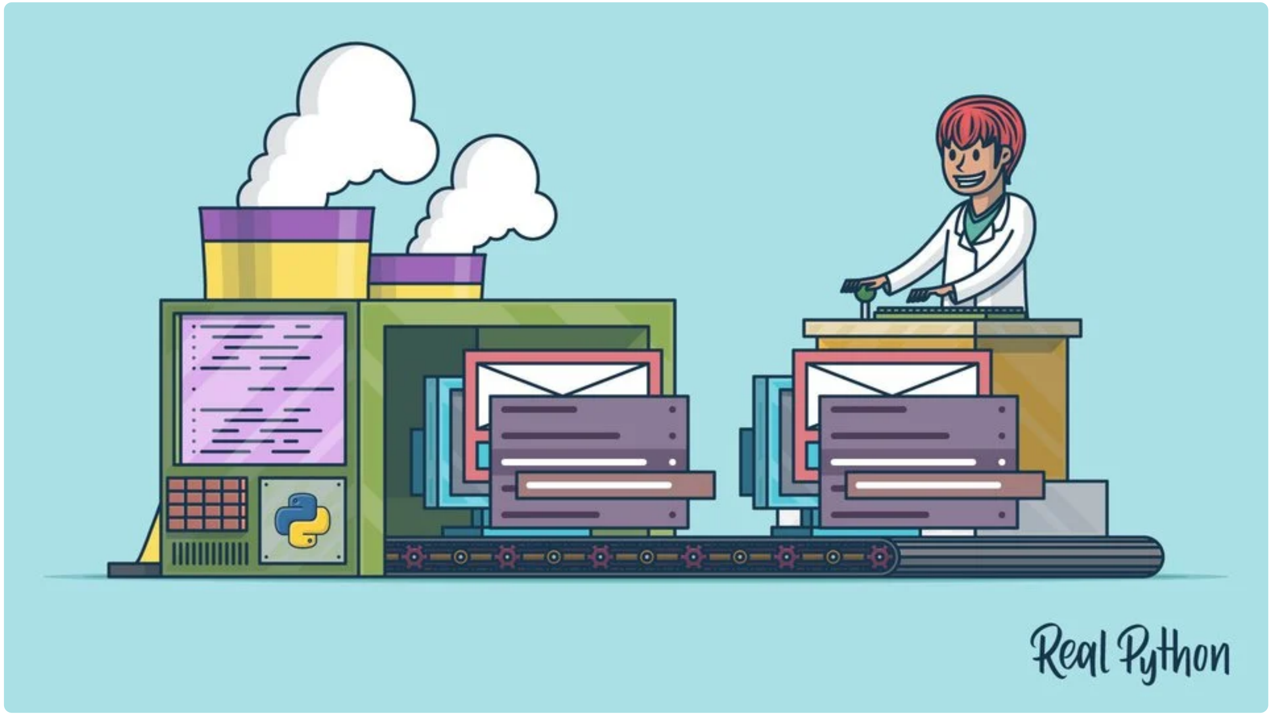


The Factory Method Pattern and Its Implementation in Python

by Isaac Rodriguez 24 Comments best-practices intermediate

Mark as Completed

Tweet Share Email

Table of Contents

- Introducing Factory Method

    - The Problems With Complex Conditional Code
    - Looking for a Common Interface
    - Refactoring Code Into the Desired Interface
    - Basic Implementation of Factory Method
- Recognizing Opportunities to Use Factory Method
    - An Object Serialization Example
    - Factory Method as an Object Factory
    - Supporting Additional Formats
- A General Purpose Object Factory
    - Not All Objects Can Be Created Equal
    - Separate Object Creation to Provide Common Interface
    - A Generic Interface to Object Factory
- Specializing Object Factory to Improve Code Readability
- Conclusion
- Further Reading

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-called Gang of Four (GoF: Gamma, Helm, Johson, 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 according to the nature of the problem. Each pattern is 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 Method is a design pattern that creates objects with a 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 know how apply it.

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 some parameter.

Instead of using a complex `if/elif/else` conditional structure to determine the concrete implementation, the application delegates that decision to a separate component that creates the concrete object. 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 representation is often called serializing. You'll often see these requirements implemented in a single function or method that contains all the logic and implementation, like in the following code:

Python

```python
# In serializer_demo.py

import json
import xml.etree.ElementTree as et

class Song:
    def __init__(self, song_id, title, artist):
        self.song_id = song_id
        self.title = title
        self.artist = artist


class SongSerializer:
    def serialize(self, song, format):
        if format == 'JSON':
            song_info = {
                'id': song.song_id,
                'title': song.title,
                'artist': song.artist
            }
            return json.dumps(song_info)
        elif format == 'XML':
            song_info = et.Element('song', attrib={'id': song.song_id})
            title = et.SubElement(song_info, 'title')
            title.text = song.title
            artist = et.SubElement(song_info, 'artist')
            artist.text = song.artist
            return et.tostring(song_info, encoding='unicode')
        else:
            raise ValueError(format)
```

In the example above, you have a basic `Song` class to represent a song and a `SongSerializer` class that can convert a `song` object into its `string` representation according to the value of the `format` parameter.

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:

Python

```
>>> import serializer_demo as sd
>>> song = sd.Song('1', 'Water of Love', 'Dire Straits')
>>> serializer = sd.SongSerializer()

>>> serializer.serialize(song, 'JSON')
'{"id": "1", "title": "Water of Love", "artist": "Dire Straits"}'

>>> serializer.serialize(song, 'XML')
'<song id="1"><title>Water of Love</title><artist>Dire Straits</artist>

>>> serializer.serialize(song, 'YAML')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "./serializer_demo.py", line 30, in serialize
    raise ValueError(format)
ValueError: YAML
```

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 `song` object as a parameter, as well as a string value 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` parameter. This may not seem like a big deal, and you've probably seen code with more complexity than this, but the above example is still pretty hard to maintain.

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 behavior of an application. Using `if/elif/else` conditional 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 method should have a single, well-defined 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 existing functionality when changes are made. 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 accommodate the new structure.

- **When the string representation for a format changes (plain JSON vs JSON API):** The `.serialize()` method will have to change if the desired string representation for a format changes because 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 you can do that in the following sections.

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 to its `string` representation using a different 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 `song` object and returns a `string`.

Once you have a common interface, you provide separate implementations for each logical path. In the example above, you will provide an implementation to serialize to JSON and another for XML.

Then, you provide a separate component that decides the concrete implementation to use based on the specified `format`. This component evaluates the value of `format` and returns the concrete 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 way that does not alter the external behavior of the code yet improves its internal structure.”

Let’s begin refactoring the code to achieve the desired structure that uses the Factory Method design pattern.

Refactoring Code Into the Desired Interface

The desired interface is an object or a function that takes a `Song` object and returns a `string` representation.

The first step is to refactor one of the logical paths into this interface. You do this by adding a new method `._serialize_to_json()` and moving the JSON serialization code to it. Then, you change the client to call it instead of having the implementation in the body of the `if` statement:

```python
class SongSerializer:
    def serialize(self, song, format):
        if format == 'JSON':
            return self._serialize_to_json(song)
        # The rest of the code remains the same

    def _serialize_to_json(self, song):
        payload = {
            'id': song.song_id,
            'title': song.title,
            'artist': song.artist
        }
        return json.dumps(payload)
```

Once you make this change, you can verify that the behavior has not changed. Then, you do the same for the XML option by introducing a new method `._serialize_to_xml()`, moving the implementation to it, and modifying the `elif` path to call it.

The following example shows the refactored code:

Python

```python
class SongSerializer:
    def serialize(self, song, format):
        if format == 'JSON':
            return self._serialize_to_json(song)
        elif format == 'XML':
            return self._serialize_to_xml(song)
        else:
            raise ValueError(format)

    def _serialize_to_json(self, song):
        payload = {
            'id': song.song_id,
            'title': song.title,
            'artist': song.artist
        }
        return json.dumps(payload)

    def _serialize_to_xml(self, song):
        song_element = et.Element('song', attrib={'id': song.song_id})
        title = et.SubElement(song_element, 'title')
        title.text = song.title
        artist = et.SubElement(song_element, 'artist')
        artist.text = song.artist
        return et.tostring(song_element, encoding='unicode')
```

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.

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 based on some specified parameter. That 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 value of `format` and returns the matching serialization function:

Python

```python
class SongSerializer:
    def _get_serializer(self, format):
        if format == 'JSON':
            return self._serialize_to_json
        elif format == 'XML':
            return self._serialize_to_xml
        else:
            raise ValueError(format)
```

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 example shows the complete code:

Python

```python
class SongSerializer:
    def serialize(self, song, format):
        serializer = self._get_serializer(format)
        return serializer(song)

    def _get_serializer(self, format):
        if format == 'JSON':
            return self._serialize_to_json
        elif format == 'XML':
            return self._serialize_to_xml
        else:
            raise ValueError(format)

    def _serialize_to_json(self, song):
        payload = {
            'id': song.song_id,
            'title': song.title,
            'artist': song.artist
        }
        return json.dumps(payload)

    def _serialize_to_xml(self, song):
        song_element = et.Element('song', attrib={'id': song.song_id})
        title = et.SubElement(song_element, 'title')
        title.text = song.title
        artist = et.SubElement(song_element, 'artist')
        artist.text = song.artist
        return et.tostring(song_element, encoding='unicode')
```

The final implementation shows the different components of Factory Method. The `.serialize()` method is the application code that depends on an interface to complete its task.

This is referred to as the **client** component of the pattern. The interface defined is referred to as the **product** component. In our case, the product is a function that takes a `Song` and returns a string representation.

The `._serialize_to_json()` and `._serialize_to_xml()` methods are concrete implementations of the product. Finally, the `._get_serializer()` method is the **creator** component. The creator decides which concrete implementation to use.

Because you started with some existing code, all the components of Factory Method are members of the same class `SongSerializer`.

Usually, this is not the case and, as you can see, none of the added methods use the `self` parameter. This is a good indication that they should not be methods of the `SongSerializer` class, and they can become external functions:

Python

```python
class SongSerializer:
    def serialize(self, song, format):
        serializer = get_serializer(format)
        return serializer(song)


def get_serializer(format):
    if format == 'JSON':
            return _serialize_to_json
        elif format == 'XML':
            return _serialize_to_xml
        else:
            raise ValueError(format)


def _serialize_to_json(song):
    payload = {
        'id': song.song_id,
        'title': song.title,
        'artist': song.artist
    }
    return json.dumps(payload)


def _serialize_to_xml(song):
    song_element = et.Element('song', attrib={'id': song.song_id})
    title = et.SubElement(song_element, 'title')
    title.text = song.title
    artist = et.SubElement(song_element, 'artist')
    artist.text = song.artist
    return et.tostring(song_element, encoding='unicode')
```

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 `SongSerializer` and replace the calls to the new 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 requests the implementation from a creator 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 complete its task.

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

Python >>>

```
>>> import serializer_demo as sd
>>> song = sd.Song('1', 'Water of Love', 'Dire Straits')
>>> serializer = sd.SongSerializer()

>>> serializer.serialize(song, 'JSON')
'{"id": "1", "title": "Water of Love", "artist": "Dire Straits"}'

>>> serializer.serialize(song, 'XML')
'<song id="1"><title>Water of Love</title><artist>Dire Straits</artist>

>>> serializer.serialize(song, 'YAML')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "./serializer_demo.py", line 13, in serialize
    serializer = get_serializer(format)
  File "./serializer_demo.py", line 23, in get_serializer
    raise ValueError(format)
ValueError: YAML
```

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 supported format, `ValueError` is raised.

Recognizing Opportunities to Use Factory Method

Factory Method should be used in every situation where an application (client) depends on an interface (product) to perform a task and there are multiple concrete implementations of that interface. You need to provide a parameter that can identify the concrete implementation and use it in the creator to decide the concrete implementation.

There is a wide range of problems that fit this description, so let’s take a look at some concrete examples.

Replacing complex logical code: Complex logical structures in the format `if/elif/else` are hard to maintain because new logical paths are needed as requirements change.

Factory Method is a good replacement because you can put the body of each logical path into separate functions or classes with a common interface, and the creator can provide the concrete implementation.

The parameter evaluated in the conditions becomes the parameter to identify the concrete implementation. The example above represents this situation.

Constructing related objects from external data: Imagine an application that needs to retrieve employee information from a database or other external source.

The records represent employees with different roles or types: managers, office clerks, sales associates, and so on. The application may store an identifier representing the type of employee in the record and then use Factory Method to create each concrete `Employee` object from the rest of the information on the record.

Supporting multiple implementations of the same feature: An image processing application needs to transform a satellite image from one coordinate system to another, but there are multiple algorithms with different levels of accuracy to perform the transformation.

The application can allow the user to select an option that identifies the concrete algorithm. Factory Method can provide the concrete implementation of the algorithm based on this option.

Combining similar features under a common interface: Following the image processing example, an application needs to apply a filter to an image. The specific filter to use can be identified by some user input, and Factory Method can provide the concrete filter implementation.

Integrating related external services: A music player application wants to integrate with multiple external services and allow users to select where their music comes from. The application can define a common interface for a music service and use Factory Method to create the correct integration based on a user preference.

All these situations are similar. They all define a client that depends on a common interface known as the product. They all provide a means to identify the concrete implementation of the product, so they all can use Factory Method in their design.

You can now look at the serialization problem from previous examples and provide a better design by taking into consideration the Factory Method design pattern.

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 provides features related to music, so it is plausible that 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 implementation. The application requires objects to be serialized to multiple formats like JSON and XML, so it seems natural to define an interface `Serializer` that can have multiple implementations, one per format.

The interface implementation might look something like this:

Python

```python
# In serializers.py

import json
import xml.etree.ElementTree as et

class JsonSerializer:
    def __init__(self):
        self._current_object = None

    def start_object(self, object_name, object_id):
        self._current_object = {
            'id': object_id
        }

    def add_property(self, name, value):
        self._current_object[name] = value

    def to_str(self):
        return json.dumps(self._current_object)


class XmlSerializer:
    def __init__(self):
        self._element = None

    def start_object(self, object_name, object_id):
        self._element = et.Element(object_name, attrib={'id': object_id

    def add_property(self, name, value):
        prop = et.SubElement(self._element, name)
        prop.text = value

    def to_str(self):
        return et.tostring(self._element, encoding='unicode')
```

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

Method.

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 be explicitly defined. In Python, any object that provides the desired methods or functions is said to implement the interface. The example defines the `Serializer` interface to be an object that implements the following methods or functions:

- `.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`:

Python

```python
# In serializers.py

class ObjectSerializer:
    def serialize(self, serializable, format):
        serializer = factory.get_serializer(format)
        serializable.serialize(serializer)
        return serializer.to_str()
```

The implementation of `ObjectSerializer` is completely generic, and it only mentions a `serializable` and a `format` as parameters.

The `format` is used to identify the concrete implementation of the `Serializer` and is resolved by the `factory` object. The `serializable` parameter refers to another abstract interface that should be implemented on any object type you want to serialize.

Let’s take a look at a concrete implementation of the `serializable` interface in the `Song` class:

Python

```python
# In songs.py

class Song:
    def __init__(self, song_id, title, artist):
        self.song_id = song_id
        self.title = title
        self.artist = artist

    def serialize(self, serializer):
        serializer.start_object('song', self.song_id)
        serializer.add_property('title', self.title)
        serializer.add_property('artist', self.artist)
```

The `Song` class implements the `Serializable` interface by providing a `.serialize(serializer)` method. In the method, the `Song` class uses the `serializer` object to write its own information without any knowledge of the format.

As a matter of fact, the `Song` class doesn't even know the goal is to convert the data to a string. This is important because you could use this interface to provide a different kind of `serializer` that converts the `Song` information to a completely different representation if needed. For example, your application might require in the future to convert the `Song` object to a binary format.

So far, we've seen the implementation of the client (`ObjectSerializer`) and the product (`serializer`). It is time to complete the implementation of Factory Method and provide the creator. The creator in the example is the variable `factory` in `ObjectSerializer.serialize()`.

Factory Method as an Object Factory

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 flexibility when requirements change.

Classes can provide additional interfaces to add functionality, and they can be derived to customize behavior. Unless you have a very basic creator that will never change in the future, you want to 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(format)` to retrieve the `serializer` from the object factory.

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

Python

```python
# In serializers.py

class SerializerFactory:
    def get_serializer(self, format):
        if format == 'JSON':
            return JsonSerializer()
        elif format == 'XML':
            return XmlSerializer()
        else:
            raise ValueError(format)


factory = SerializerFactory()
```

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 concrete implementation to create and return. It is 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:

```
Python                                                                    >>>

>>> import songs
>>> import serializers
>>> song = songs.Song('1', 'Water of Love', 'Dire Straits')
>>> serializer = serializers.ObjectSerializer()

>>> serializer.serialize(song, 'JSON')
'{"id": "1", "title": "Water of Love", "artist": "Dire Straits"}'

>>> serializer.serialize(song, 'XML')
'<song id="1"><title>Water of Love</title><artist>Dire Straits</artist><

>>> serializer.serialize(song, 'YAML')
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "./serializers.py", line 39, in serialize
    serializer = factory.get_serializer(format)
  File "./serializers.py", line 52, in get_serializer
    raise ValueError(format)
ValueError: YAML
```

The new design of Factory Method allows the application to introduce new features by adding new classes, as opposed to changing existing ones. You can serialize other objects by implementing 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 because

`SerializerFactory` is a class.

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 any additional formats, but you never know.

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:

Python

```python
# In serializers.py

class SerializerFactory:

    def __init__(self):
        self._creators = {}

    def register_format(self, format, creator):
        self._creators[format] = creator

    def get_serializer(self, format):
        creator = self._creators.get(format)
        if not creator:
            raise ValueError(format)
        return creator()


factory = SerializerFactory()
factory.register_format('JSON', JsonSerializer)
factory.register_format('XML', XmlSerializer)
```

The `.register_format(format, creator)` method allows registering new formats by specifying a `format` value used to identify the format and a `creator` object. The creator object happens to be the class name of the concrete `Serializer`. This is possible because all the `Serializer` classes provide a default `.__init__()` to initialize the instances.

The registration information is stored in the `_creators` dictionary. The `.get_serializer()` method retrieves the registered creator and creates the desired object. If the requested `format` has not been registered, then `ValueError` is raised.

You can now verify the flexibility of the design by implementing a `YamlSerializer` and get rid of the annoying `ValueError` you saw earlier:

Python

```python
# In yaml_serializer.py

import yaml
import serializers

class YamlSerializer(serializers.JsonSerializer):
    def to_str(self):
        return yaml.dump(self._current_object)


serializers.factory.register_format('YAML', YamlSerializer)
```

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 implementation. The format is then registered with the `factory` object to make it available.

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

Python >>>

```
>>> import serializers
>>> import songs
>>> import yaml_serializer
>>> song = songs.Song('1', 'Water of Love', 'Dire Straits')
>>> serializer = serializers.ObjectSerializer()

>>> print(serializer.serialize(song, 'JSON'))
{"id": "1", "title": "Water of Love", "artist": "Dire Straits"}

>>> print(serializer.serialize(song, 'XML'))
<song id="1"><title>Water of Love</title><artist>Dire Straits</artist><

>>> print(serializer.serialize(song, 'YAML'))
{artist: Dire Straits, id: '1', title: Water of Love}
```

By implementing Factory Method using an Object Factory and providing a registration interface, you are able to support new formats without changing any of the existing application code. This minimizes 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 avoids modifying existing code.

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. Ideally, you'll want an implementation of Object 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 challenges and implement a solution that can be reused in 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, returns fully initialized objects.

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 provided objects. This defeats the purpose of the Factory 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 different music services. These services can be external 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 integrate with services like Pandora or Spotify.

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

Imagine that the application wants to integrate with a service provided by Spotify. This service requires an authorization process where a client key and secret are provided for authorization.

The service returns an access code that should be used on any further communication. This authorization process is very slow, and it should only be performed once, so the application wants to keep the initialized service object around and use it every time it needs to communicate with Spotify.

At the same time, other users want to integrate with Pandora. Pandora might use a completely different authorization process. It also requires a client key and secret, but it returns a consumer key and secret that should be used for other communications. As with Spotify, the authorization process is slow, and it should only be performed once.

Finally, the application implements the concept of a local music service where the music collection is stored locally. The service requires that the the location of the music collection in the local system be specified. Creating a new service instance is done very quickly, so a new instance can be created every time the user wants to access the music collection.

This example presents several challenges. Each service is initialized with a different set of parameters. Also, Spotify and Pandora require an authorization process before the service instance can be created.

They also want to reuse that instance to avoid authorizing the application multiple times. The local service is simpler, but it doesn’t match the initialization interface of the others.

In the following sections, you will solve this problems by generalizing the creation interface and implementing a general purpose Object Factory.

Separate Object Creation to Provide Common Interface

The creation of each concrete music service has its own set of requirements. This means a common initialization interface for each service implementation is not possible or recommended.

The best approach is to define a new type of object that provides a general interface and is responsible for the creation of a concrete service. This new type of object will be called a `Builder`. The `Builder` object has all the logic to create and initialize a service instance. You will implement a `Builder` object for each of the supported services.

Let’s start by looking at the application configuration:

Python

```python
# In program.py

config = {
    'spotify_client_key': 'THE_SPOTIFY_CLIENT_KEY',
    'spotify_client_secret': 'THE_SPOTIFY_CLIENT_SECRET',
    'pandora_client_key': 'THE_PANDORA_CLIENT_KEY',
    'pandora_client_secret': 'THE_PANDORA_CLIENT_SECRET',
    'local_music_location': '/usr/data/music'
}
```

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 create a concrete implementation of a music service. That interface will be implemented in a `Builder`.

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

Python

```python
# In music.py

class SpotifyService:
    def __init__(self, access_code):
        self._access_code = access_code

    def test_connection(self):
        print(f'Accessing Spotify with {self._access_code}')


class SpotifyServiceBuilder:
    def __init__(self):
        self._instance = None

    def __call__(self, spotify_client_key, spotify_client_secret, **_igr
        if not self._instance:
            access_code = self.authorize(
                spotify_client_key, spotify_client_secret)
            self._instance = SpotifyService(access_code)
        return self._instance

    def authorize(self, key, secret):
        return 'SPOTIFY_ACCESS_CODE'
```

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 provided through `**_ignored`. Once the `access_code` is 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 through the authorization process multiple times as specified in the requirements.

Let’s do the same for Pandora:

Python

```python
# In music.py

class PandoraService:
    def __init__(self, consumer_key, consumer_secret):
        self._key = consumer_key
        self._secret = consumer_secret

    def test_connection(self):
        print(f'Accessing Pandora with {self._key} and {self._secret}')


class PandoraServiceBuilder:
    def __init__(self):
        self._instance = None

    def __call__(self, pandora_client_key, pandora_client_secret, **_ig
        if not self._instance:
            consumer_key, consumer_secret = self.authorize(
                pandora_client_key, pandora_client_secret)
            self._instance = PandoraService(consumer_key, consumer_secr
        return self._instance

    def authorize(self, key, secret):
        return 'PANDORA_CONSUMER_KEY', 'PANDORA_CONSUMER_SECRET'
```

The `PandoraServiceBuilder` implements the same interface, but it uses different parameters and processes to create and initialize the `PandoraService`. It also keeps the service instance around, so the authorization only happens once.

Finally, let’s take a look at the local service implementation:

Python

```python
# In music.py

class LocalService:
    def __init__(self, location):
        self._location = location

    def test_connection(self):
        print(f'Accessing Local music at {self._location}')


def create_local_music_service(local_music_location, **_ignored):
    return LocalService(local_music_location)
```

The `LocalService` just requires a location where the collection is stored to initialize the `LocalService`.

A new instance is created every time the service is requested because there is no slow authorization process. The requirements are simpler, so you don’t need a `Builder` class. Instead, a function returning an initialized `LocalService` is used. This function matches the interface of the `.__call__()` methods implemented in the builder classes.

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 a `Builder` based on a `key` value and a method to create the concrete object instances based on the `key`.

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

Python

```python
# In object_factory.py

class ObjectFactory:
    def __init__(self):
        self._builders = {}

    def register_builder(self, key, builder):
        self._builders[key] = builder

    def create(self, key, **kwargs):
        builder = self._builders.get(key)
        if not builder:
            raise ValueError(key)
        return builder(**kwargs)
```

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 callable interface. This means a `Builder` can be a function, 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 they need and ignore the rest in no particular order. 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:

Python

```python
# In music.py
import object_factory

# Omitting other implementation classes shown above

factory = object_factory.ObjectFactory()
factory.register_builder('SPOTIFY', SpotifyServiceBuilder())
factory.register_builder('PANDORA', PandoraServiceBuilder())
factory.register_builder('LOCAL', create_local_music_service)
```

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

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

Python

```python
# In program.py
import music

config = {
    'spotify_client_key': 'THE_SPOTIFY_CLIENT_KEY',
    'spotify_client_secret': 'THE_SPOTIFY_CLIENT_SECRET',
    'pandora_client_key': 'THE_PANDORA_CLIENT_KEY',
    'pandora_client_secret': 'THE_PANDORA_CLIENT_SECRET',
    'local_music_location': '/usr/data/music'
}

pandora = music.factory.create('PANDORA', **config)
pandora.test_connection()

spotify = music.factory.create('SPOTIFY', **config)
spotify.test_connection()

local = music.factory.create('LOCAL', **config)
local.test_connection()

pandora2 = music.services.get('PANDORA', **config)
print(f'id(pandora) == id(pandora2): {id(pandora) == id(pandora2)}')

spotify2 = music.services.get('SPOTIFY', **config)
print(f'id(spotify) == id(spotify2): {id(spotify) == id(spotify2)}')
```

The application defines a `config` dictionary representing the application configuration. The configuration is used as the keyword arguments to the factory regardless of the service you want to access. The factory creates the concrete implementation of the music service based on the specified `key` parameter.

You can now run our program to see how it works:

Shell

```
$ python program.py
Accessing Pandora with PANDORA_CONSUMER_KEY and PANDORA_CONSUMER_SECRET
Accessing Spotify with SPOTIFY_ACCESS_CODE
Accessing Local music at /usr/data/music
id(pandora) == id(pandora2): True
id(spotify) == id(spotify2): True
```

You can see that the correct instance is created depending on the specified service type. You can also see that requesting the Pandora or Spotify service always returns the same instance.

Specializing Object Factory to Improve Code Readability

General solutions are reusable and avoid code duplication. Unfortunately, they can also obscure the code and make it less readable.

The example above shows that, to access a music service, `music.factory.create()` is called. This may lead to confusion. Other developers might believe that a new instance is created every time and decide that they should keep around the service instance to avoid the slow initialization process.

You know that this is not what happens because the `Builder` class keeps the initialized instance and returns it for subsequent calls, but this isn't clear from just reading the code.

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 will specialize `ObjectFactory` in the context of our 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:

Python

```python
# In music.py

class MusicServiceProvider(object_factory.ObjectFactory):
    def get(self, service_id, **kwargs):
        return self.create(service_id, **kwargs)


services = MusicServiceProvider()
services.register_builder('SPOTIFY', SpotifyServiceBuilder())
services.register_builder('PANDORA', PandoraServiceBuilder())
services.register_builder('LOCAL', create_local_music_service)
```

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. You also renamed the previous `factory` variable to `services` and initialized it as a `MusicServiceProvider`.

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

Python

```python
import music

config = {
    'spotify_client_key': 'THE_SPOTIFY_CLIENT_KEY',
    'spotify_client_secret': 'THE_SPOTIFY_CLIENT_SECRET',
    'pandora_client_key': 'THE_PANDORA_CLIENT_KEY',
    'pandora_client_secret': 'THE_PANDORA_CLIENT_SECRET',
    'local_music_location': '/usr/data/music'
}

pandora = music.services.get('PANDORA', **config)
pandora.test_connection()
spotify = music.services.get('SPOTIFY', **config)
spotify.test_connection()
local = music.services.get('LOCAL', **config)
local.test_connection()

pandora2 = music.services.get('PANDORA', **config)
print(f'id(pandora) == id(pandora2): {id(pandora) == id(pandora2)}')

spotify2 = music.services.get('SPOTIFY', **config)
print(f'id(spotify) == id(spotify2): {id(spotify) == id(spotify2)}')
```

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

Shell

```
$ python program.py
Accessing Pandora with PANDORA_CONSUMER_KEY and PANDORA_CONSUMER_SECRET
Accessing Spotify with SPOTIFY_ACCESS_CODE
Accessing Local music at /usr/data/music
id(pandora) == id(pandora2): True
id(spotify) == id(spotify2): True
```

Conclusion

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

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 modifying existing code to support new requirements.

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

In this article, you learned:

- What the Factory Method design pattern is and what its components are
- How to refactor existing code to leverage Factory Method
- Situations in which Factory Method should be used
- How Object Factories provide more flexibility to implement Factory Method
- How to implement a general purpose Object Factory and its challenges
- How to specialize a general solution to provide a better context

Further Reading

If you want to learn more about Factory Method and other design patterns, I recommend Design Patterns: Elements of Reusable Object-Oriented Software by the GoF, which is a great reference for widely adopted design patterns.

Also, Heads First Design Patterns: A Brain-Friendly Guide by Eric Freeman and Elisabeth Robson provides a fun, easy-to-read explanation of design patterns.

Wikipedia has a good catalog of design patterns with links to pages for the most common and useful patterns.

Mark as Completed

Python Tricks

Get a short & sweet **Python Trick** delivered to your inbox every couple of days. No spam ever. Unsubscribe any time. Curated by the Real Python team.

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Email Address

Send Me Python Tricks »

About **Isaac Rodriguez**

Hi, I'm Isaac. I build, lead, and mentor software development teams, and for the past few years I've been focusing on cloud services and back-end applications using Python among other languages. Love to hear from you here at Real Python.

» More about Isaac

Each tutorial at Real Python is created by a team of developers so that it meets our high quality standards. The team members who worked on this tutorial are:

Aldren

Geir Arne

Joanna

What Do You Think?

Tweet Share Email

Real Python Comment Policy: The most useful comments are those written with the goal of learning from or helping out other readers—after reading the whole article and all the earlier comments. Complaints and insults generally won't make the cut here.

What's your #1 takeaway or favorite thing you learned? How are you going to put your newfound skills to use? Leave a comment below and let us know.

Keep Learning

Related Tutorial Categories: best-practices intermediate

— FREE Email Series —

🐍 Python Tricks 💌

```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```

Email…

Get Python Tricks »

No spam. Unsubscribe any time.

All Tutorial Topics

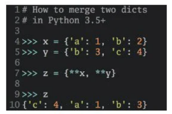
advanced api basics best-practices community databases data-science devops django docker flask front-end gui intermediate machine-learning projects python testing tools web-dev web-scraping

Table of Contents

- Introducing Factory Method
- Recognizing Opportunities to Use Factory Method
- A General Purpose Object Factory
- Specializing Object Factory to Improve Code Readability
- Conclusion
- Further Reading

Mark as Completed

Tweet Share Email



```
# How to merge two dicts
# in Python 3.5+

>>> x = {'a': 1, 'b': 2}
>>> y = {'b': 3, 'c': 4}

>>> z = {**x, **y}

>>> z
{'c': 4, 'a': 1, 'b': 3}
```