Екзамен ОООП 2 семестр

Принципи (завдання 1)

1. KISS **10**
2. Law of Demeter **10**
3. Least Astonishment **10**
4. Coupling/Cohesion **9**
5. YAGNI **9**
6. DRY **9**
7. абстракції **9**
8. OCP **9**
9. SRP **8**
10. LSP **8**
11. reuse **8**
12. ISP **8**
13. DIP **8**
14. pay as you go 7
15. підтримки **7**
16. SoC **7**
17. мінімалізм **5**
18. ETC **5**
19. orthogonality **5**
20. декомпозиції **5**
21. information hiding **4**
22. rule of least power **4**
23. Pareto principle **4**
24. перфекціонізм **4**
25. WET **3**
26. rule of three **3**
27. Chesterton’s fence **3**
28. single source of truth **3**
29. MVP **3**
30. Hyrum's Law **3**
31. cross-cutting concerns **2**
32. scout rule **2**
33. law of leaky abstractions **2**
34. zero-cost abstractions **1**
35. abstraction layers **1**
36. Occam’s razor **1**

Патерни (завдання 2)

1. Decorator **12**
2. Composite **12**
3. Strategy **11**
4. Flyweight **11**
5. Memento **9**
6. Iterator **9**
7. Builder **9**
8. Template Method **9**
9. Visitor **9**
10. Chain of Responsibility **8**
11. Proxy **8**
12. Prototype **7**
13. Command **7**
14. State **7**
15. Mediator **7**
16. Facade **6**
17. Singleton **6**
18. Abstract Factory **6**
19. Observer **6**
20. Bridge **5**
21. Adapter **4**
22. Interpreter **4**
23. Factory Method **2**

UML (завдання 3)

1. засіб для роботи з багатьма репозиторіями git одночасно (можливість створення ієрархії груп для репозиторіїв)
2. засіб для генерації коду-заготовки для патернів проєктування на заданій мові програмування (можливість створення ієрархії груп для патернів)
3. засіб для пошуку копій програмного коду (плагіату) (можливість пошуку по переліку локальних каталогів)
4. універсальну платформу для підтримки різних настільних ігор (можливість створення ієрархії груп для ігор)
5. засіб для генерації стандартних фрагментів коду (code snippets) на різних мовах програмування (можливість створення ієрархії груп для фрагментів коду)
6. засіб для ідентифікації студентів в аудиторії (наприклад, відвідування лекцій) за фотографією аудиторії (можливість додавання декількох фотографій одного студента)
7. засіб для видалення непотрібних файлів з ієрархії каталогів (наприклад, чистка коду від згенерованих артефактів) (можливість додавання переліку локальних каталогів для обробки)
8. засіб для генерації документів на основі шаблону та заповнення даних (наприклад, білети до іспиту) (можливість створення ієрархії груп для шаблонів)
9. бюрократичний навігатор – засіб для зберігання інформації про бюрократичні процедури, підказки щодо оптимального їх проходження (можливість пошуку бюрократичних процедур за різними параметрами – ключовими словами, організаціями куди треба звертатись, кількістю кроків тощо)
10. систему для вивчення університетської математики (мат. аналіз, алгебра, теорія ймовірностей, ...) в ігровій формі (можливість додавання нових ігрових активностей, зокрема одразу до декількох предметів)
11. засіб для time tracking (відслідковування часу виконання різних завдань розробником) з інтеграцією із зовнішніми системами (IDE, файлова система, веб сервіси, ...) (можливість перегляду часу виконання різних груп активностей)

Принципи (завдання 1)

1. KISS
2. Law of Demeter
3. Least Astonishment
4. Coupling/Cohesion
5. YAGNI
6. DRY
7. Абстракція
8. OCP
9. SRP
10. LSP
11. Reuse
12. ISP
13. DIP
14. Pay as you go
15. Підтримка
16. SoC
17. Мінімалізм
18. ETC
19. Orthogonality
20. Декомпозиція
21. Information hiding
22. Rule of least power
23. Pareto principle
24. Перфекціонізм
25. WET
26. Rule of three
27. Chesterton’s fence
28. Single source of truth
29. MVP
30. Hyrum's Law
31. Cross-cutting concerns
32. Scout rule
33. Law of leaky abstractions
34. Zero-cost abstractions
35. Abstraction layers
36. Occam’s razor

KISS

KISS - це дуже загальний і абстрактний принцип проектування, який містить практично всі інші принципи проектування. Принципи проектування описують як писати «хороший» код. Деякі вважають, що це код, який виконується максимально швидко, деякі – що це код, в якому задіяно якнайбільше патернів проектування… Але правильна відповідь лежить на поверхні.

Основні ідеї принципу KISS:

- **Простота:** Системи, продукти або процеси мають бути максимально простими для розуміння, використання та обслуговування.
- **Функціональність:** Фокус на основних функціях та уникання зайвих деталей, які можуть ускладнити систему.
- **Надійність:** Прості системи зазвичай менш схильні до помилок та простіші у виправленні, якщо помилки все ж виникають.
- **Ефективність:** Прості рішення часто є більш ефективними з точки зору часу, ресурсів та вартості.

Переваги використання принципу KISS:

- **Зниження витрат:** Прості рішення зазвичай вимагають менше ресурсів та часу на розробку та впровадження.
- **Підвищення ефективності:** Прості системи та процеси легше зрозуміти та використовувати, що призводить до підвищення продуктивності.
- **Покращення якості:** Прості системи менш схильні до помилок, що призводить до підвищення якості продукту або послуги.
- **Збільшення задоволеності користувачів:** Прості та інтуїтивно зрозумілі продукти та послуги легше використовувати, що підвищує задоволеність користувачів.

Law of Demeter

Law of Demeter - це правило проектування програмного забезпечення, спрямоване на зменшення зв'язності між модулями. Він підкреслює, що об'єкт повинен взаємодіяти лише з об'єктами, які є його "близькими друзями", а не з "чужими".

Основні ідеї принципу Law of Demeter:

- **Обмеження знань:** Об'єкт повинен мати обмежені знання про внутрішню структуру інших об'єктів. Він повинен взаємодіяти з іншими об'єктами лише через їх публічні інтерфейси.
- **Уникнення ланцюжкових викликів:** Не слід використовувати ланцюжки викликів методів .Це порушує інкапсуляцію та збільшує зв'язність між об'єктами.

- **"Розмовляйте" лише з друзями:** Об'єкт повинен "розмовляти" лише з такими об'єктами:
    - З самим собою (власні методи та атрибути).
    - З об'єктами, переданими йому як аргументи.
    - З об'єктами, які він створює або інстанціює.
    - З об'єктами, які є його прямими компонентами.

Переваги використання принципу Law of Demeter:

- **Зменшення зв'язності:** Модулі стають менш залежними один від одного, що полегшує їх зміну та повторне використання.
- **Підвищення гнучкості:** Зміни у внутрішній структурі одного об'єкта менше впливають на інші об'єкти.
- **Покращення тестування:** Об'єкти з меншою кількістю залежностей легше тестувати ізольовано.

Least Astonishment

Принцип **Least Astonishment** (також відомий як Принцип Найменшої Несподіванки) - це принцип проектування, який стверджує, що система повинна поводитися таким чином, щоб вона найменше здивувала своїх користувачів.

Основні ідеї принципу Least Astonishment:

- **Передбачуваність**: Система повинна діяти таким чином, щоб користувачі могли передбачити її поведінку на основі свого попереднього досвіду та знань про подібні системи.
- **Узгодженість**: Поведінка системи повинна бути узгодженою на всіх етапах, що допомагає користувачам формувати точні очікування щодо її роботи.
- **Інтуїтивний Дизайн**: Дизайн повинен бути інтуїтивно зрозумілим, дозволяючи користувачам зрозуміти та використовувати систему без плутанини або несподіваних результатів.

Переваги дотримання принципу Least Astonishment:

- **Зручність для Користувача**: Системи стають більш зручними та легшими для вивчення, коли вони поводяться так, як очікується.
- **Зменшення Помилок**: Передбачувані системи знижують ймовірність помилок користувачів.
- **Ефективна Взаємодія**: Користувачі можуть ефективніше взаємодіяти з системою, коли їм не доводиться мати справу з несподіваною поведінкою.

Coupling/Cohesion

Coupling/Cohesion - це два фундаментальні поняття в проектуванні програмного забезпечення, які стосуються того, наскільки добре компоненти системи працюють разом.

Основні ідеї принципу Coupling/Cohesion:

- **Coupling (зчеплення)** відноситься до ступеня взаємозалежності між модулями програмного забезпечення. Бажане низьке зчеплення, оскільки це означає, що зміни в одному модулі мають мінімальний вплив на інші, роблячи систему легшою для обслуговування та розширення.

- **Cohesion (згуртованість)** відноситься до ступеня, до якого елементи всередині модуля належать один до одного. Бажана висока згуртованість, оскільки це означає, що модуль має чітку мету та функцію, що робить його легшим для розуміння, обслуговування та повторного використання.

Переваги Coupling/Cohesion:

- **Легше Обслуговування**: Системи з низьким зчепленням та високою когезією легше обслуговуються, оскільки зміни в одній частині системи менше впливають на інші.
- **Покращена Зрозумілість**: Висока когезія полегшує розробникам розуміння того, що робить кожна частина системи.
- **Покращена Повторне Використання**: Модулі з високою когезією можна легко повторно використовувати в різних частинах системи або в різних проектах.

YAGNI

YAGNI, що означає “Вам це не знадобиться”, - це принцип екстремального програмування, який стверджує, що програміст не повинен додавати функціональність до тих пір, поки вона не стане необхідною.

Основні ідеї принципу YAGNI:

- **Уникайте Надмірного Проектування**: Не реалізовуйте функції або проектні рішення для майбутніх проблем, які можуть ніколи не виникнути.
- **Зосередьтеся на Поточних Потребах**: Концентруйтеся на тому, що потрібно зараз, а не на тому, що може знадобитися в майбутньому.

Переваги YAGNI:

- **Економить Час та Зусилля**: Не створюючи непотрібних функцій, розробники економлять час та зусилля.

- **Зменшує Складність**: Код залишається простішим і легшим для управління.
- **Покращує Продуктивність**: Розробники можуть зосередитися на наданні цінності зараз, а не на спекулятивних майбутніх вимогах.

DRY

DRY, що означає “Don’t Repeat Yourself” (Не Повторюйте Себе), - це принцип програмування, який спонукає до зменшення повторення інформації.

Основні ідеї принципу DRY:

- **Уникайте Дублювання:** Не пишіть однаковий код багато разів; використовуйте абстракції або модулі для уникнення повторень.
- **Покращення Читабельності:** Код стає легшим для читання та підтримки, коли в ньому менше дублювань.

Переваги DRY:

- **Легше Підтримувати:** Зміни в коді потребують внесення лише в одному місці, замість багатьох.
- **Зменшення Ймовірності Помилок:** Менше коду означає менше місць для потенційних помилок.
- **Ефективність Розробки:** Розробники можуть швидше працювати, не витрачаючи час на написання та перевірку однакового коду.

Абстракція

Абстракція в програмуванні - це процес приховування складності та деталей реалізації, зосереджуючись лише на функціональності.

Основні ідеї абстракції:

- **Спрощення:** Абстракція дозволяє розробникам використовувати складні системи, не знаючи тонкощів їх реалізації.
- **Перевикористання:** Загальні концепції можуть бути абстраговані та використані в різних програмах або модулях.

Переваги абстракції:

- **Зменшення Складності:** Розробники можуть управляти складними системами, розбиваючи їх на менш складні частини.
- **Покращення Модульності:** Абстрактні компоненти можуть бути легко замінені або модифіковані без впливу на іншу частину системи.
- **Легкість Тестування:** Абстраговані компоненти можуть бути тестовані окремо, що спрощує процес тестування.

OCP

OCP, що означає “Open/Closed Principle” (Принцип Відкритості/Закритості), - це принцип об’єктно-орієнтованого програмування, який стверджує, що програмне забезпечення має бути відкритим для розширення, але закритим для модифікації.

Основні ідеї OCP:

- **Розширюваність:** Система повинна бути спроектована таким чином, щоб нові функціональні можливості можна було додавати з мінімальними змінами в існуючий код.
- **Закритість для Модифікацій:** Існуючий код не повинен змінюватися при додаванні нових функцій.

Переваги OCP:

- **Гнучкість:** Система стає більш гнучкою та адаптивною до змін.
- **Стабільність:** Зменшується ризик введення помилок при розширенні системи.
- **Покращення Перевикористання:** Компоненти системи можуть бути легше перевикористані в інших контекстах.

SRP

SRP, що означає “Single Responsibility Principle” (Принцип Єдиної Відповідальності), - це принцип об’єктно-орієнтованого програмування, який стверджує, що клас повинен мати лише одну причину для зміни.

Основні ідеї SRP:

- **Єдина Відповідальність:** Кожен клас повинен виконувати лише одне завдання або мати лише одну відповідальність.
- **Спрощення Змін:** Зміни в одному аспекті системи не повинні впливати на інші аспекти.

Переваги SRP:

- **Легкість Підтримки:** Код стає легшим для розуміння та підтримки, коли в ньому чітко розділені відповідальності.
- **Гнучкість:** Система стає більш гнучкою та адаптивною до змін.
- **Покращення Тестування:** Компоненти з однією відповідальністю легше тестувати, оскільки їх поведінка більш передбачувана.

LSP

LSP, що означає “Liskov Substitution Principle” (Принцип Підстановки Лісков), - це принцип об’єктно-орієнтованого програмування, який стверджує, що об’єкти в програмі можуть бути замінені їх підтипами без зміни правильності програми.

Основні ідеї LSP:

- **Заміннність:** Об’єкти класу можуть бути замінені об’єктами його підкласів без впливу на функціональність програми.
- **Сумісність Поведінки:** Поведінка підкласів повинна бути сумісною з поведінкою базового класу.

Переваги LSP:

- **Покращення Модульності:** Система стає більш модульною, оскільки компоненти можуть бути легко замінені на їх підтипи.
- **Гнучкість Дизайну:** Дизайн системи стає більш гнучким та адаптивним до змін.
- **Легше Розширення:** Система легше розширюється за рахунок додавання нових підтипів.

Reuse

Reuse в програмуванні - це практика використання існуючого коду для нових функцій або проектів, щоб зменшити дублювання та спростити розробку.

Основні ідеї reuse:

- **Ефективність:** Використання готових компонентів замість створення нових з нуля економить час та ресурси.
- **Стандартизація:** Перевикористання сприяє стандартизації коду та практик, що полегшує співпрацю та інтеграцію.

Переваги reuse:

- **Зменшення Витрат:** Зниження витрат на розробку та підтримку.
- **Покращення Якості:** Використання перевірених компонентів може покращити якість кінцевого продукту.
- **Швидкість Розробки:** Прискорення процесу розробки за рахунок використання готових рішень.

ISP

ISP, що означає “Interface Segregation Principle” (Принцип Розділення Інтерфейсів), - це принцип об’єктно-орієнтованого програмування, який стверджує, що клієнти не повинні бути змушені залежати від інтерфейсів, якими вони не користуються.

Основні ідеї ISP:

- **Розділення Інтерфейсів:** Створення специфічних інтерфейсів для різних клієнтів замість одного загального інтерфейсу.
- **Видалення Непотрібних Залежностей:** Класи не повинні залежати від методів, якими вони не користуються.

Переваги ISP:

- **Гнучкість:** Система стає більш гнучкою та легко адаптується до змін.
- **Покращення Перевикористання:** Інтерфейси, орієнтовані на конкретні потреби, можуть бути легше перевикористані.
- **Зменшення Складності:** Розділення інтерфейсів допомагає уникнути “жирних” інтерфейсів та спрощує розуміння системи.

DIP

DIP, що означає “Dependency Inversion Principle” (Принцип Інверсії Залежностей), - це принцип об’єктно-орієнтованого програмування, який стверджує, що високорівневі модулі не повинні залежати від низькорівневих модулів, обидва типи модулів повинні залежати від абстракцій.

Основні ідеї DIP:

- Інверсія Залежностей: Залежності в програмуванні повинні бути від абстракцій, а не від конкретних реалізацій.
- Абстракція: Високорівневий код описує інтерфейси та базовий класи, якими можуть користуватися низькорівневі модулі.

Переваги DIP:

- Гнучкість: Система стає більш гнучкою та легко адаптується до змін.
- Покращення Тестування: Абстракції полегшують мокування та тестування компонент.
- Зменшення Зчеплення: Інверсія залежностей допомагає зменшити зчеплення між компонентами системи.

Pay as you go

У контексті програмування, **“pay as you go”** часто використовується у хмарних сервісах та платформах, де розробники платять за використання обчислювальних ресурсів, сховищ даних, або інших сервісів в залежності від їх поточних потреб.

Основні ідеї “pay as you go” в програмуванні:

- **Масштабованість:** Розробники можуть масштабувати ресурси вгору або вниз без необхідності купувати додаткове обладнання.
- **Оптимізація Витрат:** Платежі базуються на фактичному використанні, що дозволяє оптимізувати бюджет.

Переваги “pay as you go” в програмуванні:

- **Економія:** Зменшення витрат на інфраструктуру та обслуговування.
- **Гнучкість:** Легкість адаптації до змінних потреб проекту.
- **Спрощення Управління:** Автоматизація масштабування та управління ресурсами.

Підтримка

У програмуванні, **“підтримка”** може відноситися до різних аспектів, включаючи підтримку програмного забезпечення, технічну підтримку, або підтримку спільноти.

Основні ідеї підтримки:

- **Підтримка Програмного Забезпечення:** Оновлення та виправлення помилок для забезпечення безперебійної роботи програм.
- **Технічна Підтримка:** Допомога користувачам у вирішенні технічних проблем або запитань.
- **Підтримка Спільноти:** Форуми, чати, та інші ресурси для обміну знаннями та досвідом.

Переваги підтримки:

- **Забезпечення Стабільності:** Регулярна підтримка допомагає уникнути критичних помилок та збоїв.
- **Покращення Досвіду Користувача:** Швидке реагування на запити користувачів покращує загальне задоволення продуктом.
- **Розвиток Спільноти:** Активна спільнота може сприяти інноваціям та розвитку продукту.

SoC

SoC, що означає “Separation of Concerns” (Розділення Відповідальностей), - це дизайн-принцип для розбиття програми на окремі частини, кожна з яких вирішує окрему проблему, має окрему відповідальність, і може бути змінена незалежно від інших частин.

Основні ідеї SoC:

- **Модульність:** Створення чітко визначених модулів з окремими відповідальностями.
- **Зменшення Зчеплення:** Незалежність модулів дозволяє зменшити зчеплення між ними.

Переваги SoC:

- **Легкість Управління:** Кожен модуль може бути розроблений, тестований, та покращений незалежно.
- **Покращення Перевикористання:** Модулі можуть бути легше перевикористані у різних частинах програми або у різних проектах.
- **Гнучкість:** Зміни в одному модулі менше впливають на інші частини програми.

Мінімалізм

Мінімалізм у програмуванні - це філософія, яка пропагує використання мінімальної кількості коду та ресурсів для досягнення поставленої мети. Він заохочує розробників писати простий, зрозумілий та підтримуваний код, уникаючи непотрібної складності та надмірностей.

Основні принципи мінімалізму в програмуванні:

- **KISS (Keep It Simple, Stupid):** Не ускладнюй! Пишіть код максимально простим та зрозумілим. Уникайте зайвих абстракцій, складних структур даних та алгоритмів, якщо вони не є необхідними.
- **DRY (Don't Repeat Yourself):** Не повторюйся! Уникайте дублювання коду. Якщо певний фрагмент коду використовується кілька разів, винесіть його в окрему функцію або метод.
- **YAGNI (You Aren't Gonna Need It):** Вам це не знадобиться! Не додавайте функціональність, яка може знадобитися в майбутньому, але не є необхідною зараз. Краще додати її пізніше, коли вона дійсно знадобиться.
- **Принцип єдиної відповідальності (Single Responsibility Principle):** Кожен модуль або клас повинен мати лише одну причину для зміни. Це робить код більш модульним та легким для підтримки.
- **Принцип відкритості/закритості (Open/Closed Principle):** Програмні сутності (класи, модулі, функції тощо) повинні бути відкритими для розширення, але

закритими для модифікації. Це означає, що ви можете додавати нову функціональність, не змінюючи існуючий код.

Переваги мінімалізму в програмуванні:

- **Легкість розуміння та підтримки:** Мінімалістичний код простіше читати, розуміти та підтримувати, оскільки він містить менше коду та менше залежностей.
- **Зменшення кількості помилок:** Простіший код менш схильний до помилок, оскільки він має менше місць, де можуть виникнути проблеми.
- **Підвищення продуктивності:** Розробники витрачають менше часу на розуміння та модифікацію мінімалістичного коду, що дозволяє їм швидше виконувати завдання.

ETC

Принцип ETC є одним з основних принципів екстремального програмування (XP). Він стверджує, що код повинен бути написаний таким чином, щоб його було легко змінювати в майбутньому. Це означає, що код повинен бути модульним, гнучким та добре протестованим.

Як досягти ETC:

- **Дотримуйтесь принципів SOLID:** Принципи SOLID допомагають створювати модульний, гнучкий та легко змінюваний код.
- **Пишіть тести:** Тести допомагають переконатися, що зміни в коді не порушують його функціональність.
- **Використовуйте рефакторинг:** Рефакторинг - це процес покращення структури коду без зміни його поведінки. Він допомагає зробити код більш читабельним, зрозумілим та легким для зміни.
- **Використовуйте інструменти автоматизації:** Інструменти автоматизації, такі як системи контролю версій та інструменти безперервної інтеграції, допомагають відстежувати зміни в коді та забезпечувати його якість.

Переваги ETC:

- **Зниження вартості змін:** Зміни в коді, який легко змінювати, коштують дешевше та займають менше часу.
- **Підвищення гнучкості:** Легко змінюваний код дозволяє швидко реагувати на зміни вимог.
- **Покращення якості:** Зміни в коді, який добре протестований та має модульну структуру, менш схильні до помилок.

Orthogonality

У програмуванні ортогональність означає незалежність компонентів системи один від одного. Це означає, що зміна одного компонента не повинна впливати на роботу інших компонентів. Ортогональні системи зазвичай легше розробляти, тестувати та підтримувати, оскільки зміни в одній частині системи не викликають каскадних змін в інших частинах.

Переваги ортогональності:

- **Зменшення складності:** Ортогональні системи простіші для розуміння та керування, оскільки компоненти мають чітко визначені функції та не залежать один від одного.
- **Підвищення гнучкості:** Ортогональні системи легше адаптувати до нових вимог, оскільки зміни в одному компоненті не вимагають змін в інших компонентах.
- **Покращення повторного використання:** Ортогональні компоненти можна легко використовувати в інших проектах, оскільки вони не залежать від конкретного контексту.
- **Спрощення тестування:** Ортогональні компоненти можна тестувати ізольовано, що спрощує процес тестування та налагодження.

Застосування ортогональності:

- **Проектування модулів:** Створюйте модулі з чітко визначеними інтерфейсами та мінімальними залежностями між ними.
- **Використання принципів SOLID:** Принципи SOLID (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion) допомагають створювати ортогональні системи.
- **Уникнення глобальних змінних:** Глобальні змінні можуть призвести до непередбачуваних побічних ефектів та ускладнити розуміння системи.
- **Використання функціонального програмування:** Функціональне програмування сприяє створенню ортогональних систем, оскільки воно заохочує використання чистих функцій, які не мають побічних ефектів.

Декомпозиція

Декомпозиція - це процес розбиття складної системи на менші, більш керовані частини. Це дозволяє спростити проектування, розробку та підтримку системи, оскільки кожну частину можна розглядати та розробляти окремо.

Переваги декомпозиції:

- **Зменшення складності:** Розбиття системи на менші частини робить її більш зрозумілою та керованою.

- **Підвищення модульності:** Кожна частина системи може бути розроблена та протестована окремо, що спрощує процес розробки та підтримки.
- **Покращення повторного використання:** Частини системи, які були розроблені окремо, можуть бути легко використані в інших проектах.
- **Спрощення тестування:** Кожну частину системи можна тестувати ізольовано, що спрощує процес тестування та налагодження.

Застосування декомпозиції:

- **Функціональна декомпозиція:** Розбиття системи на функціональні модулі, кожен з яких відповідає за певну функцію.
- **Об'єктно-орієнтована декомпозиція:** Розбиття системи на об'єкти, кожен з яких представляє певну сутність реального світу.
- **Процедурна декомпозиція:** Розбиття системи на процедури або функції, кожна з яких виконує певну послідовність дій.

Information hiding

Приховування інформації - це принцип програмування, який полягає в тому, що деталі реалізації модуля повинні бути приховані від інших модулів. Це дозволяє змінювати внутрішню реалізацію модуля, не впливаючи на інші модулі, які його використовують.

Переваги приховування інформації:

- **Зменшення зв'язності:** Модулі стають менш залежними один від одного, що полегшує їх зміну та повторне використання.
- **Підвищення гнучкості:** Зміни у внутрішній реалізації модуля не вимагають змін в інших модулях.
- **Покращення ремонтопридатності:** Проблеми, що виникають у модулі, можна вирішити, не впливаючи на інші модулі.

Застосування приховування інформації:

- **Використання інтерфейсів:** Інтерфейси визначають контракт між модулями, приховуючи деталі реалізації.
- **Використання модифікаторів доступу:** Модифікатори доступу (public, private, protected) дозволяють контролювати, які частини модуля доступні іншим модулям.
- **Використання патернів проектування:** Деякі патерни проектування, такі як Фасад або Модуль, допомагають приховати інформацію.

Rule of least power

Цей принцип стверджує, що при виборі між різними технологіями або форматами для представлення даних слід вибирати найменш потужний варіант, який все ще може задовольнити потреби завдання. Іншими словами, не варто використовувати складні інструменти або формати, якщо простіші можуть виконати роботу.

Переваги Rule of Least Power:

- **Простота:** Простіші технології зазвичай легше зрозуміти, використовувати та підтримувати.
- **Сумісність:** Простіші формати даних, як правило, більш універсальні та сумісні з різними системами та інструментами.
- **Надійність:** Простіші рішення часто є більш надійними, оскільки вони мають менше точок відмови.
- **Безпека:** Простіші формати даних можуть бути менш вразливими до атак, оскільки вони мають менше можливостей для впровадження шкідливого коду.

Застосування Rule of Least Power:

- **Вибір мови розмітки:** Використовуйте HTML для структурування контенту, CSS для стилізації та JavaScript для інтерактивності, замість більш складних мов, таких як XML або Flash.
- **Вибір формату даних:** Використовуйте JSON або CSV для обміну даними, замість більш складних форматів, таких як XML.
- **Вибір протоколів:** Використовуйте HTTP для передачі даних, замість більш складних протоколів, таких як SOAP.

Pareto principle

Принцип Парето, також відомий як правило 80/20, стверджує, що приблизно 80% результатів виникають з 20% зусиль. У контексті програмування це означає, що більшість проблем або помилок у програмі викликані невеликою кількістю коду.

Застосування Pareto Principle:

- **Пріоритезація завдань:** Зосередьтеся на виправленні тих 20% коду, які викликають 80% проблем.
- **Оптимізація:** Оптимізуйте ті частини коду, які найчастіше виконуються або споживають найбільше ресурсів.
- **Тестування:** Зосередьтеся на тестуванні тих частин коду, які найімовірніше містять помилки.

Перфекціонізм

Перфекціонізм у програмуванні може бути як позитивним, так і негативним явищем. З одного боку, прагнення до досконалості може призвести до створення високоякісного та надійного коду. З іншого боку, надмірний перфекціонізм може призвести до прокрастинації, затримок у здачі проектів та вигорання.

Позитивні сторони перфекціонізму:

- **Висока якість коду:** Перфекціоністи прагнуть писати чистий, добре структурований та ефективний код.
- **Увага до деталей:** Перфекціоністи звертають увагу на найдрібніші деталі, що допомагає уникнути помилок.
- **Прагнення до самовдосконалення:** Перфекціоністи завжди прагнуть вчитися та розвиватися, що робить їх цінними членами команди.

Негативні сторони перфекціонізму:

- **Прокрастинація:** Перфекціоністи можуть відкладати завершення завдань, прагнучи досягти недосяжної досконалості.
- **Затримки у здачі проектів:** Прагнення до досконалості може призвести до затримок у здачі проектів.
- **Вигорання:** Постійне прагнення до досконалості може призвести до емоційного та фізичного виснаження.

Як знайти баланс:

- **Встановлення реалістичних цілей:** Важливо встановлювати досяжні цілі та не прагнути до недосяжної досконалості.
- **Визначення пріоритетів:** Зосередьтеся на найважливіших завданнях та не витрачайте час на дрібниці.
- **Делегування:** Не бійтеся делегувати завдання іншим членам команди.
- **Відпочинок:** Важливо робити перерви та відпочивати, щоб уникнути вигорання.

WET

WET - це антипатерн у програмуванні, який означає дублювання коду. Це означає, що один і той самий код або його фрагменти повторюються в різних місцях програми.

Недоліки WET:

- **Ускладнення підтримки:** Якщо потрібно внести зміни в код, який дублюється, то ці зміни потрібно внести в усі місця, де він повторюється. Це збільшує ризик помилок і ускладнює підтримку коду.

- **Збільшення розміру коду:** Дублювання коду призводить до збільшення розміру програми, що може негативно вплинути на її продуктивність.
- **Зниження читабельності:** Дубльований код ускладнює розуміння програми, оскільки один і той самий код потрібно читати кілька разів.

Як уникнути WET:

- **Використання функцій та методів:** Якщо певний код використовується кілька разів, його слід винести в окрему функцію або метод.
- **Використання класів та об'єктів:** Якщо кілька частин коду мають схожу структуру або поведінку, їх можна об'єднати в клас.
- **Використання шаблонів проектування:** Існують шаблони проектування, які допомагають уникнути дублювання коду, наприклад, шаблон "Стратегія" або "Фабричний метод".

Rule of three

Правило трьох - це евристика в рефакторингу, яка говорить, що якщо ви написали один і той самий код тричі, то його слід рефакторити, тобто винести в окрему функцію, метод або клас.

Переваги Rule of Three:

- **Уникнення дублювання коду:** Рефакторинг дозволяє уникнути дублювання коду та покращити його структуру.
- **Підвищення читабельності:** Код стає більш читабельним, оскільки один і той самий код не потрібно читати кілька разів.
- **Спрощення підтримки:** Зміни потрібно вносити лише в одному місці, а не в кількох копіях коду.

Chesterton’s fence

це принцип, який стверджує, що не слід щось змінювати або видаляти, якщо ви не розумієте, чому воно було створено. Цей принцип часто застосовується в контексті рефакторингу та внесення змін до існуючого коду.

Переваги Chesterton's Fence:

- **Уникнення непередбачуваних наслідків:** Якщо ви не розумієте, чому певний код був написаний, то його зміна або видалення може призвести до непередбачуваних наслідків.
- **Збереження цінного досвіду:** Існуючий код може містити цінний досвід або знання, які можуть бути втрачені при його зміні або видаленні.

- **Покращення розуміння коду:** Перш ніж змінювати або видаляти код, слід розібратися, чому він був написаний таким чином. Це допоможе краще зрозуміти код та уникнути помилок.

Застосування Chesterton's Fence:

- **Рефакторинг:** Перш ніж рефакторити код, слід розібратися, чому він був написаний таким чином. Це допоможе уникнути помилок та зберегти функціональність коду.
- **Видалення коду:** Перш ніж видаляти код, слід переконатися, що він дійсно не використовується і не є необхідним для роботи програми.

Single source of truth

Цей принцип стверджує, що для будь-якого елемента даних у системі повинен бути єдиний авторитетний джерело. Це означає, що кожен фрагмент даних зберігається лише в одному місці, і всі інші частини системи звертаються до цього джерела, щоб отримати актуальну інформацію.

Переваги SSOT:

- **Уникнення неузгодженості даних:** Оскільки дані зберігаються в одному місці, немає ризику, що різні частини системи будуть працювати з різними версіями одних і тих же даних.
- **Спрощення оновлення даних:** При оновленні даних потрібно змінити лише одне джерело, а не кілька копій.
- **Покращення надійності:** Зменшується ризик помилок, пов'язаних з дублюванням та несинхронізованим оновленням даних.

Застосування SSOT:

SSOT часто використовується в базах даних, системах управління контентом, хмарних сховищах та інших системах, де важлива цілісність даних.

MVP

Minimum Viable Product (MVP, Мінімально життєздатний продукт)

MVP - це версія продукту з мінімальним набором функцій, достатнім для того, щоб задовольнити початкові потреби користувачів та отримати зворотний зв'язок для подальшого розвитку.

Переваги MVP:

- **Швидкий вихід на ринок:** MVP дозволяє швидше випустити продукт та почати отримувати відгуки від користувачів.
- **Зниження ризиків:** Розробка MVP вимагає менше ресурсів, ніж повністю функціонального продукту, що знижує фінансові ризики.
- **Фокус на головному:** MVP допомагає зосередитися на найважливіших функціях продукту та уникнути розробки непотрібних функцій.

Застосування MVP:

MVP широко використовується в стартапах та при розробці нових продуктів, щоб перевірити ідею та отримати зворотний зв'язок від користувачів.

Hyrum's Law

Закон Хайрама стверджує, що з достатньою кількістю користувачів API, будь-яка поведінка, яку можна спостерігати, врешті-решт буде залежати від когось. Це означає, що навіть якщо поведінка системи не задокументована або не передбачена розробниками, користувачі можуть почати покладатися на неї, і будь-які зміни в цій поведінці можуть призвести до проблем сумісності.

Наслідки Hyrum's Law:

- **Обережність при зміні API:** Розробники повинні бути дуже обережними при внесенні змін до API, оскільки це може порушити роботу існуючих клієнтів.
- **Важливість зворотного зв'язку:** Важливо отримувати зворотний зв'язок від користувачів API, щоб розуміти, як вони його використовують та які функції їм потрібні.
- **Необхідність версійності:** Версійність API дозволяє вносити зміни, не порушуючи роботу існуючих клієнтів.

Застосування Hyrum's Law:

Закон Хайрама є важливим принципом при розробці та підтримці API, оскільки він допомагає розуміти, як зміни можуть вплинути на користувачів.

Сподіваюся, це пояснення було корисним! Якщо у вас є ще запитання, не соромтеся запитувати!

Cross-cutting concerns

це аспекти програми, які впливають на кілька модулів або компонентів, але не вписуються в традиційну ієрархію класів. Прикладами таких проблем є логування, обробка винятків, кешування, авторизація та безпека.

Принципи роботи з поперечними проблемами:

1. **Ідентифікація:** Визначте поперечні проблеми у вашій програмі. Це можуть бути функціональні вимоги (наприклад, логування) або нефункціональні вимоги (наприклад, продуктивність).
2. **Розділення:** Відокремте код, пов'язаний з поперечними проблемами, від основної бізнес-логіки. Це зробить ваш код більш модульним та легким для підтримки.
3. **Абстракція:** Створіть абстракції для поперечних проблем. Це можуть бути інтерфейси, класи або аспекти (в аспектно-орієнтованому програмуванні).
4. **Композиція:** Використовуйте механізми композиції, щоб об'єднати основну бізнес-логіку з поперечними проблемами. Це може бути зроблено за допомогою шаблонів проектування (наприклад, декоратор, проксі) або за допомогою фреймворків (наприклад, Spring).

Переваги використання принципів cross-cutting concerns:

- **Підвищення модульності:** Код стає більш модульним, оскільки поперечні проблеми відокремлені від основної бізнес-логіки.
- **Покращення повторного використання:** Код, пов'язаний з поперечними проблемами, може бути легко використаний в інших частинах програми або в інших проектах.
- **Спрощення тестування:** Код, пов'язаний з поперечними проблемами, може бути протестований окремо від основної бізнес-логіки.
- **Підвищення гнучкості:** Зміни в поперечних проблемах можуть бути внесені без впливу на основну бізнес-логіку.

Scout rule

це принцип програмування, який говорить: "Залишайте код чистішим, ніж ви його знайшли". Це означає, що кожного разу, коли ви працюєте з кодом, ви повинні намагатися покращити його, навіть якщо це невеликі зміни.**Приклади застосування Scout Rule:**

- Виправлення помилок форматування.
- Додавання коментарів або документації.
- Рефакторинг коду для покращення читабельності або продуктивності.
- Видалення непотрібного коду.

Переваги використання Scout Rule:

- **Поліпшення якості коду:** Код стає більш читабельним, зрозумілим та підтримуваним.

- **Зменшення технічного боргу:** Технічний борг - це вартість додаткової роботи, спричиненої вибором швидкого, але неідеального рішення. Дотримання Scout Rule допомагає зменшити технічний борг.
- **Підвищення продуктивності команди:** Розробникам легше працювати з чистим та добре структурованим кодом.

Law of leaky abstractions

у програмуванні стверджує, що всі нетривіальні абстракції певною мірою є "дірявими", тобто вони не можуть повністю приховати деталі реалізації, яку вони абстрагують. Це означає, що рано чи пізно розробникам доведеться зіткнутися з деталями нижчого рівня, які абстракція намагалася приховати.

Суть принципу:

- **Абстракції спрощують складність:** Абстракції є важливим інструментом у програмуванні, оскільки вони дозволяють працювати зі складними системами на більш високому рівні, не вникаючи у всі деталі їх реалізації.
- **Абстракції не ідеальні:** Однак, жодна абстракція не є досконалою. Завжди будуть випадки, коли деталі реалізації "просочуються" крізь абстракцію та впливають на поведінку програми.
- **Знання деталей важливе:** Для ефективного використання абстракцій та вирішення проблем, які виникають через їхню "дірявість", розробники повинні розуміти, як працює абстракція та які деталі вона приховує.

Приклади "дірявих" абстракцій:

- **Мережеві протоколи:** Абстрагують деталі передачі даних по мережі, але можуть виникати проблеми з затримками, втратою пакетів тощо.
- **Бази даних:** Абстрагують деталі зберігання даних, але можуть виникати проблеми з продуктивністю, блокуваннями та транзакціями.
- **Файлові системи:** Абстрагують деталі зберігання файлів на диску, але можуть виникати проблеми з фрагментацією, дозволами доступу тощо.

Як працювати з "дірявими" абстракціями:

1. **Вивчення абстракції:** Необхідно розуміти, як працює абстракція та які деталі вона приховує.
2. **Виявлення "витоків":** Звертайте увагу на випадки, коли деталі реалізації впливають на поведінку програми.
3. **Врахування "витоків":** При проектуванні та написанні коду враховуйте можливість виникнення "витоків" та передбачайте механізми їх обробки.
4. **Готовність до "занурення":** Будьте готові зануритися в деталі реалізації абстракції, якщо це необхідно для вирішення проблеми.

Zero-cost abstractions

Принцип zero-cost abstractions (абстракції з нульовою вартістю) в програмуванні стверджує, що використання абстракцій не повинно призводити до погіршення продуктивності коду порівняно з еквівалентною реалізацією без абстракцій. Іншими словами, абстракції не повинні мати додаткових витрат під час виконання програми.

Основні ідеї принципу zero-cost abstractions:

1. **Відсутність накладних витрат:** Абстракції не повинні вносити додаткових обчислень або використання пам'яті під час виконання програми.
2. **Оптимізація під час компіляції:** Компілятор повинен бути здатним оптимізувати абстракції таким чином, щоб вони не впливали на продуктивність коду.
3. **Ефективність на рівні ручного коду:** Код, написаний з використанням абстракцій, повинен бути таким же ефективним, як і код, написаний вручну без використання абстракцій.

Переваги використання zero-cost abstractions:

- **Підвищення продуктивності розробки:** Абстракції дозволяють писати код на більш високому рівні, що спрощує його розуміння та підтримку.
- **Покращення читабельності коду:** Абстракції роблять код більш виразним та зрозумілим.
- **Зменшення кількості помилок:** Абстракції допомагають уникнути типових помилок, пов'язаних з низькорівневим програмуванням.
- **Підвищення гнучкості коду:** Абстракції дозволяють легше змінювати та розширювати код.

Приклади zero-cost abstractions:

- **Шаблони (generics) в C++:** Дозволяють писати код, який працює з різними типами даних, не втрачаючи при цьому продуктивності.
- **Ітератори в C++:** Дозволяють проходити по колекціям даних, не вникаючи в деталі їх реалізації.
- **Вбудовані функції (inline functions) в C++:** Дозволяють уникнути накладних витрат на виклик функцій.
- **Макроси в Rust:** Дозволяють генерувати код під час компіляції, уникаючи накладних витрат під час виконання.

Важливо зазначити:

Не всі абстракції є zero-cost. Деякі абстракції можуть мати незначні накладні витрати, але вони зазвичай компенсуються перевагами, які надають абстракції.

Abstraction layers

це концепція в програмуванні, яка полягає у розділенні програмної системи на окремі рівні, кожен з яких має певну функціональність та приховує деталі реалізації від нижчих рівнів. Це дозволяє створювати більш модульні, гнучкі та легкі в підтримці системи.

Основні ідеї принципу рівнів абстракції:

1. **Інкапсуляція:** Кожен рівень приховує деталі своєї реалізації та надає інтерфейс для взаємодії з іншими рівнями. Це дозволяє змінювати внутрішню логіку рівня, не впливаючи на інші рівні системи.
2. **Модульність:** Рівні абстракції розбивають систему на незалежні модулі, кожен з яких має свою відповідальність. Це полегшує розробку, тестування та підтримку системи, оскільки кожен модуль можна розробляти та тестувати окремо.
3. **Гнучкість:** Рівні абстракції дозволяють легко змінювати та розширювати систему. Наприклад, можна замінити один рівень іншим, не змінюючи інші рівні системи.
4. **Повторне використання:** Модулі, створені на основі рівнів абстракції, можна повторно використовувати в інших проектах.

Приклади рівнів абстракції в програмуванні:

- **Апаратний рівень:** Нижчий рівень, який взаємодіє безпосередньо з апаратними компонентами комп'ютера.
- **Рівень операційної системи:** Надає інтерфейс для взаємодії з апаратним рівнем та керує ресурсами комп'ютера.
- **Рівень бібліотек та фреймворків:** Надає готові компоненти та функціональність для розробки програмного забезпечення.
- **Рівень прикладних програм:** Використовує бібліотеки та фреймворки для створення конкретних програм.

Переваги використання рівнів абстракції:

- **Спрощення розробки:** Розробники можуть зосередитися на конкретному рівні, не вникаючи в деталі реалізації інших рівнів.
- **Полегшення підтримки:** Зміни в одному рівні не впливають на інші рівні, що полегшує підтримку та оновлення системи.
- **Підвищення гнучкості:** Можливість легко змінювати та розширювати систему.
- **Повторне використання коду:** Модулі, створені на основі рівнів абстракції, можна використовувати в інших проектах.

Недоліки використання рівнів абстракції:

- **Зниження продуктивності:** Додаткові рівні абстракції можуть призвести до зниження продуктивності системи.
- **Збільшення складності:** Наявність багатьох рівнів абстракції може ускладнити розуміння системи.

Occam’s razor

означає, що при наявності кількох рішень задачі слід вибирати найпростіше з них, тобто те, яке вимагає найменшої кількості припущень, має менше залежностей та менш складну логіку.

Як застосовувати Occam's razor у програмуванні:

1. **Вибір алгоритму:** Якщо є кілька алгоритмів, які вирішують задачу, слід вибирати той, який є більш простим для розуміння та реалізації. Наприклад, якщо потрібно відсортувати масив, можна вибрати сортування бульбашкою (bubble sort) замість швидкого сортування (quick sort), якщо розмір масиву невеликий і простота реалізації важливіша за швидкість виконання.
2. **Проектування архітектури:** При проектуванні архітектури програмного забезпечення слід уникати зайвих шарів абстракції та компонентів. Чим простіша архітектура, тим легше її розуміти, підтримувати та модифікувати.
3. **Написання коду:** При написанні коду слід прагнути до його простоти та читабельності. Слід уникати використання складних конструкцій, якщо їх можна замінити більш простими та зрозумілими. Код має бути добре структурованим, з використанням осмислених імен змінних та функцій.
4. **Вибір бібліотек та фреймворків:** При виборі бібліотек та фреймворків слід віддавати перевагу тим, які є більш простими у використанні та мають менше залежностей. Це дозволить уникнути непотрібної складності та полегшить процес розробки.
5. **Тестування та налагодження:** При тестуванні та налагодженні коду слід починати з найпростіших тестів та припущень про причини помилок. Це допоможе швидше знайти та виправити проблеми.

Переваги використання принципу Occam's razor у програмуванні:

- **Зменшення складності:** Простіший код легше розуміти, підтримувати та модифікувати.
- **Зменшення кількості помилок:** Простіший код менш схильний до помилок, оскільки він має менше місць, де можуть виникнути проблеми.
- **Підвищення продуктивності:** Розробники витрачають менше часу на розуміння та налагодження простого коду, що дозволяє їм швидше виконувати завдання.
- **Покращення співпраці:** Простіший код легше обговорювати та спільно розробляти з іншими членами команди.

Патерни (завдання 2)

1. Decorator
2. Composite
3. Strategy
4. Flyweight
5. Memento
6. Iterator
7. Builder
8. Template Method
9. Visitor
10. Chain of Responsibility
11. Proxy
12. Prototype
13. Command
14. State
15. Mediator
16. Facade
17. Singleton
18. Abstract Factory
19. Observer
20. Bridge
21. Adapter
22. Interpreter
23. Factory Method

Decorator

Звісно! Патерн **Decorator** - це структурний патерн проектування, який дозволяє динамічно додавати нові функції до об’єктів, обгортаючи їх у корисні “обгортки”. Він корисний для розширення функціональності об’єктів без зміни їх коду.

Ось ключові моменти патерна Decorator:

- **Базовий Компонент:** визначає інтерфейс для об’єктів, які можуть мати декорації.
- **Конкретний Компонент:** клас, який реалізує базовий компонент і визначає об’єкт, до якого можна додати нову поведінку.
- **Базовий Декоратор:** клас, який має посилання на об’єкт базового компонента і реалізує його інтерфейс, перенаправляючи запити до компонента.
- **Конкретний Декоратор:** клас, який наслідується від базового декоратора і реалізує додаткову поведінку перед або після виклику методу обгорнутого компонента.

Патерн **Decorator** часто використовується у програмуванні для розширення функціональності класів у мовах з сильною типізацією, таких як C# або Java, де зміна класу після його створення може бути складною.

Composite

Патерн **Composite** - це структурний патерн проектування, який дозволяє складати об’єкти в деревоподібні структури для представлення ієрархій частин-ціле. Цей патерн дозволяє клієнтам однаково обробляти індивідуальні об’єкти та складені групи об’єктів.

Ось ключові моменти патерна Composite:

- **Компонент**: інтерфейс, який описує операції, які можуть бути виконані як окремими об’єктами, так і їх комбінаціями.
- **Лист (Leaf)**: представляє окремий об’єкт у композиції. Лист не має дочірніх елементів.
- **Композит (Composite)**: клас, який реалізує компоненти з дочірніми елементами. Він зберігає компоненти та реалізує операції компонента, використовуючи цих дочірніх елементів.
- **Клієнт**: взаємодіє з елементами через інтерфейс компонента.

Патерн **Composite** часто використовується для створення складних деревоподібних структур, таких як графічний інтерфейс користувача, системи файлової ієрархії або будь-яка структура, де компоненти можуть бути як простими, так і складеними.

Strategy

Патерн **Strategy** - це поведінковий патерн проектування, який дозволяє визначити сімейство алгоритмів, інкапсулювати кожен з них окремо та зробити їх взаємозамінними. Патерн **Strategy** дозволяє алгоритмам змінюватися незалежно від клієнтів, які їх використовують.

Ось ключові моменти патерна Strategy:

- **Контекст (Context)**: клас, який містить посилання на стратегію. Він делегує виконання алгоритму стратегії.
- **Стратегія (Strategy)**: інтерфейс, який описує, як виконувати алгоритм.
- **Конкретна Стратегія (Concrete Strategy)**: конкретні реалізації стратегії, кожна з яких інкапсулює свою версію алгоритму.

Патерн **Strategy** часто використовується, коли є кілька способів виконання завдання, і ми хочемо зберегти гнучкість у виборі потрібного алгоритму. Наприклад, сортування даних може бути реалізовано за допомогою різних алгоритмів, і патерн **Strategy** дозволяє легко перемикатися між ними.

Flyweight

Патерн **Flyweight** - це структурний патерн проектування, який дозволяє використовувати спільні об’єкти таким чином, щоб можна було ефективно підтримувати велику кількість дрібних об’єктів.

Ось ключові моменти патерна Flyweight:

- **Flyweight**: інтерфейс через який flyweights можуть отримувати та діяти на зовнішній стан.
- **Конкретний Flyweight**: реалізація інтерфейсу Flyweight, яка може поділятися. Стан, який є спільним для багатьох об’єктів, зберігається в ньому.
- **Flyweight Factory**: створює та керує flyweights, забезпечуючи правильне спільне використання ними.
- **Клієнт**: використовує фабрику для створення та керування flyweights, зберігаючи зовнішній стан, який повинен бути переданий у flyweight.

Патерн **Flyweight** часто використовується для оптимізації використання пам’яті у програмах, де є багато подібних об’єктів з однаковим станом. Наприклад, у графічних програмах для представлення символів у текстовому редакторі або у іграх для представлення багатьох однакових об’єктів на сцені.

Memento

Патерн **Memento** - це поведінковий патерн проектування, який дозволяє зберігати та відновлювати попередній стан об'єкта без розкриття деталей його реалізації.

Ось ключові моменти патерна Memento:

- **Memento**: об'єкт, який зберігає стан іншого об'єкта, відомий як оригінатор.
- **Оригінатор (Originator)**: об'єкт, стан якого потребує збереження та відновлення.
- **Опікун (Caretaker)**: об'єкт, який знає, коли та чому оригінатору потрібно зберегти та відновити стан, але не знає деталей реалізації стану.

Патерн **Memento** часто використовується для реалізації функціональності відкату (undo) або для збереження стану об'єкта у випадку помилки для подальшого відновлення.

Iterator

Патерн **Iterator** - це поведінковий патерн проектування, який дозволяє послідовно перебирати елементи складної структури даних без необхідності розкривати її внутрішнє представлення.

Ось ключові моменти патерна Iterator:

- **Iterator**: інтерфейс, який надає методи для перебору колекції.
- **Конкретний Ітератор (Concrete Iterator)**: реалізація інтерфейсу Iterator, яка знає, як перебирати елементи конкретної колекції.
- **Агрегат (Aggregate)**: інтерфейс, який описує методи для створення об'єкта ітератора для перебору своїх елементів.
- **Конкретний Агрегат (Concrete Aggregate)**: реалізація інтерфейсу Aggregate, яка створює конкретний ітератор для перебору своїх елементів.

Патерн **Iterator** часто використовується у програмуванні для надання стандартного способу доступу та перебору елементів колекції, незалежно від її конкретної реалізації.

Builder

Патерн **Builder** - це створювальний патерн проектування, який дозволяє створювати складні об'єкти крок за кроком. Патерн відокремлює конструювання складного об'єкта від його представлення, так що той самий процес конструювання може створювати різні представлення.

Ось ключові моменти патерна Builder:

- **Builder**: інтерфейс, який описує методи для створення різних частин складного об’єкта.
- **Конкретний Будівельник (Concrete Builder)**: реалізація інтерфейсу Builder, яка конструює та збирає частини продукту, а також забезпечує інтерфейс для отримання кінцевого продукту.
- **Директор (Director)**: клас, який визначає порядок виклику методів будівельника для створення продукту.
- **Продукт (Product)**: об’єкт, який будується за допомогою патерна Builder.

Патерн **Builder** часто використовується, коли необхідно створити об’єкт з багатьма можливими конфігураціями; наприклад, при створенні складних запитів або при конфігурації складних об’єктів.

Template Method

Патерн **Template Method** - це поведінковий патерн проектування, який визначає скелет алгоритму у базовому класі, дозволяючи підкласам перевизначати певні кроки алгоритму без зміни його структури.

Ось ключові моменти патерна Template Method:

- **Абстрактний Клас (Abstract Class)**: клас, який містить шаблонний метод і абстрактні операції, які підкласи повинні реалізувати.
- **Конкретний Клас (Concrete Class)**: клас, який реалізує абстрактні операції базового класу для виконання конкретних кроків алгоритму.

Патерн **Template Method** часто використовується, коли є кілька класів, які роблять подібні дії з невеликими варіаціями. Використання цього патерна дозволяє уникнути дублювання коду, визначивши загальний алгоритм у базовому класі та дозволяючи підкласам забезпечити конкретну реалізацію.

Visitor

Патерн **Visitor** - це поведінковий патерн проектування, який дозволяє додавати нові операції до існуючих класів без зміни їх коду.

Ось ключові моменти патерна Visitor:

- **Visitor**: інтерфейс, який описує операцію, яка має бути виконана на елементах структури об’єктів.
- **Конкретний Візитер (Concrete Visitor)**: клас, який реалізує інтерфейс Visitor, визначаючи операції для конкретних класів елементів.
- **Елемент (Element)**: інтерфейс, який описує метод `accept`, який приймає візитера.

- **Конкретний Елемент (Concrete Element)**: клас, який реалізує інтерфейс Element, дозволяючи візитеру виконати конкретну операцію на ньому.

Патерн **Visitor** часто використовується для реалізації операцій, які можуть бути застосовані до групи об’єктів різних класів або для збору статистичної інформації про об’єкти без зміни їх класів.

Chain of Responsibility

Патерн **Chain of Responsibility** - це поведінковий патерн проектування, який дозволяє передавати запити вздовж ланцюга обробників. При цьому, кожен обробник вирішує, чи може він обробити запит самостійно, або ж його потрібно передати наступному обробнику в ланцюгу.

Ось ключові моменти патерна Chain of Responsibility:

- **Обробник (Handler)**: інтерфейс, який описує метод для обробки запитів.
- **Конкретний Обробник (Concrete Handler)**: клас, який реалізує інтерфейс Handler і обробляє запити, якщо це можливо, або передає їх наступному обробнику в ланцюгу.
- **Клієнт (Client)**: клас, який ініціює запит та передає його першому обробнику в ланцюгу.

Патерн **Chain of Responsibility** часто використовується для зменшення залежностей між відправником та одержувачем запиту. Він також дозволяє динамічно додавати або змінювати обробників у ланцюгу.

Proxy

Патерн **Proxy** - це структурний патерн проектування, який дозволяє забезпечити замінник або запобіжник для іншого об’єкта, щоб контролювати доступ до нього.

Ось ключові моменти патерна Proxy:

- **Proxy**: клас, який містить посилання на реальний об’єкт, імітує його інтерфейс та перенаправляє всі запити до реального об’єкта.
- **Реальний Об’єкт (Real Subject)**: клас, який визначає реальний об’єкт, який використовується програмою.
- **Суб’єкт (Subject)**: інтерфейс, який описує загальний інтерфейс для Proxy та Real Subject.

Патерн **Proxy** часто використовується для контролю доступу до об’єкта, затримки його створення та ініціалізації до тих пір, поки він дійсно не потрібен, або для реалізації додаткової функціональності при доступі до об’єкта, такої як логування операцій.

Prototype

Патерн **Prototype** - це створювальний патерн проектування, який дозволяє копіювати існуючі об’єкти без зроблення коду, залежного від їх класів.

Ось ключові моменти патерна Prototype:

- **Прототип (Prototype)**: інтерфейс, який описує методи для клонування об’єктів.
- **Конкретний Прототип (Concrete Prototype)**: клас, який реалізує інтерфейс Prototype і метод клонування.

Патерн **Prototype** часто використовується, коли система не повинна залежати від класів конкретних об’єктів, які потрібно створити, або коли класи об’єктів інстанціюються динамічно в процесі виконання.

Command

Патерн **Command** - це поведінковий патерн проектування, який перетворює запити або прості операції на об’єкти. Це дозволяє параметризувати об’єкти за запитами, чергами або операціями та забезпечує можливість відкладення виконання операцій, відміни їх або ведення журналу.

Ось ключові моменти патерна Command:

- **Команда (Command)**: інтерфейс, який описує метод `execute`, який викликається для виконання команди.
- **Конкретна Команда (Concrete Command)**: клас, який реалізує інтерфейс Command і визначає зв’язок між об’єктом-отримувачем та діями, які потрібно виконати.
- **Отримувач (Receiver)**: клас, який знає, як виконати операції, необхідні для виконання запиту.
- **Викликач (Invoker)**: клас, який зберігає команду та викликає її метод `execute` для виконання запиту.
- **Клієнт (Client)**: клас, який створює конкретну команду та визначає її отримувача.

Патерн **Command** часто використовується для розмежування компонентів системи, які виконують операції, від тих, хто їх ініціює. Це також дозволяє легко додавати нові команди до системи без зміни існуючих класів.

State

Патерн **State** - це поведінковий патерн проектування, який дозволяє об'єкту змінювати свою поведінку, коли змінюється його внутрішній стан. Це виглядає так, ніби об'єкт змінив свій клас.

Ось ключові моменти патерна State:

- **Стан (State)**: інтерфейс, який описує методи, пов'язані з поведінкою об'єкта.
- **Конкретний Стан (Concrete State)**: клас, який реалізує інтерфейс State і визначає поведінку об'єкта для певного стану.
- **Контекст (Context)**: клас, який містить посилання на екземпляр класу State, який представляє поточний стан об'єкта.

Патерн **State** часто використовується, коли в об'єкті є багато умовних операторів, які залежать від його стану. Замість того, щоб зберігати численні умови всередині одного класу, можна розподілити їх між різними класами стану.

Mediator

Патерн **Mediator** - це поведінковий патерн проектування, який зменшує складність зв'язків між об'єктами системи, надаючи об'єкт, який виступає як посередник та централізований контролер взаємодії між об'єктами.

Ось ключові моменти патерна Mediator:

- **Посередник (Mediator)**: інтерфейс, який описує метод для спілкування з компонентами.
- **Конкретний Посередник (Concrete Mediator)**: клас, який реалізує інтерфейс Mediator і координує взаємодію між різними компонентами.
- **Компоненти (Colleagues)**: класи, які спілкуються один з одним через інтерфейс Mediator замість прямих зв'язків.

Патерн **Mediator** часто використовується для спрощення взаємодії між компонентами системи, особливо коли кожен компонент може спілкуватися з багатьма іншими. Це дозволяє зменшити кількість зв'язків між класами та спростити їх зміни та розширення.

Facade

Патерн **Facade** - це структурний патерн проектування, який надає простий інтерфейс до складної системи класів, бібліотеки або фреймворку.

Ось ключові моменти патерна Facade:

- **Фасад (Facade)**: клас, який надає простий інтерфейс до складної системи, роблячи систему легшою для використання.
- **Складні Системи (Complex Systems)**: класи, які мають складний інтерфейс та взаємодіють один з одним на різних рівнях.

Патерн **Facade** часто використовується для надання простого інтерфейсу до складних систем, таких як бібліотеки або API. Це дозволяє ізолювати складності системи від її користувачів та зменшити залежності між системами.

Singleton

Патерн **Singleton** - це створювальний патерн проектування, який гарантує, що клас має лише один екземпляр, і надає глобальну точку доступу до цього екземпляра.

Ось ключові моменти патерна Singleton:

- **Singleton**: клас, який обмежує інстанціювання себе до одного об’єкта.
- **Глобальний Доступ**: Singleton надає статичний метод, який повертає єдиний екземпляр класу.

Патерн **Singleton** часто використовується для управління спільними ресурсами, такими як підключення до бази даних або налаштування системи. Важливо використовувати цей патерн обережно, оскільки він може ускладнити тестування та може призвести до проблем з багатопоточністю.

Abstract Factory

Патерн **Abstract Factory** - це створювальний патерн проектування, який надає інтерфейс для створення сімейств пов’язаних або залежних об’єктів без вказівки їх конкретних класів.

Ось ключові моменти патерна Abstract Factory:

- **Абстрактна Фабрика (Abstract Factory)**: інтерфейс, який описує методи для створення абстрактних продуктів.
- **Конкретна Фабрика (Concrete Factory)**: клас, який реалізує інтерфейс Abstract Factory і створює конкретні продукти.
- **Абстрактні Продукти (Abstract Products)**: інтерфейси для різних типів продуктів, які можуть бути створені.
- **Конкретні Продукти (Concrete Products)**: класи, які реалізують інтерфейси Abstract Products і визначають продукти, які будуть створені.

Патерн **Abstract Factory** часто використовується, коли система повинна бути незалежною від способу створення, композиції та представлення своїх продуктів. Це також дозволяє замінити сімейства продуктів легко, змінивши конкретну фабрику.

Observer

Патерн **Observer** - це поведінковий патерн проектування, який створює відносини один-до-багатьох між об’єктами таким чином, що коли один об’єкт змінює свій стан, всі його залежні об’єкти повідомляються і оновлюються автоматично.

Ось ключові моменти патерна Observer:

- **Суб’єкт (Subject)**: інтерфейс, який визначає методи для приєднання, від’єднання та повідомлення спостерігачів.
- **Конкретний Суб’єкт (Concrete Subject)**: клас, який реалізує інтерфейс Subject і має стан, про зміни якого повідомляють спостерігачі.
- **Спостерігач (Observer)**: інтерфейс, який визначає метод оновлення, який викликається суб’єктом.
- **Конкретний Спостерігач (Concrete Observer)**: клас, який реалізує інтерфейс Observer і реагує на повідомлення від суб’єкта.

Патерн **Observer** часто використовується для реалізації розподілених подійових систем, моделей даних та інтерфейсу користувача.

Bridge

Патерн **Bridge** - це структурний патерн проектування, який розділяє абстракцію від її реалізації, так що обидві можуть змінюватися незалежно одна від одної.

Ось ключові моменти патерна Bridge:

- **Абстракція (Abstraction)**: інтерфейс, який визначає базовий інтерфейс і зберігає посилання на об’єкт реалізації.
- **Уточнена Абстракція (Refined Abstraction)**: розширює інтерфейс визначений Абстракцією.
- **Реалізатор (Implementor)**: інтерфейс для класів реалізації, які визначають інтерфейс для конкретних реалізацій.
- **Конкретний Реалізатор (Concrete Implementor)**: класи, які реалізують інтерфейс Реалізатора.

Патерн **Bridge** дозволяє змінювати абстракції та їх реалізації незалежно одна від одної, що сприяє гнучкості та незалежності класових ієрархій.

Adapter

Патерн **Adapter** - це структурний патерн проектування, який дозволяє об’єктам з несумісними інтерфейсами працювати разом.

Ось ключові моменти патерна Adapter:

- **Ціль (Target)**: інтерфейс, який клієнт очікує використовувати.
- **Адаптер (Adapter)**: клас, який реалізує інтерфейс Target і перетворює інтерфейс одного або декількох класів на інтерфейс, який очікує клієнт.
- **Адаптований (Adaptee)**: існуючий клас з інтерфейсом, який потребує адаптації.

Патерн **Adapter** часто використовується для забезпечення спілкування між новими та існуючими системами, що мають різні інтерфейси.

Interpreter

Патерн **Interpreter** - це поведінковий патерн проектування, який визначає представлення граматики для мови та інтерпретатор, який використовує представлення для інтерпретації виразів у мові.

Ось ключові моменти патерна Interpreter:

- **Абстрактний Вираз (Abstract Expression)**: оголошує інтерфейс для виконання операції.
- **Термінальний Вираз (Terminal Expression)**: реалізує методи, визначені в Абстрактному Виразі, для термінальних символів граматики.
- **Нетермінальний Вираз (Nonterminal Expression)**: один або декілька об’єктів Terminal або Nonterminal Expressions, які визначають складові для граматичних конструкцій.
- **Контекст (Context)**: містить інформацію, яка глобальна для інтерпретатора.
- **Клієнт (Client)**: будує (або отримує) абстрактне синтаксичне дерево, представляюче окремий вираз у граматиці. Потім клієнт інтерпретує це дерево.

Патерн **Interpreter** часто використовується для інтерпретації мов програмування або для обробки мовних конструкцій у рамках програмного забезпечення.

Factory Method

Патерн **Factory Method** - це створювальний патерн проектування, який визначає інтерфейс для створення об’єкта, але дозволяє підкласам змінювати тип створюваних об’єктів.

Ось ключові моменти патерна Factory Method:

- **Творець (Creator)**: клас, який оголошує метод фабрики, який повертає об’єкт типу Product. Творець може також визначити реалізацію методу фабрики за замовчуванням.
- **Конкретний Творець (Concrete Creator)**: підклас Творця, який перевизначає фабричний метод для повернення екземпляра конкретного продукту.
- **Продукт (Product)**: інтерфейс для об’єктів, які створює метод фабрики.

- **Конкретний Продукт (Concrete Product)**: клас, який реалізує інтерфейс Product і визначає продукт, який буде створений конкретним творцем.

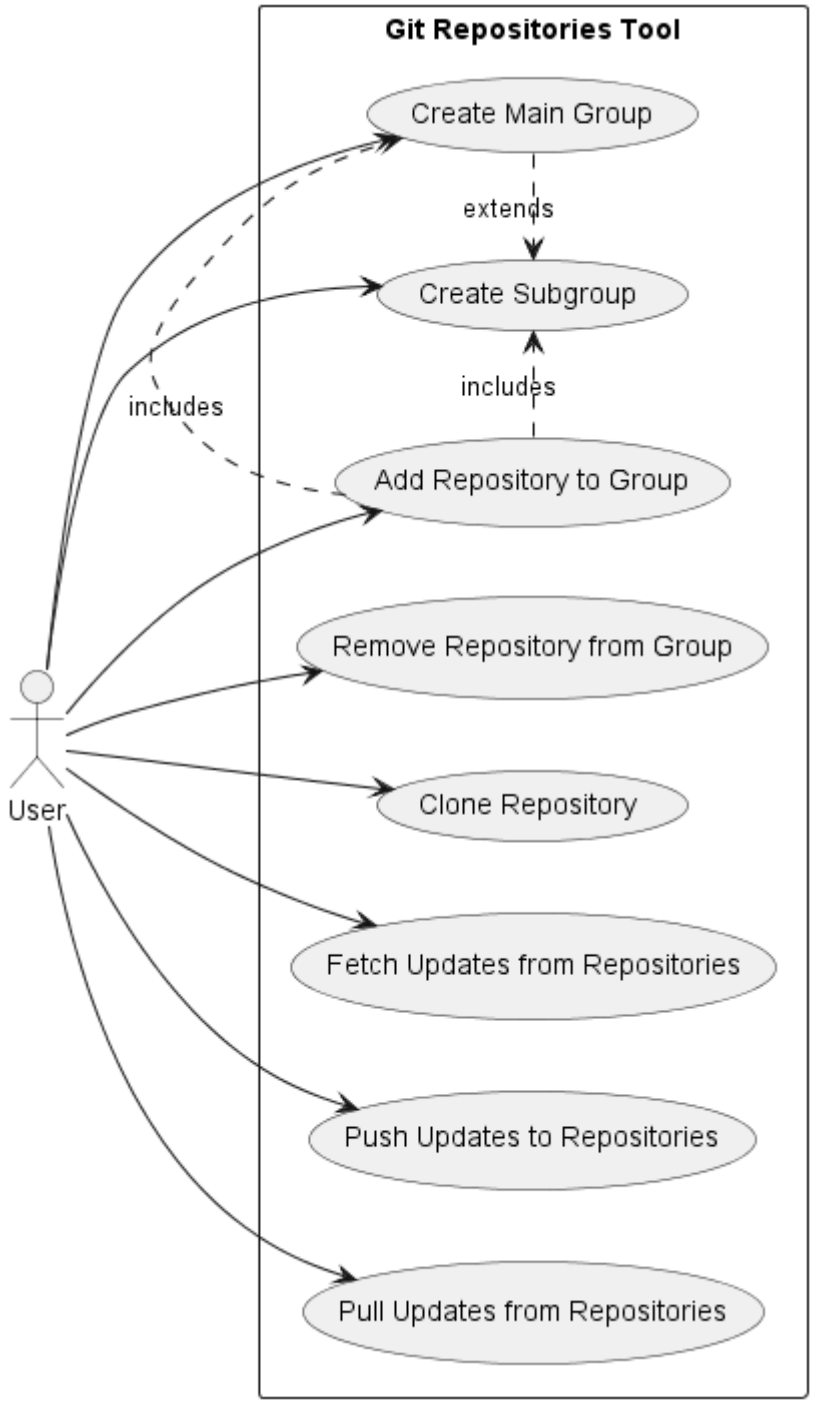
Патерн **Factory Method** часто використовується для делегування логіки створення об’єктів підкласам, що дозволяє системі бути більш гнучкою при введенні нових типів продуктів.

UML (завдання 3)

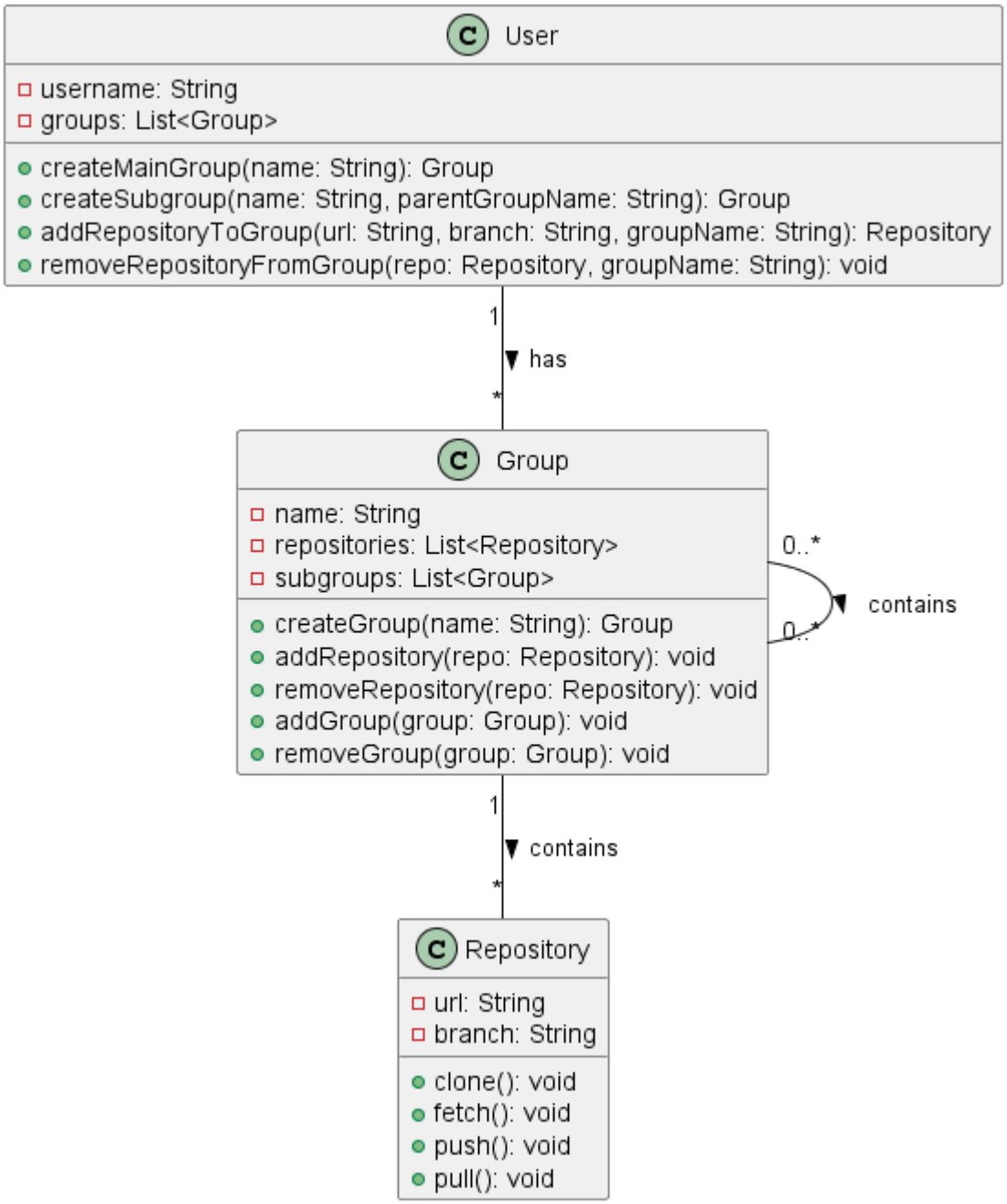
1. **засіб для роботи з багатьма репозиторіями git одночасно (можливість створення ієрархії груп для репозиторіїв)**
2. **засіб для генерації коду-заготовки для патернів проєктування на заданій мові програмування (можливість створення ієрархії груп для патернів)**
3. **засіб для пошуку копій програмного коду (плагіату) (можливість пошуку по переліку локальних каталогів)**
4. **універсальну платформу для підтримки різних настільних ігор (можливість створення ієрархії груп для ігор)**
5. **засіб для генерації стандартних фрагментів коду (code snippets) на різних мовах програмування (можливість створення ієрархії груп для фрагментів коду)**
6. **засіб для ідентифікації студентів в аудиторії (наприклад, відвідування лекцій) за фотографією аудиторії (можливість додавання декількох фотографій одного студента)**
7. **засіб для видалення непотрібних файлів з ієрархії каталогів (наприклад, чистка коду від згенерованих артефактів) (можливість додавання переліку локальних каталогів для обробки)**
8. **засіб для генерації документів на основі шаблону та заповнення даних (наприклад, білети до іспиту) (можливість створення ієрархії груп для шаблонів)**
9. **бюрократичний навігатор – засіб для зберігання інформації про бюрократичні процедури, підказки щодо оптимального їх проходження (можливість пошуку бюрократичних процедур за різними параметрами – ключовими словами, організаціями куди треба звертатись, кількістю кроків тощо)**
10. **систему для вивчення університетської математики (мат. аналіз, алгебра, теорія ймовірностей, ...) в ігровій формі (можливість додавання нових ігрових активностей, зокрема одразу до декількох предметів)**
11. **засіб для time tracking (відслідковування часу виконання різних завдань розробником) з інтеграцією із зовнішніми системами (IDE, файлова система, веб сервіси, ...) (можливість перегляду часу виконання різних груп активностей)**

Засіб для роботи з багатьма репозиторіями git одночасно (можливість створення ієрархії груп для репозиторіїв)

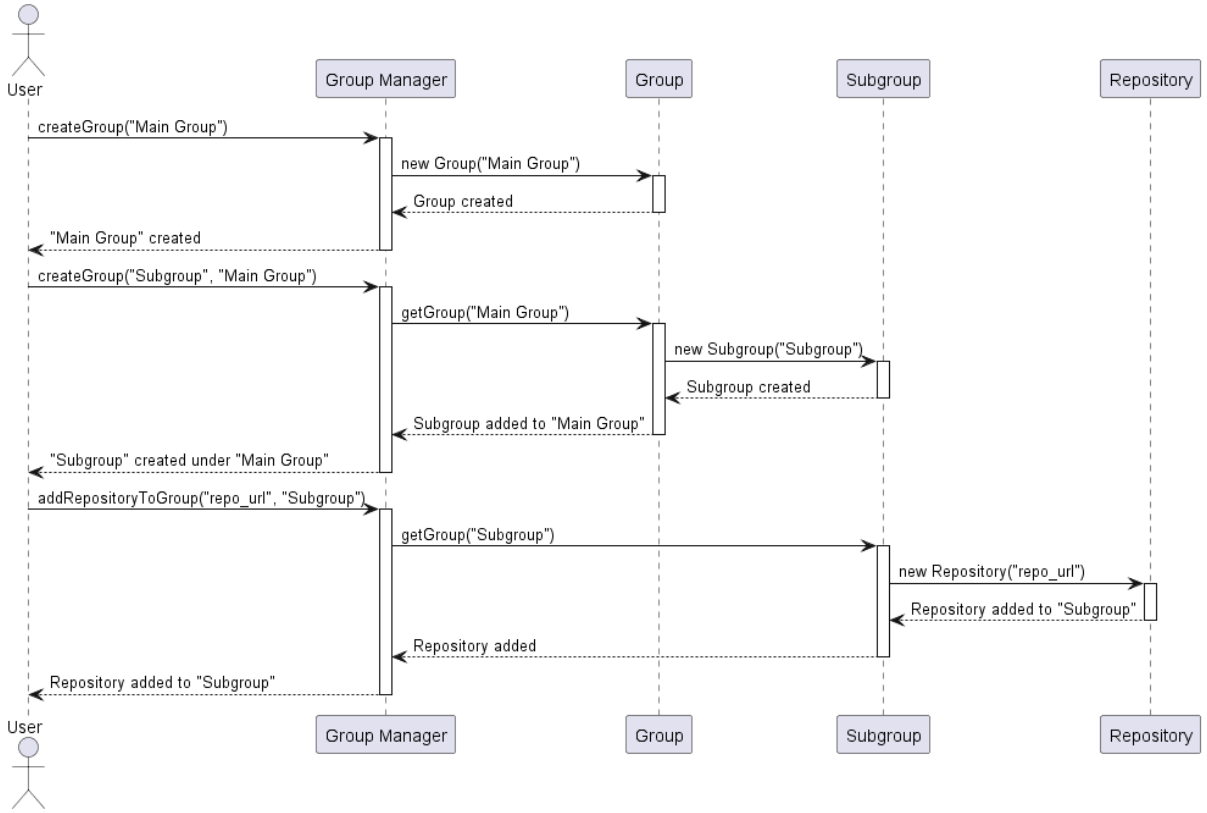
Use case



Class

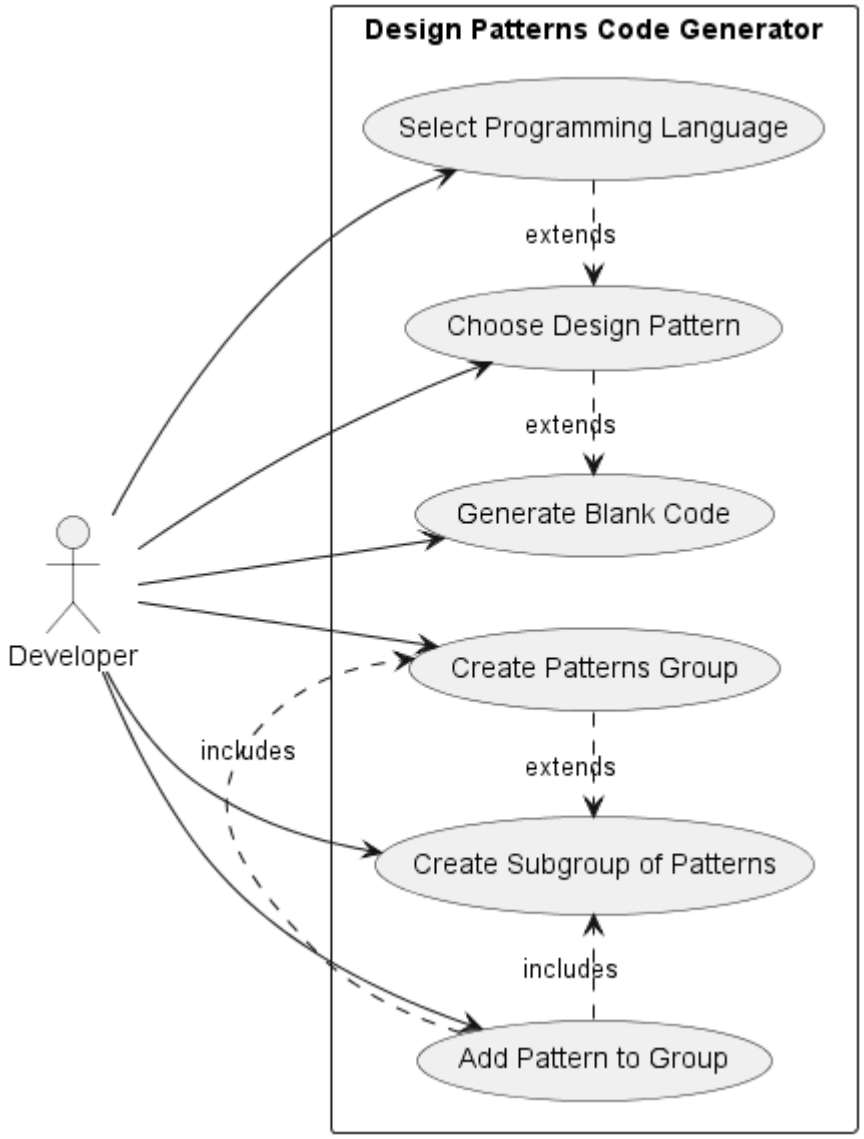

User
username: String
groups: List<Group>
createMainGroup(name: String): Group
createSubgroup(name: String, parentGroupName: String): Group
addRepositoryToGroup(url: String, branch: String, groupName: String): Repository
removeRepositoryFromGroup(repo: Repository, groupName: String): void
1
has
*
Group
name: String
repositories: List<Repository>
subgroups: List<Group>
createGroup(name: String): Group
addRepository(repo: Repository): void
removeRepository(repo: Repository): void
addGroup(group: Group): void
removeGroup(group: Group): void
0..*
contains
0..*
1
contains
*
Repository
url: String
branch: String
clone(): void
fetch(): void
push(): void
pull(): void

Sequence

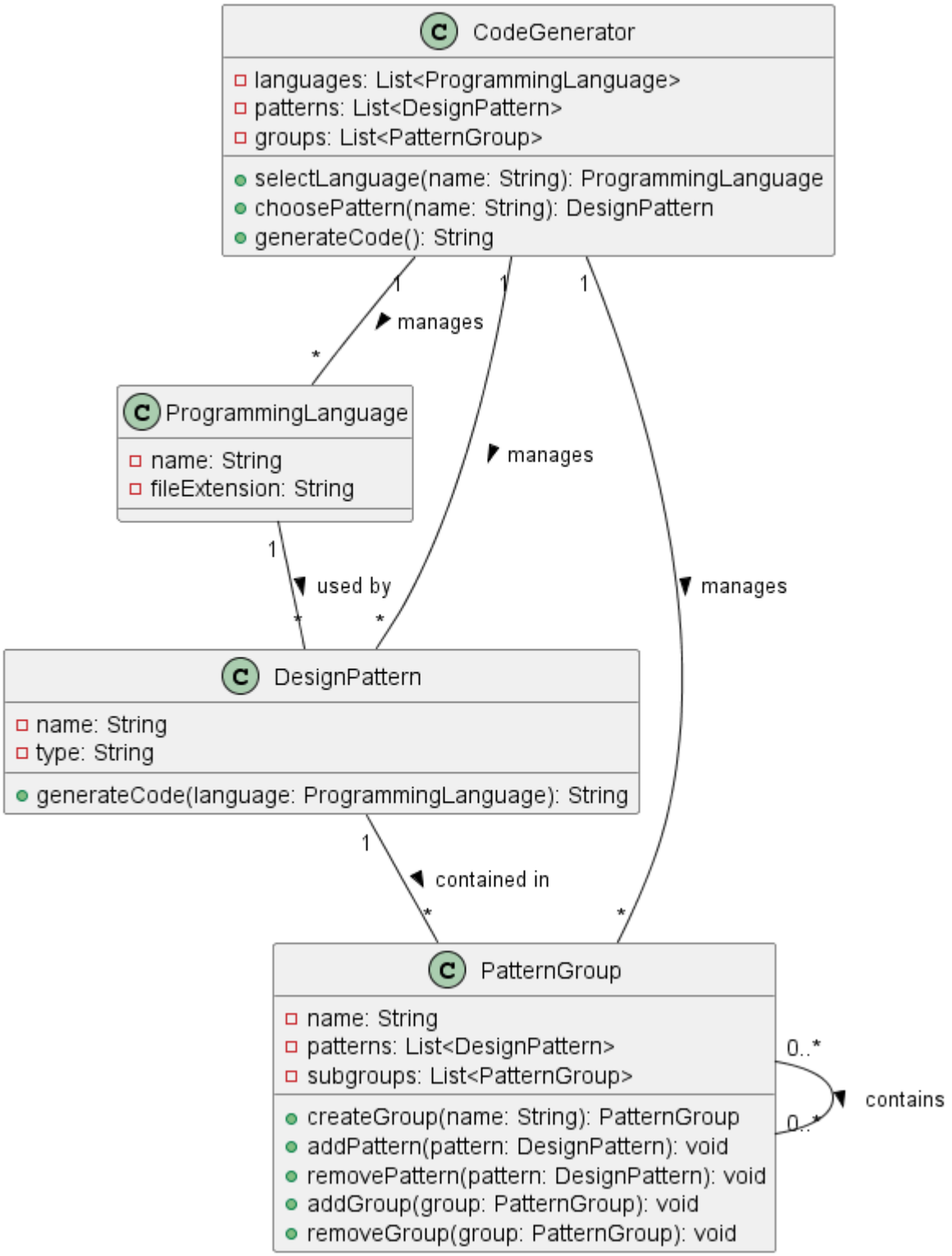

User
Group Manager
Group
Subgroup
Repository
createGroup("Main Group")
new Group("Main Group")
Group created
"Main Group" created
createGroup("Subgroup", "Main Group")
getGroup("Main Group")
new Subgroup("Subgroup")
Subgroup created
Subgroup added to "Main Group"
"Subgroup" created under "Main Group"
addRepositoryToGroup("repo_url", "Subgroup")
getGroup("Subgroup")
new Repository("repo_url")
Repository added to "Subgroup"
Repository added
Repository added to "Subgroup"
User
Group Manager
Group
Subgroup
Repository

Засіб для генерації коду-заготовки для патернів проєктування на заданій мові програмування (можливість створення ієрархії груп для патернів)

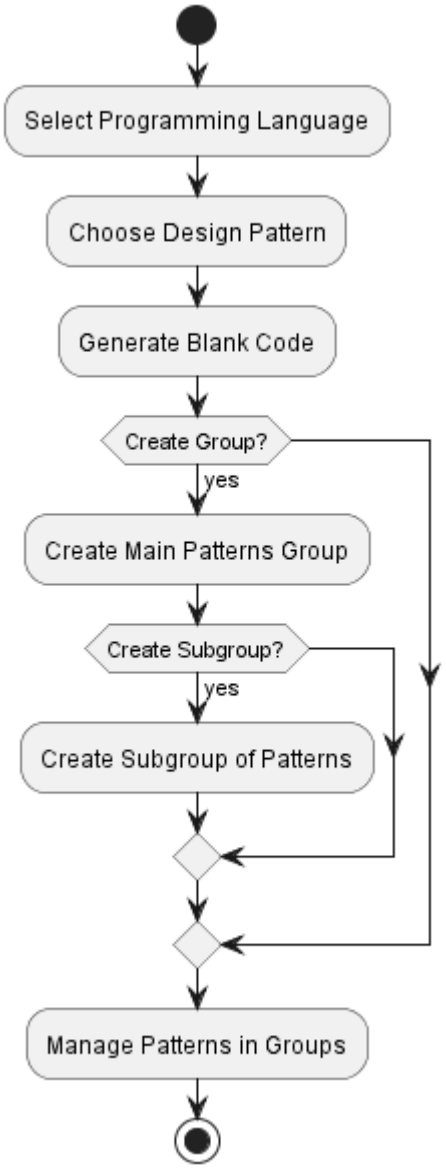
Use case



Class

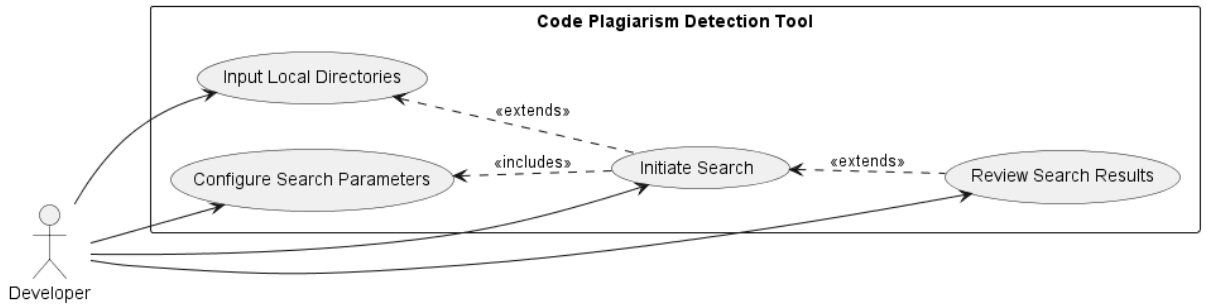

CodeGenerator
languages: List<ProgrammingLanguage>
patterns: List<DesignPattern>
groups: List<PatternGroup>
selectLanguage(name: String): ProgrammingLanguage
choosePattern(name: String): DesignPattern
generateCode(): String
1
manages
*
1
manages
1
manages
ProgrammingLanguage
name: String
fileExtension: String
1
used by
*
*
DesignPattern
name: String
type: String
generateCode(language: ProgrammingLanguage): String
1
contained in
*
*
PatternGroup
name: String
patterns: List<DesignPattern>
subgroups: List<PatternGroup>
createGroup(name: String): PatternGroup
addPattern(pattern: DesignPattern): void
removePattern(pattern: DesignPattern): void
addGroup(group: PatternGroup): void
removeGroup(group: PatternGroup): void
0..*
contains
0..*

Activity

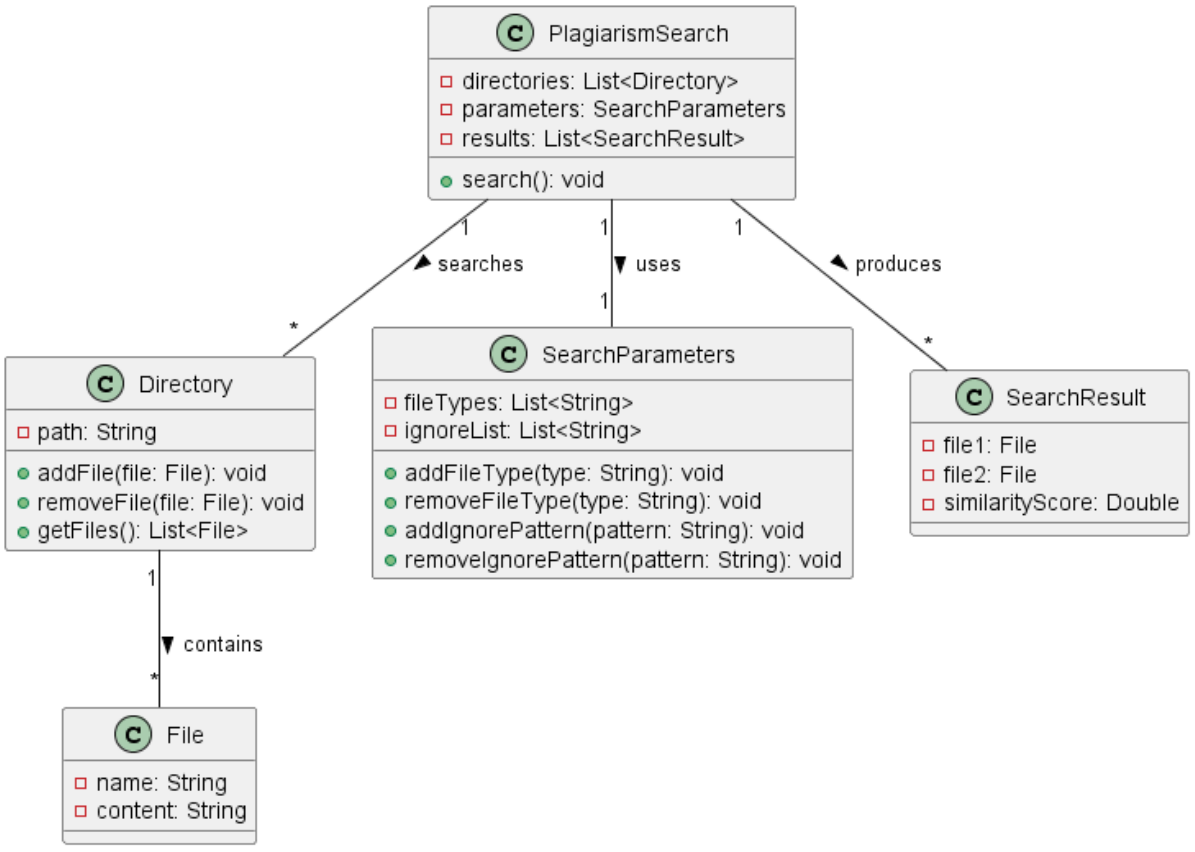

Select Programming Language
Choose Design Pattern
Generate Blank Code
Create Group?
yes
Create Main Patterns Group
Create Subgroup?
yes
Create Subgroup of Patterns
Manage Patterns in Groups

Засіб для пошуку копій програмного коду (плагіату) (можливість пошуку по переліку локальних каталогів)

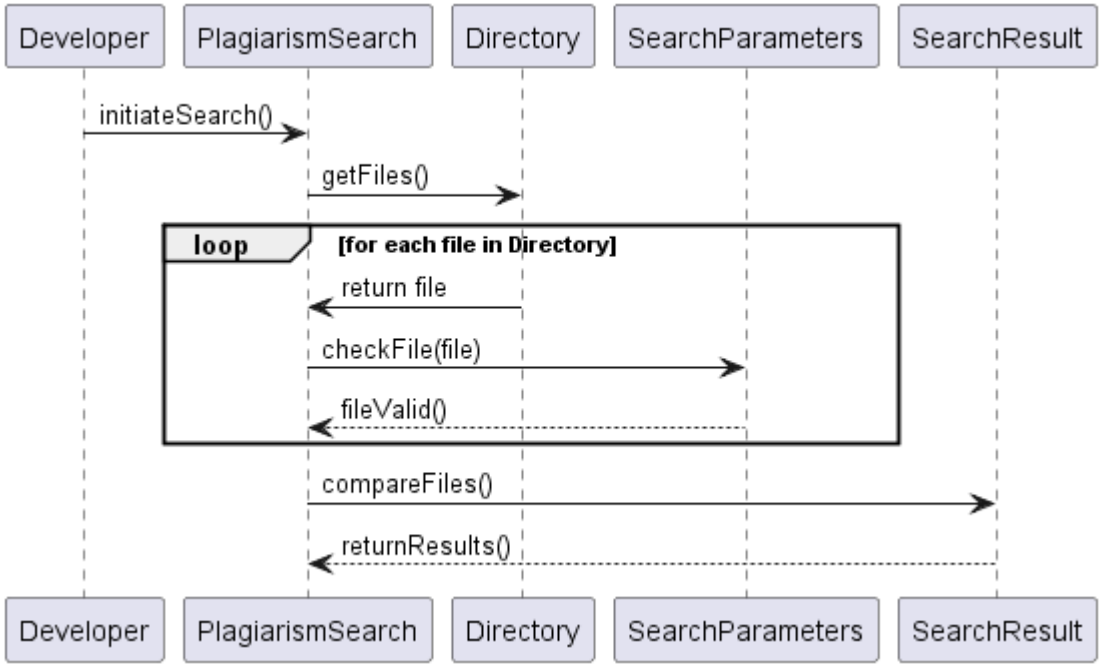
Use case



Class

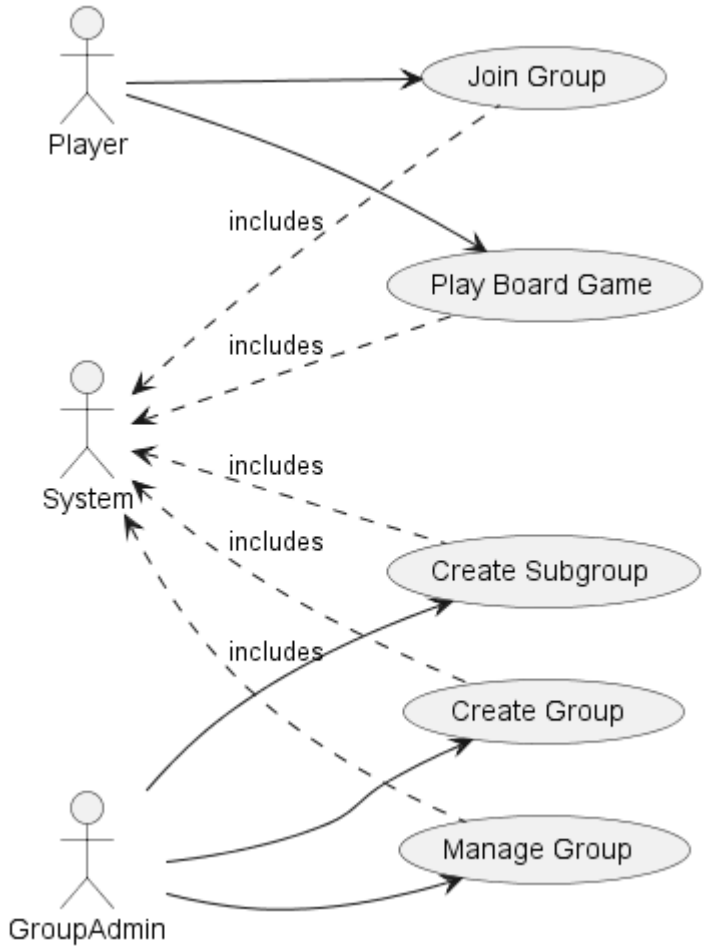


Communication

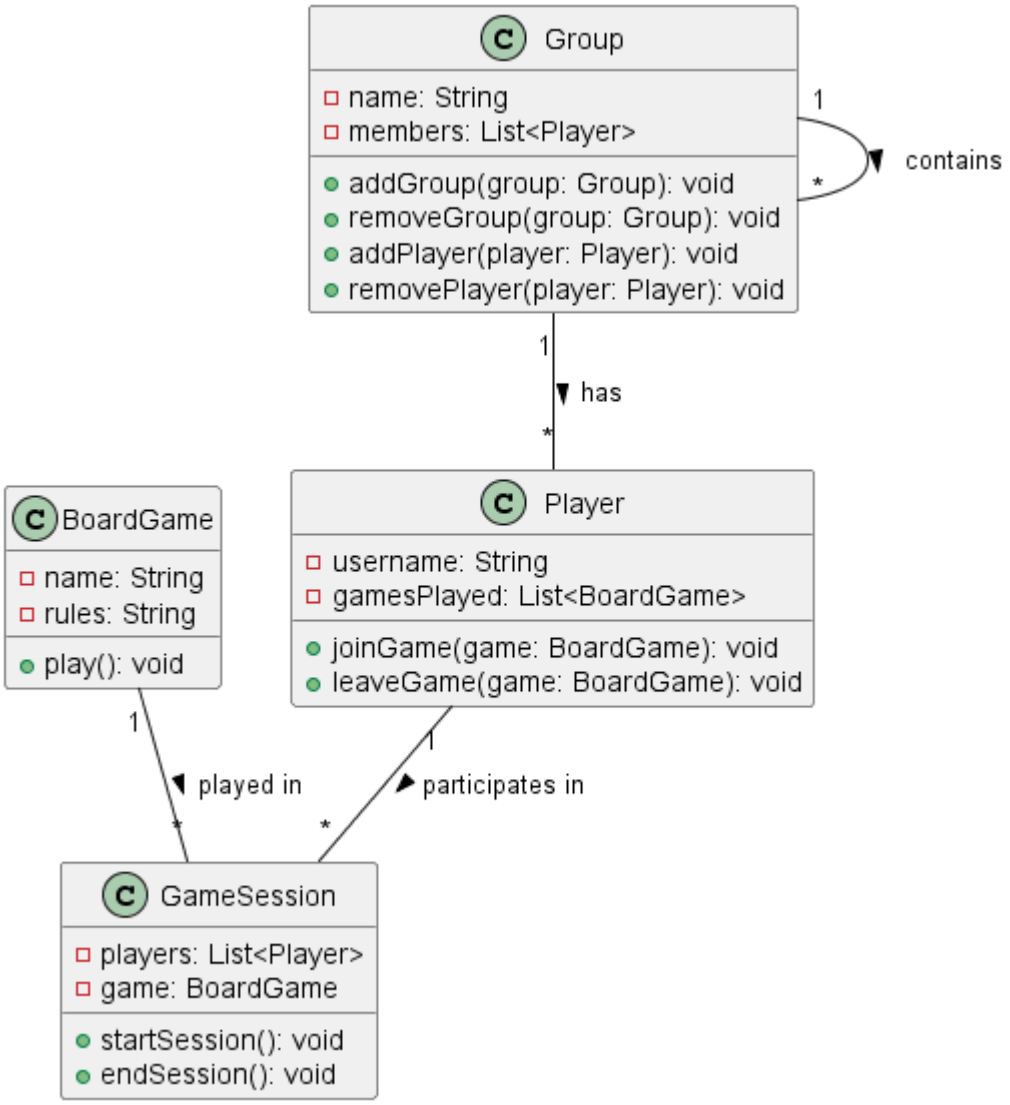


Універсальну платформу для підтримки різних настільних ігор (можливість створення ієрархії груп для ігор)

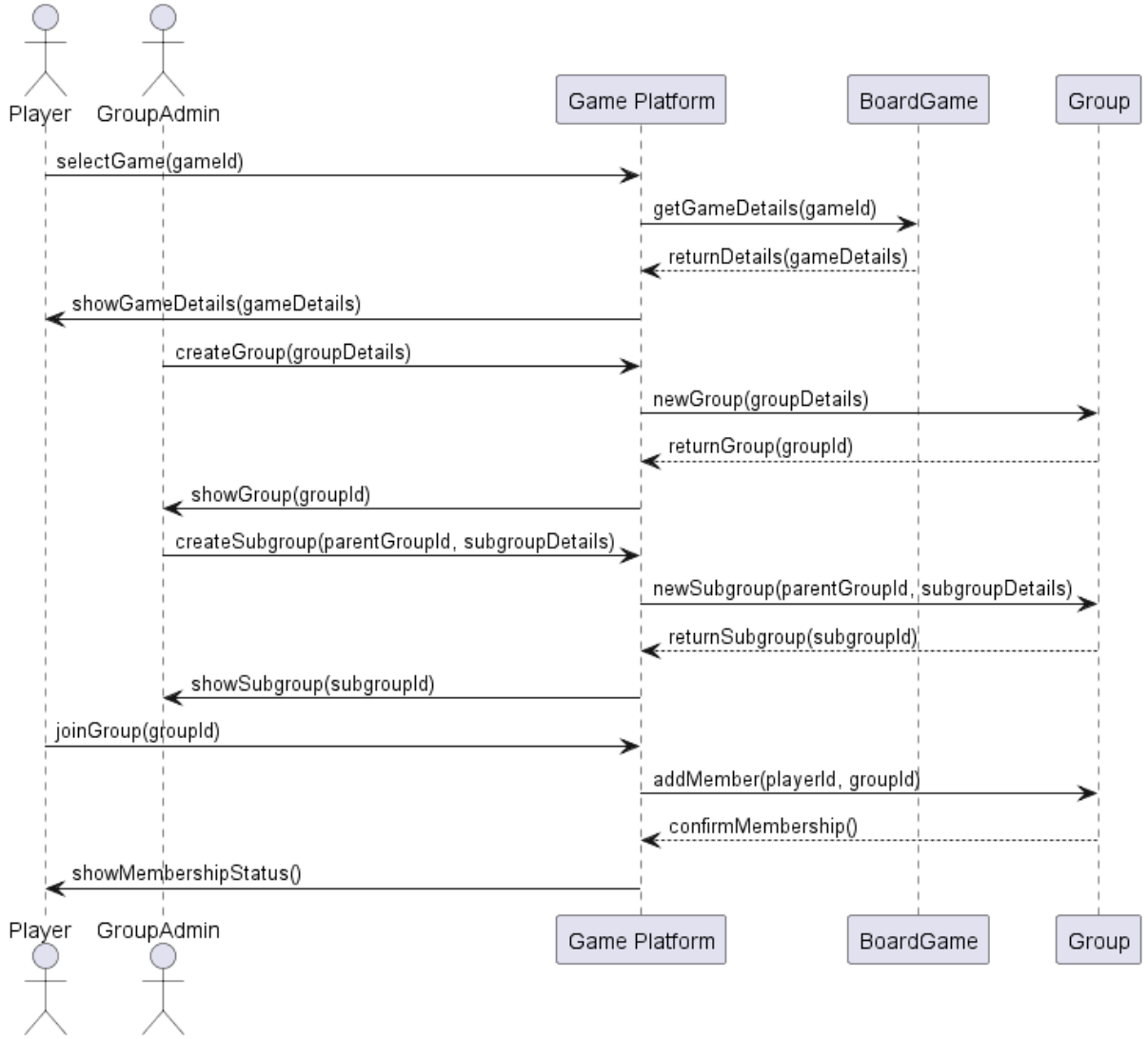
Use case



Class

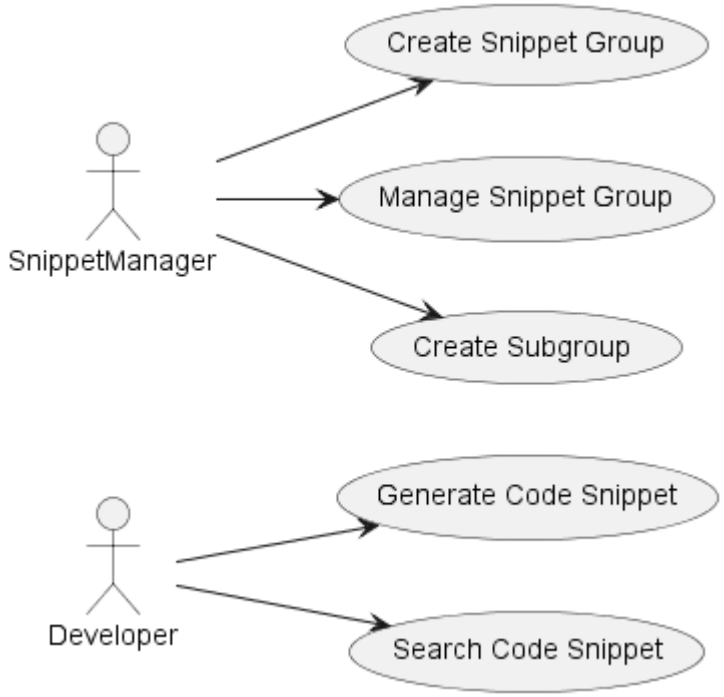

Group
name: String
members: List<Player>
addGroup(group: Group): void
removeGroup(group: Group): void
addPlayer(player: Player): void
removePlayer(player: Player): void
1
*
contains
1
has
*
BoardGame
name: String
rules: String
play(): void
Player
username: String
gamesPlayed: List<BoardGame>
joinGame(game: BoardGame): void
leaveGame(game: BoardGame): void
1
played in
*
1
participates in
*
GameSession
players: List<Player>
game: BoardGame
startSession(): void
endSession(): void

Sequence

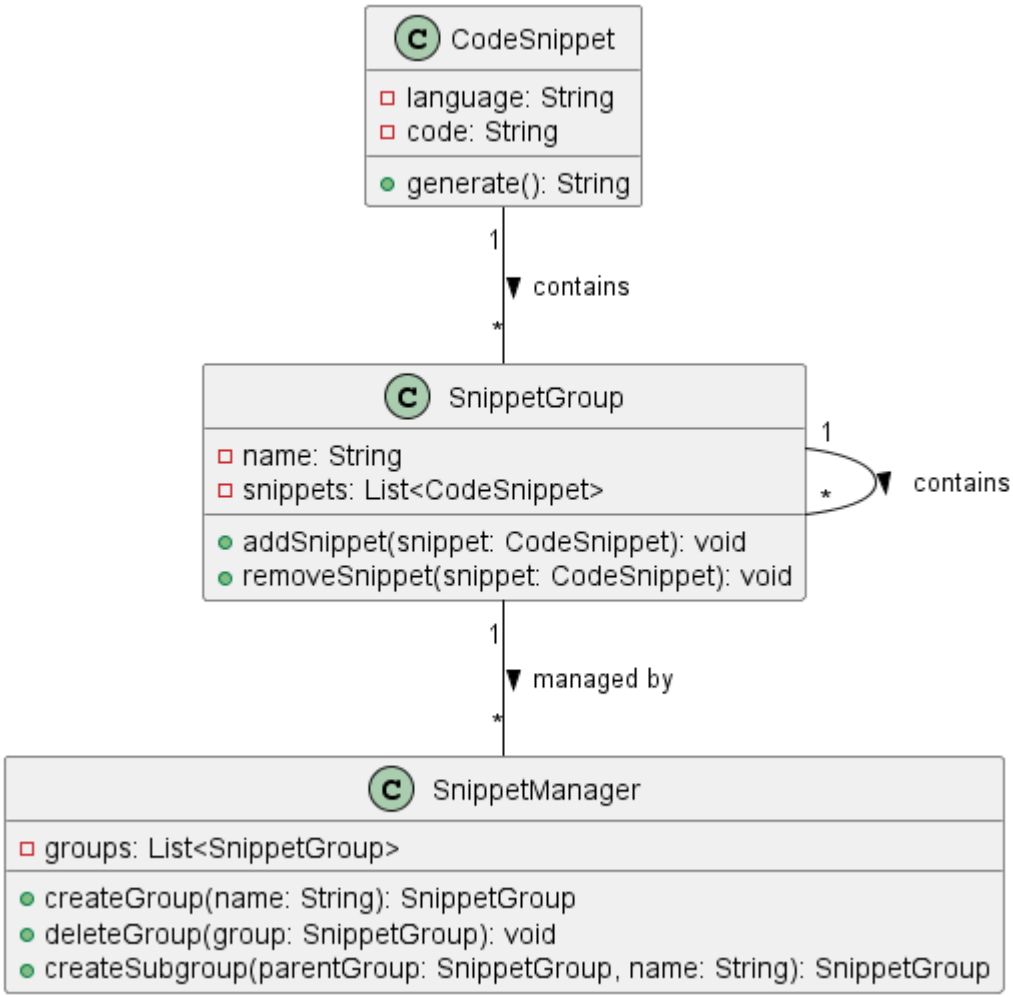

Player
GroupAdmin
Game Platform
BoardGame
Group
selectGame(gameId)
getGameDetails(gameId)
returnDetails(gameDetails)
showGameDetails(gameDetails)
createGroup(groupDetails)
newGroup(groupDetails)
returnGroup(groupId)
showGroup(groupId)
createSubgroup(parentGroupId, subgroupDetails)
newSubgroup(parentGroupId, subgroupDetails)
returnSubgroup(subgroupId)
showSubgroup(subgroupId)
joinGroup(groupId)
addMember(playerId, groupId)
confirmMembership()
showMembershipStatus()
Player
GroupAdmin
Game Platform
BoardGame
Group

Засіб для генерації стандартних фрагментів коду (code snippets) на різних мовах програмування (можливість створення ієрархії груп для фрагментів коду)

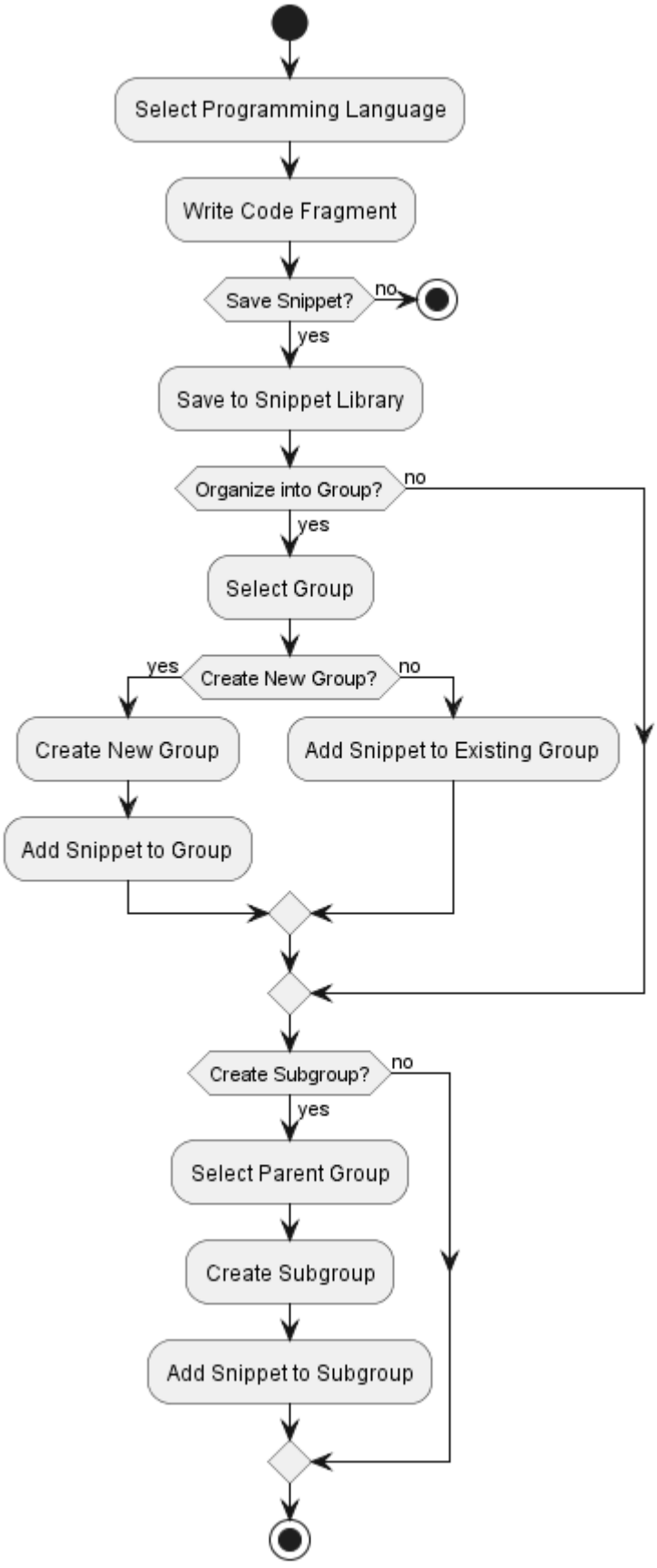
Use case



Class

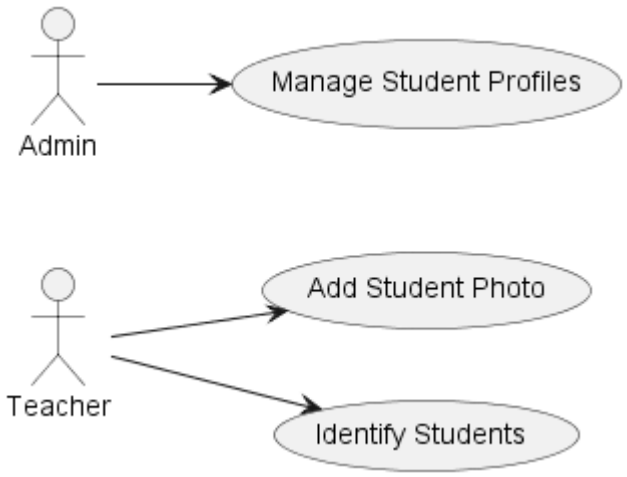

CodeSnippet
language: String
code: String
generate(): String
1
contains
*
SnippetGroup
name: String
snippets: List<CodeSnippet>
addSnippet(snippet: CodeSnippet): void
removeSnippet(snippet: CodeSnippet): void
1
*
contains
1
managed by
*
SnippetManager
groups: List<SnippetGroup>
createGroup(name: String): SnippetGroup
deleteGroup(group: SnippetGroup): void
createSubgroup(parentGroup: SnippetGroup, name: String): SnippetGroup

Activity

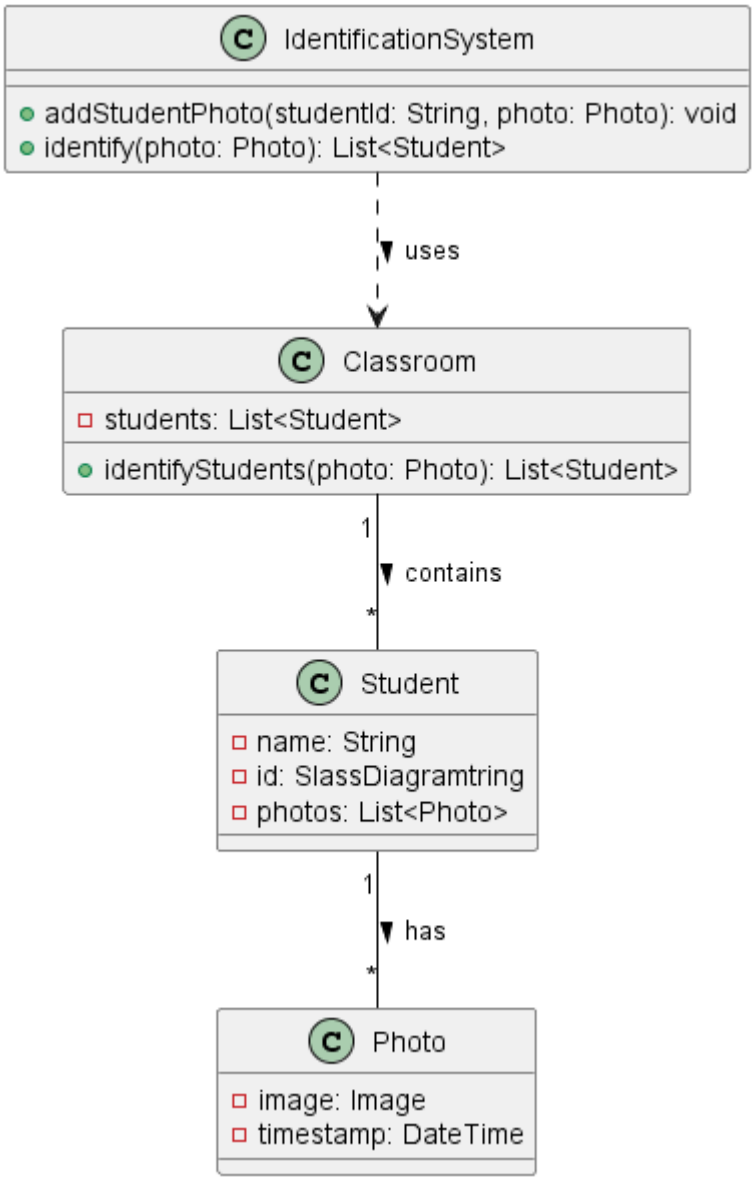

Select Programming Language
Write Code Fragment
Save Snippet?
no
yes
Save to Snippet Library
Organize into Group?
no
yes
Select Group
yes
Create New Group?
no
Create New Group
Add Snippet to Existing Group
Add Snippet to Group
Create Subgroup?
no
yes
Select Parent Group
Create Subgroup
Add Snippet to Subgroup

Засіб для ідентифікації студентів в аудиторії (наприклад, відвідування лекцій) за фотографією аудиторії (можливість додавання декількох фотографій одного студента)

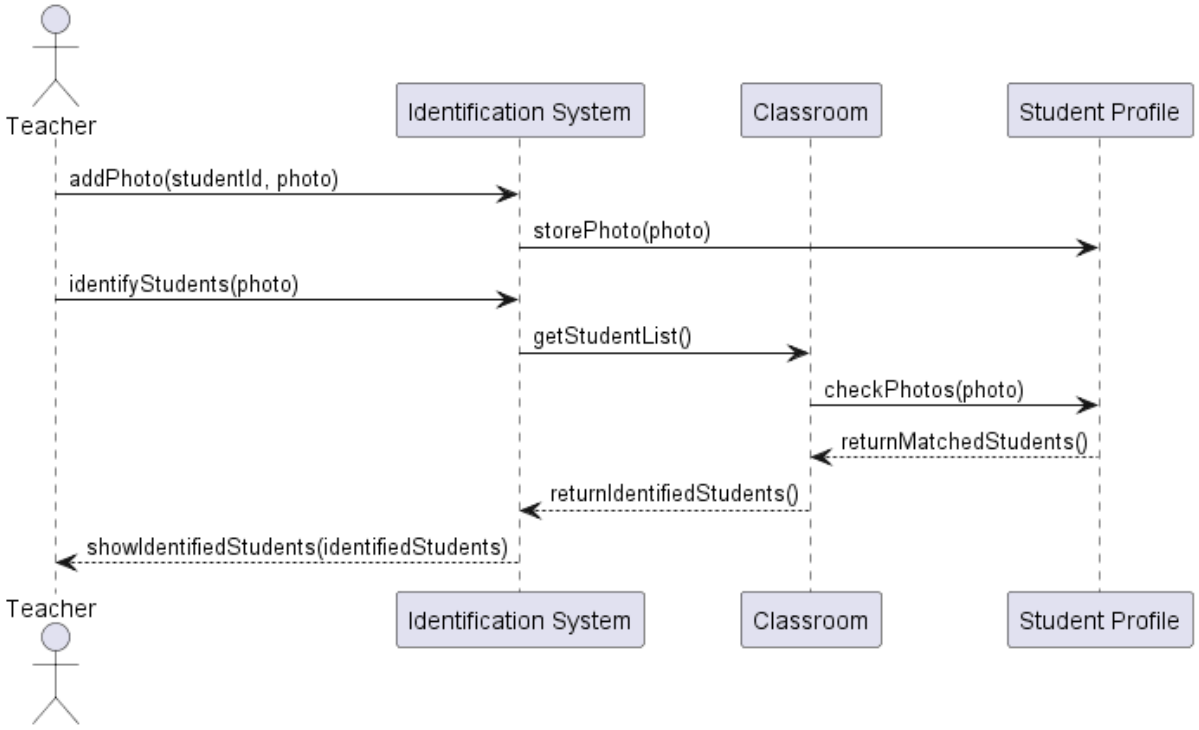
Use case



Class

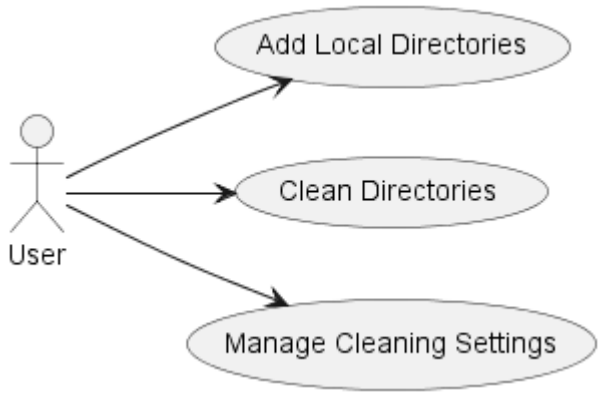

IdentificationSystem
addStudentPhoto(studentId: String, photo: Photo): void
identify(photo: Photo): List<Student>
uses
Classroom
students: List<Student>
identifyStudents(photo: Photo): List<Student>
1
contains
*
Student
name: String
id: SlassDiagramtring
photos: List<Photo>
1
has
*
Photo
image: Image
timestamp: DateTime

Communication

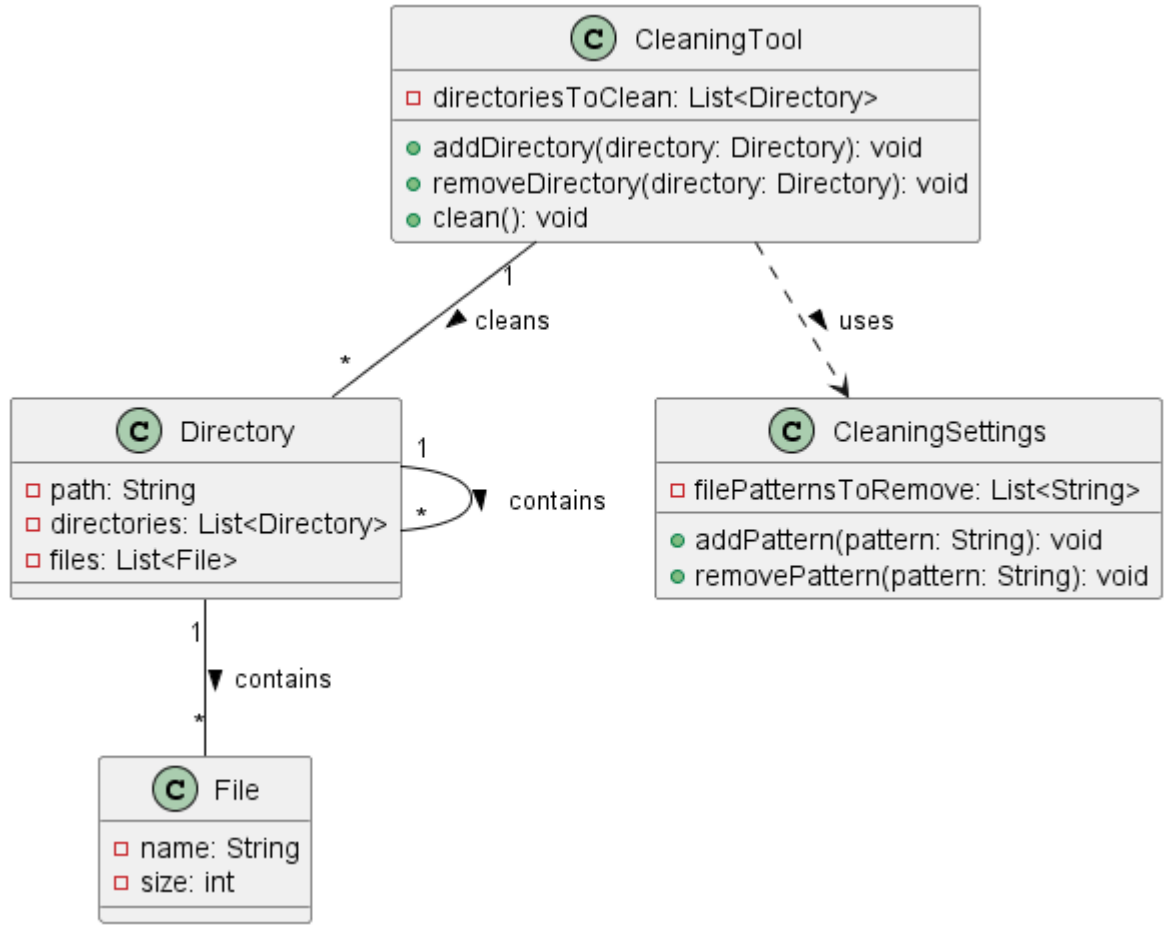


Засіб для видалення непотрібних файлів з ієрархії каталогів (наприклад, чистка коду від згенерованих артефактів) (можливість додавання переліку локальних каталогів для обробки)

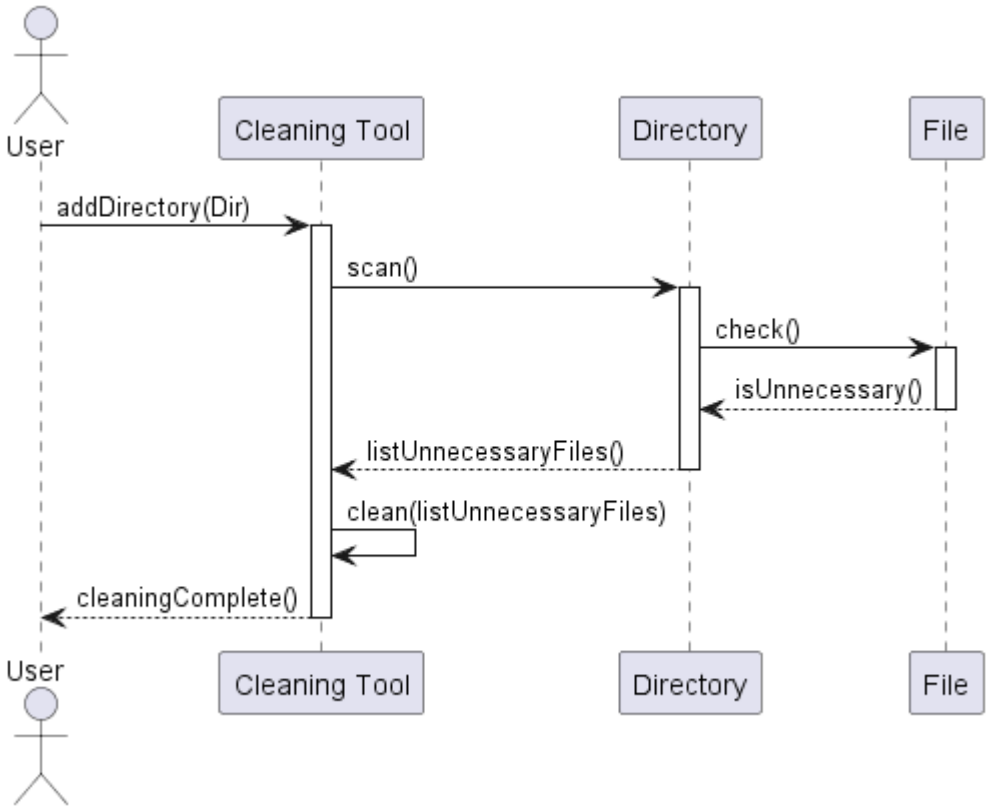
Use case



Class

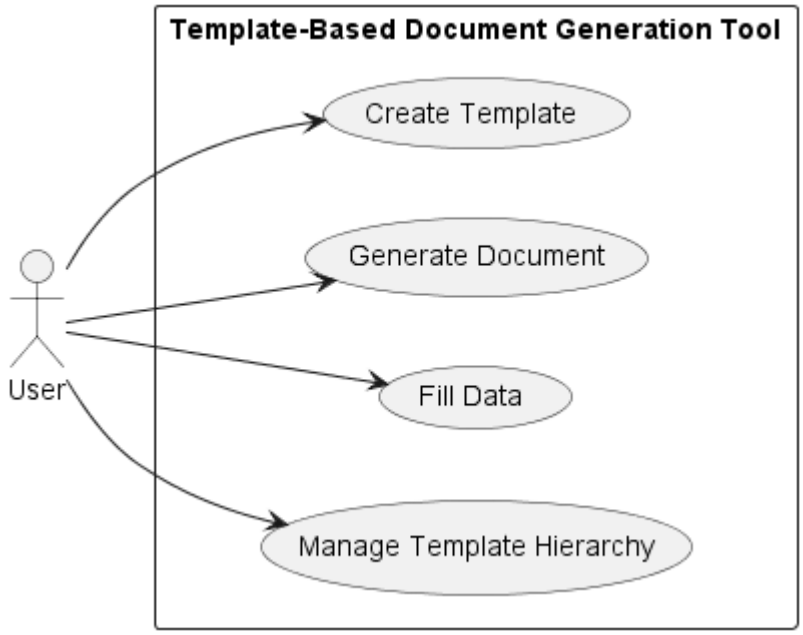


Sequence

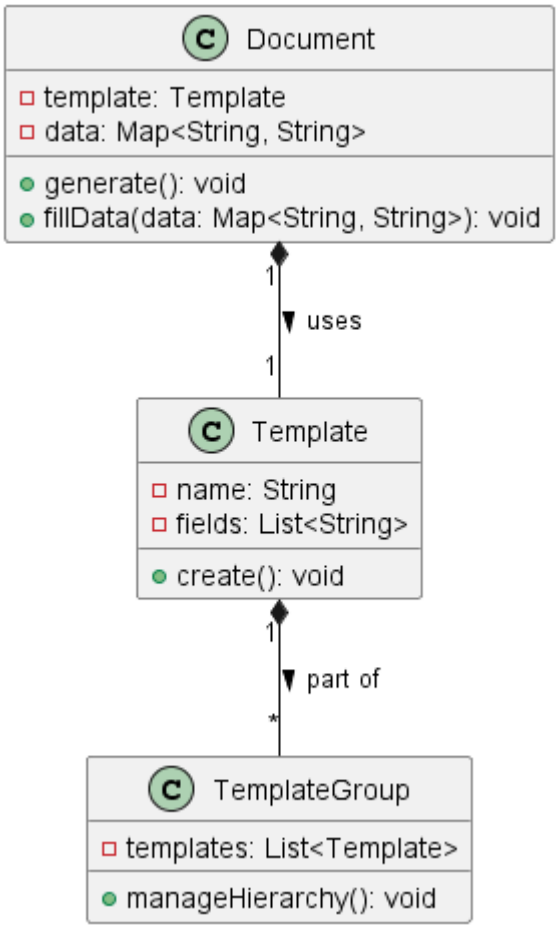

User
Cleaning Tool
Directory
File
addDirectory(Dir)
scan()
check()
isUnnecessary()
listUnnecessaryFiles()
clean(listUnnecessaryFiles)
cleaningComplete()
User
Cleaning Tool
Directory
File

Засіб для генерації документів на основі шаблону та заповнення даних (наприклад, білети до іспиту) (можливість створення ієрархії груп для шаблонів)

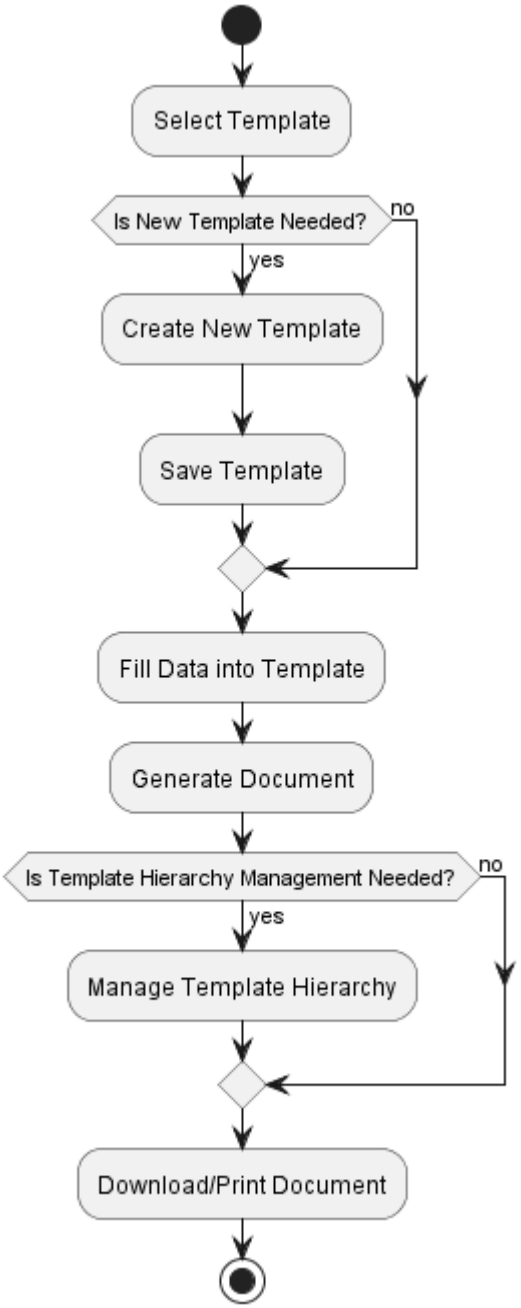
Use case



Class

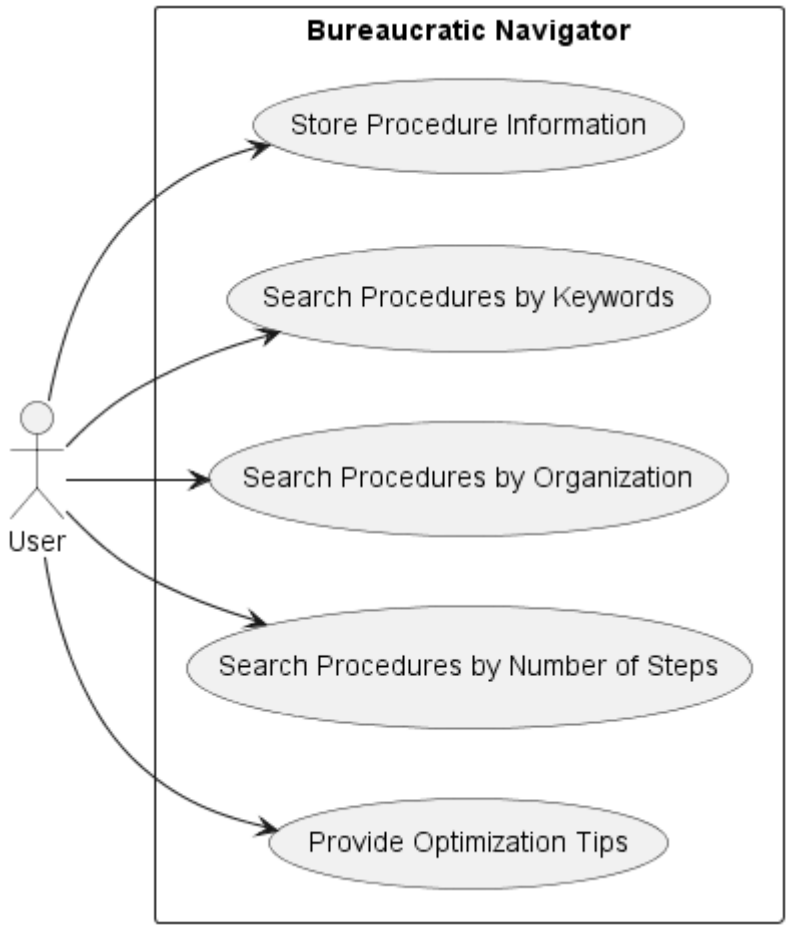

Document
template: Template
data: Map<String, String>
generate(): void
fillData(data: Map<String, String>): void
1
uses
1
Template
name: String
fields: List<String>
create(): void
1
part of
*
TemplateGroup
templates: List<Template>
manageHierarchy(): void

Activity

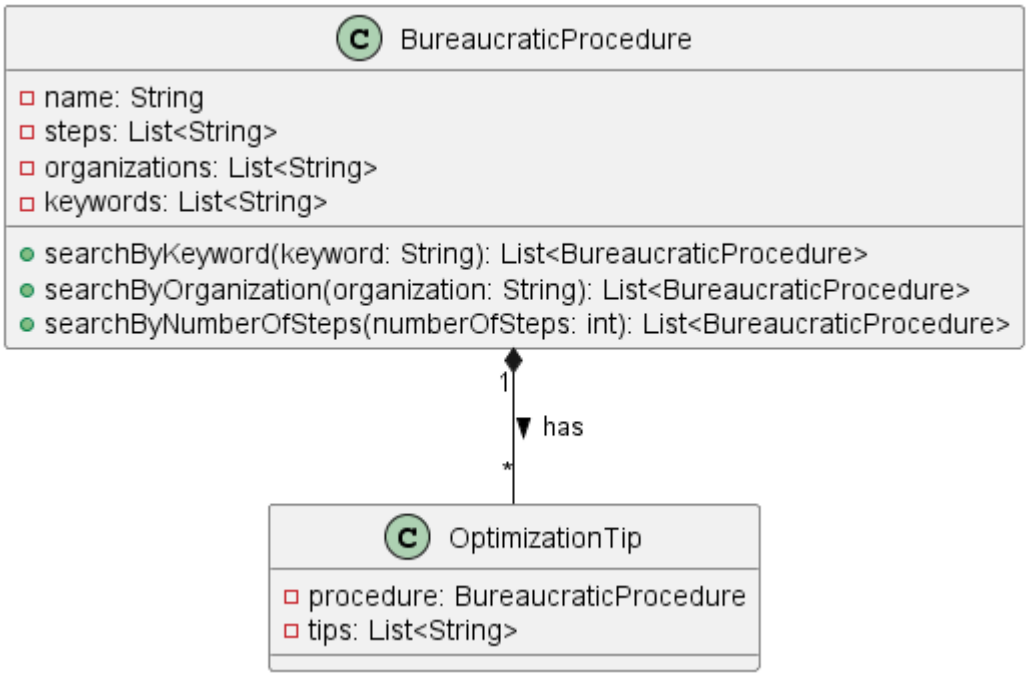

Select Template
Is New Template Needed?
no
yes
Create New Template
Save Template
Fill Data into Template
Generate Document
Is Template Hierarchy Management Needed?
no
yes
Manage Template Hierarchy
Download/Print Document

Бюрократичний навігатор – засіб для зберігання інформації про бюрократичні процедури, підказки щодо оптимального їх проходження (можливість пошуку бюрократичних процедур за різними параметрами – ключовими словами, організаціями куди треба звертатись, кількістю кроків тощо)

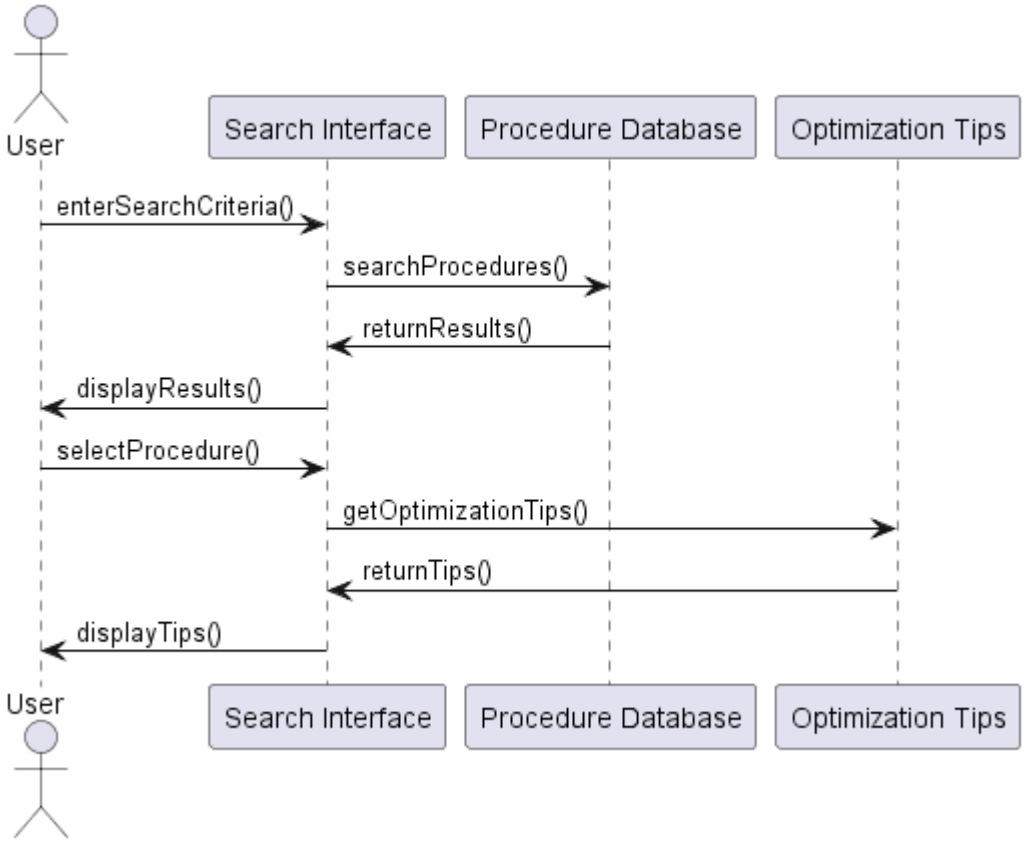
Use case



Class

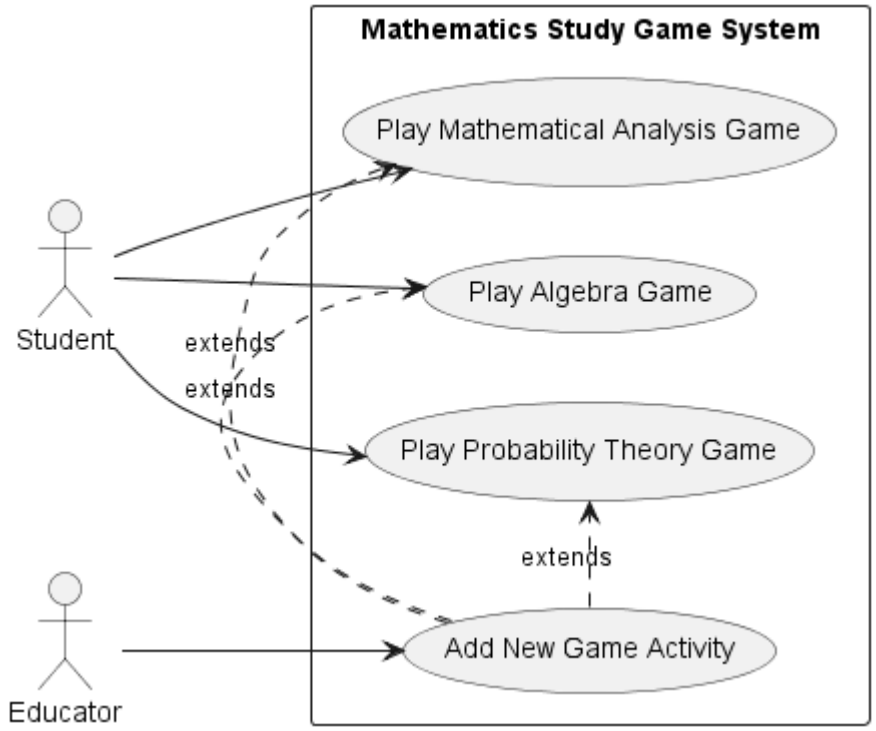


Communication

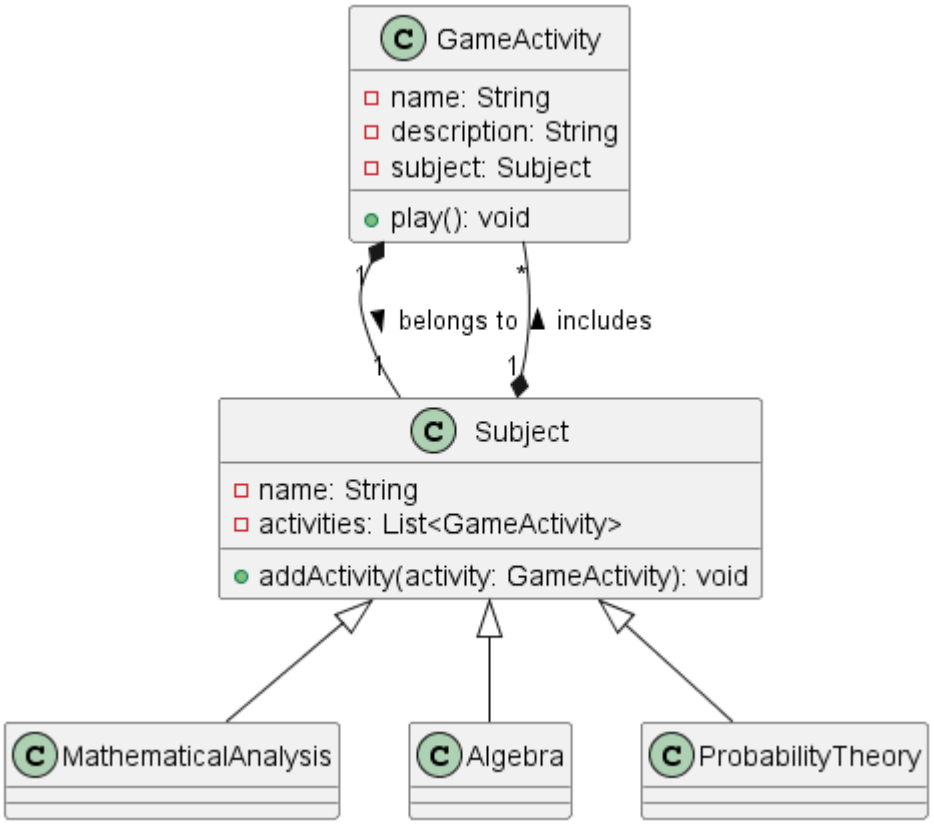


Систему для вивчення університетської математики (мат. аналіз, алгебра, теорія ймовірностей, ...) в ігровій формі (можливість додавання нових ігрових активностей, зокрема одразу до декількох предметів)

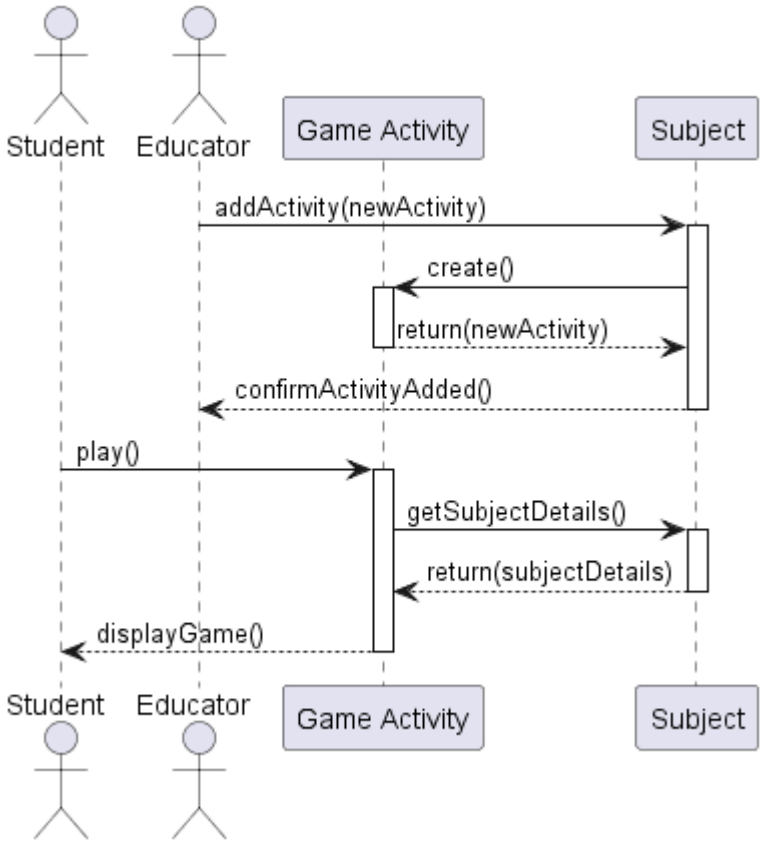
Use case



Class

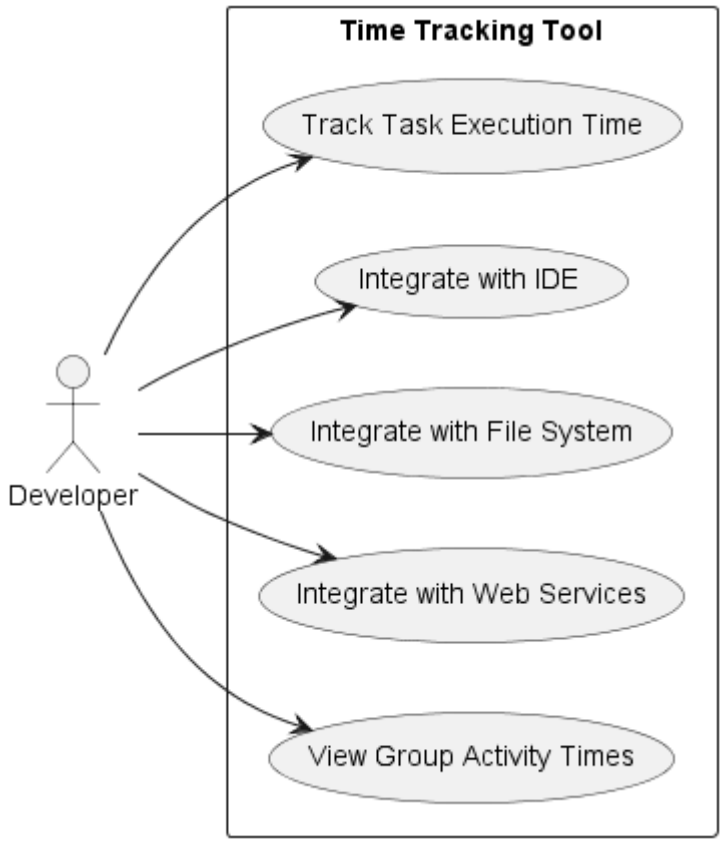

GameActivity
name: String
description: String
subject: Subject
play(): void
1
*
belongs to
includes
1
1
Subject
name: String
activities: List<GameActivity>
addActivity(activity: GameActivity): void
MathematicalAnalysis
Algebra
ProbabilityTheory

Sequence

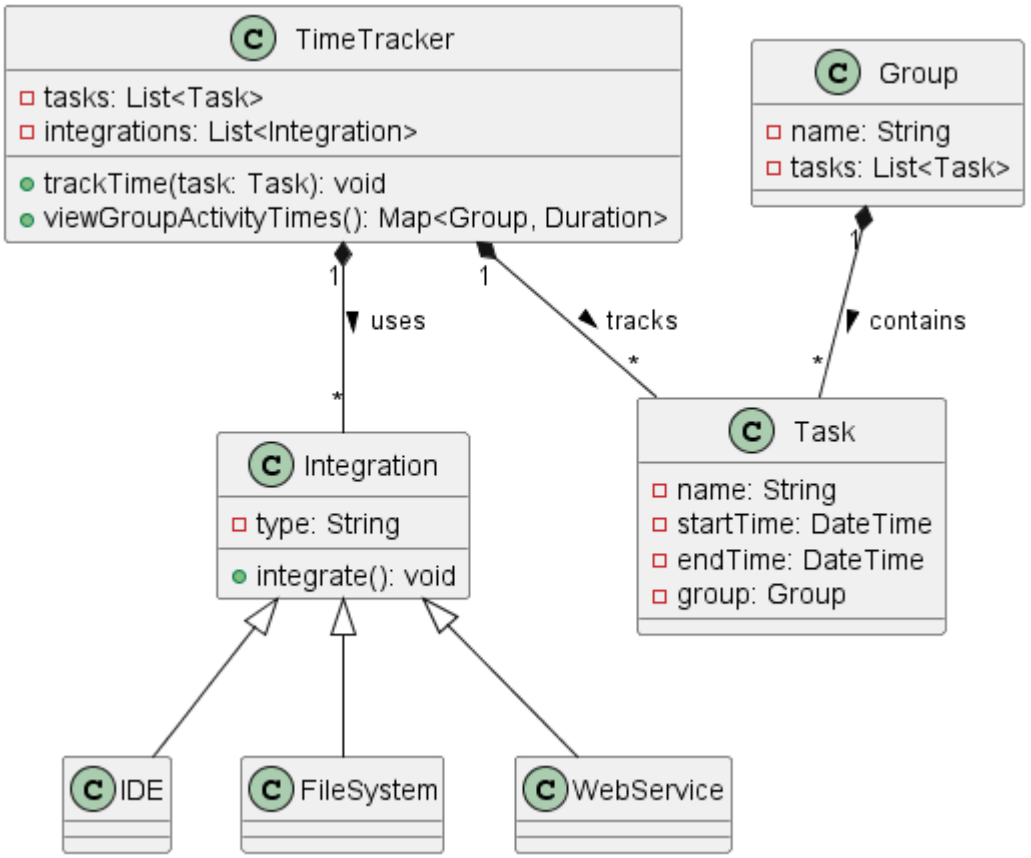

Student
Educator
Game Activity
Subject
addActivity(newActivity)
create()
return(newActivity)
confirmActivityAdded()
play()
getSubjectDetails()
return(subjectDetails)
displayGame()
Student
Educator
Game Activity
Subject

Засіб для time tracking (відслідковування часу виконання різних завдань розробником) з інтеграцією із зовнішніми системами (IDE, файлова система, веб сервіси, ...) (можливість перегляду часу виконання різних груп активностей)

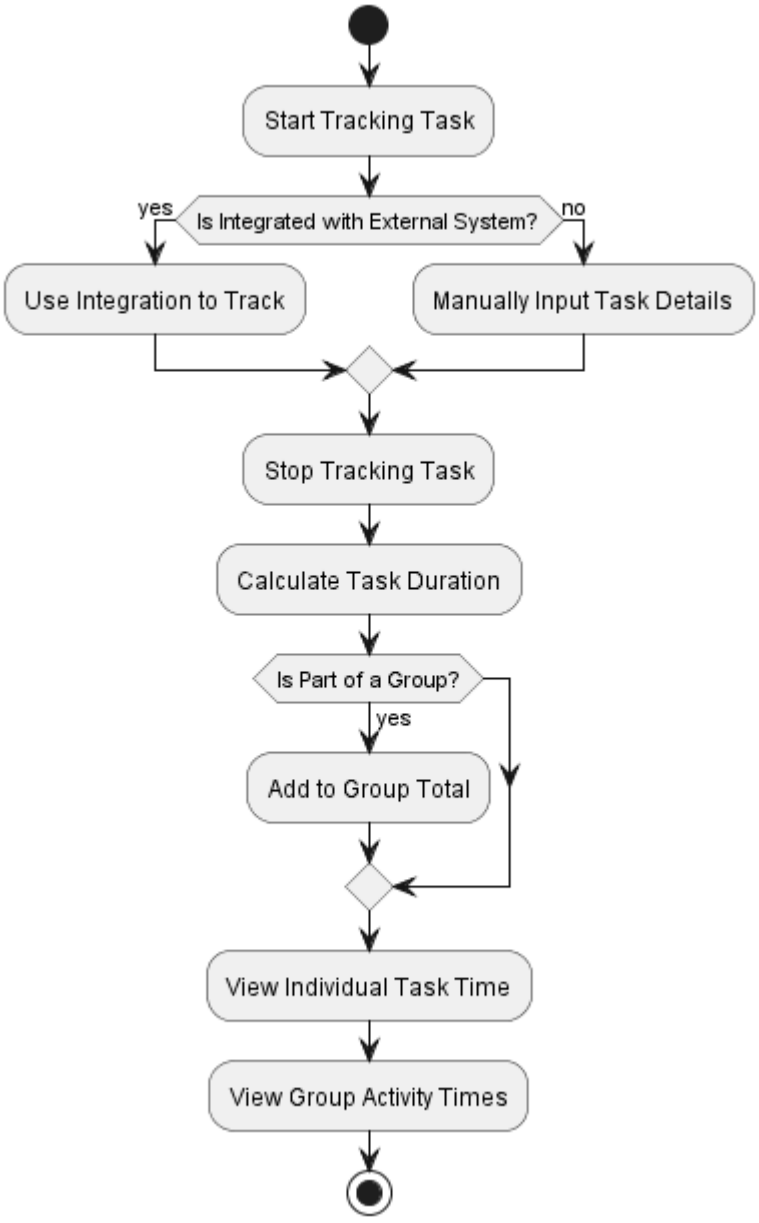
Use case



Class


TimeTracker
tasks: List<Task>
integrations: List<Integration>
trackTime(task: Task): void
viewGroupActivityTimes(): Map<Group, Duration>
Group
name: String
tasks: List<Task>
uses
tracks
contains
1
1
1
*
*
*
Integration
type: String
integrate(): void
Task
name: String
startTime: DateTime
endTime: DateTime
group: Group
IDE
FileSystem
WebService

Activity


Start Tracking Task
Is Integrated with External System?
yes
no
Use Integration to Track
Manually Input Task Details
Stop Tracking Task
Calculate Task Duration
Is Part of a Group?
yes
Add to Group Total
View Individual Task Time
View Group Activity Times