The Factory Method Pattern and Its Implementation in Python

by Isaac Rodriguez 27 Comments **best-practices** **intermediate**

Mark as Completed

Tweet Share Email

Table of Contents

This article explores the Factory Method design pattern and its implementation in Python. Design patterns became a popular topic in late 90s after the so
and Vlissides) published their book Design Patterns: Elements of Reusable Object-Oriented Software.

The book describes design patterns as a core design solution to reoccurring problems in software and classifies each design pattern into categories acco
given a name, a problem description, a design solution, and an explanation of the consequences of using it.

The GoF book describes Factory Method as a creational design pattern. Creational design patterns are related to the creation of objects, and Factory Met
common interface.

This is a recurrent problem that **makes Factory Method one of the most widely used design patterns**, and it's very important to understand it and kno

By the end of this article, you will:

- Understand the components of Factory Method
- Recognize opportunities to use Factory Method in your applications
- Learn to modify existing code and improve its design by using the pattern
- Learn to identify opportunities where Factory Method is the appropriate design pattern
- Choose an appropriate implementation of Factory Method
- Know how to implement a reusable, general purpose solution of Factory Method

Introducing Factory Method

Factory Method is a creational design pattern used to create concrete implementations of a common interface.

It separates the process of creating an object from the code that depends on the interface of the object.

For example, an application requires an object with a specific interface to perform its tasks. The concrete implementation of the interface is identified by s

Instead of using a complex `if/elif/else` conditional structure to determine the concrete implementation, the application delegates that decision to a sep
With this approach, the application code is simplified, making it more reusable and easier to maintain.

Imagine an application that needs to convert a `Song` object into its `string` representation using a specified format. Converting an object to a different repr
see these requirements implemented in a single function or method that contains all the logic and implementation, like in the following code:

The `.serialize()` method supports two different formats: JSON and XML. Any other `format` specified is not supported, so a `ValueError` exception is raised.

Let's use the Python interactive shell to see how the code works:

```
>>
>>
>>

>>
'{

>>
'<

>>
Tr



Va
```

You create a `song` object and a `serializer`, and you convert the song to its string representation by using the `.serialize()` method. The method takes the
representing the format you want. The last call uses `YAML` as the format, which is not supported by the `serializer`, so a `ValueError` exception is raised.

This example is short and simplified, but it still has a lot of complexity. There are three logical or execution paths depending on the value of the `format` pa
you've probably seen code with more complexity than this, but the above example is still pretty hard to maintain.

Remove ads

The Problems With Complex Conditional Code

The example above exhibits all the problems you'll find in complex logical code. Complex logical code uses `if/elif/else` structures to change the behavio
structures makes the code harder to read, harder to understand, and harder to maintain.

The code above might not seem hard to read or understand, but wait till you see the final code in this section!

Nevertheless, the code above is hard to maintain because it is doing too much. The single responsibility principle states that a module, a class, or even a
responsibility. It should do just one thing and have only one reason to change.

The `.serialize()` method in `SongSerializer` will require changes for many different reasons. This increases the risk of introducing new defects or breaking
Let's take a look at all the situations that will require modifications to the implementation:

- **When a new format is introduced:** The method will have to change to implement the serialization to that format.
- **When the `Song` object changes:** Adding or removing properties to the `Song` class will require the implementation to change in order to accommod
- **When the string representation for a format changes (plain JSON vs JSON API):** The `.serialize()` method will have to change if the desired st
  the representation is hard-coded in the `.serialize()` method implementation.

The ideal situation would be if any of those changes in requirements could be implemented without changing the `.serialize()` method. Let's see how yo

Looking for a Common Interface

The first step when you see complex conditional code in an application is to identify the common goal of each of the execution paths (or logical paths).

Code that uses `if/elif/else` usually has a common goal that is implemented in different ways in each logical path. The code above converts a `song` object
format in each logical path.

Based on the goal, you look for a common interface that can be used to replace each of the paths. The example above requires an interface that takes a `s`

Once you have a common interface, you provide separate implementations for each logical path. In the example above, you will provide an implementati

Then, you provide a separate component that decides the concrete implementation to use based on the specified `format`. This component evaluates the v
implementation identified by its value.

In the following sections, you will learn how to make changes to existing code without changing the behavior. This is referred to as refactoring the code.

Martin Fowler in his book Refactoring: Improving the Design of Existing Code defines refactoring as "the process of changing a software system in such a
the code yet improves its internal structure."

```
cl
```

The new version of the code is easier to read and understand, but it can still be improved with a basic implementation of Factory Method.

Remove ads

Basic Implementation of Factory Method

The central idea in Factory Method is to provide a separate component with the responsibility to decide which concrete implementation should be used b
parameter in our example is the `format`.

To complete the implementation of Factory Method, you add a new method `._get_serializer()` that takes the desired `format`. This method evaluates the v
serialization function:

```
cl
```

Note: The `._get_serializer()` method does not call the concrete implementation, and it just returns the function object itself.

Now, you can change the `.serialize()` method of `SongSerializer` to use `._get_serializer()` to complete the Factory Method implementation. The next exa

```
cl
```

```
cl



de





de





de
```

Note: The `.serialize()` method in `SongSerializer` does not use the `self` parameter.

The rule above tells us it should not be part of the class. This is correct, but you are dealing with existing code.

If you remove `SongSerializer` and change the `.serialize()` method to a function, then you'll have to change all the locations in the application that use
function.

Unless you have a very high percentage of code coverage with your unit tests, this is not a change that you should be doing.

The mechanics of Factory Method are always the same. A client (`SongSerializer.serialize()`) depends on a concrete implementation of an interface. It rec
component (`get_serializer()`) using some sort of identifier (`format`).

The creator returns the concrete implementation according to the value of the parameter to the client, and the client uses the provided object to complet

You can execute the same set of instructions in the Python interactive interpreter to verify that the application behavior has not changed:

```
>>
>>
>>

>>
'{

>>
'<

>>
Tr




Va
```

You create a `song` and a `serializer`, and use the `serializer` to convert the song to its `string` representation specifying a `format`. Since, `YAML` is not a support

An Object Serialization Example

The basic requirements for the example above are that you want to serialize `Song` objects into their `string` representation. It seems the application provide
the application will need to serialize other type of objects like `Playlist` or `Album`.

Ideally, the design should support adding serialization for new objects by implementing new classes without requiring changes to the existing implement
serialized to multiple formats like JSON and XML, so it seems natural to define an interface `Serializer` that can have multiple implementations, one per fo

The interface implementation might look something like this:

```
#

im
im

cl
```

```
cl
```

Note: The example above doesn't implement a full `Serializer` interface, but it should be good enough for our purposes and to demonstrate Factory M

The `Serializer` interface is an abstract concept due to the dynamic nature of the Python language. Static languages like Java or C# require that interfaces
provides the desired methods or functions is said to implement the interface. The example defines the `Serializer` interface to be an object that implemen

- `.start_object(object_name, object_id)`
- `.add_property(name, value)`
- `.to_str()`

This interface is implemented by the concrete classes `JsonSerializer` and `XmlSerializer`.

The original example used a `SongSerializer` class. For the new application, you will implement something more generic, like `ObjectSerializer`:

```
#

cl
```

h

In the original example, you implemented the creator as a function. Functions are fine for very simple examples, but they don't provide too much flexibilit

Classes can provide additional interfaces to add functionality, and they can be derived to customize behavior. Unless you have a very basic creator that w implement it as a class and not a function. These type of classes are called object factories.

You can see the basic interface of `SerializerFactory` in the implementation of `ObjectSerializer.serialize()`. The method uses `factory.get_serializer(form` factory.

You will now implement `SerializerFactory` to meet this interface:

```
#

cl








fa
```

The current implementation of `.get_serializer()` is the same you used in the original example. The method evaluates the value of `format` and decides the a relatively simple solution that allows us to verify the functionality of all the Factory Method components.

Let's go to the Python interactive interpreter and see how it works:

```
>>
>>
>>
>>

>>
'{

>>
'<

>>
Tr





Va
```

The new design of Factory Method allows the application to introduce new features by adding new classes, as opposed to changing existing ones. You ca the `Serializable` interface on them. You can support new formats by implementing the `Serializer` interface in another class.

The missing piece is that `SerializerFactory` has to change to include the support for new formats. This problem is easily solved with the new design becau

Remove ads

Supporting Additional Formats

The current implementation of `SerializerFactory` needs to be changed when a new format is introduced. Your application might never need to support a

You want your designs to be flexible, and as you will see, supporting additional formats without changing `SerializerFactory` is relatively easy.

The idea is to provide a method in `SerializerFactory` that registers a new `Serializer` implementation for the format we want to support:

```
#

cl
```

```
#

im
im

cl





se
```

Note: To implement the example, you need to install `PyYAML` in your environment using `pip install PyYAML`.

JSON and YAML are very similar formats, so you can reuse most of the implementation of `JsonSerializer` and overwrite `.to_str()` to complete the implem
the `factory` object to make it available.

Let's use the Python interactive interpreter to see the results:

```
>>
>>
>>
>>
>>

>>
{"

>>
<s

>>
{a
```

By implementing Factory Method using an Object Factory and providing a registration interface, you are able to support new formats without changing a
the risk of breaking existing features or introducing subtle bugs.

A General Purpose Object Factory

The implementation of `SerializerFactory` is a huge improvement from the original example. It provides great flexibility to support new formats and avoid

Still, the current implementation is specifically targeted to the serialization problem above, and it is not reusable in other contexts.

Factory Method can be used to solve a wide range of problems. An Object Factory gives additional flexibility to the design when requirements change. Id
Factory that can be reused in any situation without replicating the implementation.

There are some challenges to providing a general purpose implementation of Object Factory, and in the following sections you will look at those challeng
any situation.

Not All Objects Can Be Created Equal

The biggest challenge to implement a general purpose Object Factory is that not all objects are created in the same way.

Not all situations allow us to use a default `.__init__()` to create and initialize the objects. It is important that the creator, in this case the Object Factory, r

This is important because if it doesn't, then the client will have to complete the initialization and use complex conditional code to fully initialize the provic
Method design pattern.

To understand the complexities of a general purpose solution, let's take a look at a different problem. Let's say an application wants to integrate with diffe
to the application or internal in order to support a local music collection. Each of the services has a different set of requirements.

Note: The requirements I define for the example are for illustration purposes and do not reflect the real requirements you will have to implement to in

The intent is to provide a different set of requirements that shows the challenges of implementing a general purpose Object Factory.

```
#

co




}
```

The `config` dictionary contains all the values required to initialize each of the services. The next step is to define an interface that will use those values to c
service. That interface will be implemented in a `Builder`.

Let’s look at the implementation of the `SpotifyService` and `SpotifyServiceBuilder`:

```
#

cl





cl
```

Note: The music service interface defines a `.test_connection()` method, which should be enough for demonstration purposes.

The example shows a `SpotifyServiceBuilder` that implements `.__call__(spotify_client_key, spotify_client_secret, **_ignored)`.

This method is used to create and initialize the concrete `SpotifyService`. It specifies the required parameters and ignores any additional parameters provid
retrieved, it creates and returns the `SpotifyService` instance.

Notice that `SpotifyServiceBuilder` keeps the service instance around and only creates a new one the first time the service is requested. This avoids going t
specified in the requirements.

Let’s do the same for Pandora:

```
#

cl






cl
```

A Generic Interface to Object Factory

A general purpose Object Factory (`ObjectFactory`) can leverage the generic `Builder` interface to create all kinds of objects. It provides a method to register
create the concrete object instances based on the `key`.

Let's look at the implementation of our generic `ObjectFactory`:

```
#

cl
```

The implementation structure of `ObjectFactory` is the same you saw in `SerializerFactory`.

The difference is in the interface that exposes to support creating any type of object. The builder parameter can be any object that implements the callab
a class, or an object that implements `.__call__()`.

The `.create()` method requires that additional arguments are specified as keyword arguments. This allows the `Builder` objects to specify the parameters t
For example, you can see that `create_local_music_service()` specifies a `local_music_location` parameter and ignores the rest.

Let's create the factory instance and register the builders for the services you want to support:

```
#
im

#

fa
fa
fa
fa
```

The `music` module exposes the `ObjectFactory` instance through the `factory` attribute. Then, the builders are registered with the instance. For Spotify and Pa
corresponding builder, but for the local service, you just pass the function.

Let's write a small program that demonstrates the functionality:

```
#
im

co





}

pa
pa

sp
sp

lo
lo

pa
pr
```

A good solution is to specialize a general purpose implementation to provide an interface that is concrete to the application context. In this section, you v
music services, so the application code communicates the intent better and becomes more readable.

The following example shows how to specialize `ObjectFactory`, providing an explicit interface to the context of the application:

```
#

cl




se
se
se
se
```

You derive `MusicServiceProvider` from `ObjectFactory` and expose a new method `.get(service_id, **kwargs)`.

This method invokes the generic `.create(key, **kwargs)`, so the behavior remains the same, but the code reads better in the context of our application. Yo
to `services` and initialized it as a `MusicServiceProvider`.

As you can see, the updated application code reads much better now:

```
im

co





}

pa
pa
sp
sp
lo
lo

pa
pr

sp
pr
```

Running the program shows that the behavior hasn't changed:

```
$
Ac
Ac
Ac
id
id
```

Conclusion

Factory Method is a widely used, creational design pattern that can be used in many situations where multiple concrete implementations of an interface e

The pattern removes complex logical code that is hard to maintain, and replaces it with a design that is reusable and extensible. The pattern avoids modif

This is important because changing existing code can introduce changes in behavior or subtle bugs.

In this article, you learned: