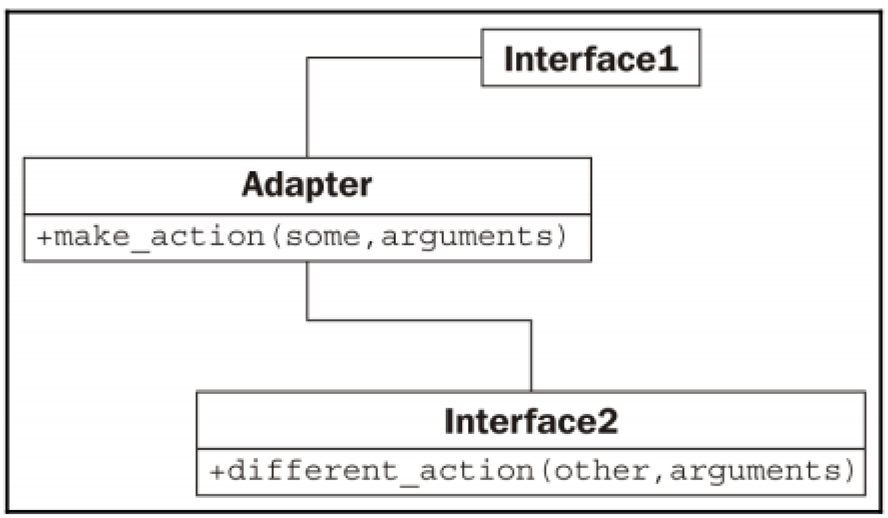
Chapter 11 - Python Design Patterns II

April 4, 2021

0.1 Adapter Pattern

- designed to interact with existing code
- adapters are used to allow two preexisting objects to work together, even if their interfaces are not `compatible`
- sits between two different interfaces, translatin between them on the fly

- adapter pattern is simmular to the decorator pattern
- decorators typically provide the same interface that they replace, whereas adapter map between two different interfaces
- `interface1` is excepting to call a method called `make_action(some_arguments)`
- we already have this perfect `Interface2` class that does everything we want and we dont want to rewrite it
- but it provides a method called `different_action(other, arguments)` instead
- the `Adapter` class implements the `make_action` interface and maps the arguments to existing interface



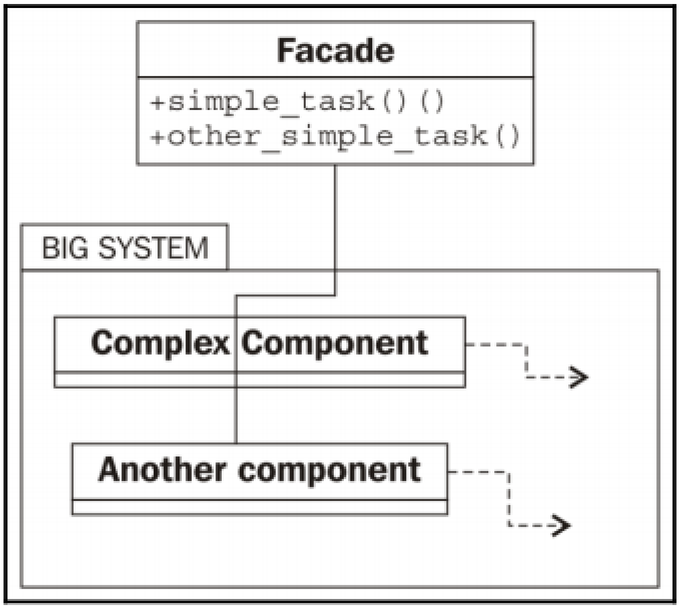
- thw advantage here is that the code that maps from one interface to another is all in one
- the alternative would be really ugly; we'd have to perform the translation i multiple places whenever we need to access this code
- image the case below where we are provided a date from a third party libary
- we can use an adapator to use pythons built in `datetime` libary

[1]:
```python
import datetime

class AgeCalculator:
    def __init__(self, birthday):
        self.year, self.month, self.day = (
            int(x) for x in birthday.split("-")
        )

    def calculate_age(self, date):
        year, month, day = (int(x) for x in date.split("-"))
        age = year - self.year
        if (month, day) < (self.month, self.day):
            age -= 1
        return age

class DateAgeAdapter:
    def _str_date(self, date):
        return date.strftime("%y-%m-%d")

    def __init__(self, birthday):
        birthday = self._str_date(birthday)
        self.calculator = AgeCalculator(birthday)

    def get_age(self, date):
        date = self._str_date(date)
        return self.calculator.calculate_age(date)
```

- the adapter converts `datetime.date` and `datetime.time` into a sring that our orginal `AgeCalculator` can use

0.2 Facade Pattern

- the `facade` pattern is designed to provide a simple interface to a complex system of components
- for complex tasks, we may need to interact with these objects directly, but there is often a `typical` usage for the system for which these complicated interactions are not necessary
- the facade pattern allows us to define a new object that encapsulates this typical usage of the system
- anytime we want to access to common functionality, we can use the single objects simplified interface



- a facade is simmilar to an adapter
- the primary difference is that a facade tried to abstract a simpler interface out of a complex one, while an adapter only tired to map one existing interface to another
- the EmailFacade class is initialized with the hostname of the email server, a username and a password to log in

[6]:
```
import smtplib
import imaplib

class EmailFacade:
    def __init__(self, host, username, password):
        self.host = host
        self.username = username
        self.password = password

def send_email(self, to_email, subject, message):
    if not "@" in self.username:
        from_email = f"{self.username}@{self.host}"
    else:
        from_email = self.username
        message = (
            f"From: {from_email}\r\n To: {to_email} \r\n Subject␣
↪{subkect}\r\n\r\n {message}"
        )
        smtp = smtplib,SMTP(self.host)
        smpt.login(self.username, self.password)
        smpt.sendmail(from_email, [to_email], message)
```

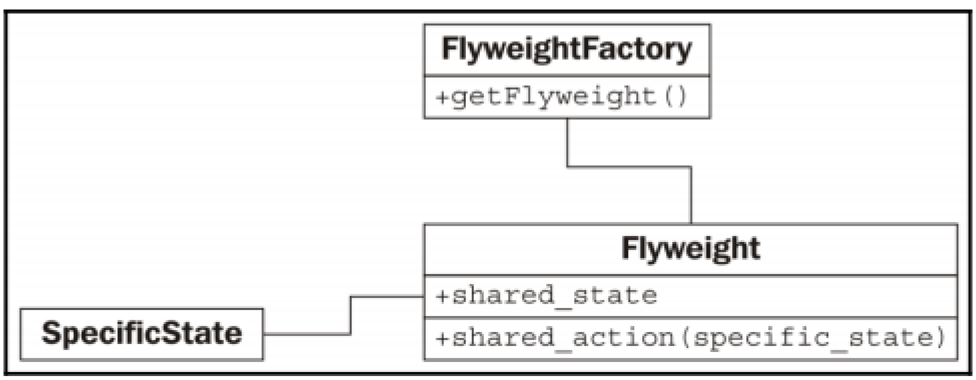
```python
def get_inbox(self):
    mailbox = imaplib.IMAP4(self.host)
    mailbox.login(
        bytes(self.username, "utf8"), bytes(self.password, "utf8")
    )
    mailbox.select()
    x, data = mailbox.search(None, "ALL")
    messages = []
    for num in data[0].split():
        x, message = mailbox.fetch(num, "(RFC822)")
        messages.append(message[0][1])
    return messages
```

- taken together, we have simple facade class that can send and receive messages in a fairly straightforward manner

- facade pattern is an integral part of the python ecosystem
- an example is `for` loops or `list` comprehensions
- the `defaultdict` implementation is a facade that abstracts away annoying corner cases whena key doesn't exist in a dictionary
- the `requests` libary is a powerful facade over less readable libaries for `HTTP` requests, which are themselves a facade over managing the text-based `HTTP` protocol

0.3 Flyweight Pattern

- this is a memory optimization pattern
- flyweight pattern ensures that objects that share a state can use the same memory for that shared state
- implementd only after a program has demonstrated memeory problems



- each `Flyweight` has no specific state
- any time it needs to perform an operation on `SpecificState`, that state needs to be passed into the `Flyweight` by calling code
- traditionally, the factory that returns a flywieght is a seprate object
- its purpose is to retunr a flywieght for a given key identifying that flyweight
- if the flyweight exists, we return it, otherwise, we create a new one
- works simmilar to the `signleton` pattern

- `flyweight` factory is often implemented using the `__new__constructor`
- unlike the signleton pattern which only needs to return one instace of the class, we need to be able to return different instances depending on the key
- we could store the items in a dictionary and look them up based on the key
- the problem is that we want to get rid of the object from memory if there are no more

- we can solve this by taking advantage of Pythons `weakref` module
- the module provides a `WeakValueDictionary` object, which basically allows us to store items in a dictionary without the garbage collector caring about them
- if a value is in a weak refrenced dictionary, and there are no other refrences to that object stored anywhere in the application, the garbase collector will eventually clean up for us

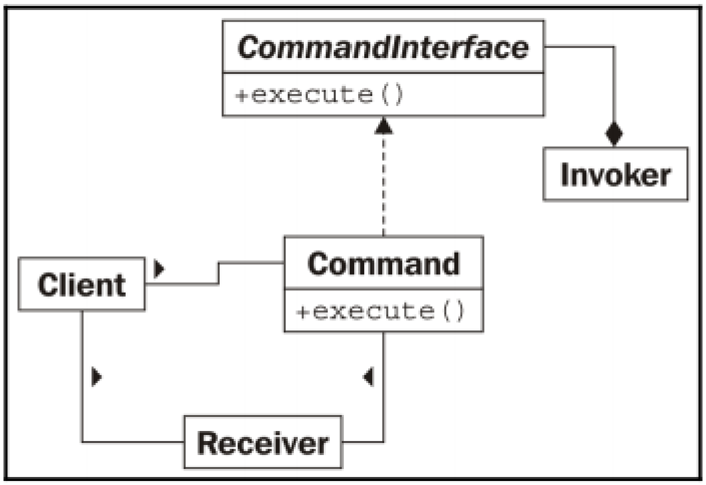
[7]:
```python
import weakref

class CarModel:
    _models = weakref.WeakValueDictionary()

    def __new__(cls, model_name, *args, **kwargs):
        model = cls._model.get(model_name)
        if not model:
            model = super().__new__(cls)
            cls._models[model_name] = model
        return model
```

- basically, whenever we construct a new flyweight with a given name, we first look up that name in the weak refrenced dictionary
- if it exists, we return that model; if not, we create a new one

- flyweight pattern is complex but it is designed for conserving memeory
- if we have hundreds of thousands of similar objects, combining similar properties into a flyweight can have enormous impact on memory consumption

0.4 Command Pattern

- the command pattern adds a level of abstraction between actions that must be done and the object that invokes those actions, normally at a later time
- in the command pattern, client code creates a `Command` object that can be executed at a later date
- the object knows about a receiver object that manages its own internal state when the command is executed on it
- the `Command` object implements a specific interface
- typically it has an `execute` or `do_action` method and also keeps track of any arguments required to perform the action
- finally, one or more `Invoker` object executes the command at the correct time



- an example of this can be seen on graphical window
- often an action can be invoked by a menu item on the menu bar
- these are all examples of `Invoker` objects
- the action that actually occurs such as `Exit`, `Save`, or `Copy` are implementations of `CommandInterface`
- a GUI window to receive exit, a dcoument to receive save and `ClipboardManager` to receive a copy command are all examples of possible `Receivers`

- the command pattern does not feel pythonic
- the one below is more pythonic but we have done some decoupling in exchange for it

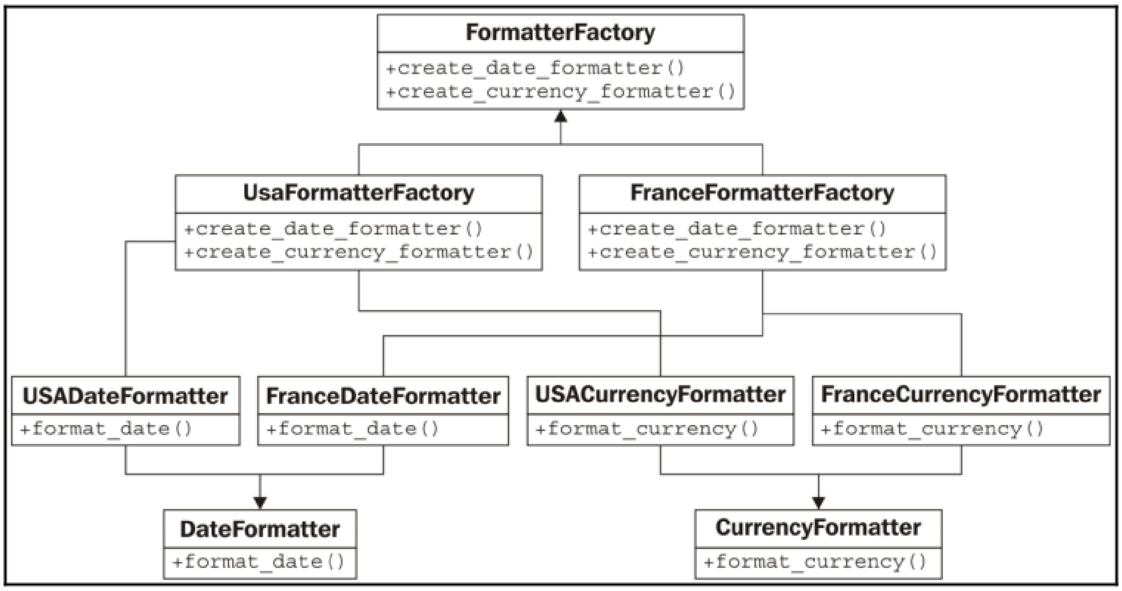
[8]:
```
import sys

class Window:
    def exit(self):
        sys.exit(0)

class MenuItem:
    def click(self):
        self.command()

window = Window()
menu_item = MenuItem()
menu_item.command = window.exit
```

0.5 Abstract Factory Pattern

- normally used when we have multiple possible implementations of a system that depend on some configuration or platform issue
- the calling code requests an object from the abstract factory, not knowing what class of object will be returned
- the underlying implementation returned may depend on a variety of factors, such as current locale, operating system or local configuration

- Django provides an abstract factory that returns a set of object relation classes for interacting with a specific database backend depending on a configuration

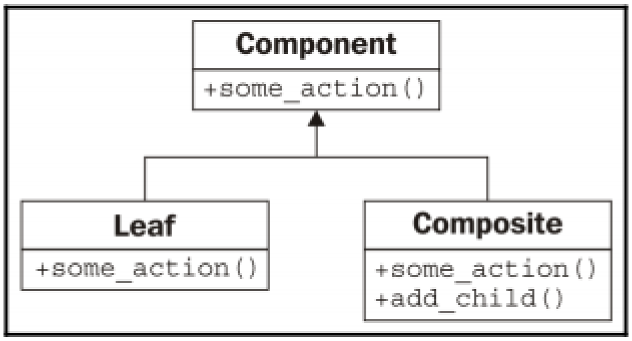
- in the example below, we create a set of formatters that depend on a specific locale and help us format dates and currencies
- there will be an abstract factory class that picks that specific factory, as well as a couple of example concrete factories
- each of these will create formatter objects for dates and times, which can be queried to format a specific value



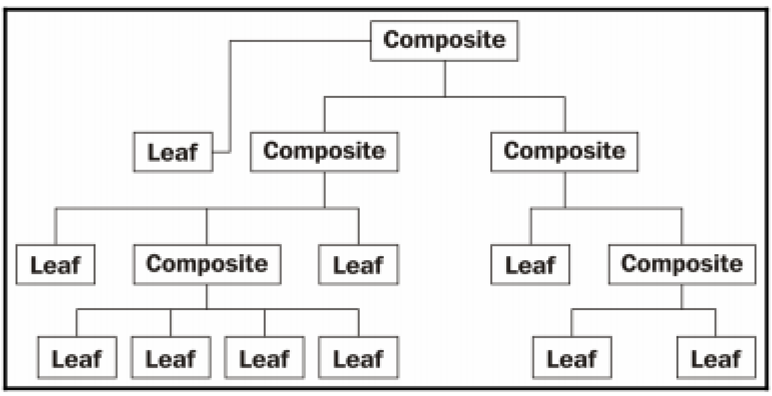
- the trick for the abstract factory is the `__init__.py` in the `localize` packagte can contain logic that redirects all requests to the correct backend
- if we know that the backend is never going to change dynamically without a program restart, we can just put some `if` statements in `__init__.py` that check the current country code, import the stuff we need

0.6 Composite Pattern

- the composite pattern allows complex tree-like structures to be built from simple components
- these components, called composite objects, are able to behave sort of like a container and sort of like a variable, depending on whether they have child components
- composite objects are container objects, where the contents may actually be another composite object
- traditionally, each component in a composite object must be either a leaf node (that cannot contain other objects or a composite node
- the key is that both composite and leaf nodes can have the same interface



- the simple pattern allows us to create arangements of elements all of whihc satisfy the interface of the component object
- the diagram depicts a concrete instance of such a complecated arrangement



- composite pattern is commonly useful in a file/folder like tree
- regardless of whether a node in the tree is a normal file or a folder, it is still subject to operations such as moving, copying or deleting the node
- we can create a component interface that supports these operations, and then use a composite object to represent folders and leaf nodes to represent normal files

- in python we can take advantage of duck typing to implicitly provide the interface

- the composite pattern is useful for a variety of tree-like structures, including GUI widget hierarchies, file hierarchies, tree sets, graphs and HTML DOM