Design Phase Using UML

List of Material

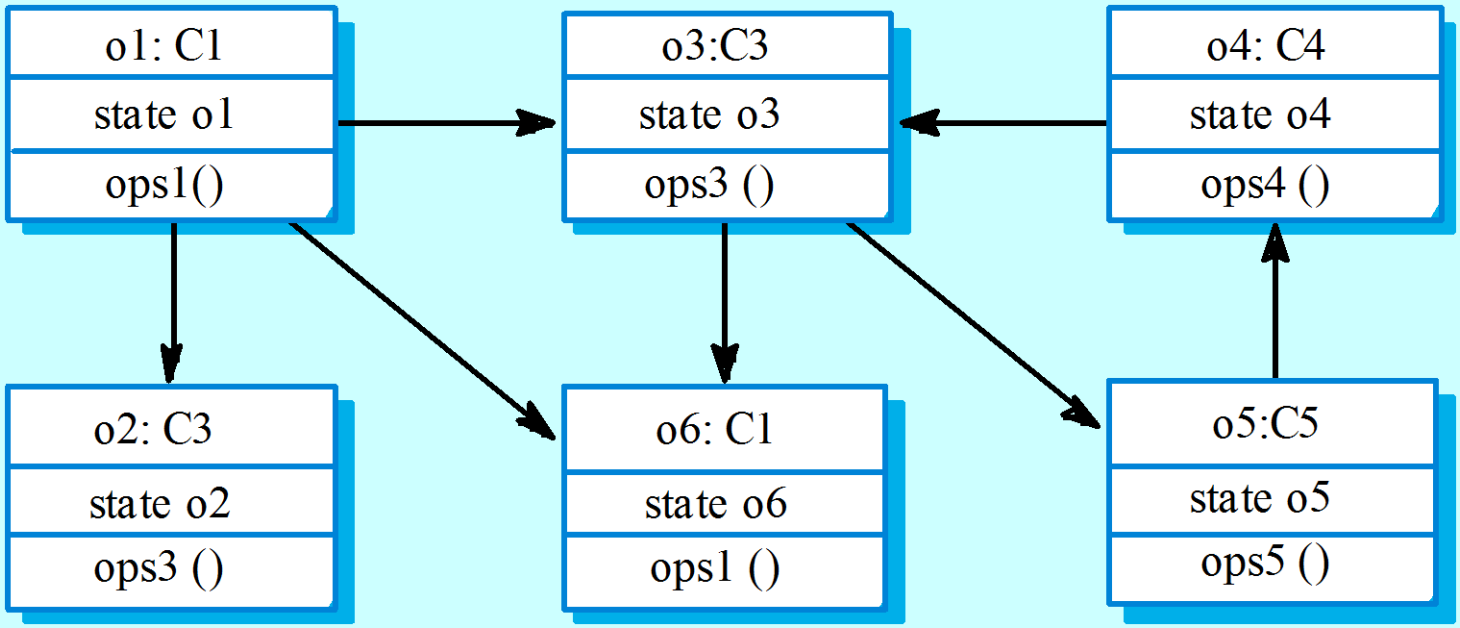
- Requirement Engineering Process
- Design Models
- Moving on to Design
- Class and Method Design
- Object Design Activities
- Collaboration Diagram and Sequence Diagram

Object Oriented Design

Characteristics of OOD

- Objects are abstractions of real-world or system entities and manage themselves.
- Objects are independent and encapsulate state and representation information.
- System functionality is expressed in terms of object services.
- Shared data areas are eliminated. Objects communicate by message passing.
- Objects may be distributed and may execute sequentially or in parallel.

Interacting objects



Advantages of OOD

- Easier maintenance. Objects may be understood as stand-alone entities.
- Objects are potentially reusable components.
- For some systems, there may be an obvious mapping from real world entities to system objects.

Objects and object classes

- Objects are entities in a software system which represent instances of real-world and system entities.
- Object classes are templates for objects. They may be used to create objects.
- Object classes may inherit attributes and services from other object classes.

Object communication

- Conceptually, objects communicate by message passing.
- Messages
  - The name of the service requested by the calling object;
  - Copies of the information required to execute the service and the name of a holder for the result of the service.
- In practice, messages are often implemented by procedure calls
  - Name = procedure name;
  - Information = parameter list.

Concurrent objects

- The nature of objects as self-contained entities make them suitable for concurrent mplementation.
- The message-passing model of object communication can be implemented directly if objects are running on separate processors in a distributed system.

An object-oriented design process

- Structured design processes involve developing a number of different system models.
- They require a lot of effort for development and maintenance of these models and, for small systems, this may not be cost-effective.
- However, for large systems developed by different groups design models are an essential communication mechanism.

Process stages

- Highlights key activities without being tied to any proprietary process such as the RUP.
  - Define the context and modes of use of the system;
  - Design the system architecture;
  - Identify the principal system objects;
  - Develop design models;
  - Specify object interfaces.

Architectural design

- Once interactions between the system and its environment have been understood, you use this information for designing the system architecture.
- A layered architecture as discussed in Chapter 11 is appropriate for the weather station
  - Interface layer for handling communications;
  - Data collection layer for managing instruments;
  - Instruments layer for collecting data.
- There should normally be no more than 7 entities in an architectural model.

Object identification

- Identifying objects (or object classes) is the most difficult part of object oriented design.
- There is no 'magic formula' for object identification. It relies on the skill, experience
  and domain knowledge of system designers.
- Object identification is an iterative process. You are unlikely to get it right first time.

Approaches to identification

- Use a grammatical approach based on a natural language description of the system (used in Hood OOD method).
- Base the identification on tangible things in the application domain.
- Use a behavioural approach and identify objects based on what participates in what behaviour.
- Use a scenario-based analysis. The objects, attributes and methods in each scenario are identified.

Further objects and object refinement

- Use domain knowledge to identify more objects and operations
  - Weather stations should have a unique identifier;
  - Weather stations are remotely situated so instrument failures have to be reported automatically. Therefore attributes and operations for self-checking are required.
- Active or passive objects
  - In this case, objects are passive and collect data on request rather than autonomously. This introduces flexibility at the expense of controller processing time.

Design Models

Design models

- Design models show the objects and object classes and relationships between these entities.
- Static models describe the static structure of the system in terms of object classes and relationships.
- Dynamic models describe the dynamic interactions between objects.

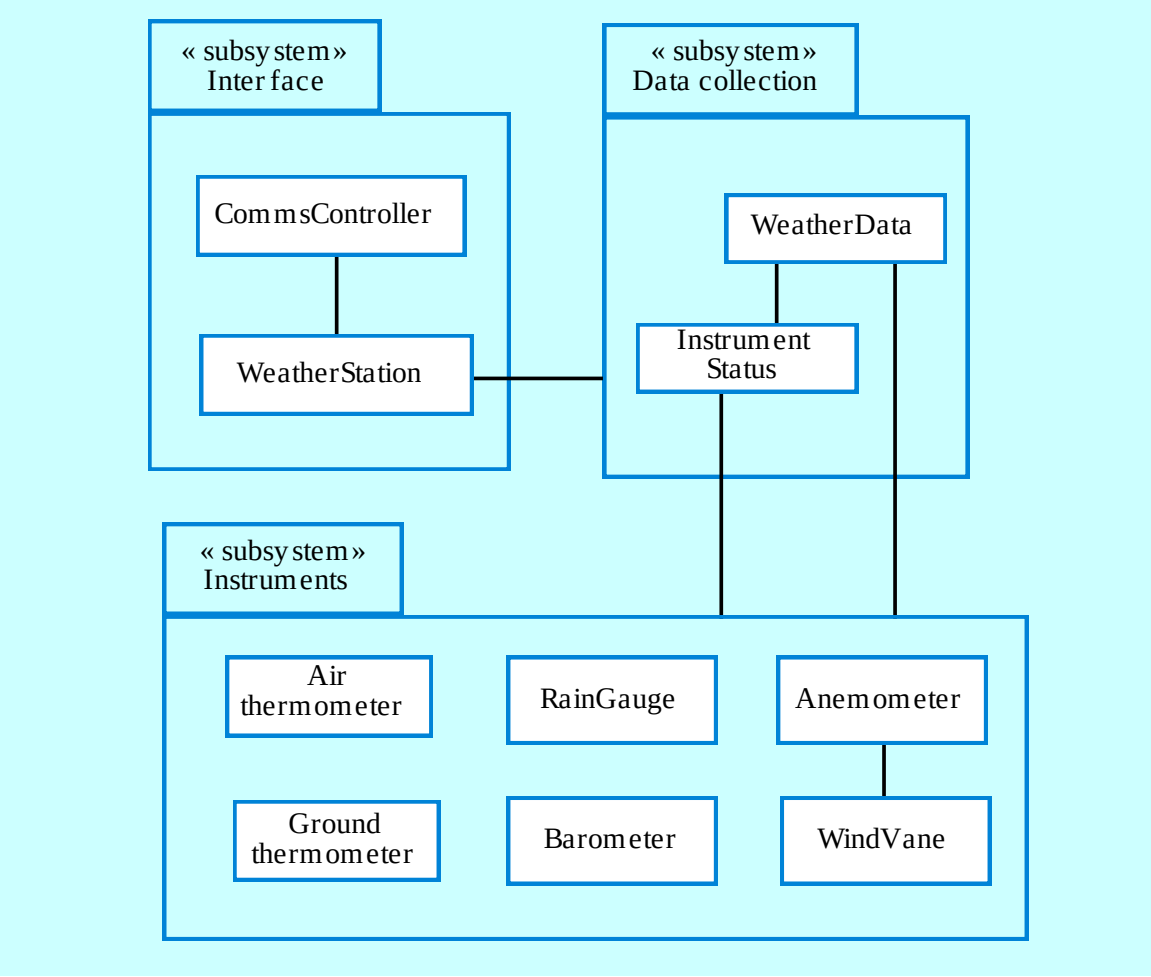
Examples of design models

- Sub-system models that show logical groupings of objects into coherent subsystems.
- Sequence models that show the sequence of object interactions.
- State machine models that show how individual objects change their state in response to events.
- Other models include use-case models, aggregation models, generalisation models, etc.

Subsystem models

- Shows how the design is organised into logically related groups of objects.
- In the UML, these are shown using packages - an encapsulation construct. This is a logical model. The actual organisation of objects in the system may be different.

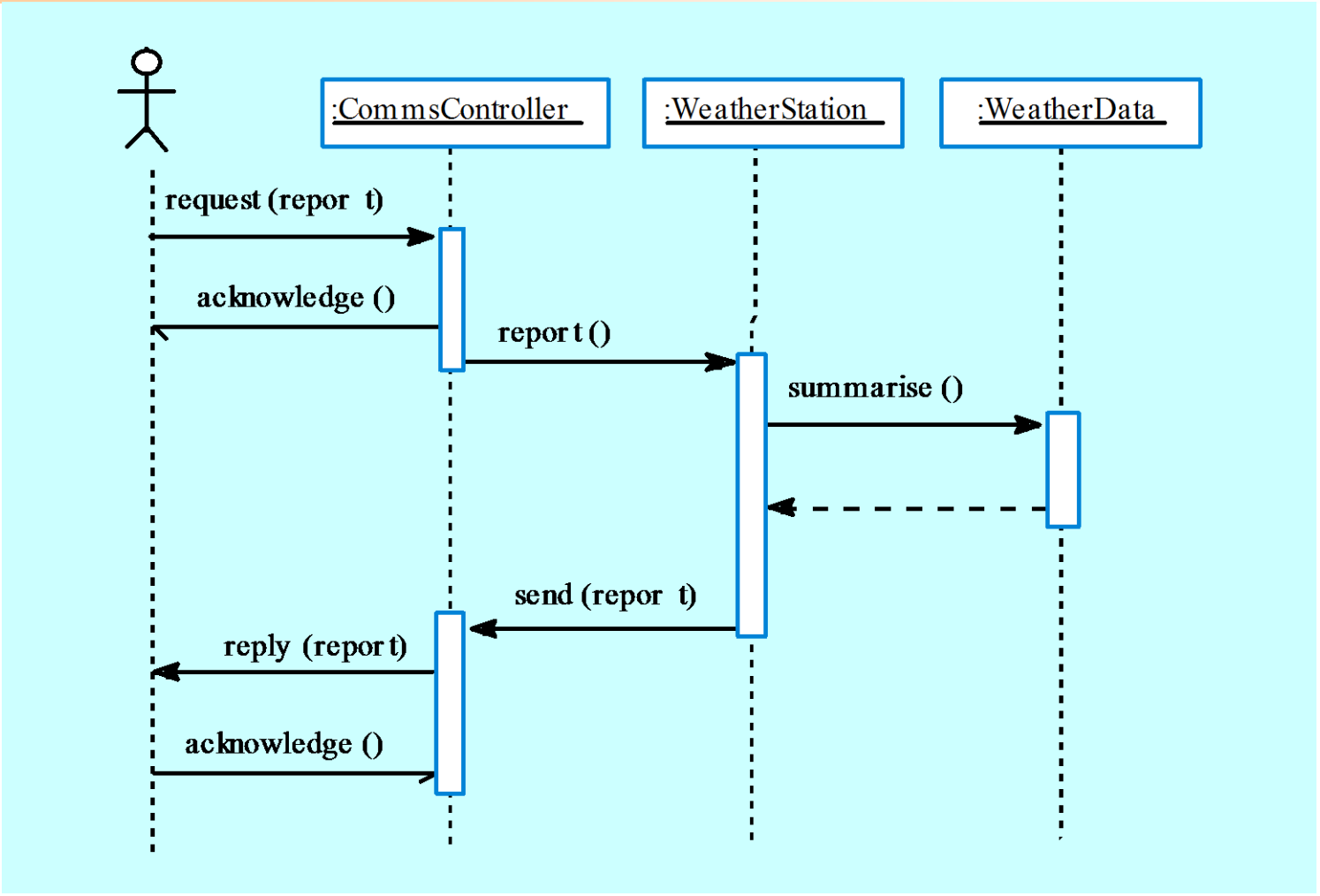
Example of Subsystem models



Sequence models

- Sequence models show the sequence of object interactions that take place
  - Objects are arranged horizontally across the top;
  - Time is represented vertically so models are read top to bottom;
  - Interactions are represented by labelled arrows, Different styles of arrow represent different types of interaction;
  - A thin rectangle in an object lifeline represents the time when the object is the controlling object in the system.

Data collection sequence



Object interface specification

- Object interfaces have to be specified so that the objects and other components can be designed in parallel.
- Designers should avoid designing the interface representation but should hide this in the object itself.
- Objects may have several interfaces which are viewpoints on the methods provided.
- The UML uses class diagrams for interface specification but Java may also be used.

Weather station interface

```
interface WeatherStation {

        public void WeatherStation () ;

        public void startup () ;
        public void startup (Instrument i) ;

        public void shutdown () ;
        public void shutdown (Instrument i) ;

        public void reportWeather ( ) ;

        public void test () ;
        public void test ( Instrument i ) ;

        public void calibrate ( Instrument i) ;

        public int getID () ;

} /WeatherStation
```

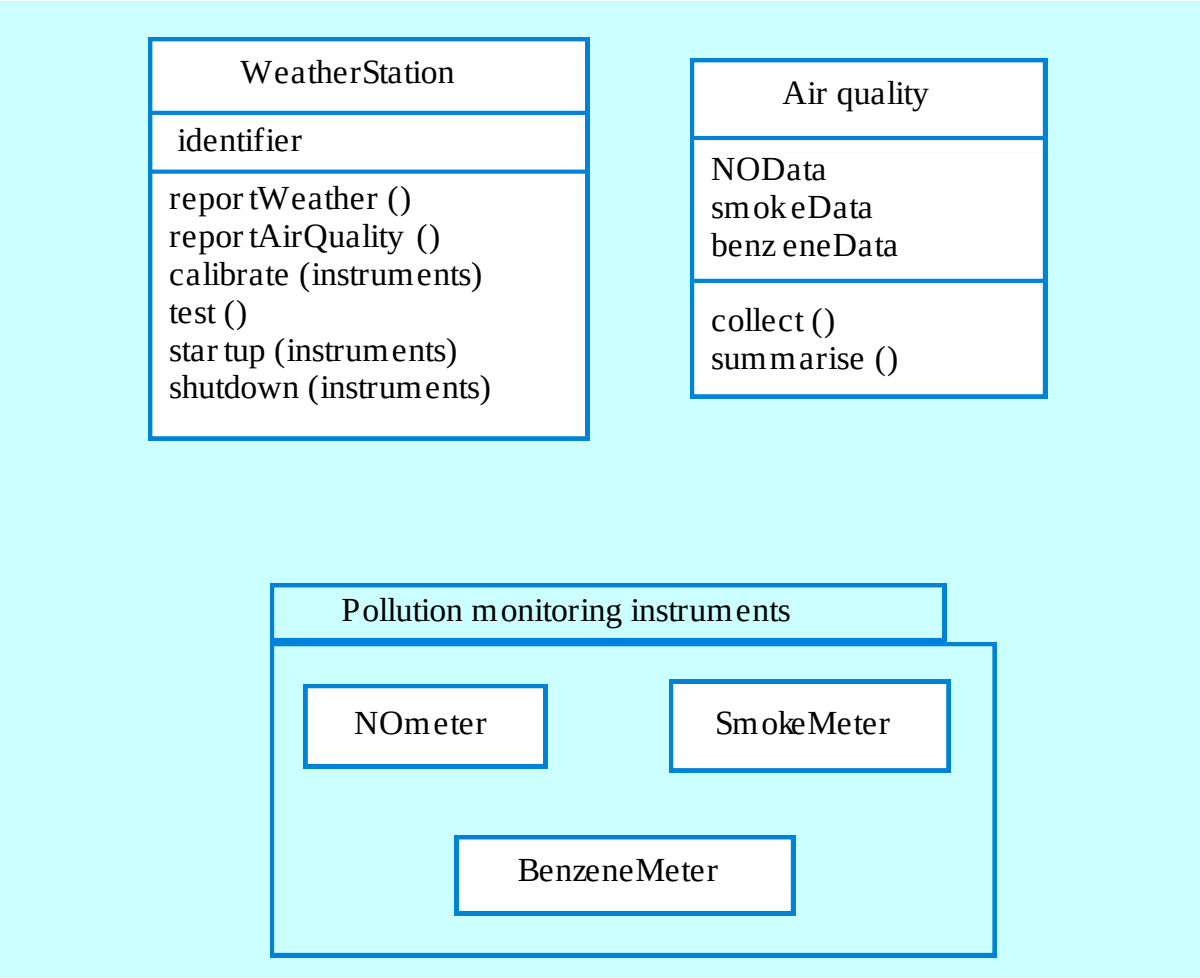
Design evolution

- Hiding information inside objects means that changes made to an object do not affect other objects in an unpredictable way.
- Assume pollution monitoring facilities are to be added to weather stations. These sample the air and compute the amount of different pollutants in the atmosphere.
- Pollution readings are transmitted with weather data.

Changes required

- Add an object class called Air quality as part of WeatherStation.
- Add an operation reportAirQuality to WeatherStation. Modify the control software to collect pollution readings.
- Add objects representing pollution monitoring instruments.

Pollution monitoring



Moving on to Design

Key Ideas

- The purpose of the analysis phase is to figure out what the business needs. The purpose of the design phase is to figure out *how to provide it*.
- The steps in both analysis and design phases are highly *interrelated* and may require much “going back and forth”

OO Analysis and Design Foundation

- Use-case driven
- Architecture centric
- Iterative and incremental

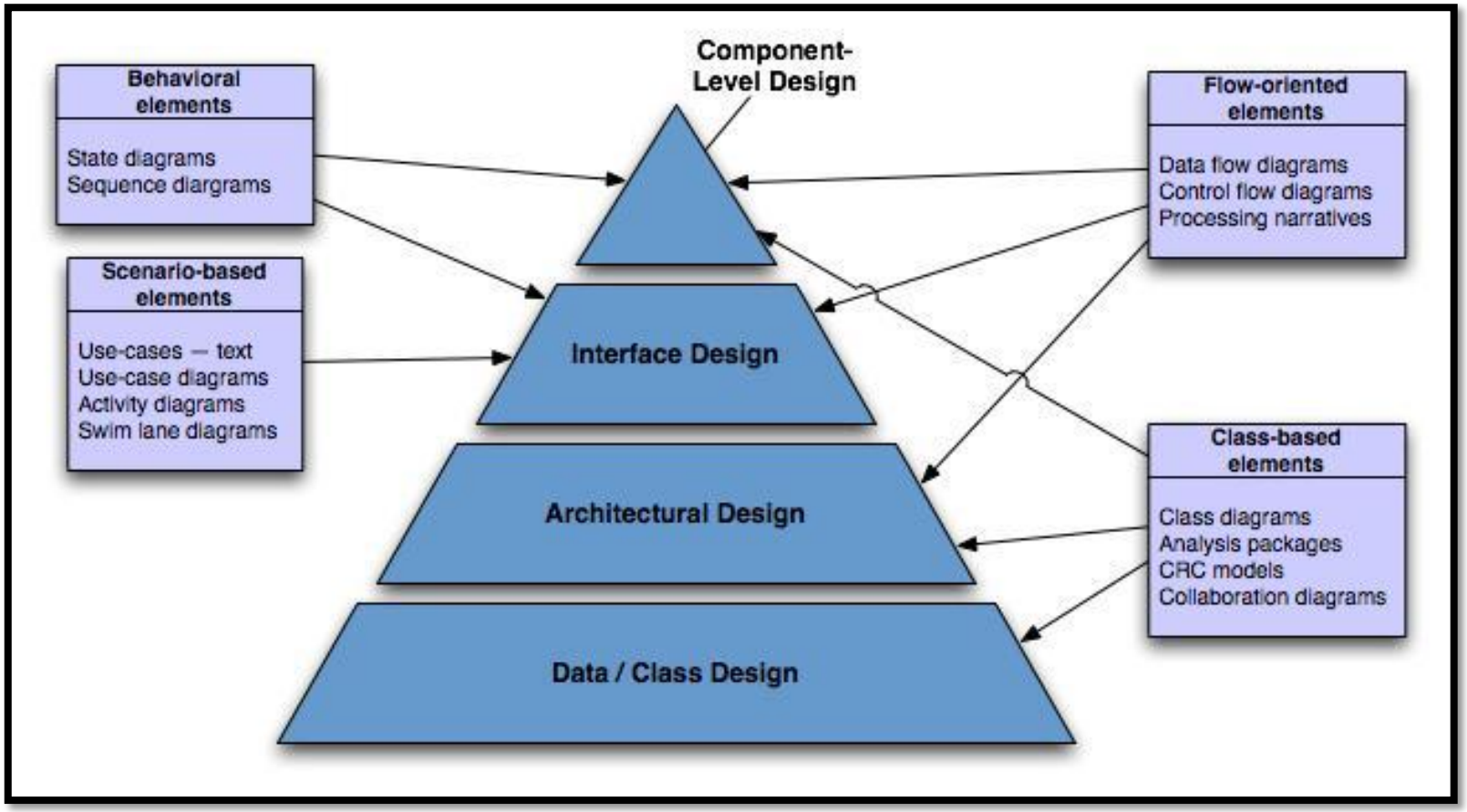
Combining Three Views

- Functional
- Static
- Dynamic

A “Minimalist” Approach

- Planning
- Gathering requirements
- Perform a series of “builds”
- Use results of each build as feedback for design and implementation

Analysis into Design (Remember?)



Avoid Classic Design Mistakes

- Reducing design time
- Feature creep
- Silver bullet syndrome
- Switching tools in mid-project

Factoring

- Creating modules that account for similarities and differences between units of interest
- New classes
  - Generalization
  - Aggregation
- Abstracting
- Refinement

Partitions and Collaborations

- Creating “subsystems” or larger units
- Grouping units that collaborate
- May have collaboration among units or partitions
- The more messages or contracts between objects, the more likely they are in the same partition

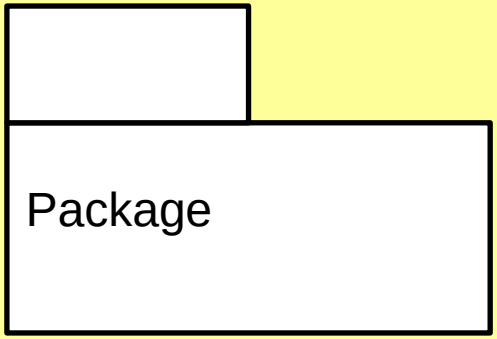
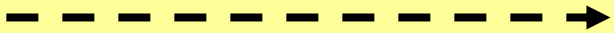
Purpose of Layers

- Model-view-controller (MVC) architecture
  - Models implement application logic
  - Views and controllers do user interfaces
- Separating application logic from user interface logic

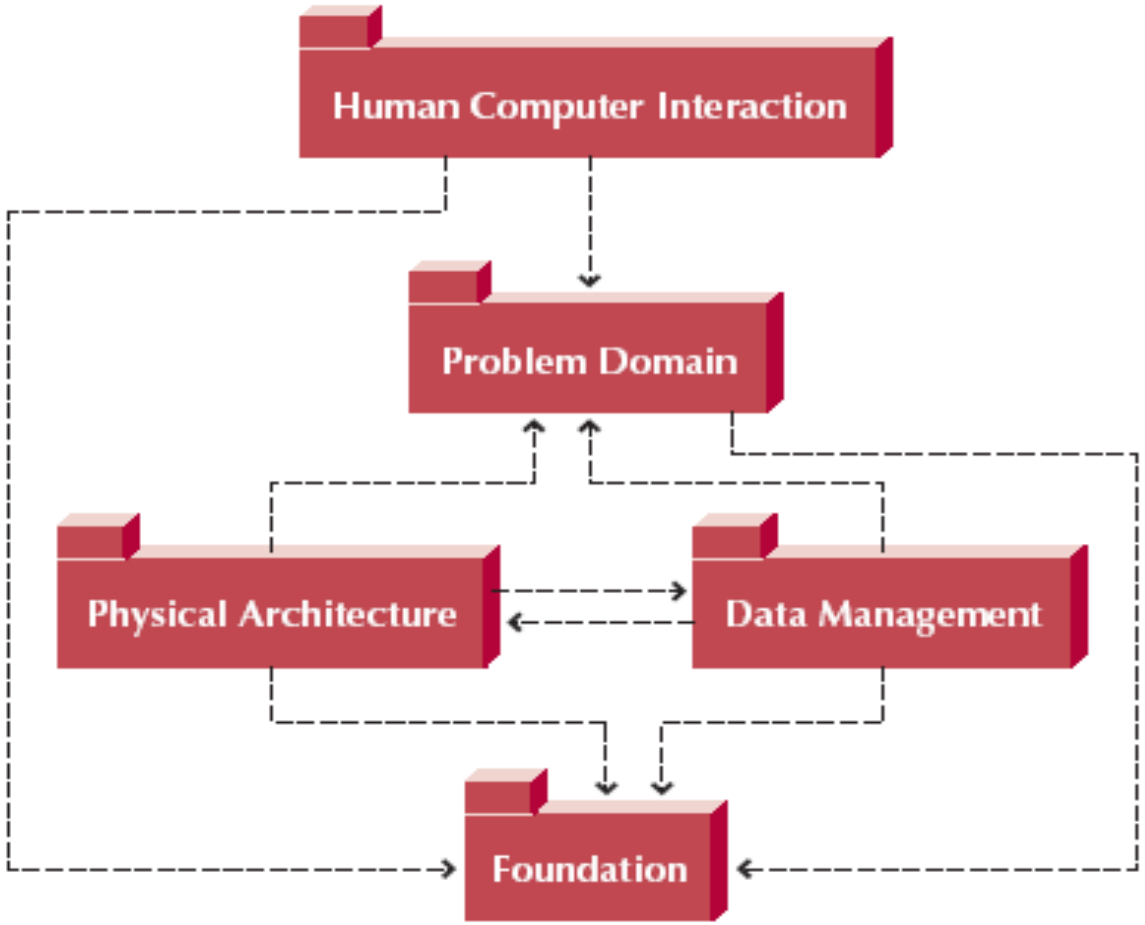
Packages

- Logical grouping of UML elements
- Simplifies UML diagrams
  - Groups related elements into a single higher-level element
- Dependency relationships
  - Shows a dependency between packages

Syntax for Package Diagram

A PACKAGE	
A DEPENDENCY RELATIONSHIP	

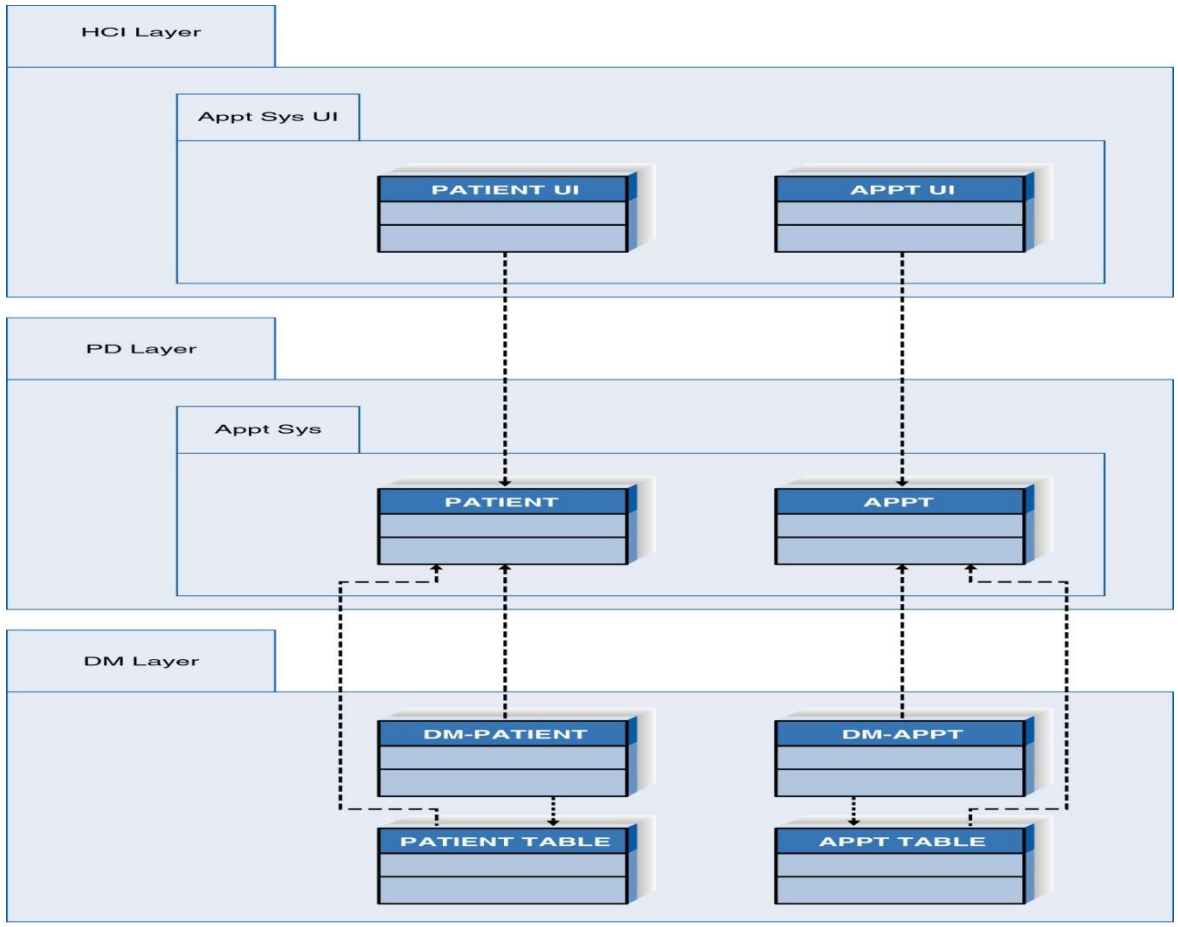
Package Diagram of Dependency Relationships Among Layers



Modification Dependency

- Indicates that a change in one package could cause a change to be required in another package.
- Example:
  - A change in one method will cause the interface for all objects of this class to change. Therefore, all classes that have objects that send messages to the instances of the modified class could have to be modified.

Package Diagram of Appointment System



Dennis: SAD
Fig: 9-6 W-42 100% of size
Fine Line Illustrations (516) 501-0400

Package Diagram of the PD Layer for the Appointment System

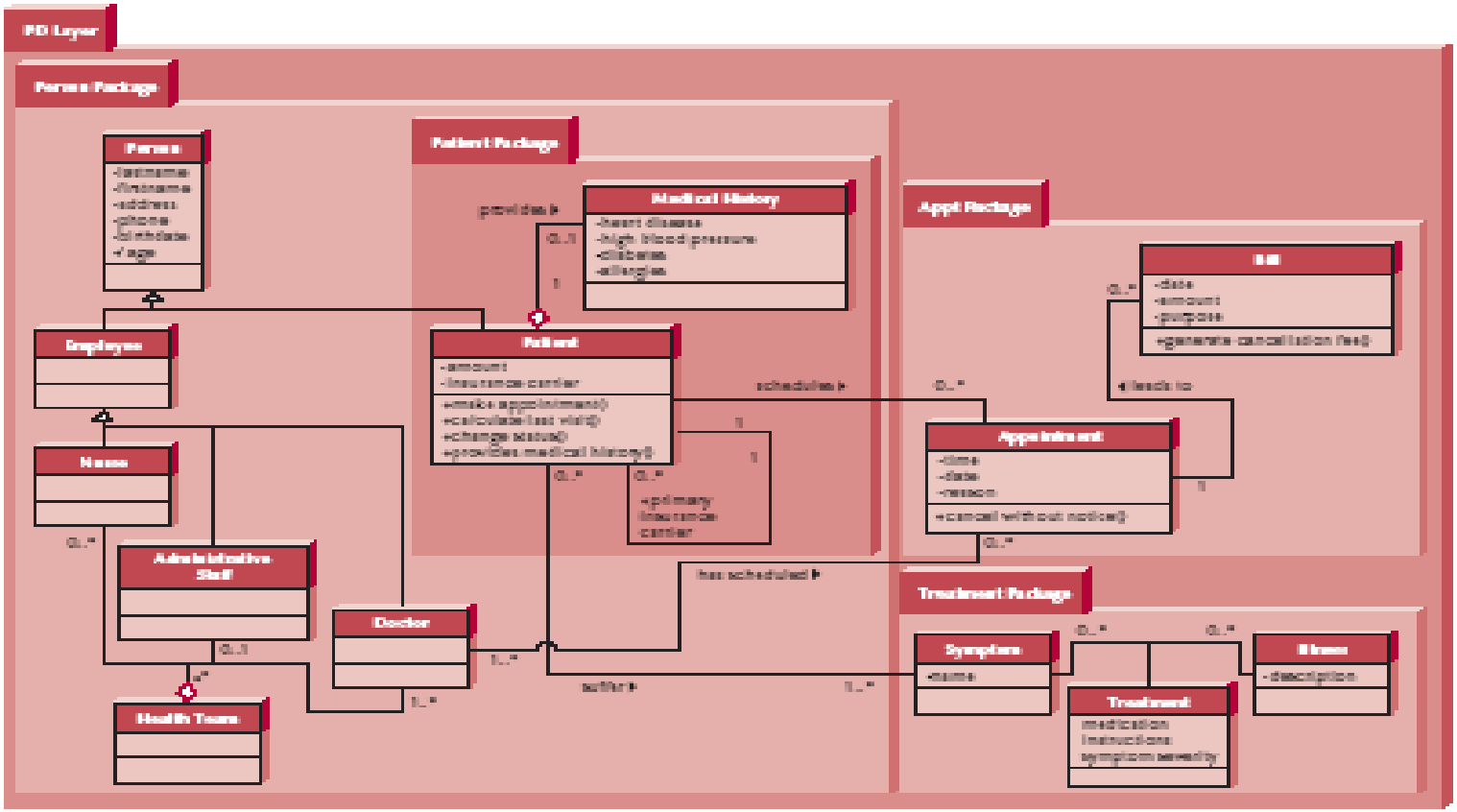
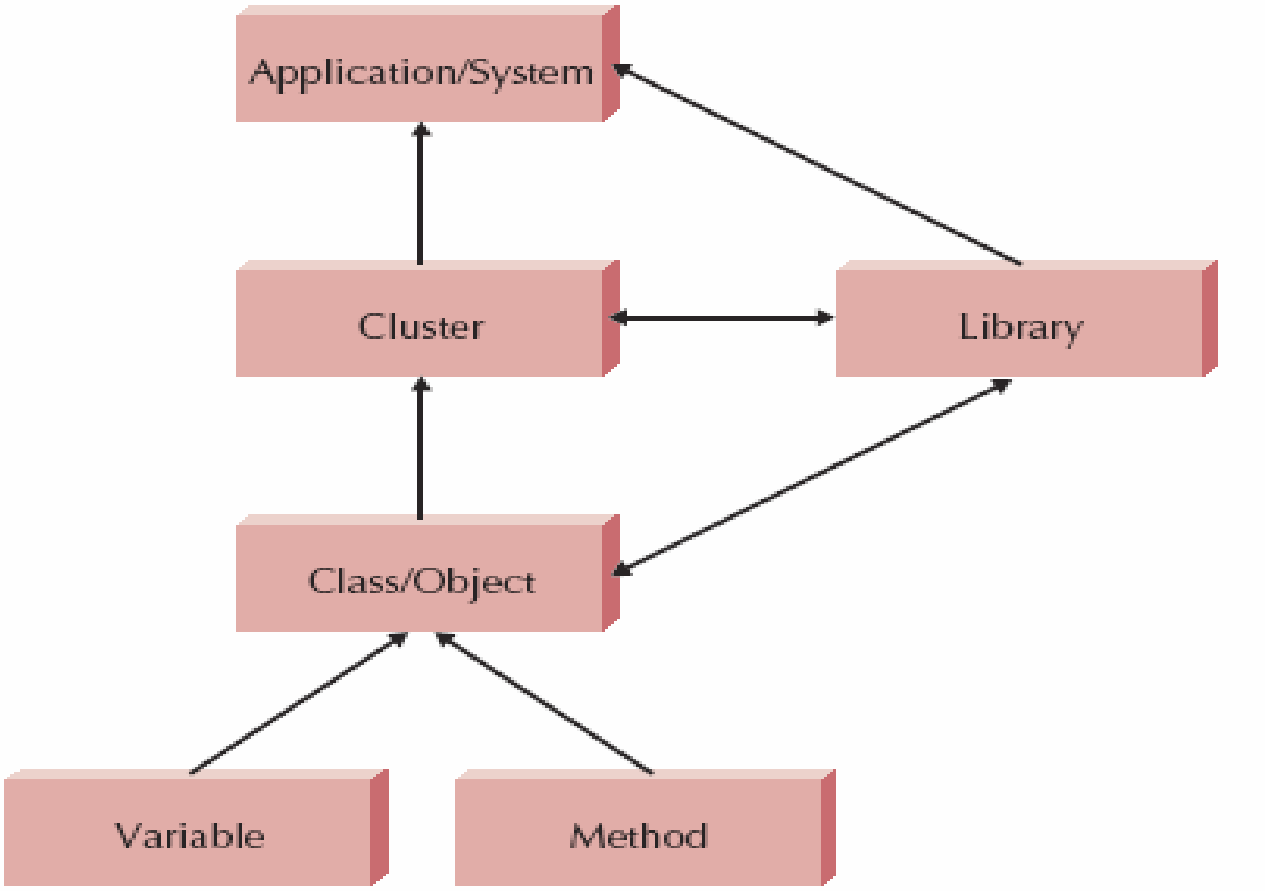


FIGURE 9-6 Package Diagram of the PD Layer for the Appointment System

Class and Method Design

Levels of Abstraction



Elements

- Classes
- Objects
- Attributes
- States
- Methods
- Messages

Encapsulation

- Hiding the content of the object from outside view
- Communication only through object's methods
- Key to reusability

Rumbaugh's Rules

Query operations should not be redefined
Methods that redefine inherited ones should only restrict the semantics of the inherited ones
The underlying semantics of the inherited method should never be changed
The signature (argument list) of the inherited method should never be changed

Object Design Activities

Types of Connascence

Type	Description
Name	If a method refers to an attribute, it is tied to the name of the attribute. If the attribute's name changes, the content of the method will have to change.
Type or Class	If a class has an attribute of type A, it is tied to the type of the attribute. If the type of the attribute changes, the attribute declaration will have to change.
Convention	A class has an attribute in which a range of values has a semantic meaning (e.g., account numbers whose values range from 1000 to 1999 are assets). If the range would change, then every method that used the attribute would have to be modified.
Algorithm	Two different methods of a class are dependent on the same algorithm to execute correctly (e.g., insert an element into an array and find an element in the same array). If the underlying algorithm would change, then the insert and find methods would also have to change.
Position	The order of the code in a method or the order of the arguments to a method is critical for the method to execute correctly. If either is wrong, then the method will, at least, not function correctly.

Source: Page-Jones, "Comparing Techniques by Means of Encapsulation and Connascence" and Page-Jones, *Fundamentals of Object-Oriented Design in UML.*

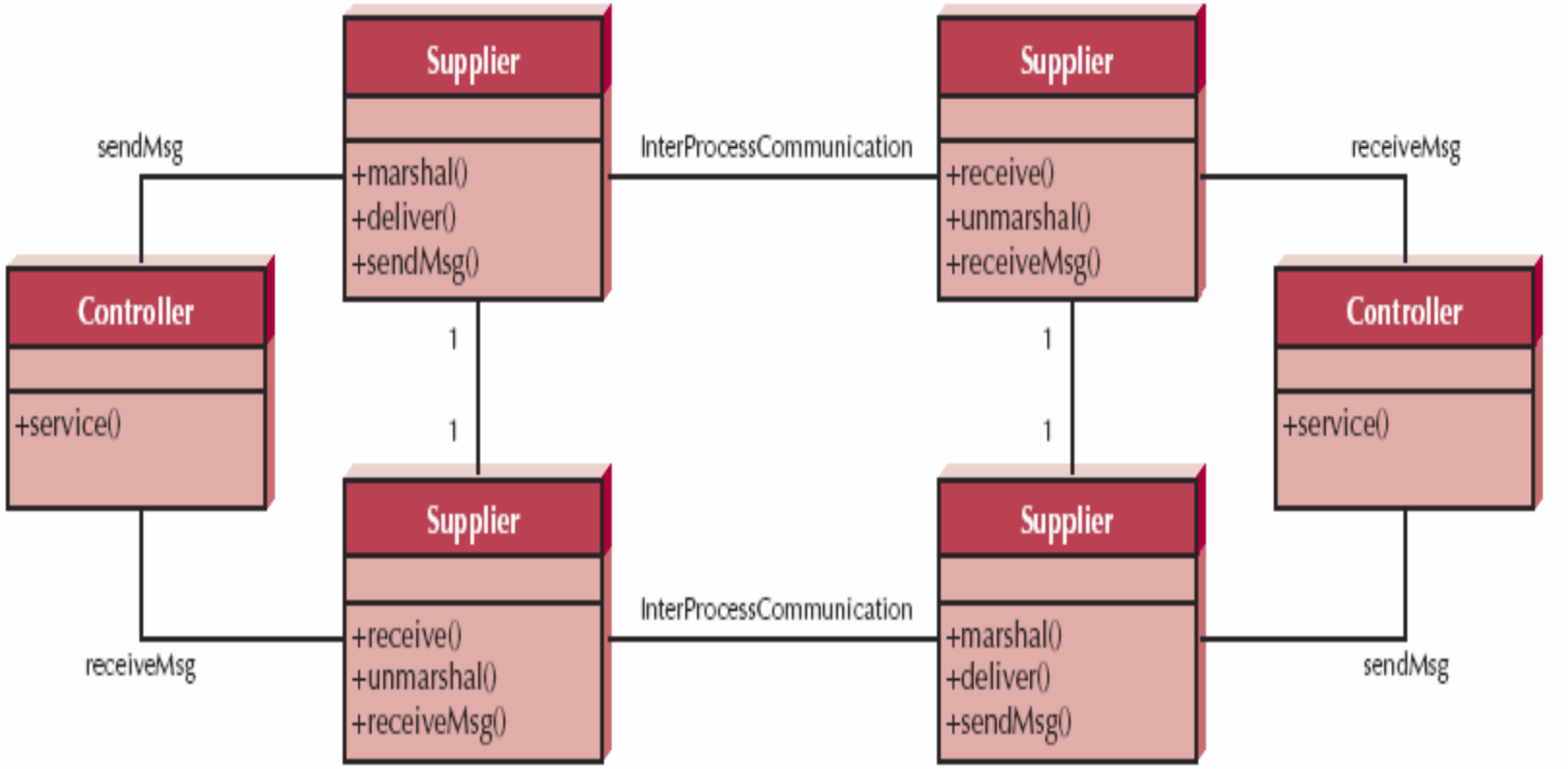
Additional Specification

- Ensure the classes are both necessary and sufficient for the problem
- Finalize the visibility of the attributes and methods of each class
- Determine the signature of every method of each class
- Define constraints to be preserved by objects

Identifying Opportunities for Reuse

- Analysis patterns
- Design patterns
- Frameworks
- Libraries
- components

Sample Design Pattern



Restructuring the Design

- Factoring
  - Separate aspects of a method or class into a new method or class
- Normalization
  - Identifies classes missing from the design
- Challenge inheritance relationships to ensure they only support a generalization/specialization semantics

Optimizing the Design

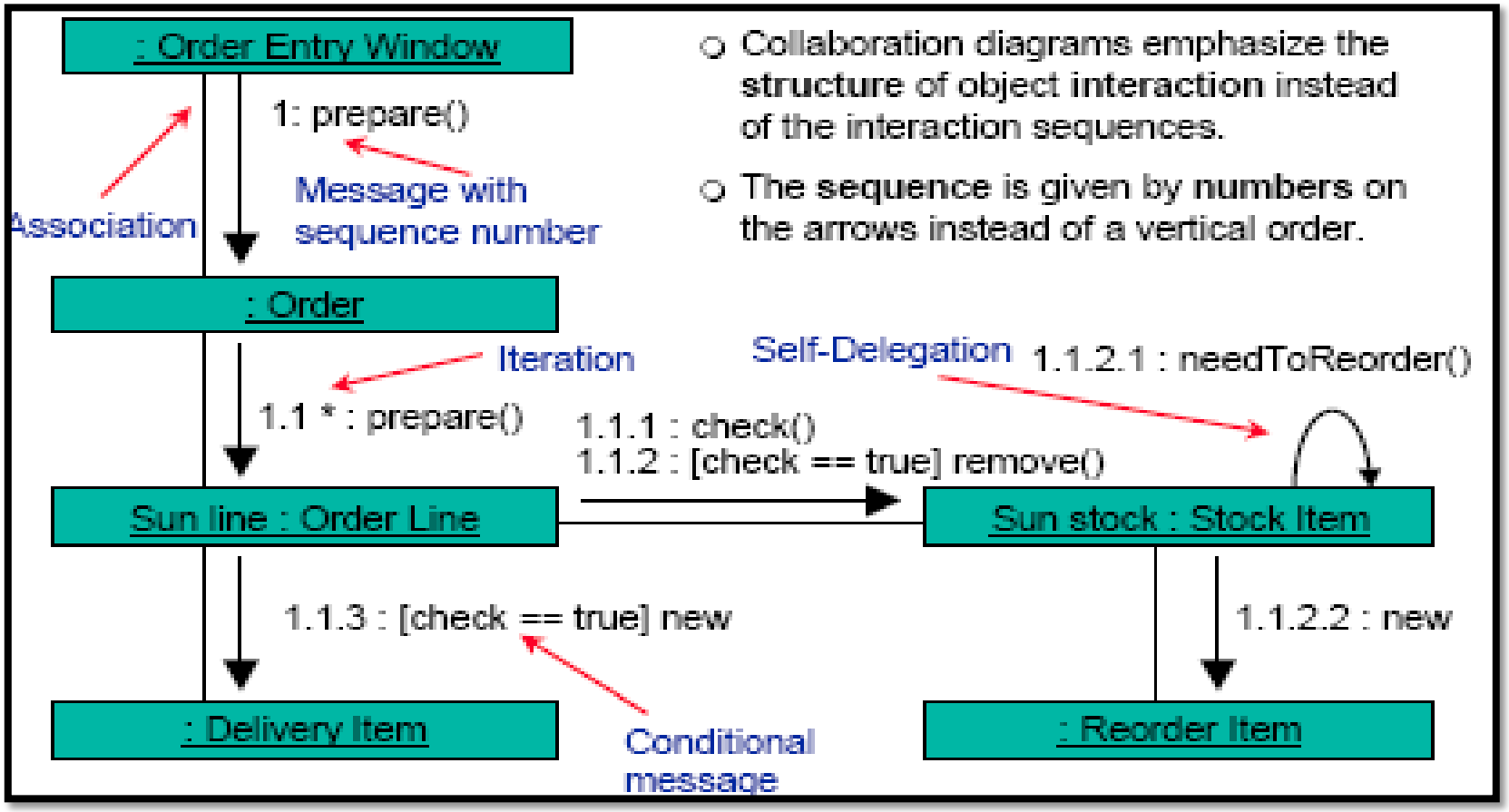
- Review access paths between objects
- Review each attribute of each class
- Review fan-out of each method
- Examine execution order of statements
- Create derived activities

Collaboration Diagram and Sequence Diagram

Collaboration Diagram (1)

- An alternative way to present a scenario
- Displays object interactions organized around objects and their relationships with other objects.
- Contains
  - Object, which is described in the rectangle
  - Links between object, shown as a line connecting with other objects.
  - Messages are shown as text and arrows that lead from the client to the supplier.

Collaboration Diagram (2)



Sequence Diagram

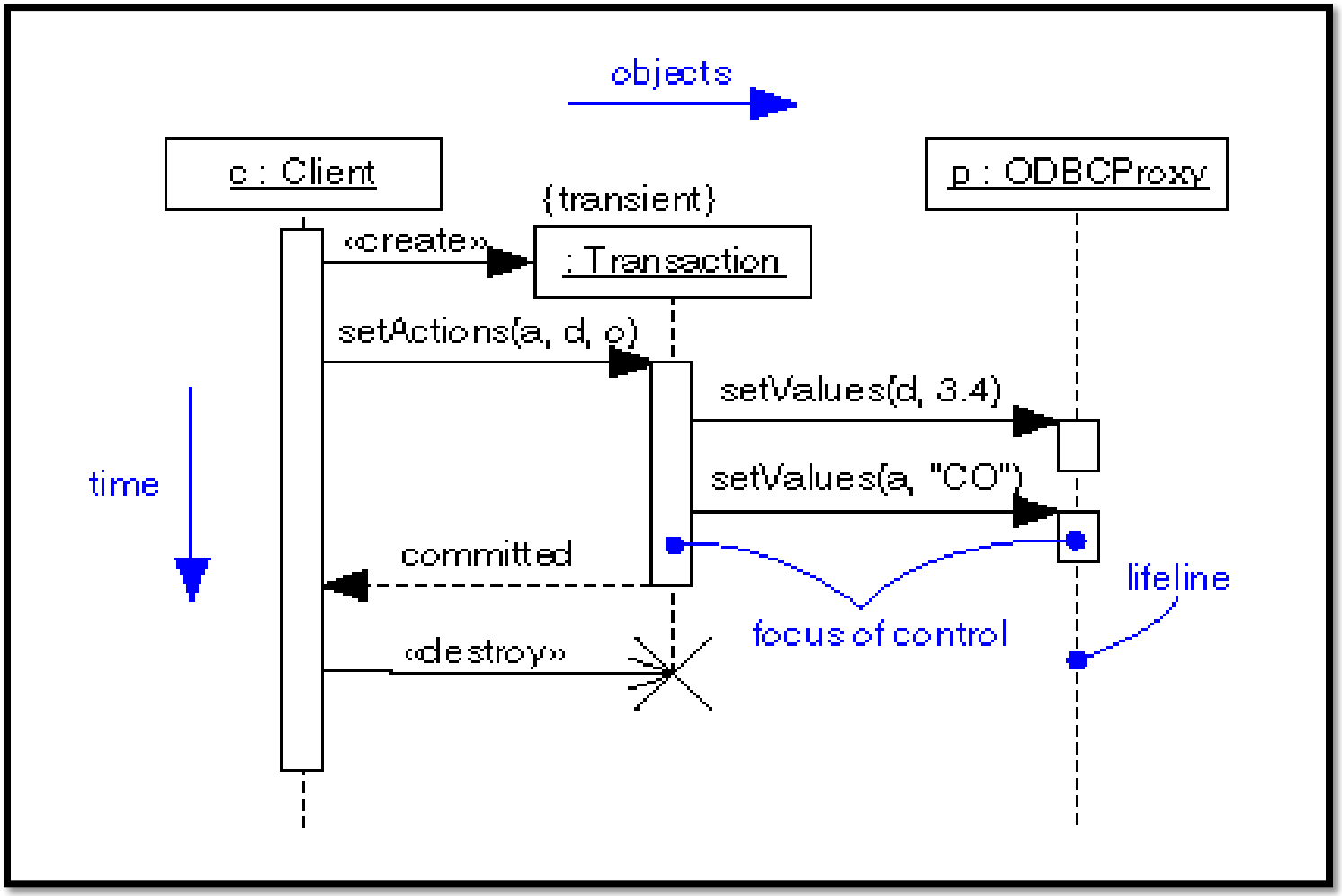
Sequence diagrams provide perspective scenario in order of time. Useful in early stages of analysis and design.

- **Collaboration diagrams memberikan suatu gambaran besar skenario karena kolaborasi yang terorganisasi antar obyek satu dengan yang lainnya.**
- **Digunakan lebih banyak pada fase desain**

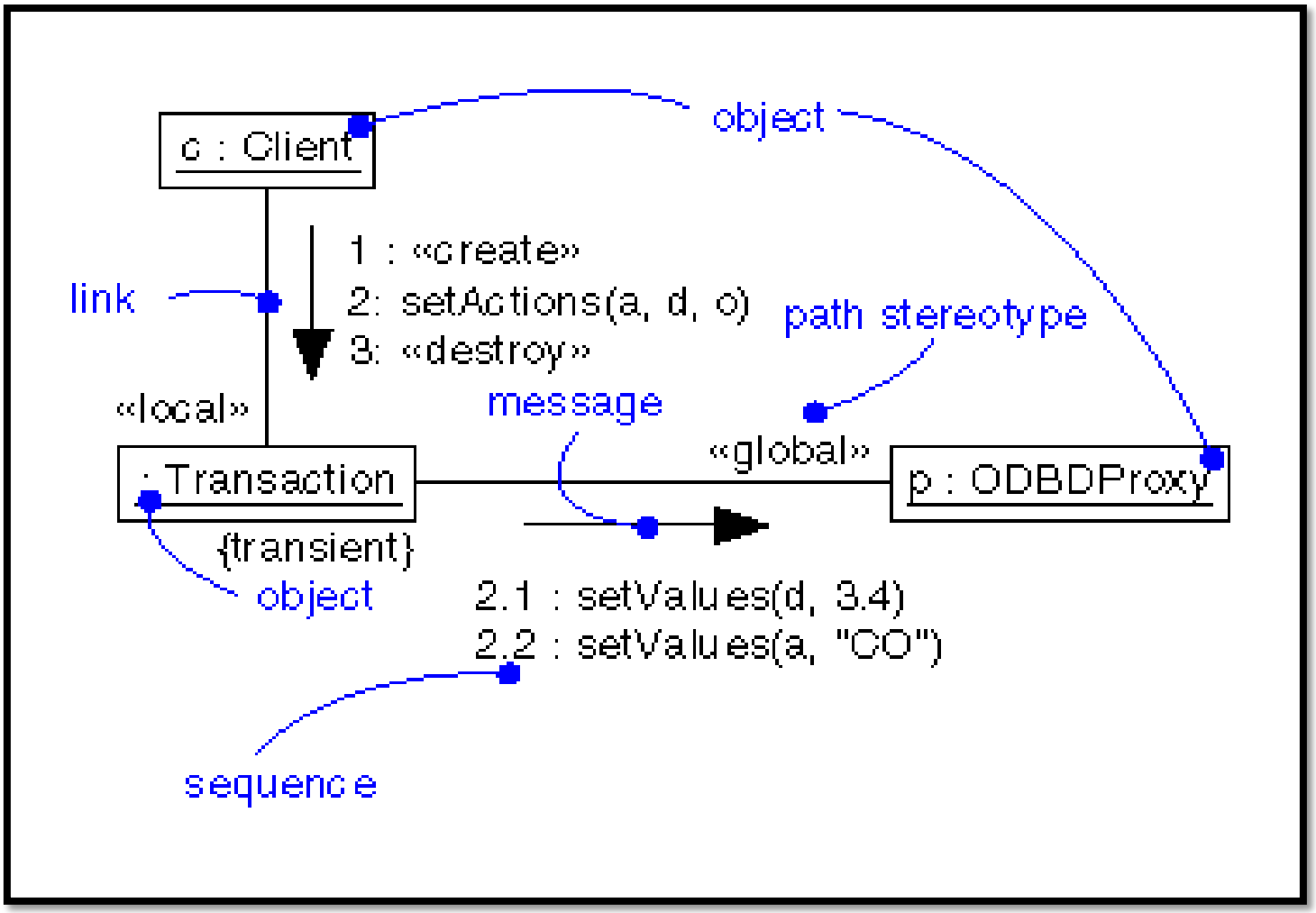
Collaboration VS Sequence

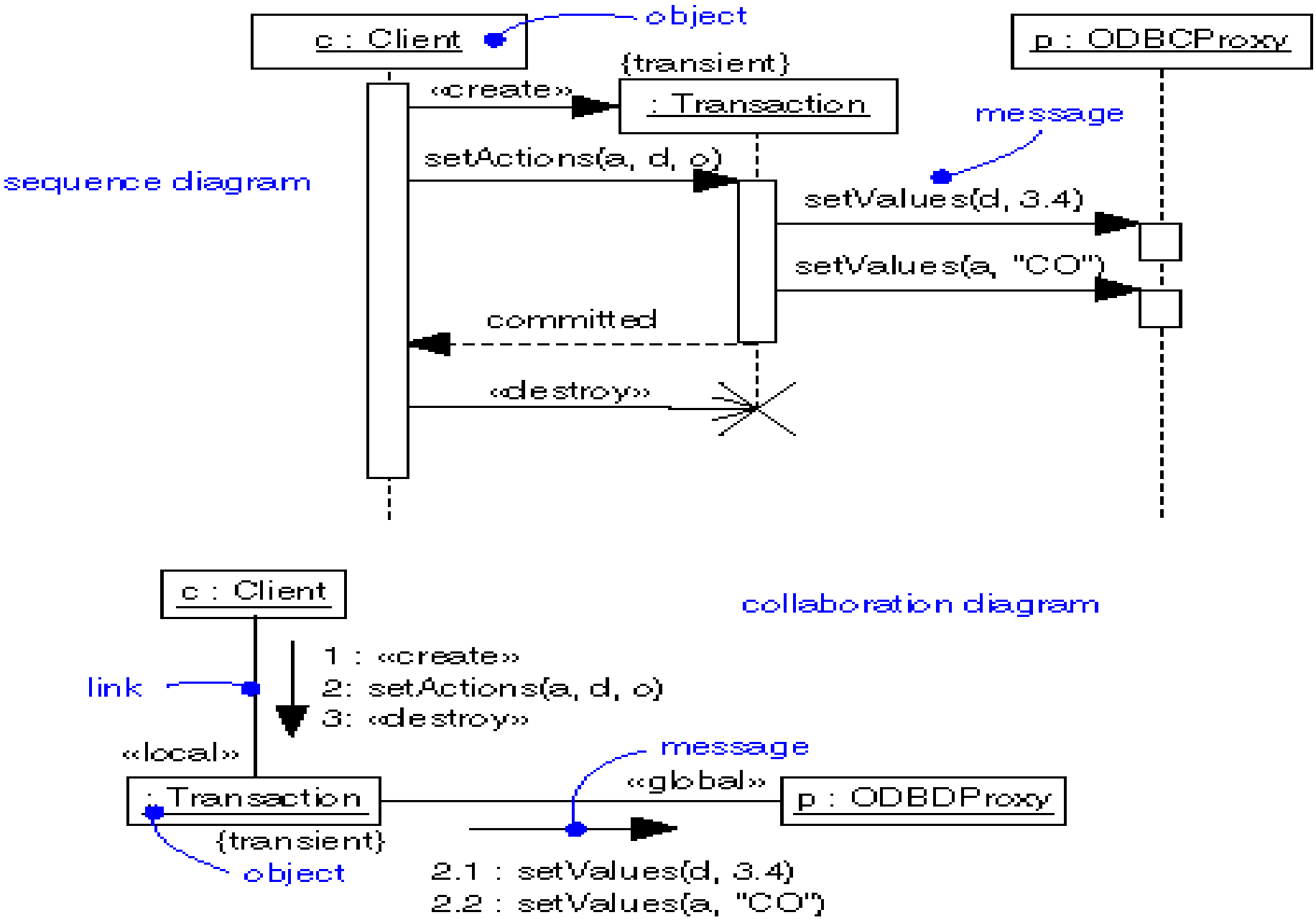
- Collaboration diagrams memberikan suatu gambaran besar skenario karena kolaborasi yang terorganisasi antar obyek satu dengan yang lainnya.
- Collaboration is more usable than sequence in design phase.

Example Sequence Diagram



Example of Collaboration diagram




sequence diagram
c : Client
object
{transient}
«create»
: Transaction
p : ODBCProxy
message
setActions(a, d, c)
setValues(d, 3.4)
setValues(a, "CO")
committed
«destroy»
collaboration diagram
c : Client
link
1 : «create»
2: setActions(a, d, c)
3: «destroy»
«local»
message
«global»
: Transaction
p : ODBDProxy
{transient}
object
2.1 : setValues(d, 3.4)
2.2 : setValues(a, "CO")

Fun Example Objects

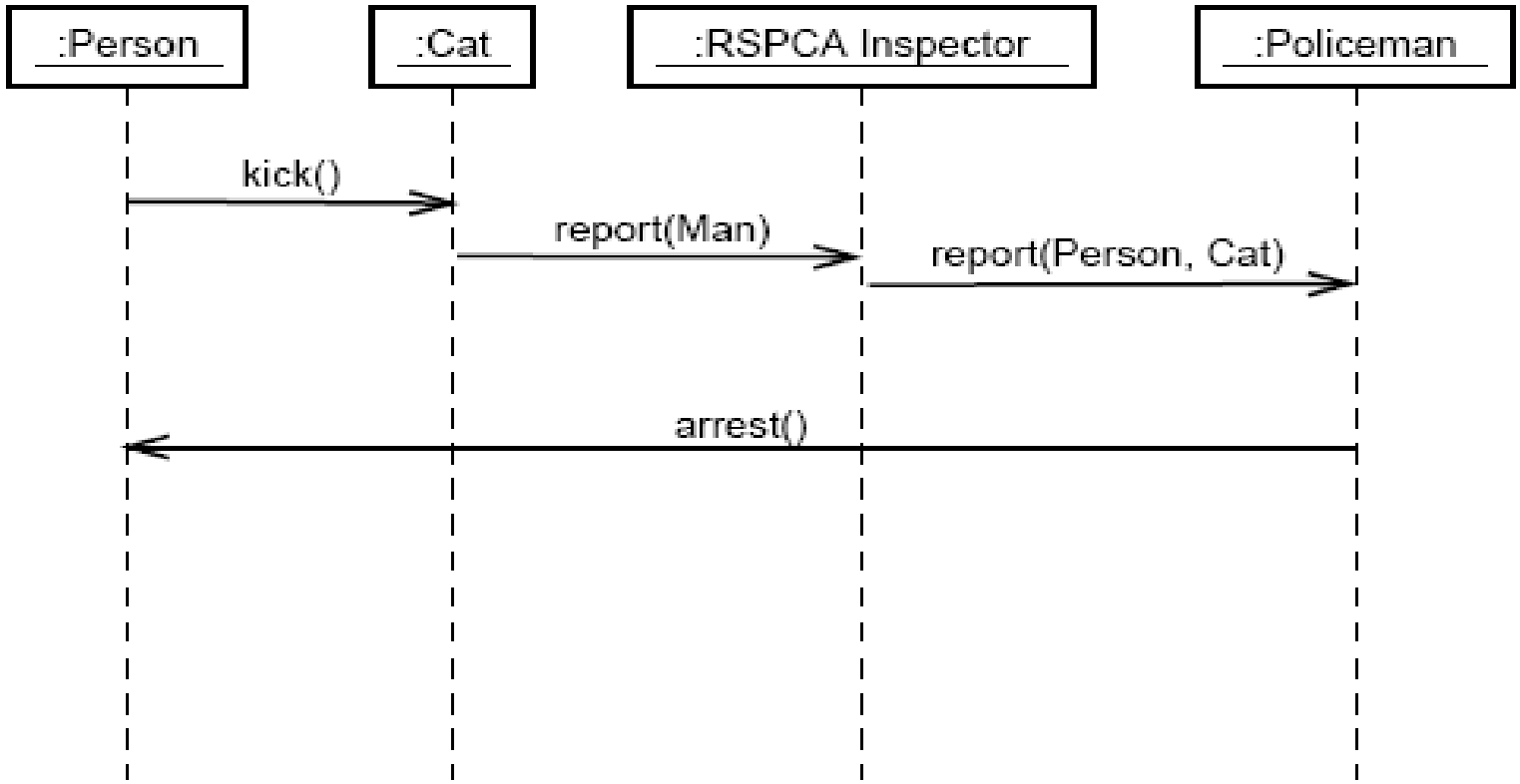
:Cat

:Policeman

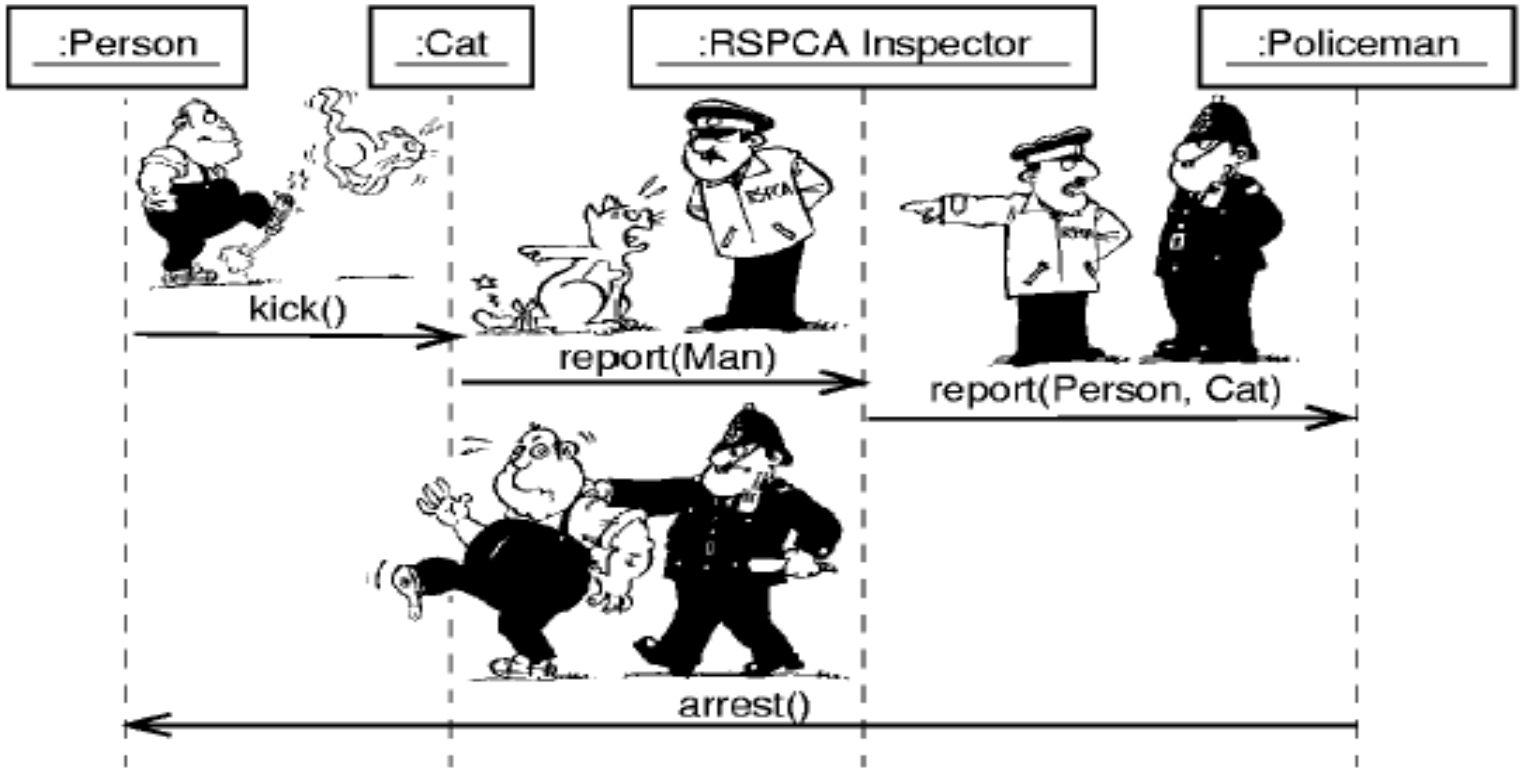
:Person

:RSPCA Inspector

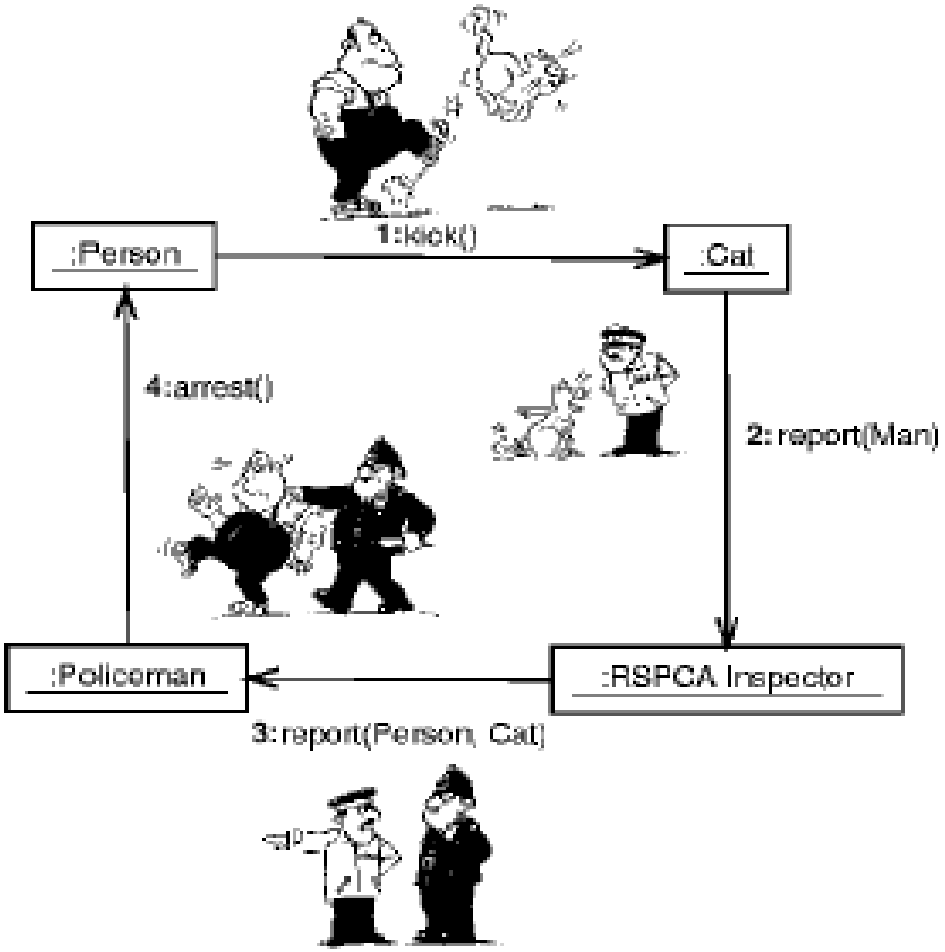
Fun Example Sequence diagram



Fun Example Sequence diagram



Fun Example Collaboration diagram



References

1. Roger S. Presmann, Software Engineering, 6th edition.
2. Kendall, System Analysis and Design, 7th edition.
3. Ian Sommerville, Software Engineering, 8th Edition
4. PPT of Roger S. Pressman (chung and zheng)
5. PPT of Kendall
6. Saiful Akbar, Handouts PPL – ITB, 2011
7. Scott W. Embler, Elements of UML Style 2.0
8. Martin Fowler, UML Distilled 3, Third Edition