The Decorator Pattern

Welcome back to CS 2100!

Prof. Rasika Bhalerao

Defining distance

Algorithms for "shortest distance" (root to each node, MST) -> must define "distance"

Part of HW9: defining the distance between train stations

- Basic measure of Euclidean distance (used in code)
- Number of minutes it takes to travel: "the station is 15 minutes from here"
- Financial cost of building th track (which is bigger if we have to dig a tunnel underwater or through a mountain)
- `1/n`, where `n` is the number of passengers who regularly travel between the two stations

Strategy pattern

Many definitions of distance, one definition of Uniform Cost Search

Strategy (Duration) Pattern: allow user to choose / change distance definition at run time

Ex: Strategies for Tic Tac Toe

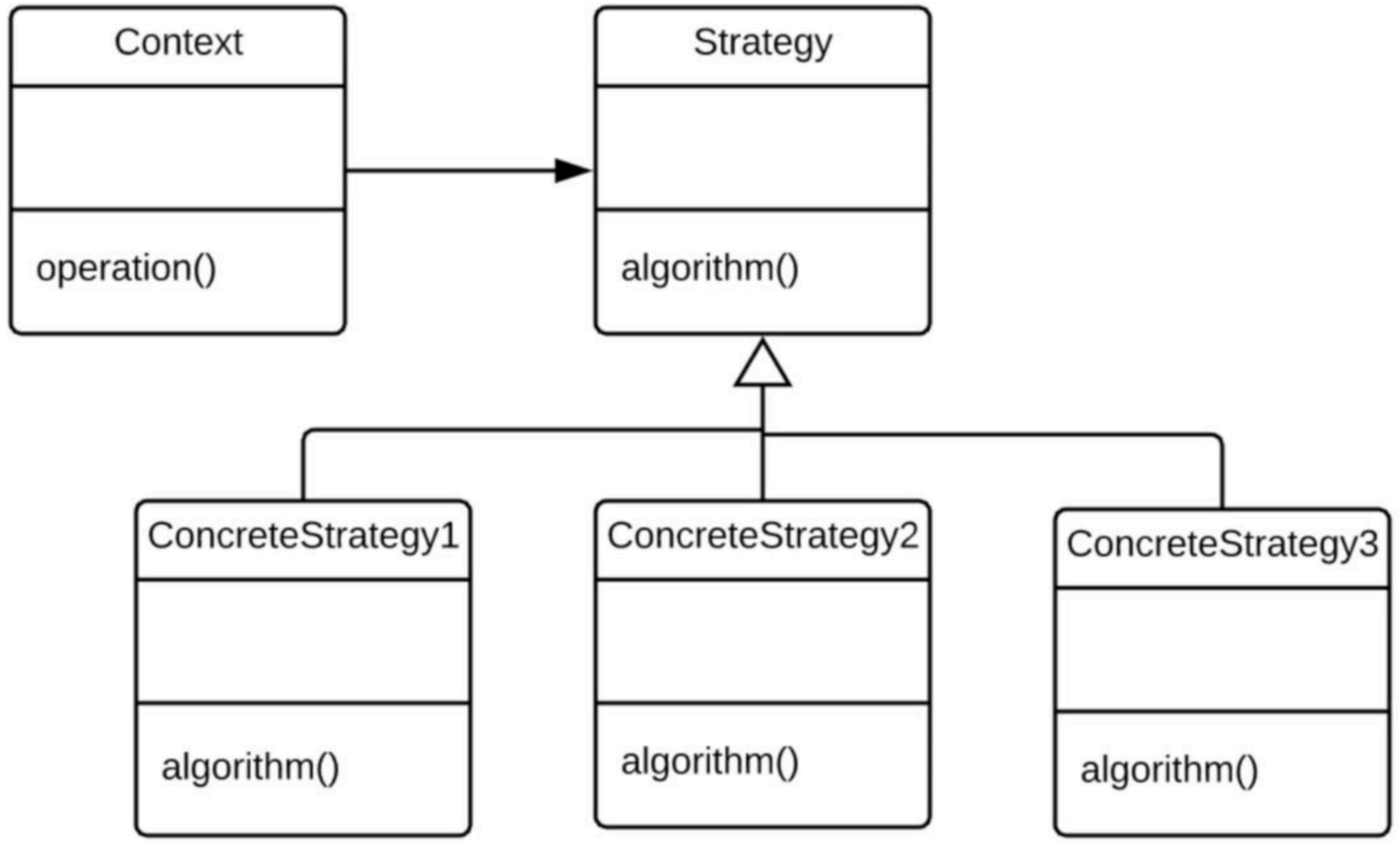
- Place a third piece in a row to win
- Block the opponent if they're about to win
- Place in an open corner
- Place in any open square

Strategy for choosing a place changes based on game state

Ex: maps directions

- Shortest path
- Shortest time
- Least emissions
- Least tolls
- Maximize sightseeing

User chooses the path-finding algorithm at runtime


Context
operation()
Strategy
algorithm()
ConcreteStrategy1
algorithm()
ConcreteStrategy2
algorithm()
ConcreteStrategy3
algorithm()

Poll: Why do we use composition to hold the Strategy (instead of inheritance through subclasses)?

1. Because subclasses of the Strategy class would not work properly
2. Using inheritance would require methods to copy over the history when we switch strategies (it would need a method for every possible pair of strategies)
3. When we invent a new Strategy, if we were using inheritance, we would need to write methods to copy over the history to/from that new Strategy
4. Um actually, we do use inheritance to extend the Strategy, not composition to hold it

Example: Algorithms for sorting lists

Merge Sort is most efficient for long lists (O(n log n)):

1. Split the list in half
2. Sort each half (recursion)
3. Merge the two halves together

Insertion Sort is more efficient for short (< 30) lists (O(n^2)):

Insert each element into the right position in the list s.t. it's sorted

Example code to implement this is attached. Let's step with a debugger.

`AdaptiveSorter` chooses the sorting strategy based on list length

Function Decorators

We have already seen some decorators:

- `@property` (to create a property named after the method)
- `*.setter` (to give that property a "setter" method)
- `@classmethod` (for `setUpClass(cls)` and `tearDownClass(cls)` to run once before / after unit testing)

We will find out how these decorators were implemented / how to create our own decorators

Functions are objects that can be mutated

We already knew functions are objects:

```python
from typing import Callable

def double(num: int) -> int:
    return num * 2

def triple(num: int) -> int:
    return num * 3

functions: list[Callable[[int], int]] = [double, triple]

def apply_functions(num: int) -> list[int]:
    return [func(num) for func in functions]

print(apply_functions(5))  # [10, 15]
```

A function decorator is a way to mutate a function to modify what it does.

Function decorators mutate functions

Example: `@time_calls` modifies a given function to print the time it took to run it:

```python
@time_calls
def multiply_list(nums: list[int], multiplier: int = 1) -> list[int]:
    return [num * multiplier for num in nums]

print(multiply_list([i for i in range(5)], multiplier = 3))
```

Because we modified `multiply_list()` to do what `@time_calls` tells it to do, this is the printed output:

```
Executed multiply_list with ([0, 1, 2, 3, 4],) and {'multiplier': 3} in 7.152557373046875e-06ms
[0, 3, 6, 9, 12]
```

How we created that `@time_calls` decorator:

```python
def time_calls(func: Callable[..., Any]) -> Callable[..., Any]:
    def wrapper(*args: Any, **kwargs: Any) -> Any:
        now = time.time()
        return_value = func(*args, **kwargs)
        print(f'Executed {func.__name__} with {args} and {kwargs} in {time.time() - now}ms')
        return return_value
    return wrapper
```

`time_calls()` takes a function as its argument and returns a function.

The returned function is a modified version of the argument function.

That modification: in addition to running the function as usual, it prints the number of milliseconds that it took to run it.

Poll: What does the `@mystery` decorator do?

```python
T = TypeVar('T')

def mystery(func: Callable[[float], T]) -> Callable[[float], T]:
    def wrapper(x: float) -> T:
        if x < 0:
            raise ValueError("Argument must be positive")
        return func(x)
    return wrapper

@mystery
def square_root(x: float) -> Any:
    return x ** 0.5

print(square_root(9))
```

1. Modifies the function to print the number of milliseconds it took to run it
2. Modifies the function such that it raises a `ValueError` if the argument is negative
3. Modifies the function to run it twice
4. Nothing

Caching / memoization: a common use of decorators

Memoization / caching: temporarily storing values that are recalculated often (so they only need to be calculated once).

```python
def fibonacci(n: int) -> int:
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)
```

In order to calculate the 5th Fibonacci number, we need to calculate the 4th and 3rd ones.
And to calculate the 4th number, we need to calculate the 3rd and 2nd ones.
And to calculate the 3rd number, we need to calculate the 2nd and 1st ones...

There are a lot of recalculated numbers here. Let's memoize.

Let's store each Fibonacci number *the first time it is calculated*, so that each subsequent time, we can just look it up

```
R = TypeVar('R')

def memoize(func: Callable[..., R]) -> Callable[..., R]:
    cache: dict[tuple[Any, ...], R] = {}
    def wrapper(*args: Any) -> R:
        if args in cache:
            return cache[args]
        result = func(*args)
        cache[args] = result
        return result
    return wrapper

@memoize
def fibonacci(n: int) -> int:
    if n < 2:
        return n
    return fibonacci(n-1) + fibonacci(n-2)

print(fibonacci(10))
```

Poll: Can we re-use the `@memoize` decorator to perform memoization for other functions in the same file?

1. Yes
2. Yes, but that other function would share the "cache" of stored values, which are specific to Fibonacci
3. No, it would not run

Easiest way to test this: create a duplicate method called `fibonacci2()`, decorate it, call it after the regular `fibonacci()`, and step through it in the debugger.

Preserving function metadata

Functions have metadata such as their name and docstring:

```python
def double(num: int) -> int:
    """Doubles a number."""
    return num * 2

print(double.__name__) # double
print(double.__doc__) # Doubles a number.
```

Its metadata is overwritten by the wrapper's metadata:

```python
def my_decorator(
        func: Callable[..., Any]
) -> Callable[..., Any]:

    def wrapper(
            *args: Any, **kwargs: Any
    ) -> Any:
        print("Before calling function")
        result = func(*args, **kwargs)
        print("After calling function")
        return result
    return wrapper

@my_decorator
def double(num: int) -> int:
    """Doubles a number."""
    return num * 2

print(double.__name__) # wrapper
print(double.__doc__) # None
```

`@wraps` decorator copies metadata from passed function to wrapper function

```python
from functools import wraps

def my_decorator(func: Callable[..., Any]) -> Callable[..., Any]:
    @wraps(func)  # This copies func's metadata to wrapper
    def wrapper(*args: Any, **kwargs: Any) -> Any:
        print("Before calling function")
        result = func(*args, **kwargs)
        print("After calling function")
        return result
    return wrapper

@my_decorator
def double(num: int) -> int:
    """Doubles a number."""
    return num * 2

print(double.__name__)  # double
print(double.__doc__)   # Doubles a number.
```

Poll: Why do we use `wraps` from the `functools` package?

1. To copy the metadata from the original (un-decorated) function to the wrapper function that decorates it
2. To copy the metadata from the decorating wrapper function to the original (un-decorated) function
3. To make it so we can use a function as a decorator with the `@` symbol
4. `wraps` is like a type hint: it doesn't do anything and isn't enforced in Python by default

Class decorators

Function decorators are popular in Python.

Class decorators are more popular in other languages including Java.

A *class* decorator is also a decorator that uses the `@` symbol. The only difference is that it decorates a class instead of a function.

`@add_logging` modifies a class's `__init__()` to add a `print()` statement for logging.

```python
class HasInit(Protocol):
    # to ensure we only use`@add_logging on classes that have `__init__()`
    def __init__(self, *args: Any, **kwargs: Any) -> None: ...

C = TypeVar('C', bound=HasInit)

def add_logging(cls: Type[C]) -> Type[C]:
    original_init = cls.__init__

    @wraps(original_init)
    def new_init(self: Any, *args: Any, **kwargs: Any) -> None:
        print(f"Creating instance of {cls.__name__}")
        original_init(self, *args, **kwargs)

    cls.__init__ = new_init  # type: ignore[misc]
    return cls

@add_logging
class User:
    def __init__(self, name: str) -> None:
        self.name = name

user = User('Mini')
```

Poll:

1. What is your main takeaway from today?
2. What would you like to revisit next time?