Architecture

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. It works because **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).

Software qualities

Software will often have specialized requirements e.g., startup time may be important for a web service, but less important for a desktop app.

Here are characteristics that we should consider for any software we design:

- **Usability**: Software is “fit for purpose” and meets requirements.
- **Extensibility**: We can extend functionality or add new functionality.
- **Scalability**: Software can grow to increased demand e.g., more users.
- **Performant**: Software should be “fast enough” for its purpose.
- **Robustness**: Software should be stable and reliable.
- **Reusability**: We should reuse design/code whenever possible.

> Usability

When designing systems, we need to explicitly identify the problem we are solving, detailed requirements that must be met. ***Usability refers to how well our solution meets product requirements***.

- **Functional requirements** refer to the functionality of your application. e.g., “I want to be able to display a report in this format”.
- **Non-functional requirements** refer to the qualities of our software. e.g., power-consumption, startup-time, records processed/second.
  - To address these, it’s important to understand and quantify requirements, so that we can measure them to know if they have been achieved.
- Much of our project management overhead is tracking requirements!

> Extensibility/Flexibility

Extensibility or flexibility implies the ability to expand our features, without compromising the *existing* features. It’s the opposite of “brittle code”.

The challenge in building flexible systems is that it’s easy to “over-engineer” a generalizable solution, for something that will never be needed (or conversely, made a system so rigid that it’s impractical to adapt it later).

Examples:

- add a new image format to an image editor (PNG).
- add a new payment method (Visa) to a payment system.
- add a new input modality (support both kb + voice dictation).

> Robustness

Software rarely works in a vacuum.

- Your operating environment may change (OS, libraries may be updated).
- You are probably getting inputs from many sources (messages, data files, user input). Sometimes they are in a format you don’t expect.
- These things can result in expected behavior.

Robustness means that your software needs to continue to work correctly, even when faced with unintended inputs, or changes to the operating environment.

- It cannot crash. Ever. Manage errors and attempt to resume. Log details.
- Performance and other characteristics should remain constant over time.
- Data should never, ever get lost.

> Reusability 1/2

Software is expensive and time-consuming to produce, so anything that reduces cost is welcome. Code reusability helps to reduce cost and time to delivery.

- It’s usually faster to repurpose something you’ve already written than produce it.

Reusability reduces risk, since you are reusing tested code, instead of writing new, potentially defective code.

- New code is always risky until proper testing is complete (which takes time and cost to do).
- You should always reuse existing, tested code when possible.

Reusability is also normally an implicit requirement i.e., something that you are assumed to do as a best-practice; often not an explicit project goal.

What is architecture?

These concerns are in the domain of software architecture:

> 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*.

Software structure has a significant impact on our ability to deliver these qualities, as well as on the qualities themselves.

Architecture determines qualities

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

- Some architectural decisions are necessary for your software to work properly, or at-all.
  - e.g., handling remote data efficiently requires specific design choices.
  - e.g., scaling to larger amounts of data also requires specific design considerations.
- The structure of your software will determine how well it runs, how quickly it performs essential operations, how well it handles errors.

Poorly design software is frustrating to work with, difficult to evolve and change. It also takes longer and is more costly to build and maintain.

Architectural principles

What does architecture ”look like”? How can it help?

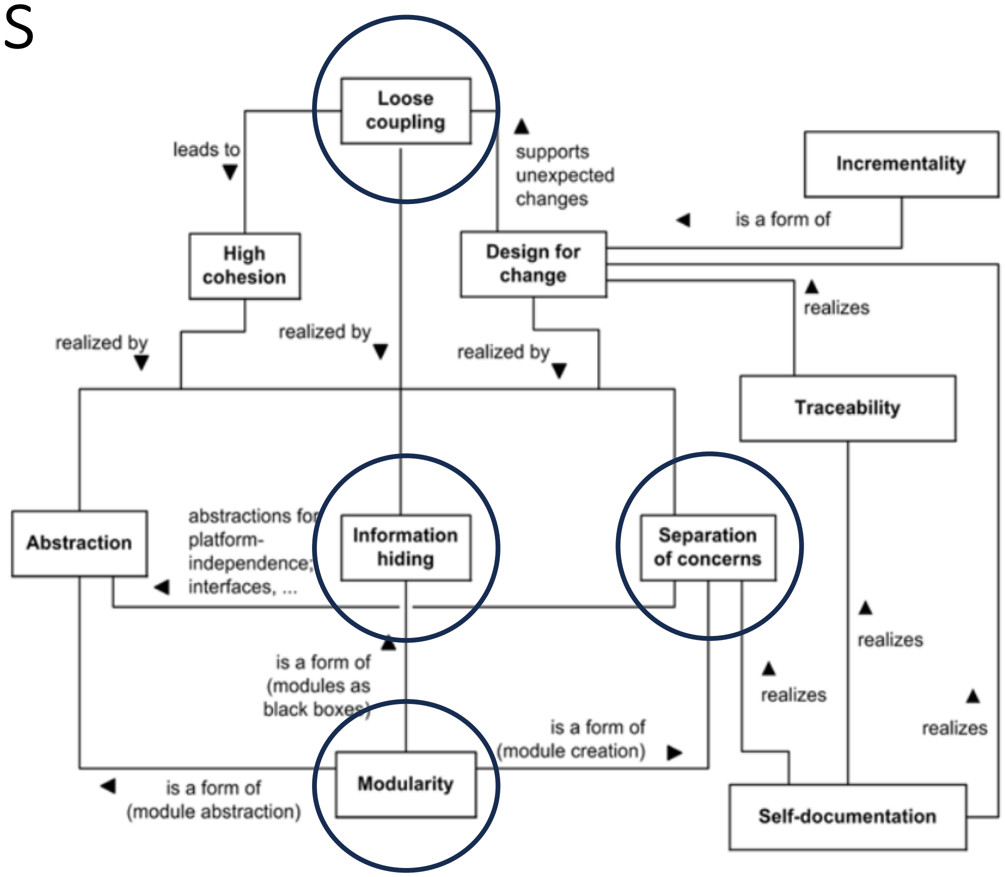
Architecture principles

How do we meet our goals?

- We apply architectural principles that improve our ability to maintain and extend our software.

We'll focus on these:

- Loose coupling & high cohesion
- Modularity
- Separation of concerns
- Information hiding



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

Coupling & cohesion

Loose Coupling: reduce coupling between components as much as possible; functionality should be isolated.

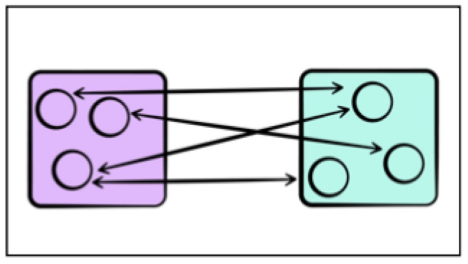
High Cohesion: class/module should be self-contained.

These work together:

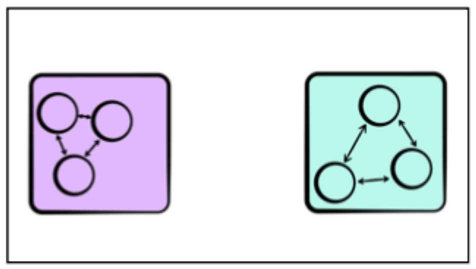
- Modules are easier to understand if is self-contained.
- Modules are easier to modify if changes are contained.

Examples of coupling:

- **High**: classes access each other's data directly. (BAD)
- **Medium**: classes share a global data structure. (BETTER)
- **Low**: classes communicate through public methods. (BEST)


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.

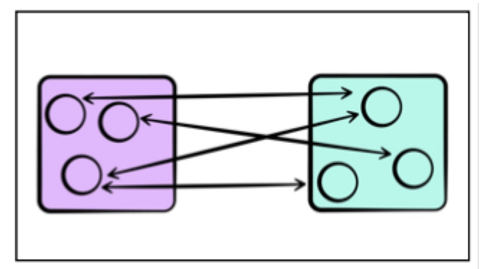
Aim for low coupling

Single Responsibility Principle

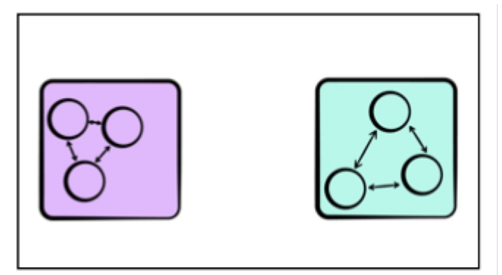
- Classes (or similar code structures) should have a single principle in their design.
- We will use modularity to reduce interdependencies as much as possible.

Avoid global data structures

- If you have them, then you tend to use them from multiple places – leading to coupling.



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



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

Modularity

Modularity refers to the logical grouping of source code into related groups e.g., namespaces (C++), or packages (Kotlin). Modularity enforces a separation of concerns and encourages reuse of source code.

Kotlin can support modularity in multiple ways, representing different levels of abstraction.

- **Modules**: we use modules for top-level components or deliverables.
  - Restrict module use to platform targets, or shared libraries/components.
- **Packages**: groups of logically related functionality.
  - Packages can contain related classes, functions. e.g., views, models.
  - This is your main mechanism for modularity.

Use modules and packages

You create a hierarchy of packages (folders) for your code.

How you group your code is called partitioning:

- **Technical partitioning**: group according to technical capabilities. e.g., all models for all features are grouped, all views are together and so on.
- **Domain partitioning**: group according to the area of interest. e.g., each folder represents a feature, which may have its own view, view-model and model.

```
└── src
    ├── main
    ├── kotlin
    │   └── shared
    │       ├── common
    │       ├── database
    │       ├── domain
    │       ├── model
    │       ├── service
    │       └── settings
    └── resources
        └── local.properties
```

A partial source tree.

The `shared’ module has multiple packages, reflecting different application areas e.g., domain, model, service. This is an example of **technical partitioning**.

Adopt a suitable architectural style

An architectural style (aka *pattern*) is an **overall structure that describes how our components are organized and structured, and how they communicate**.

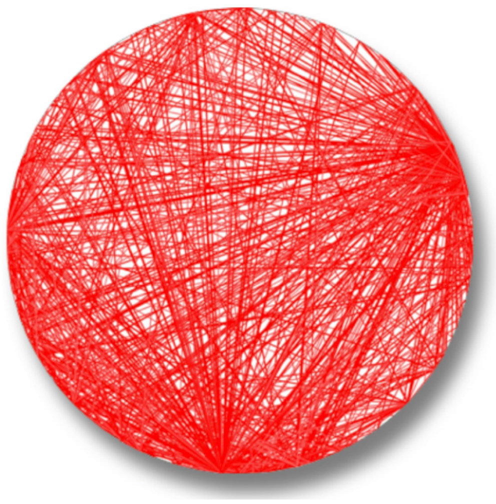
- Each style describes an example of **modularity** + **class relations**.
- Like design patterns, an architectural style is a general solution that has been found to work well at solving specific types of problems.
- An architectural style has a unique **topology** (organization of components) and **characteristics** (qualities) for that topology.

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 isn’t intentional—it’s the result of a system being *tightly coupled*, where any module can reference any other module. A system like this is *extremely* difficult to extend or modify.

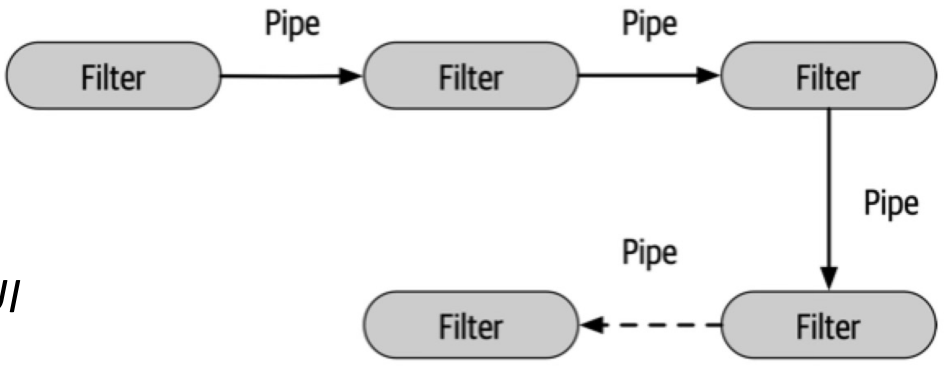
Console: Pipeline Architecture

A pipeline architecture transforms data in a sequential manner. e.g., streams.

Usually one outbound starting point (source) and one or more inbound termination points (sinks).

- **Pipes** are unidirectional, accepting input, and producing output.
- **Filters** are entities that perform operation on data that they are fed. Each filter performs a single operation, and they are stateless.
  - Easy to extend by adding nodes.
  - Filters are stateless, and testable.
  - Broadly applicable.

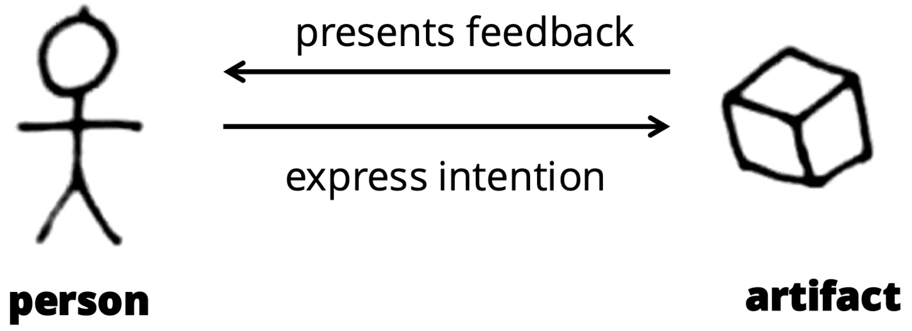
Can we produce something like this for GUI applications?



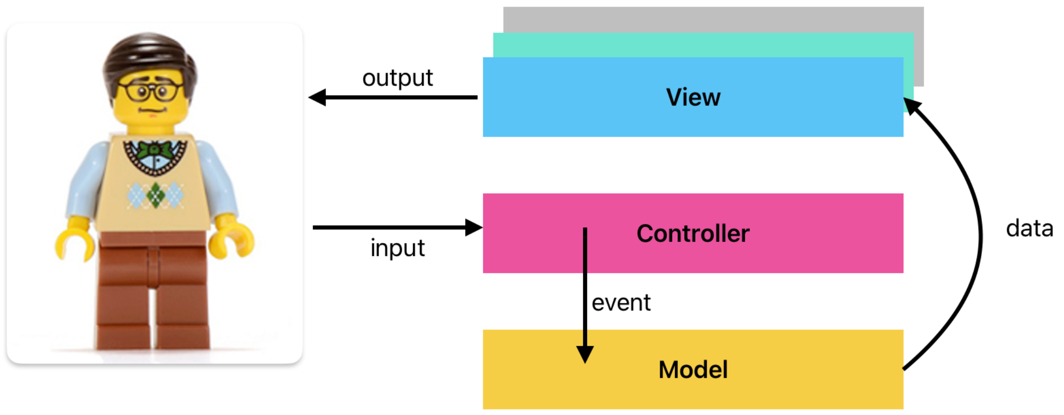
GUI: Which architecture?

GUI applications are designed to processing iterative commands from the user and display the results. Designed around an **interaction loop**:

- accept **input** for text (keyboard), positional input (mouse, touch)
- produce **output** in response to changes in state.



Attempt 1: Model-View Controller



MVC originated with Smalltalk (1988).

- Input is accepted and interpreted by the **Controller**, and
- Data is routed to the **Model**, where it changes the program state (in some meaningful way).
- Changes are published to the View(s) so that they can be reflected to the user in the **View**.

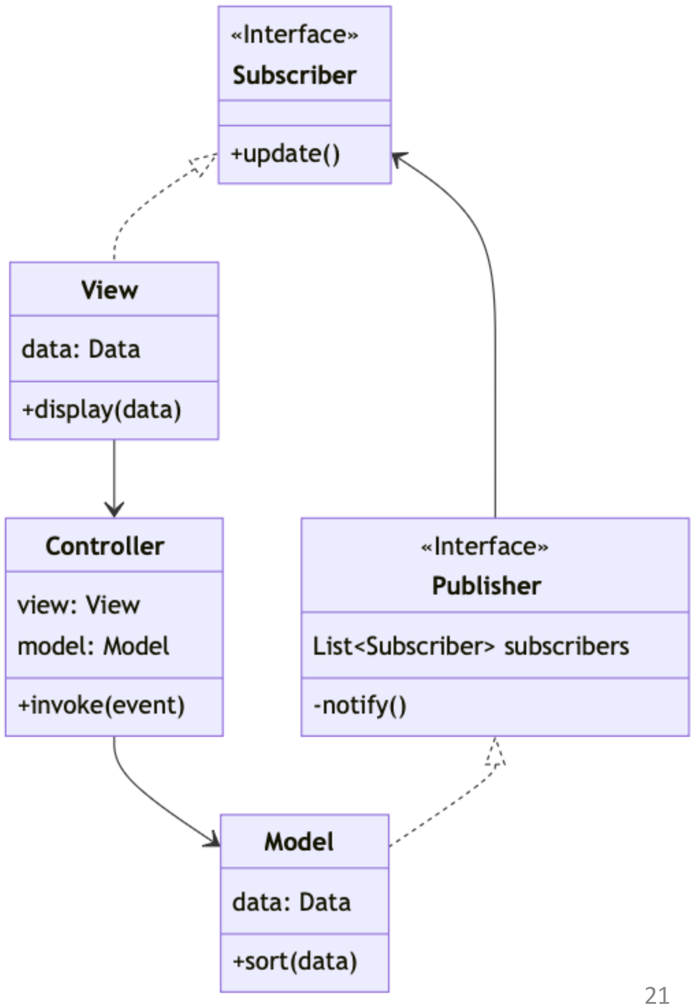
MVC Implementation

Components
- **View**: displays data (or a portion of it)
- **Controller**: handles input from the user.
- **Model**: stores the data.

There are often multiple views.

MVC uses the Observer pattern to notify Subscribers. Any Subscriber (i.e. any class that implements the interface) can accept notification messages from the Publisher.

This is “standard” MVC. There are many variations!



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.
- This architecture is completely standalone.
  - How do you handle services? Databases?

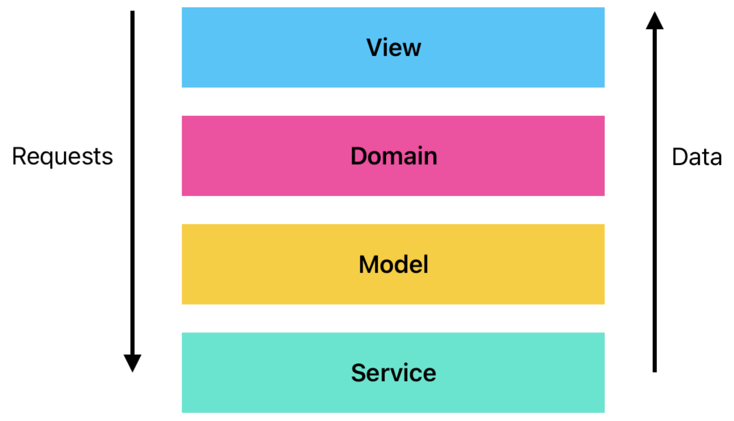
Attempt 2: Layered Architecture

Let's generalize the MCV approach.

- Remove the controller.
- Add a Domain layer for UI specific data.
- Add Services e.g., DB, web API.

How does it work now?

- Input is accepted by the **View** aka UI layer.
- The **Domain** layer does any screen-specific logic, and forwards requests to the **Model**.
- The **Model** interacts with the **Service** (which could be a DB or web API) as needed.
- The **Service** returns data to the **Model**, which then returns data up the chain.

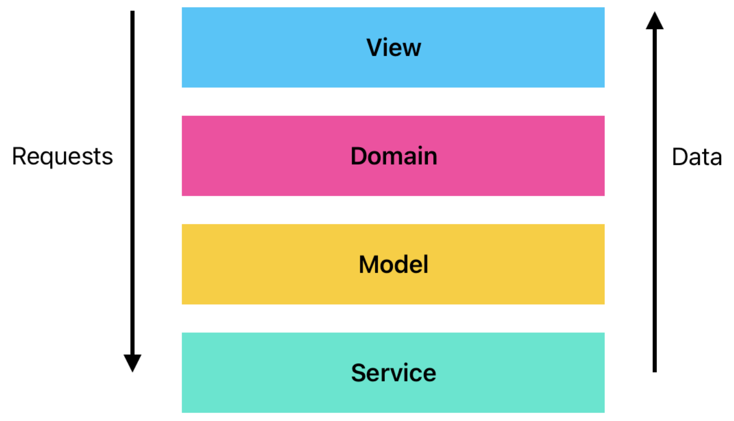


Layers remain isolated ("separation of concerns")
High testability: components in specific layers.
High ease of development.

Attempt 2: Layered Architecture

A **layered** or *n-tier* architecture organizes software into horizontal layers, where each layer represents a ***logical*** division of functionality.

- Each layer has specific functionality that is presents to the layer above (i.e. lower layers provide services to layers above).
- Requests flow down, and data flows up.
- *This also means that dependencies extend down*.



There is a clear separation of concerns.

- Each layer is independent, and testable.
- Use **dependency injection** to decouple layers.

Remember: these are layers, and each may require multiple classes to implement them.

MVVM

Implementation time!

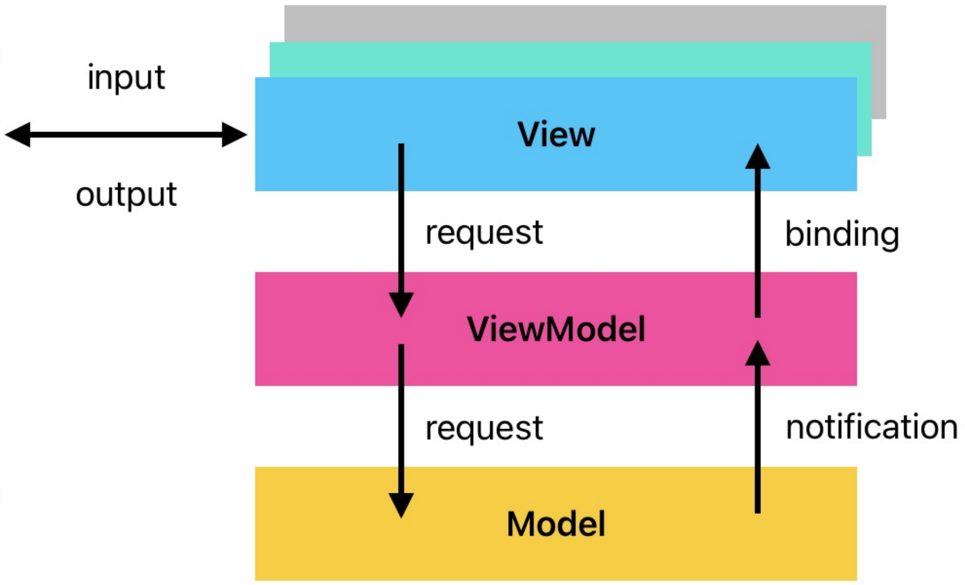
Model View View-Model (MVVM)

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**: As with MVC, the Model is the Domain object, holding application state.
- **View**: The presentation of data to an output device. Handles both input and output.
- **View-Model**: A component that stores data that is relevant to the View to which it is associated. This may be a subset of Model data but is more often a reinterpretation of that data in a way that makes sense to the View.

Layered architecture

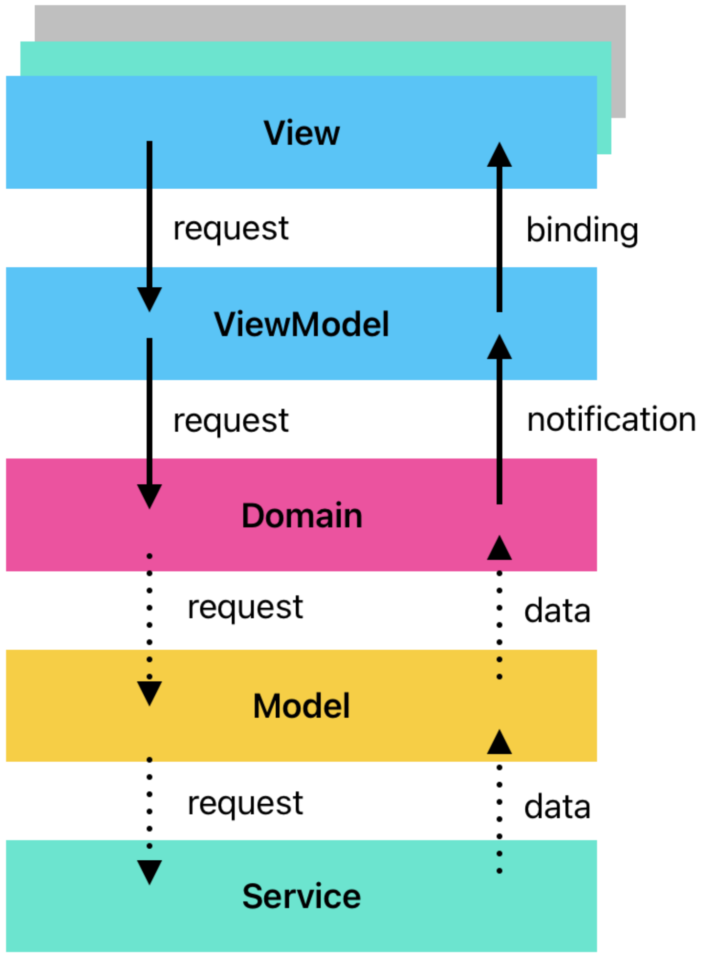
Refinement from our layered architecture:

UI layer consists of View + VM

- Each View has one VM.
- Requests flow from VM to Domain classes.

Same flow as earlier

- Requests down, data up.
- Notification used with VM, which in turn propagates data into Views.

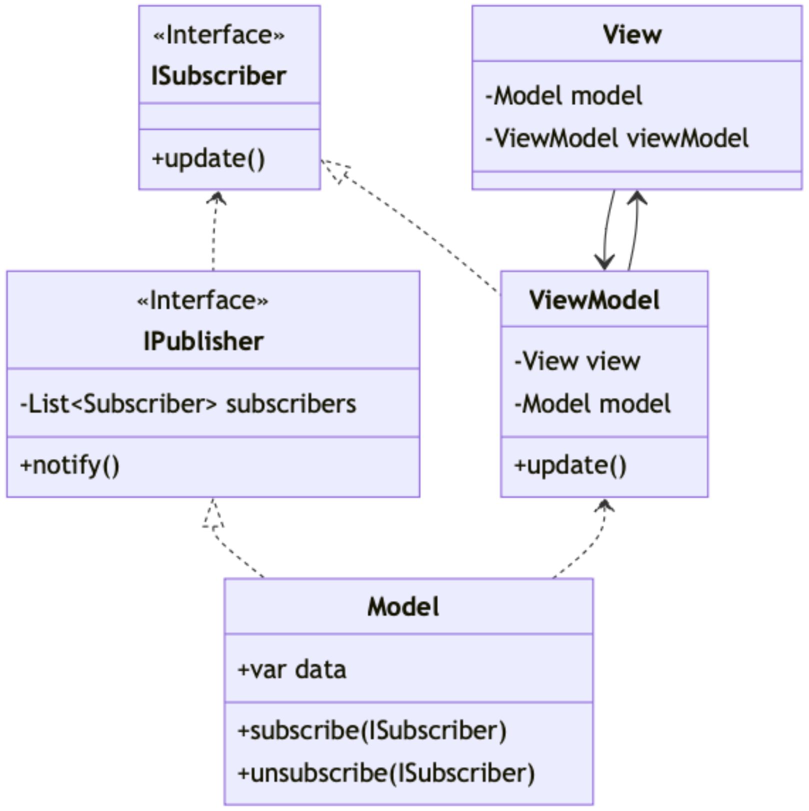


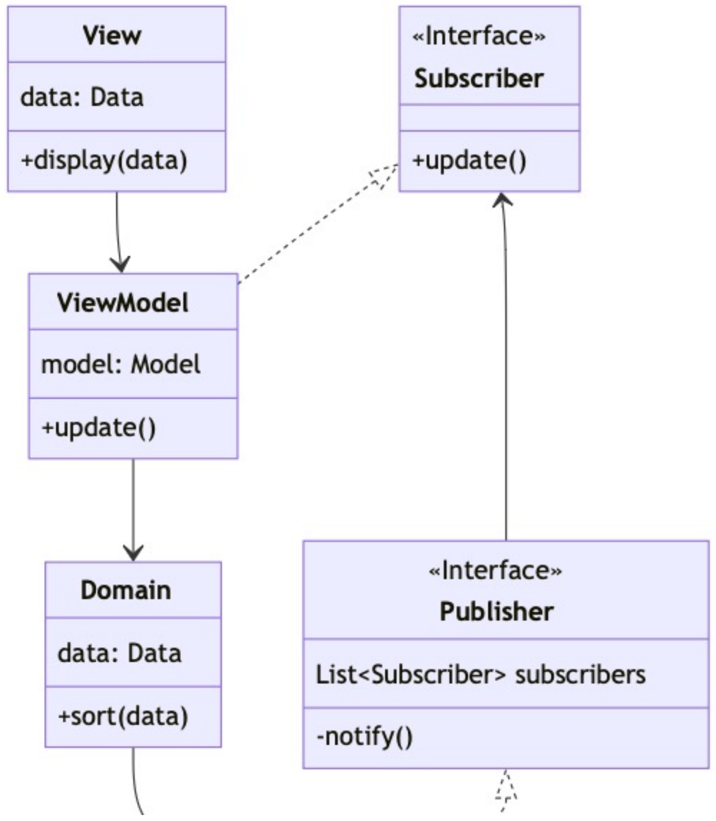
MVVM Implementation

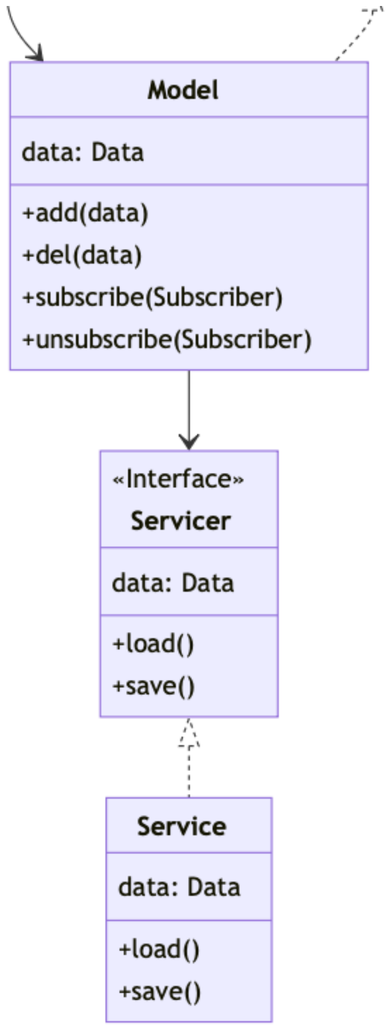
- **View**: displays data (or a portion of it)
- **ViewModel**: localized data for the view.
- **Model**: stores the main data.

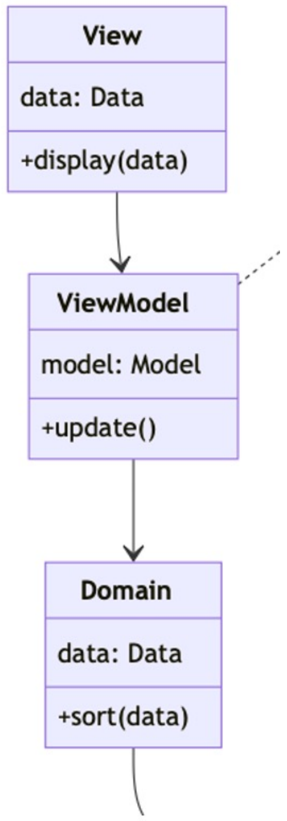
There are often multiple views. They may each display different data, or views may display the same data. Each View typically has one ViewModel associated with it.

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.




View
data: Data
+display(data)
«Interface»
Subscriber
+update()
ViewModel
model: Model
+update()
Domain
data: Data
+sort(data)
«Interface»
Publisher
List<Subscriber> subscribers
-notify()

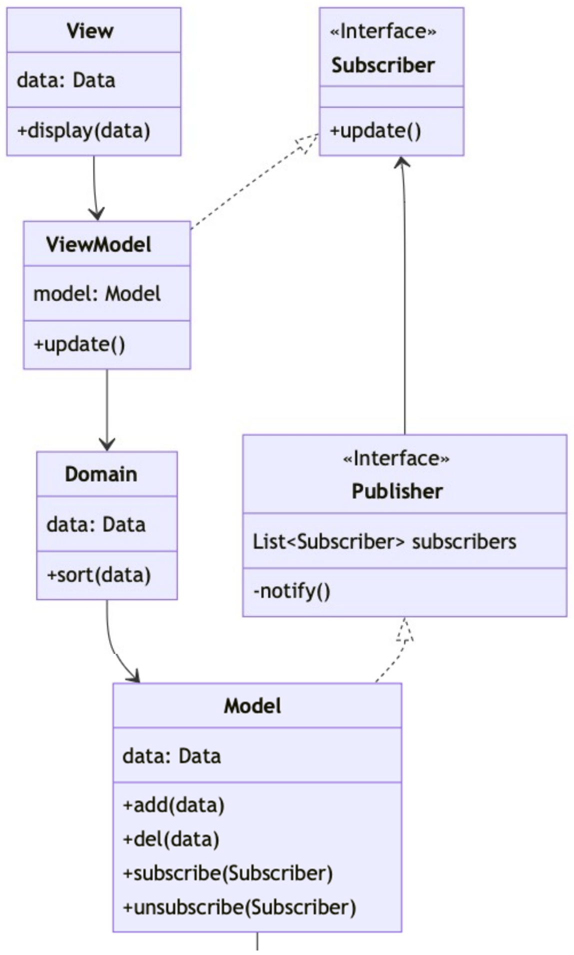

Model
data: Data
+add(data)
+del(data)
+subscribe(Subscriber)
+unsubscribe(Subscriber)
«Interface»
Servicer
data: Data
+load()
+save()
Service
data: Data
+load()
+save()



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).



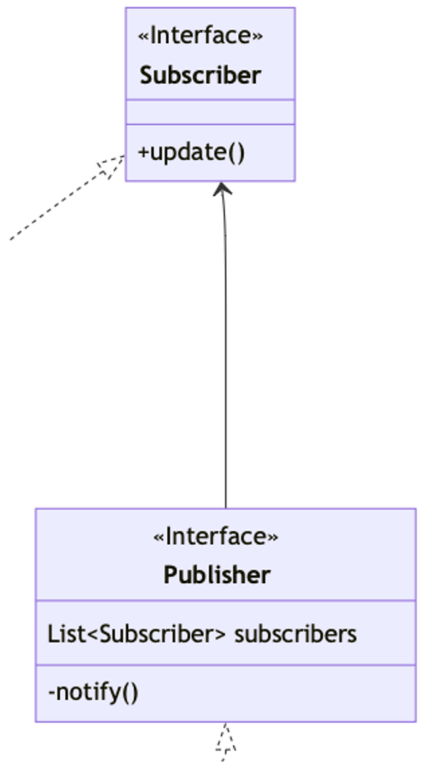
Update rule

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 "up".

Use interfaces



We use interfaces instead of inheritance, when possible, to encourage loose coupling.

- This provides flexibility in what classes can participate in notifications and other updates.
  - e.g., a view could be a screen, or a printer, or a text-to-speech device.
  - e.g., an input device could be a mouse, or pen, or touchpad.
  - e.g., our service could be a database or a web api.
- We also use dependency injection: class instances are never created as part of one class's constructor; we create them externally and pass them in at the call site.

Benefits

Layering our architecture really helps to address our earlier goals (reducing coupling, setting the right level of abstraction). Additionally, it provides these other benefits:

- **Independence from frameworks**. The architecture does not depend on a particular set of libraries for its functionality. This allows you to use such frameworks as tools, rather than forcing you to cram your system into their limited constraints.
- **It becomes 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 Mongo, BigTable, CouchDB, or something else. Your business rules are not bound to the database or to the source of your data.

Reference

John Ousterhout. 2018. **A Philosophy of Software Design**. Yaknyam Press. ISBN 978-1732102200.

Martin Fowler. 2002. **Patterns of Enterprise Application Architecture**. Addison-Wesley. ISBN 978-0321127426.

Robert C. Martin. 2017. **Clean Architecture**. Prentice Hall. ISBN 978-0134494166.

Shvets. 2021. Refactoring Guru: Design Patterns