Architecture & Design

CS 346 Application Development

Building software “correctly”

“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, Clean Architecture (2016).

Desirable traits

As developers, we would like these traits in software that we build:

- **Usability**: Our software should be "fit for purpose" and meet functional requirements. It should address our problem statement and user stories.
- **Extensibility**: Over time, we can extend existing functionality or add new functionality. The design should accommodate this.
- **Scalability**: Our software should be scalable to increased demand e.g., more users, more transactions, greater throughput.
- **Robustness**: It should be stable, handle uncertain inputs and conditions, and recover from errors.
- **Reusability**: We should reuse design/code whenever possible and design our solution in a way that fosters reuse. (*However,* beware YAGNI*).*

Types of 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”.

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.

- Non-functional requirements are often referred to indirectly by users, especially when explaining how they use an existing system.
  - e.g., “Retrieving the report I need always take too long, so I keep a spreadsheet open and just copy the data myself”.
- You can log and track them like any other requirement.

Architectural Principles

How do we achieve well-designed software?

Architecture principles

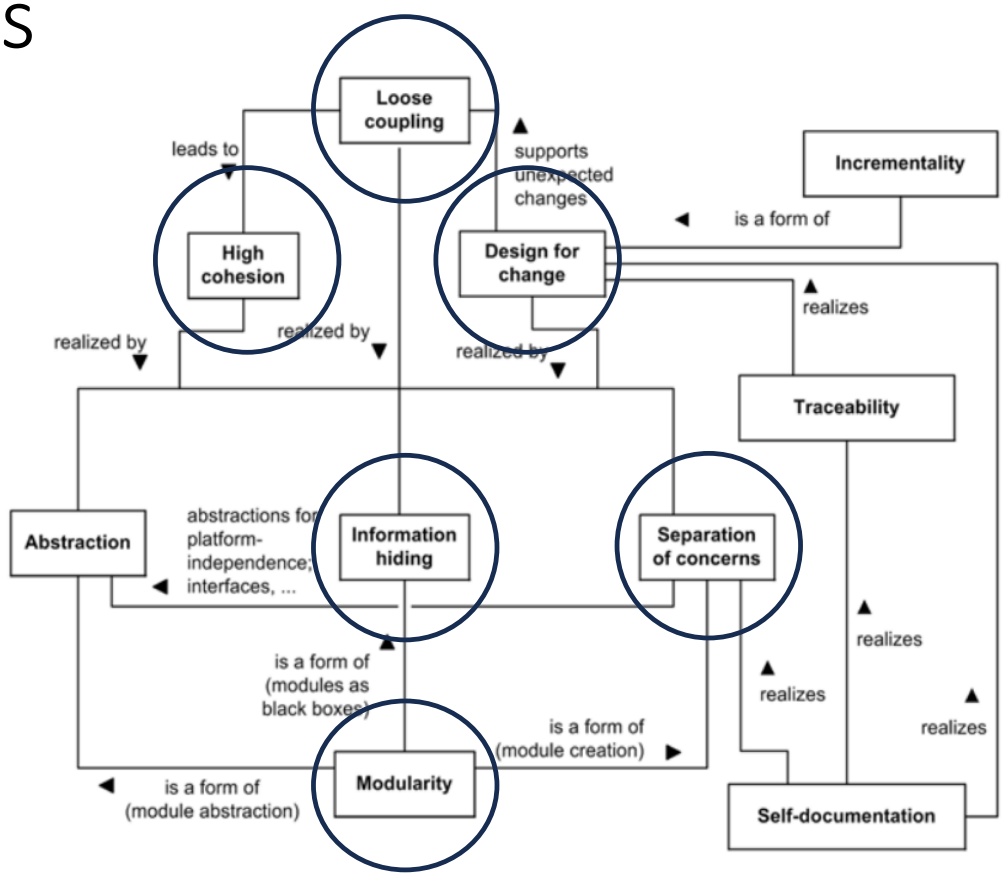
Our goals should include making our software:

- **Flexible** – *changeable over time*
- **Scalable** – *handle increase load*
- **Robust** – *manages changing conditions*
- **Reusable** – *modular, portions reusable*

These are aspirational goals.

Relative to some quality standard.

i.e. software is never “robust enough”, or “too reusable”.



- 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., a class or library with a specific purpose.
- Components can be classes, or functions, or module. It's "loosely defined" on purpose (but in practice often means "classes").
- This responsibility should be clearly defined and should not extend outside of this component.

Benefits

- Clearly defined roles results in "cleaner" code that is easier to read, maintain and test.
- Required for "separation of concerns" (next slide!)

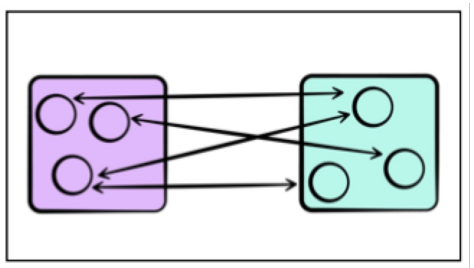
Separation of concerns

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.

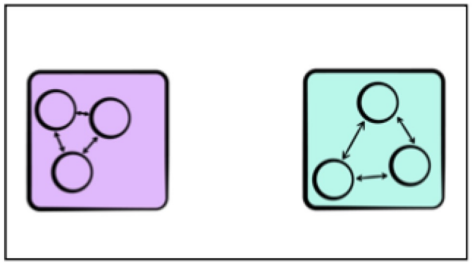
Benefits

- Clear separation == stable APIs (between components).
- Easier to make changes and changes tend to be isolated.



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

- The internal state of a function | class | module should not be visible outside of that entity.
- Multiple benefits
  - High cohesion: entities are responsible for their own state.
  - Minimized coupling: entities only communicate through well-defined interfaces.
  - Avoids side-effects and helps avoid ”accidental mutation”.
  - Makes entities more reusable if implementation can be changed with minimum impact on the rest of a system.

Modularity

Modularity refers to the logical grouping of source code into related groups i.e. 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.

Why is it important?

- Clear division of responsibilities, easier to manage code.
- Provides opportunities for code reuse e.g., exporting a distinct module.
- Greatly improves our ability to test!

Modularity -> Design for Change

- **Independent of frameworks**. The architecture does not depend on a particular set of libraries for its functionality.
- **Independent of the UI**. The UI can change easily, without changing the rest of the system. A web UI could be replaced with a console UI, for example, without changing the business rules.
- **Independent of the database**. You can swap out Oracle or SQL Server for Mongo, BigTable, CouchDB, or something else. Your business rules are not bound to the database.

Modularity (and *separation of concerns*) is of key importance going forward. 

Layered Architecture

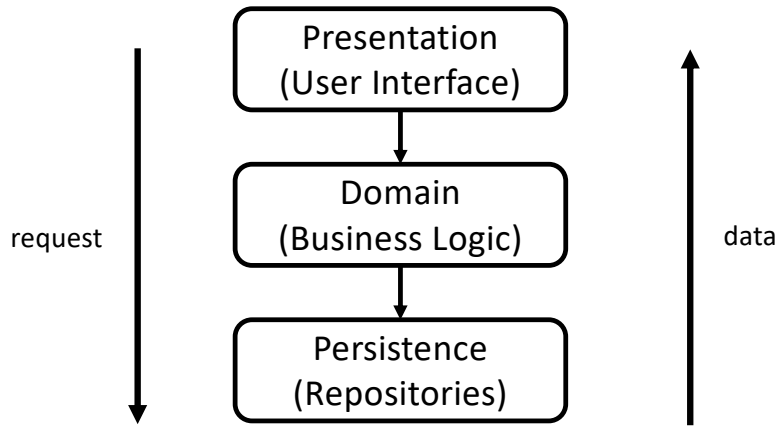
Mid-level component organization.

Layered architecture

A layered architecture is an **application pattern** that groups components into horizontal layers, where each layer represents a ***logical*** division of functionality.

Each layer communicates with the layer below it; requests flow top-down.

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



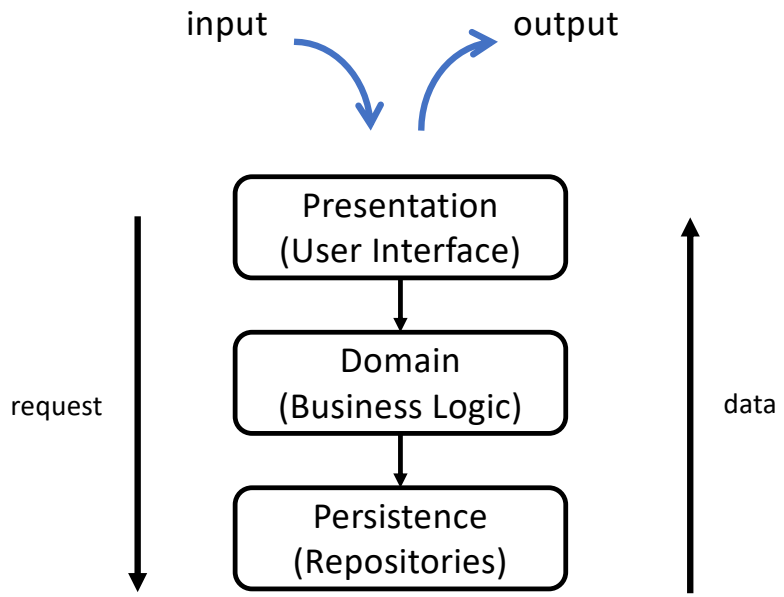
https://architectural-patterns.net/layers

Layered architecture

Why this structure?

- Most interactions are driven by the user.
- The domain layer contains most of the program logic, expressed in terms of the problem domain.
- The domain layer can push through requests to the persistence layer, which returns data/messages.

Dependencies flow inwards to the domain layer.



https://architectural-patterns.net/layers

Dependency Concerns

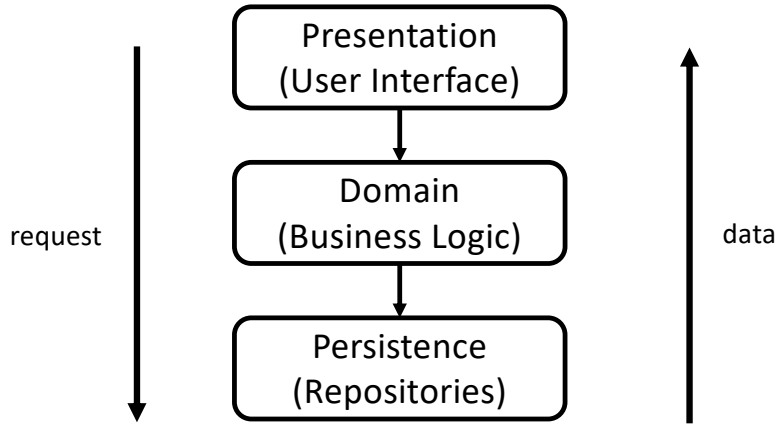
We need to make sure we avoid circular dependencies!

A **dependency** reflects a relationship between two modules

- e.g., if A → B, then A requires B.

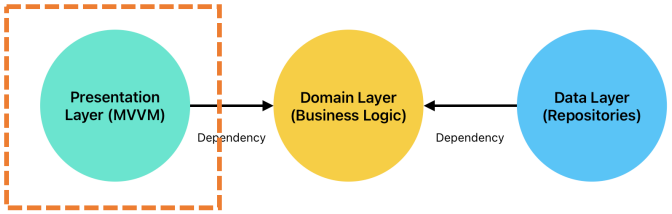
If A and B require each other, then we have a **circular dependency**.

- Circular dependencies make code more complex, difficult to debug and test.
- How do you create A without having a reference to B and vice-versa? How do you test them in isolation?



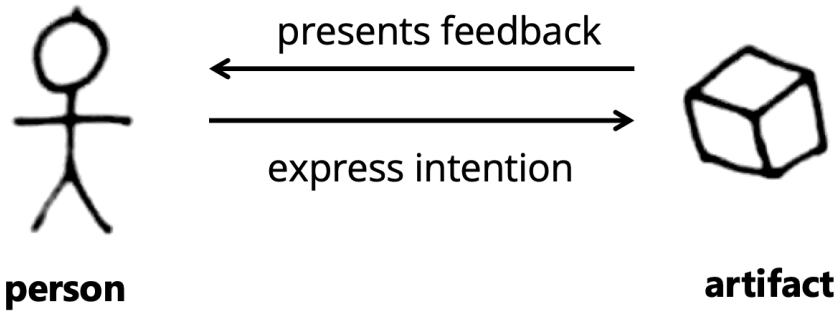
https://architectural-patterns.net/layers

1. Presentation layer

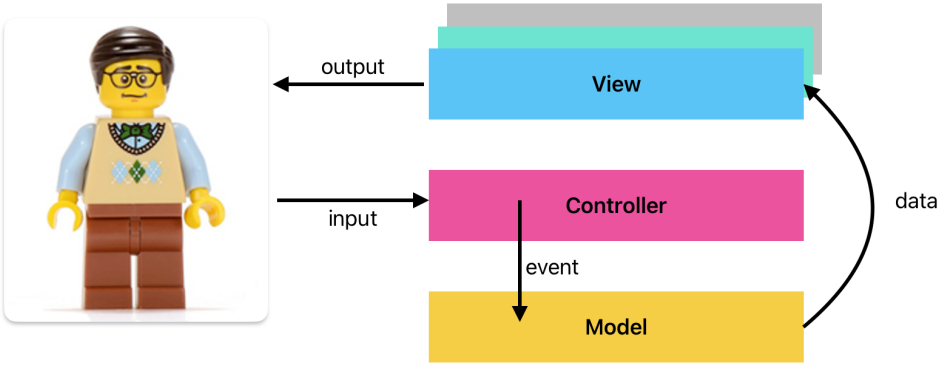


The presentation layer handles interaction between the user and the system:

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



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

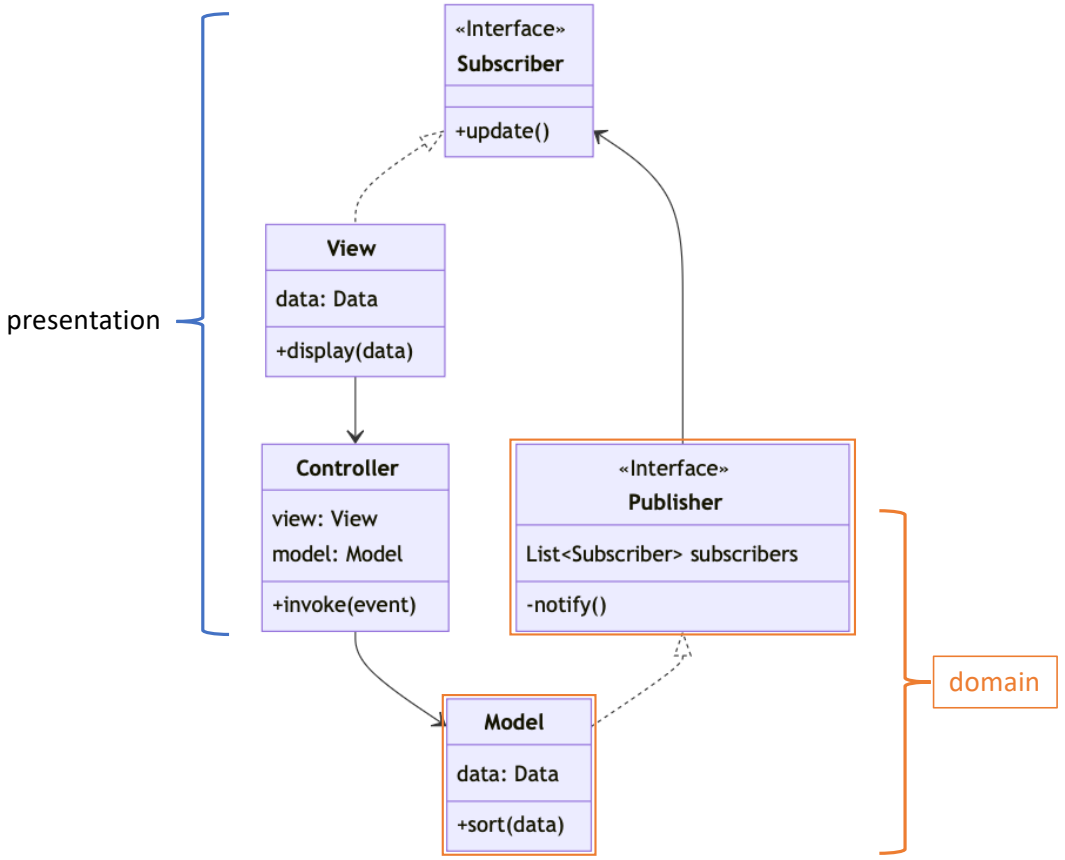
Components

- **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 (*we normally won't create one*).
- 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.
- MVC is fundamentally 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 global Model e.g., undo-redo where you don't persist the changes until the user clicks a Save button.

Model View View-Model (MVVM)

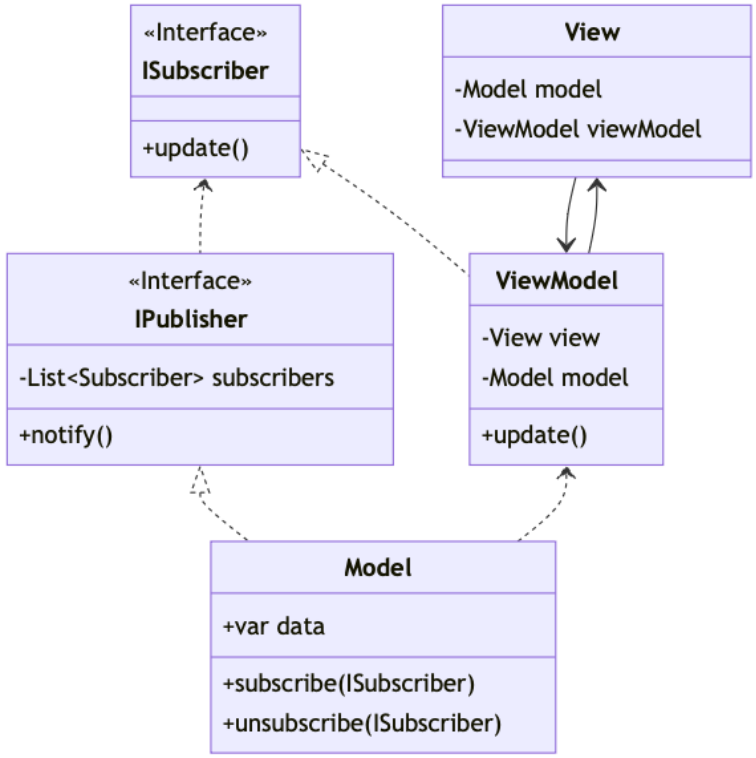
Components

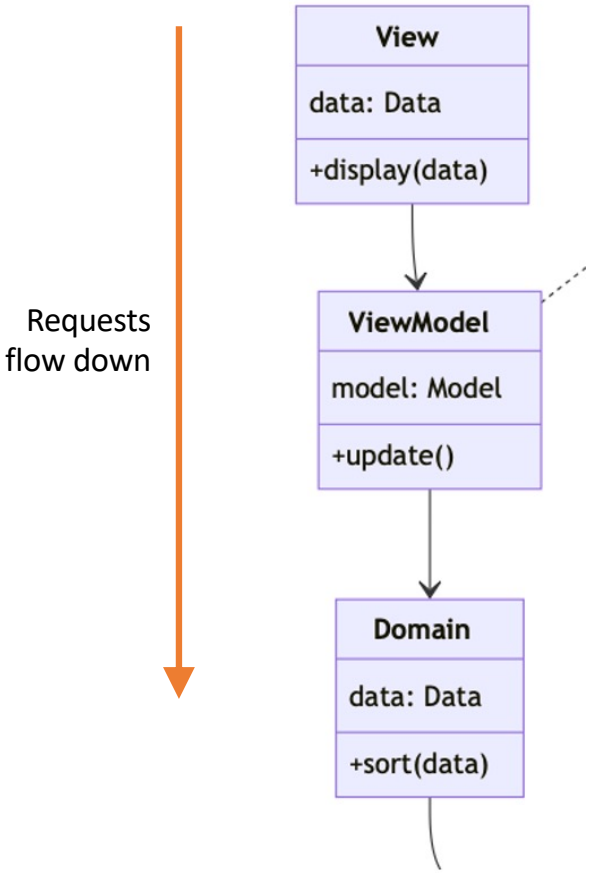
- **View**: input and output.
- **ViewModel**: localized data for the view.
- **Model**: stores the main data.

Each View typically has a matching ViewModel.

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

- Updating the ViewModel data will refresh the View.

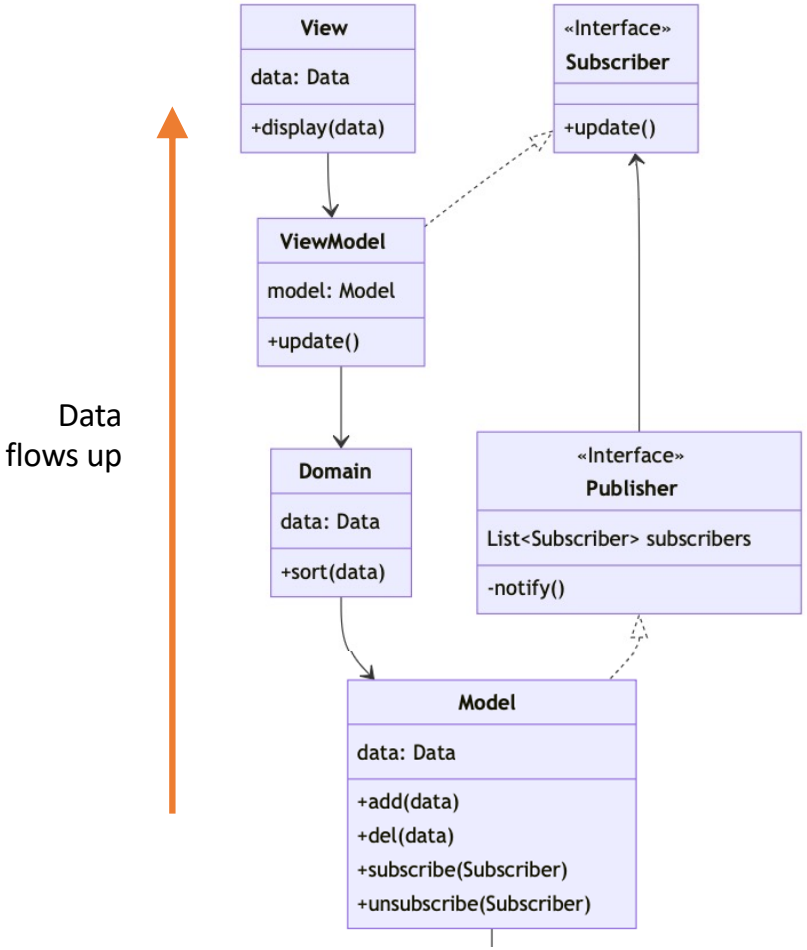




Dependency rule

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).

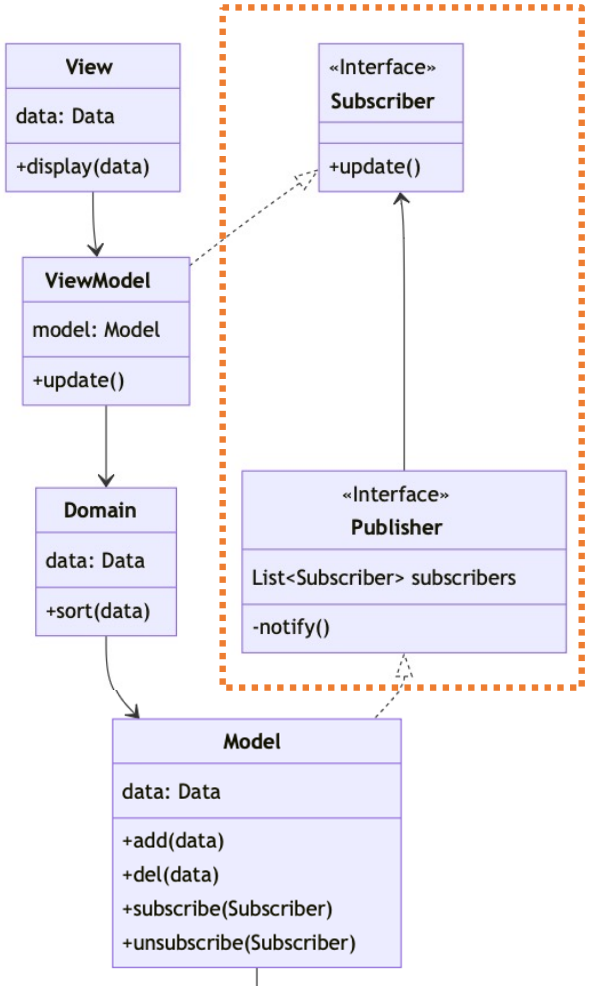


Update rule

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

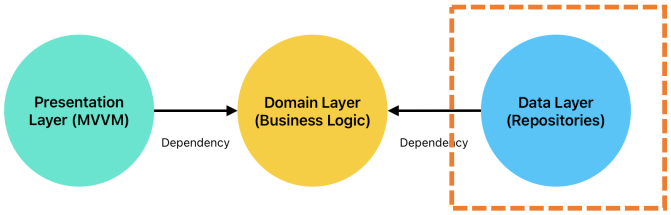
MVVM leverages the **Observer pattern**.

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 layer



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.

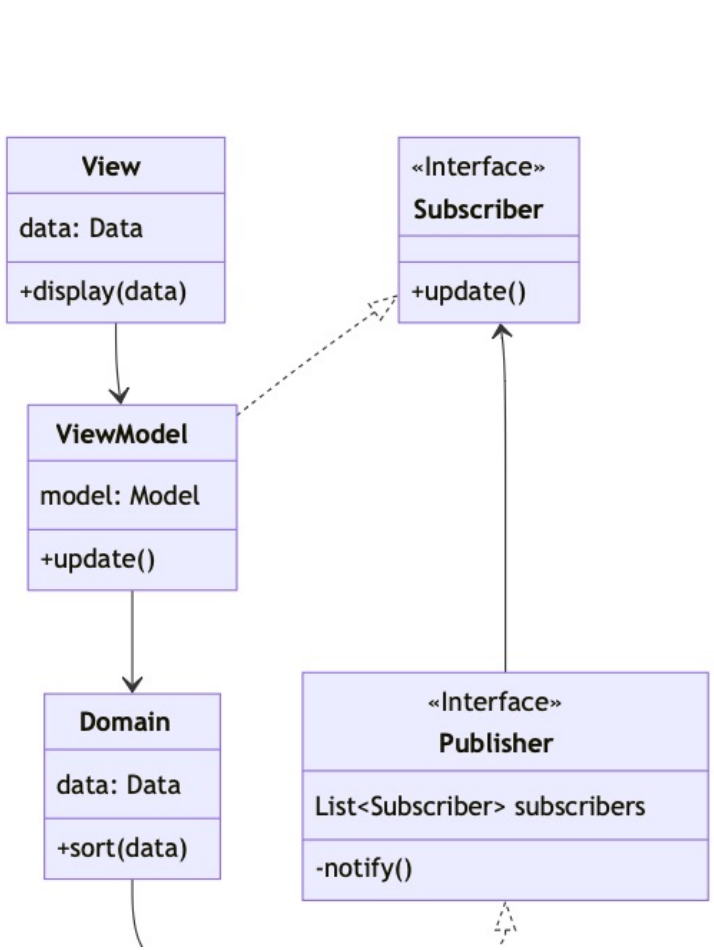
We use the term repository for a "source of data" and abstract it behind an interface.

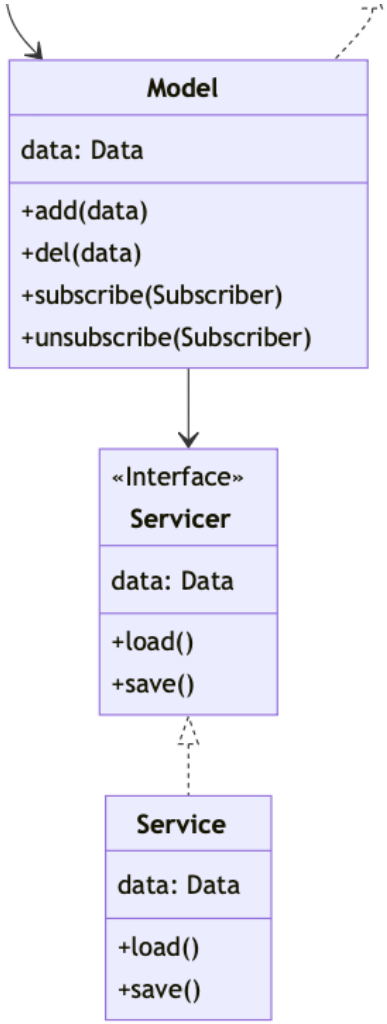
- **IRepository**: interface of common functions used across repositories.
- **Repository**: common term for any data source.

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

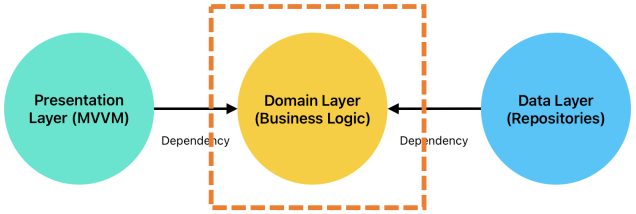
Classes in this package are allowed to access the `domain` package.

- 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 layer



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.

MVVM is popular

- Key characteristics:
  - Layered – benefits of modularity
  - Addresses dependency issues
  - Clear package structure

- Downside?
  - It models a very specific architecture.
  - What do you do when the data layer is more than a database? What about services?

Achieving Modularity

Packages? Multi-module structures? What to do?

Option 1: Using packages for modules

A **package** is a mechanism for grouping related classes, interfaces and other related code together.

```
presentation
├── controllers
|   └── ProductController.kt
|   └── StoreController.kt
└── views
    └── CustomerView.kt
    └── ProductView.kt
    └── StoreView.kt
```

```
package presentation.controllers

class ProductController {
    fun handlEvent(evt: Event): String {
        return "Handling event: $evt"
    }
}
```

```
package presentation.views
import  presentation.controllers

class ProductView {
    fun render(con: ProductController) {
        return "Rendering view: $con"
    }
}
```

Using packages

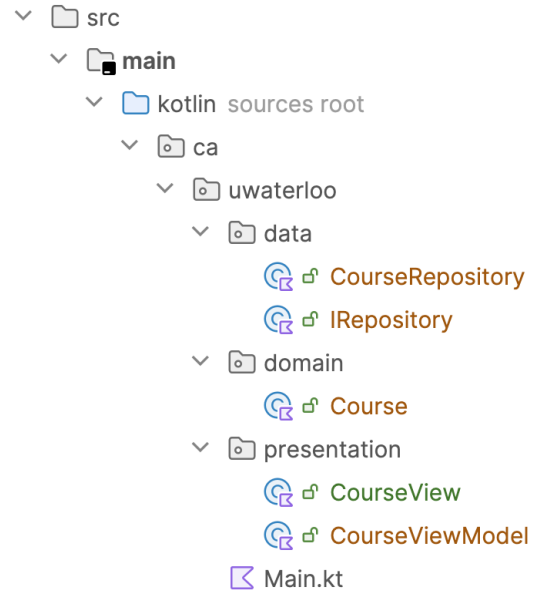
Create package structure for each level.

Benefits

- Easiest to implement!
- 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.



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

Option 2: Multi-module structure

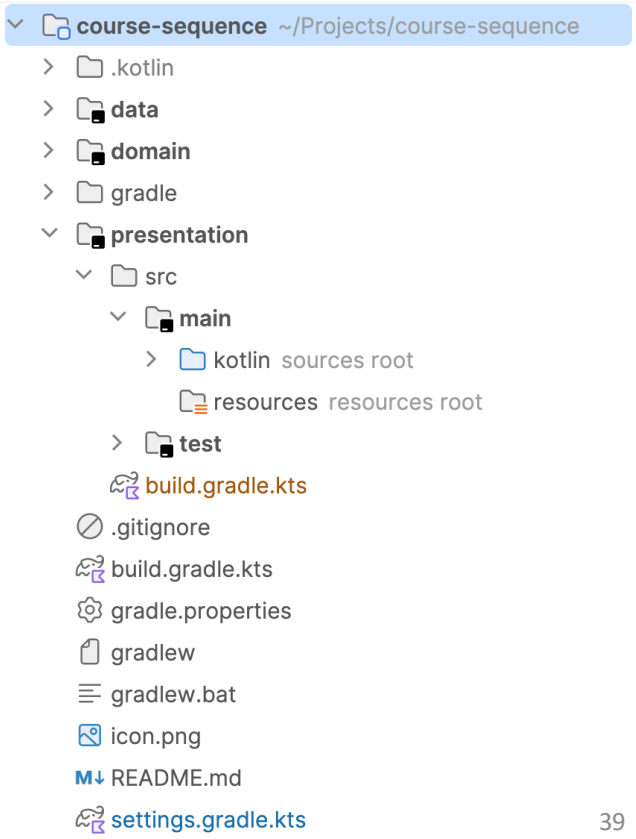
Modules for `/data`, `/domain`, `/presentation`.

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 and maintain



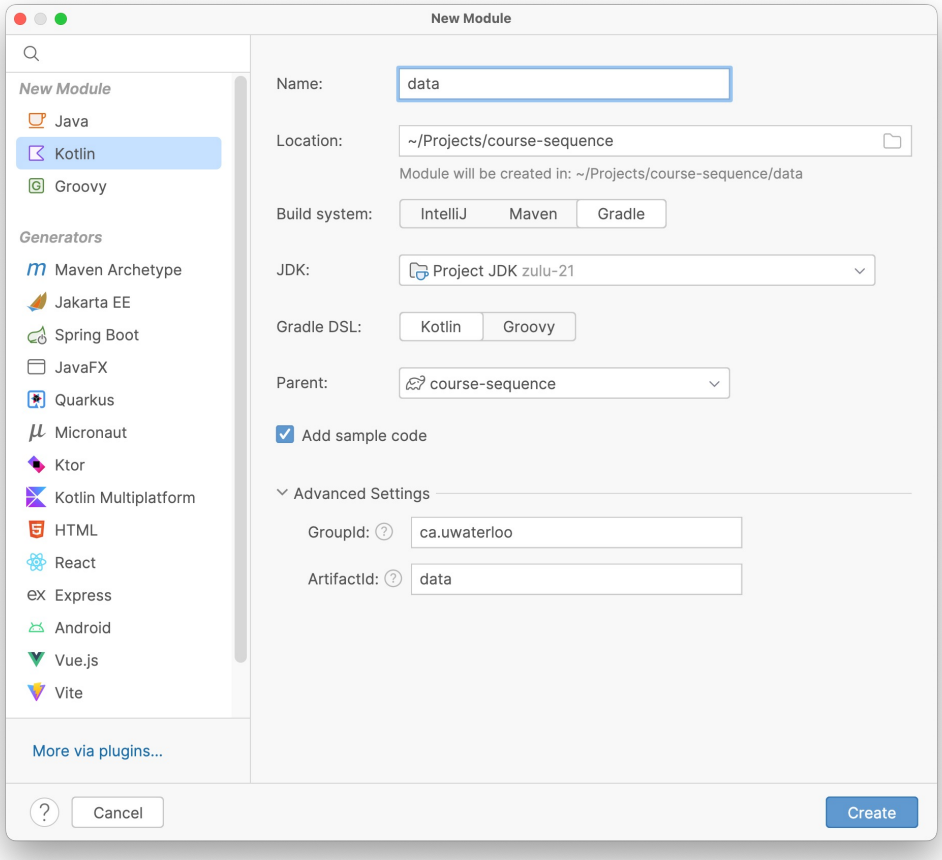
Adding modules

Modules:

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

Use

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



Benefits of Modularity

Layering our architecture provides these architectural benefits:

- **Independence from frameworks**. The architecture does not depend on a particular set of libraries for its functionality.
  - **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.
- **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.

mm-desktop

Review of the desktop application structure.

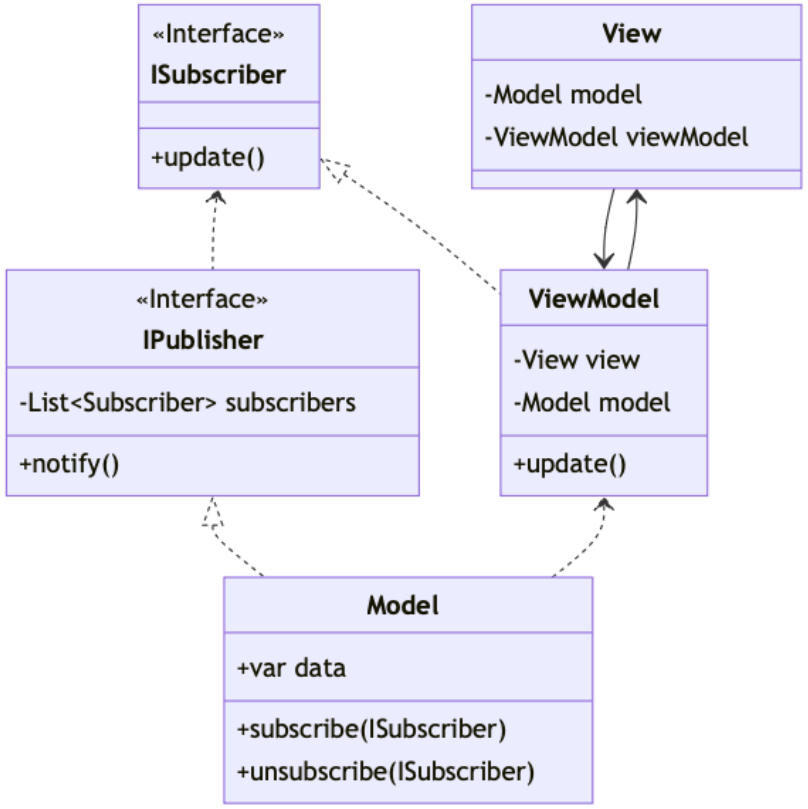
Recall: MVVM Implementation

Main classes

- **View**: input/output
- **ViewModel**: state for the view.
- **Model**: stores the application data.

There are often multiple views. Each View typically has one ViewModel associated with it.

MVVM uses the Observer pattern. The model typically notifies the ViewModel of state changes. The View and ViewModel are often tightly coupled so that updating the ViewModel data will refresh the View.



Main function

```
fun main() = application {
    MaterialTheme {
        Window(
            title = "Mastermind TODO",
            state = rememberWindowState(
                position = WindowPosition(Alignment.Center),
                size = DpSize(400.dp, 600.dp)
            ),
            onCloseRequest = ::exitApplication,
        ) {
            // wire dependencies together
            // storage <-- model <-- viewModel <-- view
            val storage = DBStorage(".mm.db")
            val model = Model(storage)
            val viewModel = ViewModel(model)

            // top-level composable
            View(viewModel)
        }
    }
}
```

Main
➔ View
➔ ViewModel
➔ Model
➔ Storage

Window setup

Dependencies

Root of scene graph

See GitHub: demos/mm-desktop

View

```
@Composable
fun View(viewModel: ViewModel) {
    val tasks = viewModel.tasks
    val scaffoldState = rememberScaffoldState()
    var showAddDialog by remember { mutableStateOf(false) }

    Scaffold(
        scaffoldState = scaffoldState,
        topBar = {
            TopAppBar(
                title = { Text("Task Manager") },
                actions = {
                    IconButton(onClick = { showAddDialog = true }) {
                        Icon(Icons.Default.Add,
                                contentDescription = "Add Task")
                    }
        // ……
```

See GitHub: demos/mm-desktop

ViewModel

```
class ViewModel(private val model: Model) : Subscriber {

  var tasks by mutableStateOf(model.tasks.filterNotNull())
  init { model.add(this)}

  override fun update() { tasks = model.tasks.filterNotNull()}

  fun addTask(title: String) { model.add(title)}

  fun deleteTask(position: Int) { model.del(position)}

  fun updateTask(task: Task) {
    val existingTask = model.tasks.find { it?.id == task.id }
    if (existingTask != null) {
      existingTask.title = task.title
      // update more things
      model.notifySubscribers()
    }
  }
}
```

Receives updates from the Model

See GitHub: demos/mm-desktop

Model

```
class Model(private val storage: IStorage): Publisher() {
    var tasks = mutableListOf<Task?>()

    init {
        tasks = storage.readAll().toMutableList()
    }

    fun add(contents: String) {
        val task = Task(position = tasks.size + 1, title = contents,
          description = "", dueDate = "", tags = ""
        )
        storage.create(task)
        tasks.add(task)
    }

    fun del(position: Int) {
        val pos = position
        val task = tasks.find { it?.position == pos } ?: return
        storage.delete(task)
        tasks.remove(task)
    }
}
```

Sends updates to the ViewModel

See GitHub: demos/mm-desktop

Storage interface

```
interface IStorage {
    // canonical operations
    fun create(task: Task): Int
    fun read(id: Int): Task?
    fun readAll(): List<Task?>
    fun update(task: Task)
    fun delete(task: Task)
    fun deleteAll()

    // extended operations
    fun upsert(task: Task)
}
```

See GitHub: samples/demos/mm-desktop

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.