Принципы проектирования и дизайна ПО

Лекция №13

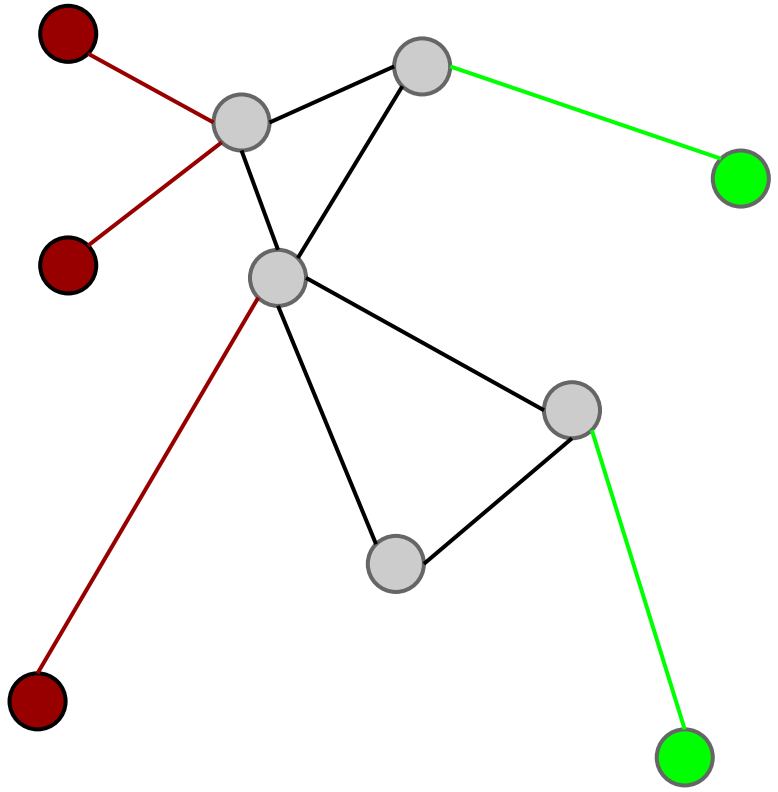
Агошков Илья 2016

What is an architectural style?

A *component* is a basic architectural computational component - clients, servers, filters, layers, a database, etc.

A *connector* provides the interaction between components.

A style is a *vocabulary* of possible components and connector types and constraints on how they can be applied.



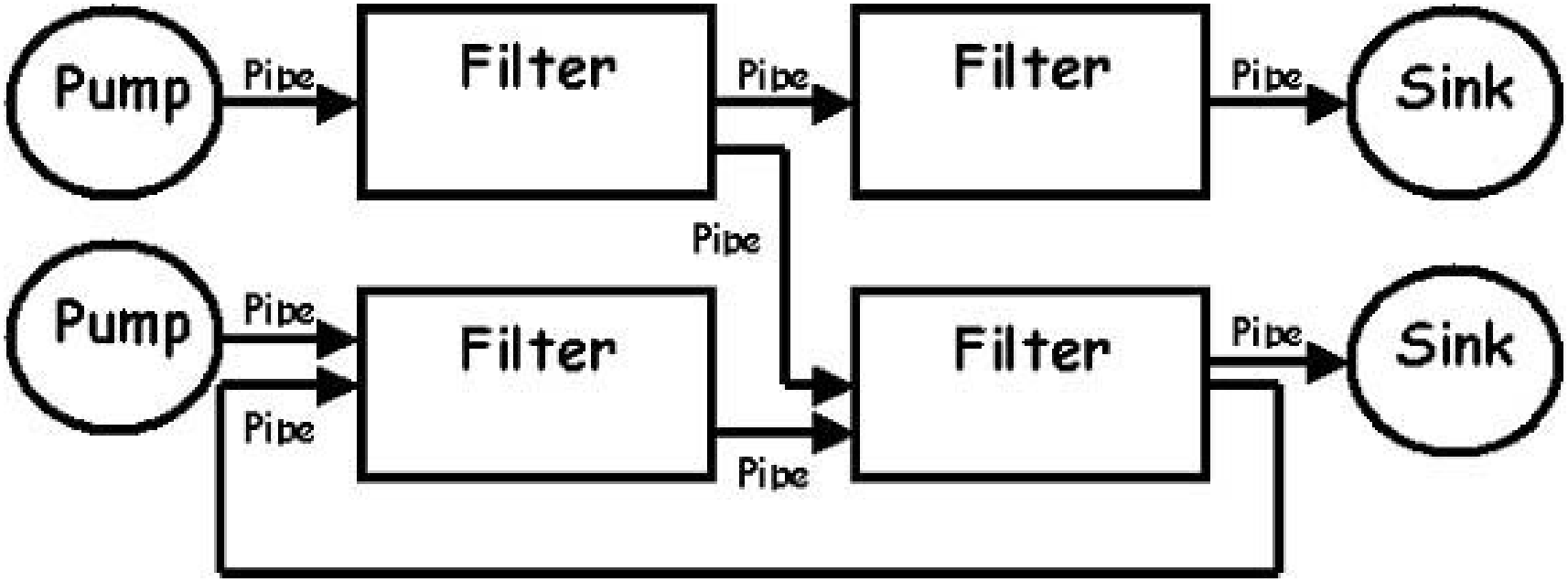
Styles

Data Flow	Dominated by movement of data through the system, with no “upstream” content control by recipient	Pipes and filters Batch sequential
Data Centered	Dominated by a complex central data store manipulated by independent components	Repository Blackboards
Data Sharing	Dominated by sharing of data amongst components	Hypertext systems Compound document
Call & Return systems	Dominated by order of computation, usually with a single thread of control	Procedural Object oriented Naive client server
Interacting Processes	Dominated by communication between independent, usually concurrent, processes	Event Driven Event Sourcing Broker SOA
Hierarchical Systems	Dominated by reduced coupling with the partition of the system into subsystems with limited interactions	Interpreters Rule-Based systems Layers CQRS Plugins

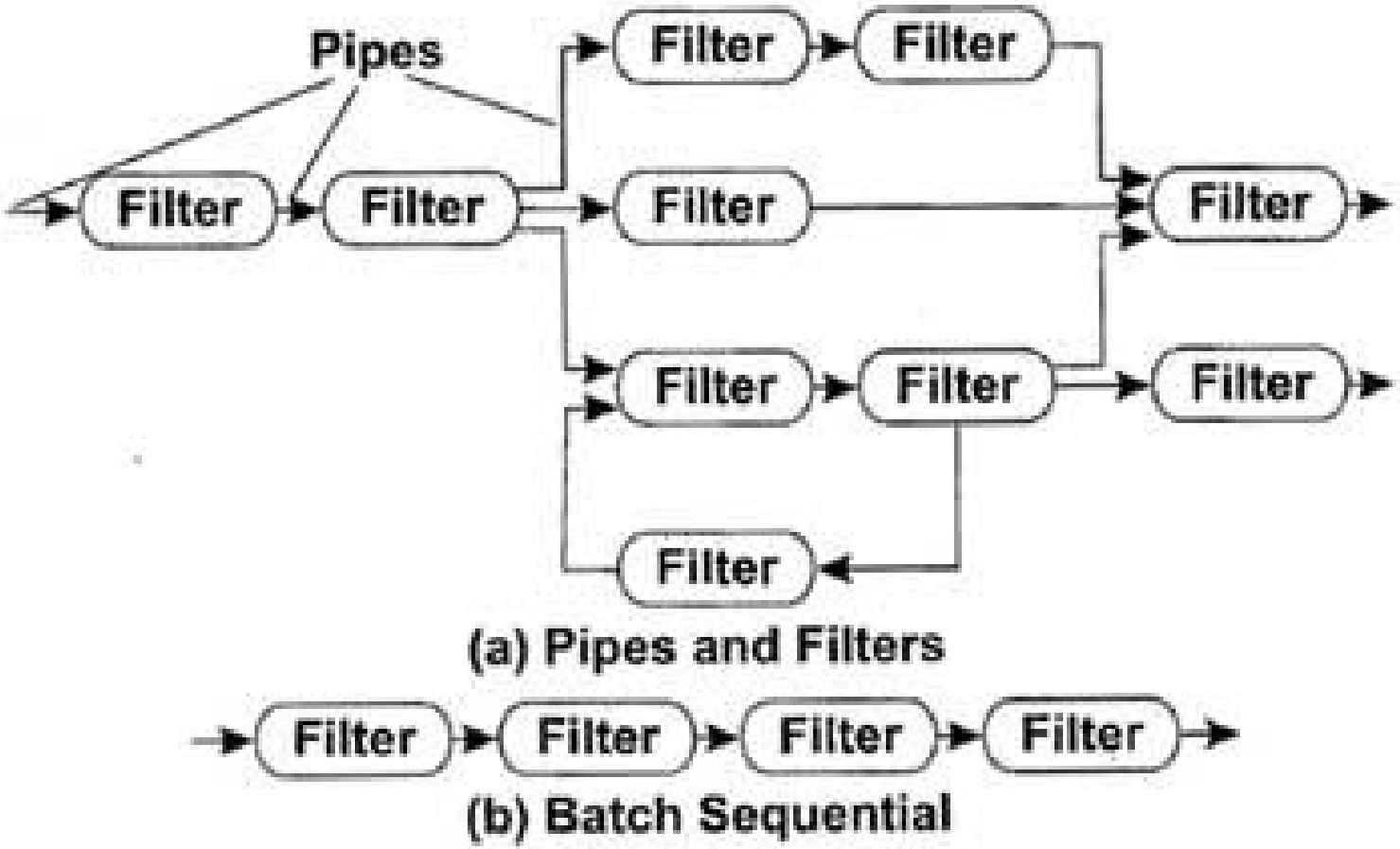
Styles

Data Flow	Dominated by movement of data through the system, with no “upstream” content control by recipient	Pipes and filters Batch sequential
Data Centered	Dominated by a complex central data store manipulated by independent components	Repository Blackboards
Data Sharing	Dominated by sharing of data amongst components	Hypertext systems Compound document
Call & Return systems	Dominated by order of computation, usually with a single thread of control	Procedural Object oriented Naive client server
Interacting Processes	Dominated by communication between independent, usually concurrent, processes	Event Driven Event Sourcing Broker SOA
Hierarchical Systems	Dominated by reduced coupling with the partition of the system into subsystems with limited interactions	Interpreters Rule-Based systems Layers CQRS Plugins

Pipes and filters



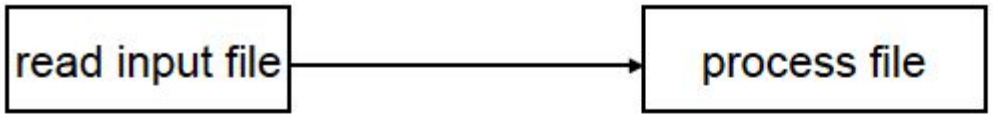
Batch sequential

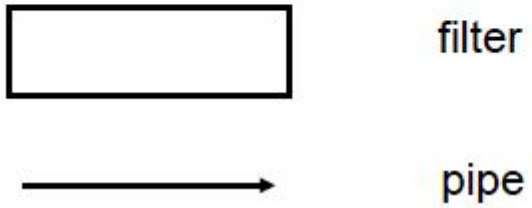


Pipe and Filter Architecture

Main components:

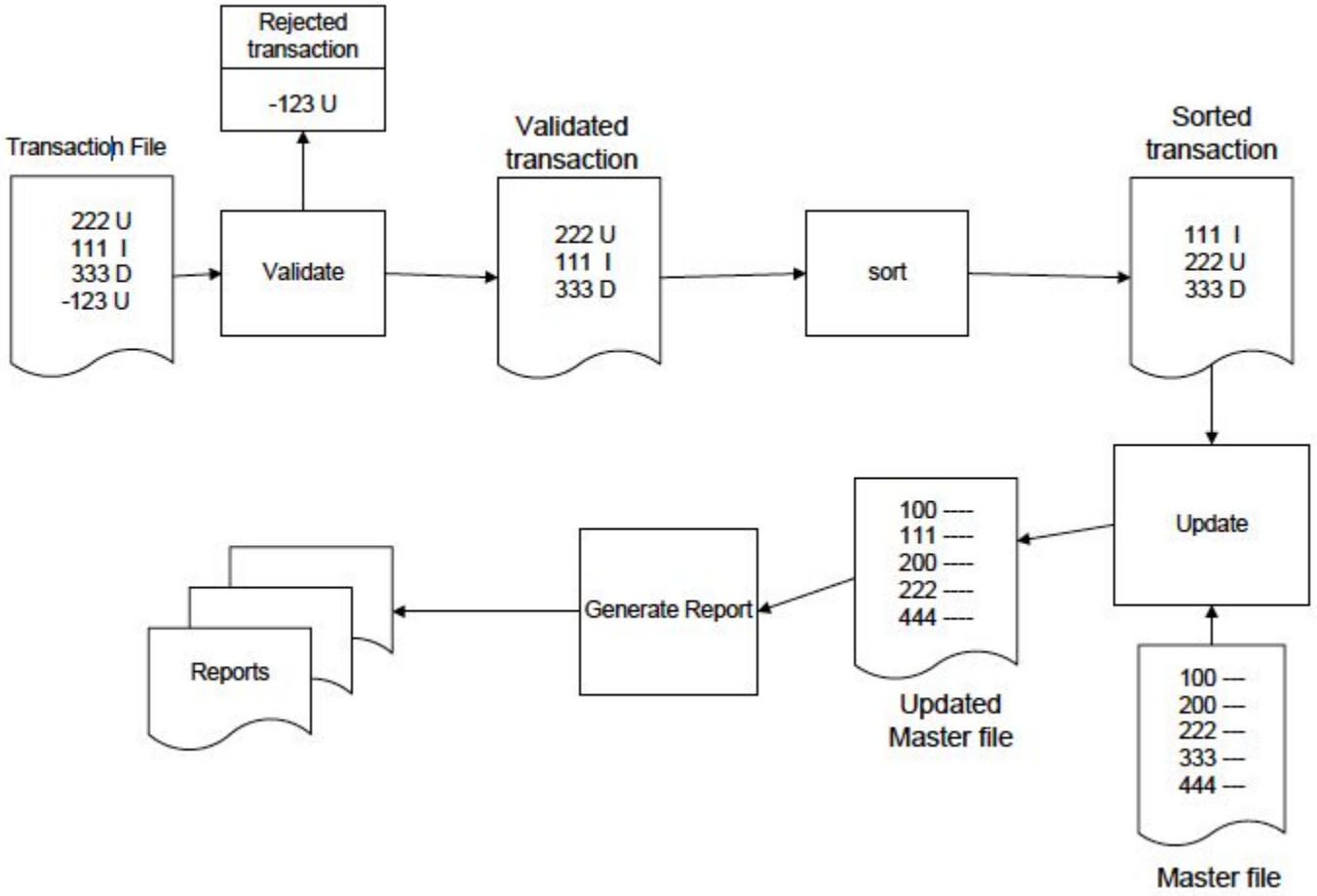
- Filter: process the a stream of input data to some output data
- Pipe: a channel that allows the flow of data





This architecture style focuses on the dynamic (interaction) rather than the structural

Batch sequential data flow



Pipe and Filter: UNIX shell

UNIX shell command line processing of the pipe symbol: “|”

the output from the command to the left of the symbol, |, is to be sent as input to the command to the right of the pipe; this mechanism got rid of specifying a file as std output of a command and then specifying that same file as the std input to another command, including possibly removing this intermediate file afterwards

Example : counting occurrences in a file

I have a mailinglist file called “swarch.txt”

```
cat swarch.txt | grep studio | wc
```

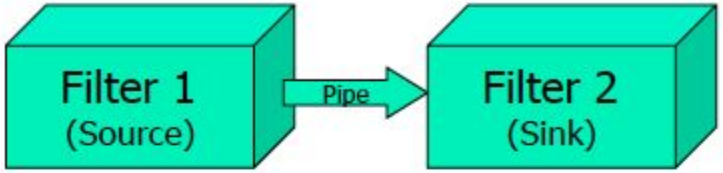
Note the pipe symbol, |

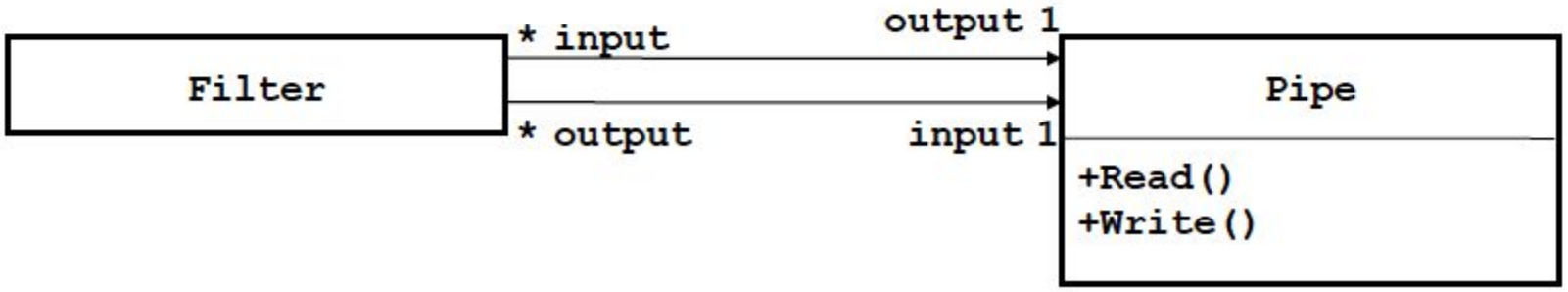
More Modern Version of Pipe-filter

- Consider the MS Office Product
- Specifically --- think about how the component called “copy”works in conjunction with the component called “paste” across office product (e.g. spread sheet to word document)
- The clipboard as a “pipe”

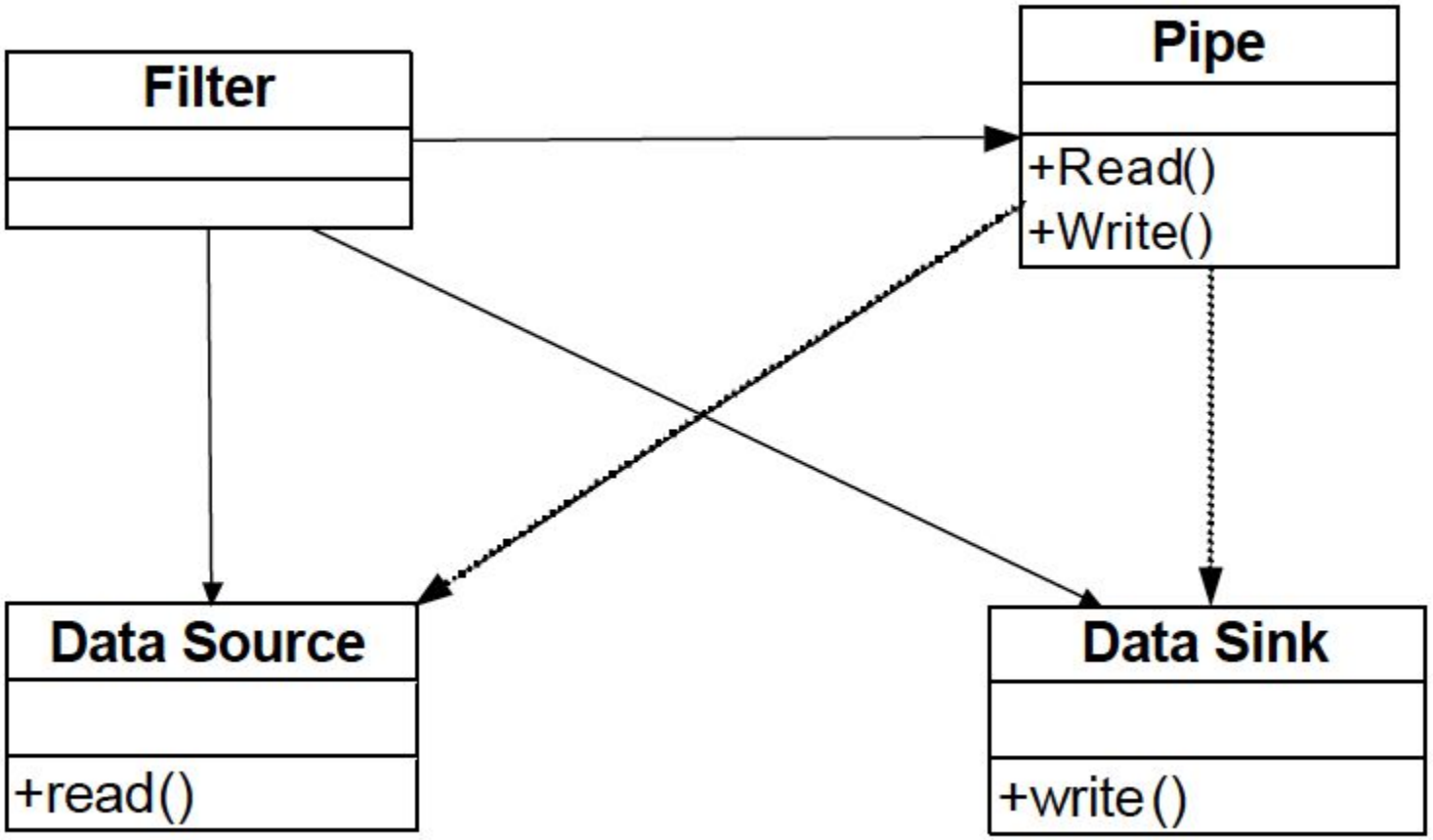
Pipes and Filters

- Pipe: communication channel
- Filter: computing component
  - Filter 1 may only send data to Filter 2
  - Filter 2 may only receive data from Filter 1
  - Filter 1 may not receive data
  - Filter 2 may not send data
  - Pipe is the data transport mechanism

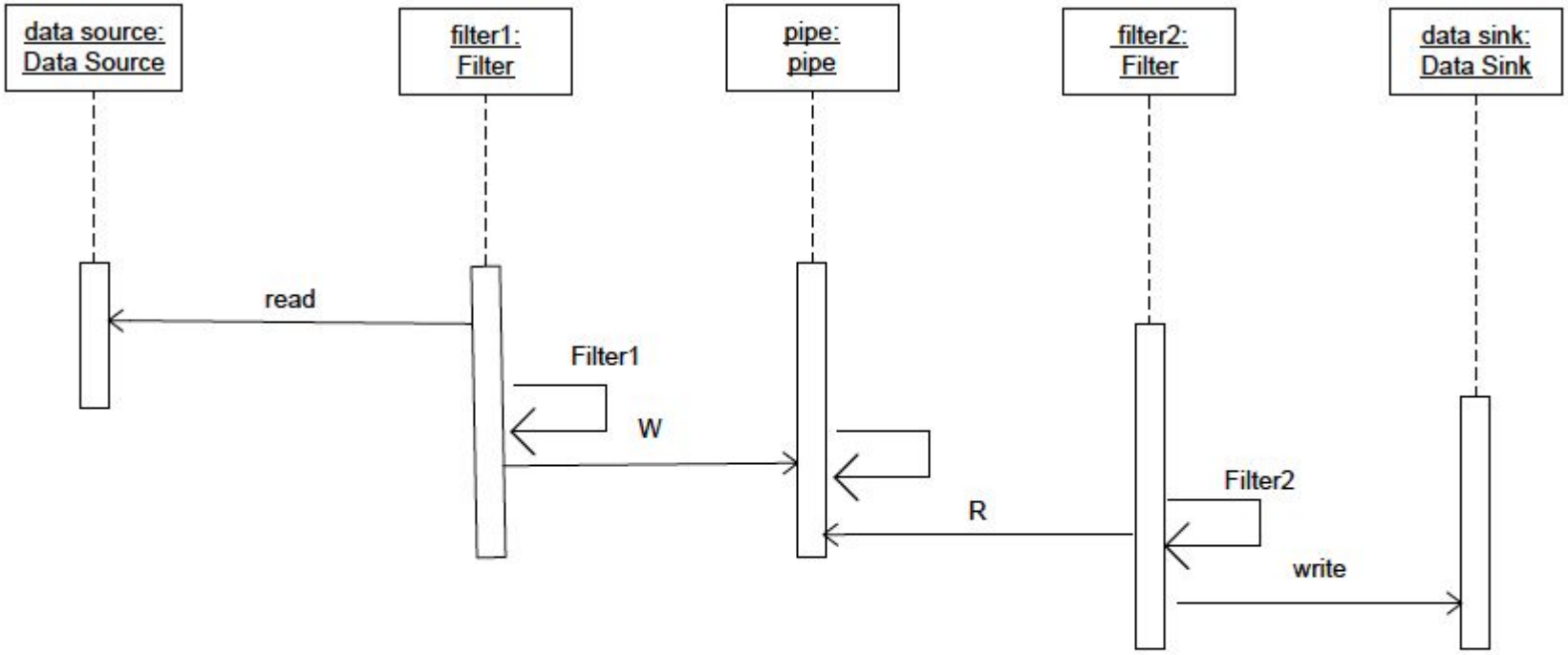


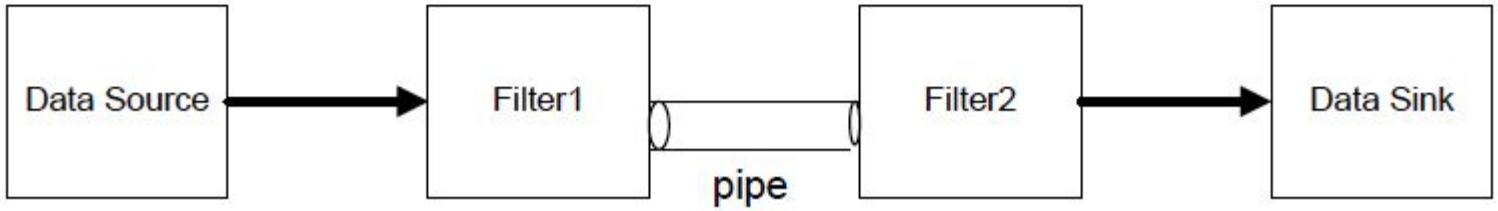


Pipe and Filter Class Diagram



Pipe and Filter Sequence Diagram





Data flow types

There are three ways to make the data flow:

- Push only (Write only)
  - A data source may pushes data in a downstream
  - A filter may push data in a downstream
- Pull only (Read only)
  - A data sink may pull data from an upstream
  - A filter may pull data from an upstream
- Pull/Push (Read/Write)
  - A filter may pull data from an upstream and push transformed data in a downstream

Active or passive filters

An **active filter** pulls in data and push out the transformed data (pull/push); it works with a passive pipe which provides read/write mechanisms for pulling and pushing.

A **passive filter** lets connected pipes to push data in and pull data out. It works with active pipes that pull data out from a filter and push data into the next filter. The filter must provide the read/write mechanisms in this case.

Pipes and Filters: pros and cons

Advantages:

- Filters are self containing processing services that perform a specific function thus the style is cohesive
- Filters communicate (pass data most of the time) through pipes only, thus the style results in low coupling

Disadvantages:

- The architecture is static (no dynamic reconfiguration)
- Filter processes which send streams of data over pipes is a solution that fits well with heavy batch processing, but may not do well with any kind of user-interaction.
- Anything that requires quick and short error processing is still restricted to sending data through the pipes, possibly making it difficult to interactively react to error-events.

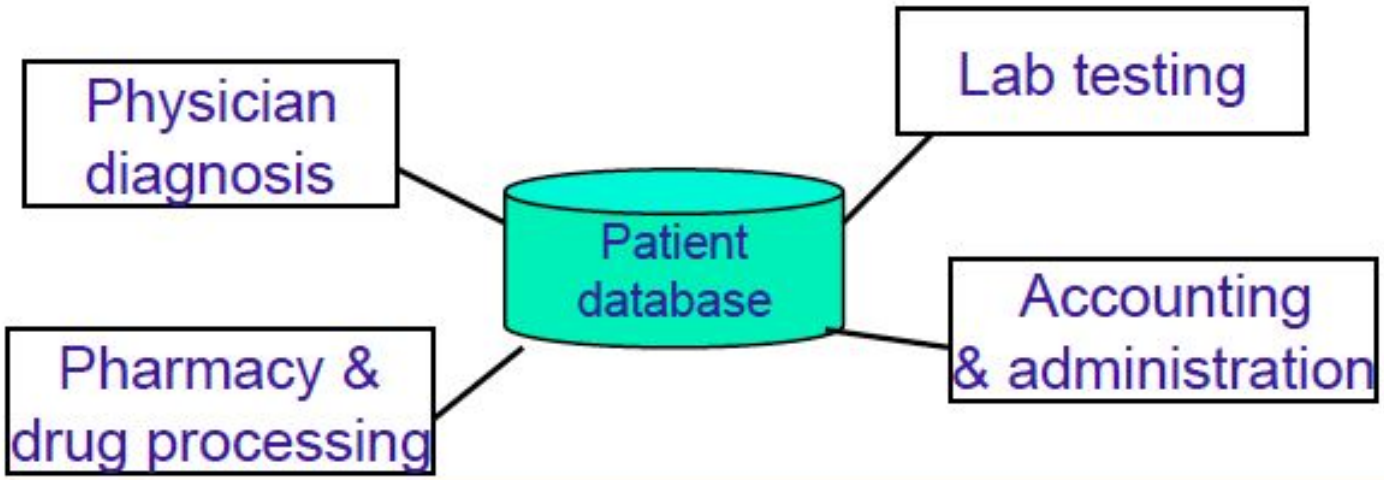
Styles

Data Flow	Dominated by movement of data through the system, with no “upstream” content control by recipient	Pipes and filters Batch sequential
Data Centered	Dominated by a complex central data store manipulated by independent components	Repository Blackboards
Data Sharing	Dominated by sharing of data amongst components	Hypertext systems Compound document
Call & Return systems	Dominated by order of computation, usually with a single thread of control	Procedural Object oriented Naive client server
Interacting Processes	Dominated by communication between independent, usually concurrent, processes	Event Driven Event Sourcing Broker SOA
Hierarchical Systems	Dominated by reduced coupling with the partition of the system into subsystems with limited interactions	Interpreters Rule-Based systems Layers CQRS Plugins

Data centered

- Very common in information systems where data is shared among different functions
- A repository is a shared data-store with two variants:
  - Repository style: the participating parties check the data-store for changes
  - Blackboard (or tuple space) style: the data-store alerts the participating parties whenever there is a data-store change (trigger)

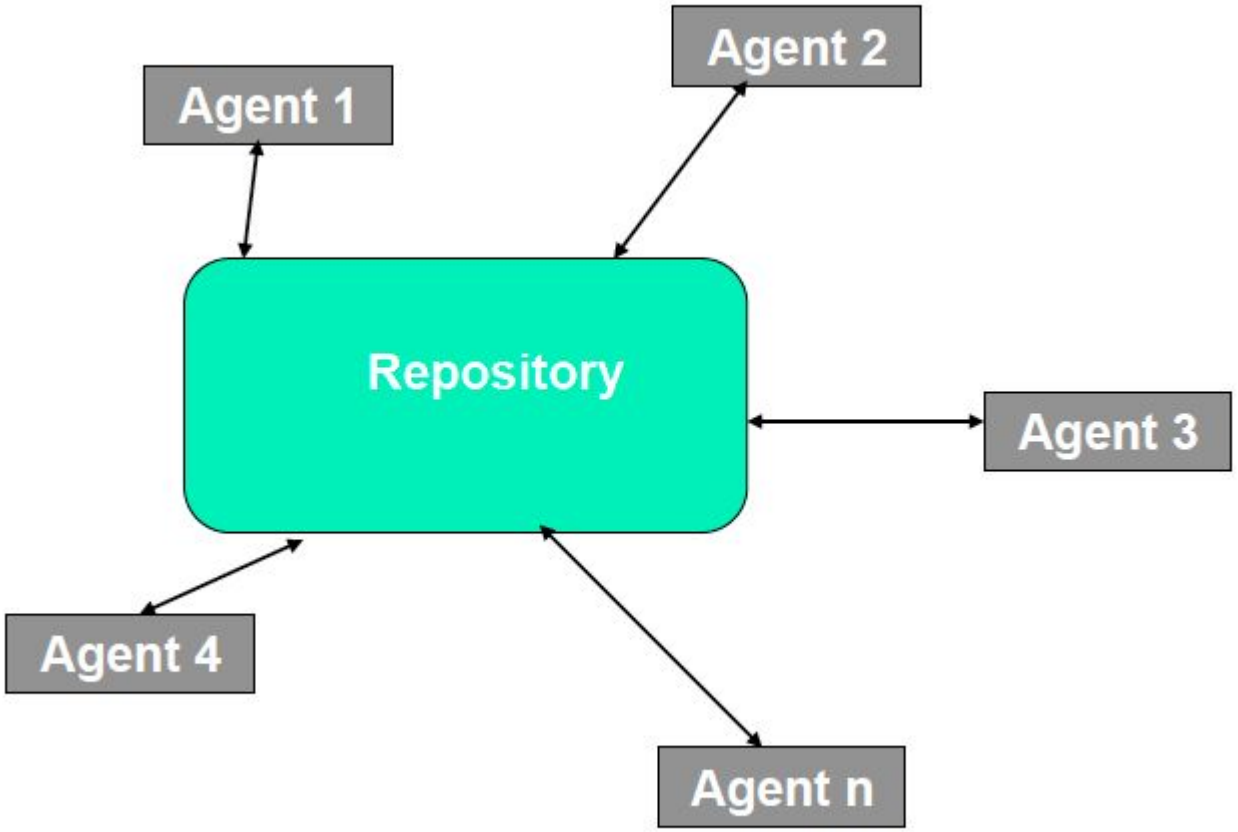
Data centered



Problems that fit this style have the following properties:

1. All the functionalities work off a shared data-store
2. Any change to the data-store may affect all or some of the functions
3. All the functionalities need the information from the data-store

Repository architecture

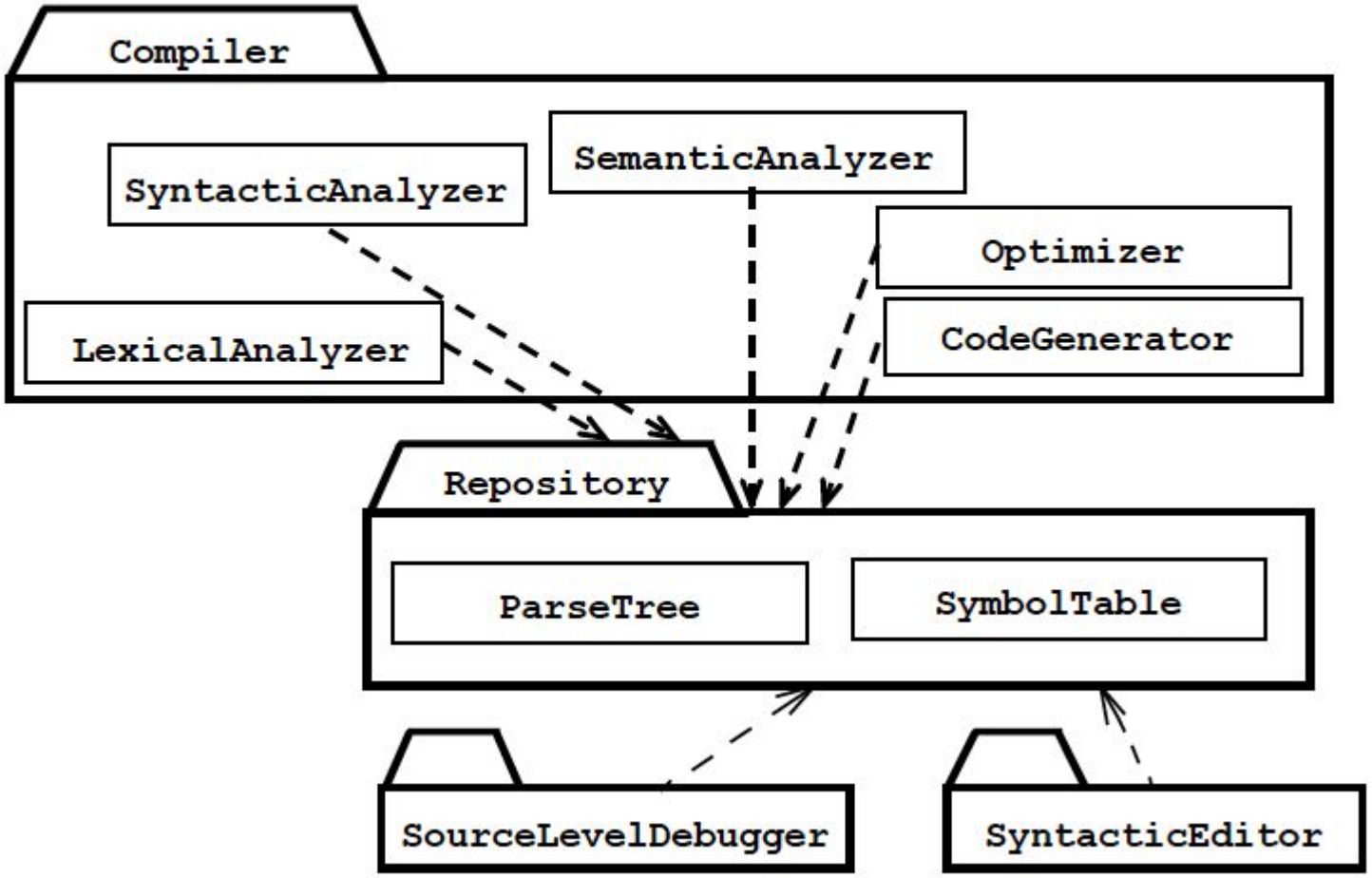


Repository architecture

The subsystems use and modify (i.e., they have access to) a shared data structure named repository.

The subsystems are relatively independent, in that they interact only through the repository.

Example: a compiler architecture based on a repository



Repository: two flavors

Data-centered style supporting user interactions

- The control flow into the system can be managed both by the repository (if stored data have to be modified) and by the subsystems (independent control flow)
- Passive repository: agents write or read items in the repository
- Active repository (blackboard, tuple space): the repository calls registered agents. This can be implemented with publish/subscribe

Repository: pros and cons

Benefits

- It is an effective way to share huge amounts of data: write once for all to read
- Each subsystem has not to take care of how data are produced/consumed by other subsystems
- It allows a centralized management of system backups, as well as of security and recovery procedures
- The data sharing model is available as the repository schema, hence it is easy to plug new subsystems

Repository: pros and cons

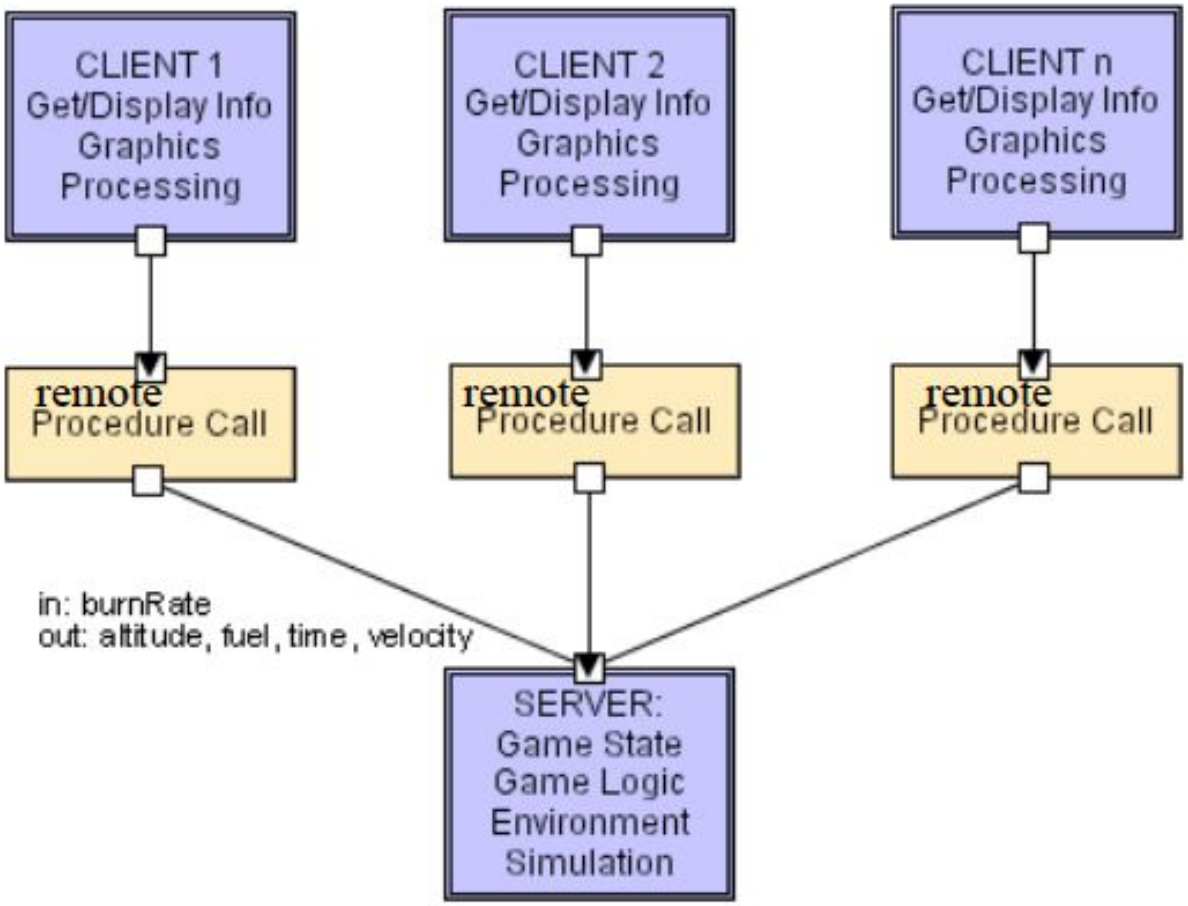
Pitfalls

- Subsystems have to agree on a data model, thus impacting on performance
- Data evolution: it is “expensive” to adopt a new data model since (a) it has to be applied on the entire repository, and (b) all the subsystems have to be updated
- Not for all the subsystems’ requirements in terms of backup and security are always supported by the repository
- It is tricky to deploy the repository on several machines, preserving the logical vision of a centralized entity, due to redundancy and data consistency matters

Styles

Data Flow	Dominated by movement of data through the system, with no “upstream” content control by recipient	Pipes and filters Batch sequential
Data Centered	Dominated by a complex central data store manipulated by independent components	Repository Blackboards
Data Sharing	Dominated by sharing of data amongst components	Hypertext systems Compound document
Call & Return systems	Dominated by order of computation, usually with a single thread of control	Procedural Object oriented Naive client server
Interacting Processes	Dominated by communication between independent, usually concurrent, processes	Event Driven Event Sourcing Broker SOA
Hierarchical Systems	Dominated by reduced coupling with the partition of the system into subsystems with limited interactions	Interpreters Rule-Based systems Layers CQRS Plugins

Client server



Client-server paradigm

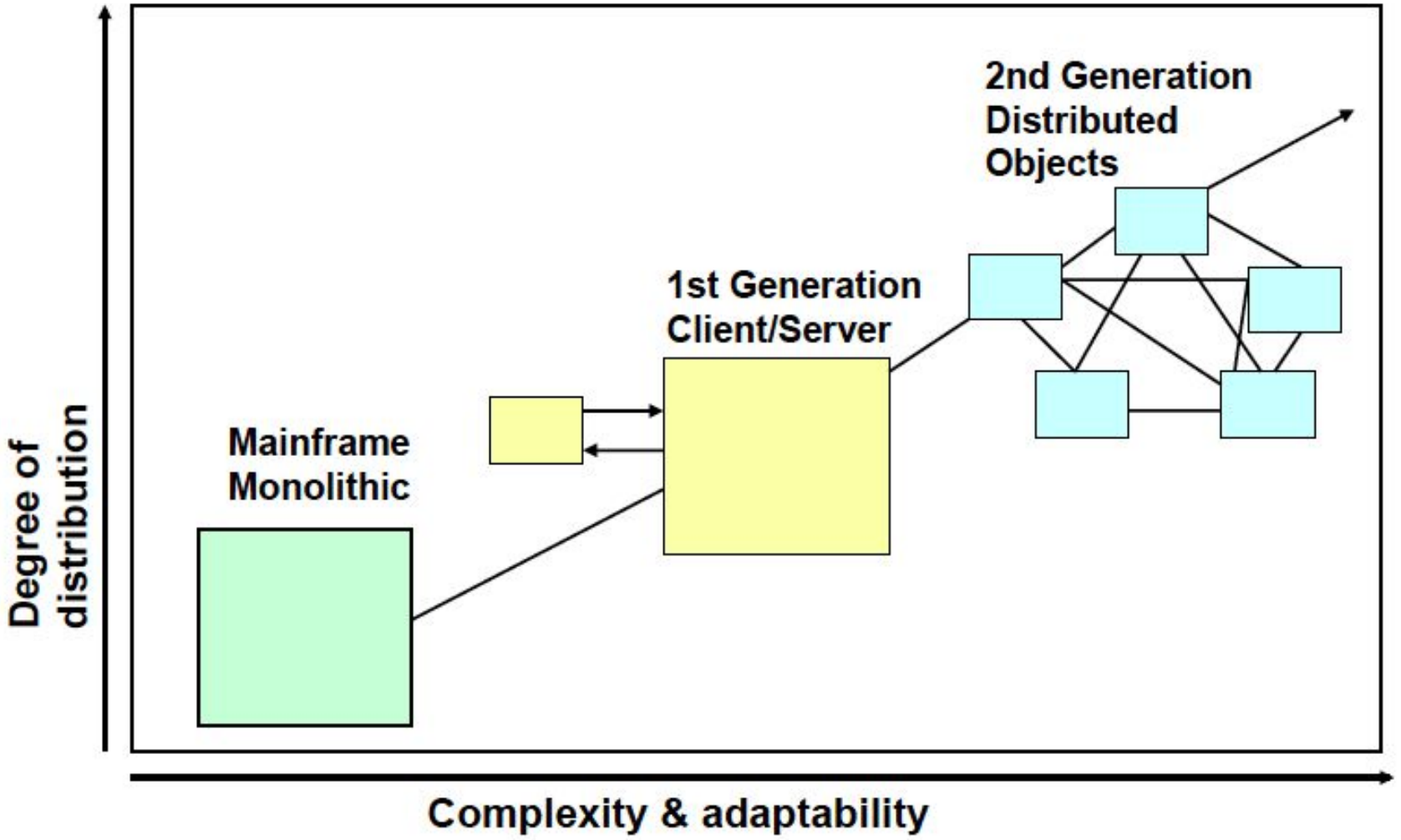
It has been conceived in the context of distributed systems, aiming to solve a synchronization issue:

- A protocol was needed to allow the communication between two different programs

The driving idea is to make unbalanced the partners' roles within a communication process

- Reactive entities (named servers):
  - They cannot initiate a communication
  - … they can only answer to incoming requests (reactive entities)
- Active entities (named clients):
  - They trigger the communication process
  - They forward requests to the servers, then they wait for responses

Client-server paradigm



Client-server style

- From the architectural viewpoint, a client/server system is made up of components which can be classified according to the service abstraction:
  - Who provides services is a server;
  - Who requires a service is a client;
- A client is a program used by a user to communicate with a system
  - …but a server can be a client for a different service
- Clients are aware of the server interface
- Servers cannot foresee clients' requests

Client-server style

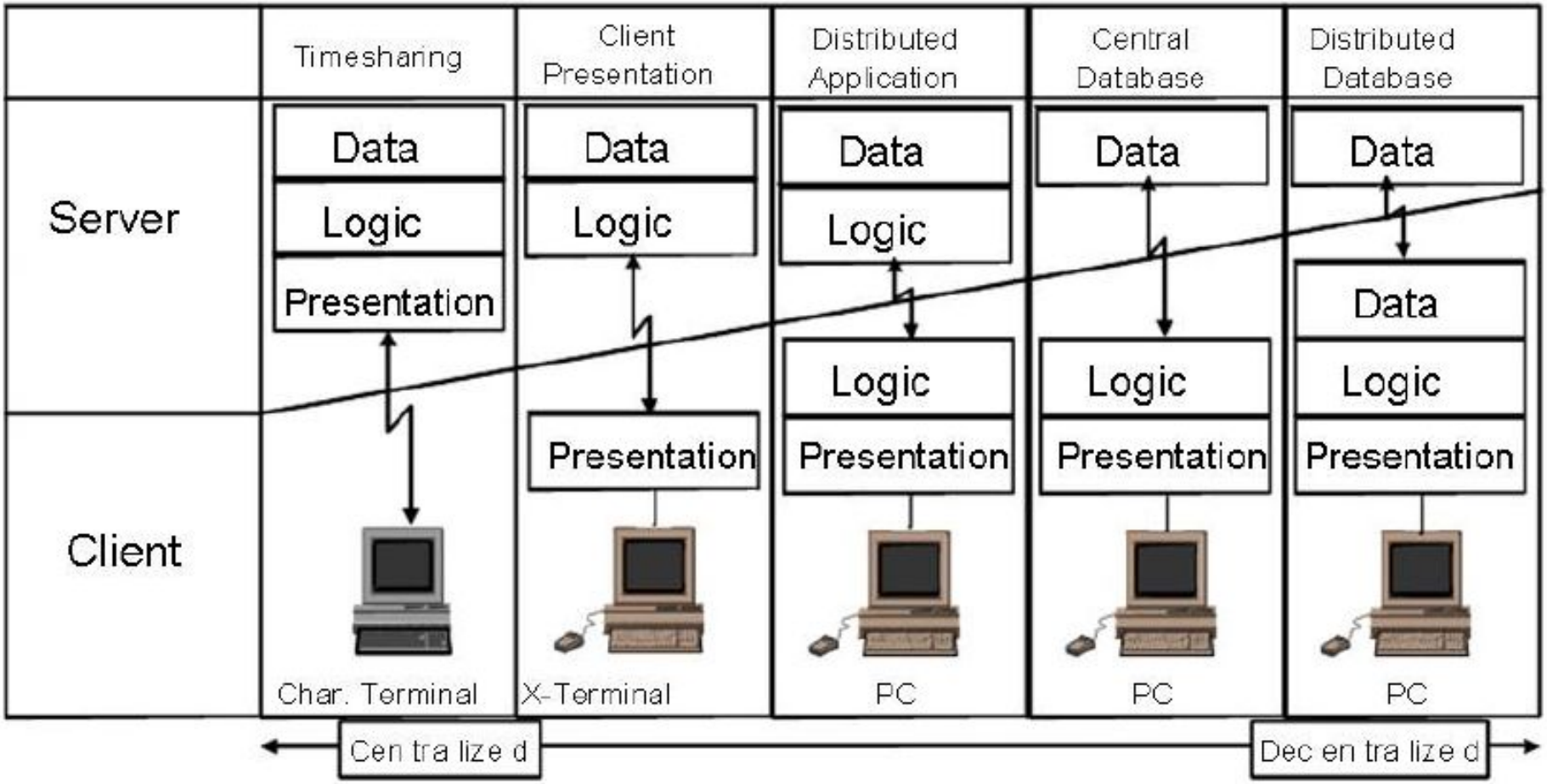
Most systems can be logically divided into three parts:

- The presentation, which is in charge of managing the user interface (graphic events, input fields check, help..)
- The actual application logic
- The data management tier for the management of persistent data.

The system architecture is defined according to how these parts are organized :

- 2-tiered, 3-tiered, or n-tiered

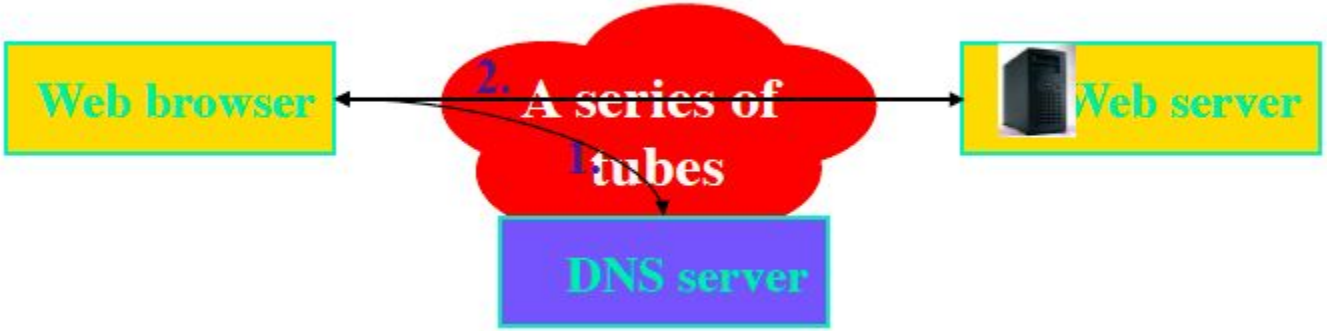
Client-server style



Web has a Client-Server style

HTTP (Hypertext Transfer Protocol), ASCII-based request/reply protocol that runs over TCP

HTTPS: variant that first establishes symmetrically-encrypted channel via public-key handshake, so suitable for sensitive info



Two-tier C/S architectures

Pitfalls

Two-tiers C/S architectures exhibit a heavy message traffic since the front-ends and the servers communicate intensively

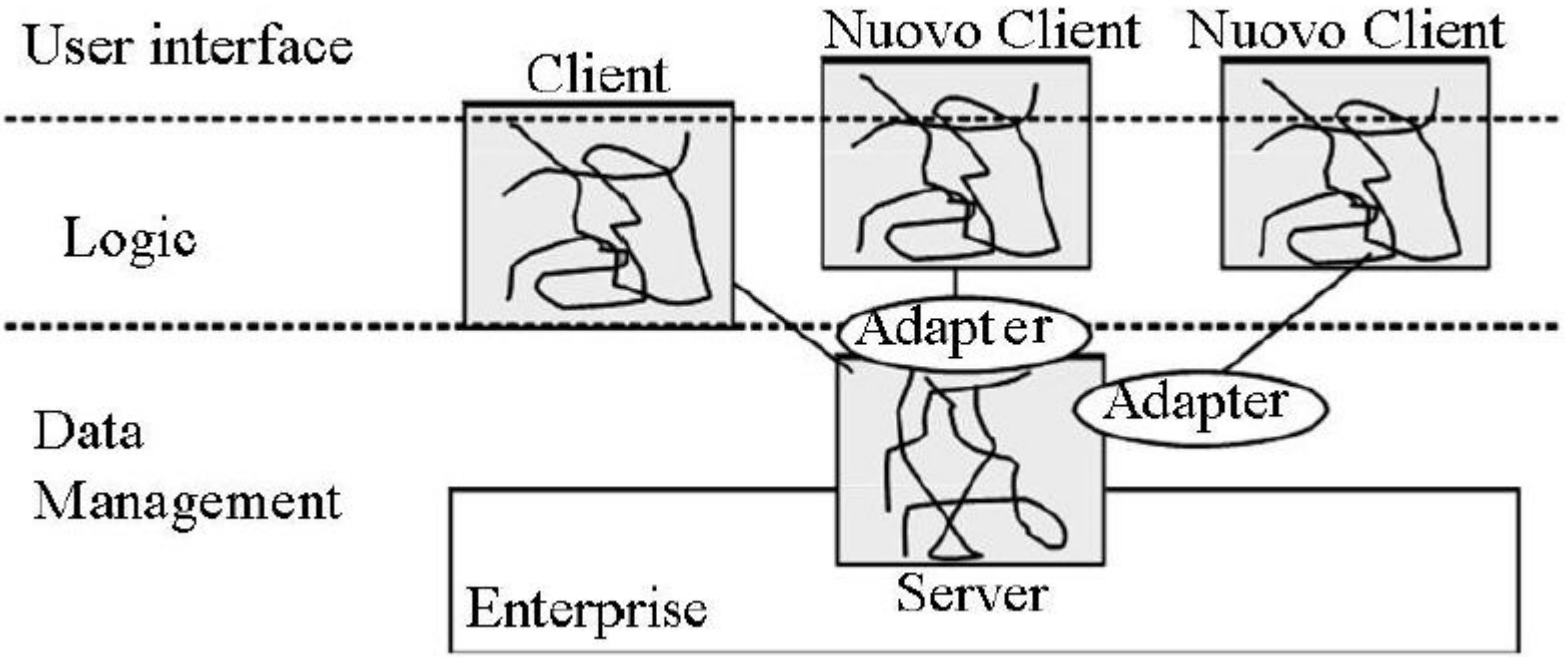
The business logic is not managed by an ad-hoc component, but is it “shared” by front-end and back-end

- client and server depend one from each other
- It is difficult to reuse the same interface to access different data
- It is difficult to interact with databases which have different frontends
- The business logic is encapsulated into the user interface
- If the logic changes, the interface has to change as well

Two-tier C/S architectures

Recurrent problem

Business and presentation logic are not clearly separate. E.g., if there is a service which can be accessed by several devices (e.g., mobile phone, desktop PC). The same logic but different interface.



Three-tier architectures

Early 90's; they propose a clear separation among logics:

- Tier 1: data management (DBMS, XML files, …..)
- Tier 2: business logic (application processing, …)
- Tier 3: user interface (data presentation and services)

Each tier has its own goals, as well as specific design constraints. No assumptions at all about the structure and/or the implementation of the other tiers , i.e.:

- Tier 2 does not make any assumptions neither on how data are represented, nor on how user interfaces are made
- Tier 3 does not make any assumptions on how business logic works

Three-tier architectures

Tiers 1 and 3 do not communicate, i.e.:

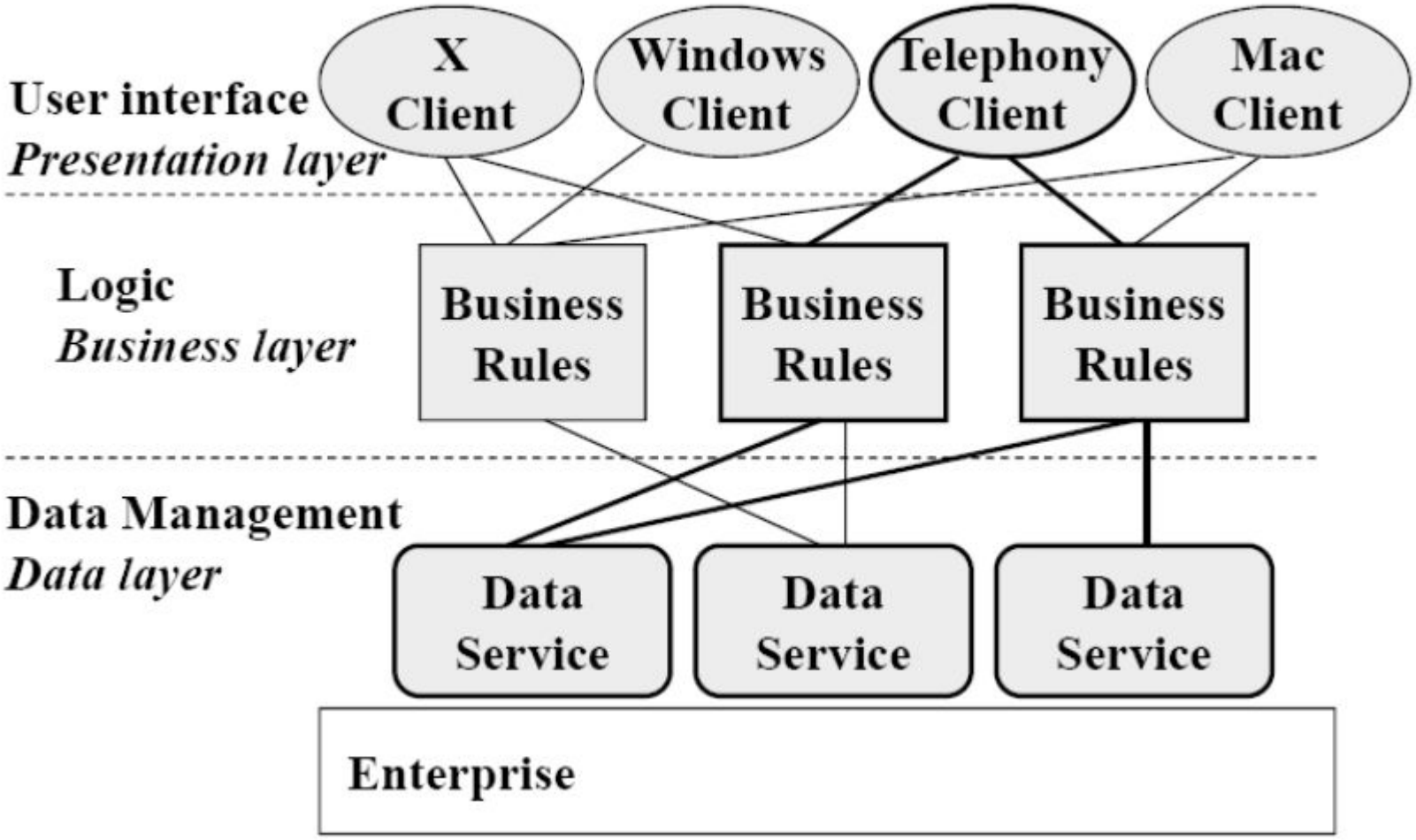
- The user interface neither receives any data from data management, nor it can write data
- Information passing (in both the directions) are filtered by the business logic

Tiers work as they were not part of a specific

application:

- Applications are conceived as collections of interacting components
- Each component can take part to several applications at the same time

Three-tier architectures



Three-tier architectures

Benefits:

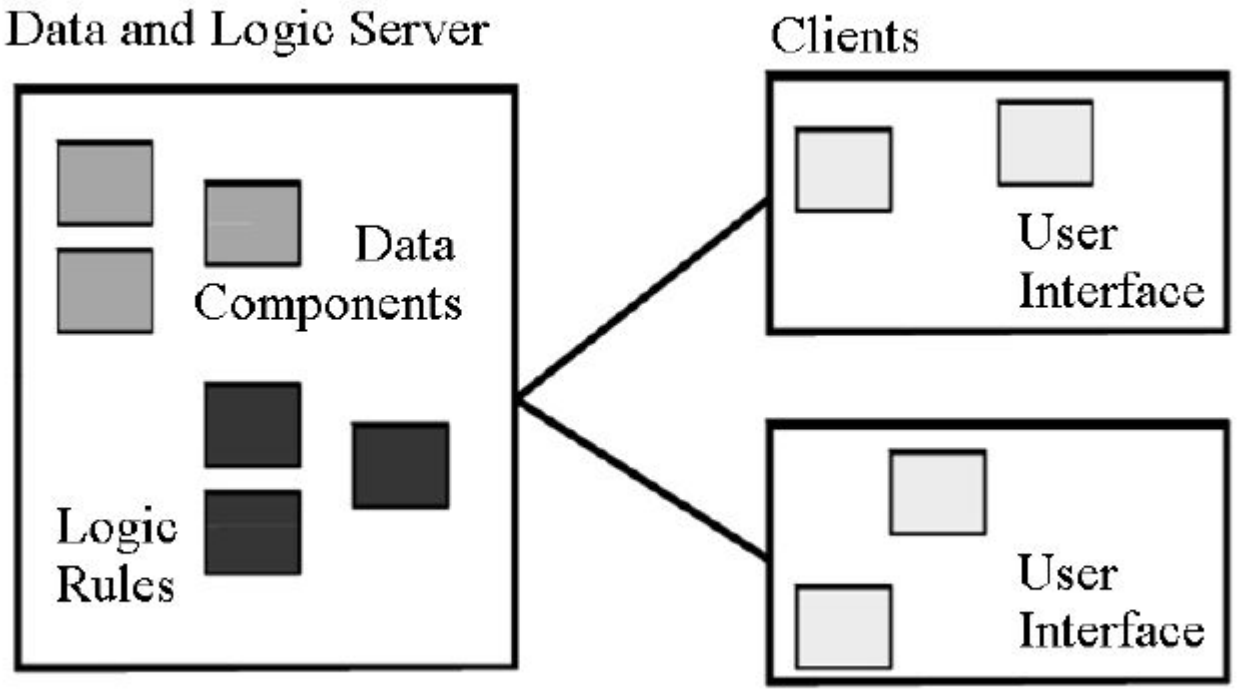
They leverage the flexibility of and the high modifiability of modular systems

- Components can be used in several systems
- A change into a given component do not have any impacts on the system (apart from changes into the API)
- It is easier to localize a bug since the system functionalities are isolated
- New functionalities can be added to the system by only
- modifying the components which are in charge of realizing them, or by plugging new components

Three-tier architectures

Tiers are **logical** abstractions, not physical entities

Three-tier architectures can be realized by using only one or two levels of machines



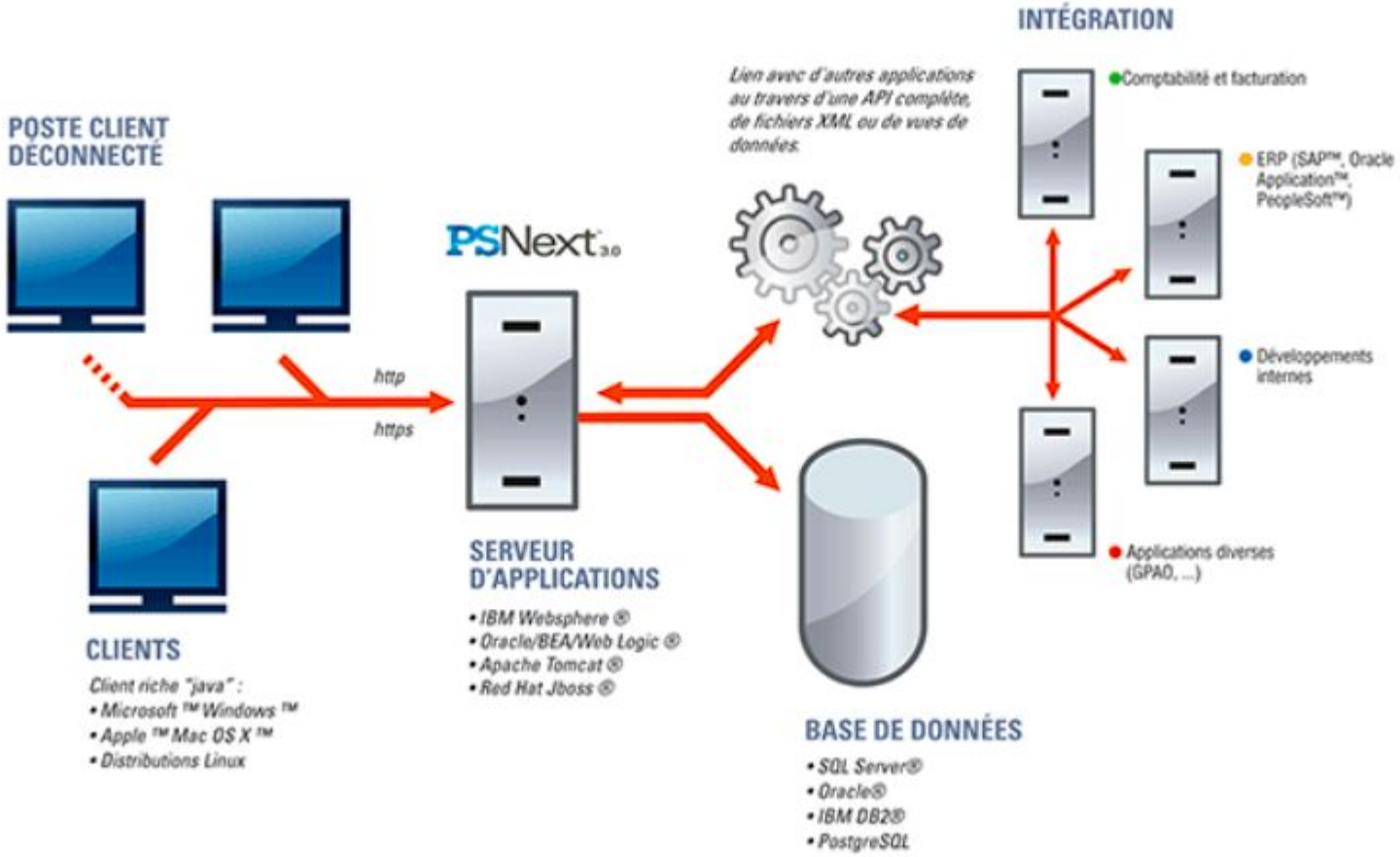
N-tier architectures

They provide high flexibility

Fundamental items:

- User Interface (UI): e.g., a browser, a WAP minibrowser, a graphical user interface (GUI)
- Presentation logic, which defines what the UI has to show and how to manage users' requests
- Business logic, which manages the application business rules
- Infrastructure services
  - The provides further functionalities to the application components (messaging, transactions support);
- Data tier:
  - Application Data level

N-tier example

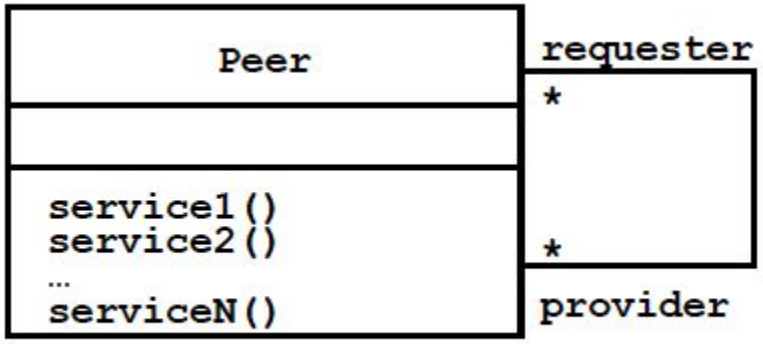


Styles

Data Flow	Dominated by movement of data through the system, with no “upstream” content control by recipient	Pipes and filters Batch sequential
Data Centered	Dominated by a complex central data store manipulated by independent components	Repository Blackboards
Data Sharing	Dominated by sharing of data amongst components	Hypertext systems Compound document
Call & Return systems	Dominated by order of computation, usually with a single thread of control	Procedural Object oriented Naive client server
Interacting Processes	Dominated by communication between independent, usually concurrent, processes	Event Driven Event Sourcing Broker SOA
Hierarchical Systems	Dominated by reduced coupling with the partition of the system into subsystems with limited interactions	Interpreters Rule-Based systems Layers CQRS Plugins

Peer-to-peer style (P2P)

- They can be considered a generalization of the client/server architecture
- Each subsystem is able to provide services, as well as to make requests
  - Each subsystem acts both as a server and as a client



Peer-to-peer architectures

- In such an architecture, all nodes have the same abilities, as well as the same responsibilities. All communications are (potentially) bidirectional
- The goal:
  - To share resources and services (data, CPU cycles, disk space, …)
- P2P systems are characterized by:
  - Decentralized control
  - Adaptability
  - Self-organization and self-management capabilities

Peer-to-peer architectures

- Typical functional characteristics of P2P systems
  - File sharing system
  - File storage system
  - Distributed file system
  - Redundant storage
  - Distributed computation
- Typical non-functional requirements
  - Availability
  - Reliability
  - Performance
  - Scalability
  - Anonymity

Event sourcing

We can query an application's state to find out the current state of the world, and this answers many questions. However there are times when we don't just want to see where we are, we also want to know how we got there.

Event Sourcing ensures that all changes to application state are stored as a sequence of events. Not just can we query these events, we can also use the event log to reconstruct past states, and as a foundation to automatically adjust the state to cope with retroactive changes.

Event sourcing

Let's consider a simple example to do with shipping notifications. In this example we have many ships on the high seas, and we need to know where they are. A simple way to do this is to have a tracking application with methods to allow us to tell when a ship arrives or leaves at a port.

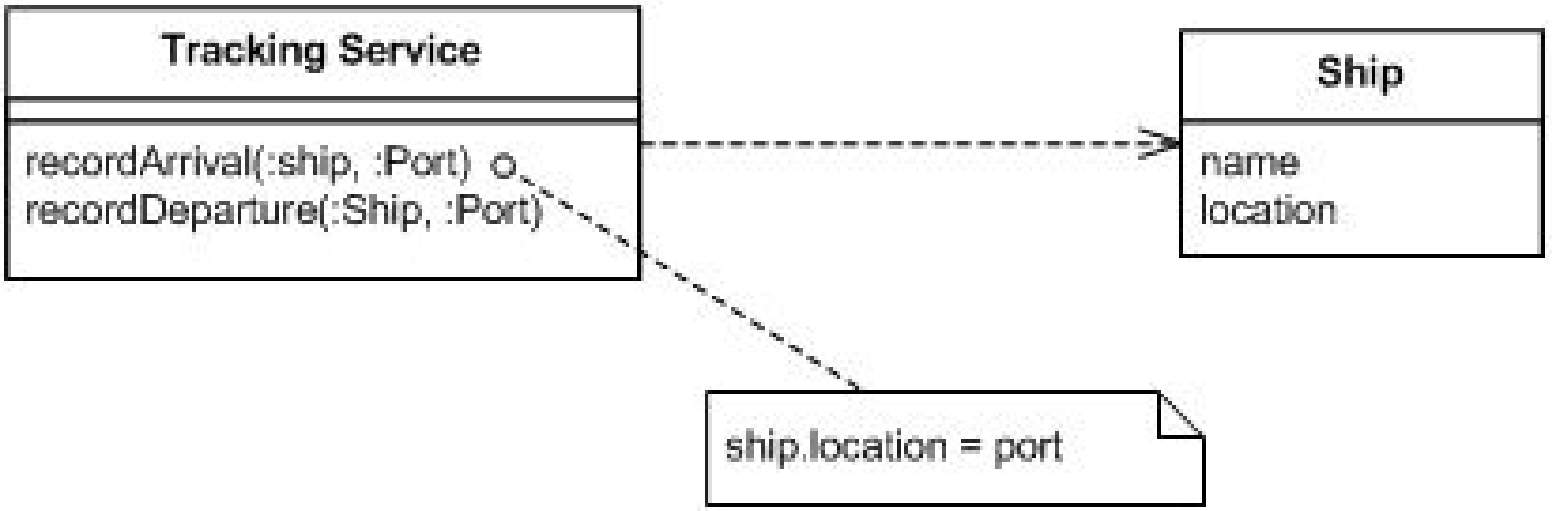


Figure 1: A simple interface for tracking shipping movements.

Introducing Event Sourcing adds a step to this process. Now the service creates an event object to record the change and processes it to update the ship.

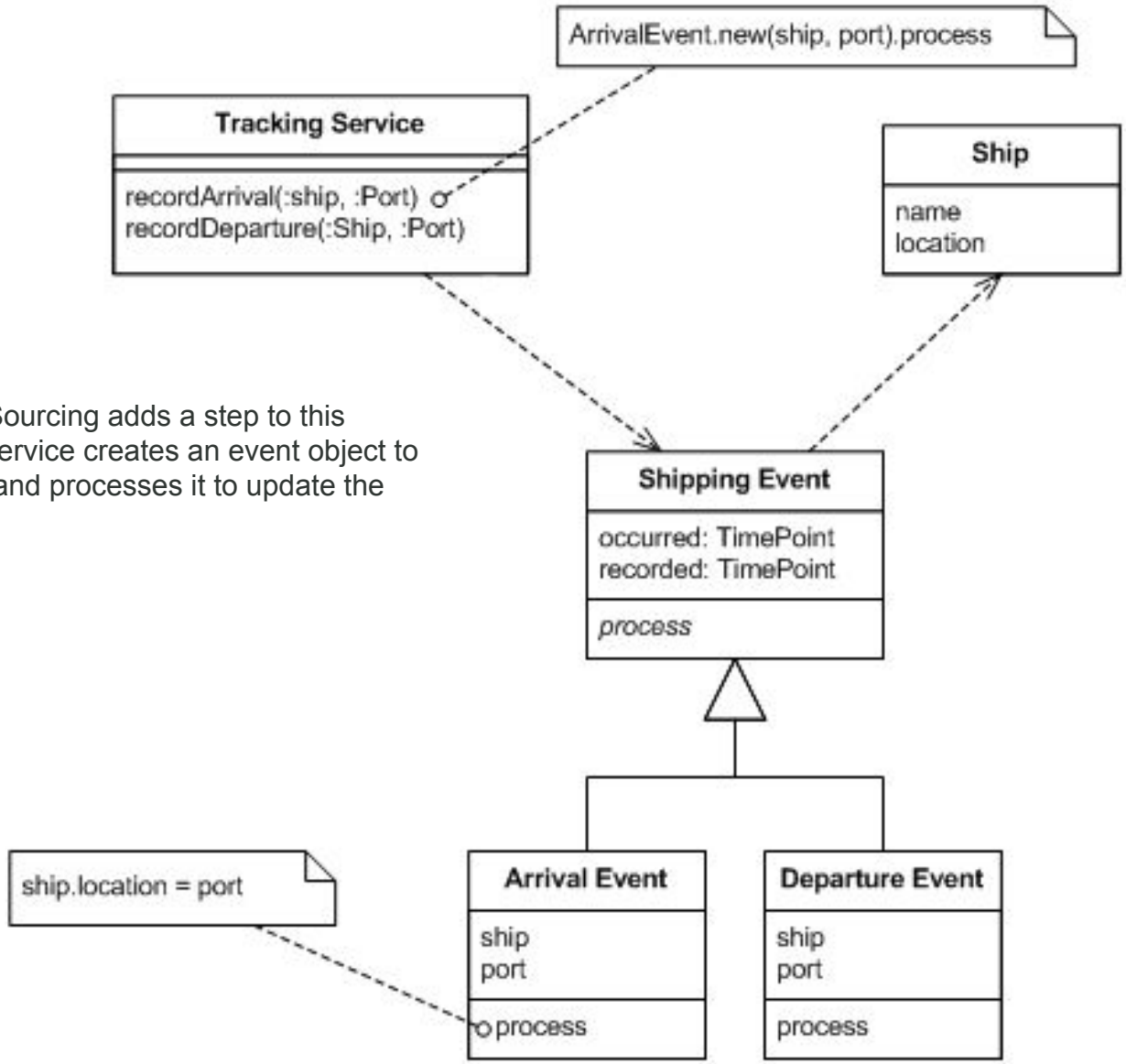


Figure 2: Using an event to capture the change.

Event sourcing

Looking at just the processing, this is just an unnecessary level of indirection. The interesting difference is when we look at what persists in the application after a few changes. Let's imagine some simple changes:

- The Ship 'King Roy' departs San Francisco
- The Ship 'Prince Trevor' arrives at Los Angeles
- The Ship 'King Roy' arrives in Hong Kong

With the basic service, we see just the final state captured by the ship objects. I'll refer to this as the application state.

Figure 3: State after a few movements tracked by simple tracker.

Event sourcing

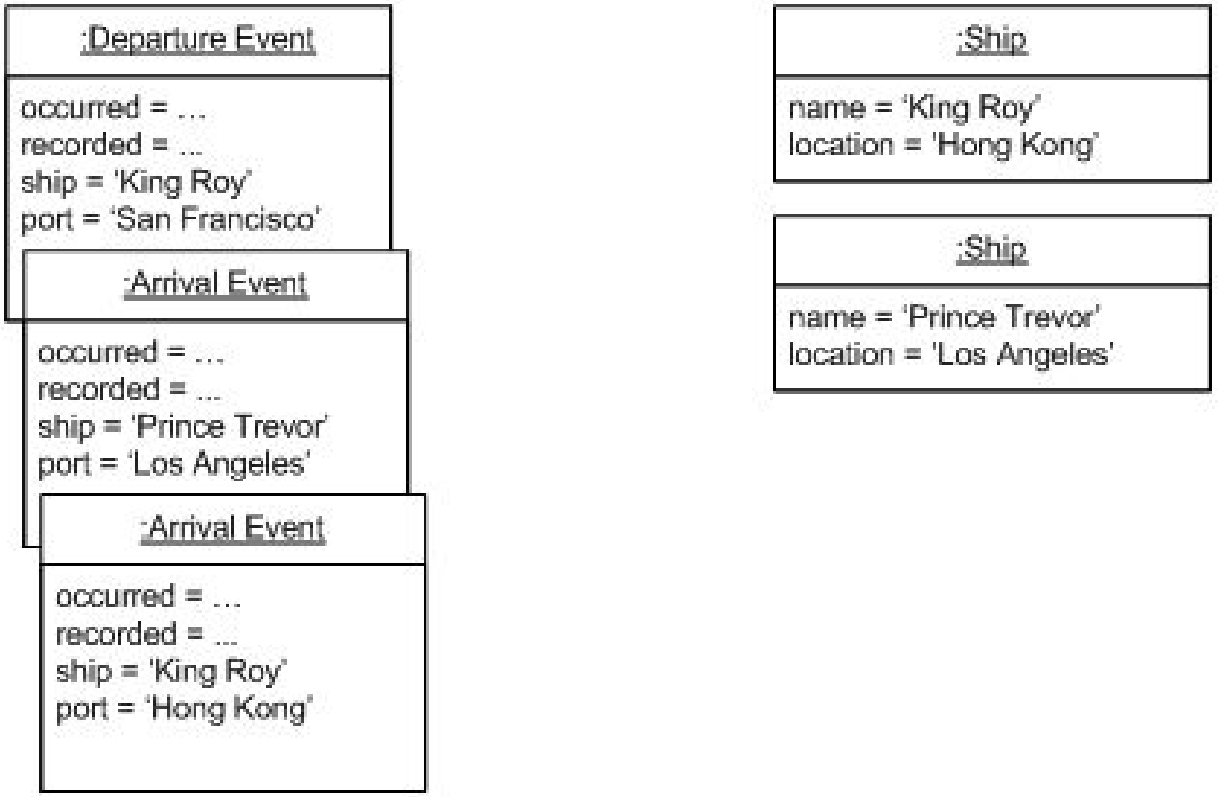


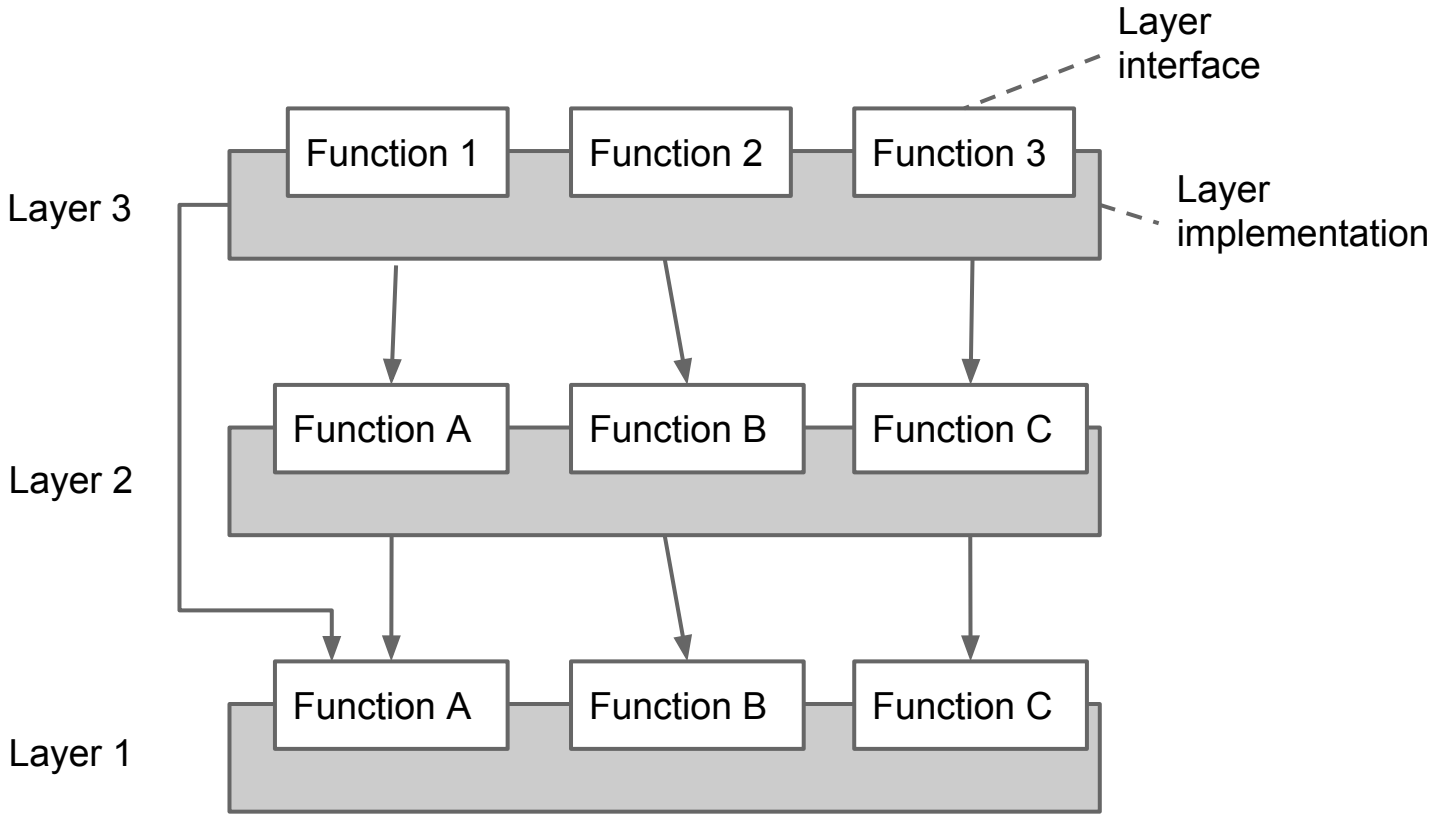
Figure 4: State after a few movements tracked by event sourced tracker.

Event sourcing

The key to Event Sourcing is that we guarantee that all changes to the domain objects are initiated by the event objects. This leads to a number of facilities that can be built on top of the event log:

- Complete Rebuild: We can discard the application state completely and rebuild it by re-running the events from the event log on an empty application.
- Temporal Query: We can determine the application state at any point in time. Notionally we do this by starting with a blank state and rerunning the events up to a particular time or event. We can take this further by considering multiple time-lines (analogous to branching in a version control system).
- Event Replay: If we find a past event was incorrect, we can compute the consequences by reversing it and later events and then replaying the new event and later events. (Or indeed by throwing away the application state and replaying all events with the correct event in sequence.) The same technique can handle events received in the wrong sequence - a common problem with systems that communicate with asynchronous messaging.

Layered systems



Layered systems

Each layer *provides services* to the layer above and acts as a *client* for a layer below.

The connectors are the *protocols* that define how the layers can interact.

Lower layers can pass data and service requests to higher layers via notifications (i.e. an observable).

Layered systems

Closed vs open layers

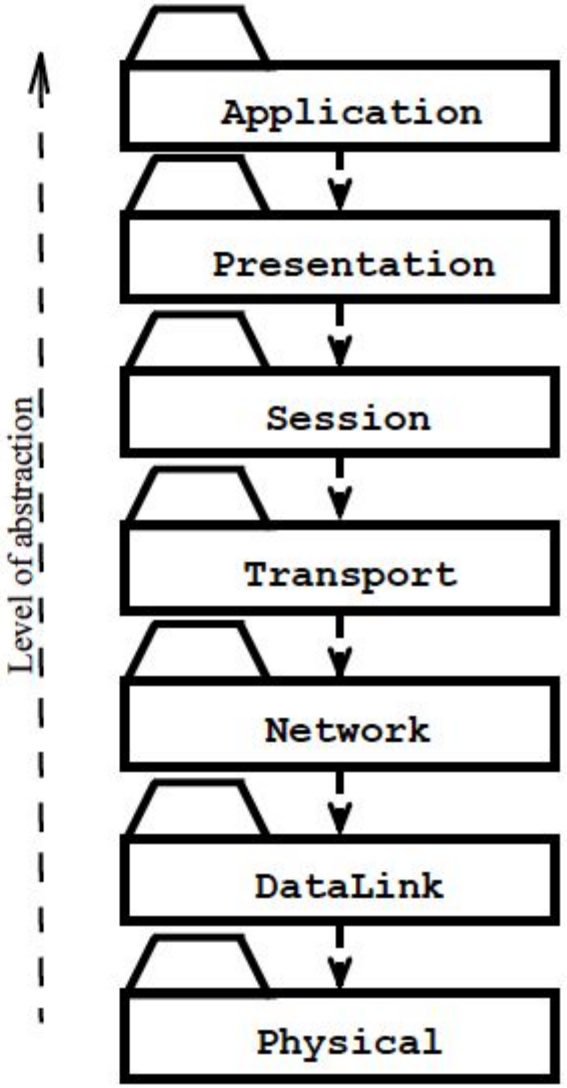
Closed architecture: the i-th layer can only have access to the layer (i-1)-th

Open architecture: the i-th layer can have access to all the underlying layers (i.e., the layers lower than i)

Layered systems

The ISO/OSI reference model defines 7 network layers, characterized by an increasing level of abstraction

Eg. The Internet Protocol (IP) defines a VM at the network level able to identify net hosts and transmit single packets from host to host



Layered systems

Invariants

Layers can only interact with adjacent layers.

A layer can provide services to a layer above, and consume services provided by a layer below.

A layer depends only on lower layers and has no knowledge of the higher layers.

The structure can be compared with a stack or onion.

Properties

Designs are based on increasing level of abstraction. This allows portioning of a complex problem into a series of steps.

Changes to one layer impact at most two layers - below and above, supporting change.

Different implementations of the same layer can be reused interchangeably provided they support the same interface.

Implementation

Define the abstraction criteria for grouping tasks into layers

Determine the number of levels according to that criteria. Name those layers and assign tasks to them

Specify an interface for each layer

Specify the approach for communication between adjacent layers

Design error-handling strategy

Creating layers: façade

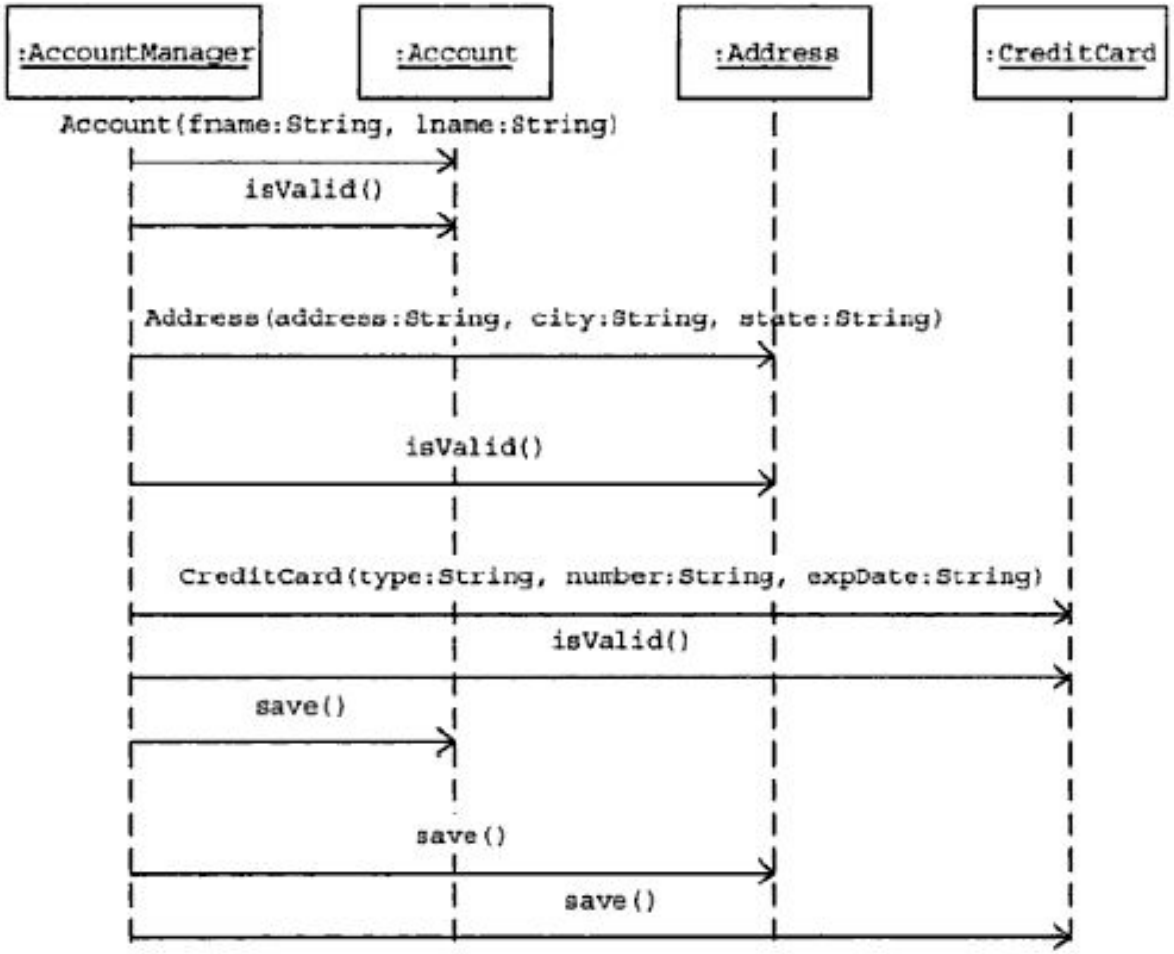


Figure 22.5 How a Client Would Normally Interact (Directly) with Subsystem Classes to Validate and Save the Customer Data

Creating layers: façade

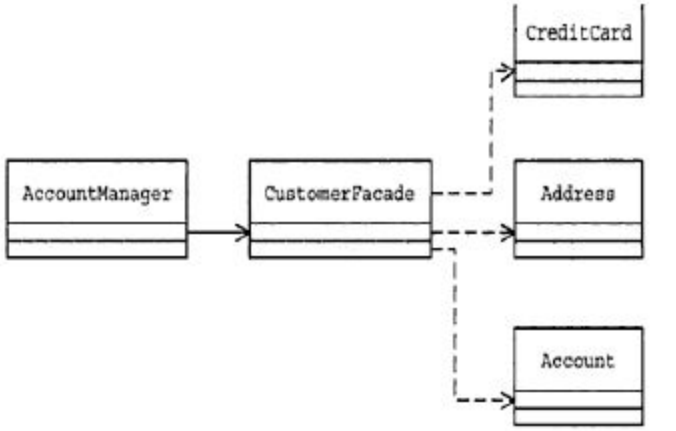


Figure 22.7 Class Association with the Façade Class in Place

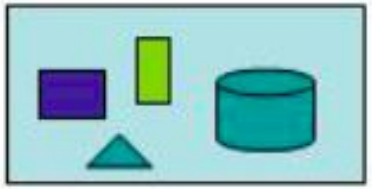
Figure 22.8 In the Revised Design, Clients Interact with the Façade Instance to Interface with the Subsystem

Tiers

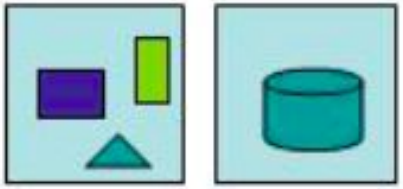
Another approach to managing complexity consists of partitioning a system into sub-systems (peers), which are responsible for a class of services.

The Computing Evolution

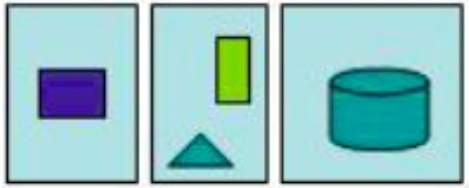
Monolithic
(one tier)



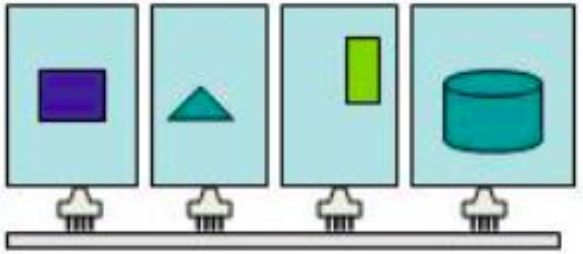
Client / Server



3-Tier



N-Service



Typical tiered architecture

Presentation tier

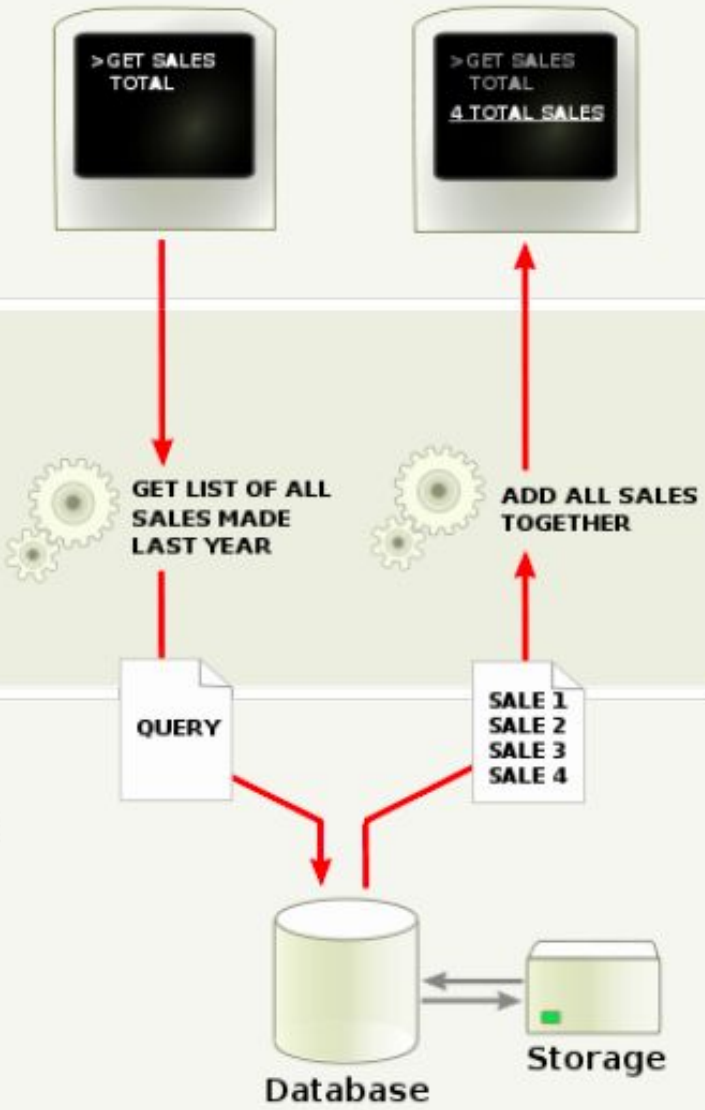
The top-most level of the application is the user interface. The main function of the interface is to translate tasks and results to something the user can understand.

Logic tier

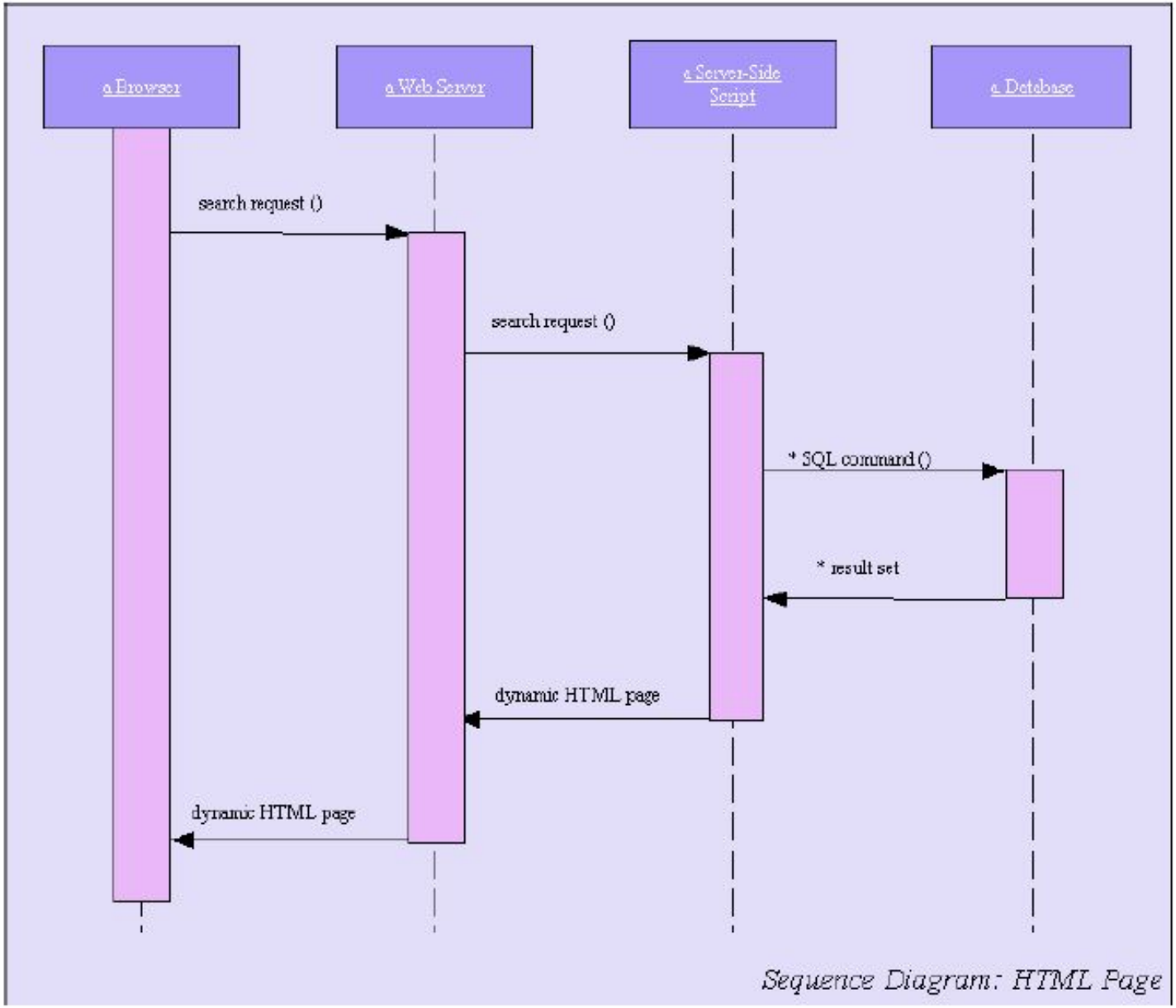
This layer coordinates the application, processes commands, makes logical decisions and evaluations, and performs calculations. It also moves and processes data between the two surrounding layers.

Data tier

Here information is stored and retrieved from a database or file system. The information is then passed back to the logic tier for processing, and then eventually back to the user.



4 tiers architecture



Layers and tiers

Typically, a complete decomposition of a given system comes from **both layering and partitioning.**

- First, the system is divided into top level subsystems which are responsible for certain functionalities (partitioning)
- Second, each subsystem is organized into several layers, if necessary, up to the definition of simple enough layers

Monolithic and Microservices

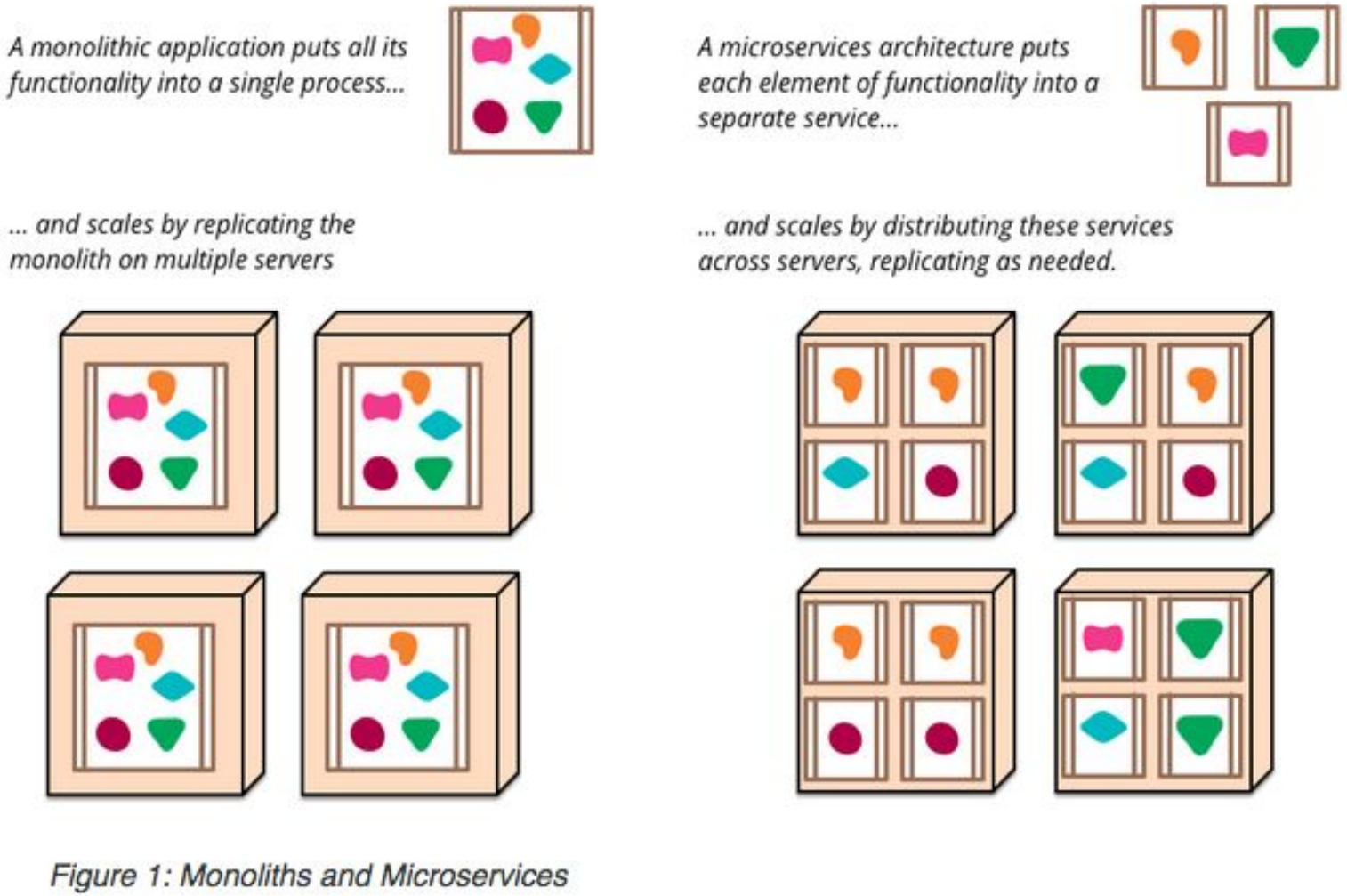
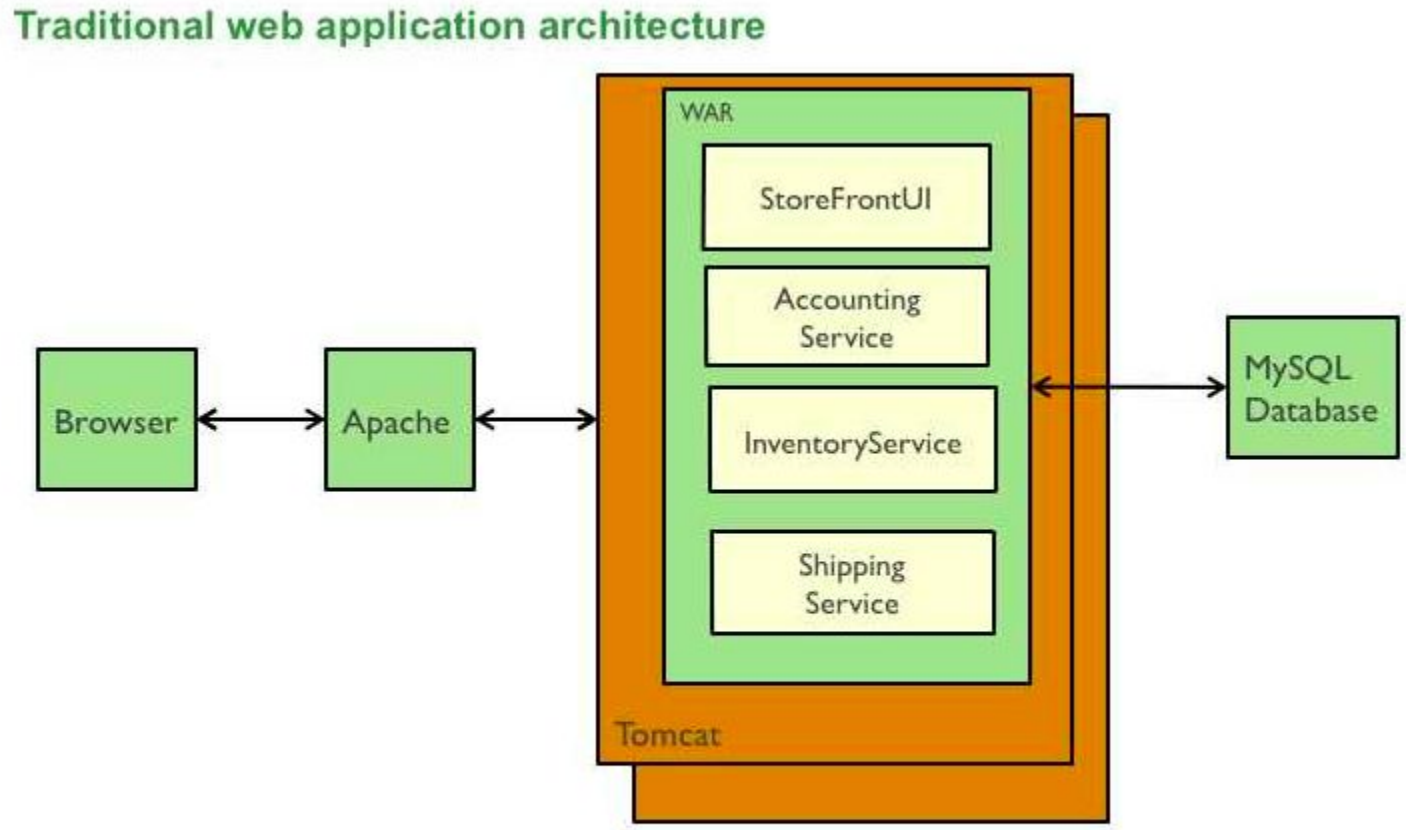


Figure 1: Monoliths and Microservices

by http://martinfowler.com/articles/microservices.html

Monolithic architecture



Benefits

Monolith solution has a number of benefits:

- **Simple to develop** - the goal of current development tools and IDEs is to support the development of monolithic applications
- **Simple to deploy** - you simply need to deploy the WAR file (or directory hierarchy) on the appropriate runtime
- **Simple to scale** - you can scale the application by running multiple copies of the application behind a load balancer

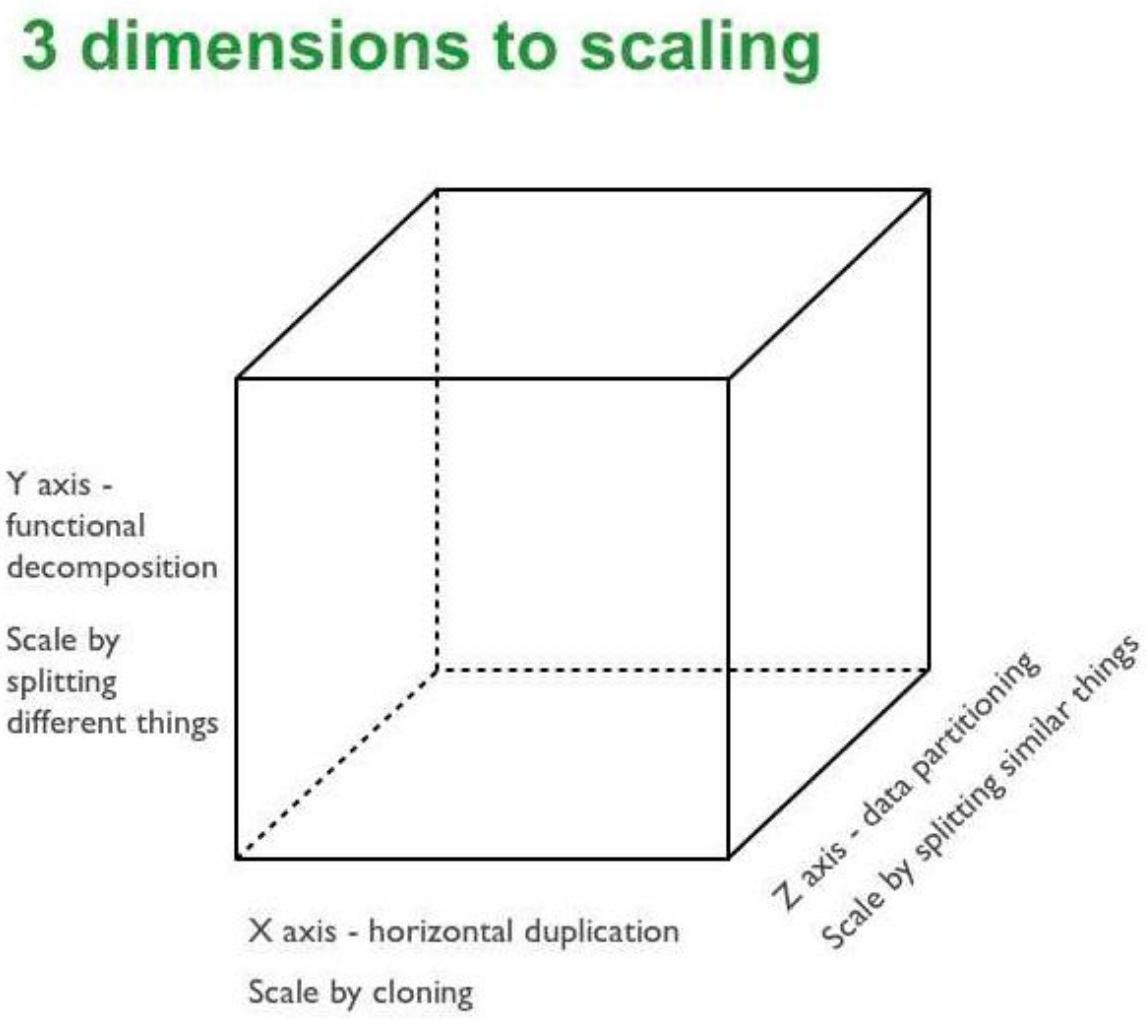
Drawbacks

- The application can be **difficult to understand and modify**. As a result, development typically slows down. Also, because there are not hard module boundaries, modularity breaks down over time.
- **Scaling** the application **can be difficult** - a monolithic architecture is that it can only scale in one dimension. Different application components have different resource requirements - one might be CPU intensive while another might memory intensive. With a monolithic architecture we cannot scale each component independently
- **Obstacle to scaling development** - Once the application gets to a certain size its useful to divide up the engineering organization into teams that focus on specific functional areas. We might want to have the UI team, accounting team, inventory team, etc. The trouble with a monolithic application is that it prevents the teams from working independently. The teams must coordinate their development efforts and redeployments. It is much more difficult for a team to make a change and update production.

Drawbacks

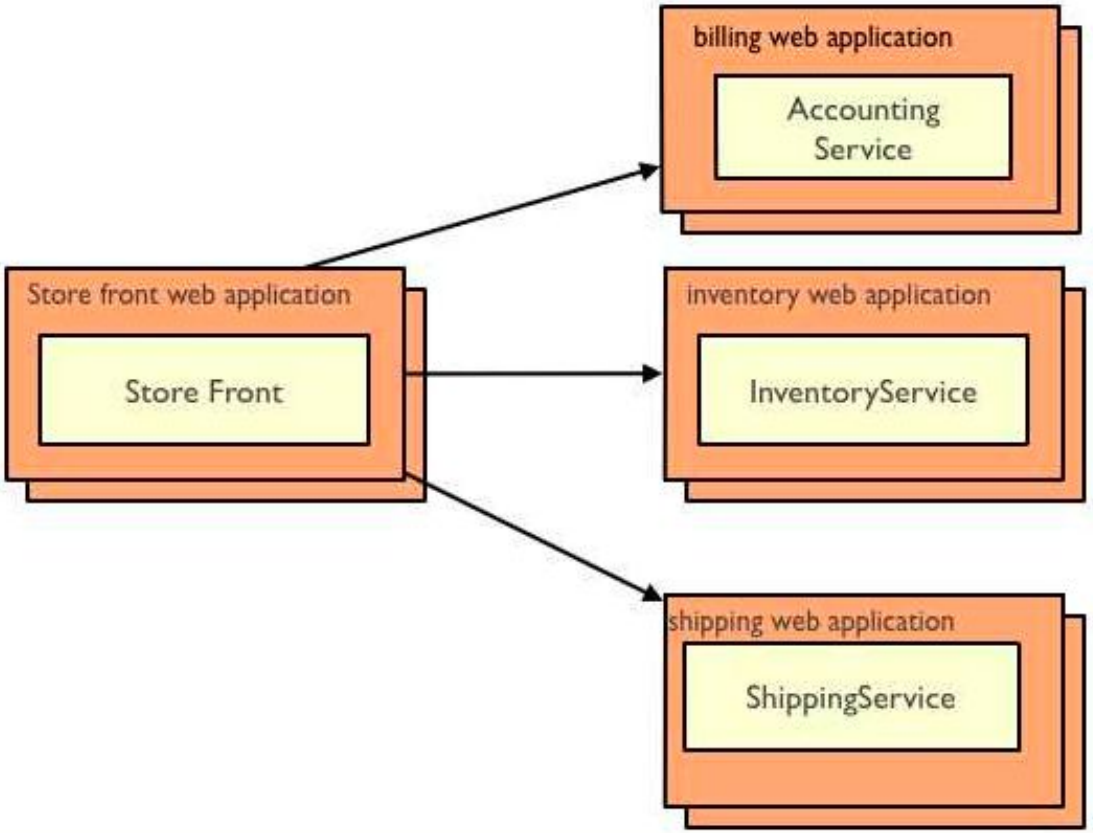
- Requires a **long-term commitment to a technology stack** - a monolithic architecture forces you to be married to the technology stack (and in some cases, to a particular version of that technology) you chose at the start of development. With a monolithic application, can be difficult to incrementally adopt a newer technology. It's possible that in order to adopt a newer platform framework you have to rewrite the entire application, which is a risky undertaking.
- **Continuous deployment is difficult** - a large monolithic application is also an obstacle to frequent deployments. In order to update one component you have to redeploy the entire application.
- **Overloaded IDE** - the larger the code base the slower the IDE and the less productive developers are.

Scale cube


3 dimensions to scaling
Y axis - functional decomposition
Scale by splitting different things
X axis - horizontal duplication
Scale by cloning
Z axis - data partitioning
Scale by splitting similar things

Microservices

Y axis scaling - application level



Apply X axis cloning and/or Z axis partitioning to each service

Benefits

- Each microservice is relatively small
  - Easier for a developer to understand
  - The IDE is faster making developers more productive
  - The web container starts faster, which makes developers more productive, and speeds up deployment
- Each service can be deployed independently of other services - easier to deploy new versions of services frequently
- Easier to scale development. It enables you to organize the development effort around multiple teams. Each (two pizza) team is responsible a single service. Each team can develop, deploy and scale their service independently of all of the other teams.
- Improved fault isolation. For example, if there is a memory leak in one service then only that service will be affected. The other services will continue to handle requests. In comparison, one misbehaving component of a monolithic architecture can bring down the entire system.
- Eliminates any long-term commitment to a technology stack

Drawbacks

- Developers must deal with the additional complexity of creating a distributed system.
  - Developer tools/IDEs are oriented on building monolithic applications and don't provide explicit support for developing distributed applications.
  - Testing is more difficult
  - Developers must implement the inter-service communication mechanism.
  - Implementing use cases that span multiple services without using distributed transactions is difficult
  - Implementing use cases that span multiple services requires careful coordination between the teams
- Deployment complexity In production, there is also the operational complexity of deploying and managing a system comprised of many different service types.

Drawbacks

- Increased memory consumption The microservices architecture replaces N monolithic application instances with NxM services instances. If each service runs in its own JVM (or equivalent), which is usually necessary to isolate the instances, then there is the overhead of M times as many JVM runtimes. Moreover, if each service runs on its own VM (e.g. EC2 instance), as is the case at Netflix, the overhead is even higher.

How to split microservices?

By verb/use-case (shipping/billing/etc)

By nouns (Inventory service)

Ideally each service should conform to SRP (Single Responsibility Principle).

One reason for change.

Conway’s law.

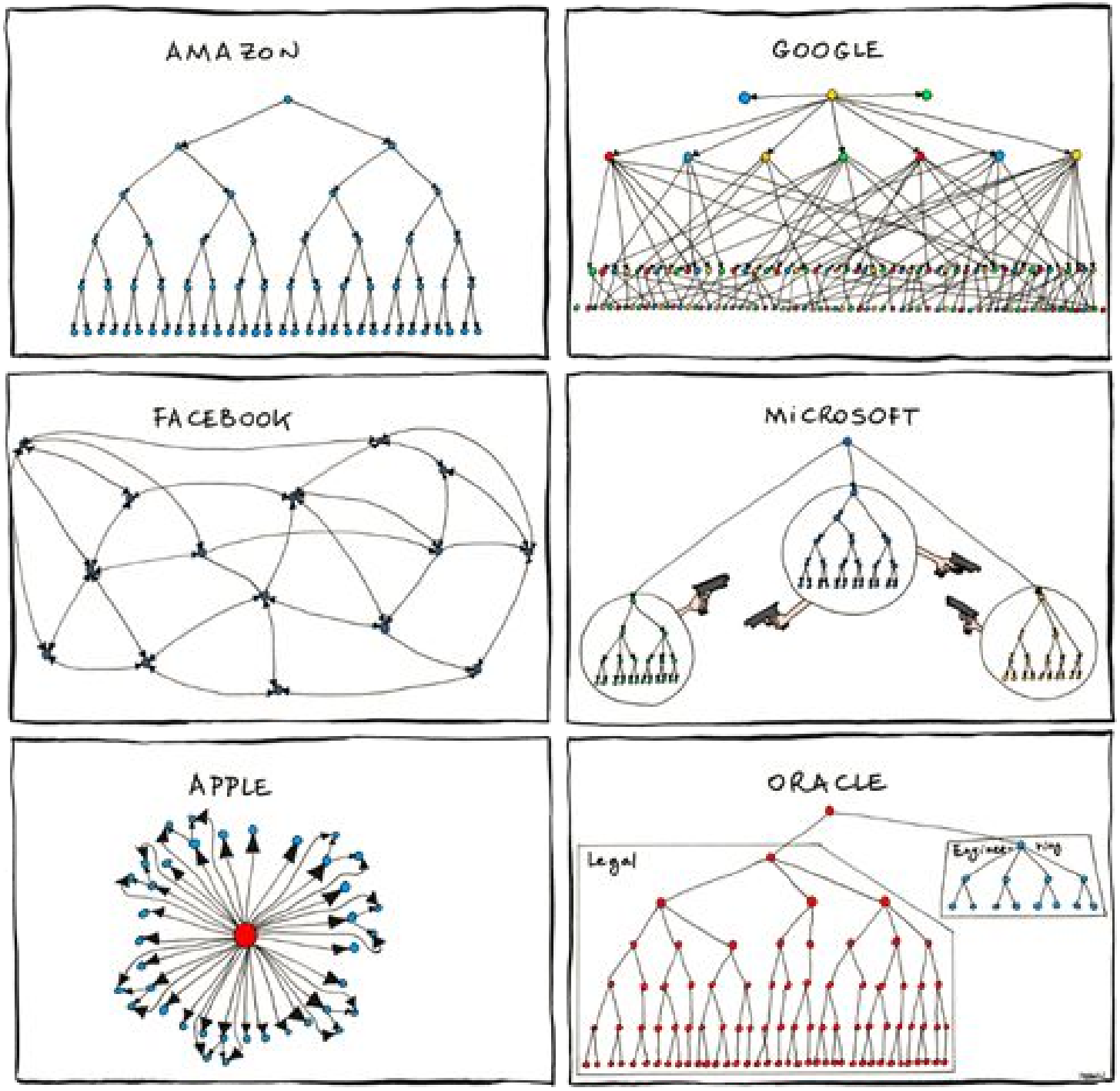
Conway’s law

"Any organization that designs a system (defined more broadly here than just information systems) will inevitably produce a design whose structure is a copy of the organization's communication structure."

Conway’s law

If you have four groups working on a compiler, you'll get a 4-pass compiler.

If a development team is set up with a SQL database specialist, a JavaScript/CSS developer and a C# developer you they will produce a system with three tiers: a database with stored procedures, a business middle tier and a UI tier. This design will be applied whether it is the best or not. Indeed, the system might not even need a relational database or a graphical interface - a flat file might be quite adequate for data storage and a command line interface perfectly acceptable.


AMAZON
GOOGLE
FACEBOOK
MICROSOFT
APPLE
ORACLE
Legal

Known uses

Most large scale web sites including Netflix, Amazon, eBay and LinkedIn have evolved from a monolithic architecture to a microservices architecture.

eBay

- 5th generation today
- Monolithic Perl -> Monolithic C++ -> Java -> microservices

Twitter

- 3rd generation today
- Monolithic Rails -> JS / Rails / Scala -> microservices

Amazon

- Nth generation today
- Monolithic C++ -> Java / Scala -> microservices

LinkedIn

- Monolithic Java app -> more than 100 microservices (2010)

Sample app + talk

Talk + app about event sourcing, cqrs and microservices

http://plainoldobjects.com/presentations/building-and-deploying-microservices-with-event-sourcing-cqrs-and-docker/

https://github.com/cer/event-sourcing-examples

Links

http://c2.com/cgi/wiki?HexagonalArchitecture

http://www.dossier-andreas.net/software_architecture/

http://blog.8thlight.com/uncle-bob/2012/08/13/the-clean-architecture.html

http://jeffreypalermo.com/blog/the-onion-architecture-part-1/

http://martinfowler.com/eaaDev/EventSourcing.html

http://codebetter.com/gregyoung/2010/02/16/cqrs-task-based-uis-event-sourcing-agh/

http://martinfowler.com/bliki/CQRS.html

http://www.udidahan.com/2009/12/09/clarified-cqrs/

http://blog.8thlight.com/uncle-bob/2011/09/30/Screaming-Architecture.html

http://microservices.io/patterns/microservices.html

http://allankelly.blogspot.ru/2013/03/conway-law-v-software-architecture.html