Analysis, Design, and Software Architecture (BDSA)

Paolo Tell

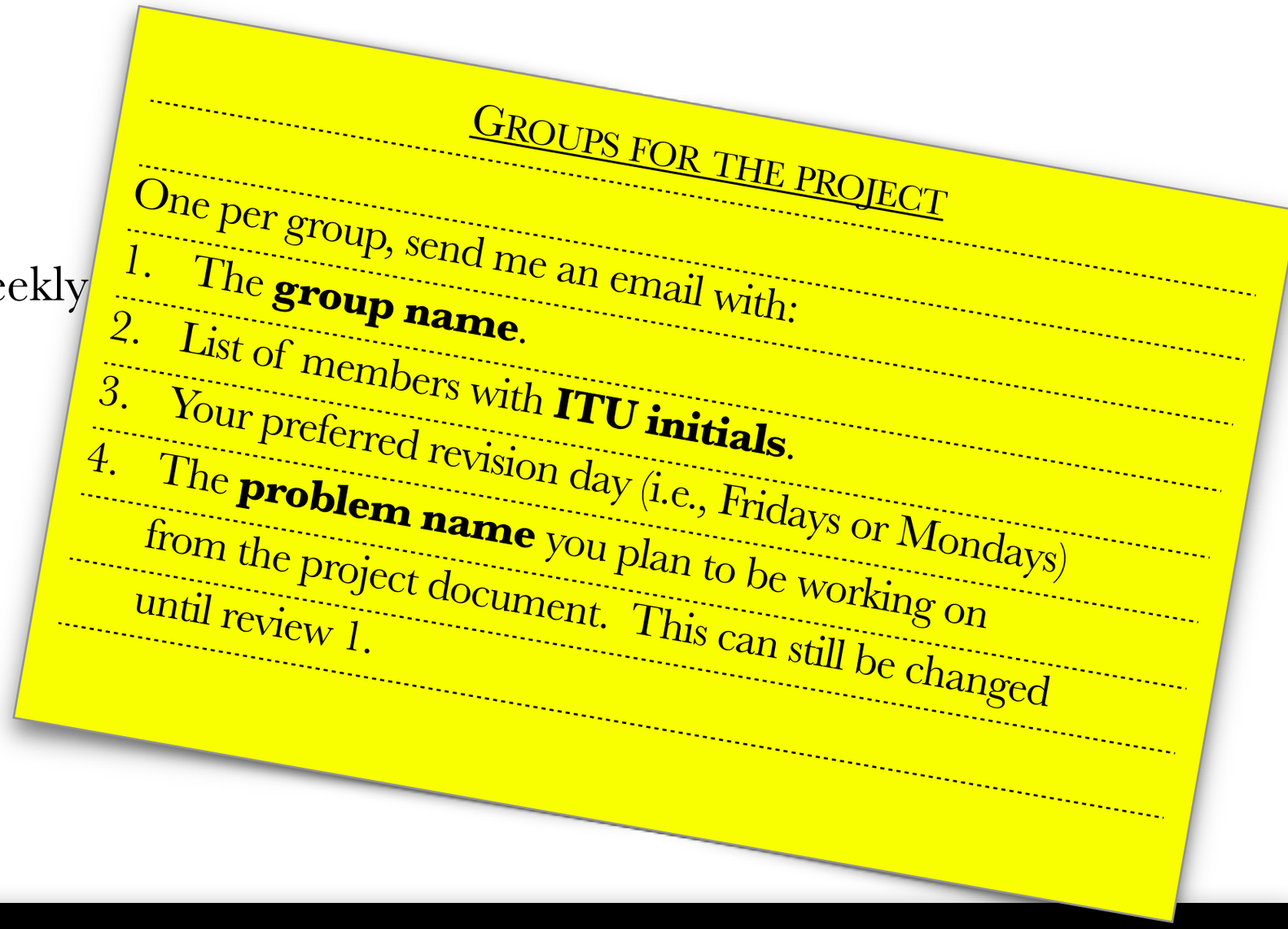
Architectural and Object Oriented Design

IT University
of Copenhagen

- Week 42 — Vacation, no weekly assignment
- Week 43 — Project starts!
- Week 44 — Trial exam

Logistics

- Week 42 — Vacation, no weekly
- Week 43 — Project starts!
- Week 45 — Trial exam (?)



Logistics

Go to **www.menti.com** and use the code **4372 9680**

Question **1** of **12**

Quiz

IT University of Copenhagen

- Confusing. A good software should explain what it is doing.

1. Rigidity. To make a modification, you need "touch" a lot of other stuff.

2. Fragility. You change something somewhere, and it breaks the software in unexpected areas that are entirely unrelated to the change you made.

3. Immobility (no-reusability). The software does more than what you need. The desirable parts of the code are so tightly coupled to the undesirable ones that you cannot use the desirable parts somewhere else.

Recap - Symptoms of bad software

- Single responsibility principle: “a class should only have one, and only one, reason to change”.
- Open/closed principle: “software entities should be open for extensions but closed for modifications”. (Bertrand Meyer 1988)
- Liskov substitution principle: “derived classes should be usable through the base class interface, without the need for the user to know the difference”. (Barbara Liskov 1987)
- Interface segregation principle: “many client-specific interfaces are better than one general-purpose interface”.
- Dependency inversion principle: “depend upon abstractions, do not depend upon concretions”.

Recap - SOLID

Outline

- Literature
  - [OOSE] ch. 6-(7-8)
  - [SE9] ch. 6-7

- Topics covered:
  - Software Architecture
  - Object-Oriented Architecture & Design
  - (Architectural and Design Patterns)

- Special:
  - If you want to be a software architect?
    - [SA3] Software Architecture in Practice (3rd ed.) by Len Bass, Paul Clements, and Rick Kazman.
    - … is a "must have / must read"
    - … is often referenced in [SE10]

Architecture design

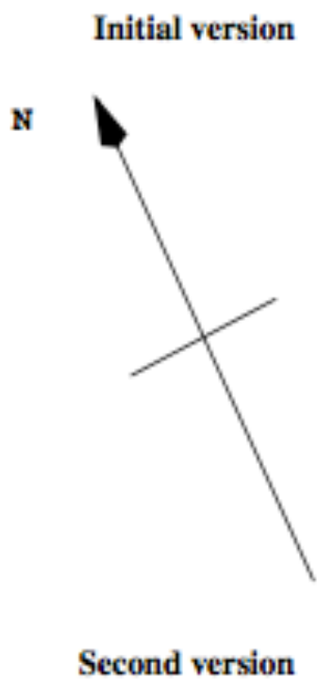
An example from architecture

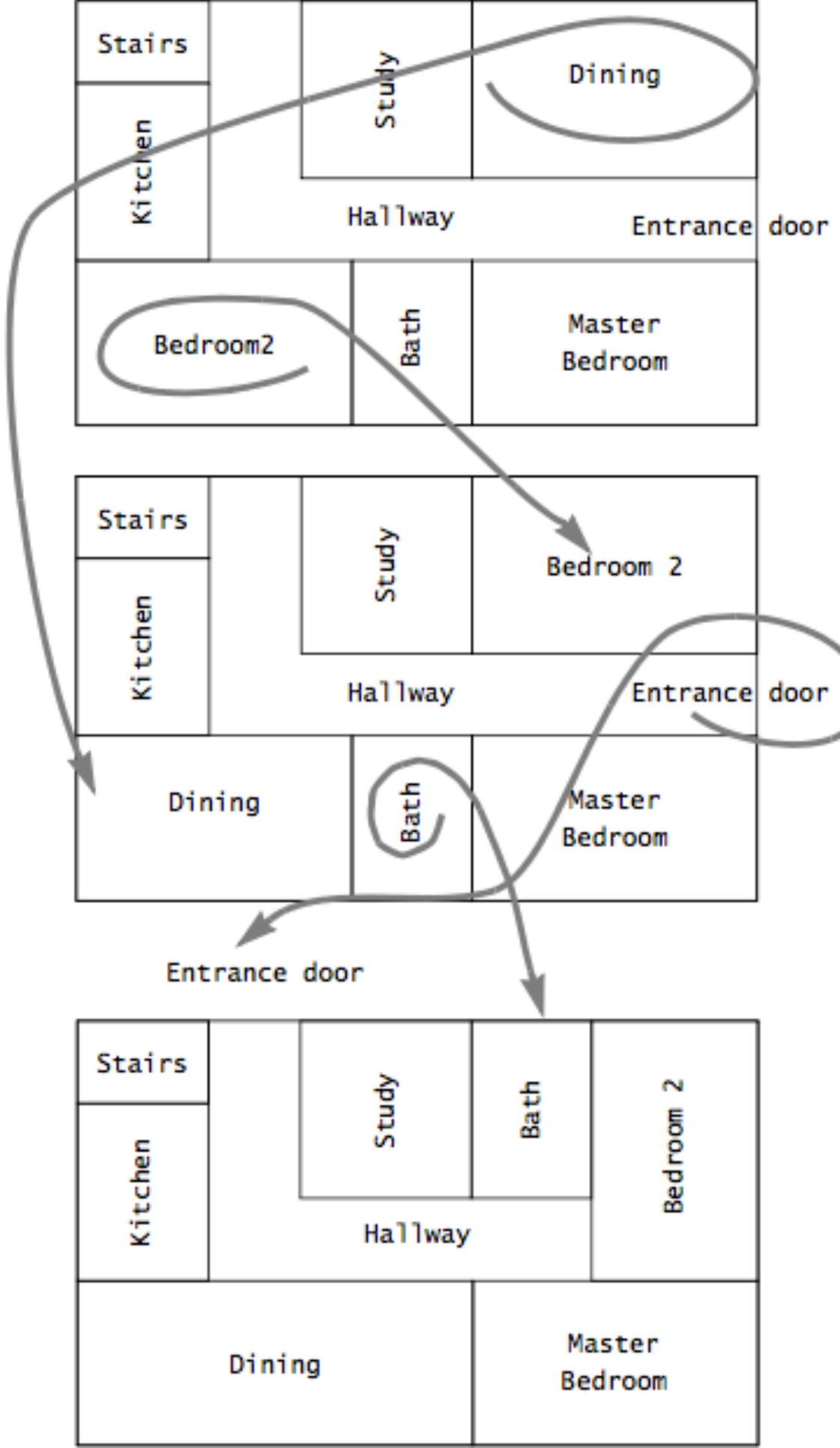
1. This house should have two bedrooms, a study, a kitchen, and a living room area.
2. The overall distance the occupants walk every day should be minimised.
3. The use of daylight should be maximised.

1. This house should have two bedrooms, a study, a kitchen, and a living room area.

2. The overall distance the occupants walk every day should be minimised.

3. The use of daylight should be maximised.





Architectural design

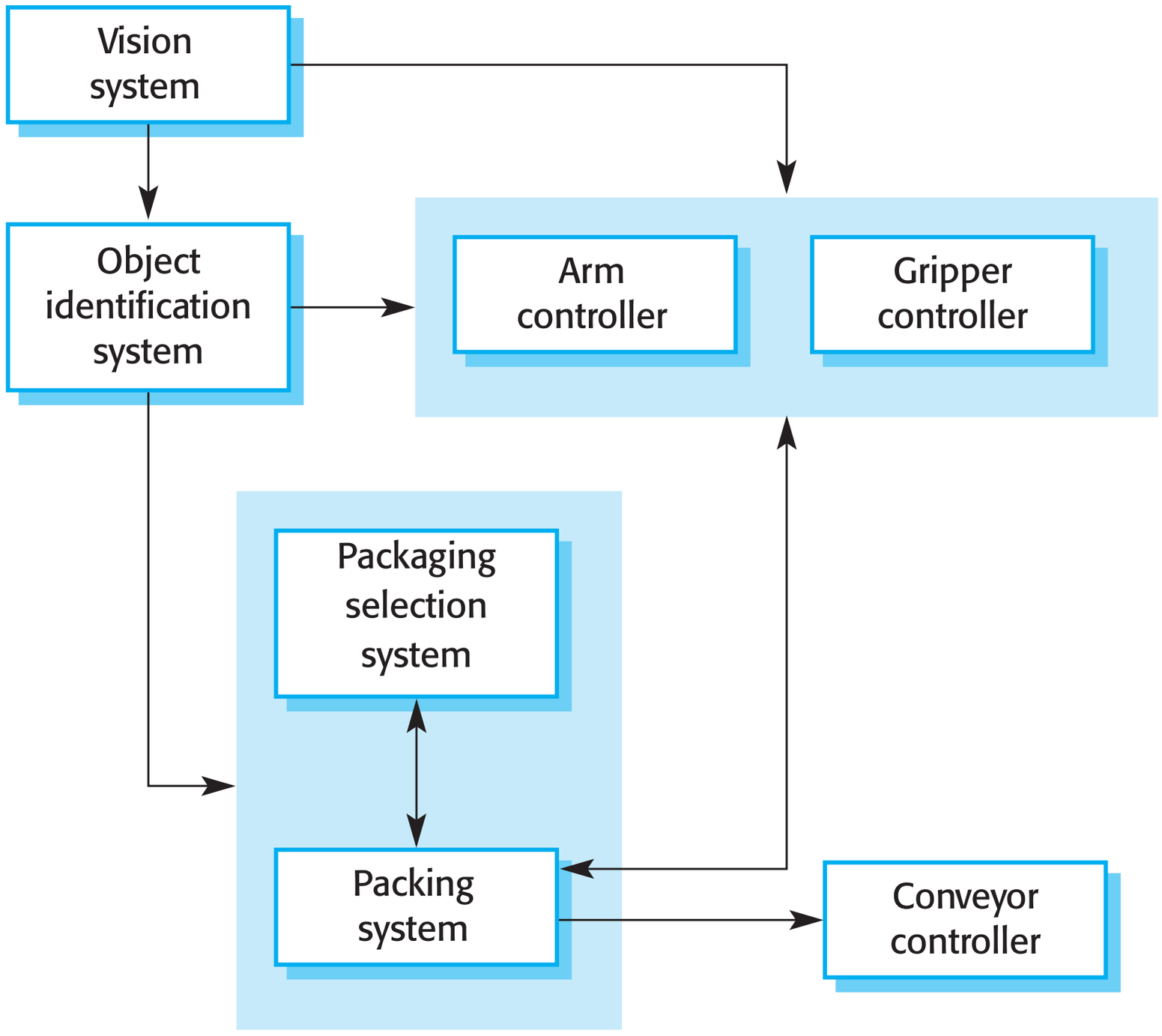
- An early stage of the system design process.
  - Represents the link between specification and design processes.
  - Often carried out in parallel with some specification activities.
  - It involves identifying major system components and their communications.

- Advantages
  - Stakeholder communication
    - Architecture may be used as a focus of discussion by system stakeholders.
  - System analysis
    - System qualities or non-functional requirements having an impact at the system level can be discussed without focusing on the details of the sub-systems.
  - Large-scale reuse
    - The architecture may be reusable across a range of systems.

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.

Example of a system architecture: Packing robot control system



Box and line diagrams

- Box&Line diagrams
  - Very abstract – they do not show the nature of component relationships nor the externally visible properties of the sub-systems.
  - However, useful for communication with stakeholders and for project planning.

- As an architect you have to choose whether your architecture diagram is used for:
  - facilitating a discussion about the system design
  - documenting (precisely) an architecture that has to be designed and implemented –> UML (in the context of this course)

Architectural design decisions

- Architectural design is a creative process so the process differs depending on the type of system.
- However, a number of common decisions span all design processes.
- Important Questions
  - Is there a generic application architecture that can be used?
  - How will the system be distributed?
  - What architectural styles are appropriate?
  - What approach will be used to structure the system?
  - How will the system be decomposed into modules?
  - What control strategy should be used?
  - How will the architectural design be evaluated?
  - How should the architecture be documented?

Architecture reuse

- Systems in the same domain often have similar architectures that reflect domain concepts.
- Application product lines are built around a core architecture with variants that satisfy particular customer requirements.
  - architecture patterns will be discussed in a minute…
- Application architectures in general
  - is part of this course described in chapter 6 of [SE10] and [OOSE];
  - product lines will not be covered in this course.
- If you are interested in software architecture as a topic, there is a dedicated course in the master program.

Architectural views

- It is impossible to represent all relevant information about the architecture of a system in one single architectural model
  - each model only shows one view or perspective.

- Krutchen (1995) 4+1 views
  1. Logical view – concerned with the functionality that the system provides to end-users; shows key abstractions in the system as objects or object classes.
     - E.g.: class diagram, communication diagram, sequence diagram
  2. Process view – shows how, at run-time, the system is composed of interacting processes.
     - E.g.: activity diagram
  3. Development view – shows how the software is decomposed for development.
     - E.g.: component package diagram
  4. Physical view – shows the system hardware and how software components are distributed across the processors in the system.
     - E.g.: deployment diagram
  - +1 – Use Case view

- (3+1 framework)
  - Christensen, H., Aino Corry, and K. Hansen. "An approach to software architecture description using UML." Computer Science Department, University of Aarhus (2004).

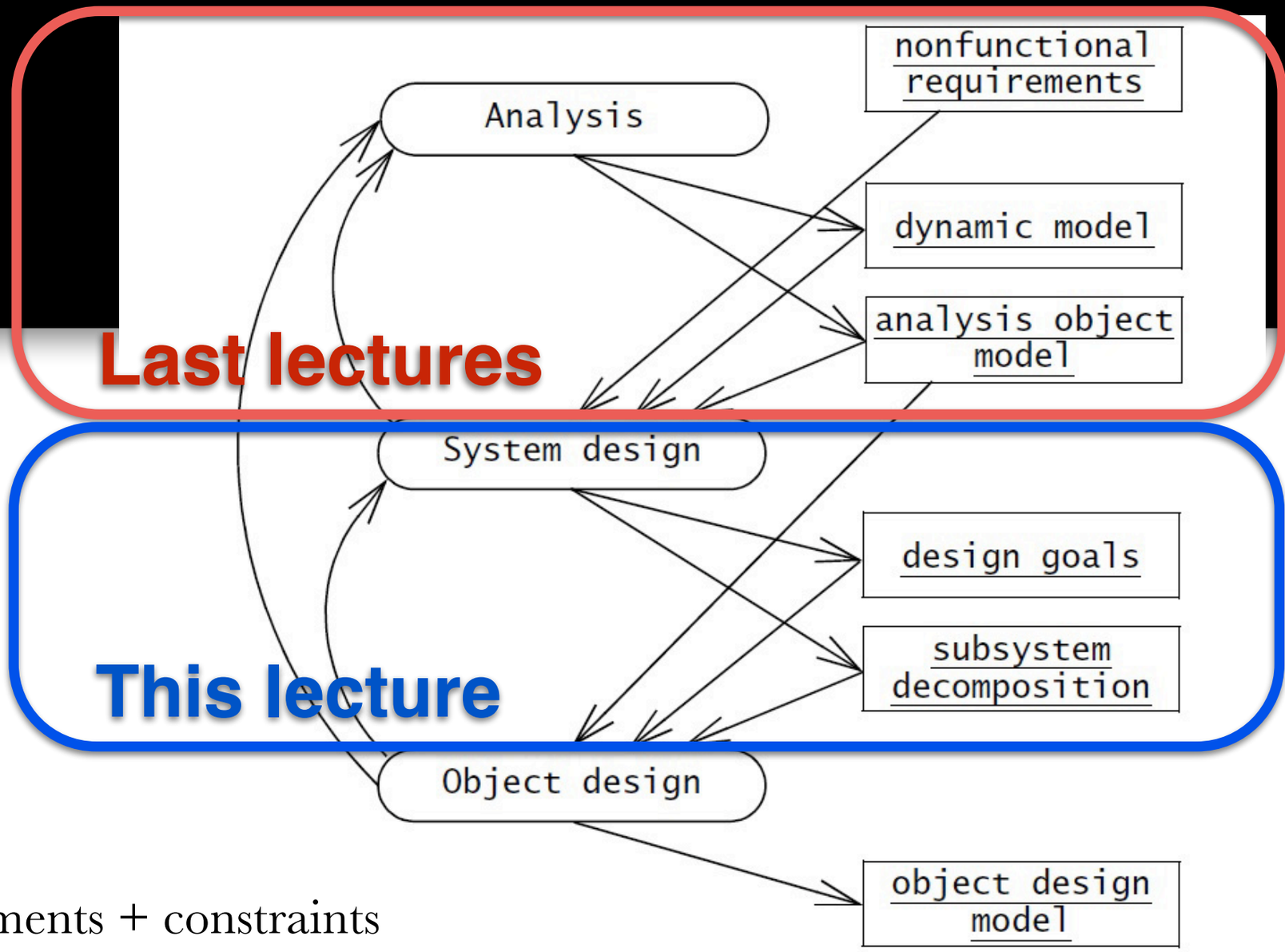
Object oriented design

Analysis versus design

- Analysis
  - focus on the application domain
- Design
  - focus on the solution domain

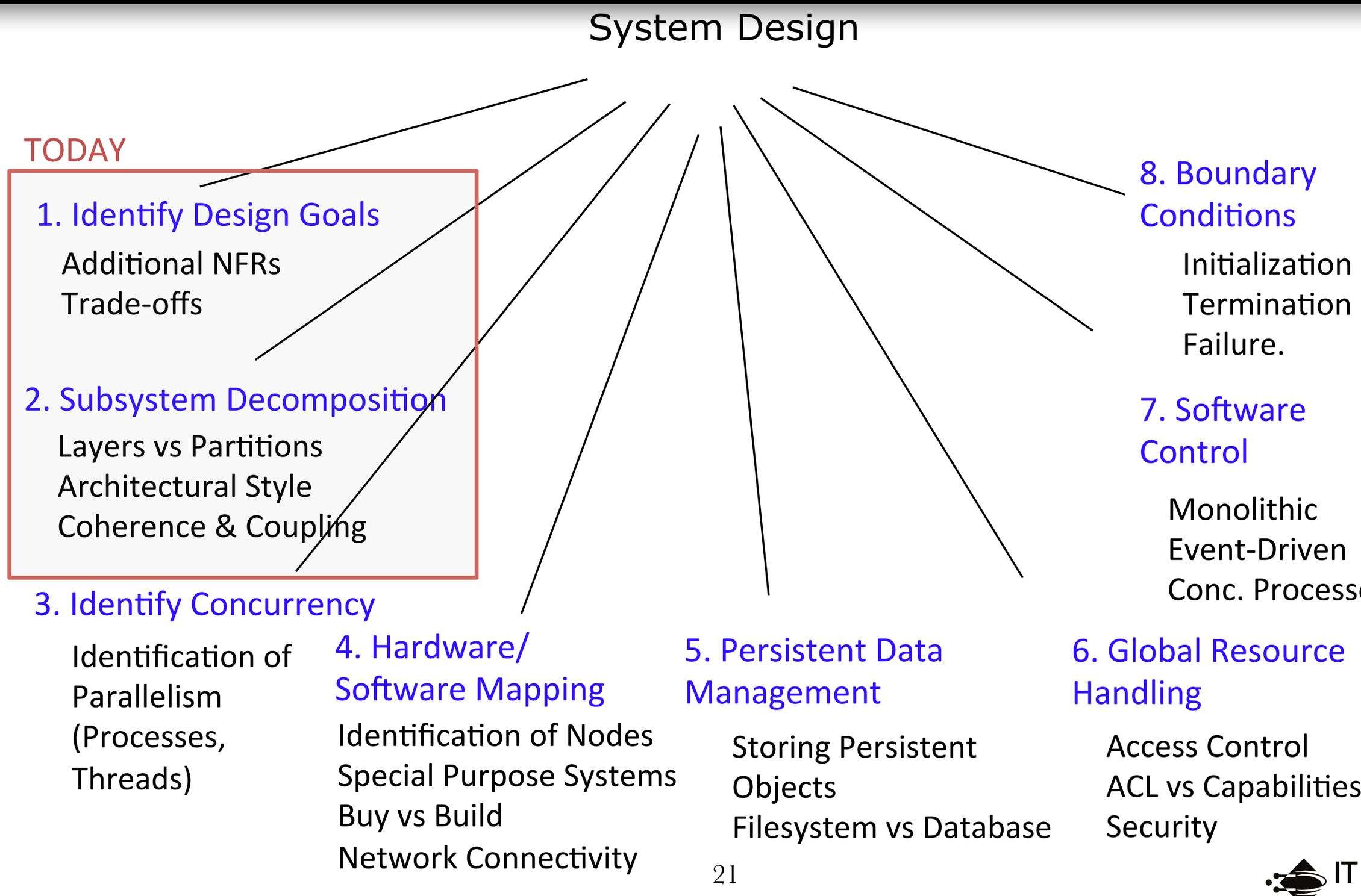
- Design is
  - transforming an analysis model into a system design model
  - definition/identification of design goals
  - decompose into sub-systems
  - selecting the right design/architectural strategies on things like
    - hardware/software; persistence; global flow control; access policies; handling boundary conditions; …

From analysis to design



- OOA
  - non-functional requirements + constraints
  - use case model
  - static model (class/object diagrams)
  - dynamic model (sequence/communication/state/activity diagrams)
- OOD
  - design goals (from non-functional reqs)
  - software architecture (based on styles/patterns)
  - boundary use cases (refinement of OOA model)

System design: eight concerns



Design goals

- Identifying design goals
  - first step in OOD
  - identify software qualities
  - inferred from non-functional requirements (and the client/user)

- Typical design goals
  - performance
  - dependability
  - cost
  - maintenance
  - end-user criteria

Architectural concerns and system characteristics

- Performance
  - Localise critical operations and minimise communications. Use large rather than fine-grain components.

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

- Safety
  - Localise 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.

- Others
  - Robustness, distributability, configurability, …

Architectural concerns and system characteristics

- Performance
  - Localise critical operations and minimise communications. Use large rather than fine-grain components.

- Security
  - Use a layered a…vers.

- Safety
  - Localise safety …tems.

- Availability
  - Include redun…lerance.

- Maintainability
  - Use fine-grain, replaceable components.

- Others
  - Robustness, distributability, configurability, …

In general – the non-functional requirements depend on the system architecture; the way in which these components are organised and communicate (Bosch, 2000).

Performance & dependability

Table 6-2 Performance criteria.

Design criterion	Definition
Response time	How soon is a user request acknowledged after the request has been issued?
Throughput	How many tasks can the system accomplish in a fixed period of time?
Memory	

Table 6-3 Dependability criteria.

Design criterion	Definition
Robustness	Ability to survive invalid user input
Reliability	Difference between specified and observed behavior
Availability	Percentage of time that system can be used to accomplish normal tasks
Fault tolerance	Ability to operate under erroneous conditions
Security	Ability to withstand malicious attacks
Safety	Ability to avoid endangering human lives, even in the presence of errors and failures

Cost & maintenance

Table 6-4 Cost criteria.

Design criterion	Definition
Development cost	Cost of developing the initial system
Deployment cost	Cost of installing the system and training the users
Upgrade cost	Cost of translating data from the previous system. This criteria results
Maintenance cost	
Administration cost	

Table 6-5 Maintenance criteria.

Design criterion	Definition
Extensibility	How easy is it to add functionality or new classes to the system?
Modifiability	How easy is it to change the functionality of the system?
Adaptability	How easy is it to port the system to different application domains?
Portability	How easy is it to port the system to different platforms?
Readability	How easy is it to understand the system from reading the code?
Traceability of requirements	How easy is it to map the code to specific requirements?

End-user

Table 6-6 End user criteria.

Design criterion	Definition
Utility	How well does the system support the work of the user?
Usability	How easy is it for the user to use the system?

Design goal trade-offs

Table 6-7 Examples of design goal trade-offs.

Trade-off	Rationale
Space vs. speed	If the software does not meet response time or throughput requirements, more memory can be expended to speed up the software (e.g., caching, more redundancy). If the software does not meet memory space constraints, data can be compressed at the cost of speed.
Delivery time vs. functionality	If development runs behind schedule, a project manager can deliver less functionality than specified on time, or deliver the full functionality at a later time. Contract software usually puts more emphasis on functionality, whereas off-the-shelf software projects put more emphasis on delivery date.
Delivery time vs. quality	If testing runs behind schedule, a project manager can deliver the software on time with known bugs (and possibly provide a later patch to fix any serious bugs), or deliver the software later with fewer bugs.
Delivery time vs. staffing	If development runs behind schedule, a project manager can add resources to the project to increase productivity. In most cases, this option is only available early in the project: adding resources usually decreases productivity while new personnel are trained or brought up to date. Note that adding resources will also raise the cost of development.

System design concepts

- Subsystems and Classes
- Services and Subsystems Interfaces
- Coupling and Cohesion
- Layers and Partitions
- Architectural Styles

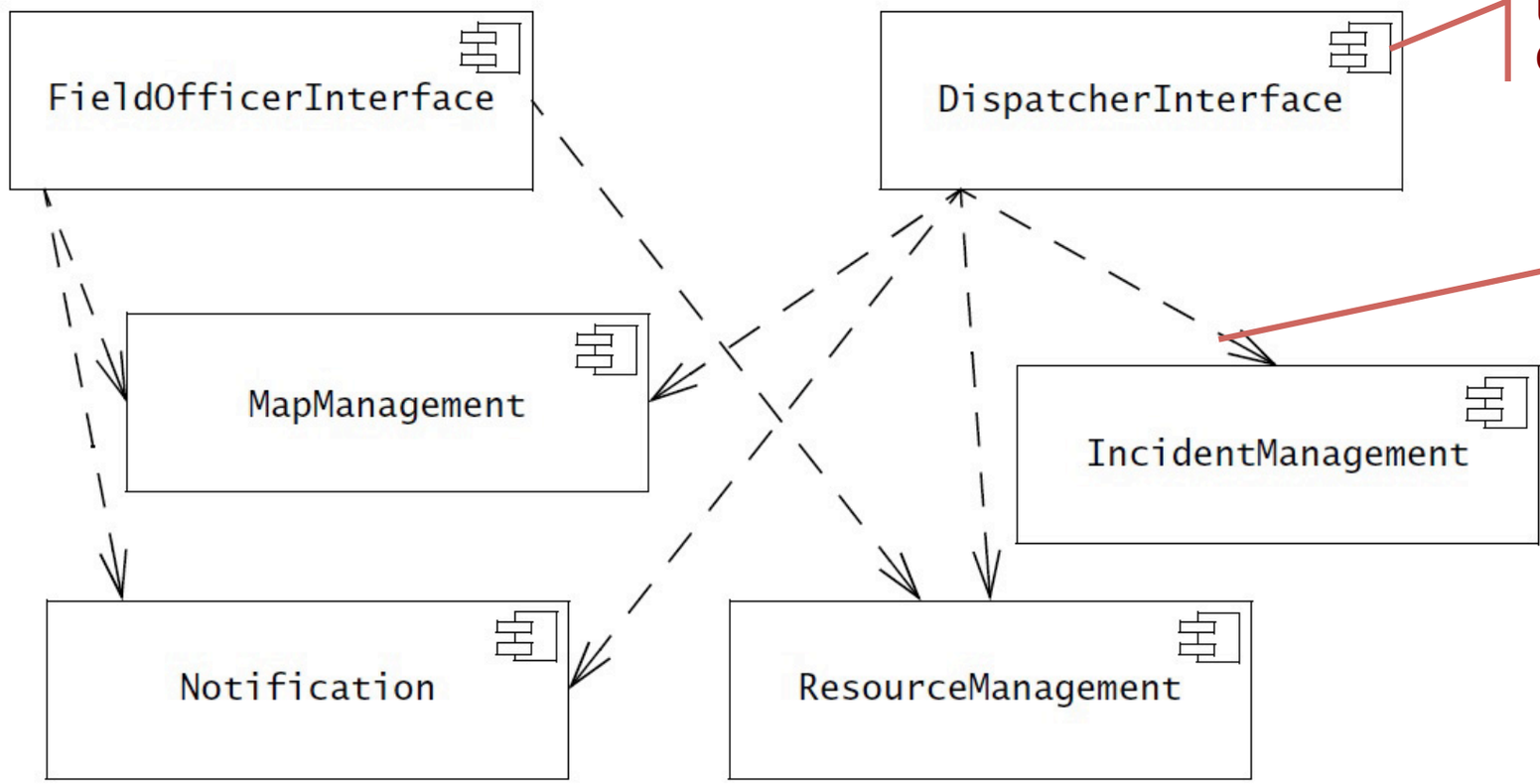
Identifying subsystems

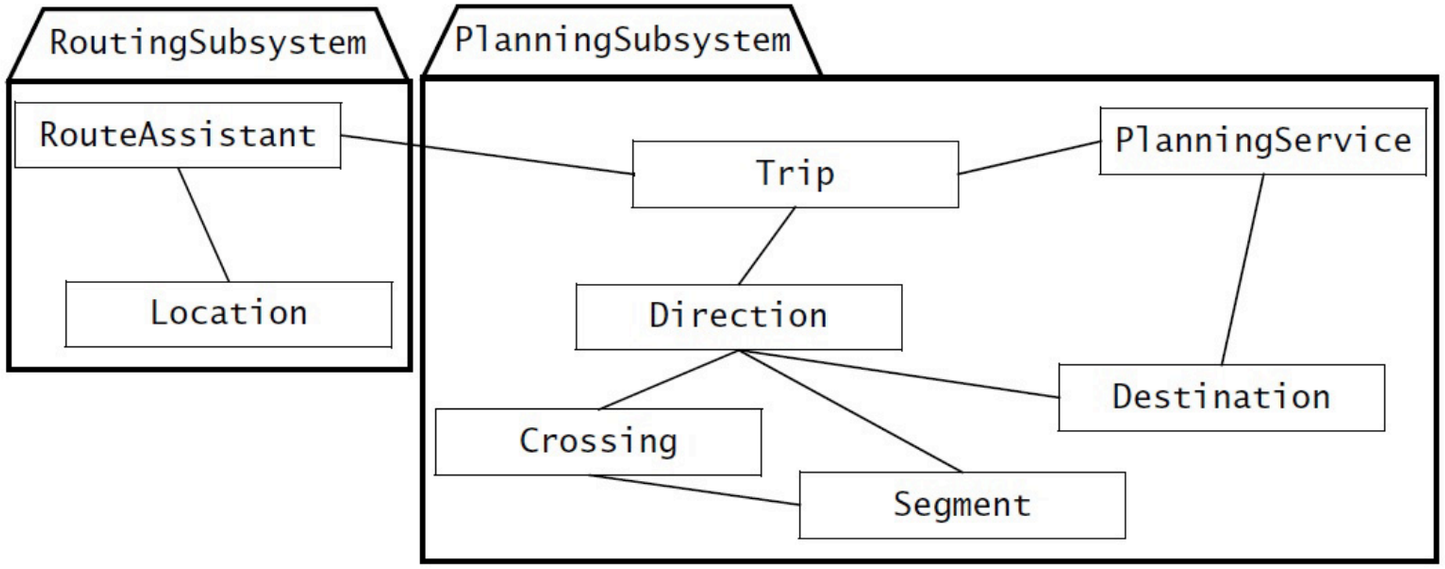
- Identifying subsystems
  - initially derived from the functional requirements
  - keep functionally related objects together (cohesion)

- Heuristics
  - assign objects identified in one use case into the same subsystem
  - create a dedicated subsystem for objects used for moving data among subsystems
  - minimise the number of associations crossing subsystem boundaries (coupling)
  - all objects in the same subsystem should be functionally related (SOLID)

- Eventually …
  - Definition: a subsystem is a replaceable part of the system with well-defined interfaces that encapsulates the state and behaviour of its contained classes

Examples



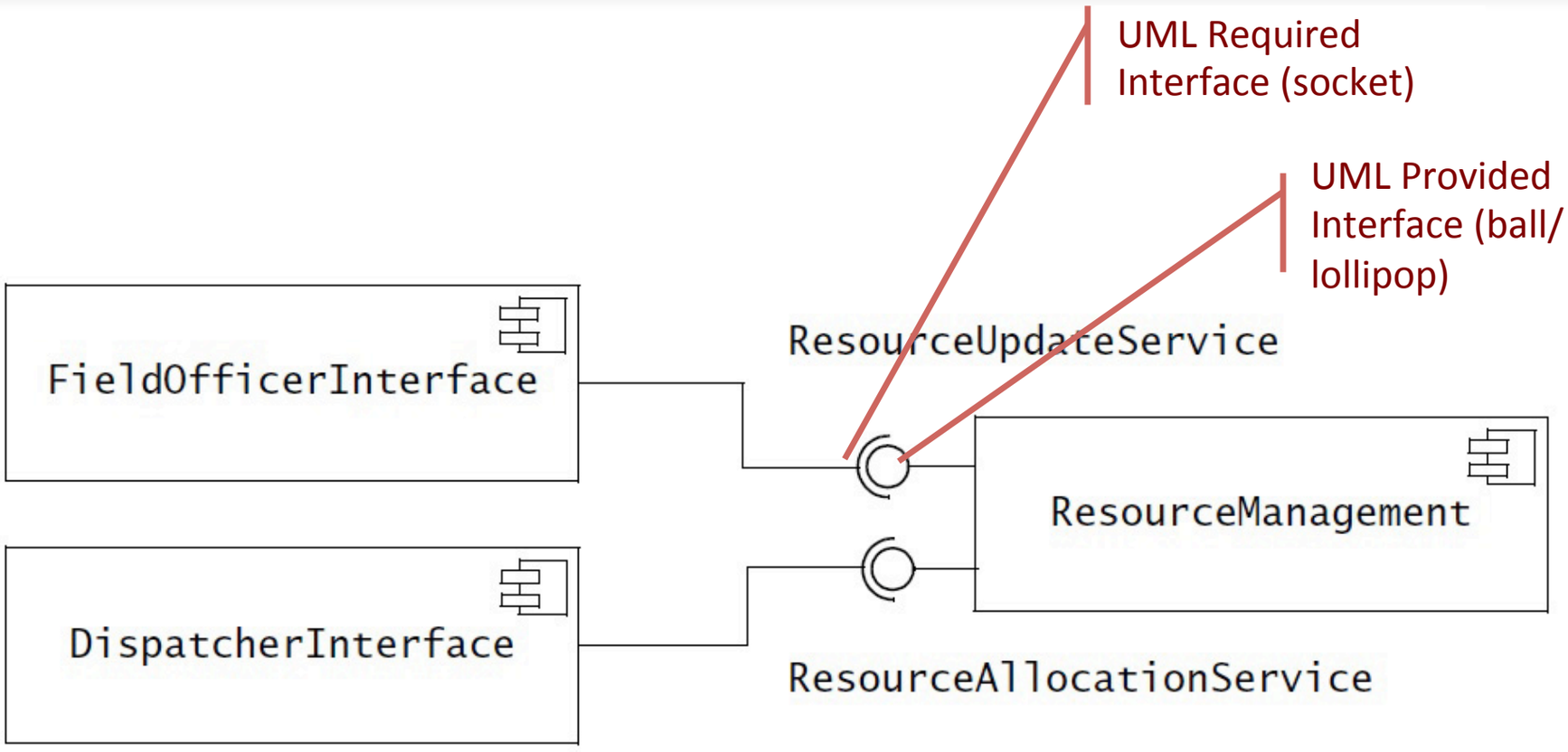


PlanningSubsystem The PlanningSubsystem is responsible for constructing a Trip connecting a sequence of Destinations. The PlanningSubsystem is also responsible for responding to replan requests from RoutingSubsystem.

RoutingSubsystem The RoutingSubsystem is responsible for downloading a Trip from the PlanningService and executing it by giving Directions to the driver based on its Location.

Figure 6-29 Initial subsystem decomposition for MyTrip (UML class diagram).

Services & subsystem interfaces



- Service
  - a set of related operations that share a common purpose
  - interface of a subsystem
    - main focus during design – not implementation
  - application programming interface (API)

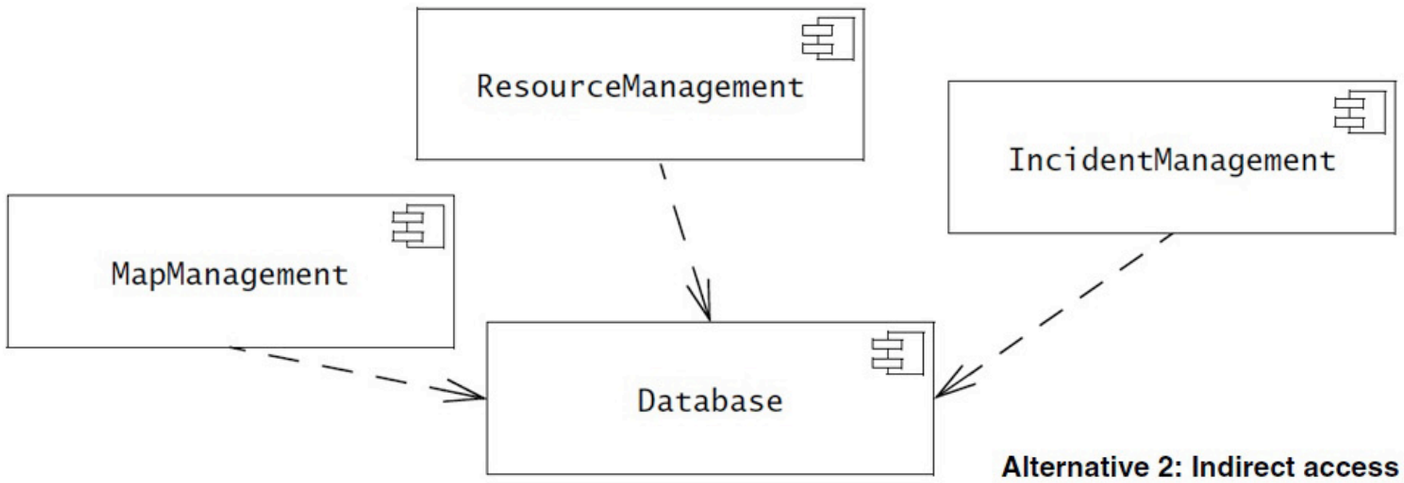
Coupling & cohesion

- Coupling
  - the number of of dependencies between subsystems
  - “loosely” or “strongly” coupling
  - what is best?

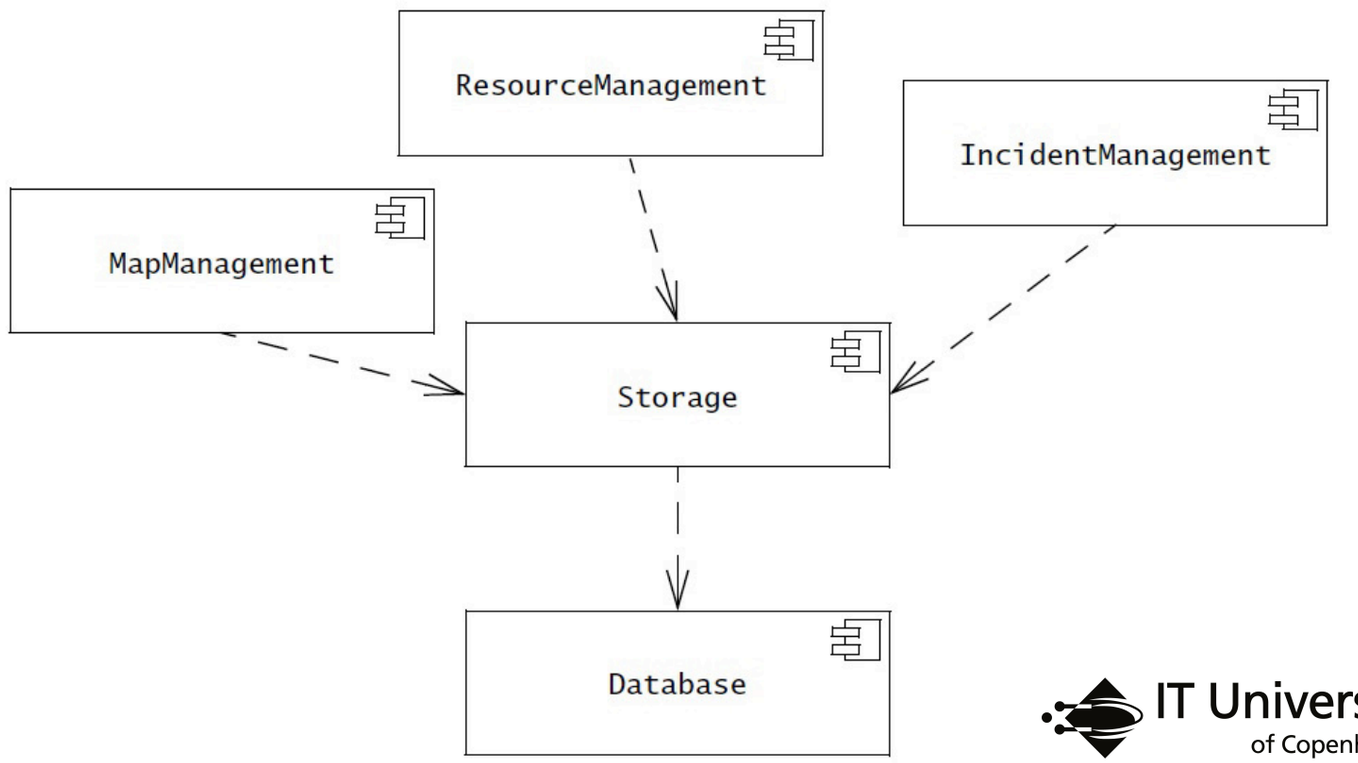
- Cohesion
  - the number of dependencies within a subsystem
  - “high” or “low” cohesion
  - what is best?

Coupling

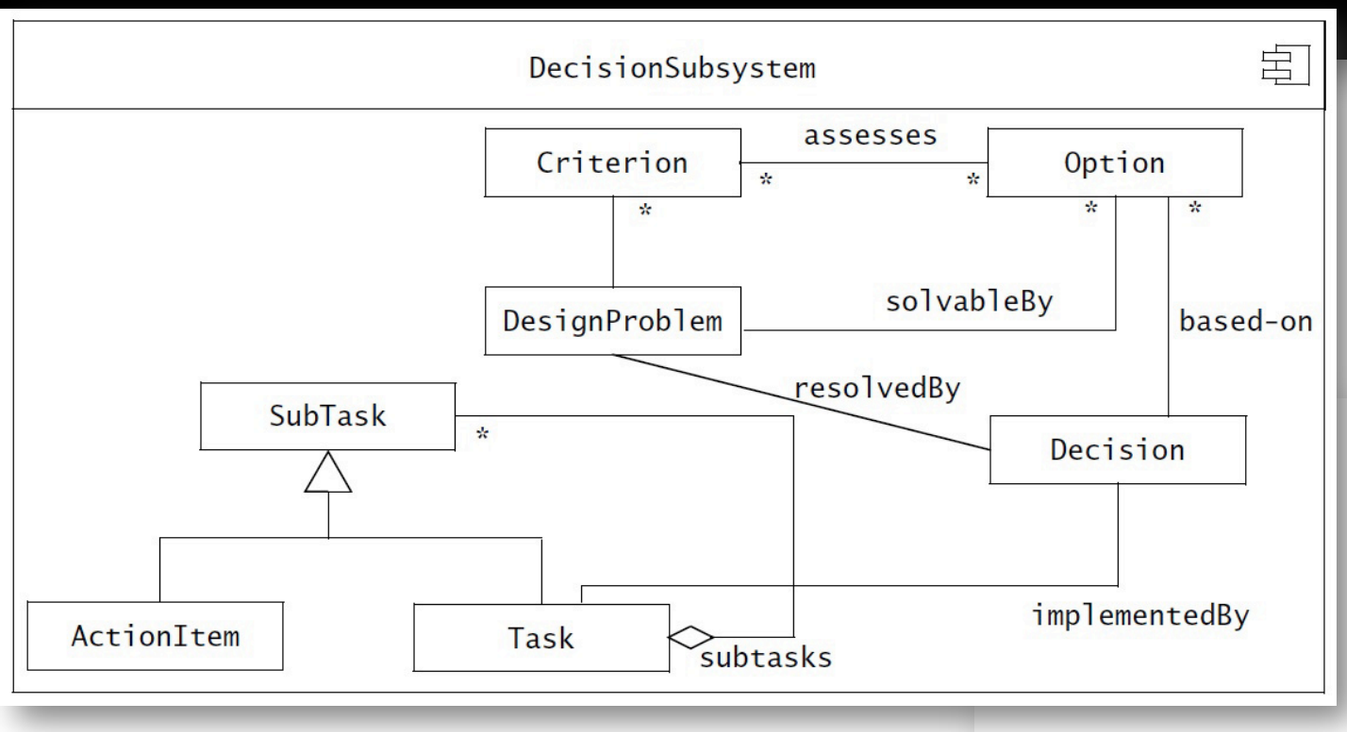
Alternative 1: Direct access to the Database subsystem

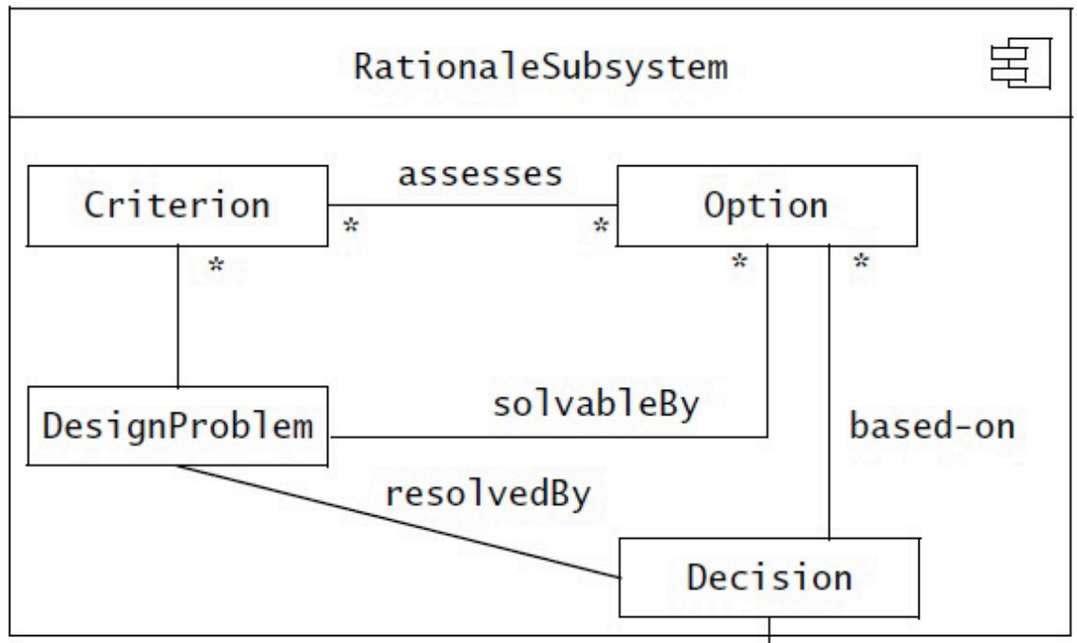


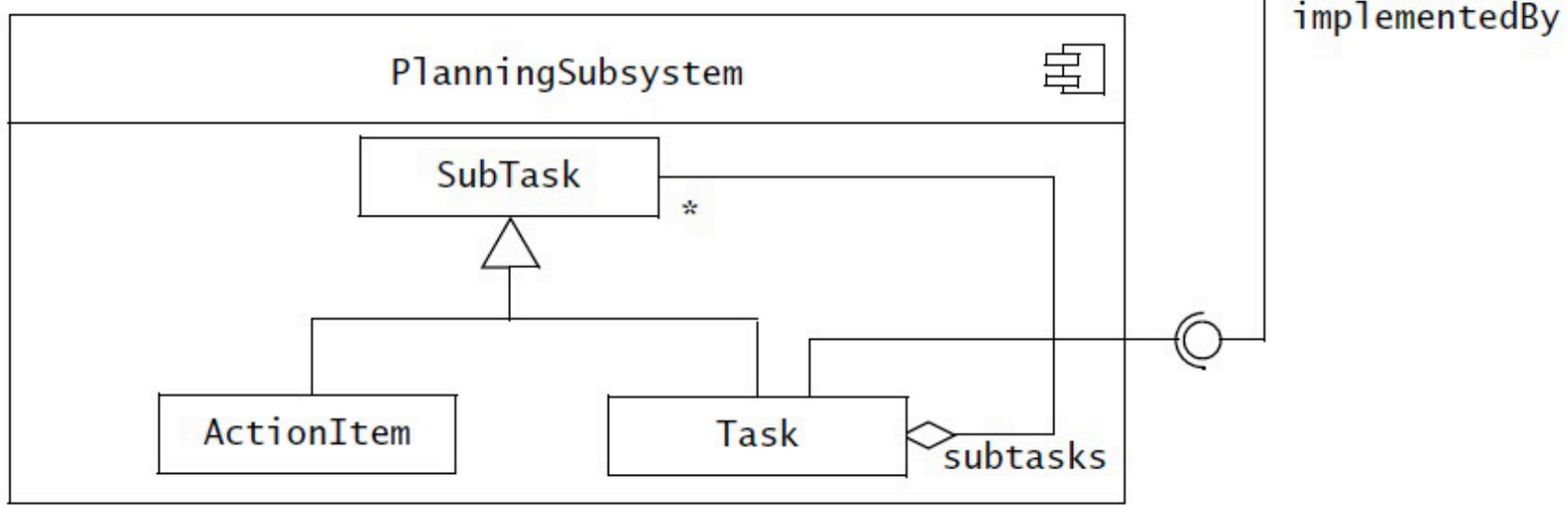
Alternative 2: Indirect access to the Database through a Storage subsystem



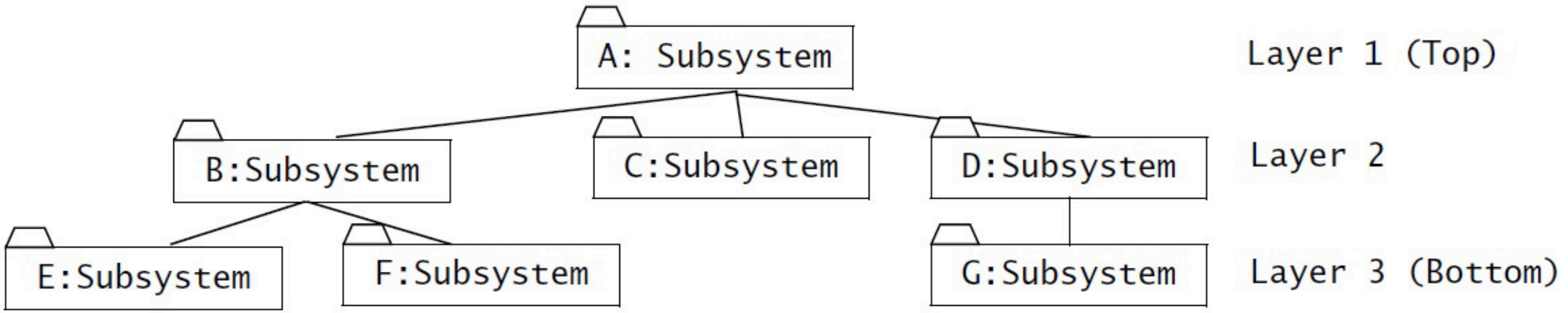
Cohesion


DecisionSubsystem
Criterion
assesses
Option
DesignProblem
solvableBy
based-on
resolvedBy
SubTask
Decision
ActionItem
Task
subtasks
implementedBy


RationaleSubsystem
Criterion
assesses
Option
DesignProblem
solvableBy
based-on
resolvedBy
Decision
implementedBy


PlanningSubsystem
SubTask
ActionItem
Task
subtasks

Layers & partitions



- Hierarchical decomposition into layers
- Layer
  - grouping of subsystems providing related services
  - closed architecture = each layer can only access the layer below
  - open architecture = each layer can access all layers below

Architectural styles

- … in a moment …