Design Engineering

Dr. Marouane Kessentini
Department of Computer Science
Design

- **Starts** mostly from/with **requirements** (evolving mostly from functionalities and other non-functional characteristics)

- **How is the software solution going to be structured?**
  - What are the **main components** ***--- (functional comp)***
    - Often directly from **Requirements’ Functionalities (use Cases)**
  - How are these **components related**?
    - possibly re-organize the components (composition/decomposition)

- **Two main levels of design:**
  - Architectural (high-level)
  - Detailed design
Relationship between Architecture and Design

Detailed Design come from Requirements & Architecture
Detailed Design

- **Further Refine Architecture** and match with Requirements

- How detailed?

- **Maybe of different levels of detail** for different views
Functional **Decomposition** Technique

• Dates back to “structured programming” [now (non-OO)Web apps with PHP tool]

  • **Start with**: main (task/requirements) -> module

  • **Refine into sub-modules**

• There are alternative decompositions
“Alternative” Decomposition/Composition

0. Main

1. Students
2. Courses
3. Sections
4. Registration
5. Database
   5.1 Add
   5.2 Modify
   5.3 Delete
OO Design

• **First step**: Review & Refine use cases

• Decide
  – Which classes to create
  – How the classes are related

• Use UML as the Design Language
Essentials of UML Class Diagrams

• The main symbols shown on class diagrams are:
  – **Classes**
    – represent the types of data themselves
  – **Associations**
    – represent linkages between instances of classes
  – **Attributes**
    – are simple data found in classes and their instances
  – **Operations**
    – represent the functions performed by the classes and their instances
  – **Generalizations**
    – group classes into inheritance hierarchies
Class Design

- **Classes** represent real-world entities or system concepts
- Organized into **classes**: objects in a class have similar characteristics
- **Classes** have properties (attributes or data)
- **Classes** also have methods (performs functions)

<table>
<thead>
<tr>
<th>Student</th>
</tr>
</thead>
<tbody>
<tr>
<td>dateOfBirth : Date</td>
</tr>
<tr>
<td>name : String</td>
</tr>
<tr>
<td>getAgeInYears() : int</td>
</tr>
<tr>
<td>getAgeInDays() : int</td>
</tr>
</tbody>
</table>
Classes

• A class is simply represented as a box with the name of the class inside
  – The diagram may also show the attributes and operations
  – The complete signature of an operation is:

    `operationName(parameterName: parameterType ...): returnType`
Associations and Multiplicity

- An association is used to show how two classes are related to each other
  - Symbols indicating *multiplicity* are shown at each end of the association
Labelling associations
– Each association can be labelled, to make explicit the nature of the association
Analyzing and validating associations

– Many-to-one

• A company has many employees,
• An employee can only work for one company.
  – This company will not store data about the moonlighting activities of employees!
• A company can have zero employees
  – E.g. a ‘shell’ company
• It is not possible to be an employee unless you work for a company
Analyzing and validating associations

- **Many-to-many**
  - An assistant can work for many managers
  - A manager can have many assistants
  - Assistants can work in pools
  - Managers can have a group of assistants
  - Some managers might have zero assistants.
  - Is it possible for an assistant to have, perhaps temporarily, zero managers?
Analyzing and validating associations

– One-to-one

• For each company, there is exactly one board of directors
• A board is the board of only one company
• A company must always have a board
• A board must always be of some company
Analyzing and validating associations

- Avoid unnecessary one-to-one associations

![Diagram]

Avoid this: Person

do this: PersonInfo

Person
Association classes

- Sometimes, an attribute that concerns two associated classes cannot be placed in either of the classes
- The following are equivalent
Reflexive associations

– It is possible for an association to connect a class to itself
Directionality in associations

- Associations are by default *bi-directional*
- It is possible to limit the direction of an association by adding an arrow at one end

![Diagram showing a bidirectional association labeled Day to Note]
Generalization

• Specializing a superclass into two or more subclasses
  – A generalization set is a labeled group of generalizations with a common superclass
  – The label (sometimes called the discriminator) describes the criteria used in the specialization
More Advanced Features: Aggregation

- Aggregations are special associations that represent ‘part-whole’ relationships.
  - The ‘whole’ side is often called the *assembly* or the *aggregate*
  - This symbol is a shorthand notation association named *isPartOf*

![Diagram](image-url)
When to use an aggregation

• As a general rule, you can mark an association as an aggregation if the following are true:
  – You can state that
    • the parts ‘are part of’ the aggregate
    • or the aggregate ‘is composed of’ the parts
  – When something owns or controls the aggregate, then they also own or control the parts
Composition

- A *composition* is a strong kind of aggregation
  - if the aggregate is destroyed, then the parts are destroyed as well

- Two alternatives for addresses
Suggested sequence of activities

- Identify a first set of candidate **classes**
- Add **associations** and **attributes**
- Find **generalizations**
- List the main **responsibilities** of each class
- Decide on specific **operations**
- **Iterate** over the entire process until the model is satisfactory
  - Add or delete classes, associations, attributes, generalizations, responsibilities or operations
  - Identify interfaces
  - Apply design patterns

- *Don’t be too disorganized. Don’t be too rigid either.*
A simple technique for discovering classes

– Look at a source material such as a description of requirements
– Extract the *nouns* and *noun phrases*
– Eliminate nouns that:
  • are redundant
  • represent instances
  • are vague or highly general
  • not needed in the application
Identifying associations and attributes

– 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.
– Avoid adding many associations and attributes to a class
  • A system is simpler if it manipulates less information
Tips about identifying and specifying valid associations

- An association should exist if a class
  - possesses
  - controls
  - is connected to
  - is related to
  - is a part of
  - has as parts
  - is a member of, or
  - has as members

  some other class in your model

- Specify the multiplicity at both ends
- Label it clearly.
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
Identifying generalizations and interfaces

– There are two ways to identify generalizations:
  • bottom-up
    – Group together similar classes creating a new superclass
  • top-down
    – Look for more general classes first, specialize them if needed

– Create an *interface*, instead of a superclass if
  • The classes are very dissimilar except for having a few operations in common
  • One or more of the classes already have their own superclasses
  • Different implementations of the same class might be available
Prototyping a class diagram on paper

– As you identify classes, you write their names on small cards

– As you identify attributes and responsibilities, you list them on the cards
  • If you cannot fit all the responsibilities on one card:
    – this suggests you should split the class into two related classes.

– Move the cards around on a whiteboard to arrange them into a class diagram.

– Draw lines among the cards to represent associations and generalizations.
Identifying operations

• Operations are needed to realize the responsibilities of each class
  – There may be several operations per responsibility
  – The main operations that implement a responsibility are normally declared \textbf{public}
  – Other methods that collaborate to perform the responsibility must be as private as possible
Implementing Class Diagrams in Java

- Attributes are implemented as instance variables
- Generalizations are implemented using extends
- Interfaces are implemented using implements
- Associations are normally implemented using instance variables
  - Divide each two-way association into two one-way associations
    - so each associated class has an instance variable.
  - For a one-way association where the multiplicity at the other end is ‘one’ or ‘optional’
    - declare a variable of that class (a reference)
  - For a one-way association where the multiplicity at the other end is ‘many’:
    - use a collection class implementing List, such as Vector
Difficulties and Risks when creating class diagrams

– Modeling is particularly difficult skill
  * Even excellent programmers have difficulty thinking at the appropriate level of abstraction
  * Education traditionally focus more on design and programming than modeling

– Resolution:
  * Ensure that tem members have adequate training
  * Have experienced modeler as part of the team
  * Review all models thoroughly