Software design

- Moving from analysis to design
- Design principles
  - Abstraction
  - Modularity
  - Coupling
  - Cohesion

CE202 Software Engineering, Autumn term

Dr Michael Gardner, School of Computer Science and Electronic Engineering, University of Essex

In This Lecture You Will Learn:

- The difference between analysis and design
- The difference between logical and physical design
- The difference between system and detailed design
- The characteristics of a good design
- The need to make trade-offs in design
- About some design principles

How is Design Different from Analysis?

- Design states 'how the system will be constructed without actually building it'

  (Rumbaugh, 1997)

- Analysis identifies 'what' the system must do
- Design specifies 'how' it will do it

How is Design Different from Analysis?

- The analyst seeks to understand the organization, its requirements and its objectives
- The designer seeks to specify a system that will fit the organization, provide its requirements effectively and assist it to meet its objectives

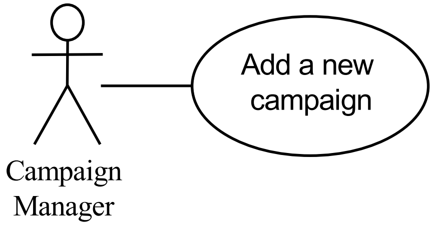
How is Design Different from Analysis?

- As an example, in a Campaign case study:
  - analysis identifies the fact that the **`Campaign`** class has a **`title`** attribute
  - design determines how this will be entered into the system, displayed on screen and stored in a database, together with all the other attributes of **`Campaign`** and other classes

When Does Analysis Stop and Design Start?

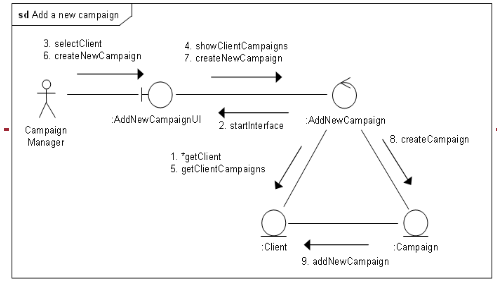
- In a waterfall life cycle there is a clear transition between the two activities
- In an iterative life cycle the analysis of a particular part of the system will precede its design, but analysis and design may be happening in parallel
- It is important to distinguish the two activities and the associated mindset
- We need to know ‘what’ before we decide ‘how’

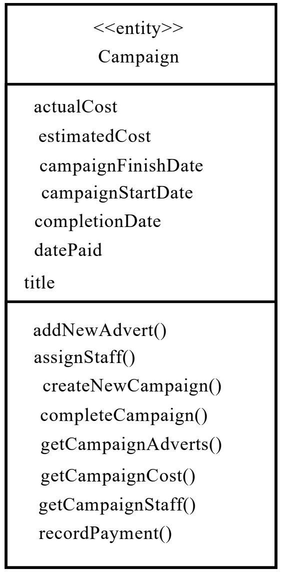
Requirements



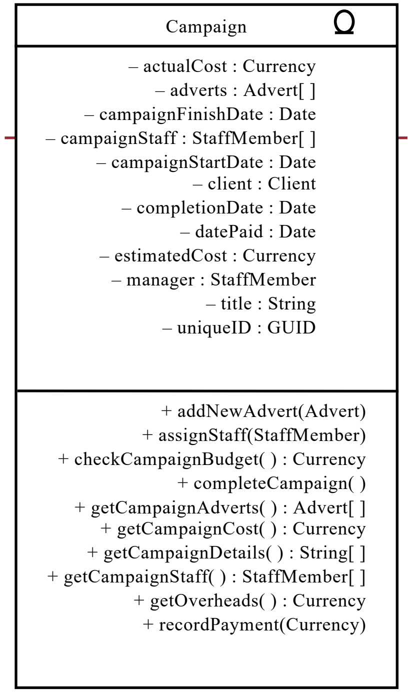
To record the details of each campaign for each client. This will include the title of the campaign, planned start and finish dates, estimated costs, budgets, actual costs and dates, and the current state of completion.

Analysis

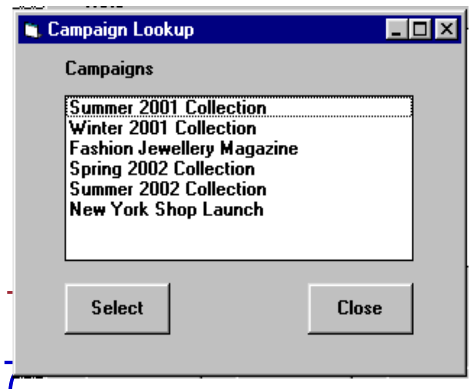




Design




```
CREATE TABLE Campaigns
(VARCHAR(30) uniqueID PRIMARY KEY NOT NULL,
FLOAT actualCost,
DATE campaignFinishDate,
DATE campaignStartDate,
VARCHAR(30) clientID NOT NULL,
DATE completionDate,
DATE datePaid,
FLOAT estimatedCost,
VARCHAR(30) managerID,
VARCHAR(50) title);
CREATE INDEX campaign_idx ON Campaigns (clientID, managerID, title);
```



Traditional Design

- Making a clear transition from analysis to design has advantages
  - project management—is there the right balance of activities?
  - staff skills—analysis and design may be carried out by different staff
  - client decisions—the client may want a specification of the ‘what’ before approving spending on design
  - choice of development environment—may be delayed until the analysis is complete

Design in the Iterative Life Cycle

- Advantages of the iterative life cycle include
  - risk mitigation—making it possible to identify risks earlier and to take action
  - change management—changes to requirements are expected and properly managed
  - team learning—all the team can be involved from the start of the project
  - improved quality—testing begins early and is not done as a ‘big bang’ with no time

Seamlessness

- The same model—the **class model**—is used through the life of the project
- During design, additional detail is added to the analysis classes, and extra classes are added to provide the supporting functionality for the user interface and data management
- Other diagrams are also elaborated in design activities

Logical and Physical Design

- In structured analysis and design a distinction has been made between **logical** and **physical** design
- **Logical design** is independent of the implementation language and platform
- **Physical design** is based on the actual implementation platform and the language that will be used

Logical and Physical Design Example

- Some design of the user interface classes can be done without knowing whether it is to be implemented in Java, C++ or some other language- types of fields, position in windows
- Some design can only be done when the language has been decided upon — the actual classes for the types of fields, the layout managers available to handle window layout

Logical and Physical Design

- It is not necessary to separate these into two separate activities
- It may be useful if the software is to be implemented on different platforms
- Then it will be an advantage to have a platform-independent design that can be tailored to each platform

System Design and Detailed Design

- **System design** deals with the high level architecture of the system (see lecture next week)
  - structure of sub-systems
  - distribution of sub-systems on processors
  - communication between sub-systems
  - standards for screens, reports, help etc.
  - job design for the people who will use the system

System Design and Detailed Design

- Traditional **detailed design** consists of four main activities
  - designing inputs
  - designing outputs
  - designing processes
  - designing files and database structures

System Design and Detailed Design

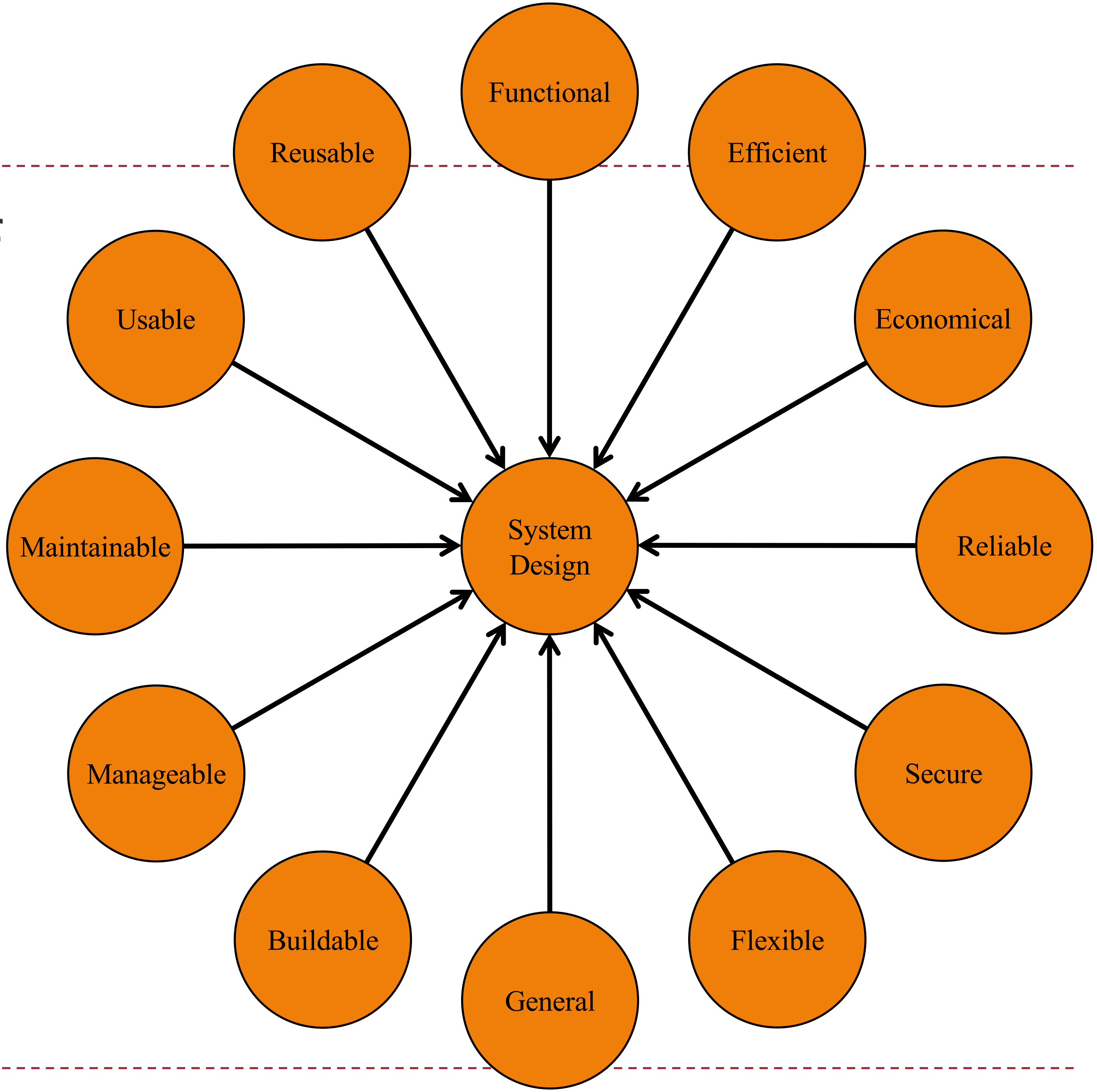
- Traditional detailed design tried to **maximise cohesion**
  - elements of a module of code all contribute to the achievement of a single function
- Traditional detailed design tried to **minimise coupling**
  - unnecessary linkages between modules that made them difficult to maintain or use in isolation from other modules

- Discussed later in the lecture

System Design and Detailed Design

- Object-oriented detailed design adds detail to the analysis model
  - types of attributes
  - operation signatures
  - assigning responsibilities as operations
  - additional classes to handle user interface
  - additional classes to handle data management
  - design of reusable components
  - assigning classes to packages

Qualities of Design



Qualities of Design (1 of 3)

- Functional—system will perform the functions that it is required to
- Efficient—the system performs those functions efficiently in terms of time and resources
- Economical—running costs of system will not be unnecessarily high
- Reliable—not prone to hardware or software failure, will deliver the functionality when the users want it

Qualities of Design (2 of 3)

- Secure—protected against errors, attacks and loss of valuable data
- Flexible—capable of being adapted to new uses, to run in different countries or to be moved to a different platform
- General—general-purpose and portable (mainly applies to utility programs)
- Buildable—Design is not too complex for the developers to be able to implement it

Qualities of Design (3 of 3)

- Manageable—easy to estimate work involved and to check of progress
- Maintainable—design makes it possible for the maintenance programmer to understand the designer's intention
- Usable—provides users with a satisfying experience (not a source of dissatisfaction)
- Reusable—elements of the system can be reused in other systems

Prioritizing Design Trade-offs

- Designer is often faced with design objectives that are mutually incompatible.
- It is helpful if guidelines are prepared for prioritizing design objectives.
- If design choice is unclear users should be consulted.

Trade-offs in Design

- Design to meet all these qualities may produce conflicts
- Trade-offs have to be applied to resolve these
- Functionality, reliability and security are likely to conflict with economy
- Level of reliability, for example, is constrained by the budget available for the development of the system

Measurable Objectives in Design

- In the requirements phase a set of **non-functional requirements** are described
- How can we tell whether these have been achieved?
- Measurable objectives set clear targets for designers
- Objectives should be quantified so that they can be tested

Measurable Objectives in Design

- To reduce invoice errors by one-third within a year
- How would you design for this?

Measurable Objectives in Design

- To reduce invoice errors by one-third within a year
- How would you design for this?
  - sense checks on quantities
  - comparing invoices with previous ones for the same customer
  - better feedback to the user about the items ordered

Measurable Objectives in Design

- To process 50% more orders at peak periods
- How would you design for this?

Measurable Objectives in Design

- To process 50% more orders at peak periods
- How would you design for this?
  - design for as many fields as possible to be filled with defaults
  - design for rapid response from database
  - design system to handle larger number of simultaneous users

Some general design principles

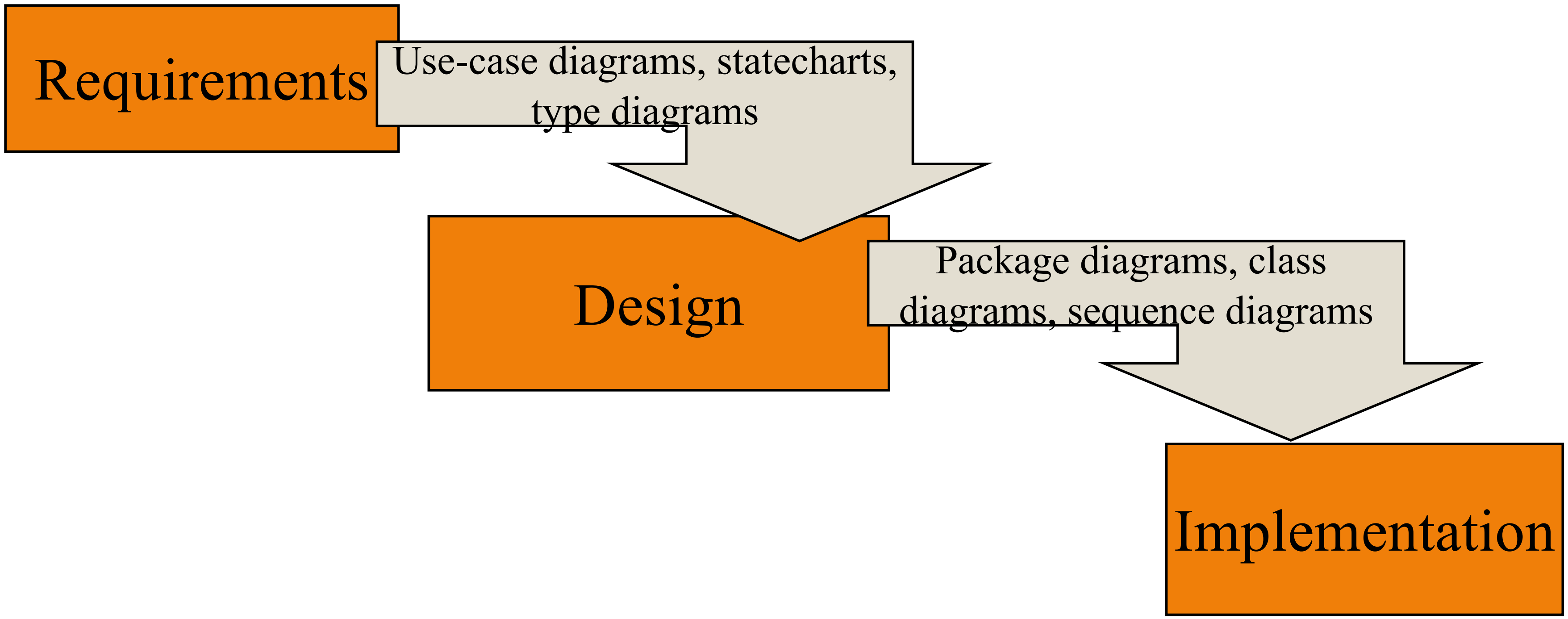
- Abstraction
- Modularity
- Coupling
- Cohesion

Abstraction

- Process:
  - *“Abstraction is a process whereby we identify the important aspects of a phenomenon and ignore its details.”* [Ghezzi et. al 1991]
- Product:
  - *“[A] simplified description, … that emphasizes some of the system's … properties while suppressing others.*
  - *A good abstraction is one that emphasizes details that are significant to the reader or user and suppress details that are, at least for the moment, immaterial or diversionary."* [Shaw 1984]

Software design's level of abstraction

- A level of abstraction that is between requirements and implementation



Modularity

Kinds of modules

Cohesion

Coupling

Modularity

- The quality of being divided into modules
- Quality attributes:
  - Well-defined: Modules are clearly distinguished
  - Separation of concerns: each module has one concern, with minimal overlap between modules
  - Loosely coupled
  - Cohesive

Why should design be modular?

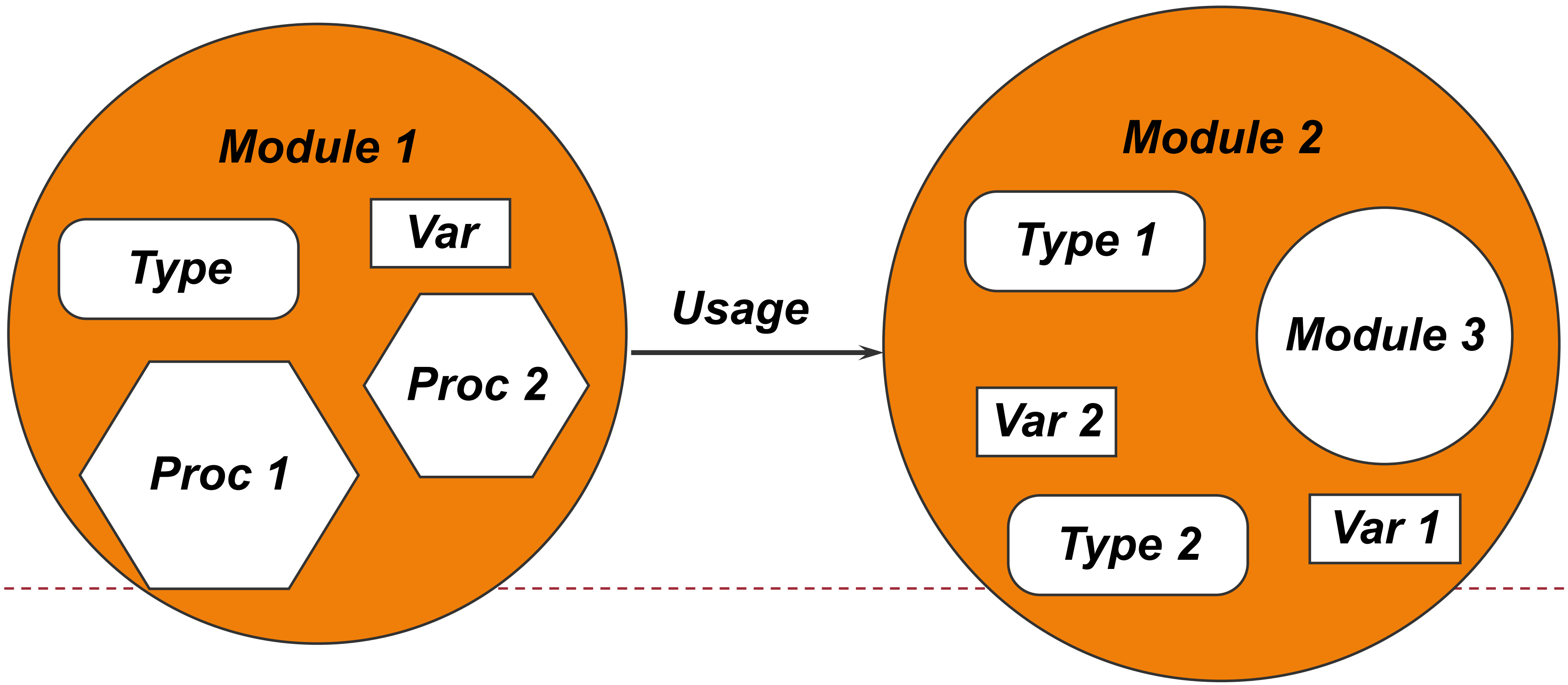
- Comprehensibility
- Parallel development
- Quality: easier to—
  - Test (see Unit testing lecture)
  - Verify and validate
  - Measure reliability
- Maintainability: easier to—
  - Locate faults and correct them
  - Change
  - Enhance
- Reusability

Module: Definition 1 (general)

- Any kind of an “independent” software unit
  - Routines, subroutines (in Java: “methods”; in C: “functions”)
  - Various physical units
    - Files (as in C, C++)
    - Library files (.a is the UNIX convention, DLL in Windows, JAR in Java)

Module: Definition 2 (object-based programming)

- Packaging of "related" variables and procedures
  - Packages (as in ADA)
  - Modules (as in Modula)
- Programming paradigm: "modular programming" or "Object-based programming"

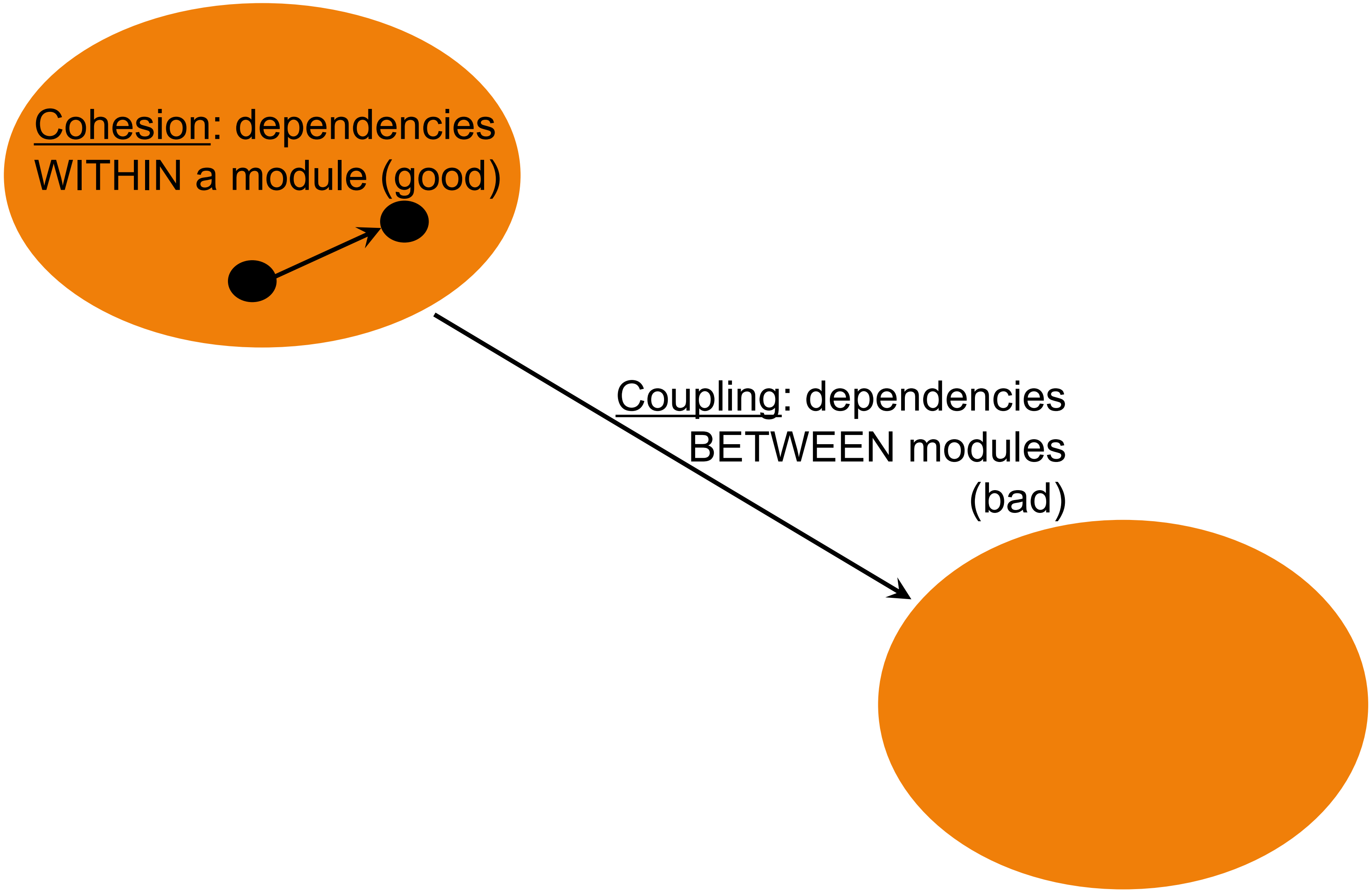


Module: Definition 3 (OOP)

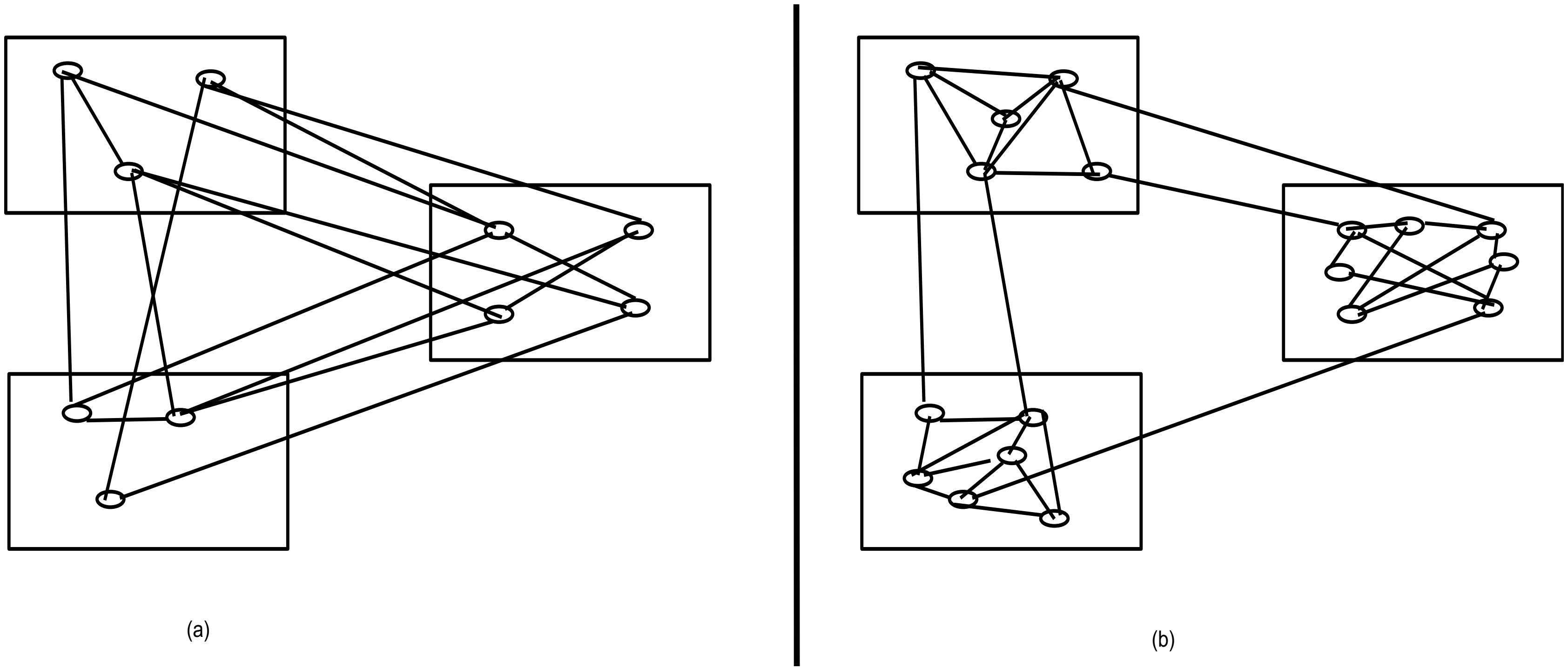
- "The act of grouping into a single object both data and operations that [directly] affect that data is known as encapsulation. …" [Wirfs-Brock 90, p.6]
  - A 'class'
  - An 'object'
- Also known as: encapsulation, separation of concerns
- Programming paradigm: "object-oriented programming" or "class-based programming"

Coupling and cohesion

Cohesion vs. coupling



A visual representation



high coupling

low coupling

Coupling

- Coupling is measured by the answer to this question:

  How much of one module must be known in order to understand another module?
  - The more that we must know of module B in order to understand module A, the more they are coupled
- Objective: **loosely coupled systems**

Kinds of Direct Dependencies

[Briand et. al 99]

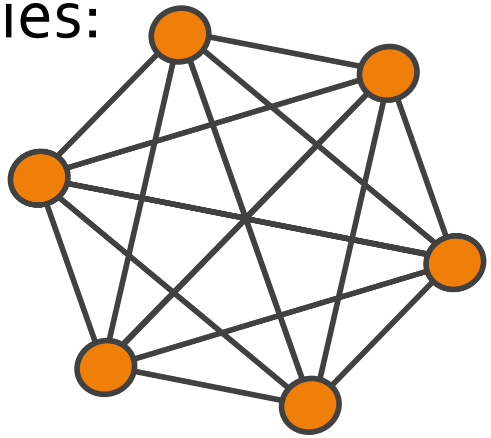
- From a class to its --
  - Super class
    - Emp → Person
  - Field (class)
    - Emp → Date
- From a method to its --
  - Parameter types
    - Emp.hire → Reason
  - Return type
    - Emp.hire → SuccessCode
  - (Class of) local variables
    - Emp.hire → DB
  - Invoked method
    - Emp.hire → DB.getDB

```
class Employee extends Person {
  Date birthday;
  SuccessCode hire(Reason why) {
    why.print();
    DB theDB = DB.getDB(); ...
    return SuccessCode.sucessful; }
};
```

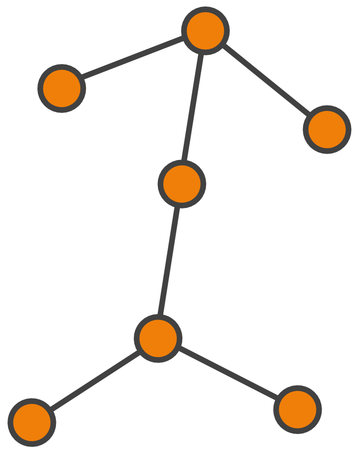
Briand et al, 1999, Empirical studies of object-oriented artifacts, methods and processes: state of the art and future direction. Int. J of Empirical Software Engineering 4(4).

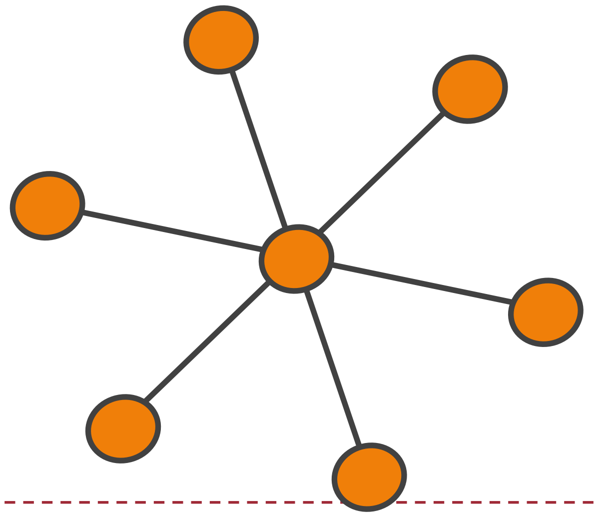
Dependency graphs

- Maximal number of dependencies: $n(n-1)/2$
  - Complete Graph



- Minimal number of interfaces $n-1$
  - Tree
    - Star





How to achieve Low Coupling

- Low coupling can be achieved if a calling class does not need to know anything about the internals of the called class (Principle of information hiding, Parnas)
- Questions to ask:
  - Does the calling class really have to know any attributes of classes in the lower layers?
  - Is it possible that the calling class calls only operations of the lower level classes?

David Parnas, *1941,
Developed the concept of
modularity in design.

Cohesion

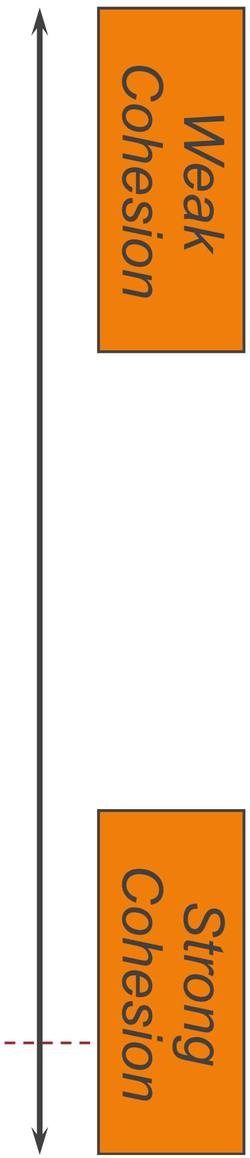
- Coupling between elements within a module

  co·here v. co·hered, co·her·ing, co·heres. --intr. 1. To stick or hold together in a mass that resists separation. 2. To have internal elements or parts logically connected so that aesthetic consistency results. [American Heritage Dictionary]
- Module Cohesion is "how tightly bound or related are its internal elements to one another."
- A desirable property!
  - E.g., cohesion within the methods of a class, within classes in a module

Levels of cohesion

[Myers '78]

- **Coincidental** - Occurs when carelessly trying to satisfy style rules.
  - print_prompt_and_check_parameters
- **Logical** - Related logic, but no corresponding relation in control or data.
  - Library of trigonometric functions, in which there is no relation between the implementation of the functions.
- **Temporal** - Series of actions related in time.
  - Initialization module.
  - Communication- Series of actions related to a step of the processing of a single function
- **Data item**. May occur in the attempt to avoid control coupling.
  - clear_window_and_draw_its_frame
- **Functional** - Execute a single, well-defined function or duty.
- **Data** - Collection of related operations on the same data.



How to achieve high Cohesion

- High cohesion can be achieved if most of the interaction is within subsystems, rather than across subsystem boundaries
- Questions to ask:
  - Does one subsystem always call another one for a specific service?
    - Yes: Consider moving them together into the same subystem.
  - Which of the subsystems call each other for services?
    - Can this be avoided by restructuring the subsystems or changing the subsystem interface?
  - Can the subsystems even be hierarchically ordered (in layers)?

Summary

- The difference between analysis and design
- The difference between logical and physical design
- The difference between system and detailed design
- The characteristics of a good design
- The need to make trade-offs in design
- About some design principles
  - Abstraction
  - Modularity
  - Coupling
  - Cohesion

Further reading

- Bennett – Chapter 14, Detailed Design
- Bruegge – 6.3.3 Coupling and Cohesion
- Ghezzi 1991 – Fundamentals of Software Engineering, Prentice-Hall
- Shaw 1984 – Abstraction techniques in modern programming languages, IEEE Software, Vol 1(4)
- Wirfs-Brock, 1990, Designing object-oriented software, Prentice-Hall
- Briand et al, 1999, Empirical studies of object-oriented artifacts, methods and processes: state of the art and future direction. Int. J of Empirical Software Engineering 4(4).
- Myers 1978. Composite/Structured Design. Van Nostrand Reinhold, New York.
- Hoffman, Daniel M.; Weiss David M. (Eds.): Software Fundamentals – Collected Papers by David L. Parnas, 2001, Addison-Wesley

Exercise: coupling and cohesion (1)

- Coupling is the unnecessary dependency of one component upon another component's implementation. An example would be if you had a Dog class that should be able to bark and jump. You could write it like this:

```
class Dog
{
void doAction(int number) {
        if(number==1)
                //Jump code goes here
        if(number==2)
                //Bark code goes here
}
}
```

- What is wrong with this?

Exercise: coupling and cohesion (2)

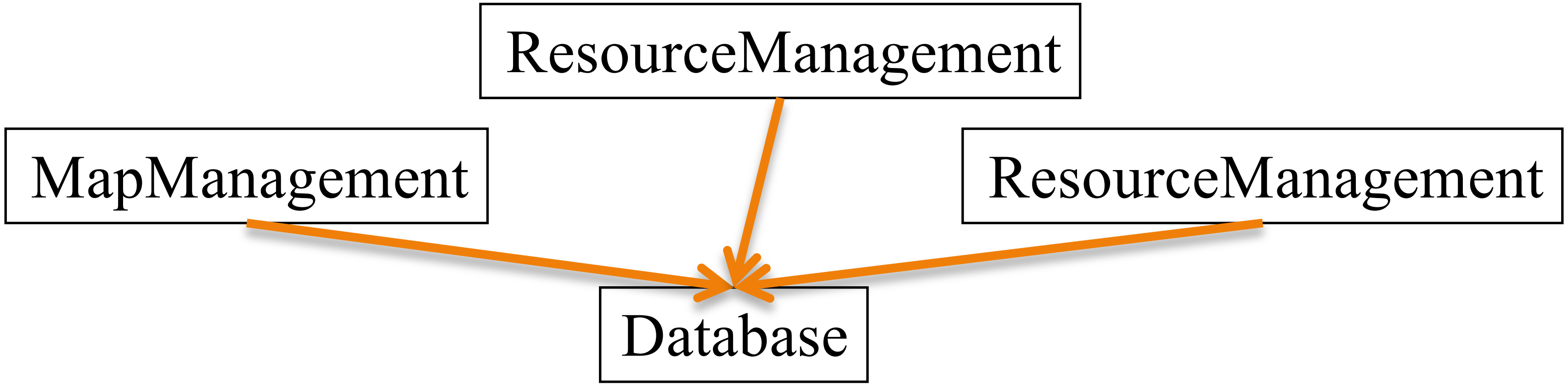
- What is wrong with this implementation of the Car class?

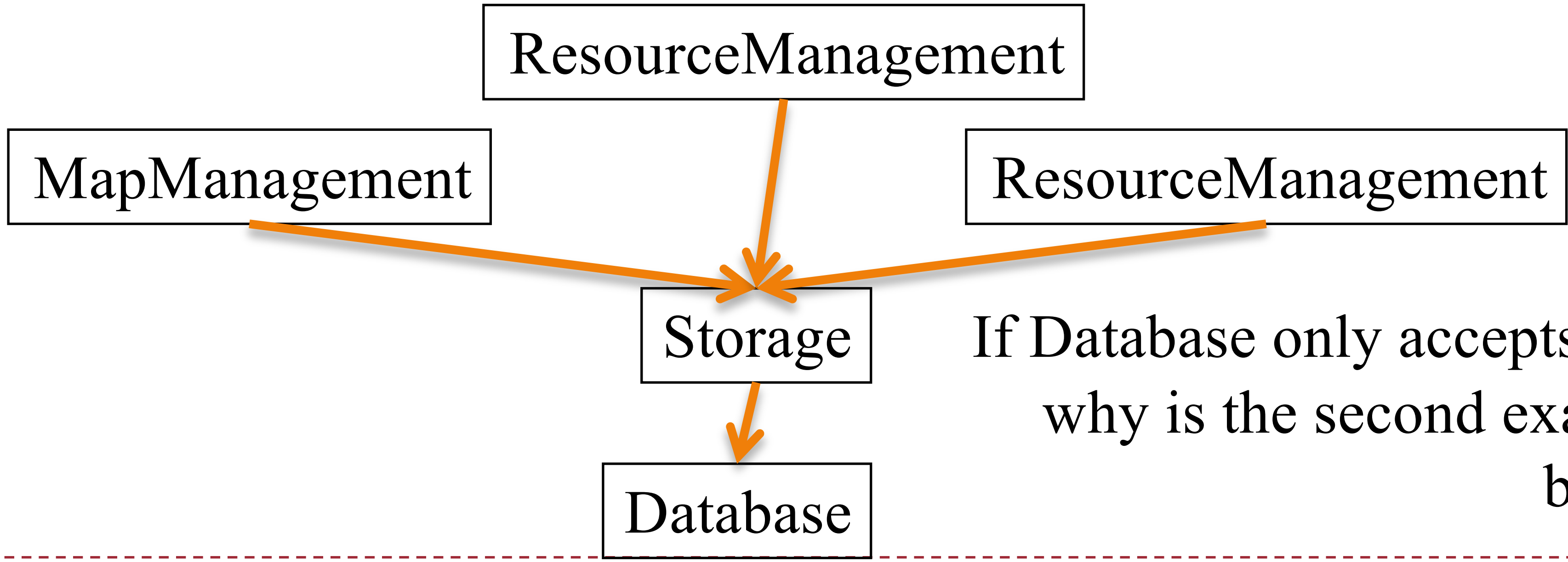
```
Driver campbell = new Driver();
Car ford = new Car("Ford", "red");
. . .
public class Driver {
    Car myCar;
    . . .
    }

public void goFaster(int speed) {
    myCar.speed += speed;
. . .
}
```

How can you avoid tight coupling?

Exercise: Reducing coupling by adding complexity





If Database only accepts SQL why is the second example better?