

Design Guidelines for Software Components

M.R.V. Chaudron

M.R.V.Chaudron@tue.nl

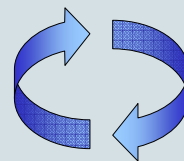
www.win.tue.nl/~mchaudro/

Technische Universiteit Eindhoven

1

Topics

- Guidelines for **designing** components
- Notions of coupling and cohesion
- Design guidelines for components and interfaces



2

Language shapes the way we think

- The concepts we know, determine the way we think
- Any component model we select, provides a set of concepts, but also fails to provide some others.

Civil engineering: building constructions out of stone bricks could make pyramids or 4 storey buildings, but no sky-scrapers.

Software Engineers are indoctrinated with the client/server (request/response) model of interaction. There are other options ...

3

Component Design

- How should we scope the functionality of components?
- How big should a component be?
 - Which functions should be grouped together
 - Which data should be grouped by which functions?
- How complex should a component be?
- How can we minimize interaction between components?

4

What is Modularity?

We can “see it” via a
Design Structure Matrix (DSM) Map

5

What is a dependency?

- Component A requires B for it to *work*
 - Functional coupling
- A change in module B requires change in module A
 - Implementation coupling
 - Typically requires: re-testing A & B

6

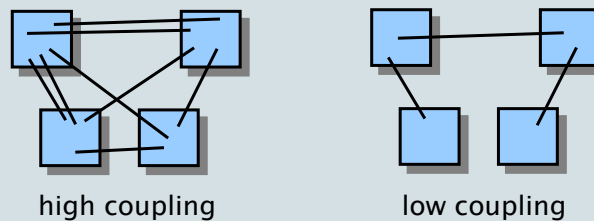
- There is coupling between two classes **A** and **B** if:
 - **A** has an attribute that refers to (is of type) **B**.
 - **A** calls on services of an object **B**.
 - **A** has a method which references **B** (via return type or parameter).
 - **A** is a subclass of (or implements) class **B**.

This is not an exhaustive definition

7

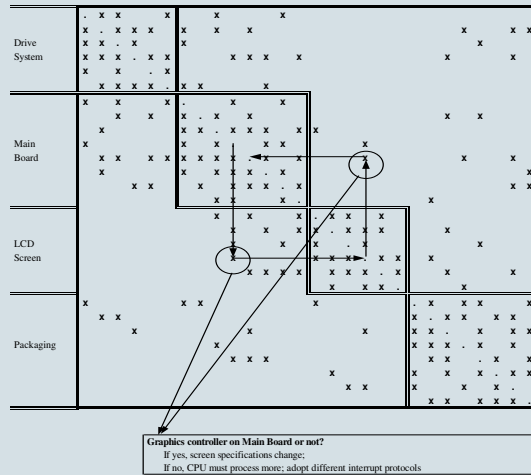
Dependency: Coupling

Coupling is the degree of interdependence between modules



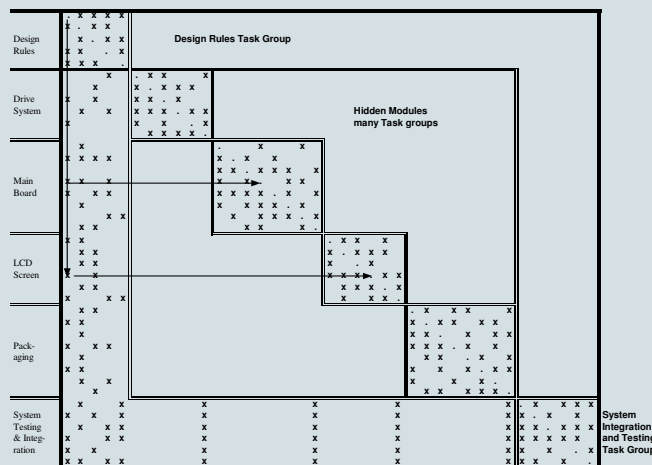
8

Design Structure Matrix Map of a Laptop Computer



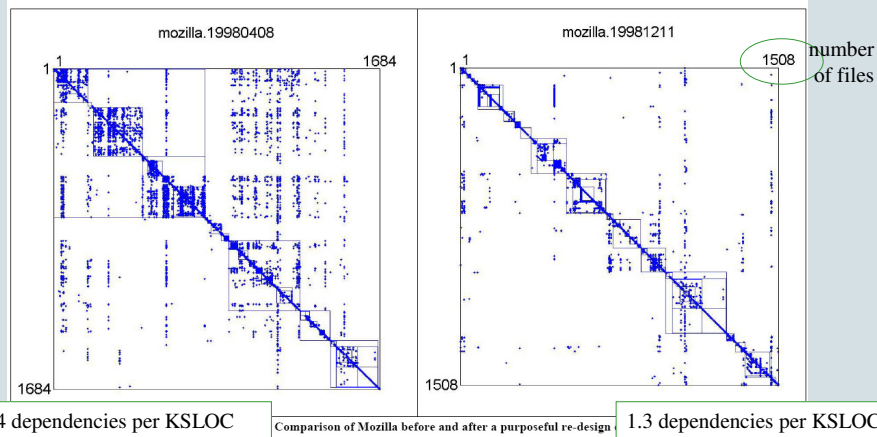
9

Design Structure Matrix Map of a Modular System



10

DSM of Mozilla before and after redesign



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

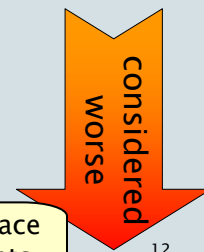
11

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



12

Modularity & Coupling violations exorcised at Prog. Lang's level

- Alteration of code of other components
 - The 'alter' statement in COBOL
- Entering half-way a body
 - FORTRAN provided multiple entry-points into a module
 - ENTRY statement
 - Assembler
 - Jumps to an arbitrary address
 - GOTO considered harmful (Dijkstra)
 - ➔ structured programming, single point of entry & exit

13

Temporal Coupling

A component X expects an input from component Y
every second.

A component should handle all cases where attempts are made to use it inappropriately (be it intentionally or not).

A RT-component should have a fall-back scenario:

If I don't receive an input, then I do 'plan B'.

So that other components that depend on X will not also have to deal with this problem.

This is a way of 'fault containment' – prevent domino-effect.

14

Temporal Coupling

Program A

```
...
{
  ...
  openfile(data)
  do_long_processing(data);
  closefile(data)
  ...
}
```

Program B

```
...
{
  ...
  openfile(data)
  closefile(data)
  do_long_processing(data);
  ...
}
```

15

Inheritance = Coupling

99.9% of the time I find that it makes sense to assign all of the classes of a hierarchy, either an inheritance hierarchy or a composition hierarchy, to the same component.

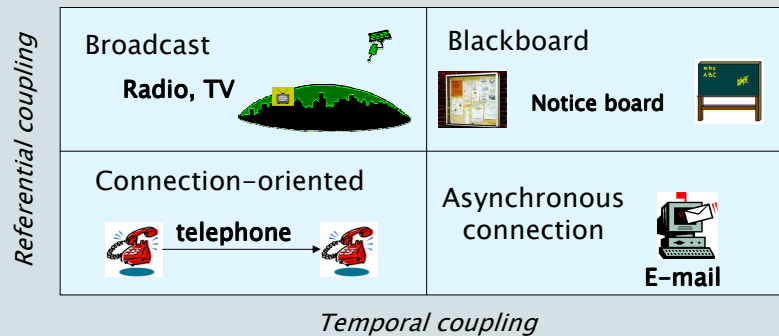
Scott Ambler

Inheritance implies cohesiveness

16

Coupling induced by Interaction Style

Component Models often use one particular interaction style.
Interaction styles imply some type of coupling!



Referential coupling: sender has reference to receiver's name

Temporal coupling: sender and receiver synchronize in time

17

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

18

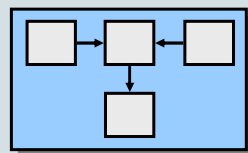
Benefits of Low Coupling/Dependencies

1. fewer interconnections between modules reduce the chance that a fault in one module will cause a **failure** in other modules;
2. fewer interconnections between modules reduce the chance that **changes** in one module cause **problems** in other modules, which enhances reusability; and
3. fewer interconnections between modules reduce programmer time in **understanding** the details of other modules.

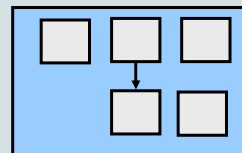
Page-Jones, M. 1980. *The Practical Guide to Structured Systems Design*. New York, Yourdon Press, 1980.

Cohesion

Cohesion is concerned with the dependencies within a module



high cohesion



low cohesion

Heuristic:

Keep things together that belong together
High cohesion within a module is good

20

Characteristics of Components: Cohesive

conceptually whole

- size depends on
 - complexity of the concept
 - abstraction level of the concept

related: unit of abstraction,
abstraction of a design decision

If components were people and computers didn't exist,
one might ask, "Who is responsible for this task?"

21

Types of Cohesion

Functional cohesion:

a module performs a single well-defined task

Communicational/Data cohesion:

a module performs multiple functions on the same data

Heuristic: Describe the purpose of a component in a single
sentence using a single verb and a single subject

22

Bad Types of Cohesion

Temporal cohesion:

a module performs a set of functions that must occur in a limited/continuous time-span.

Utility cohesion:

a module contains a set of similar utilities
e.g. output to screen + output to printer + output to file
problem: units may change independently

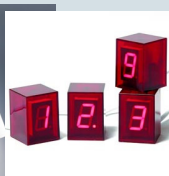
23

Radio Alarm Clock

What should be the responsibility & interface of each component?



lamp

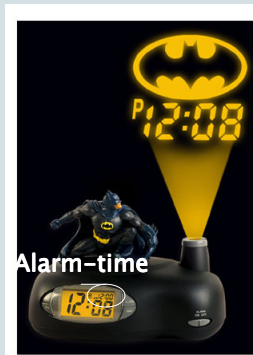


(wireless) atomic clock



temperature

CD



Alarm-time

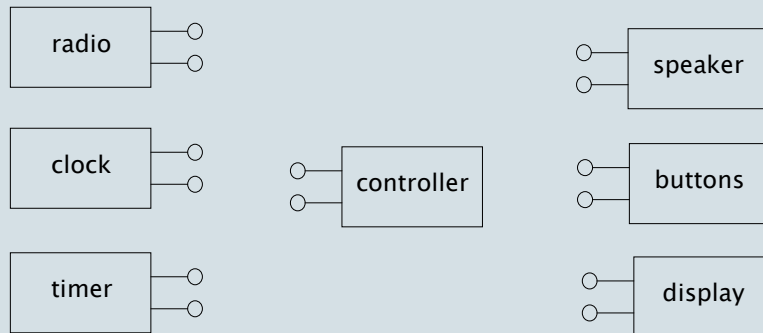
Bat-alarm

Train strike/traffic delays

24

Radio Alarm Clock

What should be the responsibility & interface of each component?



25

Alternative Interfaces

- Traffic Light



- Level of abstraction

26



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()

Traffic Light1'

- SwitchNextColour(c)

Traffic Light2'

- SetColour(c)

Malfunction?

'Secrets'

- Initial state
- Order of states
(easy to change)

'Secrets'

- Initial state
- Order of states

'Secrets'

- Initial state
- Order of states

Synchronisation/Timing?

More Generic
(lights not exclusive)

Higher Level of
abstraction

27

Guidelines for Interface Design (1)

- *Consistency*
 - applies to many aspects of interface design such as naming conventions, parameter passing and exception handling.
- *Essential/Minimal:*
 - omit needless features.
- *Orthogonality:*
 - Keep independent features separately
 - Avoid offering the same service in multiple ways.
- *General:*
 - do not limit the applicability of an interface to its initial purpose as modules may be used in unexpected ways.
- *Completeness:*
 - include all functions
- *Open-ended:*
 - leave room for future expansion.
- *Opacity/Information-hiding:*
 - an interface should hide the details of the implementation.

28

Based on Hoffman [Hof90] based on o.a. Parnas.

Guidelines for Interface Design (2)

1. Keep interfaces *cohesive* and *small* (in that order)
2. Use *different interfaces* for users of the interface that play *different roles* with respect to the functionality
3. Don't combine *generic* and *specific* functionality in the same interface
4. Group *optional* functionality in *separate* interfaces
5. Avoid the introduction of *convenience functions*
6. Use *strongly typed* interfaces
7. Use *systematic naming* conventions

From Henk Jonkers c.s., Philips Research 2002 29

Interface Suites

For optimizing reusability, it may be beneficial to specify **suites** of interfaces.

I.e. sets of interfaces that capture common

- functionality
- patterns of collaboration

in a domain.

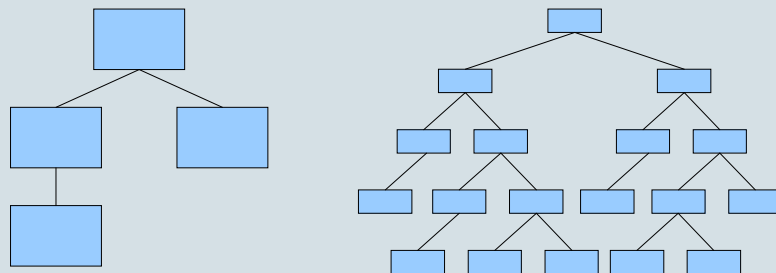
An interface can be related to a **role**.
Components can implement one or more roles

30

Component Design Heuristics

31

Trade-off between Size and Interactions

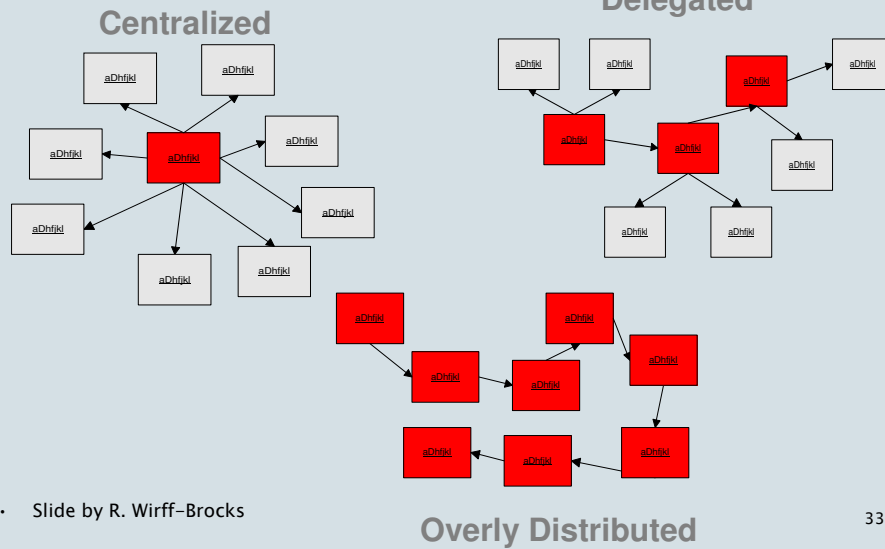


Module should be understandable

- single abstraction
- low complexity

32

Three Control Styles



• Slide by R. Wirff-Brocks

33

Questions?

Later: design for testability

34

Components & State

Avoid state where possible
→ easier to replace

Suppose state is present within a component.
Then how is this state transferred to its replacement.

Replace a tic-tac-toe playing component

- one strategy 'assumes' that it has made some other moves earlier
- a better strategy is to look at the state of the board and compute the best move for that situation

35

Autonomy & Dependencies

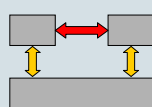
Two sides of the same coin:

- maximize autonomy ⇔ minimize dependencies

self-contained

Adding, replacing & removing components all deal with making & breaking of dependencies.

To make 'life' easier focus should be on **avoiding** dependencies.



Can be avoided by using a good component model and good design



Can never be completely avoided

36

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

37

Benefits of Low Coupling/Dependencies

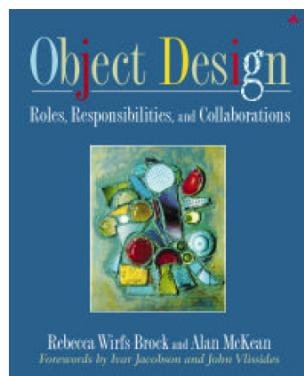
1. fewer interconnections between modules reduce the chance that a fault in one module will cause a **failure** in other modules;
2. fewer interconnections between modules reduce the chance that **changes** in one module cause **problems** in other modules, which enhances reusability; and
3. fewer interconnections between modules reduce programmer time in **understanding** the details of other modules.

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

Scoping guidelines for components

39

Resources



www.wirfs-brock.com for
articles, resources,
training, and consulting
rebecca@wirfs-brock.com
alan@wirfs-brock.com

40

What To Look For

Look for inventions that represent:

- The *work* your software performs
- The *things* your software affects or is connected to
- The *information* that flows through your software
- Your software's *decision-making, control* and *coordination* activities
- Ways to *structure* and *manage* groups of objects
- Representations of *real world things* your software needs to know something about



Copyright 2002, Wirfs-Brock Associates, Inc.

24

41

Role Stereotypes

Doing, Knowing and Deciding

Stereotypes are simplified views that help us characterize object essentials. Use them to characterize the roles objects play in an application

Service providers do things: a Mailer

Interfacers translate requests and convert from one level of abstraction to another: Presenter and Selector

Information holders know things: Letters, Words, Sentences

Controllers direct activities: the Guesser

Coordinators delegate work: the MessageBuilder

Structurers manage object relations: Objects that manage Letters, Words, Sentences and Commands



Copyright 2002, Wirfs-Brock Associates, Inc.

25

42