Software Architecture

CS 346 Application Development

What is architecture?

Expert developers working on a project have a shared understanding of the system design. This shared understanding is called ‘architecture’ and includes how the system is divided into components and how the components interact through interfaces.

— Martin Fowler, Who Needs an Architect? (2003).

Architecture is *the holistic understanding of how your software is structured, and the effect that structure has on its characteristics and qualities*.

Architecture as a discipline suggests that structuring software should be a *deliberate action*.

Why is it important?

Decisions like “how to divide a system into components” have a *huge* impact on the characteristics of the software that you produce.

- Architectural decisions like this are necessary for your software to work properly.
- The structure of your software will determine how well it runs, how quickly it performs essential operations, how it handles errors.
- Well-designed software can be a joy to work with; poorly-designed software is frustrating to work with, difficult to evolve and change.

Architecture supports requirements

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”, “display a sales report”.

Defined during our requirements process. Explicitly designed.

Non-functional requirements refer to the *qualities* of our software. e.g., scalability, robustness, power-usage, speed, efficiency.

- User will often ask for non-functional requirements in general terms. e.g., “I want this operation to be fast!”; “I don’t want the app to use very much memory.”
- To address these requirements, we may need to quantify them and measure them to know if they have been achieved.

Emerge from our architecture, based on design decisions that we make.

Qualities

What are the properties and qualities of “correct” software?

Software qualities that **developers** care about

Goals as software developers:

- **Usability**: Our software is “fit for purpose” and meets functional requirements. It should address our user stories and problem statement.
- **Extensibility**: Over time, we should be able to extend existing functionality or add new functionality. e.g., adding a new file format.
- **Scalability**: Our software should be scalable to increased demand e.g., more users, more transactions, at a faster rate.
- **Robustness**: Our software should be stable and handle uncertain inputs.
- **Reusability**: We should reuse design/code whenever possible and design our solution in a way that fosters reuse. (*However,* beware YAGNI*).*

Software qualities that **customers** care about

Goals as users of a system:

- **Performance**: Does the system return results to users in a reasonable time?
- **Reliability**: Do the system features behave as expected?
- **Availability**: Can the system deliver its services when requested by users?
- **Security**: Does the system protect itself & data from unauthorized access?
- **Usability**: Can system users easily and quickly access features?
- **Maintainability**: Can the system be readily updated and new features added without undue costs?
- **Resilience**: Can the system deliver services in the event of a failure?

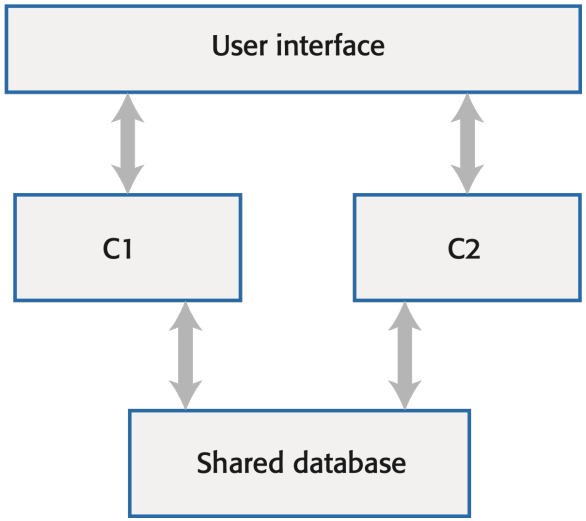
Trade-offs between qualities

Focusing on one quality will often affect other qualities.

- Improving reliability may reduce performance or responsiveness (+reliability, -performance).
- Improving security may require adding security features, which reduce usability for users (+security, -usability).
- Design entails compromise. Which outcome is preferable?

Example

- A system may share a database between components.
- Assume C1 runs slowly because it must reorganize the information in the database before using it.
- Changes that help C1 may negatively affect C2.



Sharing a database between two components may slow down performance.

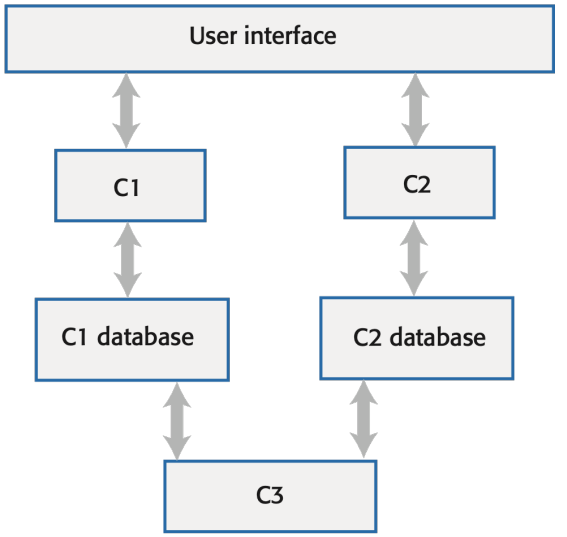
Example: Maintainability & Performance

This diagrams shows a different architecture where each component has its own copy of the parts of the database that it needs.

- If one component needs to change the database organization, this does not affect the other component (+maintainability)

However, a multi-database architecture may run more slowly and may cost more to implement and change.

- A multi-database architecture needs a mechanism (component C3) to ensure that the data shared by C1 and C2 is kept consistent when it is changed.
- This reduces performance (-performance).



This design improves maintainability but will probably perform worse than the shared database.

Architectural Principles

How do we achieve these goals?

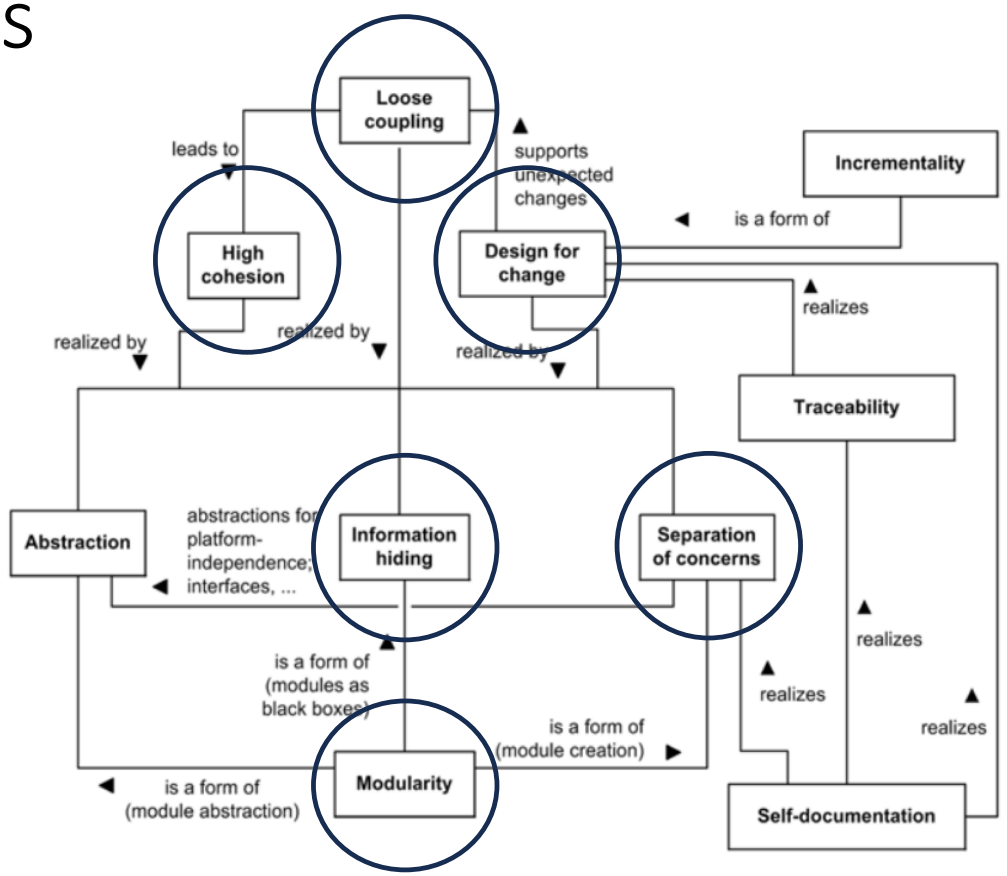
Architecture principles

There are well-known approaches that we can take to produce code that is

- Flexible
- Extensible
- Robust
- Reusable

There is no “magic bullet” solution; you will need apply these principles to your specific project.

Our goal is for you to recognize and learn to apply these principles.



- Oliver Vogel et al. 2011. **Software Architecture**. Springer.

Single Responsibility (SRP)

Your architecture should consist of **components**.

- Each component is a relatively independent software entity that implements a coherent set of functionality e.g., class, library.
- This responsibility should be clearly defined and should not extend outside of this component (as much as possible).
- Components can be classes, or functions, or module. It's "loosely defined" on purpose.

Benefits

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

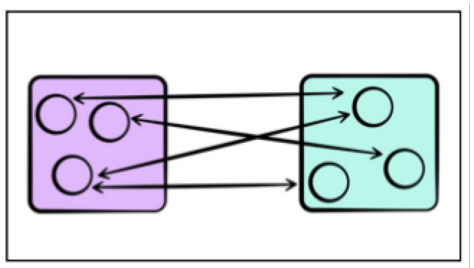
Separation of concerns

Separation of concerns states that your components should be independent of one another. We express this in terms of of cohesion and coupling:

- **High cohesion** within a component i.e. each component is self-contained with clear responsibilities.
- **Loose coupling** between components i.e. they should be self-reliant and there should be few dependencies between them.

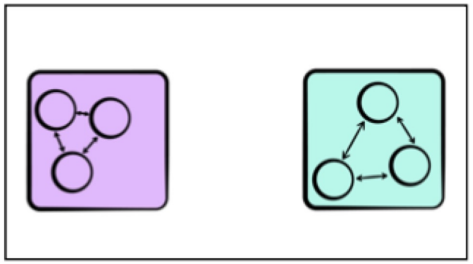
Benefits

- Clear separation == stable APIs (between components).
- Changes tend to me isolated, and don't "spread" as far.



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.

Information Hiding

Information hiding is the idea that a component's state should not be exposed.

- State should be internal and private.
- Components should provide a well-described interface that exposes public functionality. Other components MUST use this interface to communicate.

Information hiding is a necessary part of separation of concerns.

Modularity

Modularity refers to the logical grouping of source code into related groups i.e. it's the structure that we want when we talk about separation of concerns.

- e.g., namespaces in C++.

Why is it important?

- Clarity and simplicity of our code.
- Supports a clear division of responsibilities.
- Opportunities for code reuse.

Distribution Patterns

High-level component distribution.

Distribution Patterns

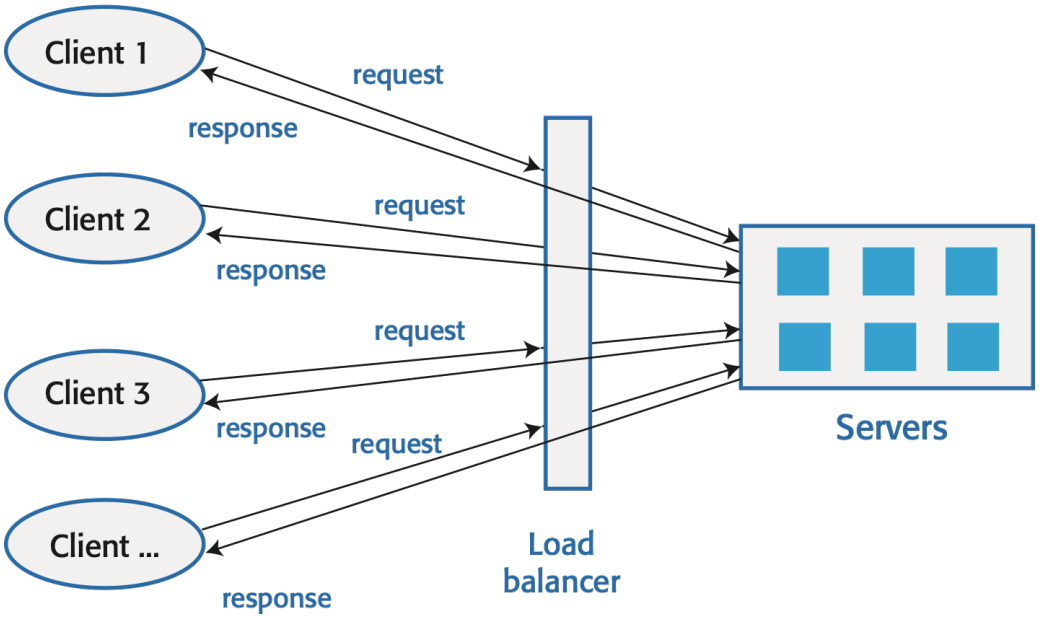
The **distribution architecture** of a software system defines the distribution of your application across physical hardware. In a distributed architecture, application functionality is distributed across more than one system

Definitions

- A **client** is the system where your application is running. It's the system that the user interacts with.
- A **server** is a system provides capabilities to one or more clients. It's often common logic or shared data that we need to have available to all clients.

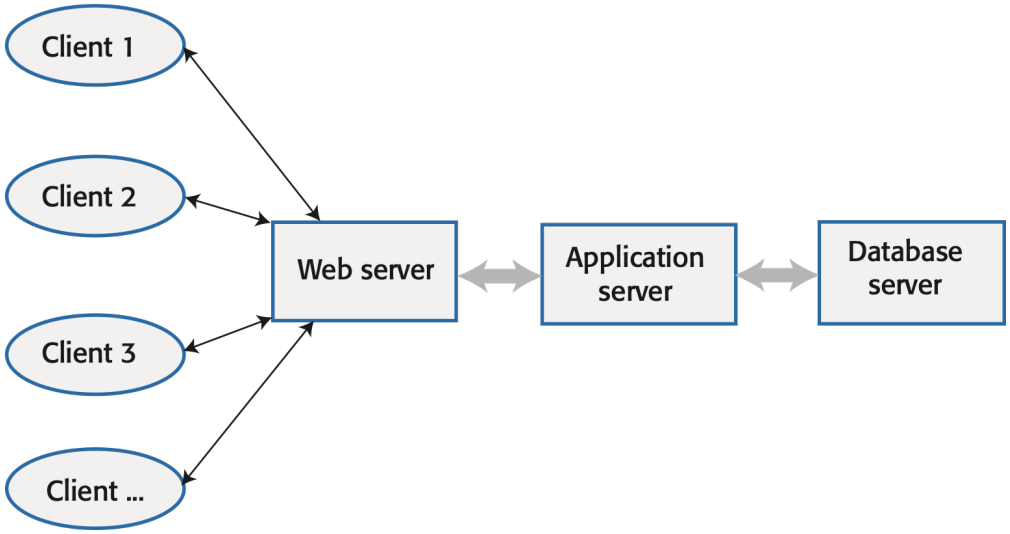
Client-server architecture

We have a web services lecture that discusses this.



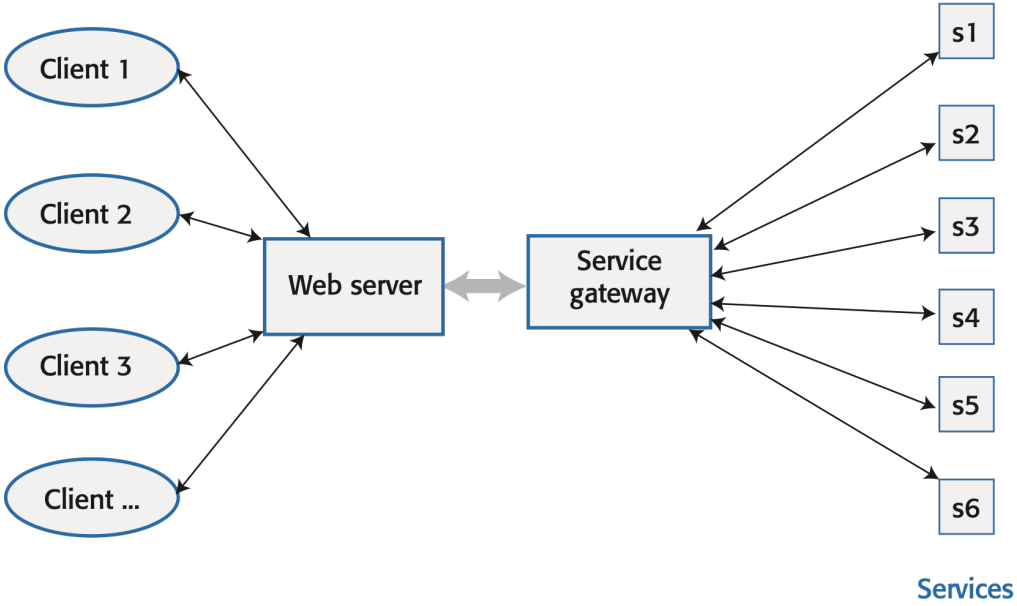
A client-server architecture splits processing between the application (client) and a remote server which is typically shared by all clients e.g., a simple web server that provides services to browsers running on many different systems, or a many clients using a shared database.

Multi-tier client-server architecture



A client-server architecture splits processing between the application (client) and one or more remote systems, which coordinate work. e.g., a web server that provides services to browsers, which in turn relies on an application server and database server.

Service-oriented architecture

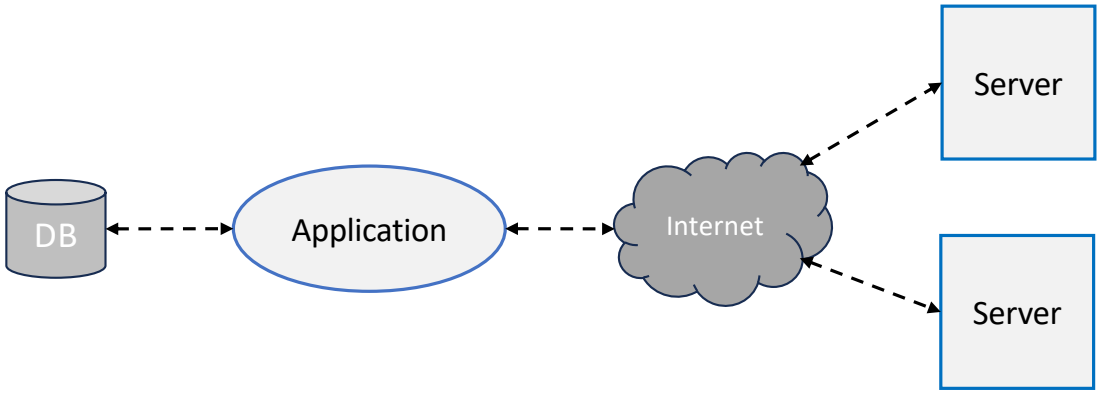


Services in a service-oriented architecture are stateless components, which means that they can be replicated and can migrate from one computer to another. A service-oriented architecture is usually easier to scale as demand increases and is resilient to failure.

Application Patterns

How to structure *your* application.

Standalone architecture



A simple standalone application tends to be self-contained. It may form ad-hoc connections but has few external dependencies, and most computation is handled within the application executable.

What is an application pattern?

Every one of the patterns that we just discussed includes a **client**: the main application that the user interacts with.

Reminder: client applications need to

- Handle user interaction with a graphical interface.
- Communicate with servers and databases.
- Manage the application state between these entities.

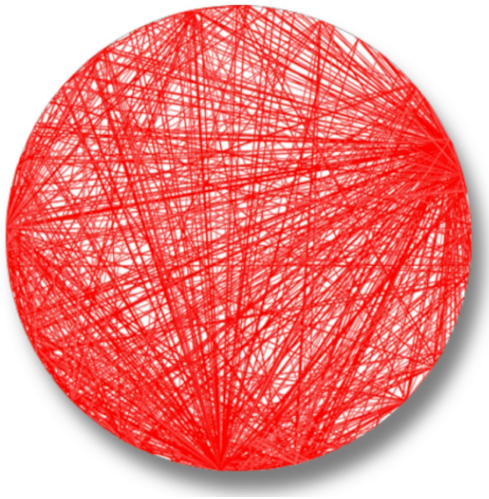
How do we structure our application to do this?

Antipattern: “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**.

-- Foote & Yoder 1997.



A Big Ball of Mud is the result of a system being *tightly coupled*, where any module can reference any other module. A system like this is difficult to extend or modify.

Example: Pipe & filter architecture

A pipeline architecture transforms data as it passes through a series of components:

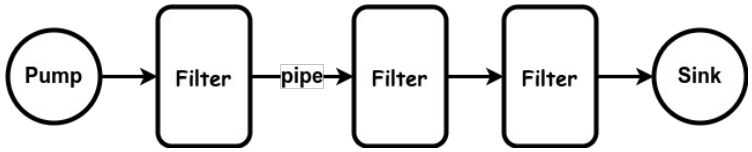
- **Pump** is the data producer. This can be any input e.g., keyboard, data file.
- **Pipes** are unidirectional, accepting input passing to the next component.
- **Filters** perform operations on data before passing it along.
- **Sink** is the data target e.g., output file, database.

Examples:

- Unix programs, compilers.

Suitability for us?

- Great for consoles, but not interactive applications.



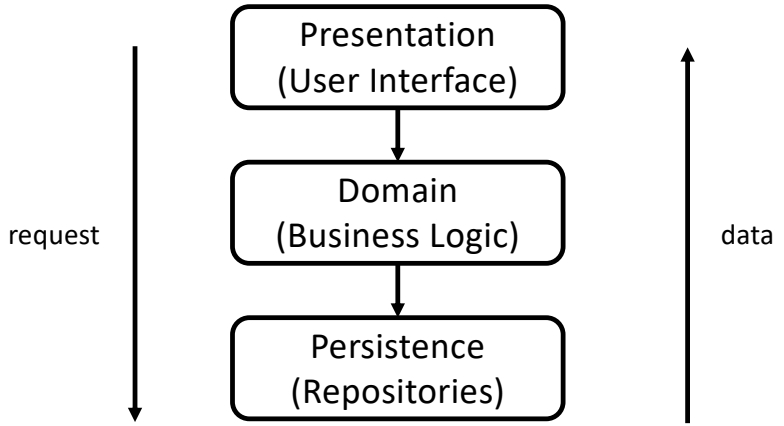
https://architectural-patterns.net/pipe-and-filter

Example: Layered architecture

A layered architecture groups components into horizontal layers, where each layer represents a ***logical*** division of functionality. Each layer can communicate with the layer immediately below it; requests flow down from the top.

This is a **common application pattern.**

- Presentation handles the user interface, including all input and output.
- The Domain layer handles so-called "Business Logic" i.e. components or classes to handle the functionality related to your problem domain.
- Persistence layer handles saving data to a file, database or other service.



https://architectural-patterns.net/layers

Layered architecture

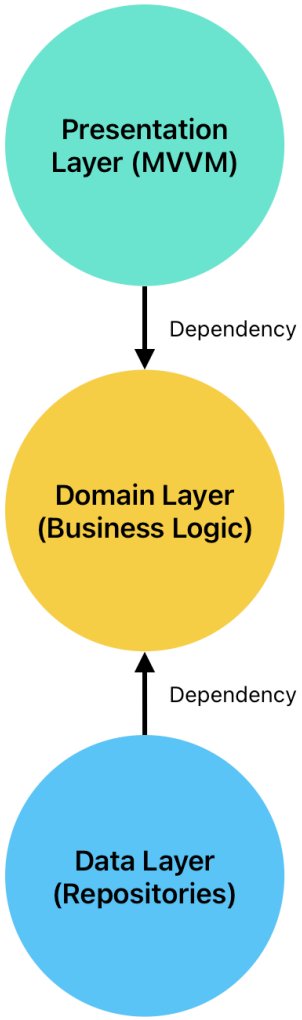
We'll use a modified layered architecture*

- Three layers or modules.
- Dependencies flow towards the center (domain layer).
  - User interaction can supply input data, but so can external sources like a DB or web service.
  - The domain layer coordinates interaction; inner logic is expressed in terms of the user's problem domain.
- Outer layers can ONLY communicate through the domain layer.

Advantages

- Layers reinforce a clear separation of concerns.
- Dependencies reflect the actual flow of data/control.
- This approach should handle all our requirements.

* This is similar to Martin's idea of a Clean Architecture.



What's all this about dependencies?

A dependency reflects a relationship between two modules.

- If A requires B to compile, then A has a dependency on B i.e., A → B
- If A and B require each other i.e. A → B and B → A, then we have a **circular dependency**. We want to avoid these since they make code more complex and difficult to debug and test.
  - e.g., how do you create A without having a reference to B and vice-versa? How do you test them in isolation?

Our layered architecture avoids any circular dependencies since you should only have:

Presentation → Domain ← Data

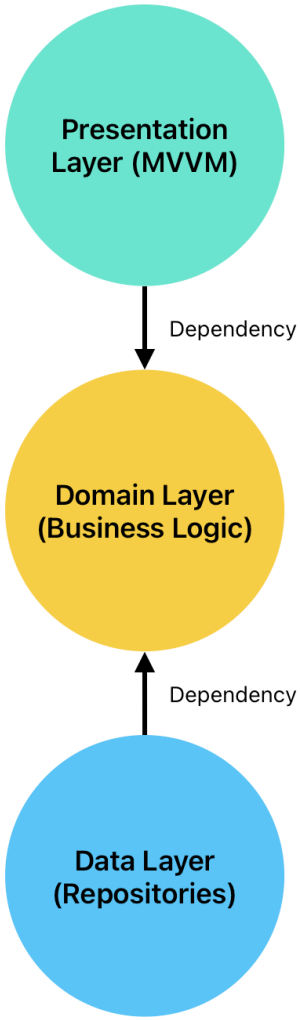
Source code structure

This means that your application code should be split into these modules. You should have packages containing the classes and functions for each layer:

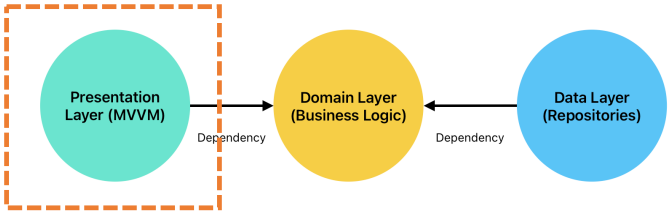
/presentation -- all user interface classes

/domain/ -- all data and custom classes that you produce

/data/ -- all external services e.g., your database



1. “presentation” package



In many ways, the user interface is the most complex part of the application.

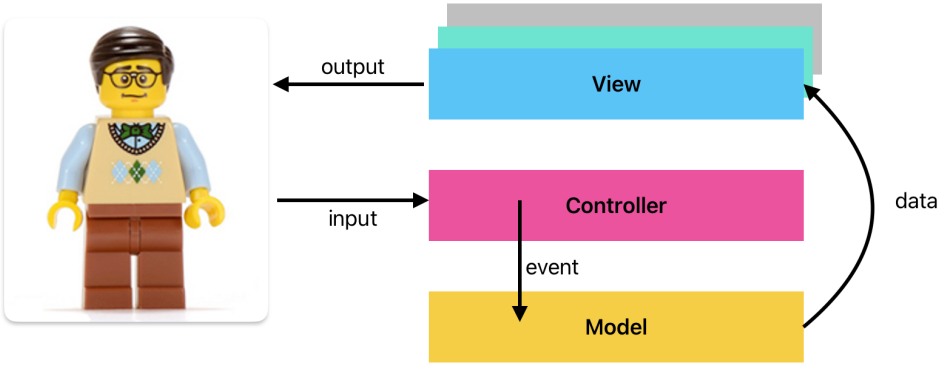
You need to handle the interaction cycle:

- Accept input from the user e.g., mouse or keyboard.
- Process it through the domain layer e.g., fetch data from web, or save data to DB.
- Output results e.g., on a screen.

In a GUI application, the system should prioritize accepting user input, and responding to it. This **interaction cycle** is critical.

Model-View Controller

MVC is a pattern **within** the presentation layer.



MVC originated with Smalltalk (1988), as a UI pattern for interactive applications. It is a particular implementation of the `Observer` pattern.

- Input is accepted and interpreted by the **Controller**,
- Data is routed to the **Model**, where it changes program state.
- Changes are published to the **View**(s) and are reflected to the user as output.

MVC Classes

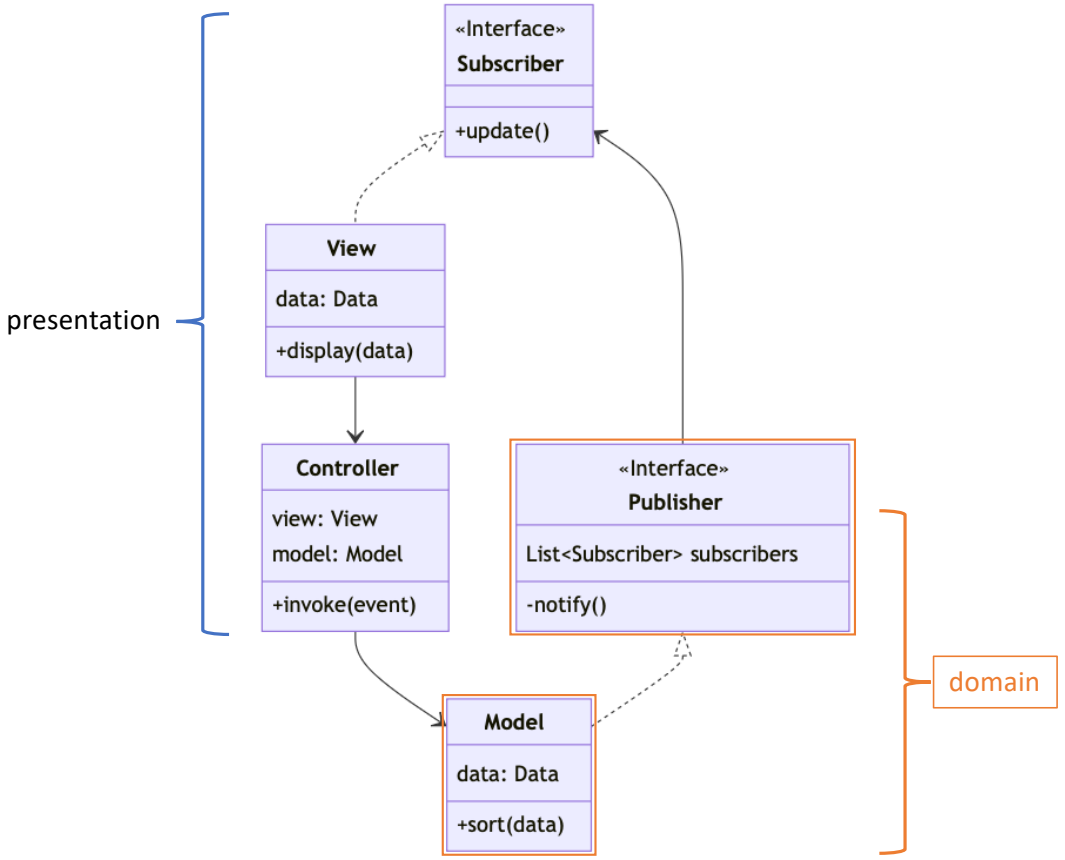
Components

- **View**: output
- **Controller**: handles input.
- **Model**: manages state

MVC uses the Observer pattern to notify Subscribers.

Flow

- The user provides input.
- The controller passes to the model which acts on it.
- The model notifies subscribers aka views and they update themselves.



Problems with MVC?

However, there are a few challenges with standard MVC.

- Graphical user interfaces bundle the input and output together into graphical “widgets” on-screen (*see user interfaces lecture*).
  - This makes input and output behaviours difficult to separate
  - In-practice, the controller class is rarely implemented.
- Modern applications tend to have multiple screens.
  - Need something like a coordinator class to control visibility of screens.
  - Each screen may have its own data needs which cannot be handled by a single model.
- It’s a Presentation layer architecture!
  - We need to adapt it to work with our domain and data layers.

Model View View-Model (MVVM)

Model-View-ViewModel was invented by Ken Cooper and Ted Peters in 2005. It was intended to simplify event-driven programming and user interfaces in C#/.NET.

MVVM adds a **ViewModel** that sits between the View and Model.

Why? Localized data.

- We often want to pull "raw" data from the Model and modify it before displaying it in a View e.g., currency stored in USD but displayed in a different format.
- We sometimes want to make local changes to data, but not push them automatically to the Model e.g., undo-redo where you don't persist the changes until the user clicks a Save button.

Model View View-Model (MVVM)

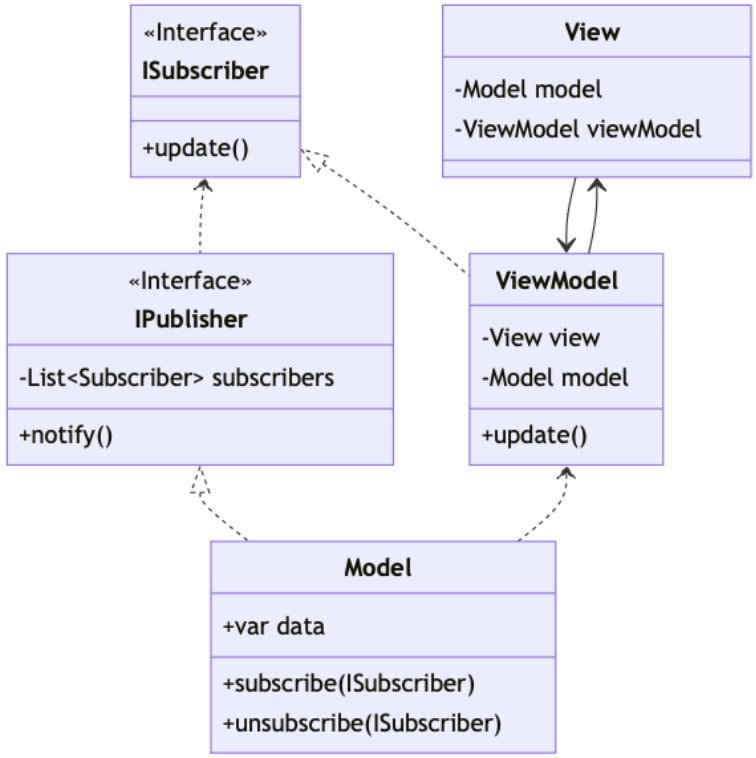
Components

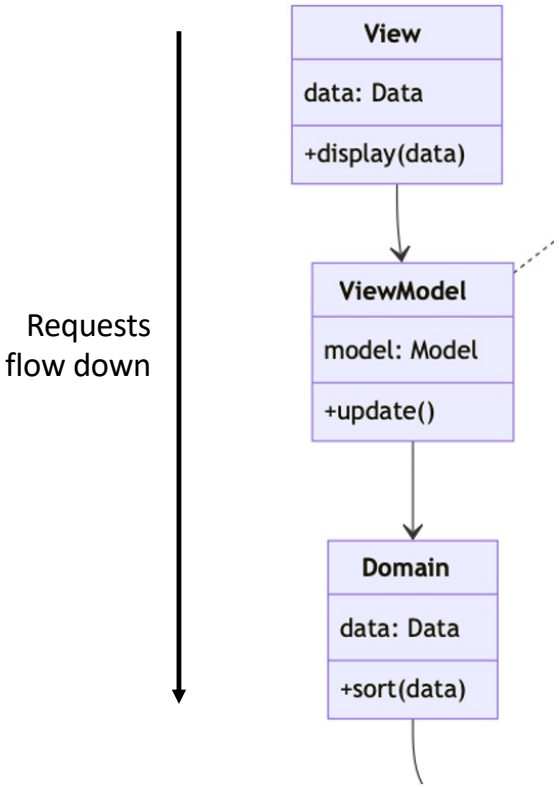
- **View**: input and output.
- **ViewModel**: localized data for the view.
- **Model**: stores the main data.

Each View typically has one ViewModel associated with it.

MVVM also uses the Observer pattern to notify Subscribers, but unlike MVC, the subscriber is typically a ViewModel.

The View and ViewModel are often tightly coupled so that updating the ViewModel data will refresh the View.

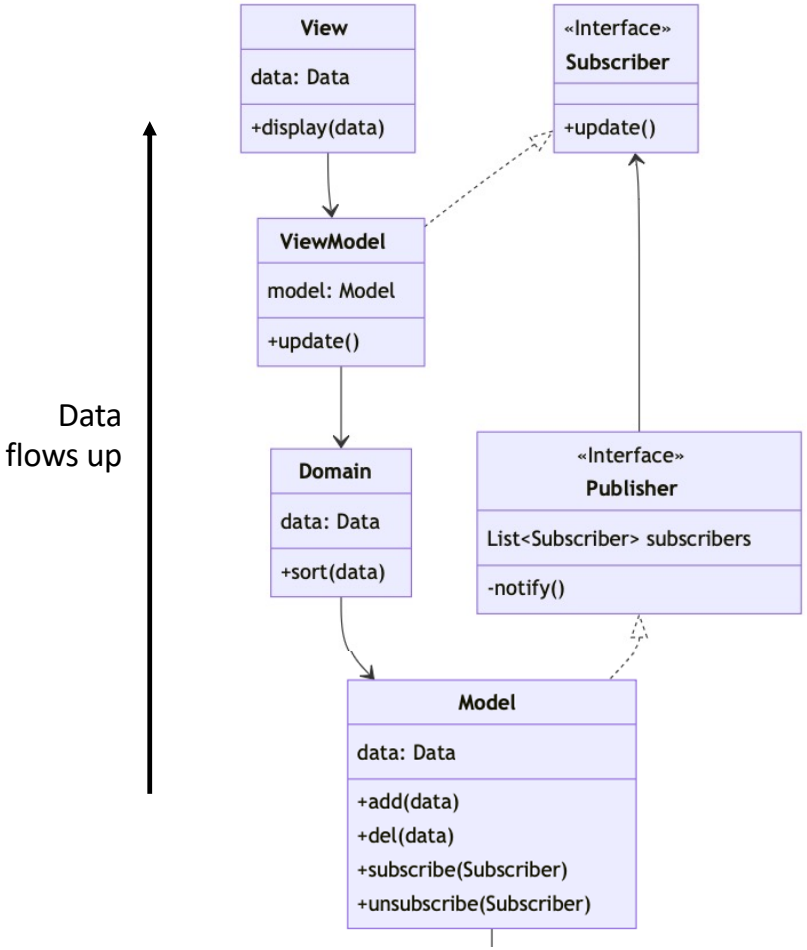




Dependency rule

Dependencies flowing “down” means that each layer can only communicate directly with the layer below it.
In this example, the UI layer can manipulate domain objects, which in turn can update their own state from the Model.

e.g., a Customer Screen might rely on a Customer object, which would be populated from the Model data (which in turn could be fetched from a remote database).

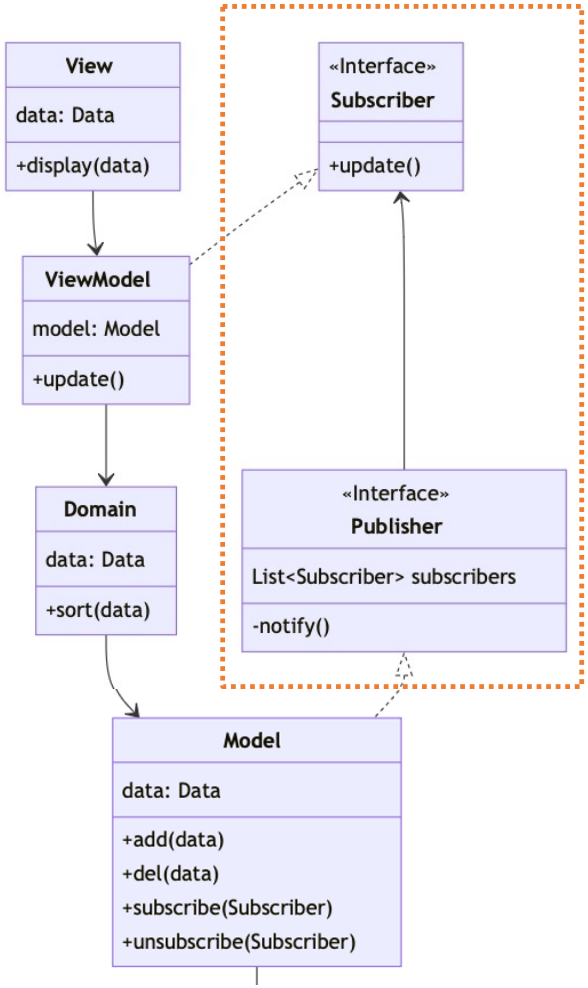


Update rule

Notifications flowing up means that data changes must originate from the "lowest" layers.

e.g., a Customer record might be updated in the database, which triggers a change in the Model layer. The Model in turn notifies any Subscribers (via the Publisher interface), which results in the UI updating itself.

In other words, updates flow back "up" the hierarchy.



Observer Pattern

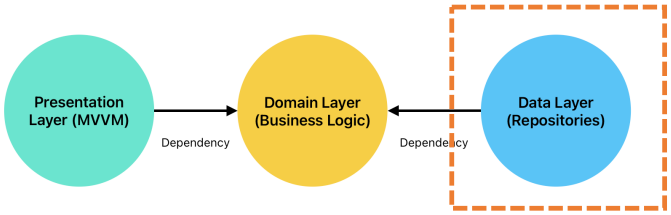
MVVM leverages the **Observer pattern**.

That design pattern describes how a source-of-data (model) sends updated data to a subscriber (view-model) when the data changes.

This pattern will emerge over-and-over in this course, in the user-interface lecture and the concurrency lecture.

We will revisit it several times in more detail.

2. “data” package



Your data package will consist of classes and functions for accessing external data. This can be from any external source.

- e.g., database, web service, file, network socket.

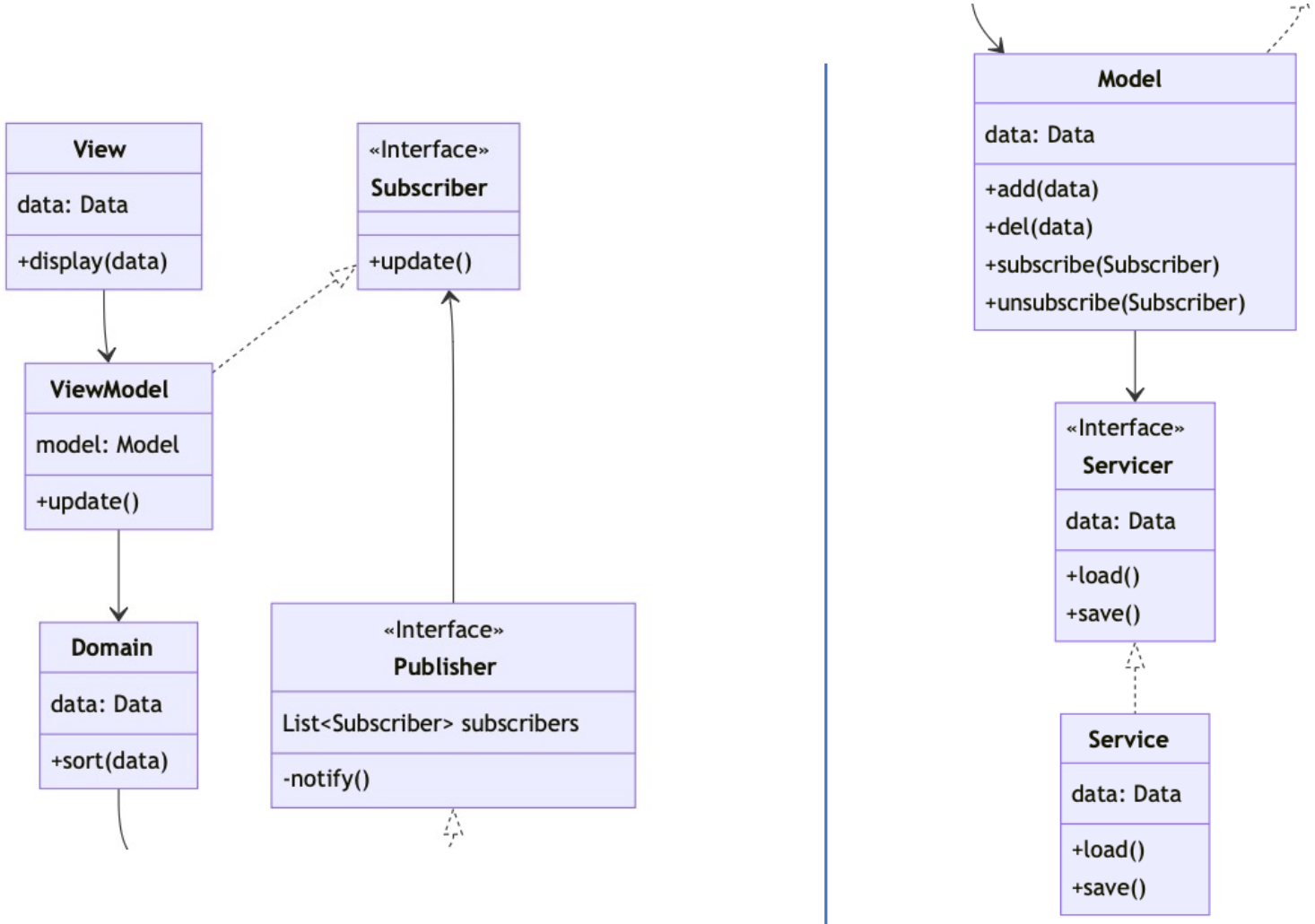
We use the term repository for a “source of data” and abstract it behind an interface.

- **IRepository**: interface of common functions used across repositories.
- **Repository**: common term for any data source.

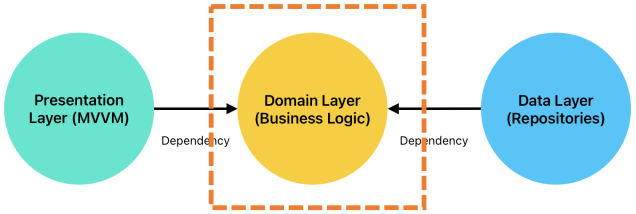
In practice, you should use specific names e.g., `CustomerRepository`, `SalesRepository`.

Classes in this package are allowed to access the `domain` package.

- Data will often be returned from a data source and “converted” into a domain specific data class. e.g., Customer data as a `List<Customer>`.


View
data: Data
+display(data)
«Interface»
Subscriber
+update()
ViewModel
model: Model
+update()
Domain
data: Data
+sort(data)
«Interface»
Publisher
List<Subscriber> subscribers
-notify()
Model
data: Data
+add(data)
+del(data)
+subscribe(Subscriber)
+unsubscribe(Subscriber)
«Interface»
Servicer
data: Data
+load()
+save()
Service
data: Data
+load()
+save()

3. “domain” package



Classes and functions specific to your application and the problem you are solving. Data classes are usually stored here.

- e.g., Customer data class (used by the `CustomerRepository` when loading data, and by the `CustomerView` layer to display it).
- e.g., Sales class which may reference a `List<Customer>`.

You might also create top-level functions that align with your use cases.

- e.g., if you have a use case to display Customer Sales data, your `domain layer` would pull in data from both the `CustomerRepository` and `SalesRepository`, format it for the report and send it to the `presentation layer` to display it.

This tends to be the most specialized layer since it contains classes specific to the problem you are addressing. The external layers *tend* to be more generic.

- Not 100% true, since (for example) the Repository will contain your table structure, and the presentation layer displays domain data. However, domain *processing* stays here.

Handling modularity

Packages? Multi-module structures? What to do?

Implementing modularity

Recall: modularity is the logical grouping of components to enforce separation-of-concerns (loose coupling).

Kotlin supports modularity in two ways.

- **Modules**: A top-level collection of related components.
- **Packages**: A collection of logically related functionality.

Do you store your layers in separate modules, or just separate packages?

Option 1: Using packages for modules

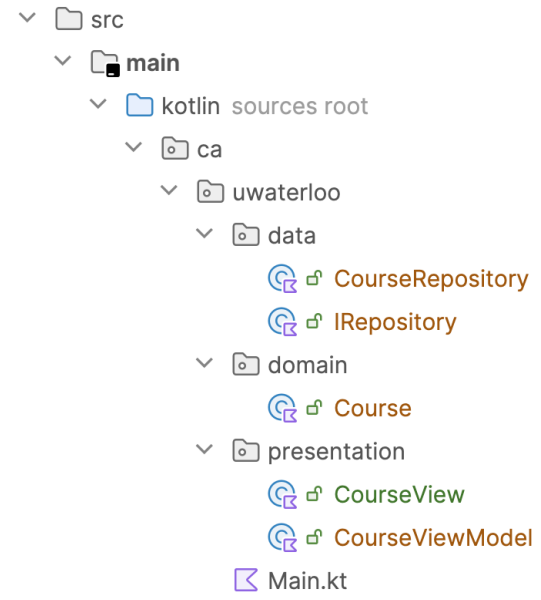
Create directories for each layer, where all source code is saved.

Benefits

- Easiest to implement! Just create the packages.
- Resilient. You can move classes around easily.

Drawbacks

- Easy to accidentally allow the wrong dependencies e.g., you can import `data` into `presentation`.
- Slow to build since Gradle will treat all of this as a single module and rebuild it all.
- Everyone has access to the entire source tree, all of the time.



One module ‘main’ and packages for data, domain and presentation layers.

Option 2: Multi-module structure

Create a separate module for each layer:

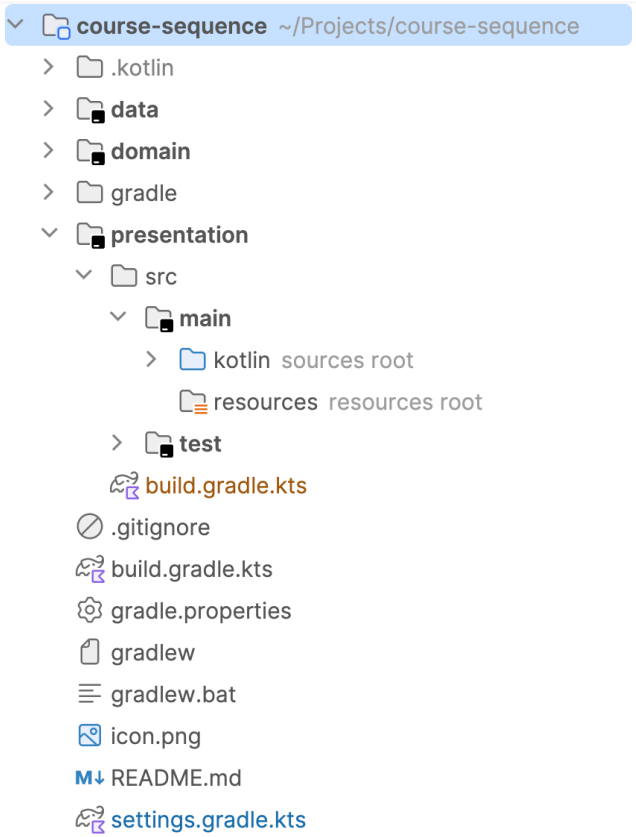
`/data`, `/domain`, `/presentation`.

Benefits

- Enforces dependencies since you define them in each module's `build.gradle.kts` file.
- Clean separation, so you can restrict access by module.
- Gradle can incrementally build modules, so faster builds.
- Easier to reuse code at the module level.

Drawbacks

- Harder to setup!
- Harder to maintain!
- Easy to break Gradle.



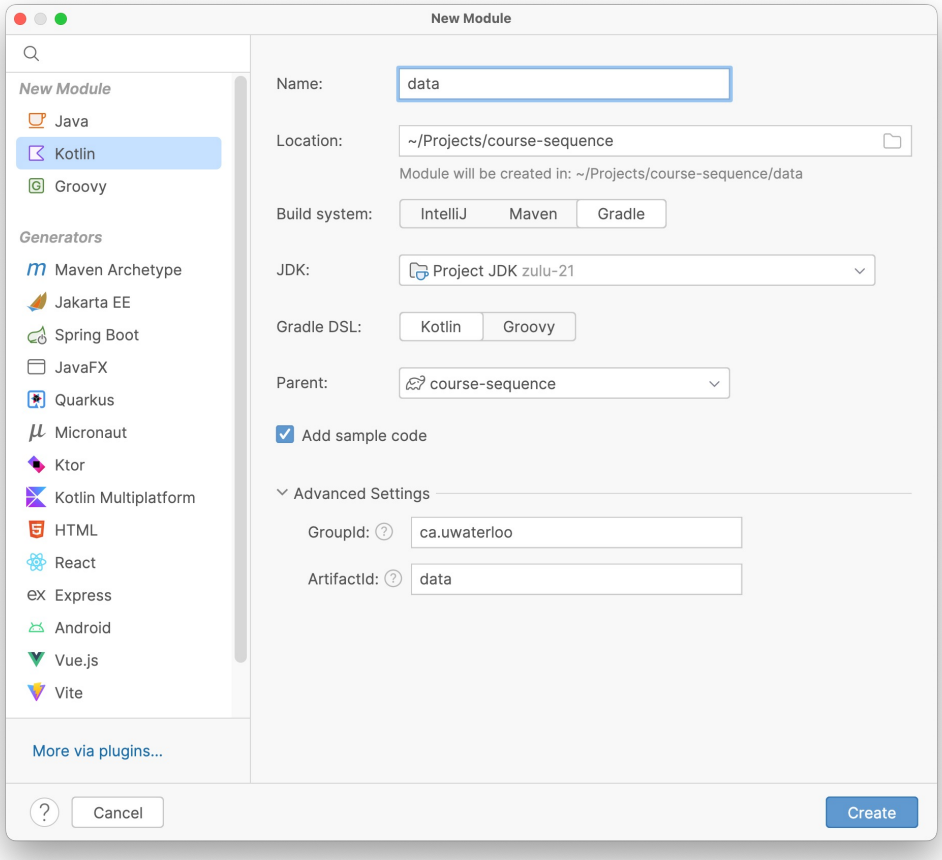
Adding modules

Modules:

- `/data`
- `/domain`
- `/presentation`

Use

- `File -> New -> Module -> Kotlin`
- Add module dependencies in `build.gradle.kts`



Benefits

Layering our architecture really helps to address our earlier goals (reducing coupling). It also provides these benefits:

- **Independence from frameworks**. The architecture does not depend on a particular set of libraries for its functionality.
- **Layers are more testable**. Layers can be tested independently of one another. e.g., the business rules can be tested without the UI, database, web server.
- **Independence from the UI**. The UI can be changed without changing the rest of the system. A web UI could be replaced with a console UI, for example, without changing the business rules.
- **Independence from the data sources**. You can swap out Oracle or SQL Server for MongoDB, or something else. Your domain logic is not bound to the database or to a specific data source.

References

- Fowler. 2002. Patterns of Enterprise Application Architecture. Addison-Wesley. ISBN 978-0321127426.
- Fowler. 2019. Software Architecture Guide.
- Martin. 2017. Clean Architecture: A Craftsman's Guide to Software Structure and Design. Pearson. ISBN 978-0134494166.
- Richards & Ford. 2020. Fundamentals of Software Architecture: An Engineering Approach. O'Reilly. ISBN 978-1492043454.
- Sommerville. 2021. Engineering Software Products: An Introduction to Modern Software Engineering. Pearson. ISBN 978-1292376356.