Software Design

Lecture 8

Design and implementation

- Software design and implementation is the stage in the software engineering process at which an executable software system is developed.
- Software design and implementation activities are invariably inter-leaved.
  - ✓ Software design is a creative activity in which you identify software components and their relationships, based on a customer's requirements.
  - ✓ Implementation is the process of realizing the design as a program.

Build or buy

- In a wide range of domains, it is now possible to buy off-the-shelf systems (COTS) that can be adapted and tailored to the users' requirements.
  - ✓ For example, if you want to implement a medical records system, you can buy a package that is already used in hospitals. It can be cheaper and faster to use this approach rather than developing a system in a conventional programming language.
- When you develop an application in this way, the design process becomes concerned with how to use the configuration features of that system to deliver the system requirements.

Elements of Complete Software Design

- Architecture (big picture)
  - ✓ Components / Packages etc
- Data Structure Design – data dictionary
  - INFO2110 – Data Structures for IT
  - COMP3161 - Database Design
- User Interface Design
  - INFO3170 – User Interface Design for IT
- Algorithm Design
  - INFO2110 Data Structures for IT
- Modelling aspects of Operations and context (how things will be done)
  - ✓ Classes/modules, Use Case Diagrams, sequence diagrams, etc.
  - ✓ Annotations/ descriptive texts

An object-oriented design process

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

Process stages

- There are a variety of different object-oriented design processes that depend on the organization using the process.
- Common activities in these processes include:
  - ✓ Define the context and modes of use of the system;
  - ✓ Design the system architecture;
  - ✓ Identify the principal system objects;
  - ✓ Develop design models;
  - ✓ Specify object interfaces.

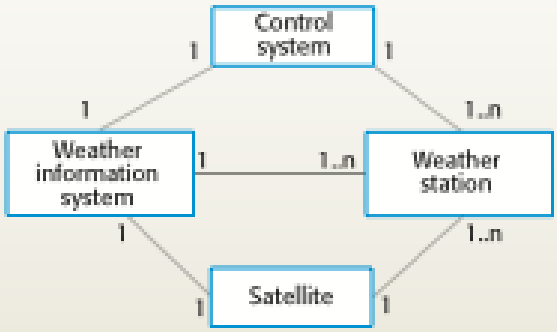
System context and interactions

- Understanding the relationships between the software that is being designed and its external environment is essential for deciding how:
  - ✓ to provide the required system functionality
  - ✓ and how to structure the system to communicate with its environment
- Understanding of the context also lets you establish the boundaries of the system.
- Setting the system boundaries helps you decide what features are implemented in the system being designed and what features are in other associated systems.

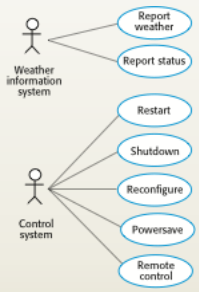
Context and interaction models

- A system context model is a structural model that demonstrates the other systems in the environment of the system being developed.
- An interaction model is a dynamic model that shows how the system interacts with its environment as it is used.

System context for the weather station



Weather station use cases



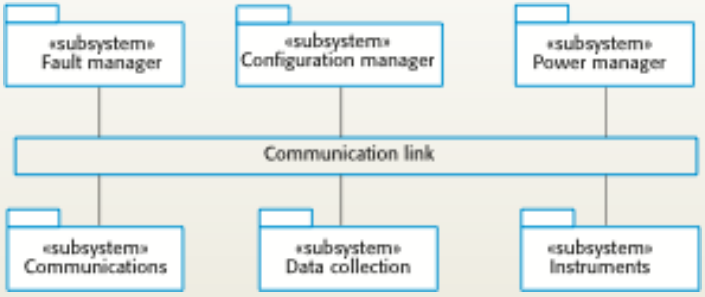
Use case description—Report weather

System	Weather station
Use case	Report weather
Actors	Weather information system, Weather station
Description	The weather station sends a summary of the weather data that has been collected from the instruments in the collection period to the weather information system. The data sent are the maximum, minimum, and average ground and air temperatures; the maximum, minimum, and average air pressures; the maximum, minimum, and average wind speeds; the total rainfall; and the wind direction as sampled at five-minute intervals.
Stimulus	The weather information system establishes a satellite communication link with the weather station and requests transmission of the data.
Response	The summarized data is sent to the weather information system.
Comments	Weather stations are usually asked to report once per hour but this frequency may differ from one station to another and may be modified in the future.

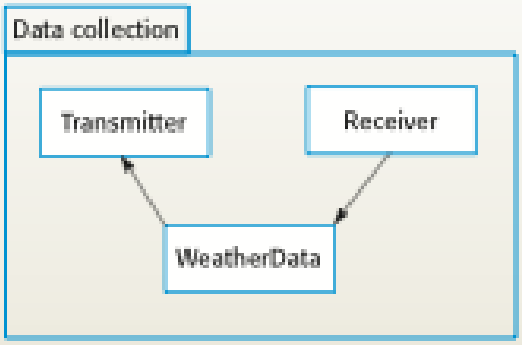
Architectural design

- Once interactions between the system and its environment have been understood, you use this information for designing the system architecture.
- You identify the major components that make up the system and their interactions, and then may organize the components using an architectural pattern such as a layered or client-server model.
- For example:
  - ✓ The weather station is composed of independent subsystems that communicate by broadcasting messages on a common infrastructure.

High-level architecture of the weather station



Architecture of data collection system



Object class identification

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

Approaches to identification

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

Weather station description

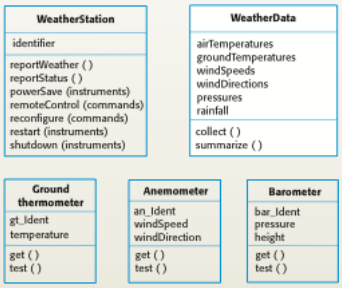
A weather station is a package of software controlled instruments which collects data, performs some data processing and transmits this data for further processing. The instruments include air and ground thermometers, an anemometer, a wind vane, a barometer and a rain gauge. Data is collected periodically.

When a command is issued to transmit the weather data, the weather station processes and summarises the collected data. The summarised data is transmitted to the mapping computer when a request is received.

Weather station object classes

- Object class identification in the weather station system may be based on the tangible hardware and data in the system:
  - ✓ Ground thermometer, Anemometer, Barometer
    - Application domain objects that are 'hardware' objects related to the instruments in the system.
  - ✓ Weather station
    - The basic interface of the weather station to its environment.
    - It therefore reflects the interactions identified in the use-case model.
  - ✓ Weather data
    - Encapsulates the summarized data from the instruments.

Weather station object classes



Design models

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

Examples of design models

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

Subsystem models

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

Class Diagrams

(review)

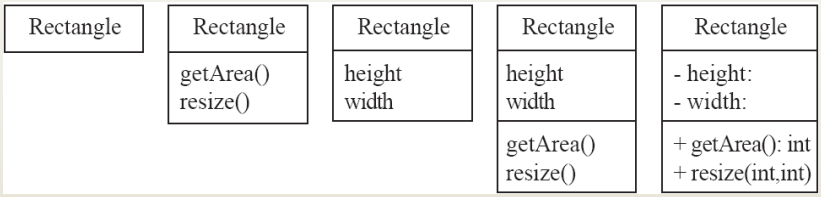
Essentials of UML Class Diagrams

- *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

Classes

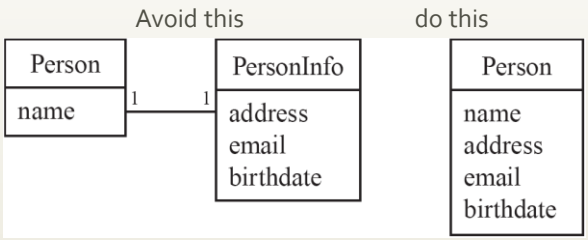
- A class is simply represented as a box with the name of the class inside
  - ✓ The diagram may also show the attributes and operations
  - ✓ The complete signature of an operation is:

    operationName(parameterName: parameterType ...): returnType



Analyzing and validating associations

- Avoid unnecessary one-to-one associations



Generalization

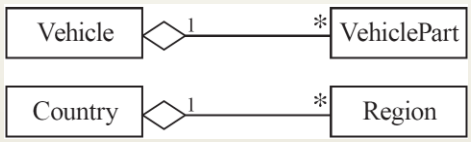
- Specializing a superclass into two or more subclasses
  - ✓ A *generalization set* is a labeled group of generalizations with a common superclass
  - ✓ The label (sometimes called the *discriminator*) describes the criteria used in the specialization

Associations versus generalizations in object diagrams

- ✓ Associations describe the relationships that will exist between *instances* at run time.
  - When you show an instance diagram generated from a class diagram, there will be an instance of *both* classes joined by an association
- ✓ Generalizations describe relationships between *classes* in class diagrams.
  - They do not appear in instance diagrams at all.
  - An instance of any class should also be considered to be an instance of each of that class's superclasses

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`



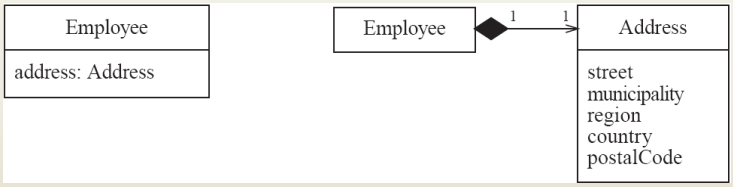
When to use an aggregation

- ❖ 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



Notes and descriptive text

- ✓ **Descriptive text and other diagrams**
  - Embed your diagrams in a larger document
  - Text can explain aspects of the system using any notation you like
  - Highlight and expand on important features, and give rationale
- ✓ **Notes**:
  - A note is a small block of text embedded *in* a UML diagram
  - It acts like a comment in a programming language

Types of Design Classes

- **User interface classes** – define all abstractions necessary for human-computer interaction (usually via metaphors of real-world objects)
- **Business domain classes** – refined from analysis classes; identify attributes and services (methods) that are required to implement some element of the business domain
- **Process classes** – implement business abstractions required to fully manage the business domain classes
- **Persistent classes** – represent data stores (e.g., a database) that will persist beyond the execution of the software
- **System classes** – implement software management and control functions that enable the system to operate and communicate within its computing environment and the outside world

33

Structured Design

- Focused on describing systems based on a hierarchy of functions (modules)
- Fundamental Modelling tools
  - ✓ Data Flow Diagram
    - Shows the movement and transformations of data within the system
  - ✓ Context Diagram
    - Shows the external actors that interacts with the system
  - ✓ Structure Charts
    - Shows the hierarchy of modules in the system
  - ✓ Data Dictionary
    - Identifies and explains the role of each datum element in the system

Design Quality Guidelines

1) A design should exhibit an architecture that
   a) Has been created using recognizable architectural styles or patterns
   b) Is composed of components that exhibit good design characteristics
   c) Can be implemented in an evolutionary fashion, thereby facilitating implementation and testing
2) A design should be modular; that is, the software should be logically partitioned into elements or subsystems
3) A design should contain distinct representations of data, architecture, interfaces, and components
4) A design should lead to data structures that are appropriate for the classes to be implemented and are drawn from recognizable data patterns

35

Quality Guidelines (continued)

5) A design should lead to components that exhibit independent functional characteristics
6) A design should lead to interfaces that reduce the complexity of connections between components and with the external environment
7) A design should be derived using a repeatable method that is driven by information obtained during software requirements analysis
8) A design should be represented using a notation that effectively communicates its meaning

"Quality isn't something you lay on top of subjects and objects like tinsel on a Christmas tree."

36

Design Concepts

- Abstraction
  - ✓ Procedural abstraction – a sequence of instructions that have a specific and limited function
  - ✓ Data abstraction – a named collection of data that describes a data object
- Architecture
  - ✓ The overall structure of the software and the ways in which the structure provides conceptual integrity for a system
  - ✓ Consists of components, connectors, and the relationship between them
- Patterns
  - ✓ A design structure that solves a particular design problem within a specific context
  - ✓ It provides a description that enables a designer to determine whether the pattern is applicable, whether the pattern can be reused, and whether the pattern can serve as a guide for developing similar patterns

(more on next slide)

Design Concepts ||

- Modularity
  - ✓ Separately named and addressable components (i.e., modules) that are integrated to satisfy requirements (divide and conquer principle)
  - ✓ Makes software intellectually manageable so as to grasp the control paths, span of reference, number of variables, and overall complexity
- Information hiding
  - ✓ The designing of modules so that the algorithms and local data contained within them are inaccessible to other modules
  - ✓ This enforces access constraints to both procedural (i.e., implementation) detail and local data structures
- Functional independence
  - ✓ Modules that have a "single-minded" function and an aversion to excessive interaction with other modules
  - ✓ High cohesion – a module performs only a single task
  - ✓ Low coupling – a module has the lowest amount of connection needed with other modules

(more on next slide)

Design Concepts III

- Stepwise refinement
  - ✓ Development of a program by successively refining levels of procedure detail
  - ✓ Complements abstraction, which enables a designer to specify procedure and data and yet suppress low-level details
- Refactoring
  - ✓ A reorganization technique that simplifies the design (or internal code structure) of a component without changing its function or external behavior
  - ✓ Removes redundancy, unused design elements, inefficient or unnecessary algorithms, poorly constructed or inappropriate data structures, or any other design failures
- Design classes
  - ✓ Refines the analysis classes by providing design detail that will enable the classes to be implemented
  - ✓ Creates a new set of design classes that implement a software infrastructure to support the business solution

39