Architectural Styles

Съдържание

- Архитектурни Стилове
  - *Pipe and Filter*
  - *Data–Centered*
  - *Layered*
  - *Client–Server*
  - *Model–View–Controller*
  - *Implicit Invocation*
  - *Microservices*

Архитектурен Стил

- ***… определя множество от системи по отношение на модел (pattern) на структурна организация.***

- Всеки архитектурен стил определя:
  - речник на ***компоненти*** и ***конектори***, които могат да се използват в дадена инстанция от този стил.
  - множество от ***ограничения*** за това как те могат да бъдат комбинирани. Те могат да включват ***топологични*** ограничения (пр. без цикли) и ограничения свързани със ***семантика на изпълнението*** (пр. паралелно изпълнение).

source: *An Introduction to Software Architecture, D. Garlan and M. Shaw, 1994*

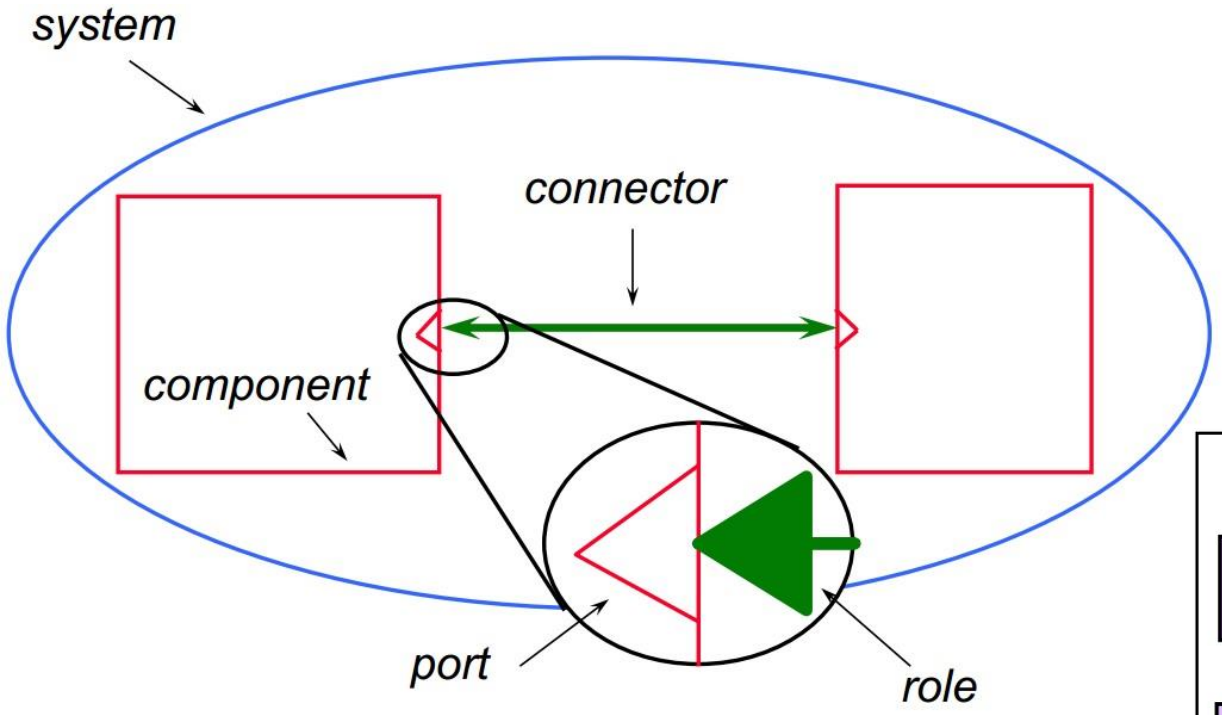
Component, Connector, Topology

- ***Компоненти:*** изчислителните елементи и елементи за съхранение на данни. Комуникацията с другите елементи се осъществява през техните интерфейси (***портове***).

- ***Конектори:*** архитектурните елементи за комуникацията. Подобно на компонентите комуникират с други елементи през техните интерфейси (***роли***).

- ***Топология:*** Успешна комуникация между 2 архитектурни елемента се случва когато *една роля се прикачи към един порт* и техните интерфейси са *съвместими*. Архитектурните елементи, техните взаимовръзки и ограничения, представляват ***топологията***.

Component, Connector, Topology



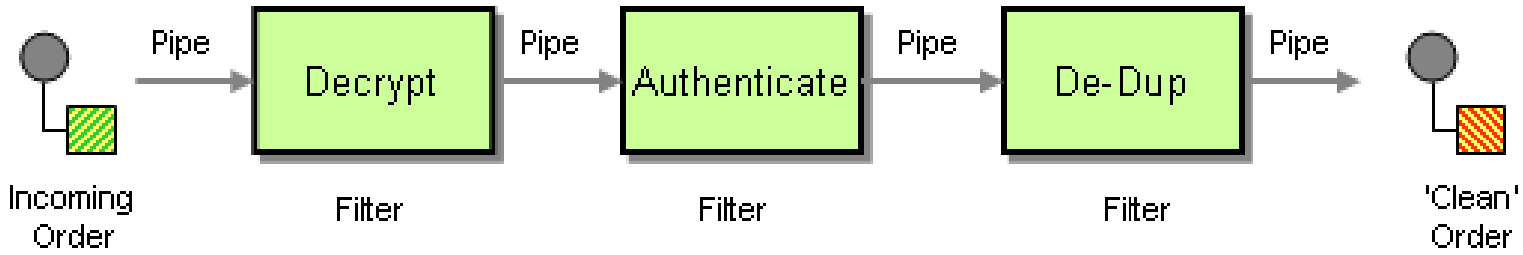
Текстово и графично представяне на проста система.

```
System simple-cs = {

  Component client = { port call-rpc; };
  Component server = { port rpc-request; };

  Connector rpc = {
     role client-side;
     role server-side;
  };

  Attachments = {
     client.call-rpc to rpc.client-side;
     server.rpc-request to rpc.server-side;
  }
}
```

Pipe and Filter

- В този архитектурен стил компонентите (***filters***) обработват данните които получават и ги пращат към следващия компонент.

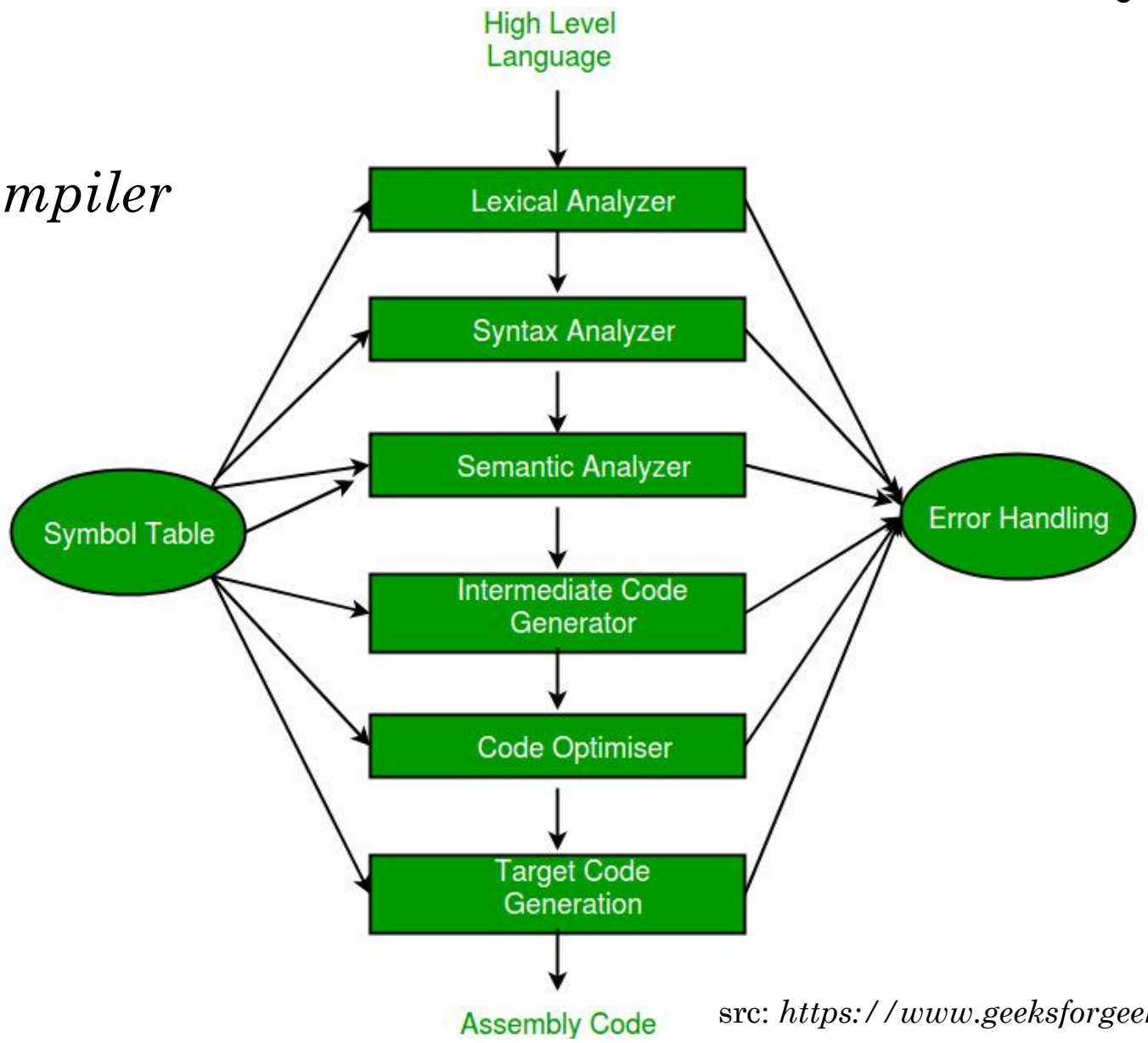
- Конекторите (***pipes***) представляват комуникационните механизми между компонентите.



img: *https://www.enterpriseintegrationpatterns.com/patterns/messaging/PipesAndFilters.html*

Pipe & Filter – Example

Phases of a compiler



src: *https://www.geeksforgeeks.org/phases-of-a-compiler/*

Pipe and Filter

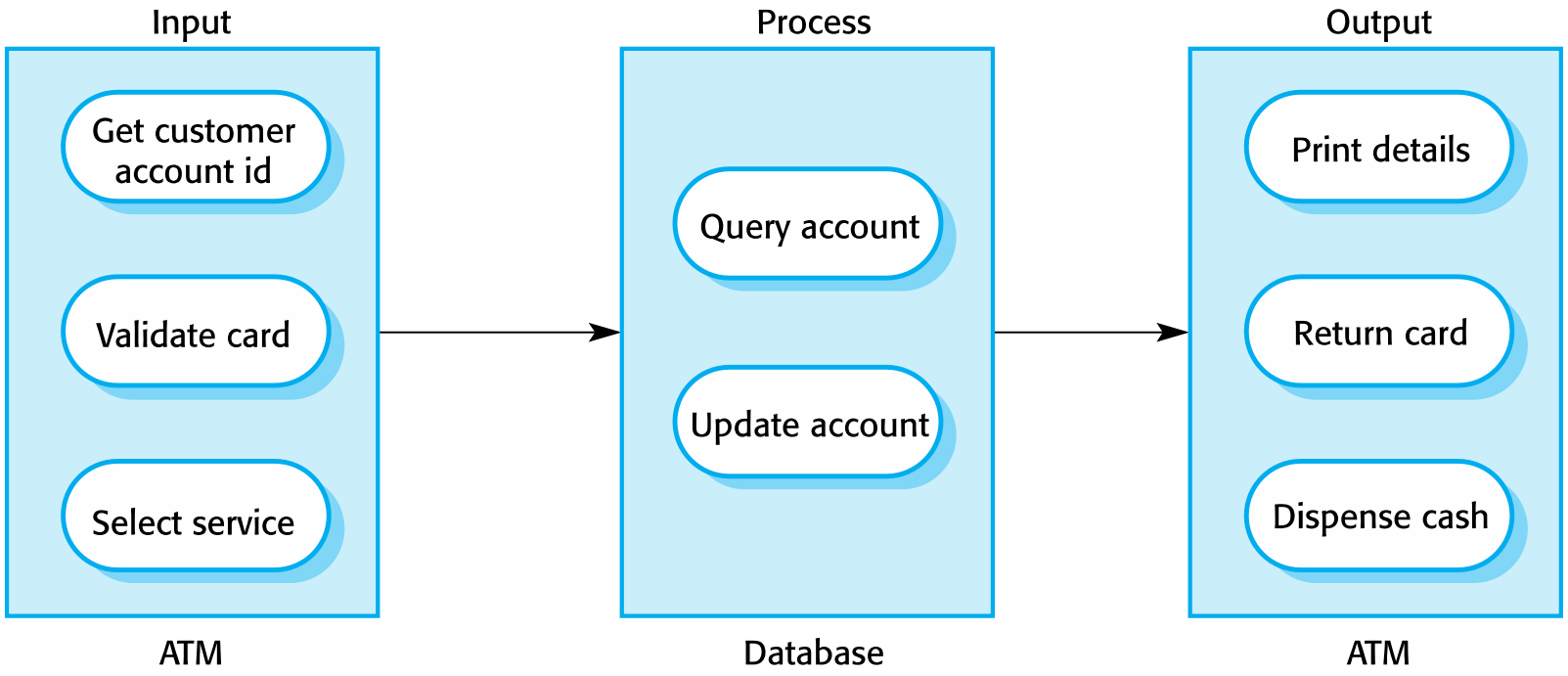
- Обработката на данни е организирана така, че всеки обработващ компонент (***filter***) е *отделен* и извършва един вид трансформация на данни.

- Компонентите *нямат* информация относно състоянието на другите компоненти-филтри.

- Предаването на данни се осъществява (като в тръба – ***pipe***) от един компонент към следващия за обработка.

- Обработката на данни може да започне веднага след получаването на първия байт от филтъра.

Pipe & Filter – Example (2)



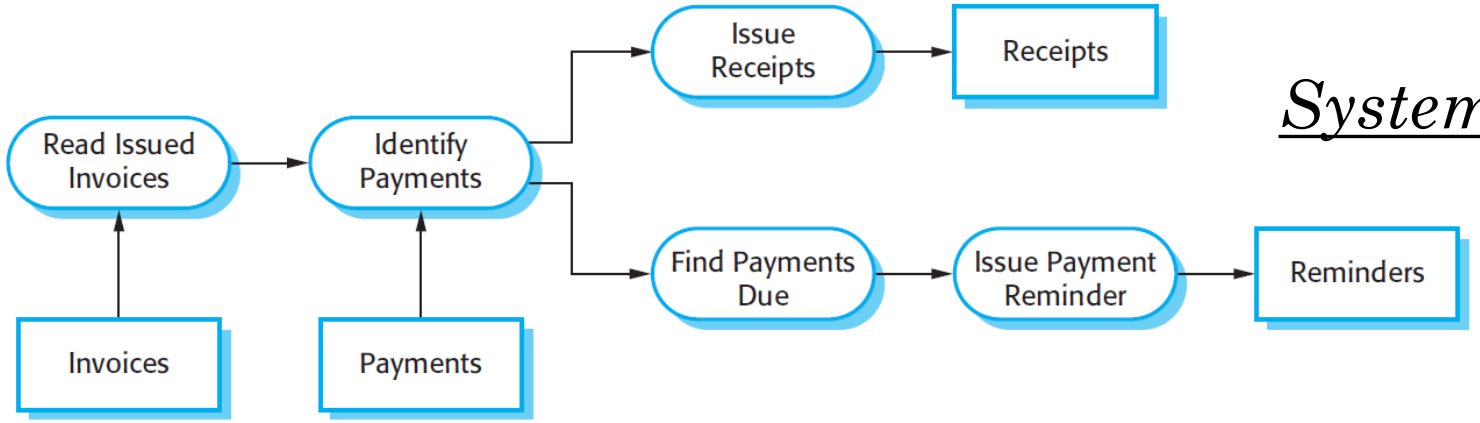
Software architecture of an ATM system.

src: *I. Sommerville, Software Engineering, Pearson.*

Pipe & Filter – Advantages

- ✓ Лесен и прост за композиция (*composition*) и еволюция (*evolution*)
- ✓ Интуитивен и лесен за разбиране
- ✓ Филтрите са дискретни:
  - улесняване на повторната употреба (*reuse*)
  - ниско свързване (*low–coupling*) между компоненти => изменяемост (*modifiability*)
- ✓ Поддръжка на паралелизъм (*concurrency*)

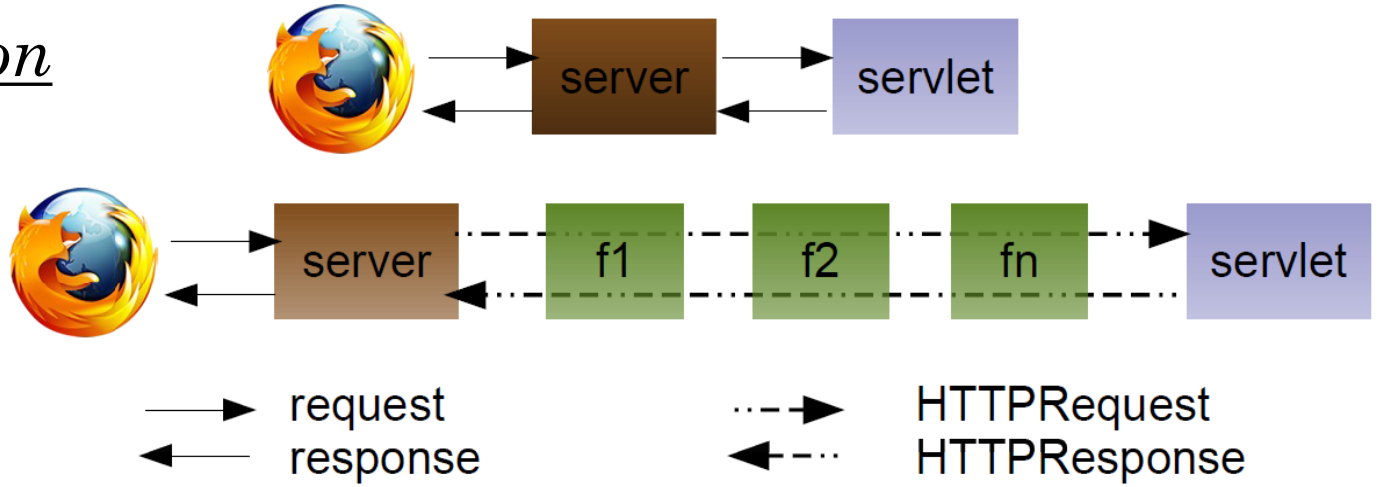
Pipe & Filter - Example (3)



System for processing invoices

src: *I. Sommerville, Software Engineering, Pearson.*

Typed pipes for communication



src: *An Introduction to Software Architecture (slides), D. Garlan and M. Shaw, 1994*

Pipe & Filter – Disadvantages

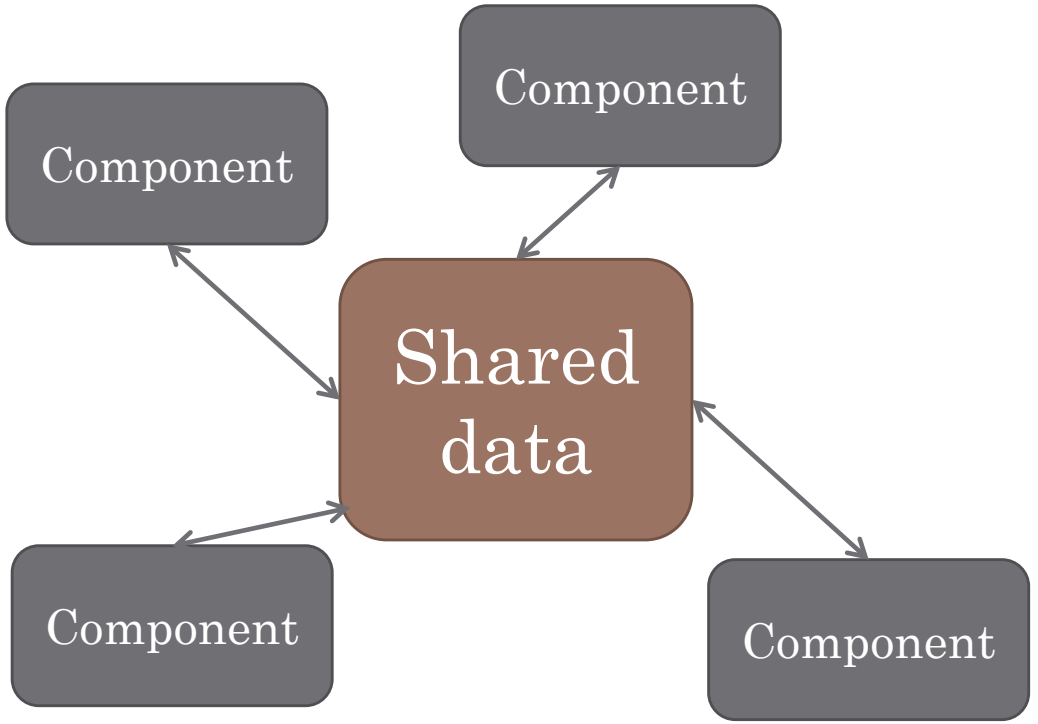
- Всеки филтър трябва да анализира данните си.
- Трудно споделяне на глобални данни.
- Проблеми при повторна употреба – пр. функционални трансформации с несъвместими данни.
- Форматът за предаване на данни трябва да бъде съгласуван.
- Не предоставя начин филтрите да си взаимодействат съвместно, за да решат проблем.

Data-Centered

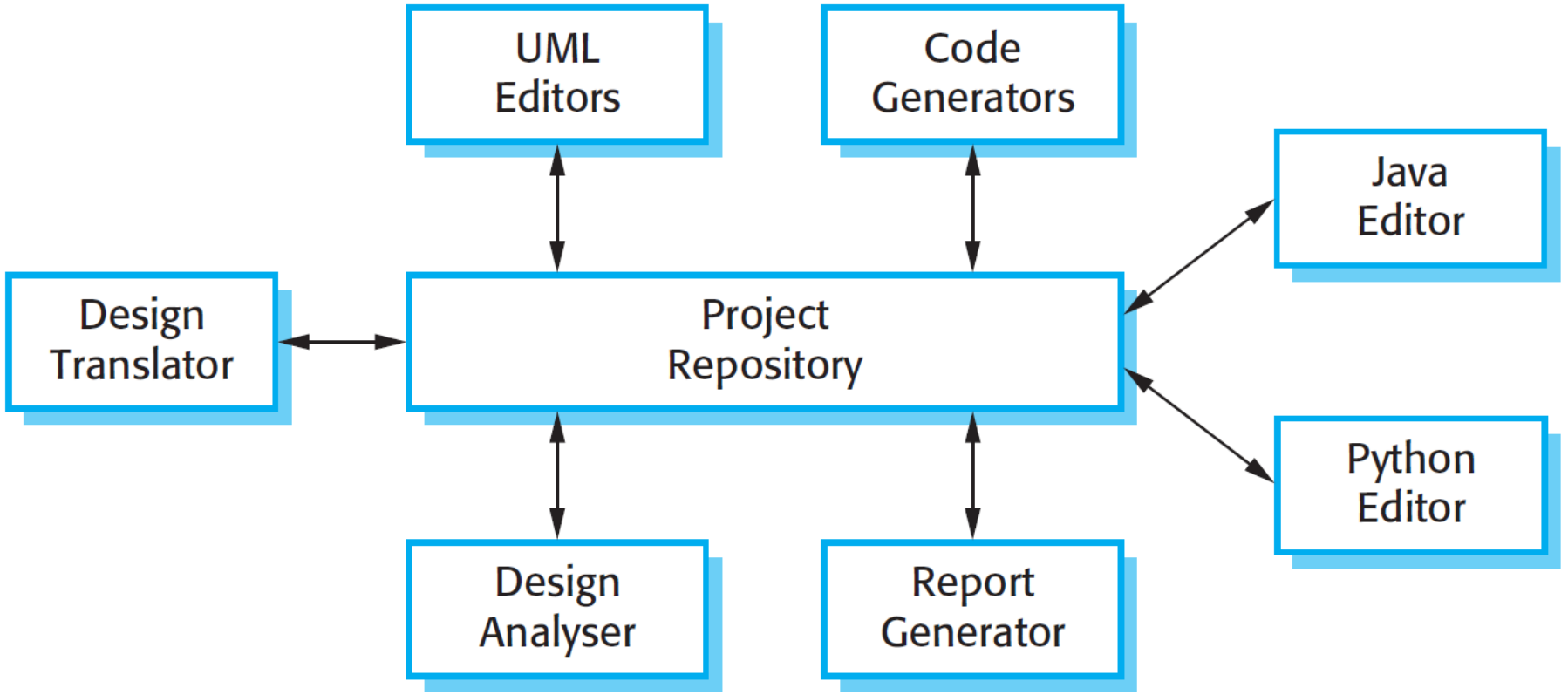
Data-Centered архитектурата се състои от различни компоненти, които комуникират чрез споделено хранилище на данни.

Данните са централизирани и всички компоненти могат да ги достъпват / модифицират.

Споделените данни могат да се разглеждат като конектор между компонентите.



Data-Centered - Example



A repository architecture for an IDE

src: *I. Sommerville, Software Engineering, Pearson.*

Data-Centered

- Има два типа компоненти:
  - Централна структура на данни или хранилище на данни, което е отговорно за осигуряването на постоянно съхранение на данни. Той представлява текущото състояние (state).
  - Множество от независими компоненти, които работят с централното хранилище на данни, извършват изчисления и може да върнат резултатите.
- Взаимодействията или комуникацията между компонентите се осъществяват само чрез хранилището на данни (може да се разглежда като конектор).
- Данните са единственото средство за комуникация между клиентите. Потокът на управлението (flow of control) диференцира архитектурата в две категории: ***Repository*** и ***Blackboard***.

Data-Centered

Repository

- Централната структура на данните е пасивна и клиентите (компоненти) на хранилището на данни са активни.
- Участващите компоненти проверяват хранилището на данни за промени.
- Не се изпращат известия до компонентите.

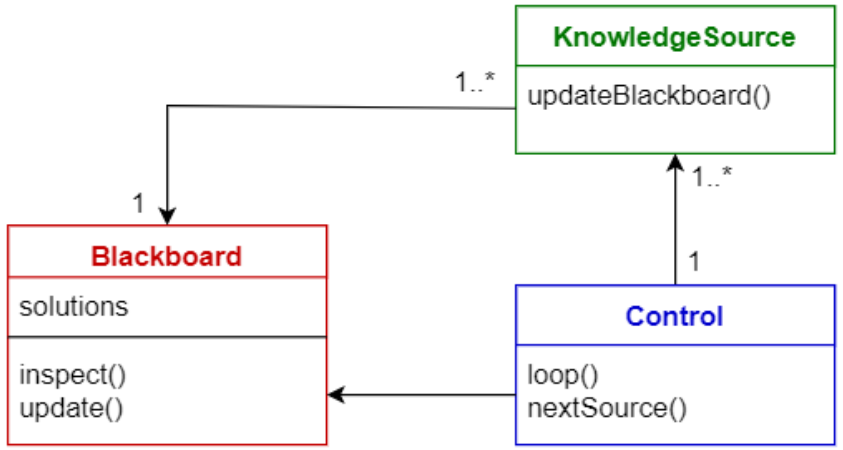
Blackboard

- Централната структура на данните е активна, а клиентите й са пасивни.
- Логичният поток се определя от текущото състояние на данните в хранилището на данни.
- Всички компоненти трябва да бъдат информирани за промените в данните.

Data-Centered – Blackboard

Този стил обикновено се представя с три основни части:

- Източници на знанието (Knowledge Sources – KS): отделни и независими единици. Те решават части от проблема и обобщават частични резултати. Взаимодействието между KS се осъществява единствено чрез “дъската”.
- Структура на данните на “дъската” (Blackboard Data Structure): Данните за състоянието на решаване на проблема. KS правят промени в “дъската”, които постепенно водят до решение на проблема.
- Контрол (Control): Контролът управлява задачите и проверява състоянието на работата.



Data-Centered – Advantages

- Scalability – могат да се добавят нови компоненти
- Concurrency – всички компоненти могат да работят паралелно
- Повторна употреба – компонентите нямат пряка комуникация помежду си
- Централизирано управление на данните
  - По-добри условия за сигурност, архивиране и т.н.
  - Компонентите са независими от производителя на данни

Data-Centered – Disadvantages

- Висока зависимост между структурата на данните на хранилището и компонентите.
- Хранилището – единична точка на отказ (single point of failure).
- Промените в структурата на данните силно влияят на клиентите.
- Проблеми в синхронизацията на множество компоненти.

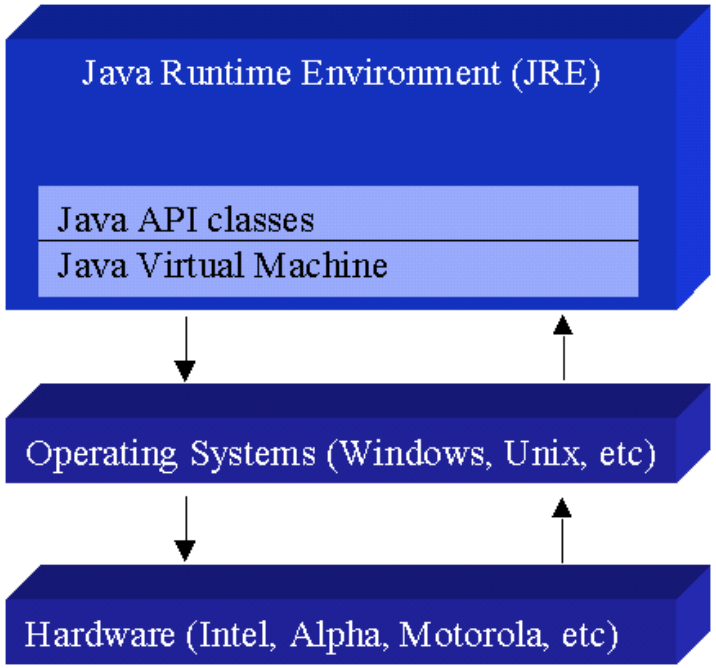
Layered Architecture

Организира системата в слоеве със свързана функционалност между всеки слой.

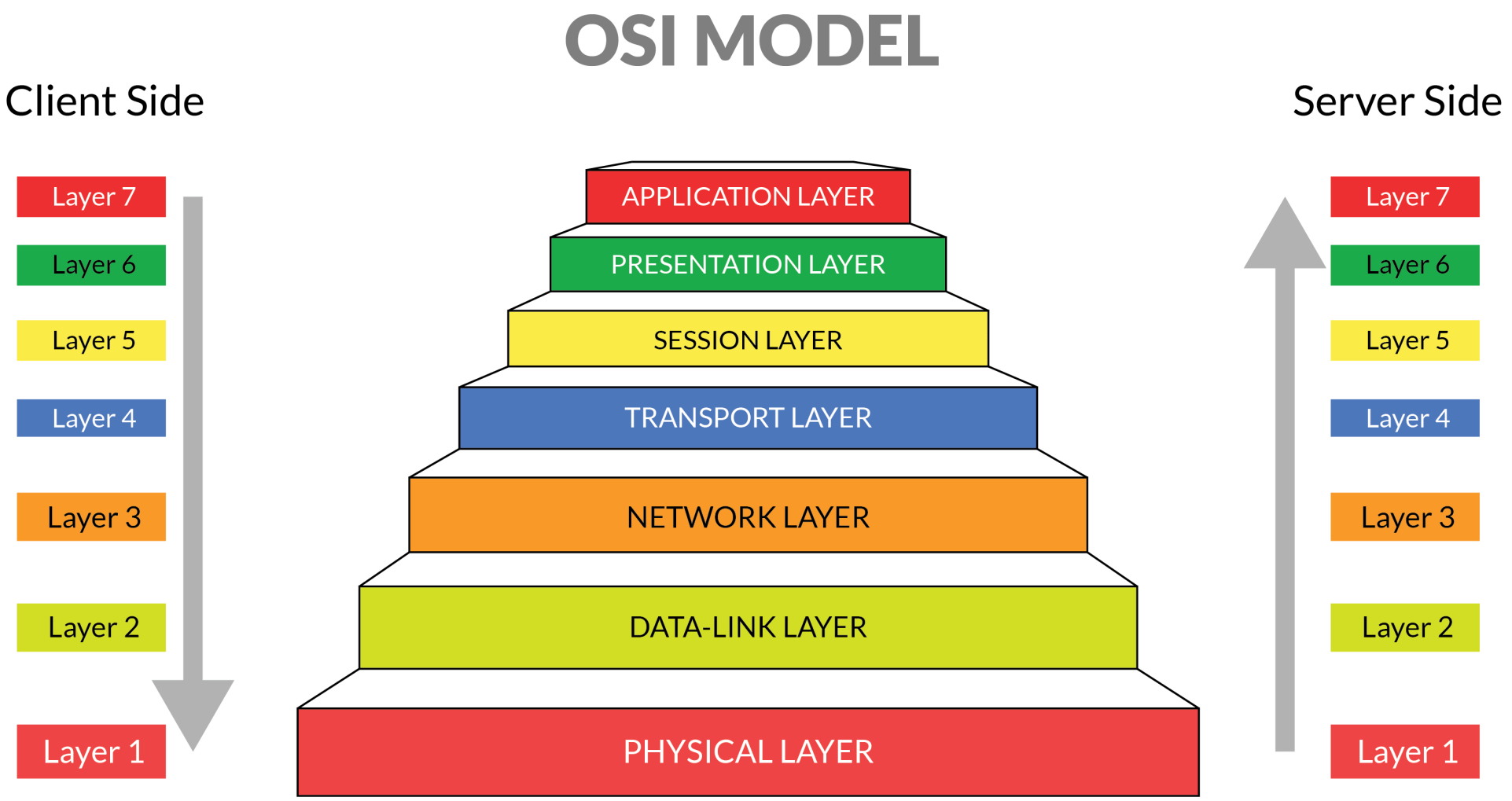
Всеки слой предлага услуги чрез интерфейс, но само за слоя, който е директно над него и използва услугите от слоя, който е директно под него.

По този начин всеки слой представлява:

- Сървър - за слоя отгоре
- Клиент - за слоя отдолу



Layered Architecture - Example

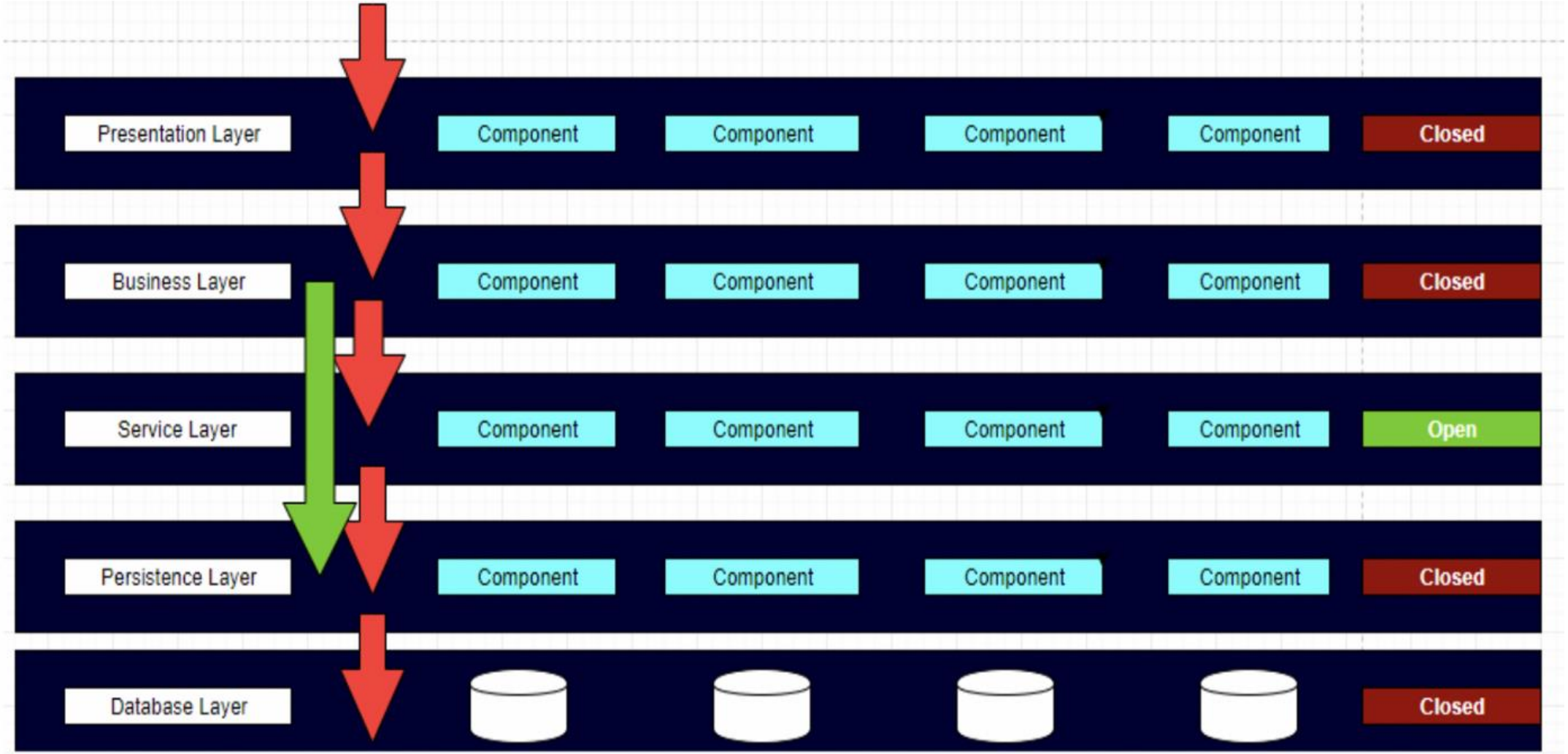


src: *http://www.bytesofgigabytes.com/networking/osi-model/*

Layered Arch. – Advantages

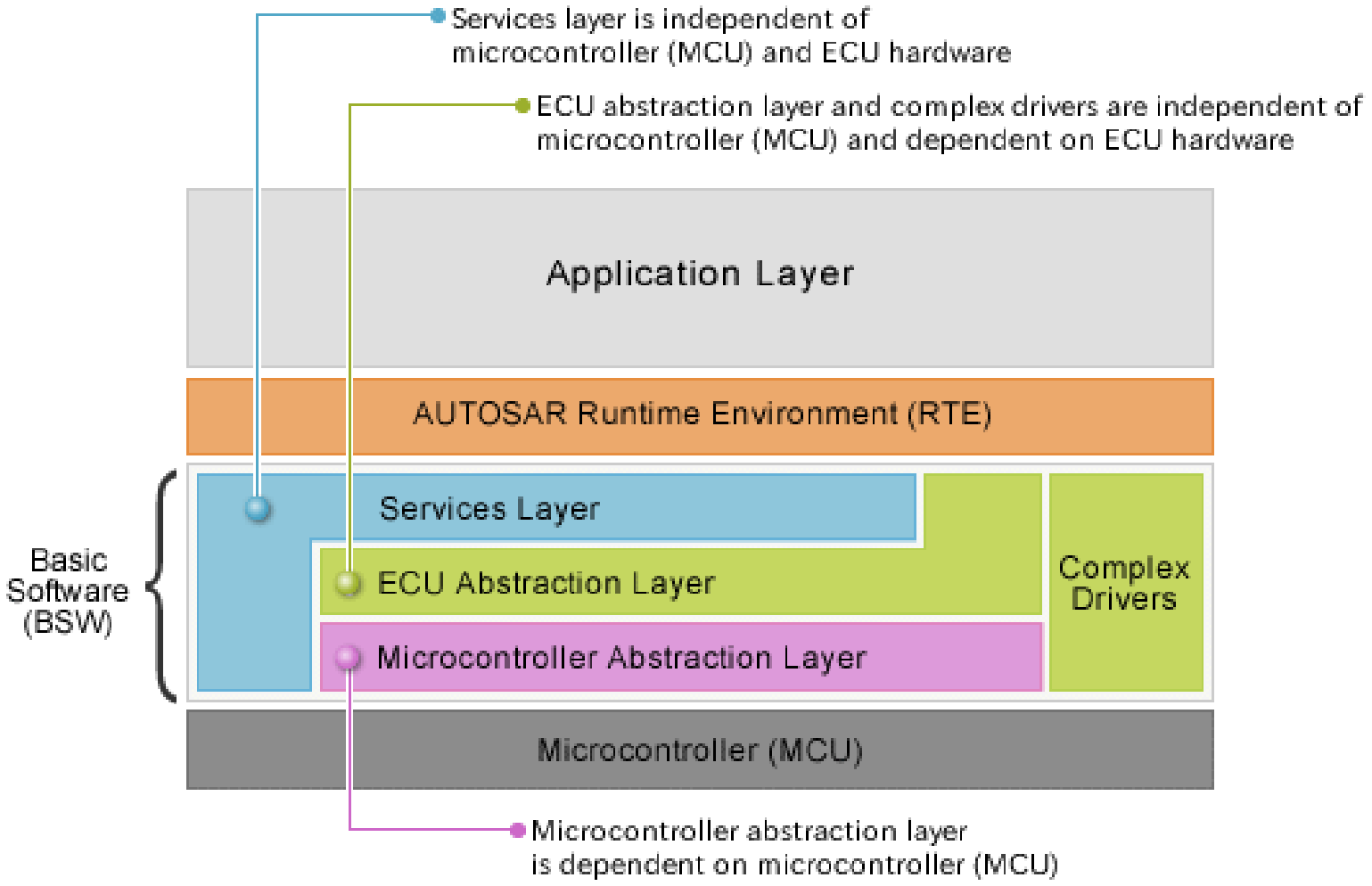
- Всеки слой поддържа подобни задачи (по-добра кохезия)
- Абстракция - вътрешната структура на слоевете е скрита
- Позволява подмяна на цели слоеве
- Допълнителни услуги (например удостоверяване) могат да бъдат осигурени във всеки слой, за да се увеличи надеждността на системата

Layered Architecture – Example (2)



src: *https://medium.com/@priyalwalpita/software-architecture-patterns-layered-architecture-a3b89b71a057*

Layered Architecture - Example (3)



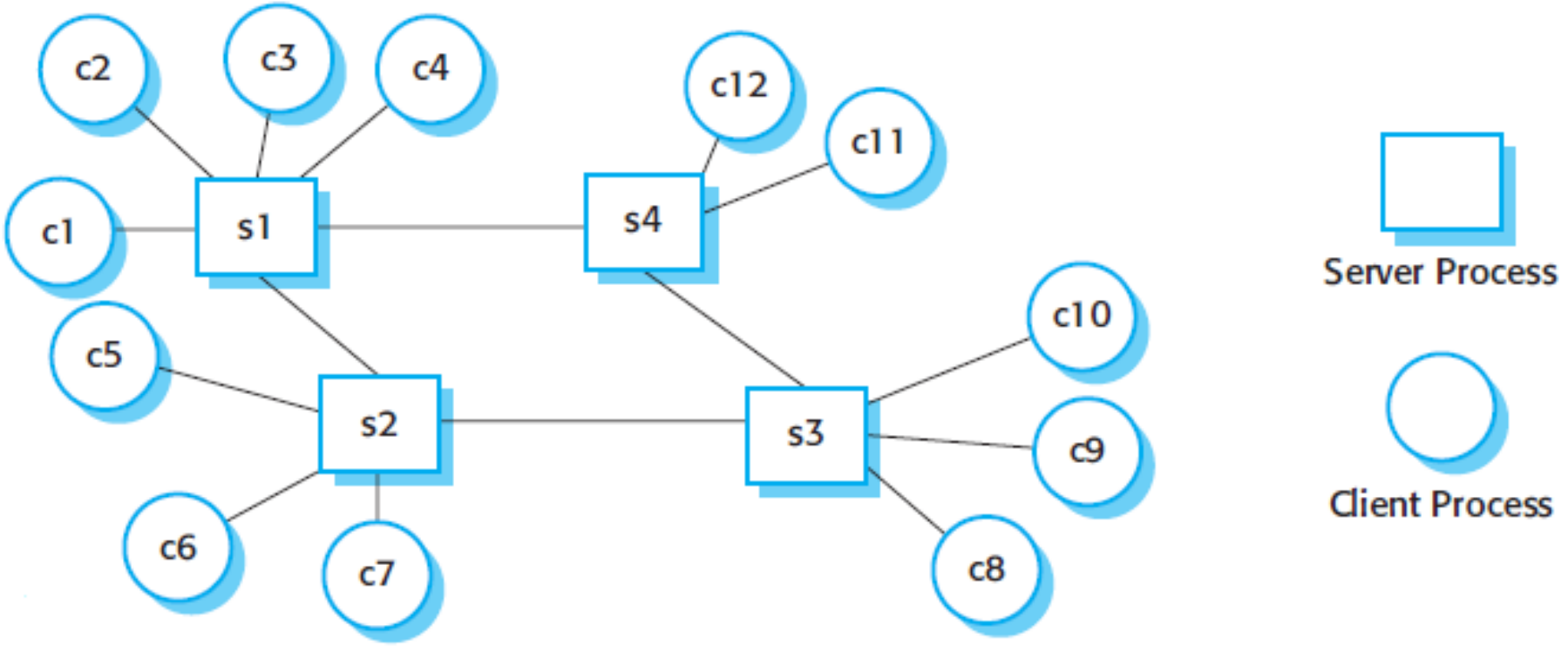
src: *http://www.renesas.eu/applications/automotive/technology/autosar/peer/autosar_layerdarch.jsp*

Layered Arch. – Disadvantages

- Осигуряването на чисто разделяне между слоевете често е трудно - слой от високо ниво може да се наложи да взаимодейства директно с по-ниско ниво.

- Производителността може да бъде проблем поради множество нива на интерпретация на заявка за услуга, тъй като тя се обработва на всеки слой.

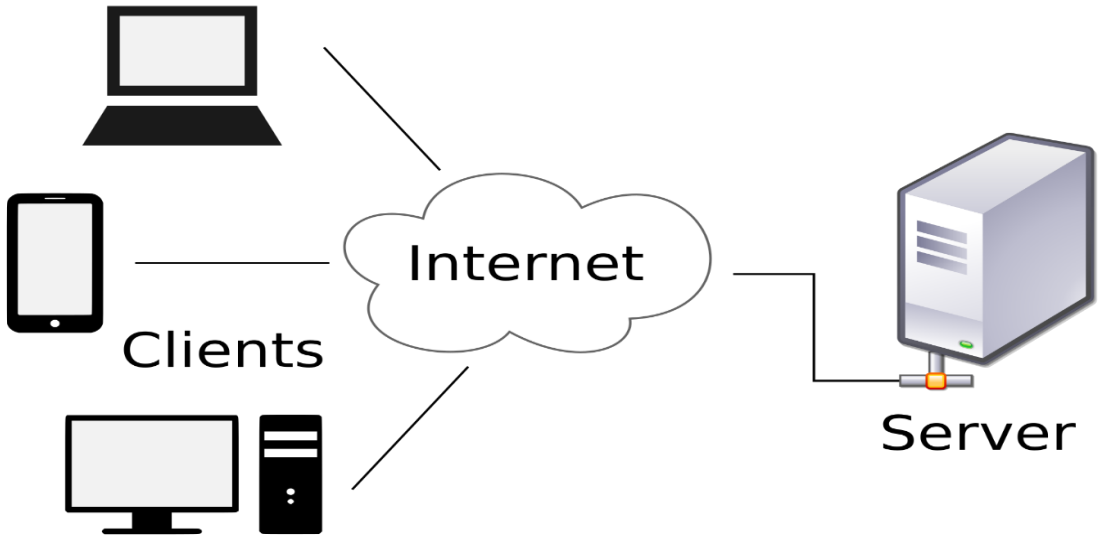
Client-Server



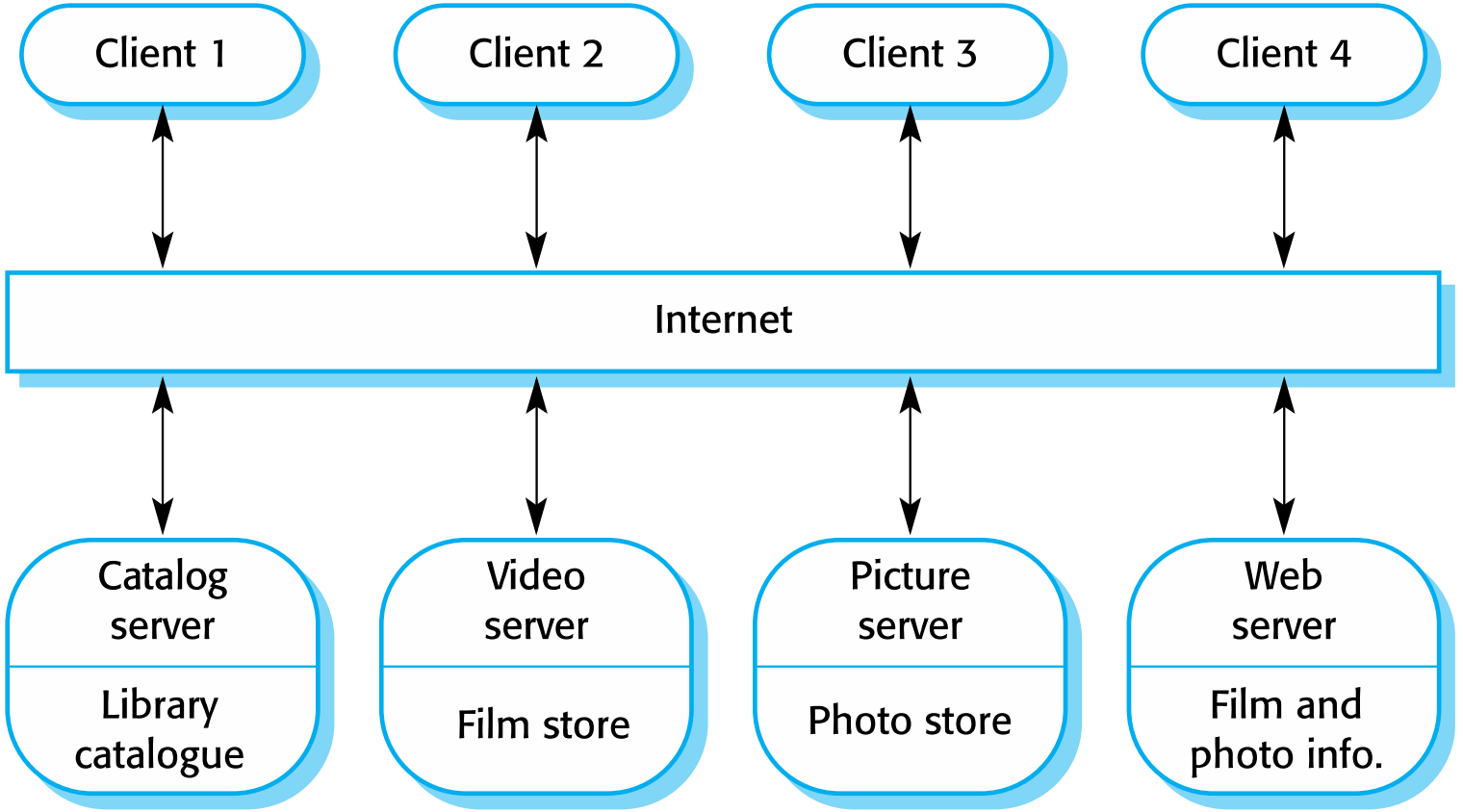
src: *I. Sommerville, Software Engineering, Pearson.*

Client-Server

- При Клиент-Сървър функционалността на системата е организирана в услуги, които се предоставят от сървъри.
- Клиентите са потребители на тези услуги и достъпват сървърите за да се възползват от тях.
- Не е необходимо за сървър, да има информация за неговите клиенти



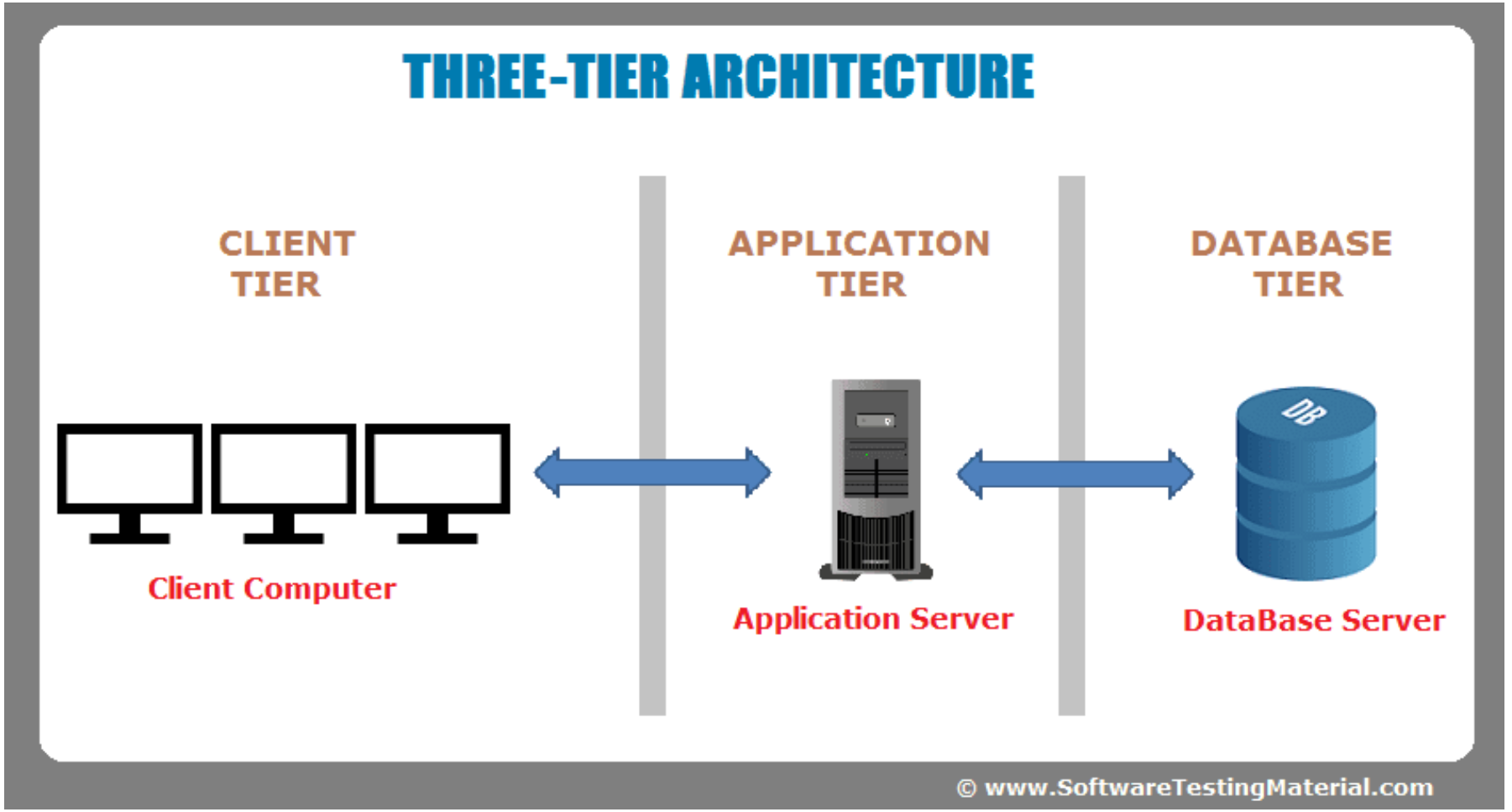
Client-Server - Example



A Client-Server Architecture for a film library

src: *I. Sommerville, Software Engineering, Pearson.*

Client-Server - N-tier



Client-Server - Advantages

- Централизираната му архитектура улеснява защитата на данните.
- Сървърите могат да бъдат разпространени в мрежа.
- Данните се прехвърлят чрез протоколи, които не се интересуват от платформите.
- Обща функционалност (например принтер) може да бъде достъпна за всички клиенти.
- Капацитетът на Клиента и Сървърите може да се променя отделно.

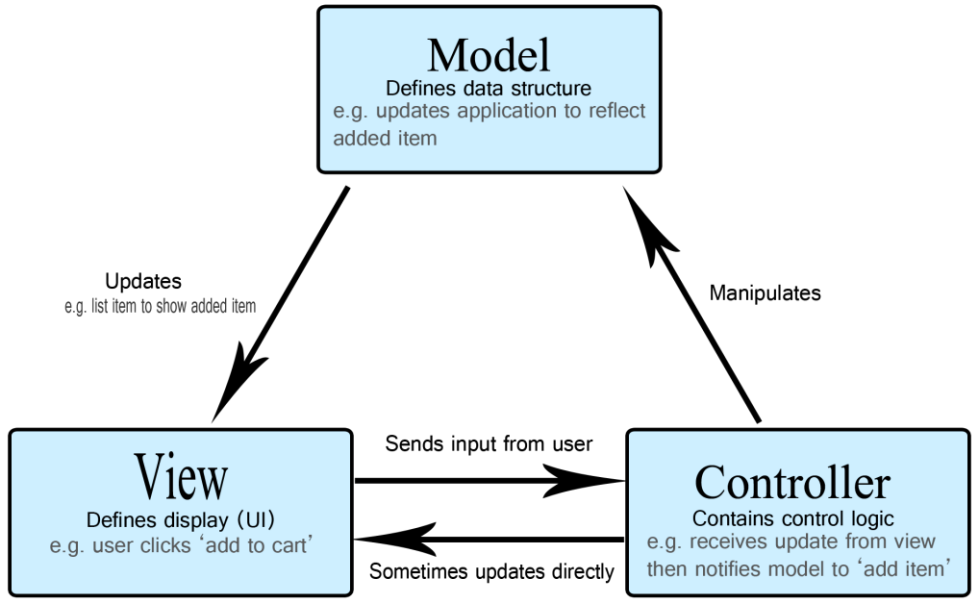
Client-Server - Disadvantages

- Производителността зависи от мрежата, както и от системата.
- Ако твърде много клиенти едновременно изискват данни от сървър, може да бъде претоварен.
- Пакетите с данни може да бъдат променени по време на предаването.
- Може да има проблеми с управлението, ако сървърите са собственост на различни организации.

Model-View-Controller (*MVC*)

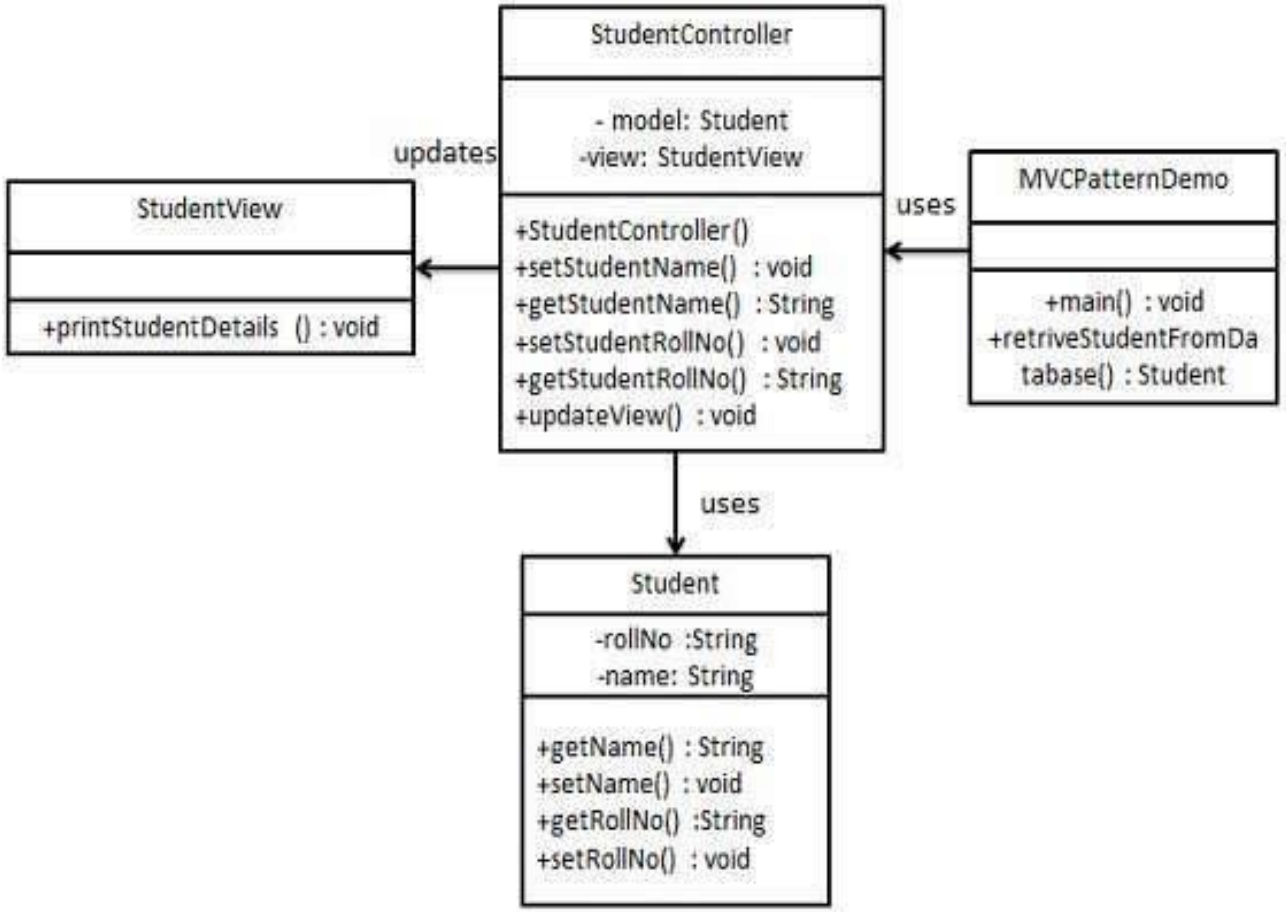
Системата е структурирана в три логически компонента, които си взаимодействат помежду си:

- Компонентът ***Model*** управлява системните данни и свързаните с тях операции.
- Компонентът ***View*** определя и управлява как данните се представят на потребителя.
- Компонентът ***Controller*** управлява взаимодействието с потребителя (пр. натискане на клавиши и т.н.) и предава тези взаимодействия на компонента View или Model.



src: *I. Sommerville, Software Engineering, Pearson.*
img: *https://developer.mozilla.org/en-US/docs/Glossary/MVC*

MVC – Example



src: *https://www.tutorialspoint.com/design_pattern/mvc_pattern.htm*

MVC – Advantages

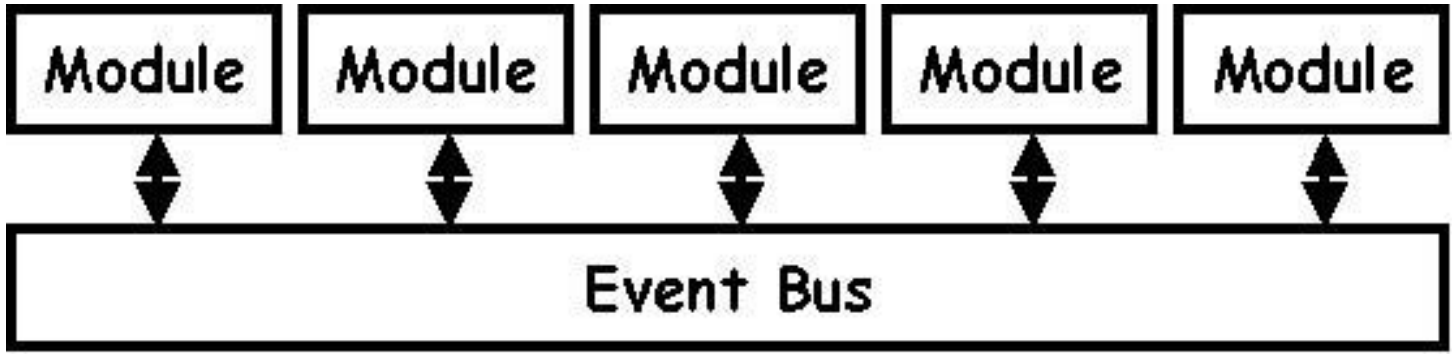
- Голяма гъвкавост
  - Лесен за поддръжка и прилагане на бъдещи подобрения
  - Ясно разделяне между логиката на представяне и бизнес логиката
- Позволява на данните да се променят независимо от тяхното представяне и обратно.
- Множество изгледи за един модел

MVC – Disadvantages

- Дори ако моделът на данни и взаимодействията са прости, този стил може да въведе сложност и да изисква много код.
  - *Не е подходящ за малки приложения.*
- Проблем с производителността при чести актуализации в модела.
- Разработчиците на софтуер, използващи MVC, трябва да са квалифицирани в множество технологии.

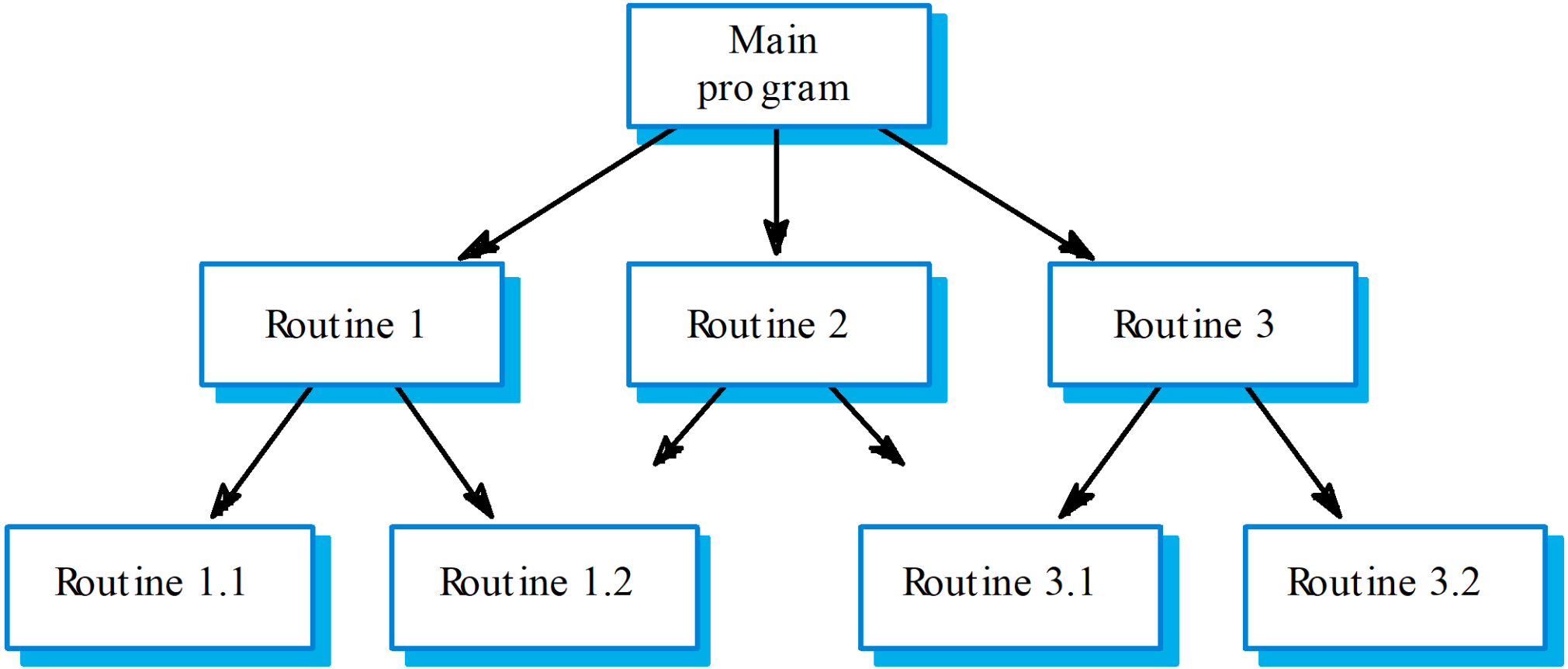
Implicit Invocation

- При неявното извикване, вместо директно извикване на процедура, компонента може да обяви едно или повече събития.
- Други компоненти в системата могат да регистрират интерес към събитие, като свържат процедура със събитието.
- Когато събитието е обявено, системата сама извиква всички процедури, които са били регистрирани за събитието.
- По този начин събитието извиква процедури в други модули.



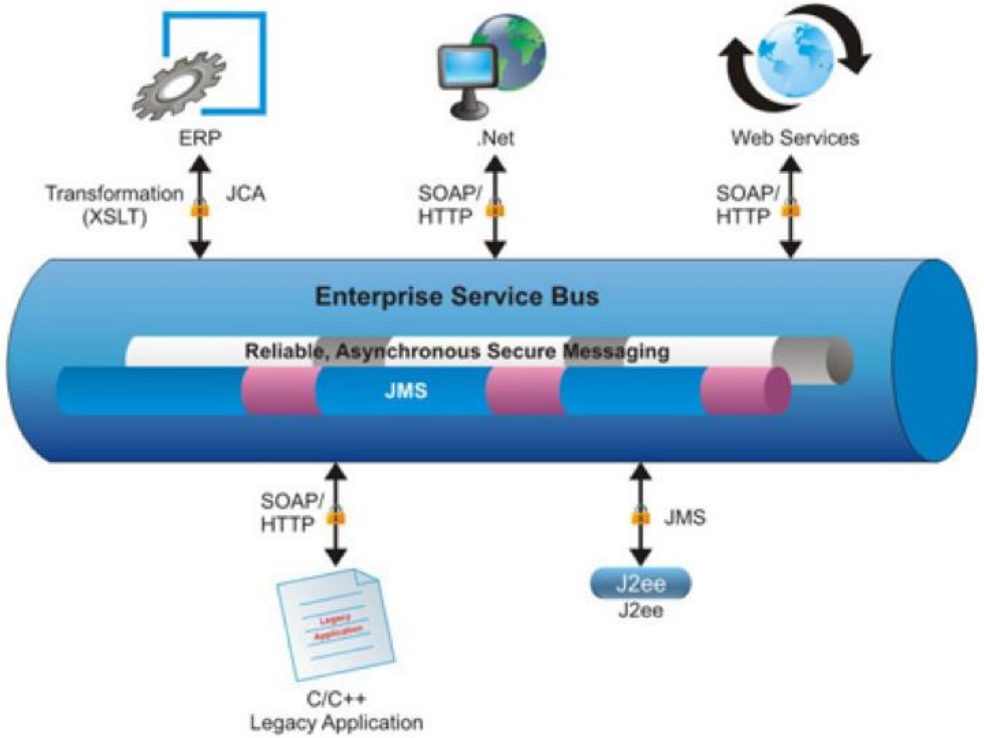
source: *An Introduction to Software Architecture, D. Garlan and M. Shaw, 1994*

Explicit Invocation



Implicit Invocation

- Компонентите в системата взаимодействат помежду си чрез излъчване на събития
- Компонентите работят едновременно и комуникират чрез получаване или излъчване на събития
- Шината за събития може да се разгледа като конектор – Всички компоненти взаимодействат само чрез нея
- Събитията могат да съдържат и данни



img: *https://dzone.com/articles/microservices-versus-esb*

Implicit Invocation – Advantages

- Слаби връзки (loose coupling):
  - Компонентите могат да бъдат много разнородни
  - Компонентите лесно се заменят / добавят / използват повторно

- Голяма ефективност за разпределените системи - събитията са независими и могат да пътуват през мрежата

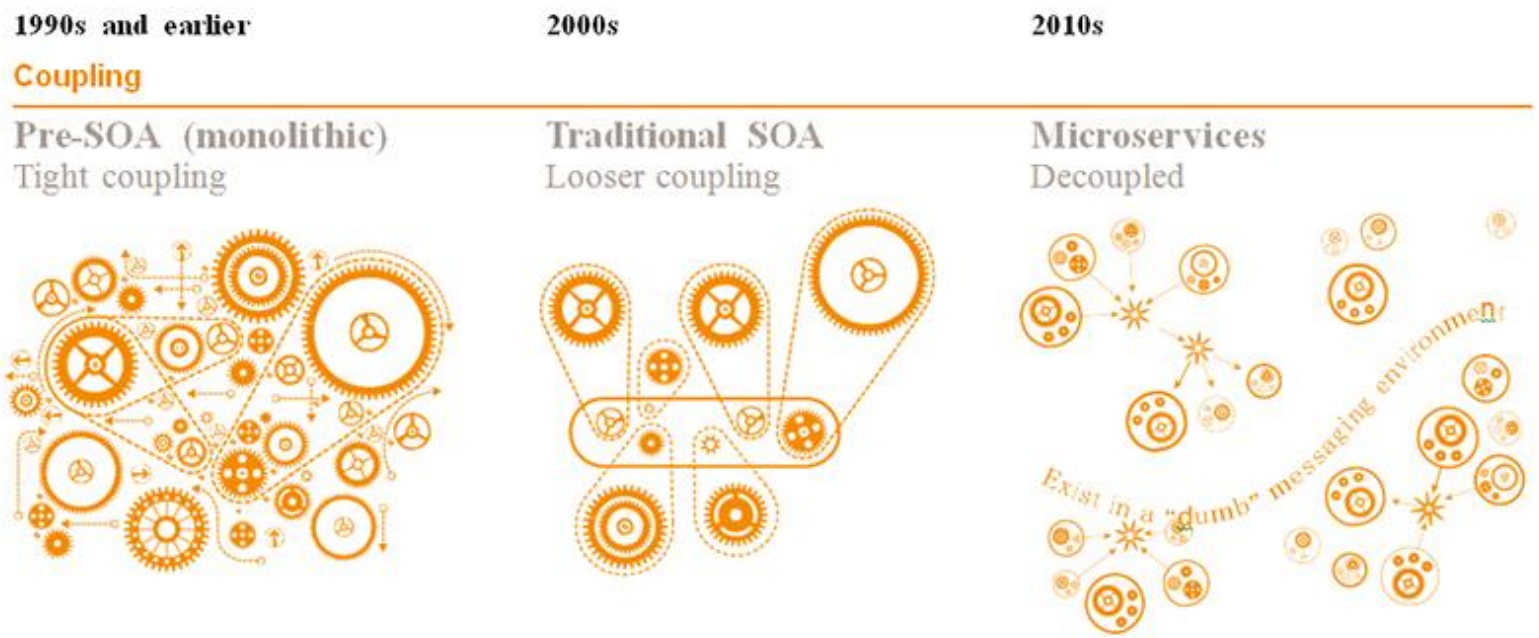
- Сигурност - събитията лесно се проследяват и регистрират

Implicit Invocation – Disadvantages

- Последователността на изпълнение на компоненти е трудно да се контролира
- Не е сигурно дали има компонент, който да реагира на дадено събитие
- Големи количества данни са трудни за пренасяне от събития
- Шина за събития (event bus) – единична точка на отказ (single point of failure)

MicroServices

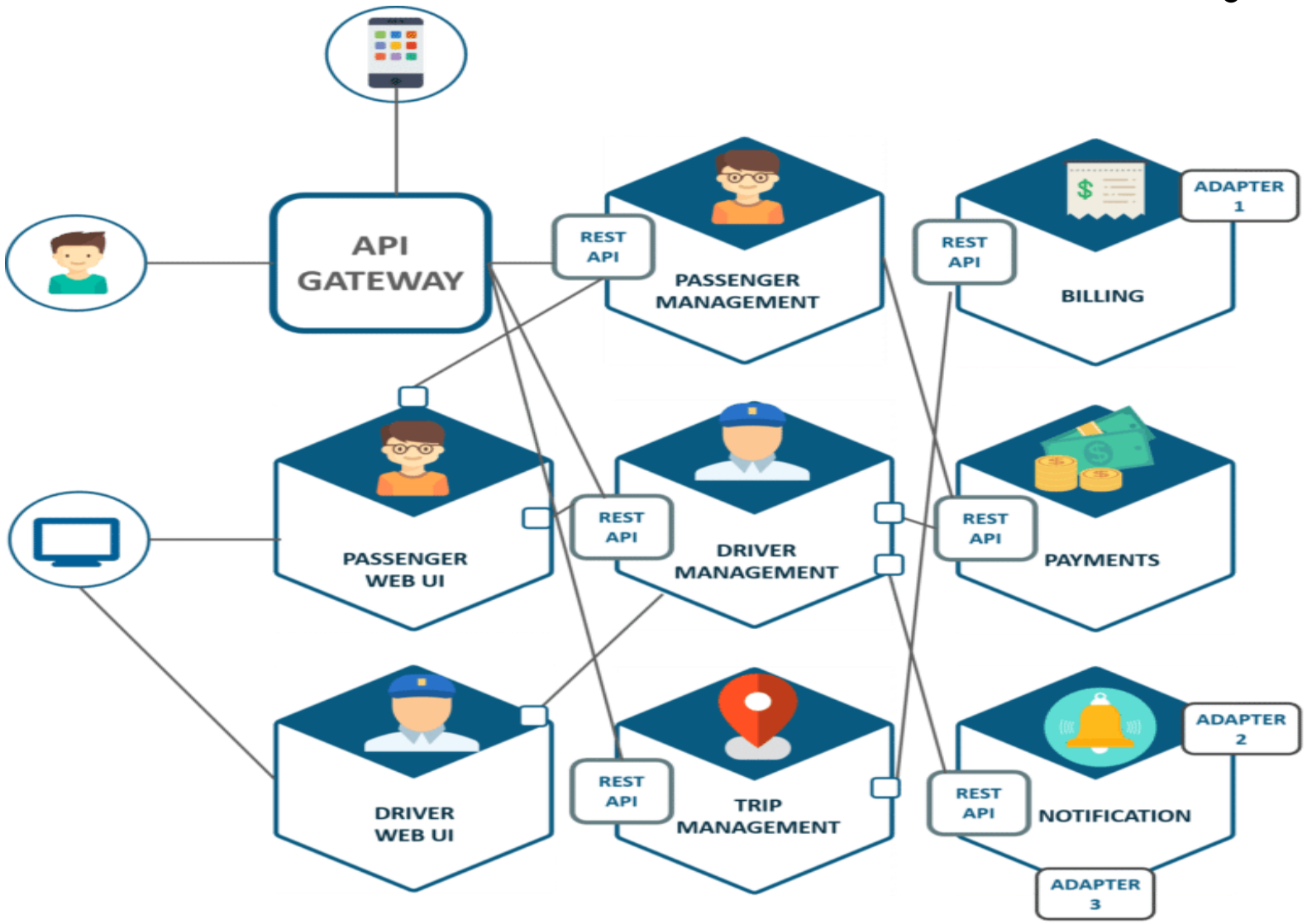
- **Fowler & Lewis** [1] определят архитектурният стил на микроуслугите (микроуслуги) като „*подход към разработването на едно приложение като набор от малки услуги, всяка от които работи в свой собствен процес и общува с леки механизми*.“ .



[1] *M. Fowler, J. Lewis. – http://martinfowler.com/articles/microservices.html*

img: *https://open-organization.com/en/2018/01/04/francais-la-vision-technologique-de-philippe-mourant-le-nouveau-cto-de-presans/microservices/*

MicroServices - Example



src: *https://dzone.com/articles/microservice-architecture-learn-build-and-deploy-a*

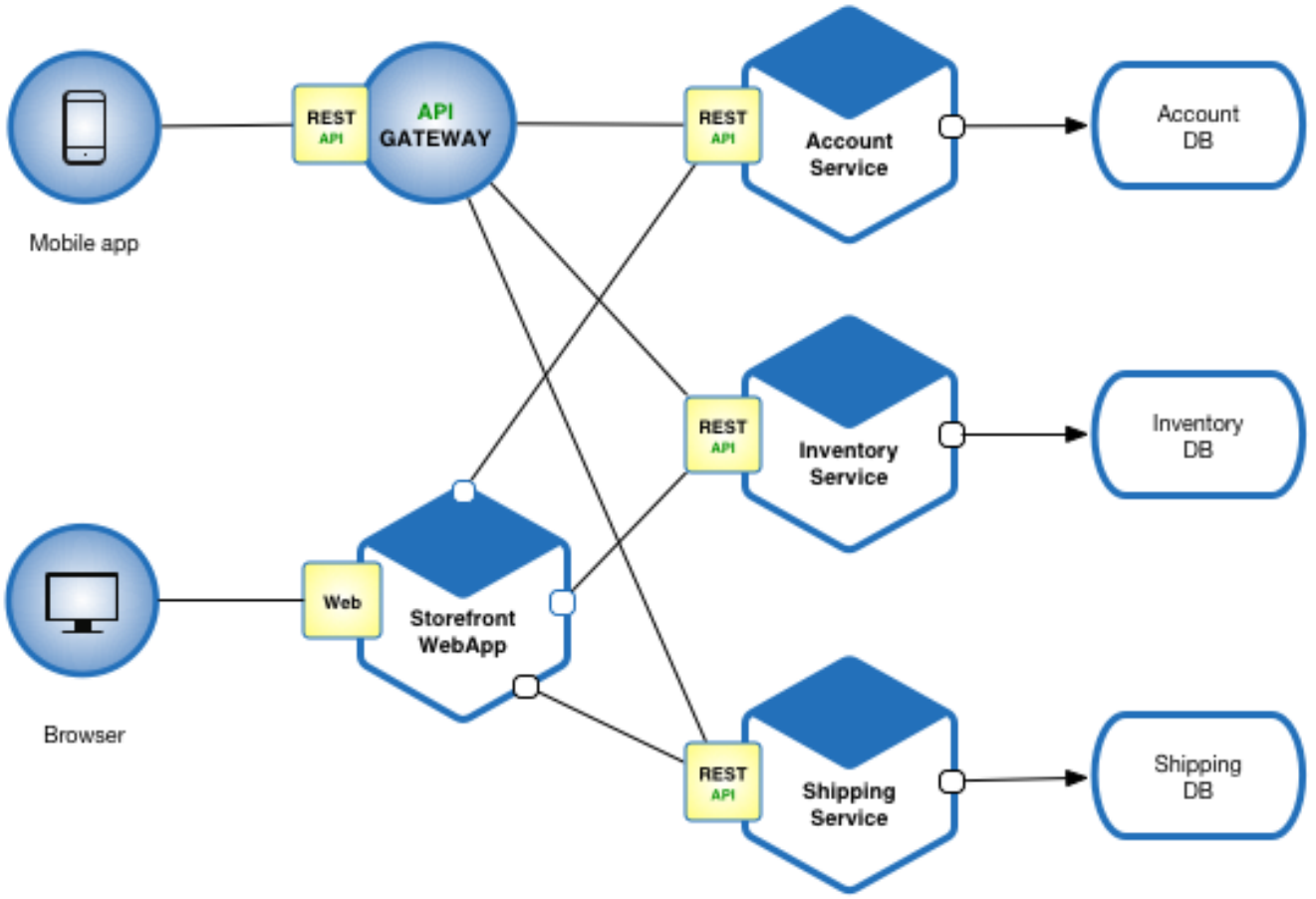
MicroServices - Communication

- Микроуслугите се насочват в приложения където свързването е възможно *най-слабо* (loose coupling) и кохезията е възможно *най-силна* (strong cohesion) – (*smart endpoints and dumb pipes*).

- Два начина са използвани (главно) за комуникация:
  - директна комуникация чрез леки протоколи (например REST) или
  - съобщения/събития през шина за съобщения/събития (message/event bus).

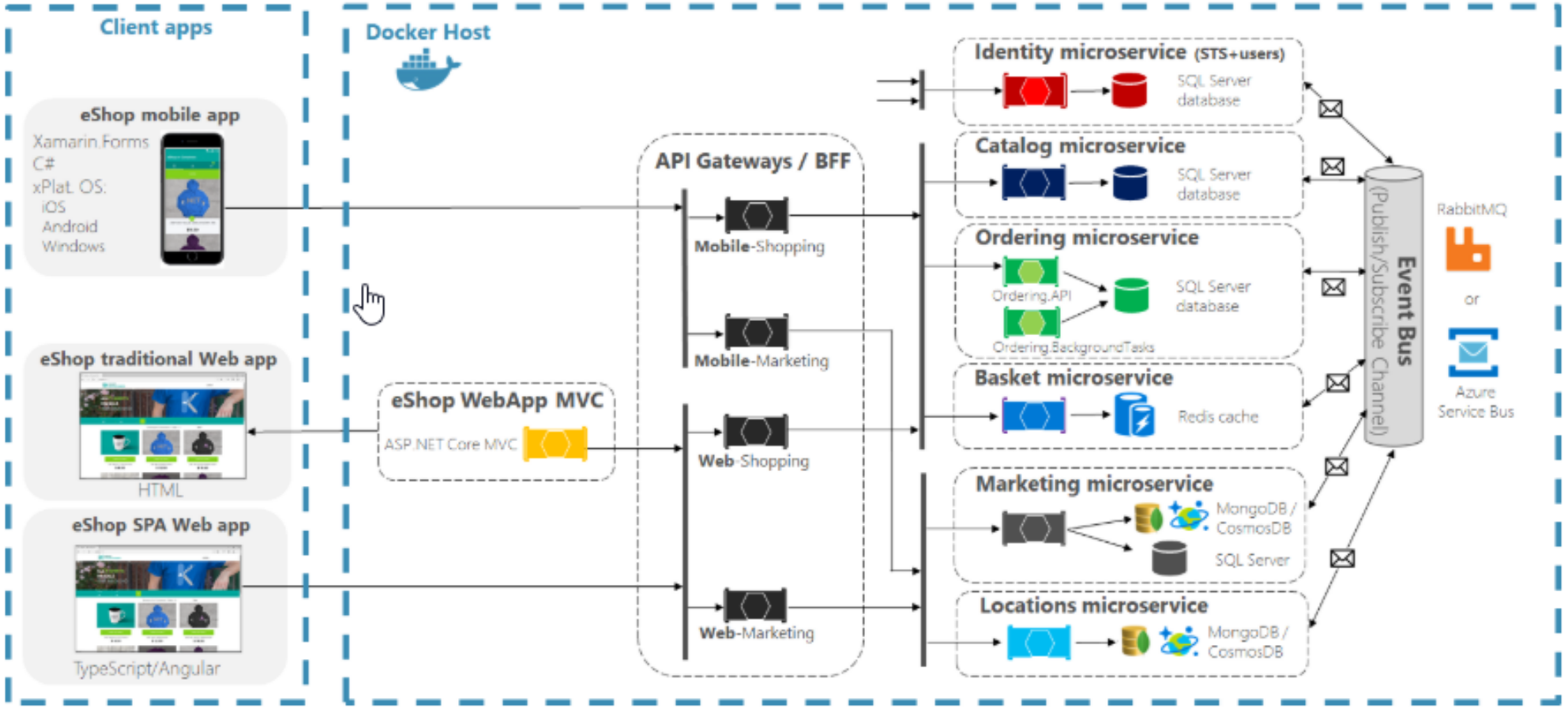
[1] *M. Fowler, J. Lewis. – http://martinfowler.com/articles/microservices.html*

MicroServices - Communication



src: *https://microservices.io/patterns/microservices.html*

MicroServices - Communication

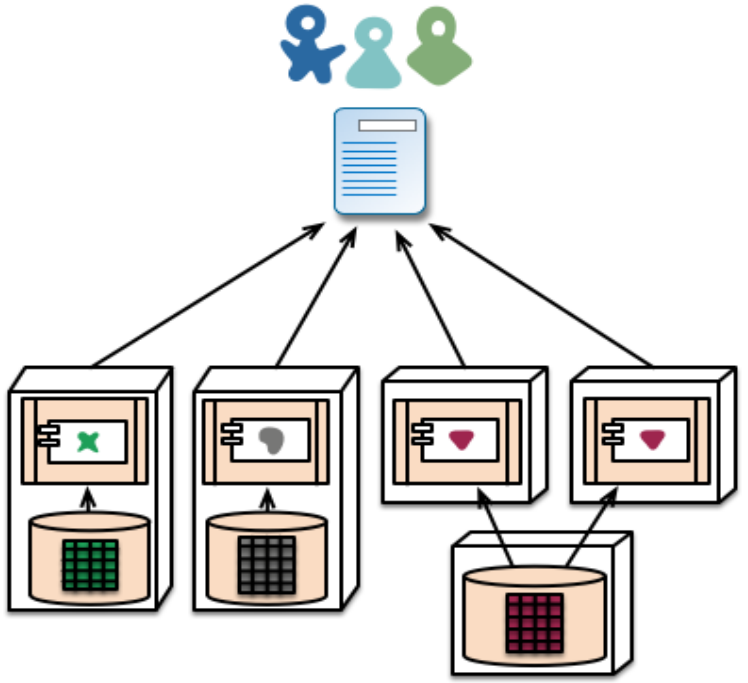


src: *https://www.asptricks.net/2019/05/distributed-microservice-architecture.html*

MicroServices – Data Storage

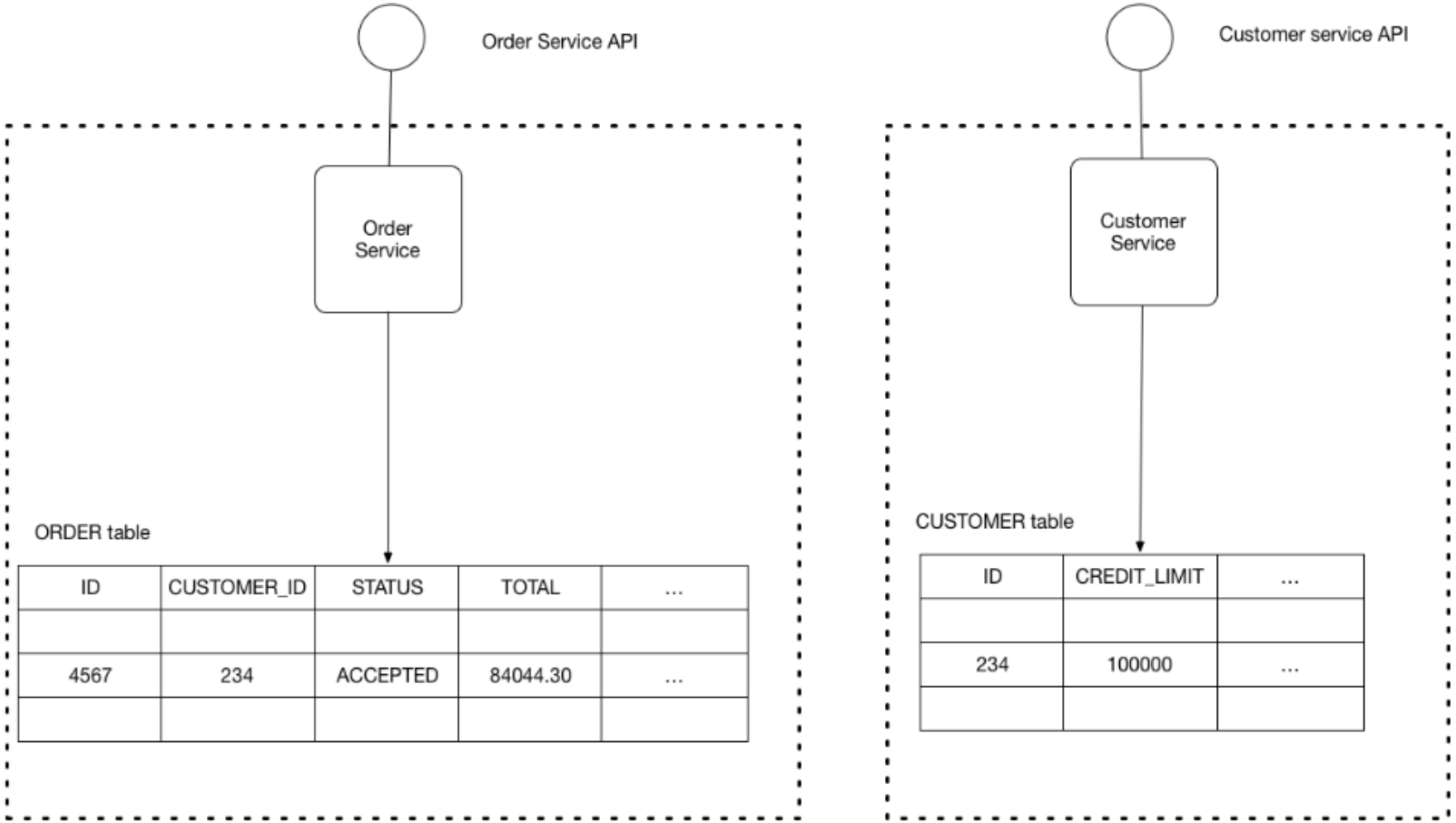
- **Всяка микроуслуга управлява собствената си база данни.**

- Различни *инстанции на една и съща* технология на базата данни или *напълно различни* системи от бази данни (*Polyglot Persistence*) [1].



[1] *M. Fowler, J. Lewis. – http://martinfowler.com/articles/microservices.html*

MicroServices - Data Storage



src: *https://microservices.io/patterns/data/database-per-service.html*