Structuring the System

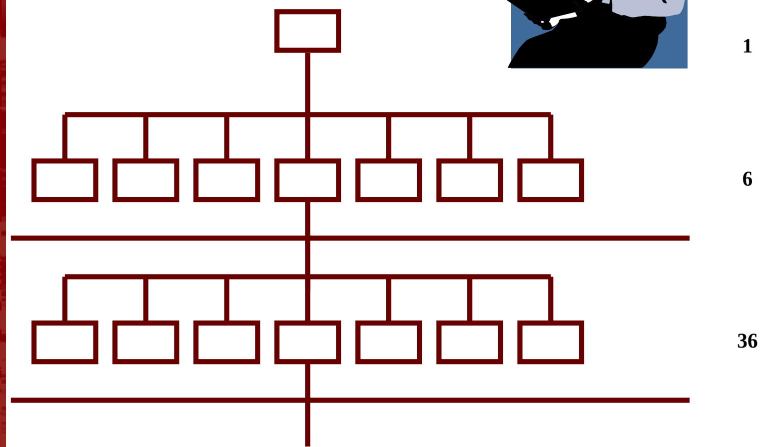
A System Consisting of 100.000 - 1000.000 Lines of Code:

- How do you create a good design?
- How do you understand the system?
- How do you divide responsibilities between people?
- How do you pinpoint faults?
- How do you find reusable parts?
- How do you make parts replaceable?
- …..

Structuring:

Reduce complexity of a unit by identifying and relating understandable parts

Understandability: 5-7 Principle



In Practice: 5-20 Principle

- Bottom level: File, class… ~ 1.000 LoC
- Level 2: ~ 10.000 LoC
- Level 3: ~ 100.000 LoC
- Level 4: ~ 1000.000 LoC

Architectural Style

- Choice of paradigm
- Choice of architectural patterns
- Choice of vocabulary
- Design rules
- Prescribed and recommended design patterns
- Naming conventions
- ….

Architecture

- Decisions that affect several or all parts of the system.
- High level structures
- Choice of platform
- Choice of development environment
- ….

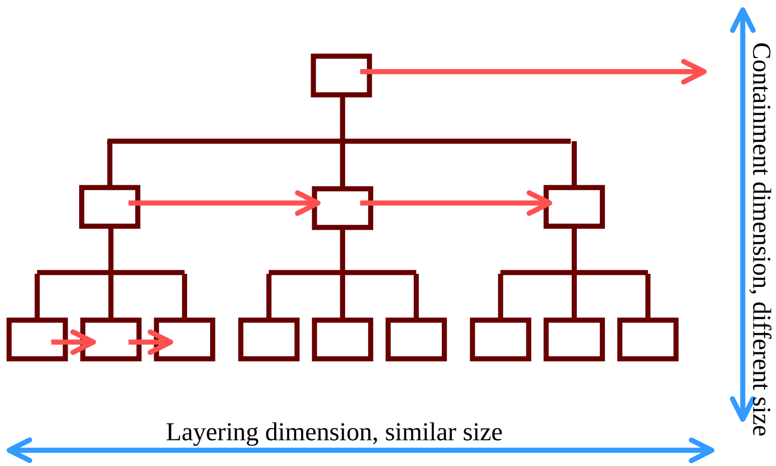
Main Structuring Principles:

- Hierarchy
- Layering

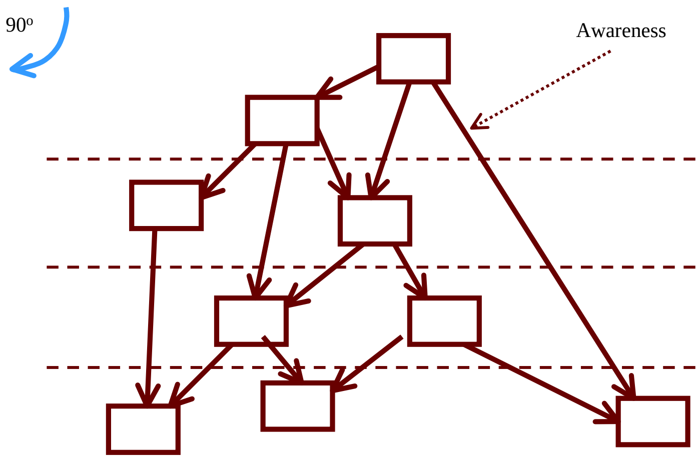
Layering

- Improves understanding
- Helps planning
- Simplifies process
- Promotes reuse

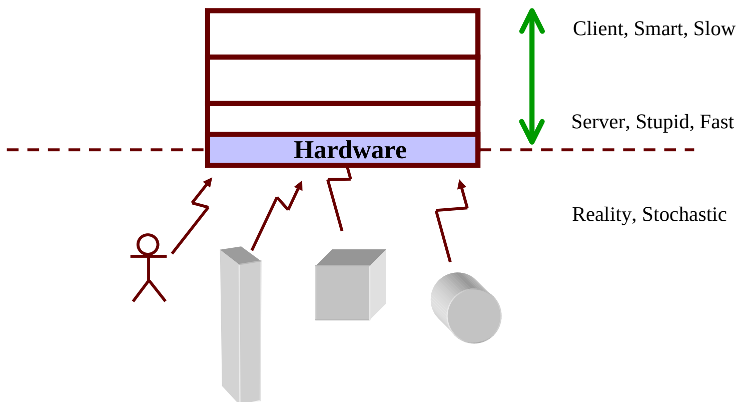
Hierarchy vs Layering



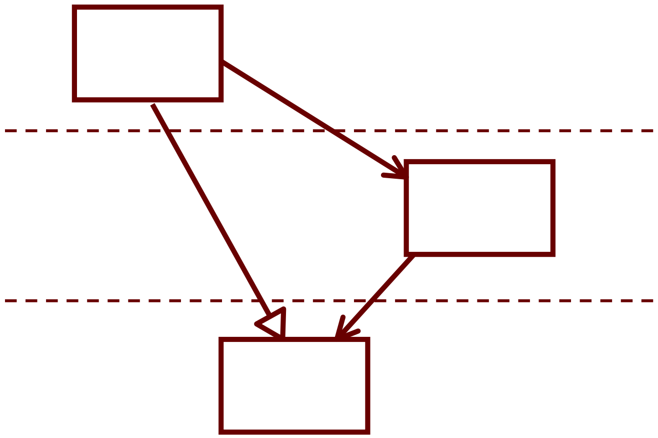
Layering: Acyclic Dependencies



System Layers vs Reality



Breaking up Cycles: (C++ Example)



Client/Server Pattern

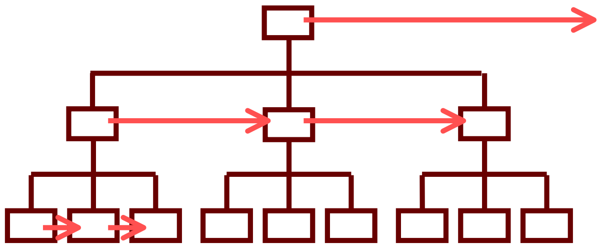
- Client has knowledge of server.
- Client requests data or operations from server and expects replies.
- Server sends replies to the client that requested it.
- Server has no specific knowledge of its clients. It keeps a list of subscribers though.
- Notifications are sent to all subscribers. server expects no reply on notifications.

Different Perspectives

- Structure of developed code
- Structure of run-time entities

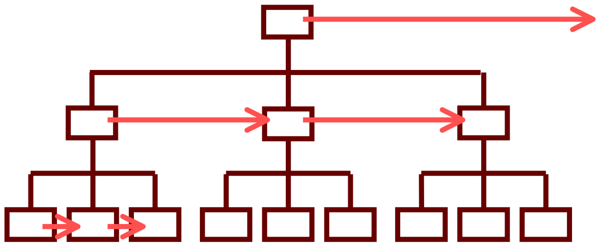
Structure of Developed Code

- **Hierarchy represents groupings of code**
- **Dependency represents compile time dependency**



Structure of Run-Time Entities

- **Hierarchy represents composition of functionality**
- **Dependency represents navigability**



Conflict Between Perspectives

- Development and Run-Time Perspective don't always match:
  - Static Libraries have no corresponding run-time entity.
  - Run-time modularization is corrupted by encapsulated development time dependencies.
  - ….
- Language dependent!
  - C++
  - Java
  - ROOM
  - EJB

Self Containment and Modularity

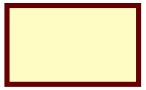
- Self Containment: Can be compiled without any other parts present.
  Don't sacrifice type-safety!
- Replacement Modularity: Can be separately delivered to target.
- Execution Modularity: Contains state, visible input and output

Language dependent!

Semantics of Parts (UML Structural Concepts)

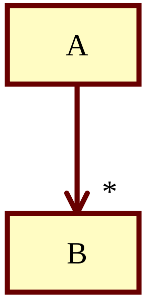
- Class
- Capsule
- Package
- Subsystem
- Component

Class



- Encapsulates code and data definitions
- Functional interface (methods)
- Instantiatable
- Corresponds to run time concept “object”
- Objects encapsulate behaviour and values.
- Classes may have hidden (implementation) compile time dependencies.
- Hidden compile time dependencies may result in run time relations with unclear semantics.

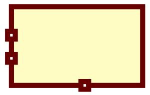
Class Relations: Conflict between Encapsulation and Traceability?




```
class A
{
public:
.
.
private:
   int myInt;
   B theBarray[];
};
```

Shouldn’t all traceable relations be public?

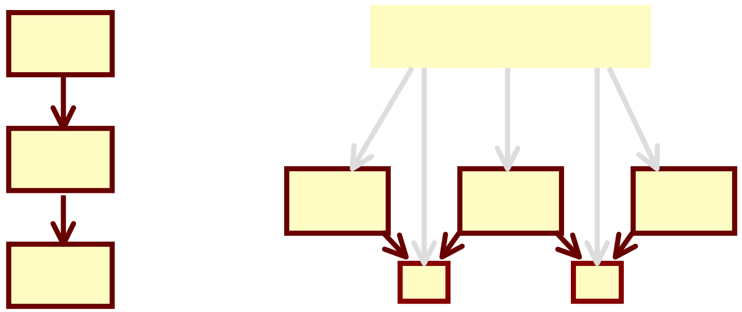
Capsule



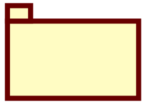
- Specialization of a class.
- Avoids implementation dependency
- Relations are public and under *external* control

Externally controlled relations:
A somewhat different paradigm.

Internally vs Externally Controlled Relations

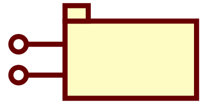


Package



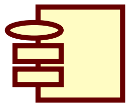
- Formally: A grouping of anything.
- In practice: A grouping of code, typically classes.
- Non-encapsulating. Contents is directly accessible from other packages.
- Development time grouping with unclear runtime correspondence.

Subsystem



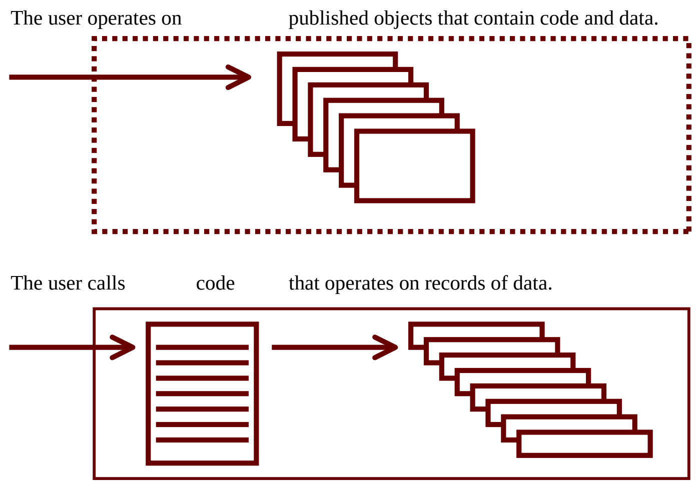
- Package with an interface.
- Development time concept.
- Unclear semantics, possible interpretation: A package that publishes a subset of its contents.
- May correspond to *component* delivered to target system.

Component



- Formally: Grouping of physical information (~files).
- Typically used for:
  - Static libraries
  - Dynamic libraries
  - Executables
- Somewhat unclear semantics:
  - Often assumed to correspond to modular run time instances having behaviour and state.
  - Correspondence holds for self contained subsystems?

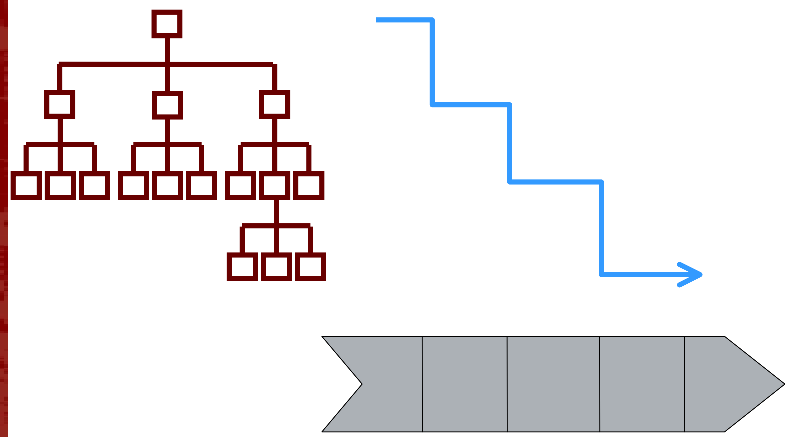
Analogy between OO and Functional Interfaces



Analogy between object oriented and functional Interfaces

- Operation type
- Target (as first argument for functional interfaces)
- Arguments
- Semantically identical
  Except:
  - For functional interfaces the targets type is implicit and not known by user.

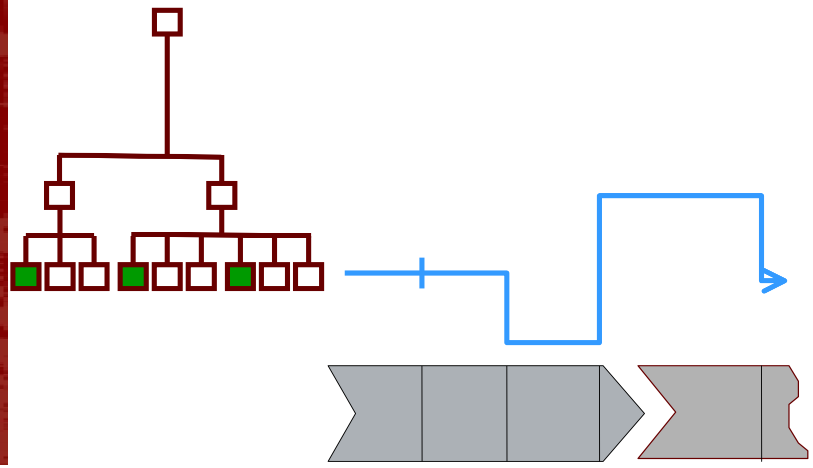
Top-Down Decomposition



Top-Down

- Subsystem interfaces specified early, to insulate parts of the system from each other.
- Interfaces may be inappropriately specified.
- Used as a means to make an early divisioning of work between groups of people.
- Suitable for large systems.

Bottom-Up Composition



Bottom-Up

- Basic modeling element is low level metaphoric classes.
- Non-encapsulating groupings fall out naturally when system matures.
- Interfaces fall out as the parts of the grouping that is accessible from outside the grouping.
- Suitable for small groups (< 20) of people.

What is Good Partitioning?

- Right level of granularity (5-7 rule)
- Understandable abstractions
- Acyclic dependencies
- Low coupling
- High cohesion
- Fan-in/Fan-out balance
- ….
- Somewhat depends on used paradigm

Measuring Module Quality

Reliable and early data with significant impact on quality.

- Classical metrics:
  - LOC
  - Cyclomatic number (McCabe)
  - Control variable complexity (McClure)
  - Software science (Halstead)
  - “Wrong” understanding of module
  - Late applicability
- More useful:
  - Coupling
  - Cohesion
  - Fan-in/fan-out
  - Graph-oriented metrics
  - Weighted methods per class
  - Depth/width of inheritance trees
  - ...

Coupling

- Measures the degree of independence between different modules
  - Content coupling (bad)
  - Common coupling
  - Control coupling
  - Stamp coupling
  - Data coupling (good)
- Each module should communicate with as few as possible other module
- Communicating modules should exchange as few as possible data
- All communication should be explicit

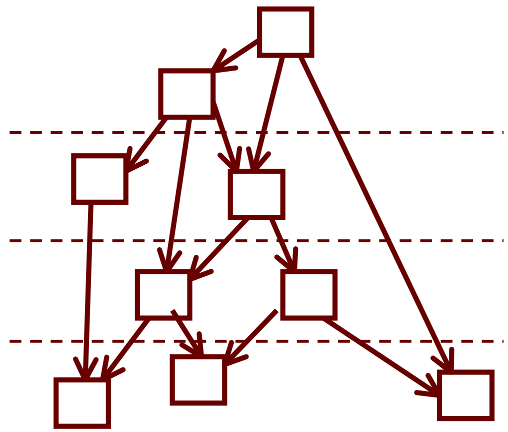
Cohesion

- Measures “relatedness” of the resources encapsulated in one module
  - Coincidental cohesion
  - Logical cohesion
  - Temporal cohesion
  - Procedural cohesion
  - Communicational cohesion
  - Sequential cohesion
  - Functional/informational cohesion

bad → good

➨ Each element in a module should be a necessary and essential part of one and only task

Would Layers be Good Modules?

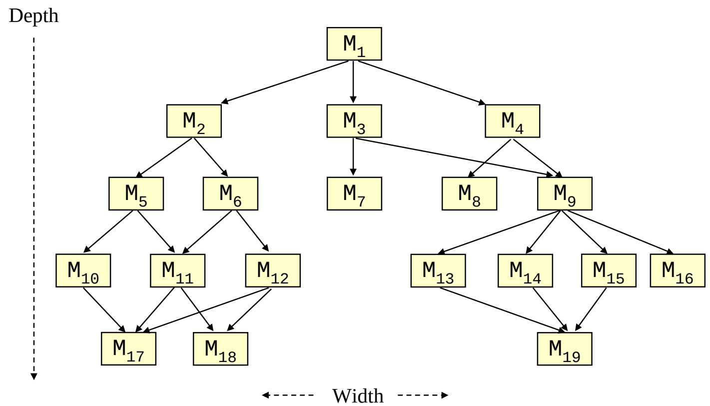


- **Coupling?**
- **Cohesion?**

Layers:

- High coupling
- Low cohesion
- Still a useful concept for reducing complexity
- Often better realized as a pattern than as a module.

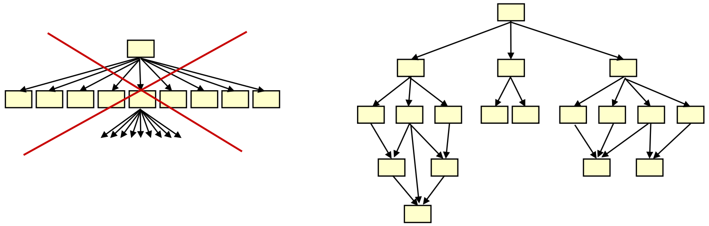
Fan-in vs Fan-out (Layering dimension)



M_9: Fan-in = 2, fan-out = 4

Fan-in vs Fan-out Rules

- Minimise structures with high fan-out
- Strive for fan-in as depth increases



Architectural Design Document Example

Slightly adapted from ESA's Software Engineering Standards PSS-05-0 (see [ESA 96])

A Traceability Matrix

- Relates requirements to design artefacts
  - ➨ Shows dependencies
  - ➨ Supports change management

	Module 1	Module 2	Module 3	Module 4	Module ...
Requ 1	X	X		X	
Requ 2	X		X		X
Requ 3				X	
Requ 4	X				
Requ ...				X	

➨ Useful for other traceability purposes