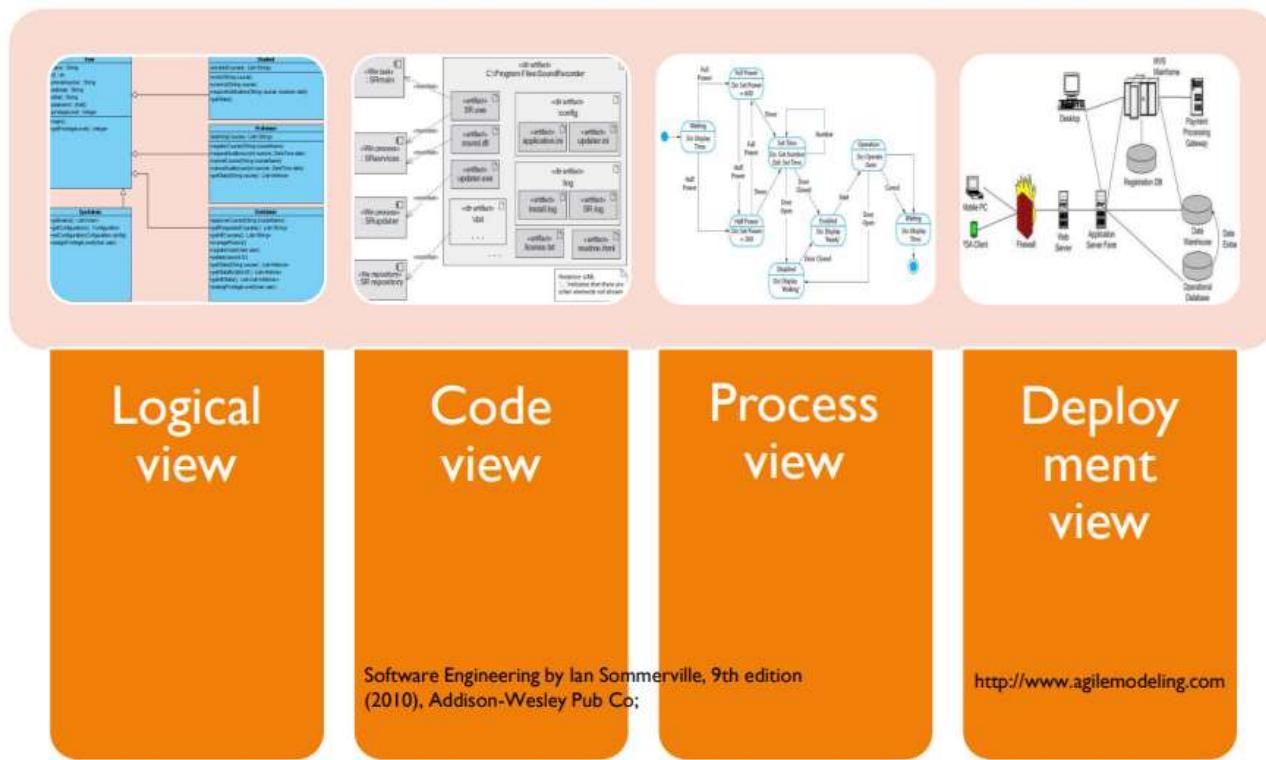


### 3. Архитектурни стилове на софтуерни системи- Резюме

Софтуерна архитектура(СА)-Колекция от структури, които представят различни гледни точки върху системата. Всяка структура се състои от елементи, техните външно видими характеристики и връзките между тях. 4+1 изгледи: (по Kruchten):

- Logical view (Логически изглед)
- Process view (Процесен изглед)
- Development view (Изглед на разработка)
- Deployment view (Изглед на внедряване)
  - Scenarios (Сценарии)



#### Архитектурни стилове:

- **Logical view** на софтуера има 4 нива на абстракция:
  - **Components and connectors** (Компоненти и конектори)
  - Техните интерфейси

- **Architectural configurations** – специфична топология на взаимосвързани компоненти и конектори
- **Architectural styles** – шаблони за успешни и практически доказани архитектурни конфигурации

**Architectural style** дефинира семейство от системи под формата на шаблон за структурна организация. Стиловете определят:

- **Речника от components and connectors**, които могат да се използват в даден стил
- **Набор от ограничения за това как могат да бъдат комбинирани**, например:
  - Топология на описанията (например: без цикли)
  - Семантика на изпълнението (напр. процесите се изпълняват паралелно)

Пример за архитектурен стил: изграждаме софтуер за управление на библиотека. Използваме архитектурния стил "клиент-сървър".

**Компоненти:** В този стил имаме два основни вида компоненти – клиент (например мобилно приложение или уебсайт) и сървър (система, която обработва заявки и съхранява данни). **Свързващи елементи:** Това е начинът, по който клиентът и сървърът комуникират, например чрез HTTP заявки.

**Ограничения:** 1/**Топология:** Клиентът винаги изпраща заявки към сървъра, но не и обратно. Няма цикли в комуникацията.

**2/Семантика на изпълнението:** Клиентът и сървърът могат да работят едновременно (паралелно), но всеки има ясна роля.

Този стил улеснява разработката, защото ясно определя какви компоненти можем да използваме и как трябва да взаимодействват.

### **Architectural Styles:**

- **Pipe-and-Filter style** (Стил „Тръба и филтър“)
- **Layered** (Слоест стил)

- **Client-server** (Клиент-сървър)
- **Repository/Blackboard** (Репозитори/Табло)
- **Model-View-Controller (MVC)**
- **Implicit invocation/ Message passing** (Имплицитно извикване/Предаване на съобщения)

### **Pipe-and-Filter Style (Стил „Тръба и филтър“)**

- Всяка компонента (**filter**) в системата предава данни последователно към следващата компонента.
- Конекторите (**pipes**) между филтрите представляват механизмите за реален трансфер на данни.
- Името „*pipe and filter*“ идва от оригиналната Unix система, където е било възможно да се свързват процеси чрез **pipes**.
- **Pipes** предават текстови потоци от един процес към друг.
- **Filters** представляват изчислителни единици в системата:
  - Четат данни чрез входните си интерфейси
  - Обработват данните
  - Изпращат обработените данни към изходните си интерфейси
- Филтрите нямат информация за съседните си компоненти.
- **Pipes** имат задачата да прехвърлят данните от изхода на един филтър към входа на следващия.

#### **Вариации на Pipe-and-Filter Style:**

- **Batch-sequential** (Партидно-последователен)
- **Parallelism / Redundancy** (Паралелизъм/Излишък) - разклонен
- **Loopback – има цикли**
- В контекста на комуникационен протокол:
  - **Pipeline/Stream** – Обработката на данни може да започне веднага след получаването на **първия байт от filter**.
  - **Batch-sequential style**-необходимо **всички данни да бъдат прехвърлени**, преди filter да започне работа с тях.

### **Предимства of Pipe-and-Filter Style**

- **Интуитивен и лесен за разбиране**
- **Filters** работят самостоятелно и могат да се разглеждат като „**black boxes**“ (черни кутии), което води до гъвкавост по отношение на поддръжка и повторно използване
- Лесна реализация на **паралелност (concurrency)** (не е приложимо при batch-sequential)
- Лесно приложение в структури на бизнес процеси
- Подходящ, когато обработката може да се раздели на **независими стъпки**

## Недостатъци of Pipe-and-Filter Style

- Трудно реализиране на интерактивни приложения поради последователността на стъпките
- Слаба производителност
- Всеки **filter** трябва да извършва парсинг/депарсинг на данни
- Трудност при споделяне на глобални данни
- **Filters** трябва да се съгласуват за формата на данните

## Проблеми при стил „Pipe-and-Filter“

- **Complexity** (Сложност) – В разпределена среда, ако филтрите работят на различни сървъри
- **Reliability** (Надеждност) – Необходима е инфраструктура, гарантираща непрекъснат поток от данни
- **Idempotency** (независимо колко пъти изпълниш една и съща операция, резултатът винаги ще бъде един и същ) – Откриване и премахване на дублиращи се съобщения
- **Контекст и състояние** – Всеки филтър трябва да има достатъчен контекст за изпълнение на задачите си, което може да изисква значителна информация за състоянието

## Shared-Data Style (Стил „Споделени данни“)

- Активно използван в системи, където компонентите трява да прехвърлят големи обеми от данни
- **Shared-data** може да се разглежда като **connector** между компонентите

### Вариации на Shared-Data Style:

- **Blackboard** – При изпращане на данни към **shared-data connector**, всички компоненти се **информират** (активен агент)
- **Repository** – е пасивен, **без известяване** към компонентите

### Предимства на стил „Споделени данни“-Shared-Data Style:

- **Scalability** (Мащабируемост) – Лесно добавяне на нови компоненти
- **Concurrency** (Паралелност) – Всички компоненти могат да работят едновременно
- Ефективен при работа с големи обеми от данни
- Централизирано управление на данните
- Подобрени условия за **сигурност и резервно копиране**
- Независимост на компонентите от източника на данни

### Недостатъци на Shared-Data Style:

- Трудно приложение в разпределени среди
- Поддържане на единен модел на данни
- Промените в модела могат да доведат до допълнителни разходи
- Зависимост между **blackboard** и източниците на знания
- Възможност за „**bottleneck**“ (тясно място) при твърде много клиенти

### Client-Server Style (Клиент-Сървър Стил)

- Системата е изградена от **servers** (сървъри), предлагащи услуги, и **clients** (клиенти), които ги използват
- **Servers** не е нужно да знаят информация за клиентите си
- Класическа реализация – **thin client** (тънък клиент)

- Клиентът управлява потребителския интерфейс
- Сървърът обработва данните и бизнес логиката
- **Fat clients** (дебели клиенти) - изпълняват част от обработката

### Three-Tier Client-Server Model (Три-слоен клиент-сървър):

- По-добра производителност
- По-добра сигурност

**Предимства:** Централизация на данни, Сигурност, Лесно резервно копиране/ back-up / и възстановяване/ recovery /

### Недостатъци:

- Натоварване на сървъра при много клиенти
- Проблеми при отказ на сървъра
- Необходимост от устойчивост на грешки (redundancy/fault-tolerance)

**Layered Style (Слоест Стил)-** най-разпространеният стил: Java applications, OSI Networking model – 7 layers of OSI, които комп. системи използват за комуникация по мрежа

- Представя системата като йерархично организирани слоеве
- Всеки слой предлага услуги на слоя над него и използва услугите на слоя под него

### Rules:

- Всеки слой действа като:
  - **Server** за слоя над него
  - **Client** за слоя под него
- Интерфейсите наподобяват **APIs (Application Programming Interfaces)**

### Предимства:

- Скриване на вътрешната структура на слоевете
- Абстракция – минимизиране на сложността
- Подобрена **кохезия** – всеки слой поддържа сходни задачи
- Подобрено тестване чрез „**stub**“ слоеве

**„Stub“ е временен заместител** на реален слой, който симулира неговото поведение. Използва се, за да тестваш един слой, без да се налага да имаш напълно готови останали части на системата.

### Недостатъци:

- Трудност при разграничаване на отделни слоеве в някои системи
- Ограничения в комуникацията между слоевете може да повлияят на производителността
- Понякога се използват вертикални слоеве

### Object-Oriented Style (Обектно-Ориентиран Стил)

- **Objects** представляват **изчислителни единици**
- Отговорни са за поддържане на вътрешната си интегритет
- Вътрешното представяне е скрито от други обекти
- Свързват се чрез съобщения и/или извиквания на методи

### Предимства:

- **Encapsulation** на данни и логика
- Декомпозиция на системата на взаимодействащи агенти

### Недостатъци:

- Обектите трябва да знаят идентичността на други обекти, за да взаимодействат
- Нежелани ефекти при извикване на методи

**Implicit Invocation Style (Стил „Имплицитно извикване“)** – др. имена: Event-based style, Message passing style

- Компонентите взаимодействат чрез **events** (събития) – получаване или излъчване
- **Event bus** е основният **connector** – Всички компоненти взаимодействат чрез него

### Предимства:

- Ниска свързаност (**loose coupling**)
- Лесна подмяна и повторно използване на компоненти; може компоненти да са хетерогенни
- Подходящ за разпределени системи- events са независим и може да пътува в мрежата
- Security –Лесно проследяване и записване на събития

### Недостатъци:

- Неясна структура на системата
- Трудност при контролиране на последователността на изпълнение на С
- Трудно отстраняване на грешки и дебъгване
- Не е сигурно дали съществува компонент, който да реагира на дадено събитие
- Големи количества данни трудно се пренасят от събития
- Проблеми с надеждността при повреда на **event bus**

**Заключение:** **Architectural styles** представляват полезно знание за решаване на често срещани проблеми в софтуерните системи. Стиловете имат теоретично значение, но рядко се използват изолирано. Дизайнът на системи обикновено комбинира различни стилове. Тази комбинация трябва да отговаря на максимален брой изисквания на потребителите.