UNIVERSITATEA “LUCIAN BLAGA” DIN SIBIU
FACULTATEA DE INGINERIE
DEPARTAMENTUL DE CALCULATOARE ŞI INGINERIE ELECTRICĂ

PROIECT DE DIPLOMĂ

Conducător ştiinţific: Asist. Dr. Ing. Dorobanțiu Alexandru
Îndrumător: Asist. Dr. Ing. Dorobanțiu Alexandru

Absolvent
Banea Marcus-Andrei

Specializarea
Calculatoare

- Sibiu, 2023 -

UNIVERSITATEA “LUCIAN BLAGA” DIN SIBIU

FACULTATEA DE INGINERIE
DEPARTAMENTUL DE CALCULATOARE ŞI INGINERIE ELECTRICĂ

Concurs de boți implementat in cadrul unui joc de carti

Conducător ştiinţific: Asist. Dr. Ing. Dorobanțiu Alexandru
Îndrumător: Asist. Dr. Ing. Dorobanțiu Alexandru

Absolvent
Banea Marcus-Andrei

Specializarea
Calculatoare

- Sibiu, 2023 -

Cuprins

1. Prezentarea temei (2-4 pagini):

1.1. Introducere

Chemarea mea catre acest proiect isi are originile in unele dintre cele mai frumoase perioade ale vietii: copilaria, si perioada studentiei. Jocul de carti Duel Masters, unul dintre obiectele principale ale acestui proiect, a facut parte din copilaria mea, fiind captivat atat de interactiunea pe care o aveam cu ceilalti prieteni cu care jucam, cat si de complexitatea acestuia.

Anii au trecut, copilaria a ramas in urma, iar ideea acestui joc a ramas pentru mine doar o amintire. In schimb, o alta pasiune a aparut: programarea. Liceul m-a introdus in aceasta lume, iar studentia mi-a aratat ca singurul obstacol din acest domeniu este imaginatia.

Astfel, avand oportunitatea de alege o tema de proiect pentru incheierea studiilor universitare, am decis sa utilizez cunostiintele dobandite in ultimii ani si sa combin cele 2 elemente care au marcat cele 2 perioade ale vietii mele.

1.2. Descrierea temei

Dupa cum se precizeaza si in titlul temei alese, aplicatia dezvoltata va presupune realizarea unui mediu de desfasurare a unui meci de Duel Masters, intre boti. Acesti boti vor inlocui jucatorul real, si vor lua decizii in functie de contextul si starea meciului. Calitatea deciziilor va fi diversa, intrucat jocul este unul complex.

Totodata, aplicatia va permite desfasurarea de meciuri intre jucatori reali (useri) si boti. Nivelul de dificultate al meciului va fi reprezentat de catre algoritmul din spatele bot-ului. Despre metodologia de luare a unei decizii se va vorbi, mai tarziu, in capitolul ...

1.3. Scop si obiective

Dezvoltarea aplicatiei a urmarit cateva obiective principale, prioritare, care reprezentau baza temei. Pe langa acestea, vom prezenta si cate obiective secundare, dezvoltate pe baza celor principale, ce aduc noi functionalitati si imbogatesc placut aplicatia.

1.3.1. Obiective principale

1.3.1.1. Obtinerea setului de carti

Dupa cum am precizat anterior, jocul Duel Masters reprezinta un joc de carti. Fiecare carte are utilizari diferite, determinate de diferiti factori precum: starea curenta a meciului, mutarile facute de cei 2 jucatori in tururile anterioare ale meciului, etc.

Figura 1.0: Carte de tip Creature

Astfel, avand in considerare timpul disponibil dezvoltarii acestui proiect, cat si dorinta autorului de a realiza o aplicatie functionala, care sa prezinte cat mai multe dintre elementele acestui joc, a fost necesara trierea setului complet de carti ale jocului.

Setul contine 3 tipuri principale de carti:

1. Creature (creaturi) : carti pe care cei 2 jucatori le plaseaza pe terenul de joc si care participa la dueluri intre ele, fiind elementul principal al jocului. O astfel de carte este prezentata in figura 1.1.

2. Spell: carti cu one-time effect; cu alte cuvinte, aceste carti sunt utilizate pentru a realiza o actiune (abilitate) in momentul utilizarii, dupa care sunt eliminate din joc

3. Evolution (evolutii): carti similare cu cele de tip Creature, diferenta reprezentand-o necesitatea de a plasa o astfel de carte deasupra unei carti de tip Creature, realizand, dupa cum ii spune si numele, o evolutie a cartii de baza.

Figura 1.1: Carte de tip Spell

Cartile de tip Creatura reprezinta majoritatea cartilor din setul complet de carti, iar din acest motiv s-a ales utilizarea, pentru dezvoltarii primei versiuni a aplicatiei, doar acest tip de carti.

Figura 1.2: Carte de tip Evolution

Alti factori care au contribuit la trierea setului de carti:

- Limitarea la doar cateva tipuri speciale de Creatura (eliminarea celor de tip Cross Gear, Wave Striker sau Accelerator)

- Imaginile cartilor propriu-zise: intru-cat jocul isi are originile in Japonia, exista carti pentru care nu s-au putut obtine imagini cu textul inscris pe acestea, in limba engleza.

Dintr-un total de aproximativ 1400 de carti, s-a obtinut un set de 433 de carti de tip Creatura ce vor fi utilizate in prima faza a dezvoltarii aplicatiei.

1.3.1.2. Realizarea interfetei jocului

Pentru desfasurarea unui meci, fie el intre 2 boti, 1 bot si un jucator real (user) sau intre 2 jucatori (planificat pentru dezvoltari ulterioare), este necesara crearea unei interfete vizuale, in care sunt dispuse cele 2 table de joc, corespunzatoare fiecarui jucator.

Stuctura unei table de joc este urmatoarea:

- Battle Zone: zona de batalie, unde sunt plasate cartile de tip Creatura ce se vor duela cu cele ale jucatorului advers.

- Mana Zone: mana reprezinta resursa necesara pentru a putea plasa in zona de batalie o carte. Fiecare carte va consuma o anumita cantitate de mana, aceasta valoare fiind inscriptionata pe carte. Astfel, fiecare carte plasata in zona de mana va creste capacitatea jucatorului de a plasa carti in zona de batalie.

- Hand: cartile din mana jucatorului. Acestea pot fi plasate in una din cele 2 zone prezentate anterior.

- Graveyard: zona cartilor distruse. In urma duelurilor sau executiei abilitatiilor, cartile ce vor fi distruse vor fi plasate in aceasta zona.

- Deck: pachetul de carti. In fiecare tura, jucatorul poate lua o carte din acest pachet, fapt ce asigura continuitatea meciului.

- Shields: scuturile jucatorului. Acestea reprezinta punctele de viata ale jucatorului. Jucatorul care ramane, in timpul meciului, fara nici un scut, va pierde meciul.

1.3.1.3. Implementarea mecanicii jocului

Jocul prezinta un set de reguli si mecanici care se aplica in diferite momente ale meciului. Spre exemplu: in fiecare tura, un jucator poate lua o singura carte din pachet. Aceste mecanici vor fi implementate si integrate atat pentru conceptul de meci bot vs bot, cat si in celelalte variante.

1.3.2. Obiective secundare

Obiectivul principal este, in mod evident, crearea unui mediu prin care poate avea loc un meci de Duel Masters. Realizarea acestui obiectiv necesita indeplinirea altor cateva obiective, dintre care enumeram:

- Optiunea de a crea mai multi useri (playeri)
- Posibilitatea unui user de a isi gestiona colectia de carti:
    - Vizualizarea colectiei
    - Renuntarea la cartile nedorite
    - Imbogatirea colectiei cu alte carti, prin cupararea de pachete de carti sau schimburi de carti cu alti jucatori

1.4. Tehnologii utilizate

Arhitectura aplicatiei prezinta 3 componente principale: serverul, clientul si baza de date.

Tehnologiile utilizate pentru gestionarea celor 3 sunt urmatoarele:

- Server: Limbajul de programare JAVA; Mediul de programare Eclipse (varianta SpringToolSuite4)
- Client: Limbajul de programare JavaScript; Frameworkul Vue.js; Mediul de programare Visual Studio Code
- Baza de date: baza de date document-oriented MongoDB; software-ul utilizat pentru administrarea bazei de date: MongoExpress

Totodata, pentru testarea request-urilor venite dinspre client spre server, am utilizat API-ul Postman.

1.4.1. Biblioteci

Impreuna cu framework-ul Vue.js, am utilizat librariile Pinia si Xstate.

Pinia reprezinta o librarie utila pentru managementul starilor din ecosistemul Vue. La randul ei, Xstate reprezinta o librarie utila pentru implementarea conceptului de finite state machine (aparat cu stari finite). Cele 2 s-au dovedit a fi cruciale in procesul de dezvoltare a aplicatiei, ajutand la o gestiune eficienta si fluida a informatiei cu care lucreaza aplicatia.

1.5. Cazuri principale de utilizare a aplicatiei

Aplicatia poate fi observata din 2 perspective. Partea de concurs de boti se adreseaza unui public mai rezervat, care urmareste dezvoltarea si implementarea unor algoritmi de joc asupra unui subiect, care in cazul acesta este jocul Duel Masters. Totodata, jocul propriu-zis este adresat in special copiilor, insa complexitatea acestuia poate crea interes si din partea unor persoane mai mature.

1.6. Organizarea lucrarii

2. Teorie (25-30 pagini)

2.1. Source Control

Precum orice proces de dezvoltare a unei aplicatii, nu vor exista niciodata doar 2 etape: faza initiala, in care aplicatia este complet nefunctionala, si faza finala, in care aplicatia prezinta toate functionalitatiile dorite de proiectant. Procesul va fi marcat de o succesiune de pasi ce vor fi urmati pentru a realiza produsul final.

Fara a intra in detalii specifice acestui proiect, enumeram cateva situatii care prezinta mai clar pasii amintiti anterior, cat si impedimentele provocate de acestia:

- Implementarea unei functionalitati specificate de proiectant

- Aparitia unor schimbari de proiectare, ce necesita modificarea unei functionalitati deja implementate
- Rezolvarea unor bug-uri aparute in urma implementarii

Se observa clar necesitatea pastrarii unei istorii a versiuniilor aplicatiei, pe parcursul dezvoltarii. Pentru a realiza asta, s-a decis utilizarea unui sistem de control al versiunilor unui proiect: GIT.

2.1.1. GIT

Git reprezinta unul dintre cele mai utilizate sisteme de control al versiunilor. Acesta a fost creat de catre Linus Torvalds, creatorul Linux, in anul 2005.

Este folosit pentru a salva si urmarii modificarile ce sunt efecutate asupra fisierelor unui proiect. Astfel, dezvoltatorul proiectului poate urmarii atat stadiul actual al acestuia, cat si starea la diferite momente de timp.

Aceste proprietati il transforma in tool-ul esential colaborarii in echipele de dezvoltare al unui soft. Mai multi developeri pot lucra simultan pe acelasi proiect si isi pot share-ui progresul, functionalitatile Git-ului permitand aceasta tranzitie eficienta si fara conflicte. Despre acestea se va detalia in cele ce urmeaza.

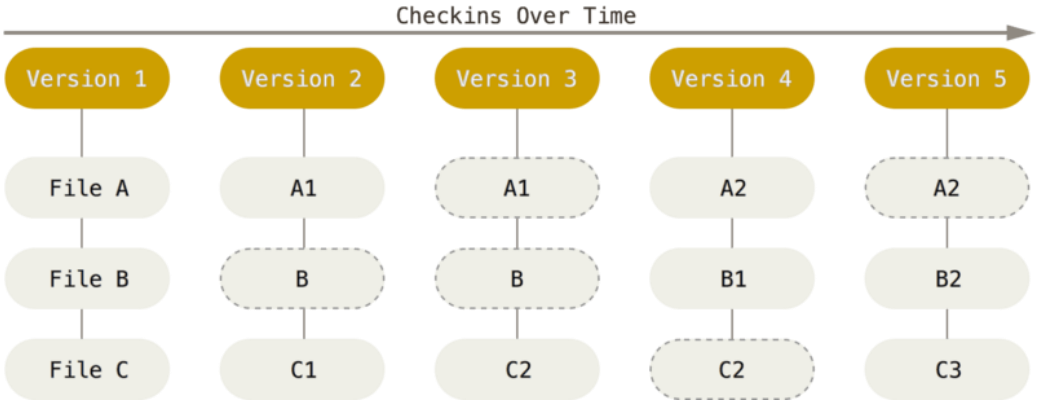

Figura 2.1.1. Salvarea datelor sub forma de snapshots

2.1.1.1. Functionalitatiile Git-ului

Elementul central in cadrul utilizarii Git-ului este bine cunoscutul Repository (sau, pe scurt, repo). Acesta reprezinta, practic, un spatiu de stocare in care se afla atat fisierele, cat si istoricul modificarilor (commit-urile).

Pentru a crea un repository, trebuie ales un folder de catre utilizator, si apoi anuntat Git-ul printr-o comanda ce va seta acel folder ca fiind un repository. Orice schimbare adusa fisierelor din acest folder va fi remarcata de catre Git.

Aceste fisiere se pot afla in una din cele 2 stari: tracked (urmarite) si untracked (neurmarite). Fisierele untracked reprezinta fisiere nou adaugate, ce nu se regasesc in ultimul snapshot. Fiserele tracked sunt fisiere ce se regasesc in ultimul snapshot.

Pe masura ce modificam un fisier, acesta trece prin urmatoarele stari:

1. Untracked: fisier pe care Git-ul nu l-a „vazut” pana in acest moment
2. Unmodified: fisier existent in ultimul snapshot, si asupra caruia nu s-au efectuat inca modificari
3. Modified: fisier existent in ultimul snapshot, si care a fost modificat
4. Staged: fisier modificat, ce va fi inregistrat in urmatorul snapshot

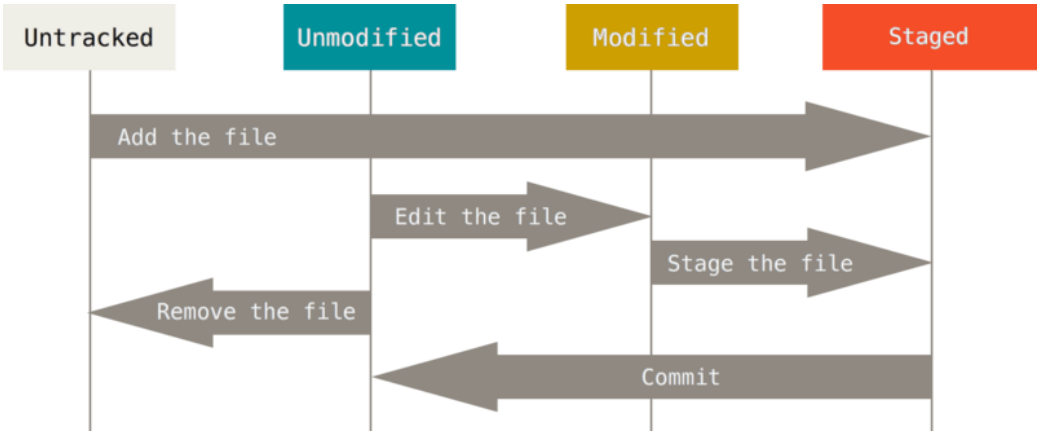

Figura 2.1.2. Ciclul de viata al starilor unui fisier

Aceste stari prezinta circuitul pe care un fisier il urmeaza local. O imagine de ansamblu, ce include si legatura cu Repository-ul, ar fi urmatoarea:

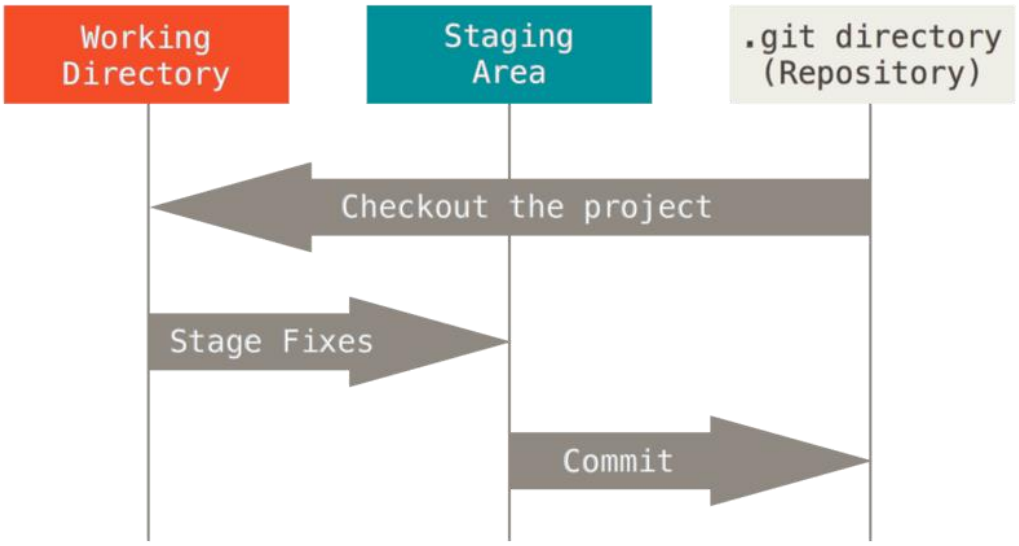

Figura 2.1.3. Structura unui proiect si tranzitiile intre stari: directorul de lucru, zona de staging si repository-ul

Se pot observa tranzitiile intre stari:

- Directorul de lucru → Staging Area: marcarea fisierelor ce se doresc a fi salvate in urmatorul snapshop din repository
- Staging Area → Repository: salvarea fisierelor (commit)
- Repository → Directorul de lucru: preluarea starii proiectului la un anumit moment de timp (checkout)

2.1.1.2. Utilizarea Git-ului

Git-ul poate fi utilizat prin diferite metode. Utilizarea command-line-ului reprezinta metoda clasica, insa pot fi utilizate si alte interfete grafice (GUI) precum SmartGit sau SourceTree.

Diferenta dintre cele 2 metode consta in facilitatile pe care le ofera cea din urma: spre deosebire de linia de comanda, unde este necesar a fi cunoscute comenzile propriu-zise, un client GUI pentru Git simplifica managementul Repository-ului.

2.1.1.3. SmartGit

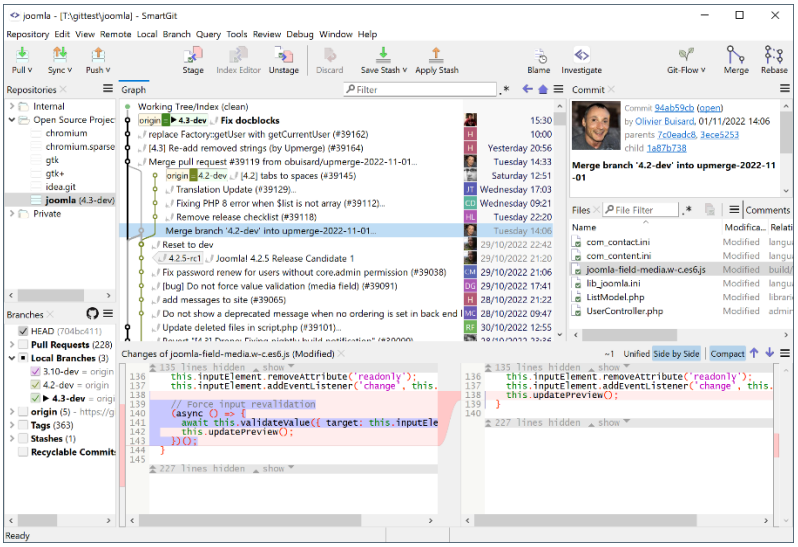

Figura 2.1.2. Interfata grafica a clientului SmartGit

SmartGit reprezinta un client grafic pentru Git, cu ajutorul caruia se poate gestiona un repository. SmartGit ofera functionalitatiile specifice Git-ului, alaturi de o interfata vizuala intuitiva, ce simplifica intregul proces.

Printre avantajele utilizarii acestui soft enumeram:

- Client multi-platform, ce ofera aceeasi interfata pentru multiple sisteme de operare: Windows, macOS si Linux
- Vizualizarea arborelui sursa
- Suport si tool incorporat pentru rezolvarea conflictelor

2.2. Java

2.2.1. Istoric

Java reprezinta un limbaj de programare orientat pe obiect, creat de James Gosling, Patrick Naughton, Chris Warth, Ed Frank si Mike Sheridan de la compania Sun Microsystems, in anul 1991. Dezvoltarea primei versiuni functionale al acestui limbaj a durat 18 luni, si initial s-a numit Oak, dar a fost redenumit in Java in anul 1995.

Limbajul Java a aparut, precum predecesorii sai (C, C++, Assembly, etc), ca urmare a necesitatilor tehnologice determinate de evolutia calculatoarelor, retelelor, si mai ales, a internetului. Pentru a intelege mai bine, vom prezenta o scurta istorie a modului in care limbajele de programare au evoluat in a doua jumatate a secolului 20.

Fiecare inovatie in designul limbajelor de programare a fost motivata de nevoia de a rezolva probleme fundamentale pe care limbajele deja existente nu le puteau solutiona.

Aparitia limbajului C a fost determinata de necesitatea unui limbaj structurat, eficient, care sa inlocuiasca limbaje precum Assembly, Fortran sau Cobol. Totodata, complexitatea ridicata a programelor a dus la aparitia succesorului limbajului C, si anume C++, numit initial „C cu clase”.

Momentul de cotitura l-a reprezentat apritia World Wide Web. Terenul este acum pregatit pentru Java. [sietk] Internetul a catapultat Java in varful ierarhiei limbajelor de programare, iar limbajul Java, la randul lui, a avut un efect major asupra internetului.

2.2.2. Overview

Precum in orice alt limbaj de programare, elementele Java nu sunt izolate unele de celelalte, ci sunt utilizate impreuna. Programarea orientata pe obiect (OOP) sta la baza Java. Orice program scris in Java va fi orientat pe obiect. Dupa cum se precizeaza si Herbert Schildt in [sietk], exista 2 paradigme ce guverneaza modul in care este construit un program.

Prima dintre acestea se numeste *model orientat pe proces* (process-oriented model). Acest model reprezinta o succesiune de pasi si actiuni (cod) ce lucreaza asupra unor date. Limbajele procedurale precum C utilizeaza acest concept, insa complexitatea si numarul ridicat de linii de cod al programelor vor crea probleme in utilizarea acestui model.

A doua paradigma o reprezinta *programarea orientata pe obiect*. Acest concept presupune organizarea codului in jurul datelor, sau mai bine spus, *obiectelor*. Astfel, un program orientat pe obiect poate fi caracterizat ca *un mediu in care datele controleaza accesul la cod.*

2.2.3. Abstractizare

Un aspect esential al programelor orientate pe obiect il constituie abstractizarea. Pentru a gestiona complexitatea mare a programelor, se vor folosi tehnici de abstractizare.

Spre exemplu, cand o persoana conduce o masina, acea persoana nu concepe masina precum un set enorm de componente individuale. Ea este vazuta precum un singur obiect cu un comportament bine definit si o suita de caracteristici si functionalitati. Aceasta abstractizare face ca detaliile despre modul in care functioneaza motorul sau franele sa fie ignorate.

Clasificarea ierarhica reprezinta o tehnica de abstractizare. Aceasta presupune „desfacerea” intregului sistem in subsisteme. Revenind la exemplul masinii, putem vedea aceste subsisteme precum sistemul de franare, sistemul audio, sistemul de incalzire, etc. Acest ansamblu compune intregul obiect, iar intregul obiect este gestionat cu ajutorul elementelor ce il formeaza.

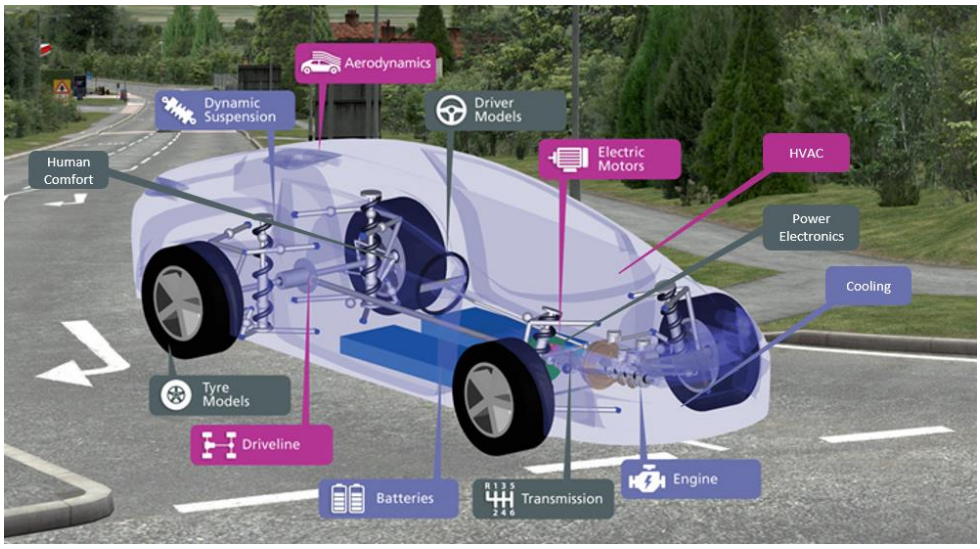

Figura 2.2.1. Ansamblu de subsisteme din structura unui autovehicul

Aceasta abstractizare poate fi aplicata, evident, si programelor scrise in limbaj orientat pe obiect. Un program complex poate fi structurat in multiple componente (obiecte), cu comportament unic si bine definit. [sietk] Ele pot fi tratate precum entitati care raspund la apeluri ce le spun *realizeaza aceasta actiune*. Aceasta este esenta programarii orientate pe obiect.

2.2.4. Principiile programare orientate pe obiect

Cele 3 principii ce guverneaza aceasta tehnica sunt: incapsularea, mostenirea si polimorfismul. In cele ce urmeaza, vom prezenta detalii despre modul in care aceste tehnici influenteaza modul in care lucram cu limbajul Java.

2.2.4.1. Incapsularea

Incapsularea reprezinta „invelisul” protector al codului. Cu alte cuvinte, incapsularea reprezinta un mecanism prin care datele si metodele ce gestioneaza aceste date sunt grupate intr-o unitate, ce le „ascunde” de mediul exterior. Aceasta unitate se numeste *clasa.* Aceasta prezinta structura si modul de functionare pe care il vor avea obiectele de tipul clasei. Obiectele se mai numesc *instante ale clasei*.

Prin „ascundere” ne referim la managementul accesului la membrii clasei. Accesul se stabileste utilizand cuvinte cheie numite *modificatori de acces* (*access modifiers)*. Acestia sunt de 4 tipuri:

- Private: accesul la un membru *privat* este permis doar in interiorul clasei.
- Public: accesul al un membru *public* este permis de oriunde, atat din interiorul clasei, cat si din afara ei.
- Protected: accesul la un membru *protected* este permis doar in interiorul package-ului in care se afla clasa, sau in alte package-uri, utilizand clasa derivata, prin *mostenire*, despre care vom vorbi in urmatorul subcapitol.
- Default: accesul la un membru *default* este permis doar in interiorul package-ului in care se afla clasa.

Modifier	Class	Package	Subclass	Global
Public	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	No
Default	Yes	Yes	No	No
Private	Yes	No	No	No

Figura 2.2.2. Modificatorii de acces din limbajul Java

2.2.4.2. Mostenirea

Mostenirea, dupa cum sugereaza si termenul, reprezinta proprietatea prin care un obiect preia caracteristicile „parintelui” sau. Prin parinte, ne referim la clasa din care aceasta deriva. Acest concept sustine ideea de clasificare ierarhica.

Vom considera un exemplu din lumea reala: un cimpanzeu. Acesta face parte din clasa primatelor, care, la randul ei, face parte din clasa mamiferelor. Acestea din urma fac parte din clasa animalelor. Se observa astfel ierarhia claselor, in care fiecare subclasa mosteneste toate atributele superclaselor.

Daca nu ar exista mostenire, atunci fiecare obiect ar trebui sa isi defineasca intregul set de proprietati, care sa ii ateste unicitatea. Utilizand conceptul de mostenire, un obiect trebuie sa isi defineasca doar acele caracteristici ce il fac diferit de celelalte din clasa lui. Astfel, mostenirea confera obiectului statusul de *instanta mai specifica* a clasei.

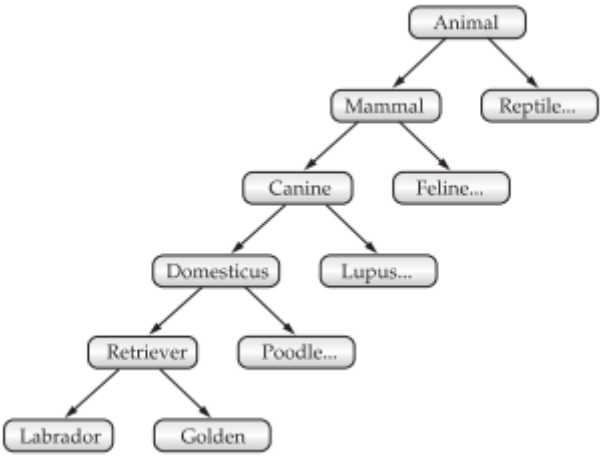

Figura 2.2.3. Lantul mostenirii in lumea animala

2.2.4.3. Polimorfism

Termenul de polimorfism isi are originile in limba greaca: *poly* insemnand „multe”, iar *morphe* insemnand „forme”. In lumea limbajelor de programare orientate pe obiect, polimorfismul reprezinta proprietatea de a avea multiple implementari ale aceleiasi interfete, fiecare din ele gestionand datele intr-un mod unic.

Putem vedea acest concept precum o interfata si un set de metode, care vor definii diferite sarcini, insa toate implementarile ale acestor metode, indiferent de modul de functionare si de rezultatul final, au aceeasi baza: ideea metodei. Vom exemplifica, din nou, cu o situatie din lumea animaliera: simtul olfactiv al unui caine. Cand acesta simte mirosul hranei, el va actiona mergand spre bolul cu mancare. In situatia in care simte mirosul unei pisici sau un alt caine, va alerga dupa ea/el si va latra. Asadar, conceptul de baza este simtul mirosului, dar in functie de *datele* pe care „lucreaza”, va avea un alt rezultat final.

[sietk] Cele 3 elementele fundamentale ale programarii orientate pe obiect incapsularea, mostenirea si polimorfismul lucreaza impreuna, pentru a crea un mediu care suporta o dezvoltare robusta si scalabila.

2.2.5. Spring Boot

Spring Boot reprezinta un framework al limbajului Java, util pentru dezvoltarea microserviciilor si aplicatilor Web. Acesta este dezvoltat pe baza framework-ului Spring, si vine cu multiple avantaje fata de predecesorul sau.

Framework-ul Spring punea la dispozitie Spring MVC pentru aplicatii web, insa configurarea acestuia reprezenta un pas dificil de trecut pentru programatorii nefamiliarizati cu acesta. Solutia a reprezentat-o Spring Boot. Acesta contine toate caracteristicile Spring, si faciliteaza dezvoltarea aplicatiilor, reducand din complexitatea configuratiilor. Astfel, programatorul se poate axa direct pe logica de functionare a aplicatiei, si nu pe procesul de configurare.

2.2.5.1. Proprietatiile Spring Boot

Spring Boot vine cu o lista de proprietati, din care enumeram:

- Permite configurarea proiectului in XML: mai facila decat in cazul Spring, totul fiind deja *auto-configured* (configurare realizata pe baza dependintelor de jar-uri adaugate in XML; utilizata prin adnotarile *@EnableAutoConfiguration* si *@SpringBootApplication*).
- Crearea unui REST API: simpla utilizare a adnotarii *@RestController* si a celor specifice *endpoint-urilor*, precum *@RequestMapping, @GetMapping*, etc, va simplifica acest proces.
- Include un server de Tomcat incorporat, pe care pot fi lansate pachete *JAR* (Java Archive) sau *WAR* (Web Application Resource)

2.2.5.2. Arhitectura Spring Boot

Arhitectura Spring Boot este structurata pe nivele (*layers*). Un nivel comunica cu cel aflat imediat sub sau deasupra lui, in ierarhie. Aceasta structura poate fi vizualizata in imaginea urmatoare:

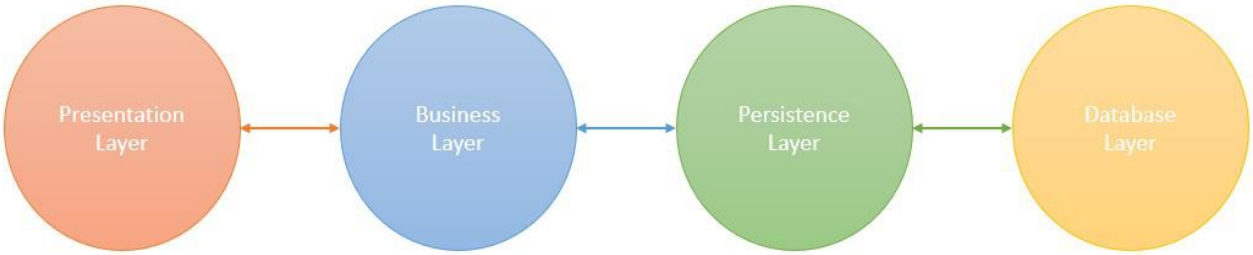

Figura 2.2.4. Nivelurile ierarhice ale arhitecturii Spring Boot

Nivelul *Prezentare* reprezinta nivelul din varful ierarhiei. Acesta are rolul de a gestiona request-urile de HTTP, impreuna cu datele (in format JSON) sosite de la client. Aceste date sunt convertite in obiecte specifice fiecarui proiect, care mai apoi sunt trimise nivelului *Business*.

Nivelul *Business* contine business logic-ul: serviciile care opereaza pe datele primite de la nivelul Prezentare. Nivelul *Persistence* prezinta logica de salvare a datelor bazei de date. Converteste date ce vor fi salvate in baza de date in formatul necesar, si vice-versa.

Nivelul *Database* contine baza de date si gestioneaza datele din aceasta. Se ocupa cu operatiile CRUD (Create, Read, Update si Delete).

Vom prezenta in continuare, sumar, flow-ul functionarii acestei arhitecturi. Imaginea urmatoare este, in acest sens, utila.

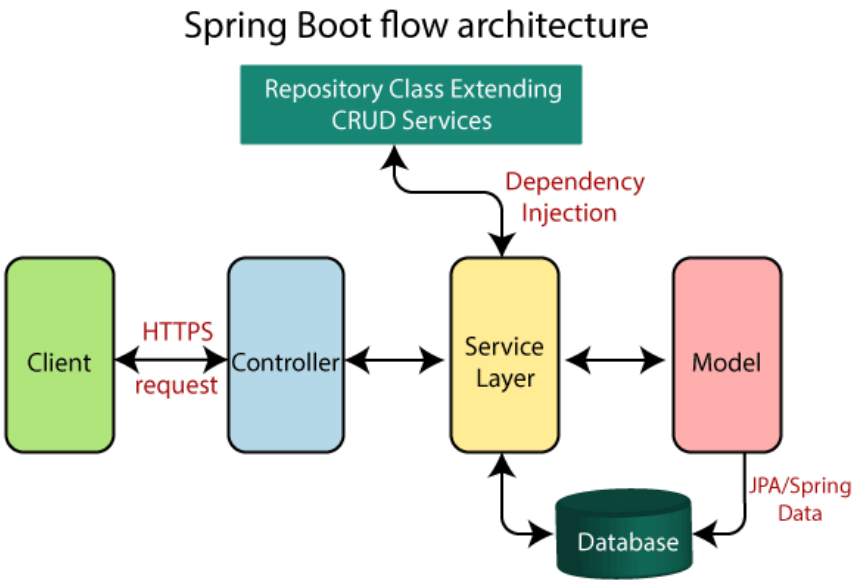

Figura 2.2.5. Flow-ul de functionare al arhitecturii Spring Boot

Clientul trimite request-uri catre Controller. Aceste request-uri pot fi de urmatoarele tipuri: GET, POST, PUT, PATCH, etc. Fiecare dintre acestea are un anumit rol. Spre exemplu, GET este utilizat pentru a obtine o anumita data de la server, in timp de PATCH este utilizat cand se doreste modificarea unei date din baza de date.

Controller-ul mapeaza request-ul la un endpoint, si apeleaza metodele din service pentru a indeplinii sarcina specifica acelui request.

Service-ul realizeaza business logic-ul si actioneaza asupra bazei de date. Datele prelucrate sunt intoarse (in anumite cazuri) la client.

2.3. Javascript

2.3.1. Istoric

JavaScript este un limbaj de scripting dezvoltat de catre Brendan Eich de la compania Netscape, in 1995. Scopul acestuia a fost sa creeze un limbaj care se plia browser-ului web dezvoltat de catre compania Netscape: *Netscape Navigator*.

Dupa cum spune insusi creatorul JavaScript, Brendan Eich, in [jsBible], „javascript s-a nascut din nevoia de a permite dezvoltatorilor de HTML sa scrie script-uri direct in documentele lor”. La mijlocul anilor `90, dezvoltarea aplicatiilor Web intra intr-o noua era. Traditionalul HTML urma sa fie acompaniat de nou-aparut-ul Java Applet, vazut drept [jsBible] adevaratul mod de a naviga pe paginile Web. Astfel, javascript urmarea sa fie utilizat pentru a crea scripturi pentru Java Applets.

Desi sintaxa javascript se bazeaza, partial, pe cea a limbajului Java, fapt ce ar fi incurajat dezvoltatorii sa il utilizeze, acesta nu respecta ideea fundamentala a unui script: sintaxa simpla, putin cod si functional. Nu se dorea declararea tipurilor obiectelor, necesitatea de a pune punct punct-si-virgula dupa fiecare linie de cod si alte elemente de sintaxa ce ar fi incetinit scrierea codului.

2.3.2. Overview

JavaScript este un limbaj de scripting, utilizat pentru a crea pagini web, al caror content este dinamic, permitand utilizatorilor sa interactioneze cu acesta, nemaifiind necesara reincarcarea paginii pentru a vedea noua stare a content-ului.

Unul dintre aspectele esentiale ale acestui limbaj, care in unele cazuri poate vi vazut drept un dezavantaj, il reprezinta libertatea si flexibilitatea pe care o confera programatorului, precum lipsa tipurilor de date clasice: in limbajul Java, o variabila poate fi declarata de tipul *int*, *string*, *boolean*, dar in JavaScript, aceste tipuri sunt inlocuite de keyword-ul *let* sau *var*. Aceasta flexibilitate trebuie gestionata atent de catre programator.

JavaScript poate fi utilizat atat pe partea de client, cat si pe partea de server. Client-side JavaScript vine cu cateva trasaturi specifice limbajelor de programare, prin care poate realiza sarcini precum:

- Stocarea valorilor utile, din cadrul paginii aplicatiei Web, in *variabile*.
- Rularea unor functii, ca raspuns la diferite evenimente ce apar in pagina. Spre exemplu, in momentul in care un utilizator da click pe un button, se va apela o functie care schimba culoarea background-ului (cunoscuta functionalitate de Dark Mode).
- Realizarea legaturii cu server-ul, pentru preluarea datelor de la acesta si gestionarea acestora in pagina.

O alta trasatura importanta o reprezinta API-urile (Application Programming Interface). Un API este set de protocoale si tool-uri ce permit comunicarea intre diferite aplicatii software. In cazul unei aplicatii Web, API-urile au rolul de a stabili comunicatia si modul in care datele sunt transferate intre client si server. In figura de mai jos se prezinta schematic legatura dintre cele 2 elemente: cu ajutorul Fetch API pe partea de client si Rest API pe partea de server.

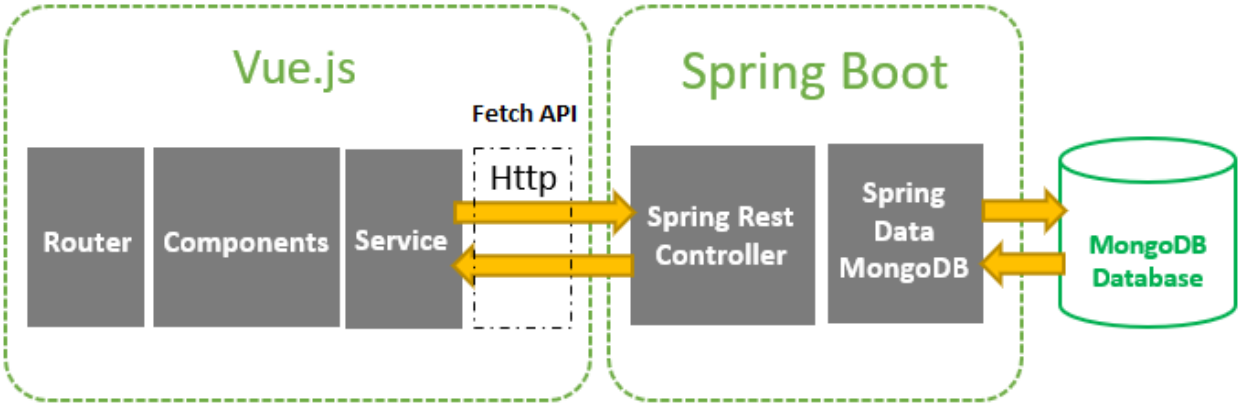

Figura 2.2.6. Utilizarea API-urilor in lantul comunicarii dintre Client (Vue.js) si Server (Spring Boot).

Pentru dezvoltarea unei aplicatii Web pot fi utilizate o multitudine de API-uri. Aceste se incadreaza in mai multe categorii, din care enumeram:

- Service API: sunt cele mai comune, rolul lor fiind de a realiza transferul de date intre aplicatii. Utilizeaza protocoale HTTP si date in format JSON sau XML. Un exemplu de un astfel de API este RESTful API.

- Authentication API: au rolul de a asigura autentificarea utilizatorilor intr-o aplicatie Web. Ofera dezvoltatorului functii si tool-uri pentru realizarea functionalitatiilor de register si login. Exemple de acest gen sunt OpenID Connect si OAuth.

- Payment API: rol in efectuarea sigura a tranzactiilor online. Un exemplu de acest gen este PayPal Rest API.

- Social Media API: util pentru a integra diferite platforme de socializare intr-o aplicatie Web, prin adaugarea de functionalitati specifice acestora: postarea de articole, vizualizarea profilului unui alt utilizator, etc. Un astfel de API il reprezinta Twitter API.

Astfel, se observa impactul major pe care il au aceste interfete. Elimina necesitatea proiectarii si dezvoltarii de la zero al unui set urias de functionalitati, fiind eficiente din punct de vedere al timpului si costului dezvoltarii. Printre alte avantaje se numara modularitatea, reutilizabilitatea si flexibilitatea.

Nu putem vorbi despre JavaScript si despre API-uri fara sa amintim si despre DOM (Document Object Model). DOM-ul este, la randul lui, un Web API. Acesta reprezinta legatura dintre paginile web si script-urile ce actioneaza pe aceste pagini. Prin intermediul lui se controleaza elementele din pagina.

DOM-ul poate vi vazut precum un arbore cu noduri, in care fiecare nod reprezinta un obiect. Un obiect poate fi un element de HTML, CSS, cu alte cuvinte contentul paginii. Script-urile scrise in limbajul JavaScript actioneaza asupra nodurilor acestui arbore. Astfel are loc modificarea dinamica a acestuia. Dezvoltatorii de aplicatii Web utilizeaza JavaScript pentru a:

- Accesa elemente, utilizand metode ale DOM-ului precum: *getElementById* sau *getElementByClassName* (unui element de HTML i se poate atribui un id sau o clasa)

- Adauga sau elimina dinamic elemente: spre exemplu, afisarea unui mesaj de eroare in cazul in care un field dintr-un form nu a fost completat.

- Modifica contentul: continuand exemplul anterior, se poate realiza activarea unui buton daca toate field-urile au fost completate.

- Gestiona evenimente: unui buton i se poate atribui un eveniment precum cel de *click*, care poate fi „prins” de catre JavaScript si mapat la o functie ce va fi apelata.

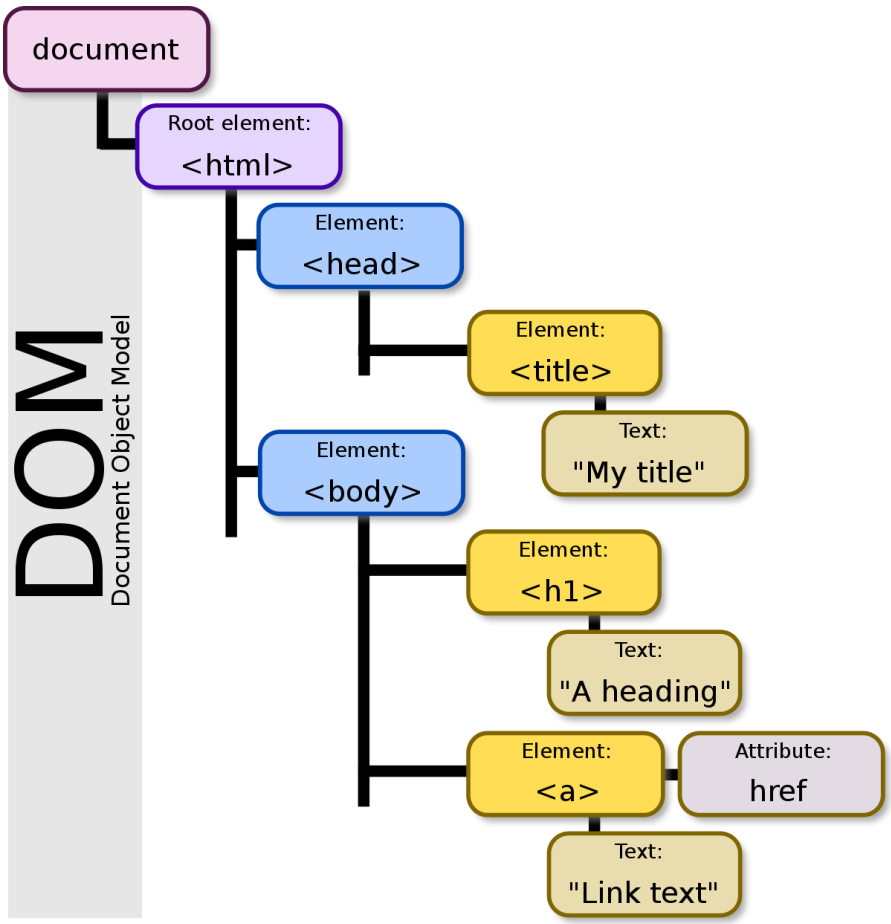

Figura 2.2.7. Reprezentarea arborescenta a DOM-ului

Ca ultim aspect, vom prezenta cateva detalii despre o alta trasatura esentiala a limbajului JavaScript: versatilitatea. JavaScript poate fi utilizat front-end, back-end, este compatibil cu majoritatea browser-elor actuale, utilizeaza o multitudine de API-uri, etc. Pe langa cele enumerate anterior, amintim:

- Utilizarea Cross-platform: pe langa aplicatiile Web, acest limbaj poate fi utilizat si in dezvoltarea aplicatiilor Mobile (cu ajutorul React Native sau Mobile Angular UI).

- Prezinta un ecosistem bogat in framework-uri precum React, Vue sau Angular, si librarii precum jQuery, dar si multe plug-in-uri. Acestea faciliteaza procesul de dezvoltare, si ajuta la crearea unor aplicatii Web complexe.

- Se incearca migrarea acestuia si in domeniul IoT, prin framework-uri precum Node.js sau IoT.js .

In concluzie, limbajul JavaScript prezinta o lista lunga de caracterstici ce il transforma intr-unul dintre cele mai, daca nu cel mai util limbaj din industria IT la ora actuala.

2.4. Vue.js

Amintit in capitolul anterior, Vue reprezinta un framework al limbajului JavaScript, utilizat pentru a crea interfete pentru utilizatori. Ofera tool-uri pentru dezvoltarea frontend-ului aplicatiei. Se bazeaza pe traditionalele HTML, CSS si JavaScript, si pe un model de programare bazat pe componente si reactivitate.

2.4.1. Componenta

Conceptul de componenta descrie o unitate independenta si reutilizabila ce contine design-ul si logica de randare a unei parti a interfetei utilizatorului. Putem vedea aceasta precum chenarul in care este prezentat un produs dintr-o lista de produse de pe pagina unui magazin online. Acest model este utilizat pentru diferite seturi de date, dar prezinta acelasi design si functionalitate.

Asadar, o componenta este structurata in 3 elemente:

1. Script: contine codul de JavaScript ce defineste comportamentul si componentei. Aici se gestioneaza datele, se afla metodele care actioneaza asupra acestor date si asupra elementelor din DOM, cat si elemente caracteristice Vue-ului: proprietati *computed* si reactive.

2. Template: reprezinta structura vizuala a componentei, si anume elementele DOM-ului (HTML, CSS), dar si elemente din sintaxa Vue (directive), ce ajuta la randarea conditionata sau capturarea evenimentelor.

3. Styles: sectiune in care se definesc atributele CSS specifice claselor sau id-urilor setate elementelor de HTML din sectiunea Template.

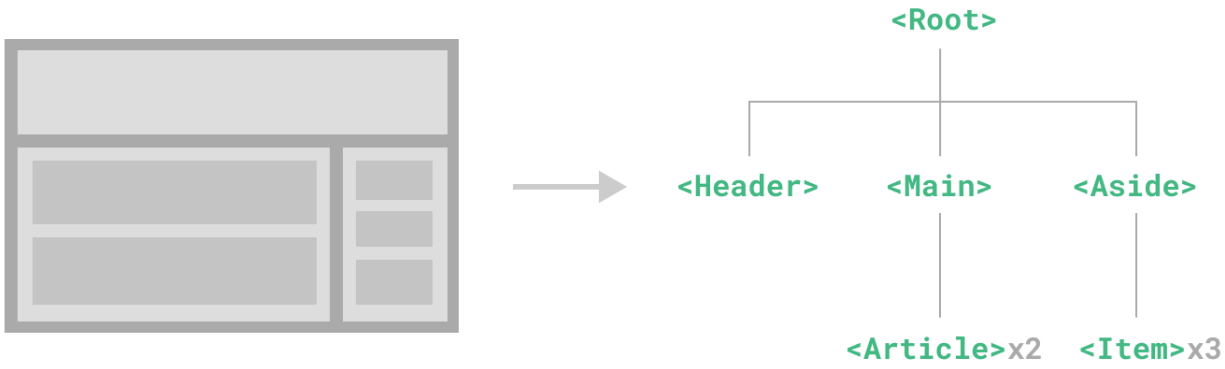

Figura 2.4.1. Structura unei pagini in care se observa reutilizabilitatea componentelor *Article* si *Item*

O componenta poate fi inclusa intr-o alta componenta, prin utilizarea keyword-ului *import*, astfel:

```
<script setup>
import ComponentaTemp from './ComponentaTemp.vue'
</script>

<template>
  <h1>Aceasta este o componenta!</h1>
  <ComponentaTemp />
</template>
```

2.4.2. Reactivitate

Reactivitatea reprezinta un concept fundamental in Vue. Acesta permite legatura automata intre datele din aplicatie si interfata utilizatorului. Cu ajutorul lui, Vue detecteaza schimbarile in timp real ale datelor si modifica starea elementelor de UI ce sunt legate de acestea.

Pentru a defini o variabila ca fiind reactiva putem utiliza unul dintre urmatoarele keyword-uri:

- *ref():* primeste ca parametru o valoare initiala, si returneaza un obiect reactiv, mutabil. Pentru a ii accesa valoarea acestuia, se utilizeaza atributul *.value* al obiectului.

- *reactive():* util cand se lucreaza cu obiecte cu multiple atribute, in care se doreste urmarirea modificarilor oricarui atribut. Primeste ca parametru un obiect, si returneaza un proxy al acestuia. Se poate accesa direct orice atribut al obiectului reactiv, fara utilizeaza keyword-ului *.value*, precum la *ref.*

Diferenta dintre cele 2 consta in faptul ca *ref* se utilieaza pentru a atribui starea reactiva unor date primitive, in timp ce *reactive* realizeaza aceeasi sarcina, dar pentru obiecte.

2.4.3. View-uri, proprietati si evenimente

Am vazut ca o aplicatie Web este formata din componente, iar componentele isi pot schimba starea si reactiona la modificari ale datelor, prin reactivitate. Dar cum pot comunica multiple componente intre ele si reactiona unele la schimbarile altora?

In primul rand, trebuie sa definim termenul de View. View-ul reprezinta pagina Web pe care utilizatorul poate naviga. Aceasta are elementele ei proprii de UI, dar utilizeaza, in cele mai multe cazuri, o suita de componente ce imbogatesc intregul UI. Din nou ne intrebam, cum vor comunica aceste componente intre ele?

Componentele sunt gandite pentru a fi reutilizate. Poate exista scenariul in care o componenta este utilizata mereu in acelasi mod. Spre exemplu, un buton personalizat de catre dezvoltator, dar care va avea aceeasi vizualizare si functionalitate in toate View-urile in care este utilizat. Totusi, pot exista cazuri in care se doreste ca fiecare componenta (de acelasi tip), sa aiba caractersticile proprii, care sa o diferentieze de celelalte componente. In acest sens, ne putem imagina o lista de produse pe o pagina Web a unui magazin online. Fiecare produs este prezentat printr-o componenta, printr-o imagine, cateva detalii legate de acesta, un buton de redirectionare catre pagina acestuia, etc. Asadar, componenta trebuie sa cunoasca aceste informatii. Aceste informatii se numesc *proprietati* (*props*), si sunt declarate in zona de script a componentei. View-ul (sau o alta componenta) ce utilizeaza componenta are rolul de a ii asigna aceste proprietati. Pentru a intelege mai bine, utilizam exemplul urmator:

Componenta *Produs*:

```
<script setup>
const props = defineProps([“nume”, “descriere”, “pret”])

</script>
```

View-ul *PaginaProduse:*

```
<script setup>
import Produs from './Produs.vue'

let produs1 = {
      “nume”: “Tastatura”,
      “descriere”: “Tastatura mecanica”,
      “pret”: “270”
}

let produs2 = {
      “nume”: “Mouse”,
      “descriere”: “Mouse optic”,
      “pret”: “30”
}

let produs3 = {
      “nume”: “Imprimanta”,
      “descriere”: “Imprimanta color”,
      “pret”: “550”
}

</script>

<template>
      <Produs :nume =”produs1.nume” :descriere:”produs1.descriere”
:pret=”produs1.pret”>
      <Produs :nume =”produs2.nume” :descriere:”produs2.descriere”
:pret=”produs2.pret”>
      <Produs :nume =”produs3.nume” :descriere:”produs3.descriere”
:pret=”produs3.pret”>
</template>
```

Utilizand in zona de script a componentei macro-ul *definePropsProps([...])* putem definii o lista de proprietati pe care componenta le va avea. In acelasi timp, in View-ul (sau componenta) parinte ne definim, in zona Template, unde utilizam componenta, valorile pe care le vor avea aceste proprietati. In componenta, variabila *props* este reactiva. Asadar, elementele pot fi accesate cu ajutorul atributului *.value*.

In exemplul de mai sus se poate o repetitie deranjanta a declararii componentei *Produs*. Solutia o reprezinta directiva *v-for*, care va fi prezentata in captiolul urmator.

Am vazut acum metoda de personalizare a unei componente. Dar cum va anunta componenta pe parinte, sau chiar pe celelalte componente, daca are loc o schimbare in datele cu care lucreaza aceasta? Modalitatea de comunicare intre componente si view-uri o reprezinta *evenimentele* (*emits*).

Precum *props*, intr-o componenta ne putem defini o lista de *emits*. Prin *emits*, componenta trimite un „mesaj”, ce poate fi un simplu trigger, sau chiar un mesaj ce contine si date. Se utilizeaza macro-ul *defineEmits([..])* pentru a defini aceasta lista. Urmatorul exemplu prezinta un mod de utilizare a evenimentelor. Ne vom referii la exemplul anterior.

Componenta *Produs:*

```
<script setup>
const props = defineProps([“nume”, “descriere”, “pret”])
const emit = defineEmits(['adaugaInCos'])

function clickAdauga() {
  emit(“adaugaInCos”)
}
</script>


<template>
      <button type="button" @click=”clickAdauga()”>Adauga</button>
</template>
```

View-ul *PaginaProduse:*

```
<script setup>
import Produs from './Produs.vue'
import { ref } from 'vue'

const cos = ref([]);

let produs1 = {
      “nume”: “Tastatura”,
      “descriere”: “Tastatura mecanica”,
      “pret”: “270”
}

function functieAdaugaInCos() {
      cos.value.push(produs1);
}

</script>

<template>
      <Produs :nume =”produs1.nume” :descriere:”produs1.descriere”
:pret=”produs1.pret” @adauga-in-cos=”functieAdaugaInCos()”>
</template>
```

In acest exemplu se observa modul in care componenta *Produs* ii comunica View-ului *PaginaProduse* sa adauge in cos produsul selectat. Keyword-ul *@adauga-in-cos*, de tip *@nume-eveniment* este utilizat in acest sens, si apeleaza functia *functieAdaugaInCos()*. In exemplu nu este realizata o comunicare intre 2 componente, dar pentru a realiza asta, o metoda ar fi fost ca, la captarea evenimentului de la componenta *A*, parintele (View-ul) sa ii modifice *props*-urile componentei *B*.

2.4.4. Directive

Directivele sunt tool-uri ce sunt utilizate pentru atribui elementelor de UI diferite functionalitati si comportamente. Le vom enumera pe cele mai comune dintre acestea:

- *v-if* / *v-else-if* / *v-else*: directiva cu caracter conditional. Utilizata pentru a modifica atributul de vizibilitate al unui element de html sau chiar a unei componente.
- *v-for*: util pentru reprezentarea unei liste formate din acelasi element vizual (componenta). Utilizeaza un array.
- *v-on*: util pentru captarea evenimentelor (echivalentul lui @ din exemplele anterioare).
- *v-model*: util pentru realizarea unei legaturi intre un element specific *form-urilor* si un element de html sau o componenta.

Vom prezenta acum acelasi exemplu de mai devreme cu cele 3 produse, dar modificat, astfel incat sa utilizam aceste directive, pentru a reduce din redundanta codului.

```
<script setup>
import Produs from './Produs.vue'
import { ref } from 'vue'

const cos = ref([]);

let produs1 = {
      “nume”: “Tastatura”,
      “descriere”: “Tastatura mecanica”,
      “pret”: “270”
}

let produs2 = {
      “nume”: “Mouse”,
      “descriere”: “Mouse optic”,
      “pret”: “30”
}

let produs3 = {
      “nume”: “Imprimanta”,
      “descriere”: “Imprimanta color”,
      “pret”: “550”
}

let produse = [produs1, produs2, produs3];

function functieAdaugaInCos(index) {
      cos.value.push(produse[index]);
}


</script>

<template>
      <div v-for=”(produs, index) in produse”>
            <Produs v-if=”produs.pret < 300“
            :nume=”produs.nume”
            :descriere:”produs.descriere”
            :pret=”produs.pret”
            @adauga-in-cos=”functieAdaugaInCos(index)”>
      </div>
</template>
```

In acest exemplu, am utilizat directia *v-for* pentru a afisa o lista de produse prin componenta *Produs*, careia i-am setat proprietatiile, utilizand atributele elementului curent din lista. Totodata, am afisat doar produsele care au pretul mai mic decat 300. Pentru a realiza asta, am utilizat directiva *v-if*. Variabila *index* reprezinta indexul elementului din lista, si este utilizata impreuna cu functia *functieAdaugaInCos(index)* pentru a adauga in array-ul *cos* elementul de la indexul *index* din array-ul *produse*.

In concluzie, se pot observa facilitatiile introduse de catre Vue. Dezvoltatorul are la dispozitie un set larg de tool-uri cu care poate lucra, pentru a crea o aplicatie Web dinamica si complexa.

2.5. Pinia

In exemplele prezentate anterior nu sunt utilizate cantitati mari de date: un simplu array de obiecte, hardcoded. Pot exista insa situatii in care o pagina, sau chiar aplicatia cu totul, functioneaza pe baza unui set mare de date, supuse modificarilor in timp real. Este necesara o modalitate de a „stoca" aceste date intr-un „magazin" (store), de unde pot fi accesate de orice pagina, componenta sau functie are nevoie de ele. Acest concept de store este implementat de catre Pinia.

Pinia reprezinta o librarie a framework-ului Vue, ce realizeaza sarcinile prezentate in paragraful precedent: permite inmagazinarea datelor si share-uirea acestora intre componente si view-uri. Pinia a fost dezvoltat ca alternativa la o alta librarie cu caracter foarte similar: Vuex. Ambele au acelasi rol de state management, dar prezinta cateva diferente:

- Pinia a fost dezvoltat specific pentru versiunea Vue 3, si mai ales pentru API-ul Composition al acestuia, utilizand natura reactiva a acestuia, in timp ce Vuex a fost bazat pe Vue 2.
- Pinia foloseste reactivitatea data de hook-urile „ref", „reactive" si „computed", in timp ce Vuex se bazeaza pe modelul getter/setter pentru a aplica reactivitatea.
- Pinia permite crearea de store-uri multiple, in timp ce Vuex impune crearea de *module* pentru a realiza acest task.

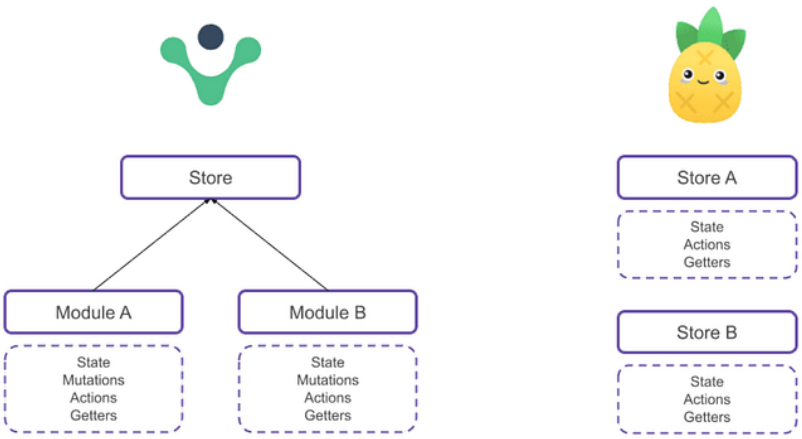

Figura 2.5.1. Diferente de concept intre Vuex (stanga) si Pinia (dreapta).

Vom prezenta in continuare un exemplu bazat pe cel anterior cu produsele.

Store-ul *Magazin:*

```
import { defineStore } from 'pinia'

export const useMagazinStore = defineStore(‘magazin’, {
  state: () => ({
        cos: [],

        produse: []
  }),

  actions: {
    adaugaProdusInCos(produs) {
      this.cos.push(produs);
    },
    adaugaProdusInListaDeProduse(produs) {
      this.produse.push(produs);
    },
    initializareListaProduse() {

        let produs1 = {
              “nume”: “Tastatura”,
              “descriere”: “Tastatura mecanica”,
              “pret”: “270”
        }

        let produs2 = {
              “nume”: “Mouse”,
              “descriere”: “Mouse optic”,
              “pret”: “30”
        }

        let produs3 = {
              “nume”: “Imprimanta”,
              “descriere”: “Imprimanta color”,
              “pret”: “550”
        }


        this.produse.push(produs1);
        this.produse.push(produs2);
        this.produse.push(produs3);
    },


  },
})
```

View-ul *PaginaProduse*:

```
<script setup>
import Produs from './Produs.vue'
import { useMagazinStore } from './MagazinStore.js'

let magazin = useMagazinStore();

magazin.initializareListaProduse();

</script>

<template>
      <div v-for="(produs, index) in magazin.produse">
            <Produs v-if="produs.pret < 300"
            :nume="produs.nume"
            :descriere:"produs.descriere"
            :pret="produs.pret"
            @adauga-in-cos="magazin.functieAdaugaInCos(produs)">
      </div>
</template>
```

Logica de gestionare a datelor a fost mutata din interiorul View-ului, in store-ul *Magazin*. Astfel se observa inca un avantaj al utilizarii acestui concept: clean code. Bineinteles, acest exemplu nu este unul complex, ci unul cu scopul de a prezenta ideea din spatele Pinia.

Definirea unui store se realizeaza cu ajutorul functiei *defineStore(...)*. Structura unui store este urmatoarea:

- State: zona ce contine datele cu care lucreaza store-ul (cu alte cuvinte, starea acestuia).
- Actions: functiile prin care este gestionat store-ul. Acestea sunt utilizate pentru a schimba valorile variabilelor declarate in state.

Accesarea store-ului in view-uri sau in componente se realizeaza prin apelul functiei *use...Store()*, care returneaza instanta store-ului. Acest obiect este de tipul *reactive*, asadar, nu se va utiliza *.value* pentru a se accesa atributele acestuia.

Concluzionand, conceptul de store simplifica logica de gestionare a datelor cu care lucreaza aplicatia, avand, totodata, proprietatea de reactivitate. Am vazut ca putem utiliza un sistem de management al datelor. Urmatorul pas ar fi utilizarea unui sistem de management al starilor aplicatiei. Capitolul urmator va pune accent pe aceasta idee.

2.6. Xstate

In orice aplicatie Web exista un flow, reprezentand starile si actiunile ce determina tranzitiile dintre aceste stari. In multe cazuri, acest flow poate fi vazut precum un FSM (finite state machine), sau *aparat cu stari finite.* Un aparat cu stari finite este caracterizat, dupa cum ii spune si numele, dintr-o lista finita de stari prin care poate trece un proces. Se doreste o modalitate de a gestiona o astfel de situatie, in cazul unei aplicatii Web, iar solutia este oferita de catre Xstate.

Xstate reprezinta o librarie ce ofera tool-uri pentru crearea si gestionarea starilor unei aplicatii. De multe ori, aplicatiile devin complexe, bazate pe un element critic: o modificare in starea unei variabile determina o schimbare in starea unei componente/pagini. Spre exemplu, apasarea unui buton va incrementa un contor, iar daca acel contor are o anumita valoare, se va face o tranzitie intre 2 componente. O astfel de situatie se rezolva, intr-un mod *not clean,* printr-o variabila booleana de genul *isCounterSet* sau *isComponentVisible*. O astfel de abordare reprezinta un exemplu bun de *bad practice*. Xstate doreste sa solutioneze astfel de probleme.

2.6.1. Machine

Elementul de baza al Xstate este, deloc surprinzator, o *masina* (*machine*). Aceasta contine intregul set de stari posibile si toate datele si actiunile prin care se face tranzitia intre aceste stari.

Structura unei masini este urmatoarea:

1. id: identificatorul masinii.
2. initial: starea initiala a masinii (starile sunt definite in sectiunea „states” a masinii).
3. context: datele cu care lucreaza masina (concept similar cu cel din store).
4. states: setul de stari al masinii
5. actions: functii care pot fi apelate fie la intrarea intr-o stare, sau la iesirea dintr-o stare

Pentru a realiza tranzitiile dintre stari, se utilizeaza proprietatea *on*. In exemplul urmator ne vom definii o masina, utilizand functia *createMachine()*, si vom declara campurile amintite anterior.

```
import {               } from 'xstate';

const                = createMachine({
    : 'exemplu',
        : 'stare1',
       : {
         : {
            : ['intrareStare1'],
           : ['iesireStare1'],

       : {
       goToStare2: 'stare2',

       goToStare3: {
               : 'stare3'
       }
     }
   },
         : {
            : ['intrareStare2'],
         : {
         goToStare1: 'stare1',
       }
   },
         : {
          : ['intrareStare3'],
         : {
         goToStare1: 'stare1',
       }
   }
  }
},
{
          : {
     intrareStare1: (       ,      ) => {
               .log('Am intrat in starea 1!');
     },
     iesireStare1: (       ,      ) => {
               .log('Am iesit din starea 1!');
     },
     intrareStare2: (       ,      ) => {
               .log('Am intrat in starea 2!');
     },
     intrareStare3: (       ,      ) => {
               .log('Am intrat in starea 3!');
     }
   }
  }

);
```

Trigger-ele *goToStare1, goToStare2* si *goToStare3*, impreuna cu starile la care pointeaza, realizeaza tranzitiile. Keyword-urile *entry* si *exit* au rolul de a stabili *actiuniile* ce se apeleaza in momentul intrarii, respectiv iesirii din starea curenta.

2.7. MongoDB

MongoDB reprezinta un sistem de management de tip document-oriented al bazelor de date. Spre deosebire de bine-cunoscutele SQL si MySQL, MongoDB este un sistem de tip NoSQL. Primele sisteme de acest tip au aparut la inceputul secolului, si aveau ca focus scalabilitatea, query-uri rapide si doresc sa faciliteze lucrul cu bazele de date in programare.

Sistemele SQL si cele NoSQL prezinta multe diferente, din care enumeram:

- Modelul de stocare al datelor la SQL este bazat pe tabele si inregistrari, in timp ce un sistem NoSQL este construit pe colectii si documente.
- SQL impune o organizare fixa, in timp ce NoSQL prezinta flexibilitate.
- SQL organizeaza datele relational, spre deosebire de NoSQL care este non-relational.

Figura 2.7.1. Modelul de stocare al datelor utilizat in SQL, respectiv NoSQL

Colectiile reprezinta grupari de documente. Un document poate fi vazut precum o lista de atribute de tip *cheie-valoare*, similar structurii JSON (JavaScript Object Notation), fiind chiar o varianta a acesteia: BSON (Binary JSON). Aceasta notatie permite utilizarea mai multor tipuri de date, inclusiv array-uri sau documente, sau chiar array-uri de documente.

Cateva dintre avantajele utilizarii MongoDB:

- Modalitatea de stocare: MongoDB permite stocarea datelor in structuri largi si variate, fiind scalabil atat vertical, cat si orizontal.

- Structuri de date complexe: dupa cum am amintit anterior, este posibila stocarea imbricata (*nested*) a documentelor.

- Integrarea facila in diverse medii de programare: MongoDB vine cu librarii si tool-uri prin care poate fi utilizat eficient in limbaje precum Java sau JavaScript.

Figura 2.7.2. Exemplu de document stocat in MongoDB

2.8. Mediul de dezvoltare

Pentru dezvoltarea oricarei aplicatie software, este necesara utilizarea unui mediul de dezvoltare, cunoscut in engleza sub numele de Integrated Development Environment (IDE).

2.8.1. Visual Studio Code

Visual Studio Code (VSC) este un editor de cod puternic si eficient ce poate fi descarcat si instalat pe desktop. Este un tool gratuit, open-source proiectat pentru a oferi dezvoltatorilor un mediu de programare simplu, dar puternic customizabil.

VSC este unul dintre cele mai utilizate IDE-uri de catre programatori, fapt determinat de avantajele pe care le prezinta, din care enumeram:

- Ofera suport integrat pentru multiple limbaje de programare (JavaScript, Python, C++, etc)

- Permite instalarea de extensii ce imbogatesc mediul. Extensiile pot reprezenta limbaje de programare, tool-uri pentru debug, teme vizuale si multe altele.

- Permite gestionarea repo-ului din git direct din aplicatie: vizualizarea diferentelor fata de ultimul snapshot, realizarea de commit-uri, push/pull, etc.

- Vine cu posibilitatea realizarii debugging-ului.

- IntelliSense: tool pentru completarea automata a codului, in functie de context, in timpul tastarii caracterlor in cod.

- Conectarea la medii de containerizare precum docker.

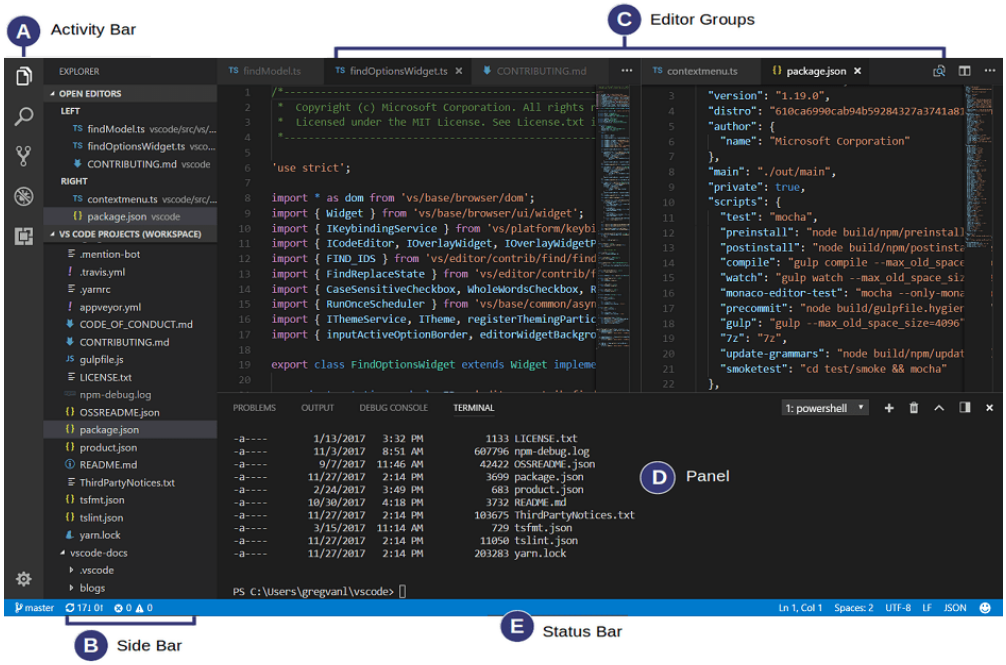

Figura 2.8.1. Interfata vizuala a Visual Studio Code si structura acesteia

2.8.2. Eclipse

Eclipse reprezinta un IDE utilizat de programatori, ce ofera posibilitatea de scriere a codului, debugging-ul acestuia si deployment-ul aplicatiei. Ofera un set de tool-uri, librarii si framework-uri, si suport pentru limbaje de programare precum Java, C++, si altele.

Precum alte medii de dezvoltare, Eclipse prezinta o lista larga de caracteristici, precum:

- Editor de cod: elementul principal al acestuia, insotit de feature-uri ca *syntax highlighting*, autocompletor de cod, formator de cod, si altele, utile pentru cresterea productivitatii si eficientizarea procesului de programare.

- Tool-uri pentru debugging: breakpoints, parcurgerea codului step-by-step, vizualizarea starii variabilelor la diferite momente de timp, etc.

- Integrarea unui sistem de versionare: utilizarea functionalitatiilor Git-ului direct din aplicatie: commit-uri, rezolvare de conflicte si gestionarea branch-urilor sunt cateva dintre acestea.

Acestea, cat si altele, il transforma intr-un IDE preferat de multi dezvoltatori de aplicatii.

2.9. Wireframe

Dezvoltarea oricarui proiect presupune existenta unei faze de proiectare. In cazul unei aplicatii Web, in aceasta faza se defineste, printre altele, design-ul si structura interfetei vizuale. Dintre caracteristicile acesteia, amintim:

- Paginile care vor constitui aplicatia
- Elementele vizuale ce vor compune paginile si caracteristicile acestora: detalii legate de dimensiune, aseazarea in pagina, forma acestora, etc.

Pentru a putea realiza acest pas se utilizeaza conceptul de *Wireframe*. Prin wireframe se intelege o schema sau un plan pe care design-erii si programatorii il folosesc pentru a definii structura aplicatiei pe care o vor dezvolta.

Un wireframe reprezinta schita aplicatiei, in care elementele vizuale sunt reprezentate intr-un format simplu, de tipul *low-fidelity*. Acesta este o caracteristica esentiala a conceptului de wireframe si este utilizat din cateva motive:

1. Schita nu reprezinta design-ul final: accentul se pune pe structura paginii, si nu pe detalii, intru-cat nu sunt utilizate culori, forme sofisticate si alte caracteristici bine definite.
2. Schita reprezinta o forma de comunicare, supusa schimbarilor.

Totodata, schita se realizeaza inaintea scrierii codului, astfel reducand necesitatea schimbarilor multiple in cod, determinate de posibile schimbari ale structurii aplicatiei.

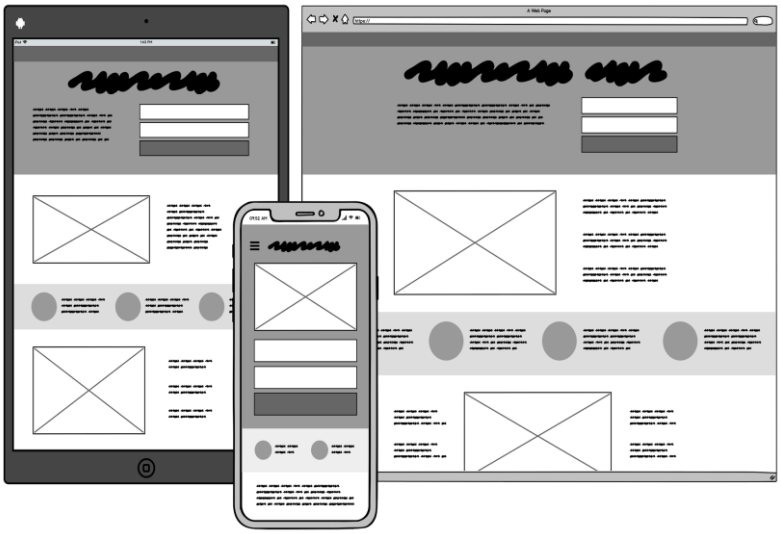

Figura 2.9. Exemplu de wireframe

2.10. Mockup

Pasul precedent a ajutat la crearea structurii interfetei vizuale a aplicatiei: o schema simpla, fara detalii bine definite, fara elemente cromatice. O data ce aceasta etapa este finalizata, e timpul sa fie adaugate aceste detalii, rezultand intr-un *Mockup.*

Un mockup este o reprezentare statica a interfetei aplicatiei, in care sunt utilizate multe dintre elementele de design in versiunea lor finala. Cu alte cuvinte, un mockup poate fi vazut precum o reprezentare artistica a aplicatiei. Este important de precizat faptul ca mockup-ul este o reprezentare non-functionala. Flow-ul interactiunii utilizatorului cu aplicatia nu este regasit intr-o prezentare de tip mockup.

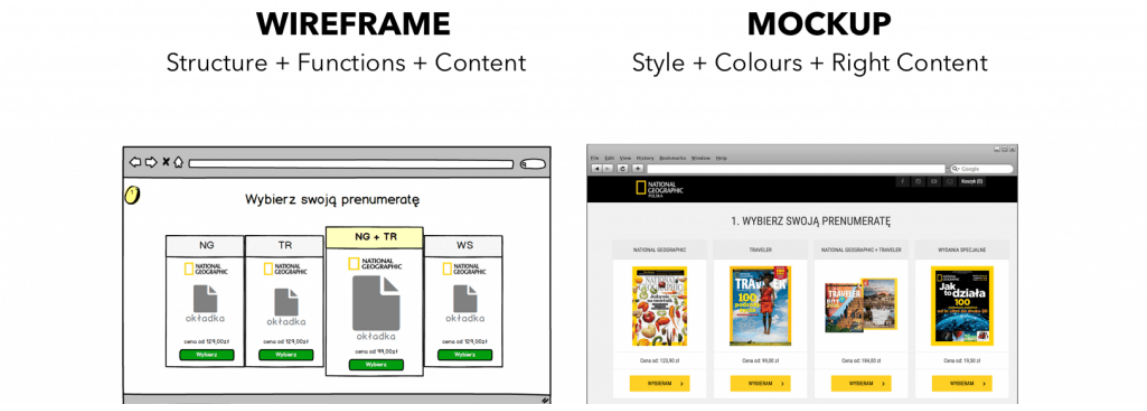

Figura 2.10. Diferenta intre Wireframe si Mockup

2.11. Algoritmi de joc

Jocurile au intrigat omul de secole, poate chiar milenii. Cel mai bun exemplu il reprezinta jocul de Sah. Acesta isi are originile in India secolului 6, asadar, oamenii joaca sah de peste 1400 de ani. Numarul total de pozitii (mutari) ale pieselor este de 10^{40}! Numarul este unul coplesitor, si doar atesta complexitatea acestui joc. Avansul tehnologic din ultimii 70 de ani a facut posibila implementarea asa-zisilor *algoritmi de joc* pe sistemele de calcul. Scopul lor il reprezinta determinarea celei mai bune mutari ce o poate face un jucator, intr-un anumit context.

Astfel, am definit bazele unui algoritm de joc. Complexitatea acestuia este determinata de factori specifici jocului, precum numarul de jucatori, obiectul jocului (spre exemplu, la sah, piesele) sau caracteristicile obiectului jocul (spre exemplu, la sah, mutarile specifice fiecarei piese).

Cele mai multe jocuri din ziua de azi permit desfasurarea jocului cu inlocuirea jucatorului real cu un *bot*. Un bot reprezinta un algoritm de luare a deciziei, in contextul curent al jocului. Contextul poate fi vazut din multiple perspective, din care prezentam urmatoarele:

- Arbore de decizie: un mecanism usor de implementat si de inteles. Fiecare nod din arbore reprezinta un nod de decizie, iar in functie de rezultatul acestuia, se pleaca spre alt nod, pana se ajunge la rezultatul final (frunzele arborelui).

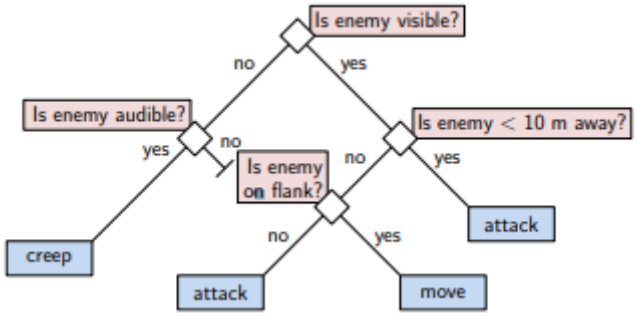

Figura 2.11.1. Mecanism de luare a deciziei pe baza unui arbore de decizie

- Aparat cu stari finite: in cadrul unui joc, exista posibilitatea existentei a unui numar limitat de actiuni si evenimente, ceea ce incurajeaza implementarea unui FSM (finite state machine). Decizia este luata in functie de starea curenta. Reprezentarea starilor poate fi realizata prin intermediul unui graf sau tabelar.

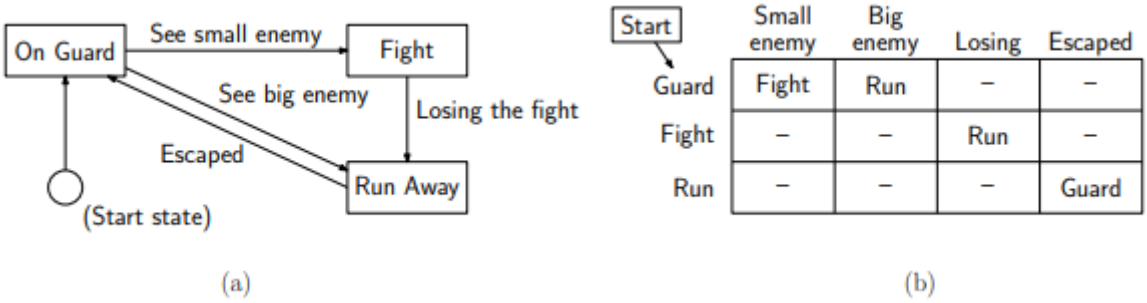

Figura 2.11.2. Reprezentarea starilor unui FSM implementat intr-un joc. Figura a) expune reprezentarea de tip graf, iar figura b) prezinta un tabel al starilor si tranzitiilor

Unul dintre cei mai cunoscuti algoritmi de calcul a deciziei intr-un joc este algoritmul *Mimimax.*

2.11.1. Algoritmul Minimax

Minimax este un algoritm de tip backtracking utilizat in industria jocurilor pentru determinarea mutarii optimale. Algoritmul se pliaza pe jocuri la care participa 2 jucatori, precum sah, backgammon, sau chiar jocuri de societate.

Ideea de baza a algoritmului este sugerata de cele 2 cuvinte din denumirea acestuia: min si max. Fiind necesara existenta a 2 jucatori, unul dintre ei va reprezenta Maximizantul, iar celelalt jucator Maximizantul. Maximizantul cauta sa obtina valoarea maxima ce poate fi obtinuta in contextul curent, in timp ce minimizantul actioneaza in logica opusa, cautand valoarea minima.

Intrebarea care se pune acum este: Ce reprezinta aceasta valoare si de unde provine? Fiecare stare a jocului are o valoare asignata. Aceasta valoare este determinata cu ajutorul unei functii euristice, care va avea ca rezultat o valoare pozitiva sau negativa. Fiecare nod din arbore are asignata o astfel de valoare, reprezentand o star a jocului. O valoare pozitiva va indica o imbunatatire favorabila jucatorului curent, daca se ajunge in aceasta stare. Pe acelasi rationament, o valoare negativa este favorabila jucatorului advers.

Acum ca am definit valorea de tip *scor* a starilor si reprezentarea arboreului de joc, e timpul sa ne axam pe logica de functionare a maximizantului si minimizantului. Cei 2 alterneaza pe nivelurile din arbore, incepand cu maximizantul pentru nodul parinte. Acesta va lua valoarea maxima a nodurilor copil ai acestuia. Pe nivelul urmator va actiona minimizantul, care va lua valoarea minima a nodurilor copil relativ la nodul sau. Maximizantul va fi utilizat pentru nodul urmator s.a.m.d.

Algoritmul, insa, va porni de la bazele arborelui, si anume, de la *frunze.* Logica prezentata anterior se respecta, fiecarui nivel fiindui asociat unul dintre cei 2, incepand cu maximizantul pentru nodul parinte. Pentru nodurile finale se calculeaza valoarea *scor*. Vom avea initial un astfel de arbore:

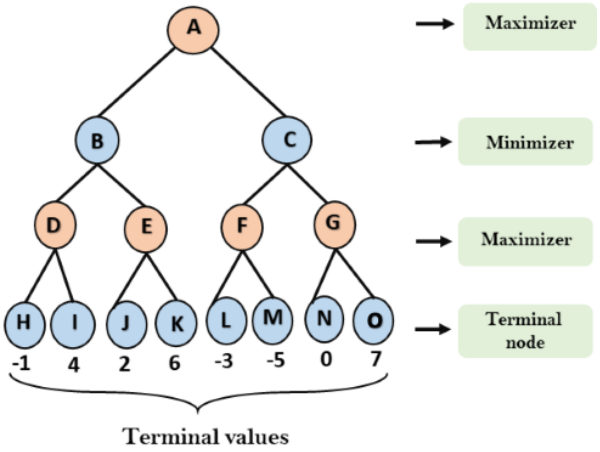

Figura 2.11.3. Starea initiala dupa faza de calcul a valorii scor pentru frunzele arborelui

Urcand cu un nivel de la frunze, observam ca pe acest nivel lucreaza maximizantul. Pentru fiecare nod (stare) al acestui nivel, el va considera valoarea maxima a nodurilor copil. Arborele va avea urmatoarea configuratie dupa acest pas:

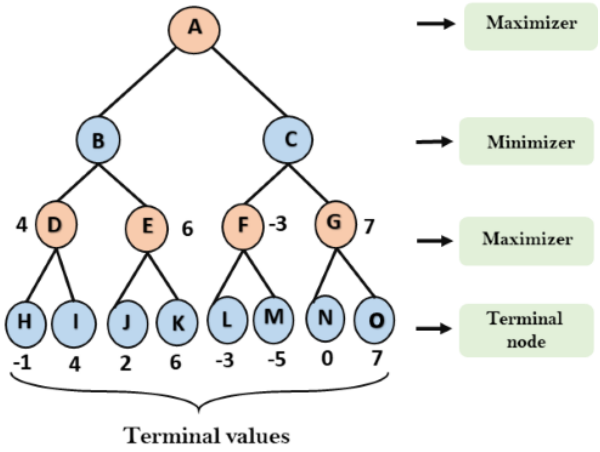

Figura 2.11.4. Starea arborelui dupa evaluarea facuta de maximizant

Se trece la urmatorul nivel din arbore, unde va intra in joc minimzantul. Acesta va considera valorile minime de pe nodurile copil, ce tocmai au fost evaluate de catre maximizant. Rezultatul este urmatorul:

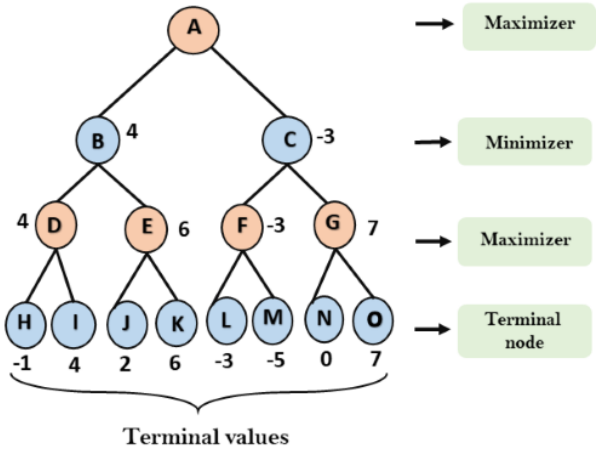

Figura 2.11.5. Starea arborelui dupa evaluarea facuta de maximizant

S-a ajuns la nodul parinte al arborelui, pe care actioneaza maximizantul. Asadar, se va considera valoarea maxima de pe cei 2 copii ai acestuia, acesta fiind si rezultatul final determinat de algoritmul Minimax, pentru acest exemplu:

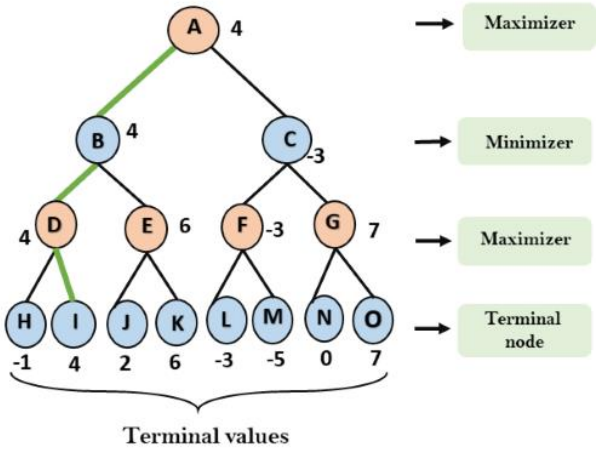

Figura 2.11.6. Starea finala a arborelui, dupa aplicarea algoritmului Minimax

In final, enumeram cateva proprietati ale acestui algoritm:

- Complet: algoritmul mimimax va genera intotdeauna o solutie (ne bazam pe faptul ca arborele de joc este finit).
- Optimal: algoritmul lucreaza pe decizii optime luate de ambii jucatori.
- Complexitate temporala: intru-cat algoritmul utilizeaza tehnica de cautare in adancime in arbore (DFS), complexitatea temporala a acestuia este relativa la rata de ramificatie a acestuia (rr) si la adancimea acestuia (a), si este egala cu $O(rr^a)$.
- Complexitatea spatiala: pentru verificarea tuturor nodurilor din graf (DFS), vom avea o complexitate spatiala egala cu O(rr * m).

3. Descrierea formala a aplicatiei (5-15 pagini)

3.1. Actori

3.2. Use

3.2.1. Admin Use-case

3.2.2. Player Use-case

3.2.3. System Use-case

3.3. Wireframes

3.4. Mockups

3.5. Arhitectura

4. Detalii de implementare (25-30 pagini)

Descriu use-case

5. Concluzii si dezvoltari ulterioare (1-2 pagini)

5.1. Concluzii

5.2. Dezvoltare ulterioare

6. Bibliografie