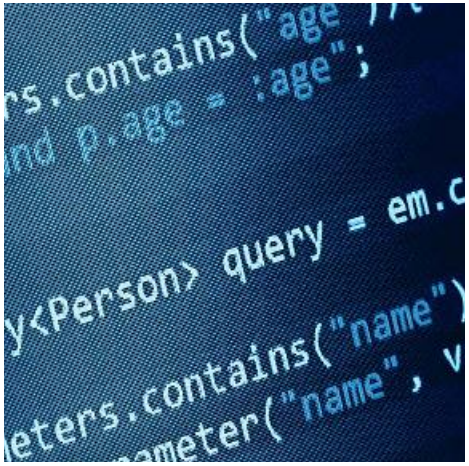
CS3500 Software Engineering

2017/2018



Dept. Computer Science
Dr. Klaas-Jan Stol

UCC
University College Cork, Ireland
Coláiste na hOllscoile Corcaigh

Welcome to CS3500

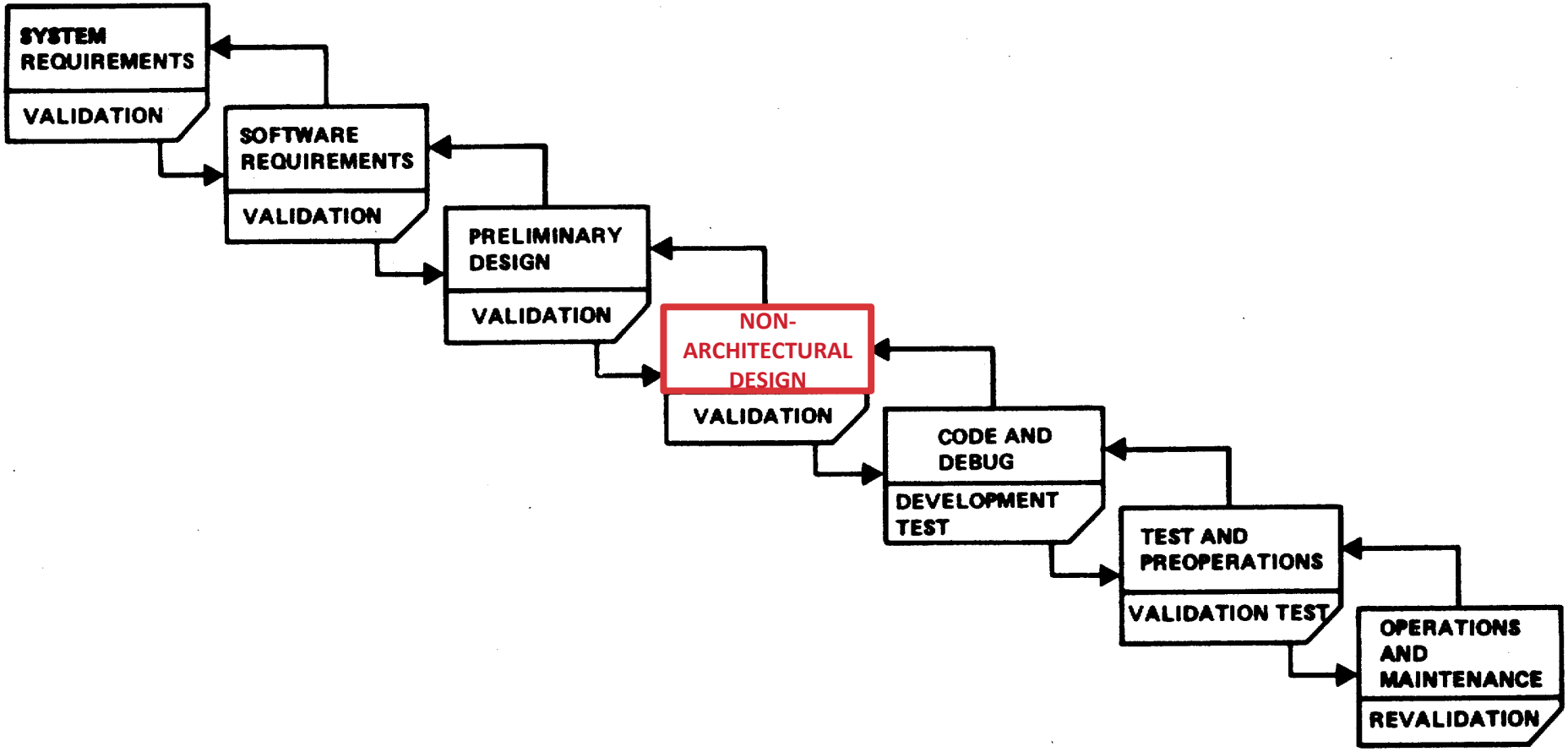
Software Design

CS3500 – Software Engineering – Software Design
Version 2017.10.13-002 k.stol@ucc.ie

After studying this material and associated papers, you should be able to:

- Understand what is meant by “design.”
- Understand, describe, and apply general principles for software design:
  - Modularization
  - Low coupling, high cohesion
  - Abstraction
  - Information hiding

Scope of this lecture



Contents

1. Definitions (or, what is design?)
2. Design Rules

Assumed knowledge of Object-Oriented programming as covered in CS2514 Introduction to Java

SECTION I

Definitions

1.

Design

Definitions

Design

A specification of an object, manifested by some agent, intended to accomplish goals, in a particular environment, using a set of primitive components, satisfying a set of requirements, subject to some constraints.

Specification, goals, environment, components, requirements, constraints

SECTION II

Design Rules

There are no fixed rules to sound design. There is no foolproof formula for good design. Instead, there are some general principles

Design methods

There are many design methods and notations available, but:

- There is no fixed set of steps to follow that lead to a perfect design.
- There is no perfect notation that is appropriate in all cases.
- There is no “fool proof” approach to software design.

Software design is hard.

General principles of software design

- Modularization
- Abstraction
- Information Hiding
- Coupling & Cohesion

Note:

There is no such thing as “the” or a “final” set of principles. The ones presented here are widely accepted. If you stick to these, you’ll have a good foundation.

Modularization
also known as Separation of Concerns

A module should have only a single responsibility.

If a module has more than one responsibility, then the responsibilities become coupled.

Changes to one responsibility may inhibit module's ability to meet other responsibilities.

"Just because you can, doesn't mean you should."

Modularization
also known as Separation of Concerns

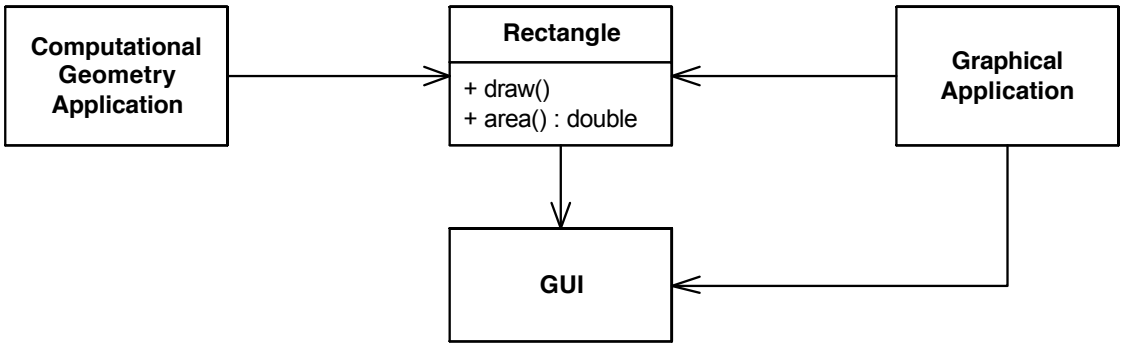
- Concerned with meaningful decomposition of a system.

- Modules are containers for functionality or responsibilities.

- Why modularization is good:
  1. Handle system complexity by introducing well-defined boundaries (interfaces)
  2. Supports software reuse.
  3. Improve system maintainability.
  4. Supports replacing modules with alternatives.
  5. Support task decomposition (team work)

EXAMPLE

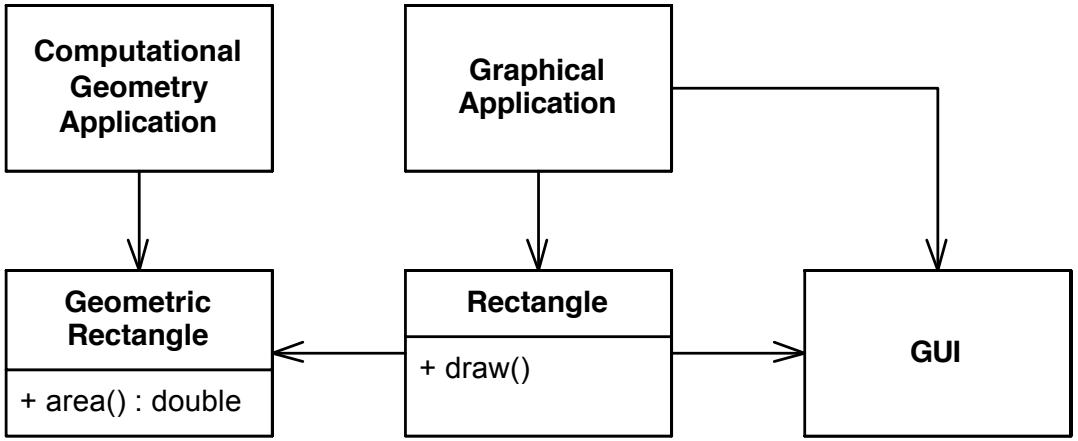
Modularity



Top:
Rectangle has responsibilities to:

- Draw itself
- Calculate its area

These are used by 2 different programs. Thus, the initial design violates the principle of modularity, because it has more than one responsibility.



Bottom:
Solution: split up responsibilities into 2 different modules.

Information hiding (black box)

A module should expose its functionality through an interface, but hide its implementation details.

Why?

- Increases maintainability.
- Increases reuse.
- Supports task decomposition (team work).

> Every module is characterized by its knowledge of a design decision which it hides from all others.
>
> —David L. Parnas

EXAMPLE

Information hiding: bad example

```
// Java method to store Irish customer address.
// BEFORE introduction of Eircode!
void store_customer(String name,
                    String address1,
                    String address2,
                    String address3) {
...
}

// After adding Eircode. Have to change all code
// that invokes store_customer()!
void store_customer(String name,
                    String address1,
                    String address2,
                    String address3,
                    String eircode) {
...
}
```

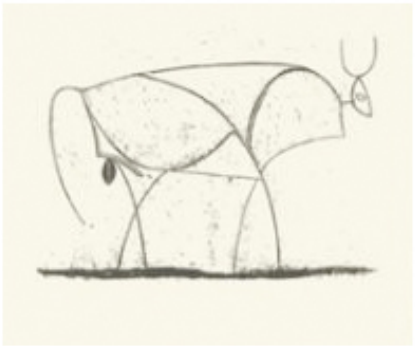
EXAMPLE

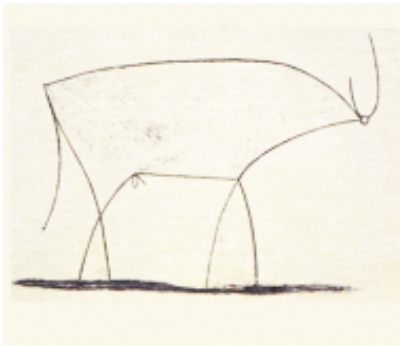
Information hiding: better example

```
class Customer {
    String name;
    String address1, address2, address3;
    String eircode;
    ...
}

// Hide information in a class.
void store_customer(Customer cData) {
    ...
}

// No change needed to interface. Yay!
void store_customer(Customer cData) {
    ...
    this.eircode = cData.eircode;
}
```

Abstraction: Picasso's Bull

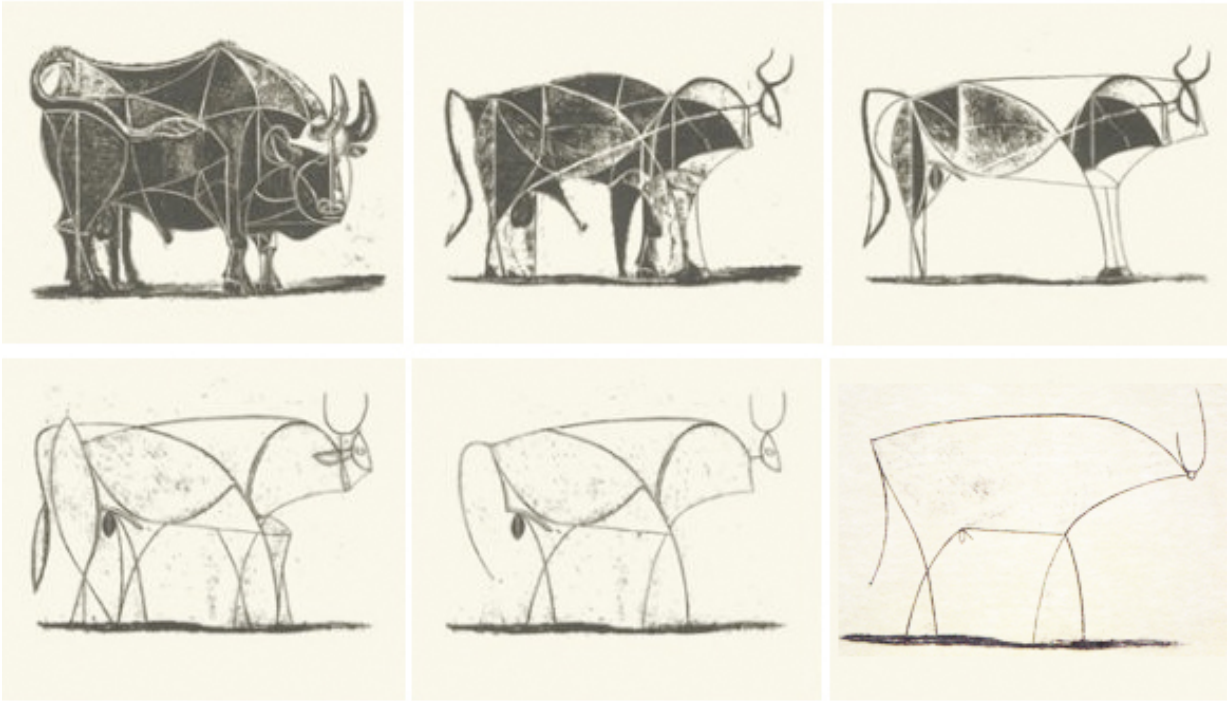
Abstraction: Picasso's Bull





Abstraction: Picasso's Bull

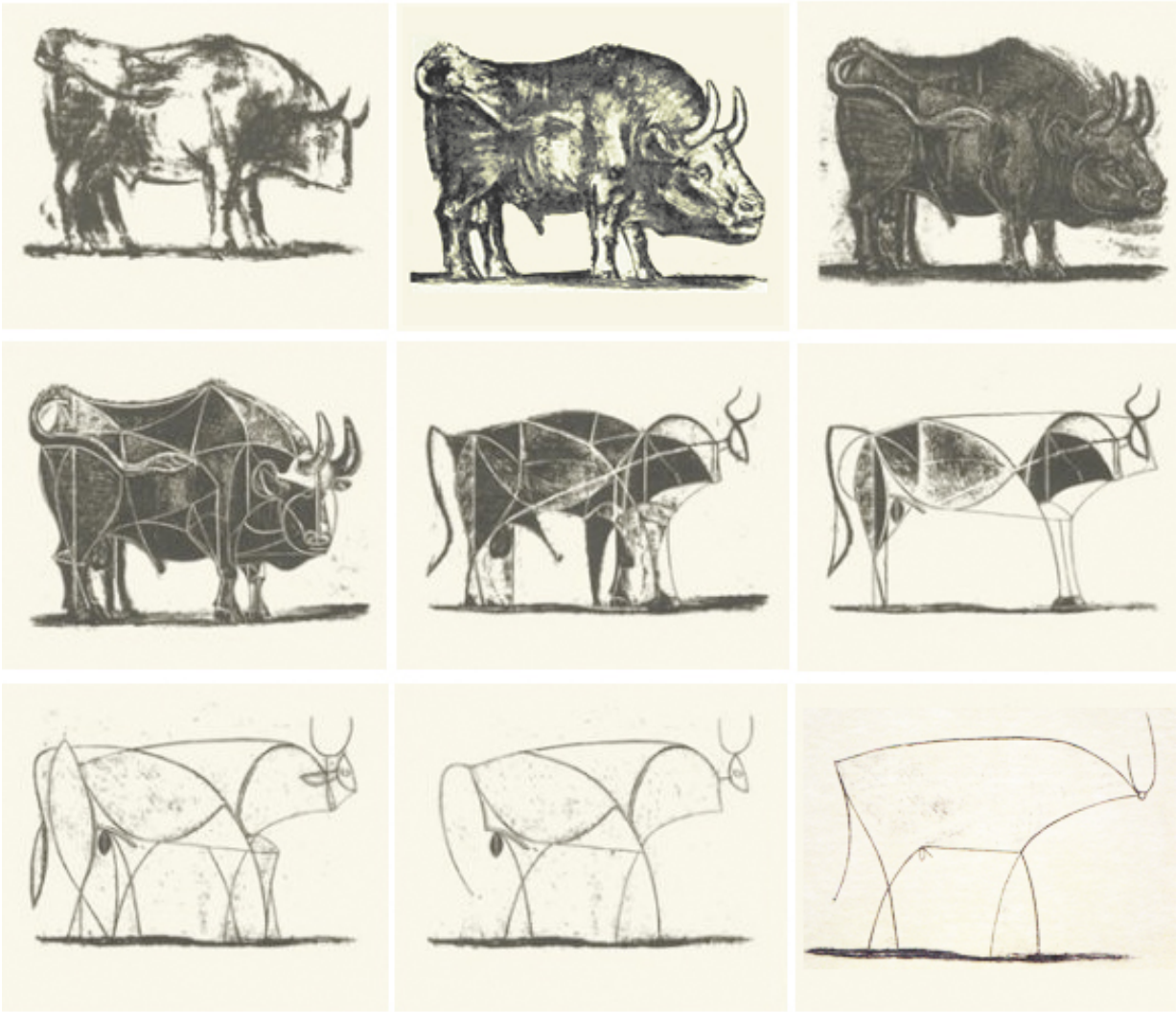
Abstraction: Picasso's Bull



Abstraction: Picasso's Bull

Essence of Picasso's Bull is 4 pen strokes.

Not easy:
Had to draw the bull in detail first to understand.

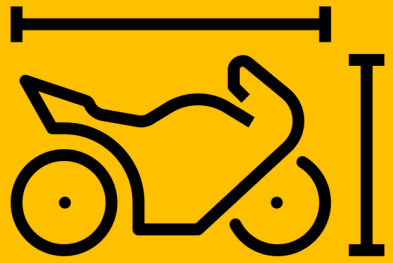


Abstraction

The process of forgetting information, so that things that are different can be treated as if they were the same.

2 common mechanisms:

Parameterization



Specification

Abstraction by parameterization

Some common mechanisms:

- C++ templates
- Java generics

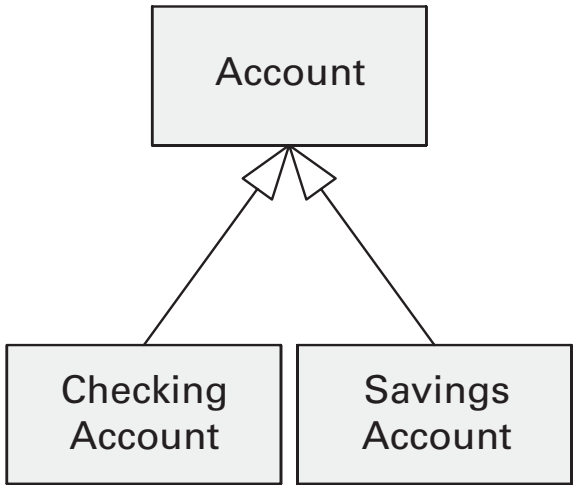
```
// Example of C++ function template.
// when passing in integers, it prints out
// integer result; if passed in floats, it'll
// print a float. This function has the data
// type 'abstracted'.
template<class TYPE>

void print_twice(TYPE data)
{
    cout << "Twice: " << data * 2 << endl;
}
```

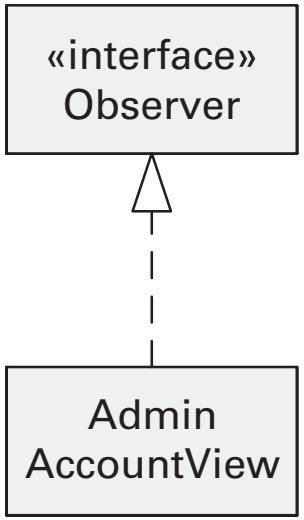
Abstraction by specification

Refers to different implementations. How?

Object-Oriented mechanism: Inheritance

(This includes interfaces & method overriding)





Object-Oriented Inheritance: different things, but both can be treated as an “Account”. Any operation allowed on Account can also be allowed on Checking-Account and Savings-Account.

Interfaces: can give different classes the same type when all implement the same interface. (this mechanism passes on the type)

Why abstraction?

- Helps to prevent duplication of code through reusability.

- Duplication is bad, because a change in one part of the system may lead to changes in other parts.

Other popular & related acronyms: DRY, not WET

- DRY: Don't Repeat Yourself
- WET: Write Everything Twice [don't WET]

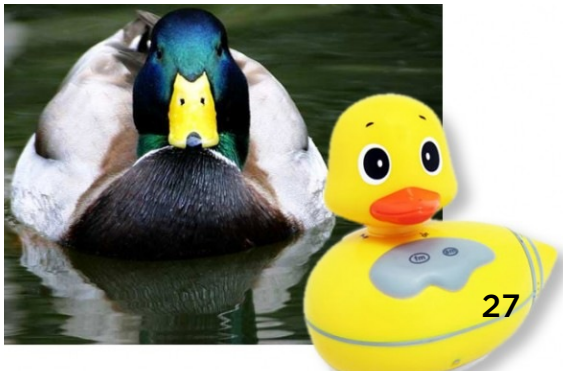
Abstraction in Object-Oriented Designs:
The Liskov substitution principle

Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

Example:

Wherever a Shape object is expected, you should be able to pass in a subclass of Shape – e.g. Triangle, Square, Circle

If it looks like a duck, quacks like a duck, but it needs batteries, you probably have the wrong abstraction.



Coupling & Cohesion

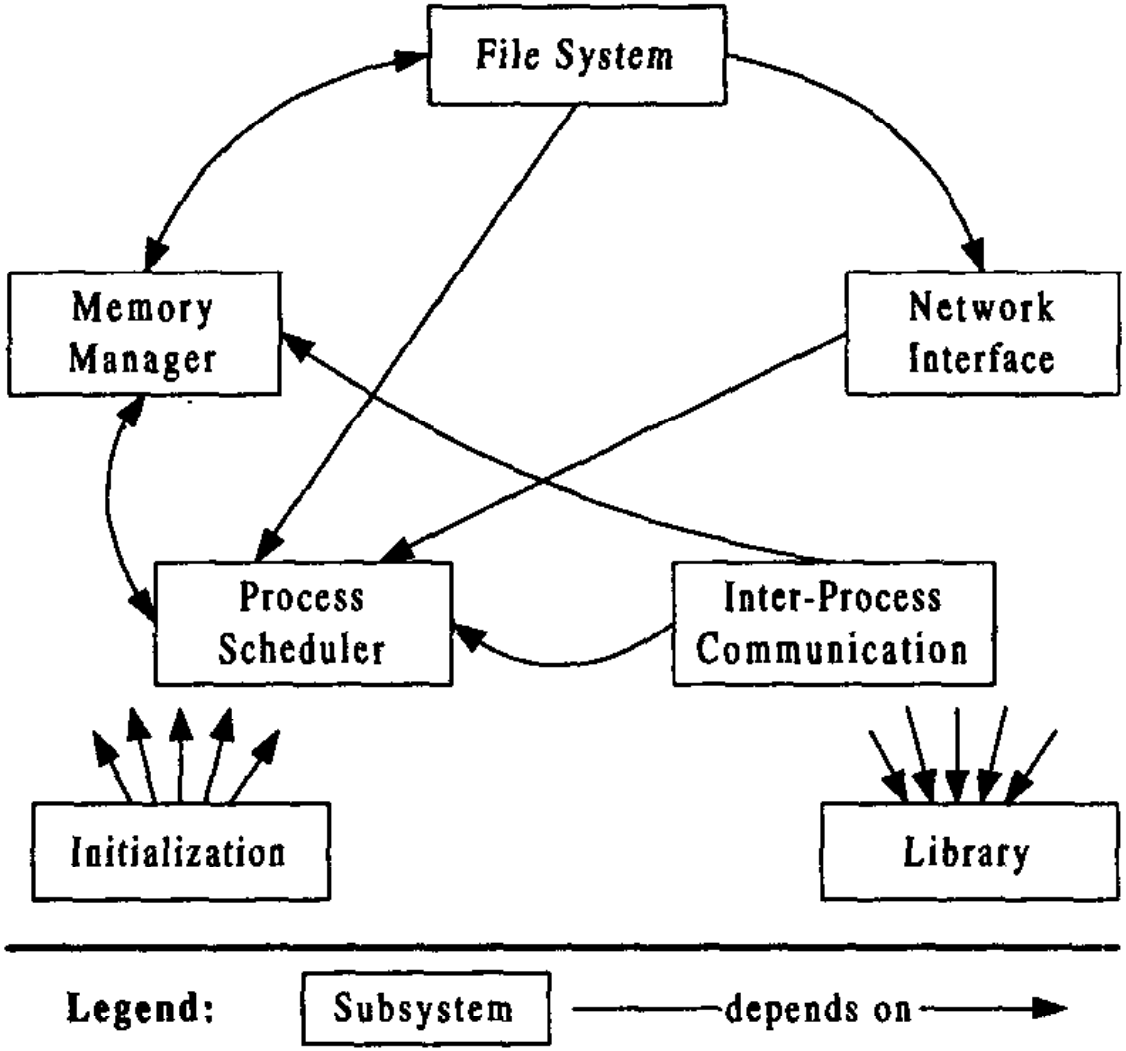
Coupling is a measure of strength of association **between** modules.

- Strong coupling means "many" connections between modules. This reduces maintainability.

- Why you want low coupling:
  - Supports program comprehension.
  - Supports maintainability.
  - Supports task decomposition.

EXAMPLE

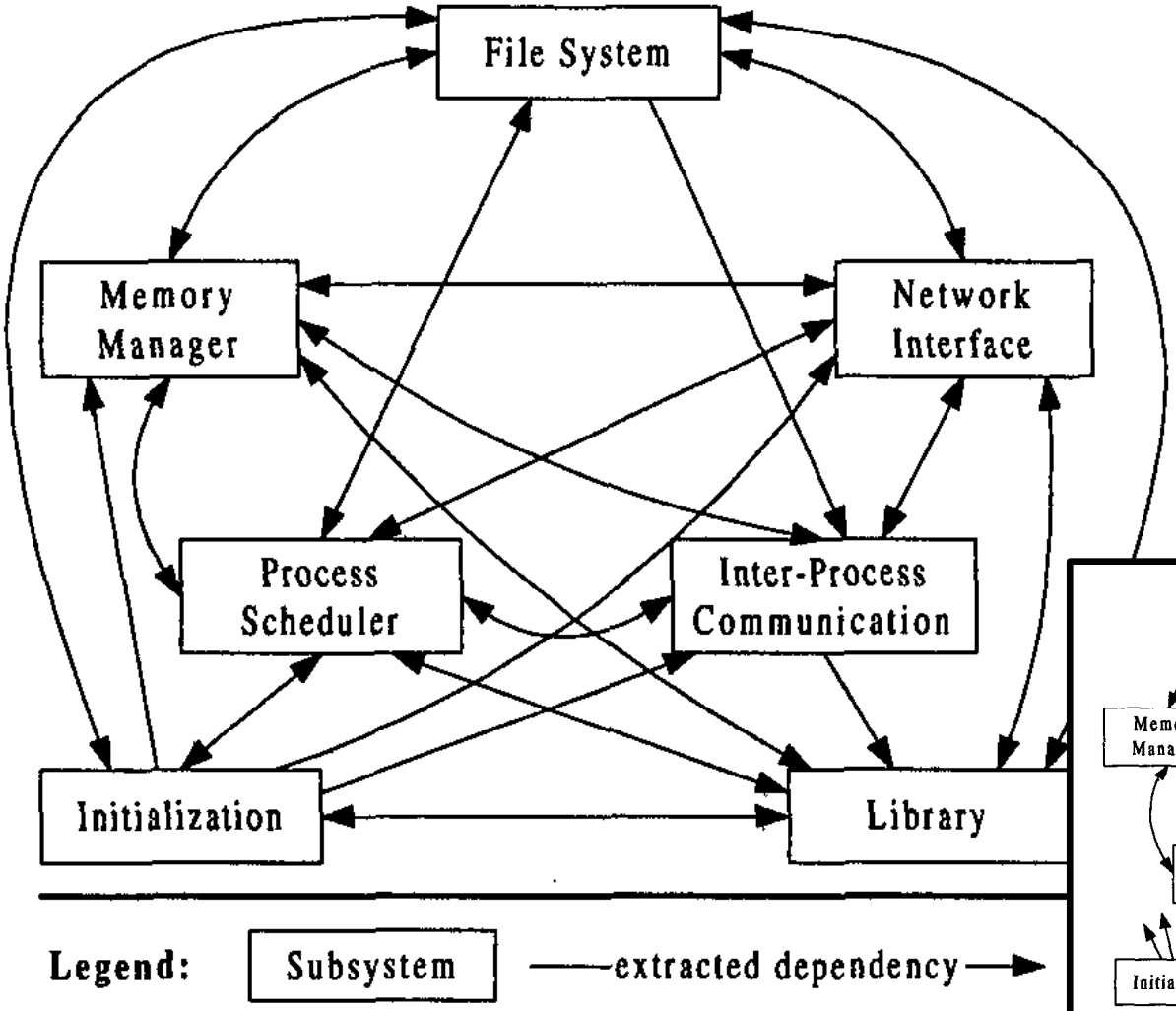
Coupling in the Linux kernel



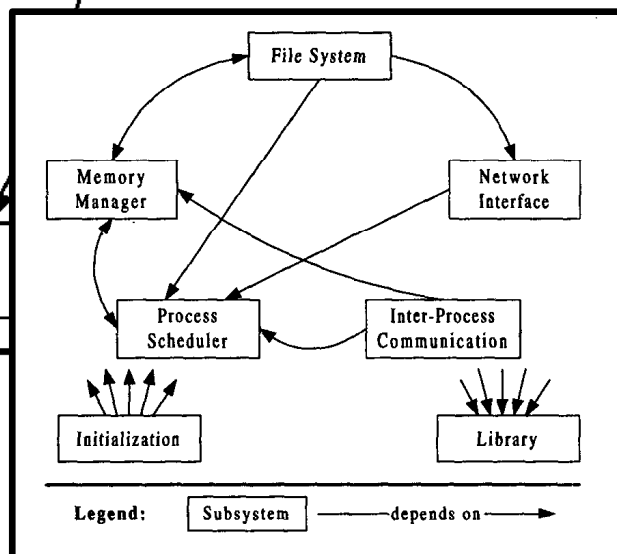
The conceptual (expected) architecture has low coupling between modules. Changes to a module are not likely to require changes elsewhere. Yay!

EXAMPLE

Coupling in the Linux kernel



The concrete architecture is an almost fully connected graph. A change to any module is likely to require changes in dependent modules! Boo!



Coupling & Cohesion

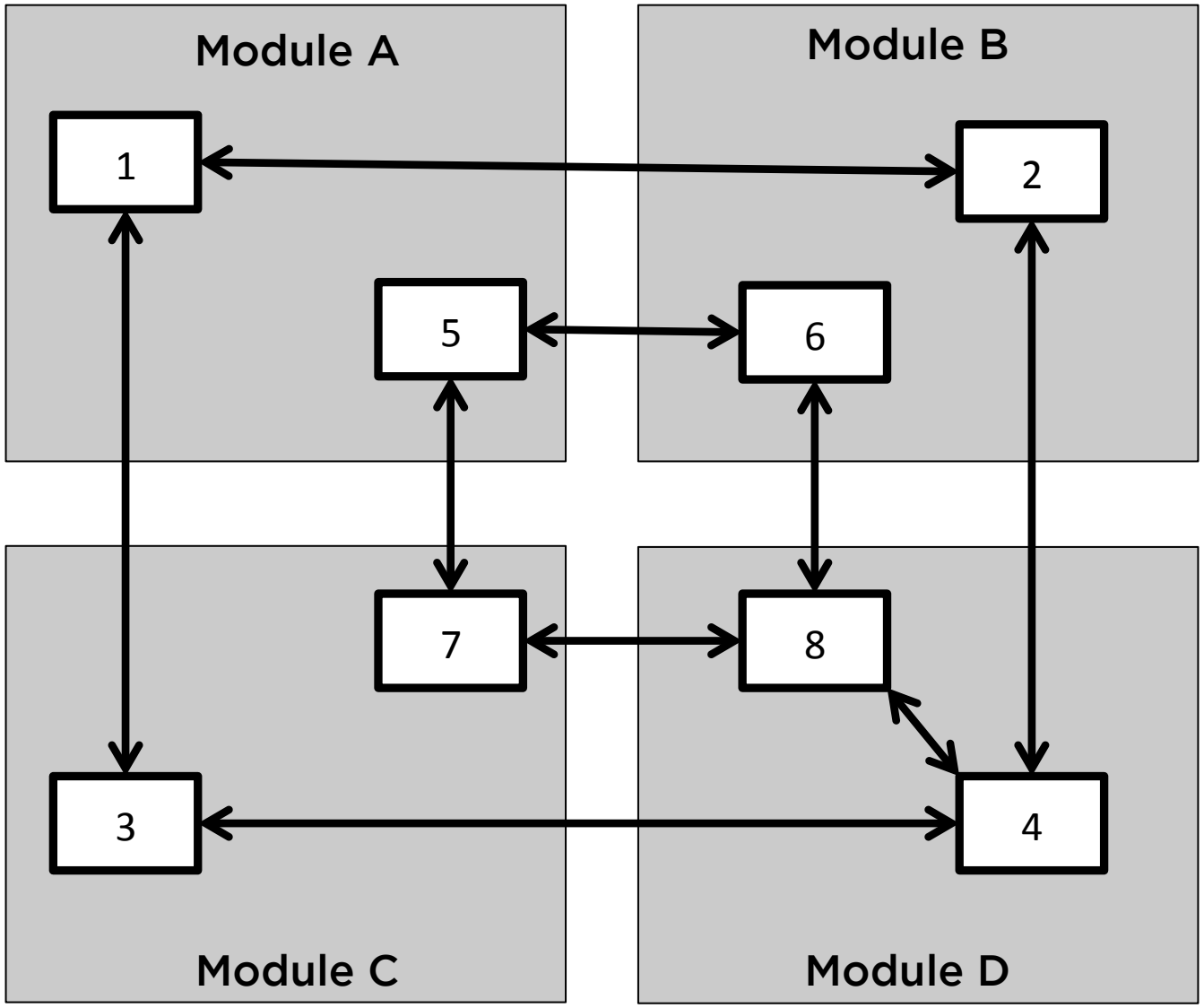
Cohesion measures the degree of connectivity between elements within a module.

- Low cohesion means that a module contains many unrelated elements. This violates the principle of modularity (single responsibility) and the principle of abstraction.

- Highly cohesive modules tend to be:
  - Robust
  - Reliable
  - Reusable
  - Understandable

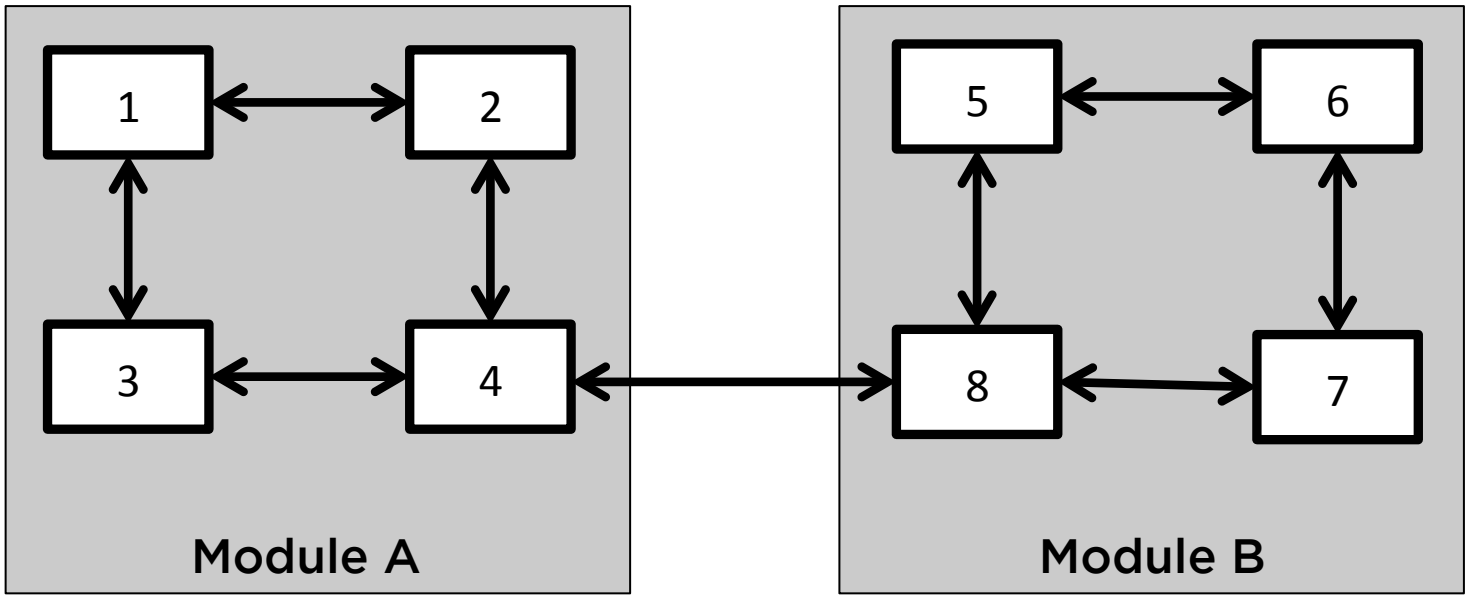
EXAMPLE

Low cohesion, high coupling



EXAMPLE

High cohesion, low coupling

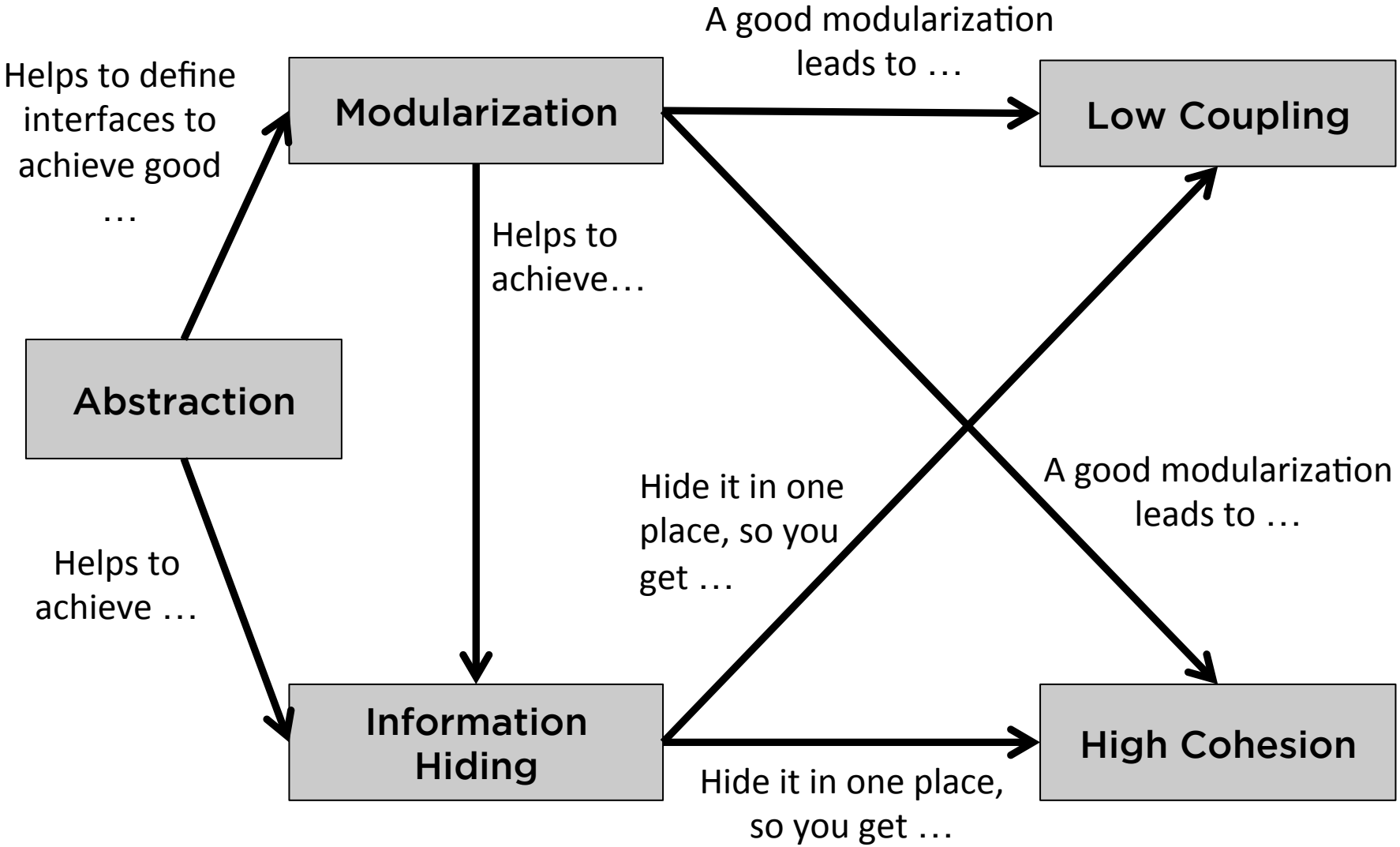


This refactored design is better, because changes to module A are mostly isolated to module A itself – only changes to element 4 might affect module B (through element 8). Changes to elements 1-3 and 5-7 do not affect the other module.

Note how boxes 5-8 were “rotated out” around box 4.

Relating principles

This overview is not necessarily complete, but it may help to understand how principles can reinforce each other.



Summary

- No foolproof formula for good design, but depend on general principles of sound design.
- Modularization:
  aim for clear separation of concerns.
- Abstraction:
  aim for appropriate level of abstraction.
- Information hiding:
  separate interface from implementation.
- Aim for High Cohesion and Low Coupling.

Thank you for your attention

Questions & suggestions can be sent to:
k.stol@ucc.ie