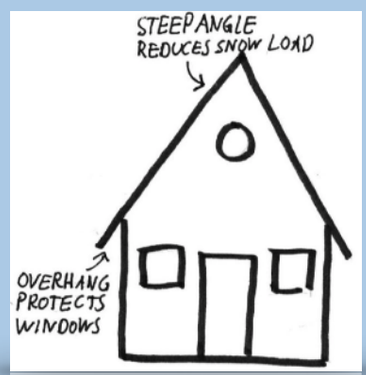
b
UNIVERSITÄT
BERN

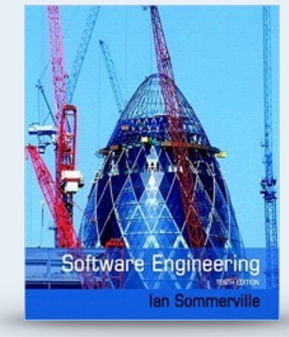
Introduction to Software Engineering

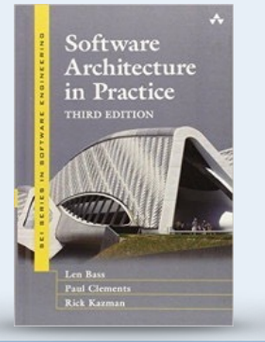
Software Architecture

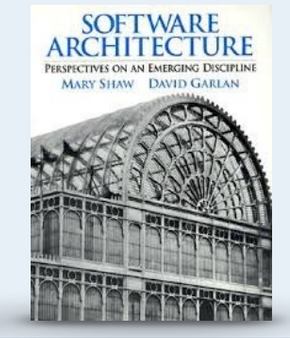


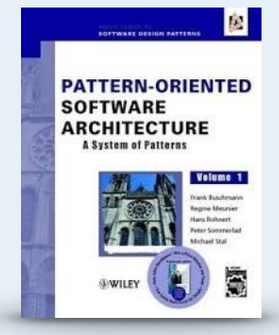
Selected material by Mircea Lungu and Andrea Caracciolo

Sources









- *Software Engineering*. Ian Sommerville. Addison-Wesley, 10th edition, 2015
- *Software Architecture in Practice*. L. Bass, P. Clements, and R. Kazman., Addison Wesley, 3rd edition, 2012
- *Software Architecture: Perspectives on an Emerging Discipline*, M. Shaw, D. Garlan, Prentice-Hall, 1996
- *Pattern-Oriented Software Architecture — A System of Patterns*, F. Buschmann, et al., John Wiley, 1996

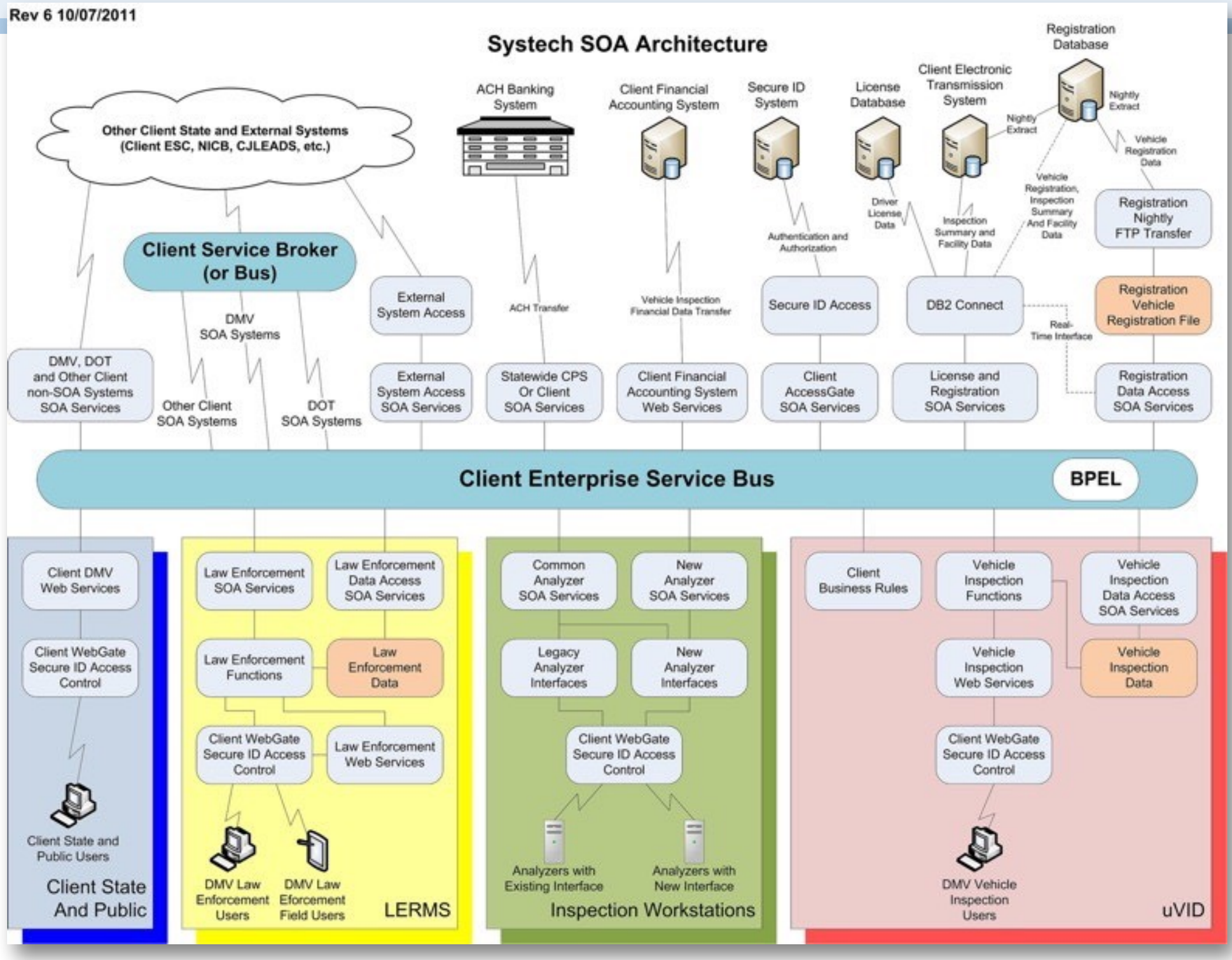
Roadmap

- What is Software Architecture?
- Cohesion and Coupling
- Architectural styles
- UML diagrams for architectures

Roadmap

- What is Software Architecture?
- Cohesion and Coupling
- Architectural styles
- UML diagrams for architectures

Is this Software Architecture?



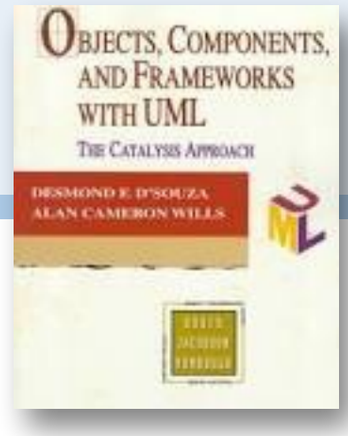
Usually if you ask what the architecture is of a given software system, you will get a diagram something like this, showing a high-level view of the systems components and their relationships.

What makes this an “architecture”? What is it that a software architecture is intended to express?

Diagram taken from this blog post:

https://blog.ndepend.com/visualizing-software-architecture/

What it is not …



A neat-looking drawing of some boxes, circles, and lines, laid out nicely in Powerpoint or Word, does not constitute an architecture.

— D'Souza & Wills

What then?

D’Souza and Wills (chapter 12) argue that such high-level diagrams are not the essence of architecture. That does not mean that these diagrams are useless, but that architecture is more than just the diagram.

So, what exactly is it?

D’Souza and Wills. *Objects, Components and Frameworks with UML: The Catalysis Approach*, Addison Wesley, 1999.

http://www.catalysis.org/books/ocf/index.htm

What is Software Architecture?

> [Architecture is] the set of ***design decisions*** about any system (or subsystem) that keeps its implementors and maintainers from exercising ***needless creativity***.
>
> — D'Souza & Wills

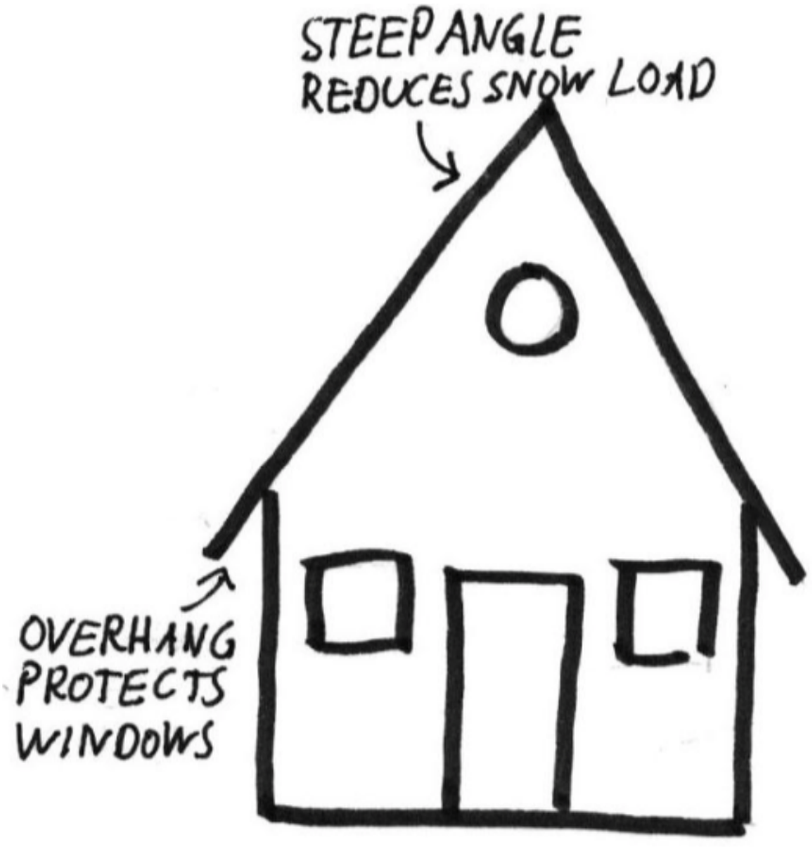
What kind of decisions?

This provocative definition cuts to the chase: *Software Architecture constrains design by fixing certain decisions.*

But what kinds of decisions are important to fix in this way, and how are those decisions made?

Why do we need Software Architecture?

Architectural decisions are those that permit a system to meet its *quality attributes* and *behavioral requirements*.



Well, how are they specified?

Architectural decisions are all about *design constraints intended to guarantee certain global requirements*. Typically these are *non-functional* requirements having to do with performance, scalability, maintenance, and so on.

See chapter 2 of Bass et al:

Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*, Addison Wesley, 3rd edition, 2012.

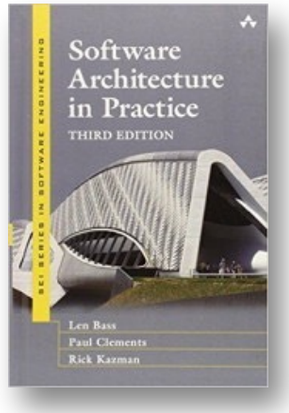
http://www.ece.ubc.ca/~matei/EECE417/BASS/index.html

What is Software Architecture?

The architecture of a system consists of:

1. the *structure(s) of its parts*
   e.g. design-time, test-time, and run-time software and hardware parts
2. the *externally visible properties* of those parts
   e.g. interfaces of modules, hardware units, objects
3. the *relationships and constraints* between them

— *Bass, Clements, Kazman*



This definition of Software Architecture captures most of the key points: it expresses the *coarse structure* of a software system in terms of "*components*". This structure captures *externally visible properties and interfaces* of those components. Furthermore, and most critically, the architecture expresses *constraints* over the configuration of these components.

As we have seen, these constraints are intended to guarantee essential *quality attributes* of the system as a whole.

There are numerous alternative definitions on the website of SEI:

http://www.sei.cmu.edu/architecture/start/glossary/community.cfm

Architecture is a shared mental model

> The architecture is a mental model shared by the stakeholders.
>
> — Holt

Since there are multiple stakeholders of a software system, there are necessarily *multiple viewpoints* of architecture.

Holt, Ric. “Software architecture as a shared mental model.” Proceedings ASERC Workshop on Software Architecture, University of Alberta (2002).

http://plg.uwaterloo.ca/~holt/papers/sw-arch-mental-model-020314c-1.pdf

Architectural Viewpoints

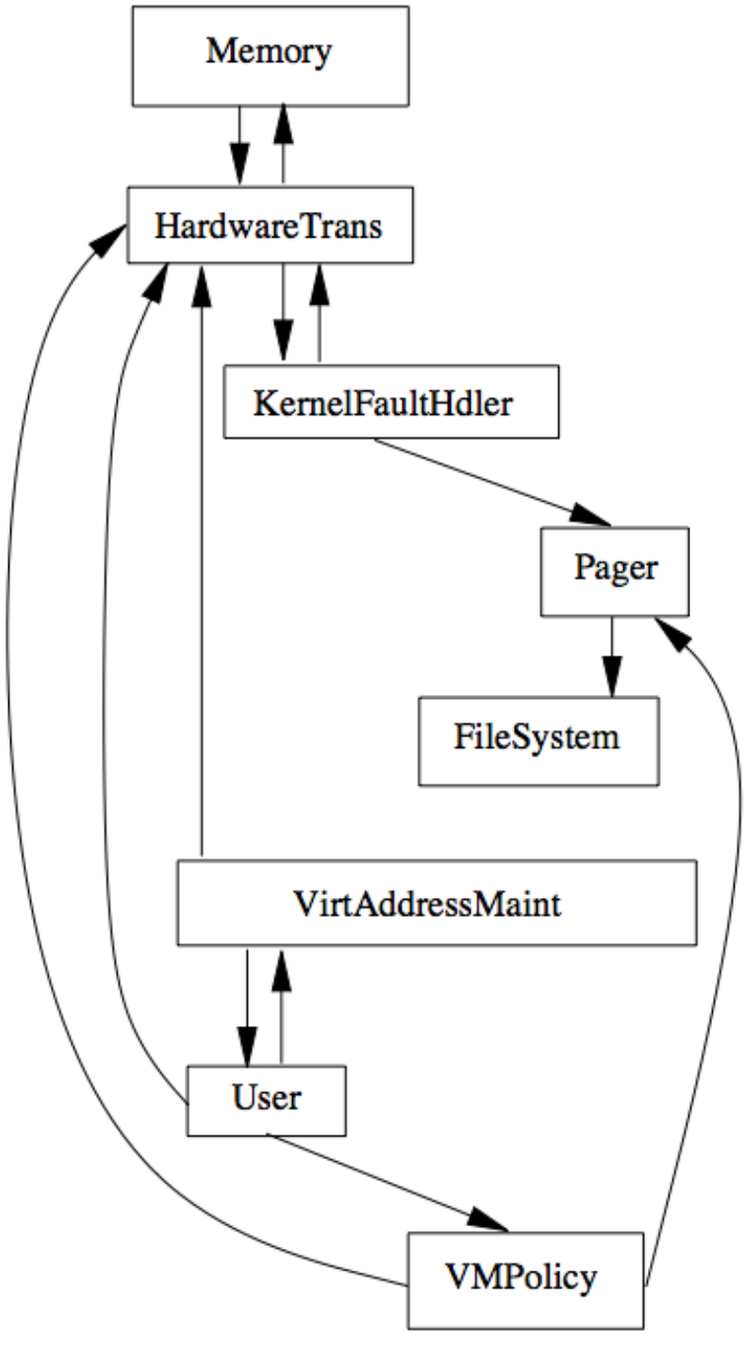
Run-time	How are responsibilities distributed amongst run-time entities?
Process	How do processes communicate and synchronize?
Dataflow	How do data and tasks flow through the system?
Deployment	How are components physically distributed?
Module	How is the software partitioned into modules?
Build	What dependencies exist between modules?

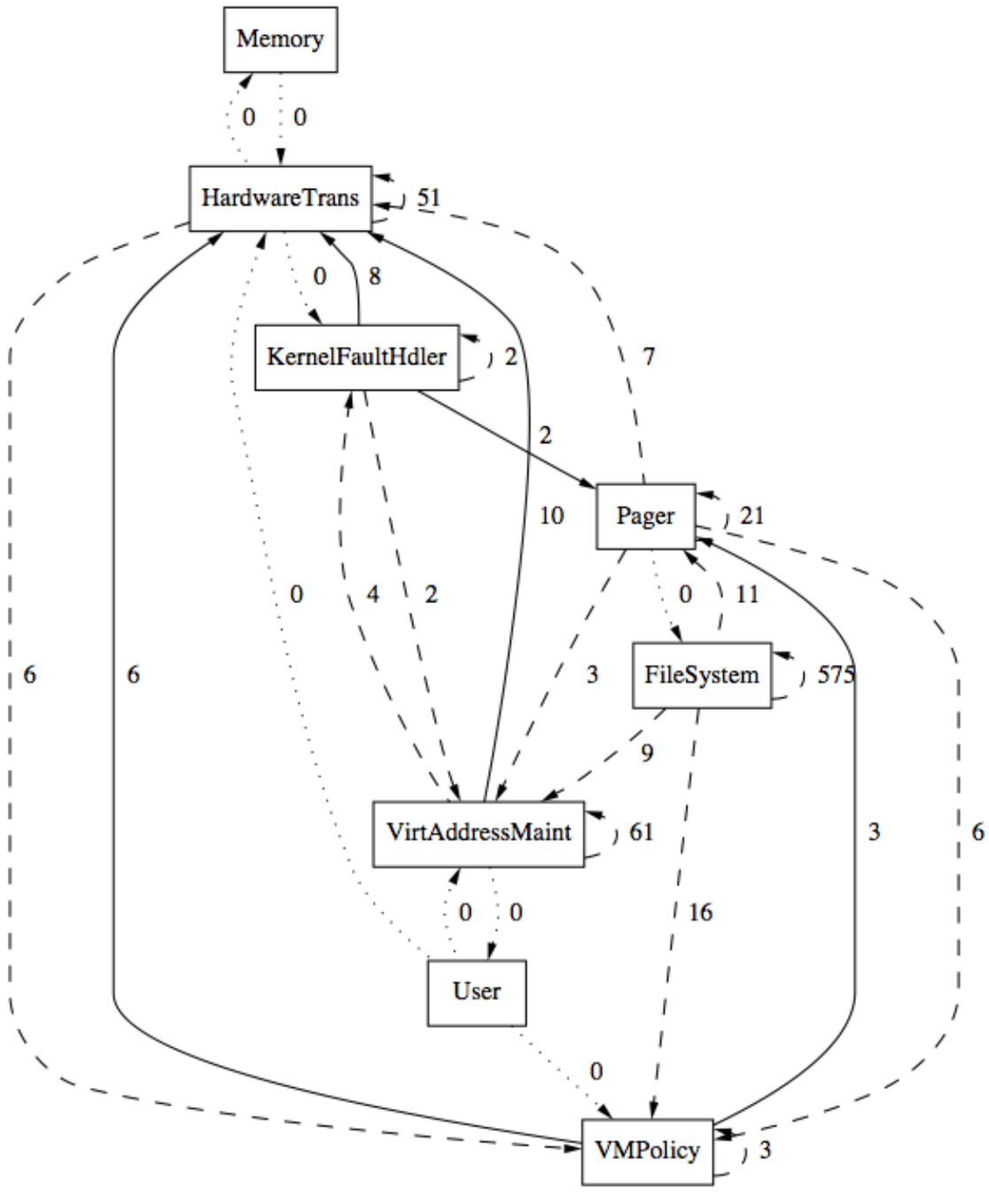
A physical building does not have a single architecture, but rather many architectural viewpoints: physical structure, plumbing, electricity, all view the architecture in different ways, according to different quality concerns.

The same holds for software architecture. The *run-time architecture* is concerned with what happens on a grand scale when the system is running. The *module architecture*, on the other hand, is concerned purely with the static distribution of code into packages. The *build architecture*, although closely related to the module architecture, is strictly concerned with how the system is incrementally compiled and built from the source code.

All of these different viewpoints form part of the software architecture.

Example of Architectural Diagram for a Unix Subsystem



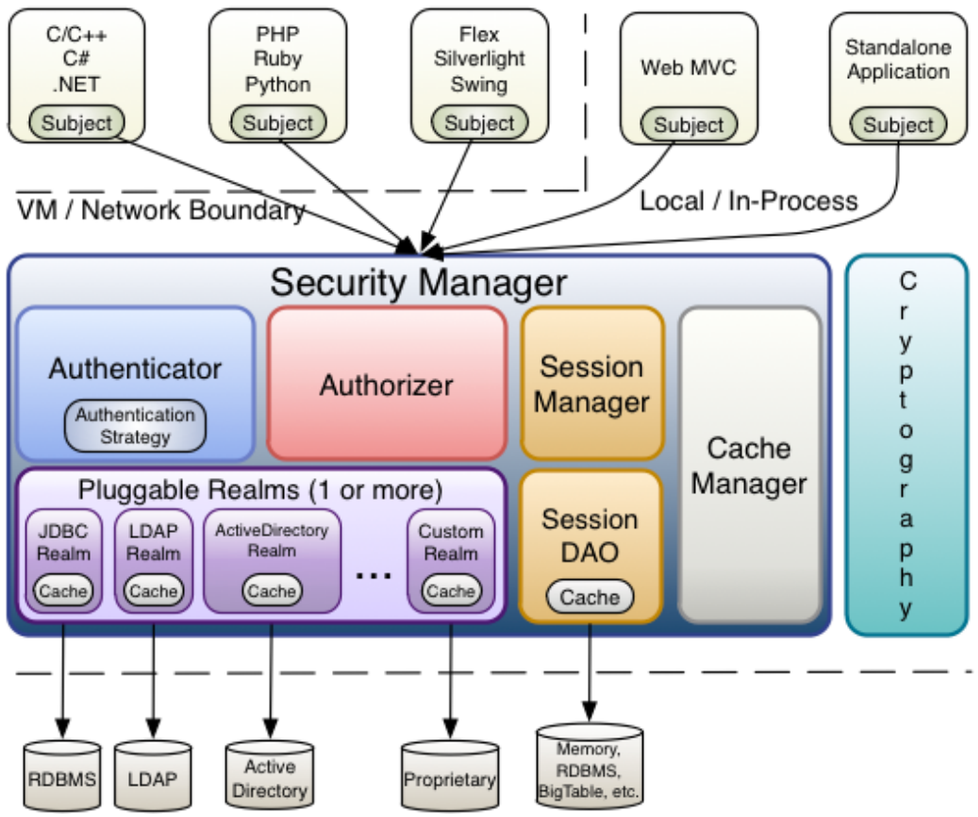


Very often the diagrams that the programmers draw diverge from the actual state of the system.

On the left we see the “ideal” architecture of the dependencies between components of the Unix system, while at the right analysis shows a slightly different picture.

How Architecture Is Usually Specified

"Use a *3-tier client-server architecture*: all business logic must be in the middle tier, presentation and dialogue on the client, and data services on the server; that way you can scale the application server processing independently of persistent store."



2002 Email of Jeff Bezos @ Amazon.com

All teams will henceforth expose their data and functionality through **service interfaces.**
Teams must **communicate exclusively through these interfaces** with each other.
It doesn't matter what technology they use.
There will be **no other form of inter-process communication** allowed: no direct linking, no direct reads of another team's data store, no shared-memory model, no back-doors whatsoever.
Anyone who doesn't do this will be fired.
Thank you; have a nice day!

Bezos imposed a *service-oriented architecture* on Amazon's developers by fiat. This had an enormous business impact on how Amazon evolved over the following years, and also led to Amazon's expansion into offering compute services from its platform.

https://en.wikipedia.org/wiki/Service-oriented_architecture

For an interesting blog discussion, see this archived copy of Stevey's Google Platforms Rant:

https://gist.github.com/chitchcock/1281611

Roadmap

- What is Software Architecture?
- **Cohesion and Coupling**
- Architectural styles
- UML diagrams for architectures

Subsystems, Modules and Components

- A subsystem is a system in its own right whose operation is *independent* of the services provided by other sub-systems.

- A module is a system component that *provides services* to other modules but would not normally be considered as a separate system.

- A component is an *independently deliverable unit* of software that encapsulates its design and implementation and offers interfaces to the out-side, by which it may be composed with other components to form a larger whole.

“Component” is a general term that can be applied to small or large scale entities. Modules and subsystems can both be considered “components”.

A subsystem can often be swapped out in its entirety, for example, a graphics subsystem, a database subsystem, or a communications subsystem may be replaced (at some cost) by a different one.

A module is also often called a “package”.

https://en.wikipedia.org/wiki/Modular_programming

Cohesion

Cohesion is a measure of *how well the parts of a component "belong together".*

> Cohesion is weak if elements are bundled simply because they perform similar or related functions (e.g., `java.lang.Math`).

> Cohesion is strong if all parts are needed for the functioning of other parts (e.g. `java.lang.String`).

—Strong cohesion *promotes maintainability* and adaptability by *limiting the scope of changes* to small numbers of components.

There are many definitions and interpretations of cohesion.
Most attempts to formally define it are inadequate!

Software architectures are designed to *maximize cohesion* and *minimize coupling*.

"Cohesion" means that *things that belong together are found together*. If cohesion is weak, then whenever you need to make a change to the system, you will have to touch many parts of the system. With strong cohesion, related functionality is all in one place.

Responsibility-driven design promotes strong coupling by organizing design around responsibilities.

Although various software metrics have been defined to formalize cohesion, for example in terms of operations that work with common data, unfortunately there always exist counterexamples whose metric values contradict their perceived cohesion.

(For example, `java.lang.Math` is conceptually cohesive, though the static methods it defines mostly share no common data.)

Coupling

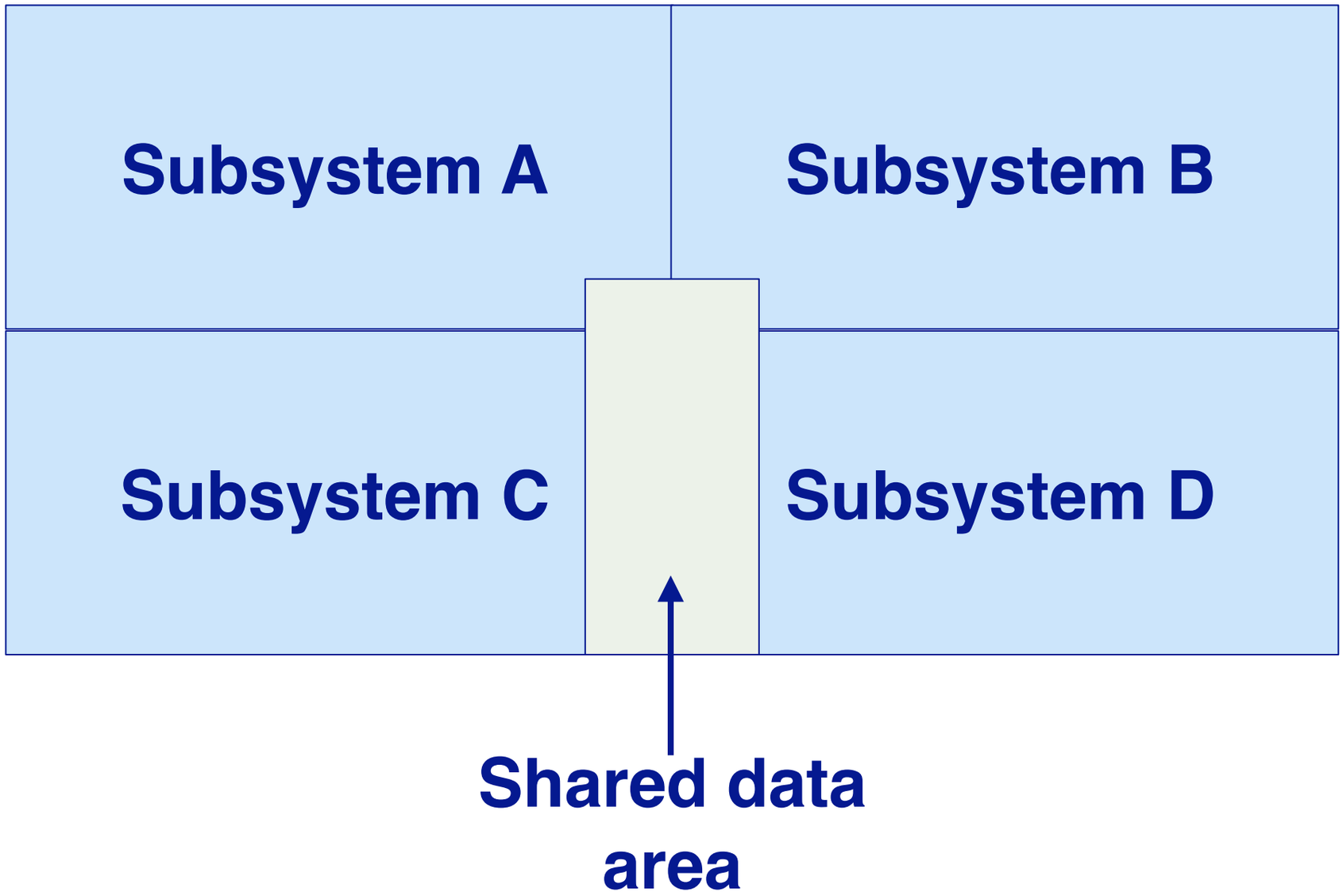
Coupling is a measure of the *strength of the interconnections* between system components.

- Coupling is tight between components if they depend heavily on one another, (e.g., there is a lot of communication between them).
- Coupling is loose if there are few dependencies between components.
  - —Loose coupling *promotes maintainability* and adaptability since *changes in one component are less likely to affect others.*
  - —*Loose coupling increases the chances of reusability.*

Software architecture also seeks to *minimize coupling*.

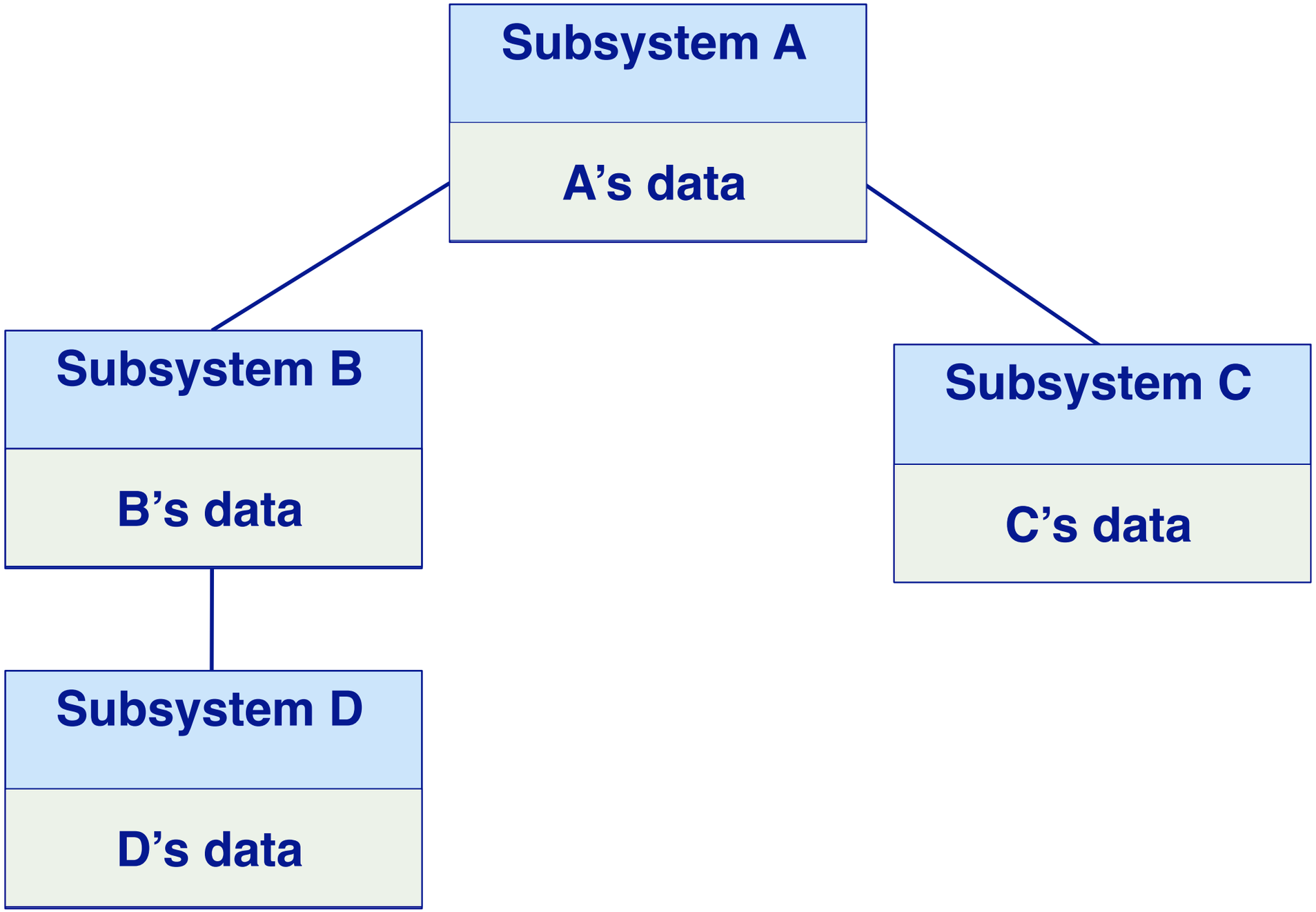
A software system with strong coupling is *fragile to change*: whenever you touch a component, you risk breaking the other components it is coupled with. By minimizing coupling you reduce the risk that a change will break something.

Tight Coupling



Here 4 subsystems are all coupled to a common shared data area. Any change to the data representation risks to impact all of the subsystems.

Loose Coupling



In a more object-oriented design, each subsystem is responsible for its own data. Subsystems talk to each other only through a defined API. Changes to the data representation only affect a single subsystem. Changes to a subsystem's API affect only client subsystems.

Roadmap

- What is Software Architecture?
- Cohesion and Coupling
- **Architectural styles**
  - Layered
  - Client-Server
  - Repository, Event-driven, Dataflow, ...
- UML diagrams for architectures

Architectural Styles

Architectural Styles in Software

> An architectural style defines a *family of systems* in terms of a *pattern of structural organization*. More specifically, an architectural style defines a *vocabulary of components and connector types*, and a *set of constraints* on how they can be combined.
>
> — Shaw and Garlan

In other words, an architectural style defines *what kinds of components* exist in a software system, and *how these components interact*.

The *reason* for the architecture is to *guarantee some desirable properties* (e.g., low coupling, high cohesion) that will *impact some quality attributes* of the system (e.g., maintainability, scalability, performance, build time etc.)

Roadmap

- What is Software Architecture?
- Cohesion and Coupling
- Architectural styles
  - **Layered**
  - Client-Server
  - Repository, Event-driven, Dataflow, ...
- UML diagrams for architectures

Layered Architectures

A layered architecture organises a system into a set of layers each of which provide a set of services to the layer "above".

> Normally layers are *constrained* so elements only see
  —other elements in the same layer, or
  —elements of the layer below

> *Callbacks* may be used to communicate to higher layers

> Supports the *incremental development* of sub-systems in different layers.
  —When a layer interface changes, *only the adjacent layer is affected*

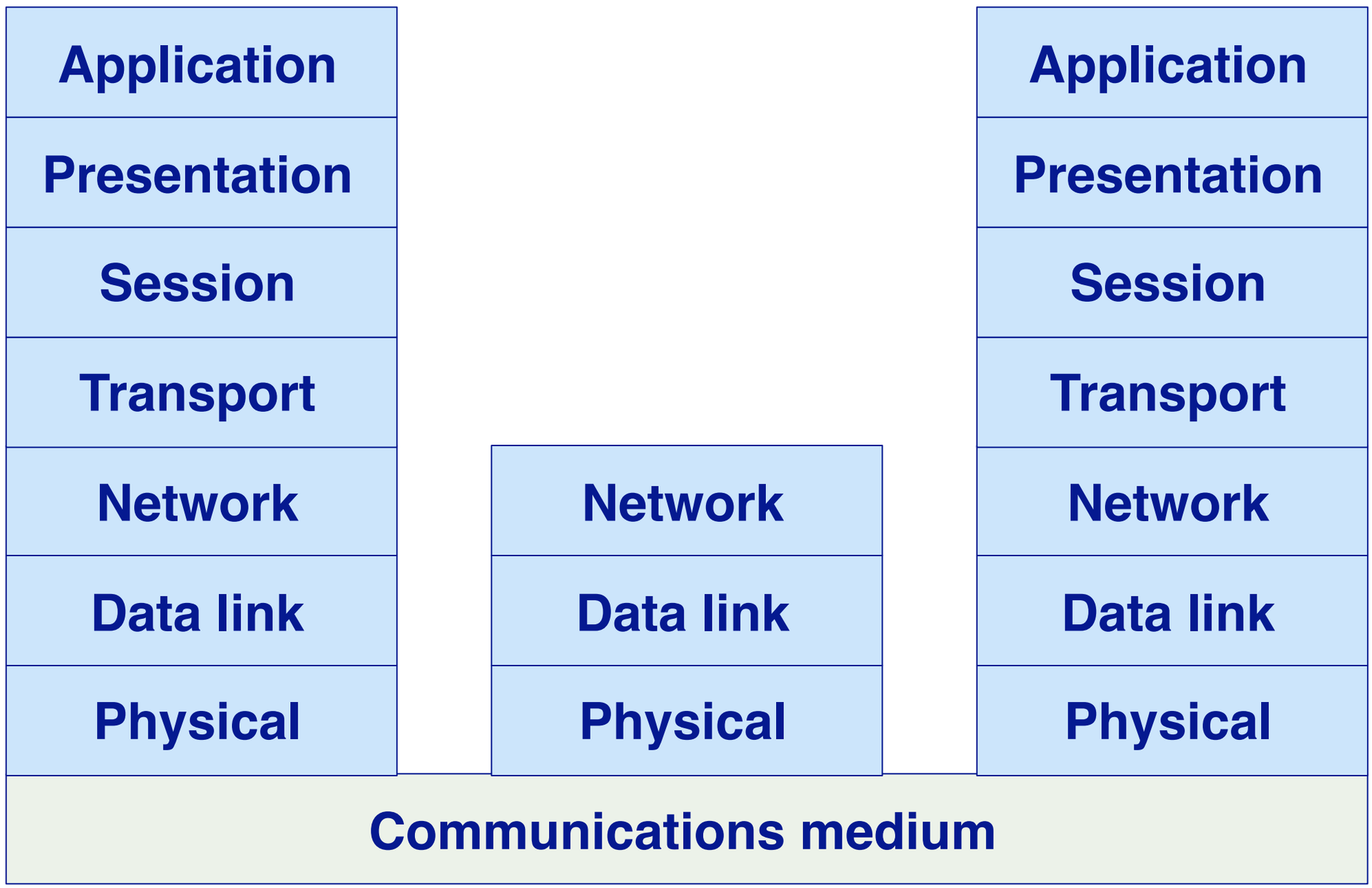
The *components* of this style are *layers*.

The *constraint* is that *each layer only sees the layer immediately below it*.

The desirable *property* is that *changes to the interface of a layer will only impact the layer above*.

One affected *quality attribute* is *maintainability*: it becomes cheap to replace or reuse an entire layer. Changes to a layer have limited impact. (What other quality attributes might be affected? Scalability? performance? …)

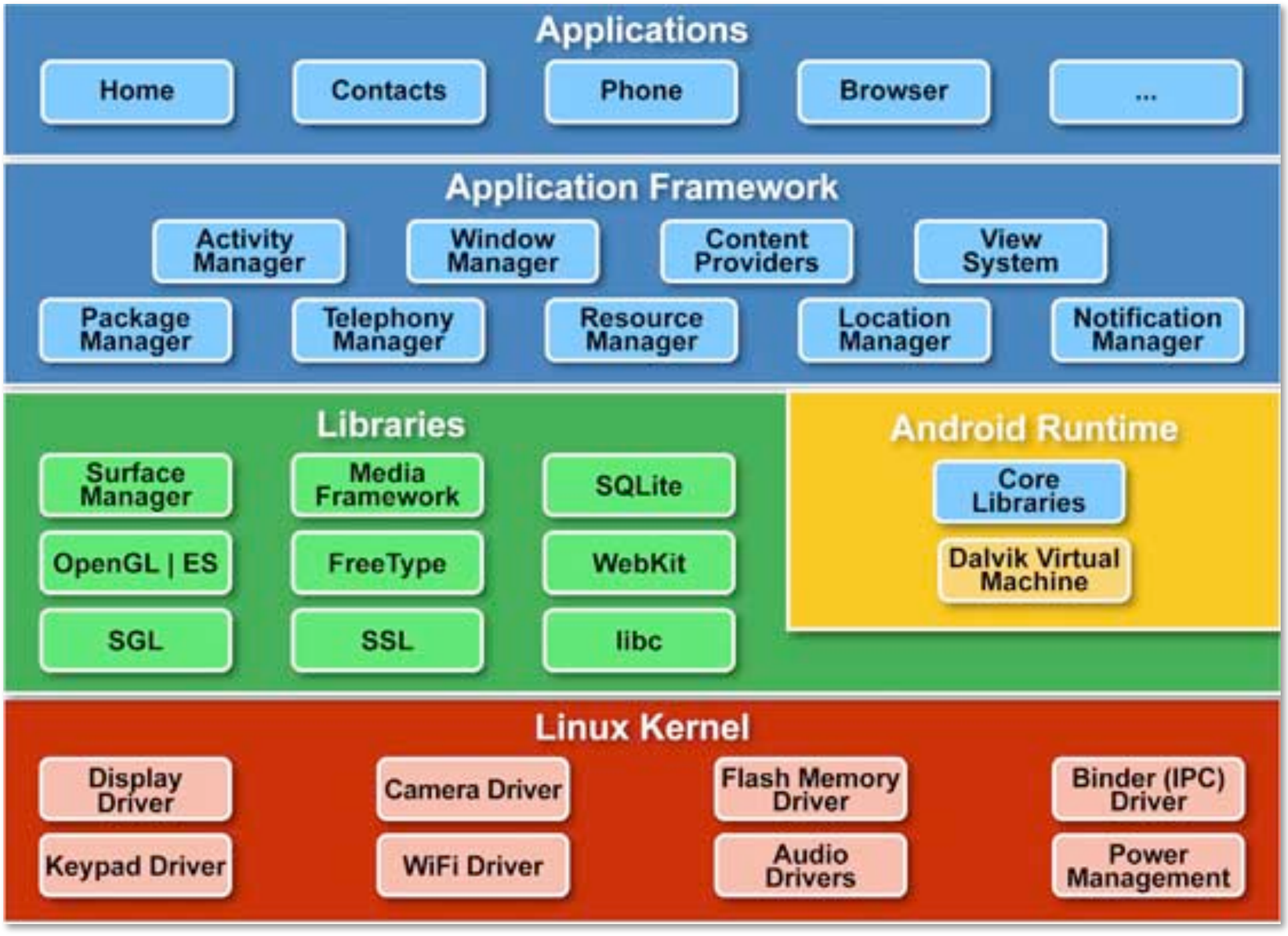
OSI reference model



The most famous example is the TCP/IP protocol stack. Each layer is implemented in terms of the layer below.

Importance of layers: imagine if you had to worry about the shape of the signals that travel on the wire... or even about the error correction... or even about routing to the desired router.

The Android Architecture



Unix and its derivatives owe their great success to a layered architecture. A very small kernel is the only part of the O/S that talks to hardware. To port the O/S to a new piece of hardware, in principle only the kernel needs to be ported, i.e., the lowest level device drivers. All layers above can be migrated “for free”.

Roadmap

- What is Software Architecture?
- Cohesion and Coupling
- Architectural styles
  - Layered
  - **Client-Server**
  - Repository, Event-driven, Dataflow, ...
- UML diagrams for architectures

Client-Server Architectures

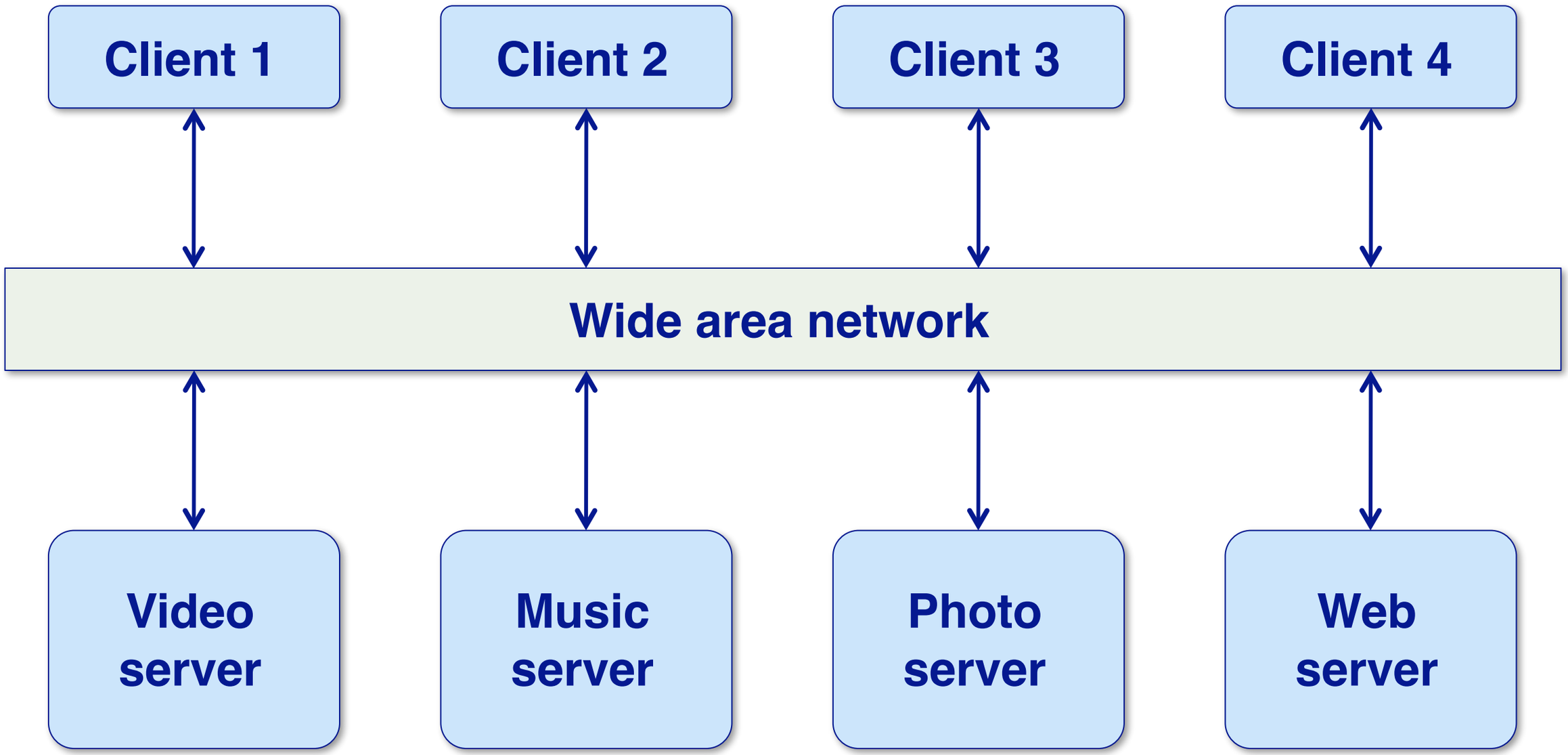
A client-server architecture *distributes application logic and services* respectively to a number of client and server sub-systems, each potentially running on a different machine and communicating through the network (e.g, by RPC).

The *components* here are the clients and the servers. The *constraints* are that clients are responsible for interaction with users, servers are responsible for services, and clients make requests to servers to get obtain these services.

The nice *properties* are low coupling (between client and server) and high cohesion (each has its own responsibilities). This is good for maintenance, but can also impact scalability and performance.

https://en.wikipedia.org/wiki/Client–server_model

Film and picture library



Client-Server Architectures

Advantages

- *Distribution* of data is straightforward
- Makes *effective use of networked systems*. May require cheaper hardware
- Easy to *add new servers* or upgrade existing servers

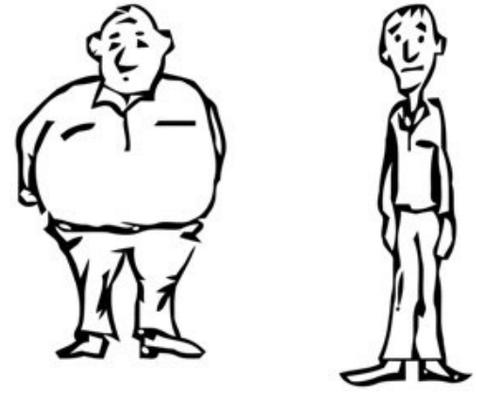
Disadvantages

- *No shared data model* so sub-systems use different data organisation. Data interchange may be inefficient
- *Redundant management* in each server
- May require a *central registry* of names and services — it may be hard to find out what servers and services are available

Fat client / thin client

- In a *fat client* architecture, the server only provides computational services and the client takes care of the rest (UI, input validation, presentation updates)
  - good for responsiveness; places more constraints on users

- In a *thin client* architecture the client only acts as a presentation engine, and the server does everything else
  - good for low-cost clients; impacts latency



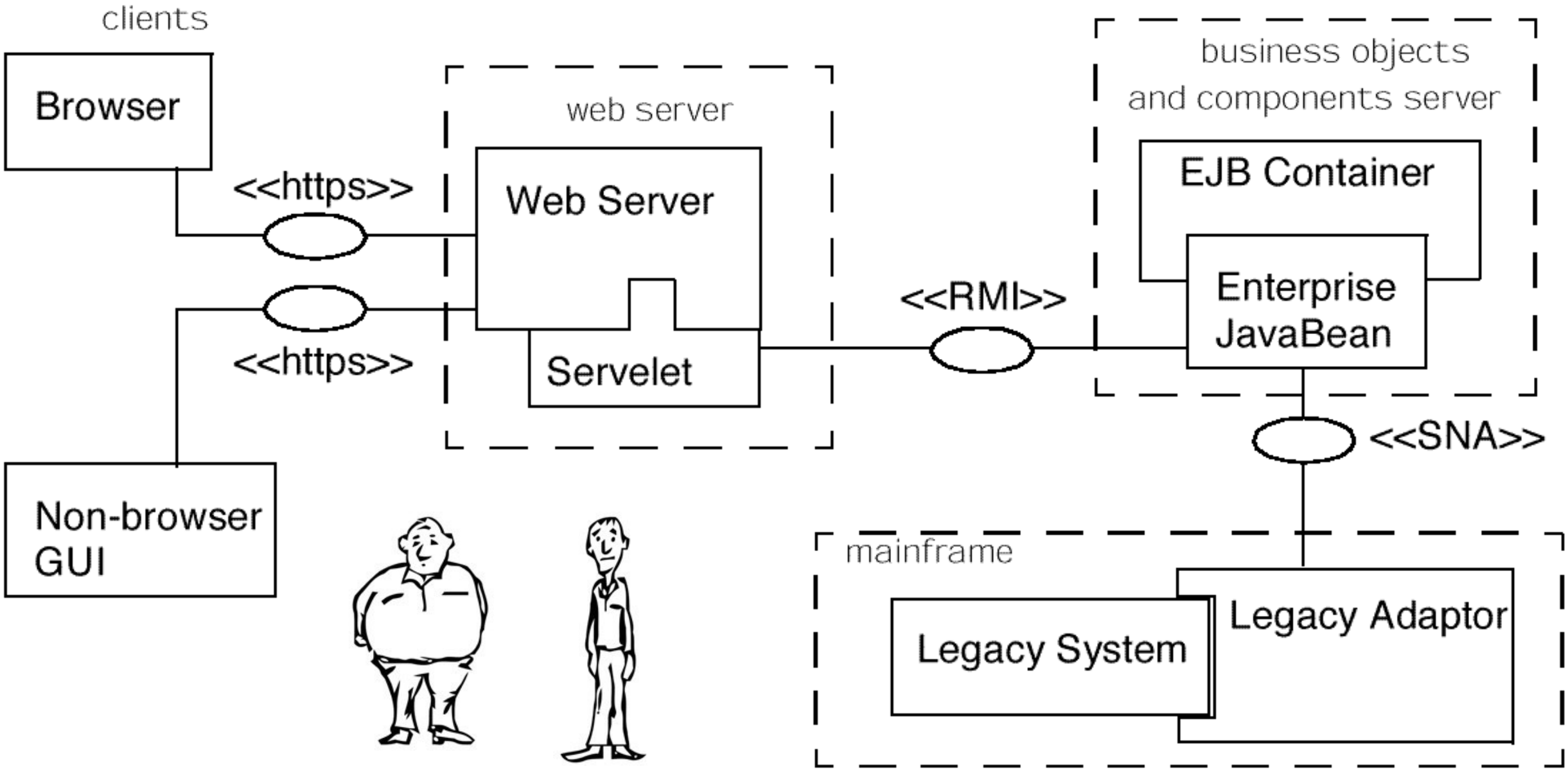
Early (Web 1.0) web applications are thin clients: the client is just a web browser, and every single click requires the server to generate a new web page.

Web 2.0 applications are essentially fat clients that make heavy use of JavaScript to do much more work in the browser. As a result many requests can be fulfilled without contacting the server at all. This decreases the load on the server and gives the end-user a more responsive experience. On the other hand, it places a greater burden on the client, assuming the existence of a modern web browser.

https://en.wikipedia.org/wiki/Fat_client

https://en.wikipedia.org/wiki/Thin_client

Multi-tier Architectures



This is a well-established, generic architectural style for business application.

A four-tier architecture distinguishes:

1. the client GUI (fat or thin)
2. the web service
3. business objects providing business logic
4. a mainframe legacy application or database system

https://en.wikipedia.org/wiki/Multitier_architecture

Service-Oriented Architectures (SOA)

- The *extreme generalization* of Client-Server
- Instead of monolithic systems one has many concise services
- A Service is a "*loosely coupled, reusable software component, which can be distributed*"
- Services use message-based communication
- Service discovery becomes a challenge

The latest trend in SOA is “Microservices”, fine-grained services that maximize cohesion and minimize coupling to an extreme.

https://en.wikipedia.org/wiki/Service-oriented_architecture

https://en.wikipedia.org/wiki/Microservices

RESTful Architectures

- Inspired from the architecture of the largest distributed application ever: the Web
  - Stateless requests
  - Every resource has an individual URI
  - Uniform interface for all resources (GET, POST, PUT, DELETE)
- The structure of a response is not specified

By being stateless, REST improves performance, scalability, portability and reliability:

https://en.wikipedia.org/wiki/Representational_state_transfer

Roadmap

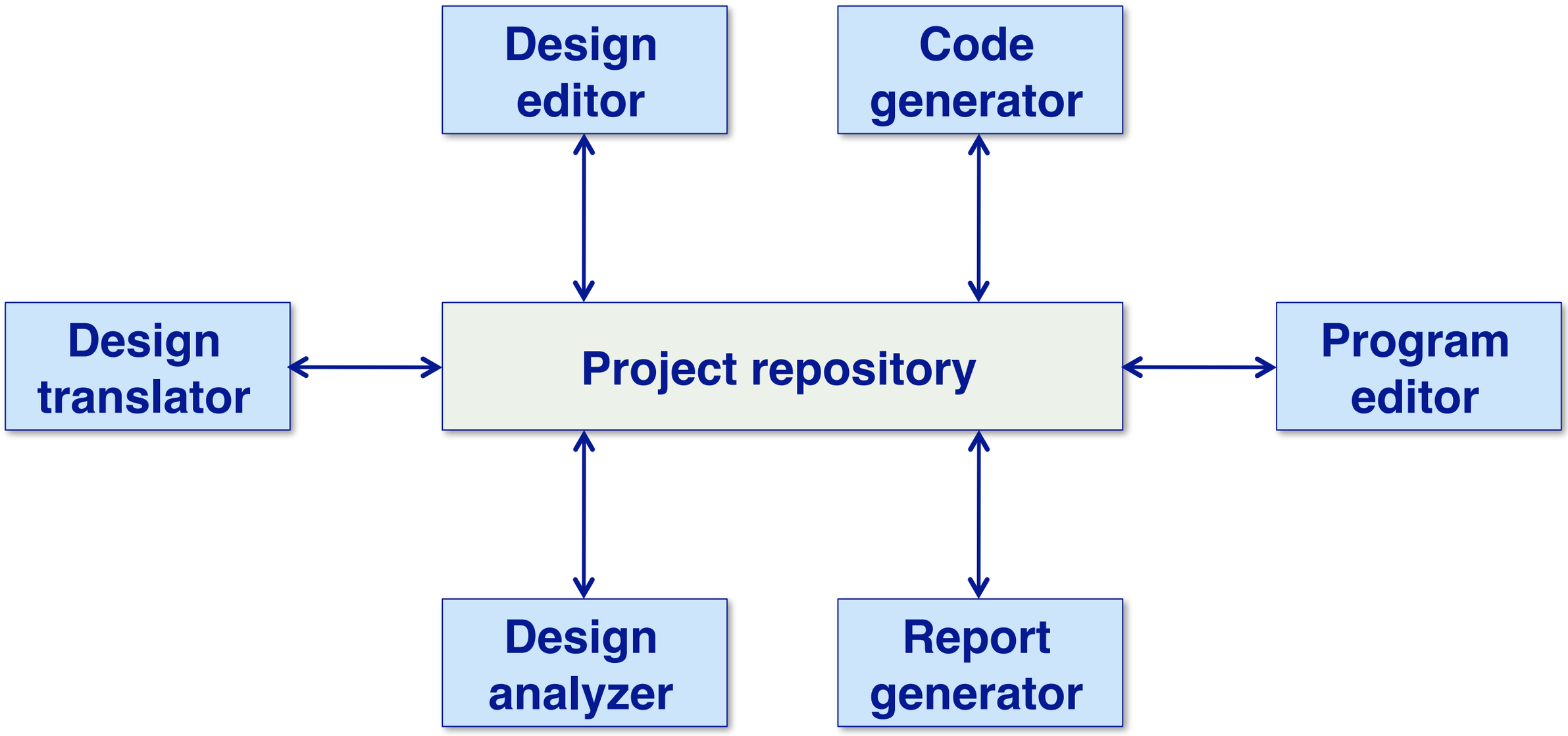
- What is Software Architecture?
- Cohesion and Coupling
- Architectural styles
  - —Layered
  - —Client-Server
  - —**Repository, Event-driven, Dataflow, ...**
- UML diagrams for architectures

Repository Architectures

A repository architecture distributes application logic to a number of independent sub-systems, but *manages all data in a single, shared repository*.

This classical pattern is the grandfather of storing data in the cloud. Your google docs is an instance of a repository architecture. Your docs are always in the cloud.

IDE architecture



When looking at the picture, think Eclipse. Notice that cohesion is strong, as each subsystem has its own clear responsibilities. However the repository is strongly coupled to all other subsystems. Sharing of data is maximized, but the repository format can only be changed at the risk of impacting all other components.

Repository Architectures

Advantages

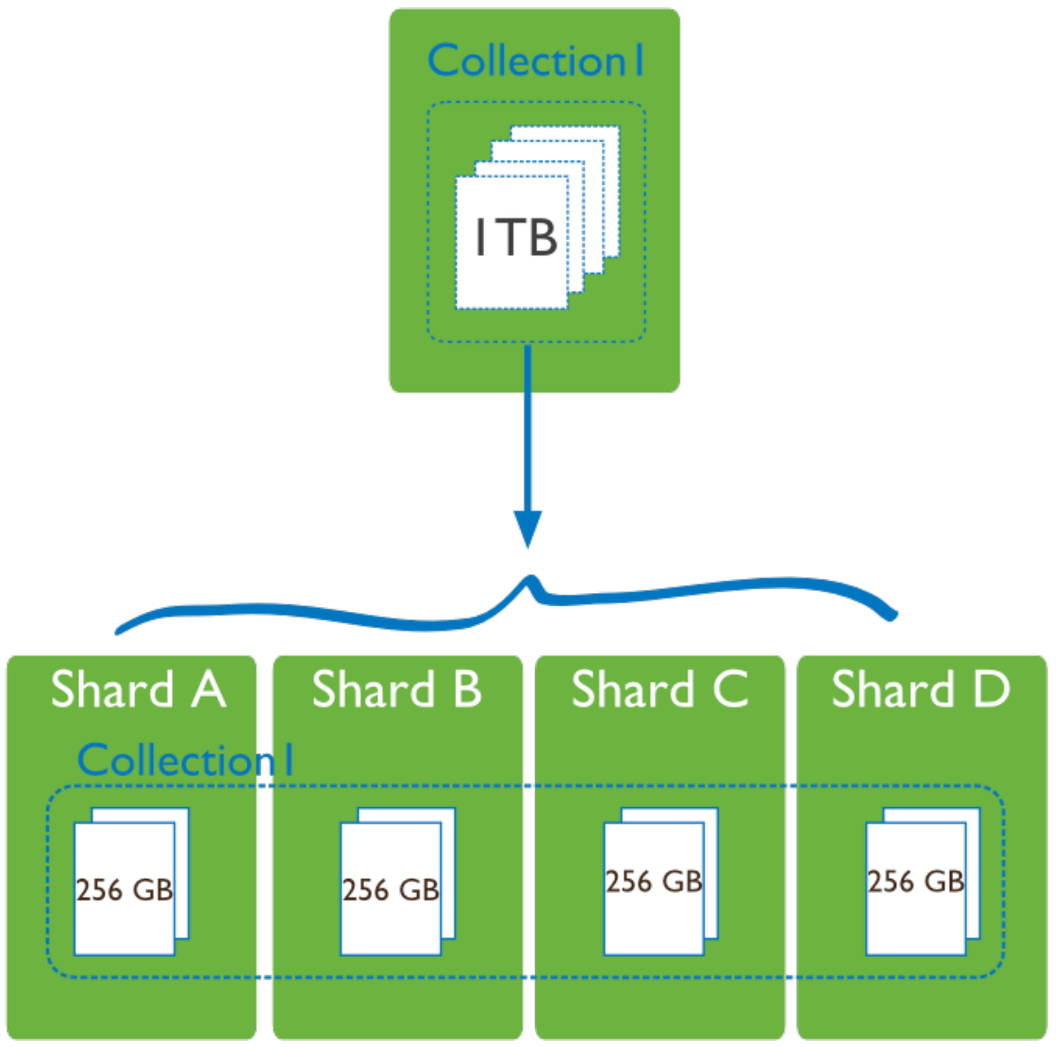
- *Efficient way to share* large amounts of data
- Sub-systems need not be concerned with how data is produced, backed up etc.
- Sharing model is published as the *repository schema*

Disadvantages

- Sub-systems must agree on a repository data model
- *Data evolution* is difficult and expensive (unless NoSQL)
- Repository can become performance bottleneck

Sharding

- A method of storing data across multiple machines
  - reduces processing needs
  - reduces storage needs
- Queries must be routed to the corresponding shards



Event-driven Systems

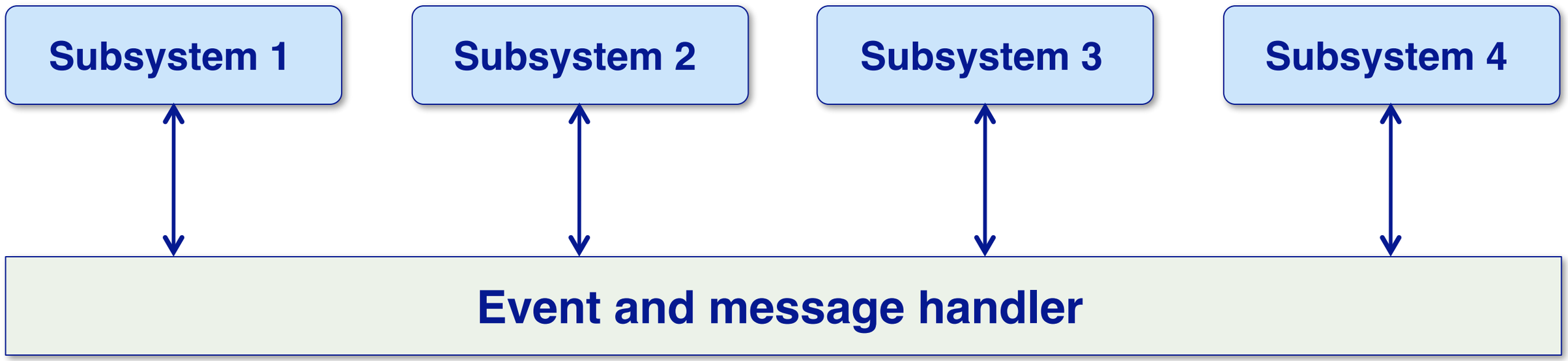
In an event-driven architecture components perform services in *reaction to external events* generated by other components.

- In broadcast models an event is broadcast to all sub-systems. Any sub-system which can handle the event may do so.
- In interrupt-driven models real-time interrupts are detected by an interrupt handler and passed to some other component for processing.

Such event-driven systems scale well due to extremely low coupling between components.

This architectural style will become increasingly relevant with the advent of multi-core machines due to asynchronous communication (events are distributed without requiring a call and response).

Broadcasting



Broadcast model

- Effective in *integrating sub-systems* on different computers in a network
- Can be implemented using a *publisher-subscriber* pattern:
  - —Sub-systems register an interest in specific events
  - —When these occur, control is transferred to the subscribed sub-systems
- However, sub-systems don’t know if or when an event will be handled

Dataflow Models

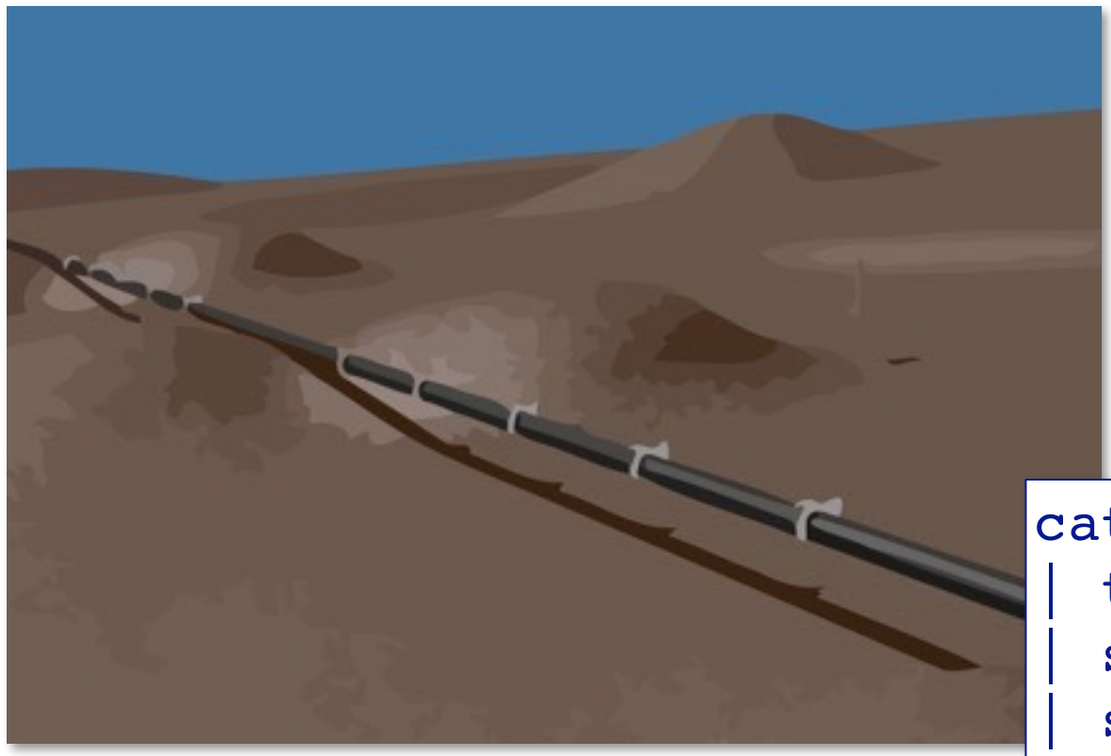
In a dataflow architecture each component performs *functional transformations* on its inputs to produce outputs.

- Highly effective for *reducing latency* in parallel or distributed systems
  - —No call/reply overhead
  - —But, fast processes must wait for slower ones

- Not really suitable for *interactive systems*
  - —Dataflows should be free of cycles

Pipes and Filters

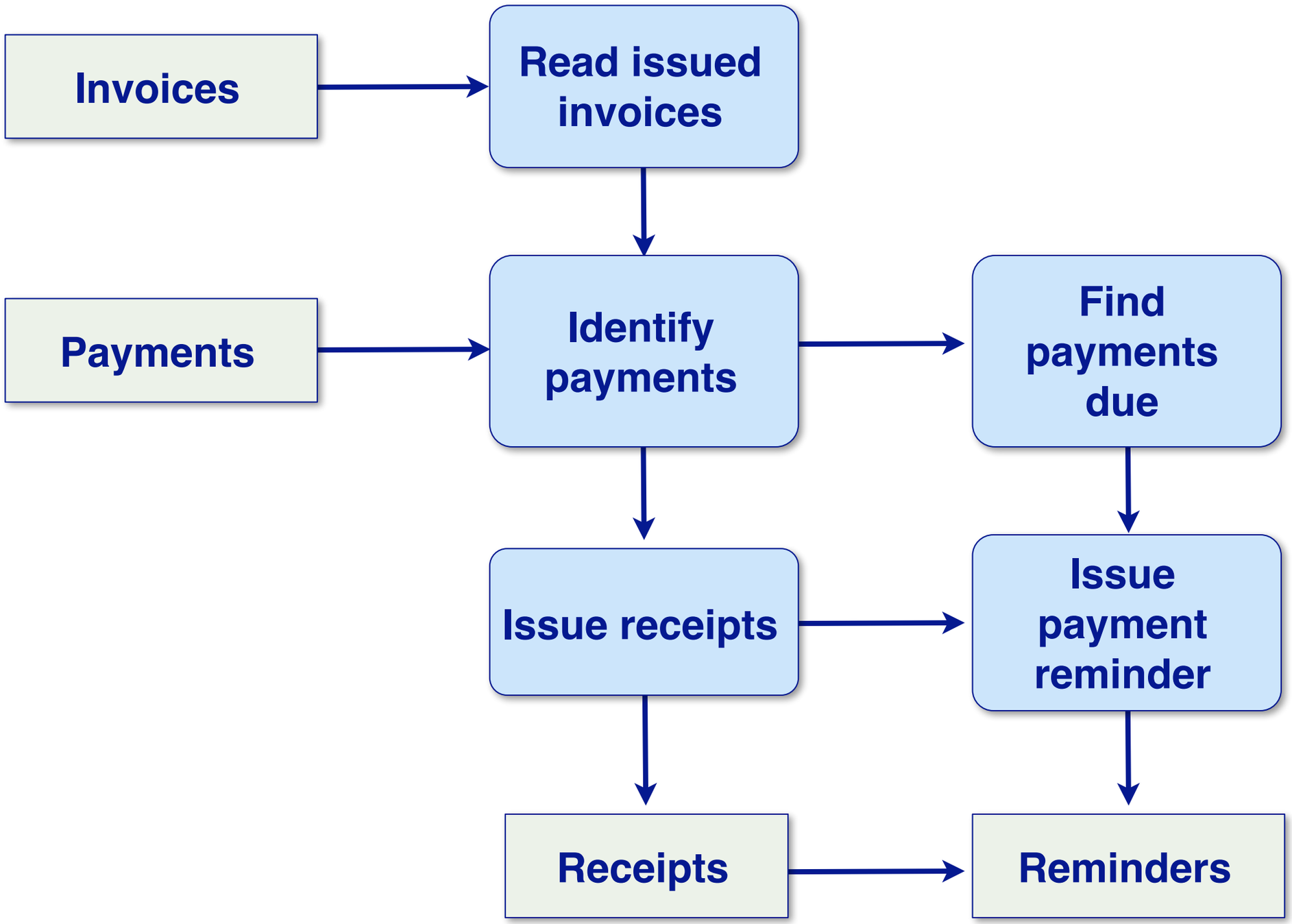
Domain	*Data source*	*Filter*	*Data sink*
Unix	`tar cf - .`	`gzip -9`	`rsh picasso dd`
CGI	HTML Form	CGI Script	generated HTML page



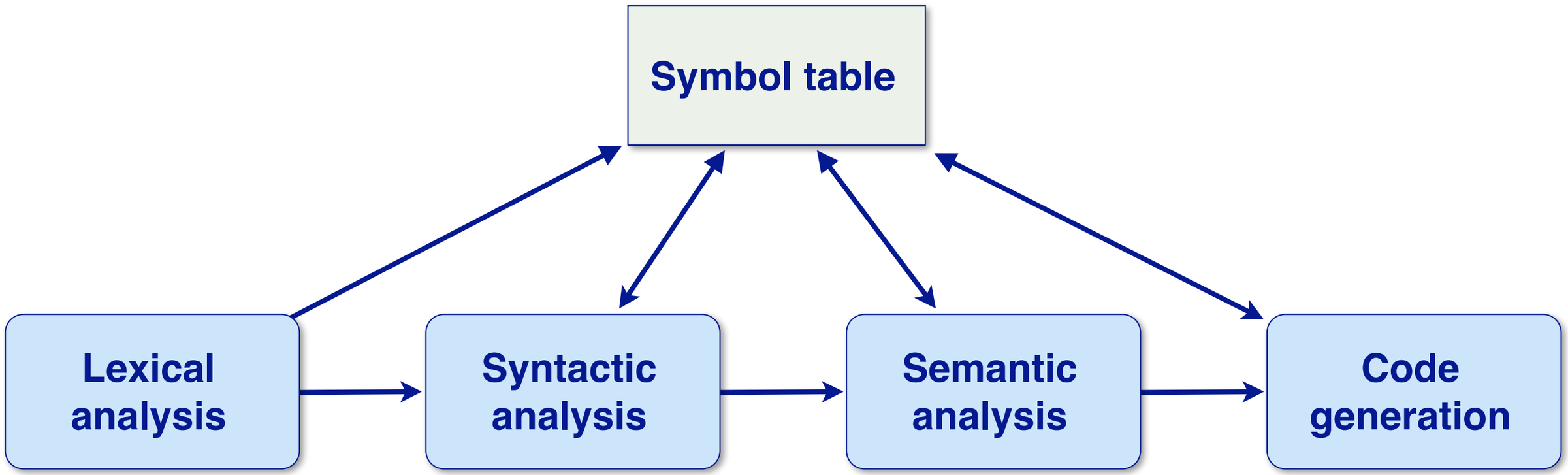
```
cat Notes.txt
| tr -c '[:alpha:]' '\012'
| sed '/^$/d'
| sort
| uniq –c
| sort –rn
| head -5
```

```
14 programming
14 languages
 9 of
 7 for
 5 the
```

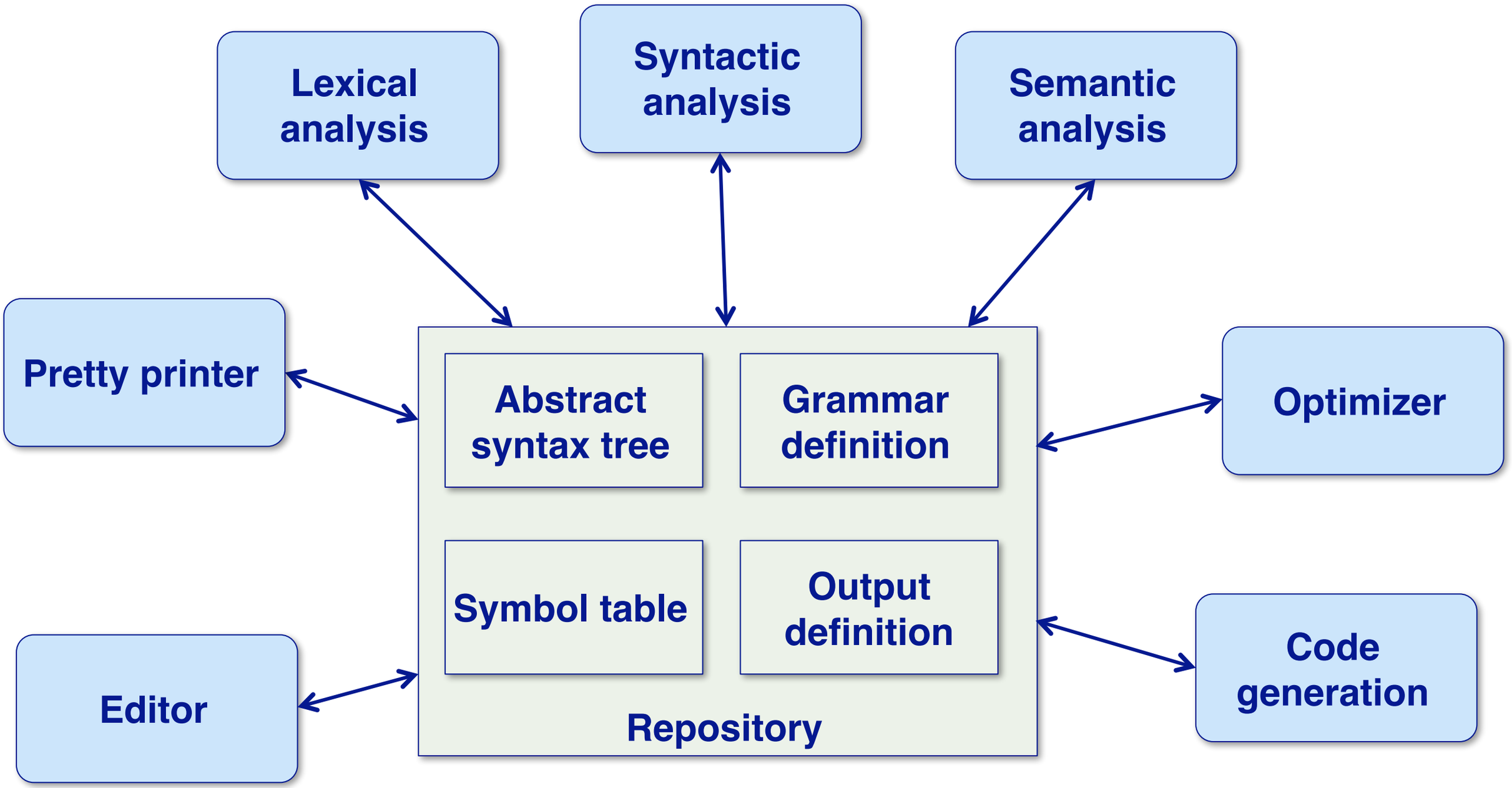
Invoice Processing System



Compilers as Dataflow Architectures



Compilers as Repository Architectures



Roadmap

- What is Software Architecture?
- Cohesion and Coupling
- Architectural styles
- **UML diagrams for architectures**

UML support: Package Diagram

Decompose system into *packages* (containing any other UML element, incl. packages)

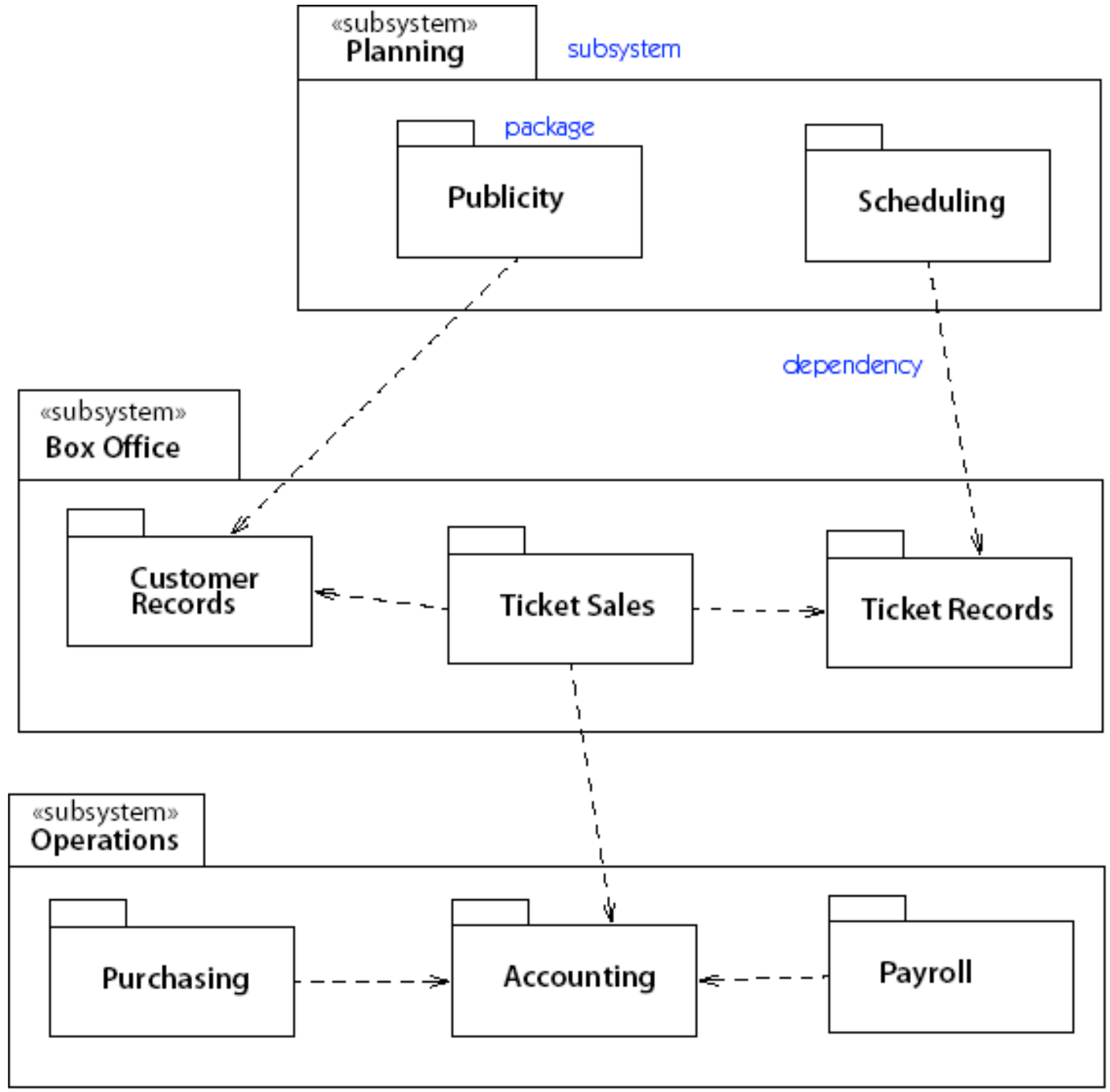


Figure 3-10. *Packages*

UML support: Deployment Diagram

Physical layout of run-time components on hardware nodes.

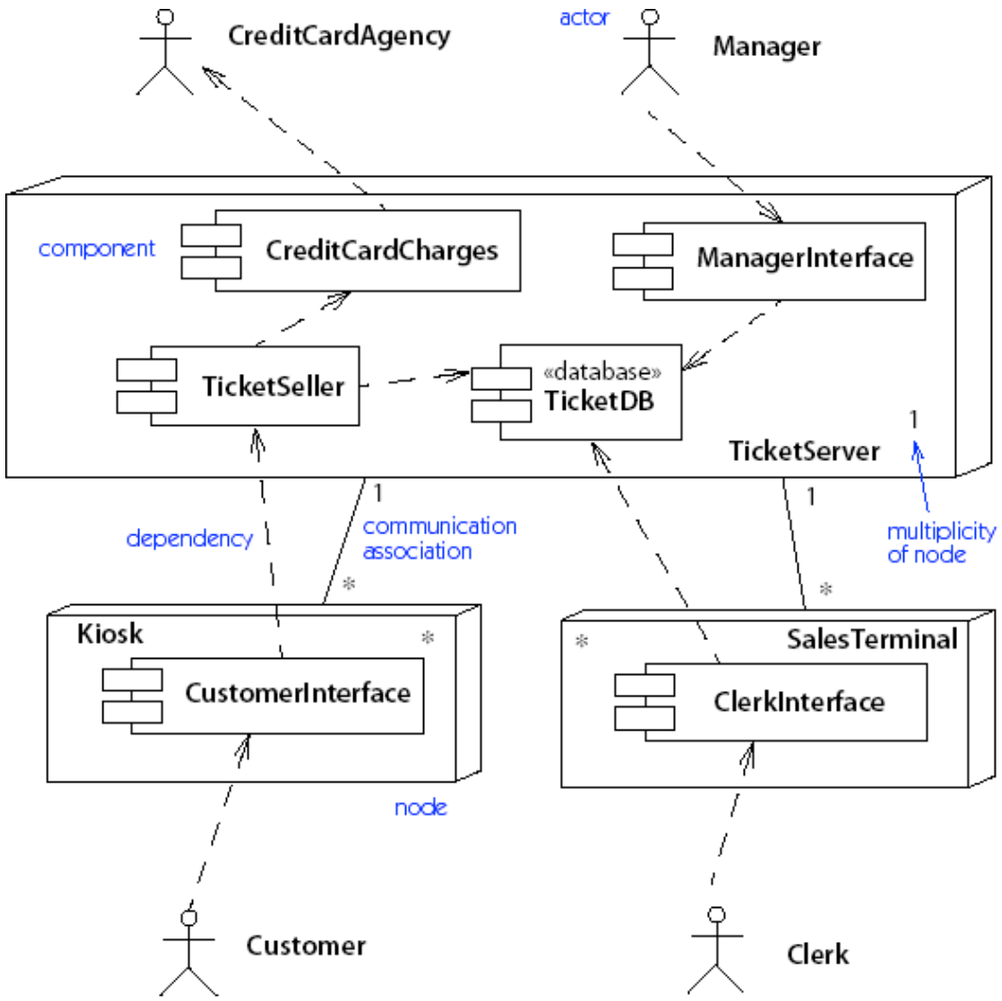


Figure 3-8. *Deployment diagram (descriptor level)*

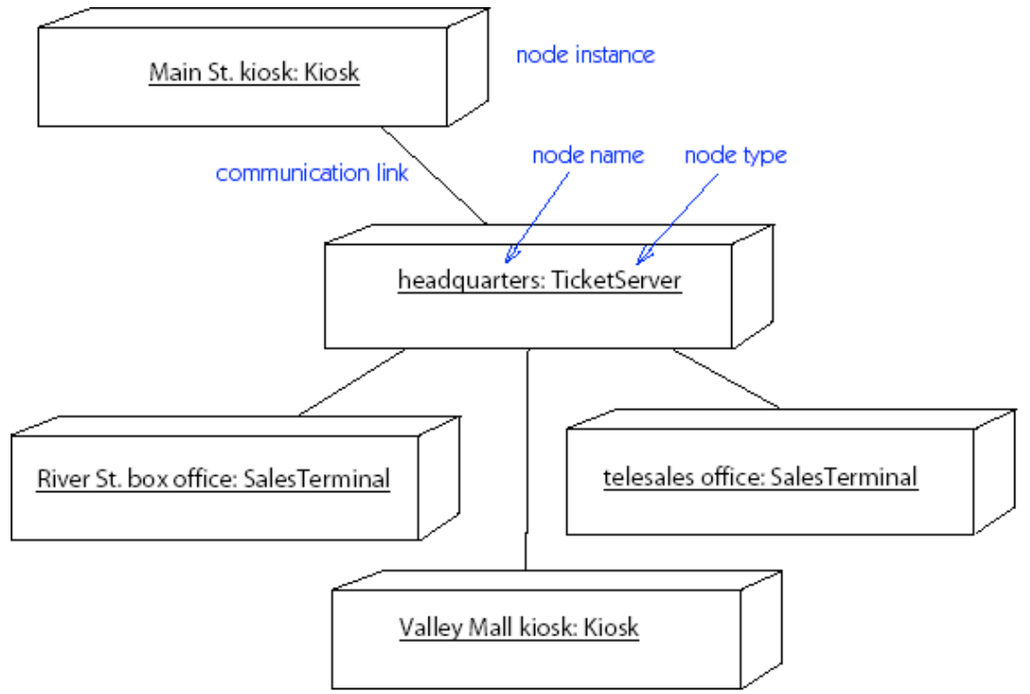


Figure 3-9. *Deployment diagram (instance level)*

What you should know!

- What is software architecture?
- How does software architecture constrain a system?
- How does choosing an architecture simplify design?
- What are architectural viewpoints and architectural styles?
- What are coupling and cohesion?
- What are advantages and disadvantages of classical architectural styles?
- Why shouldn’t elements in a software layer “see” the layer above?

Can you answer the following questions?

- What is meant by a “fat client” or a “thin client” in a multitier architecture?
- What kind of architectural styles are supported by the Java AWT?
- How do callbacks reduce coupling between software layers?
- How would you implement a dataflow architecture in Java?
- What are the coupling and cohesion characteristics of each architectural style?