Chapter 01 - The Factory Pattern

September 18, 2021

0.1 Overview

- factories typically come in two forms - the `factory method`, which is a method that returns a different object per input parameter and the `abstract factory`, which is a group of factory methods used to create a family of related objects

0.2 The Factory Method

- the factory method is based on a single function written to handle our object creation taks
- we execute it, passing a parameter that provides information about what we want and, as a result, the object is created

0.2.1 Real-World Examples

- think of factory methods as a plastic toy construction kit
- the molding material used to construct plaatic toys is the same, but different toys can be produced using the right plastic molds
- this is like having a factory method in which the input is the name of the toy that we want (duck or car) and the output (after the modling) is the plastic toy that was requested

0.2.2 Use Cases

- if you realize that you cannot track the objects created by your application because the code that creates them is in may different places instead of a dingle function/method, you should consider using the factory method pattern
- the factory method centralize object creation and tracking your objects becomes much easier
- there is more than one factory method and each factory method logically groups the creation of objects that have similarities
- one factory method might be responsible for conencting you to a different database (`MySQL`, `SQLite`) and another factory method might be responsible for creating the geometrical object that you requested
- fatory method is also useful when you want to decouple object creation from object usage
- we are not coupled/bound to a specific class when creating an object; we just provide partial information about what we want by calling a function
- this means that introducing changes to the function is easy and does not require any changes to the code
- another use case is related to improving performance and memory usage of an application

- a factory method can improve the performance and memory usage by creating new objects only if its absolutely necessary
- when we create objects using a direct class instantiation, extra memeory is allocated every time a new object is created
-

0.2.3 Implementing the factory method

- the two types of data files are `human-readable` or `binary`
- we have some input data stored in an `XML` and a `JSON` file and we want to parse them and retrieve some information
- at the same time we want to centralize the clients conenction to those (and all future) external services
- we will use the factory method to solve the problem

- the `JSON` is a data set containing information about movies
- the `XML` file contains information baout individuals
- we have the global container `persons` anf then their detials as tags
- we can use two libraries to work with `JSON` and `XML`

[2]:
```
import json
import xml.etree.ElementTree as etree
```

- the `JSONDataExtractor` class parses the JSON file and has a `parsed_data()` method that returns all data as a dictionary (`dict`)
- the property decorator is used to make `parsed_data()` appear as a normal attribute instead of a method

[3]:
```
class JSONDataExtractor:
    def __init__(self, filepath):
        self.data = dict()
        with open(filepath, mode='r', encoding='utfc') as f:
            self.data = json.load(f)

        @property
        def parsed_data(self):
            return self.data
```

- the `XMLDataExtractor` class parses the XML file and has a `parsed_data()` method that returns all data as a list of `xml.etree.Element`

[4]:
```
class XMLDataExtractor:
    def __init__(self, filepath):
        self.tree = etree.parse(filepath)

    @property
    def parsed_data(self):
        return self.tree
```

- the `dataextraction_factory()` function is a factory method

- it returns an instance of `JSONDataExtractor` or `XMLDataExtractor` depending on the extension of the input file path as

[7]:
```python
def dataextraction_factory(filepath):
    if filepath.endswith('json'):
        extractor = JSONDataExtractor
    elif filepath.endswith('xml'):
        extractor = XMLDataExtractor
    else:
        raise ValueError('Cannot extract data from {}'.format(filepath))
    return extractor(filepath)
```

- the `extract_data_from()` function is a wrapper of `dataextraction_factory()`
- it adds exception handling as follows

[8]:
```python
def extract_data_from(filepath):
    factory_obj = None
    try:
        factory_obj = dataextraction_factory(filepath)
    except ValueError as e:
        print(e)
    return factory_obj
```

- the `main` function demonstrates how the factory method design pattern can be used
- the first part makes sure that exception handling is effective

[9]:
```python
def main():
    sqlite_factory = extract_data_from('data/person.sq3')
```

[12]:
```python
# json_factory = extract_data_from('data/movies.json')
# xml_factory = extract_data_from('data/person.xml')
```

0.3 Abstract Factory

- the `abstract factory` design pattern is a generalization of the factory method
- basically an `abstract factory` is a (logical) group of factory methods, where each factory method is responsible for generating a different kind of object

0.3.1 Real-World Examples

- abstract factory is used in car manufacturing
- the same machinery is used for stamping the parts (doors, pannels, hoods) of different car models
- the model that is assembled by the machinery is configurable and easy to change at any time

- the `factory_boy` provides an abstract factory implementation for creating Django models in tests
- it is used for creating instances of models that support `test-specific attributes`

0.3.2 Use Cases

- sicne the abstract factory pattern is a generalization of the factory method pattern, it offers the same benefits, it makes tracking an object creation easier
- it decouples object creation from object usage and it gives us the potential to improve the memeory usage and performance of our application
- always start with the `factory method` and if you find your application requiring many `factory methods` its time to switch to the `abstract factory` method
- another importat benefit is that it gives us the ability to modify the behavior of our application dynamically (at runtime) by changing the active factory method

0.3.3 Implementing the Abstract Factory pattern

- imagine we are creating a game or we want to include a mini-game as part of our application to entertain our users
- we want to include at least two games, one for children and one for adults
- we will decide which game to create and launch at runtime, based on user input
- lets start with the kids game, called `FrogWorld`
- the main hero is a frog who enjoys eating bugs
- every hero needs a good name and the name is given by the user at runtime
- the `interact_with()` method is used to describe the interaction of the frog with an obstacle

[15]:
```python
class Frog:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return self.name

    def interact_with(self, obstacle):
        act = obstacle.action()
        msg = f'{self} the Frong encounters {obstacle} and {act}'
        print(msg)
```

- there can be many different kinds of obstacles but for our example, an obstacle can only be a bug
- when the frog encounters a bug, only one action is supported, it eats it

[16]:
```python
class Bug:
    def __str__(self):
        return 'a bug'

    def action(self):
        return 'eats it'
```

- the `FrogWorld` class is an abstract factory
- its main responsibilities are creating the main character and the obstacles in the game

- keeping the creating methods separate and their names generic (for example `make_character()` and `make_obstacle()` allows us to change the active factory (and therefore the active game) dynamically without any code changes
- in a statically typed language, the abstract factory would be an abstract class/interface with empty methods, but in Python, this is not required because the types are checked at runtime

[ ]:
```
class FrogWorld:
    def __init__(self, name):
        print(self)
        self.player_name = name

    def __str__(self):
        return '\n\n\t-------- Frog World -------'

    def make_character(self):
        return Frog(self.player_name)

    def make_obstacle(self):
        return Bug()
```

- the `WizardWorld` game is similar
- the only difference is that the wizard battles against monsters such as `orks` instead of eating bugs

[18]:
```
class Wizard:
    def __init__(self, name):
        self.name = name

    def __str__(self):
        return self.name

    def interact_with(self, obstacle):
        act = obstacle.action()
        msg = f'{self} the Wizard battles against {obstacle} and {act}!'
        print(msg)

class Ork:
    def __str__(self):
        return 'an evil ork'

    def action(self):
        return 'kills it'
```

- we also need to define the `WizardWorld` class similar to the `FrogWorld` one that we have discussed

[ ]:
```
class WizardWorld:
    def __init__(self, name):
        print(self)
        self.player_name = name

    def __str__(self):
        return '\n\n\t------- Wizard World -------'

    def make_character(self):
        return Wizard(self.player_name)

    def make_obstacle(self):
        return Ork()
```

- the GameEnvironment class is the main entry point of our game
- it accepts the factory as an input and uses it to create the world of the game
- the play() method initates the interaction betweeen the created hero and the obstacle

[19]:
```
class GameEnvironment:
    def __init__(self, factory):
        self.hero = factory.make_character()
        self.obstacle = factory.make_obstacle()

    def play(self):
        self.hero.interact_with(self.obstacle)
```

- the validate_age() function prompts the user to give a valid age
- if the age is not valid, it returns a tuple with the first element set to False
- if the age is fine, the first element of the tuple is set to True and thats the case where we actually care about the second element of the tuple, which is the age given by the user

[21]:
```
def validate_age(name):
    try:
        age = input(f'Welcome {name}. How old are you? ')
        age = int(age)
    except ValueError as err:
        print(f"Age {age} is invalid, please try again...")
        return (False, age)
        return (True, age)
```

- finally, we need the main() function
- it asks for the user's name and age, and decides which game should be palyed, given the age of the user, as follows

[ ]:
```
def main():
    name = input('Hello. Whats your name? ')
    valid_input = False
    while not valid_input:
        valid_input, agae = validate_age(name)
    game = FrogWorld if age < 18 else WizardWorld
    environment = GameEnvironment(game(name))
    environment.play()
```