The Factory Method Pattern and Abstract Factory Pattern

Intermediate Application Development

Otago Polytechnic
Dunedin, New Zealand
Kaiako: Tom Clark

Introduction: Factory Method

I always used to hate this pattern. Most descriptions I've seen for it are confusing. The UML diagram is confusing. It always seemed to me like it was making things *more* complicated.

"Define an interface for creating an object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses."
(*GoF*)
That didn't really help. What we need is a good example.

Our Problem

I work on games that are available around the world. Naturally, we want to be able to present those games to players in the preferred languages. We don't know what language we're going to use until runtime. Clearly we need some sort of translator class.

Let's Do it Wrong

```
class Translator:
    def localise(self, lang, expression):
        if lang == 'en':
            # load up English language stuff
            # look up expression
            # return English version
        elif lang == 'es':
            # load up Spanish language stuff
            # look up expression
            # return Spanish version
        ...
```

You see where this is going. What's wrong with this?

What's Wrong?

- What's the process for adding a language? (Open-closed)
- What if we change the "`load up X language stuff`" process? (Single responsibility)

Improved Version

```
class Translator:
    def localise(self, lang, expression):
        translator = self.get_translator(lang)
        return translator.interpret(expression)

# using the Translator
weather = 'WEATHER.CLOUDY'
localiser = Translator()
print(localiser.localise(player.preferred_language, weather))
```

Get Translator Method

How do we implement `get_translator()`?

```
def get_translator(self, lang):
    if lang == 'en':
        return English()
    elif lang == 'es':
        return Spanish()
    ...
```

Not bad, but it could be better.

A better process

- Set up directories. like `localisations/en` and `localisations/es`.
- Each directory contains files, like `weather`, with tables to look up expressions and their localised values.
- Now `get_translator()` looks for a directory and loads up its files.
- Also, switching from one strategy to the other is just a matter of changing the `get_translator()` method.

Taking it further

Suppose my employer gets bought out by Disney. Our new corporate overlords insist that every game include a Jar Jar Binks option where characters speak with Jar Jar's accent thing in whatever language.

Suppose also that we come up with an algorithm to take test in an arbitrary language and transform it into the Jar Jar Binks accent.

Subclassing Translator

```
class JarJarTranslator(Translator):
    def get_translator(self, lang):
        # This would load up the language info and
        # somehow JarJar-ise it? I never said it was
        # a good idea.
```

Notice that we don't have to override the `localise()` method. We just change the factory method `get_translator()`.

Programming Activity

1. Pull the course materials repo.
2. Create a new branch, `14-practical` in your practicals repo.
3. Add a subdirectory, `14-practical` and copy `14-practical.ipynb` from the class materials into it.
4. Open a shell, cd to this directory, and run `jupyter notebook` to open the notebook. Complete the first two questions.
5. We will discuss results in 20ish minutes.

Introduction: Abstract Factory

The Abstract Factory may be thought of as related to the Factory Method and is sort of is, but it's not a very strong relationship. They are both *Creational Patterns*, but that may be as far as it goes.

"Provide an interface for creating **families** of related or dependent objects without specifying their concrete classes"
(*GoF*)

The bit about families of objects is the key. With the Factory Method we wanted one object and we weren't too particular about its concrete type. With the Abstract Factory we want families of related objects.

Some Abstract Classes

```
from abc import ABC, abstractmethod

class PetFactory(ABC):

    @abstractmethod
    def get_animal(self):
        pass

    @abstractmethod
    def get_pet_food(self):
        pass

    @abstractmethod
    def get_pet_toy(self):
        pass
```

More Classes

```
class Animal(ABC):
    pass

class Dog(Animal):
    pass

class PetFood(ABC):
    pass

class DogRoll(PetFood):
    pass

class PetToy(ABC):
    pass

class Ball(PetToy):
    pass
```

And Now a Concrete Factory

```
class DogFactory(PetFactory):

    def get_animal(self):
        return Dog()

    def get_pet_food(self):
        return DogRoll()

    def get_pet_toy(self):
        return Ball()
```

Using the Factory

```
def give_a_pet_to(person, factory):
    pet = {'animal': factory.get_animal(),
           'food': factory.get_pet_food(),
           'toy': factory.get_pet_toy()
          }
    person.add_pet(pet)

factory = DogFactory()
# I have a dog named Star
give_a_pet_to(tom, factory)
```

I also have a cat named Lola. Can you see how we would implement cats?

Some Closing Notes

In Python and similar dynamic languages we might not bother with the abstract classes. They make the design a just little more clear.

Similarly, both of these patterns are seen less often in a language like Python than in some other languages, in particular because classes and functions are first-class values and this removes some of the need for these patterns.

Finally, a technique called *dependency injection* can serve some of the same purposes, especially in the case of the Factory Method. We won't cover dependency injection in this paper, however.