МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ТЕХНОЛОГІЙ ТА ДИЗАЙНУ

ФАКУЛЬТЕТ МЕХАТРОНІКИ ТА КОМП'ЮТЕРНИХ ТЕХНОЛОГІЙ

КАФЕДРА КОМП'ЮТЕРНИХ НАУК

Кваліфікаційна робота

на тему:

Розроблення програмного забезпечення клієнт-серверного додатка з використанням патернів проектування

Рівень вищої освіти другий (магістерський)

Спеціальності 122 Комп'ютерні науки

Освітня програма Комп'ютерні науки

Виконала: студент групи МгІТ-1-22

Мирець Р.В.

Науковий керівник:

к.т.н., доц. Астістова Т.І.

Рецензент:

к.т.н., доц. Мельник Г.В.

Київ 2023

КИЇВСЬКИЙ НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ ТЕХНОЛОГІЙ ТА ДИЗАЙНУ

Факультет мехатроніки та комп'ютерних технологій

Кафедра комп'ютерні науки

Рівень вищої освіти   другий (магістерський)

Спеціальність 122 Комп'ютерні науки

Освітня програма Комп'ютерні науки

ЗАТВЕРДЖУЮ

Завідувач кафедри КН

__________________Володимир ЩЕРБАНЬ

«_____»____________________2023______року

З А В Д А Н Н Я

НА КВАЛІФІКАЦІЙНУ РОБОТУ СТУДЕНТА

Мирець Роману Вікторовичу

1.Тема роботи: Розроблення програмного забезпечення клієнт-серверного додатка   з використанням патернів проектування ,

науковий керівник роботи: Астістова Тетяна Іванівна, к.т.н., доц.,

затверджені наказом закладу вищої освіти від _12 . 09.2023 року , № 210-уч.

2. Строк подання студентом роботи 12.11. 2023р.

3. Вихідні дані до роботи:

Розробка кафедри комп'ютерних наук

4. Зміст кваліфікаційної роботи (перелік питань, які потрібно розробити)

РОЗДІЛ 1.Теоретичний;   РОЗДІЛ 2. Моделі та архітектурні рішення розробки додатків; РОЗДІЛ 3. Розробка програмних модулів патернів Додатки -програмні коди; презентація.

5. Дата видачі завдання: 08. 2023р.

КАЛЕНДАРНИЙ ПЛАН

№ з/п	Назва етапів кваліфікаційної роботи	Терміни виконання етапів	Примітка про виконання
1	Вступ	20.08.2023	
2	Розділ 1. Теоретичний	10.09.2023	
3	Розділ 2. Моделі та архітектурні рішення розробки додатків	5.10.2023	
4	Розділ 3. Розробка програмних модулів патернів	25.10.2023	
5	Висновки	28.10.2023	
6	Оформлення кваліфікаційної роботи (чистовий варіант)	31.10.2023	
7	Подача кваліфікаційної роботи науковому керівнику для відгуку	01.11.2023	
8	Подача кваліфікаційної роботи для рецензування	04.11.2023	
9	Перевірка кваліфікаційної роботи на наявність ознак плагіату	04.11.2023	
10	Подання кваліфікаційної роботи на затвердження завідувачу кафедри	07.11.2023	

Студент ____________ Роман МІРЕЦЬ

Науковий керівник роботи ____________ Тетяна АСТІСТОВА

АНОТАЦІЯ

Мирець Р.В. Розроблення програмного забезпечення клієнт-серверного додатка з використанням патернів проектування.

Кваліфікаційна робота за спеціальністю 122 - «Комп'ютерні науки» – Київський національний університет технологій та дизайну, Київ, 2023 рік.

Кваліфікаційна робота присвячена розробленню програмного забезпечення клієнт-серверного додатка з використанням патернів проектування та удосконаленню проектування додатків на основі порівняльної характеристики сучасних технологій.

В роботі була проаналізована класифікація найбільш поширених патернів, розглянуто види та складові у кожній групі категорій патернів, запропоновані найбільш відповідні види шаблонів для різного типу задач, розглянута концепція патерну Model -View – Controller, проаналізована логіка та види інтерфейсів, розглянуто інструментарії побудови інтерфейсів на базі шаблонів, розглянуто моделі та архітектурні рішення розробки клієнт-серверного додатка, розроблено модулі різних видів патернів на різних мовах програмування.

Ключові слова: програмне забезпечення, патерни проєктування, клієнт-серверні додатки, інтерфейс, PHP, Python.Ruby.

.

ANNOTATION

Mirets R.V. Development of client-server application software using design patterns.

Qualification thesis for specialty 122 - "Computer Science" - Kyiv National University of Technology and Design, Kyiv, 2023.

The master's thesis is devoted to the development of client-server application software using design patterns and the improvement of application design based on the comparative characteristics of modern technologies.

The work analyzed the classification of the most common patterns, considered the types and components in each group of pattern categories, proposed the most suitable types of templates for different types of tasks, considered the concept of the Model -View - Controller pattern, analyzed the logic and types of interfaces, considered the tools for building interfaces based on patterns, models and architectural solutions for the development of a client-server application were considered, modules of various types of patterns were developed in various programming languages

Keywords: software, design patterns, client-server applications, interface, PHP, Python, Ruby

ЗМІСТ

ВСТУП…………………………………………………………………

РОЗДІЛ 1. Теоретичний…………………………………………………..

1.1.Постановка завдання та розкриття проблеми………………………

1.2.. Загальні поняття та класифікація патернів …………………………

1.3. Види патернів та їх складові……………………………………………

1.3.1. Патерни Creational ……………………………………………..

1.3.2. Патерни Structural ……………………………………………..

1.3.3 Патерни Behavioral…………………………………………

1.4. Концепція патерну Model -View - Controller………………………..

Висновки до першого розділу…………………………….................

РОЗДІЛ 2. Моделі та архітектурні рішення розробки додатків…..

2.1. Модель технології клієнт-сервер ………………………………………

2.2 . Архітектурні рішення клієнт - серверної технології …………………

2.3. Методи аналізу структури програм……………………………………

2.4 . Інструментарії побудови інтерфейсів на базі шаблонів…………

2.4.1**.** Специфікування інтерфейсів…………………………………

Висновки до другого розділу…………………………………………………

РОЗДІЛ 3. Розробка програмних модулів патернів ……………………

3.1. Мова програмування для патернів ……………………………………

3.1. 1. Мова програмування Swift………………………………………

3.1. 2. Мова програмування Python……………………………………

3.2 . Розробка програмного модуля патерну Одинак……………………..

3.3 Розробка програмного модуля патерну Абстрактна фабрика….. …..

3.4. Розробка програмного модуля патерну Спостерігач……………

Висновки до третього розділу……………………………………………

ВИСНОВКИ……………………………………………………………..

Список використаних джерел………………………………………….

Додаток А – програмні коди…………………………………………………

Додаток Б - презентація………………………………………………………

ВСТУП

Актуальність теми. Технології та архітектура веб - додатків постійно змінюються. При розробці клієнт - серверної програми, що претендує на тривале існування, важливо вибрати патерн проектування, який повністю зможе задовольнити всі потреби команди - розробника та самого продукту.

Патерни проектування є ефективним способом розв'язання задач проектування програмного забезпечення.

Розробники програм дуже часто вирішують абсолютно ідентичні завдання та знаходять схожі рішення. Різні програмісти вирішують подібні завдання різними способами , хтось швидше та ефективніше Щоб не створювати створене, доречно використовувати готові шаблони (патерни) проектування.

При розробленні клієнт-серверного додатка доречно використовувати архітектурні шаблони.

Мета роботи. Метою кваліфікаційної роботи є розроблення програмного забезпечення клієнт-серверного додатка з використанням патернів проектування.

Завдання роботи. Завданням роботи було провести аналіз найбільш поширених видів патернів, знайти плюси та мінуси кожного патерну, розробити коди патернів різними мовами програмування. Вибрати найбільш привабливий варіант для нашого прототипу абстрактного завдання або знайти готове рішення.

Об’єкт та предмет дослідження. Об'єктом дослідження є розробка програмного забезпечення з використанням патернів проектування, що відрізняються відсутністю готового рішення при встановленні системи управління проектом на власному сервері.

Методи та засоби дослідження. Аналіз літературних джерел, дослідження архітектури програмного додатку. Дослідити логіку розробки клієнтської частині та моделі даних.

Наукова новизна та практичне значення отриманих результатів. Аналіз розглянутих патернів показав, що немає єдиного паттерна, який можна використати при розробки програм. Кожен патерн краще використовувати в тій чи іншій ситуації та в залежності від складності додатку розбивати класи на компоненти залежно від їхньої ролі.

Вихідні дані досліджень будуть впроваджуватись в кампанії по розробці програмних продуктів, а також іншим компаніям, що потребують планування проектів .

Результати магістерської роботи були апробовані в наукової статті:

1. Використання патернів проектування для розробки програмного забезпечення / Т. І. Астістова, Р. В. Мирець // Інформаційні технології в науці, виробництві та підприємництві : збірник наукових праць молодих вчених, аспірантів, магістрів кафедри комп’ютерних наук та технологій / за заг. наук. ред. В. Ю. Щербаня. – Київ : ТОВ "Фастбінд Україна", 2023. – С. 105-108.

https://er.knutd.edu.ua/handle/123456789/24121

РОЗДІЛ 1.ТЕОРЕТИЧНИЙ

1.1 **Постановка завдання та розкриття проблеми**

Створення сучасної програми ефективною та прибутковою, потребує від програмістів використання сучасних технології розробки. З такими проблемами проектування у світі постійно хтось стикається. Часто багато розробників вирішують абсолютно ідентичні завдання і знаходять схожі рішення. Різні програмісти вирішуватимуть подібні задачі подібним чином, швидше та ефективніше. Багато розробників шукали шляхи підвищення гнучкості і ступеня повторного використання своїх програм, а знайдені рішення втілювати в короткій і легко застосовної на практиці формі.

Використовувати готові шаблони (патерни) проектування дозволяє не створювати те, що вже створено. Саме тому і з'явилися патерни.

Патерни придумали для того, щоб пришвидшити реалізацію певних стандартизованих задач згідно класифікації шаблонів-заготовок, а це значно полегшить життя програмісту

Патерни — це не тільки алгоритм побудови коду, але й опис проблем, які призвели до такого рішення.

Розробляючи клієнт-серверні додатки, які претендують на довготривале існування, важливо вибрати патерни, які повністю задовольнять всі потреби замовника та самої розробки.

Аналіз розробок показує**, що** найбільш ефектним способом при проектуванні програмного забезпечення є використання шаблонів програмного забезпечення . Шаблони не можна транслювати в програмний код, тому що вони не є закінченим зразком.

Патерни програмування – це напрацьовані ефективні підходи, техніки та правила вирішення задач при створенні програмного забезпечення. Вони не прив'язуються до певної мови програмування і можуть бути застосованими в основному незалежно від конкретної мови [1].

Від різноманітних програмістів існує вже багато різноманітних категорій

і видів патернів, ми дамо класифікацію найбільш поширених патернів , які групуються у категорії патернів, їх види та складові.

При проєктуванні клієнт-серверних додатків рекомендовано обирати архітектурні патерні, що значно спрощує основні операції при проектуванні таких видів додатків та дає досить добрі перспективи у майбутньому.

1.2 Загальні поняття та класифікація патернів

Патерни проектування (шаблони) є керівництвом для вирішення проблем, які повторюються. Вони не є класами, пакетами або бібліотеками, які можна підключити до вашого за стосунку. Це методика, яка вирішує певні проблеми в залежності від ситуації.

Повторювана архітектурна конструкція, яка вирішує проблеми у рамках часто виникаючого контексту при розробці програмного забезпечення, називають шаблоном проектування або патерном.

Шаблони проектування не є вирішенням усіх проблем, це підходи до вирішення проблем, які можуть стати порятунком при використанні в потрібних місцях. Патерн являє собою загальний принцип вирішення певної проблеми, який треба підлаштовувати для потреб програми.

Патерни та алгоритми описують типові рішення відомих проблем Але алгоритм — це чіткий набір дій, а патерн — це високорівневий опис рішення, реалізація якого може відрізнятися у двох різних програмах.

Патерн вирішує наступне:

- проблему;
- мотивацію до способу вирішення проблеми;
- структуру класів ;
- особливості реалізації в різних контекстах;
- приклад однією з мов програмування;
- зв’язки з іншими патернами.

Такі вимоги до опису розв'язуваних завдань, дозволили зібрати великий каталог патернів, додатково перевіривши кожен патерн на дієвість.

Користь патернів:

- Використовують готові та перевірені рішення (витрачаєте менше часу).
- Стандартизація коду (типові уніфіковані рішення), - менше прорахунків при проектуванні.
- Загальний словник програмістів.

Всі види патернів, а їх 23, можна класифікувати:

- за рівнем деталізації системи проектування ;
- за призначенням.

За рівнем деталізації системи проектування можна виділити низькорівневі та прості патерни, які мають назву, *ідіоми.* Цей вид патернів має сенс в рамках однієї мови програмування і є не дуже універсальним .

До універсальних патернів можна віднести *архітектурні патерни*, які можна реалізувати практично будь- якою мовою. Вони потрібні для проектування всієї програми, а не окремих її елементів.

Класифікація за призначенням, виділяє три основні групи патернів:

- ***Породжуючі патерни -*** дбають про гнучке створення об'єктів без внесення до програми зайвих залежностей, надають рекомендації для створення нових об'єктів. Існує 5 породжуючих патернів.
- ***Структурні патерни -*** показують різні способи побудови зв'язків між об'єктами. Всього є 7 структурних патернів.
- ***Поведінкові патерни -*** дбають про ефективну комунікацію між об'єктами, надають рекомендації для реалізації функції існуючого об'єкта. Є 11 поведінкових патернів.

1. 3. Види патернів та їх складові

1.3.1. Патерни Behavioral

Патерни Behavioral відносяться до поведінкових патернів . Вони вирішують проблеми безпечної , ефективної взаємодії між об'єктами програми. Ці ознаки цього шаблону дають можливість зробити більш гнучкою програму.

До цього типу патерну входить 6 шаблонів. Розглянемо їх.

1. State

Стан (State) – поведінковий шаблон використовують, коли під час виконання програми об’єкт повинен змінити свою поведінку в залежності від стану.

Патерн с кладається із 3 блоків:

1. **Context** – клас, об'єкти якого повинні змінювати свою поведінку залежно від стану.
2. **State** – інтерфейс, який має реалізувати кожен із конкретних станів. Через цей інтерфейс Context взаємодіє зі станом, делегуючи виклики методів.
3. **ConcreteState1, ConcreteState2** – класи конкретних станів. Повинні містити інформацію про умови та стан в які може переходите об’єкт з поточного стану. Наприклад : з ConcreteState1 об’єкт може переходите в стан ConcreteState2 і ConcreteState3, а з ConcreteState2 – назад у ConcreteState1 тощо. Об’єкт одного з них повинен містити Context під час створення.

Машину станів найчастіше реалізують за допомогою множини умовних операторів, if або switch, які перевіряють поточний стан об’єкта та виконують відповідну поведінку

```
class Document is
    field state: string
    // ...
    method publish() is
        switch (state)
            "draft":
                state = "moderation"
                break
            "moderation":
                if (currentUser.role == "admin")
                    state = "published"
```

```
            break
        "published":
            // Do nothing.
            break
    // ...
```

Такий код дуже складно підтримувати. Навіть найменша зміна логіки переходів змусить перевіряти роботу всіх методів, які містять умовні оператори машини станів.

Основна ідея в тому, що програма може знаходитися в одному з кількох станів, які увесь час змінюють один одного. Набір цих станів, а також переходів між ними, визначений наперед. Перебуваючи в різних станах, програма може по-різному реагувати на одні і ті самі події, що відбуваються з нею.

Наприклад , розробляти гру, де персонаж може бігати, плавати та літати. Якщо персонаж потрапив у воду, то може обмежити його поведінку у воді: він тепер не може стріляти, але в нього збереглися якісь дії: плисти вперед, праворуч, ліворуч тощо.

Стан персонажа можна описати об'єктом State, що має методи, які можна викликати і які щось будуть робити. Після того, як т персонаж заліз у воду, потрібно просто змінити в середині його посилання на інший об'єкт State і він змінює свій стан.

Ще приклад: Ваш смартфон поводиться по-різному в залежності від поточного стану:

- Якщо телефон розблоковано, натискання кнопок телефону призведе до якихось дій.
- Якщо телефон заблоковано, натисканная кнопок призведе до появи екрану розблокування.
- Якщо телефон розряджено, натискання кнопок призведе до появи екрану зарядки.

2. Strategy .

Стратегія (Strategy) – поведінковий шаблон проєктування, призначений для визначення сімейства алгоритмів, інкапсуляції кожного з них та забезпечення їх взаємозамінності. Він визначає сімейство схожих алгоритмів та розміщує їх у власному класі . Ці класи називають *стратегіями*. Це дозволяє обирати алгоритм шляхом визначення відповідного класу під час виконання програми..

Шаблон Strategy дозволяє під час виконання програми змінювати обраний алгоритм незалежно від об'єктів-клієнтів..

Стратегію патерна застосовують у випадках, коли використовують різні алгоритми в залежності від поточного стану системи або від контексту.

відмова від умовних Позитивні сторони:

- операторів та перемикачів ;
- виклик одним стандартним чином всіх алгоритмів;
- інкапсуляція реалізації різних алгоритмів;
- система незалежна від змін бізнес-правил.

Патерн має схожість з патерном State. Наприклад, якщо персонаж гри буде змінювати зброюу , можна просто змінити посилання на об'єкт, який описує, як ця зброя працює.

Всі стратегії повинні мати єдиний інтерфейс, це дає можливість контексту бути незалежним від конкретних класів стратегій.

Цей метод стане також у нагоді клієнтському коду навігатора, наприклад, кнопкам-перемикачам типів маршрутів в інтерфейсі користувача.

3. Template Method або Шаблонний метод

Цей вид патерну визначає алгоритм, в якому етапи делегуються підкласами , які перекладають відповідальність на них, при цьому не змінюють структуру алгоритму.

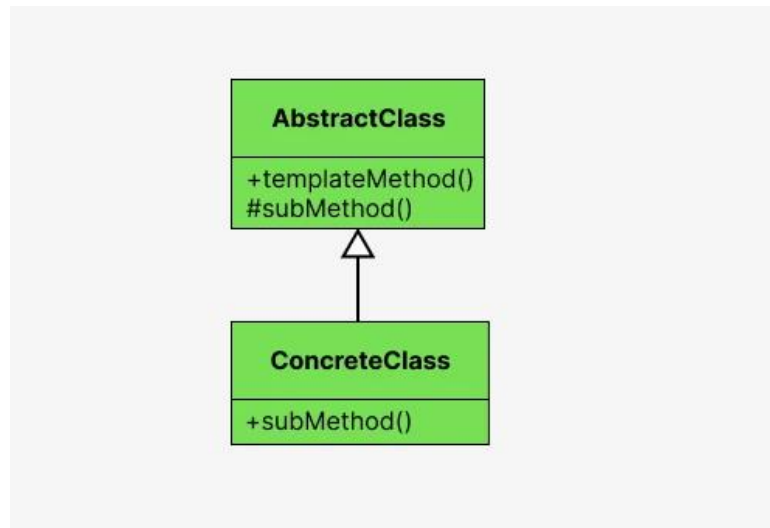


Рисунок 1.1 - Поведінковий шаблон проєктування Strategy

Патерн Шаблон має два класи:

1. Abstract class (абстрактний клас) – це шаблоний метод, що визначає скелет алгоритму та абстрактні операції, що заміщуються у спадкоємцях для реалізації кроків алгоритму. Цей метод викликає визначені в Abstract class потрібні для заміщення операції.

2. ConcreteClass (конкретний клас) – передбачає, що інваріантні кроки алгоритму виконаються в AbstractClass, а Concrete class реалізує операції заміщення необхідним способом.

Патерн використовують :

- одноразове використання інваріантної частини алгоритму ;
- для уникнення дублювання , відбувається локалізація та ви членування кількох класів загального коду;
- дає дозвіл на розширення коду лише у певних місцях.

Зазвичай патерн використовують у парі : абстрактний клас та його реалізація .

Розглянемо приклад для розуміння поняття патерну Template Method.

Якщо розробляється програма з великим обсягом даних (дата - майнінгу), то завантажуватимуться документи з різними форматами (PDF, DOC, CSV), але мета програми, це дістати з них корисну інформацію. Тобто потрібно знайти правила та закономірності.

На рисунку 1.2 представлено різні класи дата - майнінгу обробки документів.

У першій версії обмежились обробкою тільки DOC файлів. Далі, у наступній версії, додали підтримку CSV. А через деякий час «прикрутили» роботу з PDF документами.

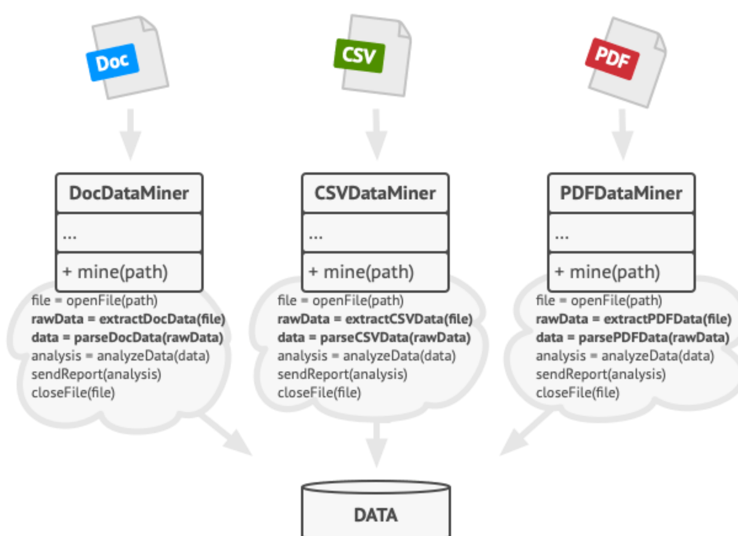


Рисунок 1. 2 - Класи дата - майнінгу обробки різних файлів

Код обробки документів усіх трьох класів містить багато спільного. Повторної реалізації видобування даних у кожному з класів не повинно бути

Патерн розбиває алгоритм на послідовність кроків, описуючи їх в окремих методах. Підкласи перевизначать кроки алгоритму, а структура алгоритму залишається незмінним.

Розглядаючи приклад з дата - майнінгом можемо створити для всіх трьох алгоритмів загальний базовий клас. Він послідовно викликає кроки розбору документів. Код, який працює з об'єктами, містить умови та загальний інтерфейс може спростити його.

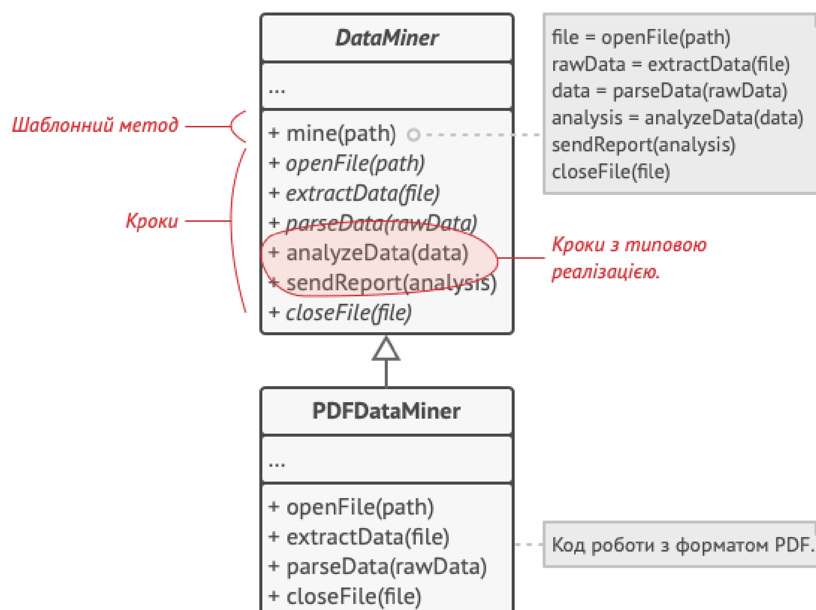


Рисунок 1.3 - Структура шаблонного патерна

Спільну поведінку для всіх трьох класів треба винести до суперкласу. Кроки відкривання та закривання документів залишаться абстрактними, тому що вони відрізняються для всіх підкласів. В базовому класі з'явиться код обробки даних , який є однаковим для всіх типів документів..

З'явилося три типу кроків: абстрактні, де підклас обов'язково має реалізувати кроки *з* типовою реалізацією і третій тип кроків — хуки. Ці кроки виглядають як звичайні методи, але не містять коду. Хук дає підкласам додаткові точки «вклинювання» в хід шаблонного методу.

Наприклад: Під час будівництва типових будинків будівельники використовують підхід, схожий на шаблонний метод. У них є основний архітектурний проєкт, в якому розписані кроки будівництва: заливка фундаменту, витягування стін, покриття даху, встановлення вікон тощо.

Але, незважаючи на стандартизацію кожного етапу, будівельники можуть робити невеликі зміни на кожному з етапів, щоб зробити будинок трішечки не схожим на інші.

4. Chain of Responsibility

Ланцюжок обов'язків **(Chain of responsibility)** – поведінковий шаблон проєктування, призначений для організації в системі рівнів відповідальності,

передає запит далі ланцюжком в залежності від рішення обробника .

Шаблон Ланцюжок використовують, коли:

- розробляється система, в якій є група об'єктів , які можуть обробляти повідомлення певного типу;
- всі повідомлення повинні оброблятися хоча б одним об'єктом системи;
- одні повідомлення в системі на рівні де вони отримані , а інші- пересилаються об'єктам іншого рівня.

5. Зберігач (Знімок, Memento) – поведінковий шаблон проєктування, що дозволяє без порушення інкапсуляції зафіксувати та зберегти внутрішній стан об'єкта таким чином, щоб пізніше відновити його до цього стану ((Вікіпедія).

Шаблон Зберігач використовують, коли:

- необхідно зберегти знімок стану об'єктів для відновлення;
- прямий інтерфейс отримання стану об'єкта розкриває деталі реалізації та порушує інкапсуляцію об'єкта.

Як і багато інших поведінкових патернів, ланцюжок обов'язків базується на тому, щоб перетворити окремі поведінки на об'єкти. В нашому випадку кожна перевірка переїде до окремого класу з одним методом виконання. Дані запиту, що перевіряється, передаватимуться до методу як аргументи.

Важливо зауважити , що патерн пропонує зв'язати всі об'єкти обробників в один ланцюжок. Кожен обробник міститиме посилання на наступного обробника в ланцюзі. Таким чином, після отримання запиту обробник зможе не тільки опрацювати його самостійно, але й передати обробку наступному об'єкту в ланцюжку.

5.Command

Команда (Command) – поведінковий шаблон проєктування, що використовується в об'єктно-орієнтованому програмуванні, що являє собою дію. Об'єкт команди містить саму дію та її параметри.

Щоб викликати якийсь метод, зазвичай потрібно:

- посилання на об'єкт;

- ім'я методу (посилання на метод);
- значення параметрів методу;
- посилання на контекст, який містить об'єкти, що використовуються.

Всі ці дані потрібно запакувати до одного об'єкту - Команди (**command**).
Але й це ще не все: адже команду має хтось виконати. До складу патерну входять ще чотири сутності: команди (**command**), приймач команд (**receiver**), той, що викликає команди (**invoker**) та клієнт (**client**).

Об'єкт Command знає про приймач і викликає метод приймача. Значення параметрів приймача зберігаються у команді. Об'єкт, що викликає (invoker) робить облік та запис виконаних команд. Він не знає про конкретну команду, він знає тільки про інтерфейс.
Обидва об'єкти (об'єкт, що викликає, і кілька об'єктів команд) належать об'єкту клієнта (client). Клієнт вирішує, які команди виконати і коли. Щоб виконати команду, він передає об'єкт команди об'єкту, що викликає (invoker).
Використання командних об'єктів спрощує побудову загальних компонентів, які необхідно делегувати або виконувати виклики методів у будь- який час без необхідності знати методи класу чи параметрів методу.
Використання об'єкта, що викликає (invoker) дозволяє вести облік виконаних команд без необхідності знати клієнту про модель обліку .
Наприклад, розсилка SMS, розсилка листів, розсилка повідомлень у Telegram тощо.

6. Observer Спостерігач Спостерігач — це поведінковий патерн проектування, який створює механізм підписки, що дає змогу одним об'єктам стежити й реагувати на події, які відбуваються в інших об'єктах.

Шаблон **Observer** використовують, коли після зміни стану одного об'єкта потрібно щось зробити в інших, але ви не знаєте наперед, які саме об'єкти мають відреагувати.

Описана проблема може виникнути при розробленні бібліотек користувацького інтерфейсу, якщо вам необхідно надати можливість стороннім класам реагувати на кліки по кнопках.

1.3.2. Патерни Structural

Структурні патерни проєктування

До складу структурних патернів проєктування входить сім видів патернів цього класу, які відповідають за побудову зручних в підтримці ієрархій класів,

- Адаптер (Adapter)- Несумісні інтерфейси різних об'єктів можуть працювати разом.
- Міст (Bridge) - Розділяє класи на дві окремі ієрархії — абстракцію та реалізацію. Патерн дозволяє змінювати код в одній гілці класів, незалежно від іншої.
- Компонувальник (Composite)- Групує декілька об'єктів у деревоподібну структуру, працювати з нею як з одиничним об'єктом.
- Декоратор Decorator Дає змогу динамічно додавати об'єктам нову функціональність, загортаючи їх у корисні «обгортки».
- Фасад Facade Надає простий інтерфейс до складної системи класів, бібліотеки або фреймворку.

1. Адаптер - Wrapper, Обгортка, Adapte

Адаптер — це структурний патерн проектування, що дає змогу об'єктам із несумісними інтерфейсами працювати разом.

Розглянемо приклад:

Програма для торгівлі на біржі, спочатку завантажує біржові котирування з декількох джерел в XML, а потім малює гарні графіки.

Якщо захочемо покращити програму, застосувавши сторонню бібліотеку аналітики, яка підтримує формат даних JSON, несумісний із вашим додатком., зробити це не вдасться.

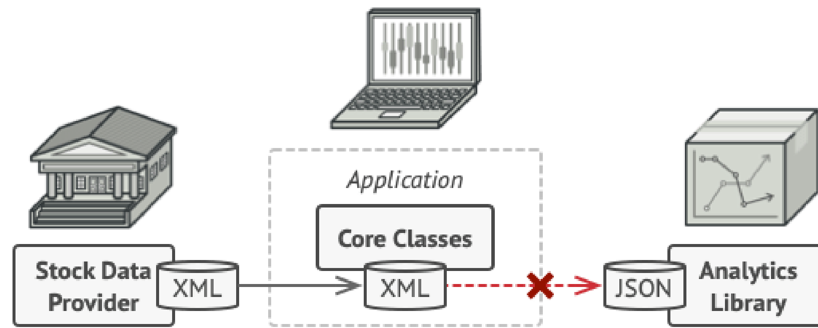


Рисунок 1.4 - Під’єднати сторонню бібліотеку неможливо через несумісність форматів даних

Адаптери допоможе не тільки конвертувати дані з одного формату в інший, але буде допомагати об’єктам із різними інтерфейсами працювати разом. Адаптер буде працювати немов перекладач, трансформуючи інтерфейс таким чином, щоб він став зрозумілим іншому об’єкту. Цей об’єкту навіть не підозрює про існування першого.

Іноді вдається створити навіть *двосторонній адаптер*, який може працювати в обох напрямках.

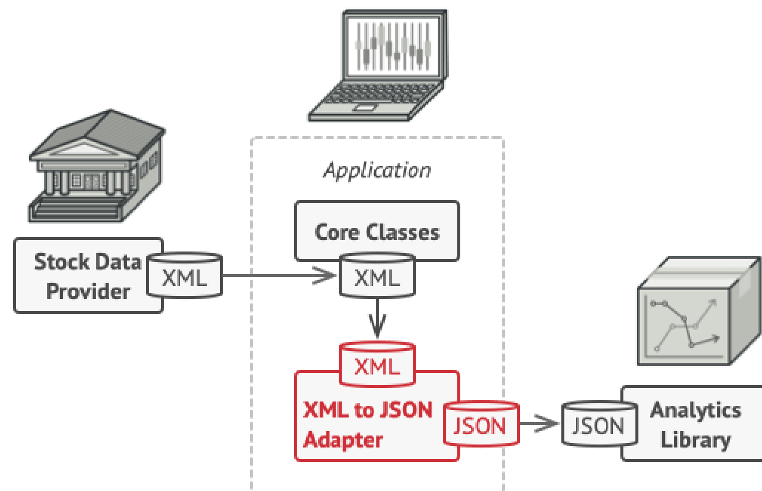


Рисунок 1.5 - Двосторонній адаптер

Таким чином, в прикладі для програми біржових котирувань могли б створити клас XML_To_JSON_Adapter, який би обгортав об’єкт того чи іншого класу бібліотеки аналітики..

Структура патерна **Адаптер**

Адаптер складається :

- адаптера об’єктів,

- адаптера класів та псевдокоду

1. Адаптер об'єктів

Об'єкт адаптера «загортає», тобто містить посилання на службовий об'єкт. Підхід агрегації працює в усіх мовах програмування.

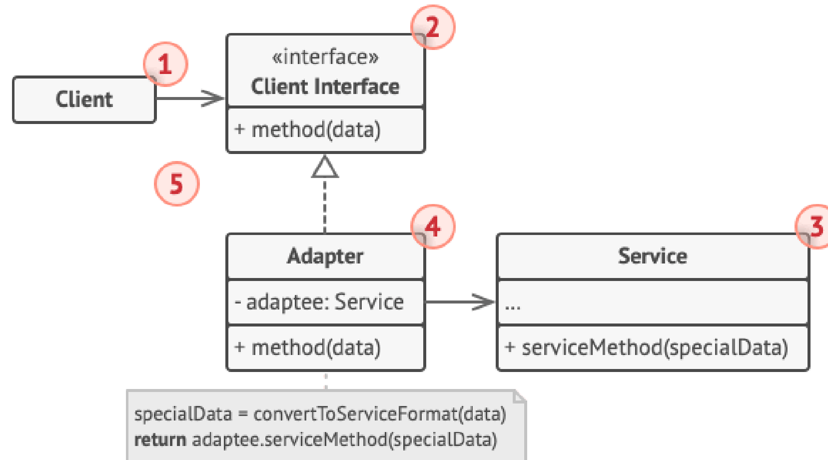


Рисунок 1.6 - Структура адаптера об'єктів

1. Клієнт — клас з бізнес - логікою програми .
2. Клієнтський інтерфейс - протокол, через який клієнт може працювати з іншими класами.
3. Сервіс — це який-небудь корисний клас, який має незрозумілий інтерфейс.
4. Адаптер — клас, який працює одночасно і з клієнтом, і сервісом . Він реалізує клієнтський інтерфейс і містить посилання на об'єкт сервісу.

2. Адаптер класів

Адаптер класів може успадкувати й частину існуючого класу, й частину класу сервісу. Тобто, адаптер успадковує обидва інтерфейси одночасно. Такий підхід можливий тільки в мовах, які підтримують множинне спадкування, наприклад у C++.

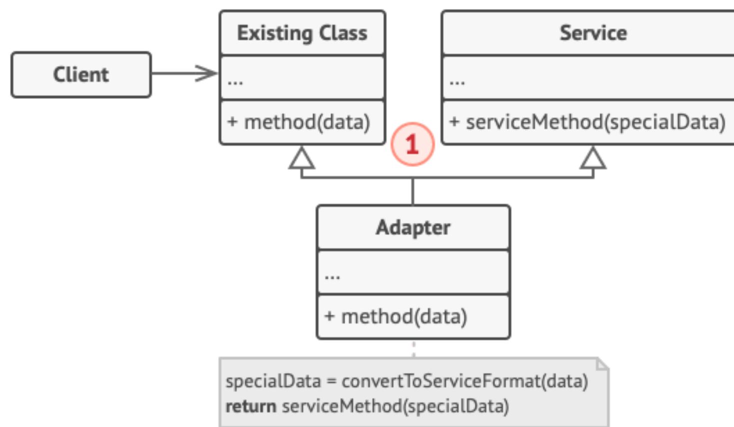


Рисунок 1.7 - Структура адаптера класів

Застосування шаблону:

Якщо потрібно використати сторонній клас, але його інтерфейс не відповідає решті кодів програми. Адаптер створює об'єкт - прокладку, що перетворюватиме виклики програми у формат, зрозумілий сторонньому класу.

Якщо потрібні декілька існуючих підкласів яких не вистачає спільної функціональності, а розширити суперклас не можете. Для вирішення цієї проблеми можливо створити ще один рівень підкласів та додати до них необхідну функціональність. Для цього доведеться дублювати один і той же код в обох гілках підкласів.

1.3.3 Патерни Creational

За означенням у Вікіпедії, патерни Creational, це патерни, що породжують шаблони проектування, які абстрагують процес інстаціювання. Патерні дозволяють зробити систему незалежною від композиції, способу створення та представлення об'єктів. Шаблон, що породжує класи, використовує спадкоємство, щоб змінювати успадкований клас, а шаблон, що породжує об'єкти, делегує інстанціювання іншому об'єкту [Вікіпедія]. Породжувальні Патерни конкурують між собою.

Породжувальні патерни (Creational) :

- Factory Method (фабричний метод);

- Abstract Factory (абстрактна фабрика);
- Builder (будівельник);
- Prototype (прототип);
- Singletone (одинак).

Розглянемо деякі з них (табл.1).

Таблиця 1.

Породжувальні патерни

Singletone	Клас має лише один екземпляр, та надає глобальну точку доступу до нього.
Abstract Factory	Клас, який представляє собою інтерфейс для створення компонентів системи.
Builder	Клас, який представляє собою інтерфейс для створення складного об'єкта.
Prototype	Визначає інтерфейс створення об'єкта через клонування іншого об'єкта замість створення через конструктор
Factory Method	Визначає інтерфейс для створення об'єкта, але залишає подклассам рішення про те, який клас інстанціювати

1. Singleton

Одинак (Singleton) — породжувальний шаблон проектування, який гарантує, що в одно поточному за стосунку буде єдиний екземпляр певного класу, і надає глобальну точку доступу до цього екземпляра

Дуже часто програмісти-початківці люблять зібрати утилітні методи в певний статичний клас – клас, що містить лише статичні методи. Такий підхід має низку мінусів – наприклад, не можна передати посилання на об'єкт цього класу, а методи важко тестувати тощо.

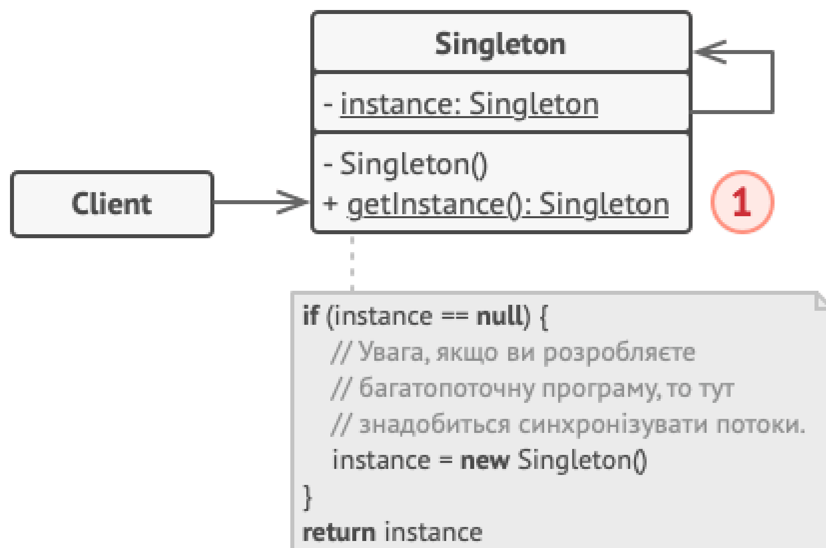


Рисунок 1.6 - Структура класів патерна Одинак

Де: 1- Одинак визначає статичний *метод getInstance,* який повертає один екземпляр свого класу.

Конструктор Одинака повинен бути прихований від клієнтів. Виклик методу *getInstance* повинен стати єдиним способом отримати об’єкт цього класу.

Існує альтернативне рішення – клас- singleton: клас, який може мати лише один об'єкт. Цей об'єкт створюється лише в тому випадку, якщо він ще не існує, інакше повертається посилання на існуючий екземпляр.

Істотно те, що користування саме екземпляром класу, дає доступнішою широку функціональність. Наприклад, цей клас може реалізовувати деякі інтерфейси та його об'єкт можна передати до інших методів, як імплементацію інтерфейсу. Що не можна зробити із набором статичних методів.

Плюси:

- Методи прив'язані до об'єкта, а не до статичного класу – можна передавати об'єкт за посиланням.
- Методи об'єкта значно легше тестувати та мокувати.
- Об'єкт створюється лише за потребою: відкладена ініціалізація об'єкта.

- Прискорення початкового запуску програми, якщо є безліч одиниць, які не потрібні для запуску.
- Одинака можна в подальшому перетворити на шаблон Стратегія або кілька таких об'єктів

.**Мінуси:**

- Ускладнюється контроль над міжпотоковими гонками та затримками.
- Багатопотокового "одинака" складно писати "з голови": доступ до давно побудованого одинака в ідеалі не повинен відкривати м'ютекс. Краще перевірені рішення.
- Конфлікт двох потоків через недобудованого одинака призведе до затримки.
- Якщо об'єкт створюється довго, затримка може заважати користувачеві або порушувати час. У разі його створення краще перенести на стадію ініціалізації програми.
- Потрібні особливі функції для модульного тестування - наприклад, щоб перевести бібліотеку в "не одинокий" режим і повністю ізолювати тести один від одного.
- Потрібна особлива тактика тестування готової програми, адже пропадає навіть поняття "найпростий процес запуску", адже запуск залежить від конфігурації.

2. Factory [Method]

Фабричний метод (Factory Method) — породжувальний шаблон проєктування, який надає підкласам (класам-спадкоємцям) інтерфейс для створення екземплярів певного класу. На момент створення спадкоємці можуть визначити, який клас створювати.

Патерн делегує створення об'єктів спадкоємцям батьківського класу. Це дозволяє використовувати у коді програми не конкретні класи, а маніпулювати абстрактними об'єктами. Цей патерн визначає інтерфейс для створення об'єкта, але залишає підкласам рішення про те, на підставі якого

класу створювати об'єкт. Фабричний метод дозволяє класу делегувати створення підкласів.

Патерн використовують, коли:

- класу заздалегідь невідомо, об'єкти яких підкласів йому потрібно створювати.
- клас спроектовано так, щоб об'єкти, що він створює, специфікувалися підкласами.
- клас делегує свої обов'язки одному з кількох допоміжних підкласів, і планується визначити, який клас забере на себе ці обов'язки.

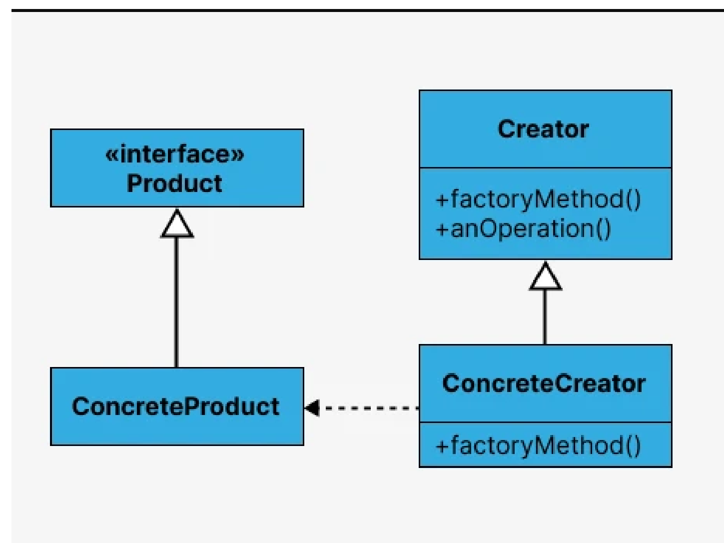


Рисунок 1.7 - Структура класів патерна Factory

3. Abstract Factory

Абстрактна фабрика (Abstract factory) — породжувальний шаблон проєктування, який надає інтерфейс для створення сімейств взаємопов'язаних або взаємозалежних об'єктів без зазначення їх конкретних класів.

Патерн дає змогу створювати сімейства пов’язаних об’єктів, не прив’язуючись до конкретних класів створюваних об’єктів.

Шаблон реалізується створенням абстрактного класу Factory, який є інтерфейсом для створення компонентів системи (наприклад, для іконного

інтерфейсу він може створювати вікна та кнопки). Потім пишуться класи, які реалізують цей інтерфейс. Застосовується у випадках, коли програма має бути незалежною від процесу та типів створюваних нових об'єктів. Коли необхідно створити сімейства або групи взаємопов'язаних об'єктів, без можливості одночасного використання об'єктів із цих різних наборів в одному контексті.

Сильні сторони:

- ізолює конкретні класи;
- полегшує заміну сімейств продуктів;
- гарантує поєднання продуктів.

Припустимо, програма працює з файловою системою. Тоді для роботи в Linux потрібні об'єкти LinuxFile, LinuxDirectory, LinuxFileSystem. А для роботи у Windows потрібні класи WindowsFile, WindowsDirectory, WindowsFileSystem.

Клас Path, який створюється через Path.of(), – це саме той випадок. Path — це насправді не клас, а інтерфейс, і він має реалізації WindowsPath та LinuxPath. А який об'єкт буде створено, приховано від твого коду і вирішуватиметься під час роботи програми.

Клієнтський код повинен працювати як із фабриками, так і з продуктами тільки через їхні загальні інтерфейси. Це дозволить подавати у ваші класи будь-які типи фабрик і виробляти будь-які типи продуктів, без необхідності вносити зміни в існуючий код.

Структура патерна:

1. Абстрактні продукти оголошують інтерфейси продуктів, що пов'язані один з одним за змістом, але виконують різні функції.
2. Конкретні продукти — великий набір класів, що належать до різних абстрактних продуктів (крісло/столик), але мають одні й ті самі варіації (Вікторіанський /Модерн).

3. Абстрактна фабрика оголошує методи створення різних абстрактних продуктів (крісло /столик).
4. Конкретні фабрики - кожна належить до своєї варіації продуктів (Вікторіанський /Модерн) і реалізує методи абстрактної фабрики, даючи змогу створювати всі продукти певної варіації.
5. Незважаючи на те, що конкретні фабрики породжують конкретні продукти, сигнатури їхніх методів мусять повертати відповідні абстрактні продукти. Це дозволить клієнтському коду, що використовує фабрику, не прив’язуватися до конкретних класів продуктів. Клієнт зможе працювати з будь-якими варіаціями продуктів через абстрактні інтерфейси.

Абстрактна фабрика на Java

Абстрактна фабрика задає інтерфейс створення всіх доступних типів продуктів, а кожна конкретна реалізація фабрики породжує продукти однієї з варіацій.

Клієнтський код викликає методи фабрики для отримання продуктів, замість самостійного створювання їх за допомогою оператора *new*. При цьому, фабрика сама стежить за тим, щоб створюваний продукт був потрібної варіації.

Застосування: Патерн можна часто зустріти в Java-коді, особливо там, де потрібно породжувати сімейства класів, не прив’язуючи до них свій код (наприклад, у фреймворках).

Приклади Абстрактної фабрики в стандартних бібліотеках Java:

javax.xml.parsers.DocumentBuilderFactory#newInstance()

javax.xml.transform.TransformerFactory#newInstance()

javax.xml.xpath.XPathFactory#newInstance()

Переваги та недоліки

- Гарантує поєднання створюваних продуктів.
- Звільняє клієнтський код від прив’язки до конкретних класів продукту.
- Виділяє код виробництва продуктів в одне місце, спрощуючи підтримку коду.

- Спрощує додавання нових продуктів до програми.
- Реалізує *принцип відкритості/закритості*.
- Ускладнює код програми внаслідок введення великої кількості додаткових класів.
- Вимагає наявності всіх типів продукту в кожній варіації.

1.4. Архітектура патерна MVC.

Найпопулярніша архітектура додатків є MVC. MVC розшифровується як Model-View-Controller

MVC - це схема поділу даних програми та керуючої логіки на три окремі компоненти: модель, уявлення та контролер - таким чином, що модифікація кожного компонента може здійснюватися незалежно.

• **Модель** (Model) надає дані та реагує на команди контролера, змінюючи свій стан. Там знаходяться об'єкти моделей, парсери, менеджери та мережевий код.

• **Подання** (View) відповідає за відображення даних моделі користувача, реагуючи на зміни моделі.

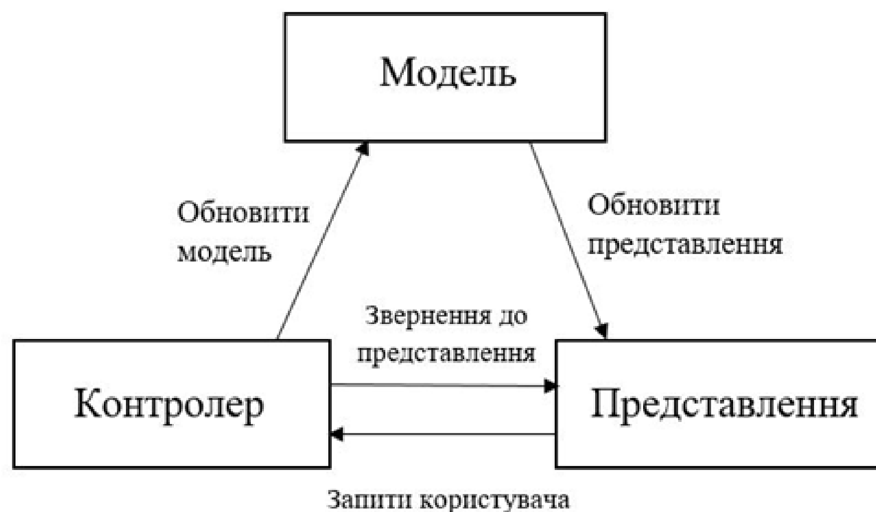


Рисунок 1.8.- Архитектура патерна MVC

• **Контролер** (Controller) інтерпретує дії користувача, сповіщаючи

Модель надає дані та методи роботи з ними: запити до бази даних,

перевірку на коректність. Модель не залежить від подання та контролера надаючи доступ до даних та управління ними.

Модель будується таким чином, щоб відповідати на запити, змінюючи свій стан, при цьому може бути вбудоване повідомлення "спостерігачів". Модель, за рахунок незалежності від візуального подання, може мати кілька різних уявлень для однієї моделі.

Подання відповідає за отримання необхідних даних з моделі та надсилає їх користувачу. Подання не опрацьовує введені дані користувача.

Контролер забезпечує "зв'язок" між користувачем та системою. Контролює та спрямовує дані від користувача до системи та навпаки. Використовує модель та подання для реалізації необхідної дії.

Певна складність полягає в тому, що дана модель за десятки років трохи еволюціонувала. Тобто назва залишилася тією ж, а призначення частин почало змінюватися.

1.4.1. Архітектура MVC у WEB.

Ідея, яка лежить в основі конструкційного шаблону MVC, дуже проста: потрібно чітко розділяти відповідальність за різне функціонування наших додатків:

Model – обробка даних та логіка програми.

View — надання даних користувачеві в будь-якому форматі, що підтримується.

Controller – обробка запитів користувача та виклик відповідних ресурсів.

Додаток поділяється на три основні компоненти, кожен з яких відповідає за різні завдання.

Компоненти клієнт-серверної програми на прикладі

Контролер (Controller)

Користувач натискає на різні елементи на сторінці в браузері, в результаті чого браузер надсилає різні HTTP запити: GET, POST або інші. До контролера можна віднести браузер та JS-код, які працюють усередині сторінки.

Основна функція контролера— викликати методи в необхідних об'єктів, управляти доступом до ресурсів виконання завдань, заданих користувачем. Зазвичай контролер викликає відповідну модель завдання і вибирає відповідний вид.

Модель (Model)

Модель у широкому розумінні – це дані та правила, які використовуються для роботи з даними – разом вони становлять бізнес-логіку програми. Проектування програми завжди починається з побудови моделей об'єктів, якими вона оперує.

Припустимо, у нас є Інтернет - магазин, який торгує книгами, тоді людина — це лише користувач програми , але може бути й автором книги. Ці важливі питання слід вирішити під час проектування моделі.

Модель дає контролеру подання даних, які запросив користувач (повідомлення, сторінку книги, картинки тощо). Модель даних буде однаковою, незалежно від того, як ми хочемо представляти їх користувачеві. Тому ми вибираємо будь - який доступний вид для відображення даних.

Модель містить найважливішу частину логіки нашої програми, логіку, яка вирішує завдання, з яким ми маємо справу (форум, магазин, банк тощо). Контролер містить в основному організаційну логіку для самої програми (як Project Manager).

Вид (View)

View забезпечує різні способи представлення даних, отриманих з моделі. Він може бути шаблоном, що заповнюється даними. Можливо кілька різних views і контролер вибирає, який найкраще підходить для поточної ситуації.

Веб-додаток зазвичай складається з набору контролерів, моделей та видів (views). Контролер може бути тільки на бекенді, але також може бути варіант кількох контролерів, коли його логіка розмазується і по frontend'у. Хороший приклад такого підходу – будь- який мобільний додаток.

1.4.3. Шаблон MVC у Інтернеті

Допустимо потрібно розробити Інтернет - магазин, який займатиметься

продажем книг. Користувач може виконувати такі дії: переглядати книги, реєструватися, купувати, додавати пункти до поточного замовлення, відзначати книги, що сподобалися, і купувати їх.

У додатку має бути модель, яка відповідає за всю бізнес-логіку. Також потрібен контролер, який оброблятиме всі дії користувачів і перетворюватиме їх на виклики методів з бізнес-логіки. При цьому один метод контролера може спричинити багато різних методів моделі.

Також потрібні набори views: список книг, інформація про одну книгу, кошик, список замовлень. Кожна сторінка веб- застосунку — це фактично і є окремий view, який відображає користувачеві певний аспект моделі.

Якщо користувач відкриє список рекомендованих магазином книг для перегляду назв, то всю послідовність дій можна описати у вигляді 6 кроків:

Кроки:

1. Користувач натискає на посилання «рекомендовані» і браузер відправляє запит на, припустимо, /books/recommendations.

2. Контролер перевіряє запит: користувач має бути залогіненим. Або в нас мають бути добірки книг для незалогінених користувачів. Потім контролер звертається до моделі та просить її віддати список книг, рекомендованих користувачеві N.

3. Модель звертається до бази даних, дістає звідти інформацію про книжки: популярні зараз книжки; куплені користувачем книжки, або книжки, куплені його друзями, книжки з його wish list. На основі цих даних модель будує список із 10 рекомендованих книг та повертає їх контролеру.

4. Контролер одержує список рекомендованих книг і дивиться на нього. На цьому етапі контролер ухвалює рішення! Якщо книг мало або список взагалі порожній, він запитує список книг для незалогенного користувача. Якщо зараз триває акція, то контролер може додати до списку акційні книги.

5. Контролер визначається про те, яку сторінку показати користувачеві. Це може бути сторінка з помилкою, сторінка зі списком книг, сторінка привітання, що користувач став мільйонним відвідувачем.

6. Сервер віддає клієнту сторінку (view), вибрану контролером. Вона заповнюється потрібними даними (ім'я користувача, список книг) і йде до клієнта.

7. Клієнт отримує сторінку та відображає її користувачеві.

У чому переваги такого підходу?

1. Найочевидніша перевага, яку ми отримуємо від використання концепції MVC – це чіткий поділ логіки уявлення (інтерфейсу користувача) та логіки програми (серверна частина).

2. Друга перевага – це поділ серверної частини на дві: розумна модель (виконавець) та контролер (центр прийняття рішень).

У попередньому прикладі був момент, коли контролер міг отримати від моделі порожній список рекомендованих книг і вирішував, що з ним робити. Теоретично цю логіку можна було б засунути відразу до моделі.

Модель.

Спочатку при запиті рекомендованих книг модель вирішувала б, що робити, якщо список порожній. Потім довелося б у це місце додати код, що робити, якщо зараз йде акція, потім ще різні варіанти.

Потім виявилося, що адміністратор магазину хоче подивитися, як виглядатиме сторінка користувача без акції, чи навпаки зараз акції немає, а він хоче подивитися, як відображатиметься майбутня акція. А методів для цього немає. Тому було вирішено відокремити центр прийняття рішень (контролер) від бізнес-логіки (модель).

ухвалення рішень (контролер) від бізнес-логіки (модель).

Крім ізолювання видів від логіки програми, концепція MVC істотно зменшує складність великих додатків. Код виходить набагато структурованим, і, тим самим, полегшується підтримка, тестування та повторне використання рішень.

Розуміючи концепцію MVC, ти як розробник усвідомлюєш, де потрібно додати сортування списку книг:

• На рівні запиту до бази даних.

• На рівні бізнес-логіки (моделі).

• На рівні бізнес-логіки (контролер).

• У виставі — на стороні клієнта.

І це не риторичне питання. Ось прямо зараз і подумай: де і чому потрібно додати код сортування списку книг.

1.4.4. Класична модель MVC

Взаємодія між компонентами MVC реалізується по-різному у веб-додатках та мобільних додатках. Це відбувається через те, що веб- додаток є короткоживучим, обробляє один запит користувача і завершується, а мобільний додаток обробляє багато запитів без перезапуску.

У веб - застосунках зазвичай використовується "пасивна" модель, а в мобільних додатках - "активна". Активна модель, на відміну від пасивної, дозволяє підписуватись і отримувати повідомлення про зміну в ній. У випадку з веб- застосунками цього не потрібно.

Приблизно так виглядає взаємодія компонентів у різних моделях

1. У мобільних програмах (активна модель) активно використовуються події та механізм підписки на події. За такого підходу view (вид) підписується зміни моделі. Потім, коли відбувається якесь подія (наприклад, користувач натискає кнопку), викликається контролер. Він дає і моделі команду зміну даних.

Якщо якісь дані змінилися, модель генерує подію про зміну цих даних. Всі view, які підписалися на цю подію (для яких важлива зміна саме цих даних), отримують цю подію та оновлюють дані у своєму інтерфейсі.

2. У веб - додатках все організовано трохи інакше. Основна технічна відмінність - це те, що клієнт не може отримувати повідомлення з боку сервера з ініціативи сервера.

Тому контролер у веб-застосунку зазвичай не надсилає view будь-які повідомлення, а віддає клієнту нову сторінку, яка технічно є новим view або навіть новим клієнтським додатком (якщо одна сторінка нічого не знає про іншу).

Нині ця проблема частково вирішена за допомогою таких підходів:

• Регулярне опитування сервера щодо зміни важливих даних (раз на хвилину або частіше).

• WebSocket'и дозволяють клієнтку підписуватись на повідомлення сервера.

• Web-push-повідомлення з боку сервера.

• Протокол HTTP/2 дозволяє серверу ініціювати надсилання повідомлень клієнту.

Висновки до першого розділу.

В першому розділі кваліфікаційної роботи було проведено аналіз та дослідження на основі класифікацій найбільш вагомих шаблонів, які мають назву патерни**.**

Патерни слугують задачам підвищення реалізації стандартизованих задач для втілення в короткій і легко застосовної на практиці формі підвищення гнучкості та ступеня повторного використання вже існуючих програм.

В розділі розглянуто види та складові категорій патернів: породжуючі патерни (Creational), структурні патерни проектування (Structural), поведінкові патерни проектування (Behavioral). На основі аналізу було зроблено висновки, що шаблонний патерн , який працює на рівні класів, використовують для розширення частини лгоритму. Стратегія використовує делегування для зміни «на льоту» алгоритму, який виконується. Стратегія дозволяє змінювати логіку окремих об'єктів.

На основі аналізу запропоновані найбільш відповідні види шаблонів для різного типу задач, розглянута концепція архітектурного патерну Model - View – Controller..

РОЗДІЛ 2. МОДЕЛІ ТА АРХІТЕКТУРНІ РІШЕННЯ РОЗРОБКИ ДОДАТКІВ

2.1. Модель технології клієнт-сервер

Клієнт-серверні технології пройшли кілька етапів свого розвитку. Існують різні моделі цієї технології засновані на поділі структури на три компоненти:

- інтерфейс з користувачем (введення і відображення даних);
- прикладний компонент (запити, події, правила, процедури та функції предметної області);
- функції керування файловою системою (базою даних).

Архітектура «клієнт-сервер» ділить додаток на дві частини: клієнтську частину і серверну частину. Клієнтська частина може надати інтерфейс для користувачів і поширюватися серед них. Серверну частину можна зберігати та запускати на власному сервері компанії (рис. 2.1.). Клієнтська програма може відображати інформацію і використовуватись для передачі інформації на сервер для пошуку, наприклад, заголовка книги. Серверна частина архітектури забезпечує центральну функціональність. Сервер приймає ці запити, виконує необхідне завдання та, відповідно, повертає будь-які результати клієнту.

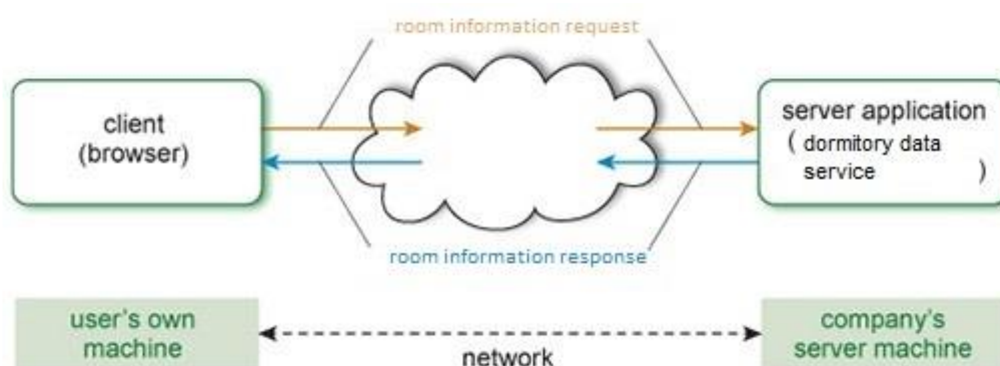


Рисунок 2.1 – Схема клієнт-серверної архітектури

Основним поняттям в архітектурі клієнт-сервер є поняття взаємодії. Модель клієнт-серверної взаємодії, це розподіл обов'язків між клієнтом та сервером, а основним поняттям в архітектурі клієнт-сервер є поняття взаємодії.

2.2. Архітектурні рішення клієнт - серверної технології

Архітектура клієнт-сервер є архітектурним шаблоном програмного забезпечення.

Існує три види клієнт-серверної архітектури: дворівнева архітектура, трирівнева архітектура та багаторівневі архітектури. Розглянемо ці види архітектури:

1. *Дворівнева клієнт-серверна архітектура.*

Дворівнева клієнт-серверна архітектура передбачає взаємодію двох програмних модулів – клієнтського та серверного.

Важливим фактором цієї архітектурі є Поняття часу доступу, оскільки о Оскільки обмін даними, запитами між пристроями, збереження і обробки даних здійснюється постійно, то важливим фактором цієї архітектурі є поняття часу.

2. *Трирівнева клієнт-серверна архітектура*.

Цей вид архітектури передбачає відділення прикладного рівня від управління даними. На окремому програмному рівні зосереджується прикладна логіка застосування - сервер застосувань. Він є третім рівнем клієнт-серверної архітектури і обслуговує запити сервера застосувань.

3. Багаторівневі архітектури

Багаторівнева архітектура клієнт-сервер передбачає пряме продовження розділення застосувань на рівні інтерфейсу користувача, компонентів обробки і даних.

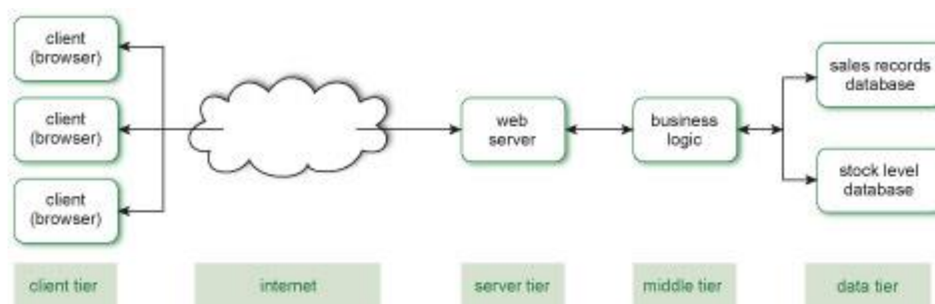


Рисунок 2.2- Рисунок 2.2 Схема багаторівневої архітектури

Особливістю цієї архітектури є розміщення логічно різних

компонентів на різних машинах. У сучасній архітектурі розподіл на клієнти і сервери відбувається способом, який називається *горизонтальним розподілом.* При такому типі розподілу клієнт або сервер може містити фізично розділені частини логічно однорідного модуля, причому робота з кожною з частин може відбуватися незалежно. Це робиться для вирівнювання навантаження.

У веб – додатках , які є розподіленими за стосунками, клієнтом виступає браузер, а сервером – веб - сервер. В такій архітектурі клієнт не залежить від конкретної операційної системи користувача, тому веб- додатки є міжплатформовими сервісами.

Розглянемо модель роботи веб-додатку

Веб - додаток отримує запит від клієнта, виконує обчислення, формує веб-сторінку і відправляє її клієнтові за допомогою протоколу HTTP по мережі Слід зауважити, веб - додаткок може бути клієнтом бази даних або стороннього веб –додатку, розташованого на іншому сервері. Прикладом такого типу додатку є система управління вмістом статей Вікіпедії. Учасники Вікіпедії можуть брати участь у створенні мережевої енциклопедії, використовуючи для цього браузери своїх операційних систем (Microsoft Windows, GNU/Linux) без завантаження додатку (модулів для роботи з базою даних статей).

Для створення вебзастосунків використовуються різноманітні серверні технології та мови програмування(таблиця 2.1).

Таблиця 2.1.

Назва	**Ліцензія**	**Вебсервер**
ASP	власницька	спеціалізований
ASP.NET	власницька	спеціалізований
Java	вільна	безліч, зокрема вільних
Groovy	вільна	практично будь-який

Назва	Ліцензія	Вебсервер
Perl	вільна	практично будь-який
PHP	вільна	практично будь-
Python	вільна	практично будь-який
Ruby	вільна	практично будь-який

2.3. Методи аналізу структури програм.

Серед методів аналізу структури програм можна виділити процес валідації *Валідація вимог* – це процес виявлення помилок в представленні сценарних вимог. Він ітераційний та складається з наступних кроків:

1) формалізований опис вимог у вигляді сценаріїв;

2) створення виконуваної моделі вимог;

3) створення спеціальних сценаріїв для валідації вимог;

4) застосування валідаційних сценаріїв до моделі вимог;

5) оцінювання результатів поведінки моделі вимог;

6) перевірка умов завершення процесу валідації, при виявленні будь-яких неточностей проводиться повторення кроків, починаючи з кроку 2.

При виконанні сценаріїв виникають помилкові ситуації, за яких поведінка системи стає недетермінованою. З цією метою (з метою виявлення ризиків або помилок) проводиться контроль покриття сценаріїв у моделі вимог валідаційними сценаріями. Створюється модель помилок, яка покриває модель вимог системи і містить типові помилки, використовувані при виведенні сценаріїв. Складова частина валідації вимог у сценаріях – визначення класів еквівалентності вхідних та вихідних даних, використовуваних і для синтезу сценаріїв. Вхідна інформація для синтезу сценаріїв – сценарна модель – задається мовою взаємодії (рис.2.3). Ця інформація використовується при генерації додаткових сценаріїв з метою покращення процесу валідації, автоматичного синтезу сценаріїв моделі та одержання моделі поведінки системи. Модель перевіряє неповноту

початкових вимог або суперечливості у вимогах за допомогою тестів та моделі помилок.

Автоматичний синтез засновано на наступних процедурах:

- валідація вимог шляхом виконання валідаційних сценаріїв;
- додавання перевірених сценаріїв до набору валідаційних сценаріїв та їх використання як вхідних даних для синтезу;
- пошук помилок у сценаріях та перевірка різних композицій сценаріїв

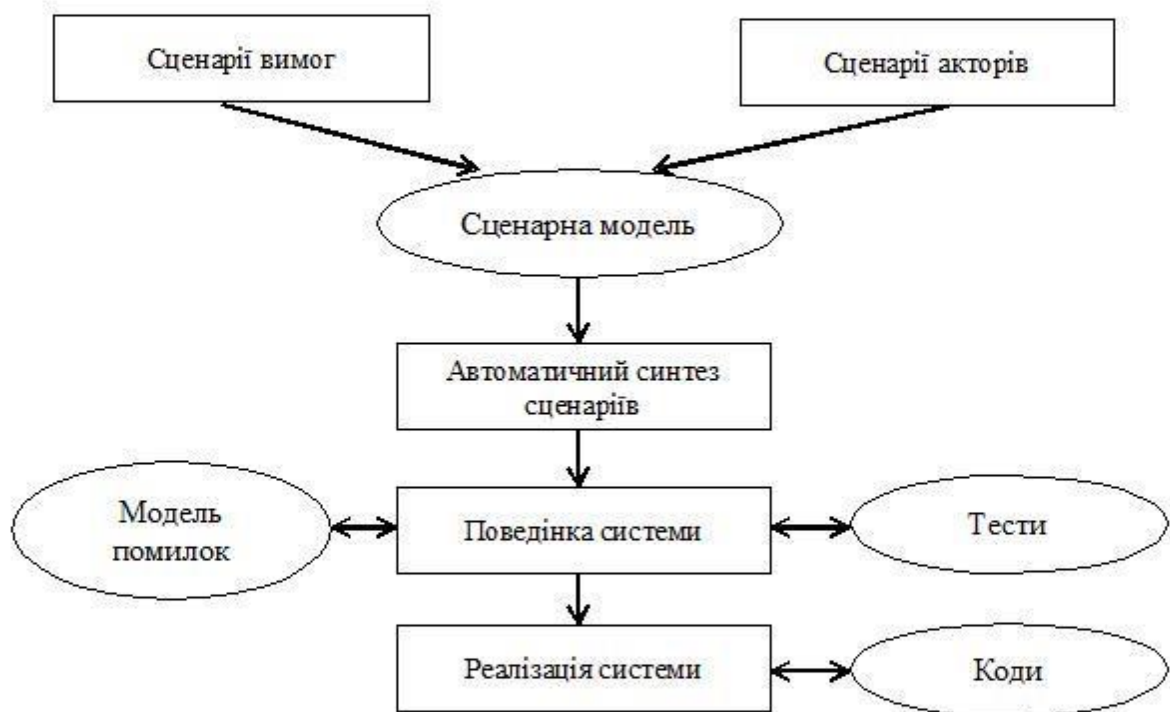


Рисунок 2.3 - Валідація сценаріїв вимог до системи

Також до методів доведення правильності програм належать і *методи аналізу структури програм*, які полягають у інспекції незалежними експертами за участю самих розробників. Вони перевіряють повноту, цілісність, однозначність та несуперечливість визначень у програмі. Сутність інспекції полягає у тому, що експерти намагаються поглянути на програму «з боку», піддати її всебічному критичному аналізу, розглянути словесні пояснення розробників про способи її розроблення. Мета інспекції – виявлення помилок у логіці та у програмі в статиці (статичний аналіз). До методів аналізу структури програм належать наскрізний контроль, який дозволяє виявити

помилки при багаторазовому перегляді коду; метод простого структурного аналізу; метод аналізу потоків даних; метод символьної перевірки.

Існує три рівні специфікації ПЗ. Це вимоги користувача, системні вимоги та специфікації структури ПЗ. Вимоги користувача найбільш узагальнені, специфікація структури найбільш детальна. Формальні математичні специфікації знаходяться між системними вимогами та специфікацією структури. Вони не містять деталей реалізації системи, але повинні представляти її повну математичну модель. На рис.2.4 показано етапи розроблення специфікації ПЗ та їх взаємозв'язки із процесом проектування. На рис.2.5 показано, що розроблення специфікації та проектування можуть виконуватись паралельно.

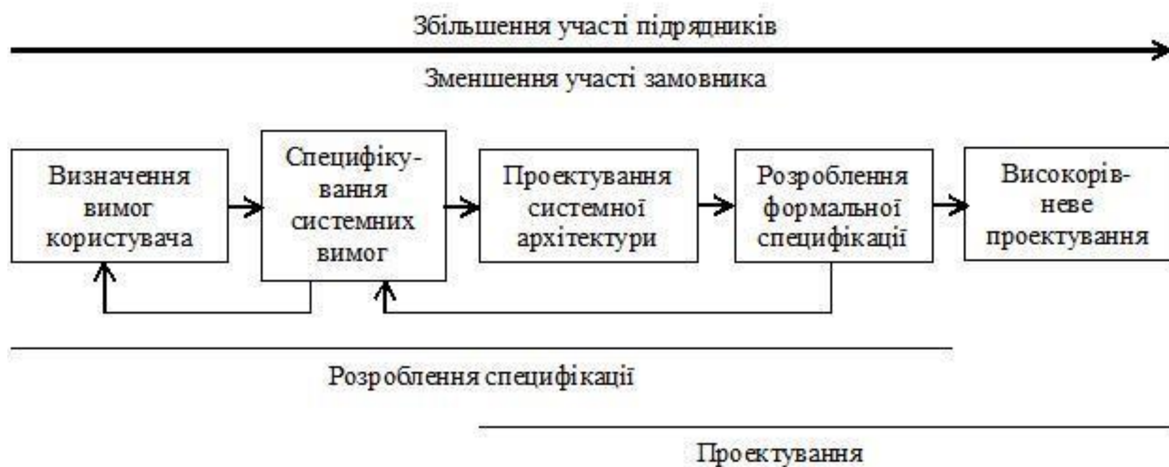


Рисунок 2.4. - Розроблення специфікації та проектування

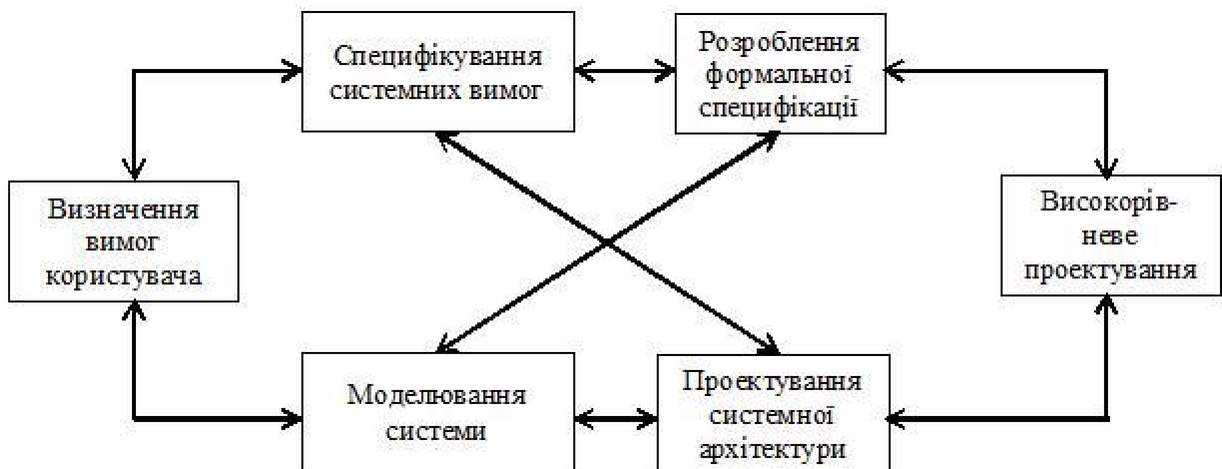


Рисунок 2.5- Створення формальної специфікації

Створення формальної специфікації вимагає детального аналізу системи, який дозволяє виявити помилки та невідповідності у специфікації неформальних вимог. Ця можливість виявлення помилок – найважливіший аргумент для використання формальної специфікації, хоча її розроблення та аналіз вимагають додаткових витрат. При звичайному процесі розроблення ПЗ вартість атестації системи складає близько 50% всієї вартості розроблення, а вартість проектування і реалізації системи вдвічі більше вартості розроблення специфікації. При використанні формальної специфікації вартості розроблення специфікації і реалізації системи практично однакові, а вартість атестації значно знижується, оскільки в процесі розроблення формальної специфікації виявляються і усуваються не доопрацювання у вимогах і виключається переробка системи на останніх стадіях її створення.

Існує *два основних підходи до розроблення формальної специфікації*:

1) алгебраїчний підхід, при якому система описується в термінах операцій та їх відношень;

2) підхід, орієнтований на моделювання, при якому модель системи будується з використанням математичних конструкцій, таких як множини і послідовності, а системні операції визначаються тим, як вони змінюють стан системи.

2.4. Інструментарії побудови інтерфейсів на базі шаблонів

При розробленні клієнт- сервреного додатку завжди потрібно розробляти інтерфейс, який буде зручним, практичним і зрозумілим користувачу.

Користувацький інтерфейс (від англійського user interface) в комп'ютерній практиці має назву інтерфейс - UI. Екрани, сторінки, кнопки та іконки, призначені для взаємодії з пристроєм і є користувацьким інтерфейсом.

На сьогодні термін «користувацький інтерфейс» означає скоріш за все

графічний інтерфейс користувача .

Можна угрупувати всі види у чотири групи інтерфейсів:

1. GUI - графічний інтерфейс користувача . Користувач здійснює дії по вводу через клавіатуру і мишу, а комп'ютер забезпечує графічний висновок на моніторі.
2. WUI - веб-інтерфейс. Користувач взаємодіє з веб-сайтом або додатком через браузер. В реалізації використовують для цього Java, JavaScript, AJAX, Adobe Flex, Microsoft. NET або аналогічні технології..
3. Адміністративні веб-інтерфейси - панеллю керування. Існують для роботи з серверами і віддаленими комп'ютерами
4. Сенсорні екрани - дисплеї, які виконують введення інформації для дій через торкання пальцями або стилусом.

При розробки інтерфейсів використовують два поняття UX і UI дизайн, які відносяться до напрямків розробки веб-інтерфейсів.

Дизайн користувацького інтерфейсу це

1.Візуальний дизайн –сайт, особливістю якого є бренд.

2. Дизайн взаємодіїї - взаємодія користувача з вашим сайтом.

UX (User eXperience), означає «досвід взаємодії» (User eXperience) . Інформаційна архітектура , проектування взаємодії, графічний дизайн і контент все це відноситься до цього виду інтерфейсу.

UX дизайнери - це архітектори марко взаємодій, а UI дизайнери - займаються деталями. UX- дизайн розробляється UX- дизайнером. Для розробки інтерфейсу використовують комплексний підхід взаємодії користувача з інтерфейсом мобільного додатку , веб-сайту або будь-якої іншої програми.

Команда компанії Apple Computer продовжила свій розвиток і розширення саме на базі ідеї GUI.

Розробники UI працюють над веб-сайтами, додатками, гаджетами (wearables) .

Розробка UX - інтерфейсу включає наступній алгоритм :

1. Користувачі   шукають продукт компанії.
2. Дії користувача при взаємодії з інтерфейсом.
3. Почуття при виконанні свого завдання .
4. Враження від взаємодії в цілому.

На рисунку 1.7. представлені складові концепції UX-дизайну**.**

Рисунок 1.7.- Складові концепції UX-дизайну

Візуальна ієрархія пояснює користувачу що робити, щоб досягти своєї мети. Саме це і реалізують UI дизайнери.

Ієрархія дає можливість   користувачам зрозуміти що вони можуть зробити в будь-який даний момент часу . Шаблонів, які вже знайомі користувачам підказують   подальший напрямок.

Правила UI дизайну:

- елементи інтерфейсу повинні бути структуровані ;
- логічно пов'язані елементи потрібно об'єднувати в групи, наприклад, меню , форми;
- всі елементи інтерфейсу повинні бути вирівняні для зручності користувачів;
- єдине стильове   оформлення елементів;
- присутність вільного простору.

Дотримуючись цих правил сайт стане конкурентоспроможним

Засоби побудови UI, орієнтовані на форми введення даних з прив'язкою до реляційних систем керування базами даних.. Інструментарій побудови інтерфейсів дозволяють виконувати навігацію по такій формі і встановлювати прямий зв'язок між елементами управління (текстовими полями, списками, кнопках - прапорцями, таблицями) і даними.

Прив’язка елемента управління декларування до джерела даних є сильною стороною даного методу.

Завдання планування і стилізації UI вирішуються за допомогою дизайнерів форм і спеціалізованих об'єктно-орієнтованих API. UI для управління поведінкою пропонують обробники подій. Типові представники цієї категорії інструментарію - Microsoft Access і Oracle Forms.

Технології побудови інтерфейсів для користувача на основні шаблонів дає багато можливостей для створення динамічних веб-інтерфейсів. Мови розмітки використовують для планування і структури UI, а прив'язка до даних здійснювалася за допомогою мов високого рівня (Java, C #, PHP, Python. Типовий представник цієї групи - технологія JavaServer Pages (JSP), бібліотека тегів якої JSP Standard Tag Library підтримує такі завдання, як: маніпуляція з XML- документами, цикли, умови, опитування СУБД (прив'язка до даних) і інтернаціоналізація (форматування даних).

Мова виразів JSP, як засіб прив'язки до даних пропонує зручну нотацію для роботи з об'єктами і властивостями додаткиу.

Для форматування даних використовують ASP, PHP, Struts, WebWork, Struts2, Spring MVC, Spyce і Ruby on Rails.

Створення UI - користувацьких інтерфейсів використовують об'єктно - орієнтованої моделі.

Сюди можна віднести середовища: Visual Basic, MFC, AWT, Swing, SWT, Delphi, Google Web Toolkit, Cocoa Touch UIKit, Vaadin.

Гібридна технологія відносно нова технологія в світі розробки UI. В

гібридній технології поряд з шаблонами і мовами виразів, застосовується об'єктний API (ASP.NET MVC, Apache Wicket, Apache Tapestry, Apache Click і ZK Framework) .

Після технологій на основі шаблонів з'явилися об'єктно-орієнтованими API з шаблонами (в разі гібридних підходів) або повністю замінювали їх (GWT і Vaadin).

2.4.1. Специфікування інтерфейсів

Великі системи, як правило, розбиваються на підсистеми, які розробляються незалежно одна від одної. Підсистеми можуть використовувати інші підсистеми, тому необхідною частиною процесу специфікування є визначення інтерфейсів підсистем. Якщо інтерфейси визначені і узгоджені, підсистеми можна розробляти незалежно одна від одної. Інтерфейс підсистеми часто визначається як набір абстрактних типів даних та об'єктів, при цьому лише через інтерфейс доступні опис даних та операції над ними. Тому специфікацію інтерфейсу підсистеми можна розглядати як об'єднання специфікацій компонентів, що в результаті і складе опис інтерфейсу підсистеми.

Точні специфікації інтерфейсів підсистем необхідні для розробників, які пишуть програмний код, який звертається до сервісів інших підсистем. Специфікації інтерфейсів містять інформацію про те, які сервіси доступні в інших підсистемах і як одержати до них доступ. Ясний та однозначний інтерфейс підсистем зменшує ймовірність помилок у взаємовідносинах між ними.

Алгебраїчний підхід початково був розроблений для опису інтерфейсів абстрактних типів даних, де типи даних визначаються швидше специфікаціями операцій над даними, ніж способом представлення самих даних. Це дуже схоже на визначення класів об'єктів. Алгебраїчний підхід до формальних специфікацій визначає абстрактний тип даних в термінах операцій над даними.

Структуру специфікації об'єкту показано на рис.2.2.

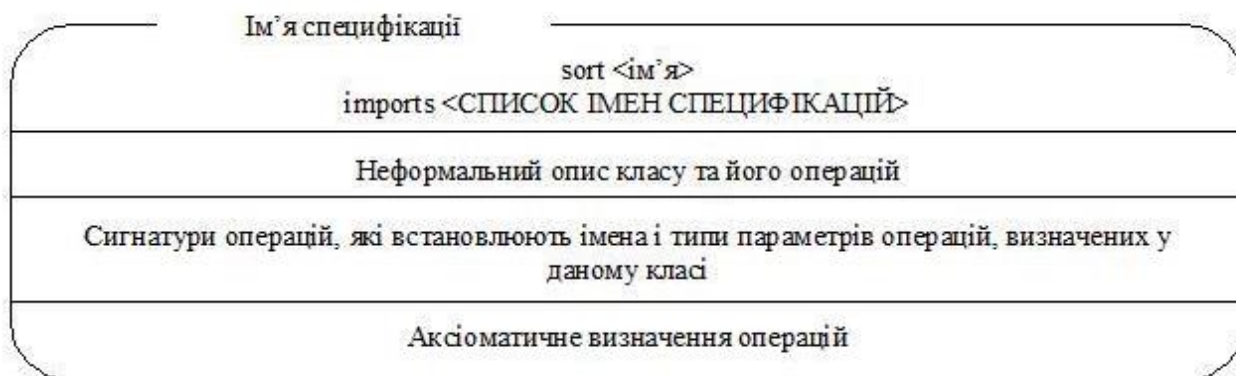


Рисунок 2.2 - Структура алгебраїчної специфікації

Специфікація об'єкту складається з 4-х компонентів:

- вступ, де оголошується клас об'єктів, а також відбувається включення оголошень імпорту з іменами специфікацій для інших класів;
- описова частина, в якій неформально описуються операції, асоційовані з класом, із забезпечення однозначного синтаксису та семантики операцій;
- частина сигнатур, в якій визначається синтаксис інтерфейсу об'єктного класу або абстрактного типу даних (описуються імена операцій, кількість та типи їх параметрів, класи вихідних результатів операцій);
- частини аксіом, де визначається семантика операцій завдяки створення ряду аксіом, які характеризують поведінку абстрактного типу даних; ці аксіоми зв'язують операції створення об'єктів класу з операціями, які перевіряють їх значення.

Процес розроблення формальної специфікації інтерфейсу підсистеми включає наступні дії:

1) структурування специфікації – представлення неформальної специфікації у вигляді множини абстрактних типів даних або об'єктних класів;

2) іменування специфікацій – задаються імена кожної специфікації абстрактного типу, визначаються параметри специфікацій та імена класів;

3) визначення операцій – на основі списку виконуваних інтерфейсом функцій для кожної специфікації визначається зв'язаний з нею набір операцій; операції над абстрактним типом даних належать до операцій конструювання, які створюють або змінюють об'єкти класу, або до операцій перевірки, які

повертають атрибути класу; добрим емпіричним правилом для написання алгебраїчної специфікації є створення аксіом для кожної операції конструювання із застосуванням всіх операцій перевірки - це означає, що, якщо є m операцій конструювання та n операцій перевірки, то повинно бути визначено m*n аксіом.

4) неформальна специфікація операцій – написання неформальної специфікації для кожної операції із вказівками , як операції впливають на клас;

5) визначення синтаксису і параметрів операцій – це частина сигнатури формальної специфікації;

6) визначення аксіом – визначення семантики операцій шляхом опису умов, які повинні виконуватись для різних комбінацій операцій.

Висновки до другого розділу.

Висновки до другого розділу.

В другому розділі кваліфікаційної роботи були розглянути питання: моделі та архітектурні рішення розробки клієнт-серверного додатка, інструментарії побудови інтерфейсів на базі шаблонів, специфікування інтерфейсів .

Перед розробниками клієнт - серверного важлива схема її внутрішньої організації. В моделі – клієнт-сервер, сервіси реалізуються у вигляді процедур в окремих модулях додатку . Сервер відповідає за рівень даних, а інтерфейс користувача реалізується на стороні клієнта.

При розробці клієнт-серверного додатку важливо вибрати патерн, який повністю задовольнить всі потреби розробників. Було розглянуто моделі та архітектурні рішення розробки клієнт-серверного додатка. При розробці клієнт-серверного додатку важливо вибрати патерн, який повністю задовільнить всі потреби розробників та самого продукту . Кожен патернів краще використовувати в тій чи іншій ситуації. Патерн не є класом і це не бібліотека, немає єдиного вірного патерну. Подальшим розвитком

компонентної архітектури застосувань є сервісно-орієнтована архітектура, зокрема архітектура веб - сервісів.

При розробці додатку завжди присутнє завдання - розробити інтерфейс, який буде зручним, практичним та зрозумілим по відношенню до користувача інтерфейсу. В розділі була проаналізована логіка розробки та обрання виду інтерфейсу, розглянуто інструментарії побудови інтерфейсів на базі шаблонів.

РОЗДІЛ 3. РОЗРОБКА ПРОГРАМНИХ МОДУЛІВ ПАТЕРНІВ

3.1.Мови програмування.

3.1. 1. Мова програмування Swift

Мова програмування Swift це багатопарадигмова, безпечна, швидка, компільована та інтерактивна мова програмування.
Основним застосуванням Swift є розробка користувацьких застосунків для macOS, iOS, tvOS, watchOS.

Мова Swift дозволяє програмісту вирішувати поставлені завдання в декількох різних моделях, таких, як функціонального програмування, процедурного та об'єктно-орієнтованого програмування.
Код на Swift скомпільовано і оптимізований, щоб отримувати максимальну віддачу від сучасного обладнання.
Swift виключає пласт поширених програмних помилок за допомогою застосування сучасних програмних патернів:

• Змінні завжди ініціалізовані до того, як будуть використані.
• Індекси масивів завжди перевіряються на out-of-bounds помилки.
• Цілі числа перевіряються на переповнення.
• Автоматичне управління пам'яттю.
• Обробка помилок дозволяє здійснювати контрольоване відновлення від непередбачених помилок

Swift поєднує патерн - матчінг і вивод типів з сучасним синтаксисом, дозволяючи складним ідеям бути вираженим коротко ,а це дає можливість не тільки простіше писати код, але і читати його і підтримувати.

Розглянемо проклад програмного коду , написаний мовою Swift:

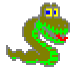
```
/* багаторядкові коментарі
        /* можуть бути вкладені! */
          тому ви можете поміщати коментарі в блоки
         */
/* оголошення змінних у Swift починається із службового слова "var"
```

```
*/
        var explicitDouble: Double = 70
            // якщо тип змінної не вказано, Swift обере його автоматично на
        основі    початкового значення
        var implicitInteger = 70
        var implicitDouble = 70.0
            var 国 = "日本"
      /* оголошення константи у Swift починається із службового слова "let",
після якого повинні  бути ім'я змінної, тип змінної та початкове значення */
let numberOfBananas: Int = 10
        // якщо тип константи не вказано, Swift обере його автоматично на
основі початкового значення
                let numberOfApples = 3
                let numberOfOranges = 5
     // значення змінних та констант можуть бути вставлені у рядки (змінні
типу string) наступним чином
              let appleSummary = "I have \(numberOfApples) apples."
             let fruitSummary = "I have \(numberOfApples + numberOfOranges)
pieces of fruit."
   // оголошення масиву даних
 var fruits = ["mango", "kiwi", "avocado"]
  // приклад оголошення  "словника" (dictionary ) з 4 елементів, кожен із яких
містить ім'я та вік
          let people = ["Anna": 67, "Beto": 8, "Jack": 33, "Sam": 25]
  // використовуючи можливості мови Swift, ми надрукуємо обидва значення
в єдиному циклі
    for (name, age) in people {
          print("\(name) is \(age) years old.")

}
```

```
// оголошення методів починається із службового слова "func"
// тип результату описується після "->"
func sayHello(personName: String) -> String {
            let greeting = "Hello, " + personName + "!"
   return greeting
}
// як вивести в консолі словосполучення "Hello, Jane!", використовуючи вище описаний метод
print(sayHello("Jane"))
```

3.1.2. Мова програмування Python



Python — інтерпретована об'єктно-орієнтована мова програмування високого рівня зі суворою динамічною типізацією, високою гнучкістю та динамічністю, розроблена в 1990 році Гвідо ван Россумом.

У 2023 році, найпопулярнішою мовою програмування серед IT- спеціалістів згідно рейтингу IEEE Spectrum стала Python. Рейтинг 2023 року охоплював 59 мов програмування .

Серед основних переваг можна назвати :

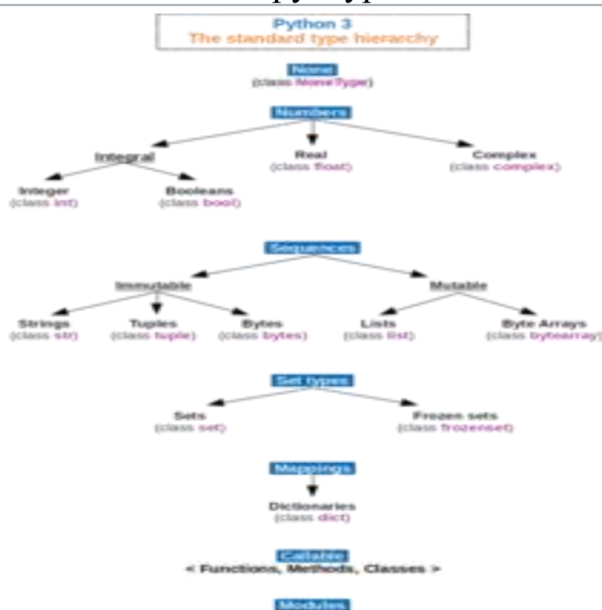
- чистий синтаксис;
- переносність програм;
- стандартний дистрибутив (має модуль для розробки графічного інтерфейсу);
- використання Python в діалоговому режимі ;
- стандартний дистрибутив (IDLE написане мовою Python);
- зручний для розв'язання математичних проблем (у діалоговому режимі може використовуватися як потужний калькулятор) ;
- відкритий код (можливість редагувати його іншими користувачами).

Python підтримує модулі та пакети модулів, що сприяє модульності та повторному використанню коду. Інтерпретатор Python та стандартні бібліотеки доступні як у скомпільованій, так і у вихідній формі на всіх

основних платформах. В мові програмування Python підтримується кілька парадигм програмування, зокрема: об’єктно - орієнтована ,процедурна, аспектно – орієнтована та функціональна.

Python має ефективні структури даних високого рівня та простий, але ефективний підхід до об'єктно – орієнтованого програмування. Елегантний синтаксис Python, динамічна обробка типів роблять її ідеальною для написання скриптів та швидкої розробки прикладних програм у багатьох галузях на більшості платформ.

Типи й структури даних



Python підтримує динамічну типізацію, коли тип змінної визначається під час виконання. З базових типів слід зазначити підтримку цілих чисел довільної довжини. Python має багату бібліотеку для роботи з рядками кодованими в юні коді .

Подібно Ліспу та Прологу в режимі відлагодження, інтерпретатор Python має інтерактивний режим роботи, при якому введені з клавіатури вирази відразу ж виконуються, а результат виводиться на екран. Цей режим цікавий не тільки новачкам, але й досвідченим програмістам, які можуть протестувати в інтерактивному режимі будь- який фрагмент коду, перш ніж використовувати його в основній програмі, або просто використовувати як калькулятор з великим набором функцій.

3.2. Розробка програмного модуля патерну Одинак.

Клас має лише один екземпляр, та надає глобальну точку доступу до нього.

Застосування

- Коли в програмі повинен бути єдиний екземпляр якого-небудь класу, доступний усім клієнтам (наприклад, спільний доступ до бази даних з різних частин програми).
- Одинак приховує від клієнтів всі способи створення нового об'єкта, окрім спеціального методу. Цей метод або створює об'єкт, або віддає існуючий об'єкт, якщо він вже був створений.
- Коли ви хочете мати більше контролю над глобальними змінними.

На відміну від глобальних змінних, Одинак гарантує, що жоден інший код не замінить створений екземпляр класу, тому ви завжди впевнені в наявності лише одного об'єкта

Модуль «Підключення до бази даних».

У цьому прикладі роль Одинака грає клас підключення до бази даних.

Цей клас не має публічного конструктора, тому єдиним способом отримання його об'єкта є виклик методу getInstance. Цей метод збереже перший створений об'єкт і повертатиме його в усіх наступних викликах.

```
1. // Клас одинака визначає статичний метод `getInstance`, котрий
   // дозволяє клієнтам повторно використовувати одне і теж
   // підключення до бази даних по всій програмі.
class Database is
       // Поле для зберігання об'єкта-одинака має бути оголошено
      // статичним.
   private static field instance: Database
           // Конструктор одинака завжди повинен залишатися приватним,
          // аби клієнти не могли самостійно створювати екземпляри
           // цього класу через оператор `new`.
   private constructor Database() is
           // Тут може жити код ініціалізації підключення до
           // сервера баз даних.
```

```
// ...
```

2. Модуль конструктора підключення до точки доступу класу.

```
// Головний статичний метод одинака служить альтернативою
    // конструктору і є точкою доступу до екземпляра цього
    // класу.
  public static method getInstance() is
    if (Database.instance == null) then
      acquireThreadLock() and then
          // Про всяк випадок, ще раз перевіримо, чи не
          // було створено об'єкт в іншому потоці, поки
          // даний потік чекав на звільнення блокування.
        if (Database.instance == null) then
          Database.instance = new Database()
    return Database.instance
```

3.

```
      // будь-який клас одинака повинен мати якусь
      // корисну функціональність, яку клієнти будуть запускати
      // через отриманий об'єкт одинака.
  public method query(sql) is
      // Усі запити до бази даних проходитимуть через цей
      // метод. Тому є сенс помістити сюди якусь логіку
      // кешування.
        // ...

class Application is
  method main() is
    Database foo = Database.getInstance()
    foo.query("SELECT ...")
    // ...
    Database bar = Database.getInstance()
    bar.query("SELECT ...")
```

```
// Змінна "bar" містить той самий об'єкт, що і змінна
// "foo".
```

3.3. Розробка програмного модуля патерну Абстрактна фабрика

В цьому прикладі абстрактна фабрика створює крос - платформові елементи інтерфейсу і стежить за тим, щоб вони відповідали обраній операційній системі. Крос – плат формова програма може відображати одні й ті самі елементи інтерфейсу по-різному, в залежності від обраної операційної системи. Важливо, щоб у такій програмі всі створювані елементи завжди відповідали поточній операційній системі. Іноді, при розробки модуля може бути випадок, коли програма, запущена на Windows, раптом почала показувати чек-бокси в стилі macOS.

Абстрактна фабрика оголошує список методів, які клієнтський код буде використовувати для отримання різновидів елементів інтерфейсу. Конкретні фабрики відносяться до різних операційних систем і створюють елементи, сумісні з цією системою.

Програма на самому початку визначає фабрику, що відповідає поточній операційній системі. Потім створює цю фабрику та віддає її клієнтському коду. У подальшому, щоб виключити несумісність продуктів, що повертаються, клієнт працюватиме тільки з цією фабрикою.

Клієнтський код не залежить від конкретних класів фабрик чи елементів інтерфейсу. Він спілкується з ними через загальні інтерфейси, не залежачи від конкретних класів фабрик чи елементів користувацького інтерфейсу.

Таким чином, щоб додати до програми нову варіацію елементів інтерфейсу (наприклад, для підтримки Linux), не потрібно змінювати клієнтський код. Достатньо створити ще одну фабрику, що виготовляє ці елементи.

1. Модуль розробки спільного інтерфейсу.

```
/ Цей патерн передбачає, що ви маєте кілька сімейств продуктів,
// які знаходяться в окремих ієрархіях класів (Button/Checkbox).
// Продукти одного сімейства повинні мати спільний інтерфейс.

interface Button is
    method paint()

// Сімейства продуктів мають однакові варіації (macOS/Windows).
class WinButton implements Button is
    method paint() is
        // Відобразити кнопку в стилі Windows.

class MacButton implements Button is
    method paint() is
        // Відобразити кнопку в стилі macOS.

interface Checkbox is
    method paint()

class WinCheckbox implements Checkbox is
    method paint() is
        // Відобразити чекбокс в стилі Windows.

class MacCheckbox implements Checkbox is
    method paint() is
        // Відобразити чекбокс в стилі macOS.

// Абстрактна фабрика знає про всі абстрактні типи продуктів.
interface GUIFactory is
    method createButton():Button
    method createCheckbox():Checkbox
```

```
// Кожна конкретна фабрика знає лише про продукти своєї варіації
// і створює лише їх.
class WinFactory implements GUIFactory is
    method createButton():Button is
        return new WinButton()
    method createCheckbox():Checkbox is
        return new WinCheckbox()

// Незважаючи на те, що фабрики оперують конкретними класами,
// їхні методи повертають абстрактні типи продуктів. Завдяки
// цьому фабрики можна заміняти одну на іншу, не змінюючи
// клієнтського коду.
class MacFactory implements GUIFactory is
    method createButton():Button is
        return new MacButton()
    method createCheckbox():Checkbox is
        return new MacCheckbox()

// Для коду, який використовує фабрику, не важливо, з якою
// конкретно фабрикою він працює. Всі отримувачі продуктів
// працюють з ними через загальні інтерфейси.
class Application is
    private field factory: GUIFactory
    private field button: Button
    constructor Application(factory: GUIFactory) is
        this.factory = factory
    method createUI()
        this.button = factory.createButton()
    method paint()
        button.paint()



// Програма вибирає тип конкретної фабрики й створює її
```

```
// динамічно, виходячи з конфігурації або оточення.
class ApplicationConfigurator is
    method main() is
        config = readApplicationConfigFile()

        if (config.OS == "Windows") then
            factory = new WinFactory()
        else if (config.OS == "Mac") then
            factory = new MacFactory()
        else
            throw new Exception("Error! Unknown operating system.")
        Application app = new Application(factory)
```

3.4. **Розробка програмного модуля патерну Спостерігач**

У цьому прикладі Спостерігач дає змогу об'єкту текстового редактора сповіщати інші об'єкти про зміни свого стану. Для додавання до програми нових підписників не потрібно змінювати класи видавців, допоки вони працюють із підписниками через загальний інтерфейс.

1. *Модуль «Базовий клас-видавець»*

```
// Базовий клас-видавець. Містить код керування підписниками та
// надсилання їм сповіщень.
class EventManager is
    private field listeners: hash map of event types and listeners

    method subscribe(eventType, listener) is
        listeners.add(eventType, listener)

    method unsubscribe(eventType, listener) is
        listeners.remove(eventType, listener)

    method notify(eventType, data) is
        foreach (listener in listeners.of(eventType)) do
```

```
            listener.update(data)

// Конкретний клас-видавець, що містить цікаву для інших
// компонентів бізнес-логіку. Ми могли б зробити його прямим
// нащадком EventManager, але в реальному житті це не завжди є
// можливим (наприклад, якщо в класу вже є предок). Тому тут ми
// підключаємо механізм підписки за допомогою композиції.
class Editor is
    public field events: EventManager
    private field file: File

    constructor Editor() is
        events = new EventManager()

    // Методи бізнес-логіки, які сповіщають підписників про
    // зміни.
    method openFile(path) is
        this.file = new File(path)
        events.notify("open", file.name)

    method saveFile() is
        file.write()
        events.notify("save", file.name)
    // ...
```

2. Модуль «Загальний інтерфейс підписників»

```
// Загальний інтерфейс підписників. У багатьох мовах, що мають
// функціональні типи, можна обійтися без цього інтерфейсу та
// конкретних класів, замінивши об'єкти підписників функціями.
interface EventListener is
    method update(filename)
```

```
// Набір конкретних підписників. Кожен з них виконує якусь
// поведінку, реагуючи на сповіщення від видавця.
class LoggingListener implements EventListener is
    private field log: File
    private field message: string

    constructor LoggingListener(log_filename, message) is
        this.log = new File(log_filename)
        this.message = message

    method update(filename) is
        log.write(replace('%s',filename,message))

class EmailAlertsListener implements EventListener is
    private field email: string
    private field message: string

    constructor EmailAlertsListener(email, message) is
        this.email = email
        this.message = message

    method update(filename) is
        system.email(email, replace('%s',filename,message))


// Програма може сконфігурувати видавців та підписників, як
// завгодно, залежно від цілей та оточення.
class Application is
    method config() is
        editor = new Editor()

        logger = new LoggingListener(
            "/path/to/log.txt",
```

```
    "Someone has opened file: %s");
editor.events.subscribe("open", logger)

emailAlerts = new EmailAlertsListener(
    "admin@example.com",
    "Someone has changed the file: %s")
editor.events.subscribe("save", emailAlerts)
```

ВИСНОВКИ

Кваліфікаційна робота присвячена розробленню програмного забезпечення клієнт-серверного додатка з використанням патернів проектування та удосконаленню проектування додатків на основі порівняльної характеристики сучасних технологій.

В роботі була проаналізована класифікація найбільш вагомих шаблонів, які мають назву патерни , розглянуто види та складові у кожній групі категорій патернів, запропоновані найбільш відповідні види шаблонів для різного типу задач, розглянута концепція патерну Model -View – Controller.

Однією із поставлених задач було розглянути інструменти та середовища для розробки клієнт-серверного додатку.

В будь-якому проекті перед розробниками клієнт- сервреного додатку поставлено завдання розробити інтерфейс, який буде зручним , практичним і інтуїтивним та зрозумілим по відношенню до користувача інтерфейсу. В роботі проаналізована логіка розробки та обрання виду інтерфейсу, розглянуто інструментарії побудови інтерфейсів на базі шаблонів.

Було розглянуто моделі та архітектурні рішення розробки клієнт-серверного додатка. При розробці клієнт-серверного додатку важливо вибрати патерн, який повністю задовільнить всі потреби розробників та самого продукту . Патерн це не клас і не бібліотека, кожен патернів краще використовувати в тій чи іншій ситуації. Немає єдиного вірного патерну. В роботі розроблено модулі патернів Adapter, Observer та Singleton на різних мовах програмування та розглянуто їх вплив та внесок у розробку систем

СПИСОК ЛІТЕРАТУРНИХ ДЖЕРЕЛ

1. Методи представлення, збереження та аналізу даних інформаційних систем: колективна монографія / В. Ю. Щербань, С. М. Краснитський, Т. І. Астістова, В. М. Яхно. – Київ : ТОВ "Фастбінд Україна", 2023
2. Використання патернів проектування для розробки програмного забезпечення / Т. І. Астістова, Р. В. Мирець // Інформаційні технології в науці, виробництві та підприємництві : збірник наукових праць молодих вчених, аспірантів, магістрів кафедри комп'ютерних наук та технологій / за заг. наук. ред. В. Ю. Щербаня. – Київ : ТОВ "Фастбінд Україна", 2023. – С. 105-108.
3. Зеркалов Д.В., Полукаров Ю. О. Організація та управління безпекою життєдіяльності. Навч. посіб. — К.: Основа, 2011. – 236
4. Електронний ресурс. – Режим доступу: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design
5. Казарін О.В. Безпека програмного забезпечення комп'ютерних систем./ О.В. Казарін //Тернопіль , 2013.-212 с.
6. Огірко І., Паславська І., Пілат О. Інформаційна система оцінювання якості електронних видань. Вісник Львівського університету. Серія економічна. 2013. Вип. 49. С. 391–398.
7. JavaScript [Електронний ресурс] – Режим доступу до ресурсу: https://developer.mozilla.org/en-US/docs/Web/JavaScript
8. Огурцов В.В., Гриньов Д.В., Щербаков О.В. Основи веб та веб-дизайн, програмування на боці клієнта: лабораторний практикум з навчальної дисцип-ліни "Веб-технології та веб-дизайн" для студентів напряму підготовки 6.050101 "Комп'ютерні науки". Харків: ХНЕУ ім. С. Кузнеця, 2015. 208 с.
9. Hershel Harris, Bert Nicol. SQL/DS: IBM's First RDBMS // IEEE Annals of the History of Computing, Volume 35, Number 2, April–June 2013, pp. 69-71

10. Мова програмування Swift [Електронний ресурс] Режим доступу: https://metanit.com/swift/tutorial/pics/1.8.png
11. Демківська Т.І. Розробка інформаційно-пошукової системи підприємства з використанням концепції MVC// Т.І Демківська, О.М. Адвена//Інформаційні технології в науці, виробництві та підприємництві: зб. наук. праць молодих вчених, аспірантів, магістрів кафедри інформаційних технологій проектування. – К. : КНУТД, 2018. – С. 223– 226. – ISBN 978-966
12. Електронний ресурс. – Режим доступу: http://citforum.ua/ Портал аналітичної інформації в галузі інформаційних технологій CitForum
13. Інформаційно-комунікаційні технології. Веб - сайт ООН [Електронний ресурс]. – Режим доступу: http://www.un.org/development/ict/index.shtml
14. Електронний ресурс - Режим доступу: http://www.pil-network.com/#uk// Microsoft Partners in Learning
15. Електронний ресурс. – Режим доступу: http://www.informika.ru/ State Institute of Information Technologies and Telecommunications
16. Електронний ресурс. – Режим доступу: https://docs.microsoft.com/en-us/dotnet/architecture/microservices/microservice-ddd-cqrs-patterns/infrastructure-persistence-layer-design
17. Angular [Електронний ресурс] –Режим доступу https://angular.io/
18. Вікіпедія [Електронний ресурс]–Режим доступу: https://uk.wikipedia.org/wiki/IOS Вікіпедія
19. Android , [Електронний ресурс] -Режим доступу: https://uk.wikipedia.org/wiki/Android// Android
20. Грицюк Ю. І., Аналіз вимог до програмного забезпечення. Навчальний посібник/ Ю. І. Грицюк .// К.:КНЕУ ,2018, С.425
21. Філдінг Р.Т. Архітектурні стилі та проектування мережевих архітектур програмного забезпечення /Р.Т. Філдінг, Р.Н. Тейлор, М.С. Акерман, Д.С. Розенблюм. - К.: УНІВЕРСИТЕТ КАЛІФОРНІЇ, 2000. – 180 с.

22. Ситник Н.В. «Проектування баз і сховищ даних» - навч. посібн. – К.:КНЕУ, 2004 – 347с.
23. Аносов А. Критерії вибору СУБД при створенні інформаційних систем [Електронний ресурс]. – Режим доступа: http://www.google.ua (опубліковано у 2016р.)
24. Careerfoundry [Електронний ресурс] – Режим доступу до ресурсу: https://careerfoundry.com/en/blog/web-development/what-is-html-a-beginners-guide
25. Конолі Т., Бегг К., Страчан А. "Бази даних: Проектування, реалізація та супровід. Теорія і практика" – навч. посібник - К.:КНЕУ ,2000. – 1093
26. Резанова В.Г., Бартніцький С.Ю., Марченко В.І., Дослідження та розробка програмного забезпечення для малого автопідприємства // В.Г. Резанова , С.Ю. Бартніцький, В.І Марченко / Тези IV Міжнародної науково-практичної конференції «Мехатронні системи: інновації та інжиніринг - MSIE-2020» присвяченої 90-й річниці заснування КНУТД , «Мехатронні системи: інновації та інжиніринг - MSIE-2020» К. КНУТД, 22 жовтня 2020.- С.121-123
27. Zhelezniak, Mykola (2019).Encyclopedia of Modern Ukraine in the challenges oftoday [Електроннийресурс], Режим доступу до ресурсу :https://web.archive.org/web/20210127034837/http:/evu.encyclopedia.kyiv.ua/en/vol-11/encyclopedia-of-modern-ukraine-in-the-challenges-of-today/*The Encyclopedia Herald of Ukraine* (англ.). doi:10.37068/evu.11.1. Архів оригіналу за 27 січня 2021. Процитовано 23 грудня 2020.
28. Резанова В.Г., Марченко В.І., Архітектурні патерни при розробці клієнт-серверних додатків / В.Г. Резанова , В.І Марченко// Інформаційні технології в науці, виробництві та підприємництві: зб. наук. праць молодих вчених, аспірантів, магістрів кафедри інформаційних технологій проектування. – К. : Освіта України, 2020. – С.185-188

29. Visual Studio Code [Електронний ресурс] – Режим доступу до ресурсу: https://code.visualstudio.com/docs
30. Berkeley Extension [Електронний ресурс] – Режим доступу до ресурсу: https://bootcamp.berkeley.edu/resources/coding/learn-css/how-does-css-work/
31. Ситник В. Ф. Основи інформаційних систем: Навч. посібник. Вид. 2-ге, перероб. і доп. / В. Ф. Ситник, Т. А. Писаревська, Н. В. Єрьоміна, О. С. Краєва; За ред. В. Ф. Ситника.// К.: КНЕУ, 2001. – 420 с.
32. Організація баз даних: практичний курс: Навч. посіб. для студ. / А. Ю. Берко, О. М. Верес; Нац. ун-т «Львівська політехніка» - 2003.
33. Бойко Юлія. (2016), Енциклопедія Сучасної України — основний науковий проєкт Інституту енциклопедичних досліджень НАН України. Режим доступу до ресурсу *Гілея: науковий вісник*, *111*, 96–99.
34. Карвіш Б.К 21 Програмування баз даних SQL. Типові помилки та їх усунення Б. Карвін. - Рід Груп, 2016 р.
35. Кравець П.о. К 77 об'єктно - орієнтоване програмування: навч. посібник / П.о. Кравець. - Львів: Видавництво Львівської політехніки, 2018.