Architectural Pattern

April 2014

Ying SHEN

SSE, Tongji University

Lecture objectives

This lecture will enable students to

- define what is meant by “architectural pattern”
- list several examples of architectural pattern and describe the key characteristics of each
- give examples of how the use of particular architectural patterns helps achieve desired qualities

Architectural patterns

An architectural pattern is a description of component types and a pattern of their runtime control and/or data transfer.

A pattern

- is found repeatedly in practice
- is a package of design decisions
- has known properties that permit reuse
- describes a class of architectures

Architectural patterns

An architectural pattern establishes a relationship between:

- A context
  - A recurring, common situation in the world that gives rise to a problem.
- A problem
  - The problem, appropriately generalized, that arises in the given context.
- A solution
  - A successful architectural resolution to the problem, appropriately abstracted.

Architectural patterns

The solution for a pattern is determined and described by

- A set of element types
- A set of interaction mechanisms or connectors
- A topological layout of the components
- A set of semantic constraints covering topology, element behavior, and interaction mechanisms

It should also make clear what quality attributes are provided.

Architectural pattern catalog

The catalog is not meant to be exhaustive.

There is no unique, non-overlapping list.

Systems exhibit multiple patterns at once.

- A web-based system:
  - A three-tier client-server architectural pattern
  - and replication (mirroring), proxies, caches, firewalls, MVC…
  - more patterns and tactics

Applying a pattern is not an all-or-nothing proposition.

- Violate them in small ways to have a good design tradeoff
- Example: layered pattern

Architectural pattern catalog

Patterns can be categorized by the dominant type of elements:

- module patterns
  - Layered pattern
- component-and-connector (C&C) patterns
  - Broker
  - MVC
  - Pipe-and-filter
  - Client-server
  - Peer-to-peer
  - Service-oriented
  - Publish-subscribe
  - Shared-data
- allocation patterns
  - Map-reduced

Architectural pattern catalog

Patterns can be categorized by the dominant type of elements:

- module patterns
  - Layered pattern
- component-and-connector (C&C) patterns
  - Broker
  - MVC
  - Pipe-and-filter
  - Client-server
  - Peer-to-peer
  - Service-oriented
  - Publish-subscribe
  - Shared-data
- allocation patterns
  - Map-reduced

Module pattern: Layered pattern

Context: All complex systems experience the need to develop and evolve portions of the system independently.

Problem: The software needs to be segmented.

- Modules are developed and evolved separately to support portability, modifiability, and reuse.

Solution: The layered pattern divides the software into units called layers.

- Each layer is a grouping of modules that offers a cohesive set of services.
- The *allowed-to-use* relationship is unidirectional.

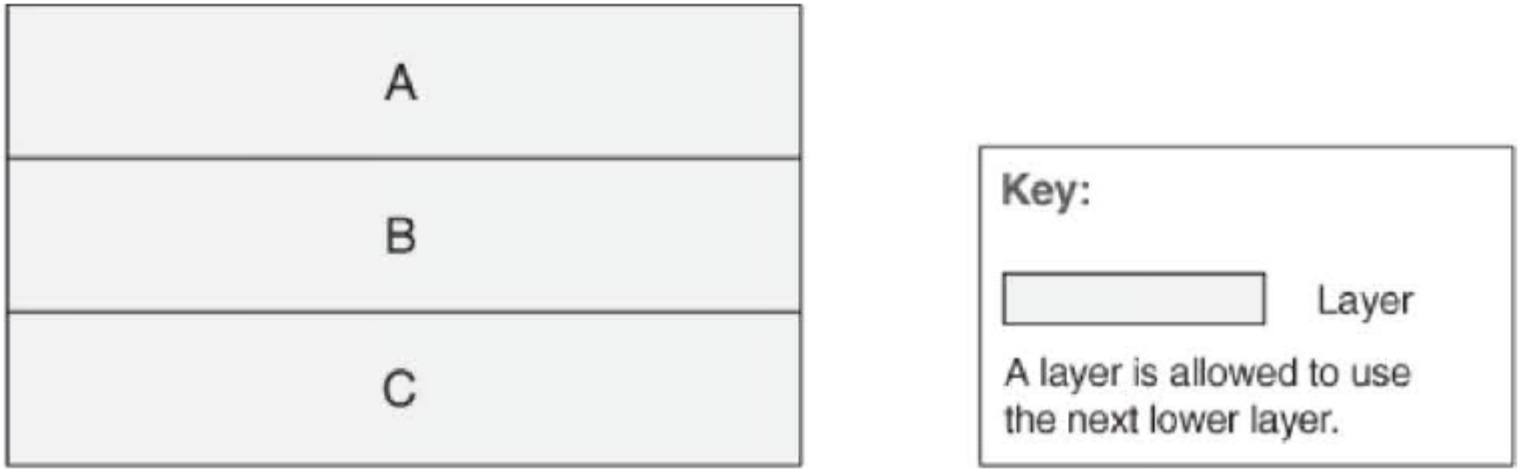
Module pattern: Layered pattern

The layers are created to interact according to a strict ordering relation.

- (A, B) means layer A is allowed to use any of the public facilities provided by layer B.
- Normally only next-lower-layer uses are allowed.
- A higher layer using modules in a nonadjacent lower layer is called *layer bridging*.
  - Portability and modifiability will be harmed.
- Upward usages are not allowed.

Module pattern: Layered pattern

Layers are almost always drawn as a stack of boxes. The *allowed-to-use* relation is denoted by geometric adjacency and is read from the top down.



Weakness

The addition of layers adds up-front cost and complexity to a system.

Incorrect design of layers will not provide the lower-level abstraction.

Layers contribute a performance penalty.

Architectural pattern catalog

Patterns can be categorized by the dominant type of elements:

- module patterns
  - Layered pattern
- component-and-connector (C&C) patterns
  - Broker
  - MVC
  - Pipe-and-filter
  - Client-server
  - Peer-to-peer
  - Service-oriented
  - Publish-subscribe
  - Shared-data
- allocation patterns
  - Map-reduced

C&C pattern: Broker pattern

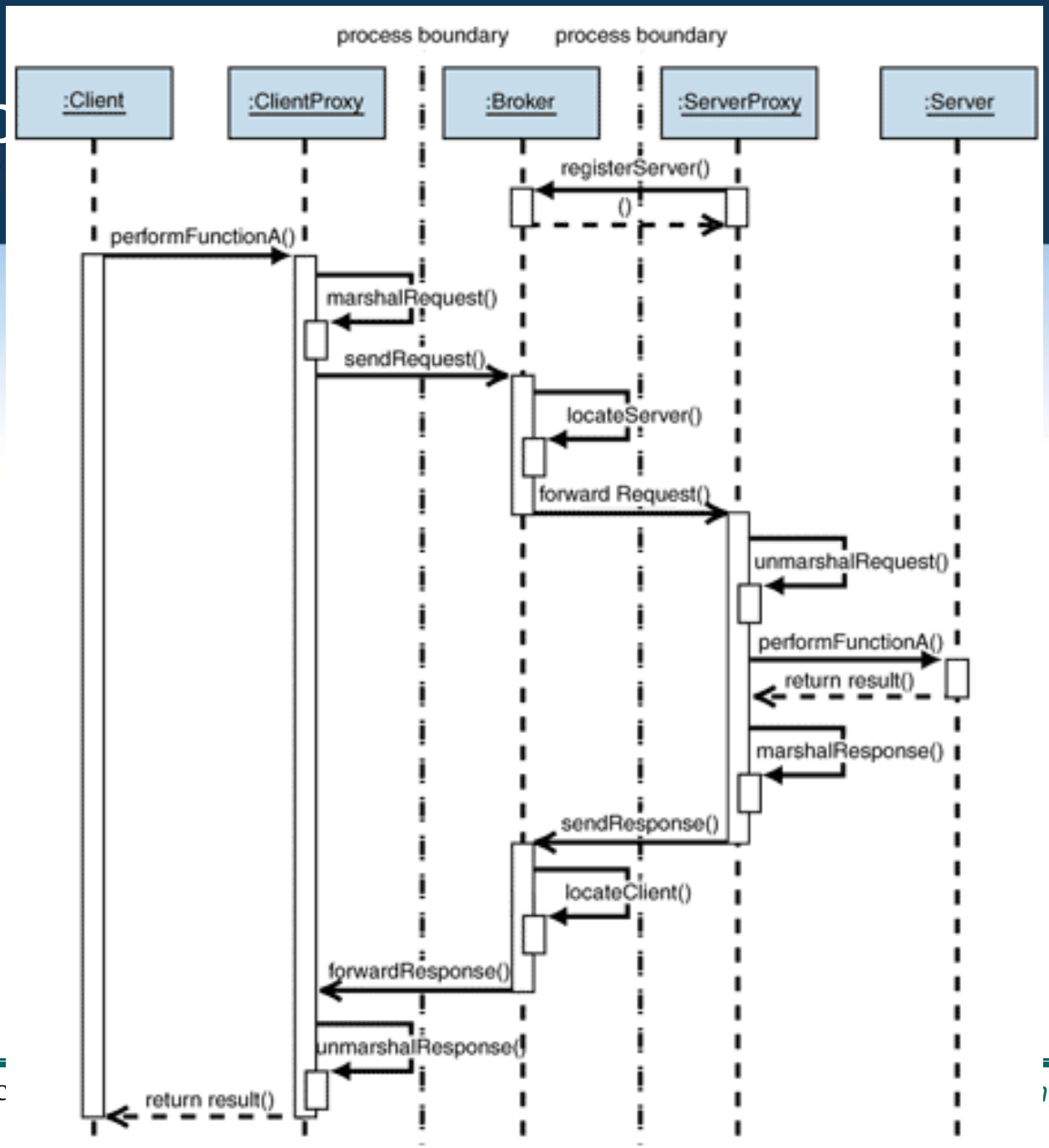
Context: Many systems are constructed from a collection of services distributed across multiple servers.

- The systems will interoperate with each other
- The availability of the component services

Problem: How do we structure distributed software so that service users do not need to know the nature and location of service providers.

Solution: The broker pattern separates users of services (clients) from providers of services (servers) by inserting an intermediary, called a broker.

C&C p

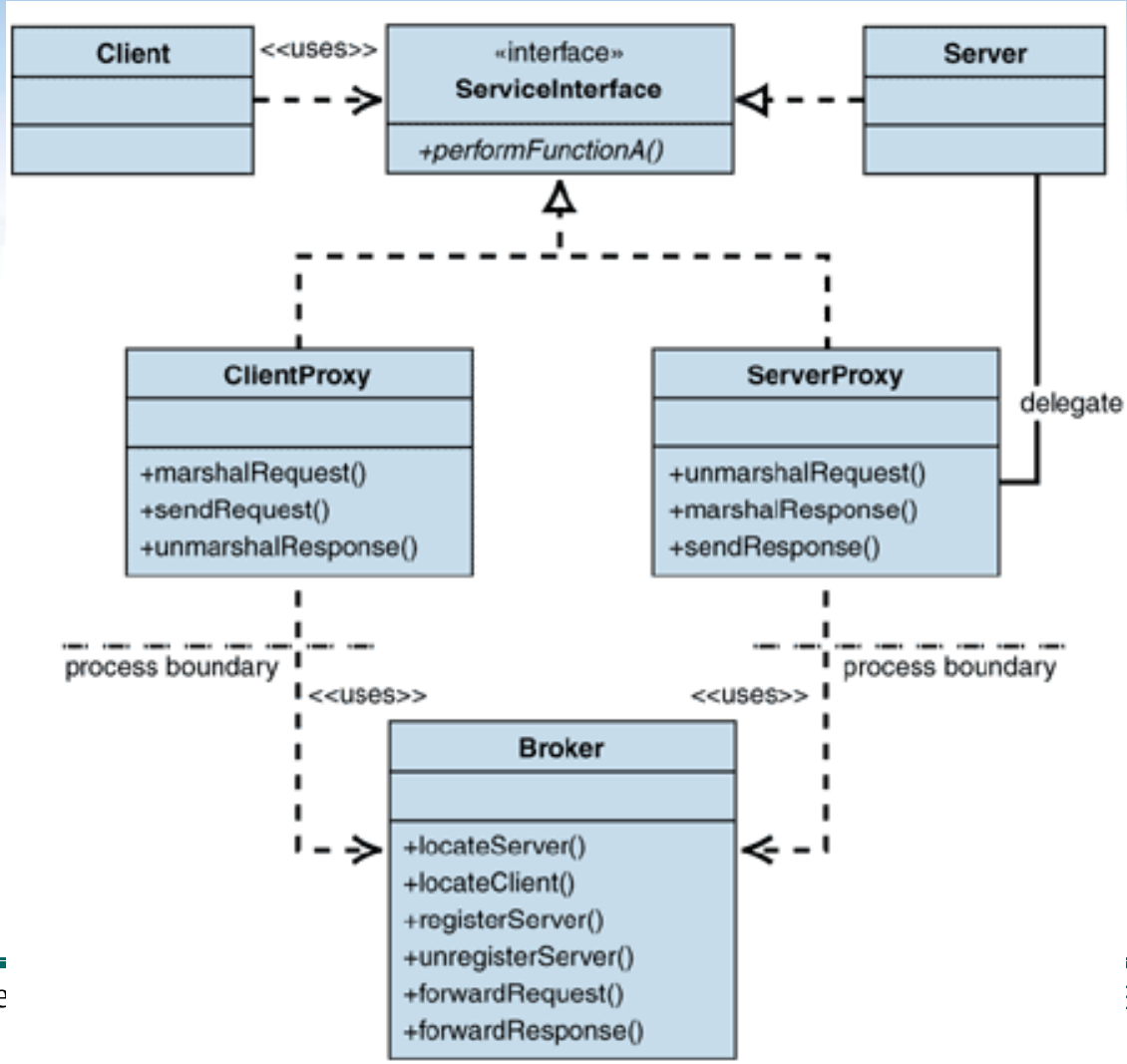

process boundary
process boundary
:Client
:ClientProxy
:Broker
:ServerProxy
:Server
registerServer()
()
performFunctionA()
marshalRequest()
sendRequest()
locateServer()
forward Request()
unmarshalRequest()
performFunctionA()
return result()
marshalResponse()
sendResponse()
locateClient()
forwardResponse()
unmarshalResponse()
return result()

C&C pattern: Broker pattern

The client remains completely ignorant of the identity, location, and characteristics of the server.

- A server is unavailable
- A server is replaced

C&C pattern: Broker pattern



Weakness

Brokers add a layer of indirection, and hence latency, between clients and servers, and that layer may be a communication bottleneck.

The broker can be a single point of failure.

A broker adds up-front complexity.

A broker may be a target for security attacks.

A broker may be difficult to test.

C&C pattern: Model-View-Controller Pattern

Context: User interface is the most frequently modified portion.

- Keep modifications to the user interface software.
- Users often wish to look at data from different perspectives.
  - A bar graph or a pie chart

Problem:

- How to separate user interface functionality from application functionality?
- How to create, maintain, and coordinate multiple views of the UI?

C&C pattern: Model-View-Controller Pattern

Solution: Application functionality is separated into three kinds of components:

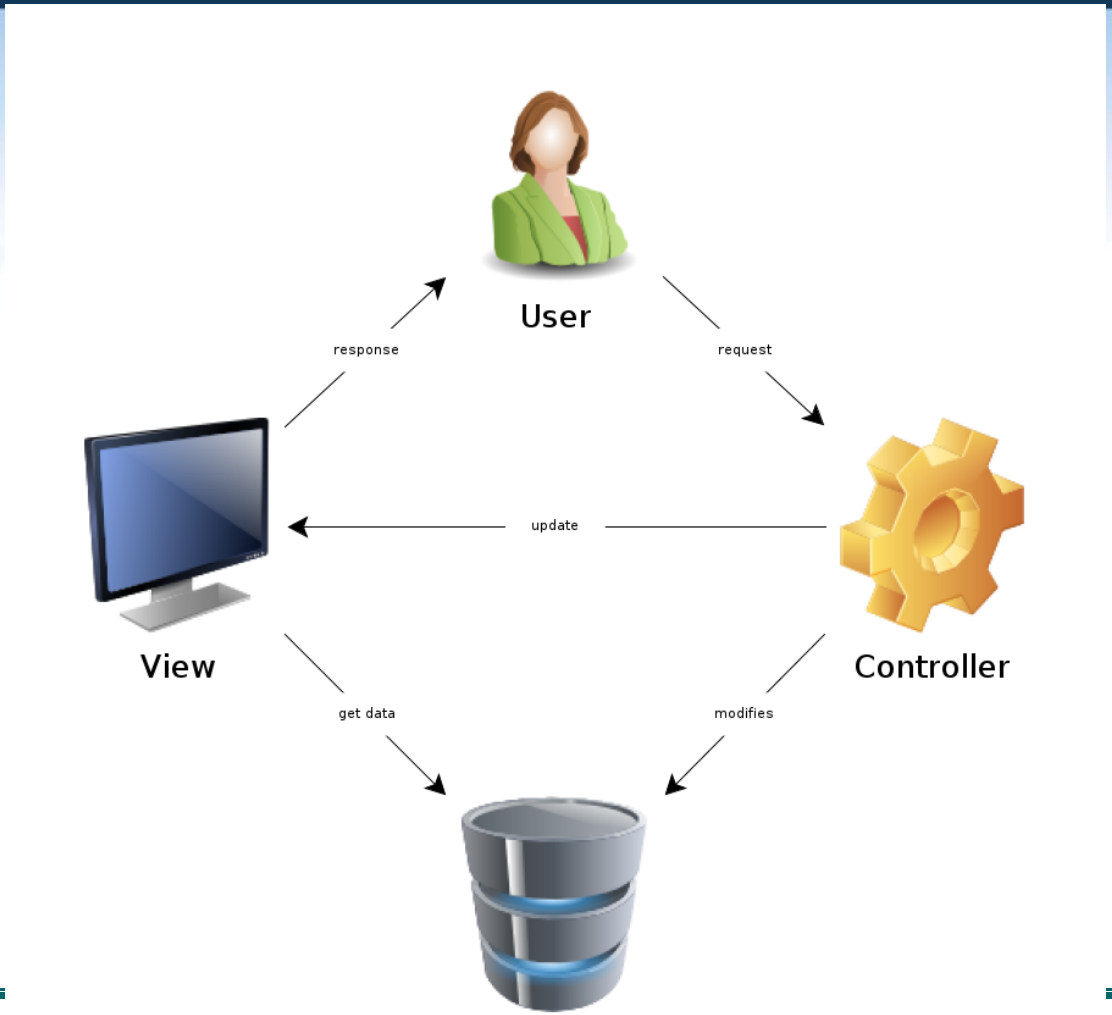
- A *model*, encapsulates the behavior and data of the application domain
- A *view*, renders the model for presentation
- A *controller*, reacts on user input, modifies the model and dispatches to the view

Both controller and view depend on the model.

Controller and view are part of the UI.

There must be at least one instances of each component.

C&C pattern: Model-View-Controller Pattern



C&C pattern: Model-View-Controller Pattern

MVC is often used for web applications.

Many existing frameworks:

- JavaServer Faces (JSF), Struts, CakePHP, Django, Ruby on Rails, ...

MVC - Model

The model:

- Encapsulates the application state
- Response to state queries
- Exposes application functionality
- Notify view of changes
- Note: Notification only necessary, if the model and view realize an observer pattern

A model can be associated with many controllers.

MVC - View

The view:

- Renders the model
- Requests updates from model
- Prepares the user interface for the controller
- Usually multiple views

MVC - Controller

The controller:

- Defines application behavior
- Manipulates the model
- Selects view for response

Advantages

Separation of concerns, helps reusability

Multiple different user interfaces without changes to the model

Helps configurability (as interface changes are easier, with less expected side effects than changes to the application logic)

Disadvantages

- Increases the complexity by additional components
- If updates to the view are based on notifications, it might be hard to find errors
- In this cases, it is hard to ensure a good usability (no control when an update happens)

C&C pattern: Pipe-and-filter pattern

Context: Many systems are required to transform streams of discrete data items.

- It is desirable to create independent, reusable components.

Problem: How to design a system composed by reusable, loosely coupled components with simple, generic interaction mechanisms?

C&C pattern: Pipe-and-filter pattern

Solution: The system can be designed as successive transformations of streams of data.

Data enter the system and then flows through the components one at a time until

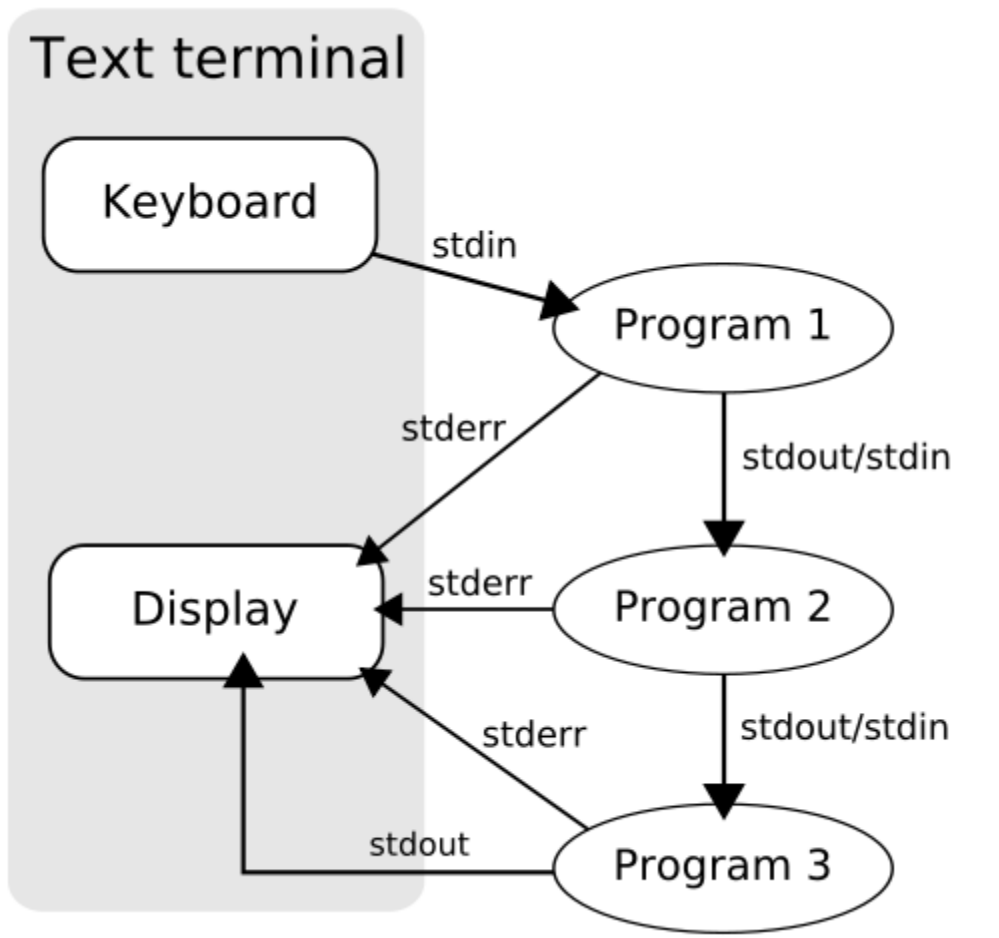
- the data is assigned to some final destination (output or a data store).

The goal is to achieve the quality of reuse and modifiability.

Example: Unix command line pipes

```
% program1 | program2 | program3
% ls -l | grep key | more
```

C&C pattern: Pipe-and-filter pattern



C&C pattern: Pipe-and-filter pattern

Conceptually filters consume data from inputs and write data to outputs.

Filters do not know anything about other filters.

Ideally they are completely independent from each other.

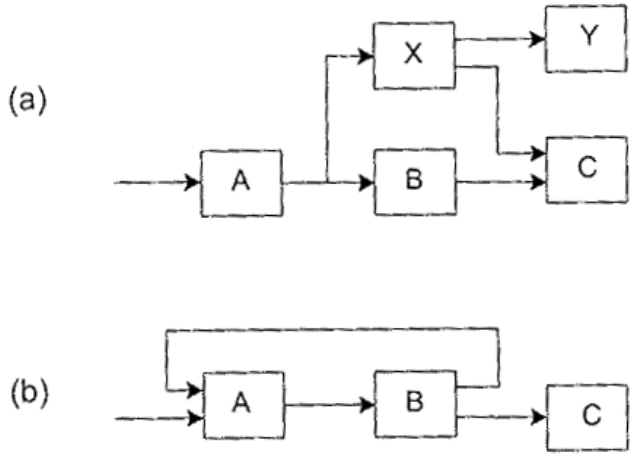
Data flows in streams: good for processing of images, audio, video, ...

C&C pattern: Pipe-and-filter pattern

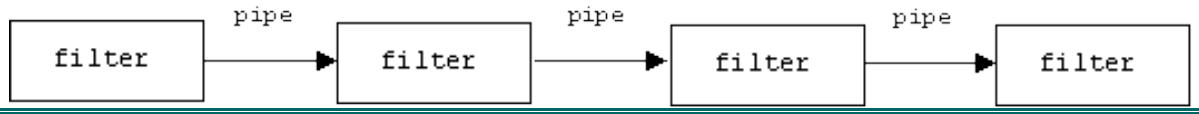
Variations: **structural** and **communicational**

Structural: more complex topologies might be used

- E.g. loops, branches, more than one input, ...



- Term **pipeline** used for linear sequence of filters



C&C pattern: Pipe-and-filter pattern

Variations: **structural** and **communicational**

Communicational: are filters blocked and wait for data?

Term **bounded pipe** for limited amount of data in the pipe

C&C pattern: Pipe-and-filter pattern

What is the data-structure within the pipe?

All components in the pipe have to agree

Term **typed pipe** if data is structured

The more specific the data-structures are, the tighter the coupling

Advantages

Pipes are conceptually simple (helps maintainability)

Components can be reused

Easy to add and remove components (helps evolvability)

Allow injection of special components to address cross-cutting concerns

- E.g. monitor throughput, logging, ...

Allow concurrent/parallel execution (helps scalability)

Disadvantages

Pipes often lead to batch processing

Therefore not well suited for interactive applications

- E.g. hard to implement incremental updates

Each filter has to parse/unparse the data (bad for performance)

- Adds complexity to each component

C&C pattern: Client-server pattern

Context: Large numbers of distributed clients wish to access shared resources and services.

Problem:

- How to manage a set of shared resources and services?
  - Multiple physical servers
- How to improve modifiability and reuse, scalability and availability?

C&C pattern: Client-server pattern

Solution:

- Clients request services of servers.
- There may be one central server or multiple distributed ones.

The principal connector type for the client-server pattern is a **data connector**

- driven by a request/reply protocol used for invoking services.

C&C pattern: Client-server pattern

Basic concept:

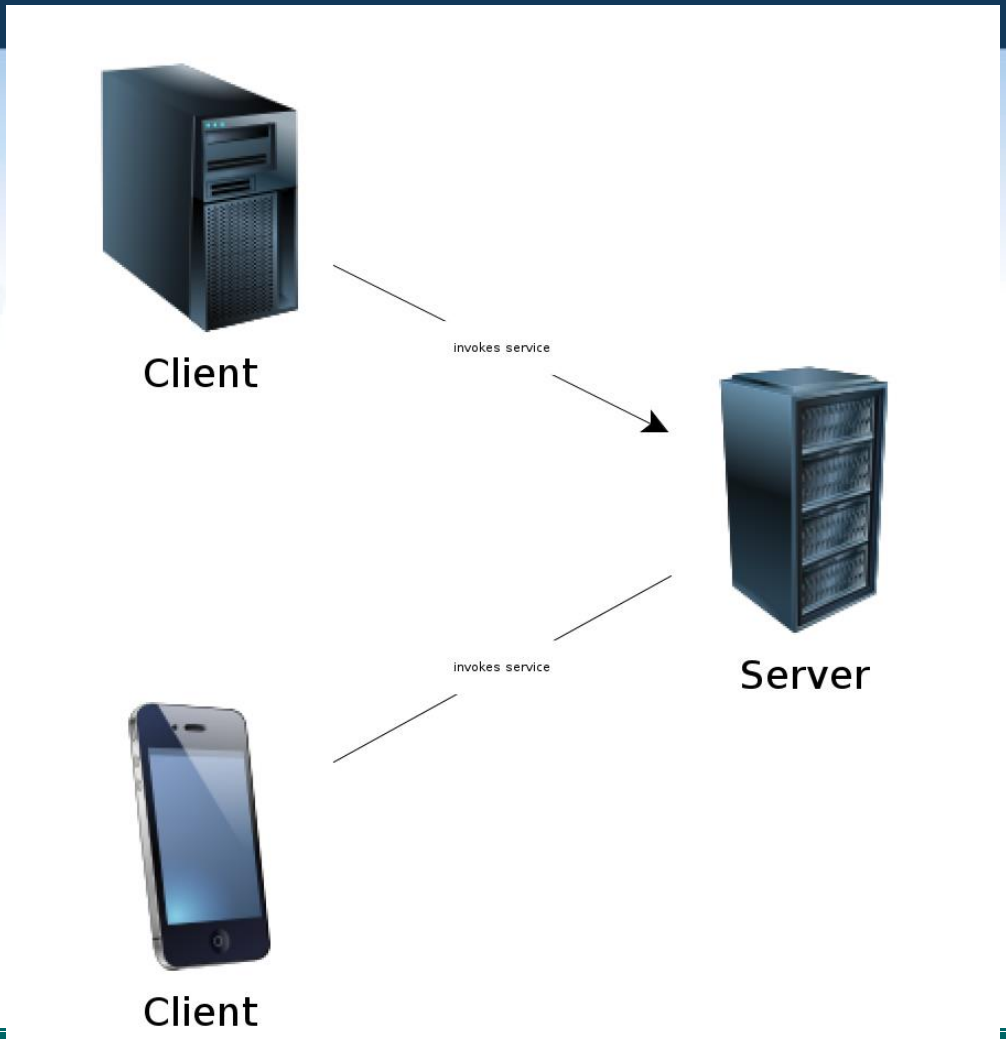
- The **client** uses a service
- The **server** provides a service
- The service can be any resource
  - E.g. data, file, CPU, display device

Typically connected via a network

Clients are *independent* from each other

Clients know the identity of the server but the server does not knows the identities of clients

C&C pattern: Client-server pattern



C&C pattern: Client-server pattern

Separation of concerns (SoC)

Functionality is clearly split into separate components.

- Also motivation for the layered architecture style, where each layer is responsible for its own abstraction

Supports independent evolvability

- if the communication between client and server is well designed.

C&C pattern: Client-server pattern

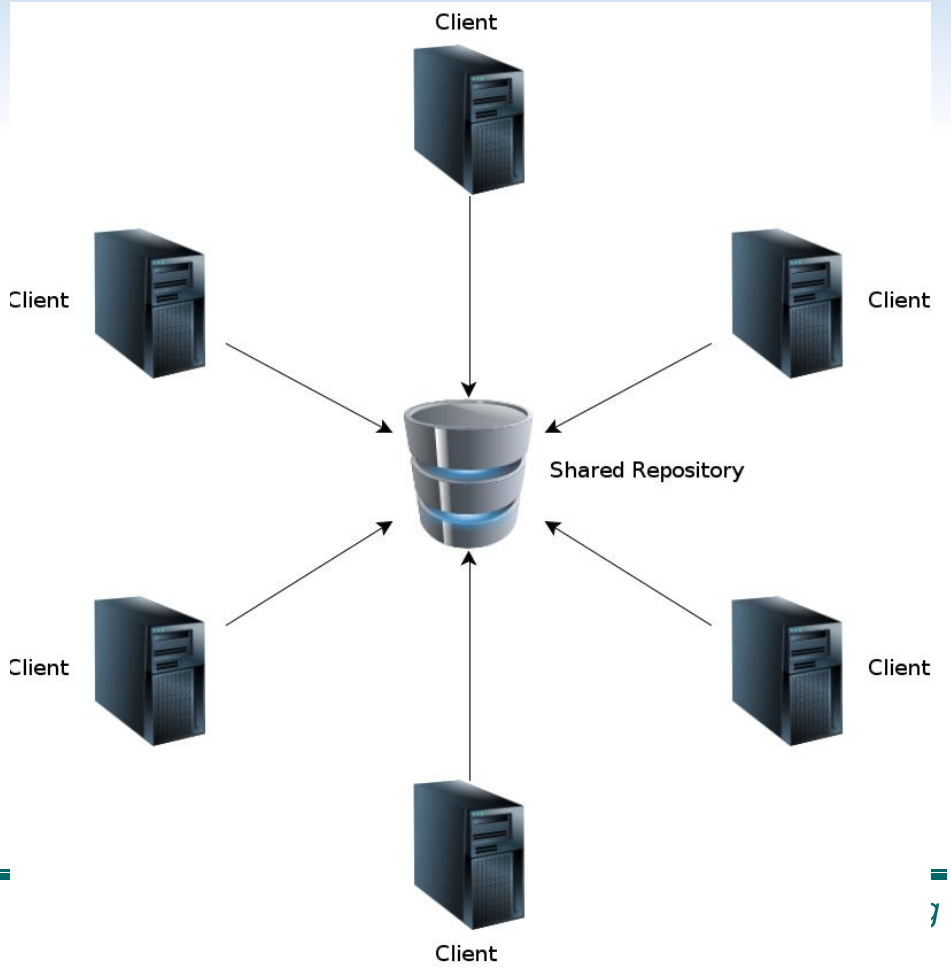
Client-server pattern is used by other architectural styles

It can be used to realize a *shared repository*

- E.g. for the data-centric repository pattern
- E.g. for filters which operate on a single shared data structure

C&C pattern: Client-server pattern

Client-server - Shared Repository:



C&C pattern: Client-server pattern

Two basic types of topology of the server

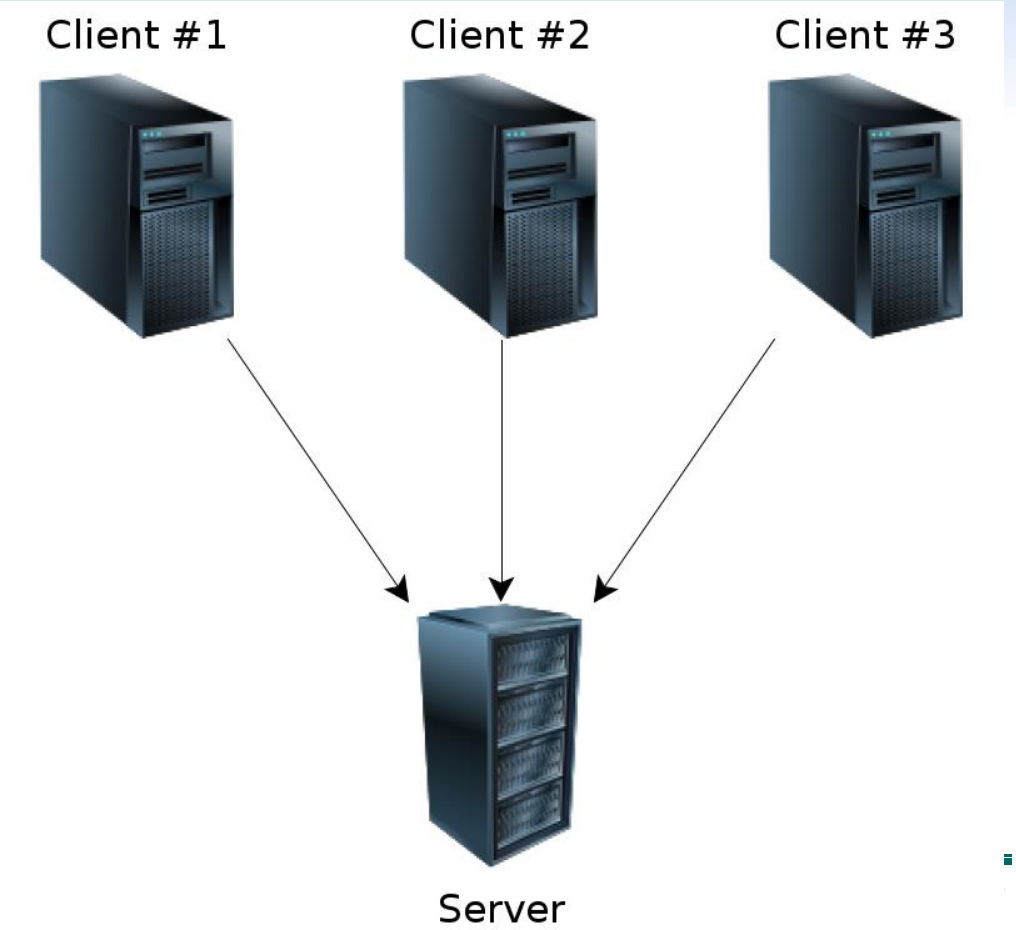
- Single, **centralized** server or
- Multiple, **distributed** servers

Centralized servers are easier to administer (install, deploy updates, maintain, monitor, ...).

Distributed servers scale better, but could introduce complexity (e.g. require two-phase commits).

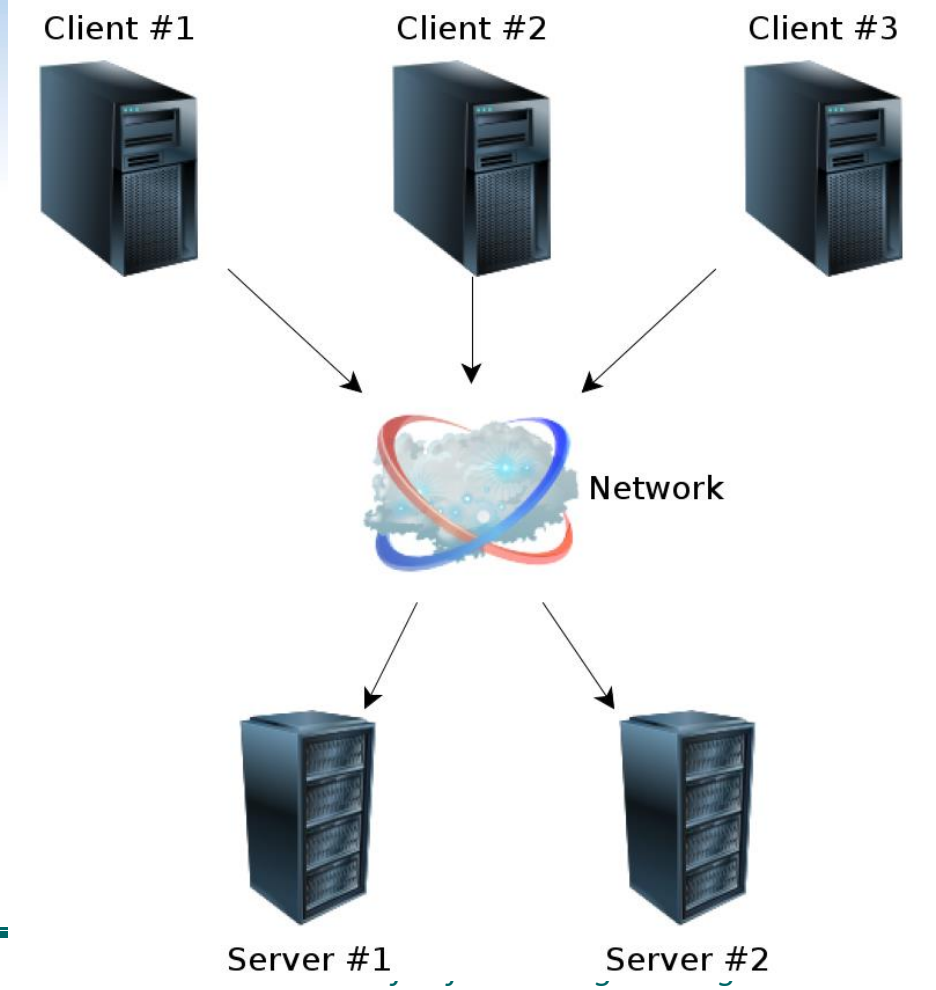
C&C pattern: Client-server pattern

Client-Server - Centralized



C&C pattern: Client-server pattern

Client-Server – Distributed:



C&C pattern: Client-server pattern

The server is no longer in the organizations network, but somewhere in the Internet.

- Example: cloud services by Salesforce, Google, Microsoft

Scalability, security, reliability is expected to be handled by a specialized team.

Loss of control, legal issues (data is exported to another country)

Advantages

Conceptually simple

Clear separation of responsibilities, eases evolvability, help testability

Good scalability

Good for security, as data can be held at the server with restricted access

Disadvantages

Risk of bad usability/performance

Need to develop/agree on a protocol between client and server

Integrability into existing systems might not be possible (e.g. if the communication is not possible, or not allowed)

C&C pattern: Peer-to-peer pattern

Context: Distributed *equally important* entities cooperate and collaborate to provide a service to distributed users.

Problem: How can a set of “equal” distributed entities be connected to each to provide services with high availability and scalability?

C&C pattern: Peer-to-peer pattern

Solution: In the peer-to-peer (P2P) pattern, components directly interact as peers.

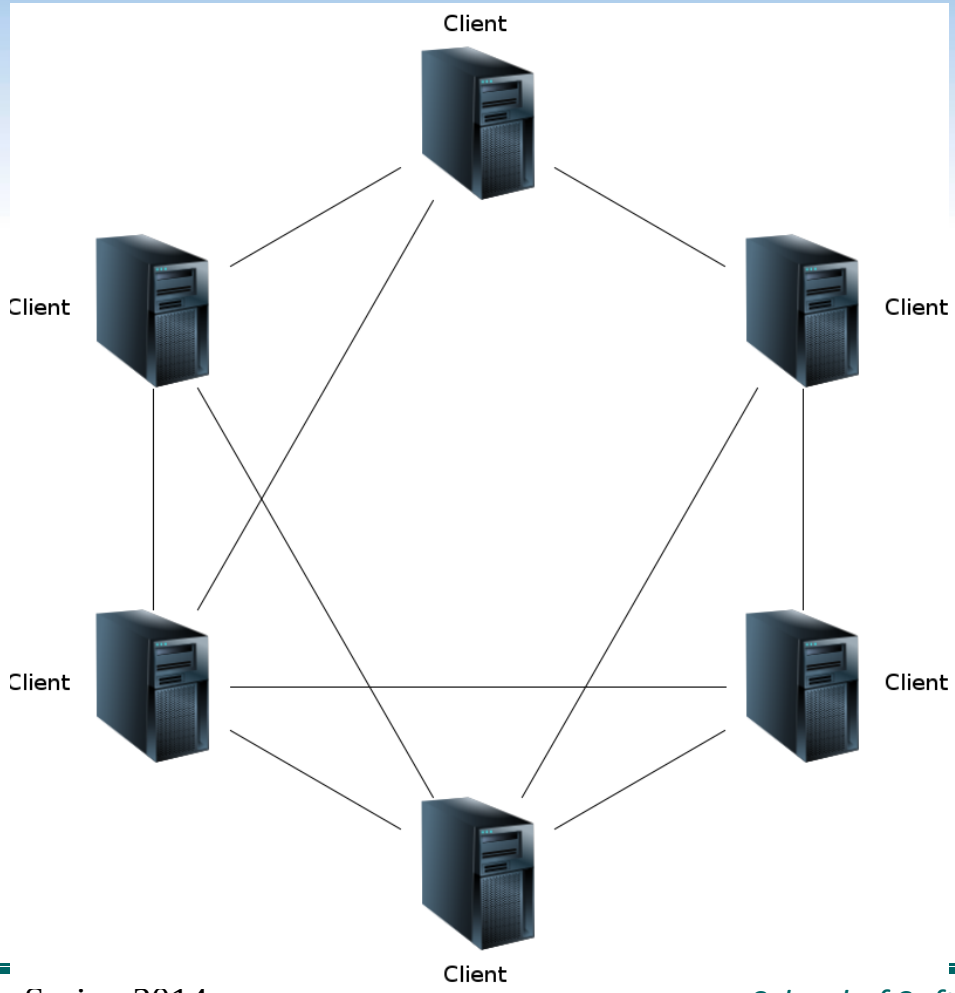
- All peers are “equal”.

Peer-to-peer communication is typically a *request/reply* interaction.

- It’s a symmetric relationship

Separation between client and server is removed.

C&C pattern: Peer-to-peer pattern



C&C pattern: Peer-to-peer pattern

Each peer provides services and consumes services and uses the same protocol.

Number of peers is dynamic.

Once a peer is initialized, it needs to be come part of the network.

Each peer has to know how to access other peers (discover, search, join).

A bootstrapping mechanism is needed:

- For example via a broadcast message
- For example a public list of network addresses

C&C pattern: Peer-to-peer pattern

Centralized P2P:

- Some aspects are centralized.
- For example, a central component keeps track of the available peers.

Hybrid P2P:

- Not all peers are equal, some have additional responsibilities.
- They are called supernodes.
- Example: Skype uses a peer-to-peer protocol, but also uses supernodes and a central login servers.

Advantages

Good for scalability

Improve system's performance

Good for reliability, as data can be replicated over peer

No single point of failure

Disadvantages

Quality of service is not deterministic, cannot be guaranteed

Very complex, hard to maintain and test

- Security, data consistency, availability, backup, recovery…

C&C pattern: Shared-data pattern

Context: Various computational components need to share and manipulate large amounts of data.

- Data does not belong to any one of those components.

Problem: How can systems store and manipulate persistent data that is accessed by multiple independent components?

C&C pattern: Shared-data pattern

Solution: In the shared-data pattern, interaction is dominated by the exchange of data between

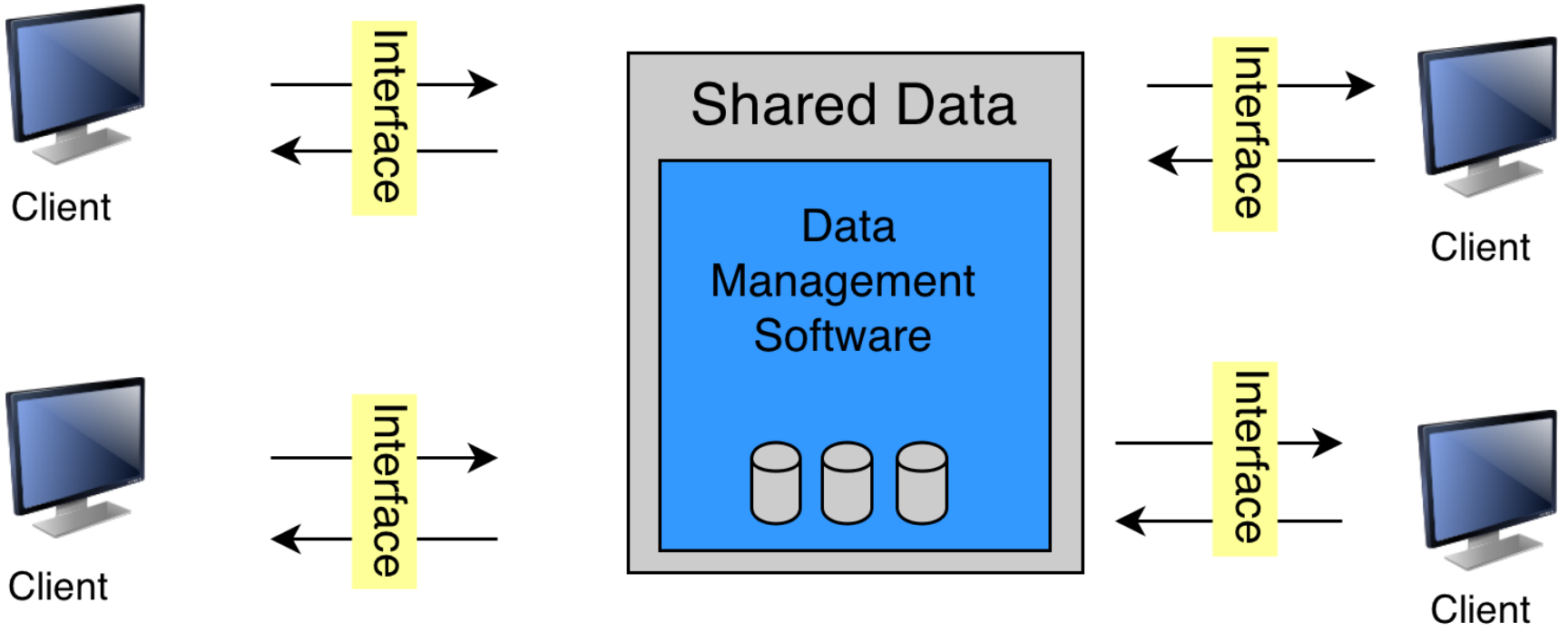
- multiple *data accessors* and
- at least one *shared-data store*.

Exchange may be initiated by the accessors or the data store.

The connector type is *data reading and writing*.

In a pure shared-data system, data accessors interact only through shared-data store(s).

C&C pattern: Shared-data pattern



Advantages

Ensures data integrity

Reliable, secure, testability guaranteed

Clients independent from the system: performance and usability on the client side is typically good

Disadvantages

Problems with performance, reliability (single point of failure)

- Solutions: shared repositories, replication but this increases complexity

Data producers and consumers are tightly coupled.

Some case studies

1. KWIC

In his paper (1972) David Parnas proposed the following problem

> The KWIC (Key Word in Context) index system accepts an ordered set of lines; each line is an ordered set of words, and each word is an ordered set of characters. Any line may be “circularly shifted” by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.

Circular shifts

Original title

- Gone with the Wind

Circular shifts (key words underlined)

- Gone with the Wind
- with the Wind Gone
- the Wind Gone with
- Wind Gone with the

Stop word removal

- Gone with the Wind
- Wind Gone with the

Example with multiple titles

Gone with the Wind

War and Remembrances

The Winds of War

```
                Gone                with the Wind
and             Remembrances                    War
Winds of        War                             The
                War             and Remembrances
with the        Wind                        Gone
     The        Winds           of War
```

Architectural solutions for KWIC

KWIC with

- (main program/subroutine with) shared data style
- pipe-and-filter
- abstract data types (Object-Oriented)
- implicit invocation (event-based)

KWIC with shared data style

Historical example: Shared data style is the way that systems were built for performance reasons until the early 1970s.

Shared data style is not normally used today due to concerns with other qualities. (Shared data style does not easily scale up to large architectures.)

KWIC with shared data style

Problem decomposed according to 4 basic functions

- Input, shift, alphabetize, output

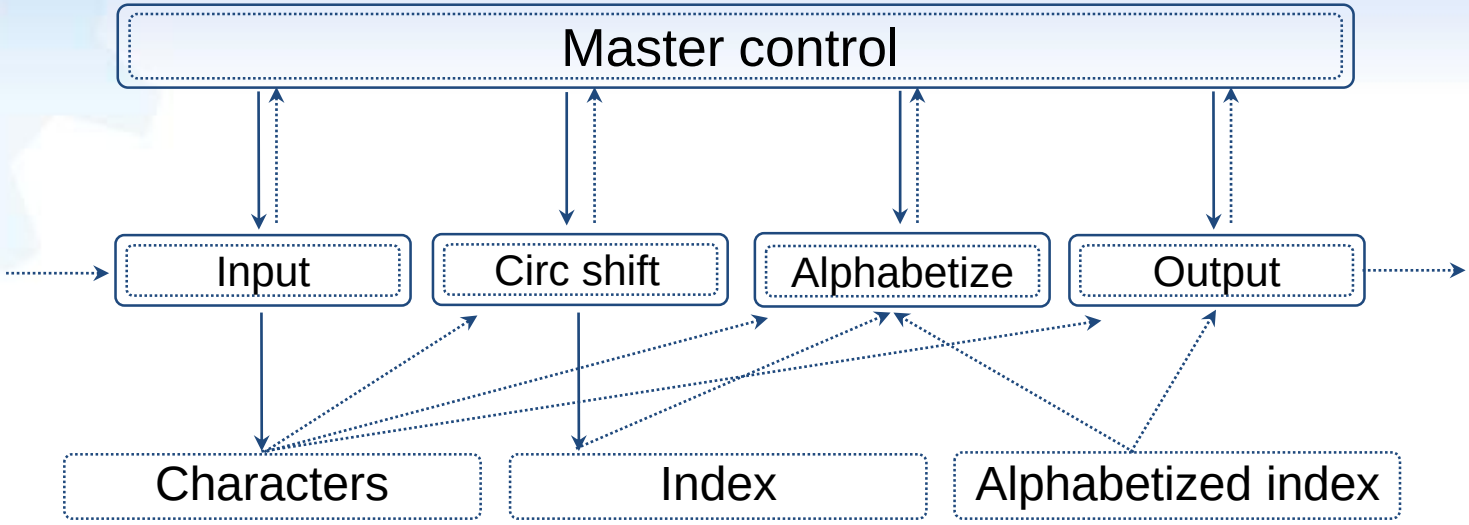
Components coordinated by main program that sequences through them.

Data in shared storage

Communication: unconstrained read-write protocol

- Coordinator ensures sequential access to data

KWIC with shared data style



KWIC with shared data style

Advantages

- Data can be represented efficiently
- Intuitive appeal

Disadvantages

- Modifiability
  - Change in data format affects all components
  - Enhancements to system function
- Reuse not easy to do

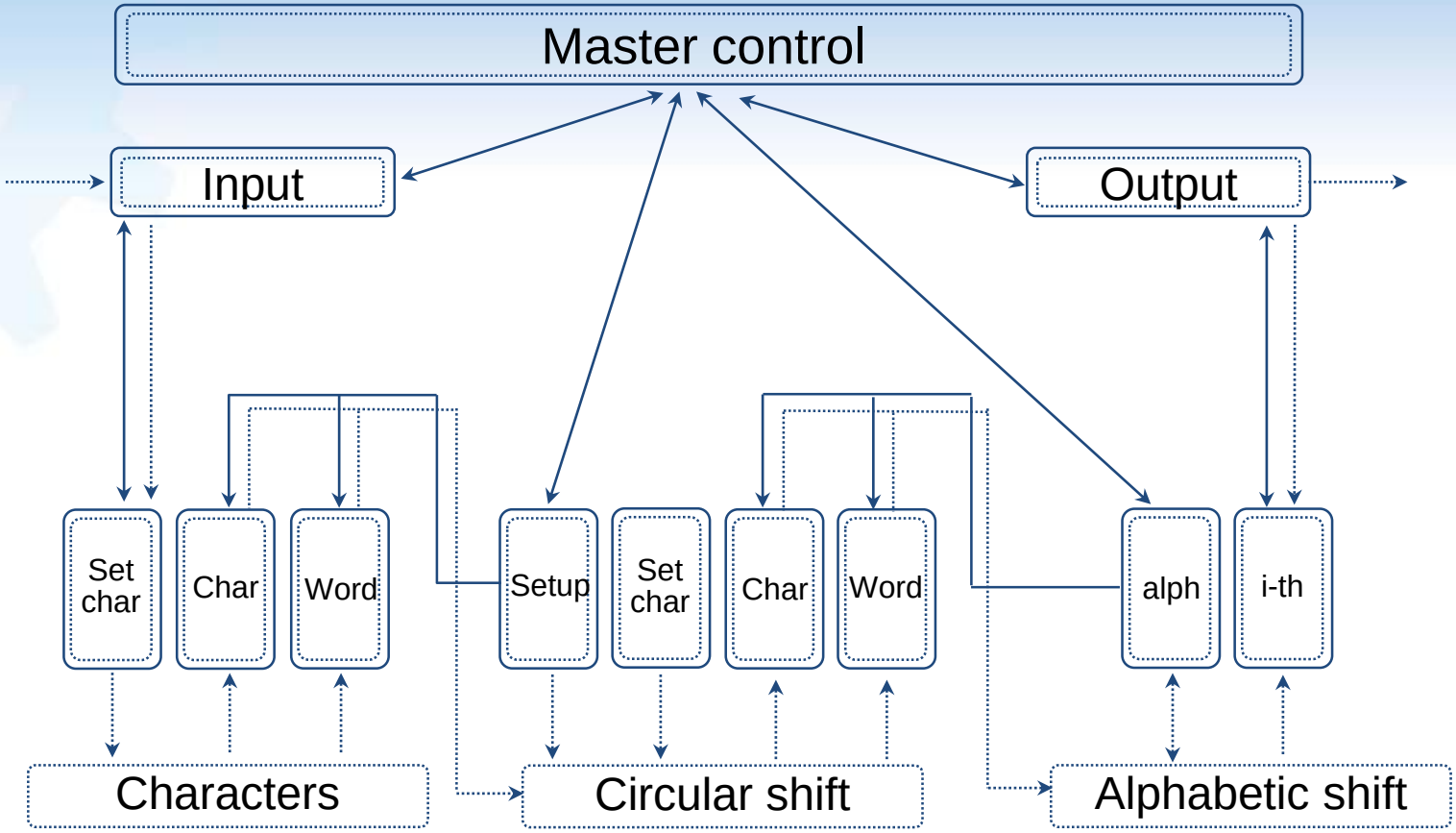
KWIC with abstract data types

Similar set of five modules, with interfaces

Data is not shared by computational components

- Accessed via interfaces

KWIC with abstract data types



KWIC with abstract data types

Advantages

- Logical decomposition into processing modules similar to shared data
- Algorithms/data can be changed in individual modules w/o affecting others
- Better reuse (module has fewer assumptions about other modules)

Disadvantages

- Enhancing the function
  - Modify existing modules -> bad for simplicity, integrity
  - Add new modules -> performance penalties

KWIC with implicit invocation

Shared data as the integration mechanism

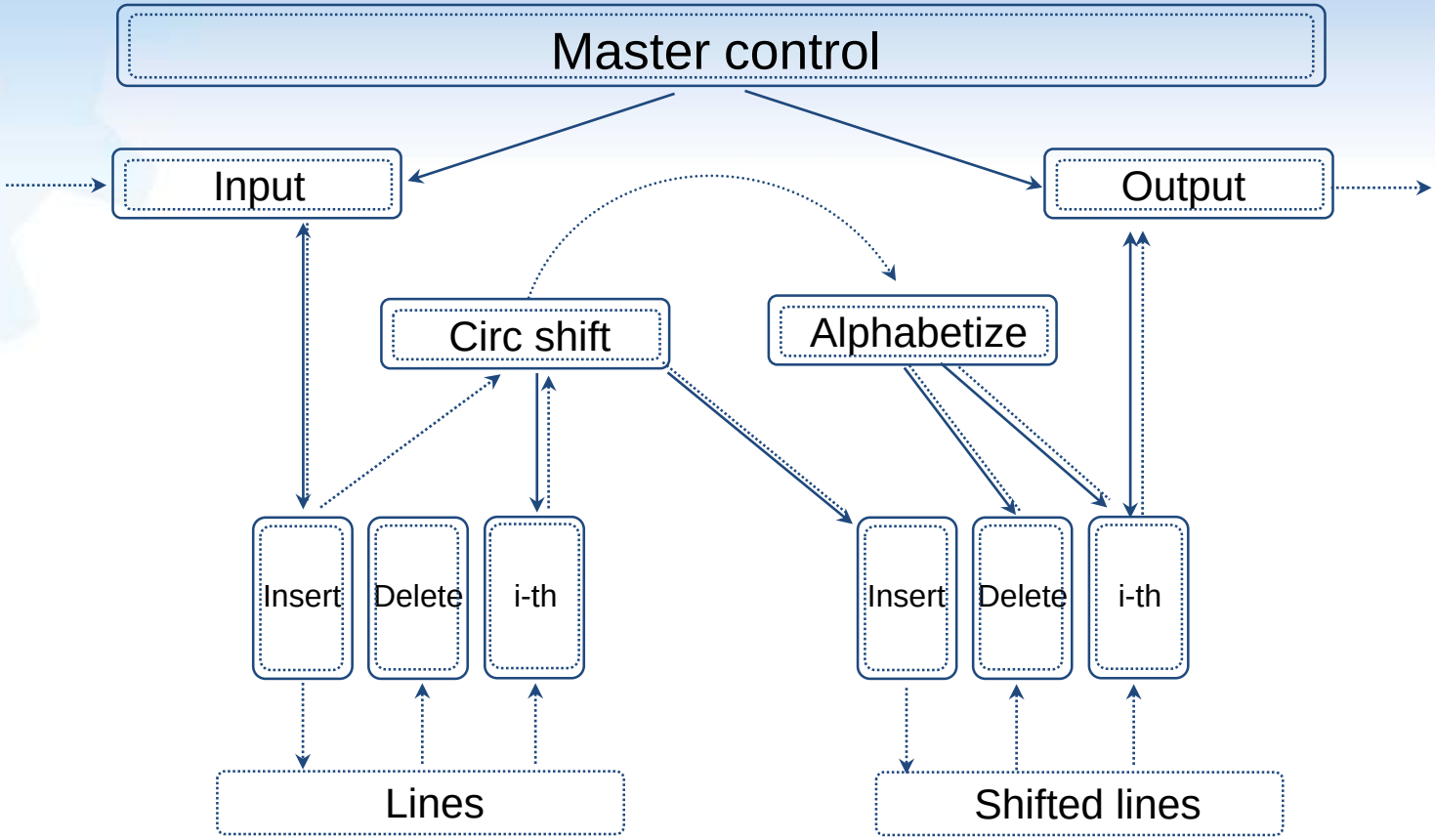
More abstract data interfaces

- Data accessed as a list/set

Computations invoked implicitly when data is modified

- Line added -> event to shift module
- Circular shifts produced in another shared data store -> event to alphabetizer, invoked

KWIC with implicit invocation



KWIC with implicit invocation

Advantages

- Functional enhancements easily
- Data changes possible
- Reuse

Disadvantages

- Difficult to control processing order of implicitly invoked modules
- Data representation uses more space

KWIC with pipe-and-filter

Two filters

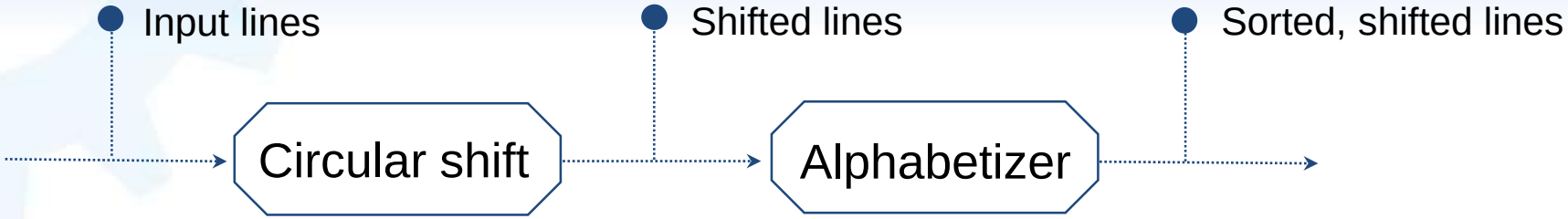
- Circular shift, alphabetizer
- Process data and send it to the next

Distributed control

Data sharing

- Only the one transmitted on pipes

KWIC with pipe-and-filter



KWIC with pipe-and-filter

Advantages

- Maintains intuitive flow of processing
- Reuse supported
- New functions easily added
- Amenable to modifications

Disadvantages

- Impossible to modify design to get interactive system
- Data is copied between filters –> space used inefficiently
- Slower performance speed -> parsing input

Rough comparison of KWIC architectures

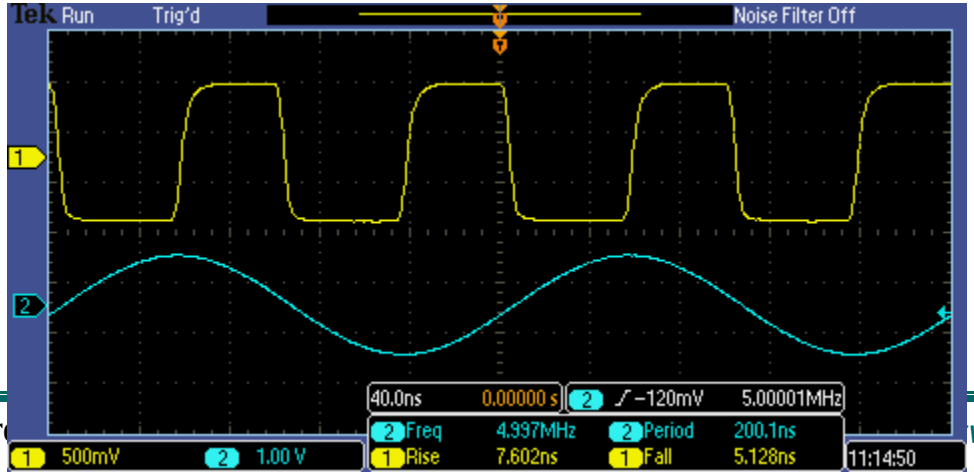
	Shared data	Abstract data type	Implicit invocation	Piper and filter
Change in algorithm	-	-	+	+
Change in data representation	-	+	-	-
Change in function	-	-	+	+
performance	+	+	-	-
Reuse	-	+	-	+

2. Instrumentation software

Develop a reusable system architecture for oscilloscopes

Rely on digital technology

Have quite complex software



Problems to solve

Reuse across different oscilloscope products

- Tailor a general-purpose instrument to a specific set of users

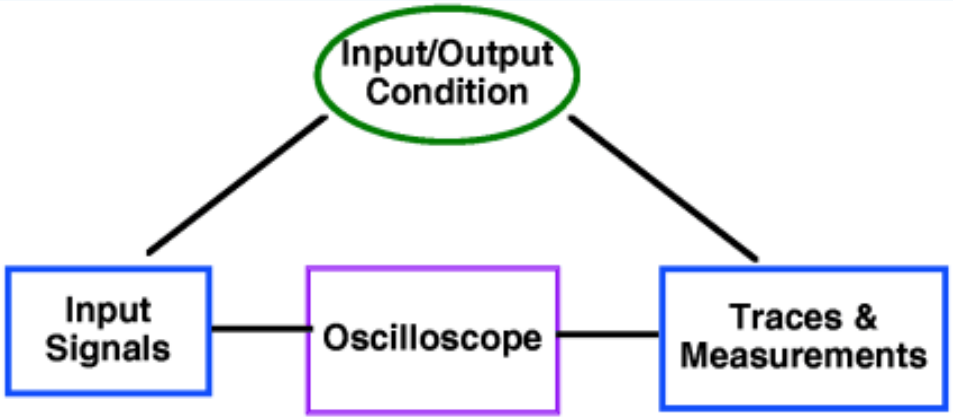
Performance important

- Rapid configuration of software within the instrument

⇒Domain-specific software architecture

We outline the stages in the architectural development

The problem frame



JSP Problem Frame?

Oscilloscope: OO Approach

Clarified the data types used for oscilloscopes

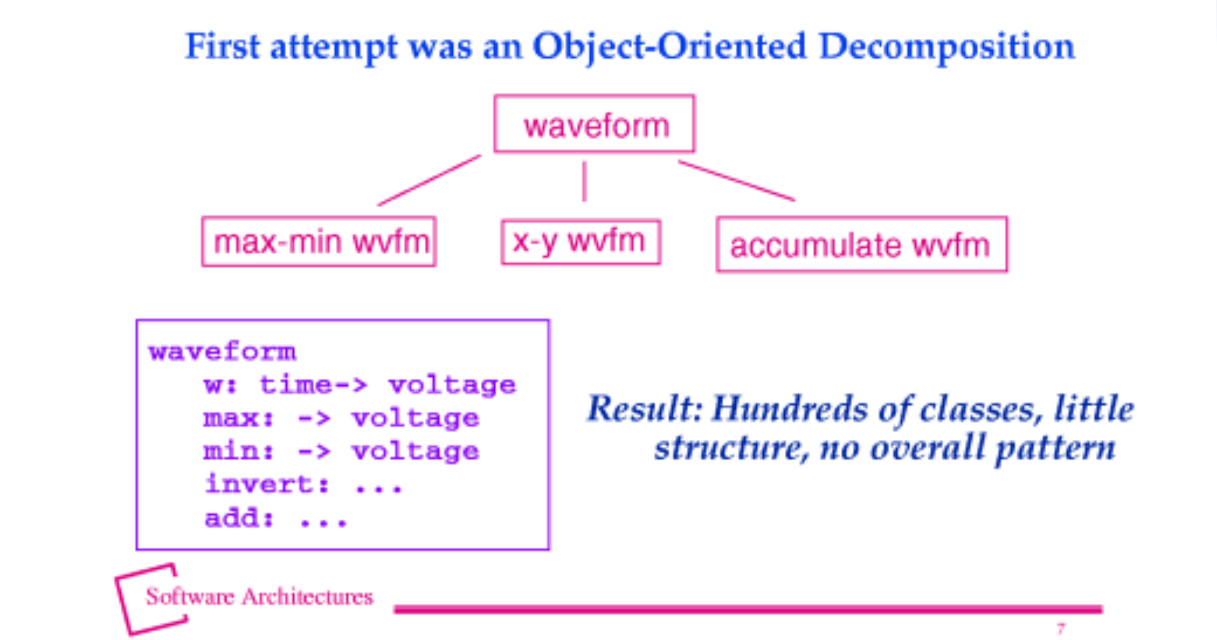
- Waveforms, signals, measurement, trigger modes, …

No overall model to explain how the types fit together

Confusion about partitioning of functionality

- Should measurements be associated with types of data being measured or represented externally?
- Which objects should the user interface interact with?

Oscilloscope: OO Approach



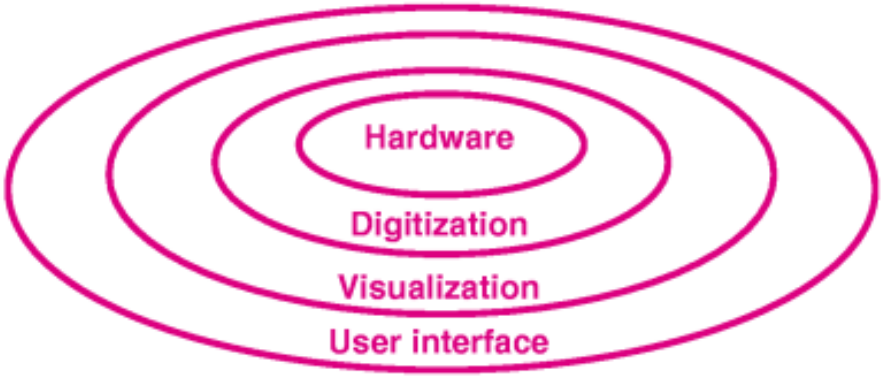
Oscilloscope: Layered approach

Well-defined grouping of functions

Wrong model for the application domain

- Layer boundaries conflicted with the needs of the interaction among functions
  - The model suggest user interaction only via visual representation, but in practice this interaction affects all layers (setting parameters, etc)

Oscilloscope: Layered approach



Result: Boundaries of abstraction not realistic

Software Architectures

Oscilloscope: Pipe-and-filter approach

Signal transformers used to condition external signals

Acquisition transformers derive digitized waveforms from these signals

Display transformers convert these waveforms into visual data

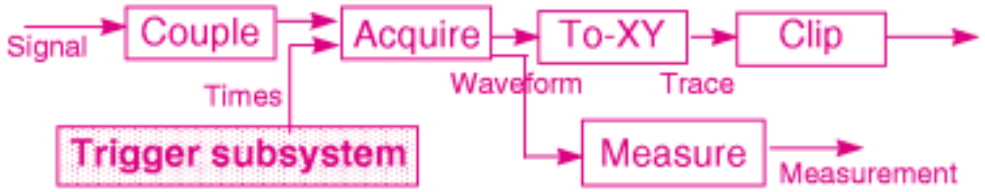
Oscilloscope functions were viewed as incremental transformers of data

Corresponds well with the engineers' view of signal processing as a dataflow problem

Main problem:

- How should the user interact?

Oscilloscope: Pipe-and-filter approach



Result: A better model,
but not clear how to model user input.

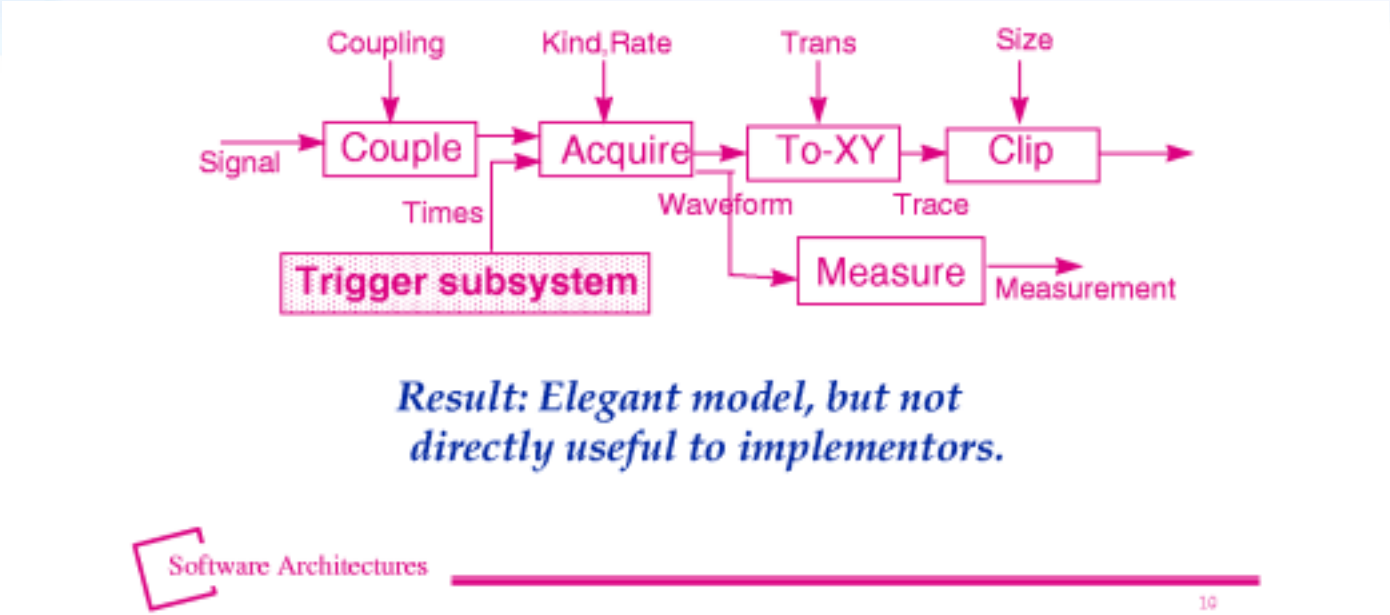
Oscilloscope: Extended pipe-filter approach

Each filter was associated with a control interface

- Provides a collection of settings to be modified dynamically by the user
- Explains how the user can make incremental adjustments to SW
- Decouples signal-processing from user interface

Signal-processing SW and HW can be changed without affecting the user interface as long as the control interface remains the same

Oscilloscope: Extended pipe-filter approach



Oscilloscope: Extended pipe-filter approach

Further specialization

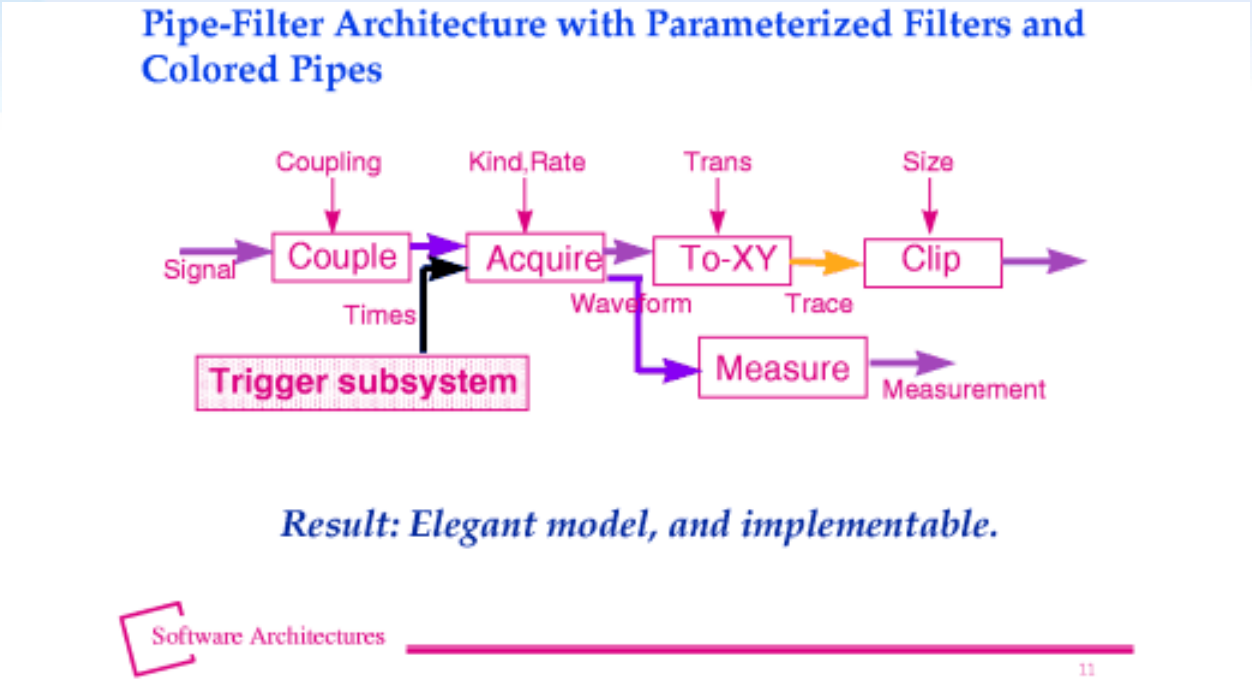
- Pipe-and-filter lead to poor performance
  - Problems with internal storage and data exchange between filters
    - Waveforms have large internal storage => not practical for filters to copy waveforms every time they process them
  - Filters may run at radically different speeds
    - Not good to slow faster filter just to keep the pace with slower ones

Oscilloscope: Extended pipe-filter approach

Further specialization

- Solution: several types of pipes (distinct colors)
  - Some allowed data processing without copying
  - Slow filters allowed to ignore incoming data when already processing other data
  - => the pipe/filter computations more tailorable

Oscilloscope: Extended pipe-filter approach



Instrumentation software summary

Case study shows

- Some issues for developing architectures for industrial SW
- Different styles => different effects on solution

Software must be typically adapted from pure forms to specialized styles (domain specific)

Here the result depended on properties of pipe-and-filter architecture adapted to satisfy the needs of the product family

Rules of thumb for choosing styles

The goal of style catalogs is to develop a design handbook: “If your problem looks like x, use style y.”

The practice is not that advanced yet. The best that we can do is offer rules of thumb.

The End