Software Construction and User Interfaces (SE/ComS 319)

Ali Jannesari

Department of Computer Science

Iowa State University, Spring 2019

ARCHITECTURAL STYLES

Administrative

- Final exam:
- https://www.registrar.iastate.edu/students/exams/springexams
- **LAGOMAR W0142 (lecture room)**

Thursday	**May 9th**	**9:45-11:45 a.m.**

Outline

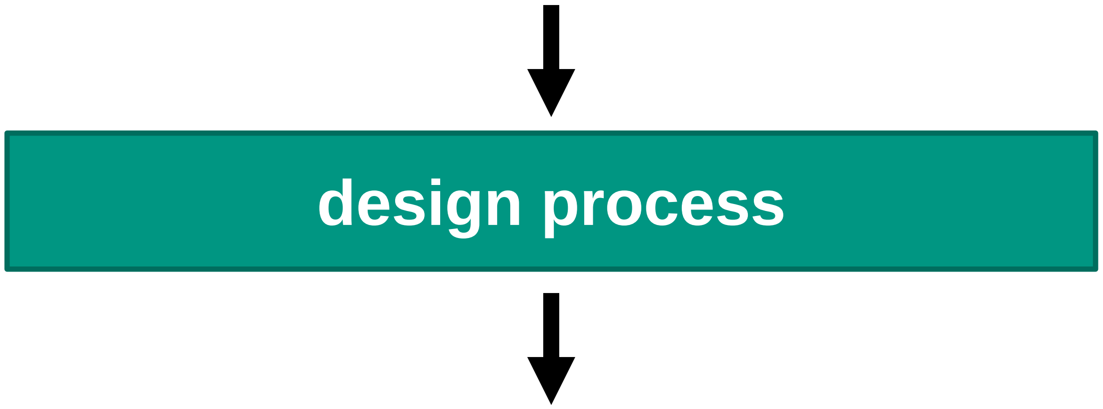
- Design (review)
    - Modular design
    - OO design
- Architectural style
    - Layered architecture
    - Client/server
    - Peer-to-peer
    - Model View Controller
    - Pipeline (pipe and filter)
    - …

Software architecture

- **Software architecture**: Structure of a software system into components (modules or classes) and subsystems (packages, libraries).This creates a hierarchy.
- **Specification** of components and subsystems
- Set up the **"uses”-relation** between components and subsystems
- Optional: fine draft
    - Specification of data structures and algorithms, pseudocode
- Optional: Assignment of SW components and subsystems to HW units
    - In distributed systems, this means systems that are distributed across multiple machines

Design – Software architecture

Specification (including models)
User interfaces
User manual + help concept



Architectural design can be conducted right **after use-case study!**

Software architecture
- Organization, structure
- Output:
  - Components
  - The interaction among components

Questions that need clarification during design

- Which components already exist and can be **reused** / **purchased**, or are freely available?
    - In this case, the architecture may need to be adapted!
- Are there **real-time conditions**, and how are they kept?
- How are **persistent** data stored permanently?
    - In files or in databases?
- How is the flow control organized?
    - Monolithic (via calls), event-driven, parallel?
- How is the system installed, started and stopped?
- How are exceptions / errors handled?
- How are access rights granted and monitored?

Why should we do architectural design?

- We need it even in agile process (XP process)
  - Refactoring does not help (it changes architectural design; simple design!)
- **Non-functional requirements** can be affected by architectural design
  - Maintainability: the less interaction among components, the better
  - Availability: redundant component can be used to improve availability
- Only a subset of non-functional requirements can be met simultaneously, usually, we need to make a trade-off:
  - Runtime vs. memory footprint, runtime vs. portability (platform independence), development time vs. robustness, reliability, …
- As soon as the non-functional requirements are clear, a first architecture can be taken.

Methodology of the design

- There are two different design methods:
  1. **Modular design**
  2. **Object-oriented design**
- OO design extends modular design with inheritance, polymorphism and data modeling

MODULAR DESIGN

Modular design (1)

System architecture in modular design:

- **Module Guide** (rough design, software architecture):
  - Structure into components (modules) and subsystems
  - Description of the function of each module
  - Uses design patterns, e.g. layered or pipeline architecture.
- **Module interfaces**:
  - Exact description of the elements provided by each module (types, variables, subroutines, etc.), informal or formal.
  - For modules with input/output also exact description of the respective formats (e.g. with XML or grammars)

Modular design (2)

System architecture in modular design:

- **Uses-relation:** Structure in components (modules) and subsystems
  - Describes how modules and subsystems use each other. The uses-relation should be an acyclic, directed graph for incremental construction and incremental testing.
- **Detailed design** (optional)**:** Description of the module-internal data structures and algorithms;
  - Complete programming of modules in pseudocode (e.g. when implementing in assembler or low-level languages).
  - The pseudocode is written in a higher, mostly hypothetical programming language (e.g. with control structures borrowed from C and instructions in natural language) and it will be implemented in the implementation phase to another language (e.g. assembler).

Modular design

- **External Design:** high-level rough design and module interfaces
- **Internal design:** uses-relation and detailed design
- The concepts of modular design retain their validity for object-oriented design.

Modular concept (1)

- Modules should be able to be modified and used independently of each other.
  - A module should be designed, implemented, tested and revised **without knowledge** of its **later use**;
  - The implementation of a module should be possible **without** having to know anything about the **implementation details** of other modules and without influencing the behavior of other modules;
  - A module should be able to be used without knowledge of its internal structure (**encapsulation** of internal structures).

Modular concept (2)

- Modules should be able to be modified and used independently of each other.
  - Normal case: A module contains several subroutines that manipulate a data structure
    - Direct access of other modules to this data structure is not possible **– data encapsulation** (similar to an object).
  - **Strong cohesion** within a module
    - Cohesion between modules less.
  - A module should be simple enough to be fully understood

Module definition

- **Module**: A module is a lot of program elements that follow the **information hiding principle** to be designed and changed together.
  - Program elements are types, classes, constants, variables, data structures, procedures, functions, processes (threads), process inputs, macros, etc.

- ***Information hiding principle***: Each module hides an important design decision behind a well-defined interface, which does not change when the decision is changed (David Parnas).
  - Reason: Only what is hidden and unused can be changed without risk.
- **Example**: `public class RouteWithEncapsulationOne{`
  `private int startpoint_x;... } //encapsulation on attributes`

Candidates for hiding – Design decisions for modules

- The classic: **data structures** (choice of data structure, size) and implementation of operations on these data structures
- **Machine-related details** (e.g. device driver, I/O control, character codes and their order, etc.)
- **Operating system-related details** (input/output interfaces, file formats, network protocols, command language, etc.)
- **Basic software** such as databases, GUI-libraries, or similar.
- **Input/output formats** (only one module knows a given format)
- **User interfaces** (command interface, graphical interface, gesture-controlled interfaces, web, voice control, combinations of them),…

Module interfaces

- Design and documentation of the module interfaces:
  - **Result**: "**Black Box**" description of each module
  - Exactly the information required for both the use and implementation of the module, not more.
  - The interface is invariant with regard to foreseeable changes.

Module interfaces

- Description of the module interfaces consists of:
    - List of **public** program elements
    - Input/output formats (if I/O module)
    - **Parameters** and **return values** of the subprograms / operations
    - Description of the effect of the subroutines
    - Timing, accuracy, storage space requirements and other quality features where required
    - Error description and error handling
    - **Exceptions** that are raised and not handled.

OO DESIGN

Object-oriented design

- In object-oriented design, the principles of modular design retain their validity
  - Flexible software through the **information hiding principle**
- Interfaces **hide** design decisions that should remain changeable

Object-oriented design

- **External** design
  - The analogues to the module are the class and the package
  - The package combines several classes that encapsulate common design decisions.
  - A single class can also realize a whole module; however, you need several classes per module.

Object-oriented design

- **External** design
  - Instead of the module guide, the package and class guide is used, usually as a UML class and UML package diagram with explanatory text
    - Documents the design decisions
  - The analogue to the module interfaces are the interfaces of the classes, abstract classes and interfaces
    - Interfaces

Object-oriented design

- **Internal** design
  - The uses-relation is documented at the level of packages and stand-alone classes (which are not themselves in packages).
  - The detailed design provides the description of the module-internal data structures and algorithms, as well as pseudocode where necessary, as in the modular design.

Object-oriented design

- However, there are additional possibilities in the OO design that are more difficult to implement in the modular design.
  - Multiple instancing of classes,
  - Inheritance and polymorphism,
  - Variant formation in a program by multiple implementation of an interface
- For this purpose, new structures and architectures have emerged, which are called **design patterns**.
- Architectural styles come next!
  - And we have learned so far several design patterns!

ARCHITECTURAL STYLES

Abstract / Virtual machine

- Abstract / virtual Machine is a set of software commands and objects that can build on (abstract or real) machine and completely or partially hide the underlying machine.
- Examples of abstract machines:
  - Programming language, operating system, application core, GUI library, Java VM.
- The uses-relationship between several abstract machines is hierarchical, i.e. cycle-free.

Abstract machine

- An abstract machine is typically implemented by **one or more modules** or packages. The software commands and objects are provided by the interfaces of these modules.
- The underlying machine must be completely or partially hidden to avoid inconsistencies between the two machines.
- The commands of the abstract machine should be chosen so that they can be used in a variety of programs.
- An abstract machine is often implemented as a program library or **Application Programming Interface (API)**.

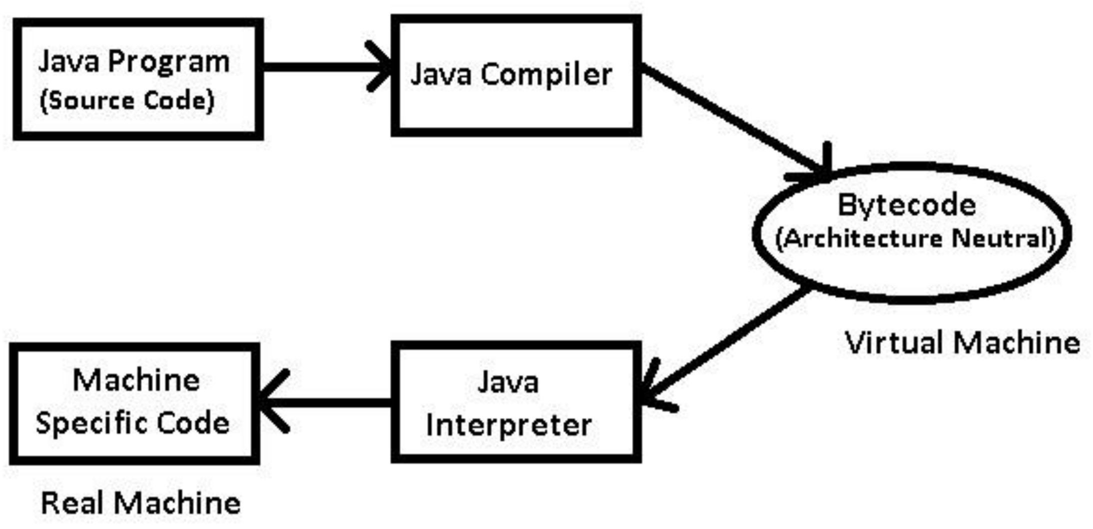
Examples of abstract machines

- An **operating system** provides a powerful abstract machine for
  - Process management,
  - Virtual memory,
  - Data storage on background memory,
  - Communication
  - Command languages, graphical user interface
- The operating system hides certain privileged commands needed to implement the operating system services.
- The non-use of the privileged commands is checked at runtime and results in an interrupt or ignored.

Examples of abstract machines – JVM

- Java Virtual Machine: interprets **byte code**
- Java compiler
  - Provides abstract machine that is programmed in Java
  - Is based on another abstract machine, i.e. the Java VM
  - Machine commands, bytecode are hidden
  - The masking is checked

    by the compiler



- **Interpreter vs. compiler?**

Examples of abstract machines – JVM

- **Interpreter:** (example: Python, Ruby).
  - Translates program one statement at a time ➔ less amount of time to analyze the source code but the overall execution time is slower.
  - No intermediate object code is generated ➔ memory efficient.
  - Continues translating the program until the first error is met, in which case it stops ➔ debugging is easy.
- **Compiler:** (example: C/C++)
  - Scans the entire program and translates it into machine code ➔ large amount of time to analyze the source code but the overall execution time is comparatively faster.
  - Generates intermediate object code and requires linking ➔ more memory
  - Error message after scanning the whole program ➔ Debugging hard

Examples of abstract machines

- **Virtualization** is very important to Internet service providers ("**cloud computing**").
  - For example, a customer can rent an IBM computer. This can be "virtualized" on a computer of another type, i.e. the instruction set of the IBM computer is completely simulated as a virtual machine on another computer. The customer can install his desired operating system on this virtual machine.
  - For a real computer, several (even different) virtual machines can be virtualized.
  - The virtualization is done either by software, but also by dynamic translation of command sequences of the virtual machine in commands of the simulating machine (see Just-in-time compiler for Java VM)
  - Virtualization ensures a good utilization of computer resources. It is also possible to "move" applications from one computer to another for load balancing purposes.

Architectural styles

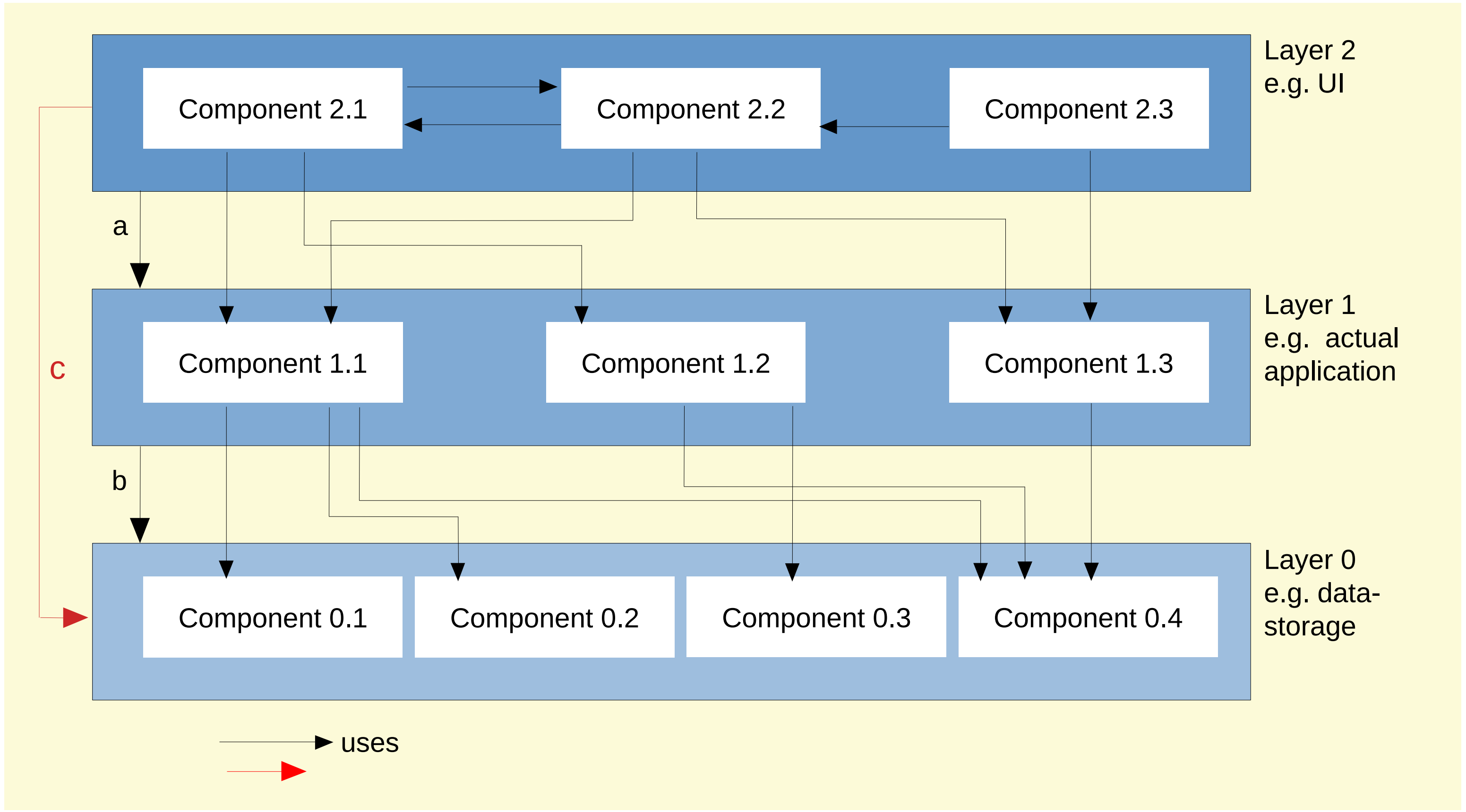
- Architectural styles determine the rough structure of a software system:
  - Layered architecture
  - Client/server
  - Peer-to-peer
  - Repository (Data-centered architecture)
  - Model View Controller
  - Pipeline (pipe and filter)
  - Framework
  - Service oriented architecture

The classic: Layered architecture

- A **layered architecture** is the structure of a software architecture in hierarchically ordered layers.
- A layer (tier) consists of a set of software components (modules, classes, objects, packages) with a well-defined interface, uses the underlying layers, and provides its services to layers above it.
- Between the individual layers, the uses relation is linear, tree-like, or an acyclic graph. Within a layer, the uses-relationship is arbitrary.

Layered architecture

- Example of a three-tier architecture



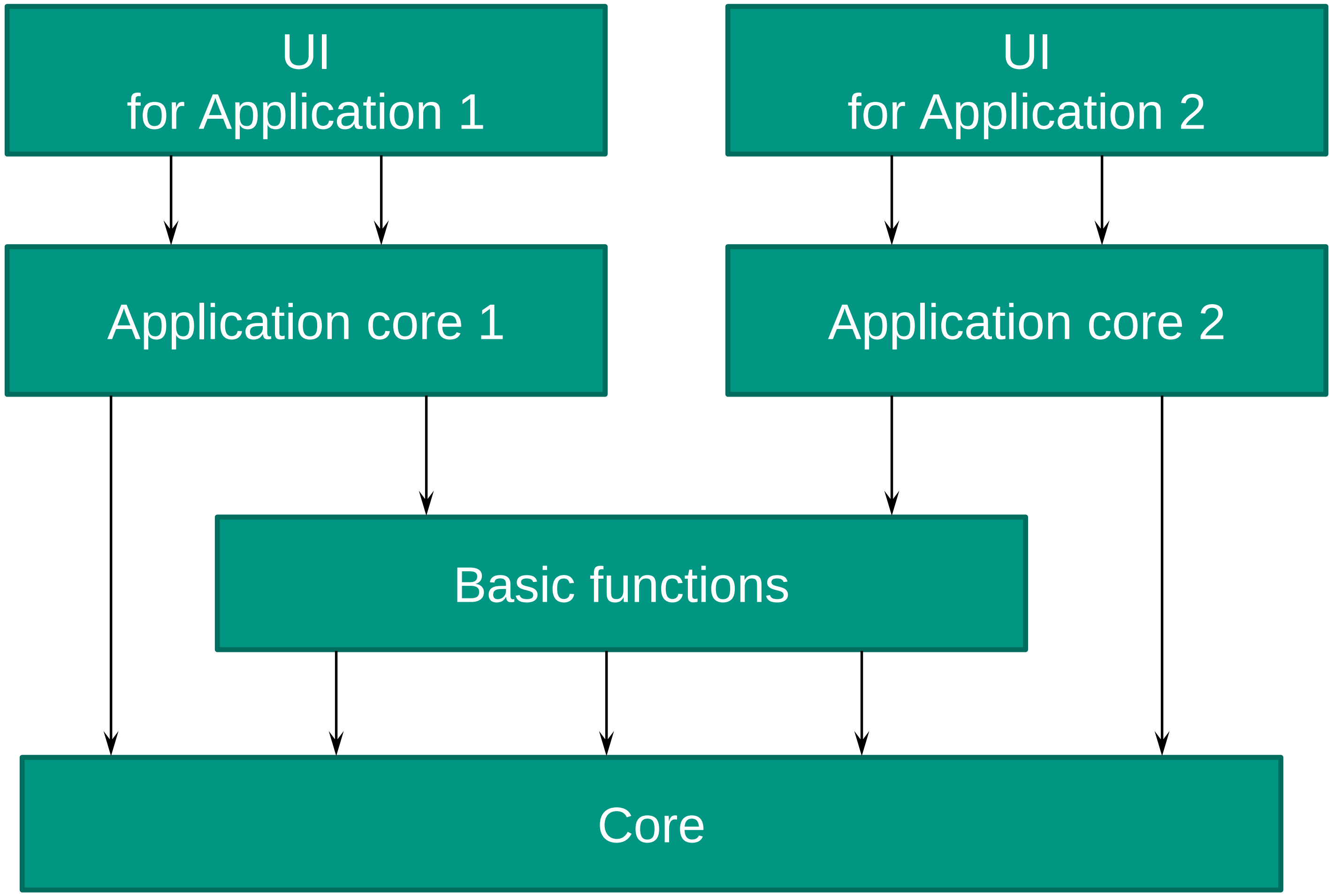
Layers

- A layer (tier) is a subsystem which provides services to other layers, with the following restrictions:
  - A layer only uses services of lower layers
  - A layer does not use higher layers
- A layer can be divided horizontally into several independent subsystems, also known as partitions
  - Partitions offer services for other partitions of the same layer

Examples

- Information systems (based on databases)
  - 3-tier architecture: A user interface, application core, and a database system.
- Operating systems: the most important layers are:
  - Process management,
  - Memory management,
  - File management,
  - Communication (network interfaces),
  - Command interface,
  - Graphical user interface,
  - Applications

4-tier architectures



Examples – 4-tier architectures

- 4-tier architectures are often used in **e-commerce web services**:
- The web browser represents the user interface.
- The web server delivers static HTML pages.
- An application service provider manages sessions and generates dynamic HTML pages.
- A database manages the data.

Layered architecture – Advantages

- Clear structuring in abstraction levels or virtual machines (the layers represent abstract machines)
- Not too restrictive for the designer, because besides a strict hierarchy he still has a liberal structuring possibility within the layers
- Easy testing
  - It supports reusability, changeability, maintainability, portability and testability (layers can be exchanged, added, ported, enhanced and reused, bottom or top testing).
  - Modify one layer won't affect the whole system

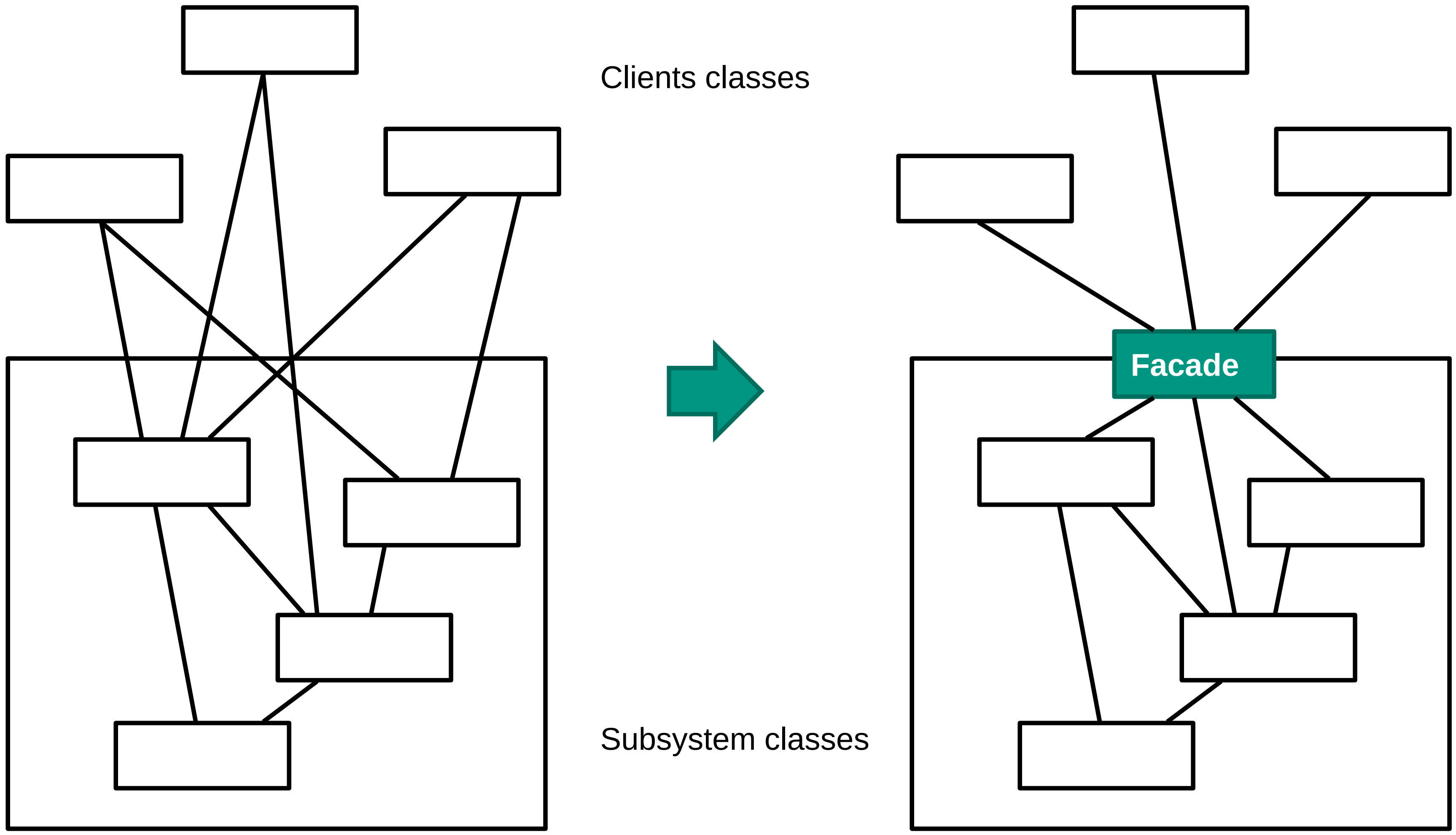
Layered architecture – Disadvantages/problems

- Performance
  - In the case of non-transparent layering, efficiency losses can occur because calls that exceed several layers must be passed through several layers and require multiple parameter transfers and result returns.
    - This also applies to error messages (but with catch/throw it is no longer a problem)
- Strict layering may be difficult in practice
  - Clearly distinguishable abstraction layers can not always be defined.
- There must be no chaos within a layer!

Layered architecture vs. design pattern *facade*

- A layer often consists of a large number of elements (packages, modules, classes, objects and functions), which are not all have to be provided to the layers above (because they are too complex to operate or not necessary).
- To provide a **clean** and **simplified** interface, a **facade** is used.
- The facade is one or more classes that contains only the available elements and delegates to the actual elements in the layer.
- The facade can also combine certain command sequences, which are often used together, into new methods (convenience methods).

Layered architecture vs. design pattern *facade*

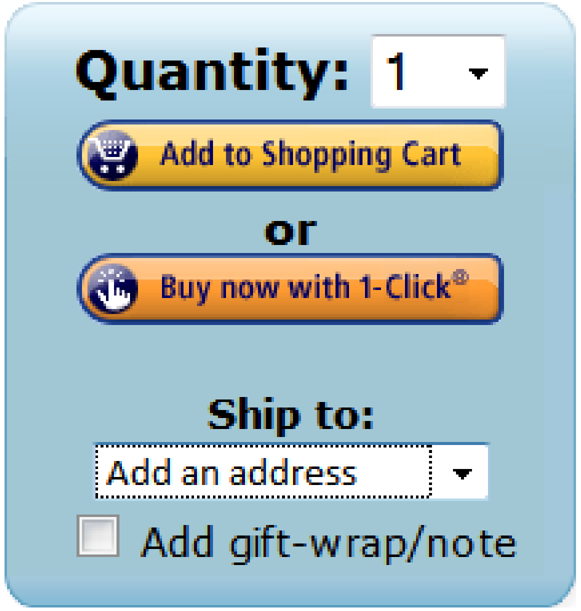


Convenience design patterns: Facade (Façade)

- **Purpose**
  - Provide a **unified interface** to a set of interfaces of a subsystem.
  - The facade class defines an abstract interface that simplifies the use of the subsystem.

Facade: Application example

- The 1-Click ordering system from Amazon.com



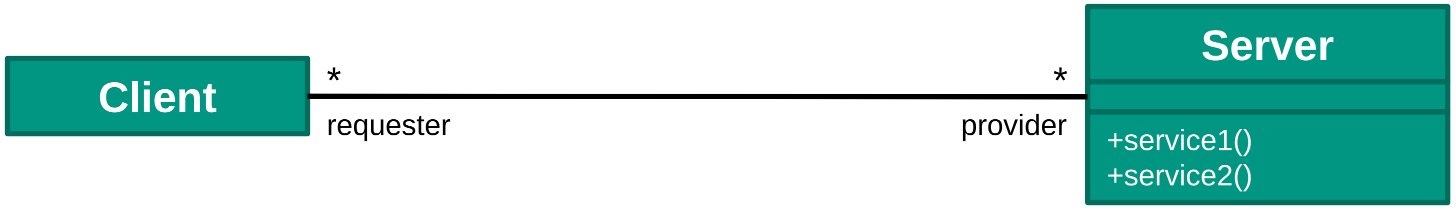
- Normally, the customer must specify his data (delivery / billing address, bank details, ...) during every order process.
- If the customer has already entered his data to Amazon and activated the 1-Click system, the customer can order the selected article with one click.

Facade: Applicability

- When a **simple interface** to a **complex** subsystem has to be offered.
    - A facade can provide a simple pre-set view of the subsystem that will satisfy most clients.
- When there are **many dependencies** between the clients and the implementation classes of an abstraction.
    - The introduction of a facade decouples the subsystems of clients and other subsystems.
- When subsystems should be split into layers.
    - We could use a facade to define an entry point to each subsystem layer.

Client/Server

- One or more servers provide services for other subsystems called clients.
- Each client invokes a function of the server which performs the desired service and returns the result.
    - For this, the client must know the interface of the server.
    - Conversely, the server does not need to know the client's interface.
- An example of a 2-tier, distributed architecture



Client/Server

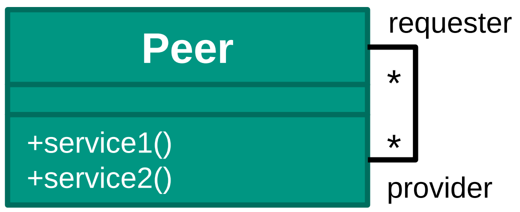
- Often used when designing database systems:
    - Front-end: user interface (client)
    - Back-end: database access and manipulation (server)
- Functions that the client performs:
    - Accept input from the user
    - Preprocessing of the inputs
- Functions that the server performs:
    - Data management
    - Data integrity and consistency
    - Security
- Client and server usually run on different computers (but not necessarily)

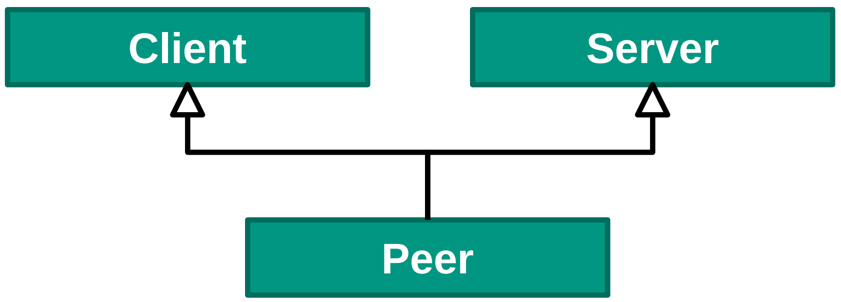
Client/Server – Example

- File transfer to an FTP server:
  - The client (e.g. an FTP program like Filezilla) initiates the transfer of a file.
  - The server responds to the client's request and receives or sends the file.

Peer-to-Peer

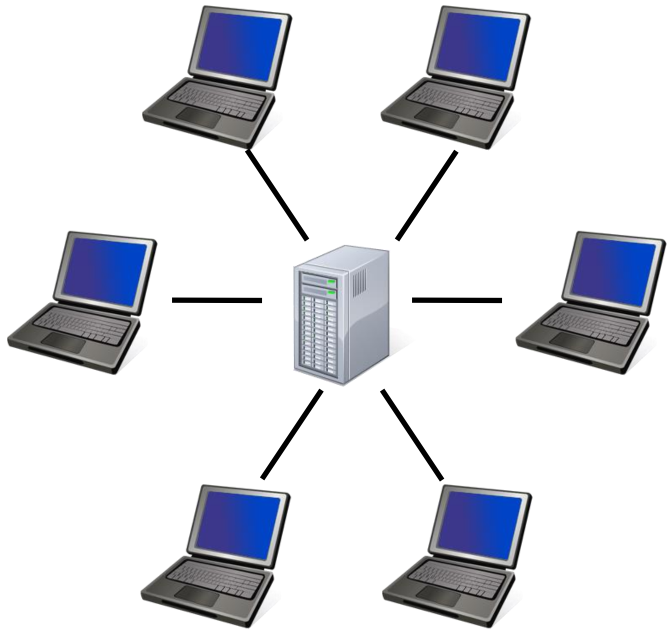
- Generalization of client / server architectural style.
- In a peer-to-peer network, all subsystems are equal ("peer" means "equal", "peer pressure").
- Peers run on different computers.
- Simplified form:



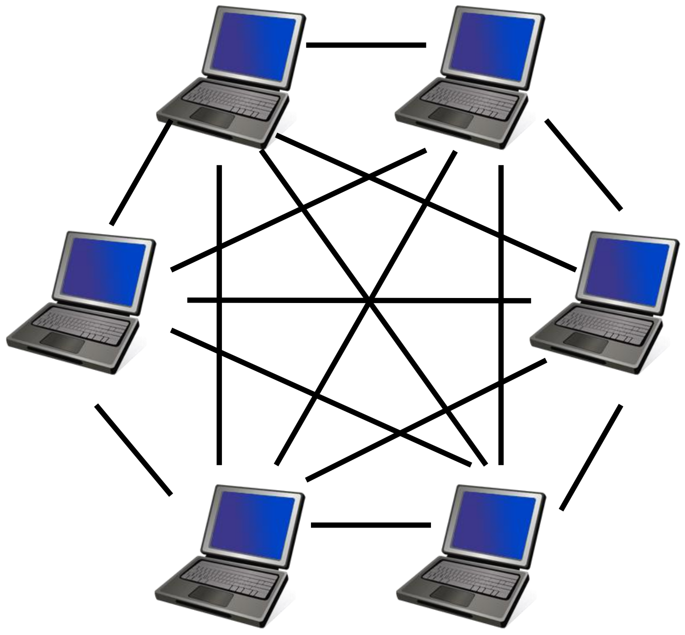


Peer-to-Peer vs. Client/Server

Client/Server



Peer-to-Peer



Peer-to-Peer – Features (1)

- Role symmetry:
  - Each peer is both client and server
- Decentralization:
  - There is no central coordination and no central database. Each partner usually only knows a subset of the other peers (his "neighborhood")
- Self-organization:
  - The overall behavior of the system is made up of the interaction between your individual partners
- Autonomy:
  - Peers make decisions autonomously and behave autonomously

Peer-to-Peer – Features (2)

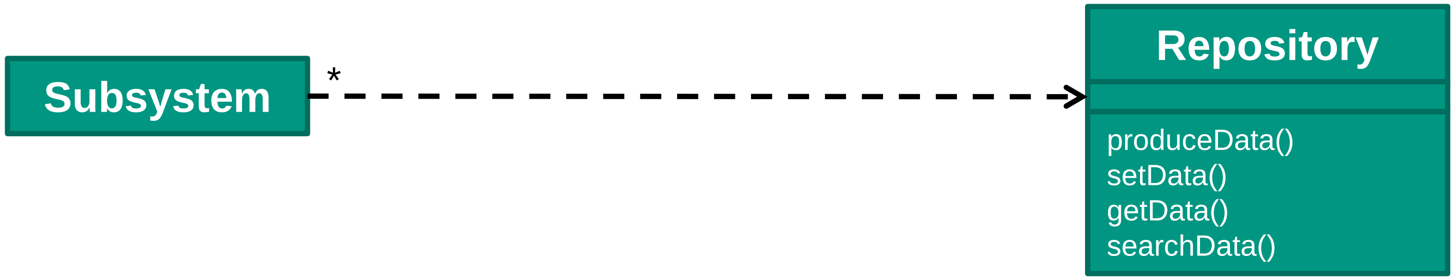
- Reliability:
  - Peers are unreliable (e.g. not always on). That is, mechanisms must be found which compensate for this unreliability
- Availability:
  - All data stored in the system must be available redundantly due to unreliability of (some) peers

Peer-to-Peer – Examples

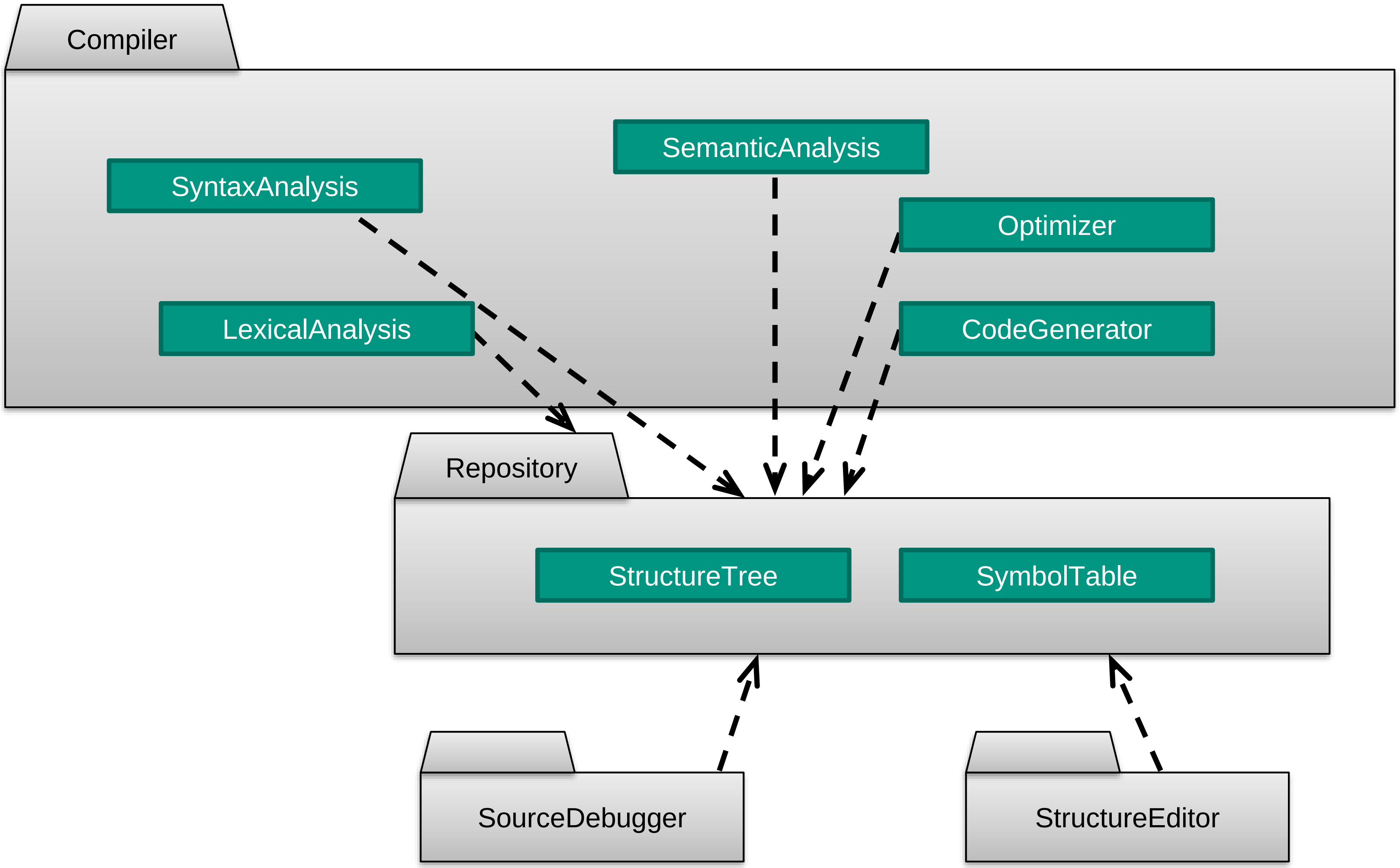
- At service level: Exchange of files in a peer-to-peer network (e.g. Bittorrent):
  - A peer is both client and server:
    - It can request files from other peers (client)
    - It can offer files (and other services) to other peers (server)
    - If an inquiry can not be satisfied, it will be passed on to another peer.
- At the network level: TCP/IP, DNS:
  - Requests are distributed across the physical network regardless of application.

Repository (Data-centered architecture)

- Subsystems modify data from a centralized data structure called Repository.
- Subsystems are loosely coupled and only interact via the Repository.
- Subsystems can access the Repository in parallel or in sequence. In a parallel case, **synchronization** is necessary (see Databases).
- Can be realized with local or remote access.
- Examples: Subversion, Git



Repository (Data-centered architecture) – Example (1)



Repository (Data-centered architecture) – Example (2)

- The three tools (compiler, editor and debugger) are executed by the user in any order, even in parallel.
- Repository ensures that concurrent accesses to shared data are synchronized.
  - Transaction concept: in concurrent accesses, it always looks as if they had been completely performed in a sequential order.
- The state of the repository can influence the execution of the various tools.
  - For example, editing files with the editor may be prohibited during a compilation process.

Repository (Data-centered architecture)

- **Strengths:**
- Little to no interaction among different data-processing components

- **Weaknesses:**
- Slow for different components to interact with each other
- The data repository is the single-point-of-failure, performance bottleneck

Model/View/Controller (MVC)

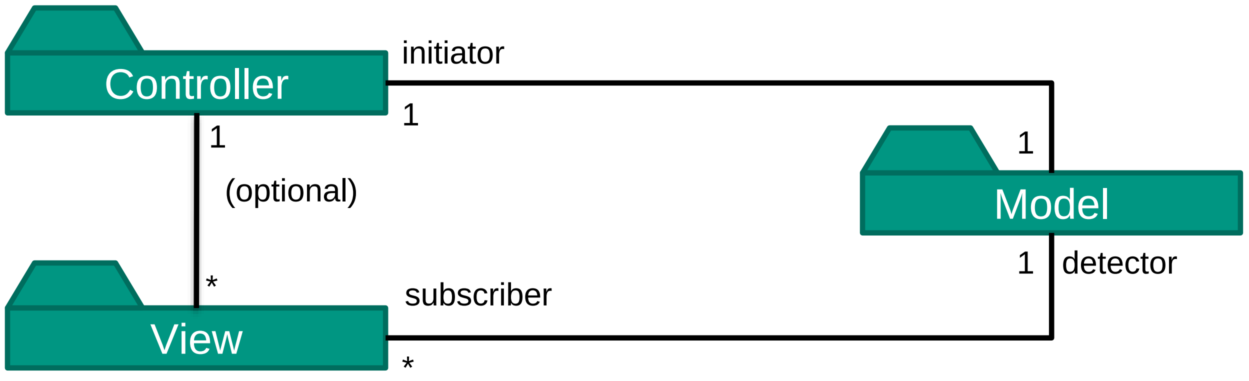
- **Problem:** Assume that a given system has a high coupling, so changes to the user interface result in changes to the data objects.
    - The user interface can not be re-implemented without changing the appearance of the data objects.
    - The data objects can not be reorganized without changing the user interface.
- What is the solution?

Model/View/Controller (MVC)

- **Solution:** The architecture style "Model-View-Controller", which separates data from their representation.
  - The subsystem for data storage is called the (data) **Model**.
    - In addition to the data, it often also contains the application logic.
  - The subsystem for data presentation is called **View**.
  - The subsystem for accepting user input and controlling the interaction between model and view is called a **Controller**.
    - The controller may additionally include some of the application logic.

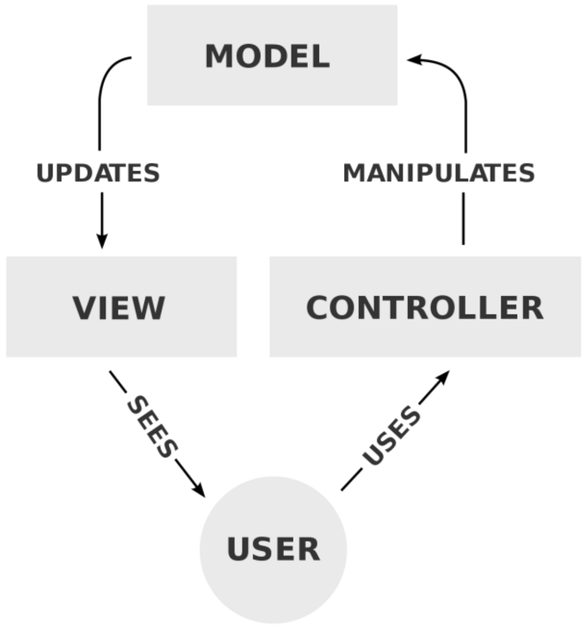
Model/View/Controller (MVC)

- **Model**: Responsible for application specific data.
- **View**: Responsible for the presentation of the objects of the application.
- **Controller**: Responsible for user interaction; updates the model; reports changes in the model data to the view(s).



Model/View/Controller (MVC)

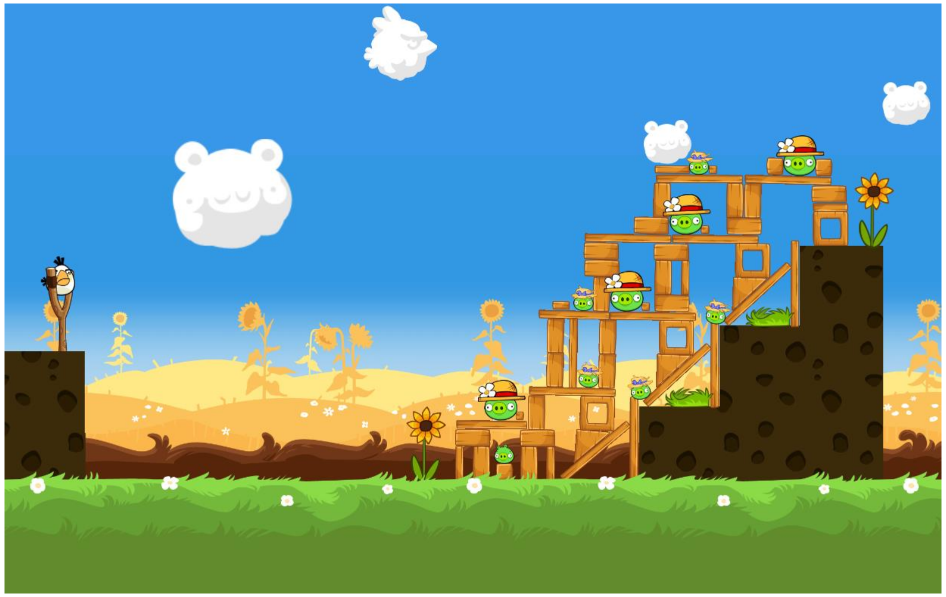
- The Model informs View that relevant data has changed, which View gets and displays. This update process is initiated by Controller.
- In general, View is combined with the buttons of Controller in a single graphic display. There may also be additional interaction between View and Controller (but not necessarily).
- With MVC, Model can be reused and provided with a different View implementation, e.g. for Mac or Linux. The View can also be reused if another application wants to display similar data.

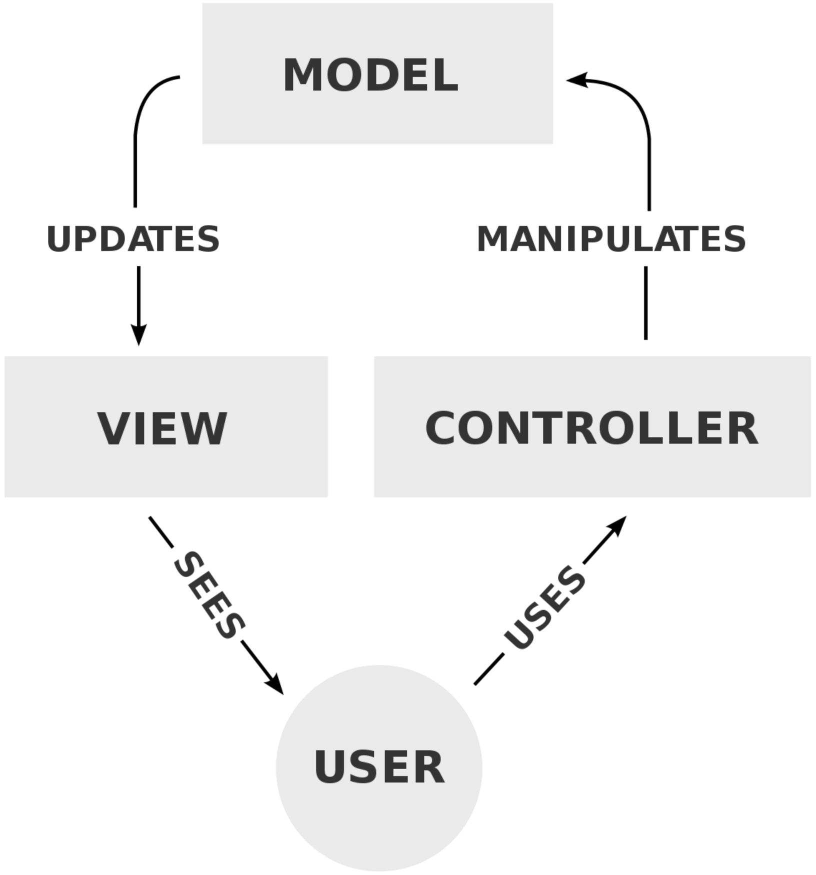


Examples?

MVC – Examples

- Angry birds





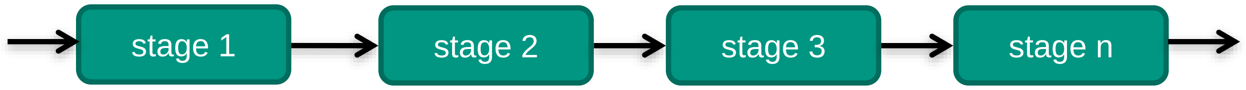
- Google search

Design concerns in MVC

- Where to put M, V, C, given multiple nodes?
  - M is most suitable for server machines
  - C is most suitable for client machines
  - V depends on network bandwidth between server and client

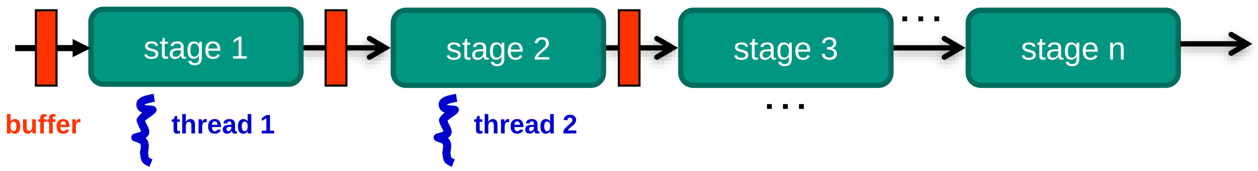
Pipeline (Pipe and Filter)

- Each stage or filter is a **self-running process** or **thread**, with its own **instruction counter**
- Data flows through the pipeline, each stage receiving data from the previous stage, processing it, and passing it on to the next one.
- Consecutive stages are connected to a size limited buffer to compensate for speed variations.



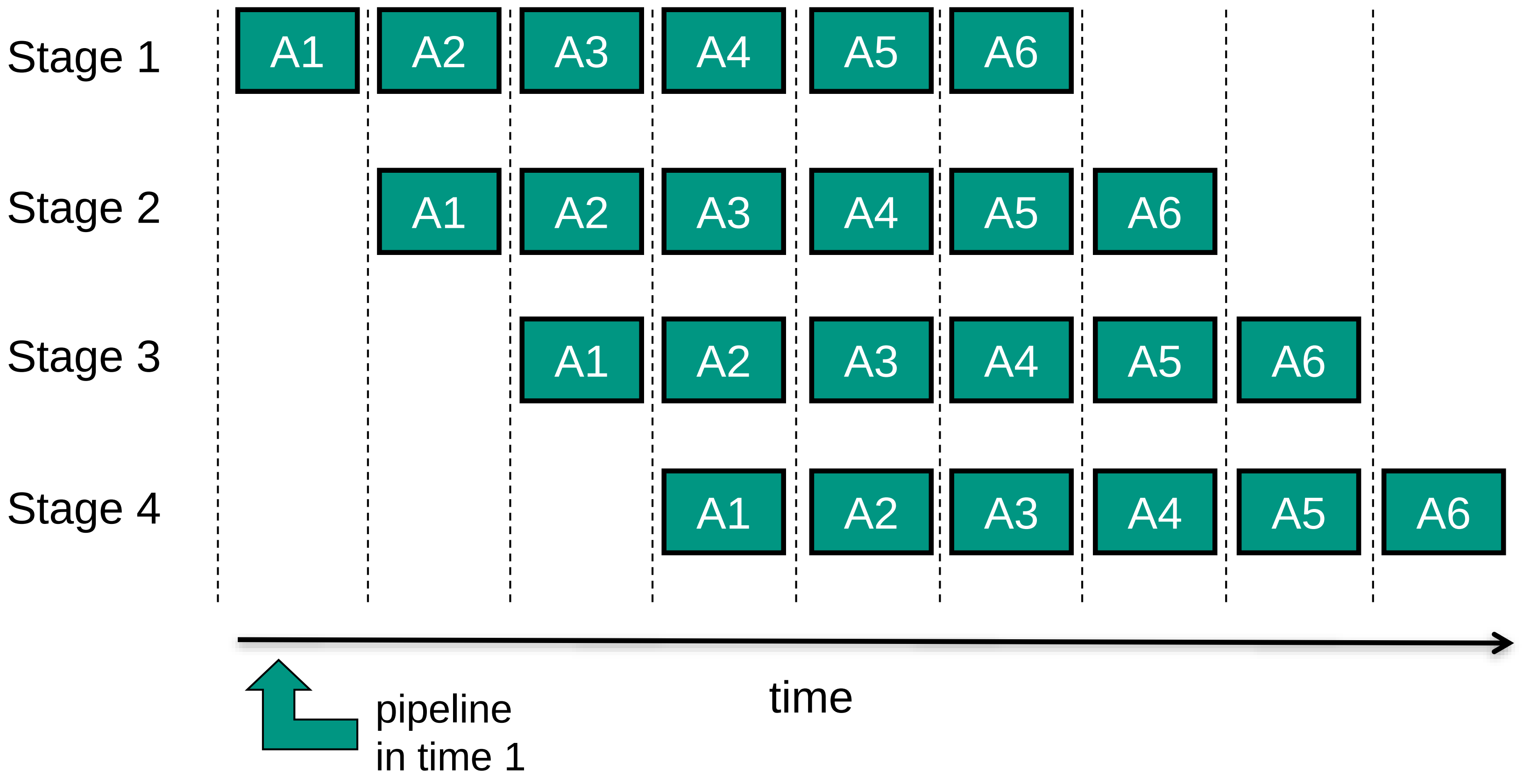
Pipeline (Pipe and Filter)

- In parallel computers, the individual stages can run (real) parallel and thus lead to an acceleration.
- In sequential computers, the individual processes are carried out alternately.



Pipeline (Pipe and Filter)

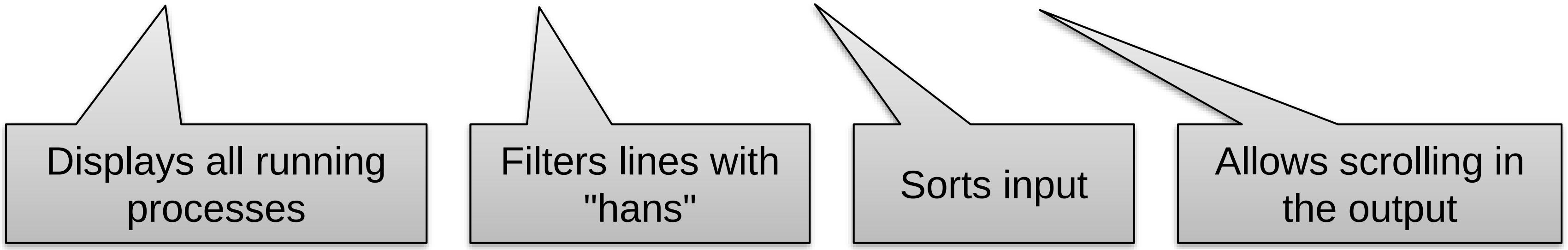
- Processing principle

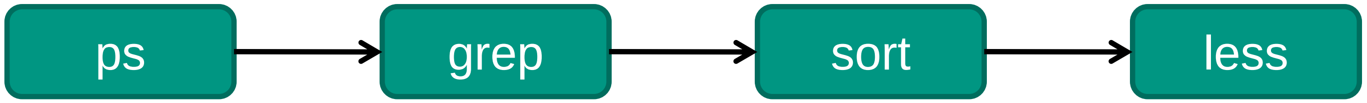


Pipeline (Pipe and Filter) – Examples

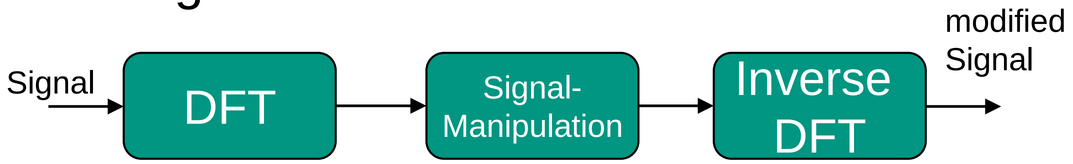
- An example of this architectural style is the Unix shell.

```
ps auww | grep hans | sort | less
```


Displays all running processes
Filters lines with "hans"
Sorts input
Allows scrolling in the output



- Signal processing



Pipeline (Pipe and Filter) – Applicability

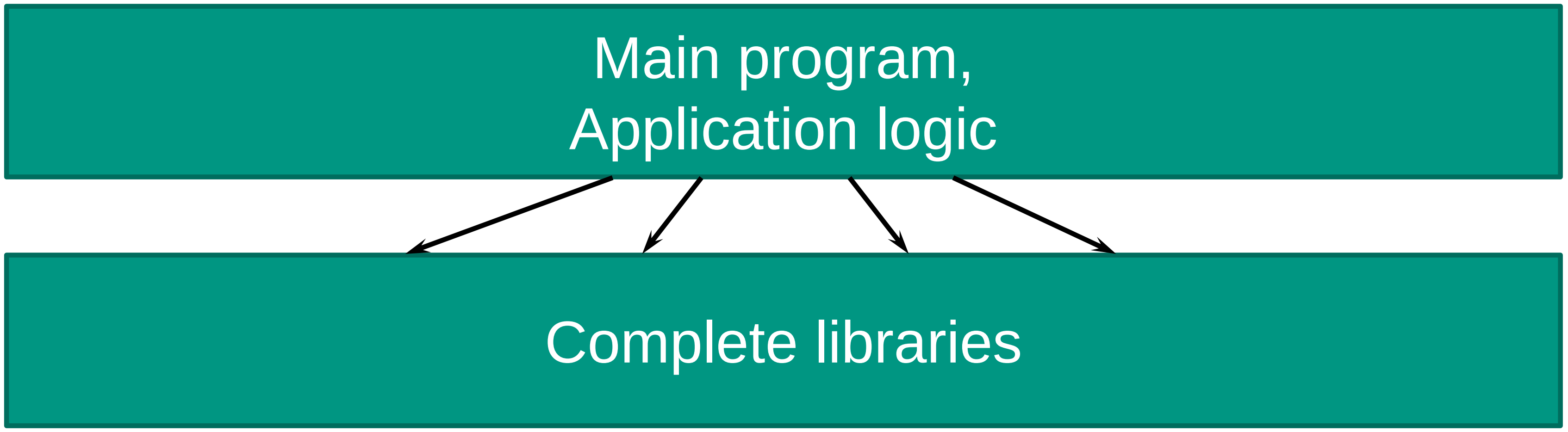
- Well suited for processing data streams, e.g. for video coding and decoding, video editing, translators, batch processing.
- For good performance on parallel computers, the individual stages should run approximately equally fast (irrelevant for single processors)
- **Strengths**
  - Filters can be reused
- **Weaknesses**
  - Data format compatibility among filters is crucial; format violation will break down the system

Framework architecture (Plug-ins architecture)

- Provides a (nearly) complete program that can be expanded by filling in planned "gaps" or extension points
- It contains the complete application logic, mostly even a complete main program
- Some of the classes in the program allow users to create subclasses overriding methods or implementing predefined abstract methods
  - The framework program takes care of the user-supplied extensions to be called correctly.
  - The extensions are also called **plug-ins**.

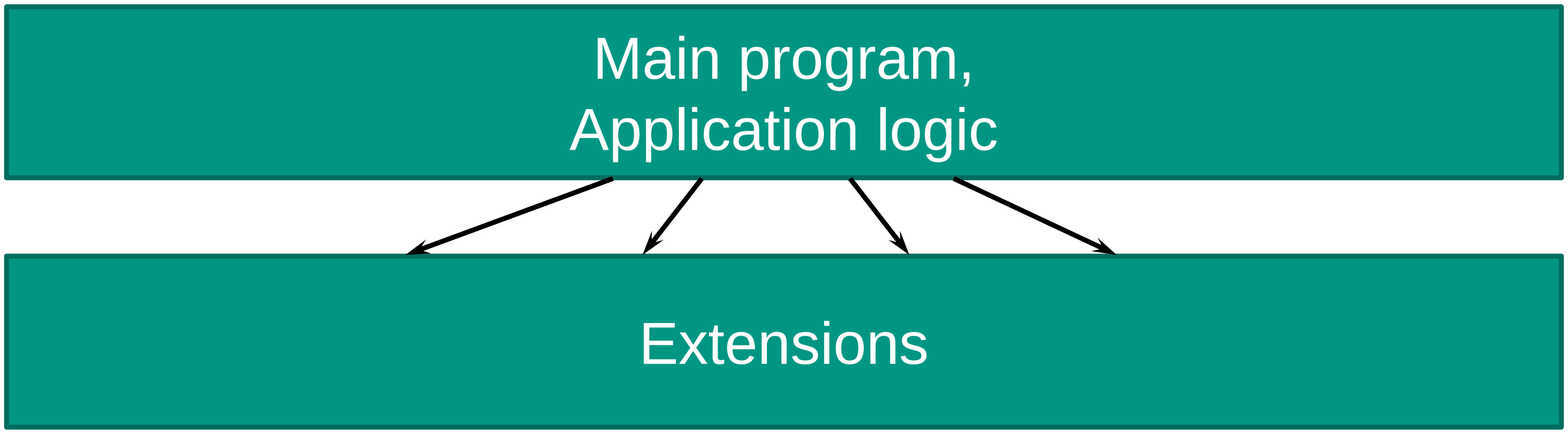
Framework – Structure (1)

- Conventional system structure:
  - Manufacturer supplies libraries
  - User writes main program and application logic



Framework – Structure (2)

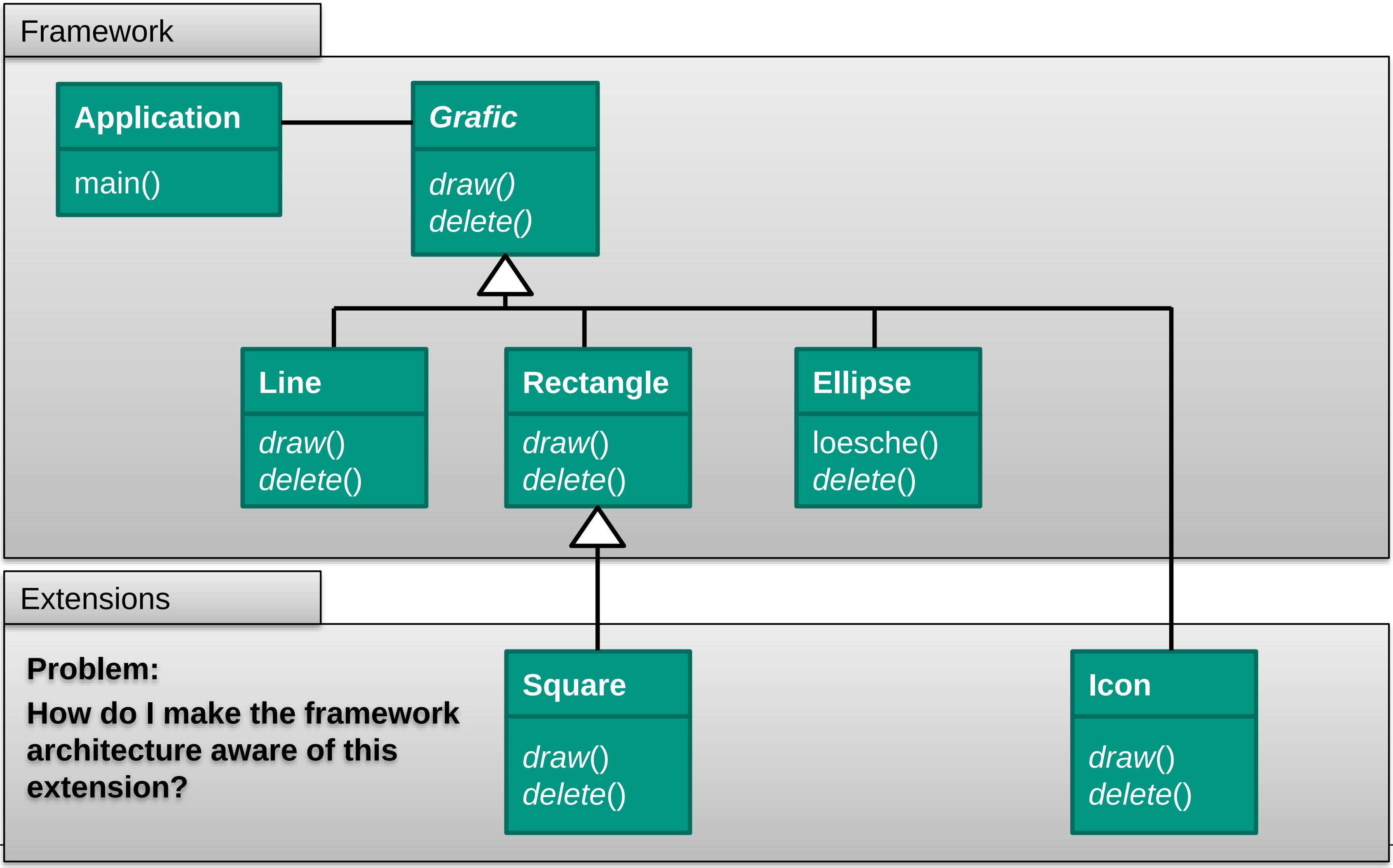
- **Framework**:
  - A supporting program follows the "Hollywood principle": "Don't call us - we call you".
  - The main program already exists and calls the extensions of the users. (extensions must fit into the constraints of an existing framework)



Framework – Example (1)

- A drawing framework program provides a class of graphics with some subclasses (e.g., line, rectangle, ellipse, etc.).
- The user may create new **subclasses** of graphics and their subclasses, such as square or icon, but must provide a `draw()` method.
- The framework program then ensures that objects of the new classes can be created correctly, positioned on the drawing board, moved, saved, etc.

Framework – Example (2)



Problem:

How do I make the framework architecture aware of this extension?

Framework – Example (3)

- **Eclipse**: open source development platform
- Consists of a relatively small core with a large number of extension points for plug-ins, which in turn can offer further extension points.
- Large parts of the standard distribution of Eclipse consist of plug-ins.
- Plug-ins are saved as jar-files whose manifest-file is read by the Eclipse core to obtain the extension class identifiers.

Framework – Applicability

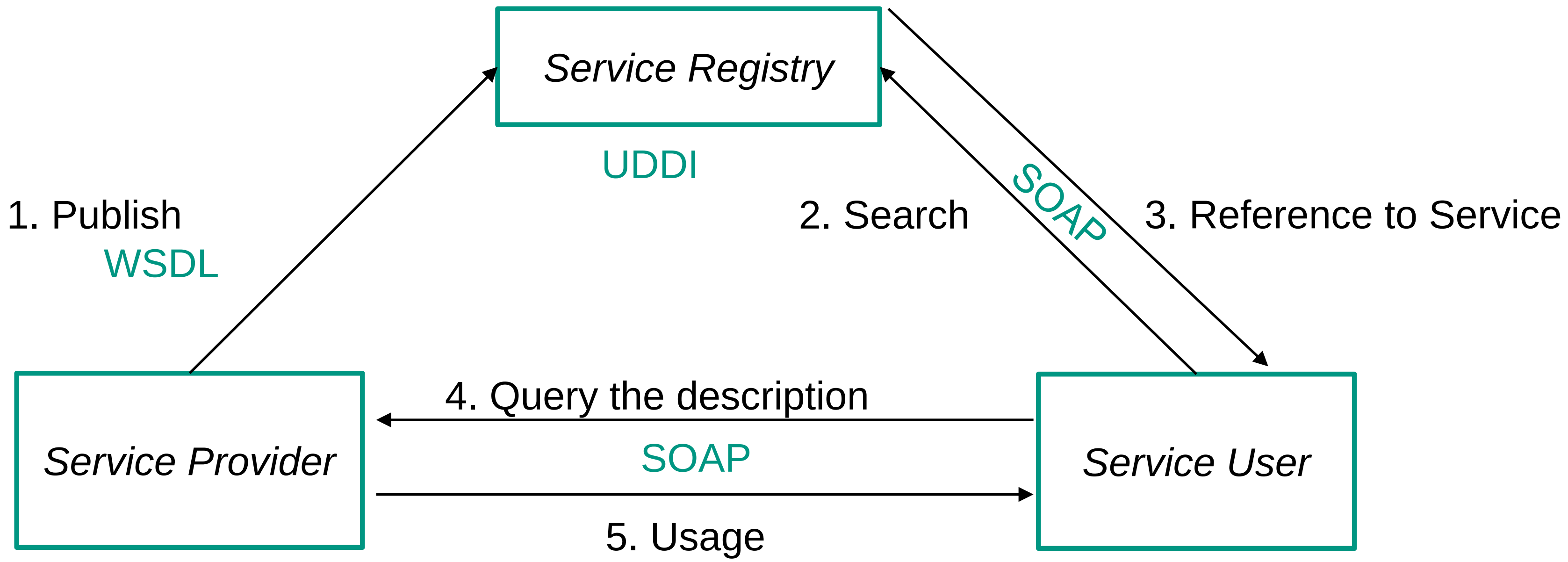
- When a **basic version** of the application should already be functional
- When extensions should be possible that behave consistently
  - Application logic in the framework program
- When complex application logic should not be reprogrammed
- The design patterns strategy, factory method, abstract factory, and template method are often needed in framework architectures (see design patterns)

Service Oriented Architecture (SOA)

- SOA is an architecture style that assembles applications from (independent) services
- SOA is an abstract concept of a software architecture
- Services are considered as central elements of a company (keyword: services)
  - Provide encapsulated functionality to other services and applications
  - Allow capsules from legacy systems and (later) exchanges
  - Service description (interface) is available in standardized form
  - Language and technology independent

Service model as the core of a SOA

- Required functionality can be integrated in the form of a service at runtime
- The basis for **dynamic binding** is the service model



SOA model

- **Service provider**: Owner of a service with description and implementation
- **Service consumer**: Discovers and invokes offered service
- **Service registry**: Manages information of provided services
- The provider creates a service in the **Web Service Description Language** (WSDL)
- The **Universal Description, Discovery and Integration** (UDDI) standard enables publishing and discovery of the service
- All communication takes place via the **Simple Object Access Protocol** (SOAP) or XML-RPC or JSON or …

Static binding vs. dynamic binding (recap)

- **Static binding** (also known as early binding) occurs during compile time and uses type information (Class in Java) for binding
  - Used by private, final and static methods and variables (bonded by compiler)
  - Overloaded methods are bonded using static binding
- **Dynamic binding** (also known as late binding) occurs during runtime and uses object to resolve binding
  - Used by virtual methods that are bonded during runtime based upon runtime object.
  - Overridden methods are bonded using dynamic binding at runtime

Features and goals of SOA

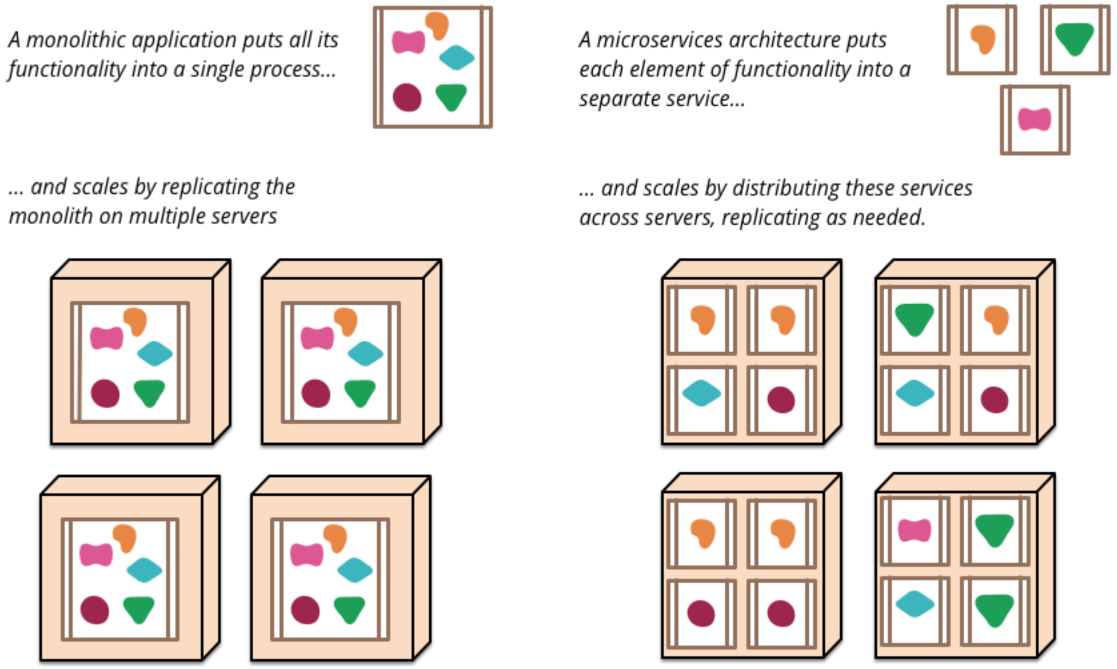
- Loose coupling
  - The goal is the simple detachment and replacement of a service at runtime
  - Dynamic binding is enabled by the service directory as a central part of the service model
- Support of business processes
  - Services encapsulate business-relevant functionality
  - (Complex) application = composition of services
- Use of (open) standards
  - Programming language and platform independent provision of services and service compositions

SOA extension: Event-driven architecture

- Event-driven architecture can be an extension or variant of service-oriented architecture (SOA)
- Services can be activated by triggers fired on incoming events.
- Event-driven architecture:
  - Processing loop
  - Event queue
  - Call-back

Microservices

- Monolithic application vs. Microservices


A monolithic application puts all its functionality into a single process...
... and scales by replicating the monolith on multiple servers
A microservices architecture puts each element of functionality into a separate service...
... and scales by distributing these services across servers, replicating as needed.

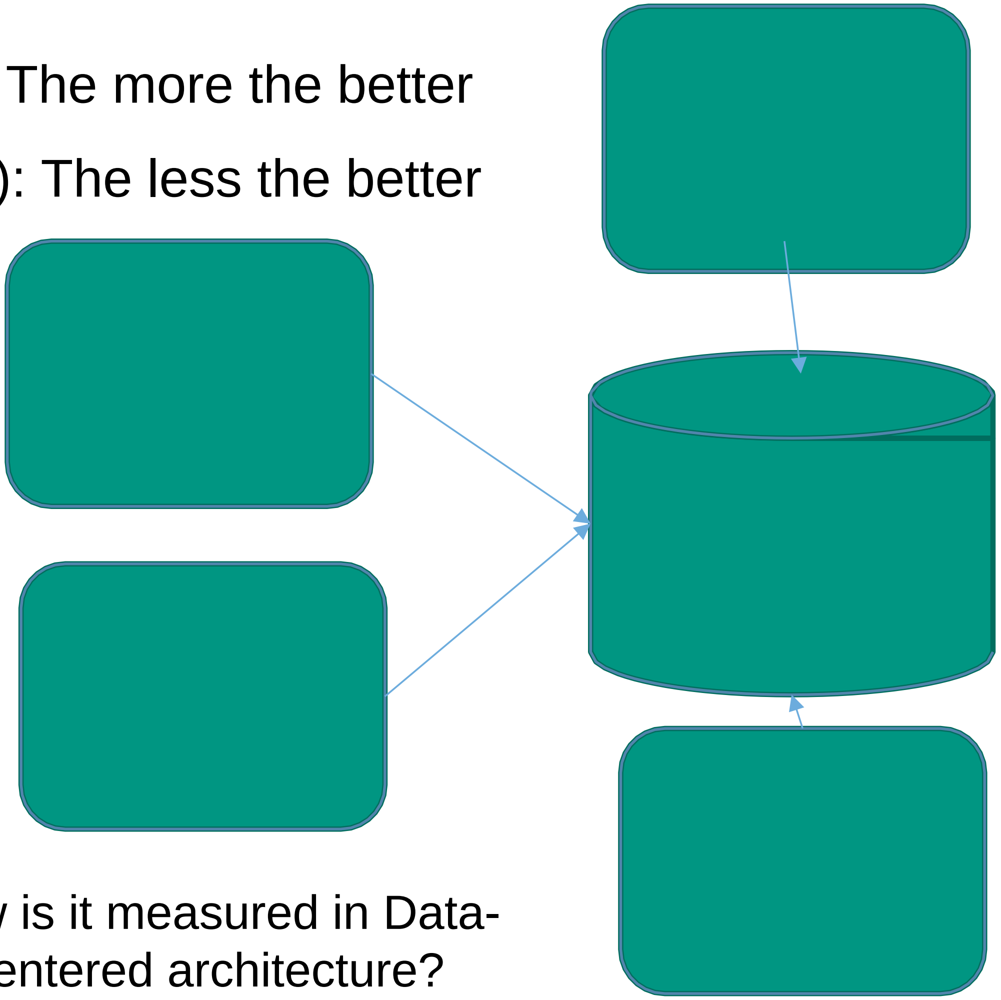
A change made to a small part of the application, requires the entire monolith to be rebuilt and deployed.

https://martinfowler.com/articles/microservices.html

Services are independently deployable and scalable, each service also provides a firm module boundary, even allowing for different services to be written in different programming languages.

How to assess a software architecture?

- **Cohesion** (within component): The more the better
- **Coupling** (across components): The less the better



How is it measured in Data-centered architecture?

Summary

- Design overview
- Architectural style
    - Layered architecture
    - Client/server
    - Peer-to-peer
    - Model View Controller
    - Pipeline (pipe and filter)
    - SOA