Екзаменаційна робота

З основ об’єктно орієнтованого програмування

Студента групи ІПС-21

Факультету комп’ютерних наук та кібернетики

Ольховатого Ігоря

**Екзаменаційний білет № 50**

1. Принципи розвитку та DIP. Обов’язково навести приклади з власного коду.
2. Порівняти патерни Composite, Interpreter, Visitor. Приклади з власного коду та/або питання 3.
3. Спроектувати (наприклад, використовуючи UML) засіб для ідентифікації студентів в аудиторії (наприклад, відвідування лекцій) за фотографією аудиторії.
4. Принцип розвитку.

Оскільки вимоги можуть до продукту і відповідного програмного коду можуть часто змінюватись в силу нових потреб замовника, змін на ринку чи технічного прогресу, то будь-яка програмна система повинна підтримувати можливість розвитку, для того аби швидко і ефективно адаптуватися до цих змін з мінімальним затратами людських ресурсів і часу. Серед змін часто виникають потреби внесення змін до існуючого коду (знову ж, з мінімальними затратами), додавання нової функціональності або ж створення системи плагінів (якщо проєкт достатньо великий і вирішує складну задачу). Варто також зауважити, що принцип YAGNI (You Ain’t Gonna Need It) рекомендує не додавати ту функціональність, яка не використовується на даний момент, аби не ускладнювати код.

Приклад використання принципу розвутку:

Інтерфейс замість конкретної реалізації.

У проєкті (приватний репозиторій, посилання не надається) пов’язаному з використанням комп’ютерного зору є частина коду, що відповідає за аугментацію даних для фотографій (аугментація даних – техніка у машинному навчанні, що певним чином видозміняє дані, аби змінити їх розподіл, таким чином можна отримати нові дані за допомогою існуючих і збільшити тренувальний набір).

Можна було б просто використовувати конкретну реалізацію аугментацій з бібліотеки albumentations, проте таких підхід не дає можливостей розвитку. Тож вирішено зробити інтерфейс для аугментацій, від якого будуть успадковуватися класи для аугментації за допомогою albumentations та будь-які інші класи з кастомними аугментаціями, яких немає у цій бібліотеці. Наприклад, додавання ефекту диму до фотографій.

class Augmentation:  
 *"""  
 Interface for augmentations  
 """* @abc.abstractmethod  
 def \_\_call\_\_(self, sample: Sample) -> Sample:  
 pass

class AlbumentationTransform(Augmentation):  
 *"""  
 Adapter (wrapper) for albumentations augmentations.  
 """* def \_\_init\_\_(self, transform: callable):  
 *"""  
 Can be a separate augmentation or Albumentations.Compose* ***:param*** *transform:  
 """* self.transform = transform  
  
 def \_\_call\_\_(self, sample: Sample) -> Sample:  
 # Some code  
 return transformed\_sample

Варто зауважити, що клас AlbumentationTransform також працює як адаптер, адже ми працюємо з власною стуктурою, яка зберігає додаткову інформацію, окрім фотографій, а методи бібліотеки albumentations працюють з тензорами для зображень.

class Smoke(Augmentation):  
 *"""  
 Adds smoke effect to a given image.  
 """*  
 def \_\_init\_\_(self, …):

def \_\_call\_\_(self, sample: Sample) -> Sample:

class Augmentor:

@measure\_time  
def apply2folder(self,  
 augmentations: list[Augmentation],  
 source\_directory: str,  
 export\_directory: str,  
 name\_postfix: Union[str, None]) -> None:

Бачимо, що клас Augmentor працює приймає просто інтерфейс, від якого успадковуються потрібні аугментації. Це допоможе швидко та зручно додавати нові аугментації за потреби, якщо розподіл даних для тренування треба буде сильно змінити.

Принцип DIP (Dependency inversion principle)

Принцип інверсії залежностей в об'єктно-орієнтованому проектуванні - це специфічна методологія для слабко пов'язаних програмних модулів. Цей принцип полягає в тому, що залежності між високорівневими модулями, які визначають політику, та низькорівневими модулями, які залежать від них, змінюються на протилежні. Це дозволяє зробити високорівневі модулі незалежними від деталей реалізації низькорівневих модулів.

Можна виділити два основні постулати цього принципу:

1. Клієнти (високорівневі модулі) не залежать від сервісів (низькорівневих модулів), але вони всі залежать від інтерфейсів (абстракцій).
2. Абстракції не залежать від деталей(конкретних реалізацій), але деталі залежать від абстракцій.

class Tracker:  
 def update(self, bbox):  
 raise NotImplementedError  
  
  
class OpticalFlowTracker(Tracker):  
 def update(self, bbox):  
 # Implementation of updating bbox  
  
class TrackerAlgorithm:  
 def \_\_init\_\_(self, model, tracker):  
 self.model = model  
 self.tracker = tracker  
  
 def track(self, video):  
 frames = []  
 # get frames from video  
 for frame in frames:  
 bboxes = self.model.predict(frame)   
 updated\_bboxes = [self.tracker.update(bbox) for bbox in bboxes]  
 # use updated bboxes  
  
  
optical\_flow = OpticalFlowTracker()  
model = SomeCoolNeuralNetwork()  
algorithm = TrackerAlgorithm(model, tracker)  
algorithm.track()

Клас Tracker - це абстрактний клас, який визначає спільний інтерфейс для всіх трекерів. У ньому оголошено єдиний метод update, який представляє поведінку оновлення обмежувальної рамки (bbox).

Клас OpticalFlowTracker є конкретною реалізацією класу Tracker. Він забезпечує специфічну реалізацію методу update з використанням методів оптичного потоку.

Клас TrackerAlgorithm представляє алгоритм, який виконує відстеження за допомогою моделі та трекера. Він отримує екземпляри моделі та трекера як залежності у своєму конструкторі.

Метод track всередині TrackerAlgorithm отримує кадри з відео і застосовує модель для прогнозування обмежувальних рамок для кожного кадру. Потім він використовує об'єкт трекера, щоб оновити кожну передбачувану область, в результаті чого створюються оновлені області. Ці оновлені рамки можуть бути використані в подальшому за потреби.

Дотримуючись DIP, код досягає декількох переваг:

Клас TrackerAlgorithm залежить від абстракцій (Tracker і модель), а не від конкретних реалізацій. Це зменшує зв'язок між класами, роблячи їх більш незалежними і легшими для модифікації або розширення в майбутньому.

Клас TrackerAlgorithm не прив'язаний до конкретного алгоритму відстеження (наприклад, OpticalFlowTracker). Він може працювати з будь-яким трекером, який дотримується інтерфейсу Tracker, що забезпечує гнучкість і дозволяє легко замінювати або додавати різні трекери без зміни алгоритму.

Клас TrackerAlgorithm не залежить від конкретної реалізації моделі. Він може працювати з будь-якою моделлю, яка відповідає інтерфейсу моделі, що дозволяє використовувати різні моделі машинного навчання без зміни алгоритму.

2.

Composite (компонувальник)

Компонувальник — це структурний патерн проектування, що дає змогу згрупувати декілька об’єктів у деревоподібну структуру, а потім працювати з нею так, ніби це одиничний об’єкт.

У проєкті (приватний репозиторій) де потрібно багато працювати з даними, що зберігаються на диску у вигляді файлів у папках, а не оперативній пом’яті (в останньому випадку можна було б тоді просто використати вже готові класи для представлення датасету, наприклад torch.Dataset) розроблено такі класи для зручних манімпуляцій з даними, використовуючи компонувальник:

from abc import ABC, abstractmethod  
  
  
class DataComponent(ABC):  
 *"""  
 Базовий абстрактний клас, що визначає загальний інтерфейс для всіх компонентів даних.  
 """* @abstractmethod  
 def process(self) -> None:  
 *"""  
 Метод для обробки компонента даних.  
 """* pass  
  
  
class Dataset(DataComponent):  
 *"""  
 Клас, що представляє набір даних.  
 """* def \_\_init\_\_(self, name: str):  
 self.name = name  
  
 def process(self) -> None:  
 *"""  
 Обробка набору даних.  
 """* print(f"Processing dataset: {self.name}")  
  
  
class DataFolder(DataComponent):  
 *"""  
 Клас, що представляє папку з даними, що містить інші компоненти даних.  
 """* def \_\_init\_\_(self, name: str):  
 self.name = name  
 self.components = []  
  
 def add\_component(self, component: DataComponent) -> None:  
 *"""  
 Додавання компонента даних до папки.  
 """* self.components.append(component)  
  
 def remove\_component(self, component: DataComponent) -> None:  
 *"""  
 Видалення компонента даних з папки.  
 """* self.components.remove(component)  
  
 def process(self) -> None:  
 *"""  
 Обробка папки з даними шляхом обробки всіх її компонентів.  
 """* print(f"Processing data folder: {self.name}")  
 for component in self.components:  
 component.process()  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 # Створюємо компоненти даних  
 dataset1 = Dataset("Dataset 1")  
 dataset2 = Dataset("Dataset 2")  
 dataset3 = Dataset("Dataset 3")  
  
 # Створюємо папки та додаємо до них компоненти даних  
 folder1 = DataFolder("Folder 1")  
 folder1.add\_component(dataset1)  
 folder1.add\_component(dataset2)  
  
 folder2 = DataFolder("Folder 2")  
 folder2.add\_component(dataset3)  
  
 # Додаємо папки до іншої папки  
 main\_folder = DataFolder("Main Folder")  
 main\_folder.add\_component(folder1)  
 main\_folder.add\_component(folder2)  
  
 # Обробка головної папки, що виконає обробку всіх компонентів рекурсивно  
 main\_folder.process()

Використання патерну Interpreter тут менш доречне. Цей патерн зазвичай використовується для визначення граматики мови, а також для інтерпретації та виконання різних операцій над цією граматикою. Основна ідея полягає в тому, щоб мати клас для кожного символу (термінального або нетермінального) у спеціалізованій комп'ютерній мові, а кожне синтаксичне дерево речення у мові реалізовано через патерн composite використовується для оцінки (інтерпретації) речення для клієнта.

Очевидно, така структура тут менш доречна, адже вона могла б підійти для якоїсь більш складної файлової системи.

Щодо патерну Visitor, то він порівнювати його з Composite не дуже доречно, адже вони використовуються трохи для різних цілей.

Відвідувач - це поведінковий патерн проектування, який дозволяє відокремити алгоритми від об'єктів, на яких вони працюють. Він дозволяє визначати нові операції над набором об'єктів без зміни їхніх класів, інкапсулюючи операції в об'єкти-відвідувачі та приймаючи відвідувачів в об'єктах-відвідувачах.

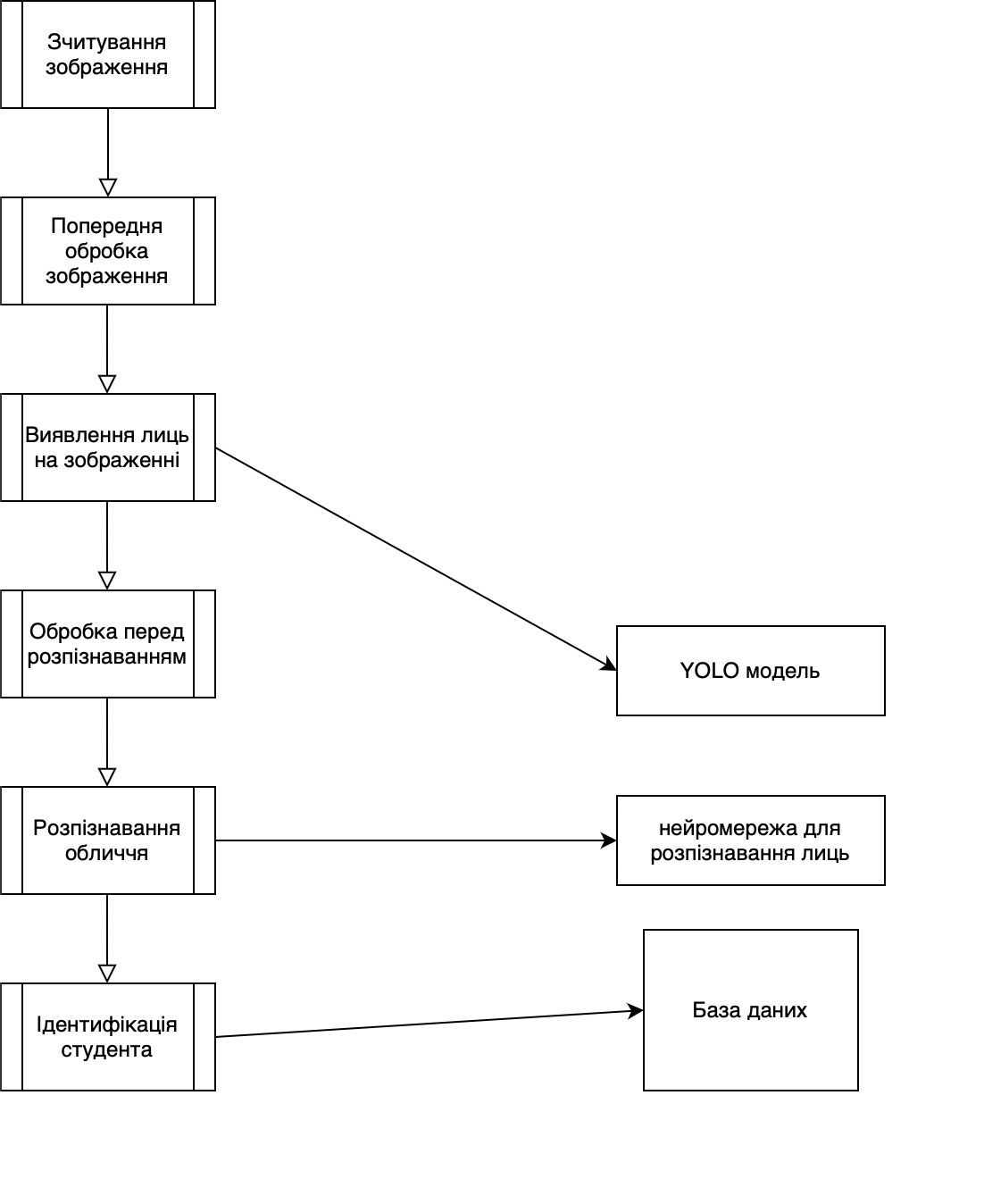
Тому тут більш доцільно поєднати патерни Composite та Visitor, використавши останній при обробці кожного з еллементів набору даних. (Як і було реалізовано в насправді у проєкті)

from abc import ABC, abstractmethod  
  
class DataComponent(ABC):  
 *"""  
 Базовий абстрактний клас, що визначає загальний інтерфейс для всіх компонентів даних.  
 """* @abstractmethod  
 def accept(self, visitor: 'DataVisitor') -> None:  
 *"""  
 Метод для прийняття відвідувача.  
 """* pass  
  
class Dataset(DataComponent):  
 *"""  
 Клас, що представляє набір даних.  
 """* def \_\_init\_\_(self, name: str):  
 self.name = name  
  
 def accept(self, visitor: 'DataVisitor') -> None:  
 *"""  
 Прийняття відвідувача для набору даних.  
 """* visitor.visit\_dataset(self)  
  
class DataFolder(DataComponent):  
 *"""  
 Клас, що представляє папку з даними, що містить інші компоненти даних.  
 """* def \_\_init\_\_(self, name: str):  
 self.name = name  
 self.components = []  
  
 def add\_component(self, component: DataComponent) -> None:  
 *"""  
 Додавання компонента даних до папки.  
 """* self.components.append(component)  
  
 def remove\_component(self, component: DataComponent) -> None:  
 *"""  
 Видалення компонента даних з папки.  
 """* self.components.remove(component)  
  
 def accept(self, visitor: 'DataVisitor') -> None:  
 *"""  
 Прийняття відвідувача для папки з даними.  
 """* visitor.visit\_data\_folder(self)  
 for component in self.components:  
 component.accept(visitor)  
  
class DataVisitor(ABC):  
 *"""  
 Базовий абстрактний клас, що визначає загальний інтерфейс для відвідувачів даних.  
 """* @abstractmethod  
 def visit\_dataset(self, dataset: Dataset) -> None:  
 *"""  
 Відвідування набору даних.  
 """* pass  
  
 @abstractmethod  
 def visit\_data\_folder(self, data\_folder: DataFolder) -> None:  
 *"""  
 Відвідування папки з даними.  
 """* pass  
  
class DataProcessingVisitor(DataVisitor):  
 *"""  
 Конкретний відвідувач, що здійснює обробку даних.  
 """* def visit\_dataset(self, dataset: Dataset) -> None:  
 *"""  
 Обробка набору даних.  
 """* print(f"Processing dataset: {dataset.name}")  
  
 def visit\_data\_folder(self, data\_folder: DataFolder) -> None:  
 *"""  
 Обробка папки з даними.  
 """* print(f"Processing data folder: {data\_folder.name}")  
  
  
if \_\_name\_\_ == "\_\_main\_\_":  
 # Створюємо компоненти даних  
 dataset1 = Dataset("Dataset 1")  
 dataset2 = Dataset("Dataset 2")  
 dataset3 = Dataset("Dataset 3")  
  
 # Створюємо папки та додаємо до них компоненти даних  
 folder1 = DataFolder("Folder 1")  
 folder1.add\_component(dataset1)  
 folder1.add\_component(dataset2)  
  
 folder2 = DataFolder("Folder 2")  
 folder2.add\_component(dataset3)  
  
 # Додаємо папки до іншої папки  
 main\_folder = DataFolder("Main Folder")  
 main\_folder.add\_component(folder1)  
 main\_folder.add\_component(folder2)  
  
 # Створюємо відвідувача для обробки даних  
 visitor = DataProcessingVisitor()  
  
 # Приймаємо відвідувача компонентами даних для обробки  
 main\_folder.accept(visitor)

1. Засіб для ідентифікації студентів в аудиторії за фотографією аудиторії.

Ідея: за фотографію використовуючи модель машинного навчання знаходити лиця людей та визначати їх положення на фотографії. Використовуючи окрему модель машинного навчання ефективно порівнювати лице людини з існуючою базою даних. Можна також за позицією лиця на фотографії (якщо камера завжди в одному місці) визначати положення людини в кімнаті та з’ясовувати, чи перебуває вона на своєму місці.

Діаграма:



1. Зчитування зображення класу, зняте камерою або завантажене користувачем.

2. Попередня обробка зображення: Обробляє фотографію класу, щоб покращити якість зображення та підготувати його для подальшого аналізу.

3. Виявлення лиць на зобаженні: Використовує алгоритми комп'ютерного зору для виявлення та знаходження людських облич на обробленому зображенні, скоріш за все YOLO модель останніх версій (найкращий на даний момент інструмент)

4. Обробка перед розпізнаванням: можливо, зображення потрібно вирівняти чи змінити перспективу, для того, аби підвищити якість розпізнавання у подальшому.

5. Розпізнавання обличчя: Порівнює виявлені риси обличчя з попередньо створеною базою даних облич студентів для пошуку потенційних збігів за допомогою алгоритмів машинного навчання.

6. Ідентифікація студента: Визначає ідентичність знайдених облич шляхом отримання інформації про студента з бази даних студентів. Остання зберігає інформацію про студентів, включаючи їхні імена, фотографії та будь-яку іншу додаткову інформацію.

References

1. Лекції з предмету Основи об’єктно орієнтованого програмування.
2. <https://en.wikipedia.org/wiki/Dependency_inversion_principle>
3. <https://refactoring.guru/uk/design-patterns/composite>
4. <https://en.wikipedia.org/wiki/Interpreter_pattern>
5. <https://refactoring.guru/uk/design-patterns/visitor>