L13 - Introduction to architectural patterns (1)

CS3028 - Principles of Software Engineering

Ernesto Compatangelo

Department of Computing Science

13.1 Reminding past issues and mapping them to current topics

Where are we now?

Software development paradigms

⇒ The Unified Process (UP) paradigm

⇒ UP phases and UP disciplines (activities) within each phase

⇒ Elaboration (second UP phase)

⇒ · · · · · ·

⇒ Elaboration Design

⇒ Architectural design

⇒ Architectural patterns

⇒ · · · · · ·

13.2 From patterns to architectural patterns

Patterns: motivations

- Expert designers usually produce much better designs than novices in terms of both *basic* and *constructive* design principles
- Designs produced by experts are based on the implicit reuse of successful designs devised in the past to tackle similar problems
- Once successful designs for classes of general, recurrent problems are identified and properly documented, they can be reused by novices to produce professional results
- Patterns outline reusable design solutions to specific problems, abstracting from their concrete form which recurs in different contexts

Patterns: taxonomy

- **Architectural patterns (aka architectural styles)** are high-level models for subsystems which describe kinds of architectural design components and their mutual interactions
- **(Detailed) design patterns** are mid-level design patterns at an intermediate level of abstraction and model collaboration between several modules/units (typically classes)
- **(Abstract) data structures and algorithms** are patterns for storing, retrieving, and manipulating data and model single modules/units (typically single classes) during the last, most in-depth design stage

Architectural patterns

- A general model used as a starting point for system design.
- Defines system decomposition, global control flow, handling of boundary conditions, inter-subsystem communication
- There are only a few architectural patterns around:
  - layered architectures
  - pipe and filter
  - shared-data
  - event-driven
  - model-view-controller

13.3 The layered architectural pattern

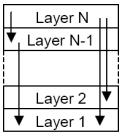
The layered architecture pattern in general

- A system is divided into an array of modules (layers)
- Each layer represents a different level of abstraction
- Each layer *provides services* to the layer(s) 'above' and *makes use of services* from the layer(s) 'below'
- Each layer is a module that provides a cohesive set of *services* through a well-defined *interface*
- Layered architectures are **open** (aka *relaxed*) if each layer provides services to and/or makes use of services from two or more other layers
- Layered architectures are **closed** (aka *strict*) if each layer only provides services to the layer immediately above it or makes use of services from the layer immediately below it

Open and closed layered architectures

Each layer corresponds to one or more sub-systems

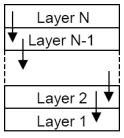
Open: messages can be sent to any lower layer



Open

- More compact code
- No need for extra code to pass messages through each layer
- Increases layer dependencies layers and hence maintenance overhead

Closed: messages can only be sent to adjacent lower layer



Closed

- Minimises layer dependencies
- Reduces the impact of a change to the interface of any one layer

Widely used layered architectures

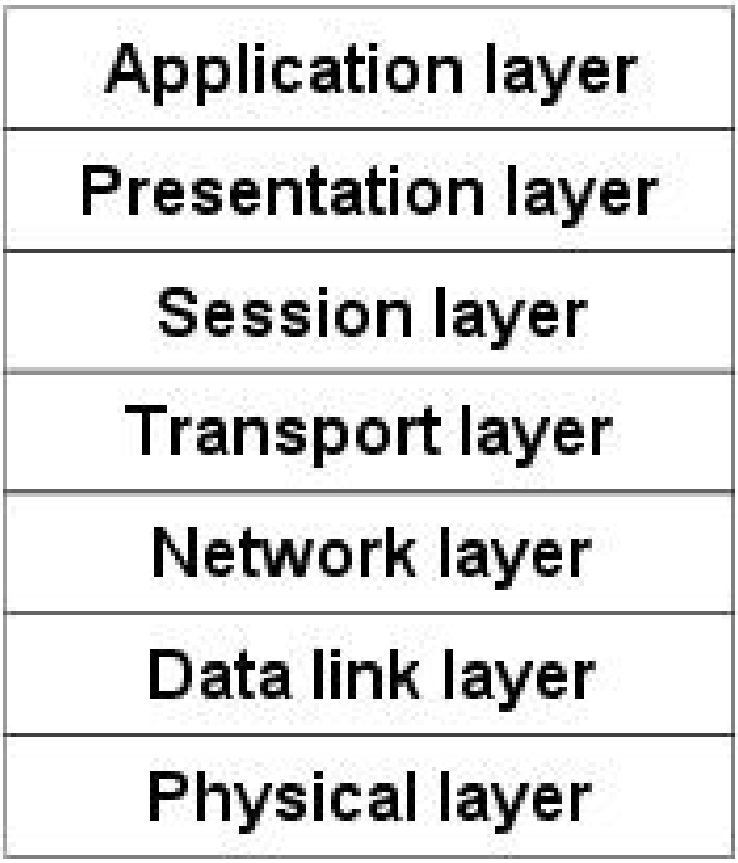
- **Three-layer** (historically three-tier) architecture: commonly used for business-oriented information systems

Presentation layer
Application layer
Storage layer

- **Four-layer** (historically four-tier) architecture: Separates business logic into application logic and domain layers so that several applications can share the same domain. Commonly used in big corporate information systems (*e.g.*, the UoA IS)

Presentation layer
Application layer
Domain layer
Storage layer

Layered architecure example: The ISO-OSI strict closed layered architecture



Advantages of layered architectures

- Layers are by definition highly cohesive, thus maximising cohesion
- Layers support information hiding
- Layers are constrained to use only lower layers, *i.e.*, they are not strongly coupled to the layers above them. Hence, layered architectures decrease coupling. Coupling is minimised by closed layered architectures
- Layering helps to sub-divide complex system parts into separate modules, simplifying the system and supporting changeability and portability

Disadvantages of layered architectures

- Data often pass through many layers, thus reducing performance. The workaround of adopting an open layered architecture increases coupling, while decreasing simplicity and information hiding
- Multi-layered programs can be hard to debug as operations tend to be implemented through calls across layers
- Getting the layers right may be difficult. Too few layers may result in low individual cohesion, may be too big and complicated, and may discourage their reuse. Too many layers may result in high interlayer coupling and in degraded performance

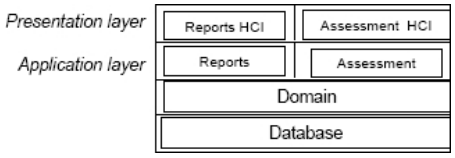
Guidelines for developing layered architectures

1. Define **criteria** to be used to group application into layers
   (Level of abstraction from hardware is common strategy)
2. Determine the **number of layers**
3. **Name** layers and **assign functionality** to them
   (top layer should implement main functions as perceived by user)
4. Specify the **services** for each layer
   (lower layers should have small number of services used by larger number of services in higher layers)
5. **Refine** the layering (by iterating through steps 1-4)
6. Specify **interfaces** for each layer
7. Specify **structure** of each layer (NB. may involve *partitioning*)
8. Specify **communication** between the layers
9. Reduce **coupling** between layers

13.4 The notion of partitioning

Partitioning

- Due to **complexity**, some layers within a layered architecture may have to be further decomposed (*e.g.*, into packages)
- Sub-systems within a layer should have clearly defined boundaries and well specified interfaces to provide high levels of **information hiding**
- Each partition includes loosely coupled sub-systems, each delivering a single service or a coherent set of services



13.5 Architectural pattern templates

- **Name:** states the pattern name
- **Application:** describes the pattern objectives
- **Form:** specifies the architectural outcome of the pattern once applied to a system (or part thereof)
- **Consequences:** explains the implications of the pattern application for the resulting architectural structure in terms of design principles

Example: A layered architecture pattern

Name: Layered

Application: Structure a program into an array of cohesive modules with well-defined interfaces to realise levels of abstraction, increase changeability, and increase reusability.

Form: The program is partitioned into cohesive modules with well-defined interfaces arranged in a chain or sequence from highest to lowest: the layers. Each layer is allowed to use only the layer immediately below it (closed layered architecture), or all the layers below it (open layered architecture).

Consequences: Layers should be cohesive, hide information, be simple, and be coupled only to the layer or layers beneath them. All these characteristics make layers easy to alter or replace, improving changeability. The simplicity of a layered architecture increases reliability and maintainability. Layers are likely to form reusable components.

Layered programs may be hard to debug; program behaviour must often be realised with communications across several layers, which may cause performance problems and be more work to program. It may be hard to form a good collection of layers.

13.6 Layered architectures in width and in depth

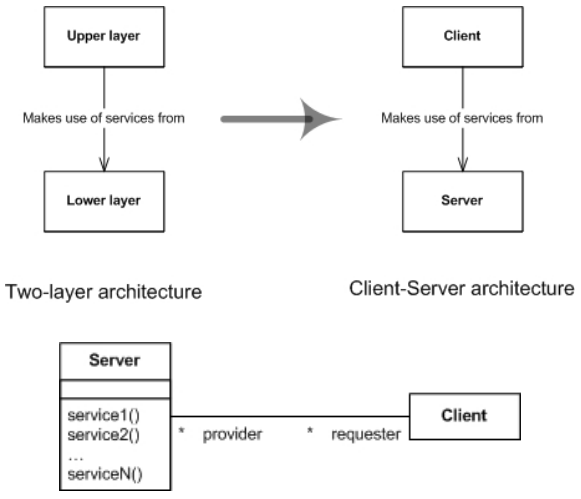
13.6.1 The client-server architecture and its extensions

Two-layer (client-server) architectures

- **Q:** what is a two-layer architecture?

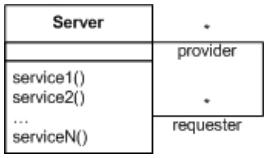
  A: it is the simplest layered architecture. For historic reasons it is known as the **client-server** architecture
- A **server** instance provides services to **client** instances, which in turn interact with users
- A service request is usually addressed via a remote procedure call or via a common object broker (*e.g.*, CORBA, Java RMI, HTTP)
- Control flow in clients and servers is independent except for synchronisation to manage requests or to receive results
- Client-server systems are not restricted to a single server: on the WWW, a single client can easily access data from 1,000s of different servers (even from what is apparently a single URL)

From two-layer to client-server



From client-server to peer-to-peer

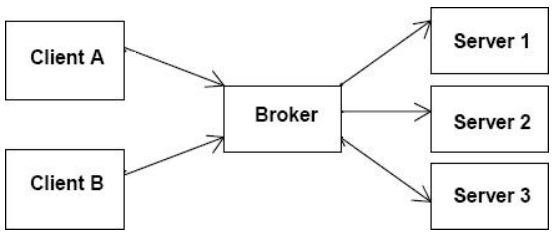
- A peer-to-peer architecture is a generalisation of the client-server one where each subsystem can act both as a client and as a server
- For this reason, peer-to-peer is no longer a layered architecture (use of services goes in both directions)
- Peer-to-peer architectures carry with them the possibility of deadlocks and complicate the control flow, *increasing* coupling and *decreasing* maintainability, changeability, program robustness, and fault tolerance



13.7 The client-broker-server architecture and its extensions

The client-broker-server architecture

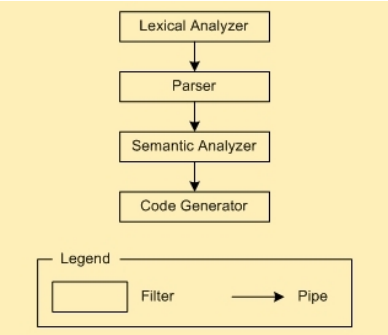
- A **Broker** increases the flexibility of a (distributed) system by *decoupling* client and server
- A broker may offer the services of many **servers**, identifying relevant server to which a request should be forwarded (typical of WWW)
- **Clients** does not need to know location of server(s)
- Client requests are sent to broker, which forwards them to server(s)



13.8 The pipe-and-filter architecture

The pipe-and-filter architectural pattern

- A filter is a component that transform an input data stream into an output data stream
- A pipe is the logical medium over which the data stream flows
- The pipe-and-filter architectural pattern is a dynamic model where software components are filters connected by pipes



Advantages of pipe-and-filter architectures

- Filters can be modified or replaced easily, making it simple to change the module to fix a problem or modify its behaviour
- Filters can be rearranged with little effort, making it easy to develop several programs that do similar tasks
- Filters are highly reusable
- Concurrency is supported and is relatively easy to implement, provided synchronising pipes are available
- This architectural pattern is primarily used in the UNIX shell

Disadvantages of pipe-and-filter architectures

- Filters communicate only through pipes, making it difficult for them to coordinate their activities
- Filters usually consume and produce very simple data streams, such as streams of characters, which means that they may have to spend a lot of effort converting input into a usable form and then converting results back to a simple format for output
- Error handling is difficult; error information can only be passed along a pipe. The difficulty of error detection and recovery makes this style inappropriate when reliability and safety are important
- Gains from concurrency may be illusory. Pipes may not synchronise filters effectively, and some filters may need to wait for all their input before doing any output

13.8.1 Pipe-and-filter architecture: pattern specification

Name: Pipe-and-Filter

Application: Model the collaboration of components in programs that transform input streams to output streams.

Form: This style does not specify a static form. Its dynamic form is a collection of data stream transformers (filters) connected to one another by data stream conduits (pipes). Filters are independent components that are simple to construct and highly reusable. They may run in parallel, synchronised by pipes.

Consequences: Pipe-and-Filter-style systems are easy to construct, have reusable components, are easy to change, and may have good performance thanks to concurrent filters.

However, filters are hard to coordinate and have difficulty handling errors cooperatively (degrading reliability and safety). In addition, performance gains may not materialise if the filters are not well synchronised or do not lend themselves to concurrent processing.

13.9 Preparing for the topic ahead

Next lecture. . .

Further catalogue of architectural patterns

More specifically, we will focus on:

- Shared-data architectural patterns
  (including AI-centric blackboard architectures)
- Event-driven architectural patterns
- Model-View-Controller architectural patterns