Architecture

CS 346 Application Development

https://student.cs.uwaterloo.ca/~cs346

Contents

Introduction

Defining Success

> It doesn't take a huge amount of knowledge and skill to get a program working. Kids in high school do it all the time. The code they produce may not be pretty; but it works. Getting something to work once just isn't that hard.
>
> Getting software "right" is hard. When software is done right, it requires a fraction of the human resources to create and maintain. Changes are simple and rapid. Defects are few and far between. Effort is minimized, and functionality and flexibility are maximized."
>
> — Robert C. Martin

Software Qualities

As users, we can think of features that we want in all of our software [1].

- **Effectiveness**: Our software should be "fit for purpose"
- **Reliability**: It should work as expected most (all) of the time.

As developers, we can add to this list:

- **Flexibility**: we should be able to extend existing functionality or easily add new functionality. The design should accommodate this as a long-term goal.
- **Scalability**: it should be scalable to handle increased demand e.g., more users, more transactions, greater throughput.
- **Reusability**: We should reuse design/code whenever possible and design our solution in a way that fosters reuse (however, beware YAGNI).

You should consider these as goals for any software that you create.

- We may have *even more* qualities that we want to prioritize.

Software Qualities

`Functional requirements` are related to the functionality of your application. These are often what users are talking about in user stories. e.g., “save a file”.

`Non-functional requirements` refer to the qualities of our software. e.g., scalability, robustness, power-consumption, speed, efficiency. These often reflect the architectural decisions that you make.

Both types of requirements can “fall out” of user scenarios or user stories.

- You can log and track them like any other requirement.

> **Tip**
>
> Non-functional requirements can sometimes just “fall out” of conversations with users. e.g., “Retrieving this report takes too long, so I often keep the spreadsheet open and just copy the data myself“.
> - What’s “too long” in this scenario? That sounds like a performance goal.

Why It Matters

What is it?

Software architecture is the “fundamental organization of a software system, encompassing its components, their relationships, and the principles guiding its design and evolution” [1].

In other words, the architecture is the *structure* of your software: what components exist, how they are arranged and how they communicate with one another.

Why is this important?

A solid architecture ensures a system is robust, testable, and adaptable to future changes. Without proper architecture, systems often become difficult to maintain and expensive to upgrade [2]

Architectural decisions are those decisions that will enable you to achieve long-term goals for the life of your software. They are an investment in the future.

Design Principles

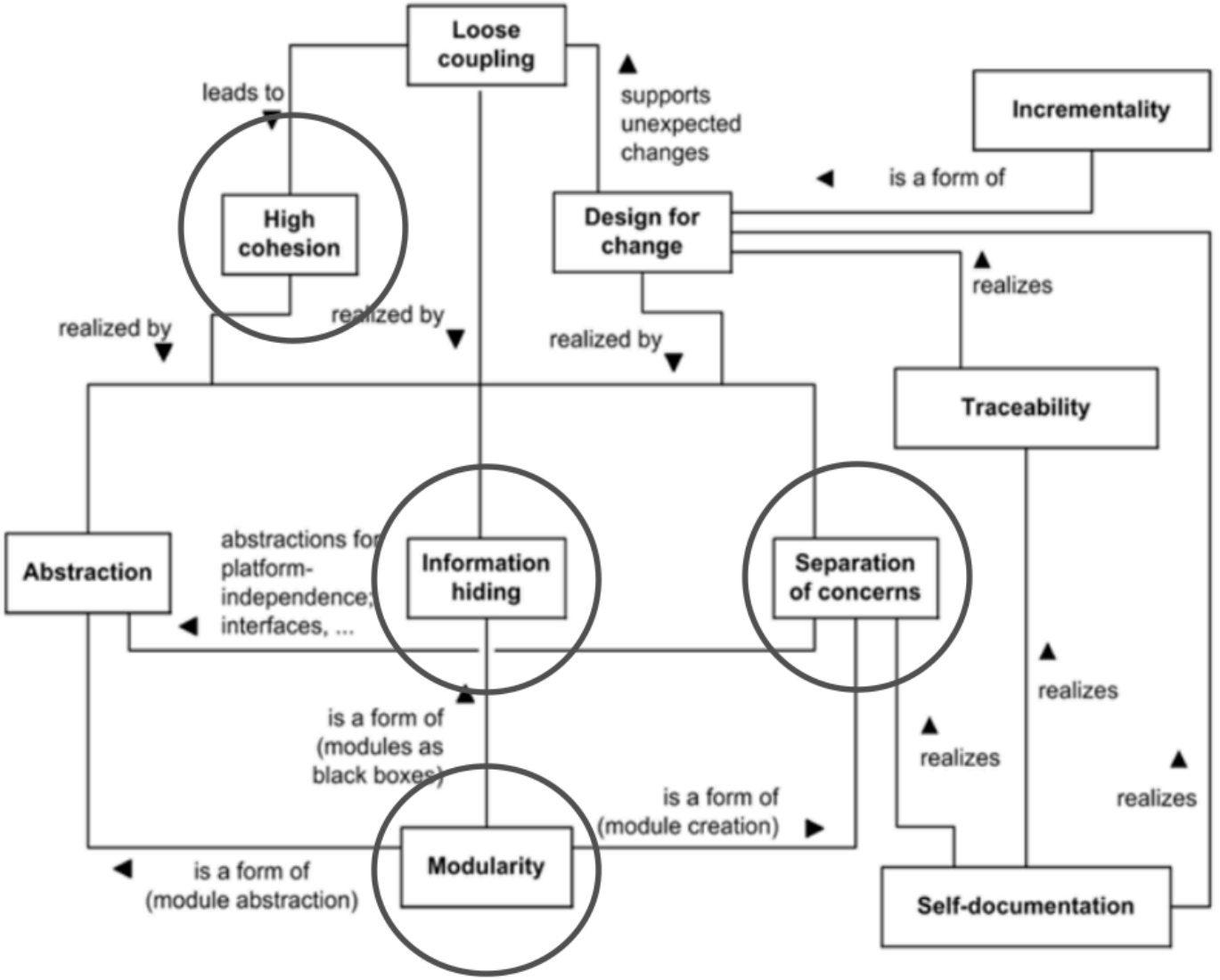


Figure 1: Principles to address flexibility, scalability and robustness.

Separation of Concerns

The most important design principle in any system is separation of concerns: separating your app into methods, classes, files, packages, modules and layers that have clearly defined responsibilities and boundaries [3].

Layers (modules) represent modules or groups of related components.

Components (classes) represent more discrete functionality.

- Each module or component has its own area of responsibility.
- Components or modules are independent of one another.
- Internal state is hidden, and only exposed through a well-designed API.

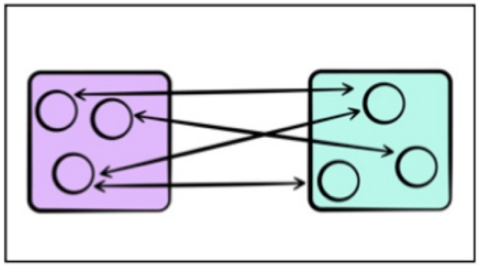
Benefits

- Clearly defined roles results in “cleaner” code that is easier to read, maintain and test.

High Cohesion, Loose Coupling

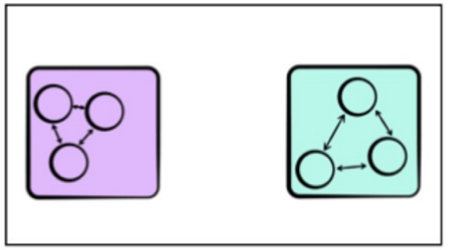
Separation of concerns dictates how components are defined. This principle describes the *relationships* between those components.

- **High cohesion**: Each component should be internally cohesive. That is, it should be self-contained and not rely on any external components for its responsibilities. We strive for high cohesion.
- **Loose coupling**: There should be few dependencies between them. We want our components to be loosely-coupled.



Coupling

Coupling refers to how closely linked components or modules are to each other.



Cohesion

Cohesion is a measure of how closely related the parts of a module are.

Clear separation leads to stable APIs.

- Code is easier to understand and debug.
- It's also easier to make changes in one component without affecting the others.

Single Source of Truth

You should design your application so that there is always a single, trusted source for any data which “owns” that data. In most cases, that will be a database.

This means that your application should have:

- Clearly defined models that manage data e.g., a source for Customer data, or Transaction data, or configuration settings.
- A mechanism to load that data, and keep it in a consistent and trusted state.

> **Tip**
>
> You will likely have multiple data models handling your data, but one of them needs to be “trusted”.
>
> - e.g., a Customer record might exist in the database AND be displayed on-screen simultaneously, but one needs to be “trusted”.
> - i.e., in the case of conflict, which one do you keep?

Drive the UI from Data

Your user interface should be designed to display information based on the data model. Let the data drive the UI.

To achieve this, you should expect expect to have *two data models*:

- a persistant data model, which is your trusted source, and
- an internal data model, which the application and user interface will use.

Keeping two models provides some benefits:

- You can perform validation in the user interface prior to accepting any data changes e.g,. editing a customer record, the UI can ensure that the phone number is valid before updating a record.
- Data isn't lost in the case of an application crashing.
- Your app continues to work in cases when a network connection is intermittent or unavailable.

Obviously we will need a way to push changes from one model to the next.

Structure

Anti-Patterns

Anti-patterns are software design patterns that at-first appear to be beneficial, but result in poor outcomes. You don't want to repeat these.

Big Ball of Mud

A Big Ball of Mud is a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle. These systems show unmistakable signs of unregulated growth, and repeated, expedient repair. Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated. [4].

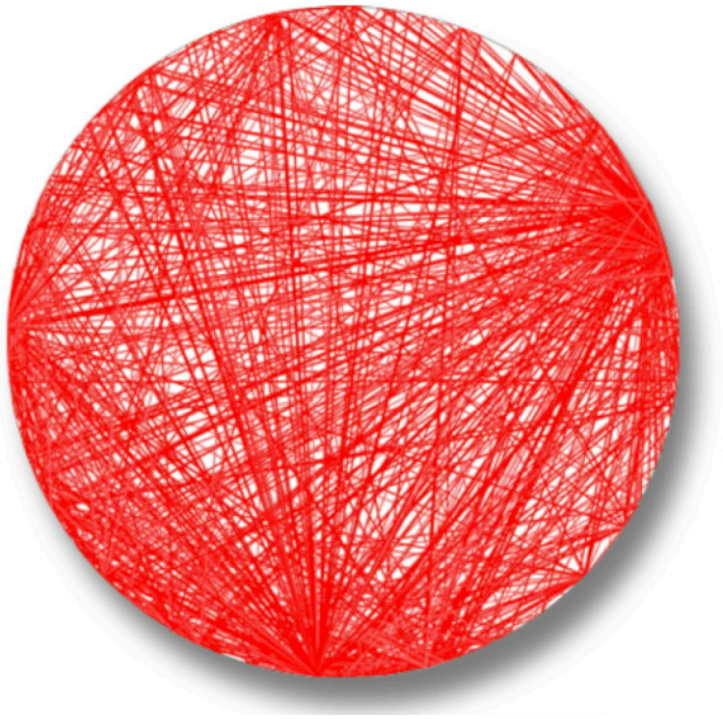


Figure 2: Big Ball of Mud is a tangled mess of function calls.

Anti-Patterns

God Objects

A God Object represents the opposite problem; we have no issues with object sharing because most functionality is contained in a *single monolithic object*!

```
class GameManager {
private:
  vector<string> players;
  int score = 0;
  bool isRunning = false;

public:
  GameManager() = default;
  ~GameManager() { /* ... */ }

  void addPlayer(const string& name) {
    players.push_back(name);
    custom_print("Added player: ");
  }

  void startGame() {
    isRunning = true;
    score = 0;
    custom_print("Game started!");
  }

  void updateGame() {
    if (isRunning) {
      score += 10;
      custom_print("Score updated: ");
    }
  }

  void endGame() {
    isRunning = false;
  }

  void draw() {
    custom_print("[Rendering]");
  }

  void handleInput(const string& input) {
    if (input == "quit") {
      endGame();
    }
  }
}
```

Application Layers

Let's introduce some basic structure.

To address separation of concerns, we divide an application into horizontal layers, where each layer represents a specific area of functionality [1].

The **UI Layer** handles the user-interface and all input and output for your application.

The **Domain Layer** handles "Business Logic" i.e. functionality related to your problem domain. This is the layer that contain the most of your application-specific logic.

The **Data Layer** handles saving data to a file, database or other service.

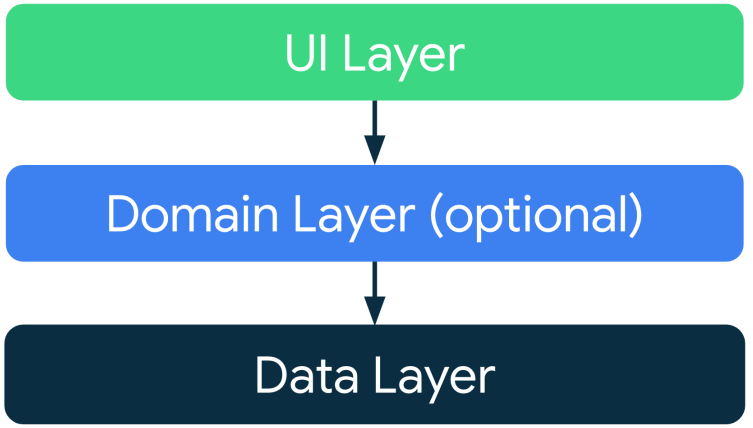


Figure 3: Courtesy of Android Guide to App Architecture

Application Layers

Layers can only directly communicate with the layers immediately below them.

This restricts our dependencies: the UI Layer for example can directly access Domain Layer classes, but has no access to the Data Layer classes. Likewise, the Data Layer cannot directly call into the UI Layer.

Benefits

- Separation of concerns
- Control flow is driven top-down.
  - Requests flow top-down.
  - Data flows bottom-up.
- No circular dependencies
  - One-way, or weak dependencies only between layers.

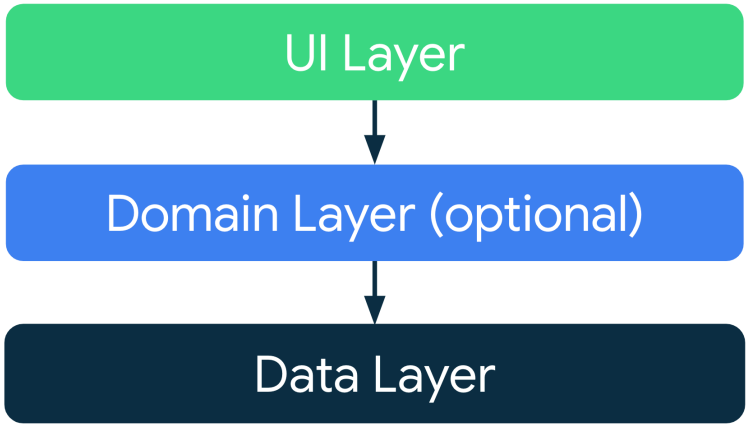


Figure 4: Courtesy of Android Guide to App Architecture

Note that Android documentation flags the “Domain Layer” as optional, but we’ll implement it in this course.

Application Layers

UI Layer

The role of the UI layer is to display the application data on screen. Whenever the data changes, either due to user interaction (such as pressing a button) or external input (such as a network response), the UI updates to reflect the changes.

The UI layer is comprised of two types of structures:

- **UI elements** that render data to the screen e.g., buttons, images, sliders.
- **State holders** that hold data, expose it to the UI, and handle logic to interact with that data.

The UI layer is the pipeline that converts application data changes to a form that the UI can present. [3].

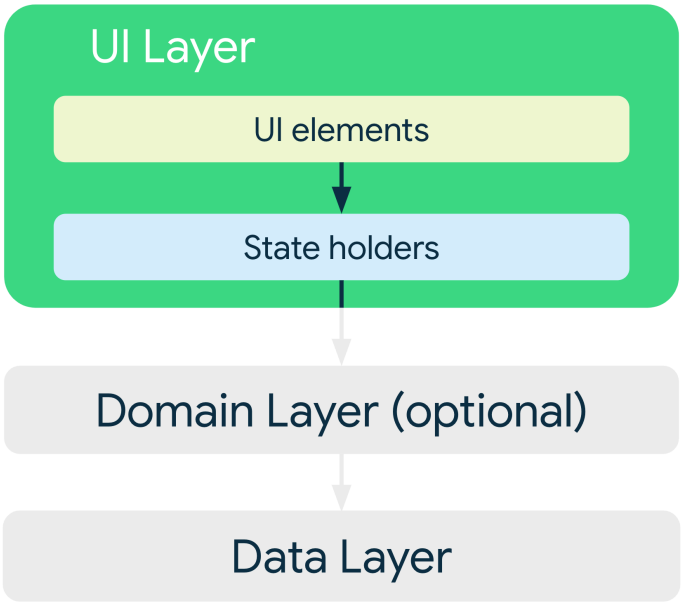


Figure 5: Courtesy of Android Guide to App Architecture

Application Layers

In order for this pipeline to work, we need a clear mechanism to push data from the Data Layer, through the Domain Layer and into the UI Layer (where the State holder manages the data for the UI elements).

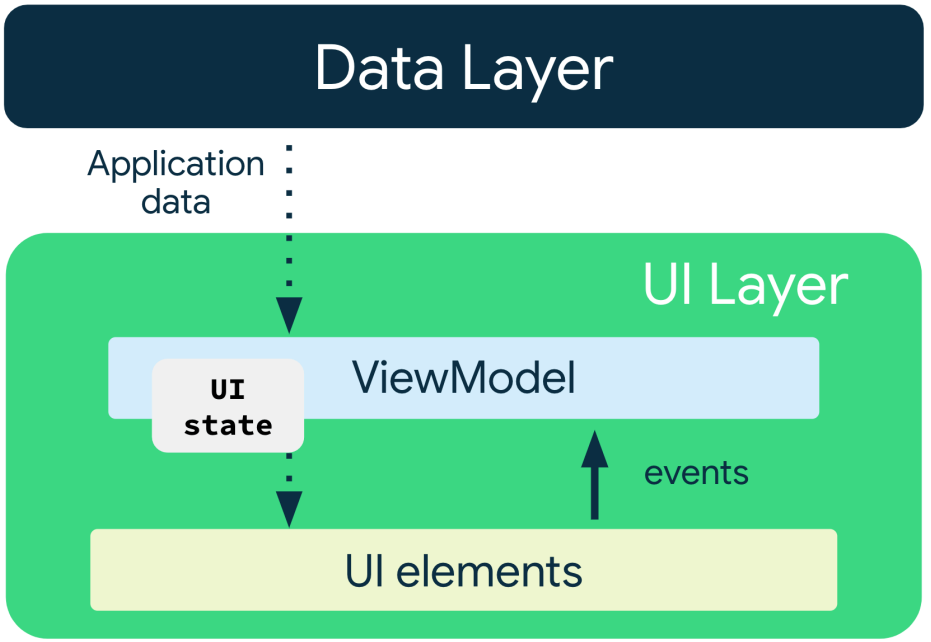


Figure 6: A ViewModel class in the UI Layer receives data from the Data Layer. The ViewModel can publish it to UI elements to be displayed. Courtesy of Android Guide to App Architecture

Application Layers

The pattern where the state flows down and the events flow up is called a unidirectional data flow (UDF). The implications of this pattern for app architecture are as follows:

- The `ViewModel` holds and exposes the state to be consumed by the UI. The UI state is application data transformed by the `ViewModel`.
- The UI notifies the `ViewModel` of user events.
- The `ViewModel` handles the user actions and updates the state.
- The updated state is fed back to the UI to render.
- The above is repeated for any event that causes a mutation of state.

> **Note**
>
> These principles apply to many modern frameworks e.g., React.
> We'll look at Compose examples of these classes soon.

Application Layers

Domain Layer

The domain layer sits between the UI and data layers. The domain layer is responsible for encapsulating complex business logic or simpler business logic that is reused by multiple view models.

This layer also provides a very clean separation of your business logic from your UI and data layers, which makes testing user stories much easier.

Use it to represent your business logic

- user stories and functionality that you need to implement,
- data classes (entities) that are required by the other layers.

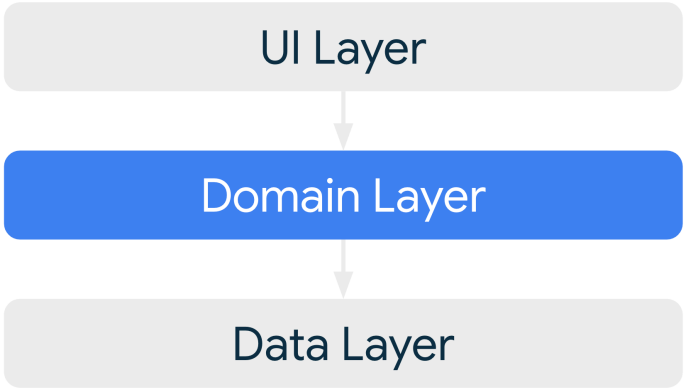


Figure 7: Courtesy of Android Guide to App Architecture

Application Layers

The main reason to add a Domain layer is to introduce another layer of abstraction, so that we can operate on multiple data repositories and merge results.

For example, a new reader application might want to combine Author information and News information into a single screen. This would require some entity (ideally Domain Layer) to coordinate requests across two repositories and return the combined results to the User Interface Layer.

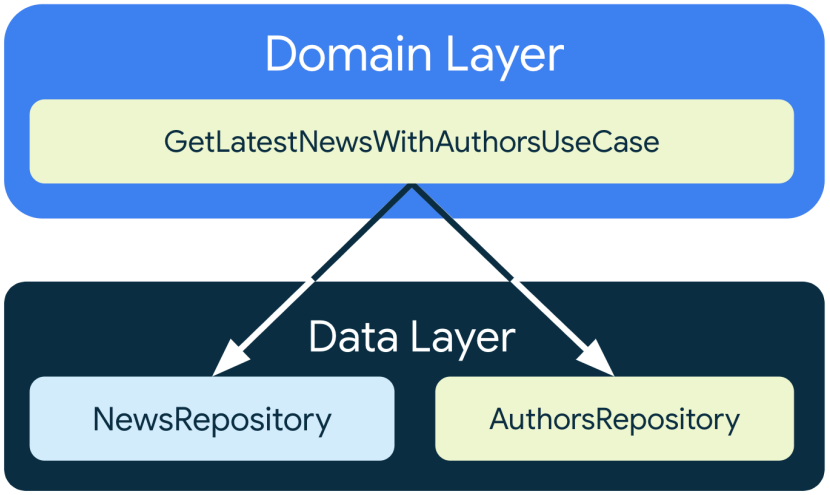


Figure 8: A domain-level user story can manage results from multiple repositories. Courtesy of [Android Guide to App Architecture]

Application Layers

Data Layer

The data layer of an app contains the core data for your application. It also comprises rules that determine how your app creates, stores, and changes underlying data.

The data layer is modelled as repositories of data sources.

- Sources represent single data locations e.g., web API.
- Repositories aggregate related data from multiple sources.

You should have a repository class for each specific type of data.

- e.g., a `MoviesRepository` repo could have a DB source for sales data, and a web source for production info.

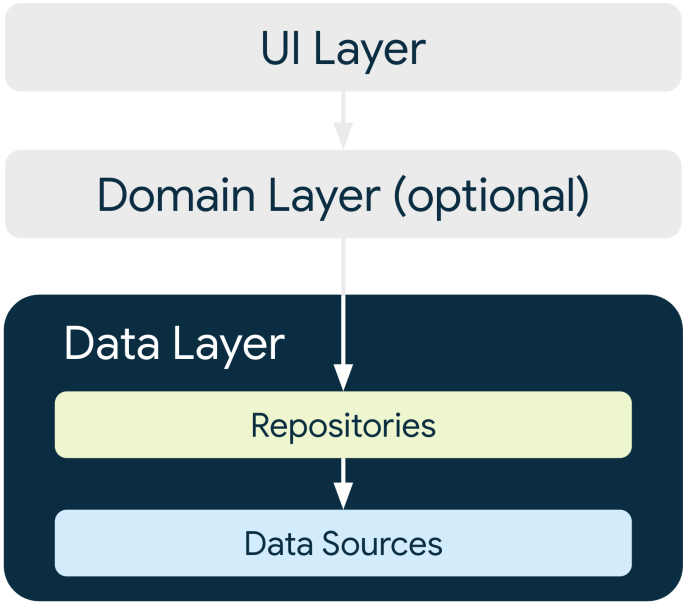


Figure 9: Courtesy of Android Guide to App Architecture

Application Layers

Repositories

Repository classes are responsible for the following:

- Exposing data to the rest of the app
- Centralizing changes to the data
- Resolving conflicts between multiple data sources
- Abstracting sources of data from the rest of the app
- Containing (local) business logic

Repositories should named named as: `(type)Repository`.

- e.g., `NewsRepository`, `MoviesRepository`, or `PaymentsRepository`.

Data published by this layer should be `immutable` to prevent accidental mutation.

- Requests to change (update, delete) data should be made through the repository API i.e. methods which are designed to verify the request and create/update data.

Sources

Each data source class has the responsibility of working with only one source of data, which can be a file, a network source, or a local database.

Data source classes are named after the data they're responsible for and the source they use. The convention is : `(type-data)(type-source)DataSource`.

- e.g., `NewsRemoteDataSource` or `NewsLocalDataSource`.

> **Note**
>
> We'll provide more specific details later e.g., Database lecture.

Modularity

Modularity refers to the logical grouping of source code into related groups. It's the high-level component structure that we want when we talk about separation of concerns e.g., namespaces in C++, packages in Java or Kotlin.

Modularity provides:
- Clear division of responsibilities, easier to manage code.
- Opportunities for code reuse e.g., exporting a distinct module.
- Improved ability to test our code!

How do we split our code into modules?

Option 1: Sub-Modules

Kotlin Toolkit (and Gradle) both support the use of modules in your project. You effectively create different compilation units (one per module) and link them together as part of your build process.

For example, you could create separate top-level modules for `userinterfaces`, `data` and `domain` layers of your application and let the build system manage it.

There are benefits:

- It enforces very strict dependencies , since your have to define them for the build configuration.
- Your build system can build each module separately which is faster.
- You can more easily export a module to share it.

This is pretty heavyweight though for a single application. Most of the time you don't want to do this.

Option 2: Packages

A package is a mechanism for grouping related classes, interfaces and other related code together. It's similar to a namespace in C++.

Use a package name that is unique, and describes that module. The naming convention is: `(unique-URL-reversed).suffix`.

- e.g., `ca.uwaterloo.cs346.taskit`, or `org.jeffavery.thesis`.

Define a package structure for each "level" of your project. Use the `package` keywork at the top of a file to associate it to a package.

For example, here's an application that works with customer and product data. We have top-level packages for `data`, `domain`, and `userinterfaces`.

Notice how the directory structure matches the package names. This is a holdover from Java, which requires these precision (Kotlin is more forgiving, but we still do it out of convention).

```
src
├── data
│   └── CustomerRepository.kt
│   └── ProductRepository.kt
└── domain
│   └── Customer.kt
│   └── Product.kt
└── userinterface
    └── CustomerView.kt
```

Package names:

```
ca.uwaterloo.cs346.data
ca.uwaterloo.cs346.domain
ca.uwaterloo.cs346.userinterface
```

Class names (fully qualified):

```
ca.uwaterloo.cs346.domain.Customer
ca.uwaterloo.cs346.domain.Product
...
```

Modularity

To “use” a class or function from a different package, you import it.

```
package ca.uwaterloo.cs346.userinterface

import ca.uwaterloo.cs346.domain.customer // make this class available
import ca.uwaterloo.cs346.domain.product // make this class available

class CustomerView(val custID: Int) {
  val user = Customer(custID)
  // ...
}
```

What are the benefits of doing this?

- It’s an easy way to keep layers separate e.g., `userinterfaces` layer should not import `data` layer directly.
- It’s resilient: you can move classes around easily.

However there are drawbacks:

- It exposes the entire source tree to your developers.
- Your build system will treat this as a single module, which can be slow.

Bibliography

[1] M. Richards and N. Ford, *Fundamentals of Software Architecture A Modern Engineering Approach*, 2nd ed. O'Reilly Media Inc., 2009.

[2] M. Fowler, “Software Architecture Guide.” [Online]. Available: https://martinfowler.com/architecture

[3] Google Inc., “Android Developer Documentation.” [Online]. Available: https://source.android.com/docs

[4] B. Foote and J. Yoder, “Big Ball of Mud,” in *Pattern Languages of Program Design*, Addison-Wesley Professional, 1997, pp. 654–692.