Modern software architecture

Sebastian Ichim, PeakIT 03,
18 October 2020

Free - Interactive - Practical exercises

Diamond Sponsors:

Partners:

Valorile AgileHub

INDEPENDENȚA

- Trainerii și participanții vin pe cont propriu, fără a promova vreo firmă
- Nimeni nu reprezentă interesele nici unei firme
- La începutul cursului, când ne prezentăm, spunem care e rolul și experiența noastră, fără a specifica firma la care lucrăm

EDUCAȚIA GRATUITĂ

- Cursurile sunt gratuite pentru toți participanții

VOLUNTARIATUL

- Toți trainerii sunt voluntari

ÎMPĂRTĂȘIREA EXPERIENȚEI PROPRII

- Majoritatea formatorilor **NU** sunt traineri profesioniști
- Formatorii lucrează în IT și au multă experiență practică în domeniul pe care îl predau

About me

- E-mail: ichim.v.sebastian@gmail.com
- Linked: https://www.linkedin.com/in/sebastian-ichim-97354737/
- AgileHub: https://agilehub.ro/
- Slack PeakIT: https://peakit003.slack.com/ - #curs-18oct-sebastian-ichim

- Expertise: Computer graphics, Software Design

Are you ready?

- Useful links:
  - https://github.com/ichimv/peakit03,
    - Source code
    - Presentation
  - Agilehub will publish recorded sessions on youtube channel
- Schedule:
  - First part: 10:30-12:00 = 1h 30
  - Second part: 12:15-13:30 = 1h 15

Answer pool...

Your programming language?

Your experience?

Type in chat...

Do you know the difference between software engineer and software developer?

Answer pool...

Who is working as an architect?

Who wants to become an architect?

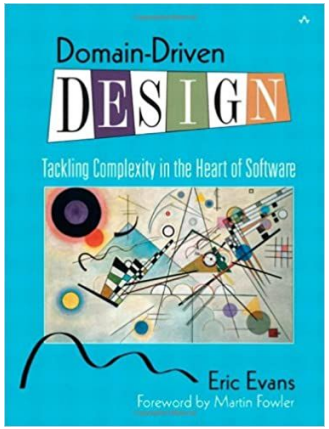
Key points

- What is Architecture and why?
- Architect role
- Design quality
- Coupling and Cohesion
- Monolith vs Microservices

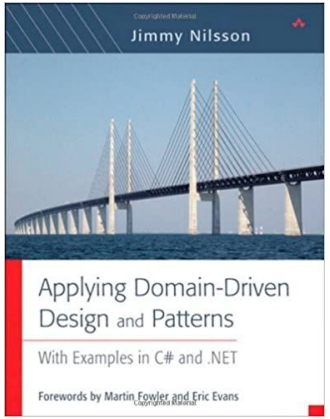
References


Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric Evans
Foreword by Martin Fowler
Copyrighted Material
The Addison Wesley Signature Series
PATTERNS OF ENTERPRISE APPLICATION ARCHITECTURE
MARTIN FOWLER
Copyrighted Material
Microsoft
Microsoft .NET: Architecting Applications for the Enterprise
Second Edition
Professional
Dino Esposito
Andrea Saltarello
AGILE SOFTWARE DEVELOPMENT
Principles, Patterns, and Practices
Robert C. Martin
Robert C. Martin Series
Clean Architecture
Robert C. Martin

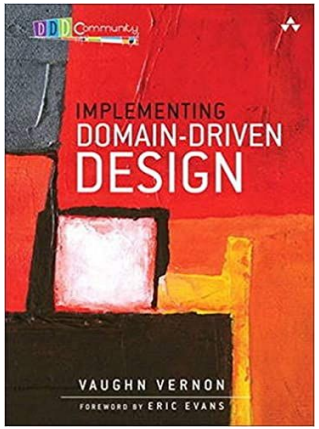
Domain-Driven Design (DDD)



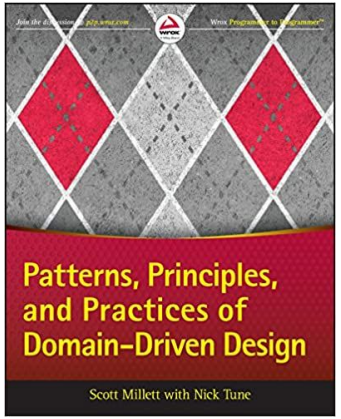
2003



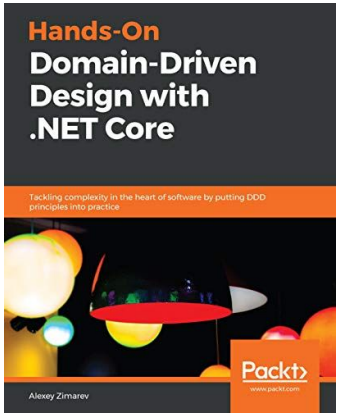
2006



2013



2014



2019

Goal of the software engineering

What's software architecture?

- Is it an art?

- Designing the fundamental structure of a system for a client

Architecture

- Requirements definitions
  - Functional
    - Required function to fulfill a given scenario
    - Description of the expected behavior
  - Nonfunctional - attribute of the system explicitly requested by stakeholders
    - Scalability
    - Security
    - Accessibility

- Implementation
  - Breakdown process -> set of specifications for the development team
  - The more agile, the more freedom and independence for developers

Architecture definition

- “The 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” from ANSI/IEEE Std 1471-2000

- **Software architecture** refers to
  - The fundamental structures of a software system
  - The discipline of creating and documenting such structures and systems
  - The strategic technical decisions

Architect role

- Strategic technical decisions
  - Has impact
    - On many people - cost, evolution
    - Performance
    - Decomposability
    - Safety, security
  - Requires long period of time
  - Are hard to change

- Examples of strategic technical decisions
  - Programming language
  - Framework/platform selection
  - Style: monolithic or based on microservices

Why architecture matter?

Friendly user interface

Low number of bugs

External quality

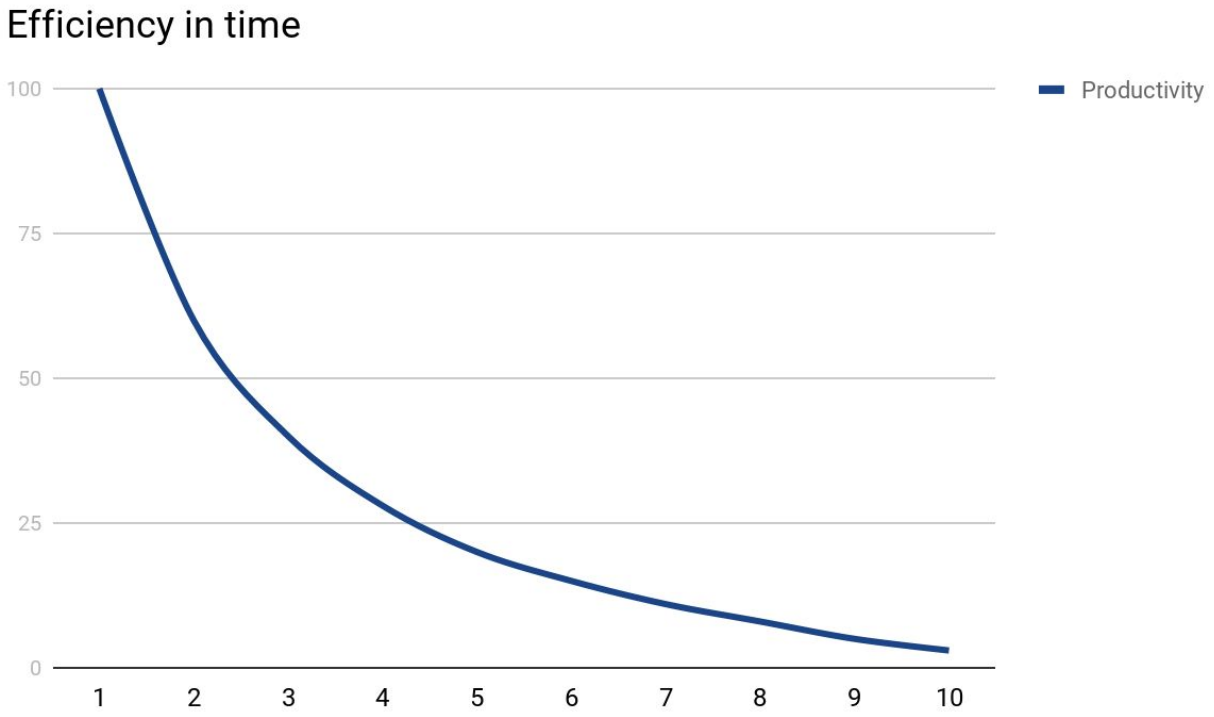
Good Modular Design

Internal quality

Is it worth the effort to design well?

- Can we reduce design activity?
  - Multiple shortcuts increase technical debt
  - Every change of the system requires more tangles, twists and knots
- Over time the mess becomes so big, so deep and so tall so we cannot clean it up
- As the mess grows, the development productivity continue to decrease, asymptomatically to zero

Bad design productivity in time



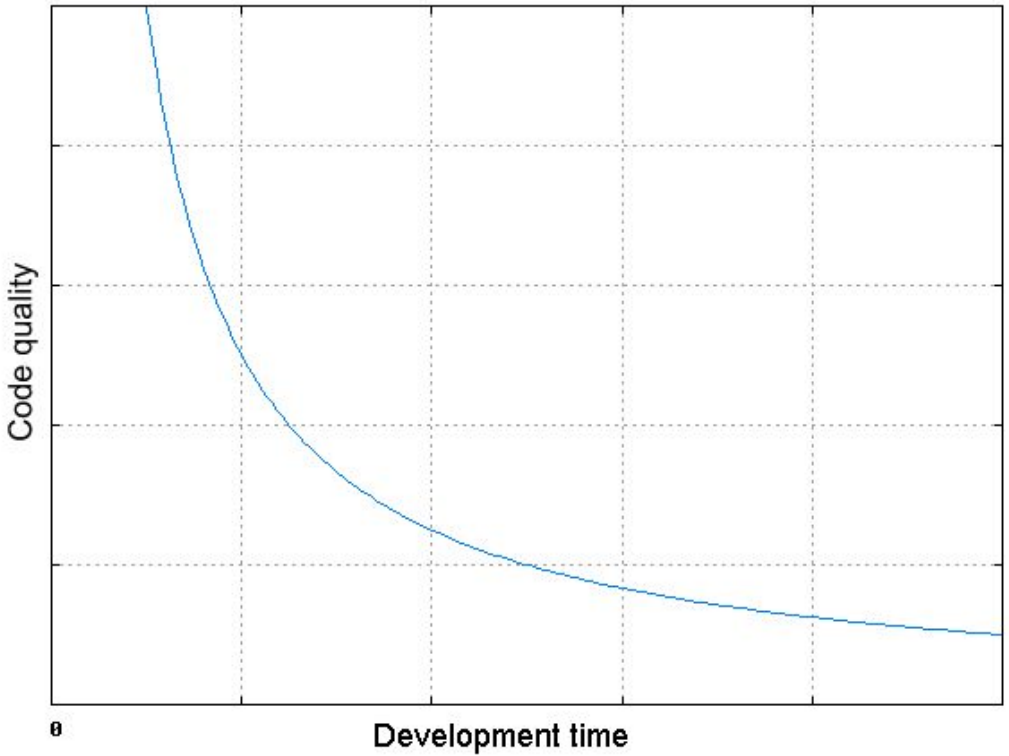
The cost of owning a mess - Clean Code, Uncle Bob

- Productivity is decreasing because of technical debt
- Product management adds more staff hopping the productivity will increase
- The new stuff does not fit in the design of the system
  - Product management doesn't know if a change matches the design or not
- Everyone on the team is under horrific pressure to increase productivity
- They all make more and more mess, driving productivity even lower

Bad code design characteristics - Uncle Bob

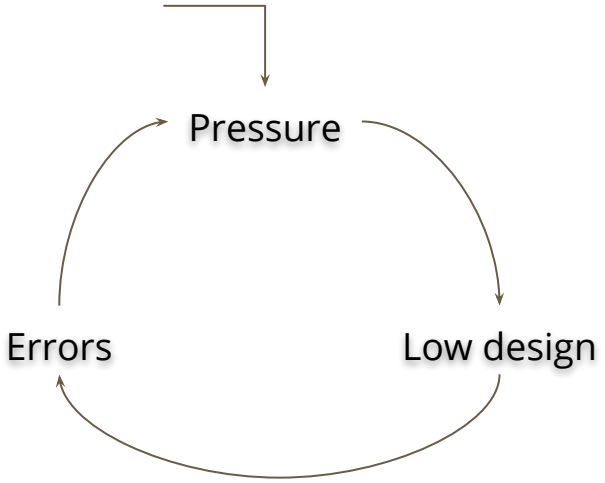
- **Rigidity**
  - It is hard to change because every change affects too many other parts of the system
- **Fragility**
  - When you make a change, unexpected parts of the system break
- **Immobility**
  - It is hard to reuse in another application because it cannot be **decoupled** from the current application

Low design quality impact on development time



The stress spiral

- When the stress level is growing
  - The more stress and pressure the less attention to design
  - Less design means quick and dirty -> bad code, -> errors, ->less efficiency
  - The more errors the team is doing, the more stress the people feels



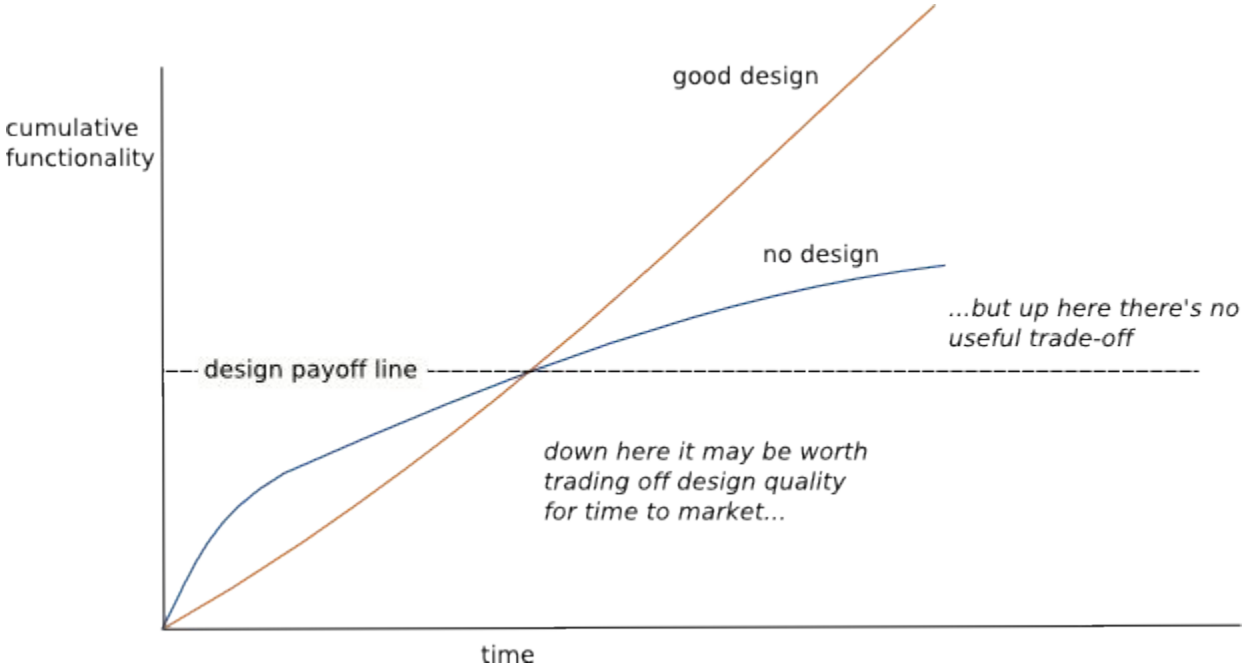
- There is no quick and dirty, there is only dirty
  - The “quick” work makes us spend more time comparing with making it clean from the beginning

Cumulative functionality in time - Martin Fowler

- **Bad design**
  - Expends no effort on design activities, whether they be up front design or agile techniques
  - This project produces function faster initially if there's no effort spent on design
  - The code base degradate and becomes messy and harder to modify -> lowers the productivity

- **Good design**
  - Keeps productivity more constant so at some point it overtakes the cumulative functionality of the no-design project and will continue to do better

Design payoff line



https://martinfowler.com/bliki/DesignStaminaHypothesis.html

Good Software Design characteristics

Correctness	Understandability	Efficiency	Maintainability
If the results are correct for every input, the design is accepted and is considered that the software produced according to this design will function correctly.	If the design is easy and self-explanatory, it would be easy for the developers to implement it and build the same software that is represented in the design.	If the design is describing software that is efficient and useful, then the developed software would also stand on the same level of efficiency.	Any change made in the software design must not affect the other available features, and if the features are getting affected, then they must be handled properly.

What is an architect really doing?

- Internal focus
  - Focused on architecting per se: architectural design, prototyping, evaluating, documenting

- External focus - interacting with other stakeholders
  - Inwards:
    - Getting input from the outside world:
      - Listening to customers, users, product manager, and other stakeholders (developers, sales, marketing, customer support, etc.).
      - Learning about technologies, other systems' architecture, and architectural practices
  - Outwards
    - Providing information or help to other stakeholders or organizations: communicating the architecture: project management, product definition

Architectural antipatterns

- Creating a perfect architecture, for the wrong system
  - Miss the target if do not communicate regularly with customer
- Creating a perfect architecture, but too hard to implement
  - Do not understand skill, capability and experience of development team (stress/frustration)
- Architects in their ivory tower
  - Isolated in other part of the organization
  - Good at manipulating abstractions, wide experience of a range of systems and technologies, good communication skills, good domain knowledge: use these skills for other tasks than just building architectural views
- The absent architects
  - Always away doing fascinating things or fighting fires
  - Very easy to slip in this mode after some initial good progress and early successes

Roles and responsibilities of an architect

- Defining architecture of the system
  - Understanding requirements, qualities
  - Extracting architecturally-significant requirements
  - Making choices, synthesizing a solution
  - Exploring and validating alternatives by prototyping activities

- Maintaining the architectural integrity of the system
  - Regular reviews, writing guidelines
  - Presenting the architecture to various parties, at different levels of abstraction

- Evaluating technical risks - Working out risk mitigation strategies/approaches

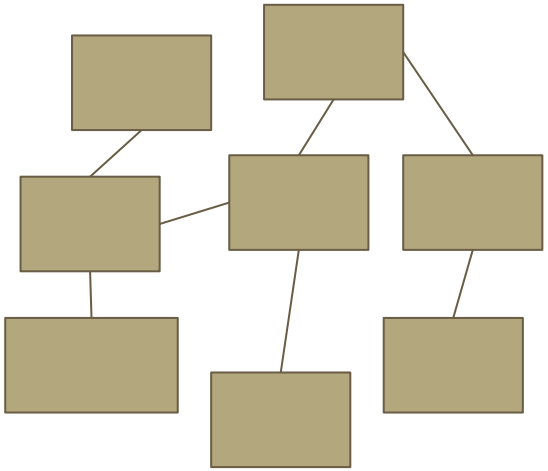
- Participating in project planning - Proposing order and content of development iterations
- Consulting with (arch/detailed) design, implementation and integration teams
- Assisting product management and future product definitions
  - Have insights into what is feasible, doable, or science fiction

Cohesion and coupling

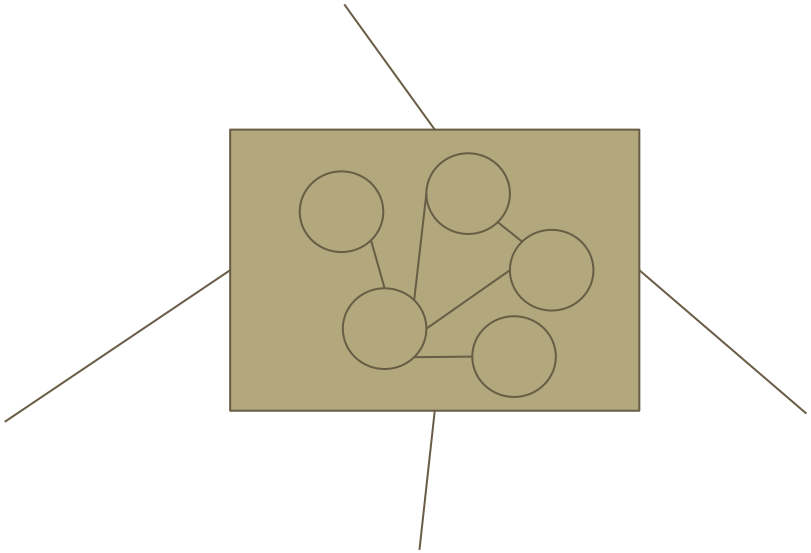
- Cohesion:
  - Degree of clarity of a component responsibility
  - How much the elements are functionally related

- Coupling:
  - Degree of interdependence between components
    - How much should know about one component to understand other component
    - How much is affected one component when change another one

Design complexity



Coupling between components



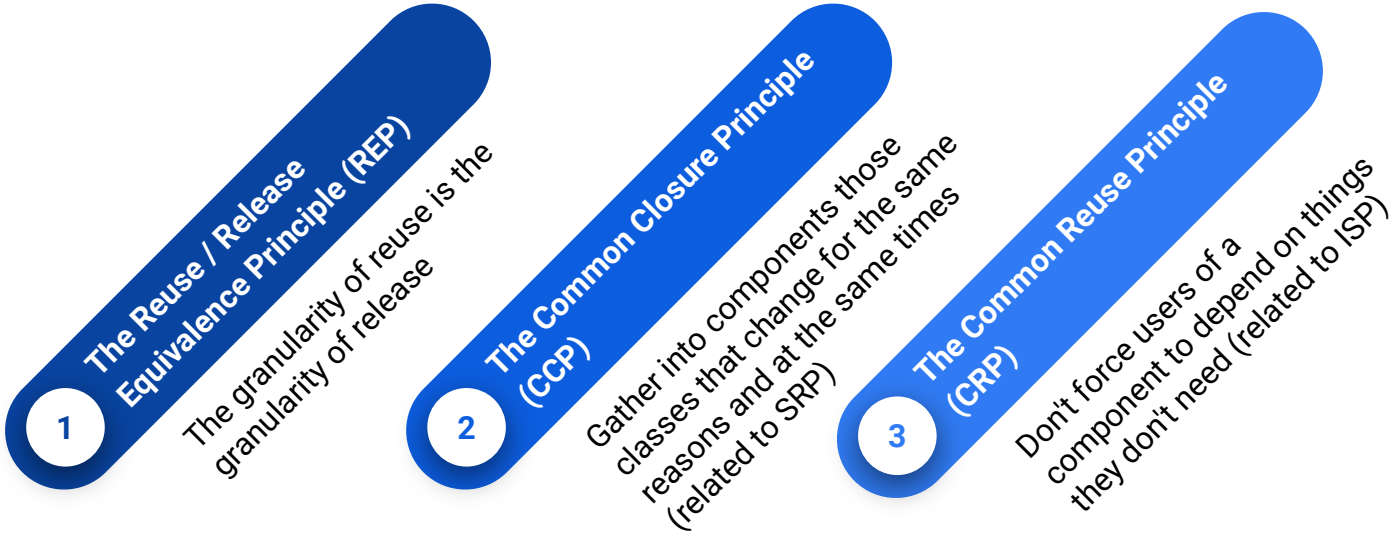
Cohesion inside a component

Cohesion and coupling evaluation

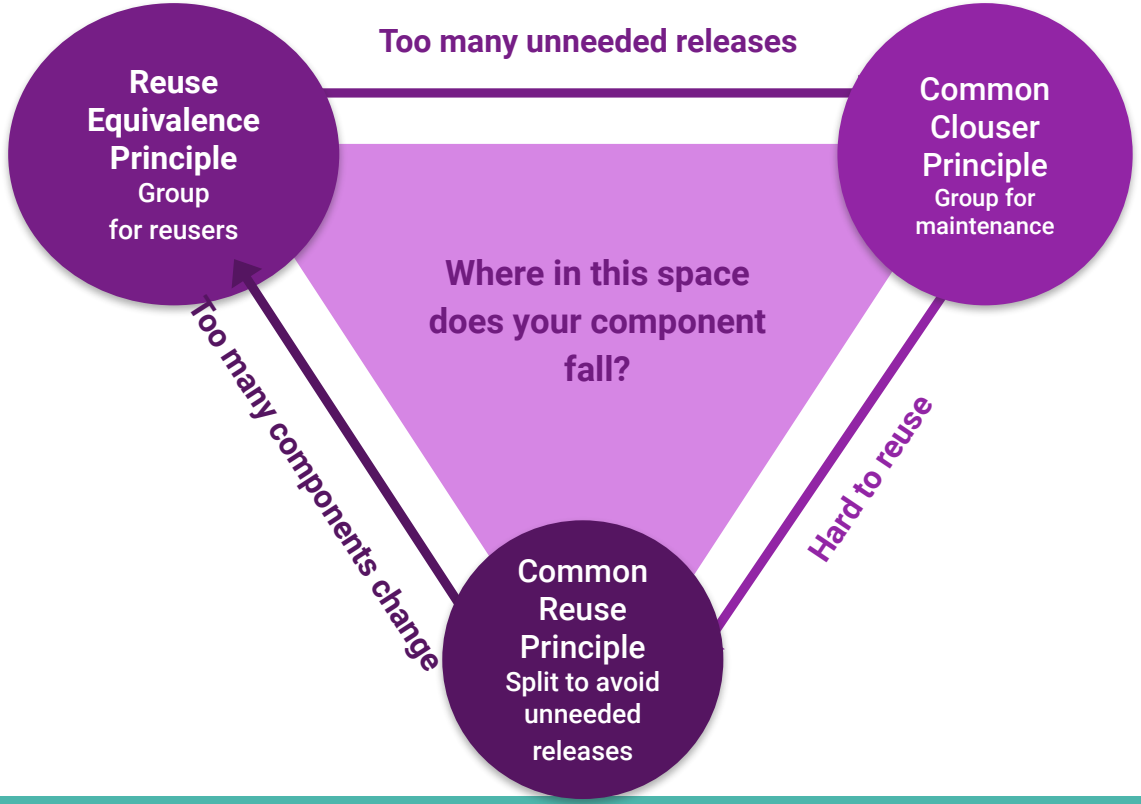
- Low cohesion
  - Many unclear/unrelated responsibilities
- **High cohesion**
  - Clear responsibility of the component

- Tightly coupled
  - Multiple connections between application modules
- **Loosely coupled**
  - Minimal connections between application modules (Lego)

Component cohesion principles - Robert C. Martin



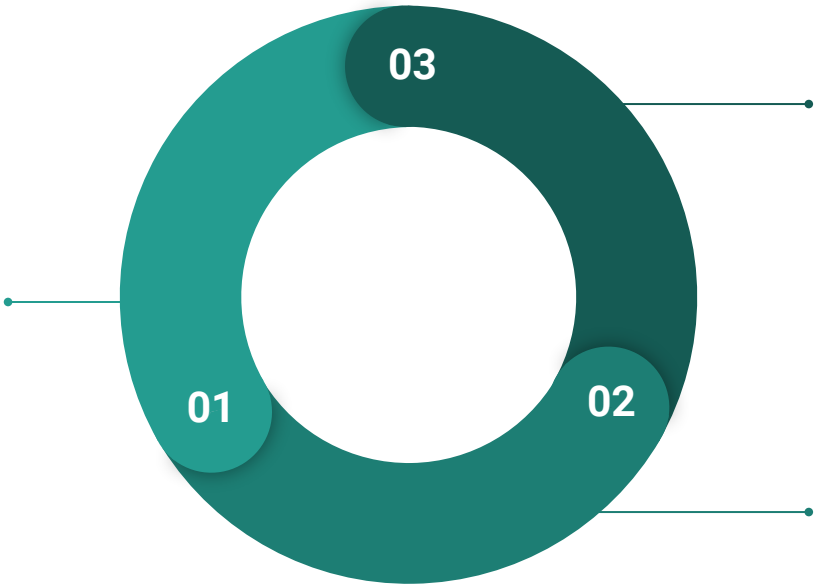
Tension Diagram for Component Cohesion - Robert C. Martin



Component Coupling principles- Robert C. Martin

The Acyclic Dependencies Principle (ADP)

Allow no cycles in the package dependency graph



The Stable Abstractions Principle (SAP)

A package should be as abstract as it is stable.

The Stable Dependencies Principle (SDP)

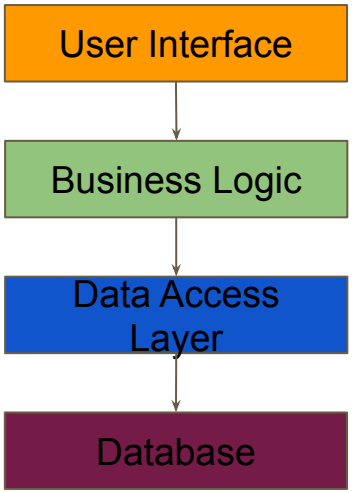
Depend in the direction of stability.

Modern architecture characteristics

- Independent of frameworks
  - Architecture does not depend on the existence of some library
- Testable
  - Business rules can be unit tested. UI, Dbase, external dependencies are mocked
- Independent of UI
  - UI can be easily changed without changing the rest of the system
- Independent of Dbase
  - Can swap Oracle or SQL
- Independent of outside world
  - Isolated business rules

Monolithic architecture

- Traditional way of building applications
- Built as a single, unified and indivisible unit
- Includes
  - A client-side user interface
  - A server side-application
  - A database
- All functions are managed and served in one place
- One large code base and lack modularity
- The whole stack is changed at once



Monolithic architecture - strengths

- Less cross-cutting concerns
  - Cross-cutting concerns are the concerns that affect the whole application such as logging, handling, caching, and performance monitoring.
  - In a monolithic application, this area of functionality concerns only one application so it is easier to handle it.

- Simple to develop
  - Standard way of building applications
  - Engineering team has the right knowledge/capabilities to develop

- Easier debugging and testing
  - A monolithic app is a single indivisible unit
  - Can run end-to-end testing much faster

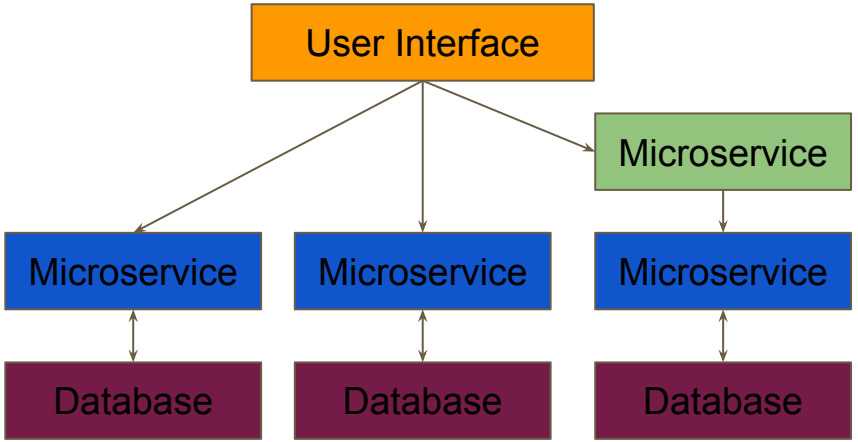
- Simple to deploy
  - Deploy just one file or directory

Monolithic architecture - weaknesses

- Complexity
  - Scaled up monolithic application becomes too complicated to understand
  - Hard to manage a complex system within one application
- Maintenance
  - Harder to implement changes in such a large and complex application with highly tight coupling.
  - Any code change affects the whole system so it has to be thoroughly coordinated. This makes the overall development process much longer.
- Scalability
  - Cannot scale components independently, only the whole application
  - The more users acquire, the more problems with monolith -> redevelop completely
- New technology barriers
  - Extremely problematic to apply a new technology in a monolithic application -> rewrite

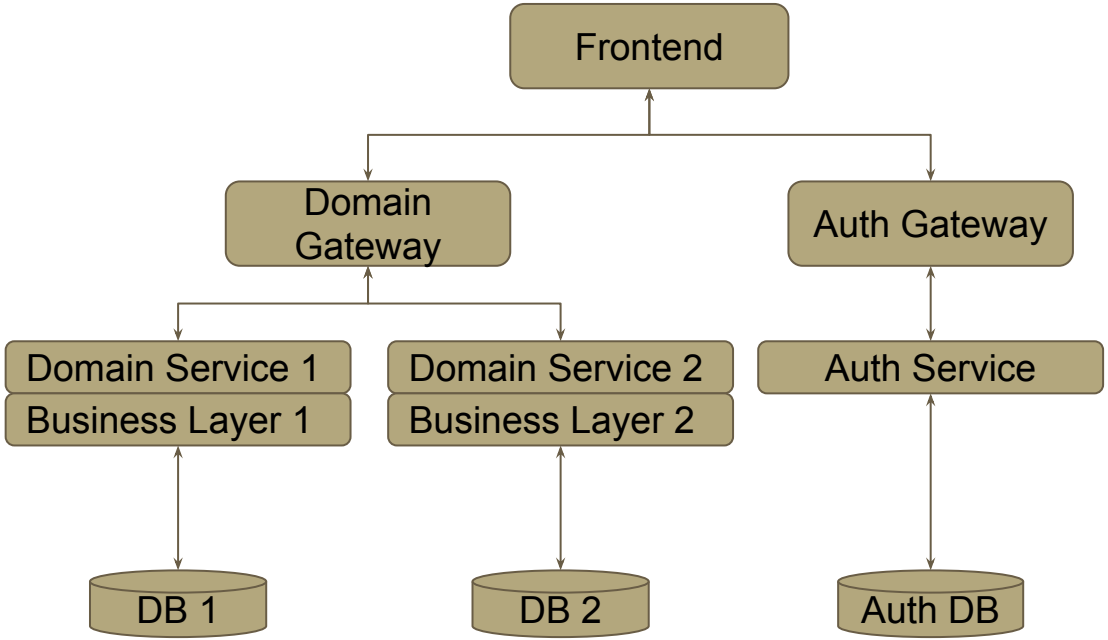
Microservices architecture

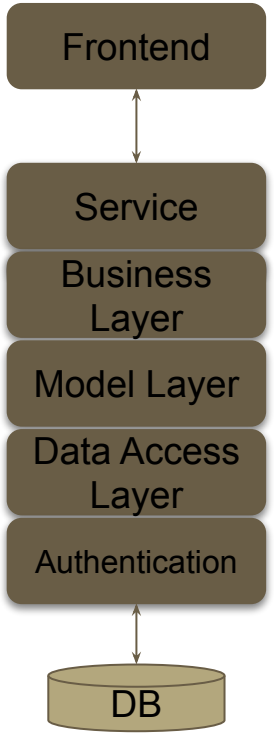
- A collection of independently deployable small services



- Every microservice
  - Perform specific function
  - Covers its own scope
  - Has own business logic and database
  - Running in its own process
  - Communicating through API
  - Can be updated, deployed and scaled independently

Microservices vs Monoliths





Gateways benefits

- Break up authentication types or access methods.
- Break up domains
- Direct traffic by user type, region, or tenant
- Break up external traffic from administration traffic

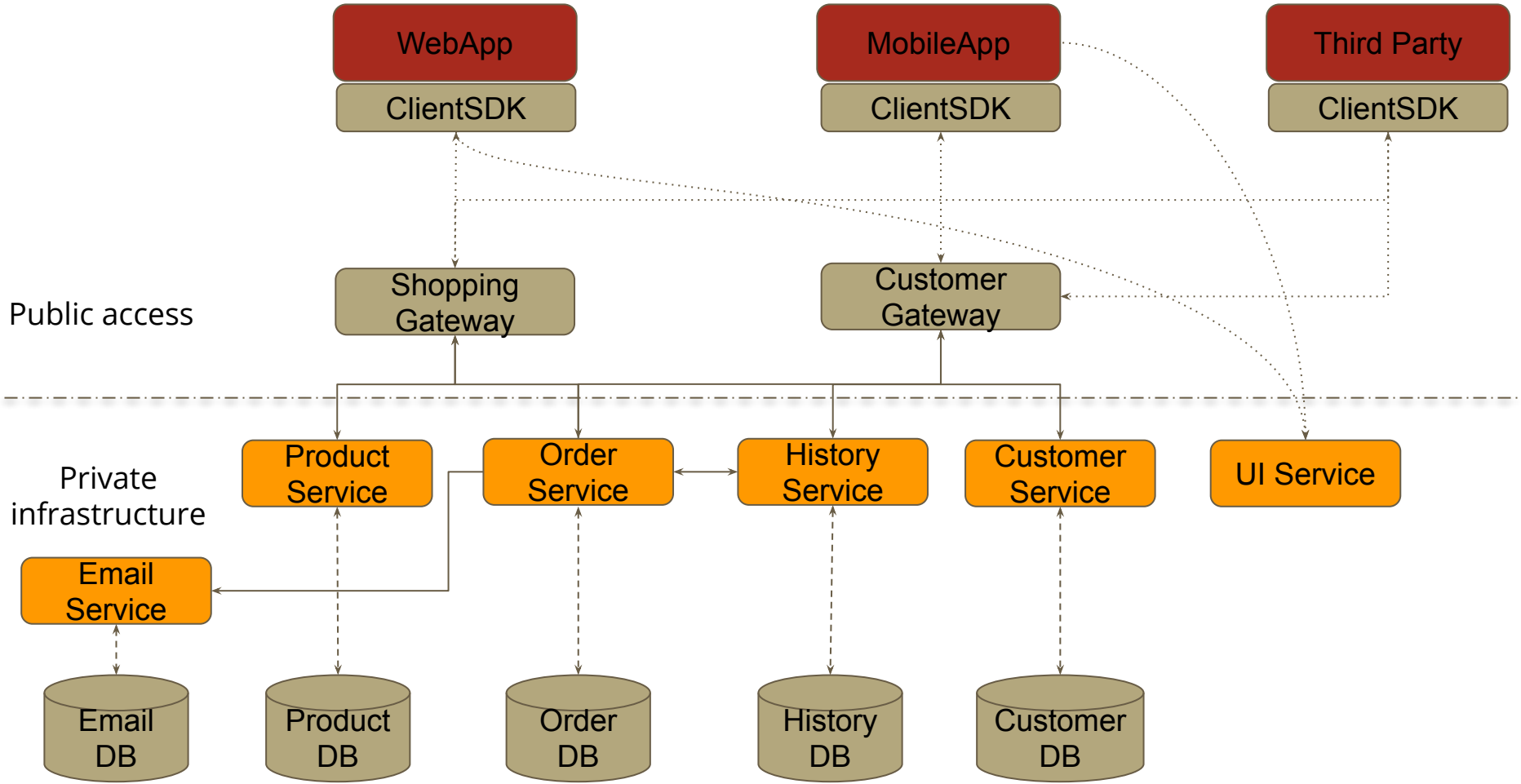
Exercise - simple e-commerce application

- Build the architecture of an App that:
  - Tracks Customers
  - Allows Orders to be placed
  - Keeps History of the orders

- We want to allow this API to support:
  - Cross platform modern App with web front-end
  - Mobile app
  - Third party integrations

- We also know that users browse Products 100 times at peak more often than they place orders

- We want to send a user an email when he makes an order


WebApp
MobileApp
Third Party
ClientSDK
ClientSDK
ClientSDK
Public access
Shopping Gateway
Customer Gateway
Private infrastructure
Product Service
Order Service
History Service
Customer Service
UI Service
Email Service
Email DB
Product DB
Order DB
History DB
Customer DB

Microservices architecture - strengths

- Independent components (flexibility)
  - All the services can be deployed and updated independently
  - A bug in one microservice has an impact only on a particular service and does not influence the entire application -> lower risk and fewer errors
  - It is much easier to add new features to a microservice application than a monolithic one
- Easier understanding
  - Split up into smaller and simpler components, a microservice application is easier to understand and manage.
  - Can just concentrate on a specific service that is related to a business goal
- Better scalability
  - Microservices can be scaled independently
- Flexibility in choosing the technology and framework for every service

Microservices architecture - weaknesses

- Extra complexity
  - Have to choose and set up the connections between all the modules and databases
  - All independent services have to be deployed independently
- System distribution
  - A microservices architecture is a complex system of multiple modules and databases, so all the connections have to be handled carefully
- Cross-cutting concerns
  - Have to deal with a number of cross-cutting concerns: externalized configuration, logging, metrics, health checks, and others
- Testing
  - A multitude of independently deployable components makes testing a microservices-based solution much harder

Choosing a monolithic architecture

- Small team
  - A startup do not need to deal with the complexity of the microservices architecture
  - A monolith can meet all business needs, so there is no emergency to follow the hype and start with microservices.

- Simple application
  - Small applications which do not demand much business logic, superior scalability, and flexibility work better with monolithic architectures

- No microservices expertise
  - Microservices require profound expertise to work well and bring business value (If you want to start a microservices application from scratch with no technical expertise in it, most probably, it will not pay off)

- Quick launch
  - A monolithic architecture is the best choice for releasing ASAP an application
  - Allow to spend less time initially and validate business idea

Choosing a microservices architecture

- Microservices expertise
  - Extremely risky to build a microservice application without proper skills and knowledge
  - DevOps and Containers experts are required, since the concepts are tightly coupled
  - Domain modelling expertise is a must. Dealing with microservices means splitting the system into separate functionalities and dividing responsibilities

- A complex and scalable application
  - The microservices architecture will make scaling and adding new capabilities to your application much easier
  - Develop a large application with multiple modules and user journeys

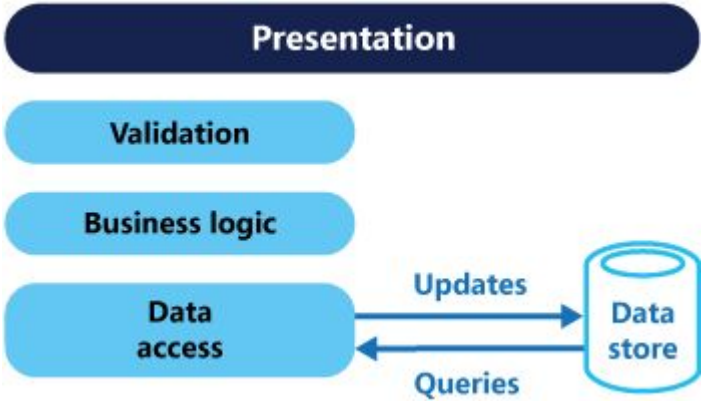
- Enough engineering skills
  - Enough available resources to handle all the processes, since a microservice project comprises multiple teams responsible for multiple services

“Don’t even consider microservices unless you have a system that’s too complex to manage as a monolith. The majority of software systems should be built as a single monolithic application. Do pay attention to good modularity within that monolith, but don’t try to separate it into separate services.”

- Martin Fowler

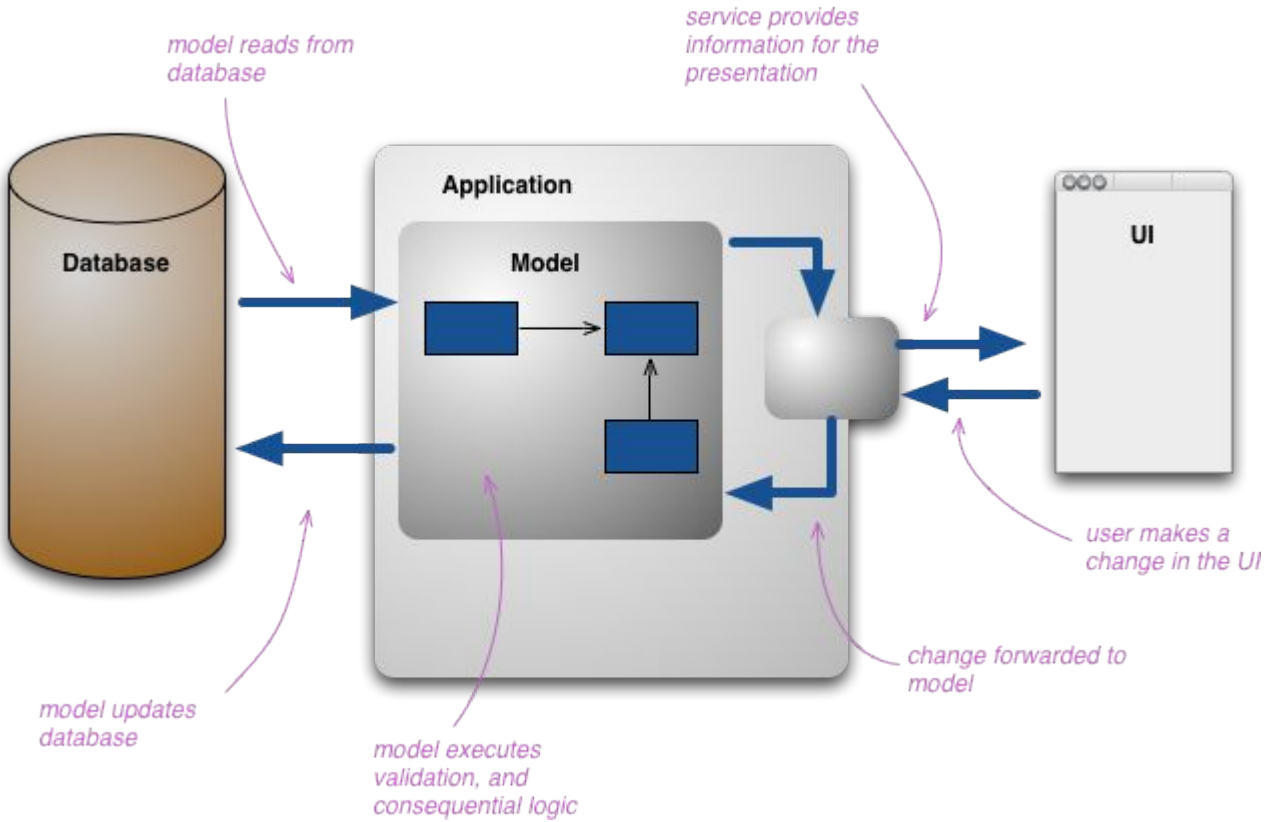
Single model information system

- **C**reate new records
- **R**ead records
- **U**pdate existing records
- **D**elete records




```
public interface IObject {
        int ID { get; set; }
}
public interface IDocument {
        int Create(IObject newObject);
        IObject Read(int objectId);
        bool Update(IObject existingObject);
        bool Delete(int objectId);
}
```

https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs

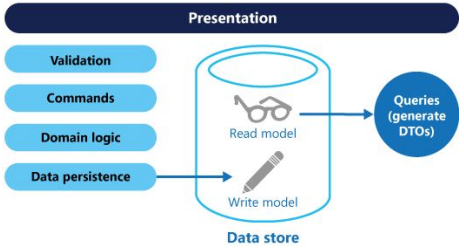


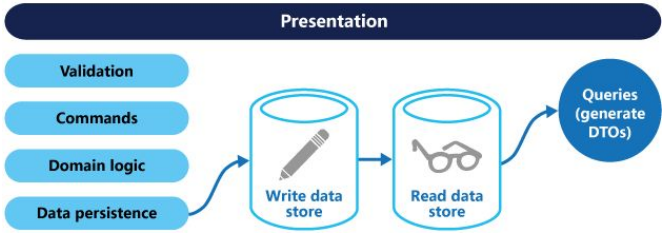
https://martinfowler.com/bliki/images/cqrs/single-model.png

Command Query Responsibility Segregation (CQRS)

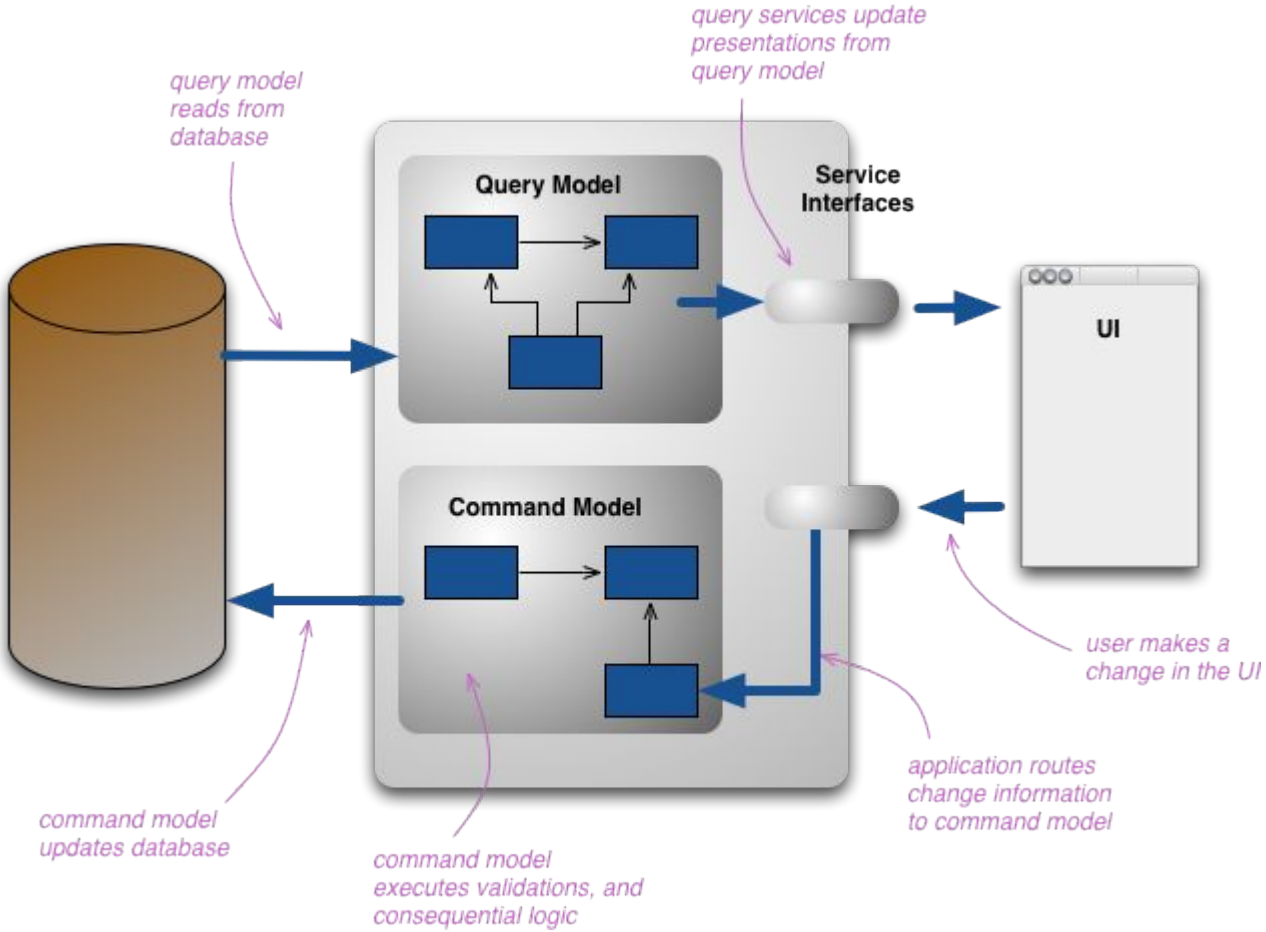
- It is a design pattern

- Separate reads and writes into different models
  - Display information using queries
    - Returns a DTO without database modification -> it is fast
    - Queries => GET methods
  - Update information using commands
    - Command is task based and it's not data centric
    - Can be placed in a queue and can be processed asynchronous
    - Commands => POST/PUT/DELETE methods





https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs



https://martinfowler.com/bliki/images/cqrs/cqrs.png

Why CQRS?

- Single Responsibility Principle by design
  - Loosely coupled architecture
  - Segregation of read and write models -> more flexible and maintainable

- Can optimize Read model or Comand model any time

- Models separation
  - Different object models, could run in different logical processes, even on separate hardware
  - Each command is responsible only for a single operation and change the state
  - Each query do not change the state

CQRS benefits

- Independent scaling
  - Allows the read and write workloads to scale independently -> fewer lock contentions
- Can optimize data schemas
  - Read side optimized for queries and Write side optimized for updates
- Security
  - Easier to ensure that only the right domain entities are performing writes on the data
- Separation of concerns
  - More maintainable and flexible models
  - Complex business logic goes into the write model
  - Read model can be relatively simple
- Simpler queries
  - Application can avoid complex joins when querying

Exercice - Hotel management App

- Build an App that:
  - Tracks Guests
    - ID, First Name, Last Name
  - Allows Orders to be placed
    - Room price:
      - Single - 50 EU, Double - 100 EU, Triple - 150 EU, Penthouse - 200 EU
  - Keeps Room History
    - Room Id, Guest Id and Order Id
  - Generate a Hotel report
    - Number of unique guests, Total number of orders, Total amount of money

Draw components diagram

Summary

- Architecture matters
- Architect role
- Design quality
- Coupling and Cohesion
- Monolith vs Microservices

Bibliografie

- Martin Fowler defines Software Architecture
- Microservice Architecture & Design
- The Clean Code Blog
- Clean Architecture – Components and Component Cohesion
- The Total Cost of Owning a Messy Code
- What do software architects really do?

Întrebări

Feedback

http://bit.ly/peakit003-feedback

Completați acum

Durează 2-3 minute

Feedback anonim - pentru formator si AgileHub