styles.css” este integrat în documentul HTML în secțiunea <head> prin folosirea marcajului <link>.



Figura 2.3.2.3.1. Folosirea fișierelor de stiluri externe.

1. **Crearea stilurilor interne documentului HTML:**

Folosirea stilurilor interne se recomandă atunci când un stil se aplică numai unui singur document HTML. Pentru includerea unui astfel de stil într-un document HTML, se va utiliza marcajul <style>, în secțiunea <head> a documentului HTML.

Un exemplu de utilizare a stilurilor interne se poate observa în figura de mai jos. Selectorii, împreună cu prioritățile și valorile definite, sunt cuprinse în marcajul <style>, care se află în secțiunea <head>.

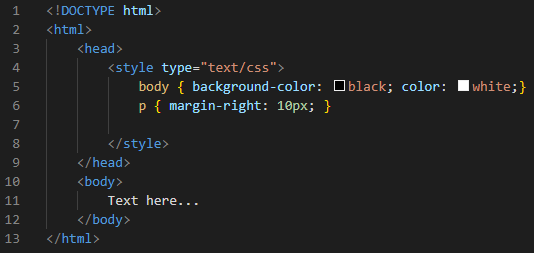


Figura 2.3.2.3.2. Folosirea stilurilor interne.

1. **Crearea stilurilor inline, definite în cadrul elementelor HTML:**

Stilurile inline se folosesc atunci când un element HTML dintr-o pagină Web apare o singură dată în document. Deși este o metodă des utilizată, aceasta duce la pierderea avantajelor CSS, deoarece în cazul în care se utilizează această metodă pentru un element HTML care apare de mai multe ori, se vor amesteca informațiile legate de conținutul paginilor HTML cu modul de prezentare al acestora.

Pentru a scrie un stil inline, se va folosi atributul “style” în interiorul marcajului pentru care se va face stilizarea. Un exemplu de folosire a stilurilor inline se poate observa în figura de mai jos: pentru elementul HTML <p>, se vor aplica proprietatea de culoare cu valoarea “blue” și proprietatea de margine stânga cu valoarea “15px”.

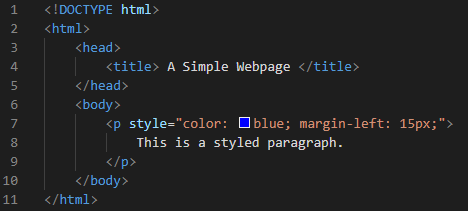


Figura 2.3.2.3.3. Folosirea stilurilor inline.

#### Clasificarea stilurilor CSS

CSS permite stilizarea pentru următoarele componente dintr-o pagină Web:

1. **Stiluri pentru fundal:** permit controlul fundalului oricărui element (de exemplu, modificarea culorii fundalului, setarea unei imagini ca și fundal, poziționarea imaginii de fundal etc.);
2. **Stiluri pentru text:** permit controlul asupra textului dintr-un document HTML (de exemplu, schimbarea culorii textului, modificarea spațiului dintre caractere, alinierea textului, indentarea textului etc.);
3. **Stiluri pentru font:** permit controlul asupra fonturilor textului dintr-un document HTML (de exemplu, schimbarea familiei de fonturi, schimbarea stilului textului etc.);
4. **Stiluri pentru chenar:** permit schimbarea stilului și culorii chenarului unui element dintr-un document HTML;
5. **Stiluri pentru margine:** permit definirea spațiilor din jurul elementelor;
6. **Stiluri pentru distanțare:** permit definirea spațiului de la marginea elementului până la conținutul elementului;
7. **Stiluri pentru liste:** permit controlul listelor (de exemplu, schimbarea simbolului de marcaj, înlocuirea sau poziționarea acestuia etc.);

### JavaScript

#### Noțiuni generale

JavaScript este unul dintre cele mai populare limbaje de programare la ora actuală, folosit pentru dinamizarea paginilor Web. Este un limbaj de scripting, folosit pentru a oferi instrucțiuni către alte software-uri (în acest caz, browser-ele Web) și de a introduce anumite funcționalități în paginile Web.

Un script reprezintă un program înglobat în fișierul HTML și care este interpretat și executat de browserul Web, în momentul în care se încarcă documentul HTML. Un script poate modifica în mod dinamic pagina Web la apariția unor evenimente (click al mouse-ului, apăsarea unei taste etc.).

Chiar dacă limbajele de programare JavaScript și Java conțin similarități în nume și sintaxă, între cele două nu există nicio legătură.

#### Integrarea script-urilor JavaScript în documentul HTML

1. **Integrarea script-ului în corpul fișierului HTML:**

Această modalitate se referă la adăugarea unui marcaj <style> în secțiunea <body> a unui document HTML. Un exemplu care ilustrează această modalitate de integrare a script-urilor se poate observa în figura de mai jos.

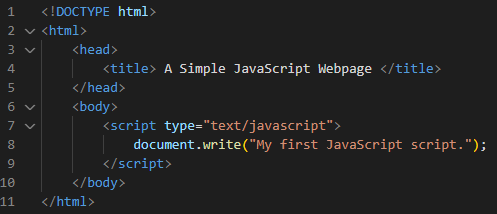


Figura 2.3.3.2.1. Integrarea unui script în corpul unui document HTML.

1. **Integrarea script-ului în partea de antet:**

Această modalitate se utilizează atunci când se dorește ca acesta să se execute înainte de afișarea oricărui element HTML pe pagină. Un exemplu care ilustrează această modalitate se poate observa în figura de mai jos.

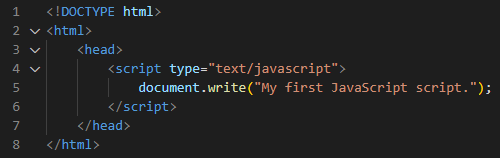


Figura 2.3.3.2.2. Integrarea unui script în partea de antet.

1. **Integrarea script-urilor dintr-un fișier extern:**

Ca și în cazul fișierelor .css, un script extern (un fișier cu extensia .js) poate fi folosit pe mai multe pagini Web, acesta fiind scris o singură dată, ci nu în fiecare fișier HTML. În acest caz, fișierul .js nu poate conține marcajul <script>, ci va conține doar codul necesar pentru definirea script-ului.

Un exemplu de integrare a unui script dintr-un fișier extern într-un document HTML se poate observa în figura de mai jos.

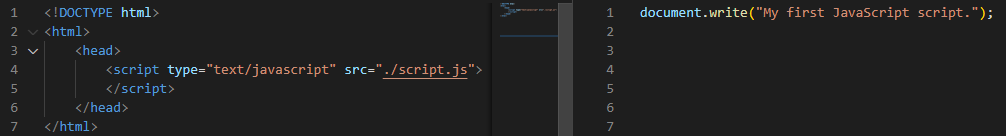


Figura 2.3.3.2.3. Integrarea unui script dintr-un fișier extern.

## 

# Analiză, proiectare, implementare

Aceasta parte a lucrării este flexibilă și depinde foarte mult de natura lucrării, poate fi organizată în mai multe capitole și conține contribuțiile personale ale autorului.

Includeți:

* + Detalii referitoare la analiză și proiectare:
    - descrierea metodelor pe care le-ați aplicat pentru rezolvarea problemei,
    - descrierea materialelor, procedurilor
    - calcule, tehnici, descrierea echipamentelor
    - metodologia de proiectare
    - informațiile necesare pentru ca cineva să poată reface lucrarea
  + Implementare :
    - Descrieți detaliile tehnice ale implementării aplicației: mediul de implementare, modul de prezentare, modul de utilizare al aplicației, etc.
  + Testare si validare :
    - Descrieți metodologia de testare a aplicației și rezultatele
    - Includeți experimentele pe care le-ați realizat, analiza rezultatelor pe care le-ați obținut.

# Concluzii

## Rezultate obținute

Evidențiați toate rezultatele pe care le-ați obținut și trageți concluzii din ele. Puteți prezenta o analiză critică a ceea ce ați realizat comparativ cu alte lucrări/studii anterioare.

Includeți o listă a contribuțiilor pe care le-ați avut în domeniul temei abordate.

## Direcții de dezvoltare

Pentru a îmbuntăți calitatea aplicației desktop din punct de vedere al interfeței grafice a acesteia, se pot lua în considerare următoarele direcții de dezvoltare:

* Crearea unei ferestre de bun-venit, vizibilă după autentificarea/înregistrarea utilizatorului, în care utilizatorul are acces rapid la secțiunile din aplicația Visual OETPN Simulator – Web, precum: pagina principală, pagina cu informații utile despre rețele Petri și modele OETPN, documentații, contact și pagina de descărcare a aplicației;
* Crearea unui panou cu file (JTabbedPane), în editorul principal, pentru specificarea, proiectarea, implementarea și simularea a mai multor rețele Petri și modele OETPN în paralel;
* Crearea unor opțiuni de zoom (zoom in, zoom out), în editorul principal, pentru o mai bună vedere a rețelei Petri sau a modelului OETPN, în cazul în care aceasta ocupă mult loc în editorul principal;

De asemenea, o altă direcție de dezvoltare interesantă este implementarea unei posibilități pentru specificarea, proiectarea și simularea modelelor FLETPN (Fuzzy Logic Enhanced Petri Nets), aplicația căpătând astfel nu numai un nume nou (Visual OETPN & FLETPN Simulator), dar și o funcționalitate utilă, realizându-se astfel o aplicație centralizată, capabilă să simuleze tipurile de rețele Petri familiare.

Mai mult, pentru site-ul Visual OETPN Simulator – Web, pe lângă posibilitățile de documentare din sursele puse la dispoziție, vizualizarea demonstrațiilor video a aplicației Visual OETPN Simulator și posibilitatea de descărcare a aplicației din aceasta, este posibilă implementarea unei noi secțiuni, care constă într-un editor și simulator exclusiv online pentru rețelele Petri clasice și modele OETPN.

# Bibliografie

[1] Cristian Frăsinaru, Curs practic de Java, București, Editura Matrix Rom.

[2] “Inheritance”, The Java Tutorial [Online]. Available: <https://docs.oracle.com/javase/tutorial/java/IandI/subclasses.html>

[3] Mihai Hulea, Cursuri ISP [Online]. Available: <http://207.154.194.102>

[4] Ștefan Tanasă, Cristian Olaru, Ștefan Andrei, “Clase, interfețe și tablouri” în Java de la 0 la expert, Iași, Editura Polirom, 2003.

[5] “What is a Database?”, Oracle [Online]. Available: <https://www.oracle.com/database/what-is-database/>

[6] “A Quick-Start Tutorial on Relational Database Design” [Online]. Available: <https://www3.ntu.edu.sg/home/ehchua/programming/sql/Relational_Database_Design.html>

[7] “Database design basics (Microsoft Access 2007

[7] Peter Loshin, „Structured Query Language (SQL)” [Online]. Available: <https://www.techtarget.com/searchdatamanagement/definition/SQL>

[8] Szilard Enyedi, Tehnologii Web, suport de curs, Cluj-Napoca.

[9] “HTML Introduction”, W3Schools [Online]. Available: <https://www.w3schools.com/html/html_intro.asp>

[10] “CSS Introduction”, W3Schools [Online]. Available: <https://www.w3schools.com/css/css_intro.asp>

[11] „JavaScript Introduction”, W3Schools [Online]. Available: <https://www.w3schools.com/js/js_intro.asp>

|  |  |
| --- | --- |
| [1] | P. Nume, "Titlul capitolului," în *Titlul cartii*, Oras, Editura, 2016, pp. 1-24. |
| [2] | P. Nume, "Titlul articolului," *Titlul revistei,* vol. 1, no. 2, pp. 22-30, 2016. |
| [3] | P. Nume, "Titlul articolului," în *Numele conferintei*, Oras, 2015. |
| [4] | "IEEE Citation Reference," 2009. [Online]. Available: https://www.ieee.org/documents/ieeecitationref.pdf. |
| [5] | "IEEE Editorial Style Manual," 2016. [Online]. Available: https://www.ieee.org/documents/style\_manual.pdf. |