

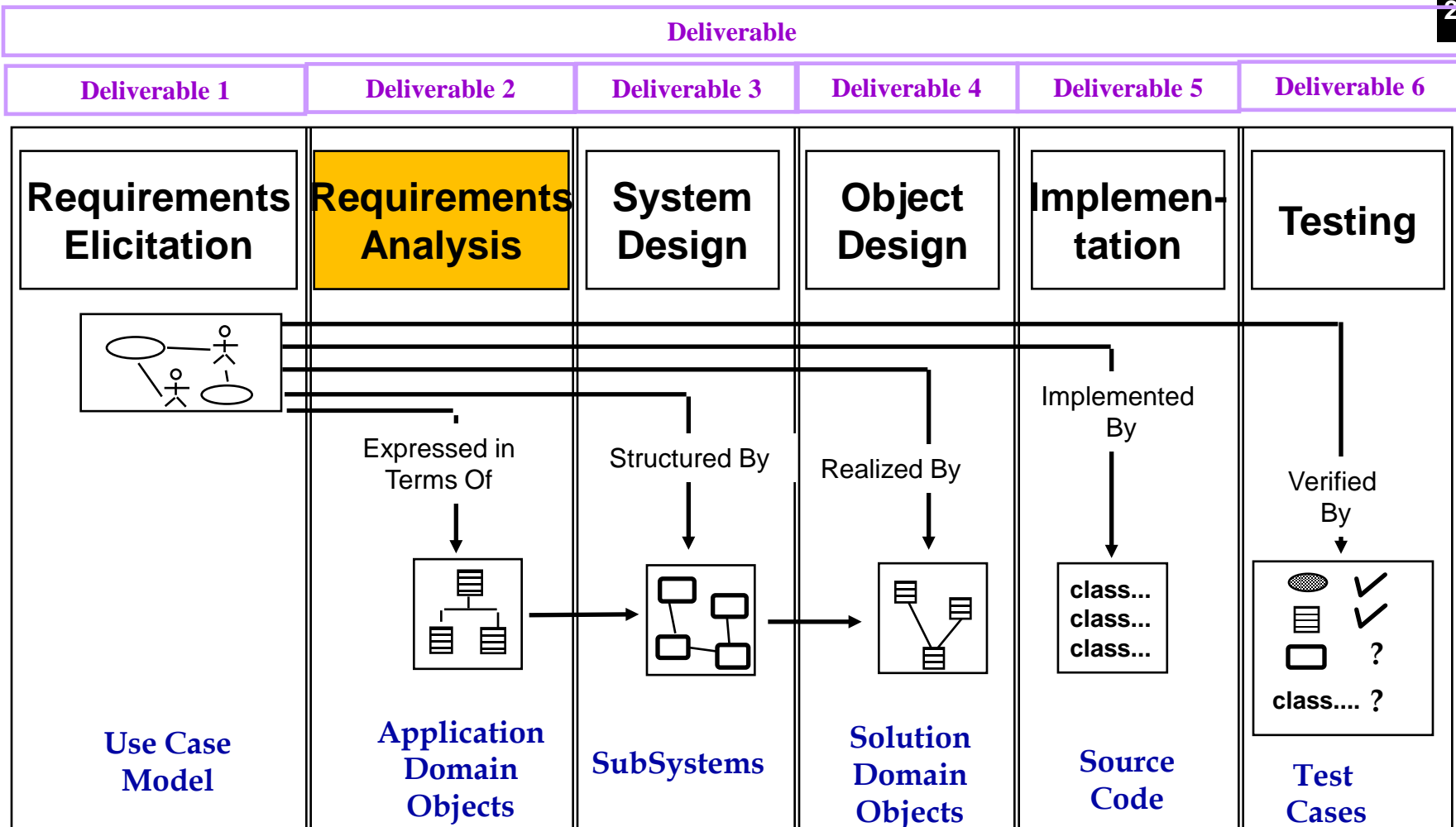


SENG2130 – Week 6 Analysis

Dr. Joe Ryan

SENG2130 – Systems Analysis and Design

University of Newcastle



Outline of the Lecture

- Analysis Object Model
- Dynamic modeling
 - Interaction Diagrams
 - Sequence diagrams
 - Communication diagrams
 - State diagrams
- Requirements analysis model validation
- Analysis Example

An Overview of Analysis

4/69

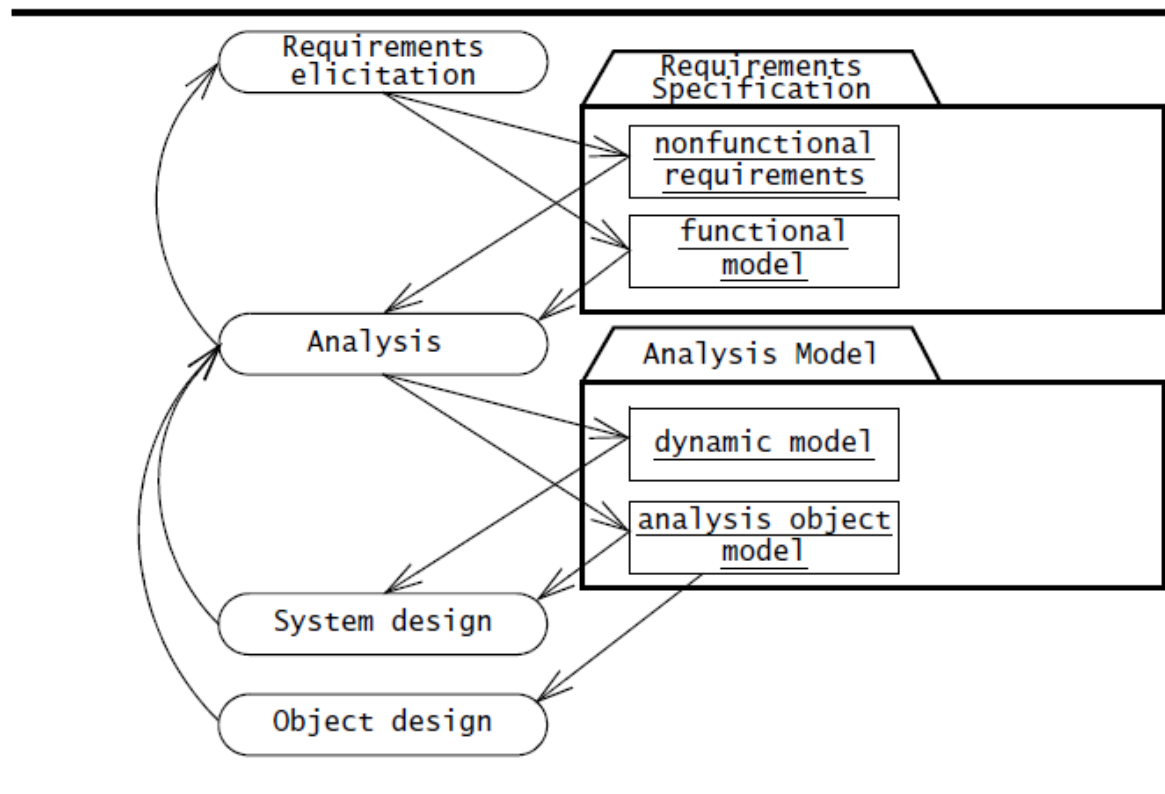


Figure 5-2 Products of requirements elicitation and analysis (UML activity diagram).

Analysis

- Analysis results in a model of the system that aims to be correct, consistent, and unambiguous.
 - Developers **formalize** the requirements specification
 - Developers **validate, correct** and **clarify**
 - The client and the user are usually involved in Analysis activity when the requirements specification must be **changed** and when **additional information** must be gathered

Object-oriented Analysis

- Developers build **a model** describing the application domain.
 - eg., the analysis model of a watch
 - Does the watch know about leap years?
 - Does it know about the day of the week?
 - Does it know about the phases of the moon?
- The analysis model is then extended to describe how the actors and the system **interact** to manipulate the application domain model
 - eg., the analysis model of a watch
 - How does the watch owner reset the time?
 - How does the watch owner reset the day of the week?
- Developers use the analysis model, together with non-functional requirements, to prepare for the architecture of the system developed during high-level design (week 7, System design: decomposing the system)

Analysis Activities

- Identification of objects
 - Objects' behaviour
 - Objects' relationship
 - Objects' classification
 - Objects' organization
- Analysis issues and validation/verification

Analysis Object Model

April 18, 2018

SENG2130 Systems Analysis and Design

Activities during Analysis Object Modeling

Main goal: Find the important abstractions

- Steps during Analysis object modeling



1. Identify objects/classes
2. Identify class attributes
3. Identify class methods
4. Identify relationship between classes
5. Develop class diagram

1. Identify Objects/Classes
2. Identify Class Attributes
3. Identify Class Operations
4. Identify Class Relationships
5. Develop Class Diagrams

1. Identify objects/classes

Class identification is crucial to object-oriented modeling

- Helps to identify the important entities of a system
- Basic assumptions:
 1. We can find the classes for a new software system
(Forward Engineering)
 2. We can identify the classes in an existing system (Reverse Engineering)
- Why can we do this?
 - Philosophy, science, experimental evidence.

1. Identify Objects/Classes
2. Identify Class Attributes
3. Identify Class Operations
4. Identify Class Relationships
5. Develop Class Diagrams

1. Identify objects/classes

- Approaches
 - Application domain approach
 - Ask application domain experts to identify relevant abstractions
 - Syntactic approach
 - Start with use cases
 - Analyze the text to identify the objects
 - Extract participating objects from flow of events
 - Design patterns approach
 - Use reusable design patterns
 - Component-based approach
 - Identify existing solution classes.

Class identification is a Hard Problem

12/69




- One problem: Definition of the system boundary:
 - Which abstractions are outside, which abstractions are inside the system boundary?
 - Actors are outside the system
 - Classes/Objects are inside the system.
- Another problem: Classes/Objects are not just found by taking a picture of a scene or domain
 - The application domain has to be analyzed
 - Depending on the purpose of the system, different objects might be found
 - How can we identify the purpose of a system?
 - Scenarios and use cases => Functional model

Object Types allow us to deal with Change

13/69

- Having three types of object (Boundary, Control, Entity) leads to models that are more resilient to change
 - The interface of a system changes are more likely than the control
 - System control changes are more likely than entities in the application domain
- Object types originated in Smalltalk:
 - Model, View, Controller (MVC)
 - Model <-> Entity Object
 - View <-> Boundary Object
 - Controller <-> Control Object

Finding Participating Objects in Use Cases

- Pick a use case and look at flow of events
- Do a textual analysis (noun-verb analysis)
 - Nouns are candidates for objects/classes
 - Verbs are candidates for operations
 - This is also called **Abbott's Technique**
- After objects/classes are found, identify their types
 - Identify **real world entities** that the system needs to keep track of (FieldOfficer  Entity Object)
 - Identify **real world procedures** that the system needs to keep track of (EmergencyPlan  Control Object)
 - Identify **interface artifacts** (PoliceStation  Boundary Object).

1. Identify objects/classes: example

15/69

- **Examples: Bank Accounts Management System**

This system provides the basic services to manage bank accounts at a bank called OOBank. OOBank has many branches, each of which has an address and branch number. A client opens accounts at a branch. Each account is uniquely identified by an account number; it has a balance and a credit or overdraft limit. There are many types of accounts, including: A mortgage account (which has a property as collateral), a chequing account, and a credit card account (which has an expiry date and can have secondary cards attached to it). It is possible to have a joint account (e.g. for a husband and wife). Each type of account has a particular interest rate, a monthly fee and a specific set of privileges (e.g. ability to write cheques, insurance for purchases etc. OOBank is divided into divisions and subdivisions (such as Planning, Investments and Consumer), the branches are considered subdivisions of the Consumer Division. Each division has a manager and a set of other employees. Each customer is assigned a particular employee as his or her ‘personal banker’.

1. Identify objects/classes: example

16/69

- Example: bank system

This *system* is called *bank* and *branches* uniquely *client* overdraft *account* *card* attached *wife*). E specific *purcha* *Plannin* subdivi set of c his or h

Bank accounts

Bank

Branches

Client

Mortgage account

Property

Cheque account

Credit card

account

Card

Divisions

Manager

Employees

services to manage bank accounts at a bank many branches, each of which has an address opens accounts at a branch. Each account is account number and a credit or debit limit. Types of accounts include: A mortgage account, and a credit card account. A cheque account is attached to a credit card (for a *husband* and *wife*). E specific *purcha* *Plannin* subdivi set of c his or h *Consumer* Division. Each division has a manager and a customer is assigned a particular employee as

Address

Joint account

Privileges

Cheques

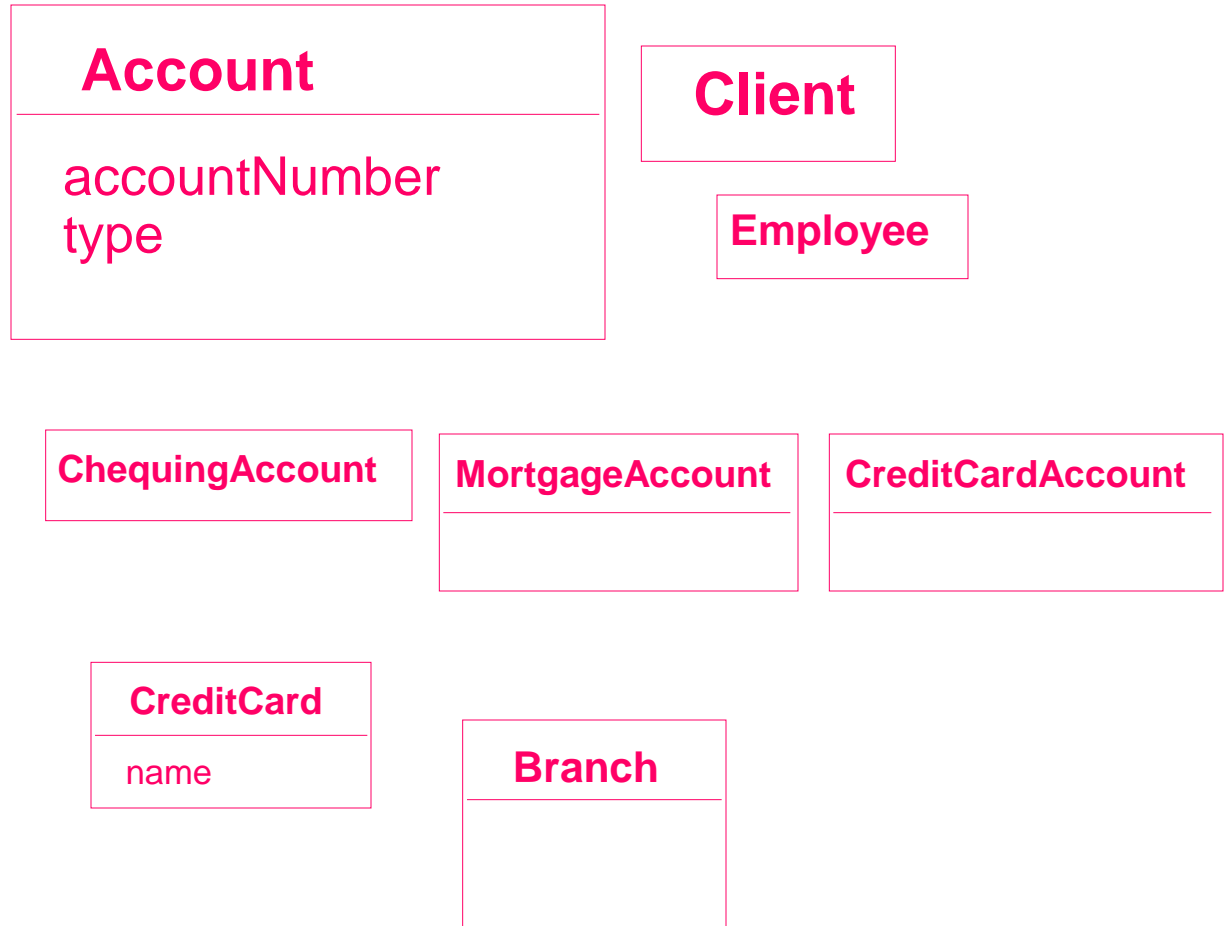
Insurance

Purchases

1. Identify objects/classes: exam

1. Identify Objects/Classes
2. Identify Class Attributes
3. Identify Class Operations
4. Identify Class Relationships
5. Develop Class Diagrams

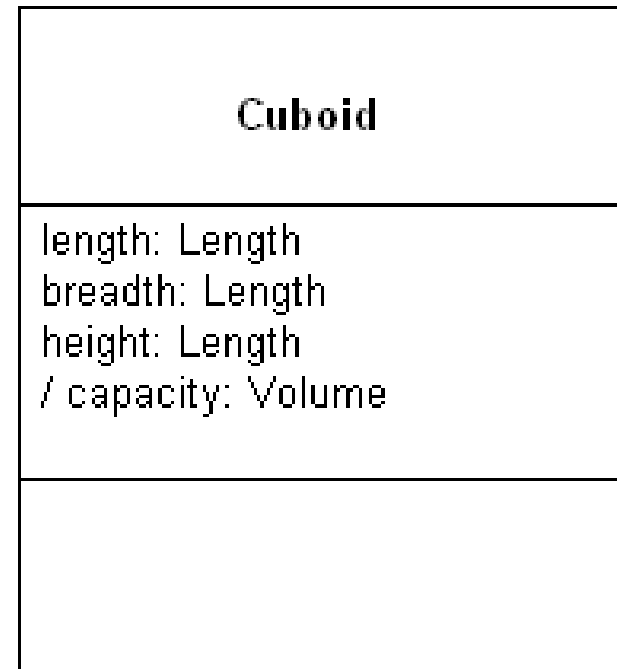
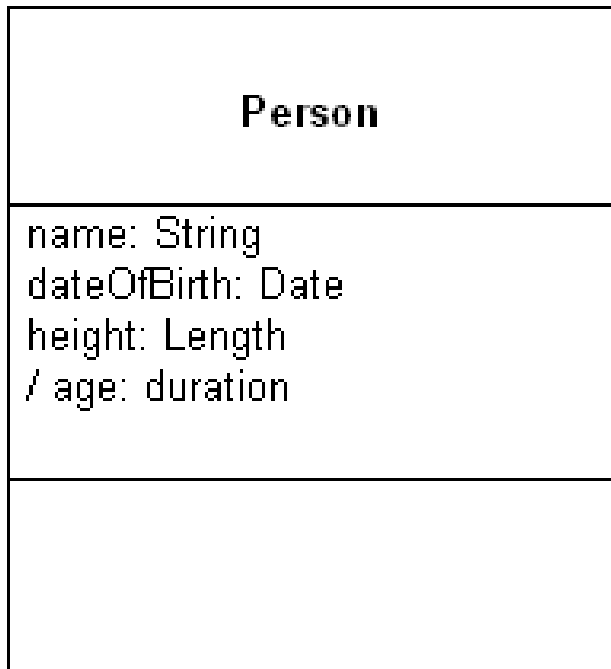
Bank accounts
Bank
Branches
Client
Mortgage account
Property
Cheque account
Credit card
account
Card
Divisions
Manager
Employees



2. Identify Class Attributes

1. Identify Objects/Classes
2. Identify Class Attributes
3. Identify Class Operations
4. Identify Class Relationships
5. Develop Class Diagrams

- An attribute represents information about an object
- An attribute is both gettable and settable from outside the object



2. Identify Class Attributes

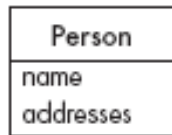
Tips about identifying attributes:

- Look for information that must be **maintained** about each class
- Several nouns rejected as classes, may now become attributes
- An attribute should generally contain a simple value
 - E.g. string, number

2. Identify Class Attributes

Tips about identifying and specifying valid attributes:

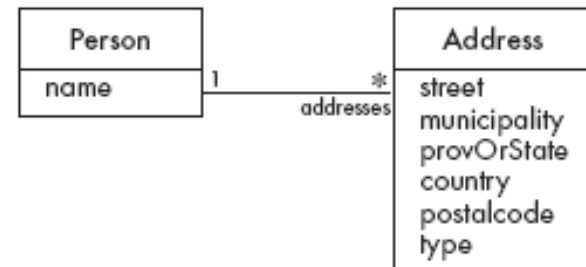
- Avoid duplicate attributes
- If a subset of a class's attributes form a **coherent** group, then create a distinct class containing these attributes



Bad, due to
a plural attribute



Bad, due to too many
attributes, and the
inability to add more
addresses



Good solution. The type indicates whether it
is a home address, business address etc.

3. Identify Class Operations

1. Identify Objects/Classes
2. Identify Class Attributes
3. Identify Class Operations
4. Identify Class Relationships
5. Develop Class Diagrams

- operations appear in the lowest compartment with their full formal signature.
- Each of the signature comprises the operation name, together with the list of method's formal input and output arguments.

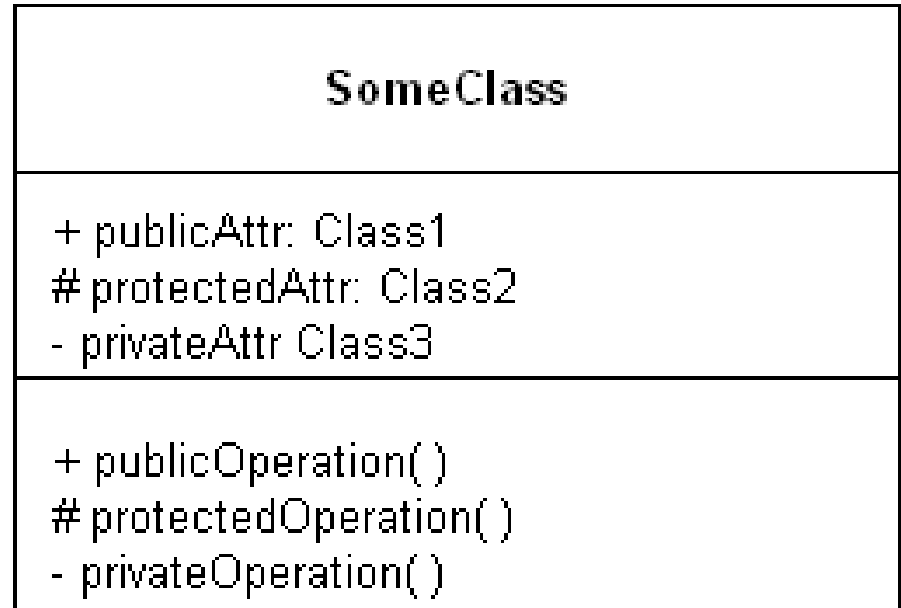
Person
name: String dateOfBirth: Date height: Length / age: duration
getName (out name: String) setName (name: String) getAge(out age:Duration)

Cuboid
length: Length breadth: Length height: Length / capacity: Volume
getLength (length: Length) setLength (length: Length) getCapacity (out capacity: Volume)

Visibility of Attributes and Operations

22/69

- UML attaches a prefix to an attribute or operation name to indicate the feature's visibility.



+ public level - can be accessed by any class.

protected level - open only to classes that inherit from original class

- private level - only the original class can use the attribute or operation

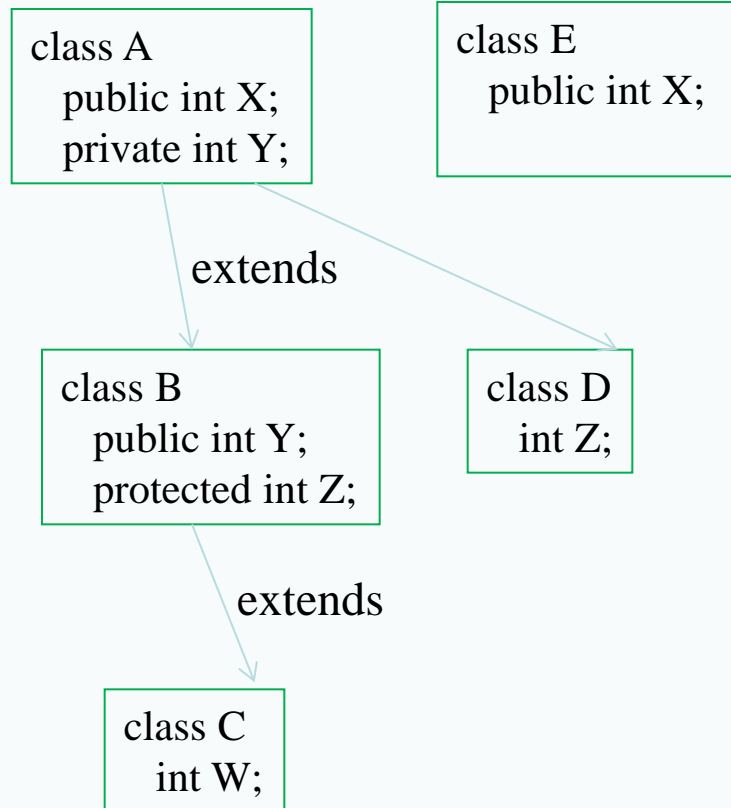
Visibility of Attributes and Operations

23/69

+ **public level** - can be accessed by any class.
protected level - usability is open only to classes that inherit from original class
- **private level** - only the original class can use the attribute or operation

- [redacted] A.X
- [redacted] A.Y
- [redacted] B.Y
- [redacted] B.Z
- [redacted] access C.W and D.Z
- [redacted] E.X

Package



4. Identify Class Relationships

1. Identify Objects/Classes
2. Identify Class Attributes
3. Identify Class Operations
4. Identify Class Relationships
5. Develop Class Diagrams

- Start with classes you think are most **central** and important
- Decide on the clear and obvious data it must contain and its relationships to other classes
- Work outwards towards the classes that are less important

4. Identify Class Relationships

25/69

- **Tips about identifying and specifying valid relationships:**

An relationship should exist if a class

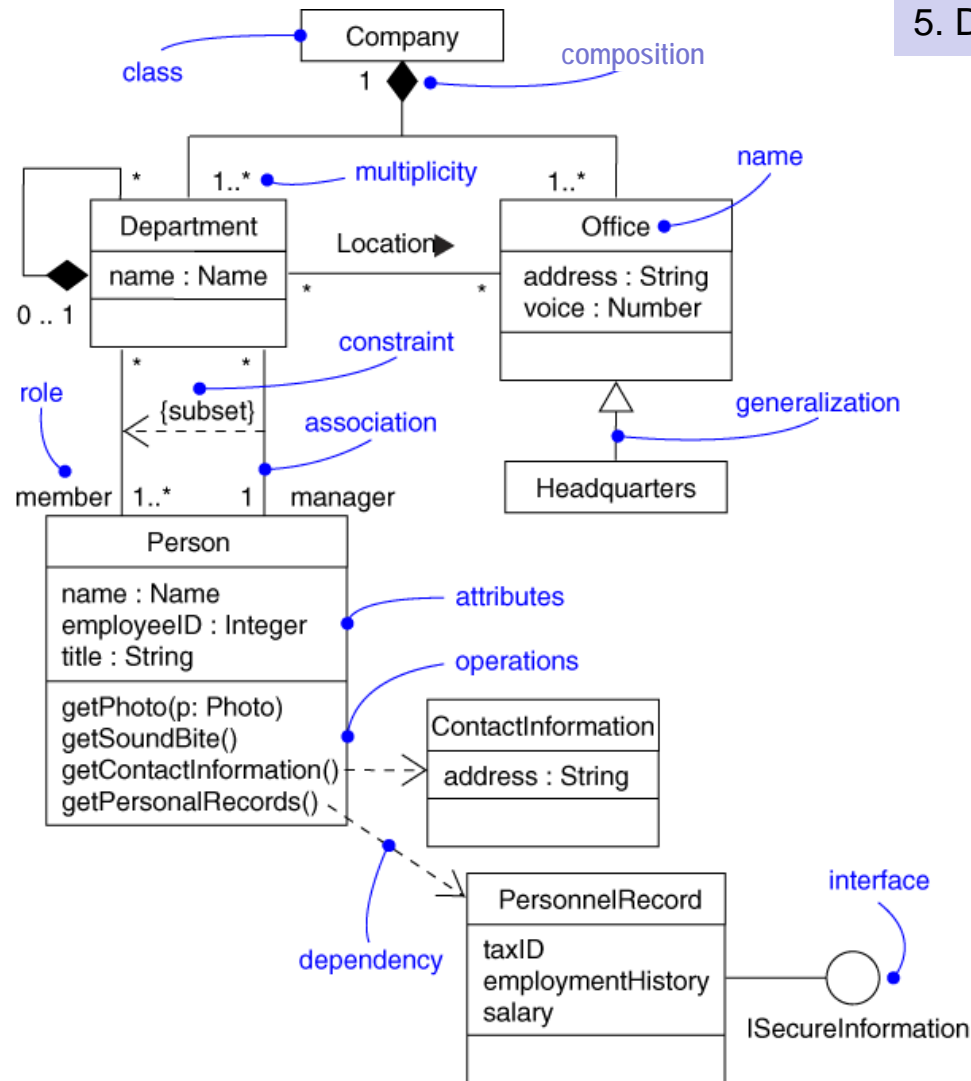
- *possesses*
- *controls*
- *is connected to*
- *is related to*
- *is a part of*
- *is a member of*

Add some class to your model

- Specify the multiplicity at **both ends**
- Label it clearly.

5. Develop Class Diagrams

1. Identify Objects/Classes
2. Identify Class Attributes
3. Identify Class Operations
4. Identify Class Relationships
5. Develop Class Diagrams



Who uses Class Diagrams?

- Purpose of class diagrams
 - The description of the static properties of a system
- The main users of class diagrams:
 - The application domain expert
 - uses class diagrams to model the application domain (including taxonomies)
 - during requirements elicitation and analysis
 - The developer
 - uses class diagrams during the development of a system
 - during analysis, system design, object design and implementation.

Who does not use Class Diagrams?

- The **client** and the **end user** are usually not interested in class diagrams
 - Clients focus more on project management issues
 - End users are more interested in the functionality of the system.

Developers have different Views on Class Diagrams

- According to the development activity, a developer plays different roles:
 - Analyst
 - System Designer
 - Object Designer
 - Implementor
- Each of these roles has a different view about the class diagram (the object model).

The View of the Analyst

- The **analyst** is interested
 - in **application classes**: The **associations between classes are relationships** between abstractions in the application domain
 - operations and attributes of the application classes
- The analyst uses inheritance in the model to reflect the taxonomies in the application domain
 - **Taxonomy**: An is-a-hierarchy of abstractions in an application domain
- The analyst is not interested
 - in the exact signature of operations
 - in solution domain classes

The View of the Designer

- The **designer** focuses on the solution of the problem, that is, the solution domain
- The **associations between classes are now references** (pointers) between classes in the application or solution domain
- An important design task is the specification of interfaces:
 - The designer describes the interface of classes and the interface of subsystems
 - Subsystems originate from modules (term often used during analysis):
 - **Module**: a collection of classes
 - **Subsystem**: a collection of classes with an interface
- Subsystems are modeled in UML with a package.

Goals of the Designer

- The most important design goals for the designer are design usability and design reusability
- **Design usability:** the interfaces are usable from as many classes as possible within in the system
- **Design reusability:** The interfaces are designed in a way, that they can also be reused by other (future) software systems
 - => Class libraries
 - => Frameworks
 - => Design patterns.

The View of the Implementor

- **Class implementor**
 - Must realize the interface of a class in a programming language
 - Interested in appropriate data structures (for the attributes) and algorithms (for the operations)
- **Class extender**
 - Interested in how to extend a class to solve a new problem or to adapt to a change in the application domain
- **Class user**
 - The class user is interested in the signatures of the class operations and conditions under which they can be invoked
 - The class user is not interested in the implementation of the class.

Why do we distinguish different Users of Class Diagrams?

- Models often don't distinguish between application classes and solution classes
 - **Reason:** Modeling languages like UML allow the use of both types of classes in the same model
 - “address book”, “array”
 - **Preferred:** No solution classes in the analysis model
- Many systems don't distinguish between the specification and the implementation of a class
 - **Reason:** Object-oriented programming languages allow the simultaneous use of specification and implementation of a class
 - **Preferred:** We distinguish between analysis model and object design model. The analysis design model does not contain any implementation specification.

Analysis Model vs Object Design Model

- The analysis model is constructed during the analysis phase
 - Main stake holders: End user, customer, analyst
 - The class diagrams contains only application domain classes
- The object design model (sometimes also called specification model) is created during the object design phase
 - Main stake holders: class specifiers, class implementers, class users and class extenders
 - The class diagrams contain application domain as well as solution domain classes.

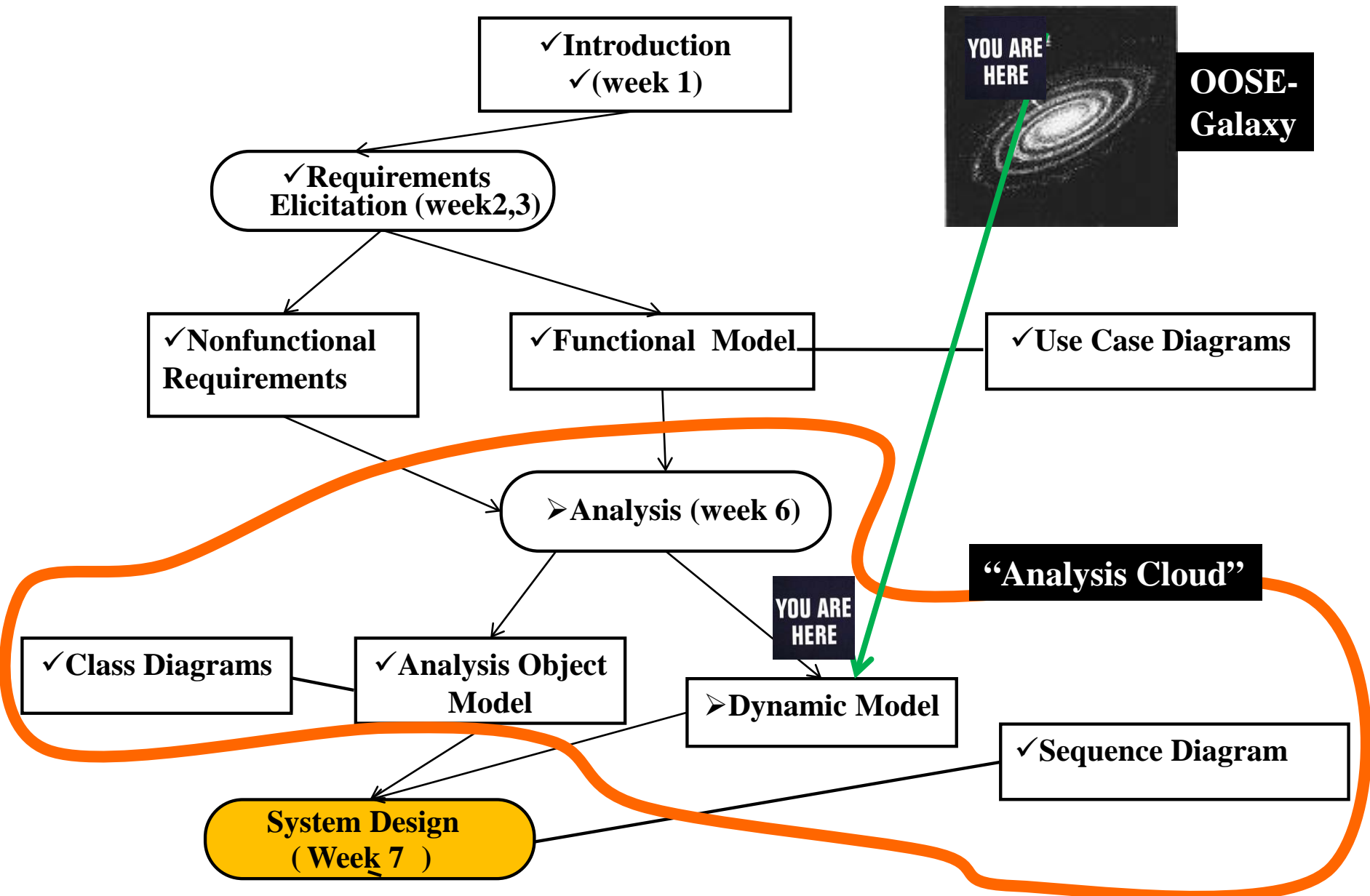
Analysis Model vs Object Design Model

- The analysis model is the basis for communication between analysts, application domain experts and end users.
- The object design model is the basis for communication between designers and implementers.

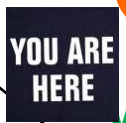
Dynamic Modeling

April 18, 2018

SENG2130 Systems Analysis and Design



"Analysis Cloud"



Dynamic Modeling with UML

- Two UML diagrams types for describing dynamic models:
 - Statechart diagrams describe the dynamic behavior of a *single object*
 - Interaction diagrams describe the dynamic behavior *between objects*.

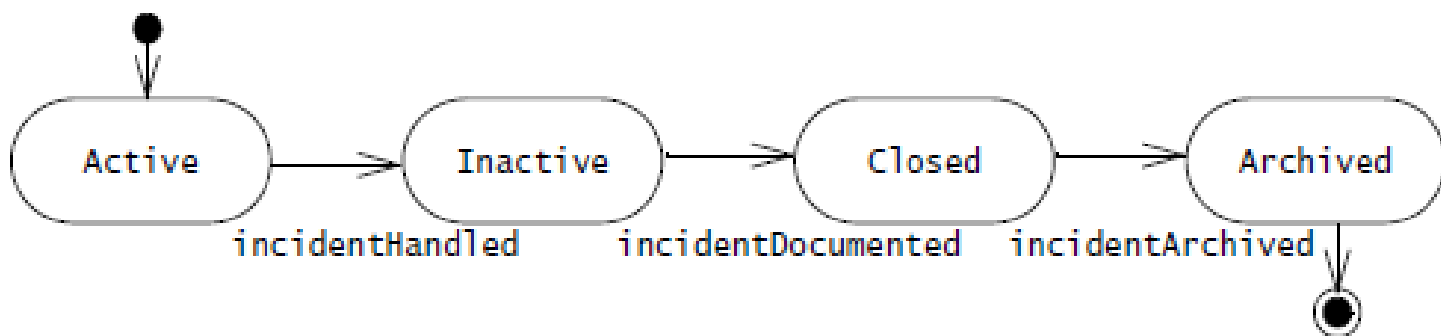
UML State Chart Diagram

- State Chart Diagram

- A *notation* for a state machine that describes the response of an object of a given class to the receipt of outside stimuli (Events)

- State Machine

- A *model* of behavior composed of a finite number of states, transitions between those states, and actions



STATE MACHINE DIAGRAM FOR INCIDENT CLASS

UML Interaction Diagrams

- Two types of interaction diagrams:
 - Communication Diagram:
 - Shows **the temporal relationship** among objects
 - Good for identifying the protocol between objects
 - **Does not show time**
 - Sequence Diagram:
 - Describes the dynamic behavior *between* several objects over time
 - Good for real-time specifications.

Dynamic Modeling

- Definition of a dynamic model:
 - Describes the components of the system that have interesting dynamic behavior
- The dynamic model is described with
 - Sequence diagrams: For the interaction between classes
- Purpose:
 - Identify new classes in the object model and supply operations for the classes.

Identify Classes and Operations

- We have already established several sources for class identification:
 - **Application domain analysis**: We find classes by talking to the client and identify abstractions by observing the end user
 - **General world knowledge and intuition**
 - **Textual analysis** of event flow in use cases (Abbot)
- One additional heuristic for identifying classes from dynamic model:
 - Activity lines in sequence diagrams are candidates for objects.

How do we detect Operations?

- We look for objects, who are interacting and extract their “protocol”
- We look for objects, who have interesting behavior on their own
- Good starting point: Flow of events in a use case description
- From the flow of **events** we proceed to the sequence diagram to find the **participating objects**.

What is an Event?

- Something that happens at a point in time
- An event sends information from one object to another
- Events can have associations with each other:
 - Causally related:
 - An event happens always before another event
 - An event happens always after another event
 - Causally unrelated:
 - Events that happen concurrently
- Events can also be grouped in event classes with a hierarchical structure => Event taxonomy.

The term 'Event' is often used in two ways

- Instance of an event class:
 - “Slide 14 shown on Wednesday April 18 at 15:00”
 - Event class “Lecture Given”, Subclass “Slide Shown”
- Attribute of an event class
 - Slide Update(8:55 AM, 04/18/2018)
 - Train_Leaves(4:45pm, Manhattan)
 - Mouse button down(button#, tablet-location).

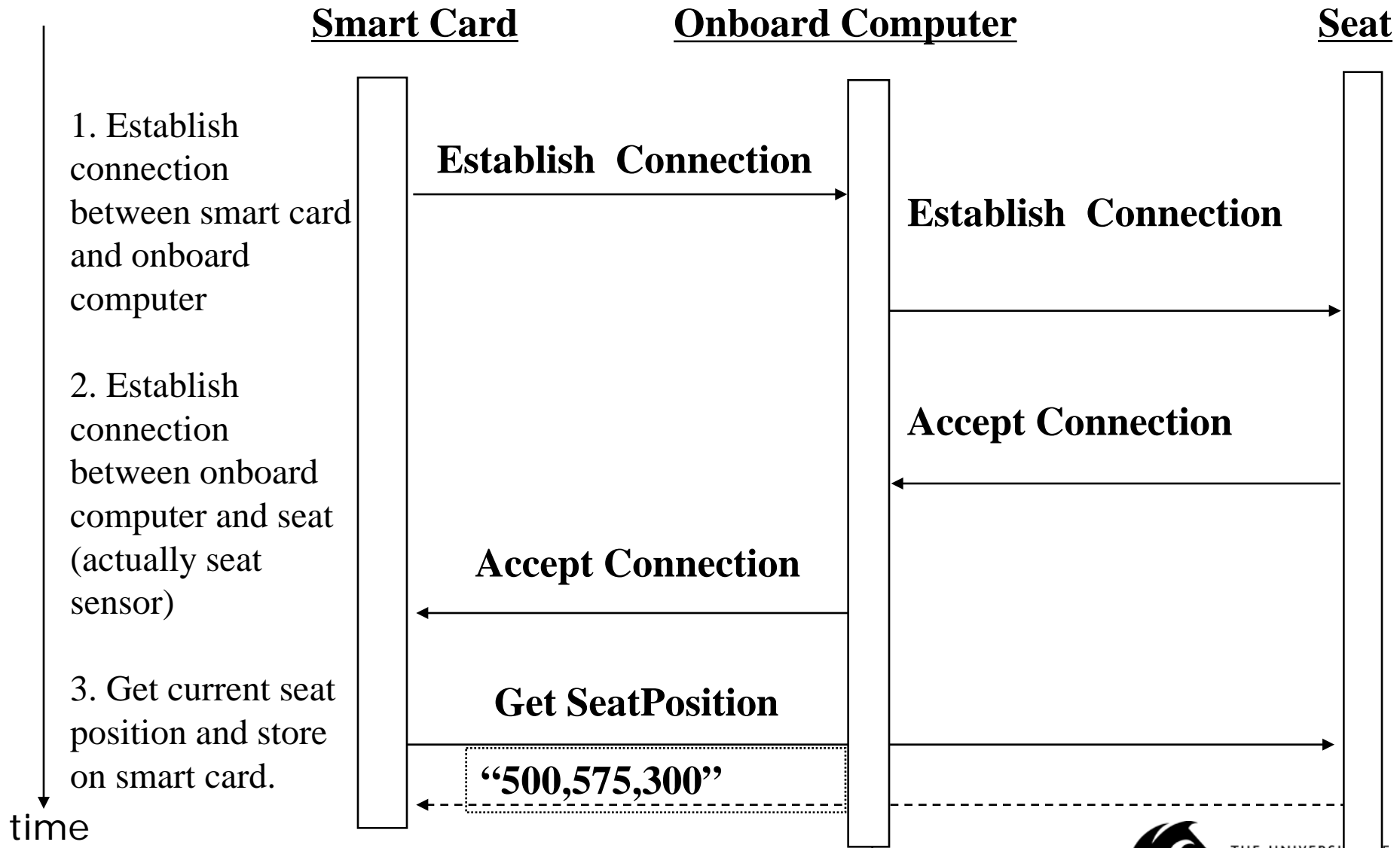
Finding Participating Objects

- Heuristic for finding participating objects:
 - A event always has a sender and a receiver
 - Find the sender and receiver for each event => These are the objects participating in the use case.

An Example

- Flow of events in “Get SeatPosition” use case :
 1. Establish connection between smart card and onboard computer
 2. Establish connection between onboard computer and sensor for seat
 3. Get current seat position and store on smart card
- Where are the objects?

Sequence Diagram for “Get SeatPosition”



Heuristics for Sequence Diagrams

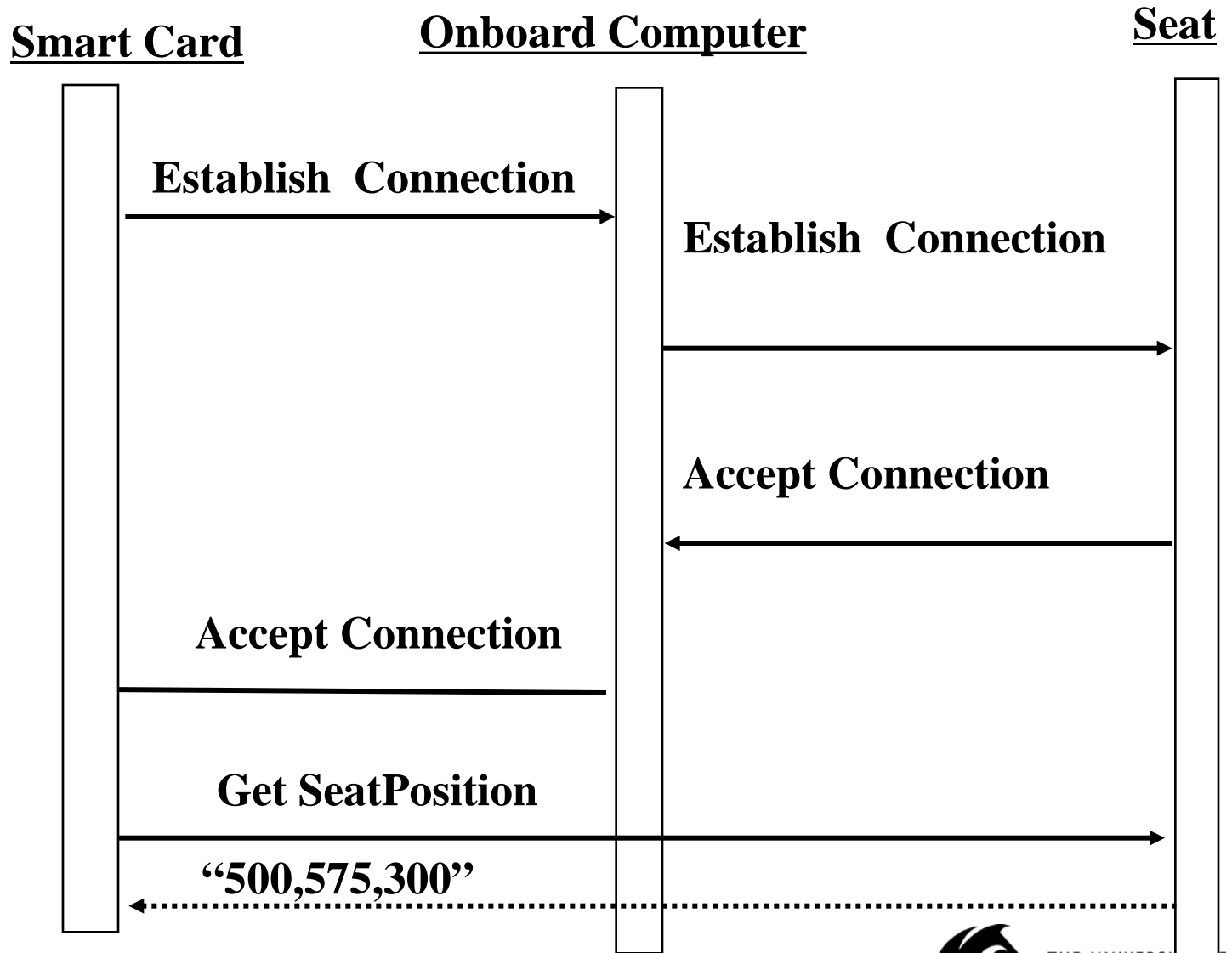
- **Layout:**
 - 1st column: Should be the **actor** of the use case
 - 2nd column: Should be a **boundary object**
 - 3rd column: Should be the **control object** that manages the rest of the use case
- **Creation of objects:**
 - Create control objects at beginning of event flow
 - The control objects create the boundary objects
- **Access of objects:**
 - Entity objects can be accessed by control and boundary objects
 - Entity objects should not access boundary or control objects.

Is this a good Sequence Diagram?

The first column is **not an actor**

It is **not clear** where the **boundary object** is

It is **not clear** where the **control object** is

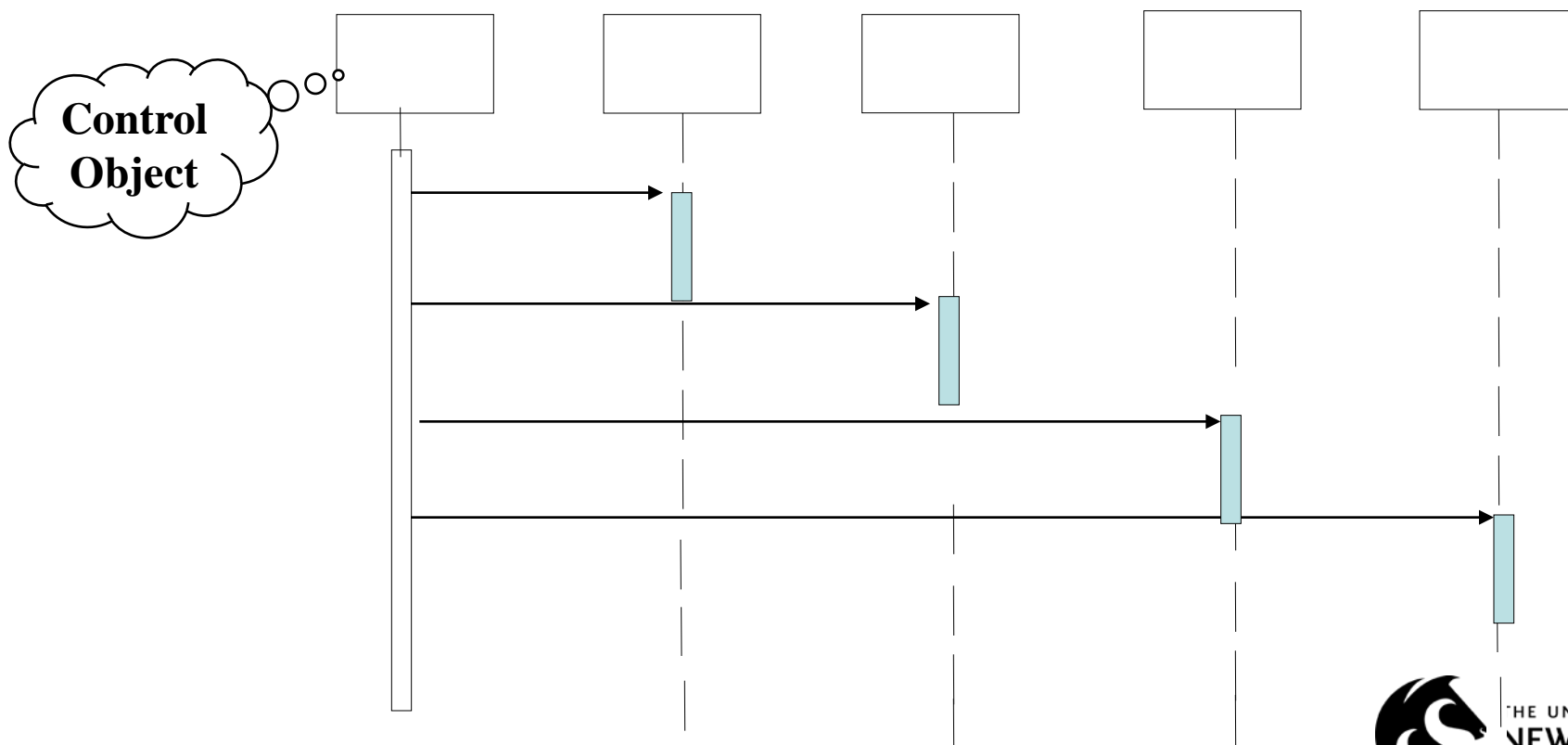


What else can we get out of Sequence Diagrams?

- Sequence diagrams are derived from use cases
- The structure of the sequence diagram helps us to determine **how decentralized the system is**
- We distinguish two structures for sequence diagrams
 - **Fork Diagrams** and **Stair Diagrams** (Ivar Jacobsen)

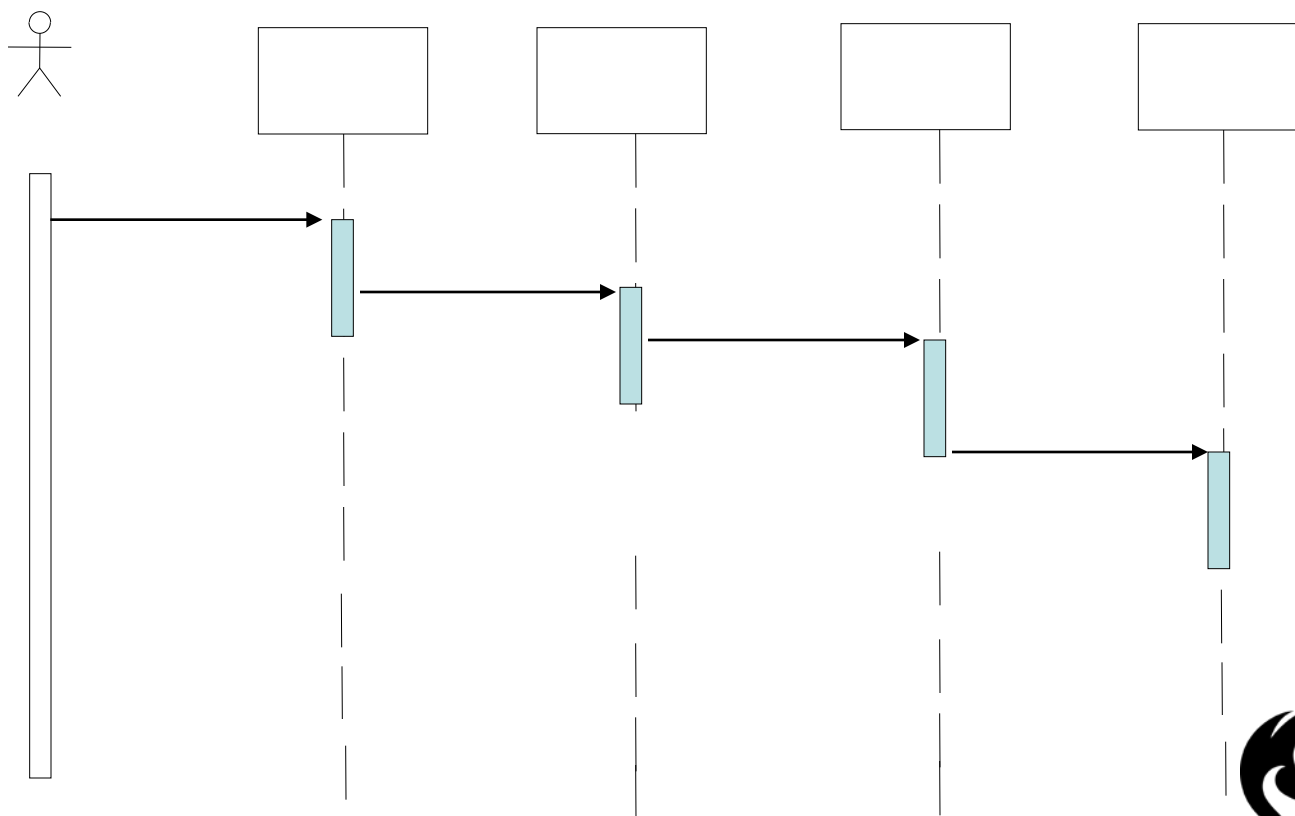
Fork Diagram

- The dynamic behavior is placed in a single object, usually a control object
 - It knows all the other objects and often uses them for direct questions and commands



Stair Diagram

- The dynamic behavior is distributed. Each object delegates responsibility to other objects
 - Each object knows only a few of the other objects and knows which objects can help with a specific behavior



Fork or Stair?

55/69

- Object-oriented supporters claim that the **stair structure is better**

Practical Tips for Dynamic Modeling

56/69

- Construct dynamic models only for classes with significant dynamic behavior
 - Avoid “analysis paralysis”
- Consider only relevant attributes
 - Use abstraction if necessary
- Look at the granularity of the application when deciding on actions and activities
- Reduce notational clutter
 - Try to put actions into superstate boxes (look for identical actions on events leading to the same state).

Model Validation and Verification

- **Verification** is an equivalence check between the transformation of two models
- **Validation** is the comparison of the model with reality
 - Validation is a critical step in the development process
Requirements should be validated with the client and the user.
 - Techniques: Formal and informal reviews (Meetings, requirements review)
- **Requirements validation** involves several checks
 - Correctness, Completeness, Ambiguity, Realism

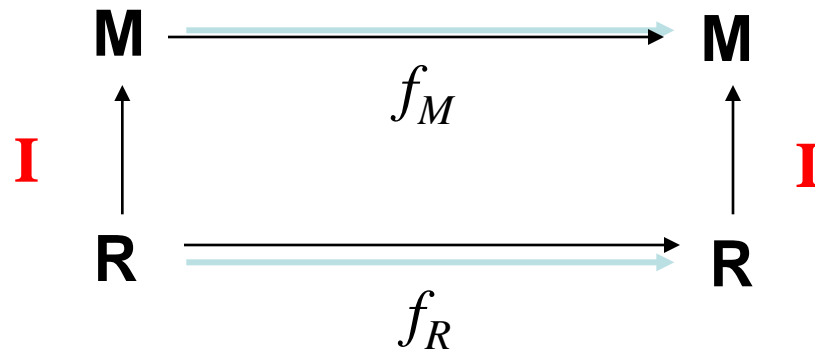
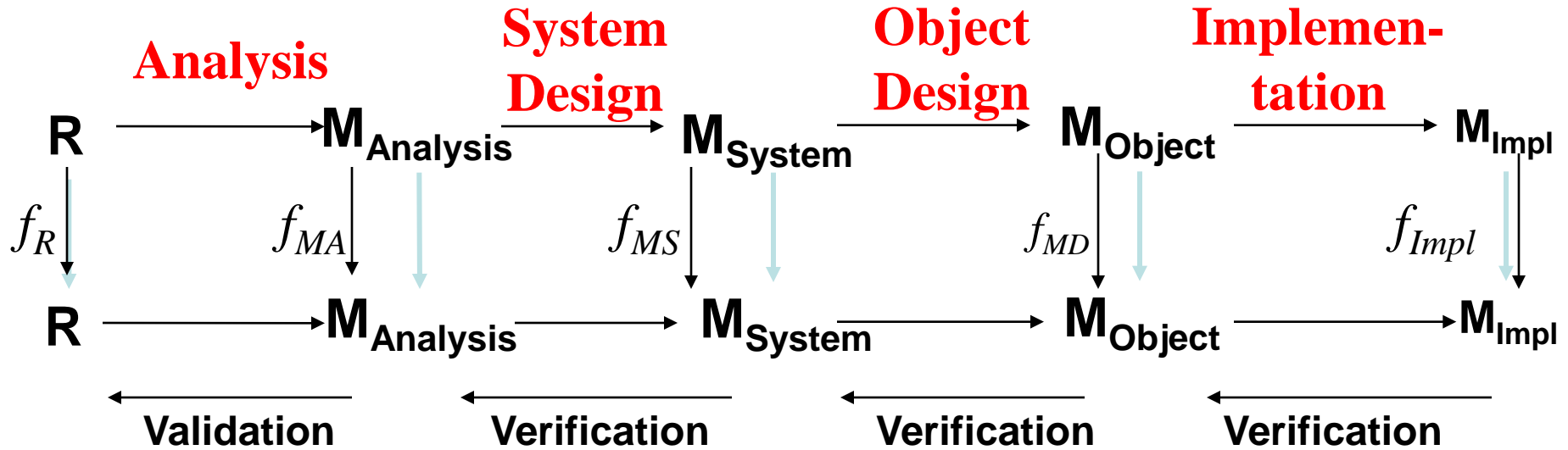
Checklist for a Requirements Review - 1

58/69

- Is the model **correct**?
 - A model is correct if it represents the client's view of the system
- Is the model **complete**?
 - Every scenario is described
- Is the model **consistent**?
 - The model does not have components that contradict each other
- Is the model **unambiguous**?
 - The model describes one system, not many
- Is the model **realistic**?
 - The model can be implemented

Verification vs Validation of models

59/69



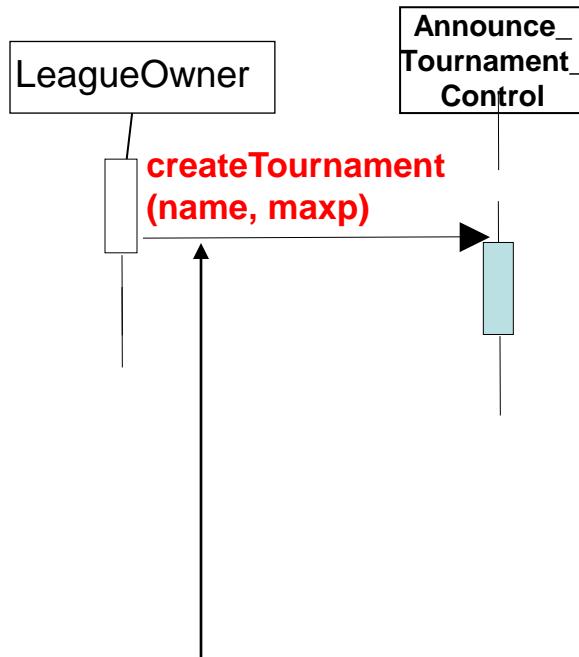
Examples for Inconsistency and Completeness Problems

- Different spellings in different UML diagrams
- Omissions in diagrams

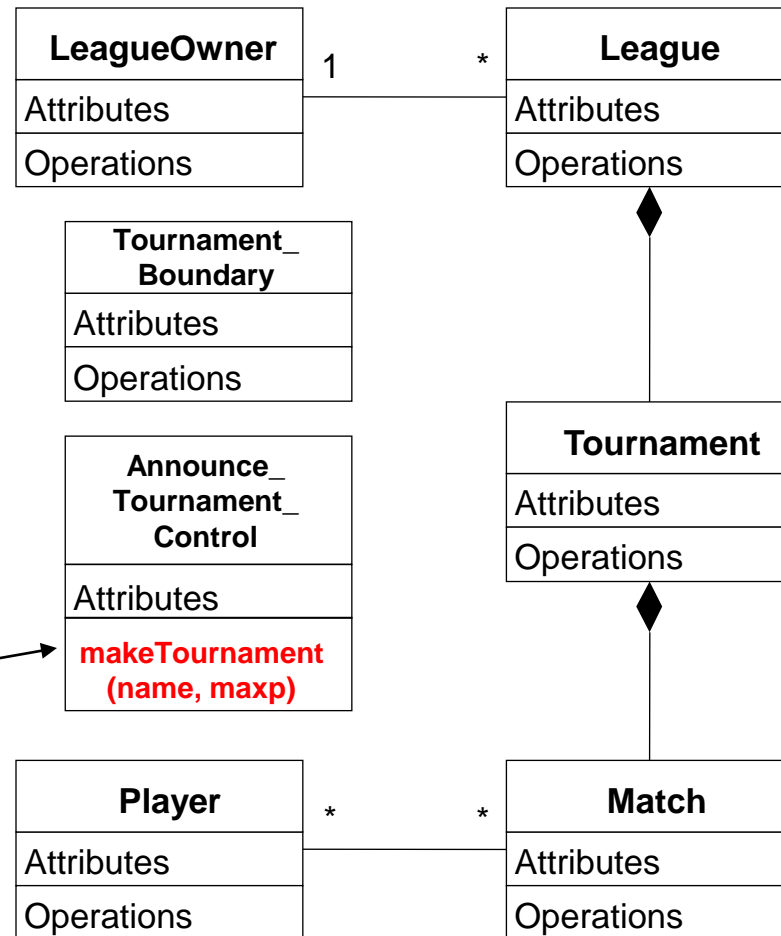
Different spellings in different UML diagrams

61/69

UML Sequence Diagram



UML Class Diagram

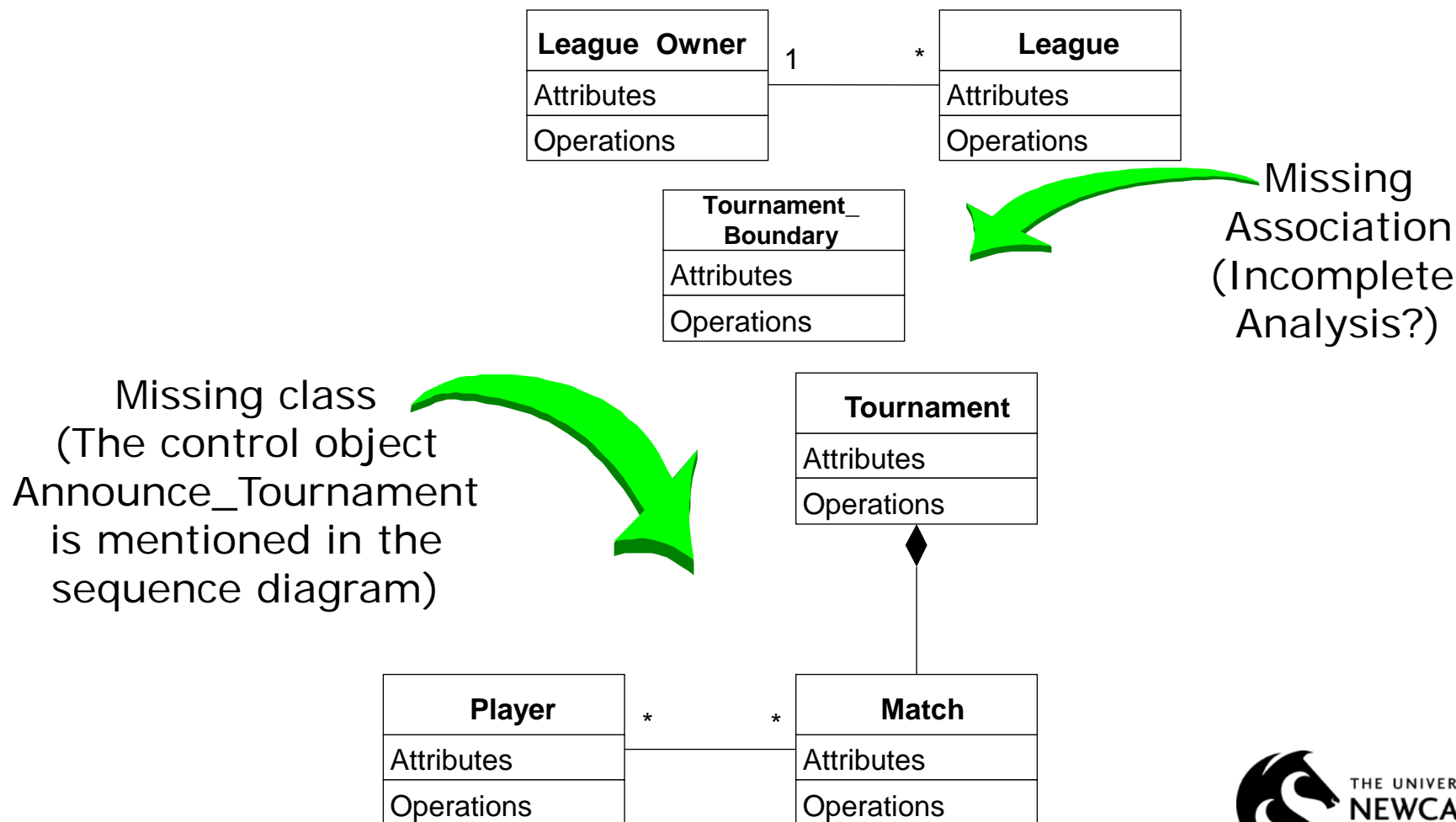


Different spellings
in different models
for the same operation



Omissions in some UML Diagrams

Class Diagram



Checklist for a Requirements Review - 2

63/69

- Syntactical check of the models
- Check for consistent naming of classes, attributes, methods in different subsystems
- Identify dangling associations (“pointing to nowhere”)
- Identify double- defined classes
- Identify missing classes (mentioned in one model but not defined anywhere)
- Check for classes with the same name but different meanings

When is a Model Dominant?

- **Object model:**
 - The system has classes with nontrivial states and many relationships between the classes
- **Dynamic model:**
 - The model has many different types of events: Input, output, exceptions, errors, etc.
- **Functional model:**
 - The model performs complicated transformations (eg. computations consisting of many steps)
- Which model is dominant in these applications?
 - Compiler
 - Database system
 - Spreadsheet program

Examples of Dominant Models

- Compiler:
 - The functional model is most important
 - The dynamic model is trivial because there is only one type input and only a few outputs
- Database systems:
 - The object model most important
 - The functional model is trivial, because the purpose of the functions is to store, organize and retrieve data
- Spreadsheet program:
 - The functional model most important
 - The dynamic model is interesting if the program allows computations on a cell
 - The object model is trivial.

Requirements Analysis Questions

66/69

1. What are the transformations?

 **Functional Modeling**

Create *scenarios and use case diagrams*

- Talk to client, observe, get historical records

2. What is the structure of the system?

 **Object Modeling**

Create *class diagrams*

- Identify objects.
- What are the associations between them?
- What is their multiplicity?
- What are the attributes of the objects?
- What operations are defined on the objects?

3. What is its behavior?

 **Dynamic Modeling**

Create *sequence diagrams*

- Identify senders and receivers
- Show sequence of events exchanged between objects.
- Identify event dependencies and event concurrency.

Summary

- Object modeling is the central activity
 - Class identification is a major activity of object modeling
 - Easy syntactic rules to find classes and objects
 - Abbot's Technique (noun-verb analysis)
- Analysts, designers and implementors have different modeling needs
- There are three types of implementors with different roles during
 - Class user, class implementor, class extender.

Summary

- Dynamic Modeling – Sequence diagram
 - Describes the components of the system that have interesting dynamic behaviour
 - 1st column: should be the actor of the use case
 - 2nd column: should be a boundary object
 - 3rd column: should be the control object that manages the rest of the use case
 - Entity objects are accessed by control and boundary objects
- Object-oriented supporters claim that the stair structure is better

Next Week

69/69

