

## - **Software Design**

Deriving a solution that satisfies software requirements.

### ■ Understand Problem

Look at the problem from different angles to discover the design requirements.

### ■ Identify one or more solutions

Evaluate possible solutions and choose the most appropriate one, depending on the designer's experience and available resource.

### ■ Describe the solution

Use graphical, formal or other descriptive notations to describe the components of the design.

## - **Design Phases**

### ■ Architectural Design

Identify sub-systems. → System Architectural

### ■ Abstract Specification

Specify sub-systems. → Software Specification

### ■ Interface Design

Describe sub-system interfaces.  
→ Interface Specification

### ■ Component Design

Decompose sub-system into component.  
→ Component Specification

### ■ Data Structure Design

Design data structures to hold problem data.  
→ Data structure Specification.

### ■ Algorithm Design

Design algorithms for problem functions.  
→ Algorithm Specification

## - **Computer Systems and Design**

Computer systems are usually composed of multiple and interactive modules. The goal of system design is to decode: What the modules are / What the modules should be / How the modules interact with one another.

## - **Modular Programming**

Split a whole system into small units. Units are different according to programming approach.

In procedural programming, units are functions. Procedures represent distinct logical as functions in a program.

In Object-Oriented units are objects. The system is decentralized and each object manages its own state. Objects may be instances of a class and communicate by exchanging methods.

## - **Evaluate Design Methods**

### ■ Modular Decomposability

A method that supports the decomposition of a big problem into smaller sub-problems, which can be solved independently.

### ■ Modular Reusability

A method that supports the reuse of existing modules to produce a new system.

### ■ Modular Understandability

A method in which modules are easily understandable.

### ■ Modular Connectivity

A method that allows a small change in problem specification by changing one or small number of modules.

### ■ Modular Protection

A method in which an abnormal condition affects only one or small number of modules.

## - **Good Design Method**

Linguistic modular units / Few interfaces / Small interfaces / Explicit interfaces / Information hiding

## - **Linguistic Modular Units**

A good programming language (or design language) should support linguistic modular units

In JAVA, modules can be methods and classes that are basic units in JAVA language. [GOOD]

But subroutines in BASIC are called by giving a line number where excursion is to proceed from. [BAD]

## - **Few Interfaces**

The overall number of communication channels between modules should be as small as possible. That is, every module should communicate with as few others as possible.

In the system with  $n$  modules, the ideal number of communication channels is  $n - 1$ .

There may be a minimum of  $n - 1$  and a maximum of  $(n(n - 1))/2$ .

## - **Small Interfaces (Loose Coupling)**

When any two modules communicate, they should exchange as little information as possible.

Loose Coupling can be achieved by system decentralization (such as OOP) and module communication via parameters or message passing. Shared variables lead to tight coupling.

In inheritance, an object class is coupled to its super class. Changes made to the attributes or operations in a super class propagate all sub-classes.

Declaring all instance variables as public. [BAD]

## - **Explicit Interfaces**

Inter-connection between two modules should be explicit. If a module is changed, all modules affected by this change should be identified.

두 모듈 간의 상호작용은 명시적이어야 한다. 모듈이 변경되면 변경으로 인해 영향을 받는 모든 모듈을 식별해야 한다.

## - **Information Hiding**

All information about a module should be private to the module.

- **Structured Design** analyze the system functional views.

- **Object-oriented Design** abstract objects and their properties and function.

- **Structure Design** is high-level design

Description of sub-systems and their interface.  
→ Structure chart (or diagram)

- **Detail Design** is low-level design

Description of modules, data structure, algorithm, and user interface  
→ Program specification, Layout design.

- **Architectural Design** is a design process for identifying. Establish the overall structure of a software system. → Software architecture.

- **Software Architecture** can be used in communication between stakeholders.

Therefore, software architecture should provide a clear and correct description of the system.

Illustrate major components or sub-systems and their interconnections. Explain what the graphical representation is.

## - **Types of Software Architecture**

Repository / Client-server / Hierarchical / Pipe Filter

## - Repository Architecture

Sub-systems can access to a central repository (database) and change data. They are independent and can communicate each other just using repository data.

When large amounts of data are to be shared, the repository architecture is most commonly used.

### ■ Pros.

Efficient way to share large amounts of data. Sub-systems don't need to be concerned with how data is produced and managed such as backup, security, etc.

### ■ Cons.

It is difficult and expensive to extend or evolve data. Data managements is centralized on repository.

## - Client-Server Architecture

Server(s) provides services to sub-systems called client and should control the number of clients that access to the server. Clients can obtain data from database managed by each server.

### ■ Pros.

Data can be distributed by multiple servers. Network system is effectively used. Easy to add new servers or upgrade existing servers.

### ■ Cons.

When several servers provide same services, all the server should maintain coherent (일관된) data. If servers have different data, clients may receive different data or services.

Redundant management in server such as backup and recovery.

## - Hierarchical Architecture

Each sub-system constructs a layer. A sub-system can communicate with sub-system of higher layer of lower layer. Ex) OSI 7 layer.

### ■ Pros.

It is easy to change or revise a sub-system.

### ■ Cons.

Inefficient communication between sub-systems.

## - Pipe Filter Architecture

A sub-system receives data, process, and transfer them to the connected sub-system.

Sub-systems are called filter, and connection is called pipe. Filters can be executed concurrently.

### ■ Pros.

Simple architecture, Easy to maintain the system.

Every filter has own service (operation), that is, filter can be regarded as a component (reusable).

### ■ Cons.

Incorrect output from a filter can affect the performance of the next filter.

All filters can be executed concurrently, but a certain filter may be a bottleneck.

- **Sub-system** is a system that has its own functions (services). Its operation is independent of the services provided by other sub-system.

- **Module** is a part of a sub-system, which provides explicit services to other components. It is not considered as a separate system.

- **System Structuring** identify communications between the sub-systems and decompose principal sub-systems.

- **Modular Decomposition** decompose a system or sub-system into modules.

## - How to Decompose Modules

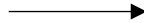

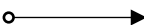
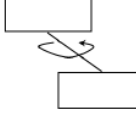
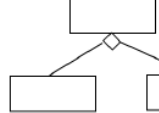
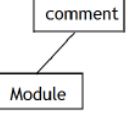
Make module cohesion as stronger as possible.

Make module coupling as weaker as possible.

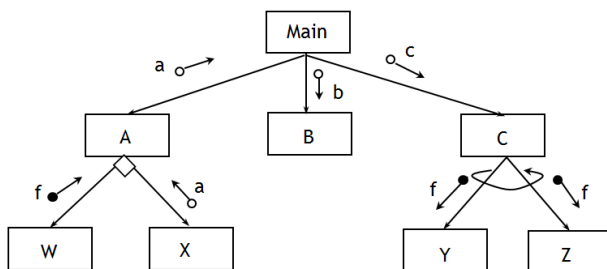
## - System Structure is defined by

Sub-systems or modules / Modular hierarchy / IO interface / Name and function of subsystem or modules.

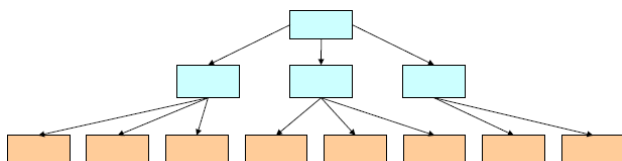
Notations

|   |                                      |
|---|--------------------------------------|
|  | Relation between two modules         |
|  | Flow of control flag (true or false) |
|  | Flow of data                         |
|  | Iterative call                       |
|  | Select one of two modules            |
|  | Make a comment                       |

Example



## - Efficient Structure



Make a balanced structure

Minimize depth and width of structure

## - Software Testing

Finds software defects to improve the software quality. It takes more than 50% of overall development cost for most of projects.

The result of software testing is directly related to the success in the software project.

## - Component Testing

Testing *individual* program components. Usually conducted by the component developer.

Testing methods and types derived from the developer's experience.

## - Integration Testing

Testing *groups* of components: a system or sub-system. Usually conducted by an independent testing team.

- **Test Data** is input data for testing system.

- **Test Cases** are various input elements to test the system. According to test cases, the predicted output can be different.

## - Equivalence Partitioning

Test data are partitioned into several groups. Investigate what kinds of outputs are generated from the partitioned data.

In general, test data consist of *valid data* and *invalid data* and they are partitioned into *valid data groups* and *invalid data groups*.

All data belonging to a group are assumed to be equivalent and expected to behave in an equivalent way for the system.

- **Black-box Testing** also called the functional testing.

To test a system, focusing on functions of the system or sub-system.

The program (or system) is considered as a 'black-box'. The test cases are based on the system specification.

Testing guide lines...

- Test the software using only a single value as input.
- Use sequences having different size as input
- Test the software so that the first, middle and last elements of the sequence are searched (Use each element as a key).

순서의 첫 번째, 중간 및 마지막 요소가 검색되도록 소프트웨어를 테스트.

- Use an element that is not included in the sequence as a key.

시퀀스에 포함되지 않은 요소를 키로 사용.

- Ex.) Search System

```

procedure Search (
    Key:    ELEM;
    T:      ELEM_ARRAY;
    Found:  in out BOOLEAN;
    L:      in out ELEM_INDEX);

pre-condition
    --the array has at least one element
    T'FIRST <= T'LAST

post-condition
    --the element is found and is
    referenced by L
    (Found and T(L) = Key)
    or
    --the element is not in the array
    (not Found and
    not exists i ( T'FIRST <= i <= T'LAST,
    T(i) = Key))

```

Search routine specification

| Array             | Element                    |
|-------------------|----------------------------|
| Single value      | In sequence                |
| Single value      | Not in sequence            |
| More than 1 value | First element in sequence  |
| More than 1 value | Last element in sequence   |
| More than 1 value | Middle element in sequence |
| More than 1 value | Not in sequence            |

Test cases

| Input Sequence ( T )               | Key(Key) | Output (Found, L) |
|------------------------------------|----------|-------------------|
| <u>17</u>                          | 17       | True, 1           |
| 17                                 | 0        | False, ?          |
| <u>17</u> , 29, 21, 23             | 17       | True, 1           |
| 41, 18, 9, 31, 30, 16, <u>45</u>   | 45       | True, 7           |
| 17, 18, 21, <u>23</u> , 29, 41, 38 | 23       | True 4            |
| 21, 23, 29, 33, 38                 | 25       | False, ?          |

Testing results (input, output)

- **White-box Testing** also called the structural test

Investigate whether each of all program statements or modules operate correctly or not. Available for short-size programs.

Derive test cases according to program structure.

■ Statements test

Execute each of all statements many times.

■ Decision test

Test a conditional statement for all conditions

■ Loop test

Test a loop structure when loop is not executed, executed only one time or executed more than one time.

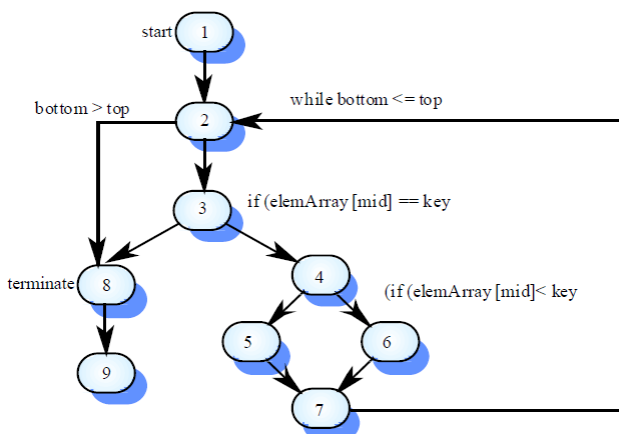
This testing can be conducted by software developers.

Testing guidelines...

- Test when a key element exists in array or not.
- Test when input array has a single value, an even number of values or odd number of values.

- **Path Testing** confirm whether each path of the program is executed at least once.

A kind of structural testing or white-box testing. Requires several test cases and a flow graph of the program.



Test case should be derived all possible paths

- **Integration Testing**

Tests complete systems or sub-systems rather than components. Test functions and performance of the system.

Integration testing should be block-box testing because it is based on a system specification.

It is difficult to find a specific sub-system or module that induces error. *Incremental Integration Testing* reduces this problem.

■ Top-down testing

Start with high-level and go down to low-level. Similar to incremental testing (easy to find a cause of errors).

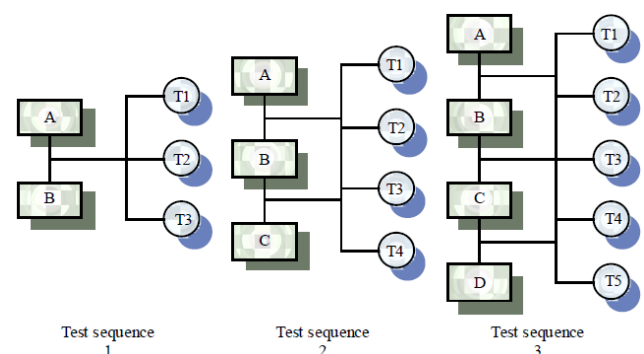
■ Bottom-up testing

Start with low-level and go up to high-level. Easy to detect errors because of a tendency that more errors occur in lower level systems.

In practice, most integration testing combines these two strategies.

- **Incremental Integration Testing**

In each test sequence, a sub-system is integrated into the system. It is possible to check which sub-system induces errors.



- **Acceptance Testing** is final testing before releasing the system. To show that the developed system is ready to be used in real environment.

*Alpha test* is conducted by selected users in dev's env.

*Beta test* is conducted by real users in user's env.

- **Software Maintenance** is the set of activities after software delivery. The activities evolve software systems.

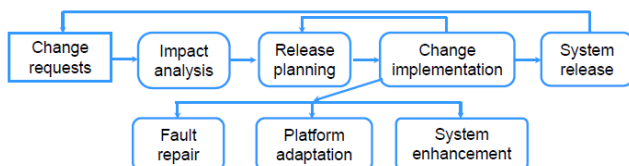
Software maintenance is a general process for software evolution (or software changes).

The change may be simple changes to correct coding errors, be more extensive changes to correct design errors, or be significant changes to correct specification error or reflect new requirements.

Three types of maintenance: Fault Repair / Software Adaption / Functionality Addition or Modification

- **Fault Repair** maintenance to repair software faults. It changing a system to correct faults.
- **Software Adaptation** maintenance to adapt software to a different operating environment.
- **Functionality Addition or Modification** maintenance to add or modify the system's functionality. It modifying the system to satisfy new requirements.

#### - Maintenance Process



#### - Why is Maintenance Difficult and Inefficient?

Lack of models or ignorance of available models.  
 Lack of documentation.  
 Lack of time to update existing documentation.  
 Low quality of original program.  
 Low quality of documents.  
 Lack of human resources.  
 Different programming styles.

#### - Example of Change Request Form

|  |                                   |
|--|-----------------------------------|
| <b>Project:</b> Proteus/PCL-Tools  | <b>Number:</b> 23/94              |
| <b>Change Requester:</b> I.Sommerville   | <b>Data:</b> 1/9/98               |
| <b>Requested Change:</b> When a component is selected from the structure, display the name of the file where it is stored.   |                                   |
| <b>Change Analyzer:</b> G.Dean   | <b>Analysis Date:</b> 10/9/98     |
| <b>Components Affected:</b> Display-icon.Select, Display-icon.Display  |                                   |
| <b>Associated Component:</b> File Table  |                                   |
| <b>Change Assessment:</b> Relatively simple to implement as a file name table is available. Requires the design and implementation of a display field. No changes to associated components are required. |                                   |
| <b>Change Priority:</b> Low  |                                   |
| <b>Change Implementation:</b>  |                                   |
| <b>Estimated Effort:</b>   |                                   |
| <b>Date to CCB:</b> 15/9/98  | <b>CCB Decision Date:</b> 1/11/98 |
| <b>Change Implementor:</b>   | <b>Date of Change:</b>            |
| <b>Date Submitted to QA:</b>   | <b>QA Decision:</b>               |
| <b>Date Submitted to CM:</b>   |                                   |
| <b>Comments</b>  |                                   |

## - Problems of Maintenance

### ■ Changing Priorities

When new requests arrive before finishing the current maintenance, the previous or current maintenance may be useless.

### ■ Inadequate Testing or Performance Measurement

Caused by lack of testing time or test cases. It can generate additional maintenance.

### ■ Incomplete System Documentation

May lead to undesirable (incorrect) maintenance.

### ■ Rapidly changing business environment

Hardware and software may become obsolete (out-of-date) and it may require significant change (correction) of the system.

Tightly coupled systems require changes whenever the environment is changed.

HW와 SW가 구식이 되어 시스템의 상당 부분 변경해야 할 수도 있음. 밀접하게 결합된 시스템은 환경이 변경 될 때마다 변경해 줘야함.

## - Maintenance Costs

Usually greater than development costs. Affected by both technical and non-technical factors.

Increases as software is maintained. Maintenance corrupts the software structure, so makes further maintenance more difficult.

유지관리로 인해 소프트웨어 구조가 손상되므로 유지관리가 더 어려워 짐.

## - Maintenance Prediction

To assess (평가하다) which parts of the system may cause problems and have high maintenance costs. To predict the number of changes to minimize the maintenance costs.

## - Maintenance Prediction Model

$$M = p + K \exp(c - d)$$

$M$  = total maintenance effort over entire lifecycle.

$p$  = productive efforts: analysis, design, implementation, test

$c$  = complexity due to lack of well-formed design and documentation.

$d$  = degree of familiarization with the system.

$K$  = empirically determined constant.

Cost of maintenance increases exponentially. It can be reduced by well-formed design and documentation. Giving the maintenance team chances to become familiar with the system.

## - Maintenance Cost Factors

### ■ Team Stability

Maintenance costs can be reduced more if the same staffs are involved for more time.

### ■ Responsibility of Developers

Costs may increase when developers of a system have no duties to maintain the system.

### ■ Staff Skills

Maintenance staff are often inexperienced and have limited domain knowledge.

### ■ Program Age and Structure

As programs age, their structure is degraded and they become difficult to be changed.



- **Unified Modeling Language (UML)** is a modeling language to express and visualize the design of a software or system.

It accepted as a standard by Object Management Group (OMG) and International Organization for Standardization (ISO).

Open standard, graphical notation for specifying, visualizing, constructing, and documenting software systems.

Can be used from general initial design to very specific detailed design across the entire software development lifecycle.

Useful for customers and developers to understand the product.

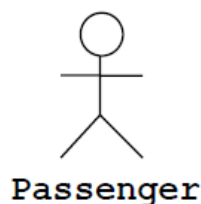
Independent of programming languages, but particularly useful for Object-oriented design.

Popular software Integrated Development Environment (IDE) supports for UML.

- **Use Case Diagram** is efficient to obtain user requirements by defining actors and their roles.

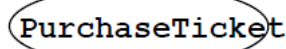
- **Actor** is an external entity which communicates with the other actor(s).

An actor has a *unique name* and an optional *description*. Actor can be either a *user* or a *system*.



- **Use Case** represents a functionality provided by the system as an event flow.

A use case has: Unique name / Participating actors / Entry conditions / Flow of events / Exit conditions / Special requirements



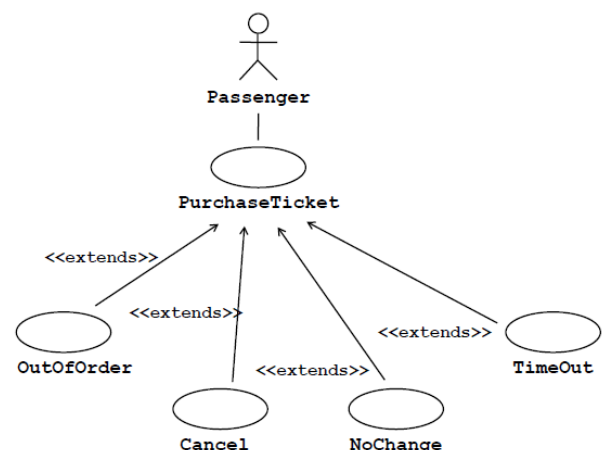
- **Ex.) Use Case Diagram**

|  |                                       |
|--|---------------------------------------|
| <b>Name:</b> Purchase ticket                               | <b>Participating actor:</b> Passenger |
| <b>Entry condition:</b>                                    |                                       |
| - Passenger standing in front of ticket distributor        |                                       |
| - Passenger has sufficient money to purchase a ticket      |                                       |
| <b>Exit condition:</b>                                     |                                       |
| - Passenger has a ticket                                   |                                       |
| <b>Event flow:</b>   |                                       |
| 1. Passenger selects the number of place to travel.        |                                       |
| 2. Distributor displays the price.                         |                                       |
| 3. Passenger inserts money same as or more than the price. |                                       |
| 4. Distributor returns change.                             |                                       |
| 5. Distributor issues ticket.                              |                                       |

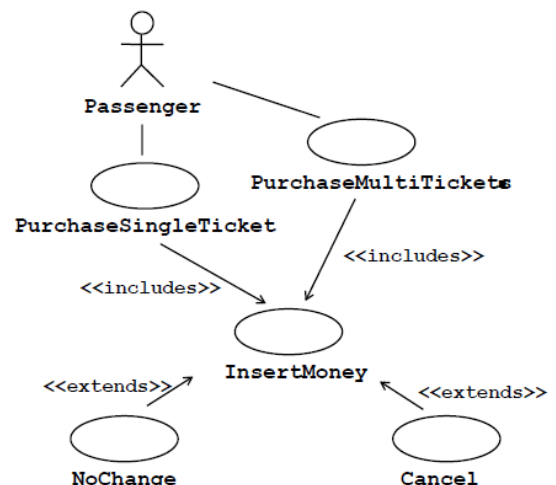
Exceptional cases are missing.

Take a look <<extends>>.

- **The <<extends>> Relationship** represent exceptional cases. The direction of <<extends>> relationship is to the extended *use case*.



- **The <<include>> Relationship** is used to indicate that a particular use case must include another use case to perform its function. The direction of a <<include>> relationship is to the included *use case*.



## - Usefulness of Use Case Diagram

### ■ Determine Requirements

New use cases often generate new requirements.

### ■ Communicate with Customers

Simplicity of notation of use case diagrams provides a good way for developers to communicate with customers.

### ■ Generate Test Cases

Easy to collect and reflect event scenarios. Possible to generate various test cases for each of scenarios.

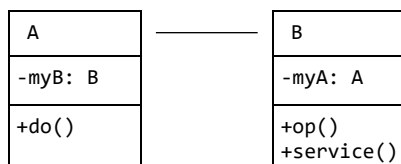
## - Class Diagram gives an overview of a system by representing classes and the relationship between them.

Describe attributes and operations of each class. Provide a good way to describe the overall architecture of system components.

|            |  |                              |
|------------|--|------------------------------|
| Name       | Account_Name                             | '+' denotes <i>Public</i>    |
| Attributes | -Custom_Name<br>-Balance                 | '#' denotes <i>Protected</i> |
| Operations | +Deposit()<br>+Withdraw()<br>+Transfer() | '-' denotes <i>Private</i>   |

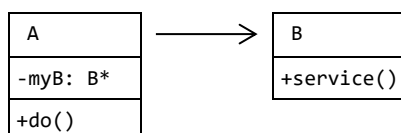
## - Association is a relationship between classes. Class must know about the other associated class to do its work.

### ■ Bi-directional Association



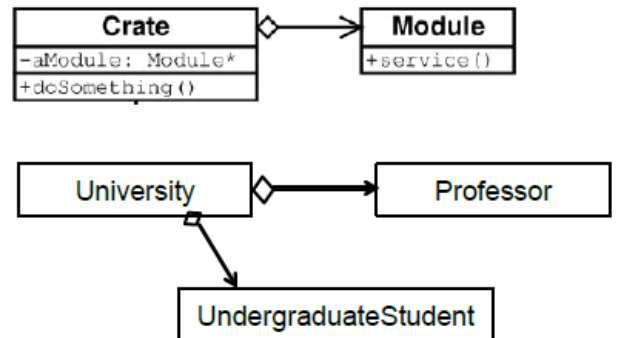
Both entities know each other

### ■ Uni-directional Association

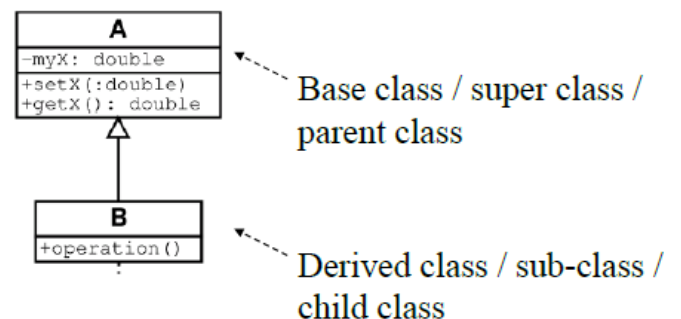


A knows B, but B knows nothing about A.

## - Aggregation is an association in which a class belongs to the other class. Used to describe a collection-member relationship.



## - Generalization is an inheritance link indicating that a class is a superclass of the other class.



## - Multiplicity Notation

| Multiplicities | Meaning  |
|----------------|--|
| 0..1           | The class has <u>zero or one</u> instance.               |
| 0..* or *      | The number of instance can be <u>zero or limitless</u> . |
| 1              | The class has <u>exactly one</u> instance.               |
| 1..*           | The class has <u>at least one</u> instance.              |

