COMP 6471
Software Design Methodologies

Fall 2011

Dr Greg Butler

http://www.cs.concordia.ca/~gregb/home/comp6471-fall2011.html

Week 7 Outline

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

Course Objectives

- Software architecture
  - Its role in the software process
  - Its role in software design

- Software architecture
  - Importance
  - describing/modeling software architecture
  - Common styles of software architecture

- Layers
  - Especially in web applications

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…

Architectural Patterns

An architectural pattern express a fundamental structural organization schema for software systems.

It provides a set of predefined subsystems, their responsibilities, and includes rules and guidelines for organizing the relationships between them.

Design patterns

A design pattern provides a scheme for refining the subsystems or components of a software system, or the relationships between them.

It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context.

Idioms

An Idiom is a low-level pattern specific to a programming language.

An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.

Framework

A framework is a partially complete software (sub-)system that is intended to be instantiated.

It defines the architecture for a family of (sub-)systems and provides the basic building blocks to create them. It also defines the places where adaptations for specific functionality should be made.

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

Software Architecture

- Perry and Wolf, 1992
    - A set of architectural (or design) <u>elements</u> that have a particular form
-
- Boehm et al., 1995
    - A software system architecture comprises
        - A collection of software and system <u>components, connections, and constraints</u>
        - A collection of <u>system stakeholders'</u> need statements
        - A <u>rationale</u> which demonstrates that the components, connections, and constraints define a system that, if implemented, would satisfy the collection of system stakeholders' need statements

Clements et al., 1997

- The software architecture of a program or computing system is the structure or structures of the system, which comprise <u>software components</u>, the externally visible properties of those components, and the relationships among them

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.

Architecture Development

- Unified Process:
  - **Focus on implementing the most valuable and critical use cases first**
  - **Produce an architectural description by taking those design elements that are needed to explain how the system realizes these use cases at a high level**

- Use past and proven experience by applying architectural styles and patterns

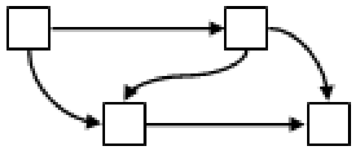
Architectural Styles and Patterns

- An *architectural style* defines a family of architectures constrained by
  - **Component/connector vocabulary, e.g.,**
    - **layers and calls between them**
  - **Topology, e.g.,**
    - **stack of layers**
  - **Semantic constraints, e.g.,**
    - **a layer may only talk to its adjacent layers**

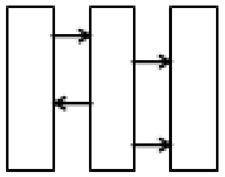
- For each architectural style, an *architectural pattern* can be defined
  - **It's basically the architectural style cast into the pattern form**
  - **The pattern form focuses on identifying a problem, context of a problem with its forces, and a solution with its consequences and tradeoffs; it also explicitly highlights the composition of patterns**

A Classification of Software Architectures

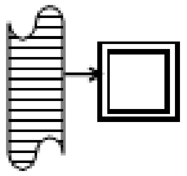
- ***Data Flow***
  - Data flowing between functional elements



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

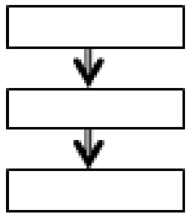


- ***Virtual Machines***
  - Interpreter + program in special-purpose language



- ***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

K GUI windows
B reports
B speech interface
B HTML, XML, XSLT, JSP, Javascript, ...

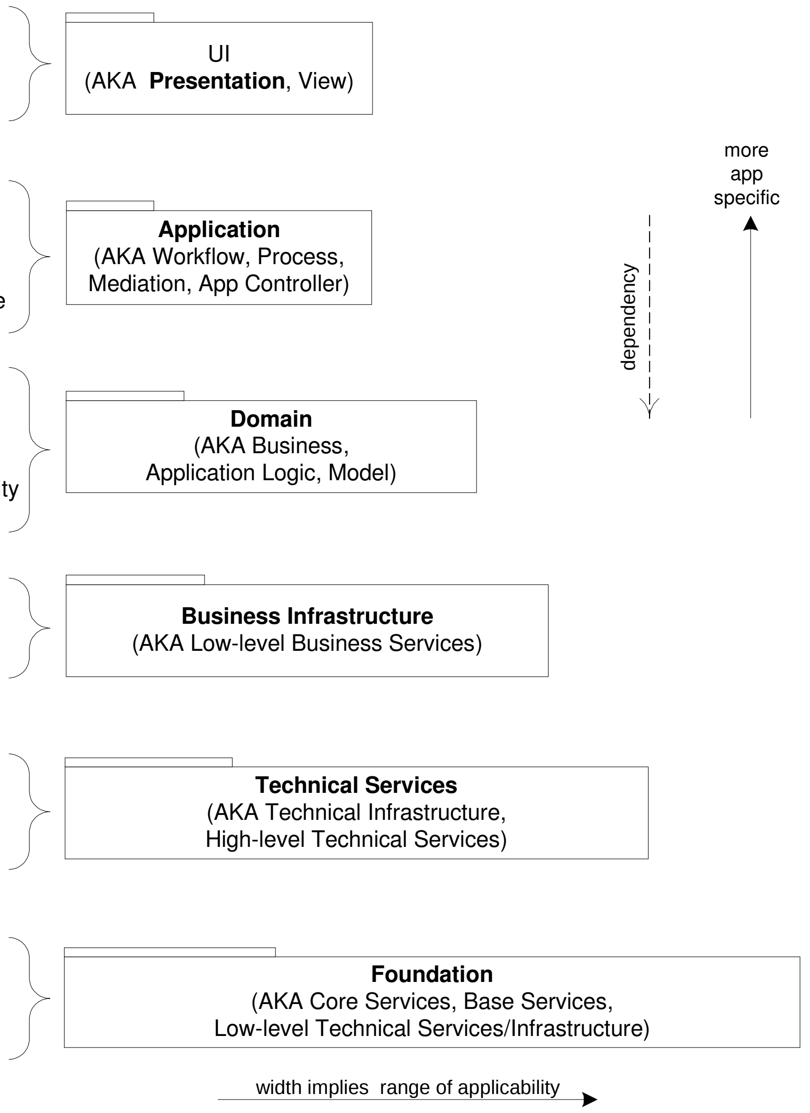
B handles presentation layer requests
B workflow
B session state
B window/page transitions
B consolidation/transformation of disparate data for presentation

> handles application layer requests
> implementation of domain rules
> domain services (*POS*, *Inventory*)
- services may be used by just one application, but there is also the possibility of multi-application services

> very general low-level business services used in many business domains
> *CurrencyConverter*

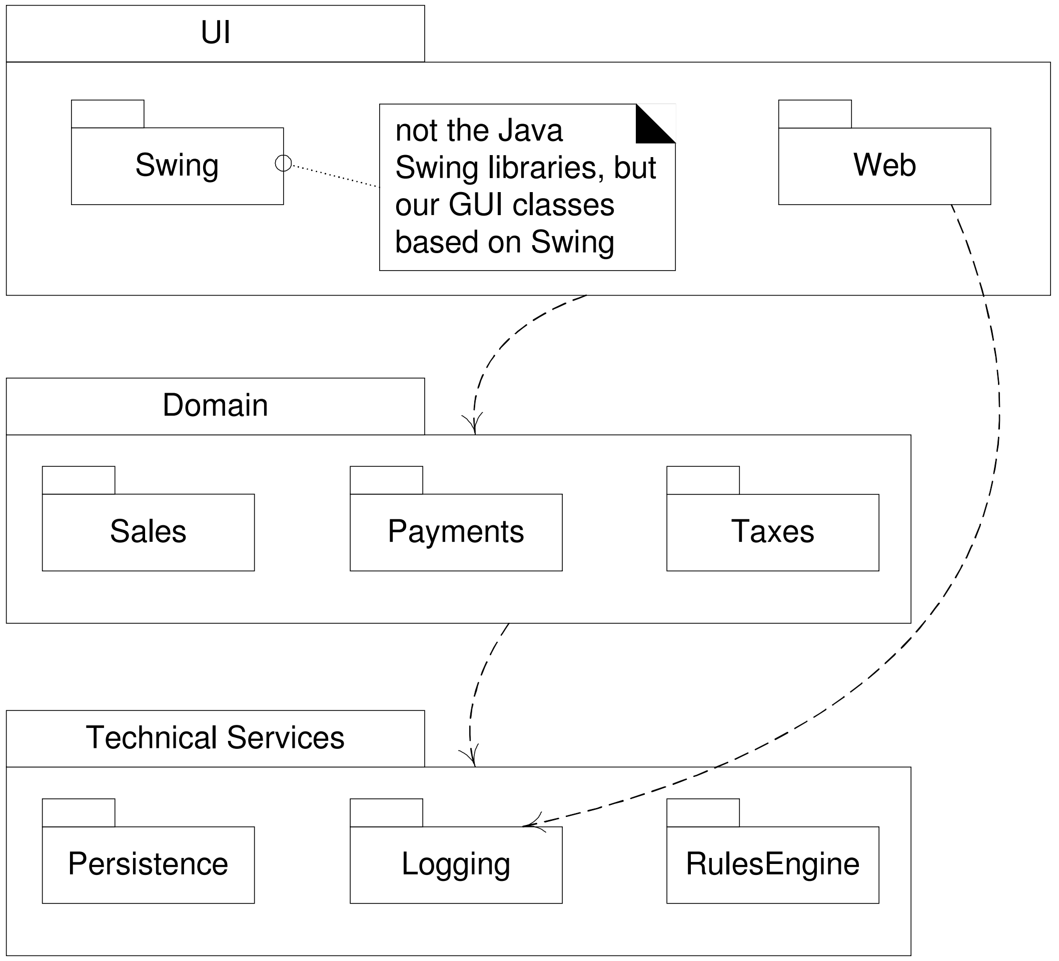
9 (relatively) high-level technical services and frameworks
9 *Persistence*, *Security*

9 low-level technical services, utilities, and frameworks
9 *data structures, threads, math, file, DB, and network I/O*



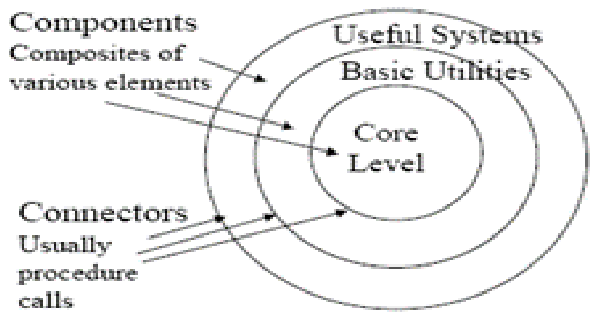
UI
(AKA **Presentation**, View)
Application
(AKA Workflow, Process, Mediation, App Controller)
Domain
(AKA Business, Application Logic, Model)
Business Infrastructure
(AKA Low-level Business Services)
Technical Services
(AKA Technical Infrastructure, High-level Technical Services)
Foundation
(AKA Core Services, Base Services, Low-level Technical Services/Infrastructure)
dependency
more app specific
width implies range of applicability

Typical Software Architecture Layers (Simplified)



Layered Systems

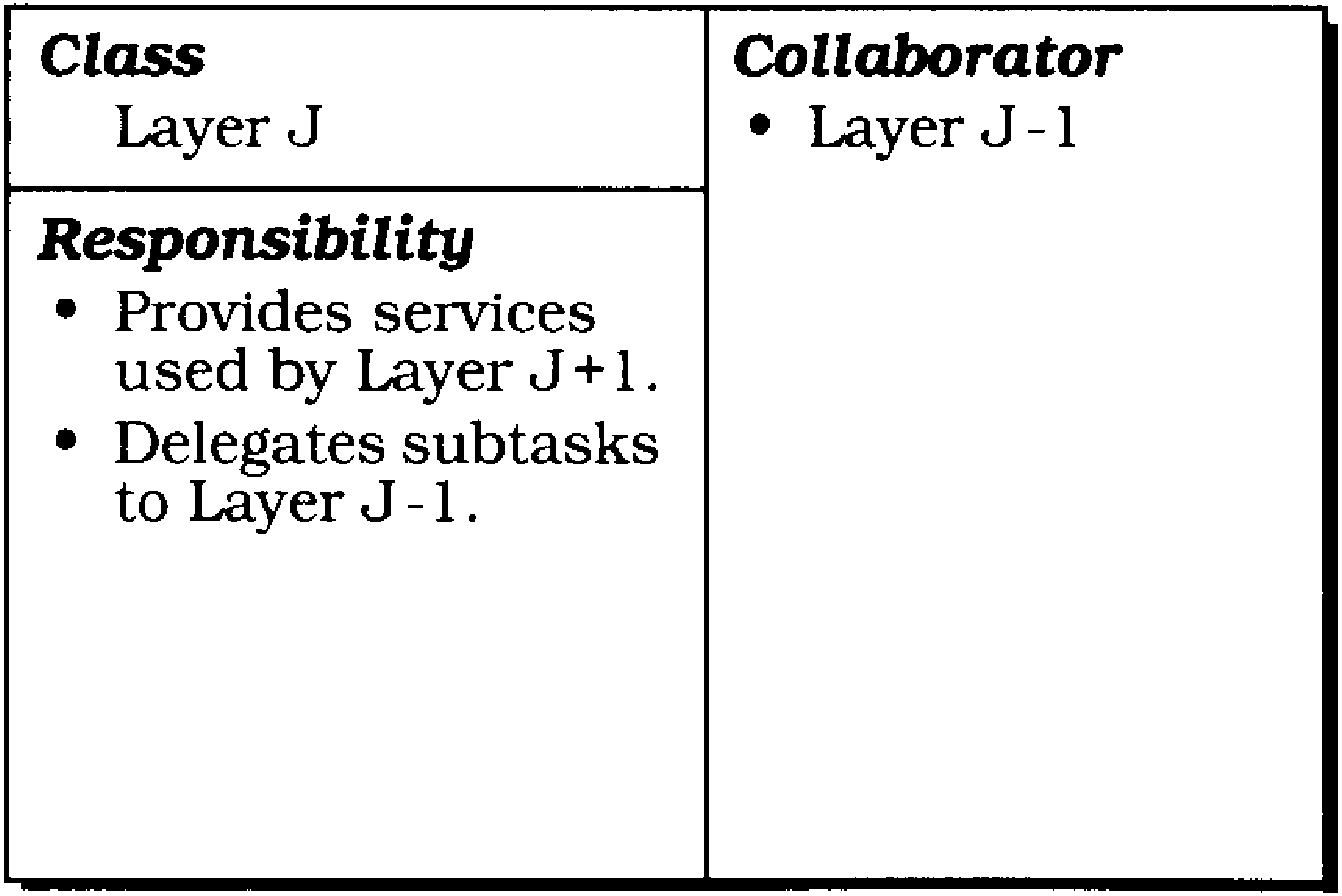
- "A layered system is organised hierarchically, each layer providing service to the layer above it and serving as a client to the layer below." (Garlan and Shaw)
- Each layer collects services at a particular level of abstraction.
- In a pure layered system: Layers are hidden to all except adjacent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



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

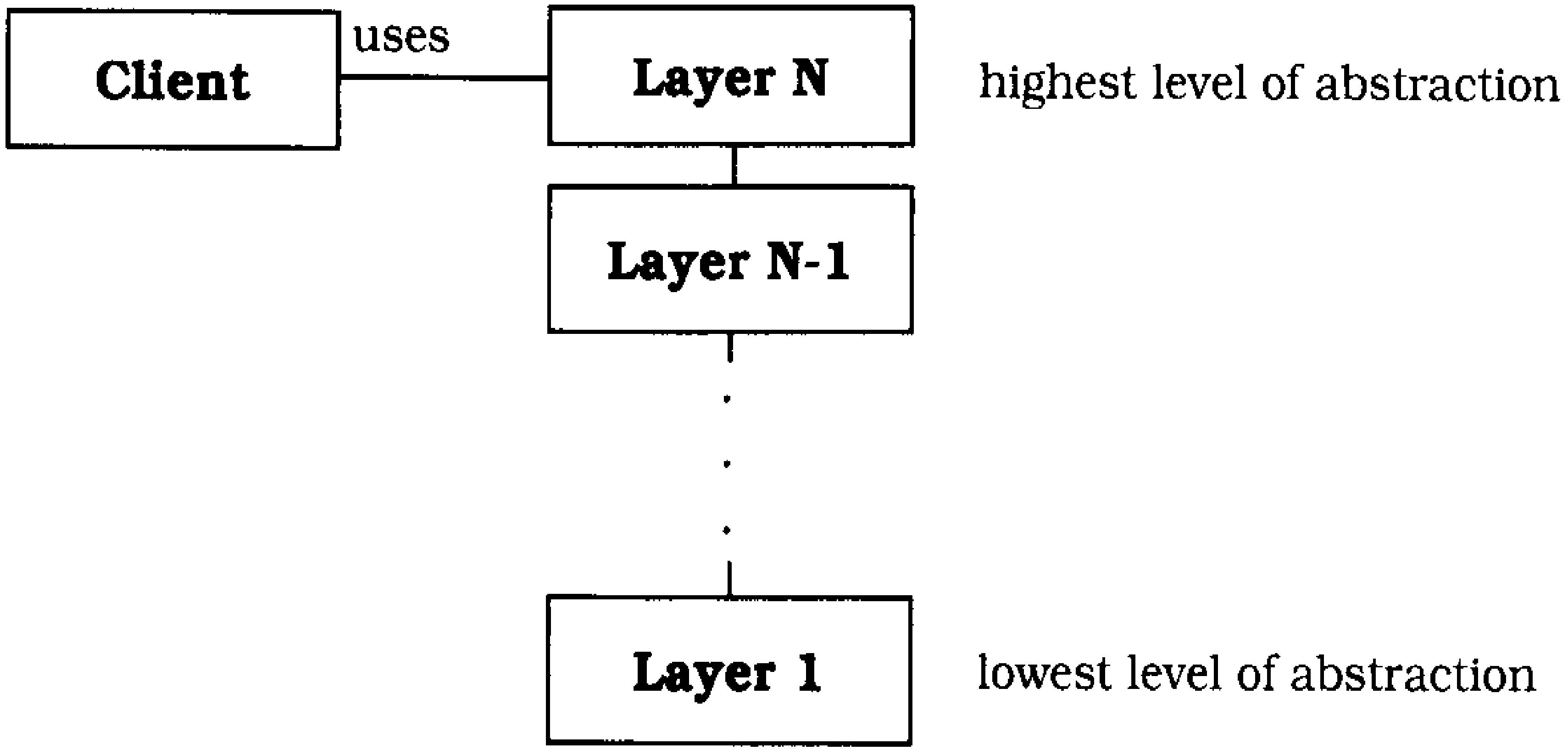
Useful Systems
Basic Utilities
Core Level

Layers: Structure

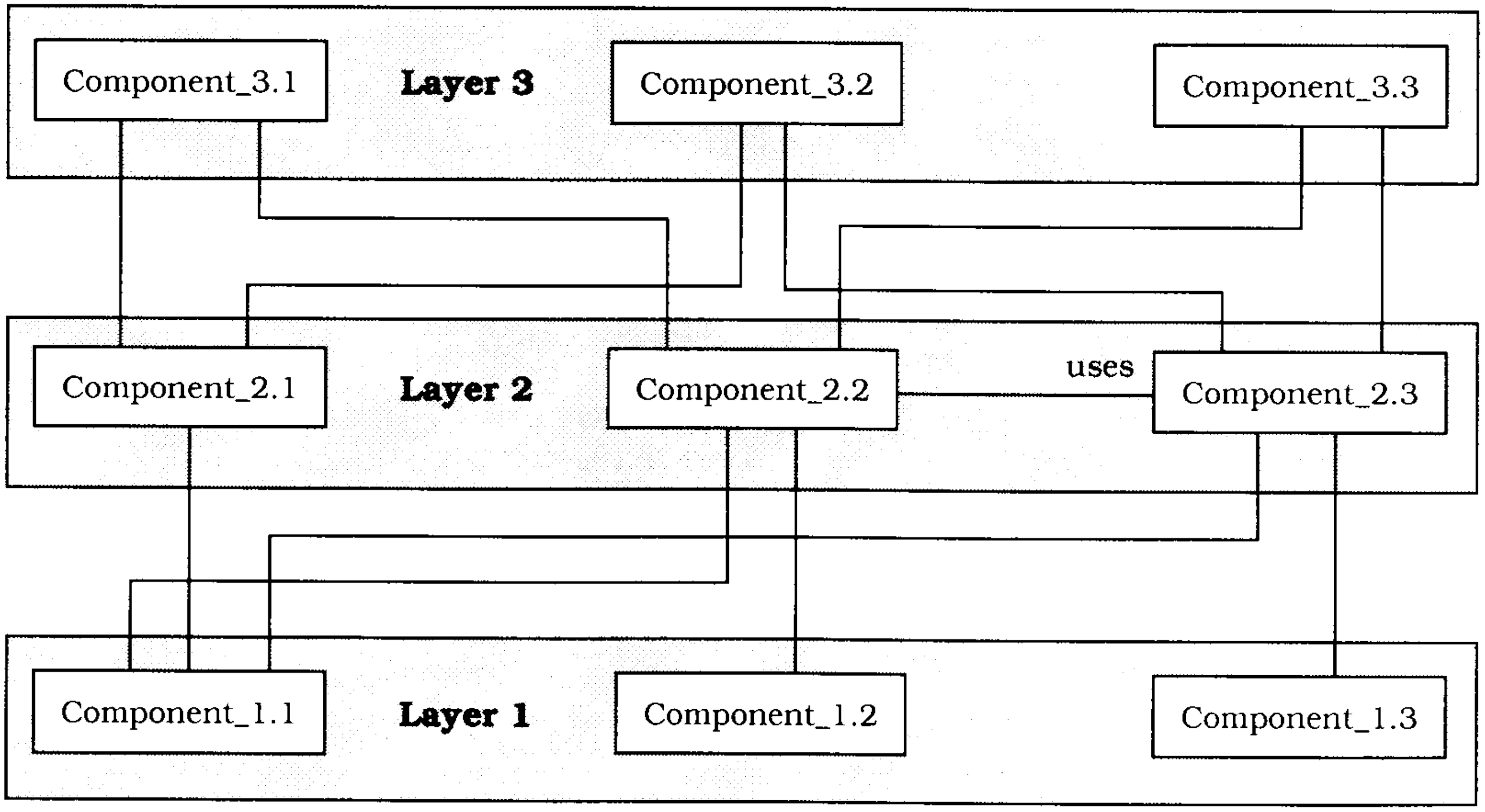


Class	***Collaborator***
Layer J	• Layer J - 1
Responsibility	
• Provides services used by Layer J + 1.	
• Delegates subtasks to Layer J - 1.	

Layers: Structure



Layers and Components



Layers: Variants

- relaxed layered system:
  - a layer j can use services from j-1, j-2…
- layering through inheritance:
  - lower layers are implemented as base classes
  - higher level can override lower level

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)

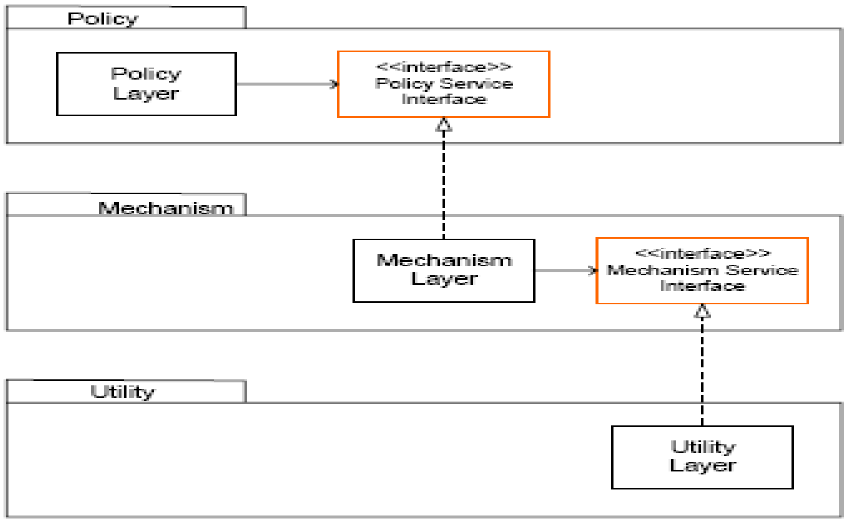
Layered Systems

- 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


Policy
Policy Layer
<<interface>> Policy Service Interface
Mechanism
Mechanism Layer
<<interface>> Mechanism Service Interface
Utility
Utility Layer

Layered Systems

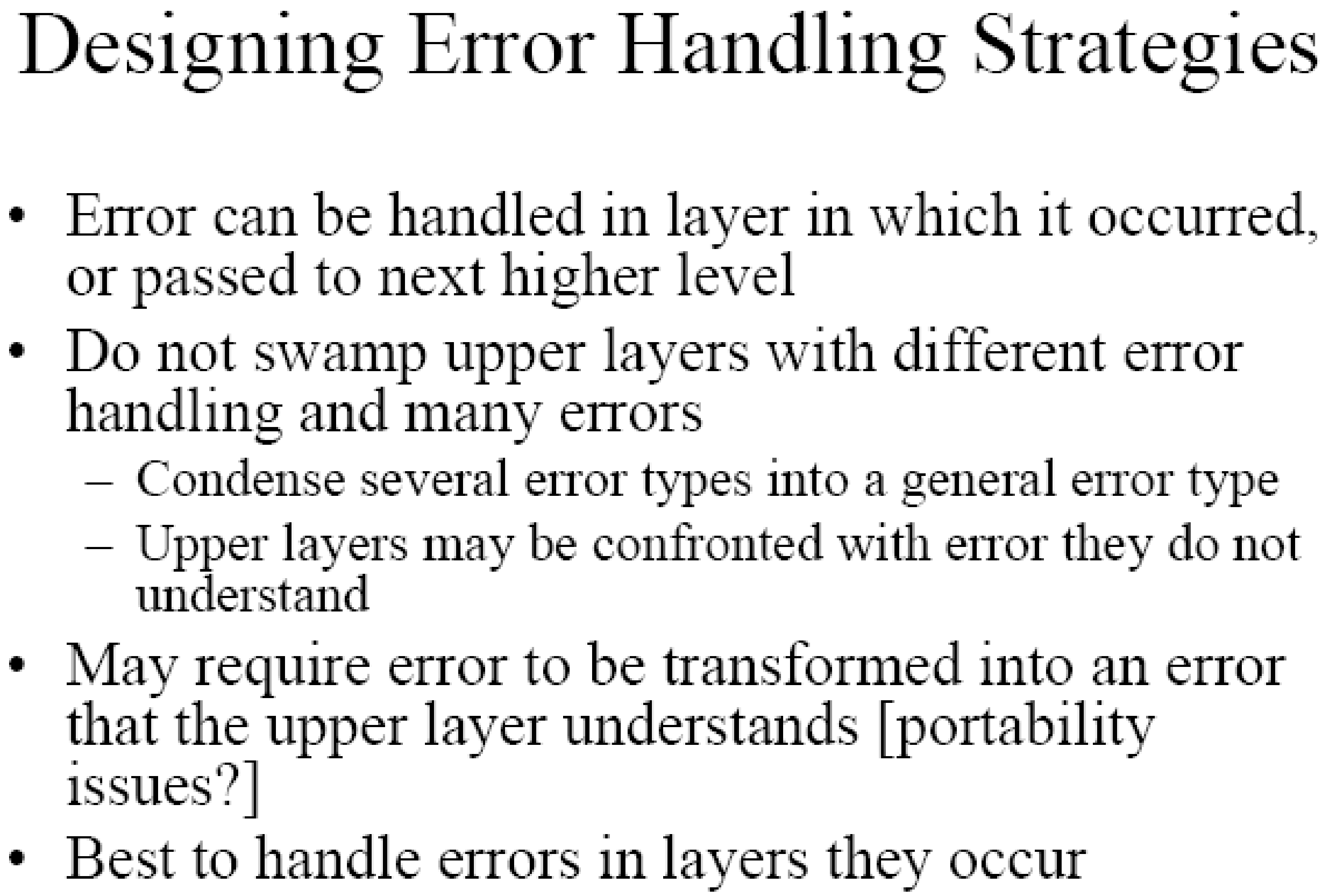
- Strengths
    - **Increasing levels of abstraction as we move up through layers – partitions complex problems**
    - **Maintenance - in theory, a layer only interacts with layers above and below. Change has minimum effect.**
    - **Reuse - different implementations of the same level can be interchanged**
    - **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



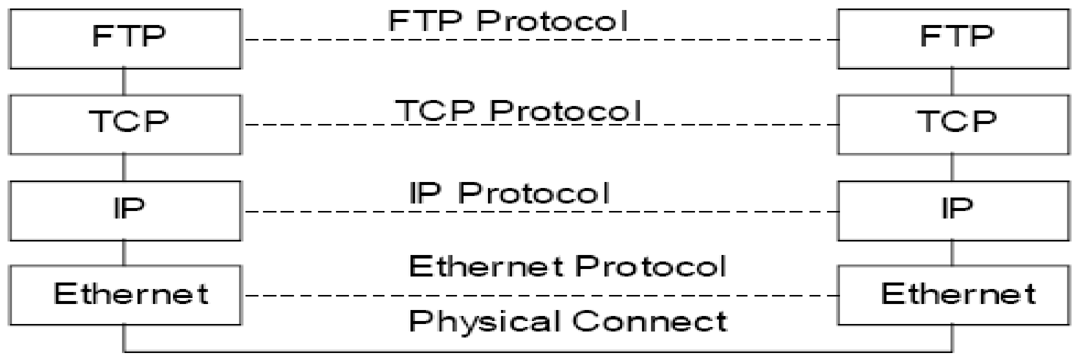
Designing Error Handling Strategies

- Error can be handled in layer in which it occurred, or passed to next higher level
- Do not swamp upper layers with different error handling and many errors
  - Condense several error types into a general error type
  - Upper layers may be confronted with error they do not understand
- May require error to be transformed into an error that the upper layer understands [portability issues?]
- Best to handle errors in layers they occur

Layered System Examples

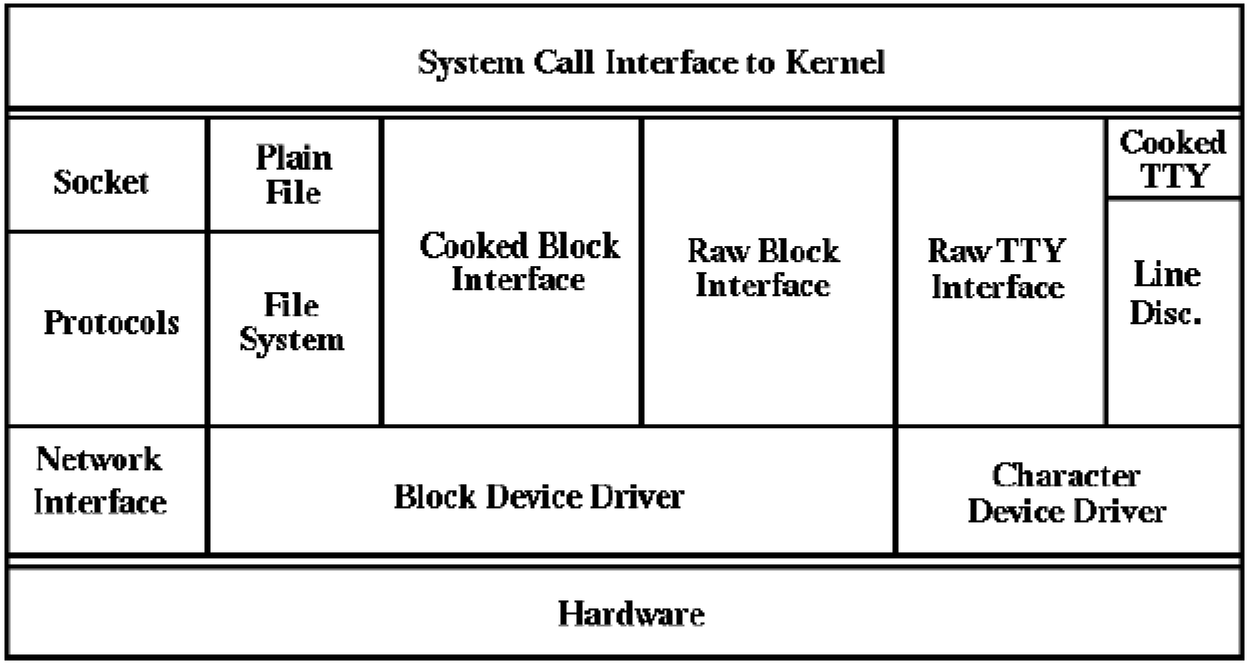
- Example 1: ISO defined the OSI 7-layer architectural model with layers: Application, Presentation, …, Data, Physical.
  - Protocol specifies behaviour at each level of abstraction (layer).
  - Each layer deals with specific level of communication and uses services of the next lower level.
- Example 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".
- ...

Example: FTP on top of TCP/IP

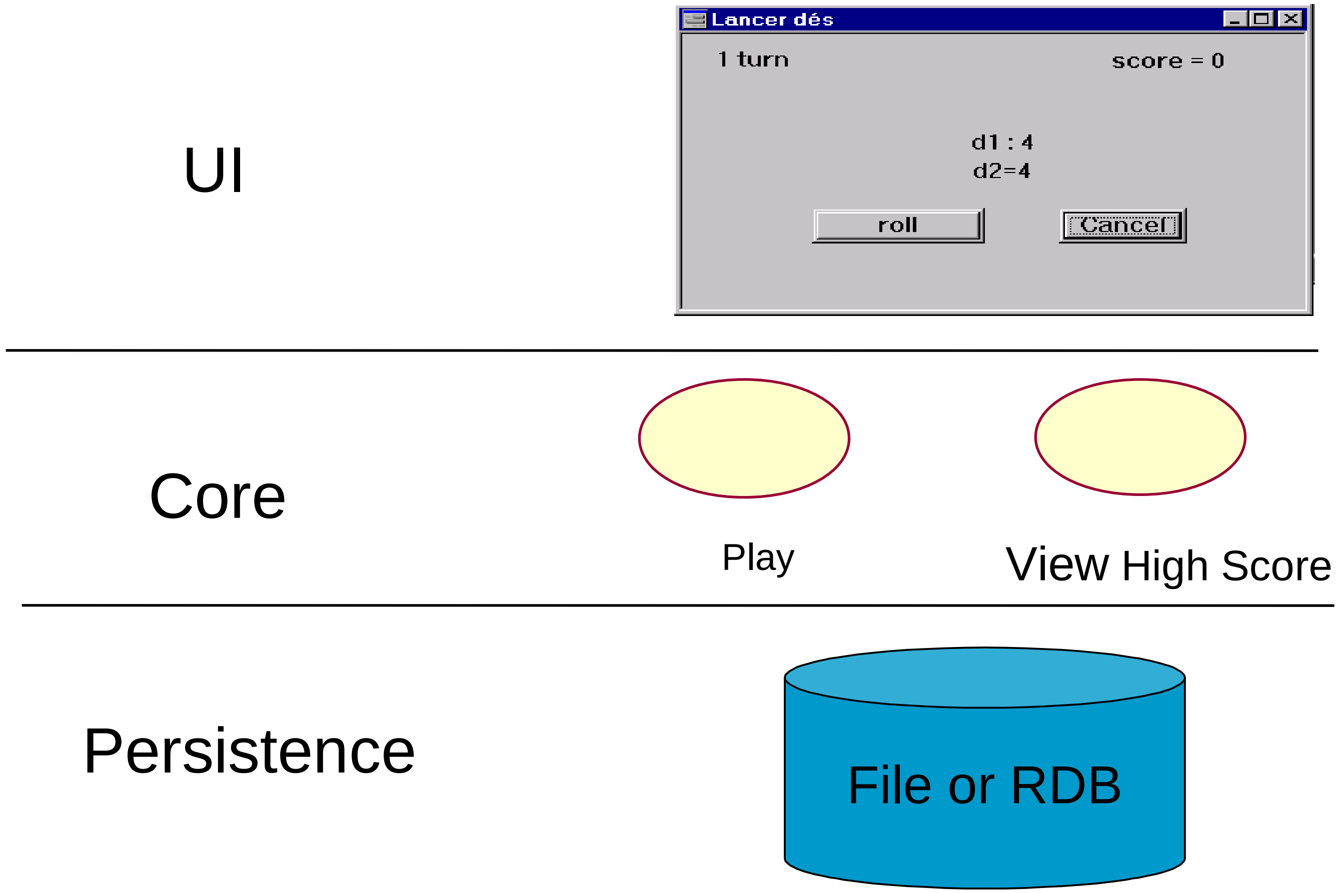


Layered Architectures

The Unix Layered Architecture



Applying Layers Architecture



Model-View-Controller

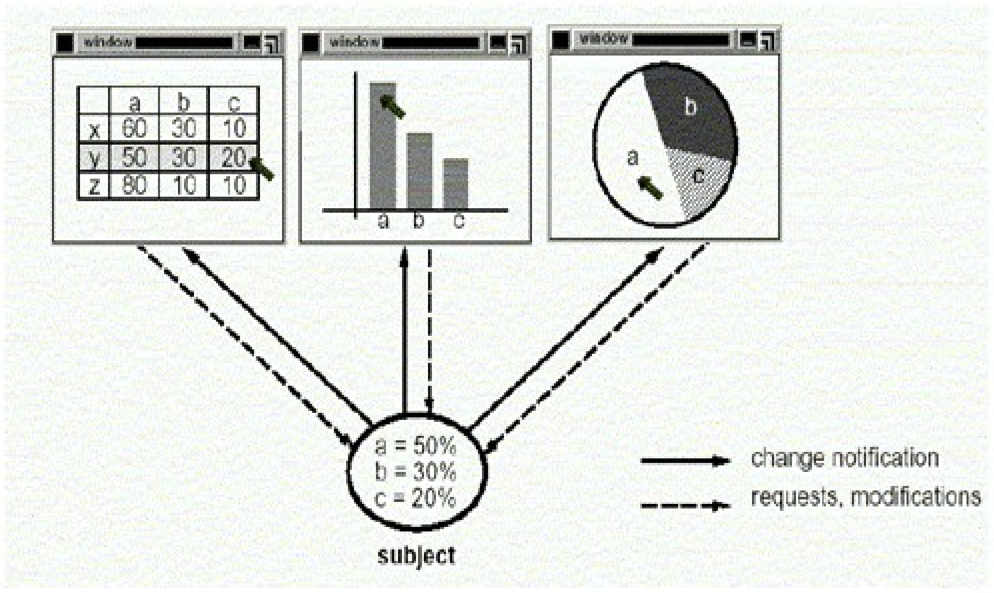
- A decomposition of an interactive system into three components:
  - **A model containing the core functionality and data,**
  - **One or more views displaying information to the user, and**
  - **One or more controllers that handle user input.**

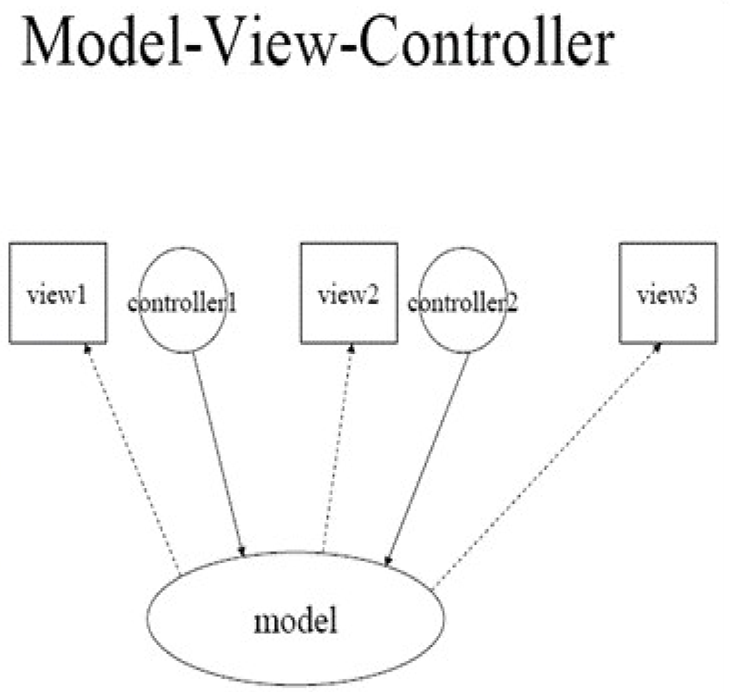
- A change-propagation mechanism (i.e., observer) ensures consistency between user interface and model, e.g.,
  - **If the user changes the model through the controller of one view, the other views will be updated automatically**

- 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*.

Compound MVC

MVC components:

- Strategy

  The view is configured with a given strategy, as provided by the controller. Yes, this implies that the same view could work with a different controller if you want the system's behaviour to change.

- Observer

  The model is the concrete observable object, and the views are concrete observers.

- Composite

  The view may include nested components as part of a GUI. When the controller tells the view to update itself, all the subcomponents will be taken care of as well.

Observer

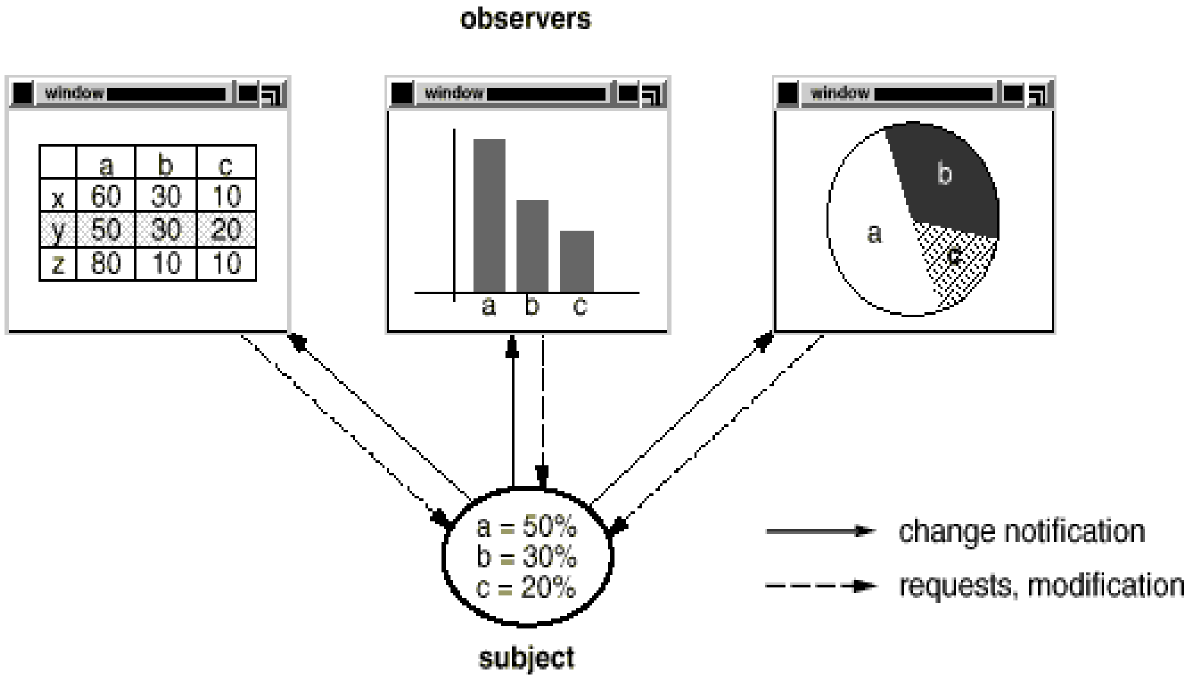
Intent: Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Translation:

Set up the moral equivalent of a newspaper or magazine subscription service. :-)

Observer

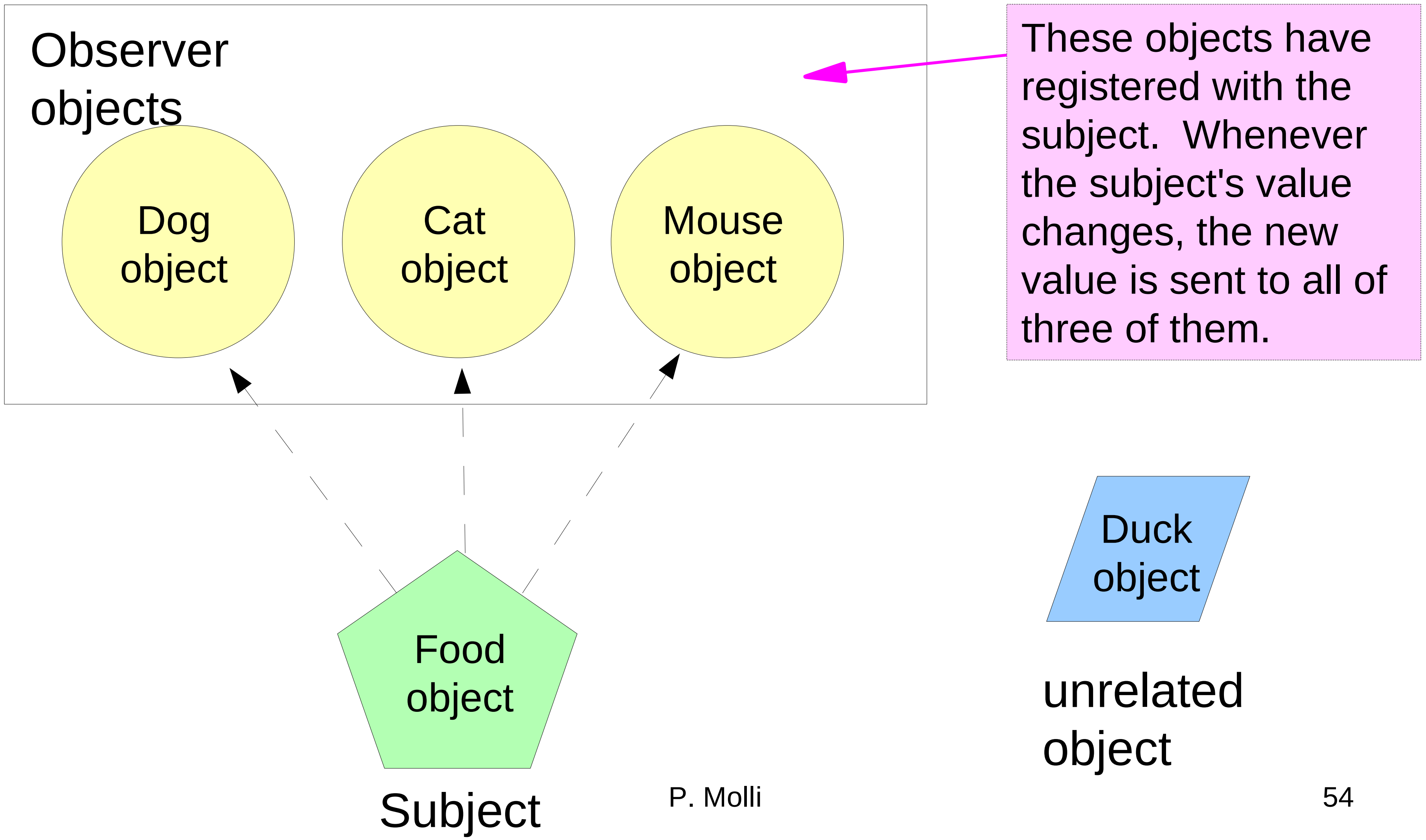
When an object's value is updated, observers watching it are notified that the change has occurred.



Observer: Applicability

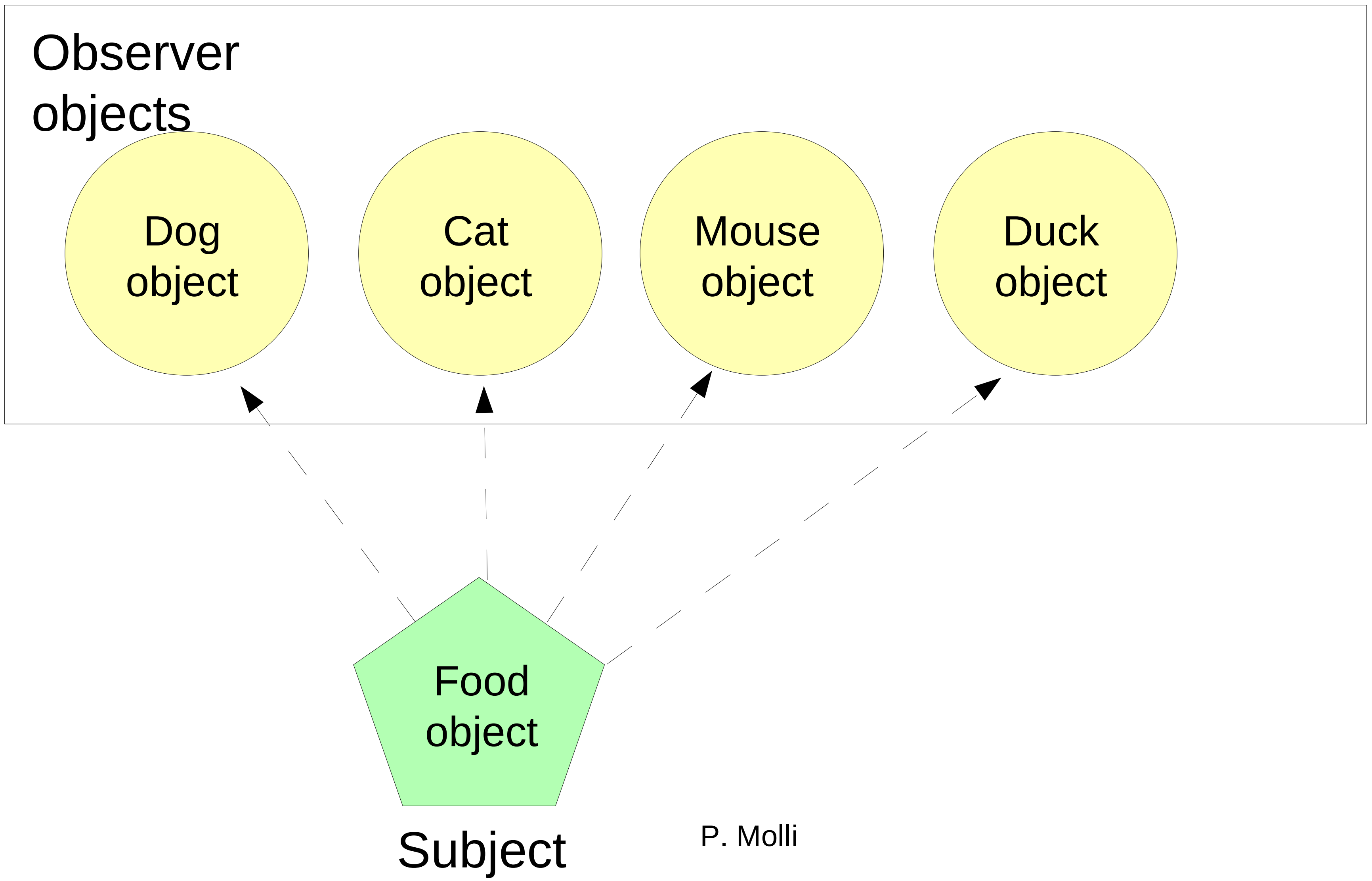
- A change to one object requires changing an unknown set of other objects.
- Object should be able to notify others that may not be known from the beginning.
- Metaphor = newspaper or magazine subscription:
  - A publisher goes into business and starts printing a periodical.
  - You subscribe.
  - Every 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: Data Flow

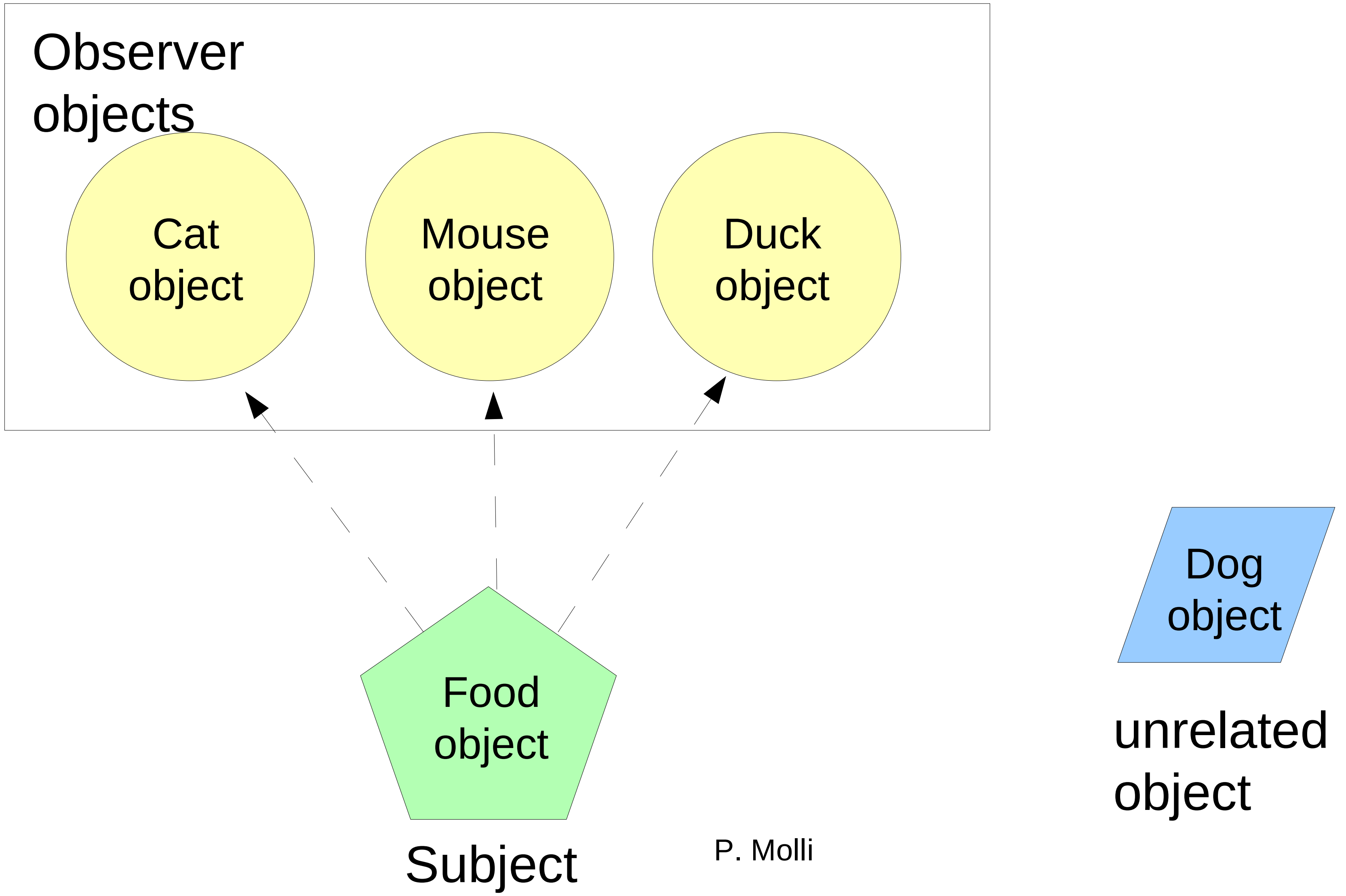


These objects have registered with the subject. Whenever the subject's value changes, the new value is sent to all of three of them.

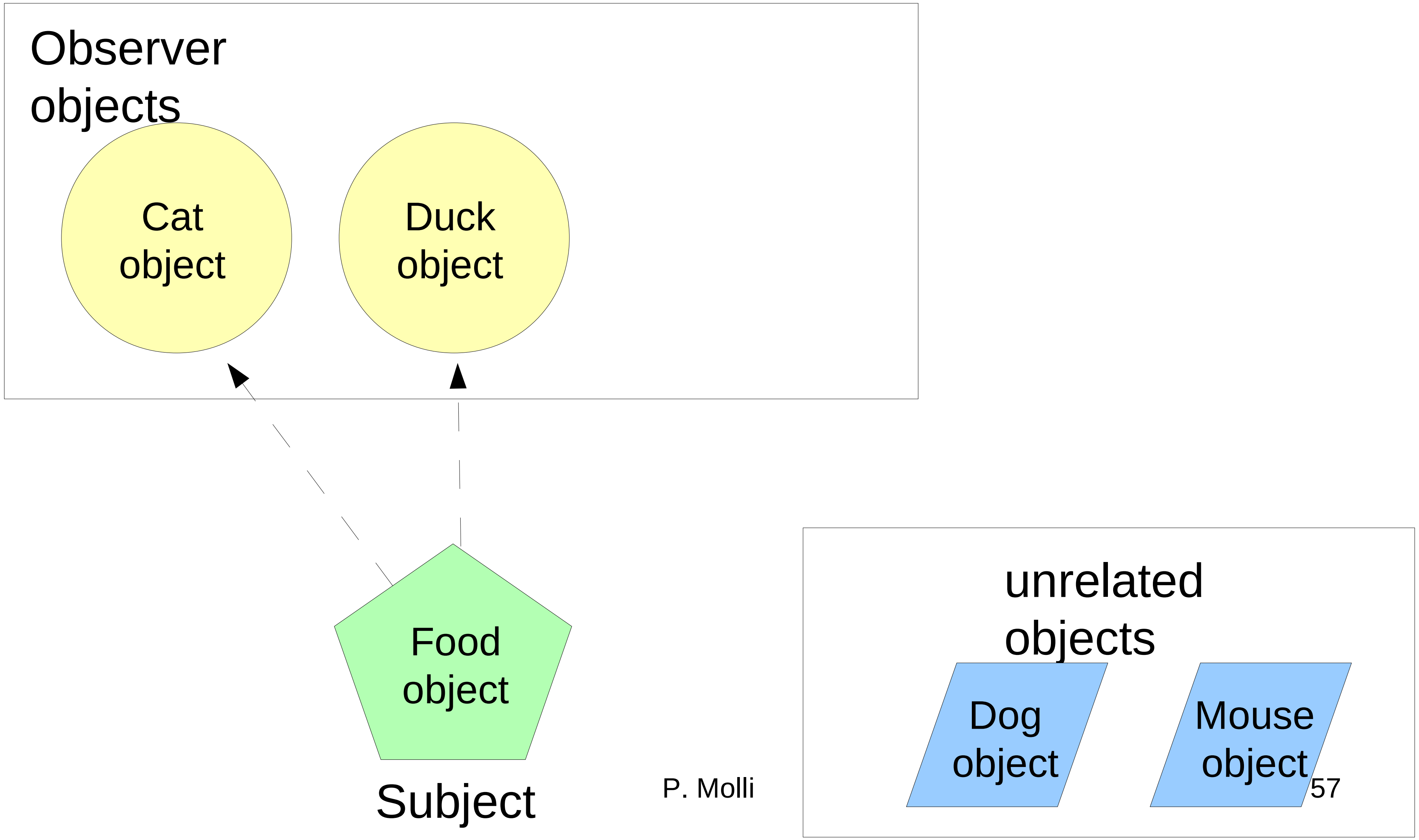
Observer: Data Flow



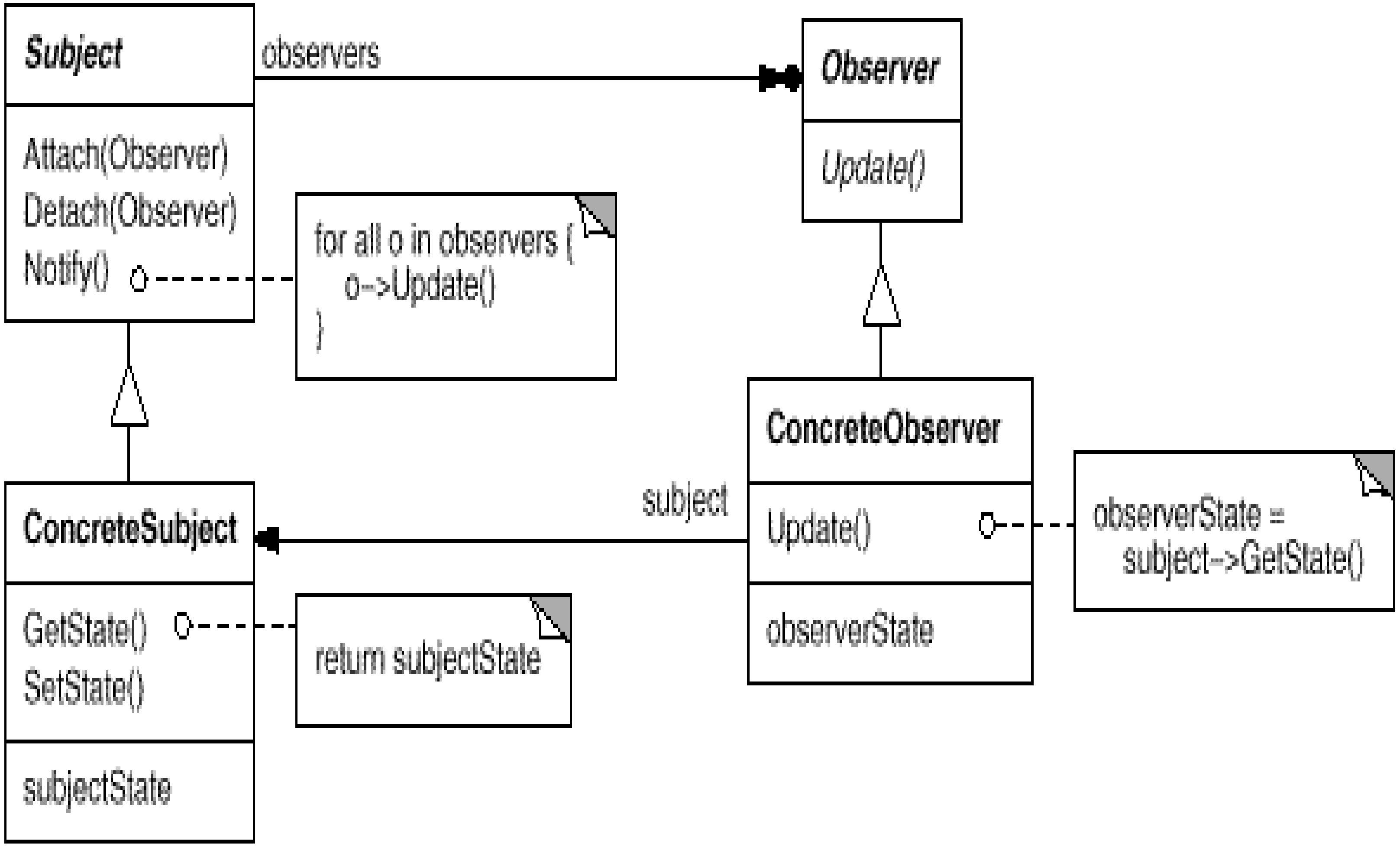
Observer: Data Flow



Observer: Data Flow



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).

Composite Pattern

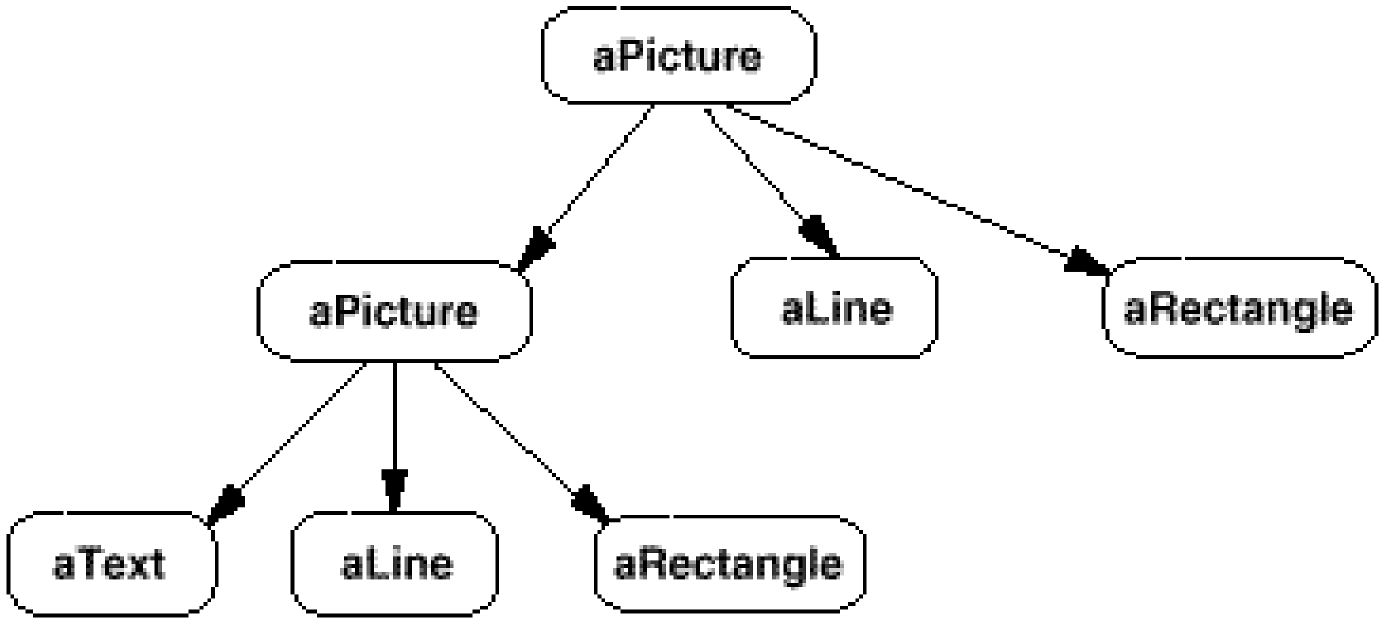
Intent: Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Translation:
Allow a container to contain itself. :-)

Composite Example

Consider a graphical drawing editor that allows you to build an image out of components — and also allows you to use images you've constructed as new components.

This supports the creation of recursive structures, *e.g.*

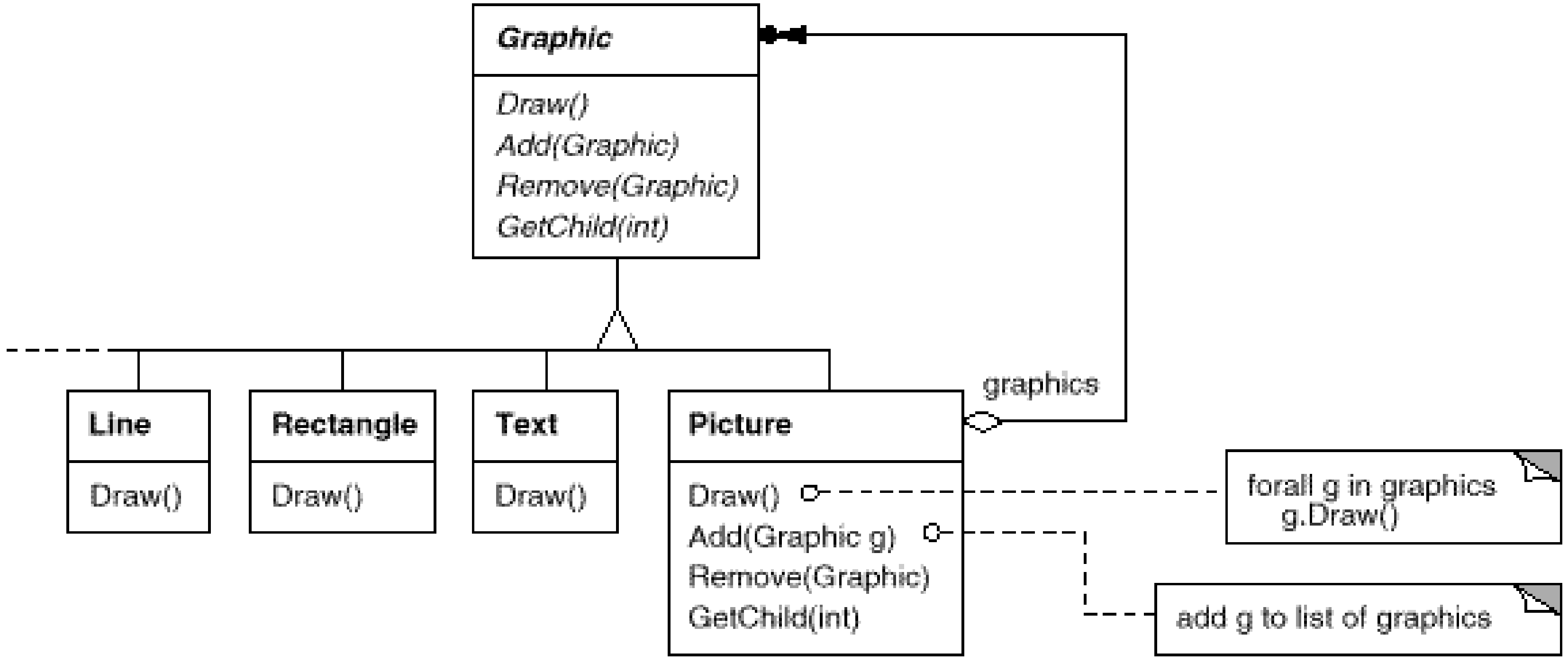


Composite Question

- The obvious implementation is a standard tree structure, in which each leaf node represents a primitive object (text, line, circle, rectangle, etc.) and each subtree represents a container (*i.e.*, an image built from primitive objects in this particular example).
- ...but the problem with this approach is that code which uses these classes must treat primitive objects and containers differently. This means that knowledge of the two node types must be built in, and is thus subject to change.
- How can we encapsulate this knowledge?

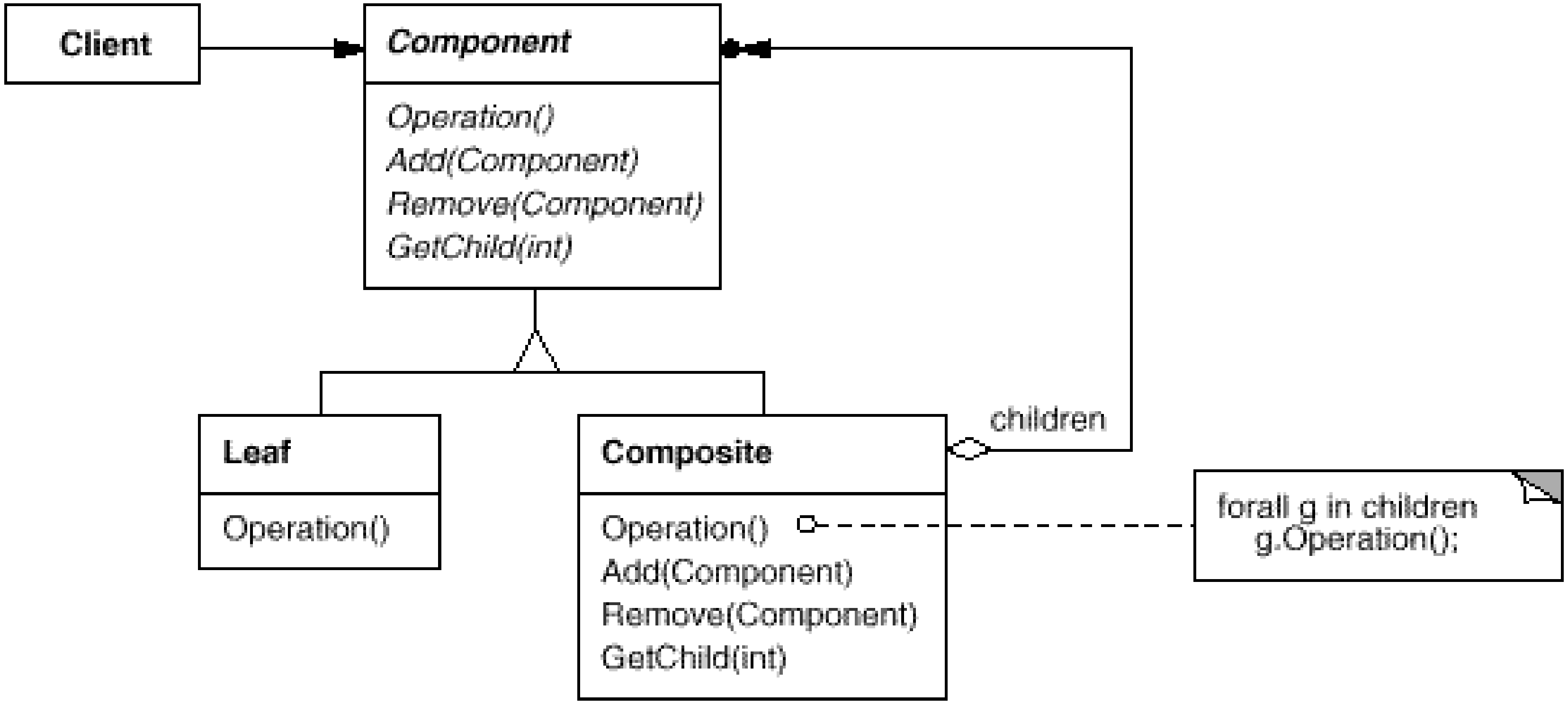
Composite Example

The key is that in the Composite pattern, both leaf nodes and containers are implementations of the same interface or abstract class:



Composite Structure

In general, that becomes:



Composite Consequences

Composite...

- ...defines class hierarchies consisting of primitive objects and composite objects in a way such that wherever client code expects a primitive object, it can also take a composite object.

- ...simplifies the client, by allowing it to treat composite structures and individual objects uniformly. The client doesn't need to know which kind of node it's dealing with.

- ...makes it easier to add new kinds of components, without having to change any existing client code.

- ...can make your design **too** general, by making it difficult to restrict the components of a composite object. There's nothing in the pattern to prevent any kind of component being added to a container, so applications which care must check at run-time.

Composite Consequences

A coding issue with Composite:

- Since every component in a Composite hierarchy must implement the same interface, it can happen that some operations won't make sense in a given context.

- For example, suppose we use Composite to manage a restaurant menu. The menu as a whole may have submenus for breakfast, lunch and supper; the lunch and supper menus may have submenus of their own (*e.g.* appetizers, soups, desserts, ...). Of course, each menu also has actual food items. :-)

- Suppose that the food items implement query methods such as isVegetarian() or containsNuts(). This means that a typical tree traversal may attempt to call these methods on the submenus as well. How can this be handled cleanly?

Composite Consequences

Possibilities include:

- Implement stub methods that return zero, false or an empty string.
- Throw an exception when a method is called in a context where it makes no sense.
- Have the client test which type of node it's dealing with before attempting to call a potentially inappropriate method.
- other (use your imagination :-)

Composite Consequences

- Notice that Composite loses some cohesion by forcing a single class to include both its intended purpose (*e.g.* draw() in the case of the image editor) and also tree management operations (*i.e.* add() and remove()).
- This is an example of a design tradeoff. On the one hand, we lose cohesion, but on the other, we gain *transparency (i.e.* the ability to use the same code to traverse all nodes in the tree, without having to care about the type of each node).
- Which principle is more important? There's no one right answer; the best choice will depend on your circumstances and priorities.