Patterns in Software Engineering

04 Architectural Patterns I

Stephan Krusche

21 November 2022
Technical University of Munich

Course schedule

#	Date	Subject
	17.10.22	No lecture, repetition week (self-study)
1	24.10.22	Introduction
	31.10.22	No lecture, repetition week (self-study)
2	07.11.22	Design Patterns I
3	14.11.22	Design Patterns II
4	**21.11.22**	**Architectural Patterns I**
5	28.11.22	Architectural Patterns II
6	05.12.22	Antipatterns I
7	12.12.22	Antipatterns II
	19.12.22	**No lecture**
8	09.01.23	Testing Patterns I
9	16.01.23	Testing Patterns II
10	23.01.23	Microservice Patterns I
11	30.01.23	Microservice Patterns II
12	08.02.21	Course Review

Roadmap of the lecture

- **Context and assumptions**
  - You have understood the basic concepts of patterns
  - You have implemented many different design patterns
- **Learning goals: at the end of this lecture you are able to**
  - Differentiate between the covered architectural patterns
  - Choose the right architectural style
  - Model the application of an architectural pattern

Terminology: architecture style vs. architectural pattern

- Buschmann et al. define
  - Architecture style: overall structural frameworks for applications
  - Architectural pattern: basic structure of an application
    - Styles are independent from each other, patterns depend on smaller patterns
    - Patterns are more problem oriented than styles

- PSE definition
  - Architecture style = architectural pattern: a pattern for a subsystem decomposition
  - Software architecture: instance of an architectural pattern

F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal, Pattern-Oriented Software Architecture: A System of Patterns, John Wiley & Sons, 1996

What is a 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
- Reuse of established engineering knowledge
  - Application of architectural patterns
  - Architectural patterns → system design
  - Design patterns → object design

How do real architects work?

- An architectural pattern defines the main components and their layout
- The architect picks an architectural pattern based on customer requirements
- The architect changes details according to customer requirements
- Reuse and adaption of established engineering knowledge

→ We apply the same approach to software architecture

Ranch style

T-Ranch style

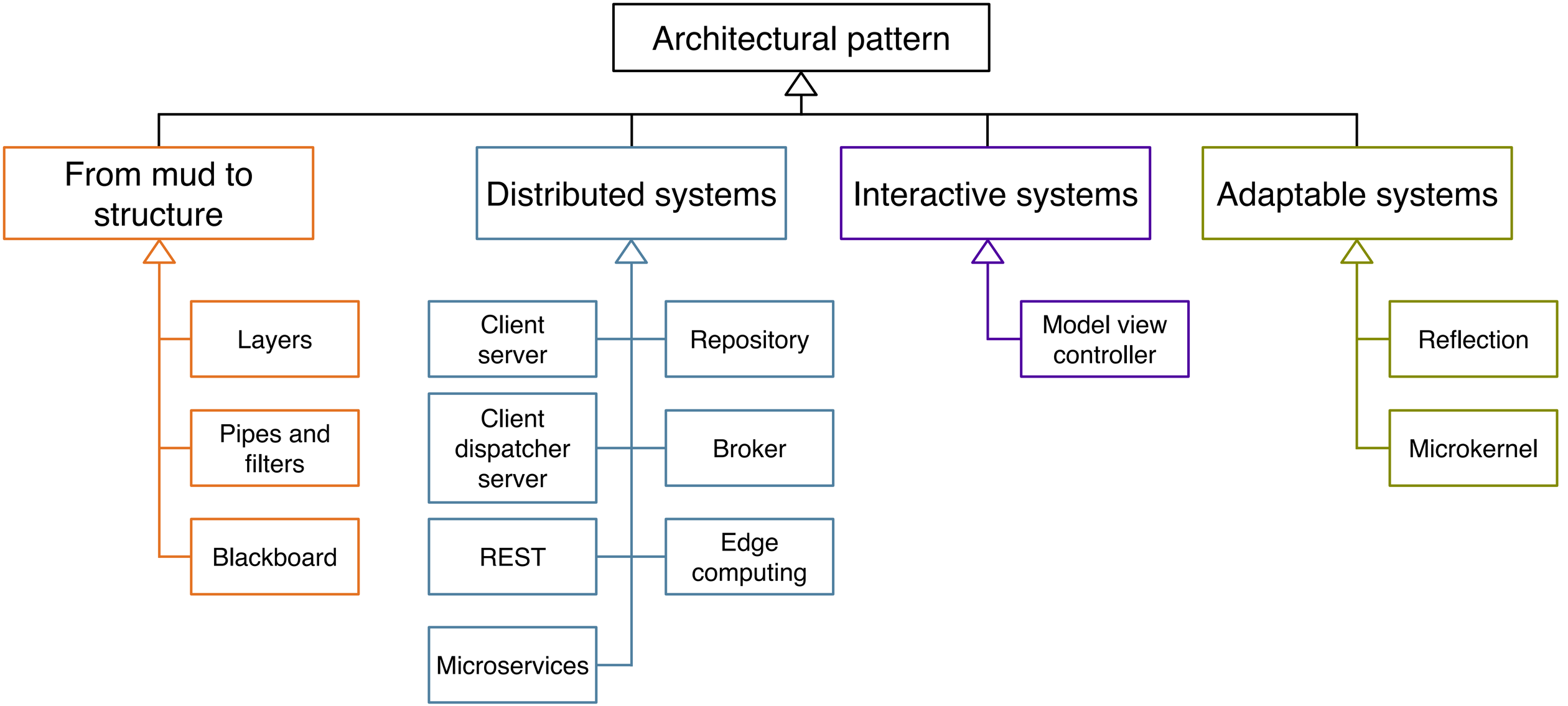
Raised ranch style

Elements of a software architecture

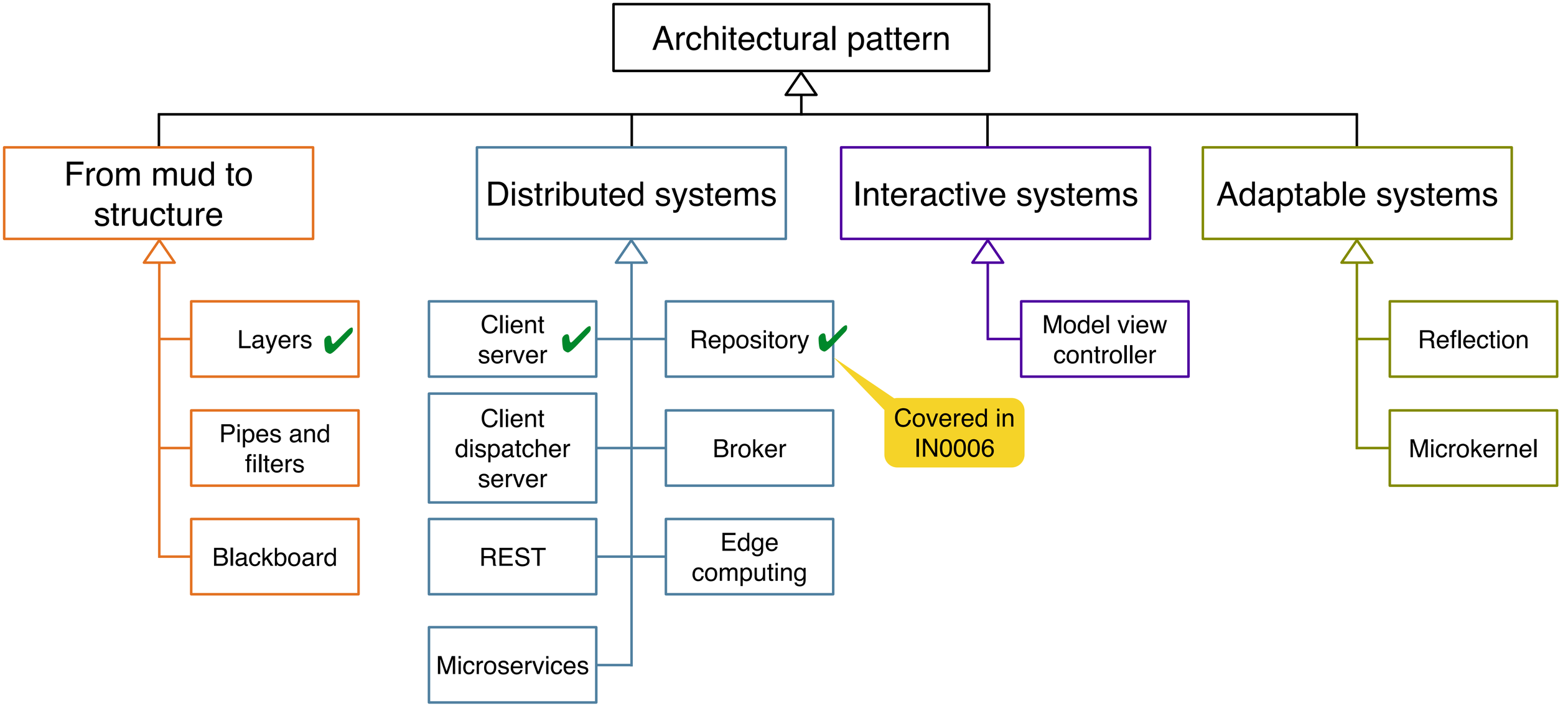
- A software architecture consists of **components** and **connectors**
- Components (subsystems)
  - Computational units with specified interfaces
  - Examples: filters, databases, layers, objects
- Connectors (communication)
  - Interactions between the components (subsystems)
  - Examples: method calls, pipes, event broadcasts, shared data

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

Architectural pattern overview



Architectural pattern overview



Architecture pattern overview (1)

- **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 connected by pipes
  - **Blackboard:** subsystems are knowledge experts working together to solve a problem without a known solution strategy
- **Distributed systems:** collaborating systems on different nodes
  - **Repository:** exchange of persistent data between multiple clients
  - **Client server:** clients interact directly with servers
  - **Client dispatcher server:** clients interact with servers via an additional name server
  - **Broker:** systems interact via remote service invocations
  - **REST:** representational state transfer, often used for web services, based on client server, layers and proxy
  - **Edge computing:** synchronization of data, real-time access, and availability are realized simultaneously
  - **Microservices:** system consists of many small services

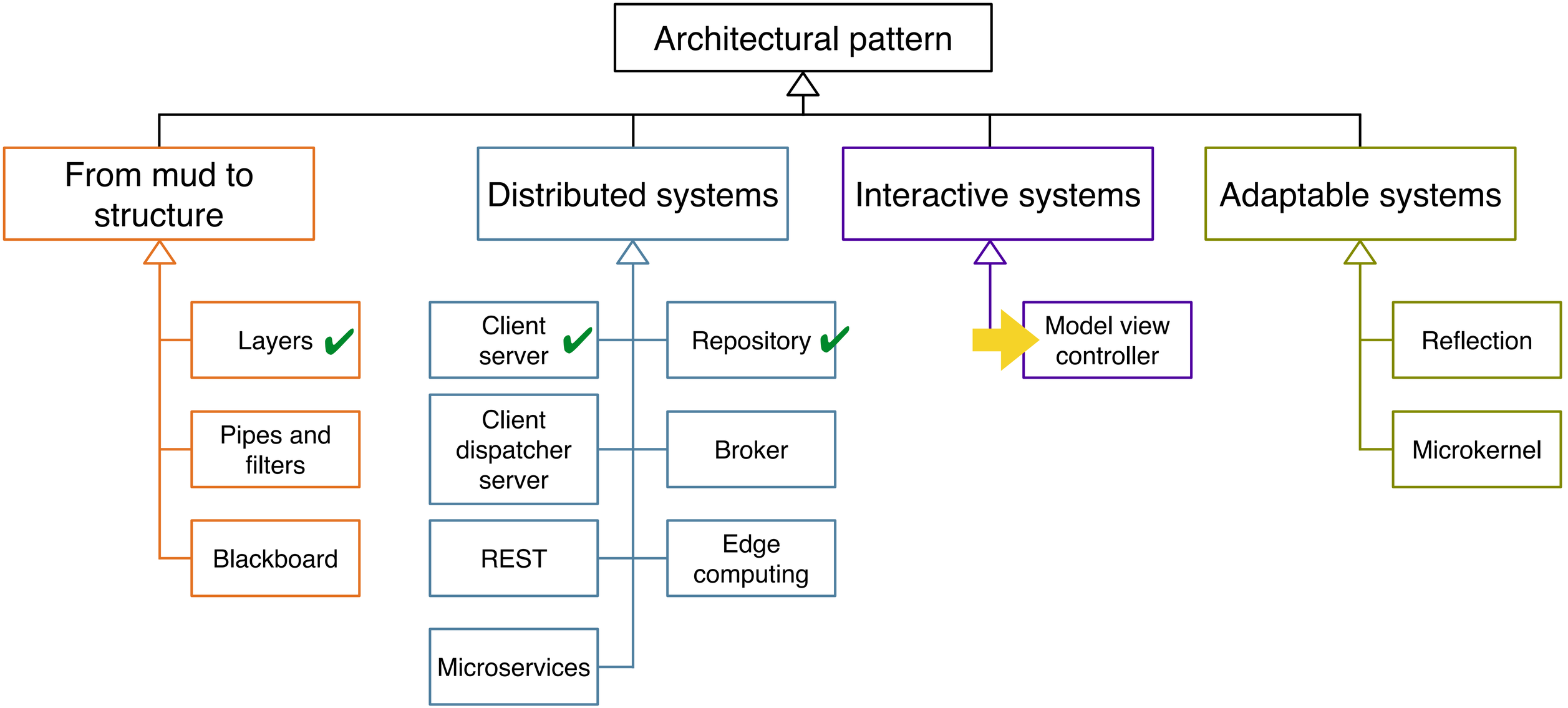
Architecture pattern overview (2)

- **Interactive systems:** systems interacting with a user
  - **Model view controller:** subsystem decomposition with 3 components: model (entity objects), view (boundary objects), controller (controller objects)
- **Adaptable systems:** systems evolving over time
  - **Reflection:** self awareness provides information about system properties
  - **Microkernel:** extensible minimal functional core

Outline

- Model view controller (part 1)
- Model view controller (part 2)
- Client dispatcher server
- Blackboard pattern

Architectural pattern overview

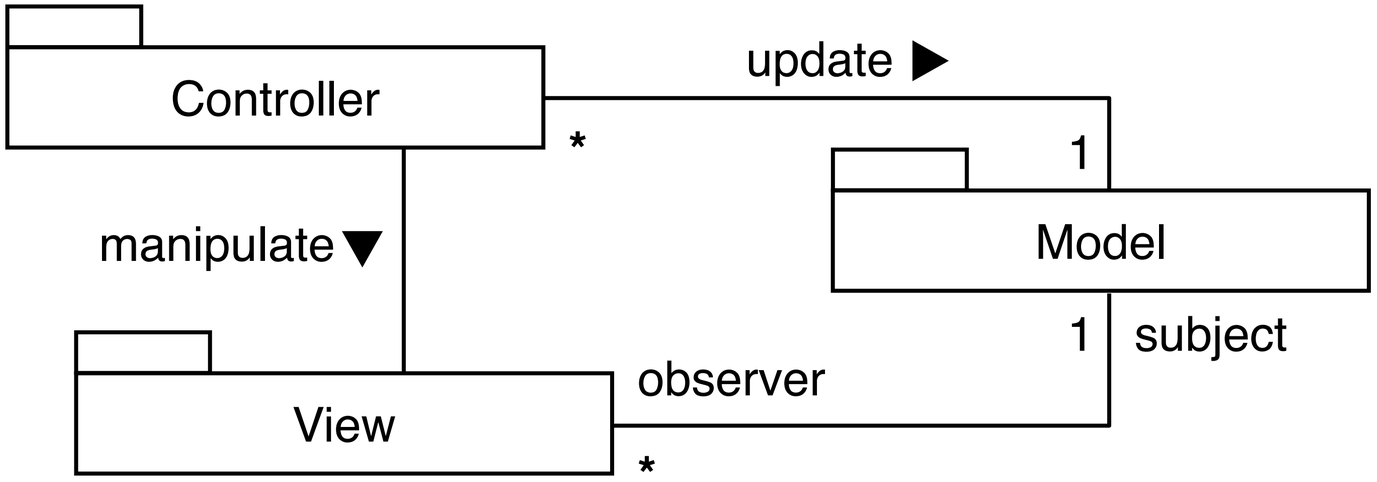


Model view controller (MVC)

- **Problem:** in systems with high coupling any change to the boundary objects (user interface) often forces changes to the entity objects (data)
  - The user interface cannot be re-implemented without changing the representation of the entity objects
  - The entity objects cannot be reorganized without changing the user interface
- **Solution:** decouple data access (entity objects) and data presentation (boundary objects)
  - Separation of concerns
  - Improved testability and extensibility

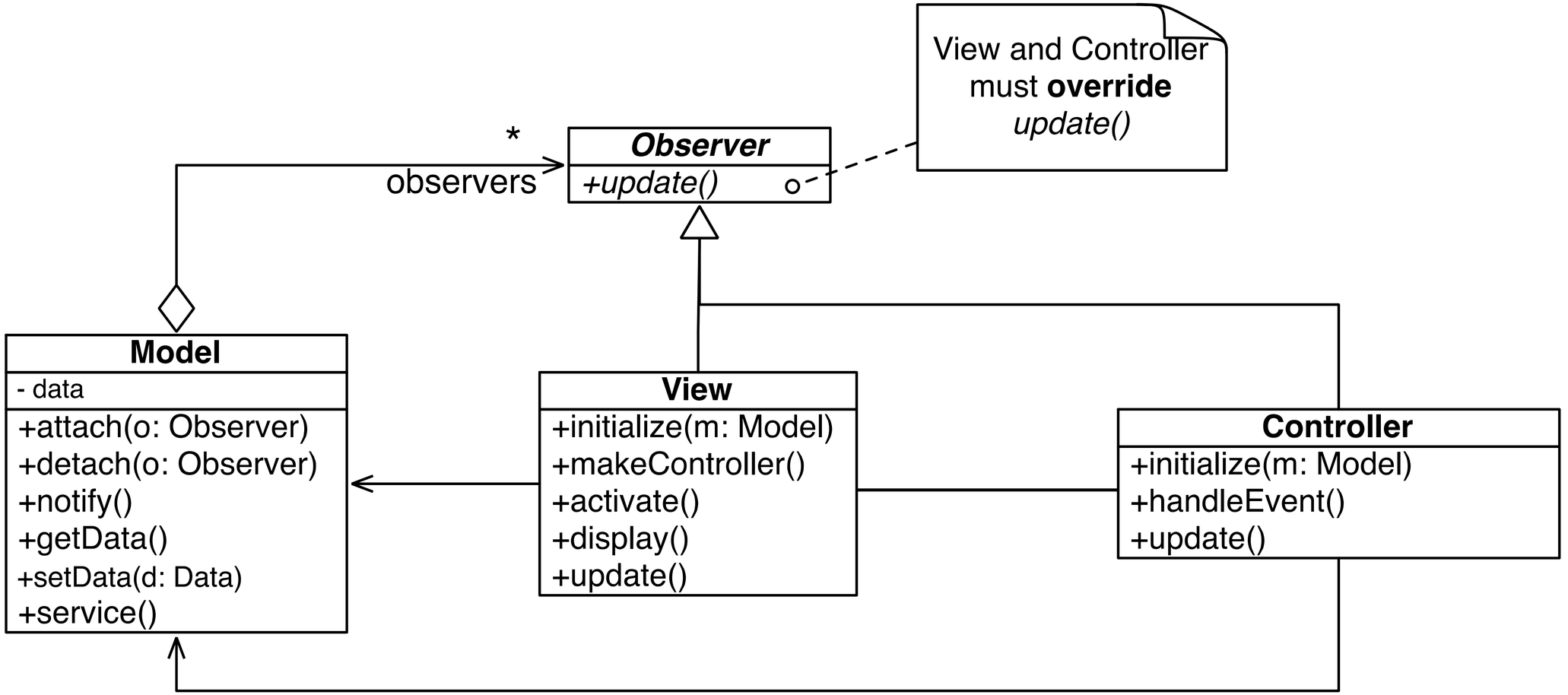
Model view controller

- **Model:** process and store application domain data (entity objects)
- **View:** display information to the user (boundary objects)
- **Controller:** interact with the user and update the model (which notifies the view)
  - **Important:** view and controller together comprise the user interface
  - A change propagation mechanism ensures consistency between the user interface(s) and the model



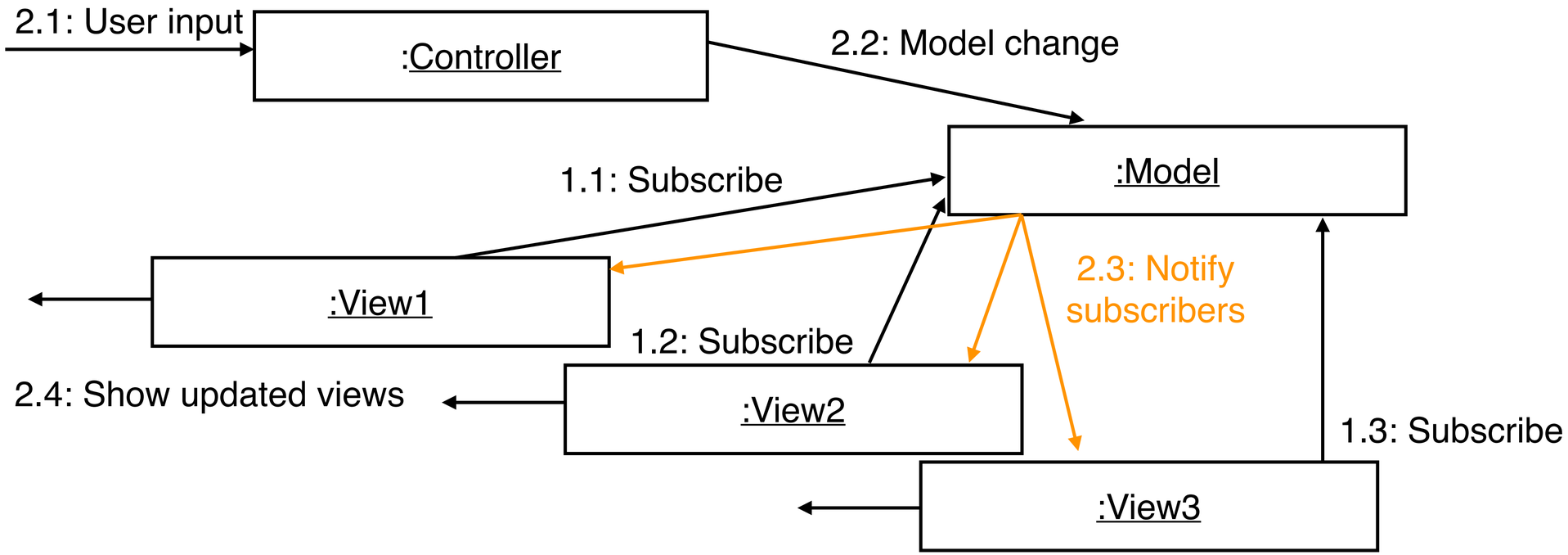
Note: informal UML class diagram: packages represent multiple classes in subsystems here. Associations between packages represent the associations between classes in these packages

Model view controller



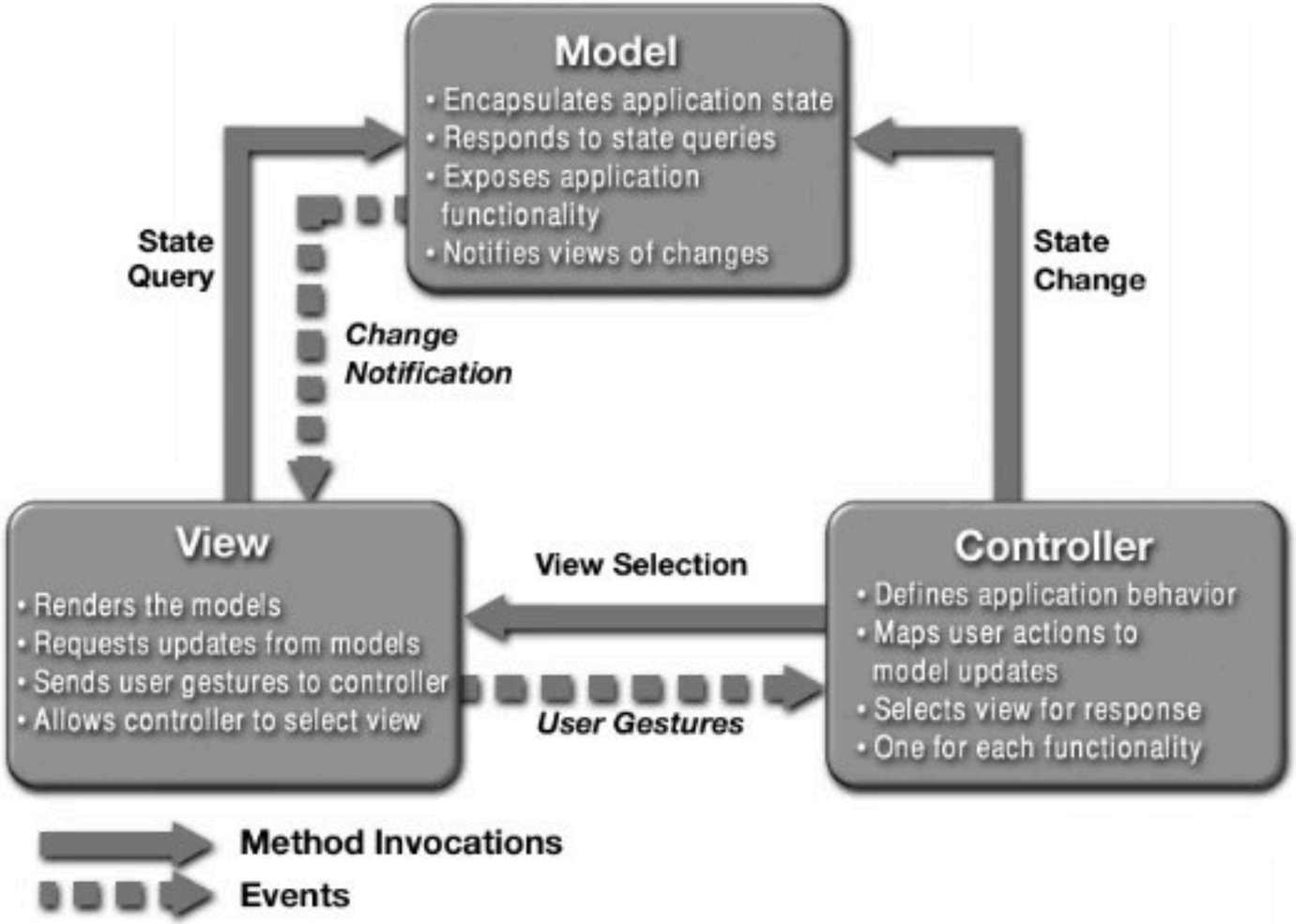
adapted from: [Buschmann et. al. 1996]

Dynamic model



Note: Informal UML communication diagram: messages are natural language, methods in objects are missing

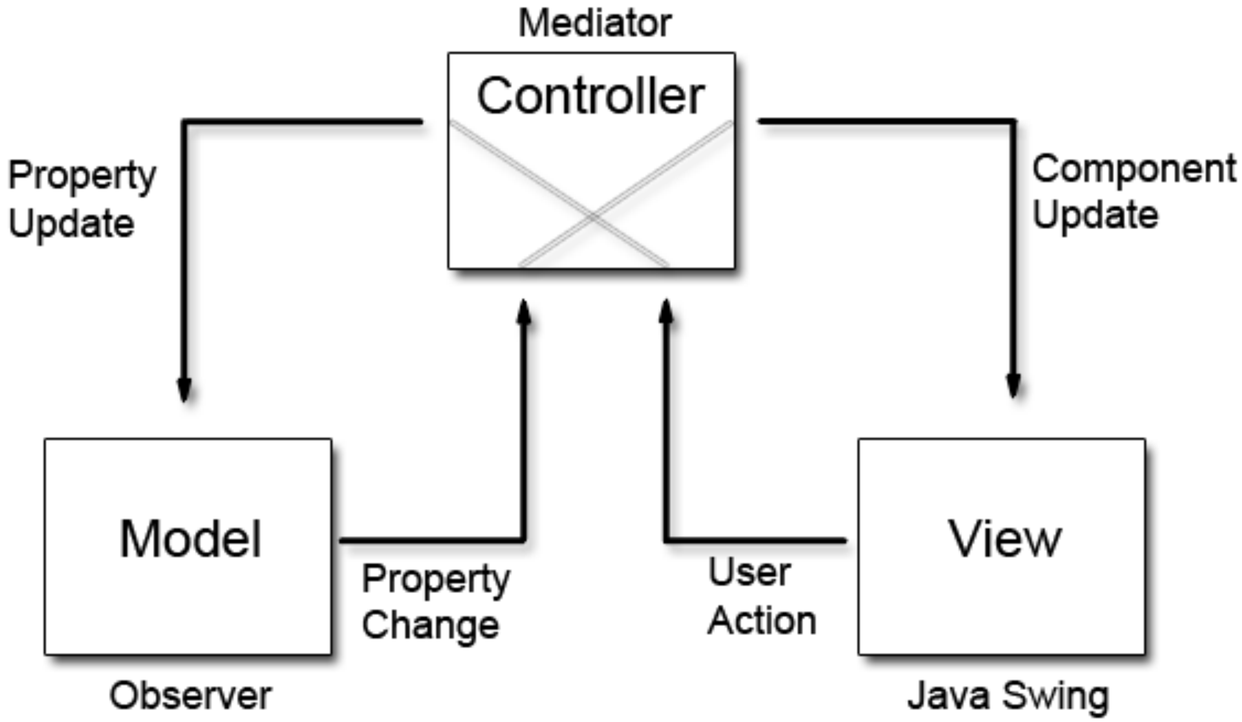
Common MVC implementation



https://www.oracle.com/technical-resources/articles/javase/application-design-with-mvc.html

Modified MVC

- A more recent implementation of the MVC design places the controller between the model and the view.
- This design is e.g. common in the Apple Cocoa framework



https://www.oracle.com/technical-resources/articles/javase/application-design-with-mvc.html

L04E01 Model View Controller 1

Not started yet.

Start exercise

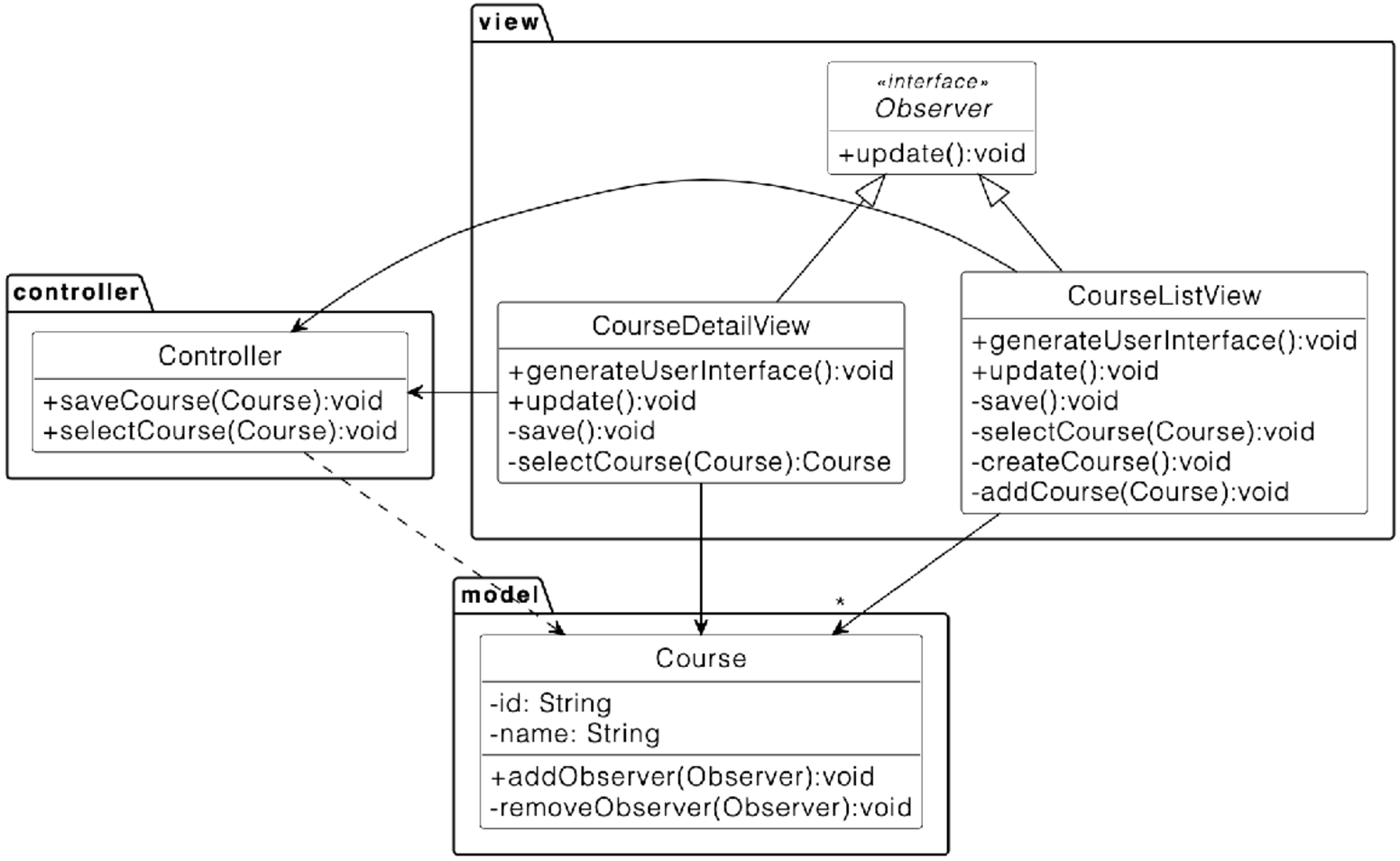
Easy

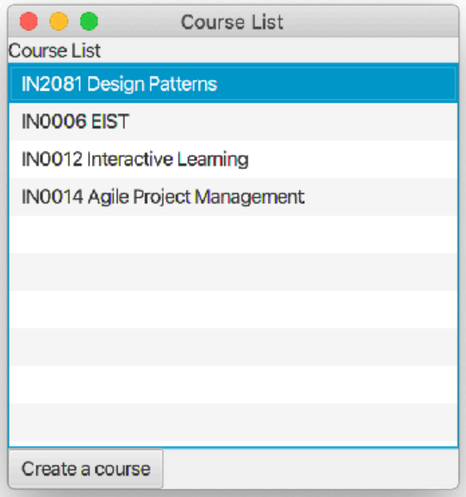
Due date in 7 days

15 min

5 pts

- Problem statement: course management



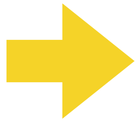


Outline

- Model view controller (part 1)
- Model view controller (part 2)
- Client dispatcher server
- Blackboard pattern

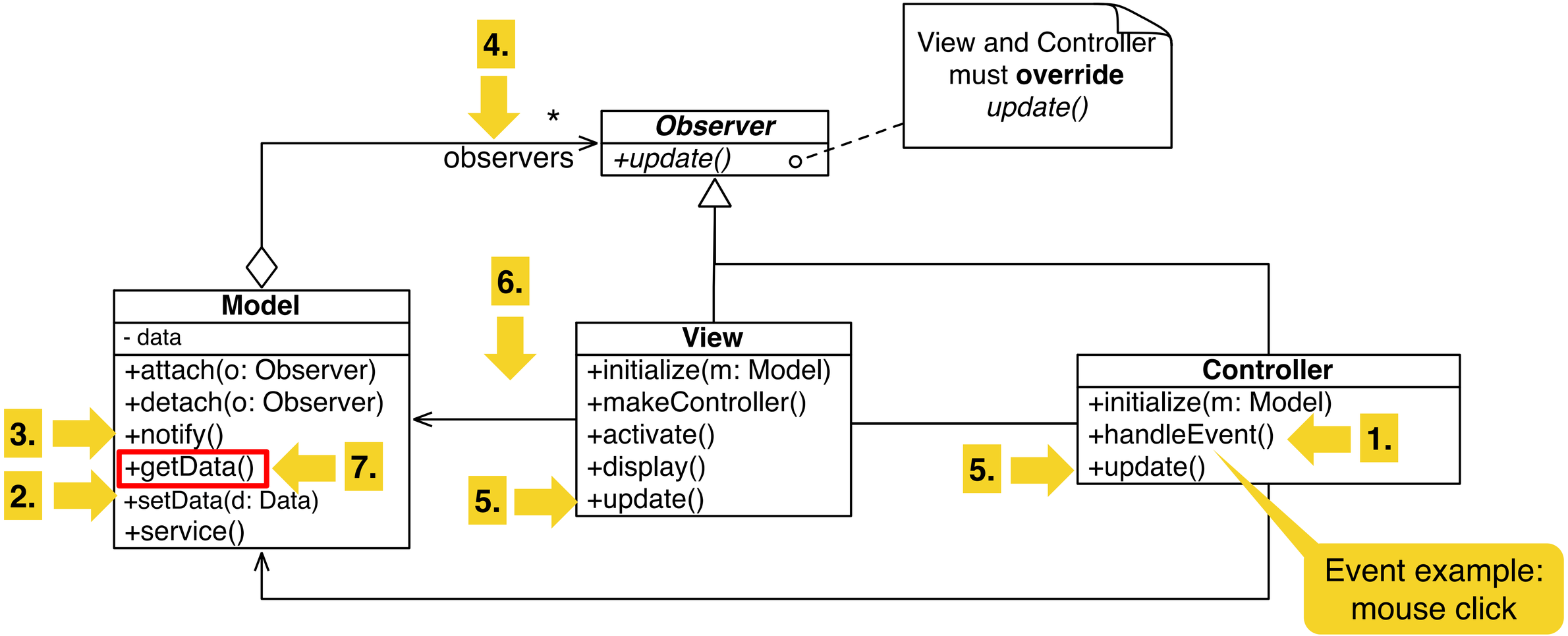
Instantiating the model view controller architectural style

- **Software architecture:** instance of an architectural style
- There are many possibilities to instantiate MVC
- Two choices for the notification



1. Pull notification variant: view and controller obtain the data from the model
2. Push notification variant: the model sends the changed state to view and controller

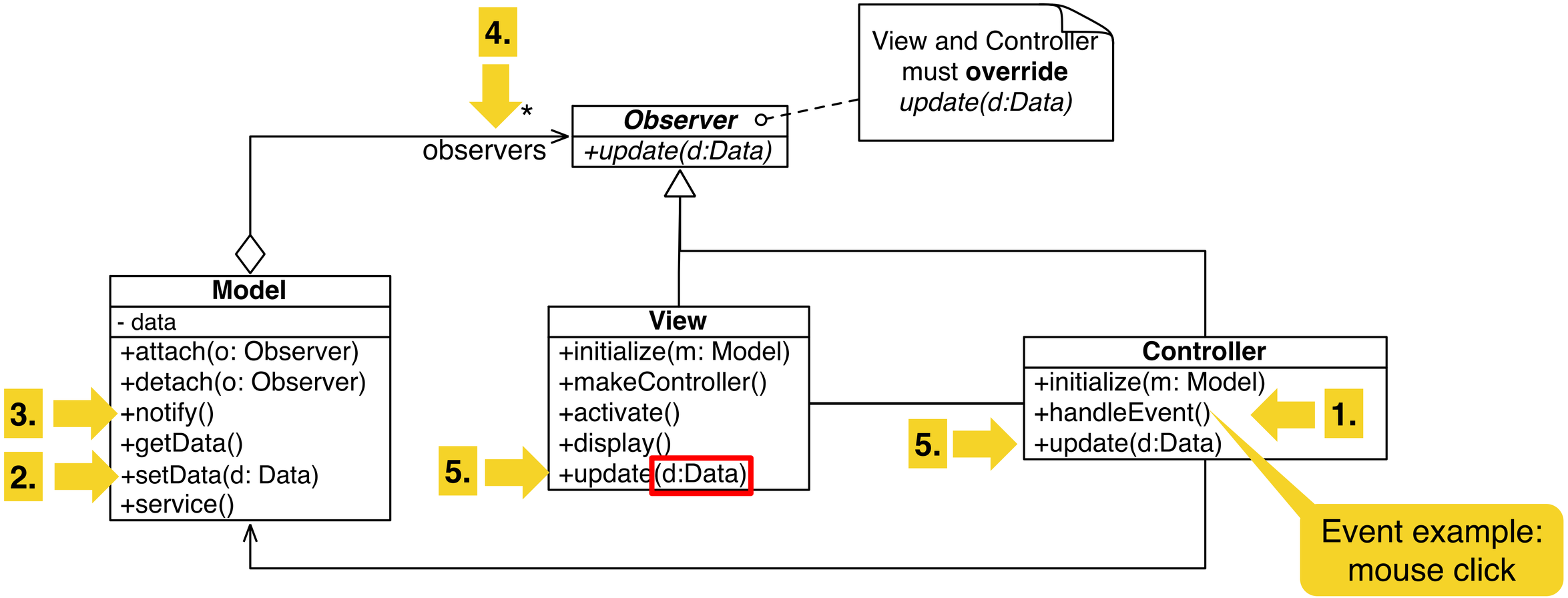
Pull notification variant



Instantiating the model view controller architectural style

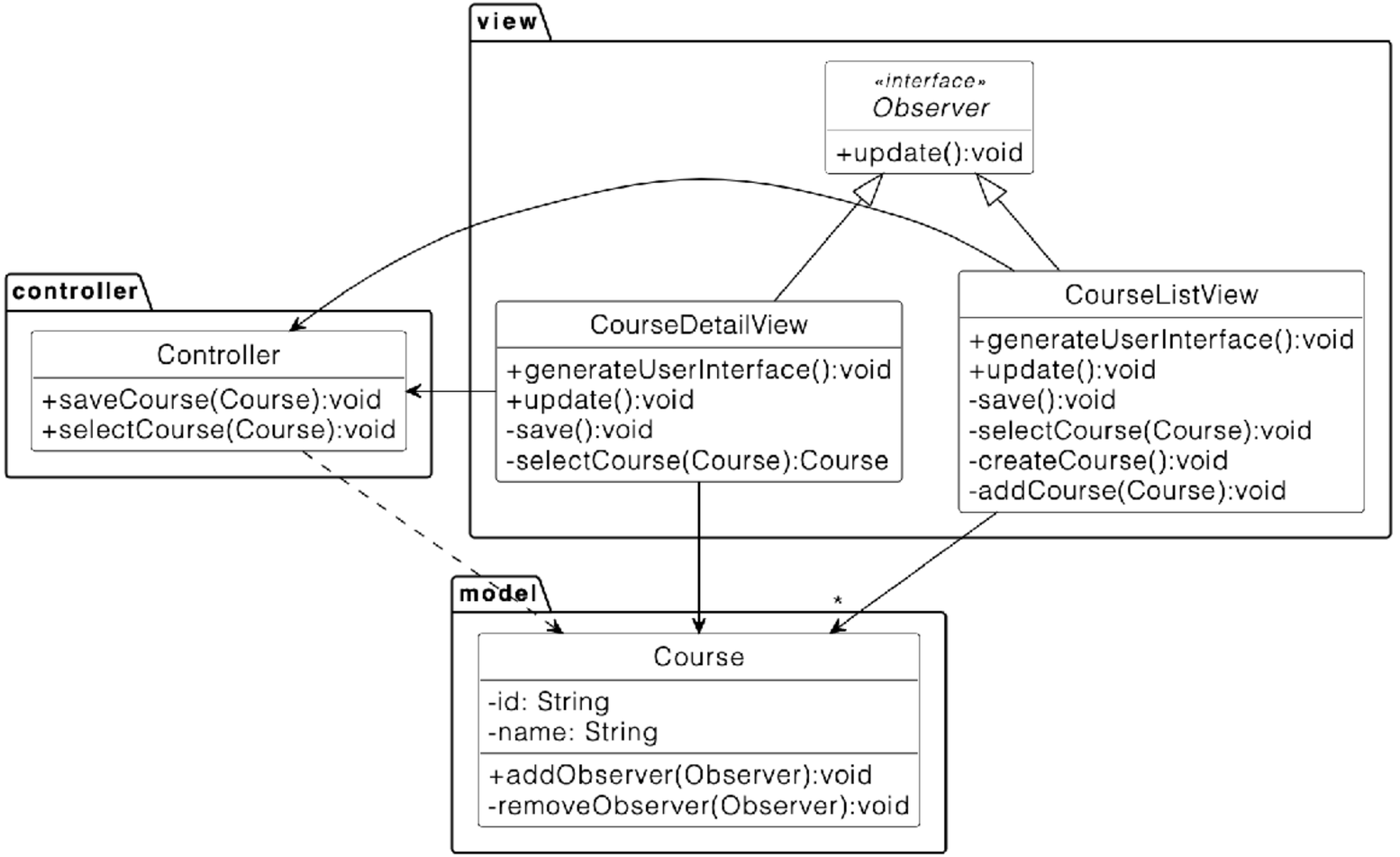
- **Software architecture:** instance of an architectural style
- There are many possibilities to instantiate MVC
- Two choices for the notification
  1. Pull notification variant: view and controller obtain the data from the model
  2. Push notification variant: the model sends the changed state to view and controller

Push notification variant

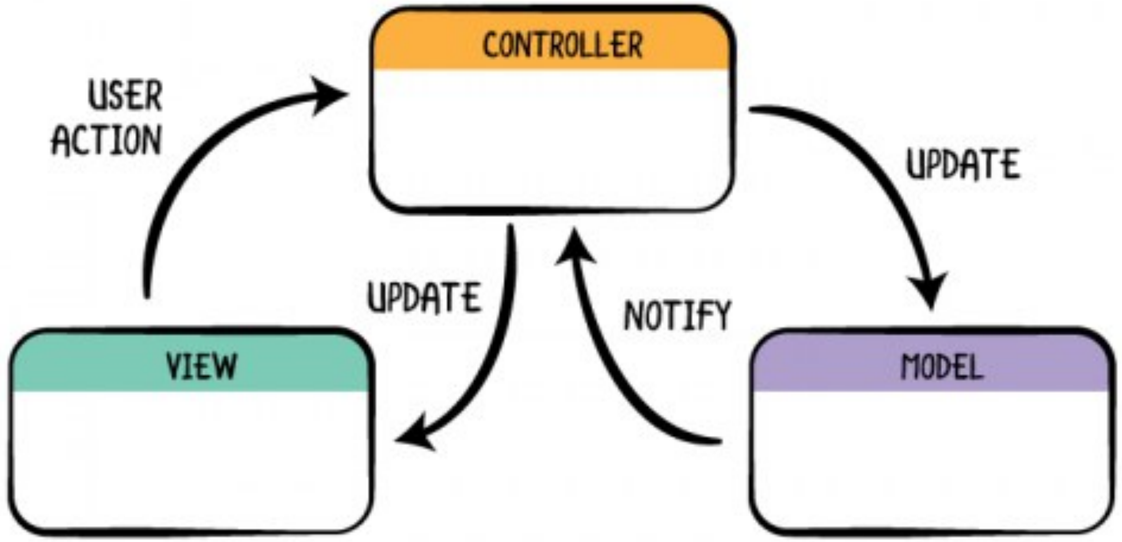


Reflection and review of L04E01

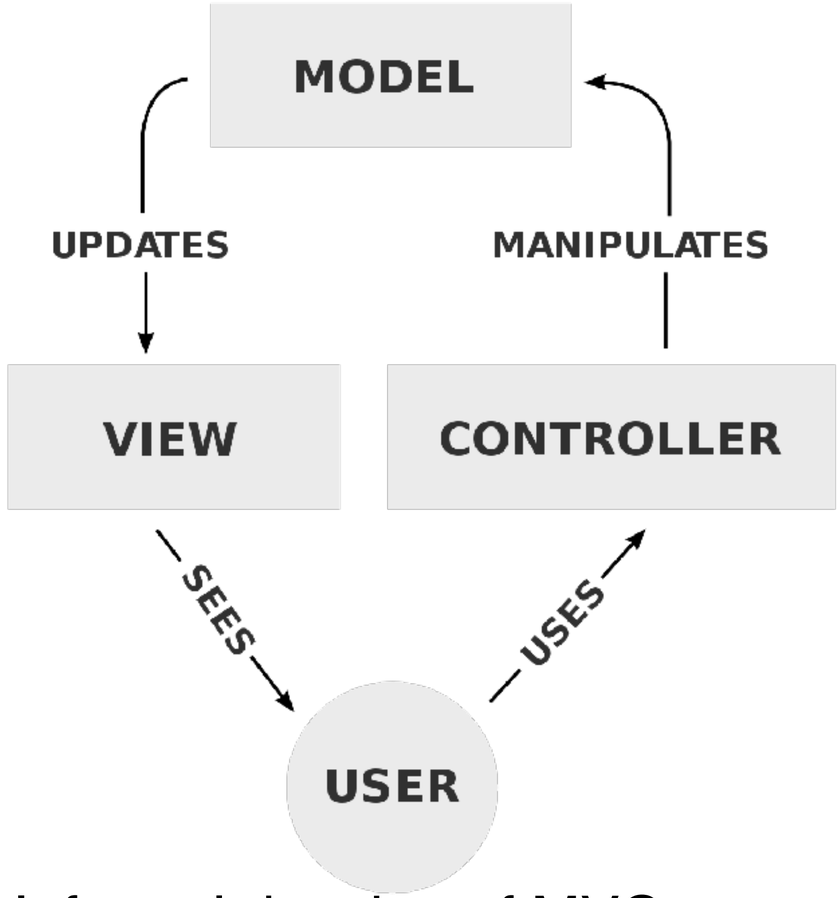
- Which notification style was used?
- How are model, view and controller objects connected?



Common instances of model view controller

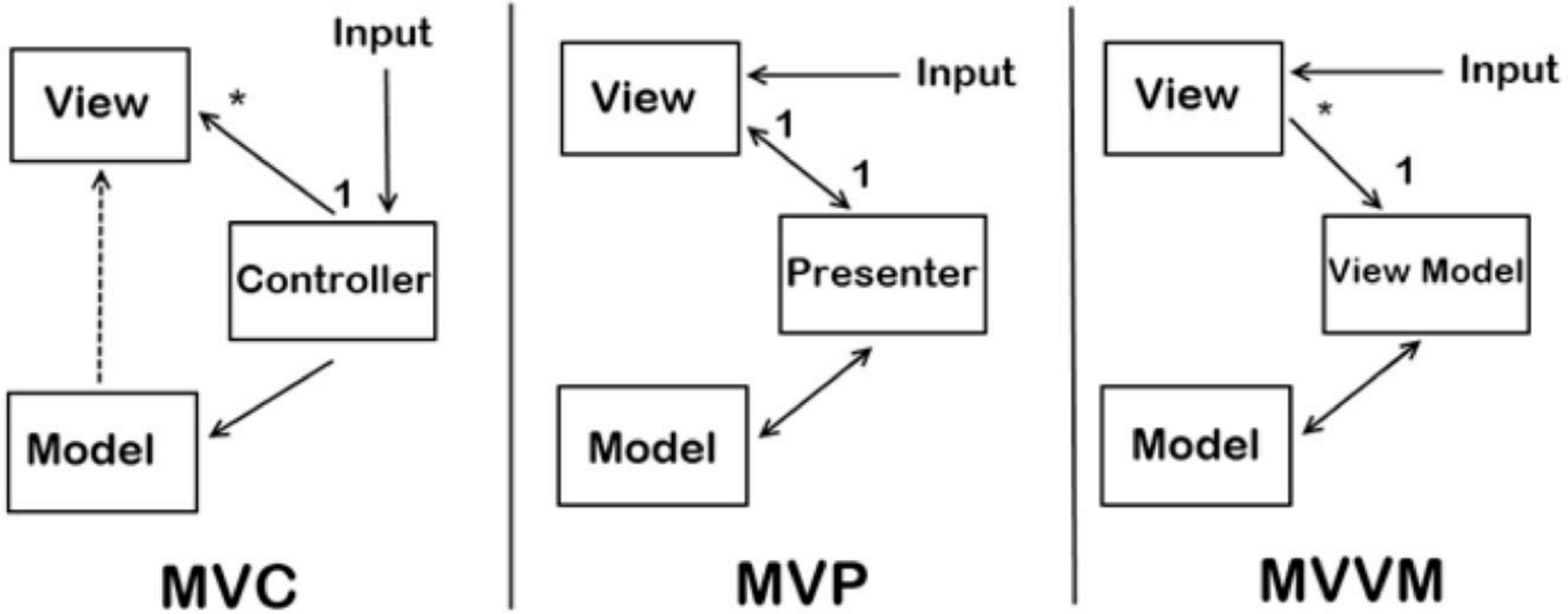


- Informal drawing of MVC in iOS
- Cocoa Touch decouples view and model
- The controller acts as mediator



- Informal drawing of MVC
- Similar to the pull notification variant, but without updating the controller

Other variations of model view controller



Benefits and challenges

- **Benefits**
  - Multiple synchronized views of the same model
  - Pluggable views and controllers
  - Exchangeability of look and feel
  - Framework potential
- **Challenges**
  - Increased complexity
  - Potential for excessive number of updates
  - Close connection between view and controller
  - Close coupling of views and controllers to model

Depends on the used version of MVC

L04E02 Model View Controller 2

Start exercise

Medium

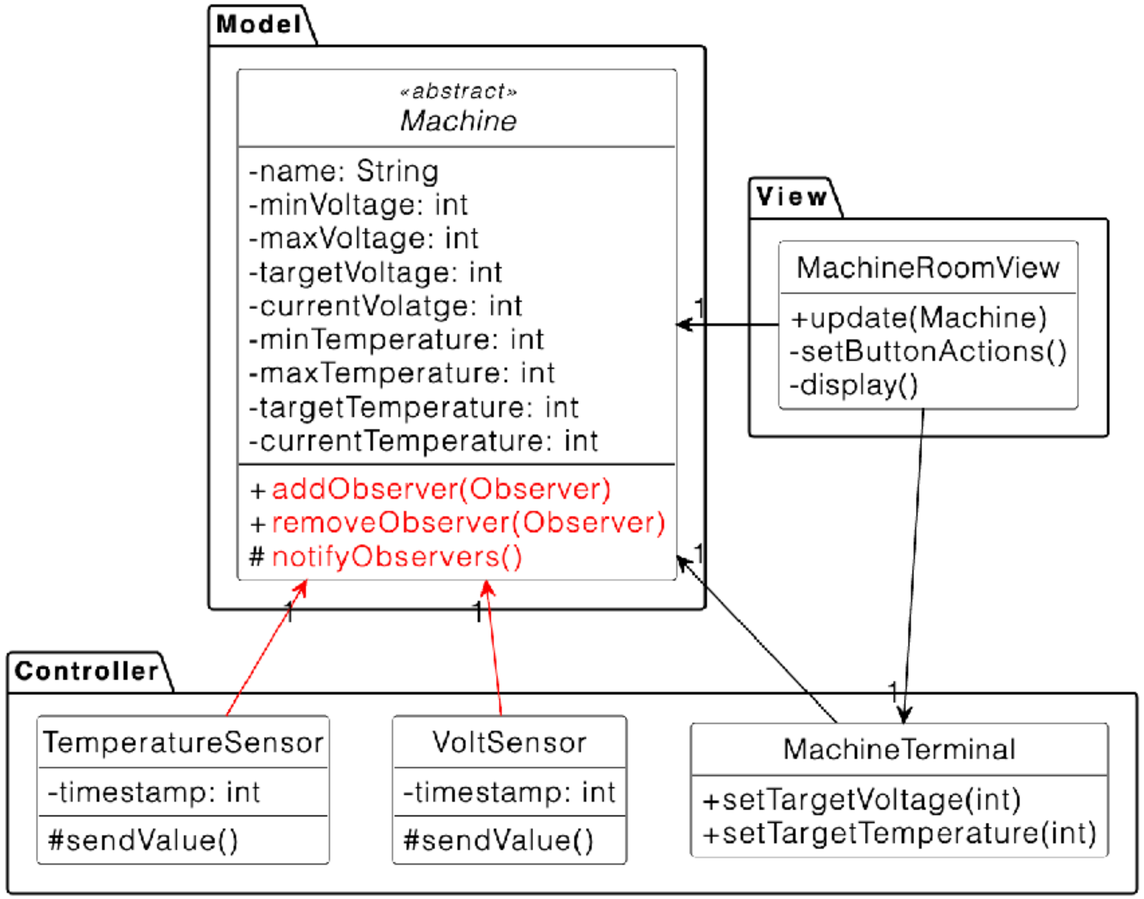
Not started yet.

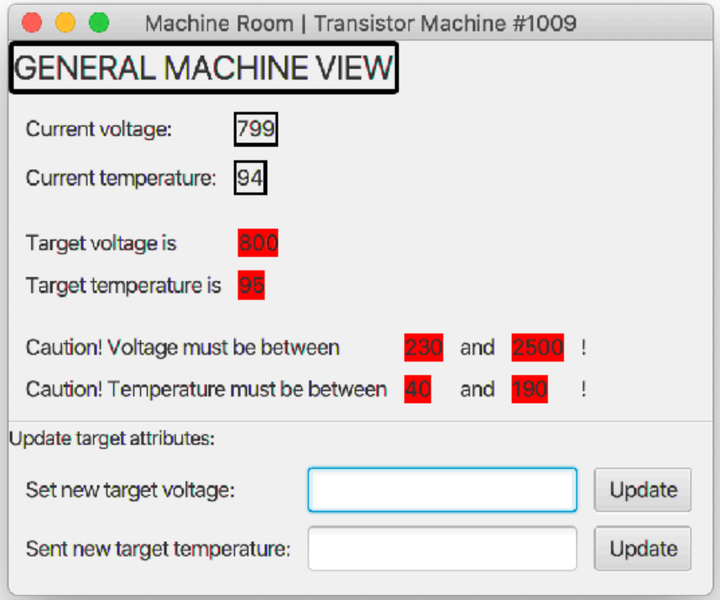
Due date in 7 days

25 min

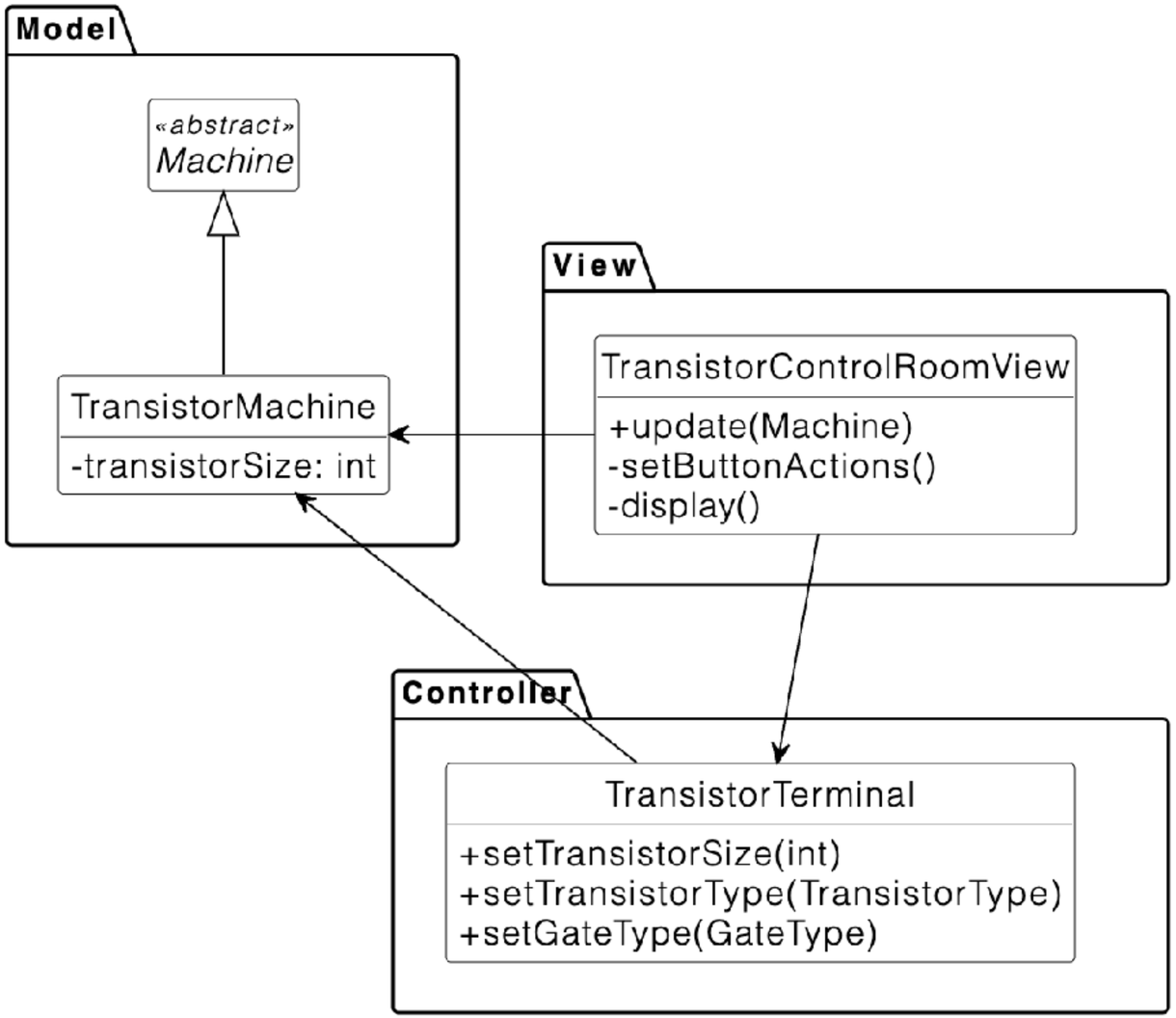
10 pts

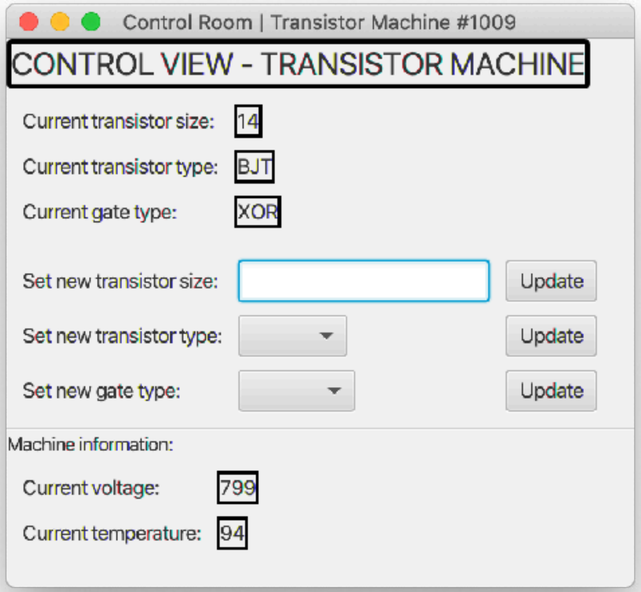
- Problem statement: chip factory



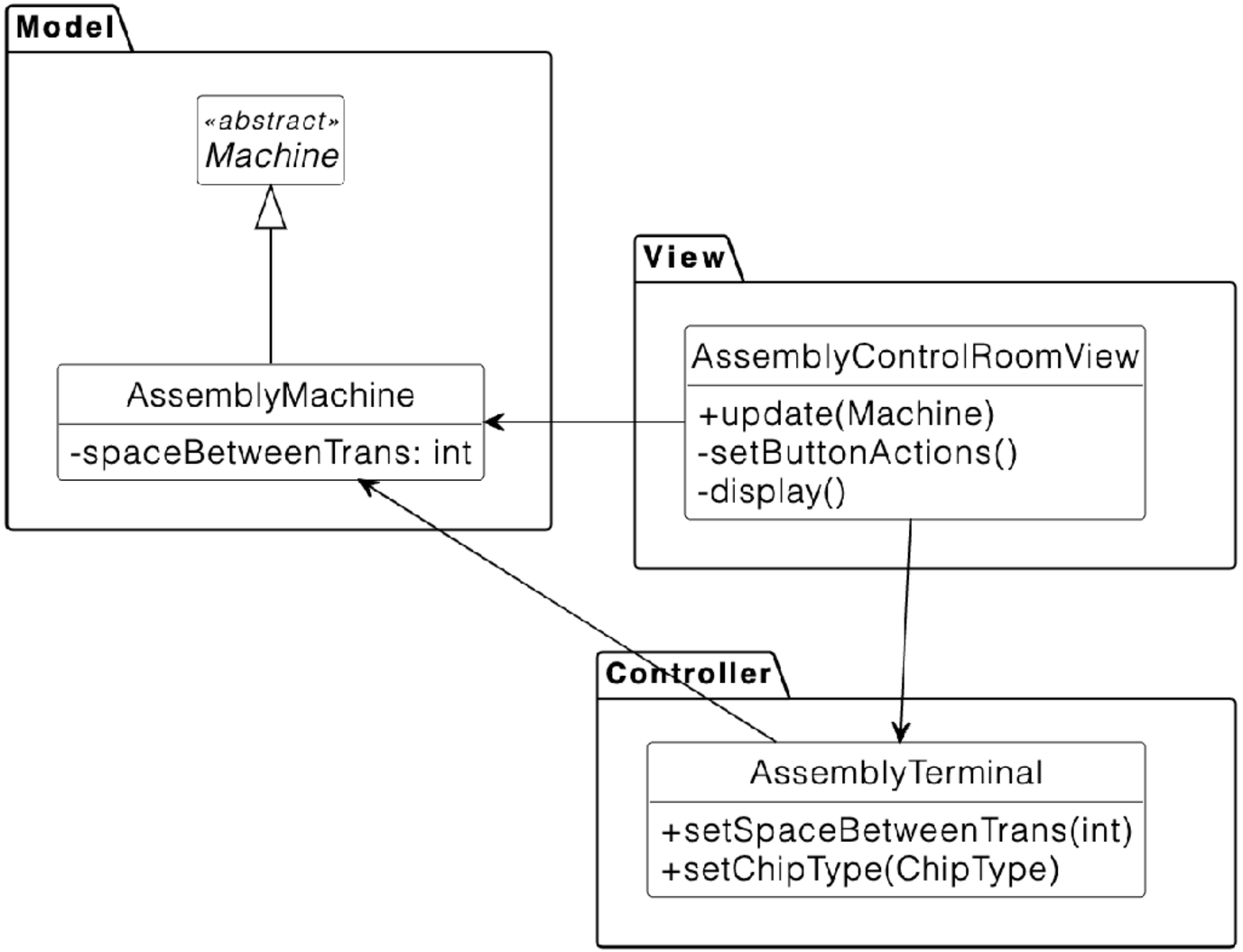


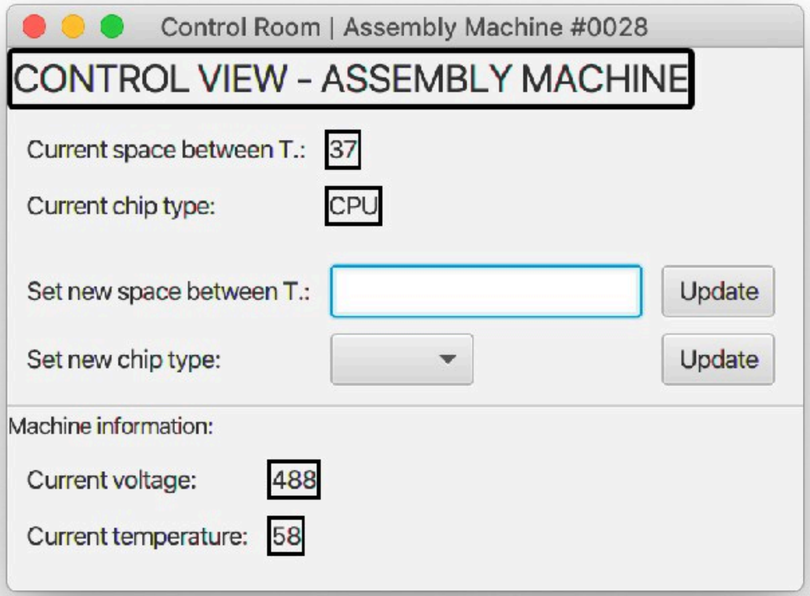
Hint: transistor machine view





Hint: AssemblyControlRoomView





Outline

- Model view controller (part 1)
- Model view controller (part 2)
- Client dispatcher server
- Blackboard pattern

Challenges in distributed systems

- Connectivity
  - Systems are distributed and have to find and communicate with each other

  → Client server, client dispatcher server — Covered in L04
- Heterogeneity
  - The distributed systems use different languages, platforms and protocols

  → Broker — Covered in L05
- Security
  - Access control, authentication and encryption

  → Proxy pattern — Covered in IN0006

Distributed systems: problem and solutions

- Problem: distributed components need to communicate with each other
  - **Solution A** — Client server
    - Components implement the communication mechanism themselves
    - Disadvantage: clients need to know the location of the server
  - **Solution B** — Client dispatcher server
    - Place the communication mechanism in a name server, which decouples clients and servers and sets up the communication between both
    - Addresses the nonfunctional requirement location transparency
  - **Solution C** — Broker
    - Coordinate the communication between nodes
    - Forwarding of requests, transmission of results, handling of exceptions
    - Addresses the nonfunctional requirements location transparency and interoperability

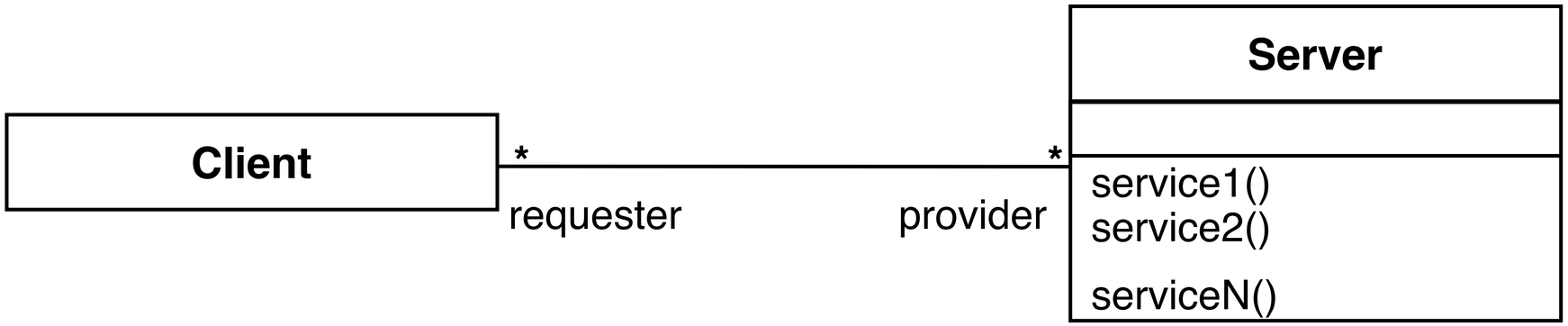
Client server

Small repetition of the concepts taught in IN0006

- **Context:** there are shared resources and services that large numbers of distributed clients wish to access, and for which we wish to control the access or the quality of service
- **Problem:** how can we manage a set of shared resources and services, and still make them modifiable, reusable, allowing scalability and availability, while distributing the resources themselves across multiple physical servers?
- **Solution:** clients interact by requesting services of servers, which provide a set of services: there may be one central server or multiple distributed ones

Client server

- One or more servers provide **services** to clients
- Each client calls a **service** offered by the server
  - The server performs the **service** and returns the result to the client
  - The client knows the **interface** of the server
  - The server does not know the **interface** of the client
- The response is typically **immediate** (i.e. less than a few seconds)
- **End users** interact only with the client



Typical design goals for client server

Portability	Server runs on many operating systems and many networking environments
Location transparency	Server might itself be distributed, but provides a single "logical" service to the user
High performance	Client optimized for interactive display-intensive tasks Server optimized for CPU-intensive operations
Scalability	The server can handle large amounts of clients
Flexibility	The user interface of the client supports a variety of end-devices (phone, laptop, smart watch)
Reliability	Server should be able to survive client and communication problems

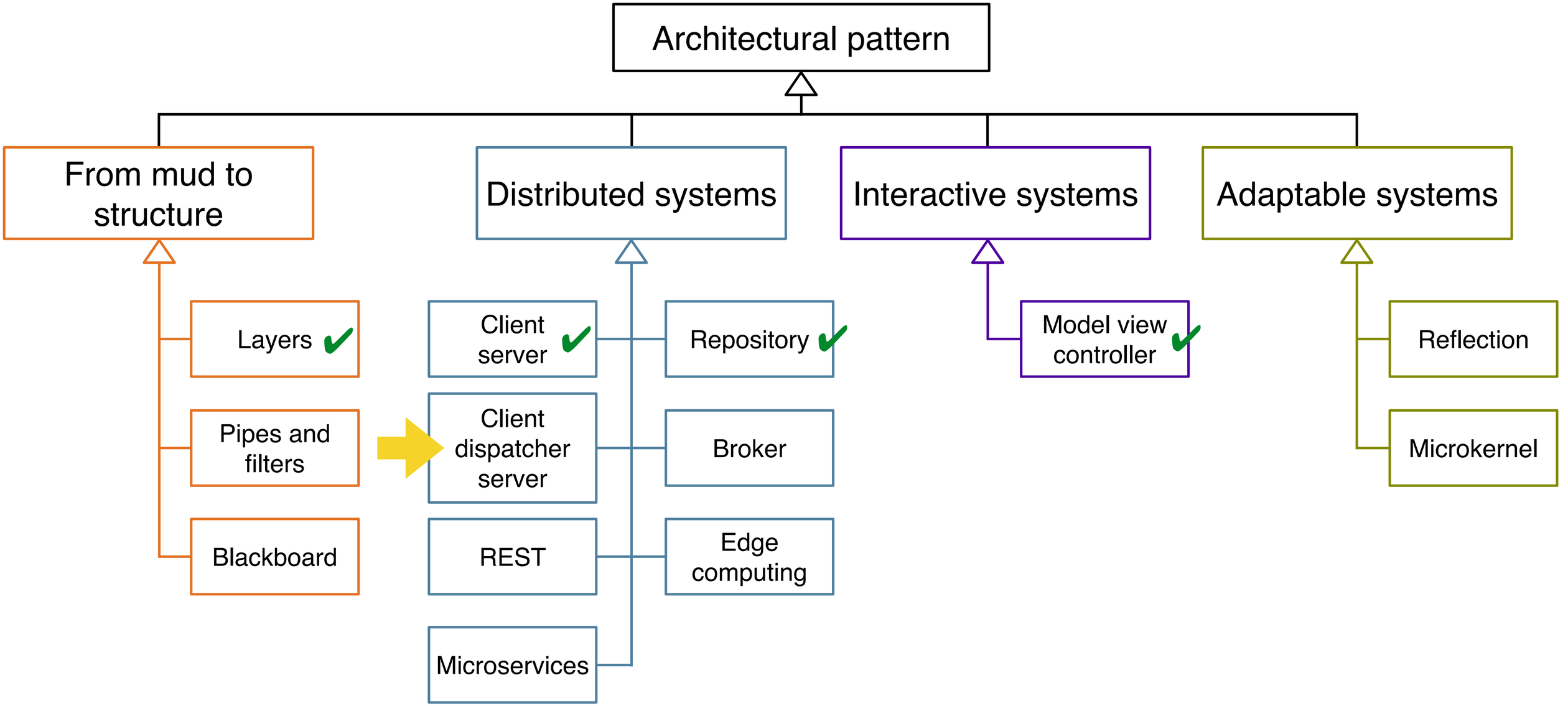
Properties

- Often used in the design of database systems
  - Client: user application
  - Server: database access and manipulation
- Functions performed by the client
  - Input by the user (customized user interface)
  - Sanity checks of input data
- Functions performed by the server
  - Centralized data management
  - Provision of data integrity and database consistency
  - Provision of database security

Limitations

- **Constraints**
  - Clients are connected to servers through request / reply connectors
  - Server components can be clients to other servers
- **Weaknesses**
  - The server can be a performance bottleneck
  - The server can be a single point of failure
  - Decisions about where to locate functionality (in the client or in the server) are often complex and costly to change after a system has been built

Architectural pattern overview

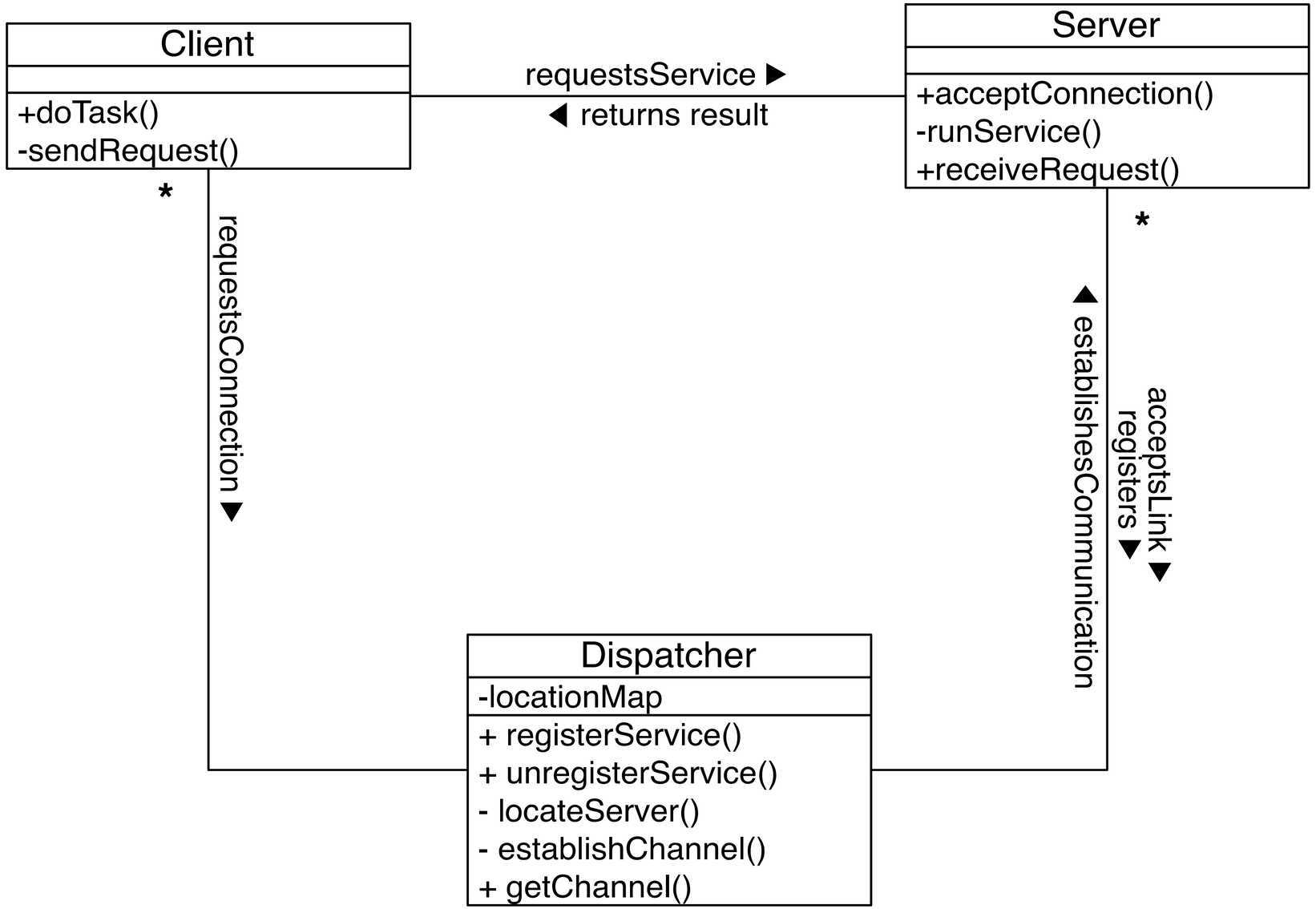


Client dispatcher server

- **Context**
  - When a client uses a remote server over a network it needs to establish a connection before the client can **communicate** with the server
- **Problem**
  - Client and server needs are not separated in many applications, causing unnecessary code complexity in service invocations → It would be better to
  - Separate the core functionality provided by the server from the details of the **communication** mechanism between client and server
  - Allow servers to **dynamically change their location** without impacting client code
- **Solution**
  - Insert a **dispatcher** component between client and server that provides the connection
  - Allow the client to refer to the server by **name** instead of the physical location (→ **location transparency**)
  - Establish a **channel** between client and server, reducing a possible communication bottleneck

From client server ….

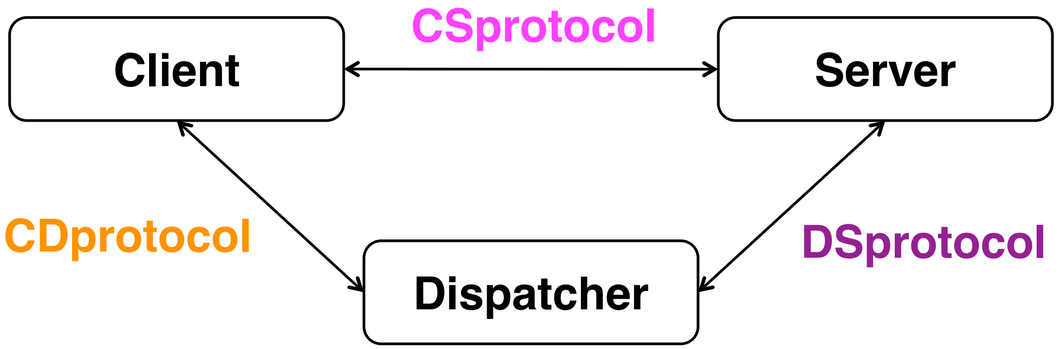
… to client dispatcher server



Benefits of the client dispatcher server pattern

+ Exchangeability of servers
  - Changing and adding servers without modifications to the dispatcher or client components
+ Location and migration transparency
  - Clients do not need to know where servers are located
  - Clients do not depend on any location information
+ Reconfiguration
  - Defer decisions about which servers should run until the startup time of the system, or during run-time
+ Fault tolerance
  - In case of server failures, new servers can be activated at a different network node without any impact to clients
  - System is more robust and fault tolerant

Protocols in the client dispatcher server pattern



- **CDprotocol**: specifies how a client must look for a particular server, deals with communication errors (time out, server does not exist, etc.)
- **DSprotocol**: specifies how a server registers with the dispatcher and determines the activities needed to establish a communication channel between client and server
- **CSprotocol**: specifies how client and server communicate with each other
  1. The client sends a message to the server using the communication channel
  2. The server receives the message, interprets it, and invokes one of its services. After the service is completed, the server sends a return message to the client
  3. The client extracts the result from the return message and continues its task

Steps to implement client dispatcher server (1 - 4)

1. During system design (in particular during the hardware-software mapping), identify the subsystems that act as clients and as servers and as dispatcher
2. Decide on the communication mechanism to be used for the protocols
   - Communication between client and server, between client and dispatcher, between dispatcher and server
   - Using a single communication mechanism decreases the complexity of the implementation but may not be efficient
3. Specify the protocols between the components (set up the channel and define the transmission of data)
4. Decide on a naming scheme for the dispatcher
   - Introduce a naming scheme that uniquely identifies servers and is location transparent (URLs are ok, IP addresses are ok if the IP location is static)

Steps to implement client dispatcher server (5 - 6)

5. Implement the dispatcher

- Decide on how to implement the 3 protocols using message based communication (sockets) or procedure calls (**RPC**, **RMI**)
  - **Example**: use local procedure calls if client and dispatcher reside in the same address space
- Design and implement the following classes: **`Request`**, **`Response`** and **`ErrorMessage`**
- Design and implement a class repository that maps server names to physical locations (use the repository pattern, implemented with hash table)

6. Implement the client and the server

- Decide whether server registers with the dispatcher at system startup time
- Decide if server can register und unregister during runtime

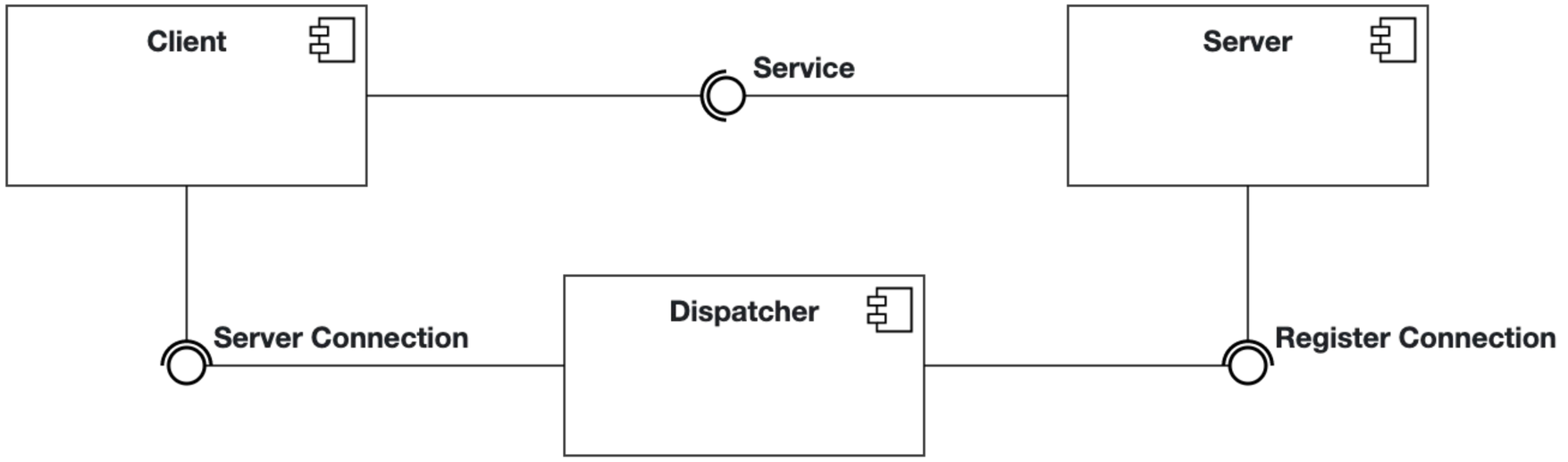
L04E03 Client Dispatcher Server Pattern Not started yet.

Start exercise **Medium** Due date in 7 days

20 min

7 pts

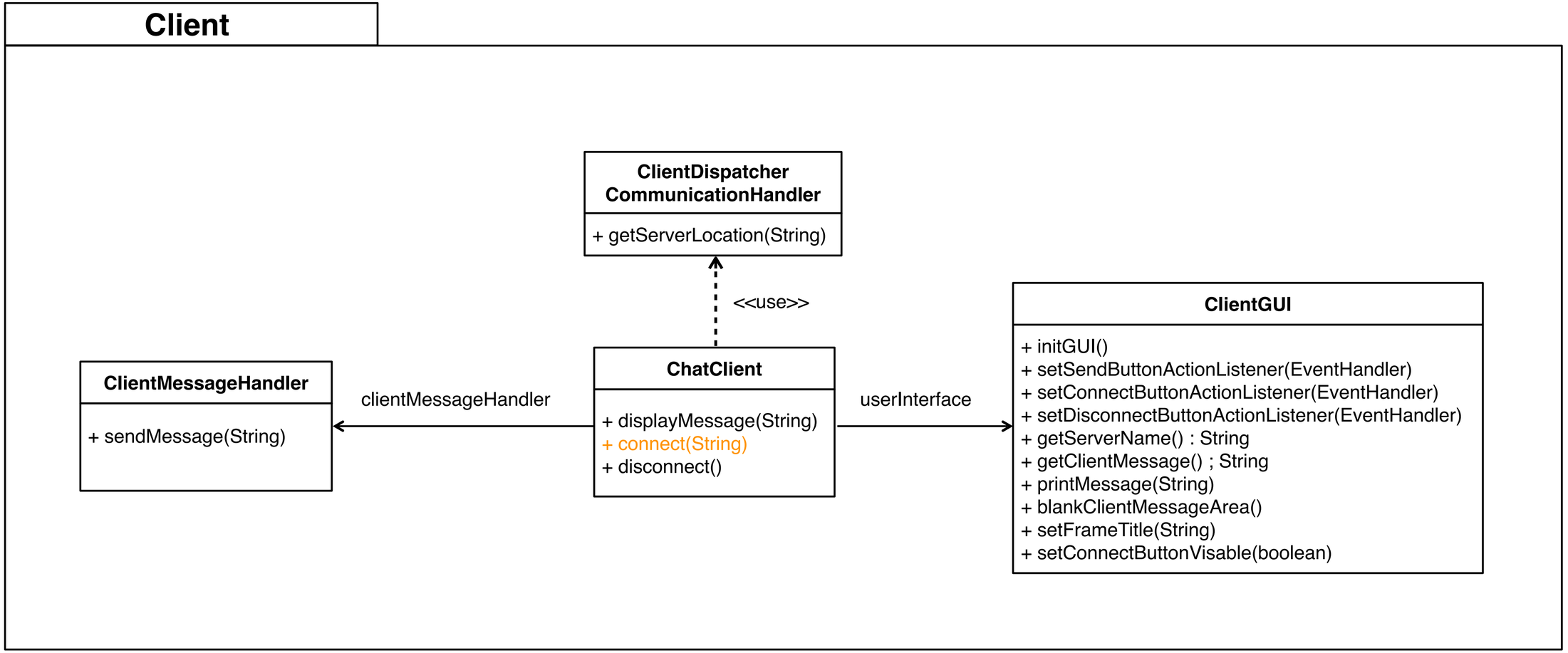
- **Problem statement:** provide an instant messaging service which provides location transparency, by reusing an existing system which uses a client server architecture



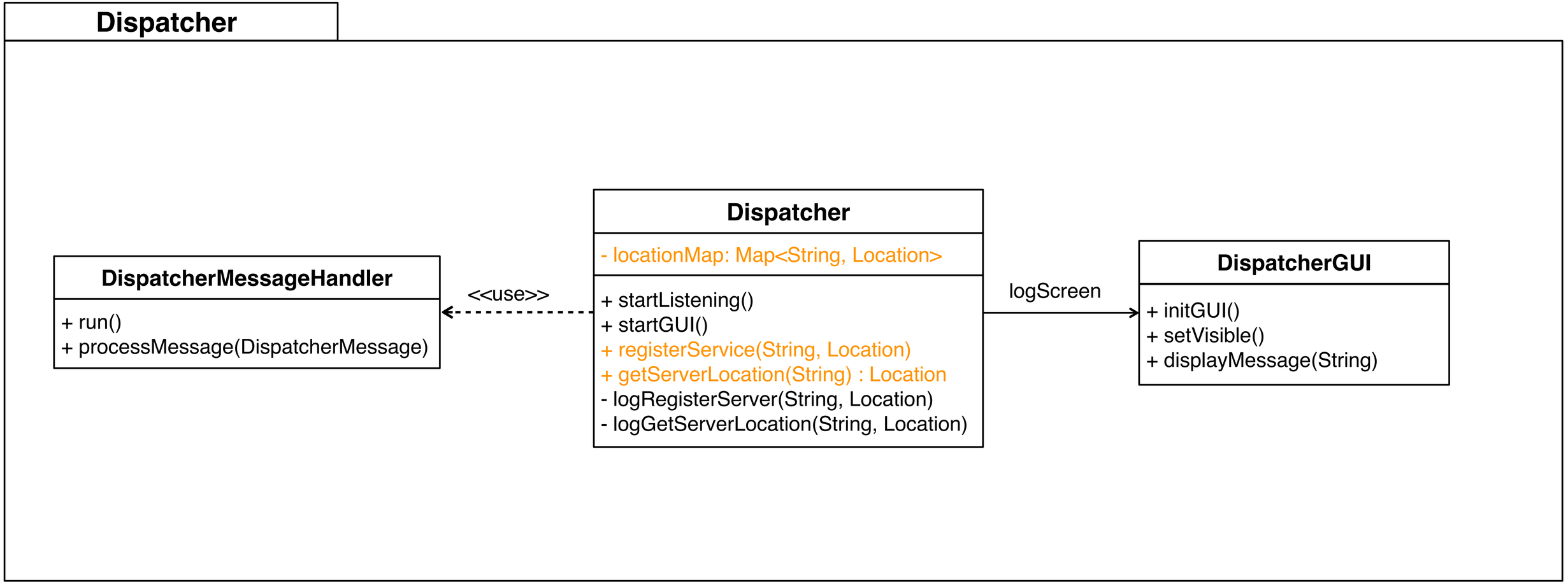
Hint: system startup

- Startup the subsystems in the following order
  1. Dispatcher
  2. Server
  3. Client
- Register the Server, for example with port 50000
- Connect the Client to the registered Server
- Send a message

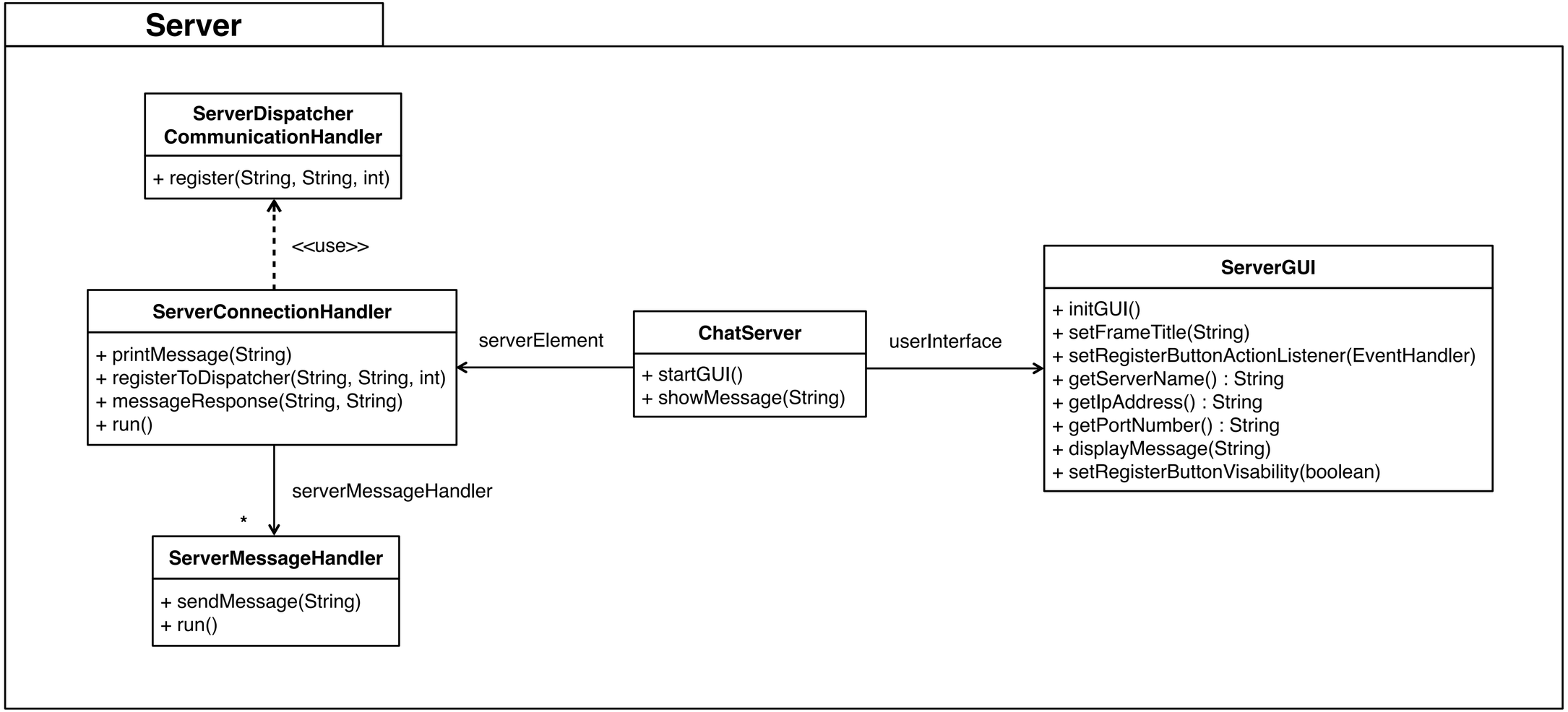
Hint: client subsystem



Hint: dispatcher subsystem



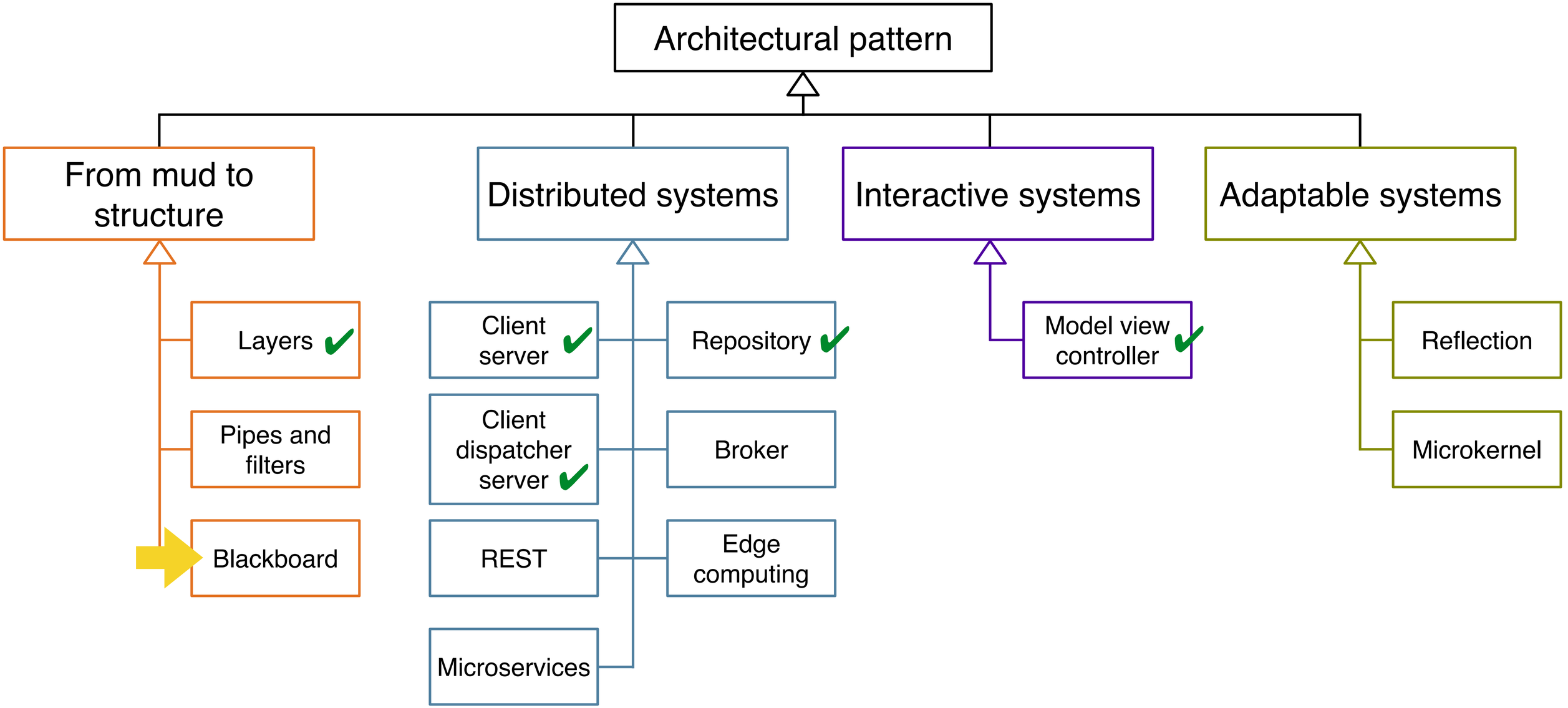
Hint: server subsystem



Outline

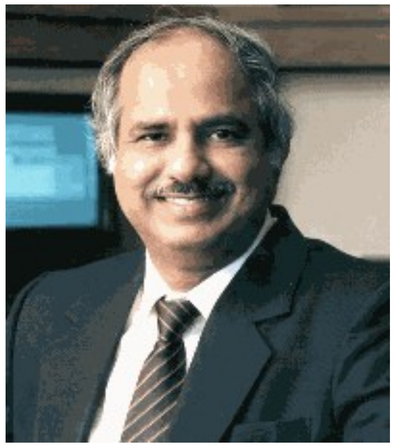
- Model view controller (part 1)
- Model view controller (part 2)
- Client dispatcher server
- Blackboard pattern

Architectural pattern overview



Historic context of the blackboard pattern

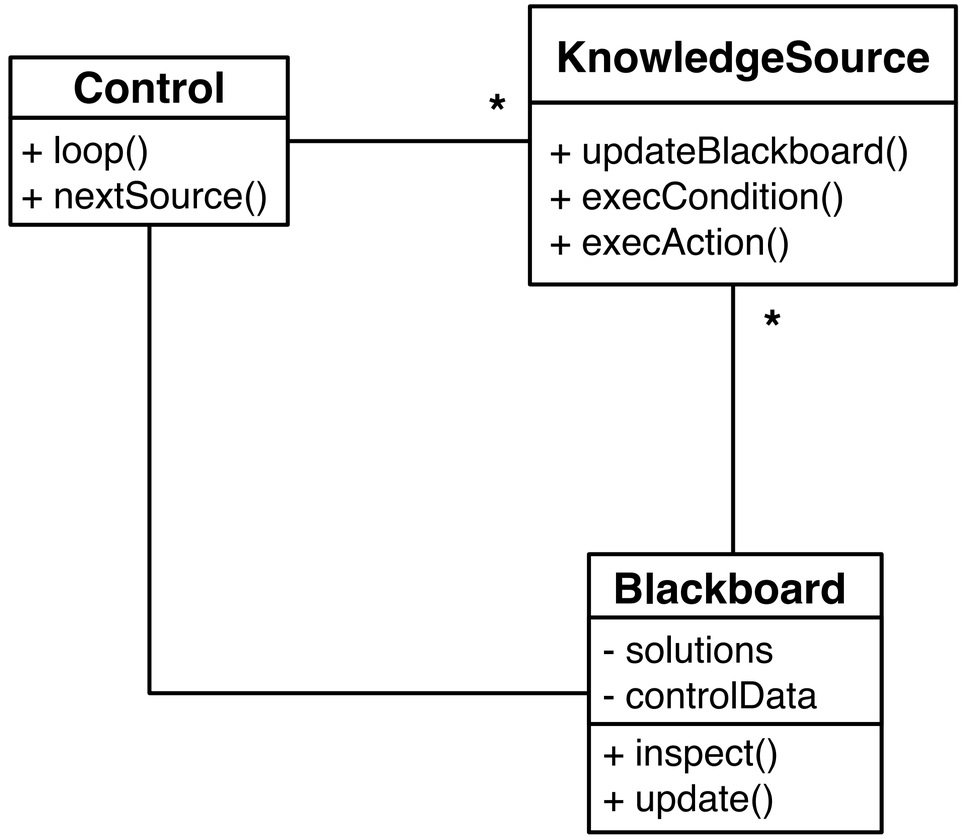
- The blackboard pattern was initially used in the **Hearsay II** speech recognition system for recognizing sentences from a vocabulary of 1200 words (first called the **blackboard architecture**)
- In **Hearsay II**, hypotheses about the sentence were kept in different data structures, so-called levels, in the blackboard (they are called solutions in the **blackboard pattern**)
- Name originates from „blackboard situation“: several human experts collaboratively solve a problem using a blackboard



Raj Reddy, *1937,
Carnegie Mellon University
- Major contributions in speech recognition (Hearsay II, Harpy), vision understanding, robotics, machine learning
- 1994: Turing Award (with Ed Feigenbaum)

Blackboard

- **Blackboard** is the repository for the problem, partial solutions, and new information (hypotheses,…)
- **Knowledge source**s read anything that is placed on the blackboard and place new information created by them on the blackboard
- **Control** governs the flow of the problem solving activity in the system: how knowledge sources are notified of new information on the blackboard



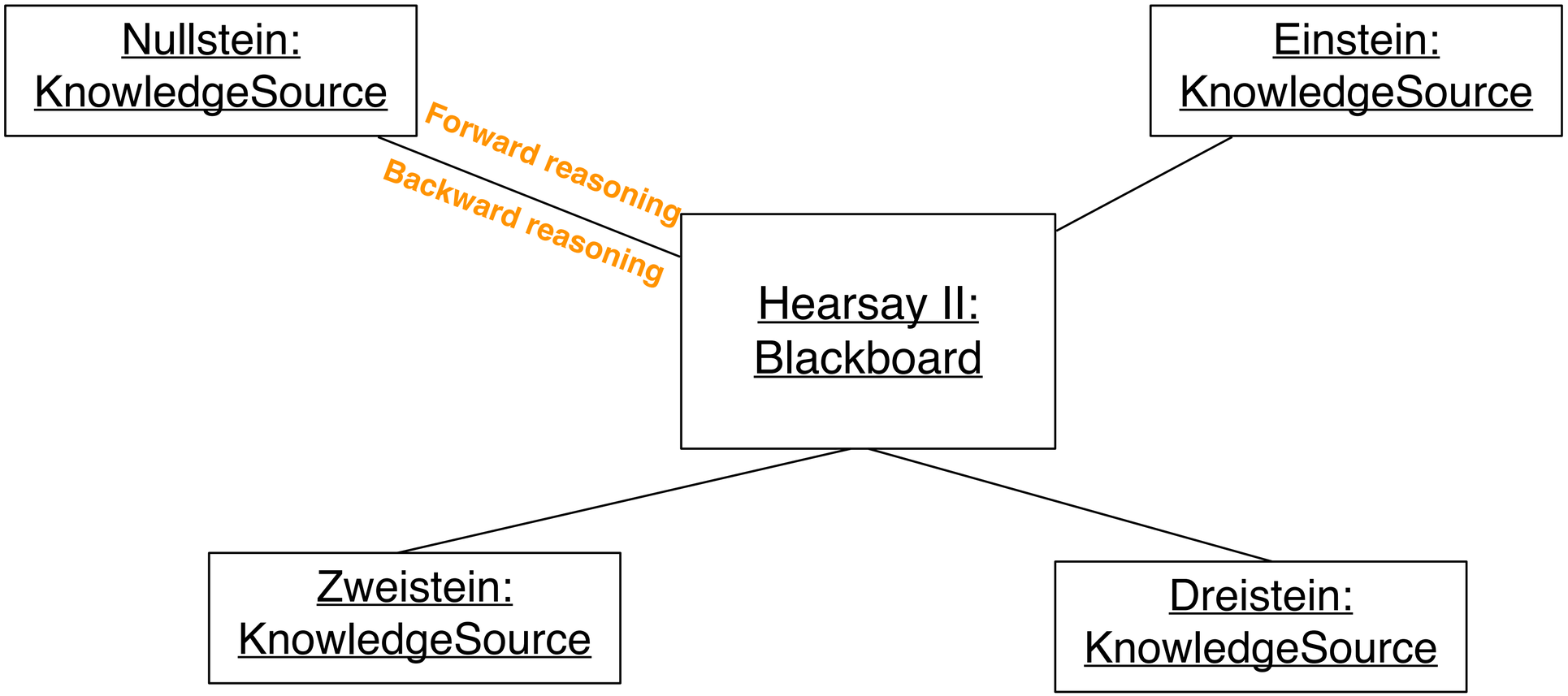
Synonyms

- **Control:** supervisor
- **Knowledge source:** specialist, expert
- **Blackboard:** knowledge sharing area

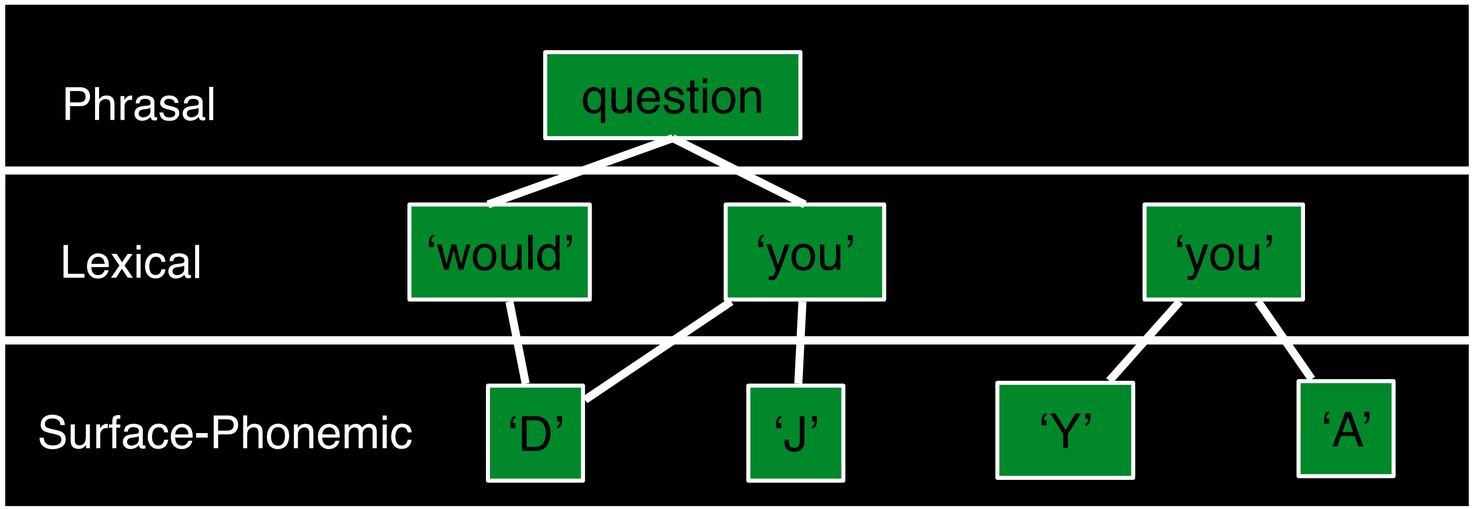
More general: hypotheses and reasoning

- Hypothesis
  - The blackboard consists of several hypotheses added by knowledge sources
  - Hypotheses rejected by a knowledge source are removed from the blackboard
- Forward reasoning
  - A knowledge source transforms a particular hypothesis for a solution into a higher-level solution
- Backward reasoning
  - A knowledge source searches at a lower level for support of a hypothesized solution

Object diagram for the Hearsay II example

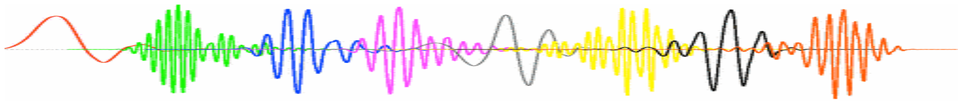


Example: interacting knowledge sources in Hearsay II



Blackboard

Incoming sentence:



‘A’ is certainly there

It sounds like ‘Y’

This means, we have ‘you’

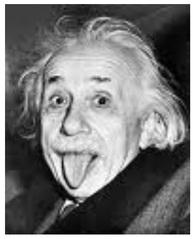
Combining these two, we got ‘you’

Wait, I think I heard ‘D’ too

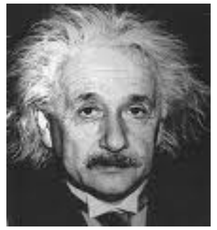
I think I heard ‘J’

Hey, this is a question!

Knowledge sources:

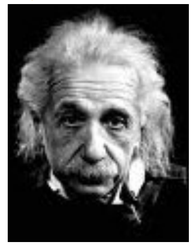


Einstein (SP)



There can only be one ‘you’, it must be ‘would’

Zweistein (L)

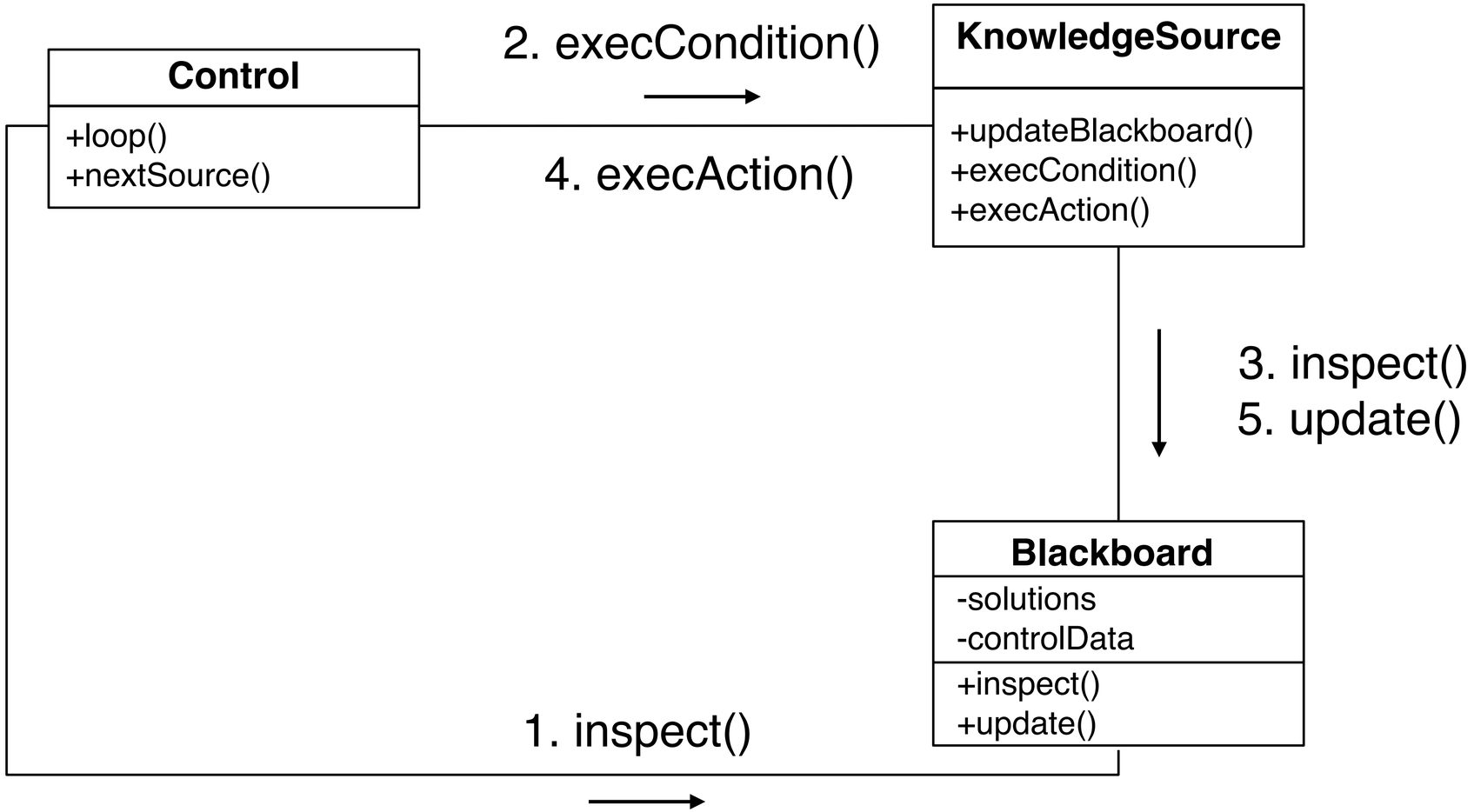


Dreistein (SP)

Nullstein (P)

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

Communication diagram



A sentence generator

- Given an input string a sentence generator produces an alternative sentence
- The sentence generator is based on the blackboard architecture style
- It has **two knowledge sources**
  - The **WordTagger** reads a text in a given language and assigns parts of speech to each word of the text (such as noun, verb, adjective)
  - The **SynonymIdentifier** identifies all the synonyms for a given word (e.g. based on information from the WordNet dictionary)

Sentence generator examples

Input sentence: Software can analyze, understand, and generate languages that humans use naturally

Processed output: *Software can analyze, understand, and make languages that humans practice naturally*

Input sentence: Google occasionally produces weird translations of the German language input

Processed output*:* *search_engine occasionally produces supernatural written_record of the German communication signal*

Input sentence: the quick brown fox jumps over the lazy dog

Processed output*:* *the fast chromatic canine jumps over the slow canine*

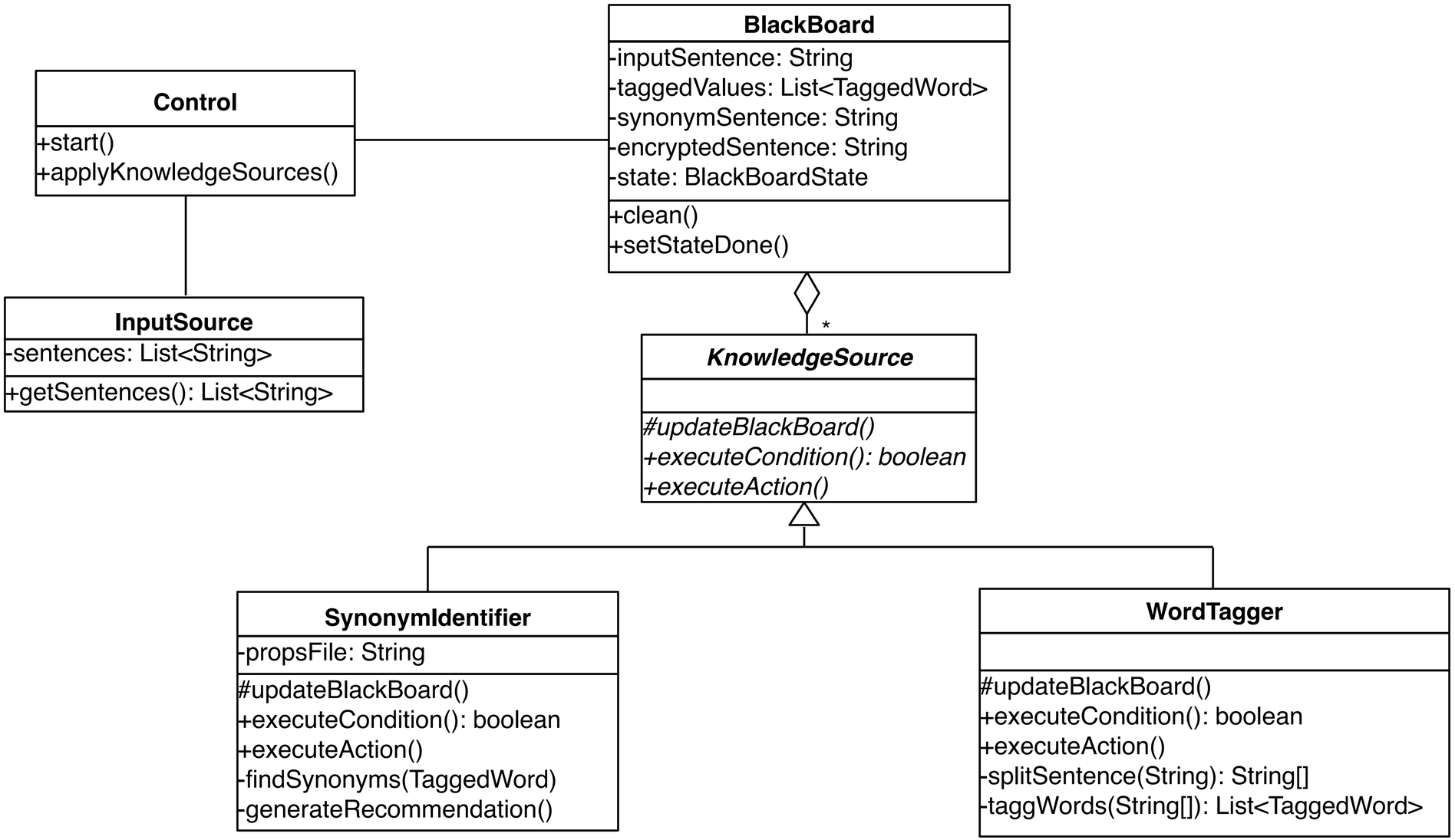
Input sentence: bob's phone was intercepted by the nsa

Processed output: bob's electronic_equipment was intercepted by the United_States_intelligence_agency

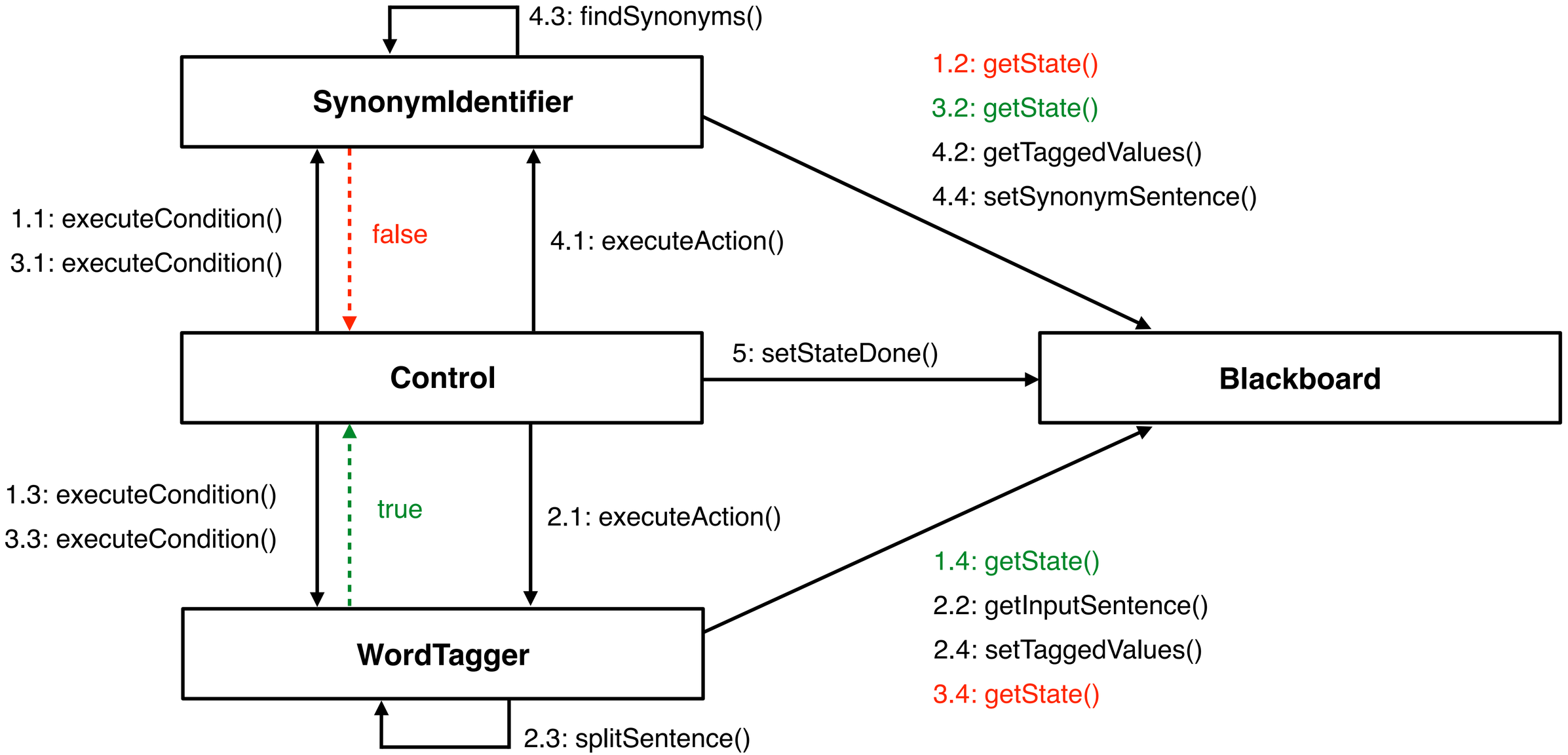
6 steps to realize a blackboard

1. Define the problem (make sure there is no algorithmic solution for the problem)
   - Identify the application domain and specify the requirements
   - Identify the actors (in particular the end user)
2. Define the solution space
   - Specify top level and intermediate solutions (for higher and lower abstraction levels)
   - Identify partial/complete solution candidates: a complete solution solves the problem, partial solutions solve parts of the problem
3. Identify the knowledge sources (KS): Identify their input and outputs
4. Define the blackboard: identify all the representations that are needed to allow the KSs to read from and contribute to the blackboard (not every KS needs to understand every representation!)
5. Define the control: implement the problem-solving strategy (Who can construct hypotheses? Who decides the credibility of hypotheses? Who decides what is an acceptable solution?)
6. Implement the knowledge sources (KS)
   - Split each KS into a condition part and an action part
   - Use different technologies: rule based systems, neural networks, in fact any computational intelligence method, but also conventional procedures/functions

Sentence generator (UML class diagram)



Sentence generator dynamic model (UML communication diagram)



Advantages and limitations of the blackboard

- **Advantages**
  - \+ Problem solving support: the blackboard pattern 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: blackboards are tolerant against noisy data, all results are just hypotheses: only hypotheses supported by strong data or other empirical evidence survive
- **Limitations**
  - \- 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

L04E04 Blackboard Pattern

Not started yet.

Start exercise

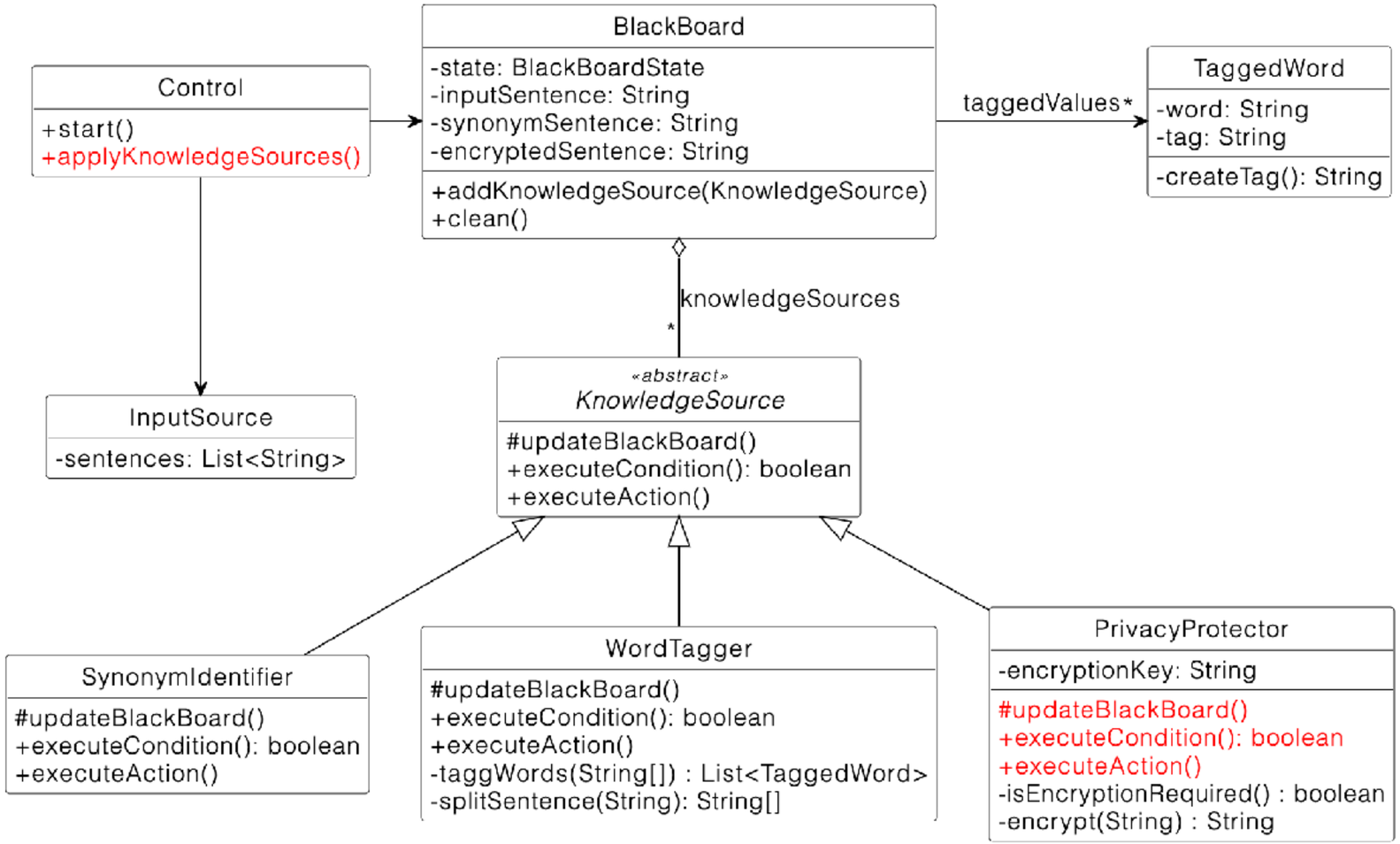
Hard

Due date in 7 days

20 min

10 pts

- Problem statement: natural language processing (NLP)



Summary

- Software architectures are instances of architectural styles, which are patterns for a subsystem decomposition based on specific design goals

- **Model view controller:** subsystem decomposition with 3 components: model (entity objects), view (boundary objects), controller (controller objects)

- **Client dispatcher server:** clients interact with servers via an additional name server

- **Blackboard:** subsystems are knowledge experts working together to solve a problem without a known solution strategy

Literature

- https://www.oracle.com/technical-resources/articles/javase/application-design-with-mvc.html
- D. Parnas, On the criteria to be used in decomposing systems into modules, CACM, 15(12), pp. 1053-1058, 1972
- R. Kahn and V. Cerf, A Protocol for Packet Network Intercommunication, 1974: http://www.cs.princeton.edu/courses/archive/fall06/cos561/papers/cerf74.pdf
- V. Lesser, R. Fennell, L. Erman and R. Reddy, Organization of the Hearsay-II Speech Understanding System, IEEE Trans. on Acoustics, Speech & Signal Processing, Vol. ASSP-23, Nr 1, pp.11–24, 1975
- J. Coutaz,PAC: An Object-Oriented model for Dialog Design, Human-Computer Interaction, Proceedings of the Interact 87 conference, pp. 431-436, Elsevier, 1987
- D.E. Perry and A.L. Wolf, Foundations for the study of software architecture, ACM SIGSOFT Software Engineering Notes, 17(4), pp. 40-52, 1992
- M. Shaw and D. Garlan, 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, Recommended Practice for Architecture Description of Software-Intensive Systems
- Kristina Toutanova and Christopher D. Manning, Enriching the Knowledge Sources Used in a Maximum Entropy Part-of-Speech Tagger, Proceedings of the Joint SIGDAT Conference on Empirical Methods in Natural Language Processing and Very Large Corpora (EMNLP/VLC), Hong Kong 2000
- Kristina Toutanova, Dan Klein, Christopher Manning and Yoram Singer, Feature-Rich Part-of-Speech Tagging with a Cyclic Dependency Network, Proceedings of HLT-NAACL pp. 252-259, 2003
- Alan Cooper, The Inmates Are Running The Asylum: Why High-tech Products Drive Us Crazy and How to Restore the Sanity, Que 2004