UNIT-II ANALYSIS AND OBJECT MODELLING
What is this?

Location: Hochschule für Musik und Theater, Arcisstraße 12

Question: How do you mow the lawn?

Lesson: Find the functionality first, then the objects
Where are we right now?

- Three ways to deal with complexity:
  - Abstraction
  - Decomposition (Technique: Divide and conquer)
  - Hierarchy (Technique: Layering)
- Two ways to deal with decomposition:
  - Object-orientation and functional decomposition
  - Functional decomposition leads to unmaintainable code
  - Depending on the purpose of the system, different objects can be found
- What is the right way?
  - Start with a description of the functionality (Use case model). Then proceed by finding objects (object model).
- What activities and models are needed?
  - This leads us to the software lifecycle we use in this class
  - [Association and classification are sub-divisions of static modeling.]
  - [Association relates to problem analysis, classification to solution design.]
Software Lifecycle Definition

♦ Software lifecycle:
  • Set of activities and their relationships to each other to support the development of a software system

♦ Typical Lifecycle questions:
  • Which activities should I select for the software project?
  • What are the dependencies between activities?
  • How should I schedule the activities?
  • What is the result of an activity
Example: Selection of Software Lifecycle Activities for a specific project

The Hacker knows only one activity

Activities used this lecture

Requirements Elicitation  Analysis  System Design  Object Design  Implementation  Testing

Each activity produces one or more models
Software Lifecycle Activities

<table>
<thead>
<tr>
<th>Requirements Elicitation</th>
<th>Requirements Analysis</th>
<th>System Design</th>
<th>Object Design</th>
<th>Implementation</th>
<th>Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>Use Case Model</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Application Domain Objects</td>
<td>SubSystems</td>
<td>Solution Domain Objects</td>
<td>Source Code</td>
<td>Test Cases</td>
</tr>
<tr>
<td></td>
<td>Expressed in Terms Of</td>
<td>Structured By</td>
<td>Realized By</td>
<td>Implemented By</td>
<td>Verified By</td>
</tr>
<tr>
<td></td>
<td>class...</td>
<td>class...</td>
<td>class...</td>
<td>class....?</td>
<td></td>
</tr>
</tbody>
</table>
First Step in Establishing the Requirements:
System Identification

♦ The development of a system is not just done by taking a snapshot of a scene (domain)
♦ Two questions need to be answered:
  ❖ How can we identify the purpose of a system?
  ❖ Crucial is the definition of the system boundary: What is inside, what is outside the system?
♦ These two questions are answered in the requirements process
♦ The requirements process consists of two activities:
  ❖ Requirements Elicitation:
    ❖ Definition of the system in terms understood by the customer ("Problem Description")
  ❖ Requirements Analysis:
    ❖ Technical specification of the system in terms understood by the developer ("Problem Specification")
Defining the System Boundary is Often Difficult

What do you see here?
Products of Requirements Process

- Requirements Elicitation
- Requirements Analysis
- Problem Statement Generation

- Problem Statement
- Model
- System Specification: Model
- Analysis Model: Model
Requirements Elicitation

♦ Very challenging activity
♦ Requires collaboration of people with different backgrounds
  ♦ Users with application domain knowledge
  ♦ Developer with solution domain knowledge (design knowledge, implementation knowledge)
♦ Bridging the gap between user and developer:
  ♦ Scenarios: Example of the use of the system in terms of a series of interactions with between the user and the system
  ♦ Use cases: Abstraction that describes a class of scenarios
System Specification vs Analysis Model

- Both models focus on the requirements from the user’s view of the system.
- **System specification** uses natural language (derived from the problem statement)
- The **analysis model** uses formal or semi-formal notation (for example, a graphical language like UML)
- The starting point is the problem statement
Problem Statement

♦ The problem statement is developed by the client as a description of the problem addressed by the system

♦ Other words for problem statement:
  ♦ Statement of Work

♦ A good problem statement describes
  ♦ The current situation
  ♦ The functionality the new system should support
  ♦ The environment in which the system will be deployed
  ♦ Deliverables expected by the client
  ♦ Delivery dates
  ♦ A set of acceptance criteria
Ingredients of a Problem Statement

- Current situation: The Problem to be solved
- Description of one or more scenarios
- Requirements
  - Functional and Nonfunctional requirements
  - Constraints (“pseudo requirements”)
- Project Schedule
  - Major milestones that involve interaction with the client including deadline for delivery of the system
- Target environment
  - The environment in which the delivered system has to perform a specified set of system tests
- Client Acceptance Criteria
  - Criteria for the system tests
Current Situation: The Problem To Be Solved

- There is a problem in the current situation
  - Examples:
    - The response time when playing letter-chess is far too slow.
    - I want to play Go, but cannot find players on my level.
- What has changed? Why can address the problem now?
  - There has been a change, either in the application domain or in the solution domain
    - Change in the application domain
      - A new function (business process) is introduced into the business
      - Example: We can play highly interactive games with remote people
    - Change in the solution domain
      - A new solution (technology enabler) has appeared
      - Example: The internet allows the creation of virtual communities.
**ARENA: The Problem**

- The Internet has enabled virtual communities
  - Groups of people sharing common interests but who have never met each other in person. Such virtual communities can be short lived (e.g., people in a chat room or playing a multi-player game) or long lived (e.g., subscribers to a mailing list).

- Many multi-player computer games now include support for virtual communities.
  - Players can receive news about game upgrades, new game levels, announce and organize matches, and compare scores.

- Currently each game company develops such community support in each individual game.
  - Each company uses a different infrastructure, different concepts, and provides different levels of support.

- This redundancy and inconsistency leads to problems:
  - High learning curve for players joining a new community,
  - Game companies need to develop the support from scratch
  - Advertisers need to contact each individual community separately.
ARENA: The Objectives

♦ Provide a generic infrastructure for operating an arena to
  ♦ Support virtual game communities.
  ♦ Register new games
  ♦ Register new players
  ♦ Organize tournaments
  ♦ Keeping track of the players scores.

♦ Provide a framework for tournament organizers
  ♦ to customize the number and sequence of matches and the accumulation of expert rating points.

♦ Provide a framework for game developers
  ♦ for developing new games, or for adapting existing games into the ARENA framework.

♦ Provide an infrastructure for advertisers.
**Types of Requirements**

♦ **Functional requirements:**
  - Describe the interactions between the system and its environment independent from implementation
  - Examples:
    ♦ An ARENA operator should be able to define a new game.

♦ **Nonfunctional requirements:**
  - User visible aspects of the system not directly related to functional behavior.
  - Examples:
    ♦ The response time must be less than 1 second
    ♦ The ARENA server must be available 24 hours a day

♦ **Constraints (“Pseudo requirements”):**
  - Imposed by the client or the environment in which the system operates
    ♦ The implementation language must be Java
    ♦ ARENA must be able to dynamically interface to existing games provided by other game developers.
What is usually not in the requirements?

♦ System structure, implementation technology
♦ Development methodology
♦ Development environment
♦ Implementation language
♦ Reusability

♦ It is desirable that none of these above are constrained by the client. Fight for it!
**Requirements Validation**

- Requirements validation is a critical step in the development process, usually after requirements engineering or requirements analysis. Also at delivery (client acceptance test).

- **Requirements validation criteria:**
  - **Correctness:**
    - The requirements represent the client’s view.
  - **Completeness:**
    - All possible scenarios, in which the system can be used, are described, including exceptional behavior by the user or the system.
  - **Consistency:**
    - There are functional or nonfunctional requirements that contradict each other.
  - **Realism:**
    - Requirements can be implemented and delivered.
  - **Traceability:**
    - Each system function can be traced to a corresponding set of functional requirements.
Requirements Validation

♦ Problem with requirements validation: Requirements change very fast during requirements elicitation.

♦ Tool support for managing requirements:
  - Store requirements in a shared repository
  - Provide multi-user access
  - Automatically create a system specification document from the repository
  - Allow change management
  - Provide traceability throughout the project lifecycle

♦ RequisitPro from Rational

♦ Request Tool (Allen Dutoit)
  - Tomorrow’s tutorial (November 9)

♦ [Agile Programming adapts by fine-grain incremental dev.]
Types of Requirements Elicitation

♦ Greenfield Engineering
  ♦ Development starts from scratch, no prior system exists, the requirements are extracted from the end users and the client
  ♦ Triggered by user needs
  ♦ Example: Develop a game from scratch: Asteroids

♦ Re-engineering
  ♦ Re-design and/or re-implementation of an existing system using newer technology
  ♦ Triggered by technology enabler
  ♦ Example: Reengineering an existing game

♦ Interface Engineering
  ♦ Provide the services of an existing system in a new environment
  ♦ Triggered by technology enabler or new market needs
  ♦ Example: Interface to an existing game (Bumpers)
  ♦ Example: CORBA (Common Object Request Broker Architecture)
Scenarios

♦ “A narrative description of what people do and experience as they try to make use of computer systems and applications” [M. Carrol, Scenario-based Design, Wiley, 1995]

♦ A concrete, focused, informal description of a single feature of the system used by a single actor.

♦ Scenarios can have many different uses during the software lifecycle

   - Requirements Elicitation: As-is scenario, visionary scenario
   - Client Acceptance Test: Evaluation scenario
   - System Deployment: Training scenario.
Types of Scenarios

♦ As-is scenario:
  ♦ Used in describing a current situation. Usually used in re-engineering projects.
    ♦ The user describes the system.
    ♦ Example: Description of Letter-Chess

♦ Visionary scenario:
  ♦ Used to describe a future system. Usually used in greenfield engineering and reengineering projects.
  ♦ Can often not be done by the user or developer alone
    ♦ Example: Description of an interactive internet-based Tic Tac Toe game tournament.

♦ Evaluation scenario:
  ♦ User tasks against which the system is to be evaluated.
    ♦ Example: Four users (two novice, two experts) play in a TicTac Toe tournament in ARENA.

♦ Training scenario:
  ♦ Step by step instructions that guide a novice user through a system
    ♦ Example: How to play Tic Tac Toe in the ARENA Game Framework.
How do we find scenarios?

- Don’t expect the client to be verbal if the system does not exist (greenfield engineering)
- Don’t wait for information even if the system exists
- Engage in a dialectic approach (evolutionary, incremental engineering)
  - You help the client to formulate the requirements
  - The client helps you to understand the requirements
  - The requirements evolve while the scenarios are being developed
  - [Agile Programming: they continue to evolve while prototyping]
Heuristics for finding Scenarios

♦ Ask yourself or the client the following questions:
  ♦ What are the primary tasks that the system needs to perform?
  ♦ What data will the actor create, store, change, remove or add in the system?
  ♦ What external changes does the system need to know about?
  ♦ What changes or events will the actor of the system need to be informed about?

♦ However, don’t rely on questionnaires alone.
♦ Insist on task observation if the system already exists (interface engineering or reengineering)
  ♦ Ask to speak to the end user, not just to the software contractor
  ♦ Expect resistance and try to overcome it
Example: Accident Management System

- What needs to be done to report a “Cat in a Tree” incident?
- What do you need to do if a person reports “Warehouse on Fire?”
- Who is involved in reporting an incident?
- What does the system do, if no police cars are available? If the police car has an accident on the way to the “cat in a tree” incident?
- What do you need to do if the “Cat in the Tree” turns into a “Grandma has fallen from the Ladder”?
- Can the system cope with a simultaneous incident report “Warehouse on Fire?”
Scenario Example: Warehouse on Fire

- Bob, driving down main street in his patrol car notices smoke coming out of a warehouse. His partner, Alice, reports the emergency from her car.

- Alice enters the address of the building, a brief description of its location (i.e., north west corner), and an emergency level. In addition to a fire unit, she requests several paramedic units on the scene given that area appear to be relatively busy. She confirms her input and waits for an acknowledgment.

- John, the Dispatcher, is alerted to the emergency by a beep of his workstation. He reviews the information submitted by Alice and acknowledges the report. He allocates a fire unit and two paramedic units to the Incident site and sends their estimated arrival time (ETA) to Alice.

- Alice received the acknowledgment and the ETA.
Observations about Warehouse on Fire Scenario

♦ Concrete scenario
  ♦ Describes a single instance of reporting a fire incident.
  ♦ Does not describe all possible situations in which a fire can be reported.

♦ Participating actors
  ♦ Bob, Alice and John
Next goal, after the scenarios are formulated:

♦ Find all the use cases in the scenario that specifies all possible instances of how to report a fire
  ♦ Example: “Report Emergency” in the first paragraph of the scenario is a candidate for a use case

♦ Describe each of these use cases in more detail
  ♦ Participating actors
  ♦ Describe the Entry Condition
  ♦ Describe the Flow of Events
  ♦ Describe the Exit Condition
  ♦ Describe Exceptions
  ♦ Describe Special Requirements (Constraints, Nonfunctional Requirements)
Use Cases

- A use case is a flow of events in the system, including interaction with actors
- It is initiated by an actor
- Each use case has a name
- Each use case has a termination condition
- Graphical Notation: An oval with the name of the use case

Use Case Model: The set of all use cases specifying the complete functionality of the system
Example: Use Case Model for Incident Management

[Info content and flow direction(s) still unspecified.]
Heuristics: How do I find use cases?

♦ Select a narrow vertical slice of the system (i.e. one scenario)
  ▪ Discuss it in detail with the user to understand the user’s preferred style of interaction

♦ Select a horizontal slice (i.e. many scenarios) to define the scope of the system.
  ▪ Discuss the scope with the user

♦ Use illustrative prototypes (mock-ups) as visual support

♦ Find out what the user does
  ▪ Task observation (Good)
  ▪ Questionnaires (Bad)
Use Case Example: ReportEmergency

♦ Use case name: ReportEmergency
♦ Participating Actors:
  ◦ FieldOfficer (Bob and Alice in the Scenario)
  ◦ Dispatcher (John in the Scenario)
♦ Exceptions:
  ◦ The FieldOfficer is notified immediately if the connection between her terminal and the central is lost.
  ◦ The Dispatcher is notified immediately if the connection between any logged in FieldOfficer and the central is lost.
♦ Flow of Events: on next slide.
♦ Special Requirements:
  ◦ The FieldOfficer’s report is acknowledged within 30 seconds. The selected response arrives no later than 30 seconds after it is sent by the Dispatcher.
Use Case Example: ReportEmergency
Flow of Events

- The **FieldOfficer** activates the “Report Emergency” function of her terminal. FRIEND responds by presenting a form to the officer.

- The FieldOfficer fills the form, by selecting the emergency level, type, location, and brief description of the situation. The FieldOfficer also describes possible responses to the emergency situation. Once the form is completed, the FieldOfficer submits the form, at which point, the **Dispatcher** is notified.

- The Dispatcher reviews the submitted information and creates an Incident in the database by invoking the OpenIncident use case. The Dispatcher selects a response and acknowledges the emergency report.

- The FieldOfficer receives the acknowledgment and the selected response.
Another Use Case Example: Allocate a Resource

♦ **Actors:**
  - *Field Supervisor:* This is the official at the emergency site....
  
  - *Resource Allocator:* The Resource Allocator is responsible for the commitment and decommitment of the Resources managed by the FRIEND system. ...

  - *Dispatcher:* A Dispatcher enters, updates, and removes Emergency Incidents, Actions, and Requests in the system. The Dispatcher also closes Emergency Incidents.

  - *Field Officer:* Reports accidents from the Field
**Another Use Case Example: Allocate a Resource**

- **Use case name:** AllocateResources
- **Participating Actors:**
  - Field Officer (Bob and Alice in the Scenario)
  - Dispatcher (John in the Scenario)
  - Resource Allocator
  - Field Supervisor
- **Entry Condition**
  - The Resource Allocator has selected an available resource.
  - The resource is currently not allocated
- **Flow of Events**
  - The Resource Allocator selects an Emergency Incident.
  - The Resource is committed to the Emergency Incident.
- **Exit Condition**
  - The use case terminates when the resource is committed.
  - The selected Resource is now unavailable to any other Emergency Incidents or Resource Requests. *[This changes the available inventory (database state).*]
- **Special Requirements**
  - The Field Supervisor is responsible for managing the Resources
Order of steps when formulating use cases

♦ First step: name the use case
  ♦ Use case name: ReportEmergency

♦ Second step: Find the actors
  ♦ Generalize the concrete names ("Bob") to participating actors ("Field officer")
  ♦ Participating Actors:
    ♦ Field Officer (Bob and Alice in the Scenario)
    ♦ Dispatcher (John in the Scenario)

♦ Third step: Then concentrate on the flow of events
  ♦ Use informal natural language
Use Case Associations

- A use case model consists of use cases and use case associations
  - A use case association is a relationship between use cases
- Important types of use case associations: Include, Extends, Generalization
- Include
  - A use case uses another use case ("functional decomposition")
- Extends
  - A use case extends another use case
- Generalization
  - An abstract use case has several different specializations
Problem:

- A function in the original problem statement is too complex to be solvable immediately

Solution:

- Describe the function as the aggregation of a set of simpler functions. The associated use case is decomposed into smaller use cases
<<Include>>: Reuse of Existing Functionality

- **Problem:**
  - There are already existing functions. How can we *reuse* them?

- **Solution:**
  - The *include association* from a use case A to a use case B indicates that an instance of the use case A performs all the behavior described in the use case B ("A delegates to B")

- **Example:**
  - The use case "ViewMap" describes behavior that can be used by the use case "OpenIncident" ("ViewMap" is factored out)

Note: The base case cannot exist alone. It is always called with the supplier use case. [Base class is a client of the supplier as server]
Problem:

- The functionality in the original problem statement needs to be extended.

Solution:

- An extend association from a use case A to a use case B indicates that use case B is an extension of use case A.

Example:

- The [*base*] use case “ReportEmergency” is complete by itself, but can be extended by the use case “Help” for a specific scenario in which the user requires help.

Note: The base use case can be executed without the use case extension in extend associations.
Generalization association in use cases

♦ Problem:
  ♦ You have common behavior among use cases and want to factor this out.

♦ Solution:
  ♦ The generalization association among use cases factors out common behavior. The child use cases inherit the behavior and meaning of the parent use case and add or override some behavior.

♦ Example:
  ♦ Consider the use case “ValidateUser”, responsible for verifying the identity of the user. The customer might require [one of] two realizations: “CheckPassword” or “CheckFingerprint” [depending on the context].
From Use Cases to Objects

Top Level Use Case

Level 2 Use Cases

Level 3 Use Cases

Operations

A and B are called Participating Objects

A

B

Level 1

Level 2

Level 3

Level 4

Level 2

Level 3

Level 4
Use Cases can be used by more than one object

- Top Level Use Case
- Level 2 Use Cases
- Level 3 Use Cases
- Operations
- Participating Objects

Diagram:
- Level 1
  - Level 2
    - Level 3
      - Level 4
        - A
      - Level 4
        - B
How to Specify a Use Case (Summary)

♦ Name of Use Case
♦ Actors
  ♦ Description of Actors involved in use case
♦ Entry condition
  ♦ “This use case starts when…”
♦ Flow of Events
  ♦ Free form, informal natural language
♦ Exit condition
  ♦ “This use cases terminates when…”
♦ Exceptions
  ♦ Describe what happens if things go wrong
♦ Special Requirements
  ♦ Nonfunctional Requirements, Constraints
Summary

♦ The requirements process consists of requirements elicitation and analysis.
♦ The requirements elicitation activity is different for:
  ◦ Greenfield Engineering, Reengineering, Interface Engineering
♦ Scenarios:
  ◦ Great way to establish communication with client
  ◦ Different types of scenarios: As-Is, visionary, evaluation and training
  ◦ Use cases: Abstraction of scenarios
♦ Pure functional decomposition is bad:
  ◦ Leads to unmaintainable code
♦ Pure object identification is bad:
  ◦ May lead to wrong objects, wrong attributes, wrong methods
♦ The key to successful analysis:
  ◦ Start with use cases and then find the participating objects
  ◦ If somebody asks “What is this?”, do not answer right away. Return the question or observe the end user: “What is it used for?”
  ◦ [See Tom DeMarco: SSA and D for comments on communicating with users]
Outline

♦ Model and reality (more in Appendix)
♦ From use cases to class diagrams
♦ A little discourse into philosophy (in Appendix)
♦ Activities during object modeling
♦ Object identification
♦ Object types
  ♦ entity, boundary and control objects
♦ Abbott’s technique helps in object identification
♦ Users of class diagrams

Self reading
Reality and Model

- Reality R: Real Things, People, Processes happening during some time, Relationship between things
- Model M: Abstractions from (really existing or only thought of) things, people, processes and relationships between these abstractions.

Why models?

- We use models
  - To abstract away from details in the reality, so we can draw complicated conclusions in the reality with simple steps in the model
  - To get insights into the past or presence
  - To make predictions about the future
**Models are falsifiable**

- In the middle age people believed in truth
- Models of reality cannot be true
- A model is always an approximation
  - We must say “according to our knowledge”, or “with today’s knowledge”
- Popper (“Objective Knowledge”):
  - We can only build models from reality, which are “true” until, we have found **a counter example** (*Principle of Falsification*)
    - And even then we might stick with the model ("because it works quite well in most settings")
- The falsification principle is the basis of software development
  - The goal of prototypes, reviews and system testing is to falsify the software system
Models of models of models...

- Modeling is relative. **We can think of a model as reality and can build another model from it (with additional abstractions).**

---

The development of Software-Systems is a Transformation of Models:
Analysis, Design, Implementation, Testing
From Use Cases to Objects: Why Functional Decomposition is not Enough

Why?
**Object Types**

- **Entity Objects**
  - Represent the persistent information tracked by the system (Application domain objects, “Business objects”)

- **Boundary Objects**
  - Represent the interaction between the user and the system

- **Control Objects**
  - Represent the control tasks performed by the system

- Having three types of objects leads to models that are more resilient to change.
  - The interface of a system changes more likely than the control
  - The control of the system change more likely than the application domain

- **Object types originated in Smalltalk:**
  - Model, View, Controller (MVC)

*Any relationship to J2EE?*
UML provides several mechanisms to extend the language.

- UML provides the **stereotype** mechanism to present new modeling elements.
Recommended Naming Convention for Object Types

♦ To distinguish the different object types on a syntactical basis, we recommend suffixes:
♦ Objects ending with the “_Boundary” suffix are boundary objects
♦ Objects ending with the “_Control” suffix are control objects
♦ Entity objects do not have any suffix appended to their name.

Year
Month
Day

ChangeDate_Control
Button_Boundary
LCDDisplay_Boundary
Recommended Diagrammatic Convention for Object Types

http://www-01.ibm.com/support/docview.wss?rcss=faqtt_2Q09&uid=swg21199365
Recommended Diagrammatic Convention for Object Types
4.8.2 Class Stereotypes

Business objects come in the following kinds:

- actor (defined in the UML)
- worker
- case worker
- internal worker
- entity

**Worker**
A Worker is a class that represents an abstraction of a human that acts within the system. A worker interacts with other workers and manipulates entities while participating in use case realizations.

**Case Worker**
A Case Worker is a worker who interacts directly with actors outside the system.

**Internal Worker**
An Internal Worker is a worker that interacts with other workers and entities inside the system.

**Entity**
An Entity is a class that is passive; that is, it does not initiate interactions on its own. An entity object may participate in many different use case realizations and usually outlives any single interaction. In business modeling, entities represent objects that workers access, inspect, manipulate, produce, and so on. Entity objects provide the basis for sharing among workers participating in different use case realizations.
Order of activities in modeling

1. Formulate a few **scenarios** with help from the end user and/or application domain expert.
2. Extract the **use cases** from the scenarios, with the help of application domain expert.
3. Analyse the **flow of events**, for example with *Abbot's textual analysis*.
4. Generate the **class diagrams**, which includes the following steps, as before:
   1. Class identification (textual analysis, domain experts).
   2. Identification of attributes and operations (sometimes before the classes are found!)
   3. Identification of associations between classes
   4. Identification of multiplicities
   5. Identification of roles
   6. Identification of constraints
Example: Flow of events

- The customer enters a store with the intention of buying a toy for his child with the age of n.
- Help must be available within less than one minute.
- The store owner gives advice to the customer. The advice depends on the age range of the child and the attributes of the toy.
- The customer selects a dangerous toy which is kind of unsuitable for the child.
- The store owner recommends a more yellow doll.

Is this about software?
Mapping parts of speech to object model components

[Abbott, 1983]

<table>
<thead>
<tr>
<th>Part of speech</th>
<th>Model component</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proper noun</td>
<td>object</td>
<td>Jim Smith</td>
</tr>
<tr>
<td>Improper noun</td>
<td>class</td>
<td>Toy, doll</td>
</tr>
<tr>
<td>Doing verb</td>
<td>method</td>
<td>Buy, recommend</td>
</tr>
<tr>
<td>being verb</td>
<td>inheritance</td>
<td>is-a (kind-of)</td>
</tr>
<tr>
<td>having verb</td>
<td>aggregation</td>
<td>has an</td>
</tr>
<tr>
<td>modal verb</td>
<td>constraint</td>
<td>must be</td>
</tr>
<tr>
<td>adjective</td>
<td>attribute</td>
<td>3 years old</td>
</tr>
<tr>
<td>transitive verb</td>
<td>method</td>
<td>enter</td>
</tr>
<tr>
<td>intransitive verb</td>
<td>method (event)</td>
<td>depends on</td>
</tr>
</tbody>
</table>
The customer enters the store to buy a toy. It has to be a toy that his daughter likes and it must cost less than 50 Euro. He tries a videogame, which uses a data glove and a head-mounted display. He likes it.

An assistant helps him. The suitability of the game depends on the age of the child. His daughter is only 3 years old. The assistant recommends another type of toy, namely a boardgame. The customer buy the game and leaves the store.
Any other path than Happy Path?

Happy Path:
Customer Places Order with Credit Card

1. Customer supplies customer number
2. System verifies customer number
3. For each product desired the customer supplies:
   3a. Product
   3b. Quantity
4. Customer Supplies credit card
5. System verifies credit card
6. System allocates inventory
7. System supplies delivery date

http://www.theserverside.com/tt/articles/article.tss?l=JavaUML
What follows a use case diagram?
What are these two called in Java?
Requirements Analysis Document Template

1. Introduction
2. Current system
3. Proposed system
   3.1 Overview
   3.2 Functional requirements
   3.3 Nonfunctional requirements
   3.4 Constraints ("Pseudo requirements")
   3.5 System models
      3.5.1 Scenarios
      3.5.2 Use case model
      3.5.3 Object model
         3.5.3.1 Data dictionary
         3.5.3.2 Class diagrams
      3.5.4 Dynamic models
      3.5.5 User interface
4. Glossary
Summary

♦ Modeling vs reality
♦ System modeling
  ♦ Object model
  ♦ Dynamic model
  ♦ Functional model
♦ Object modeling is the central activity
  ♦ Class identification is a major activity of object modeling
  ♦ There are some easy syntactic rules to find classes/objects
♦ Different roles during software development
♦ Requirements Analysis Document Structure
Appendix: Users of class diagrams
Who uses class diagrams?

♦ Purpose of Class diagrams:
  ♦ The description of the static properties of a system (main purpose)

♦ Who uses class diagrams?
  ♦ The customer and the end user are often not interested in class diagrams. They usually focus more on the functionality of the system.
  ♦ The application domain expert uses class diagrams to model the application domain
  ♦ The developer uses class diagrams during the development of a system, that is, during analysis, system design, object design and implementation.
Application domain vs solution domain

♦ Application domain:
  ♦ The problem domain (financial services, meteorology, accident management, architecture, ...).

♦ Application domain class:
  ♦ An abstraction in the application domain. If we model business applications, these classes are also called business objects.
  ♦ Example: Board game, Tournament

♦ Solution domain:
  ♦ Domains that help in the solution of problems (tele communication, data bases, compiler construction, operating systems, ...)

♦ Solution domain class:
  ♦ An abstraction, that is introduced for technical reasons, because it helps in the solution of a problem.
  ♦ Examples: Tree, Hashtable, Scheduler

Is the System part of the Domain or part of the Solution to some Problem in the Domain?
Is the System a high level abstraction of the Program?
The Role of the Analyst

♦ The analyst is interested
  ♦ in application classes: The associations between classes are relationships between abstractions in the application domain.
  ♦ whether the use of inheritance in the model reflect the taxonomies in the application domain --- Definition Taxonomy: A hierarchy of abstractions

♦ The analyst is not interested
  ♦ in the exact signature of operations.
  ♦ in solution classes.

Designer

♦ The designer focuses on the solution of the problem, that is the solution domain.
♦ Design consists of many tasks (subsystem decomposition, selection of the hardware platform, data management system, etc.).

♦ An important design problem is the specification of interfaces:
  ♦ The designer describes the interface of classes (object design) and subsystems (system design).
  ♦ The goal of the designer is usability and reusability of interface
    ♦ Design-Usability: the interfaces are usable from as many classes as possible within in the system.
    ♦ Design-Reusability: Definition of interfaces, such that they can also be used in other (future) software systems. => Class libraries.
Why do we distinguish these different users of class diagrams?

- Models often don’t distinguish between application classes (“address book”) and solution class (“array", “tree”).
  - Reason: Modelling languages like UML allow the use of both types of classes in the same model.
  - Preferred: No solution classes in the analysis model.

- Many systems don’t distinguish between specification and implementation of a class.
  - Reason: Object-oriented programming languages allow the simultaneous use of specification and implementation of a class.
  - Preferred: The object design model does not contain implementations.

- The key for creating high quality software systems is the exact distinction between
  - Application and solution domain classes
  - Interface specification and implementation specification

Requirements Elicitation: Definition of the system in terms understood by the customer (“Problem Description”)
Requirements Analysis: Technical specification of the system in terms understood by the developer (“Problem Specification”)

**Analysis model**

- The Analysis model is constructed during the analyse phase.
  - Main stake holders: End user, Customer, Analyst.
  - The diagram contains only application domain classes.

- The analysis model is the base for communication between analysts, experts in the application domain and end users of the system.

**Object design model**

- The object design model (sometimes also called specification model) is created during the object design phase.
  - Main stake holders are class specifiers, class implementors and class users.
  - The class diagrams contain applikation and solution domain classes.

- The object design model is the basis of communication between designers and implementors.
Appendix: Additional Slides
Ways to find objects

♦ Syntactical investigation with Abbott‘s technique:
  ♦ In the problem statement (originally proposed, but rarely works if the problem statement is large (more than 5 pages)
  ♦ In the flow of events of use cases
  ♦ => Textual Analysis with Abbott

♦ Use of various knowledge sources:
  ♦ Application knowledge: Interviews of end users and experts, to determine the abstractions of the application domain.
  ♦ Design knowledge: Reusable abstractions in the solution domain.
  ♦ General world knowledge: Also use your generic knowledge and intuition.

♦ Formulation of scenarios (in natural language):
  ♦ Description of the concrete usage of the system.

♦ Formulation of use cases (natural language and UML):
  ♦ Description of functions with actors and flow of events
From Use Cases to Objects

- Level 1 Use Case
- Level 2 Use Cases
- Level 3 Use Cases
- Operations
- Participating Objects
How do we model complex systems (Natural Systems, Social Systems, Artificial Systems)?

**Epistemology**
Describes our knowledge about the system

- **Knowledge about Causality** (Dynamic Model)
  - State Diagrams (Harel)
  - Activity Diagrams (Lamport)
  - Sequence Diagrams (Lamport)
  - Petri Nets (Petri)

- **Knowledge about Relationships** (Object model)
  - Inheritance
    - Frames, Semantic Networks (Minsky)
  - Class Diagrams (“E/R + Inheritance”, Rumbaugh)

- **Knowledge about Functionality** (Functional model)
  - Data Relationship (E/R Modeling, Chen)
  - Scenarios/Use Cases (Jacobsen)
  - Formal Specifications (Liskov)
  - DataFlow Diagrams (SA/SD)
  - Neural Networks

- **Uncertain Knowledge**
  - Uncertain Knowledge
  - Fuzzy Sets (Zadeh)
  - Fuzzy Frames (Graham)

- **Hierarchical Database Model** (IMS)
- **Network Database Model** (CODASYL)
- **Relational Database Model** (Codd)
What is a “good” model?

- Relationships, which are valid in reality R, are also valid in model M.
  - **I**: Mapping of real things in reality R to abstractions in the model M (Interpretation)
  - **f_M**: relationship between abstractions in M
  - **f_R**: relationship between real things in R

- In a good model the following diagram is commutative:

![Diagram](https://via.placeholder.com/150)

A **map** or **binary operation** \( f : A \times A \rightarrow B \) from a set \( A \) to a set \( B \) is said to be **commutative** if,

\[
f(y, z) = f(z, y) \quad \forall y, z \in A
\]
A small discourse into Philosophy

♦ Philosophy works on 3 major problems
  ♦ **Metaphysics**: What is reality?
  ♦ **Epistemology**: What is knowledge? How can we store knowledge in our brain? How far can I describe reality with knowledge?
  ♦ **Ethics**: What is good, what is bad?

♦ Metaphysics and epistemology depend on each other:
  ♦ Assertions about reality depend on closely on assertions about knowledge and vice versa.

♦ Relationship to software engineering
  ♦ Metaphysics <=> Modeling
  ♦ Epistemology <=> Acquisition of knowledge, knowledge management
  ♦ Ethics: <=> Good and bad practices during software development
The four basic questions in metaphysics

1. Is reality real or not real?
   Does reality exist only in our brain or does it exist independently from our existence?

2. What is reality made out of?

3. How many realities are there (1, 2, many)?

4. Is reality constant or does it change?
1. Reality: Real or ideal?

- The metaphysical realism assumes, that reality is real
  - Reality exists outside our brain. It is “really” real. Subtypes of Realism:
    - Naïve realism: Things are real, that is a fact!
    - Critical realism (transcendental realism): Things are real, but I see only what I want to see
    - Pragmatic realism: Realism works, that’s why reality is real
- The metaphysical idealism assumes that reality is an illusion.
Categorization of the various types of realism

Metaphysical Realism

Naive Realism

Critical Realism

pragmatic realism

Example of a categorisation (Taxonomy, Ontology)

Metaphysical Realism

Naive Realism

Critical Realism

Pragmatic Realism
2. *What is reality made out of?*

- **Materialism:**
  - Reality consists of real things
  - *Socrates:* Everything is made out of water

- **Antimaterialism:**
  - Reality consists of real things as well as of ideas
  - *Plato:* A form, e.g. beauty, is as real as real things, e.g. This little train (actually forms are more real, because they are permanent, real things live only for a short time)

- **Scientific materialism:**
  - Reality consists only of things that have energy and/or mass
  - *Modern science:* mind-reading capability is not real
Model of Plato’s Antimaterialism

- Reality
  - Material Thing
  - Form (Essence, Idea)

Taxonomies, Ontologies, Inheritance Trees
Modeling Animals

Animal Kingdom
:Reality

Mammal

Tiger

Ottobrunn:Reality

5

Tiger
3. How many realities are there?

- **Monism:**
  - There is only one thing, which is simultaneously the source and essence of reality (*Thales von Milet*: Everything is made out of water)

- **Dualism:**
  - There are 2 different sources for things in Reality
  - *Plato*: Forms and Material Things are 2 types of Reality
  - *Descartes*: The mind and the body are separate things
  - *Tao*: Each thing consists of two complementary principles: Ying und Yang

- **Pluralism:**
  - *Software Engineering*: There are many realities, the customer requirements are reality
4. Is reality constant or does it change?

♦ Parmenides (600 A.D.):
  • There is a difference between appearance and underlying reality. Change is an illusion, reality is constant

♦ Heraklit (540-475 A.D.):
  • Everything flows, there is no solid substance
    • “Jupiter’s eye” is actually a hurricane
    • Modern physics: Reality is a field of vibrations

♦ Software Engineering:
  • The graphical user interface (“GUI”) changes, but the underlying business process is constant.
  • WIMP: Windows, Icons, Mouse and Pointing Device
  • The business process changes as result of technology enablers: “Change is the only constant” (Hammer&Champy, Reengineering)
The 4 basic questions in epistemology

♦ 1. How do we acquire knowledge, through our senses or through our intelligence?
♦ 2. How far can we describe or create reality with knowledge?
♦ 3. What is knowledge made out of?
♦ 4. What are the activities during knowledge acquisition?
1. How do we acquire knowledge?

♦ **Empirism:** Knowledge is acquired by experimentation and through our senses
  - Our brain is initially empty ("tabula rasa")

♦ **Rationalism:** Knowledge is acquired by our mind
  - The brain is already at birth equipped with ideas ("a priori")

♦ **Voluntarism:** Knowledge is only acquired if you want to achieve something

♦ **Intuitionism:** Knowledge is acquired by intuition
Realism:
- Concepts - fact as well as a priori concepts - are not simply copies or extensions of the sensual experience.
- Concepts are built into our mind:
  - Concepts are “remembrance” of forms. They can be triggered by senses, but they are already in our mind, they are only woken up. (Plato)
  - Concepts are categories of our mind. They are structures which allow us mentally to keep track of sensual objects. Concepts are not derived from sensor data, but are used to make sense from sensor data (Kant).

Empiricism:
- Concepts can only be produced empirically. But such concepts are not simply copies or extensions of the sensual experience.
Can we describe reality with knowledge?

♦ Epistemological idealism:
  ♦ What you know about an object, exists only in your mind. Models can only describe parts of reality, never reality.

♦ Epistemological realism:
  ♦ The knowledge about an object is independent from our mind. Models can describe reality.

♦ Epistemological idealists are pessimists:
  ♦ There are always conclusions, that you cannot draw in the model, because they depend on components in reality which are not described in the model.

♦ Epistemological realists are optimists:
  ♦ All conclusions in the model describe things in reality.
Combining metaphysics and epistemology

♦ Metaphysical realist, epistemological realist:
  ♦ There is a reality outside of my mind, I can acquire knowledge about this reality and I can represent reality with my model. (Software Engineering: Reengineering)

♦ Metaphysical realist, epistemological idealist:
  ♦ There is a reality outside of my mind, the knowledge about this reality is limited by the structures and activities of my mind (Kant)

♦ Metaphysical idealist, epistemological idealist:
  ♦ Reality depends on a (another) mind, my knowledge about this reality is limited by my mind.

♦ Metaphysical idealist, epistemological realist:
  ♦ Reality depends on a (another) mind, my mind can understand the concepts of this other mind, and I can represent this externally with models (Software Engineering: Customer specifies the system)
Combination of metaphysics and epistemology

Metaphysics
- Metaphys. Realism
- Metaphys. Idealism

Epistemology
- Epistemol. Realism
- Epistemol. Idealism

Kant

Software Engineering (Interface & Greenfield Engineering)

Reengineering
Realities for software engineers

♦ Some people say: “The computer scientist can play god, because they can create realities”. Nonsense.

♦ But: The computer scientist can model different kinds of realities and build them:
  - An existing system (physical system, technical system, social system, software system)
    - An important special case is here when the existing system is a software system. We then call it “Legacy System”
  - An idea without counterpart in reality:
    - A visionary scenario or a customer requirement.

♦ The constructed reality might actually only be part of the ideas, namely those that were realizable in software
  - Example: A visionary scenario turns out to be a dream, a customer requirement turns out to be too expensive to realize.
Object vs Class

♦ Object (instance): Exactly one thing
  ♦ This lecture on Software Engineering on November 15 from 14:30 - 16:00

♦ A class describes a group of objects with similar properties
  ♦ Game, Tournament, mechanic, car, database

♦ Object diagram: A graphic notation for modeling objects, classes and their relationships ("associations"):  
  ♦ Class diagram: Template for describing many instances of data. Useful for taxonomies, patters, schemata...
  ♦ Instance diagram: A particular set of objects relating to each other. Useful for discussing scenarios, test cases and examples
Activities during Object Modeling

♦ Main goal: Find the important abstractions
♦ What happens if we find the wrong abstractions?
  ♦ Iterate and correct the model
♦ Steps during object modeling
  ♦ 1. Class identification
    ♦ Based on the fundamental assumption that we can find abstractions
  ♦ 2. Find the attributes
  ♦ 3. Find the methods
  ♦ 4. Find the associations between classes
♦ Order of steps
  ♦ Iteration is important
Class Identification

♦ Identify the boundaries of the system
♦ Identify the important entities in the system
♦ Class identification is crucial to object-oriented modeling
♦ Basic assumption:
  ♦ 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
Class identification is an ancient problem

♦ 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
♦ Another important problem: Define system boundary.
  ♦ What object is inside, what object is outside?
How do you find classes?

- Finding objects is the central piece in object modeling
  - Learn about problem domain: Observe your client
  - Apply general world knowledge and intuition
  - Take the flow of events and find participating objects in use cases
  - Try to establish a taxonomy
  - Apply design knowledge:
    - Distinguish different types of objects
    - Apply design patterns (Lecture on design patterns)
  - Do a syntactic analysis of problem statement, scenario or flow of events
  - Abbott Textual Analysis, 1983, also called noun-verb analysis
    - Nouns are good candidates for classes
    - Verbs are good candidates for operations
**How do you find classes?**

- Finding objects is the central piece in object modeling
  - Learn about problem domain: Observe your client
  - Apply general world knowledge and intuition
  - Take the flow of events and find participating objects in use cases
  - Try to establish a taxonomy
  - Do a syntactic analysis of *problem statement, scenario* or *flow of events*
  - Abbott Textual Analysis, 1983, also called noun-verb analysis
    - Nouns are good candidates for classes
    - Verbs are good candidates for operations
  - Apply design knowledge:
    - Distinguish different types of objects
    - Apply design patterns (Lecture on design patterns)
Finding Participating Objects in Use Cases

♦ Pick a *use case* and look at its *flow of events*
  - Find terms that developers or users need to clarify in order to understand the flow of events
  - Look for recurring nouns (e.g., Incident),
  - Identify real world entities that the system needs to keep track of (e.g., FieldOfficer, Dispatcher, Resource),
  - Identify real world procedures that the system needs to keep track of (e.g., EmergencyOperationsPlan),
  - Identify data sources or sinks (e.g., Printer)
  - Identify interface artifacts (e.g., PoliceStation)

♦ Be prepared that some objects are still missing and need to be found:
  - Model the flow of events with a sequence diagram

♦ Always use the user’s terms
Another Example

Flow of events:

♦ The customer enters the store to buy a toy.
♦ It has to be a toy that his daughter likes and it must cost less than 50 Euro.
♦ He tries a videogame, which uses a data glove and a head-mounted display. He likes it.

An assistant helps him. The suitability of the game depends on the age of the child. His daughter is only 3 years old. The assistant recommends another type of toy, namely the boardgame “Monopoly”.

Is this a good use Case?

Not quite!

The use case should terminate with the customer leaving the store.
## Textual Analysis using Abbot’s technique

<table>
<thead>
<tr>
<th>Example</th>
<th>Grammatical construct</th>
<th>UML Component</th>
</tr>
</thead>
<tbody>
<tr>
<td>“Monopoly”</td>
<td>Concrete Person, Thing</td>
<td>Object</td>
</tr>
<tr>
<td>“toy”</td>
<td>noun</td>
<td>class</td>
</tr>
<tr>
<td>&quot;3 years old&quot;</td>
<td>Adjective</td>
<td>Attribute</td>
</tr>
<tr>
<td>“enters”</td>
<td>verb</td>
<td>Operation</td>
</tr>
<tr>
<td>“depends on....&quot;</td>
<td>Intransitive verb</td>
<td>Operation (Event)</td>
</tr>
<tr>
<td>“is a&quot;, “either..or&quot;, &quot;kind of...&quot;</td>
<td>Classifying verb</td>
<td>Inheritance</td>
</tr>
<tr>
<td>&quot;Has a &quot;, “consists of&quot;</td>
<td>Possessive Verb</td>
<td>Aggregation</td>
</tr>
<tr>
<td>“must be&quot;, “less than...&quot;</td>
<td>modal Verb</td>
<td>Constraint</td>
</tr>
</tbody>
</table>
Some issues in object modeling
- Improving the readability of class diagrams
- Managing object modeling
- Different users of class diagrams

Avoid Ravioli Models

Don’t put too many classes into the same package: 7+-2 (or even 5+-2)
Put Taxonomies on a separate Diagram

Account

Amount
AccountId

Deposit()
Withdraw()
GetBalance()

Savings Account
Withdraw()

Checking Account
Withdraw()

Mortgage Account
Withdraw()
Class-diagrams have different types of „users“

- According to the development activity, the developer plays different roles.
  - Analyst
  - System-Designer,
  - DetailedDesigner
  - Implementor.

- In small systems some of the roles do not exist or are played by the same person.

- Each of these roles has a different view about the models.

- Before I describe these different views, I want to distinguish the types of classes that appear in class diagrams.
  - Application domain classes
  - Solution domain classes
Pieces of an Object Model

♦ Classes
♦ Associations (Relations)
  ♦ Generic associations
  ♦ Canonical associations
    ♦ Part of- Hierarchy (Aggregation)
    ♦ Kind of-Hierarchy (Generalization)
♦ Attributes
  ♦ Detection of attributes
  ♦ Application specific
  ♦ Attributes in one system can be classes in another system
  ♦ Turning attributes to classes
♦ Operations
  ♦ Detection of operations
  ♦ Generic operations: Get/Set, General world knowledge, design patterns
  ♦ Domain operations: Dynamic model, Functional model
Class diagrams are always part of models

♦ Analysis model: Application domain model
♦ System Design and Object design models: Solution domain model

♦ Depending on our role, we look at objects and models from a different perspective. Often we are only interested in limited aspects of a model:
  ♦ => 3 kinds of interfaces in the object design model
♦ Depending on our role and the model we have different interpretations for different UML constructs:
  ♦ Different interpretations of associations
  ♦ Different interpretations of attributes
  ♦ Different interpretation of inheritance
♦ Let’s take a look at these different interpretations.
Three Types of Implementors

♦ Class implementor:
  - Implements the class. The implementor chooses appropriate data structures (for the attributes) and algorithms (for the operations), and realizes the interface of the class in a programming language.

♦ Class extender:
  - Extends the class by a subclass, which is needed for a new problem or a new application domain.

♦ Class-user (client):
  - The programmer, who wants to use an existing class (e.g. a class from a class library or a class from another subsystem).
  - The class user is only interested in the Signatures of the class operations and the preconditions, under which they can be invoked. The class user is not so much interested in the implementation of the class.
Modelling Interactions and Behaviour

Chapter 8: Modelling Interactions and Behaviour
8.1 Interaction Diagrams

♦ Interaction diagrams are used to model the dynamic aspects of a software system
  ♦ They help you to visualize how the system runs.
  ♦ An interaction diagram is often built from a use case and a class diagram.
  ♦ The objective is to show how a set of objects accomplish the required interactions with an actor.
Interactions and messages

- Interaction diagrams show how a set of actors and objects communicate with each other to perform:
  - The steps of a use case, or
  - The steps of some other piece of functionality.

- The set of steps, taken together, is called an interaction.

- Interaction diagrams can show several different types of communication.
  - E.g. method calls, messages send over the network
  - These are all referred to as messages.
Elements found in interaction diagrams

- Instances of classes
  - Shown as boxes with the class and object identifier underlined

- Actors
  - Use the stick-person symbol as in use case diagrams

- Messages
  - Shown as arrows from actor to object, or from object to object
Creating interaction diagrams

♦ You should develop a class diagram and a use case model before starting to create an interaction diagram.

♦ There are two kinds of interaction diagrams:
  ♦ Sequence diagrams
  ♦ Communication diagrams
Sequence diagrams – an example
Sequence diagrams

♦ A sequence diagram shows the sequence of messages exchanged by the set of objects performing a certain task
  ♦ The objects are arranged horizontally across the diagram.
  ♦ An actor that initiates the interaction is often shown on the left.
  ♦ The vertical dimension represents time.
  ♦ A vertical line, called a lifeline, is attached to each object or actor.
  ♦ The lifeline becomes a broad box, called an activation box during the live activation period.
  ♦ A message is represented as an arrow between activation boxes of the sender and receiver.
    ♦ A message is labelled and can have an argument list and a return value.
Sequence diagrams – same example, more details
Sequence diagrams – an example with replicated messages

- An *iteration* over objects is indicated by an asterisk preceding the message name.
Sequence diagrams – an example with object deletion

- If an object’s life ends, this is shown with an X at the end of the lifeline
Communication diagrams – an example
Communication diagrams

- Communication diagrams emphasise how the objects collaborate in order to realize an interaction
  - A communication diagram is a graph with the objects as the vertices.
  - Communication links are added between objects
  - Messages are attached to these links.
    - Shown as arrows labelled with the message name
  - Time ordering is indicated by prefixing the message with some numbering scheme.
Communication diagrams – same example, more details

1: requestToRegister(aStudent)
2: prereq := getPrerequisite
3: hasPrerequisite := hasPassedCourse(prereq)
4: create
5: addToSchedule «parameter»
6: addToRegistrationList «parameter»

:GUI
:CourseSection
:Course

aStudent:
Student
:Registration
Communication links

- A communication link can exist between two objects whenever it is possible for one object to send a message to the other one.

- Several situations can make this message exchange possible:

  1. The classes of the two objects have an association between them.
     - This is the most common case.
     - If all messages are sent in the same direction, then probably the association can be made unidirectional.
Other communication links

2. The receiving object is stored in a local variable of the sending method.
   – This often happens when the object is created in the sending method or when some computation returns an object.
   – The stereotype to be used is «local» or [L].

3. A reference to the receiving object has been received as a parameter of the sending method.
   – The stereotype is «parameter» or [P].
4. The receiving object is global.
   – This is the case when a reference to an object can be obtained using a static method.
   – The stereotype «global», or a [G] symbol is used in this case.

5. The objects communicate over a network.
   – We suggest to write «network». 
How to choose between using a sequence or communication diagram

- **Sequence diagrams**
  - Make explicit the time ordering of the interaction.
  - **Use cases make time ordering explicit too**
  - **So sequence diagrams are a natural choice when you build an interaction model from a use case.**

- Make it easy to add details to messages.
  - **Communication diagrams have less space for this**
How to choose between using a sequence or communication diagram

♦ Communication diagrams
  ♦ Can be seen as a projection of the class diagram
  ♦ Might be preferred when you are deriving an interaction diagram from a class diagram.
  ♦ Are also useful for validating class diagrams.
Communication diagrams and patterns

A communication diagram can be used to represent aspects of a design pattern.

![Diagram](attachment:image.png)

- Diagram (a) shows the interaction between Client, Proxy, and HeavyWeight.
- Diagram (b) illustrates the relationship between Student, PersistentStudent, Proxy, and CourseSection.
8.2 State Diagrams

♦ A state diagram describes the behaviour of a system, some part of a system, or an individual object.

♦ At any given point in time, the system or object is in a certain state.
  ♦ Being in a state means that it is will behave in a specific way in response to any events that occur.
  ♦ Some events will cause the system to change state.
  ♦ In the new state, the system will behave in a different way to events.

♦ A state diagram is a directed graph where the nodes are states and the arcs are transitions.
State diagrams – an example

- tic-tac-toe game (also called noughts and crosses)
States

- At any given point in time, the system is in one state.

- It will remain in this state until an event occurs that causes it to change state.

- A state is represented by a rounded rectangle containing the name of the state.

- Special states:
  - A black circle represents the start state
  - A circle with a ring around it represents an end state
Transitions

- A transition represents a change of state in response to an event.
  - It is considered to occur instantaneously.

- The label on each transition is the event that causes the change of state.
State diagrams – an example of transitions with time-outs and conditions

(a) GreenLight
   - after(25s)
   - YellowLight
     - after(30s)
     - after(5s)
   - RedLight

(b) GreenLight
   - GreenLightNoTrigger
     - vehicleWaitingToTurn
       - after(25s since exit from state RedLight)
   - GreenLightChangeTriggered
     - after(30s)
   - YellowLight
     - after(5s)
   - RedLight

© Lethbridge/Laganière 2005

Chapter 8: Modelling Interactions and Behaviour
State diagrams – an example with conditional transitions

- **Planned**
  - openRegistration

- **OpenNotEnoughStudents**
  - requestToRegister(aStudent)
  - createRegistration
  - classSize ≥ minimum

- **OpenEnoughStudents**
  - requestToRegister(aStudent)
  - createRegistration
  - classSize ≥ maximum

- **Cancelled**
  - closeRegistration
  - cancel

- **Closed**
  - closeRegistration
  - cancel

© Lethbridge/Laganière 2005
Activities in state diagrams

- An activity is something that takes place while the system is in a state.

- It takes a period of time.

- The system may take a transition out of the state in response to completion of the activity,

- Some other outgoing transition may result in:
  - The interruption of the activity, and
  - An early exit from the state.
State diagram – an example with activity

- **ProposeSelection**
  - press button
  - MusicPlaying
    - do / play chosen selection
Actions in state diagrams

- An *action* is something that takes place effectively *instantaneously*
  - When a particular transition is taken,
  - Upon entry into a particular state, or
  - Upon exit from a particular state

- An action should consume no noticeable amount of time
State diagram – an example with actions
State diagrams – another example
Nested substates and guard conditions

- A state diagram can be nested inside a state.
  - The states of the inner diagram are called substates.
State diagram – an example with substates
8.3 Activity Diagrams

- An activity diagram is like a state diagram.
  - Except most transitions are caused by *internal* events, such as the completion of a computation.

- An activity diagram
  - Can be used to understand the flow of work that an object or component performs.
  - Can also be used to visualize the interrelation and interaction between different use cases.
  - Is most often associated with several classes.

- One of the strengths of activity diagrams is the representation of *concurrent* activities.
Activity diagrams – an example

- Receive course registration request
- Check prerequisites
  - [doesNotHavePrereqs] Check special permission
    - [hasPermission] [NoSuchPermission]
- Verify course not full
  - [hasPrereqs] [notFull]
- Complete registration

Chapter 8: Modelling Interactions and Behaviour
Representing concurrency

- Concurrency is shown using forks, joins and rendezvous.

- A fork has one incoming transition and multiple outgoing transitions.
  - The execution splits into two concurrent threads.

- A rendezvous has multiple incoming and multiple outgoing transitions.
  - Once all the incoming transitions occur, all the outgoing transitions may occur.
Representing concurrency

- A *join* has **multiple** incoming transitions and **one** outgoing transition.
  - The outgoing transition will be taken when all incoming transitions have occurred.
  - The incoming transitions must be triggered in separate threads.
  - If one incoming transition occurs, a wait condition occurs at the join until the other transitions occur.
Swimlanes

♦ Activity diagrams are most often associated with several classes.
  ♦ The partition of activities among the existing classes can be explicitly shown using swimlanes.
Activity diagrams – an example with swimlanes

<table>
<thead>
<tr>
<th>Student</th>
<th>CourseSection</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Receive course registration request</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Check prerequisites</td>
<td>Verify course not full</td>
</tr>
<tr>
<td>[doesNotHavePrereqs]</td>
<td>[notFull]</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Check special permission</td>
<td>[hasPermission] [no Permission]</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>[hasPrereqs]</td>
<td>[full]</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Complete registration</td>
</tr>
</tbody>
</table>

© Lethbridge/Laganière 2005
8.4 Implementing Classes Based on Interaction and State Diagrams

- You should use these diagrams for the parts of your system that you find most complex.
  - I.e. not for every class

- Interaction, activity and state diagrams help you create a correct implementation.

- This is particularly true when behaviour is distributed across several use cases.
  - E.g. a state diagram is useful when different conditions cause instances to respond differently to the same event.
**Example**

### Class Diagram

- **Course**
  - `getPrerequisite()`
- **CourseSection**
  - `requestToRegister()`
  - `addToRegistrationList()`
- **Registration**
  - `addToSchedule()`
  - `hasPassedCourse()`
- **Student**
  - `addToSchedule()`

### State Diagram

- **Planed**
  - `openRegistration`
- **Cancelled**
  - `do / unregister students`
- **Open**
  - `requestToRegister(aStudent) / createRegistration`
  - `classSize ≥ minimum`
- **NotEnoughStudents**
  - `cancel`
  - `closeRegistration`
- **EnoughStudents**
  - `classSize ≥ minimum`
  - `closeRegistration`
- **Closed**
  - `classSize ≥ maximum`

© Lethbridge/Laganière 2005

Chapter 8: Modelling Interactions and Behaviour
Example: The CourseSection class

♦ States:
  ♦ ‘Planned’:
    closedOrCancelled == false && open == false
  ♦ ‘Cancelled’:
    closedOrCancelled == true &&
    registrationList.size() == 0
  ♦ ‘Closed’ (course section is too full, or being taught):
    closedOrCancelled == true &&
    registrationList.size() > 0
Example: The CourseSection class

♦ States:
   ♦ ‘Open’ (accepting registrations):
     
     open == true

   ♦ ‘NotEnoughStudents’ (substate of ‘Open’):
     
     open == true &&
     registrationList.size() < course.getMinimum()

   ♦ ‘EnoughStudents’ (substate of ‘Open’):
     
     open == true &&
     registrationList.size() >= course.getMinimum()
Example: The CourseSection class

public class CourseSection
{
    // The many-1 abstraction-occurrence association (Figure 8.2)
    private Course course;

    // The 1-many association to class Registration (Figure 8.2)
    private List registrationList;

    // The following are present only to determine the state
    // (as in Figure 8.19). The initial state is 'Planned'
    private boolean open = false;
    private boolean closedOrCanceled = false;


Example: The CourseSection class

```java
public CourseSection(Course course) {
    this.course = course;
    registrationList = new LinkedList();
}

public void openRegistration() {
    if(!closedOrCanceled) // must be in 'Planned' state
        {
            open = true; // to 'OpenNotEnoughStudents' state
        }
}
```
Example: The CourseSection class

```java
public void closeRegistration()
{
    // to 'Canceled' or 'Closed' state
    open = false;
    closedOrCanceled = true;
    if (registrationList.size() < course.getMinimum())
    {
        unregisterStudents(); // to 'Canceled' state
    }
}

public void cancel()
{
    // to 'Canceled' state
    open = false;
    closedOrCanceled = true;
    unregisterStudents();
}
```
Example: The CourseSection class

public void requestToRegister(Student student)
{
    if (open) // must be in one of the two 'Open' states
    {
        // The interaction specified in the sequence diagram of Figure 8.4
        Course prereq = course.getPrerequisite();
        if (student.hasPassedCourse(prereq))
        {
            // Indirectly calls addToRegistrationList
            new Registration(this, student);
        }
    }
    // Check for automatic transition to 'Closed' state
    if (registrationList.size() >= course.getMaximum())
    {
        // to 'Closed' state
        open = false;
        closedOrCanceled = true;
    }
}
Example: The CourseSection class

// Private method to remove all registrations
// Activity associated with 'Canceled' state.
private void unregisterStudents()
{
    Iterator it = registrationList.iterator();
    while (it.hasNext())
    {
        Registration r = (Registration)it.next();
        r.unregisterStudent();
        it.remove();
    }
}

// Called within this package only, by the constructor of
// Registration to ensure the link is bi-directional
void addToRegistrationList(Registration newRegistration)
{
    registrationList.add(newRegistration);
}


8.5 Difficulties and Risks in Modelling Interactions and Behaviour

♦ Dynamic modelling is a difficult skill
  ♦ In a large system there are a very large number of possible paths a system can take.
  ♦ It is hard to choose the classes to which to allocate each behaviour:
    ♦ Ensure that skilled developers lead the process, and ensure that all aspects of your models are properly reviewed.
    ♦ Work iteratively:
      – Develop initial class diagrams, use cases, responsibilities, interaction diagrams and state diagrams;
      – Then go back and verify that all of these are consistent, modifying them as necessary.
    ♦ Drawing different diagrams that capture related, but distinct, information will often highlight problems.