Software Engineering Notes

Dr Greg Butler

Outline

- Software Architecture
- Layered Architecture
- Model-View-Control

Software Architecture

- A software architecture is a description of the subsystems and components of a software system and the relationships between them.
- Subsystems and components are typically specified in different views to show the relevant functional and non-functional properties of a software system.
- The software system is an artifact. It is the result of the software design activity.

Component

- A component is an encapsulated part of a software system. A component has an interface.
- Components serve as the building blocks for the structure of a system.
- At a programming-language level, components may be represented as modules, classes, objects or a set of related functions.

Subsystems

- A subsystem is a set of collaborating components performing a given task. A subsystem is considered a separate entity within a software architecture.

- It performs its designated task by interacting with other subsystems and components…

Software Architecture

Formal definition IEEE 1471-2000

- Software architecture is the **fundamental organization** of a system, embodied in its **components**, their **relationships** to each other and the environment, and the **principles** governing its design and evolution

Software Architecture

Software architecture encompasses the set of ***significant decisions*** about the ***organization*** of a software system

- Selection of the structural elements and their interfaces by which a system is composed
- Behavior as specified in collaborations among those elements
- Composition of these structural and behavioral elements into larger subsystems
- Architectural style that guides this organization

Need for Software Architectures

Scale
Process
Cost
Schedule
Skills and development teams
Materials and technologies
Stakeholders
Risks

Why is Software Architecture Important

Representations of software architecture are an enabler for communication between all parties (stakeholders) interested in the development of a computer-based system.

The architecture highlights early design decisions that will have a profound impact on all software engineering work that follows and, as important, on the ultimate success of the system as an operational entity.

Architecture “constitutes a relatively small, intellectually graspable model of how the system is structured and how its components work together” -- L. Bass

UP: Software Architecture is Early

Architecture represents the set of earliest design decisions

- Hardest to change
- Most critical to get right

Architecture is the first design artifact where a system’s quality attributes are addressed

UP: Software Architecture is Early

UP: Inception, Elaboration, Construction, Transition

UP Inception is feasibility phase

- Develop architecture addressing all high risks

UP Elaboration

- Full description of Architecture
- Working prototype of architecture

Software Architecture Drives

Architecture serves as the ***blueprint*** for the system but also the project:

- Team structure
- Documentation organization
- Work breakdown structure
- Scheduling, planning, budgeting
- Unit testing, integration

Architecture establishes the communication and coordination mechanisms among components

Software Architecture versus Design

Architecture: where non-functional decisions are cast, and functional requirements are partitioned

Design: where functional requirements are accomplished

System Non-Functional Quality Attributes

End User's view

Performance
Availability
Usability
Security

Developer's view

Maintainability
Portability
Reusability
Testability

Business Community view

Time To Market
Cost and Benefits
Projected life time
Targeted Market
Integration with Legacy System
Roll back Schedule

Issues Addressed by an Architectural Design

- Gross decomposition of a system into interacting components
  - **Typically hierarchical**
  - **Using rich abstractions for "glue"**
  - **Often using common design idioms/styles**
- Emergent system properties
  - **Performance, throughput, latencies**
  - **Reliability, security, fault tolerance, evolvability**
- Rationale
  - **Relates requirements and implementations**
- Envelope of allowed change
  - **"Load-bearing walls"**
  - **Design idioms and styles**

Modularization

The principal problem of software systems is complexity.

It is not hard to write small programs.

Decomposing the problem (modularization) is an effective tool against complexity.

The designer should form a clear mental model of how the application will work at a high level, then develop a decomposition to match the mental model.

Modularization

Cohesion within a module is the degree to which communication takes place among the module's elements.

Coupling is the degree to which modules depend directly on other modules.

Effective modularization is accomplished by maximizing cohesion and minimizing coupling.

Good Properties of an Architecture

- Good architecture (like much good design):
  - **Result of a consistent set of principles and techniques, applied consistently through all phases of a project**
  - **Resilient in the face of (inevitable) changes**
  - **Source of guidance throughout the product lifetime**
  - **Reuse of established engineering knowledge**

Developing a Software Architecture

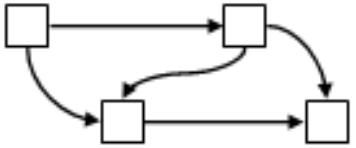
Develop a mental model of the application.
As if it were a small application, e.g., personal finance application ...
“works by receiving money or paying out money, in any order, controlled through a user interface”.

Decompose into the required components.
Look for high cohesion & low coupling, e.g., personal finance application ...
decomposes into Assets, Sources, Suppliers, & Interface.

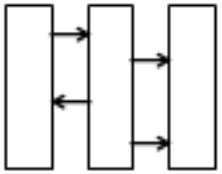
Repeat this process for the components.

A Classification of Software Architectures

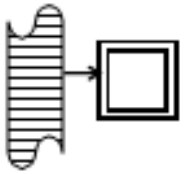
- *Data Flow*
  - Data flowing between functional elements



- *Independent Components*
  - -- executing in parallel, occasionally communicating

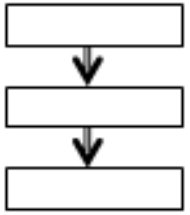


- *Virtual Machines*
  - Interpreter + program in special-purpose language



- *Repositories*
  - Primarily built around large data collection

- *Layered*
  - Subsystems, each depending one-way on another subsystem



Common Software Architectures

Layered architecture

Eg, client-server, 3-tier

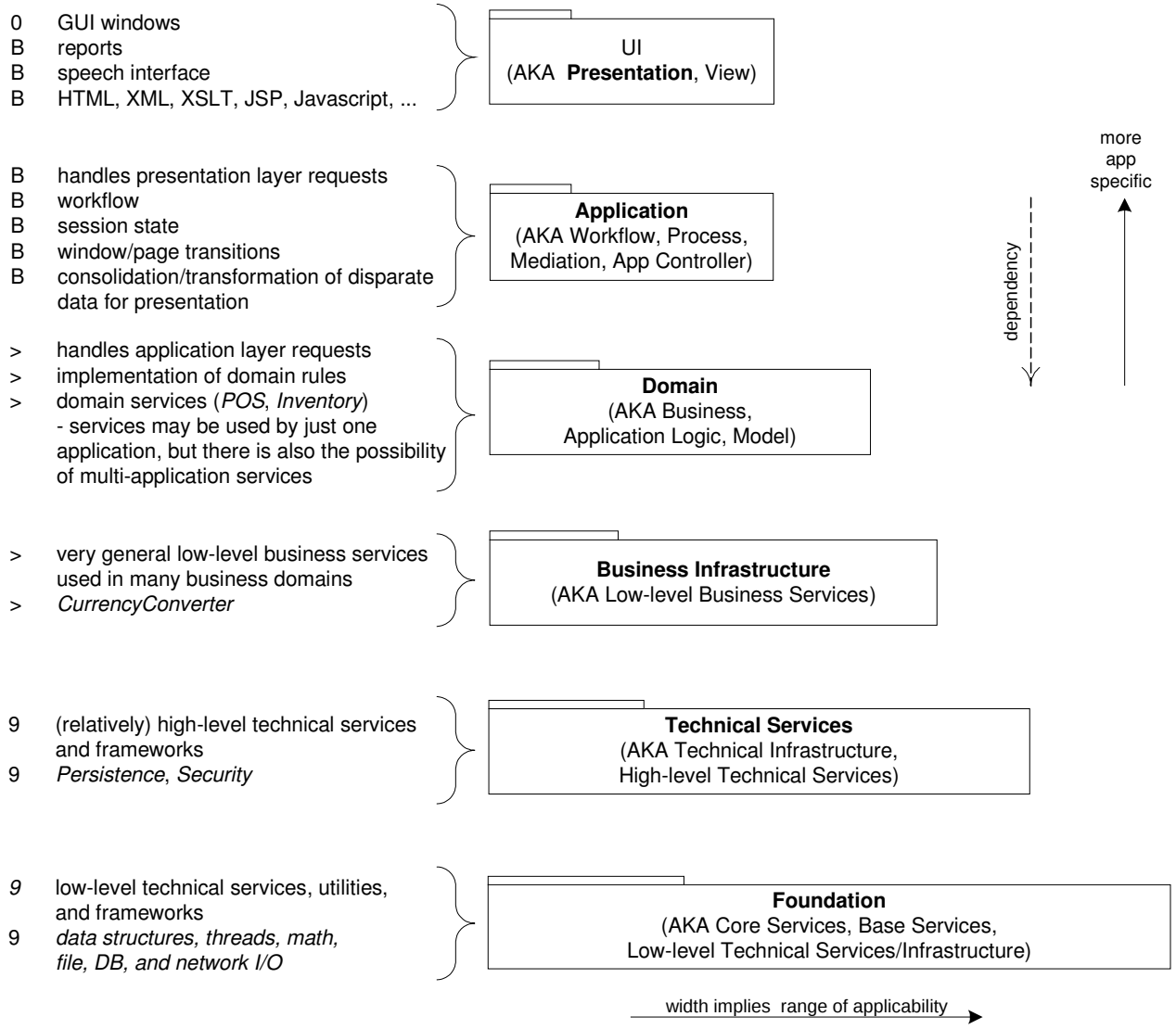
Model-View-Control architecture

Broker

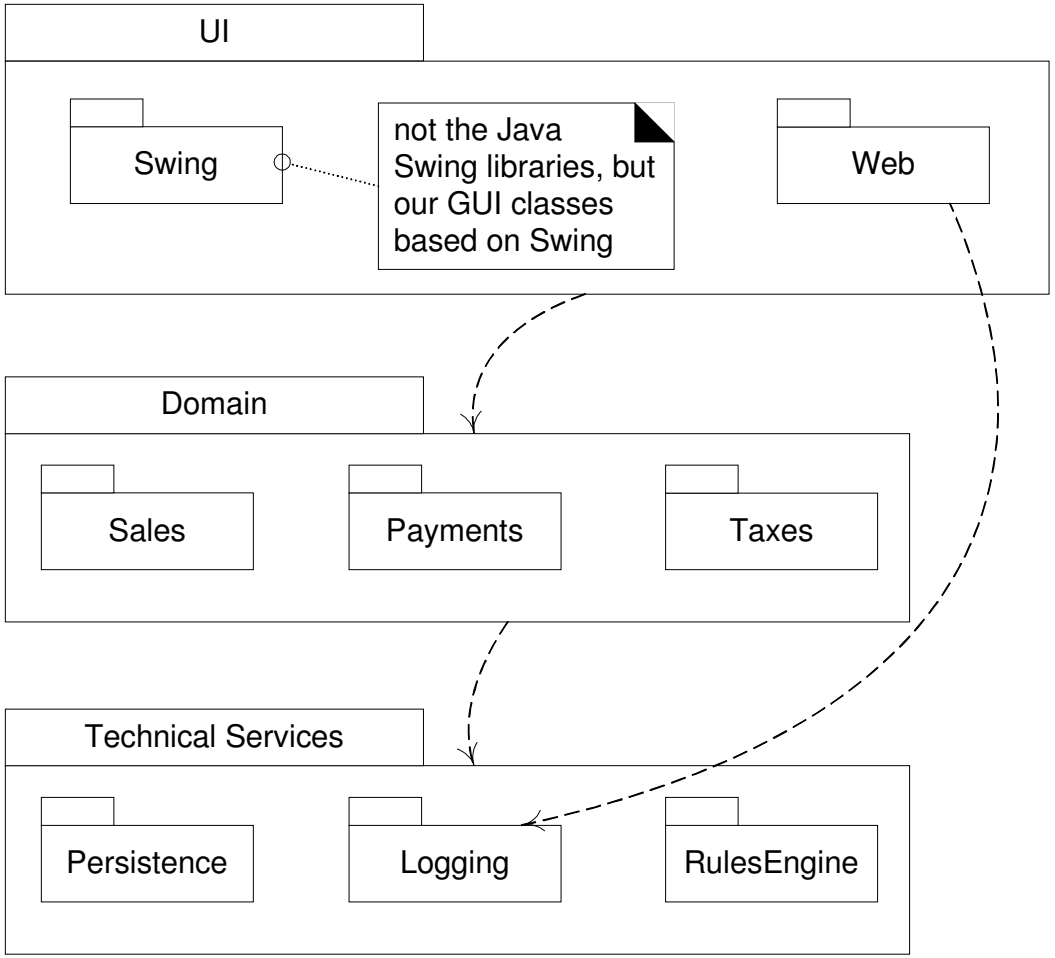
Interpreter

Pipeline

Typical Software Architecture Layers

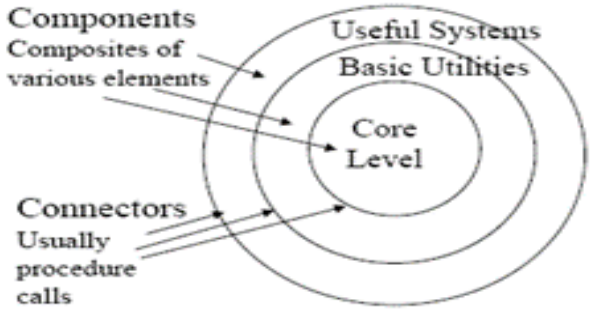


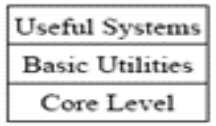
Typical Software Architecture Layers (Simplified)



Layered Systems

- "A layered system is organised hierarchically, each layer providing service to the layer above it and serving as a client to the layer below." (Garlan and Shaw)
- Each layer collects services at a particular level of abstraction.
- In a pure layered system: Layers are hidden to all except adjacent layers.



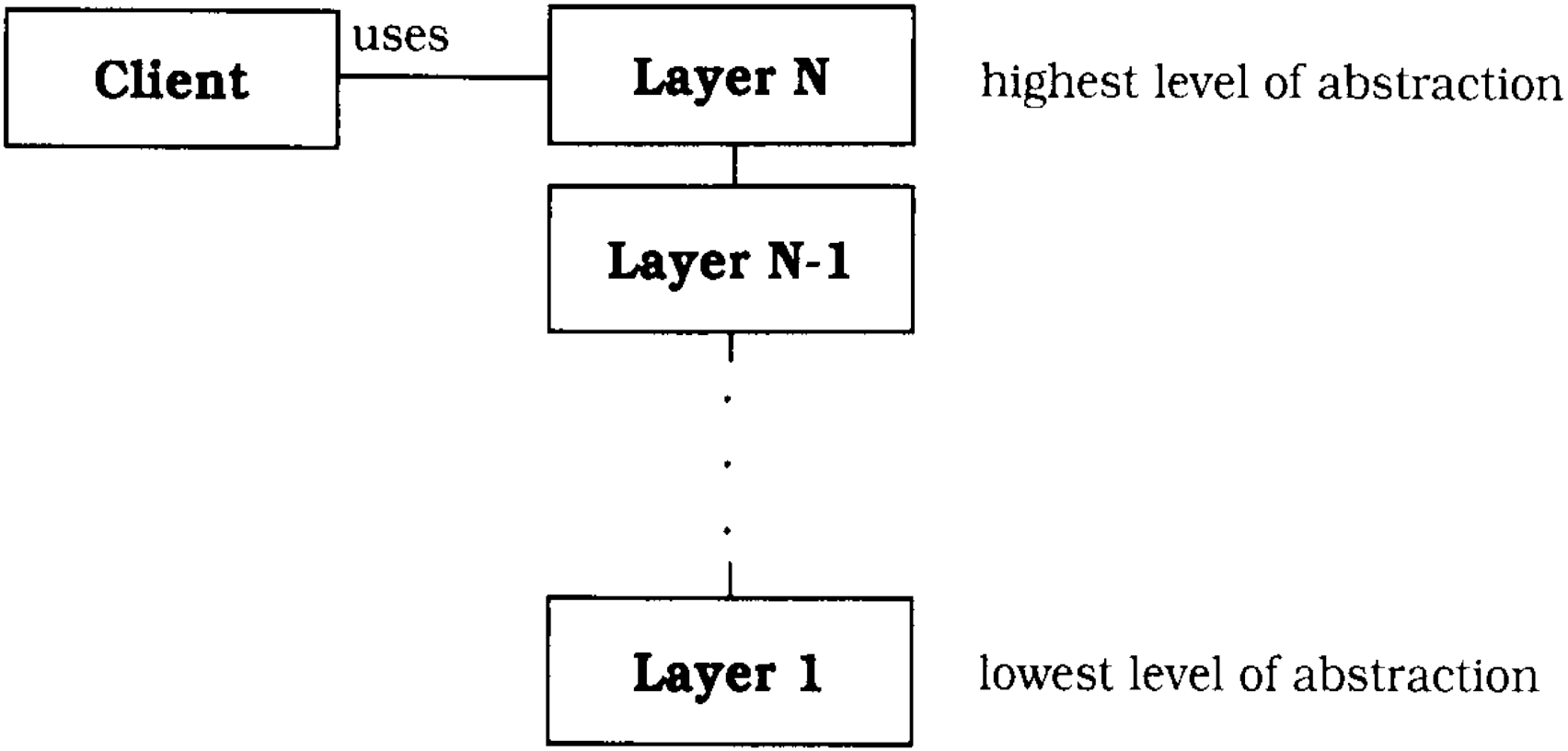


- "Onion Skin model"…
- corresponds to a stack of layers.

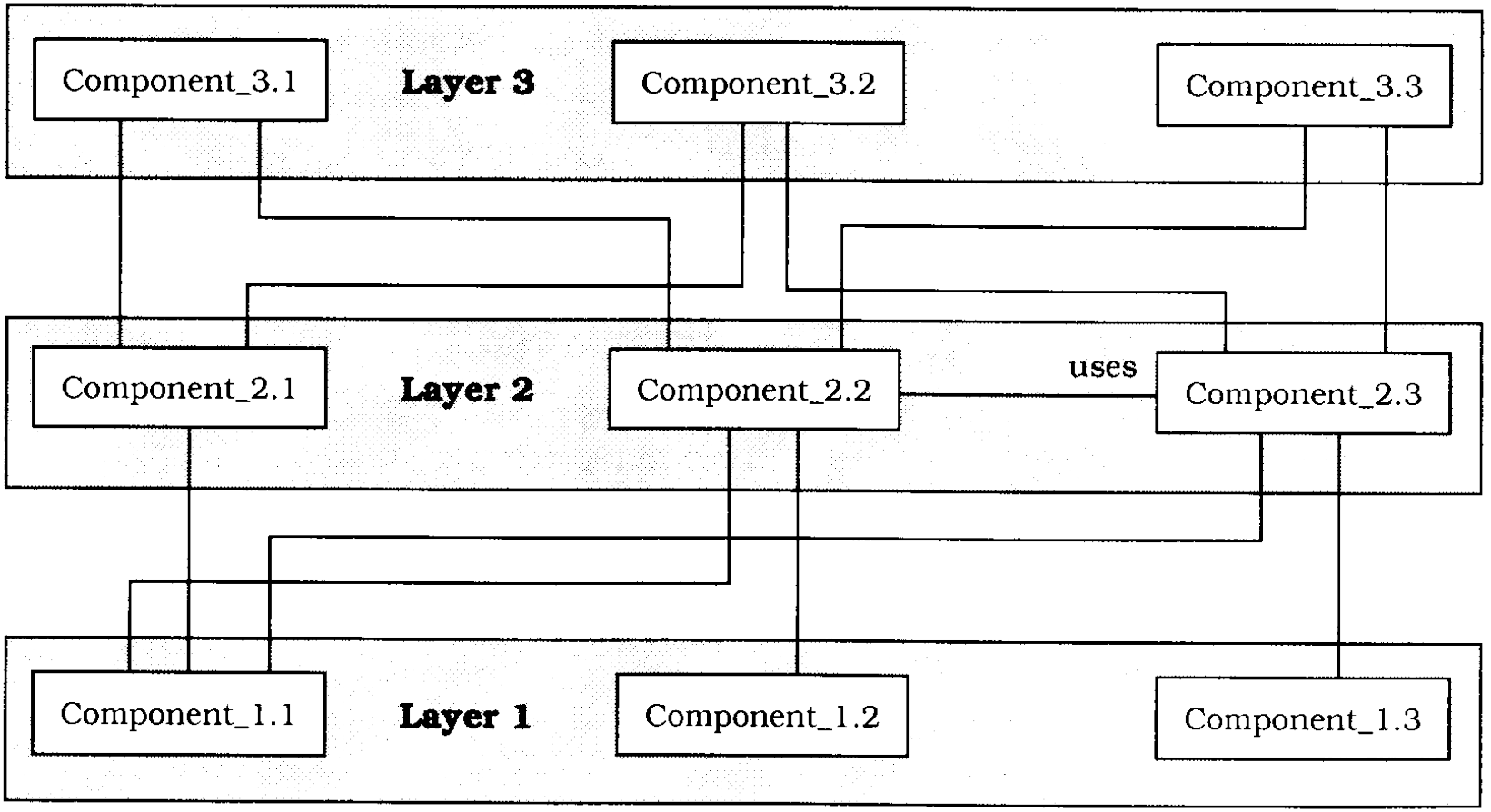
Layers: Structure

Class	***Collaborator***
Layer J	• Layer J - 1
Responsibility	
• Provides services used by Layer J+1. • Delegates subtasks to Layer J - 1.	

Layers: Structure



Layers and Components



Layers: Known Uses

- virtual machines: JVM and binary code format
- API: layer that encapsulates lower layers
- information systems
  - presentation, application logic, domain layer, database
- operating systems (relaxed for: kernel and IO and hardware)
  - system services,
  - resource management (object manager, security monitor, process manager, I/O manager, VM manager, LPC),
  - kernel (exception handling, interrupt, multiprocessor synchronization, threads),
  - HAL (Hardware Abstraction Layer, *e.g.* in Linux)

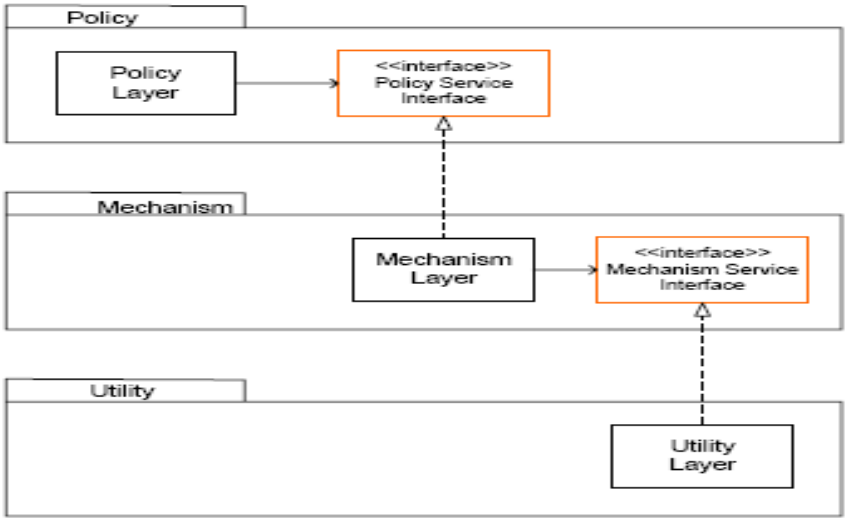
Layered Systems

- Applicability
  - **A large system that is characterised by a mix of high and low level issues, where high level issues depend on lower level ones.**
- Components
  - **Group of subtasks which implement a 'virtual machine' at some layer in the hierarchy**
- Connectors
  - **Protocols / interface that define how the layers will interact**
- Invariants
  - **Limit layer (component) interactions to adjacent layers (in practice this may be relaxed for efficiency reasons)**
- Typical variant relaxing the pure style
  - **A layer may access services of all layers below it**
- Common Examples
  - **Communication protocols: each level supports communication at a level of abstraction, lower levels provide lower levels of communication, the lowest level being hardware communications.**

Layered System Examples

- Example 1: ISO defined the OSI 7-layer architectural model with layers: Application, Presentation, …, Data, Physical.
  - **Protocol specifies behaviour at each level of abstraction (layer).**
  - **Each layer deals with specific level of communication and uses services of the next lower level.**
- Example 2: TCP/IP is the basic communications protocol used on the internet. POSA book describes 4 layers: ftp, tcp, ip, Ethernet. The same layers in a network communicate 'virtually'.
- Example 3: Operating systems e.g. hardware layer, …, kernel, resource management, … user level "Onion Skin model".
- ...

Sample Implementation

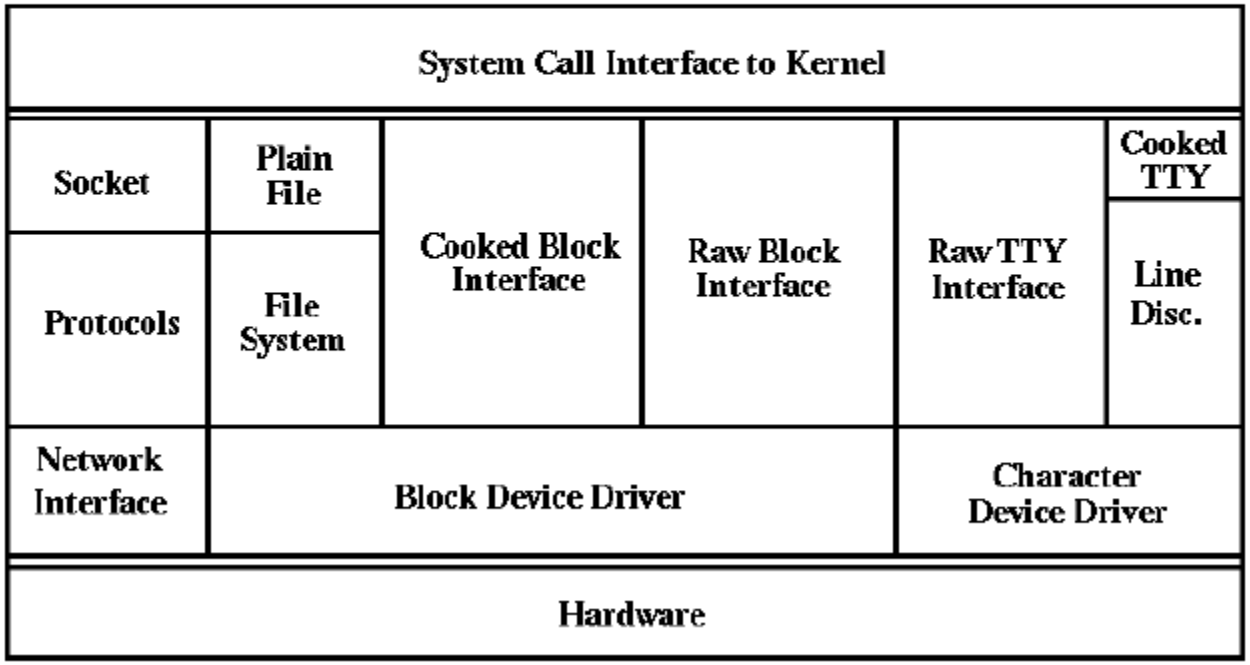


Layered Systems

- Strengths
  - **Increasing levels of abstraction as we move up through layers – partitions complex problems**
  - **Maintenance - in theory, a layer only interacts with layers above and below. Change has minimum effect.**
  - **Reuse - different implementations of the same level can be interchanged**
  - **Standardisation based on layers e.g. OSI**
- Weaknesses
  - **Not all systems are easily structured in layers (e.g., mobile robotics)**
  - **Performance - communicating down through layers and back up, hence bypassing may occur for efficiency reasons**
- Similar strengths to data abstraction / OO but with multiple levels of abstraction (e.g. well-defined interfaces, implementation hidden).
- Similar to pipelines, e.g., communication with at most one component at either side, but with richer form of communication.
- A layer can be viewed as aka "virtual machine" providing a standardized interface to the one above it

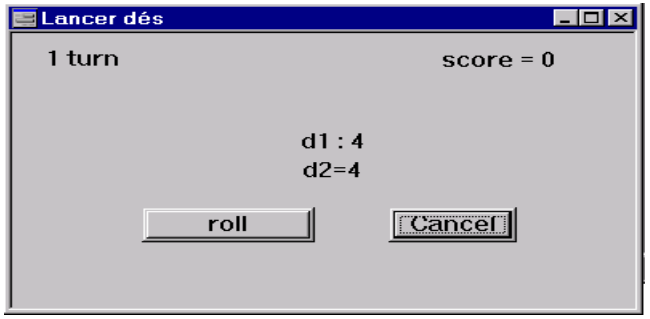
Layered Architectures

The Unix Layered Architecture

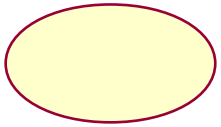


Applying Layers Architecture

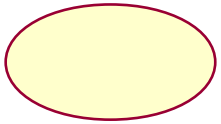
UI



Core

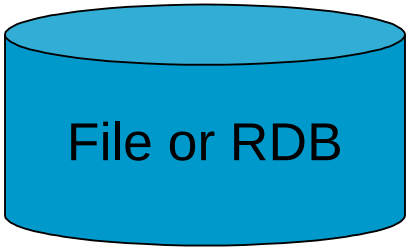


Play



View High Score

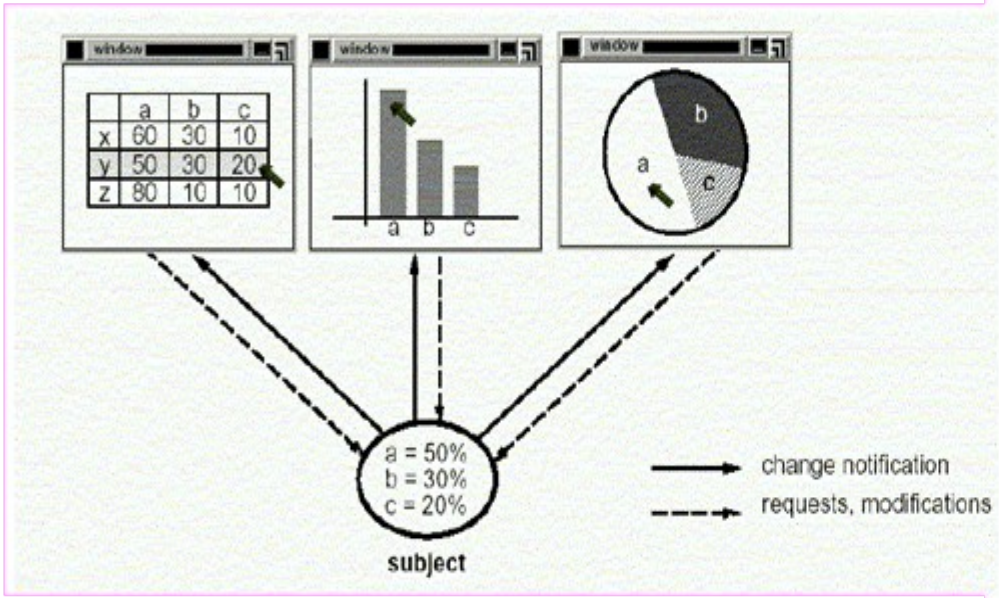
Persistence

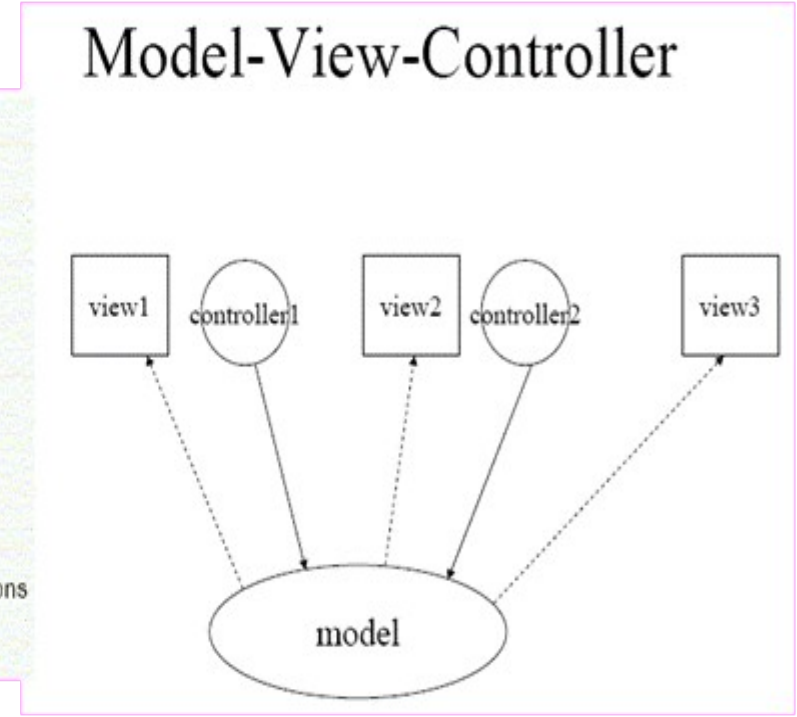


Model-View-Controller

- A decomposition of an interactive system into three components:
  - **A model containing the core functionality and data,**
  - **One or more views displaying information to the user, and**
  - **One or more controllers that handle user input.**
- A change-propagation mechanism (i.e., observer) ensures consistency between user interface and model, e.g.,
  - **If the user changes the model through the controller of one view, the other views will be updated automatically**
- Sometimes the need for the controller to operate in the context of a given view may mandate combining the view and the controller into one component
- The division into the MVC components improves maintainability

MVC


window
window
window
a b c
x 60 30 10
y 50 30 20
z 80 10 10
a b c
b
a
c
a = 50%
b = 30%
c = 20%
subject
change notification
requests, modifications


Model-View-Controller
view1
controller1
view2
controller2
view3
model

MVC

model, view, and controller communicate regularly

for example:

model notifies the view of state changes

view registers controller to receive user interface events (e.g., "onClick()"

controller updates the model when input is received

MVC Responsibilities

model responsibilities

store data in properties

implement application methods

methods to register/unregister views

notify views of state changes

implement application logic

view responsibilities

create interface

update interface when model changes

forward input to controller

controller responsibilities

- translate user input into changes in the model
- if change is purely cosmetic, update view

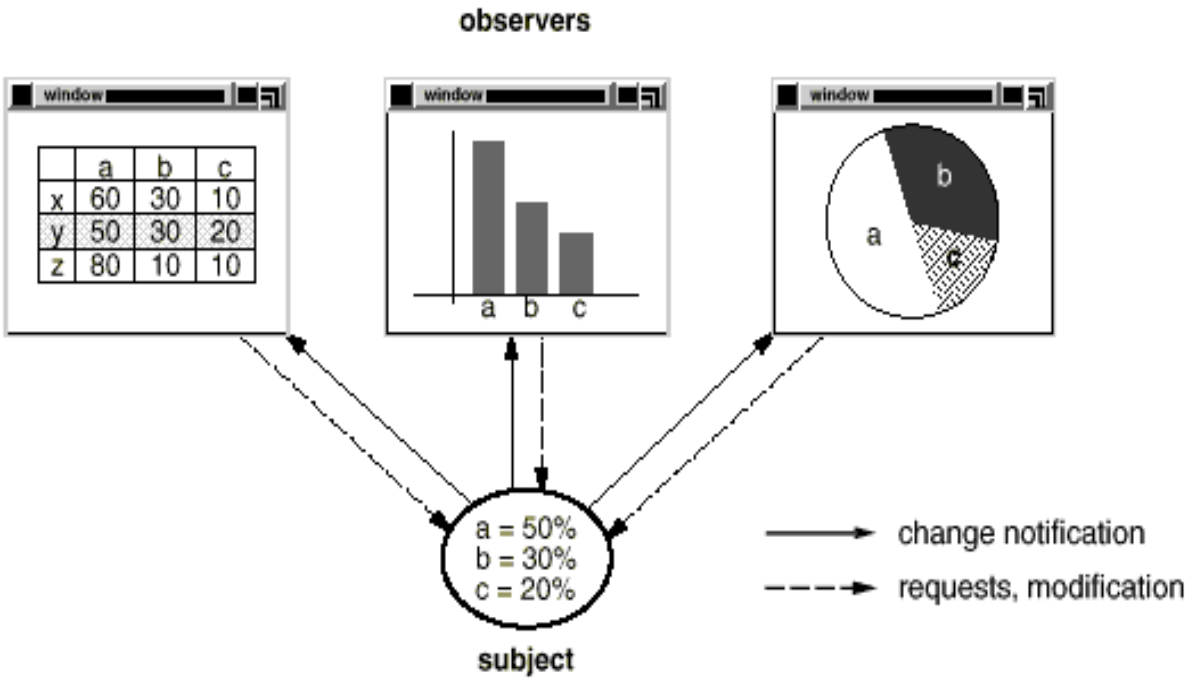
Digression: MVC

MVC dates back to Smalltalk, almost 30 years ago.

...in fact MVC actually exhibits a mix of **three** GoF design patterns: *Strategy*, *Observer* and *Composite*.

Observer

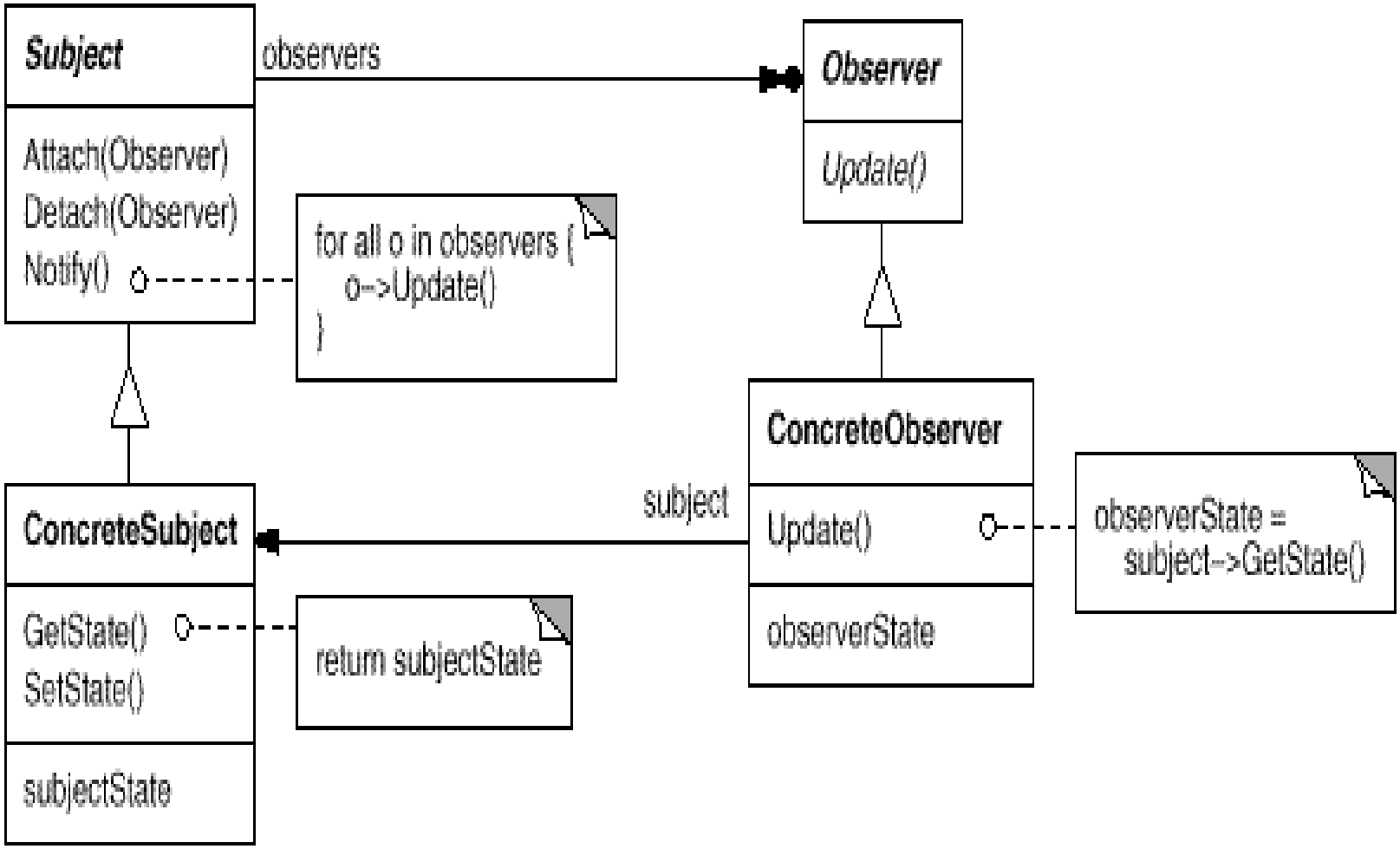
When an object's value is updated, observers watching it are notified that the change has occurred.



Observer: Applicability

- A change to one object requires changing an unknown set of other objects.
- Object should be able to notify others that may not be known from the beginning.
- Metaphor = newspaper or magazine subscription:
- A publisher goes into business and starts printing a periodical.
- You subscribe.
- Every time a new edition is printed, you receive a copy in the mail.
- You unsubscribe when you want to stop receiving new copies.
- New copies stop being delivered to you — but other people can still subscribe and receive their own copies.

Observer: Formal Structure



Observer: Pros and Cons

- promotes loose coupling between subject and observer
- the subject only knows that an observer implements an interface
- new observers can be added or removed at any time
- no need to modify the subject to add a new type of observer
- subject and objects can be reused independently of each other

- support for broadcast communication

- may become expensive if many observers, especially for small changes to a large data area (i.e., broadcasting redundant information)

Observer in Java

- In Java, Observer will usually be an interface rather than an abstract base class (no surprise, right? :-).
- In fact, the Java library already includes an Observer interface and an Observable class (in the java.util package).
- ...but the Observable class has some drawbacks:
- it **is** a class, rather than an interface, and it doesn't even implement an interface — so it can't be used by a class that already inherits from something else (no multiple inheritance in Java)
- ...and some key methods in it are protected, so it can't be used unless you **can** extend it; so much for favouring composition over inheritance :-/

For these reasons, even in Java it's often preferable to write your own Subject interface and class(es).