

АРХИТЕКТУРНИ СТИЛОВЕ

Част 1

Съдържание на лекцията

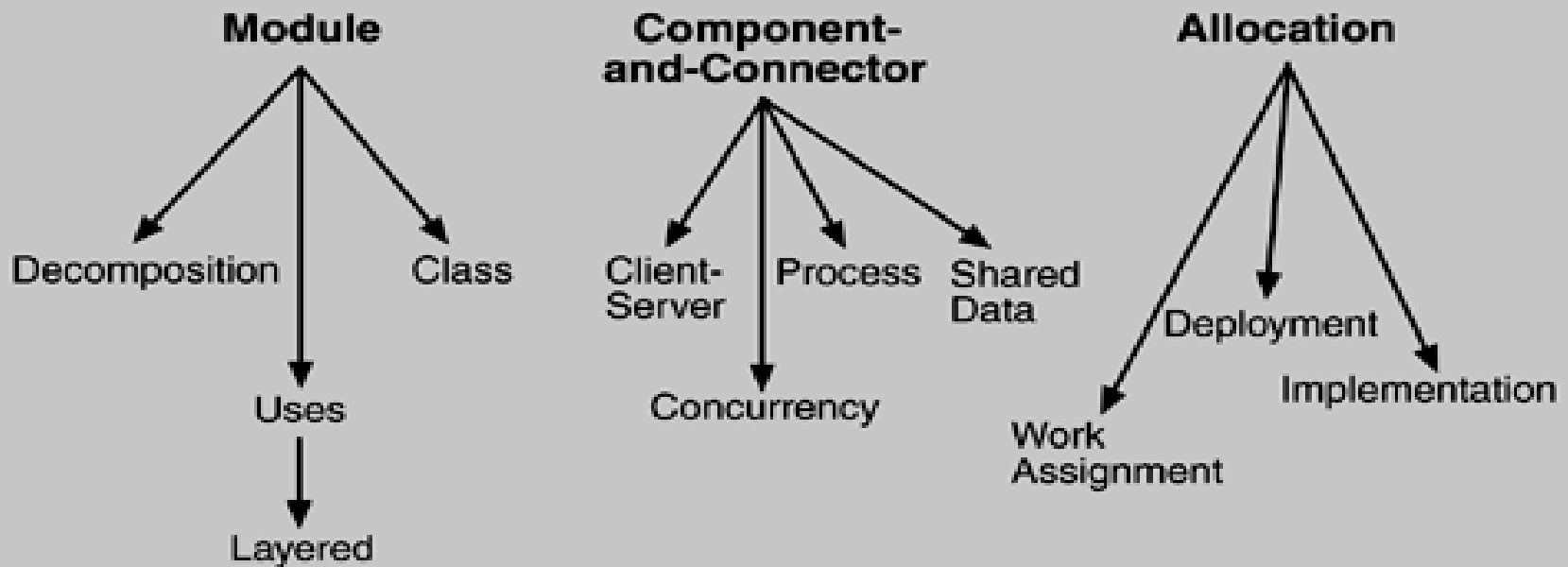
- Някои размисли върху софтуерната архитектура
- Определение на архитектурни стилове
- Различни стилове:
 - Pipe-and-Filter
 - Layered
 - Client-server
 - Repository/Blackboard
 - Model-View-Controller
 - Implicit invocation/Message passing

Софтуерна архитектура (ретроспекция)

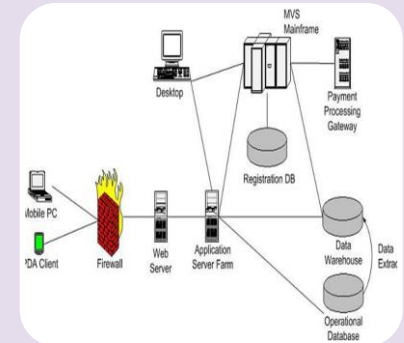
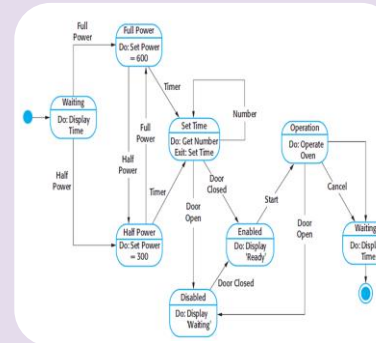
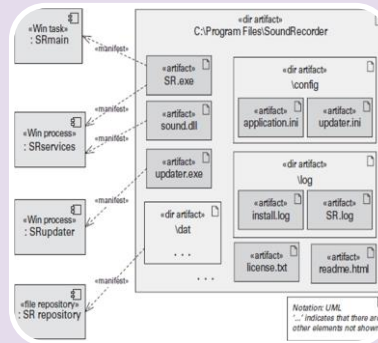
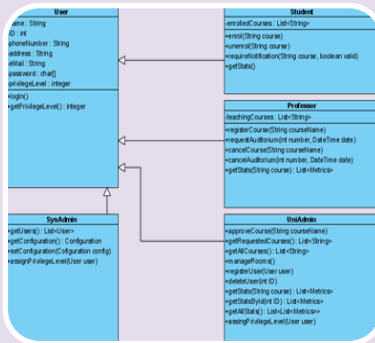
- Какво представлява софтуерната архитектура?
 - Набор (колекция) от структури, които представляват различни гледни точки върху системата
 - Всяка структура се състои от:
 - елементи,
 - техните външно видими характеристики и
 - връзките между тях
 - Структурите са представени от изгледи

Обобщение от предишни лекции

- Понятие за софтуерна архитектура (СА) и архитектурни структури



4+1 изгледа на софтуерната архитектура (по Philippe Kruchten) 1/2



Logical
view

Code
(Deve-
lopment)
view

Process
view

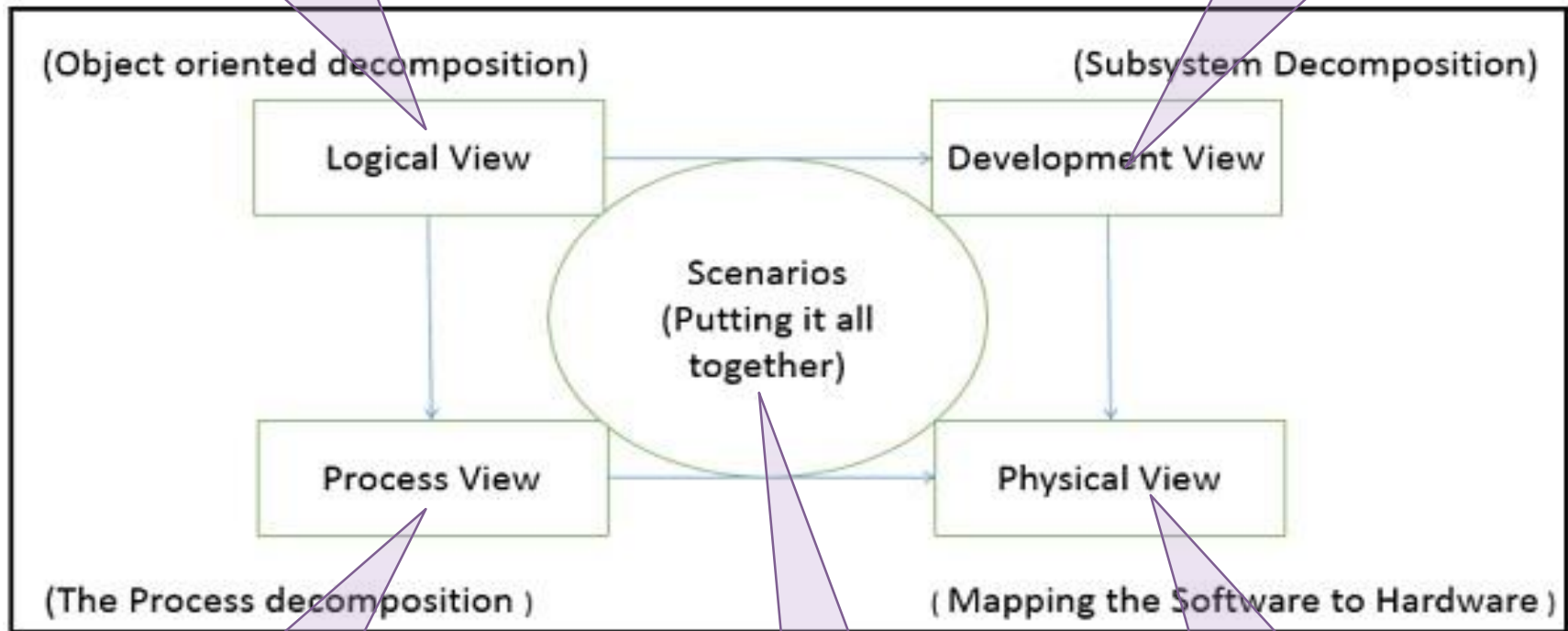
Deploy
ment
(Physical)
view

SCENARIOS

4+1 изгледа на софтуерната архитектура (по Philippe Kruchten) 2/2

т. нар. концептуален изглед - описва обектния модел на дизайна

описва статичната организация или структура на кода в средата за разработка



описва аспектите на конкурентност и синхронизация

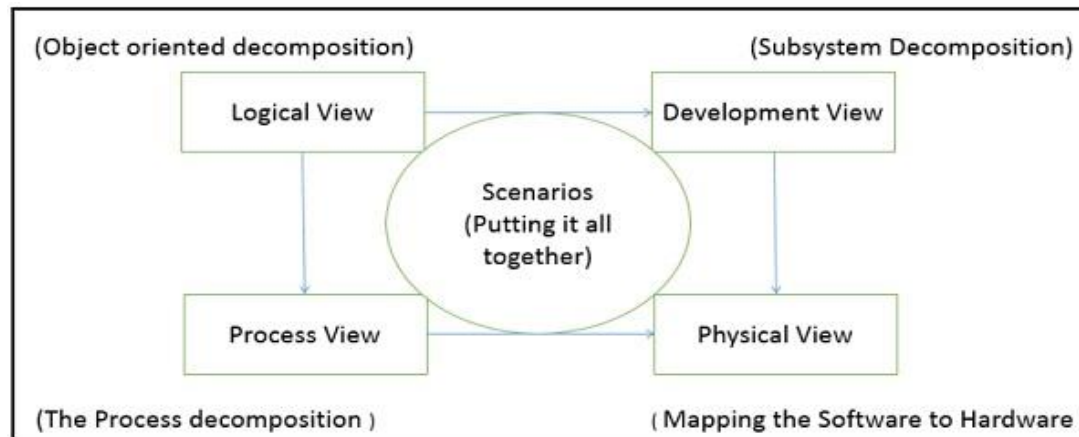
UML use cases

описва разполагането (deployment) на софтуера върху хардуера

	Logical	Process	Development	Physical	Scenario
Description	Shows the component (Object) of system as well as their interaction	Shows the processes / Workflow rules of system and how those processes communicate, focuses on dynamic view of system	Gives building block views of system and describe static organization of the system modules	Shows the installation, configuration and deployment of software application	Shows the design is complete by performing validation and illustration
Viewer / Stake holder	End-User, Analysts and Designer	Integrators & developers	Programmer and software project managers	System engineer, operators, system administrators and system installers	All the views of their views and evaluators
Consider	Functional requirements	Non Functional Requirements	Software Module organization (Software management reuse, constraint of tools)	Nonfunctional requirement regarding to underlying hardware	System Consistency and validity
UML – Diagram	Class, State, Object, sequence, Communication Diagram	Activity Diagram	Component, Package diagram	Deployment diagram	Use case diagram

Логически изглед на СА

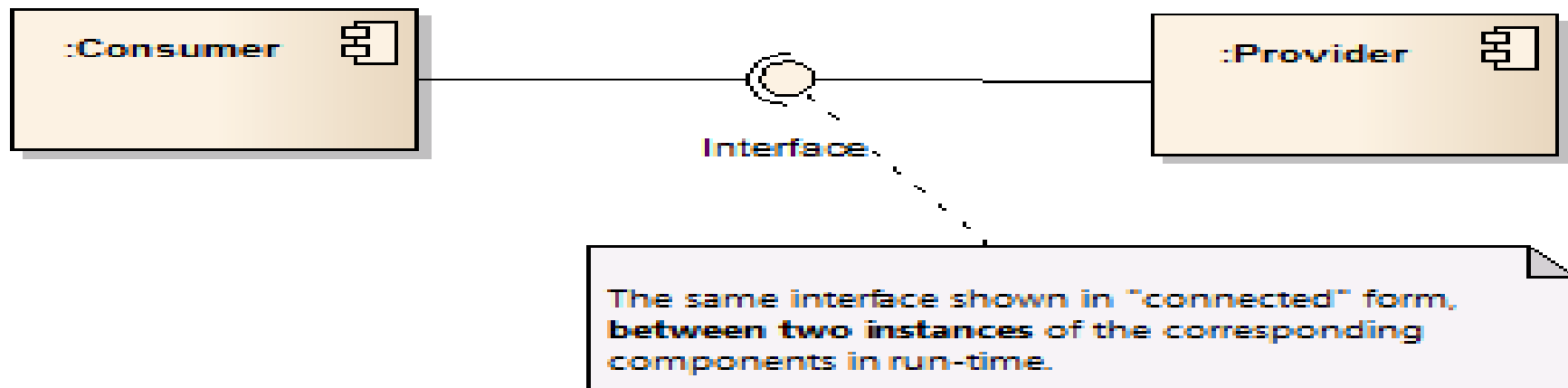
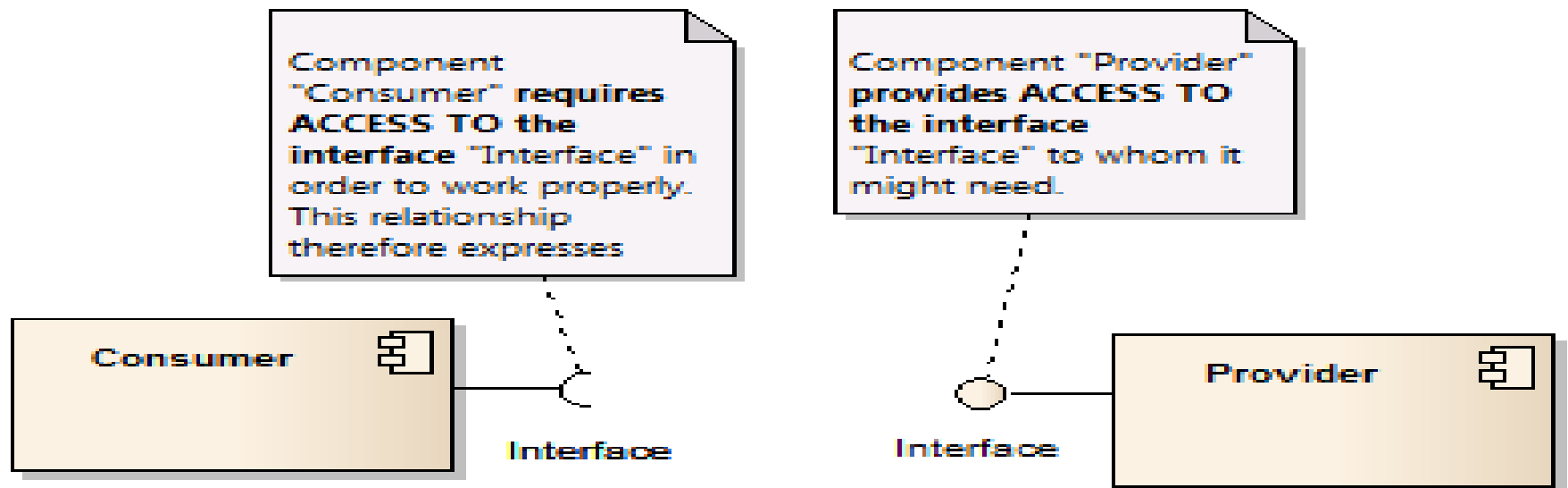
- Логическият изглед на софтуера има четири нива на абстракция:
 - **Компоненти и конектори**
 - Техните **интерфейси**
 - Архитектурни **конфигурации** - специфична топология на взаимосвързани компоненти и конектори
 - Архитектурни **стилове** – образци (patterns) за успешни и практически доказани архитектурни конфигурации



Какво представлява софтуерният компонент?

Дефиниция:

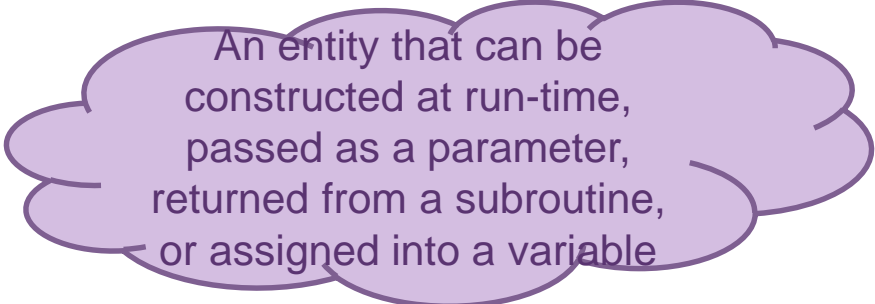
1. Изчислителна (софтуерна) единица,
2. която има определена функционалност,
3. която е достъпна чрез добре дефинирани интерфейси и
4. има изрично специфицирани зависимости:
 - Входен интерфейс (required interface), който определя какво се нуждае от компонента, за да изпълни своята функционалност
 - Изходен интерфейс (provided interface), който уточнява какво произвежда компонентът като резултат, при условие, че всичко от спецификацията на входния интерфейс е изпълнено



Какво представлява софтуерният конектор?

Дефиниция: First class entity, което представлява

1. механизма за взаимодействие (протокола) между компонентите и
2. правилата за комуникацията (за трансфер на управление и данни)
 - Конекторите моделират статичните и динамичните аспекти на взаимодействието между компонентните интерфейси
 - Конекторите също имат интерфейси, понякога наричани роли
 - Както компонентите, така и конекторите може да се използват многократно



An entity that can be constructed at run-time, passed as a parameter, returned from a subroutine, or assigned into a variable

Connector \neq Component 1/2

1. Прямо зависимостта от приложението:

- Компонентите осигуряват специфична за приложението (application-specific) функционалност или application-independent инфраструктура
- Конекторите осигуряват независими от приложението механизми за взаимодействие

2. Прямо кода:

- Компонентите могат да бъдат изградени от специфични артефакти на кода (напр. compilation units или пакети)
- Конекторите обикновено не се виждат директно в кода (напр. връзка между модули, връзки в мрежата, конфигурации на адресите на сървъра)

Connector \neq Component 2/2

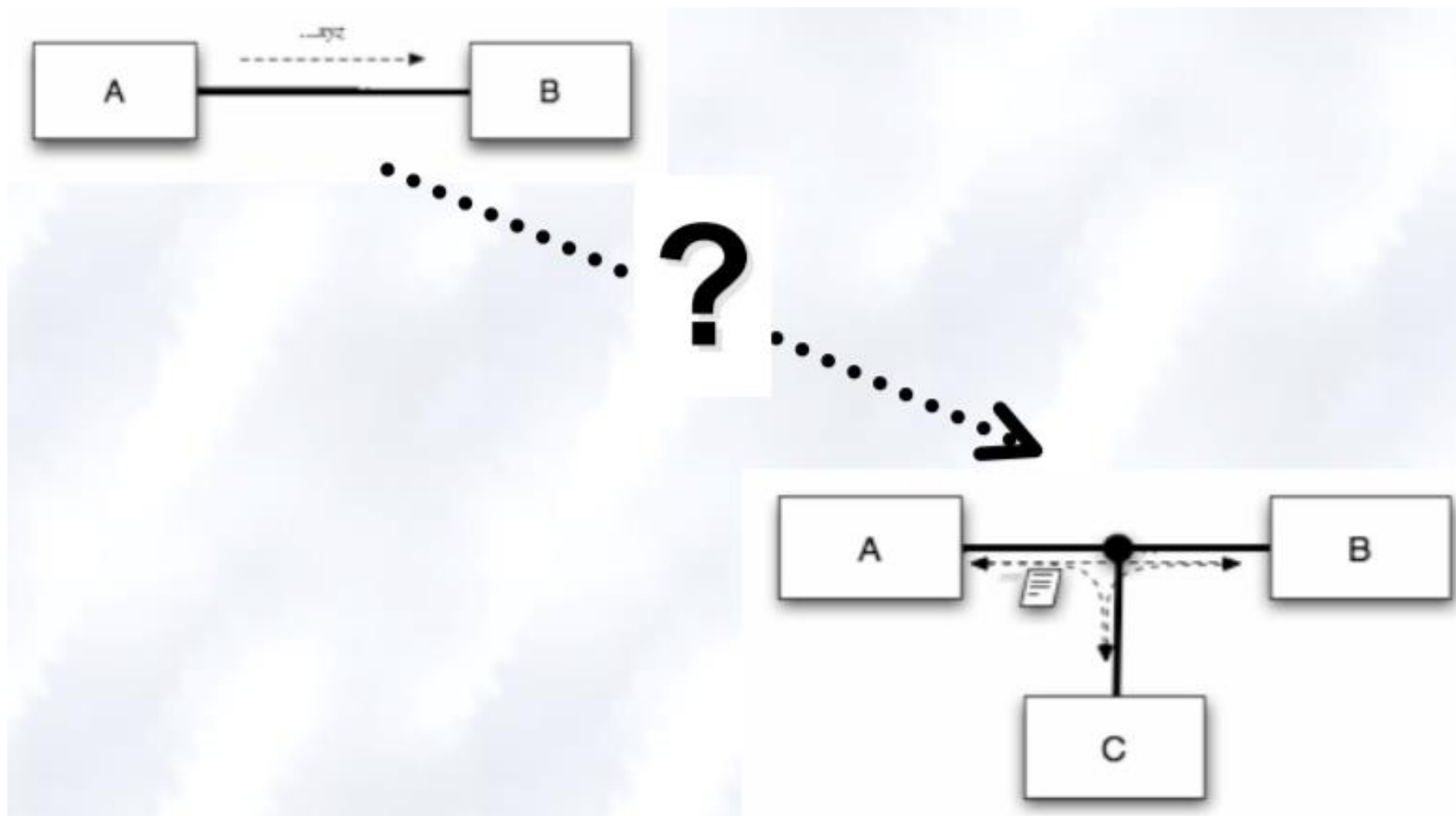
3. Прямо типа на абстракция:

- Компонентите осигуряват абстракция на групирането на функционалности
- Конекторите осигуряват абстракция на взаимодействие и/или параметризация, напр.:
 - Спецификация на сложни взаимодействия
 - Бинарни спрямо N-арни конектори
 - Асиметрични спрямо симетрични конектори
 - Протоколи за взаимодействие

Софтуерните конектори и описанието на взаимодействието м/у компонентите

- Конекторите представят взаимодействието между множество компоненти
- Спецификация на протокол (понякога неявна), която определя неговите свойства:
 - Видове интерфейси, които може да свърже
 - Гаранции за свойствата за взаимодействието
 - Правила за подреждане на взаимодействията във времето
 - Качества на взаимодействие (напр. производителност)

Пример за конектор



Примери за конектори

□ Прости взаимодействия:

- Отдалечено извикване на процедура (RPC)
- Достъп до споделени променливи (shared variables)

□ Сложни и семантично-богати взаимодействия:

- Клиент-сървър протоколи
- Протоколи до достъп до БД
- Мултикастинг на асинхронни събития

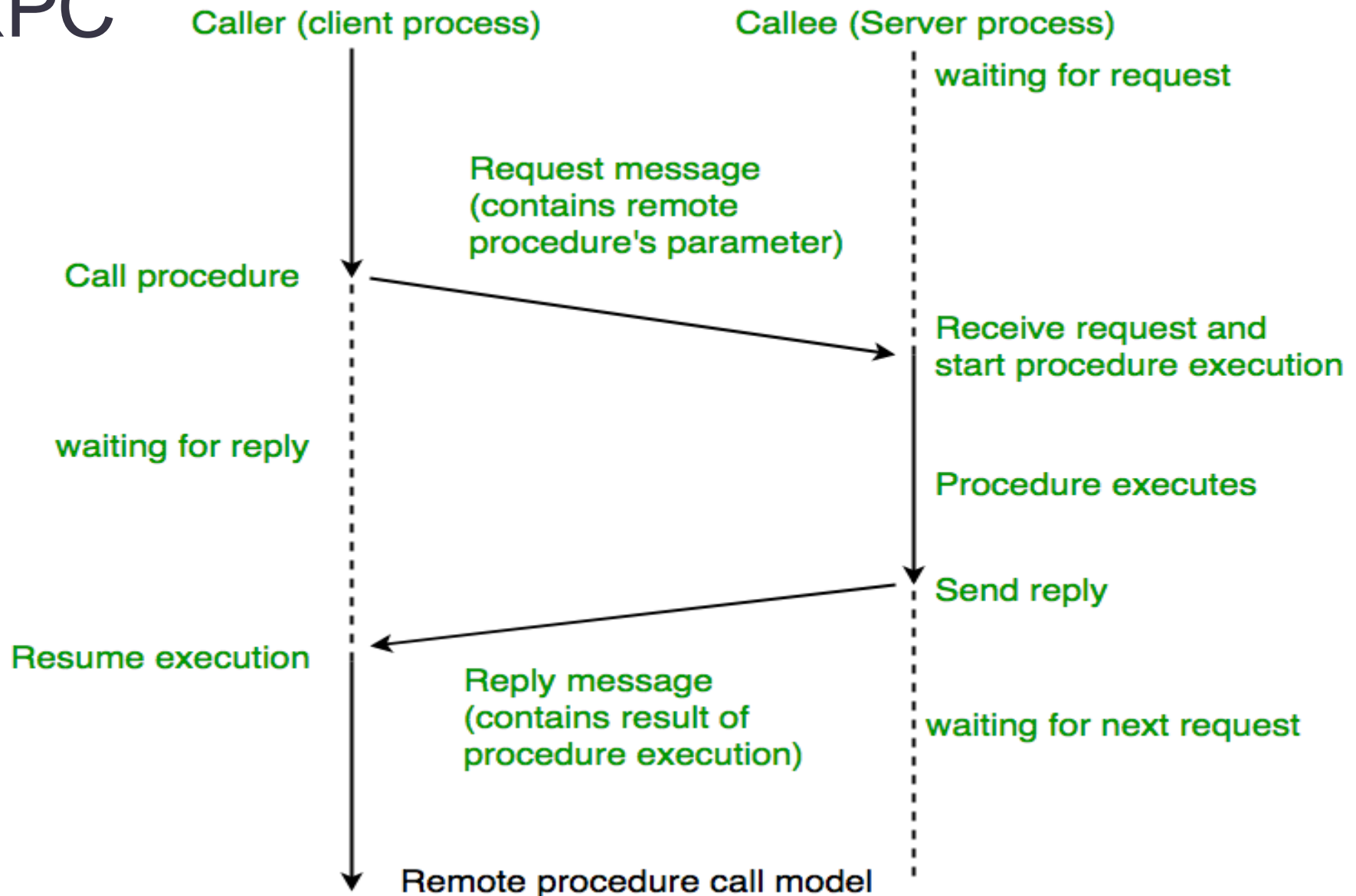
Роли на конекторите

- ☐ Комуникация
- ☐ Координация
- ☐ Преобразуване (Conversion)
- ☐ Улесняване (Facilitation)

Конекторите като *комуникатори*

- Разделя комуникацията от изчисленията
- Може да повлияе на нефункционалните характеристики на системата като производителност, мащабируемост, сигурност
- Поддържа:
 - Различни механизми за комуникация – напр. Remote Procedure Call (RPC), shared data access, message passing
 - Ограничения в структурата / посоката на комуникацията – напр. pipes
 - Ограничения за качеството на услугата – напр. persistence

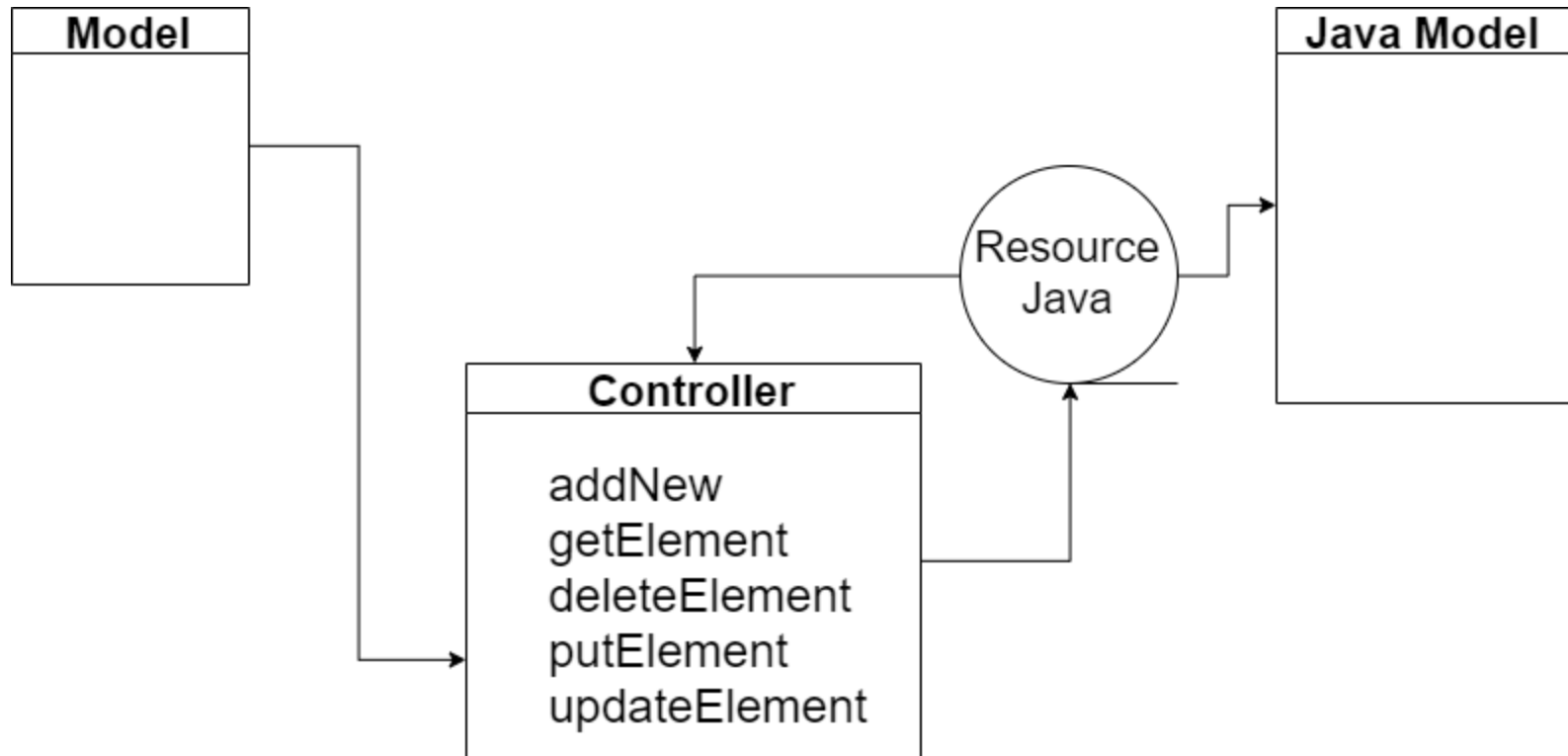
RPC



Конекторите като *координатори*

- Определят контрола над изчисленията
- Координират предаването на данните
- Разделят контрола от изчисленията
- Ортогонални са на комуникацията, конверсията и улесняване – елементи на управление/координация има в комуникацията, преобразуването и улесняването

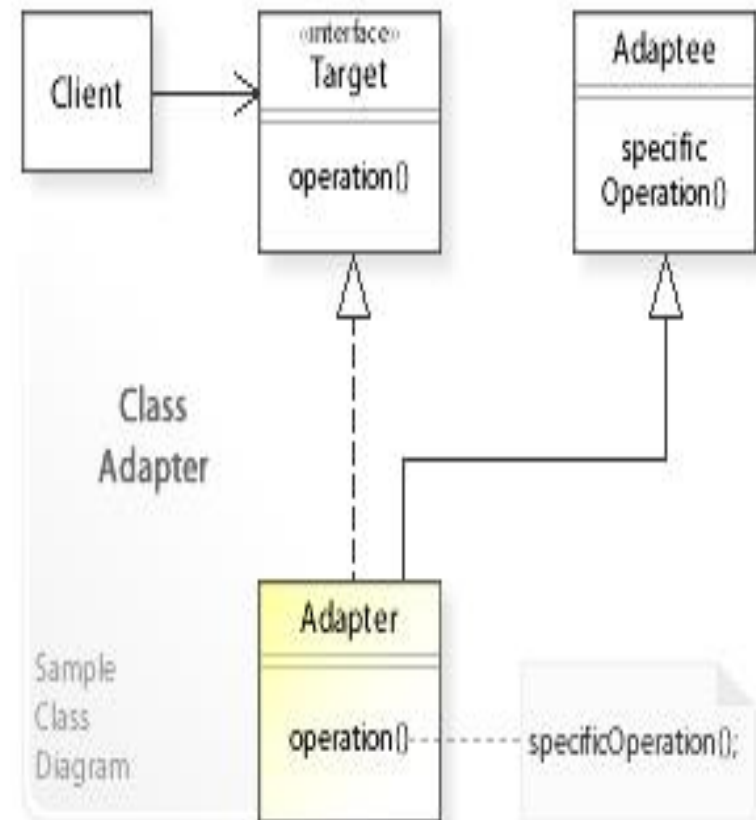
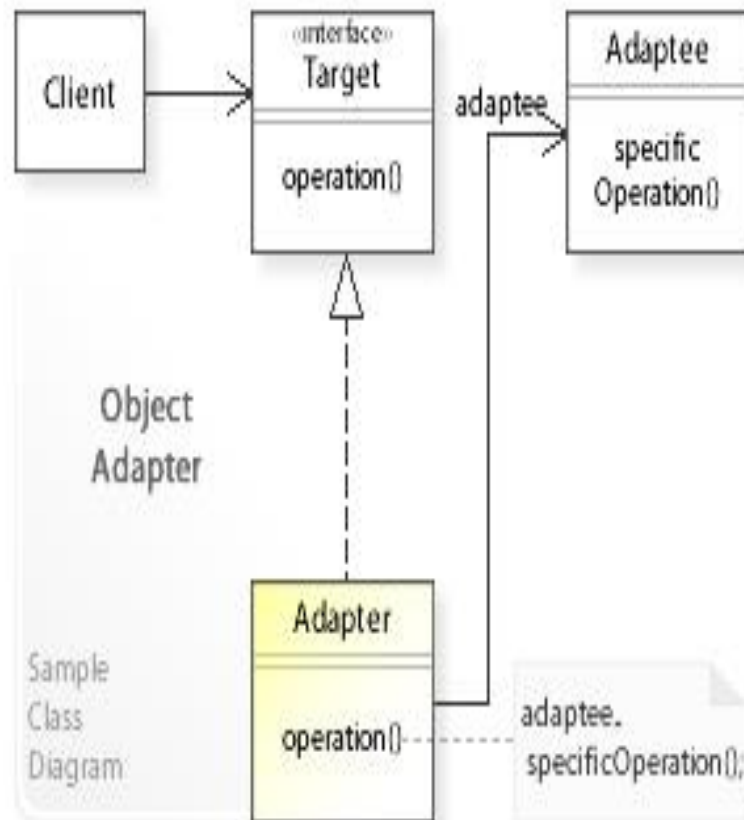
Пример – контролер на данни



Конекторите като *конвертори*

- Правят възможно взаимодействието между независимо разработени и несъответстващи си компоненти
- Несъответствия, базирани на взаимодействие:
 - по тип
 - по брой
 - по честота
 - по ред на следване
- Примери за конвертори
 - Adaptors (адаптори)
 - Wrappers (обвивки)

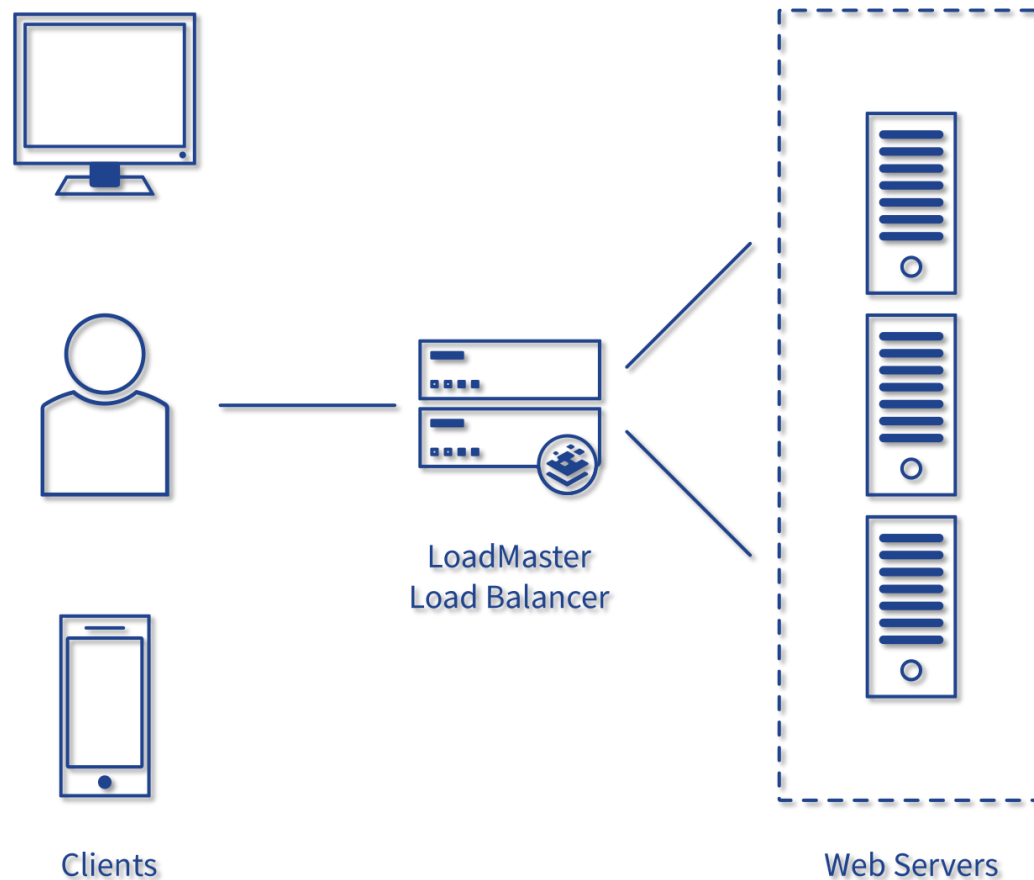
Пример - адаптери



Конекторите като *фасилитатори*

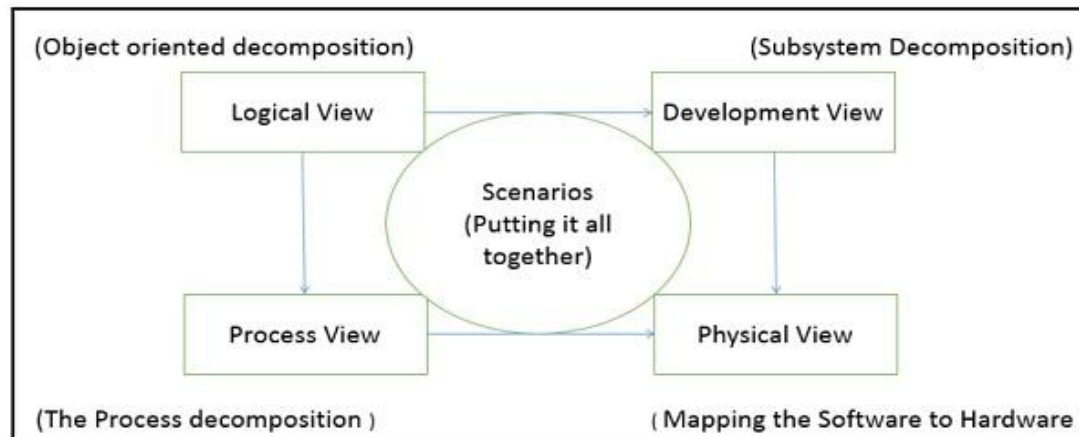
- Правят възможно взаимодействието между компоненти, предназначени да работят заедно – например ***медиатори и оптимизатори*** в поточни взаимодействия
- Осигуряват желани профили за производителност - например ***балансиране на натоварването***
- Управляват достъпа до споделена информация, като осигуряват ***механизми за синхронизация***
 - Критични секции
 - Монитори

Пример – балансиране на товар



Логически изглед на СА

- Логическият изглед на софтуера има четири нива на абстракция:
 - *Компоненти и конектори*
 - Техните *интерфейси*
 - Архитектурни *конфигурации* - специфична топология на взаимосвързани компоненти и конектори
 - Архитектурни *стилове* – образци (patterns) за успешни и практически доказани архитектурни конфигурации



Архитектурен стил

- Дефиниция: *Архитектурният стил определя семейство от системи по отношение на модел (образец) на структурна организация.*
- Архитектурните стилове определят:
 - Речникът (vocabulary) от компоненти и конектори, които могат да се използват в екземпляри от този стил
 - Набор от ограничения (constraints) за това как те могат да бъдат комбинирани, като:
 - Топология на описанията (например, без цикли)
 - Семантика на изпълнение (например, процеси които се изпълняват конкурентно/паралелно)

Архитектурни стилове

- Pipe-and-Filter
- Layered
- Client-server
- Repository/Blackboard
- Object-oriented style
- Други (следващата седмица)

Архитектурен стил Pipe-and-Filter

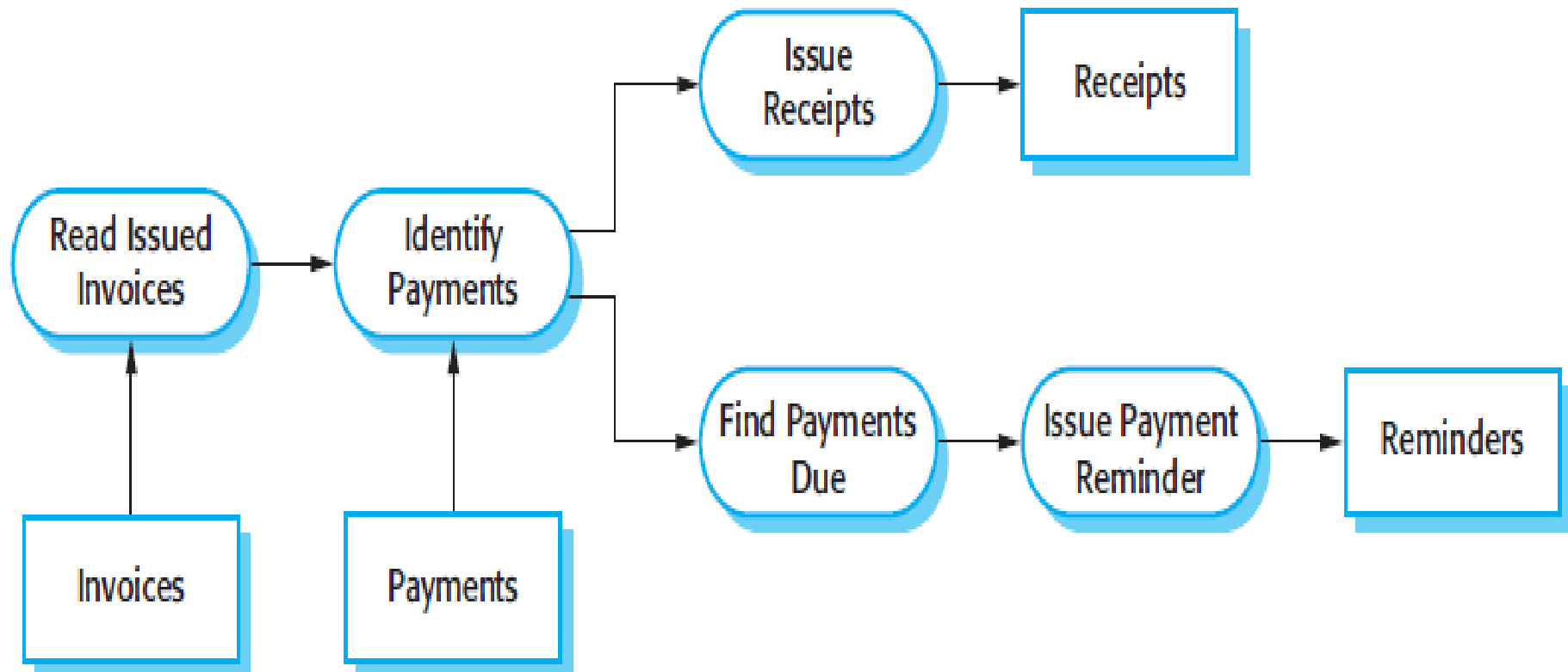
- Всеки компонент (филтър) в системата прехвърля данните в последователен ред към следващия компонент
- Конекторите (тръбите, pipes) между филтрите представляват действителните механизми за пренос на данни



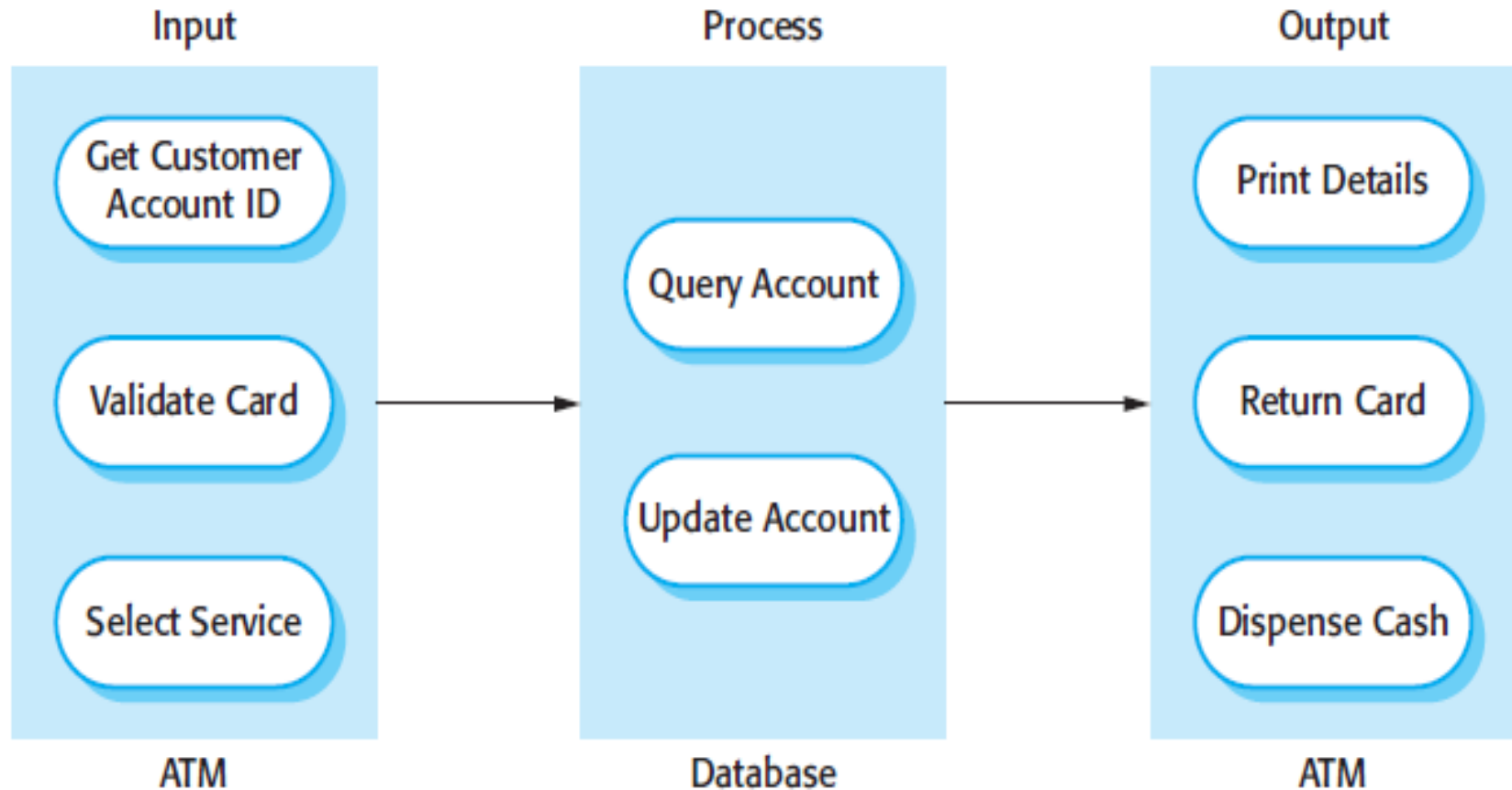
Pipe-and-Filter style

- Името „pipe and filter“ идва от системата Unix, където е възможно да се свържат процеси, използвайки „тръби“ (pipes).
- Тръбите предават текстов поток (stream) от един процес в друг.
- Филтрите представляват изчислителни единици в системата.
 - Те съдържат функционалността
 - Те четат данни чрез входните си интерфейси, след това ги обработват и накрая изпращат данните до изходните си интерфейси
 - Филтрите нямат информация за своите съседни
- Тръбите (pipes) имат задължението да прехвърлят данните от изхода на филтър до входа на следващия филтър

Pipe-and-Filter стил – пример

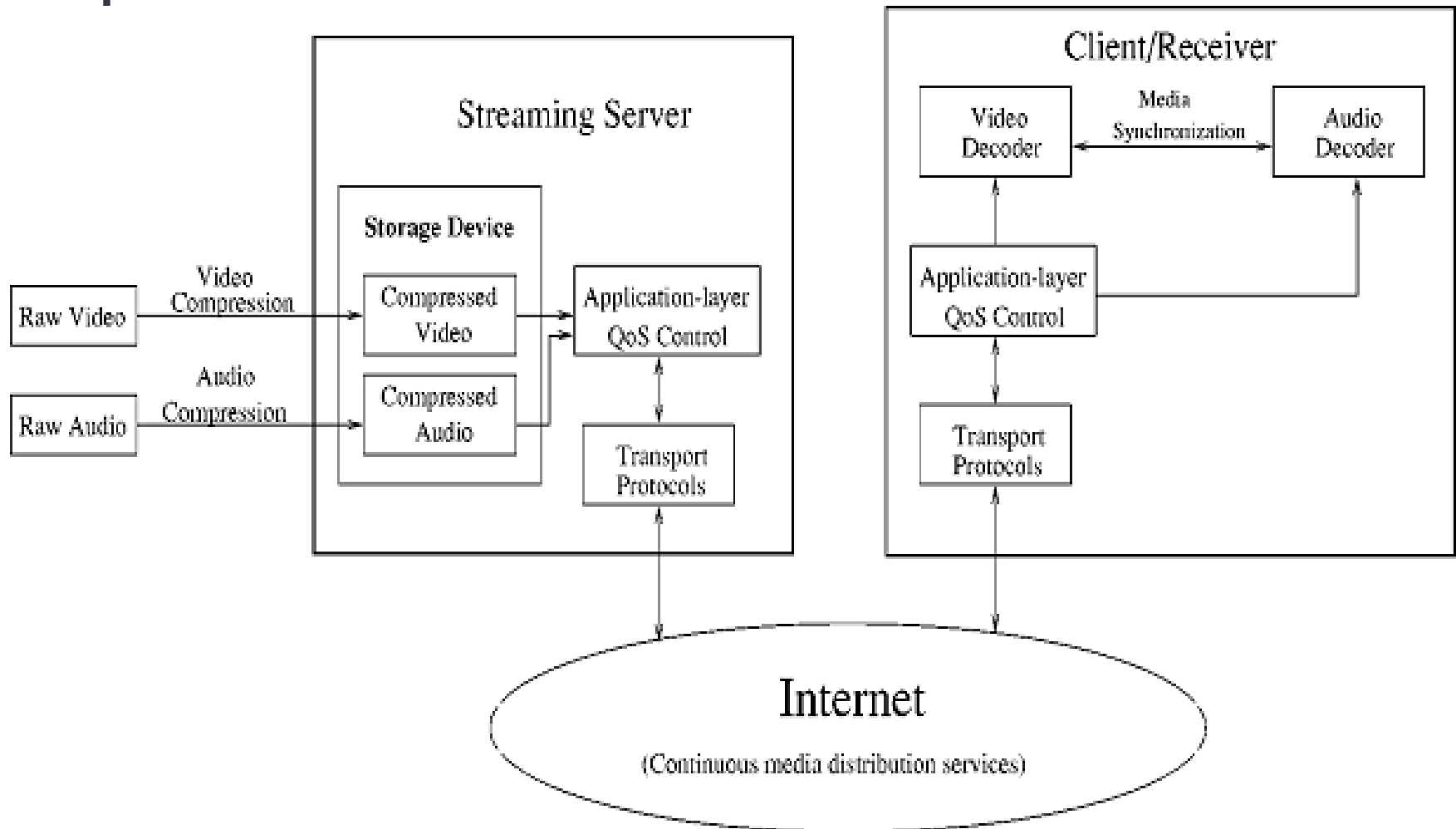


Софтуерна архитектура на АТМ (Automated Teller Machine) система

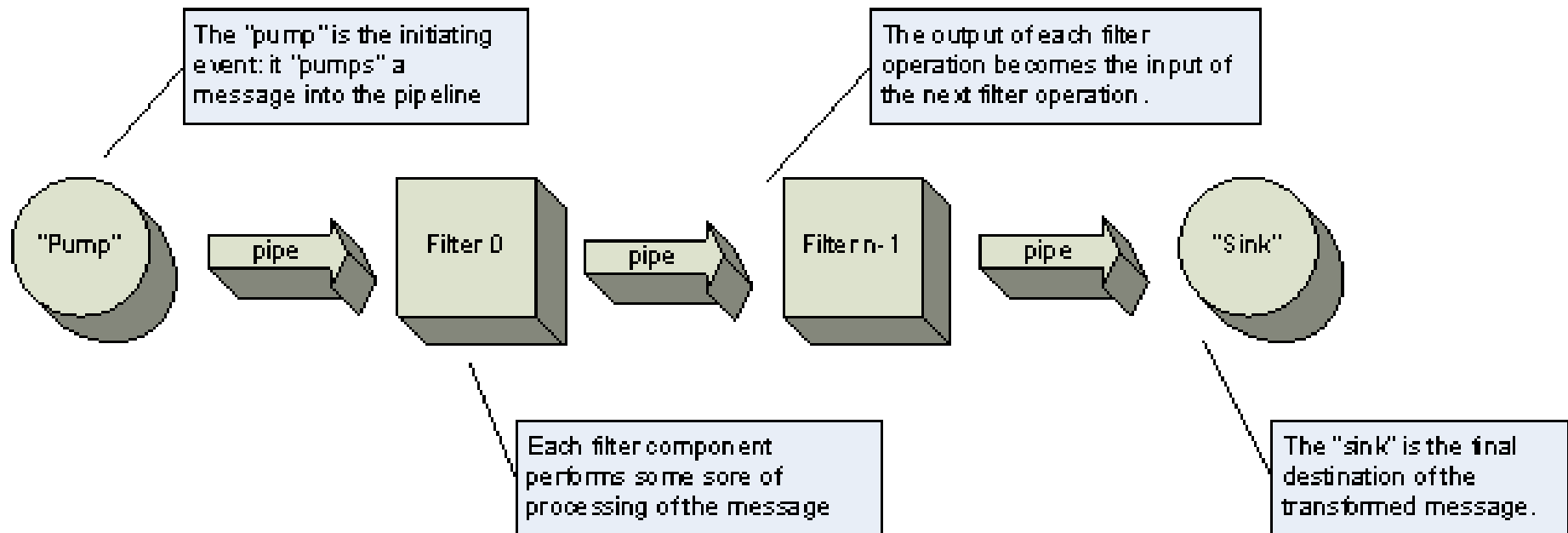


Source: Software Engineering by Ian Sommerville, 9th edition (2010), Addison-Wesley Pub Co;

Софтуерна архитектура за видео стрийминг



Какви алтернативи на този стил можем да предложим?

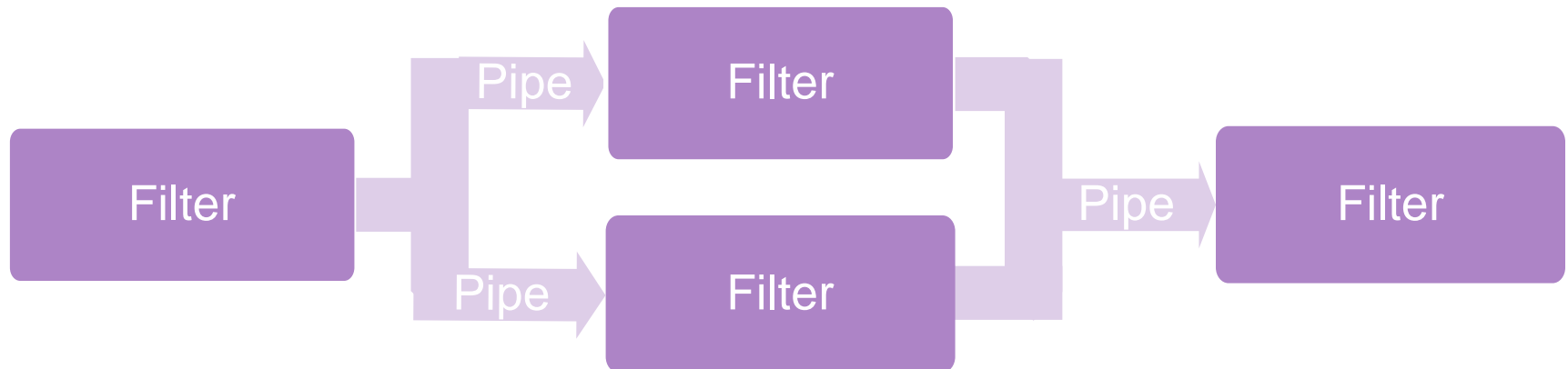


Вариации на стила pipe-and-filter 1/3

- Последователна обработка – пакетна (Batch-sequential) или поточна (Pipeline/stream)



- Паралелна обработка



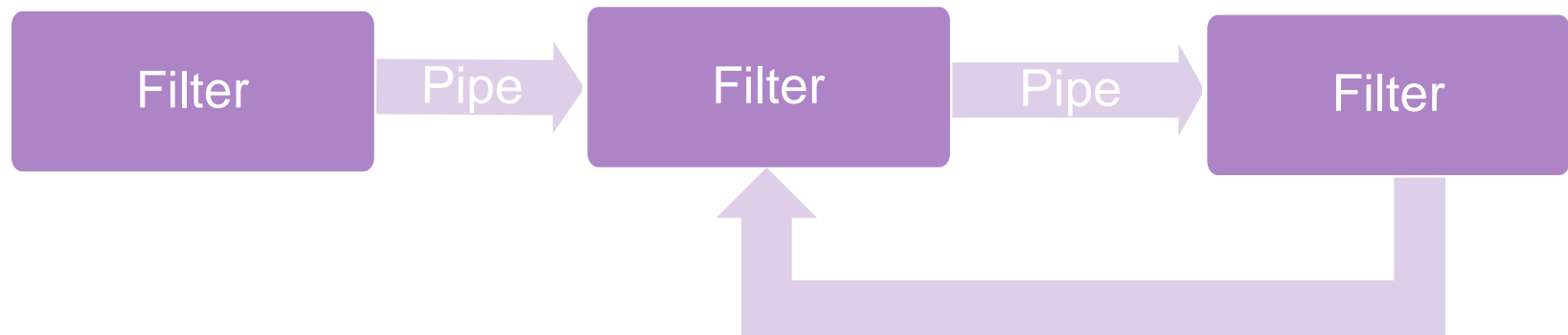
Вариации на стила pipe-and-filter 2/3

По отношение на комуникационния протокол:

- **Pipeline/stream стил (конвейерен поток)** – обработката на данни може да започне веднага след получаване на първия байт от филтъра
- Сравнете го с **batch-sequential стил (пакетна последователност)**, което изисква да бъдат прехвърлени всичките данни, преди филтърът да започне да работи с тях

Вариации на стила pipe-and-filter 2/3

- Цикъл (Loopback)



Предимства на стила pipe-and-filter

- Интуитивен и лесен за разбиране
- Филтрите са самостоятелни и могат да бъдат третираны като черни кутии, което води до гъвкавост по отношение на поддръжката и повторната употреба
- Лесна за внедряване на паралелност (не в пакетна последователност)
- Той е пряко приложим за структурите на много бизнес процеси
- Лесен за използване, когато обработката, изисквана от приложението, може лесно да бъде разложена на набор от дискретни, независими стъпки

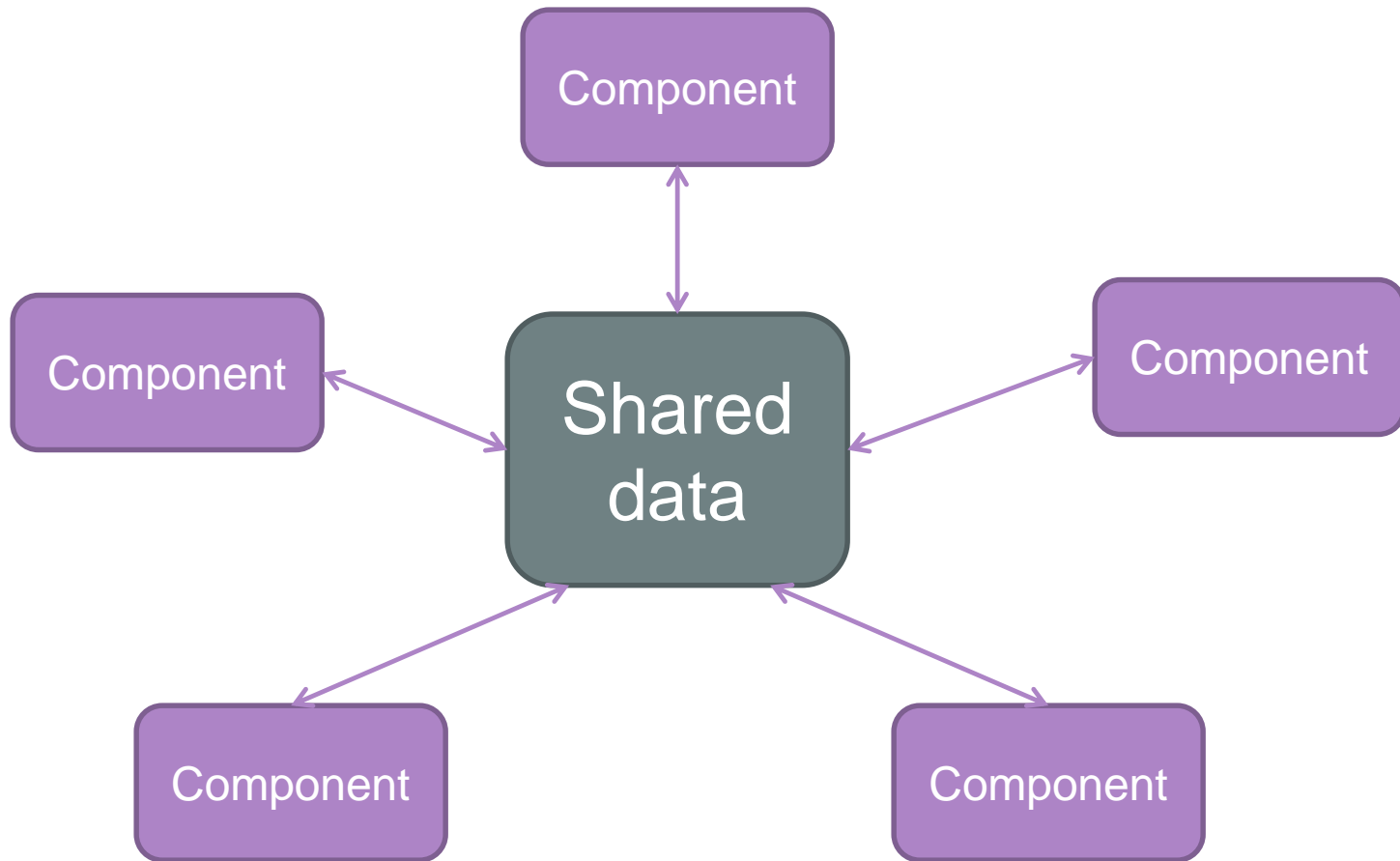
Недостатъци на стила pipe-and-filter

- Поради последователните стъпки на изпълнение е трудно да се реализират интерактивни приложения
- Ниска производителност
 - Всеки филтър трябва да анализира данните
 - Трудно споделяне на глобални данни
 - Филтрите трябва да се съгласуват относно формата на данните

Pipe-and-Filter style – особености

- **Complexity** – в разпределена среда, филтрите може да се изпълняват на различни сървъри
- **Reliability** – използва инфраструктура, която гарантира, че при преминаването на данните между филтрите в конвейера те няма да бъдат загубени
- **Idempotency** – откриване и премахване на дублиращи се съобщения
- **Context and state** – всеки филтър трябва да бъде снабден с достатъчен контекст, с който да може да изпълнява работата си, което може да изисква значително количество информация за състоянието

Shared-data (споделени данни) стил



Shared-data стил

- Активно се използва в системи, където компонентите трябва да прехвърлят големи количества данни
- Споделените данни могат да се разглеждат като конектор между компонентите
- Shared-data стил – вариации:
 - **Blackboard** (черна дъска) – когато някакви данни се изпращат към конектора за споделяни данни, всички компоненти трябва да бъдат информирани за това. С други думи, споделените данни са активен агент;
 - **Repository** (хранилище) - споделените данни са пасивни, до компонентите не се изпращат известия.

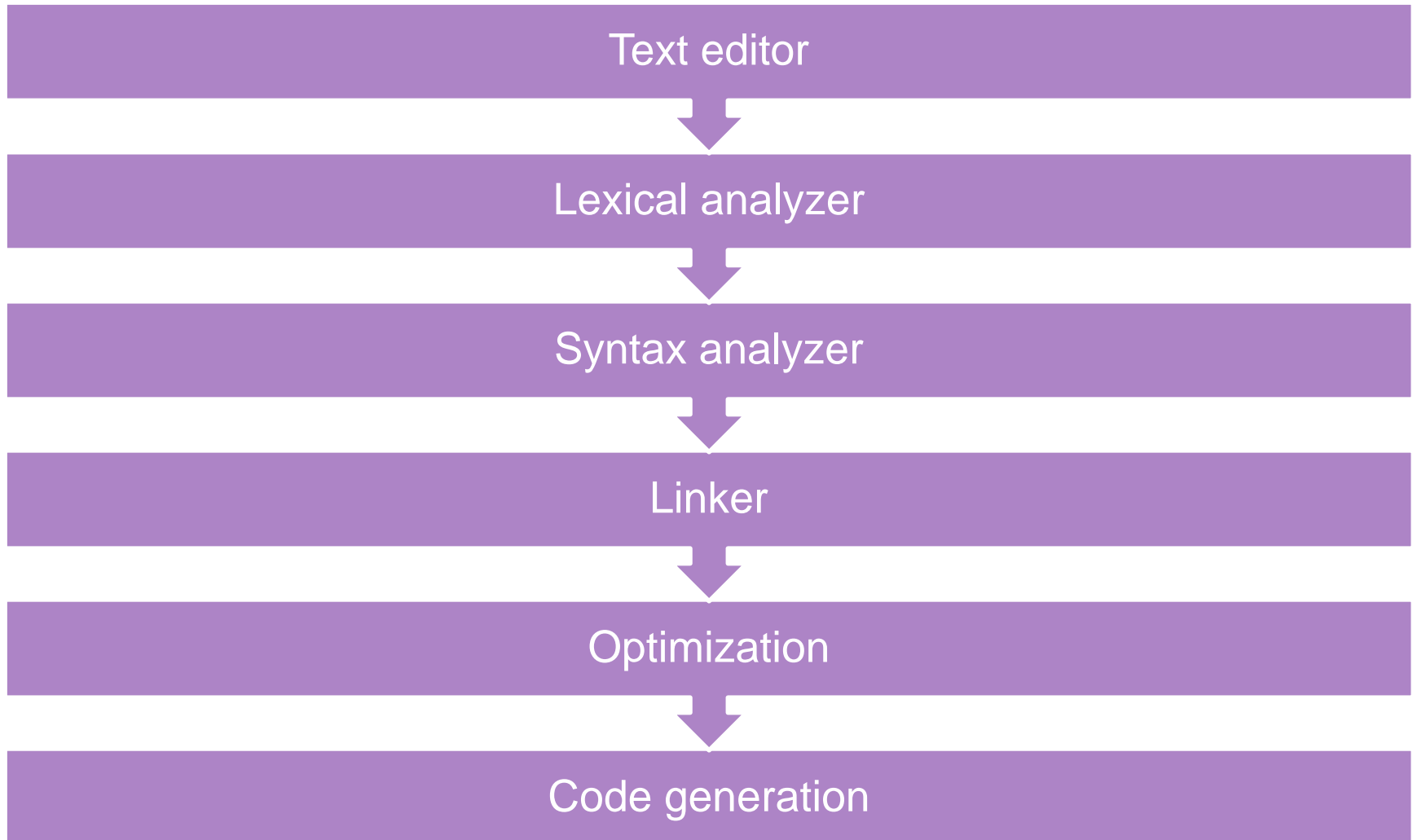
Предимства на стила shared-data style

- Scalability – могат да се добавят лесно нови компоненти
- Concurrency – всички компоненти могат да работят паралелно
- Високо ефективен при обмен на големи количества данни
- Централизирано управление на данните:
 - По-добри условия за сигурност, архивиране (backup) и т.н.
 - Компонентите са независими от производителя на данни

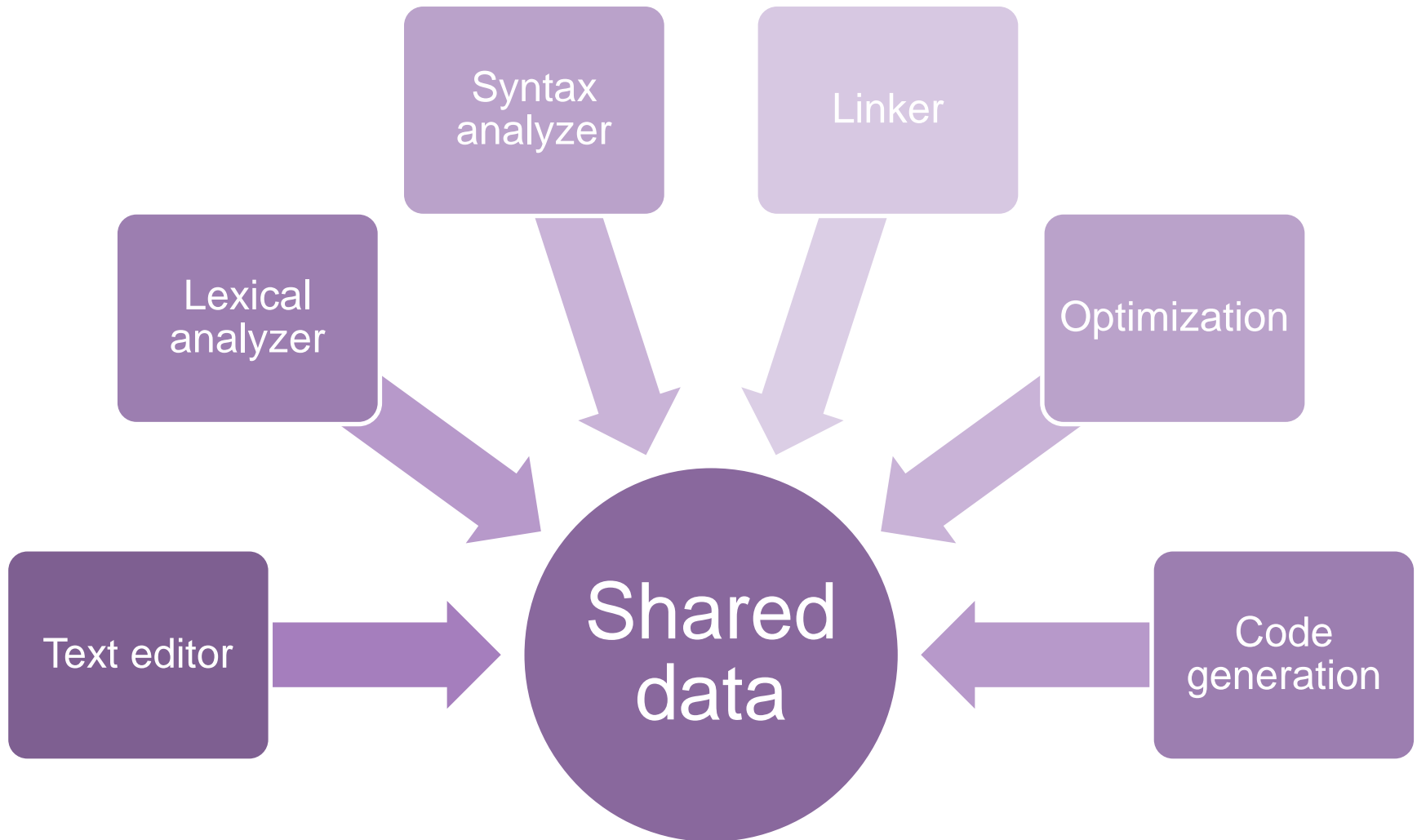
Недостатъци на стила shared-data

- Трудно се прилага в разпределена среда
- Споделените данни трябва да поддържат единен модел на данни
- Промените в модела могат да доведат до ненужни разходи
- Тясна зависимост между blackboard и източника на данни
- Може да се превърне в тясно място (bottleneck) в случай на твърде много клиенти

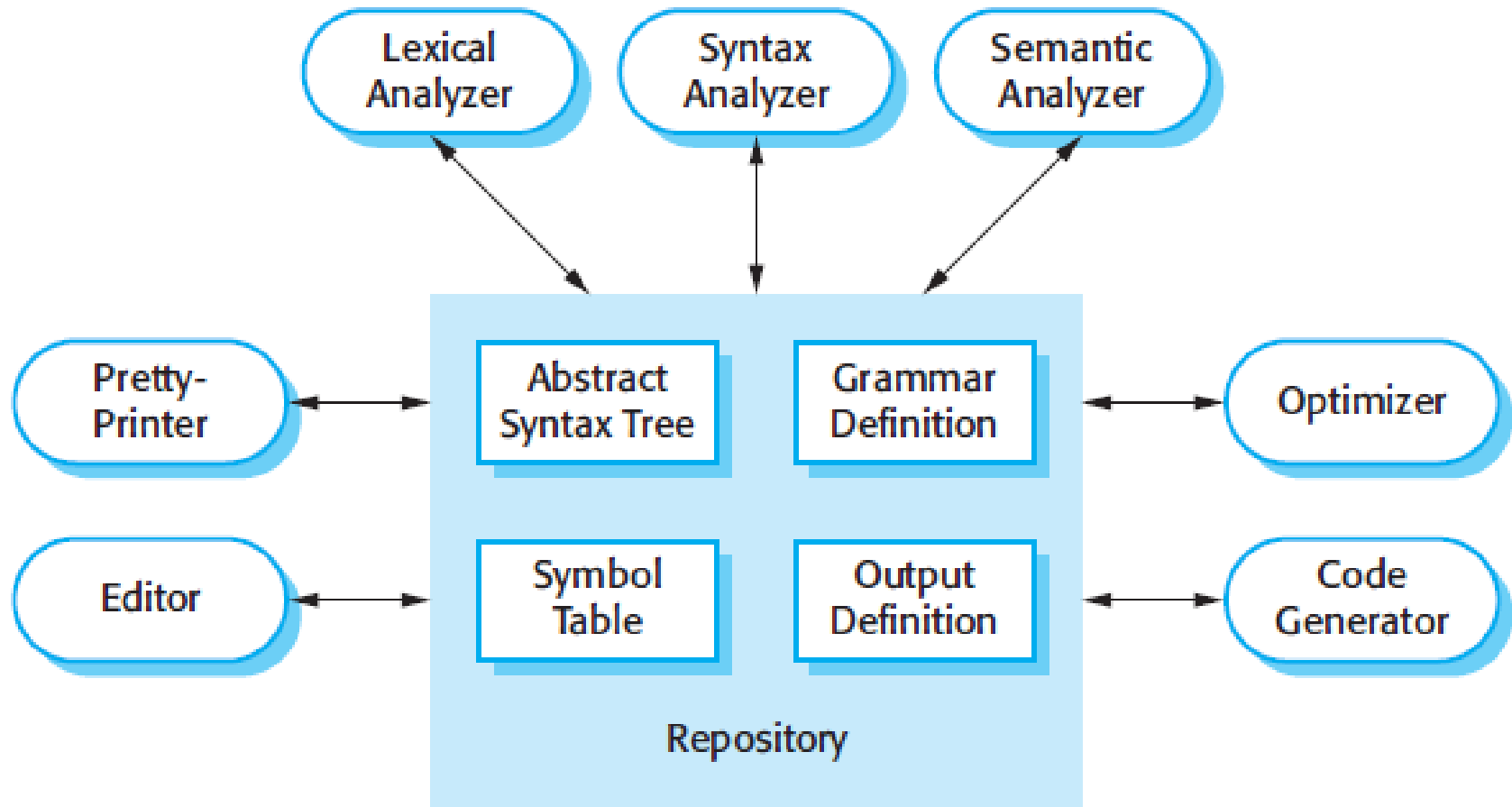
Архитектура на компилатор (pipeline)



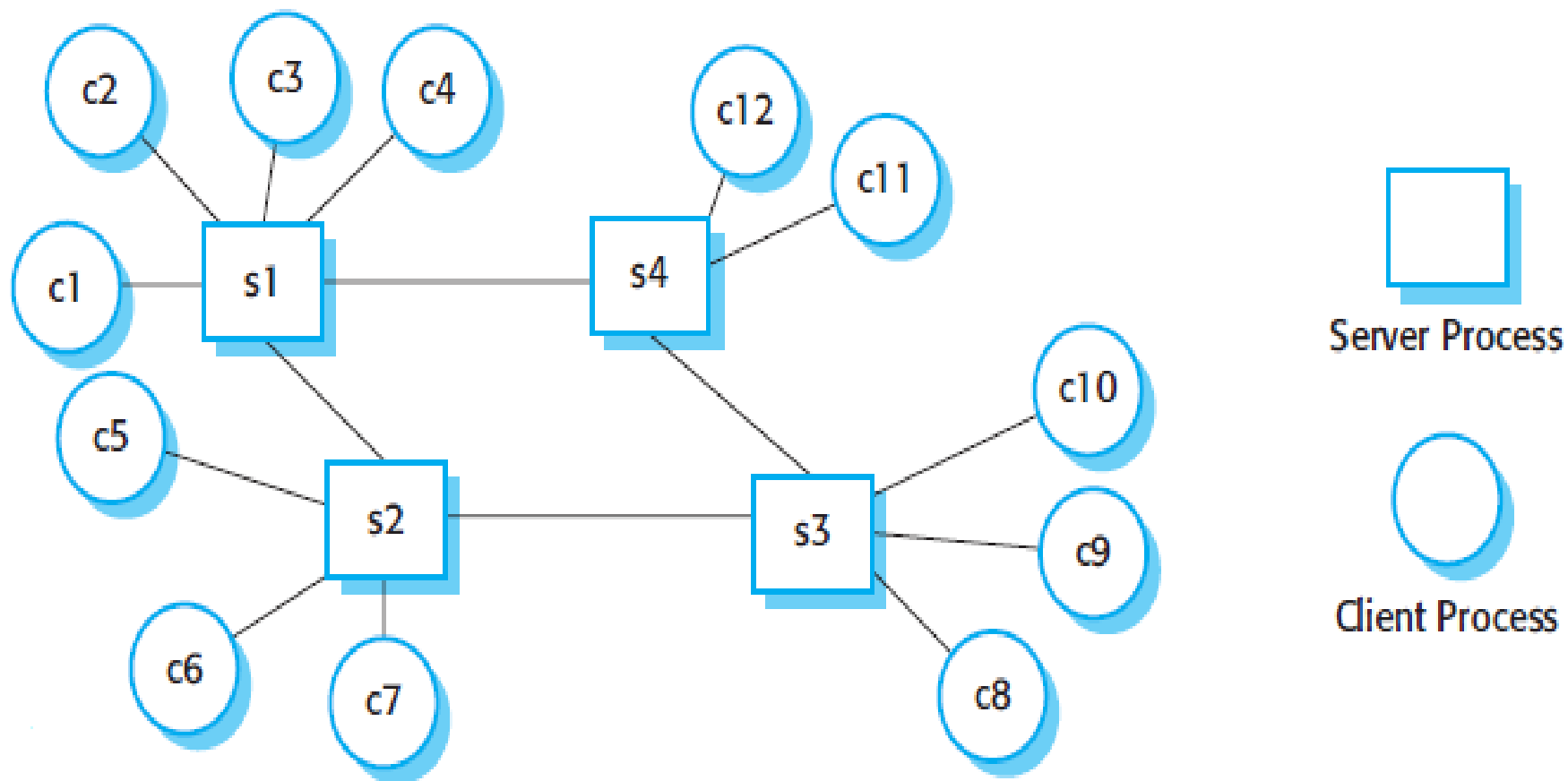
Архитектура на компилатор (shared data)



Архитектура на компилатор (shared data) - детайли



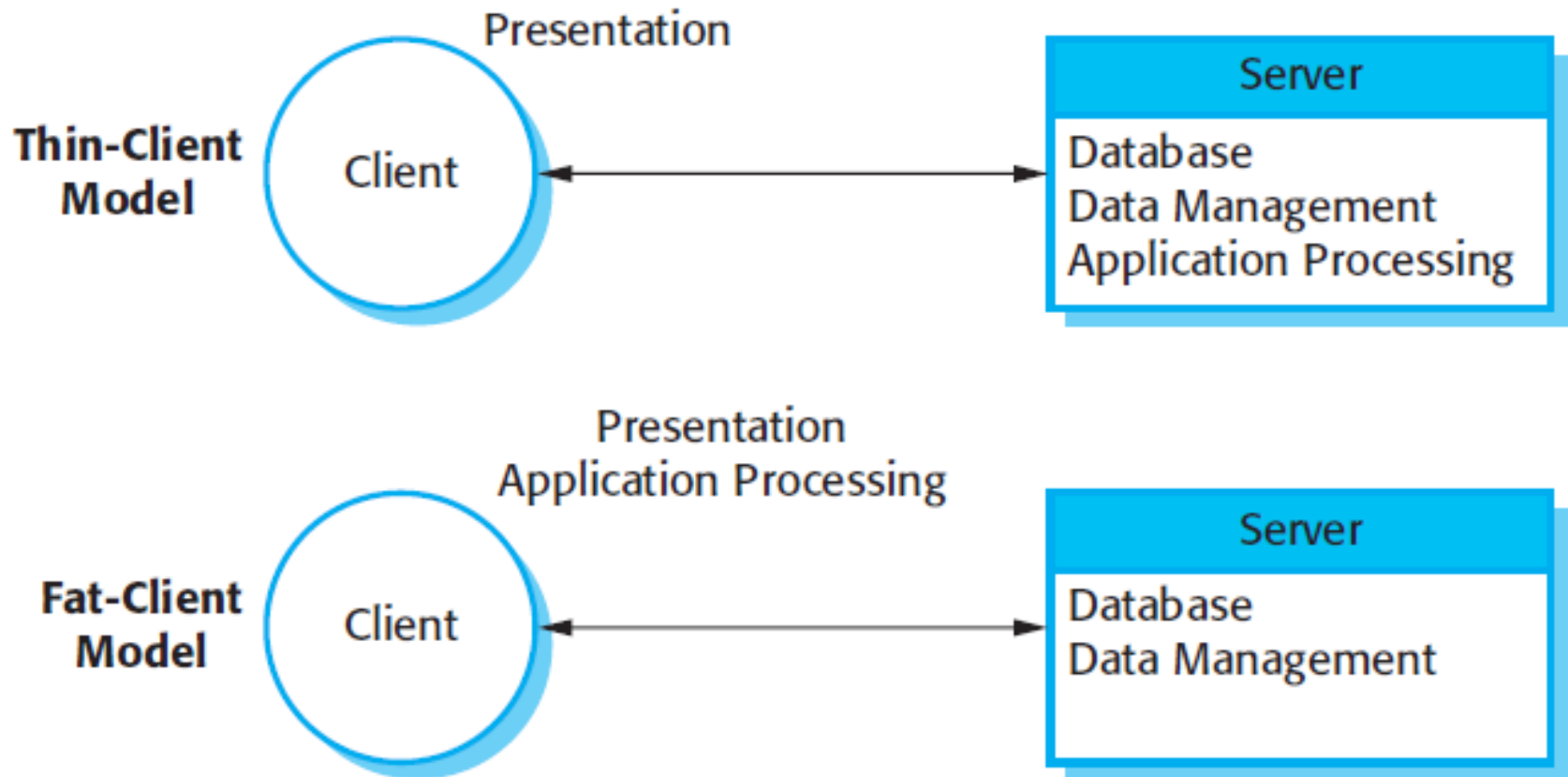
Стил клиент-сървър (client-server)



Стил клиент-сървър

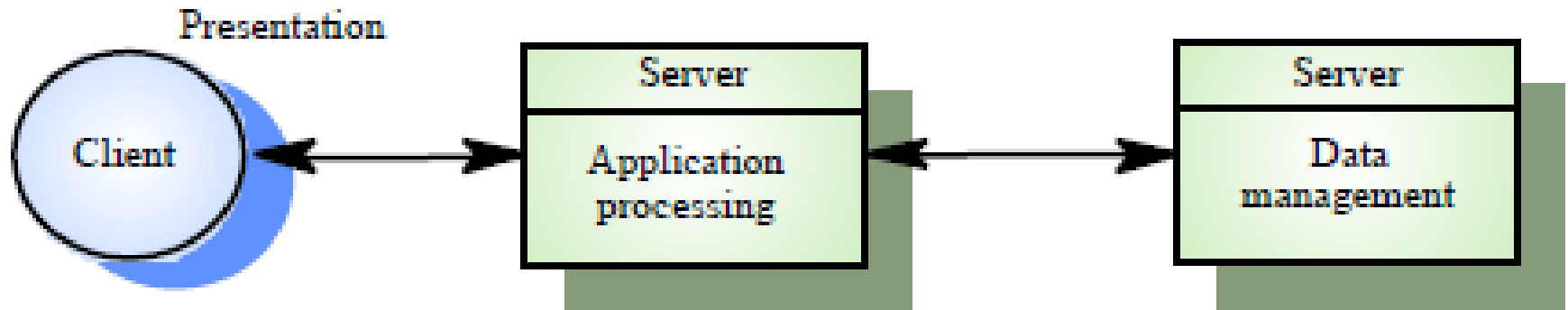
- Системата е проектирана като набор от сървъри, които предлагат услуги и редица клиенти, които използват тези услуги
- За сървърите не е необходимо да имат информация за своите клиенти
- Класическа реализация - *тънък* клиент (*thin client*)
 - Клиентът реализира функционалността на потребителския интерфейс
 - Сървърът реализира функцията за управлението на данни и приложната обработка
- Тежките клиенти (*fat clients*) могат да внедрят част от функционалността за обработка на приложения

Thin или *Fat* клиент-сървър стил



Трислоен клиент/сървър (three-tier client/server) модел

- Трислойният клиент/сървър е модел на многослойна компютърна архитектура, в която цялото приложение се разпределя в три различни изчислителни слоя или нива. Той разделя слоевете на презентация, логика на приложение и обработка на данни на клиентски и сървърни устройства



- Предимства – предлага по-добре:
 - производителност
 - сигурност
 - повторна използваемост
 - преносимост
 - модулност + абстрактност

Предимства на стила клиент-сървър

- Централизация на данните
- Сигурност – при клиента и при сървърите
- Лесна реализация на архивиране (back-up) и възстановяване (recovery)

Недостатъци на стила клиент-сървър

- Работният товар на сървъра може да нарасне прекалено при голям брой сървъри
- Какво правим при отказ (срив) на сървър
 - Необходимост от излишък/отказоустойчивост (redundancy/fault-tolerance)

Слоест (Layered) стил

Application UI

Application logic

Operating system

Device drivers

BIOS

Hardware

Правила за изграждане на layered стил

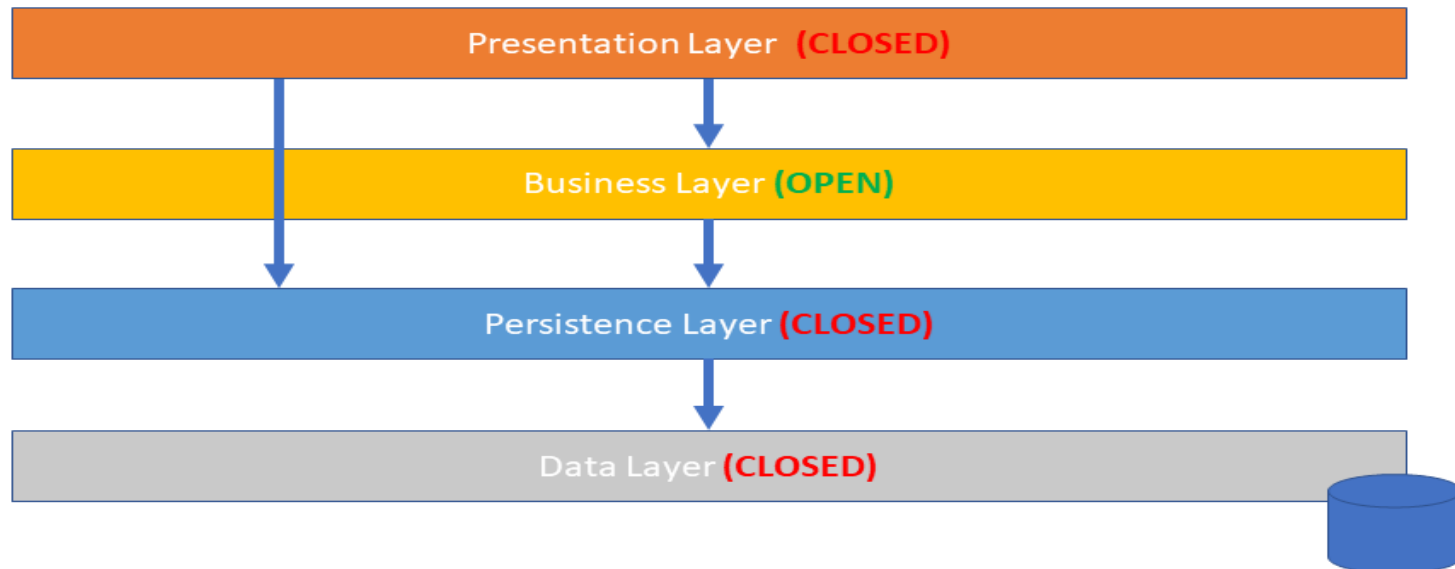
- Представя системата като организирана от йерархично-подредени слоеве
- Класическо изпълнение
 - Всеки слой предлага услуги чрез интерфейс, но само за слоя, който е директно над него и използва услугите от слоя, който е непосредствено под него
 - По този начин всеки слой представлява:
 - Сървър - за слоя по-горе
 - Клиент - за слоя отдолу
 - Интерфейсите може да са подобни на API (Application Programming Interfaces)

Слоест стил

- На практика това е най-разпространеният стил във всички видове софтуерни системи
- Слоестите архитектури са йерархични
- Йерархията намалява комплексността
- Много хора могат да спорят дали клиент-сървър стилът е по-общ от слоестия стил, или е обратното?
- Историческо развитие

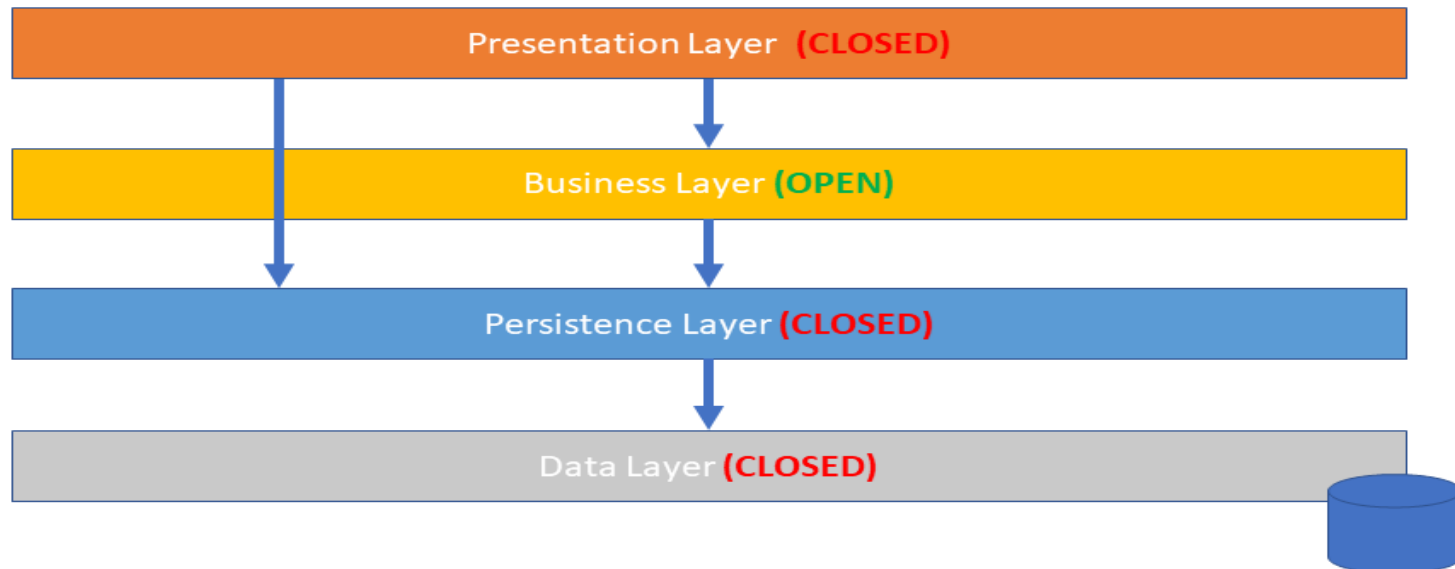
Closed layered architectures vs open layer architectures 1/2

- Затворена слоеста архитектура означава, че всеки даден слой (или ниво) може да използва услугите на следващия непосредствен слой, докато
- в архитектура с отворен слой даден слой може да използва който и да е от слоевете (или нивата) под него.



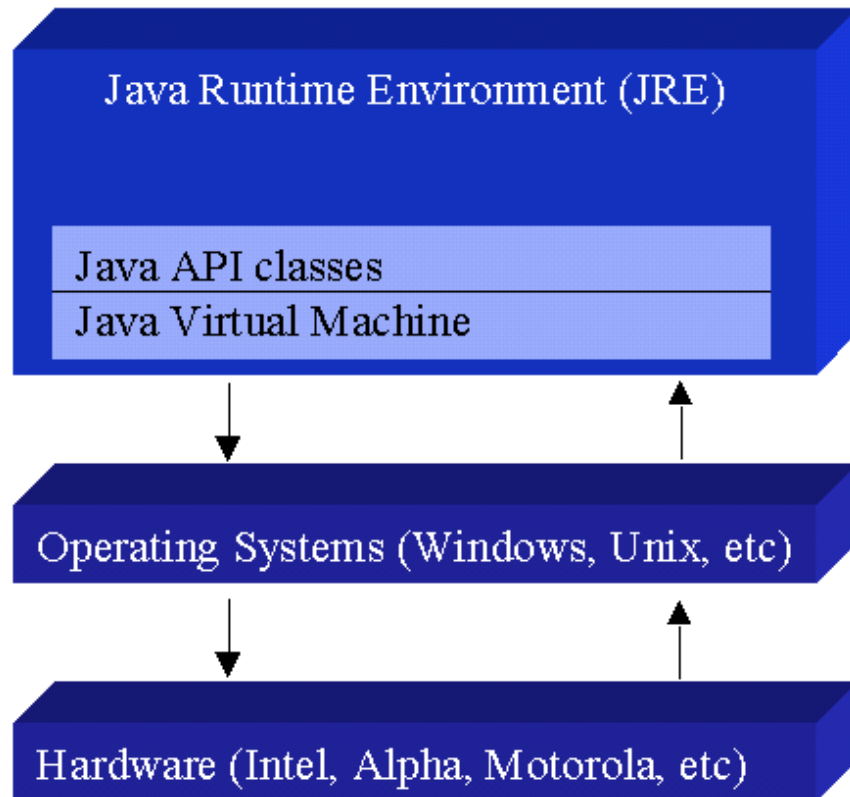
Closed layered architectures vs open layer architectures 2/2

- Затворените слоести архитектури са по-преносими (low coupling),
докато
- Отворените слой слоести архитектури са по-ефикасни.

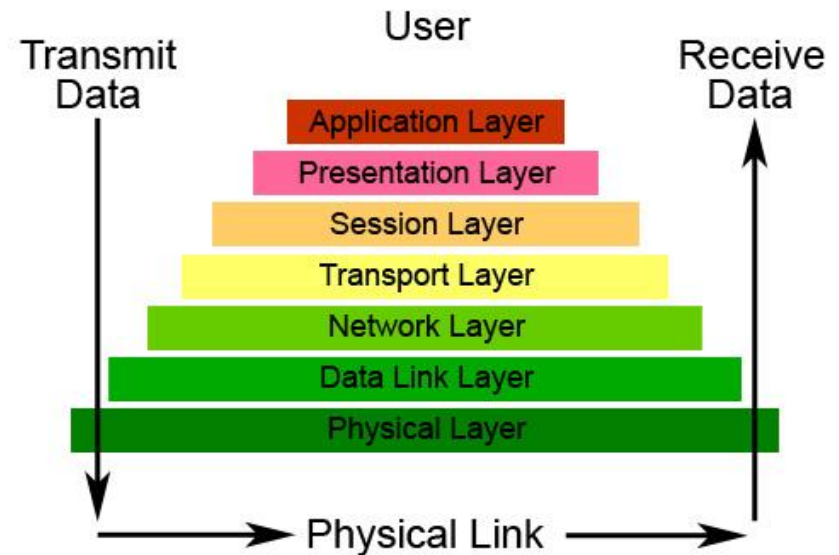


Layered стил - примери

Java applications



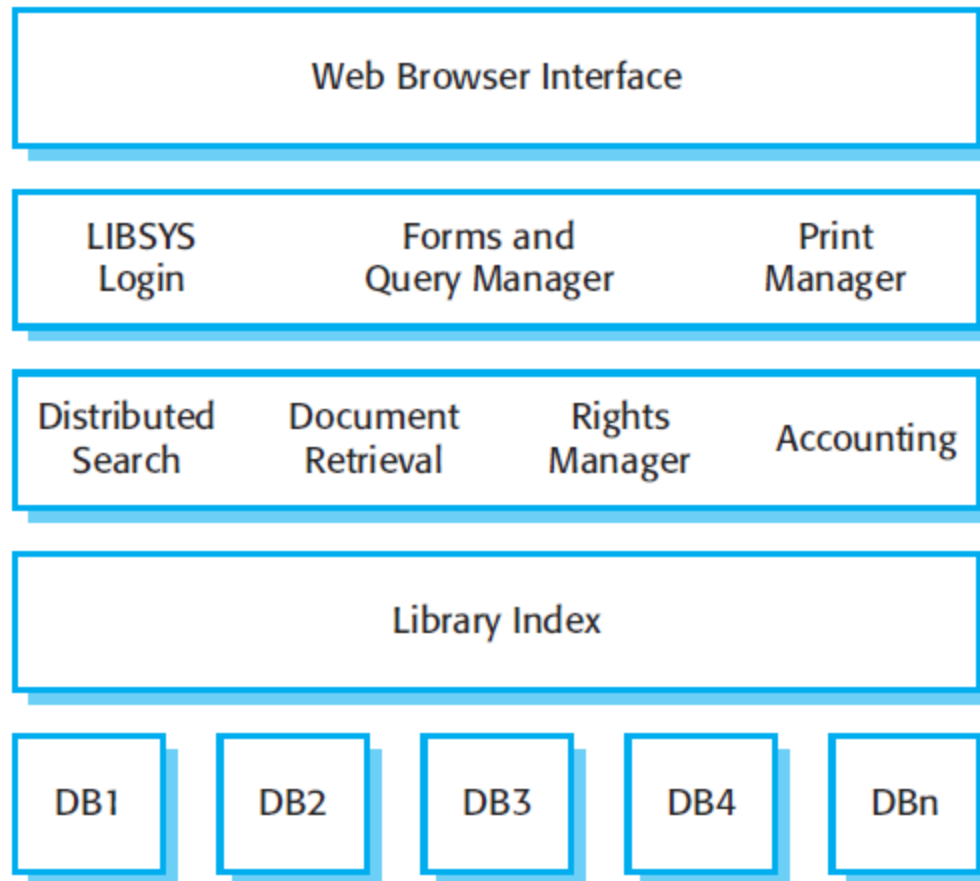
OSI Networking model The Seven Layers of OSI



Source:

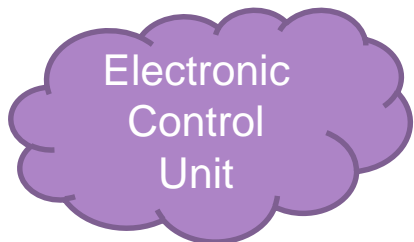
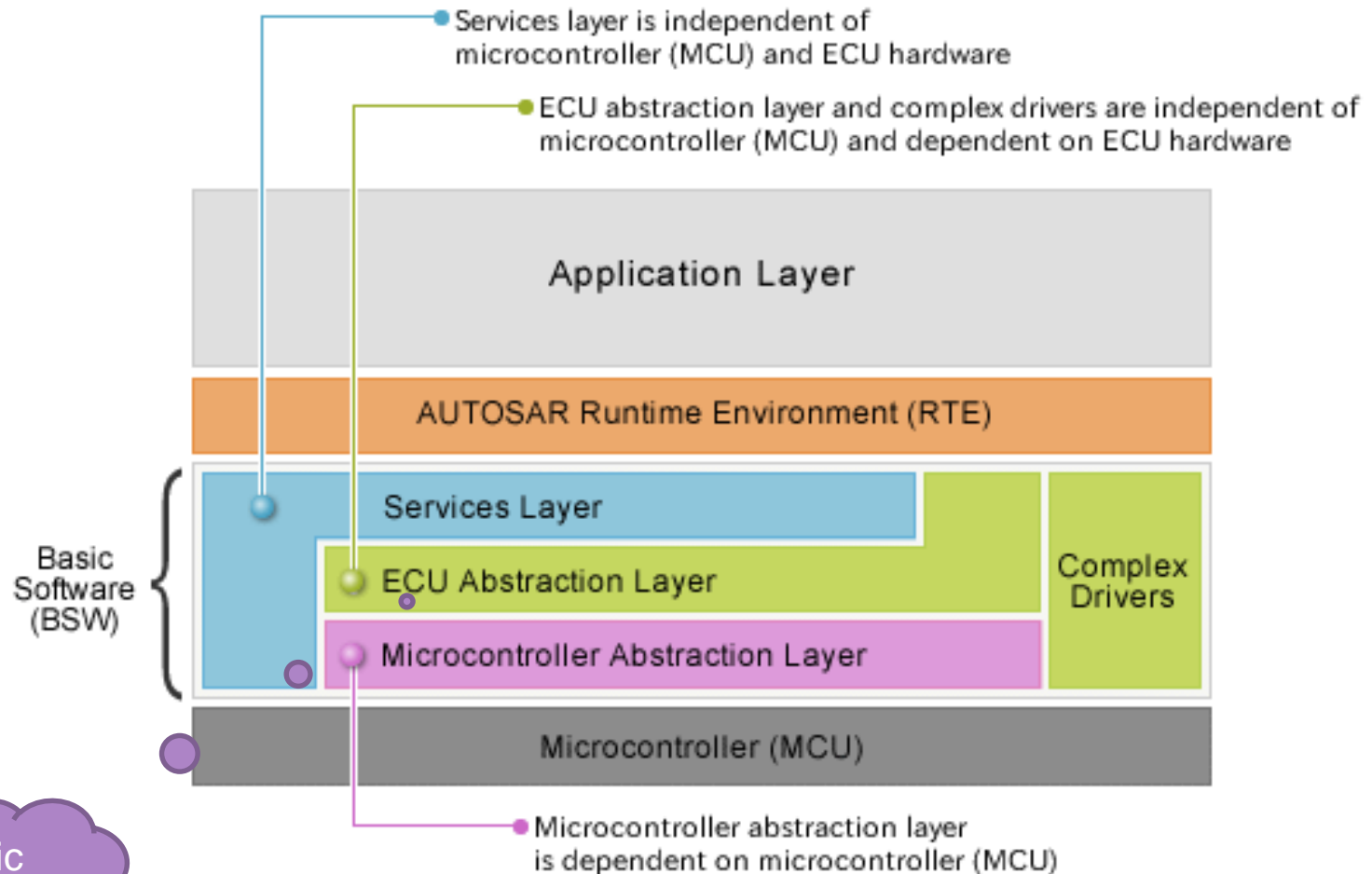
<http://www.windowsnetworking.com/articles-tutorials/common/OSI-Reference-Model-Layer1-hardware.html>

Информационна система за библиотечен обмен



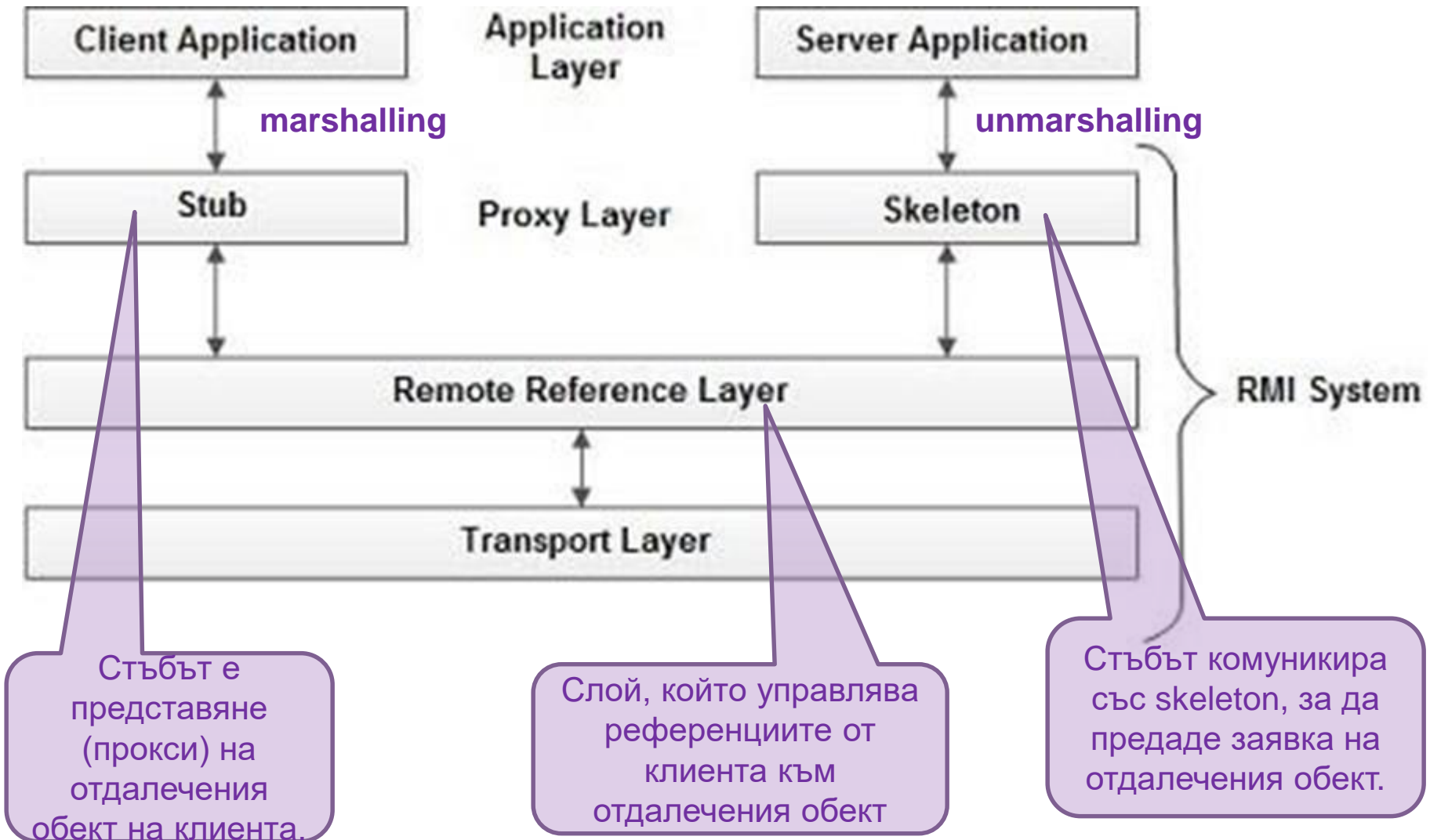
Source: Software Engineering by Ian Sommerville, 9th edition (2010), Addison-Wesley Pub Co;

Вертикални слоеве



Source: AUTOSAR Layered Architecture
http://www.renesas.eu/applications/automotive/technology/autosar/peer/autosar_layerdarch.jsp

Използване на слоеве в RMI архитектура



Предимства на layered стила

- Вътрешната структура на слоевете е скрита, ако се поддържат интерфейси
- Абстракция - минимизиране на сложността
- По-добра кохезия - всеки слой поддържа подобни (функционално-свързани) задачи
- Въвеждането на “stub” слоеве може да подобри тестването

Недостатъци на layered стила

- За много системи е трудно да се разграничат отделни слоеве и това води до увеличаване на усилията за проектиране
- Строгите ограничения за комуникация на слоя компрометират производителността
 - Понякога могат да бъдат използвани вертикални слоеве

Въпроси?

