# Factory design pattern



## Key terms

### Simple factory

> A simple factory is a static method that returns an instance of a class. It is a static method because it does not need to be instantiated. It is a factory because it creates an instance of a class.

### Factory method

> Rather than having a single static method that returns an instance of a class, the factory method pattern uses a class that has a method that returns an instance of a class. This method is not static because it needs to be instantiated.

### Abstract factory

> The abstract factory pattern is a factory of factories. It is a factory that creates other factories. It is a factory that creates other factories that create instances of classes.

# Factory

> The factory method pattern is a creational pattern that uses factory methods to deal with the problem of creating objects without having to specify the exact class of the object that will be created. This is done by creating objects by calling a factory method—either specified in an interface and implemented by child classes, or implemented in a base class and optionally overridden by derived classes—rather than by calling a constructor.

In our previous session, we learnt how to use the prototype to create a clone of the object. One of the motivations for using the prototype is to create a new object without having to know the exact class of the object that will be created. For an instance, there is an external library that we want to use in our application. We don't know the exact class of the object that will be created. We just know that the object will have a method called `doSomething()`. We can use the prototype to create a new object without having to know the exact class of the object that will be created. The library will provide us with a prototype object that we can use to create a new object. But if the library does not expose the prototype object, we will create a prototype object ourselves and use it to create a new object.

```
prototype: User = User("Tantia", "Tope")
user: User = prototype.clone()
```

In the above example, the client code is still not completely independent of the class of the object that it is creating. The client code still has to call the `new` keyword to create the prototype of the object. The client code also has to know the class of the object that it is creating. This is not ideal as the client code should not have to know the class of the object that it is creating. The client code should only know the interface of the object that it is creating. This is where the factory pattern comes into play.

If we want to just change the name of the class in our next version, the client code will have to be changed making our code backward incompatible. To avoid this, we can use the factory pattern. The factory pattern allows us to create objects without specifying the exact class of the object that will be created. The client code can request an object from a factory object without having to know the class of the object that will be returned. The factory object can create the object and return it to the client code.

## Simple Factory

> The simple factory pattern is a creational pattern that provides a static method for creating objects. The method can be used to create objects without having to specify the exact class of the object that will be created. This is done by creating a factory class that contains a static method for creating objects.

Let us create a simple factory class that can be used to create different types of users. The factory class will have a static method that can be used to create different types of users.

```
class UserFactory:
    @staticmethod
    def create_user(role) -> User:
        if role == UserRole.STUDENT:
            return Student("Tantia", "Tope")
        elif role == UserRole.TEACHER:
            return Teacher("Tantia", "Tope")
        elif role == UserRole.ADMIN:
            return Admin("Tantia", "Tope")
```

The client code can request a user object from the factory class without having to know the class of the object that will be returned.

```
user: User = UserFactory.create_user(UserRole.STUDENT)
```

The complete steps to implement the simple factory pattern are:

1. `Factory class` - Create a factory class that contains a static method for creating objects.
2. `Conditional` - Use a conditional statement to create the object based on the input.
3. `Request` - Request an object from the factory class without having to know the class of the object that will be returned.

## Factory Method

The simple factory method is easy to implement, but it has a few drawbacks. The factory class is not extensible. If we want to add a new type of user, we will have to modify the factory class. Also, the factory class is not reusable. If we want to create a factory for creating different types of objects, we will have to create a new factory class. To overcome these drawbacks, we can use the factory method pattern.

In the factory method the responsibility of creating the object is shifted to the child classes. The factory method is implemented in the base class and the child classes can override the factory method to create objects of their own type. The factory method is also known as the virtual constructor.

```
from abc import ABC, abstractmethod

class UserFactory(ABC):
    @abstractmethod
    def create_user(self, first_name, last_name) -> User:
        pass

class StudentFactory(UserFactory):
    def create_user(self, first_name, last_name) -> Student:
        return Student(first_name, last_name)

class TeacherFactory(UserFactory):
    def create_user(self, first_name, last_name) -> Teacher:
        return Teacher(first_name, last_name)
```

The client code can request a user object from the base class without having to know the class of the object that will be returned.

```
factory: UserFactory = TeacherFactory()
user: User = factory.createUser("Tantia", "Tope")
```

The complete steps to implement the factory method pattern are:

1. `Base factory interface` - Create a factory class that contains a method for creating objects.
2. `Child factory class` - Create a child class that extends the base factory class and overrides the factory method to create objects of its own type.
3. `Request` - Request an object from the factory class without having to know the class of the object that will be returned.

## Abstract Factory

> The abstract factory pattern is a creational pattern that provides an interface for creating families of related or dependent objects without specifying their concrete classes.

Let us take the example of a classroom. We have already created a `User` abstract class. Now we will create the concrete classes `Student` and `Teacher`. To restrict the usage of subclasses, we can create factories for each of the concrete classes.

The `StudentFactory` will be used to create `Student` objects and the `TeacherFactory` will be used to create `Teacher` objects.

```
class StudentFactory(UserFactory):
    def create_user(self, first_name, last_name) -> Student:
        return Student(first_name, last_name)

class TeacherFactory(UserFactory):
    def create_user(self, first_name, last_name) -> Teacher:
        return Teacher(first_name, last_name)
```

So now in order to create a classroom, we can use the respective factories to create the objects.

```
student_factory: UserFactory = StudentFactory()
student: User = student_factory.create_user("Tantia", "Tope")

teacher_factory: UserFactory = TeacherFactory()
teacher: User = teacher_factory.create_user("Tantia", "Tope")
```

But now we have a problem, we can use the factories to create any type of student and teacher. Should a teacher teaching Physics be able to teach a student of Biology class? This is where the concept of related or a family of objects comes into play. The `Student` and `Teacher` objects are related to each other. A teacher should only be able to teach a student of the same class. So we can create a factory that can create a family of related objects. The `ClassroomFactory` will be used to create `Student` and `Teacher` objects of the same class.

```
from abc import ABC, abstractmethod

class ClassroomFactory(ABC):
    @abstractmethod
    def create_student(self, first_name, last_name) -> Student:
        pass

    @abstractmethod
    def create_teacher(self, first_name, last_name) -> Teacher:
        pass
```

Now we can create concrete factories for each family of related objects that we want to create.

```
class BiologyClassroomFactory(ClassroomFactory):
    def create_student(self, first_name, last_name) -> Student:
        return BiologyStudent(first_name, last_name)

    def create_teacher(self, first_name, last_name) -> Teacher:
        return BiologyTeacher(first_name, last_name)
```

The class `ClassroomFactory` is an abstract class that contains the factory methods for creating the objects. The child classes can override the factory methods to create objects of their own type. The client code can request an object from the factory class without having to know the class of the object that will be returned.

```
factory: ClassroomFactory = BiologyClassroomFactory()
student: Student = factory.create_student("Tantia", "Tope")
teacher: Teacher = factory.create_teacher("Tantia", "Tope")
```

The class `ClassroomFactory` becomes our abstract factory that essentially is a factory of factories.

### Advantages of Abstract Factory

- `Isolate concrete classes` - The client code is not coupled to the concrete classes of the objects that it creates.
- `Easy to exchange product families` - The client code can request an object from the factory class without having to know the class of the object that will be returned. This makes it easy to exchange product families.
- `Promotes consistency among products` - The client code can request an object from the factory class without having to know the class of the object that will be returned. This makes it easy to maintain consistency among products.

### Implementation

1. `Abstract product interface` - Create an interface for the products that will be created by the factory.

```
class Button(ABC):
    @abstractmethod
    def render(self) -> None:
        pass

    @abstractmethod
    def on_click(self) -> None:
        pass
```

2. `Concrete products` - Create concrete classes that implement the product interface.

```
class RoundedButton(Button):
    def render(self) -> None:
        print("Rendered rounded button")

    def on_click(self) -> None:
        print("Clicked rounded button")
```

3. `Abstract factory interface` - Create an interface for the abstract factory that will be used to create the products.

```
class FormFactory(ABC):
    @abstractmethod
    def create_button(self) -> Button:
        pass
```

4. `Concrete factories` - Create concrete classes that implement the abstract factory interface.

```
class RoundedFormFactory(FormFactory):
    def create_button(self) -> Button:
        return RoundedButton()
```

5. `Client code` - Request an object from the factory class without having to know the class of the object that will be returned.

```
factory: FormFactory = RoundedFormFactory()
button: Button = factory.create_button()
```

## Recap

- The factory pattern is a creational design pattern that can be used to create objects without having to specify the exact class of the object that will be created.
- It reduces the coupling between the client code and the class of the object that it is creating.
- Simple factory - The factory class contains a static method for creating objects. This technique is easy to implement, but it is not extensible and reusable. It violates the open-closed principle and the single responsibility principle.
- Factory method - The responsibility of creating the object is shifted to the child classes. The factory method is implemented in the base class and the child

classes can override the factory method to create objects of their own type. This technique is extensible and reusable. It follows the open-closed principle and the single responsibility principle.