

Script TedTalk Robin Monseré

Dit zijn de notities voor mijn Ted Talk.

Dit is de Canvas presentatie:

https://www.canva.com/design/DAGFsNzr1Po/xvAjWEhgl3v4CEBfztgzJA/view?utm_content=DAGFsNzr1Po&utm_campaign=designshare&utm_medium=link&utm_source=editor

Clean Architecture en SOLID in Flutter

Robin Monseré



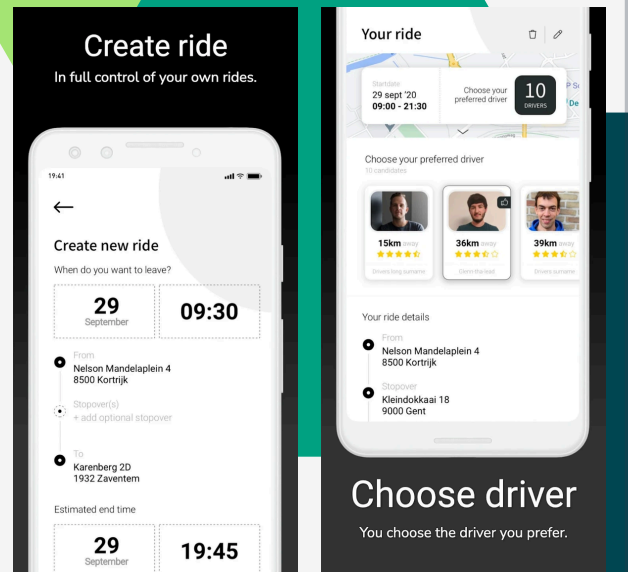
Beste juryleden,

Vandaag presenteer ik mijn bachelorproef over het optimaliseren van de ontwikkeling van Flutter projecten met Clean Architecture en SOLID principes.



Flutter bij Get Driven

- Chauffeurs
- Klanten
- Backoffice



Ik loop stage bij het bedrijf Get Driven, waar ze Flutter gebruiken om drie apps te ontwikkelen: een app voor chauffeurs, een app voor klanten en een interne backoffice app. Aan het begin van het semester was ik op zoek naar een goed onderwerp voor mijn bachelorproef. Mijn stagebegeleider stelde voor om onderzoek te doen naar de toepassing van Clean Architecture en SOLID principes in Flutter. Dit leek me een interessant idee omdat ik al ideeën had over hoe ik dit kon aanpakken.

Verloop van de bachelorproef

Onderzoek

Onderzoek naar
Clean Architecture
en SOLID principes

Experiment

SpaceGaze applicatie
creëren om de verschillen
te bepalen

Conclusie

Persoonlijke conclusies en
conclusies halen uit het
experiment

Voordat ik met mijn experimenten begon, heb ik onderzoek gedaan naar Clean Architecture en SOLID principes. Dit hielp me om de theorie en de praktische toepassingen ervan beter te begrijpen.

In deze presentatie bespreek ik wat voor voorgaand onderzoek ik gedaan heb en hoe ik mijn experiment heb uitgevoerd.

Met daarna mijn conclusie

SOLID

Single Responsibility Principle

Slechts één verantwoordelijkheid

Open/Closed Principle

Open voor uitbreiding, gesloten voor wijzigingen

Liskov substitution principle

Subklasse vervangen door superklasse zonder problemen

Voor ik begon met mijn effectief experiment, heb ik eerst onderzoek gedaan naar wat Clean Architecture en SOLID principes eigenlijk zijn. Om beter te verstaan over wat mijn bachelorproef gaat, zal ik deze ook nog eens kort uitleggen. Een meer in depth uitleg is te vinden in mijn documentatie rapport.

De SOLID principes zijn 5 object georiënteerde ontwerpprincipes die helpen met het creëren van onderhoudsvriendelijke projecten.

De S in SOLID staat voor Single Responsibility Principle, ofwel SRP. Dit principe stelt dat elke klasse of module in een softwareprogramma slechts één reden zou moeten hebben om te veranderen. Dit betekent dat elke klasse of module slechts één taak of verantwoordelijkheid moet hebben en zich daar volledig op moet focussen. Hierdoor wordt voorkomen dat klassen te complex worden door het mixen van verschillende functies of verantwoordelijkheden.

Het Open/Closed Principle, aangeduid met de 'O', stelt dat zoals klassen, modules, functies, etc. open moeten staan voor uitbreiding, maar gesloten moeten zijn voor wijzigingen. Dit betekent dat het mogelijk moet zijn om het gedrag van een module of klasse uit te breiden zonder de bestaande code van die module of klasse te wijzigen.

De 'L' in SOLID is voor de Liskov Substitution principle, LSP is een ontwerpprincipe dat stelt dat objecten van een subklasse moeten kunnen worden vervangen door objecten van de superklasse zonder dat er problemen oplopen in het programma.

SOLID

Interface Segregation Principle

Specifieke interface voor elke functionaliteit

Dependency Inversion Principle

High-level modules niet afhankelijk van low-level modules

De 'I' staat voor Interface Segregation Principle, dat ontwikkelaars aanspoort om specifieke interfaces te gebruiken in plaats van één algemene interface. Dit principe stelt dat een klasse niet gedwongen moet worden om interfaces te implementeren die ze niet gebruikt, wat leidt tot een schonere en meer modulaire code.

Tot slot staat de 'D' voor Dependency Inversion Principle. Dit principe stelt dat high-level modules niet afhankelijk moeten zijn van low-level modules, maar beide zouden moeten afhangen van abstracties. Bovendien zouden deze abstracties niet afhankelijk moeten zijn van details, maar de details zouden afhankelijk moeten zijn van abstracties. Dit leidt tot een minder directe koppeling tussen softwarecomponenten en verbetert de mogelijkheid om componenten te hergebruiken en te onderhouden.

Dit kan ingewikkeld klinken, dus leg ik het even uit aan de hand van een voorbeeld.

Voorbeeld

```
class LaunchRepository implements LaunchRepositoryInterface {  
    @override  
    Stream<List<Launch>> get upcomingLaunchesStream =>  
        _upcomingLaunchesController.stream;  
  
    @override  
    Stream<List<Launch>> get previousLaunchesStream =>  
        _previousLaunchesController.stream;  
  
    @override  
    void fetchUpcomingLaunches() async {  
        // implement code to fetch data  
    }  
  
    @override  
    void fetchPreviousLaunches() async {  
        // implement code to fetch data  
    }  
    ...  
}
```

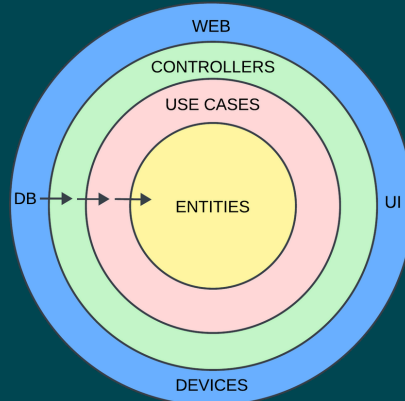
mijn applicatie haalt data op van een publieke API. Ik heb een interface gemaakt die de nodige methodes bevat om de recente en de lanceringen in de toekomst op te halen. Deze interface definieert alleen de operaties, maar niet hoe deze operaties worden uitgevoerd. (toon code snippet 1)

Vervolgens implementeer je deze interface in een klasse die de daadwerkelijke de calls naar de api uitvoert. Dit kan met extra functies (code snippet 2)

In de hogere lagen van je applicatie, zoals de business logica of de UI controllers, refereren je enkel naar de interface. Dit betekent dat de hogere lagen afhankelijk zijn van de abstractie, en niet van de concrete implementatie. Dit zorgt ervoor dat als je besluit om een andere api te gebruiken, wel of niet te cachen, je alleen de implementatie hoeft aan te passen zonder de hogere lagen van je code te beïnvloeden. Je zou dan een nieuwe klasse kunnen creëren, zoals CachedLaunchRepository, die ook de interface implementeert.

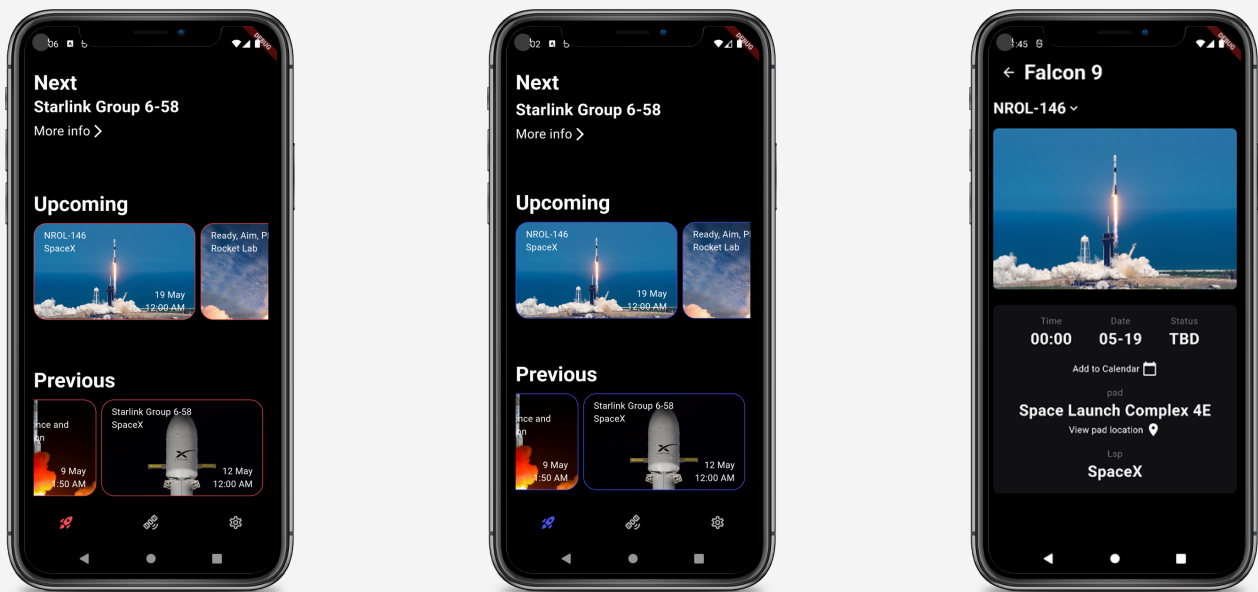
Clean Architecture

Systeem opgedeeld in concentrische lagen



Clean Architecture is een softwarearchitectuurontwerp dat stelt dat het systeem in lagen is opgedeeld. Deze lagen zijn deel van een cirkel en het doel is dat de Afhankelijkheden altijd van buiten naar binnen gericht zijn, oftewel van minder belangrijke details naar meer fundamentele aspecten van de applicatie. Elk van deze lagen heeft een specifieke verantwoordelijkheid en rol binnen het systeem.

Experiment



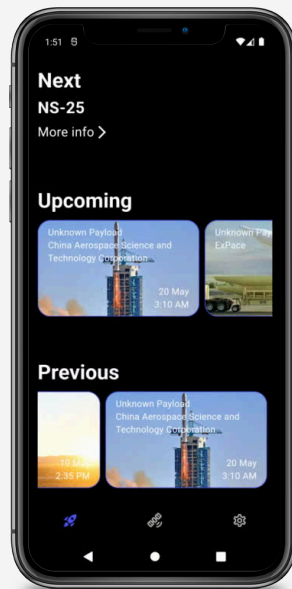
Om het experiment uit te voeren had ik een app nodig die al deze principes kon gebruiken. Achteraf gezien was de keuze van mijn app toch niet optimaal om elk aspect te testen. De app was iets te klein voor dit experiment.

De app die ik gecreëerd heb haalt de meest recente data op in verband met raket lanceringen en toont dit aan de gebruiker. Om het verschil te testen tussen het wel en niet gebruik van Clean Architecture en SOLID principes, heb ik de app 2 keer gemaakt, 1 maal zonder en 1 keer met Clean Architecture en SOLID. Uiterlijk is er zo goed als geen verschil tussen beide apps, dus heb ik het accent kleur blauw gemaakt bij de app met en rood bij de app zonder.

Naast de homepage is er ook een pagina met meer gedetailleerde uitleg over een lancering.

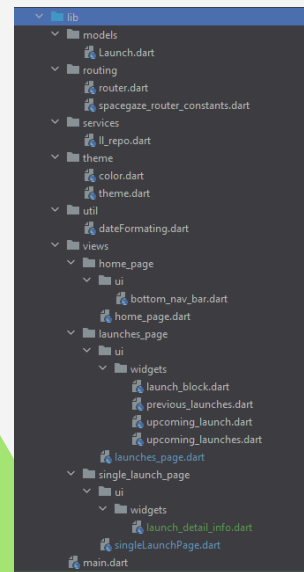
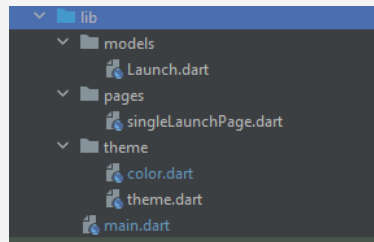
De broncode voor beide apps staan op github.

Experiment



Verschillen

- Ontwikkeltijd
- Leercurve
- Folderstructuur
- Prestaties
- App-grootte



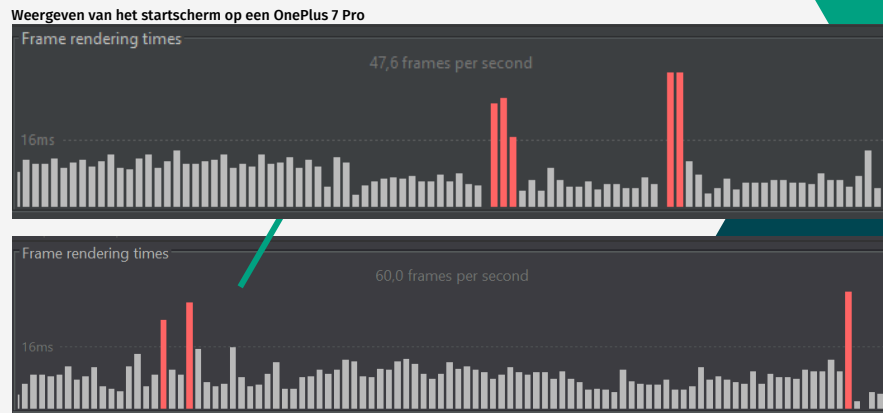
Voor de gebruiker is er zo goed als geen verschil, maar voor de ontwikkelaar is er een wereld van verschil. Ik was begonnen met de app zonder SOLID en Clean Architecture, dit was de app die het snelst klaar was en bij de eerste indruk ook het gemakkelijkst om te maken. De 2de app duurde heel wat langer en had ik ook wat meer moeite mee, omdat ik moest zorgen dat alles correct was voor SOLID en Clean Architecture, de leercurve was duidelijk groter. Ik heb later aan een medestudent gevraagd om de broncode van beide apps te bekijken, hij gaf aan dat de app met Clean Architecture en SOLID veel eenvoudiger was om te verstaan. Dit komt natuurlijk vooral door het Single Responsibility Principle (SRP), dat ervoor zorgt dat elke klasse of module slechts één verantwoordelijkheid draagt, wat het geheel overzichtelijker maakt.

Nog een groot verschil tussen beide apps is het verschil in folderstructuur. Eén van de belangrijkste aspecten van zowel SOLID als Clean Architecture is de nadruk op de scheiding van verantwoordelijkheden. Elk onderdeel van de software heeft een specifieke verantwoordelijkheid en is geïsoleerd van anderen. Dit betekent dat in plaats van enkele monolithische modules, de applicatie wordt opgesplitst in meerdere kleinere, beheersbare onderdelen. Elk van deze onderdelen kan een eigen map of submap vereisen om de code overzichtelijk en onderhoudbaar te houden.

Dit betekent ook dat in grotere projecten, het veel makkelijker is om code terug te vinden, en zorgt ervoor dat verschillende teams tegelijkertijd aan verschillende onderdelen van de applicatie kunnen werken zonder elkaar in de weg te zitten. Deze structuur zorgt er ook voor dat het hergebruiken van code eenvoudiger is en om veranderingen door te voeren zonder onverwachte bijwerkingen in andere delen van de applicatie.

Verschillen

- Ontwikkeltijd
- Leercurve
- Folderstructuur
- Prestaties
- App-grootte



Ook al hebben beide applicaties exact dezelfde functionaliteiten, er is wel een verschil te merken in de prestaties tussen beide applicaties. Dit verschil wordt duidelijk als we kijken naar de tijd die er nodig is om een frame te renderen.

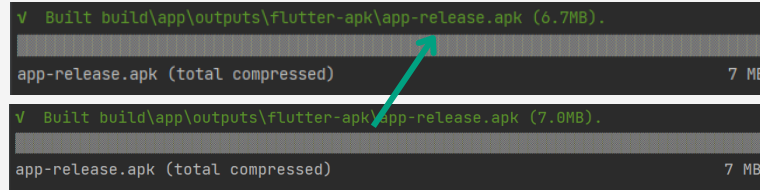
Volgens de officiële documentatie van Flutter, is het de bedoeling om 60 fps te bereiken. Dit betekent dat een frame 16ms heeft om te renderen, elke frame die langer duurt dan 16ms, wordt in het rood aangeduid op de grafiek.

Beide grafieken zijn snapshots van wanneer de homepage gerenderd wordt op een OnePlus 7 pro.

Zouls je kan zien, heeft de app met Clean Architecture weinig moeite om aan 60 fps te komen, terwijl dat niet lukt bij de andere app.

Verschillen

- Ontwikkeltijd
- Leercurve
- Folderstructuur
- Prestaties
- App-grootte



```
v Built build\app\outputs\flutter-apk\app-release.apk (6.7MB).  
app-release.apk (total compressed) 7 MB  
v Built build\app\outputs\flutter-apk\app-release.apk (7.0MB).  
app-release.apk (total compressed) 7 MB
```

Persoonlijk dacht ik eerst dat door het gebruik van Clean Architecture en SOLID dat er ook een zichtbaar verschil zou zijn in de grootte van de app, echter, dat is niet het geval, de app is zelfs 0.3MB groter. Nu, dit komt omdat beide apps veel te klein zijn om hier enige verschil tussen te zien.

Conclusie

- Duidelijk voordeel
- Meer features nodig



Het voordeel van SOLID en Clean Architecture is duidelijk voor grotere projecten, en zeker ook aangeraden voor kleinere projecten. De leercurve voor Clean Architecture en SOLID is in het begin groter, maar die tijd win je terug naarmate je later uitbreidingen wil doen, of voor andere ontwikkelaars die het project moeten verstaan.

De keuze van app was goed om persoonlijke bevindingen te maken, maar om echt inzicht te krijgen in het gebruik van Clean Architecture en SOLID, was deze app toch te klein en had niet voldoende features.

Bedankt om te luisteren naar mijn Ted talk over het nut van Clean Architecture en SOLID in Flutter.