Design Heuristics and Styles

(LL Chapter 9)

Michel Chaudron

Leiden Institute of Advanced Computer Science

Many slides based on Lethbridge and Laganiere

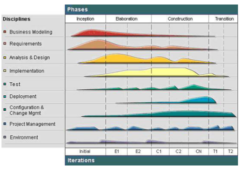
Agenda

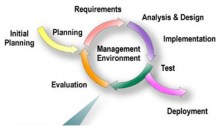
- Recap RUP
- Design heuristics & guidelines
- Architectural Styles

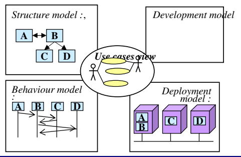
- This afternoon: geen werkcollege
- hand in assignments electronically

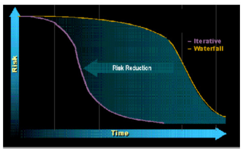
  chaudron@liacs.nl

Summary Rational Unified Process









Software Design Heuristics

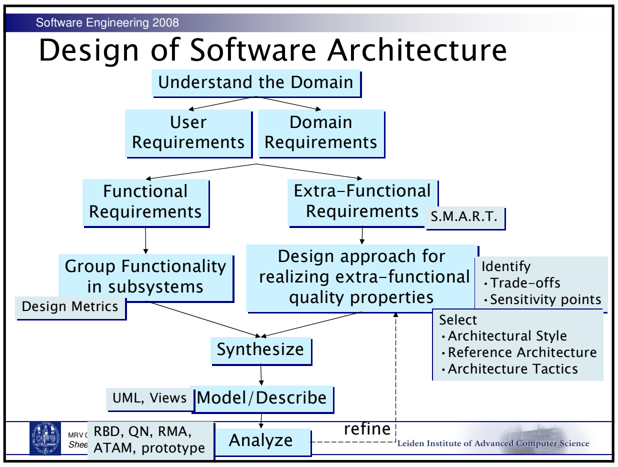
Different aspects of design

- □ *Architecture design*:
  - The division into subsystems and components,
    - □ How these will be connected:
    - □ How they will interact:
      - Interface design & architectural style
- □ *Class design*:
  - The various features of classes.
- □ *User interface design*
- □ *Algorithm design*:
  - The design of computational mechanisms.
- □ *Protocol design*:
  - The design of communications protocol.

Architecture is making decisions

> **The life of a software architect is a long (and sometimes painful) succession of suboptimal decisions made partly in the dark.**
>
> **Grady Booch**

- You will not have all information available
- You will make mistakes, but you should learn from them
- There is no objective measure for 'goodness'



Software Engineering 2008

Design of Software Architecture

Understand the Domain

User Requirements

Domain Requirements

Functional Requirements

Extra-Functional Requirements

S.M.A.R.T.

Group Functionality in subsystems

Design Metrics

Design approach for realizing extra-functional quality properties

Identify
- Trade-offs
- Sensitivity points

Select
- Architectural Style
- Reference Architecture
- Architecture Tactics

Synthesize

UML, Views

Model/Describe

RBD, QN, RMA, ATAM, prototype

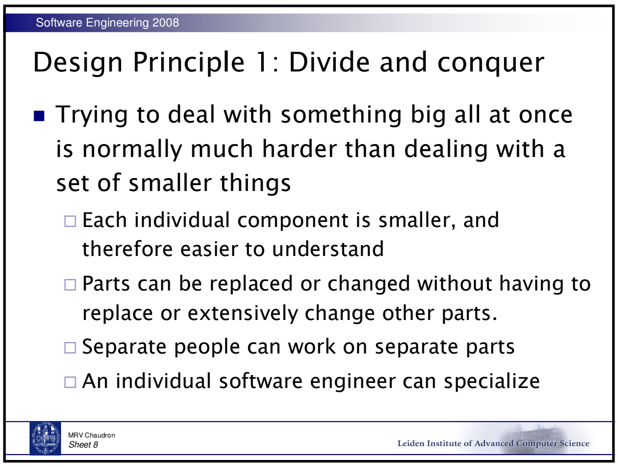
Analyze

refine

MRV C

Shee

Leiden Institute of Advanced Computer Science



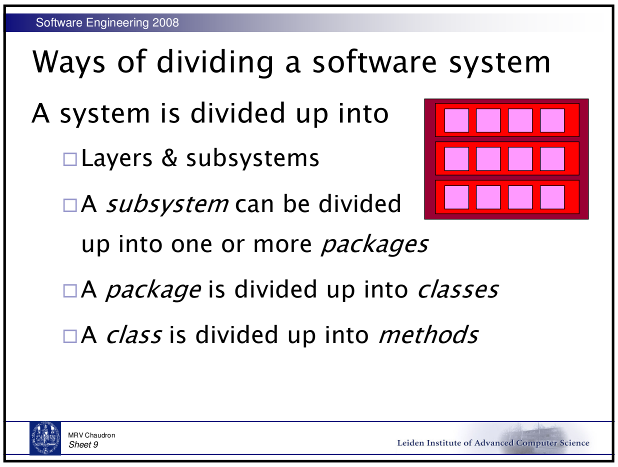
Software Engineering 2008

Design Principle 1: Divide and conquer

- Trying to deal with something big all at once is normally much 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

MRV Chaudron
Sheet 8

Leiden Institute of Advanced Computer Science



Software Engineering 2008

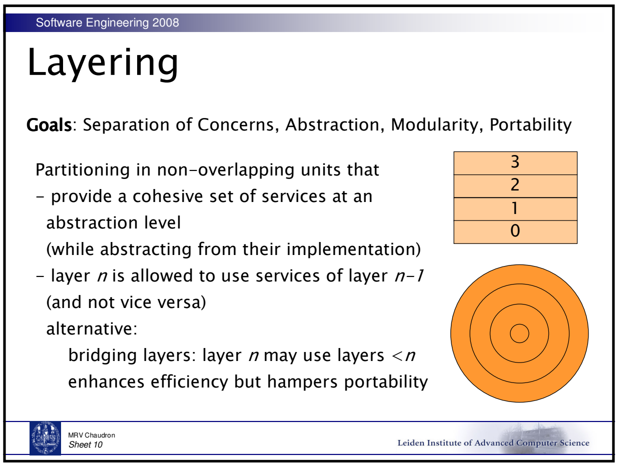
Ways of dividing a software system

A system is 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*

MRV Chaudron
Sheet 9

Leiden Institute of Advanced Computer Science



Software Engineering 2008

Layering

Goals: Separation of Concerns, Abstraction, Modularity, Portability

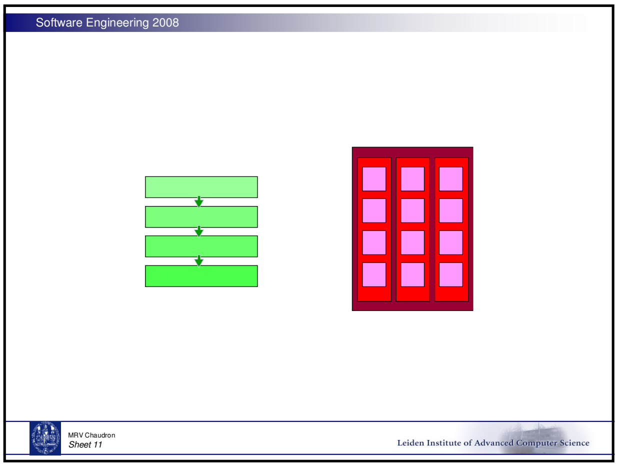
Partitioning in non-overlapping units that
- provide a cohesive set of services at an abstraction level
  (while abstracting from their implementation)
- layer *n* is allowed to use services of layer *n-1*
  (and not vice versa)

alternative:
bridging layers: layer *n* may use layers <*n*
enhances efficiency but hampers portability

3
2
1
0

MRV Chaudron
Sheet 10

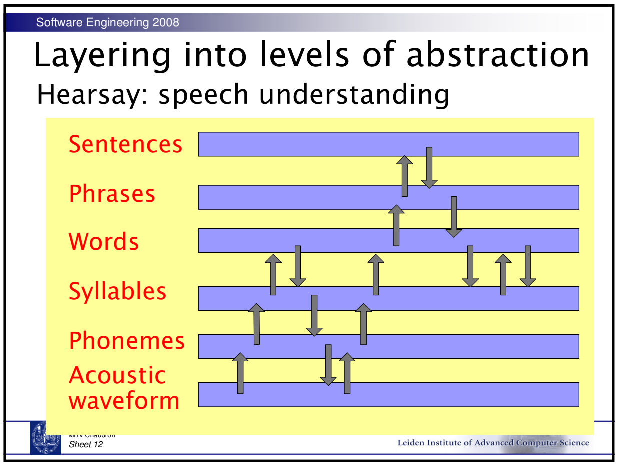
Leiden Institute of Advanced Computer Science



Software Engineering 2008

MRV Chaudron
Sheet 11

Leiden Institute of Advanced Computer Science



Software Engineering 2008

Layering into levels of abstraction

Hearsay: speech understanding

Sentences

Phrases

Words

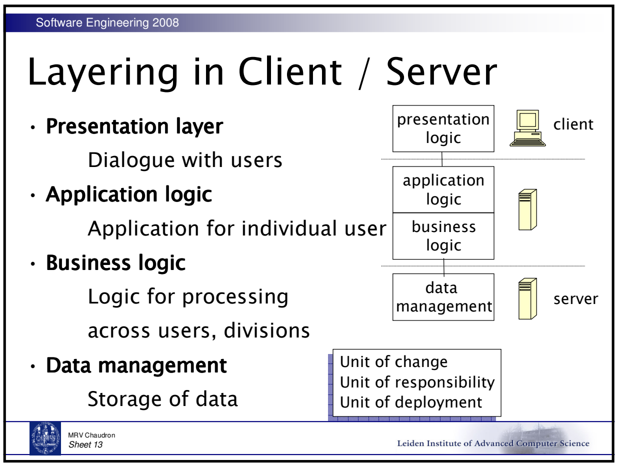
Syllables

Phonemes

Acoustic waveform

MRV Chaudron
Sheet 12

Leiden Institute of Advanced Computer Science

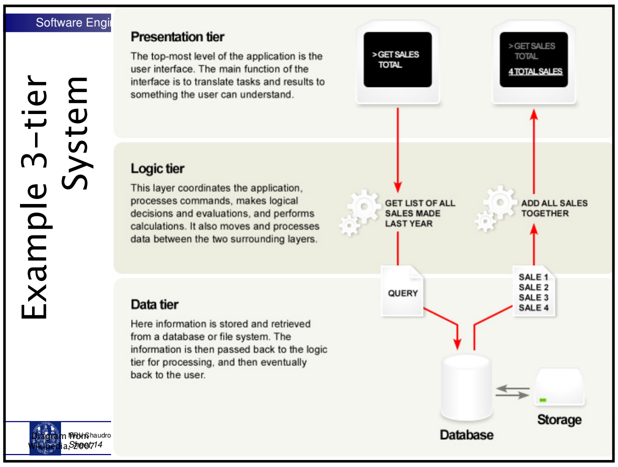


Layering in Client / Server

- **Presentation layer**
  Dialogue with users
- **Application logic**
  Application for individual user
- **Business logic**
  Logic for processing
  across users, divisions
- **Data management**
  Storage of data

presentation logic
client
application logic
business logic
data management
server

Unit of change
Unit of responsibility
Unit of deployment



Example 3-tier System

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.

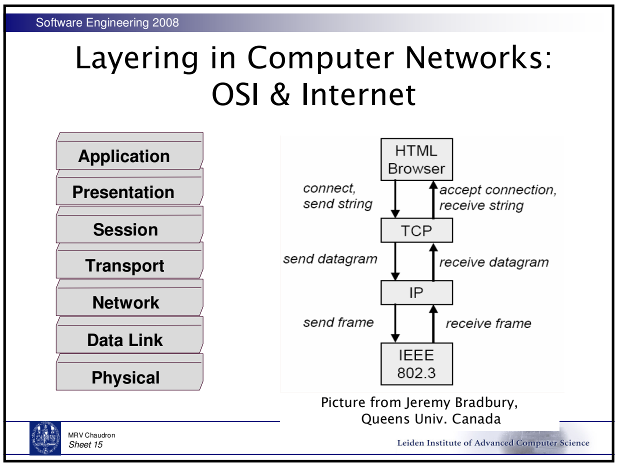
Logic tier

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.

Diagram from Wikipedia, 2007

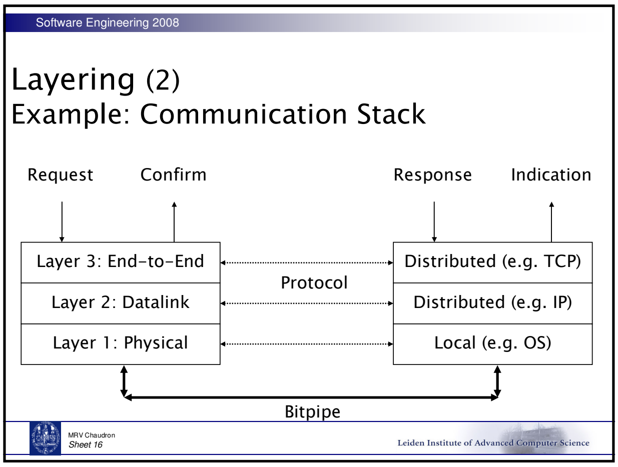


Layering in Computer Networks: OSI & Internet

Application
Presentation
Session
Transport
Network
Data Link
Physical

HTML Browser
connect, send string
accept connection, receive string
TCP
send datagram
receive datagram
IP
send frame
receive frame
IEEE 802.3

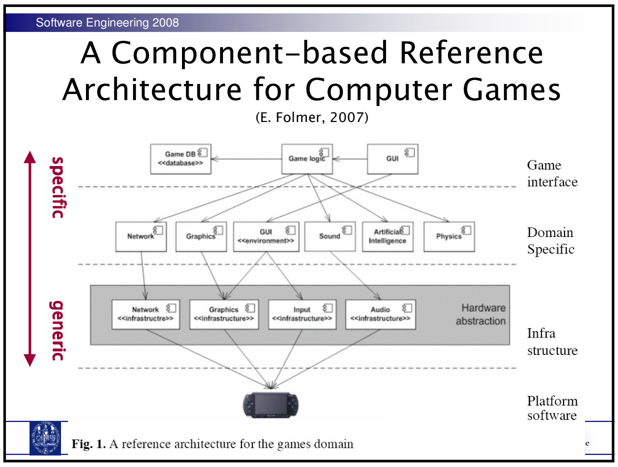
Picture from Jeremy Bradbury, Queens Univ. Canada



Layering (2)
Example: Communication Stack

Request
Confirm
Response
Indication

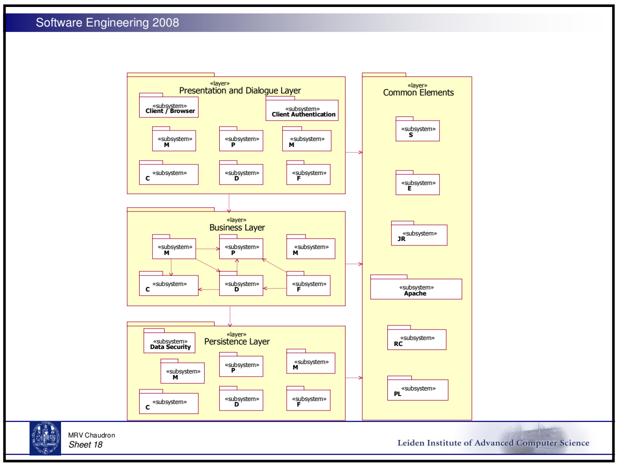
Layer 3: End-to-End
Layer 2: Datalink
Layer 1: Physical
Protocol
Distributed (e.g. TCP)
Distributed (e.g. IP)
Local (e.g. OS)
Bitpipe

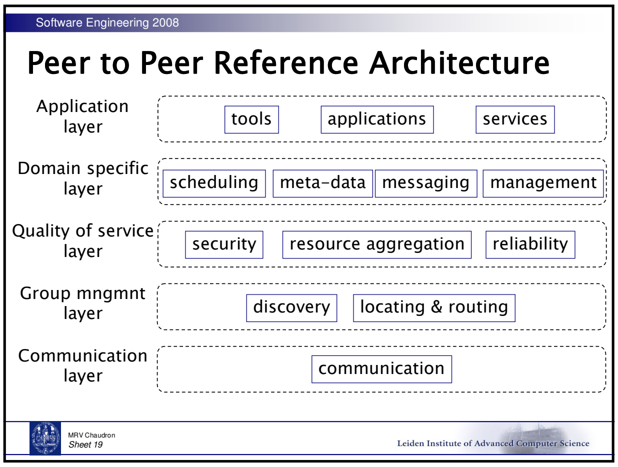


A Component-based Reference Architecture for Computer Games

(E. Folmer, 2007)

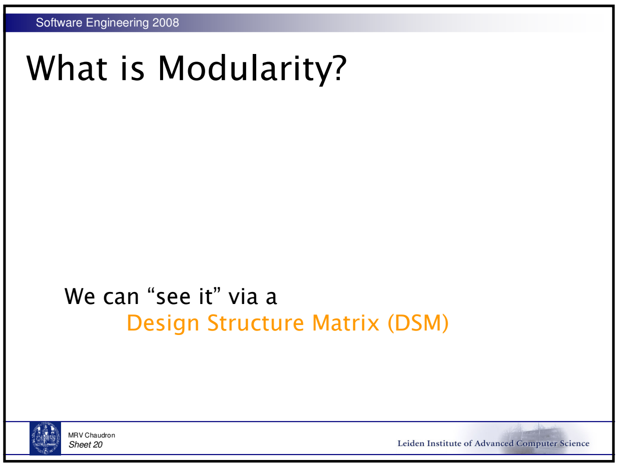
Fig. 1. A reference architecture for the games domain





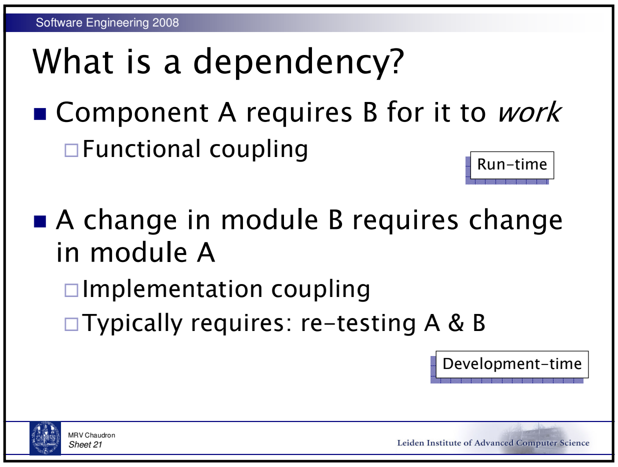
Peer to Peer Reference Architecture

Application layer	tools, applications, services
Domain specific layer	scheduling, meta-data, messaging, management
Quality of service layer	security, resource aggregation, reliability
Group mngmnt layer	discovery, locating & routing
Communication layer	communication



What is Modularity?

We can "see it" via a
Design Structure Matrix (DSM)



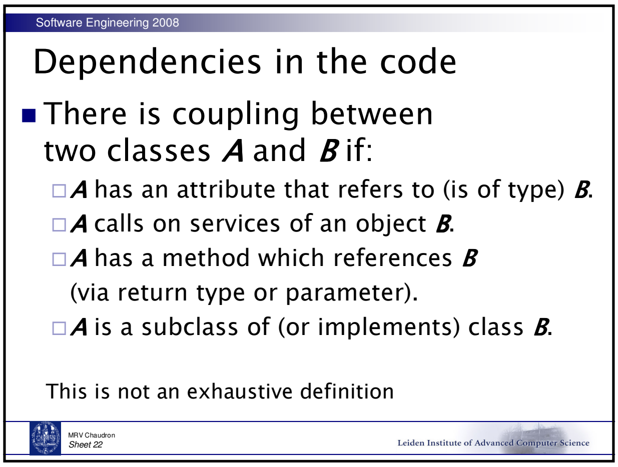
What is a dependency?

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

Run-time

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

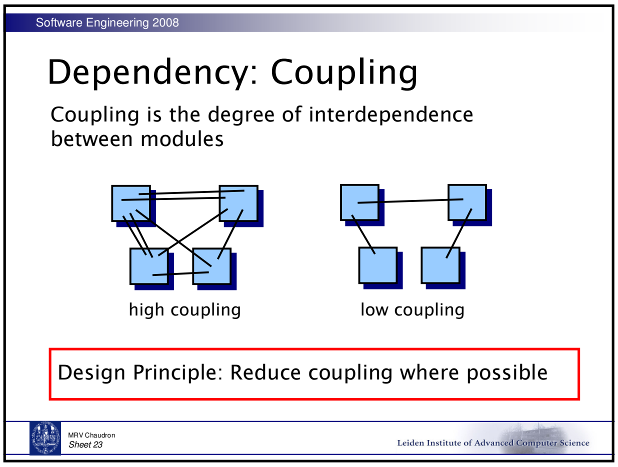
Development-time



Dependencies in the code

- There is coupling between two classes ***A*** and ***B*** if:
  - □***A*** has an attribute that refers to (is of type) ***B***.
  - □***A*** calls on services of an object ***B***.
  - □***A*** has a method which references ***B*** (via return type or parameter).
  - □***A*** is a subclass of (or implements) class ***B***.

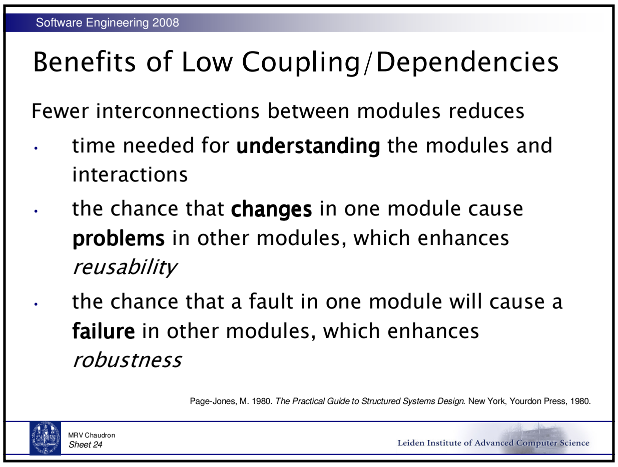
This is not an exhaustive definition



Dependency: Coupling

Coupling is the degree of interdependence between modules

Design Principle: Reduce coupling where possible



Benefits of Low Coupling/Dependencies

Fewer interconnections between modules reduces

- time needed for **understanding** the modules and interactions
- the chance that **changes** in one module cause **problems** in other modules, which enhances *reusability*
- 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.

Guideline: Minimize Dependency

Avoid dependencies where possible:

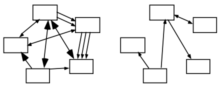
Design components so that

- □ they know about as few other components as possible
  - ▪use as few parameters as possible
- □ for as short a time as possible
  - ▪minimize number of calls between components

Ref: Component are from Mars – Chaudron & De Jong

Design Principle:
Reduce coupling where possible

- *Coupling* occurs when there are *interdependencies* between one module and another
  - □ When interdependencies exist, changes in one place will require changes somewhere else.
  - □ A network of interdependencies makes it hard to see at a glance how some component works.
  - □ Type of coupling:
    - Content, Common, Control, Stamp, Data, Routine Call, Type use, Inclusion/Import, External



Separation of Concerns

- Zaken die niet bij elkaar horen moeten in verschillende eenheden (componenten / procedures / ..) worden geaddresseerd

Example Design Principles

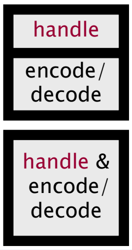
Telecom Domain:

Separate the encoding/decoding of a message from the handling of a message, so

- **decode1 ; decode2 ; decode3 ; action1 ; action2**

And not

- **decode1 ; action1 ; decode2 ; action2 ; decode3**



handle

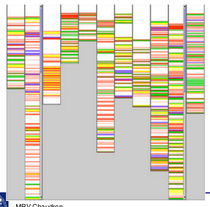
encode/ decode

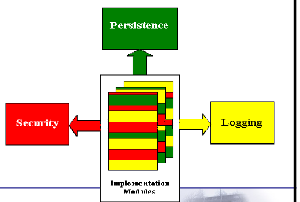
handle & encode/ decode

Aspect Orientation

Design & maintain concerns in isolation

Automatically construct implementation by ‘weaving’ concerns





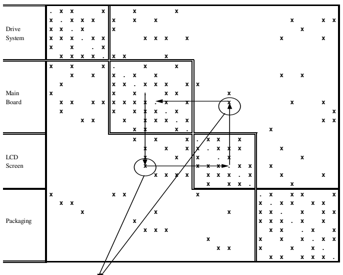
Persistence

Security

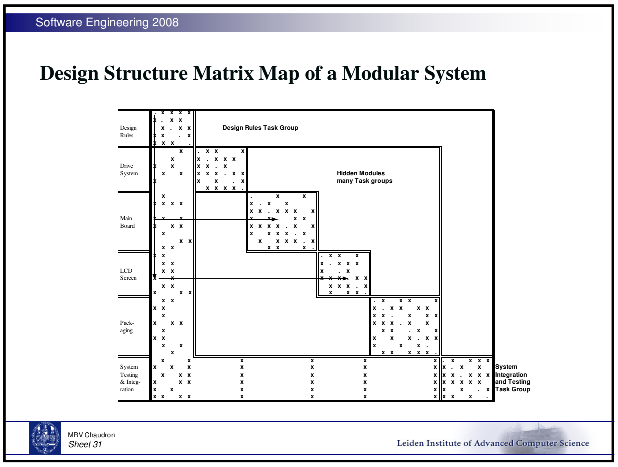
Logging

Implementation Modules

Design Structure Matrix Map of a Laptop Computer

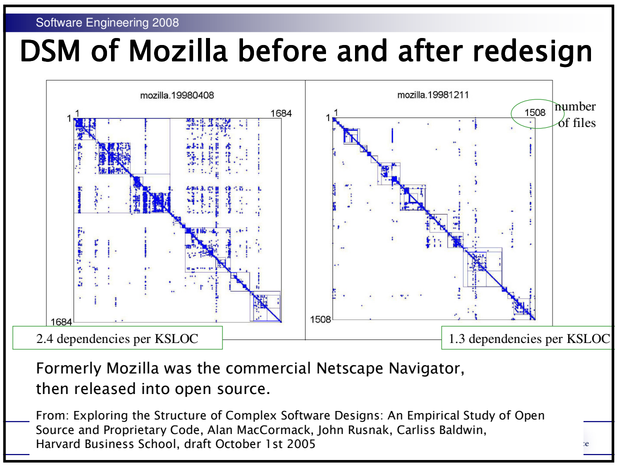


Graphics controller on Main Board or not?
If yes, screen specifications change;
If no, CPU must process more; adopt different interrupt protocols



Design Structure Matrix Map of a Modular System

Design Rules
Design Rules Task Group
Drive System
Hidden Modules many Task groups
Main Board
LCD Screen
Pack-aging
System Testing & Integ-ration
System Integration and Testing Task Group



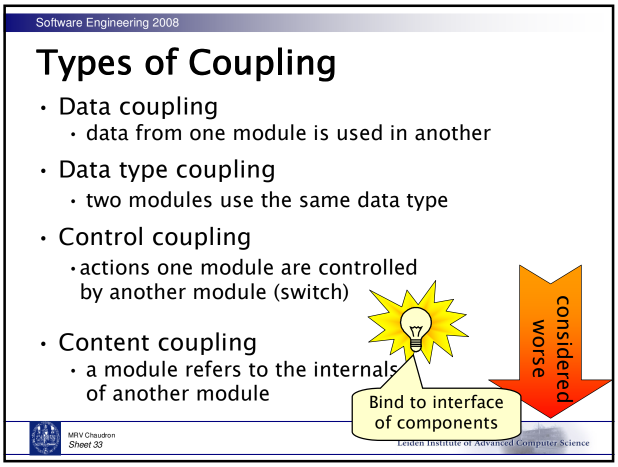
DSM of Mozilla before and after redesign

2.4 dependencies per KSLOC

1.3 dependencies per KSLOC

Formerly Mozilla was the commercial Netscape Navigator, then released into open source.

From: Exploring the Structure of Complex Software Designs: An Empirical Study of Open Source and Proprietary Code, Alan MacCormack, John Rusnak, Carliss Baldwin, Harvard Business School, draft October 1st 2005

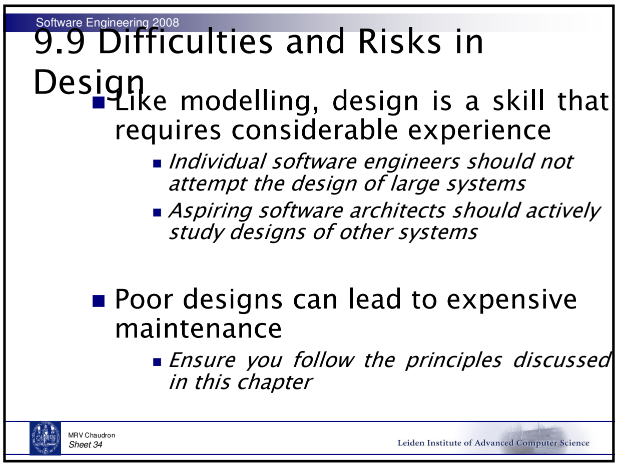


Types of Coupling

- Data coupling
  - data from one module is used in another
- Data type coupling
  - two modules use the same data type
- Control coupling
  - actions one module are controlled by another module (switch)
- Content coupling
  - a module refers to the internals of another module

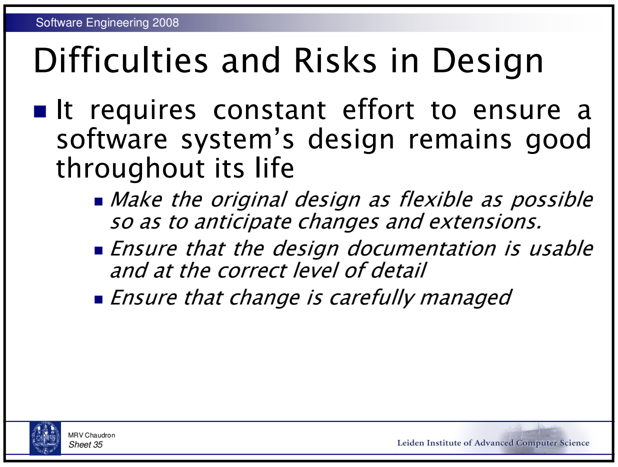
Bind to interface of components

considered worse



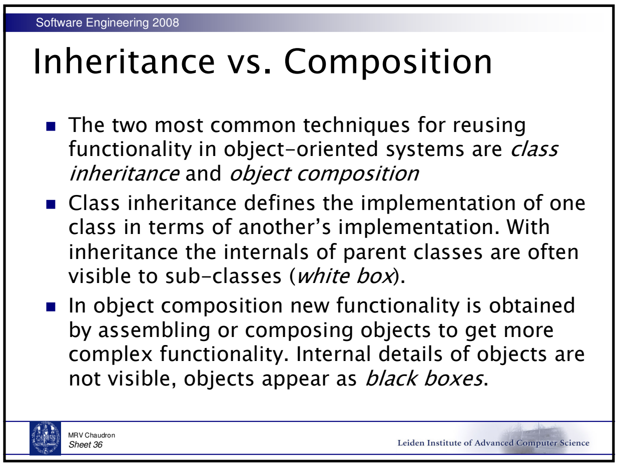
9.9 Difficulties and Risks in Design

- Like modelling, design is a skill that requires considerable experience
  - *Individual software engineers should not attempt the design of large systems*
  - *Aspiring software architects should actively study designs of other systems*
- Poor designs can lead to expensive maintenance
  - *Ensure you follow the principles discussed in this chapter*



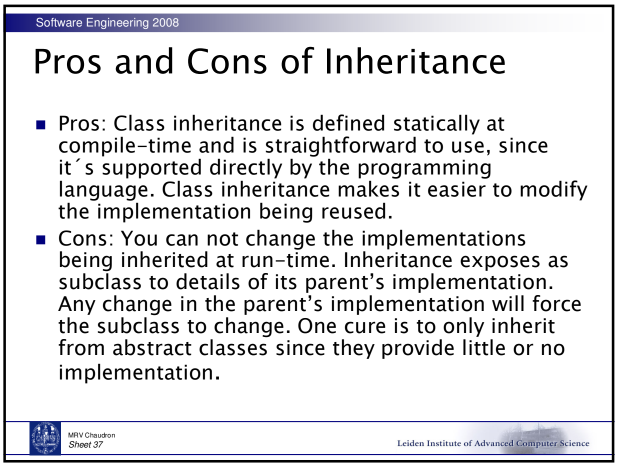
Difficulties and Risks in Design

- It requires constant effort to ensure a software system's design remains good throughout its life
  - *Make the original design as flexible as possible so as to anticipate changes and extensions.*
  - *Ensure that the design documentation is usable and at the correct level of detail*
  - *Ensure that change is carefully managed*



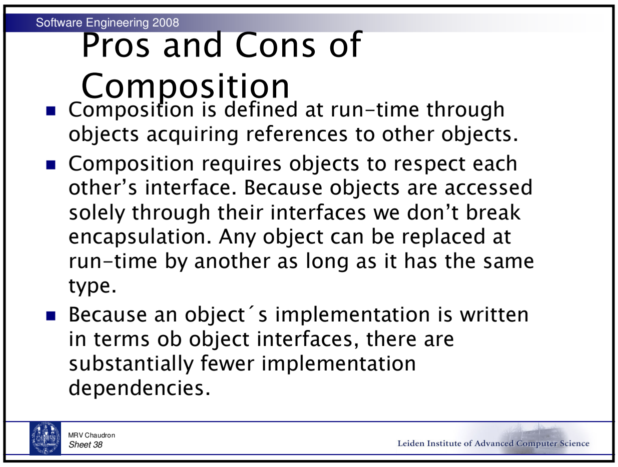
Inheritance vs. Composition

- The two most common techniques for reusing functionality in object-oriented systems are *class inheritance* and *object composition*
- Class inheritance defines the implementation of one class in terms of another's implementation. With inheritance the internals of parent classes are often visible to sub-classes (*white box*).
- In object composition new functionality is obtained by assembling or composing objects to get more complex functionality. Internal details of objects are not visible, objects appear as *black boxes*.



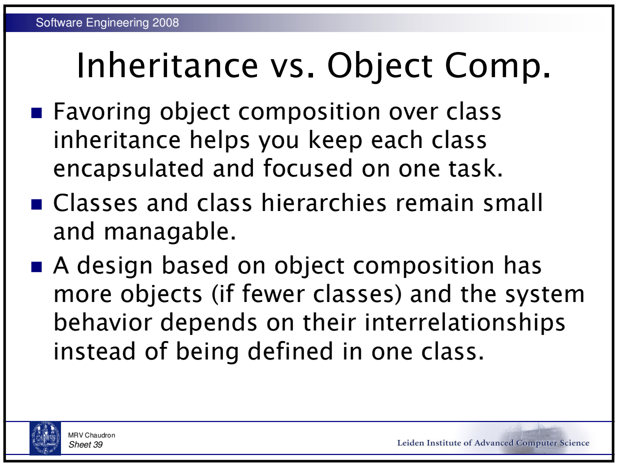
Pros and Cons of Inheritance

- Pros: Class inheritance is defined statically at compile-time and is straightforward to use, since it´s supported directly by the programming language. Class inheritance makes it easier to modify the implementation being reused.
- Cons: You can not change the implementations being inherited at run-time. Inheritance exposes as subclass to details of its parent's implementation. Any change in the parent's implementation will force the subclass to change. One cure is to only inherit from abstract classes since they provide little or no implementation.



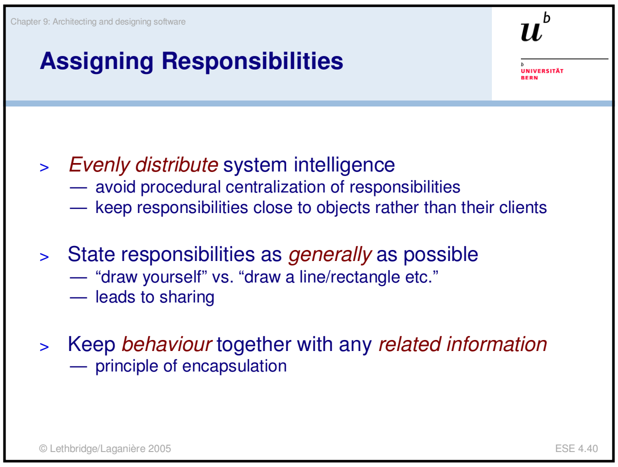
Pros and Cons of Composition

- Composition is defined at run-time through objects acquiring references to other objects.
- Composition requires objects to respect each other's interface. Because objects are accessed solely through their interfaces we don't break encapsulation. Any object can be replaced at run-time by another as long as it has the same type.
- Because an object´s implementation is written in terms ob object interfaces, there are substantially fewer implementation dependencies.



Inheritance vs. Object Comp.

- Favoring object composition over class inheritance helps you keep each class encapsulated and focused on one task.
- Classes and class hierarchies remain small and managable.
- A design based on object composition has more objects (if fewer classes) and the system behavior depends on their interrelationships instead of being defined in one class.

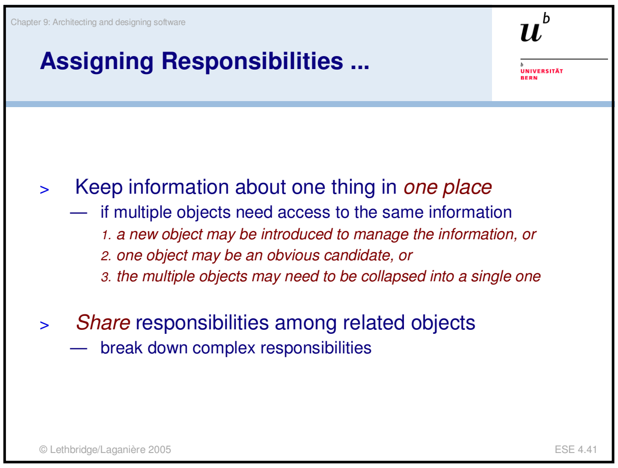


Assigning Responsibilities

> *Evenly distribute* system intelligence
> — avoid procedural centralization of responsibilities
> — keep responsibilities close to objects rather than their clients

> State responsibilities as *generally* as possible
> — "draw yourself" vs. "draw a line/rectangle etc."
> — leads to sharing

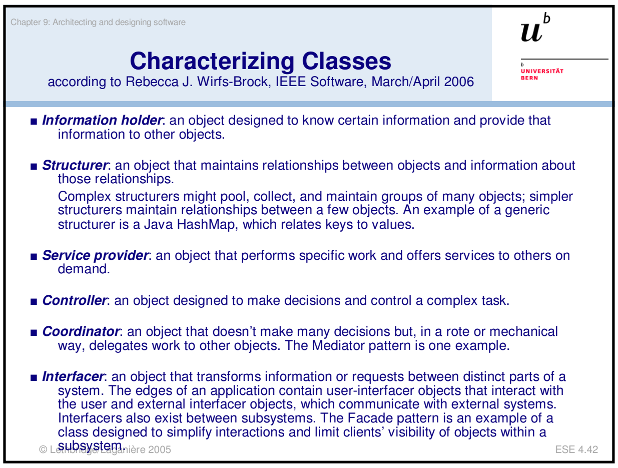
> Keep *behaviour* together with any *related information*
> — principle of encapsulation



Assigning Responsibilities ...

> Keep information about one thing in *one place*
> — if multiple objects need access to the same information
> 1. *a new object may be introduced to manage the information, or*
> 2. *one object may be an obvious candidate, or*
> 3. *the multiple objects may need to be collapsed into a single one*

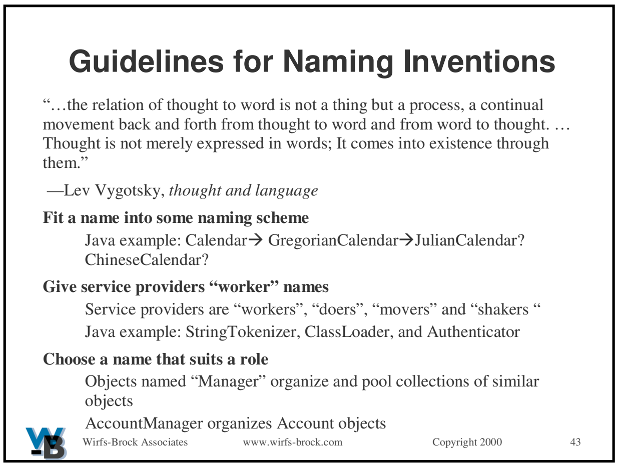
> *Share* responsibilities among related objects
> — break down complex responsibilities



Characterizing Classes

according to Rebecca J. Wirfs-Brock, IEEE Software, March/April 2006

- ***Information holder***: an object designed to know certain information and provide that information to other objects.
- ***Structurer***: an object that maintains relationships between objects and information about those relationships.
  Complex structurers might pool, collect, and maintain groups of many objects; simpler structurers maintain relationships between a few objects. An example of a generic structurer is a Java HashMap, which relates keys to values.
- ***Service provider***: an object that performs specific work and offers services to others on demand.
- ***Controller***: an object designed to make decisions and control a complex task.
- ***Coordinator***: an object that doesn't make many decisions but, in a rote or mechanical way, delegates work to other objects. The Mediator pattern is one example.
- ***Interfacer***: an object that transforms information or requests between distinct parts of a system. The edges of an application contain user-interfacer objects that interact with the user and external interfacer objects, which communicate with external systems. Interfacers also exist between subsystems. The Facade pattern is an example of a class designed to simplify interactions and limit clients' visibility of objects within a subsystem.



Guidelines for Naming Inventions

"…the relation of thought to word is not a thing but a process, a continual movement back and forth from thought to word and from word to thought. … Thought is not merely expressed in words; It comes into existence through them."

—Lev Vygotsky, *thought and language*

Fit a name into some naming scheme

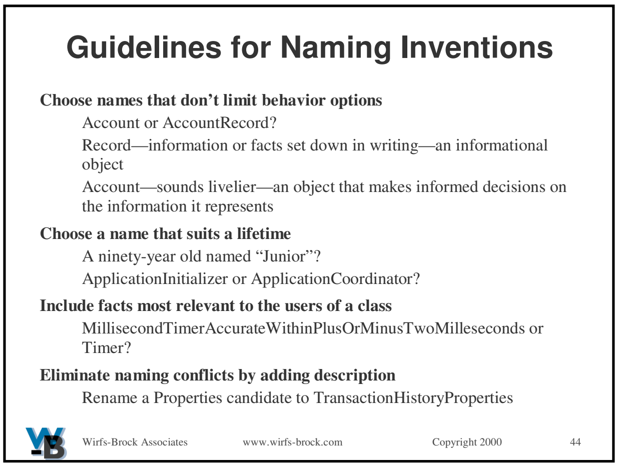
- Java example: Calendar→ GregorianCalendar→JulianCalendar? ChineseCalendar?

Give service providers "worker" names

- Service providers are "workers", "doers", "movers" and "shakers "
- Java example: StringTokenizer, ClassLoader, and Authenticator

Choose a name that suits a role

- Objects named "Manager" organize and pool collections of similar objects
- AccountManager organizes Account objects



Guidelines for Naming Inventions

Choose names that don't limit behavior options

- Account or AccountRecord?
- Record—information or facts set down in writing—an informational object
- Account—sounds livelier—an object that makes informed decisions on the information it represents

Choose a name that suits a lifetime

- A ninety-year old named "Junior"?
- ApplicationInitializer or ApplicationCoordinator?

Include facts most relevant to the users of a class

- MillisecondTimerAccurateWithinPlusOrMinusTwoMilleseconds or Timer?

Eliminate naming conflicts by adding description

- Rename a Properties candidate to TransactionHistoryProperties