



Introduction to Software Engineering

Architectural Styles

Prof. Walter F. Tichy



Def.: Architectural Styles or architectural patterns structure a software system as a whole.

We will cover the following styles or patterns:

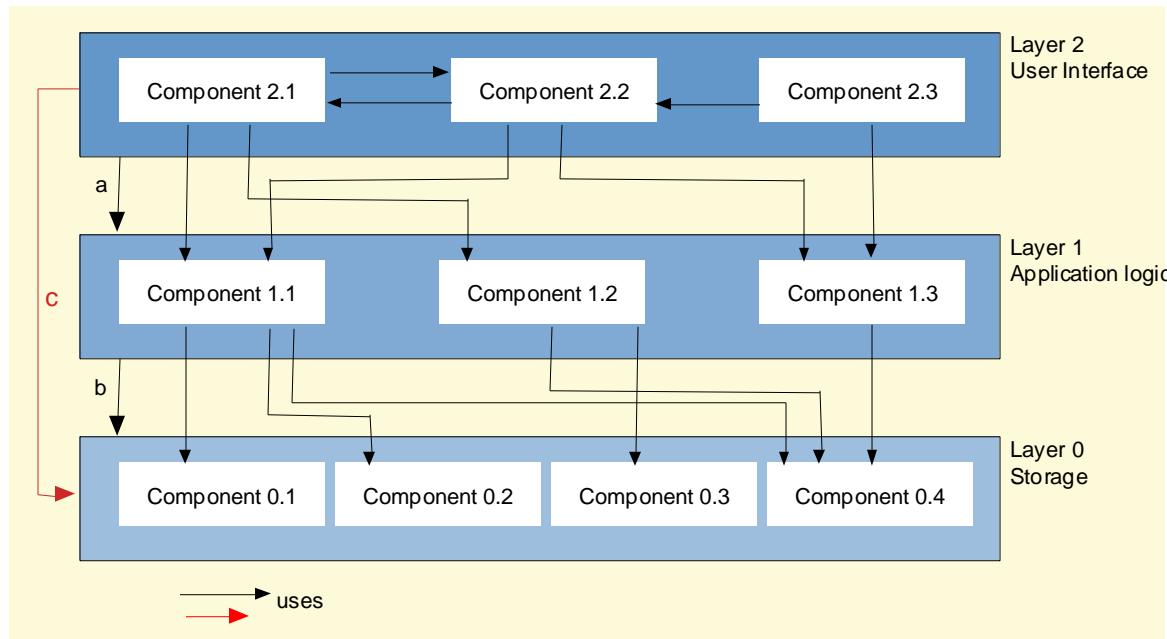
1. Layered software architecture (with optional Façade)
2. Virtual or abstract machine
3. Client/server architecture
4. Peer-to-peer network
5. Repository
6. Model-view-controller
7. Pipeline
8. Framework
9. Service-oriented architecture

The Classic: Layered Architecture

Def.: A layered architecture consists of a hierarchically ordered set of layers. Each layer consists of a set of software components for use by higher layers and may use lower layers to implement these components.

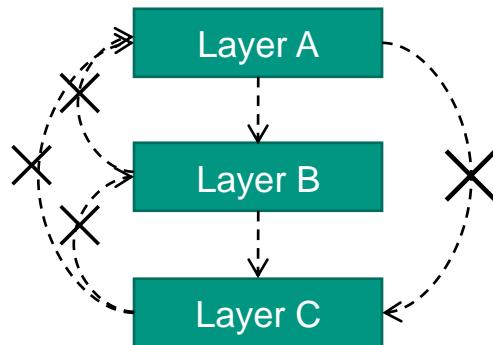
Layers

Example of a three-layer architecture



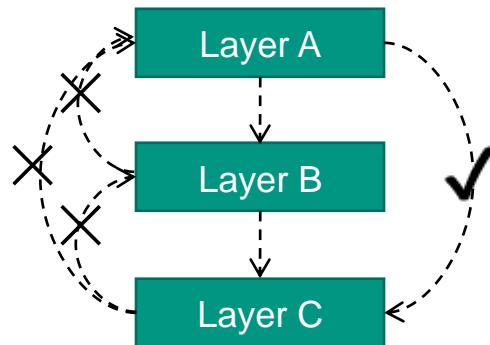
Opaque Layers

- In a layered architecture with **opaque layers**, each layer can only use the layer directly below it.



Transparent Layers

- In a layered architecture with **transparent layers**, each layer can use **all the layers below it**.



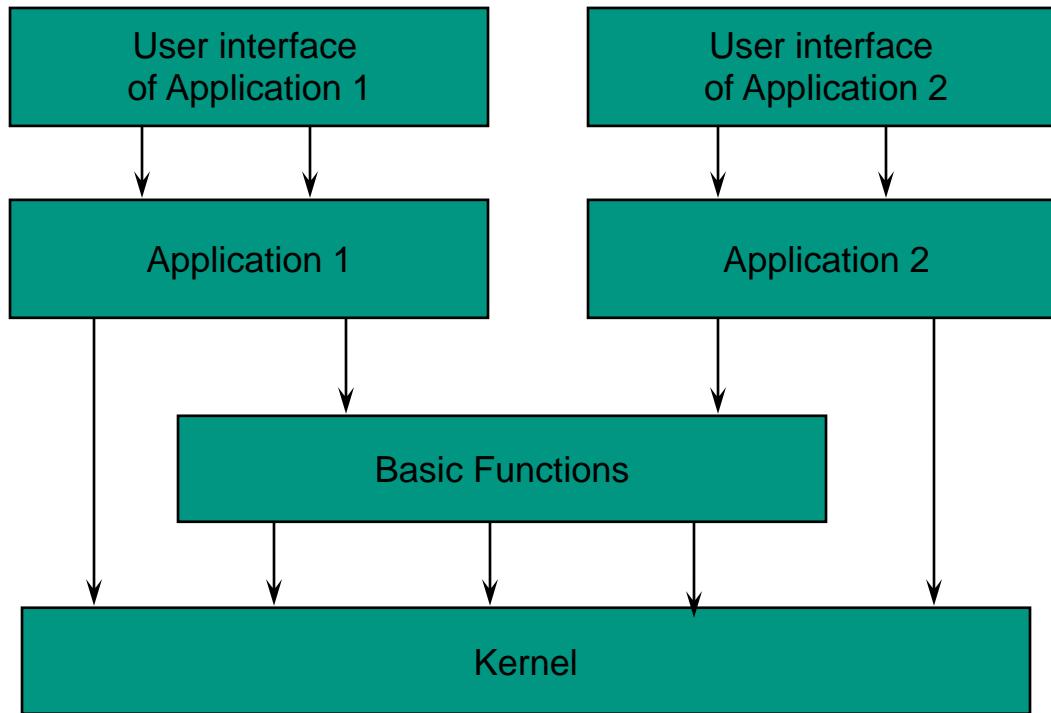
3-Tier Architecture

- A 3-tier architecture is a 3-layer architecture with some of the layers being on different machines.
 - For example, the user interface layer runs on the client machine, the application and database layers on the server.

Layers

- A 4-layer architecture is often used for web services and electronic commerce:
 - A web browser provides the user interface
 - a web server delivers static HTML pages
 - an applications server manages sessions, implements application logic, and produces dynamic HTML pages
 - a database holds the data

Sharing of Layers among applications



Layers

Examples

- operating systems, main layers (bottom to top):
 - process management
 - virtual memory
 - file system
 - communication (networking)
 - command interface (shell)
 - graphical user interface
 - applications
- protocol stacks (networking)
- business applications (3 layers: data base, application kernel, user interface)

Layers

Advantages

- Clear structuring of a system in levels of abstraction or virtual machines,
- No restrictions on the structure within layers,
- Independent development, testing, and replacement of layers,
- Bottom-up testing testing,
- Reuse of lower layers in other configurations,
- To provide a simplified interface for a complex layer (or several), one can use a Façade.

Layers

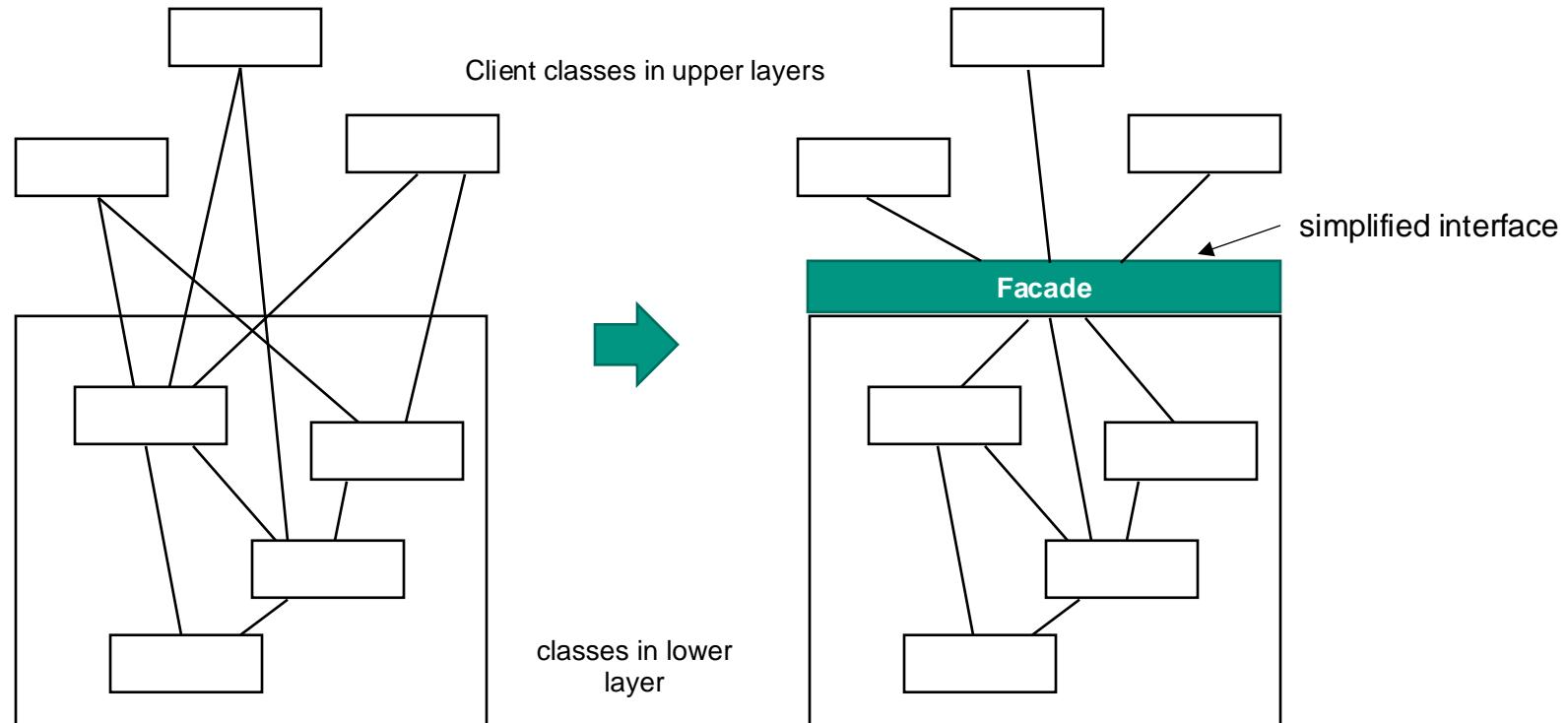
Potential problems with layers:

- opaque layers may lead to performance issues when calls need to be passed down through intervening layers to the target layer. This causes extra calls, parameter and return value passing.
- the same is true for passing error codes back, but not a problem when using exceptions with catch/throw.
- Avoid chaos within a layer!

Layers and Facade

- A layer often consists of a large number of software elements (packages, modules, classes, objects, methods, etc), only a subset of which should be used by upper layers. Using all the elements might lead to inconsistencies or failures or may be too complicated.
- A facade is an intermediate layer that provides a simplified and clean interface for a given layer, to be used by higher layers.
- The facade may provide new methods that combine calls that are often used together, plus convenience methods (with default parameters)
- The facade delegates calls to the original layer.

Layers and Facade



Abstract or Virtual Machine)

Def.: An abstract machine or virtual machine is a set of software commands and objects that are implemented on top of a real (or other abstract) machine and hides the underlying machine partially or completely.

- Examples for abstract machines:
 - Programming language, operating system, application logic, GUI-library, Java VM.
- The usage relation among abstract machines is hierarchical and thus free of cycles.

Abstract Machine

- An abstract machine is normally implemented by several modules or packages. The software commands and objects are provided by the interfaces of these modules.
- The underlying machine must be entirely or partially hidden to prevent inconsistencies between the two machines.
- The commands of the abstract machine should be chosen so they can be used in many different programs.
- An abstract machine is often implemented as library or an application programming interface (API).
- Obviously, virtual machines are special kinds of layers.
- Virtual machines can also provide the complete instructions sets of computers. This way, several (potentially different) operating systems with their applications can be run on the same computer, or shifted from one computer to another for load sharing. This capability is important for the servers in cloud computing.

Examples for abstract machines

- An **operating system** provides a powerful virtual machine with
 - process management,
 - virtual memory,
 - file system,
 - communication networks
 - command language
 - GUI
- The operating system hides certain privileged instructions which are needed for implementing its services, but would disrupt the operating system in case any application would use them. These include, for instance, instructions for changing tables that are used for mapping virtual memory addresses, or for reading and writing from disks or network connections.
- The (non-)use of these privileged instructions is checked at runtime and leads to an interruption or they are simply ignored.

Examples of abstract machines

- Java Virtual Machine: interprets **Bytecode**. Its commands are the JVM instructions
- Java compiler
 - offers an abstract machine programmed in Java
 - Is built upon another abstract machine, the Java VM, using compilation
 - Java compiler hides Java VM and the instructions of the real computer underneath
 - Hiding is accomplished by compilation.
- A text formatter is also a virtual machine, providing commands to format texts.

More examples

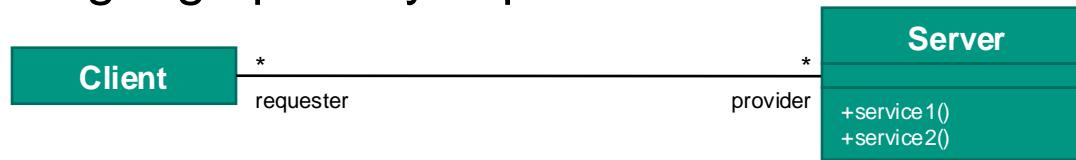
- E-mail client
 - offers commands for sending, receiving, and filing messages, and hides the underlying protocols (buffering, breaking up of long messages in packets and recombining them, monitoring failures and resending messages if corrupted or lost). The e-mail client insures reliable delivery of messages.
- Macros
 - Macros work by parameterized text replacement. The call of a macro causes the replacement of the call with the body of the macro („inlining“)
 - A macro in C: `#define Swap(X, Y) int temp=X; X=Y; Y=temp;`
 - `Swap(a,b)` is replaced by `int temp=a; a=b; b=temp;`
 - Macros cannot hide or protect an underlying machine. They have the same privileges as the abstract machine in which they are called. For instance, macros can be used to code the call and return of methods. But other instructions could disrupt the call stack. Macros cannot protect the call stack.

More examples

- Virtualization is important for servers in the internet („Cloud Computing“)
 - A customer may want to rent an IBM computer. This computer is virtualized or simulated on a different computer with a different instruction set. The customer can then install the desired operating system and applications on the virtual machine.
 - It is even possible to simulate several (potentially different) computers on the same real machine.
 - The virtualization may be done by software, or by dynamic translation of instructions of the virtual machine into instructions of the target machine. A similar technique is used by just-in-time compilers for programming languages.
 - Virtualization can be used to save energy. If there is low demand, virtual machines can be concentrated on a few active servers, and the rest of the servers can be placed in standby mode. If demand rises again, standby servers are powered up and share the load of virtual machines.

Client/Server

- One or more servers offer services for other systems or applications, called clients. Client and server are usually located on different computers and communicate via a network.
- A client calls a function of the server (by sending a packet with the function's name and parameters), which executes the function and returns the result (as a packet)
 - The client must „know“ the interface of the server
 - but not vice versa.
- In principle, this is a special case of a two-layer architecture, where the two layers are geographically separated.



Client/Server

- Often used for database applications:
 - Front-End: User interface (client)
 - Back-End: database access and manipulation (server)
- Functions at the client side:
 - Requesting inputs, displaying outputs
 - Preprocessing of inputs (consistency/completeness checks, aggregation)
- Functions at the server side:
 - Data management
 - Data integrity and consistency
 - Parallel transactions
 - Security
- Though client and server normally run on different machines, installing them on the same machine is also possible and requires no software change. Client and server continue to use network protocols for communication. This is very handy for testing.

Client/Server Example

File transfer with FTP-Server:

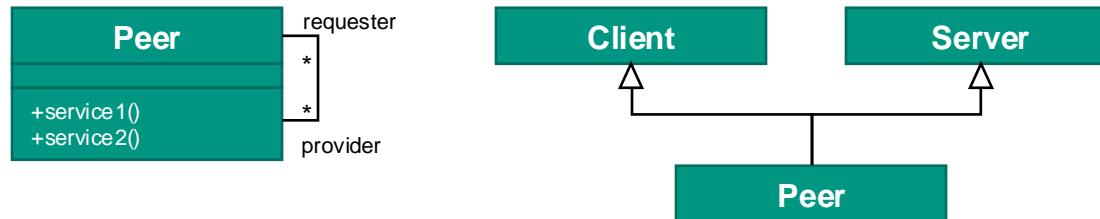
- The client (for example, a FTP client such as Filezilla) initiates a file transfer.
- The FTP server reacts to the client's request and receives or sends the file

Every Internet browser is a client that displays HTML-pages delivered by a server.

Servers can be replicated (internet servers are replicated in different countries or regions).

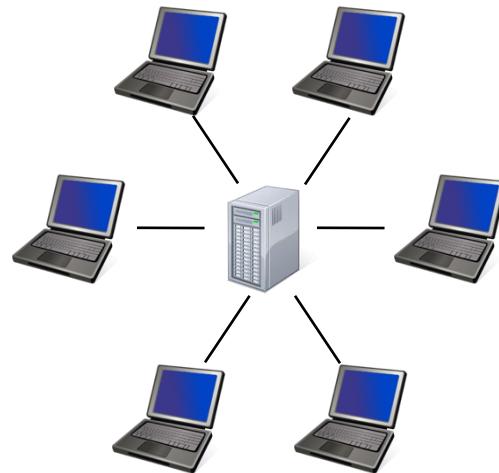
Peer-to-Peer Networks

- Generalization of the client/server architecture.
- The participating computers are called „peers“. They are co-equal. Every peer is both a client and a server.
- Simplified:

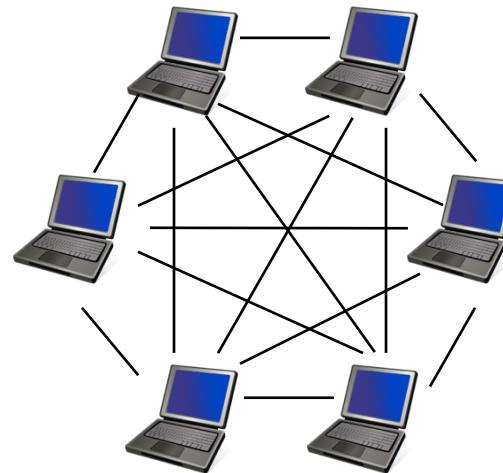


peer-to-peer vs. client/server

6 clients 1 server



peer-to-peer



Properties of peer-to-peer (1)

- Symmetry of roles:
 - every peer is both client and server
- Decentralization:
 - There is no central coordination and no central data base.
 - Every peer knows about only a subset of all peers („local neighborhood“)
- Self organization:
 - The behavior of the peer-to-peer net is the composition of the behavior of the individuals.

Properties of peer-to-peer (2)

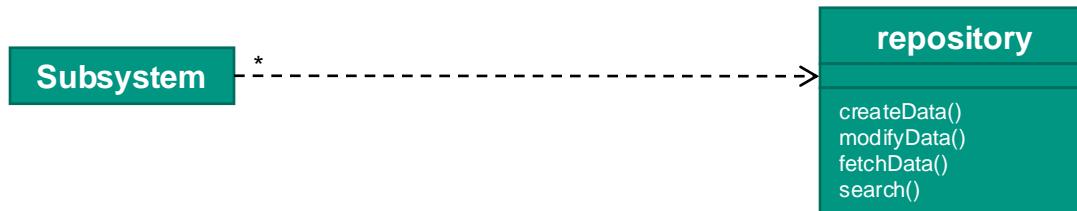
- Autonomy:
 - partners behave autonomously
- Reliability:
 - individual peers are not reliable (e.g., they may crash or be turned off). Mechanisms are needed to compensate for this unreliability.
- Availability:
 - All data must be stored redundantly because of the unreliability of (some) peers, and there must be ways to update a peer that was down.

Peer-to-peer examples

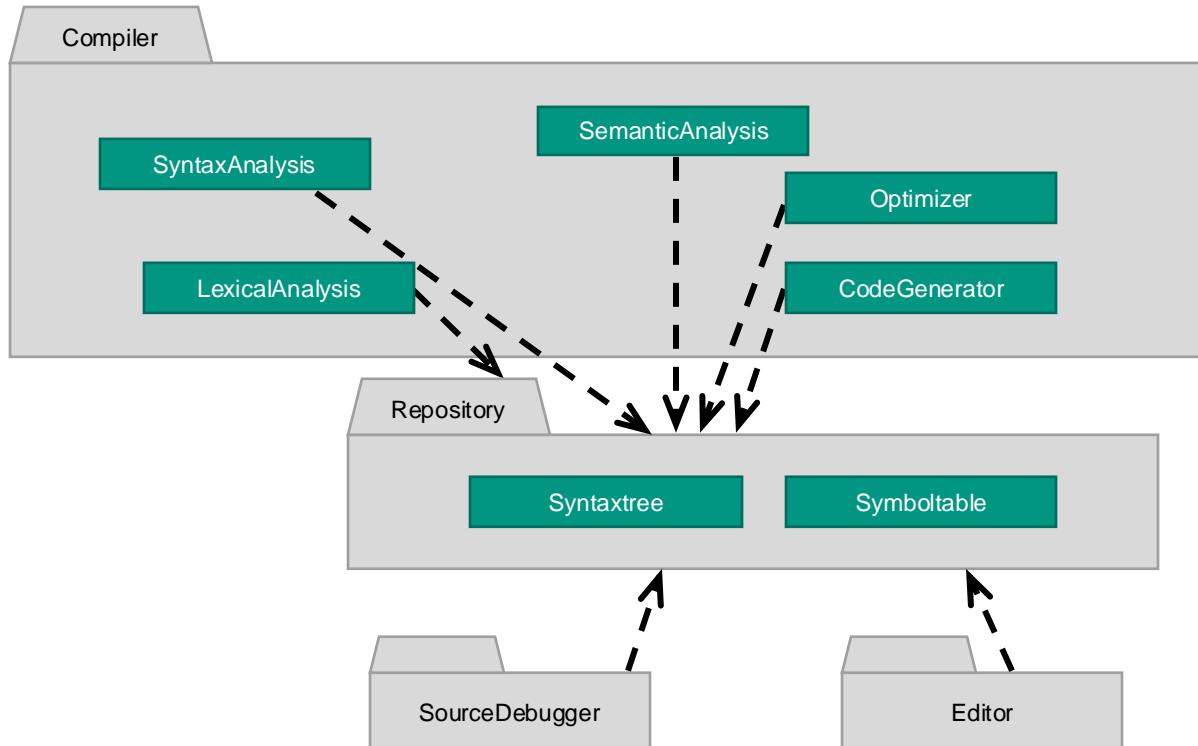
- Exchange of data in a peer-to-peer network, such as BitTorrent:
 - Each peer is server and client
 - It can request files from another peer (as client)
 - It can offer files (and other services) to other peers (as a server)
 - When a request cannot be served, it is forwarded to another peer (and to another...)
- At Internet level: TCP/IP, DNS:
 - Requests are distributed independent of application over the physical network (net neutrality).

Repository

- Subsystems read and modify data in a central repository.
- Subsystems are loosely coupled and interact only through the repository.
- Subsystems can access repository in some order or in parallel. In the latter case, we need synchronisation or a transaction mechanism
- Access can be local or remote



Repository example

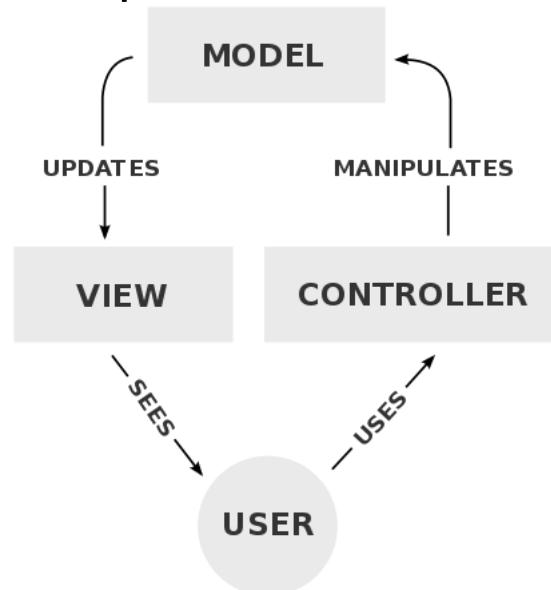


Repository example

- Explanation:
 - Compiler, editor and debugger are executed in arbitrary order, including in parallel.
 - The repository makes sure that write accesses do not lead to inconsistencies. A transaction mechanism ensures that in case of overlapping accesses it always appears as if the overlapping accesses were executed and completed in some sequential order.
 - It is also possible to lock the entire repository during updates, allowing only one process to operate on it.
- Further examples: Subversion, Git (systems for version control)

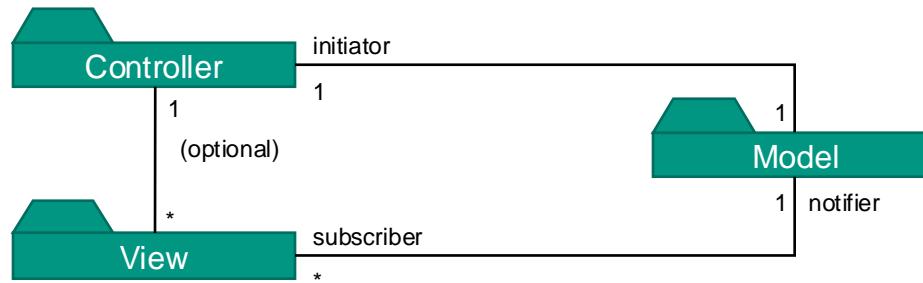
Model-View-Controller (MVC)

Model–view–controller (MVC) is commonly used for developing user interfaces. It divides the program logic into three interconnected elements. This is done to separate internal representations of data from the ways information is presented to the user and input is processed.



MVC

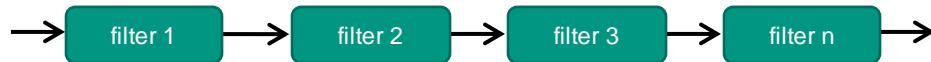
- **Model:** The central component of the pattern. It contains the application's dynamic data structure, independent of the user interface. It directly manages the data, logic and rules of the application.
- **View:** Displays the data structure to the user. It is updated when the model changes. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants. There may be several views open simultaneously.
- **Controller:** Responsible for user interaction; accepts input and converts it to commands for the model or view.



MVC

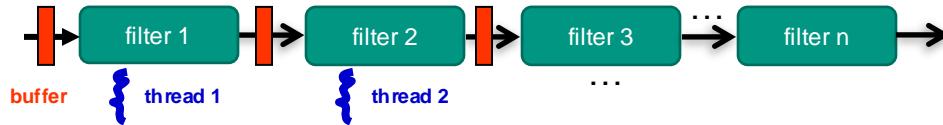
- MVC contains a sub-pattern called „observer“, consisting of model and view. The observer pattern lets views register at the model. When the model changes, it sends a notification to the registered view(s). The views then fetch the relevant data and display it (see chapter on design patterns). In MVC, this update process is initiated by the controller.
- Usually, the view and the control surfaces (buttons, menus, etc) of the controller are combined in the graphical user interface.
- There can be an (optional) connection between view and controller.
- View and model can be independently re-used in other applications.

Pipeline or Pipe and Filter



- Each filter is an independently running process or thread with its own program, instruction pointer, and memory.
- Data flows through the pipeline. Each filter receives data from the previous filter, processes the data, and passes them own to the next filter in the pipeline.
- A buffer of limited size between the filters smooths out fluctuating speeds of the filters.

Pipeline

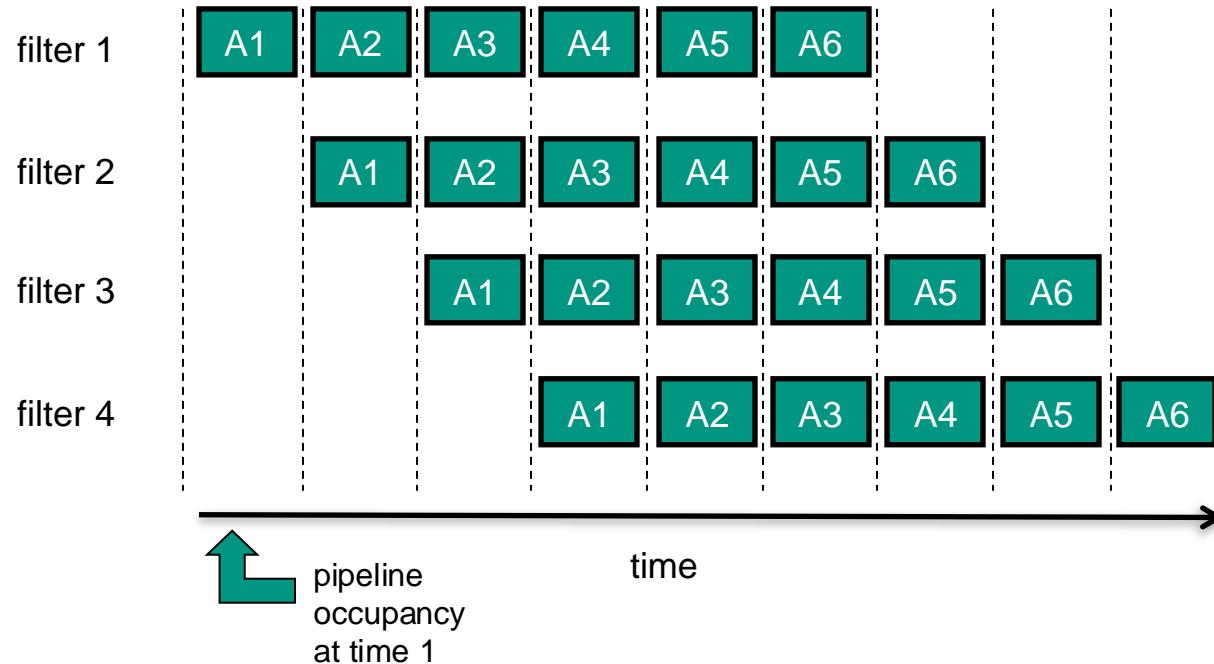


On parallel computers, the filters can be run in parallel, which improves throughput compared to a sequential pipeline (sequential means running each filter to completion before the next filter)

On a sequential computer, the threads are executed alternately, in an interleaved fashion determined by the process scheduler of the operating system.

Pipeline

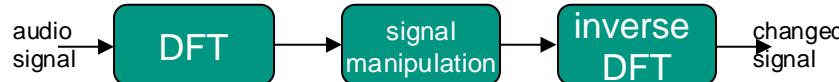
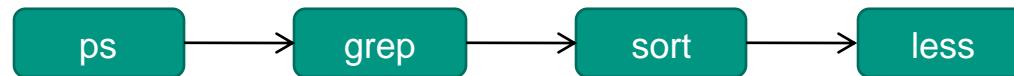
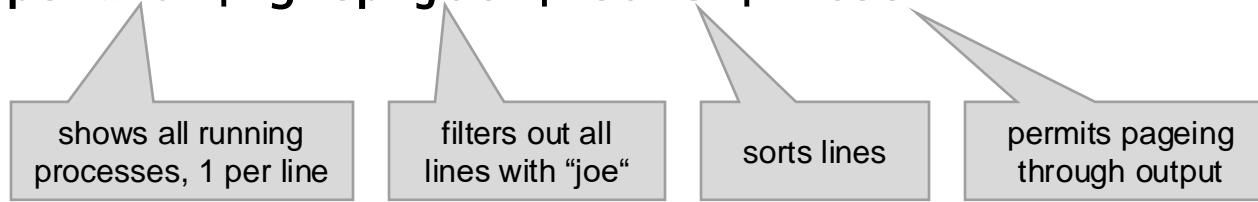
- Principle of pipeline operation on a parallel computer



Pipeline examples

The Unix Shell provides pipelines. “|“ is used for connecting filters

`ps axu | grep joe | sort | less`



Applicability of pipeline

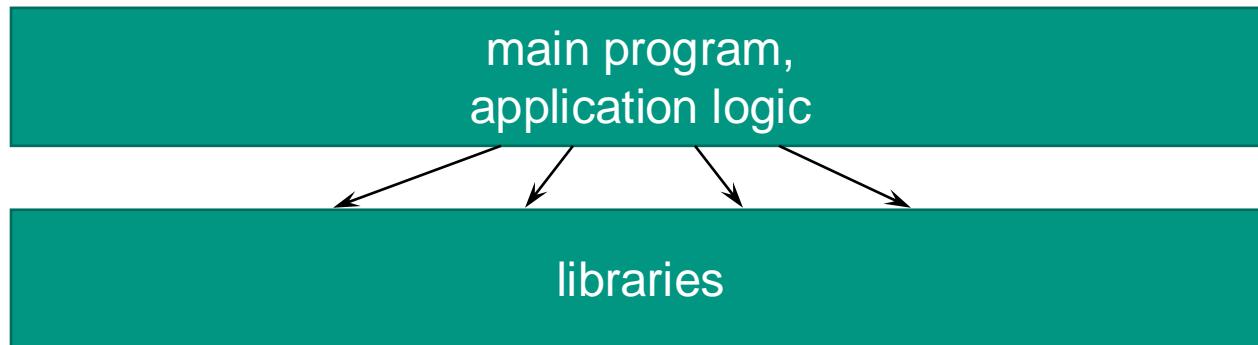
- Well suited for data streams, for example for video processing, compilers, batch processing.
- The performance of the pipeline is limited by the **slowest** filter. For good performance on parallel machines, the filters should be about the same speed. If a certain filter is much slower than others, it can be replicated to speed up throughput. (this consideration is irrelevant if there is only one processor)
- Pipelines do not speed up the response time for individual data packets or tasks, but they can increase throughput (if filters run in parallel).
Why?

Framework, Framework Architecture, Plugin Architecture

- A framework architecture is a (nearly) **complete program** that can be **extended** by the user with “**plugins**”. These plugins are added at pre-planned extension points called “hooks”.
- The framework contains the **complete application logic** and the main program. It can actually run.
- For some of the classes, users can **supply subclasses** that overwrite existing methods or implement abstract methods.
 - The framework calls these subclasses (the plugins) at the right time.

Structure of Framework (1)

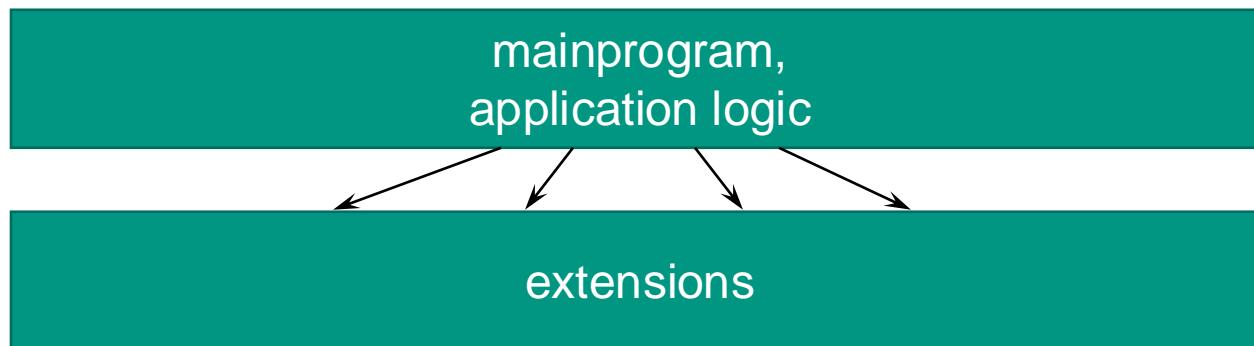
- Usual (non-framework) architecture:
 - Software producer delivers a library
 - User writes application code calling the library, including a main program



Structure of Framework (2)

■ Framework architecture

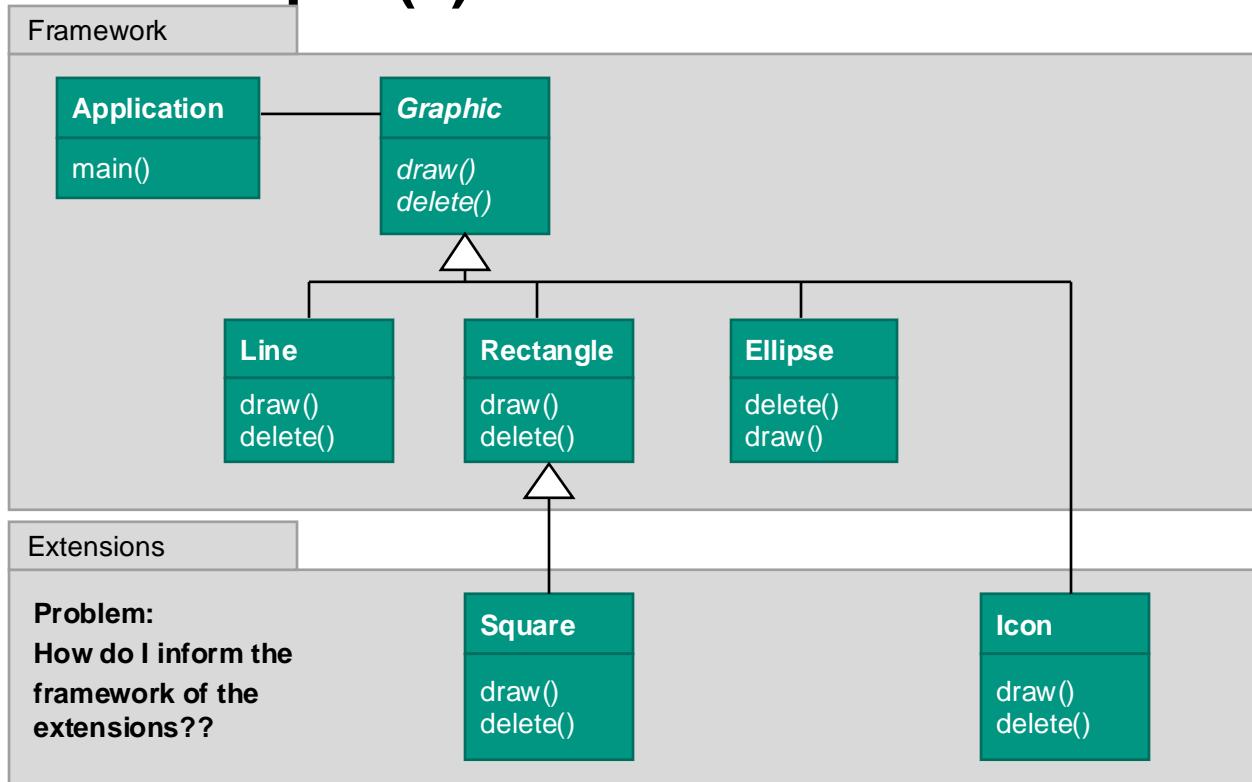
- follows the „Hollywood-Principle“:
„Don't call us – we call you“.
- most of the application logic and the main program exist already, call extensions supplied by user.



Framework example (1)

- A drawing framework supplies a class `Graphic` with several subclasses (line, rectangle, ellipse, etc.)
- The user is allowed to supply additional subclasses of `Graphic`, for example a square or icon, and must implement the method `draw()` for each such subclass.
- The framework can then create instances of the new classes, position, draw, move, and save them, etc.

Framework example (2)

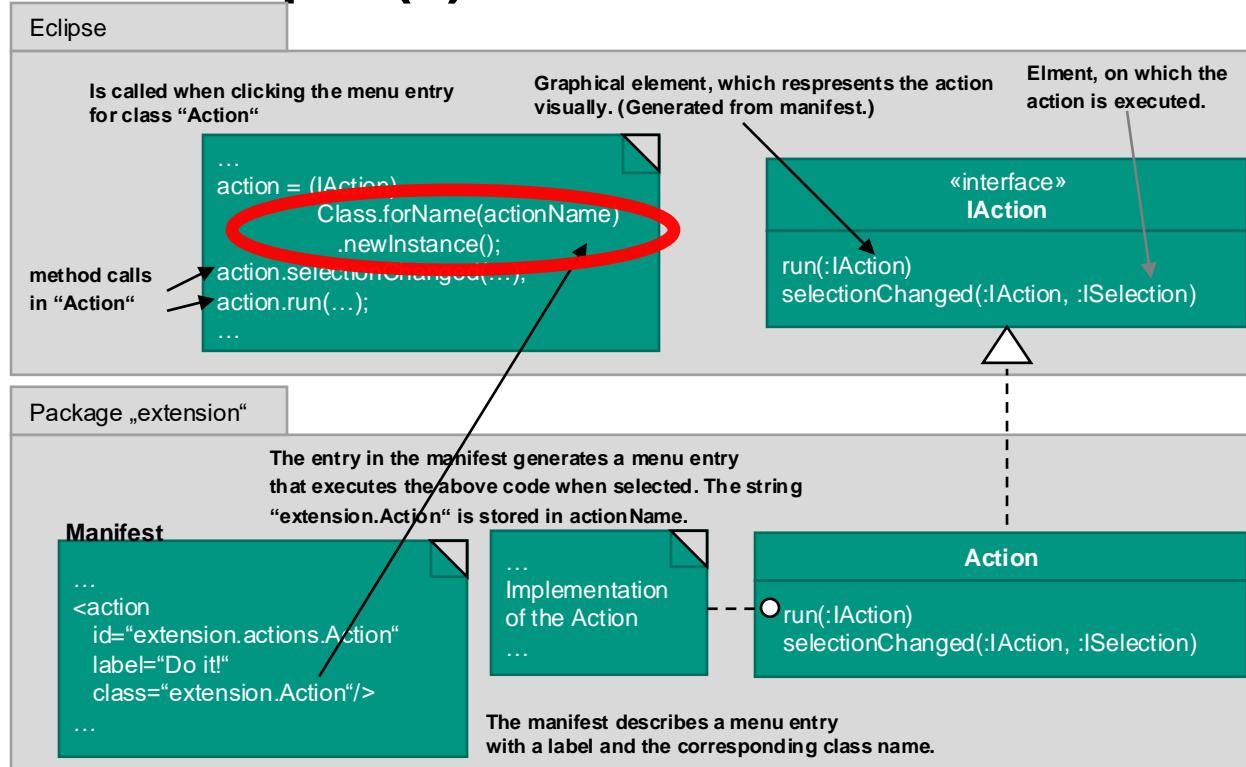


Framework example (3)

Eclipse: open-source development environment

- Consists of a relatively small kernel with a great number of extension points for plugins.
- Many parts of the standard distribution of Eclipse are plugins.
- The plugins are stored in jar-files, with a manifest, which is read by the Eclipse kernel to get the names of the plugin classes.
- If given an identifier „`Id`“, the call to
`Class.forName("Id").newInstance()`
starts a search for the class with name „`Id`“ and loads it. Then
`.newInstance()`
- creates a new instance from the new class. The framework can now call the methods on it that were advertised in the superclass.

Framework example (4)



Framework example (5)

- The manifest contains the name of the extension classes, the label for the menu entry, and further details.
- Menu entries and buttons in Eclipse are generated from this information.
 - The display of these elements is possible, without having to load the class immediately (→ delayed load prevents waits when starting the application)
- If a new method is called via the menu entry, then Eclipse loads the class with `Class.forName(...)`, instantiates it and calls the methods defined in the interface
- When initializing (`selectionChanged`) we pass the context. (editor, selection, project, ...)

Framework: Applicability

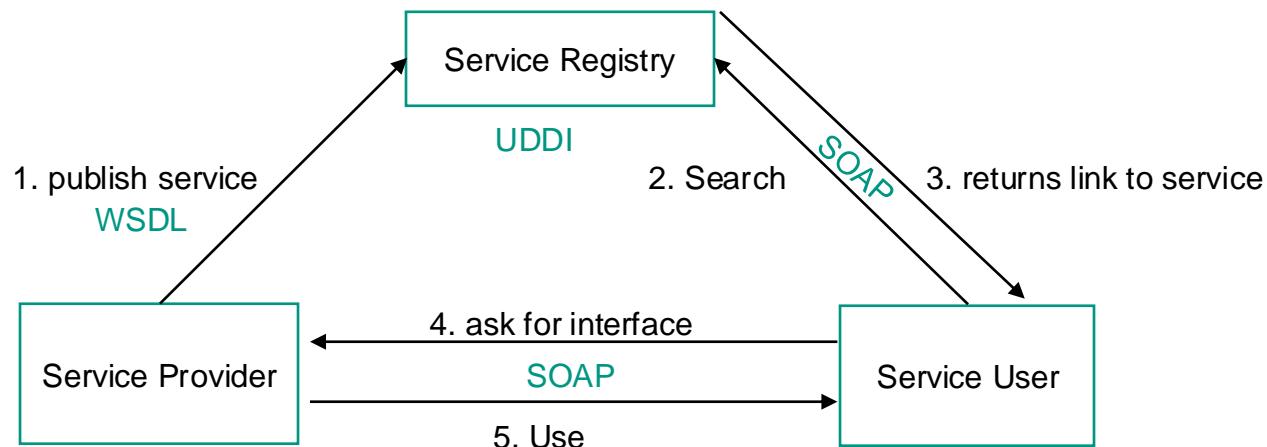
- When a **basic version** of the application needs to be working and
- when extension should be possible, which work consistently with the rest
 - place application logic into framework
- When **complex application logic** should not be re-programmed, but customizing still possible
- The design patterns **strategy**, **factory method**, **abstract factory**, and **template method** are often employed in Frameworks
 - (see next section)

Service Oriented Architecture, SOA

- SOA is an architectural style, in which applications are composed of independent services.
- SOA is an abstract concept of a software architecture
- Services are seen as the central offerings of an enterprise
 - providing encapsulated functionality for other services and applications
 - interfaces are described in standardized form
 - services are programming language independent (services can be written in any language)
 - older systems can be encapsulated (and later replaced)
- Example services: weather service, credit rating service, credit card processing service

Service model is core of a SOA

- Required functionality can be added as a service at runtime
- A service registry is needed to find services



Acronyms

WSDL: The **Web Services Description Language (WSDL)** (/'wɪz dəl/) is an XML-based interface description language that is used for describing the functionality offered by a web service. The acronym is also used for any specific WSDL description of a web service (also referred to as a *WSDL file*), which provides a machine-readable description of how the service can be called, what parameters it expects, and what data structures it returns. Its purpose is roughly similar to that of a method signature in a programming language.

UDDI: **Universal Description, Discovery and Integration (UDDI,** pronounced /'jʊdi:/) is a platform-independent markup language protocol that includes a (XML-based) registry by which businesses worldwide can list themselves on the Internet, and a mechanism to register and locate web service applications.

Acronyms

SOAP (Simple Object Access Protocol) is a messaging protocol specification for exchanging structured information in the implementation of web services in computer networks. It uses XML for its message format

SOAP allows developers to invoke processes running on different operating systems (such as Windows, macOS and Linux). SOAP allows clients to invoke web services and receive responses independent of language and platforms.

Characteristics of SOA

- Loose coupling
 - Services can be added and replaced at runtime
 - Dynamic binding of the services on any platform made possible by UDDI and the service registry.
- Support of business processes
 - Services encapsulate business-relevant functionality
 - Complex applications are the composition of services
- Using open standards
- Services can be written in any programming language on any platform.