TU/e EINDHOVEN UNIVERSITY OF TECHNOLOGY

Software Design Principles

Michel R.V. Chaudron

Software Architecture 2020

CHALMERS UNIVERSITY OF TECHNOLOGY | UNIVERSITY OF GOTHENBURG

m.chaudron@tue.nl

Outline

- Recap
  - Architectural Styles
- Design Principles
- Tactics

Learning Objectives

- Know/explain design principles
- Apply design principles
- Recognize violations of design principles

- *Hint*: try if you can think up a counter-example for each design principle

Advice on Design of Software

- Generic Design Principles
- Principles for Architectural Design
- Principles for Design of Components
- Principles for collaboration amongst Components

Design = trade-offs = gray area

➔ Principles are heuristics

Not today: User Interface design, protocol design

General Software - Design Principles 1

Information Hiding:

All information about a module should be private to the module unless required externally

Minimize Coupling

Every module should depend on as few others as possible

Coherence:

Keep things together that belong together

Keep behaviour together with related data

Keep information about one thing in one place

General Software Design Principles 2

Divide and Conquer

Break a big problem into smaller ones

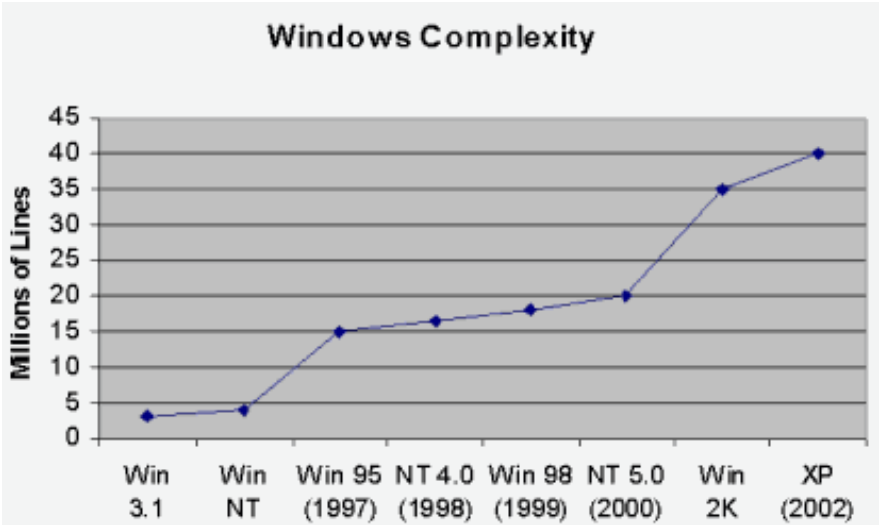
Separation of Concerns

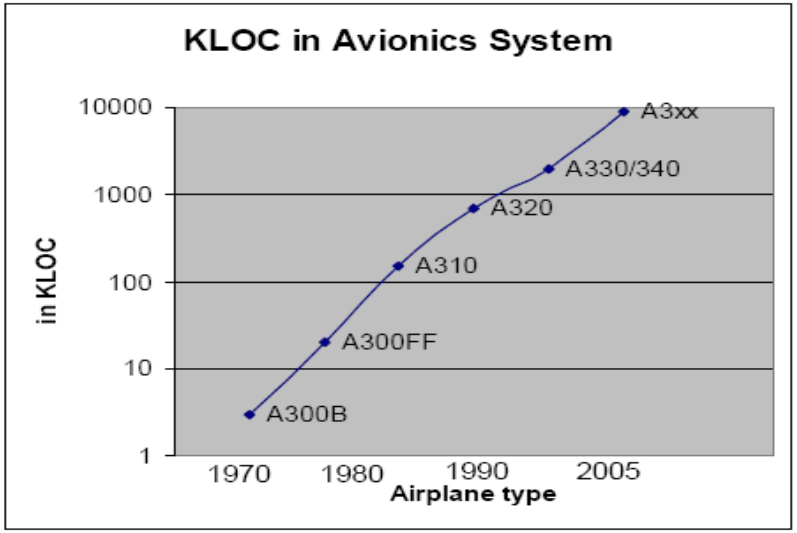
Divide into different parts logic

that addresses different issues

Keep it Simple

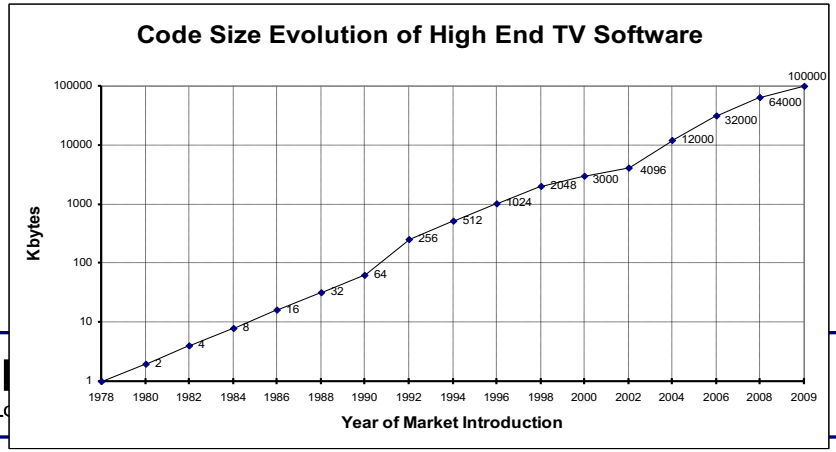
Motivation:
Increasing amount of software in systems



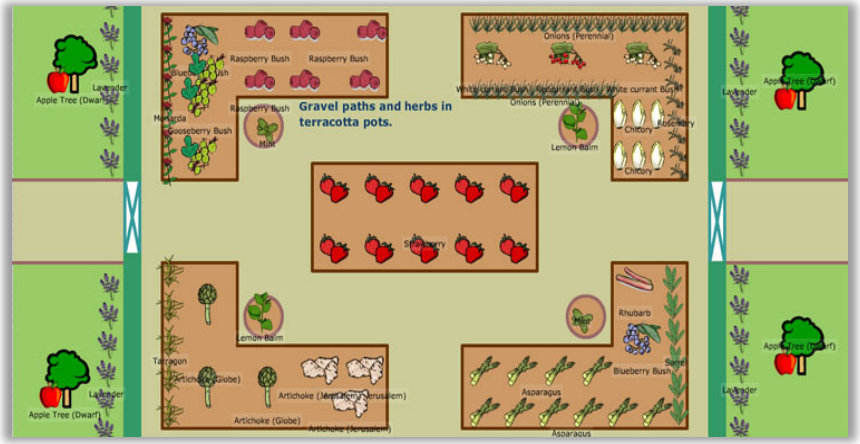


Nb: logarithmic scale

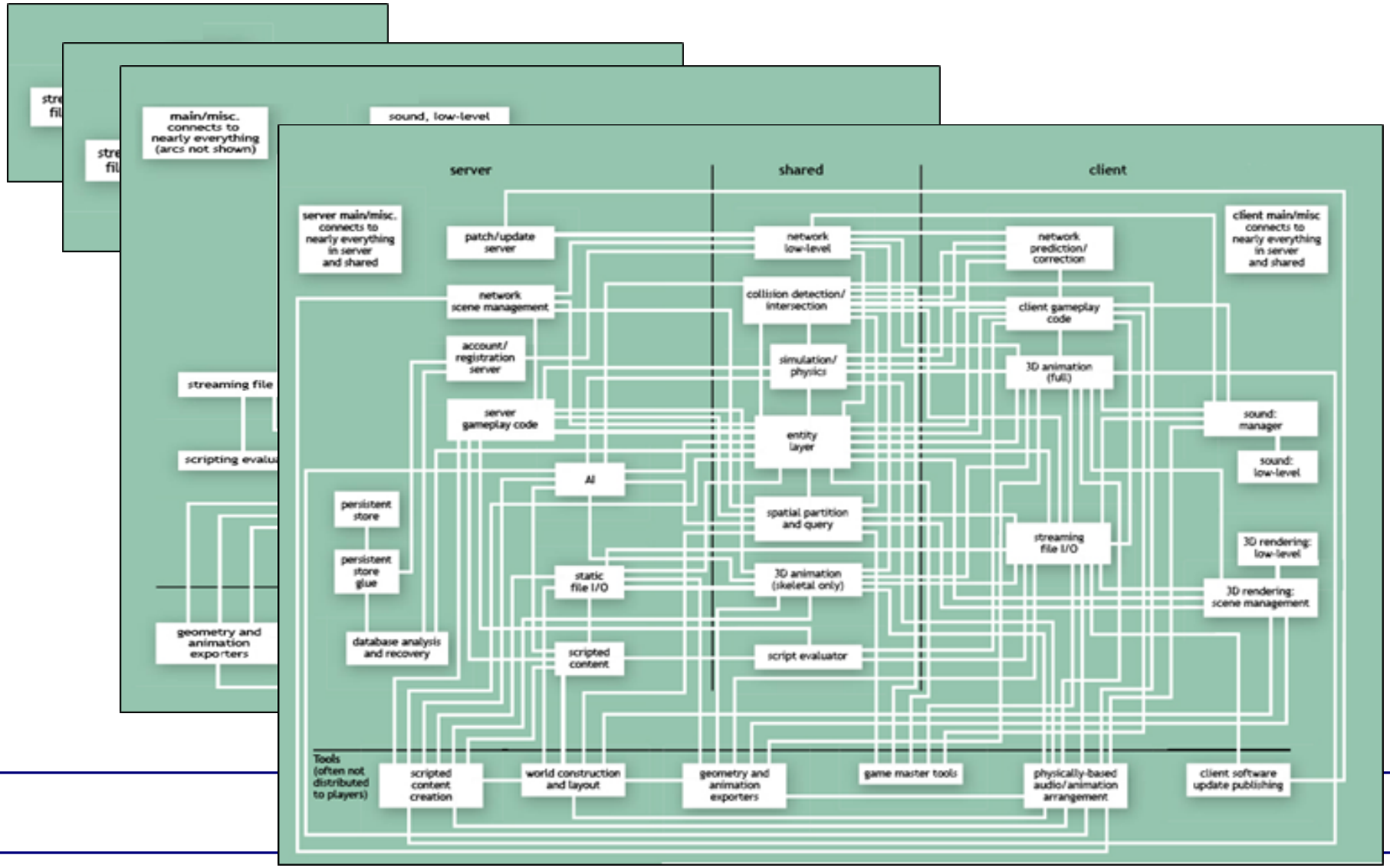
The amount of software increases 10 fold every 10 years



Software Evolves ‘Organically’



Increasing Complexity of Software



Blow, J. "Game Development: Harder Than You Think." *ACM Queue*

Complexity of Software

Slide by prof. Jurgen Vinju

If Kafka would write a book today…

This kind of software exists everywhere:

- 10K to 25M lines of code
- 2 to 10 programming languages and dialects
- 20 to 200 dependencies on library components and frameworks
- 10 to 1000 programmers
- 1 to 1M users
- 10 to 40 years lifetime
- "IT happens"

Franz Kafka

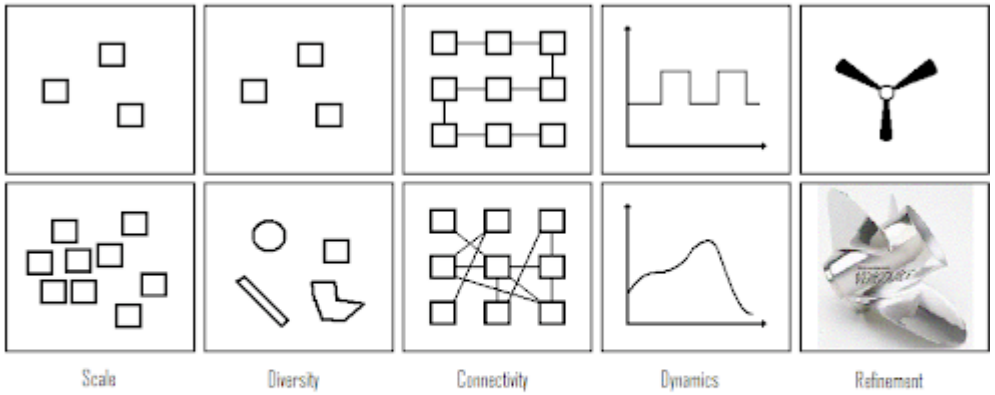
Writer

Franz Kafka was a German-language writer of novels and short stories, regarded by critics as one of the most influential authors of the 20th century. Wikipedia

Born: July 3, 1883, Prague, Czech Republic

having a nightmarishly complex, bizarre, or illogical quality

The 5 Complexity Dimensions of Software



Complexity in this regard means complex for humans to understand and contribute to.

1. **Scale**. The larger the system, the more complex.
2. **Diversity**. The more frameworks, languages, integration techniques, tools, platforms, and design patterns used, the more complex.
3. **Connectivity**. The more connections, the more complex. This relates to coupling.
4. **Dynamics**. The more number of states or the larger state space, the more complex.
5. **Refinement**. Over time every living piece of software is refined, optimized, and polished. Corner cases are found and handled, and regression test suites grow. Refinement drives complexity.

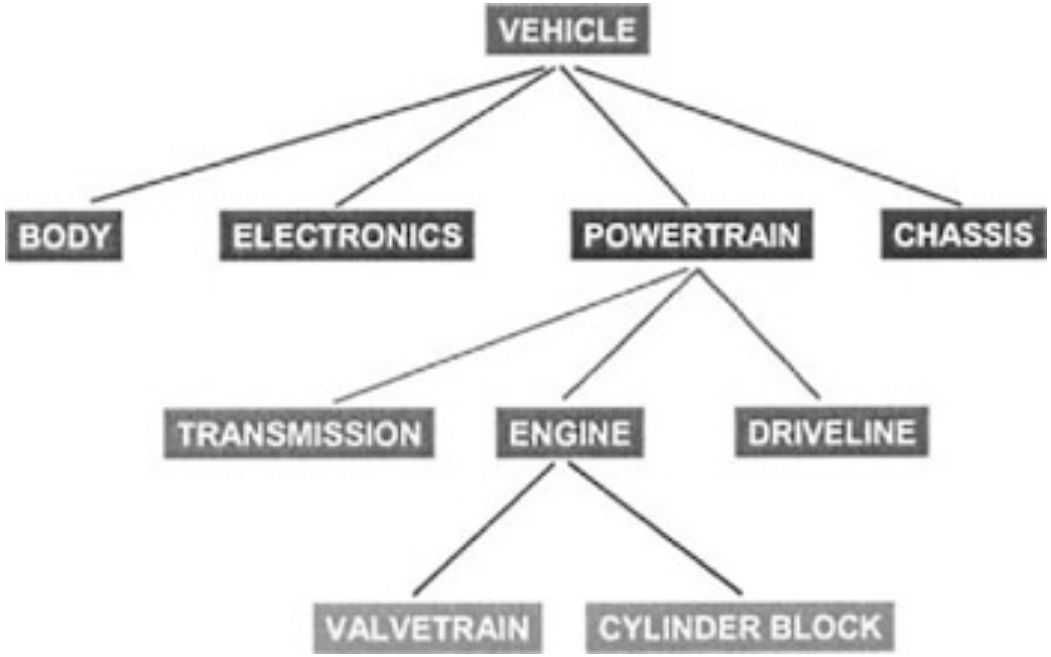
From : John Wilander http://appsandsecurity.blogspot.com/2011/03/5-complexity-dimensions-of-software.html

Generic Design Principles

- Decomposition

  Break problem into independent smaller parts

  Independent?

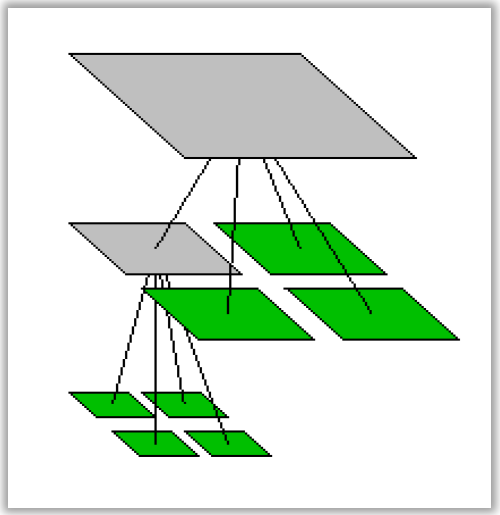


Single Responsibility

- What is a responsibility?
  - Rebecca Wirfs-Brock role-stereotypes
  - Depends on level of design
  - Relates to Parnas' principle of Information Hiding:
    - The responsibility relates to the secret
      - E.g. sorting
      - Viewer: way of displaying information
      - Model: storing & querying information
  - Alternative formulation ('Uncle Bob'):
    - A class should have only one reason to change

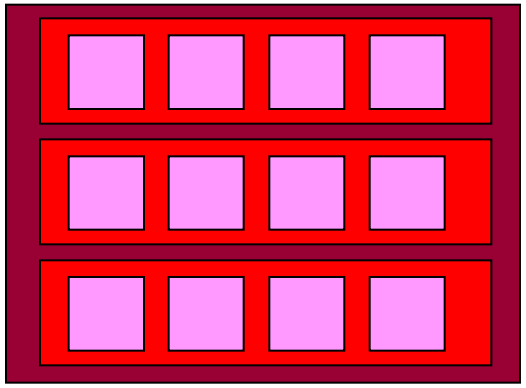
Design Principle : Divide and conquer

- Trying to deal with something big all at once is harder than dealing with a set of smaller things
  - Each individual component is smaller, and therefore easier to understand
  - Parts can be replaced or changed without having to replace or extensively change other parts.
  - Separate people can work on separate parts
  - An individual software engineer can specialize



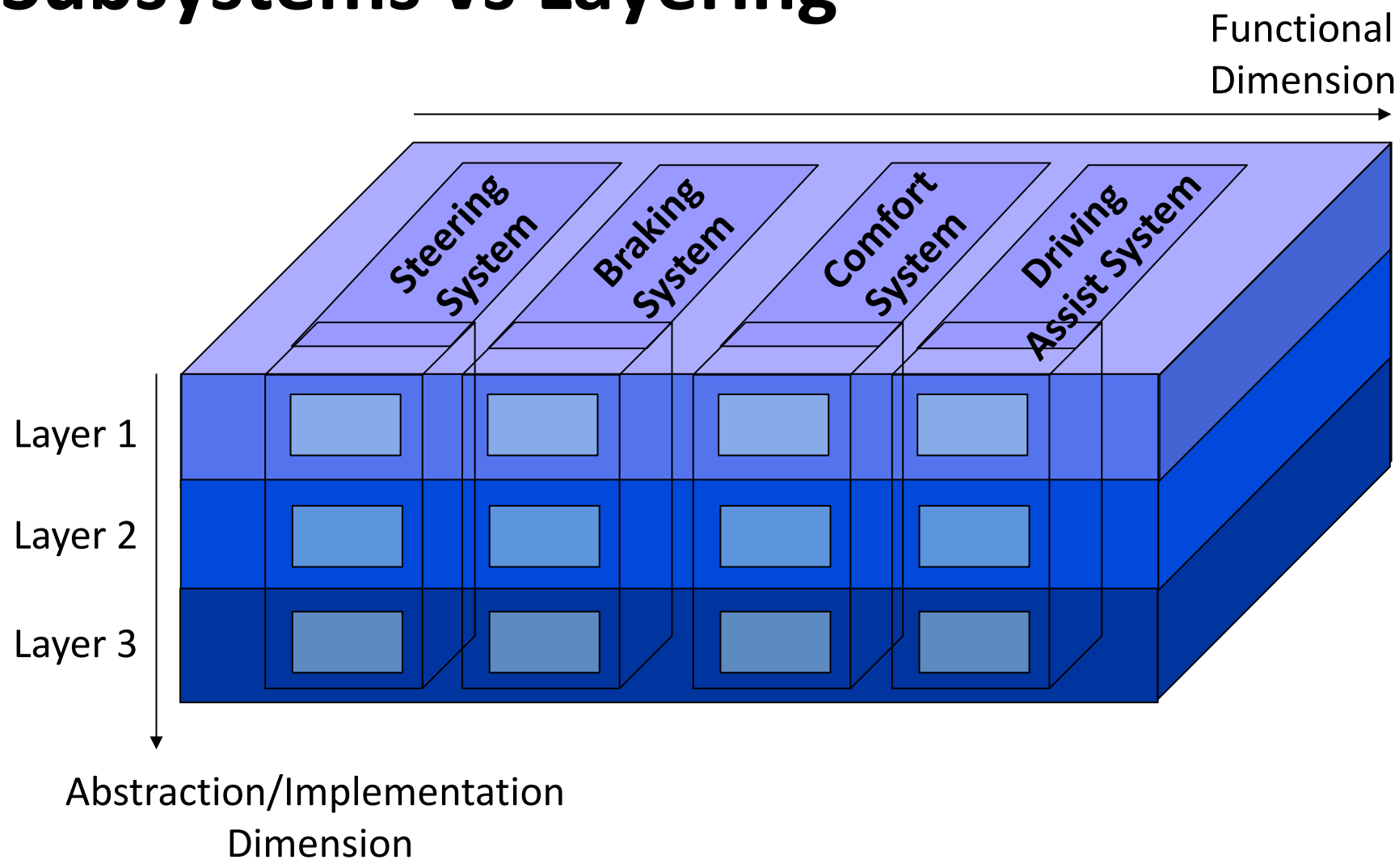
Ways of dividing a software system

A system can be divided up into



- Layers & subsystems
- A *subsystem* can be divided up into one or more *packages*
- A *package* is divided up into *classes*
- A *class* is divided up into *methods*

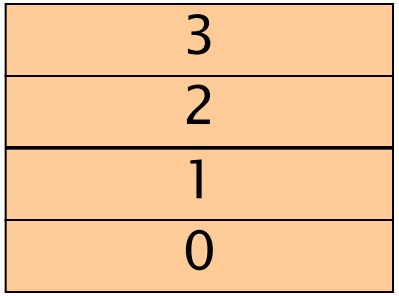
Subsystems vs Layering

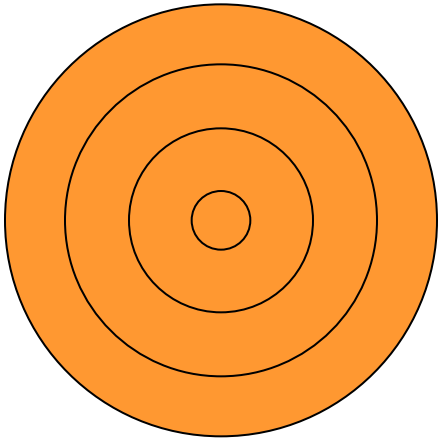


Layering

Goals: Separation of Concerns, Abstraction, Modularity, Portability

Partitioning in non-overlapping units that
- provide a cohesive set of services at an abstraction level
  (while abstracting from their implementation)
- layer *n* is allowed to use services of layer *n-1*
  (and not vice versa)
  alternative:
    bridging layers: layer *n* may use layers $< n$
    enhances efficiency but hampers portability





A Component-based Reference Architecture for Computer Games

(E. Folmer, 2007)

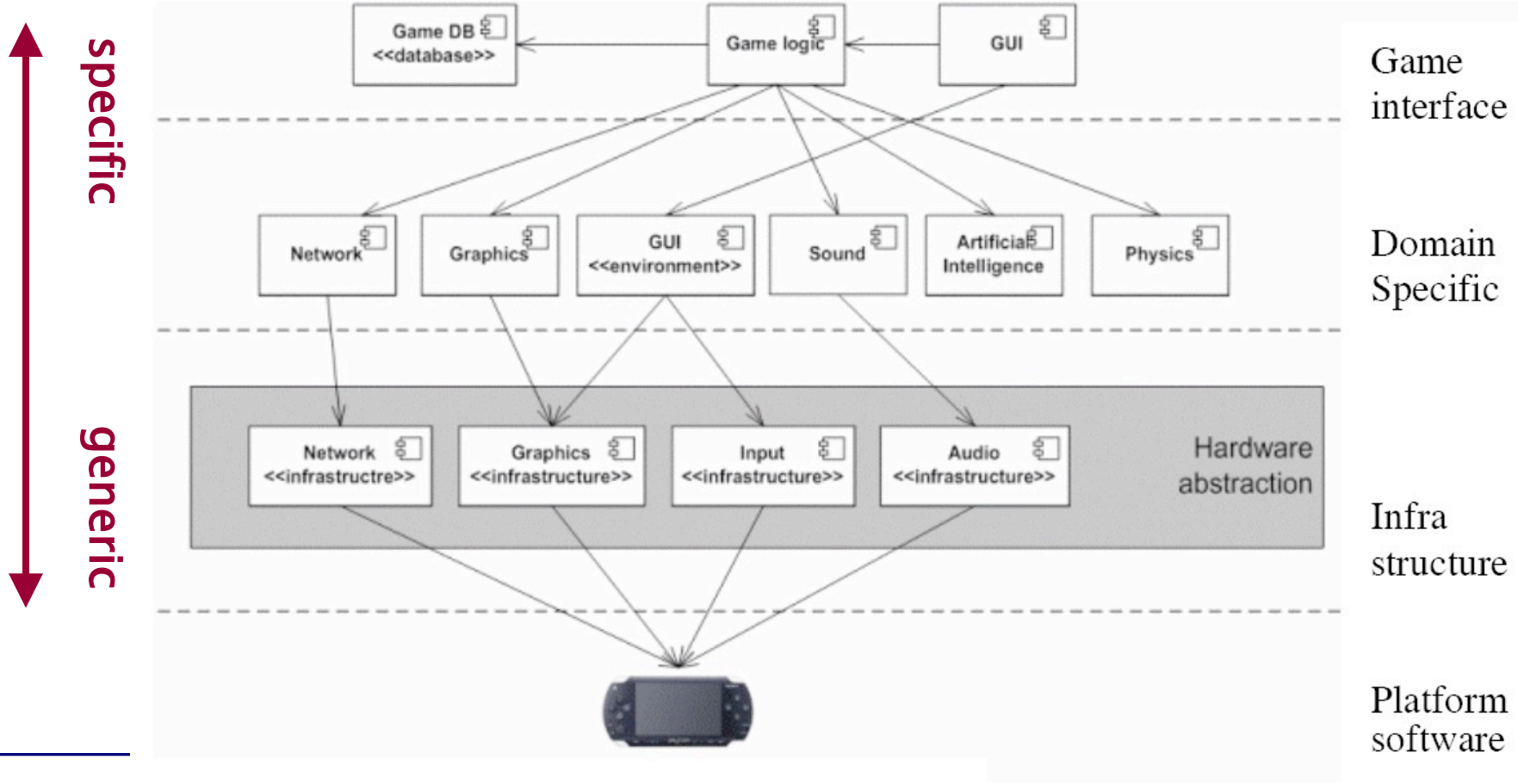
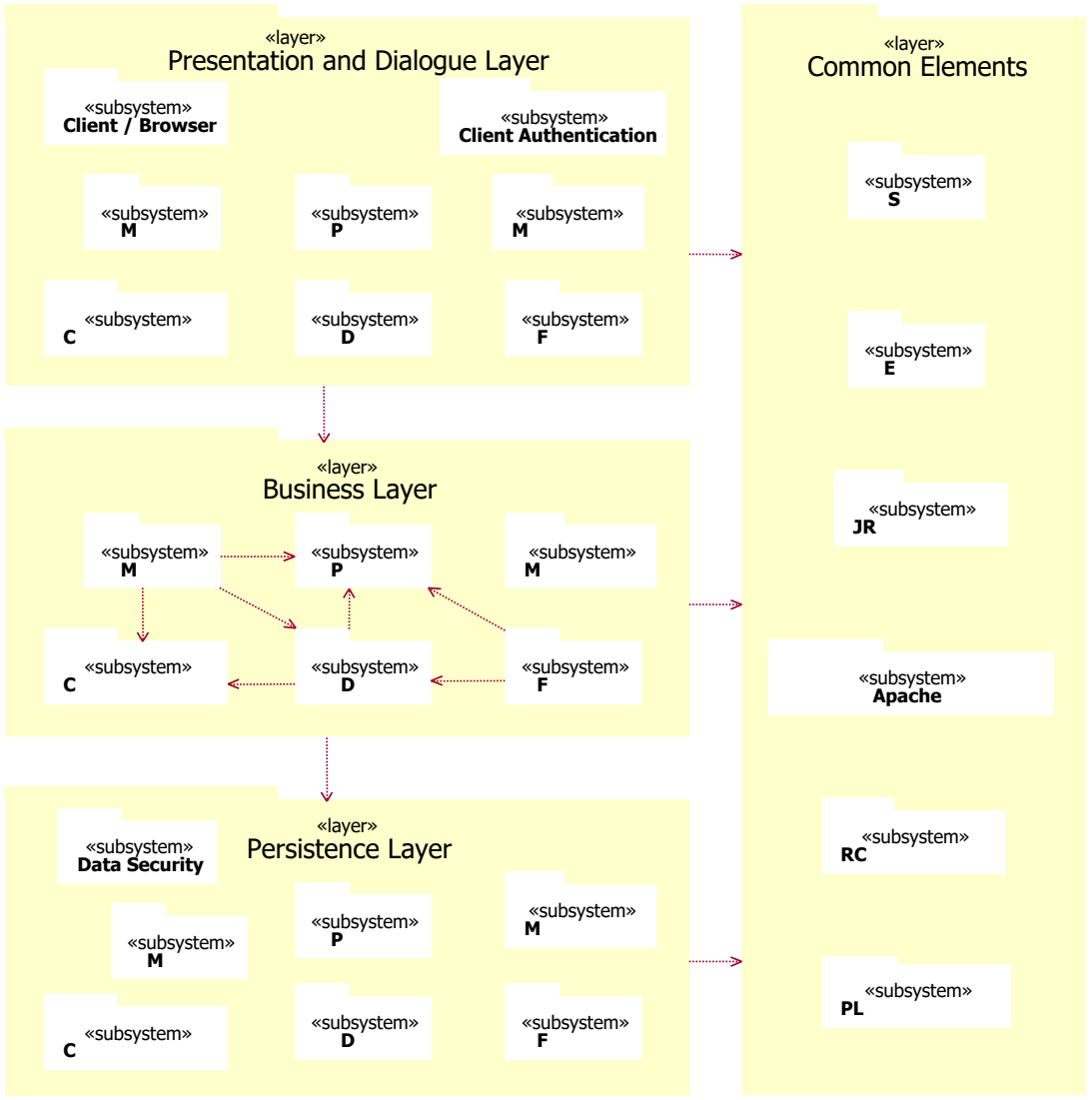
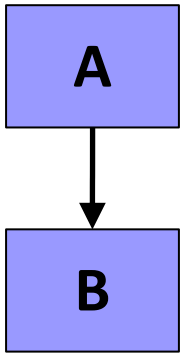


Fig. 1. A reference architecture for the games domain

Example


«layer»
Presentation and Dialogue Layer
«subsystem»
Client / Browser
«subsystem»
Client Authentication
«subsystem»
M
«subsystem»
P
«subsystem»
M
«subsystem»
C
«subsystem»
D
«subsystem»
F
«layer»
Business Layer
«subsystem»
M
«subsystem»
P
«subsystem»
M
«subsystem»
C
«subsystem»
D
«subsystem»
F
«layer»
Persistence Layer
«subsystem»
Data Security
«subsystem»
M
«subsystem»
P
«subsystem»
M
«subsystem»
C
«subsystem»
D
«subsystem»
F
«layer»
Common Elements
«subsystem»
S
«subsystem»
E
«subsystem»
JR
«subsystem»
Apache
«subsystem»
RC
«subsystem»
PL

What is a dependency?



A depends on B

- Component A requires B for it to *work*
  - Functional coupling

Run-time

- A change in module B requires a change in module A
  - Implementation coupling
  - Typically requires: re-testing A & B

Development-time

Dependency/Coupling

There is coupling between two classes ***A*** and ***B*** if:

- ***A*** calls a service of an object ***B***
- ***A*** has a method which references ***B***

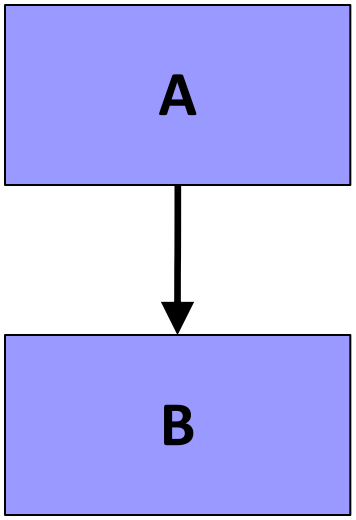
  (via return type or parameter)
- ***A*** has an attribute that refers to ***B***
- ***A*** is of type (inherits from) ***B***
  - ***A*** is a subclass of (or implements) class ***B***

This is not an exhaustive definition

A may depend on some assumption on another component B

Architecture Design Principles

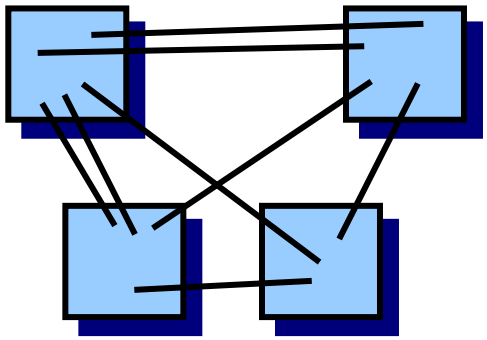
- Dependencies direct in the direction of **stability**



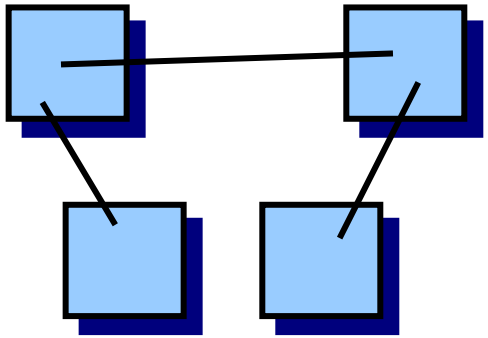
B should be
less likely to change
than A

Dependency: Coupling

Coupling is the degree of interdependence between modules



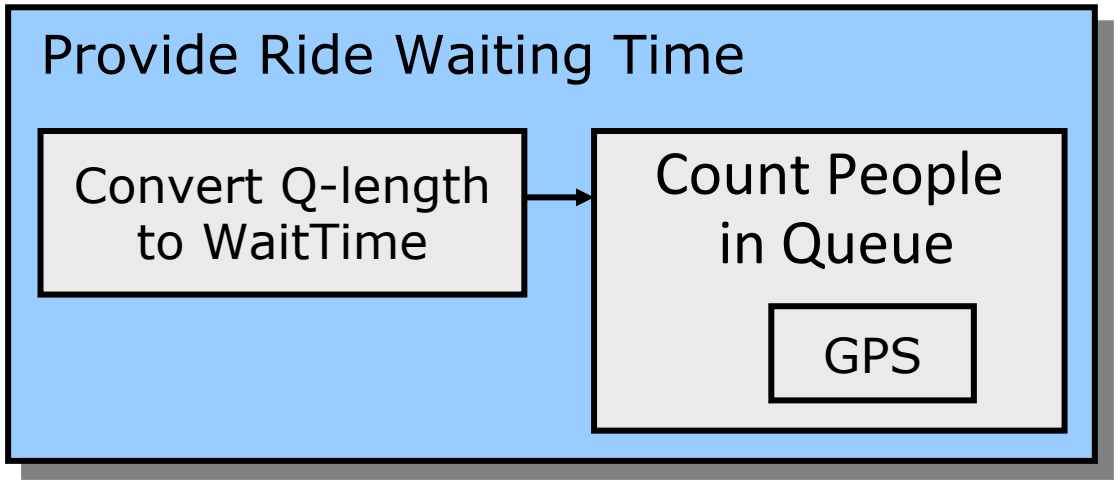
high coupling



low coupling

Cohesion

Cohesion is concerned with the *relatedness **within** a module*

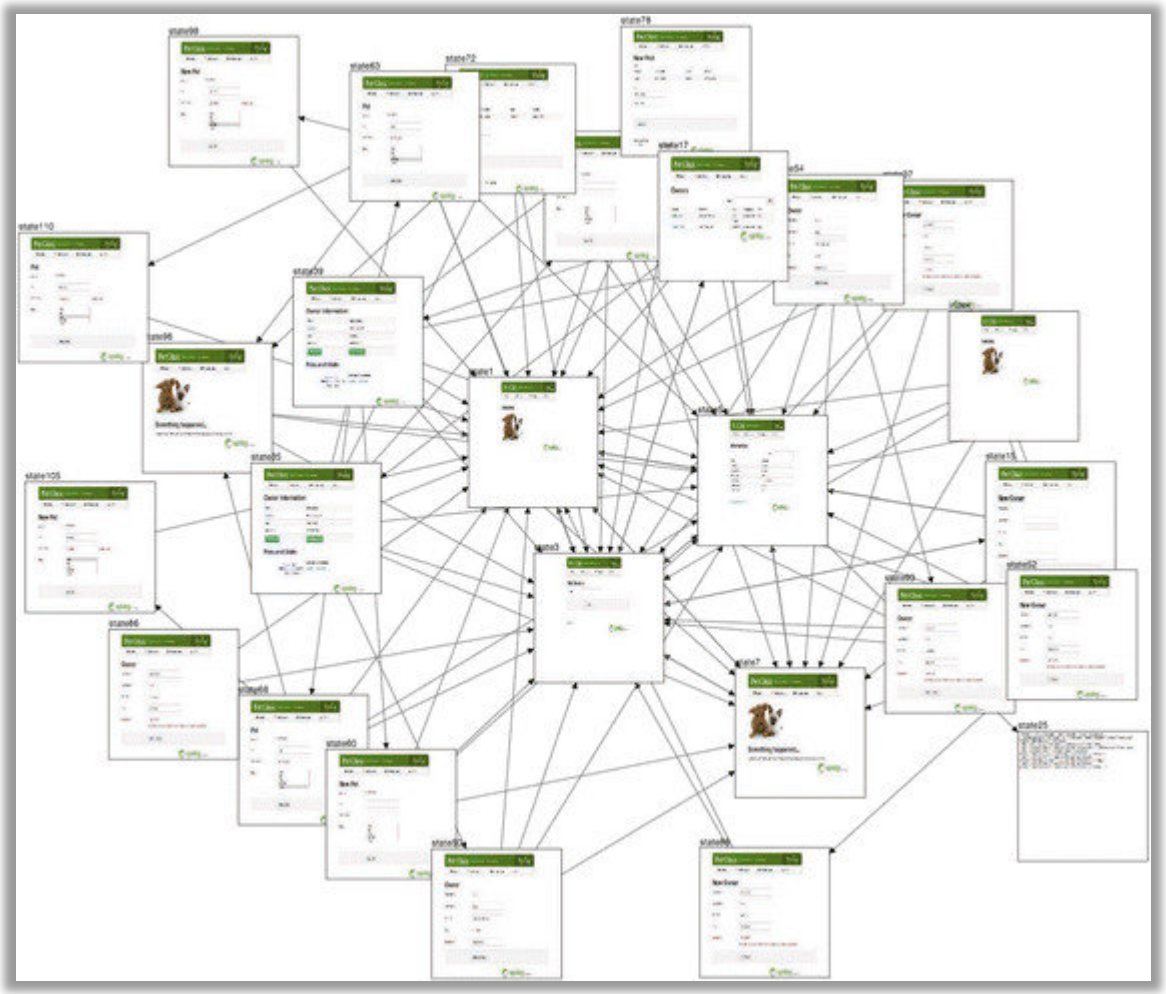


Benefits of Low Coupling/Dependencies

1. Modules are easier to replace
2. fewer interconnections between modules reduce time needed for **understanding** the modules and interactions
3. fewer interconnections between modules reduce the chance that **changes** in one module cause **problems** in other modules, which enhances *reusability*
4. fewer interconnections between modules reduce the chance that a fault in one module will cause a **failure** in other modules, which enhances *robustness*

Page-Jones, M. 1980. *The Practical Guide to Structured Systems Design*. New York, Yourdon Press, 1980.

What to avoid: many dependencies



Only 25 classes!

Reducing Coupling: Information Hiding

- Information Hiding:
  - Try to localize future change
  - Hide system details likely to change independently
  - Separate parts that are likely to have a different rate of change
  - In interfaces expose only assumptions unlikely to change

- Why is information hiding a good idea?
  - which types of coupling are prevented/reduced?

Information Hiding

Information Hiding is a means of avoiding dependencies.

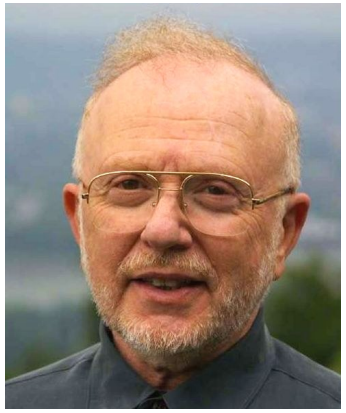
- Minimize the information interfaces disclose about the inner-workings of components
  - Balance with genericity
- Information hiding aims at avoiding dependencies on implementation details
- Corollary:
  - Components typically encapsulate volatile technologies

David Parnas

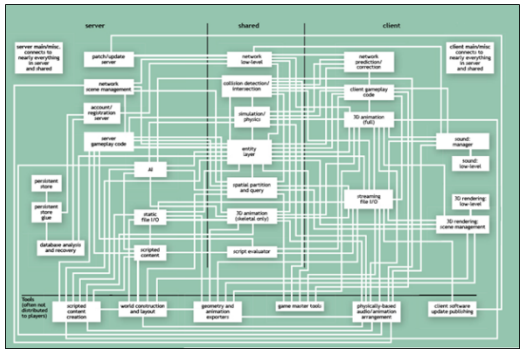
- *We propose that one begins with a list of:*
  - *difficult design decisions, or*
  - *design decisions which are likely to change*

*Each module is then designed to **hide** such a decision from the other modules.*

- *Goal: **ISOLATE CHANGE***
- *Means: Information hiding, minimizing dependencies*



David Parnas
1941-…



I advise students to pay more attention to the fundamental ideas rather than the latest technology.

The technology will be out-of-date before they graduate. Fundamental ideas never get out of date.

Design Principle: Information Hiding

- what is inside, must stay inside.

WHAT versus HOW

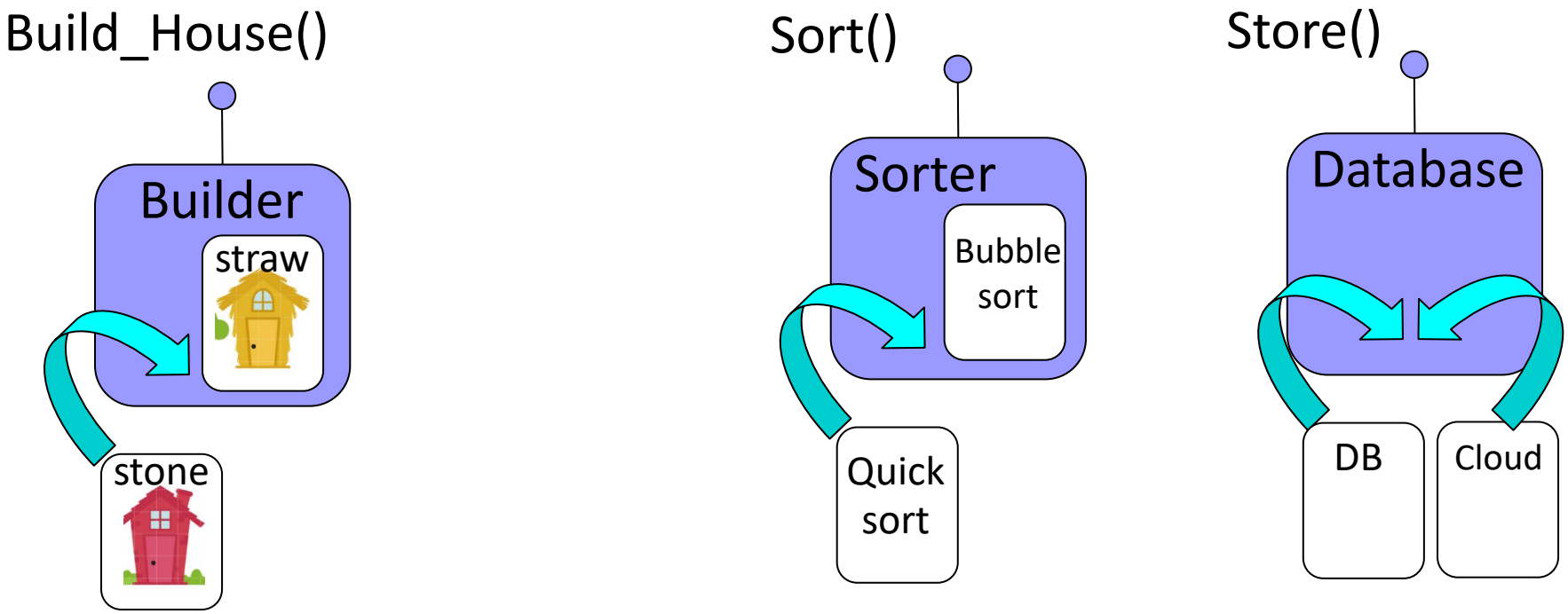
- ‘WHAT’: think Responsibility, Declarative
- Mechanisms are about ‘HOW’

WHAT: Build a house

HOW: stone, sticks, straw

Example: Change implementation

Public Interface



Supports evolution and platform-independence

Example 1

- IPrimeEncrypt(m,p)
- ICeasarEncrypt(m,s)
- IEncrypt(m)

Information hiding guides the design of the interface

The interface should aim to be:

- generic
  - We can do this by stating ‘what’, but not ‘how’
  - We can do this by avoiding unnecessary parameters in the calling of the component

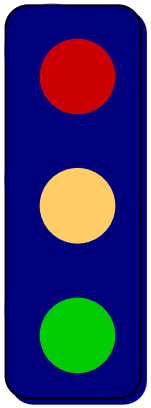
Example 2

- Steer a vehicle

- Interface
  - Option 1: Isteer = { TurnLeft, TurnRight }
  - Option 2: Isteer = { PressLeft, PressRight }
  - Option 3: Isteer = { Left, Right }

Alternative Interfaces

- Traffic Light



What should the interface of the traffic light look like?

Which secrets to hide? 

which abstraction to expose via the interface?

Take 3 minutes to design your own interface

Traffic Light - Alternative Interfaces

Traffic Light1

- Reset()
  - Postcondition: RED
- Run()
  - Red→Green→ Orange→ Red
- SetIntervalDuration(t)

Traffic Light2

- SetRed(On/Off):Exc
- SetOrange(On/Off):Exc
- SetGreen(On/Off):Exc
- Blink/Disco()
- GetState(…)

Traffic Light3

- Halt()
- Warn()
- Drive()

‘Secrets’

- Actual colours
- Initial state
- Order of lights (easy to change)
- ‘On’ is Mutual exclusive

Synchronization/Timing?

‘Secrets’

- Initial state
- Order of states

More Generic

(lights not exclusive)

‘Secrets’

- Actual colours
- Initial state
- Order of states

Higher Level of abstraction

Chest of drawers by *Droog* Design

Build from ‘modules’

But no stable architecture

Many dependencies from all drawers on all other drawers

Simplicity

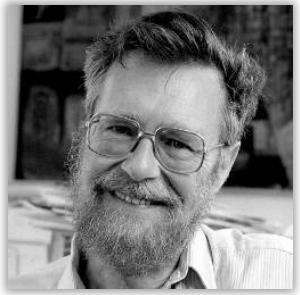
Simplicity is a great virtue but it requires hard work to achieve it and education to appreciate it.

And to make matters worse:

Complexity sells better.

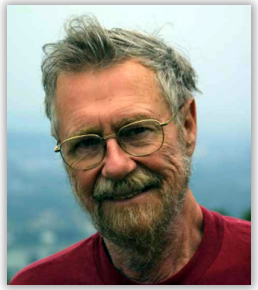
Source: Edsger W. Dijkstra

EWD896 - On the nature of Computing Science



Turing Award (1972)

Edsger Wybe Dijkstra



- 1930-2002
- Ph.D. in Physics
  Leiden University, Netherlands
- Contributions to:

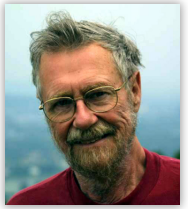
  Algorithms, Concurrency, Distributed Systems,
  Program Correctness, Discipline of Design:

  Structured Programming (Go To considered Harmful)

  Separation of Concerns
- Turing Award (1972)

O.-J. Dahl, E. W. Dijkstra, C. A. R. Hoare, *Structured Programming*, Academic Press, London, 1972

Generic Design Principle: Separation of Concerns



Edsger W. Dijkstra

- Decomposition / Divide and Conquer
- Issues that are not related should be handled in separate parts
- Single responsibility:
  - Assign a single responsibility to a single component/class

    Typical responsibilities: to know something, to do something

    E.g. to know an algorithm (worker)

    to coordinate workers (coordination)

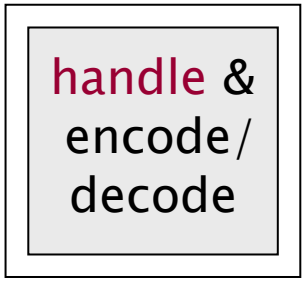
    to manage student-records (information holder)

Example Separation of Concern Principle

Telecom Domain:

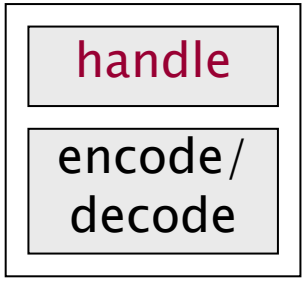
Telecom protocol:

- **decode1 ; handle1 ; decode2 ; handle2 ; decode3**



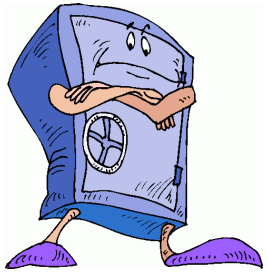
Separate the encoding/decoding of a message from the handling of a message:

decode1 ; decode2 ; decode3 ; handle1 ; handle2



Separation of Concerns in Interface Design

- Separate ***What*** from ***How***
- The interface of a component exposes **what** it do, but not **how** it does this.
- The 'how' is the *information-hiding* 'secret'
  - Details of the data representation
  - Details of the algorithm



Design Principles

- Keep things that belong together at a single place

  e.g. in OO: data and
  the operations on that data

- Don't replicate

  functionality, storage of data

Summary

- Design Principles
  - Know them, Apply them
  - Recognize violations

Information Hiding

Minimize Coupling

Divide and Conquer

Separation of Concerns

Keep it Simple

- Design Structure Matrix
  - can you read it? make it?

Questions?

- Explain how layering relates to separation of concerns?

Summary of key architecting practices

- Understand the **drivers** for the project (business, politics)
- Get stakeholder involvement early and often
- Understand the requirements incl. quality properties
  - SMART & prioritized
- Develop iteratively and incrementally
- Describe architecture using multiple views
  - abstract, but precise, design decisions & rationale
- Design for change (modularity, low coupling, inform. hiding)
- Analyze in an early stage (use maths! and scenarios)
- **Simplify, simplify, simplify**
- Regularly update planning and risk analysis
- Monitor that architecture is implemented
- Get **good people**, make them happy set them loose

Online lecture 14 Oct 2020