



Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №4

Технології розроблення програмного забезпечення

Тема: «Особиста бухгалтерія»

Виконала:

Студентка групи ІА-34

Дригант А.С

Перевірив:

Мягкий Михайло Юрійович

Тема: Вступ до паттернів проектування

Мета: Вивчити структуру шаблонів «Singleton», «Iterator», «Proxy», «State», «Strategy» та навчитися застосовувати їх в реалізації програмної системи

Тема роботи:

27. Особиста бухгалтерія (state, prototype, decorator, bridge, flyweight, SOA)
Програма повинна бути наочним засобом для ведення особистих фінансів: витрат і прибутку; з можливістю встановлення періодичних витрат / прибутку (зарплата і орендна плата); введення сканованих чеків з відповідними статтями витрат; побудова статистики; експорт/імпорт в Excel, реляційні джерела даних; різні рахунки; ведення єдиного фонду на всі рахунки (всією сім'єю) – на особливі потреби (ремонт, автомобіль, відпустка); можливість введення вкладів / кредитів для контролю банківських рахунків (звірка нарахованих відсотків з необхідними і т.д.).

Теоритичні відомості

Патерни проектування – це формалізовані рішення типових задач проектування. Вони забезпечують зрозумілу структуру системи, спрощують супровід і роблять систему гнучкою до змін. Також вони слугують єдиним словником для комунікації між розробниками.

Singleton (Одинак)

- Призначення: гарантує існування лише одного екземпляра класу та глобальну точку доступу до нього.
- Приклади: файл налаштувань, єдиний сеанс зв'язку.
- Переваги: гарантує єдиний екземпляр, проста реалізація.
- Недоліки: вважається «анти-шаблоном», оскільки схожий на глобальні змінні, складно тестувати, може маскувати поганий дизайн.

Iterator (Ітератор)

- Призначення: дозволяє проходити по елементах колекції без знання її

внутрішньої структури.

- Ідея: винести логіку обходу в окремий клас.
- Основні методи: First(), Next(), IsDone, CurrentItem.
- Переваги: різні способи обходу (вперед, назад, вибірково).
- Недоліки: не завжди виправданий – якщо достатньо простого циклу.

Proxy (Проксі / Замісник)

- Призначення: об'єкт-заступник, що контролює доступ або додає логіку перед викликом реального об'єкта.
- Приклади: заглушка для картинки під час завантаження, оптимізація запитів до зовнішніх сервісів (DocuSign).
- Переваги: можна додати проміжний рівень без зміни клієнтського коду.
- Недоліки: може знизити швидкість, ризик неправильних відповідей.

State (Стан)

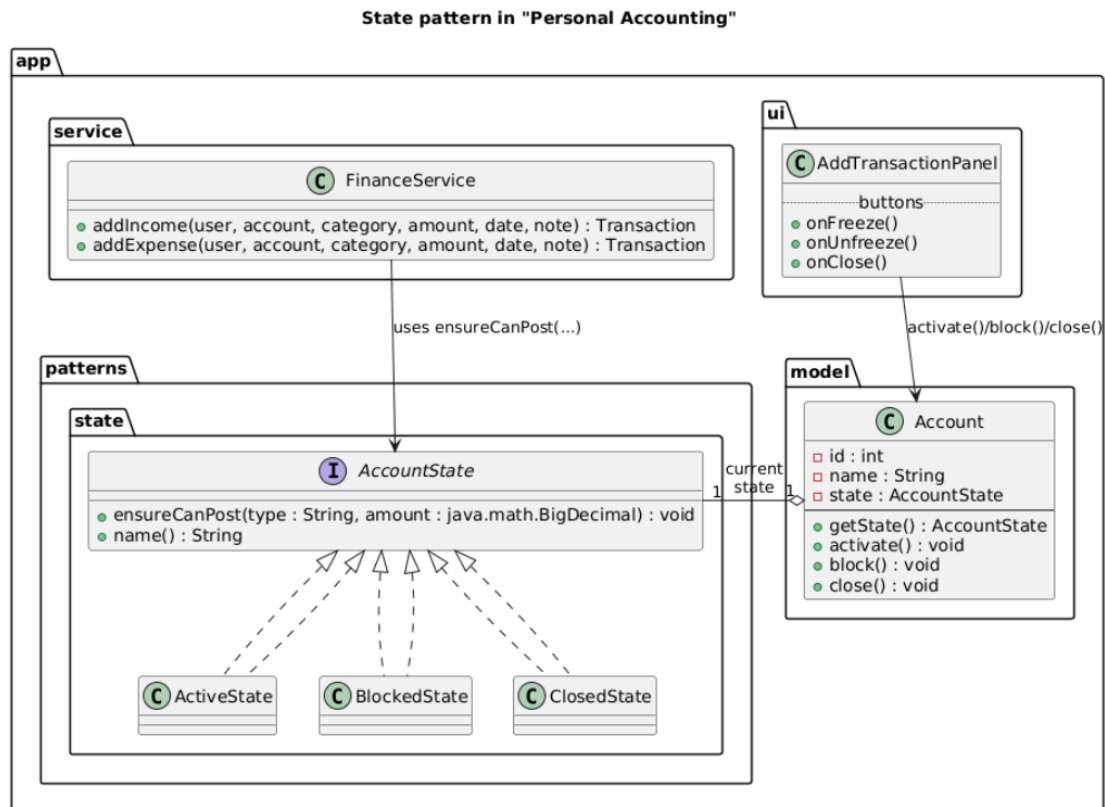
- Призначення: зміна поведінки об'єкта при зміні його внутрішнього стану.
- Приклади: банківська картка (Visa Electron, Classic, Platinum), Listener із станами Initializing, Open, Closing.
- Ідея: винести логіку станів у класи ConcreteState, між якими переходить Context.
- Переваги: гнучкість, легке додавання нових станів.
- Недоліки: ускладнює контекст через механізм переключення.

Strategy (Стратегія)

- Призначення: дозволяє замінювати один алгоритм іншим під час виконання.
- Приклади: різні алгоритми сортування, різні способи дістатися до роботи (авто, метро, пішки).
- Ідея: алгоритми оформлені як окремі стратегії, які можна підключати до одного контексту.
- Переваги: гнучкість, зменшення умовних операторів, простота супроводу.
- Недоліки: надмірна складність, якщо алгоритмів мало.

Хід роботи

Діаграма класів



На діаграмі я показала, як у моєму проєкті реалізовано та використано патерн State для керування поведінкою рахунку залежно від його стану.

Рівень `patterns.state` (ядро патерна).

Я визначила інтерфейс `AccountState` з двома методами:

`ensureCanPost(type: String, amount: BigDecimal): void` - централізована перевірка, чи дозволено проводити операцію певного типу (дохід/витрата) на поточному стані рахунку;

`name(): String` - сервісний метод для відображення поточного стану.

Конкретні стани - `ActiveState`, `BlockedState`, `ClosedState` - інкапсулюють різну поведінку:

- `ActiveState`: дозволяє всі операції (доходи/витрати).
- `BlockedState`: блокує витрати (або інші заборони за правилами).
- `ClosedState`: забороняє будь-які операції, рахунок лише для перегляду.

Модель model.Account (контекст патерна).

Клас Account зберігає посилання на поточний стан (state: AccountState) і делегує йому перевірки перед виконанням бізнес-операцій. Також тут зібрані керуючі дії над станом:

- activate() - переведення у робочий стан (якщо дозволяють правила).
- block() - блокування рахунку (наприклад, у разі виявлених ризиків).
- close() - закриття (тільки за умов, напр., баланс = 0).

Стрілка «current state» показує композицію: Account “тримає” один поточний стан і може його підміняти (перемикати).

Сервіс service.FinanceService (місце використання поведінки стану).

Операції addIncome(...) та addExpense(...) перед записом транзакції викликають ensureCanPost(...) у стані рахунку. Таким чином, бізнес-логіка повністю підконтрольна поточному стану: якщо рахунок BLOCKED або CLOSED, сервіс отримає виняток/заборону і транзакція не пройде. Це демонструє реальне “включення” патерна в робочий потік.

UI ui.AddTransactionPanel (керування переходами станів).

Події інтерфейсу - onFreeze(), onUnfreeze(), onClose() - викликають методи block()/activate()/close() у Account. Тобто UI не знає деталей станів, а тільки ініціює переходи; вся різниця поведінки живе в класах станів.

Сенс патерна на практиці.

Я уникнула великої кількості if/else у моделі та сервісі. Натомість різні правила доступності операцій винесені у дрібні, прості класи станів. Додавання нового стану (наприклад, ArchivedState) вимагатиме лише створити новий клас і налаштувати переходи - існуючий код майже не змінюється.

Переходи

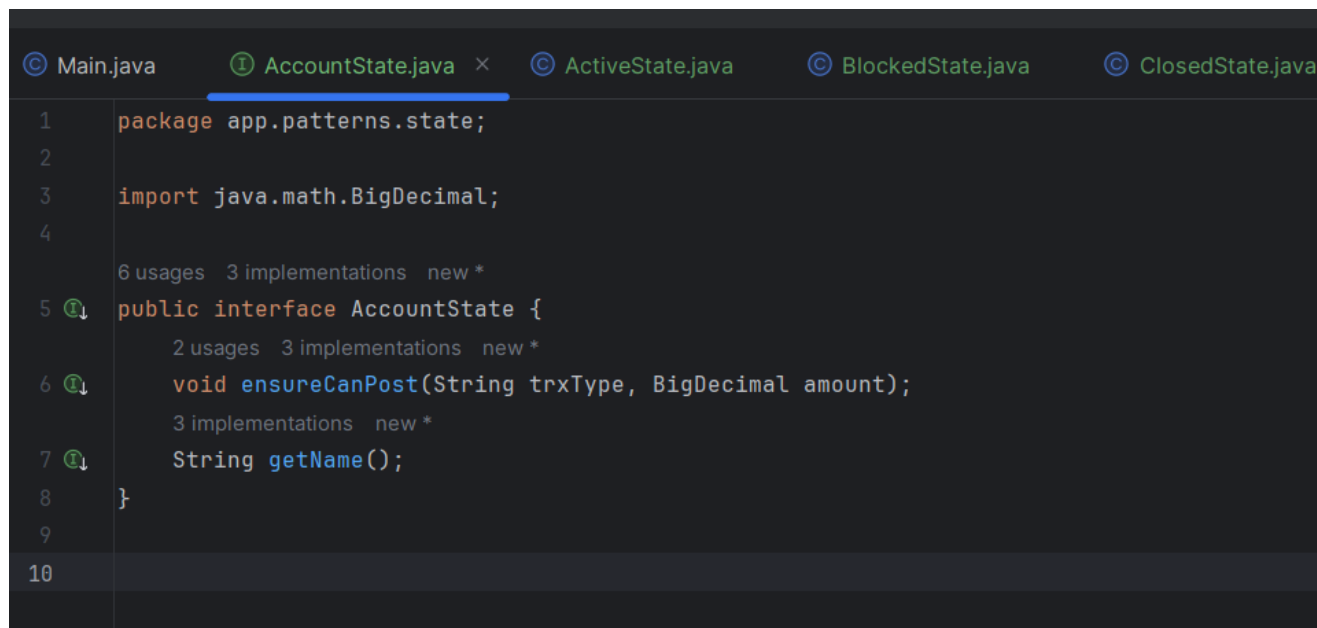
Типові правила:

ACTIVE → BLOCKED (ризики/заморозка), BLOCKED → ACTIVE (усунено причину),

ACTIVE/BLOCKED → CLOSED (лише якщо баланс = 0), CLOSED - кінцевий стан без повернення.

У демо я можу перемикаєти стани вручну (для показу поведінки), але у продакшн-сценарії їх мають змінювати бізнес-події (наприклад, результат верифікації чи закриття з нульовим балансом).

1) Інтерфейс стану



```
1 package app.patterns.state;
2
3 import java.math.BigDecimal;
4
5 6 usages 3 implementations new *
6 public interface AccountState {
7     2 usages 3 implementations new *
8     void ensureCanPost(String trxType, BigDecimal amount);
9     3 implementations new *
10    String getName();
11 }
```

У цьому файлі реалізовано інтерфейс AccountState, який є центральною частиною шаблону State. Він визначає спільний контракт для всіх можливих станів рахунку.

- Метод `ensureCanPost(String trxType, BigDecimal amount)` – використовується для перевірки, чи можна виконати операцію з певним типом транзакції (наприклад, дохід чи витрата) при поточному стані рахунку. Кожен конкретний стан (ActiveState, BlockedState, ClosedState) реалізує цей метод по-своєму, накладаючи власні правила.
- Метод `getName()` – повертає назву стану як рядок (наприклад, "ACTIVE", "BLOCKED", "CLOSED"). Це дозволяє зручно відображати стан рахунку в інтерфейсі користувача та логах.

Таким чином, інтерфейс інкапсулює змінну поведінку об'єкта «Рахунок» і робить можливим динамічне перемикання станів без використання складних умовних конструкцій у коді.

2) Конкретні стани

```
© Main.java  ⓘ AccountState.java  © ActiveState.java ×  © BlockedState.java

1  package app.patterns.state;
2
3  import java.math.BigDecimal;
4
5  2 usages new *
   public class ActiveState implements AccountState {
6      2 usages new *
7      @Override
8      public void ensureCanPost(String trxType, BigDecimal amount) {
9
10     new *
11     @Override
12     public String getName() {
13         return "ACTIVE";
14     }
15 }
16 }
```

Клас `ActiveState` реалізує інтерфейс `AccountState` і відповідає за поведінку рахунку в активному стані.

- У методі `ensureCanPost(...)` немає обмежень – дозволено виконувати всі операції (доходи і витрати).
- Метод `getName()` повертає назву стану – "ACTIVE".

Таким чином, цей стан є основним робочим режимом рахунку.

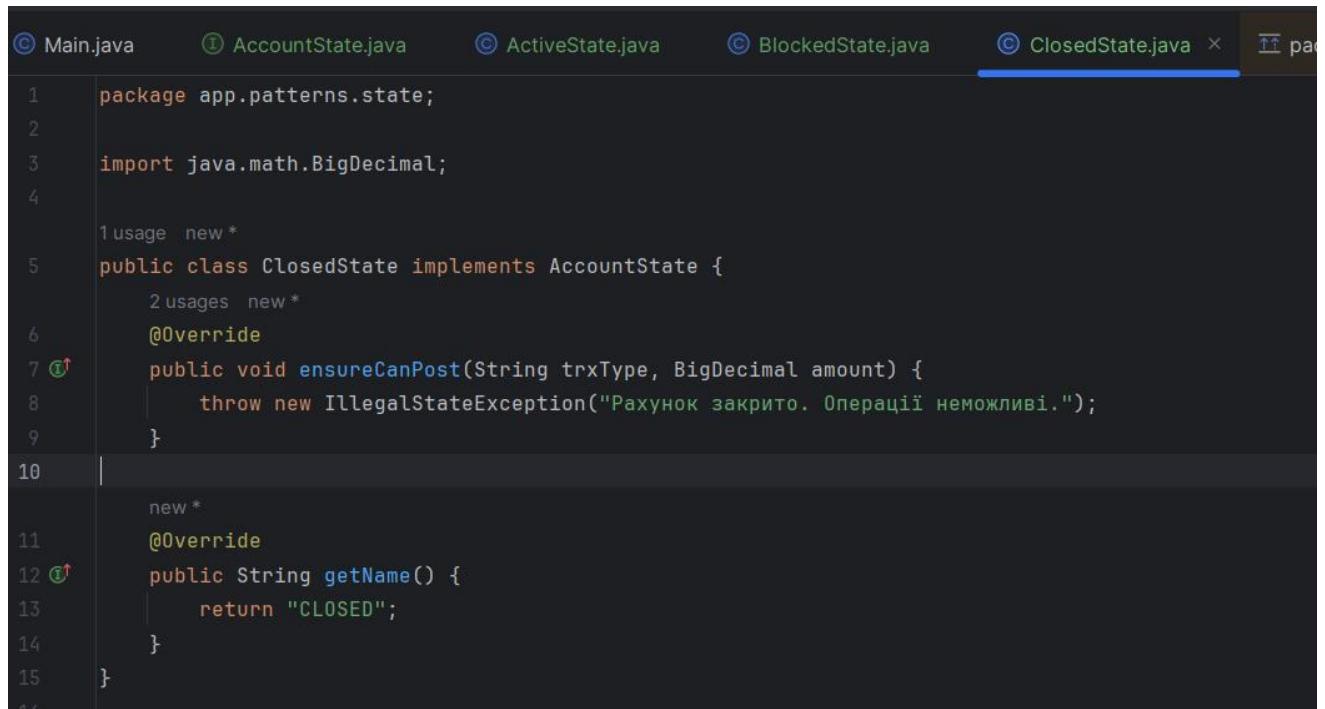
```
© Main.java  ⓘ AccountState.java  © ActiveState.java  © BlockedState.java ×  © ClosedState.java  ⓘ pack

1  package app.patterns.state;
2
3  import java.math.BigDecimal;
4
5  1 usage new *
   public class BlockedState implements AccountState {
6      2 usages new *
7      @Override
8      public void ensureCanPost(String trxType, BigDecimal amount) {
9          if ("EXPENSE".equalsIgnoreCase(trxType)) {
10             throw new IllegalStateException("Рахунок заблоковано для витрат");
11         }
12     }
13
14     new *
15     @Override
16     public String getName() {
17         return "BLOCKED";
18     }
19 }
20 }
```

Клас `BlockedState` реалізує поведінку рахунку, який заблокований.

- У методі `ensureCanPost(...)` дозволяється лише обмежений набір операцій: наприклад, забороняються витрати (EXPENSE). При спробі провести витрату генерується виняток `IllegalStateException` із повідомленням, що рахунок заблоковано.
- Метод `getName()` повертає "BLOCKED".

Цей стан демонструє, як обмежуються можливості рахунку під час блокування.



```

1 package app.patterns.state;
2
3 import java.math.BigDecimal;
4
5 1 usage new *
6 public class ClosedState implements AccountState {
7     2 usages new *
8     @Override
9     public void ensureCanPost(String trxType, BigDecimal amount) {
10         throw new IllegalStateException("Рахунок закрито. Операції неможливі.");
11     }
12
13     new *
14     @Override
15     public String getName() {
16         return "CLOSED";
17     }
18 }

```

Клас `ClosedState` реалізує поведінку закритого рахунку.

- У методі `ensureCanPost(...)` завжди генерується виняток `IllegalStateException` із повідомленням, що рахунок закритий і операції неможливі.
- Метод `getName()` повертає "CLOSED".

Цей стан повністю забороняє будь-які транзакції, залишаючи рахунок доступним лише для перегляду історії.

3) Модель Account із переходами станів

```

6 usages
private AccountState state = new ActiveState();

```



```

1 usage  new *
public void activate() { this.state = new ActiveState(); }

1 usage  new *
public void block() { this.state = new BlockedState(); }

1 usage  new *
public void close() { this.state = new ClosedState(); }

3 usages  new *
= public AccountState getState() { return state; }

```

Клас Account виступає контекстом у шаблоні State. Він містить поле state, яке зберігає поточний стан рахунку (ActiveState, BlockedState або ClosedState). Усі перевірки на можливість проведення операцій делегуються цьому об'єкту стану.

Ініціалізація:

При створенні об'єкта рахунку (private AccountState state = new ActiveState();) він автоматично перебуває у стані ACTIVE.

Методи переходів між станами:

- activate() - переводить рахунок у стан ActiveState.
- block() - змінює стан на BlockedState.
- close() - переводить рахунок у стан ClosedState.

Ці методи відповідають за зміну поведінки об'єкта динамічно, без використання умовних операторів if-else.

Метод getState() - дозволяє отримати поточний стан рахунку та відобразити його у користувацькому інтерфейсі або використати у логіці програми.

Таким чином, клас Account керує поточною поведінкою рахунку через поле state і дозволяє динамічно перемикає стани. Це і є основна ідея патерна State: інкапсуляція різних варіантів поведінки в окремих класах та делегування їх виконання залежно від стану.

4) Використання стану в бізнес-логіці

```

1 usage  Nastenaaa *
public Transaction addExpense(User user, Account account, Category category,
                             BigDecimal amount, LocalDate date, String note) {

    account.getState().ensureCanPost(trxType: "EXPENSE", amount);

    BigDecimal balance = txRepo.balanceForAccount(account.getId());
    if (balance.subtract(amount).compareTo(BigDecimal.ZERO) < 0) {
        throw new IllegalStateException(
            "Недостатньо коштів на рахунку '" + account.getName() +
            "'. Баланс: " + balance + ", сума витрати: " + amount
        );
    }
    Transaction t = new Transaction(id: 0, account, date, amount.negate(), type: "EXPENSE",
                                     category, note, user);
    return txRepo.save(t);
}

no usages  new *
public BigDecimal getCurrentBalance(Account account) {
    return txRepo.balanceForAccount(account.getId());
}

1 usage  Nastenaaa *
public Transaction addIncome(User user, Account account, Category category,
                             BigDecimal amount, LocalDate date, String note) {

    account.getState().ensureCanPost(trxType: "INCOME", amount);

    Transaction t = new Transaction(id: 0, account, date, amount, type: "INCOME",
                                     category, note, user);
    return txRepo.save(t);
}
}

```

У доменному сервісі FinanceService патерн State використовується безпосередньо під час створення транзакцій.

1. Перевірка станом рахунку:

На початку методу викликається

```
account.getState().ensureCanPost(trxType, amount);
```

- це делегування перевірки поточному стану рахунку (ActiveState, BlockedState, ClosedState).

- Для ActiveState - операції дозволені.
- Для BlockedState - витрати (EXPENSE) блокуються винятком.
- Для ClosedState - будь-які операції заборонені (виняток).

2. Додаткові бізнес-правила (для витрат):

У `addExpense(...)` після перевірки станом обчислюється поточний баланс і виконується перевірка на достатність коштів. Якщо коштів не вистачає, кидається `IllegalStateException` з пояснювальним текстом (назва рахунку, баланс, сума витрати).

3. Створення та збереження транзакції:

Якщо обмеження стану та доменні перевірки пройдені, формується об'єкт `Transaction` із відповідним типом ("EXPENSE" або "INCOME") і зберігається через репозиторій (`txRepo.save(...)`).

5) Перемикання стану з UI

```
private void onFreeze() { currentAccount().block(); refreshStateBadge(); }  
private void onUnfreeze() { currentAccount().activate(); refreshStateBadge(); }  
private void onClose() { /* перевірка балансу == 0 */ currentAccount().close();  
refreshStateBadge(); }
```

У цьому фрагменті я показую, як UI ініціює переходи станів рахунку, не знаючи деталей реалізації кожного стану. Кнопки викликають методи контексту `Account`, а далі вся різниця поведінки інкапсульована у класах станів

UI не містить жодних `if-else` по станах для бізнес-логіки - він лише тригерить події. Рішення про дозволеність операцій приймають класи станів через методи контексту, що і демонструє сутність патерна `State`.

Рахунок: Основний рахунок ▾

Статус рахунку: Активний Заморозити Розморозити Закрити

Сума: 300

Тип: EXPENSE ▾

Категорія: Одяг ▾

Дата (yyyy-mm-dd): 2025-10-03

Примітка:

Зберегти

Рахунок: Основний рахунок ▾

Статус рахунку: Заморожений Заморозити Розморозити Закрити

Сума: 300

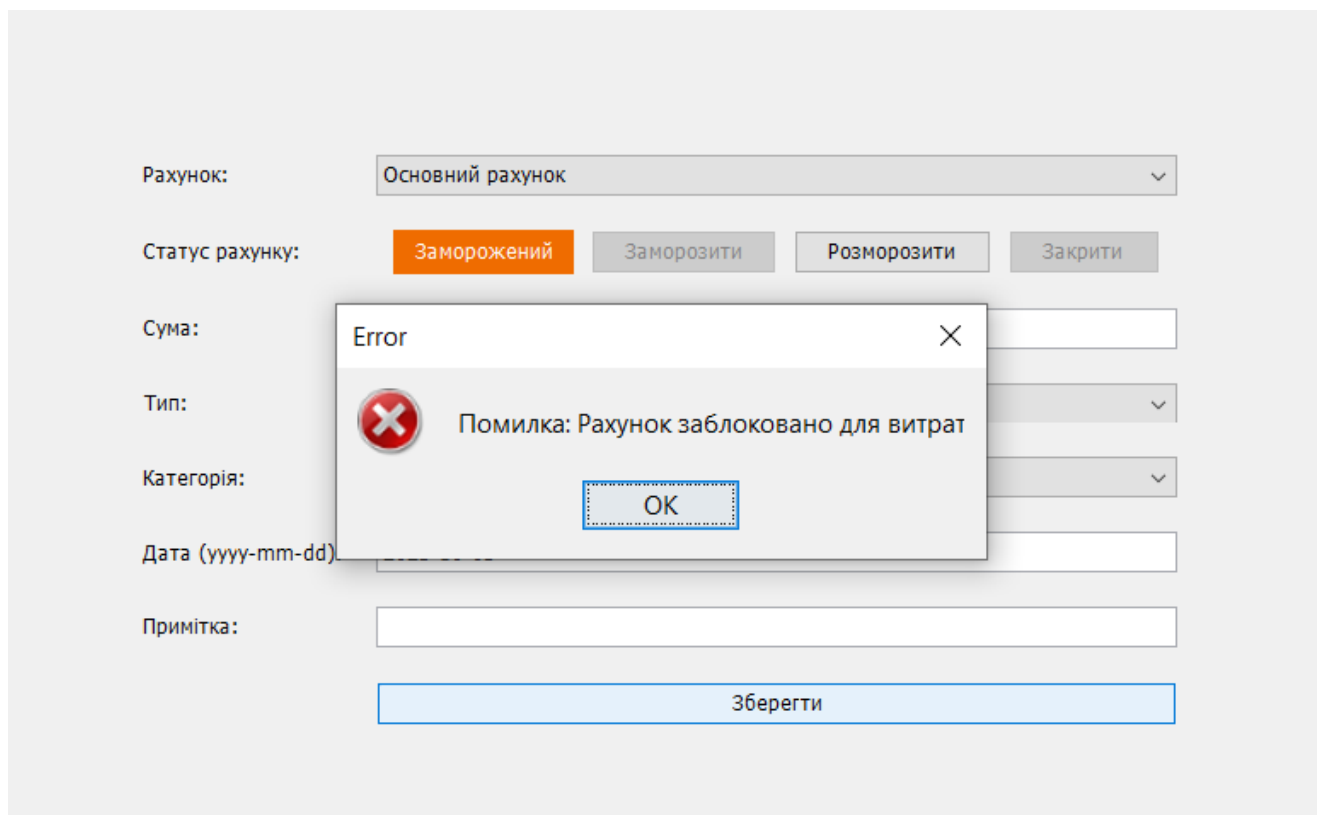
Тип: EXPENSE ▾

Категорія: Одяг ▾

Дата (yyyy-mm-dd): 2025-10-03

Примітка:

Зберегти



Висновок

У ході виконання лабораторної роботи я дослідила принцип роботи шаблону State та застосувала його у власному проєкті «Особиста бухгалтерія». Було реалізовано інтерфейс AccountState і конкретні стани рахунку: ActiveState, BlockedState, ClosedState. Клас Account виступає контекстом і делегує виконання операцій поточному стану. Це дозволило уникнути громіздких умовних конструкцій та зробити поведінку рахунку динамічною й розширюваною. Я перевірила інтеграцію патерна у різних рівнях системи: у бізнес-логіці (FinanceService), у користувацькому інтерфейсі (події для зміни стану) та у самій моделі (Account). У результаті стало зрозуміло, що використання патерна State робить код більш чистим, зрозумілим і придатним до подальшого розширення (наприклад, можна додати нові стани без зміни існуючого коду).

Отже, реалізація State показала, як за допомогою патернів можна розв'язувати типові задачі проєктування та підвищувати якість програмних систем.

Контрольні питання

1. Що таке шаблон проєктування?

Шаблон проєктування – це формалізований опис часто повторюваної задачі в програмуванні та стандартне рішення цієї задачі. Він являє собою «ескіз» архітектурного рішення, яке перевірене практикою і може бути багаторазово використане в різних проєктах. Кожен шаблон має власну назву, що дає можливість розробникам швидко розуміти один одного, використовуючи єдину термінологію.

2. Навіщо використовувати шаблони проєктування?

Вони дозволяють:

- зробити модель системи більш структурованою та зрозумілою;
- спростити підтримку та розвиток програмного коду;
- підвищити стійкість системи до змін у вимогах;
- спростити інтеграцію систем між собою;
- створюють «спільну мову» для розробників, яка допомагає швидко обмінюватися ідеями.

Іншими словами, шаблони підвищують якість проєктування і зменшують ризик помилок.

3. Яке призначення шаблону «Стратегія»?

Його мета відокремити алгоритми від класу, який їх використовує, і зробити їх взаємозамінними. Це дозволяє в будь-який момент змінити алгоритм роботи об'єкта без зміни його структури. Приклад: у програмі можна реалізувати кілька алгоритмів сортування (швидке, бульбашкою, вставками) і динамічно вибирати потрібний.

4. Нарисуйте структуру шаблону «Стратегія».

Словесний опис:

- інтерфейс Strategy описує метод для виконання алгоритму;
- класи ConcreteStrategyA, ConcreteStrategyB реалізують цей інтерфейс різними способами;
- клас Context зберігає посилання на стратегію й викликає її метод.

Таким чином, контекст делегує виконання конкретній стратегії.

5. Які класи входять в шаблон «Стратегія», та яка між ними взаємодія?

- Strategy – абстрактний інтерфейс алгоритму.
- ConcreteStrategy – реалізації алгоритмів.
- Context – викликає методи алгоритму через обраний об'єкт стратегії.

Взаємодія: Context → Strategy (інтерфейс) → ConcreteStrategy

6. Яке призначення шаблону «Стан»?

Він дозволяє змінювати поведінку об'єкта в залежності від його поточного стану. Усі стани виділені в окремі класи, а об'єкт просто переключається між ними. Це зручно, коли об'єкт має кілька режимів роботи (наприклад, банківський рахунок може бути активним, замороженим або закритим).

7. Нарисуйте структуру шаблону «Стан».

Словесний опис:

- інтерфейс State описує поведінку для конкретного стану;
- класи ConcreteStateA, ConcreteStateB реалізують цей інтерфейс по-різному;
- клас Context зберігає посилання на поточний стан і делегує йому виконання.

При зміні стану контекст просто підмінює об'єкт у полі state.

8. Які класи входять в шаблон «Стан», та яка між ними взаємодія?

- State – інтерфейс, що описує поведінку.
- ConcreteState – конкретні реалізації станів.
- Context – зберігає поточний стан і викликає його методи.

Взаємодія: Context → State → ConcreteState, при цьому заміна стану = заміна об'єкта класу.

9. Яке призначення шаблону «Ітератор»?

Забезпечує послідовний доступ до елементів колекції без розкриття її внутрішньої структури. Дозволяє реалізувати різні способи обходу (послідовний, у зворотному порядку, за певним правилом), залишаючи колекцію просто сховищем даних.

10. Нарисуйте структуру шаблону «Ітератор».

Словесний опис:

- інтерфейс `Iterator` з методами `First()`, `Next()`, `IsDone`, `CurrentItem`;
- клас `ConcreteIterator` реалізує цей інтерфейс;
- інтерфейс `Aggregate` описує метод створення ітератора;
- клас `ConcreteAggregate` створює конкретний ітератор.

11. Які класи входять в шаблон «Ітератор», та яка між ними взаємодія?

- `Iterator` – інтерфейс для доступу до елементів.
- `ConcreteIterator` – реалізує обхід.
- `Aggregate` – інтерфейс колекції, що створює ітератор.
- `ConcreteAggregate` – колекція з конкретною реалізацією.

Взаємодія: колекція створює ітератор, а користувач обходить дані через нього.

12. В чому полягає ідея шаблону «Одинак»?

Ідея гарантувати існування лише одного екземпляра класу у програмі й надати глобальну точку доступу до нього. Це зручно для налаштувань, логерів чи єдиної черги повідомлень.

13. Чому шаблон «Одинак» вважають «анти-шаблоном»?

Бо він фактично є аналогом глобальних змінних, що зберігають стан. Такі об'єкти важко тестувати, відстежувати їх поведінку та вони можуть створювати приховані залежності в коді. Це ускладнює супровід і часто вказує на поганий дизайн.

14. Яке призначення шаблону «Проксі»?

Призначений для створення об'єкта-замісника, який контролює доступ до іншого об'єкта. Він може додавати додаткову логіку (кешування, відкладена ініціалізація, контроль доступу) без зміни клієнтського коду.

15. Нарисуйте структуру шаблону «Проксі».

Словесний опис:

- інтерфейс `Subject` описує загальні методи;
- клас `RealSubject` – реальний об'єкт, який виконує роботу;

- клас Proxy – містить посилання на RealSubject і викликає його методи, додаючи власну логіку;
- клієнт працює з інтерфейсом Subject, не знаючи, що перед ним проксі.

16. Які класи входять в шаблон «Проксі», та яка між ними взаємодія?

- Subject – спільний інтерфейс.
- RealSubject – реальний об'єкт.
- Proxy – замісник, який зберігає посилання на RealSubject.

Клієнт звертається до Proxy, а той делегує виклики RealSubject, додаючи власну логіку.