
Chapter 6 – Architectural design

Topics covered



- ✧ Architectural design decisions
- ✧ Architectural views
- ✧ Architectural patterns



Software architecture



- ✧ The design process for identifying the sub-systems making up a system and the framework for sub-system control and communication is **architectural design**.
- ✧ The output of this design process is a description of the **software architecture**.

What is Design?



- ✧ The creative process of transforming a problem into a solution
 - Transforming a requirements specification into a detailed description of the software

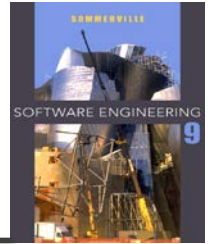
Stages of Design



- ✧ Problem understanding
- ✧ Identify one or more solutions
- ✧ Describe solution abstractions
- ✧ Repeat process for each identified abstraction until the design is expressed in primitive terms



Architectural design process



- ✧ System structuring
 - Decompose the system into several principal sub-systems
 - Identify communications between these sub-systems
- ✧ Control modelling
 - Establish a model of the control relationship between the different part of the system
- ✧ Modular decomposition
 - Decompose sub-systems into modules
- ✧ Sub-system vs. module ?

Advantages of explicit architecture



✧ Stakeholder communication

- Architecture may be used as a focus of discussion by system stakeholders.

✧ System analysis

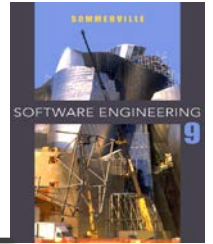
- Means that analysis of whether the system can meet its non-functional requirements is possible.

✧ Large-scale reuse

- The architecture may be reusable across a range of systems.



Architecture and system characteristics



✧ Performance

- Localize critical operations and minimize communications. Use large rather than fine-grain components.

✧ Security

- Use a layered architecture with critical assets in the inner layers.

✧ Safety

- Localize safety-critical features in a small number of sub-systems.

✧ Availability

- Include redundant components and mechanisms for fault tolerance.

✧ Maintainability

- Use fine-grain, replaceable components.

Architectural conflicts



- ✧ Using large-grain components improves performance but reduces maintainability.
- ✧ Introducing redundant data improves availability but makes security more difficult.
- ✧ Localizing safety-related features usually means more communication so degraded performance.

Architectural views (4+1 views)



✧ A logical view

- which shows the key abstractions in the system as objects or object classes.

✧ A process view

- which shows how, at run-time, the system is composed of interacting processes.

✧ A development view

- which shows how the software is decomposed for development.

✧ A physical view

- which shows the system hardware and how software components are distributed across the processors in the system.

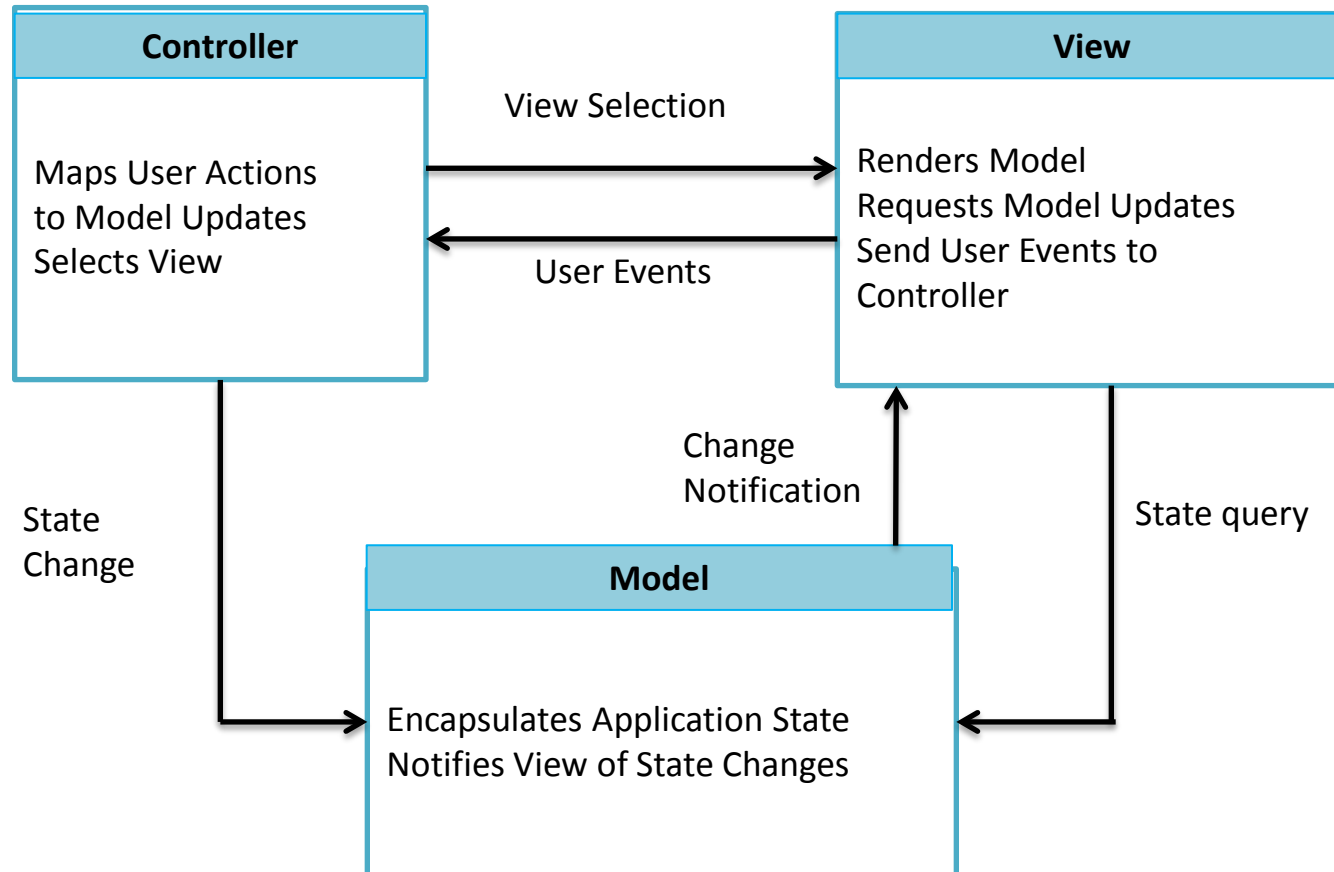
✧ Related using use cases or scenarios (+1)

Architectural patterns

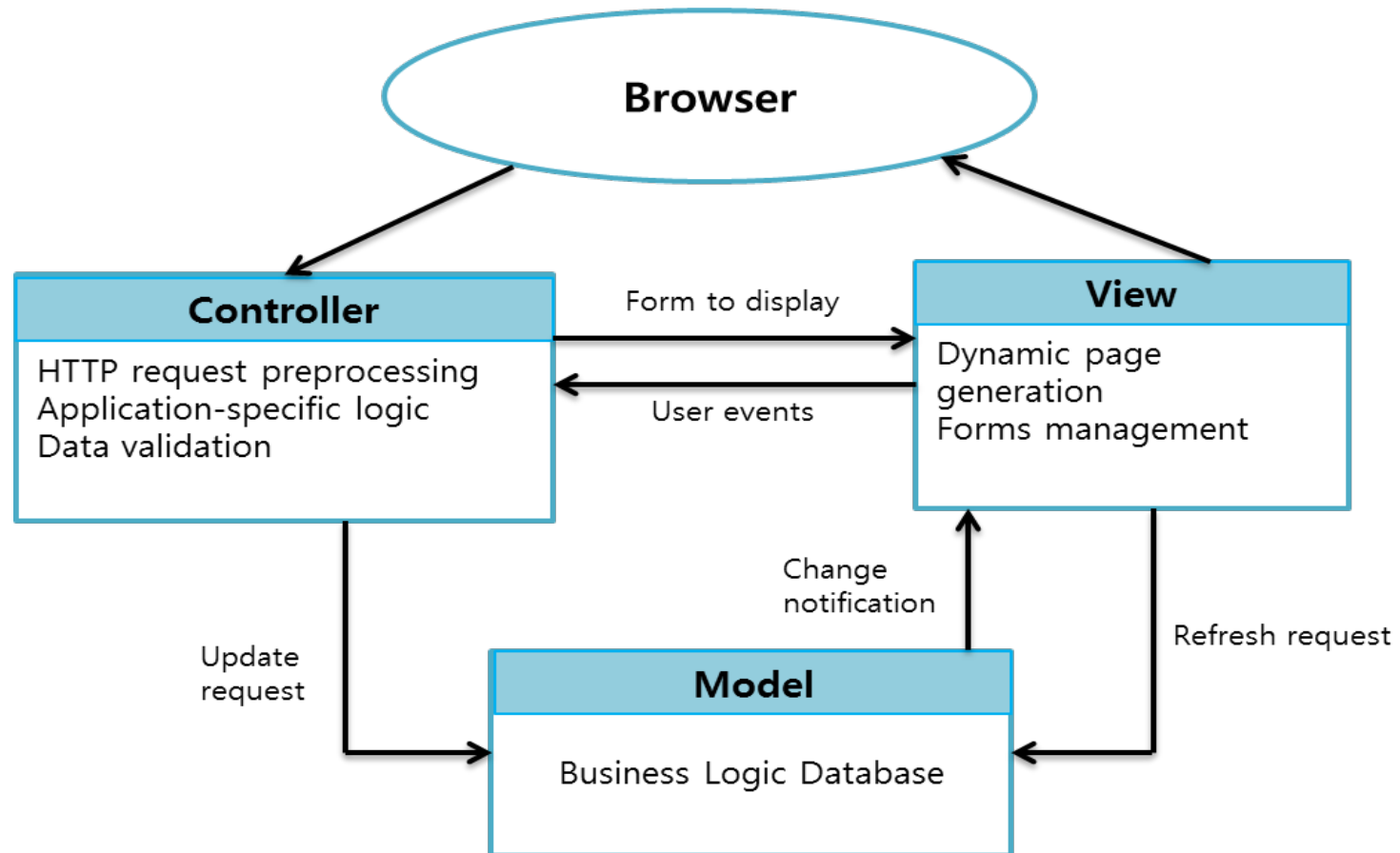


- ✧ Patterns are a means of representing, sharing and reusing knowledge.
- ✧ An architectural pattern is a stylized description of good design practice, which has been tried and tested in different environments.
- ✧ Patterns should include information about when they are and when they are not useful.
- ✧ Patterns may be represented using tabular and graphical descriptions.

The Model-View-Controller (MVC) pattern



MVC example: web application

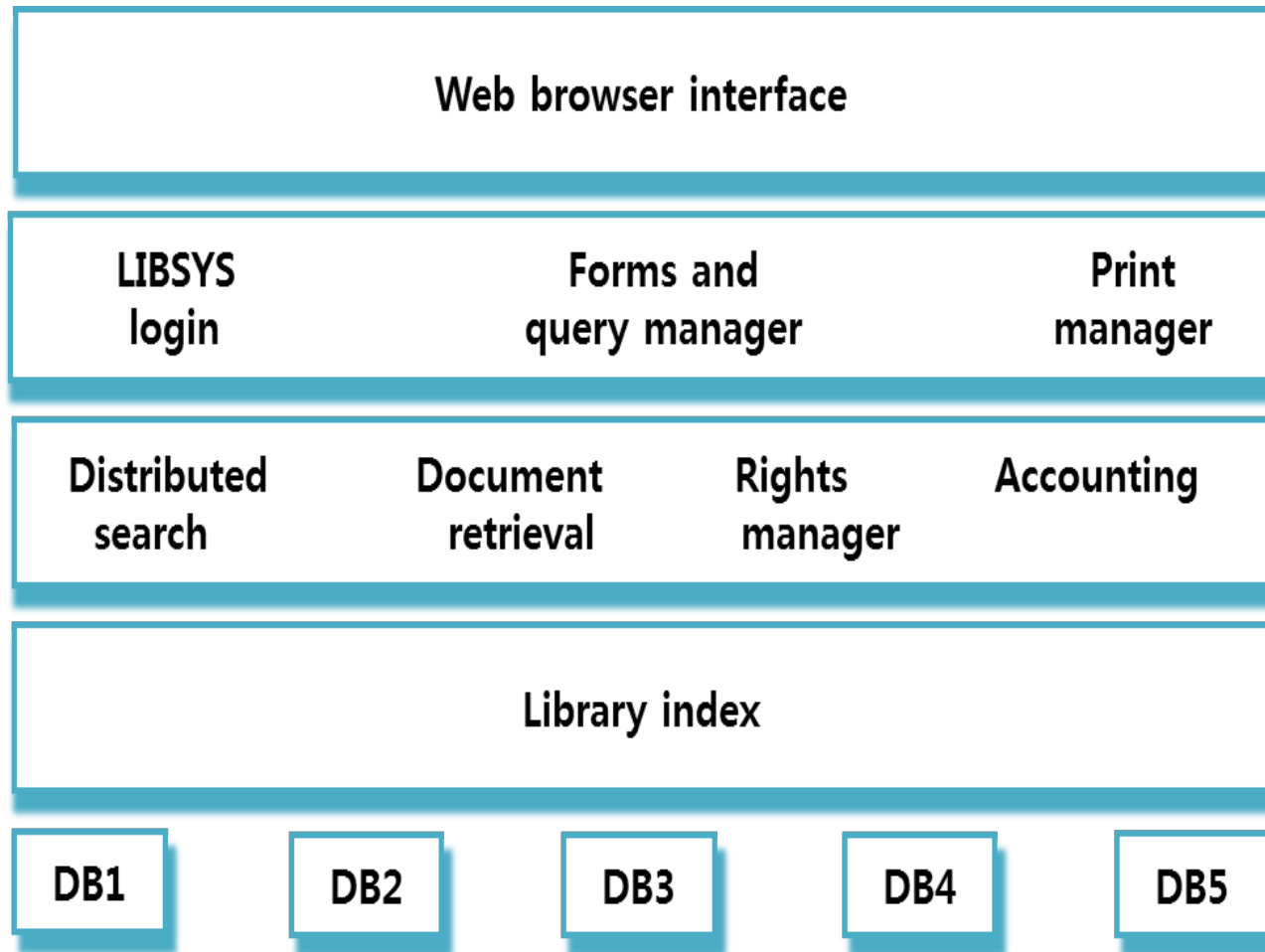


Abstract machine (layered) model



- ✧ Used to model the interfacing of sub-systems.
- ✧ Organises the system into a set of layers (or abstract machines) each of which provide a set of services.
- ✧ Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.
- ✧ However, often artificial to structure systems in this way.

Layered: LIBSYS system

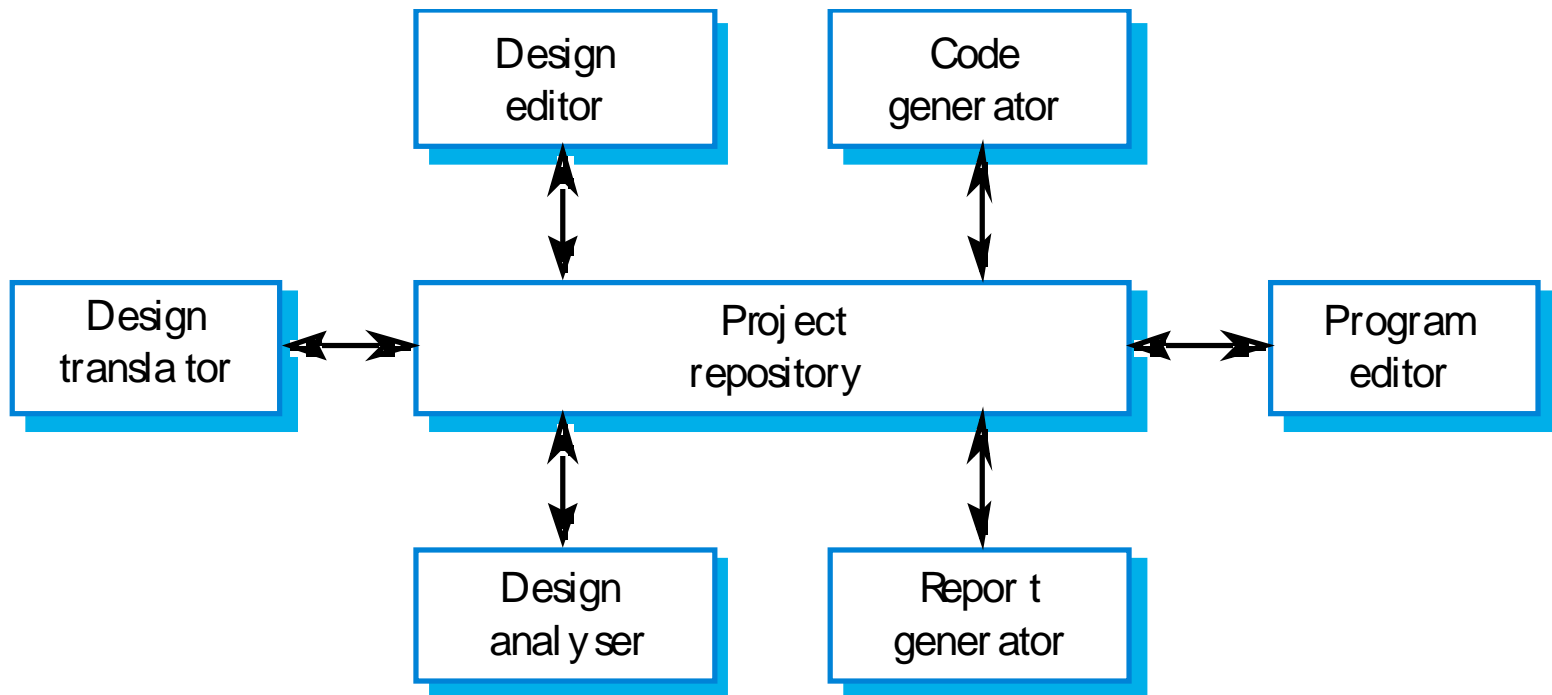
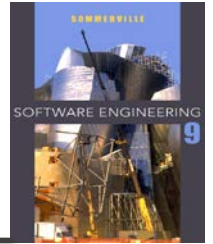


The repository model



- ✧ Sub-systems must exchange data. This may be done in two ways:
 - Shared data is held in a central database or repository and may be accessed by all sub-systems;
 - Each sub-system maintains its own database and passes data explicitly to other sub-systems.
- ✧ When large amounts of data are to be shared, the repository model of sharing is most commonly used
 - Examples: Command and Control systems, Management Information Systems, CAD systems, CASE tools

CASE toolset architecture





Repository model characteristics

✧ Advantages

- Efficient way to share large amounts of data;
- Sub-systems need not be concerned with how data is produced Centralised management e.g. backup, security, etc.
- Sharing model is published as the repository schema.

✧ Disadvantages

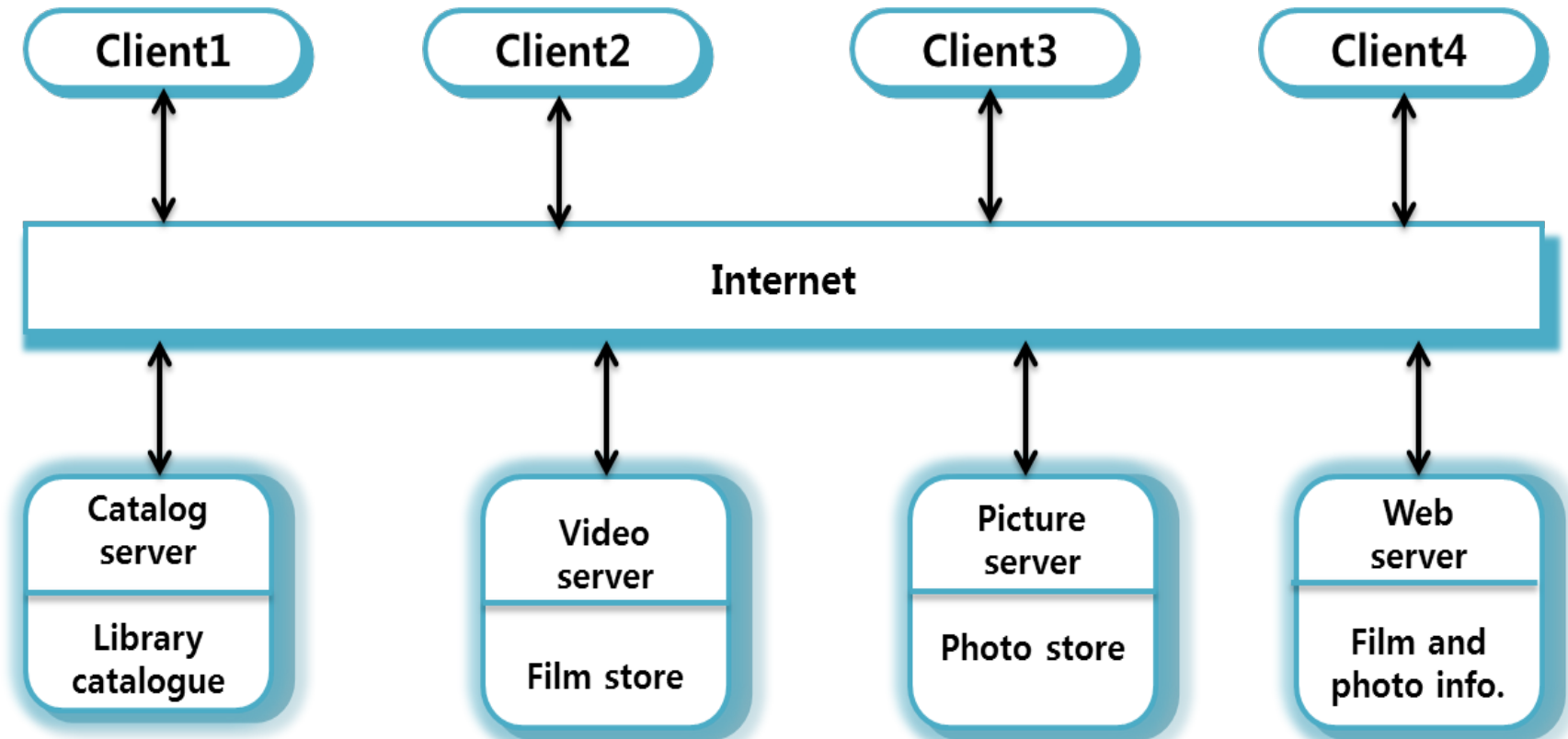
- Sub-systems must agree on a repository data model. Inevitably a compromise;
- Data evolution is difficult and expensive;
- No scope for specific management policies;
- Difficult to distribute efficiently.

Client-server model



- ✧ The application is modelled as a set of services that are provided by servers and a set of clients that use these services.
- ✧ Clients know of servers but servers need not know of clients.
- ✧ Clients and servers are logical processes
- ✧ The mapping of processors to processes is not necessarily 1 : 1.

Client-server example: film library



Client-server characteristics



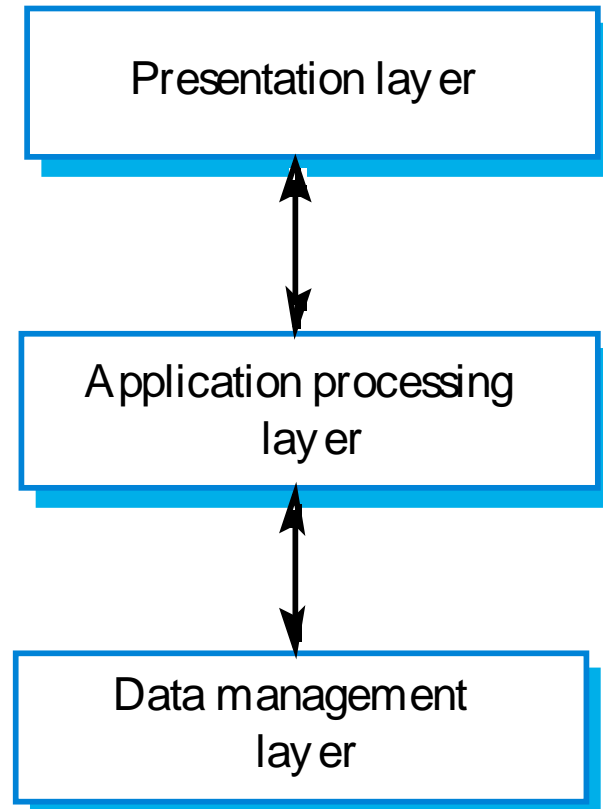
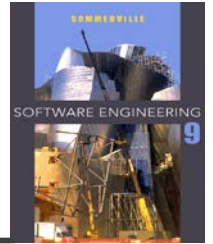
✧ Advantages

- Distribution of data is straightforward;
- Makes effective use of networked systems. May require cheaper hardware;
- Easy to add new servers or upgrade existing servers.

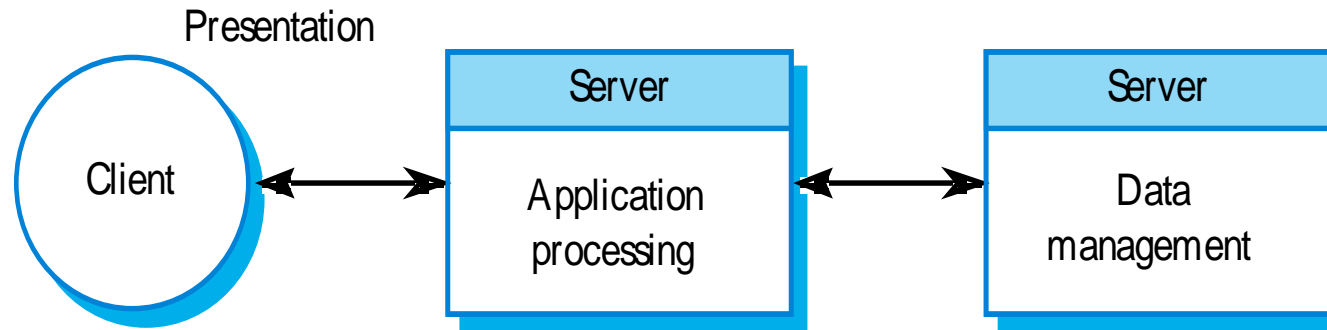
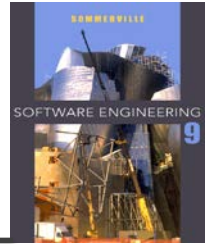
✧ Disadvantages

- No shared data model so sub-systems use different data organisation. Data interchange may be inefficient;
- Redundant management in each server;
- No central register of names and services - it may be hard to find out what servers and services are available.

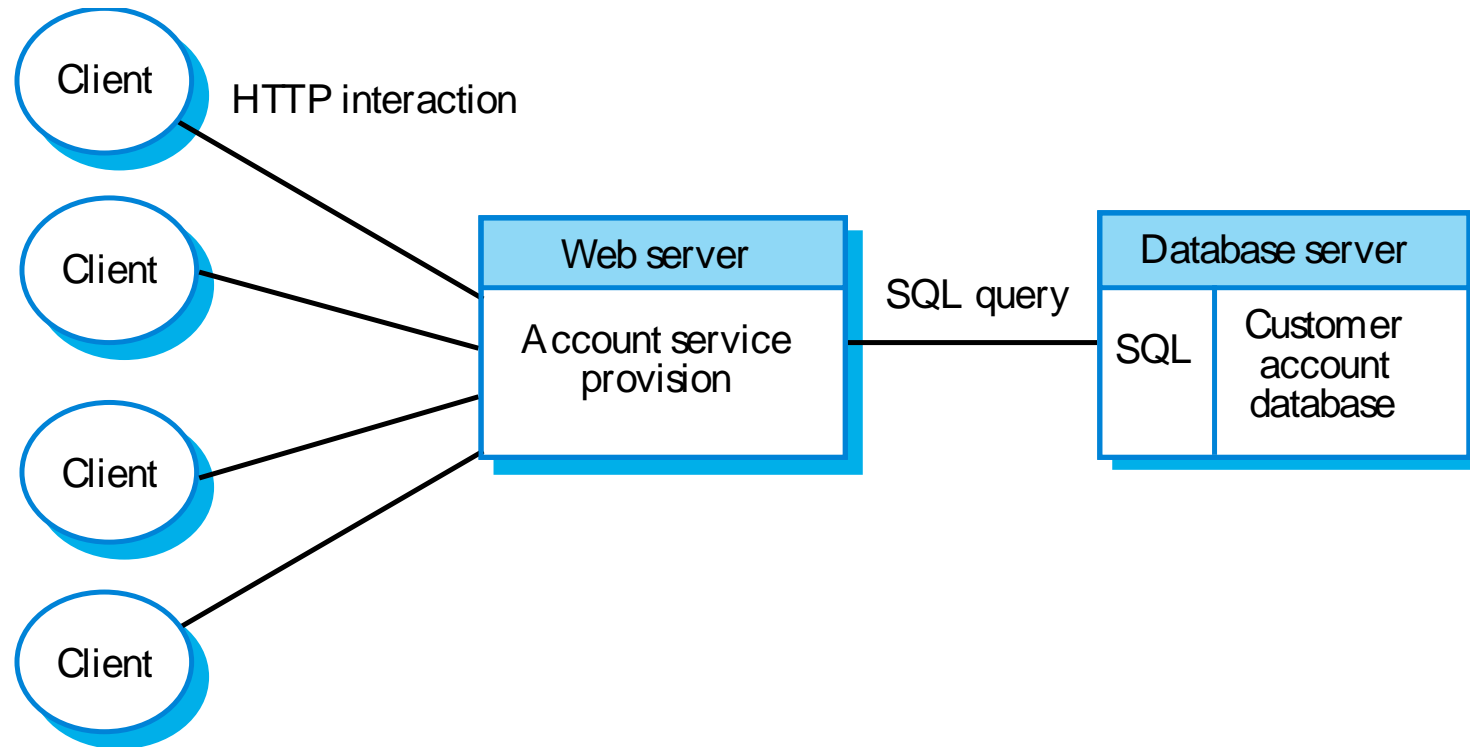
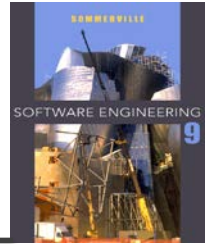
Using application layers



A 3-tier C/S architecture



Example: An internet banking system



Chapter 7 – Design and Implementation

Design and implementation



- ✧ Software design and implementation is the stage in the software engineering process at which an executable software system is developed.
- ✧ Software design and implementation activities are invariably inter-leaved.
 - Software design is a creative activity in which you identify software components and their relationships, based on a customer's requirements.
 - Implementation is the process of realizing the design as a program.

Fundamental design strategies



✧ Data flow structuring

- The software is initially treated as a single component part of the overall system
- The software is formed as a set of communicating software machines, each having clearly defined functions and responsibilities

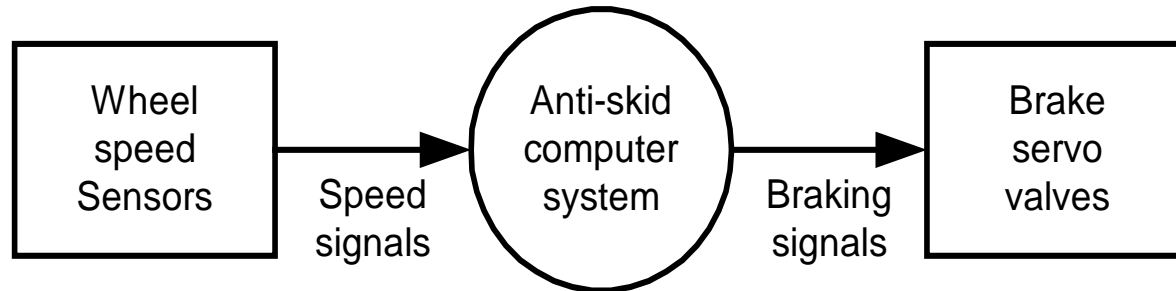
✧ Object structuring

- describes the system as a collection of objects

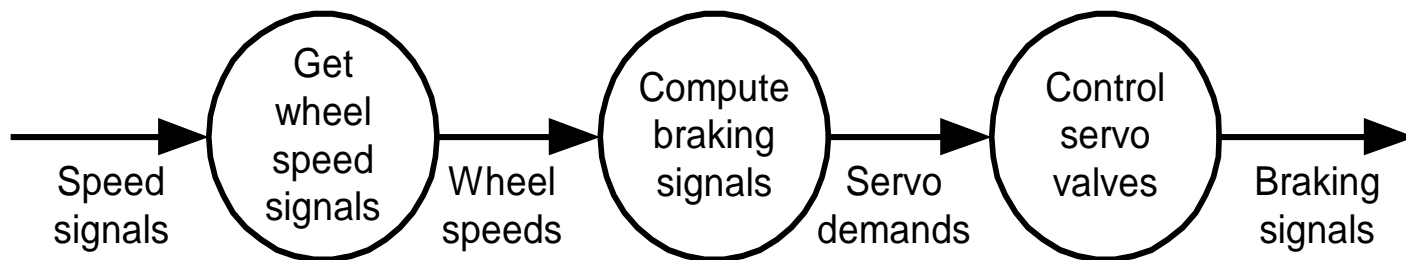
✧ Mugwump school approach

- is practiced by people who believe design is needless

An Example of a data-flow approach

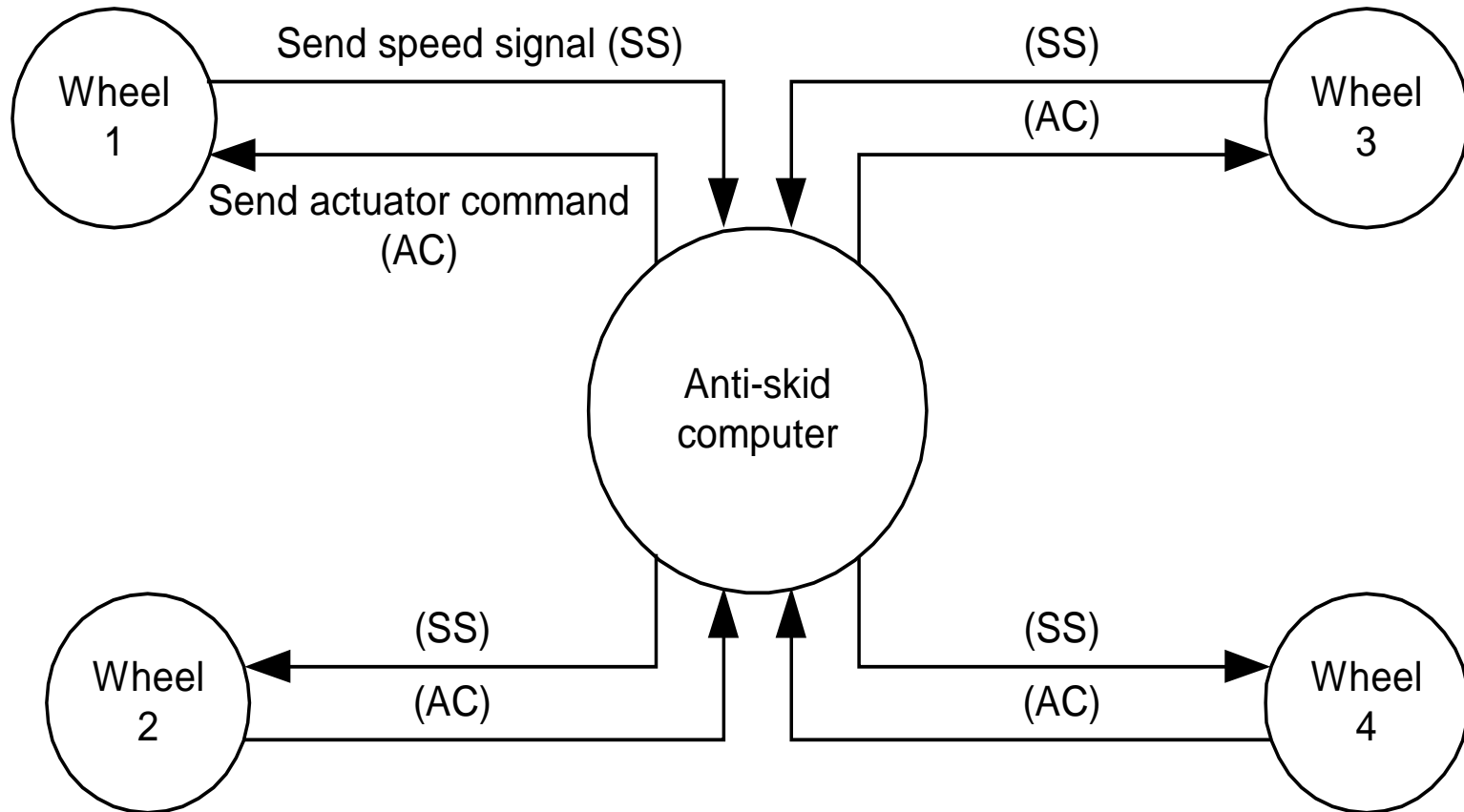
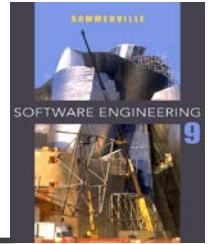


(a) System-oriented view

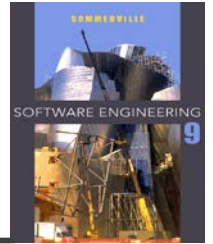


(b) Higher-level software structure

An Example of an object-based approach



The elements of modular design



✧ Modules and modularization

- a basic feature of good design is the partitioning of systems into smaller chunks
- to reduce the total problem into one of manageable proportions

✧ Coupling – a measure of module independence

- the amount of interaction between modules has a significant effect on software quality

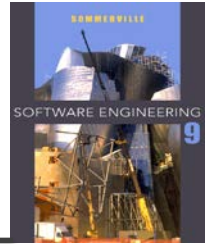
✧ Cohesion – a measure of module binding

- the parameter which defines how strongly elements are inter-related

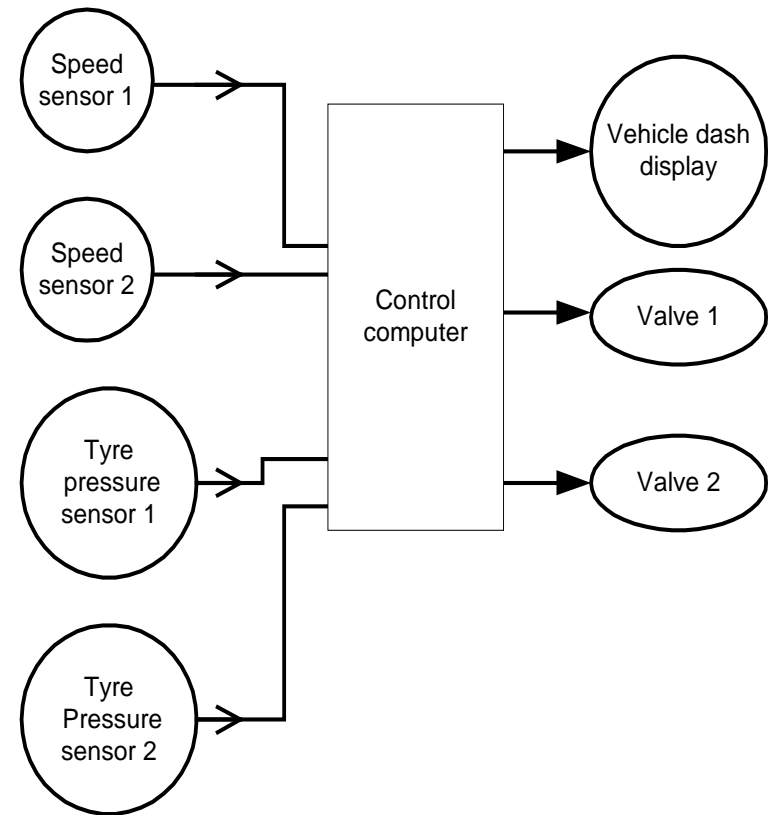
✧ Size and complexity

- don't make modules too large

Introducing modularization

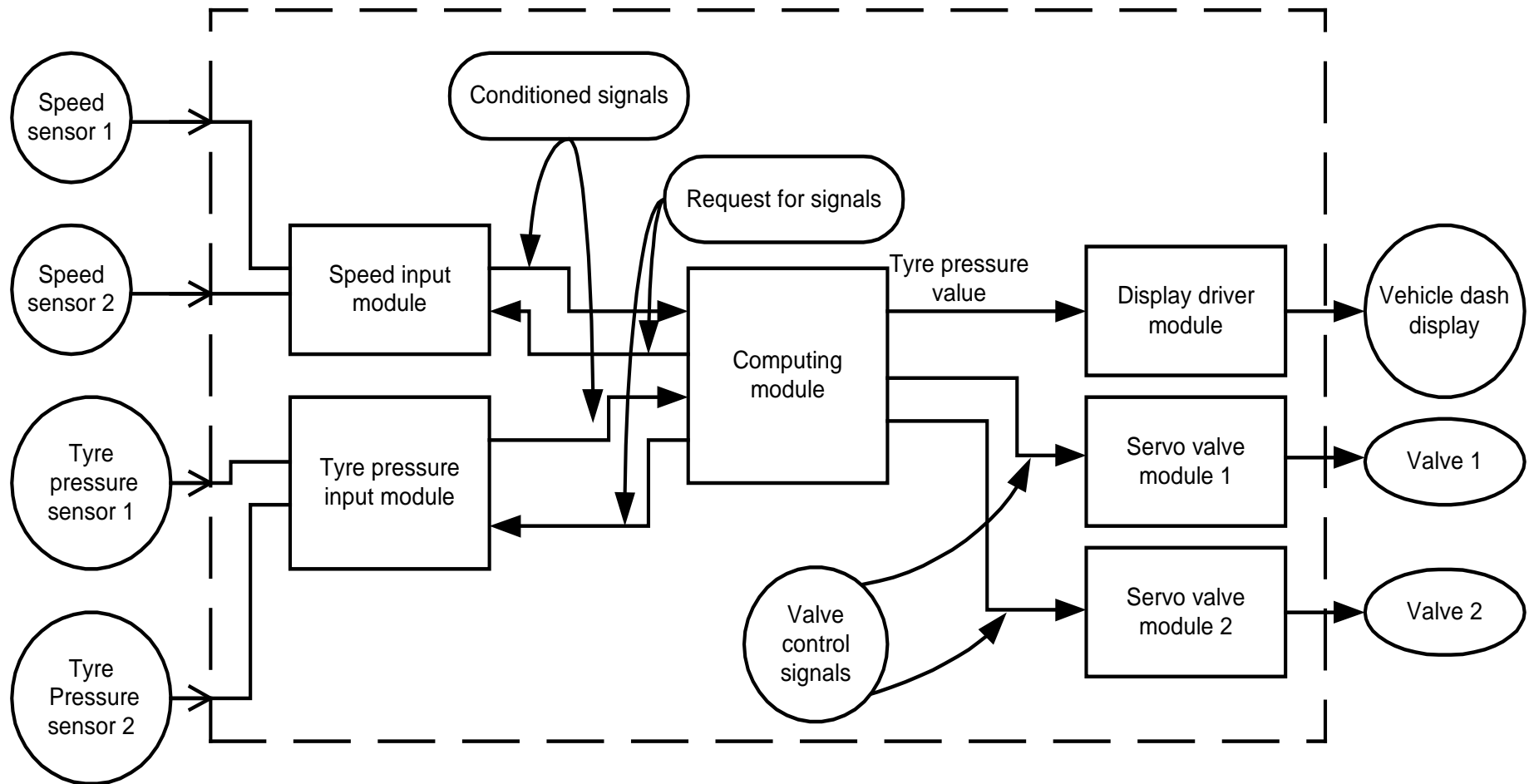


- Modularization
 - the process of forming a complete software system from a set of individual units or modules.
- Module?
 - A standardized part or independent unit in the construction of software.
- But how should a system be modularized?
- Example system: How should **this** be modularized?

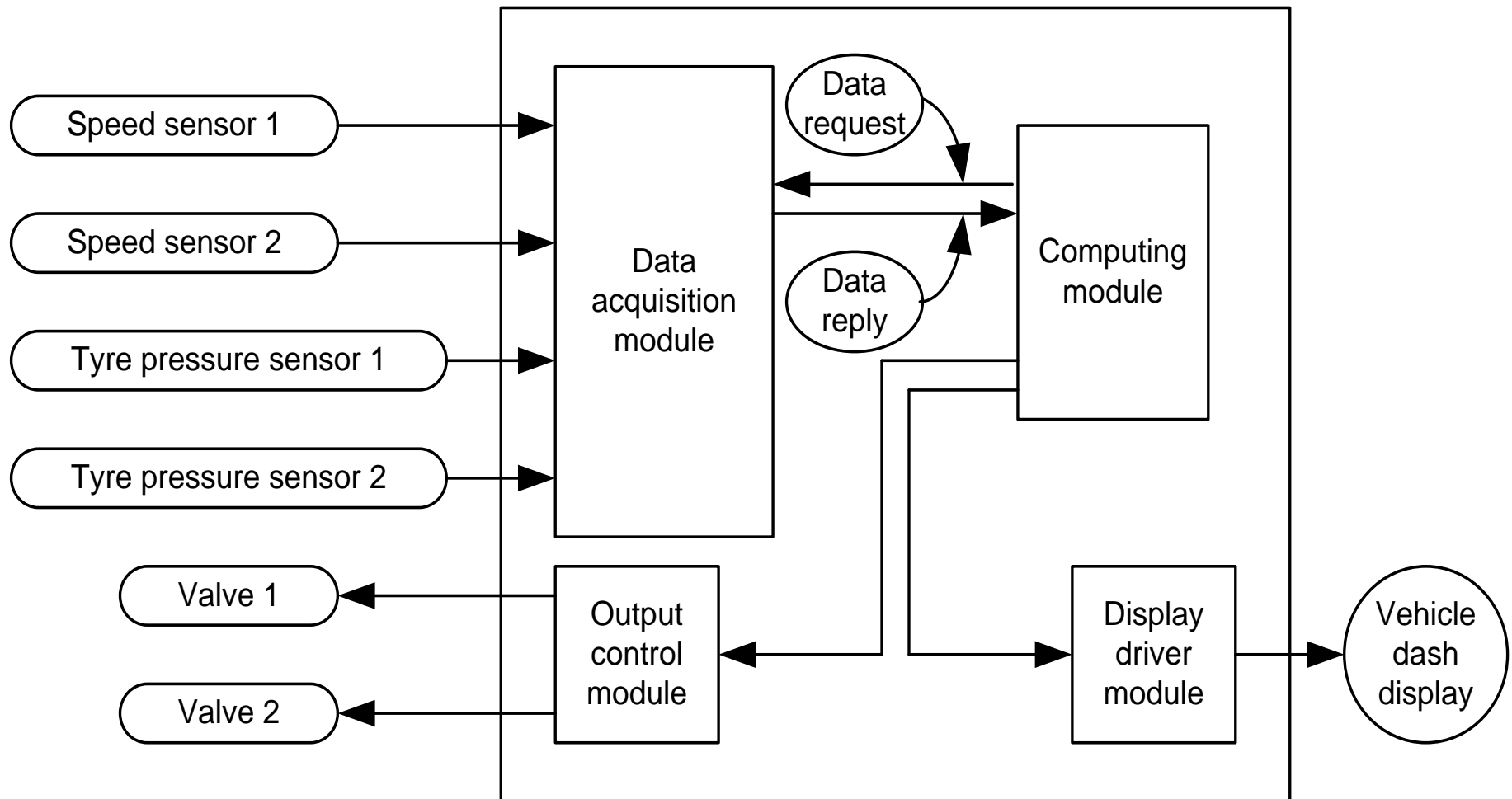
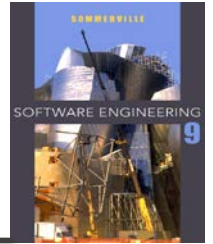


<Example>
The anti-skid braking and
tire-pressure monitoring systems

Modularization – design solution 1



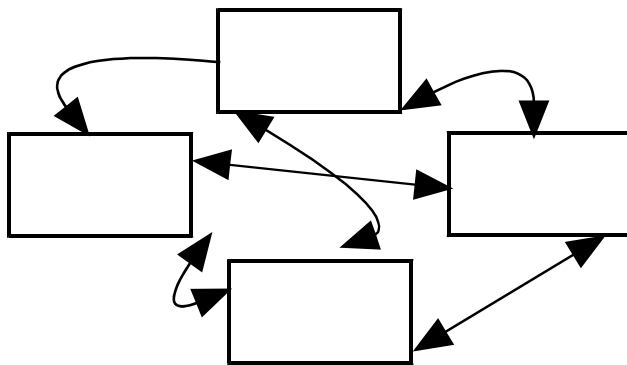
Modularization – design solution 2



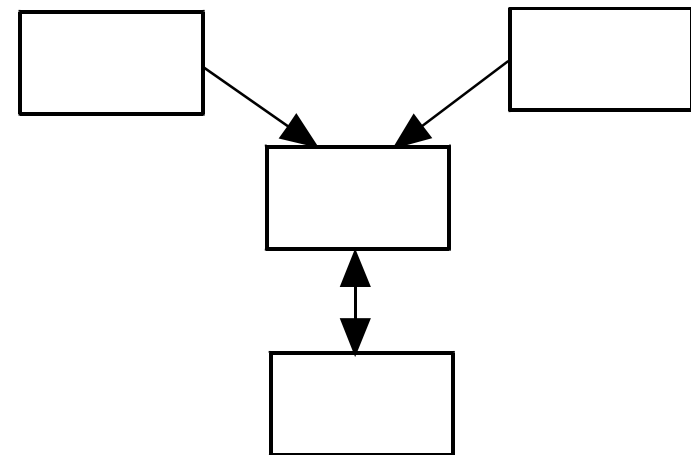
Evaluating modularization 1: coupling - *the outside view*



- Methods to evaluate modularization (1) - **coupling**.
 - *The amount of interaction **between** modules.*
- Check points:
 - number of module interconnections.
 - complexity of module interconnections.
- Desirable features - few interconnections having low complexity.

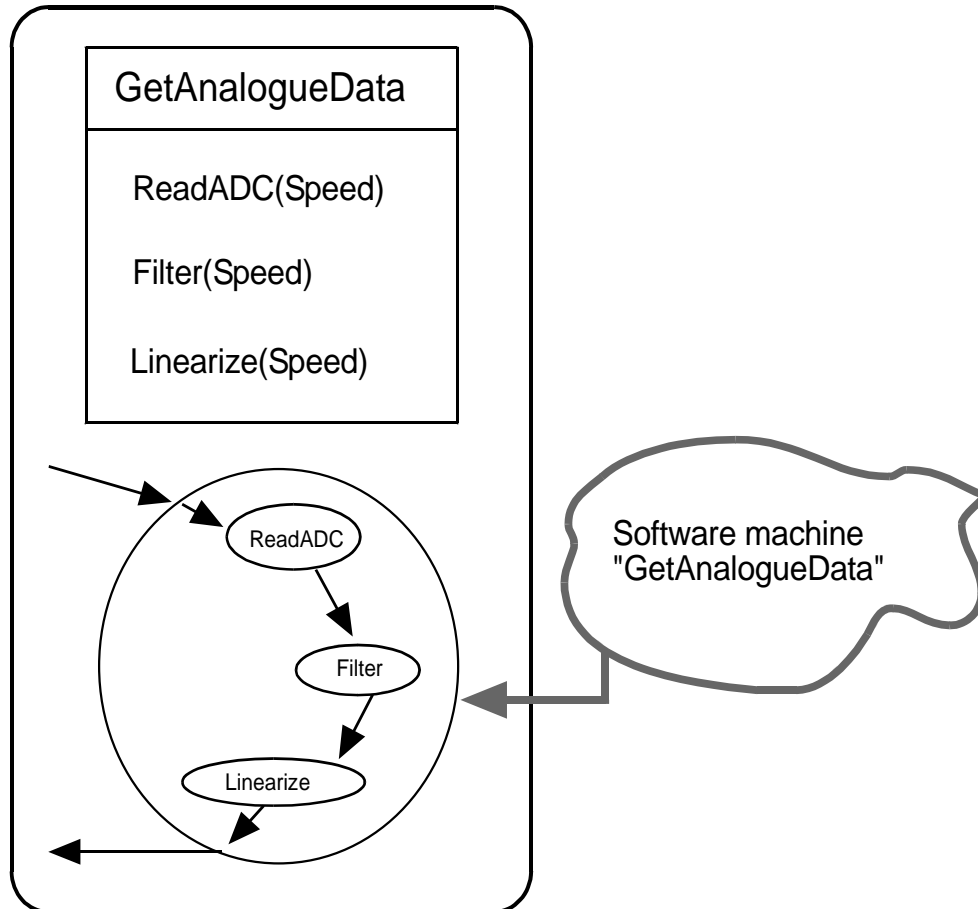


(a) High Coupling



(b) Low Coupling

Evaluating modularization 2: cohesion - *the inside view*



- Methods to evaluate modularization (2)
- **cohesion**.
- Look into the modules.
- How well do the component parts (the internal software machines) relate to each other?
- Strong relationships indicate a good cohesion or “glue” factor.
- This example has high cohesion - remove any part and the machine won't work.
- **High cohesion = “Good”**

Data Flow Design

Data Flow Design



✧ Data Flow Diagram (DFD)

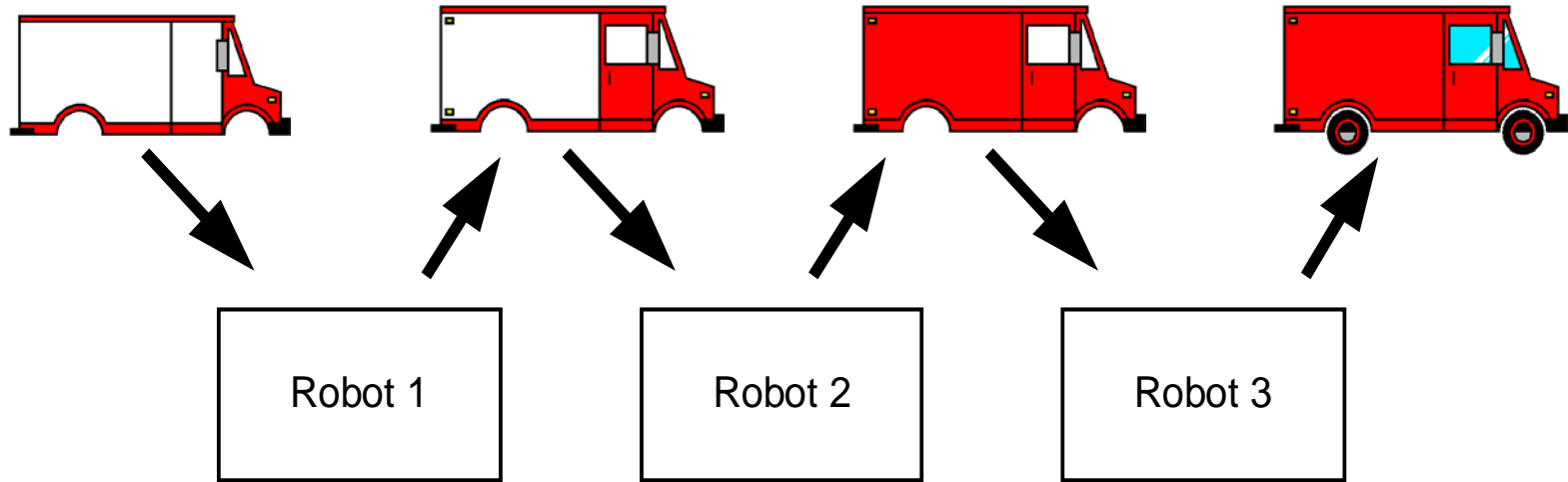
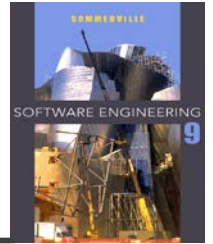
- Enable us to graphically depict the required software processing from a system perspective
- Provide extensive information concerning processing, signals, interfaces, etc
- Act as a design specification for the generation of related structured charts
- Allow us to develop designs in a top-down modularized fashion

✧ Basic ideas of DFD technique

- Top-down design
- Hierarchical Structuring
- Stepwise refinement

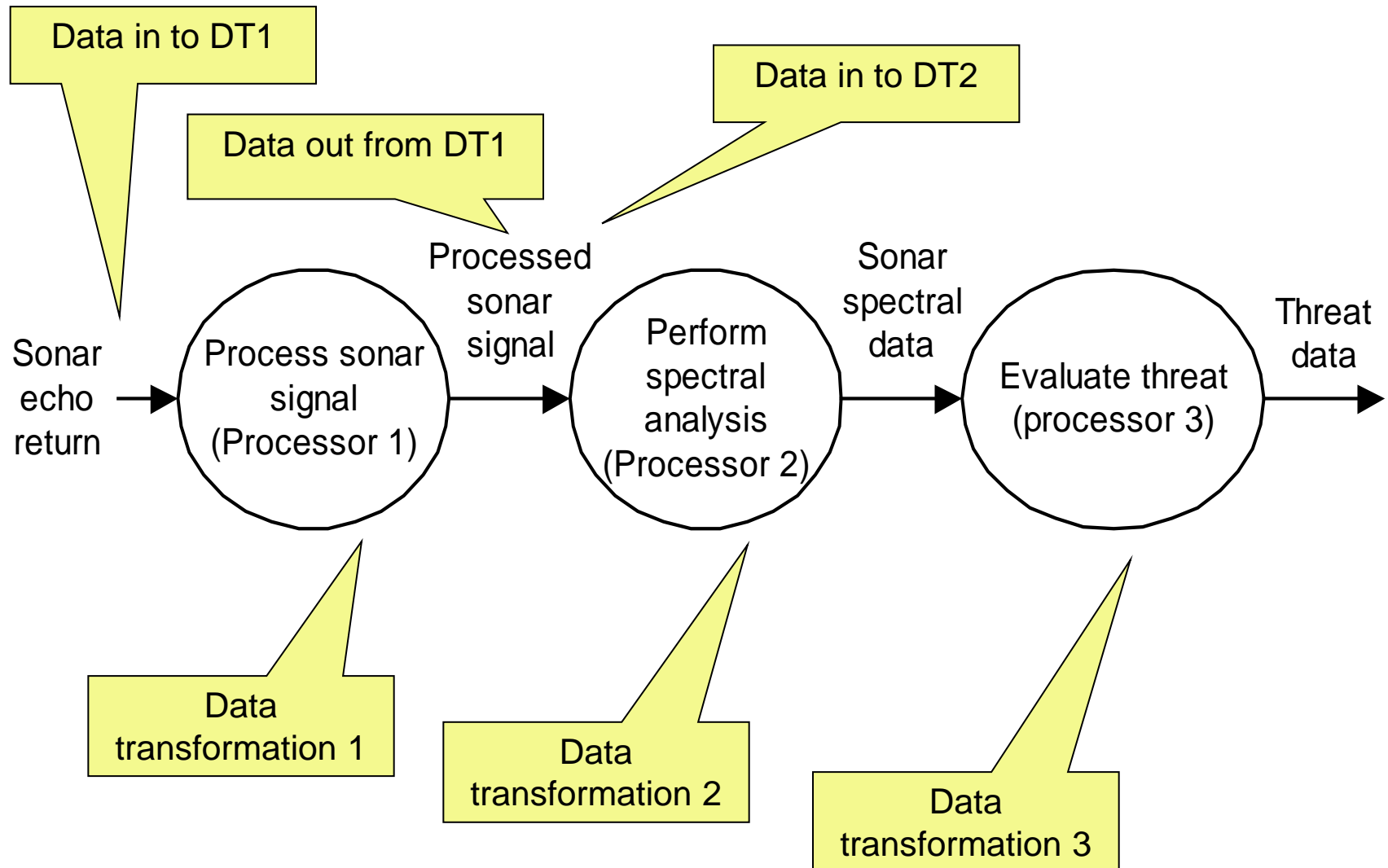
Simple example

- materials flow model



Simple DFD example

- multiprocessor implementation

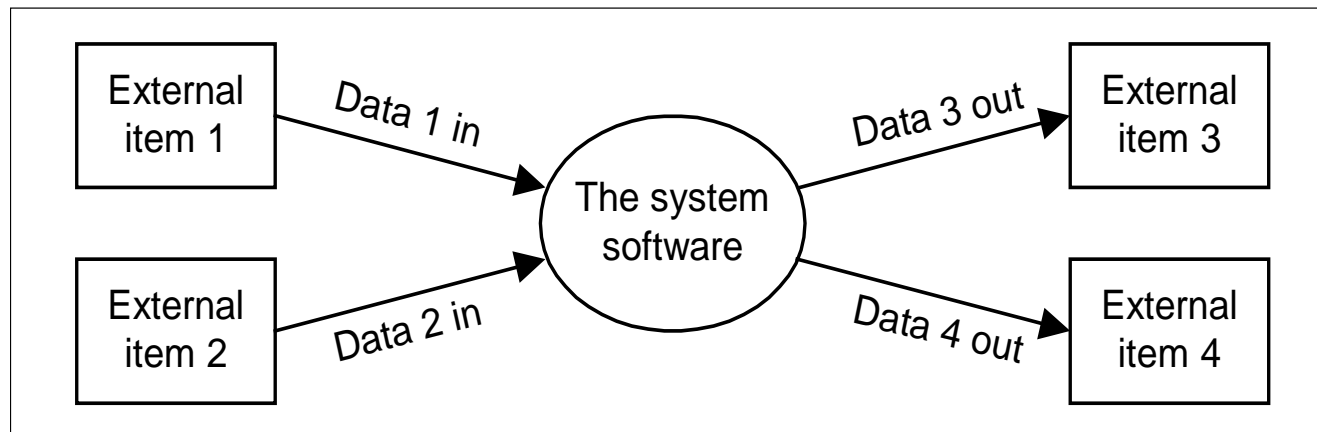


DFD design

- the context diagram



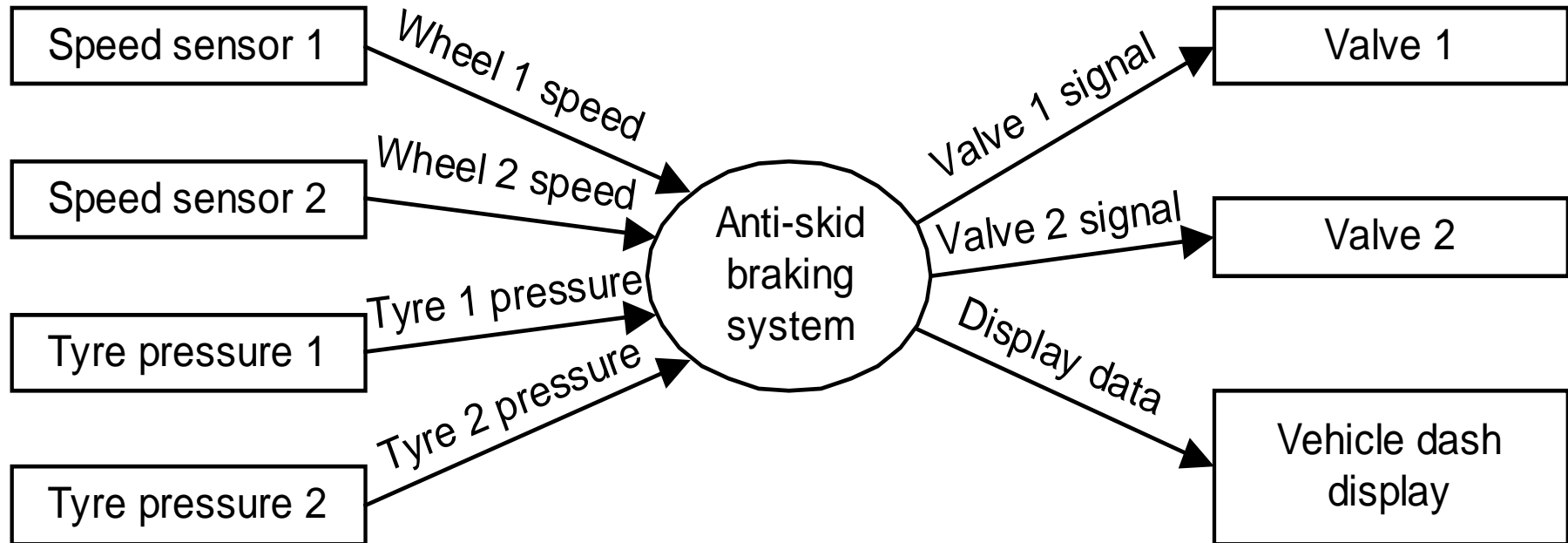
- The purpose of a context diagram is to show the relationship of the system software with its environment.
- It portrays the complete system in its simplest form: a set of external items connected to a software 'black box'.
- It contains:
 - A single processing unit (the 'bubble') representing the complete software system.
 - The external items which this software interfaces to.
 - Data flows into and out of the software.



Example of Context diagram



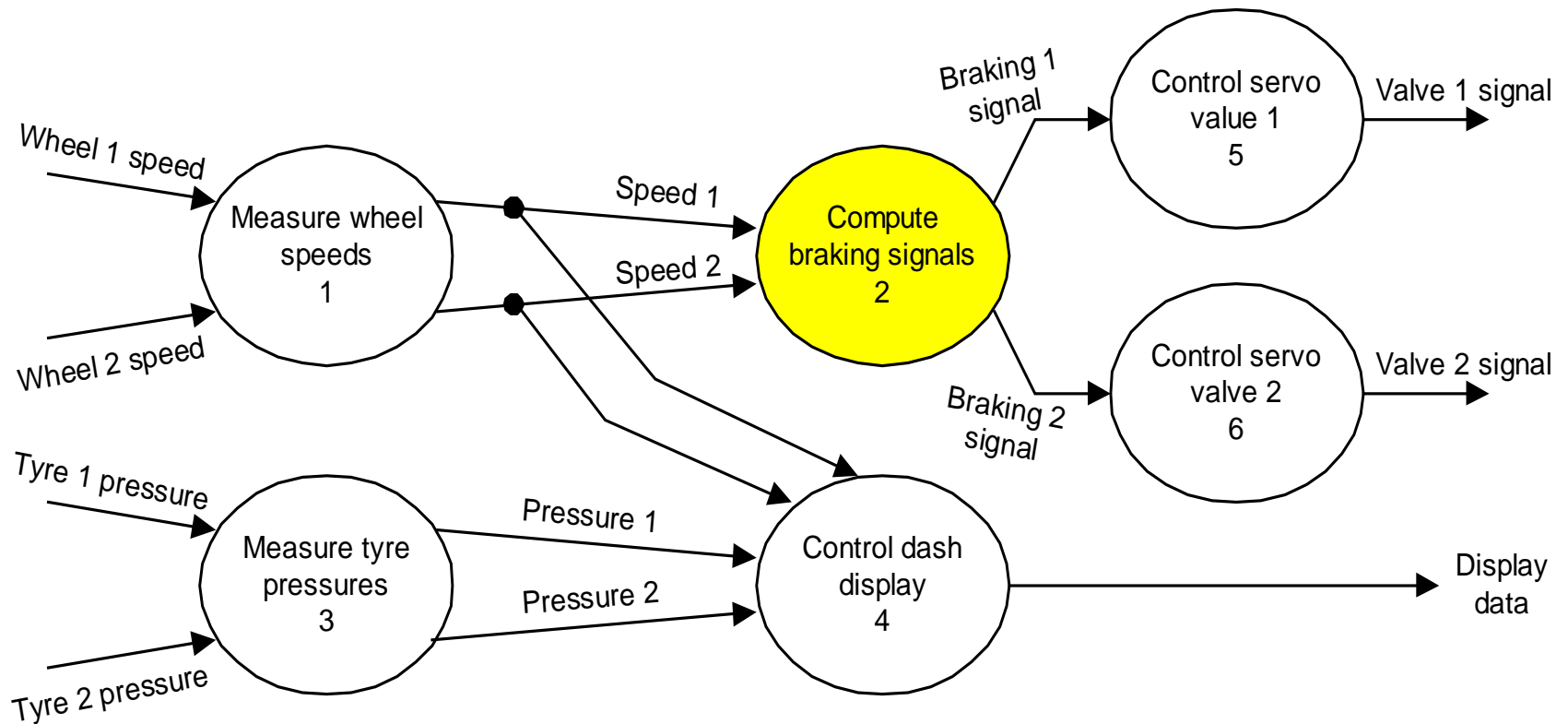
- anti-skid braking system



Example first-level DFD



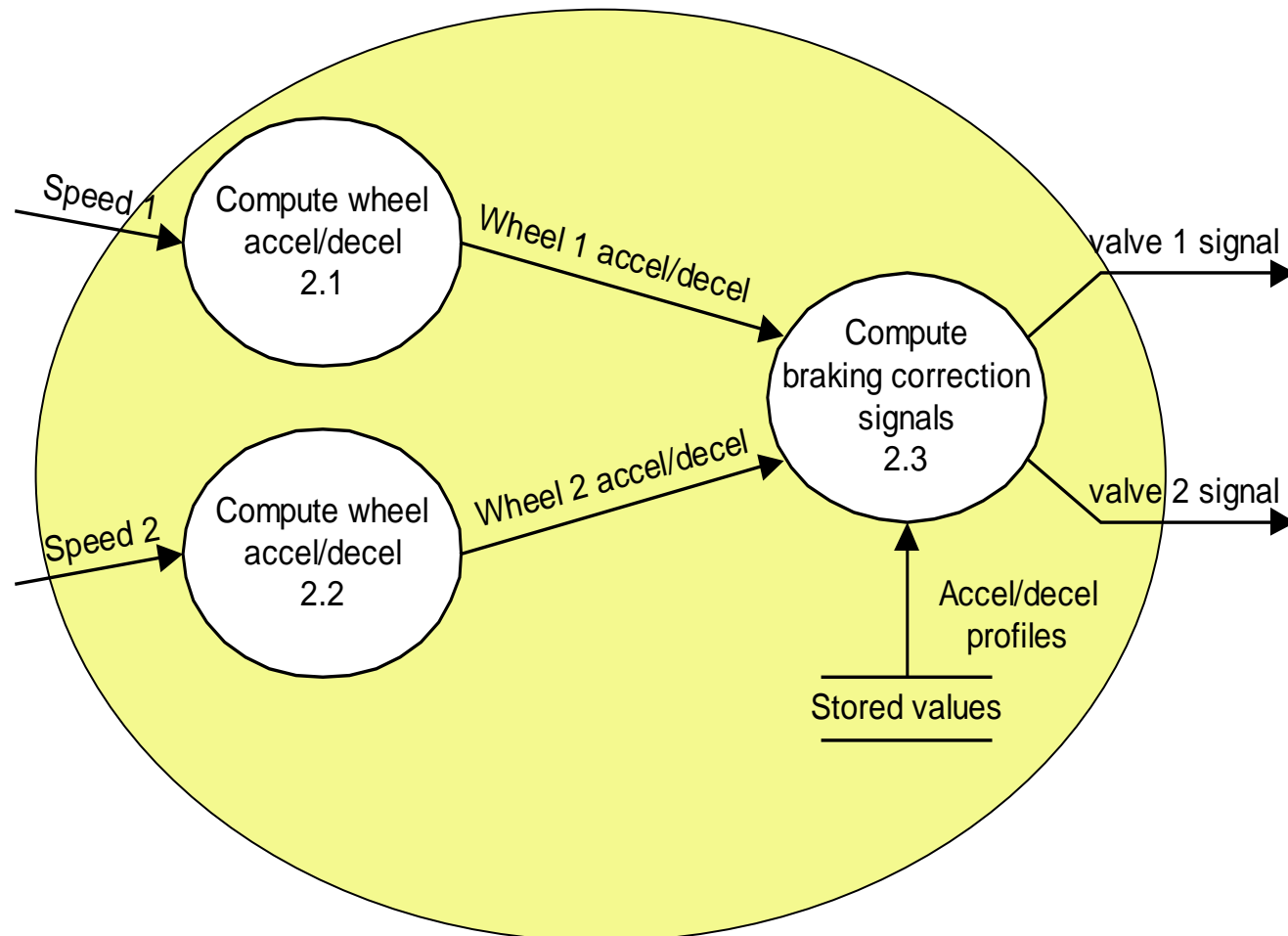
- anti-skid braking system



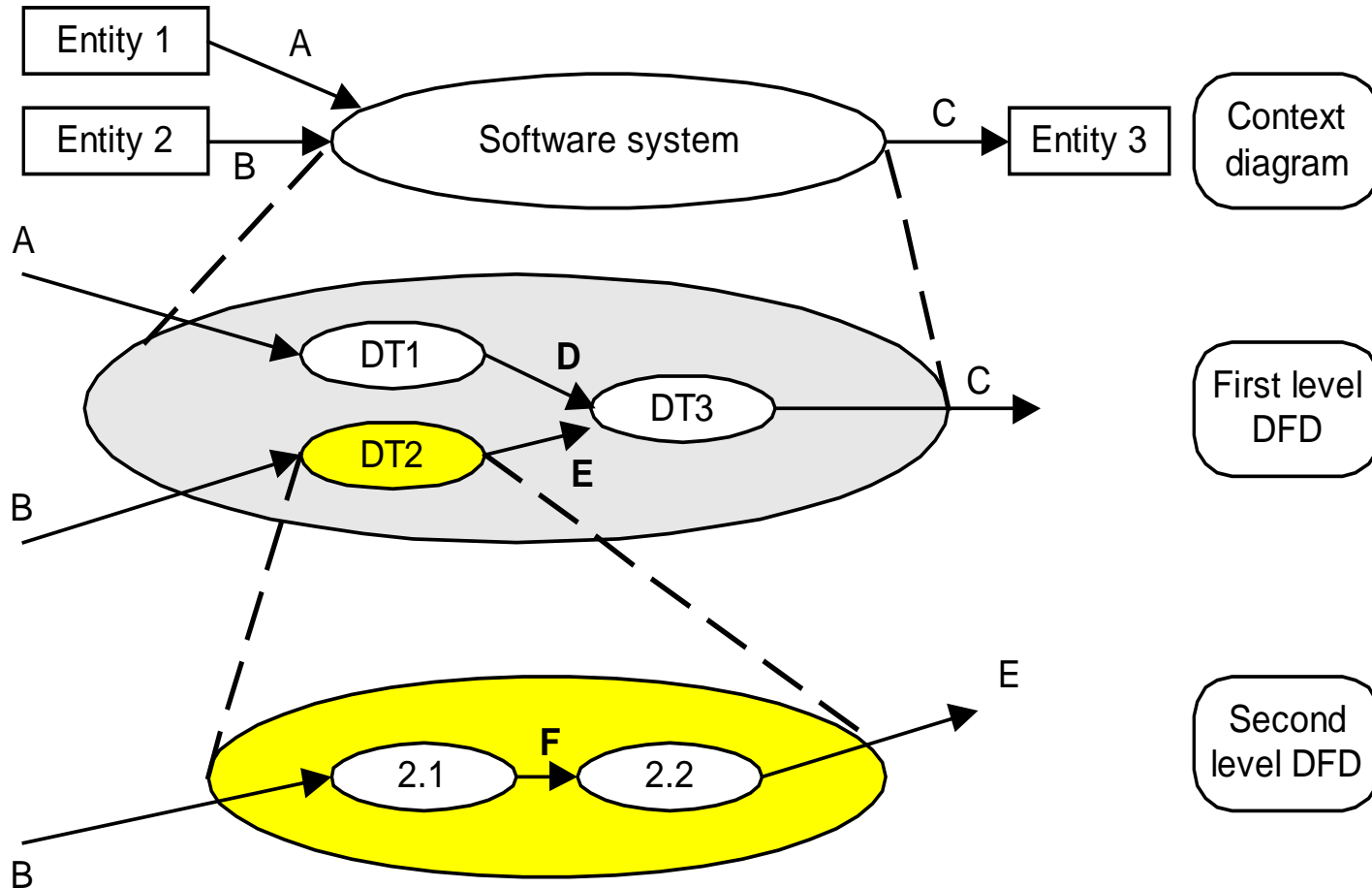
Second-level DFD



- levelled form of the DT 'Compute braking signals'

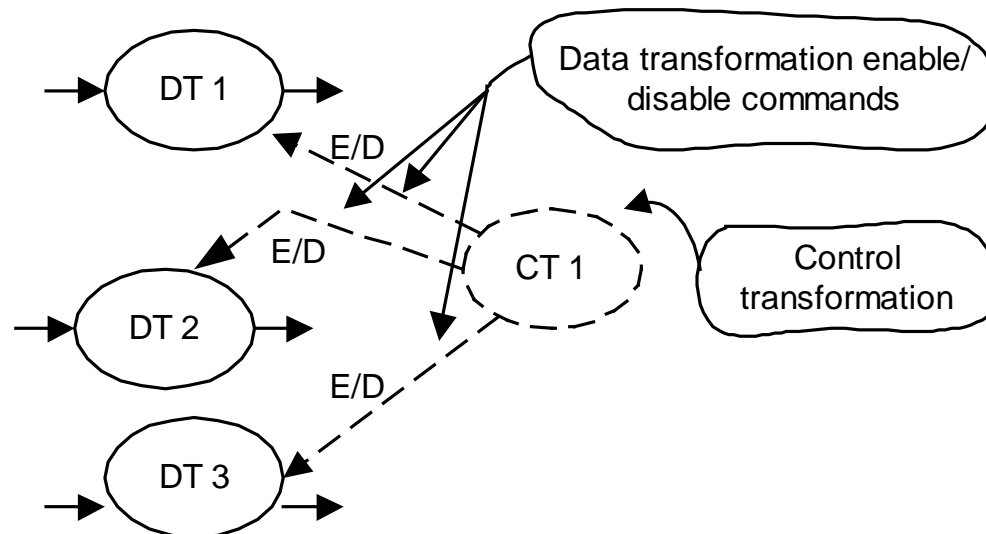


Levelling and balancing in DFD design

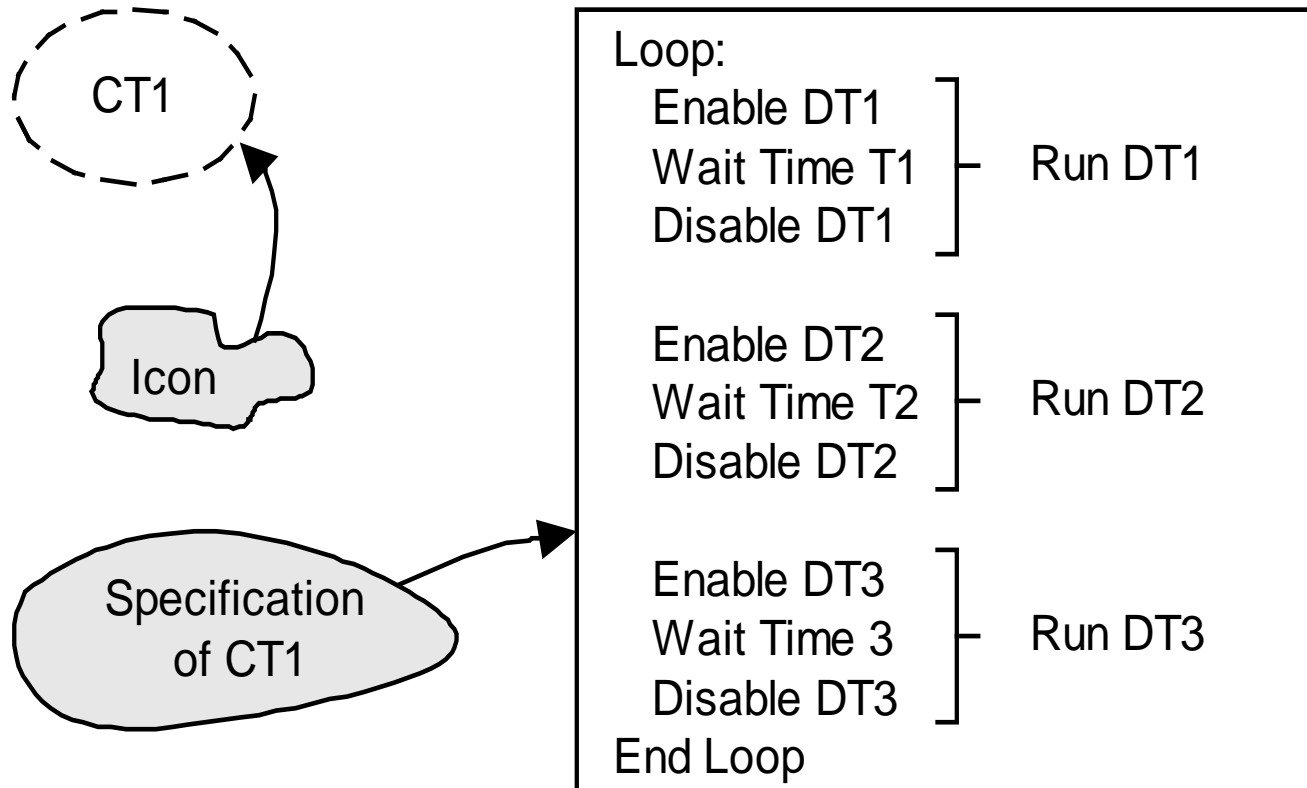


Control flow diagram

- Enactment rules of DFD
 - Will run as soon as all its data inputs are present and
 - Can run simultaneously with all other DTs
- CT(control transformation)
 - is used for handing execution orders of DTs



Control transformation - icon and specification



Object-oriented design

- Object

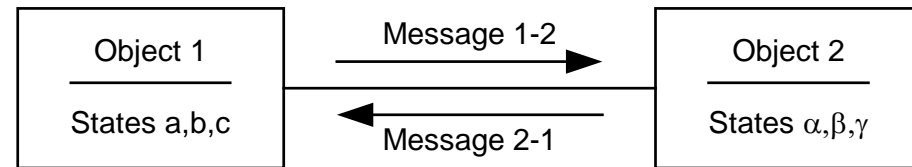
A software machine which has a number of defined operational states and a defined means to access and change these states

A change of state is achieved by passing a message into an object

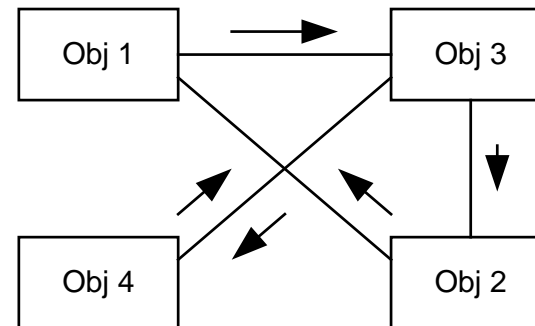
- Why use OO approach to software design

For many real-world applications, software objects map quite naturally onto system models

Concurrency can be shown simply and clearly using OOD



(a)



(b)

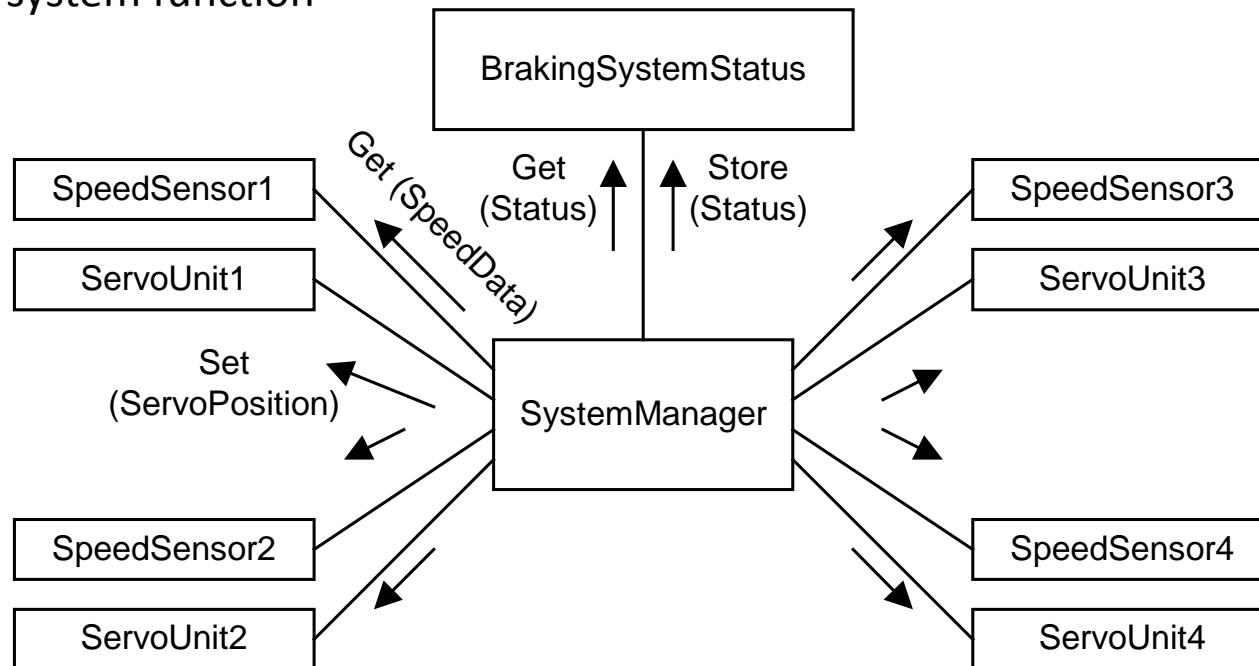
Steps of OOD



- ✧ Identify the objects and their features
- ✧ Identify the relationships between objects
- ✧ Define the communication (messaging) between objects
- ✧ Define the interface of each object
- ✧ Implement the objects

Identification Process of OOD

- Identification of objects is a key and difficult work
- Sample mapping rules
 - Object to physical device
 - Object to abstract device
 - Object to system function



Object Interfaces



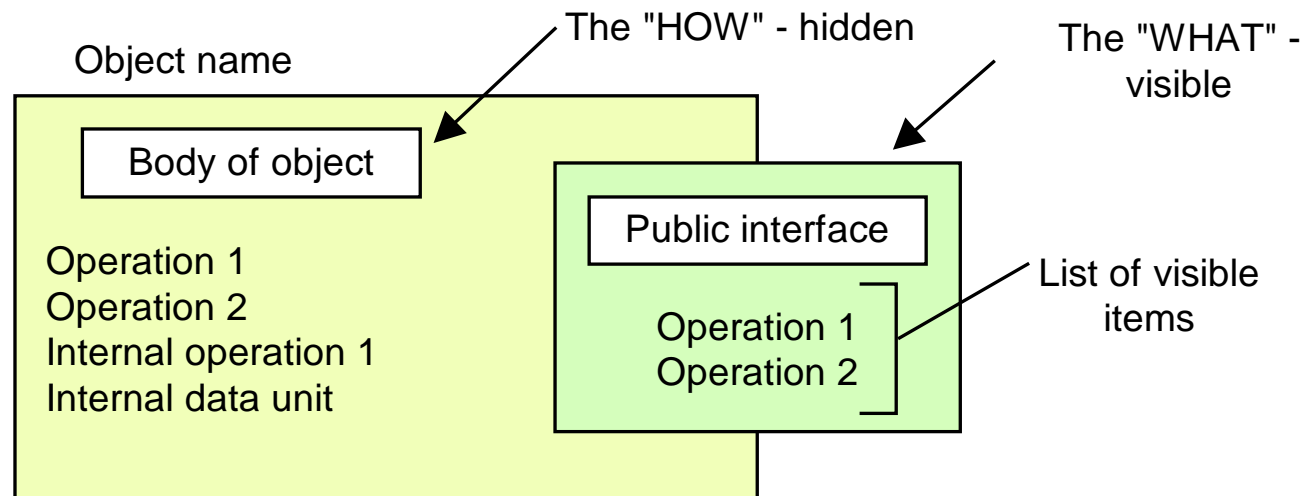
- Components of objects

Interface

- The visible section describes essentially the services it provides

Body

- The hidden section implements the required functions of the object



Object Rending

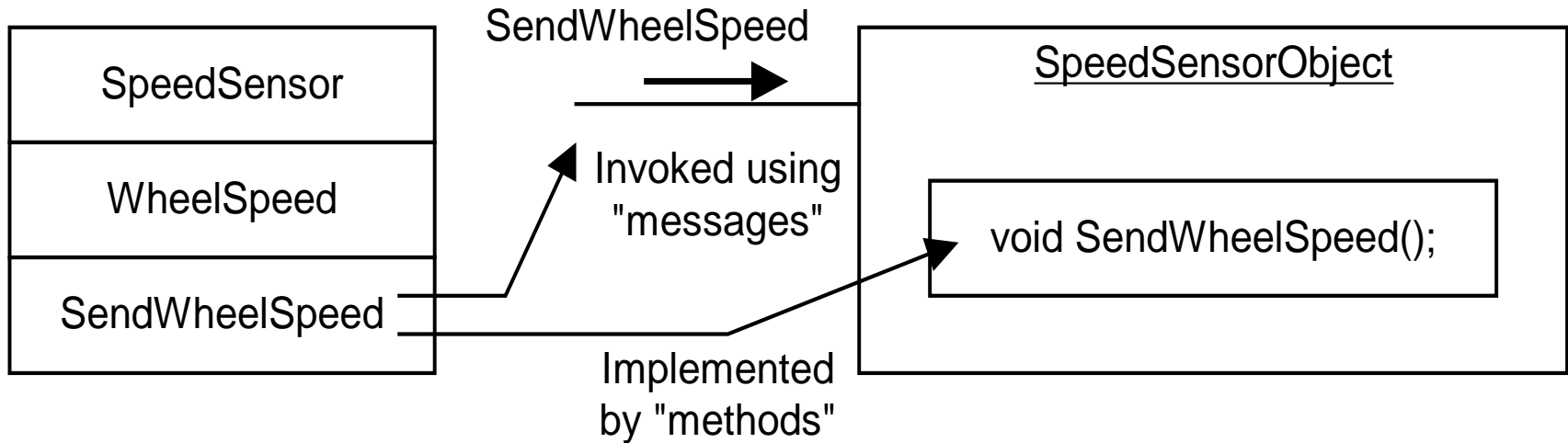


- ✧ A depicted class diagram has the following structure:
 - Name compartment (mandatory)
 - Attributes compartment (optional)
 - Operations compartment (optional)
 - Responsibilities compartment (optional)
- ✧ Every class must have a distinguishing name.

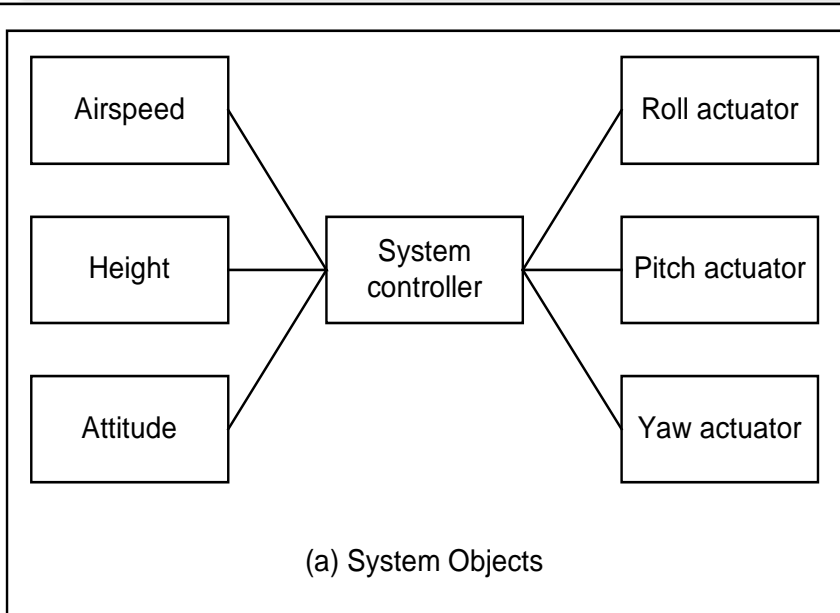
Rectangle
p1:Point p2:Point
«constructor» Rectangle(p1:Point, p2:Point) «query» area (): Real aspect (): Real ... «update» move (delta: Point) scale (ratio: Real) ...

WashingMachine
+ modelName: String - serialNumber : Integer + capacity : Integer
+ addClothes() + removeClothes() + addDetergent() + turnOn()
Taking dirty clothes as input and producing clean clothes as output

Concept - methods and messages



Inheritance, productivity and reuse



<u>Sensors</u>	<u>Controllers</u>	<u>Actuators</u>
Airspeed	System	Roll
Height		Pitch
Attitude		Yaw

(b) Grouping of system objects

One way to increase both productivity and reuse is to minimize the number of classes in the first place.

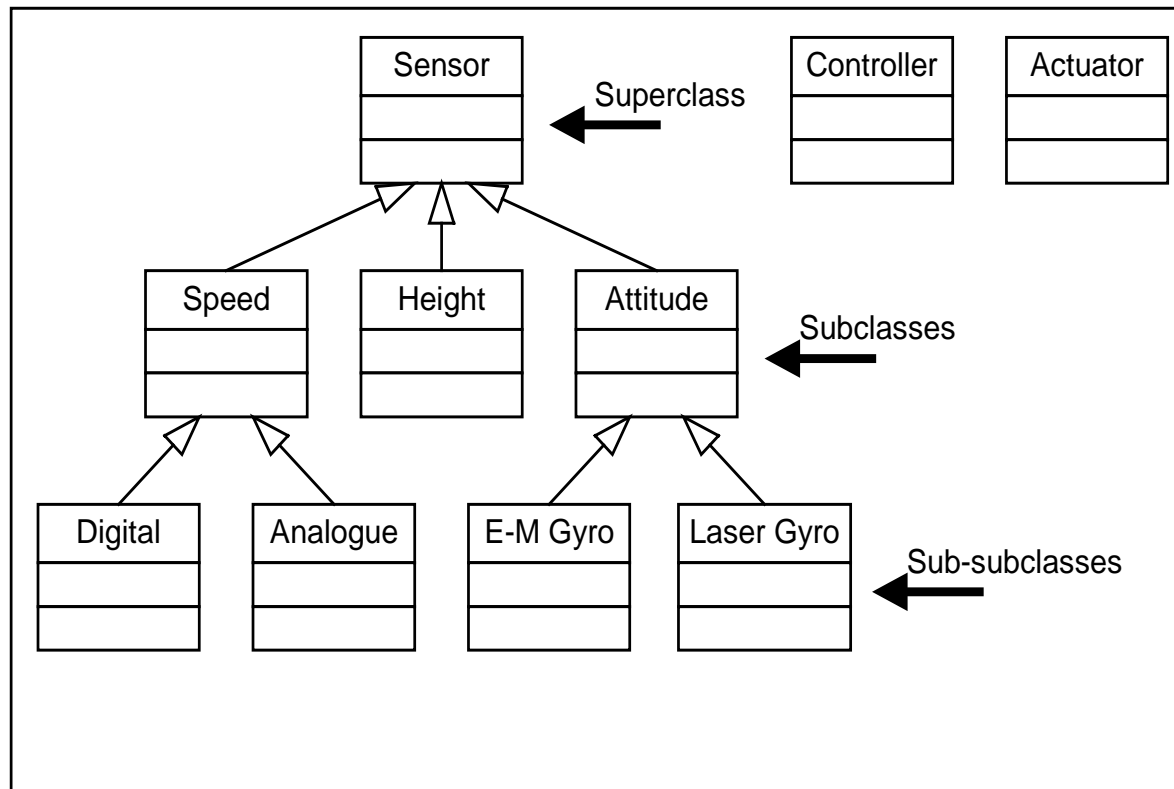
- Object classification (grouping) must be carried out. Common factors must be identified.
- Grouping is based on:
 - function and behaviour.
 - qualities (attributes).
- Example system: initial grouping produces three classes: sensor, controller and actuator.
- Actuator objects are identical but sensors differ in detail.
- How can this be handled?

Class structuring

- subclasses and superclasses

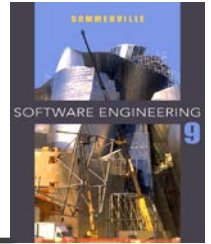


- Here one class only is needed to specify actuator objects.
- The sensors may be specified using either:
 - A set of distinct classes or
 - Inheritance techniques (specialization).

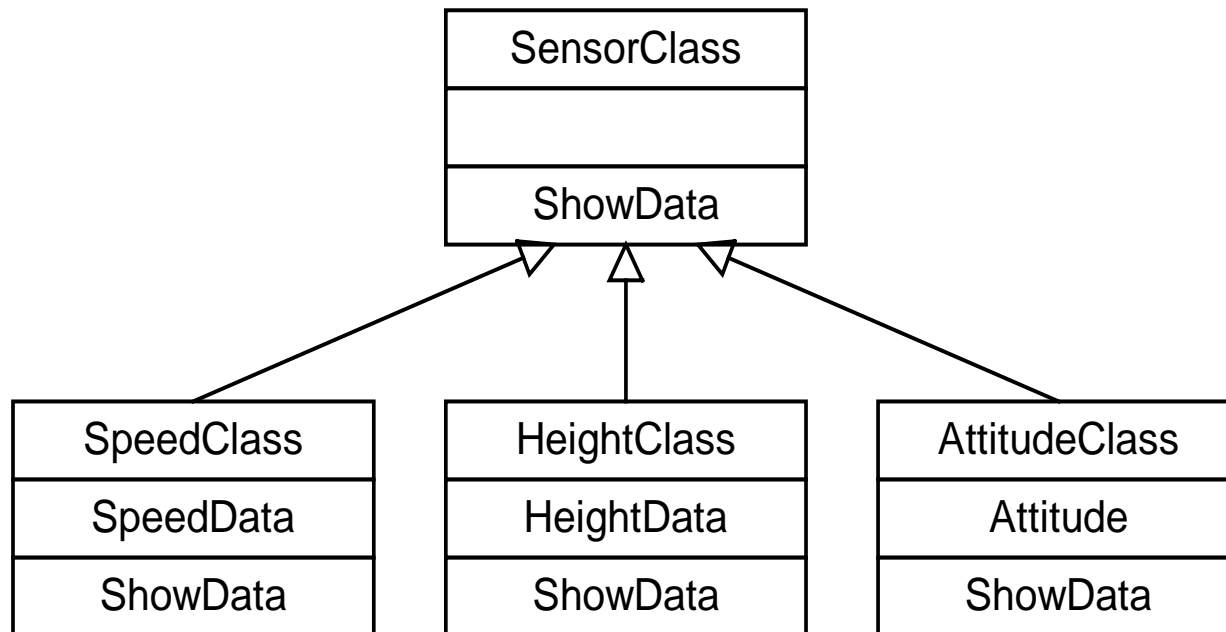


Overriding operations

- Flexibility through polymorphism



- Operation overriding - a technique which enables us to:
 - provide a consistent interface operation for all objects.
 - call on this operation as and when required.
 - allow each subclass to actually have different implementations (methods) of the operation.
 - guarantee that the correct method is automatically executed - *polymorphism*.



Dynamic polymorphism



- Suppose we wish to have the choice of operation made:

- During program execution and
- Dependent on program conditions.

- Use a source code statement of the form:

`Sensor.ShowData;`

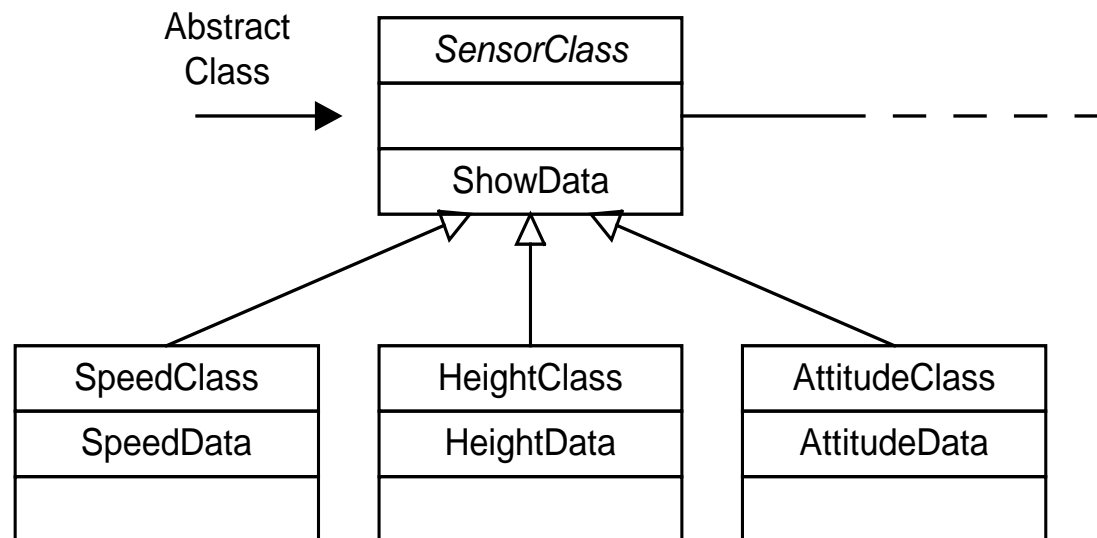
- The underlying general principles:
 - ◆ The compiler knows (from the declaration format) that the *actual* object called at run-time is undefined.
 - ◆ It also knows that it belongs to either SpeedClass, HeightClass or AttitudeClass.
 - ◆ At some stage of program execution, 'Sensor' is replaced by the actual object identifier.
 - ◆ When the statement *Sensor.ShowData* is reached, the run-time code works out which operation should be invoked - *dynamic polymorphism*.

The use of an abstract class

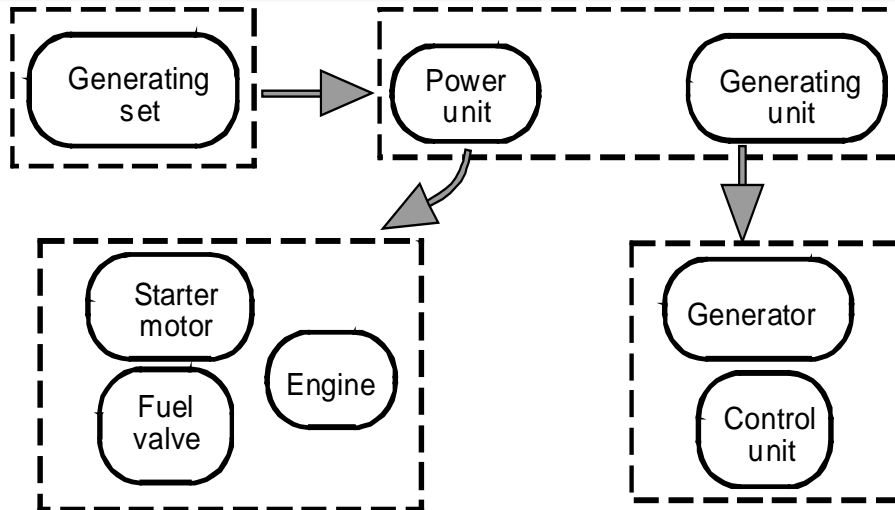
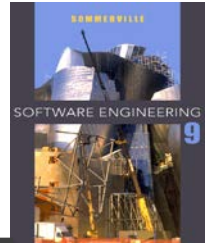
- Inheritance as a specification technique



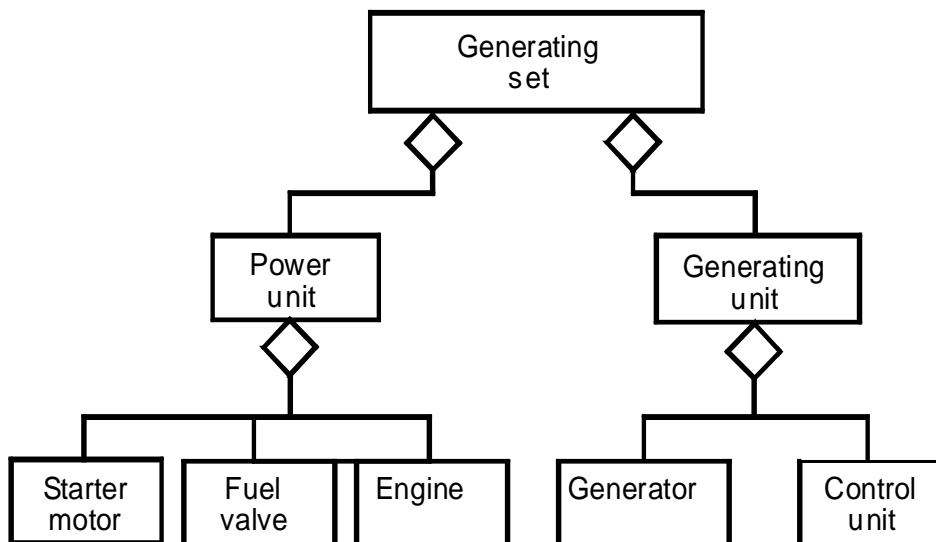
- Objective - to provide a standard interface for all subclasses.
- Why?
 - ◆ Reusing the interface gives a consistent view of objects.
 - ◆ Reusing the interface - as new subclasses are added it ensures that their interfaces will be correct.



Aggregation

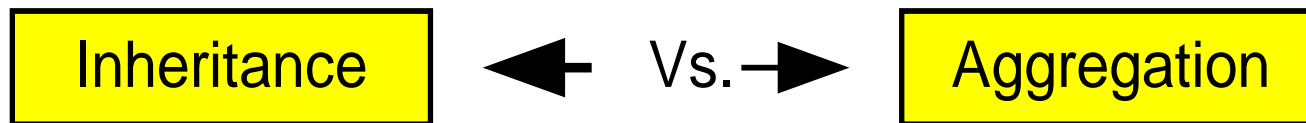


(a) Objects which consist of objects



(b) Class aggregation diagram

Using inheritance and aggregation



Use as

- An extension mechanism
- An interface specification mechanism

Use as

- A structural specification mechanism

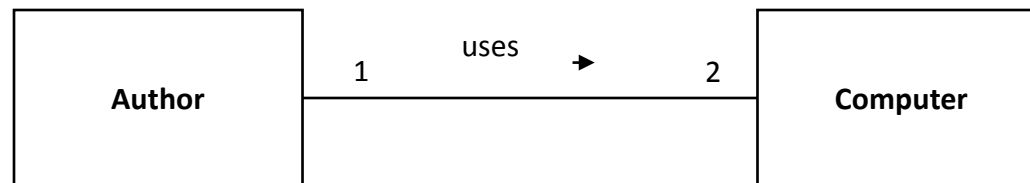
Objects Relationships

- Association

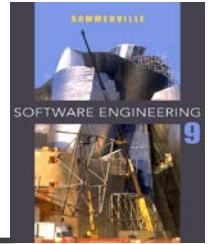


✧ Association

- OOA associations represent conceptual relationships among problem concepts.
- Association properties
 - Multiplicity
 - Role
 - Constraints
 - Predefined properties
- *Example:* “An author uses two computers

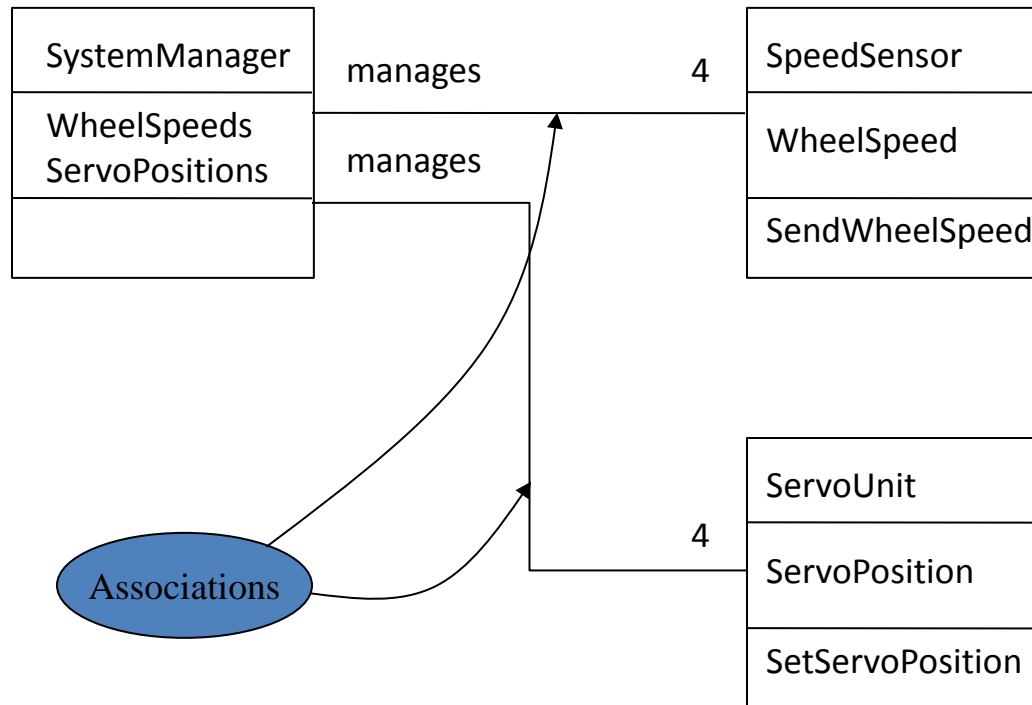


Class Diagram

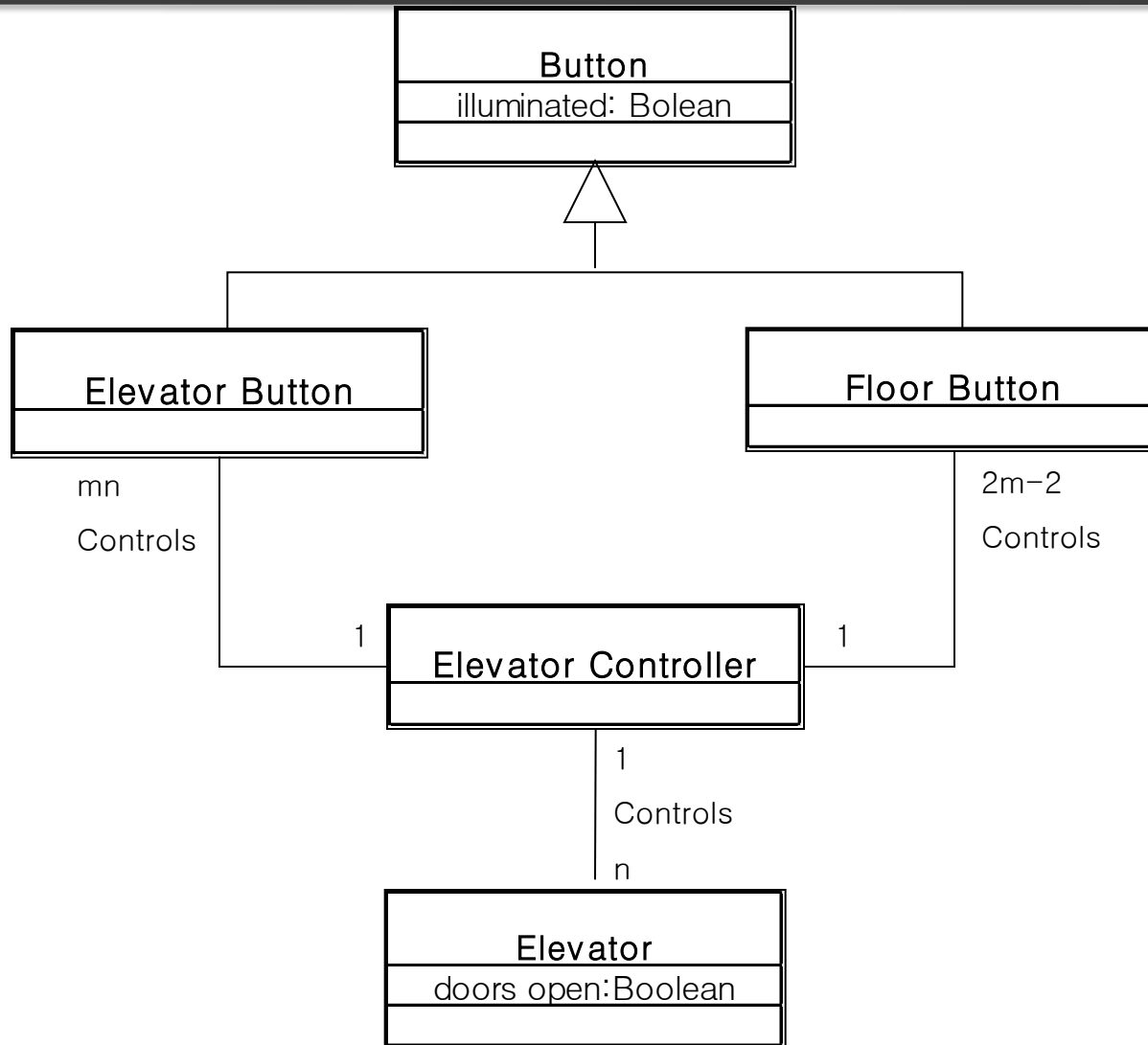


✧ Class diagram

- Represents the relationships among classes



Example of Class Diagram I



Example of Class Diagram II

