Builder design pattern

Key terms

Builder

> Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

Builder

Problems

- `Complex object creation` - There are multiple ways to create an object, but constructors are the primary technique used for creating instances of a class. However, constructors become unmanageable when there is a need to create an object with many parameters. This is known as the telescoping constructor anti-pattern. The telescoping constructor anti-pattern is a code smell that indicates that the class has too many constructors. This is a code smell because it is difficult to maintain and extend the class.
- `Validation and failing object creation` - There are cases when you want to validate the parameters before creating an object. For example, you might want to validate the parameters before creating a database connection. If the parameters are invalid, you might want to throw an exception. However, if we use the default constructor, we cannot fail object creation.
- `Immutability` - Mutable objects are objects whose state can be changed after they are created. Immutable objects are objects whose state cannot be changed after they

are created. Immutable objects are easier to maintain and extend whereas mutable objects can lead to bugs. However, if we use the default constructor, we cannot create immutable objects.

Constructor with a hash map

The above problems can be solved using a constructor with a hash map. The constructor will take a hash map as a parameter. The hash map will contain the parameters and their values. The constructor will validate the parameters and create the object.

```
class Database:
    def __init__(self, config):
        self.host = config.get("host")
        self.port = config.get("port")
        self.username = config.get("username")
        self.password = config.get("password")
```

Some problems with the above code are:

- `Type safety` - Python is a dynamically typed language. This means that the type of a variable is inferred at runtime. This allows us to pass any type of value to the constructor. However, it could lead to bugs. For example, if we pass a string instead of an integer, the code will not fail until we try to use the port variable.
- `Defined parameters` - With the above approach, identifying the parameters is difficult. We need to read the code to identify the parameters. This is not a good approach because it is difficult to maintain and extend the code.

Named parameters

In Python, you can use named parameters to solve the above problems. Named parameters allow you to pass parameters to a function using the parameter name. This enables us to pass any permutation of parameters to the constructor and out of order.

```
class Database:
    def __init__(self, host, port, username, password):
        self.host = host
        self.port = port
        self.username = username
        self.password = password

database = Database(
    host="localhost",
    port=3306,
)
```

While this approach solves the problems associated with passing a lot of unnecessary parameters and ensuring they are passed in order, however it requires you to put business logic in the constructor. This is not a good approach because it is difficult to maintain and extend the code. For example, if you would like to validate if the host is reachable or not before creating the object, you would have to add the validation logic to the constructor. Putting business logic in the constructor is an anti-pattern since it makes the code difficult to maintain and extend.

```
class Database:
    def __init__(self, host, port, username, password):
        if not self.is_host_reachable(host):
            raise Exception("Host is not reachable")
        self.host = host
        self.port = port
        self.username = username
        self.password = password

    def is_host_reachable(self, host):
        # Check if host is reachable
        return True
```

Inner class

The builder pattern addresses this issue by stating

> Separate the construction of a complex object from its representation so that the same construction process can create different representations.

This essentially means decoupling the construction of a complex object from its representation. The builder pattern is a creational design pattern that lets you construct complex objects, step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

Step 1 - Add an inner builder class

```
@dataclass
class Database:
    host: str
    port: int
    username: str
    password: str

    class Builder:
        def __init__(self):
            self.host = None
            self.port = None
            self.username = None
            self.password = None
```

In the above code, we have created a builder class inside the Database class. The builder class has the same parameters as the Database class. This allows us to modify the inner class till we are ready to create the object.

You can add a utility method to the outer class so that the developer can create the builder class instance using the outer class.

```
@dataclass
class Database:
    host: str
    port: int
    username: str
    password: str


    @staticmethod
    def builder():
        return Builder()

    class Builder:
        def __init__(self):
            self.host = None
            self.port = None
            self.username = None
            self.password = None
```

Now when you call the builder method, it will return the builder class instance. You can use this instance to set the parameters and build the object.

Step 2 - Add the setters

As mentioned, the idea with the builder pattern is to separate the construction of a complex object from its representation. We have already created the builder class. We will use this class to set the parameters. Once the parameters are set, you can get the object of the outer class. Let's add the setters to the builder class.

```
@dataclass
class Database:
    host: str
    port: int
    username: str
    password: str


    @staticmethod
    def builder():
        return Builder()

    class Builder:
        def __init__(self):
            self.host = None
            self.port = None
            self.username = None
            self.password = None

        def host(self, host):
            self.host = host

        def port(self, port):
            self.port = port

        def username(self, username):
            self.username = username

        def password(self, password):
            self.password = password
```

The above code can be used to set parameters as follows:

```
builder = Database.builder()
builder.host("localhost")
builder.port(3306)
builder.username("root")
builder.password("password")
```

To improve the readability and usability of the code, you can return the builder class instance from the setter methods.
This will allow you to chain the setter methods.

```
@dataclass
class Database:
    host: str
    port: int
    username: str
    password: str


    @staticmethod
    def builder():
        return Builder()

    class Builder:
        def __init__(self):
            self.host = None
            self.port = None
            self.username = None
            self.password = None


        def host(self, host) -> Builder:
            self.host = host
            return self

        def port(self, port) -> Builder:
            self.port = port
            return self

        def username(self, username) -> Builder:
            self.username = username
            return self

        def password(self, password) -> Builder:
            self.password = password
            return self
```

Now you can chain the setter methods as follows:

```
builder = Database.builder()
            .host("localhost")
            .port(3306)
            .username("root")
            .password("password")
```

This makes our code more readable and usable.

Step 3 - Add the build method

Once we have set the parameters, we need to create the object. We will add a build method to the builder class. This method will create the object of the outer class.

```
@dataclass
class Database:
    host: str
    port: int
    username: str
    password: str


    @staticmethod
    def builder():
        return Builder()

    class Builder:

        ...

        def build(self) -> Database:
            return Database(
                host=self.host,
                port=self.port,
                username=self.username,
                password=self.password
            )
```

The build method can be thought as of a lifecycle method or a hook that should be called at the end of the builder chain. The build method will return the object of the outer class with the parameters set. You can now also add validation logic to the build method.

```
@dataclass
class Database:
    host: str
    port: int
    username: str
    password: str


    @staticmethod
    def builder():
        return Builder()

    class Builder:
        def __init__(self):
            self.host = None
            self.port = None
            self.username = None
            self.password = None


        ...

        def build(self) -> Database:

            self.validate()

            return Database(
                host=self.host,
                port=self.port,
                username=self.username,
                password=self.password
            )

        def validate(self) -> None:
            if not self.is_host_reachable(self.host):
                raise Exception("Host is not reachable")
```

Now you can create the object as follows:

```
database = Database.builder()
    .host("localhost")
    .port(3306)
    .username("root")
    .password("password")
    .build()
```

With the above code, we have solved the problems associated with creating complex objects. We can now create objects with any permutation of parameters. We can also validate the parameters and fail object creation and all of this without putting business

logic in the constructor.

Final code

```
@dataclass
class Database:
    host: str
    port: int
    username: str
    password: str


    @staticmethod
    def builder():
        return Builder()

    class Builder:
        def __init__(self):
            self._host = None
            self._port = None
            self._username = None
            self._password = None


        def host(self, host: str) -> Builder:
            self._host = host
            return self

        def port(self, port: int) -> Builder:
            self._port = port
            return self

        def username(self, username: str) -> Builder:
            self._username = username
            return self

        def password(self, password: str) -> Builder:
            self._password = password
            return self

        def build(self) -> Database:

            self.validate()

            return Database(
                host=self._host,
                port=self._port,
                username=self._username,
                password=self._password
            )

        def validate(self) -> None:
```

```
            if not self.is_host_reachable(self._host):
                raise Exception("Host is not reachable")
```

🖥️ Code (https://github.com/scaleracademy/lld-python/blob/main/design-patterns/src/creational/builder/database.py)

Summary

- The builder pattern is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.
- Use cases of builder pattern
  - Complex object creation - Telescoping constructor anti-pattern
  - Validation and failing object creation
  - Immutability
- Add an inner class to separate the business logic from the construction.
- Add a static method to the outer class to return the builder class instance.
- Implement chained setter methods in the builder class. These methods will set the parameter value and return the builder class instance.
- Implement the `build()` method in the builder class. This method will return the outer class object with the parameters set.

Reading list

- Telescoping constructor anti-pattern (https://www.vojtechruzicka.com/avoid-telescoping-constructor-pattern/)
- Why objects should be immutable? (https://octoperf.com/blog/2016/04/07/why-objects-must-be-immutable)
- Builder Pattern (https://python-patterns.guide/gang-of-four/builder/)