Software Platform

Lecture 5

Emanuele Di Buccio

11/03/2024

Master Degree in Computer Engineering, A.A. 2023/2024

In the previous lecture . . .

Previous lecture topics

- TDD Example (Java)
- GRASP
- GoF Patterns
- Adapter Patterns

More on the Adapter Pattern (1)

- Adapter is a structural pattern
- **Intent**: Convert the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
- **Also known as** Wrapper
- **Motivation**: Sometimes a toolkit class that's designed for reuse isn't reusable only because its interface doesn't match the domain-specific interface an application requires.

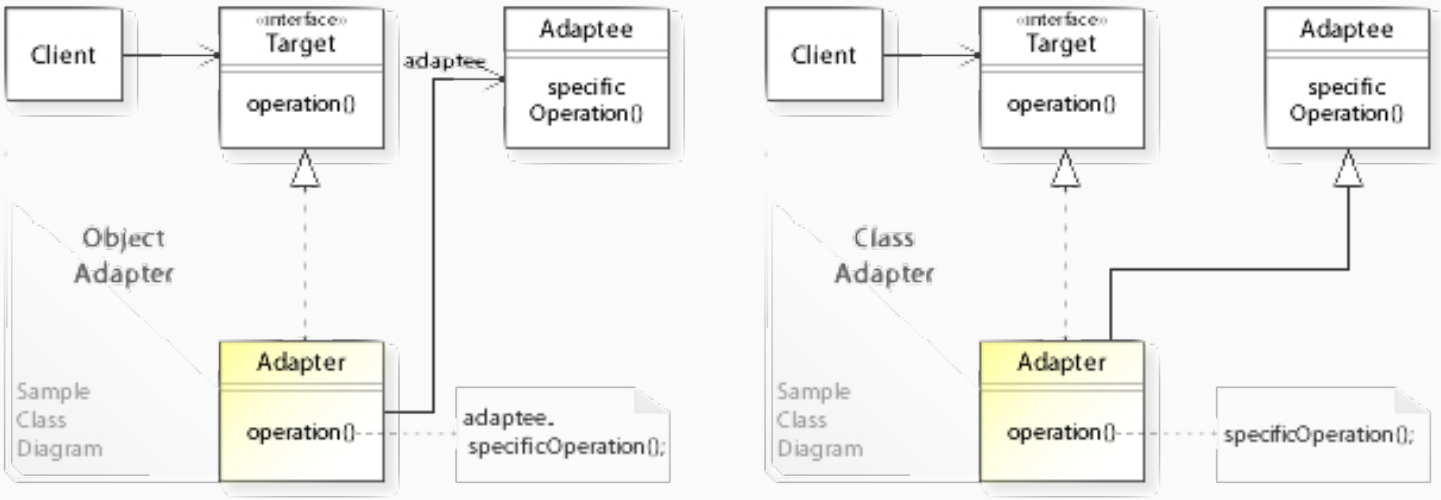
  How can existing and unrelated classes [. . .] work in an application that expects classes with a different and incompatible interface?

More on the Adapter Pattern (2)

- Applicability
  - you want to use an existing class, and its interface does not match the one you need
  - you want to create a reusable class that cooperates with unrelated or unforeseen classes, that is, classes that don't necessarily have compatible interfaces
  - (object adapter only) you need to use several existing subclasses, but it's impractical to adapt their interface by subclassing every one. An object adapter can adapt the interface of its parent class

More on the Adapter Pattern (3)

- Structure



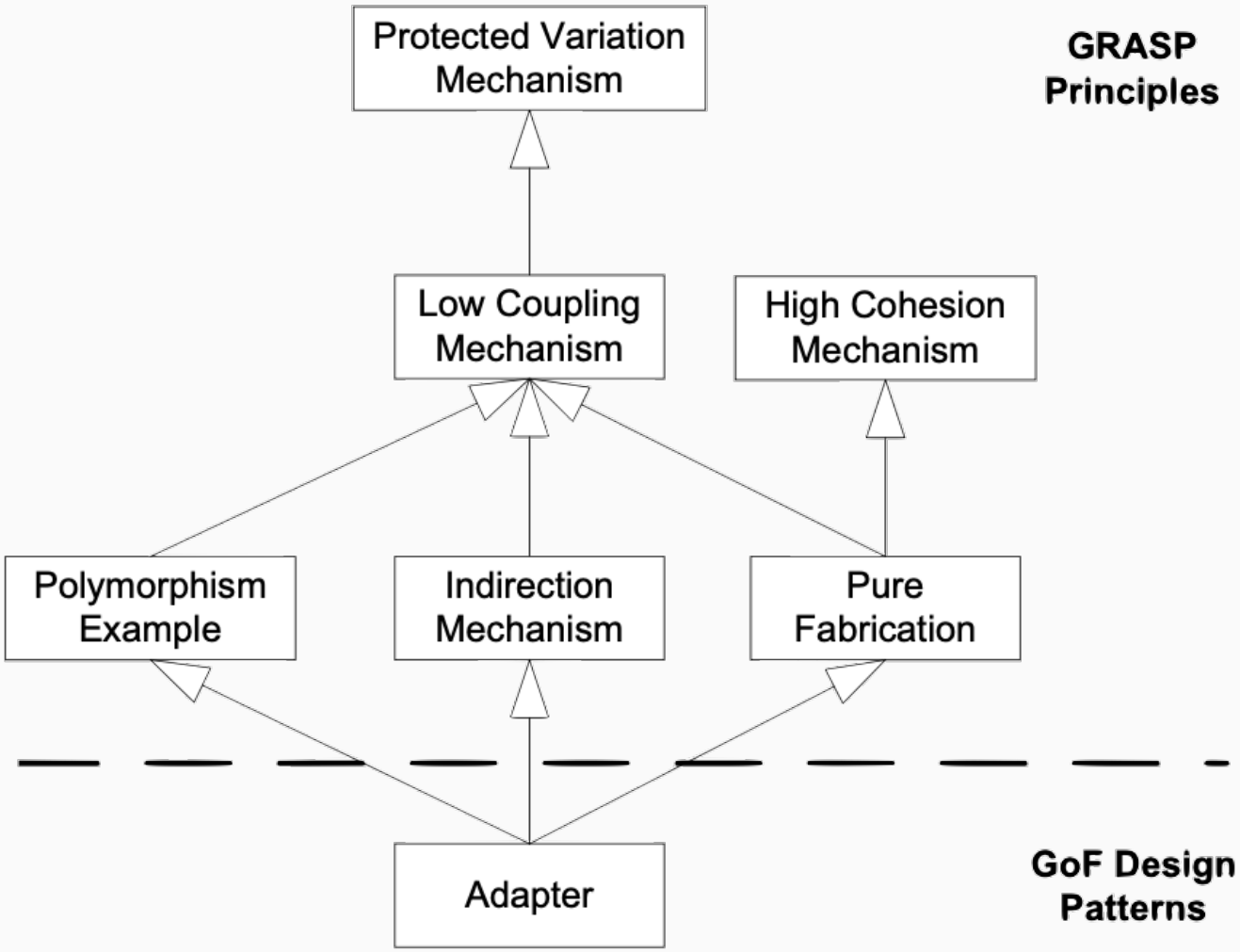
More on the Adapter Pattern (4)

- Participants
  - **Target**: defines the domain-specific interface that the client uses
  - **Client**: collaborates with objects conforming to the Target interface
  - **Adaptee**: defines an existing interface that needs adapting
  - **Adapter**: adapts the interface of Adaptee to the Target interface

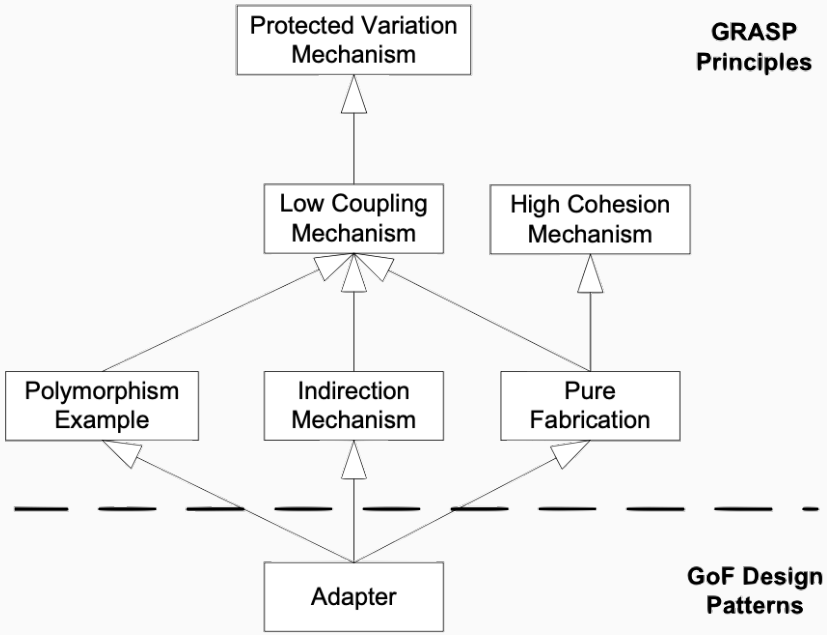
GRASP Principles and GoF Patterns (1)

- **Name**: Protected Variations
- **Problem**: How to design objects, subsystems, and systems so that the variations or instability in these elements does not have an undesirable impact on the other elements?
- **Solution**: Identify points of predicted variation or instability; assign responsibilities to create a stable interface around them.

GRASP Principles and GoF Patterns (2)



GRASP Principles and GoF Patterns (3)



"Adapter supports Protected Variations *with respect to changing external interfaces or third-party packages through the use of an* Indirection *object that applies interfaces and* Polymorphism." [Larman, 2004]

GRASP Principles and GoF Patterns (4)

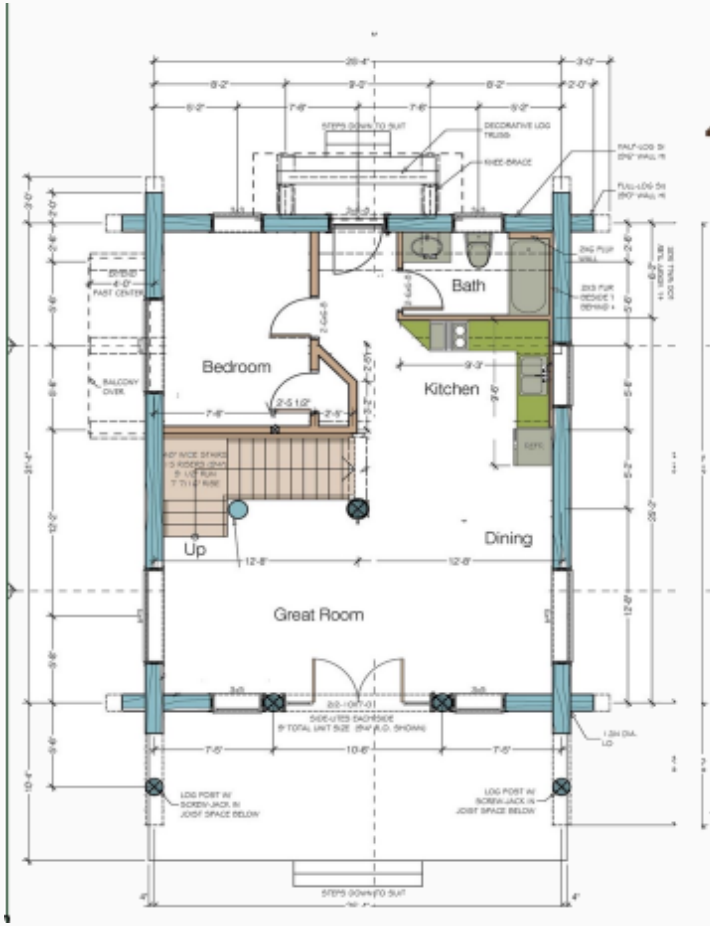
- Low coupling is a way to achieve protection at a variation point
- An indirection is a way to achieve low coupling
- Polymorphism is a way to achieve protection at a variation point, and a way to achieve low coupling
- The adapter pattern is a kind of indirection and a pure fabrication that uses polymorphism

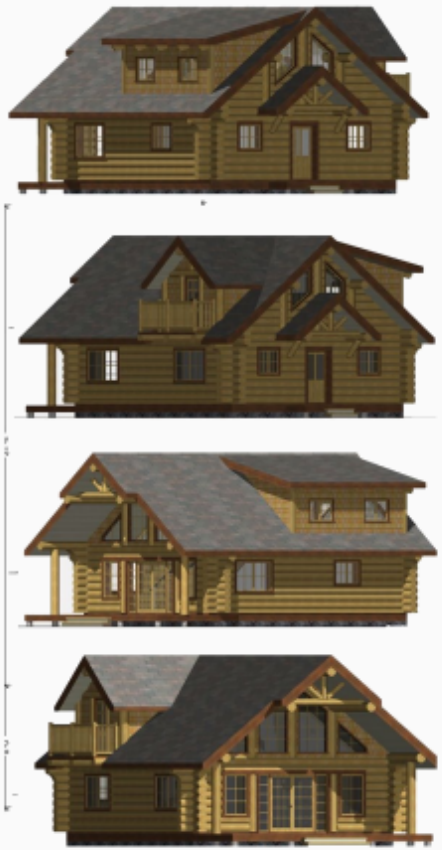
Architecture

Architectural Design

- understanding how a software system should be organized and designing the overall structure of that system
- critical link between design and requirements engineering
  - identification of the main structural components in a system and the relationship between them
- output: **architectural model**
  - describes how the system is organized as a set of communicating components

Why Architecture?





Architecture

- How deep the foundation has to be?
- How many pillars are needed?
- What are their sizes?
- Which construction material to be used?
- On what ratio material has to be mixed?
- How would the home look?
- What features would it have?
- How the home will be secured? What type of security systems is needed?
- Etc

Design

- Structural design
- Electrical design
- Utility design
- Interior design

Development

- Construction of the home

Production

- Hand over the property

Software Architect

- **Role**: Software Architect
- **Owns**: Overall application architecture and design
- **Works with**: Project Managers, Business Analysts, Designers, Developers, Testers, Infrastructure Architects
- **Defines** standards for Application Design, UI, Coding, Coding Templates, Naming convention, Software Patterns, Security, . . .
- **Produce**:
  - Application architecture (includes layering – both physical & logical, interfaces with external systems, etc.)
  - High and low level design's
  - Database Model & Design

Definitions of Software Architecture (1)

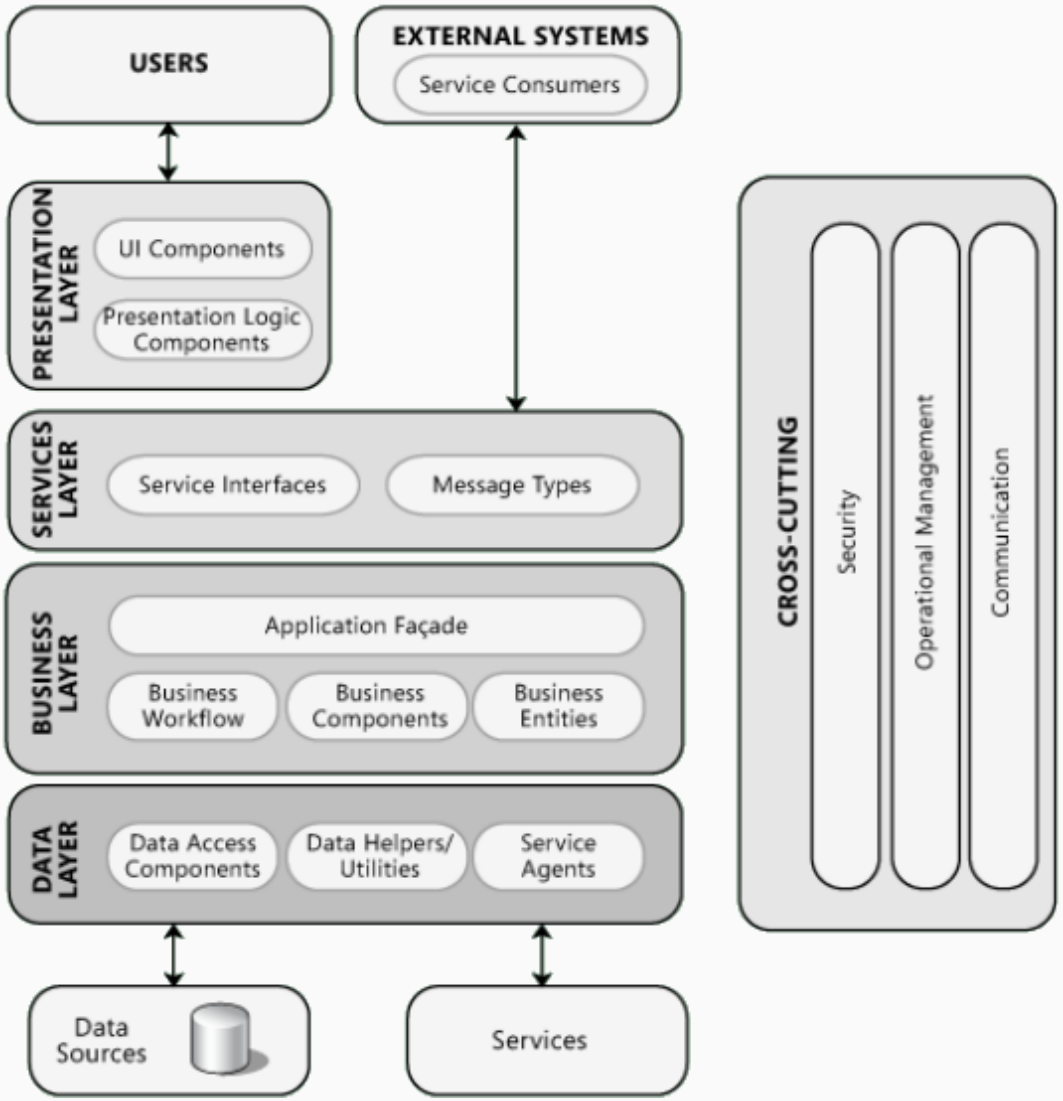
- *"The software architecture of a system is the set of structures needed to reason about the system, which comprises software elements, relations among them, and properties of both."* [Bass et al., 2012]
- In essence, an application's architecture is
  - its decomposition into parts (the elements) and
  - the relationships (the relations) between those parts

Definitions of Software Architecture (2)

Why is the **decomposition** important?

- it facilitates the division of labor and knowledge
  - ▷ It enables multiple people (or multiple teams) with possibly specialized knowledge to work productively together on an application
- it defines how the software elements interact

Software Architecture Example



Levels of abstraction

Design at two levels:

- Architecture in the small
  - concerned with the architecture of individual programs
  - concerned with the way that an individual program is decomposed into components
- Architecture in the large
  - concerned with the architecture of complex enterprise systems that include other systems, programs, and program components
  - may be distributed over different computers, which may be owned and managed by different companies

Why Architecture Matters

Two categories of requirements (for an application):

- *functional*
  - define what an application must do
  - usually in the form of use cases or user stories
  - can be implemented with many architectures
- *quality of service*
  - also known as *quality attributes* or *-ilities*
  - **runtime attributes** (e.g., scalability, reliability)
  - **development time qualities** (e.g., maintainability, testability, employability)

The Architecture we choose determines how well it meets quality requirements.

Advantages of Making the Architecture Explicit

- Stakeholder communication
  - Architecture may be used as a focus of discussion by system stakeholders
- System analysis
  - Means that analysis of whether the system can meet its non-functional requirements is possible
- Large-scale reuse
  - The architecture may be reusable across a range of systems
  - Product-line architectures may be developed

Architectural Representation (1)

- **modelled informally** using simple block diagrams
  - box $\rightarrow$ component
  - boxes within boxes $\rightarrow$ components decomposed to subcomponents
  - arrows $\rightarrow$ data/control signals passed from component to component
- **disadvantanges** of informal representations
  - don't show the type of the relationships among system components
  - don't show components' externally visible properties

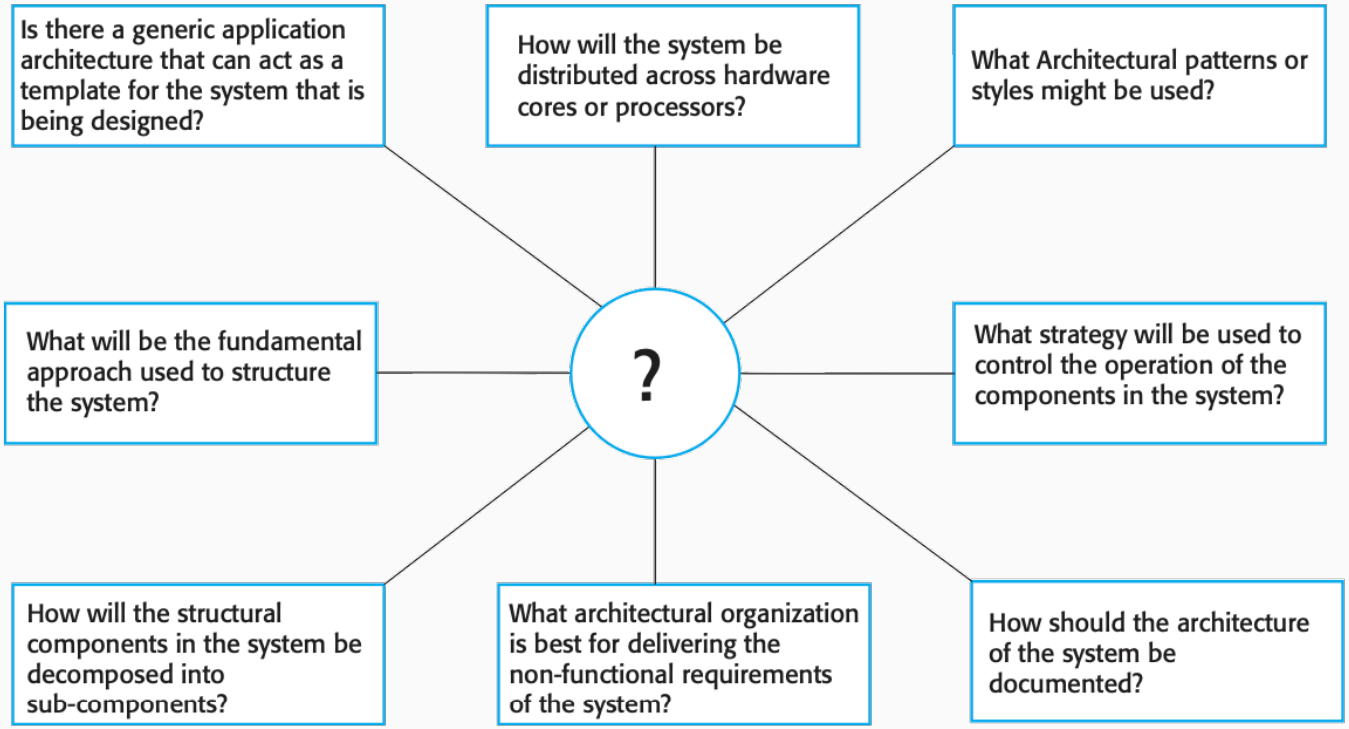
Architectural Representation (2)

- **architectural theory vs industrial practice** because of the usage of architectural models:
  - encouraging discussions about the system design
  - documenting an architecture that has been designed

Architectural Design (1)

- Architectural Design is a creative process
- Design system organization that will satisfy
  - functional requirements
  - non-functional requirements
- No formulaic architectural design process
- A number of structural decisions to make

Architectural Design (2)



From [Sommerville, 2016]

Architectural Views

Architectural Views (1)

- Architectural Models used to
  - focus discussion about the software requirements or design
  - document a design so that it can be used as a basis for more detailed design and implementation of the system
- Questions
  - What views or perspectives are useful when designing and documenting a system's architecture?
  - What notations should be used for describing architectural models?

Architectural Views (2)

First, we need to answer to this question: **What is a view?**

> *A* **view** *is a representation of a coherent set of architectural elements, as written by and read by system stakeholders. It consists of a representation of a set of elements and the relations among them. [Bass et al., 2012]*

View $\neq$ Structure

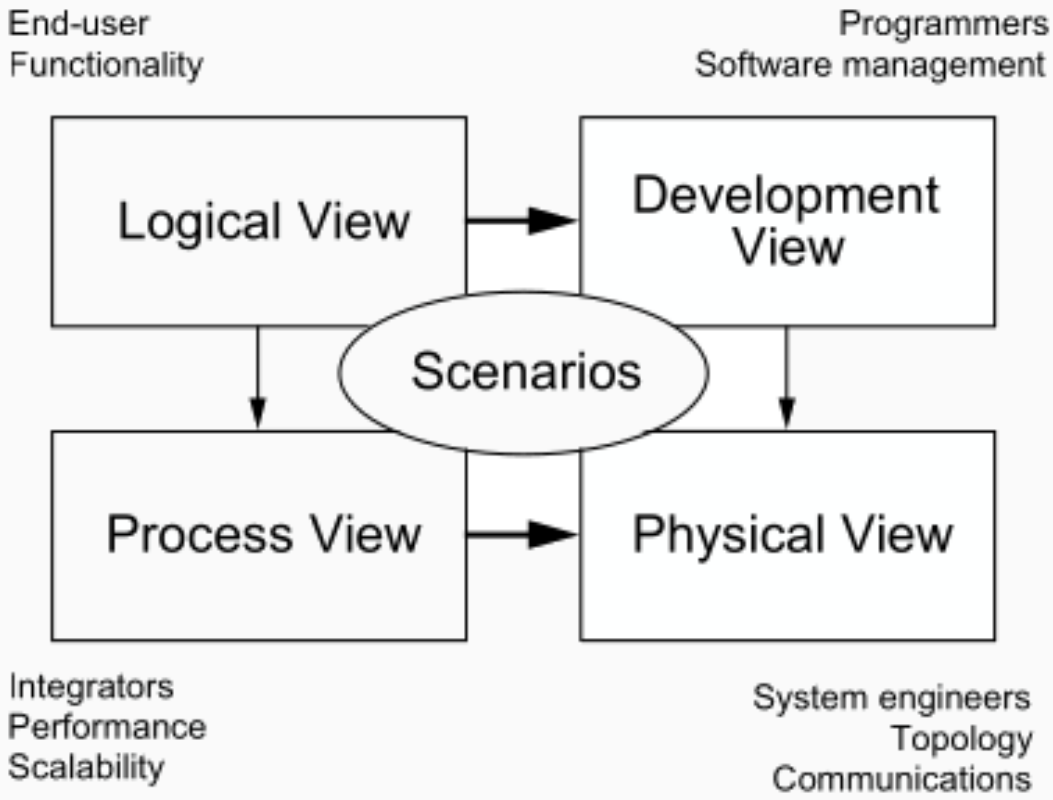
- **view**: representation of a coherent set of architectural elements
- **structure**: the set of elements itself, as they exist in software or hardware

Architectural Views (3)

- Impossible to represent all the relevant information about a system's architecture in a single diagram
- We can use *multiple views*
- Which views?
  - diverse opinions
  - possible approach: **4+1 view model** [Kruchten, 1995]

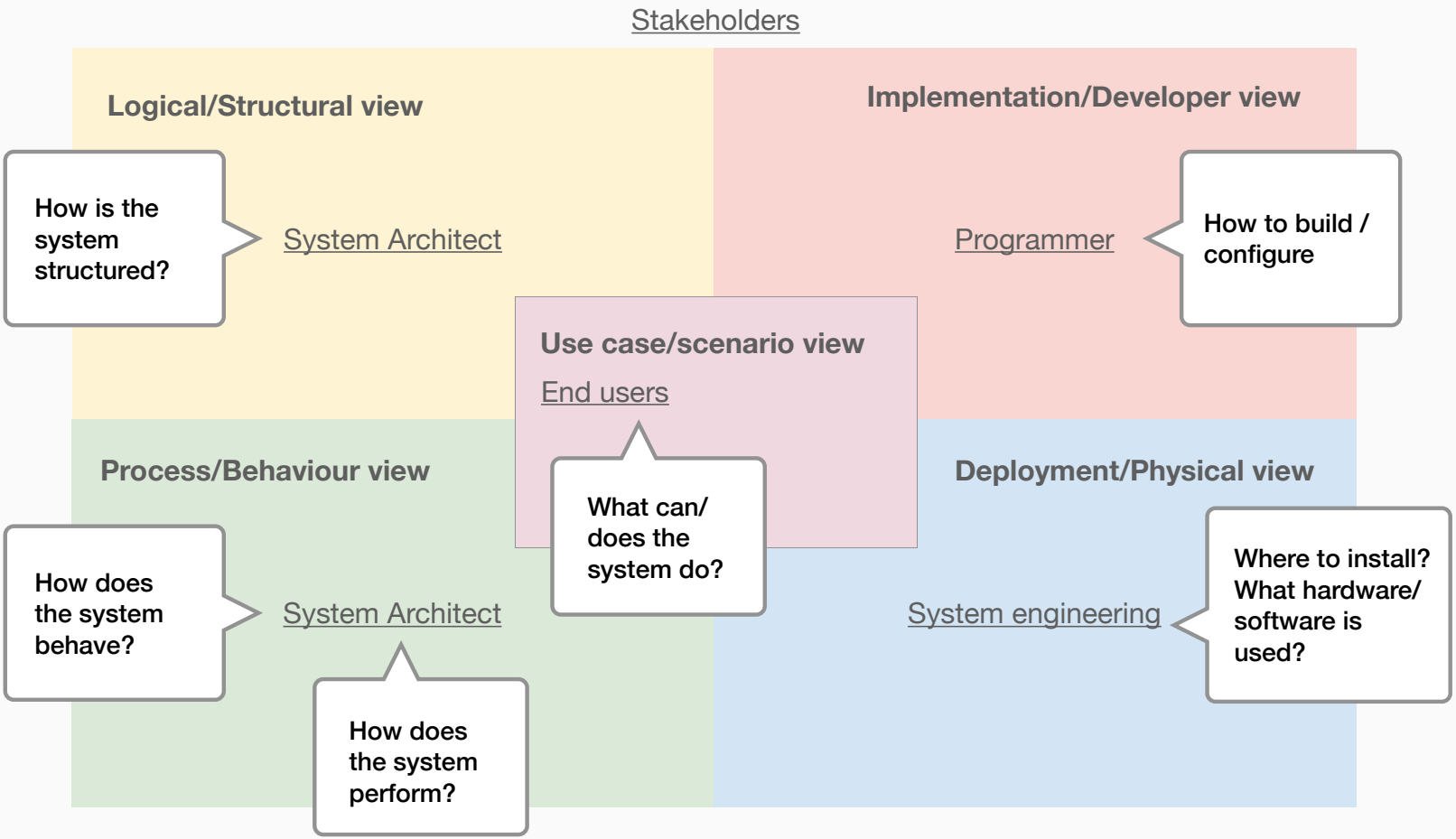
4+1 View Model (1)

Four fundamental architectural views, linked through common use cases or scenarios:



From [Kruchten, 1995]

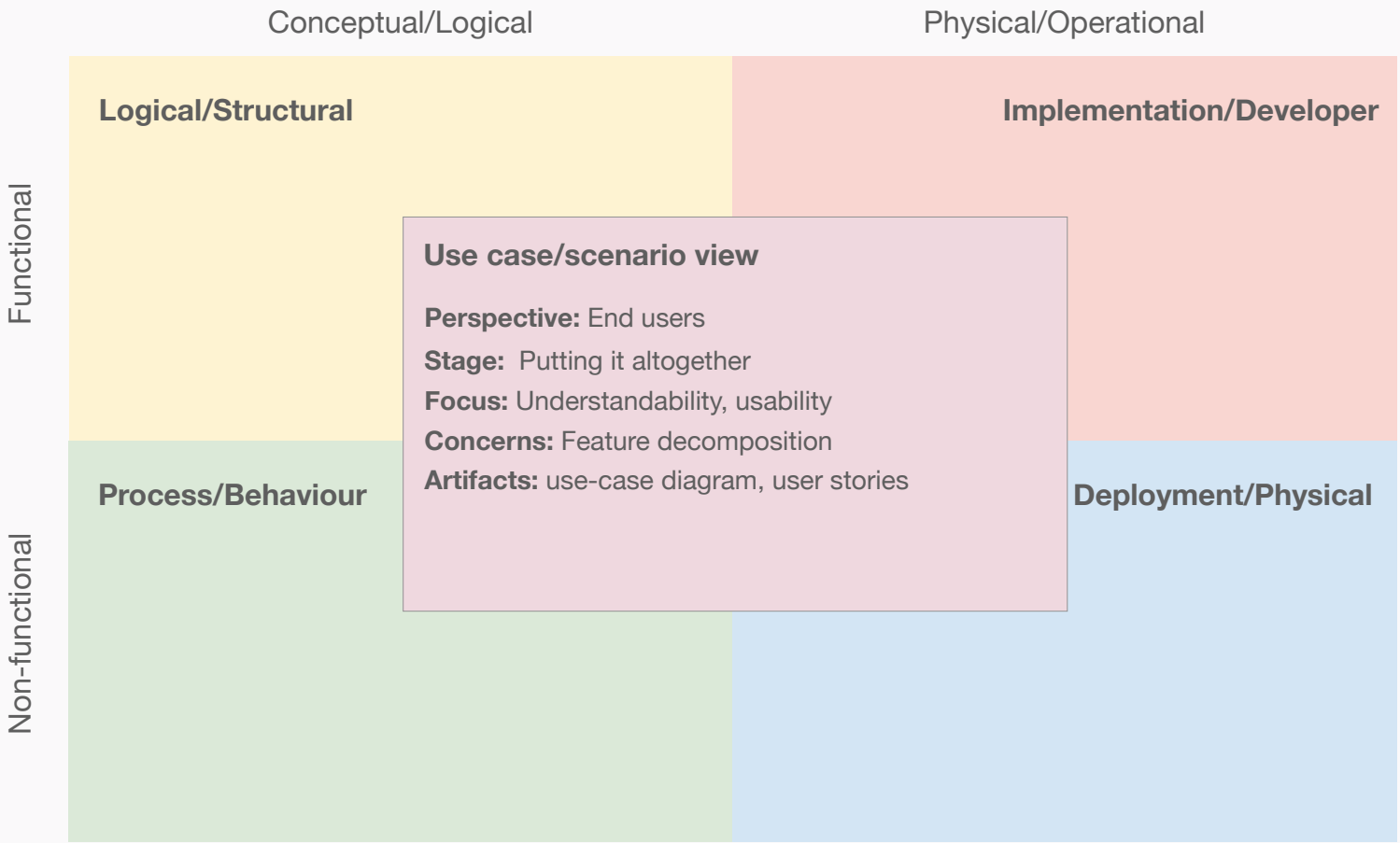
4+1 View Model (2)



4+1 View Model (3)

	Conceptual/Logical	Physical/Operational
Functional	**Logical/Structural view** **Perspective:** Analysts, Designers **Stage:** Requirement Analysis **Focus:** Object Oriented Decomposition **Concerns:** Functionality **Artifacts:** Class diagram, Object diagram, Composite structure diagram	**Implementation/Developer view** **Perspective:** Developers, project managers **Stage:** Design **Focus:** Subsystem decomposition **Concerns:** Software management **Artifacts:** Component diagram, package diagram
Non-functional	**Process/Behaviour view** **Perspective:** System integrators **Stage:** Design **Focus:** Process decomposition **Concerns:** Performance, scalability, throughput **Artifacts:** sequence diagram, communication diagram, activity diagram, state diagram, interaction overview diagram, timing diagram	**Deployment/Physical view** **Perspective:** System Engineers **Stage:** Design **Focus:** Map software to hardware **Concerns:** System topology, delivery, installation, communication **Artefatti:** Deployment diagram, Network topology (not UML)

4+1 View Model (4)



4+1 View Model (5)

- Logical/Structural View
- Process/Behaviour View
- Implementation/Developer View
- Deployment/Physical View
- Use case/scenario View

4+1 View Model (6)

- Logical/Structural View
  - shows the key abstractions in the system as objects or classes
  - helps relate the system requirements to entities
  - “describes the design’s object model when an OO design method is used. To design an application that is very data-driven, you can use an alternative approach to develop some other form of logical view, such as an entity-relationship diagram.” [Kruchten, 1995]
- Process/Behaviour View
- Implementation/Developer View
- Deployment/Physical View
- Use case/scenario View

4+1 View Model (7)

- Logical/Structural View
- Process/Behaviour View
  - shows how, at runtime, the system is composed of interacting processes
  - useful for making judgments about non-functional system characteristics such as performance and availability
  - “describes the design’s concurrency and synchronization aspects” [Kruchten, 1995]
- Implementation/Developer View
- Deployment/Physical View
- Use case/scenario View

4+1 View Model (8)

- Logical/Structural View
- Process/Behaviour View
- **Implementation/Developer View**
  - shows how the software is decomposed for development, i.e., the breakdown of the software into components that are implemented by a single developer or development team
  - useful for software managers and programmers
  - “describes the software’s static organization in its development environment” [Kruchten, 1995]
- Deployment/Physical View
- Use case/scenario View

4+1 View Model (9)

- Logical/Structural View
- Process/Behaviour View
- Implementation/Developer View
- **Deployment/Physical View**
  - shows the system hardware and how software components are distributed across the processors in the system
  - view is useful for systems engineers planning a system deployment
  - “describes the mapping of the software onto the hardware and reflects its distributed aspect” [Kruchten, 1995]
- Use case/scenario View

4+1 View Model (10)

- Logical/Structural View
- Process/Behaviour View
- Implementation/Developer View
- Deployment/Physical View
- Use case/scenario View
  - redundant; why introduce it?
  - helps designers discover architectural elements during the architectural design phase
  - helps to validate and illustrate the architecture design; it can serve as a starting point for testing a prototype of the architecture

Architectural Patterns

Architectural Patterns (1)

- Patterns
  - are a means of representing, sharing and reusing knowledge
  - should include information about when they are and when they are not useful
  - may be represented using tabular and graphical descriptions
- An architectural pattern is a **stylized description** of good design practice that has been tried and tested in different environments
- Although each software system is unique, systems in the same application domain often have similar architectures that reflect the fundamental concepts of the domain

Architectural Patterns (2)

We should be aware of

- common architectural patterns
- where they can be used
- their strengths
- their weakness

Choice of the architectural pattern

When making the choice, we should consider the non-functional requirements of the system

- Performance
- Security
- Safety
- Availability
- Maintainability

(Some) Architectural Patterns

- Layered Pattern
- Repository Pattern
- Client-server Pattern
- Pipe-and-filter Pattern

(Some) Architectural Patterns

- Layered Pattern
- Repository Pattern
- Client-server Pattern
- Pipe-and-filter Pattern

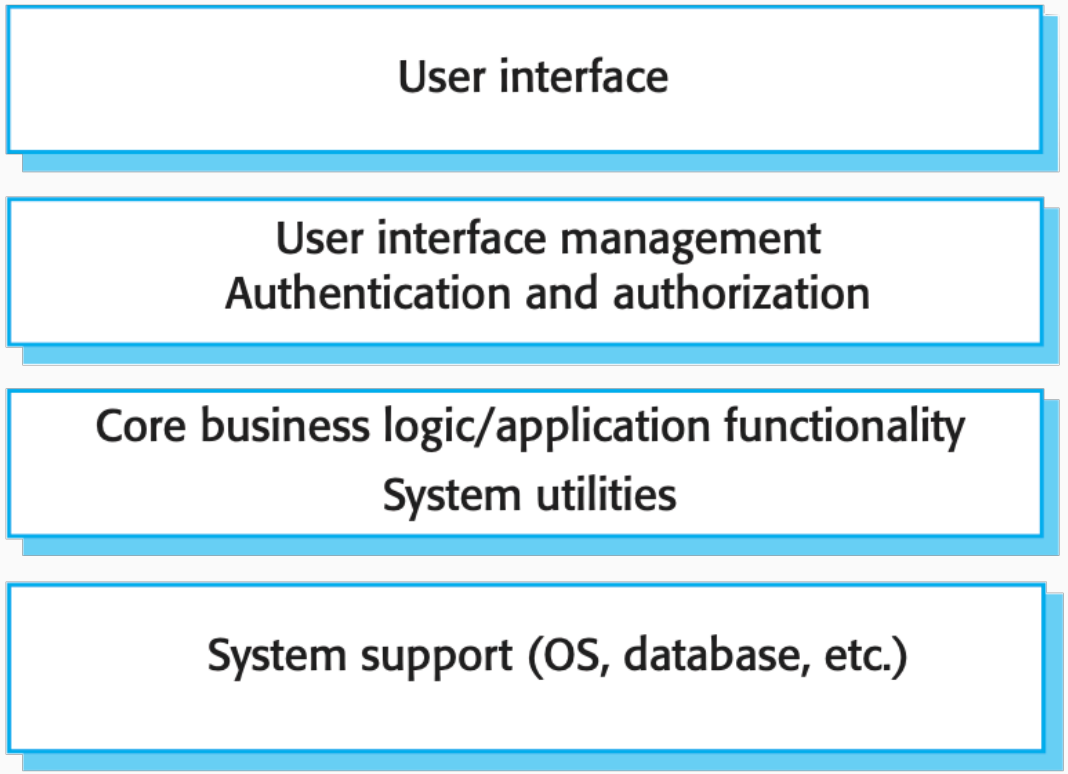
Layered architecture (1)

Name	Layered architecture
Description	Organizes the system into layers, with related functionality associated with each layer. A layer provides services to the layer above it, so the lowest level layers represent core services that are likely to be used throughout the system.
Example	A layered model of a digital learning system to support learning of all subjects in schools
When used	Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multilevel security.
Advantages	Allows replacement of entire layers as long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.
Disadvantages	In practice, providing a clean separation between layers is often difficult, and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

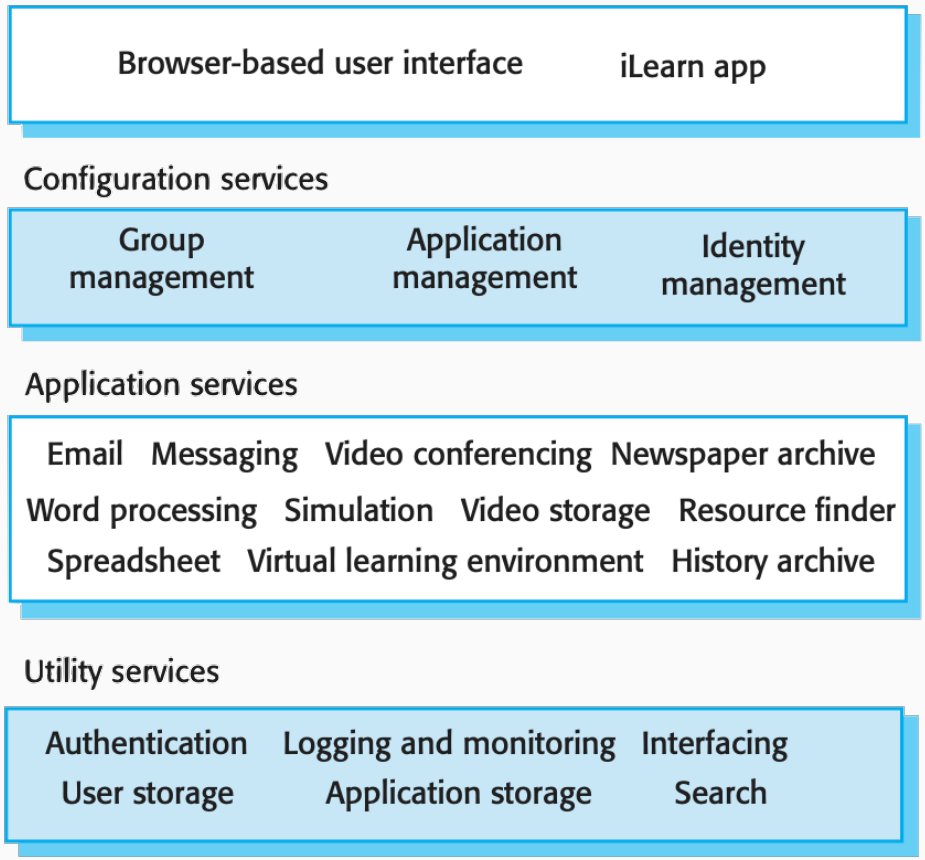
Layered architecture (2)

- also known as **n-tier architecture pattern**
- a way of achieving separation and independence
- supports the incremental development of systems
  - as a layer is developed, some of the services provided by that layer may be made available to users
- changeable and portable architecture
  - If its interface is unchanged, a new layer with extended functionality can replace an existing layer without changing other parts of the system
  - when layer interfaces change or new facilities are added to a layer, only the adjacent layer is affected

Layered architecture (3)



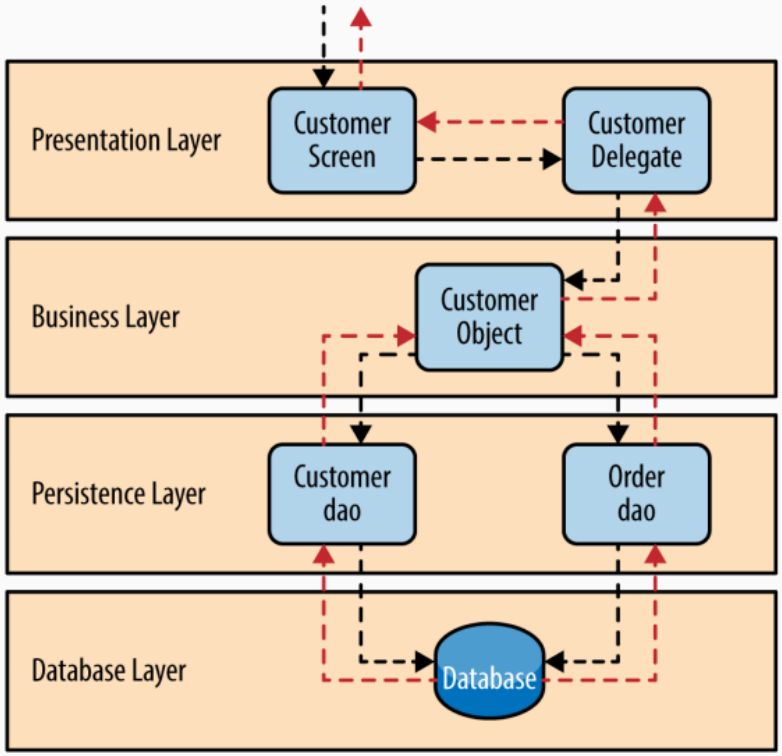
Layered architecture (4)



iLearn Architecture [Sommerville, 2016]

Layered architecture (5)

- Presentation layer (aka UI layer)
- Application layer (aka service layer)
- Business logic layer (aka domain layer)
- Data access layer (aka persistence layer)



(Some) Architectural Patterns

- Layered Pattern
- Repository Pattern
- Client-server Pattern
- Pipe-and-filter Pattern

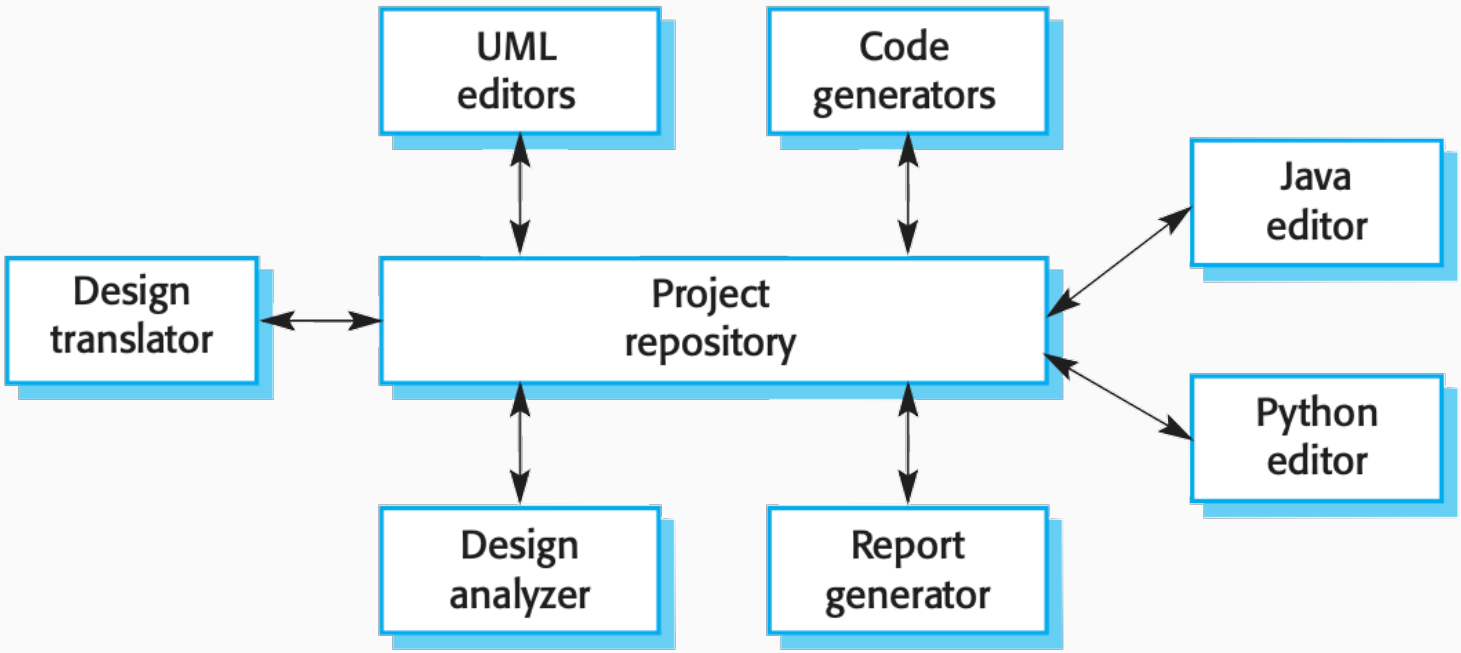
Repository Pattern (1)

- how a set of interacting components can share data
- majority of systems that use large amounts of data are organized around a shared database or repository
- the **repository pattern** is suited for applications in which data is generated by one component and used by another
  - Management information systems
  - Computer-Aided Design (CAD) systems
  - Interactive development environments for software

Repository Pattern (2)

Name	Repository Pattern
Description	All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.
Example	IDE where the components use a repository of system design information. Each software tool generates information, which is then available for use by other tools.
When used	You should use this pattern when you have a system in which large volumes of information are generated that has to be stored for a long time. You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool.
Advantages	Components can be independent; they do not need to know of the existence of the others. Changes made by one component can be propagated to all components. All data can be managed consistently (e.g., backups done at the same time) as it is all in one place.
Disadvantages	The repository is a single point of failure so problems in the repository affect the whole system. May be inefficiencies in organizing all communication through the repository. Distributing the repository across several computers may be difficult.

Repository Pattern (3)



Repository Pattern (4)

Two possible approaches:

- **passive**: the repository is passive and control is the responsibility of the components using the repository
- **blackboard model**: triggers components when particular data become available

(Some) Architectural Patterns

- Layered Pattern
- Repository Pattern
- Client-server Pattern
- Pipe-and-filter Pattern

Client–server architecture (1)

- Repository pattern
  - concerned with the static structure of a system
  - not show its runtime organization
- Client–Server pattern is organized as
  - a set of services and associated servers
  - clients that access and use the services

Client–server architecture (2)

- Set of servers
  - offer services to other components
  - Examples:
    - print servers that offer printing services
    - file servers that offer file management services
- Set of clients
  - call on the services offered by servers
  - (normally) several instances of a client program executing concurrently on different computers
- Network
  - allows the clients to access these services
  - client-server systems are usually implemented as distributed systems, connected using Internet protocols

Client–server architecture (3)

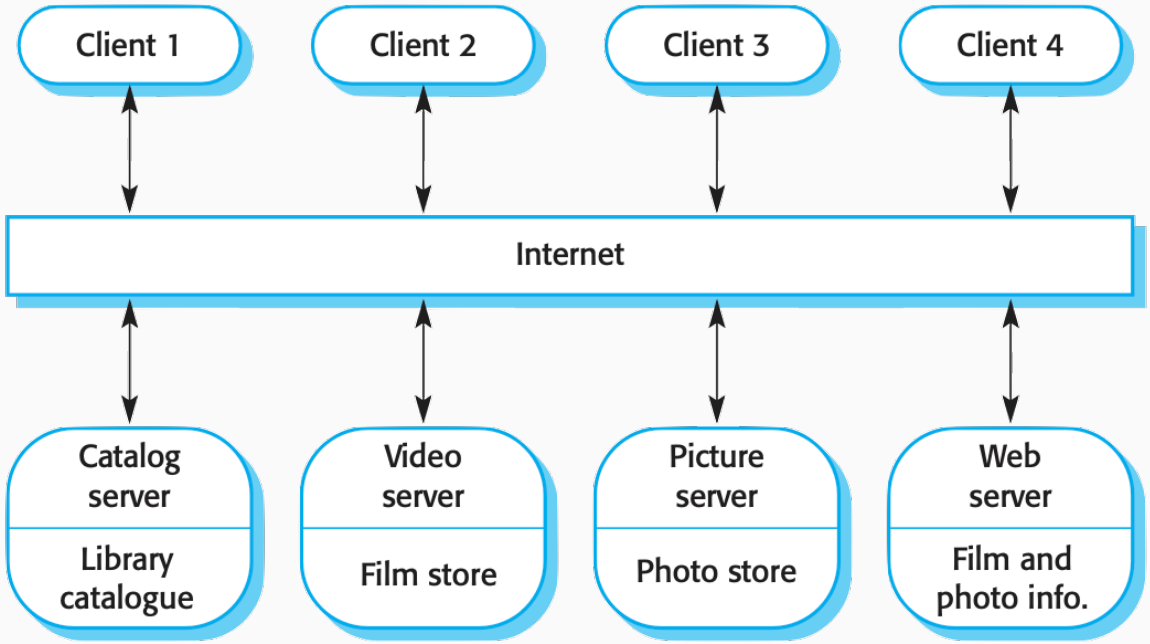
- The logical model of independent services running on separate servers can be implemented on a single computer
- Important **benefit**: Separation and independence
  - ▷ services and servers can be changed without affecting other parts of the system

Client–server architecture (4)

Name	Client–server architecture
Description	In a client–server architecture, the system is presented as a set of services, with each service delivered by a separate server. Clients are users of these services and access servers to make use of them.
Example	A film and video/DVD library organized as a client–server system
When used	Used when data in a shared database has to be accessed from a range of locations. Because servers can be replicated, may also be used when the load on a system is variable.
Advantages	The principal advantage of this model is that servers can be distributed across a network. General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.
Disadvantages	Each service is a single point of failure and so is susceptible to denial-of-service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. Management problems may arise if servers are owned by different organizations.

Client–server architecture (5)

Example



Client–server architecture (6)

Example

- A multiuser, web-based system for providing a film and photograph library
- Several servers manage and display the different types of media
  - **video server** responsible for video compression and decompression in different formats; needed to transmit video quickly and in synchrony but at relatively low resolution
  - **picture server** to maintain photos at a high resolution
- client program is simply an integrated user interface constructed using a web browser to access these services

(Some) Architectural Patterns

- Layered Pattern
- Repository Pattern
- Client-server Pattern
- Pipe-and-filter Pattern

Pipe-and-filter Pattern (1)

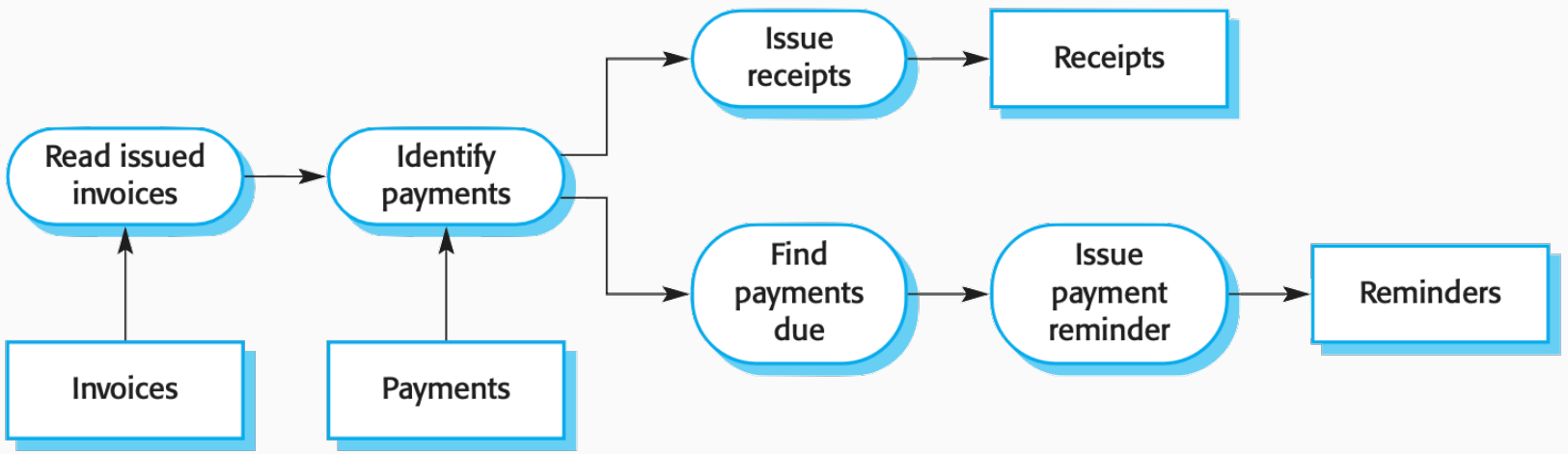
- Model of the runtime organization of a system where **functional transformations** process their inputs and produce outputs
- Data flows from one to another and is transformed as it moves through the sequence
- Each processing step is implemented as a transform
- Input data flows through these transforms until converted to output
- The transformations may execute sequentially or in parallel
- The data can be processed by each transform item by item or in a single batch

Pipe-and-filter Pattern (2)

Name	Pipe and filter
Description	The processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.
Example	Pipe and filter system used for processing invoices.
When used	Commonly used in data-processing applications (both batch and transaction-based) where inputs are processed in separate stages to generate related outputs.
Advantages	Easy to understand and supports transformation reuse. Workflow style matches the structure of many business processes. Evolution by adding transformations is straightforward. Can be implemented as either a sequential or concurrent system.
Disadvantages	The format for data transfer has to be agreed between communicating transformations. Each transformation must parse its input and unparse its output to the agreed form. This increases system overhead and may mean that it is impossible to reuse architectural components that use incompatible data structures.

Pipe-and-filter Pattern (3)

Example

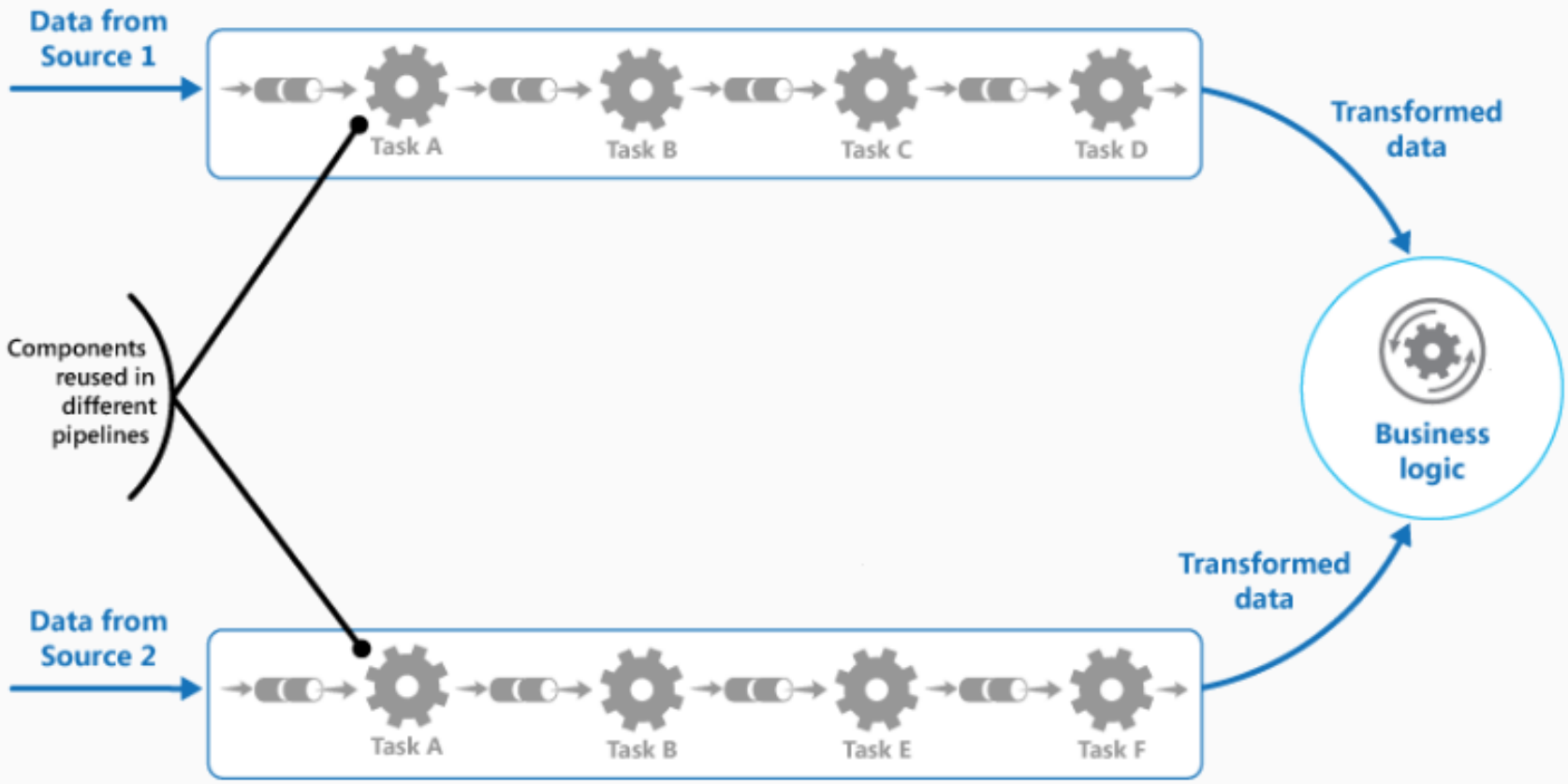


Pipe-and-filter Pattern (4)

Example

- batch processing application
- organization issues invoices to customers
- once a week, payments that have been made are reconciled with the invoices
  - invoice paid: a receipt is issued
  - invoice not been paid: a reminder is issued

Pipe-and-filter Pattern (5)



Pipe-and-filter Pattern (6)

- Pipes eliminate the need for intermediate files
- We can replace filters easily
- We can achieve different effects through recombination
- If the data stream is a standard, filters may be developed independently
- Parallelization is possible

References

References (1)

Bass, L., Clements, P., and Kazman, R. (2012).

Software Architecture in Practice.

Addison-Wesley Professional, 3rd edition.

Kruchten, P. (1995).

The 4+1 view model of architecture.

IEEE Softw., 12(6):42–50.

Larman, C. (2004).

Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition).

Prentice Hall PTR, USA.

References (2)

Sommerville, I. (2016).

Software Engineering.

Pearson, 10th - global edition edition.

Questions?