LeanCode

Mobile architecture

Mateusz Wojtczak

What is *architecture*?

Software architecture is the set of structures needed to **reason** about a software system and the **discipline** of creating such structures and systems. Each structure comprises software elements, relations among them, and properties of both elements and relations.[1][2]

What is *architecture*?

“the shared understanding that the expert developers have of the system design”

“the decisions you wish you could get right early in a project”

“Architecture is about the important stuff. Whatever that is.”

~Ralph Johnson

Step 1 - everything in a widget

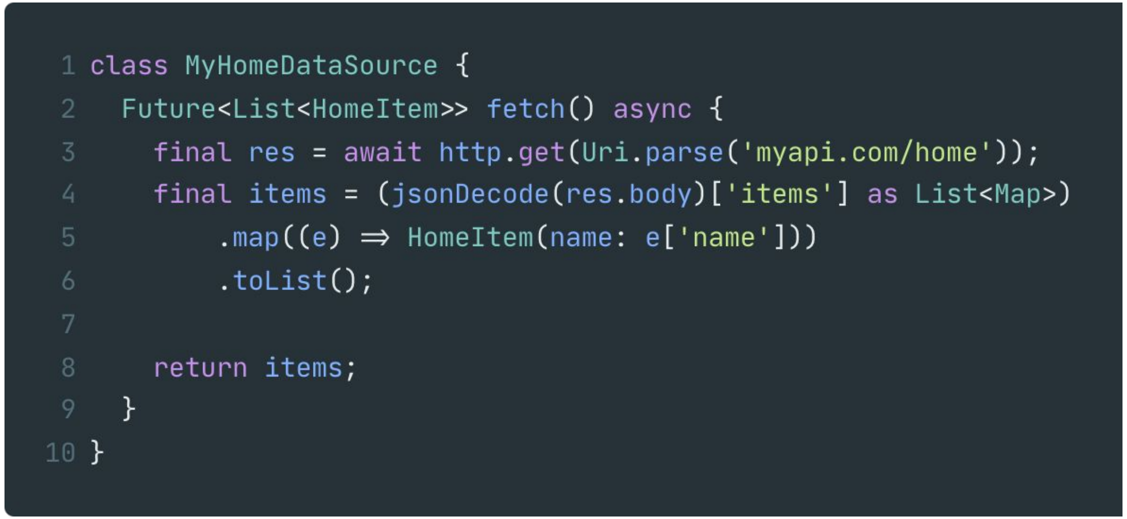
```dart
class _MyHomePageState extends State<MyHomePage> {
  List<HomeItem> homeItems = [];

  @override
  void initState() {
    super.initState();

    () async {
      final res = await http.get(Uri.parse('myapi.com/home'));
      homeItems = (jsonDecode(res.body)['items'] as List<Map>)
          .map((e) => HomeItem(name: e['name']))
          .toList();
    }();
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      body: ListView.builder(
        itemCount: homeItems.length,
        itemBuilder: (context, i) => ListTile(
          title: Text(homeItems[i].name),
        ),
      ),
    );
  }
}
```

Step 2 - UI in widget + everything else in Bloc

```dart
class MyHomeCubit extends Cubit<MyHomeState> {
  MyHomeCubit() : super(MyHomeState(items: []));

  Future<void> fetch() async {
    final res = await http.get(Uri.parse('myapi.com/home'));
    final items = (jsonDecode(res.body)['items'] as List<Map>)
        .map((e) => HomeItem(name: e['name']))
        .toList();

    emit(MyHomeState(items: items));
```

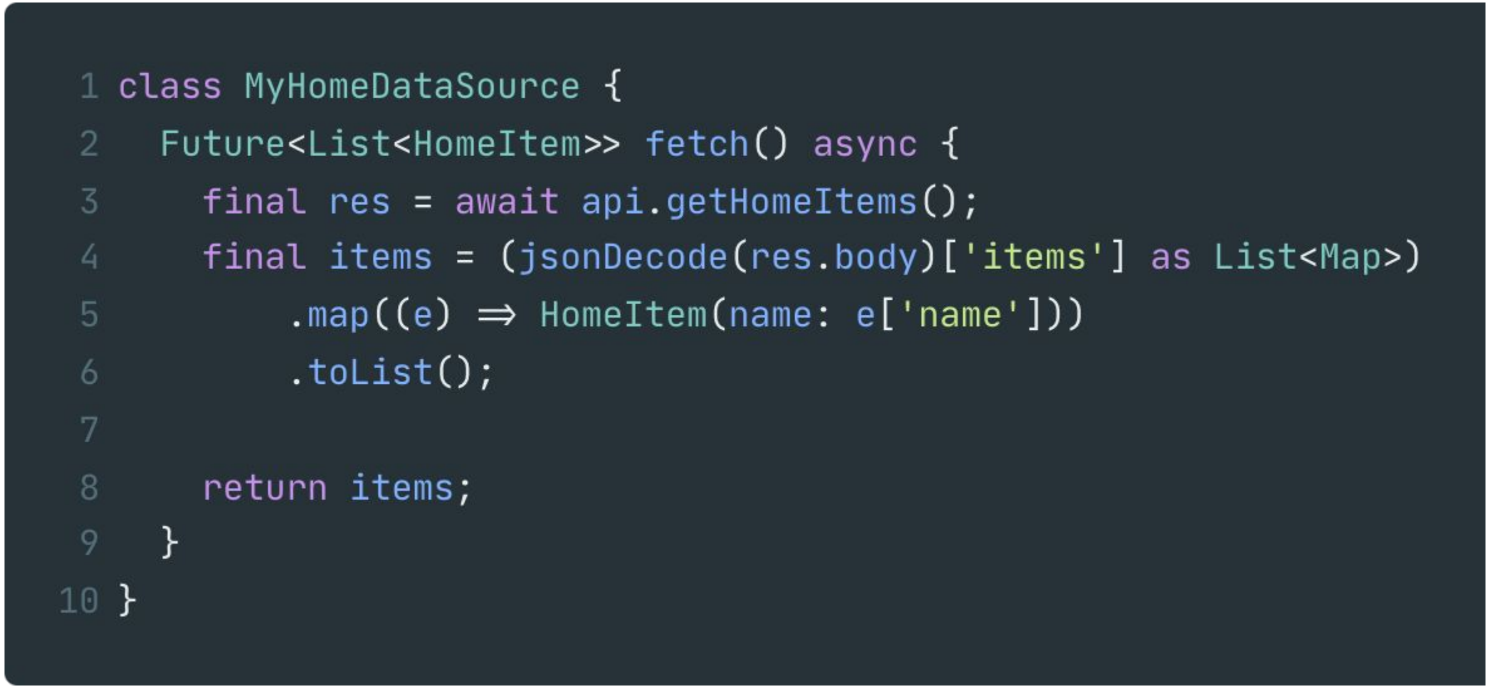
```dart
class _MyHomePageState extends State<MyHomePage> {
  List<HomeItem> homeItems = [];

  @override
  void initState() {
    super.initState();

    bloc.fetch();
  }
  ...
}
```

Step 3 - Let’s add another layer for data



```dart
class MyHomeDataSource {
  Future<List<HomeItem>> fetch() async {
    final res = await http.get(Uri.parse('myapi.com/home'));
    final items = (jsonDecode(res.body)['items'] as List<Map>)
        .map((e) => HomeItem(name: e['name']))
        .toList();

    return items;
  }
}
```

Step N - Let’s extract this responsibility



```dart
class MyHomeDataSource {
  Future<List<HomeItem>> fetch() async {
    final res = await api.getHomeItems();
    final items = (jsonDecode(res.body)['items'] as List<Map>)
        .map((e) ⇒ HomeItem(name: e['name']))
        .toList();

    return items;
  }
}
```

We build digital products.

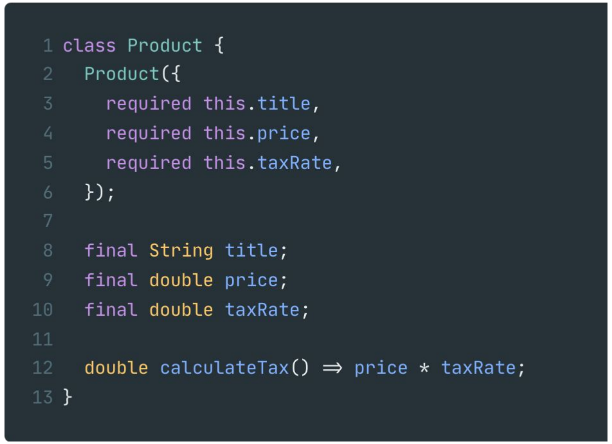
LeanCode

SOLID

SRP - Single Responsibility Principle

"A module should be responsible to one, and only one, actor."[1]

“A class should have one, and only one, reason to change.”



```dart
class Product {
  Product({
    required this.title,
    required this.price,
    required this.taxRate,
  });

  final String title;
  final double price;
  final double taxRate;

  double calculateTax() => price * taxRate;
}
```

OCP - Open-Closed Principle

"software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification"[1]

```dart
sealed class Shape {
  double get area {
    return switch (this) {
      Rectangle r => r.width * r.height,
      Circle c => c.radius * c.radius * pi,
    };
  }
}

class Rectangle extends Shape {
  Rectangle({required this.width, required this.height});

  final double width;
  final double height;
}

class Circle extends Shape {
  Circle({required this.radius});

  final double radius;
}
```

LSP - Liskov Substitution Principle

“objects of a superclass shall be replaceable with objects of its subclasses without breaking the application”

This is kind of an informal rule.
It’s not always wrong to have a different behavior in a subclass.

Bad example

```
public class Bird{
    public void fly(){}
}
public class Duck extends Bird{}
```

The duck can fly because it is a bird, but what about this:

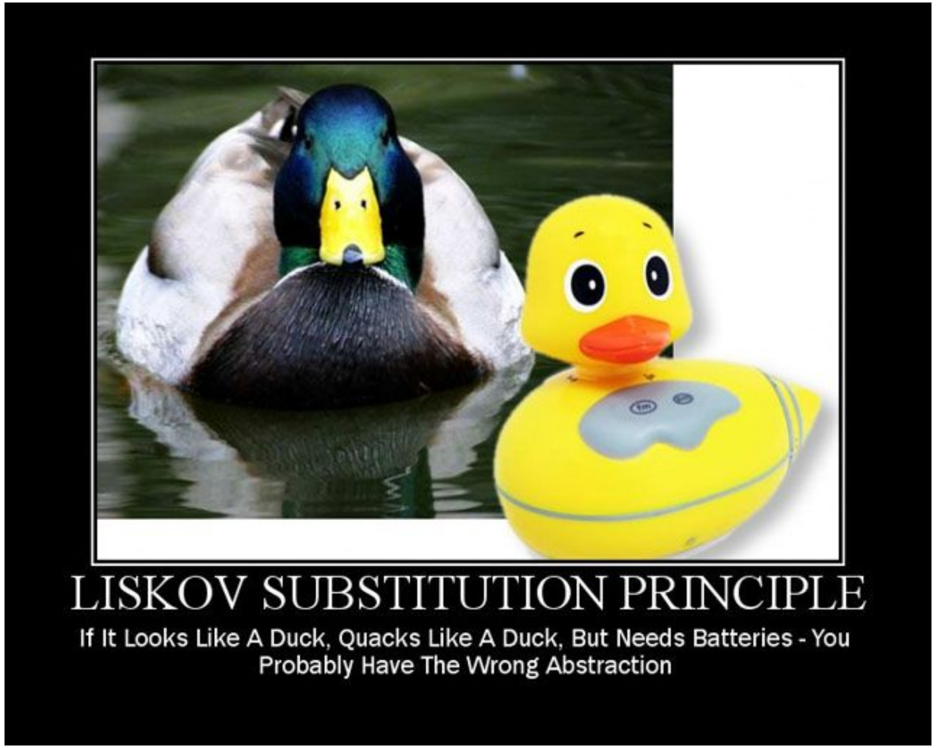
```
public class Ostrich extends Bird{}
```

Ostrich is a bird, but it can't fly, Ostrich class is a subtype of class Bird, but it shouldn't be able to use the fly method, that means we are breaking the LSP principle.

Good example

```
public class Bird{}
public class FlyingBirds extends Bird{
    public void fly(){}
}
public class Duck extends FlyingBirds{}
public class Ostrich extends Bird{}
```

Source: https://stackoverflow.com/questions/56860/what-is-an-example-of-the-liskov-substitution-principle


LISKOV SUBSTITUTION PRINCIPLE
If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Source: https://stackoverflow.com/questions/56860/what-is-an-example-of-the-liskov-substitution-principle

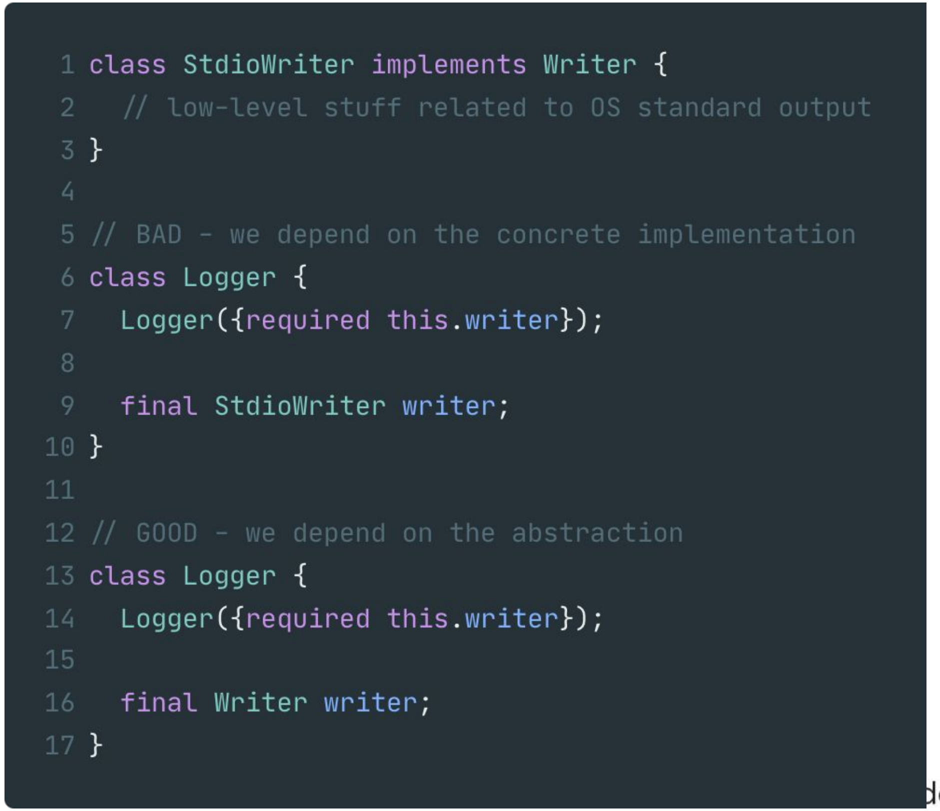
ISP - Interface Segregation Principle

“no code should be forced to depend on methods it does not use.[1]”

```
abstract class Reader {
  String readLine();
}

abstract class Writer {
  bool writeLine(String line);
}

abstract class Stdio implements Reader, Writer {}

class Logger {
  Logger({required this.writer});

  final Stdio writer;
}

class Logger {
  Logger({required this.writer});

  final Writer writer;
}
```

DIP - Dependency Inversion Principle

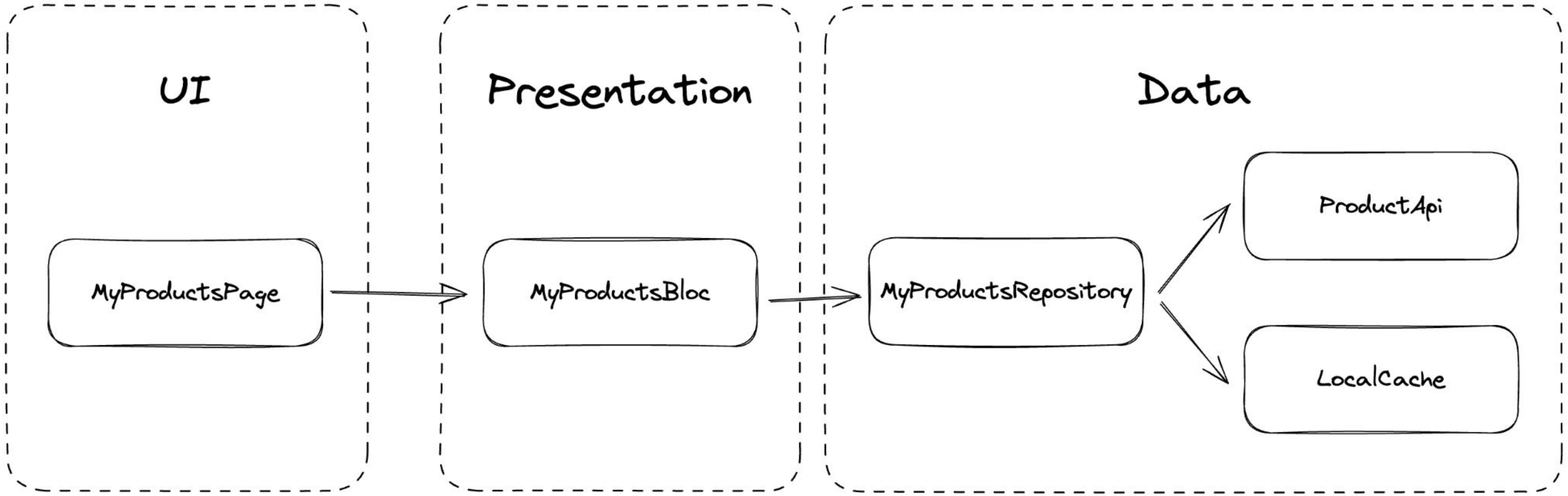
“high level modules should not depend on low level modules; both should depend on abstractions”

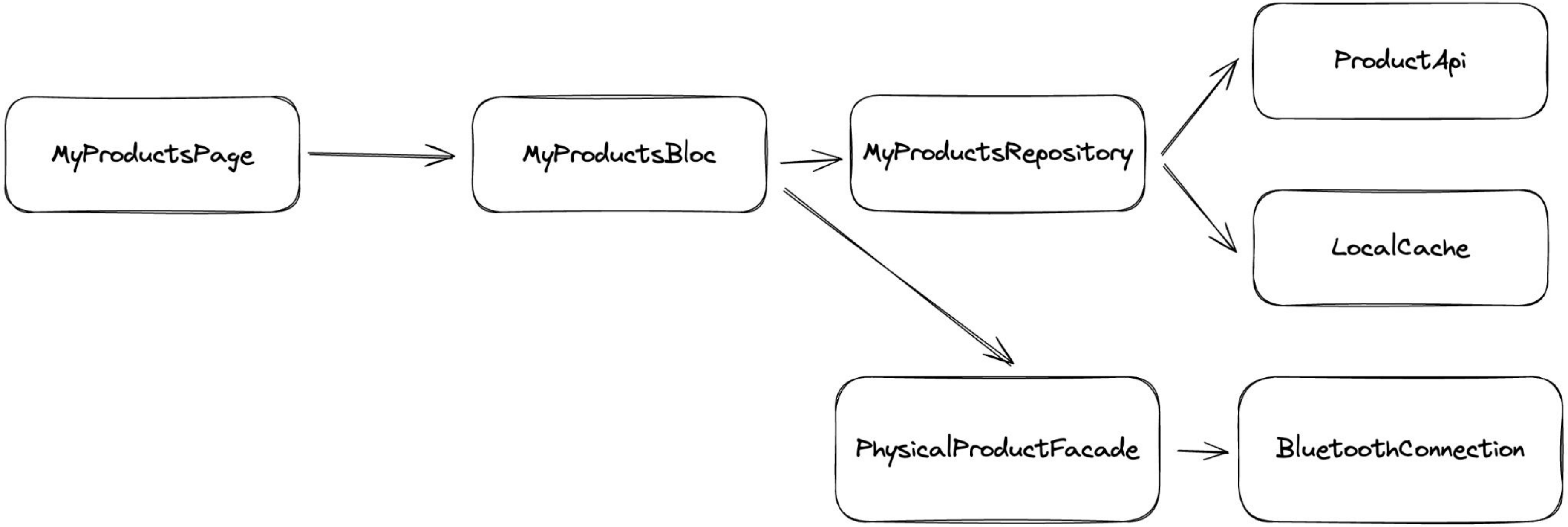


```
class StdioWriter implements Writer {
  // low-level stuff related to OS standard output
}

// BAD - we depend on the concrete implementation
class Logger {
  Logger({required this.writer});

  final StdioWriter writer;
}

// GOOD - we depend on the abstraction
class Logger {
  Logger({required this.writer});

  final Writer writer;
}
```

Minimum Viable Architecture

Architecture is **never** final


UI
Presentation
Data
MyProductsPage
MyProductsBloc
MyProductsRepository
ProductApi
LocalCache


MyProductsPage
MyProductsBloc
MyProductsRepository
ProductApi
LocalCache
PhysicalProductFacade
BluetoothConnection

Tips and Tricks

Easier To Change (ETC)

“Good Design Is Easier to Change Than Bad Design”

~”The Pragmatic Programmer” by Dave Thomas and Andy Hunt

If you have to make a choice between two possible solutions, pick the one that is easier to change.

Don’t Repeat Yourself (DRY)

“Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.”

~”The Pragmatic Programmer” by Dave Thomas and Andy Hunt

Practical application: if you have to use the same code again, you can just copy it, but if it’s a third or more time - you most likely should extract it.

You Ain’t Gonna Need It (YAGNI)

“Always implement things when you actually need them, never when you just foresee that you [will] need them.”

~Ron Jeffries, co-author of eXtreme Programming

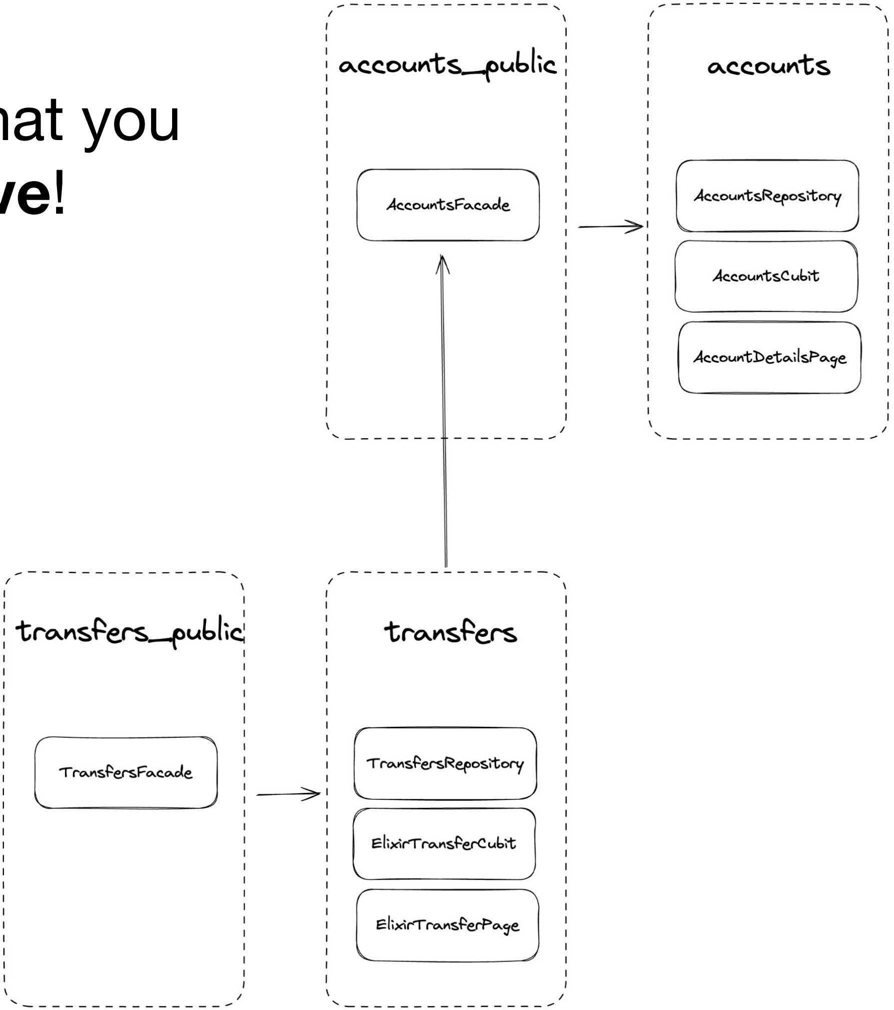
“It is hard for less experienced developers to appreciate how rarely architecting for future requirements / applications turns out net-positive.”

~John Carmack, co-founder of id Software (Quake, Doom, Wolfenstein)

Practical application: if you have to use the same code again, you can just copy it, but if it’s a third or more time - you most likely should extract it.

Real-world example

Think about what you **want to achieve**!



Questions?