Звіт

Лабораторна робота №2 ООП

Підготувала студентка 2 курсу ФКНК

групи ІПС21 Боровик Анастасія

У рамках лабораторних робіт та навчальних проєктів було реалізовано на ступні патерни проєктування:

# Decorator

**Декоратор** — це структурний патерн проектування, що дає змогу динамічно додавати об’єктам нову функціональність, загортаючи їх у корисні «обгортки».

У рамках навчального проєкту (1), створення нейромережі для генерації коду на основі моделі GPT-J, було використано реалізацію цього патерну на рівні мови програмування Python, яка хоч і працює з декорацією на рівні методів, а не на рівні класів, але є яскравим прикладом її реалізації. А саме, використано декоратори @custom\_fwd та @custom\_bwd з бібліотеки PyTorch для перевизначення алгоритмів прямого та зворотнього поширення у нейромережі. Така можливість дала змогу використати поблокову квантизацію за допомогою бібліотеки bitsandbytes та зменшити витрати пам’яті при роботі нейромережі.  
  
Самі фукнції для декорації виглядають наступним чином:

def custom\_fwd(fwd=None, \*, cast\_inputs=None):  
 *"""  
 Helper decorator for ``forward`` methods of custom autograd functions (subclasses of  
 :class:`torch.autograd.Function`). See the :ref:`example page<amp-custom-examples>` for more detail.  
  
 Args:  
 cast\_inputs (:class:`torch.dtype` or None, optional, default=None): If not ``None``,  
 when ``forward`` runs in an autocast-enabled region, casts incoming  
 floating-point CUDA Tensors to the target dtype (non-floating-point Tensors are not affected),  
 then executes ``forward`` with autocast disabled.  
 If ``None``, ``forward``'s internal ops execute with the current autocast state.  
  
 .. note::  
 If the decorated ``forward`` is called outside an autocast-enabled region,  
 :func:`custom\_fwd<custom\_fwd>` is a no-op and ``cast\_inputs`` has no effect.  
 """*

if fwd is None:  
 return functools.partial(custom\_fwd, cast\_inputs=cast\_inputs)  
  
 @functools.wraps(fwd)  
 def decorate\_fwd(\*args, \*\*kwargs):  
 args[0].\_dtype = torch.get\_autocast\_gpu\_dtype()  
 if cast\_inputs is None:  
 args[0].\_fwd\_used\_autocast = torch.is\_autocast\_enabled()  
 return fwd(\*args, \*\*kwargs)  
 else:  
 autocast\_context = torch.is\_autocast\_enabled()  
 args[0].\_fwd\_used\_autocast = False  
 if autocast\_context:  
 with autocast(enabled=False):  
 return fwd(\*\_cast(args, cast\_inputs), \*\*\_cast(kwargs, cast\_inputs))  
 else:  
 return fwd(\*args, \*\*kwargs)  
 return decorate\_fwd  
  
  
# Autograd ensures incoming gradients are the same type as forward outputs. Allowing a separate  
# cast\_inputs argument on custom\_bwd is unnecessary and could cause errors if it doesn't match  
# cast\_inputs supplied to custom\_fwd.  
def custom\_bwd(bwd):  
 *"""  
 Helper decorator for backward methods of custom autograd functions (subclasses of  
 :class:`torch.autograd.Function`).  
 Ensures that ``backward`` executes with the same autocast state as ``forward``.  
 See the :ref:`example page<amp-custom-examples>` for more detail.  
 """* @functools.wraps(bwd)  
 def decorate\_bwd(\*args, \*\*kwargs):  
 with autocast(enabled=args[0].\_fwd\_used\_autocast, dtype=args[0].\_dtype):  
 return bwd(\*args, \*\*kwargs)  
 return decorate\_bwd

Вони використані у класі для квантизації моделі:

class DequantizeAndLinear(torch.autograd.Function):  
 @staticmethod  
 @custom\_fwd  
 def forward(ctx, input: torch.Tensor, weights\_quantized: torch.ByteTensor,  
 absmax: torch.FloatTensor, code: torch.FloatTensor, bias: torch.FloatTensor):  
 weights\_deq = dequantize\_blockwise(weights\_quantized, absmax=absmax, code=code)  
 ctx.save\_for\_backward(input, weights\_quantized, absmax, code)  
 ctx.\_has\_bias = bias is not None  
 return F.linear(input, weights\_deq, bias)  
  
 @staticmethod  
 @custom\_bwd  
 def backward(ctx, grad\_output: torch.Tensor):  
 assert not ctx.needs\_input\_grad[1] and not ctx.needs\_input\_grad[2] and not ctx.needs\_input\_grad[3]  
 input, weights\_quantized, absmax, code = ctx.saved\_tensors  
 # grad\_output: [\*batch, out\_features]  
 weights\_deq = dequantize\_blockwise(weights\_quantized, absmax=absmax, code=code)  
 grad\_input = grad\_output @ weights\_deq  
 grad\_bias = grad\_output.flatten(0, -2).sum(dim=0) if ctx.\_has\_bias else None  
 return grad\_input, None, None, None, grad\_bias

Ще один приклад схожого використання цього патерну – декорація для бенчмаркінгу у проєкті (3):

def measure\_time(func: callable) -> callable:  
 *"""  
 Decorator function to measure the execution time of a given function.* ***:param*** *func: The function to measure the execution time of.* ***:type*** *func: callable* ***:return****: The wrapped function.* ***:rtype****: callable  
 """* def wrapper(\*args, \*\*kwargs):  
 *"""  
 Wrapper function that measures the execution time of the decorated function.* ***:param*** *args: Positional arguments to pass to the decorated function.* ***:type*** *args: tuple* ***:param*** *kwargs: Keyword arguments to pass to the decorated function.* ***:type*** *kwargs: dict* ***:return****: The result of the decorated function.  
 """* start\_time = time.time()  
 result = func(\*args, \*\*kwargs)  
 end\_time = time.time()  
 print(f"Execution time of {func.\_\_name\_\_}: {end\_time - start\_time:.2f} seconds")  
 return result  
  
 return wrapper

# Прототип

**Прототип** — це породжуючий патерн, який дозволяє копіювати об’єкти будь-якої складності без прив’язки до їхніх конкретних класів.

У рамках навчального проєкту (2), було використано реалізацію цього патерну на рівні мови програмування Python. Python надає власний інтерфейс Prototype через `copy.copy` і функції `copy.deepcopy`. І будь-який клас, який хоче реалізувати настроюваний реалізації мають перевантажити методи `\_\_copy\_\_` і `\_\_deepcopy\_\_`. У проєкті вони використовуються для створення копій моделі, адже окрім самої моделі об’єкти класу ModelLSTM зберігають ще додаткові атрибути.

Самі методи для реалізації прототипу виглядають наступним чином:

class ModelLSTM:  
 def \_\_init\_\_(self, input\_shape, loss, optimizer, callbacks, verbose=1):  
 self.loss = None  
 self.history = None  
 self.callbacks = callbacks  
 self.optimizer = optimizer  
  
 self.model = Sequential()  
 self.model.add(LSTM(720,  
 input\_shape=input\_shape,  
 return\_sequences=True))  
 self.model.add(LSTM(72))  
 # self.model.add(Dropout(0.01))  
 self.model.add(Dense(1))  
 if verbose:  
 print(self.model.summary())  
  
 self.model.compile(loss=loss, optimizer=optimizer)  
  
 def \_\_copy\_\_(self):  
 *"""  
 Create a shallow copy.  
 """* loss = copy.copy(self.loss)  
 history = copy.copy(self.history)  
 callbacks = copy.copy(self.callbacks)  
 optimizer = copy.copy(self.optimizer)  
 model = copy.copy(self.model)  
  
 new = self.\_\_class\_\_(  
 self.model.layers[0].input\_shape, loss, optimizer, callbacks, verbose=1  
 )  
 new.\_\_dict\_\_.update(self.\_\_dict\_\_)  
  
 return new  
  
 def \_\_deepcopy\_\_(self, memo=None):  
 *"""  
 Create a deep copy.  
 """* if memo is None:  
 memo = {}  
  
 loss = copy.copy(self.loss)  
 history = copy.copy(self.history)  
 callbacks = copy.copy(self.callbacks)  
 optimizer = copy.copy(self.optimizer)  
 model = copy.copy(self.model)  
  
 new = self.\_\_class\_\_(  
 self.model.layers[0].input\_shape, loss, optimizer, callbacks, verbose=1  
 )  
 new.\_\_dict\_\_ = copy.deepcopy(self.\_\_dict\_\_, memo)  
  
 return new

# Шаблонний метод

**Шаблонний метод** - це поведінковий шаблон проектування, який визначає скелет алгоритму в суперкласі, але дозволяє підкласам перевизначати окремі кроки алгоритму без зміни його структури.

Цей патерн можна гарно застосувати у проєкті (2) для очищення набору даних. Якщо дані приходять з різних джерел, для них можуть бути потрібні трохи різні очищення і індивідуальний підхід, але загалом структура алгоритму очищення одна. Тому, щоб уникнути дублювання коду, можна створити класс Cleaner:

**class** **Cleaner**():

**def** **clean**(self) -> **None**:

self.remove\_empty\_texts()

self.remove\_only\_hashtags()

self.round\_date()

self.remove\_irrelevant\_texts()

self.texts\_stemming()

**def** **remove\_empty\_texts**(self) -> **None**:

“””

Base operation

“””

**def** **round\_date**(self) -> **None**:

“””

Base operation

“””

**@abstractmethod**

**def** **remove\_only\_hashtags(**self) -> **None**:

“””

Operation relevant only for tweets

“””

**@abstractmethod**

**def** **remove\_irrelevant\_texts**(self) -> **None**:

“””

Operation must have different implementations

“””

**def** **texts\_stemming** (self) -> **None**:

“””

May be overridden, however it is not necessary

“””

**class** **TweetsCleaner**(Cleaner):

**def** **remove\_only\_hashtags**(self) -> **None**:

“””

Really cleans hashtags

“””

**def** **remove\_irrelevant\_texts** (self) -> **None**:

“””

Remove irrelevant tweets specifically.

“””

**class** **MediaCleaner**(Cleaner):

**def** **remove\_only\_hashtags** (self) -> **None**:

“””

May be empty or remove hashtags a different way

“””

**def** **remove\_irrelevant\_texts**(self) -> **None**:

“””

Remove irrelevant media texts specifically.

“””

**def** **texts\_stemming** (self) -> **None**:

“””

Stemms text a certin way.

“””

**def** **client\_code**(abstract\_cleaner: Cleaner) -> **None**:

# ...

abstract\_cleaner.clean()

# ...

**if** \_\_name\_\_ == "\_\_main\_\_":

print("Cleaning tweets:")

client\_code(TweetsCleaner())

print("Cleaning media texts")

client\_code(MediaCleaner())

Хоча дану логіку реалізувати легко, у проєкті вона не використовується через те, що на даний момент вхідні дані беруться з одного джерела.

# Стратегія

**Стратегія** — це поведінковий патерн проектування, який визначає сімейство схожих алгоритмів і розміщує кожен з них у власному класі. Після цього алгоритми можна заміняти один на інший прямо під час виконання програми.

У рамках проєкту (3) було реалізовано цей патерн наступним чином: потрібно було зручно використовувати сімейство алгоритмів для аугментації даних. Оскільки таких алгоритмів може бути багато, логічно зробити так, щоб контекст їх застосування не залежав від конкретної реалізації алгоритму. Створено клас Augmentor, якому у методах apply2sample та apply2folder можна передавати будь який із алгоритмів аугментації, кожен з яких реалізовано окремим класом.

class Augmentor:  
 @staticmethod  
 def apply2sample(transformations: list[callable],  
 image\_file\_path: str,  
 labels\_file\_path: str,  
 export\_images\_directory: str,  
 export\_labels\_directory: str,  
 name\_postfixes: list[str],  
 save\_originals: bool = True) -> None:

…

@measure\_time  
def apply2folder(self,  
 transformations: list[callable],  
 source\_directory: str,  
 export\_directory: str,  
 name\_postfixes: list[str],  
 save\_originals: bool = True) -> None:

…

cropper = Autocrop(probability=1.0, crop\_coefficient=0.8, crops\_per\_image=5)

flipper = Flip(probability=1.0, axis=0)

augmentor = Augmentor()  
augmentor.apply2folder(augmentations=[cropper, flipper])

# Адаптер

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

У навчальному проєкті (3) було використано цей патерн наступним чином:

Оскільки деяких алгоритмам зручно і логічно працювати з обмежувальними прямокутниками для об’єктів у вигляді координат кутів, а іншим вони потрібні у форматі центру та характеристик прямокутника, потрібно реалізувати адаптування останніх до перших. Реалізовано клас BoxAdapter:

class BoxAdapter:  
 @staticmethod  
 def get\_boxes\_corners(boxes: list[list[float]]):  
 *"""  
 Compute the corner coordinates of bounding boxes.* ***:param*** *boxes: A list of bounding boxes represented as [center\_x, center\_y, width, height].* ***:return****: An array of corner coordinates for each bounding box, where each row represents a box and each  
 column represents a corner point. The order of corners is [x1, y1, x2, y2, x3, y3, x4, y4].  
 """* assert all(len(inner\_lst) == 4 for inner\_lst in boxes), "Incorrect boxes format: must be [x, y, width, height]"  
   
 corners = []  
 for box in boxes:  
 center\_x, center\_y, width, height = box  
   
 x1 = center\_x - width / 2  
 y1 = center\_y - height / 2  
 x2 = center\_x + width / 2  
 y2 = center\_y - height / 2  
 x3 = center\_x - width / 2  
 y3 = center\_y + height / 2  
 x4 = center\_x + width / 2  
 y4 = center\_y + height / 2  
   
 corner = [x1, y1, x2, y2, x3, y3, x4, y4]  
 corners.append(corner)  
   
 return np.array(corners)

# Міст

**Міст** — це структурний патерн проектування, який розділяє один або кілька класів на дві окремі ієрархії — абстракцію та реалізацію, дозволяючи змінювати код в одній гілці класів, незалежно від іншої.

У навчальному проєкті (3) цей патерн реалізовано наступним чином: маємо багато алгоритмів для аугментації, кожен з яких реалізовано окремим класом. Ними усіма делегує клас Augmentor. Усі аугментації мають спільний інтерфейс Augmentation.

class Augmentation:  
 *"""  
 Interface for augmentation algorithms.  
 """* @abstractmethod  
 def \_\_call\_\_(self, sample: Sample) -> list[Sample]:  
 pass

class Flip(Augmentation):  
 *"""  
 Performs a vertical flip on the given image, takes bounding boxes into account.  
 """*

def \_\_call\_\_(self, sample: Sample) -> list[Sample]:  
 *"""  
 Performs augmentation* ***:param*** *sample: The sample to flip. Contains PIL.Image.Image, list[list[float]], list[int]* ***:return****: A list of augmented Sample objects.  
 """*

*…*

class Augmentor:  
 @staticmethod  
 def apply2sample(augmentations: list[Augmentation],  
 image\_file\_path: str,  
 labels\_file\_path: str,  
 export\_images\_directory: str,  
 export\_labels\_directory: str,  
 name\_postfixes: list[str],  
 save\_originals: bool = True) -> None:  
 # Applying augmentation  
 augmented\_samples.extend(augmentation(sample))

# Ланцюжок зв’язків

**Ланцюжок обов’язків** — це поведінковий патерн проектування, що дає змогу передавати запити послідовно ланцюжком обробників. Кожен наступний обробник вирішує, чи може він обробити запит сам і чи варто передавати запит далі ланцюжком.

Цей патерн можна було б застосувати наступним чином у навчальному проєкті (3):

Створити новий клас з назвою AugmentationPipeline. Цей клас отримає список об'єктів аугментацій і послідовно застосує їх до вхідного зразка:

from typing import List  
  
class AugmentationPipeline(Augmentation):  
 *"""  
 Represents a chain of commands for data augmentation.  
 """* def \_\_init\_\_(self, augmentations: List[Augmentation]) -> None:  
 self.augmentations = augmentations  
  
 def \_\_call\_\_(self, sample: Sample) -> List[Sample]:  
 augmented\_samples = [sample] # Start with the original sample  
  
 for augmentation in self.augmentations:  
 new\_samples = []  
 for augmented\_sample in augmented\_samples:  
 # Apply the current augmentation to the sample  
 augmented\_results = augmentation(augmented\_sample)  
 new\_samples.extend(augmented\_results)  
 augmented\_samples = new\_samples  
  
 return augmented\_samples

У цій реалізації **AugmentationPipeline** отримує список об'єктів аугментацій у своєму конструкторі. Коли конвеєр викликається зі зразком, він ітераційно перебирає аугментації і послідовно застосовує їх до зразка. Отримані зразки зберігаються у списку, і наступна аугментація застосовується до кожного зразка, доповненого на попередньому кроці. Нарешті, повертається список аугментацій зразків.

Для використання **AugmentationPipeline** можна створити його екземпляр з потрібними аугментаціями і викликати його разом із зразком:

pipeline = AugmentationPipeline([Rotate(angle=90), Flip(), Smoke()])  
  
# Call the pipeline with a sample  
sample = Sample(image="example.jpg")  
augmented\_samples = pipeline(sample)

(Зауважимо, що таке використання патерну не було використане через те, що був застостований патерн стратегія, який іншим чином вирішив ці ж питання).

# Команда

**Команда** - це поведінковий патерн проектування, який перетворює запит на окремий об'єкт, що містить всю інформацію про запит. Таке перетворення дозволяє передавати запити як аргументи методу, затримувати або ставити в чергу виконання запиту, а також підтримувати операції, які можна скасувати.

У навчальному проєкті (3), де ми працюємо над аугментаціями, ми можемо ввести новий клас ImageAugmentationCommand. Цей клас інкапсулює конкретну операцію доповнення у вигляді команди, що дозволяє нам параметризувати та ставити операції у чергу, планувати їх виконання, а також потенційно підтримувати операції, які можна скасувати. Ось приклад реалізації:

class ImageAugmentationCommand(ABC):  
 *"""  
 Abstract base class for image augmentation commands.  
 """* def \_\_init\_\_(self, sample: Sample):  
 self.sample = sample  
  
 @abstractmethod  
 def execute(self) -> None:  
 pass  
  
 @abstractmethod  
 def undo(self) -> None:  
 pass

Клас ImageAugmentationCommand визначено як абстрактний базовий клас з двома абстрактними методами: execute і undo. Підкласи цього класу будуть реалізовувати ці методи для визначення конкретних операцій доповнення зображення.

Розглянемо приклад конкретної операції доповнення під назвою FlipCommand:

class FlipCommand(ImageAugmentationCommand):  
 *"""  
 Concrete command class for flipping the image.  
 """* def execute(self) -> None:  
 # Perform the flip augmentation on the image  
 self.sample.image = self.sample.image.transpose(Image.FLIP\_LEFT\_RIGHT)  
 # Update the bounding boxes if necessary  
  
 def undo(self) -> None:  
 # Undo the flip operation (optional)  
 self.sample.image = self.sample.image.transpose(Image.FLIP\_LEFT\_RIGHT)  
 # Restore the original bounding boxes if necessary

У цьому прикладі метод execute виконує збільшення зображення при перевертанні, а метод undo може бути застосований для скасування операції перевертання, якщо це необхідно. Метод execute також може оновити рамки, пов'язані із зображенням, якщо це необхідно.

Використовуючи паттерн Command, ми можемо створювати екземпляри конкретних команд доповнення і передавати їх як аргументи до класу Augmentor або зберігати їх у черзі для пакетної обробки. Потім клас Augmentor може виконати команди у відповідний час:

flip\_command = FlipCommand(sample)   
flip\_command.execute()   
flip\_command.undo()

# Медіатор

**Медіатор** - це поведінковий патерн дизайну, який дозволяє зменшити хаотичні залежності між об'єктами. Паттерн обмежує прямі зв'язки між об'єктами і змушує їх співпрацювати лише через об'єкт-медіатор.

У навчальнму проєкті (3) паттерн медіатор може бути використаний для зменшення хаотичних залежностей між алгоритмами доповнення (класами доповнення) та класом доповнювача (Augmentor). Ввівши об'єкт-медіатор, ми можемо інкапсулювати зв'язок і координацію між алгоритмами і доповнювачем, таким чином зменшуючи прямі залежності і роблячи систему більш гнучкою і зручною в обслуговуванні.

Щоб реалізувати паттерн Mediator у цьому сценарії, ми можемо створити новий клас під назвою AugmentationMediator, який виступатиме посередником між доповненнями та аугментором. AugmentationMediator забезпечить централізований інтерфейс для комунікації та координації між цими об'єктами.

Ось приклад реалізації:

class AugmentationMediator:  
 def \_\_init\_\_(self):  
 self.augmentations = []  
  
 def register\_augmentation(self, augmentation: Augmentation):  
 self.augmentations.append(augmentation)  
  
 def apply\_augmentations(self, sample: Sample) -> list[Sample]:  
 augmented\_samples = [sample]  
 for augmentation in self.augmentations:  
 augmented\_samples += augmentation(sample)  
 return augmented\_samples  
  
 def export\_augmented\_samples(self, augmented\_samples: list[Sample], export\_images\_directory: str, export\_labels\_directory: str, name\_postfixes: list[str], save\_originals: bool = True):  
 # Export the augmented samples using the Augmentor's export functionality  
 Augmentor().export\_augmented\_samples(augmented\_samples, export\_images\_directory, export\_labels\_directory, name\_postfixes, save\_originals)  
  
  
# Usage example:  
augmentation\_mediator = AugmentationMediator()  
  
# Register augmentations with the mediator  
flip\_augmentation = Flip()  
resize\_augmentation = Resize()  
rotation\_augmentation = Rotate()  
  
augmentation\_mediator.register\_augmentation(flip\_augmentation)  
augmentation\_mediator.register\_augmentation(resize\_augmentation)  
augmentation\_mediator.register\_augmentation(rotation\_augmentation)  
  
# Apply augmentations and export augmented samples using the mediator  
sample = Sample(image='path/to/image.jpg')  
augmented\_samples = augmentation\_mediator.apply\_augmentations(sample)  
augmentation\_mediator.export\_augmented\_samples(augmented\_samples, export\_images\_directory='output/images', export\_labels\_directory='output/labels', name\_postfixes=['flip', 'resize', 'rotate'], save\_originals=True)

У цій реалізації клас AugmentationMediator виступає в ролі централізованого посередника, який координує зв'язок між доповненнями та аугментором. Метод register\_augmentation дозволяє доповненням зареєструватися у медіатора. Метод apply\_augmentations застосовує всі зареєстровані доповнення до заданого зразка і повертає список доповнених зразків. Нарешті, метод export\_augmented\_samples використовує клас Augmentor для експорту розширених зразків на основі наданих каталогів експорту та постфіксів імен.

Використовуючи паттерн Mediator, ми зменшили пряму залежність між доповненнями та Augmentor, що робить систему більш гнучкою. Розширенням не потрібно знати про Аугментор, а Аугментору не потрібно знати про конкретні застосовані розширення. Посередник інкапсулює логіку взаємодії і дозволяє легше розширювати і повторно використовувати компоненти.

# Спостерігач

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

У навчальному проєкті (3) один з можливих способів застосування паттерну спостерігач - це реалізація механізму, за допомогою якого декілька об'єктів можуть спостерігати за змінами, що відбуваються внаслідок доповнень, застосованих до зразків.

Для цього ми можемо ввести клас Observer, який визначає механізм підписки і дозволяє об'єктам спостерігати за доповненнями, застосованими до зразків:

class Observer:  
 *"""  
 Observer class for subscribing to augmentation events.  
 """* def \_\_init\_\_(self):  
 self.observers = []  
  
 def subscribe(self, observer):  
 *"""  
 Subscribe an observer to receive notifications.  
 """* self.observers.append(observer)  
  
 def unsubscribe(self, observer):  
 *"""  
 Unsubscribe an observer from receiving notifications.  
 """* self.observers.remove(observer)  
  
 def notify(self, sample):  
 *"""  
 Notify all subscribed observers about the augmentation event.  
 """* for observer in self.observers:  
 observer.update(sample)  
  
  
class Augmentation:  
 *"""  
 Interface for augmentation algorithms.  
 """* def \_\_init\_\_(self):  
 self.observer = Observer()  
  
 def subscribe(self, observer):  
 *"""  
 Subscribe an observer to receive notifications about the augmentation events.  
 """* self.observer.subscribe(observer)  
  
 def unsubscribe(self, observer):  
 *"""  
 Unsubscribe an observer from receiving notifications.  
 """* self.observer.unsubscribe(observer)  
  
 def notify\_observers(self, sample):  
 *"""  
 Notify all subscribed observers about the augmentation event.  
 """* self.observer.notify(sample)  
  
 @abstractmethod  
 def \_\_call\_\_(self, sample: Sample) -> list[Sample]:  
 pass  
  
  
class Flip(Augmentation):  
 def \_\_call\_\_(self, sample: Sample) -> list[Sample]:  
 # Apply the flip augmentation to the sample  
 # ...  
  
 # Notify the observers about the augmentation event  
 self.notify\_observers(sample)  
 return [sample]  
  
  
# Usage example:  
  
class SampleObserver:  
 *"""  
 Observer class that logs the augmentation events.  
 """* def update(self, sample):  
 print(f"A sample was augmented: {sample}")  
  
  
flip\_augmentation = Flip()  
observer = SampleObserver()  
  
flip\_augmentation.subscribe(observer)  
  
# Now, when the flip augmentation is applied, the observer will be notified.

Клас Observer підтримує список підписаних спостерігачів. Клас Augmentation модифіковано так, щоб він містив екземпляр Observer і надавав методи для підписки, відписки та сповіщення спостерігачів. Розширення Flip (як приклад) викликає метод notify\_observers після застосування доповнення Flip до зразка, який запускає метод оновлення всіх підписаних спостерігачів.

За допомогою патерну довзволяється декільком об'єктам спостерігати за доповненнями, застосованими до зразків, і реагувати на них відповідно. Це може бути корисно, наприклад, якщо у вашій системі є різні компоненти, які повинні виконувати додаткові дії на основі застосованих доповнень, такі як ведення журналу, візуалізація або аналіз змін, внесених до зразків.