Decorators 3

Real Life example 3:

```
def count_calls(func):
    @functools.wraps(func)
    def wrapper_count_calls(*args, **kwargs):
        wrapper_count_calls.num_calls += 1
        print(f"Call {wrapper_count_calls.num_calls} of {func.__name__!r}")
        return func(*args, **kwargs)
    wrapper_count_calls.num_calls = 0
    return wrapper_count_calls
```

```
import functools

def cache(func):
    """Keep a cache of previous function calls"""
    @functools.wraps(func)
    #####
    def wrapper_cache(*args, **kwargs):
        cache_key = args + tuple(kwargs.items())
        if cache_key not in wrapper_cache.cache:
            # ["100,200,300"]: 50
            wrapper_cache.cache[cache_key] = func(*args, **kwargs)
        return wrapper_cache.cache[cache_key]
    wrapper_cache.cache = dict()
    return wrapper_cache

@cache
@count_calls
def fibonacci(num):
    if num < 2:
        return num
    return fibonacci(num - 1) + fibonacci(num - 2)
```

Testing:

```
fibonacci(12)
fibonacci(12)
fibonacci(8)
```

Decorator-Enhanced Strategy Pattern

“Case Study: Refactoring Strategy”

Suppose you want to create a “meta-strategy” that selects the best available
discount for a given Order. The repetition is problematic because someone
may add a new promotional strategy function and forget to manually add it to the
promos
list—in which case, best_promo will silently ignore the new strategy, introducing a subtle
bug in the system

```
promos = []
def promotion(promo_func):
 promos.append(promo_func)
 return promo_func
@promotion
def fidelity(order):
    ## logic ##
 """5% discount for customers with 1000 or more fidelity points"""
 return order.total() * .05 if order.customer.fidelity >= 1000 else 0
@promotion
def bulk_item(order):
 """10% discount for each LineItem with 20 or more units"""
 discount = 0
 for item in order.cart:
     if item.quantity >= 20:
         discount += item.total() * .1
 return discount
@promotion
def large_order(order):
 """7% discount for orders with 10 or more distinct items"""
 distinct_items = {item.product for item in order.cart}
 if len(distinct_items) >= 10:
     return order.total() * .07
 return 0
def best_promo(order):
 """Select best discount available
```

```
"""
return max(promo(order) for promo in promos)
```

Explanation

- The promos list starts empty.
- promotion decorator returns promo_func unchanged, after adding it to the promos list.
- Any function decorated by @promotion will be added to promos.
- No changes needed to best_promos, because it relies on the promos list.

💡 Most decorators do change the decorated function. They usually do it by defining an
inner function and returning it to replace the decorated function. Code that uses inner
functions almost always depends on closures to operate correctly

- It is interesting to note that in Design Patterns the authors suggest: “Strategy objects
often make good flyweights.”
- A definition of the Flyweight in another part of that work states: “A flyweight is a shared object that can be used in multiple contexts simultane-ously.”
- The sharing is recommended to reduce the cost of creating a new concrete strategy object when the same strategy is applied over and over again with every new context—with every new Order instance.