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*.

What is architecture?

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.
- The structure of your software will determine how well it runs, how quickly it performs essential operations, how well 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.

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

We have explicit (stated) and implicit (assumed) goals for any software.

- **Usability**: Our software is "fit for purpose" and meets functional requirements. It should address our user stories and problem statement.
- **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, 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*).*

Usability

Functional requirements refer to explicit requirements, 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".

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

- User can and often will ask for non-functional requirements, but often 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 (be precise in your goals!) and measure them to know if they have been achieved.
- Usability requirements should be explicit and tracked as project goals.

Extensibility

Extensibility implies the ability to expand our features, without compromising the *existing* features. It's the opposite of "brittle code".

This is an implicit quality of our software (i.e. "something we do as part of building it"); it's part of the "craftsmanship" side of software development.

Examples:

- add a new image format to an image editor (PNG).
- add a new payment method (Visa) to a payment system.
- change the graphics features that you support in your game.
- add a new input modality (support both kb + voice dictation).
- a text editor adding support for code fences and syntax highlighting.

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.

How easy is it to achieve reuse?

Reusability 2/2

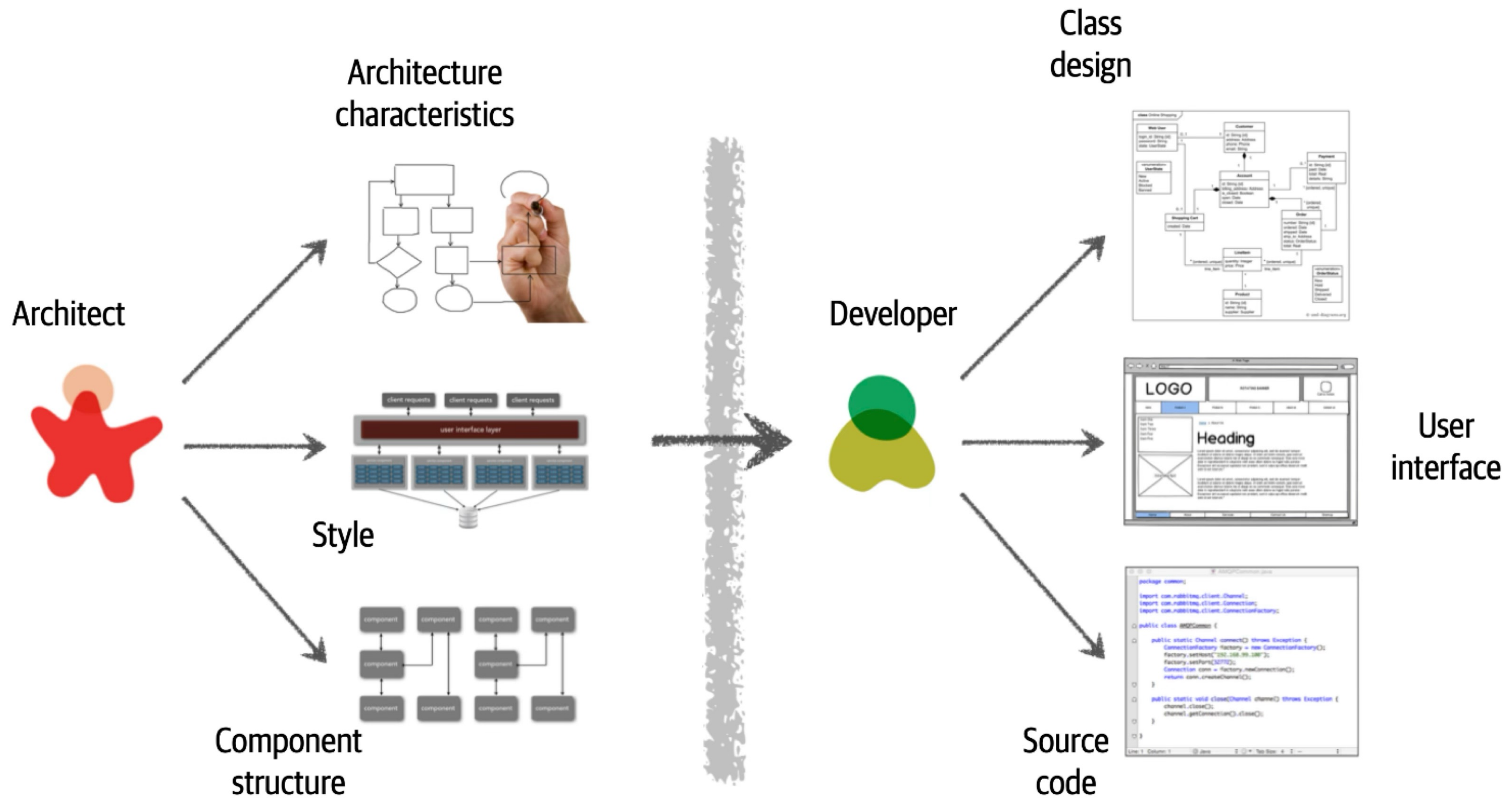
There are different levels of reuse:

- At the lowest level, you reuse **classes**: class libraries, containers, maybe some class functionality like container/iterator.
- At the highest level, you have **frameworks (or libraries)**. They identify the key abstractions for solving a problem, represent them by classes and define relationships between them.
- "There also is a middle level. This is where I see patterns: **design patterns** are both smaller and more abstract than frameworks. They're really a description about how classes can relate to and interact with each other" (reusing knowledge?)

— Eric Gamma (2005)

Architectural principles

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



Mark Richards & Neal Ford. 2020. **Fundamentals of Software Architecture: An Engineering Approach.** O'Reilly.

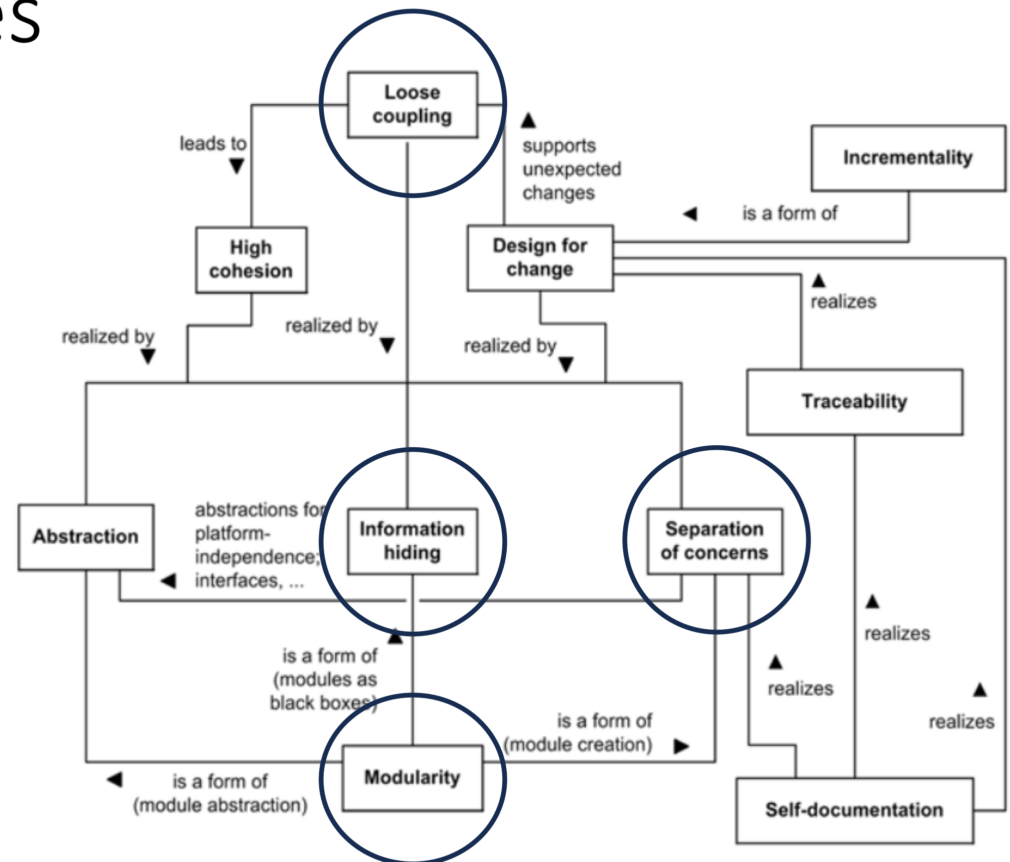
Architecture principles

How do we meet both explicit and implicit 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 within a component.

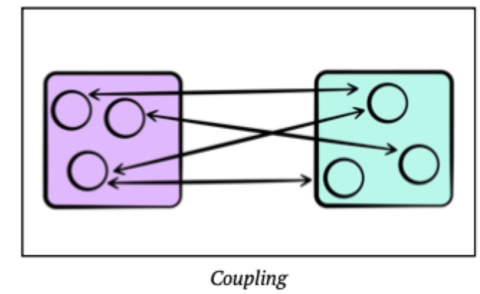
High Cohesion: parts of a class/module should be self-contained.

These work together:

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

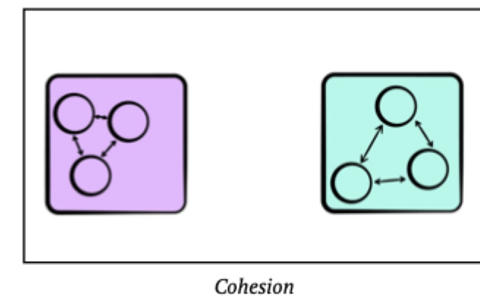
Examples of coupling:

- **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

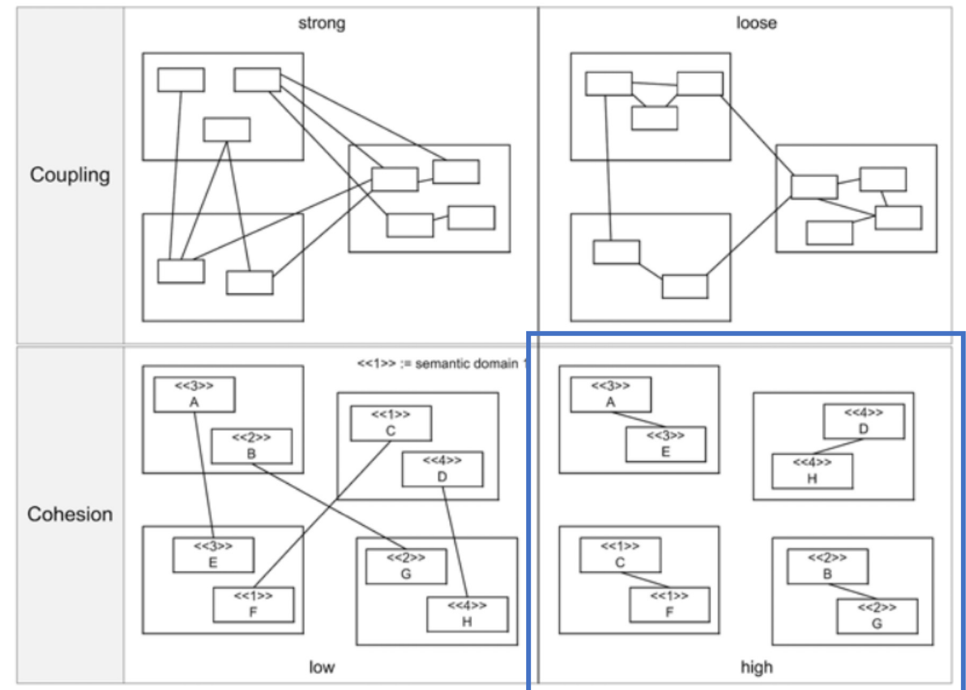
Cohesion is a measure of how closely related the parts of a module are.

You can identify the level of coupling between modules or classes by looking at the methods calls that are made.

High number of calls within a class? **High cohesion**.

High number of calls between classes? **High coupling**.

We generally want to be on the 'lower-right-hand corner' of this diagram.

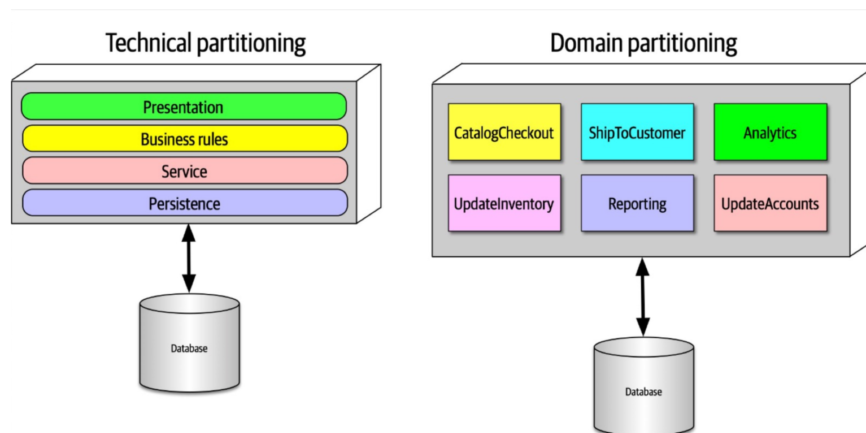


- Oliver Vogel et al. 2011. **Software Architecture**. Springer.

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.

- **Technical partitioning**: group according to technical capabilities.
- **Domain partitioning**: group according to the area of interest.



Applications often leverage technical partitioning. Why?

Mark Richards & Neal Ford. 2020. Fundamentals of Software Architecture. O'Reilly.

Architectural styles

Common patterns that we can leverage.

What is a style?

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

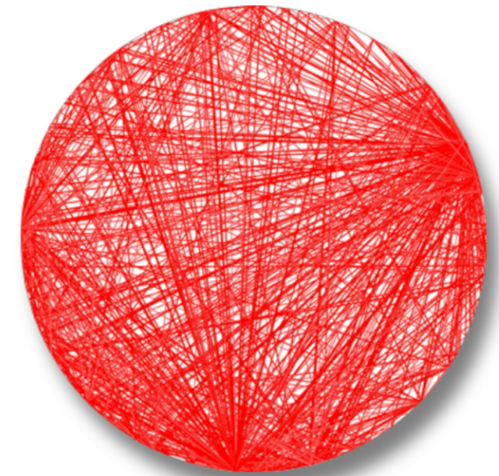
- Each style describes an example of **modularity** + **class relations**.
- Like design patterns, an architectural style is a general solution that has been found to work well at solving specific types of problems.
- An architectural style has a unique **topology** (organization of components) and **characteristics**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



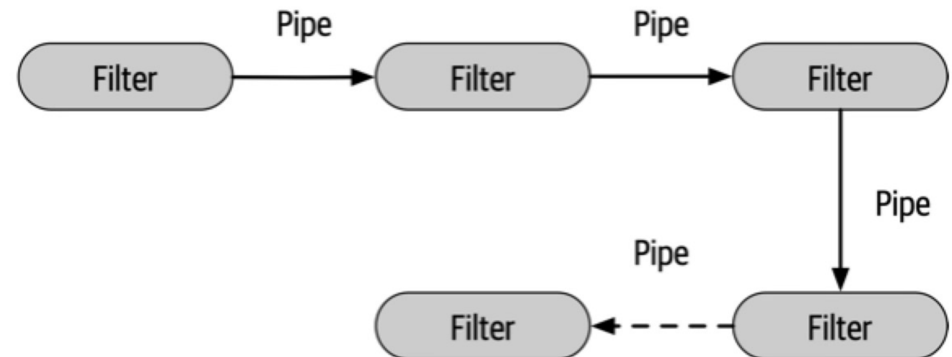
A Big Ball of Mud isn’t intentional—it’s the result of a system being *tightly coupled*, where any module can reference any other module. A system like this is *extremely* difficult to extend or modify.

Console: Pipeline Architecture

A pipeline architecture transforms data in a sequential manner. It consists of pipes and filters. There is usually one outbound starting point (source) and one or more inbound termination point (sink).

- **Pipes** form the communication channel between filters. Each pipe is 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.

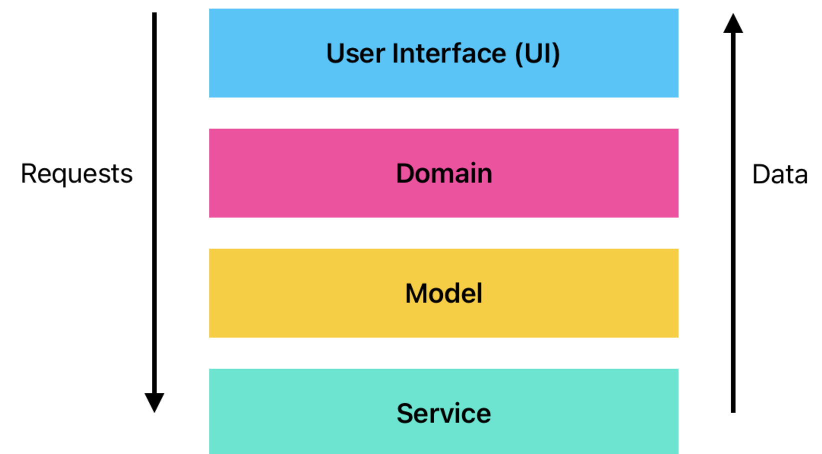
  - Easy to extend by adding nodes.
  - Filters are stateless, and testable.
  - Broadly applicable to any sequential operations.



Applications: Layered Architecture

A **layered** or *n-tier* architecture is a very common architectural style that 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 up the stack).
- *Dependencies extend down*: lower-levels provide functionality that is consumed by higher-levels.

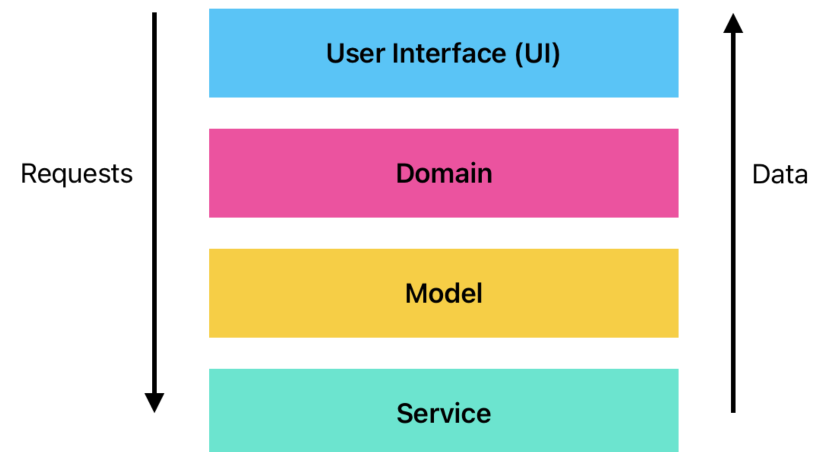


Layers remain isolated ("separation of concerns")
High testability: components in specific layers.
High ease of development.

Applications: Layered Architecture

Standard layers include:

- **User interface** (presentation layer). Handles input/output.
- **Domain**. Business logic; custom classes.
- **Model**. The underlying data that is stored/manipulated.
- **Service**. Optional layer that can communicate out to external systems.



Clear separation of concerns.

Each layer is independent; we use **dependency injection** to decouple layers.

Layers remain isolated ("separation of concerns")
Each layer has a clear, single responsibility.
High testability: components in specific layers.
High ease of development.

Break here and resume next class.

Graphical applications

Let's apply an architectural style to complex applications.

GUI requirements

What do we need our architecture to do?

1. Handle **graphical interaction**: windows, widgets, mouse-cursor.
2. Handle an **interaction cycle**: processing user input cycle.
3. Handle input (events) from **multiple data sources**.

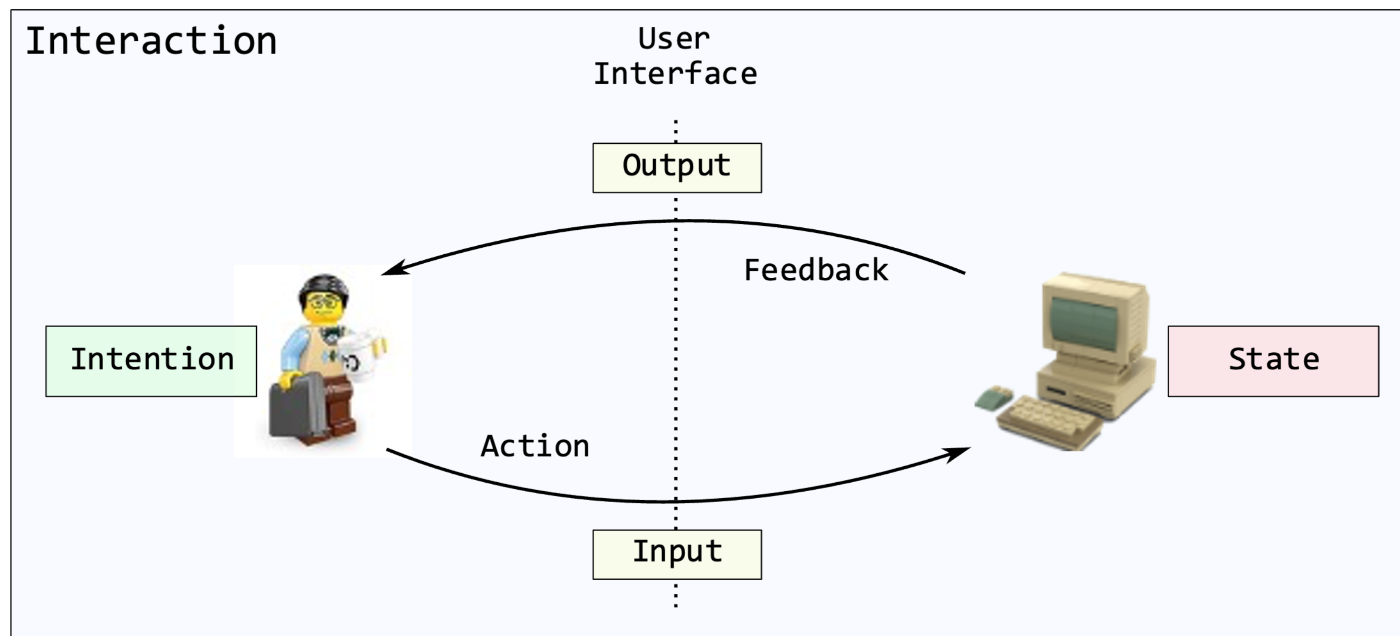
How would a simple layered architecture work here?

1. Graphical interaction

- We need to interact with the OS for rich, graphical capabilities.
  - Windowing system: create, move, resize windows.
  - Application management: memory management, forward mouse events.
  - Data manipulation: shared clipboard, drag-drop data between windows.
- Data should be shared between components (concurrency)
  - Data can show up in multiple places e.g., table data also in a graph.
- We will almost-never build this ourselves!
  - We want to use the underlying OS capabilities as much as possible.
  - We will (eventually) use a GUI toolkit to make this accessible.
- This must be isolated from the rest of the application (layered!)

2. Interactive applications

Practically all applications are interactive. Fundamentally, they revolve around an interaction cycle between the user and the system.



2. Interactive applications

We typically model this as an event-driven architecture.

- `Events` (messages) are triggered when the user interacts with the application.
- They are forwarded to any interested software components, who can then react to the messages.
- `Events` can also be generated (and consumed) by the operating system.

3. Multiple data sources

How do you handle data from multiple sources? Examples:

- The operating system may send events to your application that are not triggered by a user action. e.g., a timer ticking to indicate that time has elapsed, or a notification being sent to your application.
- Your application might request remote data which arrives after some delay. e.g., scrolling through a list of images that are on a remote site, while the list continues to populate in the background.
- Interruptions to your applications workflow based on some other high priority event. e.g., receiving a phone call while watching a video.

GUI patterns

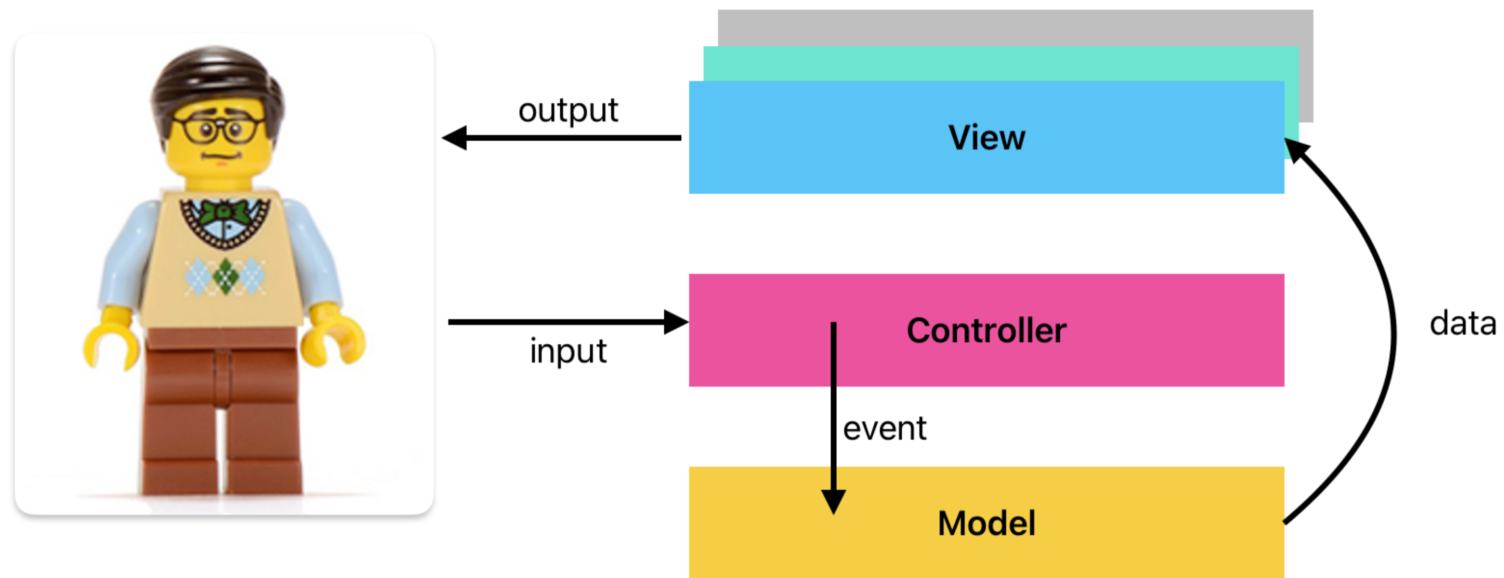
Variations on a theme, designed for our *unique set of problems*.

Model View Controller (MVC)

Model-View-Controller (MVC) was created by Trygve Reenskaug for Smalltalk-79 in the late 1970s as a method of structuring interactive applications. It suggests that an application should consist of the following components:

- **Model**: the information or program state that you are working with,
- **View**: the visual representation of the model, and
- **Controller**: which lays out and coordinates multiple views on-screen, and handles routing user-input.

MVC is specifically designed to model an interaction cycle.



In a “standard MVC” implementation:

- 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**.

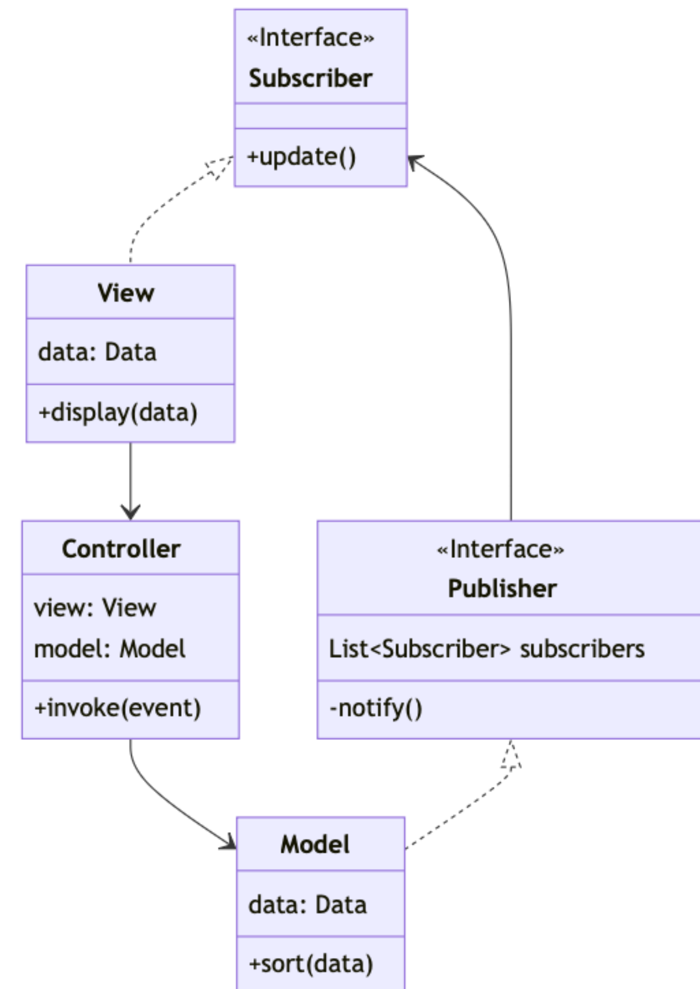
MVC Implementation

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

There are often multiple views. They may each display different data, or views may display the same data.

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 when implementing standard MVC.
- Graphical user interfaces bundle the input and output together into graphical “widgets” on-screen (which we will explore further in the user interfaces section). This makes input and output behaviours difficult to separate, so in-practice, the controller class is rarely implemented.
- Modern applications tend to have multiple screens (either multiple windows open, or multiple screens in -memory that the user switches between). This model does not handle screen coordination terribly well.
- A single monolithic model should usually be split into multiple models, to reflect specialized data needs of each screen.

Model View View-Model (MVVM)

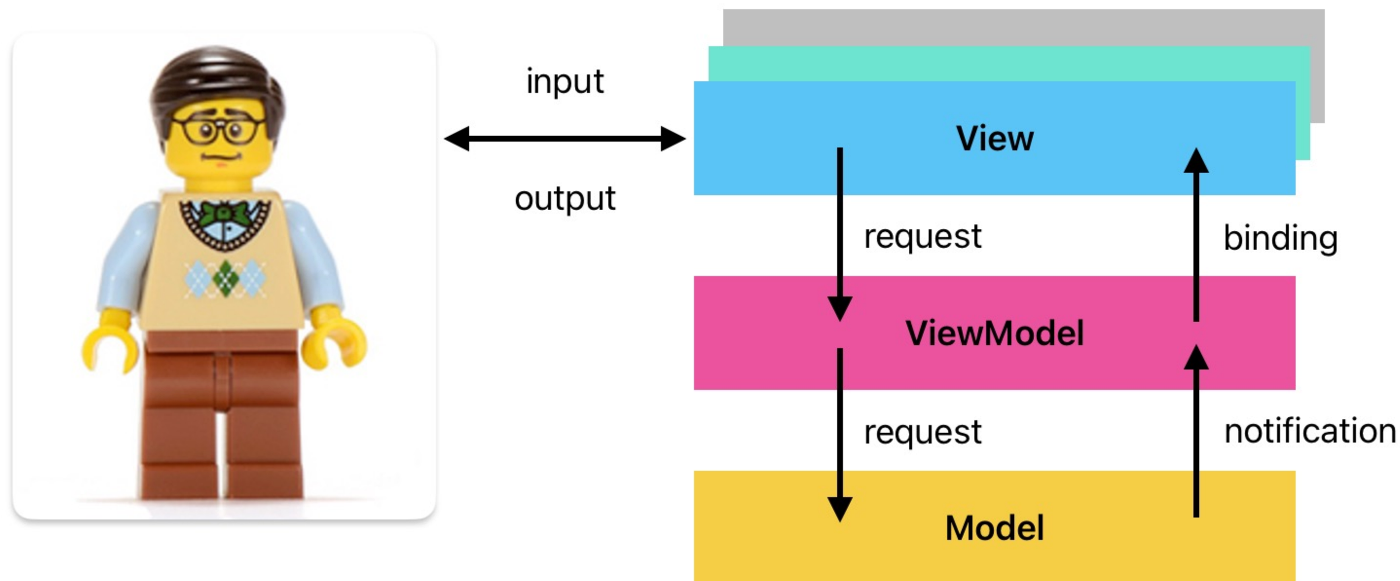
Model-View-ViewModel was invented by Ken Cooper and Ted Peters in 2005. It was intended to simplify event-driven programming and user interfaces in C#/.NET.

MVVM suggests two major changes from MVC:

- MVVM removes the Controller class, and
- 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.



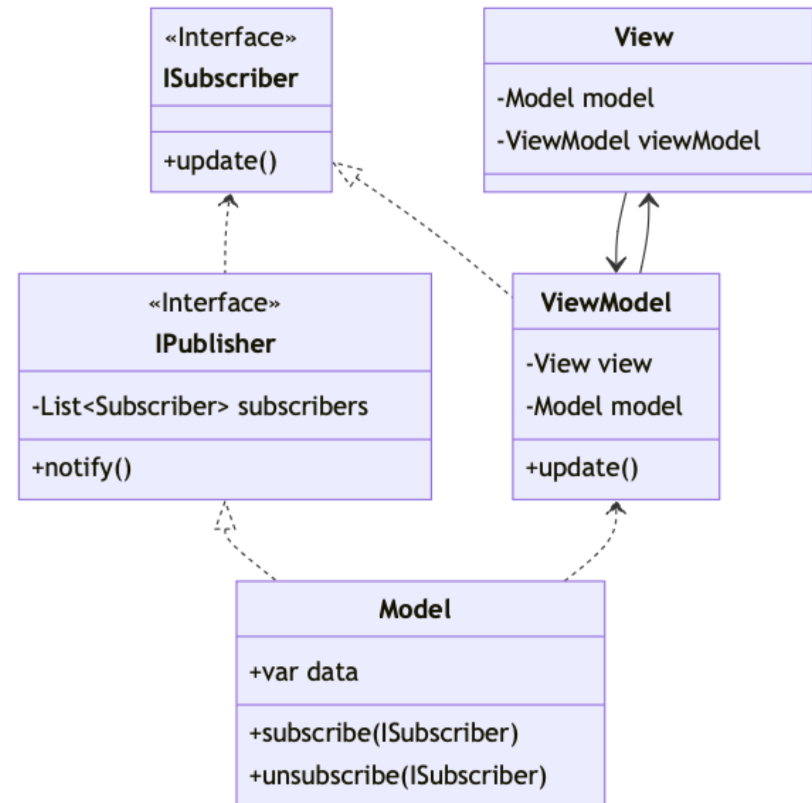
- **Model**: As with MVC, the Model is the Domain object, holding application state.
- **View**: The presentation of data to an output device. Handles both input and output.
- **View-Model**: A component that stores data that is relevant to the View to which it is associated. This may be a subset of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

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.



Layered + MVC

How to pull them together.

Revised: Layered architecture

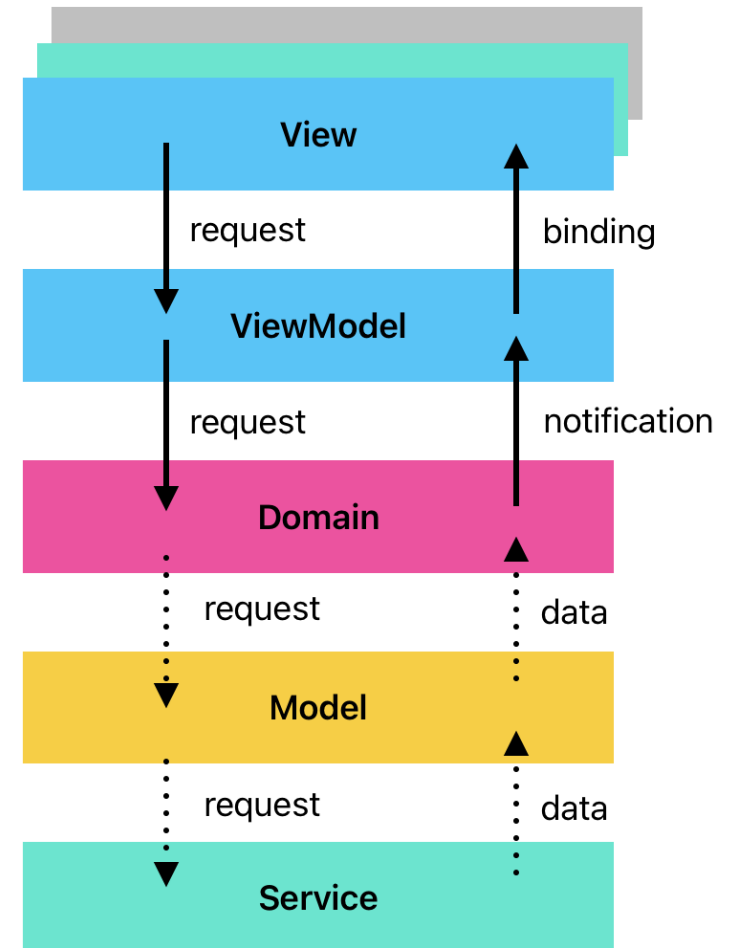
We'll reuse most of the layered architecture from earlier, but include some refinements

UI layer consists of View + VM

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

Same flow as earlier

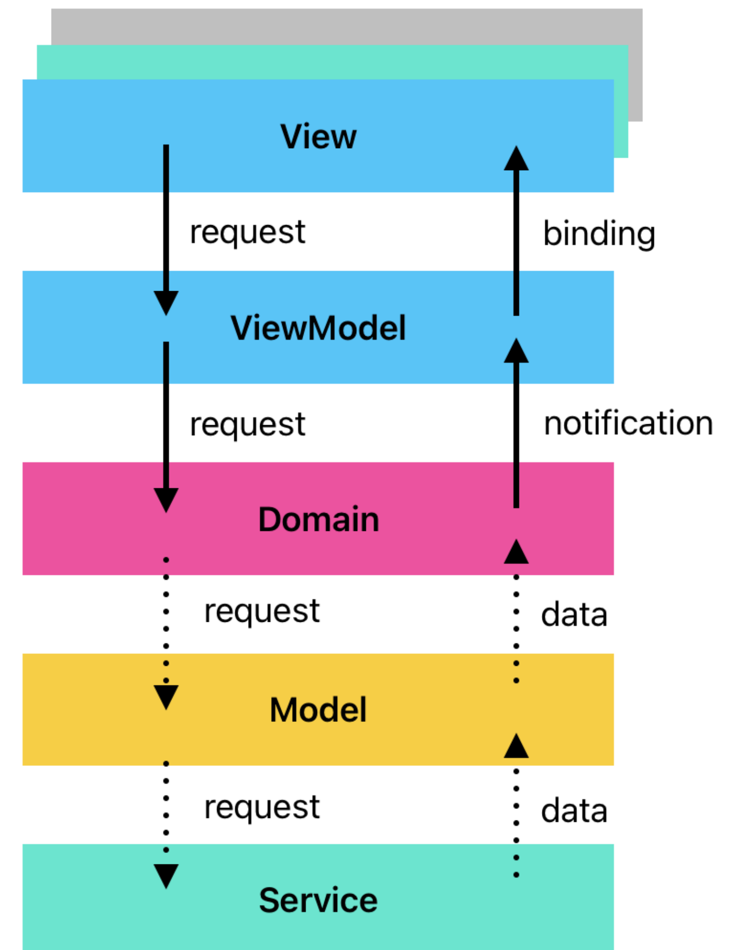
- Requests down, data up.
- Notification used with VM, which in turn propagates data into Views.

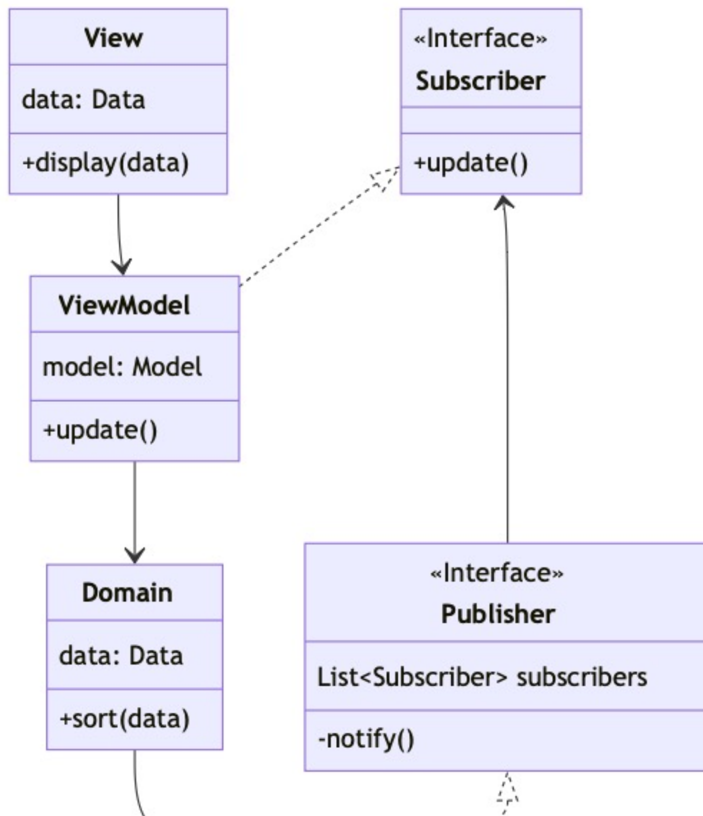


Rules for layered architecture

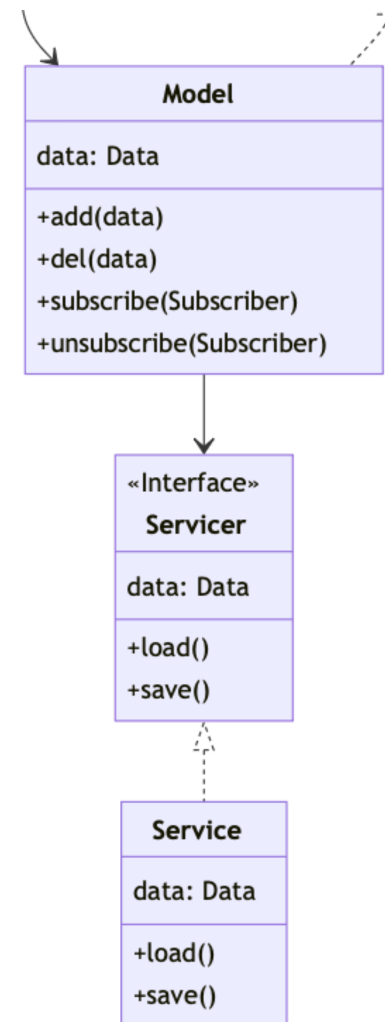
The characteristics of this approach:

- Each layer contains one or more classes with similar functionality.
- Classes can freely communicate within their layer, or make requests "down" the stack.
- Dependencies between layers are top-down. e.g., the UI has a reference to the Domain layer and can pass requests down to it.

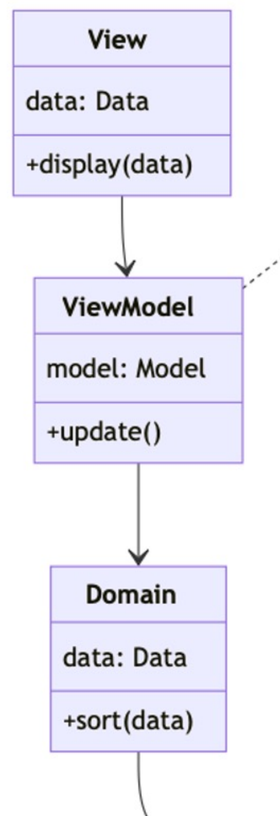




View
data: Data
+display(data)
«Interface»
Subscriber
+update()
ViewModel
model: Model
+update()
Domain
data: Data
+sort(data)
«Interface»
Publisher
List<Subscriber> subscribers
-notify()



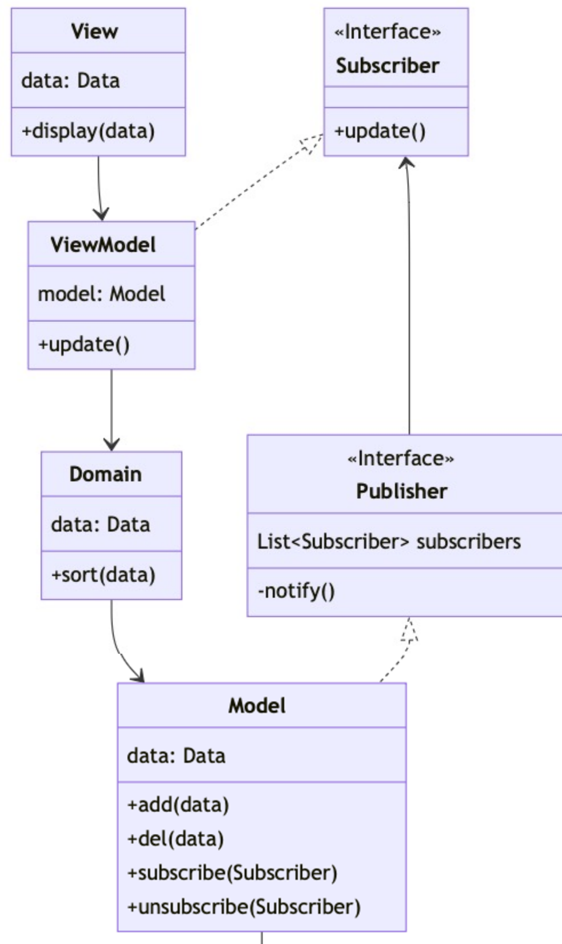
Model
data: Data
+add(data)
+del(data)
+subscribe(Subscriber)
+unsubscribe(Subscriber)
«Interface»
Servicer
data: Data
+load()
+save()
Service
data: Data
+load()
+save()



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).

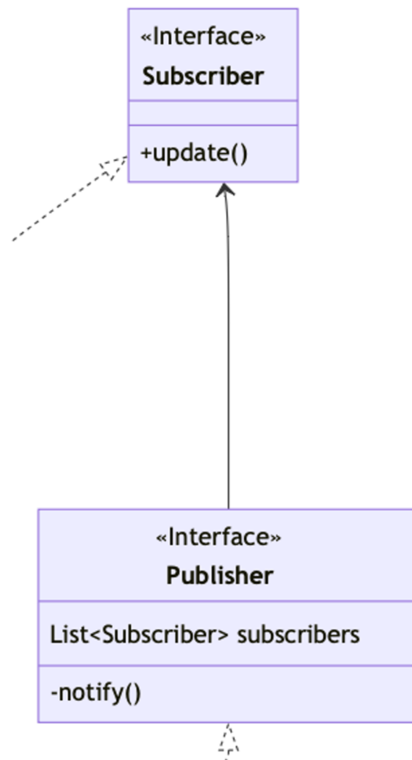


Update rule

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.

```kotlin
fun main {
    val model = Model()
    val viewModel = ViewModel(model)
    val view = View(viewModel) // dependency injection
    model.add(viewModel)
}

// UI classes & interfaces
interface Subscriber {
    fun update()
}

class View(val viewModel: ViewModel) {
    // some user interface class that relies in the viewModel for its state
}

class ViewModel(val model: Model) : Subscriber {
    override fun update() {
    // this method is called by the Publisher (aka Model) when data updates
    }
}
```

```kotlin
abstract class Publisher {
    val views: List<Subscriber> = emptyList()

    fun addView(view: Subscriber) {
        views.add(view)
    }

    fun update() {
        for (view : views) {
            view.update()
        }
    }
}

class Model {
    fun fetchData() {
        // do something that causes data to change
        update() // notify subscribers
    }
}
```

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, or any other external element.
- **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.