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

Лабораторна робота №4

Технології розроблення програмного забезпечення

«Вступ до паттернів проектування»

Виконав
студент групи ІА–34:
Бородай А.С.

Київ 2025

Зміст

Теоретичні відомості

Шаблон «Singleton»

Призначення патерну «Singleton»: являє собою клас в термінах ООП, який може мати не більше одного об'єкта. Насправді, кількість об'єктів можна задати (тобто не можна створити більш n об'єктів даного класу). Даний об'єкт найчастіше зберігається як статичне поле в самому класі.

Проблема: Використання одинака виправдано для наступних випадків:

• може бути не більше N фізичних об'єктів, що відображаються в певних класах;

• необхідно жорстко контролювати всі операції, що проходять через даний клас.

Одинак вирішує відразу дві проблеми, порушуючи принцип єдиної відповідальності класу.

Рішення:

У кожної програми є певний набір налаштувань, який як правило зберігається в окремий файл (для сучасних комп'ютерних ігор це може бути .ini файл, для .net додатків –.xml файл). Цей файл – єдиний, і тому використання безлічі об'єктів для завантаження/запису даних – нераціональне рішення.

Для демонстрації другого випадку, розглянемо наступний приклад. Припустимо, існує дві взаємодіючі системи, між якими встановлено сеанс зв'язку. Накладені обмеження, що по даному сеансу зв'язку дані можуть йти в один момент часу лише в одну сторону. Таким чином, на кожен надісланий запит необхідно дочекатися відповіді, перш ніж відсилати новий запит. Об'єкт «одинак» дозволить не тільки містити рівно один сеанс зв'язку, а й ще реалізувати відповідну логіку перевірок на основі bool операторів про можливість відправки запиту і, можливо, деяку чергу запитів.

Переваги та недоліки:

Однак слід зазначити, що в даний час патерн «Одинак» багато хто вважає т.зв. «анти-шаблоном», тобто поганою практикою проєктування. Це пов'язано з тим, що «одинаки» представляють собою глобальні дані (як глобальна змінна), що мають стан. Стан глобальних об'єктів важко відслідковувати і підтримувати коректно; також

глобальні об'єкти важко тестуються і вносять складність в програмний код (у всіх ділянках коду виклик в одне єдине місце з «одинаком»; при зміні підходу доведеться змінювати масу коду).

При цьому реалізація контролю доступу можлива за допомогою статичних змінних, замикань, мютексов та інших спеціальних структур.

+ Гарантує наявність єдиного екземпляра класу.
+ Надає до нього глобальну точку доступу.
- Порушує принцип єдиної відповідальності класу.
- Маскує поганий дизайн.

Шаблон «Iterator»

Призначення «Iterator»: шаблон реалізації об'єкта доступу до набору (колекції, агрегату) елементів без розкриття внутрішніх механізмів реалізації. Ітератор виносить функціональність перебору колекції елементів з самої колекції, таким чином досягається розподіл обов'язків: колекція відповідає за зберігання даних, ітератор – за прохід по колекції.

При цьому алгоритм ітератора може змінюватися – при необхідності пройти в зворотньому порядку використовується інший ітератор. Можливо також написання такого ітератора, який проходить список спочатку по парних позиціях (2,4,6-й елементи і т.д.), потім по непарних. Тобто, шаблон ітератор дозволяє реалізовувати різноманітні способи проходження по колекції незалежно від виду і способу представлення даних в колекції.

Проблема: Більшість колекцій виглядають як звичайний список елементів. Але є й екзотичні колекції, побудовані на основі дерев, графів та інших складних структур даних. Але як би не була структурована колекція, користувач повинен мати можливість послідовно обходити її елементи, щоб виробляти з ними якісь дії. Але яким способом слід переміщатися по складній структурі даних? Наприклад, сьогодні може бути достатнім обхід дерева в глибину, але завтра буде потрібно можливість переміщатися по дереву в ширину. А на наступному тижні і того гірше – знадобиться обхід колекції у

випадковому порядку. Додаючи все нові алгоритми в код колекції, ви потроху розмиваєте її основне завдання, яке полягає в ефективному зберіганні даних.

Рішення: Ідея патерна Ітератор полягає в тому, щоб винести поведінку обходу колекції з самої колекції в окремий клас. Об'єкт-ітератор буде відстежувати стан обходу, поточну позицію в колекції і скільки елементів ще залишилося обійти. Одну і ту ж колекцію зможуть одночасно обходити різні ітератори, а сама колекція не буде навіть знати про це. До того ж, якщо вам знадобиться додати новий спосіб обходу, ви зможете створити окремий клас ітератора, не змінюючи існуючий код колекції.

Шаблонний ітератор містить:

• First() – установка покажчика перебору на перший елемент колекції;

• Next() – установка покажчика перебору на наступний елемент колекції;

• IsDone – булевське поле, яке встановлюється як true коли покажчик перебору досяг кінця колекції;

• CurrentItem – поточний об'єкт колекції.

Переваги та недоліки:

Цей шаблон дозволяє уніфікувати операції проходження по наборам об'єктів для всіх наборів. Тобто, незалежно від реалізації (масив, зв'язаний список, незв'язаний список, дерево та ін.), кожен з наборів може використовувати будь-який з реалізованих ітераторів.

+ Дозволяє реалізувати різні способи обходу структури даних.
+ Спрощує класи зберігання даних.
- Не виправданий, якщо можна обійтися простим циклом.

Шаблон «Proxy»

Призначення «Proxy»: об'єкти є об'єктами-заглушками або двійниками/замінниками для об'єктів конкретного типу. Зазвичай, проксі об'єкти вносять додатковий функціонал або спрощують взаємодію з реальними об'єктами. Проксі об'єкти використовувалися в більш ранніх версіях інтернет-браузерів, наприклад, для відображення картинки: поки картинка завантажується, користувачеві відображається «заглушка» картинки.

Проблема: Ви супроводжуєте систему, одна із частин якої працює з зовнішнім сервісом підписання документів, нарпиклад, DocuSign. Періодично клієнти в вашій системі формують звіти и форматі pdf, далі ваша система відправляє їх на підпис в сервіс DocuSign і потім періодично перевіряєте чи документ вже підписаний. Якщо документ підписаний, ви викачуєте його з сервісу і поміщаєте в своїй базі. За останній рік кількість користувачів вашої системи виросло суттєво і почали приходити великі рахунки від DocuSign. Після аналізу ви розумієте, що рахунки зросли через велику кількість запитів з відправкою документів на підписання та запитів на перевірку чи документ вже підписаний. Після обговорення з бізнес-аналітиками та користувачам зрозуміли, що критичний інтервал доставки документів на підписання – 2 години і такий самий час критичності перевірки що документ підписаний і можна було б групувати всі запит на відправку та на отримання і відправляти пакетом раз на годину. На даний момент клієнтський код працює з класом DocSignManager через інтерфейс IDocSignManager.

Рішення: Для вирішення проблеми можна застосувати патерн "Замісник". Ви реалізовуєте клас замісник, який також реалізовує інтерфейс IDocSignManager, але він накопичує запити на відправку файлів на підписання і відправляє їх раз на годину, також він приблизно раз на годину отримує підписані документи, а на запити від клієнтів відповідає на основі інформації взятої з бази. Таким чином старий клас DocSignManager так само використовується для роботи з DocuSign сервісом, але вже набагато рідже, а клієнтський код взаємодіє з додатковим проміжним рівнем DocSignManagerProxy, хоча з точки зору клієнського коду нічого не змінилося і він працює з тим самим об'єктом IDocSignManager. Реалізувавши такий підхід ви тепер економите 40% від попередньої вартості використання DocuSign сервіса і тепер основний вплив на вартість робить розмір файлів, що передаються на підпис, а не кількість запитів до служби.

Переваги та недоліки:

+ Легкість впровадження проміжного рівня без переробки клієнтського коду.
+ Додаткові можливості по керуванню життєвим циклом об’єкту.
- Існує ризик падіння швидкості роботи через впровадження додаткових операцій.
- Існує ризик неадекватної заміни відповіді клієнтському коду

Шаблон «State»

Призначення «State»: дозволяє змінювати логіку роботи об'єктів у випадку зміни їх внутрішнього стану. Наприклад, відсоток нарахованих на картковий рахунок грошей залежить від стану картки: Visa Electron, Classic, Platinum і т.д. Або обсяг послуг, які надані хостинг компанією, змінюється в залежності від обраного тарифного плану (стану членства – бронзовий, срібний або золотий клієнт). Реалізація даного шаблону полягає в наступному: пов'язані зі станом поля, властивості, методи і дії виносяться в окремий загальний інтерфейс (State); кожен стан являє собою окремий клас (ConcreteStateA, ConcreteStateB), які реалізують загальний інтерфейс.

Об’єкти, що мають стан (Context), при зміні стану просто записують новий об'єкт в поле state, що призводить до повної зміни поведінки об'єкта. Це дозволяє легко додавати в майбутньому і обробляти нові стани, відокремлювати залежні від стану елементи об'єкта в інших об'єктах, і відкрито проводити заміну стану (що має сенс у багатьох випадках).

Проблема: Ви розробляєте систему, яка складається з мобільних клієнтів та центрального сервера. Для отримання запитів від клієнтів ви створюєте модуль Listener. Але вам потрібно щоб під час старту, поки сервер не запустився Listener не приймав запити від клієнтів, а після команди shutdown, поки сервер зупиняється, Listener вже не приймав запити від клієнтів, але відповіді, якщо вони будуть готові, відправив.

Рішення: Тут явно видно три стани для Listener з різною поведінкою: initializing, open, closing. Тому для вирішення краще за все підходить патерн State. Визначаємо загальний інтерфейс IListenerState з методами, які по різному працюють в різних станах. Під кожний стан створюємо клас з реалізацією цього інтерфейсе. Контекст Listener при цьому всі виклики буде перенаправляти на об'єкт стану простим делегуванням. При старті він (Listener) буде посилатися на об'єкт стану InitializingState, знаходячись в якому Listener, фактично, буде ігнорувати всі вхідні запити. Після того як система запуститься і завантажить всі базові дані для роботи, Listener буде переключено в робочий стан, наприклад, викликом методу Open(). Після цього Listener буде посилатися на об'єкт OpenState і буде відпрацьовувати всі вхідні повідомлення в звичайному режимі. При запуску виключення системи, Listener буде переключено в стан ClosingState, викликом методу Close().

Переваги та недоліки:

+ Код специфічний для окремого стану реалізується в класі стану.

+ Класи та об’єкти станів можна використовувати з різними контекстами, за рахунок чого збільшується гнучкість системи.

+ Код контексту простіше читати, тому що вся залежна від станів логіка винесена в інші класи.

+ Відносно легко додавати нові стани, головне правильно змінити переходи між станами.

- Клас контекст стає складніше через ускладнений механізм переключення станів.

Шаблон «Strategy»

Призначення «Strategy»: дозволяє змінювати деякий алгоритм поведінки об'єкта іншим алгоритмом, що досягає ту ж мету іншим способом. Прикладом можуть служити алгоритми сортування: кожен алгоритм має власну реалізацію і визначений в окремому класі; вони можуть бути взаємозамінними в об'єкті, який їх використовує. Даний шаблон дуже зручний у випадках, коли існують різні «політики» обробки даних. По суті, він дуже схожий на шаблон «State» (Стан), проте використовується в абсолютно інших цілях – незалежно від стану об'єкта відобразити різні можливі поведінки об'єкта (якими досягаються одні й ті самі або схожі цілі).

Рішення: Коли ви використовуєте патерн «Стратегія», то схожі алгоритми виносяться з класа контекста в конкретні стратегії, за рахунок чого клас контексту стає чистіше і його легше супроводжувати. Також одні і тіж самі стратегії можна використати з різними контекстами, що значно збільшує гнучкість вашої системи та зменшує кількість дублювань у коді. Контекст при цьому містить посилання на конкретну стратегію, а коли стратегію потрібно замінити, то замінюється об’єкт стратегії в полі Context.strategy. Важливою умовою є відносна простота інтерфейсу для алгоритмів стратегій. Якщо алгоритмам стратегій прийдеться передавати десятки параметрів, то це, скоріш за все, приведе до ускладнення системи та заплутаності коду. Якщо стратегії на вході будуть приймати об’єкт контексту, щоб отримувати з нього всі необхідні дані, то такі стратегії будуть прив’язані до конкретного контексту і їх не можна буде використати з іншим типом контексту.

Приклад з життя: Ви їдете на роботу. Можна доїхати на автомобілі, на метро, або йти пішки. Тут алгоритм, як ви добираєтеся на роботу, є стратегією. В залежності від поточної ситуації ви вибираєте стратегію, що найбільше підходить в цій ситуації, наприклад, на дорогах великі пробки тоді ви їдете на метро, або метро тимчасово не ходить тоді ви їдете на таксі, або ви знаходитеся в 5 хвилинах ходьби від місця роботи і простіше добратися пішки.

Переваги та недоліки:

+ Використовувані алгоритми можна змінювати під час виконання.
+ Реалізація алгоритмів відокремлюється від коду, що його використовує.
+ Зменшує кількість умовних операторів типу switch та if в контексті.
- Надмірна складність, якщо у вас лише кілька невеликих алгоритмів.
- Під час виклику алгоритму, клієнтський код має враховувати різницю між стратегіями.

Тема: Вступ до паттернів проектування.

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

Завдання:

• Ознайомитись з короткими теоретичними відомостями.

• Реалізувати частину функціоналу робочої програми у вигляді класів та їхньої взаємодії для досягнення конкретних функціональних можливостей.

• Реалізувати один з розглянутих шаблонів за обраною темою.

• Реалізувати не менше 3-х класів відповідно до обраної теми.

• Підготувати звіт щодо виконання лабораторної роботи. Поданий звіт повинен містити: діаграму класів, яка представляє використання шаблону в реалізації системи, навести фрагменти коду по реалізації цього шаблону.

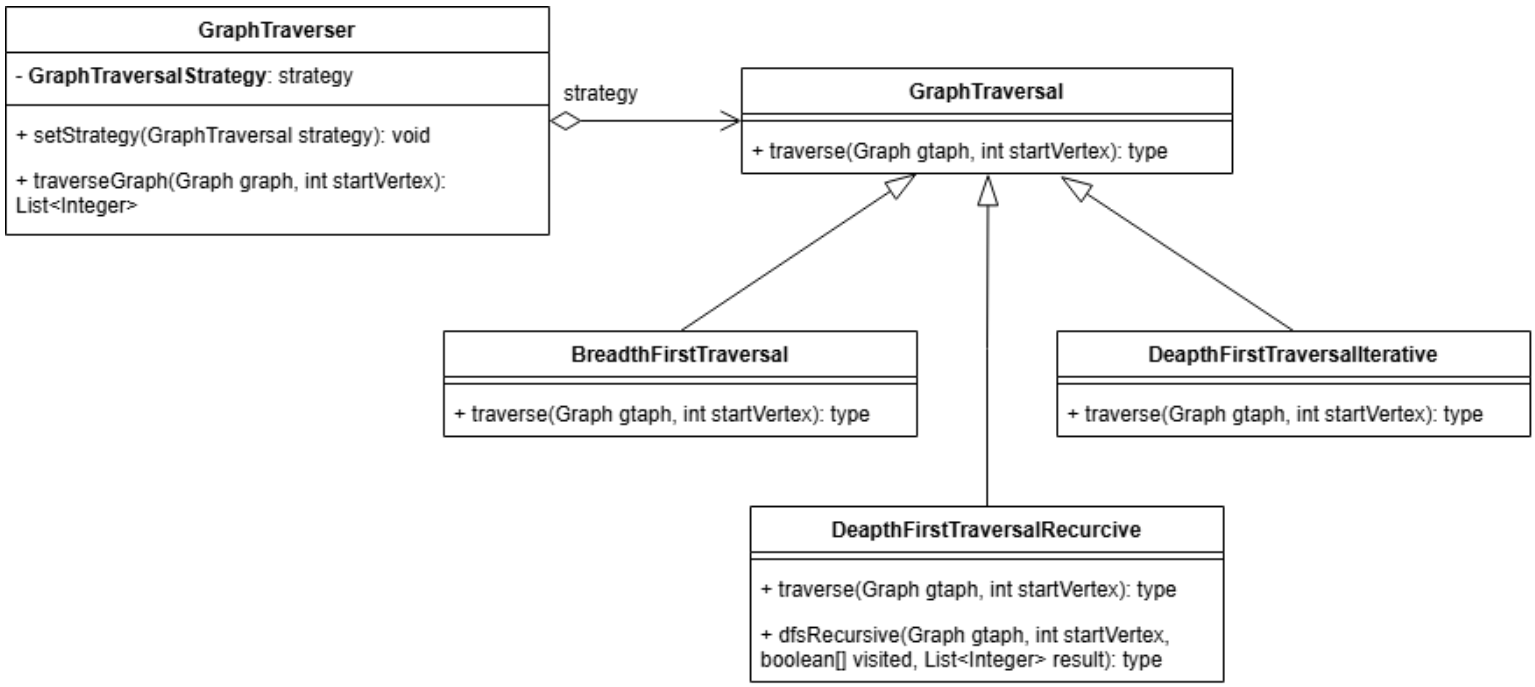
Хід роботи

20. **Mind-mapping software**

(strategy, prototype, abstract factory, bridge, composite, SOA)

Візуальний додаток для складання "карт пам'яті" з можливістю роботи з декількома картами (у вкладках), автоматичного промальовування ліній, додавання вкладених файлів, картинок, відеофайлів (попередній перегляд); можливість додавання значків категорій / терміновості, обведення областей карти (поділ пунктирною лінією).

ClassDiagram (Strategy)



Висновок

Під час виконання лабораторної роботи ми ознайомилися і шаблонами «Singleton», «Iterator», «Proxy», «State», «Strategy».

Нами були розроблені основні класи, які реалізують шаблон Strategy та її поведінку.

Нові шаблони будуть використані для подальшої реалізації системи та подальшого розуміння нашої системи.

Відповіді на питання для самоперевірки

1. Що таке шаблон проєктування?

Це узагальнене, повторюване рішення типових проблем, що виникають при проєктуванні програмного забезпечення. Це не готовий код, а опис підходу, який можна адаптувати для вирішення конкретної задачі в певному контексті. Шаблони дозволяють використовувати кращі практики та досвід інших розробників.

2. Навіщо використовувати шаблони проєктування?

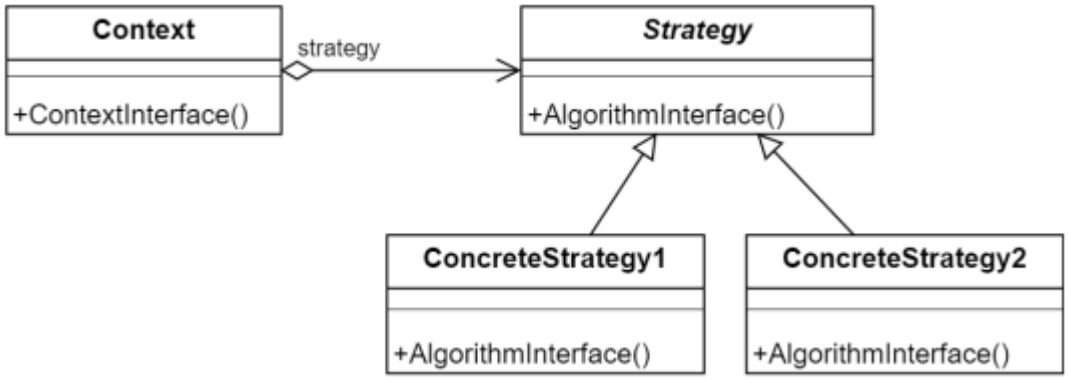
Вони дозволяють вирішувати проблеми перевіреними способами, Код, написаний з використанням зрозумілих шаблонів, легше розуміти та підтримувати іншим розробникам і розробники можуть використовувати назви шаблонів для швидкого опису архітектури системи.

Вони уникають дублювання логіки, пропонуючи оптимальні структури та багато шаблонів розділяють модулі, роблячи систему більш гнучкою до змін.

3. Яке призначення шаблону «Стратегія»?

Визначити сімейство алгоритмів, інкапсулювати кожен з них і робити їх взаємозамінними. Це дозволяє змінювати алгоритм незалежно від клієнта, який його використовує.

4. Нарисуйте структуру шаблону «Стратегія».



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

Strategy (Інтерфейс): Оголошує метод, який виконують усі конкретні стратегії.

ConcreteStrategy (Конкретна стратегія): Реалізує інтерфейс Strategy, надаючи власну реалізацію алгоритму.

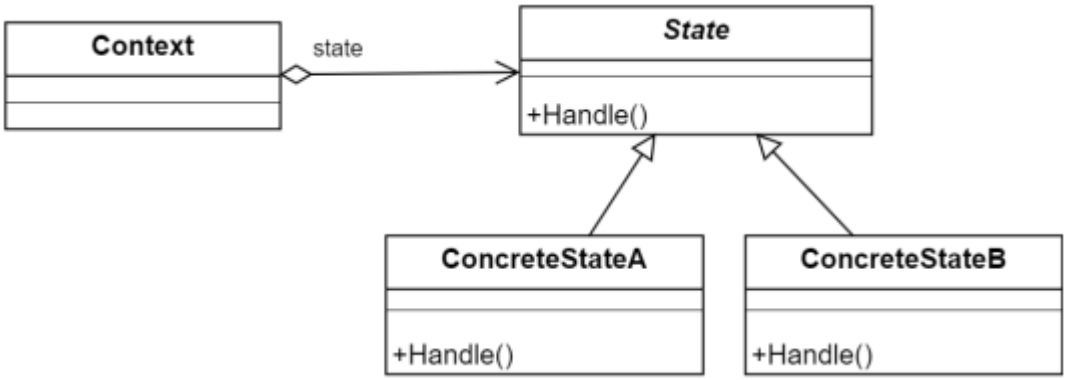
Context (Контекст): Зберігає посилання на об'єкт стратегії. Має метод для зміни стратегії. Викликає метод стратегії у своєму методі.

Взаємодія: Клієнт створює конкретну стратегію і передає її контексту. Контекст делегує виконання алгоритму об'єкту стратегії. Змінюючи стратегію в контексті, можна змінювати його поведінку на льоту.

6. Яке призначення шаблону «Стан»?

Дозволити об'єкту змінювати свою поведінку при зміні його внутрішнього стану. Створюється враження, що об'єкт змінив свій клас.

7. Нарисуйте структуру шаблону «Стан».



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

State (Інтерфейс): Оголошує методи, які представляють дії, що можуть бути виконані в кожному стані.

ConcreteState (Конкретний стан): Реалізує методи інтерфейсу State, визначаючи поведінку для конкретного стану. Зазвичай має посилання на Context для ініціювання переходу до іншого стану.

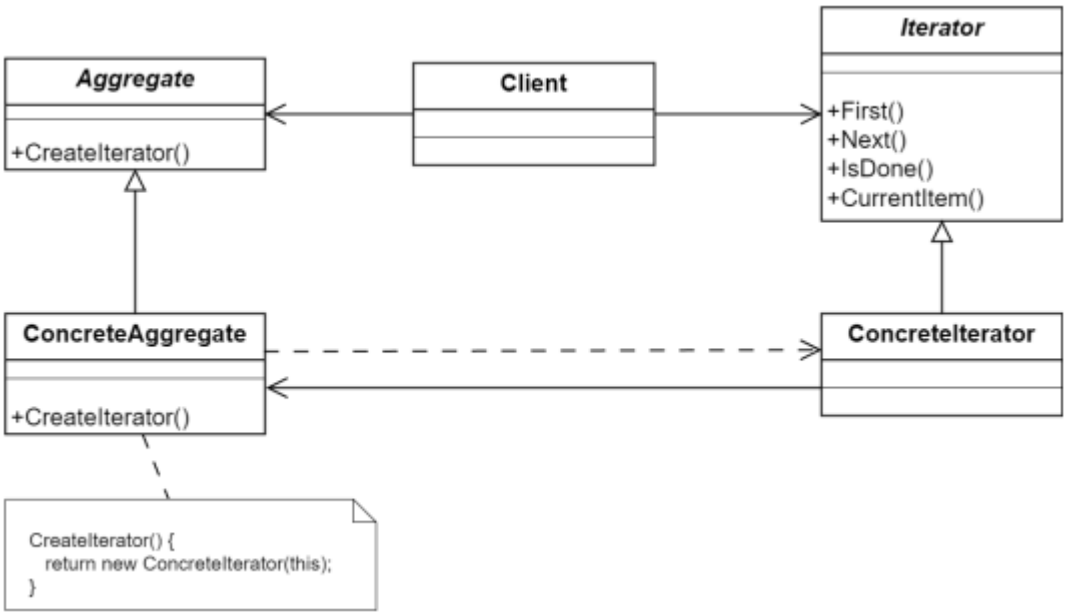
Context (Контекст): Зберігає посилання на поточний об'єкт стану. Має метод для зміни стану (setState). Делегує запити від клієнта поточному об'єкту стану.

Взаємодія: Коли контекст отримує запит, він передає його поточному стану. Конкретний стан обробляє запит і може перевести контекст в інший стан, використовуючи метод setState.

9. Яке призначення шаблону «Ітератор»?

Надати спосіб послідовного доступу до елементів агрегованого об'єкта, не розкриваючи його внутрішньої структури.

10. Нарисуйте структуру шаблону «Ітератор».



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

Iterator (Інтерфейс): Оголошує операції для доступу та обходу елементів: next(), currentItem(), isDone().

ConcreteIterator (Конкретний ітератор): Реалізує інтерфейс Iterator. Зберігає поточну позицію при обході колекції.

Aggregate (Інтерфейс колекції): Оголошує метод для створення ітератора.

ConcreteAggregate (Конкретна колекція): Реалізує метод createIterator, повертаючи конкретний ітератор, прив'язаний до себе.

Взаємодія: Клієнт отримує ітератор від колекції за допомогою createIterator(). Потім використовує методи ітератора (next(), isDone() тощо) для послідовного обходу всіх елементів колекції, не знаючи, як вони зберігаються.

12. В чому полягає ідея шаблона «Одинак»?

Гарантувати, що клас має лише один екземпляр, та надати глобальну точку доступу до цього екземпляра.

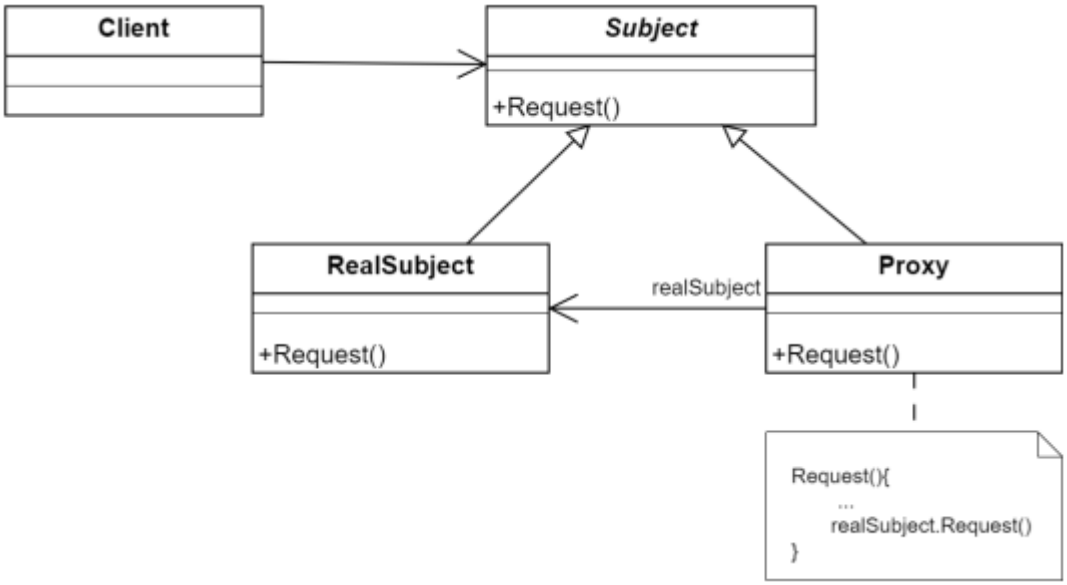
13. Чому шаблон «Одинак» вважають «анти-шаблоном»?

Він вводить глобальні змінні в програму, що ускладнює тестування і робить код складнішим для розуміння. Клас починає вирішувати дві задачі: власну основну логіку та контроль створення своїх екземплярів. Потрібні додаткові засоби (блокування), щоб гарантувати коректну роботу в багатопоточному середовищі. Залежності класу не видно в його інтерфейсі, оскільки одинак отримується глобально, а не передається через конструктор.

14. Яке призначення шаблону «Проксі»?

Надати об'єкт-замінник або утримувач для іншого об'єкта, щоб контролювати доступ до нього. Проксі дозволяє виконувати додаткові дії до або після основного запиту до реального об'єкта.

15. Нарисуйте структуру шаблону «Проксі».



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

Subject (Інтерфейс): Визначає спільний інтерфейс для RealSubject та Proxy, що дозволяє проксі використовуватися там, де очікується реальний об'єкт.

RealSubject (Реальний об'єкт): Містить реальну бізнес-логіку. Це об'єкт, до якого ми хочемо контролювати доступ.

Proxy (Проксі): Зберігає посилання на RealSubject (може створювати або управляти ним). Реалізує той самий інтерфейс Subject. Контролює доступ до RealSubject, виконуючи додаткову логіку (наприклад, перевірка прав доступу, кешування, ліниве завантаження, логування) перед (або після) виклику методу реального об'єкта.

Взаємодія: Клієнт працює з інтерфейсом Subject. Йому передається об'єкт Proxy. Коли клієнт викликає метод (наприклад, request()), проксі перехоплює цей виклик, виконує свою додаткову логіку, і потім (за потреби) перенаправляє виклик реальному об'єкту RealSubject.

Код системи, який було зроблено в цій лабораторній

Лістинг: Strategy context

```
// Контекст, що використовує стратегію обходу графа
class GraphTraverser {
    private GraphTraversalStrategy strategy;

    public GraphTraverser(GraphTraversalStrategy strategy) {
        this.strategy = strategy;
    }

    public void setStrategy(GraphTraversalStrategy strategy) {
        this.strategy = strategy;
    }

    public List<Integer> traverseGraph(Graph graph, int startVertex) {
        return strategy.traverse(graph, startVertex);
    }
}
```

Лістинг: Strategy Interface

```
interface GraphTraversalStrategy {
    List<Integer> traverse(Graph graph, int startVertex);
}
```

Лістинг: Конкретна стратегія обходу Графа (BFS)

```
// BFS
class BreadthFirstTraversal implements GraphTraversalStrategy {
    @Override
    public List<Integer> traverse(Graph graph, int startVertex) {
```

```
        List<Integer> result = new ArrayList<>();
        boolean[] visited = new boolean[graph.getVerticesCount()];
        Queue<Integer> queue = new LinkedList<>();

        visited[startVertex] = true;
        queue.offer(startVertex);

        while (!queue.isEmpty()) {
            int current = queue.poll();
            result.add(current);

            for (int neighbor : graph.getNeighbors(current)) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    queue.offer(neighbor);
                }
            }
        }
        return result;
    }
}
```

Лістинг: Конкретна стратегія обходу Графа (DFS)

```
// DFS - рекурсивна версія
class DepthFirstTraversalRecursive implements GraphTraversalStrategy {
    @Override
    public List<Integer> traverse(Graph graph, int startVertex) {
        List<Integer> result = new ArrayList<>();
        boolean[] visited = new boolean[graph.getVerticesCount()];
        dfsRecursive(graph, startVertex, visited, result);
        return result;
    }
```

```
    private void dfsRecursive(Graph graph, int vertex, boolean[] visited, List<Integer> result)
{
        visited[vertex] = true;
        result.add(vertex);

        for (int neighbor : graph.getNeighbors(vertex)) {
            if (!visited[neighbor]) {
                dfsRecursive(graph, neighbor, visited, result);
            }
        }
    }
}
```

Лістинг: Конкретна стратегія обходу Графа (DFS)

```
// DFS - ітеративна версія
class DepthFirstTraversalIterative implements GraphTraversalStrategy {
    @Override
    public List<Integer> traverse(Graph graph, int startVertex) {
        List<Integer> result = new ArrayList<>();
        boolean[] visited = new boolean[graph.getVerticesCount()];
        Stack<Integer> stack = new Stack<>();

        stack.push(startVertex);

        while (!stack.isEmpty()) {
            int current = stack.pop();

            if (!visited[current]) {
                visited[current] = true;
                result.add(current);
```

```
            List<Integer> neighbors = graph.getNeighbors(current);
            for (int i = neighbors.size() - 1; i >= 0; i--) {
                int neighbor = neighbors.get(i);
                if (!visited[neighbor]) {
                    stack.push(neighbor);
                }
            }
        }
    }

    return result;
  }
}
```

Лістинг: Модель графа

```
class Graph {
  private final int vertices;
  private final Map<Integer, List<Integer>> adjacencyList;

  public Graph(int vertices) {
    this.vertices = vertices;
    this.adjacencyList = new HashMap<>();
    for (int i = 0; i < vertices; i++) {
      adjacencyList.put(i, new ArrayList<>());
    }
  }

  public void addEdge(int source, int destination) {
    adjacencyList.get(source).add(destination);
  }
```

```
    public List<Integer> getNeighbors(int vertex) {
        return adjacencyList.getOrDefault(vertex, new ArrayList<>());
    }

    public int getVerticesCount() {
        return vertices;
    }
}
```