SYSC-3020 — Introduction to Software Engineering

Part IV - System Design

SYSC-3020 — Introduction to Software Engineering

- System design concepts
  - Definitions: Architecture (subsystem decomposition), coupling, cohesion
  - Layers and partitions
  - Software architecture (MVC, Observer pattern)
  - Process architecture (UML notation and Distribution patterns)
- System design process
  - Identify design goals
  - Initial subsystem decomposition
  - Map subsystems to components and processors
  - Persistent storage
  - Define access control policies
  - Select control flow mechanism
  - Identify boundary conditions

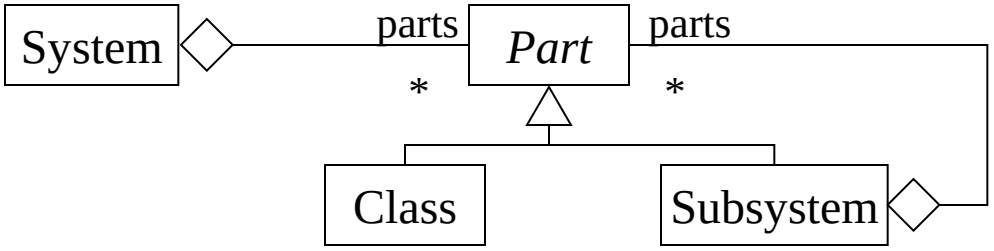
Products of System Design

- Design goals (from NF requirements)
- Software architecture: a decomposition of the system into manageable subsystems, addressing system-wide design goals.
  - e.g., from the Analysis class diagram, additional subsystems, etc.
  - Given in terms of subsystem responsibilities, dependencies among subsystems, mapping to hardware
  - Policy decisions: data storage and management, access control, etc.
- Boundary use cases, i.e., startup, shutdown, exception handling issues

Definitions

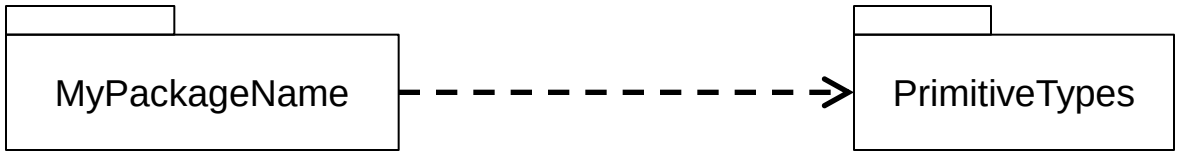
- Subsystem architecture: A system structure is composed of interconnected subsystems
- Subsystem: A collection of program parts (e.g., classes) that constitutes a whole and has well-defined responsibilities.
- Objectives:
  - Reduce complexity by separating concerns
  - Allow multiple teams to work concurrently
  - Reuse existing components

- Subsystems and classes

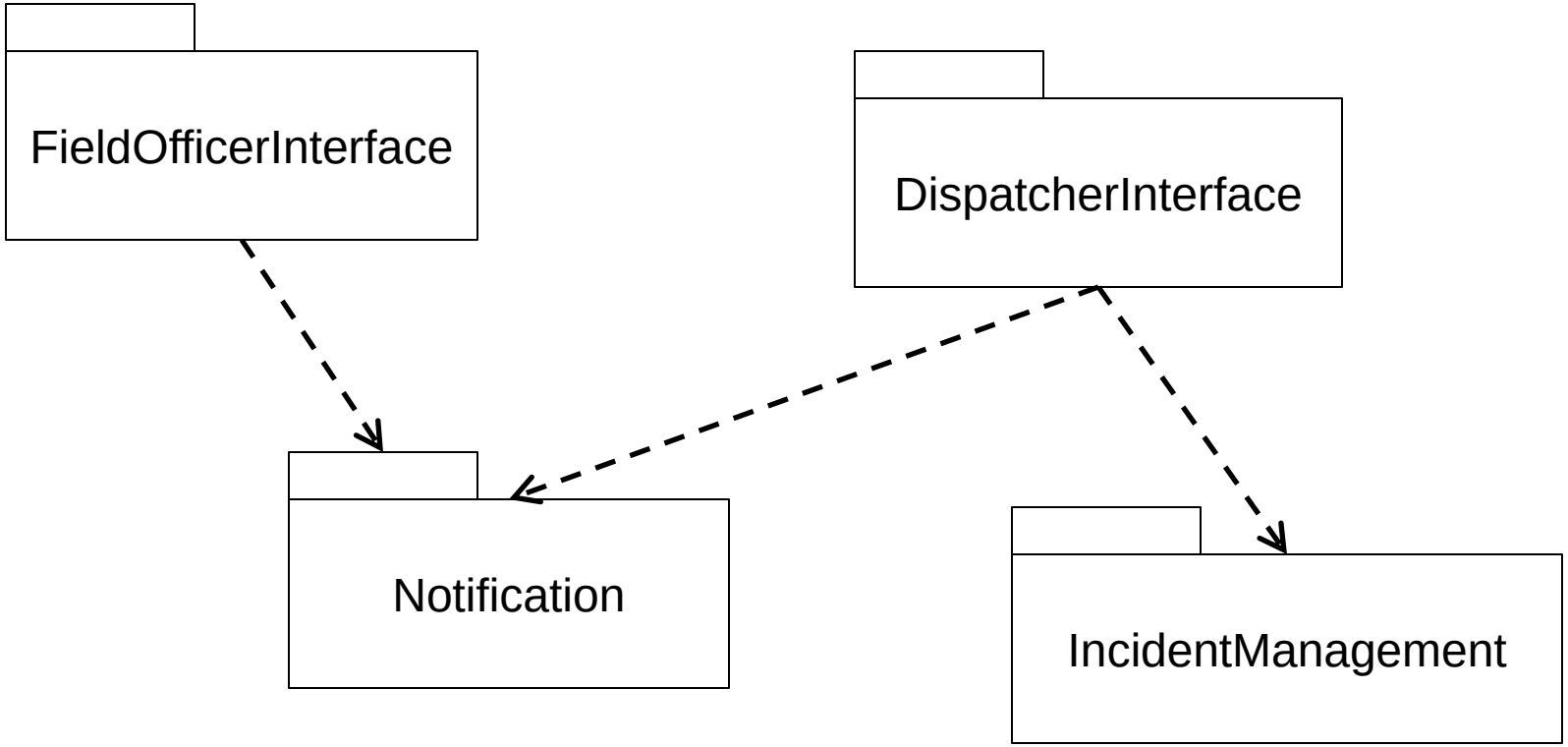


UML

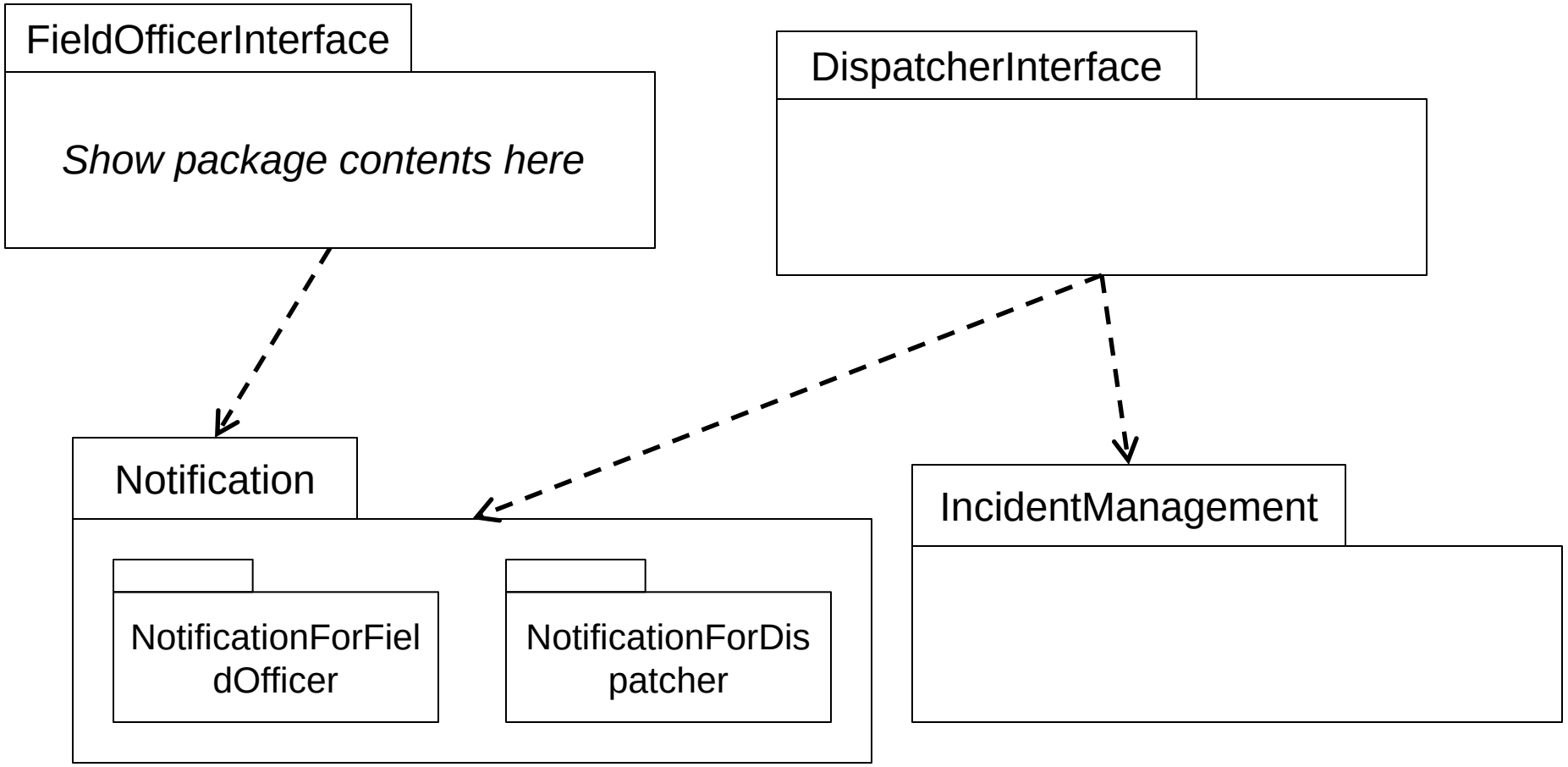
- We use packages to represent subsystems
  - a package is a collection of modeling elements that are grouped together because they are logically related.
  - establishes a namespace
- Package diagram to represent subsystems (packages) relationships
- One main package relationship: dependency
  - Elements (usually classes) of the source package can legally refer to public elements of the target package
  - If there is an association between class A of package P and class B of package Q, then there must be a dependency between the two packages (direction depends on how the association is actually used).



FRIEND Example



FRIEND Example



Subsystem Services and Interfaces

- We define subsystems in terms of the *services* they offer
- A service is a set of related *operations* that share a common purpose, e.g., store and retrieve data in database, provide communication services

- The set of public operations forms the *subsystem interface* or *application programmer interface* (API).
  - Includes their parameters, types, and return values.
  - Providing the contracts of the subsystem public operations is also a good practice as this will ease the interfacing between subsystem teams.
  - Subsystem interface should provide as little information as possible about its inner elements – minimize dependencies and impact of change.

Subsystem Properties

- We want highly *cohesive* classes within each subsystem and loosely *coupled* subsystems.
- This has an impact on comprehensibility, maintainability, the ease with which teams can work concurrently on subsystems, the integration of subsystems, etc.
- *Coupling:* A measure of how closely two classes or subsystems are connected
- *Cohesion:* A measure of how well a class or subsystem is tied together

- Note: the next 5 slides are for illustration purposes, showing that one can actually compute (and have a value of) coupling and cohesion for a subsystem (or piece of code). The details of the formalization are not to be memorized.

Coupling

- Connections among classes/subsystems: aggregation, specialization, calling public operations

1. Outside coupling: A class or a subsystem refers directly to the public members of another class or subsystem.
2. Inside coupling: An operation refers directly to other, private members in the same class.
3. Coupling from below: A specialized class refers directly to (protected) members in the super class.
4. Sideways coupling: A class refers directly to private properties in another class (e.g., Friends in C++).

Cohesion

- Classes:
  - Operations constitute a functional whole
  - Attributes and data structures describe objects in well defined states, which are modified by operations
  - Operations use each other
- Subsystems:
  - Subsystems classes are conceptually related
  - Structural relations among classes are primarily generalizations and aggregations (form clusters)
  - Key operations can be carried out within the subsystem

Measure of Coupling

- Method–Attribute Dependency (DMA).
  - A direct method–attribute dependency exists between method m and attribute a (not necessarily of the same class) if m directly accesses a.
  - This is denoted *DMA(m,a)*.
- Method–Method Dependency (DMM).
  - A direct method–method dependency exists between method m1 and method m2 (not necessarily of the same class) if m1 invokes m2 (possibly a polymorphic version of m2).
  - This is denoted *DMM(m1,m2).*
- Method–Attribute Interactions (MAI).
  - For two classes c1 and c2, MAI(*c*1,*c*2) is the set of DMA between a method of c1 and an attribute of c2.
- Set of Method–Method Interactions (MMI).
  - For two classes c1 and c2, MMI(c1,c2) is the set of DMM between a method of c1 and a method of c2.

Measure of Coupling - continue

- Method–Attribute Coupling (MAC).
  - For a given class *c*, *MAC*(*c*) counts all MAI from *c* to classes that do not have a common ancestor with *c*.
- Method–Method Coupling (MMC).
  - For a given class *c*, *MMC*(*c*) counts all MMI from *c* to classes that do not have a common ancestor with *c*.

$$MAI(c_1, c_2) = \bigcup_{m \in M(c_1)} \{(m, a) \mid a \in A(c_2) \wedge DMA(m, a)\}$$

$$MMI(c1, c2) = \bigcup_{m \in M(c1)} \bigcup_{m' \in M(c2)} \{(m, m') \mid DMM(m, m')\}$$

$$MAC(c_1) = \sum_{c_2 \in Others(c_1)} |\, MAI(c_1, c_2) \,|$$

$$MMC(c_1) = \sum_{c_2 \in Others(c_1)} |\, MMI(c_1, c_2) \,|$$

Measure of Cohesion

- Lack of Cohesion Of Methods (LCOM)
  - I_1, … I_n the sets of instance variables used by methods M_1, …, M_n
  - $P=\{(I_i,I_j)|I_i \cap I_j=\varnothing\}$ and $Q=\{(I_i,I_j)|I_i \cap I_j\neq\varnothing\}$
  - LCOM = $|P|-|Q|$ if $|P|>|Q|$ and 0 otherwise
  - A high value suggests that the methods in the class are not really related to each other nor, therefore, to a single interface

Subsystem Implementation

- Several programming languages provide constructs for implementing subsystems
- E.g., Packages in Java and Ada
- C, C++ does not model explicitly subsystems – conventions such as grouping subsystem files in a directory
- A common solution has to be found as different teams will likely develop different subsystems

SYSC-3020 — Introduction to Software Engineering

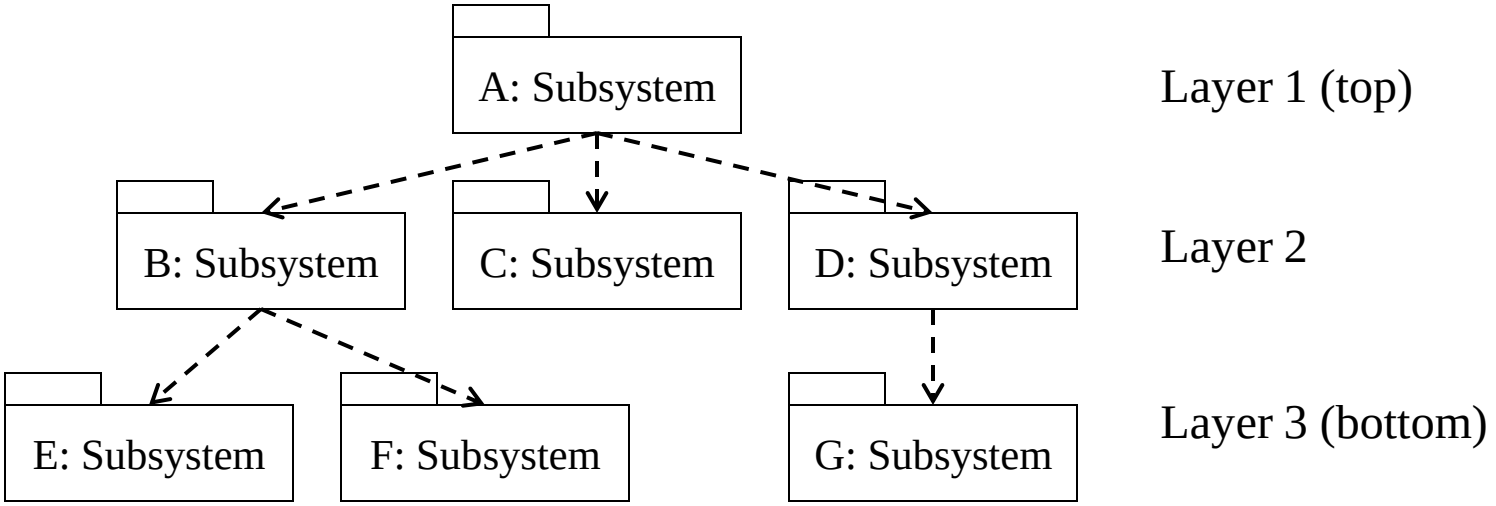
- System design concepts
  - Definitions: Architecture (subsystem decomposition), coupling, cohesion
  - Layers and partitions
  - Software architecture (MVC, Observer pattern)
  - Process architecture (UML notation and Distribution patterns)
- System design process
  - Identify design goals
  - Initial subsystem decomposition
  - Map subsystems to components and processors
  - Persistent storage
  - Define access control policies
  - Select control flow mechanism
  - Identify boundary conditions

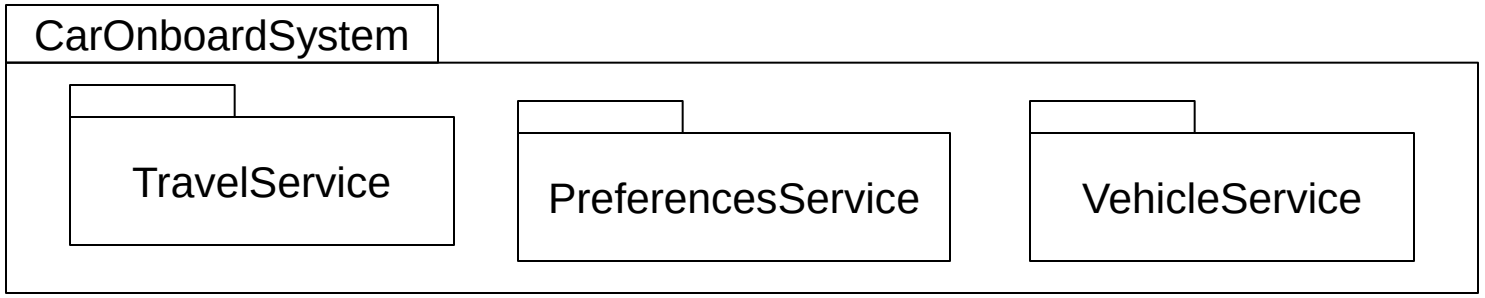
Layers and Partitions

- Systematically applying decomposition of subsystems leads to a hierarchical structure of subsystems or ***layers***
  - A layer is a grouping of subsystems providing related services
  - Services possibly realized using services from another layer
  - A system is rarely decomposed in more than 3 to 5 layers (rule of thumb)

- ***Partitions*** result from decomposition into peer subsystems
  - Each peer subsystems is responsible for a different class of services.
  - Each subsystem depends loosely on the others
  - Each subsystem may operate in isolation

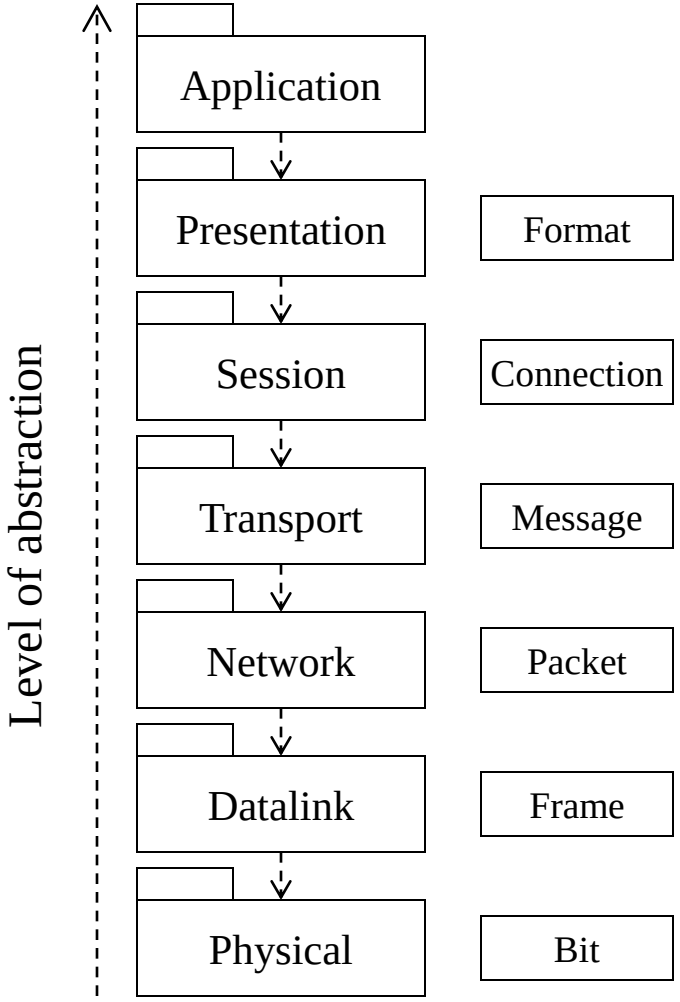
Layers and Partitions





CarOnboardSystem is decomposed into three partitions

OSI Model


Level of abstraction
Application
Presentation
Format
Session
Connection
Transport
Message
Network
Packet
Datalink
Frame
Physical
Bit

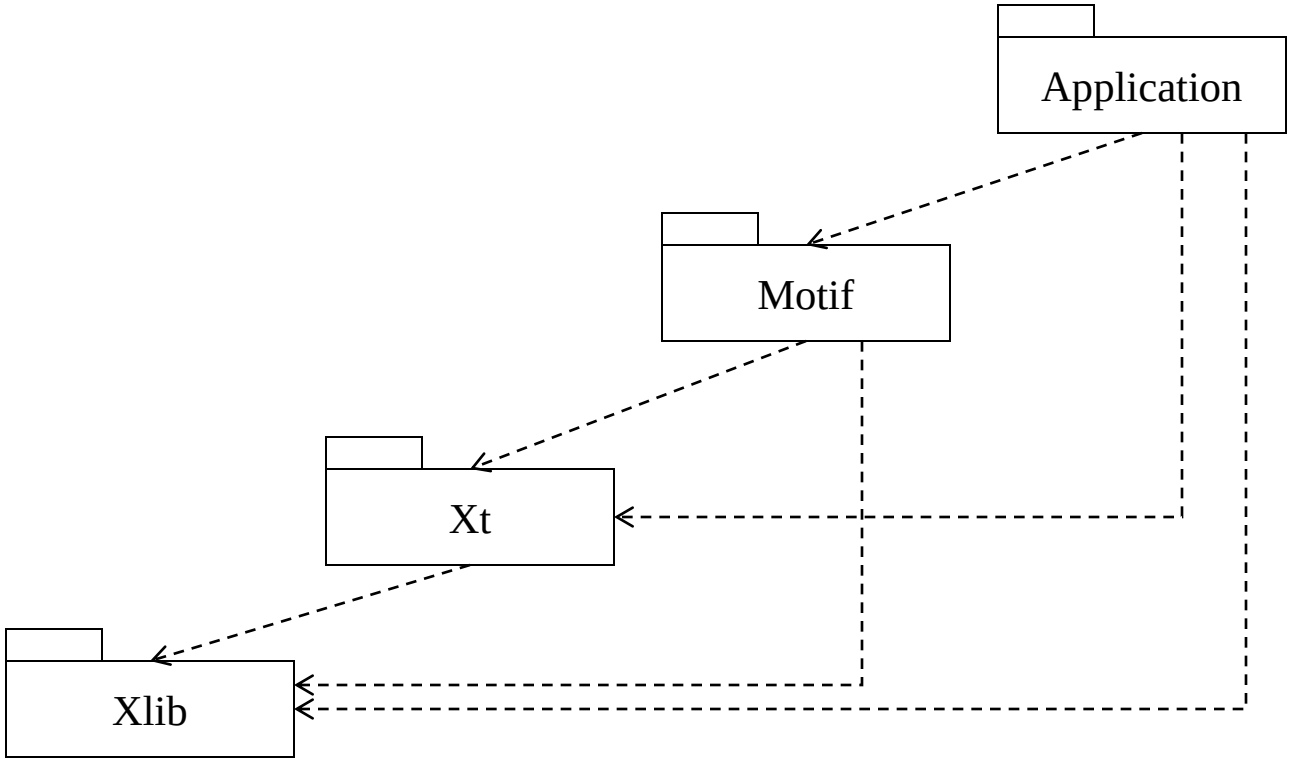
Layered Architectures

- Two axis for describing layered architectures:
  - **Closed**: A given layer (I) can only use operations from the layers immediately adjacent (I+1, I-1)
    - If not closed, it is **Open**: Any layer (at distance more than one) can be accessed
  - **Strict**: Operations in layers can only use operations in layers below.
    - If not strict, it is **Relaxed**: use layers above and below
- This results in 4 combinations:
  - Closed – Strict
  - Open – Strict
  - Open – Relaxed
  - Closed – Relaxed
- Discussion: (From Breugge)
  - Closed applications are highly maintainable and flexible, portable
  - Open applications are (runtime) efficient

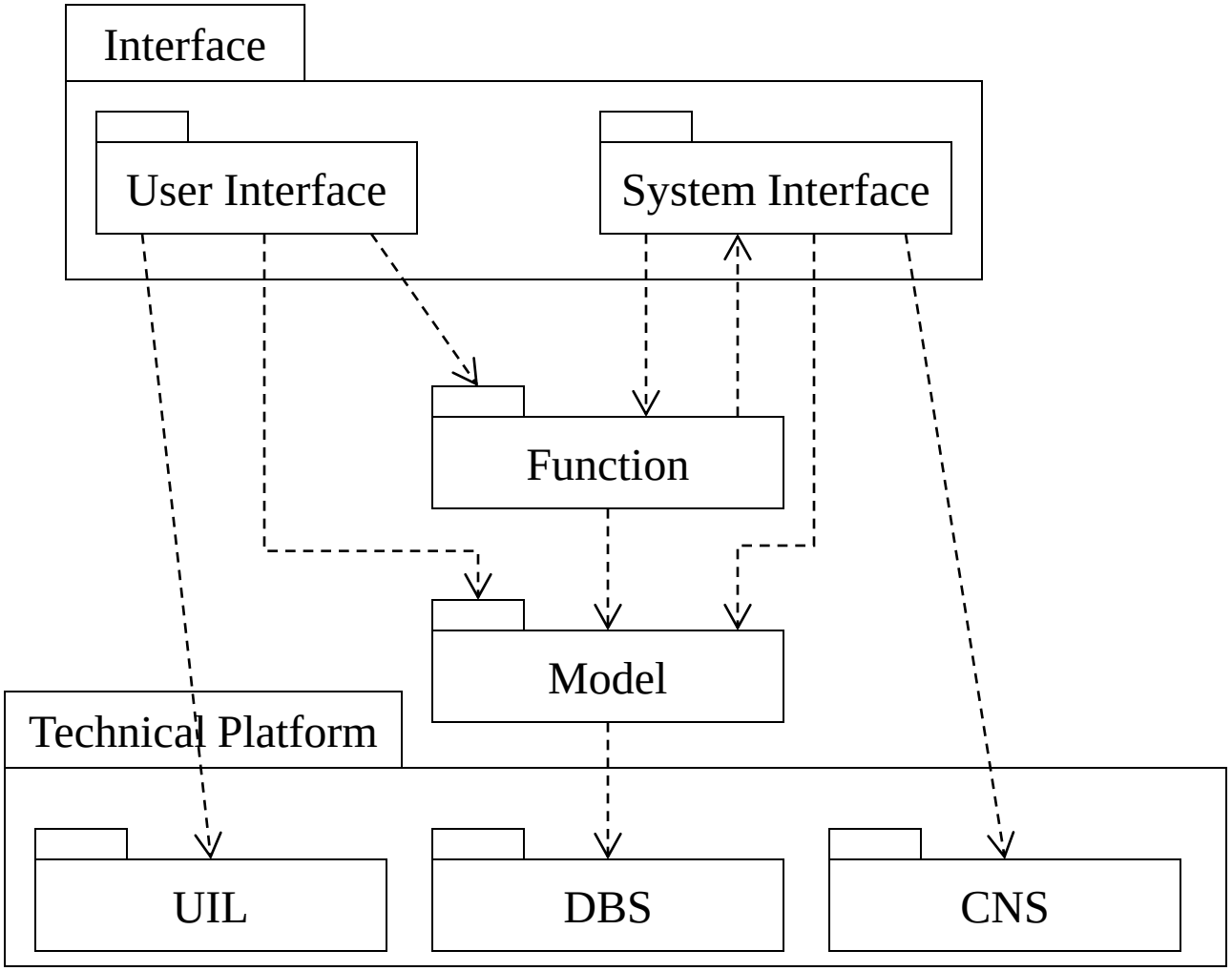
Layered Architectures (cont.)

- Closed – Strict:
  - Simple dependencies, easier changes
  - Results in many operations that merely transmit calls to the lower level
- Open – strict:
  - More efficient (no delegations from layer to layer)
  - Difficult to change layers at lower levels, as it can lead to changes to all layers above
  - Must look at the number of layers to decide
- Open – Relaxed:
  - Simple subsystems interfaces at the expense of complex dependencies
- Closed – Relaxed:
  - Potential for the highest level of inter-dependencies, resulting in low maintainability.

Motif

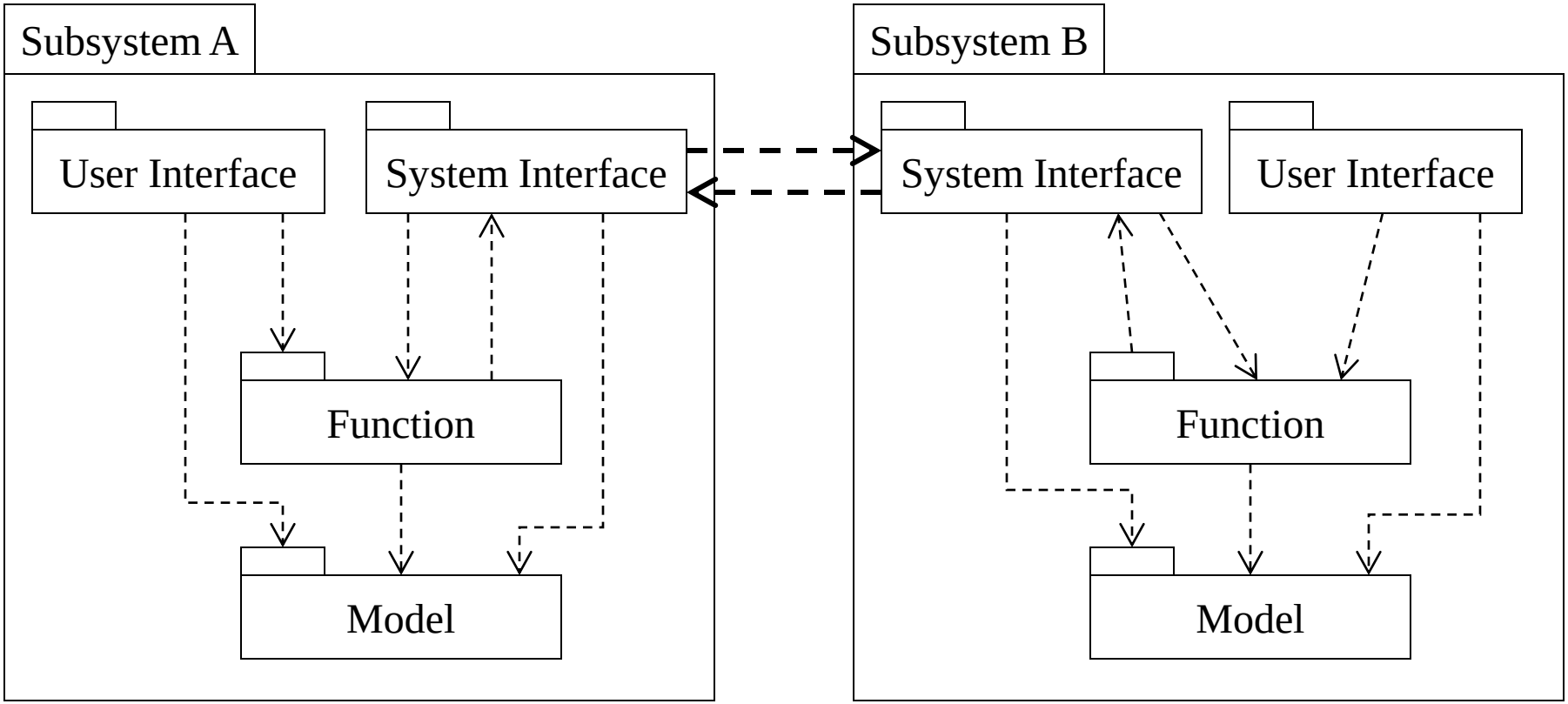


Basic Layer Pattern (Small System)



Basic Layer Pattern (Large Systems)

- Simple systems: Basic Layer Pattern
- For large systems, you must further decompose the system
- Several independent subsystems that communicate with each other
- Each subsystem is like an independent system with its own Model, Function, and Interface subsystems.
- The System interface subsystem provides a coherent interface to other subsystems for accessing the given subsystem's functionality.


Subsystem A
User Interface
System Interface
Function
Model
Subsystem B
System Interface
User Interface
Function
Model

SYSC-3020 — Introduction to Software Engineering

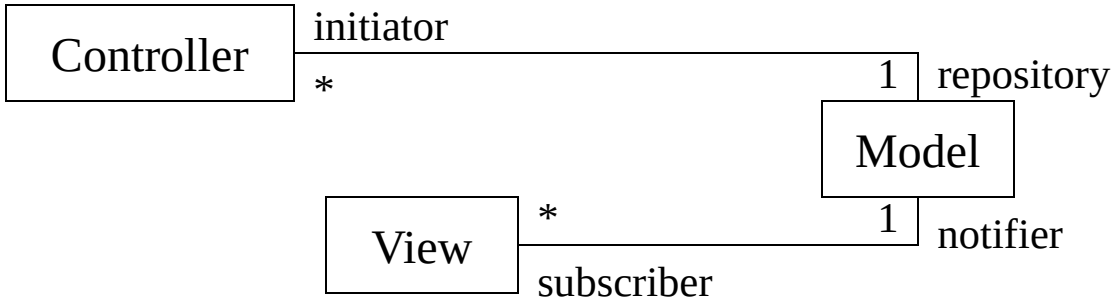
- System design concepts
  - Definitions: Architecture (subsystem decomposition), coupling, cohesion
  - Layers and partitions
  - Software architecture (MVC, Observer pattern)
  - Process architecture (UML notation and Distribution patterns)
- System design process
  - Identify design goals
  - Initial subsystem decomposition
  - Map subsystems to components and processors
  - Persistent storage
  - Define access control policies
  - Select control flow mechanism
  - Identify boundary conditions

Architectural Styles

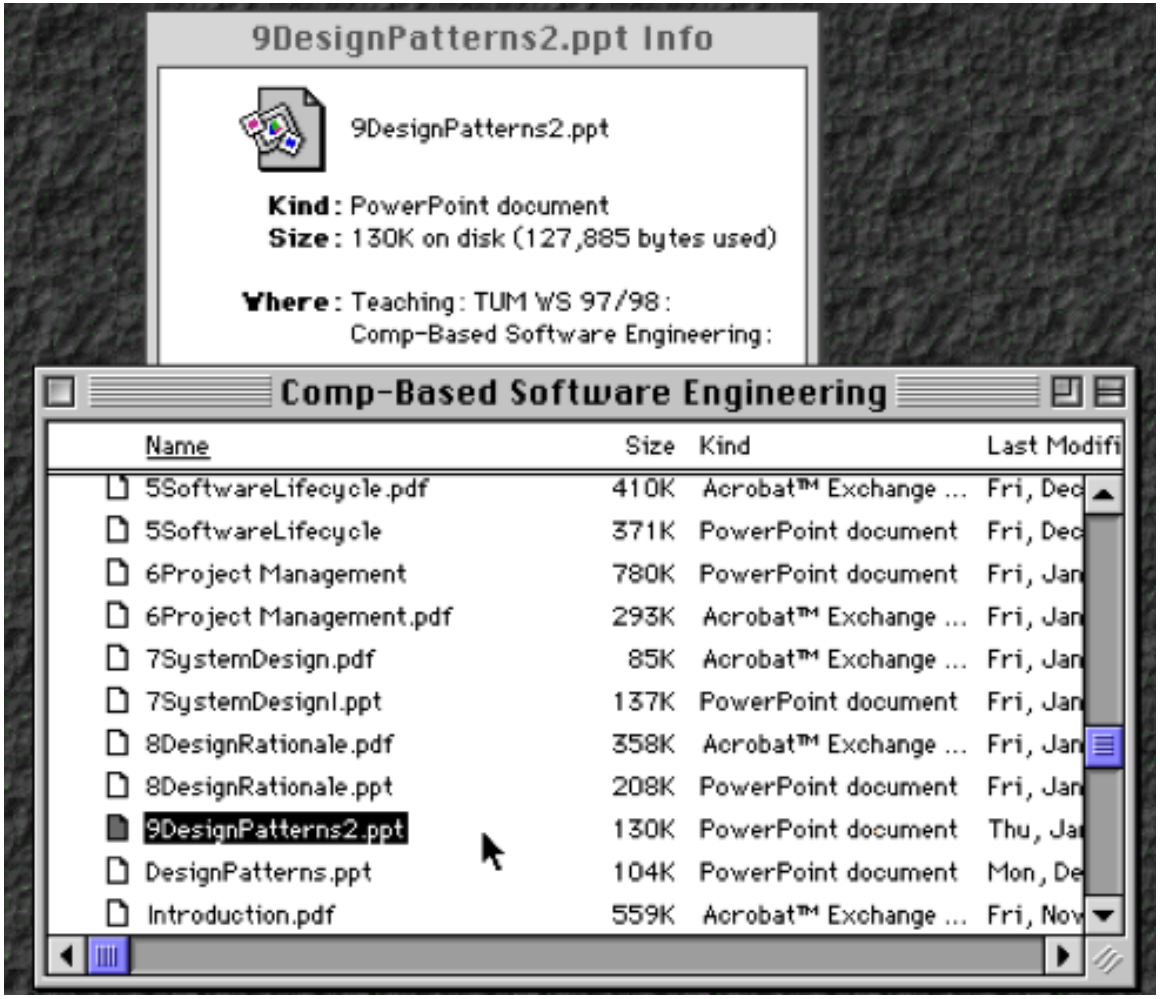
- Large software systems are complex
- General architectural solutions have been devised
  - Architectural styles
- See discussion in textbook
- To name a few styles:
  - Repository: subsystems access and modify a single data structure (in a dedicated subsystem)
  - Model/View/Controller (see next)
  - Client/server: the server subsystem provides services to client subsystems
  - Peer-to-peer: two subsystems act both as client and server
  - Three-tier: interface layer, application-logic layer, storage layer
  - …
  - Styles can be combined

Model/View/Controller (MVC)

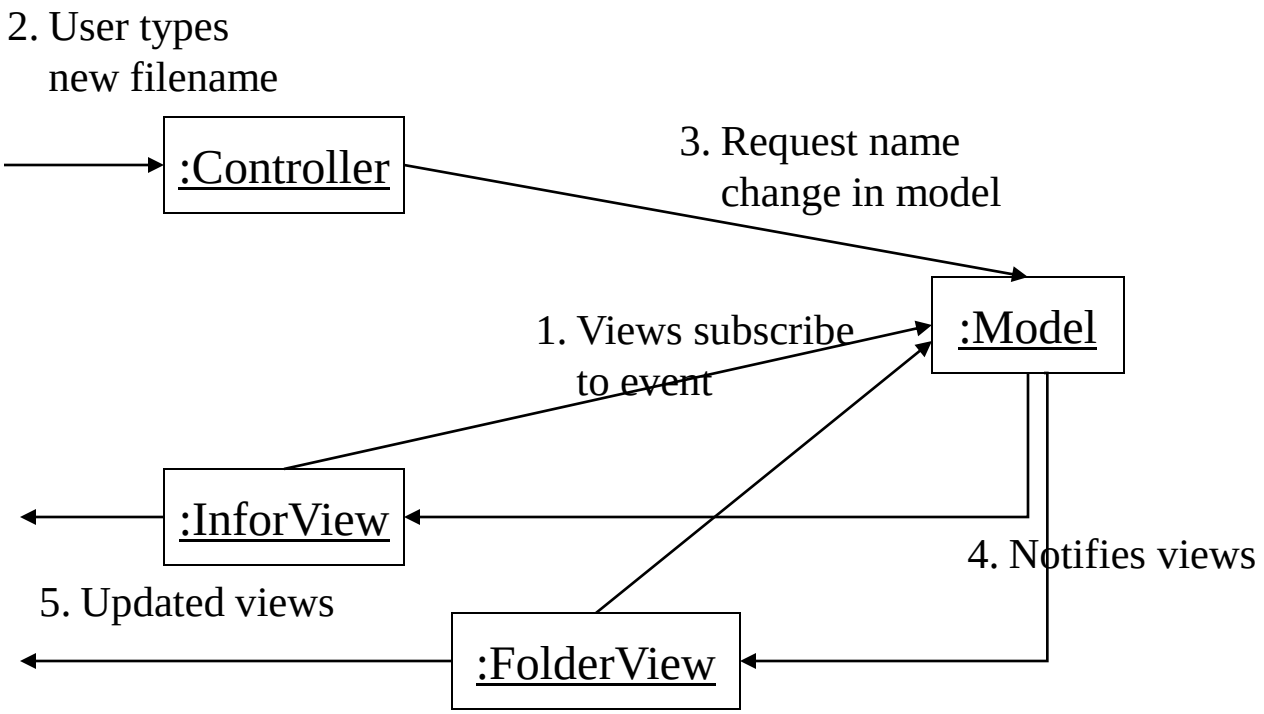
- As in the basic layer pattern, 3 types of subsystems:
  - Model: domain knowledge
  - View: display
  - Controller: determine sequences of user interactions
- Model subsystem does not depend on the other two
- Model ~ data repository, Controller -> control flow
- Change in model subsystem(s) is propagated to view subsystem



MVC Example


9DesignPatterns2.ppt Info
9DesignPatterns2.ppt
Kind: PowerPoint document
Size: 130K on disk (127,885 bytes used)
Where: Teaching: TUM WS 97/98:
Comp-Based Software Engineering:
Comp-Based Software Engineering
Name
Size
Kind
Last Modifi
5SoftwareLifecycle.pdf 410K Acrobat™ Exchange ... Fri, Dec
5SoftwareLifecycle 371K PowerPoint document Fri, Dec
6Project Management 780K PowerPoint document Fri, Jan
6Project Management.pdf 293K Acrobat™ Exchange ... Fri, Jan
7SystemDesign.pdf 85K Acrobat™ Exchange ... Fri, Jan
7SystemDesignI.ppt 137K PowerPoint document Fri, Jan
8DesignRationale.pdf 358K Acrobat™ Exchange ... Fri, Jan
8DesignRationale.ppt 208K PowerPoint document Fri, Jan
9DesignPatterns2.ppt 130K PowerPoint document Thu, Ja
DesignPatterns.ppt 104K PowerPoint document Mon, De
Introduction.pdf 559K Acrobat™ Exchange ... Fri, Nov

MVC Sequence of Events



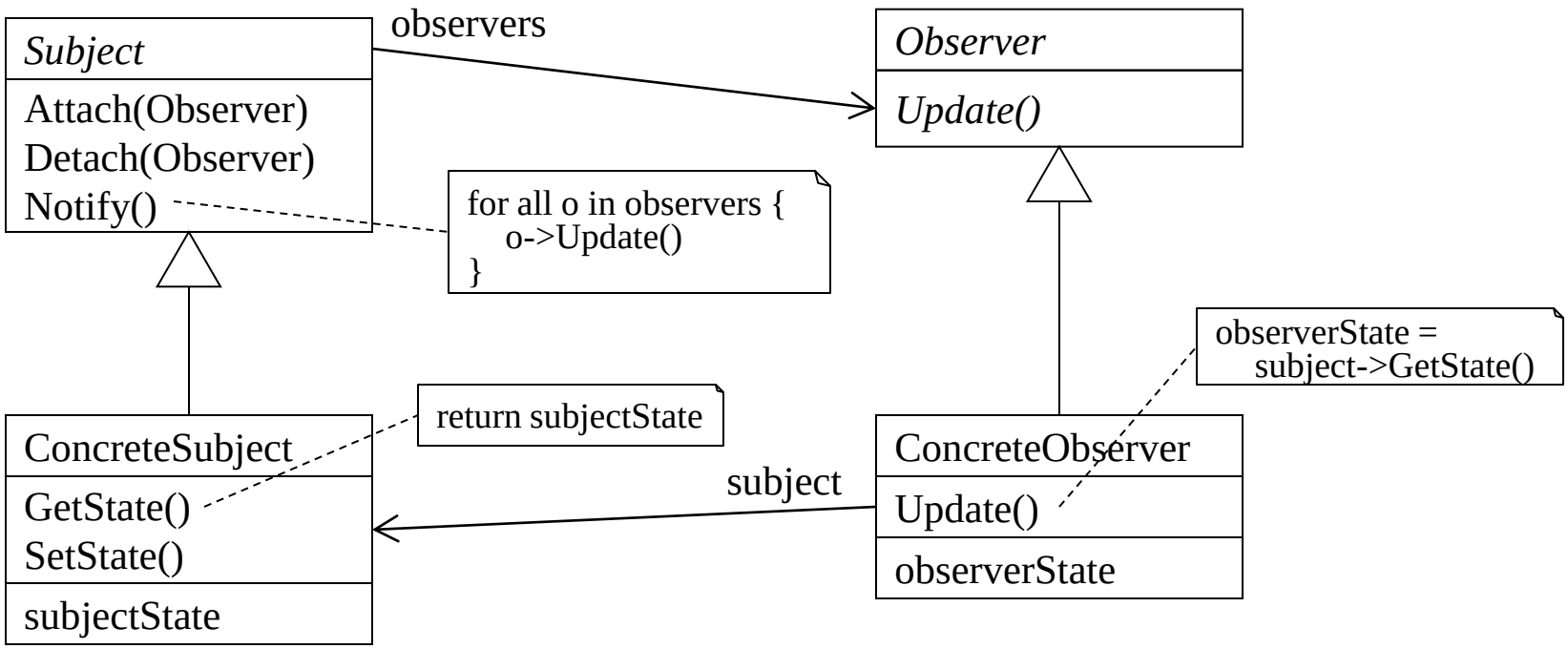
Implementation

- View and controller are more subject to change than Model – limit effect of changes in user interface
- Views can be added without any change to the Model subsystem(s)
- MVC well suited for interactive systems, when multiple views are needed
- Also useful to maintain consistency across distributed data
- Model may present performance bottlenecks
- *Observer design pattern* is usually used to implement the subscription and notification – decouple the View and Model objects

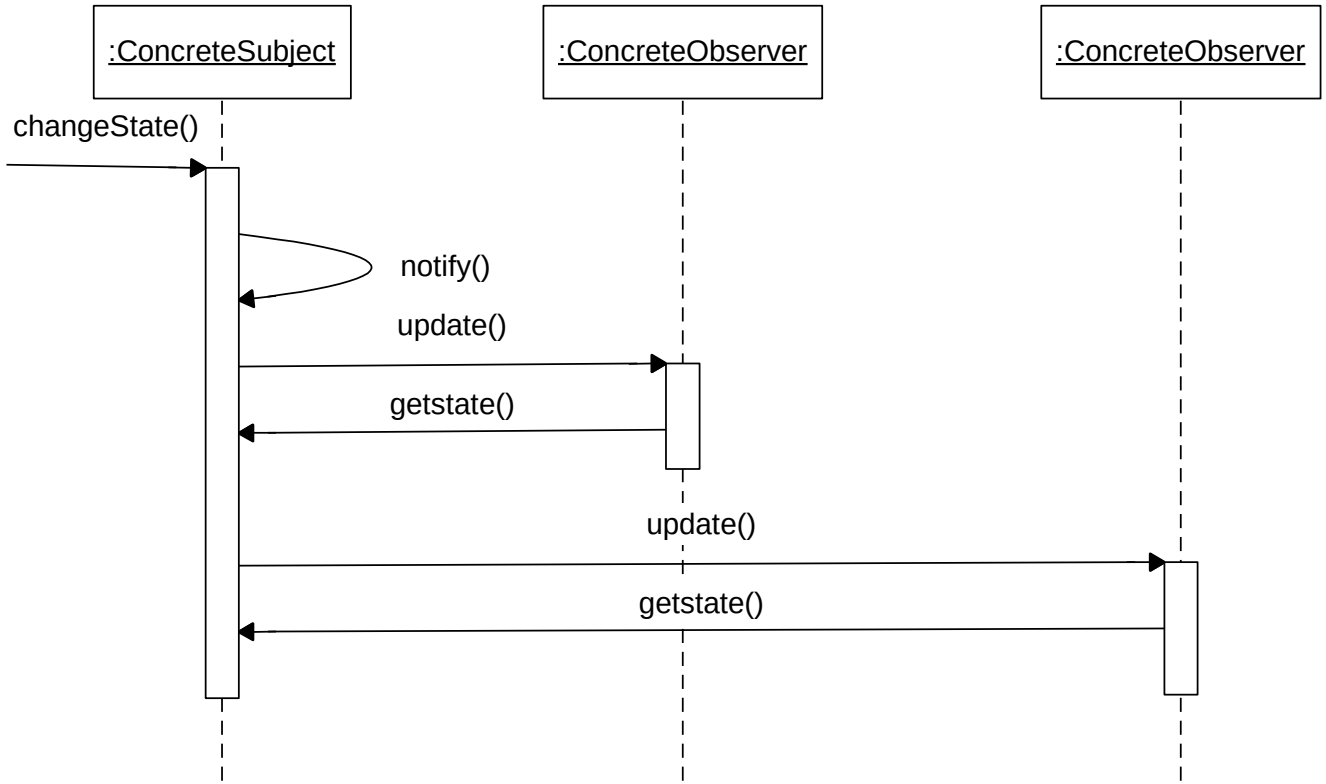
Observer Design Pattern

- **Intent:** When one object changes state, all its dependents (observers) are notified and updated automatically. A common side-effect of partitioning a system into a collection of cooperating classes is the need to maintain consistency between related objects.
- **Applicability**
  - When a change to one object requires changing others, and you don't know at design time what objects need to be changed.
  - E.g., Entity objects and views on them, Maintain consistency across classes in the system
  - An object should be able to notify other objects without making assumptions about who these objects are. In other words, you don't want these objects tightly coupled.

Observer Pattern Structure



Observer Sequence Diagram



MVC—Discussion

MVC allows an object to be informed when another object changes

(view)	(model)
Boundary object	Entity object
(need to display accurate data from data model)	
Entity object	Entity object
(to maintain consistency in data model)	
Control object	Control object
(some actions have to be performed one after the other, e.g., when state changes)	
Boundary object	Control object
(need to display accurate data about the status of some action)	

Sample Questions from Text

- Review the MVC example and discuss how the MVC architecture helps or hurts the following design goals
  - Extensibility (eg. the addition of new types of views)
  - Response time (eg. the time between a user input and the time all views have been updated)
  - Modifiability (eg. the addition of new attributes in the model)

SYSC-3020 — Introduction to Software Engineering

- System design concepts
  - Definitions: Architecture (subsystem decomposition), coupling, cohesion
  - Layers and partitions
  - Software architecture (MVC, Observer pattern)
  - Process architecture (UML notation and Distribution patterns)
- System design process
  - Identify design goals
  - Initial subsystem decomposition
  - Map subsystems to components and processors
  - Persistent storage
  - Define access control policies
  - Select control flow mechanism
  - Identify boundary conditions

Process Architecture

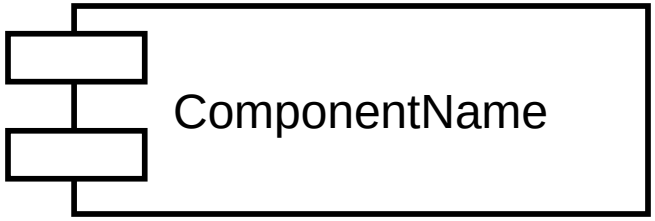
- Focus on distribution of processes and objects as opposed to classes and subsystems
- SW/HW mapping has a strong impact on performance and system complexity
- This is easy if we design a stand-alone administrative system but is not obvious for monitoring and control systems, embedded systems, etc.
- Process architecture: A system-execution structure composed of interdependent processes
- Objective: avoid execution bottlenecks
- This activity can be delayed until the subsystem architecture is complete.

UML Components

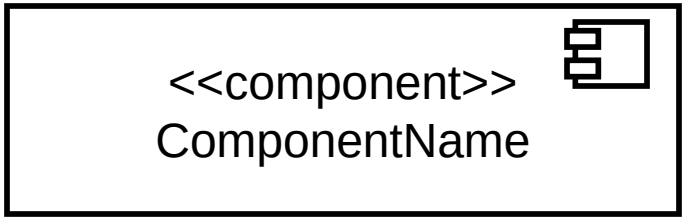
- In the context of UML, we say we distribute *components* to *nodes*.
- The definition of a component depends on the environment of the system
  - May comprise files of source code
  - But also executable, database systems, libraries, web pages, etc.
- A node is a processor available that can execute some of the system processes, e.g., on the network.

- In the simplest case, each subsystem executed as a component (or several) which are distributed on nodes

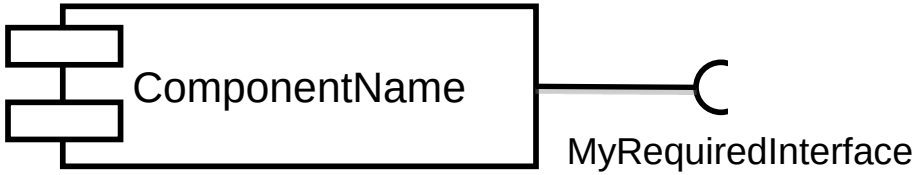
UML Notation



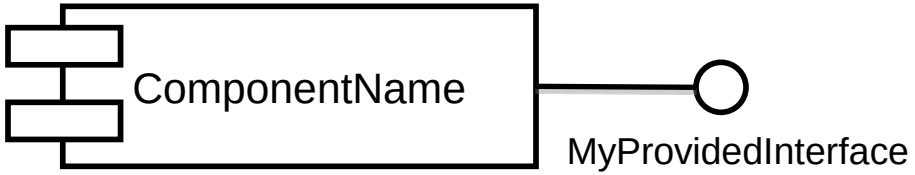
or



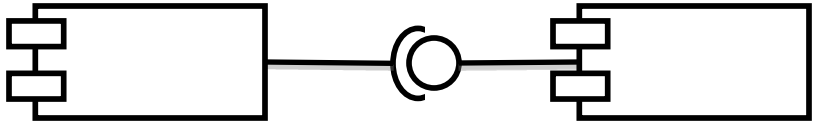
A component may require specific services, specified in a ***required*** interface *(interface is optional)*



A component may provide specific services, specified in a ***provided*** interface



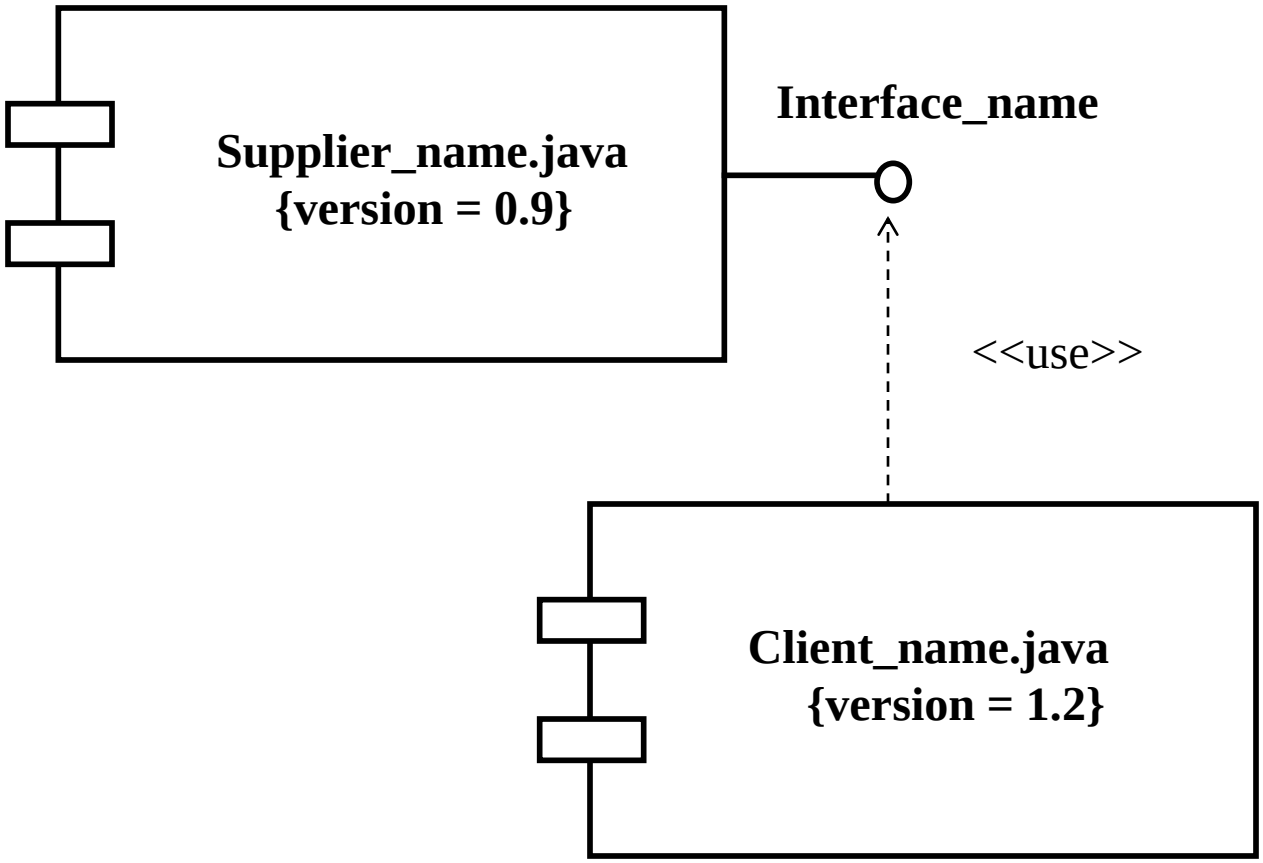
Interacting components through their interfaces



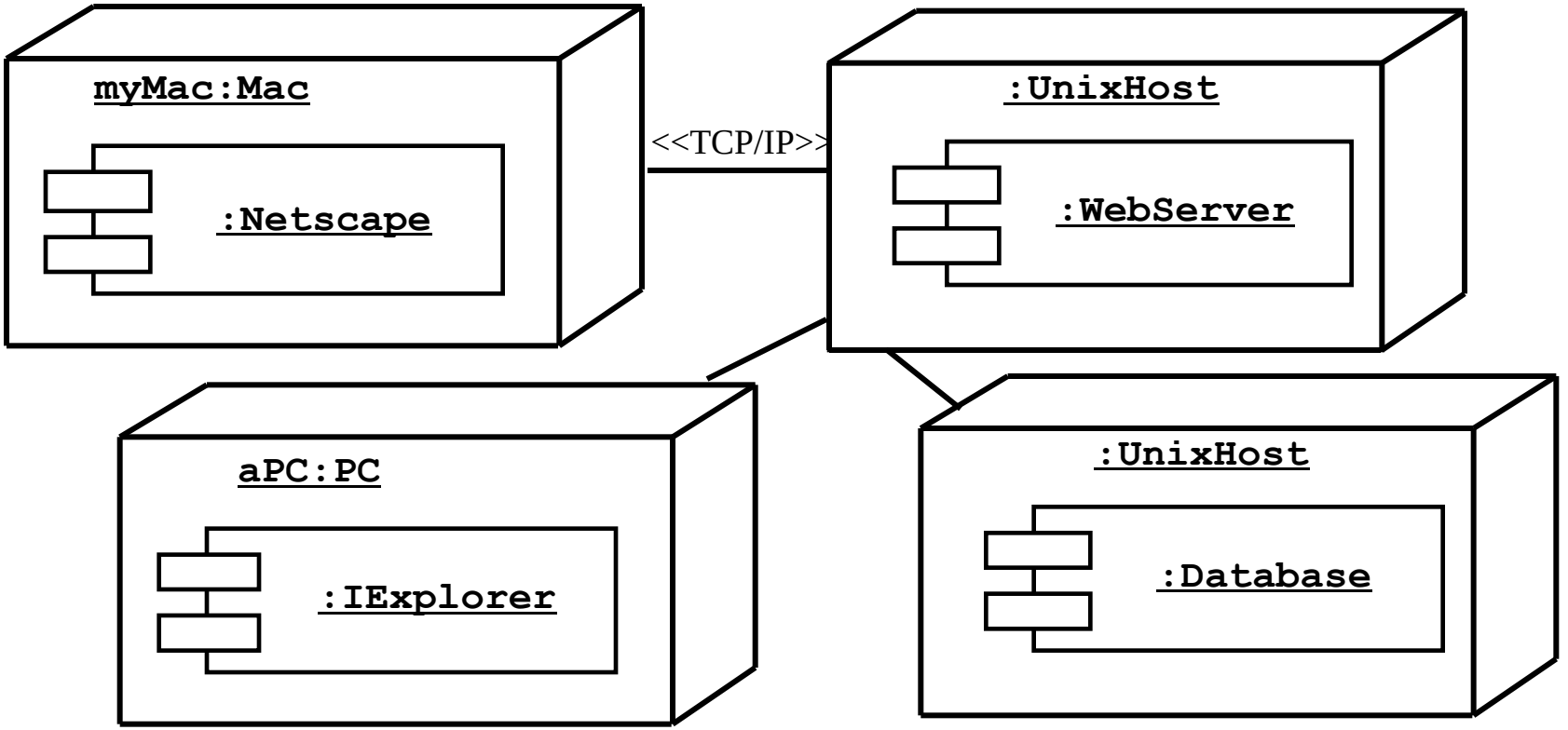
Interfaces

- A *named* set of operations
- Characterize the behavior of a component
- A component can have several (required, provided) *interfaces*
- Essentially generalize the notion of interface in Java
- Components are said to *conform* to interfaces: support all operations in interface
- Interfaces can help specify what part of a class is actually used by client classes

UML Representations



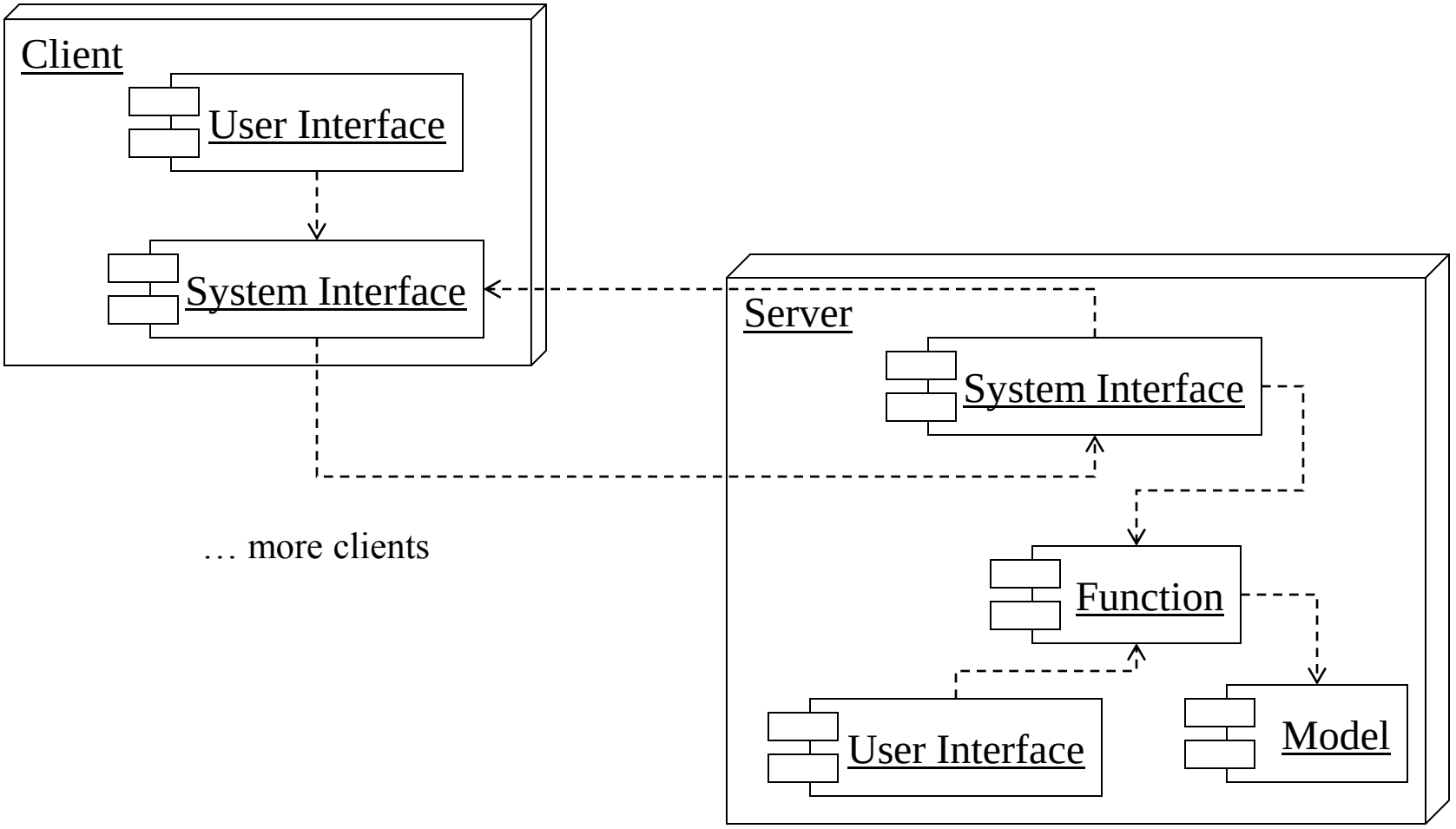
UML Deployment Diagrams



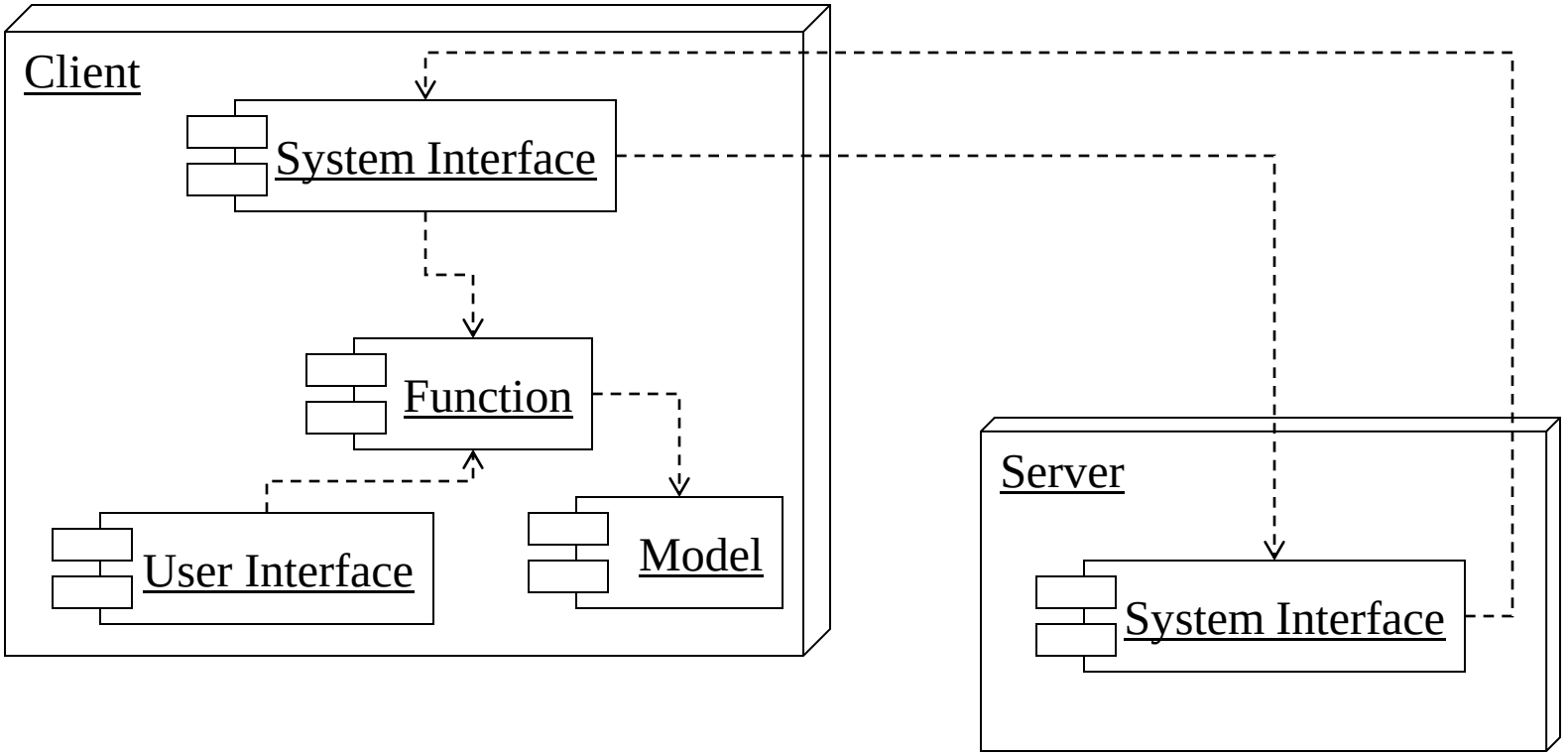
Distribution Patterns

- A typical situation is when a system is distributed on several processors over a network
- Typically, a client-Server architecture can be adopted but there are several process architecture patterns that can be employed.
- Three patterns:
  - Centralized
  - Distributed
  - Decentralized

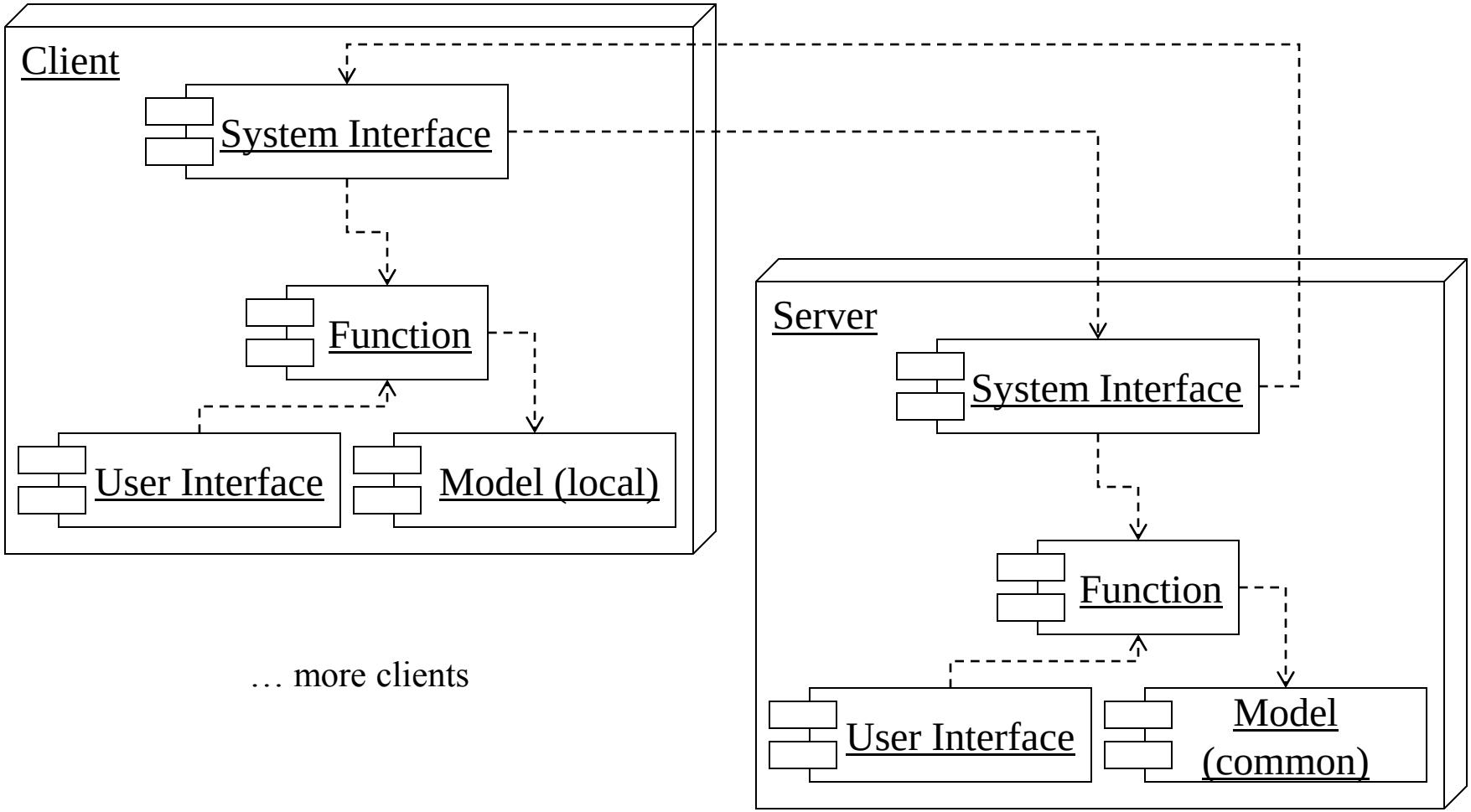
Centralized Pattern



Distributed Pattern



Decentralized Pattern



Distribution Pattern—How to Choose?

- Consider the impact of the choice of distribution patterns on non-functional aspects.
  - Constraints (non-functional requirements) drive design decisions
- Criteria (among many) to consider:
  - How many request the server will have to handle on a regular basis?
    - Few requests → no need for expensive hardware on server
      → no need for expensive network infrastructure
    - Many requests → …
  - How often do data have to be changed (and updated)?
    - Often → many updates, many requests to update (Distributed pattern)
    - Not often → …
  - Do changes to data happen more often on the local model and less often on the common model?
    - Often → …
    - Not often → …
  - What happens when the server (or the network) is down?
    - Can clients still function (even in a degraded mode, i.e., providing fewer functions)?

SYSC-3020 — Introduction to Software Engineering

- System design concepts
  - Definitions: Architecture (subsystem decomposition), coupling, cohesion
  - Layers and partitions
  - Software architecture (MVC, Observer pattern)
  - Process architecture (UML notation and Distribution patterns)
- System design process
  - Identify design goals
  - Initial subsystem decomposition
  - Map subsystems to components and processors
  - Persistent storage
  - Define access control policies
  - Select control flow mechanism
  - Identify boundary conditions

Route Planning System: MyTrip

- A driver can plan a trip from a home computer by contacting a trip planning service on the Web
- The trip is saved for later retrieval on the server and is reused on an on-board computer in the car
- The trip planning service must support more than one driver

PlanTrip Use Case

Entry Condition: The `Driver` activates her home computer and logs into the trip planning Web service

Flow of Events

1. The `Driver` enters constraints for a trip as a sequence of destinations
2. Based on a database of maps, the planning service computes the shortest way visiting the destinations in the specified order. The results is a sequence of segments binding a series of crossings and a list of directions
3. The `Driver` can revise the trip by adding or removing destinations
4. The `Driver` enters a name for the planned trip.

Entry Condition: The planned trip is saved by name in the planning service database for later retrieval

ExecuteTrip Use Case

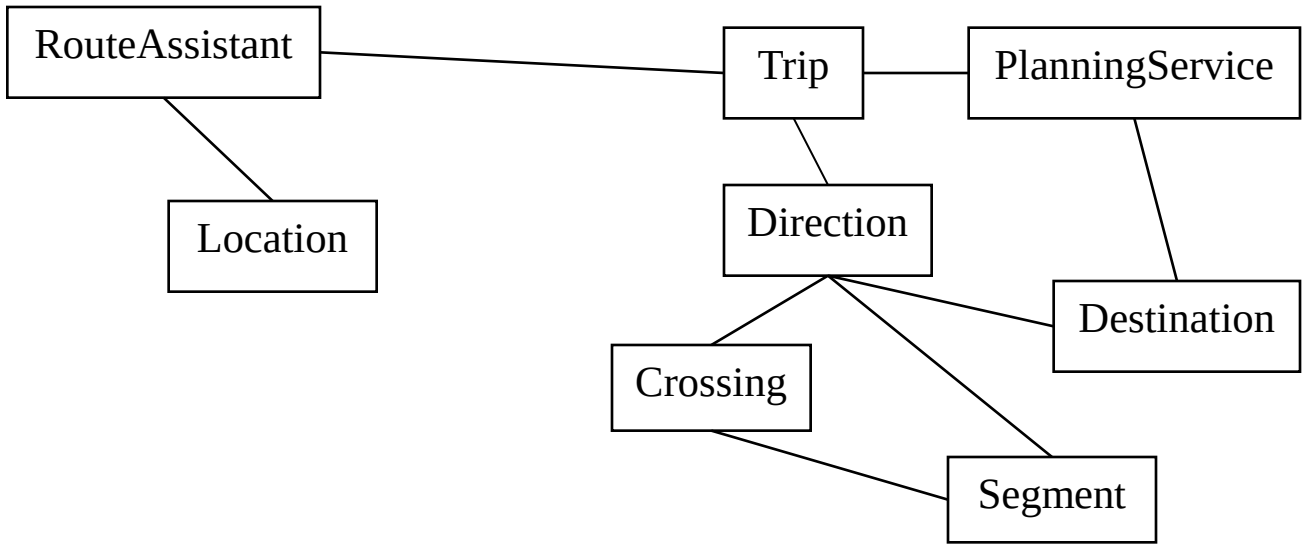
Entry Condition: The *Driver* starts her car and logs into the onboard computer route assistant

Flow of Events

1. The Driver specifies the planning service and the name of the trip to be executed
2. The onboard route assistant obtains the list of destinations, directions, segments, and crossings from the planning service
3. Given the current position, the route assistant provides the *Driver* with the next set of directions

Exit Condition: The *Driver* arrives to destination and shuts down the route assistant

Analysis Class Diagram



Non-functional Requirements

- MyTrip is in contact with a web planningService via a wireless modem.
- MyTrip should give correct directions even if the modem fails to maintain a connection with the PlanningService.
- MyTrip should minimize connection time to reduce operation costs
- Replanning is possible only if the connection to the planningService is possible
- The PlanningService can support at least 50 different drivers and 1000 trips

Identifying Design goals

- Many design goals can potentially be considered
  - Performance, dependability, end user criteria, cost, maintenance
- Can be inferred from non-functional requirements, application domain, clients, developers
- MyTrip Examples:
  - NF requirements: reliability, fault tolerance to connectivity loss with routine service
  - Application domain: security. Other drivers should not access trips from a driver.
  - Developers: modifiable to use different routing services
- Trade-offs are usually necessary

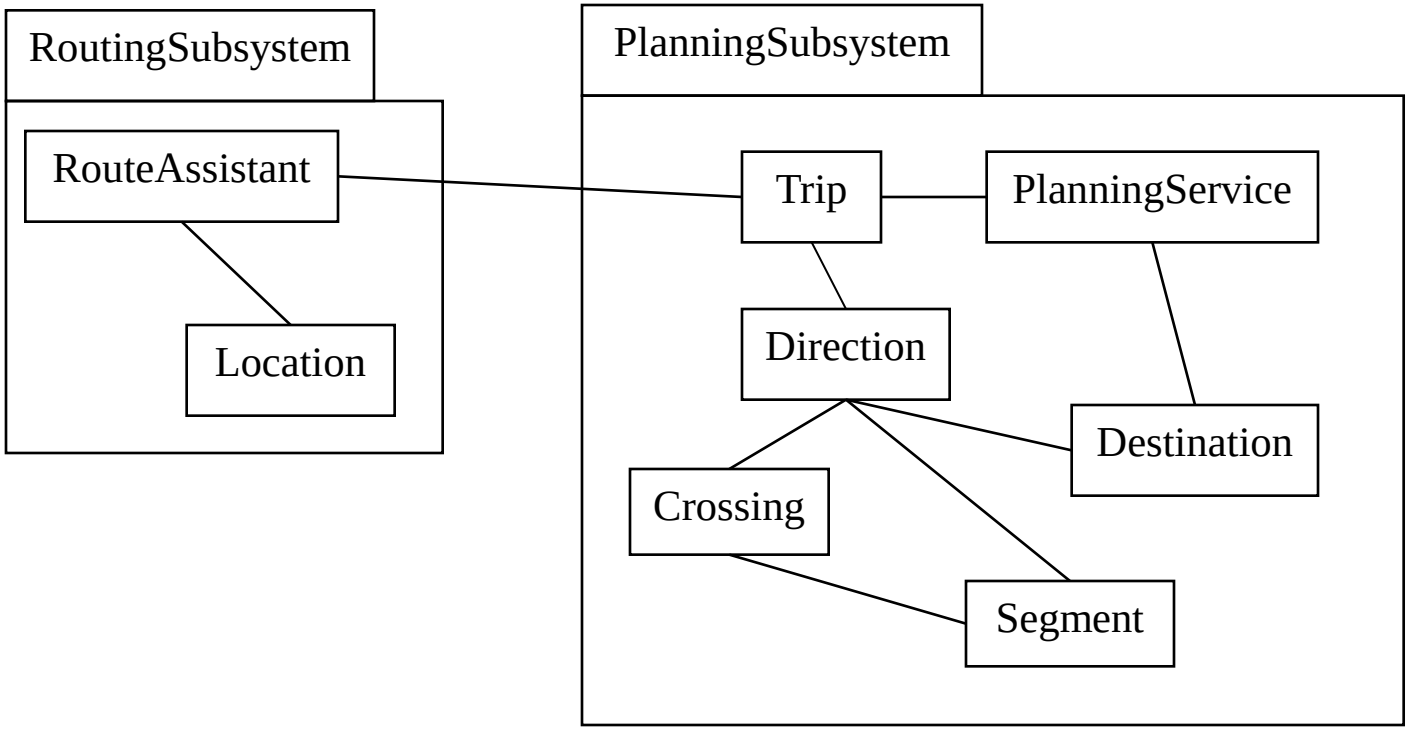
SYSC-3020 — Introduction to Software Engineering

- System design concepts
  - Definitions: Architecture (subsystem decomposition), coupling, cohesion
  - Layers and partitions
  - Software architecture (MVC, Observer pattern)
  - Process architecture (UML notation and Distribution patterns)
- System design process
  - Identify design goals
  - Initial subsystem decomposition
  - Map subsystems to components and processors
  - Persistent storage
  - Define access control policies
  - Select control flow mechanism
  - Identify boundary conditions

Identifying Subsystems

- Goal: Larger grain information-hiding solution, once subsystem interfaces defined, their design proceed independently
- Based on heuristics and iterative :
  - Subsystems are merged, split. New ones are added.
- Initial decomposition based on functional requirements, ie. objects involved in the use cases
- 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 interactions: number of associations crossing subsystem boundaries, (distinct) messages being exchanged between subsystems
  - All objects in the same subsystem should be functionally related

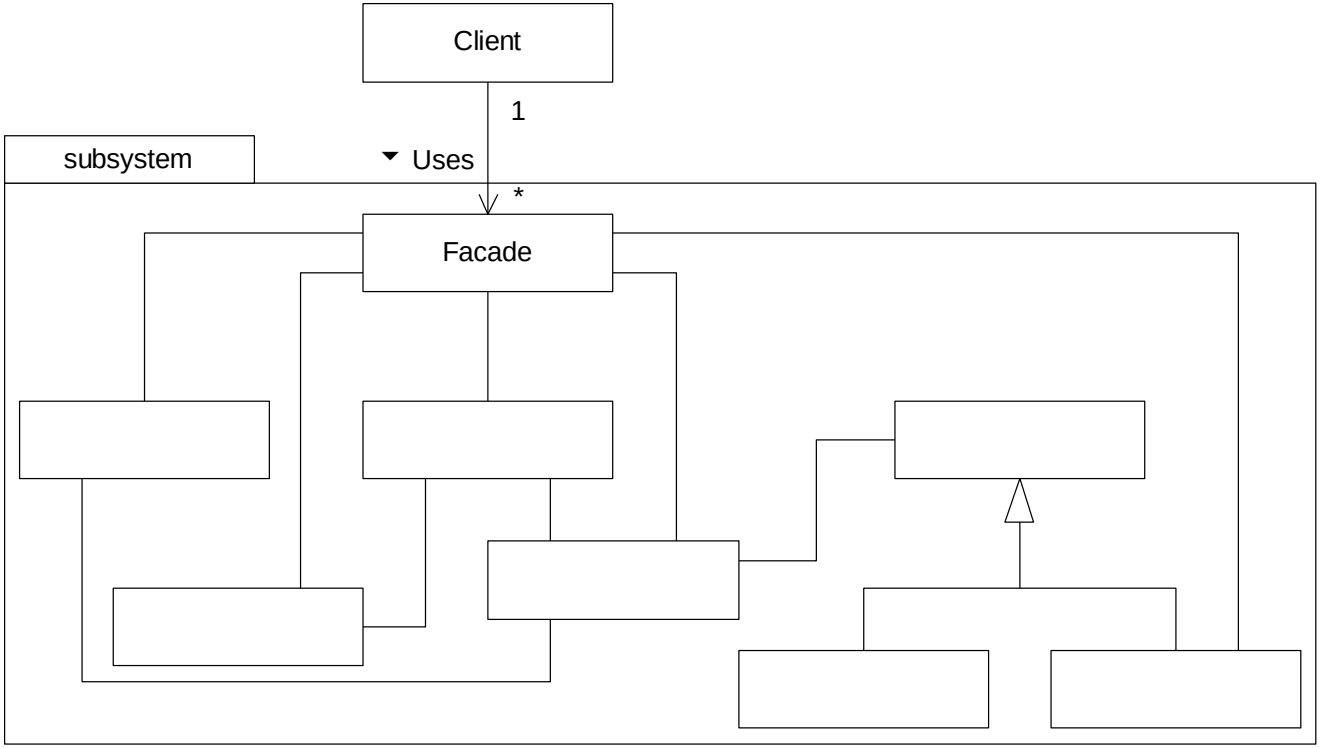
MyTrip Subsystems



Facade Design Pattern

- Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
- Facade
  - provides a unified higher-level interface to the subsystem functionality.
  - knows which subsystem classes are responsible for a request.
  - delegates (possible translation) client requests to appropriate subsystem objects.
- Subsystem classes
  - implement subsystem functionality.
  - handle work assigned by the Facade object.
  - have no knowledge of the facade.

Facade Design Pattern

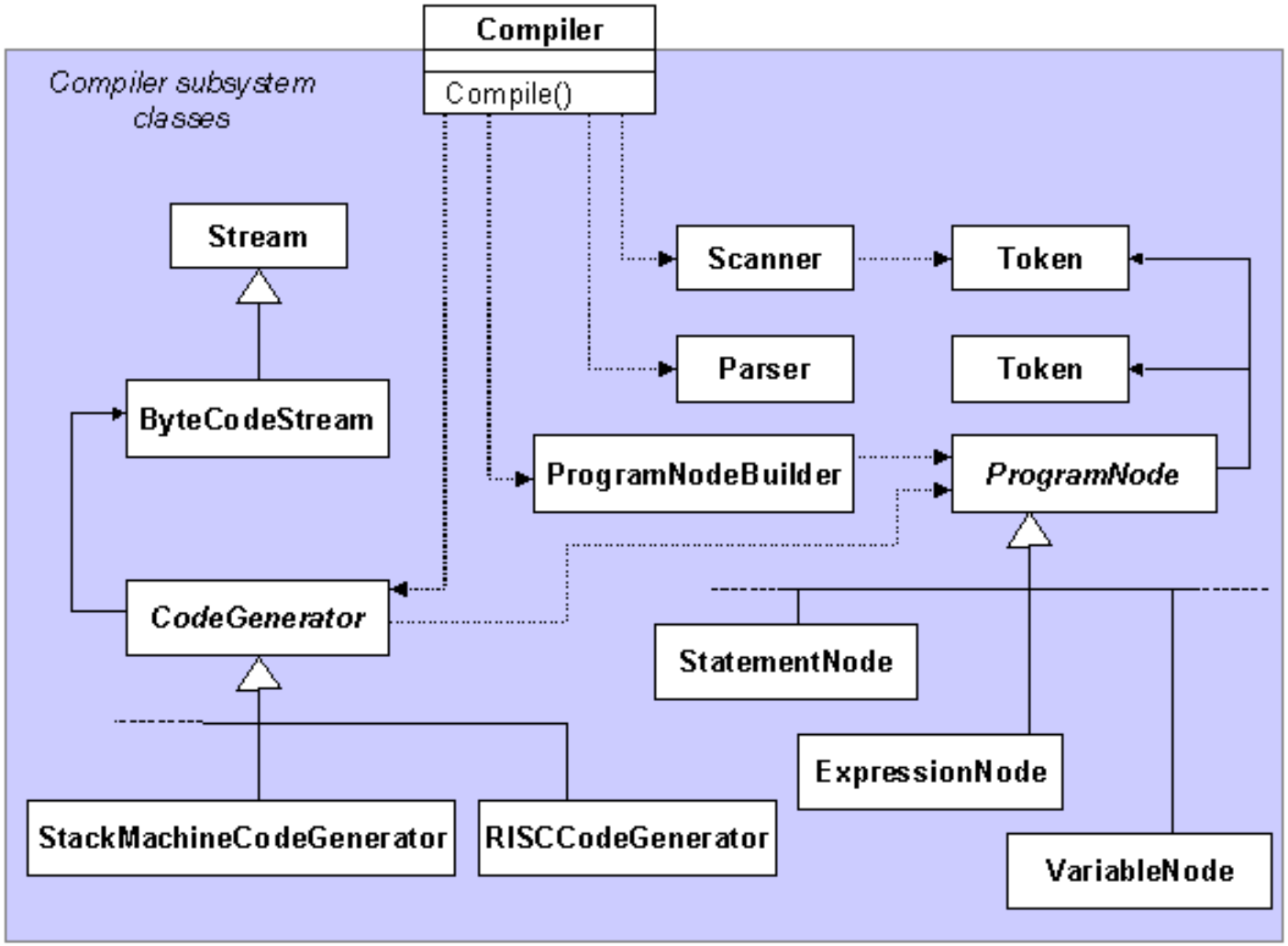


Facade Design Pattern

Consequences

1. Facade shields clients from subsystem components
   - thereby reducing the number of objects that clients deal with and making the subsystem easier to use.
2. It promotes weak coupling between the subsystem and its clients
   - thereby letting you vary the components of the subsystem without affecting its clients.
3. Facades help layer a system and the dependencies between objects
   - eliminating complex or circular dependencies.
4. A facade can also simplify porting systems to other platforms.
5. It doesn't prevent applications from using subsystem classes directly if they need to.

Facade Design Pattern (example)



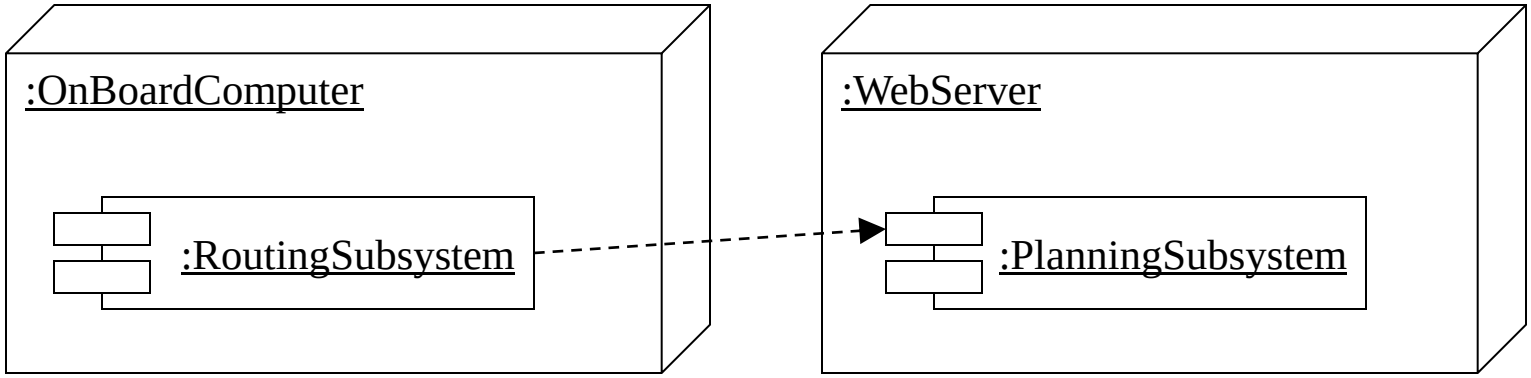
SYSC-3020 — Introduction to Software Engineering

- System design concepts
  - Definitions: Architecture (subsystem decomposition), coupling, cohesion
  - Layers and partitions
  - Software architecture (MVC, Observer pattern)
  - Process architecture (UML notation and Distribution patterns)
- System design process
  - Identify design goals
  - Initial subsystem decomposition
  - Map subsystems to components and processors
  - Persistent storage
  - Define access control policies
  - Select control flow mechanism
  - Identify boundary conditions

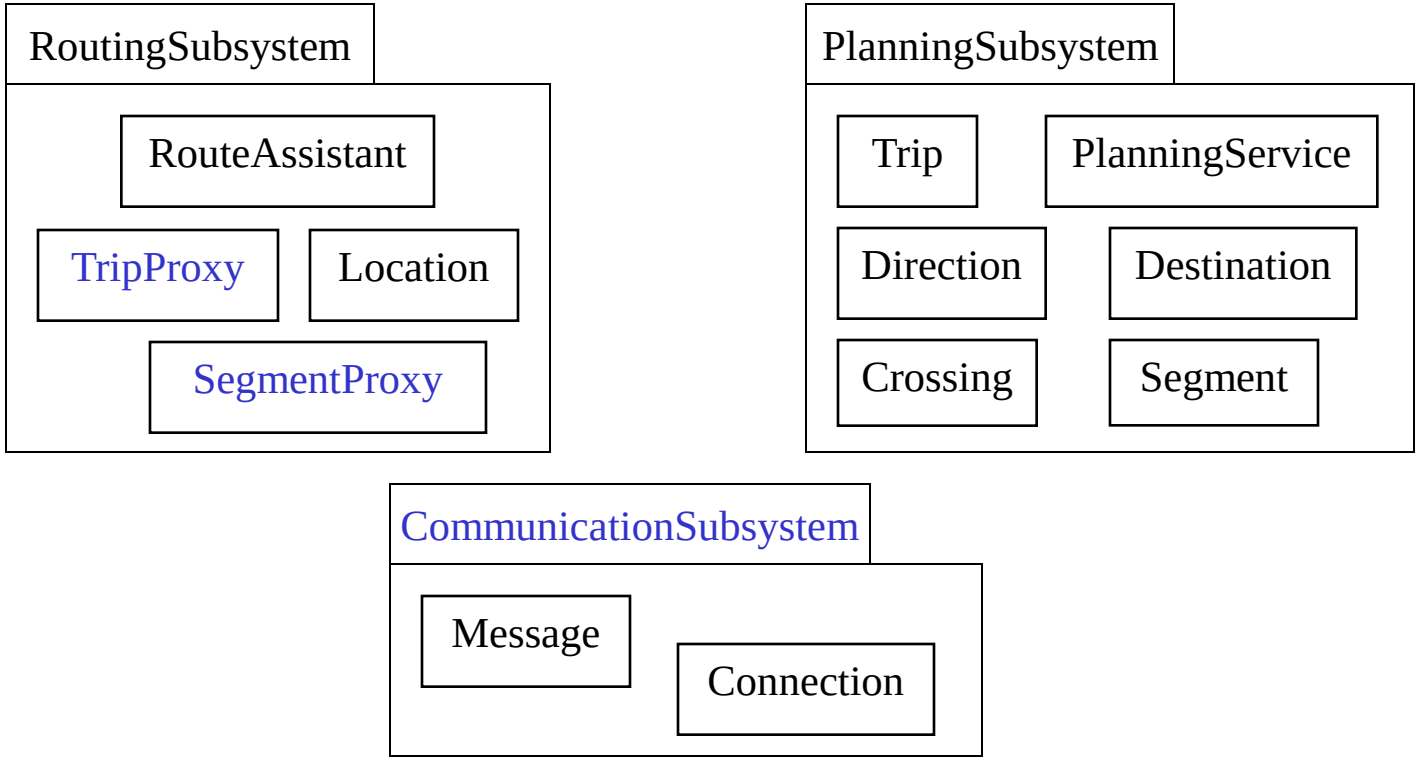
HW Configuration and Platform

- Many systems run on several computers and depend on access to an intranet or the internet :
  - High-performance needs
  - Interconnect multiple distributed users
- Allocation of subsystems to computers is done early in system design, because
  - Adds complexity and impacts performances
  - Requires communication infrastructure between nodes (potentially new subsystems)
- Subsystem mapping given by :
  - Deployment diagram (HW Configuration)
  - Virtual machine or platform: OS, components (DBMS, communication)

myTrip Deployment Diagram



New Classes and Subsystems



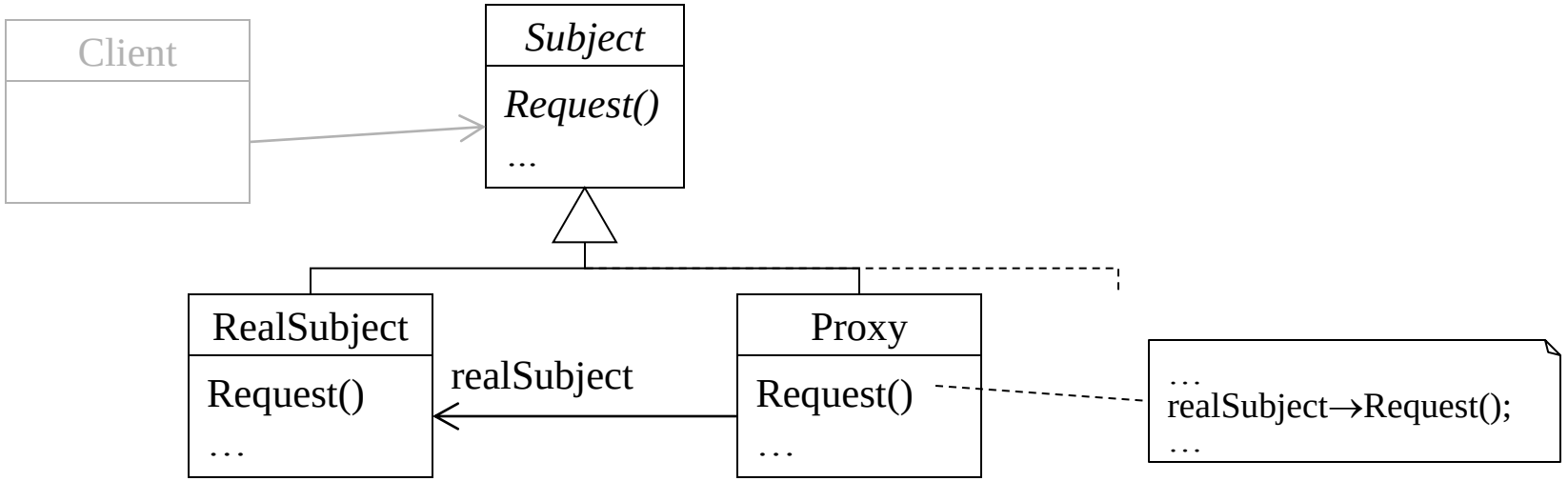
Proxy Classes

- The `RoutingSubsystem` classes need to access `Trip` and `Segment` information, e.g., create a `Trip` providing `Destination`s (replan) and retrieving corresponding `Segment`s
- That information is in the `PlanningSubsystem` but this must be transparent to `RoutingSubsystem` classes - they use *Proxy classes* for `Trip` and `Segment`
- Proxy classes use the *Proxy design pattern*

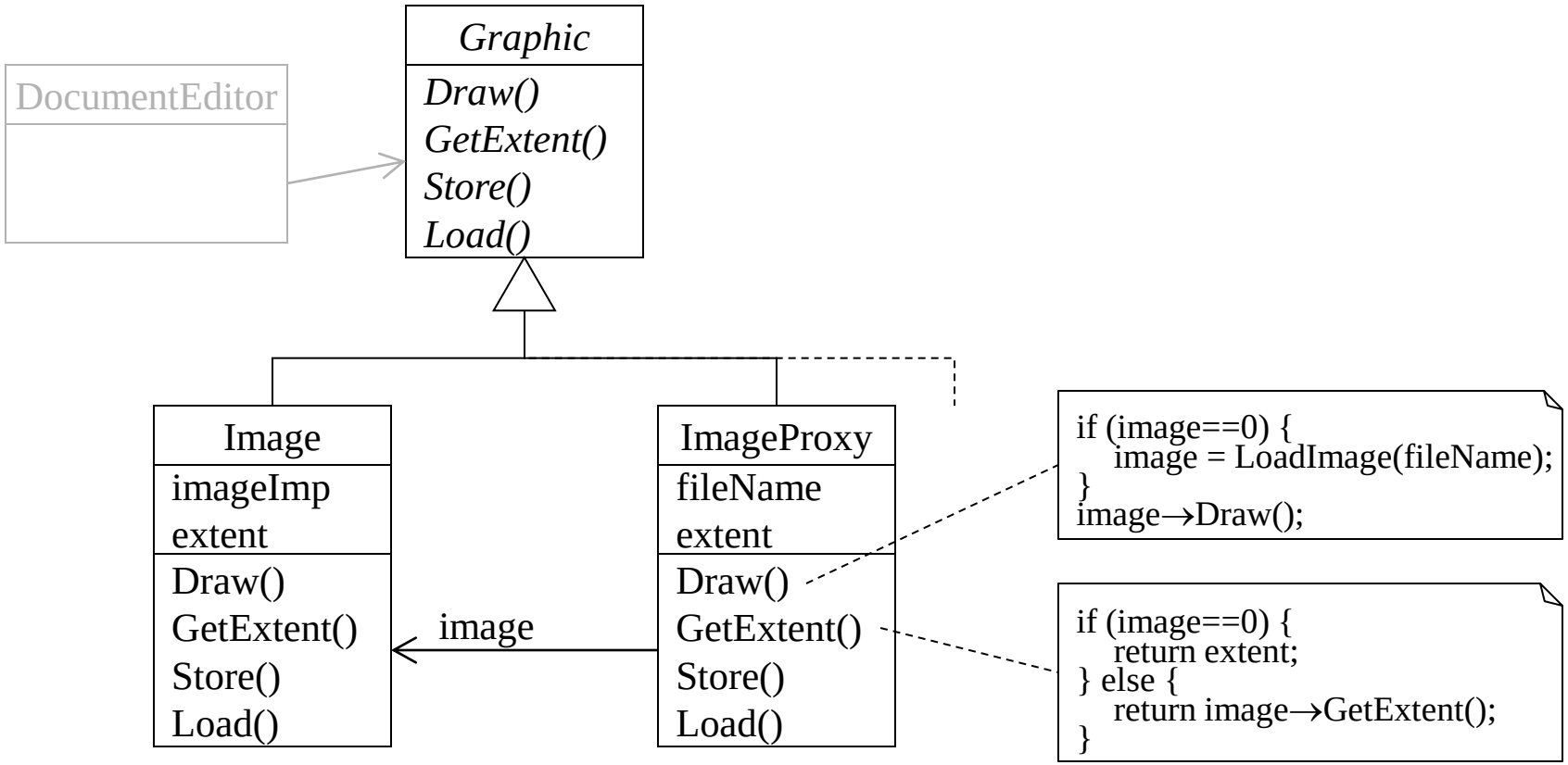
Proxy Design Pattern

- **Intent:** Provide a surrogate or placeholder for another object to control access to it.
- **Applicability:**
  - *Remote proxies* are responsible for encoding a request and its arguments and for sending the encoded request to the real subject in a different address space.
  - *Virtual proxies* may cache additional information about the real subject so that they can postpone accessing it. For example, the ImageProxy from the Document editor (next) caches the real image's extent.
  - *Protection proxies* check that the caller has the access permissions required to perform a request.
  - *Smart reference*: additional services such as reference counting, loading from persistent storage

Proxy Structure



Document Editor



Consequences

- A remote proxy can hide the fact that an object resides in a different address space or a remote machine (myTrip example). Loading it over the network might be slow at peak load periods
- A virtual proxy (our editor example) can perform optimizations such as creating an object on demand, e.g., large images on a web browser or word processor
- Both protection proxies and smart references allow additional housekeeping tasks when an object is accessed

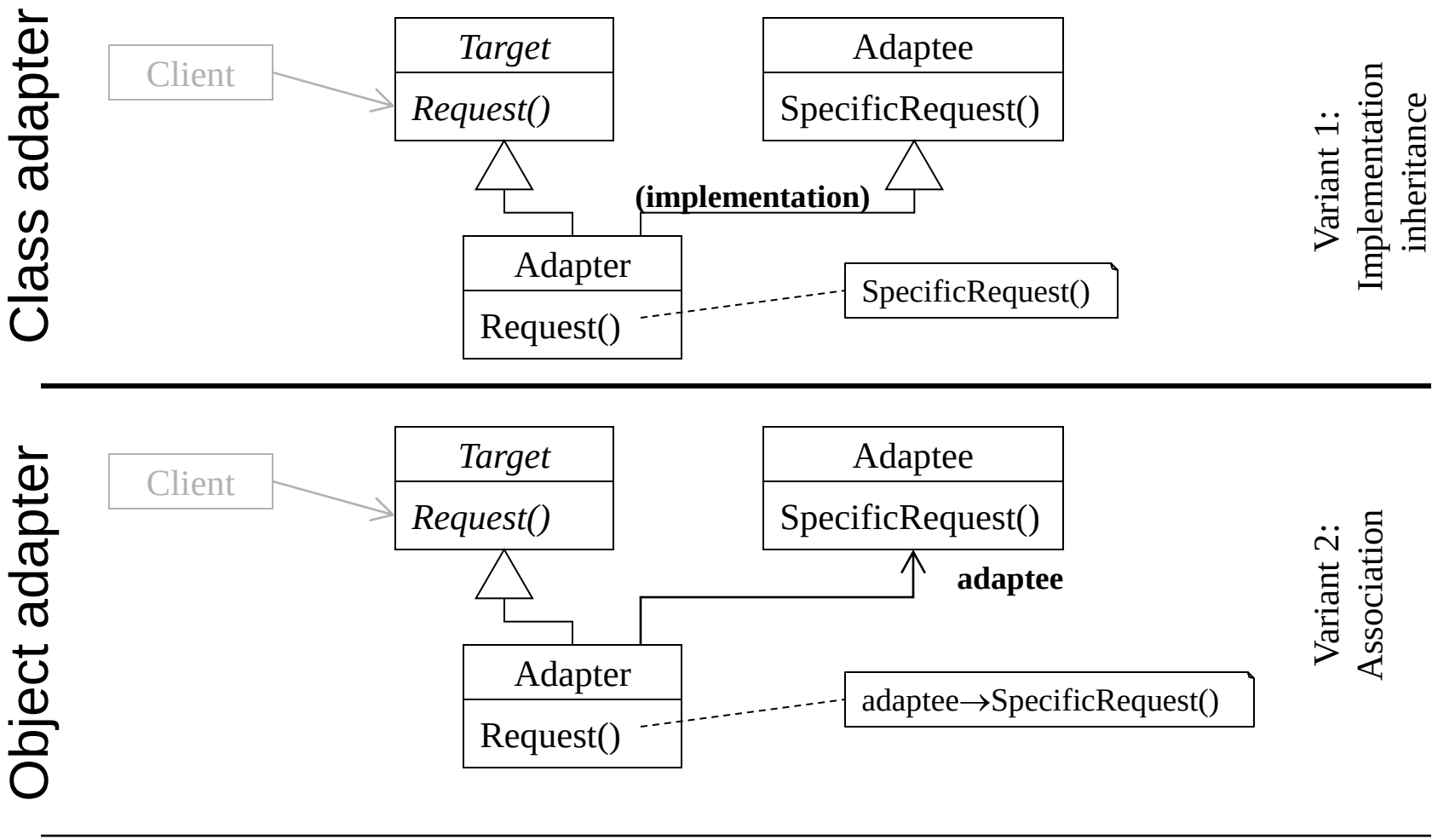
Encapsulating Components

- Strong incentives to reuse code or to rely on COTS, e.g., GUI builders providing standard interface objects
- Existing code cannot be modified and has not been specifically designed to integrate into the new system to be developed
- Solution: *Adapter design pattern*

Adapter Pattern

- **Intent:**
  - Convert the interface of a class into another interface clients expect.
  - Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
  - Create a new interface for an object that does the right stuff but has the wrong interface.
- **Typical application contexts:** reuse of existing code, off-the-shelf software

Adapter Pattern—Structures

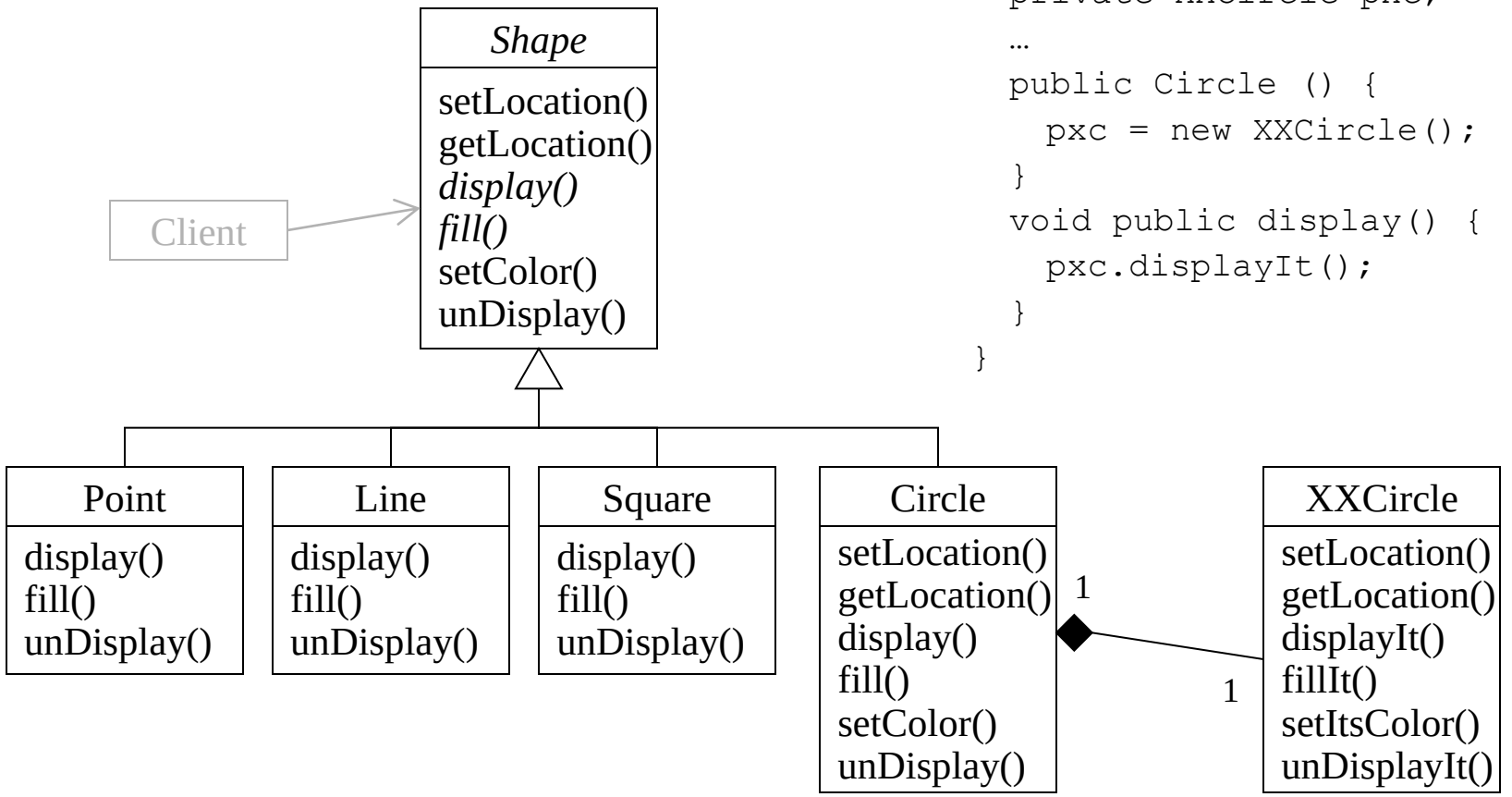


Simple Example

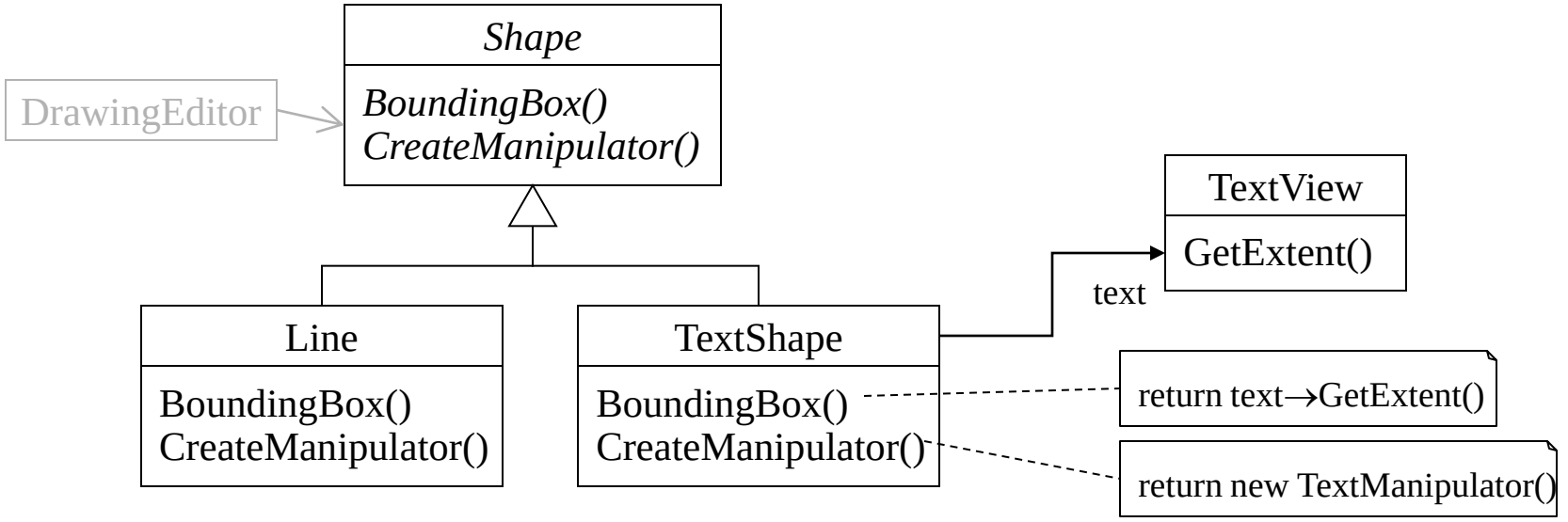
- A client object tells a point, line, square, etc. to do something such as display or undisplay itself.
- We create an abstract `Shape` class and then derive other classes from it
- Common interface for behavior of `Shape`s
- We have an existing `XXCircle` class that can be reused but with a different interface!

Class Diagram

```
class Circle extends Shape {
  …
  private XXCircle pxc;
  …
  public Circle () {
    pxc = new XXCircle();
  }
  void public display() {
    pxc.displayIt();
  }
}
```



Drawing Editor



Consequences: Class Adapter

+ Permits Adapter to override some of the methods of Adaptee
+ Use only one object since Adapter and Adaptee have the same instance
- Does not allow adaptation of a class and its subclasses with one adapter
- Use of multiple inheritance

Consequences: Object Adapter

+ Can adapt all subclasses of Adaptee with just one Adapter
- Harder for Adapter to override Adaptee's behavior
- Sometimes results in information spread over two objects
  - If Adapter adds to the implementation provided by Adaptee
  - Class Adapter avoids this problem

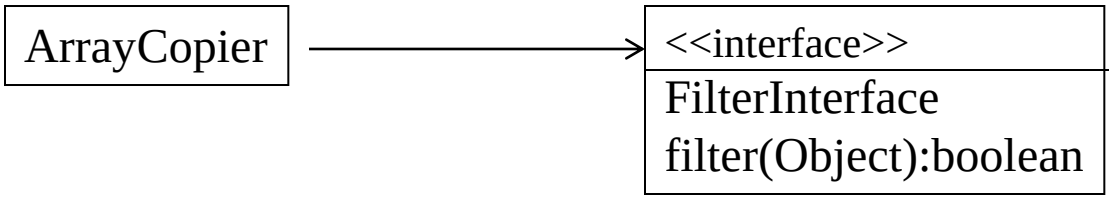
Question on Patterns : Can you analyze/apply ?

Compare Façade and Adapter.

Compare Adapter and Proxy.

Compare Proxy and Façade.

"Suppose that you are writing a method that copies an array of objects, filtering out objects that don't meet certain criteria. To promote reuse, " you have made the method independent of the actual filtering criteria used by using instances of classes that implement an interface to provide the actual filtering.



Suppose one use is for copying array of file objects, and the filter is the read/write property of the file. You notice that the File class has a method called isWritable():boolean. Of the three patterns discussed, which would you apply and how. Show a class diagram with all relevant parts.

SYSC-3020 — Introduction to Software Engineering

- System design concepts
  - Definitions: Architecture (subsystem decomposition), coupling, cohesion
  - Layers and partitions
  - Software architecture (MVC, Observer pattern)
  - Process architecture (UML notation and Distribution patterns)
- System design process
  - Identify design goals
  - Initial subsystem decomposition
  - Map subsystems to components and processors
  - Persistent storage
  - Define access control policies
  - Select control flow mechanism
  - Identify boundary conditions

Persistent Data Stores

- Persistent data outlive a single execution of a system
- “Storage Management Strategy”: How objects will be stored.
  - Criteria: performance, complex queries, space, administration capabilities
- Choose between:
  - **Flat Files**: Voluminous/unstructured data, low complexity data
  - **Database management systems** (DBMS) : Contain mechanisms for describing data, managing persistent storage and for providing a backup mechanism
    - Provides concurrent access to the stored data
    - Contains information about the data (“meta-data”), also called data schema.
    - Support multiple applications and platforms, complex queries

Database Management Systems

- **Relational DBMS** (e.g., Oracle) : Based on Tables
  - handle large datasets
  - Can perform complex queries on large data sets (e.g., SQL query language)
  - Mature technology.
- **OO DBMS** (eg. POET) : Support all fundamental object modeling concepts
  - handle medium-sized data set

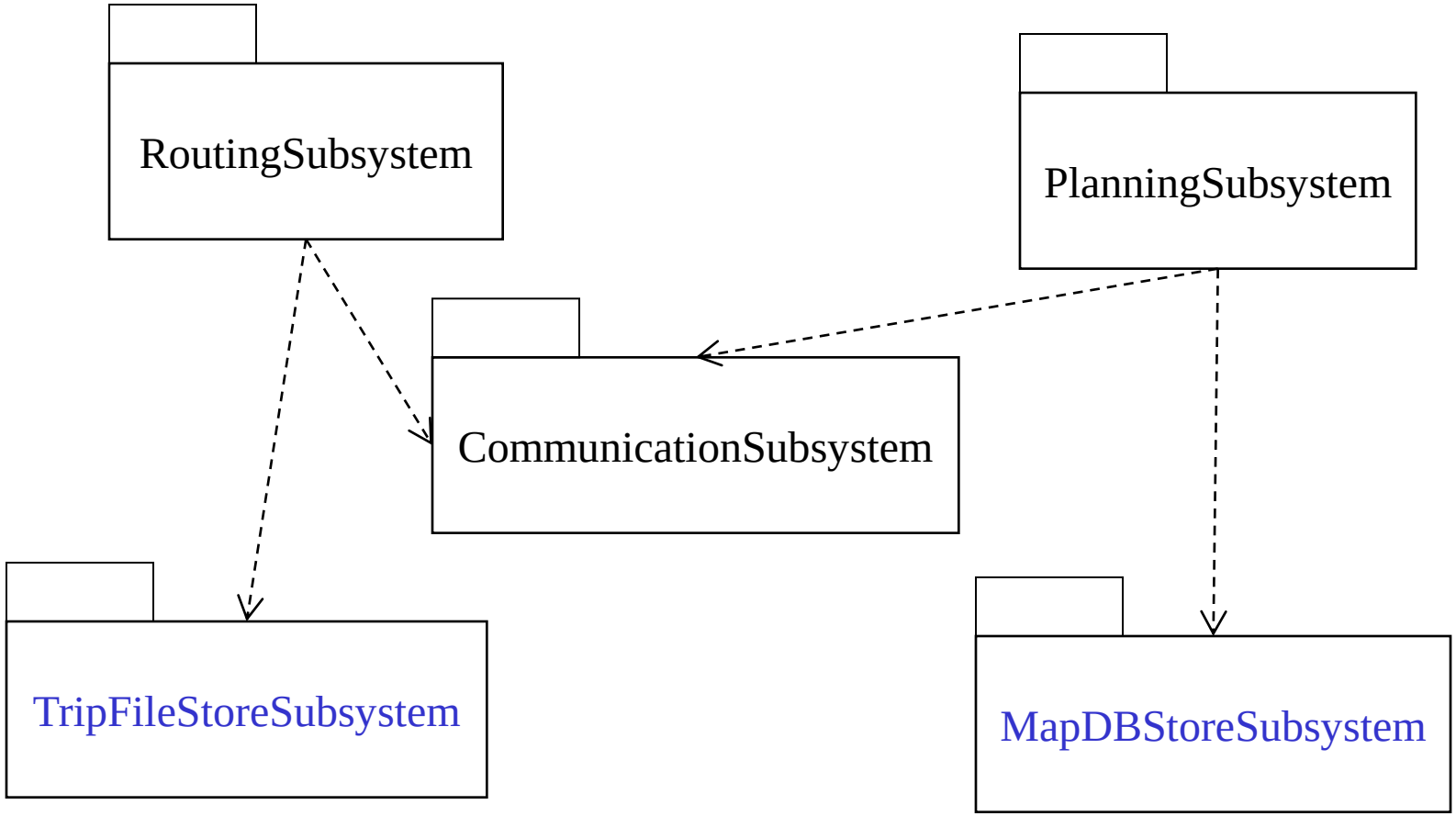
Data Management

- Data management often an important component of many systems
  - Systems usually include one or several databases
  - How to handle data storage has significant impact on system structure, performance, etc.
  - Data storage and retrieval may represent a bottleneck
- Which data needs to be persistent?
- How should it be stored, accessed?
- Additional subsystem for data management ?

Guidelines for Persistent Objects

- Need to identify which objects need to be persistent
- Entity objects are obvious candidates but not all need to be persistent
  - e.g., in MyTrip, `Trip`s and their related classes need to be stored. But `Location` and `Direction` are constantly recomputed as the car moves.
- Other examples of persistent information:
  - information related to system users (e.g., Drivers)
  - attributes of some boundary objects (eg. user interface preferences).
- Question: which classes need to survive system shutdown or crash?

MyTrip Data Storage



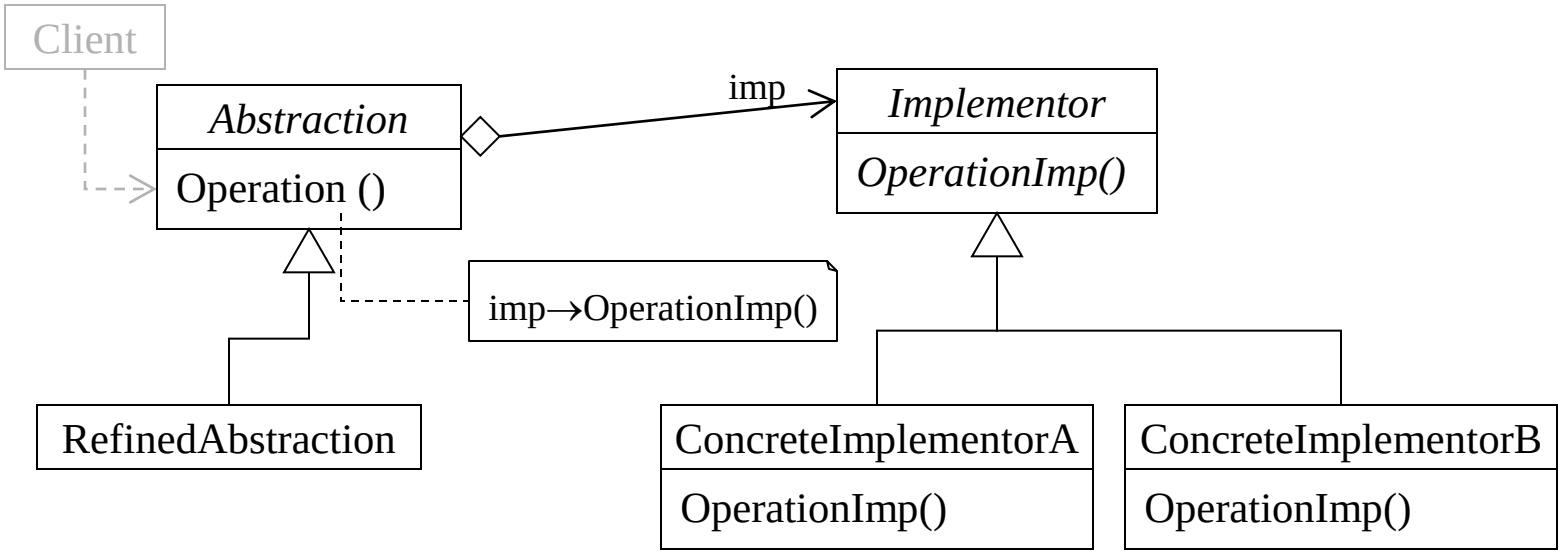
Encapsulating Data Stores

- Encapsulate into a subsystem that is vendor independent (anticipation of change)
- *Bridge design pattern*: interface and implementation are decoupled
  - Different implementations for a given class
  - Can be substituted at run-time
  - Negative impact on performance

Bridge Pattern

- **Intent:** Decouple a set of implementations from the set of objects using them
  - GOF: “Decouple an abstraction from its implementation so that the two can vary independently”
- **Typical application contexts:**
  - Variations in abstractions of a concept (eg. shape) and variations in how these concepts are implemented
  - Different implementations exist to cater for a number of special cases (e.g., special algorithms)
  - Both the abstractions and implementations may change independently overtime (both calendar and run-time)

Bridge Structure

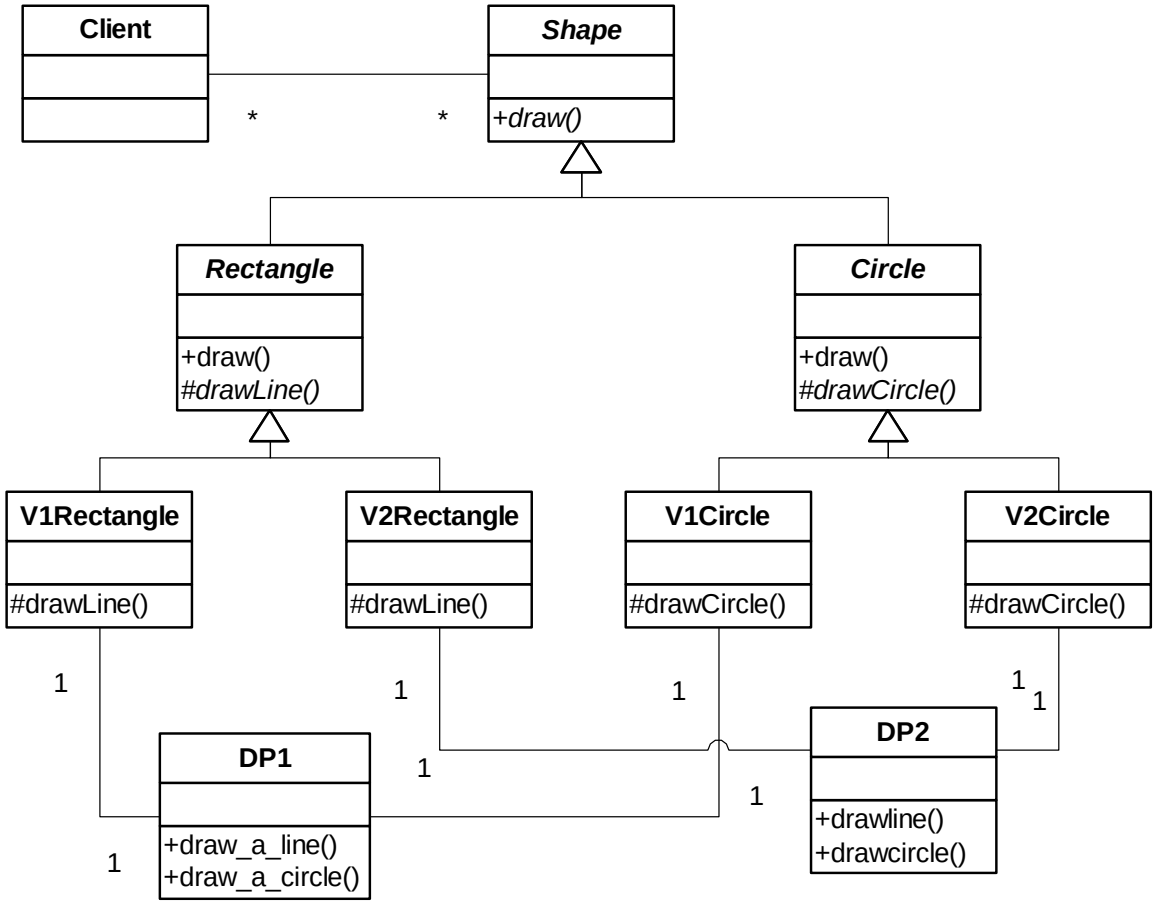


Example

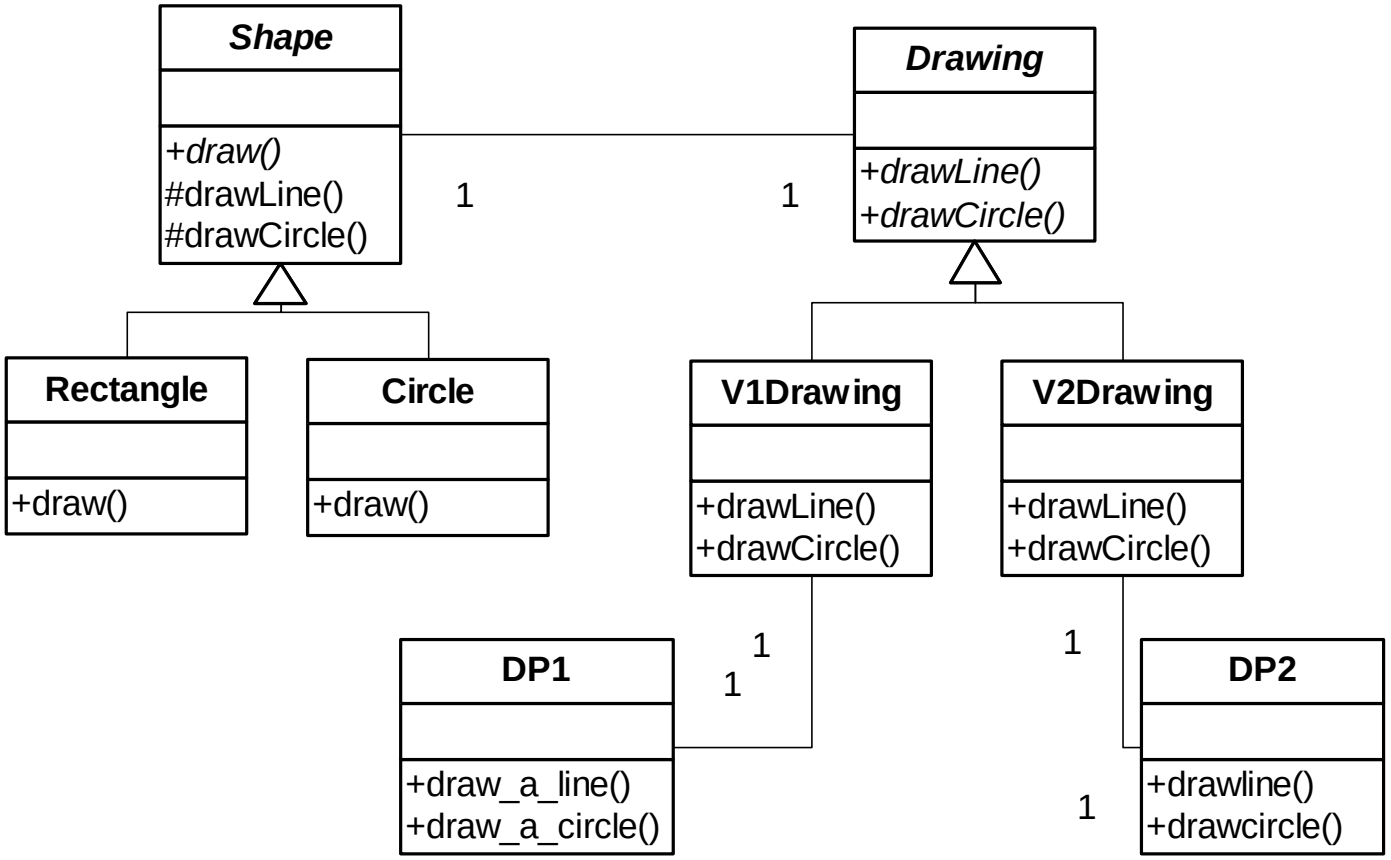
- We write a program that will draw rectangles and circles with either of two drawing programs
- When the program instantiates a `Rectangle` or `Circle`, it will know whether it should use drawing program 1 (`DP1`) or 2 (`DP2`)
- `DP1` and `DP2` have different interfaces

- Applicability to our DBMS problem: Despite standards, DBMS APIs may differ

Solution with Inheritance



Solution with Bridge Pattern



Comparison

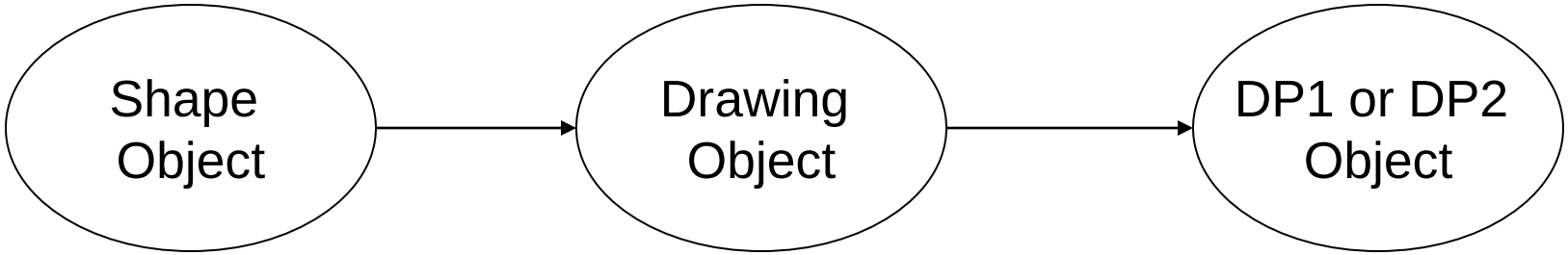
Solution with inheritance:

- Adding a new implementation → 1 + n new classes
  (a new DP3) (n being the number of shapes)
- Adding a new shape → 1 + m new classes
  (m being the number of implementations)

Solution with pattern:

- Adding a new implementation → 2 new classes
  (a new DP3)
- Adding a new shape → 1 new class
  (plus perhaps some methods in implementors)

Three Objects at a Time

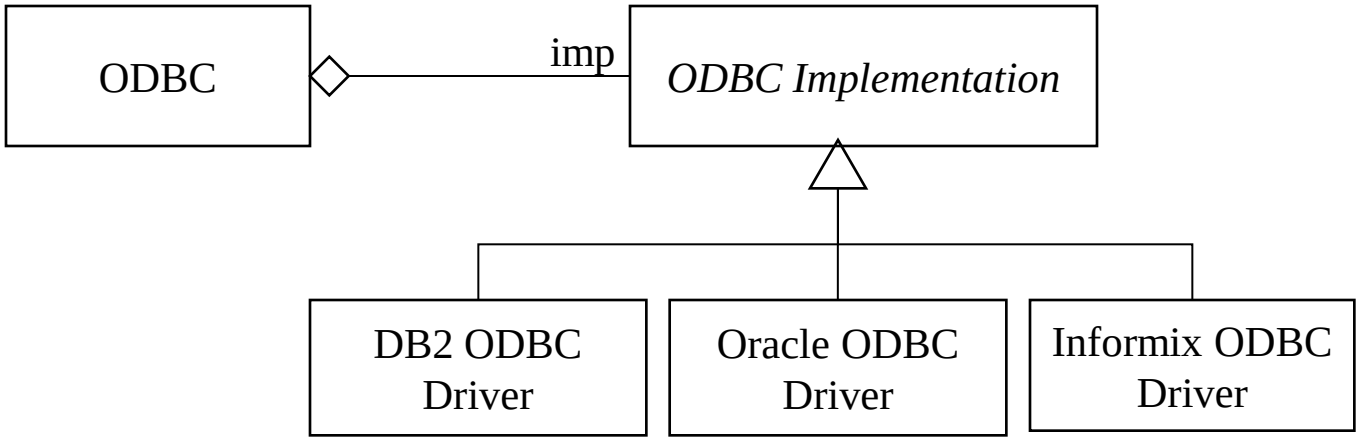


This is actually a Rectangle or a Circle, but the Client can't tell

This is actually a V1Drawing or a V2Drawing, but the Shape can't tell

This must be the correct type of the object, but the Drawing object that uses it will know which it is

Abstracting Database Vendors



SYSC-3020 — Introduction to Software Engineering

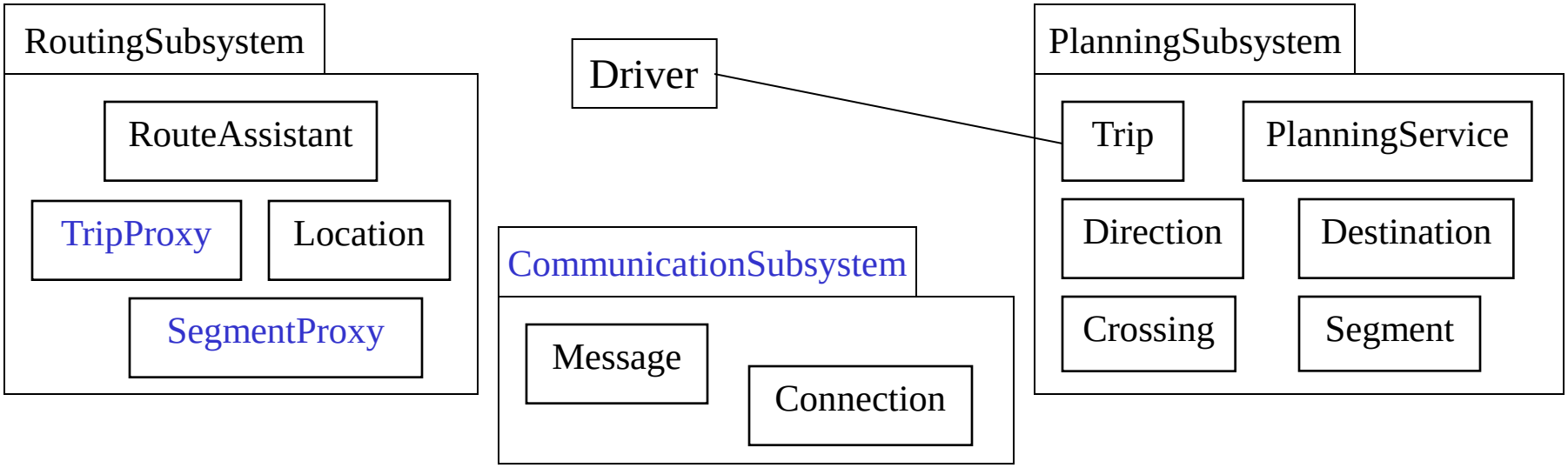
- System design concepts
  - Definitions: Architecture (subsystem decomposition), coupling, cohesion
  - Layers and partitions
  - Software architecture (MVC, Observer pattern)
  - Process architecture (UML notation and Distribution patterns)
- System design process
  - Identify design goals
  - Initial subsystem decomposition
  - Map subsystems to components and processors
  - Persistent storage
  - Define access control policies
  - Select control flow mechanism
  - Identify boundary conditions

Defining Access Control

- Security requirements:
  - Which objects are shared among actors
  - How can actors control access
  - How actors are authenticated to the system
  - How selected data in the system should be encrypted

MyTrip (I)

- Storing maps and trips in the same database for many drivers introduces security issues
- Recall security design goal: Other drivers should not access trips for a driver
  - Suggests additional Driver class, associate it with the `Trip` class
  - Leads to additional responsibilities for subsystems



MyTrip (II)

- A `Driver` represents an authenticated user. It is used by the `CommunicationSubsystem` to remember keys associated with a user and by the `PlanningSubsystem` to associate `Trip`s with users.

- The `CommunicationSubsystem` uses the Driver associated with the `Trip` being transported for selecting a key and encrypting the communication traffic

- Prior to processing any requests, the `PlanningSubsystem` authenticates the `Driver` from the `RoutingSubsystem`. The authenticated `Driver` is used to determine which `Trip`s can be sent to the corresponding `RoutingSubsystem`.

General Mechanisms

- For multiuser system, access control is usually more complex than in `MyTrip`
- For each actor, we need to define which operations they can access on which shared object
  - Static versus dynamic access control
  - Authentication: Verifying association of user/subsystem to system
  - Encryption: Preventing unauthorized access
- Access matrix is used to model access on classes: Actors $\times$ Classes
  - Each cell list operations that can be executed for the {actor,class} pair
- Alternative representations for access matrix :
  - Global access table: (actor, class, operation) tuple
  - Access control list: list of (actor, operation) for each class
  - Capability: list of (class operation) for each actor

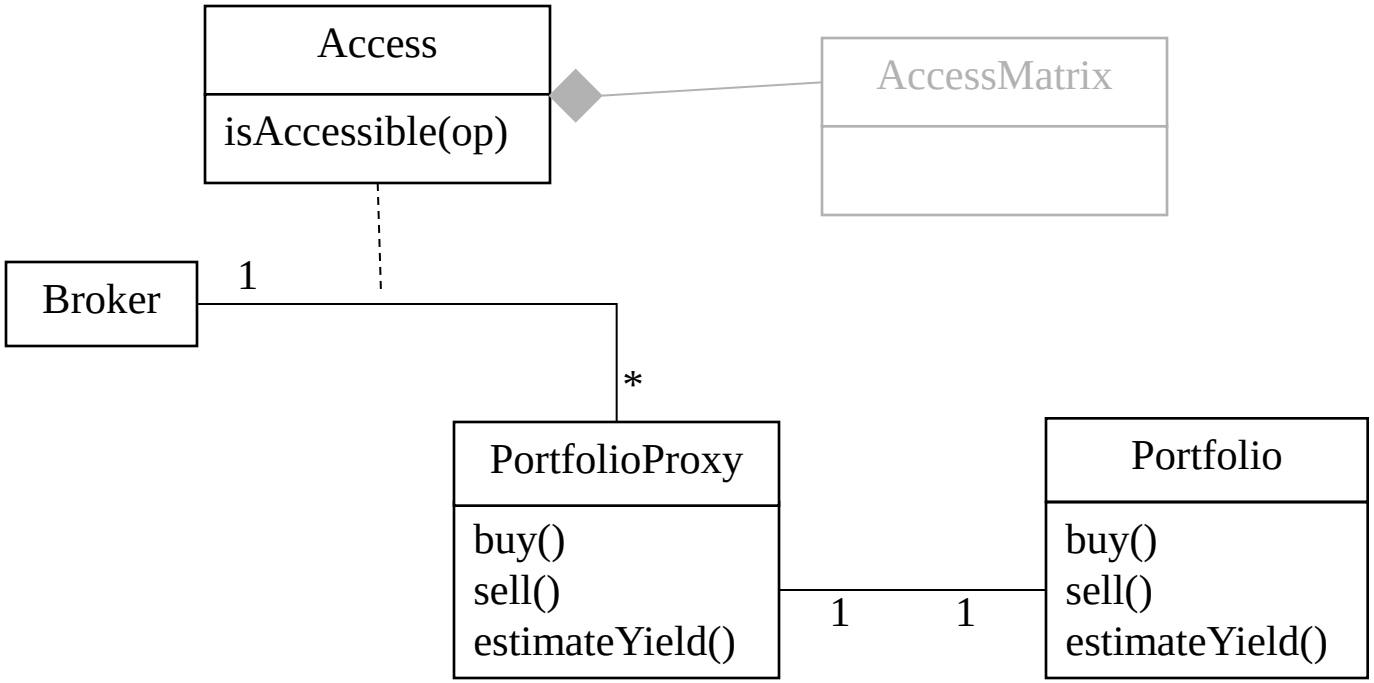
Example

- Bank information system
- A `Teller` may debit or credit an account (operations on class `Account`) up to a predefined amount
- If this amount is exceeded, a `Manager` needs to approve the transaction
- `Manager`s and `Teller`s can only access accounts in their own branch
- `Analyst`s can access information across branches but cannot post transactions on individual accounts

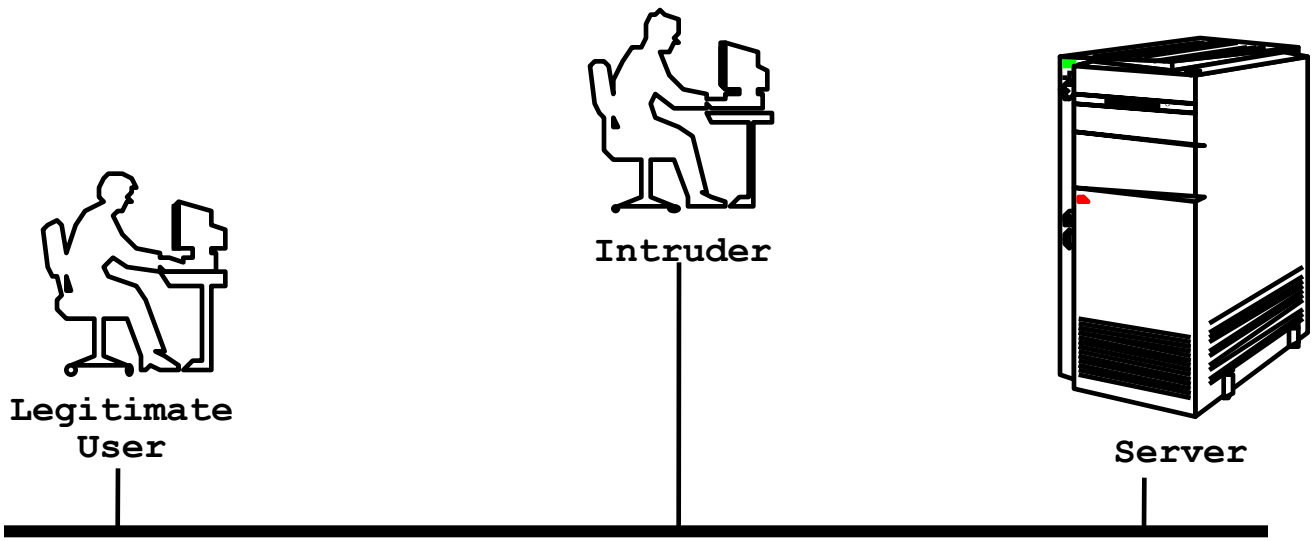
Example Access Matrix

		Objects		
		Corporation	LocalBranch	Account
Actors	Teller		lookupLocalAccount()	postSmallDebit() postSmallCredit() …
	Manager		lookupLocalAccount()	… postLargeDebit() postLargeCredit() examineHistory()
	Analyst	examineGlobalDebits() examineGlobalCredits()	examineLocalDebit() examineLocalCredits()	

Dynamic Access with Proxy Pattern Adaptation



Passive Attacks



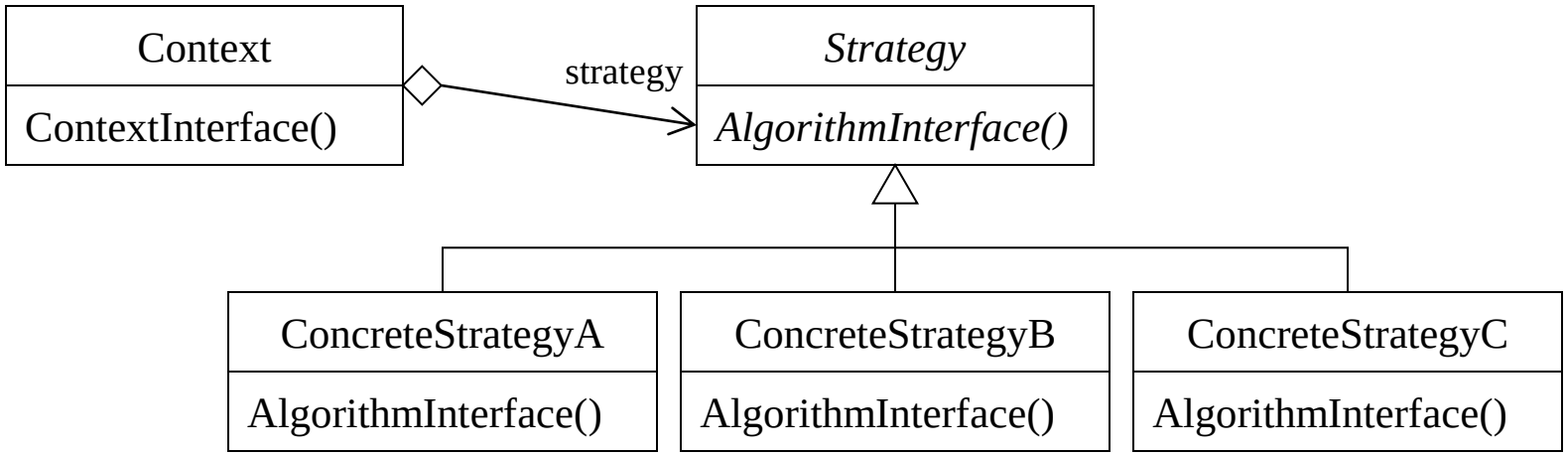
Plaintext message `CC# 1234 5678 9012 3456 EXP 8/99`

Encrypted message `XZ<ASL@#34HF*(*A2135SDA*}BKDAWR#%_AS2255`

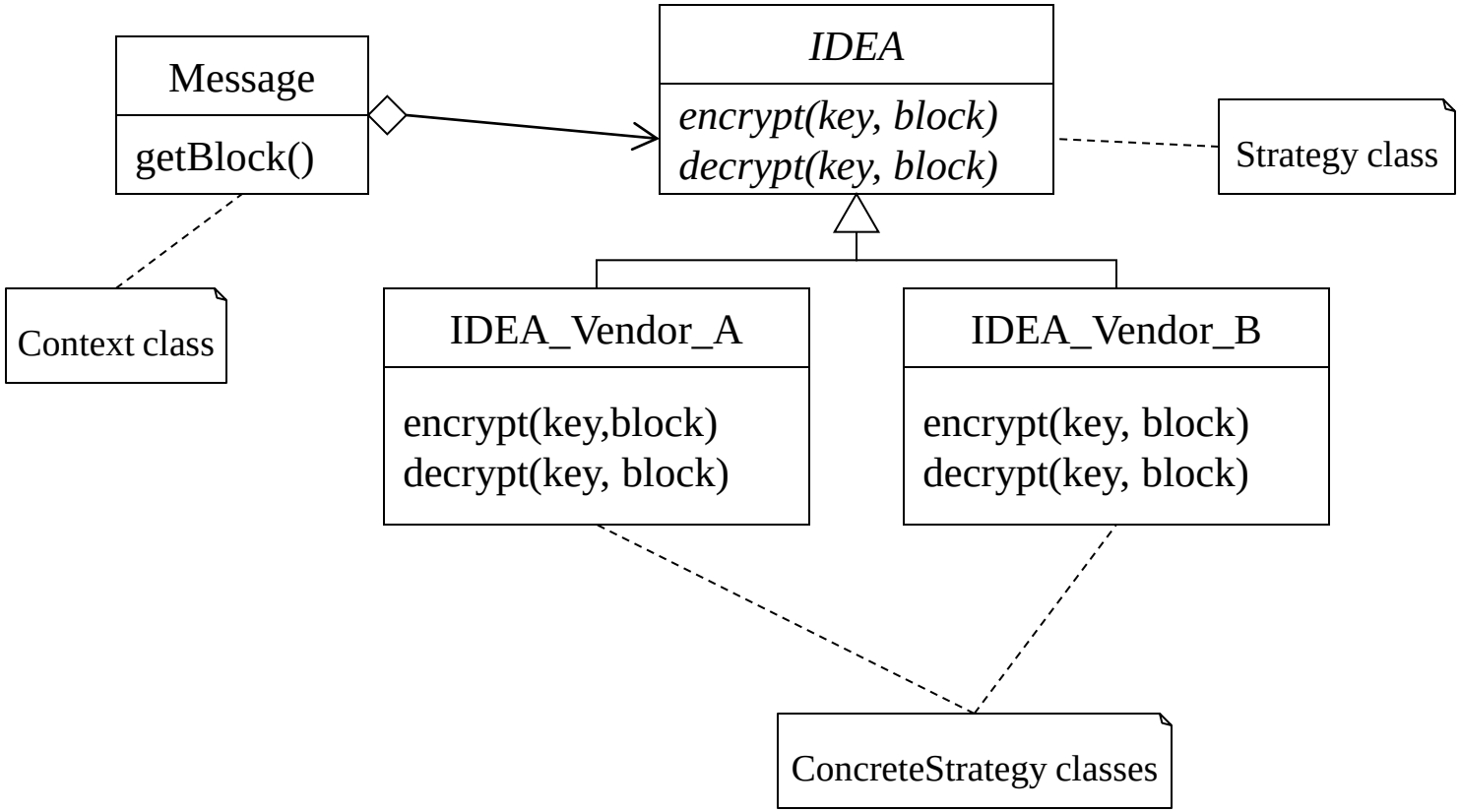
Strategy Design Pattern

- **Intent:**
  - Define a family of algorithms, encapsulate each one, and make them interchangeable.
  - Strategy lets the algorithm vary independently from clients that use it, thus selecting the appropriate one based on context
  - Separate the selection of algorithm from its implementation
- **Applicability:**
  - Different alternative algorithms are possible, they must be interchanged at run-time based on the client making the request or the data being acted upon, new/additional algorithms will be used in the future

Strategy Pattern Structure



Encapsulating Access Control



SYSC-3020 — Introduction to Software Engineering

- System design concepts
  - Definitions: Architecture (subsystem decomposition), coupling, cohesion
  - Layers and partitions
  - Software architecture (MVC, Observer pattern)
  - Process architecture (UML notation and Distribution patterns)
- System design process
  - Identify design goals
  - Initial subsystem decomposition
  - Map subsystems to components and processors
  - Persistent storage
  - Define access control policies
  - Select control flow mechanism
  - Identify boundary conditions

Select control flow mechanism

- Control flow = sequencing of actions (operations in an OO system) in a system
- Design issue rather than an Analysis issue
  - During Analysis, we simple assume that all objects are running simultaneously, executing operations any time they need to execute them
  - During Design, we need to take into account that not every object has the luxury of running on its own processor
- Three possible control flow mechanisms:
  - Procedure-driven control
  - Event-driven control
  - Threads

Select control flow mechanism

Procedure-driven control

- Operations wait for input whenever they need data from an actor
- Mostly in legacy systems and systems written in procedural languages

Event-driven control

- A main loop waits for an external event
- Whenever an event becomes available, it is dispatched to the appropriate object
- Simpler structures, inputs centralized in the loop

Threads

- The system can create an arbitrary number of threads, each responding to a different event
- If a thread needs additional data, it waits for inputs

Command Design Pattern

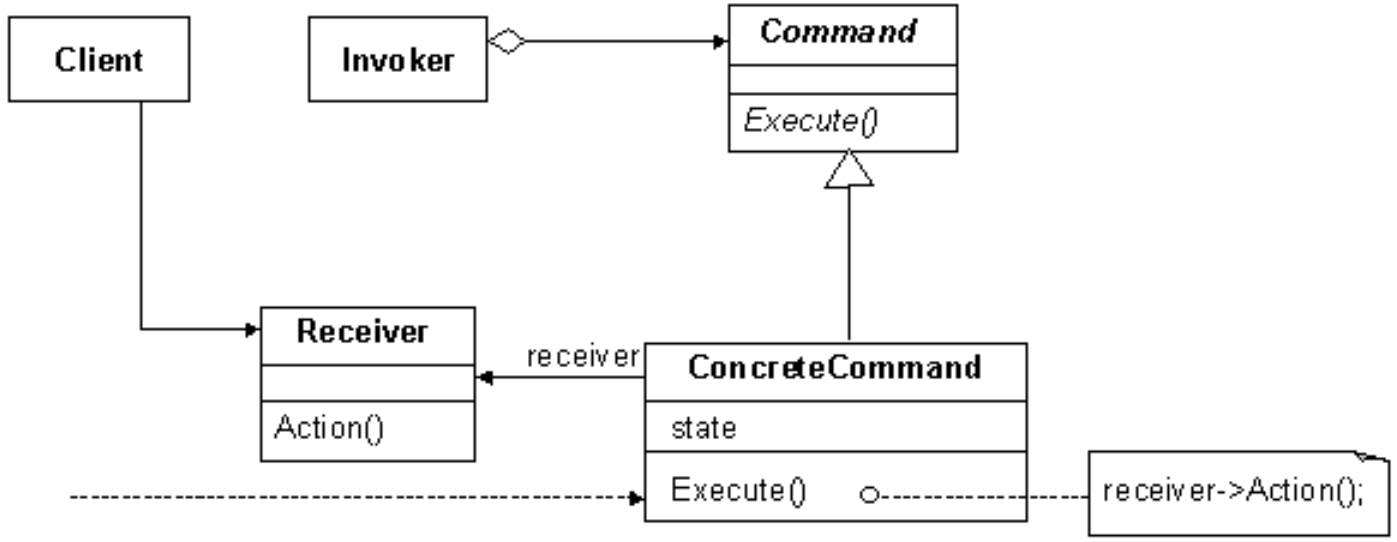
- Sometimes it's necessary to issue requests to objects without knowing anything about the operation being requested or the receiver of the request.
- **Intent**
  - Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations.

Command Design Pattern

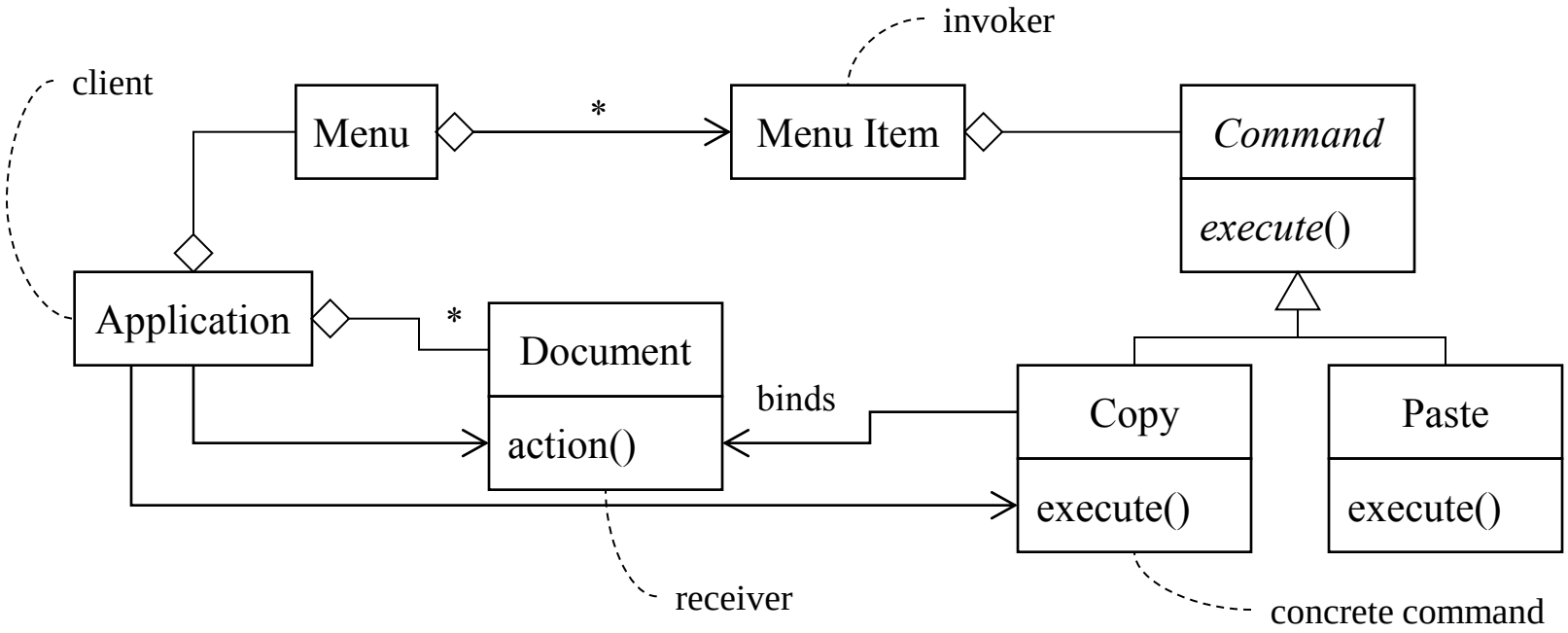
Participants

- *Command*
  - declares an interface for executing an operation.
- *ConcreteCommand*
  - defines a binding between a Receiver object and an action.
  - implements Execute by invoking the corresponding operation(s) on Receiver.
- *Client*
  - creates a ConcreteCommand object and sets its receiver.
- *Invoker*
  - invokes Execute on the command to carry out the request.
- *Receiver*
  - knows how to perform the operations associated with carrying out a request. Any class may serve as a Receiver.

Command Design Pattern



Command Design Pattern - Example



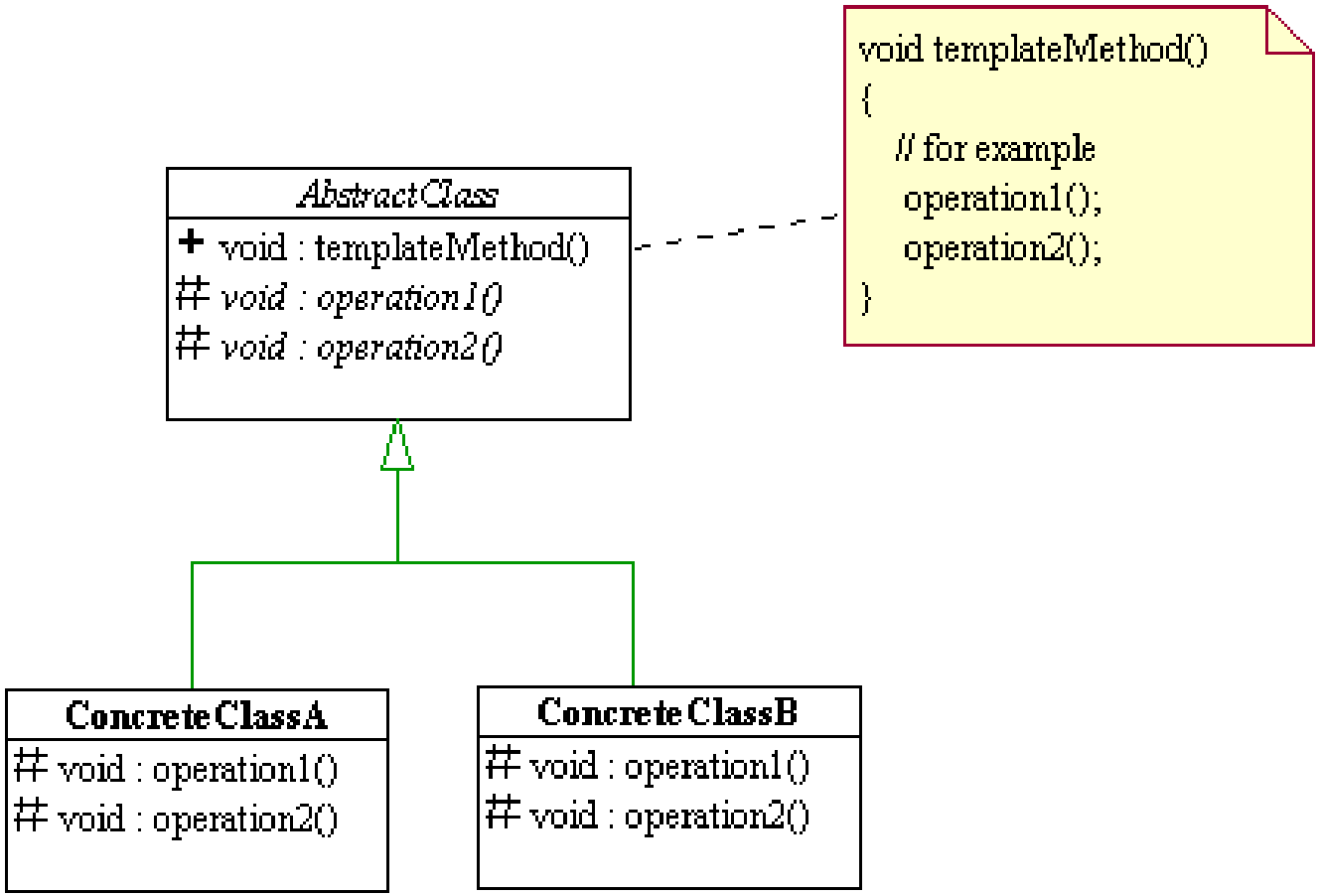
- Menu items and operations on documents are decoupled
- Enables us to centralize control flow in the document objects (Copy and Paste commands) instead of spreading it accross boundary objects (MenuItem) and entity object (Document)

Template Method Design Pattern

- **Intent**
  - Define the skeleton of an algorithm in an operation, deferring some steps to subclasses.

- Write an abstract class that contains only part of the logic needed to accomplish its purpose.
- Organize the class so that its concrete methods call an abstract method where the missing logic would have appeared.
- Provide the missing logic in subclass' methods that override the abstract methods.

Template Method Design Pattern



SYSC-3020 — Introduction to Software Engineering

- System design concepts
  - Definitions: Architecture (subsystem decomposition), coupling, cohesion
  - Layers and partitions
  - Software architecture (MVC, Observer pattern)
  - Process architecture (UML notation and Distribution patterns)
- System design process
  - Identify design goals
  - Initial subsystem decomposition
  - Map subsystems to components and processors
  - Persistent storage
  - Define access control policies
  - Select control flow mechanism
  - Identify boundary conditions

Boundary Conditions

- Most system design effort is concerned with steady-state behavior.
- However, system design phase must also address boundaries of system
  - **Initialization**
    - Describes how the system is brought from a non initialized state to steady-state ("startup use cases").
  - **Termination**
    - Describes what resources are cleaned up and which systems are notified upon termination ("termination use cases").
  - **Failure (Exception Handling)**
    - An exception is an unexpected event or error that occurs during the execution of the system
    - Sources: User error, external problems (network or hardware failure – power supply), software bug
    - Good system design foresees fatal failures ("failure use cases").
    - Exception handling: catch exceptions, treat them to minimize damage

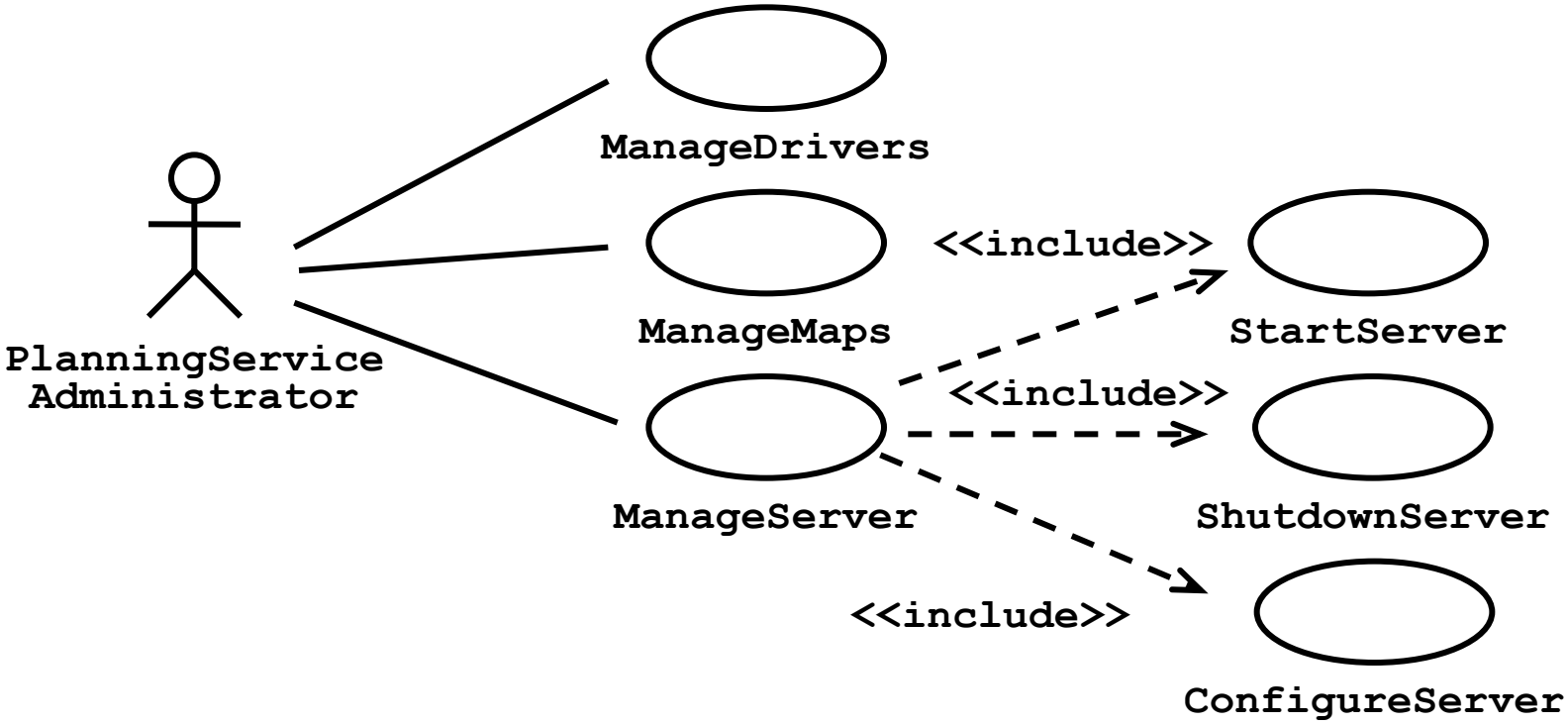
Boundary Condition Questions

- **Initialization**
  - How does the system start up?
    - What data need to be accessed at startup time?
    - What services have to registered?
  - What does the user interface do at start up time?
    - How does it present itself to the user?
- **Termination**
  - Are single subsystems allowed to terminate?
  - Are other subsystems notified if a single subsystem terminates?
  - How are local updates communicated to the database?
- **Failure**
  - How does the system behave when a node or communication link fails? Are there backup communication links?
  - How does the system recover from failure? Is this different from initialization?

My Trip Example Questions

- Initialization and Configuration:
  - How are maps loaded into the PlanningService?
  - How is MyTrip installed in the car?
  - How does MyTrip knows which PlanningService to connect to? (configuration)
  - How are drivers added to the PlanningService?
- Termination
  - What if driver stops half-way through trip ?
- Exceptions:
  - Nonfunctional requirement: FRIEND tolerates connection failures
- These lead to new administration use cases
  - Usually treated separately

MyTrip Administration



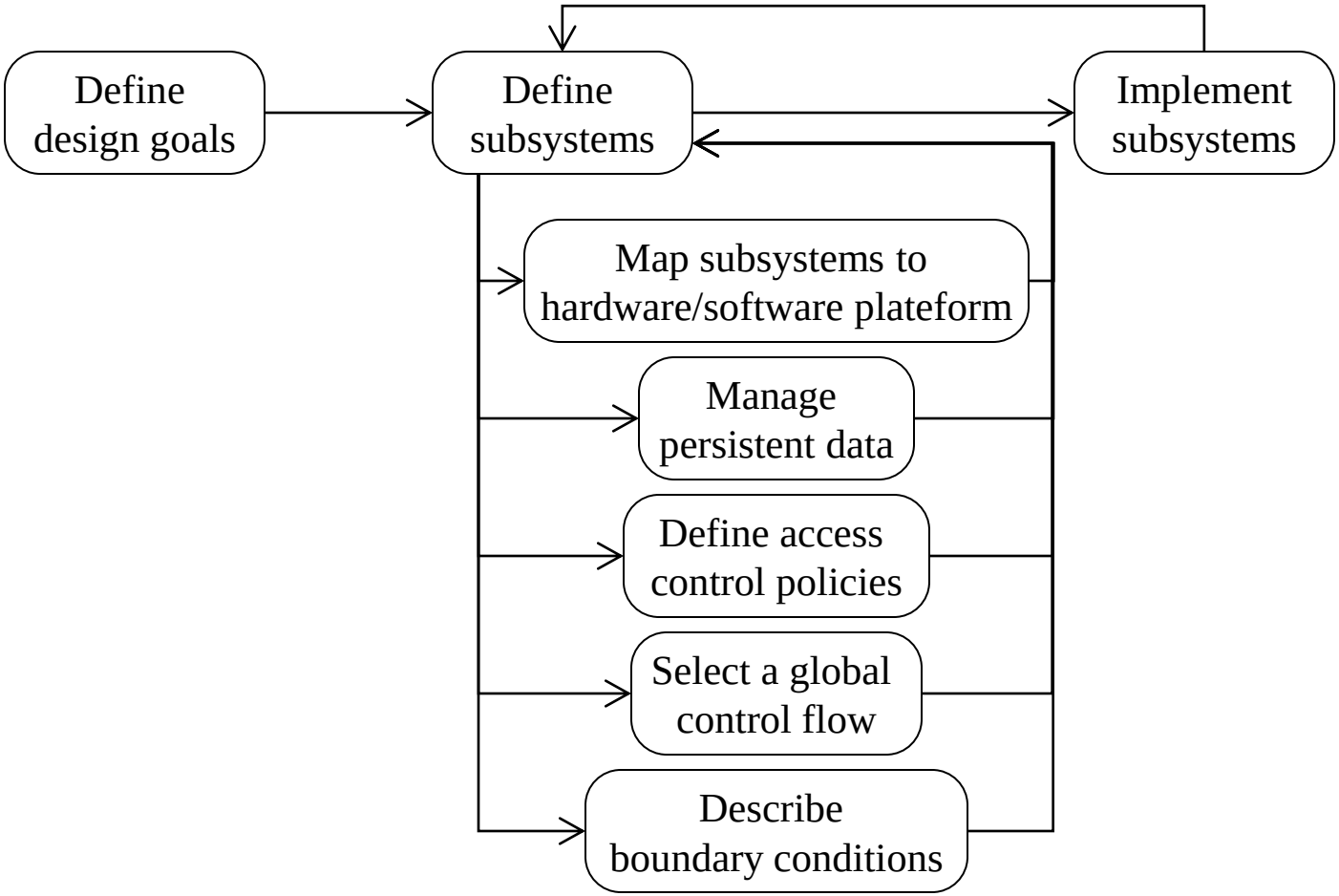
SYSC-3020 — Introduction to Software Engineering

- System design concepts
  - Definitions: Architecture (subsystem decomposition), coupling, cohesion
  - Layers and partitions
  - …
- System design process
  - Identify design goals
  - Initial subsystem decomposition
  - …
- Reviewing System Design

Reviewing System Design

- *Correct*: analysis model can be mapped to system design model
- *Complete*: Every requirement and design goal has been addressed
- *Consistent*: Does not contain any contradictions
- *Realistic*: The corresponding system can be implemented with current technology
- *Readable*: Developers can precisely understand the model

Activities in System Design



Documenting System Design

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