Flutter Design Patterns: 11 — Abstract Factory

medium.com/flutter-community/flutter-design-patterns-11-abstract-factory-7098112925d8

June 1, 2020

An overview of the Abstract Factory design pattern and its implementation in Dart and Flutter

Flutter Design Patterns
Abstract Factory
Abstract Factory
Provide an interface for creating families of related or dependent objects without specifying their concrete classes.
Class diagram
Implementation
Class diagram
The class diagram below shows the implementation of Abstract Factory design pattern.
Description
Example

In the last article, I have analysed the Factory Method design pattern. This time I would like to analyse and implement an OOP design pattern, which has a similar purpose, even a similar name, but is more flexible and suits the structure of big projects better than the Factory Method design pattern — it is the Abstract Factory.

Table of Contents

What is the Abstract Factory design pattern?

Santa’s factory (source)

Abstract Factory is a **creational** design pattern, also known as **Kit**. Its intention in the GoF book is described as:

> Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

The main purpose of the Abstract Factory design pattern is to encapsulate creating a family of objects in a separate factory object, hence abstracting the process of object creation. For all supported families of objects, a common interface for creating a family of objects is defined, then concrete factory class is created implementing this interface.

If you follow this series, I expect you have just had a sense of *deja vu,* haven’t you? The Factory Method design pattern has pretty much the same intention and purpose. Yes, it is true, simply like that. But why there is a separate factory pattern, then? The main difference between these two patterns is that the Abstract Factory pattern provides a way to create a **family of related objects** — a single factory is responsible for creating several objects. As a result, you don’t need to provide a separate factory for each specific class/component. In fact, you can consider the Factory Method design pattern as a subset of the Abstract Factory pattern — the Abstract Factory consists of several factory methods where each one of it creates only one specific object.

The Abstract Factory design pattern makes the creation of the objects more flexible:

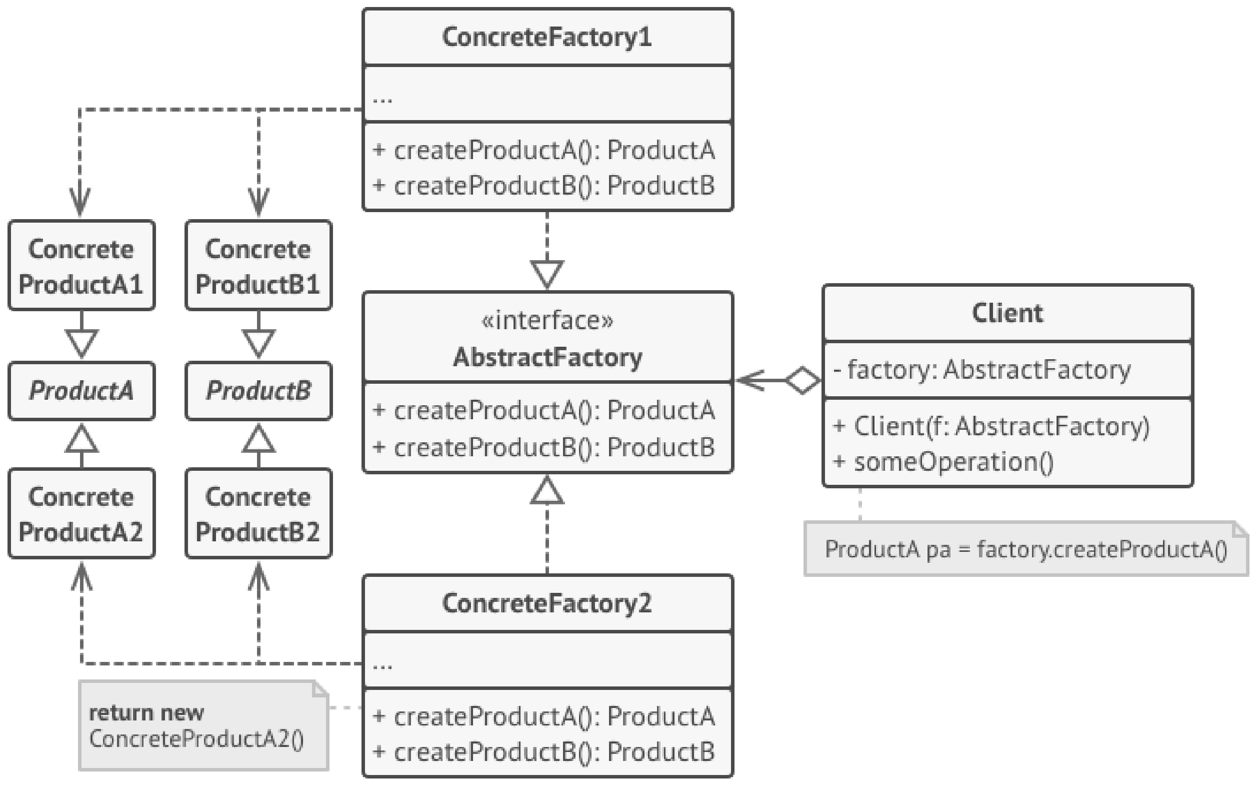
- *Compile-time* flexibility — the way objects are created can be implemented and changed independently from clients by defining new (sub)classes;
- *Run-time* flexibility — a class can be configured with a factory object, which it uses to create objects, and even more, the factory object can be exchanged dynamically.

Finally, this pattern removes the direct binding of application-specific classes into the code. Correspondingly, the code only deals with interfaces of specific objects and their factories, but not with the specific implementations.

Let’s move to the analysis to understand the details of the Abstract Factory and how this design pattern could be implemented.

Analysis

The general structure of the Abstract Factory design pattern looks like this:


ConcreteFactory1
...
+ createProductA(): ProductA
+ createProductB(): ProductB
Concrete ProductA1
Concrete ProductB1
ProductA
ProductB
Concrete ProductA2
Concrete ProductB2
«interface» AbstractFactory
+ createProductA(): ProductA
+ createProductB(): ProductB
Client
- factory: AbstractFactory
+ Client(f: AbstractFactory)
+ someOperation()
ProductA pa = factory.createProductA()
ConcreteFactory2
...
+ createProductA(): ProductA
+ createProductB(): ProductB
return new ConcreteProductA2()

Structure of the Abstract Factory design pattern (source)

- *Abstract Factory* — declares an interface of operations that create abstract *Product* objects;
- *Concrete Factory* — implements the operations to create *Concrete Product* objects. **Each *Concrete Factory* corresponds only to a single variant of products**;
- *Product* — declares an interface for a type of *Product* object;
- *Concrete Product* — implements the *Product* interface and defines a product object to be created by the corresponding *Concrete Factory*;
- *Client* — uses only interfaces declared by the *Abstract Factory* and *Product* classes.

Applicability

The usage of the Abstract Factory design pattern should be considered when a system’s code needs to work with various families of related objects (products), but it should not depend on the concrete classes of those products, on how they are created, composed and represented. The said design pattern provides an interface for the creation of objects from each class of the product family. By using this interface instead of concrete implementations of objects, the representation layer or the system's code, in general, should not worry about creating the wrong variant of a product which does not match other objects from the family. This restriction is helpful when you want to introduce platform-specific widgets/UI components to the representation layer and keep the consistency across the whole system.

Implementation

If you have read the last article, you should already be familiar with the problem that could be resolved by using the factory design pattern. If not, here is a short overview of the problem we will fix:

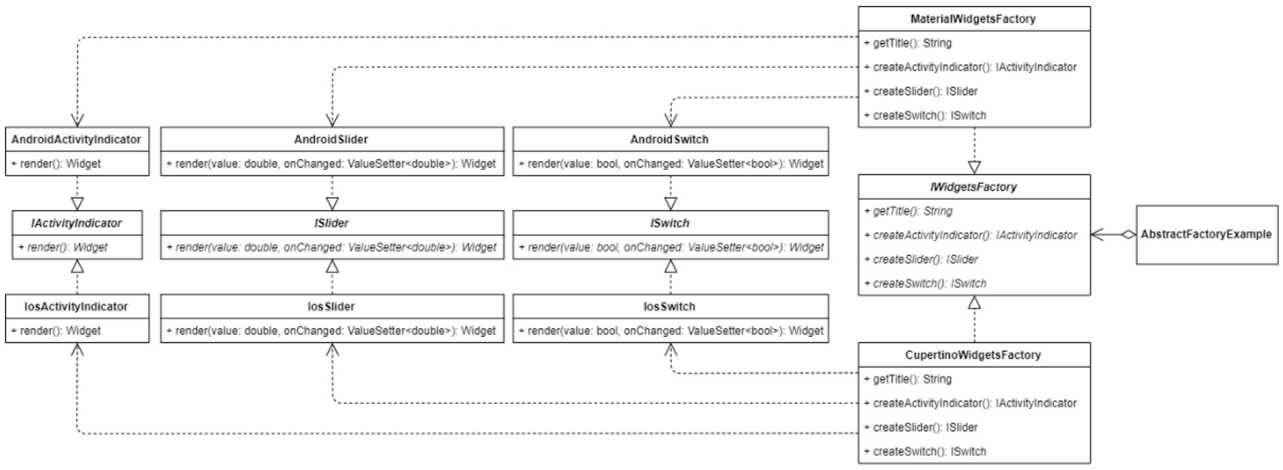
> Even though you are using the same code base with Flutter, usually there is a demand that UI components should look different on different platforms. The simplest imaginable use case in Flutter context — showing the Material or Cupertino style widgets based on whether you are using accordingly an Android or iOS device.

Last time, we have resolved this issue by introducing the Factory Method design pattern to our code and providing a separate factory for each platform-specific component which creates the required widget dynamically after checking the current platform, hence separating the business logic from the representation layer (UI). However, this approach becomes a tremendous headache when multiple components exist which should look different on distinct platforms. Based on the Factory Method design pattern, every component should have a dedicated factory class to it and e.g. if you are creating an application for Android, iOS and Web, every new component would also require to add a new abstract class and 3 extra derived classes for the implementation in each specific platform.

Having the said problems in mind, the Abstract Factory design pattern is a better option than the Factory Method since only a single factory is needed per platform, a family of components are created and used together.

Class diagram

The class diagram below shows the implementation of the **Abstract Factory** design pattern.



MaterialWidgetsFactory
+ getTitle(): String
+ createActivityIndicator(): IActivityIndicator
+ createSlider(): ISlider
+ createSwitch(): ISwitch
AndroidActivityIndicator
+ render(): Widget
AndroidSlider
+ render(value: double, onChanged: ValueSetter<double>): Widget
AndroidSwitch
+ render(value: bool, onChanged: ValueSetter<bool>): Widget
IWidgetsFactory
+ getTitle(): String
+ createActivityIndicator(): IActivityIndicator
+ createSlider(): ISlider
+ createSwitch(): ISwitch
AbstractFactoryExample
IActivityIndicator
+ render(): Widget
ISlider
+ render(value: double, onChanged: ValueSetter<double>): Widget
ISwitch
+ render(value: bool, onChanged: ValueSetter<bool>): Widget
IosActivityIndicator
+ render(): Widget
IosSlider
+ render(value: double, onChanged: ValueSetter<double>): Widget
IosSwitch
+ render(value: bool, onChanged: ValueSetter<bool>): Widget
CupertinoWidgetsFactory
+ getTitle(): String
+ createActivityIndicator(): IActivityIndicator
+ createSlider(): ISlider
+ createSwitch(): ISwitch

Class Diagram — Implementation of the Abstract Factory design pattern

IWidgetsFactory is an abstract class which is used as an interface for all the specific widget factories:

- *getTitle()* — an abstract method which returns the title of the factory. Used in the UI;
- *createActivityIndicator()* — an abstract method which returns the specific implementation (UI component/widget) of the activity (process) indicator implementing the *IActivityIndicator* interface;
- *createSlider()* — an abstract method which returns the specific implementation (UI component/widget) of the slider implementing the *ISlider* interface;
- *createSwitch()* — an abstract method which returns the specific implementation (UI component/widget) of the switch button implementing the *ISwitch* interface.

MaterialWidgetsFactory and *CupertinoWidgetsFactory* are concrete classes which implement the *IWidgetsFactory* class and its methods. *MaterialWidgetsFactory* creates Material style components (widgets) while the *CupertinoWidgetsFactory* creates Cupertino style widgets.

IActivityIndicator, *ISlider* and *ISwitch* are abstract classes which define the *render()* method for each component. These classes are implemented by both — Material and Cupertino — widgets.

AndroidActivityIndicator, *AndroidSlider* and *AndroidSwitch* are concrete implementations of the Material widgets implementing the *render()* method of corresponding interfaces.

IosActivityIndicator, *IosSlider* and *IosSwitch* are concrete implementations of the Cupertino widgets implementing the *render()* method of corresponding interfaces.

AbstractFactoryExample contains a list of factories implementing the *IWidgetsFactory* interface. After selecting the specific factory, the example widget uses its methods to create the corresponding widgets/UI components.

IWidgetsFactory

An interface which defines methods to be implemented by the specific factory classes. These methods are used to create components (widgets) of the specific type defined by the concrete factory. Dart language does not support the interface as a class type, so we define an interface by creating an abstract class and providing a method header (name, return type, parameters) without the default implementation.

Widget factories

MaterialWidgetsFactory — a concrete factory class which implements the *IWidgetsFactory* interface and its methods creating the Material style widgets.

CupertinoWidgetsFactory — a concrete factory class which implements the *IWidgetsFactory* interface and its methods creating the Cupertino style widgets.

IActivityIndicator

An interface which defines the *render()* method to render the activity indicator component (widget).

Activity indicator widgets

AndroidActivityIndicator — a specific implementation of the activity indicator component returning the Material style widget *CircularProgressIndicator*.

IosActivityIndicator — a specific implementation of the activity indicator component returning the Cupertino style widget *CupertinoActivityIndicator*.

ISlider

An interface which defines the *render()* method to render the slider component (widget).

Slider widgets

AndroidSlider — a specific implementation of the slider component returning the Material style widget *Slider*.

IosSlider — a specific implementation of the slider component returning the Cupertino style widget *CupertinoSlider*.

ISwitch

An interface which defines the *render()* method to render the switch component (widget).

Switch widgets

AndroidSwitch — a specific implementation of the switch button component returning the Material style widget *Switch*.

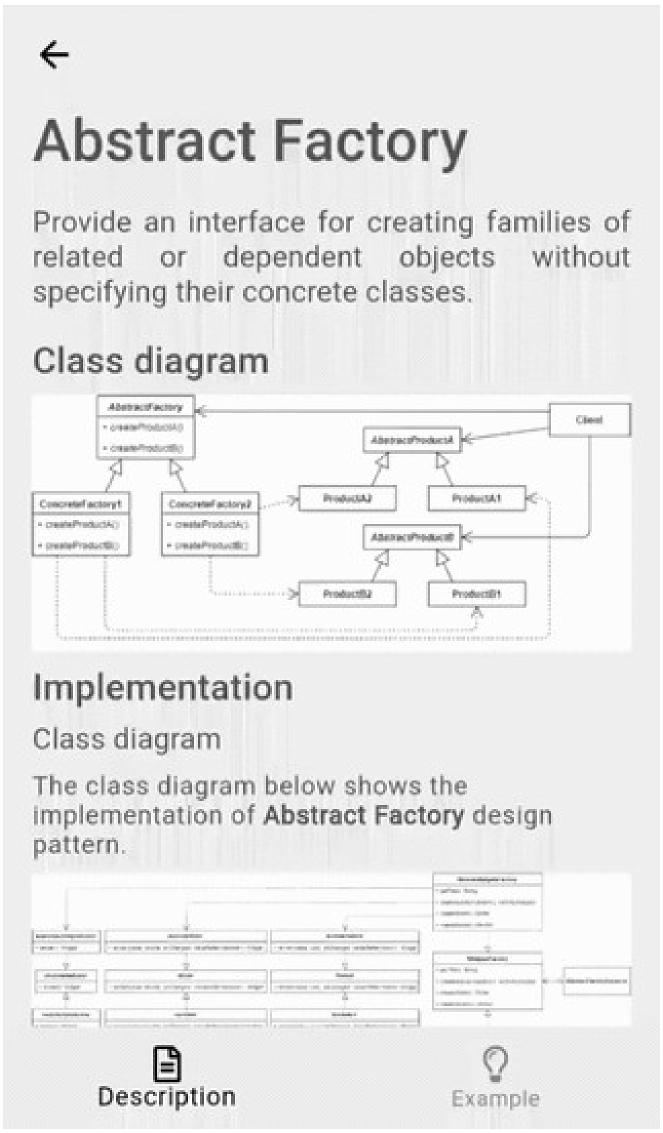
IosSwitch — a specific implementation of the switch button component returning the Cupertino style widget *CupertinoSwitch*.

Example

First of all, a markdown file is prepared and provided as a pattern's description:

AbstractFactoryExample contains a list of *IWidgetsFactory* objects (factories). After selecting the specific factory from the list, corresponding widgets are created using the factory methods and provided to the UI.

As you can see in the *build()* method, the example widget does not care about the selected concrete factory as long as it implements the *IWidgetsFactory* interface which methods return components implementing the corresponding common interfaces among all the factories and providing the *render()* methods used in the UI. Also, the implementation of the specific widgets is encapsulated and defined in separate widget classes implementing the *render()* method. Hence, the UI logic is not tightly coupled to any factory or component class which implementation details could be changed independently without affecting the implementation of the UI itself.



Abstract Factory

Provide an interface for creating families of related or dependent objects without specifying their concrete classes.

Class diagram

Implementation

Class diagram

The class diagram below shows the implementation of **Abstract Factory** design pattern.

As you can see in the example, by selecting the specific platform widgets option, appropriate widgets are created by the factory methods and provided to the user.

All of the code changes for the Abstract Factory design pattern and its example implementation could be found here.

Your contribution

👏 Press the clap button below to show your support and motivate me to write better!
💬 Leave a response to this article by providing your insights, comments or wishes for the series.
📢 Share this article with your friends, colleagues in social media.
➕ Follow me on Medium.
⭐ Star the Github repository.

Flutter Community

The latest Tweets from Flutter Community (@FlutterComm). Follow to get notifications of new articles and packages from…

www.twitter.com