الجامعة السورية الخاصة
SYRIAN PRIVATE UNIVERSITY

Topic 1

Software Architecture

General Principles in Software Design and Architecture

Dr. Riad Sonbol

Course Roadmap

- General Principles in Software Design and Architecture: *The Context of Software Architecture, System Decomposition, What is Good Design?*
- Layered Architecture
- Practical Concerns 1: Communication between Layers, Façade Design Pattern, and Dependency Injection
- N-Tier, Client-server, Peer-to-Peer, MVC
- Microservices
- Practical Concerns 2: Message Broker, Related Design Patterns: CQRS, API Gateway, Strangler Pattern, Externalized Configuration, case study
- Clean Architecture: The Hexagonal Architecture, Ports and Adapters
- More Architectural Styles: Repository, Service-Oriented Architecture (SOA), Pipes and Filters

الجامعة السورية الخاصة
SYRIAN PRIVATE UNIVERSITY

General Principles in Software Design and Architecture

"The Context of Software Architecture"

Software Crisis

- The term 'software engineering' was suggested at conferences organized by NATO in **1968 and 1969 to discuss the 'software crisis'**.
- Projects running
  - over-budget,
  - over-time,
  - with low quality,
  - and without meeting requirements

50+ Years After the Crisis

MODERN RESOLUTION FOR ALL PROJECTS

	2011	2012	2013	2014	2015
SUCCESSFUL	29%	27%	31%	28%	29%
CHALLENGED	49%	56%	50%	55%	52%
FAILED	22%	17%	19%	17%	19%

The Modern Resolution (OnTime, OnBudget, with a satisfactory result) of all software projects from FY2011–2015 within the new CHAOS database. Please note that for the rest of this report CHAOS Resolution will refer to the Modern Resolution definition not the Traditional Resolution definition.

these problems. The software crisis has been slowly fizzling out, because it is unrealistic to remain in crisis mode for more than 20 years. Software engineers are accepting that the problems of software engineering are truly difficult and only hard work over many decades can solve them

Deshpande Shweta - The Ohio State University, 2011.

50+ Years After the Crisis

- Even in NASA!
- Mars Polar Lander
  - 1998
  - 110 million USD!

The cause of the communication loss is not known. However, the Failure Review Board concluded that the most likely cause of the mishap was a ***SOFTWARE BUG*** *that incorrectly identified vibrations, caused by the deployment of the stowed legs, as surface touchdown. The resulting action by the spacecraft was the shutdown of the descent engines, while still likely 40 meters above the surface. Although it was known that leg deployment could create the false indication, the software's design instructions did not account for that eventuality*

From the "Report of the Loss of the Mars Polar Lander and Deep Space 2 Missions -- JPL Special Review Board (Casani Report) - March 2000".

- Larger List of well-know software bugs:
  - https://en.wikipedia.org/wiki/List_of_software_bugs

But.. why software projects fails?

Complexity & Changes

But.. why software projects fails?

when errors are discovered

Many functions

when the developers have a better understanding of the application.

Many participants from different disciplines

Complexity & Changes

The time between technological changes is often shorter than the duration of the project.

Many systems are so hard to understand (even during their development phase)

No idea what to build

The widespread assumptions of a software project manager that all changes have been dealt with and that the requirements can be frozen

So.. how to make software projects succeed!

- Very Big Question! But these are some of the main research directions:
  - More abstract programming languages!
    - OOP, High level languages, UML…
    - But still “the complexity is in the problem, not in the solution”
  - Better software design
    - Modularity, Abstraction, Anticipation of change, etc
  - Formal methods:
    - Mathematical approaches to solve software problems!
    - Eliminate human error by proving that our software is “correct”!
  - Development processes:
    - Waterfall development, Spiral development, Agile, etc.

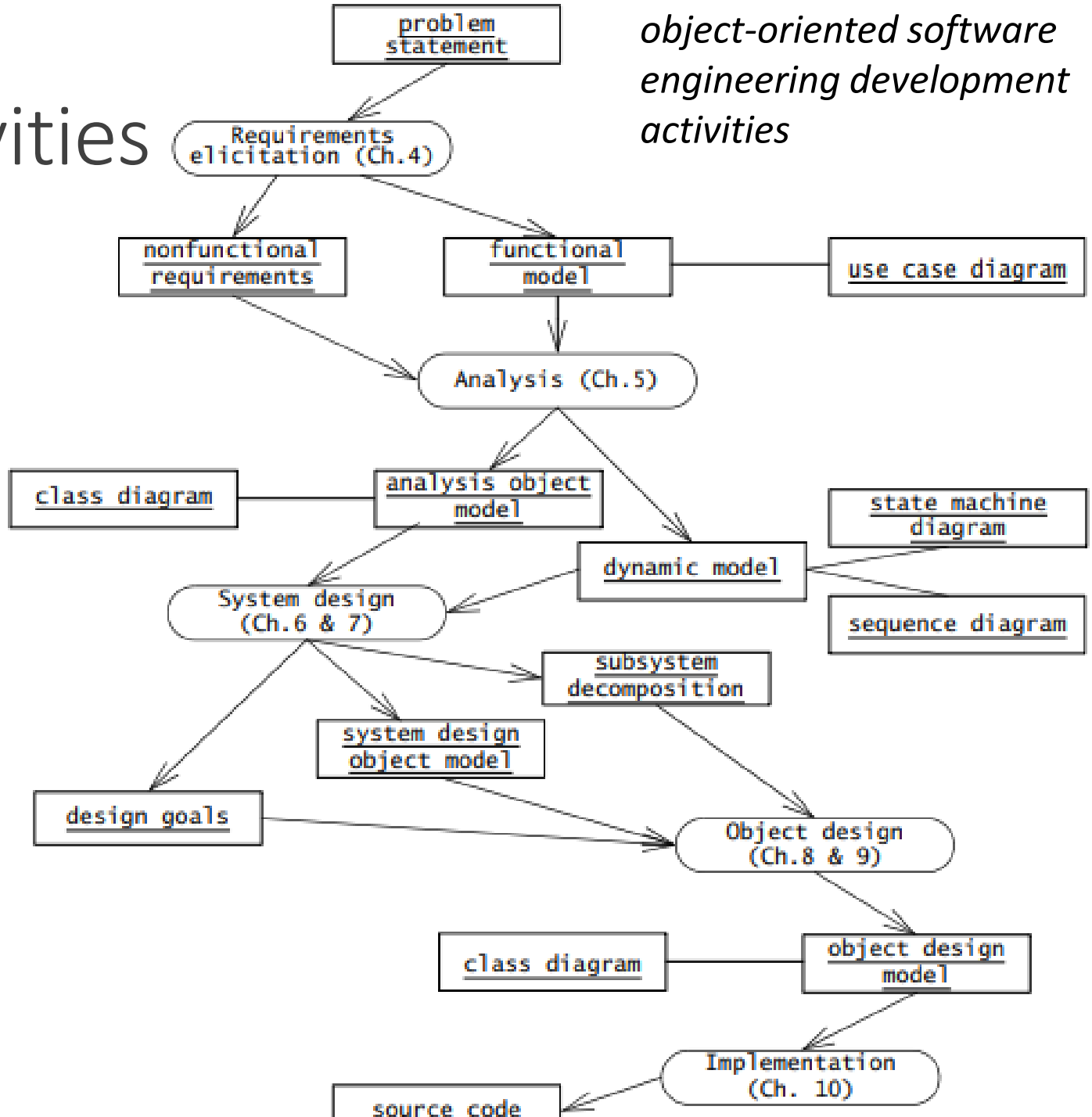
the main focus of this course

The Context: SE Development Activities

1) Requirements Elicitation.
2) Analysis.
3) System Design.
4) Object Design.
5) Implementation.
6) Testing.

- Object-oriented software engineering is **iterative**; that is, activities can occur in parallel and more than once

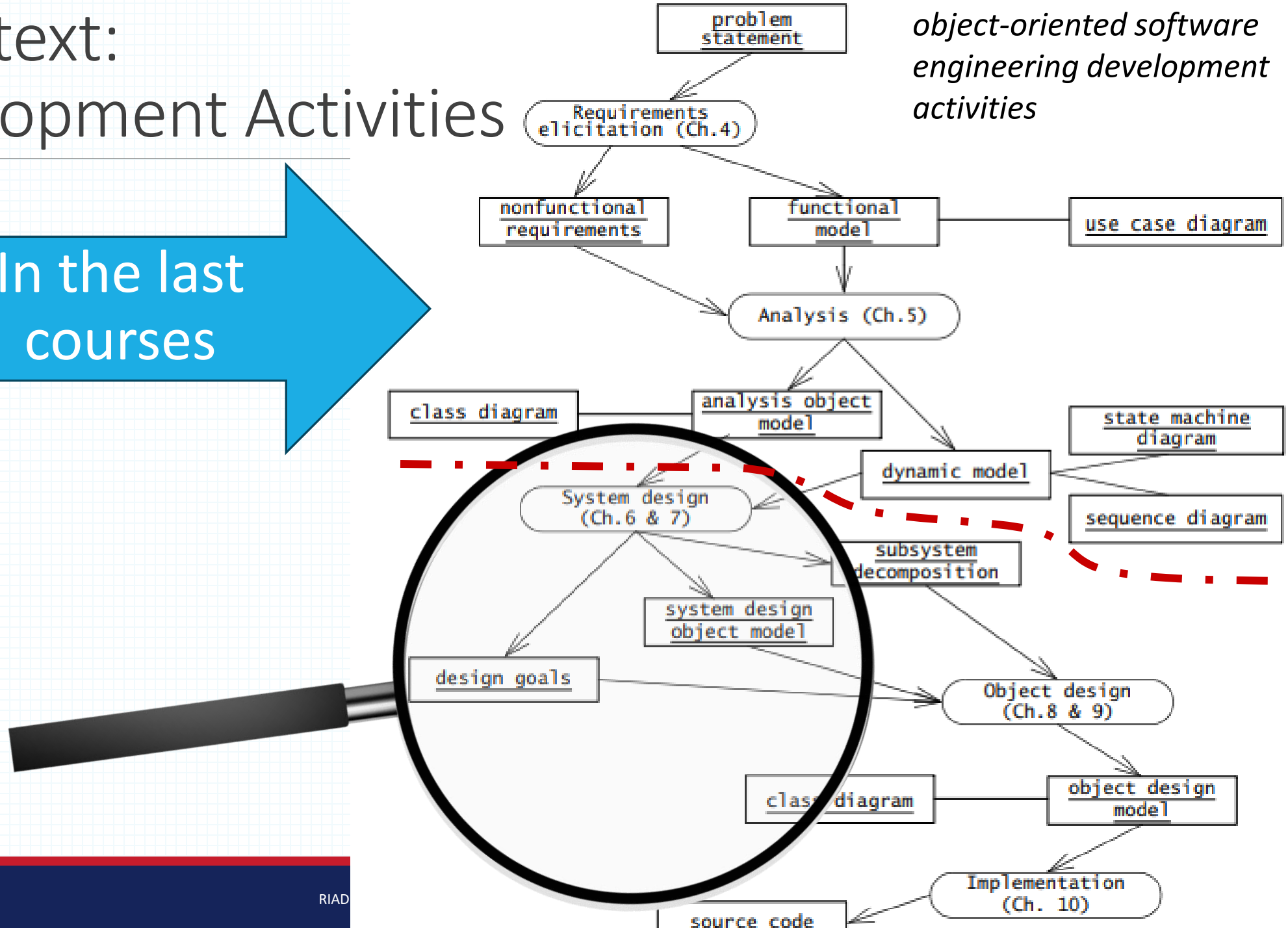
object-oriented software engineering development activities



The Context: SE Development Activities

object-oriented software engineering development activities

In the last courses



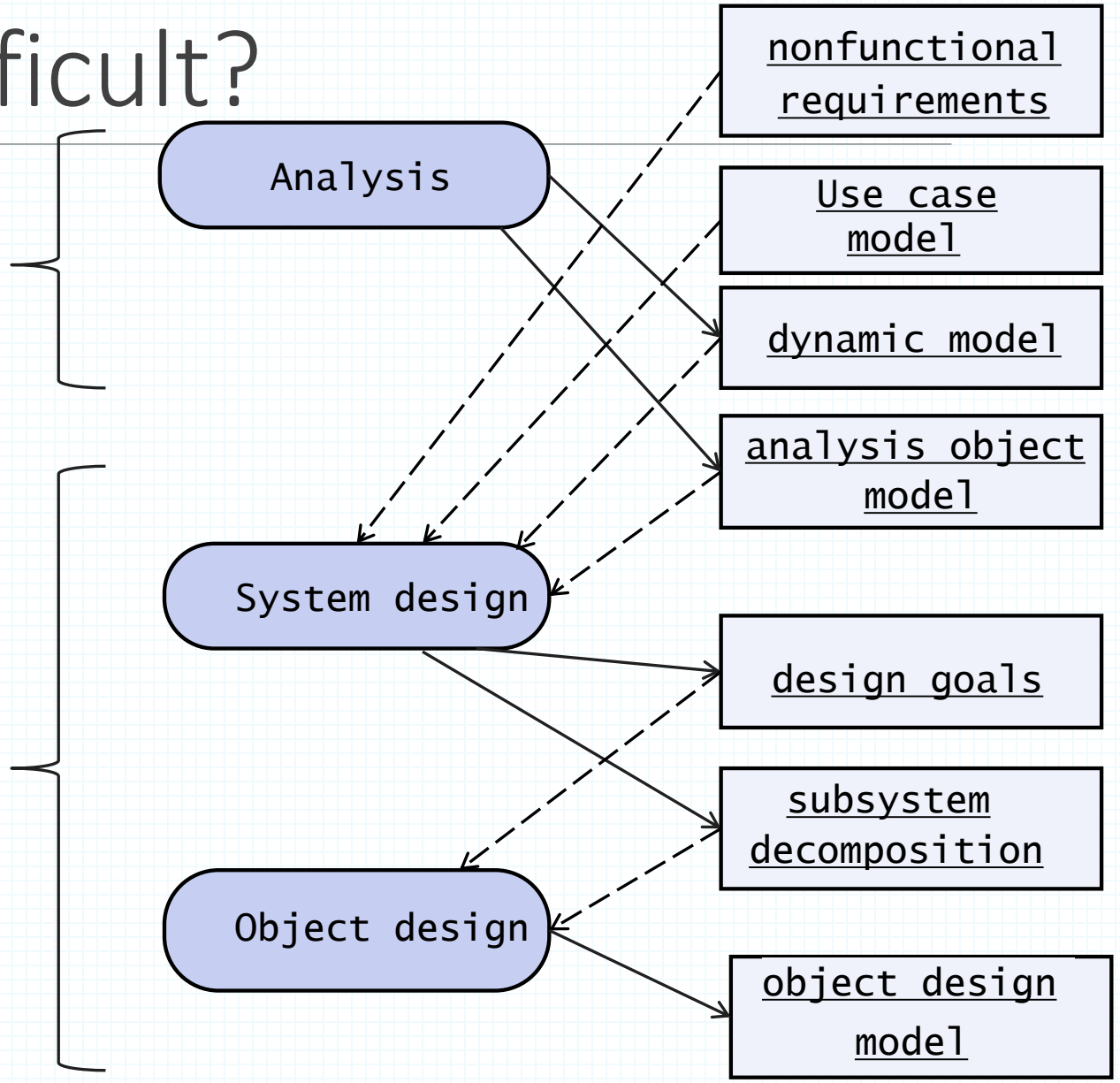
Why is Design so Difficult?

Focuses on the application domain

Focuses on the solution domain

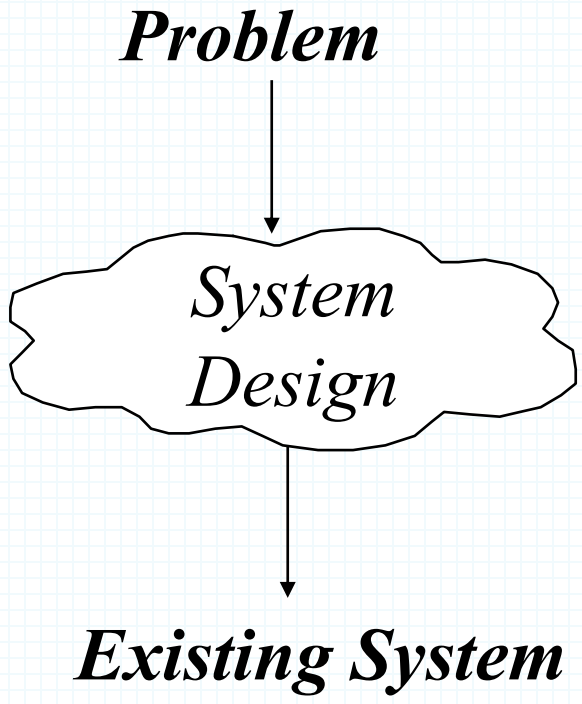
The solution domain is changing very rapidly

- *Halftime knowledge in SE ≈ 3-5 years*
- *Cost of hardware rapidly sinking*
- *Design knowledge is a moving target*



Software Architecture Definition

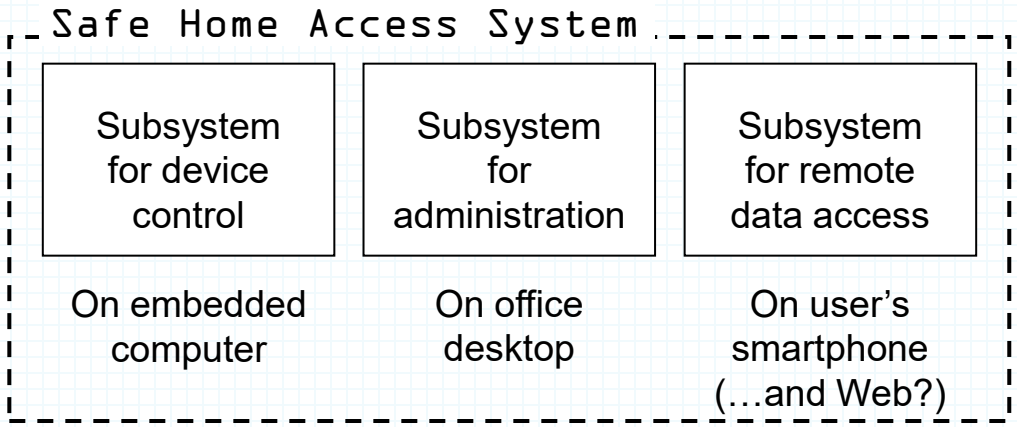
- Software Architecture =
  a set of high-level **decisions** that determine the structure of the solution
  (parts of system-to-be and their relationships)
- Bridge the gap between a problem and an existing system in a manageable way



BUT What are these decisions?

Software is not a long list of program statements but it has structure

Example Architectural Decisions



Example decisions:

- *Decision on system decomposition*
- *Decision on mapping software-to-hardware*
- *Decision on development platform or operating system (Android vs. iOS, etc.)*
- *Decision on how to fit the subsystems together ("architecture style")*

الجامعة السورية الخاصة
SYRIAN PRIVATE UNIVERSITY

General Principles in Software Design and Architecture

"System Decomposition"

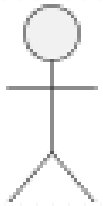
Why We Want To Decompose Systems

- Tackle complexity by “divide-and-conquer”
- See if some parts already exist & can be reused
- Support flexibility and future evolution by decoupling unrelated parts, so each can evolve separately (“separation of concerns”)

Why We Want To Decompose Systems

The system is designed to scrape social media data, perform in-depth analysis, and predict specific product limitations based on user reviews. It then presents the findings through comprehensive reports and charts, effectively illustrating the results. Users can access their accounts, enter their products info (including their related social media pages) and use various functionalities either from a mobile app or a website.

Why We Want To Decompose Systems



Mobile App

Web pages

?

Main Business Features and services

?

social media data scraper

Analyze data and predict limitations

User Mgmt.

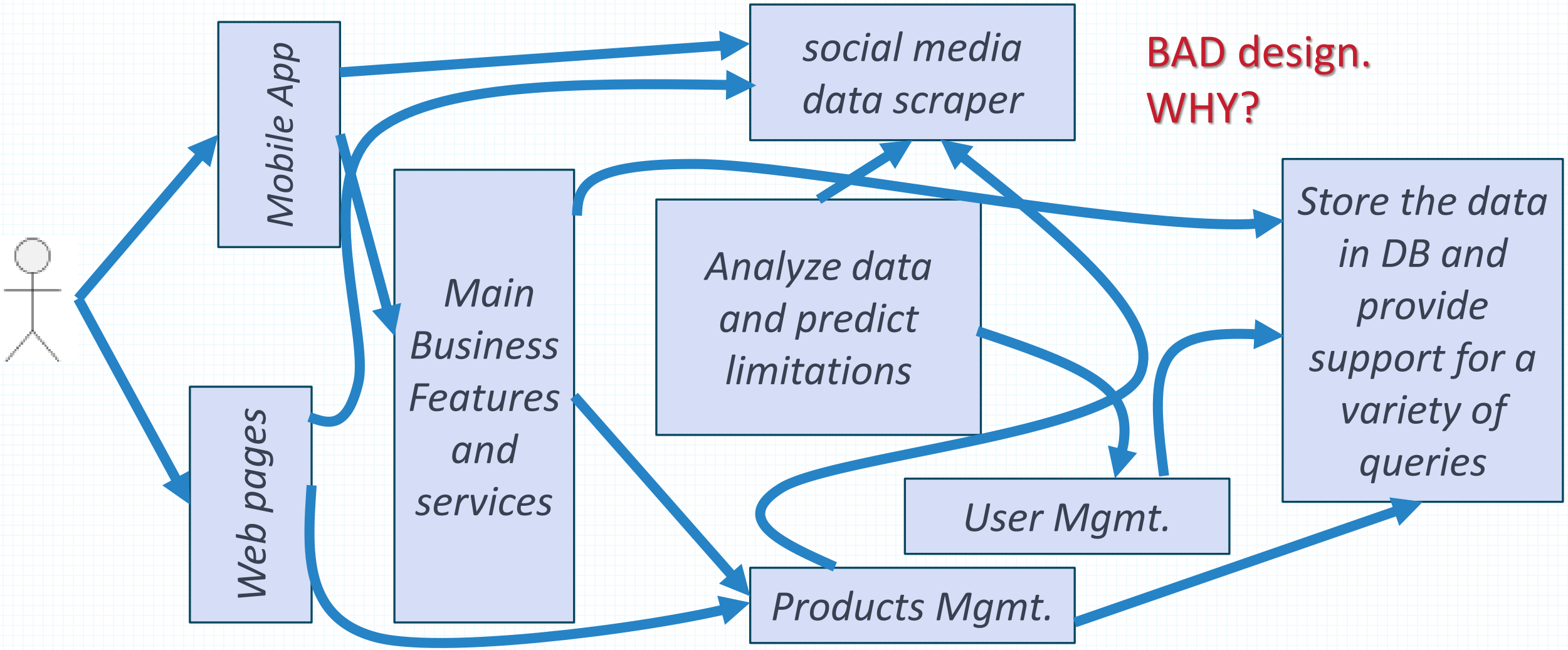
Products Mgmt.

?

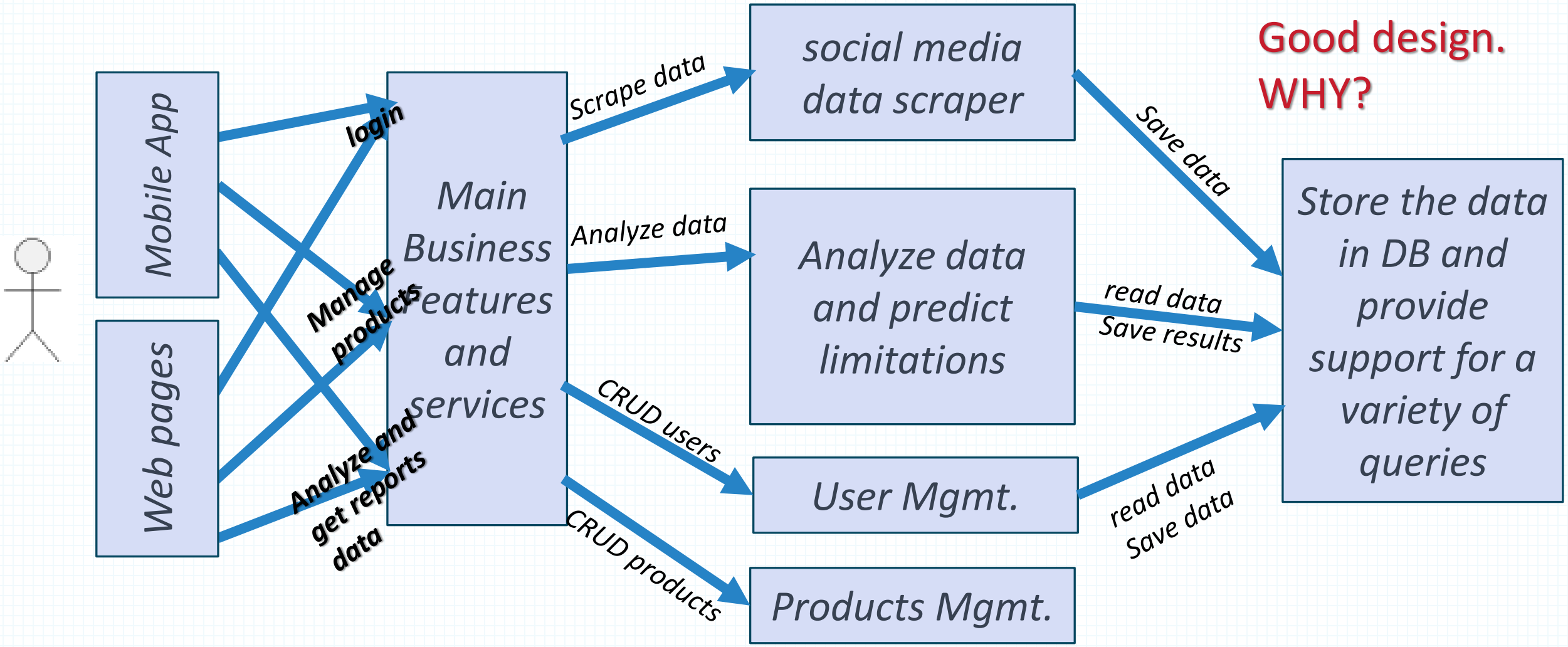
Store the data in DB and provide support for a variety of queries

But How can we connect these elements?

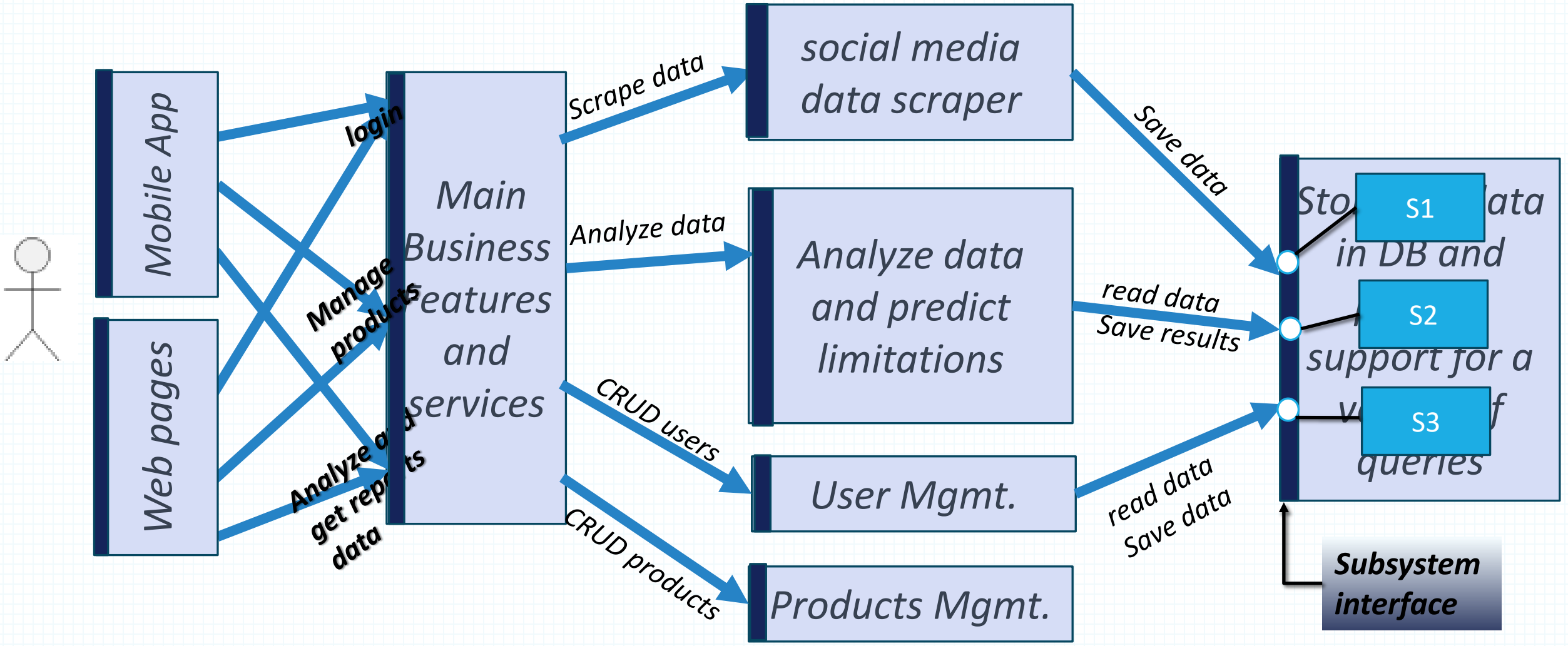
Why We Want To Decompose Systems



Why We Want To Decompose Systems



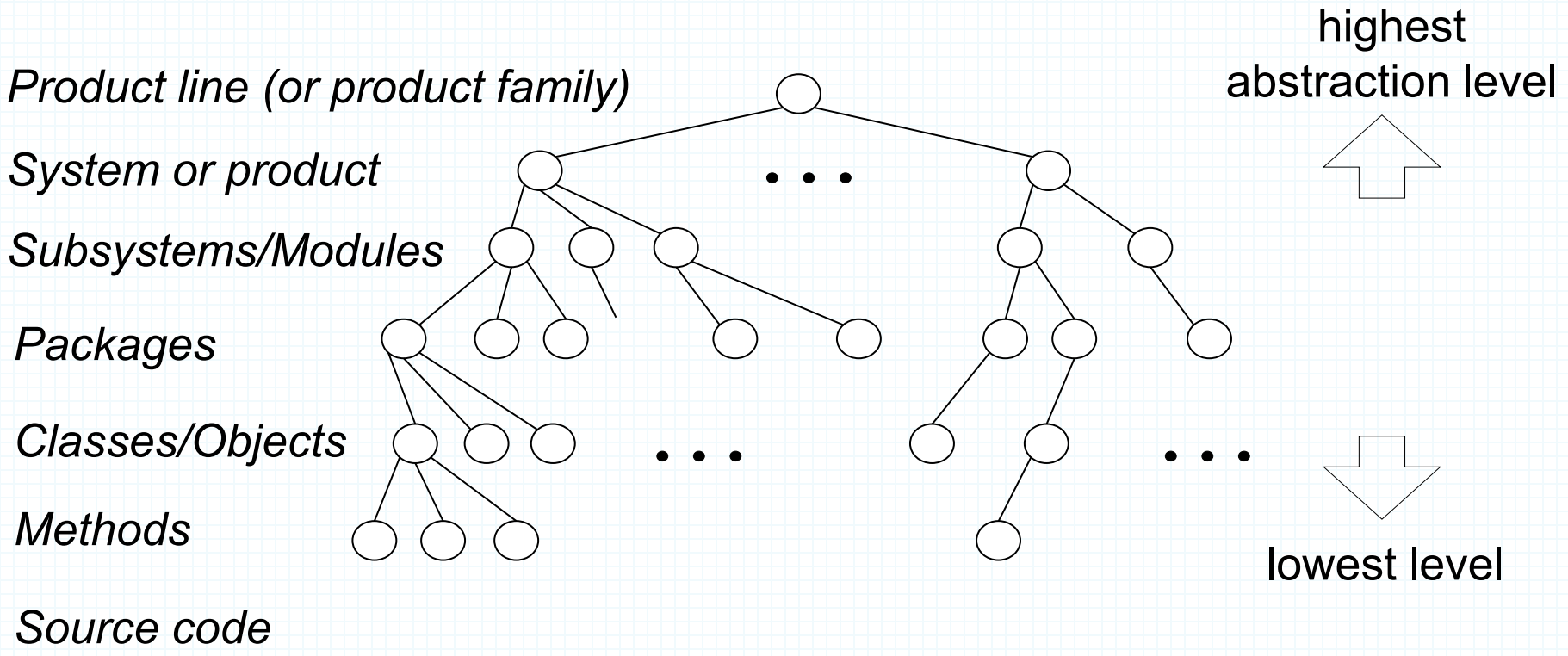
Why We Want To Decompose Systems



System Design Concepts

- A **subsystem** is a replaceable part of the system with well-defined interfaces that encapsulates the *state* and *behavior* of its contained classes.
- A **service** is a set of named operations that share a common purpose. The “seed” for services are the use cases from the functional model.
- **Subsystem interface** specifies the interaction and information flow from and to subsystem boundaries, but not inside the subsystem.
  - defined during design stage.
- **Application programmer’s interface (API)** is the specification of the subsystem interface in a specific programming language
  - defined during implementation stage.

Taxonomy


Product line (or product family)
System or product
Subsystems/Modules
Packages
Classes/Objects
Methods
Source code
highest
abstraction level
lowest level

Layers and Partitions

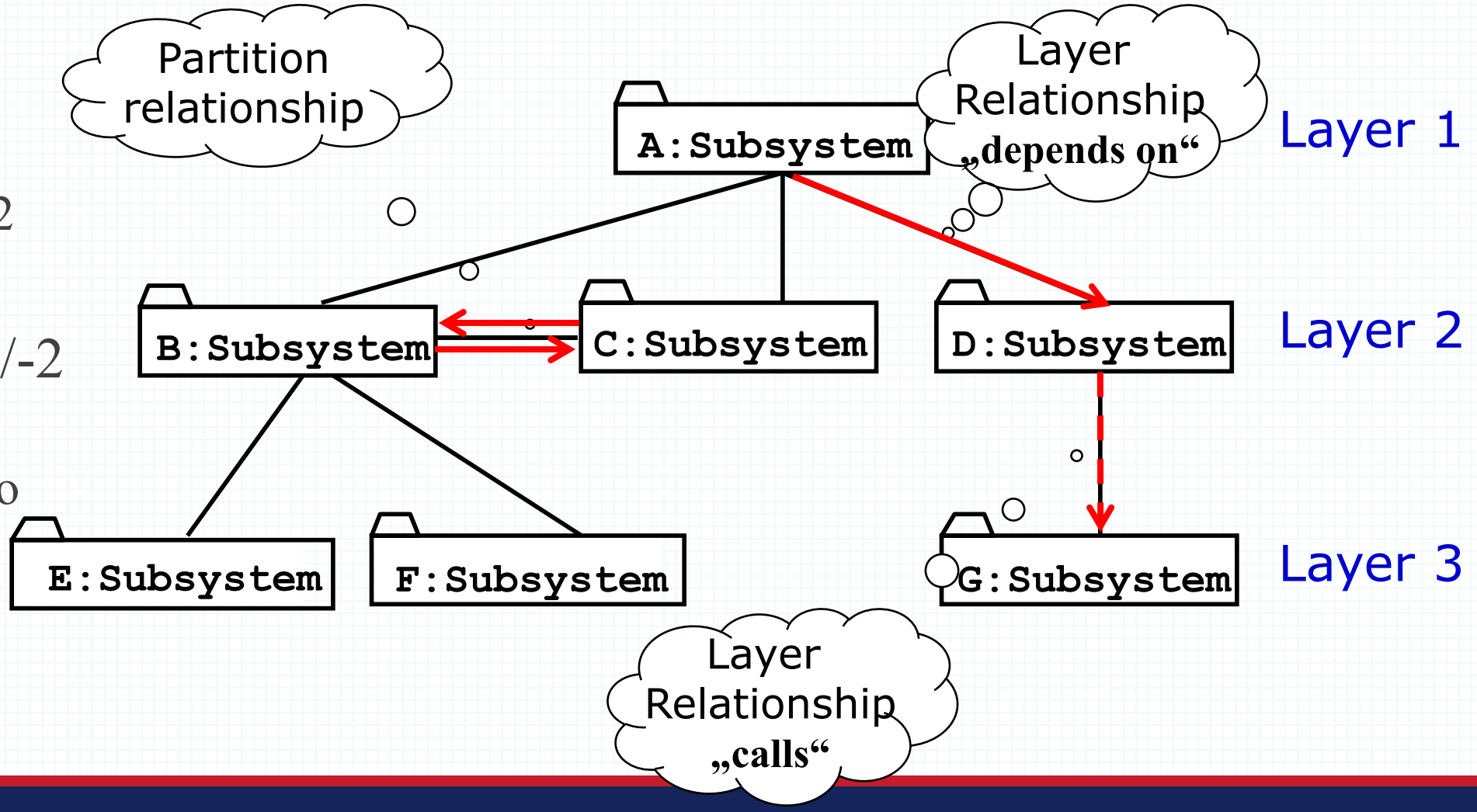
- A layer is a subsystem that provides a service to another subsystem with the following restrictions:
  - A layer only depends on services from lower layers
  - A layer has no knowledge of higher layers
- A layer can be divided horizontally into several independent subsystems called partitions
  - Partitions provide services to other partitions on the same layer
  - Partitions are also called “weakly coupled” subsystems.

Relationships between Subsystems

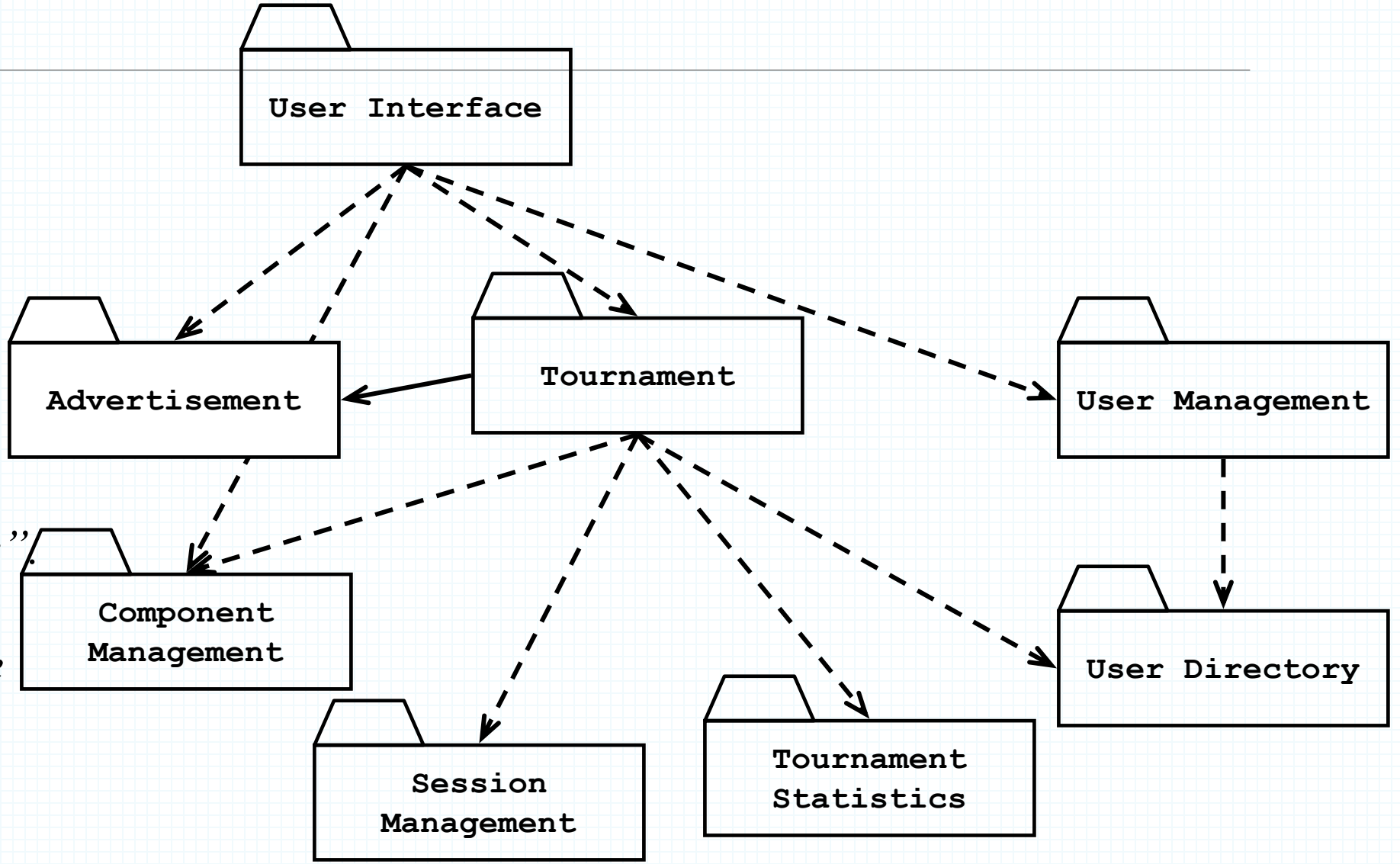
- Two major types of Layer relationships
  - Layer A "depends on" Layer B (compile time dependency)
    - Example: Build dependencies (make, ant, maven)
  - Layer A "calls" Layer B (runtime dependency)
    - Example: A web browser calls a web server
    - Can the client and server layers run on the same machine?
      - Yes, they are layers, not processor nodes
- Partition relationship
  - The subsystems have mutual knowledge about each other
    - A calls services in B; B calls services in A (Peer-to-Peer)

Example of a Subsystem Decomposition

- Subsystem Decomposition Heuristics
  - No more than 7+/-2 subsystems
- No more than 4+/-2 layers
  - Good design: Try to find 3 layers!



Example



New type of association:

- *The dashed line means "depends on at runtime".*
- *The solid line means "depends on at compile time".*

الجامعة السورية الخاصة
SYRIAN PRIVATE UNIVERSITY

General Principles in Software Design and Architecture

"What is Good Design?"

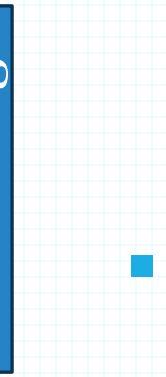
Separation of Concerns (SoC)

- Separation of Concerns (SoC) is a design principle that manages complexity by partitioning the software system so that each partition is responsible for a separate concern.

minimizing the overlap of concerns as much as possible

Coupling and Coherence of Subsystems

- Goal: Reduce system complexity while allowing change
- **Coherence** measures dependency among classes
  - High coherence: The classes in the subsystem perform similar tasks and are related to each other via many associations
  - Low coherence: Lots of miscellaneous and auxiliary classes, almost no associations



- **Coupling** measures dependency among subsystems
  - High coupling: Changes to one subsystem will have high impact on the other subsystem
  - Low coupling: A change in one subsystem does not affect any other subsystem.

Coupling and Coherence of Subsystems

- Goal: Reduce system complexity while allowing change
- **Coherence** measures dependency among classes
  - High coherence can be achieved if most of the interaction is within subsystems, rather than across subsystem boundaries

Good Design

associations

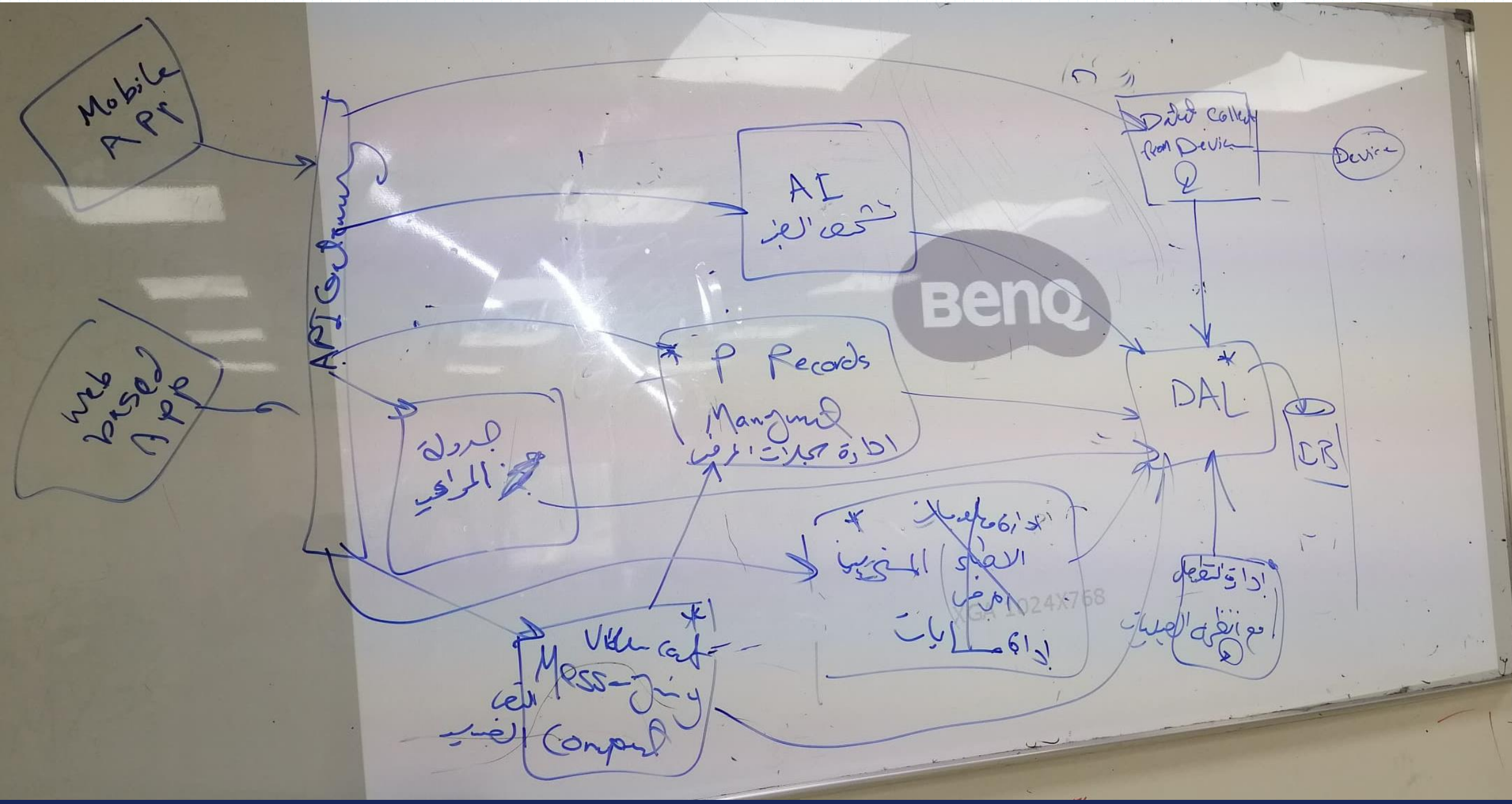
- **Coupling** measures dependency among subsystems
  - Low coupling can be achieved if a calling class does not need to know anything about the internals of the called class (Principle of information hiding)

Case Study

حالة عملية للنقاش

شركة تطوير تعمل على بناء نظام برمجي لإدارة خدمات الرعاية الصحية عن بُعد، ويتميز النظام المراد بالخصائص التالية:

- يتكامل النظام مع أجهزة قياس العلامات الحيوية للمرضى لتحصيل البيانات والقياسات الحيوية مباشرة لتحليلها.
- يوفر واجهة مستخدم تفاعلية للأطباء والممرضين تتيح لهم مراقبة الحالة الصحية للمرضى بشكل لحظي عند الحاجة وتقديم الاستشارات عبر الإنترنت.
- يتيح النظام للمرضى حجز المواعيد والتواصل مع الأطباء عبر المحادثات النصية أو مكالمات الفيديو.
- يقدم النظام توصيات طبية مخصصة بناءً على بيانات المرضى وسجلهم الصحي باستخدام تقنيات الذكاء الصنعي.
- يوفر النظام خدمة إدارة الوصفات الطبية الإلكترونية، حيث يمكن للأطباء إصدار الوصفات وتحويلها مباشرة إلى الصيدليات.
- يدعم النظام تخزين الملفات الطبية بشكل آمن مع إمكانية مشاركتها مع مقدمي الرعاية الصحية بناءً على موافقة المريض.


Mobile APP
Web based APP
API Gateway
AI
Data collect from Device
Device
P Records Manager
DAL
CR
Messaging Component