C++ Software Engineering

for engineers of other disciplines

Module 8

"Software Engineering"

2nd Lecture: Software Architecture

ALTEN

Spring 2022

Gothenburg, Sweden

Architecture

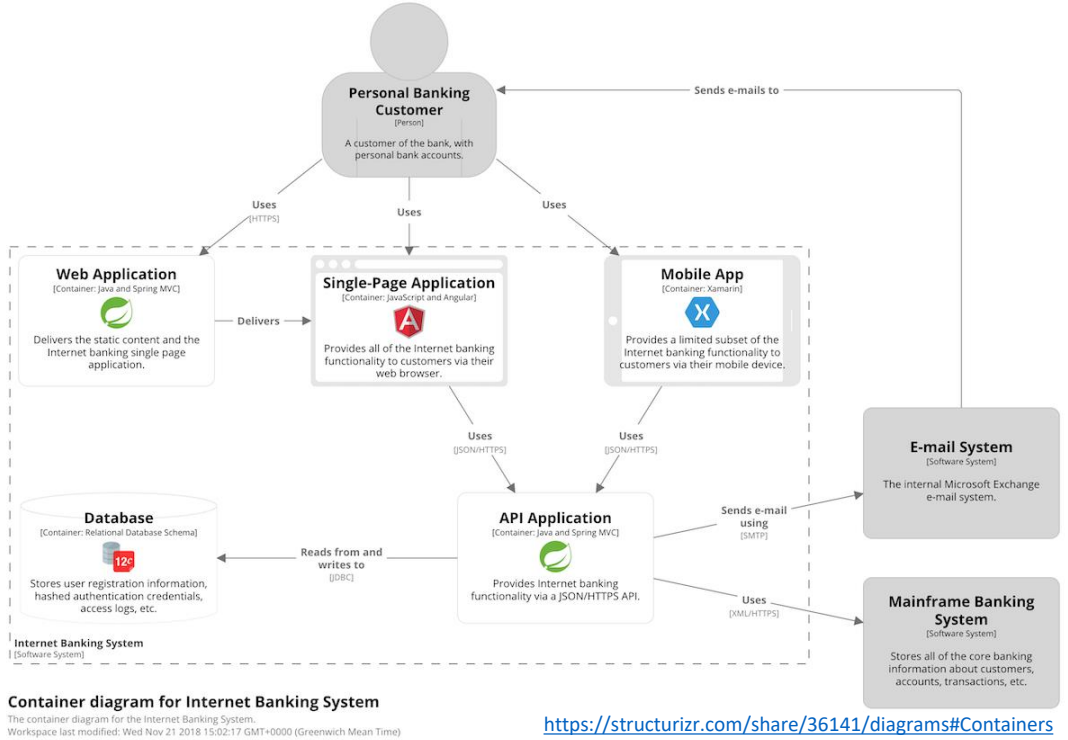
ALTEN

- Most abstract design of the system.
- Translates *requirements* into *architecture*.
- *Problem* is *decomposed* into *components* which together provide the required functionalities, they way it should!

Analysis → Design → Evaluation → Evolution

- Requirements are *dynamic* throughout the project, yet some design decision must be taken early in the project! *Modular*, design with *high cohesion* allows for smoother future modifications.

" Most decomposition paradigms suggest breaking down a program into parts to ***minimize the static dependencies among those parts****, and to* ***maximize the cohesiveness of each part****."* https://en.wikipedia.org/wiki/Decomposition_(computer_science)#Decomposition_paradigm



Container diagram for Internet Banking System

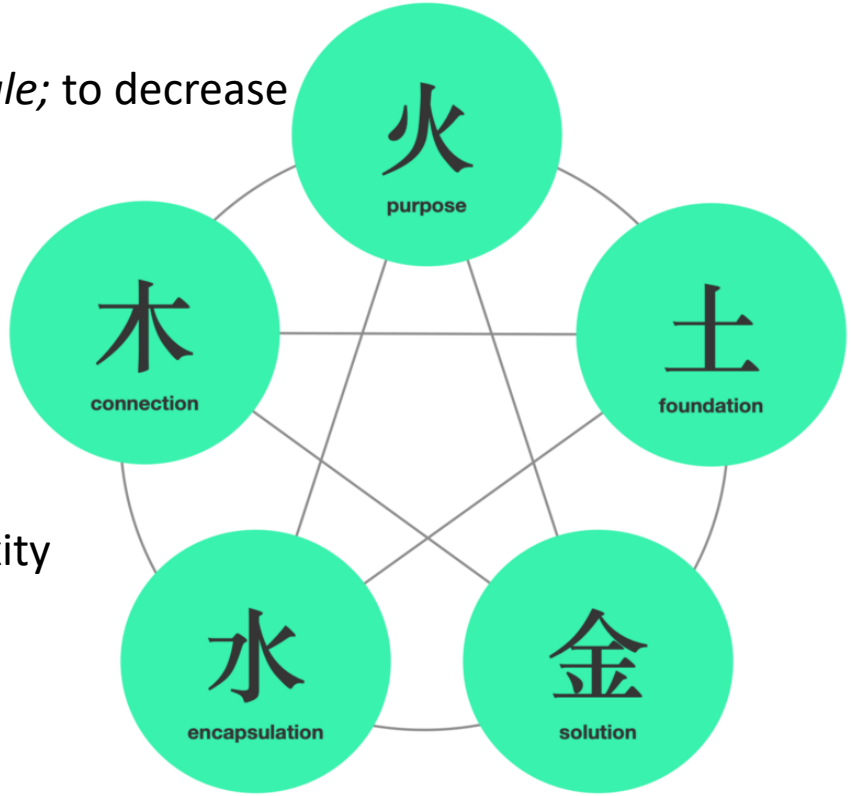
The container diagram for the Internet Banking System.
Workspace last modified: Wed Nov 21 2018 15:02:17 GMT+0000 (Greenwich Mean Time)

https://structurizr.com/share/36141/diagrams#Containers

Modularity

- A design theory which *subdivides* a system into smaller parts called *module;* to decrease *complexity* and increase *clarity.*

- *Modules:*
  - Have a *crystal-clear* purpose
  - Have well defined and minimal inter-connections to reduce complexity
  - *Encapsulate* functionality and expose as little as possible
  - *Solve* a problem
  - As the system foundation, are well *tested for compliance with requirements*

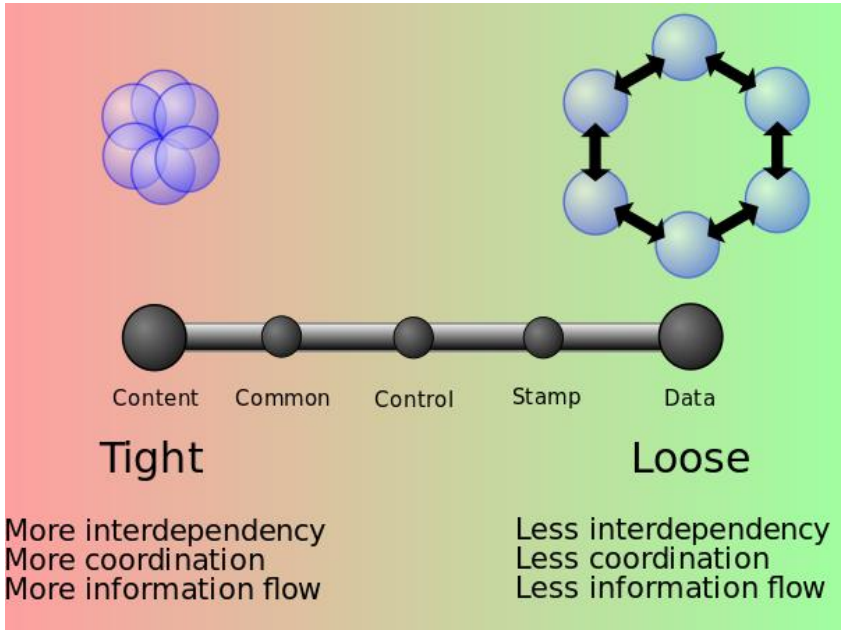


https://www.genui.com/resources/5-essential-elements-of-modular-software-design

- *Parent* and *child* relationships do not usually exist among modules, modules with *mono-dependecies* are called a submodule.

Coupling

- Coupling is the degree of ***interdependence*** between software modules.
- Different types of couplings are:
  - ***Content***: modules *share* code
  - ***Common***: modules *share* global data.
  - ***External***: modules *share* externally imposed data format
  - ***Control***: one modules *controls* the flow of others
  - ***Stamp***: modules use only parts of their inputs
  - ***Data***: modules only share *data*

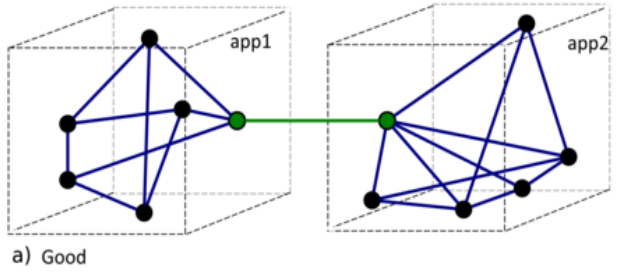


https://en.wikipedia.org/wiki/Coupling_(computer_programming)#Types_of_coupling

- *Temporal* coupling is another *bad* coupling when different functions are coupled in a module since they occur on the same time.

Cohesion

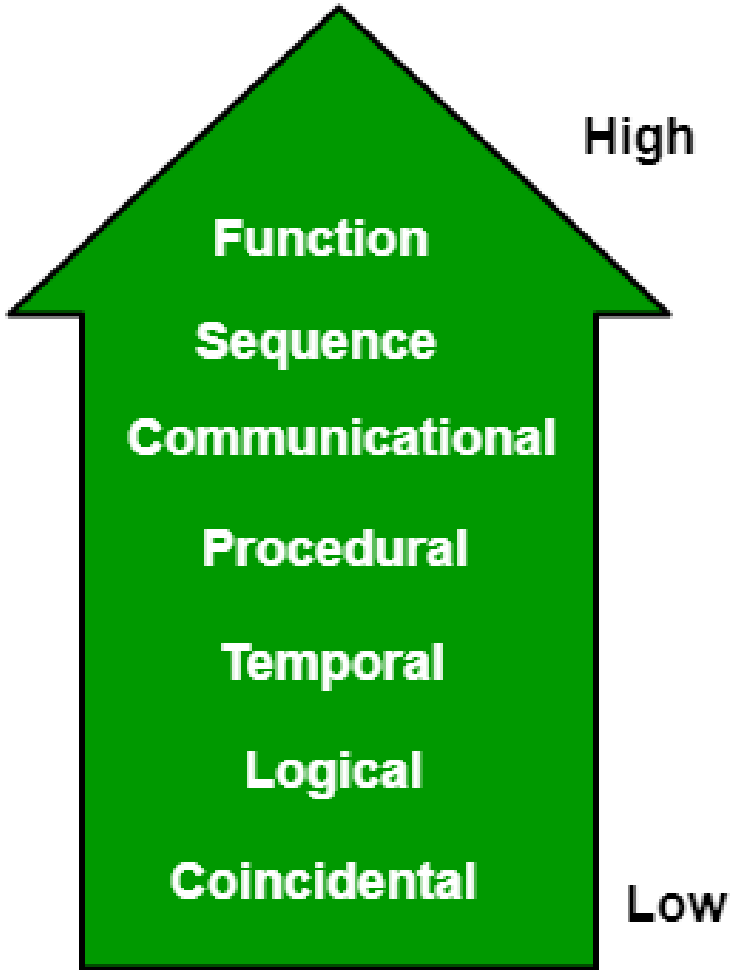
- Cohesion is the degree to which the elements inside a module belong together.
- Different types of cohesions are:
  - ***Coincidental***: modules' functionalities are grouped *arbitrarily*
  - ***Logical***: modules' functionalities are the same *logical* but not *functional*
  - ***Temporal***: functionalities processing at the same are grouped in a module
  - ***Communicational***: functionalities working on the same data are grouped together
  - ***Sequential***: when the output of one part of a module is the input to the other
  - ***Functional***: modules' functionalities contribute to a single well-defined task

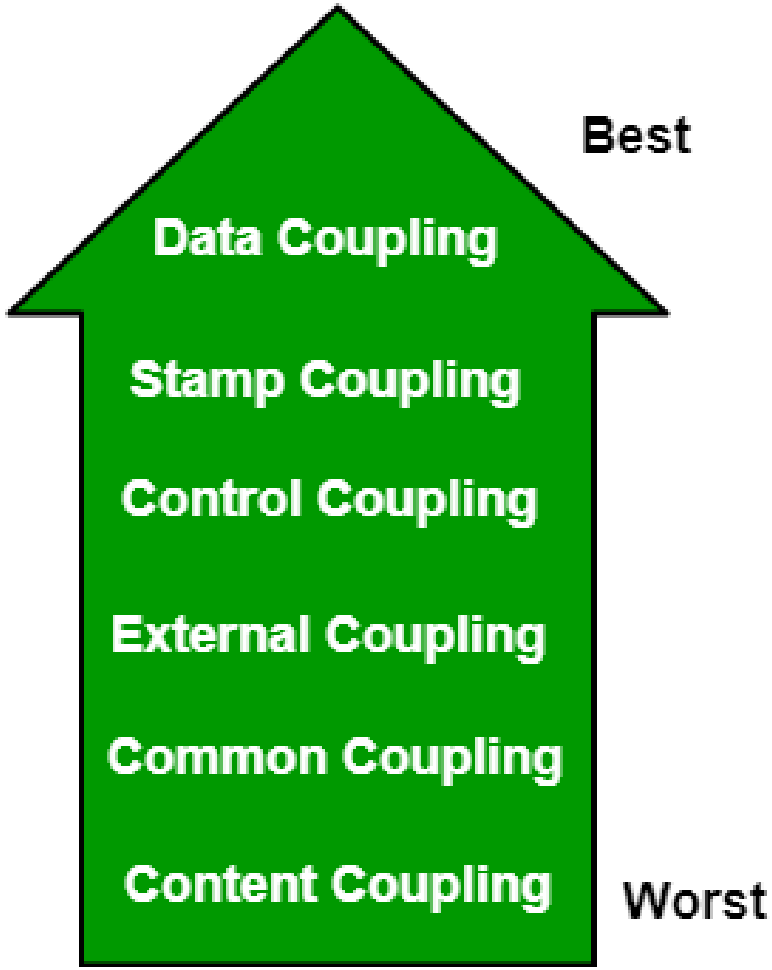


https://en.wikipedia.org/wiki/Cohesion_(computer_science)#High_cohesion

- *High* cohesion reduces complexity, increases maintainability and usability.
- *Atomic* cohesion which a module with single functionality, is the *best* cohesion, yet hard to achieve. Since either the single function is very complex, or too narrow which needs to be coupled with other modules.

Coupling vs Cohesion



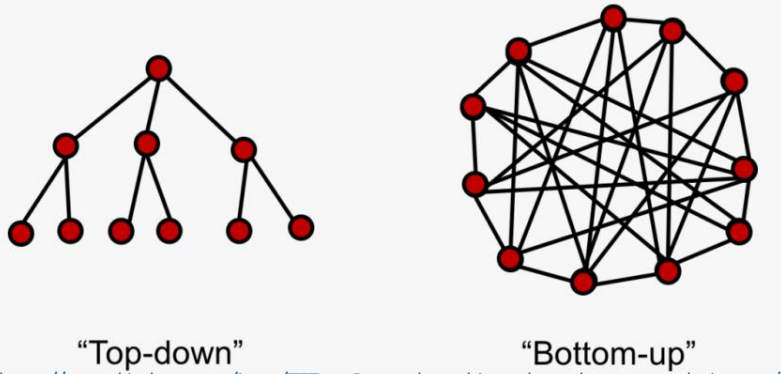


- Systems with high cohesion and low coupling are considered good designed.

https://www.geeksforgeeks.org/software-engineering-coupling-and-cohesion/

Top Down vs Bottom Up

Top-Down	Buttom-Up
Emphasizes on planning and a complete understanding of the system	Emphasizes on early implementation and testing
Stubs to be used for modules yet to be implemented which delays testing ultimate functionality	Integration of all the modules could be concerning, yet code re-use could be increased



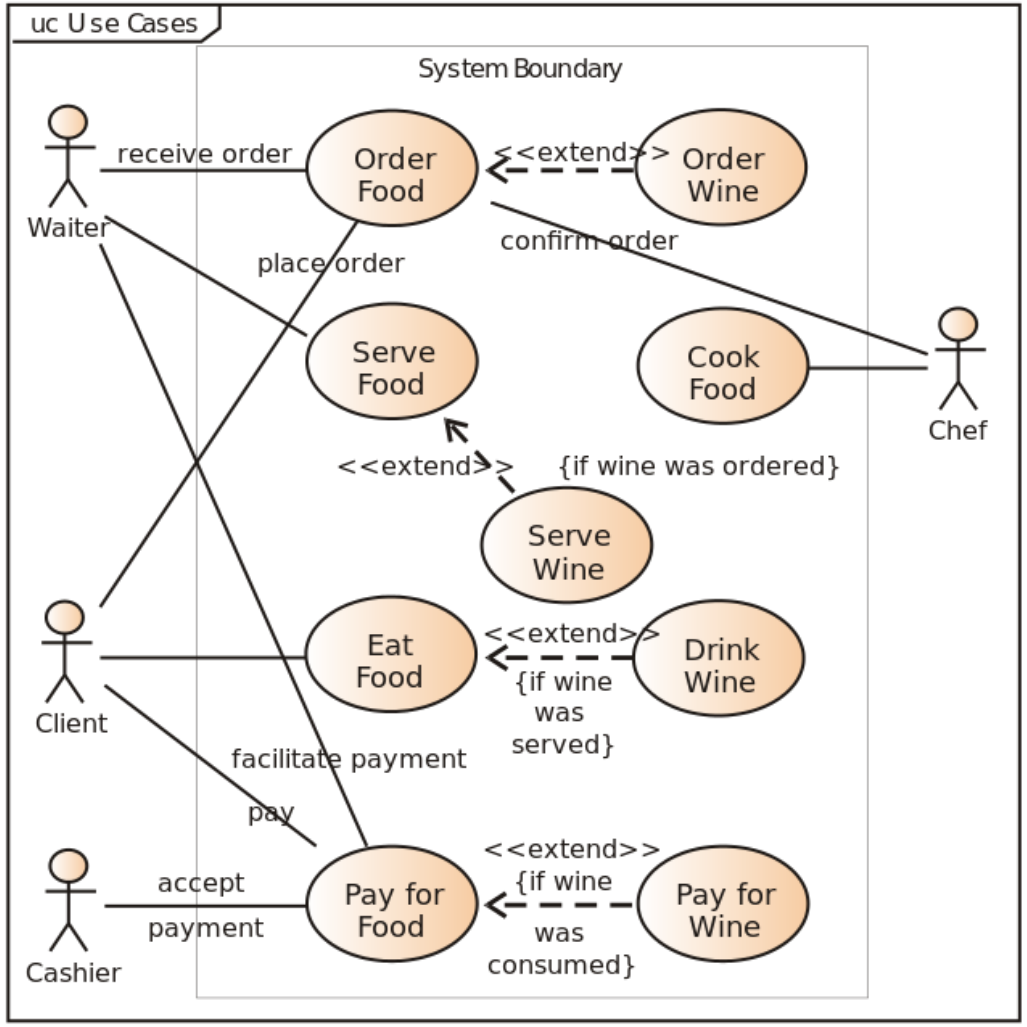
https://www.kindpng.com/imgv/TTTwmR_top-down-hierarchy-vs-bottom-up-design-top/

Modern software design approaches usually combine both top-down and bottom-up approaches. Although an understanding of the complete system is usually considered necessary for good design, leading theoretically to a top-down approach, most software projects attempt to make use of existing code to some degree. Pre-existing modules give designs a bottom-up flavor. Some design approaches also use an approach where a partially functional system is designed and coded to completion, and this system is then expanded to fulfill all the requirements for the project.

https://en.wikipedia.org/wiki/Top-down_and_bottom-up_design#Software_development

Use Case

- An abstract high-level illustration of system requirements in *laypeople's term.*
- Each use case provides a *scenario* for interaction with system.



Activity

Actor

Architectral Patterns

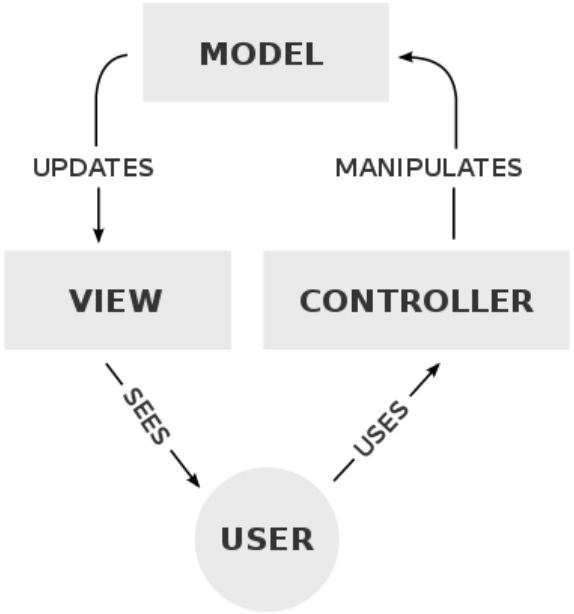
- *"General, reusable solution to a commonly occurring problem – similar to design patterns yet on broader scope."*

- Each pattern satisfy certain requirements, the most appropriate one is the one which satisfies most of the requirements.

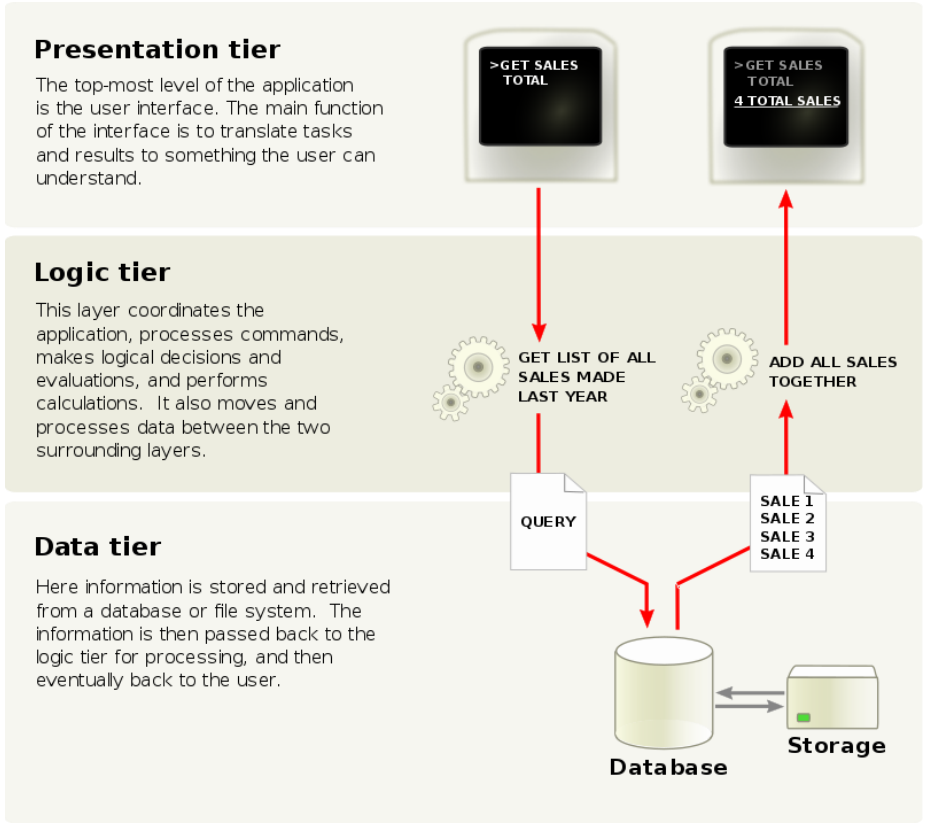
- Blackboard
- Client-server (2-tier, 3-tier, *n*-tier, cloud computing exhibit this style)
- Component-based
- Data-centric
- Event-driven (or implicit invocation)
- Layered (or multilayered architecture)
- Microservices architecture
- Monolithic application
- Model-view-controller (MVC)
- Peer-to-peer (P2P)
- Pipes and filters
- Plug-ins
- Reactive architecture
- Representational state transfer (REST)
- Rule-based
- Service-oriented
- Shared nothing architecture
- Space-based architecture

https://en.wikipedia.org/wiki/Software_architecture#Architectural_styles_and_patterns

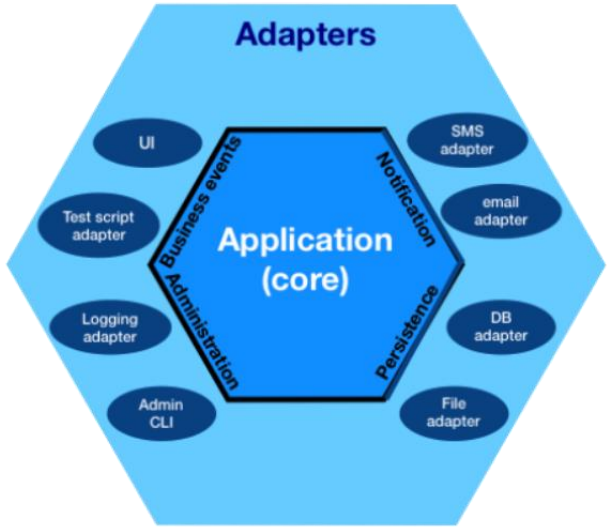
MVC vs 3(n)-tier vs Hexagonal



- *MVC* is mostly used for *user interface* development – vert common in mobile & web app developments.
- Model or the central component is the app's dynamic data structure. View is the presentation only and controller accepts input and converts it into commands for model.



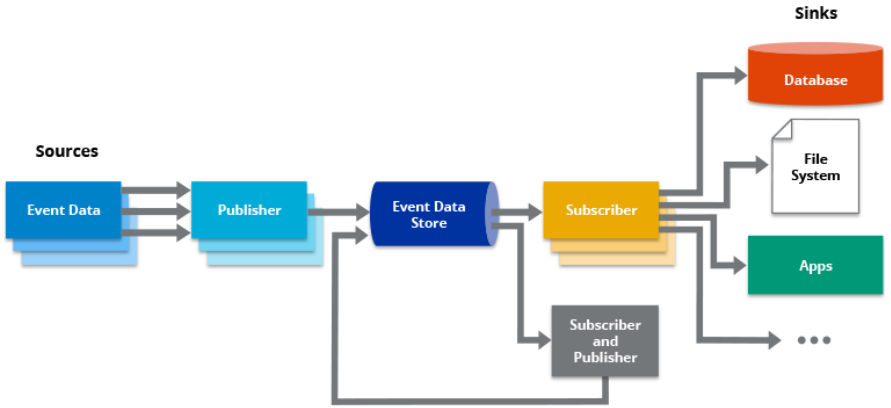
- *Multitier* architecture is a client-server where *presentation* and *logic* layers are physically separated. Generally, it provides flexibility & reusability. *Presentation*, *Application*, *Business*, and *data* layers are the most common in OO design.



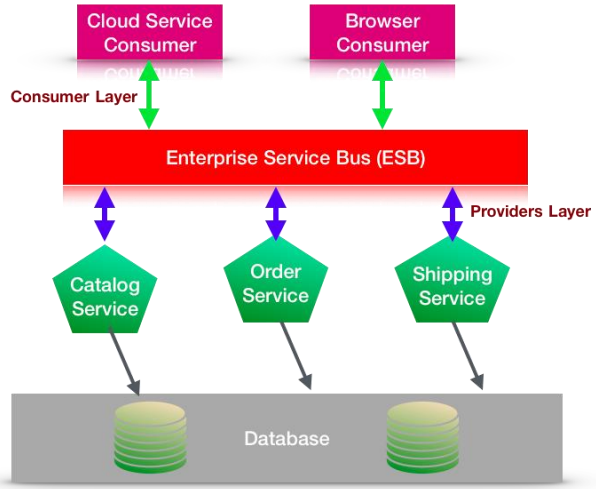
- *Hexagonal* architecture aims at creating *loosely coupled* application components that can be easily connected to their software environment by means of *ports* and *adapters*. Thus components are exchangeable at any level.

SOA vs EDA

SOA	EDA
Clients makes a request	Event Data Store pushes the event
Clients need to know the service API	Producers and consumer are less *coupled*
Clients are known to service provider	Producers has no knowledge of consumers
Communication is bi-directional	Communication is unidirectional



https://hazelcast.com/glossary/event-driven-architecture/



Service-oriented architecture (SOA) pattern

https://goalgorithm.wordpress.com/2019/05/19/software-architecture-patterns-part-2/