Osnovi softverskog inženjerstva

P-09: Projektovanje softvera

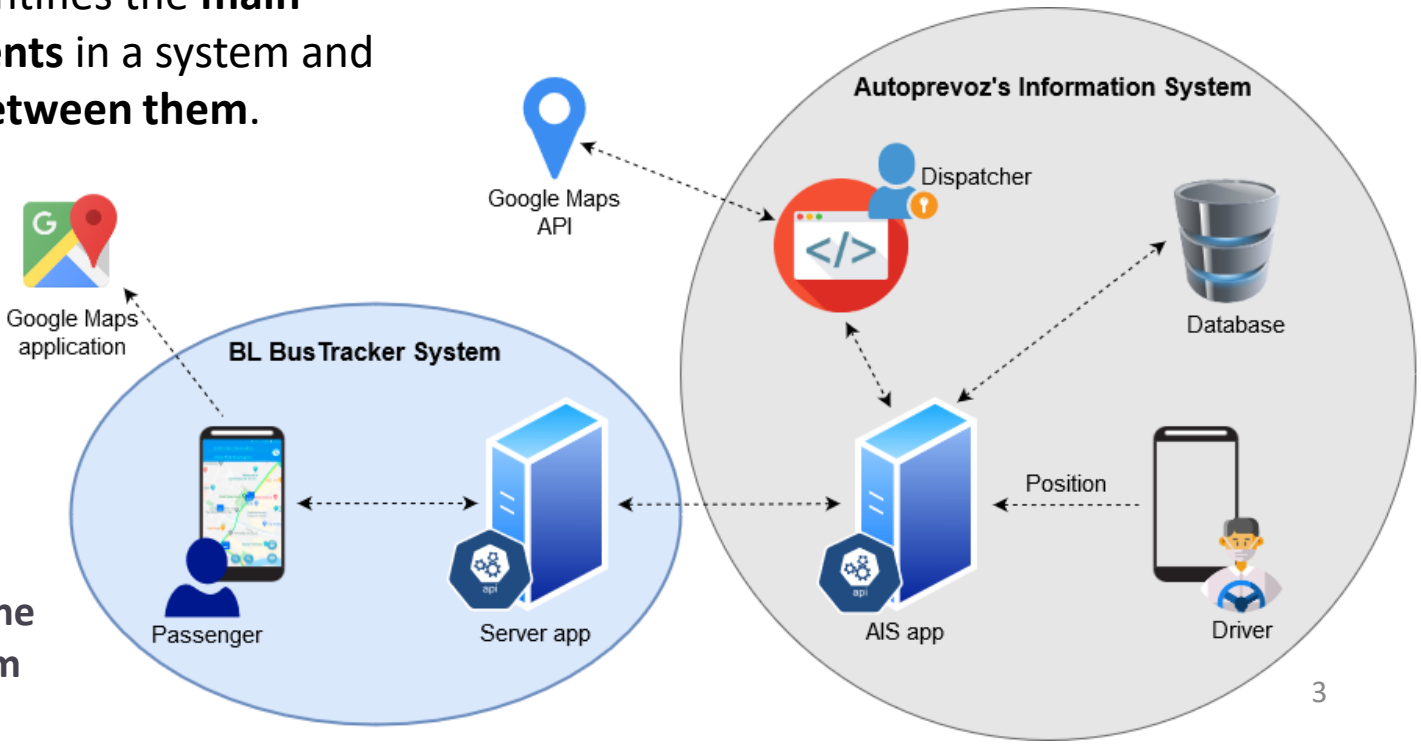
2024

Topics covered

- ✧ Architectural design decisions
- ✧ Architectural views
- ✧ Architectural patterns
- ✧ Application architectures
- ✧ Detailed software design / Reuse principles
- ✧ Design patterns

Architectural design

- **Architectural design** is concerned with:
  - understanding **how a software system should be organized** and
  - **designing the overall structure** of that system.
- Architectural design is the **critical link between design and requirements engineering** – it identifies the **main structural components** in a system and the **relationships between them**.
- The **output** of the architectural design process is an **architectural model** that describes **how the system is organized** as a **set of communicating components**.



The architecture of the BL BusTracker system

Architectural abstraction

- **Architecture in the small**
  - Concerned with the **architecture of individual programs**.
  - At this level, we are concerned with the way that an **individual program is decomposed into components**.
- **Architecture in the large**
  - Concerned with the **architecture of complex enterprise systems** that include other systems, programs, and program components.
  - These enterprise systems are **distributed over different computers**, which may be owned and managed by different companies.

Benefits of explicit architecture

- **Stakeholder communication**
  - Architecture may be used as a focus of discussion by system stakeholders.
- **System analysis**
  - Means that analysis of whether the system can meet its non-functional requirements is possible.
- **Large-scale reuse**
  - The architecture may be reusable across a range of systems.
  - Product-line architectures may be developed.

Agility and architecture

- It is generally accepted that an **early stage of agile processes** is to **design an overall systems architecture**.
- **Refactoring the system architecture** is usually **expensive** because it affects so many components in the system

Architectural representations

- **Simple, informal block diagrams** showing entities and relationships are the most frequently used method for documenting software architectures.
- But these have been criticised because they **lack semantics**, do not show the types of relationships between entities nor the visible properties of entities in the architecture.
- Depends on the use of architectural models. The requirements for model semantics depends on how the models are used.

Box and line diagrams

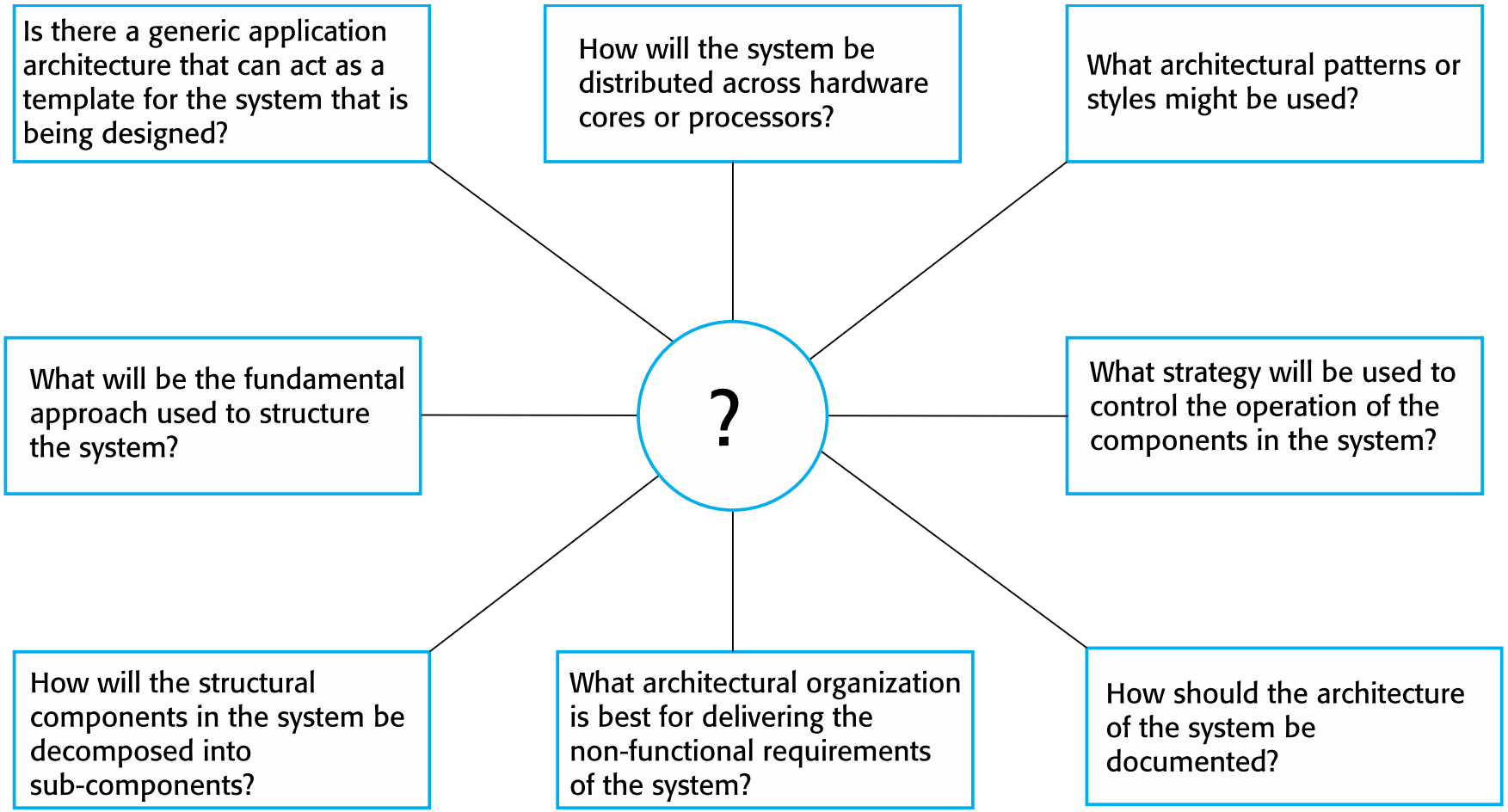
- **Very abstract** – they do not show the nature of component relationships nor the externally visible properties of the sub-systems.
- However, useful for communication with stakeholders and for project planning.

Use of architectural models

- **As a way of facilitating discussion about the system design**
  - A **high-level architectural view** of a system is **useful for communication with system stakeholders** and **project planning** because it is **not cluttered with detail**.
  - **Stakeholders can relate to it and understand an abstract view of the system**. They can then discuss the system as a whole without being confused by detail.
- **As a way of documenting an architecture that has been designed**
  - The aim here is **to produce a complete system model** that **shows the different components** in a system, **their interfaces** and their **connections**.

Architectural design decisions

- **Architectural design is a creative process** so the **process differs depending on the type of system being developed**.
- However, a **number of common decisions span all design processes** and these **decisions affect the non-functional characteristics** of the system.



Architecture reuse

- Systems in the **same domain** often have **similar architectures** that reflect domain concepts.
- **Application product lines** are built around a **core architecture** with **variants that satisfy** particular **customer requirements**.
- The **architecture of a system** may be **designed around one of more architectural patterns or 'styles'**.
  - These capture the essence of an architecture and **can be instantiated in different ways**.

Impact to sys. characteristics

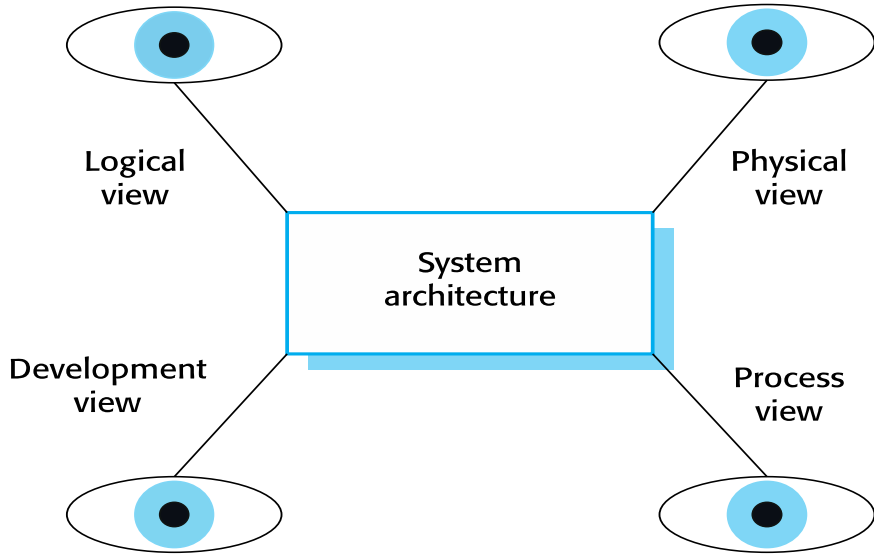
- **Performance**
  - Localise critical operations and minimise communications.
  - Use large rather than fine-grain components.
- **Security**
  - Use a layered architecture with critical assets in the inner layers.
- **Safety**
  - Localise safety-critical features in a small number of sub-systems.
- **Availability**
  - Include redundant components and mechanisms for fault tolerance.
- **Maintainability**
  - Use fine-grain, replaceable components.

Architectural views

Architectural views

- **What views or perspectives are useful** when designing and documenting a system's architecture?
- **What notations** should be used for describing architectural models?
- **Each architectural model only shows one view or perspective** of the system.
  - It might show:
    - how a system is decomposed into modules,
    - how the run-time processes interact or
    - the different ways in which system components are distributed across a network.
  - For both design and documentation, **we usually need to present multiple views of the software architecture**.

4 + 1 view model of software architecture



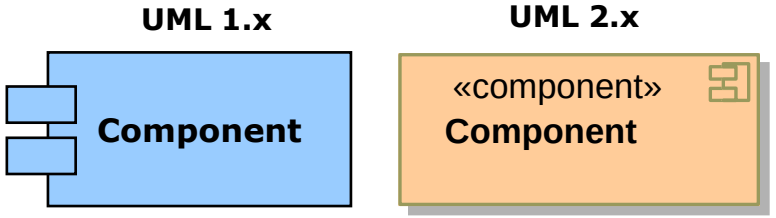
- **Logical view** - shows the key abstractions in the system as objects or object classes.
- **Process view** - shows how, at run-time, the system is composed of interacting processes.
- **Development view** - shows how the software is decomposed for development.
- **Physical view** - shows the system hardware and how software components are distributed across the processors in the system.
- Related using **use cases** or scenarios (+1)

Representing architectural views

- Two UML notations for representing software architectures:
- **Component diagram** – represents the logical architecture of the software system (design-time / development view of the "4+1" model)
- **Deployment diagram** – represents the deployment of the software artifacts on the hardware nodes (run-time / physical view of the "4+1" model)

Component diagram

- **"Software wiring diagram" – depicts software components and their connections**

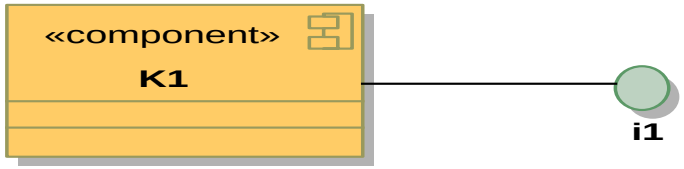


  - **Component connections = dependency**
  - **Interface** = a set of operations provided/required by some component

Representing architectural views

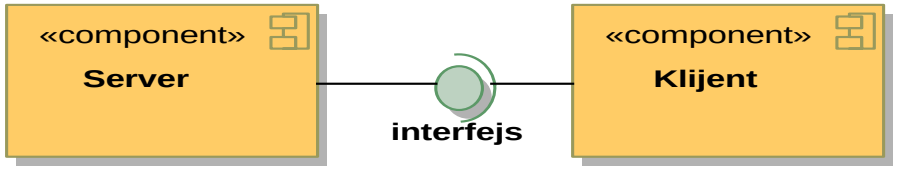
Component diagram

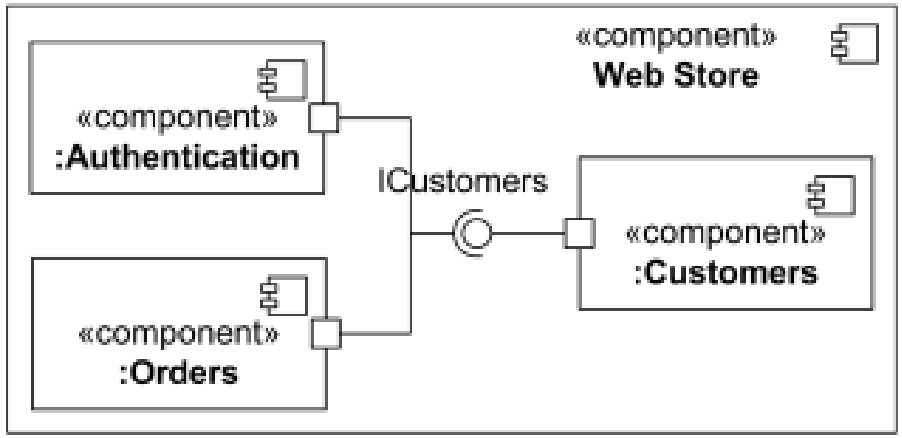
Provided interface



Required interface

Examples:

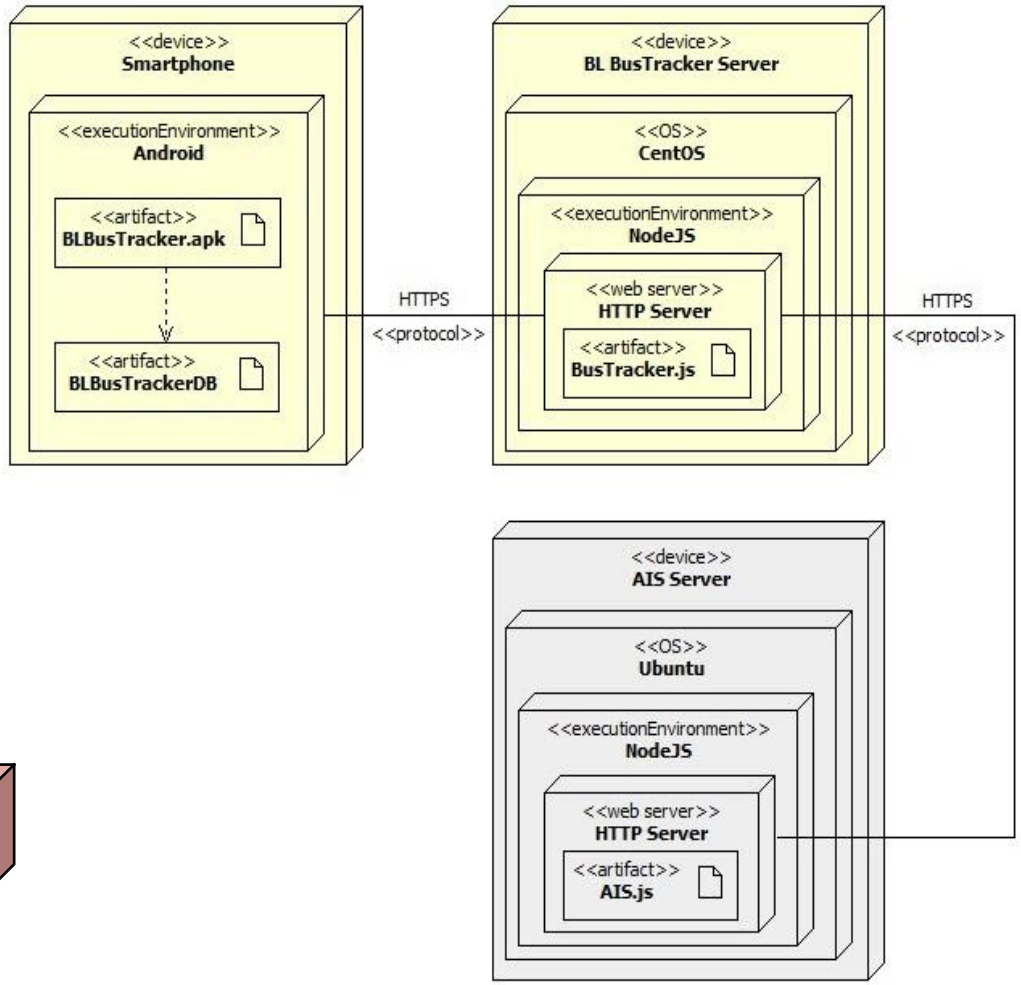


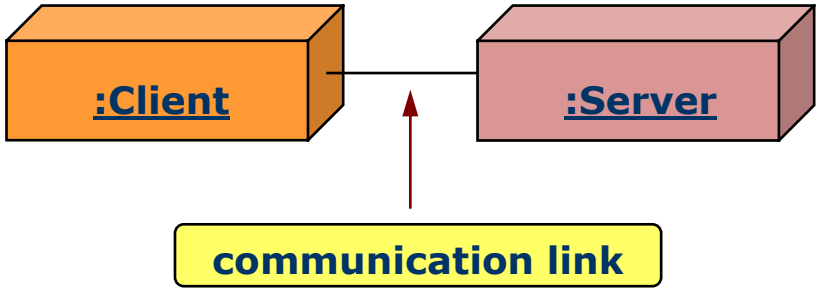


Representing architectural views

Deployment diagram

- **represents the deployment of the software artifacts on the hardware nodes (run-time)**
- **Contains:**
  - **nodes**
    - abstraction of **hardware nodes** (e.g. server) and **execution environments** (e.g. operating system)
  - **communication links**
    - connections between nodes





Architectural patterns

Architectural patterns / styles

- Patterns are a **means of representing, sharing and reusing knowledge**.
- An architectural pattern is a **stylized description of good design practice**, which **has been tried and tested in different environments**.
- Patterns should **include information** about **when** they are and **when** the are **not useful**.
- Patterns may be represented using **tabular and graphical descriptions**.

- **Some important architectural patterns:**
  - **Client-Server**
  - **MVC**
  - **Layered**
  - **Repository**
  - **Pipe and filter**

The Client-Server architectural pattern

- **Distributed system model** which shows how data and processing is distributed across a range of components.
  - Can be implemented on a single computer.
- **Set of stand-alone servers** which **provide specific services** such as printing, data management, etc.
- **Set of clients** which **call on these services**.
- Network allows clients to access servers.
- In a client–server architecture, the **functionality of the system is organized into services**, with **each service delivered from a separate server**.
- **Clients are users of these services** and access servers to make use of them.

When used

- **Used when data in a shared database has to be accessed from a range of locations.**
- Because servers can be replicated, may also be used when the load on a system is variable.

Advantages

- The principal advantage of this model is that **servers can be distributed across a network**.
- **General functionality** (e.g. a printing service) **can be available to all clients** and **does not need to be implemented by all services**.

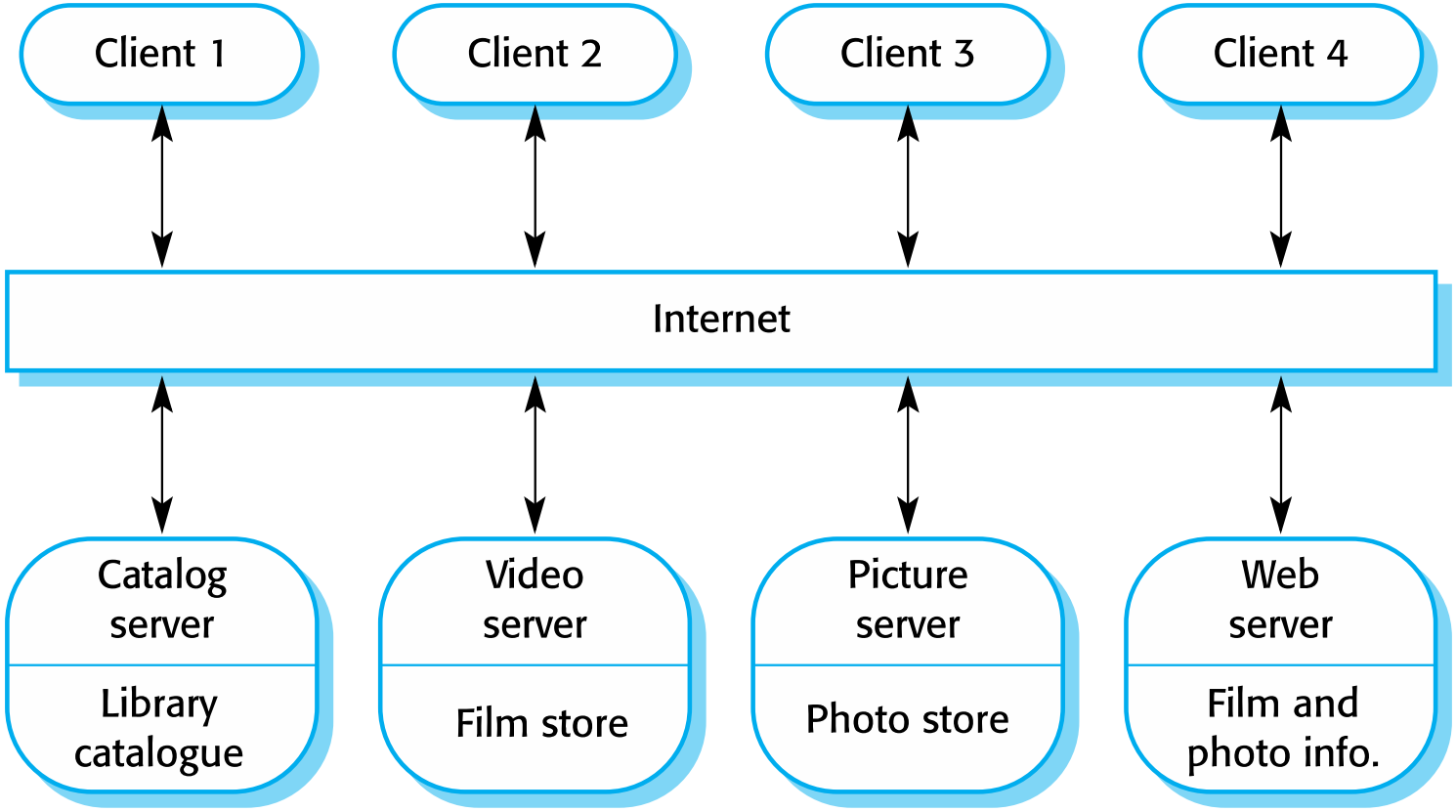
Disadvantages

- **Each service is a single point of failure** so susceptible to denial of service attacks or server failure.
- **Performance may be unpredictable** because it depends on the network as well as the system.
- **May be management problems** if servers are owned by different organizations.

The Client-Server architectural pattern

A client–server architecture for a film library

Figure shows an example of a film and video/DVD library organized as a client–server system.

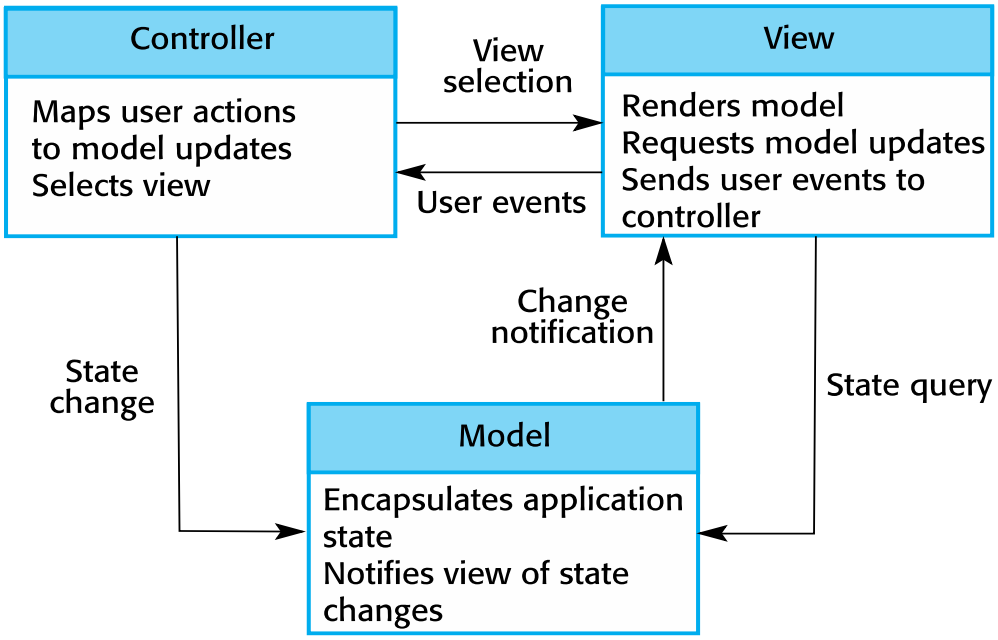


The Model-View-Controller (MVC) pattern

Description

- **Separates presentation** and **interaction** from the **system data**.
- The system is structured into **three logical components** that interact with each other.
- The **Model** component **manages the system data** and **associated operations on that data**.
- The **View** component **defines and manages how the data is presented** to the user.
- The **Controller** component **manages user interaction** (e.g., key presses, mouse clicks, etc.) and **passes these interactions to the View and the Model**.

The organization of the MVC pattern



The Model-View-Controller (MVC) pattern

When used

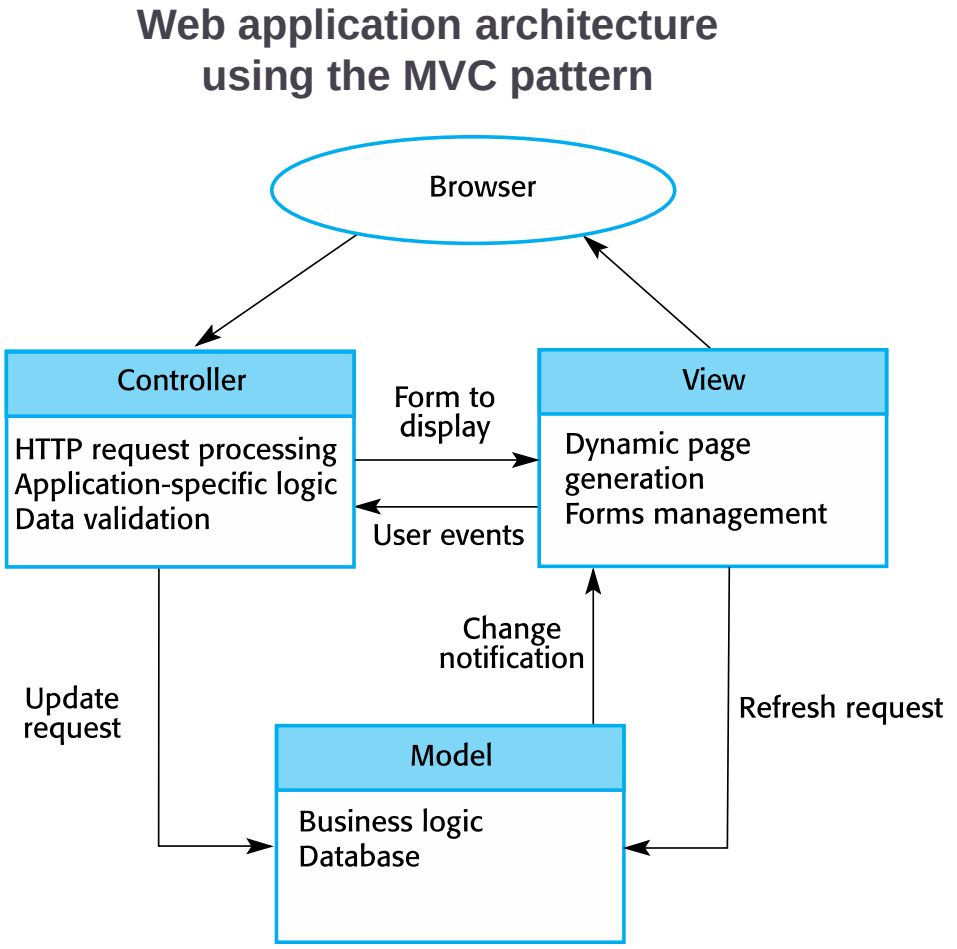
- Used **when there are multiple ways to view and interact with data**.
- Also used **when the future requirements for interaction and presentation of data are unknown**.

Advantages

- **Allows the data to change independently of its representation and vice versa**.
- Supports **presentation of the same data in different ways** with changes made in one representation shown in all of them.

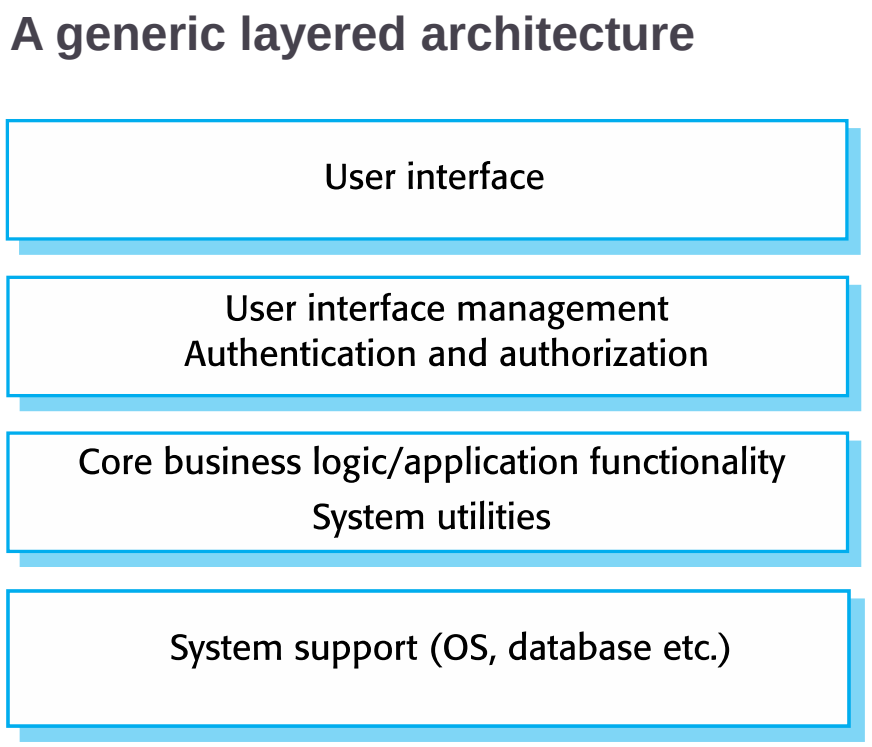
Disadvantages

- **Can involve additional code and code complexity when the data model and interactions are simple**.



The Layered architecture pattern

- Organises the system into a **set of layers** (or **abstract machines**) **each** of which **provides a set of services for the layers above**.
- A **layer provides services to the layer above** it so **the lowest-level layers represent core services** that are likely to be used throughout the system.
- **Supports the incremental development** of sub-systems in different layers.
- **When a layer interface changes, only the adjacent layer is affected**.



The Layered architecture pattern

When used

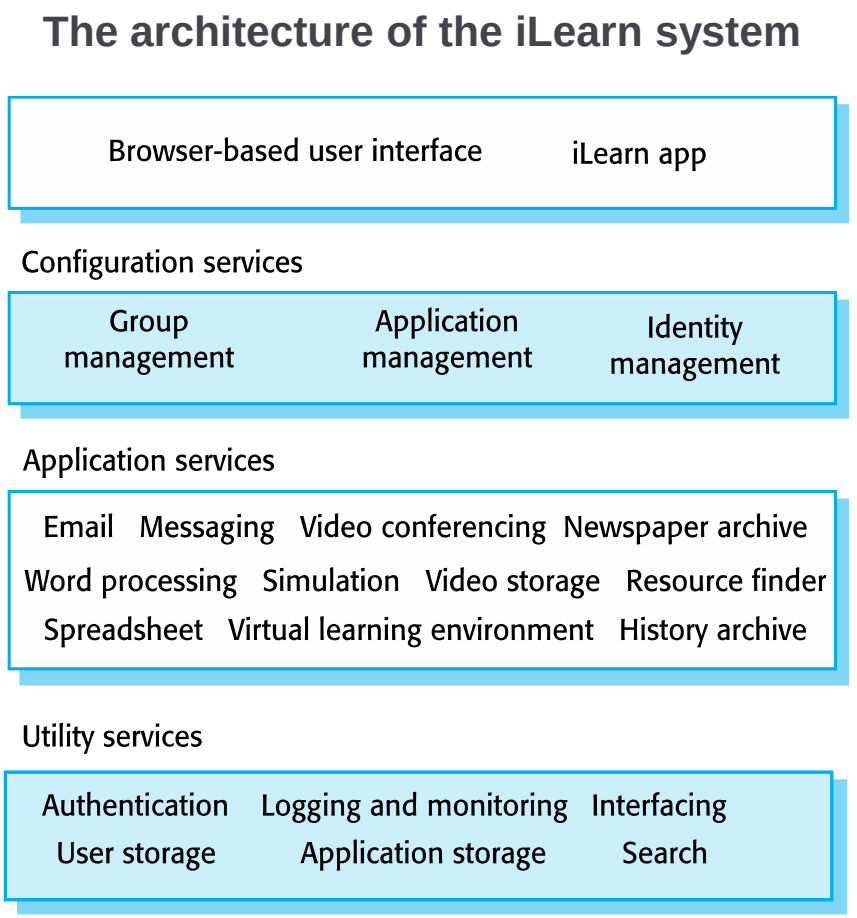
- used **when building new facilities on top of existing systems**;
- **when the development is spread across several teams with each team responsibility for a layer of functionality**;
- **when there are requirements for multi-level sec**.

Advantages

- **Allows replacement of entire layers so long as the interface is maintained**.
- **Redundant facilities** (e.g., authentication) **can be provided in each layer to increase the dependability of the system.**

Disadvantages

- In practice, **providing a clean separation between layers is difficult and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it**.
- **Performance can be a problem** because of multiple levels of interpretation of a service request as it is processed at each layer.



The Repository architectural pattern

- **Sub-systems must exchange data**. This may be done in two ways:
  - **Shared data is held in a central database or repository and may be accessed by all sub-systems**;
  - Each sub-system maintains its own database and passes data explicitly to other sub-systems.
- **When large amounts of data are to be shared**, **the repository model** of sharing is most commonly used a this is an efficient data sharing mechanism.
- **All data in a system is managed in a central repository that is accessible to all system components**.
- Components do not interact directly, only through the repository.

When used

- We should use this pattern **when we have a system in which large volumes of information** generated **that has to be stored for a long time**.
- We may also **use it in data-driven systems** where the **inclusion of data in the repository triggers an action or tool**.

Advantages

- **Components can be independent**—they do not need to know of the existence of other components.
- **All data can be managed consistently** (e.g., backups done at the same time) as it is all in one place.

Disadvantages

- The **repository is a single point of failure** so problems in the repository affect the whole system.
- **May be inefficiencies** in organizing all communication through the repository.
- **Distributing the repository across several computers may be difficult**.

The Repository architectural pattern

A repository architecture for an IDE

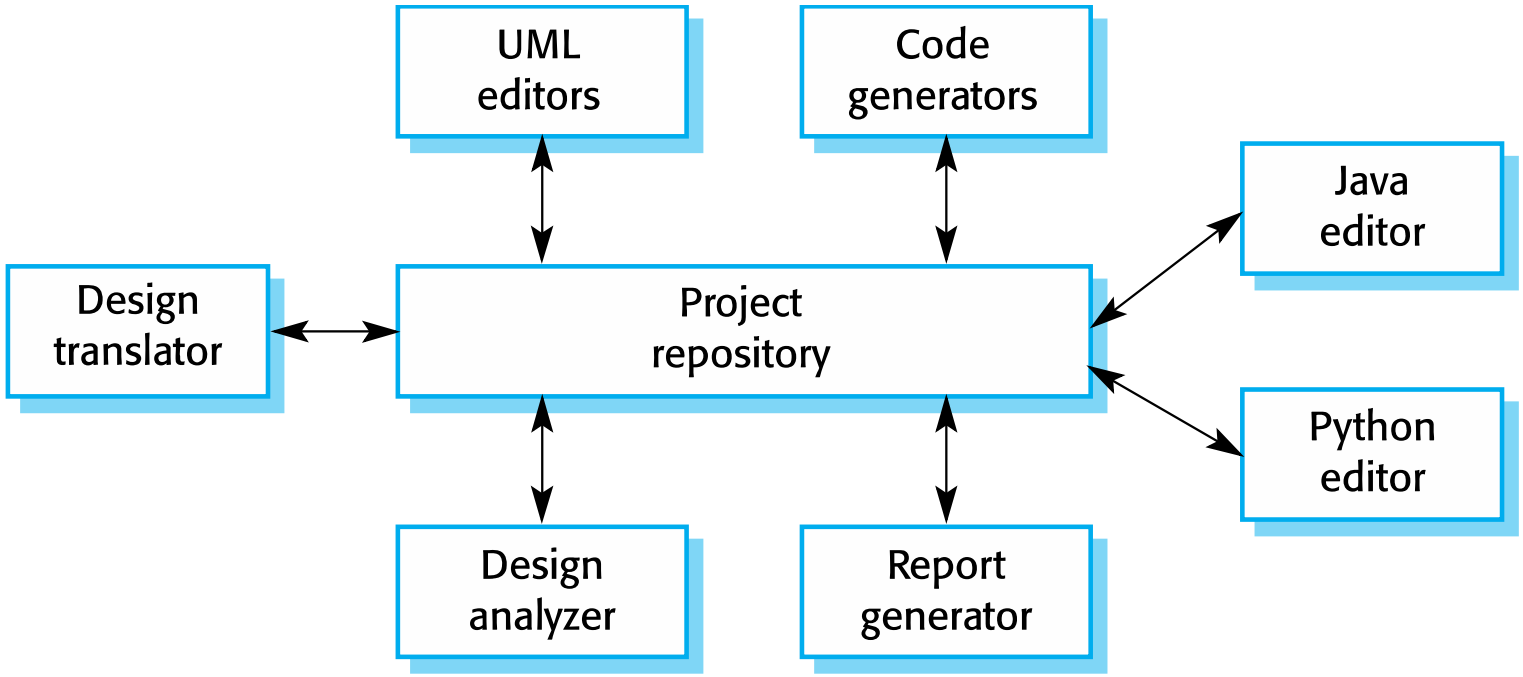


Fig. shows an example of an IDE where the components use a repository of system design information. Each software tool generates information which is then available for use by other tools.

Application architectures

Application types

- **Two very widely used generic application architectures**:
  - **transaction processing systems**
  - **language processing systems.**
- **Transaction processing systems**
  - E-commerce systems;
  - Reservation systems.
- **Language processing systems**
  - Compilers;
  - Command interpreters.

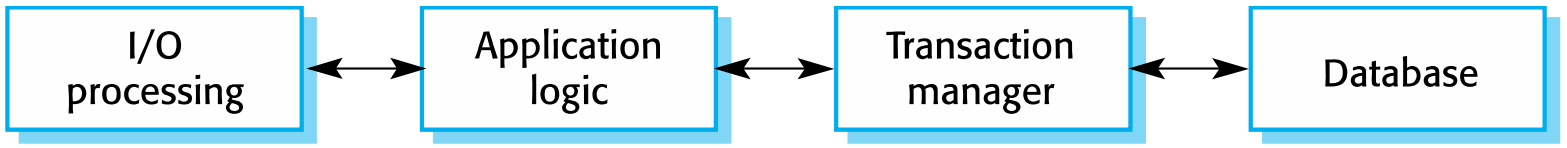
Examples of application types

- **Data processing applications**
  - Data driven applications **that process data in batches** without explicit user intervention during the processing.
- **Transaction processing applications**
  - Data-centred applications **that process user requests** and update information in a system database.
- **Event processing systems**
  - Applications **where system actions depend on interpreting events** from the system's environment.
- **Language processing systems**
  - Applications where the **users' intentions are specified in a formal language** that is processed and interpreted by the system.

Transaction processing systems

- **Process user requests for information from a database or requests to update the database**.
- From a user perspective a **transaction** is:
  - **Any coherent sequence of operations that satisfies a goal;**
  - For example – find the times of flights from London to Paris.
- **Users make asynchronous requests for service which are then processed by a transaction manager**.

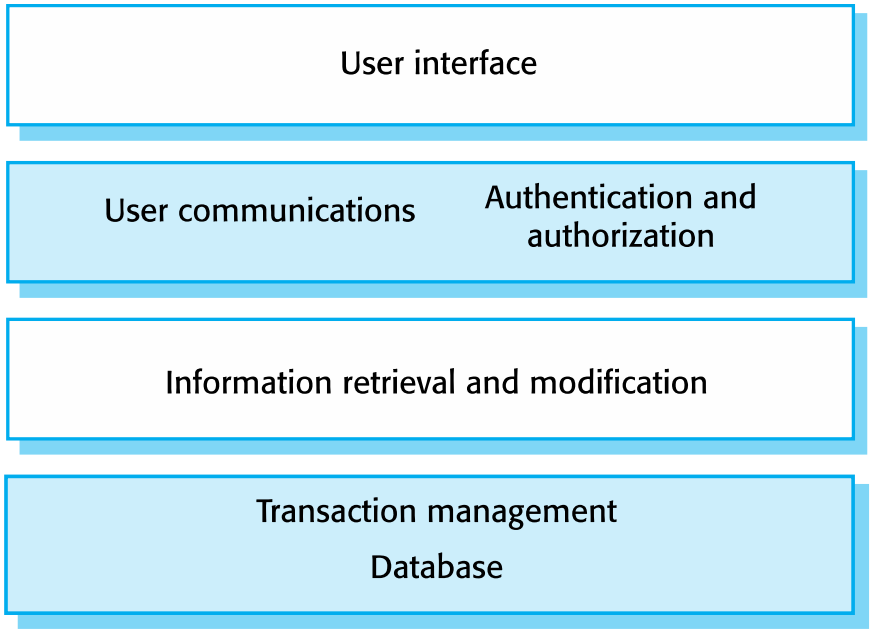
The structure of transaction processing applications



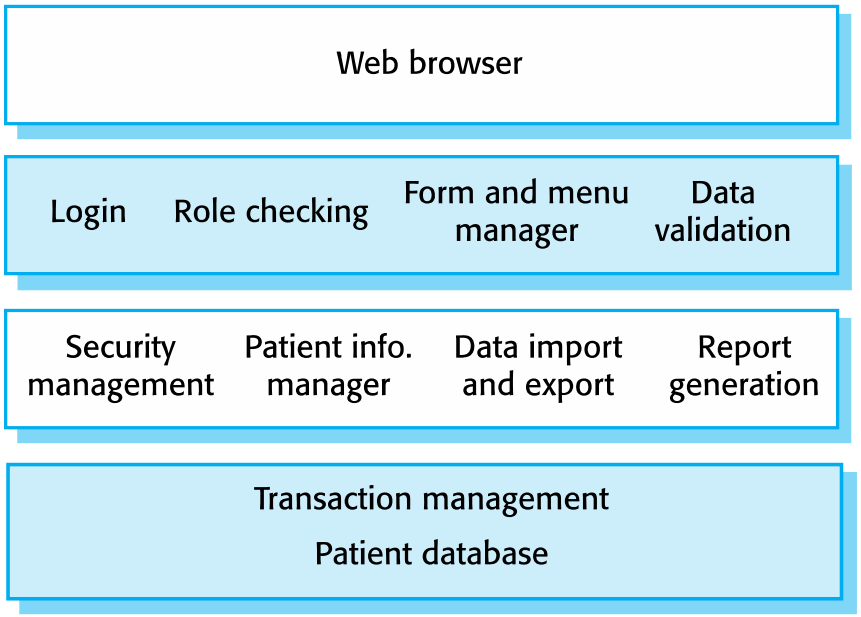
Information systems architecture

- Information systems have **a generic architecture that can be organised as a layered architecture**.
- **These are transaction-based systems** as interaction with these systems generally **involves database transactions**.
- Layers include: The **user interface / User communications / Information retrieval / System database**

Layered information system architecture



The architecture of the Mentcare system



Web-based information systems

- **Information systems are now usually web-based systems** where the user interfaces are implemented using a web browser.
  - For example, **e-commerce systems are Internet-based resource management systems** that:
    - accept electronic orders for goods or services and
    - then arrange delivery of these goods or services to the customer.
  - In an e-commerce system:
    - the application-specific layer includes additional functionality supporting a ‘shopping cart’ in which users can place a number of items in separate transactions,
    - then pay for them all together in a single transaction.

Server implementation

- **These systems are often implemented as multi-tier client/server architectures**
  - The **web server is responsible for all user communications,** with the user interface implemented using a web browser;
  - The **application server is responsible for implementing application-specific logic** as well as information storage and retrieval requests;
  - The **database server moves information to and from the database and handles transaction management**.

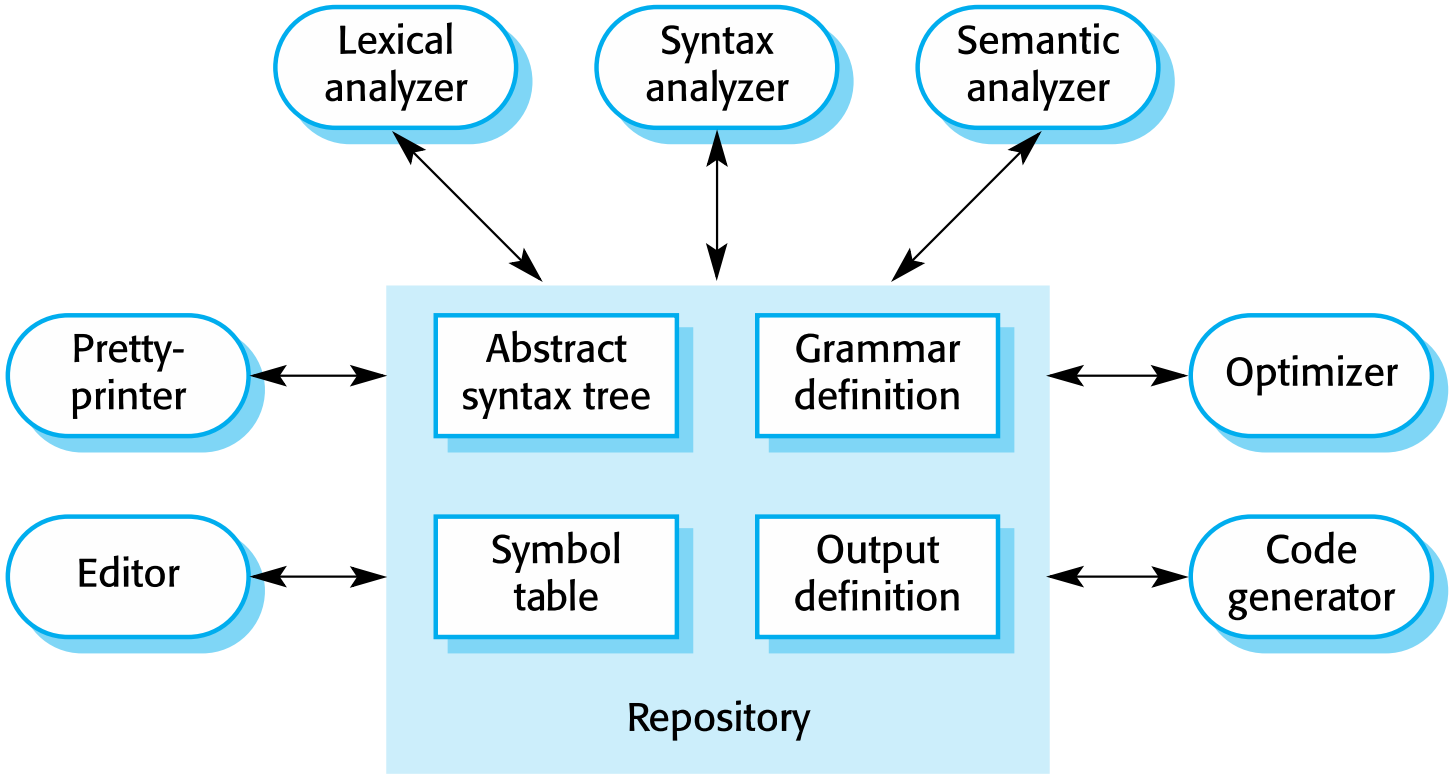
Language processing sys.

- **Accept a natural or artificial language as input** and **generate some other representation** of that/other language.
- **May include an interpreter** to act on the instructions in the language that is being processed.
- Used in situations where the easiest way to solve a problem is to describe an algorithm or describe the system data
  - Meta-case tools process tool descriptions, method rules, etc and generate tools.

Compiler components

- A **lexical analyzer**, which **takes input language tokens** and **converts them to an internal form**.
- A **symbol table**, which **holds information about the names of entities** (variables, class names, object names, etc.) used in the text that is being translated.
- A **syntax analyzer**, which **checks the syntax** of the language being translated.
- A **syntax tree**, which is an **internal structure** representing the program being compiled.
- A **semantic analyzer** that **uses information from the syntax tree and the symbol table** to **check the semantic correctness** of the input language text.
- A **code generator** that 'walks' the syntax tree and **generates abstract machine code**.

A repository architecture for a language processing system



Detailed design / Reuse principles

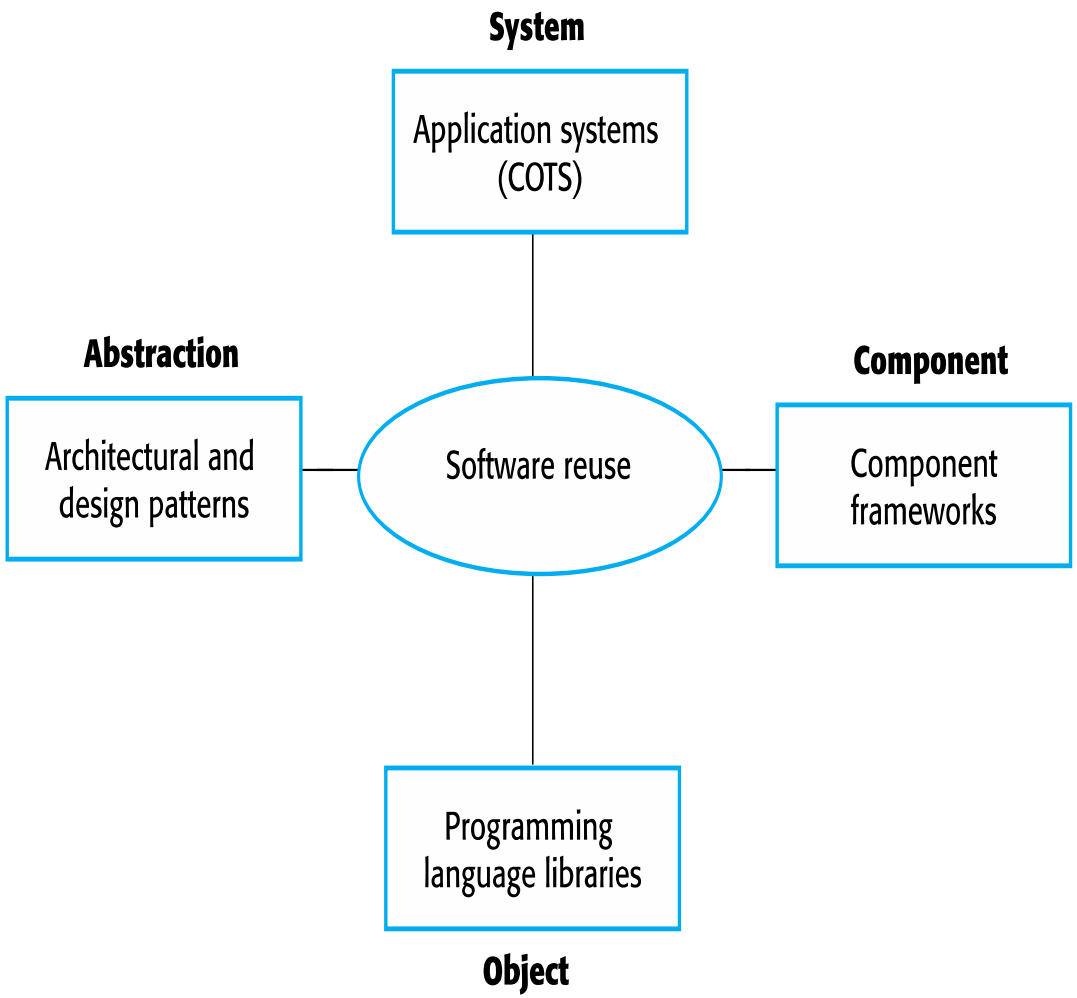
Reuse

- **From the 1960s to the 1990s**, most **new software was developed from scratch**, by writing all code in a high-level programming language.
  - The only significant reuse or software was the **reuse of functions** and **reuse of objects in programming language libraries**.
- **Costs and schedule pressure** mean that this approach became increasingly unviable, especially for commercial and Internet-based systems.
- **An approach to development based around the reuse of existing software emerged and is now generally used for business and scientific software**.

Reuse levels

- **The abstraction level**
  - **We don't reuse software directly** but **use knowledge of successful abstractions in the design** of our software (reuse of architectural styles).
- **The object level**
  - **We directly reuse objects from a library rather than writing the code our own**.
- **The component level**
  - **Components are collections of objects and object classes that we reuse in application systems**.
- **The system level**
  - **We reuse entire application systems**.

Software reuse



Reuse costs

- The **costs of the time spent in looking** for software to reuse and assessing whether or not it meets your needs.
- Where applicable, the **costs of buying the reusable software**. For large off-the-shelf systems, these costs can be very high.
- The **costs of adapting and configuring** the reusable software components or systems to reflect the requirements of the system that we are developing.
- The **costs of integrating** reusable software elements with each other (if we are using software from different sources) and with the new code that we have developed.

Design patterns

Design patterns

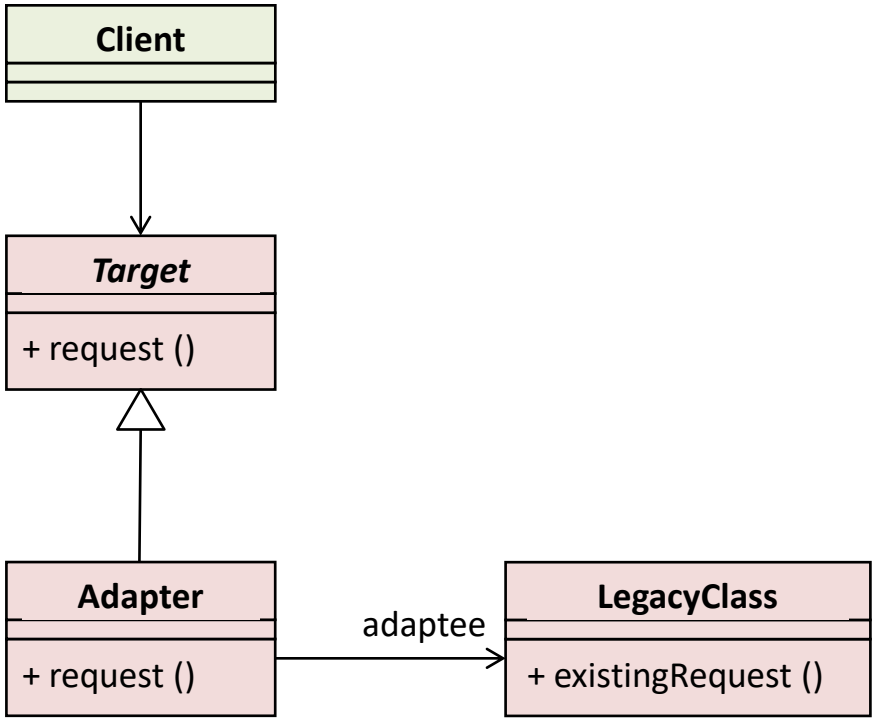
- A design pattern is a **way of reusing abstract knowledge** about a problem and its solution.
- A pattern is a **description of the problem** and the **essence of its solution**.
- It should be **sufficiently abstract to be reused in different settings**.
- Pattern descriptions **usually make use of object-oriented characteristics such as inheritance and polymorphism**.

Pattern elements

- **Name**
  - A meaningful pattern identifier.
- **Problem description**.
- **Solution description**.
  - Not a concrete design but a template for a design solution that can be instantiated in different ways.
- **Consequences**
  - The results and trade-offs of applying the pattern.

The Adapter design pattern

- **Obezbjeđuje interfejs između različitih klasa.**
- **Prilagođava interfejs jedne klase u interfejs kakav očekuje druga klasa, čime omogućava komunikaciju koja inače ne bi mogla da se ostvari.**



Client

- objekat koji zahtijeva interfejs Target

Target

- definiše specifični interfejs koji koristi klasa Client

Adapter

- prilagođava postojeći interfejs LegacyClass (Adaptee) prema interfejsu klijentske klase
- realizacija:
  - **nasljeđivanje interfejsa**: Target ← Adapter
  - **delegacija**: Adapter → Adaptee
    adaptee.existingRequest()

LegacyClass (Adaptee)

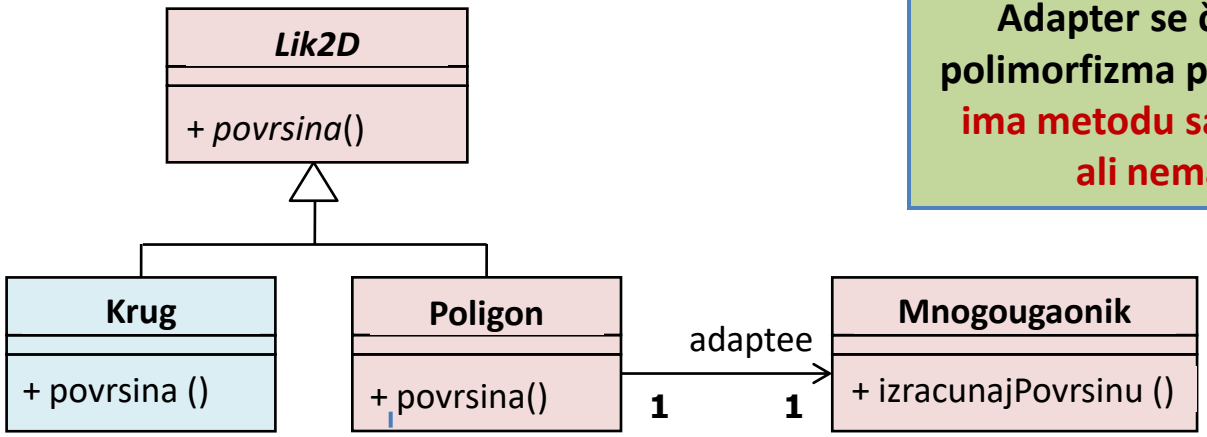
- reprezentuje postojeći interfejs koji treba da se prilagodi

Nefunkcionalni sistemski zahtjevi koji su osnov za ADAPTER:

"mora da komunicira sa postojećim objektom"

The Adapter design pattern

Example:



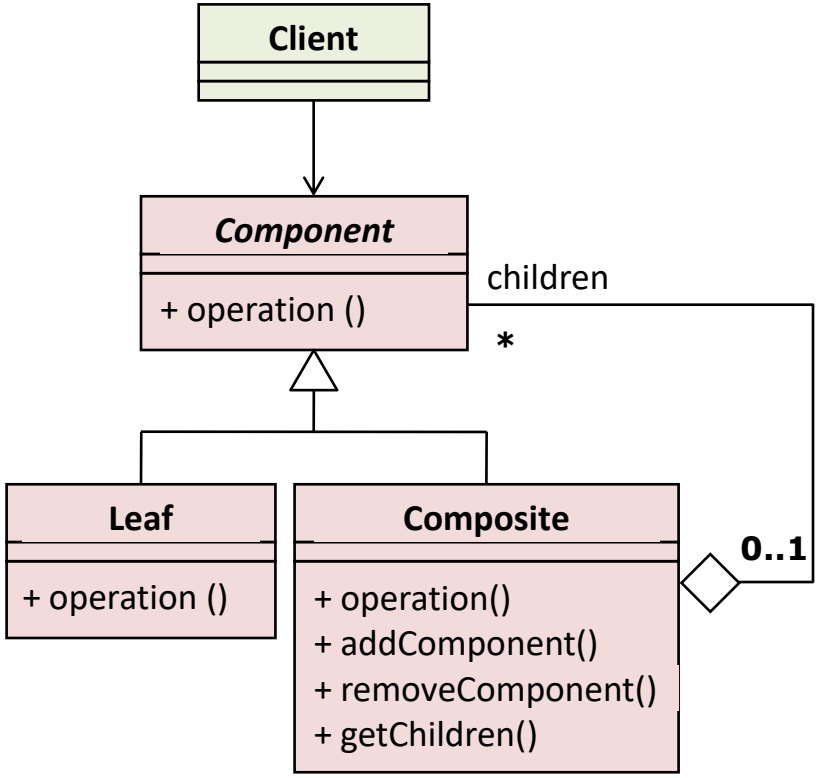
Adapter se često koristi da bi se snaga polimorfizma postigla u postojećoj klasi koja ima metodu sa željenom funkcionalnošću, ali nema adekvatan prototip.

```
double povrsina()
{
    return adaptee.izracunajPovrsinu();
}
```

```
// postojeca klasa
class Mnogougaonik
{
    // ...
    public double izracunajPovrsinu()
    {
        // izracunaj p...
        return p;
    }
}
```

The Composite design pattern

- **Formira složenu hijerarhijsku strukturu proizvoljne širine i dubine.**
- **Omogućava klijentu da na isti način tretira i proste i složene objekte koji su dio složene strukture.**



Component

- deklariše interfejs za objekte u kompoziciji
- implementira ponašanje zajedničko svim klasama
- deklariše interfejs za pristup podelementima
- (opciono) definiše i implementira interfejs za pristup roditeljskom elementu u rekurzivnim strukturama

Leaf

- reprezentuje prosti objekat u kompoziciji
- definiše ponašanje za proste objekte u kompoziciji

Composite

- definiše ponašanje složenih elemenata
- čuva podelemente
- implementira operacije za podelemente iz interfejsa klase Component

Client

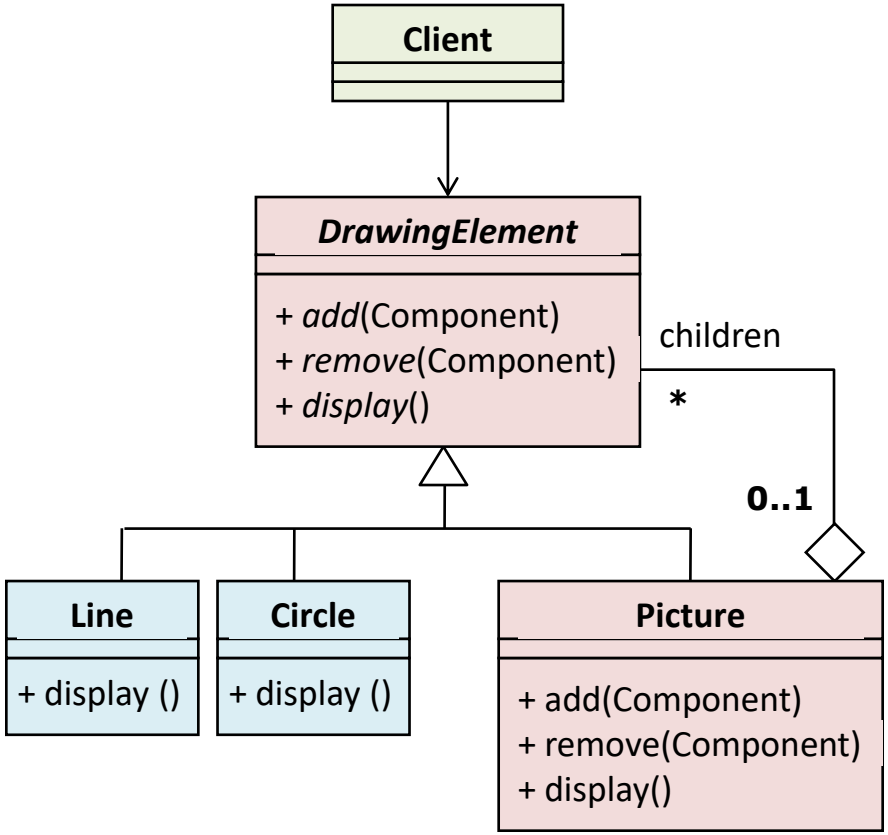
- manipuliše objektima iz kompozicije kroz interfejs klase Component

Nefunkcionalni sistemski zahtjevi koji su osnov za COMPOSITE:

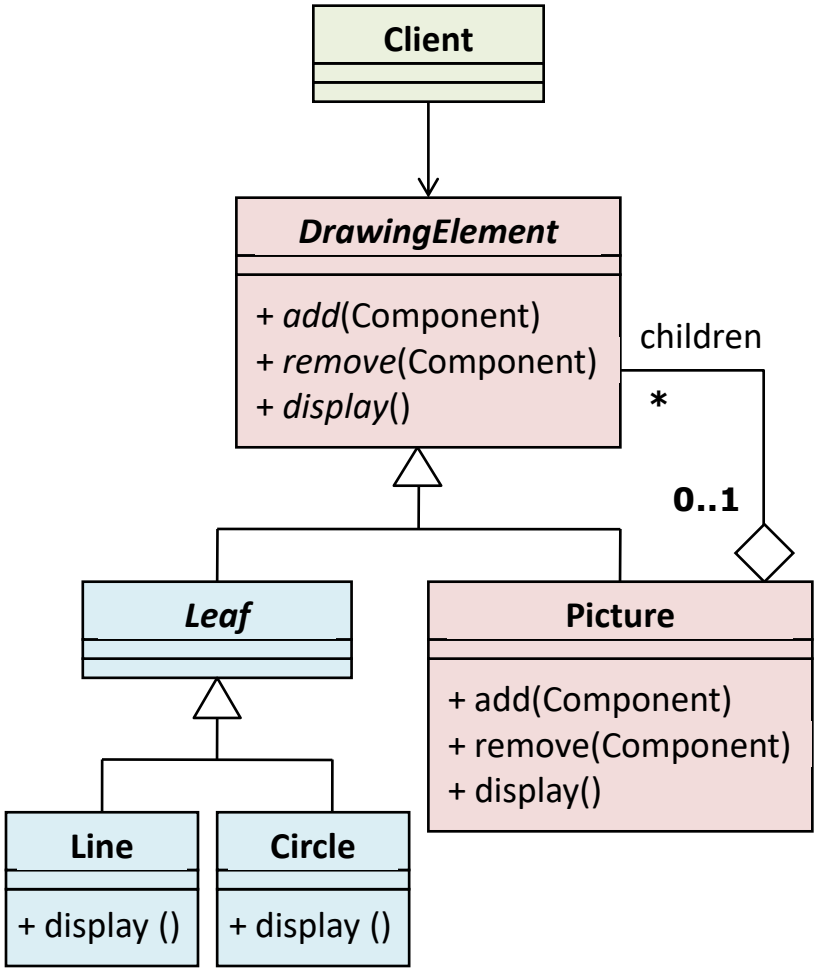
“kompleksna struktura”, “proizvoljne širine i dubine ”, ...

The Composite design pattern

Example:



Alternativa za modelovanje hijerarhije prostih elemenata



The Composite design pattern

Examples: