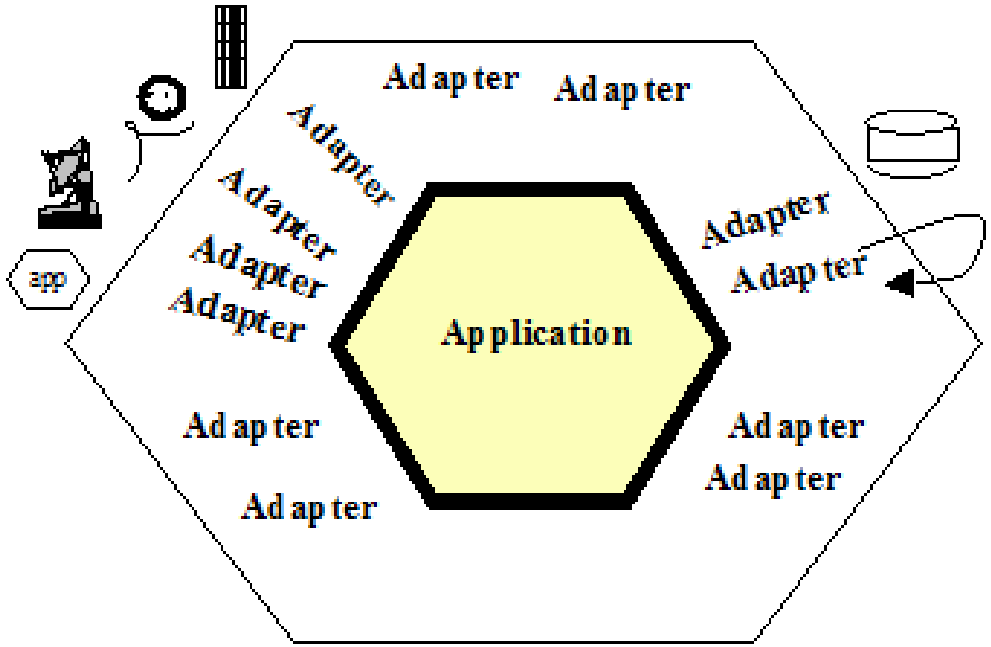
Software Architectures and Patterns

Hexagonal

CQRS pattern

Hexagonal Architecture

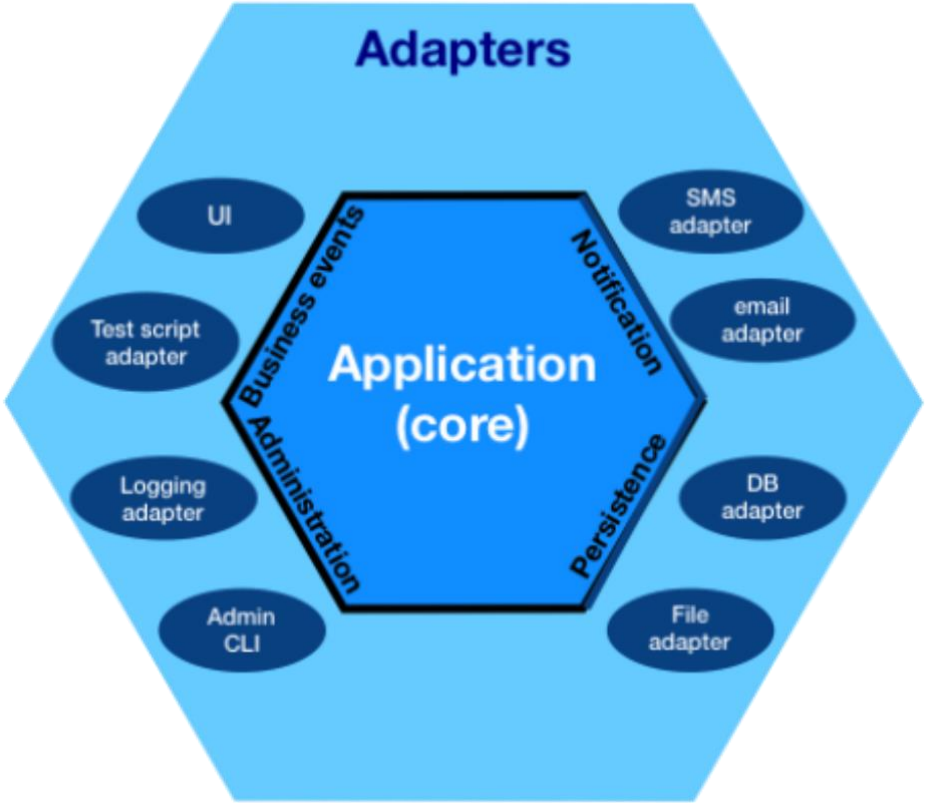
Alternative name: ”Ports & Adapters”



The hexagonal architecture was invented by Alistair Cockburn in an attempt to avoid known structural pitfalls in object-oriented software design, such as undesired dependencies between layers and contamination of user interface code with business logic, and published in 2005.

Intent

- Allow an application to equally be driven by users, programs, automated test or batch scripts, and to be developed and tested in isolation from its eventual run-time devices and databases.

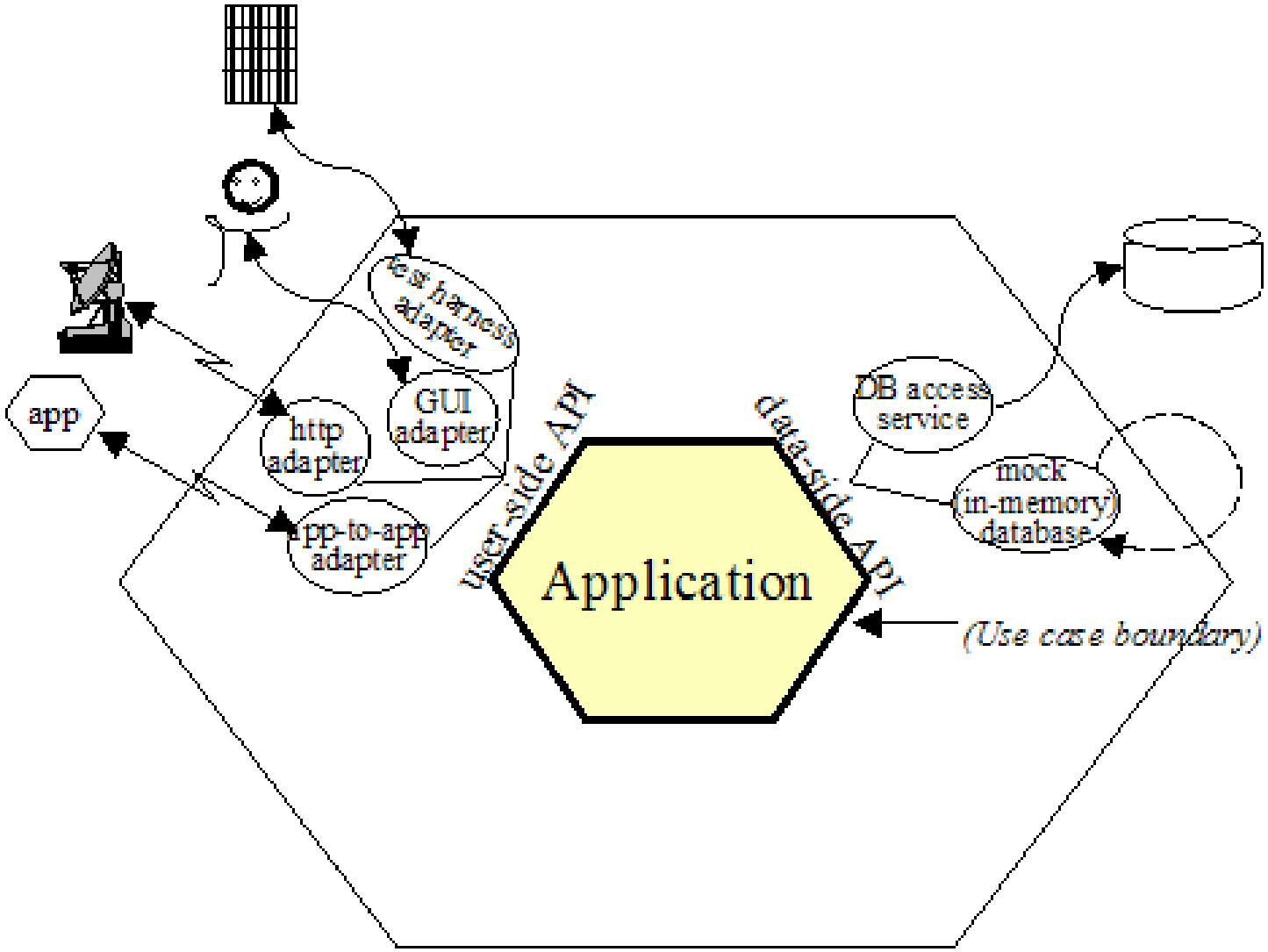


Motivation

One of the great bugaboos of software applications over the years has been infiltration of business logic into the user interface code.
The problem this causes is threefold:

- First, the system can't neatly be tested with automated test suites because part of the logic needing to be tested is dependent on oft-changing visual details such as field size and button placement;
- For the exact same reason, it becomes impossible to shift from a human-driven use of the system to a batch-run system;
- For still the same reason, it becomes difficult or impossible to allow the program to be driven by another program when that becomes attractive.

Structure


test harness adapter
GUI adapter
app
http adapter
app-to-app adapter
user-side API
Application
data-side API
DB access service
mock in-memory database
(Use case boundary)

Ports

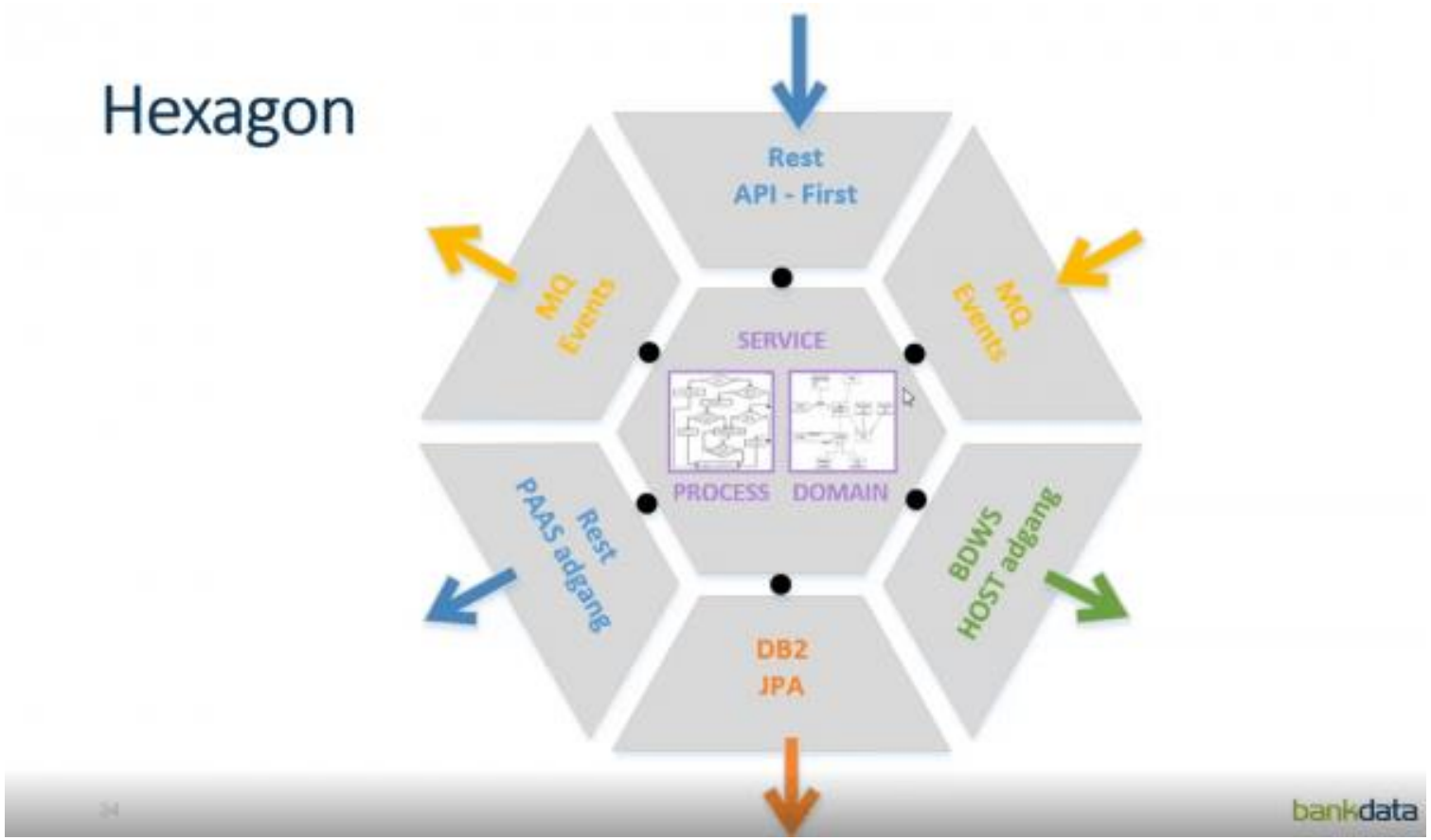
- The application communicates over ‘’ports’’ to external agencies.
  - The word “port” is supposed to evoke thoughts of ‘’ports’’ in an operating system, where any device that adheres to the protocols of a port can be plugged into it; and ‘’ports’’ on electronics gadgets, where again, any device that fits the mechanical and electrical protocols can be plugged in.
- The protocol for a port is given by the *purpose of the conversation* between the two devices.
- The protocol takes the form of an application program interface (API)
- Many applications have only two ports:
  - the user-side dialog
  - the database-side dialog

Adapters

- For each external device there is an ‘’adapter’’ that converts the API definition to the signals needed by that device and vice versa.
- A graphical user interface or GUI is an example of an adapter that maps the movements of a person to the API of the port.
- There will typically be multiple adapters for any one port

Example

Bankdata er begyndt at anvende en hexagonal arkitektur



From https://www.version2.dk/artikel/bankdata-dropper-3-lagsarkitekturen-bruger-mikroservices-her-ret-fedt-skalerer-1091708

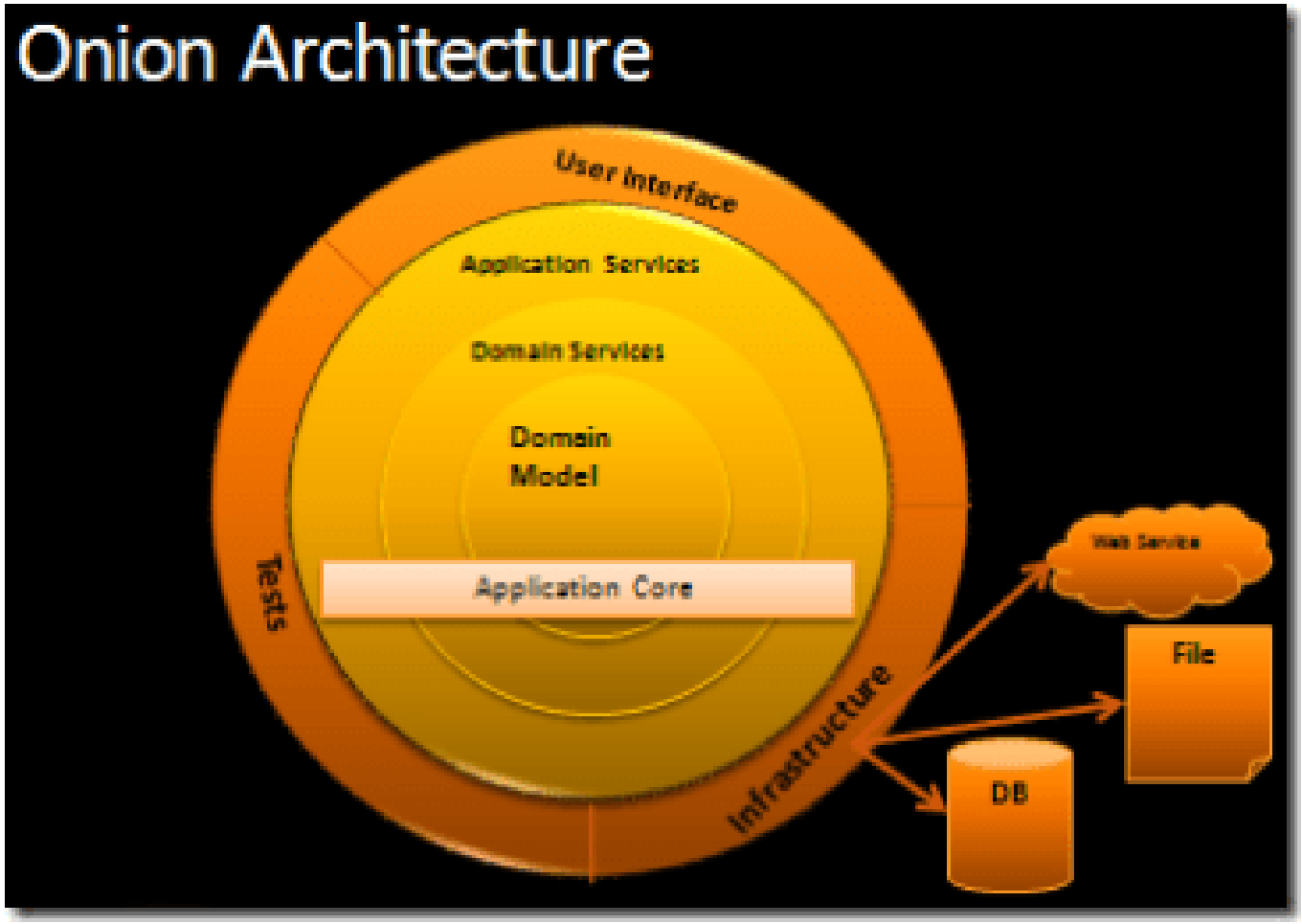
Variants

- The onion architecture proposed by Jeffrey Palermo in 2008 is similar to the hexagonal architecture:
  - it also externalizes the infrastructure with proper interfaces to ensure loose coupling between the application and the database.
  - It decomposes further the application core into several concentric rings using inversion of control.

- The clean architecture proposed by Robert C. Martin in 2012 combines the principles of the hexagonal architecture, the onion architecture and several other variants.
- It provides additional levels of detail of the component, which are presented as concentric rings.
- It isolates adapters and interfaces (user interface, databases, external systems, devices) in the outer rings of the architecture and leaves the inner rings for use cases and entities.
- The clean architecture uses the principle of dependency inversion with the strict rule that dependencies shall only exist between an outer ring to an inner ring and never the contrary.

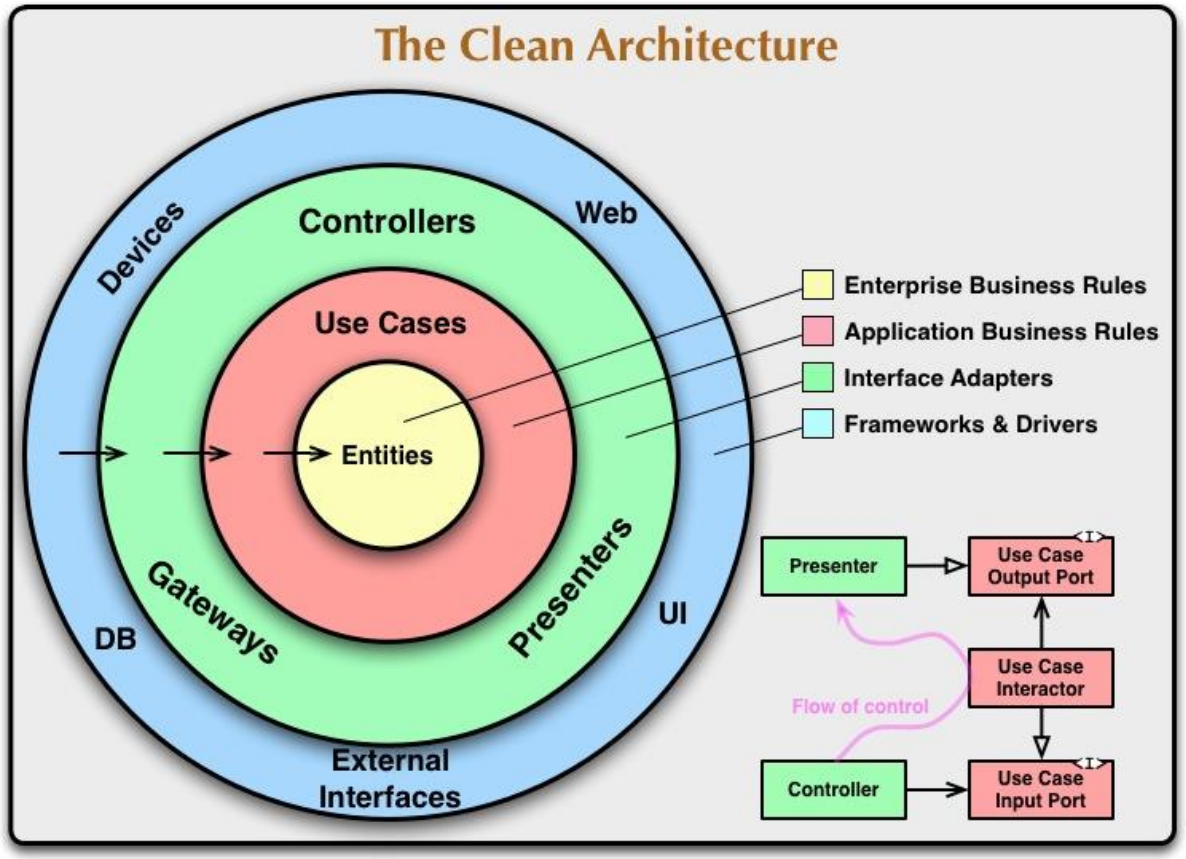
The Onion Architecture

- The fundamental rule is that all code can depend on layers more central, but code cannot depend on layers further out from the core.
- In other words, all coupling is toward the center.



The Clean Architecture

- The concentric circles represent different areas of software. In general, the further in you go, the higher level the software becomes. The outer circles are mechanisms. The inner circles are policies.
- The overriding rule that makes this architecture work is *The Dependency Rule*. This rule says that *source code dependencies* can only point *inwards*. Nothing in an inner circle can know anything at all about something in an outer circle.



Layered Architecture Style

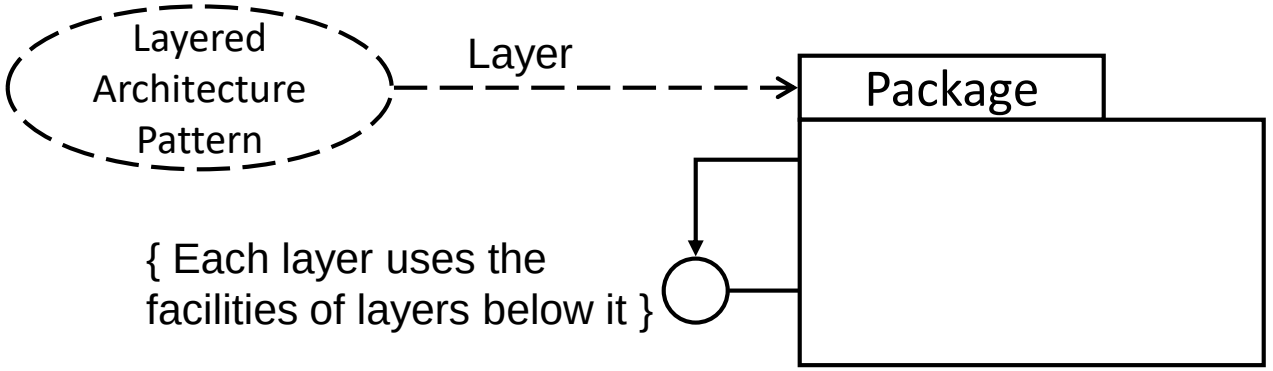
Software Layer

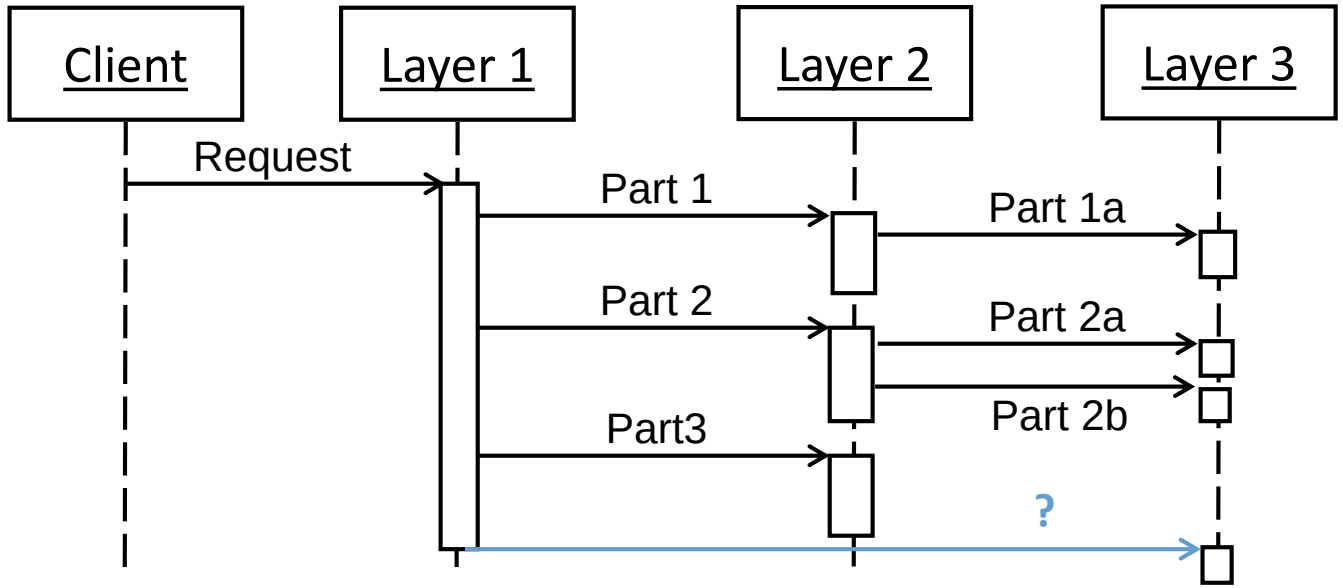
- A layer is a coarse-grained grouping of:
  - classes
  - packages
  - or subsystems
- That has **cohesive responsibility** for a major aspect of the system.

- Organization of Layers
  - **Higher layers call upon services of lower layers!**
  - **Lower layers must not call upon services of higher layers**
    - **But lower layers are allowed to notify higher layers of events by use of the observer pattern or call-back function pointers (events).**

- Layer vs. Tier
  - Layers is a logical separation.
  - Tier represent a physical separation.

Layered Architecture Pattern

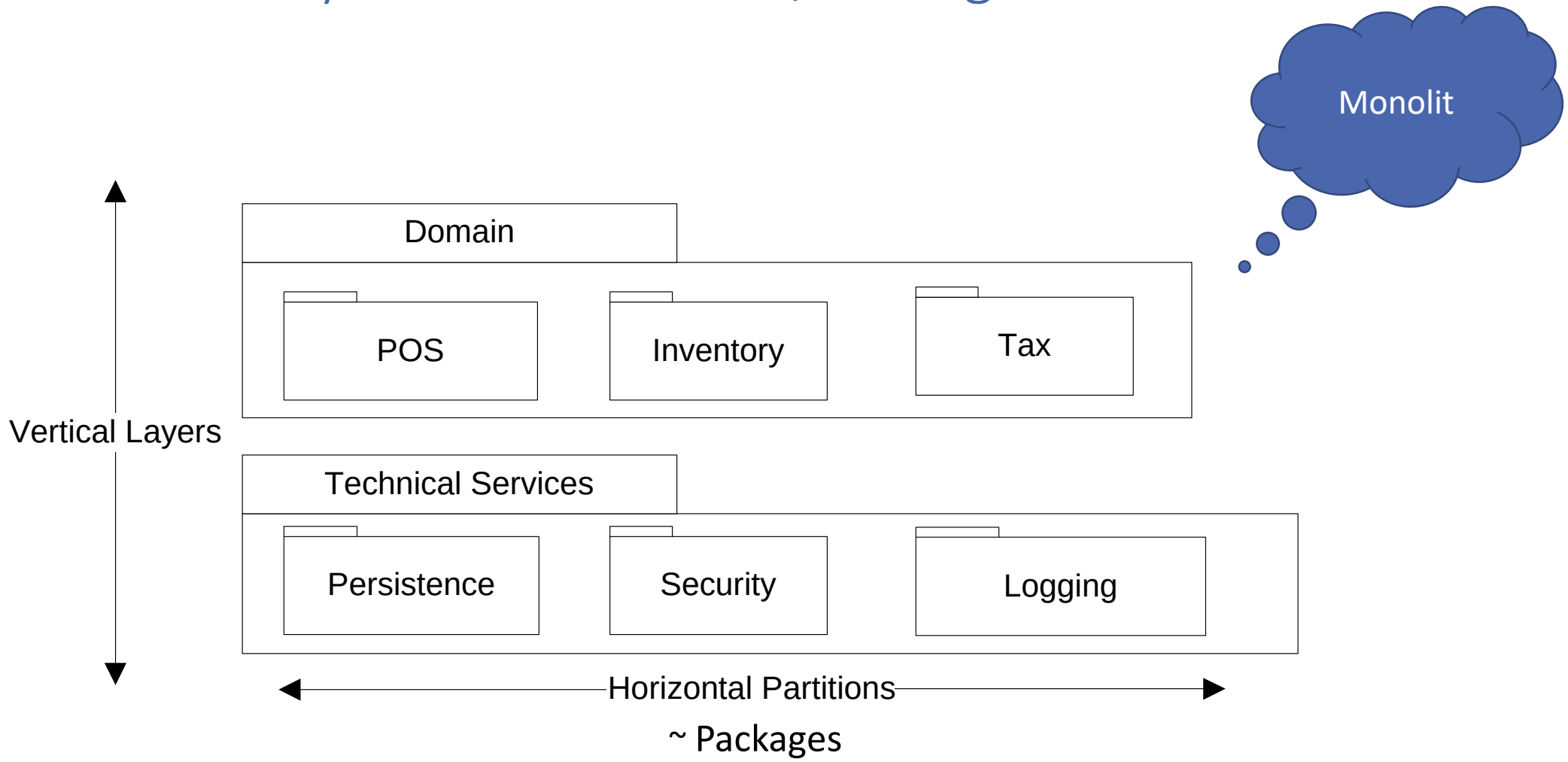




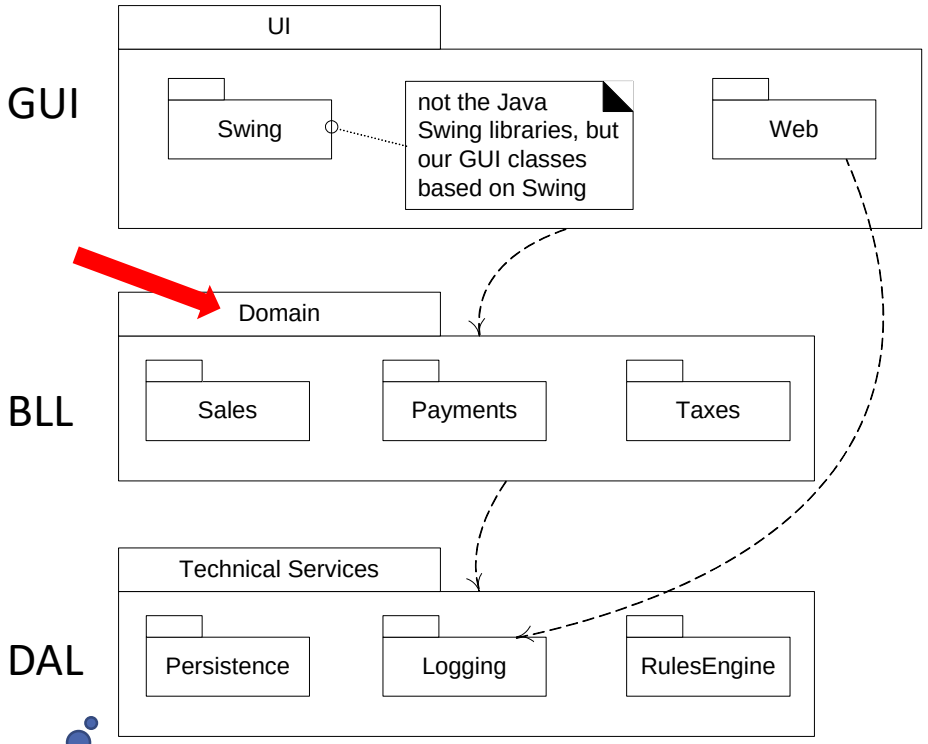
Benefits of Using Layers

- Separation of concerns (SRP)
  - separation of application-specific from general services.
  - separation of high-level from low-level services
- Reduces coupling and dependencies
- Improves cohesion
- Increases potential reuse
  - Lower layers can easily be reused in other applications
- Increases clarity
- Concurrent development by teams is aided by the logical segmentation
- A layer can be replaced
  - if interfaced based programming is used

Layers and Partitions/Packages



Typical Layers for a small Application



What is the problem?

Larman fig. 13.2

Concentrate all the code related to the domain model in one layer and isolate it from the user interface, application, and infrastructure code.

The domain objects, free of the responsibility of displaying themselves, storing themselves, managing application tasks, and so forth, can be focused on expressing the domain model.

This allows a model to evolve to be rich enough and clear enough to capture essential business knowledge and put it to work.

CQRS Pattern

CQRS stands for **Command Query Responsibility Segregation**

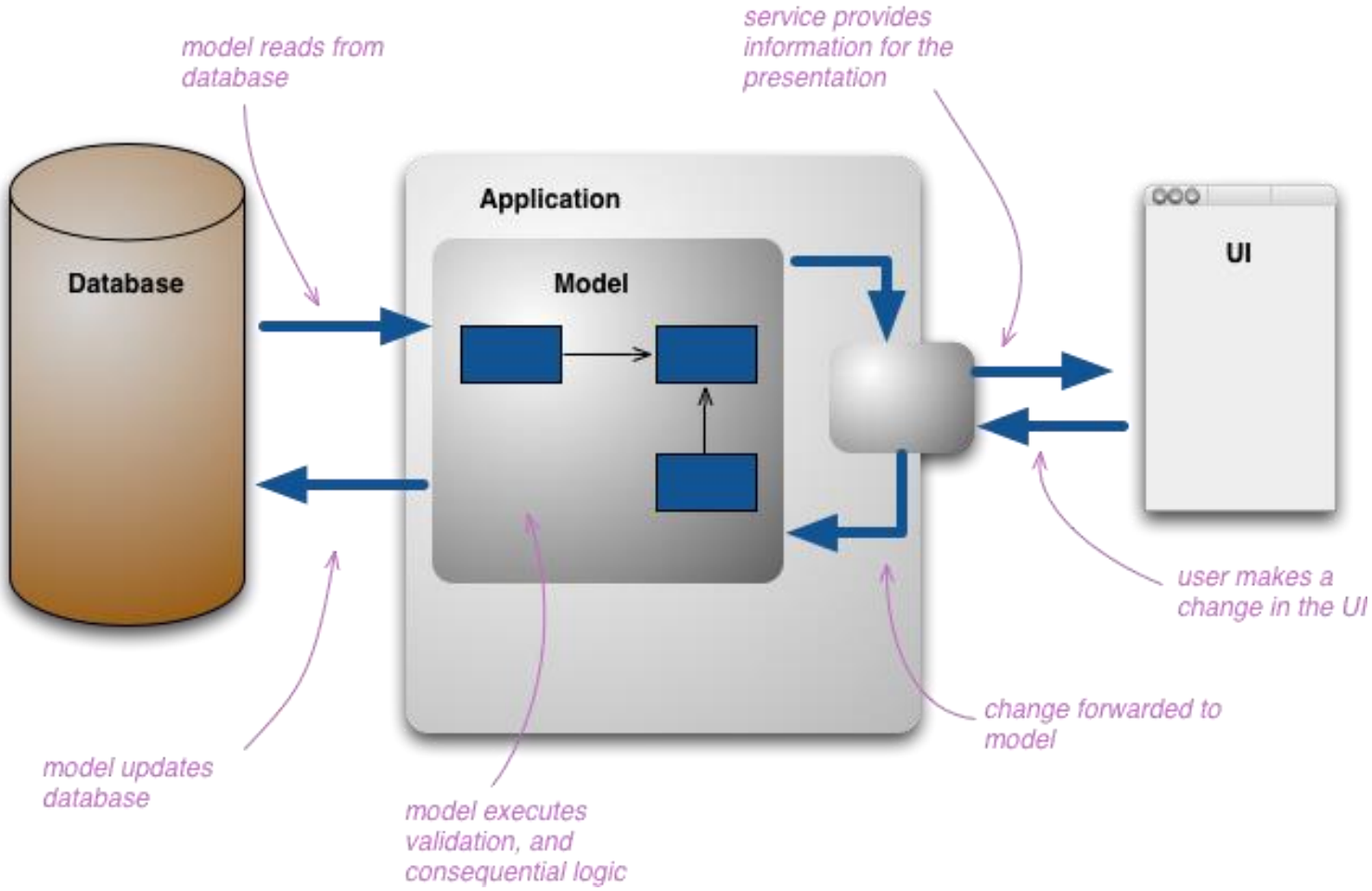
Command–query separation (CQS)

- It states that every method should either be a command that performs an action, or a query that returns data to the caller, but not both.
  - In other words, asking a question should not change the answer.
  - Formulated by Bertrand Meyer.

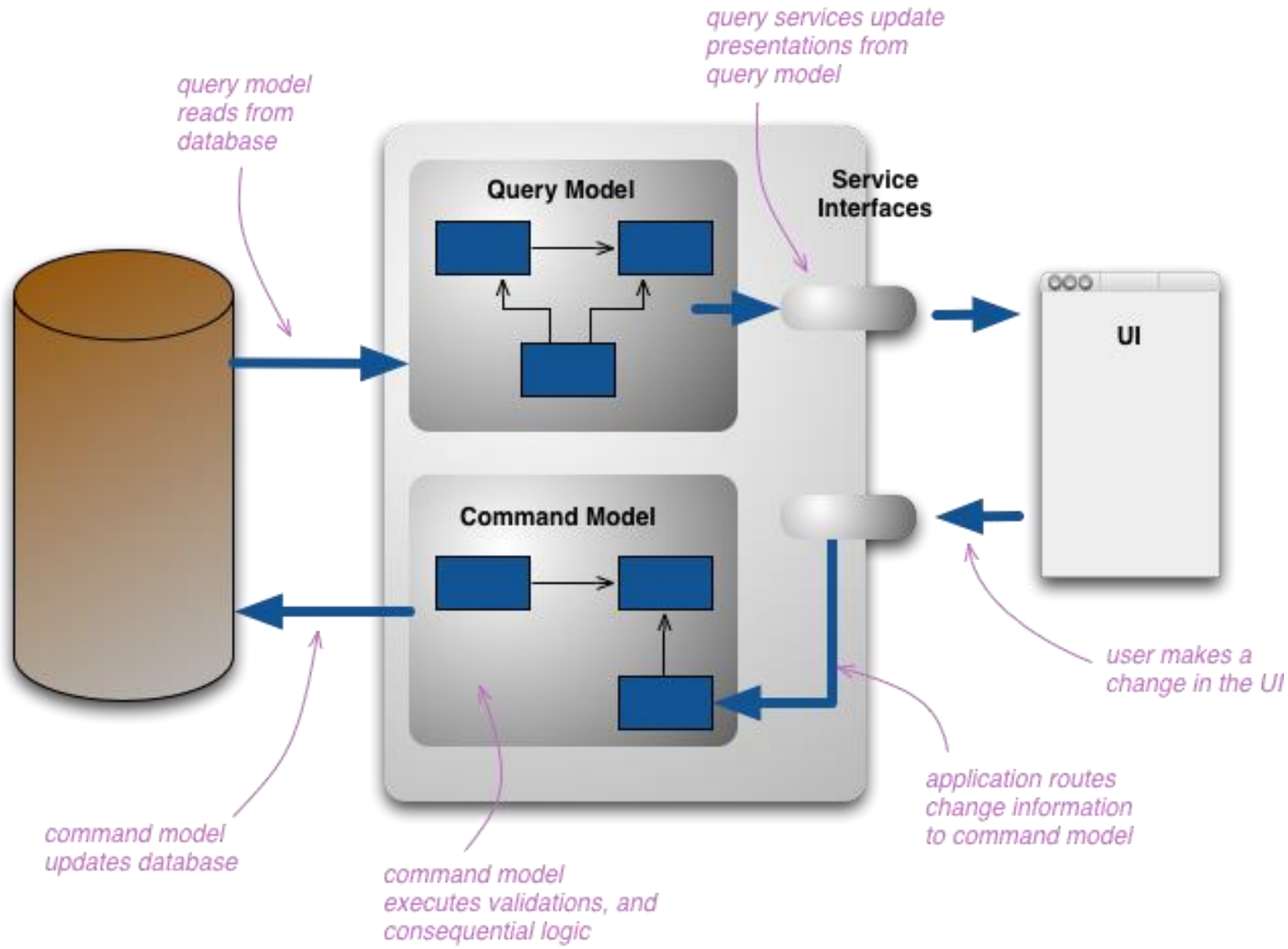
- **Command query responsibility segregation** (**CQRS**) applies the CQS principle by using separate *Query* and *Command* objects to *retrieve* and *modify* data, respectively.
  - Described by Greg Young

a CRUD datastore

- The mainstream approach people use for interacting with an information system is to treat it as a CRUD datastore.



Using CQRS


query services update presentations from query model
query model reads from database
Query Model
Service Interfaces
UI
Command Model
user makes a change in the UI
application routes change information to command model
command model updates database
command model executes validations, and consequential logic

The CQRS pattern

- The change that CQRS introduces is to split that conceptual model into separate models for update and display, which it refers to as Command and Query respectively following the vocabulary of CommandQuerySeparation.
- The rationale is that for many problems, particularly in more complicated domains, having the same conceptual model for commands and queries leads to a more complex model that does neither well.

Warning from Martin Fowler

- Despite these benefits, **you should be very cautious about using CQRS**. Many information systems fit well with the notion of an information base that is updated in the same way that it's read, adding CQRS to such a system can add significant complexity.

Gall's Law

A complex system that works is invariably found to have evolved from a simple system that worked. A complex system designed from scratch never works and cannot be patched up to make it work. You have to start over with a working simple system.

John Gall (1975)
Systemantics: How Systems Really Work and How They Fail p. 71

References & Links

- Hexagonal architecture
  https://alistair.cockburn.us/hexagonal-architecture/
- The Clean Architecture
  https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html
- The Onion Architecture
  https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/
- CQRS by Martin Fowler
  https://martinfowler.com/bliki/CQRS.html
- CQRS Documents by Greg Young
  https://cqrs.files.wordpress.com/2010/11/cqrs_documents.pdf