System Design

Lecture 09

Overview

Design I: System decomposition

1. Overview of System Design
2. Identify Design Goals
3. Design Initial Subsystem Decomposition

Design II: Refine subsystem decomposition

Design III: Object-level design

Design is Difficult

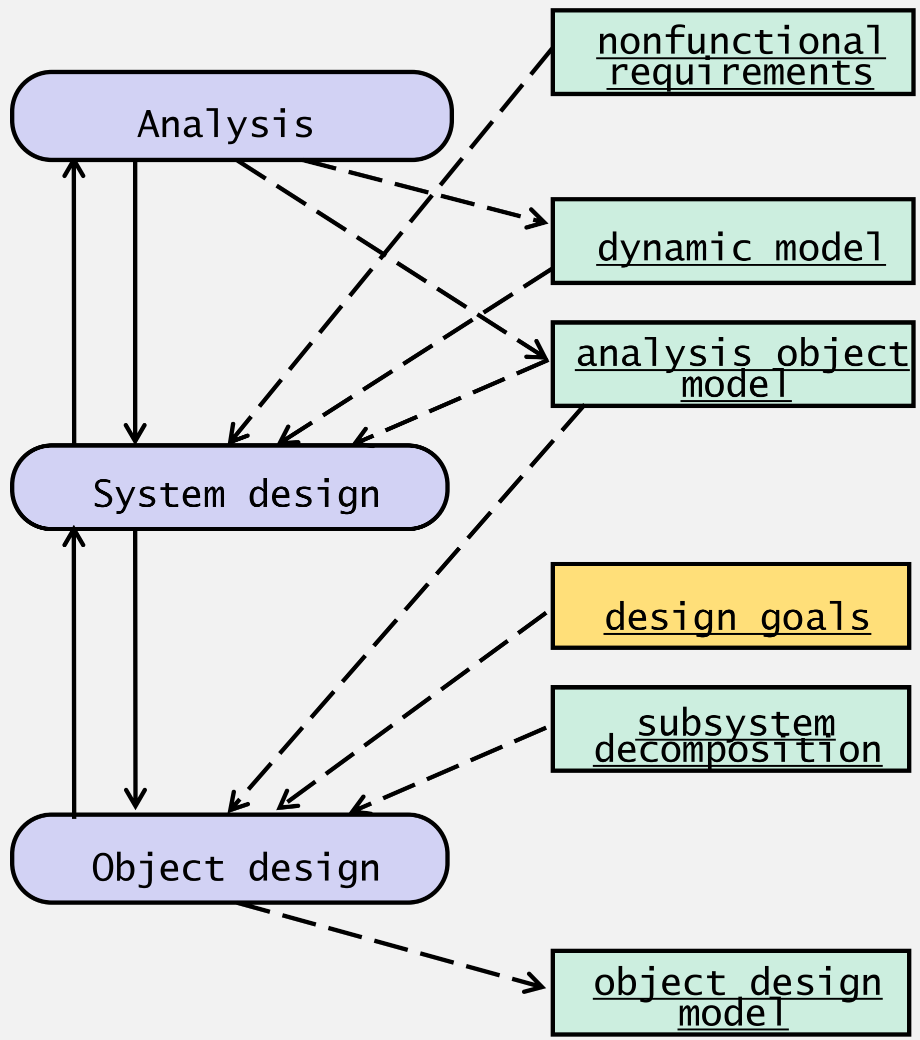
There are two ways of constructing a software design (Tony Hoare):

One way is to make it so simple that there are obviously no deficiencies

The other way is to make it so complicated that there are no obvious deficiencies.”

Sir **Antony Hoare,** *1934
- Quicksort
- Hoare logic for verification

The activities of system design.



The Scope of System Design

Bridge the gap

between a problem and an existing system in a manageable way

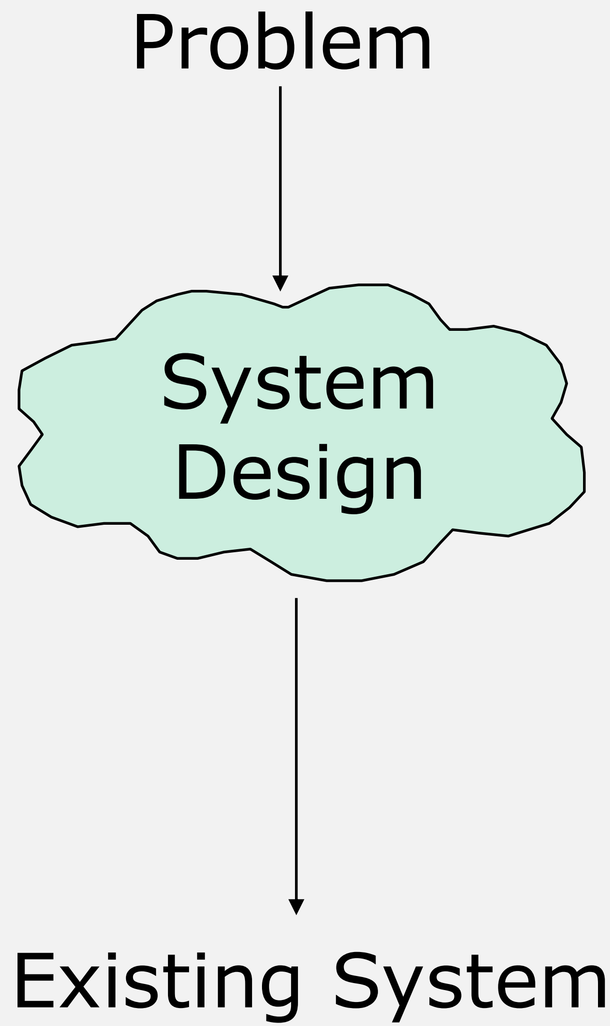
How?

Use Divide & Conquer:

1) Identify design goals

2) Model the new system design as a set of subsystems

3-8) Address the major design goals.



System Design: Eight Issues

System Design

1. Identify Design Goals
   - Additional NFRs
   - Trade-offs

2. Subsystem Decomposition
   - Layers vs Partitions
   - Coherence & Coupling

3. Identify Concurrency
   - Identification of Parallelism (Processes, Threads)

4. Hardware/ Software Mapping
   - Identification of Nodes
   - Special Purpose Systems
   - Buy vs Build
   - Network Connectivity

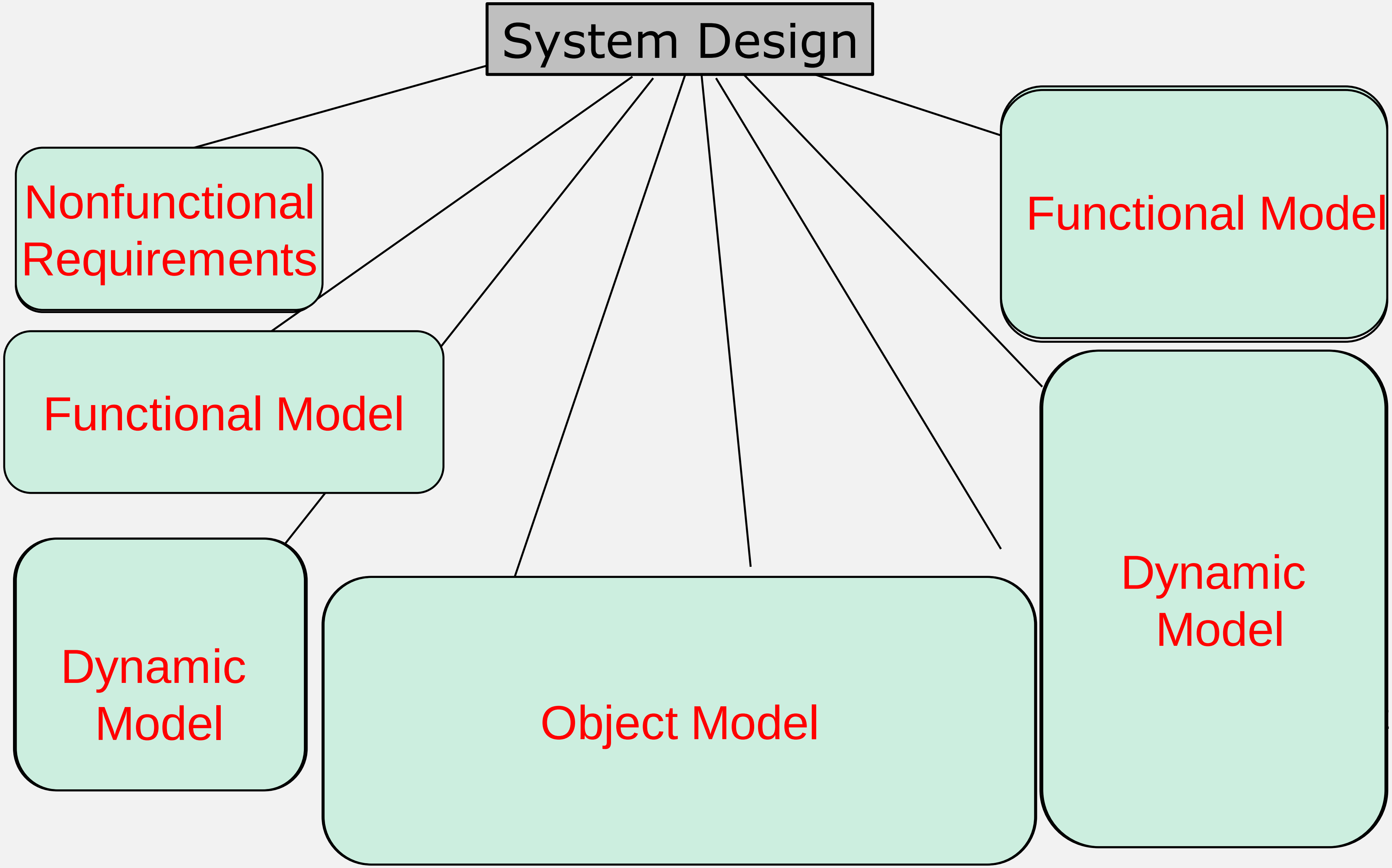
5. Persistent Data Management
   - Storing Persistent Objects
   - Filesystem vs Database

6. Global Resource Handling
   - Access Control
   - ACL vs Capabilities
   - Security

7. Software Control
   - Monolithic
   - Event-Driven
   - Conc. Processes

8. Boundary Conditions
   - Initialization
   - Termination
   - Failure.

Analysis Sources: Requirements and System Model



Why is Design so Difficult?

Analysis: Focuses on the application domain

Design: Focuses on the solution domain

The solution domain is changing very rapidly

- Halftime knowledge in software engineering: About 3-5 years
- Cost of hardware rapidly sinking
- Design knowledge is a moving target

Design window: Time in which design decisions have to be made.

System Design Concepts

- Subsystems
  - Coupling: dependency between two subsystems
  - Cohesion: dependencies within a subsystem
  - Desire LOW coupling and HIGH cohesion
- Refinement
  - Layering
  - Partitions
- Software Architecture Patterns
  - Repository
  - Model/View/Controller
  - Client/Server
  - Peer-To-Peer
  - 3-Tier (4-Tier)
  - Pipe and Filter

Coupling and Cohesion

Goal: Reduction of *complexity while change occurs*

Cohesion measures the dependence among classes

- **High cohesion:** The classes in the subsystem perform similar tasks and are related to each other (via associations)
- **Low cohesion:** Lots of miscellaneous and auxiliary classes, no associations

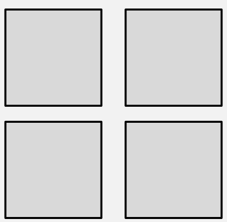
Coupling measures dependencies between subsystems

- **High coupling:** Changes to one subsystem will have high impact on the other subsystem (change of model, massive recompilation, etc.)
- **Low coupling:** A change in one subsystem does not affect any other subsystem

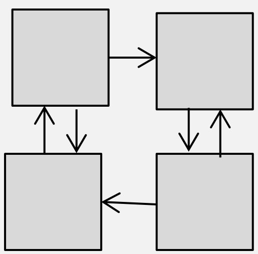
Coupling and Cohesion

Coupling is a measure of the interdependence among components in a computer program.

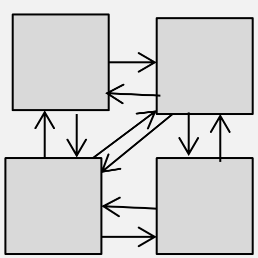
Low coupling is desired



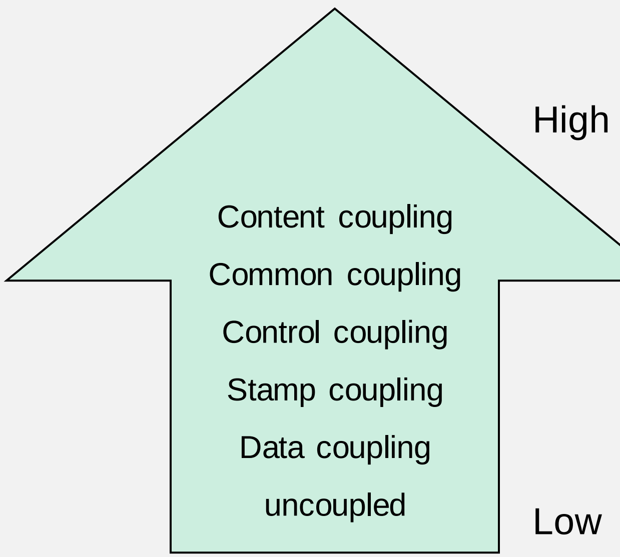
uncoupled



Loosely coupled



Highly coupled



Content coupling

```
void f(){
// block X
…. .
goto Y;
……
// block Z
….
goto X;
…. .
// block Y
……
goto Z;
```

```
class X {
void mX(Y y) {
….
// change fY value
y.fY = …;
…. .
}
}
```

```
class Y {
public int fY;
void mY(){
// change fY value
…. .
// use fY value
}
}
```

Content coupling: Content coupling exists between two modules, if they share code, e.g. a branch from one module into another module.

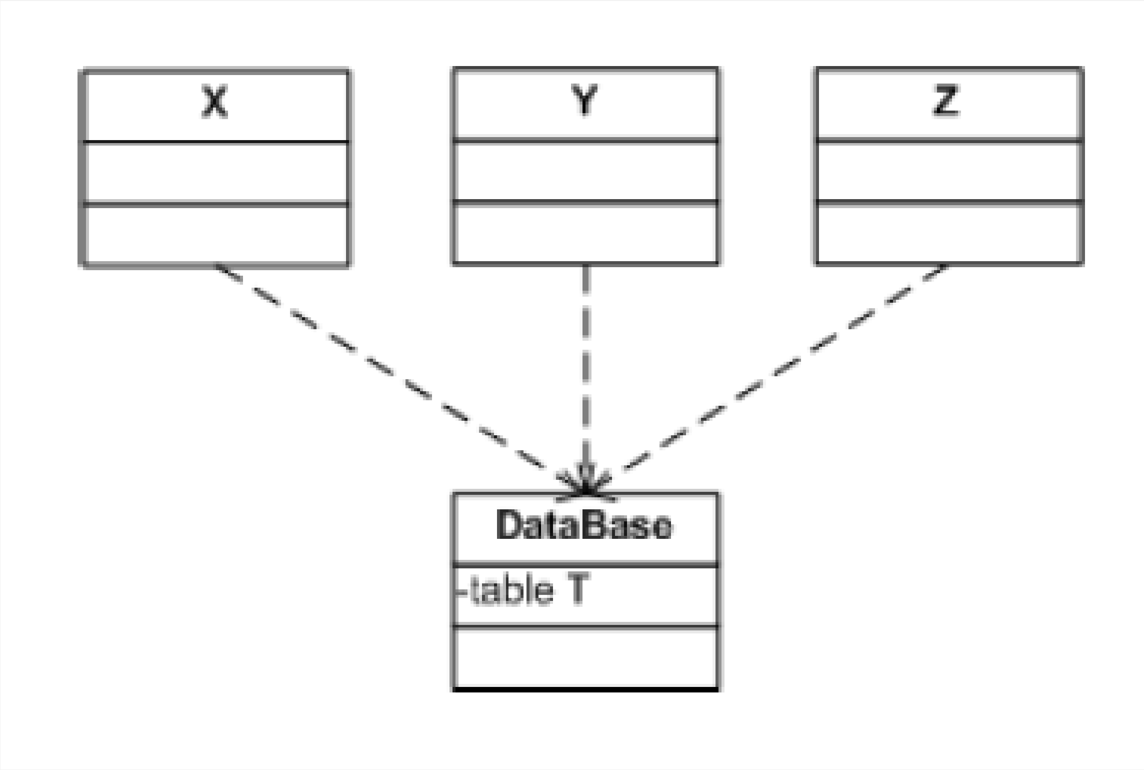
Control coupling

```
class X {
   void mX(Y y) {
   ….
   y.mY(flag);
   …..
   }
}
```

```
class Y {
   void mY(int flag){
      if (flag >= LIMIT_A) {
      …..// block A
      }
      else if (flag < LIMIT_B) {
      …. // block B
      } else { … }
   }
}
```

Control coupling: Control coupling exists between two modules, if data from one module is used to direct the order of instructions execution in another.

Common coupling



Common coupling: Two modules are common coupled, if they share data through some global data items.

Stamp coupling

```
class X {
    void mX(MyList commonRepr) {
    ….
    // accesses only 2nd element
    dataForX = commonRepr.get(2);
    …..
    }
}
```

```
class Y {
    void mY(MyList commonRepr){
    // accesses only 4th element
    dataForY = commonRepr.get(4);
    }
}
```

Stamp coupling: Two modules are stamp coupled, if they communicate using a composite data.

Data coupling

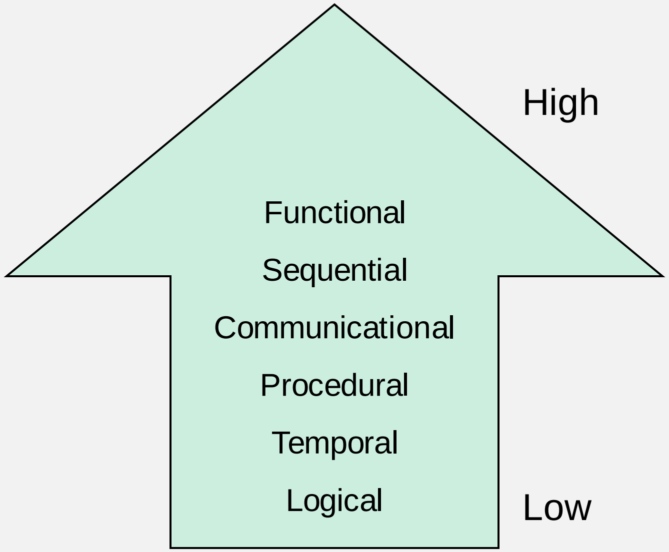
```
class X {
   void mX() {
   ….
   Y yObject;
   …..
   yObject.mY(iVar, dVar, sVar);
   }
}
```

Data coupling: Two modules are data coupled, if they communicate through parameter.

```
class Y {
   void mY(int iPar, double dPar, String sPar){
   // …..
   }
}
```

Cohesion

Cohesion is a measure of the strength of association of the elements within a component.



Coincidental Cohesion

```
class GodModule {
    void accessPrinter() {
        // many variables
        // many Lines of Code
    }
    void drawChart() {
        // many variables
        // many Lines of Code
    }
    void connectToDatabase(){
       // many variables
       // many Lines of Code
    }
}
```

Coincidental cohesion: A module is said to have coincidental cohesion, if it performs a set of tasks that relate to each other very loosely, if at all.

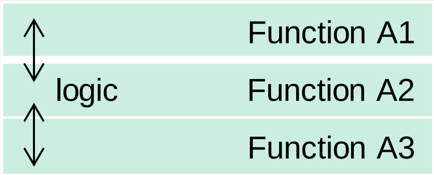
Function A	
Function B	Function C
Function D	Function E

Parts unrelated

Logical Cohesion

```
class InputReader {
    char[] getInput(int flag) {
        if(flag == 1)
        getInputFromFile();
        else
        if(flag == 2)
        getInputFromDatabase();
        else …….
    }
    char[] getInputFromFile() {
        // many variables
        // many Lines of Code
    }
    char[] getInputFromDatabase() {
        // many variables
        // many Lines of Code
    }
    char[] getInputFromKeyboard() {…}
}
```

Logical cohesion: A module is said to be logically cohesive, if all elements of the module perform similar operations.



Temporal Cohesion

```
class Initializer {
    void initAll(){
        initDatabase();
        initNetwork();
    }
    void initNetwork(){
        // many variables
        // many Lines of Code
    }
    void initDatabase(){
        // many variables
        // many Lines of Code
    }
}
```

Temporal cohesion: When a module contains functions that are related by the fact that all the functions must be executed in the same time span

Time T0
Time T0 + X
Time T0 + 2X

Related by time

Procedural Cohesion

```
class Initializer {
    void initAll(){
        initDatabase();
        initNetwork();
    }
    void initNetwork(){
        // many variables
        // many Lines of Code
    }
    void initDatabase(){
        // many variables
        // many Lines of Code
    }
}
```

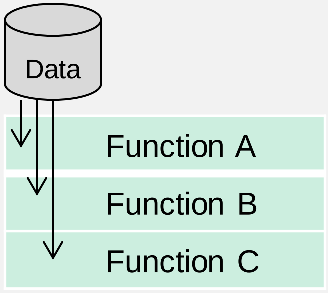
Procedural cohesion: A module is said to possess procedural cohesion, if the set of functions of the module are all part of a procedure (algorithm) in which certain sequence of steps have to be carried out for achieving an objective.

Function A
Function B
Function C

Communication Cohesion

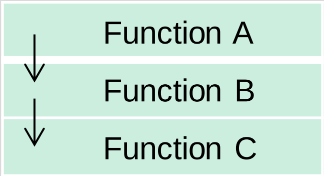
```
class Person {
    Table[] personalData;

    getName(){return personalData[0];}
    getCustomerInfo(){return personalData[1];}
    getEmployeeInfo(){return personalData[2];}
}
```

Communicational cohesion: A module is said to have communicational cohesion, if all functions of the module refer to or update the same data structure



Sequential Cohesion

```
class TravelAgent {
    void reserveTrip(location){
        date = reserveTicket(location);
        address = reserveHotel(date);
        taxi = reserveTaxi(address, date);
    }
    String reserveTicket(String location){
        // many variables
        // many Lines of Code
    }
    String reserveHotel(String date){
        // many variables
        // many Lines of Code
    }
    int reserveTaxi(String address, String date){
        // many variables
        // many Lines of Code
    }
}
```



Sequential cohesion: A module is said to possess sequential cohesion, if the elements of a module form the parts of sequence, where the output from one element of the sequence is input to the next.

Functional Cohesion

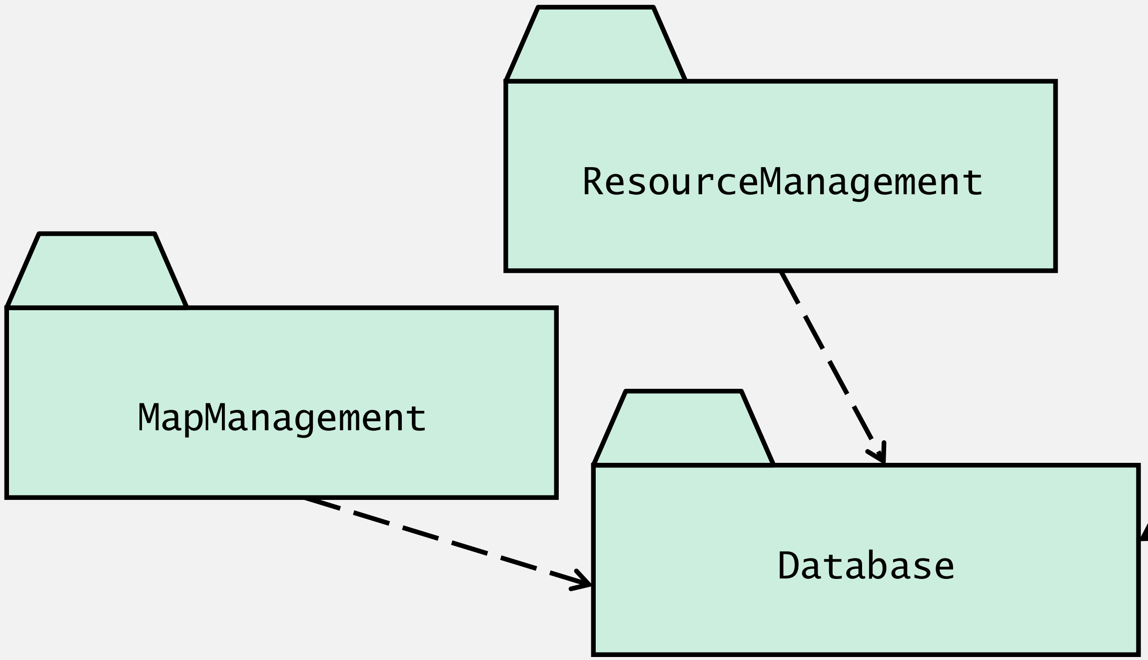
```
class TravelAgent {
    AirlineAgent airlineCompany;
    HotelAgent hotelCompany;
    TaxiAgent taxiCompany;
        void scheduleTrip(location){
        date = airlineCompany.reserve(location);
        hotel = hotelCompany.reserve(dates);
        taxi = taxiCompany.reserve(hotel);
        }
    }

    class AirlineAgent{
        String reserve(String location){
        // …
        }
        }
        class HotelAgent{
        String reserve(String date){
        // …
        }
}
………
```

Functional cohesion: Functional cohesion is said to exist, if different elements of a module cooperate to achieve a single function.

Function A: part 1
Function A : part 2
Function A : part 3

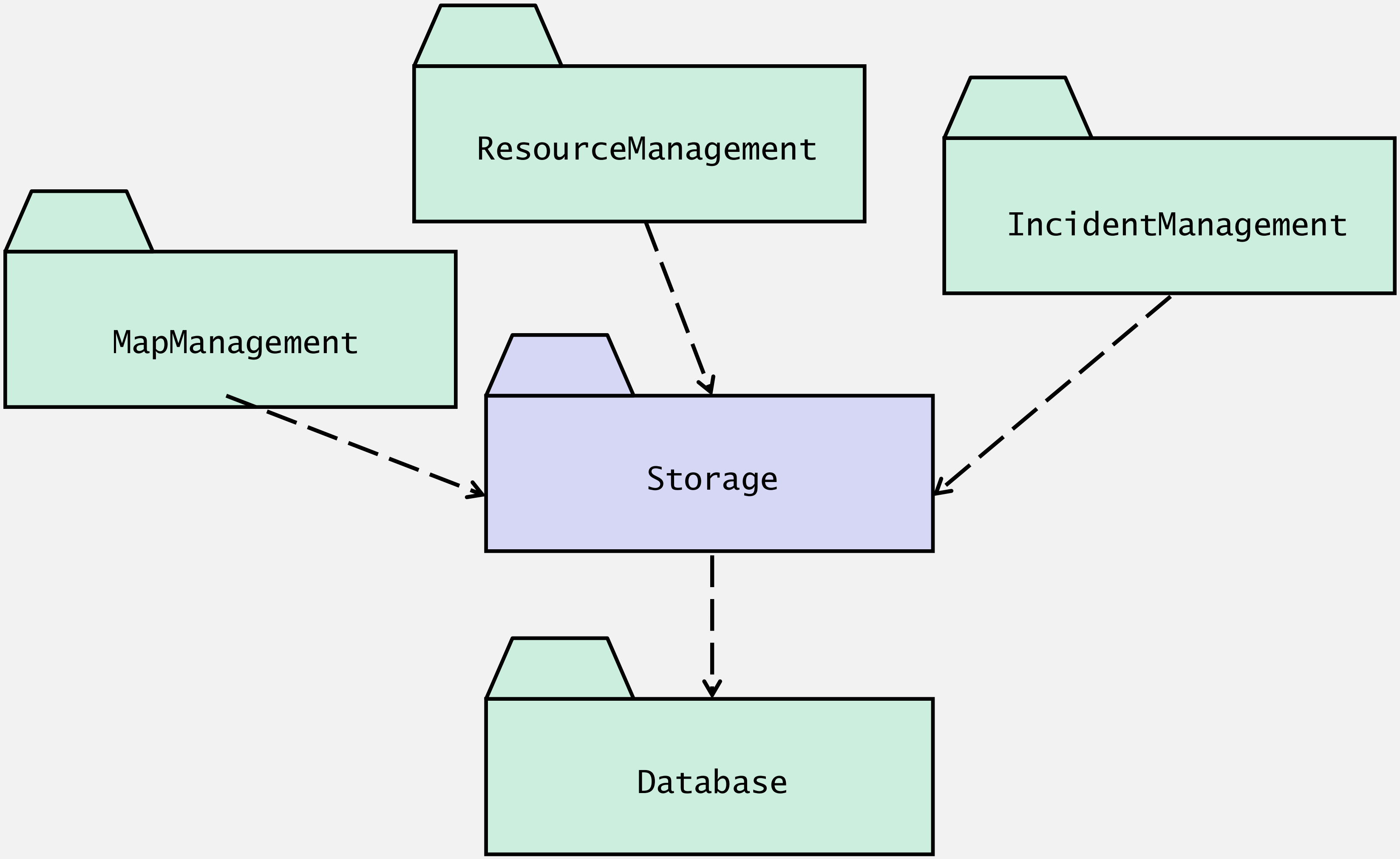
High Coupling

Alternative 1: Direct access to the Database subsystem subject to change

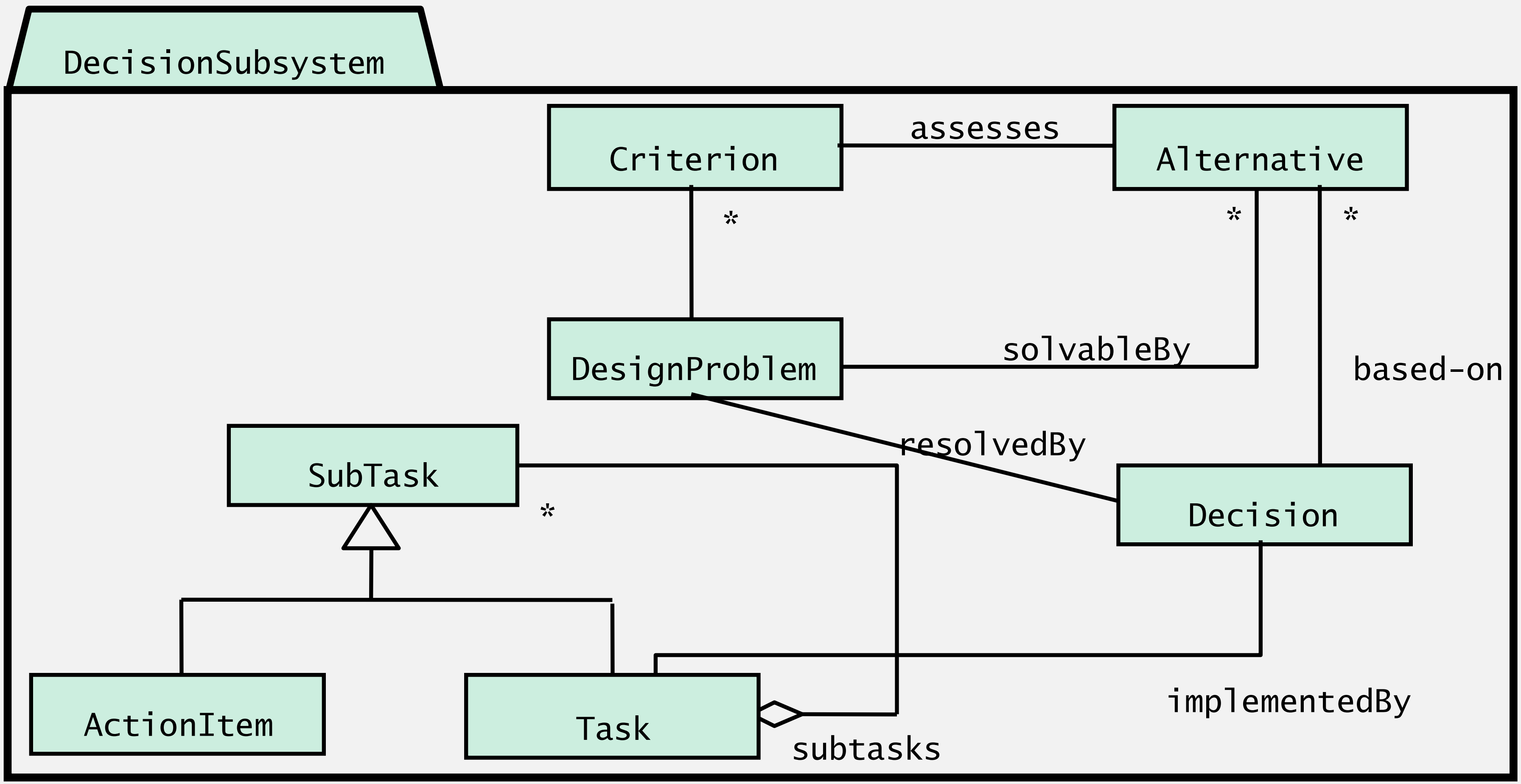


Coupling Reduced

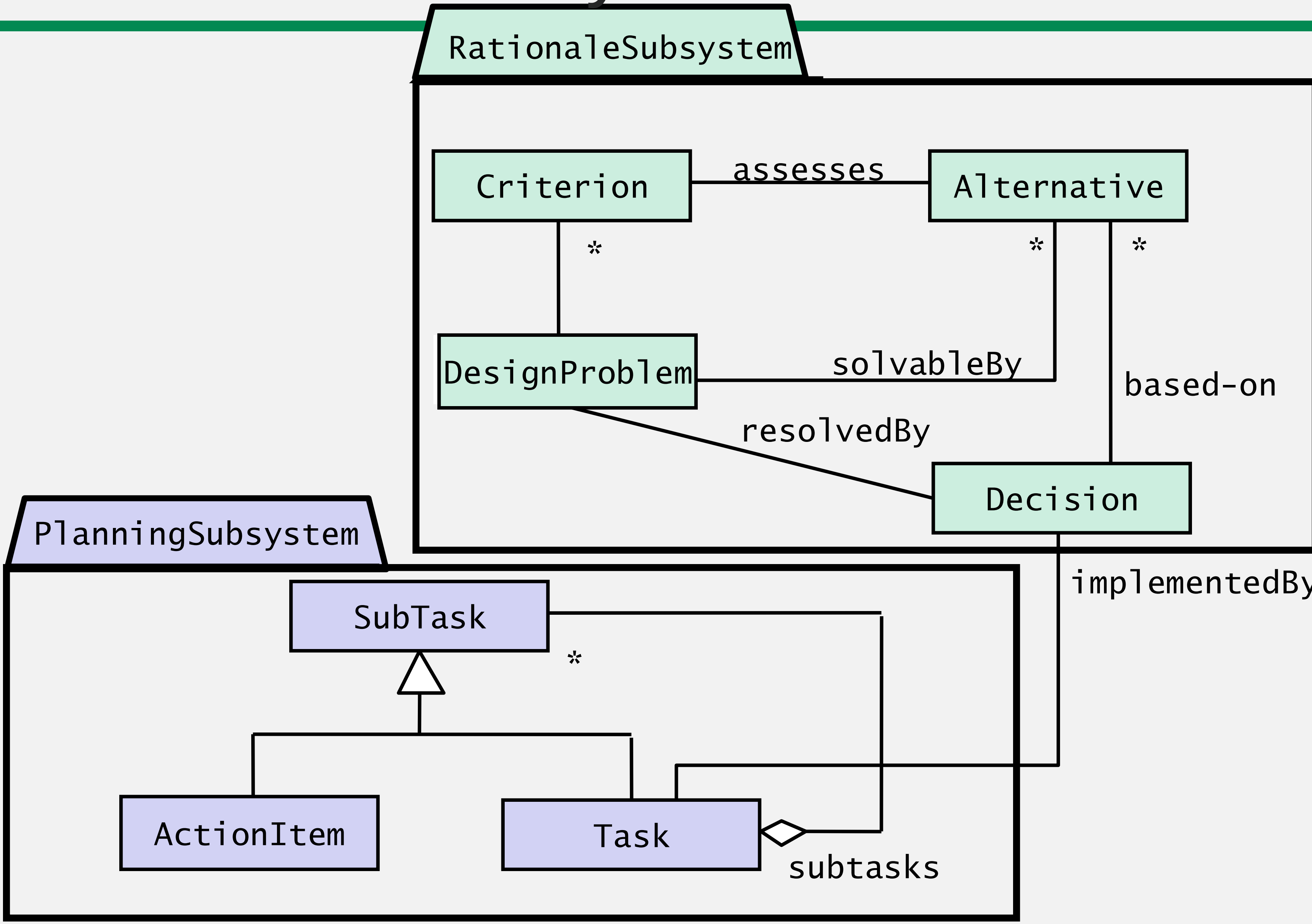
Alternative 2: Storage subsystem more stable



Decision tracking system: Low Cohesion



Better Cohesion obtained by dividing 1 subsyetm into 2



Partitions and Layers

Partitioning and layering are techniques to achieve low coupling.

A large system is usually decomposed into subsystems using both layers and partitions.

A *partition* vertically divides system into several independent (or weakly-coupled) subsystems that provide services *on the same level of abstraction*.

A ***layer*** is a subsystem that provides services to a layer *at a higher level of abstraction*

- A layer can only depend on lower layers
- A layer has no knowledge of higher layers

Properties of Subsystems: Layers and Partitions

A layer is a subsystem that provides a service to another subsystem with the following restrictions:

- A layer only depends on services from lower layers
- A layer has no knowledge of higher layers

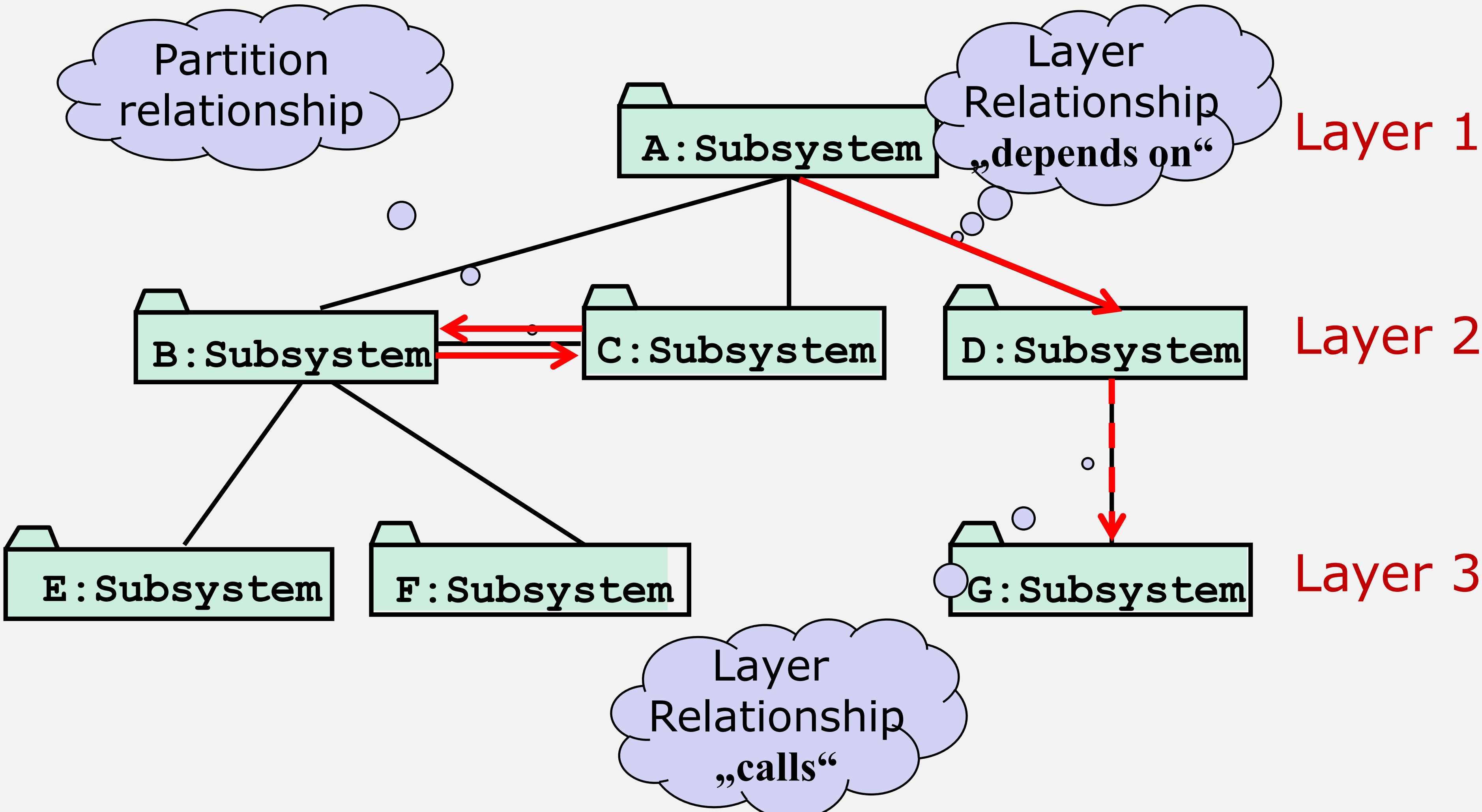
A layer can be divided horizontally into several independent subsystems called partitions

- Partitions provide services to other partitions on the same layer
- Partitions are also called "weakly coupled" subsystems.

Relationships between Subsystems

- Two major types of Layer relationships
  - Layer A "depends on" Layer B (compile time dependency)
    - Example: Build dependencies (make, ant, maven)
  - Layer A "calls" Layer B (runtime dependency)
    - Example: A web browser calls a web server
    - Can the client and server layers run on the same machine?
      - Yes, they are layers, not processor nodes
      - Mapping of layers to processors is decided during the Software/hardware mapping!
- Partition relationship
  - The subsystems have mutual knowledge about each other
    - A calls services in B; B calls services in A (Peer-to-Peer)

Example of a Subsystem Decomposition



How the Analysis Models Influence System Design

Nonfunctional requirements =>

Activity 1: Design Goals Definition

Functional model =>

Activity 2: System decomposition (Selection of subsystems based on functional requirements, cohesion, and coupling)

Object model =>

Activity 4: Hardware/software mapping

Activity 5: Persistent data management

Dynamic model =>

Activity 3: Concurrency

Activity 6: Global resource handling

Activity 7: Software control

Subsystem Decomposition

Activity 8: Boundary conditions

Sharpen the Design Goals

Location-based input

- Input depends on user location
- Input depends on the direction where the user looks (“egocentric systems”)

Multi-modal input

- The input comes from more than one input device

Dynamic connection

- Contracts are only valid for a limited time

Is there a possibility of further generalizations?

Example: location can be seen as a special case of context

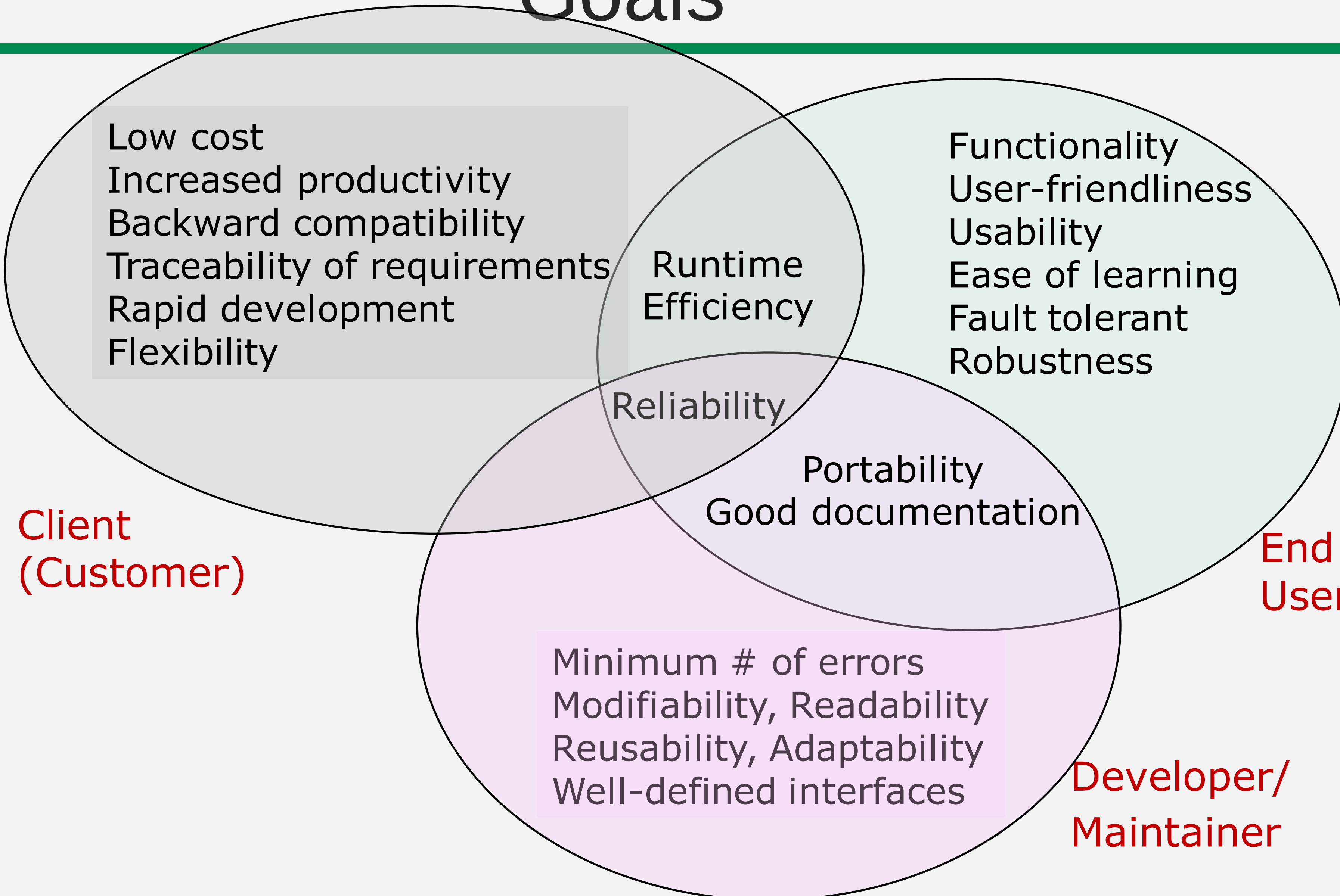
- User preference is part of the context
- Interpretation of commands depends on context

Example of Design Goals

- Reliability
- Modifiability
- Maintainability
- Understandability
- Adaptability
- Reusability
- Efficiency
- Portability
- Traceability of requirements
- Fault tolerance
- Backward-compatibility
- Cost-effectiveness
- Robustness
- High-performance

- Good documentation
- Well-defined interfaces
- User-friendliness
- Reuse of components
- Rapid development
- Minimum number of errors
- Readability
- Ease of learning
- Ease of remembering
- Ease of use
- Increased productivity
- Low-cost
- Flexibility

Stakeholders have different Design Goals



Typical Design Trade-offs

Functionality v. Usability

Cost v. Robustness

Efficiency v. Portability

Rapid development v. Functionality

Cost v. Reusability

Backward Compatibility v. Readability

Summary

- System Design
  - Reduces the gap between requirements and the (virtual) machine
  - Decomposes the overall system into manageable parts
- Design Goals Definition
  - Describes and prioritizes the qualities that are important for the system
  - Defines the value system against which options are evaluated

Thank You