devfest

Dependency Inversion: Breaking Dependencies Between Components

Property of
Erico Darmawan Handoyo

Google Developer Groups
Bogor

Erico Darmawan Handoyo

- Parahyangan Catholic University
- Bandung Institute of Technology

- Lecturer @ Maranatha Christian University
- The Most Awesome Spectacular Legendary Programming Book Author
- Flutter tutorial creator @ YouTube
- Flutter Trainer

- Traveling ✈️✈️✈️✈️✈️
- Eating 🍗🍗🍗🍗🍗
- Watching movies 🎬🎬🎬🎬
- Anything costs money 💵💵💵💵💵
- ~~Coding~~ 💻➡️⚪💎

Property of
Erico Darmawan Handoyo

Google Developer Groups

youtube.com/@ericodarmawan

s.id/fb-ericodh

s.id/komunitas-flutter

s.id/ig-ericodh

s.id/flixid

s.id/in-ericodh

ericodarmawan.com

Introduction

1. Every project owner wants his project to last a long time.
2. Every project owner wants to build his project quickly.
3. Every project owner wants to keep production costs low.
4. User's needs change.
5. Technology is always changing.
6. Large projects involve more than 1 developer.

1. Your project must support team work.
2. Your project must be easy to develop, extend, and maintain.

Separation of Concerns

Combining UI & data dayer is a bad practice

1. It's difficult to be developed by a group of developers.
2. Each system component is interconnected with each other.
    a. Updating codes in one place may affect the other codes.
    b. It's hard to test.

Cannot be done in parallel

Code for UI → Code for Data

```
ElevatedButton(
  onPressed: () async {
    try {
      await Supabase.instance.client.auth.signInWithPassword(
          email: _emailController.text,
          password: _passwordController.text);
    } catch (e) {
      // do something with the error
    }
  },
  child: const Text('Login'),
),
```

1. It's **hard** to **divide** the **work**, because all the **codes** are in **one place**.
2. **Need UI to test** the database related code.
3. If there is **any changes** to **database-related codes**, you **must update all UI** that contains the codes. E.g. The syntax of getting Supabase instance change.

Many duplication of codes will risk more bugs.

Cannot be done in parallel

Code for Data → Code for UI

```
ElevatedButton(
  onPressed: () async {
    try {
      await SupabaseAuthentication().login(
          email: _emailController.text,
          password: _passwordController.text);
    } catch (e) {
      // do something with the error
    }
  },
  child: const Text('Login'),
),
```

1. **Need to wait** the **database related class** (SupabaseAuthentication) to use in the UI.
2. **Changes** in **database** related class **may affect** the **UI** related codes. E.g. Changing database, changing method in the database-related class.

Changing code that is already correct may risk introducing bugs.

Separation of concerns

Separate your codes / system components based on its purpose and its possibility of change.

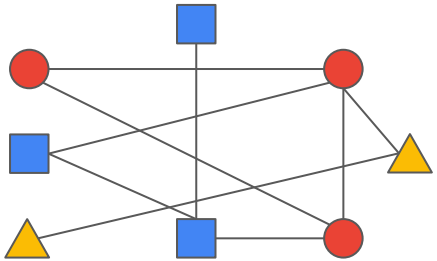
1. Each system components can be done separately in the same time.
2. Updating some codes will give no effect to the other codes.
3. Replacing a component will not affect the other part of the system.

Property of
Erico Darmawan Handoyo

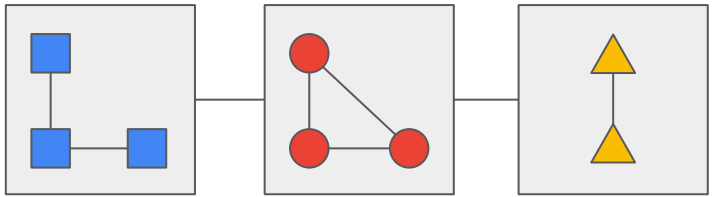
A Good Software: High cohesion, low coupling

Cohesion: the measure of the degree to which the elements of a module are functionally related to each other.

Coupling: the measure of the degree of interdependence between the modules.



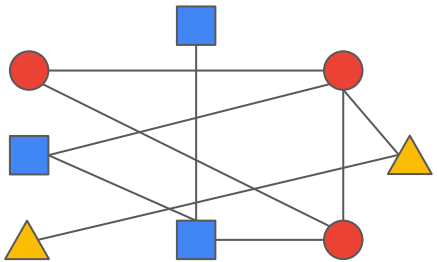
Bad



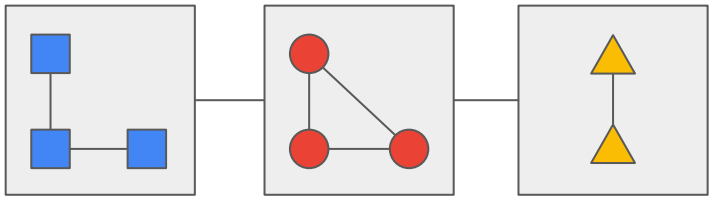
Good

How to do separation of concerns

1. Group all components that function related to be one group / layer / module.
2. Separate those groups / layers / modules with an abstraction / contract between them as a reference.
3. Each layer of the system should depends on the abstraction / contract, not the other layers.

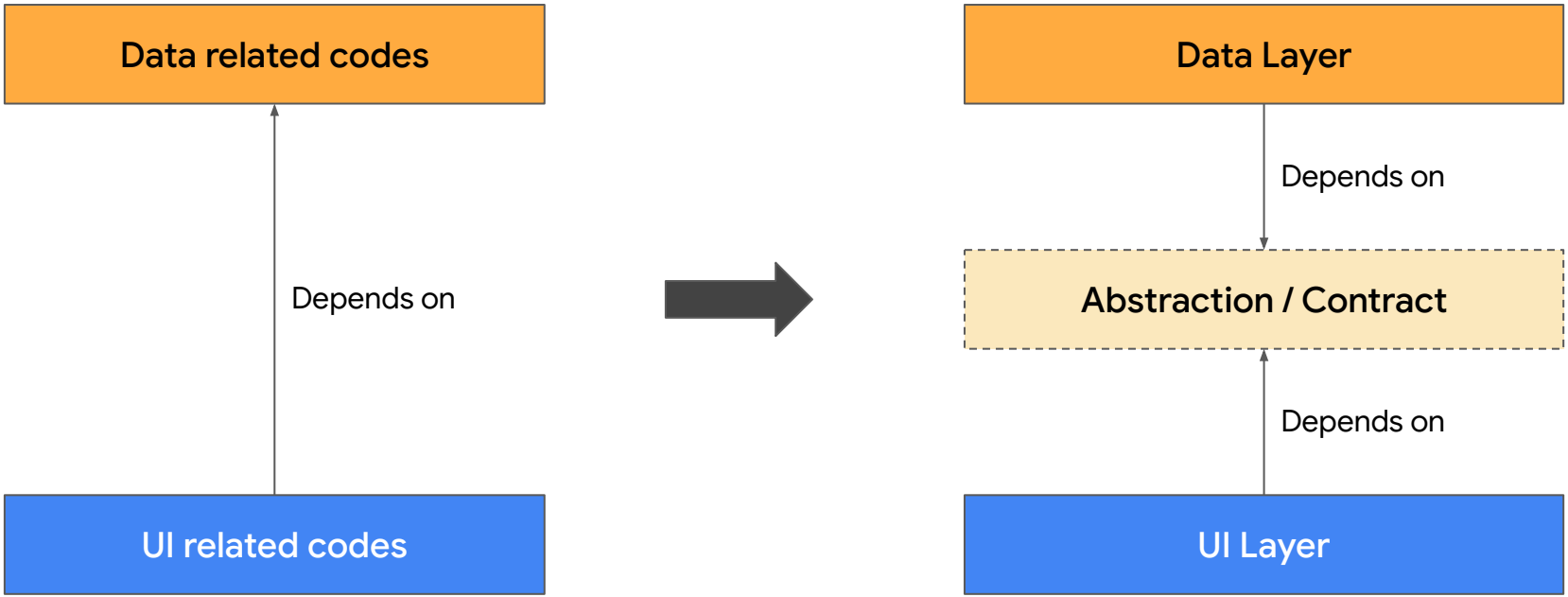


Bad



Good

The simplest separation of concerns



Flutter Showcase: Simple List to Do

Showcase 1 - List To Do (Bad ver. 1)

1. It's **difficult** to **distribute** the **task**.
2. It's **hard to test** → **Cannot be sure** that every important code is **correct** → **Difficult** to **find** the **cause of a bug** / error.
3. If you want to **change** the **database**, you have to **re-code** almost **everything** → will **risk introducing** new **bugs**.
4. It takes too **much time** and **effort** every time you **update** / **fix** your application.

Showcase 2 - List To Do (Bad ver. 2)

1. You have to **wait** the **data related codes** to finish your UI related codes.
2. **Changing** the **data related codes** may **affect** the **UI related code**. E.g. Changing the method name.
3. **Changing** the **data source** is **troublesome**.
4. If the **new database** has **differences** with the **old database**, you must **change** the **data model** and **UI codes**. E.g. Supabase table auto-id is an integer while Firebase collection id is a String.

Showcase 3 - List To Do with UI-Data separation

1. You can **complete UI code without waiting** for data-related code to complete.
2. You can **change** the **data source easily**.
3. You can **easily test** your data classes to make sure everything is working properly.
4. Your **UI doesn't care** about the **data source**. Each **data source** class **must meet** the **requirements** described in the **abstraction** (interface).
5. The **difference** between data sources **doesn't effect** the data model used in the app.

Thank you

youtube.com/@ericodarmawan

s.id/fb-ericodh

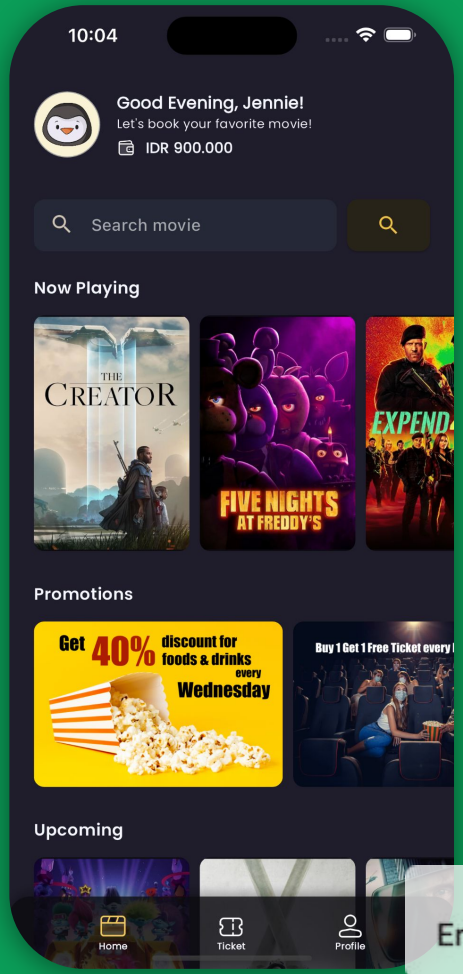
s.id/komunitas-flutter

s.id/ig-ericodh

s.id/flixid

s.id/in-ericodh

ericodarmawan.com



SPECIAL OFFER 50% OFF DISCOUNT

flixid

- Separation of Concerns
- Clean Architecture
- Riverpod State Management
- Firebase Auth, Storage, & Firestore

🔗LINK🔗
s.id/flixid

割KODE PROMO割
DEVFEST-BOGOR2023

Property of
Erico Darmawan Handoyo