Software Architecture

CS 346 Application Development

Credits

- Some content adapted from: Sommerville. 2021. Engineering Software Products: An Introduction to Modern Software Engineering. Pearson. ISBN 978-1292376356.

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 architecture

Architecture is as "fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution". (IEEE 1471)

- Architectural design includes consideration of:
  - A system's overall design and organization,
  - How the software is decomposed into components,
  - How components communicate and work together,
  - The technologies that you use to build the software.
- Many of your system's characteristics are tied to architectural decisions
  - e.g., performance, reliability, scalability.

Software qualities

What are the desirable characteristics of your software system?

Software qualities

The architecture of a system affects non-functional properties or qualities:

- **Responsiveness**: 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 and users' data from unauthorized attacks and intrusions?
- **Usability**: Can system users easily and quickly access the features that they need?
- **Maintainability**: Can the system be readily updated and new features added without undue costs?
- **Resilience**: Can the system continue to deliver services in the event of a failure?

Example: impact on system security

In the Star Wars prequel Rogue One, the evil Empire have stored the plans for their equipment in a single, highly secure, well-guarded, remote location. This is called a **centralized security architecture**. It is based on the principle that if you maintain all of your information in one place, then you can apply lots of resources to protect that information.

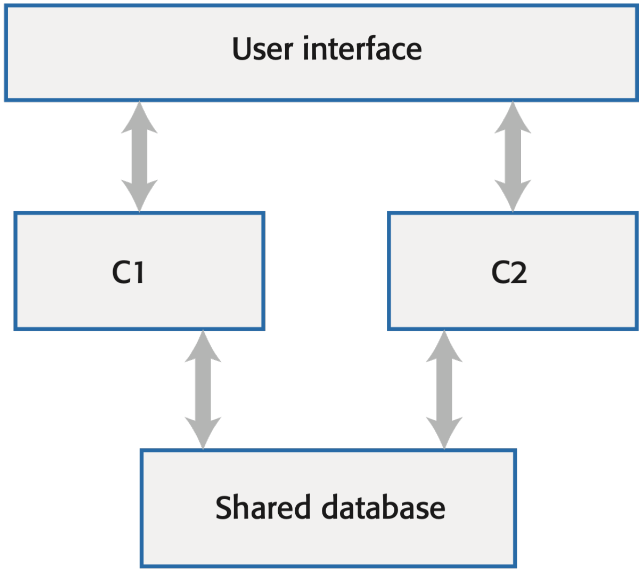
Unfortunately (for the Empire), the rebels managed to breach their security and stole the plans for the Death Star. Had the Empire chosen a **distributed security architecture**, with different parts of the Death Star plans stored in different locations, then stealing the plans would have been more difficult.

Example: impact on system security

- The benefits of a centralized security architecture are that it is easier to design and build protection and that the protected information can be accessed more efficiently.
- However, if your security is breached, you lose everything.
- If you distribute information, it takes longer to access all of the information and costs more to protect it.
- If security is breached in one location, you only lose the information that you have stored there.

Example: maintainability and performance

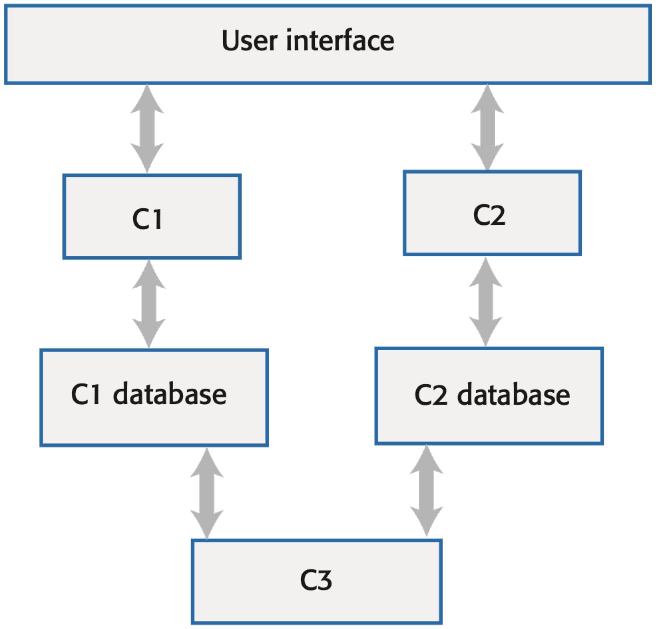
- This example shows a system with two components (C1 and C2) that share a common database.
- Assume C1 runs slowly because it must reorganize the information in the database before using it.
  - The only way to make C1 faster might be to change the database. This means that C2 also must also be changed, which may, potentially, affect its response time.



C1 and C2 share a database.

Example: maintainability and performance

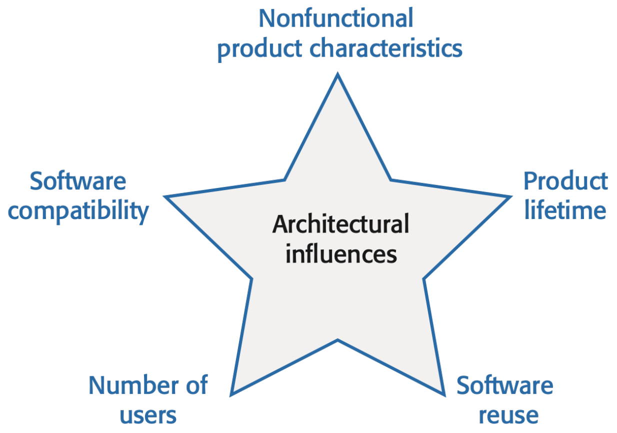
- This diagrams shows a different architecture where each component has its own copy of the parts of the database that it needs.
  - If one component needs to change the database organization, this does not affect the other component.
- However, a multi-database architecture may run more slowly and may 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.



Database reconciliation

Issues that influence architectural decisions

- **Nonfunctional product characteristics** such as security and performance affect all users
- If you anticipate a long **product lifetime**, you will need to create regular product revisions. Your architecture needs to accommodate new features and technology.
- You can save a lot of time and effort, if you can **reuse large components** e.g., open-source software. However, reusing software constrains your architectural choices.
- The number of users of your software can change. This can lead to performance issues unless you design your architecture so that your system can be **scaled** up/down.
- For some products, it is important to maintain **compatibility** with other software e.g., integration with a different system.



Factors to consider when making architectural decisions. *Everything is a tradeoff*.

Trade off: maintainability vs performance

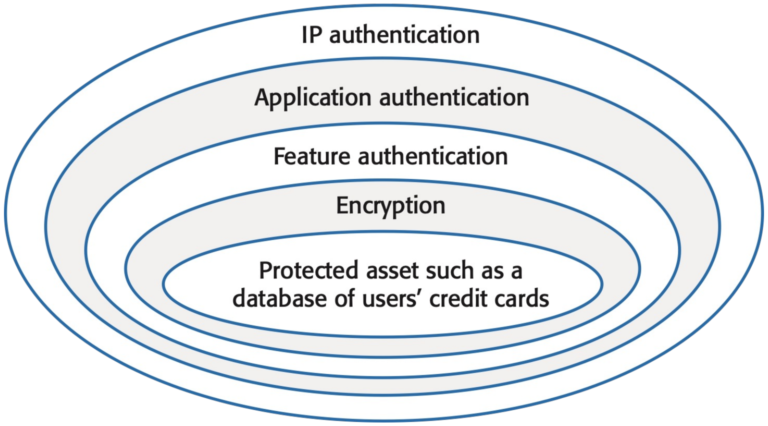
- **System maintainability** is an attribute that reflects how difficult and expensive it is to make changes to a system after it has been released to customers.
  - You improve maintainability by building a system from small self-contained parts, each of which can be replaced or enhanced if changes are required.
- In architectural terms, this means that the system should be decomposed into fine-grain components, each of which does one thing and one thing only.
  - However, it takes time for components to communicate with each other. Consequently, if many components are involved in implementing a product feature, the software will be slower.

Trade off: security vs usability

- You can achieve security by designing the system protection as a series of layers. An attacker has to penetrate all of those layers before the system is compromised.
- Layers might include system authentication layers, a separate critical feature authentication layer, an encryption layer and so on.
- Architecturally, you can implement each of these layers as separate components so that if one of these components is compromised by an attacker, then the other layers remain intact.

Trade off: security vs usability

- You can achieve security by designing the system protection as a series of layers.
- An attacker has to penetrate all those layers before the system is compromised.
- Architecturally, you can implement each of these layers as separate components so that if one of these components is compromised by an attacker, then the other layers remain intact.


IP authentication
Application authentication
Feature authentication
Encryption
Protected asset such as a database of users' credit cards

Layering software protects critical data.

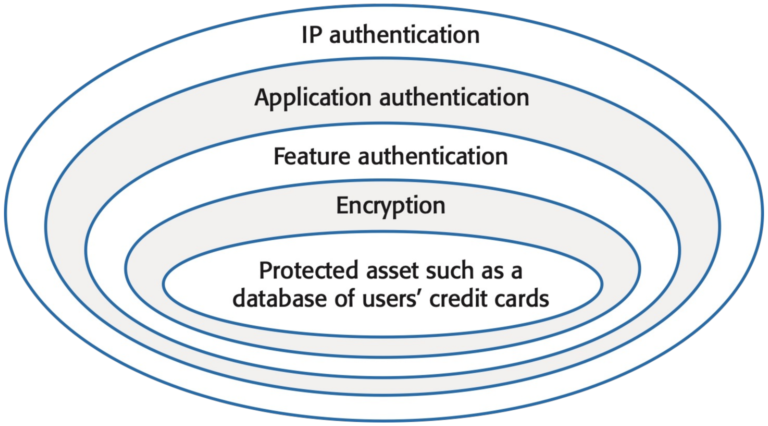
Trade off: security vs usability

However, a layered approach affects usability:

- Users must remember information, like passwords. Interaction with the system is slowed down by its security features.
- Many users find this irritating and often look for work-arounds so that they do not have to re-authenticate.

To avoid this, you need an architecture:

- That doesn't have "too many" security layers (whatever that means),
- That doesn't enforce unnecessary security,
- That provides helper components that reduce the load on users.


IP authentication
Application authentication
Feature authentication
Encryption
Protected asset such as a database of users' credit cards

Layering software protects critical data.

Trade off: availability vs time-to-market

- Availability is particularly important in enterprise products, such as products for the finance industry, where 24/7 operation is expected.
    - The availability of a system is a measure of the amount of 'uptime' of that system.
    - Availability is normally expressed as a percentage of the time that a system is available to deliver user services.
- Architecturally, you achieve availability by having redundant components in a system.
    - e.g., you include sensor components that detect failure and switching components that switch operation to a redundant component when a failure is detected.
- This takes time and increases the cost of system development.
- It also adds complexity to the system and increases the chances of introducing bugs and vulnerabilities.

Modularity

How to structure your system.

Components == Building blocks

A **component** is an element of a software system that implements a coherent set of functionality or features.

- Can be singular or plural (i.e. a common API).
- **High cohesion**: Each component works in isolation.
- **Loose coupling**: Loose dependencies between components.

When designing software architecture:

- First design the component interfaces and relationships.
- Leave the implementation of interfaces to a later stage.

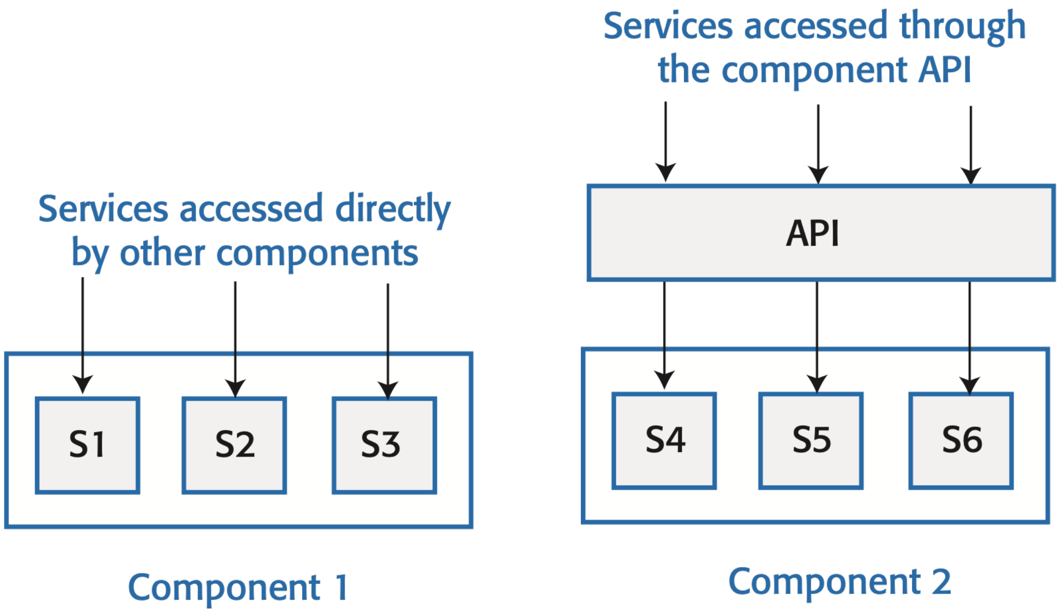
Modularity == Grouping components

Modularity is the logical grouping of components to enforce separation-of-concerns (loose coupling). Programming languages support this in multiple ways e.g., namespaces for C++ or C#.

Kotlin supports modularity in two ways.

- **Modules**: A top-level collection of related components.
  - We generally restrict module use to platform targets, or shared libraries/components.
- **Packages**: A collection of logically related functionality.
  - Packages should contain related classes, functions. e.g., views, models.
  - Use packages to keep components distinct and enforce boundaries.

Services provided by components

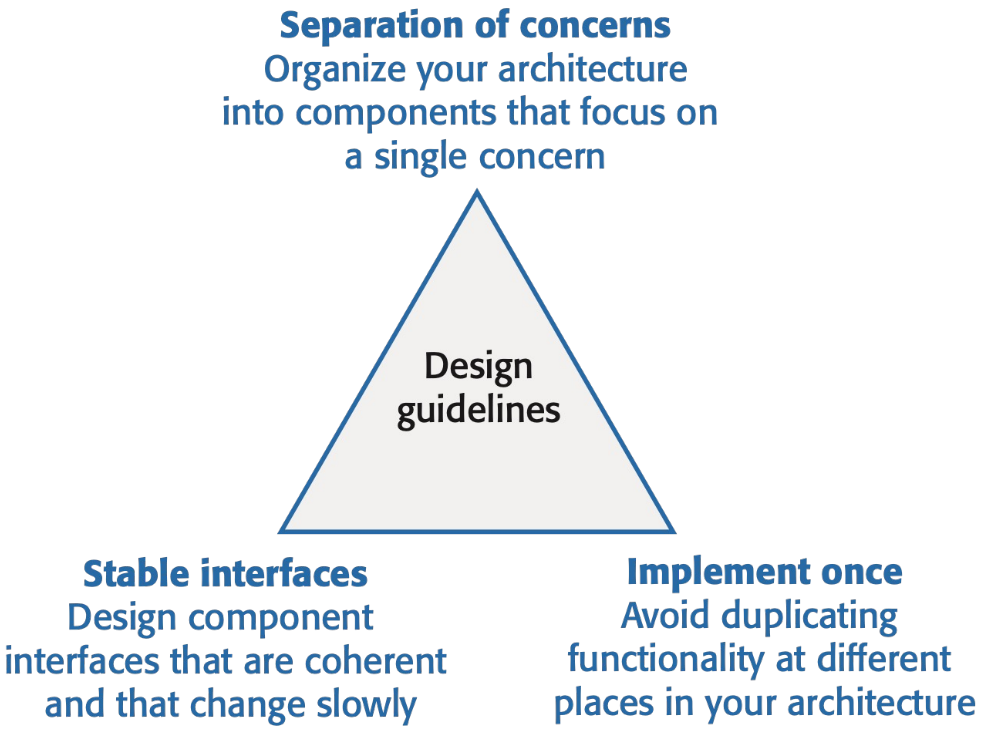

Services accessed directly by other components
Services accessed through the component API
API
S1
S2
S3
S4
S5
S6
Component 1
Component 2

Some components are standalone, but often components work together to provide a service and need a clean and consistent interface.

Architectural complexity

- Complexity in a system architecture arises because of the number and the nature of the relationships between components in that system.
- When decomposing a system into components, you should try to avoid unnecessary software complexity:
    - **Localize relationships**: If there are relationships between components A and B, these are easier to understand if A and B are defined in the same module.
    - **Reduce shared dependencies**: Where components A and B depend on some other component or data, complexity increases because changes to the shared component mean you must understand how these changes affect both A and B.
    - **Use local data**: avoid sharing data between components.

Architectural design guidelines



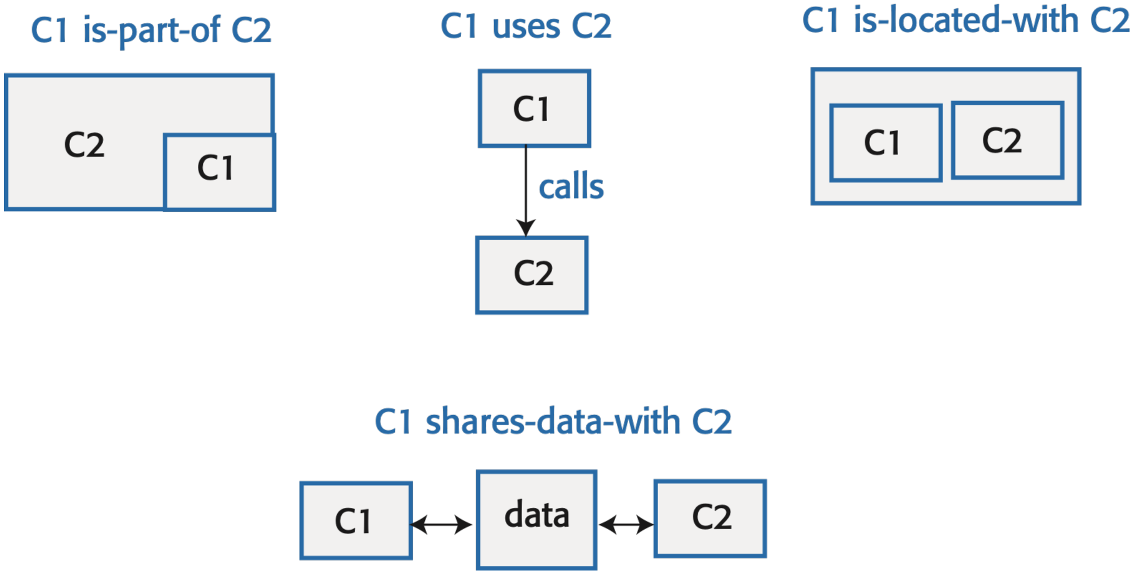
Separation of concerns
Organize your architecture into components that focus on a single concern

Design guidelines

Stable interfaces
Design component interfaces that are coherent and that change slowly

Implement once
Avoid duplicating functionality at different places in your architecture

Examples of component relationships



C1 is-part-of C2

C1 uses C2

C1 is-located-with C2

C1 shares-data-with C2

Architectural styles

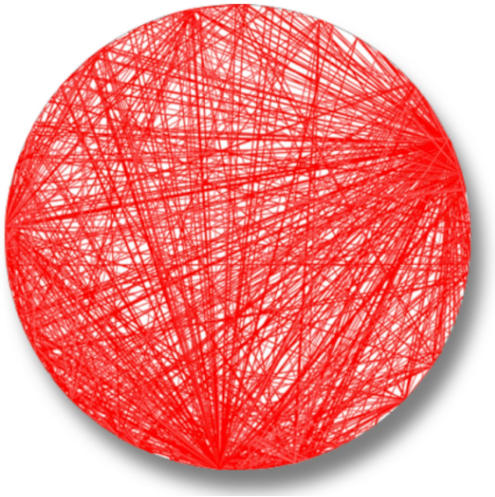
How can we organize components?

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.

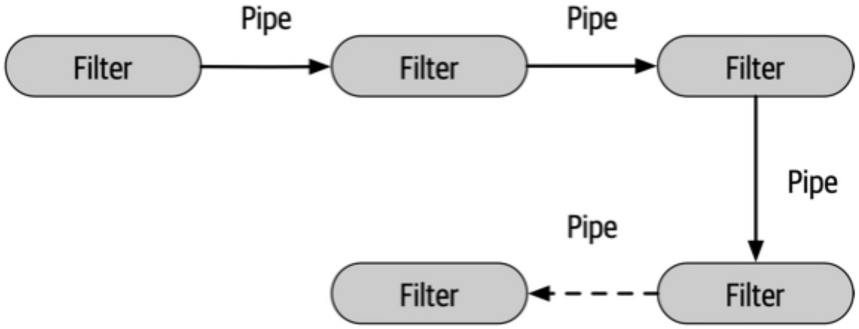
A Big Ball of Mud isn’t intentional—it’s the result of a system being *tightly coupled*, where any module can reference any other module. A system like this is *extremely* difficult to extend or modify.

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.

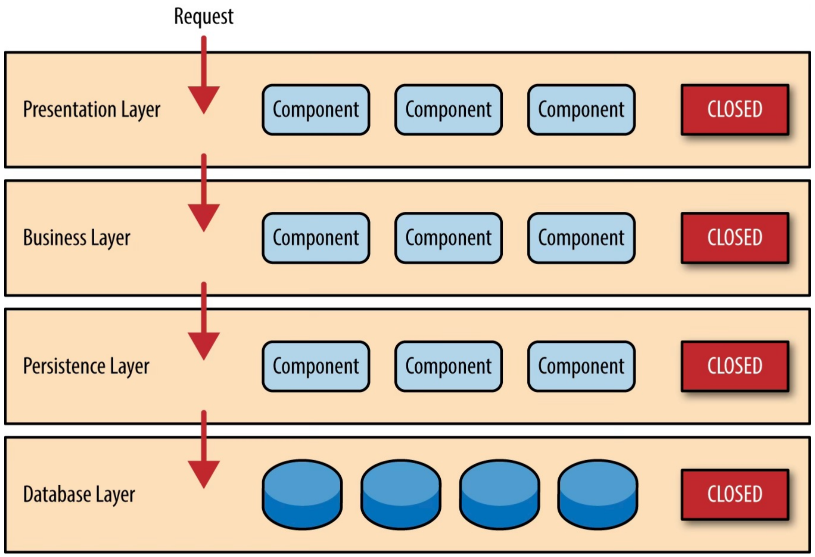


Applications: Layered architecture

A layered architecture is meant to model back-and-forth interaction between a user and an interactive. software into horizontal layers, where each layer represents a ***logical*** division of functionality. system.

Each layer has specific functionality that is presents to the layer above (i.e. lower layers provide services up the stack).

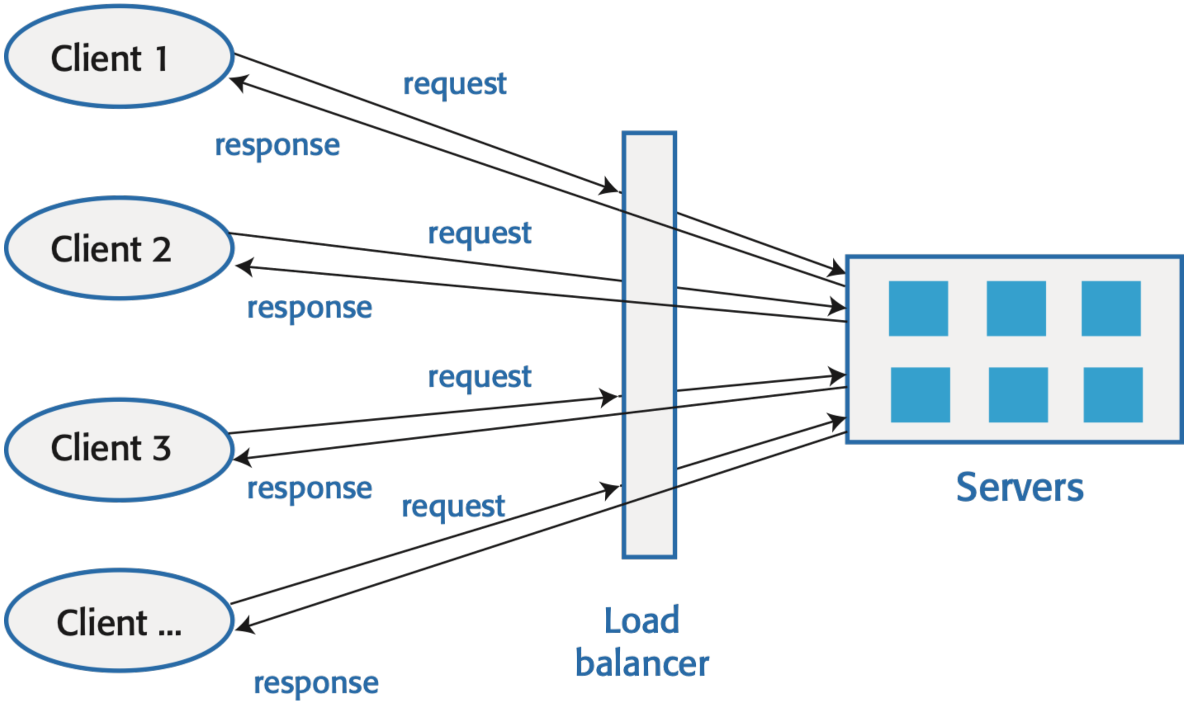
- **Presentation**: UI (input/output).
- **Business Layer**: application logic.
- **Persistence Layer**: describes how to manage and save application data.
- **Database** Layer: the underlying data store that stores data.



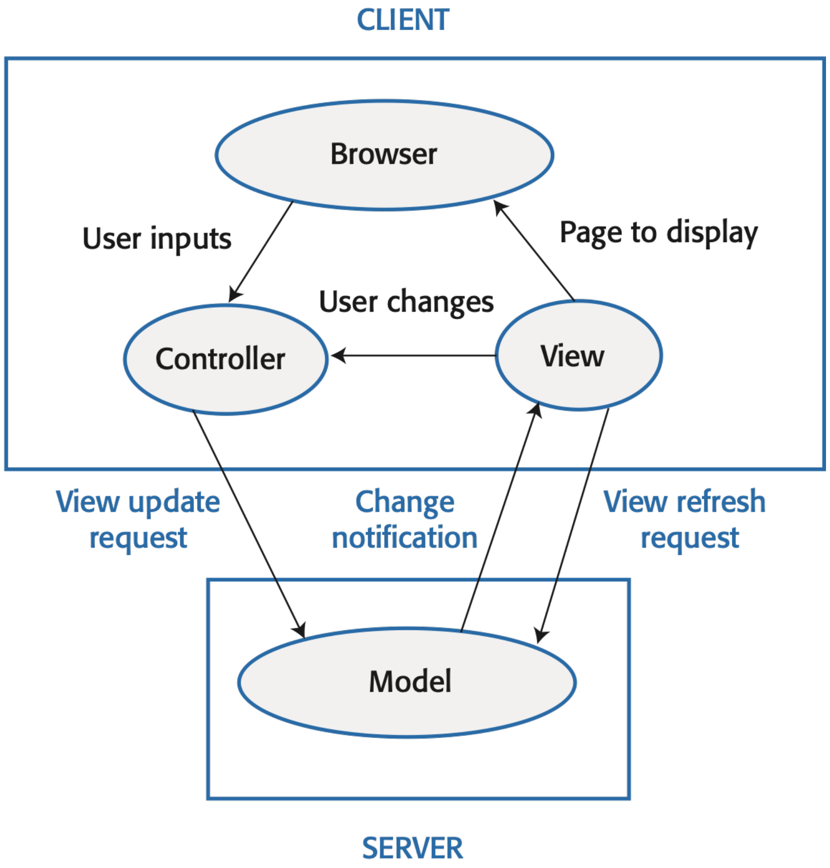
Distribution architecture

- The distribution architecture of a software system defines the servers in the system and the allocation of components to these servers.
- Client-server architectures are a type of distribution architecture that is suited to applications where clients access a shared database and business logic operations on that data.
- In this architecture, the user interface is implemented on the user’s own computer or mobile device.
  - Functionality is distributed between the client and one or more server computers.

Client-server architecture



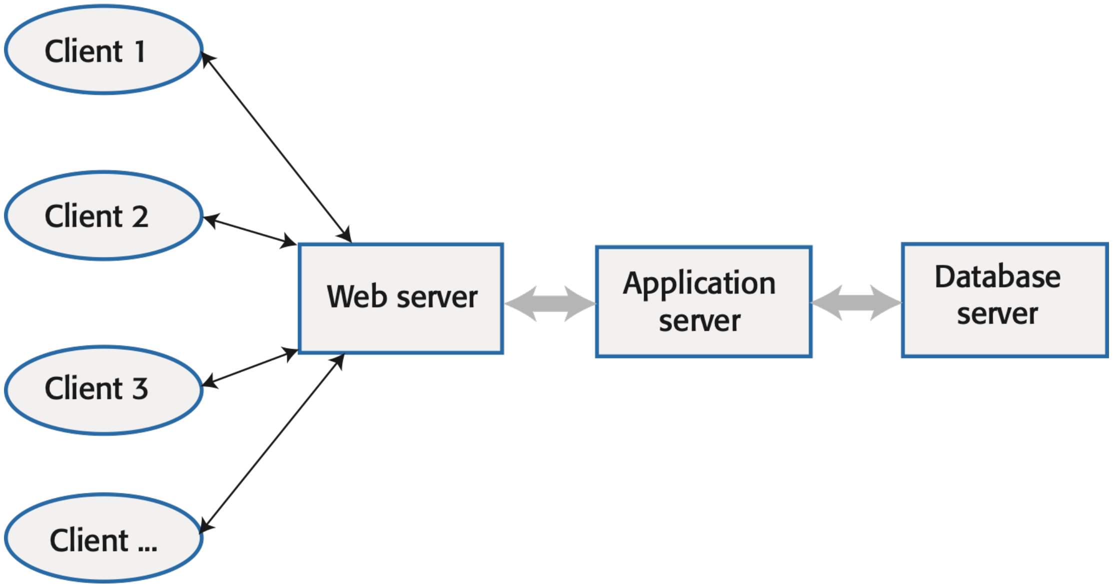
The model-view-controller pattern



Client-server communication

- Client-server communication normally uses the HTTP protocol.
  - The client sends a message to the server that includes an instruction such as GET or POST along with the identifier of a resource (usually a URL) on which that instruction should operate. The message may also include additional information, such as information collected from a form.
- HTTP is a text-only protocol so structured data has to be represented as text. There are two ways of representing this data that are widely used, namely XML and JSON.
  - XML is a markup language with tags used to identify each data item.
  - JSON is a simpler representation based on the representation of objects in the Javascript language.

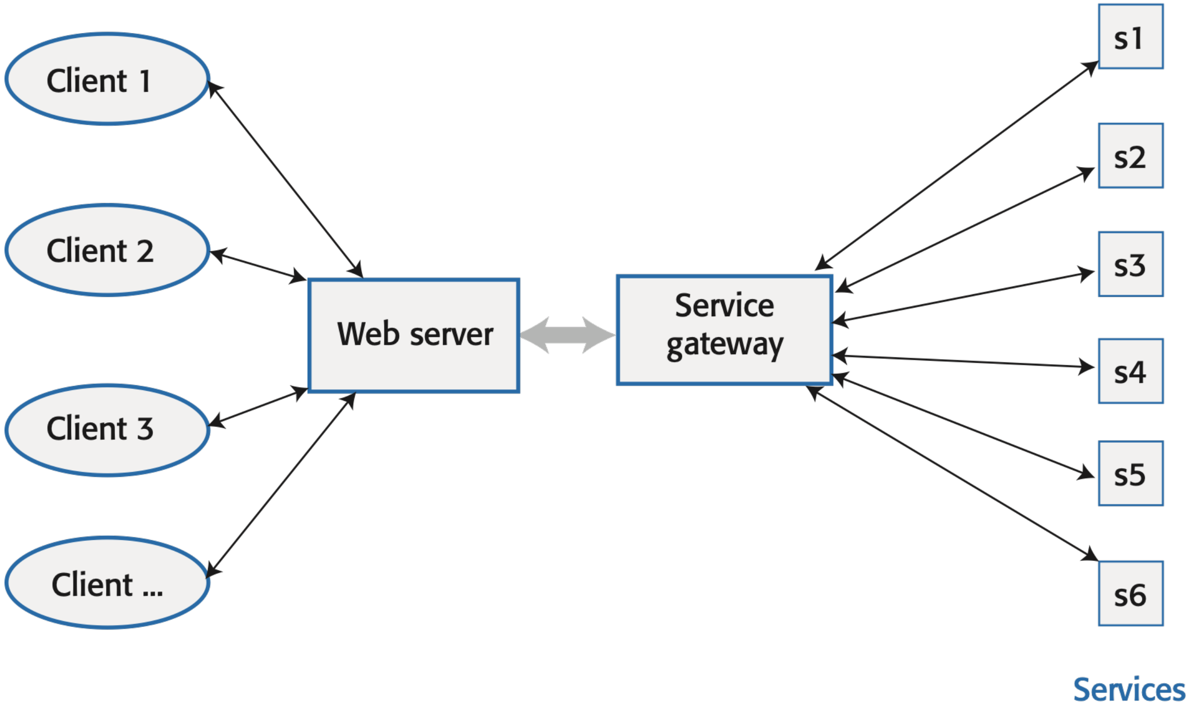
Multi-tier client-server architecture



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.
- Many servers may be involved in providing services
- A service-oriented architecture is usually easier to scale as demand increases and is resilient to failure.

Service-oriented architecture



Issues in architectural choice

- Data type and data updates
  - If you are mostly using structured data that may be updated by different system features, it is usually best to have a single shared database that provides locking and transaction management. If data is distributed across services, you need a way to keep it consistent and this adds overhead to your system.
- Change frequency
  - If you anticipate that system components will be regularly changed or replaced, then isolating these components as separate services simplifies those changes.
- The system execution platform
  - If you plan to run your system on the cloud with users accessing it over the Internet, it is usually best to implement it as a service-oriented architecture because scaling the system is simpler.
  - If your product is a business system that runs on local servers, a multi-tier architecture may be more appropriate.

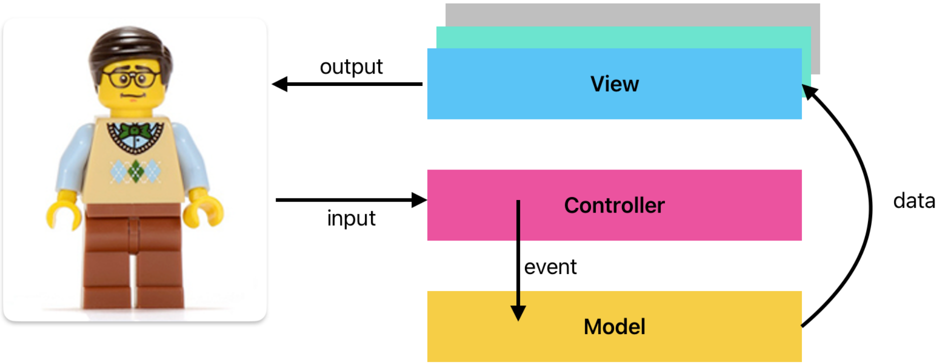
Server

- A key decision that you have to make is whether to design your system to run on customer servers or to run on the cloud.
- For consumer products that are not simply mobile apps I think it almost always makes sense to develop for the cloud.
- For business products, it is a more difficult decision.
  - Some businesses are concerned about cloud security and prefer to run their systems on in-house servers. They may have a predictable pattern of system usage so there is less need to design your system to cope with large changes in demand.
- An important choice you have to make if you are running your software on the cloud is which cloud provider to use.

MVC + MVVM

A “standard” application architecture (as much as there can be one).

Model-View Controller



MVC originated with Smalltalk (1988).

It’s an attempt to build a generic architectural model for interactive applications.

- 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 Implementation

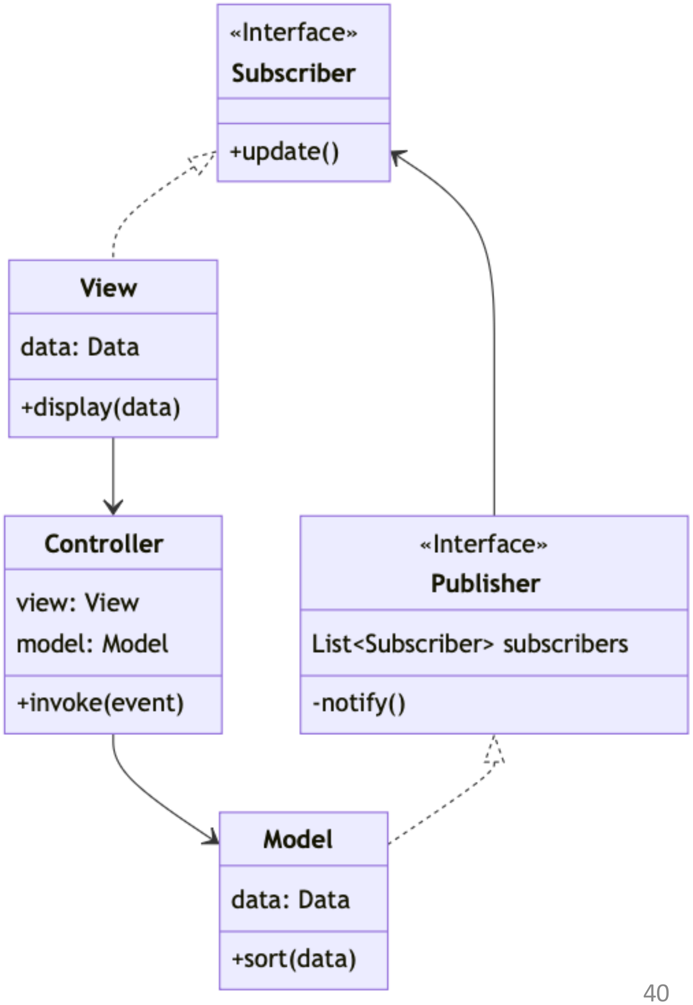
Components

- **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?

Layered Architecture

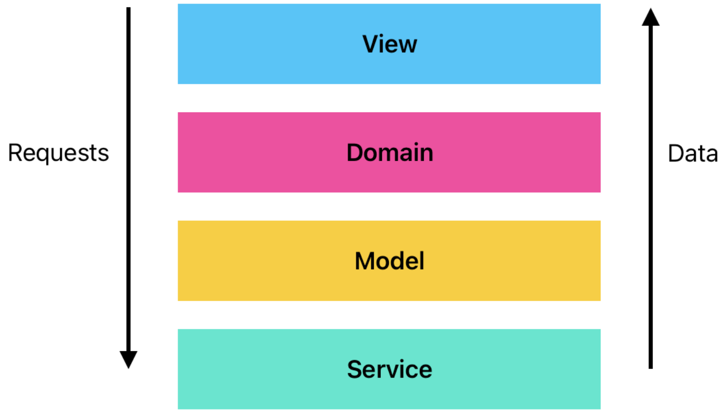
Let’s revisit the layered architecture from earlier and see if we can adapt it as a “fix” for MVC.

Reminder

- 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.



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

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.

MVVM

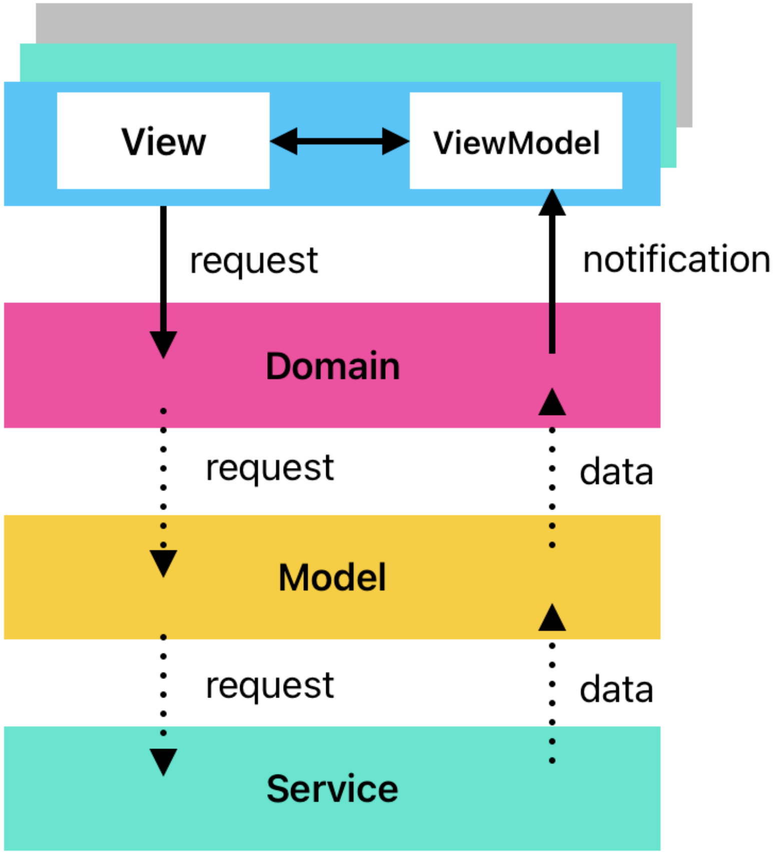
Refinement from our layered architecture:

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.

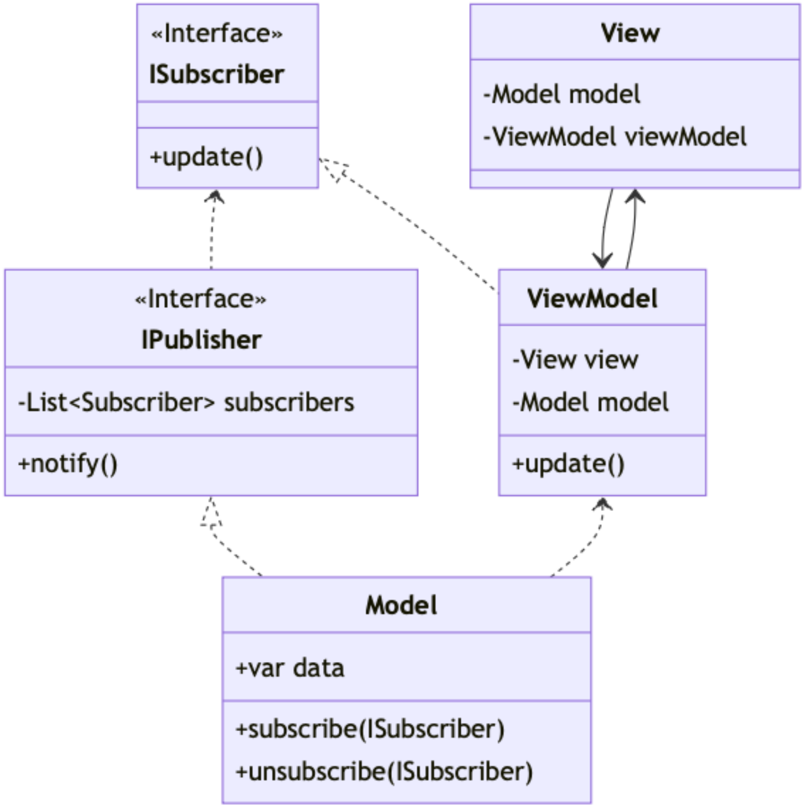


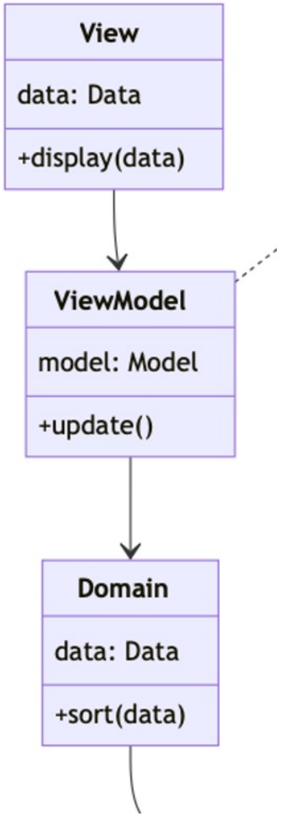
MVVM Implementation

- **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.

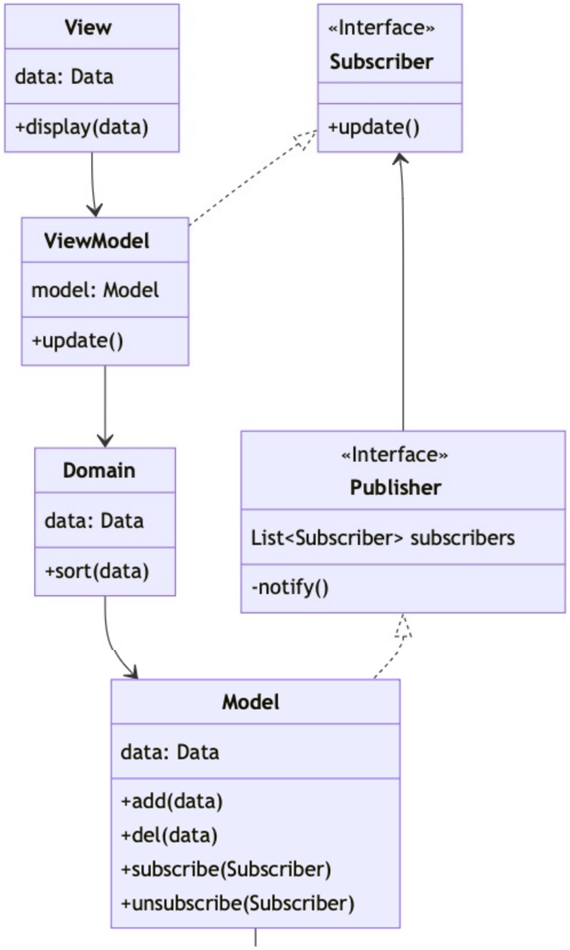




Dependency rule

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).



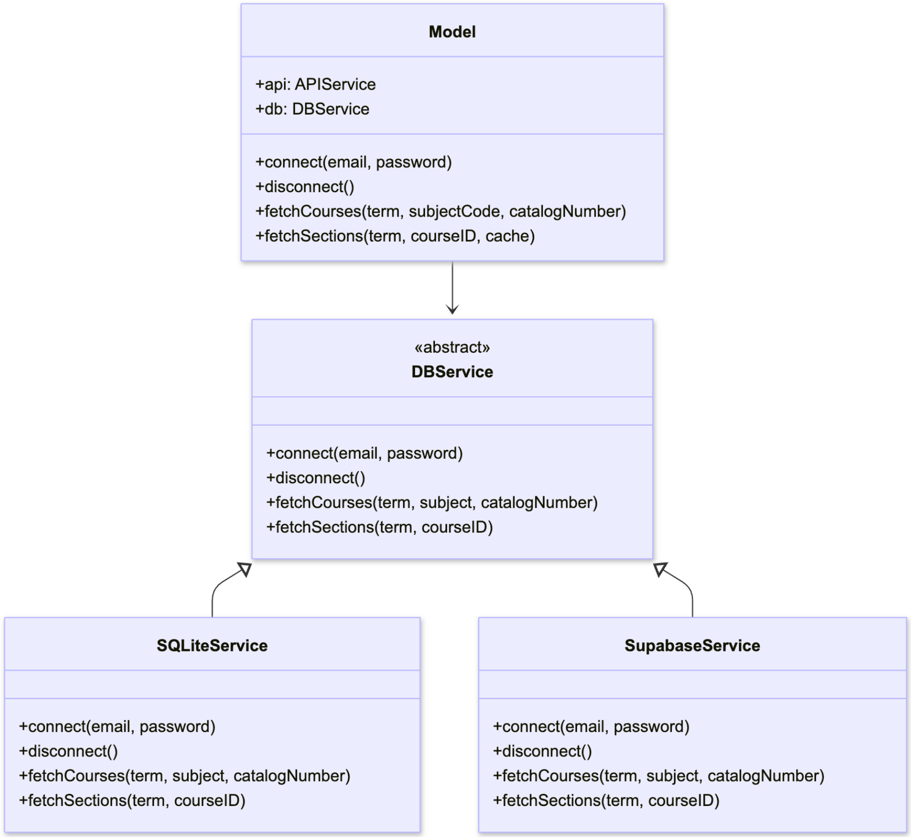
Update rule

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".

Abstractions not concretions



Model

+api: APIService
+db: DBService

+connect(email, password)
+disconnect()
+fetchCourses(term, subjectCode, catalogNumber)
+fetchSections(term, courseID, cache)

«abstract»
DBService

+connect(email, password)
+disconnect()
+fetchCourses(term, subject, catalogNumber)
+fetchSections(term, courseID)

SQLiteService

+connect(email, password)
+disconnect()
+fetchCourses(term, subject, catalogNumber)
+fetchSections(term, courseID)

SupabaseService

+connect(email, password)
+disconnect()
+fetchCourses(term, subject, catalogNumber)
+fetchSections(term, courseID)

Represent your components as interfaces, and implement based on those interfaces. This is critical for testing later!

e.g., our Database is an implementation of a generic database interface. This lets us swap in a different database for testing.

We'll discuss more when we discuss unit testing and dependency injection.

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.

Summary 1

- Software architecture is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.
- The architecture of a software system has a significant influence on non-functional system properties such as reliability, efficiency and security.
- Architectural design involves understanding the issues that are critical for your product and creating system descriptions that shows components and their relationships.
- The principal role of architectural descriptions is to provide a basis for the development team to discuss the system organization. Informal architectural diagrams are effective in architectural description because they are fast and easy to draw and share.
- System decomposition involves analyzing architectural components and representing them as a set of finer-grain components.

Summary 2

- To minimize complexity, you should separate concerns, avoid functional duplication and focus on component interfaces.
- Applications often have a common layered structure including user interface layers, application-specific layers and a database layer.
- The distribution architecture in a system defines the organization of the servers in that system and the allocation of components to these servers.
- Multi-tier client-server and service-oriented architectures are the most commonly used architectures for web-based systems.

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.