Analysis, Design, and Software Architecture (BDSA)

Paolo Tell

System Design and Object Design

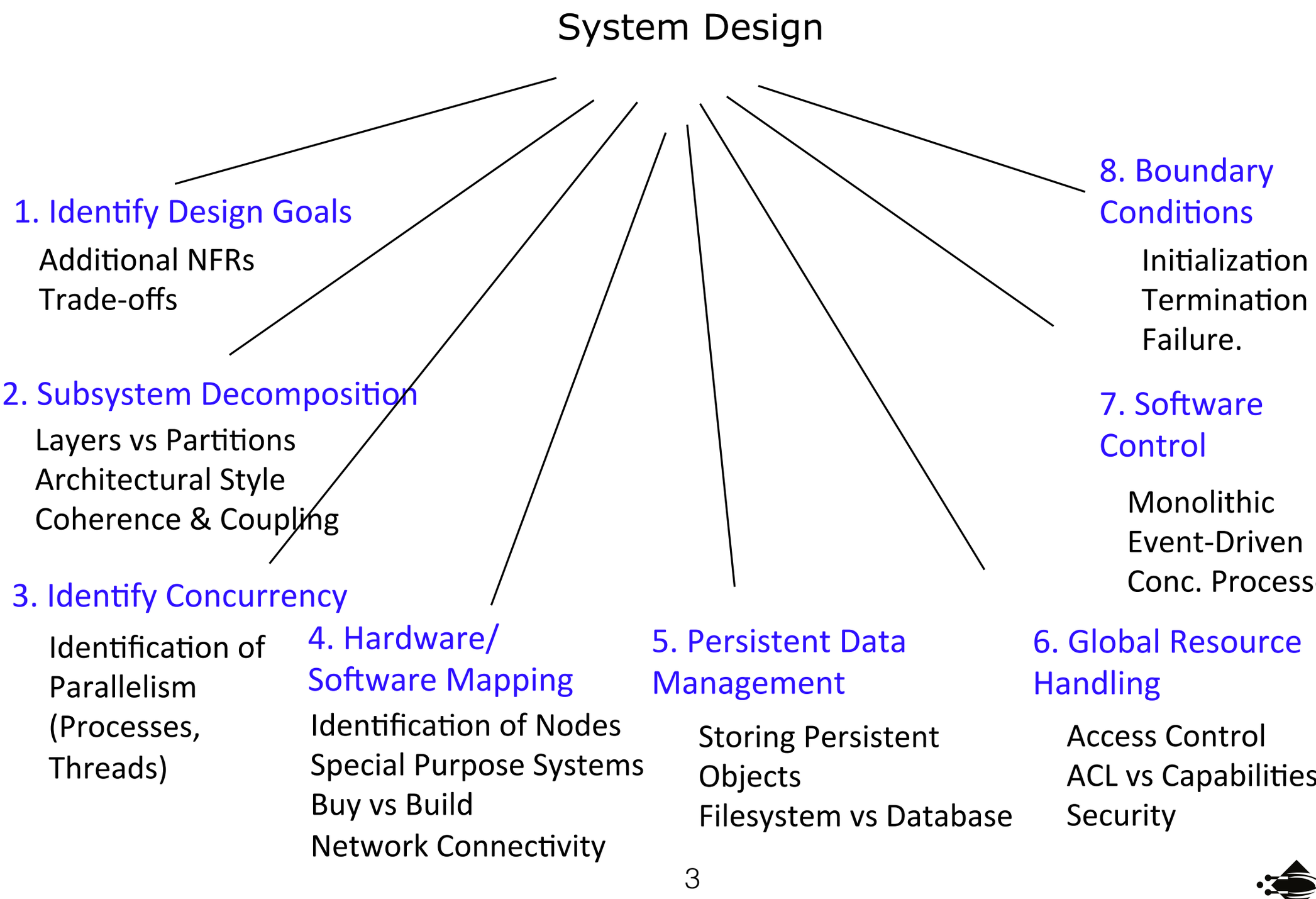
IT University
of Copenhagen

Outline

- Literature
  - [OOSE] ch. 7+8
  - [OOSE] ch. 9+10
  - [SE9] ch. 7

- Topics covered:
  - System Design
  - Object design—Specifying interfaces
  - Object design—Mapping models to code
  - Object design—Mapping OO to RDBMS

System Design Document (SDD)



System Design Document (SDD)

System Design Document

1. Introduction
   - 1.1 Purpose of the system
   - 1.2 Design goals
   - 1.3 Definitions, acronyms, and abbreviations
   - 1.4 References
   - 1.5 Overview
2. Current software architecture
3. Proposed software architecture
   - 3.1 Overview
   - 3.2 Subsystem decomposition
   - 3.3 Hardware/software mapping
   - 3.4 Persistent data management
   - 3.5 Access control and security
   - 3.6 Global software control
   - 3.7 Boundary conditions
4. Subsystem services

   Glossary

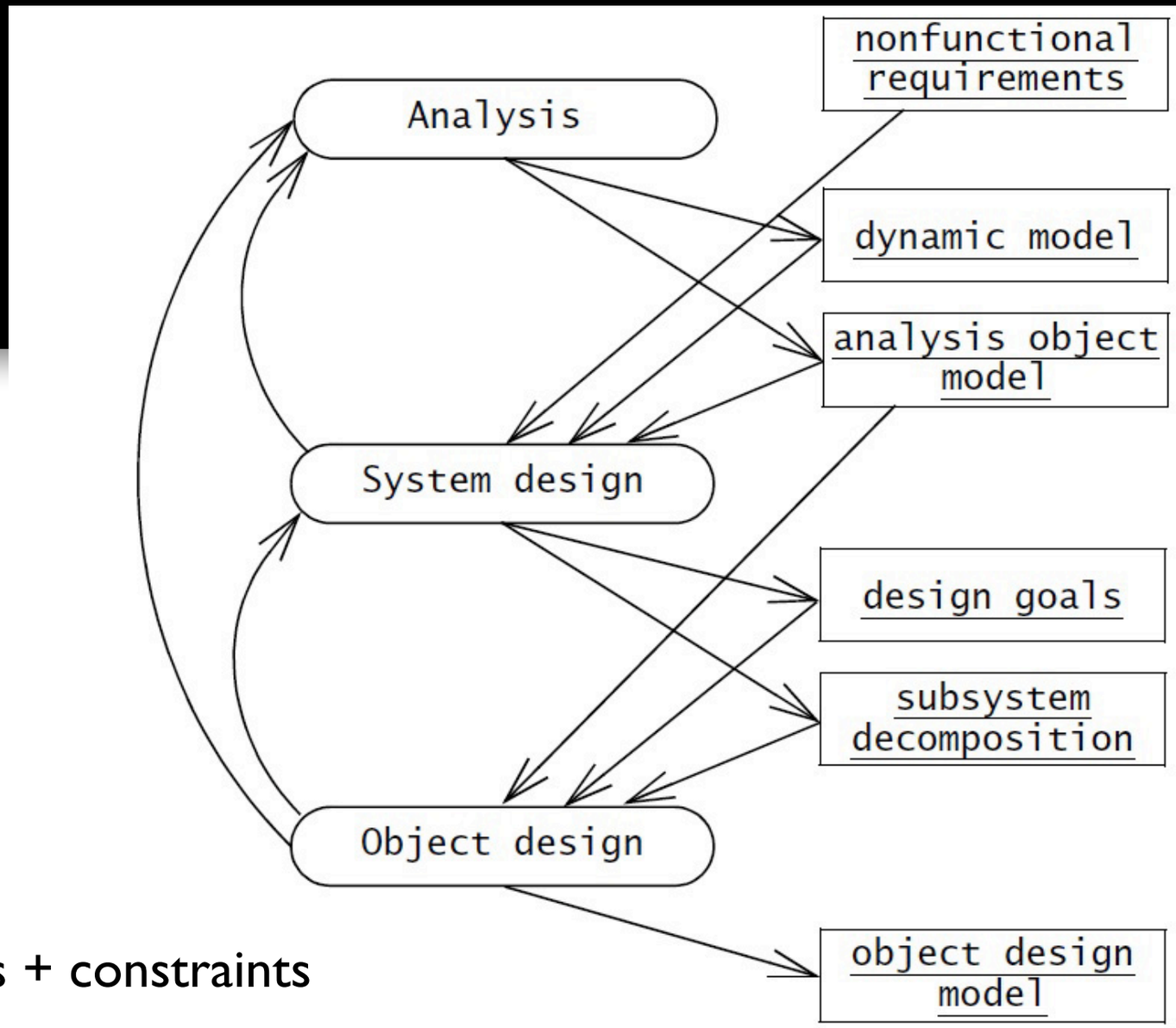
System design

Analysis versus design

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

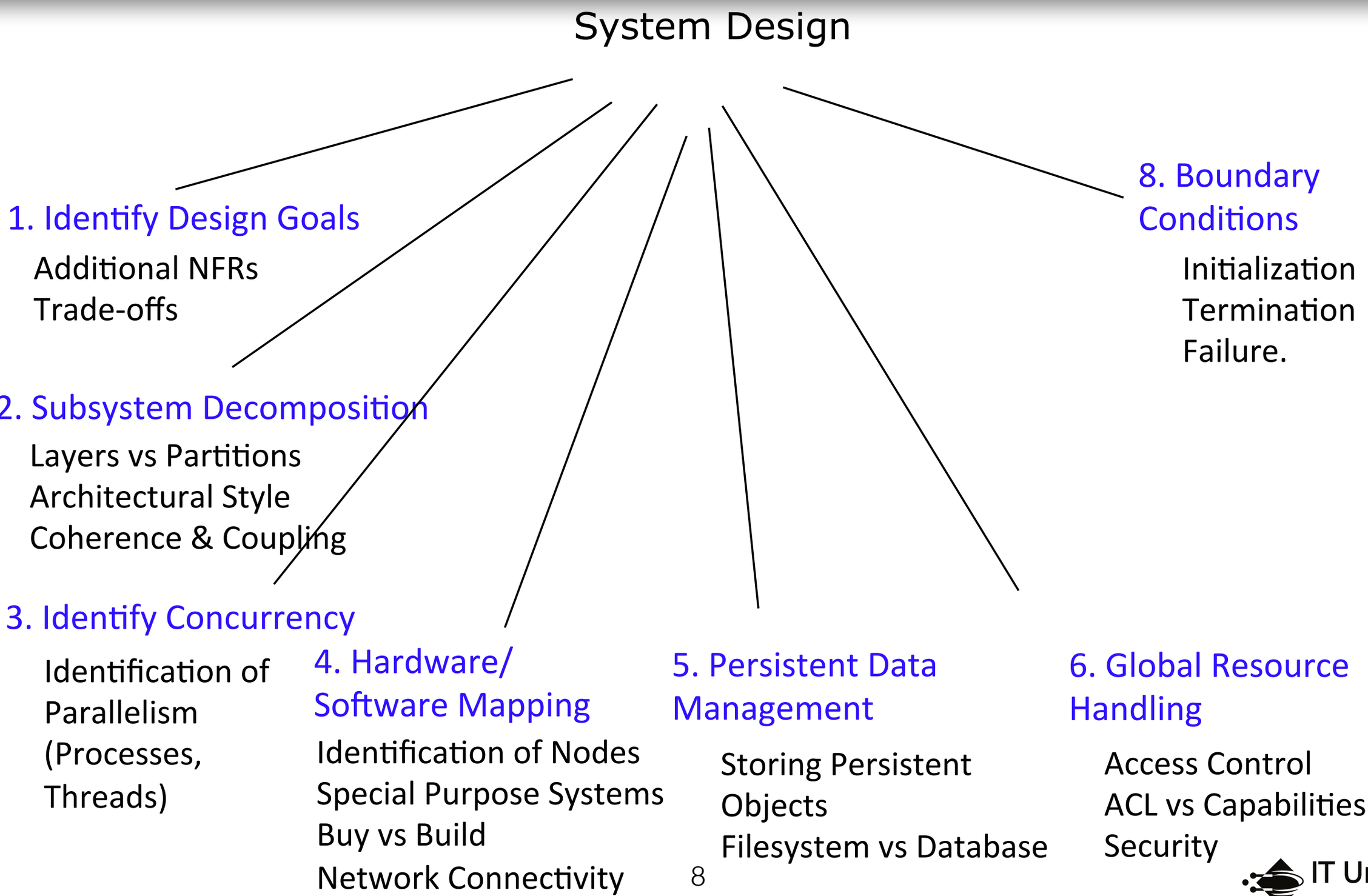
- Design is
  - transforming an analysis model into a system design model
  - definition/identification of design goals
  - decompose into sub-systems
  - selecting the right design/architectural strategies on things like
    - hardware/software; persistence; global flow control; access policies; handling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

From analysis to design

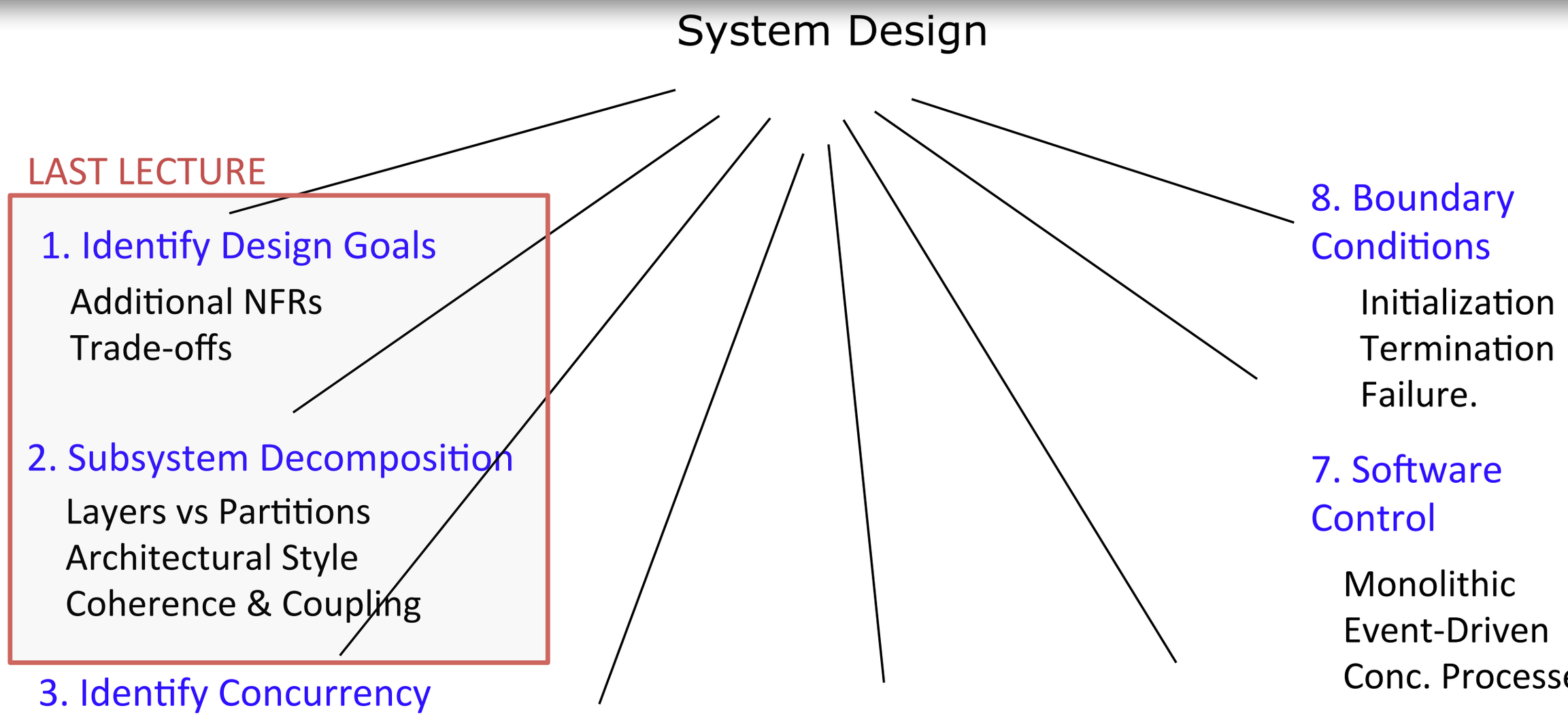


- OOA
  - non-functional requirements + constraints
  - use case model
  - static model (class/object diagrams)
  - dynamic model (sequence/communication/state/activity diagrams)
- OOD
  - design goals (from non-functional requirements)
  - software architecture (based on styles/patterns)
  - boundary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

System design

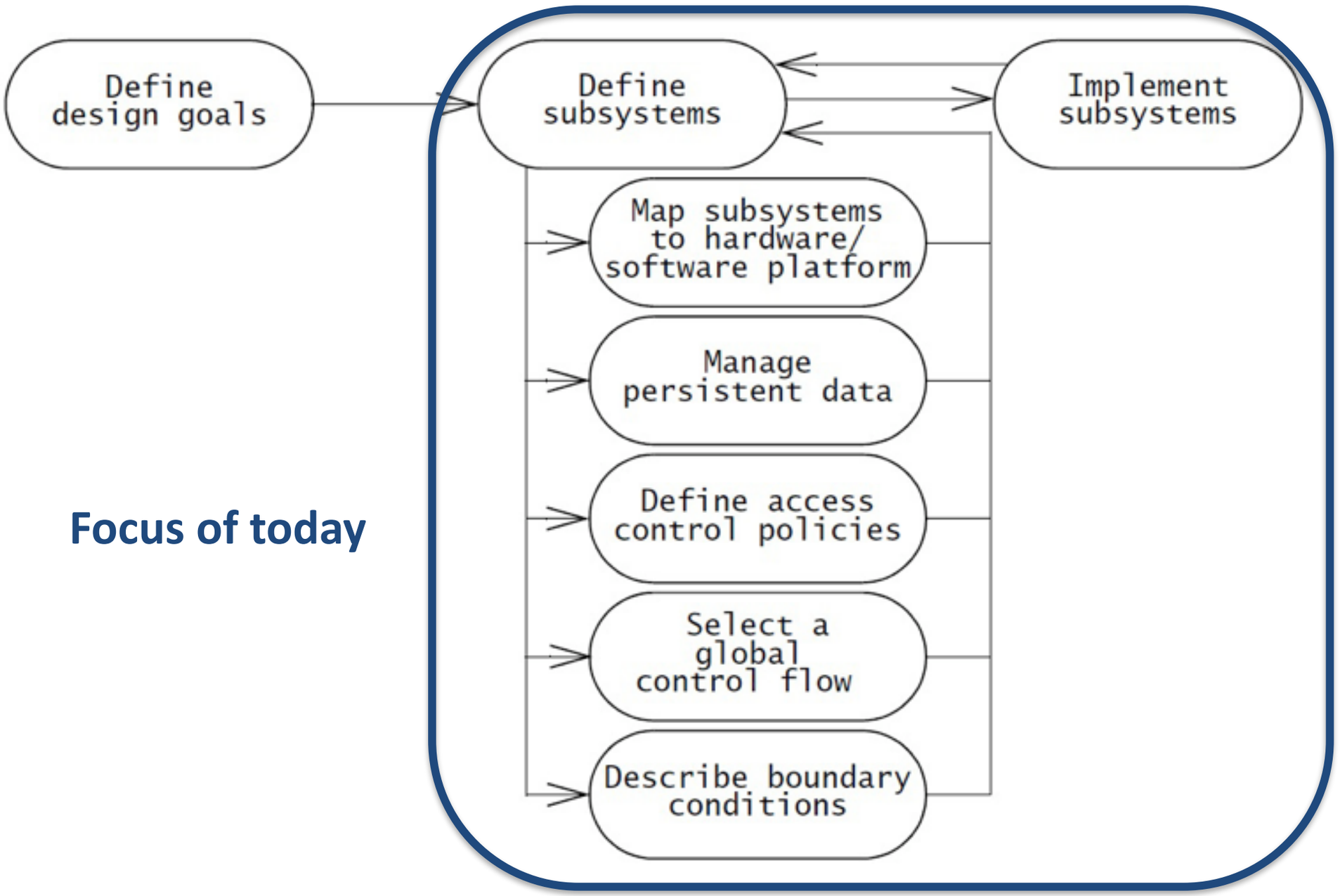


System design



System design

Focus of today



System design activities

1. Identifying Design Goals
2. Subsystem decomposition
3. Mapping subsystems to processors and components
4. Identifying and storing persistent data
5. Providing access control
6. Designing the global control flow
7. Identifying services
8. Identifying boundary conditions
9. Reviewing the system design model

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
  - 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.

- Others
  - Robustness, distributability, configurability, …

Architectural concerns and system characteristics

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

- Security
  - Use a layered a… …ayers.

- Safety
  - Localize safety … …stems.

- Availability
  - Include redunc… …olerance.

- 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 organized and communicate (Bosch, 2000).

Examples

- Performance
  - Response time, throughput, memory
- Dependability
  - Robustness, reliability, availability, fault tolerance, security, safety
- Cost
  - Development c., deployment c., upgrade c., maintenance c., administration c.
- Maintenance
  - Extensibility, modifiability, portability, readability, traceability of requirements
- End-user
  - Utility, usability

Design goal trade-offs

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

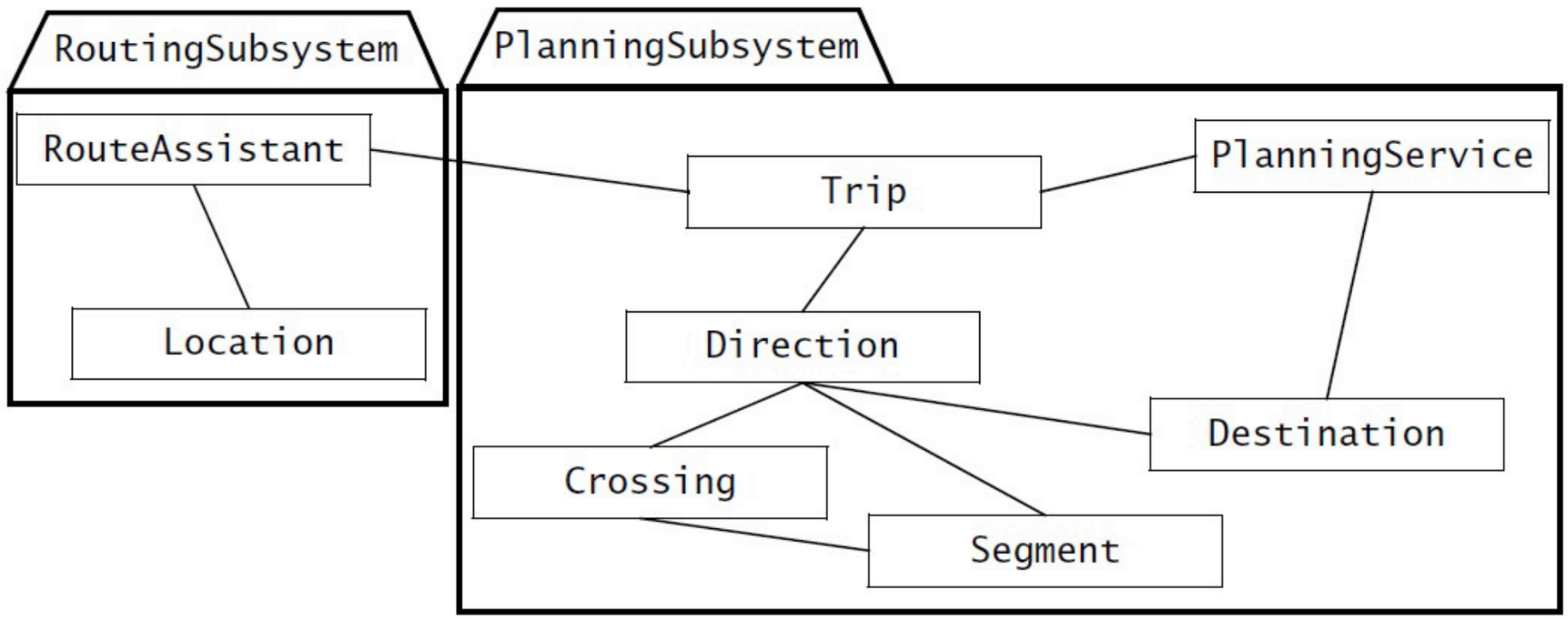
Trade-off	Rationale
Space vs. speed	If the software does not meet response time or throughput requirements, more memory can be expended to speed up the software (e.g., caching, more redundancy). If the software does not meet memory space constraints, data can be compressed at the cost of speed.
Delivery 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.

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
  - minimize the number of associations crossing subsystem boundaries
  - all objects in the same subsystem should be functionally related

Example: trip system



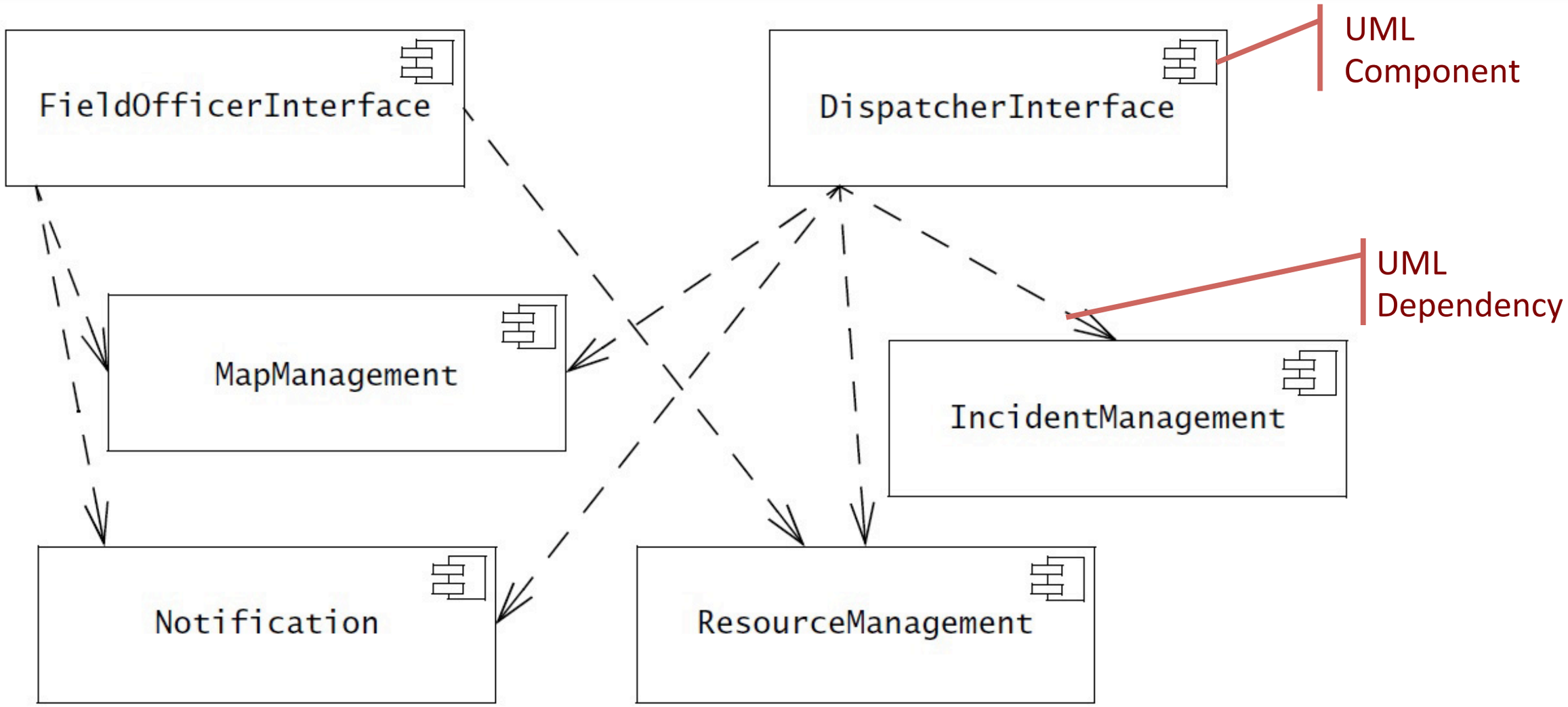
PlanningSubsystem	The PlanningSubsystem is responsible for constructing a Trip connecting a sequence of Destinations. The PlanningSubsystem is also responsible for responding to replan requests from RoutingSubsystem.
RoutingSubsystem	The RoutingSubsystem is responsible for downloading a Trip from the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

System design concepts

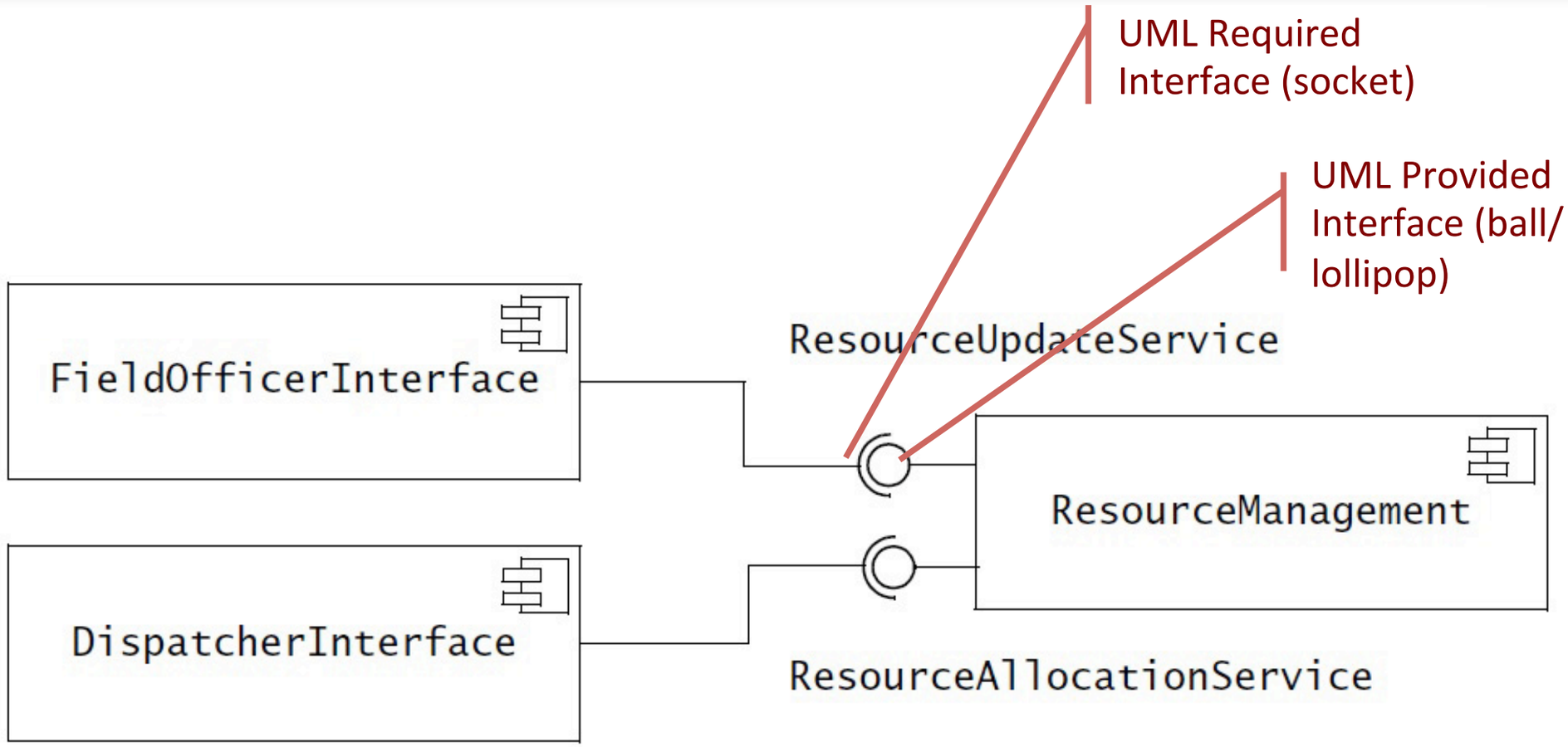
- Subsystems and Classes
- Services and Subsystems Interfaces
- Coupling and Cohesion
- Layers and Partitions
- Architectural Styles

Subsystems & classes



- Subsystem
  - a replaceable part of the system with well-defined interfaces that encapsulates the state and behavior of its contained classes

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)

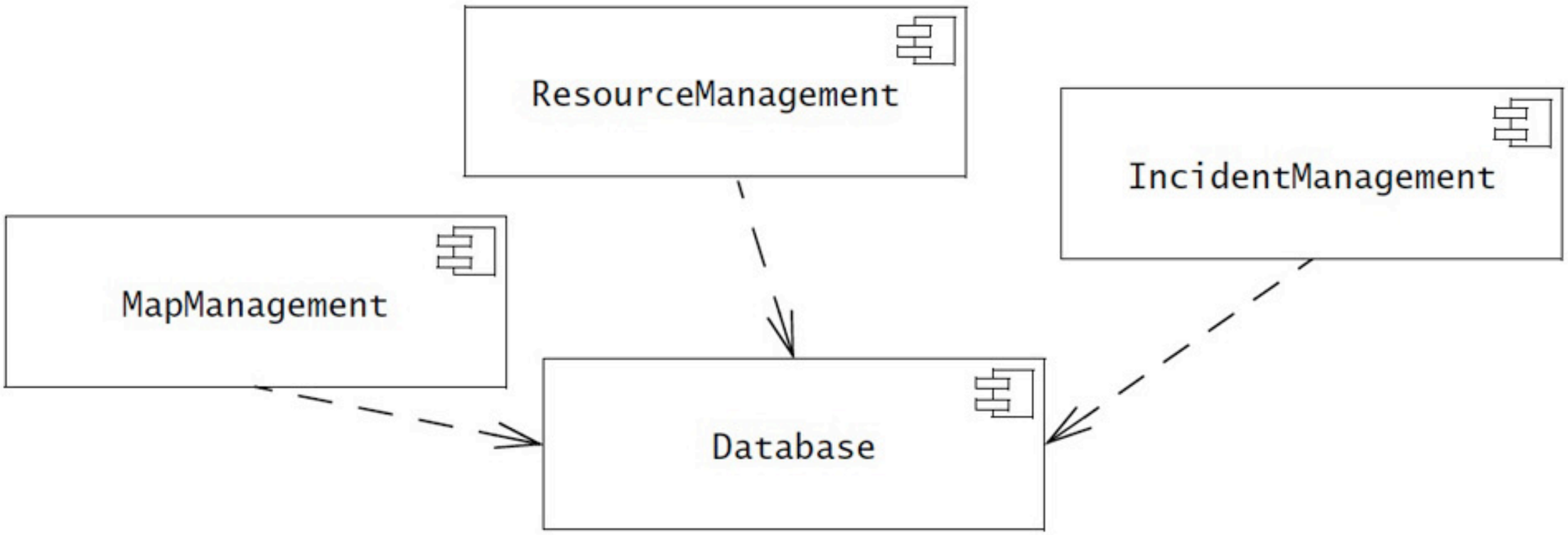
Coupling & cohesion

- 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?

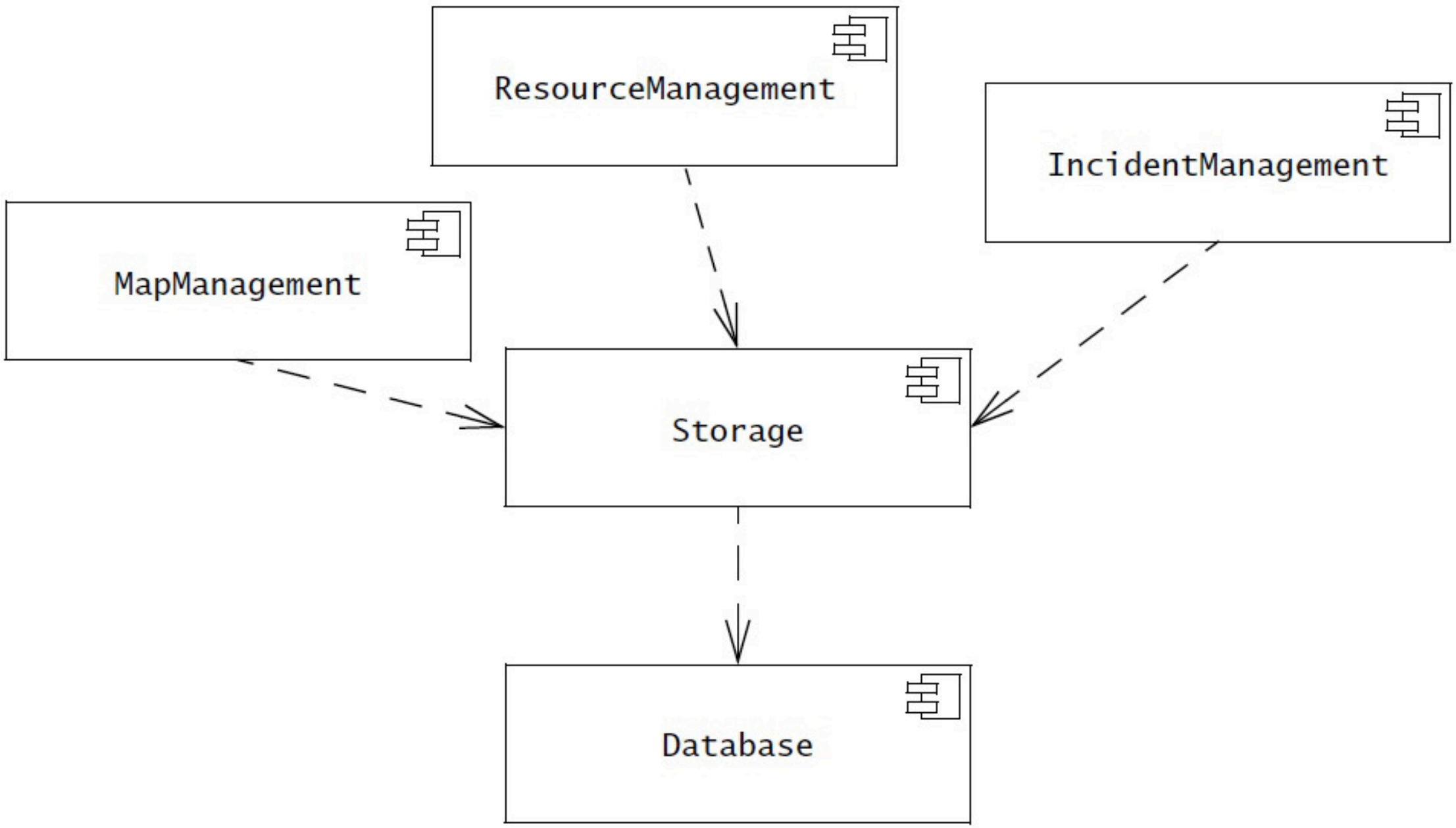
Coupling & cohesion

Alternative 1: Direct access to the Database subsystem

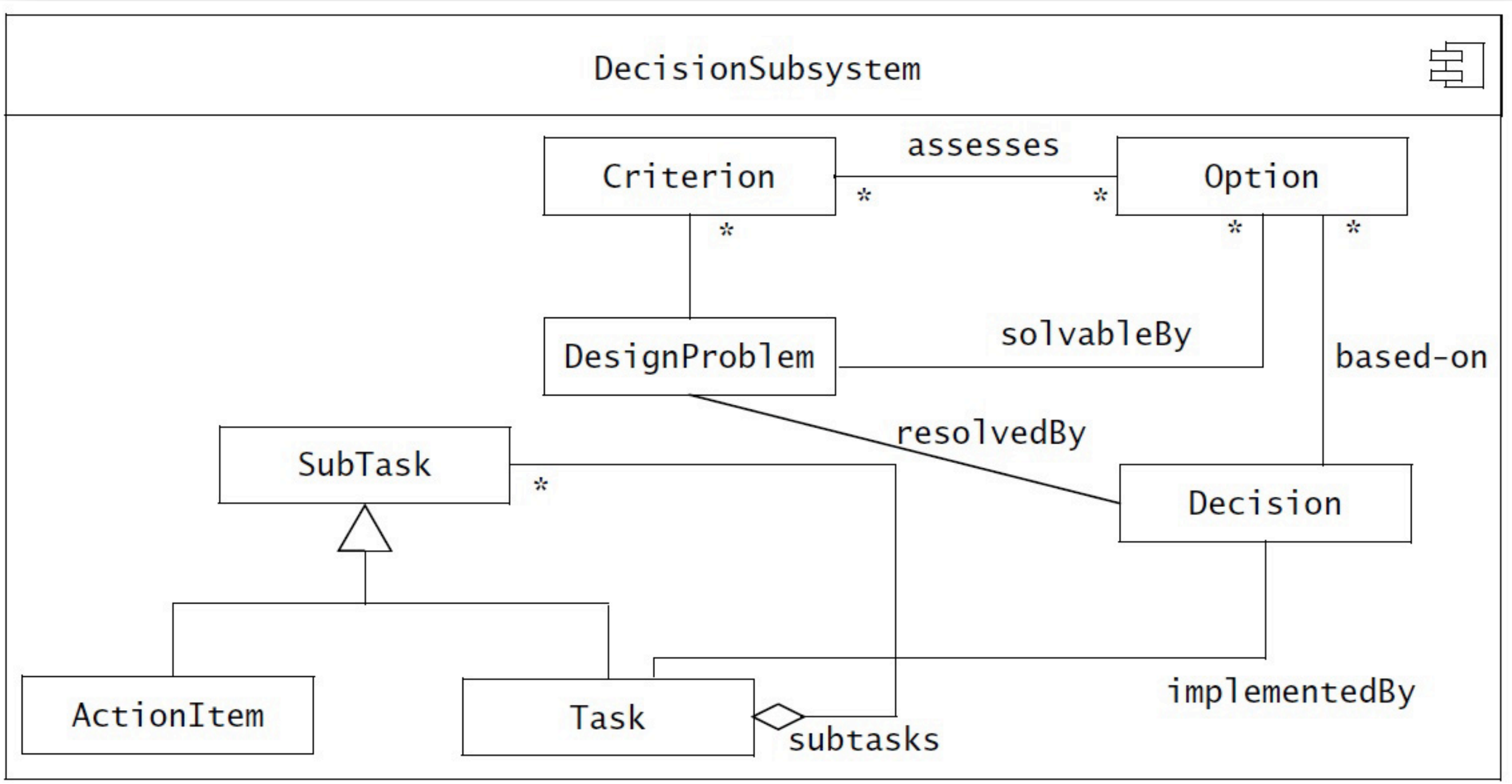


Coupling & cohesion

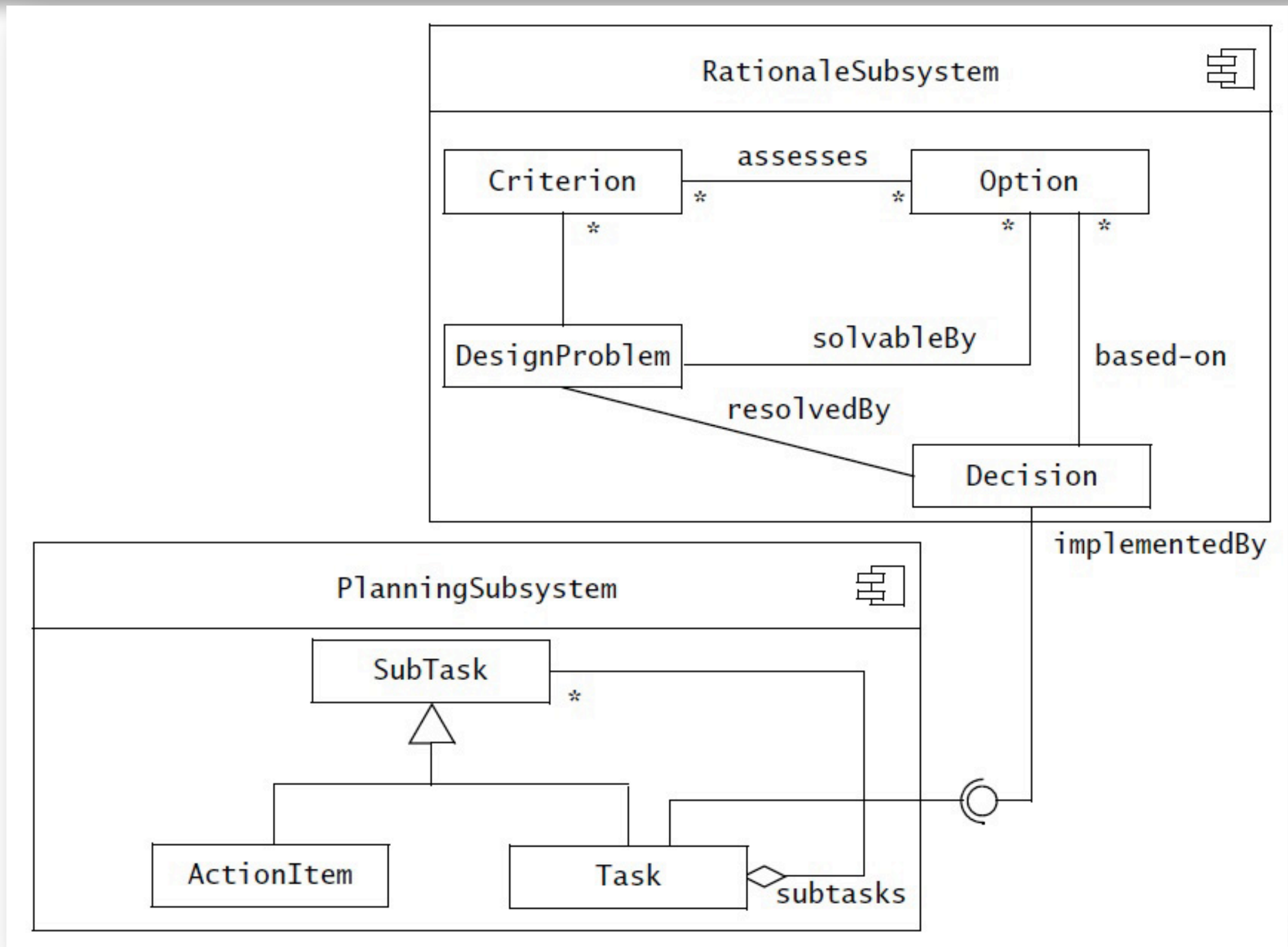
Alternative 2: Indirect access to the Database through a Storage subsystem



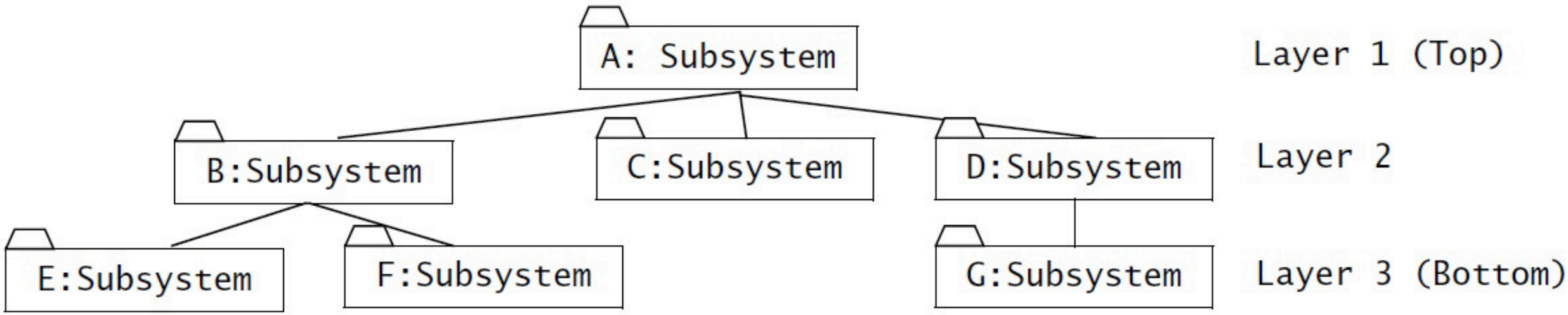
Coupling & cohesion



Coupling & cohesion



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

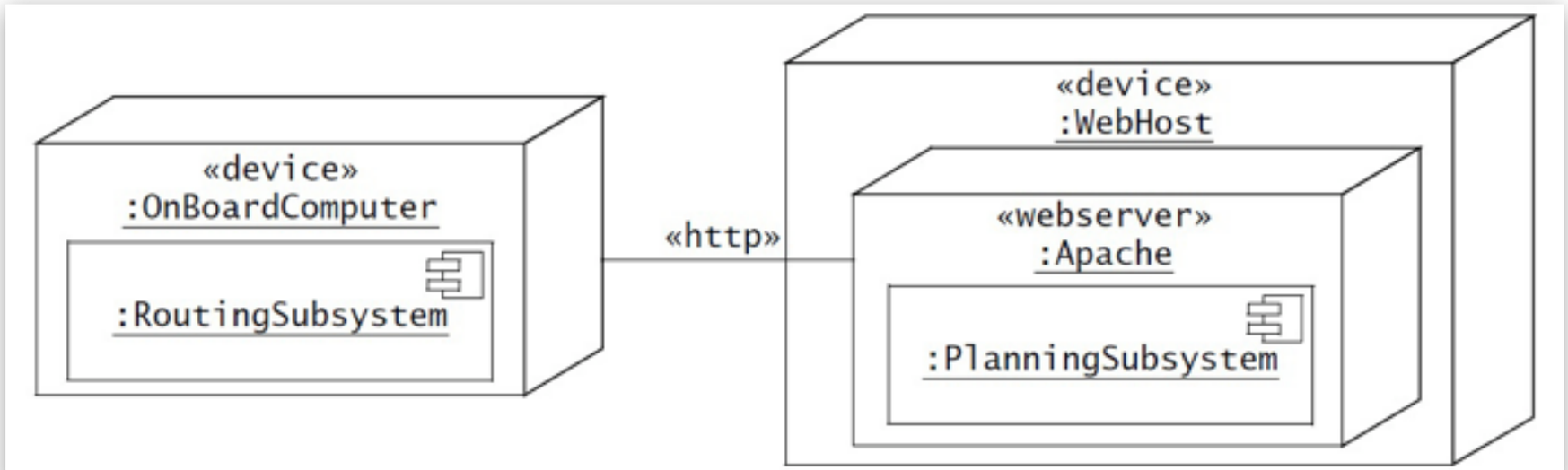
Architectural styles (patterns)

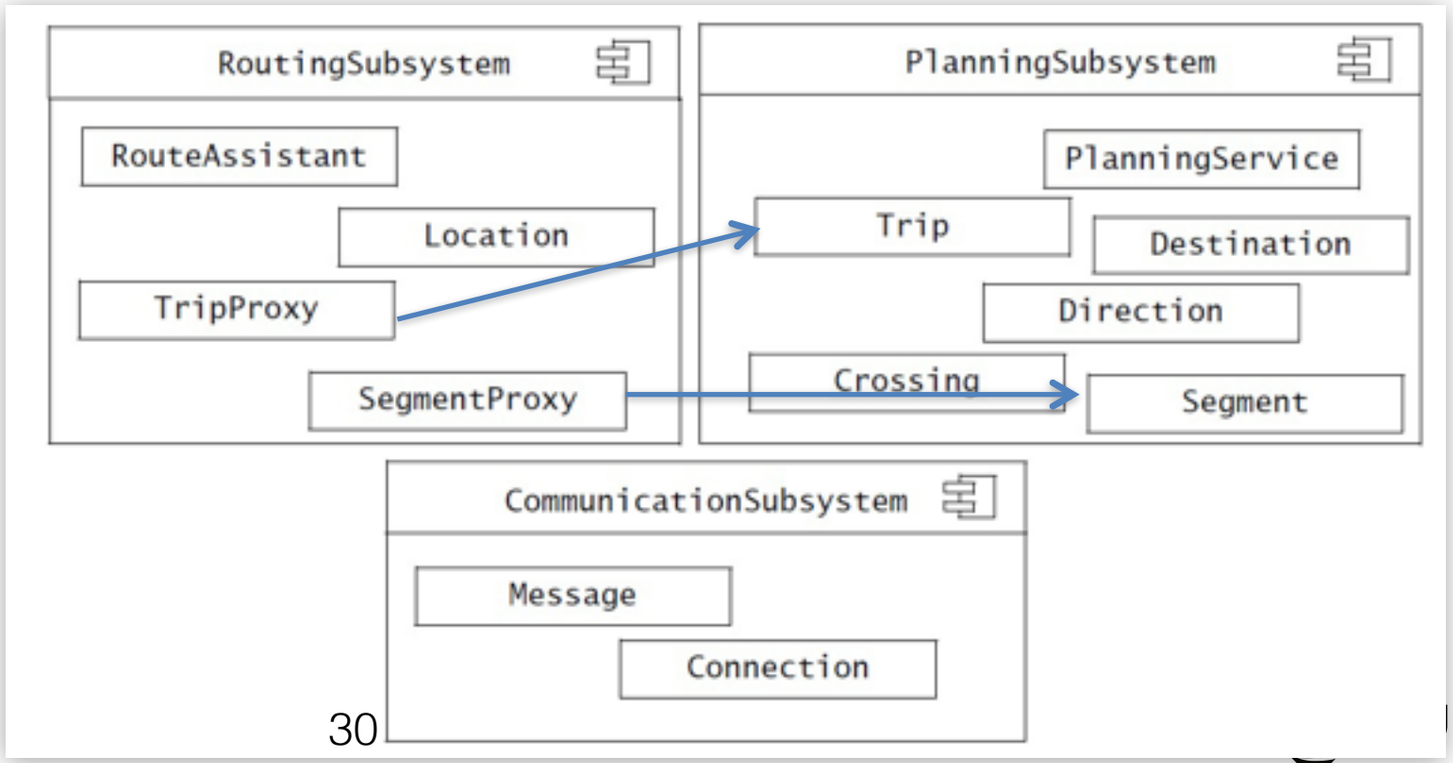
- Model-View-Controller (MVC)
- Layered
- Repository
- Client-Server
- Pipe & Filter

Mapping subsystems to processors and components

- Most systems run on multiple computers and/or virtual machines
- Hardware/VM mapping has significant impact on performance and complexity
  - is hence done early in the project
- HW/VM mapping often leads to the design of more components for data transport/mapping
- HW/VM distribution allow for exploit parallel processors, but comes with a cost in terms of
  - handling concurrency
  - storing, transferring, replicating, synchronizing data

Example


«device»
:OnBoardComputer
:RoutingSubsystem
«http»
«device»
:WebHost
«webserver»
:Apache
:PlanningSubsystem

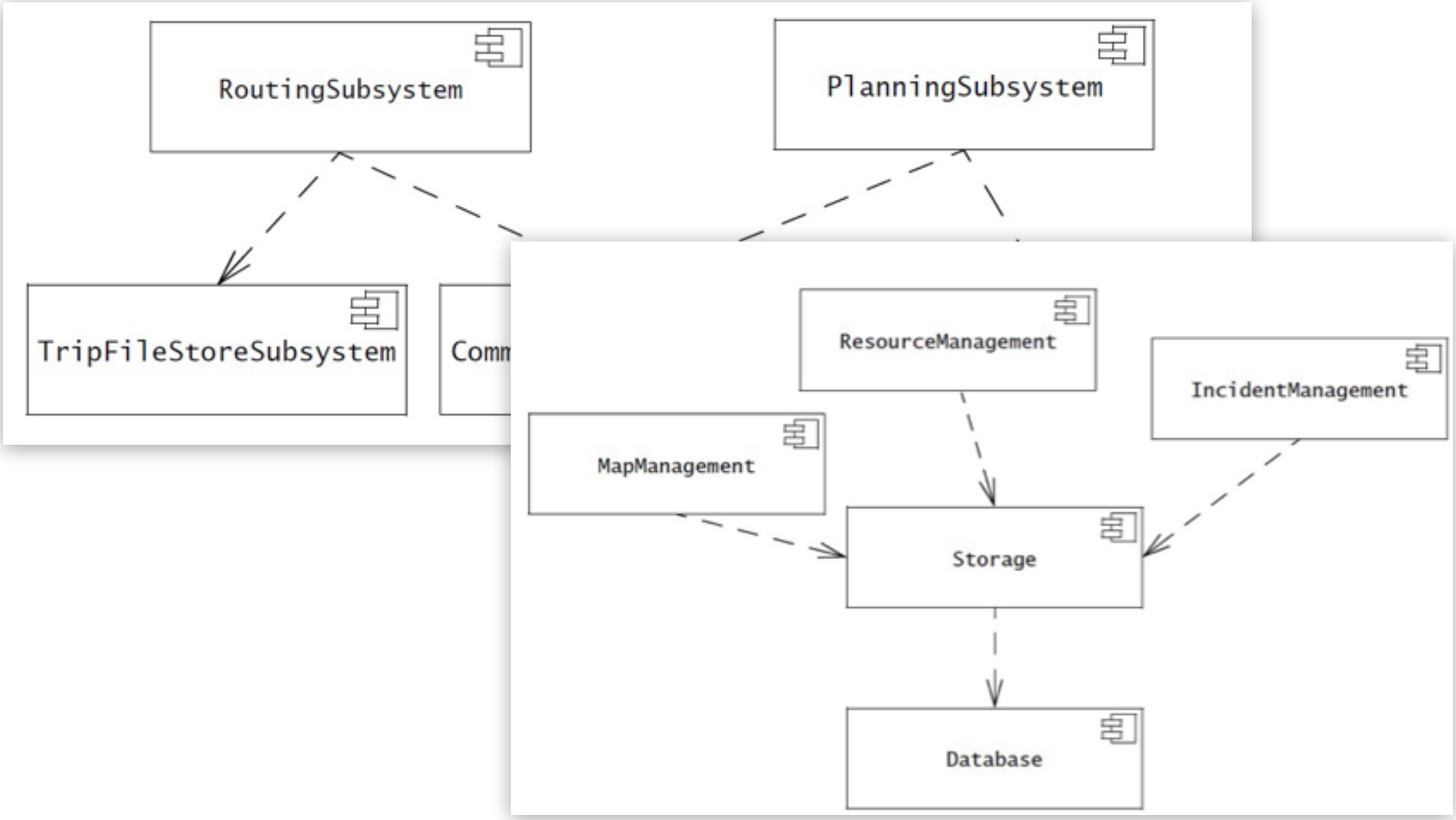

RoutingSubsystem
RouteAssistant
Location
TripProxy
SegmentProxy
PlanningSubsystem
PlanningService
Trip
Destination
Direction
Crossing
Segment
CommunicationSubsystem
Message
Connection

Identifying and storing persistent data

- Identifying persistent objects
  - typically entity objects/classes
  - all data that should survive system shutdown/restart/crash
    - can be quite a lot

- Storage management strategy
- Files
  - binary data stored in the (local) filesystem
  - no finer grained access/read/write control
  - used for large / temporary data
- Relational DB (SQL)
  - data store in table in large, distributed RDMS system
  - fine-grained concurrency control
  - used for fine-grained data management in stable data models
- noSQL (e.g., Document DB, Key Value)
  - data stored in large key-value pairs (essentially XML)
  - less constrained, simple design, horizontally scalable

Example


RoutingSubsystem
PlanningSubsystem
TripFileStoreSubsystem
Comm
ResourceManagement
IncidentManagement
MapManagement
Storage
Database

Providing access control

- In a multi-user system different actors have access to different functionality and data
  - modeled during OOA in the use case diagram
  - in OOD we model how actors can control/access objects
  - … + authentication mechanisms

- Access control matrix/list
  - rows are actors of the system
  - columns are classes
  - cells list the access right

Access matrix

Table 7-2 Access matrix for a banking system. `Tellers` can perform small transactions and inquire balances. `Managers` can perform larger transactions and access branch statistics in addition to the operations accessible to the `Tellers`. `Analysts` can access statistics for all branches, but cannot perform operations at the account level.

Objects Actors	Corporation	LocalBranch	Account
Teller			postSmallDebit() postSmallCredit() examineBalance()
Manager		examineBranchStats()	postSmallDebit() postSmallCredit() postLargeDebit() postLargeCredit() examineBalance() examineHistory()
Analyst	examineGlobalStats()	examineBranchStats()	

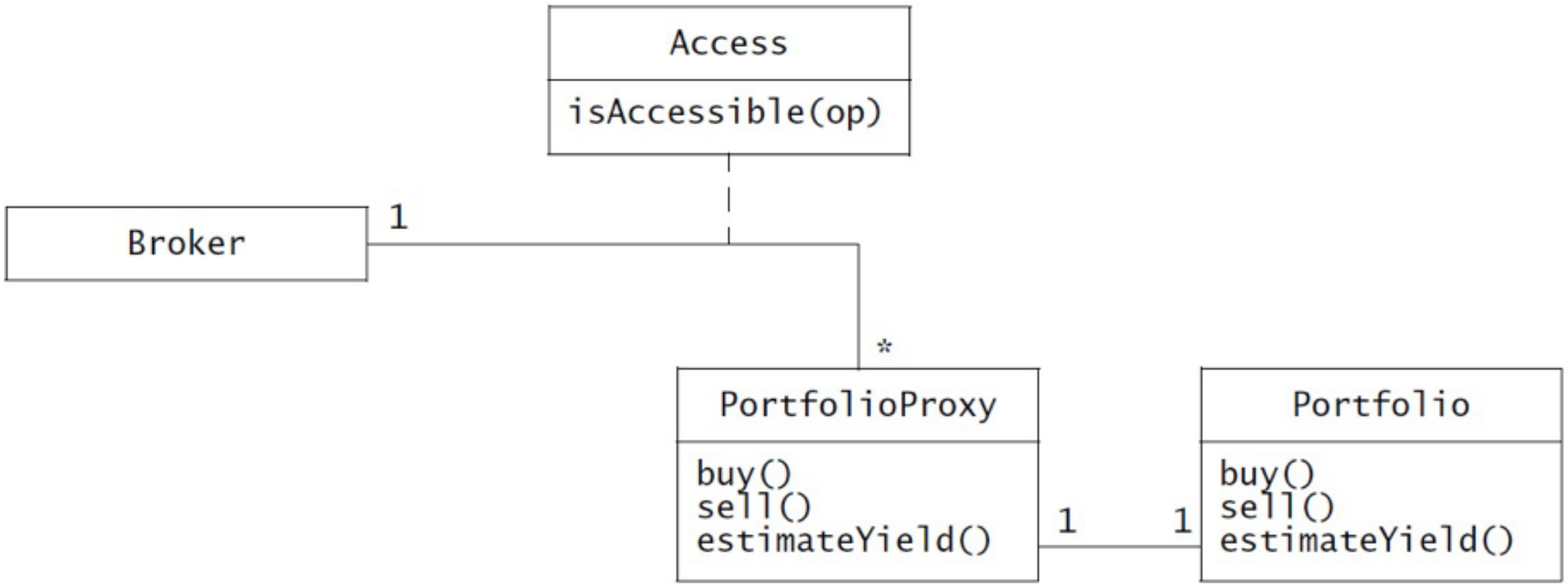
Access matrix

- Global Access Table
  - global for a whole system, controlling access globally
  - (`actor, class, operation`)
  - if no match, access denied

- Access Control List
  - list pr. class controlling access on a class level (“guest list”)
  - (`actor, operation`)
  - every time an object is accessed, its access control list is checked
  - if no entry found, access denied

- Capability List
  - list pr. actor controlling what an actor can do (“invitation card”)
  - (`class, operation`)
  - if no capability, no access

Dynamic Access Control using the Proxy Pattern



Designing the global control flow

- Procedure-driven control
  - operations wait for input from actor(s)
  - "old school" procedural, legacy systems
  - pipe-and-filters architectures

- Event-driven control
  - main loop wait for external event and dispatch them to interested parties
  - modern UI / OOP systems
  - MVC / event architectures

- Threads
  - each event spawn a concurrent thread / event handler
  - modern UI / web systems
  - client/server architectures

Example

```
    Stream in, out;
    String userid, passwd;
/* Initialization omitted */
    out.println("Login:");
    in.readln(userid);
    out.println("Password:");
    in.readln(passwd);
    if (!security.check(userid, passwd)) {
        out.println("Login failed.");
        system.exit(-1);
    }
/* ...*/
```

```
Iterator subscribers, eventStream;
Subscriber subscriber;
Event event;
EventStream eventStream;
/* ... */
while (eventStream.hasNext()) {
    event = eventStream.next();
    subscribers = dispatchInfo.getSubscribers(event);
    while (subscribers.hasNext()) {
        subscriber = subscribers.next()) {
        subscriber.process(event);
    }
}
/* ... */
```

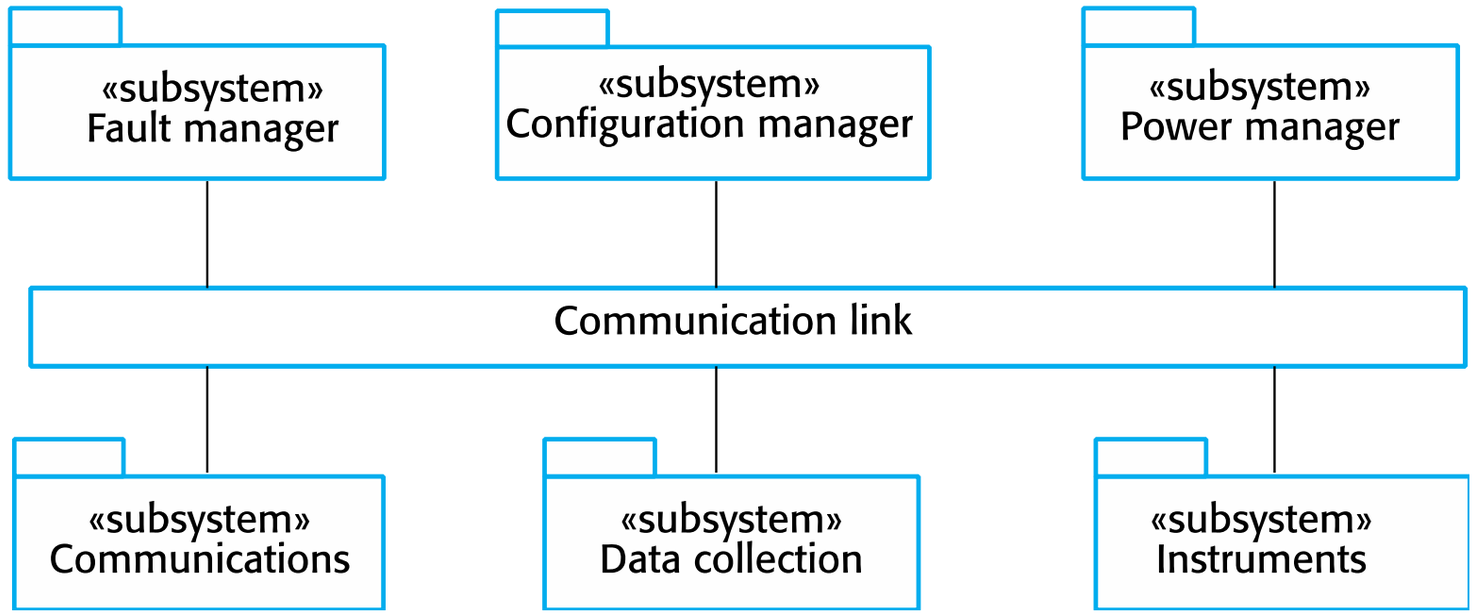
```
Thread thread;
Event event;
EventHandler eventHandler;
boolean done;
/* ...*/
while (!done) {
    event = eventStream.getNextEvent();
    eventHandler = new EventHandler(event)
    thread = new Thread(eventHandler);
    thread.start();
}
/* ...*/
```

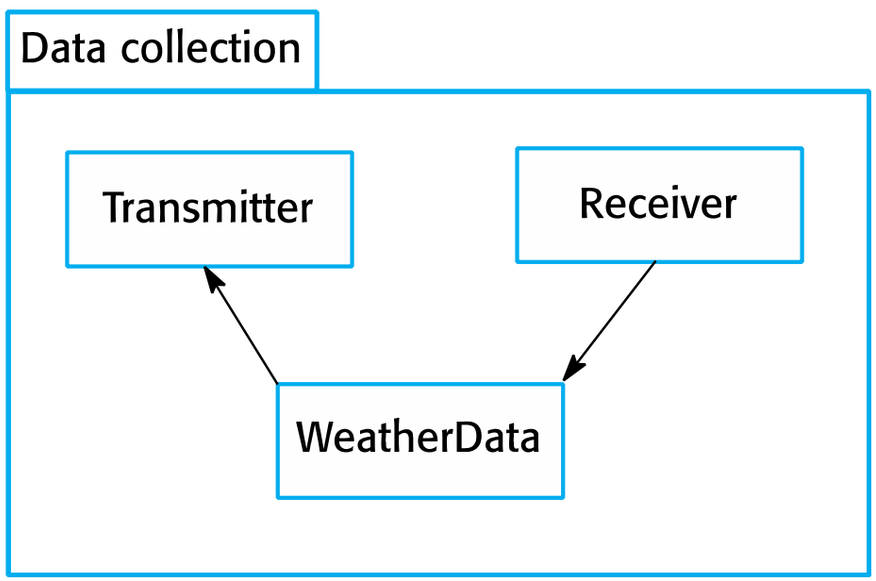
Dynamic UML Diagrams

- Dynamic UML diagrams are used to described Control Flow
  - Sequence Diagram
  - Communication Diagram
- ... and State of the system
  - State Machine Diagram

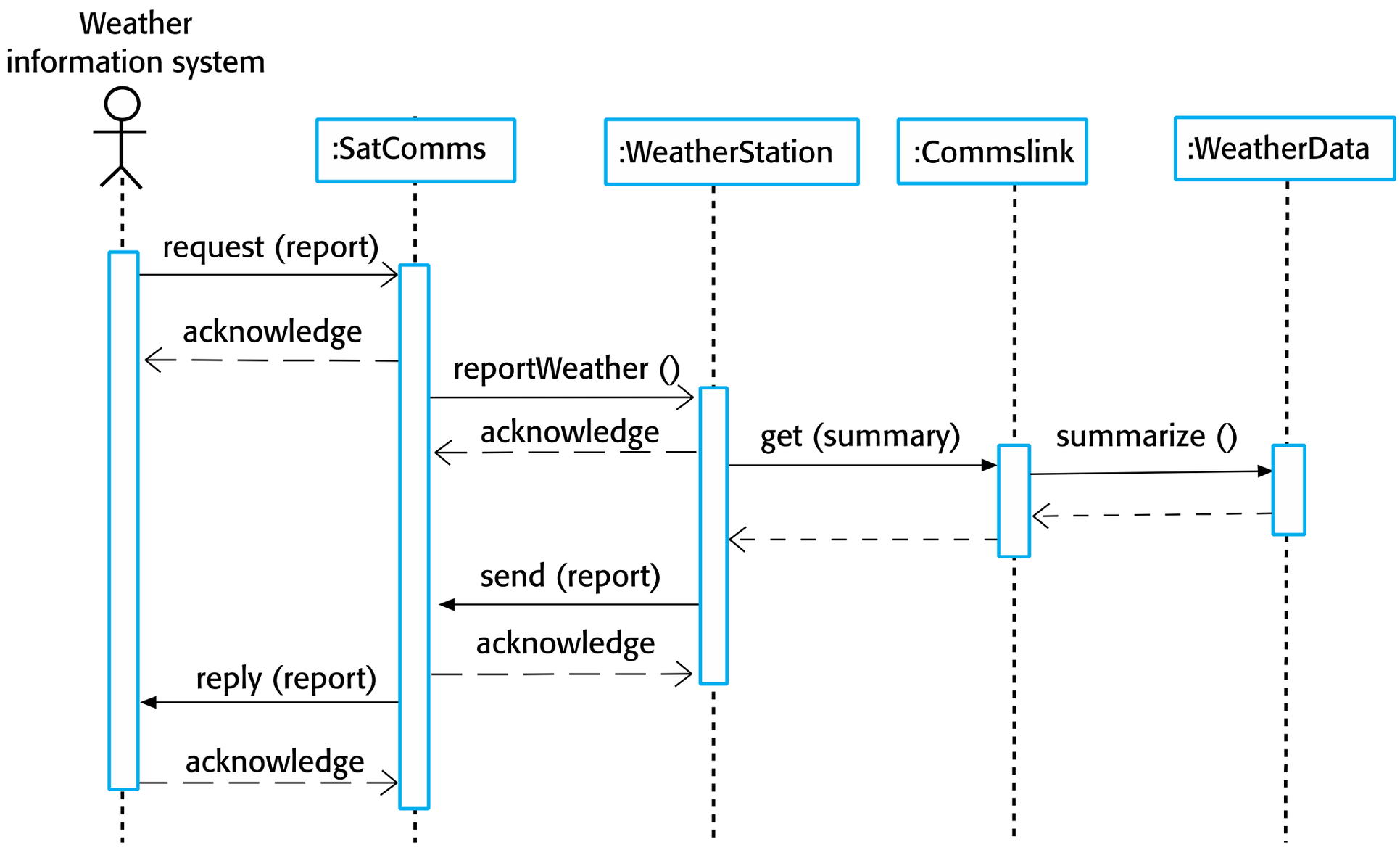
- This is IMPORTANT to remember
- ... software architecture design consists of
  - structural models
  - dynamic models
  - see [SE9] section 7.1.4

Structural models

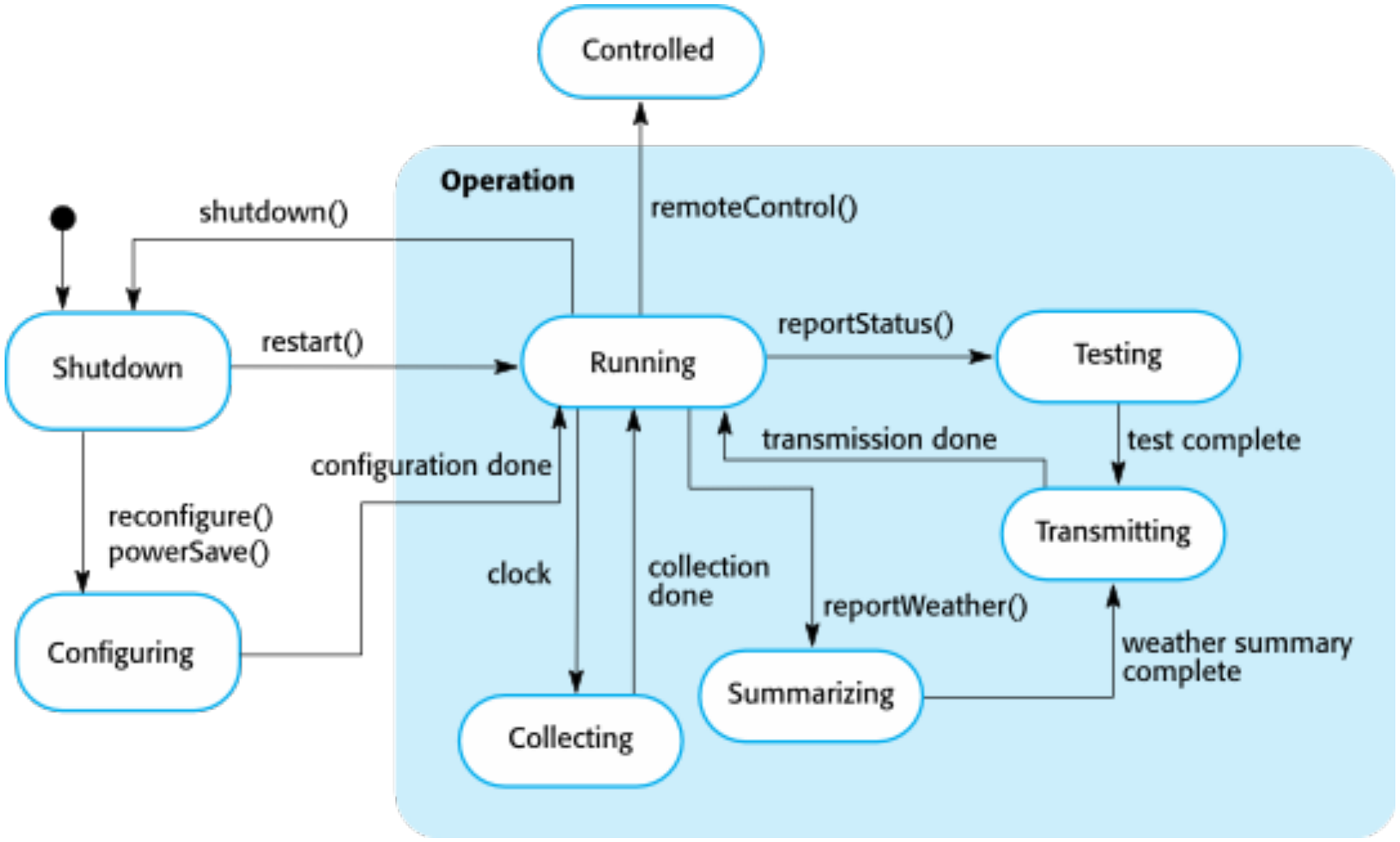




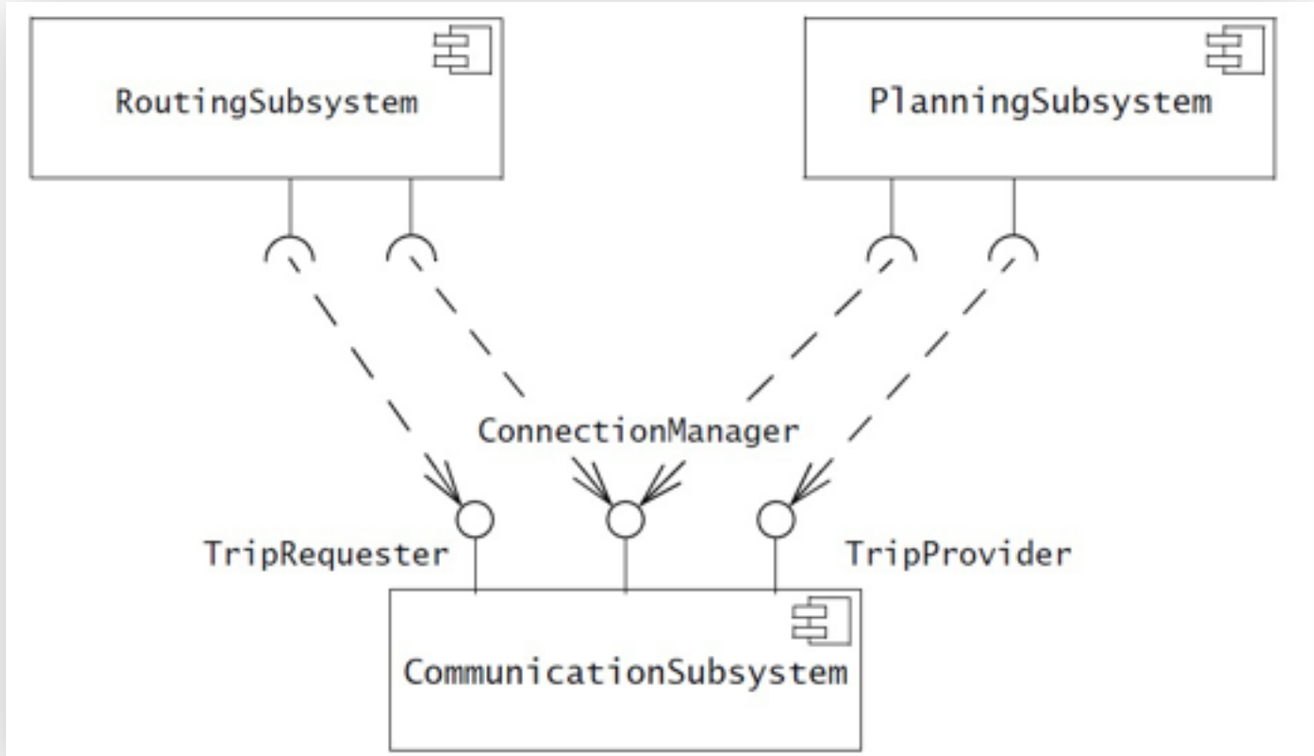
Dynamic model: sequence diagram



Dynamic model: state machine diagram



Identifying services



- Refine subsystem decomposition by identifying services provided by each subsystem
  - dependencies between subsystems
  - responsibility of each subsystem
- Services are named as nouns

Identifying boundary conditions

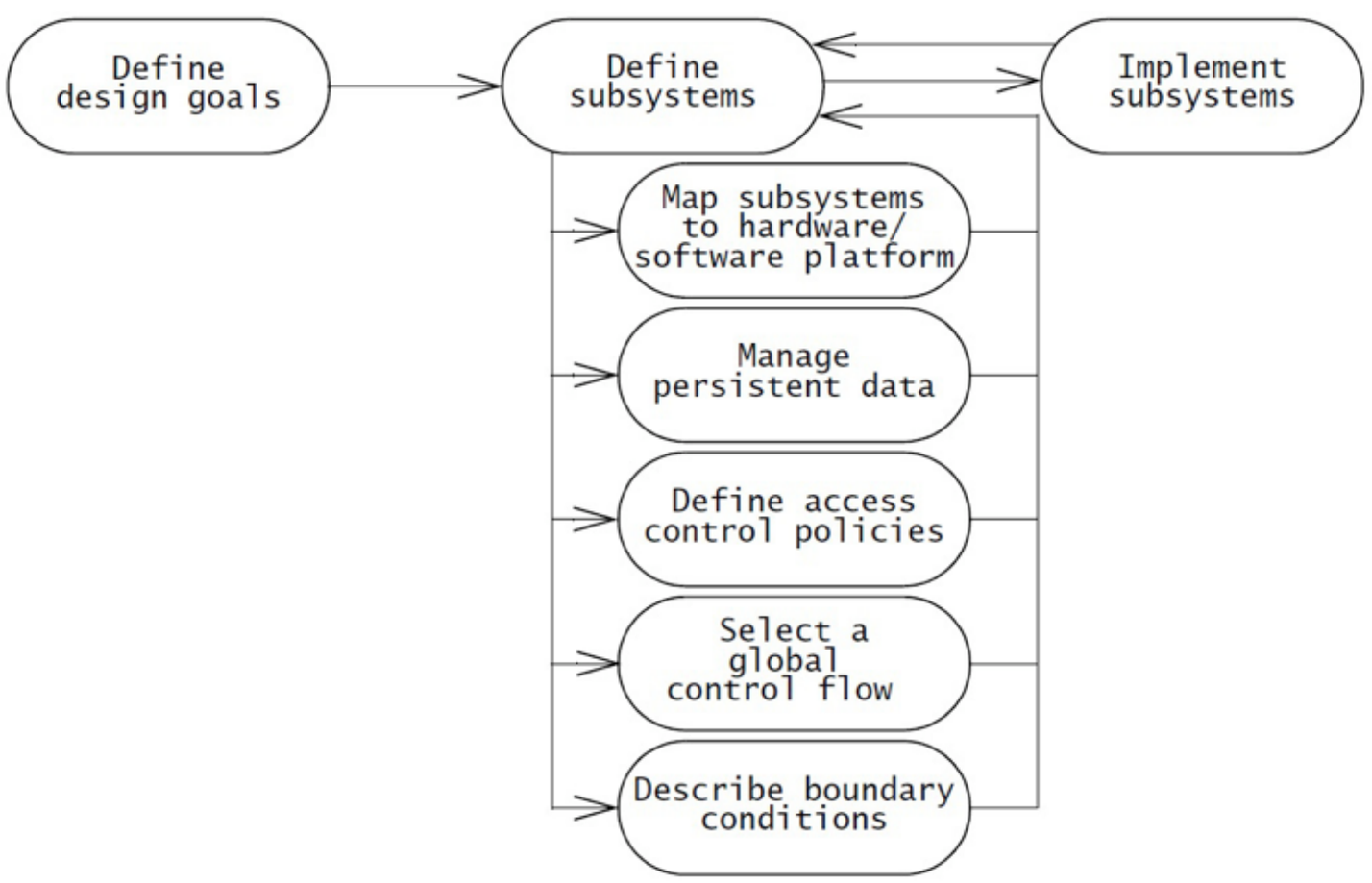
- Boundary conditions: how the system is
  - started, initialized, shut down
  - behave during exceptions, faults, data corruptions, etc.

- Extend the OOA model with Boundary Use Cases
  - ... or extend existing ones

- Typical boundary cases
  - Configuration (actor = system administrator)
  - Start-up and Shut-down
  - Exception handling—all components can fail!
    - hardware, network, etc.
    - operating environment (OS, DB, ...)
    - software components
    - your system

Reviewing the SDD

- Like the analysis, the design is an evolutionary and iterative activity
- Meet design goals, but the system design model must also be:
  - correct
  - complete
  - consistent
  - realistic
  - readable



System Design Document (SDD)

System Design Document

1. Introduction
   - 1.1 Purpose of the system
   - 1.2 Design goals
   - 1.3 Definitions, acronyms, and abbreviations
   - 1.4 References
   - 1.5 Overview
2. Current software architecture
3. Proposed software architecture
   - 3.1 Overview
   - 3.2 Subsystem decomposition
   - 3.3 Hardware/software mapping
   - 3.4 Persistent data management
   - 3.5 Access control and security
   - 3.6 Global software control
   - 3.7 Boundary conditions
4. Subsystem services

Glossary

Study the ARENA case

7.6 ARENA Case Study

In this section, we apply the concepts and methods described in this chapter to the ARENA system. We start with identifying the design goals for ARENA and design an initial subsystem decomposition. We then select a software and hardware platform and define the persistent stores, access control, and global control flow. Finally, we look at the boundary conditions of ARENA.

7.6.1 Identifying Design Goals

Design goals are qualities that enable us to prioritize the development of the system. Design goals originate from the nonfunctional requirements specified during requirements elicitation and from technical and management goals specified by the project

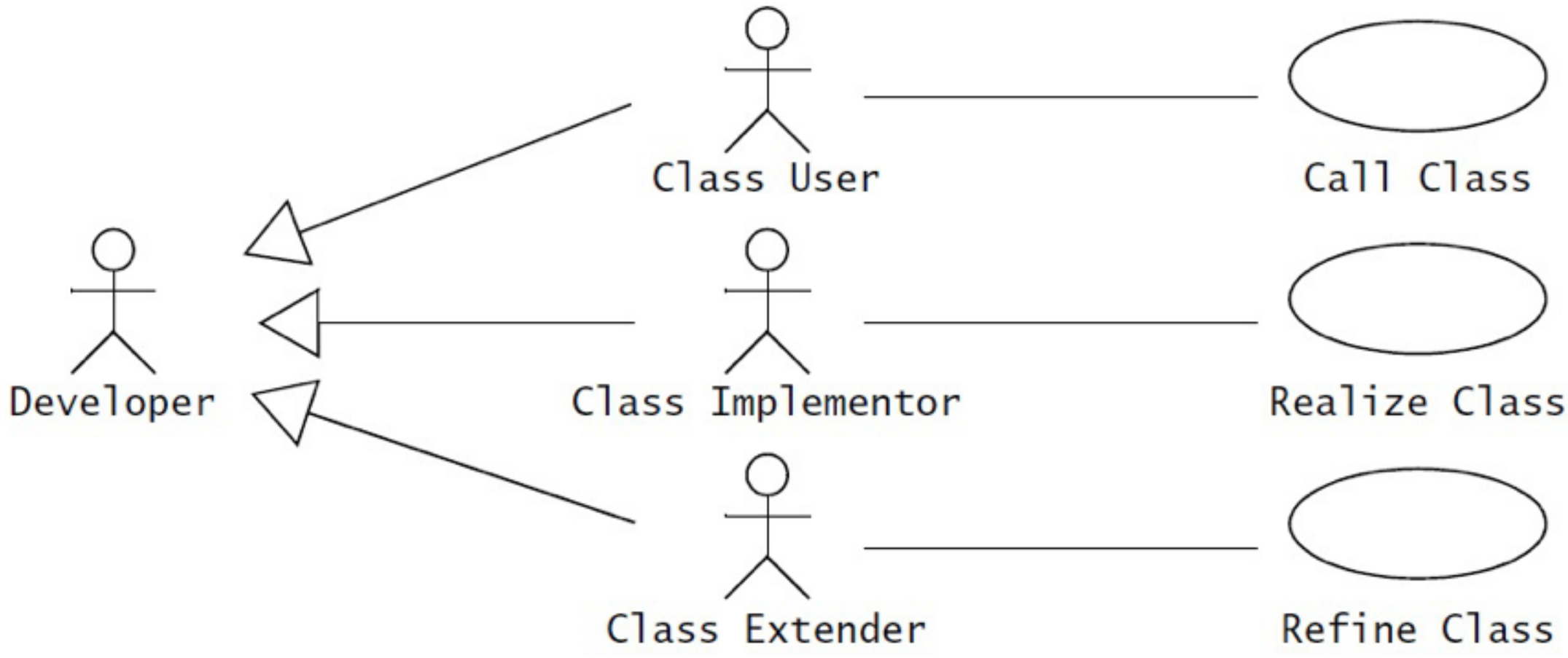
Object design—Specifying interfaces

Interface specification

- Goal
  - to describe the interface of "important" objects precisely enough to that objects realized by different developers fit together with minimal integration issues

- Activities
  - Identify missing attributes and operations
  - Specify visibility and signatures
  - Specify contracts

Roles: implementor, extended, and user



Types, signatures, and visibility

- Visibility
  - - private
  - + public
  - # protected
  - ~ package

```
public class Tournament {
    private int maxNumPlayers;
    /* Other fields omitted */

    public Tournament(League l, int maxNumPlayers)
    public int getMaxNumPlayers() {…};
    public List getPlayers() {…};
    public void acceptPlayer(Player p) {…};
    public void removePlayer(Player p) {…};
    public boolean isPlayerAccepted(Player p) {…};

    /* Other methods omitted */
}
```

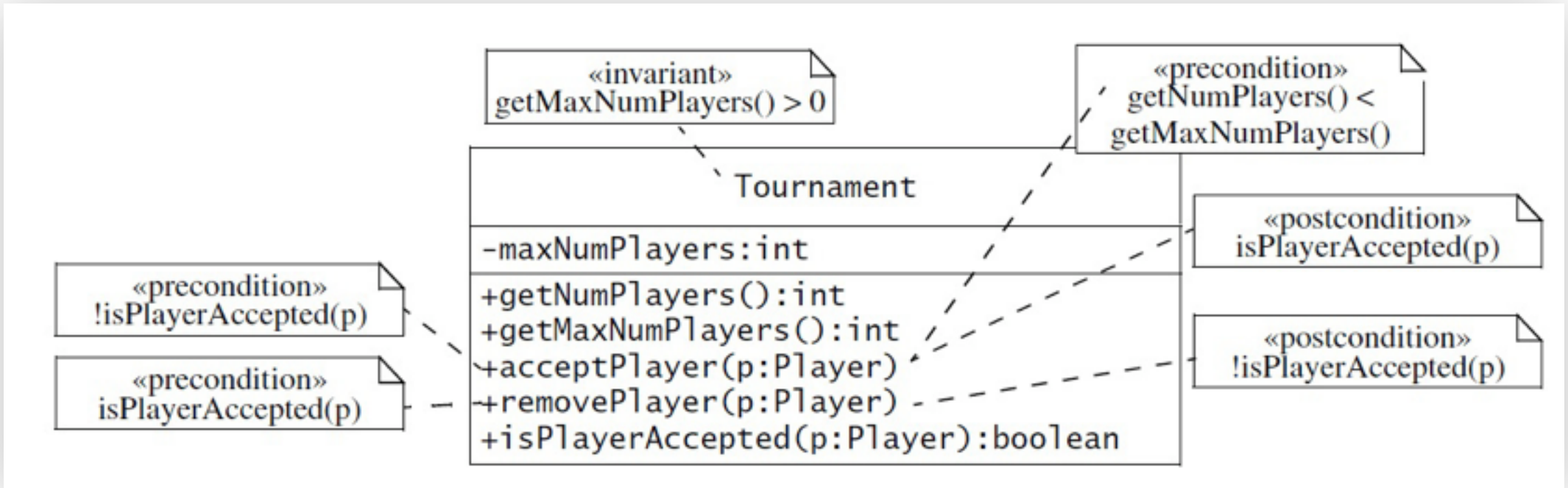
Contracts

- Invariants
  - predicate that is always true for all instances of a class
  - associated with classes and/or interfaces
  - `t.getMaxNumPlayers() > 0`

- Precondition
  - predicate that must be true before an operation is invoked
  - associated with a specific operation (can be specified on the interface)
  - `!t.isPlayerAccepted(p) and t.getNumPlayers() < t.getMaxNumPlayers()`

- Postcondition
  - predicate that must be true after an operation is done
  - associated with a specific operation (can be specified on the interface)
  - `t.getNumPlayers_afterAccept = t.getNumPlayers_beforeAccepts+1`

Object constraint language (OCL)



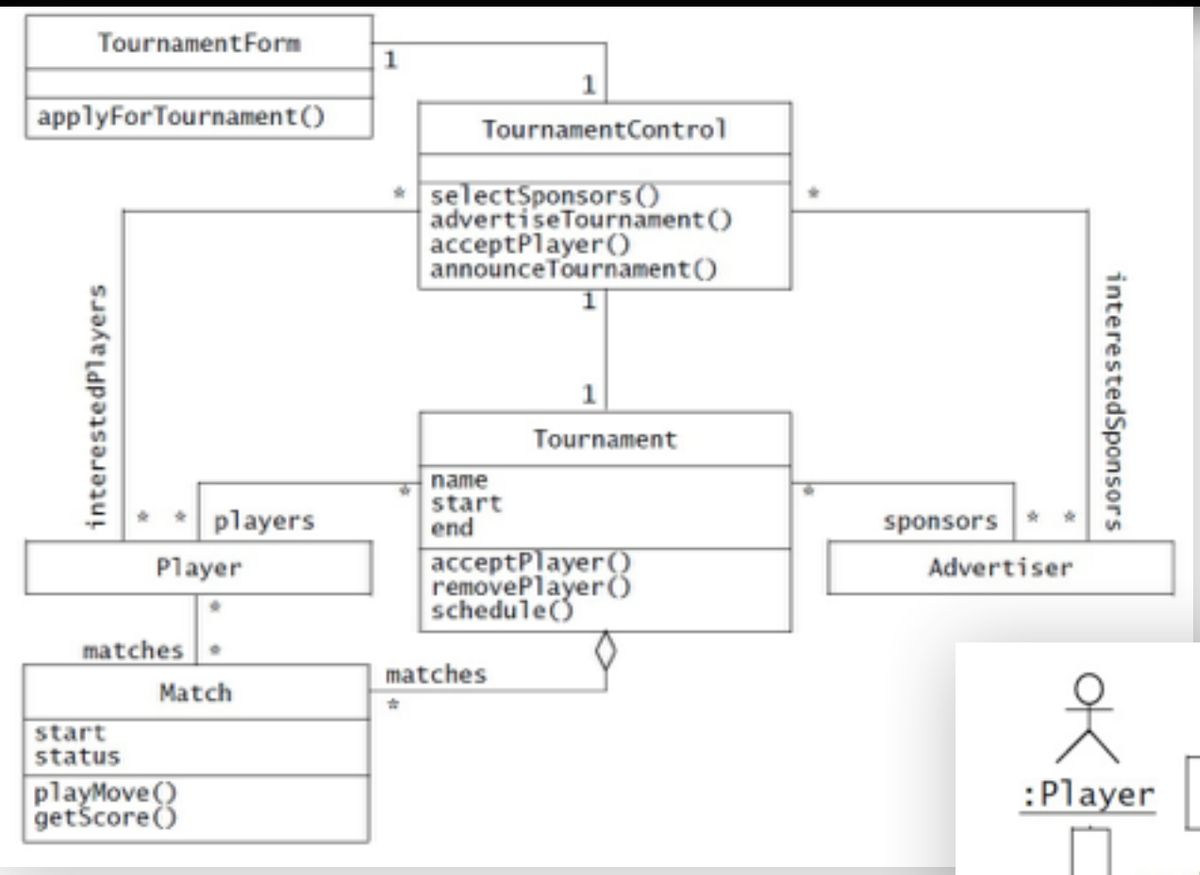
- Object Constraint Language (OCL)
  - part of UML 2.x
  - a constraint is expressed as a boolean expression
  - can be depicted as a note in an UML class diagram

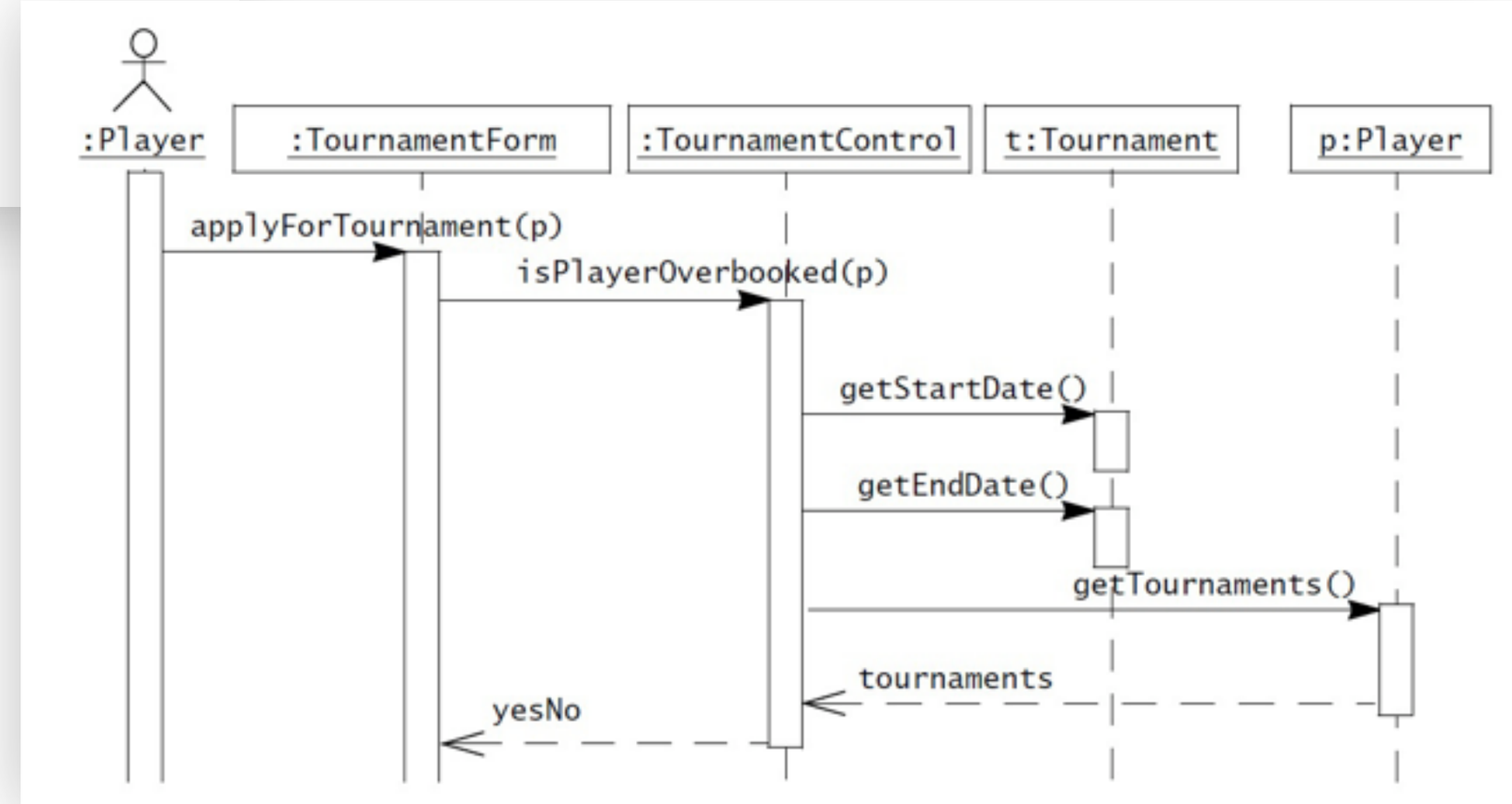
Interface specification

- Interface Specification Activities
  - Identify missing attributes and operations
  - Specify type signatures and visibility
  - Specify pre- and post-conditions
  - Specify invariants

Identifying missing attributes and operations

- Identify missing attributes and operations
- Specify type signatures and visibility
- Specify pre- and post-conditions
- Specify invariants





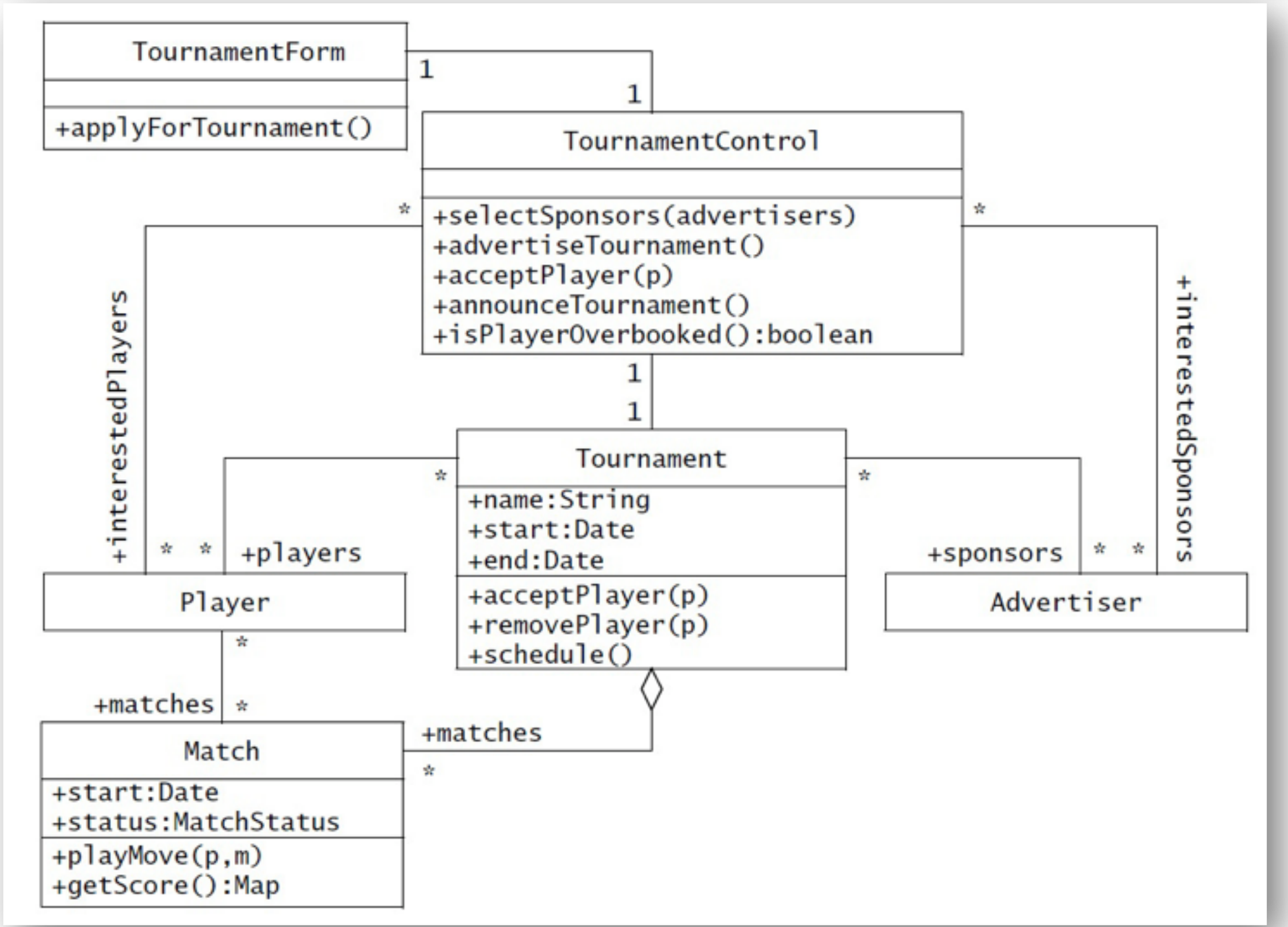
Specifying types, signatures, and visibility

Identify missing attributes and operations

Specify type signatures and visibility

Specify pre- and post-conditions

Specify invariants



Specifying pre- and post-conditions

Identify missing attributes and operations

Specify type signatures and visibility

Specify pre- and post-conditions

Specify invariants

```
public class Tournament {

    /* Fields omitted */

    /** Public constructor. A Tournament starts with a league, a name,
     * and a maximum number of players. These attributes cannot be changed.
     */
    public Tournament(League league, String name, int maxNumPlayers) {…}

    /** Returns a list of the current Players
     */
    public List getPlayers() {…}

    /** This operation accepts a new Player in the Tournament.
     * @pre !isPlayerAccepted(p)
     * @pre getPlayers().size() < getMaxNumPlayers()
     * @post isPlayerAccepted(p)
     * @post getPlayers().size() = self@pre.getPlayers().size() + 1
     */
    public void acceptPlayer (Player p) {…}

    /** The removePlayer() operation assumes that the specified player
     * is currently in the Tournament.
     * @pre isPlayerAccepted(p)
     * @post !isPlayerAccepted(p)
     * @post getPlayers().size() = self@pre.getPlayers().size() - 1
     */
    public void removePlayer(Player p) {…}
```

Specifying invariants

Identify missing attributes and operations

Specify type signatures and visibility

Specify pre- and post-conditions

Specify invariants

```
/** A Tournament is a series of Matches among a set of Players
* which ends with a single winner. The Game and TournamentStyle of a
* TournamentStyle is determined by the League in which the Tournament is
* played.
*
* The Tournament starts empty, a number of Players are accepted in
* the Tournament, the Tournament is planned, and finally,
* the Matches are played.
*
* Invariants:
* The maximum number of players is positive at all times.
*    @invariant getMaxNumPlayers > 0
* The number of players is always less or equal than the max number.
*    @invariant getPlayers().size() < getMaxNumPlayers()
* The initial attributes of the Tournament are not null.
*    @invariant getLeague() != null and getName() != null
*/
public class Tournament {

    /* Fields omitted */

    /** Public constructor. A Tournament starts with a league, a name,
     * and a maximum number of players. These attributes cannot be changed.
     */
    public Tournament(League league, String name, int maxNumPlayers) {…}
```

Documenting object design

- Fundamental challenge in OOA/OOD (!#€”%&#”€%§)
  - how to maintain CONSISTENCY between UML models and Code????
- 3 approaches
  - Self-contained OOD generated from model
    - we write and maintain UML models and generate code
    - problem : inaccurate OOA/OOD and out-of-date
  - OOD as an extension of the RAD
    - keep the OOA and OOD as “one” model
    - problem : pollution of OOA with implementation details
  - OOD embedded in source code
    - start w. the OOD UML models and move into code
    - use JavaDoc / XML Summary to describe the interfaces and classes
    - problem : no UML
    - advantage : less work

My recommendations

- OOA
  - build and maintain the OOA model to reflect requirements and analysis
    - Use cases, class diagrams, sequence diagrams, etc.
  - keep it updated to always reflect current requirements/analysis
  - document it in the RAD
- OOD
  - build an initial model to reflect architecture and design
  - document it in the OOD
- CODE
  - build the system in code
  - add exception handling (e.g. w. invariants and pre- & post conditions)
  - maintain source code documentation
  - extensive use of unit testing (and continuous integration)
- When the system code is stable [finished | released]
  - go back and update the OOA and OOD documentation
  - make it reflect what is actually implemented
- Release the code + documentation as one coherent package

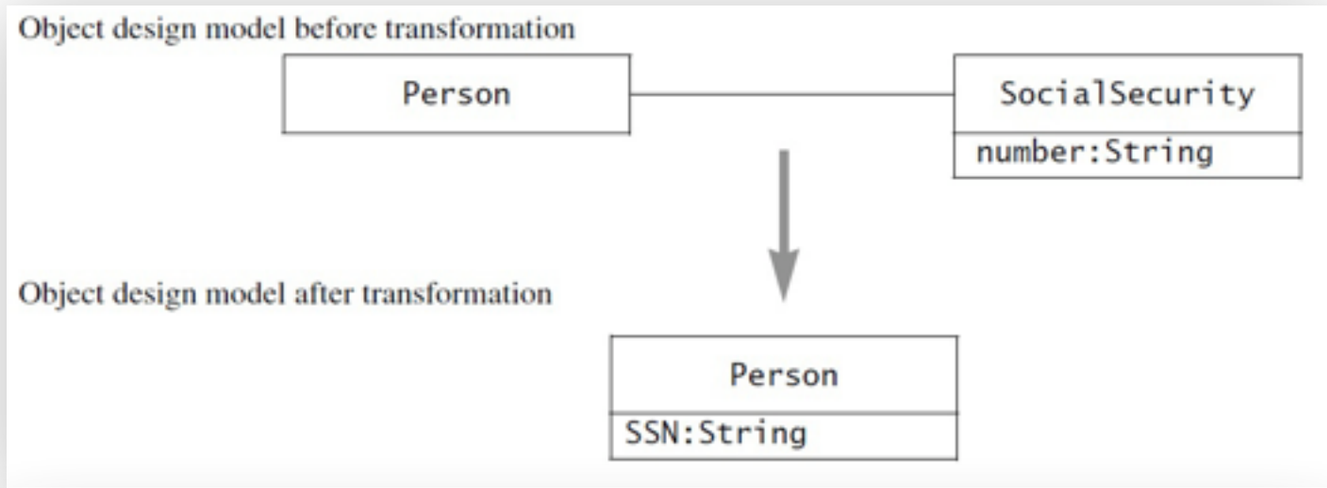
Object design—Mapping models to code

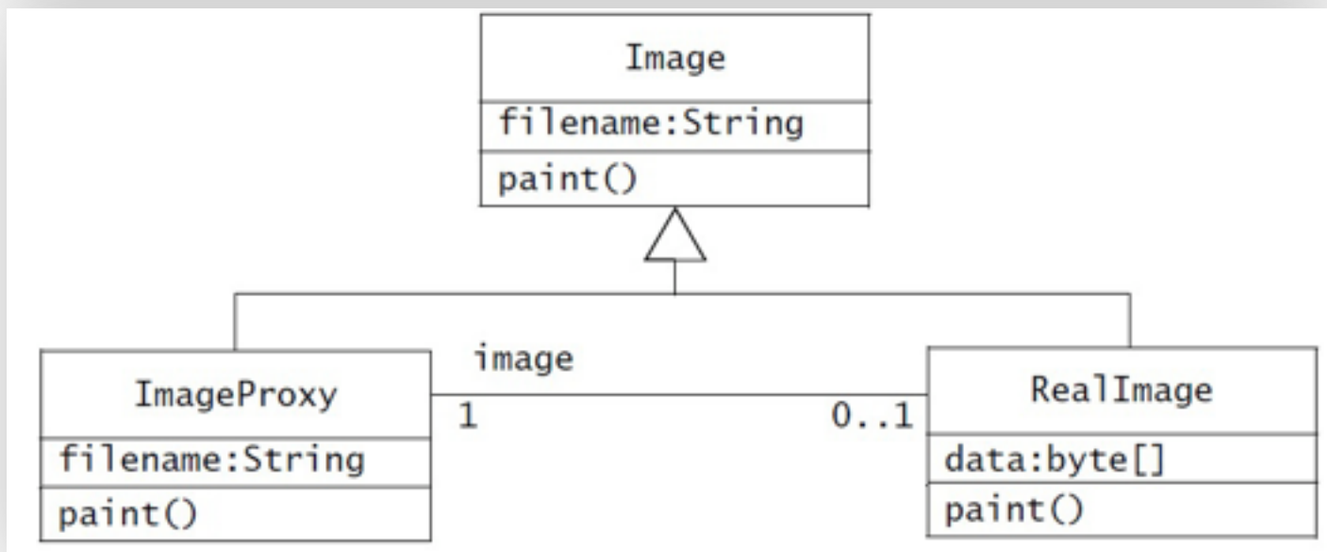
Mapping models to code

- Mapping activities
  - Optimizing the Object Design Model
  - Mapping Associations to Collections
  - Mapping Contract to Exceptions
  - Mapping Abject Model to a Persistent Storage Schema

Optimizing the Object Design Model

- Adding associations to optimizing access paths
  - make "short-cuts", use hashing, clean up
- Collapsing objects into attributes
  - remove "verbose" classes
- Delaying expensive computation
  - e.g. by using the Proxy pattern
- Caching the result of expensive computations





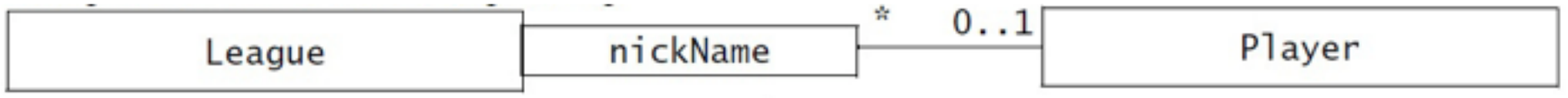
Mapping Associations to Collections

- Unidirectional one-to-one association
- Bidirectional one-to-one association

- One-to-many association

- Many-to-many association

- Qualified association



Unidirectional one-to-one association

Object design model before transformation

Source code after transformation

```
public class Advertiser {
    private Account account;
    public Advertiser() {
        account = new Account();
    }
    public Account getAccount() {
        return account;
    }
}
```

Bidirectional one-to-one association

Object design model before transformation

Source code after transformation

```
public class Advertiser {
    /* The account field is initialized
     * in the constructor and never
     * modified. */
    private Account account;

    public Advertiser() {
        account = new Account(this);
    }
    public Account getAccount() {
        return account;
    }
}
```

```
public class Account {
    /* The owner field is initialized
     * during the constructor and
     * never modified. */
    private Advertiser owner;

    public Account(Advertiser owner) {
        this.owner = owner;
    }
    public Advertiser getOwner() {
        return owner;
    }
}
```

Bidirectional one-to-many association

Object design model before transformation

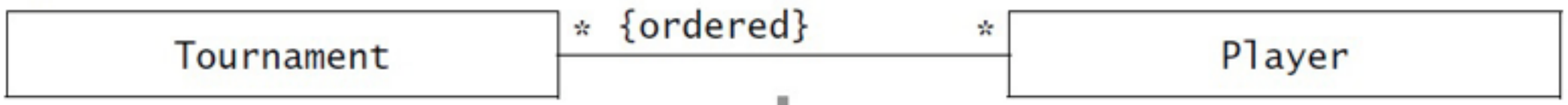
Source code after transformation

```
public class Advertiser {
    private Set accounts;
    public Advertiser() {
        accounts = new HashSet();
    }
    public void addAccount(Account a) {
        accounts.add(a);
        a.setOwner(this);
    }
    public void removeAccount
            (Account a) {
        accounts.remove(a);
        a.setOwner(null);
    }
}
```

```
public class Account {
    private Advertiser owner;
    public void setOwner
            (Advertiser newOwner) {
        if (owner != newOwner) {
            Advertiser old = owner;
            owner = newOwner;
            if (newOwner != null)
              newOwner.addAccount(this);
            if (old != null)
              old.removeAccount(this);
        }
    }
}
```

Bidirectional many-to-many association

Object design model before transformation

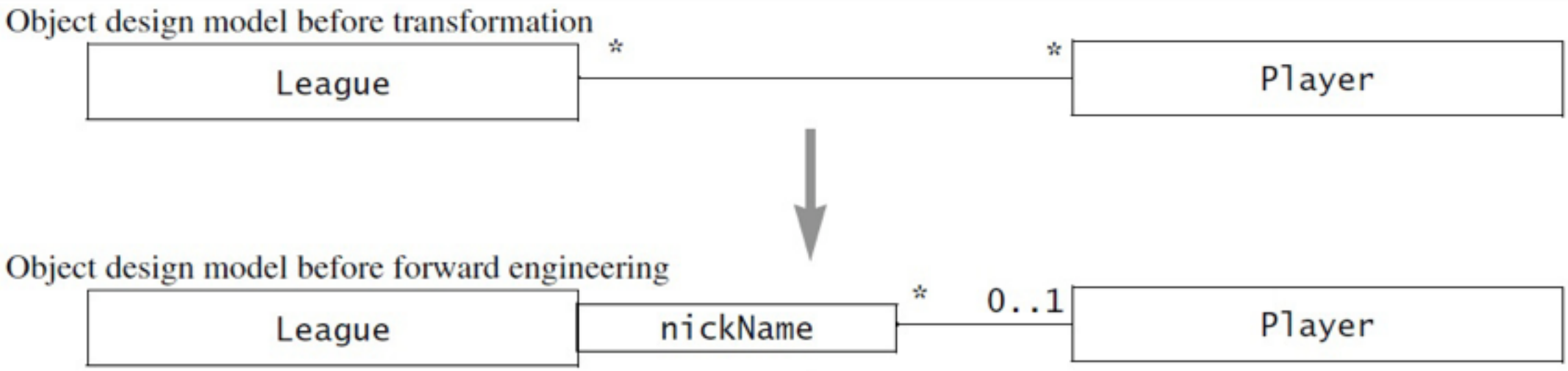


Source code after transformation

```
public class Tournament {
    private List players;
    public Tournament() {
        players = new ArrayList();
    }
    public void addPlayer(Player p) {
        if (!players.contains(p)) {
            players.add(p);
            p.addTournament(this);
        }
    }
}
```

```
public class Player {
    private List tournaments;
    public Player() {
        tournaments = new ArrayList();
    }
    public void addTournament
            (Tournament t) {
        if (!tournaments.contains(t)) {
            tournaments.add(t);
            t.addPlayer(this);
        }
    }
}
```

Bidirectional many-to-many association



Source code after forward engineering

```
public class League {
    private Map players;

    public void addPlayer
            (String nickName, Player p) {
    if (!players.containsKey(nickName))
    {
        players.put(nickName, p);
        p.addLeague(nickName, this);
    }
    }
}
```

```
public class Player {
    private Map leagues;

    public void addLeague
            (String nickName, League l) {
        if (!leagues.containsKey(l)) {
            leagues.put(l, nickName);
            l.addPlayer(nickName, this);
        }
    }
}
```

Mapping contracts to exceptions

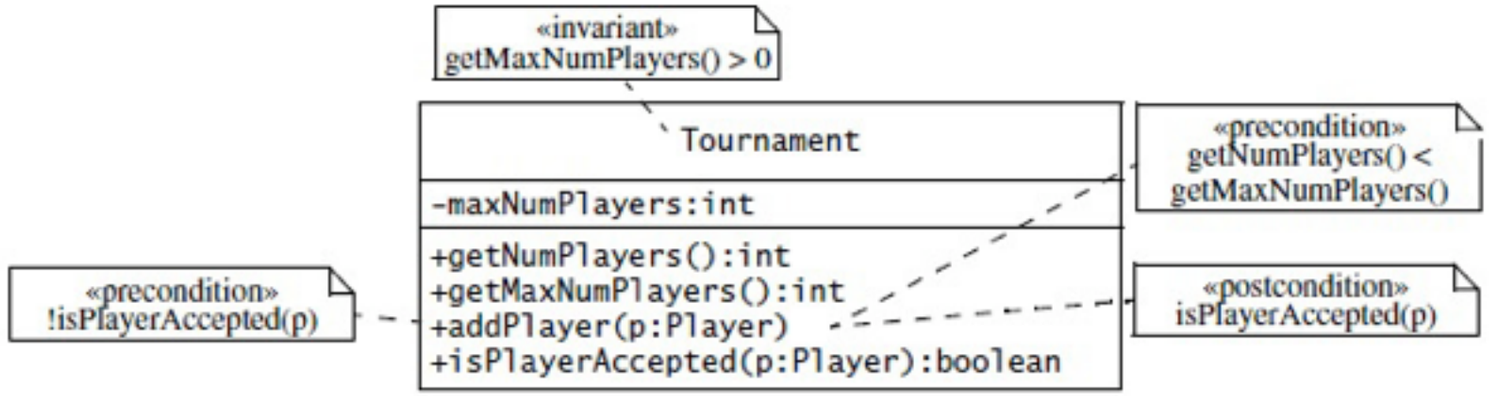
- Simple mapping
  - add code to each operation that checks the pre-, post-conditions, and invariants relevant to the operation
  - preconditions: at the beginning
  - postconditions: at the end
  - invariants: at the same time as postconditions

- Implements contract handling using exceptions
  - modeling useful and powerful exceptions is as important as modeling "regular" classes
  - a "high-quality" OO program consists of
    - interfaces
    - classes
    - exceptions

Example

```
public class TournamentControl {
    private Tournament tournament;
    public void addPlayer(Player p) throws KnownPlayerException {
        if (tournament.isPlayerAccepted(p)) {
            throw new KnownPlayerException(p);
        }
        //... Normal addPlayer behavior
    }
}
```

```
public class TournamentForm {
    private TournamentControl control;
    private List players;
    public void processPlayerApplications() {
        // Go through all the players who applied for this tournament
        for (Iterator i = players.iterator(); i.hasNext();) {
            try {
                // Delegate to the control object.
                control.acceptPlayer((Player)i.next());
            } catch (KnownPlayerException e) {
                // If an exception was caught, log it to the console, and
                // proceed to the next player.
                ErrorConsole.log(e.getMessage());
            }
        }
    }
}
```




```
public class Tournament {
//...
    private List players;

    public void addPlayer(Player p)
        throws KnownPlayer, TooManyPlayers, UnknownPlayer,
            IllegalNumPlayers, IllegalMaxNumPlayers
    {
        // check precondition!isPlayerAccepted(p)
        if (isPlayerAccepted(p)) {
            throw new KnownPlayer(p);
        }
        // check precondition getNumPlayers() < maxNumPlayers
        if (getNumPlayers() == getMaxNumPlayers()) {
            throw new TooManyPlayers(getNumPlayers());
        }
        // save values for postconditions
        int pre_getNumPlayers = getNumPlayers();

        // accomplish the real work
        players.add(p);
        p.addTournament(this);

        // check post condition isPlayerAccepted(p)
        if (!isPlayerAccepted(p)) {
            throw new UnknownPlayer(p);
        }
        // check post condition getNumPlayers() = @pre.getNumPlayers() + 1
        if (getNumPlayers() != pre_getNumPlayers + 1) {
            throw new IllegalNumPlayers(getNumPlayers());
        }
        // check invariant maxNumPlayers > 0
        if (getMaxNumPlayers() <= 0) {
            throw new IllegalMaxNumPlayers(getMaxNumPlayers());
        }
    }
    //...
}
```

While this approach results in a robust system (assuming the checking code is correct), it is not realistic....

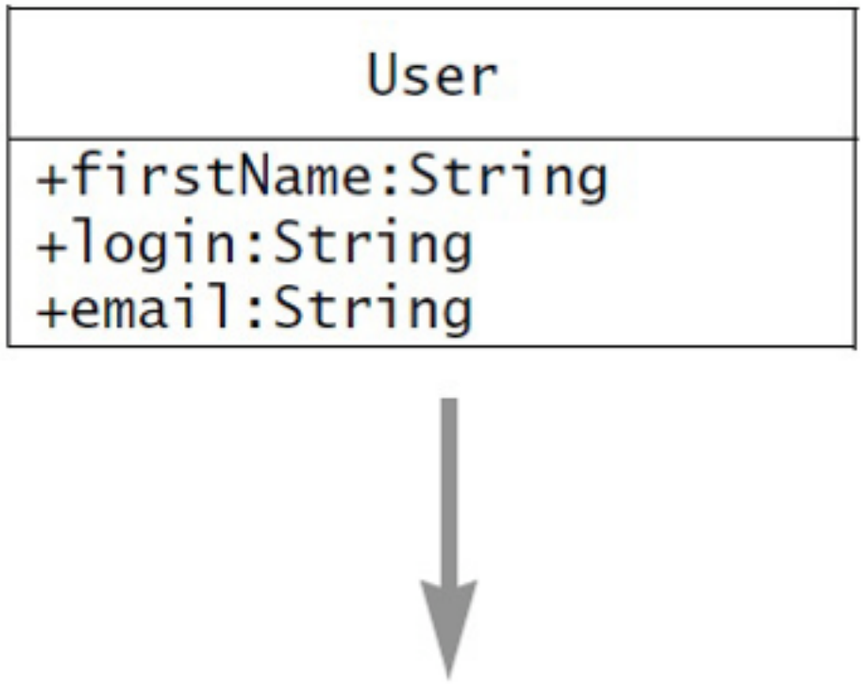
Heuristics for mapping contracts to code

Heuristics for mapping contracts to exceptions

- *Omit checking code for postconditions and invariants*. Checking code is usually redundant with the code accomplishing the functionality of the class, and is written by the developer of the method. It is not likely to detect many bugs unless it is written by a separate tester.
- *Focus on subsystem interfaces* and omit the checking code associated with private and protected methods. System boundaries do not change as often as internal interfaces and represent a boundary between different developers.
- *Focus on contracts for components with the longest life*, that is, on code most likely to be reused and to survive successive releases. Entity objects usually fulfill these criteria, whereas boundary objects associated with the user interface do not.
- *Reuse constraint checking code*. Many operations have similar preconditions. Encapsulate constraint checking code into methods so that they can be easily invoked and so that they share the same exception classes.

Object design—Mapping OO to RDBMS

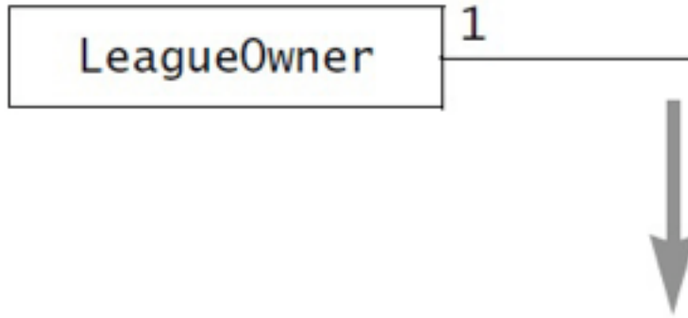
Mapping classes and attributes



User table

id:long	firstName:text[25]	login:text[8]	email:text[32]

Mapping associations

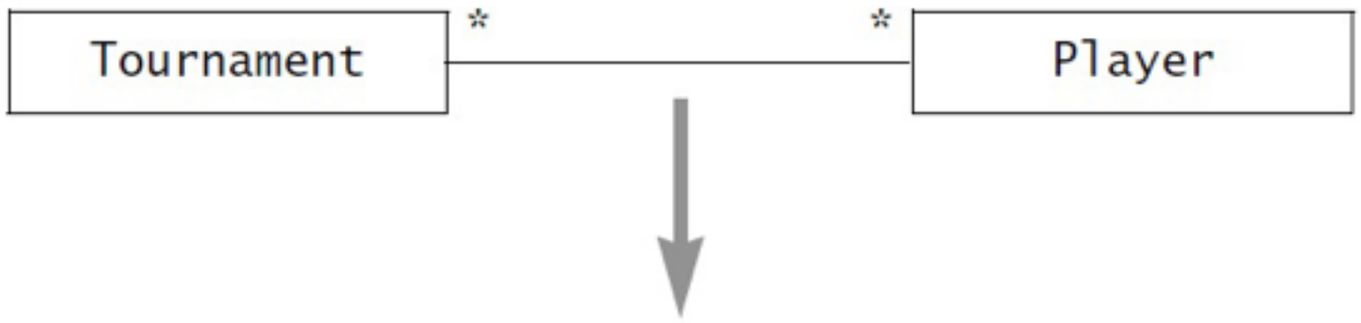


LeagueOwner table

id:long	...

League table

id:long	...	owner:long



Tournament table

id	name	...
23	novice	
24	expert	

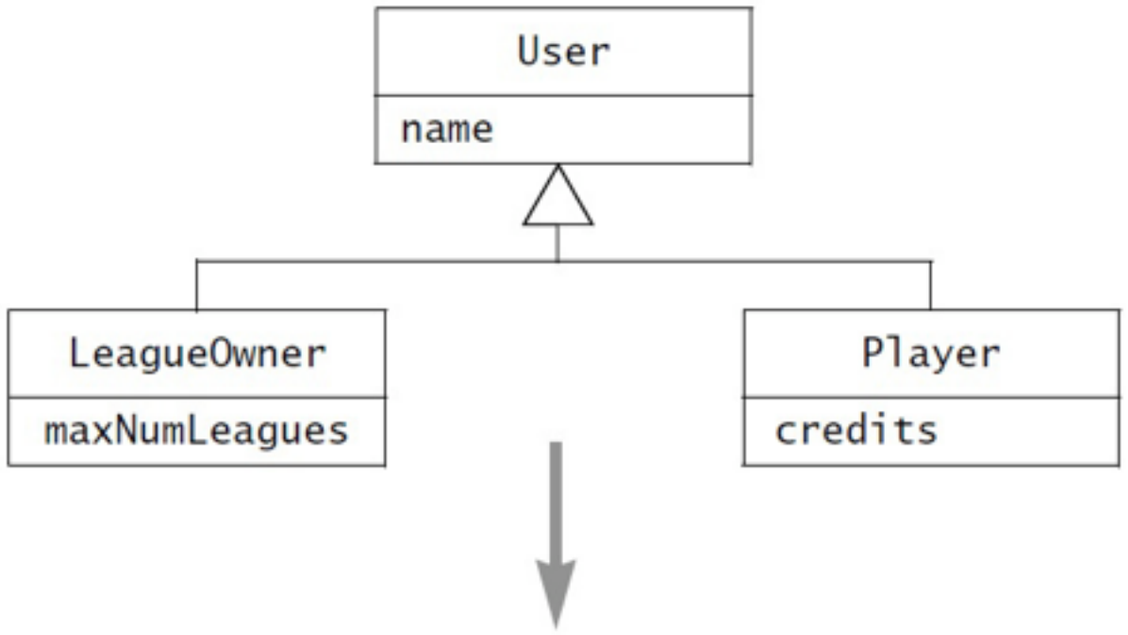
TournamentPlayerAssociation table

tournament	player
23	56
23	79

Player table

id	name	...
56	alice	
79	john	

Mapping inheritance—vertical mapping



User table

id	name	...	role
56	zoe		LeagueOwner
79	john		Player

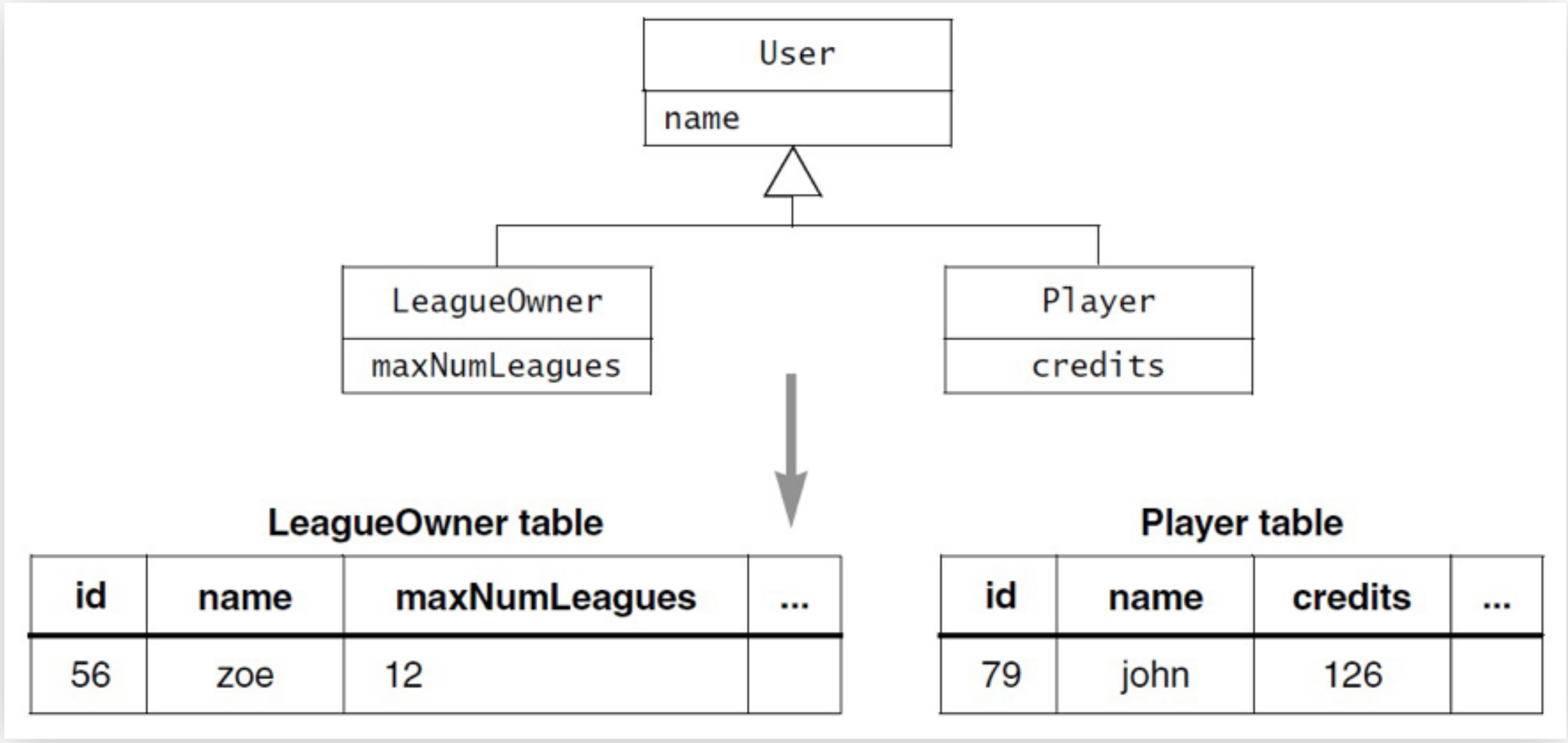
LeagueOwner table

id	maxNumLeagues	...
56	12	

Player table

id	credits	...
79	126	

Mapping inheritance—horizontal mapping



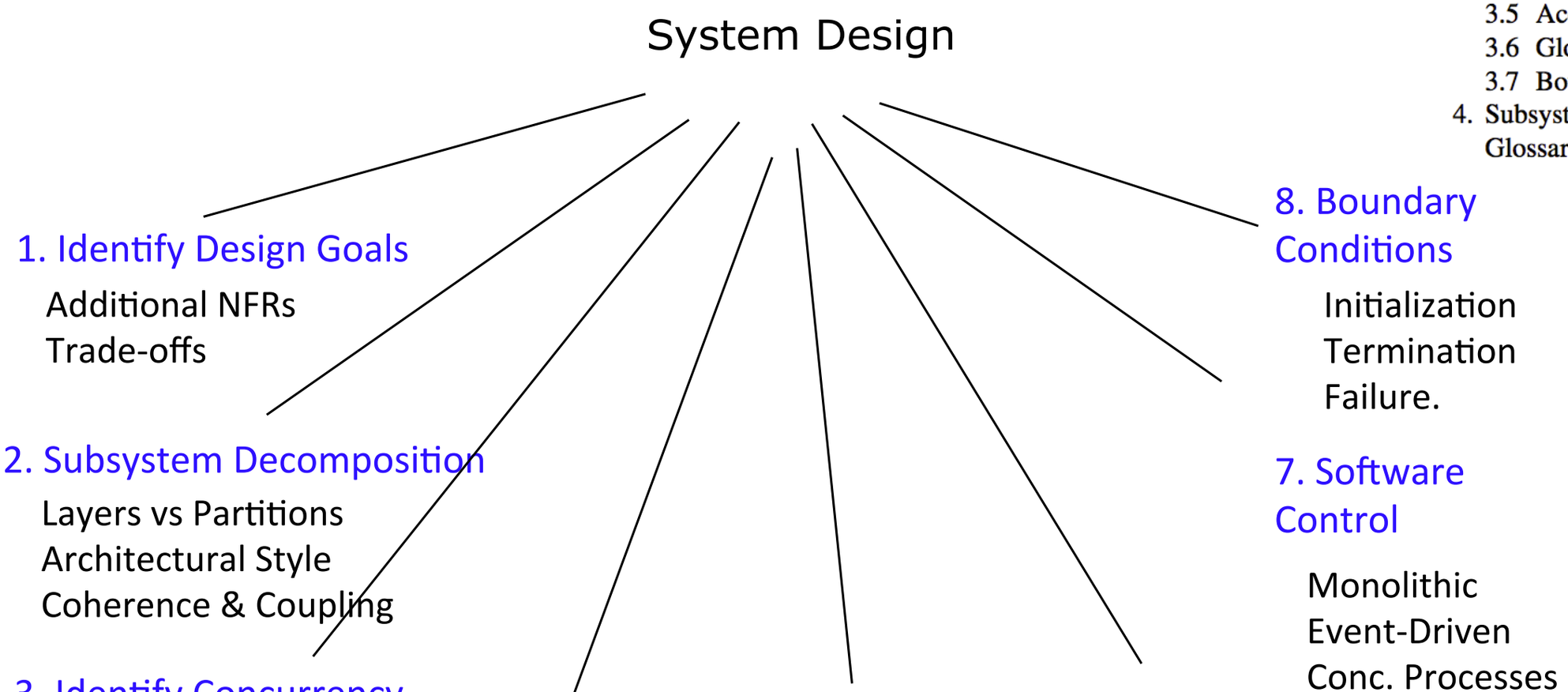
Concluding

Key points

System Design Document

1. Introduction
   - 1.1 Purpose of the system
   - 1.2 Design goals
   - 1.3 Definitions, acronyms, and abbreviations
   - 1.4 References
   - 1.5 Overview
2. Current software architecture
3. Proposed software architecture
   - 3.1 Overview
   - 3.2 Subsystem decomposition
   - 3.3 Hardware/software mapping
   - 3.4 Persistent data management
   - 3.5 Access control and security
   - 3.6 Global software control
   - 3.7 Boundary conditions
4. Subsystem services

Glossary



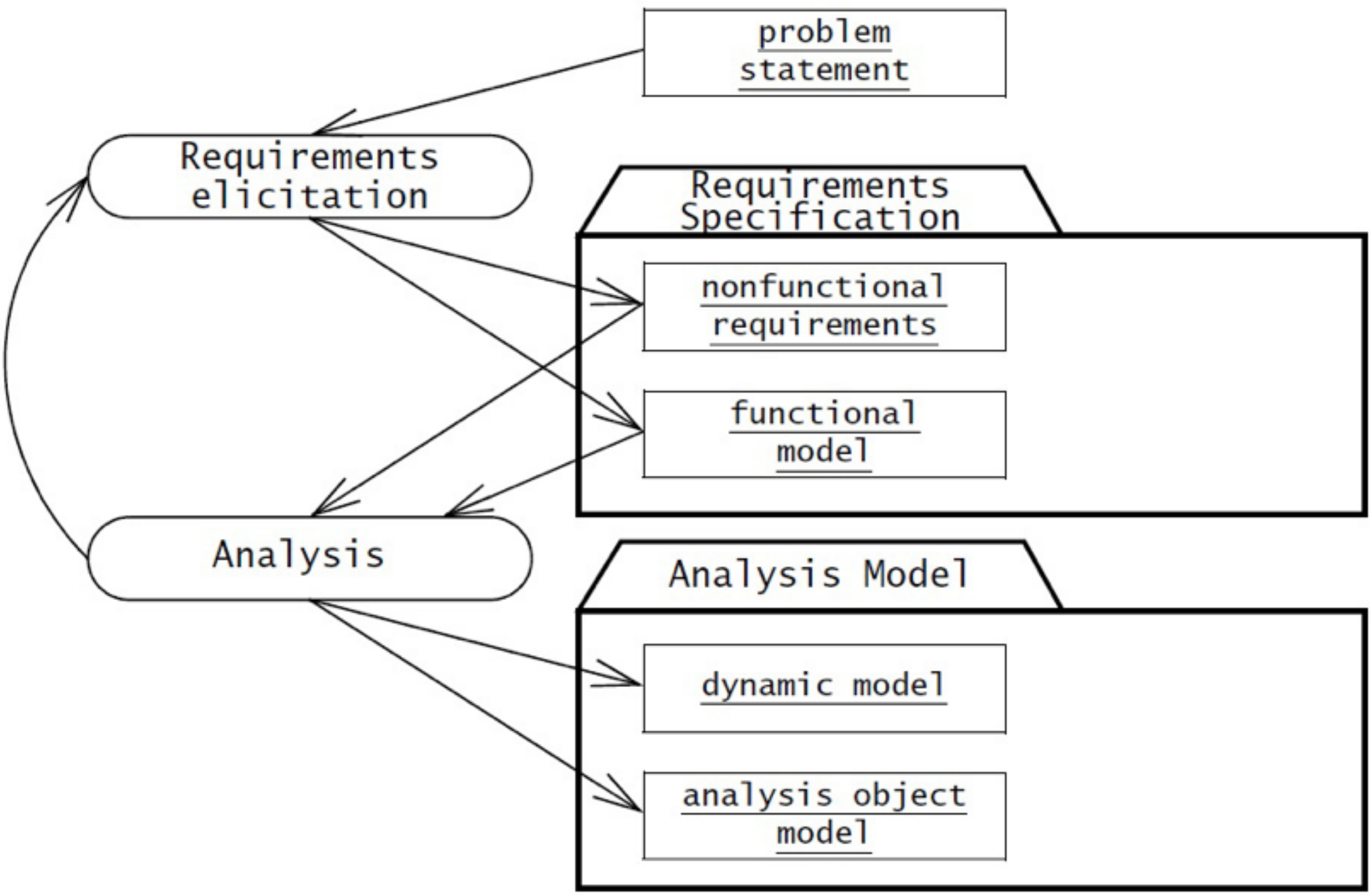
Outline

- Literature
  - [OOSE] ch. 7+8
  - [OOSE] ch. 9+10
  - [SE9] ch. 7

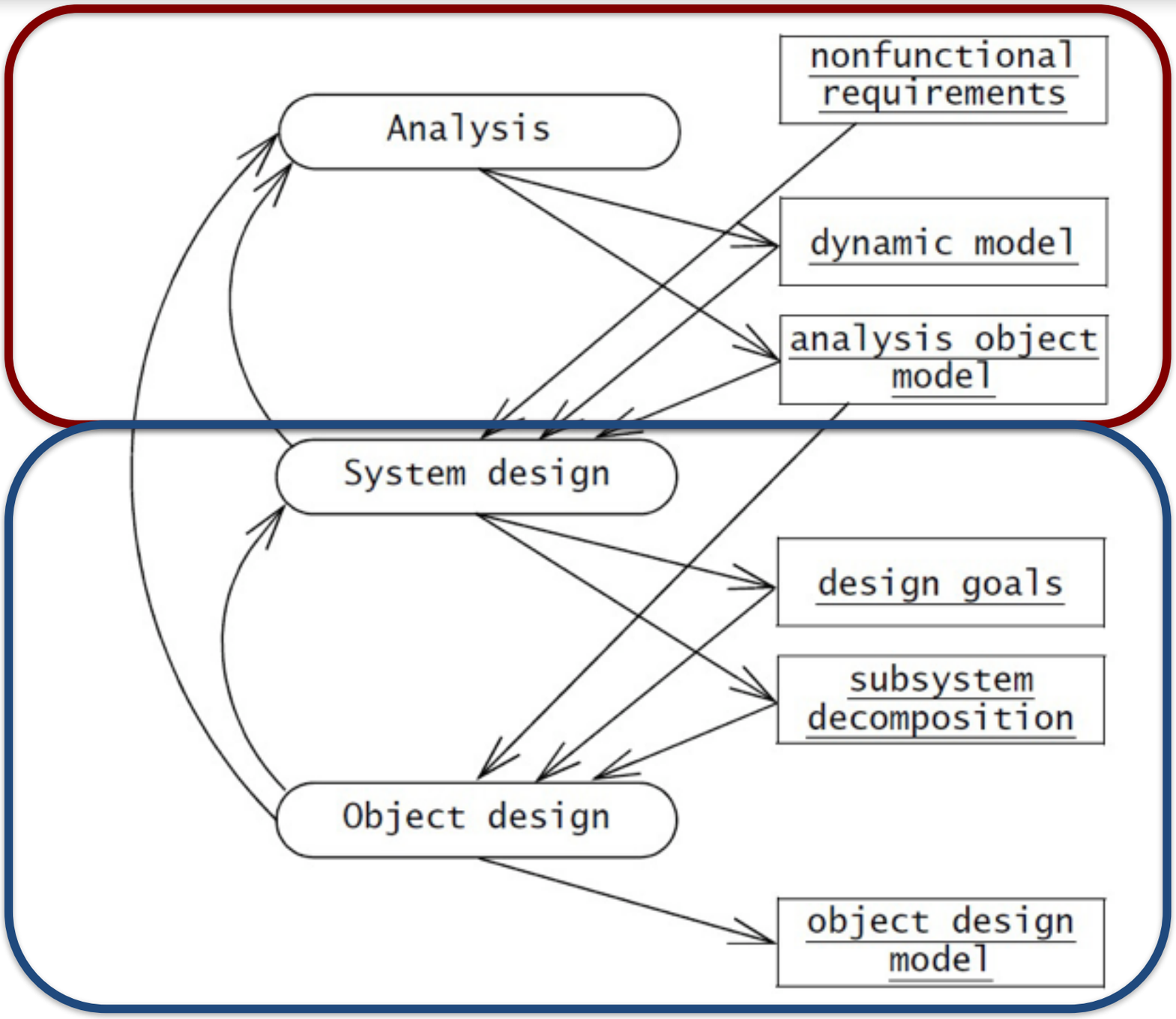
- Topics covered:
  - System Design
  - Object design—Specifying interfaces
  - Object design—Mapping models to code
  - Object design—Mapping OO to RDBMS

Things you should know by now …

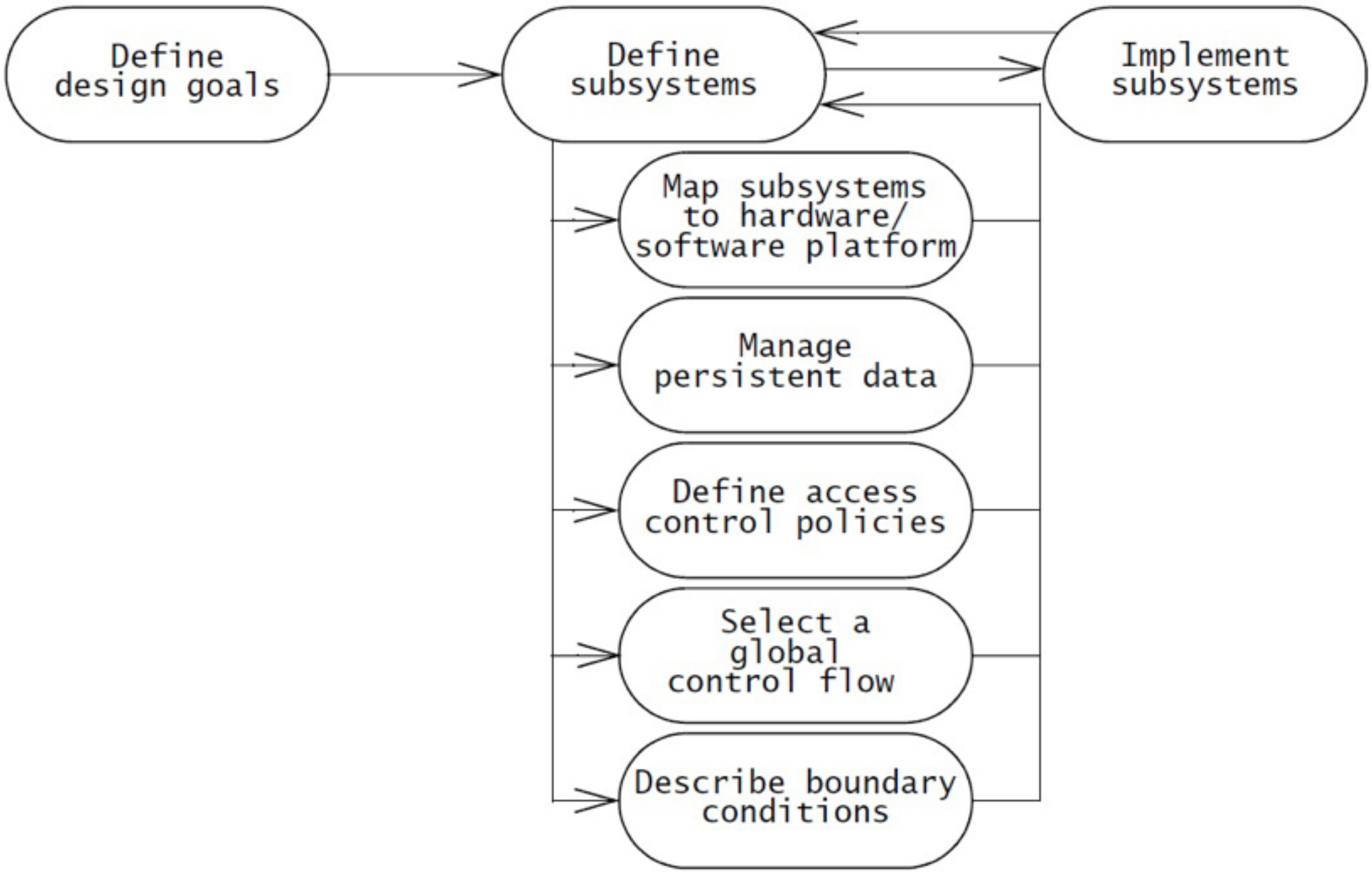
Key points
Requirements elicitation and analysis



Key points
OOA & OOD



Key points
System design

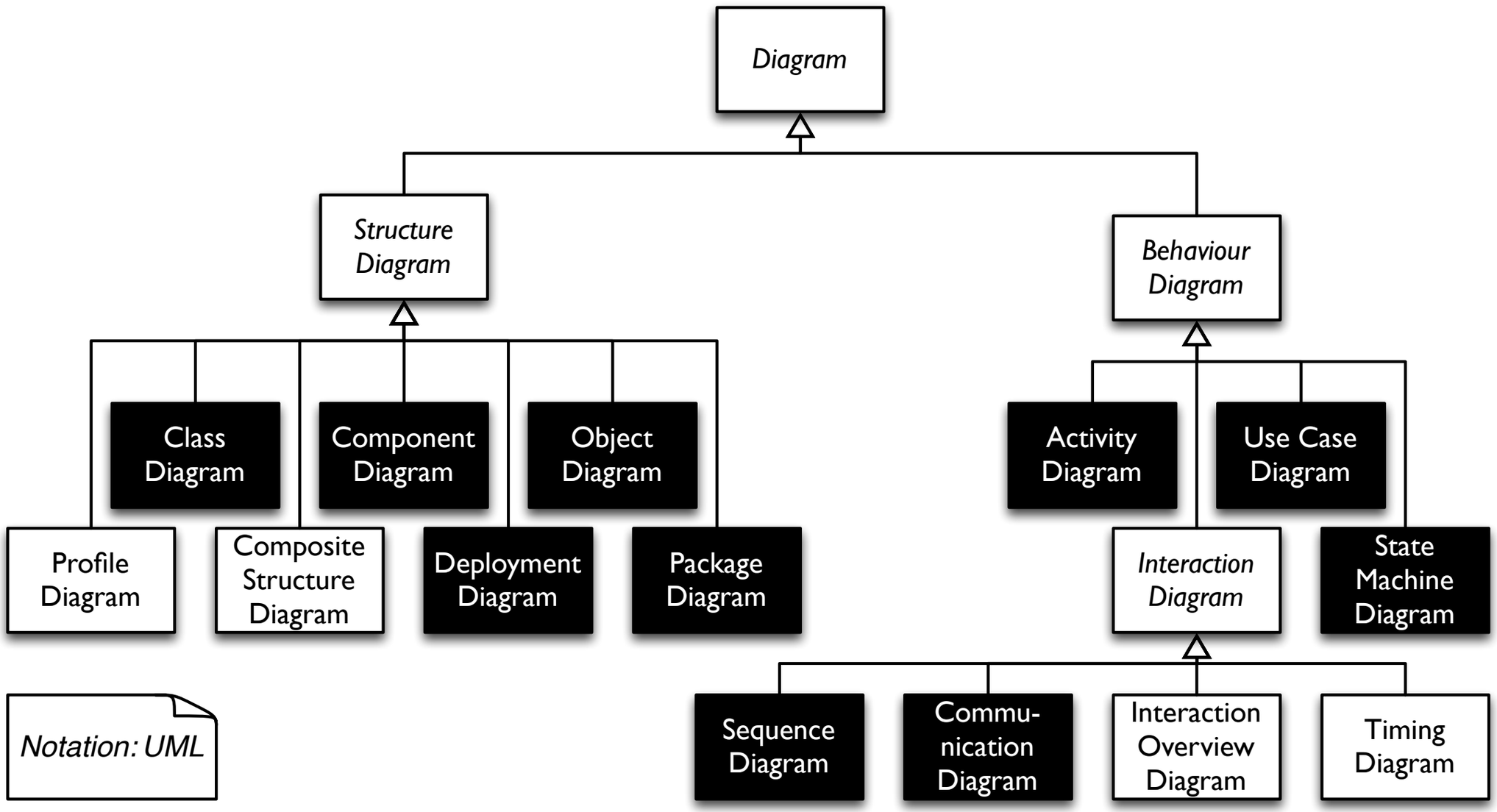


Key points
Patterns

- Architectural patterns or styles
  1. Model-View-Controller (MVC)
  2. Layered
  3. Repository
  4. Client-Server
  5. Pipe & Filter

- Design patters
  1. Observer
  2. Strategy
  3. Factory method
  4. Template method
  5. Adapter
  6. Façade
  7. Composite
  8. Decorator
  9. Bridge
  10. Proxy
  11. Command
  12. Visitor
  13. Chain of responsibility

Key points
UML



Key points
Documentation

- RAD: Requirement Analysis Document
- SDD: System Design Document