Analysis, Design, and Software Architecture (BDSA)

Paolo Tell

Architectural and Object Oriented Design

IT University
of Copenhagen

- 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 (I)

- 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 [SE9]

Architecture design

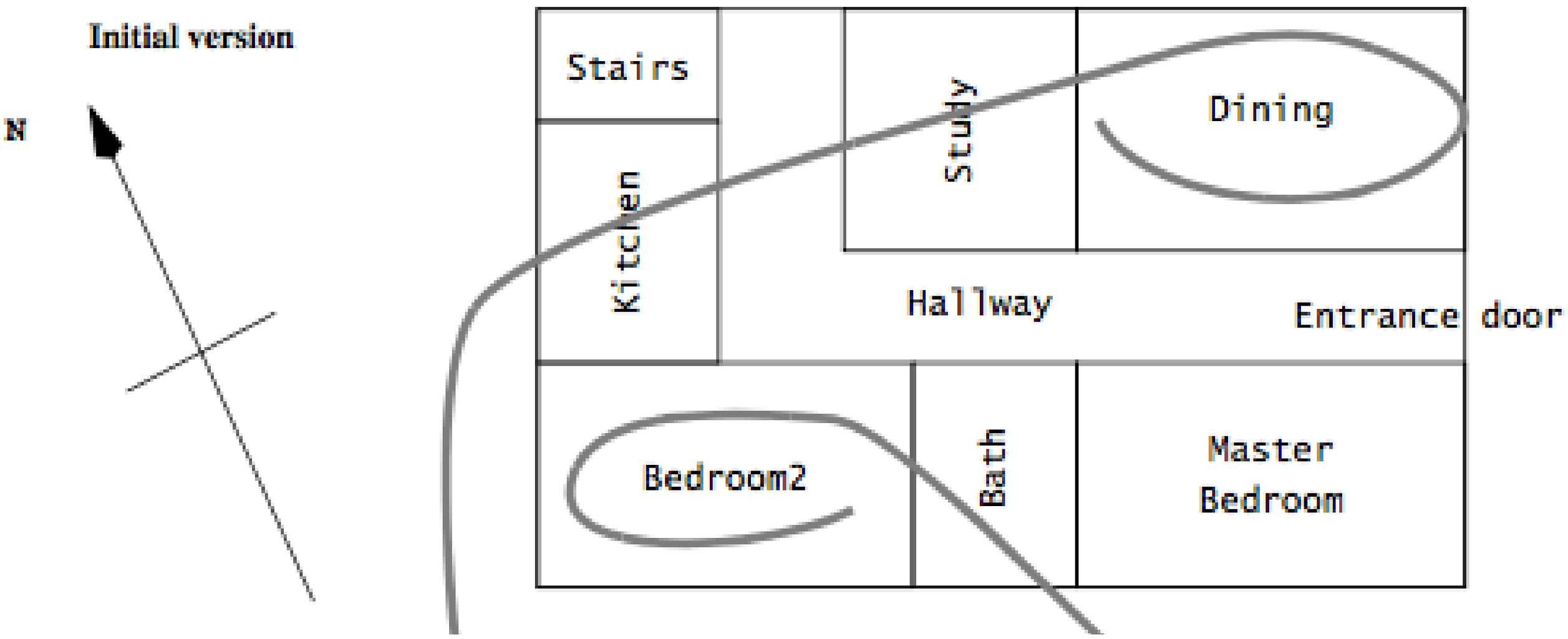
Topics covered

- Architectural design decisions
  - introduce architectural design and to discuss its importance
- Architectural views
  - an architecture is described/modelled from different viewpoints
- Architectural patterns
  - the classic architectures
- Application architectures
  - more detailed architecture

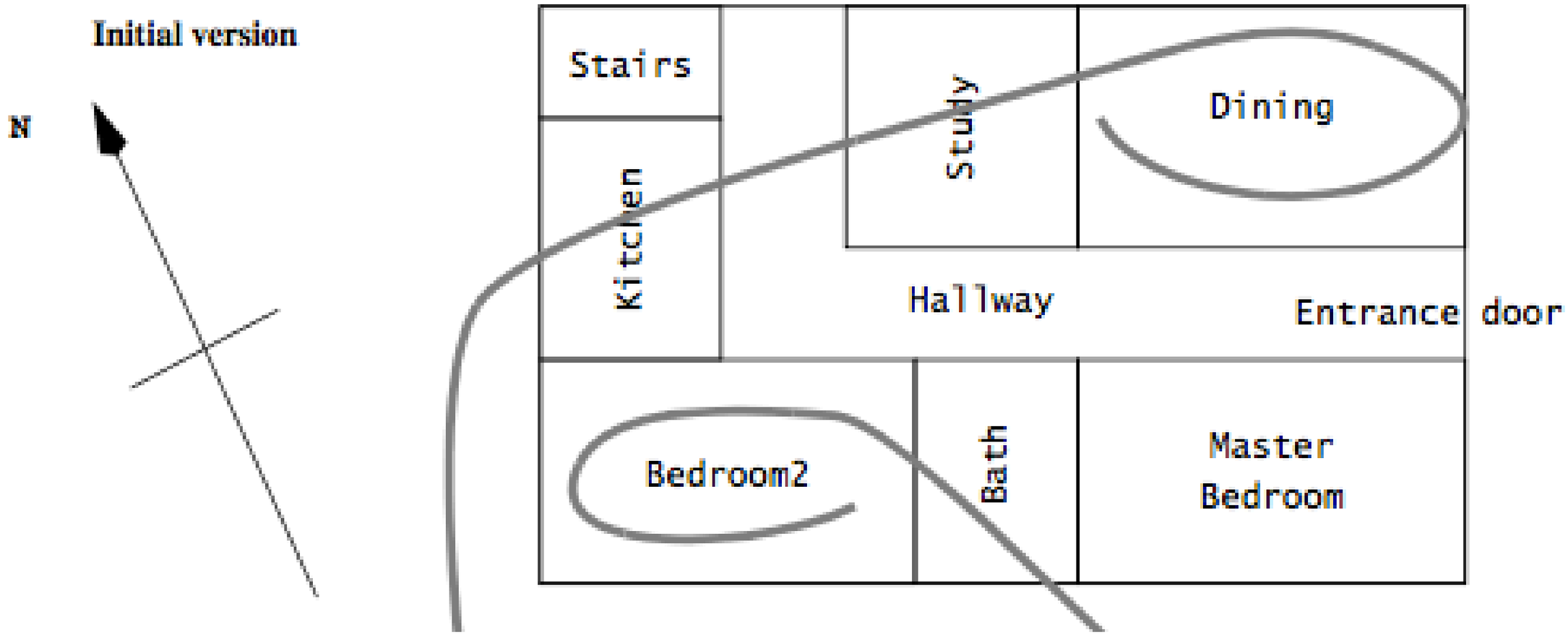
An example from architecture

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 minimized.
3. The use of daylight should be maximized.

Example


Initial version
N
Stairs
Study
Dining
Kitchen
Hallway
Entrance door
Bedroom2
Bath
Master
Bedroom

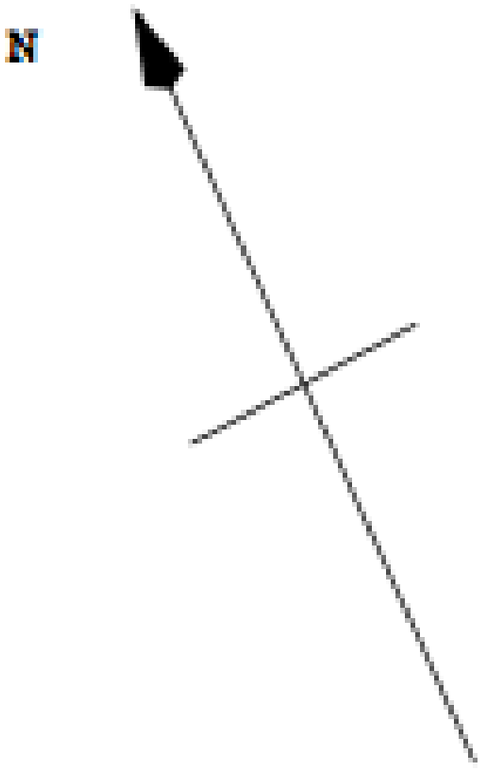
Example



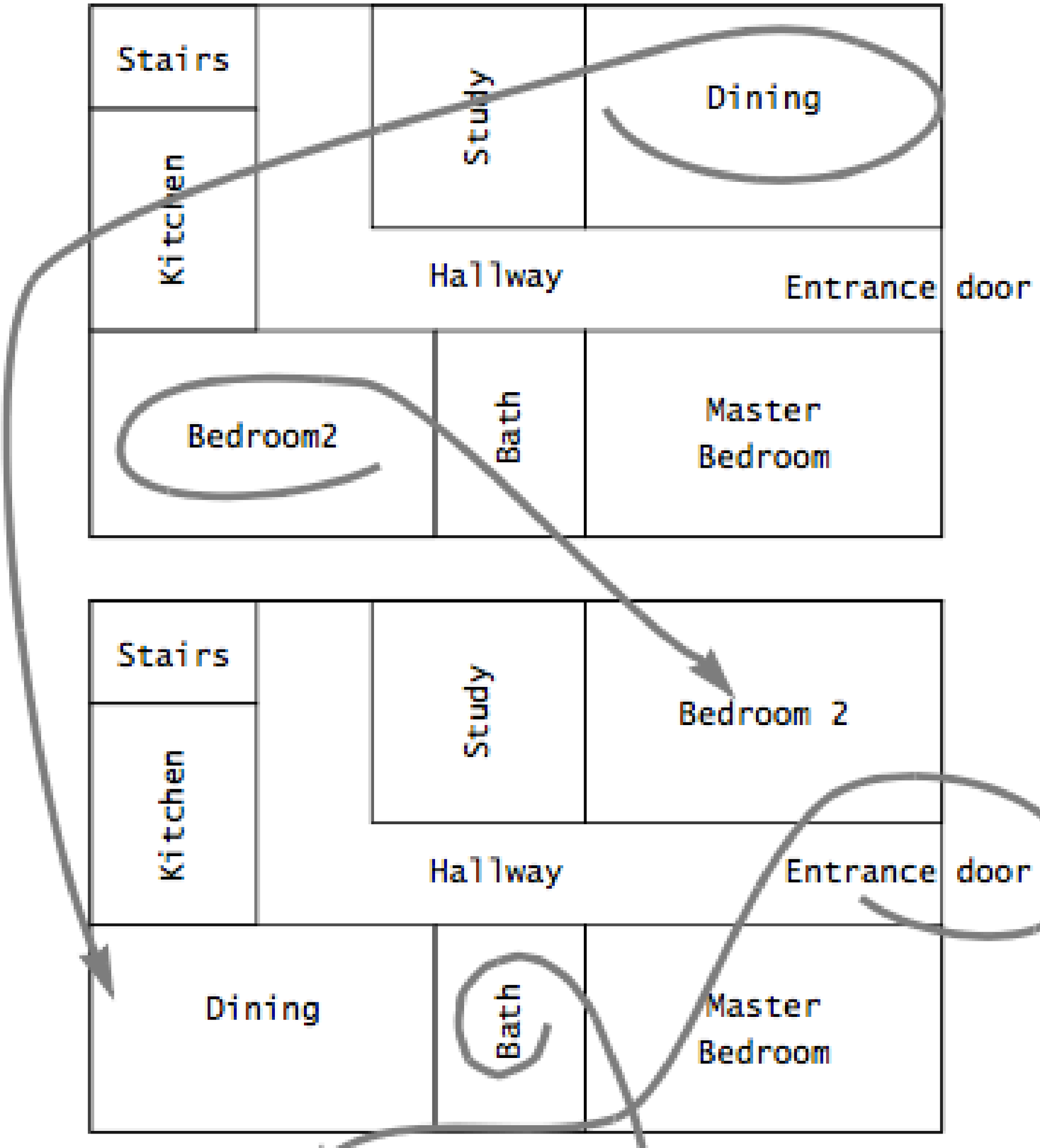
1. This house should have two bedrooms, a study, a kitchen, and a living room area.
2. The overall distance the occupants walk every day should be minimized.
3. The use of daylight should be maximized.

Initial version

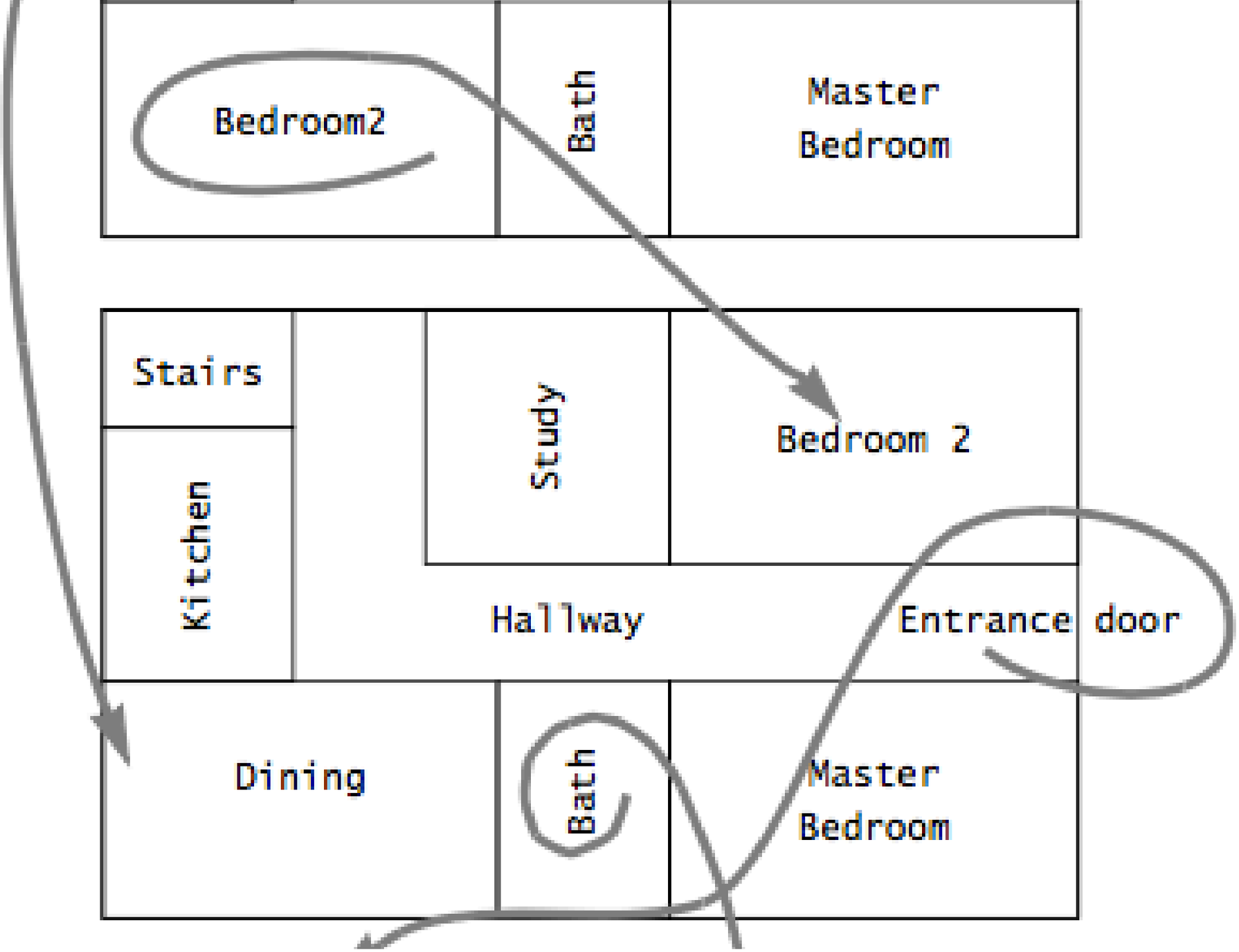
N



Second version

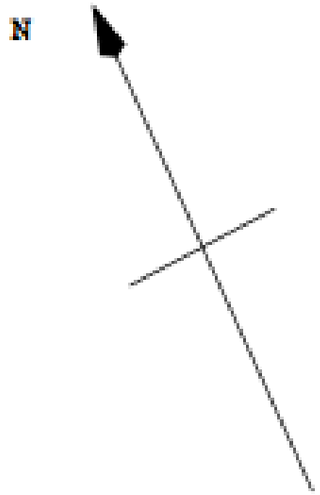


Second version

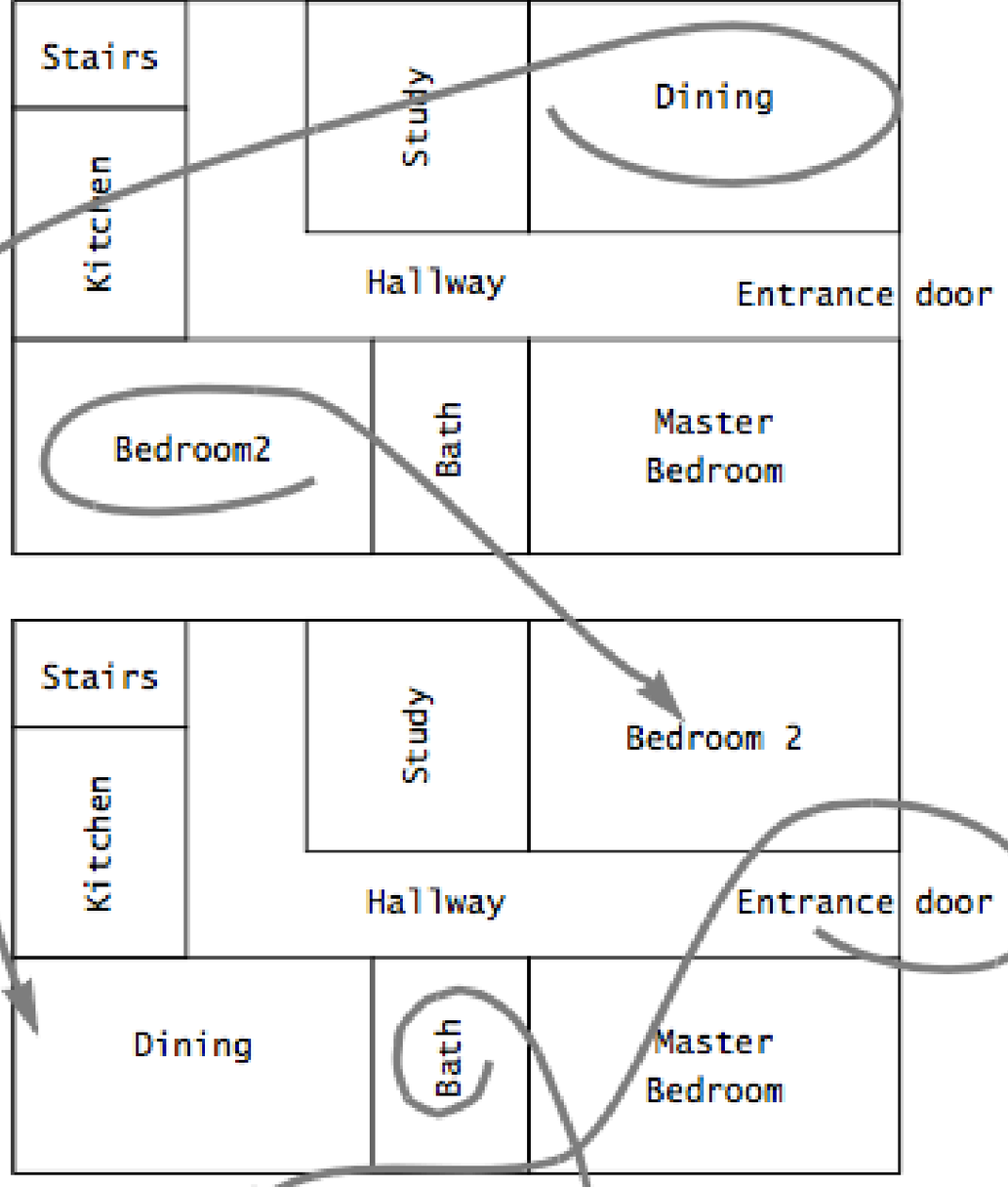


1. This house should have two bedrooms, a study, a kitchen, and a living room area.
2. The overall distance the occupants walk every day should be minimized.
3. <u>The use of daylight should be maximized.</u>

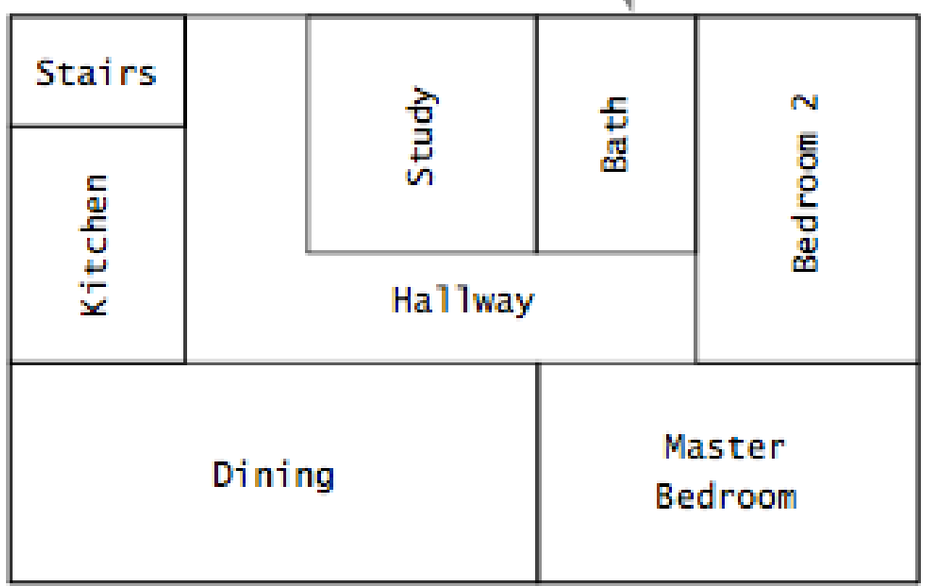
Initial version



Second version



Third version



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.

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
    - 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.

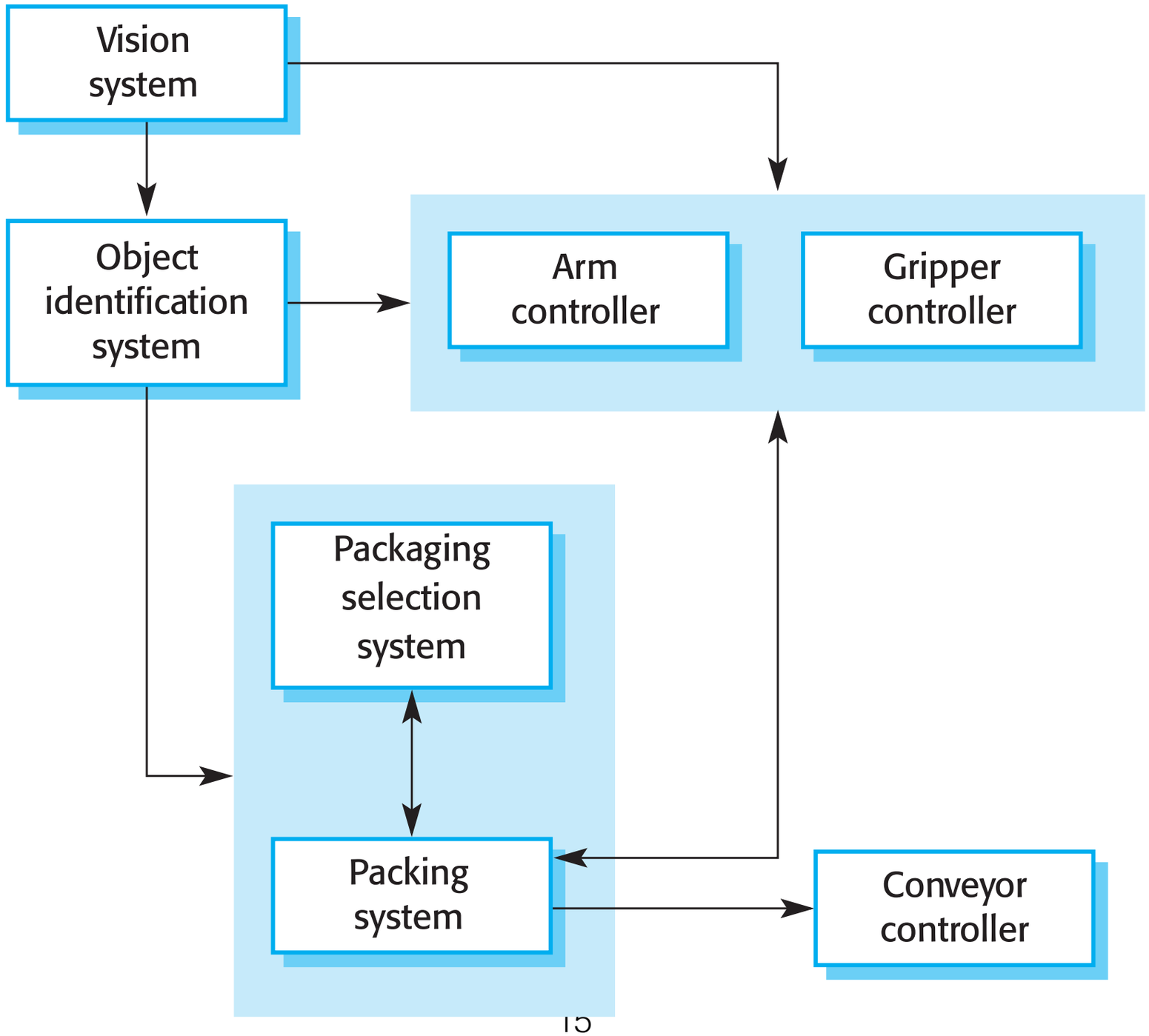
System structuring

- Concerned with decomposing the system into interacting sub-systems.

- The architectural design is normally expressed as a block diagram presenting an overview of the system structure.

- More specific models showing how sub-systems share data, are distributed and interface with each other may also be developed.

Example:
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

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 [SE9] and [OOSE];
  - product lines are covered in chapter 16 and will not be covered in this course.

Architectural views

- It is impossible to represent all relevant information about a system's architecture 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

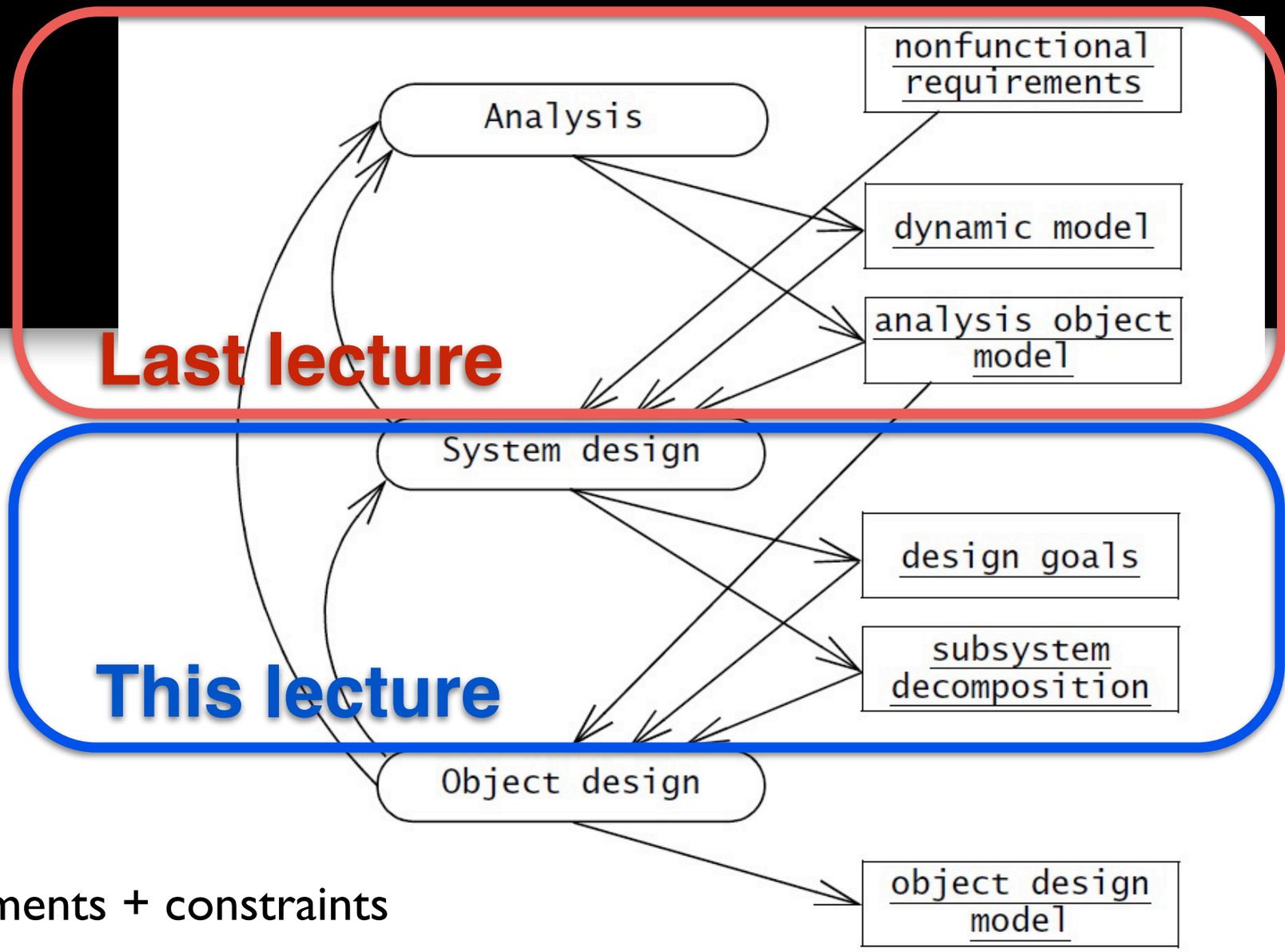
Object oriented design

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; …

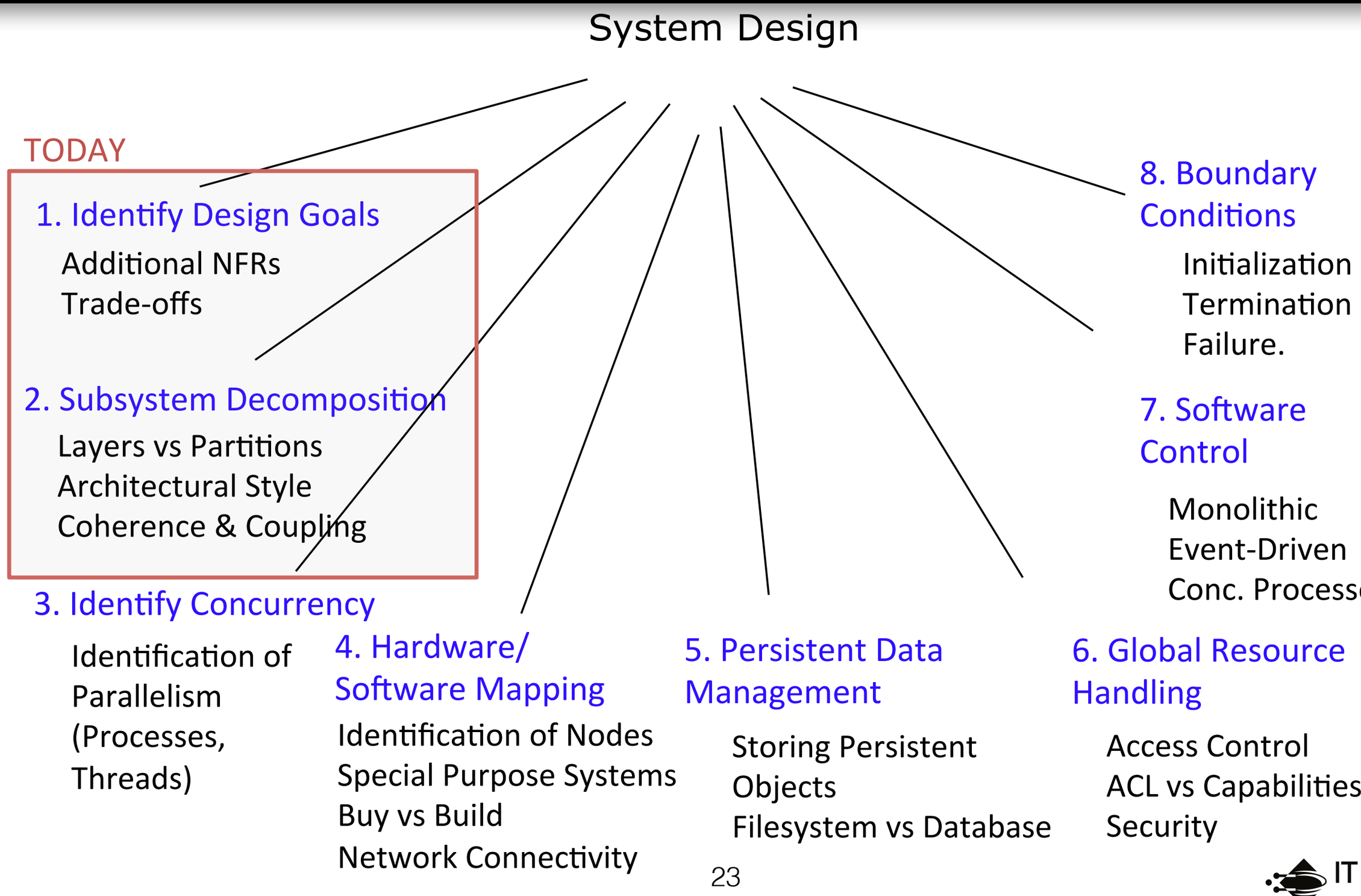
From analysis to design



- 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 issues



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

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.

- 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 redund… …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).

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.

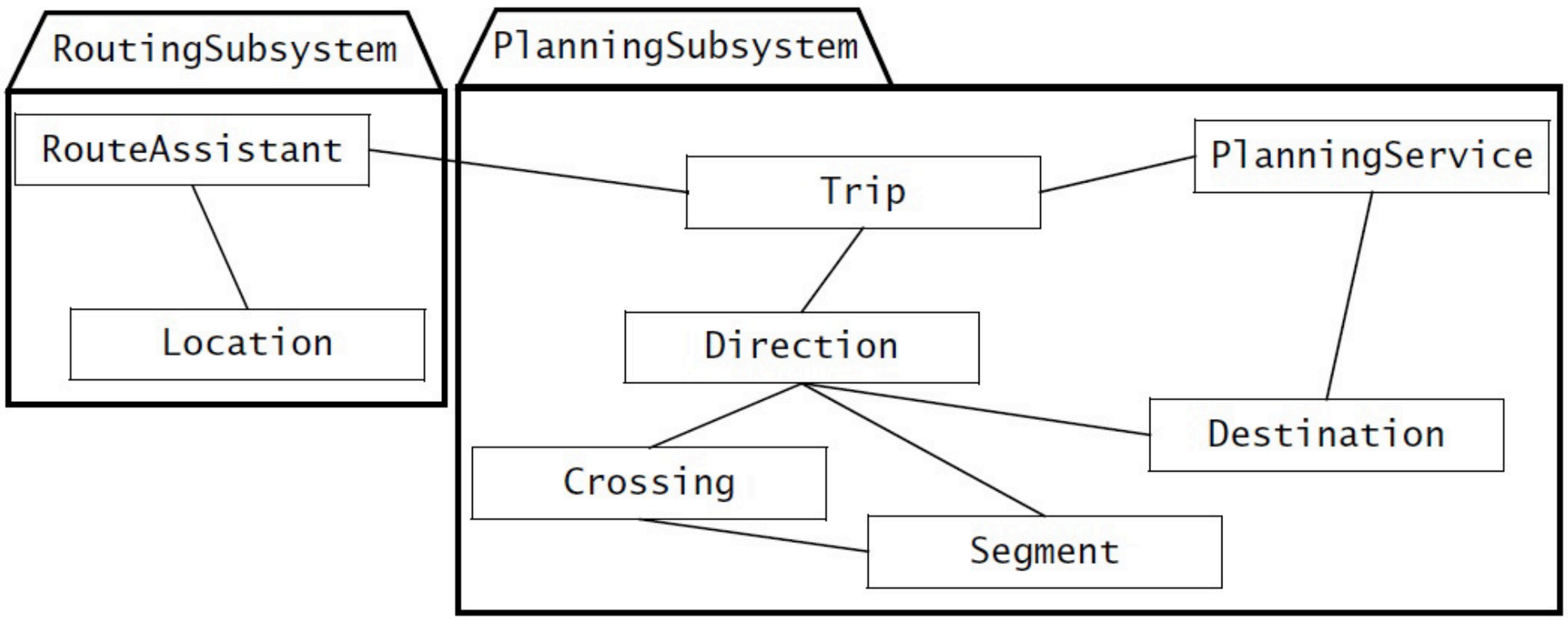
Trade-off	Rationale
Space vs. speed	If the software does not meet response time or throughput requirements, more memory can be expended to speed up the software (e.g., caching, more redundancy). If the software does not meet memory space constraints, data can be compressed at the cost of speed.
Delivery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

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



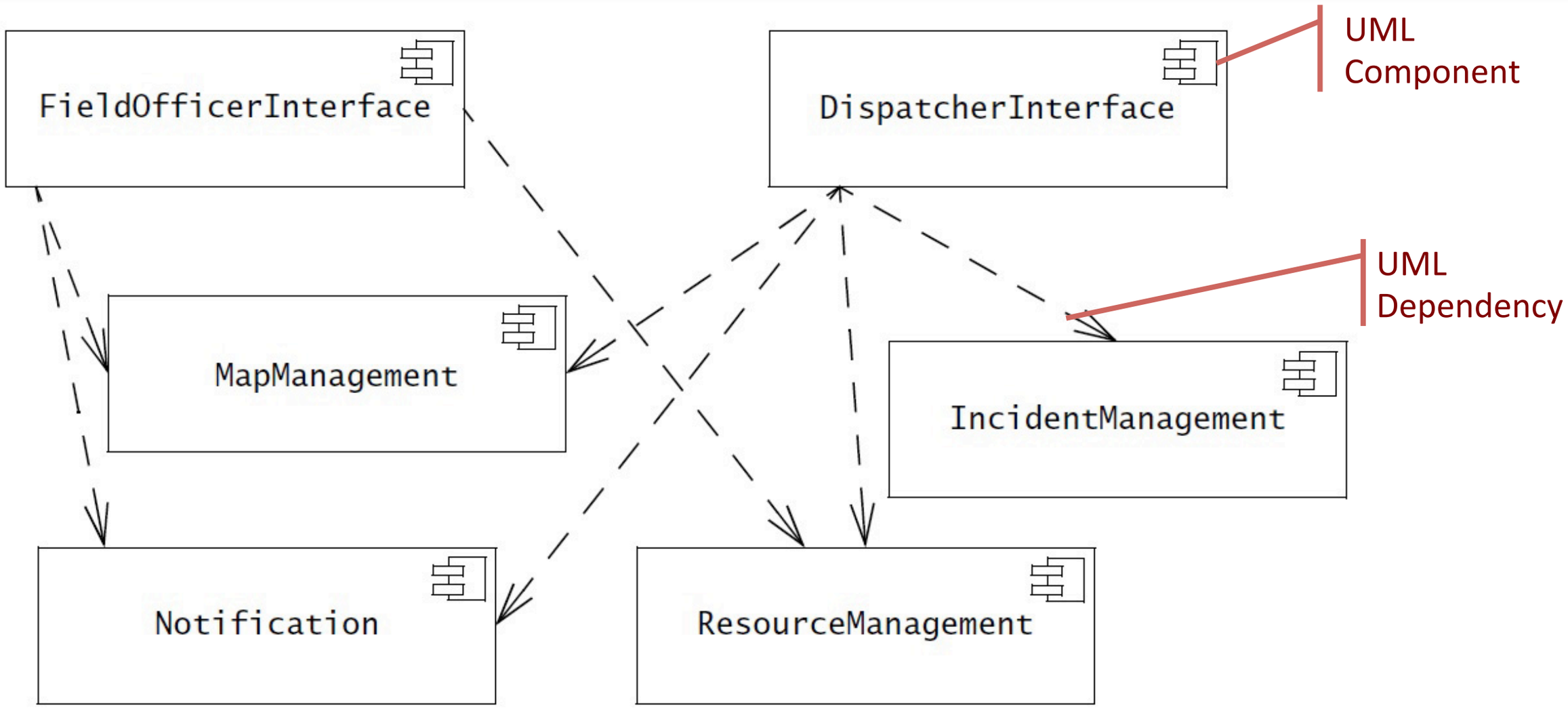
PlanningSubsystem	The PlanningSubsystem is responsible for constructing a Trip connecting a sequence of Destinations. The PlanningSubsystem is also responsible for responding to replan requests from RoutingSubsystem.
RoutingSubsystem	The RoutingSubsystem is responsible for downloading a Trip from the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

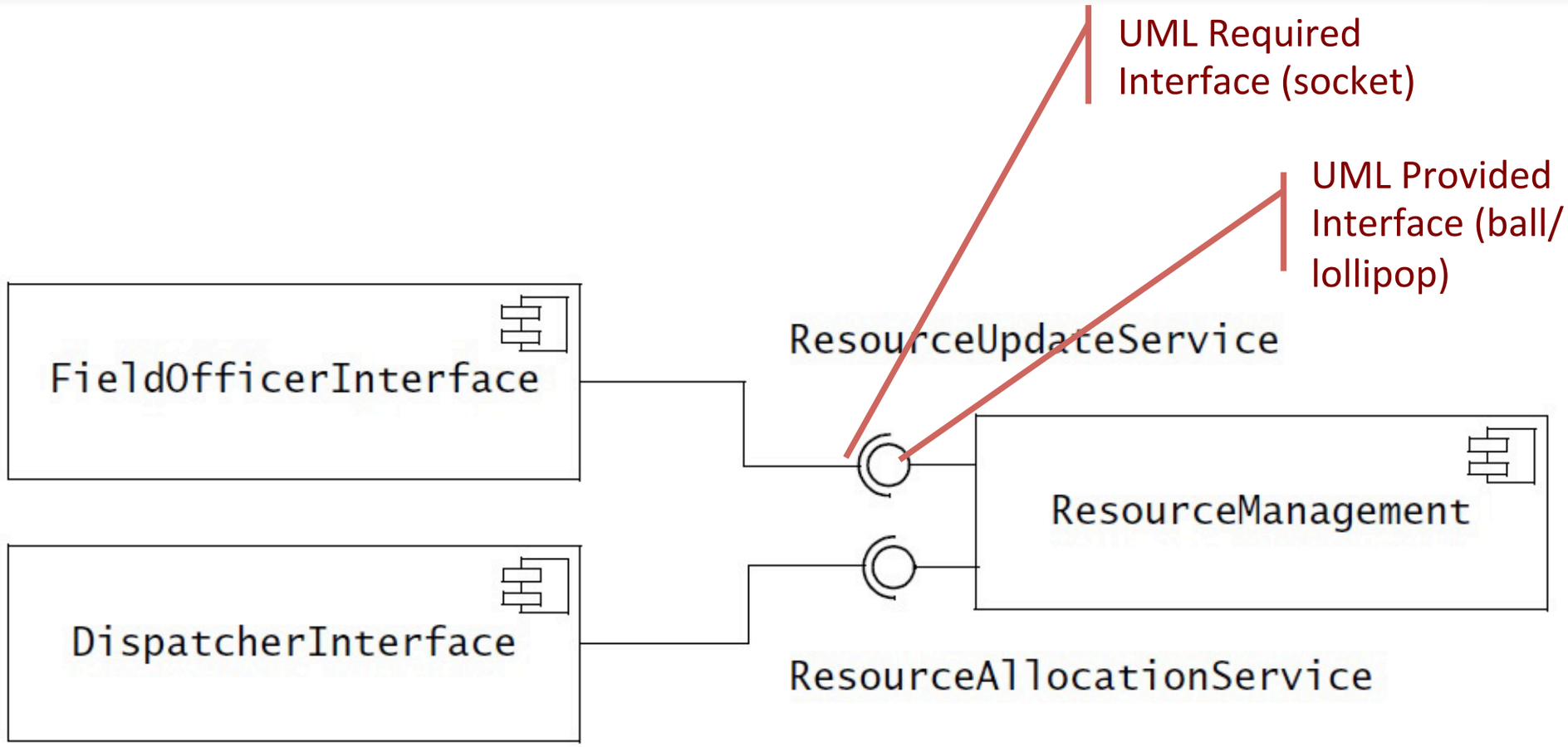
- Subsystems and Classes
- Services and Subsystems Interfaces
- Coupling and Cohesion
- Layers and Partitions
- Architectural Styles

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)

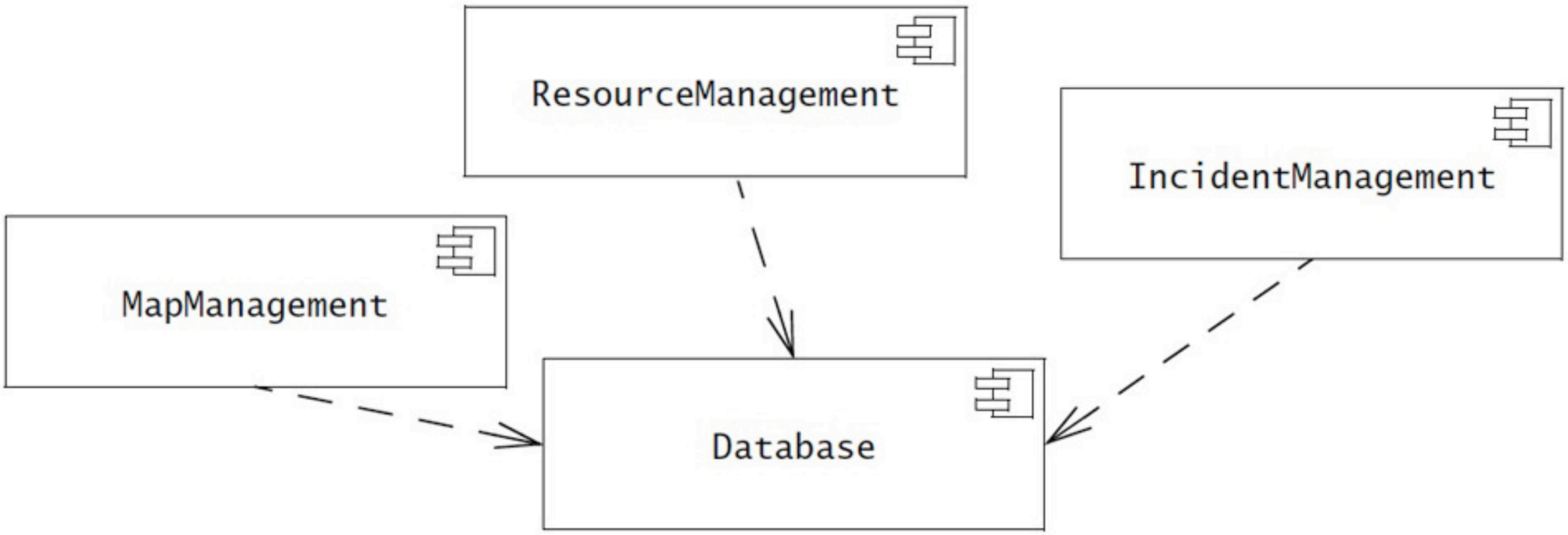
Coupling & cohesion

- 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?

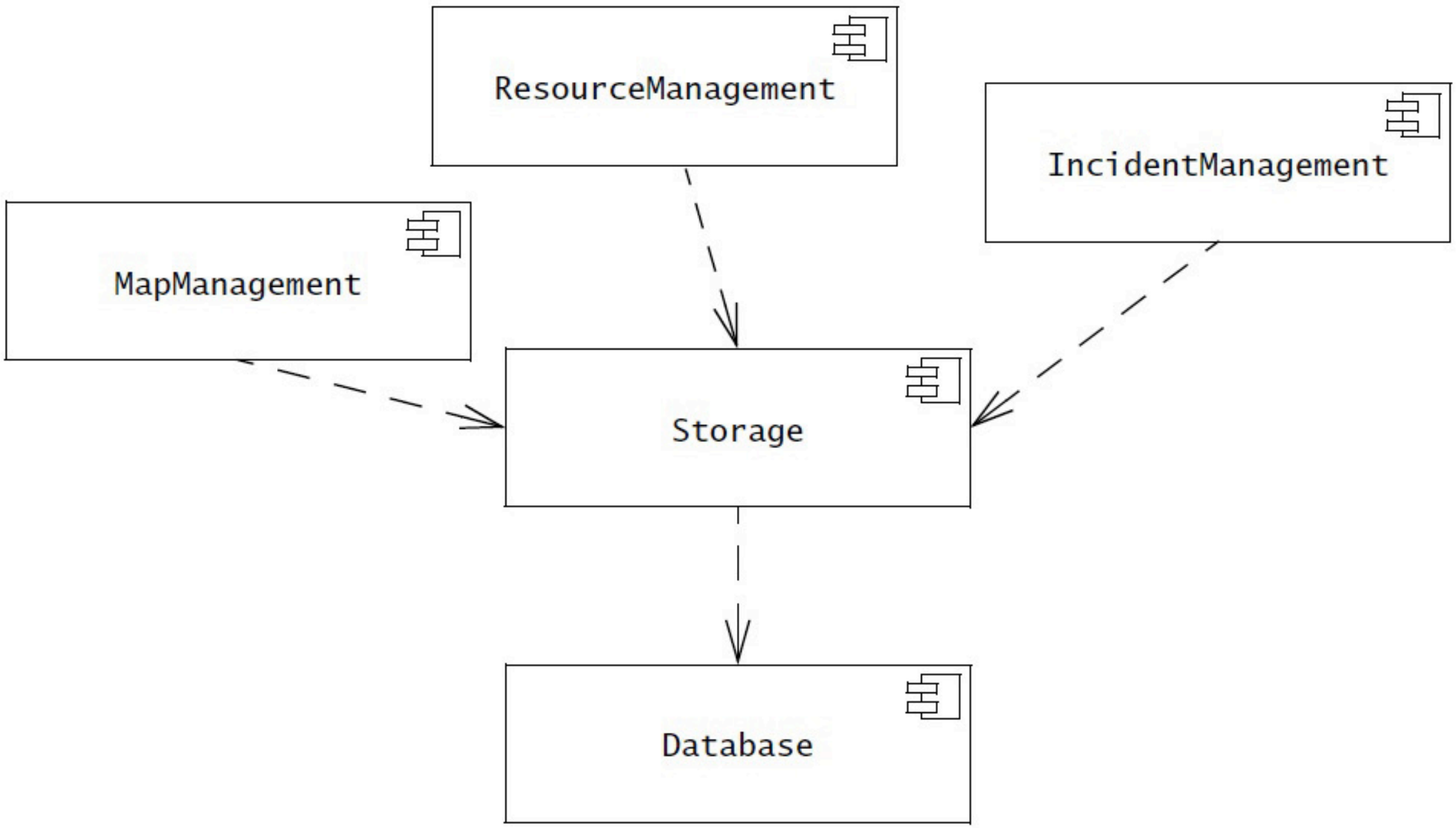
Coupling

Alternative 1: Direct access to the Database subsystem

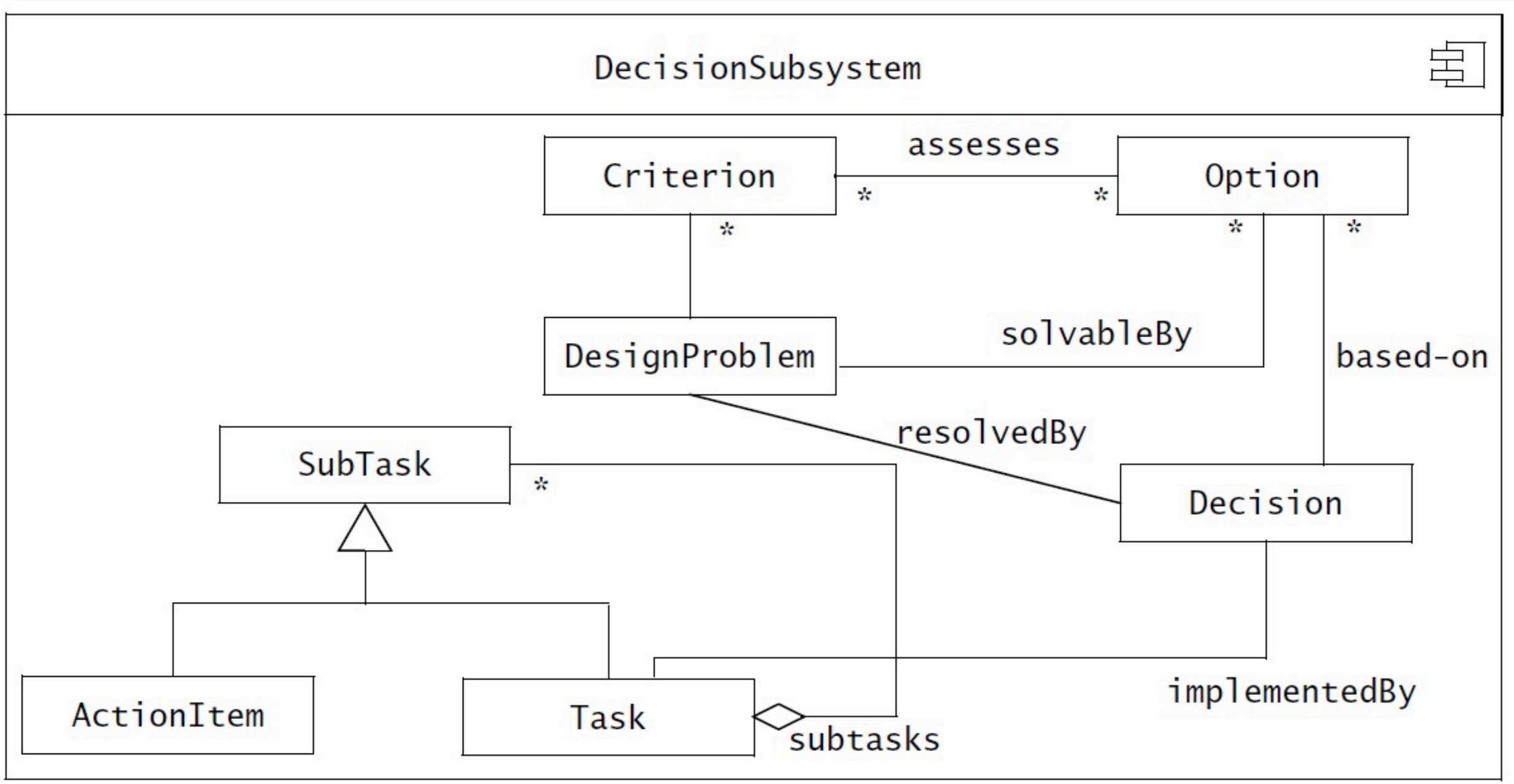


Coupling

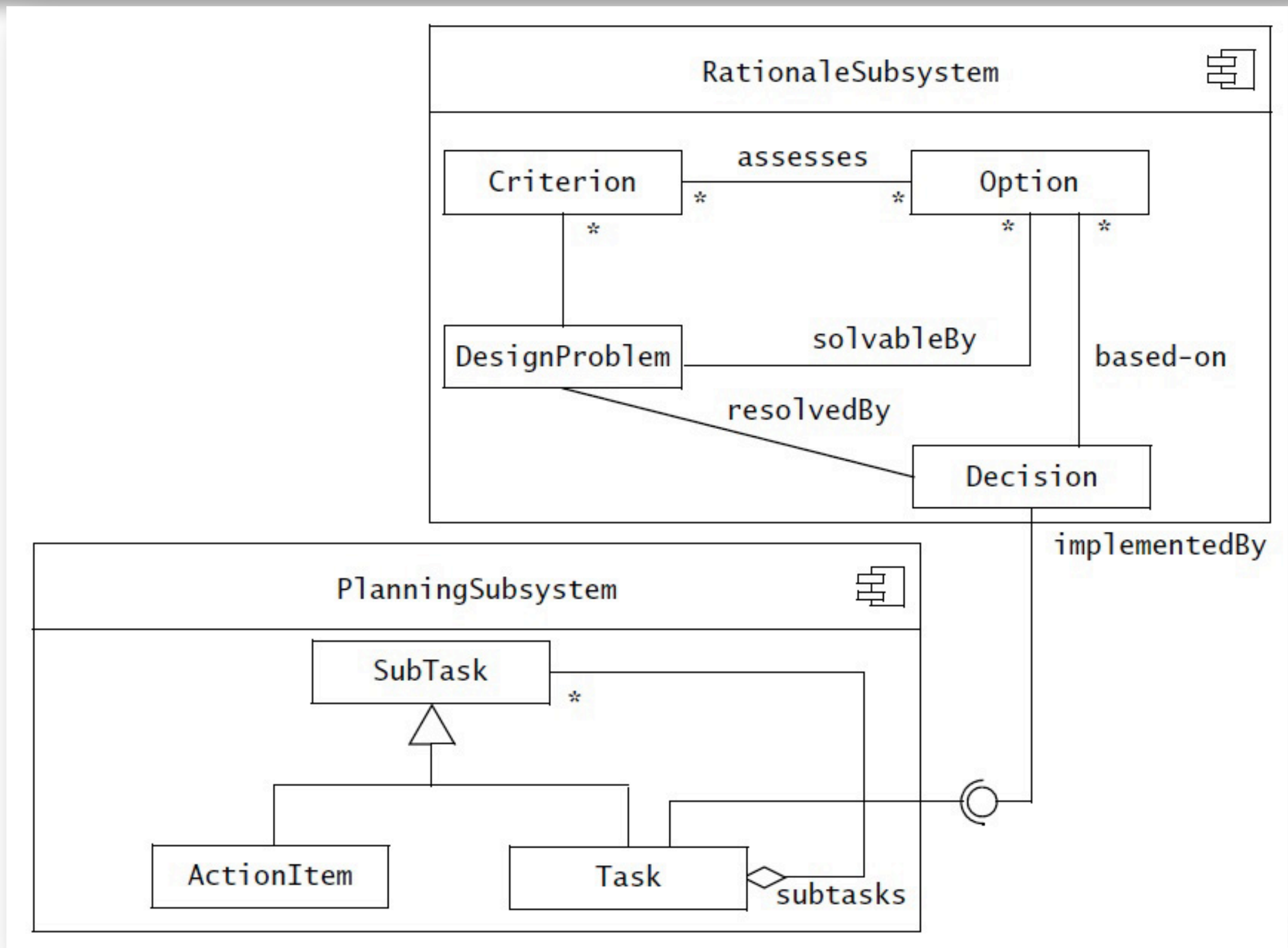
Alternative 2: Indirect access to the Database through a Storage subsystem



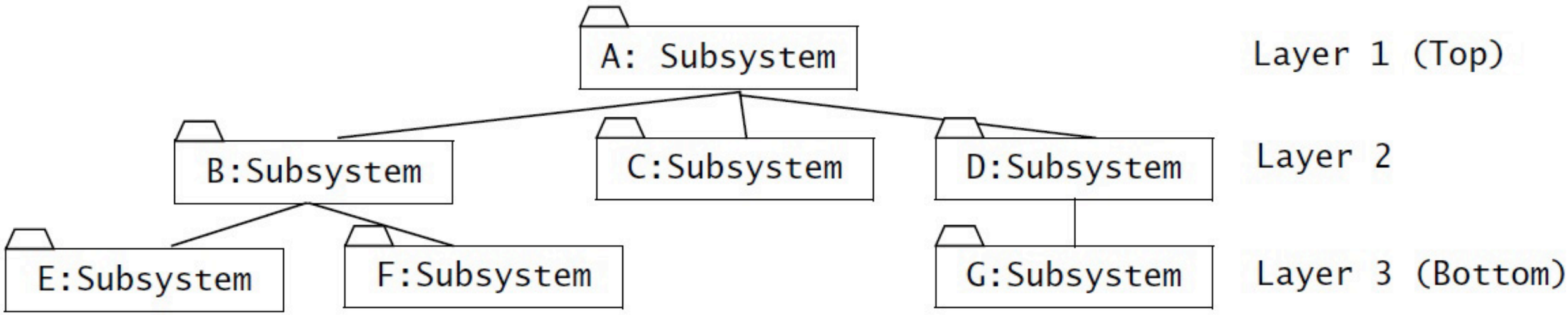
Cohesion


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

Cohesion


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 …

Patterns are ways to describe best practices, good design, and capture experience in a way that is possible for others to reuse this experience.

- stylized, abstract description of good practice;
- tried and tested in different environments;
- system organization that has proved to be successful;
- described when to use it – and when not;
- strengths and weaknesses.

Patterns

Overview

- Architectural patterns or styles
  - Model-View-Controller (MVC)
  - Layered
  - Repository
  - Client-Server
  - Pipe & Filter
- Design patters
  - Observer
  - Strategy
  - Factory method
  - Template method

Reporting patterns

Pattern name: Observer

Description: Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change.

Problem description: In many situations, you have to provide multiple displays of state information, such as a graphical display and a tabular display. Not all of these may be known when the information is specified. All alternative presentations should support interaction and, when the state is changed, all displays must be updated.

This pattern may be used in all situations where more than one display format for state information is required and where it is not necessary for the object that maintains the state information to know about the specific display formats used.

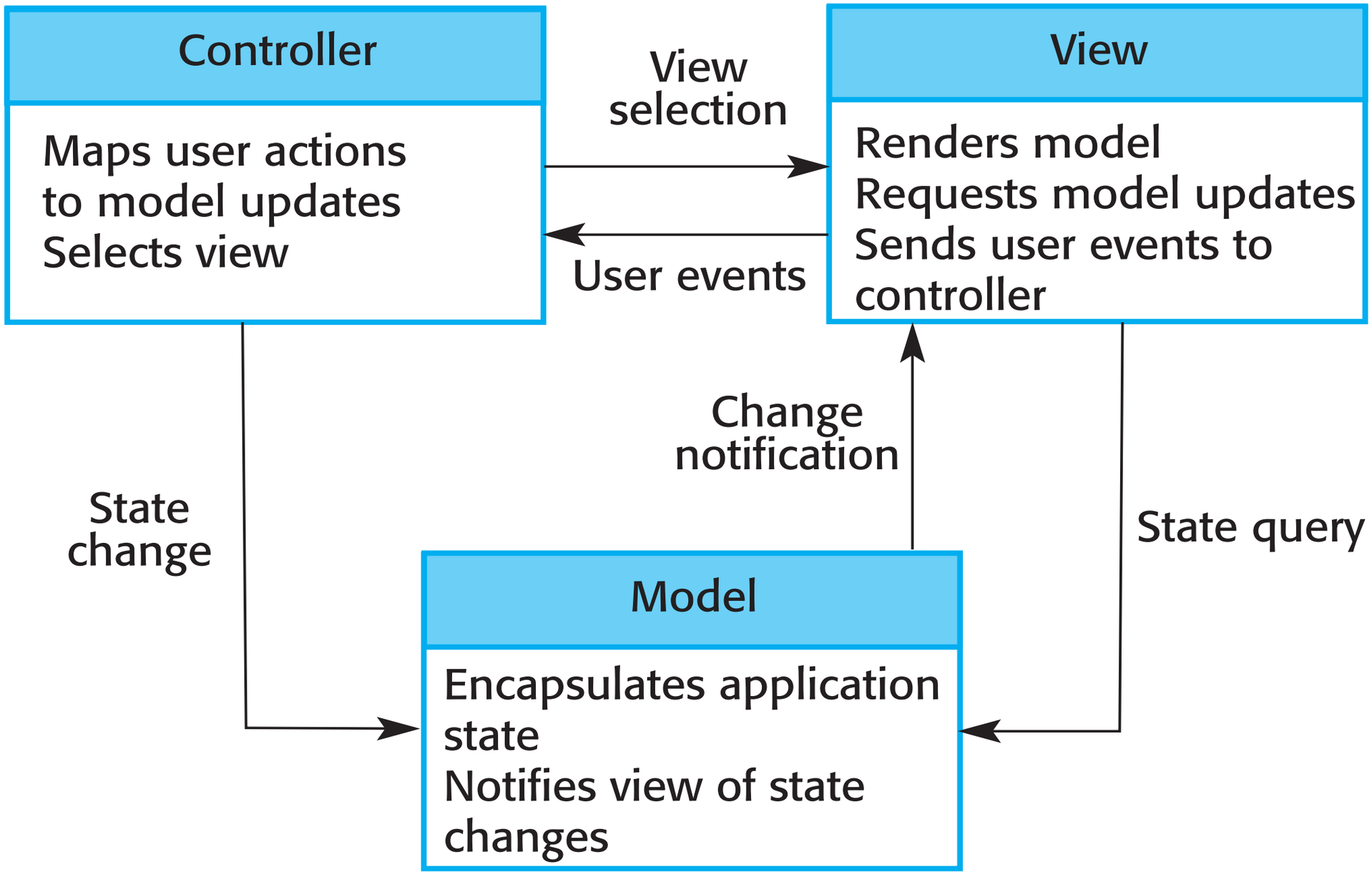
Solution description: This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObject, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all situations. The state to be displayed is maintained in ConcreteSubject, which inherits operations from Subject allowing it to add and remove Observers (each observer corresponds to a display) and to issue a notification when the state has changed.

The ConcreteObserver maintains a copy of the state of ConcreteSubject and implements the Update() interface of Observer that allows these copies to be kept in step. The ConcreteObserver automatically displays the state and reflects changes whenever the state is updated.
The UML model of the pattern is shown in Figure 7.12.

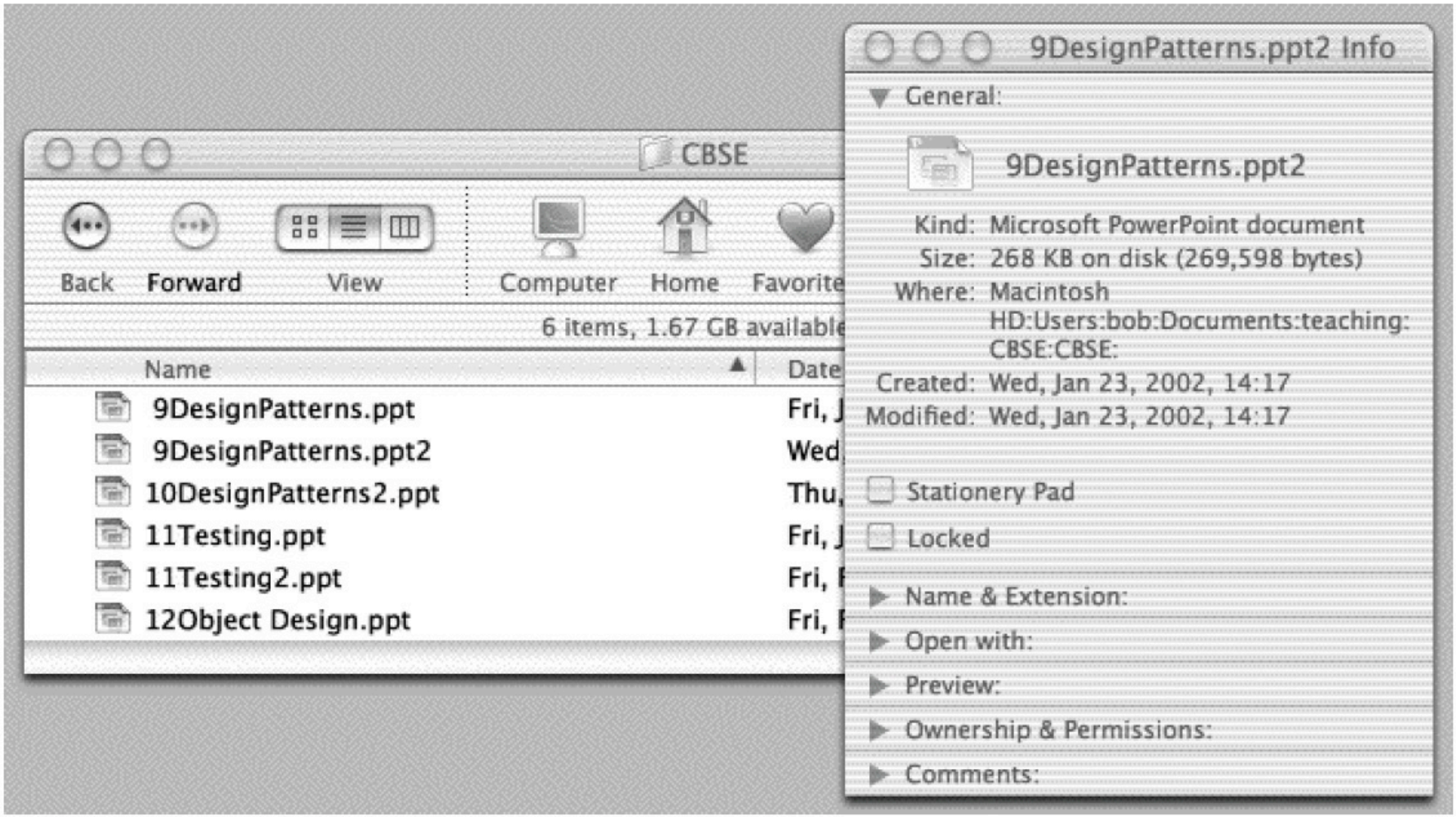
Consequences: The subject only knows the abstract Observer and does not know details of the concrete class. Therefore there is minimal coupling between these objects. Because of this lack of knowledge, optimizations that enhance display performance are impractical. Changes to the subject may cause a set of linked updates to observers to be generated, some of which may not be necessary.

Architectural patterns (styles)

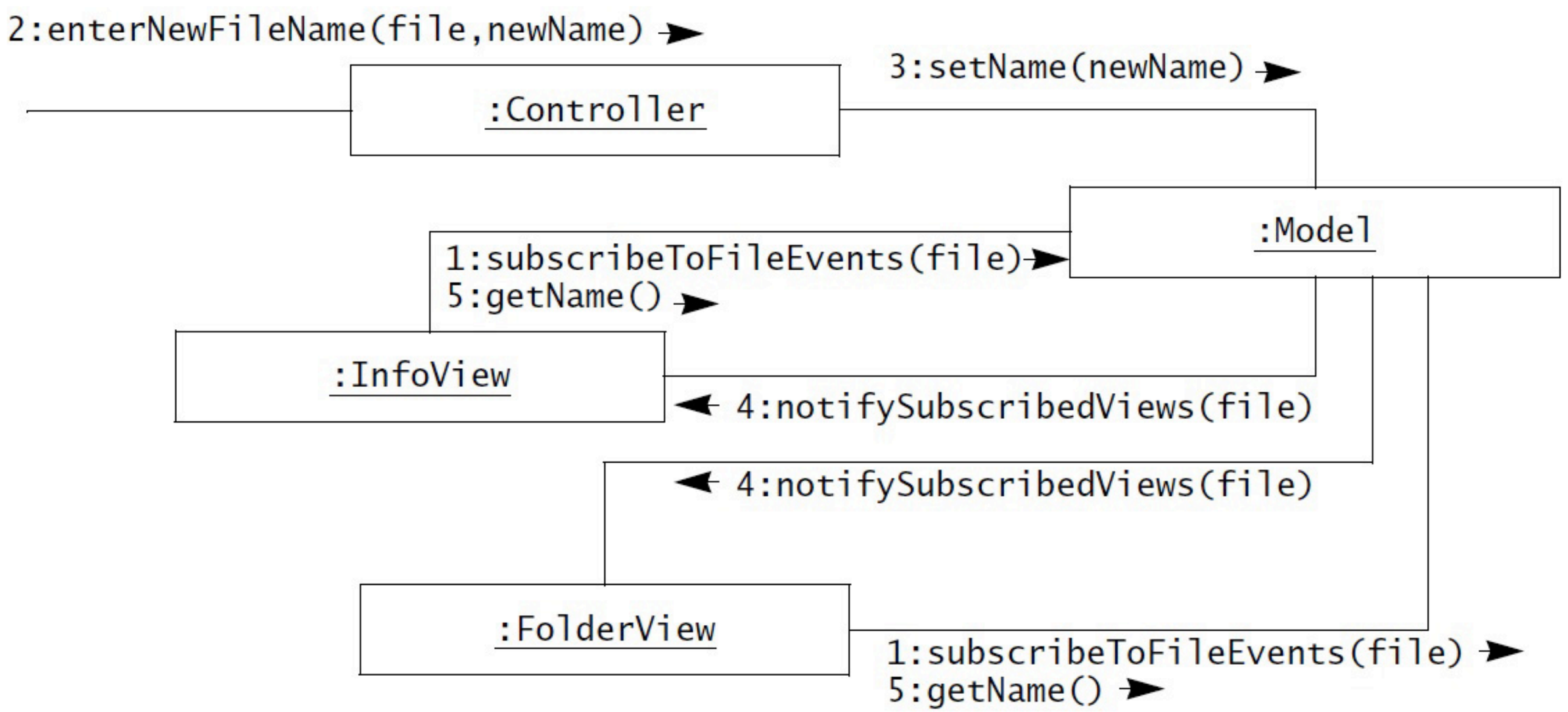
The organization of the Model-View-Controller (MVC)



Model-View-Controller (MVC)



Model-View-Controller (MVC)



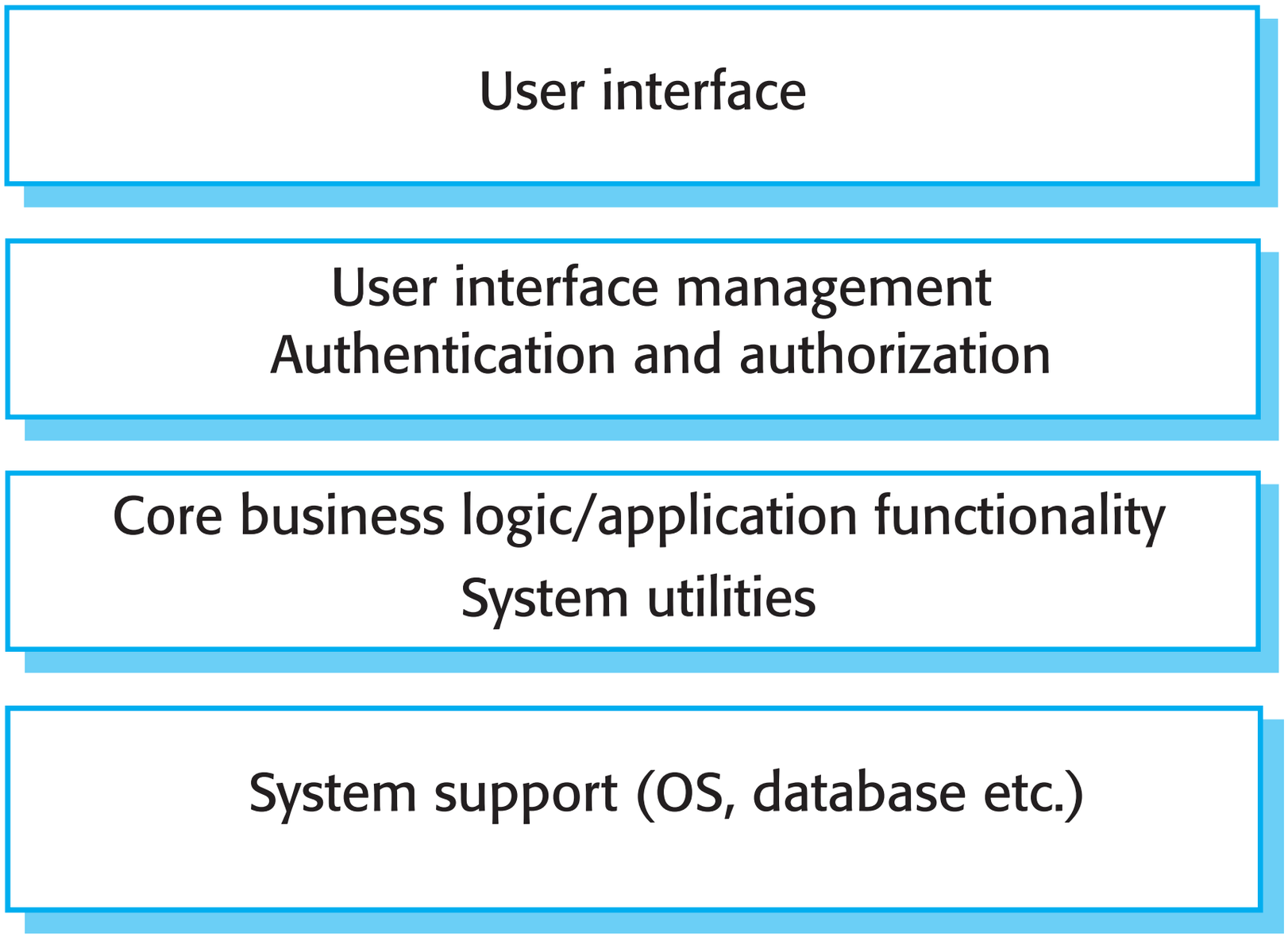
The MVC pattern

Name	MVC (Model-View-Controller)
Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model. See Figure 6.3.
Example	Figure 6.4 shows the architecture of a web-based application system organized using the MVC pattern.
When used	Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.
Advantages	Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways with changes made in one representation shown in all of them.
Disadvantages	Can involve additional code and code complexity when the data model and interactions are simple.

Layered architecture

- 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.

A generic layered architecture



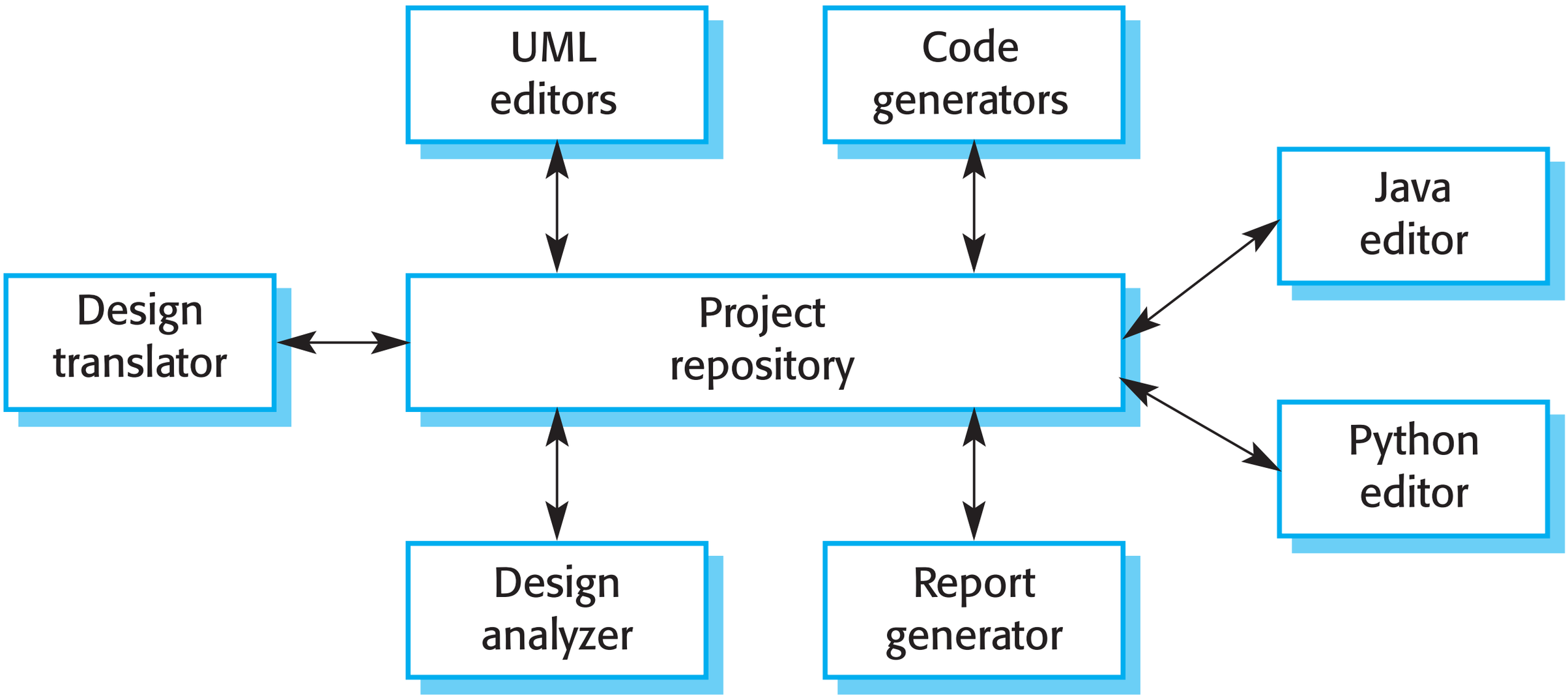
The layered architecture pattern

Name	Layered architecture
Description	Organizes the system into layers with related functionality associated with each layer. A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system. See Figure 6.6.
Example	A layered model of a system for sharing copyright documents held in different libraries, as shown in Figure 6.7.
When used	Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multi-level security.
Advantages	Allows replacement of entire layers so long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.
Disadvantages	In practice, providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

Repository architecture

- 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 a this is an efficient data sharing mechanism.

A generic layered architecture



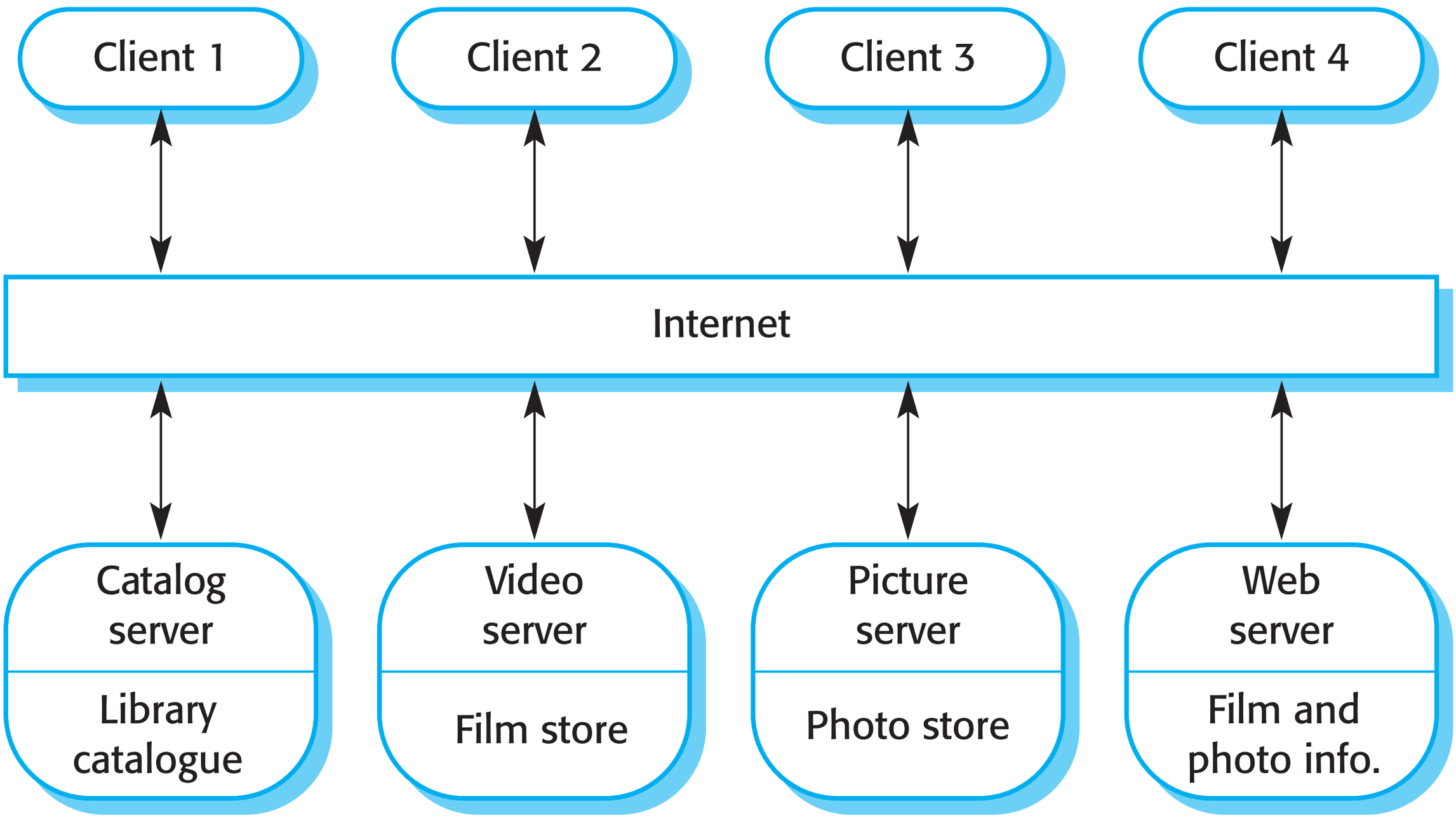
The repository pattern

Name	Repository
Description	All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.
Example	Figure 6.9 is an example of an IDE where the components use a repository of system design information. Each software tool generates information which is then available for use by other tools.
When used	You should use this pattern when you have a system in which large volumes of information are generated that has to be stored for a long time. You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool.
Advantages	Components can be independent—they do not need to know of the existence of other components. Changes made by one component can be propagated to all components. All data can be managed consistently (e.g., backups done at the same time) as it is all in one place.
Disadvantages	The repository is a single point of failure so problems in the repository affect the whole system. May be inefficiencies in organizing all communication through the repository. Distributing the repository across several computers may be difficult.

Client-Server architecture

- Distributed system model which shows how data and processing is distributed across a range of components.
  - Can be implemented on a single computer.
- Three parts:
  - Set of stand-alone servers which provide specific services such as printing, data management, etc.
  - Set of clients which call on these services.
  - Network which allows clients to access servers.

A client–server architecture for a film library



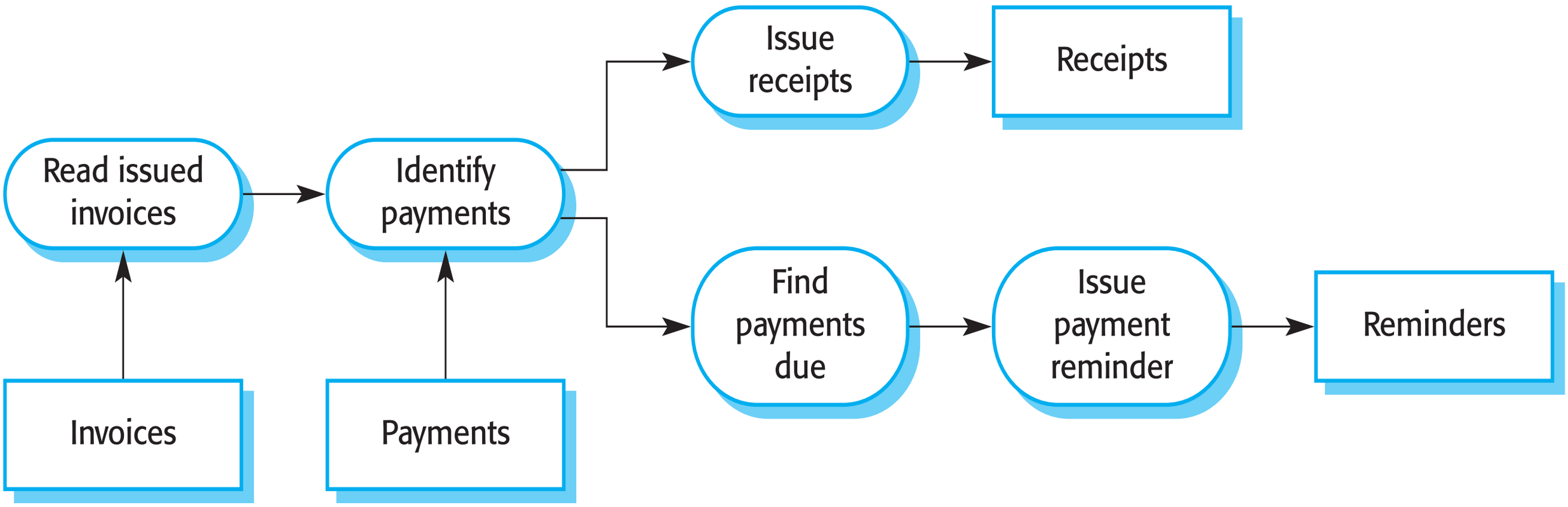
The client–server pattern

Name	Client-server
Description	In a client–server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.
Example	Figure 6.11 is an example of a film and video/DVD library organized as a client–server system.
When used	Used when data in a shared database has to be accessed from a range of locations. Because servers can be replicated, may also be used when the load on a system is variable.
Advantages	The principal advantage of this model is that servers can be distributed across a network. General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.
Disadvantages	Each service is a single point of failure so susceptible to denial of service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. May be management problems if servers are owned by different organizations.

Pipes and Filters architecture

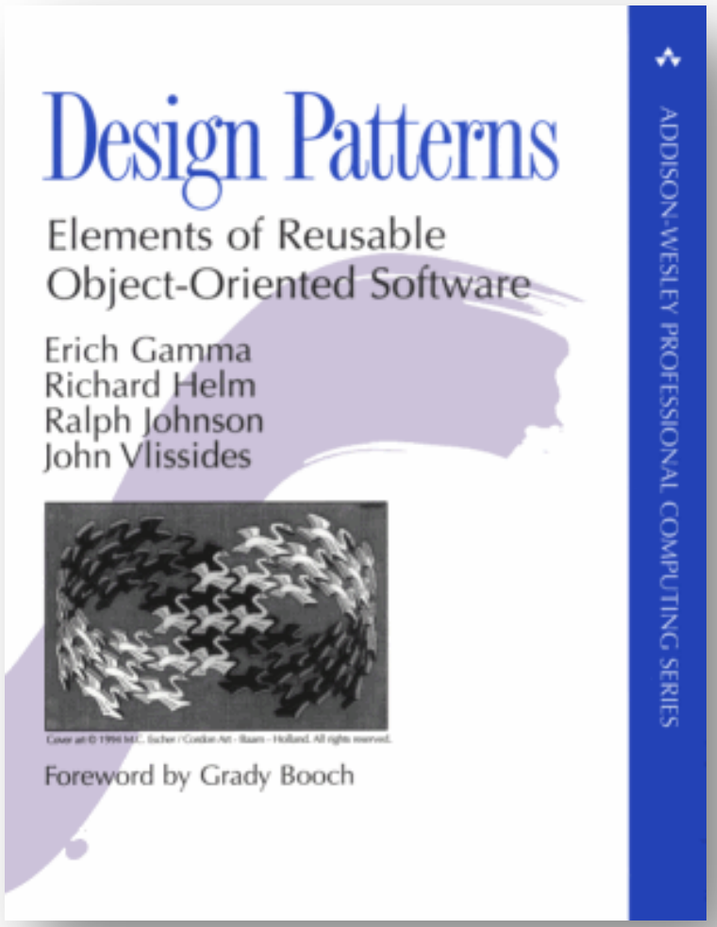
- Functional transformations process their inputs to produce outputs.
- May be referred to as a pipe and filter model (as in UNIX shell).
- Variants of this approach are very common. When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems.
- Not really suitable for interactive systems.

An example of the pipe and filter architecture



The pipe and filter pattern

Name	Pipe and filter
Description	The processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.
Example	Figure 6.13 is an example of a pipe and filter system used for processing invoices.
When used	Commonly used in data processing applications (both batch- and transaction-based) where inputs are processed in separate stages to generate related outputs.
Advantages	Easy to understand and supports transformation reuse. Workflow style matches the structure of many business processes. Evolution by adding transformations is straightforward. Can be implemented as either a sequential or concurrent system.
Disadvantages	The format for data transfer has to be agreed upon between communicating transformations. Each transformation must parse its input and unparse its output to the agreed form. This increases system overhead and may mean that it is impossible to reuse functional transformations that use incompatible data structures.



Originally proposed by the "Gang-of-Four" (GoF)

Erich Gamma, Richard Helm,
Ralph Johnson & John Vlissides

Design patterns

Overview

Structural	Behavioural	Creational
Adapter	Strategy	Builder
Façade	State	Prototype
Composite	Command	Factory method
Decorator	Observer	Abstract factory
Bridge	Memento	
Singleton	Interpreter	
Proxy	Iterator	
Flyweight	Visitor	
	Mediator	
	Template method	
	Chain of responsibility	

Overview

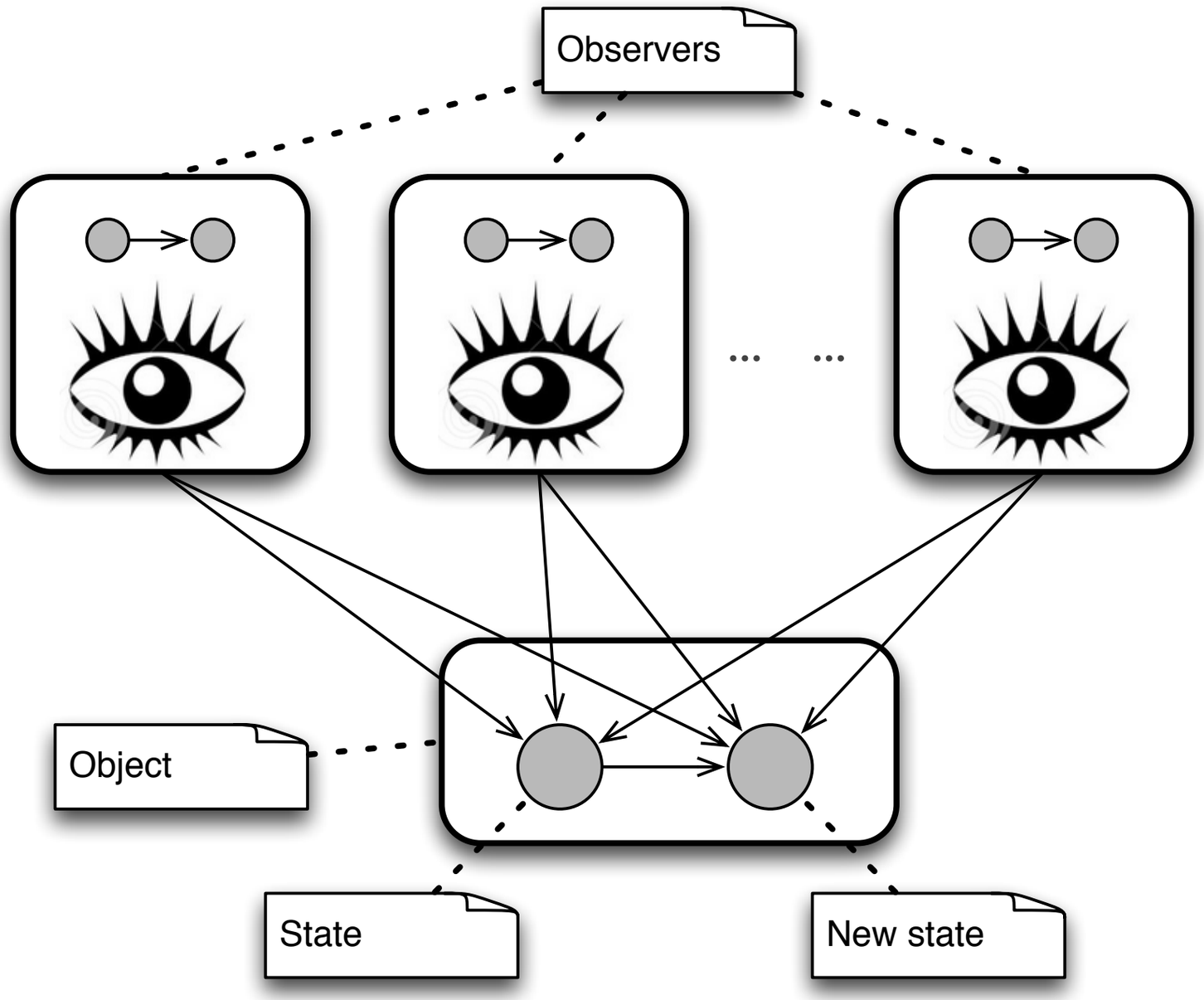
Structural	Behavioural	Creational
Adapter	**Strategy**	Builder
Façade	State	Prototype
Composite	Command	**Factory method**
Decorator	**Observer**	Abstract factory
Bridge	Memento	
Singleton	Interpreter	
Proxy	Iterator	
Flyweight	Visitor	
	Mediator	
	Template method	
	Chain of responsibility	

Observer

- Purpose
  - Have a set of objects "observing" the state of an object …
  - … so that they will be notified and updated immediately when the state changes.

- The name is counter intuitive …
  - … the observer does not observe, but awaits to be notified to go looking.

An idea? (naive)

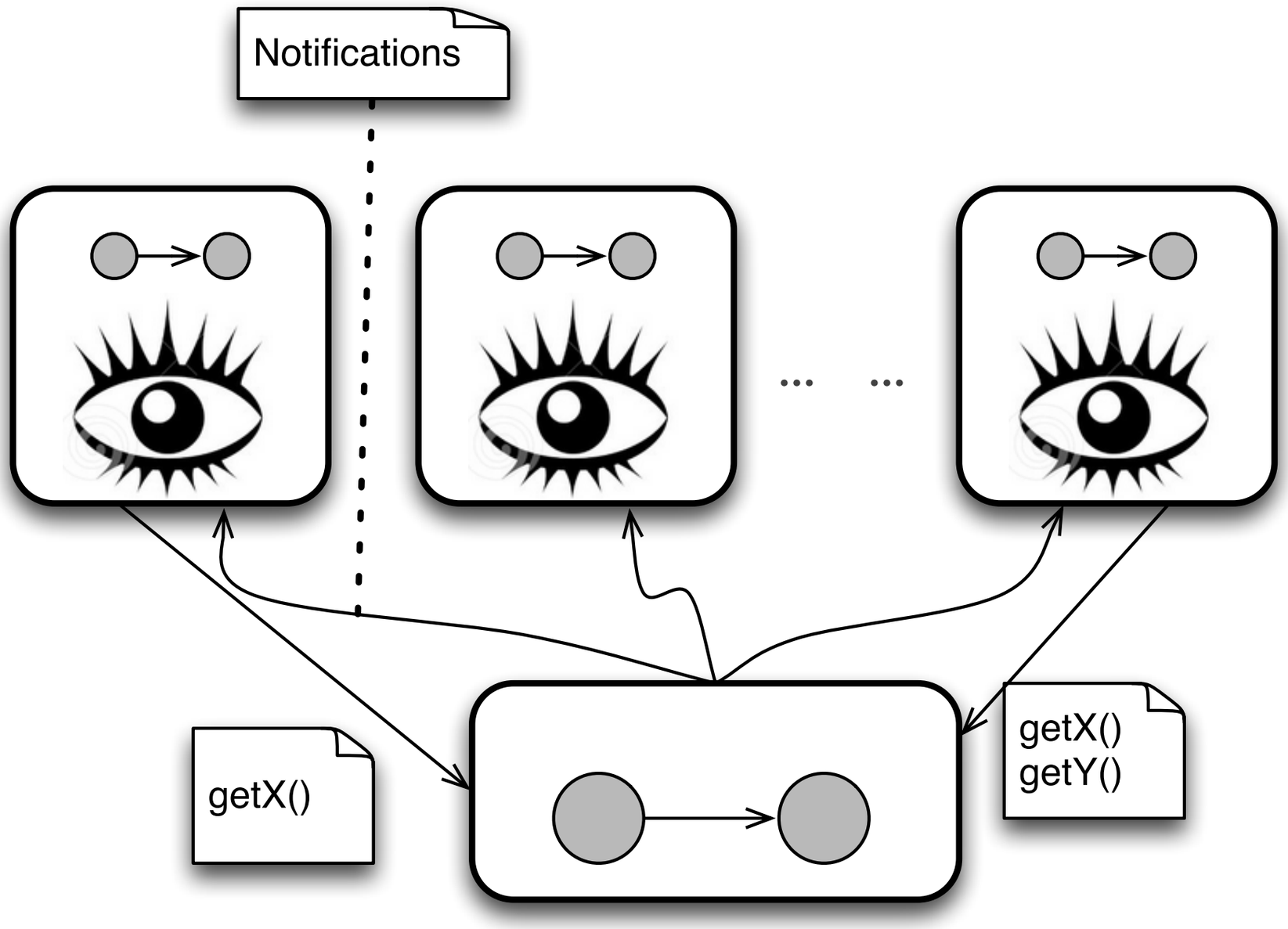


Let's try

- Public attributes? Rejected
- The observers constantly invoke a method `changed` of the observed object?

- Problems! The observer could:
  - be bothering too much the observed object
  - discover the change in the state too late
  - loose one of many rapid changes in the state
  - And if there are many observers? The solution is not scalable!
    - The observer would be spending most of the execution time replying to the observers (potentially to report no variation).

Revisited solution



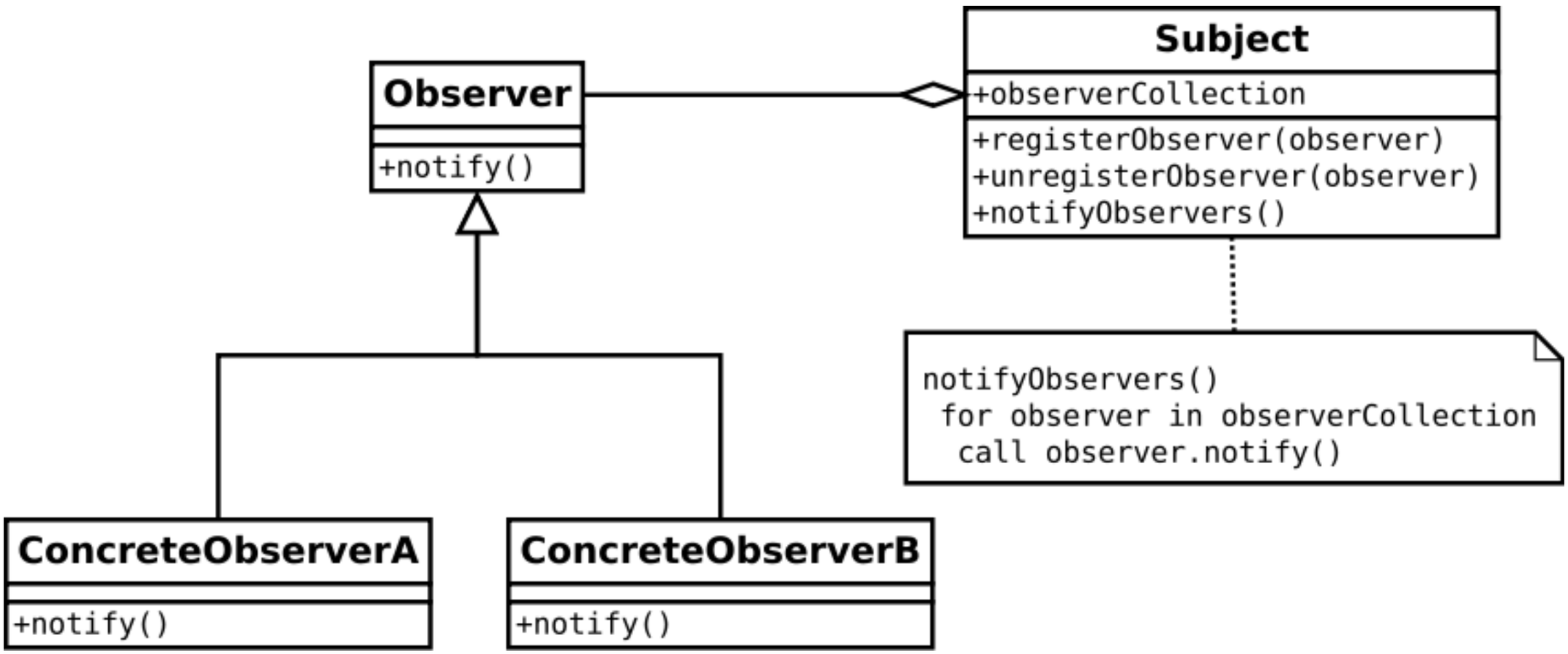
… therefore

- (Prerequisite: the observers need to register)
- When the state of the object changes, it notifies all the observers (by calling a method)
- When notified, the observers decide what to do with the knowledge
  - Nothing
  - Request information about the state of the object
  - …

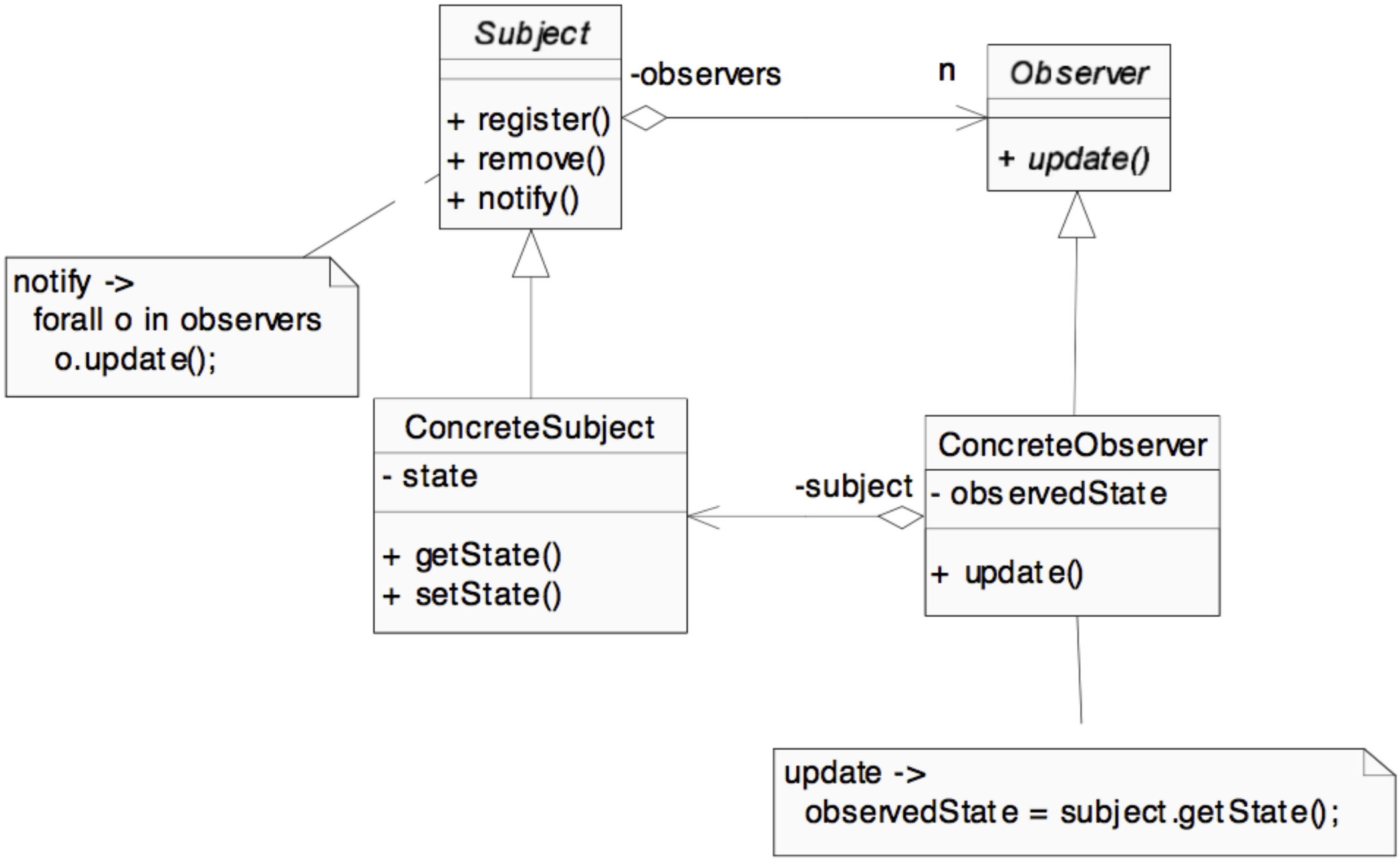
… therefore

- The observers could be “views” on the same object
  - Data on a spreadsheet and their tabular, graphical, … representations
- The observed object does not know the exact identity of the observers
- Registration and unregistration to the object happens at runtime

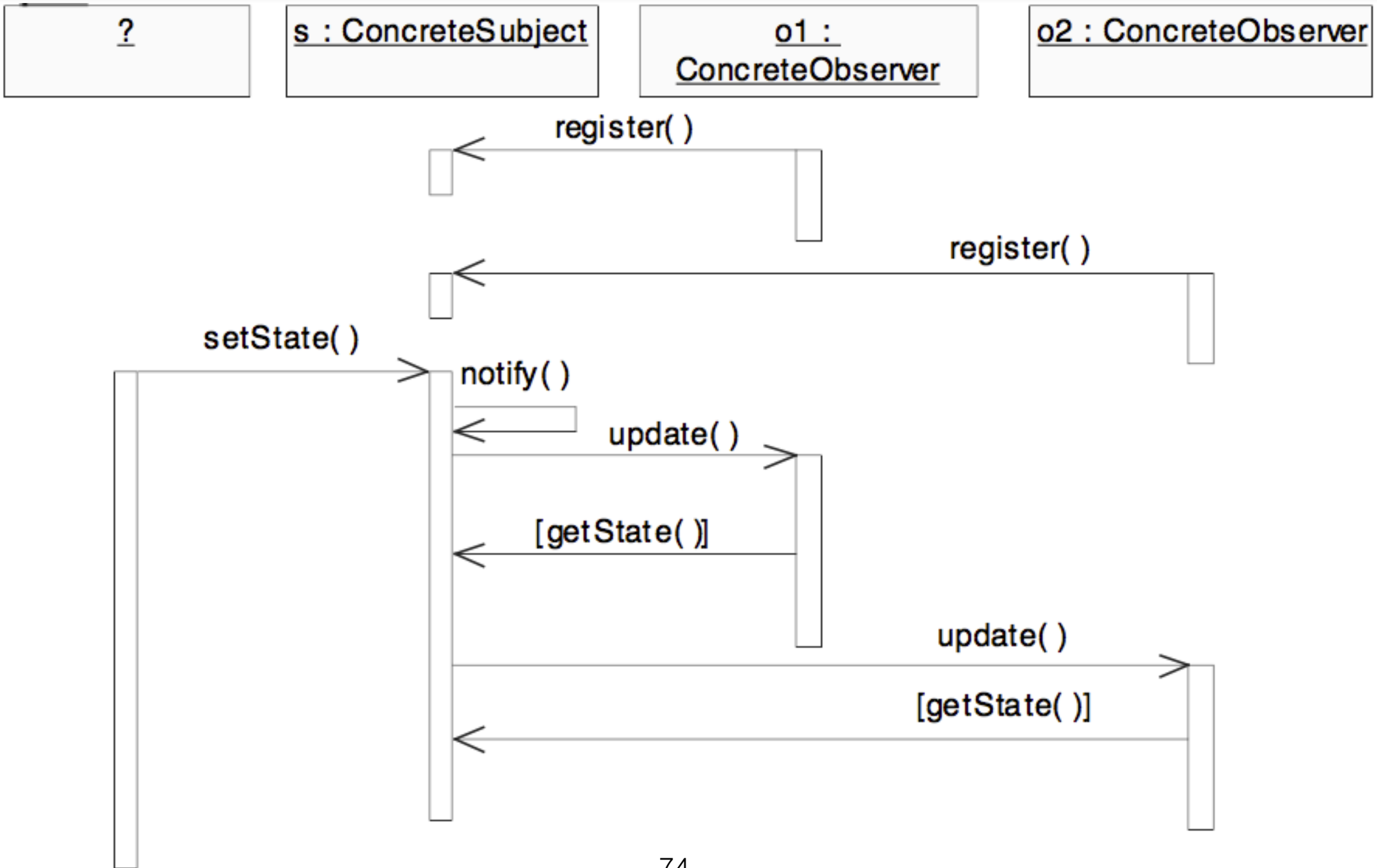
Observer class diagram



Observer class diagram



Observer sequence diagram



Comments

- Who invokes `setState()`?
  - Anybody
  - … also one of the observers could

- Who invokes `notify()`?
  - Every method of the object observed that modifies the state
  - … or, the client after having finished a sequence of modifications.

Factory method

- Purpose
  - Delegate the creation of a class to one of the subclasses
  - The subclass(es) decide(s) which instance to create, which constructor to call.

- Typically used in OO frameworks.

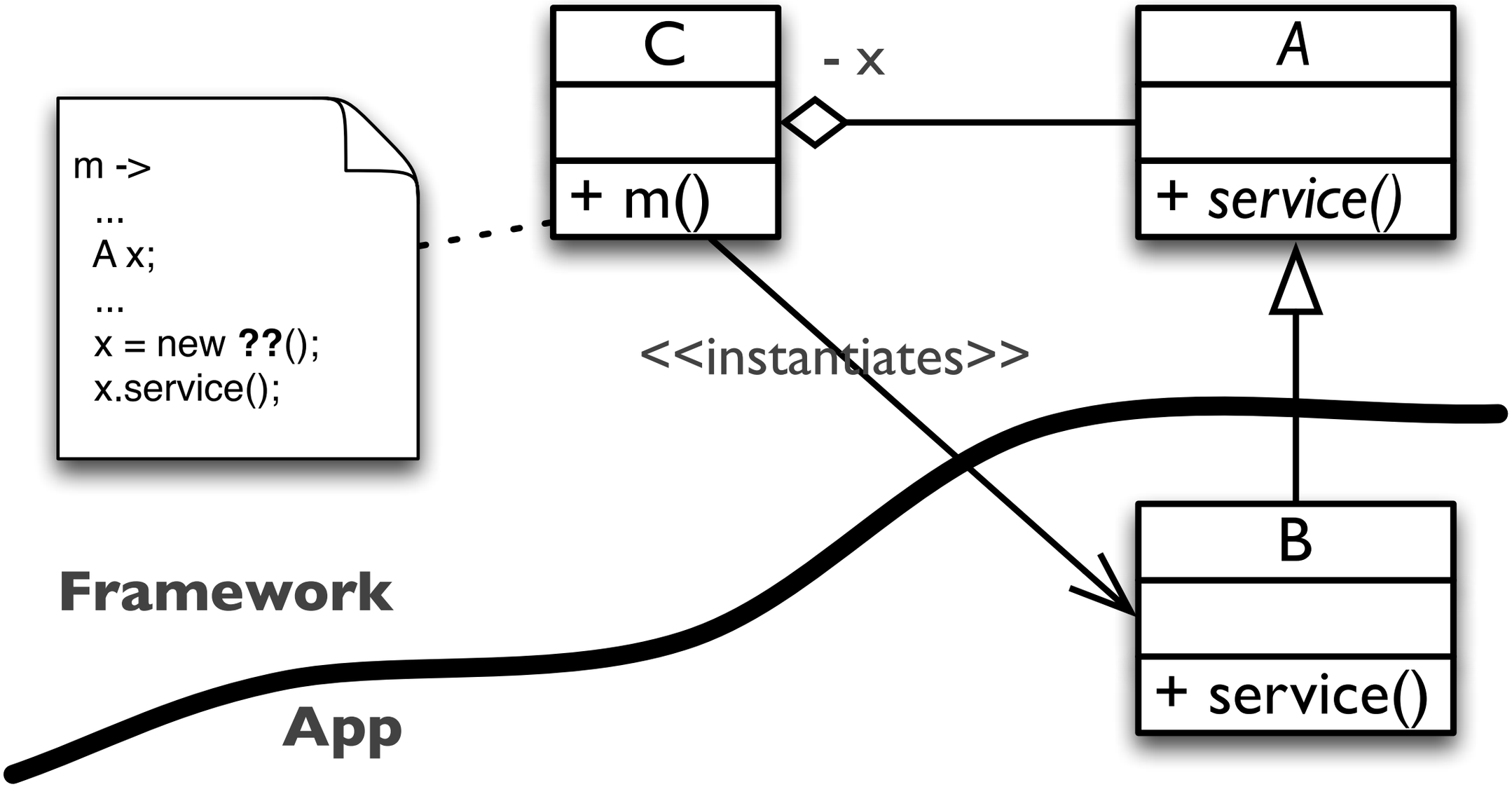
What is an OO framework?

- A set of classes
- Difference with a library:
  - Library: the developer of a system calls the methods of the library
  - Framework: the developer of a system
    - writes methods that are called by the framework
    - “fills the gaps”
    - “Do not call us, we will call you”

Typical problem in a framework

- The framework developer writes a method in a class `C` …
- … the method needs to create, at a certain precise point, an instance of a subclass `B` of a base class `A` …
- … but which subclass `B` to use is unknown as it is a decision that the developer of the application needs to take, not the framework developer.

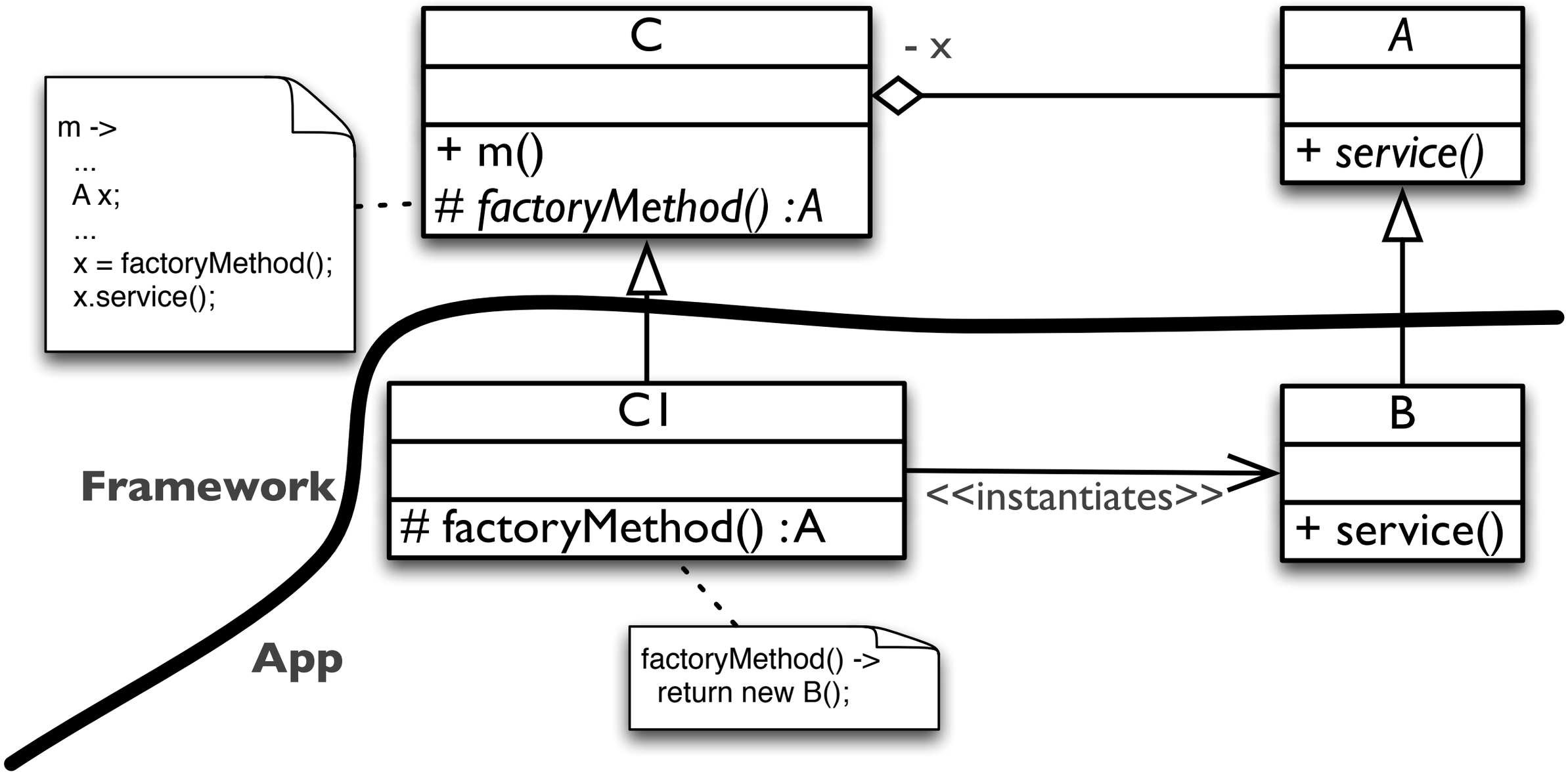
The problem


C
- x
A
+ m()
+ service()
m ->
...
A x;
...
x = new ??();
x.service();
<<instantiates>>
B
+ service()
Framework
App

Solutions?

- First (non) solution:
  - the framework developer gives up and lets the application developer do all the work.
- Second solution:
  - the framework developer writes what he can/knows: the invocation to an abstract `factoryMethod` that must to create the instance …
  - … and asks the application developer to write, together with the subclass `B`, also the subclass `C1` of `C` that implements the `factoryMethod`.

Solutions?


C
+ m()
# factoryMethod() :A
- x
A
+ service()
m ->
...
A x;
...
x = factoryMethod();
x.service();
C1
factoryMethod() :A
<<instantiates>>
B
+ service()
Framework
App
factoryMethod() ->
return new B();

Comments

- Instead of having explicit knowledge on the creation of a class `C` …
- … the creation is delegated to a subclass `C1`.
- Reason: while writing the class `C`, the knowledge about which subclass to instantiate is missing.

Factory method [GoF]

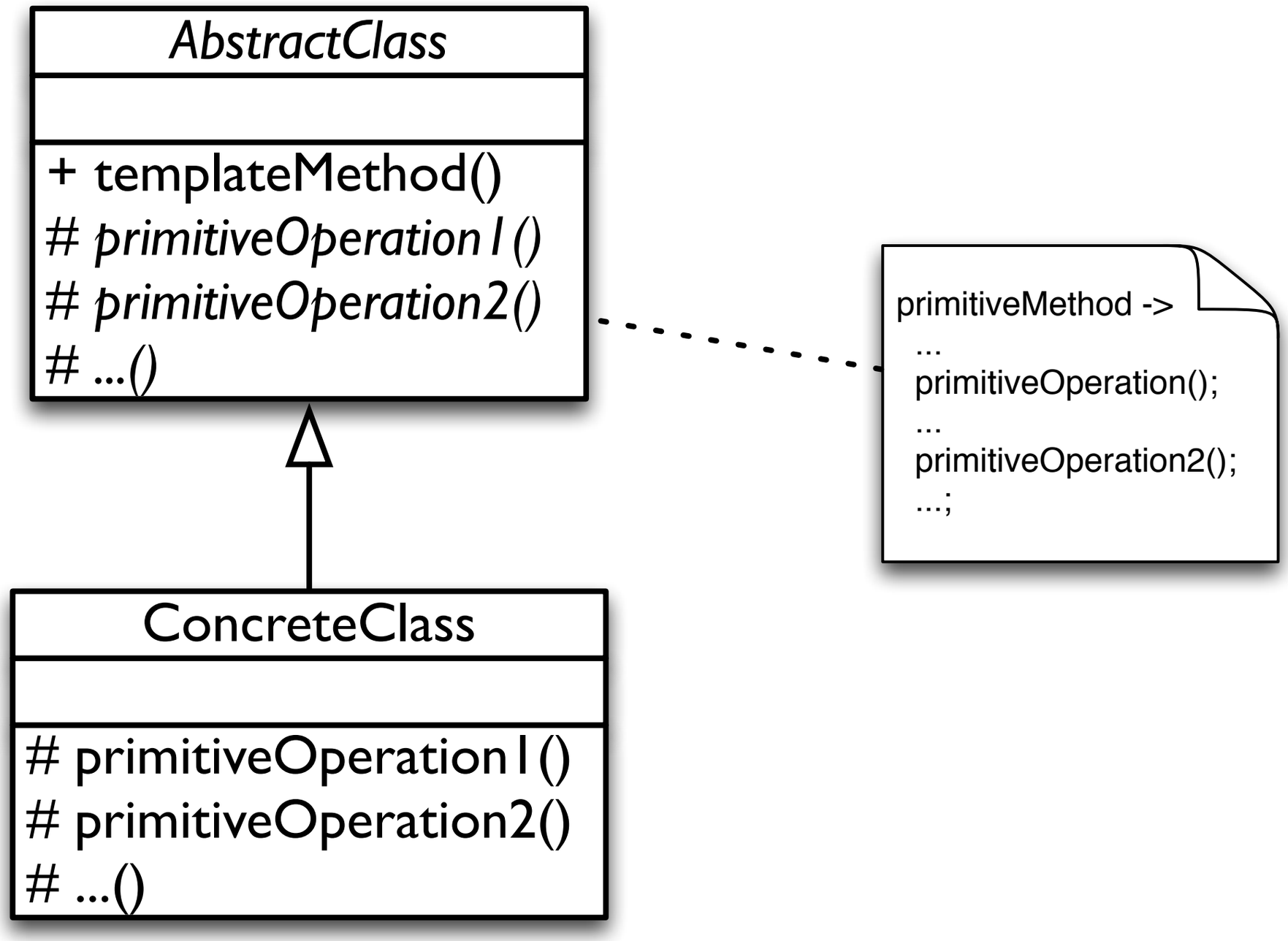
- Purpose
  - Define an interface for the creation of an object (the `factoryMethod`)…
  - … leaving to the subclasses the decision about the class that needs to be instantiated.
  - Allows to define the instantiation of a class at the subclasses level.

Template method

- Purpose
  - Define the structure of an algorithm in one method …
  - … leaving some parts undefined.
  - The implementation of the unspecified parts is contained in other methods which implementation is delegated to subclasses.

- The subclasses redefine only some parts of the algorithm, not the general structure.

Template method class diagram



Comments

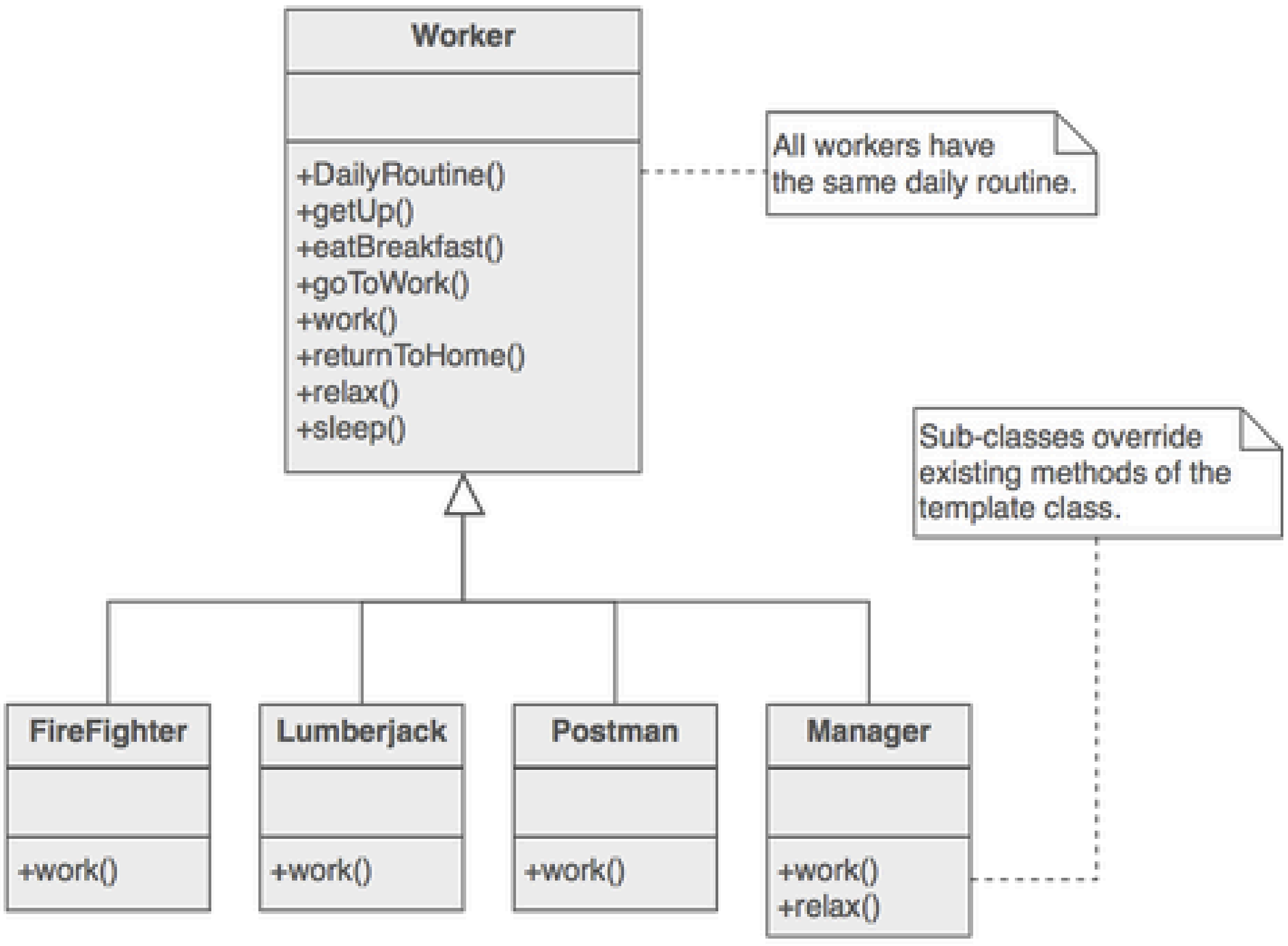
- Very useful pattern in frameworks.
- You can have many more that just one concrete subclass.
- `primitiveOperation()`
  - are also called hook methods
  - they can also be public …
  - … or they can also be stub methods (not `abstract` but `{}`)
- Avoid imposing the definitions of too many methods to the subclass(es).

Template method versus Factory method

- The basic idea is pretty similar:
  - Given a class, have its abstract methods called by another method and
  - defined in the subclass(es).

- However, they are different
  - The Template method is the one invoking the abstract methods.
  - The Factory method is the abstract method …
  - … which has to create and return the instance.

Another example

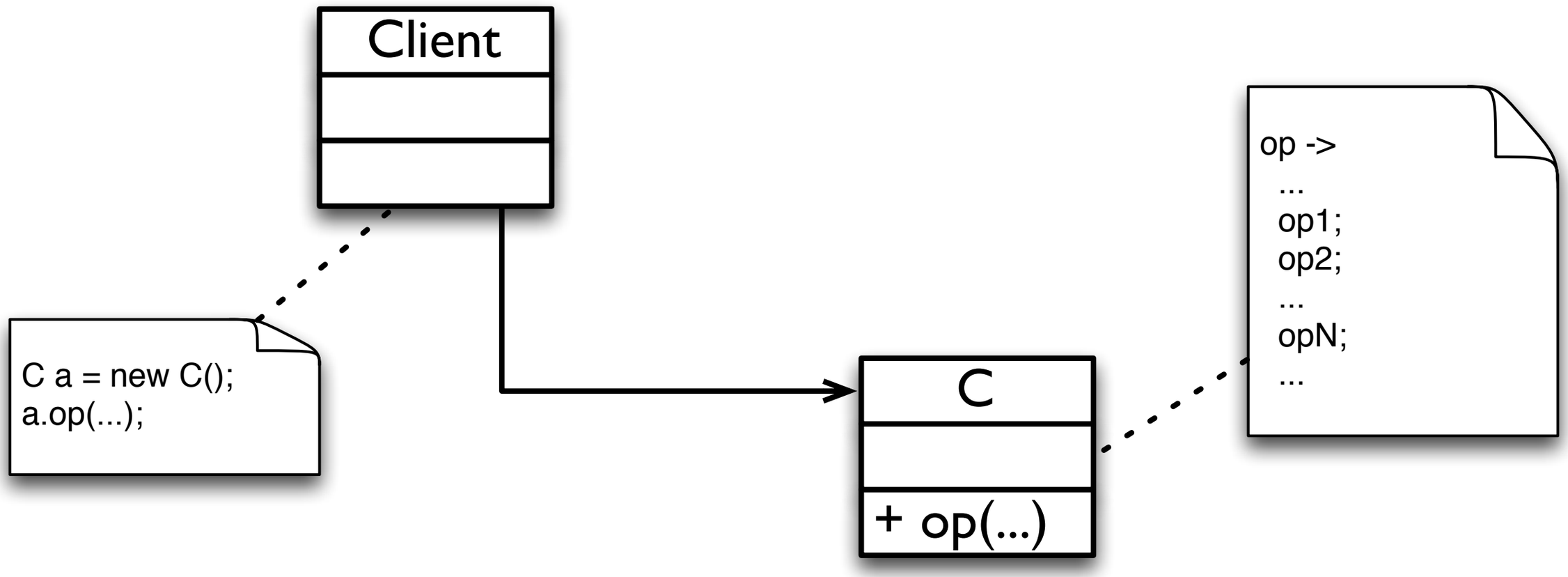


Strategy

- Purpose
  - Define a family of algorithms …
  - … encapsulate them …
  - … and make them interchangeable.
  - Allow the algorithms to vary.

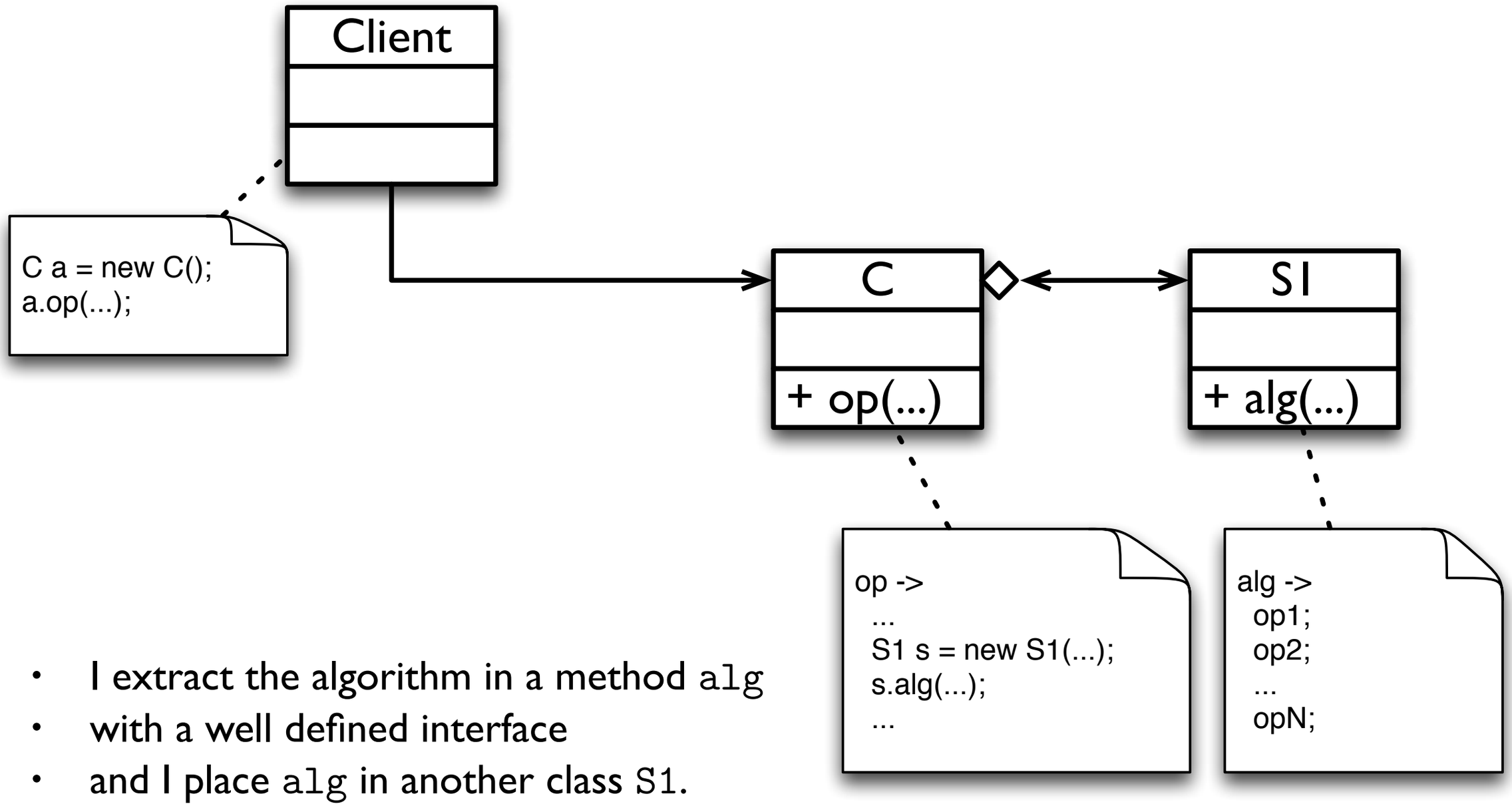
- Differently from the Template method that uses inheritance, Strategy uses composition.

Let's understand Strategy (1/6)



- The algorithm is complex and it is wired in the code of a method `op` of a class `C` (used by a `Client`).

Let's understand Strategy (2/6)



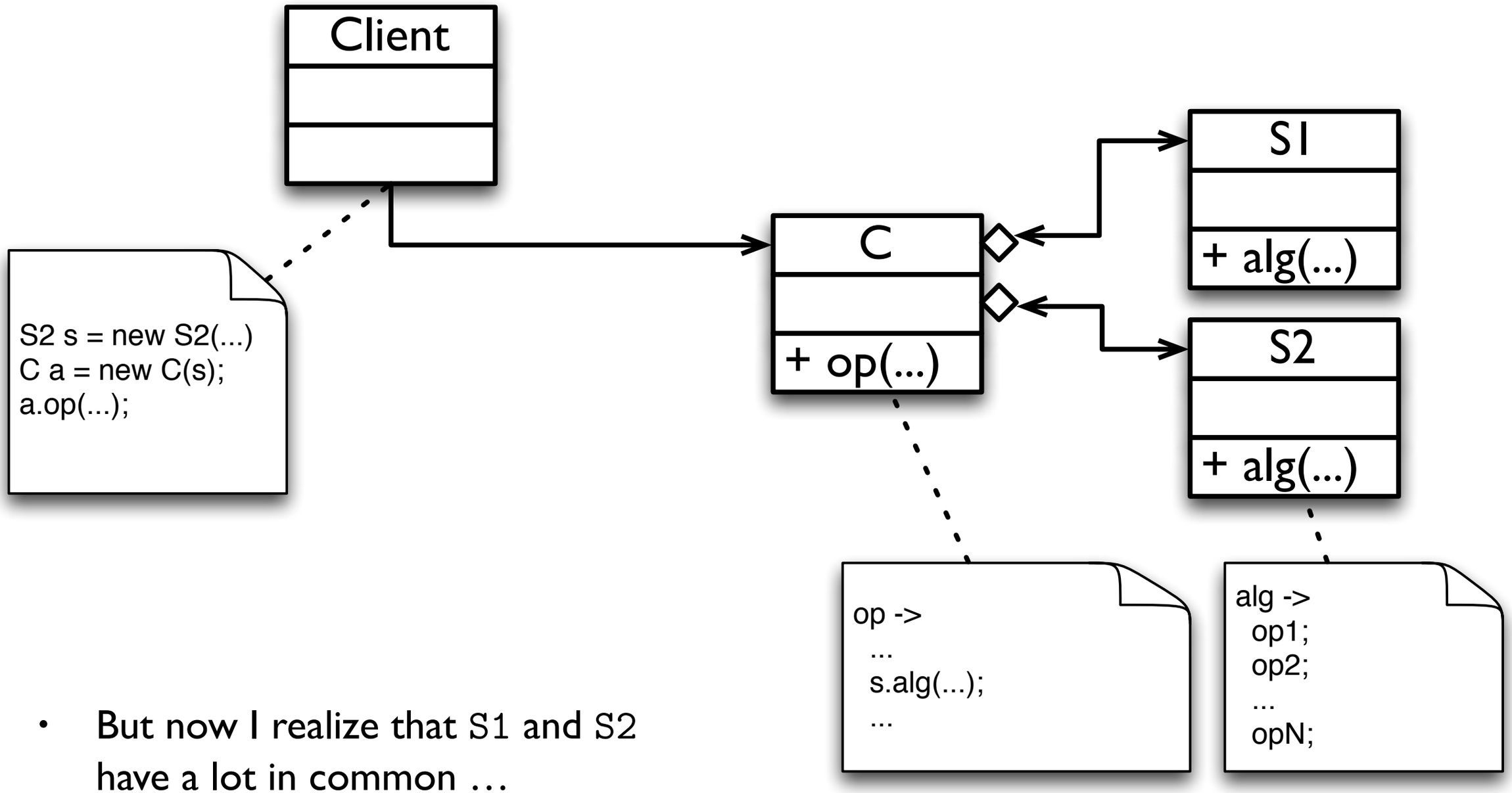
- I extract the algorithm in a method `alg`
- with a well defined interface
- and I place `alg` in another class `S1`.

Let's understand Strategy (3/6)

- To avoid loosing flexibility:
  - From `alg` I could need to access attributes/methods of `C` and local variables of op
  - that is the reason behind the navigability from `S1` to `C`.

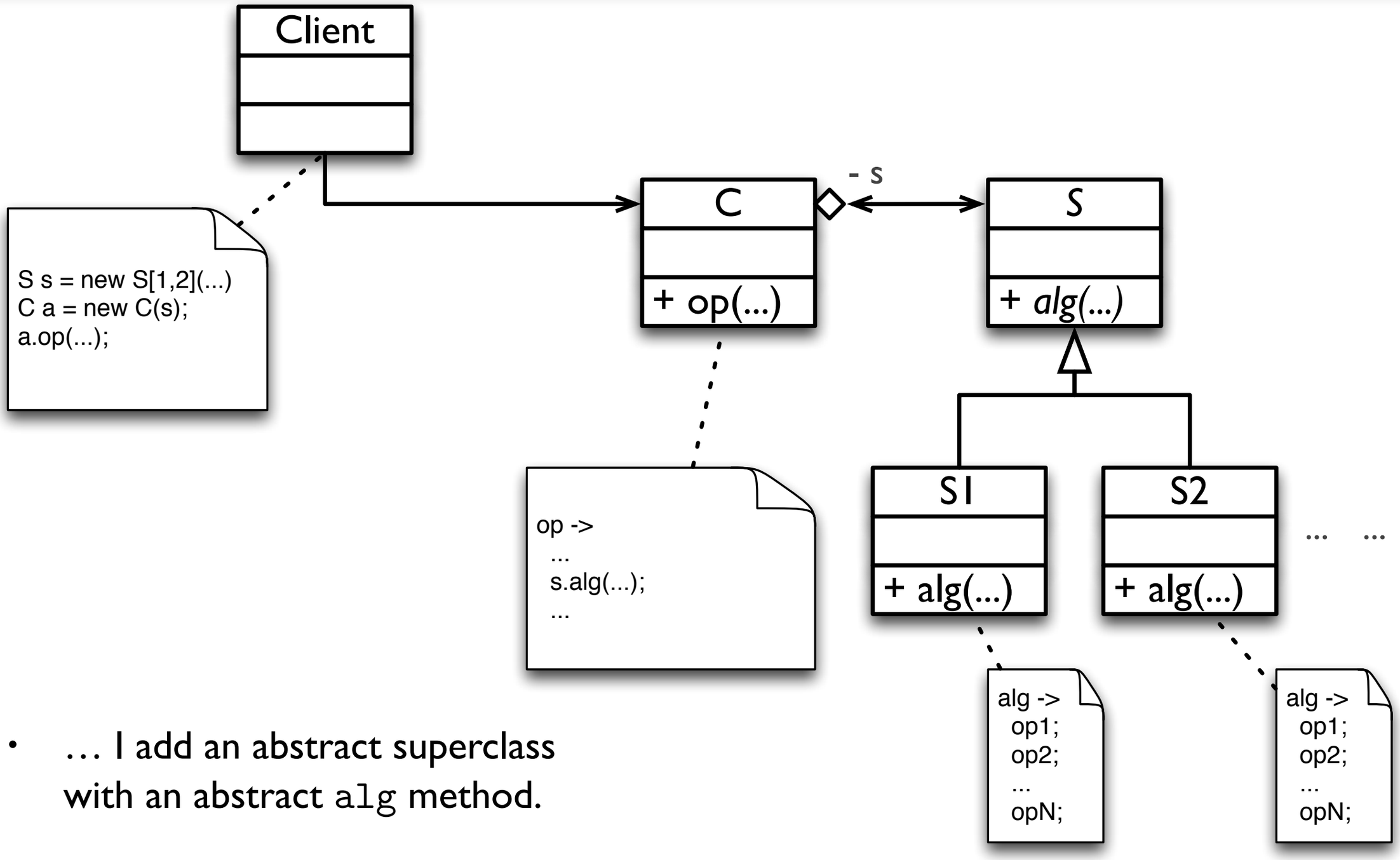
- Additional complications:
  - `C` could be configured by the `Client` to use `S1` (argument in the constructor).
  - This would allow me in the future to add `S2` with a different implementation of `alg` with very little effort.

Let's understand Strategy (4/6)



- But now I realize that `S1` and `S2` have a lot in common …

Let's understand Strategy (5/6)



- … I add an abstract superclass with an abstract `alg` method.

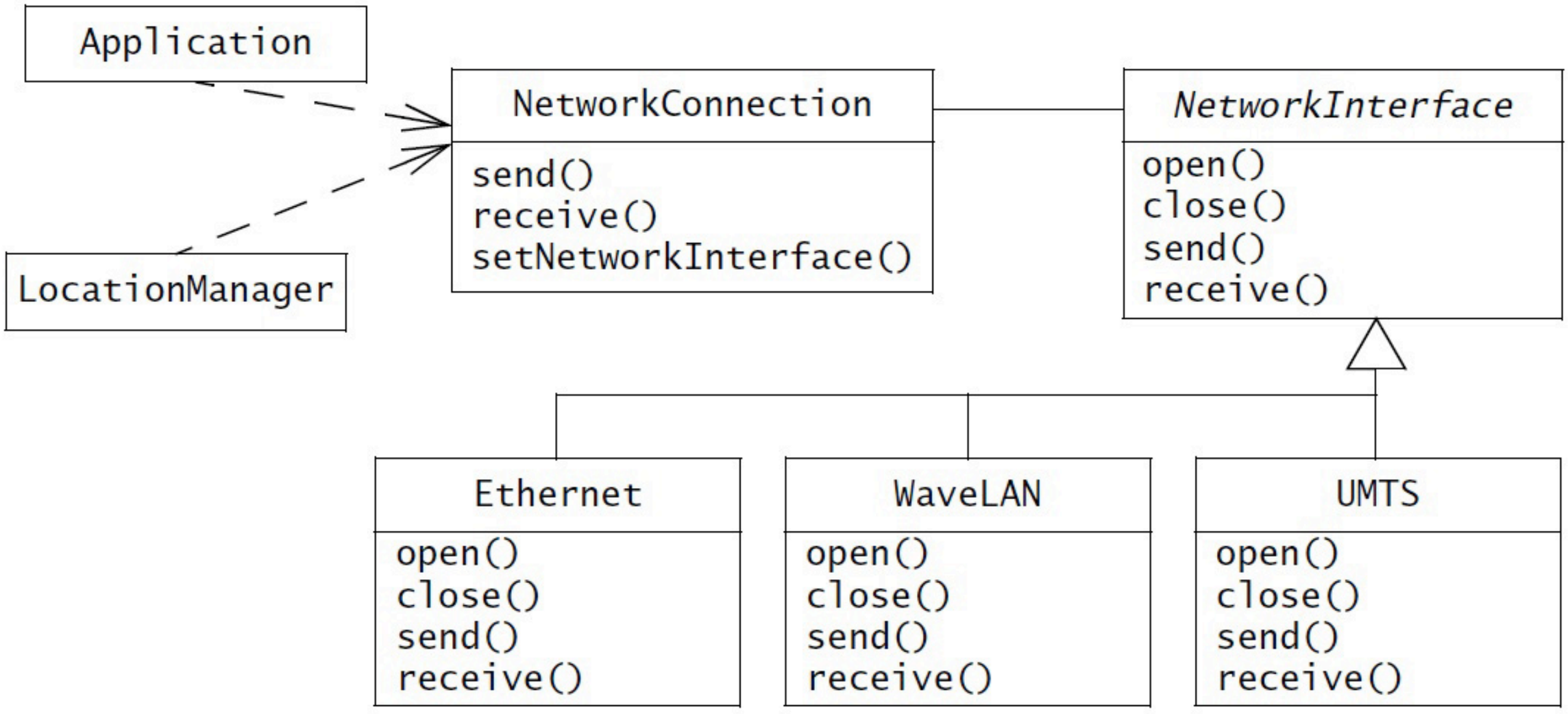
Let's understand Strategy (6/6)

- Using this approach:
  - `C` does not know about the subclasses and invokes only the base abstract class (remember that `C` could be configured by the `Client`)
  - Now—and in the future—I can add other `alg` in other subclasses
  - With very little modification, the `Client` can use another algorithm.

Comments

- The `Client` can use different algorithms with different characteristics (memory usage, speed, etc) …
  - … in a transparent manner.
- The strategy can be selected at run-time (and it would not be the case without the hierarchy generated from `C`)
- It is called Strategy not Algorithm, …
  - … strategy is a much more generic term
  - … let's see another application.

Another example



- Dynamic switching of network interfaces depending on context/location.

Concluding

Key points

- A software architecture is a description of how a software system is organized
  - mainly into components and connectors

- Architectural design decisions
  - the type of application, the distribution of the system, the architectural styles to be used.

- Architectures may be documented from several different perspectives or views
  - a conceptual view, a logical view, a process view, and a development view (4…)
  - and the use cases (…+1)

- Architectural and design patterns
  - means of reusing knowledge about generic system architectures.
  - describe the architecture, explain when it may be used
  - describe its advantages and disadvantages.

Outline

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

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

- 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 [SE9]