1. Quelles sont les différentes approches de développement mobile ?

Développement d'applications natives

Devenir natif est le rêve de la plupart des propriétaires de produits, mais tous ne peuvent pas se le permettre. Cela implique la création d'une application mobile qui s'adapte à une seule plateforme.

Développement d'applications hybrides

Native n'est pas un must pour l'expérience mobile. Votre projet peut nécessiter une autre contrainte de priorité, comme un délai de commercialisation rapide. À cet effet, une application hybride n'est pas seulement une alternative moins chère. Elle nécessite moins de temps pour le développement et permet le partage de code.

Développement d'applications multiplateformes

L'approche de développement multiplateforme utilise un moteur de rendu natif. La base de code, écrite en javascript, se connecte aux composants natifs au travers de ce que l’on appel des ponts. Cela fournit une UX proche du natif. Les applications multiplateformes sont exemptes d’adhérences particulière à une plateforme. Elles offrent une grande transparence, une mise en œuvre facile et une production rentable. Mais ne vous attendez pas à des performances exceptionnelles. La personnalisation est également un problème car elle se limite au framework que vous utilisez.

1. C’est quoi la différence entre l’approche natif et l’approche natif multiplateforme (cross Platform) ?

Les appli cross-platform ressemblent beaucoup aux appli hybrides. Cependant, contrairement aux appli hybrides dont le code s’exécute dans une « webview » native, les applications cross-platform produisent un même code source qui, compilé, produit deux applications natives. Cela induit une meilleure performance et une expérience utilisateur plus proche du natif.

1. Dans quels cas nous pouvons opter pour une approche native ?

Les applications natives sont plus performantes, plus sécurisées et offrent la meilleure expérience utilisateur. La conception, le contenu, la structure et les éléments visuels favorisent un chargement instantané. Ce qui les rend plus rapides, quelle que soit la vitesse internet du téléphone. iPhone ou téléphone Android, peu importe, l’application native reste accessible et offre une expérience de navigation optimale.

1. C’est quoi la différence entre Ionic et Flutter ?

Flutter utilise comme langage de programmation DART or Ionic utilise HTML CSS et JS , les deux peuvent être déployer dans le web et le mobile or Ionic publie aussi dans PWA , or pour la performance il sont tout les deux performant dans le mobile mais flutter est moins performant dans le web .

1. Avec l’approche web responsive, peut-on développer des applications qui font recours aux API natives (ex. contat, telephony, GPS, Camera, etc) ?

Oui, parce que le web responsive peut s'adapter à tout type d'appareil de manière transparente pour l'utilisateur .

1. Quelles sont les derniers versions de : Ionic, React Native et Flutter ?

Le 6 mai 2020, le SDK de Dart en version 2.8 et celui de Flutter en version 1.17.0 sont sortis.

Pour Ionic 6.1.3 (2022-04-27)

Pour React Native : 0.67 (19-1-2022)

7. Quel est le propriétaire de Xamarin ? Quel type d’approche support ?

En février 2016 , Xamarin est rachetée par Microsoft. Le 31 mars 2016 , durant la conférence Build de Microsoft, Nat Friedman annonce la mise à disposition gratuite de Xamarin dans Visual Studio Community Edition.

Xamarin.Forms les applications peuvent être écrites pour les systèmes d’exploitation suivants :

iOS 9 ou ultérieur.

Android 4.4 (API 19) ou ultérieur (plus d’informations). Toutefois, Android 5.0 (API 21) est recommandé comme API minimale. Cela garantit une compatibilité complète avec toutes les bibliothèques de prise en charge Android, tout en ciblant toujours la majorité des appareils Android.

Windows 10 plateforme Windows universelle, créez 10.0.16299.0 ou une version ultérieure pour la prise en charge .NET Standard 2,0. Toutefois, la version 10.0.18362.0 ou ultérieure est recommandée.

8. Quel est le rôle de Cordova ?

Cordova agit comme un conteneur : son rôle est d'encapsuler une application Web dans une application native, on parle alors d'application hybride. Techniquement, Cordova va embarquer les ressources de l'application Web (fichiers HTML / JS / CSS mais aussi images, fonts, etc.)

9. C’est quoi la différence entre Ionic et Cordova ?

Sous IONIC, c’est un webview qui preset une application Web classique (JS / CSS / HTML ) en addition à  Cordova pour la création d’une application mobile fonctionnelle.

Cordova est un framework de développement mobile open source. Il est utilisé par les développeurs pour créer des applications mobiles multiplates-formes. Il a une base de code unique. Cordova peut être développé à l'aide de HTML5, JavaScript et CSS.

Cordova sert de pont vers le code natif pour Ionic. Il compile votre HTML, JavaScript et CSS dans des applications natives, leur permettant de s'exécuter en natif sur les appareils Android, iOS et Windows Phone.

10. Avec quel langage fonctionne Flutter ? Ce langage est créé par qui ?

Les applications Flutter sont écrites en [Dart](https://fr.wikipedia.org/wiki/Dart_(langage)) et utilisent de nombreuses fonctionnalités avancées du langage.

Le Flutter SDK se base sur le [langage de programmation Dart](https://www.ionos.fr/digitalguide/sites-internet/developpement-web/le-langage-de-programmation-dart/) également créé et développé par Google. Il se veut le successeur moderne du langage JavaScript classique et, tout comme ce dernier, il s’exécute directement sur les navigateurs, sous forme d’application web. Les programmes Dart peuvent aussi être exécutés directement sur un serveur.

11. C’est quoi le principe de Factory pattern ? Donner un exemple d’utilisation dans les applications mobile.

Le factory pattern décrit une approche de programmation qui vous permet **de créer des objets sans avoir à spécifier la classe exacte de ces objets**.

La fabrique permet de créer un objet dont le type dépend du contexte : cet objet fait partie d'un ensemble de sous-classes. L'objet retourné par la fabrique est donc toujours du type de la classe mère mais grâce au polymorphisme les traitements exécutés sont ceux de l'instance créée.

Ce motif de conception est utilisé lorsqu'à l'exécution il est nécessaire de déterminer dynamiquement quel objet d'un ensemble de sous-classes doit être instancié.

Il est utilisable lorsque :

* Le client ne peut déterminer le type d'objet à créer qu'à l'exécution
* Il y a une volonté de centraliser la création des objets

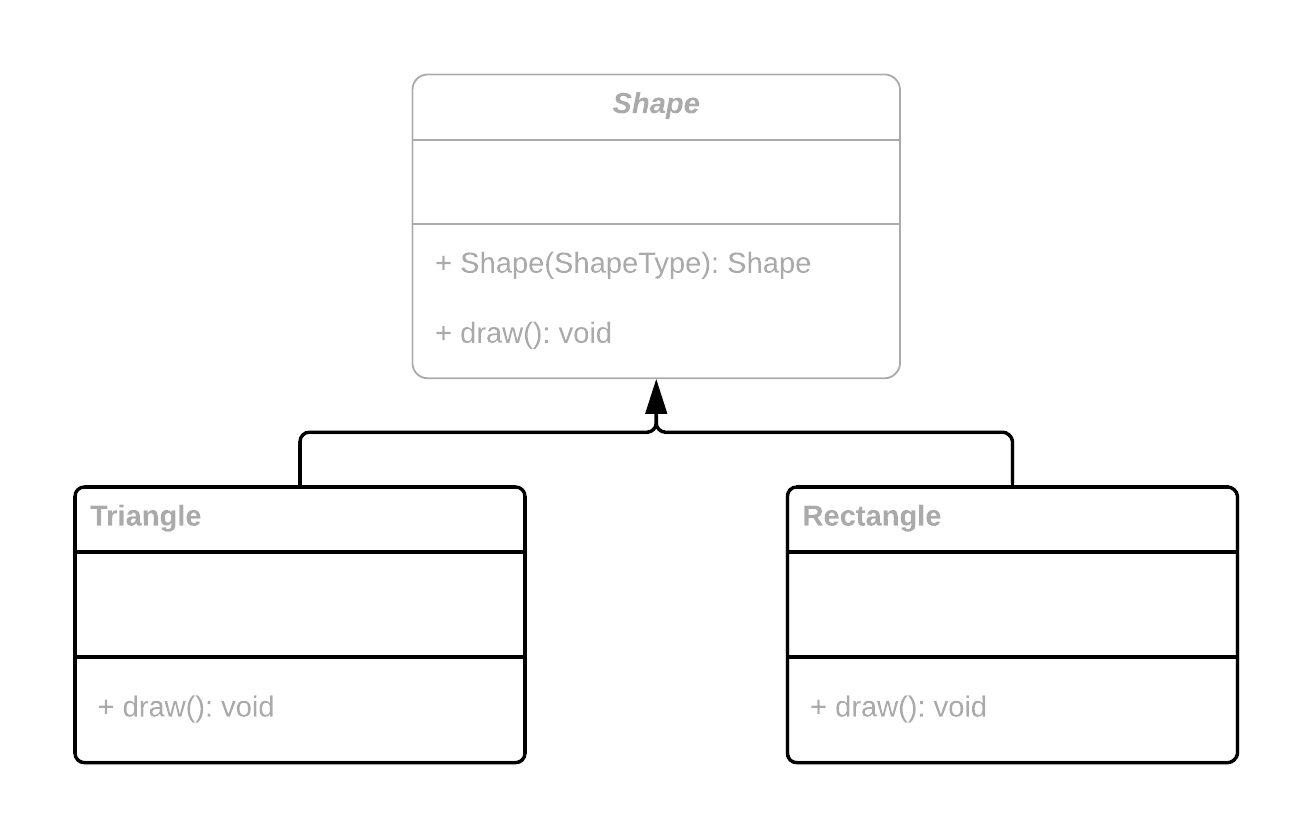
L'utilisation d'une fabrique permet de rendre l'instanciation d'objets plus flexible que l'utilisation de l'opérateur d'instanciation new.

Ce design pattern peut être implémenté sous plusieurs formes dont les deux principales sont :

* Déclarer la fabrique abstraite et laisser une de ses sous-classes créer l'objet
* Déclarer une fabrique dont la méthode de création de l'objet attend les données nécessaires pour déterminer le type de l'objet à instancier

Il est possible d'implémenter la fabrique sous la forme d'une classe abstraite et de définir des sous-classes chargées de réaliser les différentes instanciations.

**Exemple :**



The Shape class will act as the factory and the interface for the example, while Triangle and Rectangle are examples of concrete products of the factory. The products implement the factory's interface, providing concrete implementations of the draw() method.

Let's see what it looks like as [Dart](https://dart.dev/) code:

enum ShapeType {

triangle,

rectangle

}

abstract class Shape {

factory Shape(ShapeType type) {

switch (type) {

case ShapeType.triangle: return Triangle();

case ShapeType.rectangle: return Rectangle();

default: return null;

}

}

void draw();

}

class Triangle implements Shape {

@override

void draw() {

print("TRIANGLE");

}

}

class Rectangle implements Shape {

@override

void draw() {

print("RECTANGLE");

}

}

We start by creating an enum that makes it easy for client code to request a particular type of shape. The list of shapes can be expanded to support more types.

Next comes the object factory, which in this case takes the form of an abstract class called Shape. Shape has a factory constructor that acts as the factory method for this pattern. It's responsible for creating shapes of the requested type. The class is marked abstract to disallow direct instantiation of Shape, since the class has no implementation for the draw() method. Dart factory constructors act like static functions that just happen to have the same name as the housing class, and they don't necessarily return an instance of that same class (though they must return a related type). Using this syntax, we can keep the client code elegant, effectively concealing the use of the Factory Method pattern from its users. A switch statement is utilized to return the appropriate concrete shape, returning null if an invalid type is passed in. The class ends with an unimplemented declaration of a draw() method, present only to establish an interface that all shapes must implement.

The Triangle and Rectangle classes each implement the Shape interface by overriding the draw() method, as indicated by the @override metatag, which is optional but recommended as a form of self-documentation. The factory method in the Shape class cannot return a class that fails to correctly implement this interface.

Using the factory might look something like the following:

final shape1 = Shape(ShapeType.triangle);

final shape2 = Shape(ShapeType.rectangle);

shape1.draw();

shape2.draw();

Through the magic of polymorphism, the correct version of draw() for each shape will be called. Both shape1 and shape2 are of type Shape, but one is actually a Triangle and the other is a Rectangle. It would be possible to create a List<Shape> containing a mix of shape types, and the draw() method could be called for each without the caller needing to know each element's true type.

Next, we'll look at how this pattern could be used in a real Flutter application.