SOFTWARE ENGINEERING

Week 8
Software Design Engineering I

Agenda

1. Unified Modeling Language
2. Software Design Concepts
3. Structured Design
4. Object Oriented Design Principles
5. User Interface Design
6. Case Study: SafeHome

Unified Modeling Language

8.1

What is UML?

- Unified Modeling Language (UML) is the standard tool for visualizing, specifying, constructing, and documenting the artifacts of an object-oriented software.
- UML is not a programming language, but only a visual design notation.
- Can be used with all software development process models.
- Independent of implementation language.
- Many CASE tools uses UML for automatic code generation. Examples: IBM Rational, ArgoUML, etc.
- You may be familiar with some UML concepts introduced in Object Oriented Programming course.

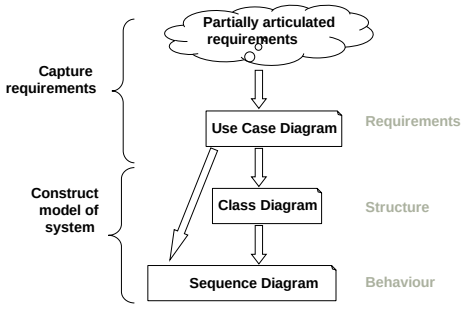
Background

- UML is the result of an effort to simplify and consolidate the large number of **Object Oriented** development methods and notations.
- Developed by the Object Management Group based on work from:
  - Grady Booch [91]
  - James Rumbaugh [91]
  - Ivar Jacobson [92]
- The latest version is UML 2.0
  (See http://www.omg.org or http://www.uml.org)

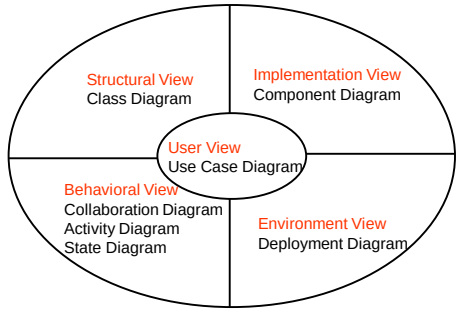
Types of UML Diagrams

- **Use Case Diagrams (*)**
- **Class Diagrams (*)**
- Interaction Diagrams
  - **Sequence Diagrams (*)**
  - Collaboration Diagrams
- State Transition Diagrams
- Activity Diagrams
- Implementation Diagrams
  - Component Diagrams
  - Deployment Diagrams

Minimal UML Process



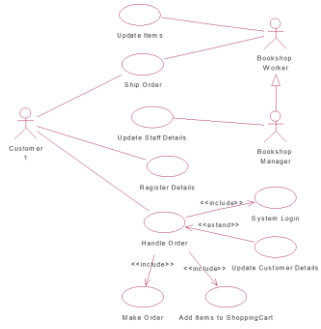
Views of UML Diagrams



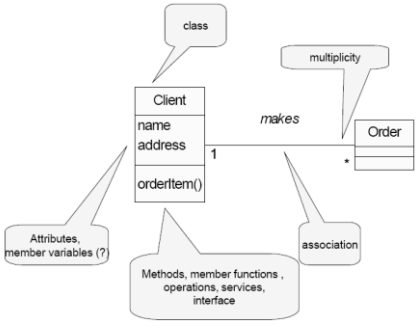
Use Cases

- Use case diagrams are used to visualize, specify, construct, and document the (intended) behavior of the system, during requirements capture and analysis.
- Provide a way for developers, domain experts and end-users to Communicate.
- Serve as basis for testing.
- Main authors: *Booch, Rumbaugh, and Jacobson*
- The Object Management Group (OMG) is responsible for standardization. (www.omg.org)
- Current version is UML 2.0

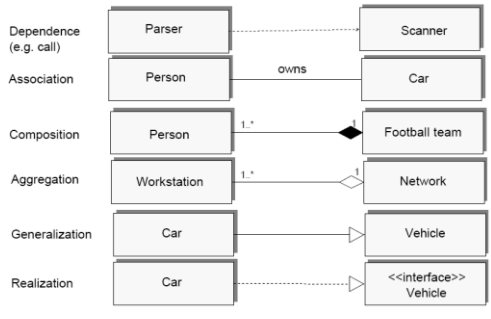
Example : Use Case Diagram



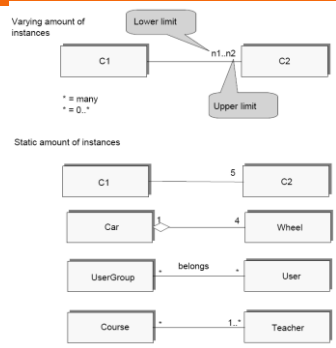
Class Diagrams



Class Relations

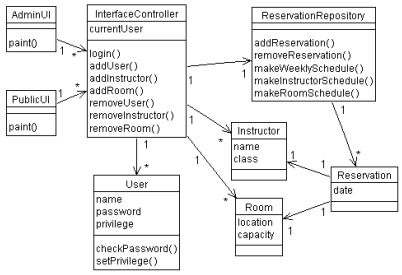


Cardinality

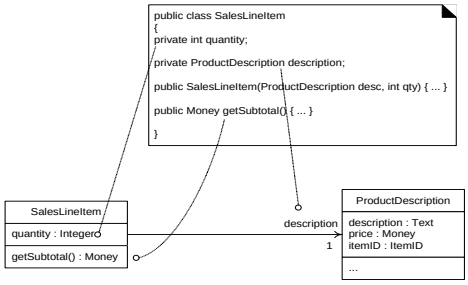


Example : Class Diagram

A classroom scheduling system: specification perspective.



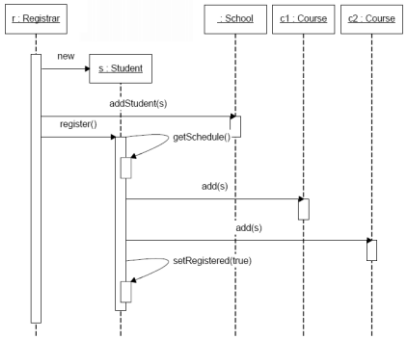
Example : From Class Diagram to Code



Interaction Diagrams

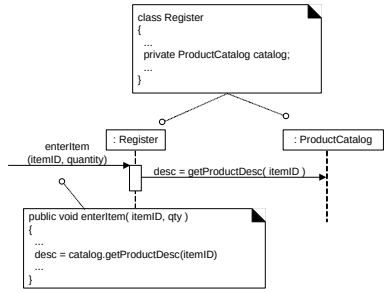
- UML presents two types of interaction diagrams.
  1. Sequence Diagrams
  2. Collaboration Diagrams

Sequence Digrams

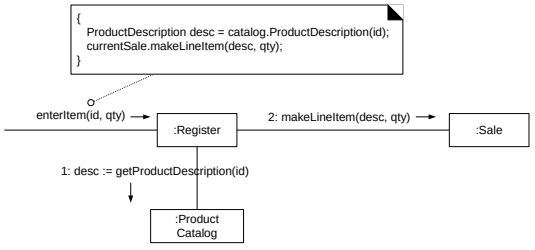


Example : Mapping Diagram to Code

- Mapping sequence diagram to Java code

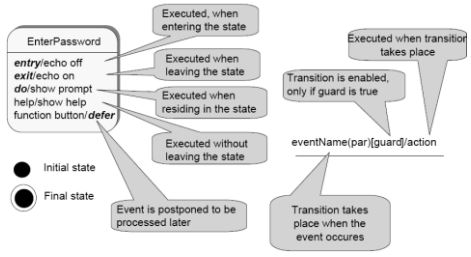


Collaboration Diagrams

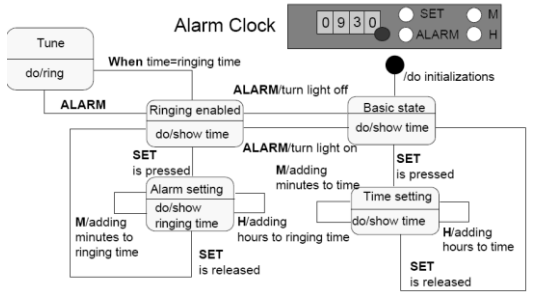


State Chart Notation

- State transition diagram shows
  - The events that cause a transition from one state to another
  - The actions that result from a state change

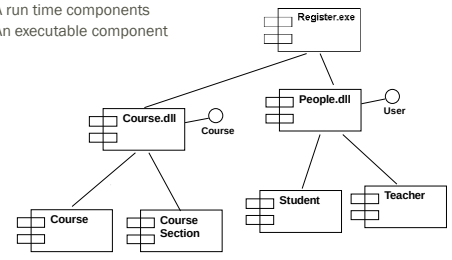


Example : State Chart Diagram



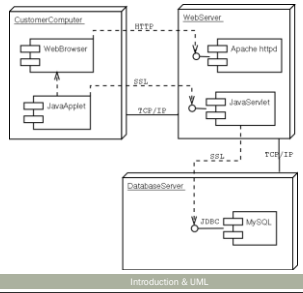
Package Diagrams

- Component diagrams illustrate the organizations and dependencies among software components in physical world.
- A component may be
  - A source code component
  - A run time components
  - An executable component

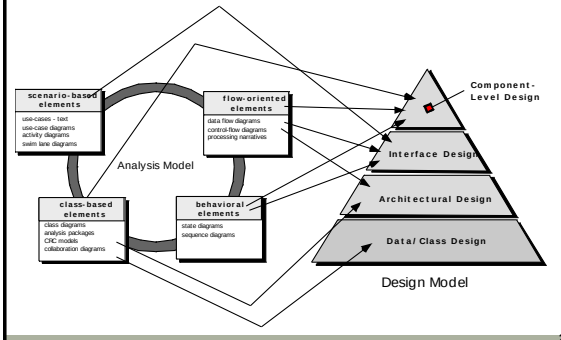


Deployment Diagrams

- Captures the distinct number of computers involved
- Shows the communication modes employed
- Component diagrams can be embedded into deployment diagrams effectively



Analysis Model ⟶ Design Model



1. Unified Modeling Language
2. Software Design Concepts
3. Structured Design
4. Object Oriented Design Principles
5. User Interface Design
6. Case Study: SafeHome

Software Design Concepts

8.2

Design Concepts (1)

- **Component:**
  - Any piece of software or hardware that has a clear role.
  - A component can be isolated, allowing you to replace it with a different component that has equivalent functionality.
  - Many components are designed to be reusable.
  - Conversely, others perform special-purpose functions.
- **Module:**
  - A component that is defined at the programming language level.
  - For example, functions are modules in C.
  - For example, methods, classes and packages are modules in Java.

Design Concepts (2)

- **Modularity:**
  - A complex system may be divided into simpler pieces called *modules*.
  - A system that is composed of modules is called *modular*.
  - When dealing with a module we can ignore details of other modules.
  - Use divide and conquer method for modularity.
  - For two modules m1 and m2, the effort relation is as follows:

$$E(m1) + E(m2) < E(m1+m2)$$

Design Concepts (3)

- **Abstraction:**
  - Abstraction is a means of achieving stepwise refinement by suppressing unnecessary details.
  - It is to conceptualize problem at a higher level.
  - Abstractions allow you to understand and concentrate on the essence of a subsystem without having to know unnecessary details.
  - The designer should keep the level of abstraction as high as possible.
- **Data abstraction (Abstract Data Type):**
  - A data type together with the operations performed on instantiations of that data type.
- **Procedural abstraction:**
  - The designer defines a procedure at a higher level.

Example: C definition without Data Abstraction

```
struct personel
{
  long int TCNum;
  char Ad[20], Soyad[20];
  int DTarihi_gun, DTarihi_ay, DTarihi_yil;
  int IGTarihi_gun, IGTarihi_ay, IGTarihi_yil;
} ;
```


Example: C definition with Data Abstraction

```
typedef struct
{
  int gun, ay, yil;
} tarih;

struct personel
{
  long int TCNum;
  char Ad[20], Soyad[20];
  tarih DogumTarihi;     //abstraction
  tarih IseGirisTarihi; //abstraction
} ;
```

Example:C program without Procedural Abstraction

```
#include <stdio.h>
#include <stdlib.h>
#define N 5
int main()
{
  int a[N] = {10,20,30,40,50};
  int b[N] = {15,25,35,45,55};
  int i;
  for (i=0; i < N; i++)  printf("%d \t", a[i]);
  printf("\n");
  for (i=0; i < N; i++)  printf("%d \t", b[i]);
  return 0;
}
```

31

Example:C program with Procedural Abstraction

```
#include <stdio.h>
#include <stdlib.h>
#define N 5

void yaz(int dizi[], int M) {
  int i;
  for (i=0; i < M; i++)
    printf("%d \t", dizi[i]);
  printf("\n");
}
int main() {
  int a[N] = {10,20,30,40,50};
  int b[N] = {15,25,35,45,55};
  yaz(a, N); //abstraction
  yaz(b, N); //abstraction
  return 0;
}
```

32

Layers of Software Design

1. Architectural Design
2. Modular Design
   1. Data Design
      - Transforms the information domain model created during the analysis into the data structures, file structures, database structures that will be required to implement software.
      - Data objects and relationships in ERD and the detailed data content depicted in the data dictionary provide the basis.
   2. Behavioral Design
      - Transforms structural elements of the program architecture into a procedural description of software components.
      - Information obtained from PDL (Algorithms / Flowcharts) serve as basis.
3. User Interface Design

33

1. Unified Modeling Language
2. Software Design Concepts
3. Structured Design
4. Object Oriented Design Principles
5. User Interface Design
6. Case Study: SafeHome

Structured Design Approach

8.3

Data-Oriented Design

- Basic principle
  - The structure of a product must conform to the structure of its data
- Three very similar methods
  - Michael Jackson [1975], Warnier [1976], Orr [1981]
- Data-oriented design
  - Has never been as popular as action-oriented design
  - With the rise of OOD, data-oriented design has largely fallen out of fashion

Operation-Oriented Design

- Data flow analysis
  - Use it with most specification methods (Structured Systems Analysis here)
- Key point: We have detailed action information from the DFD

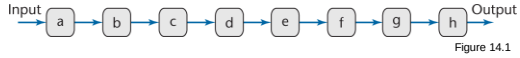


Figure 14.1

Data Flow Analysis

- Every product transforms input into output
- Determine
  - "Point of highest abstraction of input"
  - "Point of highest abstract of output"

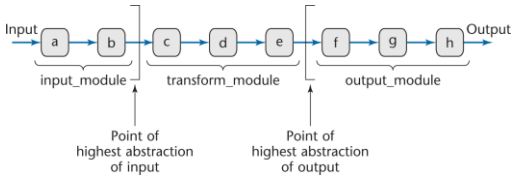


Figure 14.2

Data Flow Analysis (contd)

- Decompose the product into three modules
- Repeat stepwise until each module has high cohesion
  - Minor modifications may be needed to lower the coupling

Mini Case Study: Word Counting

- Example:

  Design a product which takes as input a file name, and returns the number of words in that file (like UNIX *wc*)

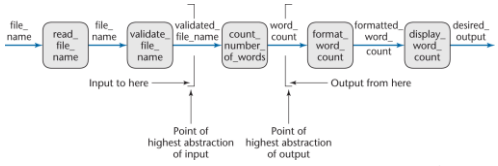


Figure 14.3

Mini Case Study: Word Counting

- First refinement

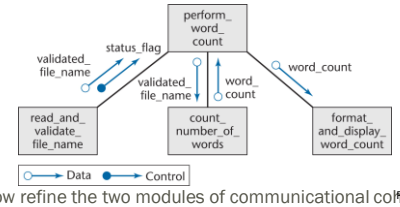


Figure 14.4

- Now refine the two modules of communicational cohesion

Mini Case Study: Word Counting

- Second refinement

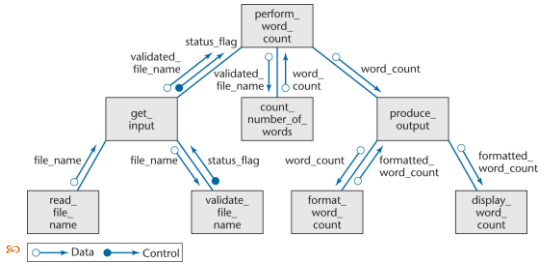


Figure 14.5

Word Counting: Detailed Design

- The architectural design is complete
  - So proceed to the detailed design
- Two formats for representing the detailed design:
  - Tabular
  - Pseudocode (PDL — program design language)

Detailed Design: Tabular Format

Module name	**count_number_of_words**
Module type	Function
Return type	**integer**
Input arguments	**validated_file_name : string**
Output arguments	None
Error messages	None
Files accessed	None
Files changed	None
Modules called	None
Narrative	This module determines whether **validated_file_name** is a text file, that is, divided into lines of characters. If so, the module returns the number of words in the text file; otherwise, the module returns **−1**.

Figure 14.6(c)

Detailed Design: PDL Format

```
void perform_word_count ( )
{
  String                validated_file_name;
  int                   word_count;

  if (get_input (validated_file_name) is null)
    print "error 1: file does not exist";
  else
  {
    set word_count equal to count_number_of_words (validated_file_name);
    if (word_count is equal to −1)
      print "error 2: file is not a text file";
    else
      produce_output (word_count);
  }
}

String get_input ( )
{
    String                file_name;

    file_name = read_file_name ( );
    if (validate_file_name (file_name) is true)
    {
      return file_name;
    }
    else
      return null;
}

void display_word_count (String formatted_word_count)
{
    print formatted_word_count, left justified;
}

String format_word_count (int word_count);
{
    return "File contains" word_count "words";
}
```

Figure 14.7

Data Flow Analysis Extensions

- In real-world products, there is
  - More than one input stream, and
  - More than one output stream

Data Flow Analysis Extensions

- Find the point of highest abstraction for each stream

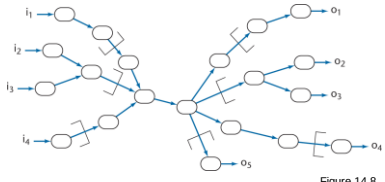


Figure 14.8

- Continue until each module Works on a single responsibility