Architectural Design Patterns

Software techniques

BME AUT
Department of Automation and Applied Informatics

Content

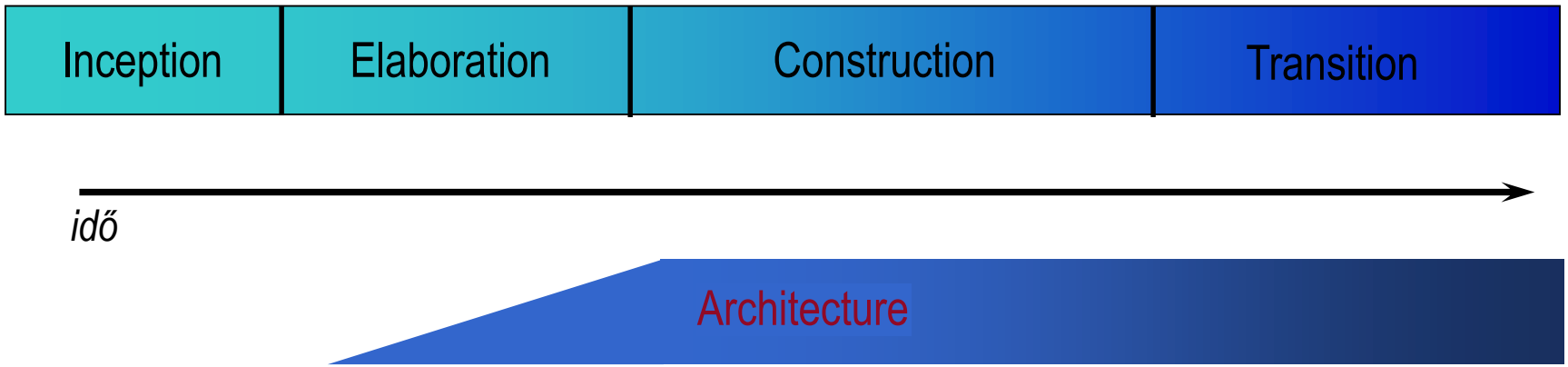
- Basic architectural design patterns
  - Layers
  - Pipes and filters
  - Architectures for enterprise information systems
  - MVC
  - Document View

What is the Architecture of an application?

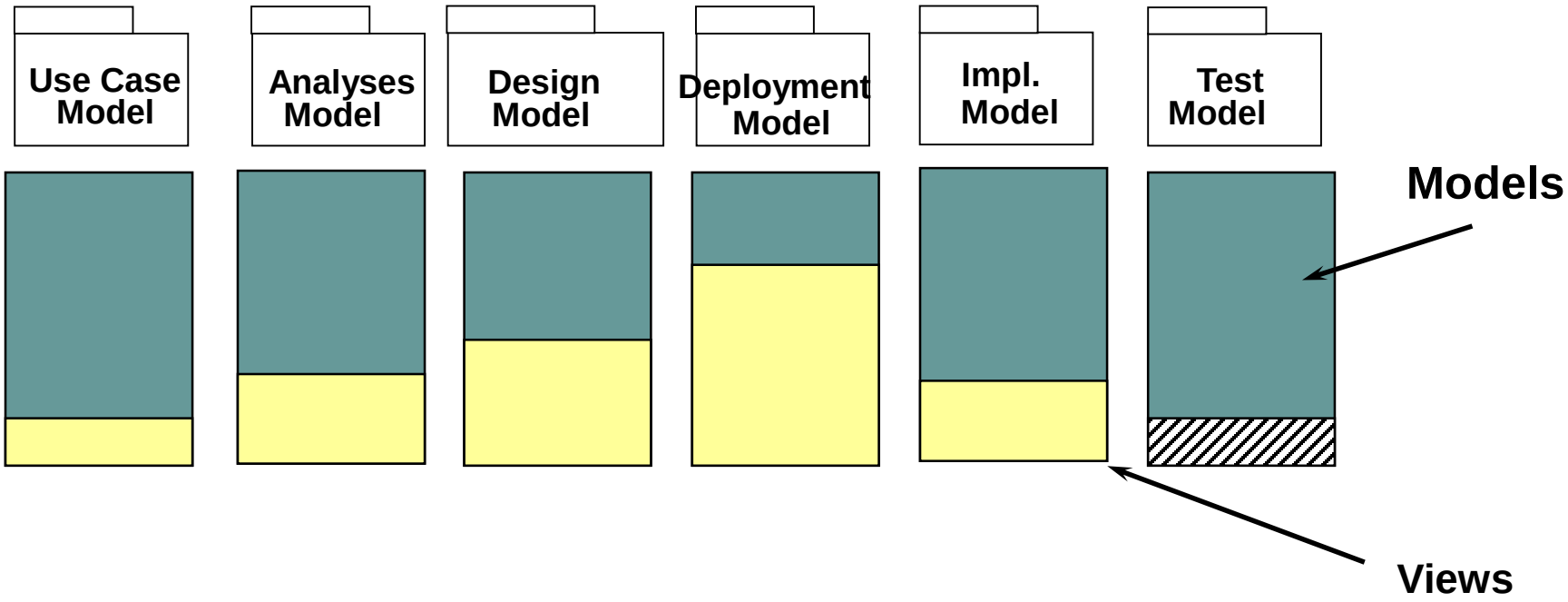
- The structure of the software system.
- The composition and co-operation of structural elements.

Architecture-centric

- Models specify, document and makes the architecture visible.
- The Unified Process continuously refines the executable architecture of the system

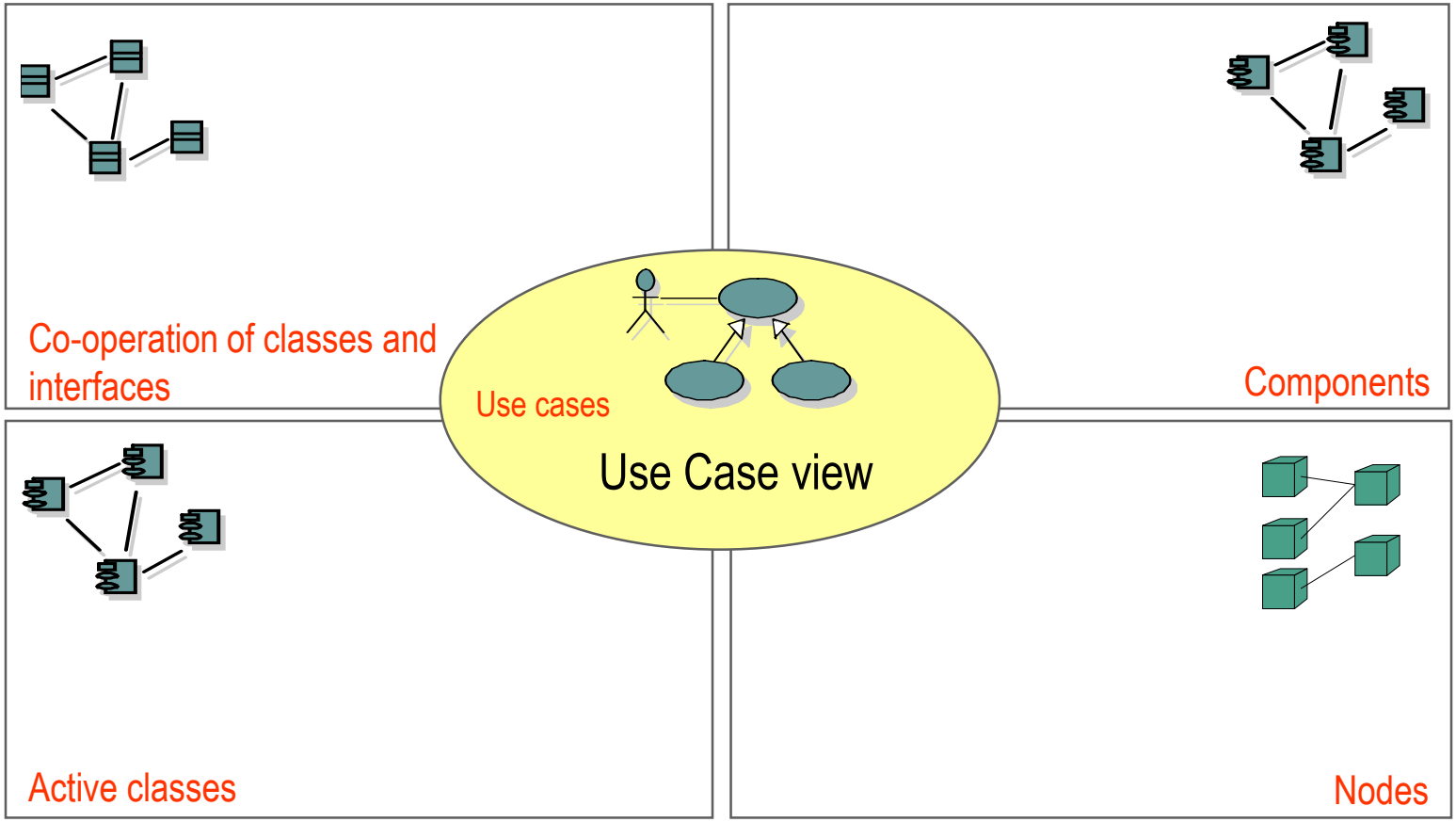


Architecture and the models



The architectural information content of model views

The views of architecture



Structure
Package, subsystem

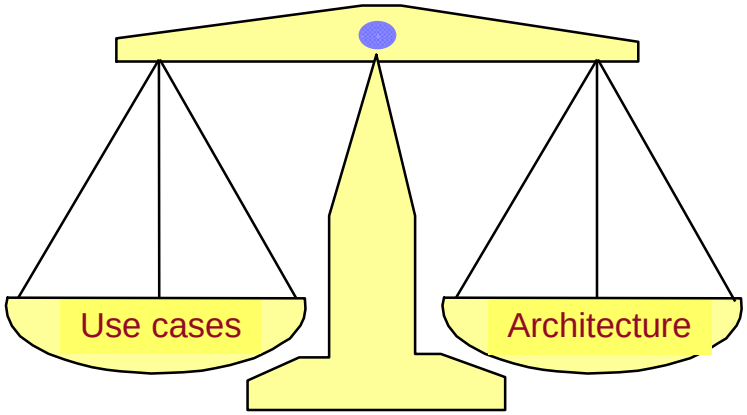
Dynamic behavior
Co-operation, States

What is the Architecture of an application?

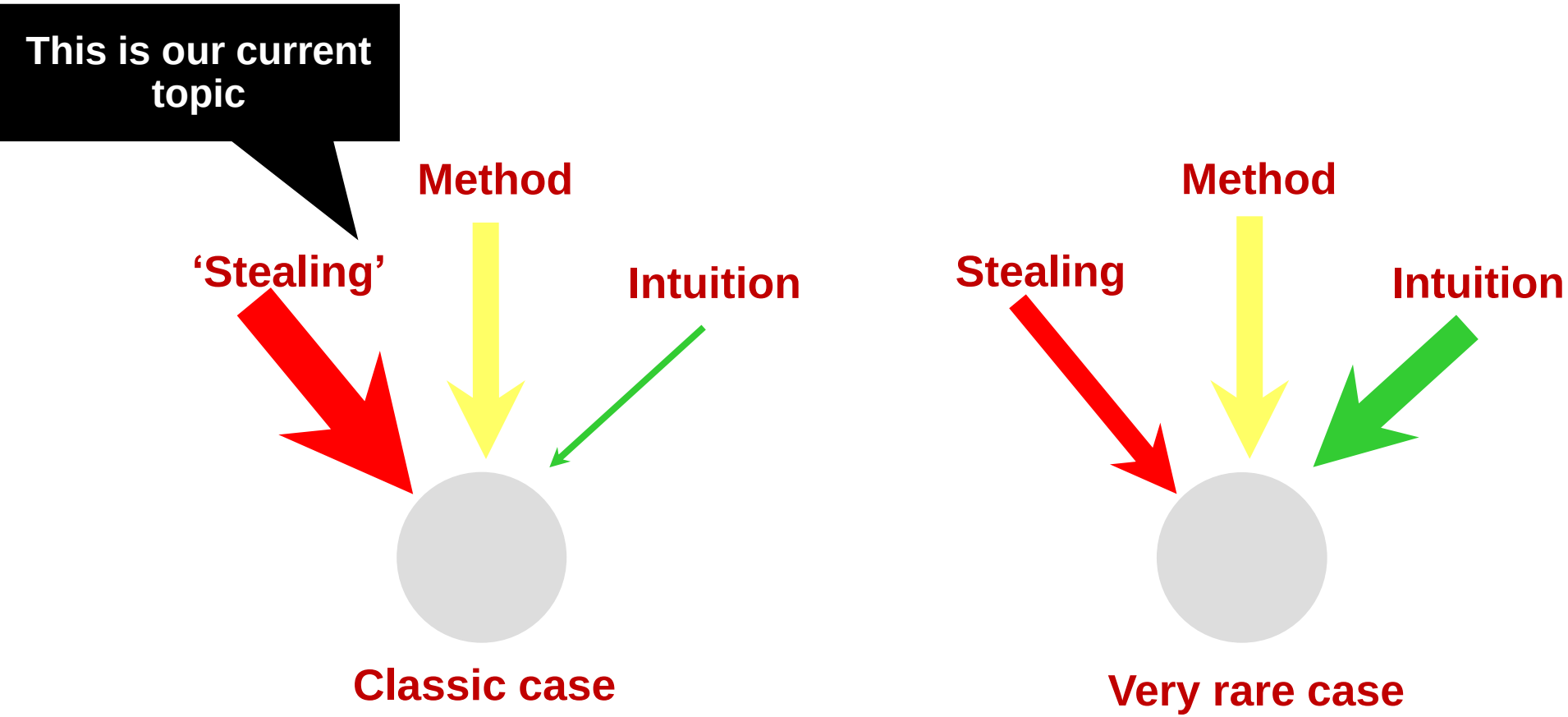
- **Use cases**
  - Functional requirements.
- **Design**
  - Most important packages (namespaces), subsystems, classes.
- **Implementation view**
  - Components, dlls, executables, structure of source code.
- **Process view**
  - Concurrency, processes, threads, avoiding deadlock, startup, shutdown, performance, scalability.
- **Deployment view**
  - How the different components are placed onto hardware resources.

Architecture and functionality

- Architecture and functionality must be in balance!
  - Function: work well! ⟷ Architecture: be well structured!



Sources of architecture



'Stealing': we replicate something that works

Architectural design patterns: collection of tried and tested architectures.

Layers

Layering

- A very basic designing concept
- Where is it used?
  - In the batch world it did not exist
  - OS: driver - OS (API) – application
  - Network protocols
  - Enterprise information systems
  - Etc.

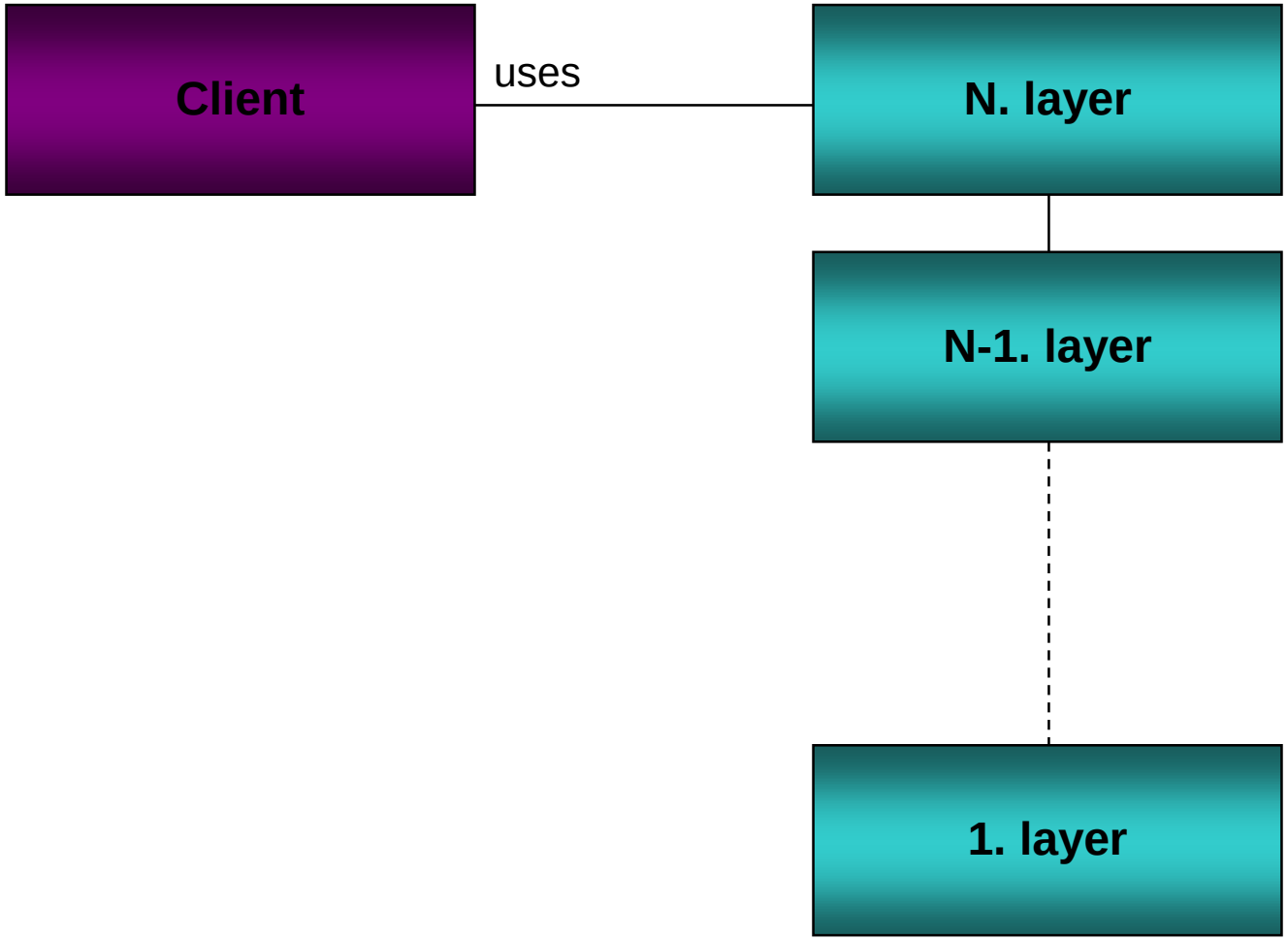
Example : OSI vs. TCP/IP

- OSI protocol layers:
  - Application
  - Presentation
  - Session
  - Transport
  - Network
  - Data Link
  - Physical

Use it…

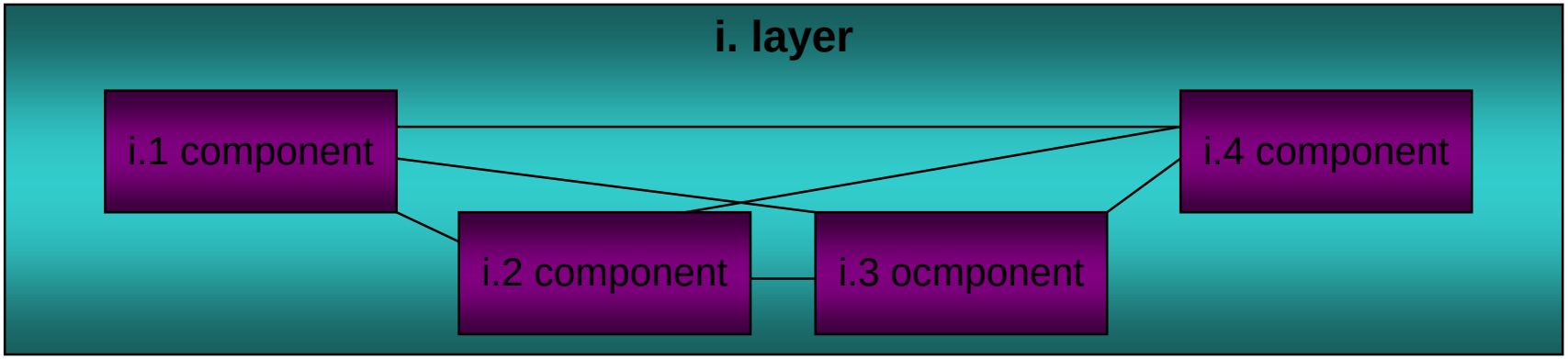
- … if the system is a combination of both low and high level functions
- …if we want the changes in the code to not affect the whole system (just the concerned layer)
- … if modularity is important
  - Components are replaceable
  - Components are reusable
  - Portability (platform independency)

Strict layering



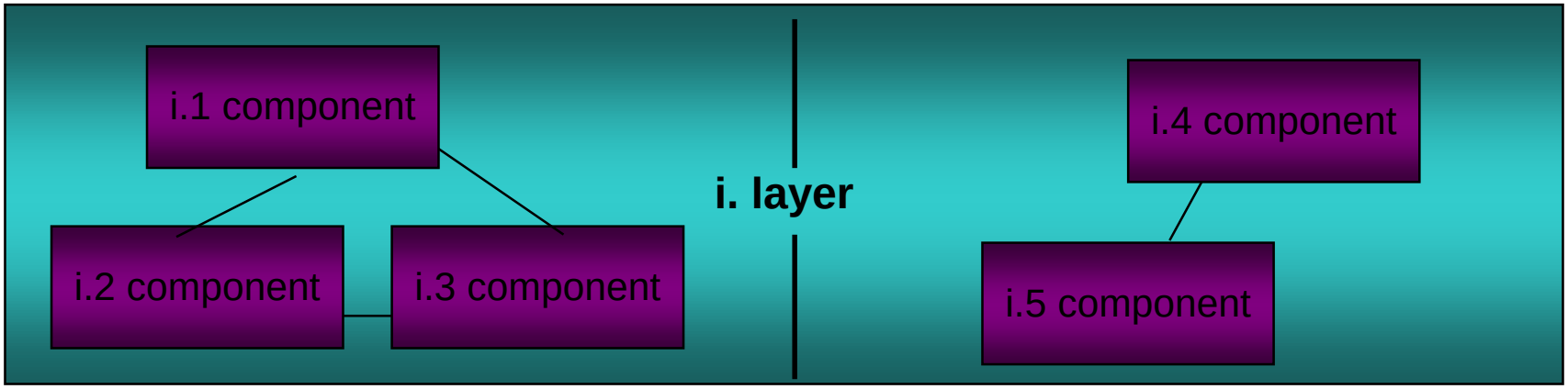
Strict layering

- The i-th layer
  - Publishes services for the (i+1)-th layer
  - Uses the services of the (i-1)-th layer
- Inside of a layer components can have any dependencies between themselves



Vertical partitioning

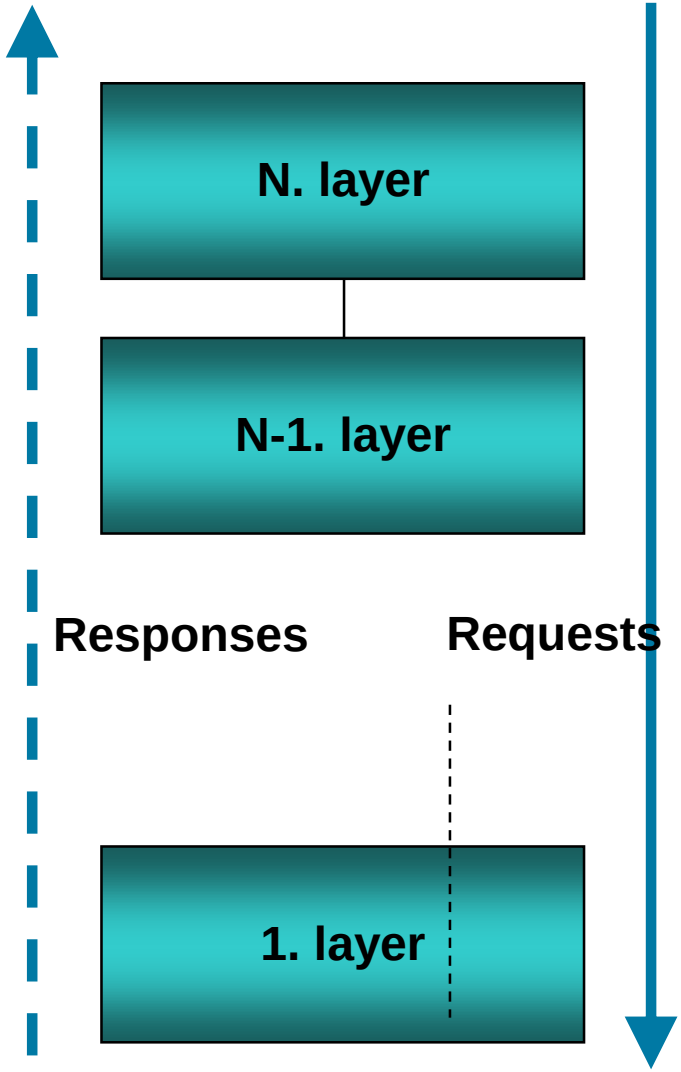
- Components can be grouped inside a layer based on their dependencies on each other.



- Advantage: The groups can be implemented and replaced independently.

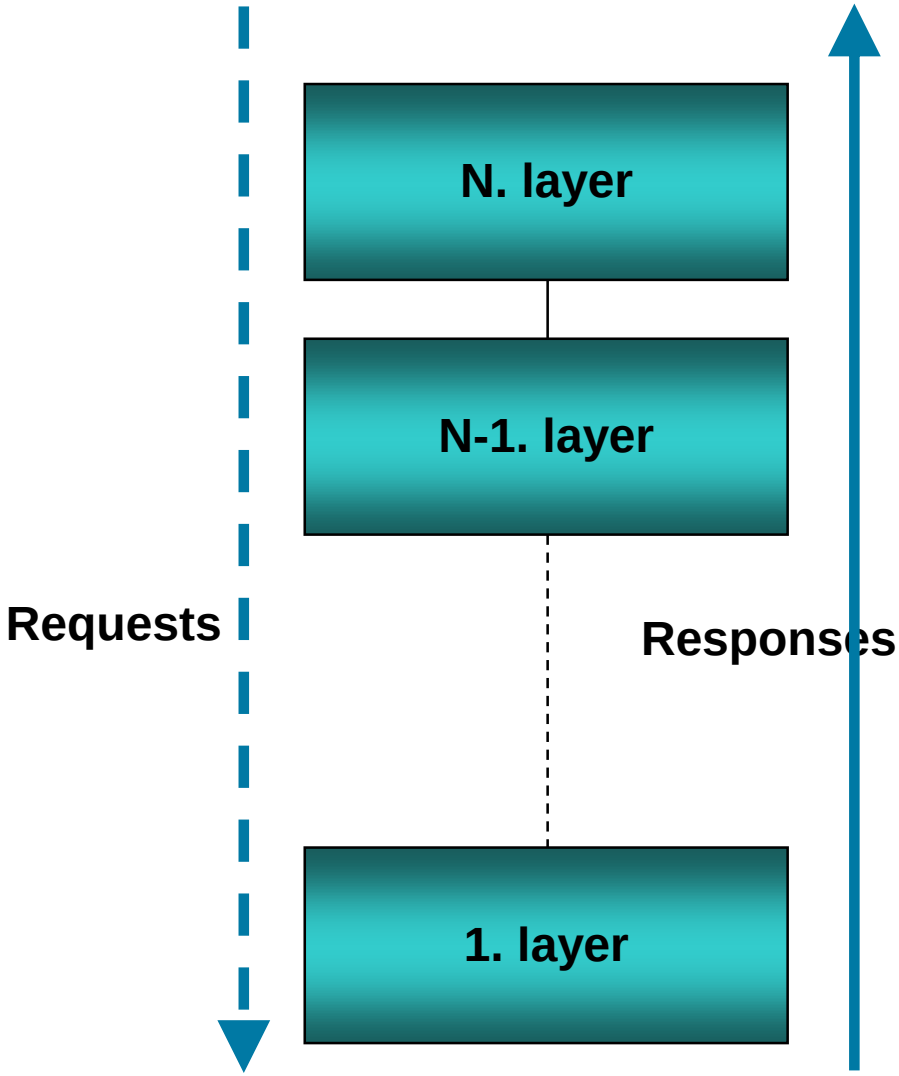
Scenario 1.

- Top-down
  - Drivers
  - „Polling”



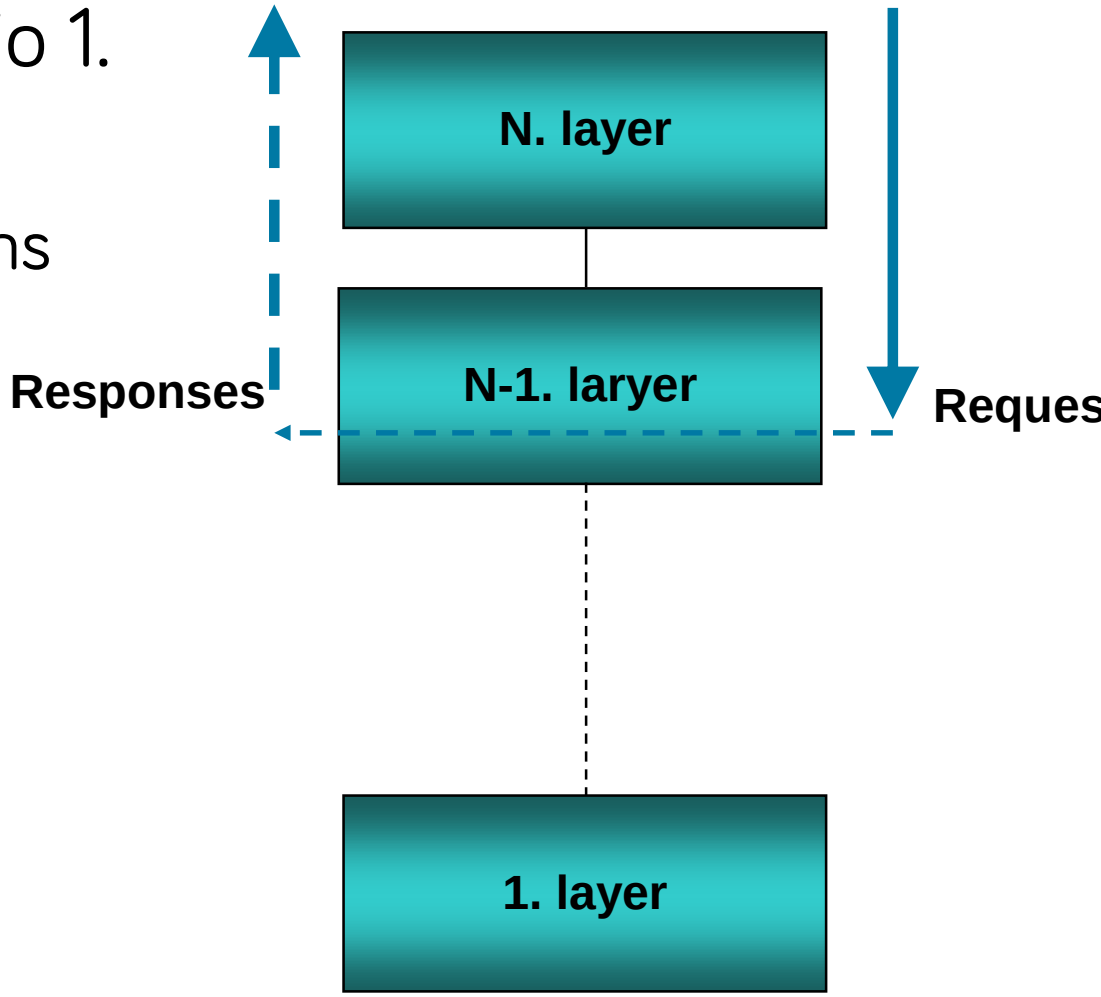
Scenario 2.

- Bottom-up
  - Drivers
  - „Interrupt driven”



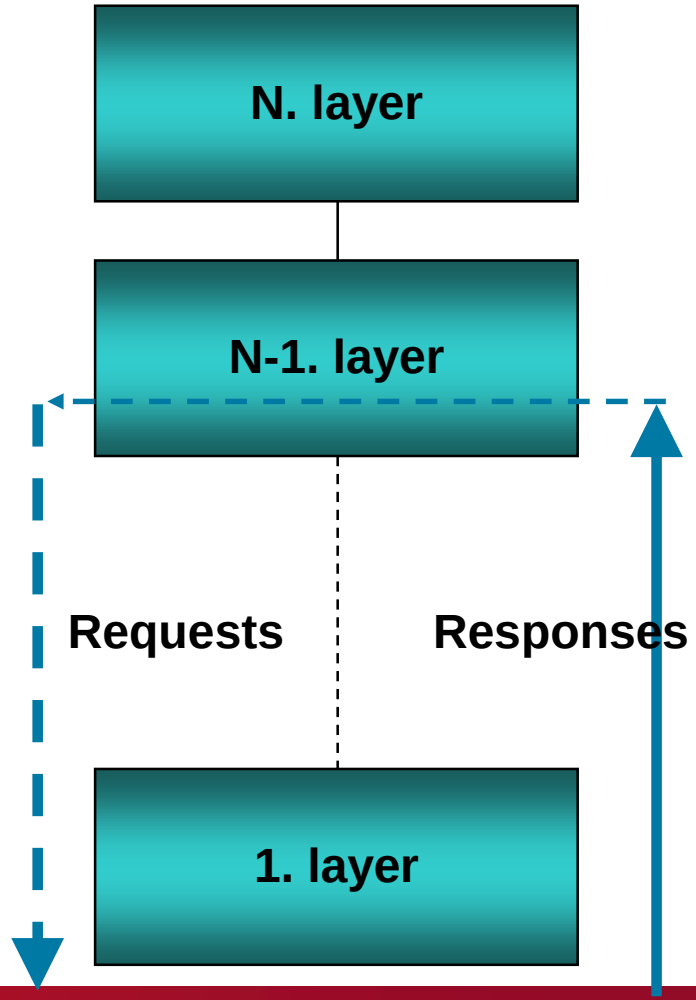
Scenario 3.

- Interrupted Scenario 1.
  - „Cache”
  - In case of exceptions



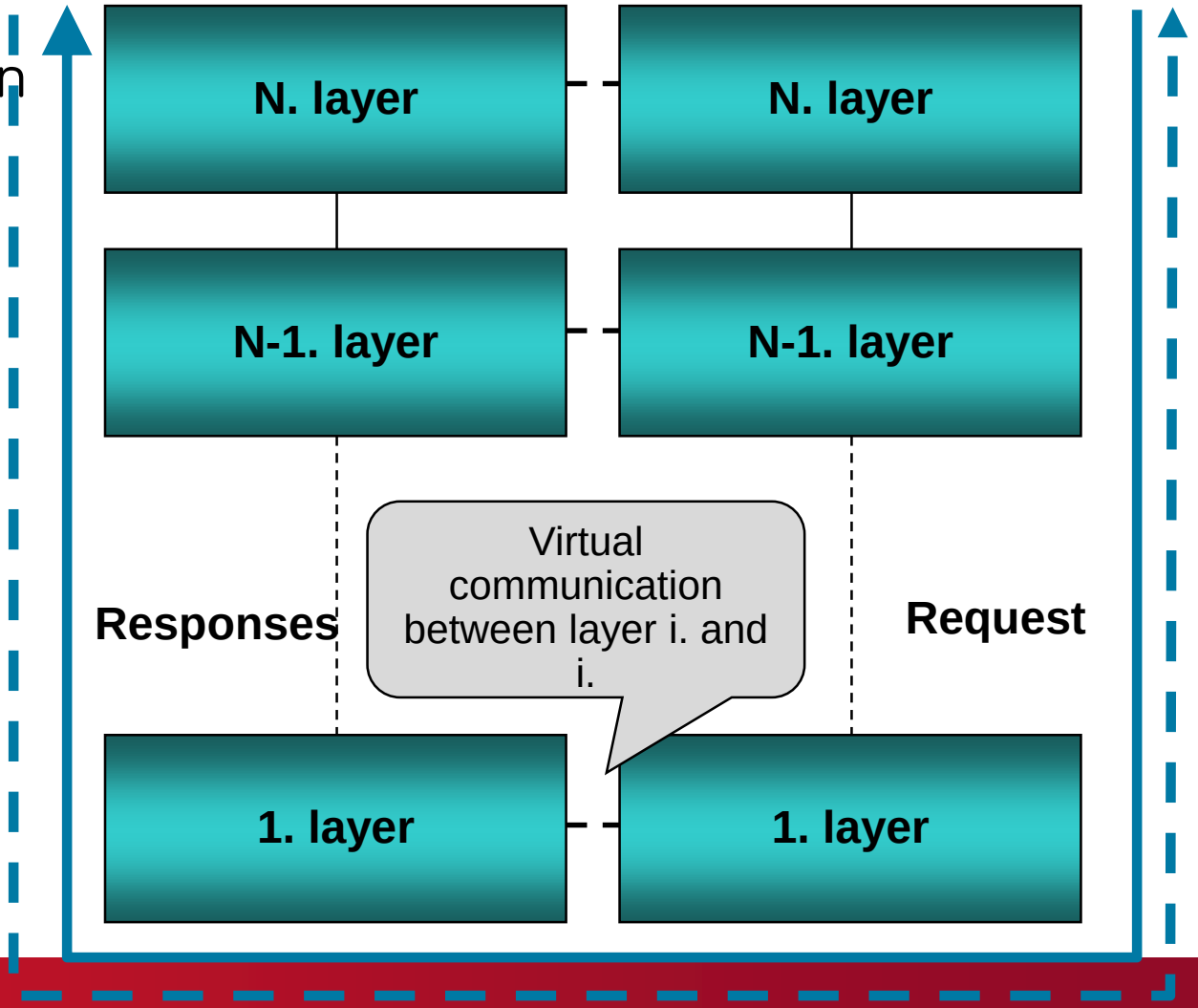
Scenario 4.

- Interrupted Scenario 2.
  - Communication of the lower layers
  - Error handling



Scenario 5.

- The communication of two layered architectures
- Network stack



WCF example

Advantages of using layers

- If the problem is too complex: introduce a new abstraction level (a new layer) and solve the problem based on that.
- It is enough to get to know just one layer, there is no need to understand the rest (e.g. we don't need to know how Ethernet works to use FTP)
- Layers can be replaced and developed independently from each other.
- Based on the same layer, many services can be developed (reusability): e.g. based on sockets -> Web server, FTP, etc.
- Under a certain layer other layers can be replaced (e.g. FTP works over Ethernet but it can also be implemented over a mobile environment)

Disadvantages

- Unnecessary complexity for simple problems
- Changes may effect all the layers in a cascade style
  - E.g. if we introduce a new database field all the layers have to be extended between the GUI and the database layer
- Performance

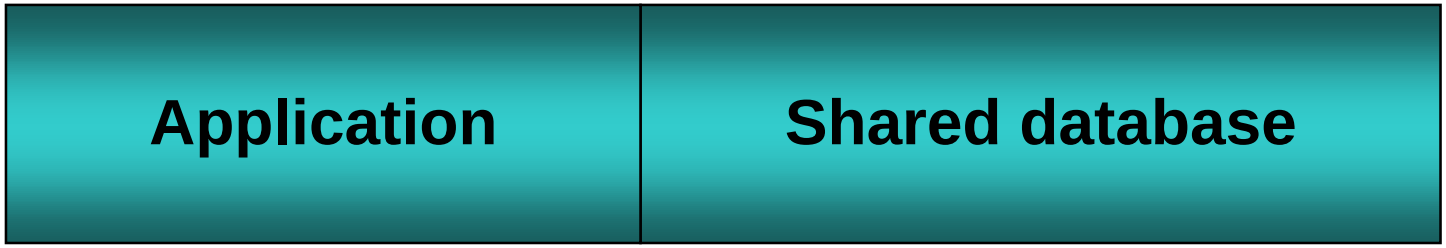
Layers of information systems (enterprise systems)

Information systems

- No layers
  - Batch systems
- Two-layer architecture
  - Client-server
- Three-layer architecture
- Service Oriented Architecture (won't be covered)

Information Systems

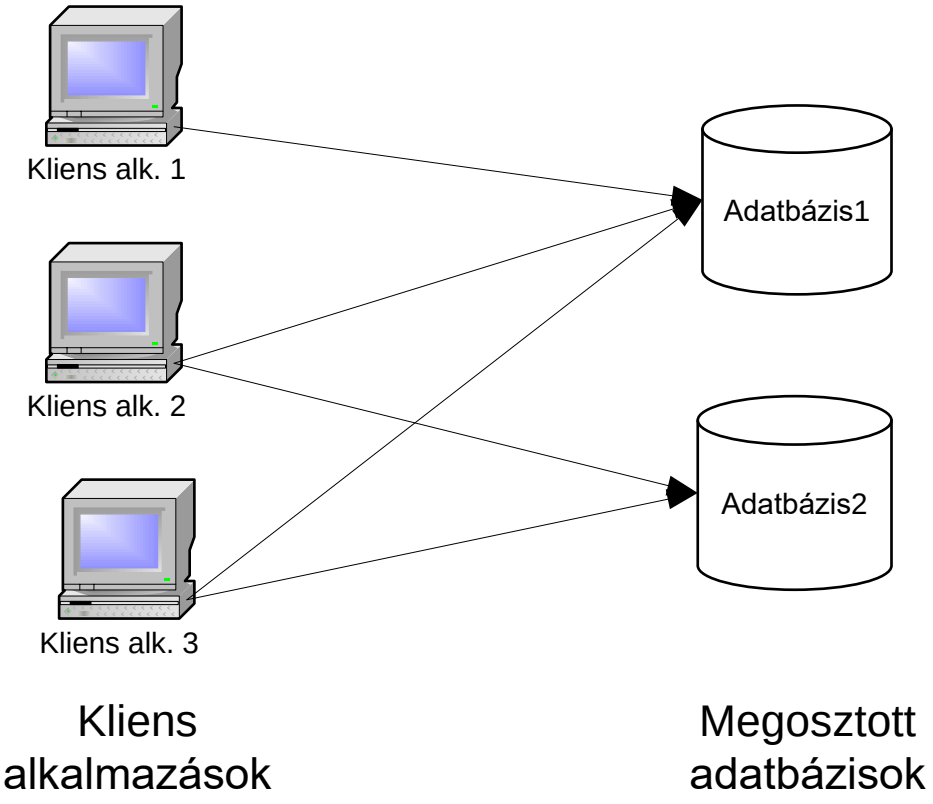
- Two-layer architecture
  - Shared database (files, DBMS)
  - Applications
  - Structure:



  - Better known as: Client-Server

Two-layer architecture

- The client applications can directly access the database
- Basic idea: the database is shared, the processing/presentation is distributed.



Advantages of the two-layer architecture

- Data management can be separated from the applications
- The shared databases can be uses from different applications (views).
- RAD support (Delphi, VB, PowerBuilder, .NET), databinding -> very popular.

Two-layer architecture - disadvantages

Basic problem: where to put the business logic (data integrity, business rules, calculations, validations)?

- A) Put it into the application
  - ☹If we create multiple applications the business logic will be duplicated in them -> more difficult to maintain
  - ☹The database can not be modified in order to be compatible with the existing applications
  - ☹GUI and business logic code is mixed
- B) Put it into the database (stored procedures)
  - ☹It is not supported by all the database management systems (MySQL)
  - ☹Not object oriented, restricted capabilities

Two-layer architectures - disadvantages

- The application is based on the physical structure of the database and not on a semantic schema.
- The database if often denormalized to improve performance: it has to be handled in the client.
- In the web world: suppose we have got a WinForms client and now we want to create a web client as well. It would be desired not to implement/put the same business logic code again in the web application. The two-layer model does not support it.

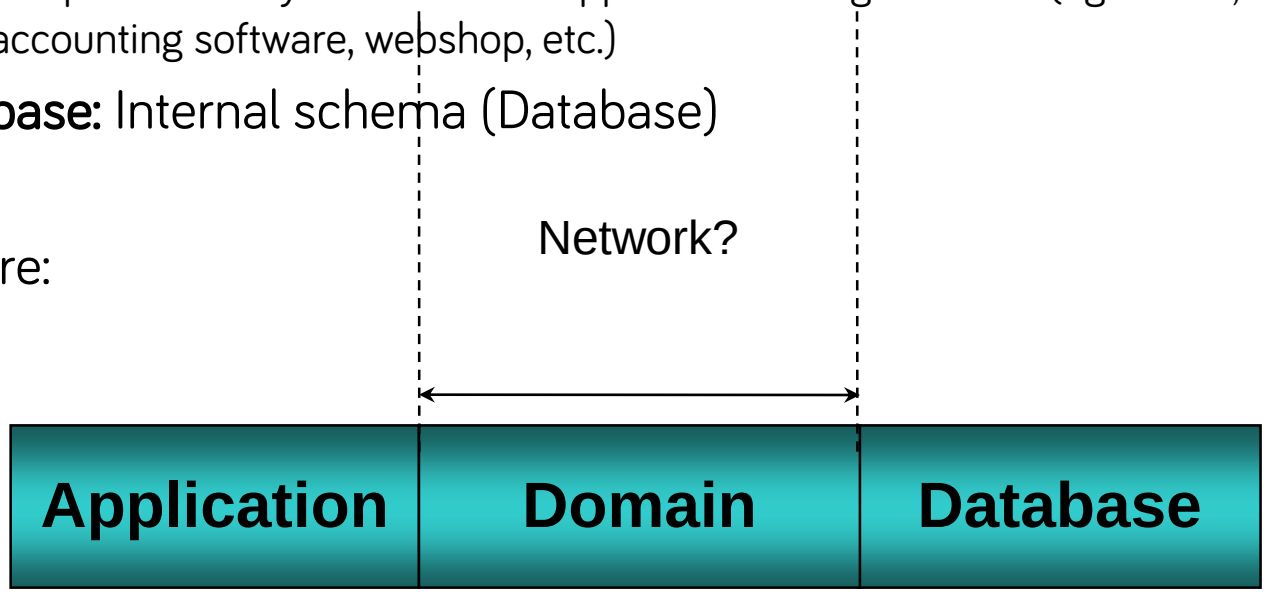
Two-layer architecture

- When should we use it?
  - For simple applications it can be a good solution
    - In this case it is not worth creating extra layers
  - If we don't want to reuse the business logic in multiple applications or subsystems

Three-layer architecture

- Three-layer architecture
  - **Presentation:** external schema (Application)
    - Application logic layer, usually the UI (user interface)
  - **Business logic:** Conceptual schema (Domain)
    - this part is strictly related to the application's usage domain (eg. Game, accounting software, webshop, etc.)
  - **Database:** Internal schema (Database)

Structure:



The newly introduced layer contains the business logic.

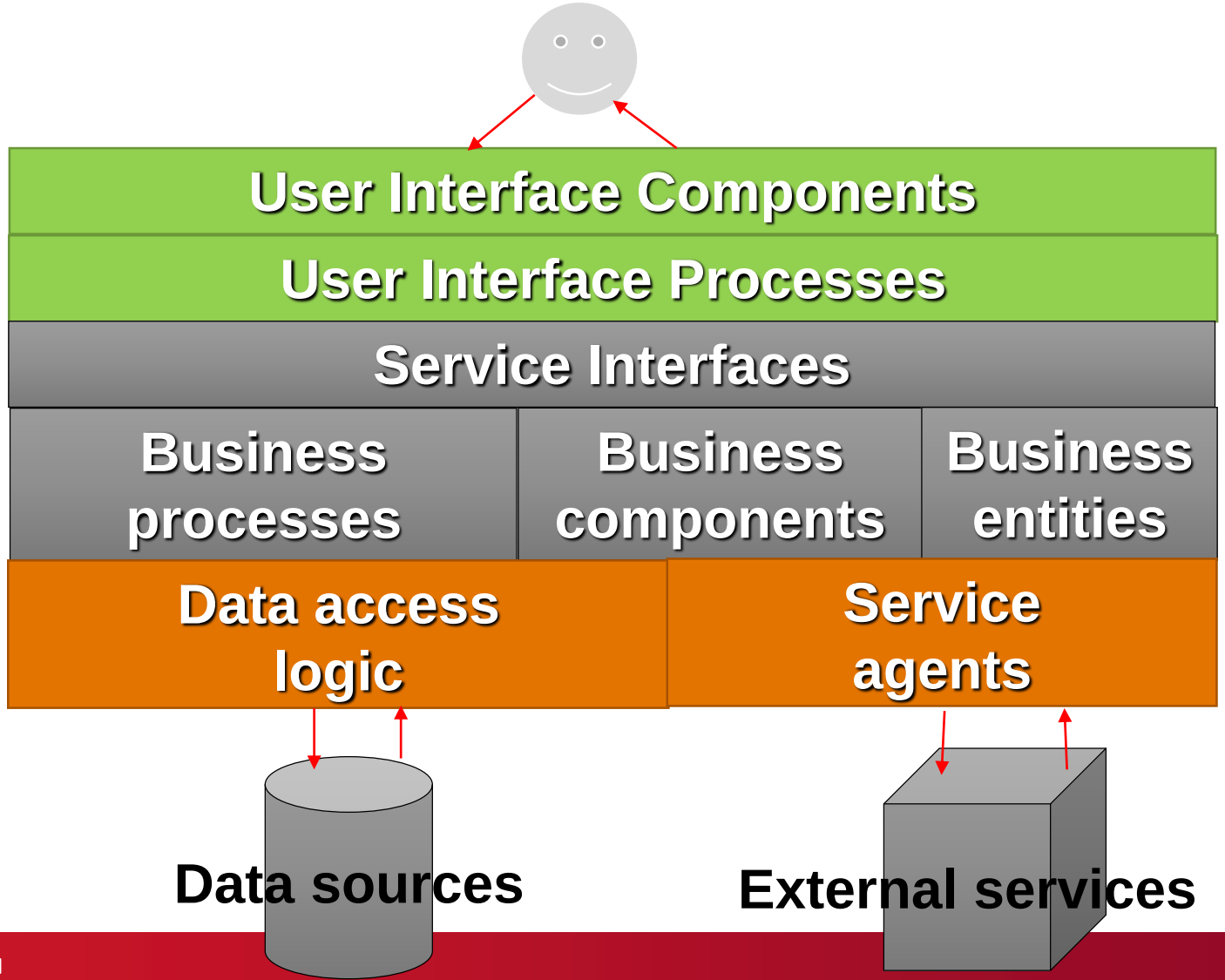
Three-layer architecture - advantages

- The business logic code
  - is not duplicated in the client applications
  - is not scattered around in the client/GUI
  - it is independent from the client/GUI -> the business logic is reusable (e.g.: reuse the BL from a thick (WinForms) and from a thin client (ASP.NET))
  - Automated tests can more easily be used (e.g. NUnit)
- The application is not so dependent on the data structure
  - The database can be re-structured independently from the application
- The middle layer may cache the data*
- IDE support*

Three-layer architecture

- Disadvantages
  - Extra complexity, more coding
  - More resource intensive
  - The business logic entities must be converted to database records
  - Not so widespread

The Microsoft three-layer model



The Microsoft three-layer model

- This is not Microsoft specific, this is a common architecture
- Explanation
  - Data Access Layer – DAL: Hides the details of the database management systems (e.g. MSSQL, Oracle).
  - Service agents: Communicates with external services.
  - Business entities: Represents some business data (e.g.: order, customer), that travels between business logic components.
  - Business components: Implements some business logic (e.g. making a new order, listing some items, business calculations, etc.)

The three-layer model of Microsoft

- Business processes: Implements and controls complex business workflows that are made up by smaller/elementary steps.
- Service interfaces: A thin layer: the client can only access the operations of the business logic through these interfaces -> the business logic can independently be modified (the client don't need to be updated while the interface remains unchanged).
- User interface components: The forms, windows and controls of the user interface and the GUI logic.
- User interface processes: Implements and controls complex user interface processes and interactions (e.g. a list of forms that should be displayed after each other). Not so important, rarely used.

Layers and tiers

- Layers and tiers, that is logical or physical separation
- Layer
  - Logical element
  - We design it first. We have been talking about it until now.
  - It is worth separating them into different modules (e.g. .NET assemblies (dll) or Java packages)
- Tier
  - Physical element, they run on different computers, network communication
- Don't mix them

Pipes and filters

Pipes and filters

- Architectural pattern that creates a structure for **data stream** processing systems
- Every step of the processing of the stream is implemented by a **filter**
- Filters are connected by **pipes**
- Filters can adequately be **combined**, different kinds of systems can be created

Pipes and filters

- A network of the followings:
  - Data source
  - Data sink
  - Filters
  - Pipes

Examples

- Video stream processing
  - Contrast adjustment
  - Lightness adjustment
  - Color depth adjustment
  - Resizing
  - Changing the format

Filters

- Performs some transformation on the data
  - The input is a pipe
  - The output is a pipe
- Active filter
  - Waiting in a loop (thread)
  - If there is data on the input pipe it pulls that
  - It pushes the result to the output pipe
- Passive filter
  - The input pipe pushes the data into the filter
  - The output pipe pulls the result out from the filter

Pipes

- Transfers data
- May buffer data*: FIFO
- Can do synchronization between filters (only if the filter is active)

Scenarios

1. Data source driven (passive filters)

Filter pseudo-code:

```
void Write(Data data)
{
    Data processedData = ProcessData(data);
    nextFilter.Write(processedData);
}
```

2. Data sink driven (passive filters)

Filter pseudo-code:

```
Data Read()
{
    Data data =  prevFilter.Read();
    Data processedData = ProcessData(data);
    return processedData;
}
```

Scenarios

2. Active filters
   - All filters execute in parallel
   - All filters are waiting for some input data in their input pipes
   - Pushing the result to the output pipe does not block just when the output pipe if full

Filter pseudo-code:

```
void Run()
{
    Data data;
    while(data = inputPipe.Read())
    {
        Data processedData = ProcessData(data);
        outputPipe.Write(processedData);
    }
}
```

There are other scenarios as well!

Advantages

- Filters can adequately be combined
- Filters can be replaced
- Filters can be reused
- Filters can work in parallel (in case of active filters)
- Fast prototyping by the help of existing filters, e.g. Unix sed and awk. The performance-critical ones can be replaced with faster custom ones.

Difficulties

- Error handling
  - What should happen if there is an error in the middle of the execution? How/where to continue?
- Data transformation overhead
  - E.g. Unix pipe – float numbers have to be converted to ASCII and then back to float for every pipe
    - As the line break fragments the dataflow

The MVC architecture

The Model-View-Controller architecture

- For applications with user interfaces
- It first appeared in SmallTalk (~70s Xerox)
- Trygve Reenskaug

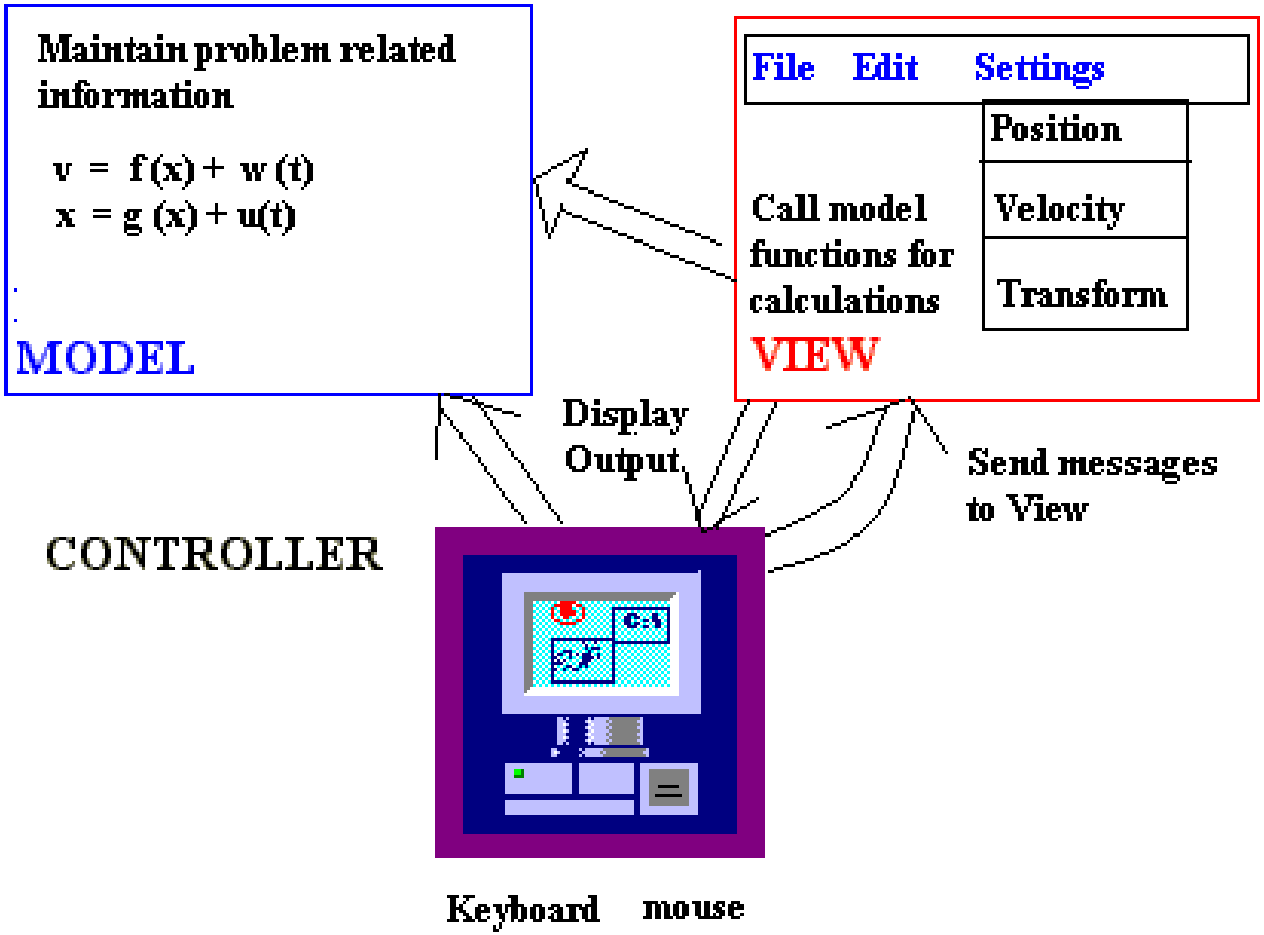
The MVC architecture

Basic concept: don't hardwire the logic of the application into the user interface (GUI)

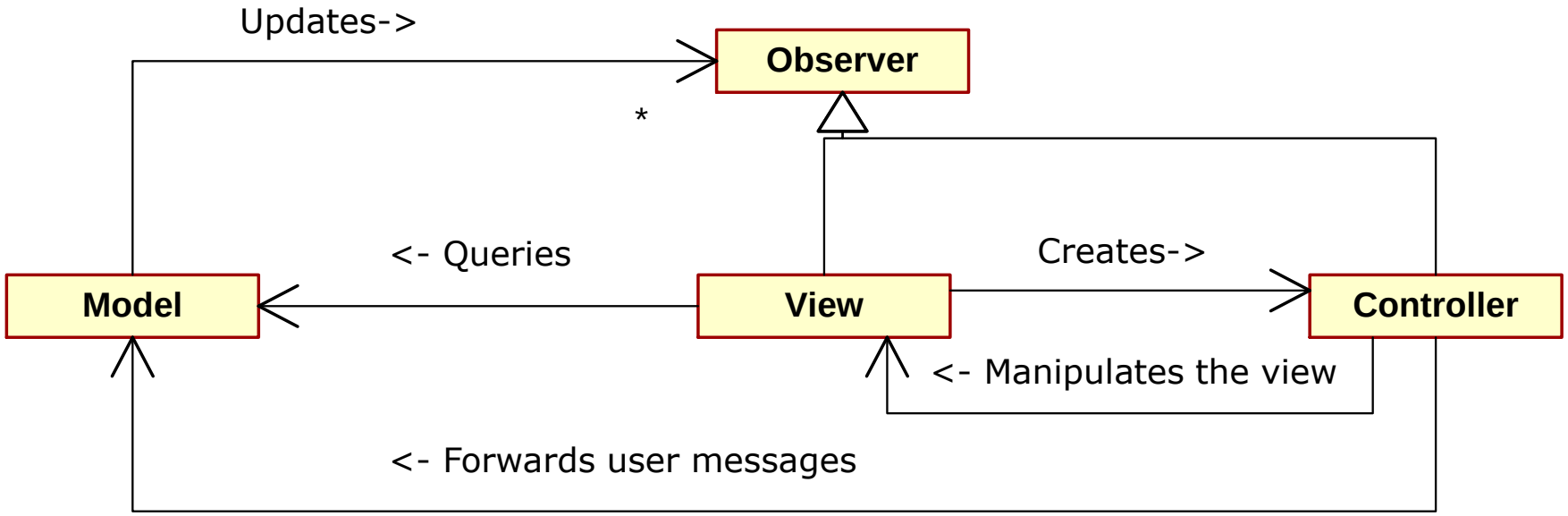
MVC

- **Model** – application logic
- **View** – presentation
- **Controller** – interaction: communication with the user

The MVC architecture



MVC class diagram



MVC class diagram

- **Model**: contains the data (as member variables) and operations that manipulate the data (e.g. Save, Clear, etc).
- **View**: Presents the data. The data is retrieved from the model (the view does not store it). It may contain data that is specific to the view (e.g. the zoom level).
- **Controller**: Handles user interactions. Methods for handling the events of the keyboard, the mouse and the menu, …. These operations usually call a method in the model or in the view (if it is something view specific, e.g.: setting the zoom level).
- **Observer:** if a controller modifies the model all the views will be notified about that so that they can refresh themselves based on the actual state of the model. This guarantees that all the views will be consistent with the model and the controllers can also enable or disable certain user commands (e.g. File/Close menu should be disabled if there is no open document). The model stores a common list of Observer type references for the views and the controllers.

Dynamic behavior

- Steps
  - The Controller handles the events
  - The Controller notifies the Model
  - The Model notifies the views about the change.
  - The Views query the state of the Model and refresh their user interfaces

- Initialization (don't learn by heart)
  - The Main application creates the Model
  - The Main application creates the View
  - The View creates the Controller
  - The View attaches itself to the Model

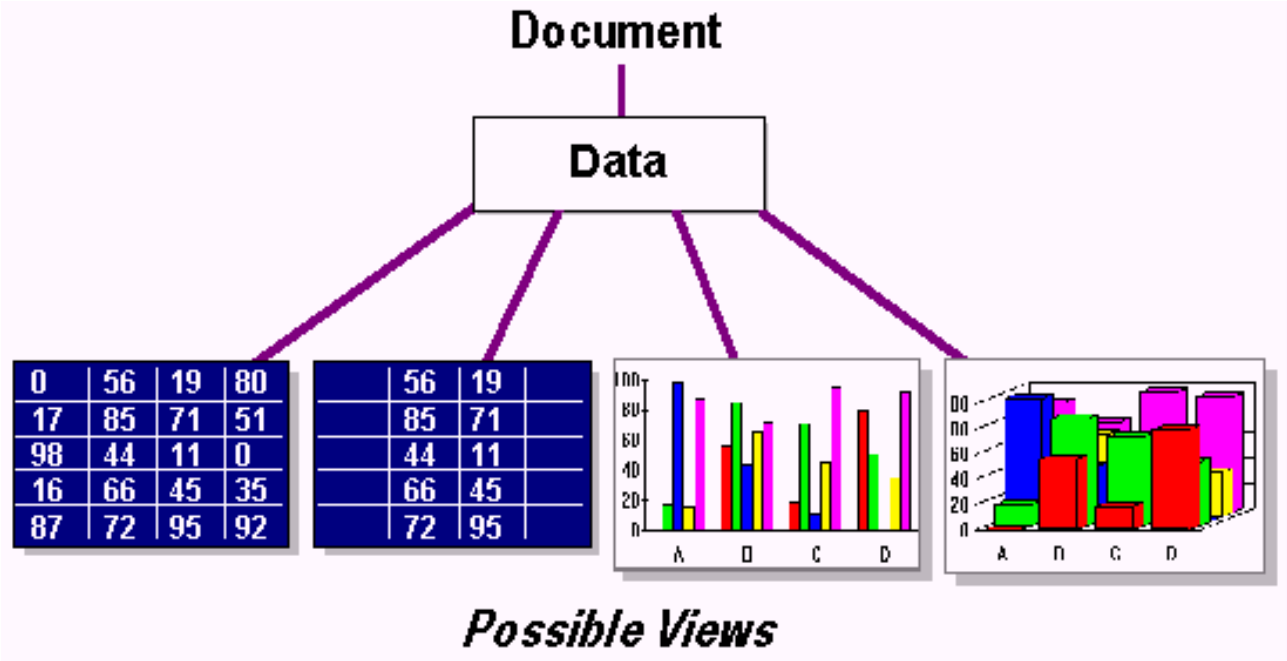
MVC, evaluation

- Advantages
  - Many views for the same data
  - Synchronized views
  - Views can be replaced and extended
  - The Model can easily be tested (e.g. unit test)
  - It is commonly used as a framework for apps because of support from IDEs
- Disadvantages
  - The application will be more complex
  - Many unnecessary notifications/refreshes
  - The View and the Controller can often not be separated. Solution: Combine the View and the Controller together:
    - Document-View architecture (MFC)

Document-View

Document-View

- A variation of the MVC architecture
- Microsoft applies it in MFC (Microsoft Foundation Classes)

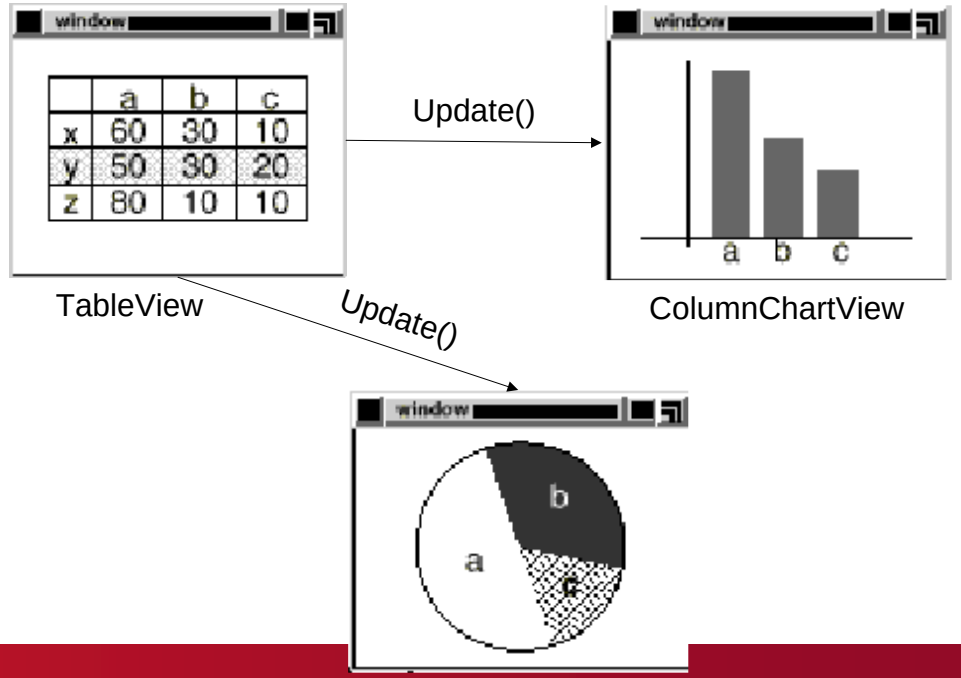


Document-View

- Document
  - Stores and manages data (this would be the Model in the MVC architecture)
- View
  - Presenting data
    - Based on the document
  - Handling user interactions
    - And usually forwards it to the Document
  - In Windows applications a view is usually a window/tab

Multiple documents and views

- Usually supports multiple documents
  - Eg. Chrome tabs, Word documents, Excel spreadsheets
- Many different Views can be attached to the same Document.
  - Crucially, the application must ensure that all views see a consistent document. For example, when the user changes data in one view, the others must update. How to do it?
    - Simple function calls (each view has a reference to all others)?

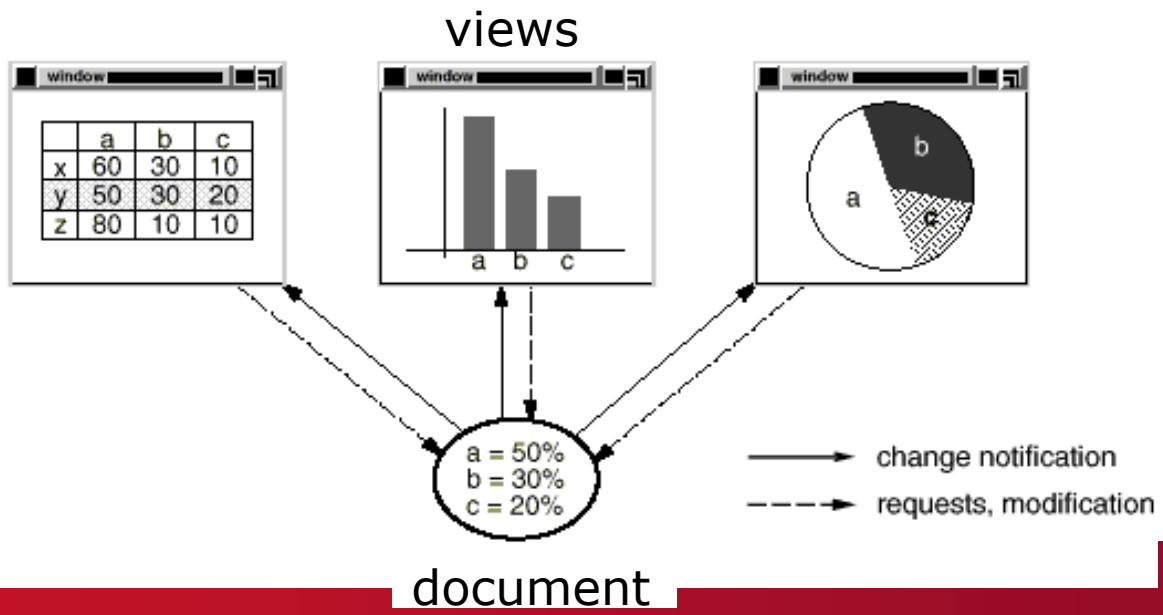


Updating views

- The disadvantage of direct function calls
  - It introduces unwanted dependencies between classes
    - Eg. TableView depends on ColumnChartView and PieChartView
  - If we add a new view, we must change all others as well
  - The business logic is not reusable because it is tied to the presentation.
    - It would be nice if the BL did not contain references to concrete view classes. That would enable reusability.
  - It is hard to maintain, develop, reuse because of the strong ties between its classes

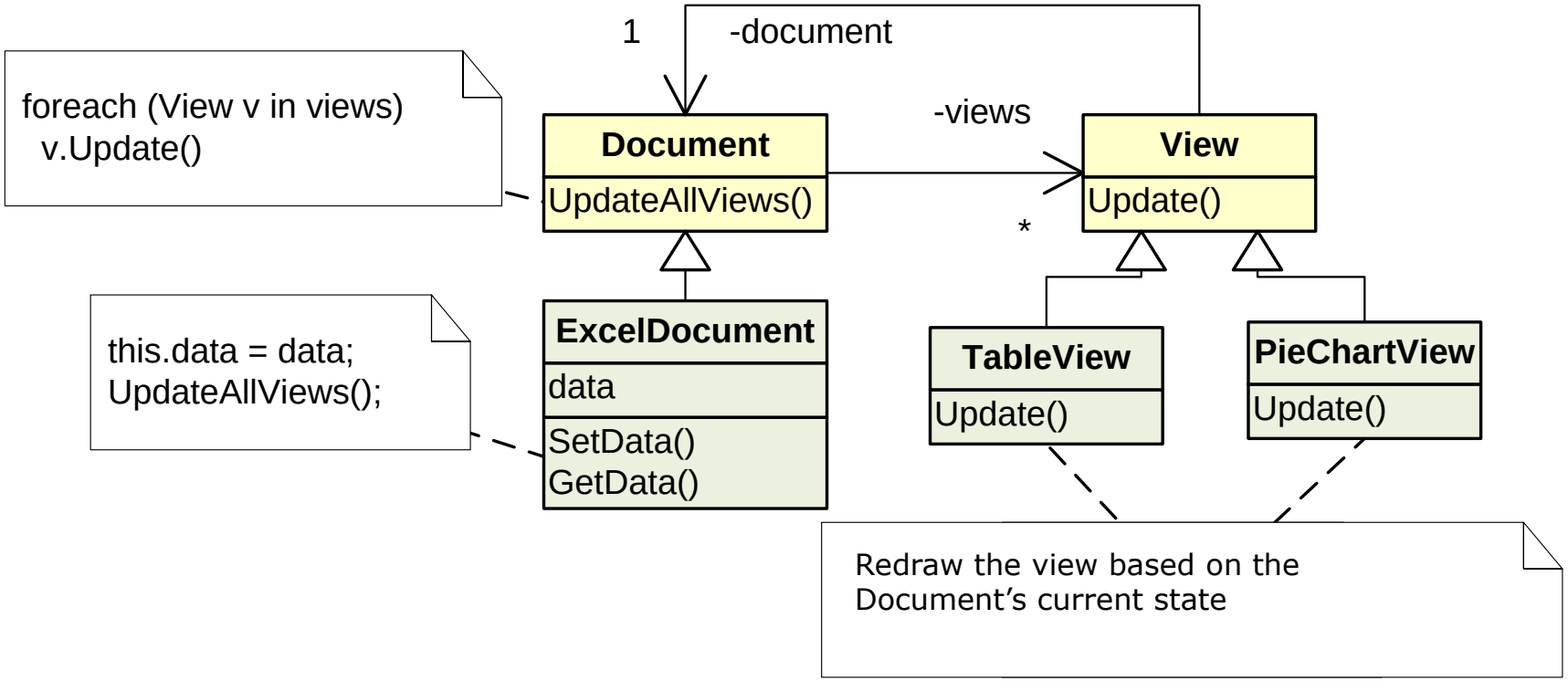
Updating views

- The good solution
  - Have all data and operations on the data into the Document class
  - Have views register themselves in the Document class
  - If a view changes the Document, it is the Document's responsibility to update all registered views. Observer pattern like in MVC.
  - When a view is notified of the change, it reads the state of the Document and redraws itself
  - The document handles views through a IView interface or abstract base class. This avoids a strong tie between the classes.

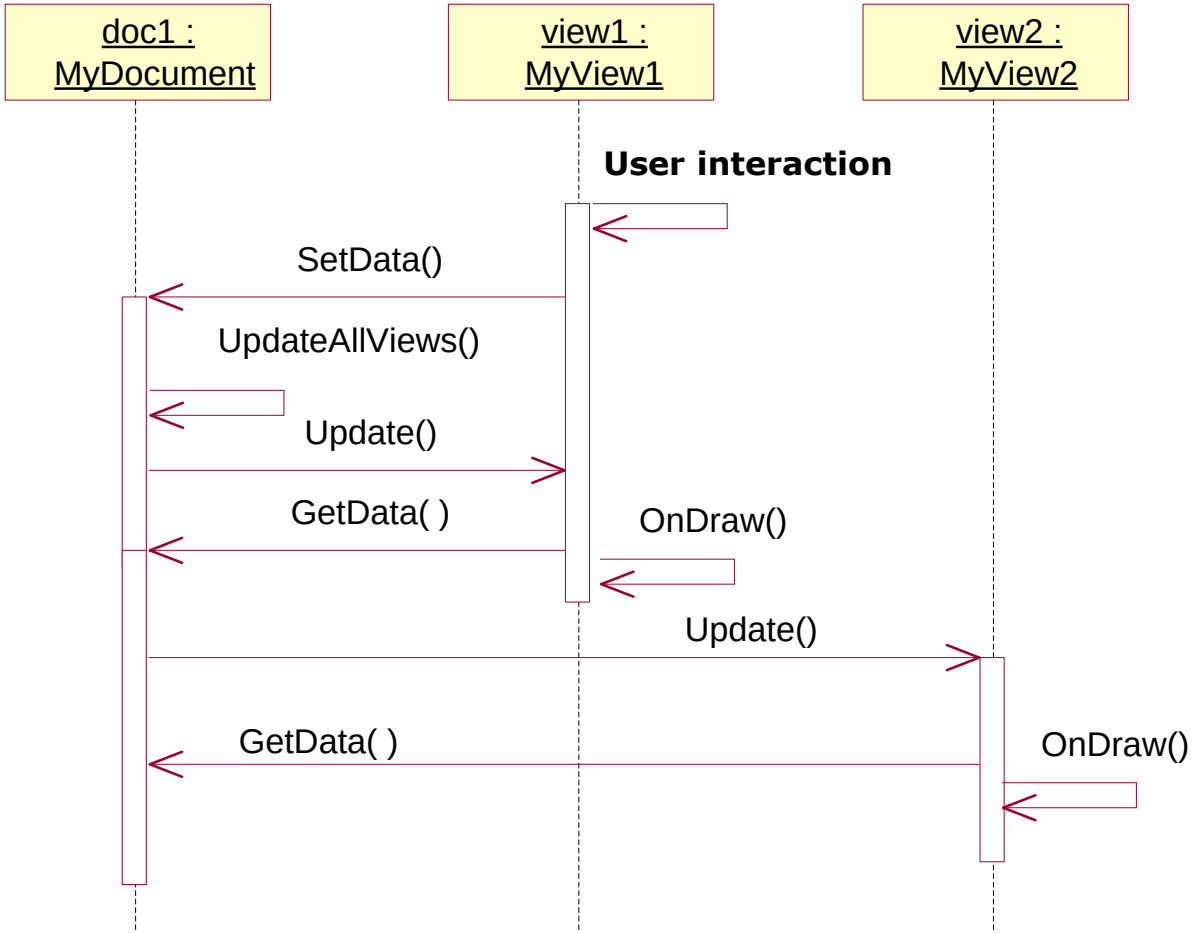


Class diagram of Document-View

- Example without the view registration functions (Excel)



Dynamic behavior of the document-view architecture



Thoughts on Document-View

- Advantages
  - The model (document) only has a list of generic Views, therefore it is not dependent on how the actual views are implemented.
  - The model is reusable!
  - Updating views to keep a consistent view of the data happens through a relatively simple mechanism.
  - We can easily add new views. Neither the model, nor existing views must be modified.
- Disadvantages
  - Increased complexity

References

- Martin Fowler: Analysis Patterns
- Frank Buschmann, ...: Pattern-Oriented Software Architecture Volume 1: A System of Patterns