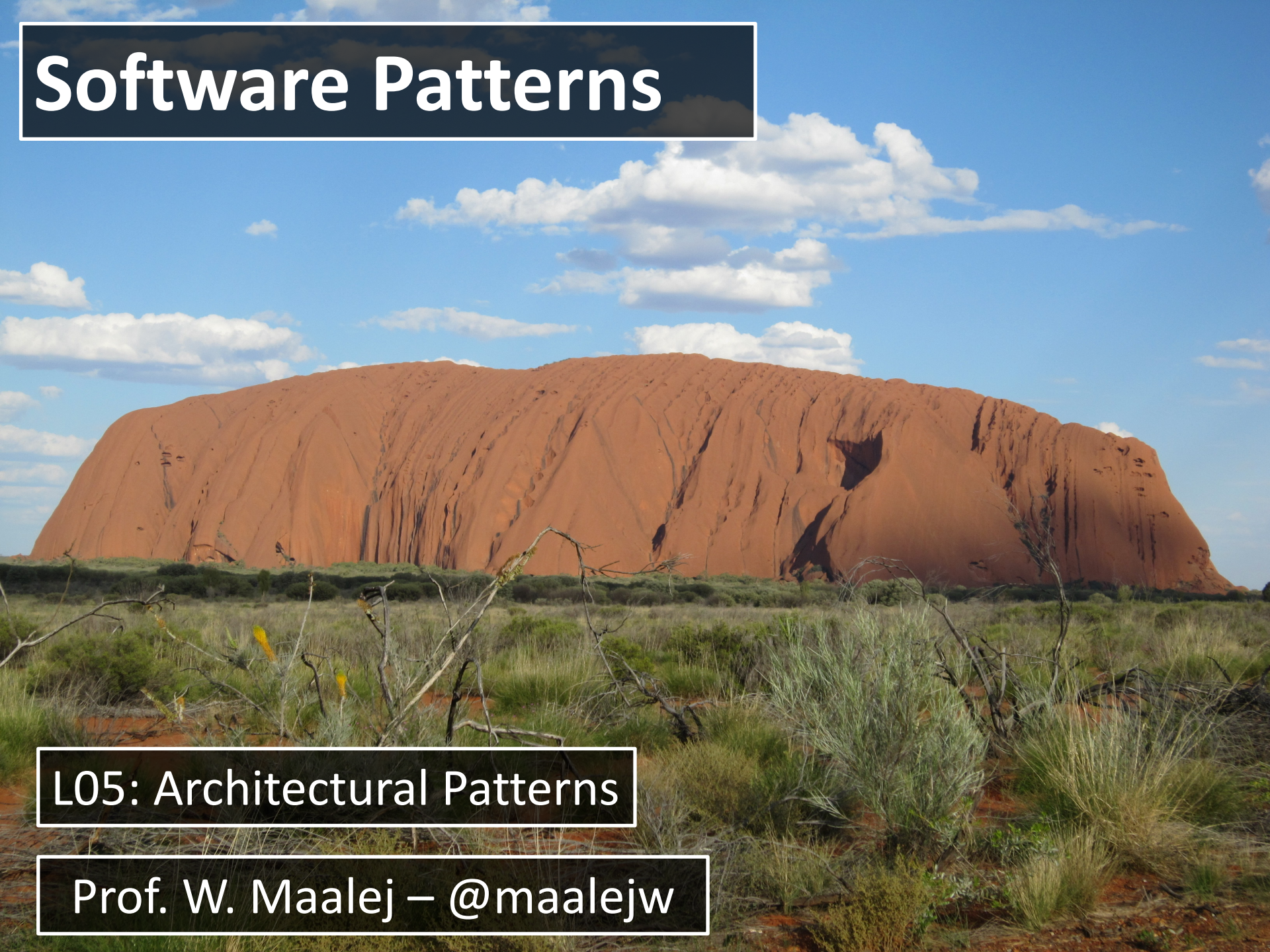
Software Patterns



L05: Architectural Patterns

Prof. W. Maalej – @maalejw

Outline

1. **Software Architecture**
2. Layers
3. Blackboard
4. Model View Controller

What is good architecture?

- ☑ 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

How?

- Reuse of established engineering knowledge
  - Application of architectural styles
  - Analogous to design patterns in detailed design

Architecture is an art

- Inventing a novel architecture is a highly creative act
- Requires
  - Knowledge of existing work
  - Experience

Styles in building architecture

Ranch style

T-Ranch style

Raised Ranch style

- Customer picks an architectural style
  - Main components of the style are fixed
- Architect changes details according to requirements of the customer
- We apply the same approach to software architecture

Elements of a software architecture

- **Components (Subsystems)**
  - Computational units with specified interface
  - Examples: databases, filters, layers, objects

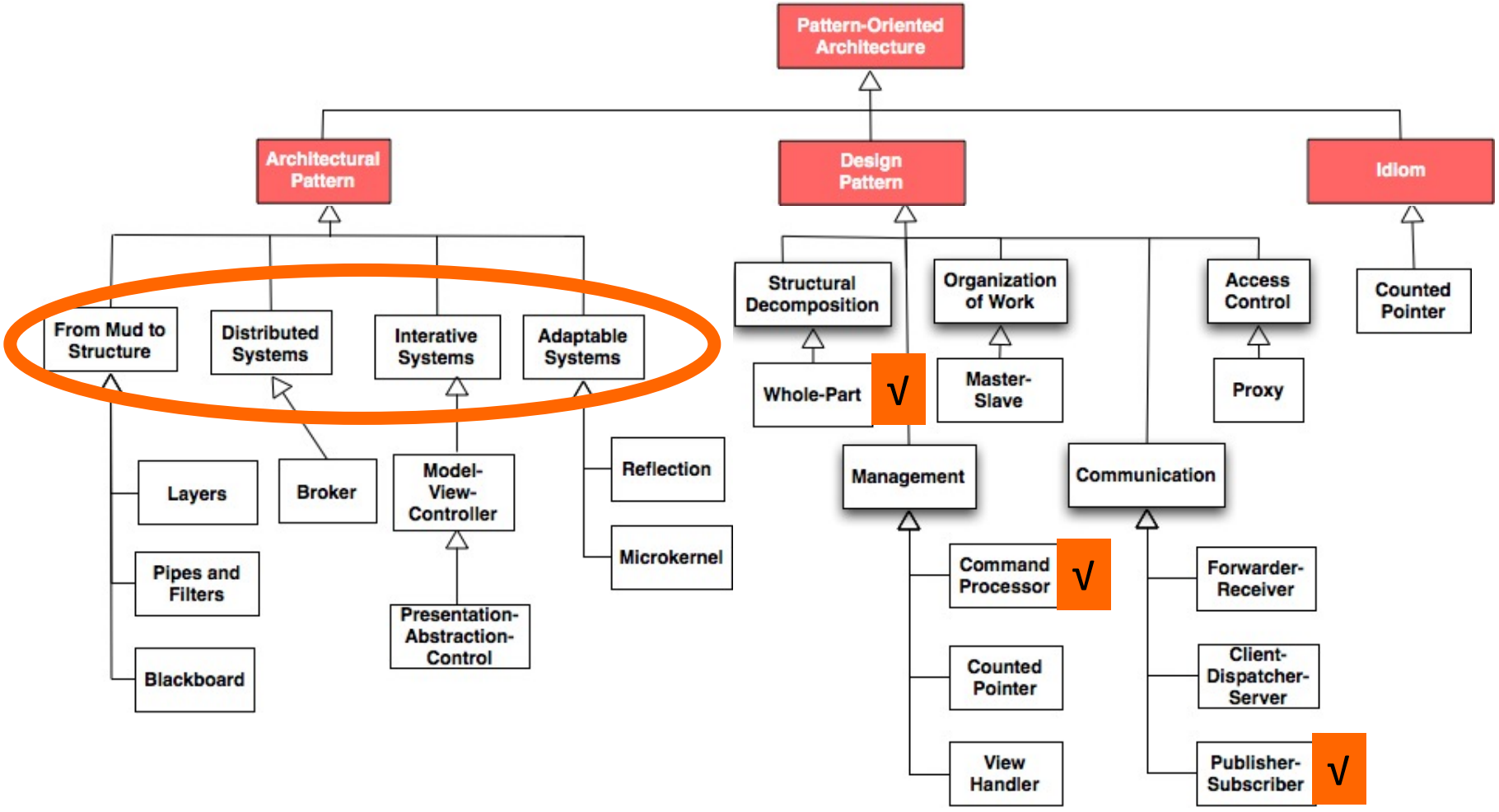
- **Connectors (Communication)**
  - Interactions between the components (subsystems)
  - Examples: method calls, pipes, event broadcasts, shared data

- See M. Shaw, D. Garlan: *Software Architecture*, Prentice Hall, 1996.

Mini-history of software architecture

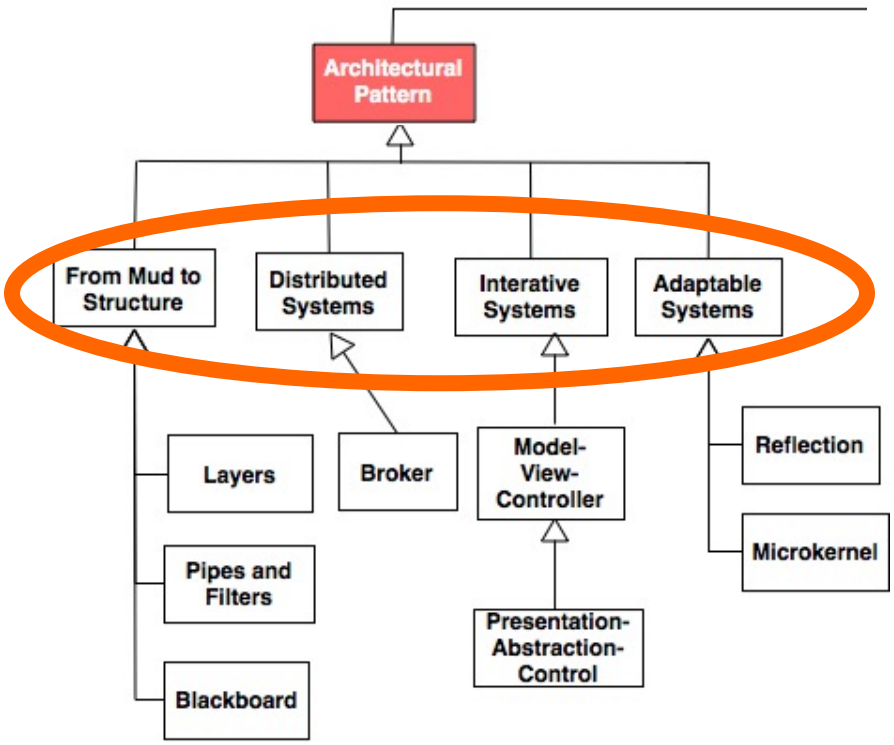
- **1968:** E.W. Dijkstra introduces **layering** in the T.H.E. system
- **1974:** D. Parnas introduces **modularity** and **information hiding** as good system design principles
- **1992:** D. Perry and A. Wolf use the term **architectural style**
- **1996:** F. Buschmann, R. Meunier, H. Rothner, P. Sommerlad, M. Stal introduce the term **architecture pattern**
  - *Pattern-Oriented Software Architecture: A System of Patterns*
- **1996:** Mary Shaw and David Garlan propose the term **software architecture**
- **2000:** IEEE Standard 1471-2000
  - Defines a **meta-model for software architectures**
  - Introduces **viewpoints** to allow the description of a software architecture from the perspective of a stakeholder

Pattern-oriented architecture (GoF taxonomy)



F. Buschmann, R. Meunier, H. Rothner, P. Sommerlad, M. Stal
Pattern-Oriented Software Architecture:A System of Patterns, 1996

Pattern-oriented architecture (GoF taxonomy)



From Mud to Structure: Subsystem decomposition

- **Layers:** Each subsystem presents a layer of abstraction
- **Pipes and Filters:** Each subsystem is a processing step formulated in a filter. Filters are connected by pipes
- **Blackboard:** The subsystems are knowledge experts working together to solve a problem without a known solution strategy

Distributed Systems: Collaborating systems on different processors

- **Broker:** Distributed subsystems interact by remote service invocations (RPC, RMI,...)

Interactive Systems: Systems interacting with a user

- **Model-View-Controller:** Subsystem decomposition with 3 components: Model (entity objects), view (boundary objects), controller (controller objects)
- **Presentation-Abstraction Control:** Hierarchy of cooperating agents ("self-reliant subsystems", [Coutaz 1987])

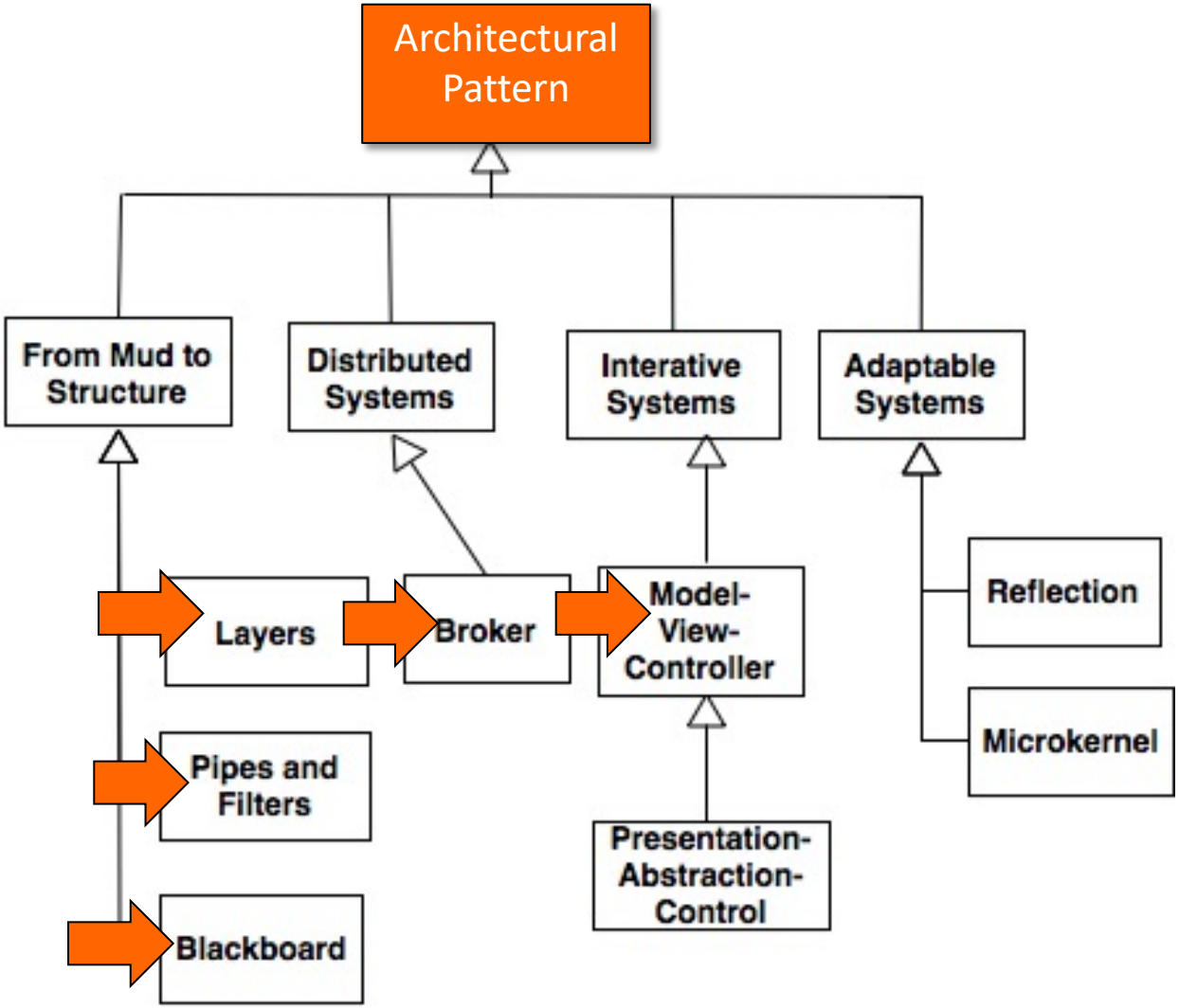
Adaptable Systems: Systems evolving over time

- **Microkernel:** Extensible minimal functional core
- **Reflection:** Self-awareness provides information about system properties

Architectural pattern vs. Architectural style

- Buschmann et al. [pp 396]:
  - Architectural styles only describe the overall structural frameworks for applications.
  - Architectural patterns define the basic structure of an application
  - Architectural styles are independent from each other, a pattern depends on smaller patterns
  - Patterns are more problem oriented than architectural styles
- We don't see any difference
  - **Architectural Style, Architecture Pattern:** A pattern for a subsystem decomposition
  - **Software Architecture:** Instance of an architectural style (architectural pattern)

Architectural patterns taxonomy



Outline

1. Software Architecture
2. **Layers**
3. Blackboard
4. Model View Controller

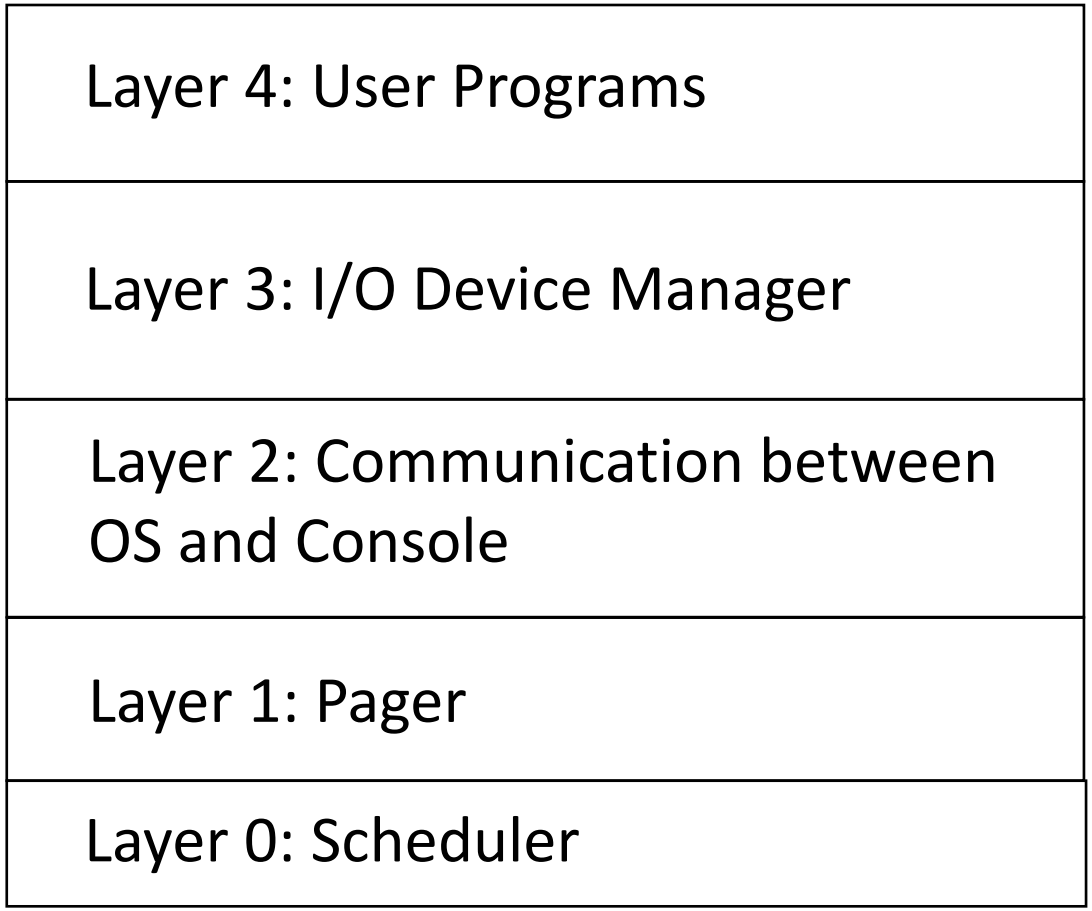
What is the problem with this drawing?

Dijkstra's answer to "spaghetti design"

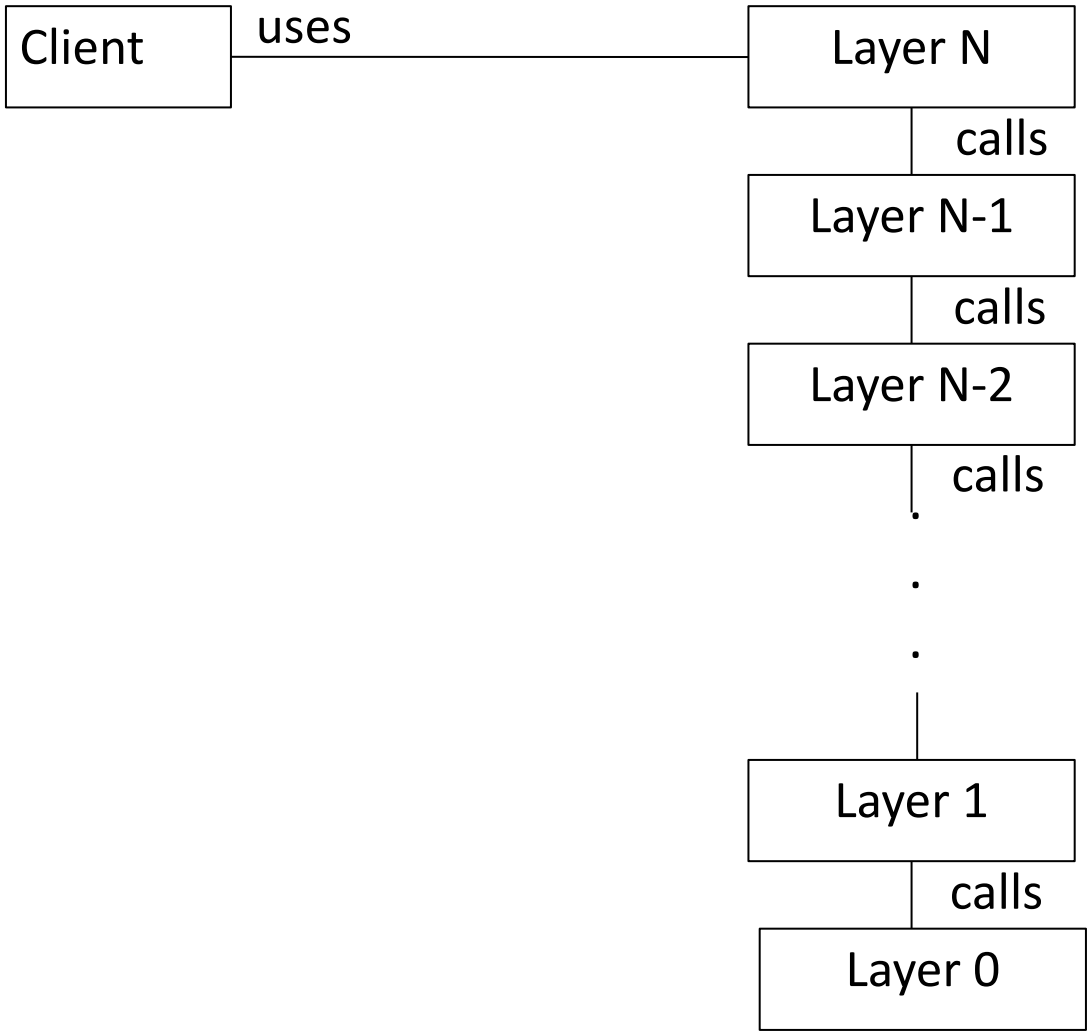
- Dijkstra revolutionary idea (1968!)
  - A system should be designed and built as a hierarchy of layers:
    - Each layer uses only the services offered by the lower layers
- The T.H.E. system
  - T.H.E. = Technische Hochschule Eindhoven
- An operating system for single user operation
  - Supporting batch-mode
  - Multitasking with a fixed set of processes sharing the CPU

The layers of the T.H.E. system

- “An operating system is a hierarchy of layers, each layers using services offered by the lower layers”

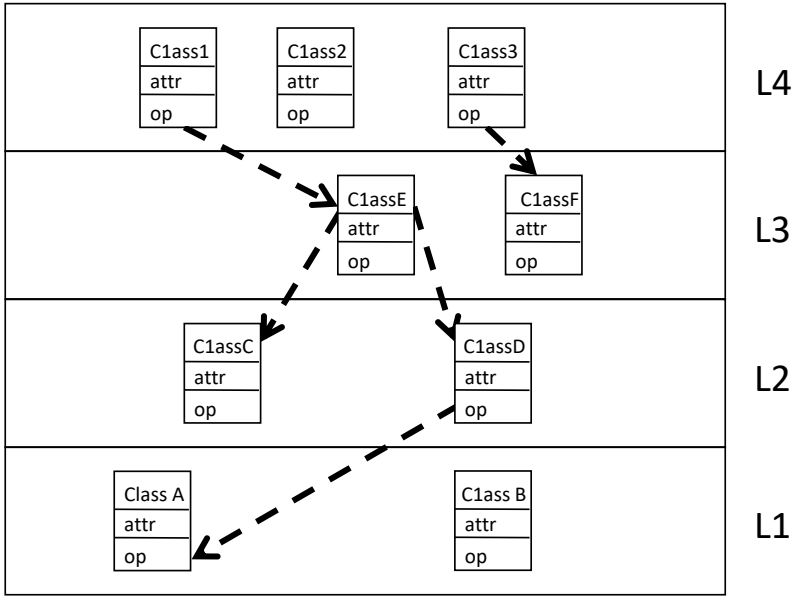


The layers pattern



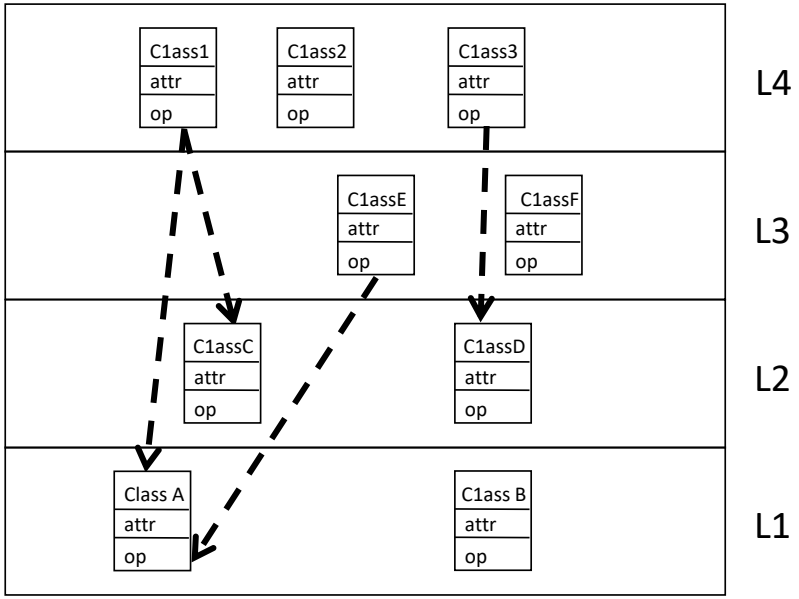
Closed architecture (opaque layering)

- Each layer can only call operations from the layer below (called “direct addressing” by Buschmann et al.)
- Design goals: Flexibility, Maintainability



Open architecture (transparent layering)

- Each layer can call operations from any layer below (“indirect addressing”)
- Design goal: Runtime efficiency



5 steps to create a layered architecture (System design)

1. Define the **abstraction criterion**
   - Also called “the conceptual distance to the existing system”
   - Example: The degree of customization for a specific domain
2. Determine the **number of abstraction levels**
   - Each abstraction layer corresponds to one layer of the pattern
3. **Name** the layers and **assign tasks** to each of them
   - The task of the highest layer is the **overall system task**, as seen by the client
   - The tasks of all the other layers are **helper layers**
4. Specify the **services**
   - Lower layers should be "slim“, while higher layers should have a broader applicability
5. **Refine** the layering

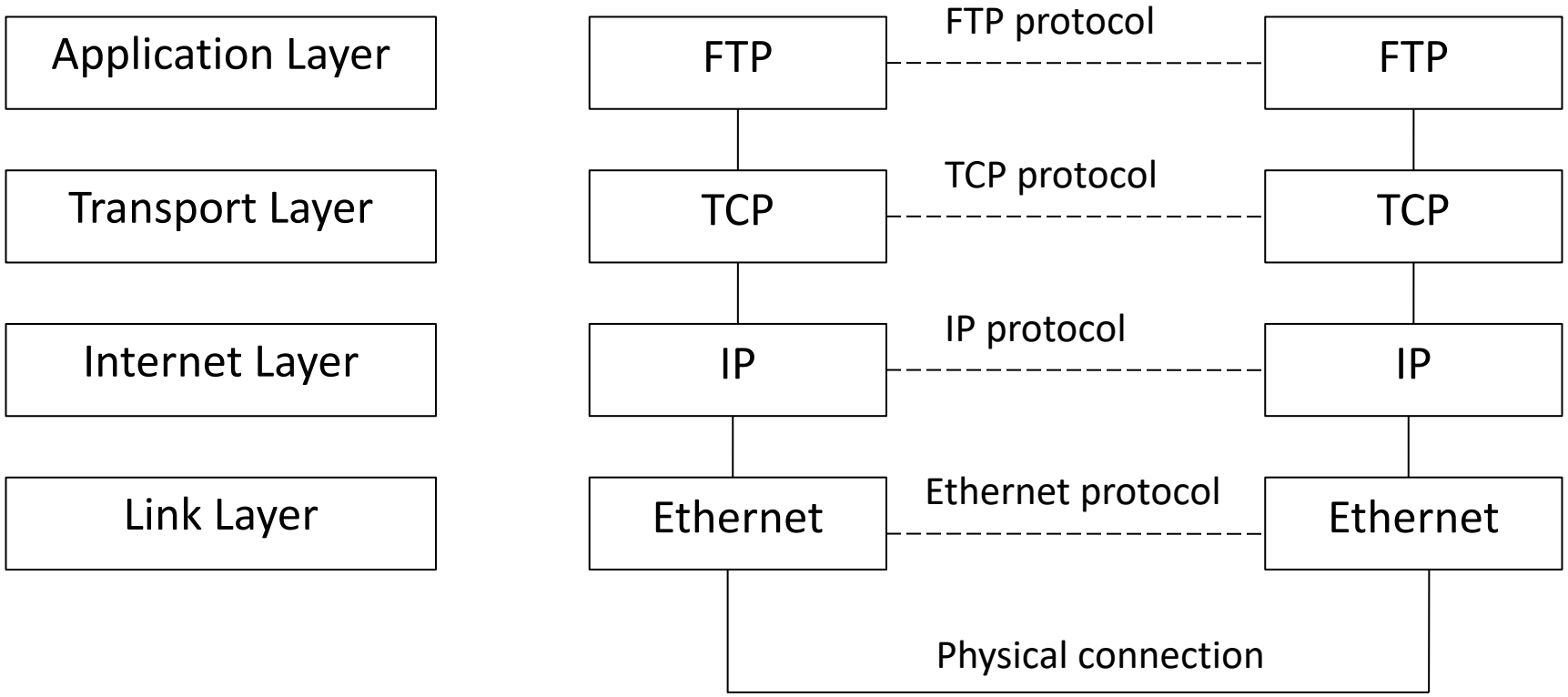
5 steps to create a layered architecture (Object design)

1. Specify an **interface** for each layer
   - **Black box** approach: Encapsulate each layer in a **façade** pattern
   - **White box** approach: Components in the layer are visible to the calling layer
2. **Structure** the individual layers ("Avoid chaos inside a layer")
   - Use the **bridge** pattern to separate stubs from implementation code
   - Use the **strategy** pattern for dynamically changing algorithms
   - Try to identify partitions in the layer and map them to separate components
3. Specify the **communication protocol** between adjacent layers
   - **Push model**: If Layer N invokes a Layer N-1 service, passing all required information in the call
   - **Pull model**: Layer N-1 fetches available information from layer N

5 steps to create a layered architecture (Object design)

4. Decouple adjacent layers
   - Use a closed architecture with an opaque call hierarchy
   - Let only return parameters bring results to the upper layer
   - For bottom up communication (Layer N-1 want to talk to Layer N), use **callbacks**:
     - Use the **command pattern** for encapsulating callback functions into objects
     - The upper layer N registers the callback functions with the lower layer N-1
5. Design an error-handling strategy
   - Rule of thumb: try to handle errors at the lowest layer possible
   - If that is not possible, translate the error into a more general error type
     - Otherwise higher layers are confronted with “cryptic” error messages

Layers in TCP/IP



The TCP/IP protocol was designed by Bob Kahn and Vinton Cerf [Kahn & Cerf, 1974]

The OSI 7-layer model

- **Application Layer 7**
  - Provides miscellaneous protocols for common activities
- **Presentation Layer 6**
  - Structures information and attaches semantics
- **Session Layer 5**
  - Provides dialog control and synchronization facilities
- **Transport Layer 4**
  - Breaks messages into packets and guarantees delivery
- **Network Layer 3**
  - Selects a route from sender to receiver
- **Data Link Layer 2**
  - Detects and corrects errors in bit sequences
- **Physical Layer 1**
  - Transmits bits over a connection (physical or wireless).

Known uses of the layer pattern

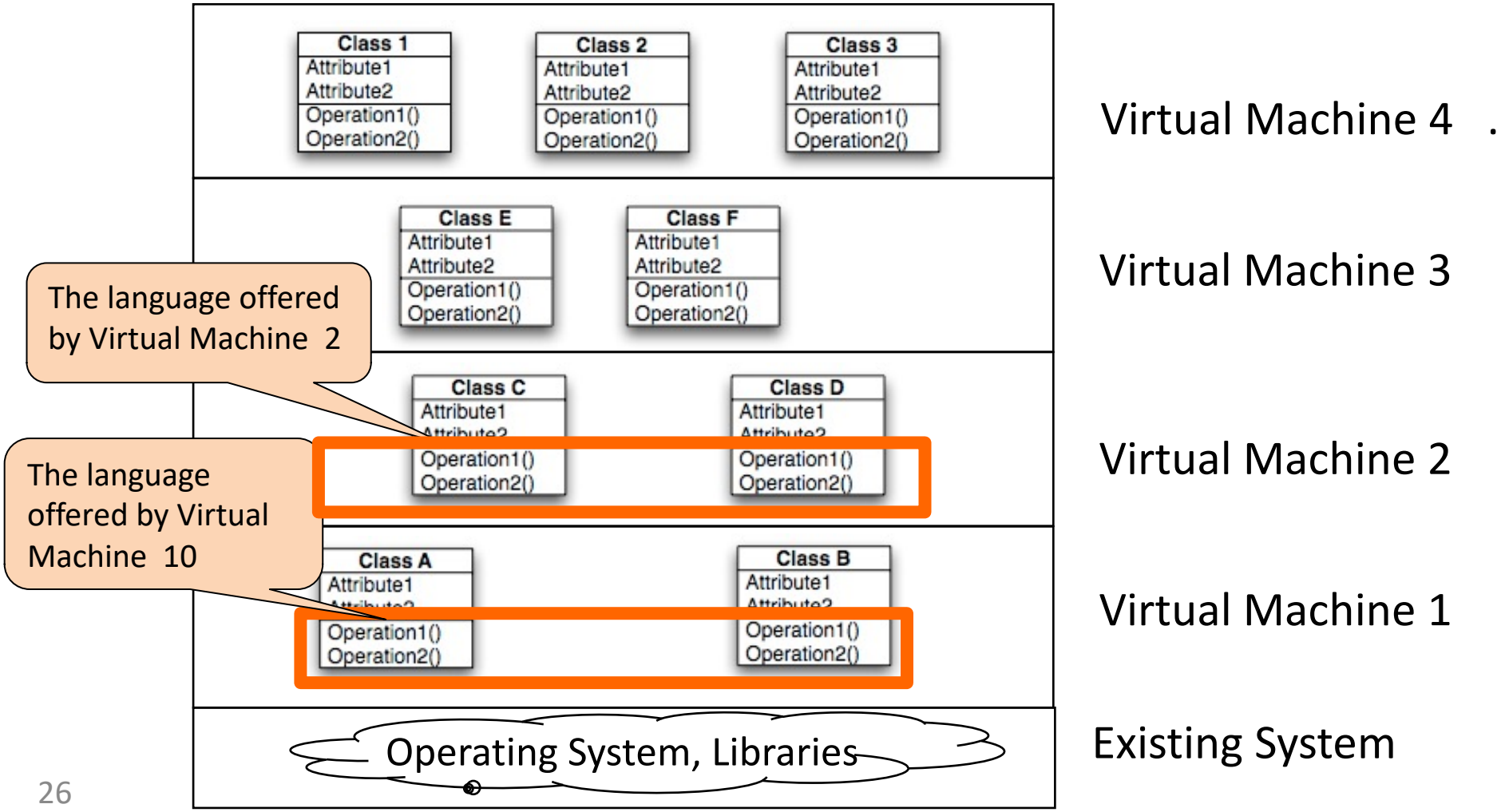
- Information Systems Architectures
  - Client Server Architecture
  - 3 Tier Architecture
  - 4 Tier Architecture
- Virtual Machine and APIs

Virtual machine

- A **virtual machine** is a subsystem connected to higher and lower level virtual machines by "provides services for" associations
- A virtual machine is an **abstraction** that provides a **set of attributes and operations**
- The terms layer and virtual machine can be used interchangeably
  - Also sometimes called “level of abstraction”

Building systems as a set of virtual machines

A system is a hierarchy of virtual machines, each using language primitives offered by the lower machines



Pros of the layers pattern

- **Reusability** of layers, especially in a closed architecture
  - Black box reuse of layers can reduce effort and defects
- Support for **standardization**
  - Different implementations of the same interface can be used interchangeable (esp. if coupled with the bridge or strategy pattern)
- **Low coupling**
  - Changes to the hardware, operating system, database system, user interface system, special data formats affect only one layer
- **Improved testability**
  - A closed architecture with a hierarchical call hierarchy requires half the test compared with a peer-to-peer architecture

Cons of the layers pattern

- A local change in a lower layer may require **rework** in higher layers
  - Example: Changing form 10Mbit/sec Ethernet to 1 Gbit/sec Ethernet
- Lower **efficiency**
  - All relevant data must be **transferred** through the intermediate layers, and may even have to be transformed several times (adding message **headers**, etc.)
  - **Error messages** produced in lower levels must be transformed to higher level messages
    - Example: TCP/IP timeout message due to a car driving into a tunnel

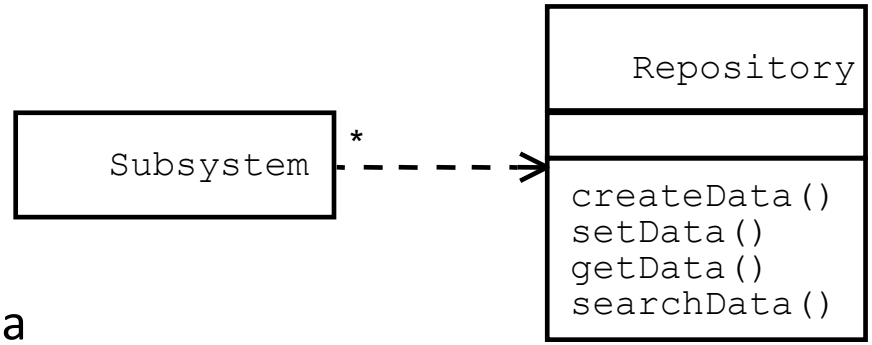
Outline

1. Software Architecture
2. Layers
3. **Blackboard**
4. Model View Controller

Repository architectural pattern

The idea:

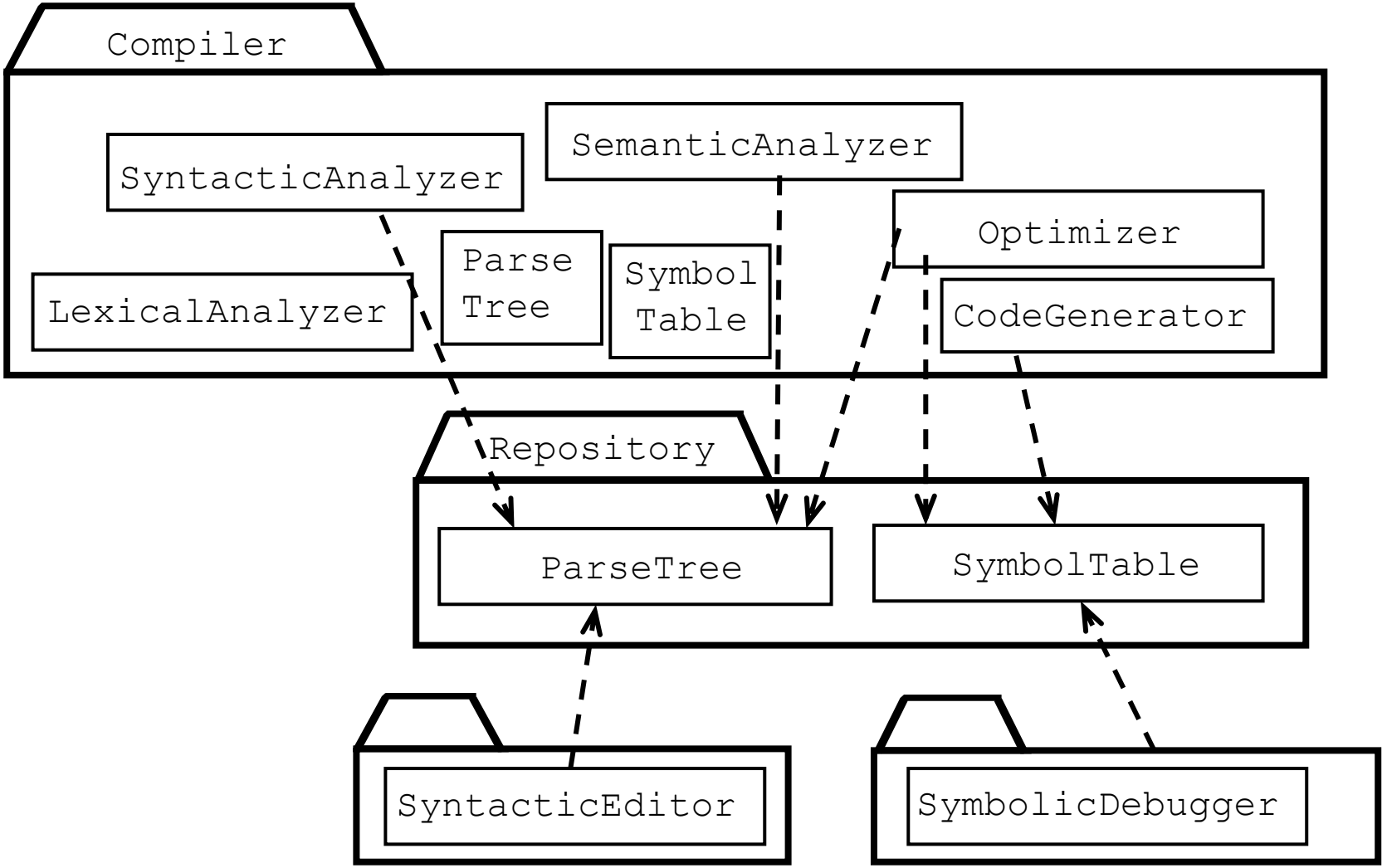
- Support a collection of independent programs that work cooperatively on a common data structure
- **Subsystems** access and modify data from a single data structure called the **repository**



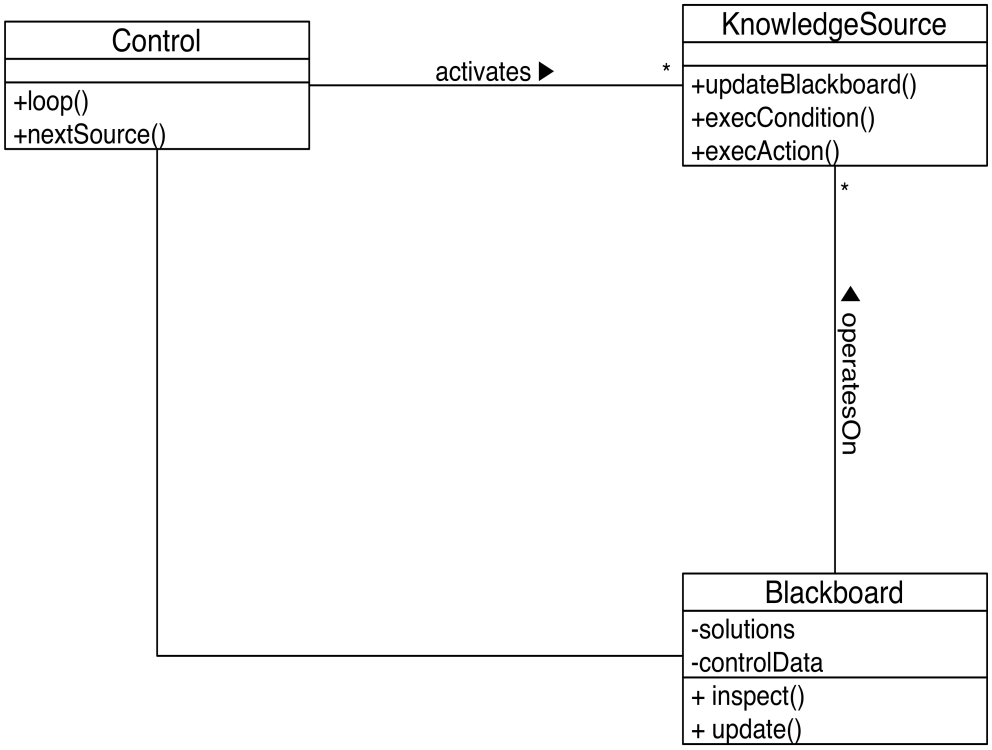
- Subsystems are **loosely coupled** (they interact only through the repository)
- Repository pattern **does not specify any control**
- Control flow is dictated by the repository
  - Through triggers or by the subsystems
  - Through locks and synchronization primitives

- The **blackboard pattern**, we model the controller more explicitly

Example of a repository: Integrated Development Environment (IDE)



Blackboard architectural pattern

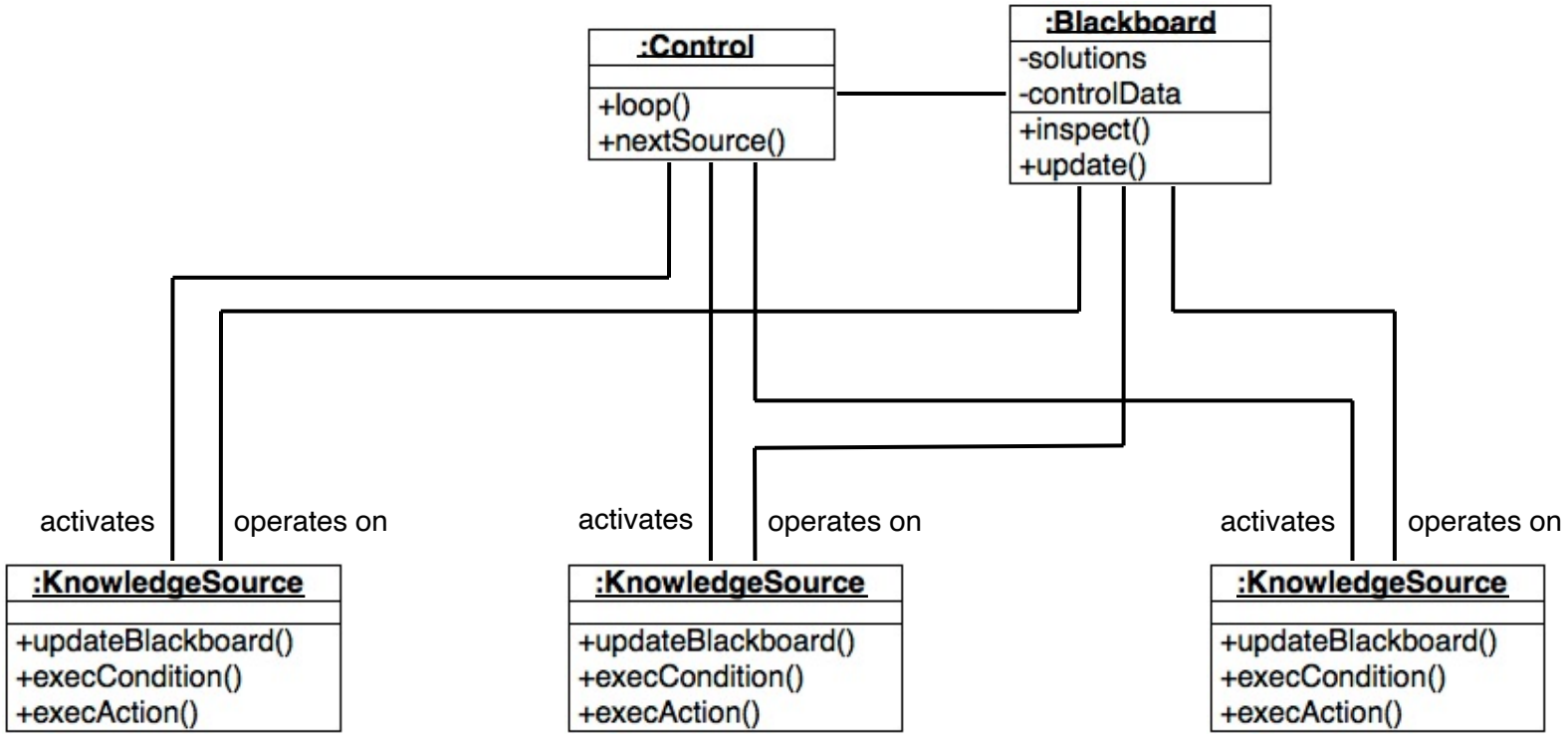


Synonyms:

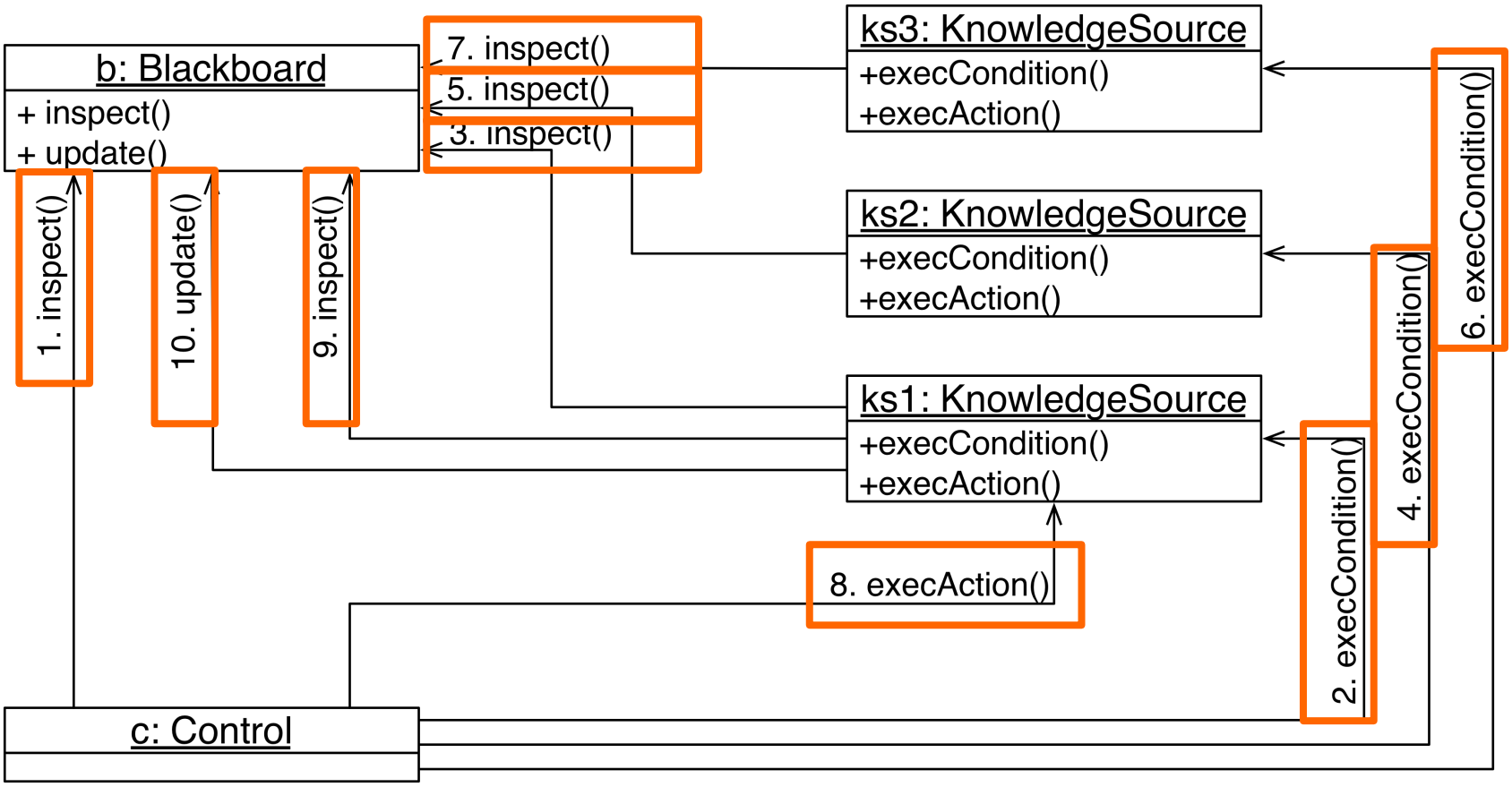
- Control: Supervisor
- Knowledge Source: Specialist, Expert
- Blackboard: Knowledge Sharing Area

- The **Blackboard** is the repository for the problem, solutions, and new information
- All **knowledge sources** read anything placed on the blackboard and place new information created by them on it
- **Control** governs the flow of problem-solving activity in the system, in particular how the knowledge sources get notified of the new information in the blackboard

Blackboard pattern with N knowledge sources (object diagram)



Communication diagram for the blackboard pattern



Blackboard pattern: intent

- Representation of knowledge as self-activating, asynchronous, parallel processes

- Support the problem solving process even when an algorithmic solution is not known

- Several specialized sub-systems assemble their knowledge to build a solution

Pros and cons of the blackboard pattern

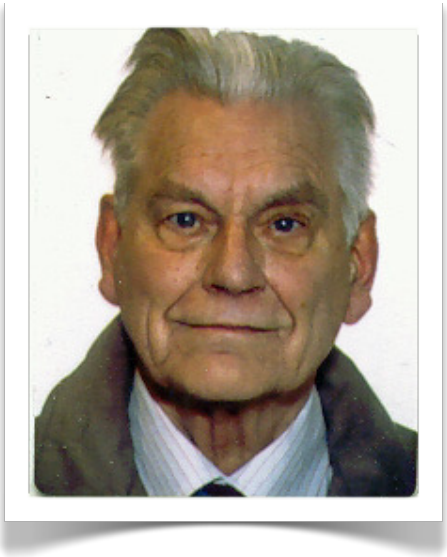
- Pros:
  - \+ **Problem solving support:** Allows to deal with problems in domains which have no closed approaches or a complete search of the solution space is infeasible
  - \+ **Changeability and maintainability:** The low coupling of the knowledge sources and the strict separation between the controller and the data structures in the blackboard allow “scalable change”
  - \+ **Fault tolerance and robustness:** Tolerant against noisy data. All results are just hypotheses. Only hypotheses supported by empirical evidence survive
- Cons:
  - − **Difficulty of testing:** Nondeterministic, results are often not reproducible
  - − **No solution guaranteed:** Cannot solve every problem
  - − **Difficulty to establish a good control strategy:** Requires an experimental approach
  - − **High development effort:** Most blackboard systems take years to evolve

Outline

1. Software Architecture
2. Layers
3. Blackboard
4. **Model View Controller**

Model view controller: A bit of history

- Invented by Trygve Reenskaug in 1979 at Xerox PARC
- In collaboration with Adele Goldberg, who introduced it in Smalltalk 80



Model view controller: history (ctd)

- Reenskaug's and Goldberg's Idea:
  - Bridge the gap between the human user's mental model and the digital model that exists in the computer
    - Donald A. Norman, The Design of Everyday things, 1990
  - Useful when **same model element** has to be **presented in different contexts simultaneously**

- MVC introduces three layers of abstraction
  - Model objects represent the application domain knowledge
  - View objects are a visual representation of the model
  - Controller objects link model objects with their views

Motivation

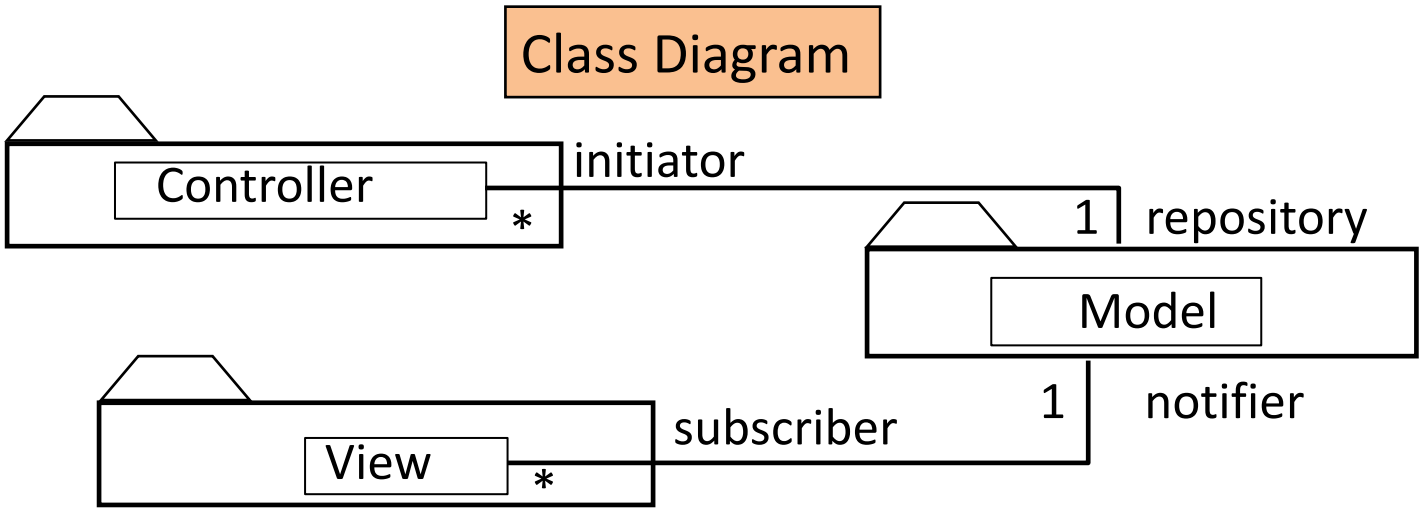
- **Problem:** In systems with high coupling, changes to the user interface (boundary objects) often force changes to the entity objects (data)
  - The user interface cannot be reimplemented without changing the representation of the entity objects
  - The entity objects cannot be reorganized without changing the user interface
- **Solution: Decoupling!** The model-view-controller pattern decouples data access (entity objects) and data presentation (boundary objects)
  - The Data Presentation subsystem is called the **View**
  - The Data Access subsystem is called the **Model**
  - The **Controller** subsystem mediates between View (data presentation) and Model (data access)
- Often called **MVC**

Three types of subsystems

Model subsystem: Responsible for application domain knowledge

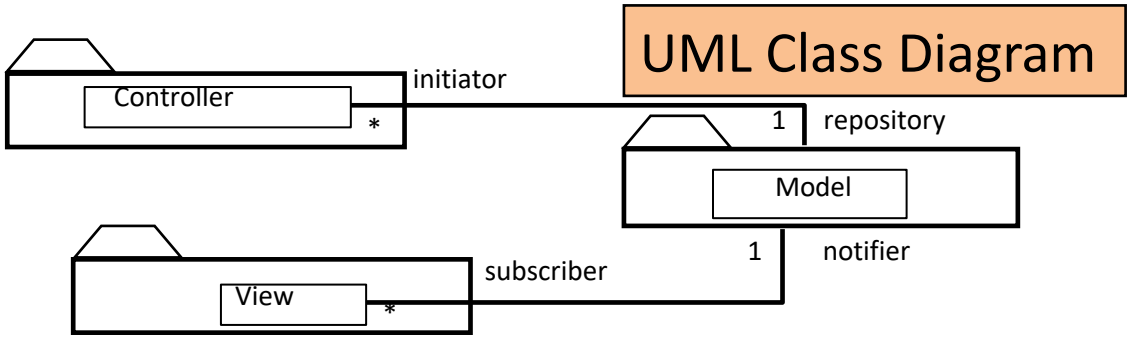
View subsystem: Responsible for displaying application domain objects to the user

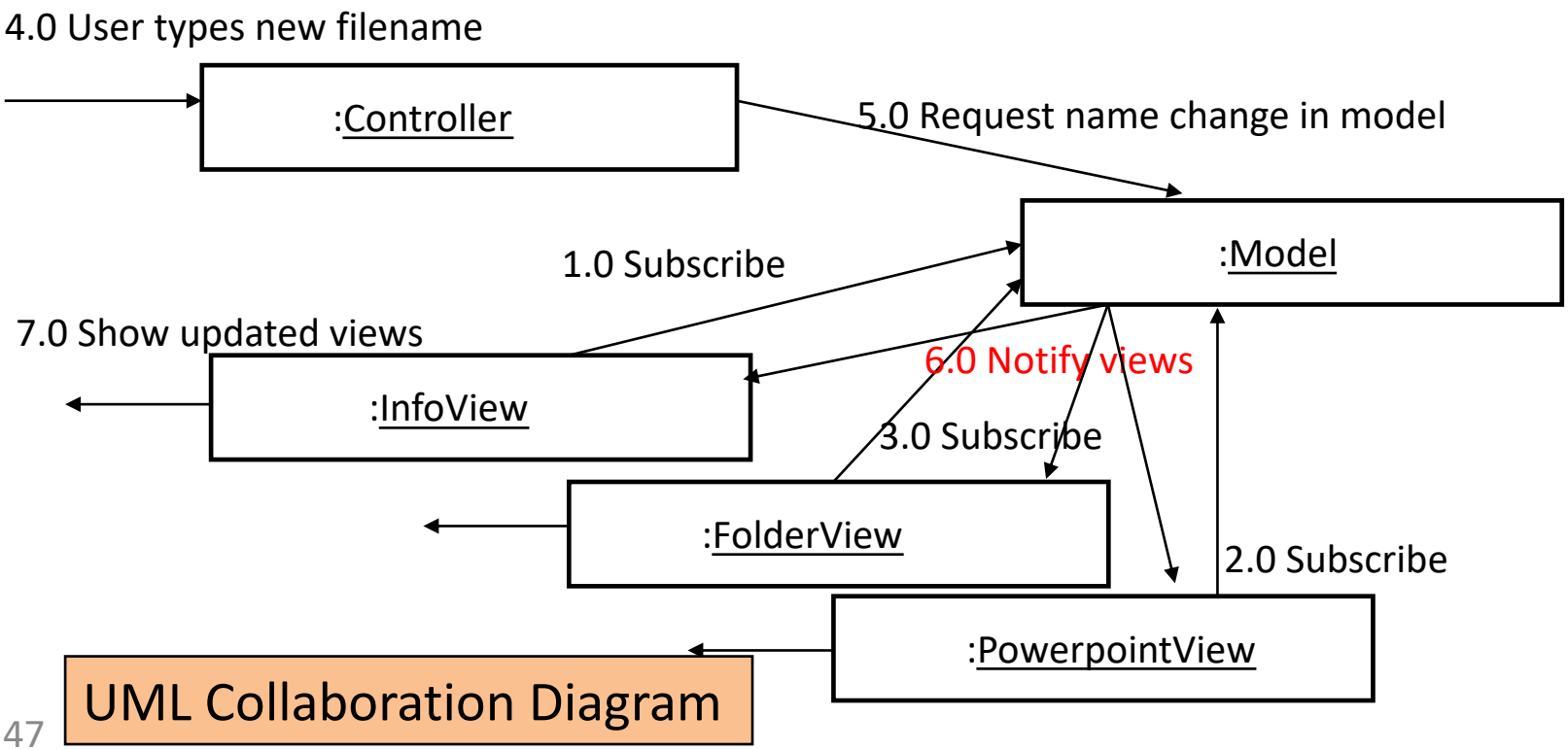
Controller subsystem: Responsible for sequence of interactions with the user and notifying views of changes in the model



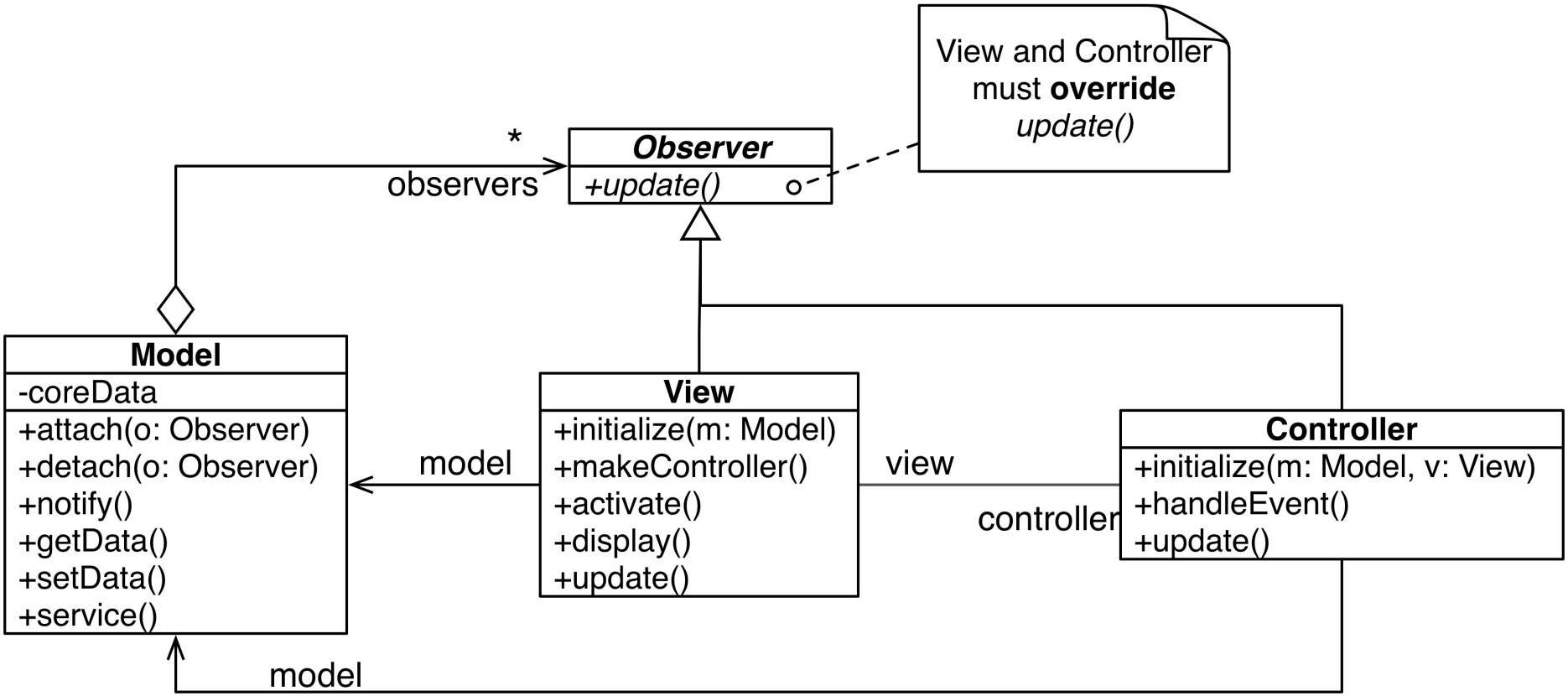
Better understanding with a Collaboration Diagram

Modeling the sequence of events in MVC





Model view controller (UML class diagram)



Adapted from: [Buschmann et. al. 1996]

Usage of other patterns in MVC

- Has deep impact on system design
- Also called a compound pattern, because it often uses other patterns:
  - Strategy Pattern: Controller follows a strategy on how the model's change affects the views
  - Composite Pattern: Views are often composite structures
  - Factory Pattern: In Smalltalk, the View can create the Controller via a factory method
  - Observer Pattern: Views (and possibly Controllers) register for updates from the Model
  - Mediator Pattern: A pattern that allows a realization of MVC where the model and its views don't know each other at all
    - The model and the views just know the connecting controller that "mediates"

MVC implementations in frameworks

- Cocoa Appkit
  - http://developer.apple.com/mac/library/DOCUMENTATION/Cocoa/Conceptual/CocoaFundamentals/CocoaDesignPatterns/CocoaDesignPatterns.html
- Eclipse Standard Widget Toolkit (SWT)
  - http://www.eclipse.org/swt/
- Java Swing (not really MVC, but inspired by it)
  - http://java.sun.com/developer/technicalArticles/javase/mvc/
- PureMVC
  - http://puremvc.org/
- ASP.net MVC
  - http://www.asp.net/learn/mvc/#MVC_Overview

Readings

E.W. Dijkstra (1968)

- The structure of the T.H.E Multiprogramming system, CACM,18(8),pp. 453-457,1968

Also: http://en.wikipedia.org/wiki/THE_multiprogramming_system

D. Parnas (1972)

- On the criteria to be used in decomposing systems into modules, CACM, 15(12), pp. 1053-1058, 1972

R. Kahn and V. Cerf, 1974

- A Protocol for Packet Network Intercommunication

http://www.cs.princeton.edu/courses/archive/fall06/cos561/papers/cerf74.pdf

V. Lesser, R. Fennell, L. Erman and R. Reddy (1975)

- Organization of the Hearsay-II Speech Understanding System, IEEE Trans. on Acoustics, Speech & Signal Processing, Vol. ASSP-23, Nr 1, pp.11–24, 1975

ftp://mas.cs.umass.edu/pub/lesser/LesserIEEE_75.pdf

J. Coutaz (1987)

- PAC, An Object-Oriented model for Dialog Design, Human-Computer Interaction, Proceedings of the Interact 87 conference, pp. 431-436, Elsevier, 1987

Readings ctd

D.E. Perry and A.L. Wolf (1992)

- Foundations for the study of software architecture, ACM SIGSOFT Software Engineering Notes, 17(4), pp. 40-52, 1992

M. Shaw and D. Garlan (1996)

- Software Architecture – Perspectives on an Emerging Discipline, Prentice Hall, 1996

F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal

- Pattern-Oriented Software Architecture: A System of Patterns, John Wiley & Sons, 1996

IEEE Standard 1471-2000 (2000)

- Recommended Practice for Architecture Description of Software-Intensive Systems
  http://standards.ieee.org/reading/ieee/std_public/description/se/1471-2000_desc.html