Ministerul Educației și Cercetării al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare Informatică și Microelectronică

Departamentul Inginerie Software și Automatică

**Proiect de curs**

Disciplina: Tehnici și Mecanisme de Proiectare Software

**Tema:** Beneficiile utilizării șabloanelor de proiectare în dezvoltarea unei aplicații de votare.

**Efectuat:** st. gr. TI-204: Gorea Andrei

**Verificat:** asistent universitar, Mihai Gaidau

**Chișinău 2023**

**Cuprins**

[**Introducere** 3](#_Toc136915150)

[**1 Analiza domeniului de studiu** 4](#_Toc136915151)

[**2.1.1 Factory Method** 7](#_Toc136915152)

[**2.1.1.1 Implementarea șablonului în aplicație** 8](#_Toc136915153)

[**2.1.1.2 Necesitatea implementării șablonului** 9](#_Toc136915154)

[**2.2.1 Singleton** 10](#_Toc136915155)

[**2.2.1.1 Implementarea șablonului în aplicație** 12](#_Toc136915156)

[**2.2.1.2 Necesitatea implementării șablonului** 13](#_Toc136915157)

[**2.3 Șabloanele de proiectare structurale** 13](#_Toc136915158)

[**2.3.1 Adapter** 14](#_Toc136915159)

[**2.3.1.1 Implementarea șablonului în aplicație** 15](#_Toc136915160)

[**2.3.1.2 Necesitatea implementării șablonului** 17](#_Toc136915161)

[**2.3.2 Decorator** 18](#_Toc136915162)

[**2.3.2.1 Implementarea șablonului in aplicație** 19](#_Toc136915163)

[**2.3.2.2 Necesitatea implementării șablonului** 21](#_Toc136915164)

[**2.4 Șabloanele de proiectare comportamentale** 22](#_Toc136915165)

[**2.4.1 Command** 22](#_Toc136915166)

[**2.4.1.1 Implementarea șablonului în aplicație** 24](#_Toc136915167)

[**2.4.1.2 Necesitatea implementării șablonului** 25](#_Toc136915168)

[**2.4.2 Strategy** 26](#_Toc136915169)

[**2.4.2.1 Implementarea șablonului în aplicație** 27](#_Toc136915170)

[**2.4.2.2 Necesitatea implementării șablonului** 28](#_Toc136915171)

[**3. Implementarea aplicației** 29](#_Toc136915172)

[**Bibliografie** 36](#_Toc136915173)

# **Introducere**

În timpurile străvechi ale Greciei și Romei, cetățenii participau la alegeri pentru a-și alege conducătorii. Această tradiție s-a păstrat și în prezent, dar în loc să se limiteze doar la alegerile politice, procesul de votare poate fi utilizat cu succes și în contexte informale, cum ar fi într-un grup de prieteni. Odată cu dezvoltarea aplicațiilor mobile, au fost create aplicații de votare pentru a ușura acest proces. Aceste aplicații aduc o nouă dimensiune divertismentului și procesului de luare a deciziilor în grup, oferind utilizatorilor o platformă interactivă și eficientă pentru a-și exprima preferințele.

Aplicația de votare poate fi utilizată în diverse situații, cum ar fi în cadrul unei întâlniri sau petreceri cu prietenii. Fiecare persoană poate să-și exprime preferințele într-un mod clar și simplu prin intermediul aplicației, iar opțiunea cu cele mai multe voturi devine aleasă. Această aplicație nu doar face procesul de votare mai interactiv, ci și mai transparent, deoarece utilizatorii pot vedea în timp real rezultatul final.

În această lucrare, se analizează importanța votării online și modul în care aceasta se integrează în era digitală. Pentru a descrie și modela sistemul într-un mod standardizat, s-a utilizat limbajul de proiectare UML (Unified Modeling Language).

Pentru dezvoltarea aplicației de votare, pe partea de backend, s-a folosit limbajul de programare TypeScript și framework-ul NestJS. Au fost aplicate diferite tipuri de șabloane de proiectare, precum cele creaționale, structurale și comportamentale, pentru a crea un software bine structurat, flexibil și ușor de întreținut. Interfața grafică și interacțiunea cu utilizatorul au fost implementate utilizând biblioteca React, bazată pe limbajul de programare JavaScript, împreună cu baza de date Redis. Designul interfeței grafice a fost conceput într-un stil minimalist, pentru a asigura o navigare flexibilă pentru utilizatori. Aplicația oferă opțiuni flexibile de configurare și personalizare, pentru a se adapta nevoilor și preferințelor fiecărui grup de persoane.

Aplicația de votare în grupuri de prieteni aduce o nouă dimensiune divertismentului și procesului de luare a deciziilor într-un grup. Prin intermediul acestei aplicații, procesul de luare a deciziilor devine mai ușor și mai interactiv. Astfel, orice dezbatere poate fi transformată într-un vot amuzant și eficient, în care fiecare membru al grupului poate să-și exprime preferințele.

# **1 Analiza domeniului de studiu**

Fiecare persoană se confruntă adesea cu situații în care se află într-un grup mare de prieteni și fiecare are propriile opinii cu privire la o problemă, sau mulți sunt indeciși. În astfel de momente, cel mai simplu este ca fiecare să-și exprime părerea, iar decizia susținută de cel mai mare număr de persoane să fie aleasă. Această aplicație de votare reprezintă o inovație în domeniul tehnologiei informaționale, având un impact semnificativ asupra modului în care luăm decizii colective. Aplicația aduce o abordare modernă și eficientă în procesul de votare, permițând utilizatorilor să-și exprime opțiunile și preferințele într-un mod simplu și accesibil.

Unul dintre aspectele-cheie ale aplicației de votare este utilizarea platformelor web și mobile pentru a facilita procesul de votare. Acest lucru deschide noi posibilități de comunicare și interacțiune, permitând persoanelor să participe la vot de oriunde și oricând, fără a fi limitate de locație sau program.

Pe măsură ce tehnologia informațională și aplicațiile mobile se dezvoltă, accesibilitatea informației și comunicării a crescut semnificativ. Aplicația de votare se bazează pe aceste progrese, oferind utilizatorilor un mediu interactiv și intuitiv pentru a-și exprima votul. Prin intermediul dispozitivelor mobile, utilizatorii pot accesa rapid și ușor aplicația, asigurând astfel o participare activă la procesul de votare.

Un alt aspect important al aplicației de votare este gestionarea eficientă a voturilor și a rezultatelor. Prin intermediul acestei aplicații, voturile pot fi centralizate și organizate într-un mod structurat și ușor de urmărit. Utilizatorii pot vedea rezultatele în timp real și pot înțelege clar preferințele și opțiunile celorlalți participanți. Acest lucru facilitează luarea deciziilor și poate reduce posibilele conflicte sau ambiguități.

Această aplicație are potențialul de a încuraja participarea și implicarea cetățenilor în procesele de luare a deciziilor comunitare sau la nivel mai larg. Prin intermediul acestei aplicații, fiecare persoană poate să-și exprime opiniile și preferințele, ceea ce poate duce la o creștere a transparenței și responsabilității în procesele decizionale.

În concluzie, aplicația de votare reprezintă o inovație semnificativă în domeniul tehnologiei informaționale, oferind un mediu accesibil și interactiv pentru luarea deciziilor colective. Prin utilizarea internetului și a platformelor mobile, această aplicație facilitează procesul de votare, promovând participarea și angajamentul cetățenilor.

**2 Șabloanele de proiectare utilizate**

Christopher Alexander a introdus ideea utilizării unui limbaj bazat pe șabloane pentru proiectarea clădirilor și orașelor, iar această inițiativă a influențat și domeniul dezvoltării software. Proiectanții de software au adoptat utilizarea șabloanelor pentru a construi sisteme complexe.

Șabloanele de proiectare, cunoscute și ca design patterns, reprezintă soluții obișnuite pentru problemele comune în proiectarea software. Fiecare șablon este ca un plan flexibil, care poate fi adaptat pentru a rezolva o problemă specifică de design în codul unei aplicații.

Este important de menționat că modelele de design nu trebuie confundate cu algoritmii, deoarece acestea oferă soluții tipice la probleme cunoscute. În timp ce un algoritm definește o secvență clară de acțiuni pentru atingerea unui obiectiv specific, un model reprezintă o descriere mai abstractă a unei soluții. Codul care implementează același model poate varia în funcție de contextul și nevoile diferitelor programe.

Fiecare șablon este descris de patru elemente:

* Intenția modelului descrie pe scurt atât problema, cât și soluția;
* Motivația explică în continuare problema și soluția pe care modelul o face posibilă;
* Structura claselor arată fiecare parte a modelului și modul în care acestea sunt legate;
* Exemplul de cod într-unul dintre limbajele de programare populare facilitează înțelegerea ideii din spatele modelului.

Exista diferite șabloane de proiectare, acestea pot fi clasificate pe baza de scop și de domeniu. In clasificarea bazata pe scop, distingem:

* Creaționale – procesul de creare a claselor și obiectelor;
* Structurale – compoziția claselor și a obiectelor;
* Comportamentale – interacțiunea și responsabilitățile claselor și obiectelor.

Clasificarea bazata pe domeniul de aplicare:

* Șabloanele obiectelor se refera la relațiile dintre obiecte, relații care au un caracter dinamic;
* Șabloanele claselor se refera la relații dintre clase, relații stabilite prin moștenire si care sunt statice (fixate la compilare).

**2.1 Șabloanele de proiectare creaționale**

Șabloanele de proiectare creationale, cunoscute și ca creational design patterns, reprezintă o categorie de șabloane de proiectare în dezvoltarea software-ului. In ingineria software, acestea sunt modele de design care oferă diverse mecanisme de creare a obiectelor, care cresc flexibilitatea și reutilizarea codului existent. Modelele de design creațional sunt compuse din două idei dominante. Una este încapsularea cunoștințelor despre clasele concrete pe care sistemul le folosește. Un altul ascunde modul în care sunt create și combinate instanțe ale acestor clase concrete.

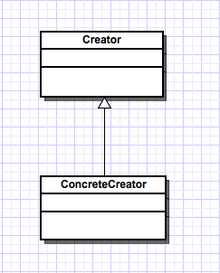
Exista 5 tipuri principale de șabloane creationale:

* Abstract Factory – creează o instanță a mai multor familii de clase;
* Builder – separă construcția obiectului de reprezentarea acestuia;
* Factory Method – creează o instanță a mai multor clase derivate;
* Prototype – o instanță complet inițializată de copiat sau clonat;
* Singleton – o clasă din care poate exista doar o singură instanță.

Desigur, pe lângă șabloanele menționate sus, exista mai multe șabloane creationale cunoscute. Fiecare dintre aceste șabloane abordează diferite aspecte ale procesului de creare a obiectelor și oferă soluții specifice pentru problemele asociate.

In figura 2.1 este reprezentata o diagrama de clasă simplă pe care majoritatea modelelor creaționale au în comun. In aceasta diagrama sunt doi participanți:

* Creator – declara interfața obiectului. Returnează obiectul;
* ConcreteCreator – implementează interfața obiectului.

  
**Figura 2.1 –** Diagrama generala pentru șabloanele creationale [1]

Șabloanele creationale sunt o resursă valoroasă în dezvoltarea software-ului și pot fi utilizate pentru a asigura un design coerent, modular și ușor de întreținut. Utilizarea adecvată a acestor șabloane poate contribui la creșterea flexibilității.

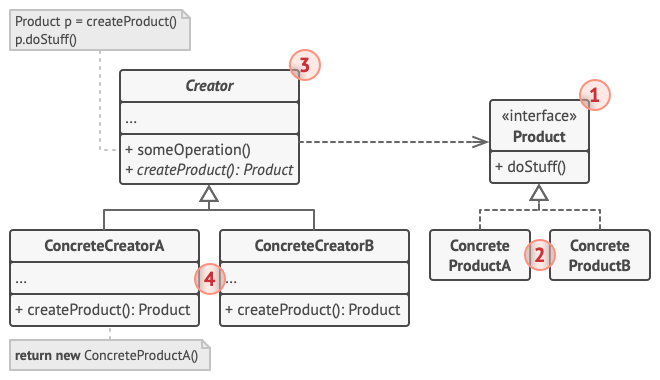
# **2.1.1 Factory Method**

Factory Method este un model de design creațional care oferă o interfață pentru crearea de obiecte într-o super clasă, dar permite sub claselor să modifice tipul de obiecte care vor fi create.

Factory Method se bazează pe principiul polimorfismului și permite obținerea unei instanțe a unei clase concrete prin intermediul unei metode definite într-o clasă abstractă sau interfață. Astfel, în loc să se creeze direct o instanță a unei clase concrete, se folosește metoda fabrică pentru a obține obiectul dorit. Aceasta oferă o mai mare flexibilitate și permite schimbarea clasei concrete returnate de metoda fabrică fără a afecta codul client.

In figura 2.2 este reprezentata structura generala a șablonului factory method. In aceasta diagrama se disting trei clase principale:

* Product - declară interfața, care este comună tuturor obiectelor care pot fi produse de către creator și sub clasele sale;
* Concrete Products - implementări diferite ale interfeței produsului;
* Clasa Creator - declară factory method care returnează obiecte produse noi. Este important ca tipul de returnare al acestei metode să se potrivească cu interfața produsului.



**Figura 2.2 –** Diagrama șablonului Factory Method [2]

Ca si orice șablon, Factory Method are avantajele și dezavantajele sale:

**Avantaje:**

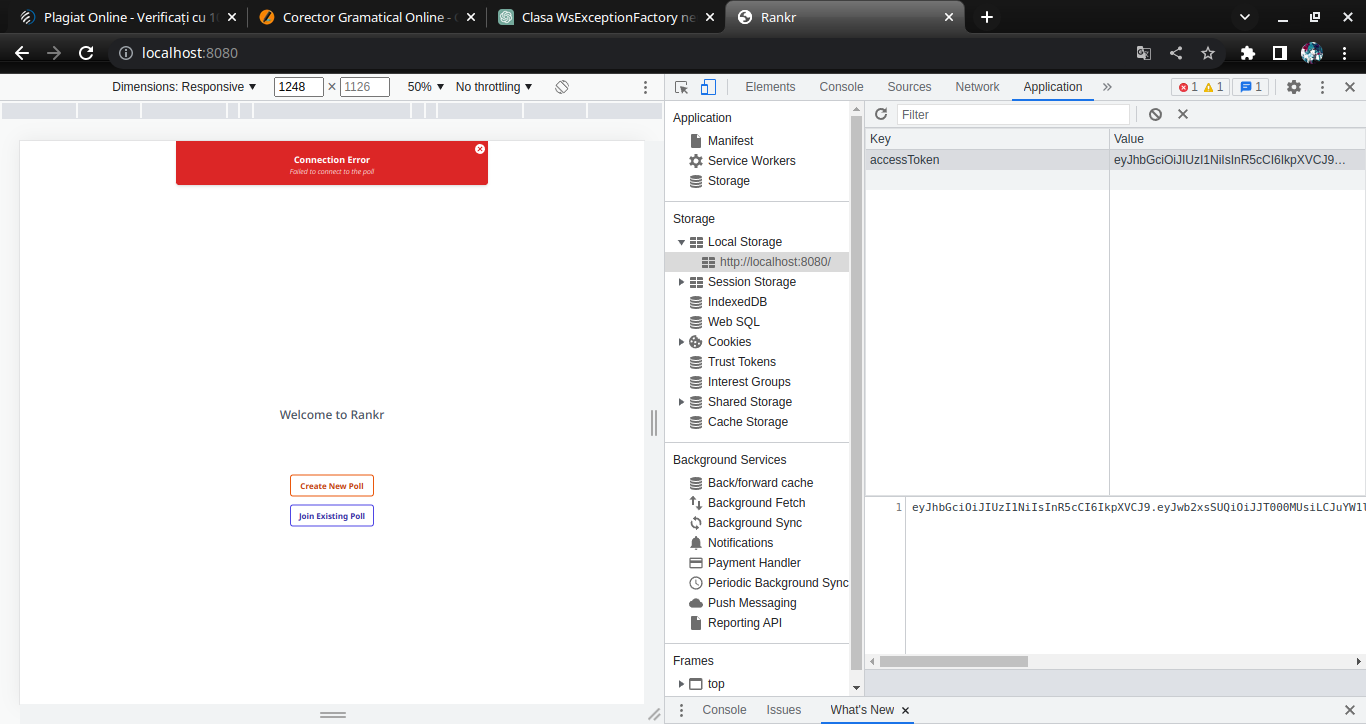
* Se evita cuplarea strânsă între creator și produsele din beton;
* *Principiul responsabilității unice*. Se poate de mutat codul de creare a produsului într-un singur loc din program, făcând codul mai ușor de suportat;
* *Principiul deschis/închis*. Se poate de introdus noi tipuri de produse în program fără a rupe codul client existent.

**Dezavantaje**:

* Codul poate deveni mai complicat, deoarece trebuie să introduceți o mulțime de sub clase noi pentru a implementa modelul. Cel mai bun scenariu este atunci când introduceți modelul într-o ierarhie existentă de clase de creatori.

# **2.1.1.1 Implementarea șablonului în aplicație**

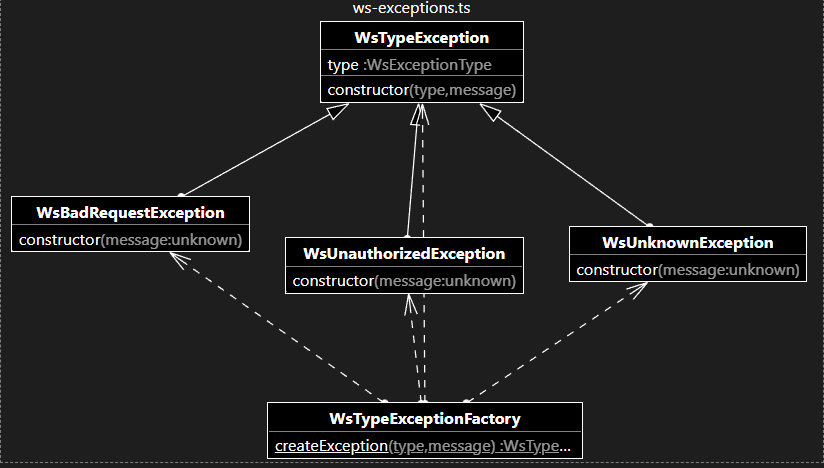
In aplicația de votare se folosește acest șablon, anume la crearea și gestionarea excepțiilor WebSocket, acest lucru se arata in figura 2.3. Aceasta eroare a apărut din cauza ca am avut accessToken diferit de acela care era inițial salvat in browser.



**Figura 2.3 –** Apariția erorii WebSocket in aplicație

În codul prezentat în anexa A, există mai multe clase care sunt folosite pentru gestionarea excepțiilor într-o aplicație WebSocket. Clasa principală este WsTypeException, care extinde clasa WsException și definește un membru numit "type" de tip WsExceptionType. Această clasă conține un constructor care primește un tip și un mesaj și inițializează valorile. Acest constructor apelează și constructorul clasei de bază.

Există trei clase concrete care extind clasa abstractă WsTypeException: WsBadRequestException, WsUnauthorizedException și WsUnknownException. Aceste clase implementează constructori care apelează constructorul clasei de bază și specifică tipurile corespunzătoare de excepții: BadRequest, Unauthorized și Unknown. Aceste clase sunt utilizate pentru a crea instanțe ale excepțiilor specifice. In figura 2.4, este reprezentat diagrama UML a șablonului factory method pentru aplicația de votare:



**Figura 2.4 –** Diagrama șablonului Factory Method pentru aplicația de votare

Utilizând clasa WsTypeExceptionFactory, utilizatorii pot crea și gestiona excepții WebSocket fără a cunoaște detaliile de implementare ale claselor concrete. Astfel, ei pot utiliza metoda "createException" pentru a crea și manipula excepții WebSocket în cadrul aplicației.

Un exemplu simplu de utilizare a acestei funcționalități ar fi:

const exception = WsTypeExceptionFactory.createException('BadRequest', 'Invalid input');

În acest exemplu, se folosește metoda "createException" pentru a crea o instanță a clasei WsBadRequestException cu mesajul "Invalid input". Aceasta oferă o modalitate simplă și unificată de a crea și manipula excepții WebSocket în cadrul aplicației tale.

# **2.1.1.2 Necesitatea implementării șablonului**

Utilizarea șablonului Factory Method în această aplicație aduce beneficii în ceea ce privește modularizarea și abstractizarea codului. Clasa WsExceptionFactory acționează ca un creator de excepții și oferă o metodă centralizată, createWsException(), pentru crearea excepțiilor. Această abordare separă logica de creare a obiectelor de restul codului și oferă flexibilitate pentru adăugarea ulterioară a unor noi tipuri de excepții.

Prin utilizarea acestui șablon, se obține o structură mai abstractă și adaptabilă. Adăugarea de noi tipuri de excepții presupune definirea de clase derivate din clasa abstractă WsTypeException și implementarea metodei corespunzătoare. Astfel, sistemul de gestionare a excepțiilor poate fi extins fără a fi nevoie de modificări în codul existent.

Utilizarea șablonului Factory Method ascunde detaliile de implementare referitoare la crearea excepțiilor. Utilizatorii clasei WsExceptionFactory nu trebuie să fie conștienți de clasele concrete de excepții, ci pot utiliza doar metoda createWsException() pentru a crea excepții dorite. Această abordare asigură o separare clară între logica de creare și utilizarea excepțiilor, fără a expune detalii specifice de implementare.

# **2.2.1 Singleton**

Singleton este un model de design creațional care permite crearea unei singure instanțe a unei clase în întreaga aplicație și oferă un punct global de acces la această instanță.

Prin implementarea Singleton, se garantează că există doar o singură instanță a clasei și că aceasta poate fi accesată de la orice parte a aplicației. Aceasta se realizează prin restricționarea accesului la constructorul clasei și prin furnizarea unei metode statice care returnează instanța existentă sau creează una nouă, dacă nu există deja.

Modelul Singleton este util în situațiile în care trebuie să existe o singură instanță a unei clase și aceasta trebuie să fie accesibilă global. De exemplu, în cazul în care o clasă reprezintă o conexiune la o bază de date sau un sistem extern, Singleton asigură că această conexiune este unică și poate fi partajată între diferitele componente ale aplicației. În general, Singleton este implementat prin folosirea unei variabile statice private în clasa Singleton, care stochează instanța unică a clasei.

Metoda statică de acces la instanță verifică dacă instanța există deja și o returnează sau o creează în caz contrar. Astfel, se asigură că fiecare apel la metoda de acces va returna aceeași instanță. Implementarea corectă a Singleton este importantă pentru a evita problemele legate de concurență și pentru a asigura consistența instanței în întreaga aplicație.



**Figura 2.2 –** Diagrama șablonului Singleton [15]

Ca si orice șablon, Singleton are avantajele și dezavantajele sale:

**Avantaje:**

* Acces global: Singleton oferă un punct global de acces la instanța sa, ceea ce facilitează interacțiunea cu aceasta de către alte componente ale aplicației.
* O singură instanță: Singleton garantează că există o singură instanță a clasei în întreaga aplicație. Aceasta poate fi utilă în cazul în care trebuie să se asigure existența unui singur obiect al unei clase.
* Eficiență: Deoarece instanța Singleton este creată o singură dată și apoi este reutilizată, se poate obține o eficiență sporită în aplicație.

**Dezavantaje**:

* Dificultăți în testare: Din cauza dependenței strânse de instanța Singleton, testarea unitară a componentelor care folosesc Singleton poate fi dificilă. Este important să se ia în considerare gestionarea dependențelor în timpul testării.

# **2.2.1.1 Implementarea șablonului în aplicație**

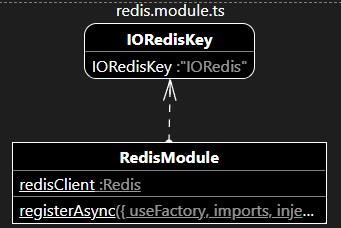
În codul prezentat în anexa C, se implementează un exemplu simplu al design pattern-ului Singleton pentru gestionarea unei conexiuni la baza de date Redis în cadrul unui modul RedisModule.

Design pattern-ul Singleton are ca scop restricționarea creării unei singure instanțe a unei clase și oferirea unui punct global de acces la acea instanță. În acest caz, clasa RedisModule are un membru privat static numit "redisClient" de tip Redis, care va reprezenta instanța unică a conexiunii la Redis.

Metoda statică "registerAsync" este folosită pentru a obține instanța Singleton a clasei RedisModule. În această metodă, se verifică dacă instanța "redisClient" nu există deja. Dacă nu există, se creează o nouă instanță a conexiunii Redis utilizând opțiunile de configurare primite prin intermediul metodei "useFactory". Această instanță este stocată în membrul static "redisClient" pentru utilizare ulterioară.

În cazul în care instanța "redisClient" deja există, se returnează modulul RedisModule fără a crea o nouă instanță. Aceasta asigură că o singură instanță a conexiunii Redis este utilizată în întreaga aplicație.

Astfel, design pattern-ul Singleton în acest cod permite asigurarea că există o singură instanță a conexiunii Redis în cadrul modulului RedisModule, evitând crearea de instanțe multiple și oferind un punct global de acces la acea instanță.



**Figura 2.4 –** Diagrama șablonului Singleton

# **2.2.1.2 Necesitatea implementării șablonului**

Utilizarea design pattern-ului Singleton în acest cod este justificată de necesitatea de a asigura existența unei singure instanțe a conexiunii la baza de date Redis în cadrul modulului RedisModule.

Există câteva motive principale pentru utilizarea design pattern-ului Singleton într-o astfel de situație:

* Limitarea resurselor: O conexiune la o bază de date poate fi o resursă costisitoare în termeni de timp și resurse de sistem. Prin utilizarea unui Singleton, se asigură că se creează și se utilizează o singură instanță a conexiunii, evitând crearea inutilă a mai multor instanțe și economisind resurse.
* Consistență și coerență: Singleton-ul asigură că toate componentele care utilizează conexiunea la baza de date accesează aceeași instanță. Acest lucru ajută la menținerea consistenței și coerenței datelor și acțiunilor realizate asupra bazei de date.
* Punct global de acces: Singleton-ul oferă un punct global de acces la instanța conexiunii. Aceasta facilitează utilizarea și gestiunea conexiunii în întreaga aplicație, deoarece nu este nevoie să se transmită și să se mențină instanța conexiunii în diferitele componente.
* Controlul asupra instanței: Utilizând Singleton, se poate controla modul în care instanța conexiunii este creată și utilizată. De exemplu, în codul prezentat, se verifică dacă instanța deja există înainte de a crea una nouă, asigurându-se astfel că se utilizează o singură instanță și evitând crearea inutilă de noi conexiuni.

În concluzie, utilizarea design pattern-ului Singleton în acest cod este necesară pentru a asigura că există o singură instanță a conexiunii la baza de date Redis în cadrul modulului RedisModule, oferind economie de resurse, consistență a datelor și un punct global de acces controlat la instanță.

# **2.3 Șabloanele de proiectare structurale**

Șabloanele de proiectare structurale, cunoscute și ca structural design patterns, reprezintă o categorie de șabloane de proiectare în dezvoltarea software-ului. In ingineria software, acestea sunt modele de design care explică modul de asamblare a obiectelor și claselor în structuri mai mari, păstrând în același timp aceste structuri flexibile și eficiente.

Exista 7 tipuri principale de șabloane structurale:

* Adapter– permite obiectelor cu interfețe incompatibile sa colaboreze;
* Bridge – permite împărțirea unei clase mari sau a unui set de clase strâns legate în două ierarhii separate - abstracție și implementare - care pot fi dezvoltate independent una de cealaltă;
* Composite– permite compunerea obiectelor în structuri arborescente și apoi putem lucra cu aceste structuri ca și cum ar fi obiecte individuale;
* Decorator – permite atașarea noilor comportamente la obiecte prin plasarea acestor obiecte în interiorul obiectelor speciale care conțin comportamentele;
* Facade – oferă o interfață simplificată la o bibliotecă, un cadru sau orice alt set complex de clase;
* Flyweight - permite încadrarea mai multor obiecte în cantitatea disponibilă de RAM, partajând părți comune de stare între mai multe obiecte, în locul la păstrarea tuturor datelor în fiecare obiect;
* Proxy - permite oferirea unui substitut sau a unui substituent pentru alt obiect.

Fiecare dintre aceste șabloane facilitează separarea responsabilităților, reutilizarea codului, extensibilitatea, modularitatea și flexibilitatea în dezvoltarea software.

# **2.3.1 Adapter**

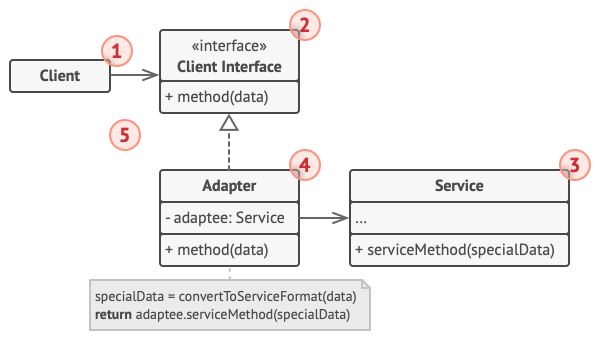
Adapter-ul este un model de design structural care permite obiectelor cu interfețe incompatibile să colaboreze.

Modelul de design al adaptorului rezolvă probleme precum:

* Cum poate fi reutilizată o clasă care nu are o interfață pe care o cere un client?;
* Cum pot lucra împreună clasele care au interfețe incompatibile? etc.

In figura 2.5 este reprezentata structura generala a șablonului adapter. In aceasta diagrama se disting patru clase principale:

* Client - clasa care conține logica de afaceri existentă a programului;
* Client Interface - descrie un protocol pe care alte clase trebuie să-l urmeze pentru a putea colabora cu codul clientului;
* Service - clasă utilă. Clientul nu poate folosi această clasă direct deoarece are o interfață incompatibilă;
* Adapter - clasă care poate lucra atât cu clientul, cât și cu serviciul: implementează interfața client, în timp ce încapsulează obiectul serviciului.

  
**Figura 2.5 –** Diagrama șablonului Adapter [3]

Avantajele și dezavantajele șablonului adapter sunt:

**Avantaje:**

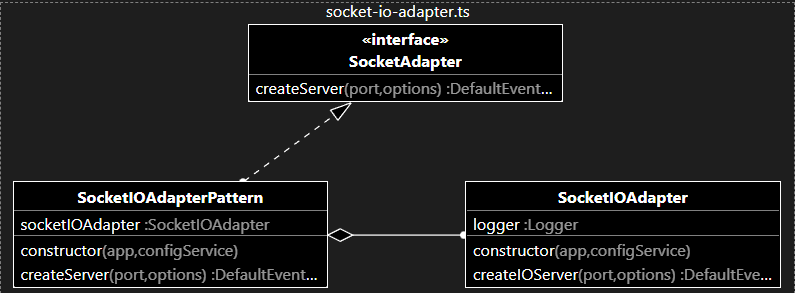
* *Principiul responsabilității unice* . Puteți separa interfața sau codul de conversie a datelor de logica comercială principală a programului;
* *Principiul deschis/închis* . Puteți introduce noi tipuri de adaptoare în program fără a rupe codul client existent, atâta timp cât funcționează cu adaptoarele prin interfața client.

**Dezavantaje:**

* Complexitatea generală a codului crește deoarece trebuie să introduceți un set de interfețe și clase noi. Uneori este mai simplu să schimbi clasa de serviciu, astfel încât să se potrivească cu restul codului tău.

# **2.3.1.1 Implementarea șablonului în aplicație**

In aplicația de votare se foloseste acest sablon pentru comunicarea în timp real între server și client prin intermediul socket-urilor. In figura 2.6, este reprezentata diagrama UML a șablonului adapter pentru codul prezentat in anexa B.



**Figura 2.6 –** Diagrama sabonului Adapter pentru aplicatia de votare

În acest cod, clasa SocketIOAdapter extinde clasa IoAdapter din biblioteca NestJS, care furnizează funcționalități de adaptare și configurare a serverului Socket.IO. Prin extinderea acestei clase, putem personaliza comportamentul și setările serverului Socket.IO în funcție de necesitățile noastre.

Constructorul clasei SocketIOAdapter primește două dependențe: app, de tip INestApplicationContext, și configService, de tip ConfigService. Aceste dependențe sunt utilizate pentru a accesa și obține servicii și configurații necesare pentru funcționarea adaptatorului.

Metoda createIOServer, suprascrisă din clasa de bază IoAdapter, este utilizată pentru a crea și configura serverul Socket.IO. În această metodă, se configurează opțiunile serverului și se personalizează setările CORS (Cross-Origin Resource Sharing). Configurațiile CORS permit specificarea originilor acceptate pentru a evita erorile de securitate legate de politica same-origin.

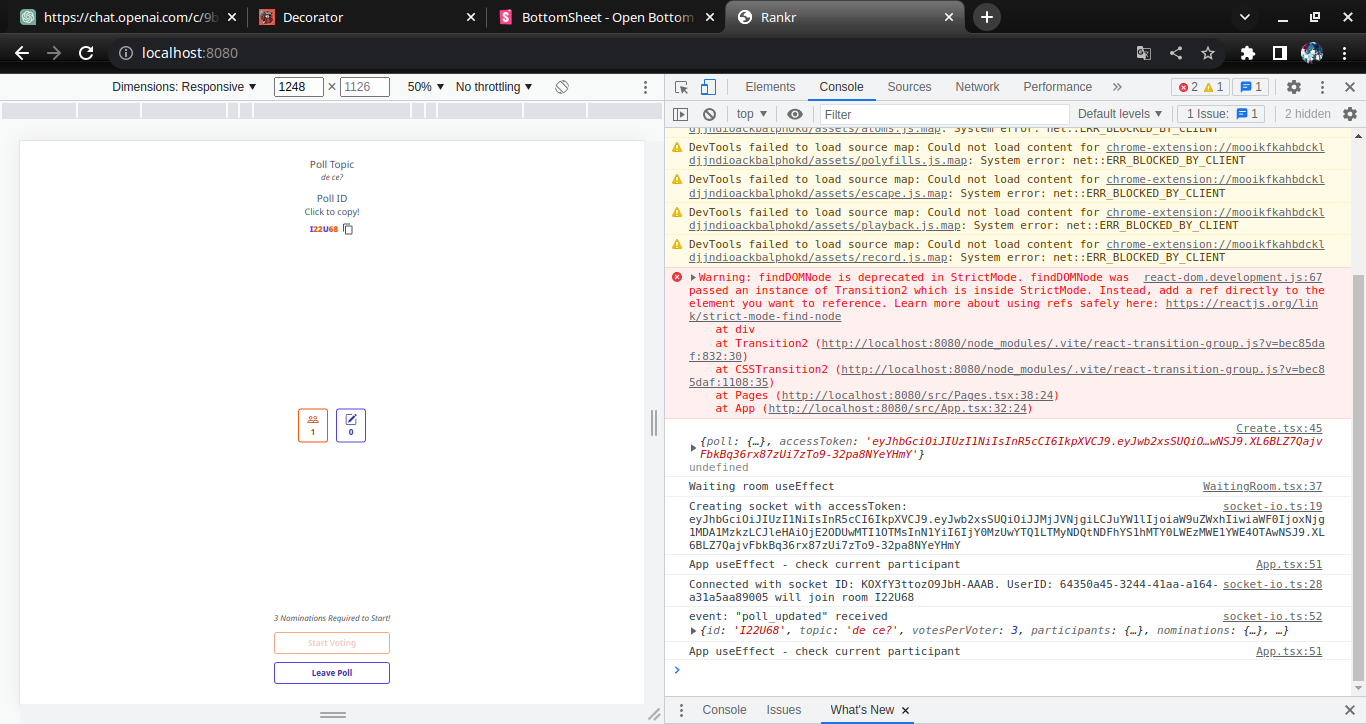
În cadrul metodei createIOServer, este definită și utilizată funcția createTokenMiddleware. Această funcție primește serviciul jwtService și obiectul logger, și returnează un middleware de autentificare pentru socket-uri. Middleware-ul validează token-ul JWT furnizat în cererea de conectare a socket-ului și setează câmpurile suplimentare ale obiectului socket pentru a stoca informații relevante despre utilizatorul conectat.

În final, serverul Socket.IO este creat prin apelarea metodei super.createIOServer și este configurat să utilizeze middleware-ul de autentificare definit anterior pentru "polls". Astfel, fiecare cerere de conectare la "polls" este verificată înainte de a permite accesul la socket.

Implementarea șablonului Adapter în codul dat demonstrează adaptarea și personalizarea serverului Socket.IO prin intermediul clasei SocketIOAdapter. Această adaptare permite gestionarea autentificării și configurarea serverului în funcție de cerințele specifice ale aplicației.

# **2.3.1.2 Necesitatea implementării șablonului**

Șablonul Adapter este utilizat în aplicație pentru a rezolva o problemă de compatibilitate între două componente diferite: serverul Socket.IO și framework-ul NestJ, acesta ajuta ca utilizatorul sa se conecteze la aplicație, figura 2.7.



**Figura 2.7 –** Conectarea utilizatorului la aplicatie prin socket

Unul dintre principalele motive pentru utilizarea șablonului Adapter în acest caz este că framework-ul NestJS furnizează propriul modul IoAdapter pentru integrarea cu serverele WebSocket. Cu toate acestea, serverul Socket.IO utilizează o interfață diferită și nu este direct compatibil cu modulul IoAdapter furnizat de NestJS. Prin urmare, este necesară implementarea unui adapter personalizat, în acest caz clasa SocketIOAdapter, care extinde clasa IoAdapter și furnizează funcționalitatea specifică a serverului Socket.IO.

Utilizarea șablonului Adapter permite o separare clară a responsabilităților între cele două componente. Serverul Socket.IO se concentrează pe gestionarea conexiunilor WebSocket și a evenimentelor asociate, în timp ce clasa SocketIOAdapter se ocupă de adaptarea interfeței și a funcționalităților specificate de framework-ul NestJS. Astfel, adaptarea este realizată într-un mod modular și flexibil, permițând extinderea și modificarea comportamentului serverului Socket.IO fără a afecta direct codul existent al aplicației.

Implementarea unui adapter personalizat, cum ar fi SocketIOAdapter, oferă beneficii suplimentare, cum ar fi abstractizarea detaliilor de implementare specifice serverului Socket.IO. Acest lucru permite programatorilor să se concentreze pe logica de aplicație și să utilizeze interfețe familiare și consistente furnizate de framework-ul NestJS. În plus, prin intermediul adapterului, se pot adăuga și configura opțiuni specifice serverului Socket.IO, precum setările CORS, într-un mod centralizat și ușor de gestionat.

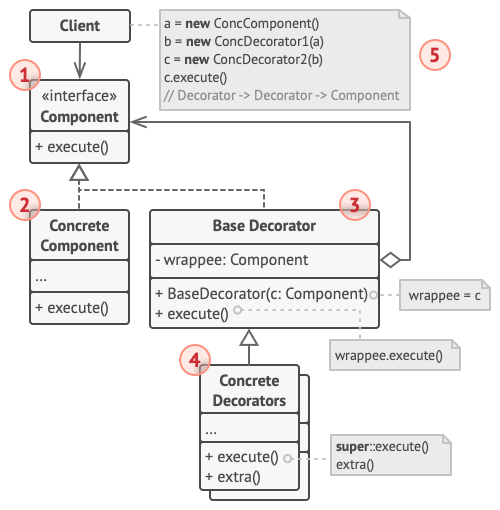
În concluzie, utilizarea șablonului Adapter în acest scenariu facilitează integrarea serverului Socket.IO în cadrul aplicației NestJS, asigurând o separare clară între logica specifică Socket.IO și restul aplicației.

# **2.3.2 Decorator**

Decorator este un model de design structural care vă permite să atașați noi comportamente la obiecte prin plasarea acestor obiecte în interiorul obiectelor speciale care conțin comportamentele.

Modelul de design al decoratorului descriu cum se rezolvă problemele recurente de proiectare și se proiectează software flexibil și reutilizabil orientat pe obiecte, adică obiecte care sunt mai ușor de implementat, schimbat, testat și reutilizat. In figura 2.8 este reprezentata structura generala a șablonului decorator. In aceasta diagrama se disting cinci clase principale:

* Component - declară interfața comună atât pentru ambalaje, cât și pentru obiectele înfășurate;
* Concrete Component - clasă de obiecte înfășurate. Acesta definește comportamentul de bază, care poate fi modificat de decoratori;
* Base Decorator - clasă care are un câmp pentru referirea unui obiect înfășurat;
* Concrete Decorators - definește comportamente suplimentare care pot fi adăugate componentelor în mod dinamic;
* Client - împachetează componente în mai multe straturi de decoratori, atâta timp cât funcționează cu toate obiectele prin intermediul interfeței componente.



**Figura 2.8 –** Diagrama șablonului Decorator [4]

Avantajele și dezavantajele șablonului decorator sunt:

**Avantaje:**

* Puteți extinde comportamentul unui obiect fără a crea o nouă subclasă;
* Puteți adăuga sau elimina responsabilități dintr-un obiect în timpul execuției;
* Puteți combina mai multe comportamente prin înfășurarea unui obiect în mai mulți decoratori;
* *Principiul responsabilității unice* . Puteți împărți o clasă monolitică care implementează multe variante posibile de comportament în mai multe clase mai mici.

**Dezavantaje:**

* Este greu să scoți un anumit ambalaj din stiva de ambalaje;
* Este greu să implementezi un decorator în așa fel încât comportamentul său să nu depindă de ordinea din stiva de decoratori;
* Codul de configurare inițial al straturilor ar putea arăta destul de urât.

# **2.3.2.1 Implementarea șablonului in aplicație**

In aplicația de votare se folosește acest șablon pentru ca proteja anumite rute sau acțiuni în aplicație. In figura 2.9, este reprezentata diagrama UML a șablonului decorator pentru aplicația creata pentru acest raport.



**Figura 2.9 –** Diagrama șablonului Decorator pentru aplicatia de votare

În codul prezentat in anexa C implementarea șablonului decorator este reprezentat prin utilizarea a două clase: GatewayAdminGuard și LoggingGatewayAdminGuardDecorator.

Clasa GatewayAdminGuard reprezintă o componentă importantă în sistemul de autorizare, responsabilă de verificarea privilegiilor de administrator pentru utilizatori. Aceasta utilizează serviciul PollsService pentru a obține informațiile necesare și JwtService pentru a verifica token-ul de autentificare furnizat.

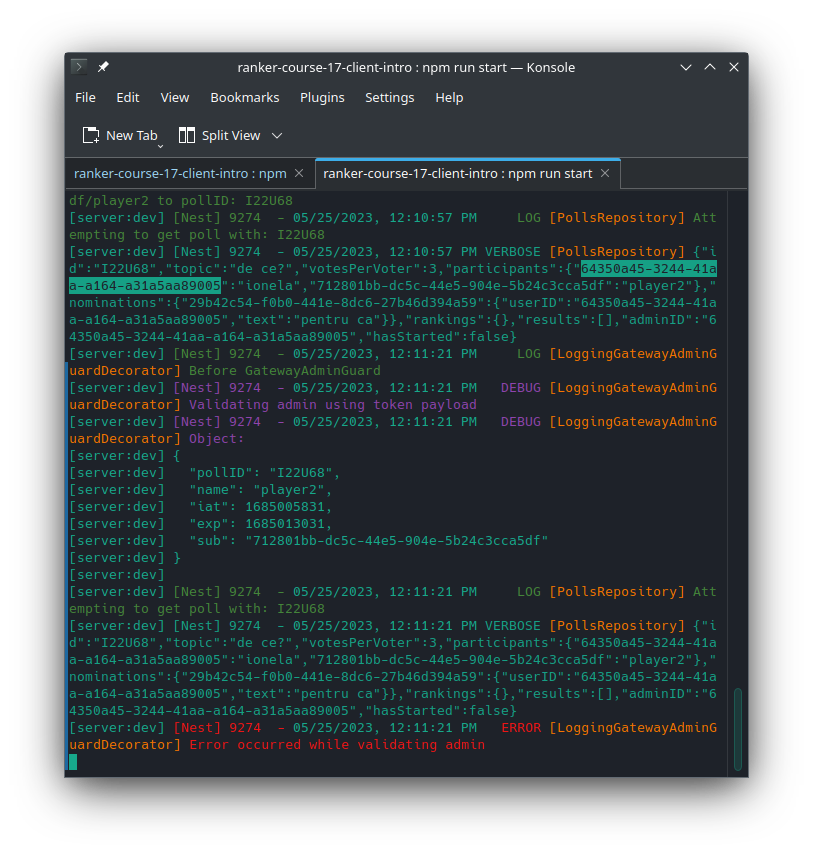
Metoda canActivate primește contextul de execuție și verifică dacă utilizatorul autentificat are privilegii de administrator pentru sondajul specificat. Dacă token-ul lipsește sau utilizatorul nu este un administrator, se aruncă o excepție WsUnauthorizedException pentru a indica eroarea de autorizare.

Clasa LoggingGatewayAdminGuardDecorator reprezintă un decorator pentru clasa GatewayAdminGuard. Acest decorator extinde comportamentul clasei de bază prin adăugarea de funcționalități suplimentare, cum ar fi înregistrarea în jurnalele de log.

Metoda canActivate a decoratorului adaugă înregistrări de log înainte și după apelul metodei canActivate a obiectului de bază. Înainte de apelul metodei de bază, se afișează un mesaj de log pentru a indica momentul în care s-a ajuns în decorator. Apoi, se apelează metoda canActivate a obiectului de bază și se obține rezultatul.

După apelul metodei de bază, decoratorul adaugă și mai multă logica suplimentară. În exemplul dat, se afișează un mesaj de log în funcție de rezultatul returnat de obiectul de bază. Astfel, se poate observa dacă metoda canActivate a obiectului de bază a returnat true sau false, și se înregistrează acest lucru în jurnalele de log pentru scopuri de depanare și monitorizare.

În concluzie, utilizarea unui decorator precum LoggingGatewayAdminGuardDecorator oferă flexibilitate în extinderea funcționalității clasei GatewayAdminGuard prin adăugarea de logica suplimentară și înregistrări de log. Acesta respectă principiile de separare a responsabilităților și modularitate, permițând extinderea și modificarea comportamentului clasei de bază fără a o modifica direct.



**Figura 2.10 –** Implementarea șablonului Decorator in clasa LoggingGatewayAdminGuardDecorator

Prin utilizarea decoratorului, putem extinde funcționalitatea clasei de bază GatewayAdminGuard fără a o modifica direct. Decoratorul ne permite să adăugăm logica suplimentară înainte, după sau în jurul metodelor de bază și să obținem rezultatele dorite fără a afecta comportamentul clasei de bază sau a crea dependențe complexe între clase.

# **2.3.2.2 Necesitatea implementării șablonului**

Utilizarea șablonului decorator în aplicație este necesară pentru a extinde flexibil și modular funcționalitatea clasei GatewayAdminGuard, fără a modifica clasa de bază și fără a afecta contractul existent. Prin aplicarea decoratorului, se realizează o separare clară între funcționalitatea de bază și funcționalitățile adiționale.

Decoratorul permite adăugarea de comportamente suplimentare în jurul metodelor de bază, menținând implementarea originală intactă. Astfel, putem adăuga funcționalități suplimentare la clasa GatewayAdminGuard fără a afecta modul în care este utilizată în alte părți ale codului.

De asemenea, decoratorul permite adăugarea sau eliminarea dinamică a funcționalităților suplimentare la obiectul decorat. Aceasta înseamnă că putem adăuga sau elimina logica suplimentară în timpul rulării aplicației, fără a fi necesare modificări în alte părți ale codului.

În plus, șablonul decorator permite compunerea mai multor decoratori pentru a obține un comportament complex și modular. Prin adăugarea și combinarea diferitelor decoratori, putem obține combinații variate de funcționalități suplimentare pentru clasa GatewayAdminGuard, în funcție de nevoile specifice ale aplicației.

Utilizarea șablonului decorator facilitează dezvoltarea unei arhitecturi flexibile și extensibile, în care putem adăuga și modifica funcționalități într-un mod controlat și modular, fără a afecta restul aplicației.

# **2.4 Șabloanele de proiectare comportamentale**

Șabloanele de proiectare comportamentale, cunoscute și ca behavioral design patterns, reprezintă o categorie de șabloane de proiectare în dezvoltarea software-ului. In ingineria software, acestea sunt modele de design care sunt preocupate de algoritmi și de atribuirea responsabilităților între obiecte.

Exista 10 tipuri principale de șabloane comportamentale:

* Chain of Responsibility – o modalitate de a transmite o solicitare între un lanț de obiecte;
* Command – incapsulează o cerere de comandă ca obiect;
* Iterator – accesați secvențial elementele unei colecții;
* Mediator – definește comunicarea simplificată între clase;
* Memento – capturați și restabiliți starea internă a unui obiect;
* Observer – o modalitate de a notifica modificarea unui număr de clase;
* State – alterează comportamentul unui obiect atunci când starea acestuia se schimbă;
* Strategy - încapsulează un algoritm în interiorul unei clase;
* Template – amânați pașii exacți ai unui algoritm la o sub clasă;
* Visitor – definește o nouă operațiune la o clasă fără modificare.

Desigur, pe lingă șabloanele menționate sus, exista mai multe șabloane comportamentale cunoscute. Fiecare dintre aceste șabloane abordează diferite aspecte ale procesului de creare a obiectelor și oferă soluții specifice pentru problemele asociate.

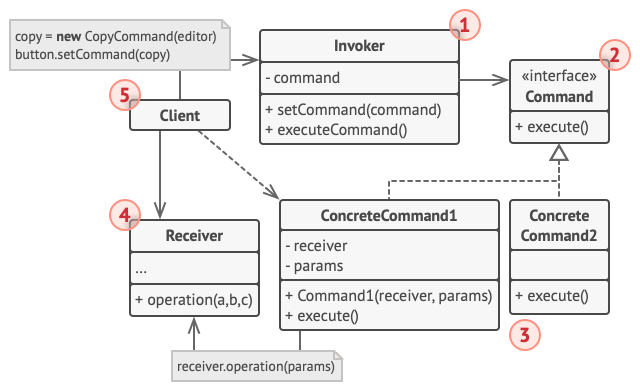
# **2.4.1 Command**

Command este un model de design comportamental care transformă o solicitare într-un obiect autonom care conține toate informațiile despre cerere. Această transformare vă permite să treceți cererile ca argumente de metodă, să întârziați sau să puneți în coadă execuția unei cereri și să suportați operațiuni care nu pot fi anulate.

Ideile centrale ale acestui model de design reflectă de-aproape semantica [funcțiilor de primă clasă](https://en.wikipedia.org/wiki/First-class_function) și [a funcțiilor de ordin superior](https://en.wikipedia.org/wiki/Higher-order_function) în [limbaje de programare funcționale](https://en.wikipedia.org/wiki/Functional_programming_language) . Mai exact, obiectul invocator este o funcție de ordin superior al cărei obiect de comandă este un argument de primă clasă.

In figura 2.13 este reprezentata structura generala a șablonului chain of responsibility. In aceasta diagrama se disting cinci clase principale:

* Sender - responsabilă pentru inițierea cererilor;
* Command – interfața care declară de obicei doar o singură metodă pentru executarea comenzii;
* Concrete Commands - implementează diverse tipuri de solicitări;
* Receiver - conține o anumită logică de afaceri;
* Client - creează și configurează obiecte de comandă concrete.

  
**Figura 2.13 –** Diagrama șablonului Command [6]

Avantajele și dezavantajele șablonului:

**Avantaje:**

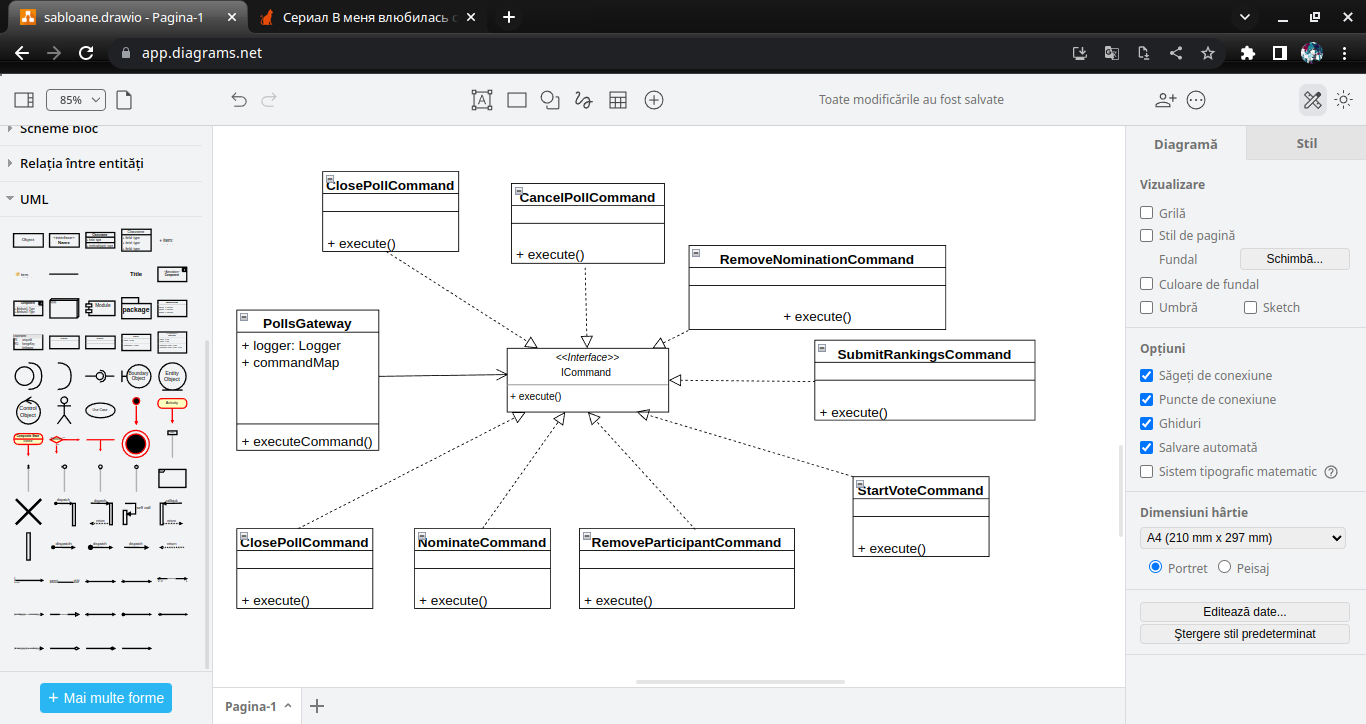
* *Principiul responsabilității unice* . Puteți decupla clasele care invocă operații de clasele care efectuează aceste operații;
* *Principiul deschis/închis* . Puteți introduce comenzi noi în aplicație fără a rupe codul client existent;
* Puteți implementa anularea/refacerea;
* Puteți implementa execuția amânată a operațiunilor;
* Puteți asambla un set de comenzi simple într-unul complex.

**Dezavantaje:**

* Codul poate deveni mai complicat, deoarece introduceți un strat complet nou între emițători și receptori.

# **2.4.1.1 Implementarea șablonului în aplicație**

Pentru a separa comenzi specifice în obiecte individuale, permițând astfel parametrizarea și gestionarea acestora într-un mod flexibil și modular în aplicație, am implementat șablonul command. In figura 2.14, este reprezentata diagrama UML a șablonului command pentru aplicația de votare.



**Figura 2.14 –** Diagrama șablonului Command pentru aplicația de votare

În codul din anexa D, se definesc mai multe clase care extind clasa abstractă Command, care la rândul ei implementează interfața ICommand. Aceasta este o implementare a șablonului Command, în care comenzi specifice sunt definite ca obiecte separate și reutilizabile.

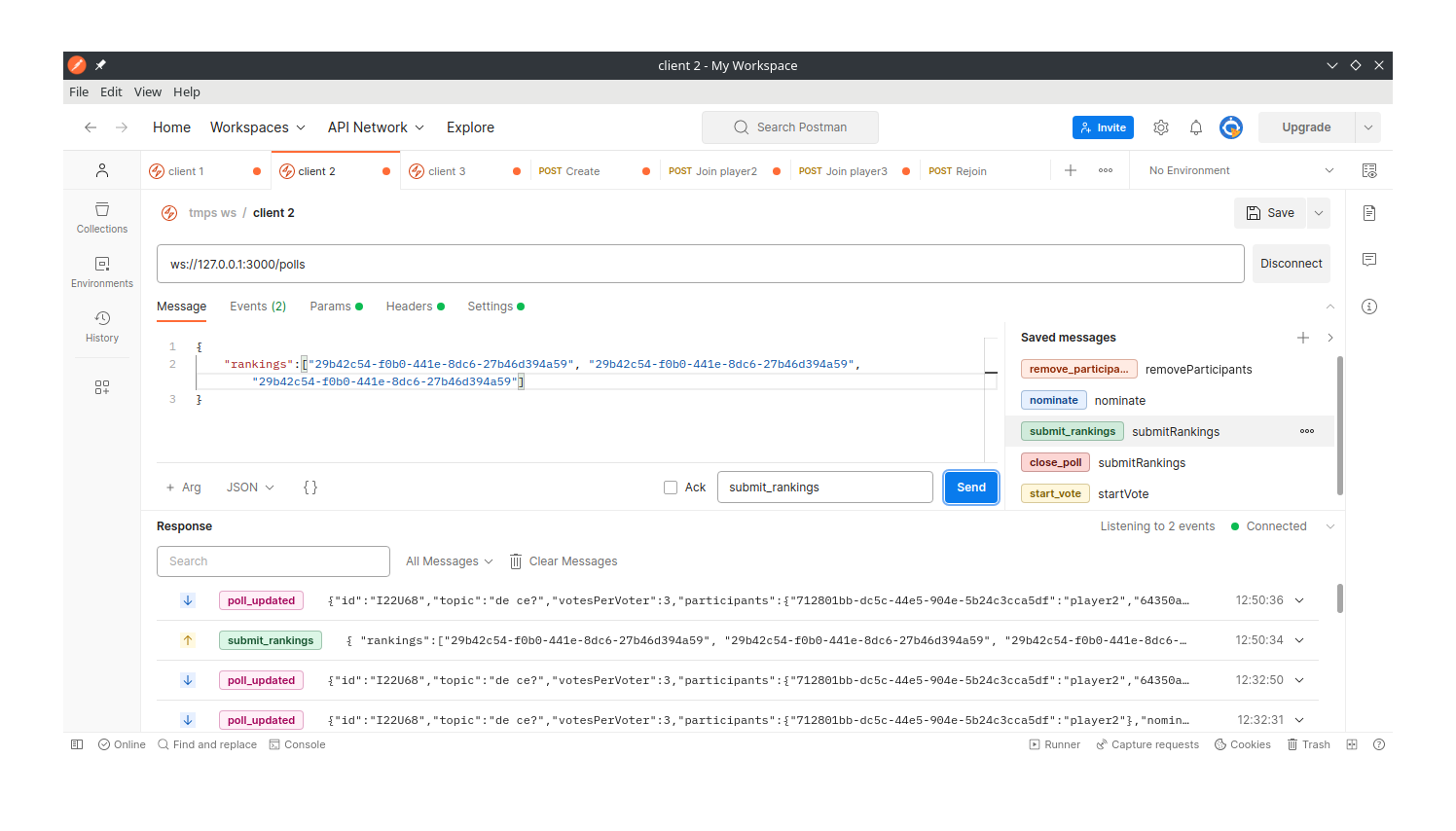
Fiecare clasă de comandă concretă, cum ar fi RemoveParticipantCommand, NominateCommand, StartVoteCommand, etc., implementează metoda execute, care primește un obiect de argumente și execută logica specifică acelei comenzi. Aceste comenzi sunt utilizate pentru a gestiona acțiunile specifice legate de sondaje și interacțiunea cu participanții și nominalizările acestora.

Clasa PollsGateway este responsabilă de gestionarea evenimentelor și conexiunilor WebSocket și utilizează șablonul Command pentru a executa comenzi specifice în funcție de evenimentele primite. În constructorul său, clasa PollsGateway înregistrează instanțe ale claselor de comandă într-o hartă (commandMap), astfel încât să poată fi accesate ulterior în timpul execuției.

Metoda executeCommand primește numele unei comenzi și argumentele corespunzătoare și caută comanda în mapa commandMap. Dacă găsește o potrivire, execută metoda execute a acelei comenzi. Astfel, se realizează o separare clară între comanda specifică și logica de executare asociată, permițând astfel o extensibilitate și o reutilizare ușoară a comenzilor.

# **2.4.1.2 Necesitatea implementării șablonului**

Șablonul command este utilizat în aceasta aplicație pentru a oferi o abordare modulară și flexibilă pentru gestionarea comenzilor în cadrul aplicației, cum ar fi exemplul din figura 2.15, unde utilizatorul voteaza optiunile care le vrea.



**Figura 2.15 –** Implementarea șablonului în aplicație

Una dintre necesitățile acestui șablon în cadrul aplicației este de a separa logica de execuție a comenzilor de logica de gestionare a evenimentelor și de comunicare prin socket-uri. Astfel, fiecare comandă este implementată într-o clasă separată, cu propria logica specifică. Aceasta permite ca fiecare comandă să fie extinsă și modificată independent, fără a afecta restul codului.

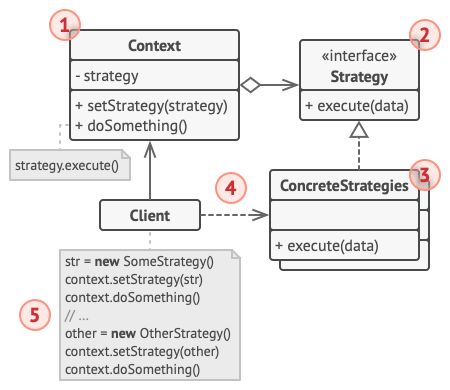
De asemenea, utilizarea șablonului Command permite o gestiune mai ușoară a comenzilor și a logicii aferente. Acestea pot fi înregistrate și stocate într-un registru, ceea ce facilitează căutarea și executarea comenzilor în funcție de numele acestora. Aceasta oferă o abordare mai flexibilă și extensibilă în gestionarea comenzilor și permite adăugarea sau eliminarea ușoară a comenzilor noi.

# **2.4.2 Strategy**

Strategy este un model de design comportamental care vă permite să definiți o familie de algoritmi, să puneți fiecare dintre ei într-o clasă separată și să faceți obiectele lor interschimbabile.

De obicei, modelul de strategie stochează o referință la un anumit cod într-o structură de date și o preia. Acest lucru poate fi realizat prin mecanisme precum [pointerul funcției](https://en.wikipedia.org/wiki/Function_pointer) native , [funcția de primă clasă](https://en.wikipedia.org/wiki/First-class_function) , clasele sau instanțe de clasă în limbaje [de programare orientate pe obiecte](https://en.wikipedia.org/wiki/Object-oriented_programming) sau accesând stocarea internă de cod a implementării limbajului prin [reflectare](https://en.wikipedia.org/wiki/Reflection_(computer_science)). In figura 2.16 este reprezentata structura generala a șablonului strategy. In aceasta diagrama se disting patru clase principale:

* Context - menține o referință la una dintre strategiile concrete și comunică cu acest obiect numai prin interfața strategiei;
* Strategie - interfața care este comună tuturor strategiilor concrete. Declară o metodă pe care o folosește contextul pentru a executa o strategie;
* Concrete Strategies -  implementează diferite variații ale unui algoritm pe care îl folosește contextul;
* Client - creează un obiect de strategie specific și îl transmite contextului. Contextul expune un setter care le permite clienților să înlocuiască strategia asociată contextului în timpul execuției.

  
**Figura 2.16 –** Diagrama șablonului Strategy [7]

Avantajele și dezavantajele șablonului:

**Avantaje:**

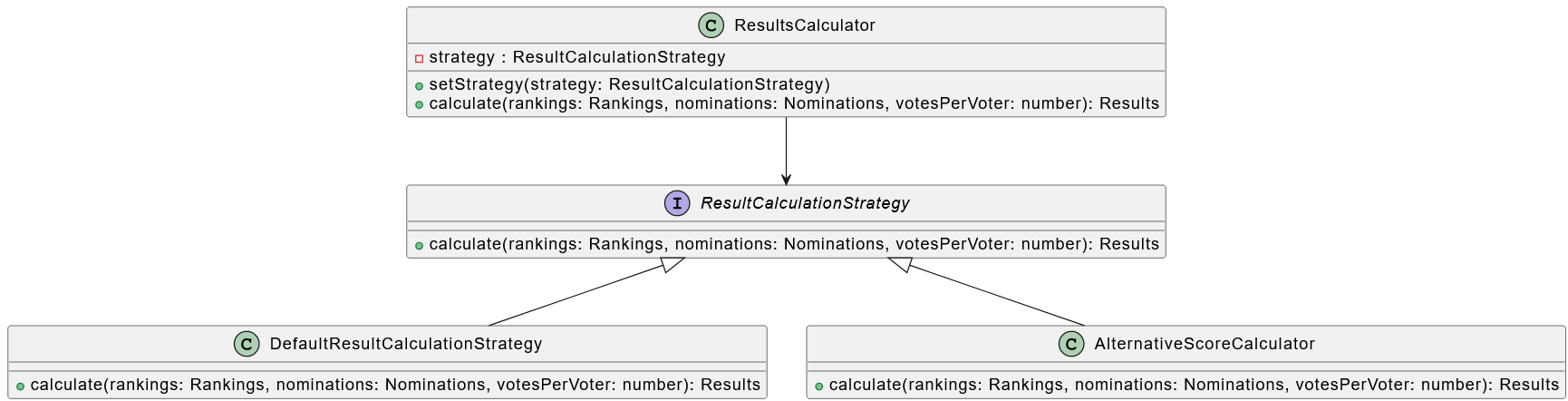
* Puteți schimba algoritmii utilizați în interiorul unui obiect în timpul execuției;
* Puteți izola detaliile de implementare ale unui algoritm din codul care îl folosește;
* Puteți înlocui moștenirea cu compoziția;
* *Principiul deschis/închis* . Puteți introduce noi strategii fără a fi nevoie să schimbați contextul.

**Dezavantaje:**

* Dacă aveți doar câțiva algoritmi și aceștia se schimbă rar, nu există niciun motiv real pentru a complica prea mult programul cu noi clase și interfețe care vin împreună cu modelul;
* Clienții trebuie să fie conștienți de diferențele dintre strategii pentru a putea alege una adecvată;
* Multe limbaje de programare moderne au suport de tip funcțional care vă permite să implementați diferite versiuni ale unui algoritm într-un set de funcții anonime. Apoi ați putea folosi aceste funcții exact așa cum ați fi folosit obiectele de strategie, dar fără a vă umfla codul cu clase și interfețe suplimentare.

# **2.4.2.1 Implementarea șablonului în aplicație**

In figura 2.17, este reprezentata diagrama UML a șablonului strategy pentru aplicația creata pentru acest raport.



**Figura 2.17 –** Diagrama șablonului Strategy pentru aplicația de votare

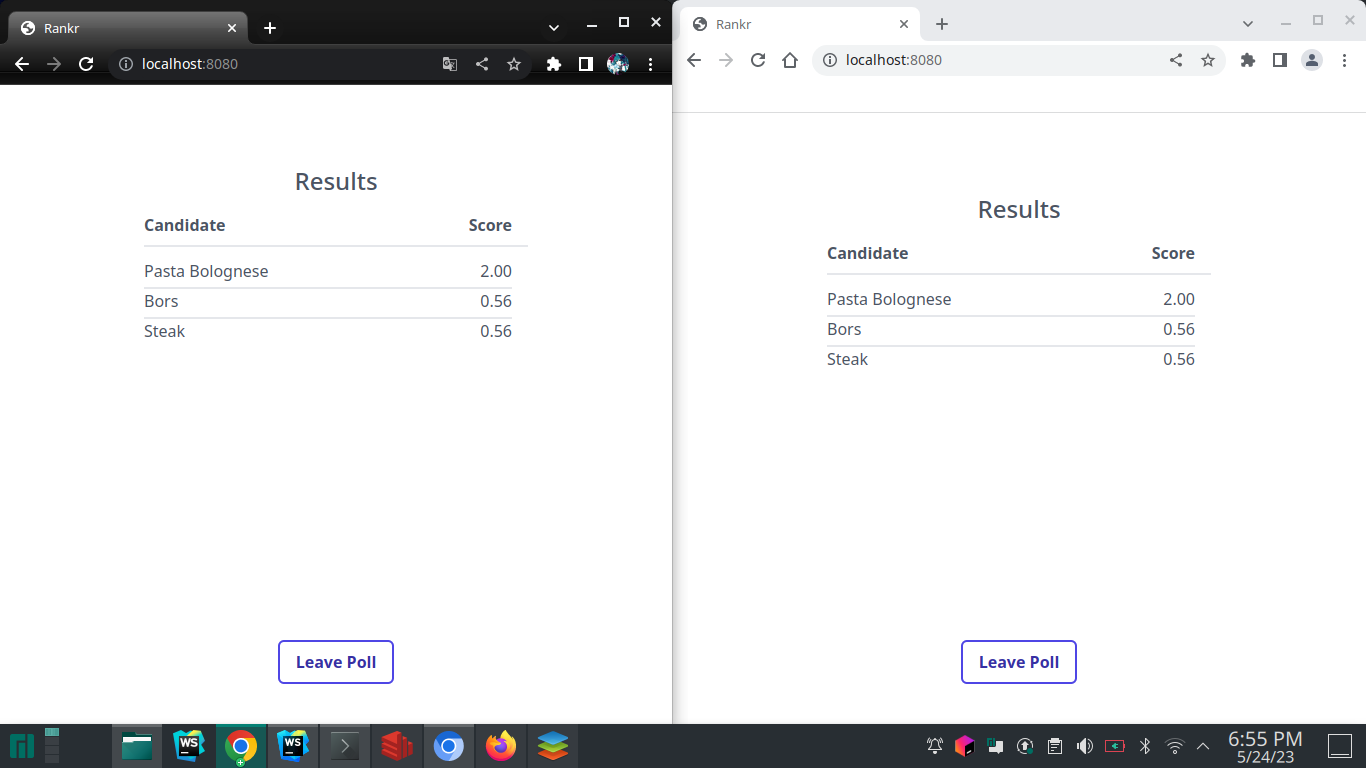
Implementarea șablonului strategy în aplicație se realizează prin utilizarea clasei ResultsCalculator și a claselor derivate DefaultResultCalculationStrategy și AlternativeScoreCalculator. Aceste clase reprezintă strategii diferite de calcul al rezultatelor în funcție de criterii specifice.

Clasa ResultsCalculator este clasa de context care utilizează o strategie specifică pentru a calcula rezultatele. Aceasta conține o referință către o implementare a interfeței ResultCalculationStrategy, care este setată prin intermediul constructorului sau metodei setStrategy.

Interfața ResultCalculationStrategy definește metoda calculate care primește clasamentele (rankings), nominalizările (nominations) și numărul de voturi per votant (votesPerVoter). Fiecare implementare a interfeței (DefaultResultCalculationStrategy și AlternativeScoreCalculator) are propria sa logică pentru calculul rezultatelor. Acestea parcurg clasamentele și calculează scorurile pentru fiecare nominalizare în funcție de strategia specifică.

Metoda calculate din clasa ResultsCalculator utilizează strategia curentă pentru a calcula rezultatele, apelând metoda calculate a strategiei cu parametrii corespunzători.

Această abordare permite schimbarea strategiei de calcul al rezultatelor în timpul rulării aplicației fără a modifica codul existent. De asemenea, facilitează extensibilitatea prin adăugarea de noi strategii de calcul în viitor.



**Figura 2.18 –** Afișarea rezultatelor

Clasele DefaultResultCalculationStrategy și AlternativeScoreCalculator implementează interfața ResultCalculationStrategy și definește metoda calculate conform logicii specifice fiecărei strategii de calcul a scorului.

Clasa ResultsCalculator reprezintă contextul și utilizează obiecte ale claselor DefaultResultCalculationStrategy și AlternativeScoreCalculator ca instanțe ale strategiilor de calcul a scorului. La construirea unui obiect ResultsCalculator, se poate furniza o strategie specifică sau se utilizează implicit DefaultResultCalculationStrategy.

Metoda setStrategy permite schimbarea strategiei utilizate în timpul execuției. Metoda calculate primește clasamentele, informațiile despre nominalizări și numărul de voturi per votant și apelează metoda calculate a strategiei curente pentru a obține rezultatele finale.

# **2.4.2.2 Necesitatea implementării șablonului**

Utilizarea șablonului strategy în aplicație este justificată de nevoia de a permite o flexibilitate și modularitate sporite în calculul scorurilor. Alegerea acestui șablon în detrimentul altora sau absența unui șablon este determinată de câteva motive cheie.

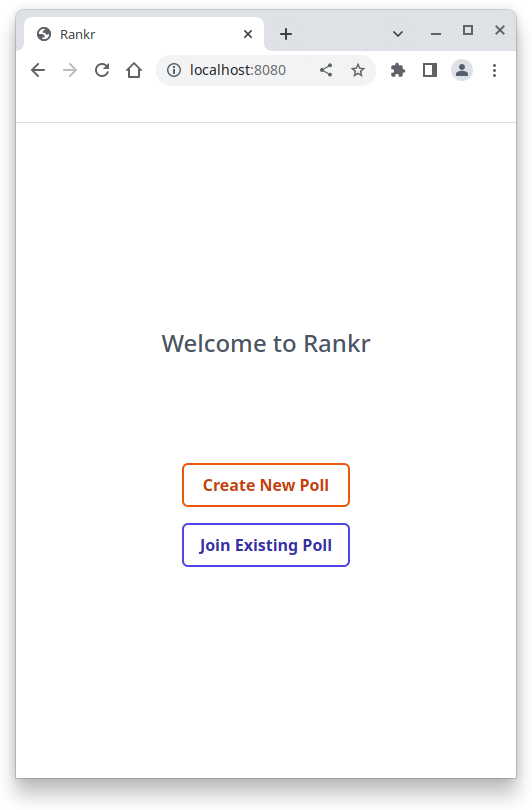
Pentru ca în aplicație există două clase derivate, DefaultScoreCalculator și AlternativeScoreCalculator, care implementează două strategii diferite de calcul al scorurilor, utilizarea șablonului strategy este ideala. Aceasta permite aplicației să se adapteze la diferite criterii și logici de calcul, în funcție de nevoile specifice.

În al doilea rând, șablonul strategy asigură o separare clară între algoritmul de calcul al scorurilor și codul care utilizează rezultatele. Clasa abstractă ScoreCalculator definește interfața comună și metoda abstractă calculateScore, care reprezintă operația de calcul specifică strategiei. Aceasta permite obținerea unei implementări modulare, unde codul care utilizează rezultatele, reprezentat de clasa ResultsCalculatorStrategyMethod, nu este afectat de schimbările în interiorul strategiilor de calcul. Astfel, se promovează principiul deschis-închis, permițând adăugarea și modificarea ușoară a strategiilor de calcul fără a afecta restul aplicației.

În plus, prin utilizarea șablonului strategy, se facilitează reutilizarea și extensibilitatea codului. Dacă dorim să adăugăm o nouă strategie de calcul a scorurilor, putem crea o nouă clasă derivată din ScoreCalculator și implementa metoda calculateScore conform logicii specifice acelei strategii. Aceasta ne permite să adăugăm noi strategii fără a modifica codul existent și să oferim o varietate de opțiuni pentru calculul scorurilor în diferite contexte.

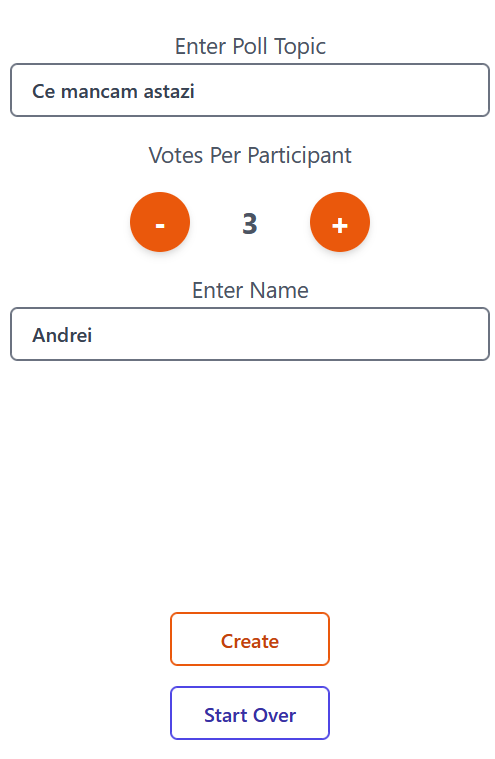
# **3. Implementarea aplicației**

In figura 3.1, este afișata pagina principala, care utilizatorul o vede în momentul când deschide aplicația. Pe aceasta pagina sunt doua butoane, prima care creează poll-ul, iar apăsând pe a doua, utilizatorul poate sa se alăture la un poll deja creat.



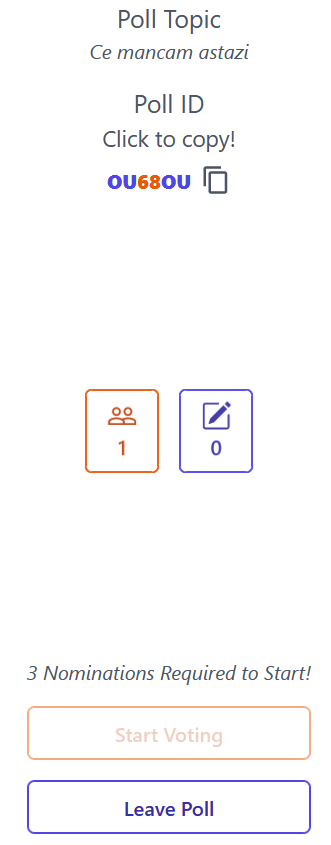
**Figura 3.1 –** Pagina de pornire

Daca utilizatorul decide sa creeze un poll nou, atunci el este redirecționat la următoarea pagina, afișata în figura 3.2. Primul lucru la crearea unui sondaj, el trebuie sa introducă tema sondajului, ulterior el decide cite voturi vor fi per participant, numărul minim fiind 1, iar maxim 5 și ultimul pas este doar sa-și introducă numele. Daca utilizatorul nu mai vrea sa facă modificări el poate apasă pe butonul „Create”. Daca el s-a răzgândit și vrea sa se alăture la un poll deja creat, atunci poate sa aleagă butonul “Start Over”, care îl va întoarce pe pagina principala.

****

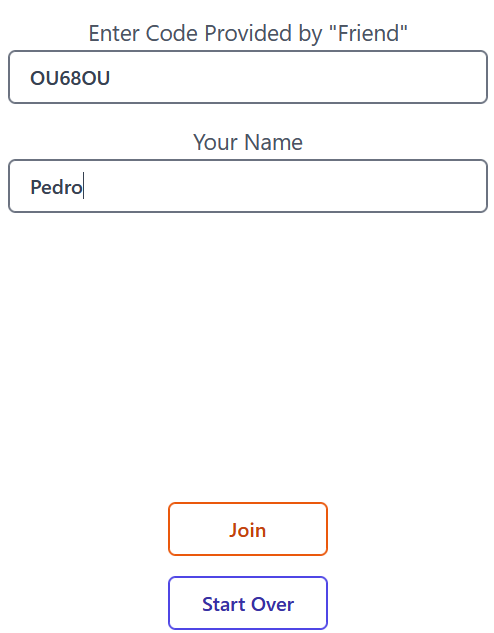
**Figura 3.2 –** Crearea sondajului

Odată ce utilizatorul creează sondajul el este redirecționat la următoarea pagina, afișata în figura 3.3 de unde poate lua id-ul poll-ului, care ulterior îl va transmite persoanelor pe care el vrea sa-i adauge la sondaj.



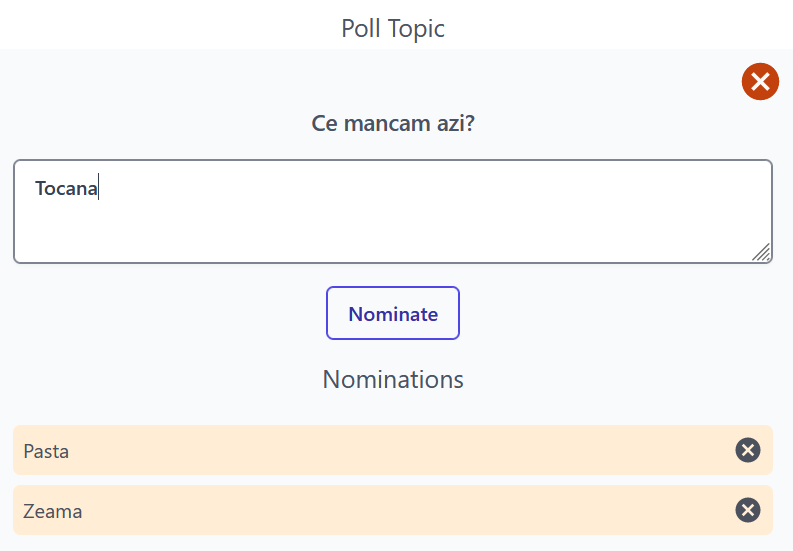
**Figura 3.3 –** Pagina pentru începerea votului

In figura 3.4, este arătata pagina care apare în cazul în care utilizatorul decide sa se alăture la un sondaj deja existent. In cazul acesta, utilizatorul introduce ID-ul sondajului, primit de la administrator (persoana care a creat acest sondaj), și ultimul pas este de a-și introduce numele.



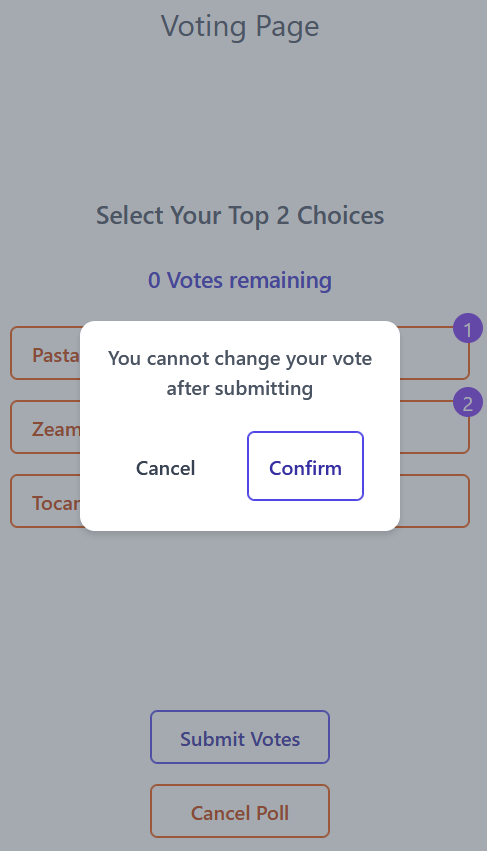
**Figura 3.4 –** Alăturarea la poll

Pentru ca adminul sa poată sa înceapă sondajul, în el trebuie sa fie introduse cel puțin doua nominalizări, cum este arătat și în figura 3.5.



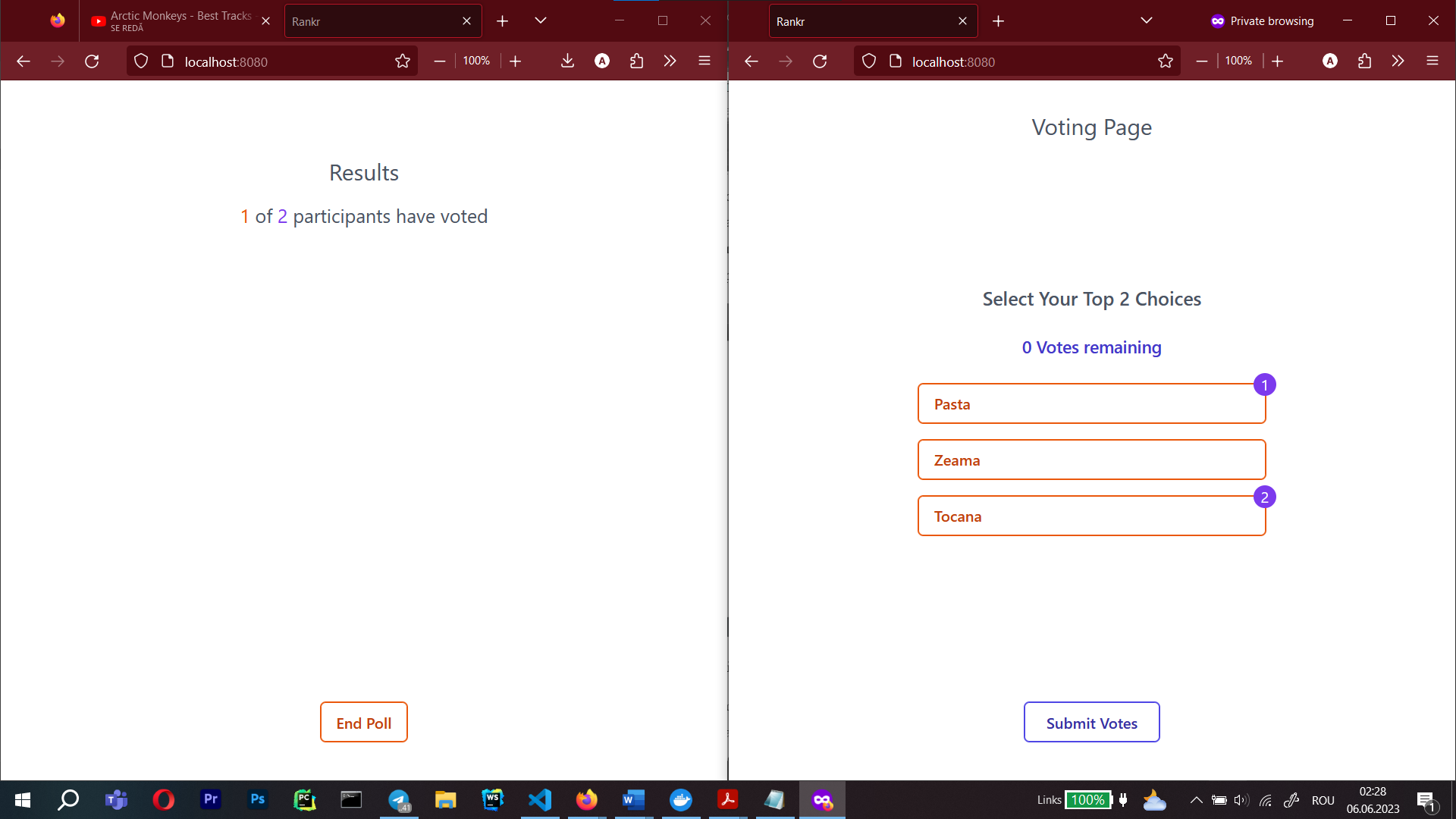
**Figura 3.5 –** Adăugarea nominațiilor

Dupa ce se incepe sondajul și utilizatorul votează pentru nominațiile care le vrea, el este întrebat daca este sigura ca vrea sa iasă din sondaj, cum este arătat și în figura 3.6.



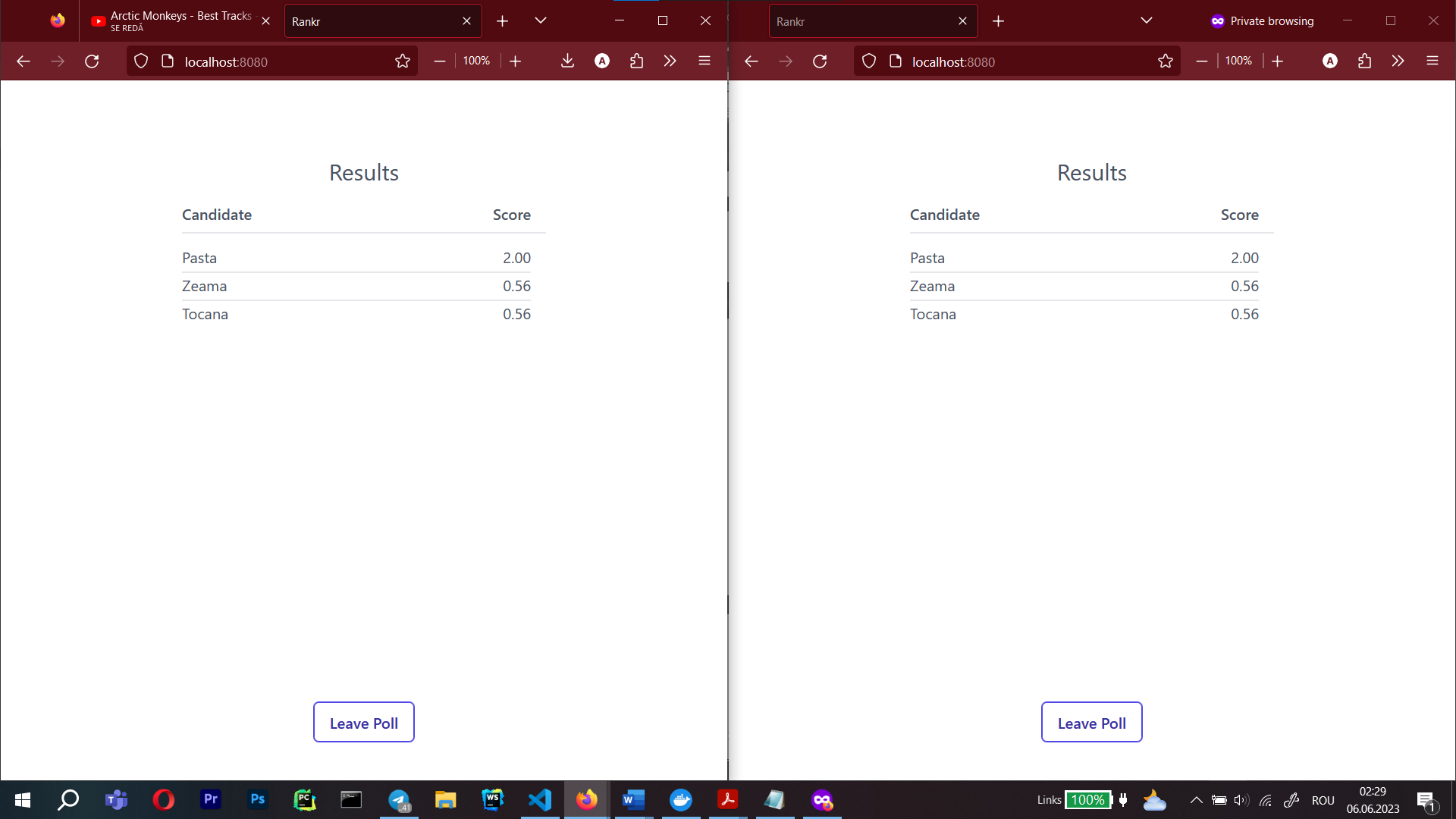
**Figura 3.6 –** Confirmarea alegerilor

In figura 3.7, este afișata statutul, în timp real, a doi utilizatori, care stau pe doua dispozitive diferite. In primul caz, utilizatorul deja și-a făcut decizia și a votat pentru nominațiile dorite și ii este afișat câți utilizatori din cei care participa la sondaj au votat. Pentru a putea fi văzute rezultatele sondajului trebuie ca toți participanții sa voteze. In al doilea caz, este afișat procesul de votare. Aici, utilizatorul poate sa vadă nominațiile, cite voturi poate sa facă și cite voturi i-au mai rămas.



**Figura 3.7 –** Afișarea în timp real a statutului utilizatorilor

In figura 3.8, sunt afișate rezultatele, în timp real, a doi utilizatori, care stau pe doua dispozitive diferite. Din aceasta imagine se distinge ca rezultatele sunt identice, utilizatorul văzând care au fost nominațiile, și care nominație a câștigat. De asemena la rezultat nominațiile se afișează în ordine descrescătoare în conformitate cu scorul obținut.



**Figura 3.7 –** Afișarea în timp real a rezultatelor

**Concluzie**

Analizând aplicarea diverselor șabloane de proiectare în cadrul unei aplicații de votare, s-a observat că acestea au adus beneficii semnificative sistemului. S-a examinat șabloanele creationale, structurale și comportamentale și s-a evidențiat modul în care acestea au fost implementate și au contribuit la îmbunătățirea funcționalității și flexibilității aplicației de votare.

Utilizarea șabloanelor de proiectare a dovedit utilitatea lor în dezvoltarea aplicației de votare. De exemplu, prin aplicarea șabloanelor Factory Method și Singleton, s-a reușit să se creeze obiecte într-un mod flexibil și modular, ceea ce a facilitat adaptabilitatea și extensibilitatea sistemului. Această abordare a devenit esențială în fața cerințelor și schimbărilor în evoluție ale procesului de votare.

Șabloanele structurale, precum Adapter și Decorator, au fost utilizate pentru a asigura o interacțiune fluentă între diferitele componente ale aplicației de votare și pentru a permite integrarea cu alte sisteme existente. Această integrare a îmbunătățit flexibilitatea și reutilizabilitatea codului, permițând aplicației de votare să funcționeze într-un mediu complex și interconectat.

În ceea ce privește șabloanele comportamentale, precum Command și Strategy, acestea au jucat un rol crucial în gestionarea fluxului de lucru al aplicației de votare și în adaptarea și schimbarea comportamentului în funcție de nevoile utilizatorilor. Implementarea acestor șabloane a contribuit la dezvoltarea unei aplicații de votare robuste, ușor de întreținut și extinsă în viitor.

În concluzie, prin utilizarea șabloanelor de proiectare în cadrul aplicației de votare, s-au obținut multiple avantaje, precum îmbunătățirea modularității, flexibilității și posibilitatea de reutilizare și extensibilitate a sistemului. Implementarea acestor șabloane a facilitat organizarea și gestionarea eficientă a codului, permițând adaptarea la schimbările și cerințele noi. În urma aplicării acestor șabloane, aplicația de votare a devenit mai robustă, eficientă și adaptabilă la nevoile utilizatorilor și evoluțiile în domeniul votării.

# **Bibliografie**

1. Wikipedia, Creational pattern, accesat 14.04.2023 [Resursa Electronica] – Regim de acces: <https://en.wikipedia.org/wiki/Creational_pattern>
2. Alexander Shvets, Factory Method, accesat 14.04.2023 [Resursa Electronica] – Regim de acces: <https://refactoring.guru/design-patterns/factory-method>
3. [Alexander Shvets, Adapter, accesat 30.04.2023 [Resursa Electronica] – Regim de acces: https://refactoring.guru/design-patterns/adapter](https://refactoring.guru/design-patterns/adapter)
4. Alexander Shvets, Decorator, accesat 30.04.2023 [Resursa Electronica] – Regim de acces: <https://refactoring.guru/design-patterns/decorator>
5. Alexander Shvets, Chain of Responsability, accesat 15.05.2023 [Resursa Electronica] – Regim de acces: <https://refactoring.guru/design-patterns/chain-of-responsibility>
6. Alexander Shvets, Command, accesat 15.05.2023 [Resursa Electronica] – Regim de acces: <https://refactoring.guru/design-patterns/command>
7. Alexander Shvets, Strategy, accesat 15.05.2023 [Resursa Electronica] – Regim de acces: https://refactoring.guru/design-patterns/strategy
8. SourceMaking, Creational pattern, accesat 14.04.2023 [Resursa Electronica] – Regim de acces: <https://sourcemaking.com/design_patterns/creational_patterns>
9. Șabloane de Proiectare, accesat 14.04.2023 [Resursa Electronica] – Regim de acces: <http://inf.ucv.ro/~mihaiug/courses/poo/slides/Curs%2012%20-%20Sabloane%20de%20Proiectare.pdf>
10. [Andrei Jechiu](https://www.slideshare.net/andreijechiu?utm_campaign=profiletracking&utm_medium=sssite&utm_source=ssslideview), Șabloane de Proiectare, accesat 14.04.2023 [Resursa Electronica] – Regim de acces: <https://www.slideshare.net/andreijechiu/design-patterns-19909962>
11. Baeldung, Structural Pattern in core Java, accesat 30.04.2023 [Resursa Electronica] – Regim de acces: [https://www.baeldung.com/java-core-structural-patterns#:~:text=Structural%20design%20patterns%20are%20those,they%20become%20repeatable%20as%20solutions.](https://www.baeldung.com/java-core-structural-patterns#:~:text=Structural design patterns are those,they become repeatable as solutions.)
12. Wikipedia, Structural pattern, accesat 30.04.2023 [Resursa Electronica] – Regim de acces: <https://en.wikipedia.org/wiki/Structural_pattern>
13. Wikipedia, Adapter pattern, accesat 30.04.2023 [Resursa Electronica] – Regim de acces: <https://en.wikipedia.org/wiki/Adapter_pattern>
14. Wikipedia, Decorator pattern, accesat 30.04.2023 [Resursa Electronica] – Regim de acces: <https://en.wikipedia.org/wiki/Decorator_pattern>
15. SourceMaking, Creational pattern, accesat 15.05.2023 [Resursa Electronica] – Regim de acces: <https://refactoring.guru/design-patterns/singleton>

Anexa A

export class WsTypeExceptionFactory {

static createException(

type: WsExceptionType,

message: string | unknown,

): WsTypeException {

switch (type) {

case 'BadRequest':

return new WsBadRequestException(message);

case 'Unauthorized':

return new WsUnauthorizedException(message);

case 'Unknown':

return new WsUnknownException(message);

default:

throw new Error(`Invalid WsExceptionType: ${type}`);

}

}

}

Anexa B

export class RedisModule {

private static redisClient: Redis;

static async registerAsync({

useFactory,

imports,

inject,

}: RedisAsyncModuleOptions): Promise<DynamicModule> {

if (!RedisModule.redisClient) {

const redisProvider = {

provide: IORedisKey,

useFactory: async (...args) => {

const { connectionOptions, onClientReady } = await useFactory(

...args,

);

RedisModule.redisClient = new IORedis(connectionOptions);

onClientReady(RedisModule.redisClient);

return RedisModule.redisClient;

},

inject,

};

return {

module: RedisModule,

imports,

providers: [redisProvider],

exports: [redisProvider],

};

} else {

// If the Redis client instance already exists, return the module without creating a new instance

return {

module: RedisModule,

imports,

exports: [IORedisKey],

};

}

}

}

Anexa C

export class SocketIOAdapter extends IoAdapter {

private readonly logger = new Logger(SocketIOAdapter.name);

constructor(

private app: INestApplicationContext,

private configService: ConfigService,

) {

super(app);

}

createIOServer(port: number, options?: ServerOptions) {

const clientPort = parseInt(this.configService.get('CLIENT\_PORT'));

const cors = {

origin: [

`http://localhost:${clientPort}`,

new RegExp(`/^http:\/\/192\.168\.1\.([1-9]|[1-9]\d):${clientPort}$/`),

],

};

this.logger.log('Configuring SocketIO server with custom CORS options', {

cors,

});

const optionsWithCORS: ServerOptions = {

...options,

cors,

};

const jwtService = this.app.get(JwtService);

const server: Server = super.createIOServer(port, optionsWithCORS);

server.of('polls').use(createTokenMiddleware(jwtService, this.logger));

return server;

}

}

const createTokenMiddleware =

(jwtService: JwtService, logger: Logger) =>

(socket: SocketWithAuth, next) => {

// for Postman testing support, fallback to token header

const token =

socket.handshake.auth.token || socket.handshake.headers['token'];

logger.debug(`Validating auth token before connection: ${token}`);

try {

const payload = jwtService.verify(token);

socket.userID = payload.sub;

socket.pollID = payload.pollID;

socket.name = payload.name;

next();

} catch {

next(new Error('FORBIDDEN'));

}

};

export interface SocketAdapter {

createServer(port: number, options?: ServerOptions): Server;

}

export class SocketIOAdapterPattern implements SocketAdapter {

private readonly socketIOAdapter: SocketIOAdapter;

constructor(app: INestApplicationContext, configService: ConfigService) {

this.socketIOAdapter = new SocketIOAdapter(app, configService);

}

createServer(port: number, options?: ServerOptions): Server {

return this.socketIOAdapter.createIOServer(port, options);

}

}

Anexa D

export class GatewayAdminGuard implements CanActivate {

private readonly logger = new Logger(GatewayAdminGuard.name);

constructor(

private readonly pollsService: PollsService,

private readonly jwtService: JwtService,

) {}

async canActivate(context: ExecutionContext): Promise<boolean> {

const socket: SocketWithAuth = context.switchToWs().getClient();

const token =

socket.handshake.auth.token || socket.handshake.headers['token'];

if (!token) {

this.logger.error('No authorization token provided');

throw new WsUnauthorizedException('No token provided');

}

try {

const payload = this.jwtService.verify<AuthPayload & { sub: string }>(

token,

);

this.logger.debug(`Validating admin using token payload`, payload);

const { sub, pollID } = payload;

const poll = await this.pollsService.getPoll(pollID);

if (sub !== poll.adminID) {

throw new WsUnauthorizedException('Admin privileges required');

}

return true;

} catch {

throw new WsUnauthorizedException('Admin privileges required');

}

}

}

export class LoggingGatewayAdminGuardDecorator implements CanActivate {

private readonly logger = new Logger(LoggingGatewayAdminGuardDecorator.name);

constructor(private readonly guard: GatewayAdminGuard) {}

async canActivate(context: ExecutionContext): Promise<boolean> {

this.logger.debug('Before executing GatewayAdminGuard.');

const result = await this.guard.canActivate(context);

this.logger.debug('After executing GatewayAdminGuard.');

return result;

}

}

Anexa E

async removeNomination(pollID: string, nominationID: string): Promise<Poll> {

return this.pollsRepository.removeNomination(pollID, nominationID);

}

async startPoll(pollID: string): Promise<Poll> {

return this.pollsRepository.startPoll(pollID);

}

async submitRankings(rankingsData: SubmitRankingsFields): Promise<Poll> {

const hasPollStarted = this.pollsRepository.getPoll(rankingsData.pollID);

if (!hasPollStarted) {

throw new BadRequestException(

'Participants cannot rank until the poll has started.',

);

}

return this.pollsRepository.addParticipantRankings(rankingsData);

}

async computeResults(pollID: string): Promise<Poll> {

const poll = await this.pollsRepository.getPoll(pollID);

const resultsCalculator = new ResultsCalculator();

const results = resultsCalculator.calculate(

poll.rankings,

poll.nominations,

poll.votesPerVoter,

);

return this.pollsRepository.addResults(pollID, results);

}

async cancelPoll(pollID: string): Promise<void> {

await this.pollsRepository.deletePoll(pollID);

}

async createPollCommand(fields: CreatePollFields): Promise<any> {

const command = new CreatePollCommand(this, fields);

return this.executeCommand(command);

}

async joinPollCommand(fields: JoinPollFields): Promise<any> {

const command = new JoinPollCommand(this, fields);

return this.executeCommand(command);

}

async rejoinPollCommand(fields: RejoinPollFields): Promise<any> {

const command = new RejoinPollCommand(this, fields);

return this.executeCommand(command);

}

}

Anexa F

class DefaultResultCalculationStrategy implements ResultCalculationStrategy {

calculate(

rankings: Rankings,

nominations: Nominations,

votesPerVoter: number,

): Results {

const scores: { [nominationID: string]: number } = {};

Object.values(rankings).forEach((userRankings) => {

userRankings.forEach((nominationID, n) => {

const voteValue = Math.pow(

(votesPerVoter - 0.5 \* n) / votesPerVoter,

n + 1,

);

scores[nominationID] = (scores[nominationID] ?? 0) + voteValue;

});

});

const results = Object.entries(scores).map(([nominationID, score]) => ({

nominationID,

nominationText: nominations[nominationID].text,

score,

}));

results.sort((res1, res2) => res2.score - res1.score);

return results;

}

}

// Alternative score calculator strategy

class AlternativeScoreCalculator implements ResultCalculationStrategy {

calculate(

rankings: Rankings,

nominations: Nominations,

votesPerVoter: number,

): Results {

const scores: { [nominationID: string]: number } = {};

Object.values(rankings).forEach((userRankings) => {

userRankings.forEach((nominationID, n) => {

const voteValue = (votesPerVoter - n) / votesPerVoter;

scores[nominationID] = (scores[nominationID] ?? 0) + voteValue;

});

});

const results = Object.entries(scores).map(([nominationID, score]) => ({

nominationID,

nominationText: nominations[nominationID].text,

score,

}));

results.sort((res1, res2) => res2.score - res1.score);

return results;

}

}

// Context class that uses the strategy

export default class ResultsCalculator {

private strategy: ResultCalculationStrategy;

constructor(

strategy: ResultCalculationStrategy = new DefaultResultCalculationStrategy(),

) {

this.strategy = strategy;

}

setStrategy(strategy: ResultCalculationStrategy) {

this.strategy = strategy;

}

calculate(

rankings: Rankings,

nominations: Nominations,

votesPerVoter: number,

): Results {

return this.strategy.calculate(rankings, nominations, votesPerVoter);

}

}