Software Design

Dr. ZhiQuan (George) Zhou
Associate Professor

- **Design** refers to both an **activity** and the **result** of the activity

What is design?

What is design?

- Provides structure to any artifact
- Decomposes system into parts, assigns responsibilities, ensures that parts fit together to achieve a global goal

Two meanings of "design" activity in our context

- Activity that acts as a bridge between requirements and the implementation of the software
- Activity that gives a structure to the artifact
  - e.g., a requirements specification document must be *designed*
    - must be given a structure that makes it easy to understand and evolve

The sw design activity

- Defined as system decomposition into modules
- Produces a Software Design Document
  - describes system decomposition into modules
- Often a *software architecture* is produced prior to a software design
  - The principles in developing an architecture and a design are similar.

Two important goals

- Design for change (Parnas)
  - designers tend to concentrate on current needs
- Product families (Parnas)
  - think of the current system under design as a member of a program family

Product families

- Different versions of the same system
  - e.g. a family of mobile phones
    - members of the family may differ in network standards, end-user interaction languages, …
  - e.g. a facility reservation system
    - for hotels: reserve rooms, restaurant, conference space, equipment, …
    - for a university
      - many functionalities are similar, some are different (e.g., facilities may be free of charge or not)

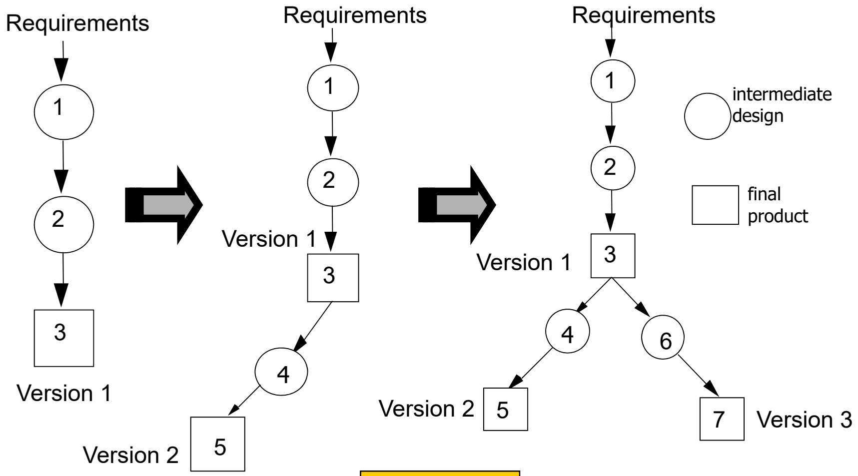
Design goal for family

- Design the whole family as one system, not each individual member of the family separately

Sequential completion: the wrong way

- Design first member of product family
- Modify existing software to get next member products

Sequential completion: a graphical view



Problems?

Product families: Sequential completion

- Biased by the design decisions in version 1
- No effort was made to isolate
  - what is common to all versions
  - what is common to smaller and smaller subsets

Product families: Sequential completion (cont.)

- New versions: by modifying code because intermediate design not documented
  - A modification to a part may adversely affect other parts
  - We may inadvertently make design decisions discarded before, but never documented.

How to do better

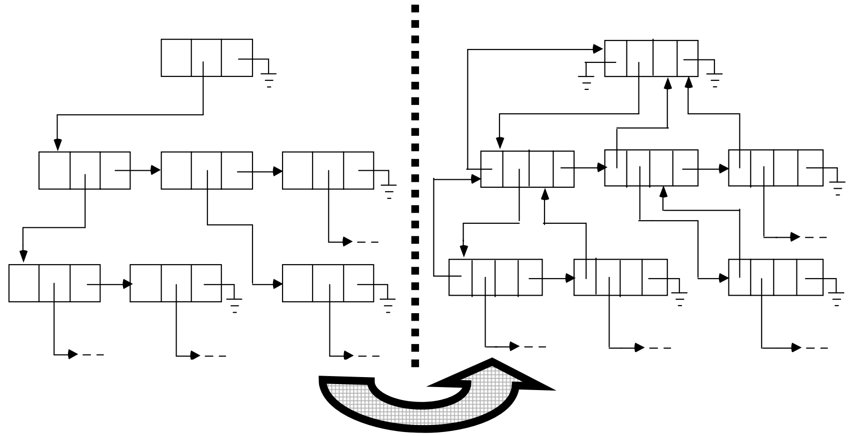
- Anticipate definition of all family members
- Identify what is common to all family members, delay decisions that differentiate among different members

Designing for change

Sample likely changes? (1)

- **Algorithms**
  - e.g., replace inefficient sorting algorithm with a more efficient one
- Change of **data representation**
  - ≈17% of maintenance costs attributed to data representation changes (Lientz and Swanson, 1980)

Example



Sample likely changes? (2)

- Change of underlying **abstract machine**
  - new release of operating system
  - new optimizing compiler
  - new version of DBMS
  - …
- Change of **peripheral devices**

Sample likely changes? (3)

- Change of **"social"** environment
  - new tax regime
  - EURO vs national currency in EU
- Change due to **development process** (transform prototype into product)

Sample likely changes? (4)

- Change of **user requirements**

Module

- A well-defined component of a software system
- Provides a set of services or resources to other modules
- E.g.
  - a collection of routines
  - a collection of data
  - a collection of type definitions
  - a mixture of all of these

Questions

- How to define the **structure** of a modular system?
- What are the desirable properties of that **structure**?

Modules and relations

- Let S be a set of modules
  $S = \{M_1, M_2, \ldots, M_n\}$
- A binary relation r on S is a subset of
  $S \times S$ (Cartesian product)
- If M_i and M_j are in S, $<M_i, M_j> \in r$ can be written as $M_i \; r \; M_j$

Relations

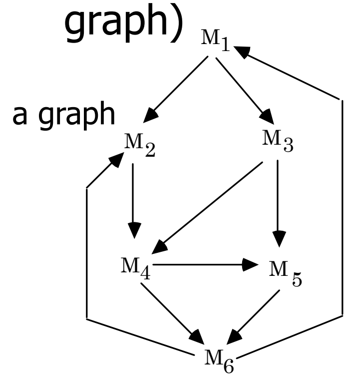
- The *transitive closure* of a relation r on S is again a relation on S, written r^+
  $M_i \; r^+ \; M_j$ <u>iff</u>
  $M_i \; r \; M_j$ or $\exists \; M_k$ in S s.t. $M_i \; r \; M_k$ and $M_k \; r^+ \; M_j$

(We assume our relations to be irreflexive, that is, $M_i \; r \; M_i$ cannot hold for any module M_i in S.)

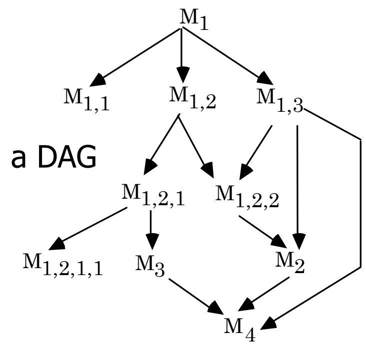
- r is a **hierarchy** iff there are no two elements M_i, M_j s.t. $M_i \; r^+ \; M_j \wedge M_j \; r^+ \; M_i$

Relations

- A relation can be represented as a directed graph
- A hierarchy is a DAG (directed acyclic graph)



a)



b)

The USES relation

- A uses B (A and B are distinct)
  - A requires the presence of B
    - Because B provides the resources that A needs to accomplish its task.
  - A can access the services exported by B through its *interface*
  - Example:
    - A calls a routine contained in B
    - A uses a type defined in B

The USES relation

- A uses B (cont.)
  - A is a client of B; B is a server
    - Don't mix with the "client-server architecture"!

Desirable property

Discussion:

- USES should / shouldn't be a hierarchy?

Desirable property

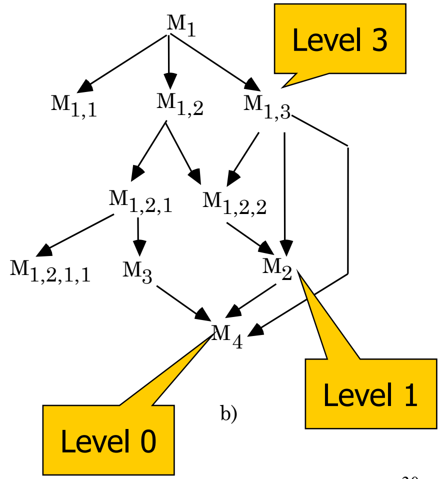
- USES should be a hierarchy
- Hierarchy makes software easier to **understand**
  - we can proceed from **leaf** nodes (who do not use others) upwards
- They make software easier to **build**
- They make software easier to **test**

Hierarchy

- Organizes the modular structure through *levels of abstraction*
- Each level defines an *abstract (virtual) machine* for the next level
  - *level* can be defined precisely
    - M_i has level 0 if no M_j exists s.t. M_i r M_j
    - let k be the maximum level of all nodes M_j s.t. M_i r M_j. Then M_i has level k+1

Hierarchy

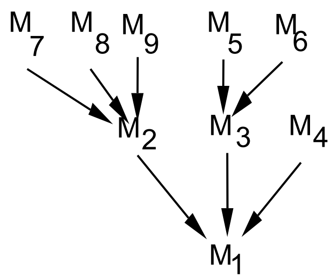
– E.g.



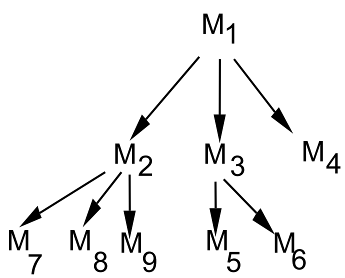
IS_COMPONENT_OF

- Used to describe a higher level module as constituted by a number of lower level modules
- A IS_COMPONENT_OF B
  - B consists of several modules, of which one is A
- B COMPRISES A
- $M_{S,i}=\{M_k|M_k \in S \wedge M_k$ IS_COMPONENT_OF $M_i\}$ we say that $M_{S,i}$ IMPLEMENTS M_i

A graphical view



(IS_COMPONENT_OF)



(COMPRISES)

They are a hierarchy

Product families revisited

- Careful recording of (hierarchical) USES relation and IS_COMPONENT_OF supports design of program families

Product families revisited

- Case study
  - Suppose you are designing a system S that you decompose into the set of modules M_1, M_2, ..., M_i, with some USES relation on it. Then, suppose you turn to the design of any of such modules, say, M_k, 1<=k<=i.

Product families revisited

- Case study (cont.)
  - At this point, you may realize that any design decision you take will separate one subset of family members from others; for example, M_k is an output module, and its design may need to discriminate between textual output and graphical output, to be dealt with by two different family members.

Product families revisited

- Case study (cont.)
  - Suppose you make the decision to follow one of the design options (e.g., the graphical output), which leads you to decompose M_k into $M_{k,1}$, $M_{k,2}$,..., $M_{k,x}$, with some USES relation defined on this set.

Product families revisited

- Case study (cont.)
  - You should record these design decisions carefully, so that future changes will be made reliably.
  - Suppose that at some later time a different member of the family needs to be designed (e.g., the system that provides support for textual output).

Question: What should we do?

Product families revisited

- Case study (cont.)

  Suppose that at some later time a different member of the family needs to be designed (e.g., the system that provides support for textual output).

  - You should never allow yourself to directly modify the code in order to meet the new requriements.

Product families revisited

- Case study (cont.)

  Suppose that at some later time a different member of the family needs to be designed (e.g., the system that provides support for textual output).

  - Rather, the recorded documents of the structure of the modules should force you to resume the design from the decomposition of module M_k, so that you may provide a different implementation in terms of lower level components.

Product families revisited

- Case study (cont.)

  Suppose that at some later time a different member of the family needs to be designed (e.g., the system that provides support for textual output).

  - Note, however, that the rest of the system will remain untouched, that is, modules $M_1, \ldots, M_{k-1}$, and $M_{k+1}, \ldots, M_i$ will not be affected by the design of the new family member.

Interface vs. implementation (1)

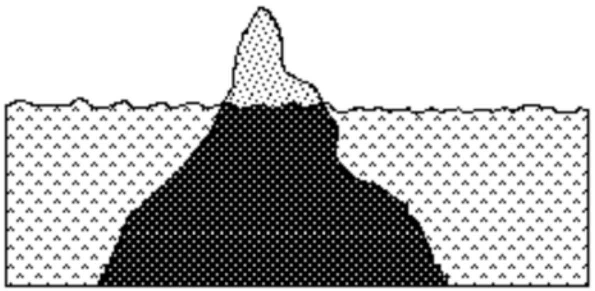
- To understand the nature of USES, we need to know what a used module *exports* through its **interface**
- **Implementation** is *hidden* to clients

Interface vs. implementation (2)

- Clear distinction between **interface** and **implementation** is a key design principle
- Supports separation of concerns
  - clients care about resources exported from servers
  - servers care about implementation

Interface vs. implementation (3)

interface is like the tip of the iceberg



Information hiding

- Basis for design (i.e. module decomposition)
- Implementation secrets are hidden to clients
- They can be changed freely if _____

the change does not affect the interface.

Information hiding

- **Golden design principle**
  - ***INFORMATION HIDING***
    - *Try to encapsulate changeable design decisions as secrets within module implementations*

Interface design

- Interface should not reveal what we expect may **change** later
- It should not reveal **unnecessary** details

Case study on interface design

We are designing an **interpreter** for a simple programming language operating on integers. We provide a **symbol-table module** that stores information about the variables of a program.

- The symbol table exports a procedure **GET** that accepts as input the symbolic name of a variable and returns the value of the variable. Similarly, the procedure **PUT** is used to store a new value for a given variable. When a new variable declaration is encountered, a new entry is created in the symbol table by calling the procedure **CREATE**, passing it the name of the variable and its size.
- The purpose of the interface we are designing is to hide the physical structure of the table from the clients of the module. To warn clients when they try to read/write a variable that does not exist, the procedures GET and PUT return a parameter, POS: a pointer to the variable stored in the table if such variable exists, or it is the null pointer if the variable does not exist.

Please comment on the design

Prototyping

- Once an interface is defined, implementation can be done
  - first quickly but inefficiently
  - then progressively turned into the final version

Prototyping

Discussion:

- If we design stable interfaces among modules, then modules may evolve independently from the prototype implementation to their final version.

Yes/No? Why?

- Case study on prototyping
  - We are developing a completely new type of search engine, a revolutionary product to the marketplace.
  - The greatest new feature is the query language: natural language and pictorial interaction
  - Before developing the new system, we wish to assess the human-computer interaction

Question: How do you do it?

- Case study on prototyping (cont.)
  - Thus, we decide to implement the user interface, but delay the implementation of the "real" DB management system (e.g., physical file structures, algorithms for retrieval, recovery procedures)
  - What we will implement is a *prototype* of the appliation that only deal with a limited amount of information, because all data will be kept in the main memory using arrays.
  - Potential users will be asked to play with the system and give feedback regarding its usability
  - They will be warned that the performance of the prototype have nothing to do with the final system.
  - Two modules may evolve indenpendently if the **module interface** is carefully designed.

Design notations

- Notations allow designs to be described precisely
- They can be textual or graphic
- We illustrate two sample notations
  - TDN (Textual Design Notation)
  - GDN (Graphical Design Notation)
- The Unified Modeling Language (UML) is being promoted as a universal standard for OO design.

TDN & GDN

- Illustrate how a notation may help in documenting design
- Illustrate what a generic notation may look like
- Are representative of many proposed notations

An example

module X
uses Y, Z
exports **var** A : integer;
    type B : **array** (1. .10) **of** real;
    procedure C (D: **in out** B; E: **in** integer; F: **in** real);
    Here is an optional natural-language description of what A, B, and C actually are, along with possible constraints or properties that clients need to know; for example, we might specify that objects of type B sent to procedure C should be initialized by the client and should never contain all zeroes.
implementation
    If needed, here are general comments about the rationale of the modularization, hints on the implementation, etc.
    is composed of R, T
end X

Example (cont.)

module R
uses Y
exports **var** K : **record** . . . **end;**
    type B : **array** (1. .10) **of** real;
    procedure C (D: **in out** B; E: **in** integer; F: **in** real);
implementation
    .
    .
    .
end R

module T
uses Y, Z, R
exports **var** A : integer;
implementation
    .
    .
    .
end T

Benefits

- Notation helps describe a design precisely
- Design can be assessed for consistency
  - having defined module X, modules R and T must be defined eventually
    - if not → *incompleteness*
  - R, T replace X
    - → either one or both must use Y, Z

Example: a compiler

module COMPILER
exports procedure MINI (PROG: in file of char;
CODE: out file of char);
MINI is called to compile the program stored in PROG and produce the object code in file CODE
implementation
A conventional compiler implementation.
ANALYZER performs both lexical and syntactic analysis and produces an abstract tree, as well as entries in the symbol table; CODE_GENERATOR generates code starting from the abstract tree and information stored in the symbol table. MAIN acts as a job coordinator.
is composed of ANALYZER, SYMBOL_TABLE,
ABSTRACT_TREE_HANDLER, CODE_GENERATOR, MAIN
end COMPILER

Other modules

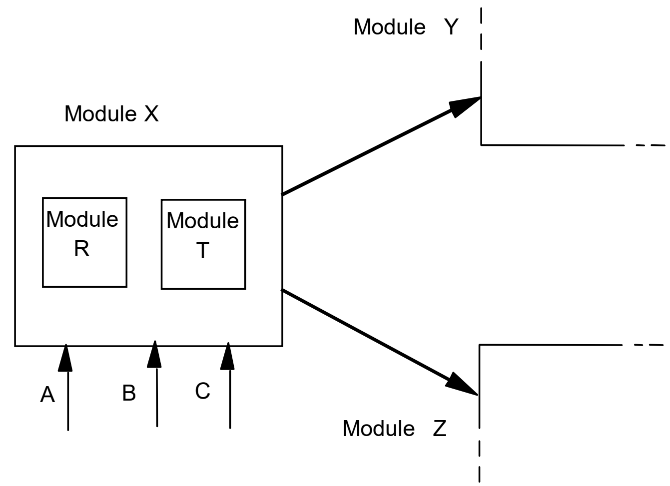
module MAIN
uses ANALYZER, CODE_GENERATOR
exports procedure MINI (PROG: in file of char;
CODE: out file of char);
…
end MAIN

module ANALYZER
uses SYMBOL_TABLE, ABSTRACT_TREE_HANDLER
exports procedure ANALYZE (SOURCE: in file of char);
SOURCE is analyzed; an abstract tree is produced by using the services provided by the ABSTRACT_TREE_HANDLER, *and recognized entities, with their attributes, are stored in the symbol table.*
...
end ANALYZER

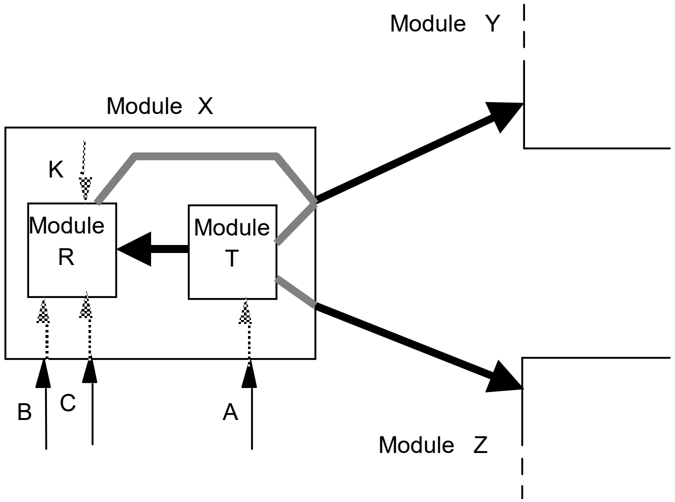
Other modules

module CODE_GENERATOR
uses SYMBOL_TABLE, ABSTRACT_TREE_HANDLER
exports procedure CODE (OBJECT: out file of char);
The abstract tree is traversed by using the operations exported by the ABSTRACT_TREE_HANDLER and accessing the information stored in the symbol table in order to generate code in the output file.
…
end CODE_GENERATOR

GDN description of module X



X's decomposition



Categories of modules

- **Functional** modules
  - traditional form of modularization
  - provide a procedural abstraction
  - encapsulate an algorithm
    - e.g. sorting module, fast Fourier transform module, …

Categories of modules (cont.)

- **Libraries**
  - a group of related procedural abstractions
    - e.g., mathematical libraries
      - implemented by routines of programming languages
- Common pools of **data**
  - data shared by different modules
    - e.g., configuration constants

Categories of modules (cont.)

- Abstract objects
  - Objects manipulated via interface functions
  - Data structure hidden to clients
    - Provides a ***state*** – different from functions in a **library**
- **Abstract data types**
  - Many instances of abstract objects may be generated
  - Can be implemented directly by a **class**

Abstract objects: an example

- A calculator of expressions expressed in Polish postfix form

  a*(b+c) → abc+*

- a module implements a stack where the values of operands are pushed until an operator is encountered in the expression

(assume only binary operators)

Example (cont.)

Interface of the abstract object STACK

exports
procedure PUSH (VAL: in integer);
procedure POP_2 (VAL1, VAL2: out integer);

Design assessment

- Question: How does the design anticipate changes?

Design assessment

- Question: How does the design anticipate changes?

  In type of expressions to be evaluated
  - e.g., unary operators?

Abstract data types (ADTs)

- A stack ADT

indicates that details of the data structure are hidden to clients

```
module  STACK_HANDLER
exports
        type  STACK = ?;
        This is an abstract data-type module; the data structure
        is a secret hidden in the implementation part.
        procedure   PUSH (S: in out  STACK ; VAL: in  integer);
        procedure   POP (S: in out  STACK ; VAL: out  integer);
        function  EMPTY (S: in  STACK) : BOOLEAN;
        .
        .
end  STACK_HANDLER
```


Specific techniques for design for change

- Use of **configuration constants**
  - constant values -> symbolic constants
    - e.g., #define in C

      #define MaxSpeed 5600;

Specific techniques for design for change (cont.)

- **Conditional compilation**

```
 ...source fragment common to all versions...
# ifdef hardware-1
...source fragment for hardware 1 ...
# endif
#ifdef hardware-2
...source fragment for hardware 2 ...
# endif
```


Object-oriented design

- One kind of module, ADT, called *class*
- A class **exports operations** (procedures) to manipulate instance objects
  - often called *methods*

A further relation: **inheritance**

- ADTs may be organized in a **hierarchy**
- Class B may specialize class A
  - B inherits from A

  conversely, A generalizes B
- A is a superclass of B
- B is a subclass of A

An example

```
class EMPLOYEE
exports
        function FIRST_NAME(): string_of_char;
        function LAST_NAME(): string_of_char;
        function AGE(): natural;
        function WHERE(): SITE;
        function SALARY: MONEY;
        procedure HIRE (FIRST_N: string_of_char;
                        LAST_N: string_of_char;
                        INIT_SALARY: MONEY);
        Initializes a new EMPLOYEE, assigning a new identifier.
        procedure FIRE();
        procedure ASSIGN (S: SITE);
        An employee cannot be assigned to a SITE if already assigned to it (i.e., WHERE
        must be different from S). It is the client's responsibility to  ensure this. The effect is to
        delete the employee from those in WHERE, add the employee to those in S, generate
        a new id card with security code to access the site overnight, and update WHERE.
end EMPLOYEE
```



```
class ADMINISTRATIVE_STAFF inherits EMPLOYEE
exports
        procedure DO_THIS (F: FOLDER);
        This is an additional operation that is specific to
        administrators; other operations may also be added.
end ADMINISTRATIVE_STAFF

class TECHNICAL_STAFF inherits EMPLOYEE
exports
        function GET_SKILL(): SKILL;
        procedure DEF_SKILL (SK: SKILL);
        These are additional operations that are specific to
        technicians; other operations may also be added.
end TECHNICAL_STAFF
```

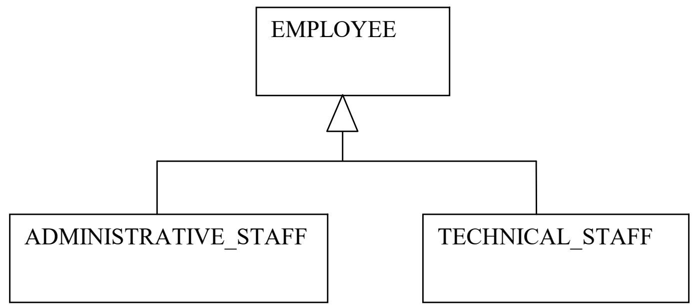

Inheritance

- A way of building software incrementally
- A subclass defines a subtype

How can inheritance be represented?

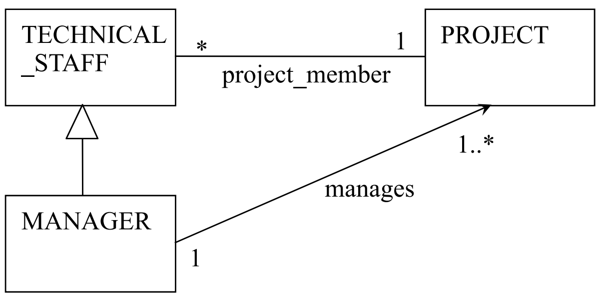
- UML (Unified Modeling Language) is a widely adopted standard notation for representing OO designs
- We introduce the UML class diagram
  - classes are described by boxes

UML representation of inheritance



UML associations

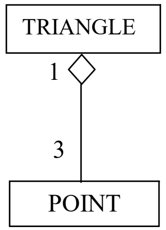
- Associations are relations that the implementation is required to support
- Can have multiplicity constraints



Aggregation

- Defines a PART_OF relation

Differs from IS_COMPOSED_OF
Here TRANGLE has its own methods
It implicitly uses POINT to define its data attributes



More on UML

- Representation of IS_COMPONENT_OF via the *package* notation