Software Engineering

Software Architecture

何明昕 HE Mingxin, Max

Send your email to c.max@yeah.net with a subject like: *SE345-Andy: On What …*

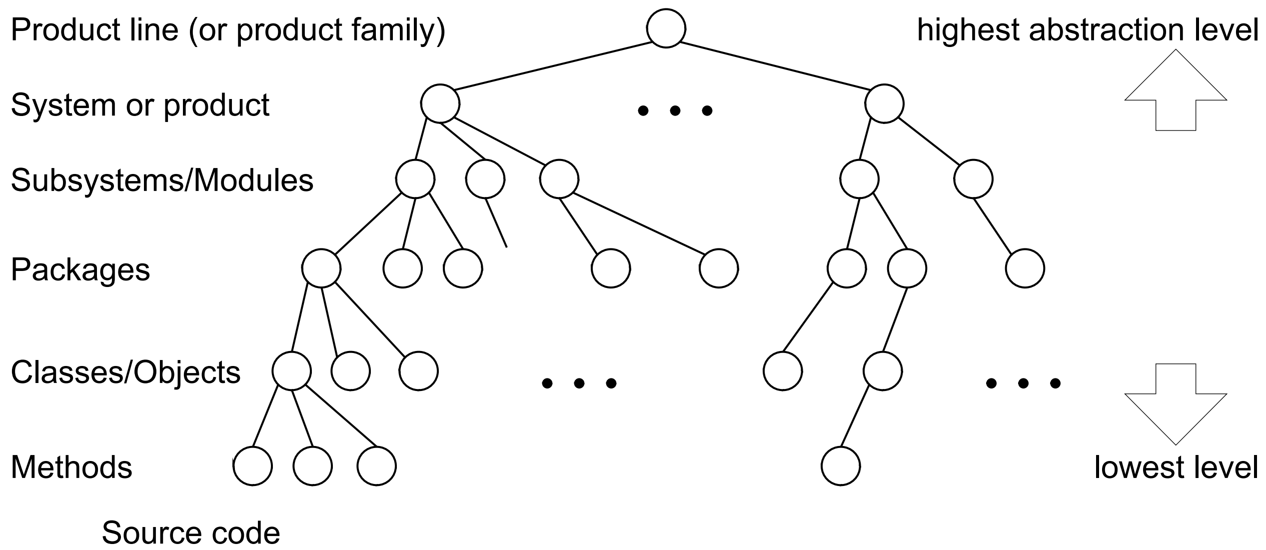
Download from c.program@yeah.net

/文件中心/网盘/SoftwareEngineering24S

Topics

- Software Architecture Definition
- Architectural Decisions & Key Concerns
- Architecture Styles
- Documenting Architecture: Views
- Problem Structure vs. Solution Structure

Hierarchical Organization of Software



- Software is not one long list of program statements but it has **structure**
- The above is a **taxonomy** of structural parts (abstraction hierarchy), but not representation of relationships between the parts and does not specify the function of each part

The 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?

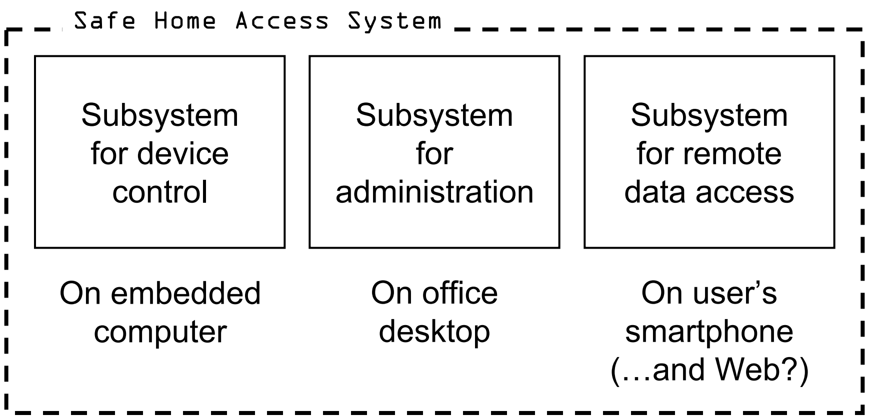
Why We Want To Decompose Systems

- 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)
  - Principal 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



Example decisions:

Decision on system decomposition

Decision on mapping software-to-hardware

Decision on development platform or operating system (Android vs. iOS, etc.)

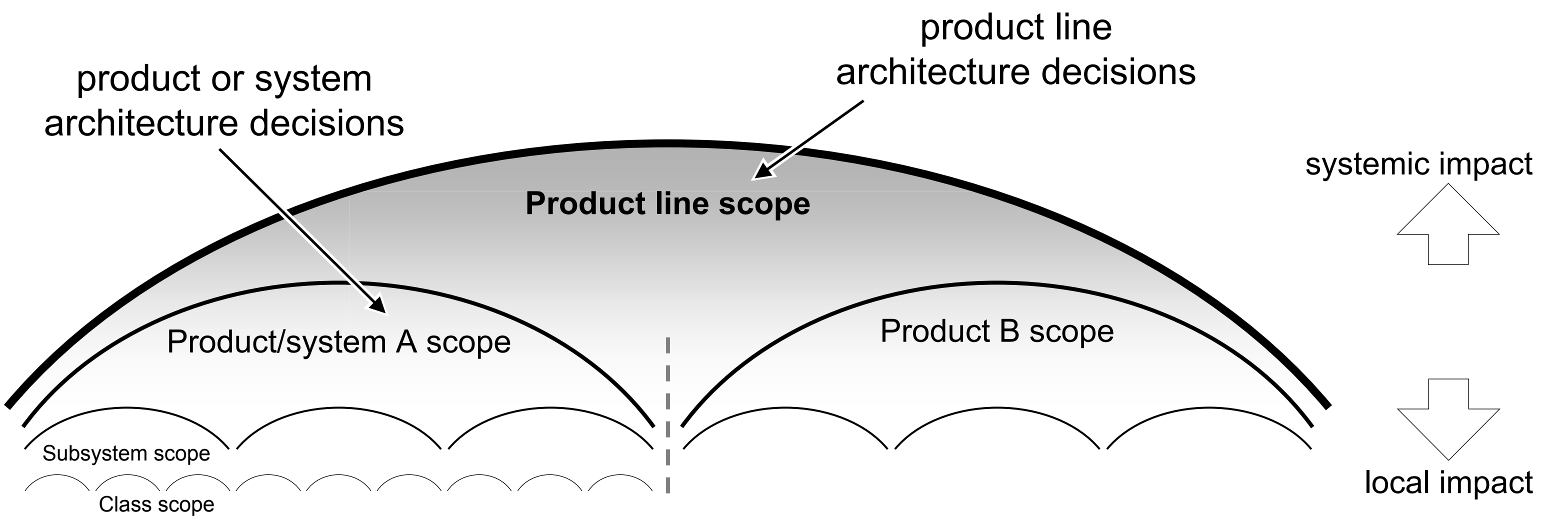
Decision on how to fit the subsystems together ("architecture style")

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

Architectural Decisions
—A matter of scope



- Given the current level of system scope, a decision is "**architectural**" if it can be made only by considering the present scope
  - I.e. could not be made from a more narrowly-scoped, local perspective
  - Architectural decisions should focus on high impact, high priority areas that are in strong alignment with the business strategy

Software Architecture Key Concerns

(Principal decisions to be made)

- 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
- Software architecture provides a set of <u>well-known solutions</u> that are proven to work for similar problems

Architectual 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, ...

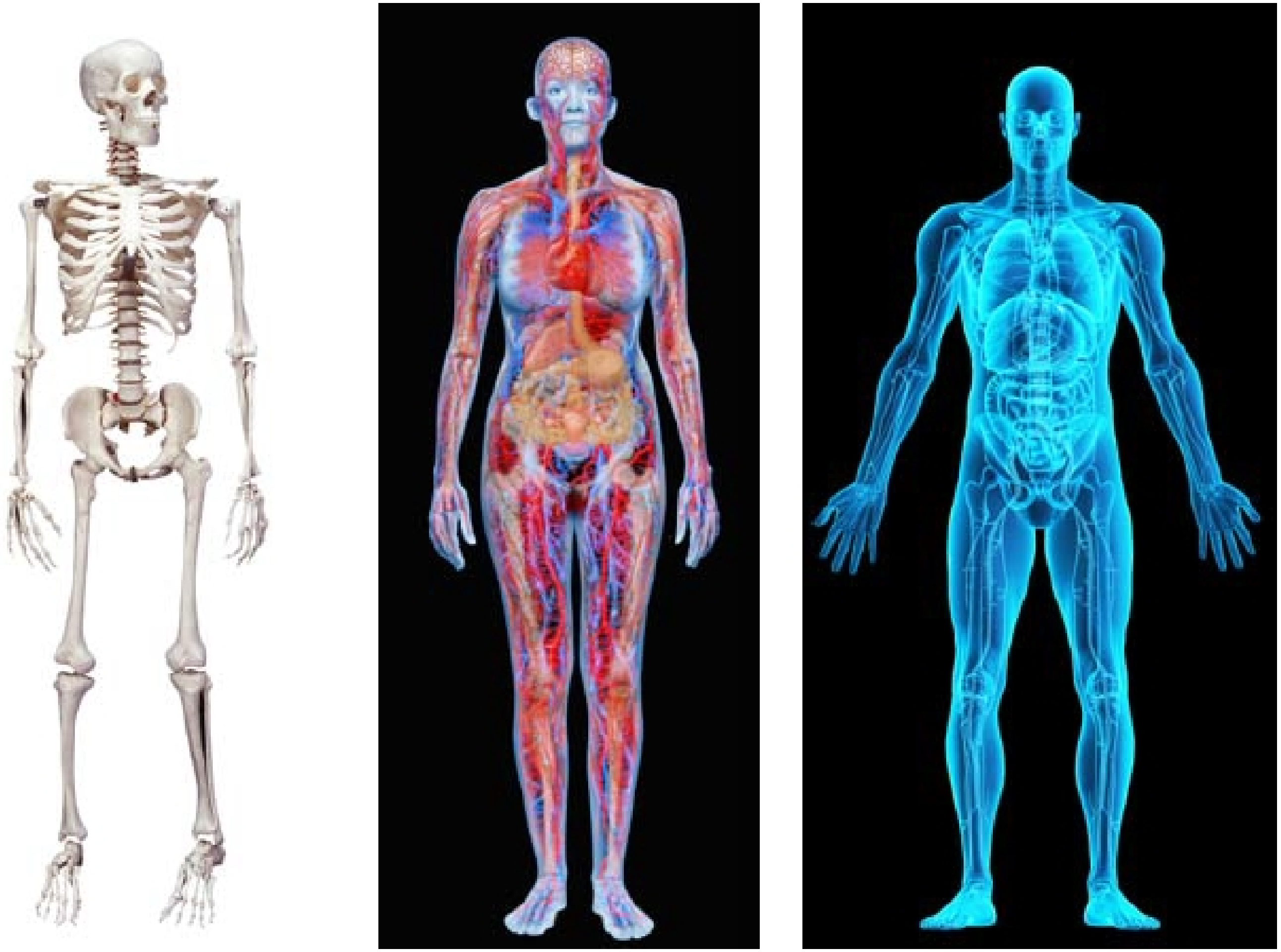


FIGURE 1.1 Physiological structures (Getty images: Brand X Pictures [skeleton], Don Farrall [woman], Mads Abildgaard [man])

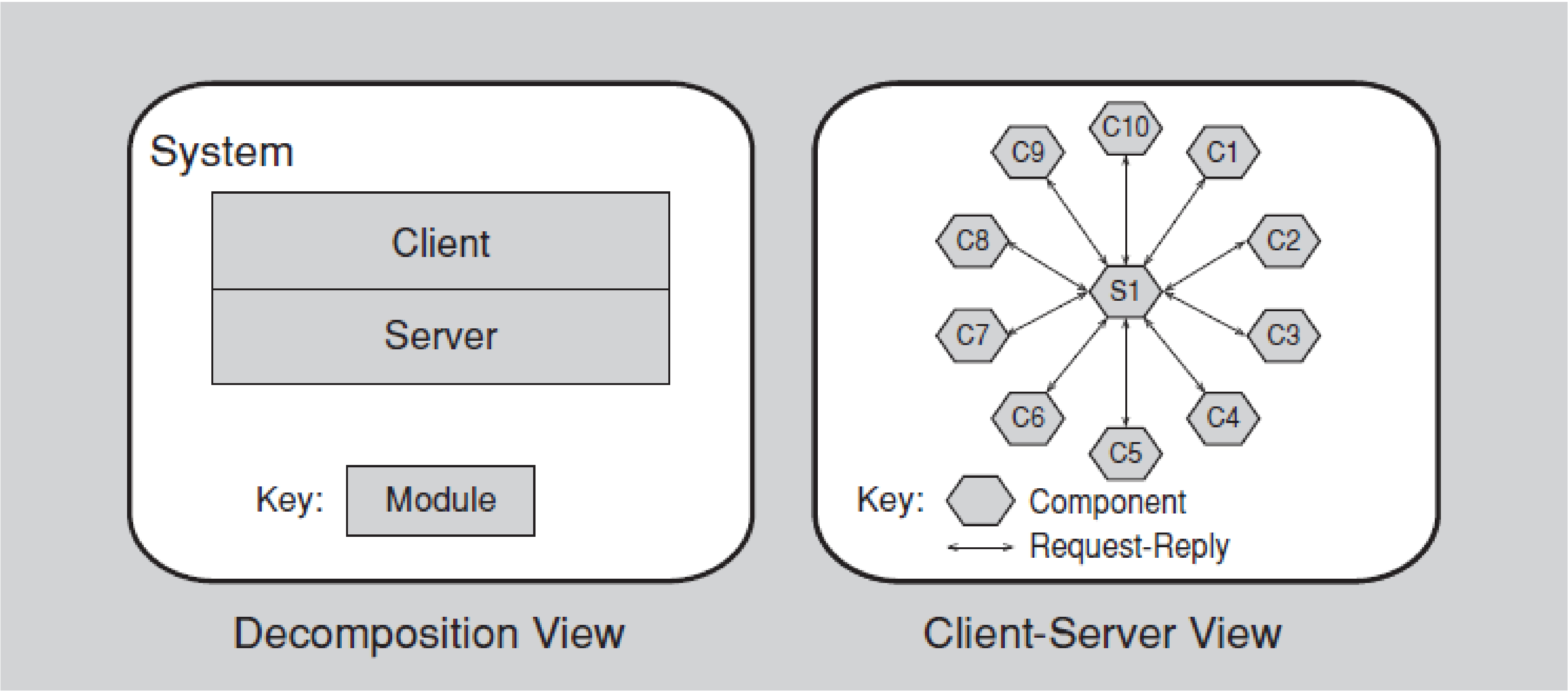
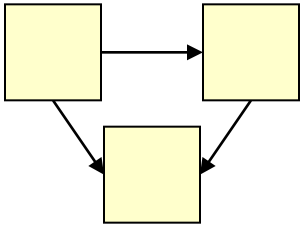
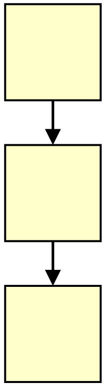


FIGURE 1.2 Two views of a client-server system

Module/Subsystem Views

- Module 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

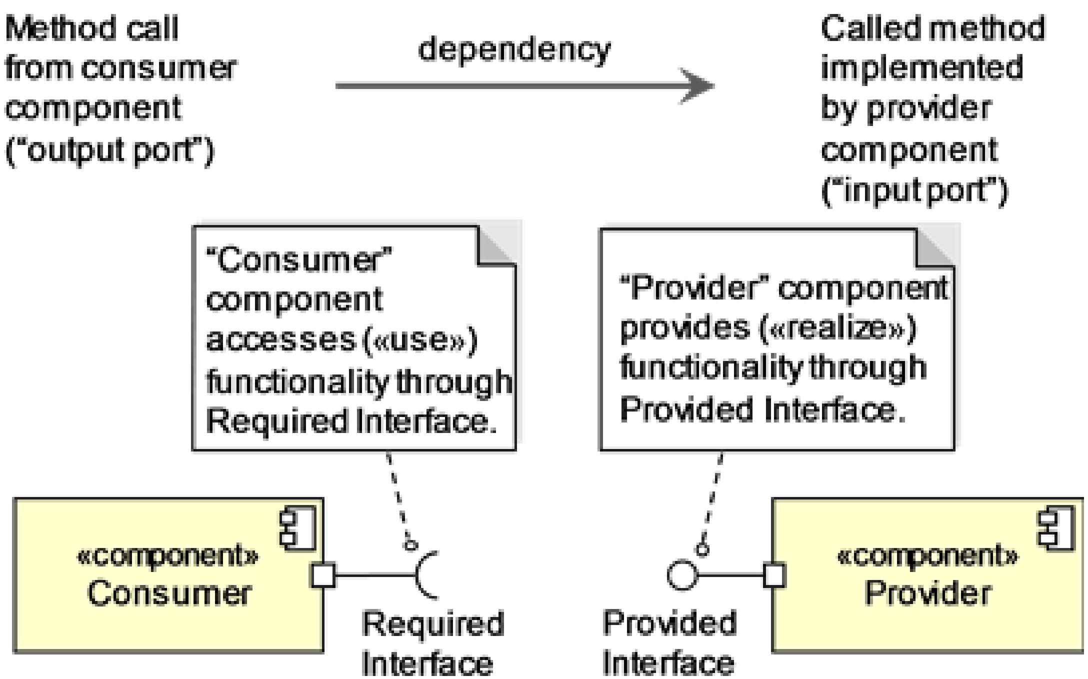
- Process View
  - Defined sequence of activities?
    System represented as a series of communicating processes
- Concurrency View
- Shared Data View
  - …
- Client/Server View
  - E.g., in Web browsing

Allocation Views

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

UML Notation for Software Components

- A component has its behavior defined in terms of **provided interfaces** and **required interfaces** (potentially exposed via ports)

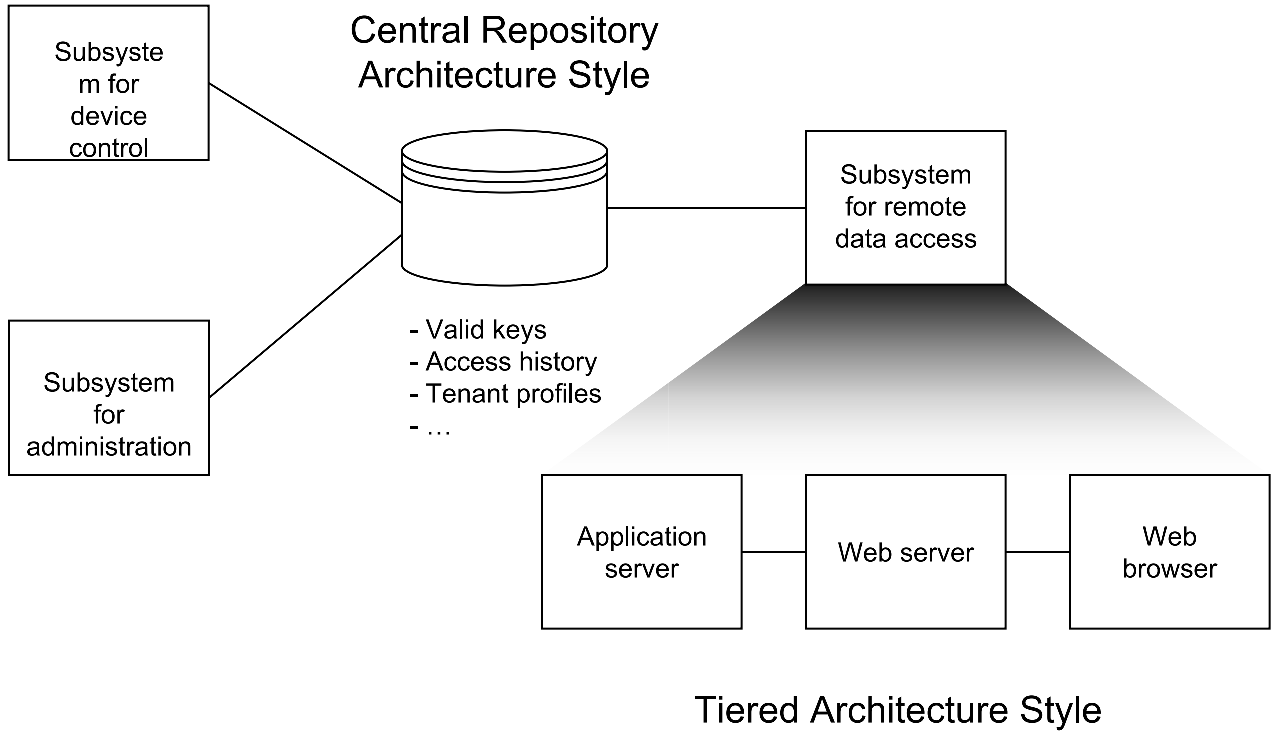


How to Fit Subsystems Together: Some Well-Known Architecture Styles

- World Wide Web architecture style:
  REST (Representational State Transfer)
- UNIX shell script architecture style:
  Pipe-and-Filter
- Client/Server
- Central Repository (database)
- Layered (or Multi-Tiered)
- Peer-to-Peer
- Microservices

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

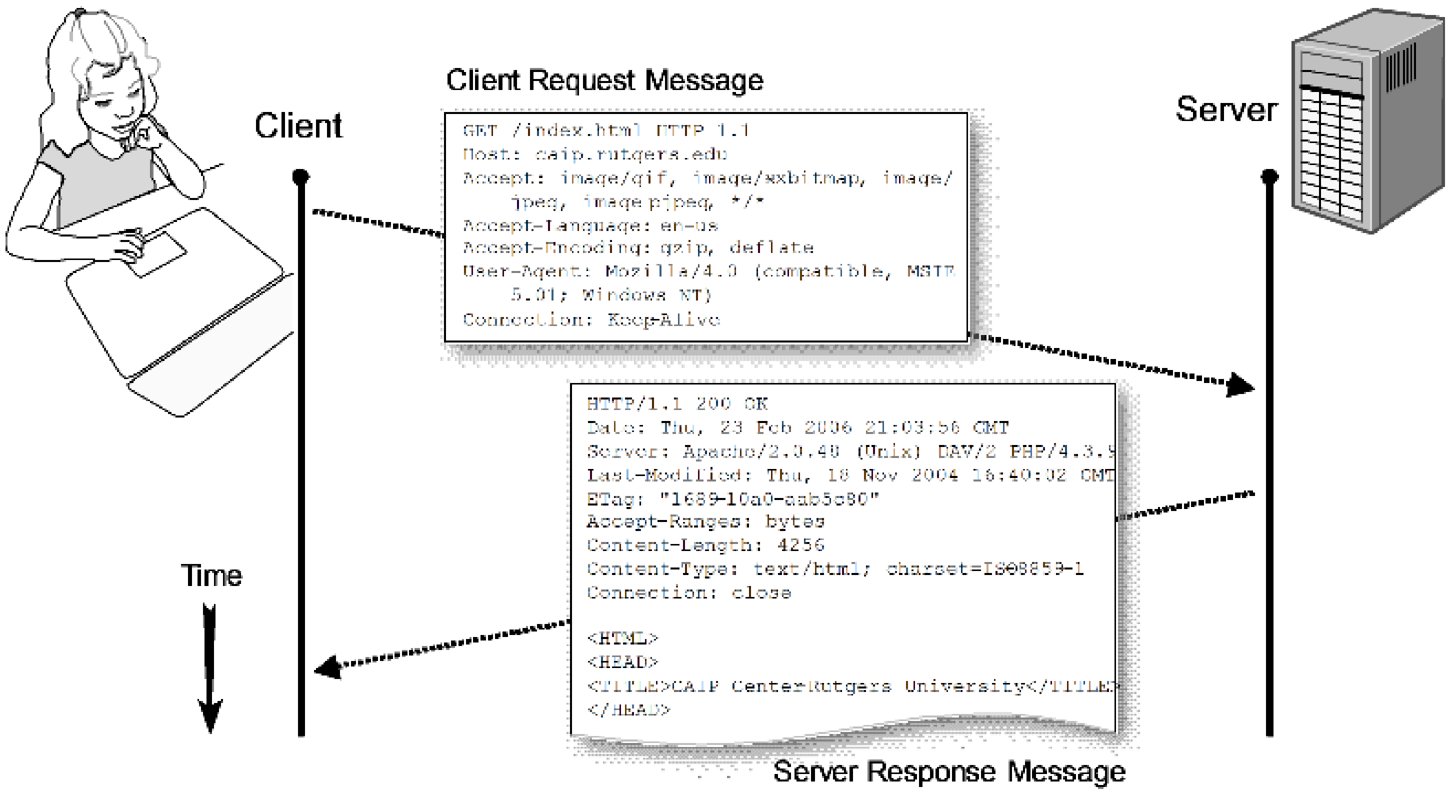
Real System is a Combination of Styles



Architecture 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, ORBs, explicit communications bus) …
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

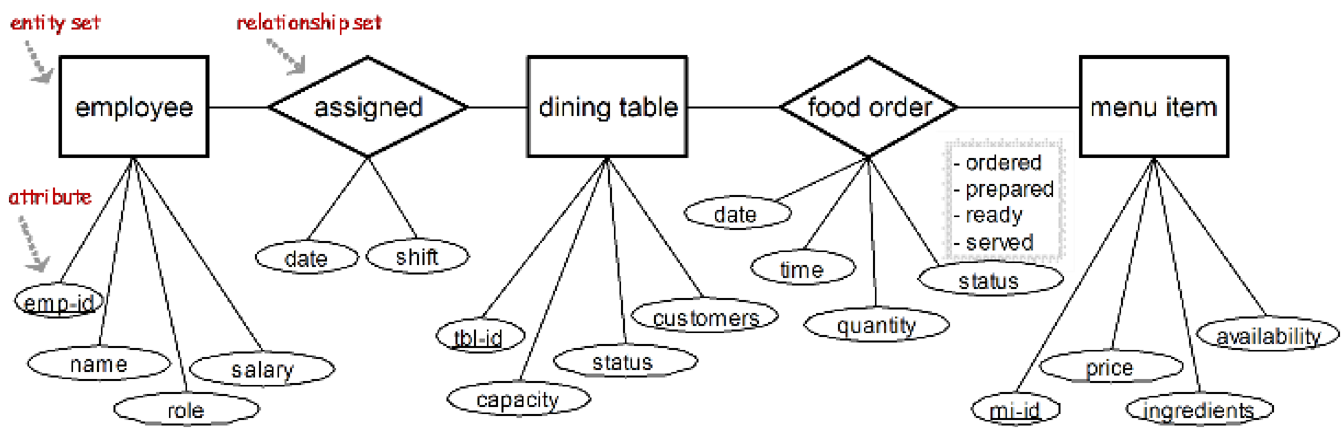
Connectors: HTTP Protocol



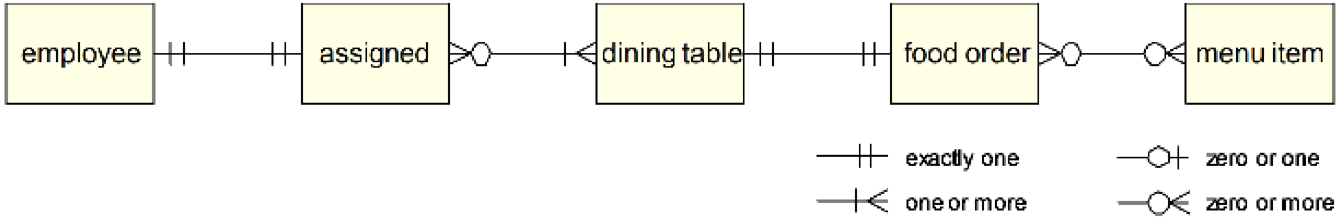
- POST method requests the server to **create** a new resource
- GET method requests the server to **read** a resource
- PUT method requests the server to **update** a resource
- DELETE method requests the server to **delete** a resource

Shared Database Connector
— Data Model for Restaurant

Entity-relationship diagram of the data model:



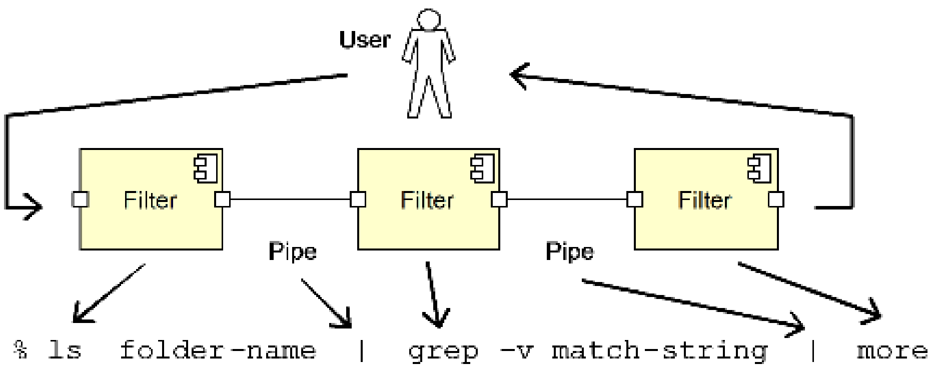
Crow's foot diagram of the same data model:



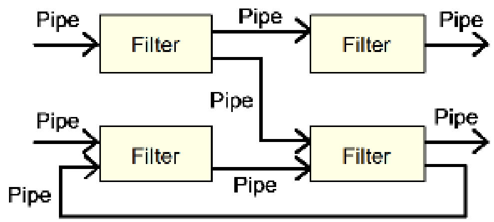
employee (identifier, name, role, salary, …)
assigned (date, working shift)
dining table (identifier, capacity, status, seated customers)
food order (time, item, quantity, total price, status)
menu item (identifier, price, ingredients, availability)
payment (time, date, amount, confirmation number)

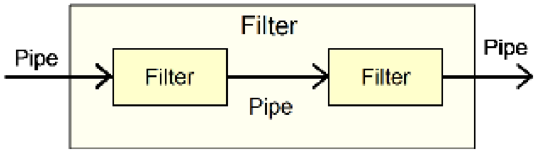
Architecture Style: Pipe-and-Filter

- Components: **Filters** transform input into output
- Connectors: **Pipe** data streams
- Example: UNIX shell commands



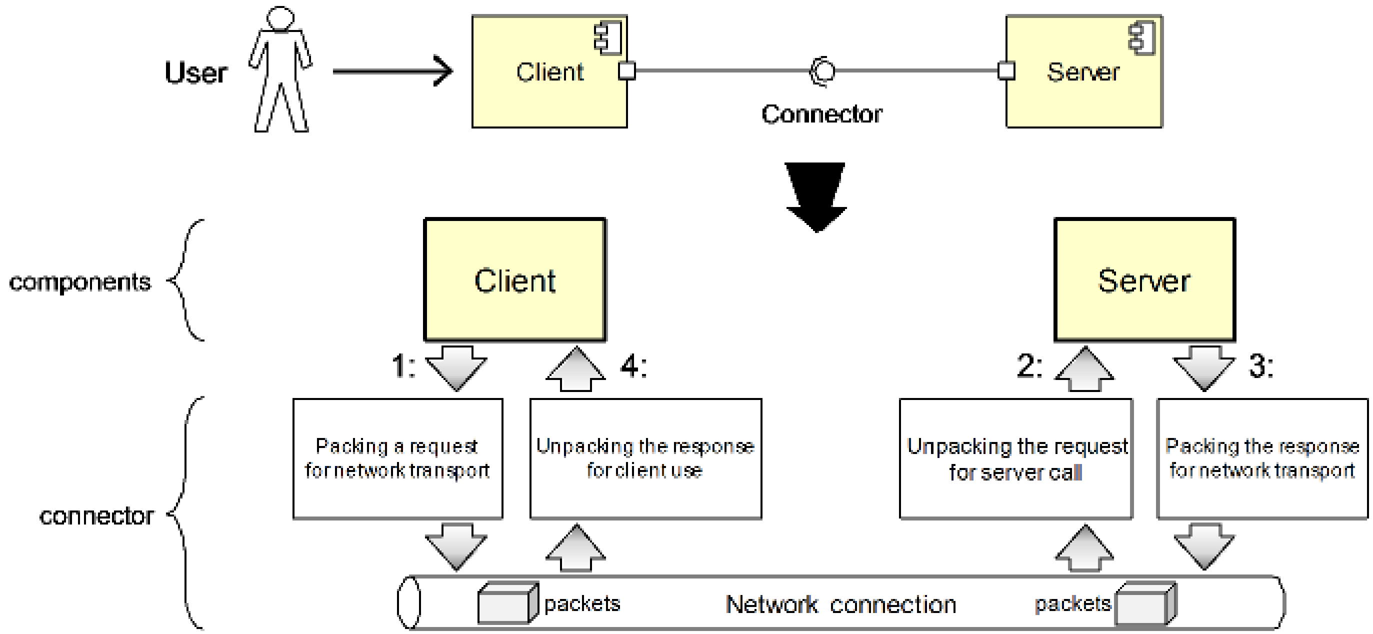
- More complex configurations:





Architecture Style: Client/Server

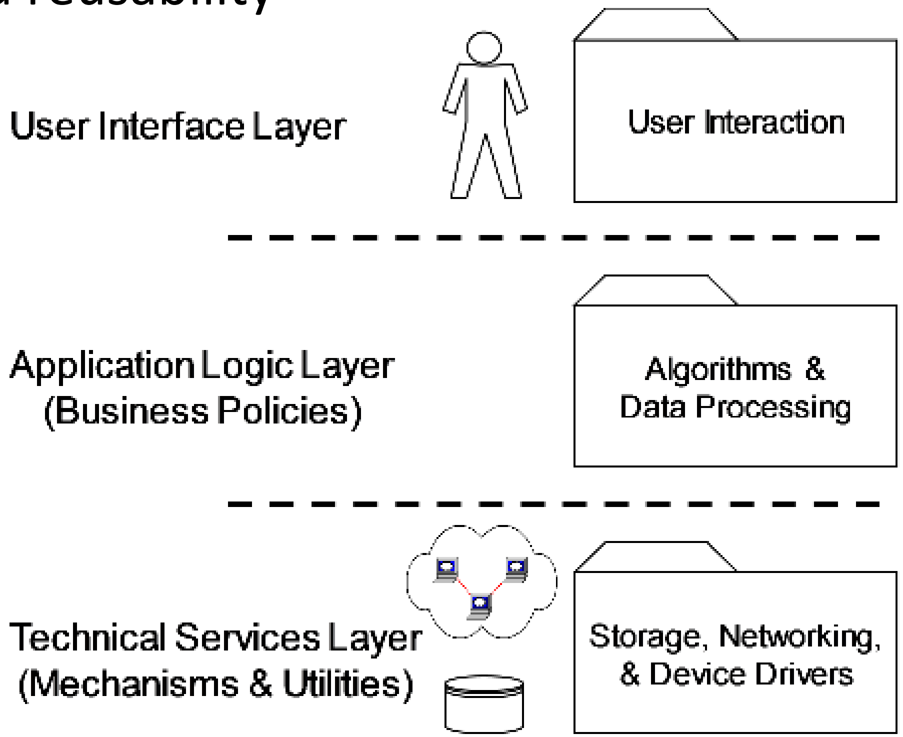
- A **client** is a triggering process; a **server** is a reactive process. Clients make requests that trigger reactions from servers.
- A **server** component, offering a set of services, listens for requests upon those services. A server waits for requests to be made and then reacts to them.
- A **client**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



Architecture Style: Layered

a.k.a. Tiered Software Architecture

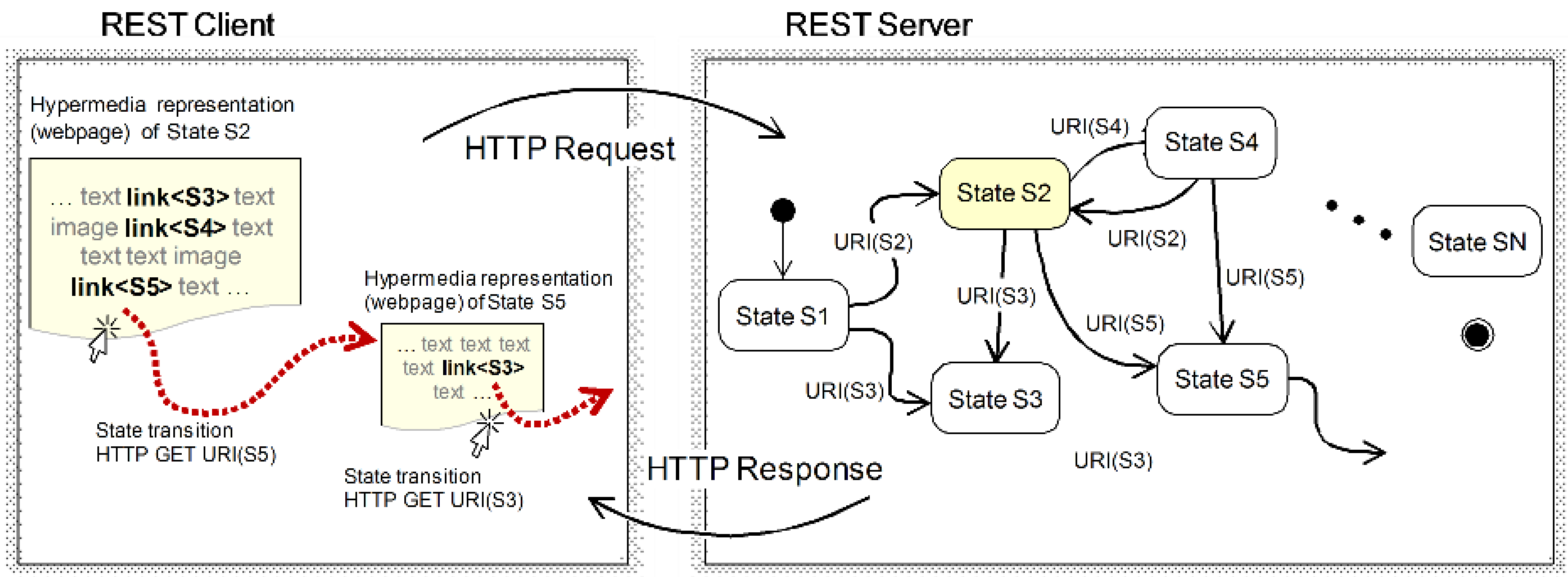
- A layered system is organized hierarchically, each layer providing services to the layer above it and using services of the layer below it
- Layered systems reduce coupling across multiple layers by hiding the inner layers from all except the adjacent outer layer, thus improving evolvability and reusability



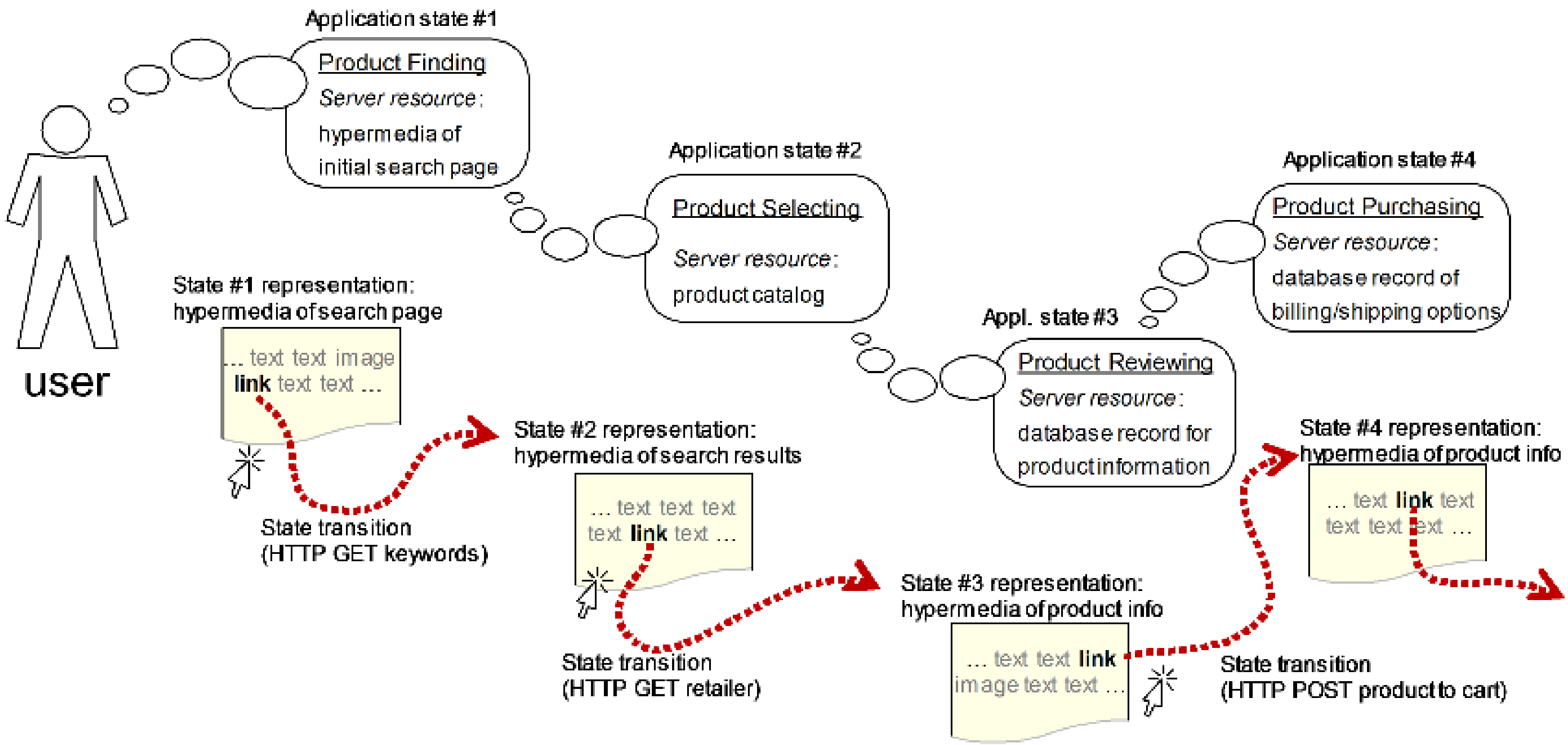
REST: Hypermedia as the Engine of Application State (HATEOAS)

REST: Representational state transfer

Conceptual model: hypermedia as the engine of application state (HATEOAS)

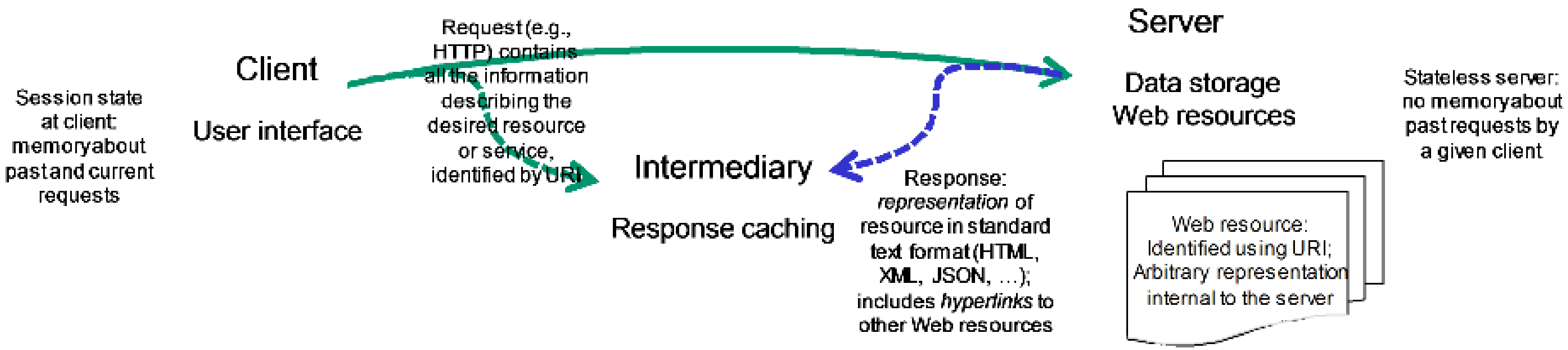


Hypermedia as the Engine of Application State



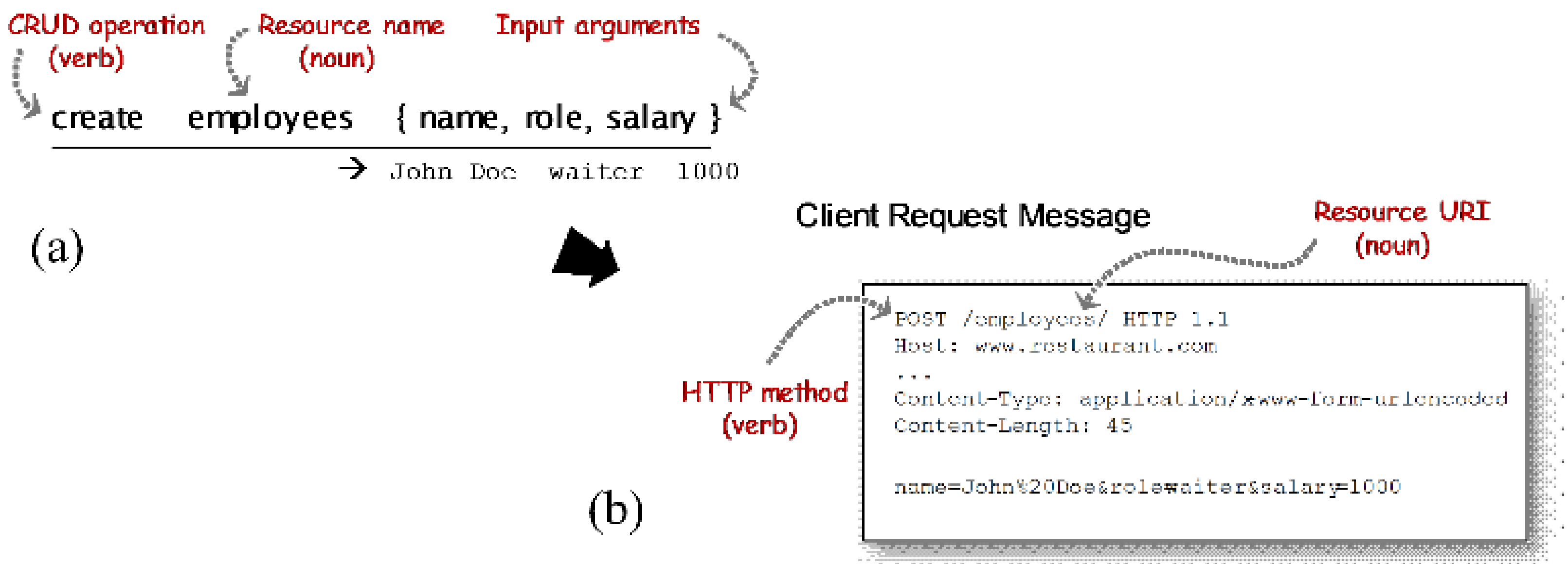
Web application:
A network of Web resources (a virtual state-machine), where the user by following the links requests the transfer of next resource ("state").

REST physical model



RESTful Resource Naming

(a) Conceptual representation of creating a new employee record



(b) HTTP request message carrying the content of the HTML form that was used to enter the employee data

REST, a resource-based architecture style

- Client/server style that supports caching and replicated repositories
- Uniform interface between components based on HTTP methods and URI-named resources, which is efficient for coarse-grain hypertext/hypermedia data transfer
- Server dynamically communicates to clients the application logic in hypertext or in code-on-demand
- Hypertext does not specify the operations allowed on a resource; it just contains embedded hyperlinks that determine all possible transitions from the current to the next application state
- For every media type, the server has defined a default processing model; standard formats for message payload representation (HTML, XML, JSON, JPEG, …) ensure interoperability between different clients and servers
- Stateless sessions—each request must carry all the information needed for its processing
- Statelessness may cause network inefficiency, so requires caching to avoid redundant responses
- A RESTful API looks like hypertext
  - Every addressable unit of information carries an address or identifier, either explicitly (e.g., link and id attributes) or implicitly (e.g., derived from the media type definition and representation structure)
    - Every media type defines a default processing model
    - Hypertext doesn't specify the operations allowed on a resource;
      it specifies which operation to use for each potential state transition

RESTful API Design versus JSON-RPC APIs over HTTP

RESTful APIs	JSON-RPC APIs over HTTP
URIs for resource addresses	URIs for object addresses
Focus on navigable resources (nouns)	Focus on data-processing procedures (verbs)
HATEOAS, application logic runs on the server	Application logic runs on the client; the server manages the data using CRUD and application-specific RPCs
HTTP methods for resource CRUD actions, plus media-specific server-side controllers activated by the media type	HTTP methods for service actions, plus application-specific server-side procedures explicitly addressed with the verb as part of the request message URI
Message body can be any standard media format, as negotiated by clients and servers	Message body encoded as JSON objects
Code-on-demand	No code in response messages—only JSON objects
Caching and replication of resource content	Caching and replication not relevant for procedures

Example Request

- Cannondale Trail 1 12 Kids' Bike:
  - https://www.rei.com/product/145832/cannondale-trail-1-12-kids-bike?CAWELAID=120217890005260197&CAGPSPN=pla&CAAGID=15877514320&CATCI=aud-553371945779:pla-539113402242&cm_mmc=PLA_Google%7C404_1050512963%7C1458320001%7Cnone%7Ccb2b5bf7-95ae-43d8-a53c-6a16aaf4a0d0%7Caud-553371945779:pla-539113402242&lsft=cm_mmc:PLA_Google_LIA%7C404_1050512963%7C1458320001%7Cnone%7Ccb2b5bf7-95ae-43d8-a53c-6a16aaf4a0d0&kclid=cb2b5bf7-95ae-43d8-a53c-6a16aaf4a0d0&gclid=CjwKCAiAyrXiBRAjEiwATI95mcKyoLfTjKFAaCiElWKC3Vt51nTPk0Fxyt9rm8S0Y99Dikidu_DQrxoC2CwQAvD_BwE

Cannondale / Cycling / Bikes / Kids' Bikes

Cannondale Kids' Cruiser Bikes (4 products)

TOP RATED

Cannondale
Quick 24 Kids' Bike - Acid Red

$399.93
Save 20%
~~$500.00~~

(20)

Compare

Cannondale
Quick 20 Kids' Bike

$347.93
Save 20%
~~$435.00~~

(3)

Compare

NEW ARRIVAL

Cannondale
Quick 20 Kids' Bike - Orchid

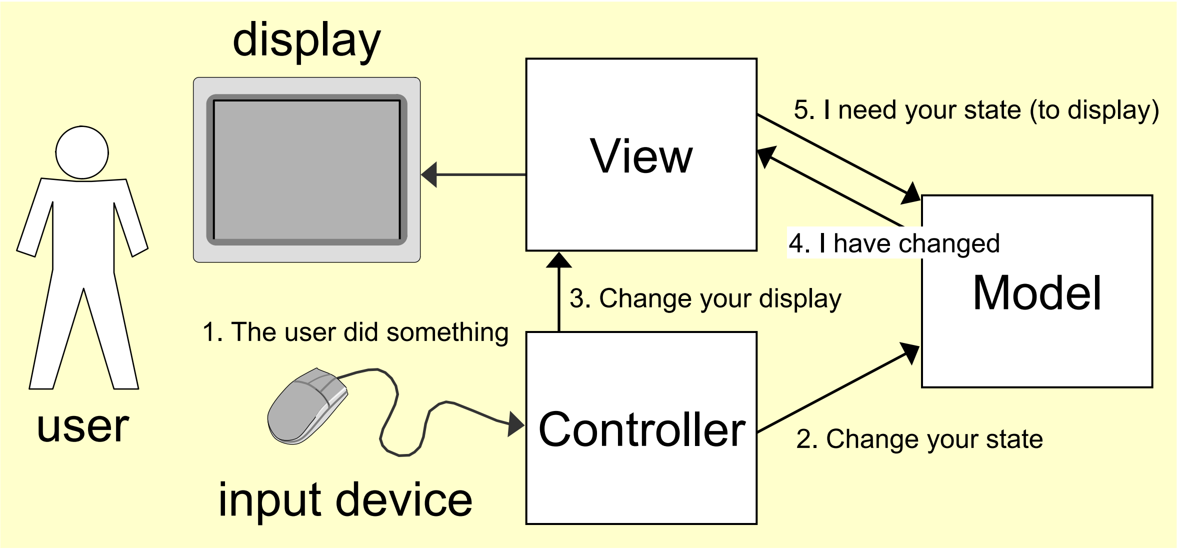
$309.99
Save 29%
~~$440.00~~

(0)

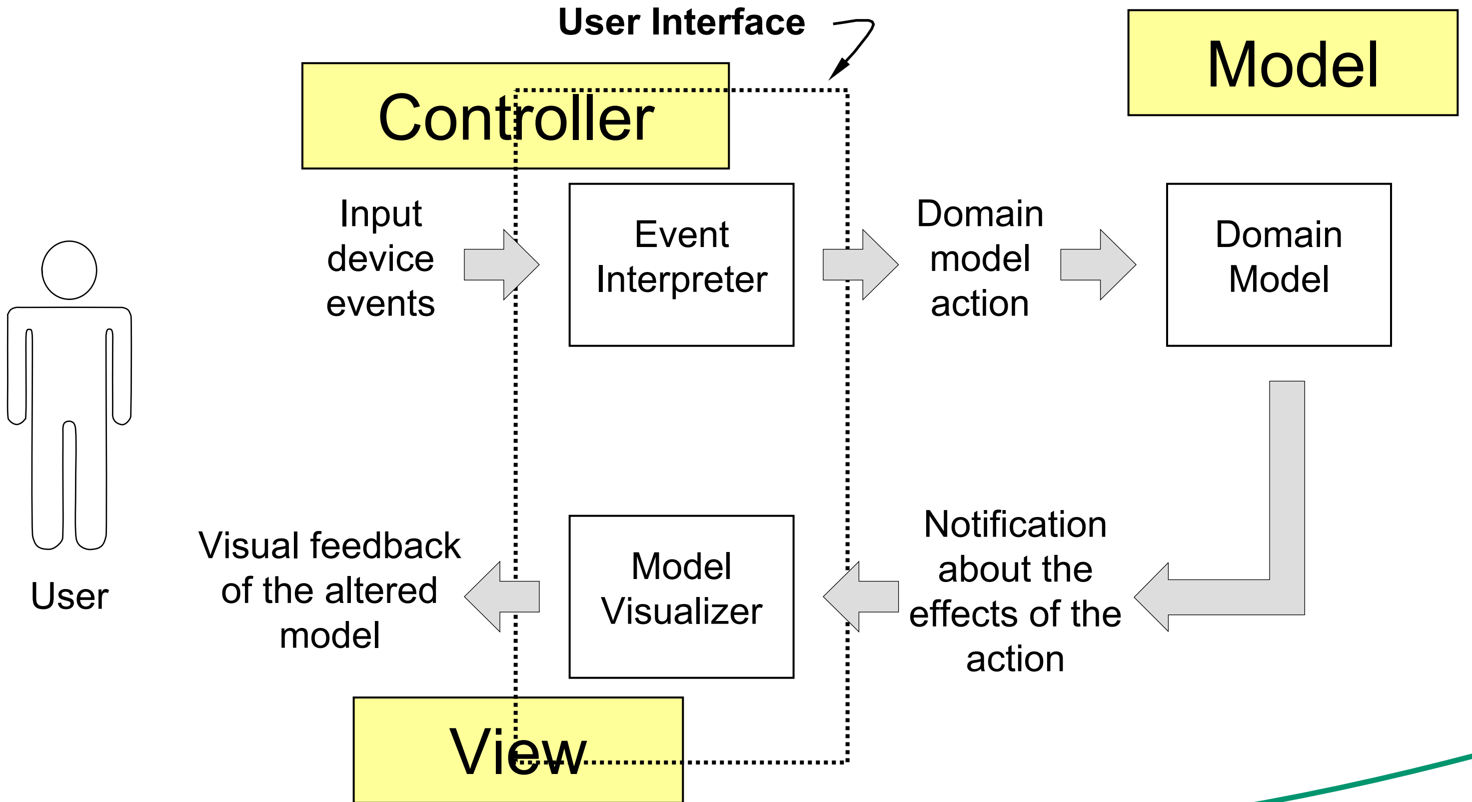
Compare

Architecture Style: Model-View-Controller

- **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

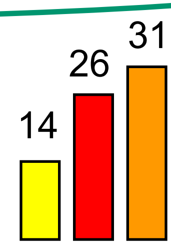


Model-View-Controller

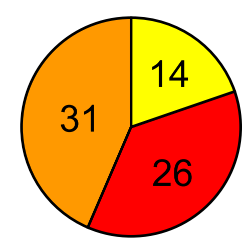


Model: array of numbers [14, 26, 31]

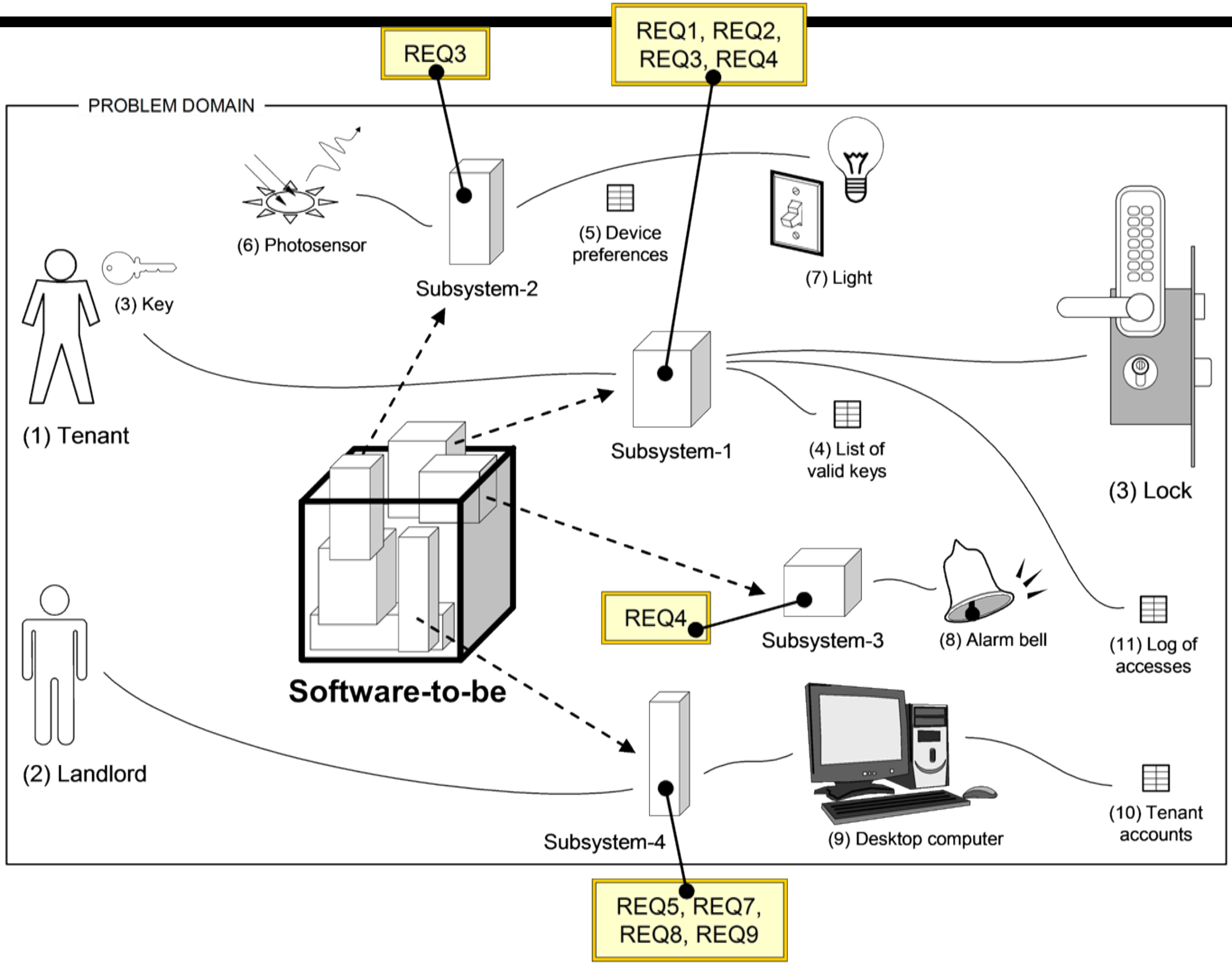
➔ Different Views for the same Model:



versus



Problem Structure

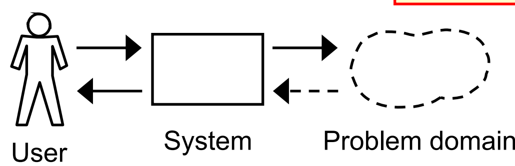


Subsystems derived from the requirements (“bottom-up” approach or induction)

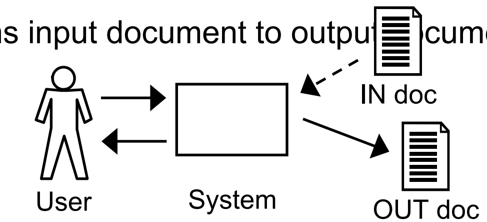
Typical Software Eng. Problems

1. User works with computer system
(problem domain is "virtual", not physical)

REQ-1: Map input data to output data as said by given rules

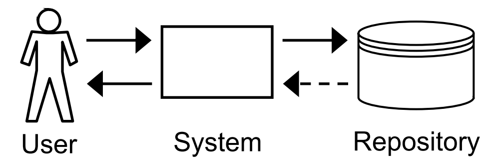


1.a) System transforms input document to output document

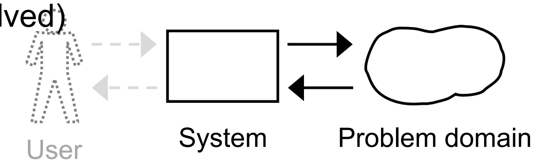


1.b) User edits information stored in a repository

REQ-2: Allow repository editing, where "repository" is a collection of data

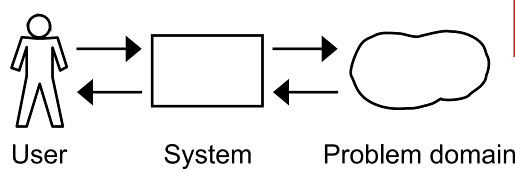


2. Computer system controls the (physical) problem domain
(user not involved)



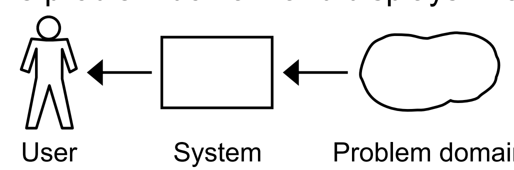
REQ-3: Autonomously control a physical object/device

3. Computer system intermediates between the user and the problem domain



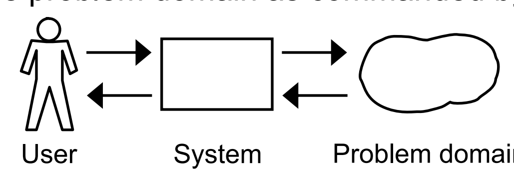
3.a) System observes the problem domain and displays information

REQ-5: Monitor and display information about an object

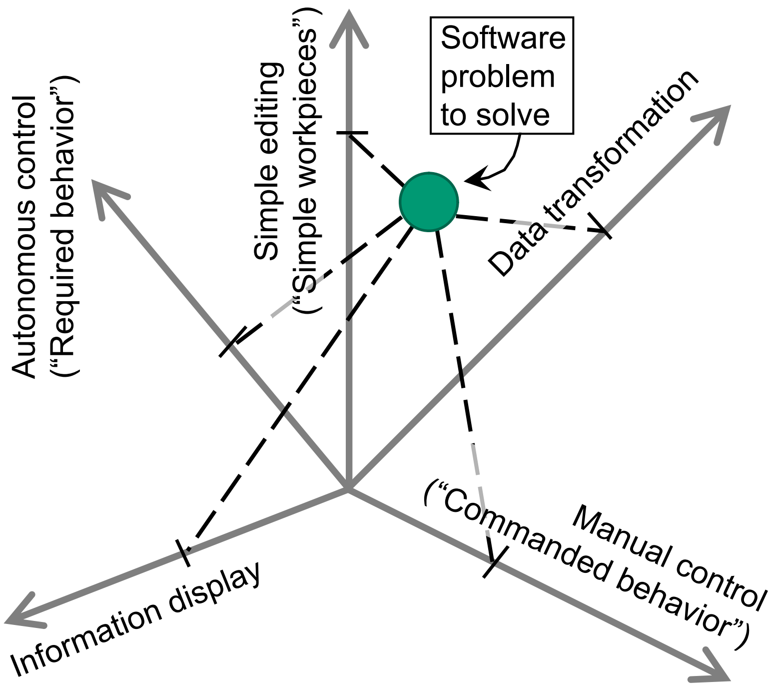


3.b) System controls the problem domain as commanded by the user

REQ-4: Interactively control a physical object/device



5-dimensional Problem Space



- The five elementary problem types represent the coordinate system of the problem space
- The "axis" projections represent the degree to which the whole problem contains a subproblem of this type
- Each subproblem can be analyzed independently and eventually recombined into the whole problem
- The structure of the solution should be selected to fit the problem structure

Software (i.e., Solution) Structure