CSEN603 – Software Engineering

Lecture 1: Introduction

Mervat Abuelkheir
Ammar Yasser
Mohamed Agamia
Nada Hisham

About The Course

What is Software?

Computer **programs** and their associated **documentation**

- Product that software professionals build and then support over the long term
- **Elements**:
  - Executable programs
  - Data associated with these programs
  - Documents: user requirements, design
- Documents, user/programmer guides
- May be developed for a particular customer or may be developed for a general market
  - **Generic** software products
  - **Customized** software products

About The Course

What is Software?

- System software
  - OS, compilers, device drivers
- Business software
  - Payroll, accounting
- Engineering/scientific software
  - Computer-aided design, simulation
- Embedded software
  - GPS navigation, Flight control, Fridge
- Product-line software (PC-like based)
  - Spreadsheets, word processing, games
- Web-based software
  - Gmail, Facebook, YouTube
- Artificial intelligence software
  - Robotics, artificial neural networks

Hardware and Software – Comparison

Hardware

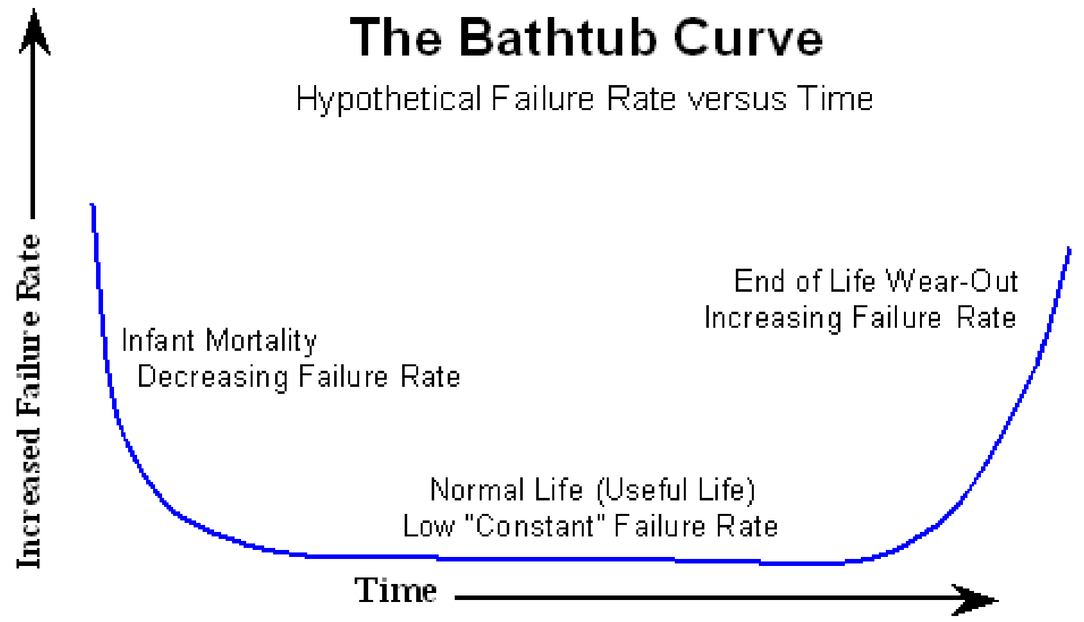
- Standardized components
- Difficult or impossible to modify
- Hiring more people causes more work done
- Costs are more concentrated on products
- Wears out

Software

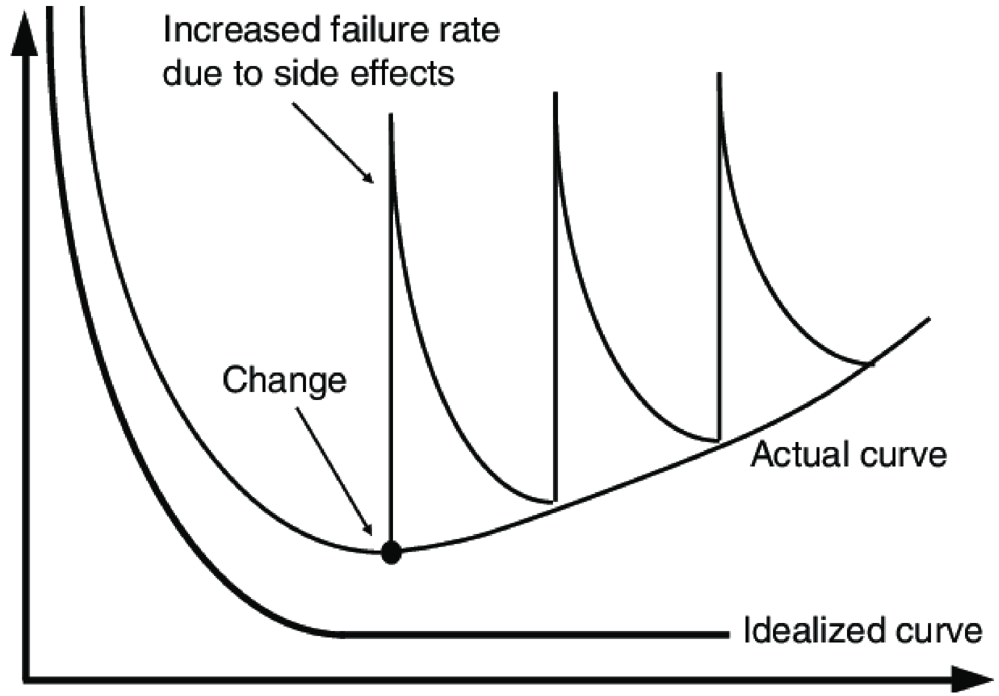
- Custom built
- Routinely modified and upgraded
- Hiring more people does not necessarily cause more work done
- Costs are more concentrated on design
- Deteriorate over time

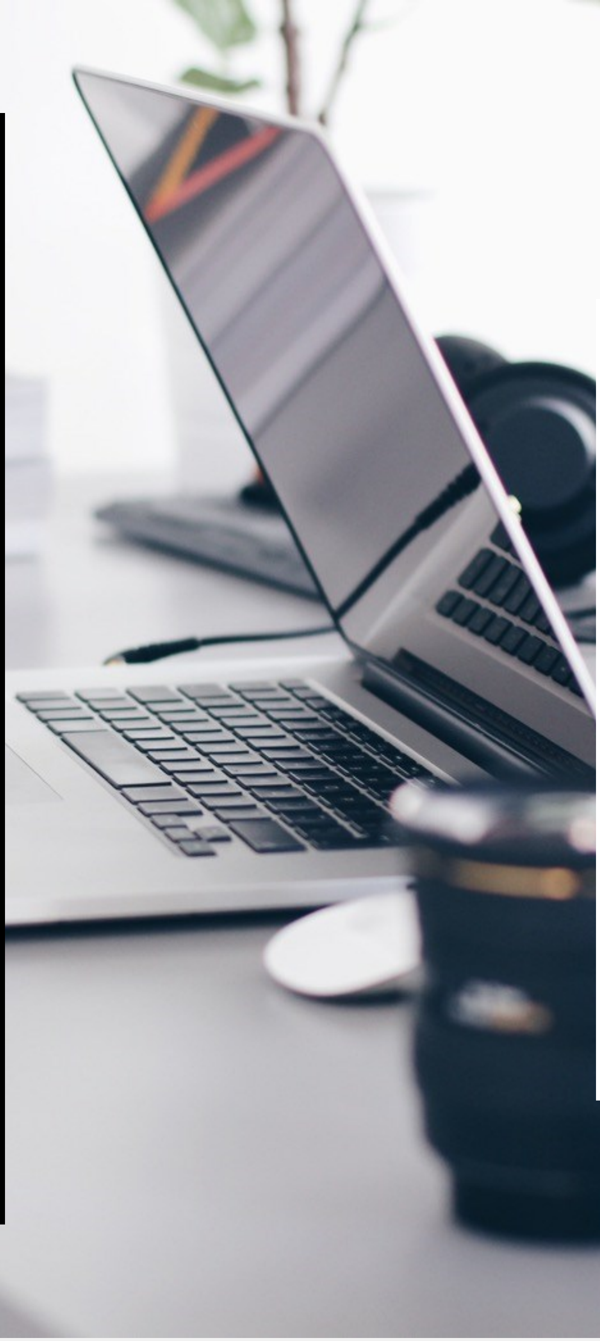
Comparison – Wear and Deterioration

Hardware Failure Curve



Software Failure Curve





Software Engineering **IS NOT** Programming

It is fairly easy to write computer programs without using software engineering methods and techniques

Many companies have drifted into software development as their products and services have evolved. They do not use software engineering methods in their everyday work. Consequently, their software is often more expensive and less reliable than it should be

About The Course

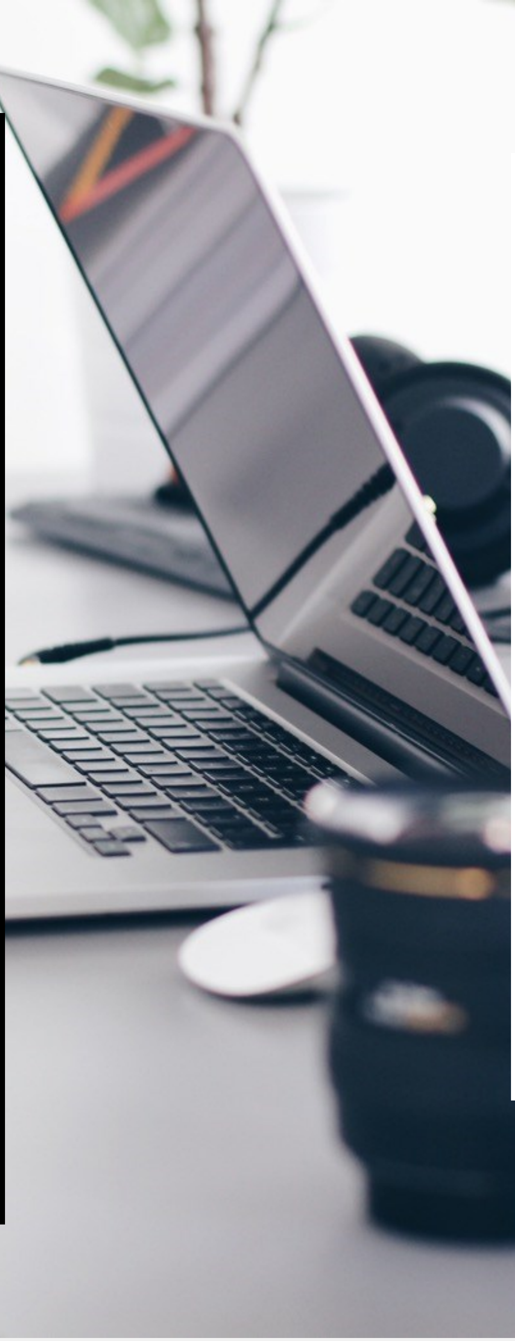
What is Software Engineering?

Software engineering is an **engineering discipline** that is concerned with **all aspects of software production** from the early stages of **system specification** through to **maintaining the system** after it has gone into use

- Using appropriate theories, **process**, **methods**, and **tools** for professional and cost-effective software development, analysis, design, construction, and testing, with organizational and financial constraints
- Not just the technical process of development, but also **project management** and the **development of tools, methods etc. to support software production**

Software Engineering **ADDRESSES** Software Problems

- Software is too expensive
- Software takes too long to build
- Software quality is low
- Software is too complex to support and maintain
- Software does not age gracefully
- Not enough highly-qualified people to design and build software
  - $81B on canceled software projects
  - $59B for budget overruns
  - Only 1/6 projects were completed on time and within budget
  - Nearly 1/3 projects were canceled



Process, Methods, Tools

Various **tasks** are required to build and maintain software

PROCESS

Organization and management of software development tasks

- How to organize and structure development tasks

METHODS

Ways of **performing** software development tasks

- How to perform development tasks

TOOLS

Assist in performing software development tasks

- UMLs
- IDEs
- Issue tracking

A Bit of History of SW Development

50s

- HW oriented

60s

- Complex
- OS & Compilers
- Code-&-fix
- Some large successes
- Major failures
- Spaghetti code

70s

- Formal approaches
- Structured programming
- Specification, development, verification
- Formal methods had their own problems

80s

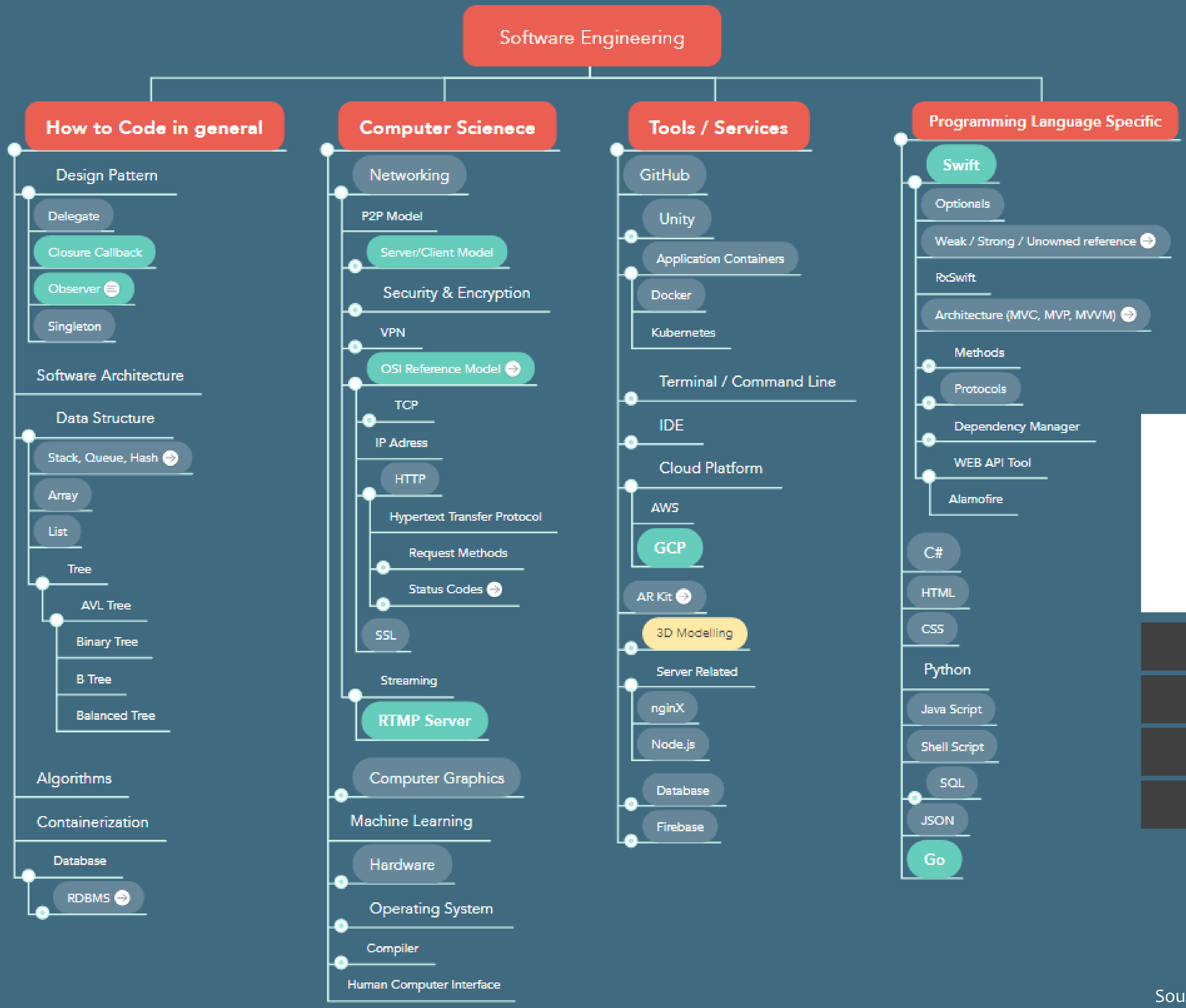
- Productivity, reuse, objects
- Working faster
- Working smarter
- Avoiding work
- Object orientation

90s

- Maturity models and documentation
- Agile methods
- Reverse engineering
- Open source

2000s

- Model-driven
- SOA
- Microservices
- DevOps



Source: MindMeister

SE MAP

Development Principles

Computer Science

Tools and Services

Programming Languages

Course Map

- Fundamentals & Architecture
- Software Requirements Engineering
- Software Design and Design Patterns
- Models & Methods
- Software Implementation
- Software Testing
- Software Quality & Maintenance
- Architecture Revisited
- Software Scalability and Evolution

Course Resources

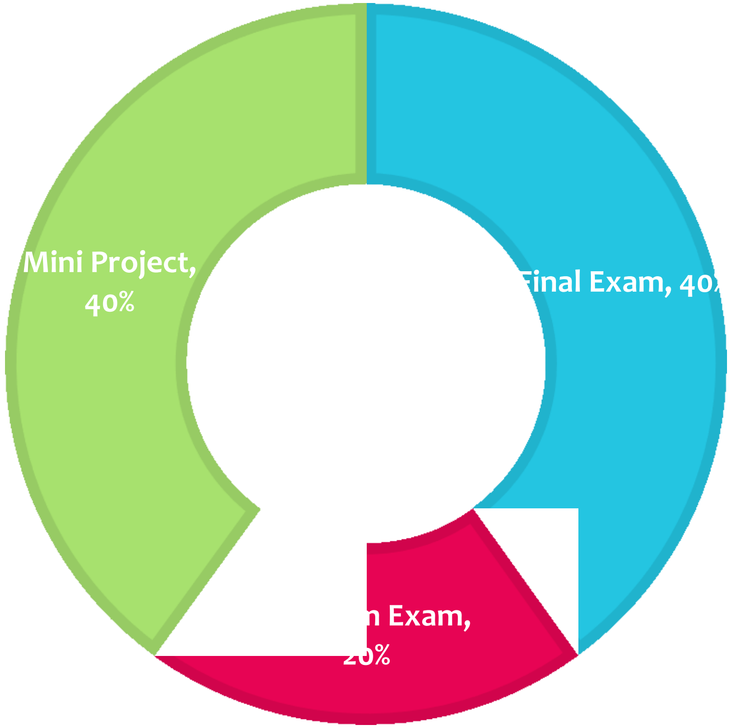
Essential

- *Software Engineering*, 10th Edition by Ian Sommerville

Additional

- *Head First Design Patterns* by Eric Gamma et. al.
- *Clean code: A Handbook of Agile Software Craftsmanship* by Robert C. Martin
- *Code Complete: A Practical Handbook of Software Construction*, 2nd Edition by Steve McConnell
- *SWEBOOK: Guide to the Software Engineering Body of Knowledge, A Project of the IEEE Computer Society*, available online upon registration

Course Grade Distribution



The Mini Project

(Re)Designing The GUC Web System

Software Architecture

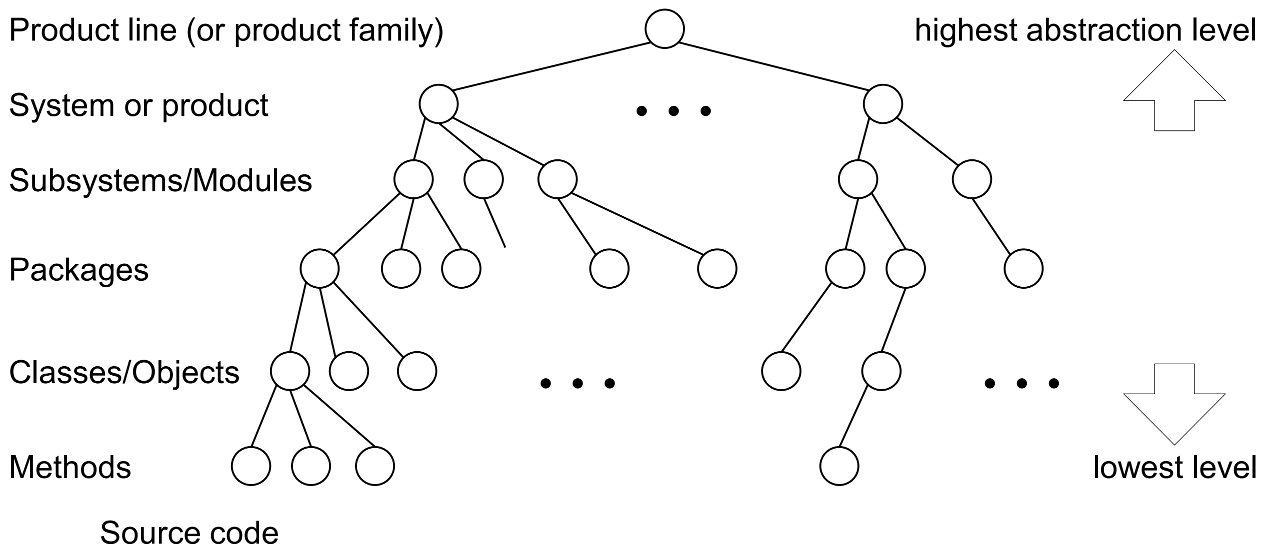
Software is not one long list of program statements – **software has structure**

Why do we start with a basic understanding of software architecture?

To get a bird's-eye view of how a software system is composed before we can dive into details of engineering software

Software's Hierarchical Organization

Taxonomy of structural parts (abstraction hierarchy)
Not a representation of relationships between the parts
Does not specify the function of each part



- Hierarchy shows a taxonomy of the system parts, but not the procedure for decomposing the system into parts — **how do we do it?**
- But first, **why do we want to decompose systems?**

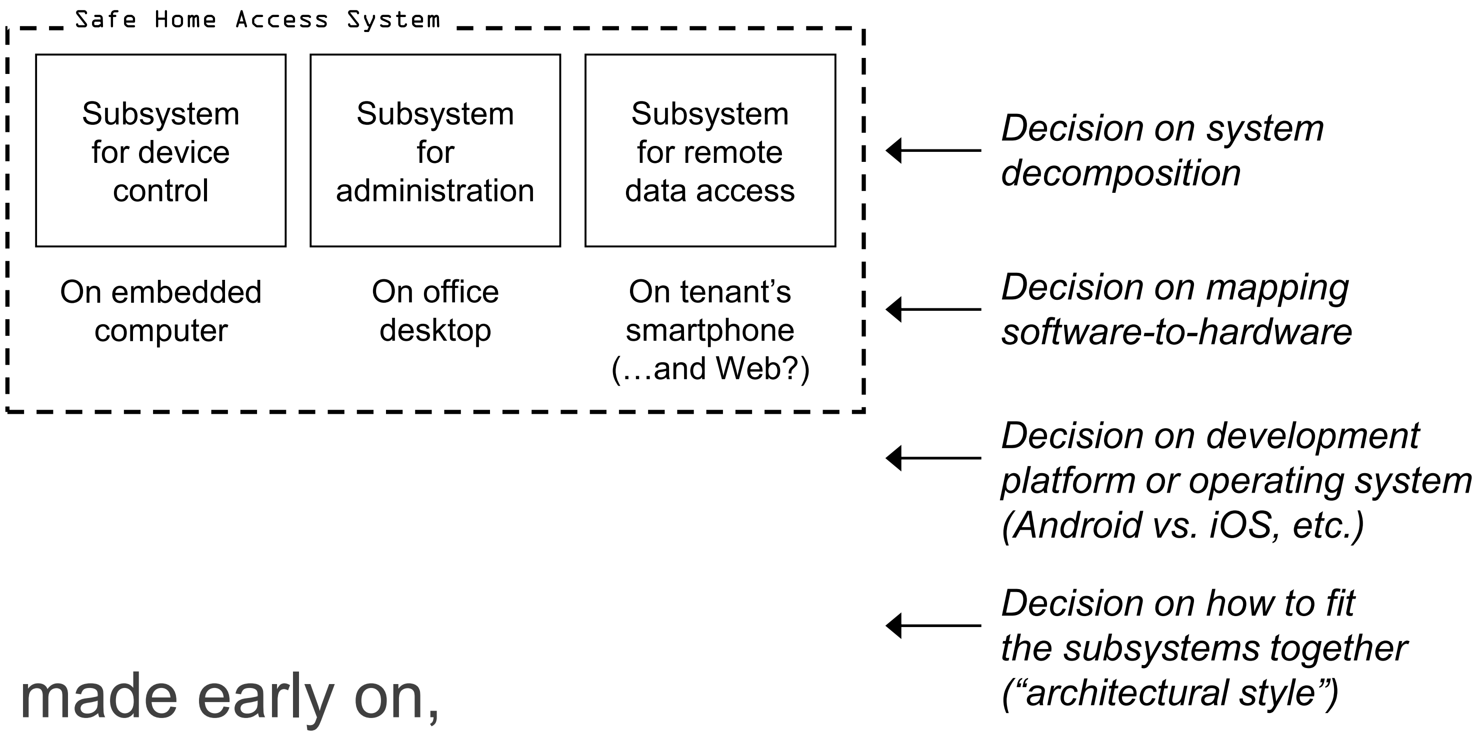
Why We Want To Decompose Systems

- Tackle complexity by “divide-and-conquer”
- See if some parts already exist & can be reused
- Focus on creative parts and avoid “reinventing the wheel”
- Support flexibility and future evolution by decoupling unrelated parts, so each can evolve separately (“**separation of concerns**”)
- Create sustainable strategic advantage

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)
  - Decisions made throughout the development *and* evolution of a software system
  - **made early** and affect large parts of the system ("**design philosophy**") — such decisions are hard to modify later
- Decisions to use well-known solutions that are proven to work for similar problems
- **Software Architecture is not a phase of development**
  - Does **not** refer to a **specific product** of a particular phase of the development process (labeled "high-level design" or "product design")

Example Architectural Decisions



Such decisions are made early on,

perhaps while discussing the requirements with the customer

to decide which hardware devices will be used for user interaction and device control

Software Architecture – Key Concerns

- **System decomposition**
  - **how do we break the system up into pieces?**
    - what functionality/processing or behavior/control to include?
  - **do we have all the necessary pieces?**
  - **do the pieces *fit* together?**
    - how the pieces interact with each other and with the runtime environment
- **Cross-cutting concerns**
  - broad-scoped qualities or properties of the system
  - tradeoffs among the qualities
- **Conceptual integrity**

Architectural Decisions Often Involve Compromise

- The "best" design for a component considered in isolation may not be chosen when components considered together or within a broader context
  - Depends on what criteria are used to decide the "goodness" of a design
  - e.g., car components may be "best" for racing cars or "best" for luxury cars, but will not be best together
- Additional considerations include business priorities, available resources, core competences, target customers, competitors' moves, technology trends, existing investments, backward compatibility, …

Fitting the Parts Together

- Specifying semantics of component **interfaces**
  - Serves as a **contract** between component providers and clients, interfaces must be:
    - fully documented
    - with semantics, not just syntax
    - understandable, unambiguous, precise
- Adding semantics
  - informal description
  - design models (e.g., UML interaction diagrams)
  - pre/post conditions

Documenting Software Architecture: Architecture Views

- Views are different kinds of "blueprints" created for the system-to-be
  - e.g., blueprints for buildings: construction, plumbing, electric wiring , heating, air conditioning, …
    (Different stakeholders have different information needs)

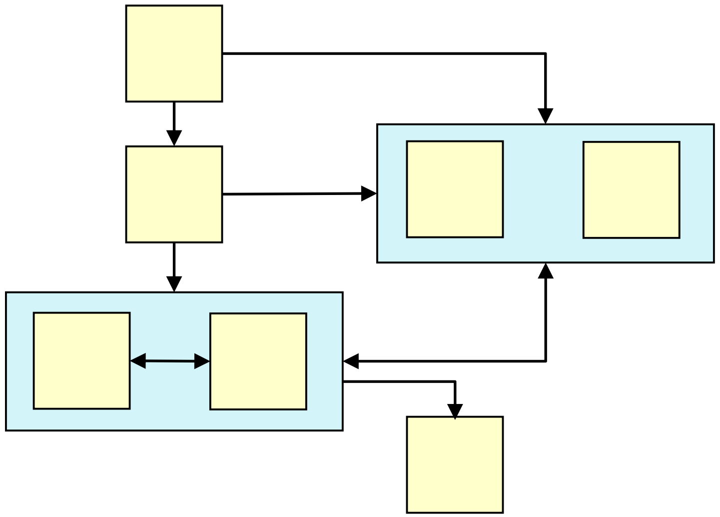
1. **Module/Subsystem Views**
2. **Component and Connector Views**
3. **Allocation Views**

Module/Subsystem Views

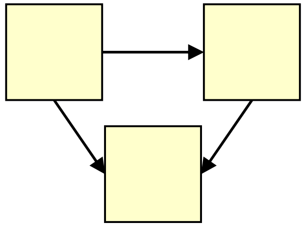
- A **subsystem** is a grouping of elements that form part of a system
- **Coupling** is a measure of the dependencies between two subsystems. If two systems are strongly coupled, it is hard to modify one without modifying the other
- **Cohesion** is a measure of dependencies within a subsystem. If a subsystem contains many closely related functions its cohesion is high

- An ideal division of a complex system into subsystems has **low coupling between subsystems and high cohesion within subsystems**

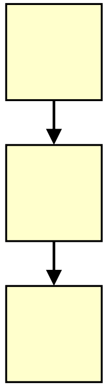
Module/Subsystem Views



- Decomposition View
  - Top-down refinement (e.g., simple “block diagram”)



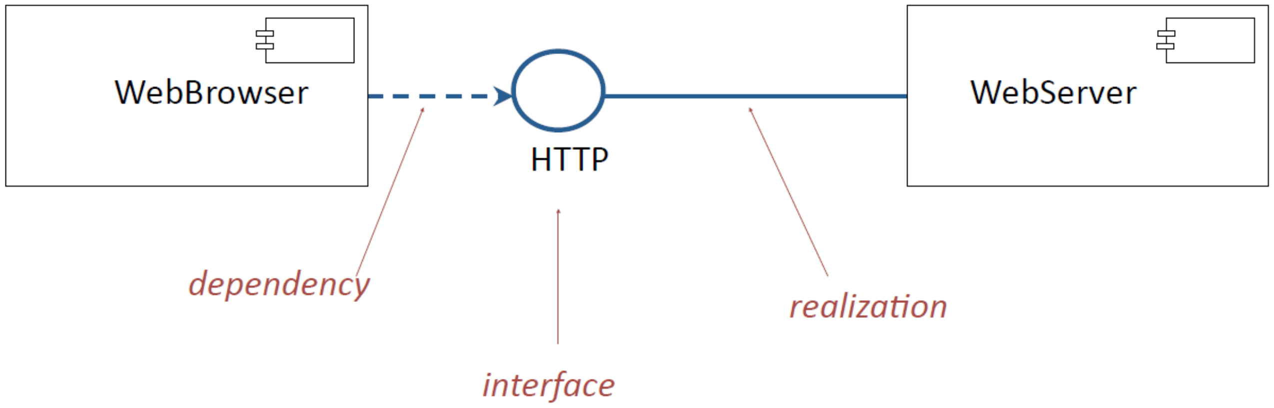
- Dependency View
  - How parts relate to one another



- Layered View
  - Special case of dependency view
- Class View
  - “domain model” in OOA and “class diagram” in OOD

Component and Connector Views

- A **component** is a replaceable part of a system that conforms to and provides the realization of a set of **interfaces**
- Components' operations are accessible only through interfaces
- A component can be thought of as an implementation of a subsystem
- Components are replaceable as long as replacements conform to interfaces

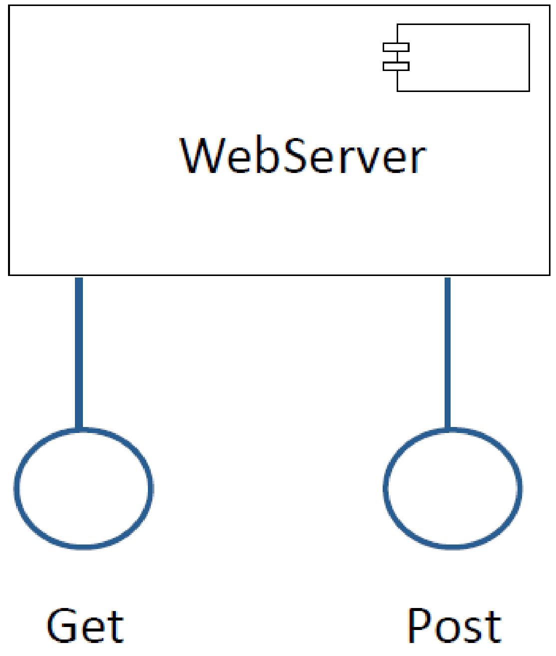


Component and Connector Views

- Process View
  - Defined sequence of activities?
  - System represented as a series of communicating processes
- Concurrency View
- Shared Data View
  - …
- Client/Server View
  - e.g., in Web browsing

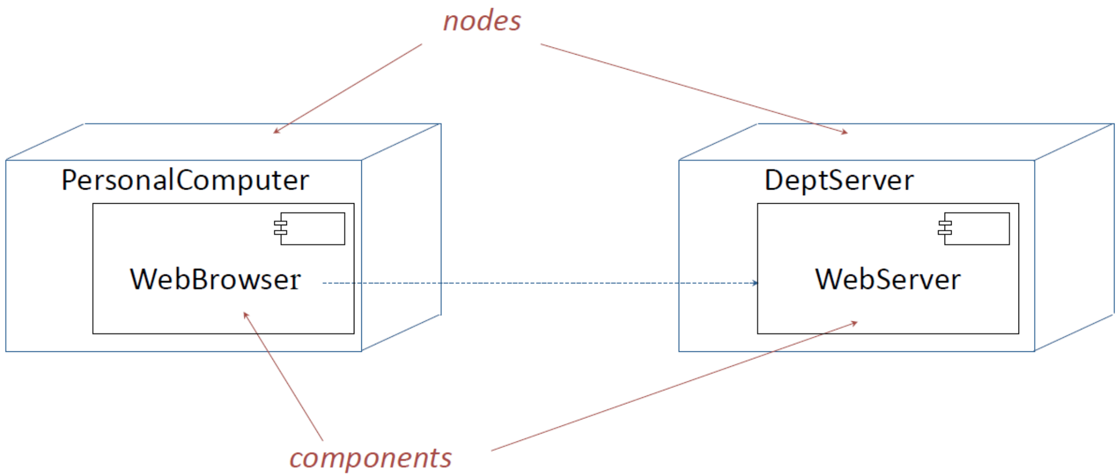
Application Programming Interface (API)

- An **API** is an interface that is realized by one or more components
- A set of **functions and procedures** allowing the creation of applications that access the features or data of an operating system, application, or other service
  - connect internal software components
  - connect components to OS
  - connect components to external services/software components
- Main principle is **information** and **complexity hiding**
- Enables modularity of software system
- Can be released as private, public, or to specific partners



Allocation Views

- Deployment View
  - Software-to-hardware assignment
- Implementation View
  - File/folder structure – “package diagram”
- Work Assignment View
  - Work distribution within the development team

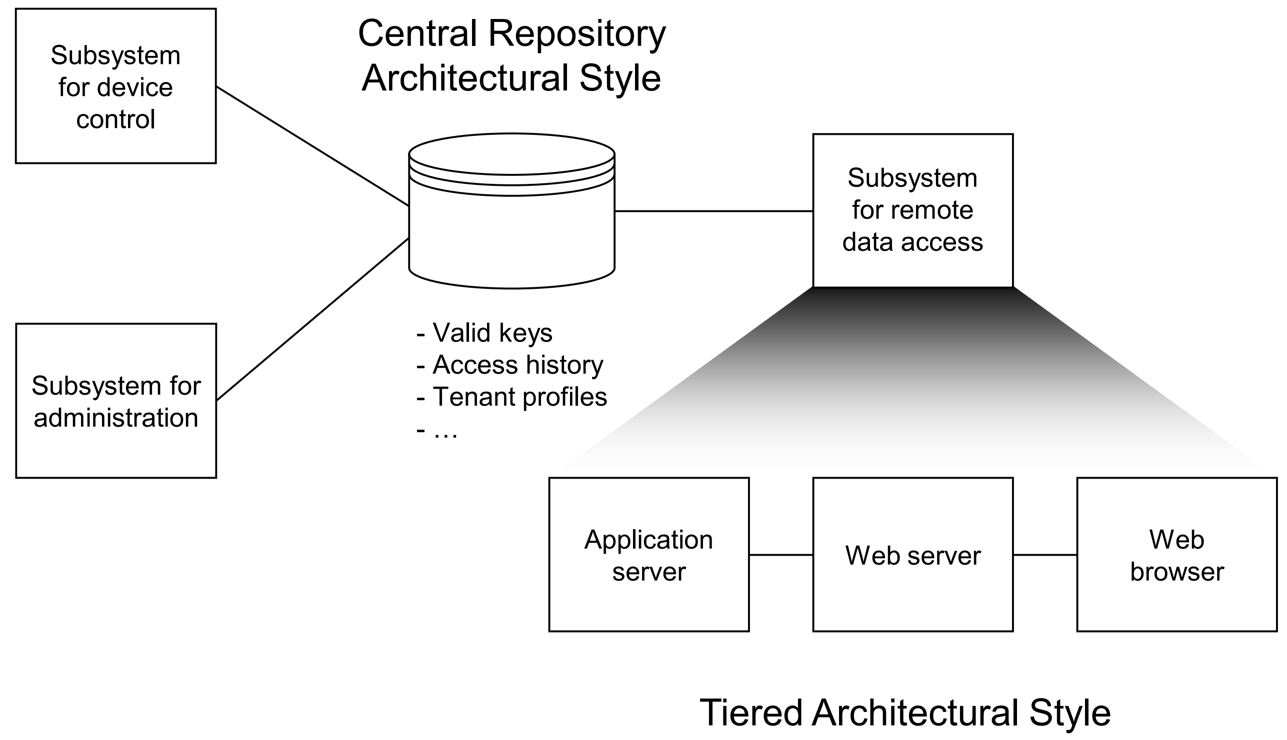


How to Fit Subsystems Together: Some Well-Known Architectural Styles

- Pipe-and-Filter
- Central Repository (database)
- Client/Server
- Layered (or Multi-Tiered)
- Peer-to-Peer
- Model-View-Controller
- World Wide Web architectural style:
  REST (Representational State Transfer)

Development platform (e.g., Web vs. mobile app, etc.) may dictate the architectural style or vice versa…

Real System is a Combination of Styles

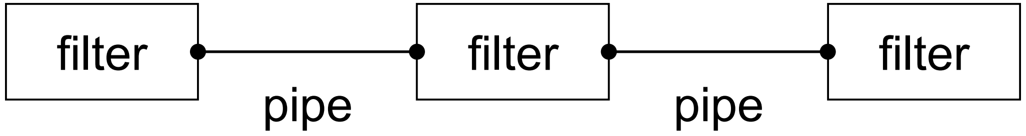


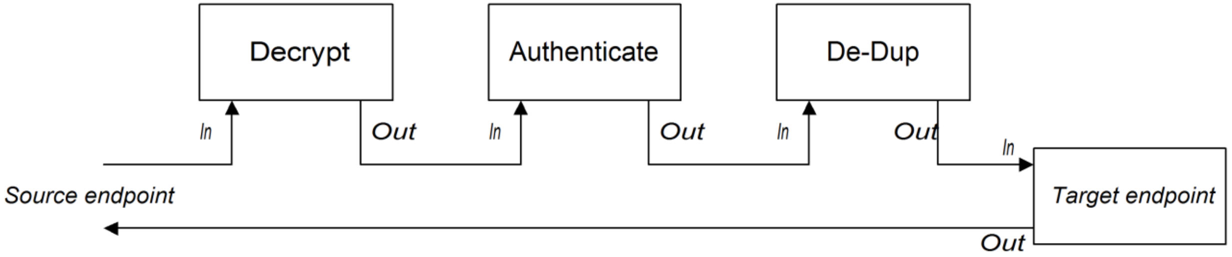
Architectural Styles – Constituent Parts

1. Components
   - Processing elements that “do the work”
2. Connectors
   - Enable communication among components
     - Broadcast Bus, Middleware-enabled, implicit (events), explicit (procedure calls) …
3. Interfaces
   - Connection points on components and connectors
     - define where data may flow in and out of the components/connectors
4. Configurations
   - Arrangements of components and connectors that form an architecture

Architectural Style: Pipe-and-Filter (General)

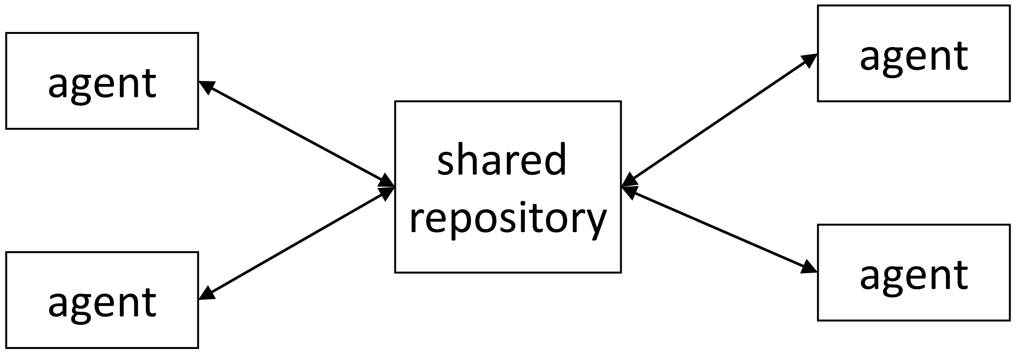
- Components: **Filters** transform input into output
- Connectors: **Pipe** data streams
- Example: Text processing pipeline in machine learning system

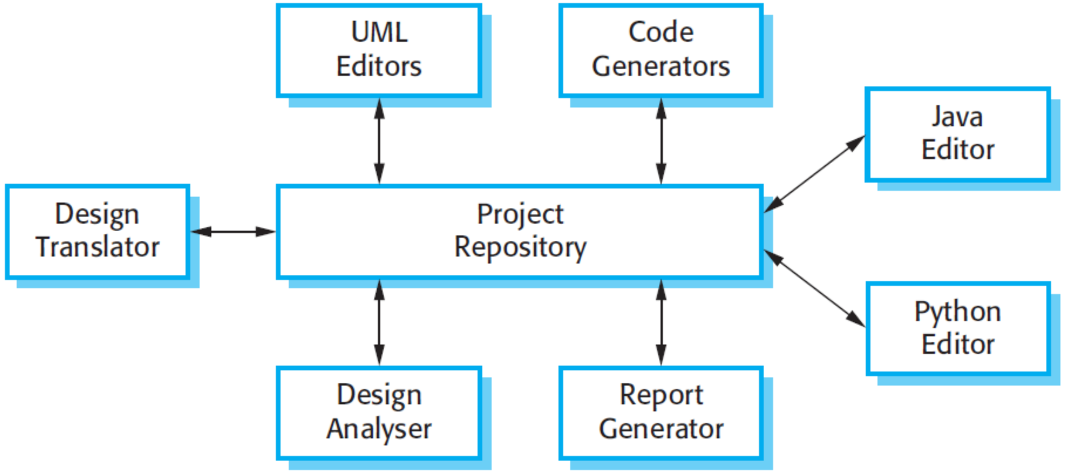




Architectural Style: Repository (General)

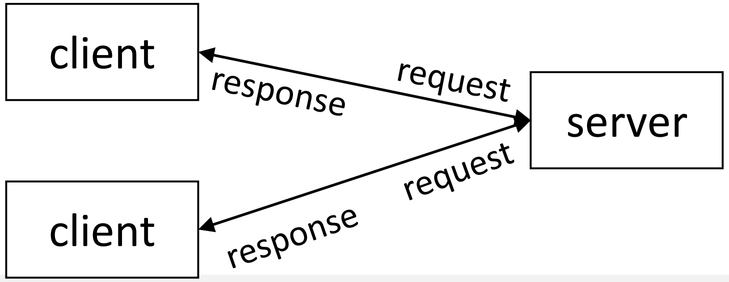
- A **repository** is used for data sharing, where data is generated by one component and used by multiple other components
- Repository is "passive" – agents perform control operations, but can be event-driven: when data becomes available, notify agents
- Also called "blackboard" in AI contexts, where data is unstructured





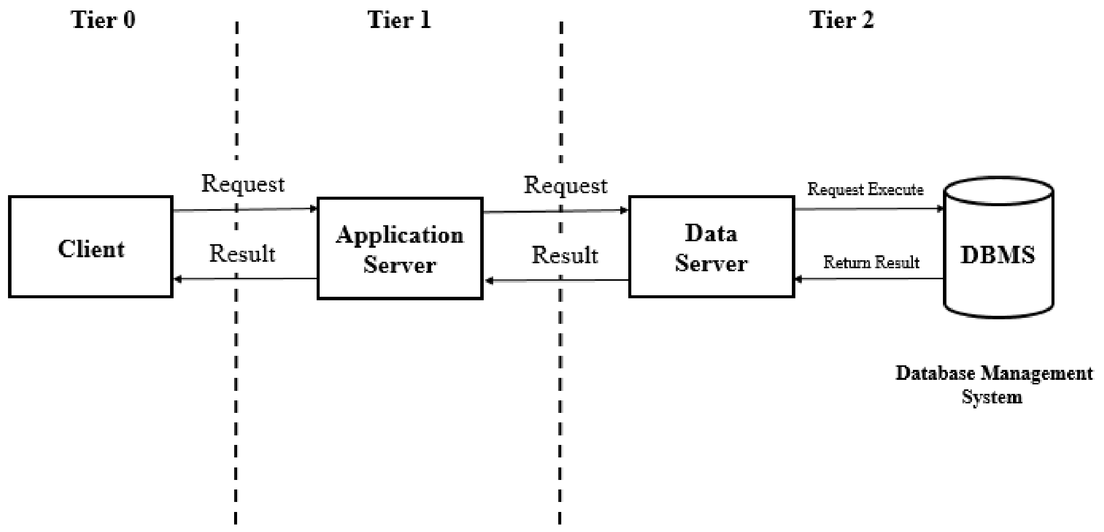
Architectural Style: Client/Server (Distributed)

- A **client** is a triggering process; a **server** is a reactive process. Clients make requests that trigger reactions from servers
- A **server** component, offering a set of services, listens for requests upon those services. A server waits for requests to be made and then reacts to them
- A **client** component, desiring that a service be performed, sends a request at times of its choosing to the server via a connector
- Server either rejects or performs the request and sends response back to client
- In a **peer-to-peer architecture**, the same component acts as both a client and a server



Architectural Style: Tiered (Distributed)

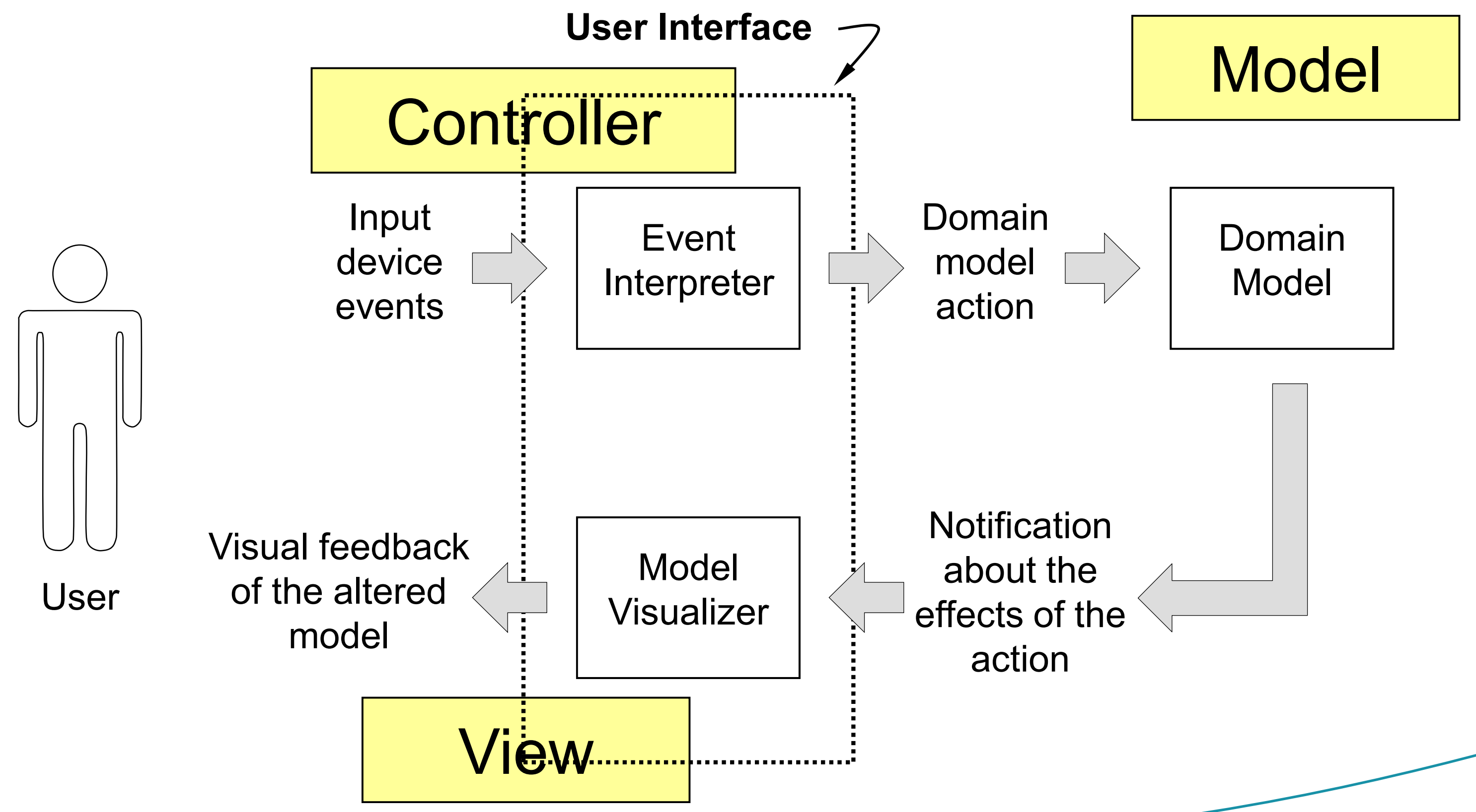
- A layered system is organized hierarchically, each layer providing services to the layer above it and using services of the layer below it
- Layered systems reduce coupling across multiple layers by hiding – and thus abstracting – the inner layers from all except the adjacent outer layer, thus improving evolvability and reusability

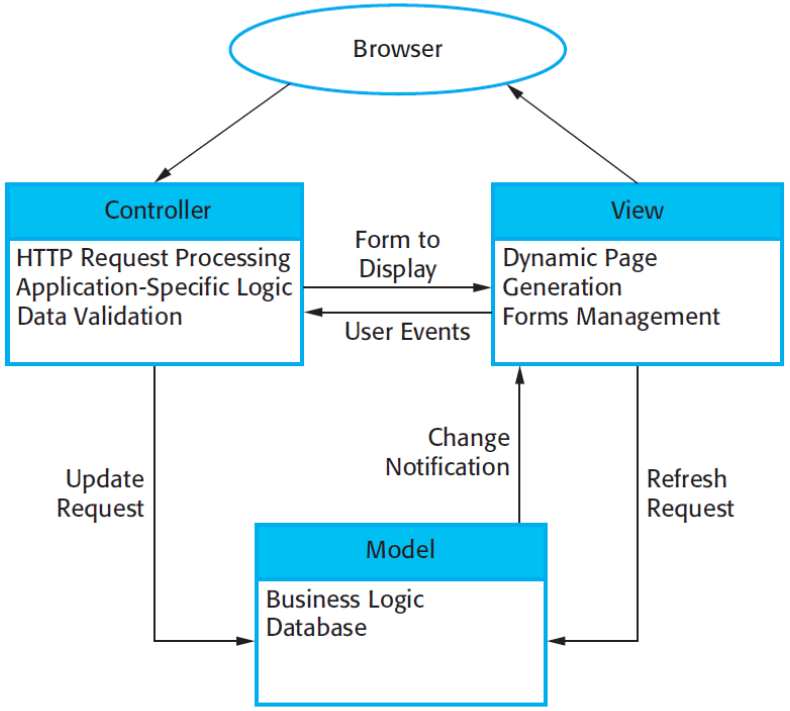


Architectural Style: Model-View-Controller (Interactive)

- **Model**: holds all the data, state and application logic. Oblivious to the View and Controller. Provides API to retrieve state and send notifications of state changes to “observer”
- **View**: gives user a presentation of the Model
  Gets data directly from the Model
- **Controller**: Takes user input and figures out what it means to the Model

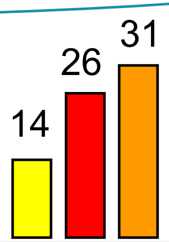
Architectural Style: Model-View-Controller



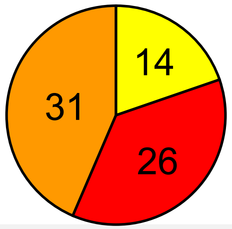


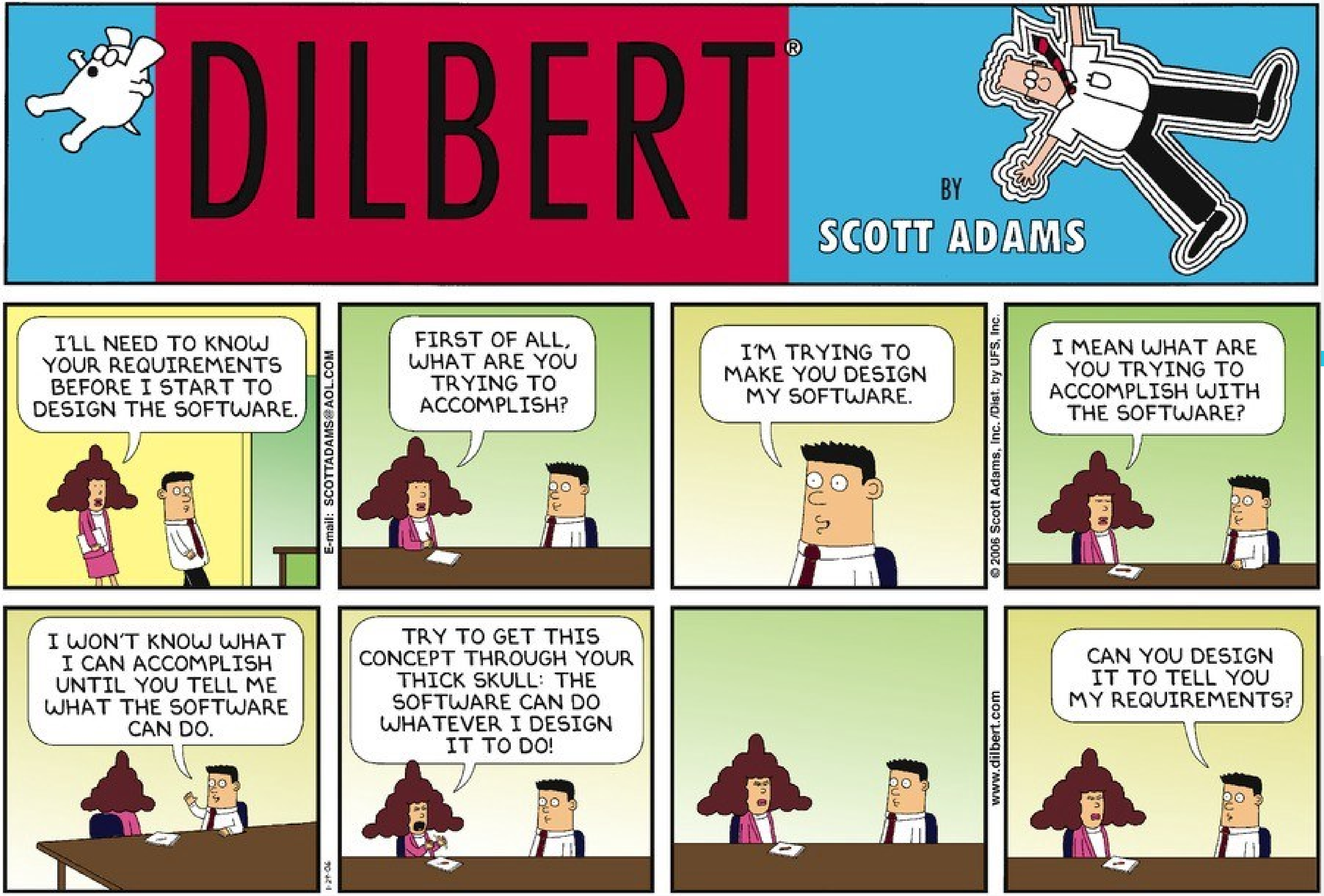
Model: array of numbers [14, 26, 31]

➔ Different Views for the same Model:



versus





Our Weekly Dilbert

Disclaimer

Content is adapted from Ian Sommerville's book slides, Ivan Marsic's lecture slides at Rutgers University, and William Y. Arms' lecture slides from Cornell University

Thank You

mervat.abuelkheir@guc.edu.eg