CHAPTER 20

PATTERNS AND UML

比上課更重要的

HTTPS://WWW.YOUTUBE.COM/WATCH?V=NWQJDLPF6UK&FEATURE=YOUTU.BE

DESIGN PATTERN

- A GENERAL REUSABLE SOLUTION TO A COMMONLY OCCURRING PROBLEM WITHIN A GIVEN CONTEXT
- A DESCRIPTION OR TEMPLATE FOR HOW TO SOLVE A PROBLEM
  - CAN BE USED IN MANY DIFFERENT SITUATIONS
- FORMALIZED BEST PRACTICES THAT THE PROGRAMMER CAN USE TO SOLVE COMMON PROBLEMS

TYPES OF DESIGN PATTERNS

- ALGORITHM STRATEGY PATTERNS
  - ADDRESSING CONCERNS RELATED TO HIGH-LEVEL STRATEGIES DESCRIBING HOW TO EXPLOIT APPLICATION CHARACTERISTICS ON A COMPUTING PLATFORM
- COMPUTATIONAL DESIGN PATTERNS
  - ADDRESSING CONCERNS RELATED TO KEY COMPUTATION IDENTIFICATION
- EXECUTION PATTERNS
  - THAT ADDRESS ISSUES RELATED TO LOWER-LEVEL SUPPORT OF APPLICATION EXECUTION
    - INCLUDING STRATEGIES FOR EXECUTING STREAMS OF TASKS AND FOR THE DEFINITION OF BUILDING BLOCKS TO SUPPORT TASK SYNCHRONIZATION

TYPES OF DESIGN PATTERNS

- IMPLEMENTATION STRATEGY PATTERNS
  - ADDRESSING CONCERNS RELATED TO IMPLEMENTING SOURCE CODE TO SUPPORT PROGRAM ORGANIZATION, AND THE COMMON DATA STRUCTURES SPECIFIC TO PARALLEL PROGRAMMING
- STRUCTURAL DESIGN PATTERNS
  - ADDRESSING CONCERNS RELATED TO GLOBAL STRUCTURES OF APPLICATIONS BEING DEVELOPED

CLASSIFICATION

- DESIGN PATTERNS CATEGORIES:
  CREATIONAL PATTERNS,
  STRUCTURAL PATTERNS, AND **BEHAVIORAL** PATTERNS
  - USING THE CONCEPTS OF DELEGATION, AGGREGATION, AND CONSULTATION.
  - FURTHER BACKGROUND ON OBJECT-ORIENTED DESIGN, SEE COUPLING AND COHESION, INHERITANCE, INTERFACE, AND POLYMORPHISM
- ANOTHER CLASSIFICATION - ARCHITECTURAL DESIGN PATTERN
  - APPLIED AT THE ARCHITECTURE LEVEL OF THE SOFTWARE
    - SUCH AS THE **MODEL–VIEW–CONTROLLER** (**MVC**) PATTERN

https://en.wikipedia.org/wiki/Design_Patterns

DESIGN PATTERN

- ADVANTAGES
  - ALWAYS CORRECT
  - USABILITY ONLY DEPENDS ON THE PROBLEM
  - SPEEDS UP THE DEVELOPMENT PROCESS OF SOFTWARE
  - MAKES COMMUNICATION BETWEEN DEVELOPERS EASIER

DESIGN PATTERN (CONTD.)

- DISADVANTAGES
  - MAY DECREASE UNDERSTANDABILITY OF THE CODE AND DESIGN
  - DANGER OF UNDERSTANDING DESIGN PATTERNS AS AN ALL-ROUND SOLUTION
  - RISK OF HIGHER MEMORY CONSUMPTION DUE TO GENERALIZED FORMAT

DIFFERENT TYPES OF DESIGN PATTERNS

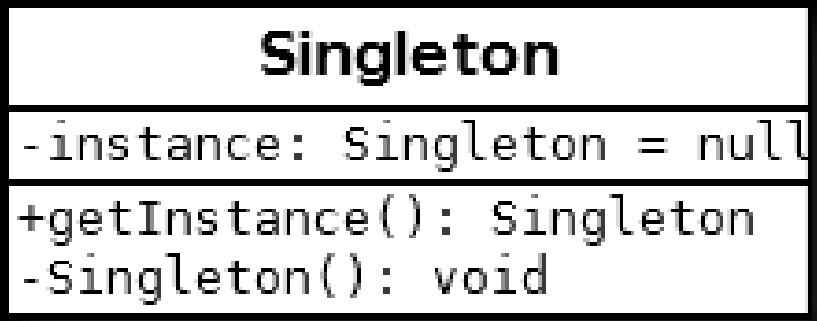
- **1. CREATIONAL**
  - - SINGLETON
  - - ABSTRACT FACTORY
  - …
- **2. STRUCTURAL**
  - - ADAPTER
  - - COMPOSITE
  - …
- **3. BEHAVIORAL**
  - - OBSERVER
  - - VISITOR
  - …

https://sourcemaking.com/design_patterns

THE SINGLETON PATTERN

- TYPICAL PROBLEM(S)
  - THERE SHALL BE ONLY ONE OBJECT OF A CLASS DURING THE EXECUTION OF A PROGRAM
  - I NEED A GLOBAL INSTANCE OF MY OBJECT
- EXAMPLES
  - A CENTRAL OBJECT FOR PRODUCING OUTPUT TO A FILE
  - JOBS FOR A PRINTER THAT A WRITTEN TO A SINGLE BUFFER
  - ACCESSING THE GPU IN VIDEO GAMES
  - DATABASE CONNECTIONS

IMPLEMENTATION OF SINGLETONS



Intro2UML

Check list

- ✓ Define a **private** **static** attribute in the "single instance" class.
- ✓ Define a **public** **static** accessor function in the class.
- ✓ Do "lazy initialization" (creation on first use) in the accessor function.
- ✓ Define all constructors to be protected or **private**.
- ✓ Clients may *only use the accessor function to manipulate* the Singleton.

```
1)  class GlobalClass
2)  {
3)      int m_value;
4)    public:
5)      GlobalClass(int v = 0)
6)      {       m_value = v;    }
7)      int get_value()
8)      {       return m_value;    }
9)      void set_value(int v)
10)     {       m_value = v;    }
11) };

12) // Default initialization
13) GlobalClass *global_ptr = 0;

14) void foo(void)
15) {
16)   // Initialization on first use
17)   if (!global_ptr)
            global_ptr = new GlobalClass;
18)   global_ptr->set_value(1);
19)   cout << "foo: global_ptr is " <<
        global_ptr->get_value() << '\n';
20) }

21) void bar(void)
22) {
23)   if (!global_ptr)
24)     global_ptr = new GlobalClass;
25)   global_ptr->set_value(2);
26)   cout << "bar: global_ptr is "
    <<global_ptr->get_value() <<'\n';
27) }

28) int main()
29) {
30)   if (!global_ptr)
31)     global_ptr = new GlobalClass;
32)   cout << "main: global_ptr is "
    << global_ptr->get_value() <<
    '\n';
33)   foo();
34)   bar();
35) }
```

```
main: global_ptr is 0
foo: global_ptr is 1
bar: global_ptr is 2
```

```
1)  class GlobalClass
2)  {
3)      int m_value;
4)      static GlobalClass *s_instance;
5)      GlobalClass(int v = 0)
6)      {       m_value = v;    }
7)    public:
8)      int get_value()
9)      {       return m_value;    }
10)     void set_value(int v)
11)     {       m_value = v;    }
12)     static GlobalClass *instance()
13)     {
14)         if (!s_instance)
15)           s_instance = new GlobalClass;
16)         return s_instance;
17)     }
18) };
```

private

```
// Allocating and initializing GlobalClass's
// static data member.  The pointer is being
// allocated - not the object inself.
GlobalClass *GlobalClass::s_instance = 0;

void foo(void)
{
  GlobalClass::instance()->set_value(1);
  cout << "foo: global_ptr is " <<
       GlobalClass::instance()->get_value() << '\n';
}

void bar(void)
{
  GlobalClass::instance()->set_value(2);
  cout << "bar: global_ptr is " <<
       GlobalClass::instance()->get_value() << '\n';
}

int main()
{
  cout << "main: global_ptr is " <<
       GlobalClass::instance()->get_value()
       << '\n';
  foo();
  bar();
}
```

```
main: global_ptr is 0
foo: global_ptr is 1
bar: global_ptr is 2
```

PRACTICE "SINGLETON"

```
1)  class Singleton
2)  {
3)  private:
4)      static Singleton *single;
5)      Singleton()   {//private constructor
6)          std::cout<< "calling constructor\n";
7)      }
8)  public:
9)      static Singleton* getInstance();
10)     void method();
11)     ~Singleton()
12)     {  //      ...    }
13) };
14) Singleton* Singleton::single = NULL;

15) Singleton* Singleton::getInstance()
16) {
17)     if (!single)   {
18)         single = new Singleton();
19)         return single;
20)     }
21)     else   {       return single;   }
22) }
```

```
29) void Singleton::method()
30) {
31)     cout << "Method of the singleton
    class" << endl;
32) }

33) int main()
34) {
35)     Singleton *sc1,*sc2;
36) //  sc1->method();
37)     sc1 = Singleton::getInstance();
38)     sc1->method();
39)     sc2 = Singleton::getInstance();
40)     sc2->method();

41)     return 0;
42) }
```

Tracing program

C++ SINGLETON SAMPLE IMPLEMENTATION CODE

```
1)  // Declaration
2)  class Singleton {
3)  public:
4)      static Singleton* Instance();
5)  protected:
6)      Singleton();
7)  private:
8)      static Singleton* _instance;
9)  }

10) // Implementation
11) Singleton* Singleton::_instance = 0;

12) Singleton* Singleton::Instance() {
13)     if (_instance == 0) {
14)         _instance = new Singleton;
15)     }
16)     return _instance;
17) }
```

MODEL VIEW CONTROLLER (MVC)

- TYPICAL APPLICATIONS HAVE THREE FUNDAMENTAL PARTS:
  - DATA (MODEL)
  - AN INTERFACE TO VIEW AND MODIFY THE DATA (VIEW)
  - OPERATIONS THAT CAN BE PERFORMED ON THE DATA (CONTROLLER)
- THE MVC PATTERN:
  - THE MODEL REPRESENTS THE DATA, AND DOES NOTHING ELSE
    - THE MODEL DOES NOT DEPEND ON THE CONTROLLER OR THE VIEW
  - THE VIEW DISPLAYS THE MODEL DATA, AND SENDS USER ACTIONS (E.G. BUTTON CLICKS) TO THE CONTROLLER
    - THE VIEW CAN BE INDEPENDENT OF BOTH THE MODEL AND THE CONTROLLER; OR
    - ACTUALLY BE THE CONTROLLER, AND THEREFORE DEPEND ON THE MODEL.
  - THE CONTROLLER PROVIDES MODEL DATA TO THE VIEW, AND INTERPRETS USER ACTIONS SUCH AS BUTTON CLICKS. THE CONTROLLER DEPENDS ON THE VIEW AND THE MODEL. IN SOME CASES, THE CONTROLLER AND THE VIEW ARE THE SAME OBJECT.

MVC EXAMPLE I

- EXAMPLE: ADDRESS BOOK
  - THE MODEL IS A LIST OF PERSON OBJECTS,
  - THE VIEW IS A GUI WINDOW THAT DISPLAYS THE LIST OF PEOPLE, AND
  - THE CONTROLLER HANDLES ACTIONS
    - SUCH AS "DELETE PERSON", "ADD PERSON", "EMAIL PERSON", ETC.

MVC EXAMPLE I (CONTD.)

- E.g., not use MVC because the model depends on the view.

```
1)  void Person::setPicture(Picture pict)
2)  {
3)      m_picture = pict; //set the member variable
4)      m_listView->reloadData(); //update the view
5)  }
```

- Use MVC

```
1)  void Person::setPicture(Picture pict){
2)      m_picture = pict; //set the member variable
3)  }

4)  void PersonListController::changePictureAtIndex(
                                  Picture newPict, int personIndex)
5)  {
6)      m_personList[personIndex].setPicture(newPict); //modify the model
7)      m_listView->reloadData(); //update the view
8)  }
```

MVC EXAMPLE I (CONTD.)

- Use MVC

```
1)    void Person::setPicture(Picture pict){
2)        m_picture = pict; //set the member variable
3)    }
4)    void PersonListController::changePictureAtIndex(Picture newPict, int personIndex)
5)    {
6)        m_personList[personIndex].setPicture(newPict); //modify the model
7)        m_listView->reloadData(); //update the view
8)    }
```

- PERSON CLASS KNOWS NOTHING ABOUT THE VIEW.
- THE PERSONLISTCONTROLLER HANDLES BOTH CHANGING THE MODEL, AND UPDATING THE VIEW.
- THE VIEW WINDOW TELLS THE CONTROLLER ABOUT USER ACTIONS
  - IN THIS CASE, IT TELLS THE CONTROLLER THAT THE USER CHANGED THE PICTURE OF A PERSON

ADVANTAGES OF MVC

- MVC MAKES MODEL CLASSES REUSABLE WITHOUT MODIFICATION

Basic Details

Interests

Social Life

Work & Play

Settings

Username

Ms. Rachel McAdams

Email

notebookchick@gmail.com

Phone Number

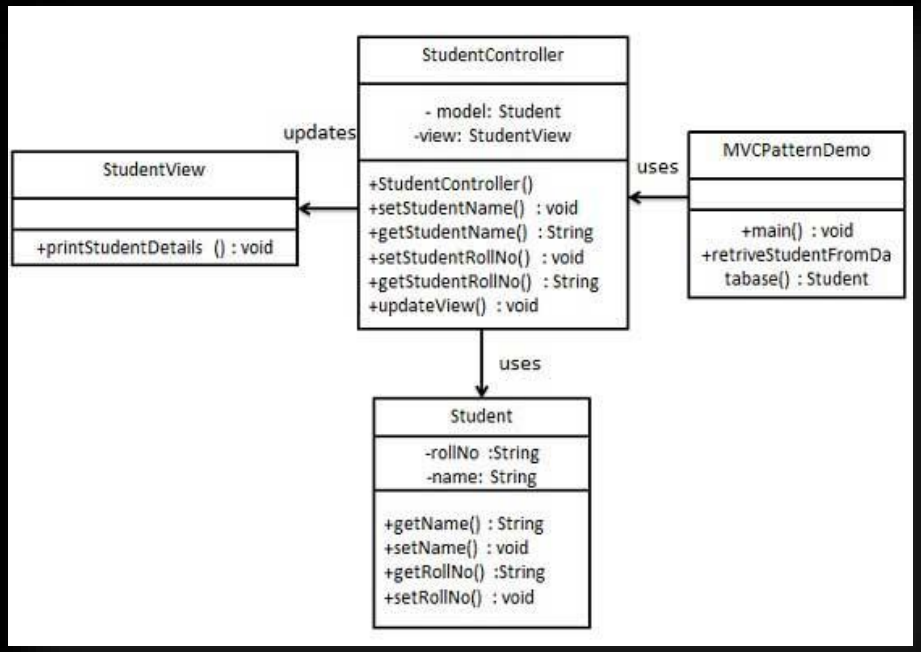
123-456-7890

Gender

Update

MVC EXAMPLE II

- STUDENT
  - ACTING AS A MODEL
- STUDENTVIEW
  - PRINT STUDENT DETAILS ON CONSOLE
- STUDENTCONTROLLER
  - RESPONSIBLE TO STORE DATA IN STUDENT OBJECT AND
  - UPDATE VIEW STUDENTVIEW ACCORDINGLY



MVC EXAMPLE II (CONTD.)

```
1)  class Student
2)  {
3)      private:
4)       String rollNo;
5)       String name;
6)      public:
7)       String getRollNo()  {        return rollNo;        }
8)       void setRollNo(String prollNo) { rollNo = prollNo;    }
9)       String getName() {      return name;   }
10)     void setName(String pname) {  name = pname;       }
11)}
```

STUDENTVIEW

```
1)  class StudentView {
2)      public:
3)        void printStudentDetails(String studentName,
                      String studentRollNo)
4)        {
5)                    cout<< "Student: “<< “Name: " << studentName
                      << "Roll No: " << studentRollNo;
6)      }
7)  }
```

STUDENTCONTROLLER

```
1)    class StudentController {
2)       private:
3)          Student model;
4)          StudentView view;

5)       public:
6)          StudentController(Student pmodel, StudentView pview)
7)          {
8)                      model = pmodel;
9)                      view = pview;
10)         }
11)         void setStudentName(String name) {  model.setName(name); }
12)         String getStudentName() { return model.getName();            }

13)         void setStudentRollNo(String rollNo){  model.setRollNo(rollNo); }
14)         String getStudentRollNo(){   return model.getRollNo();          }

15)         void updateView()
16)         {
17)            view.printStudentDetails(model.getName(), model.getRollNo());
18)         }
19)   }
```

```
1)  int main()
2)  {
3)      //fetch student record based on his roll no from the database
4)      Student model  = retriveStudentFromDatabase();
5)      //Create a view : to write student details on console
6)      StudentView view = new StudentView();
7)      StudentController controller = new StudentController(model, view);
8)      controller.updateView();
9)      //update model data
10)     controller.setStudentName("John");
11)     controller.updateView();
12)   }

13) Student retriveStudentFromDatabase()
14) {
15)     Student student = new Student();
16)     student.setName("Robert");
17)     student.setRollNo("10");
18)     return student;
19)   }
20) }
```

```
Student:
Name: Robert
Roll No: 10
Student:
Name: John
Roll No: 10
```

UML

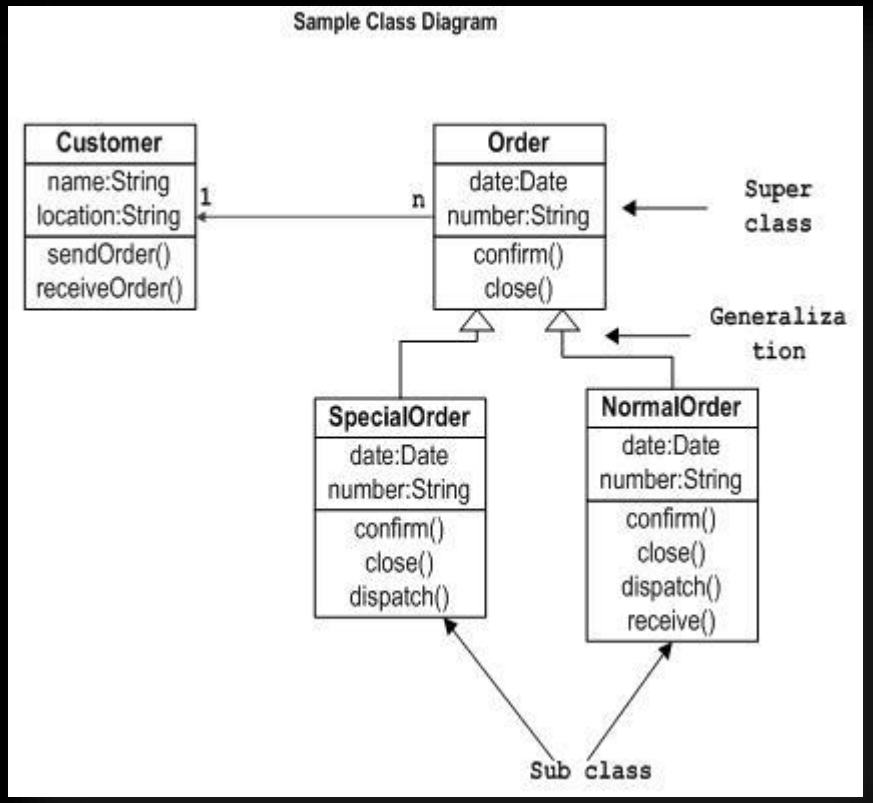
UNIFIED MODELING LANGUAGE

CLASS DIAGRAM

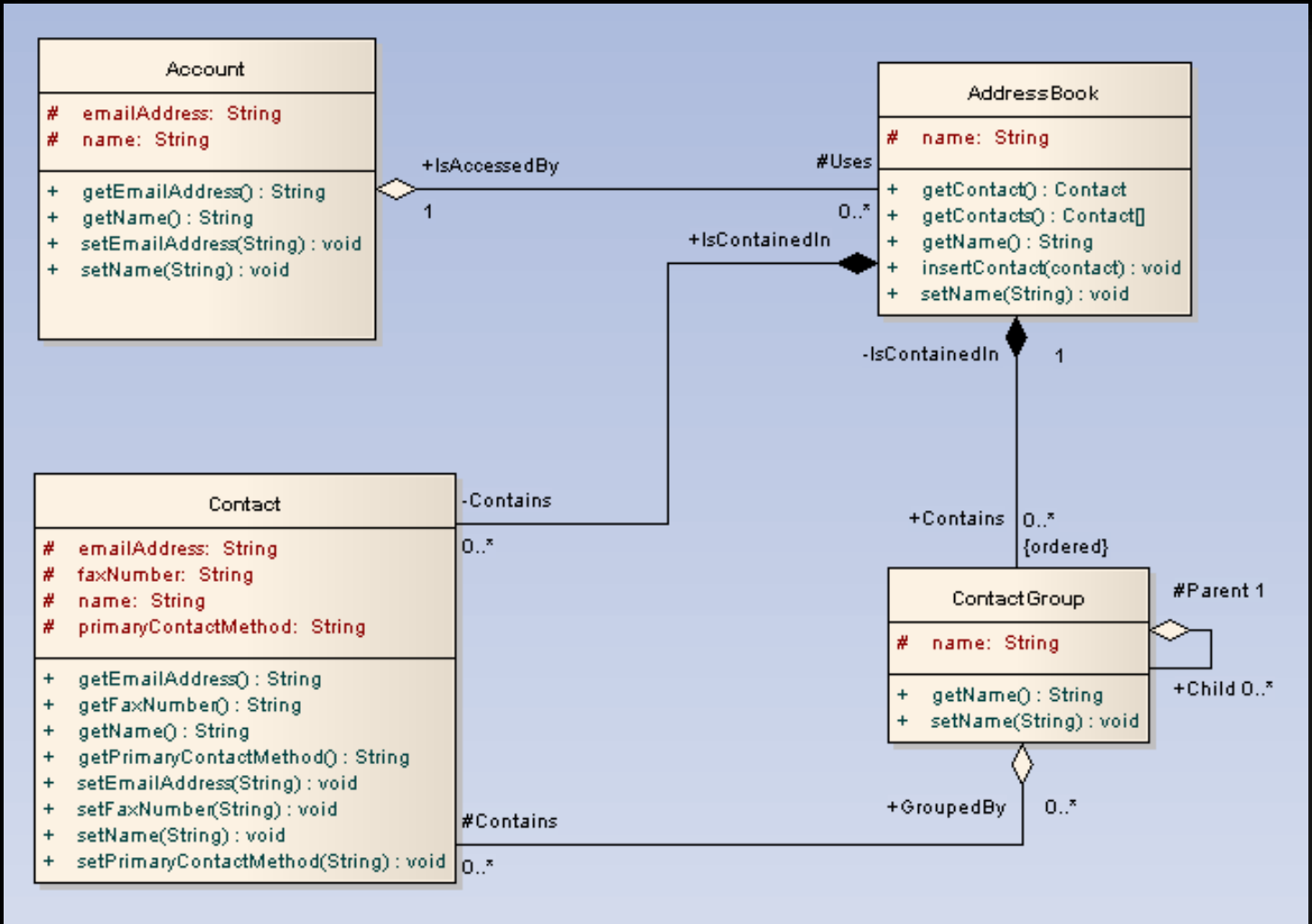
- TO MODEL THE STATIC VIEW OF AN APPLICATION
- DIRECTLY MAPPED WITH OBJECT ORIENTED LANGUAGES
- PURPOSE OF THE CLASS :
  - ANALYSIS AND DESIGN OF THE STATIC VIEW OF AN APPLICATION.
  - DESCRIBE RESPONSIBILITIES OF A SYSTEM.
  - BASE FOR COMPONENT AND DEPLOYMENT DIAGRAMS.
  - FORWARD AND REVERSE ENGINEERING.

CLASS DIAGRAM- AN ORDER SYSTEM

- ORDER AND CUSTOMER ARE IDENTIFIED AS THE TWO ELEMENTS OF THE SYSTEM
  - THEY HAVE A ONE TO MANY RELATIONSHIP BECAUSE A CUSTOMER CAN HAVE MULTIPLE ORDERS.
- ORDER CLASS IS AN ABSTRACT CLASS AND IT HAS TWO CONCRETE CLASSES
  - (INHERITANCE RELATIONSHIP) SPECIALORDER AND NORMALORDER.
- THE TWO INHERITED CLASSES HAVE ALL THE PROPERTIES AS THE ORDER CLASS.
  - IN ADDITION THEY HAVE ADDITIONAL FUNCTIONS LIKE DISPATCH () AND RECEIVE ().



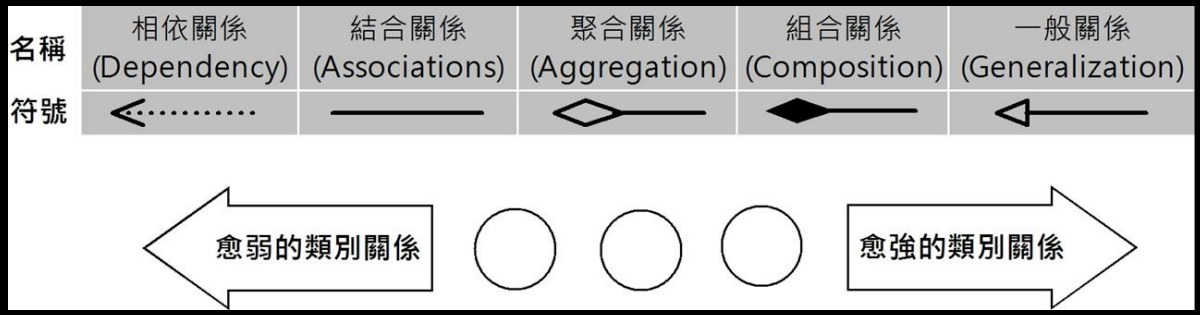
https://msdn.microsoft.com/en-us/library/dd409437.aspx



- The lighter aggregation indicates that the class "Account" uses AddressBook, but does not necessarily contain an instance of it.
- The strong, composite aggregations by the other connectors indicate ownership or containment of the source classes by the target classes,
  - for example Contact and ContactGroup values will be contained in AddressBook.

CLASS NOTATION

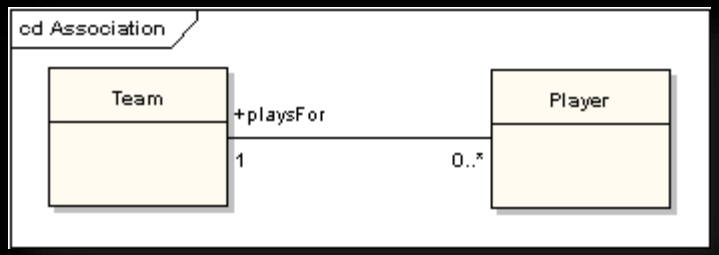
- +: A PUBLIC LEVEL OF VISIBILITY
- - : PRIVATE
- # : PROTECTED
- ~ : PACKAGE
- : DATA TYPE
- =: DEFAULT VALUE



Symbol	Explanation
1	Just one instance
3	Just three instances
* Or 0..*	From 0 to many
m..n	From m to n

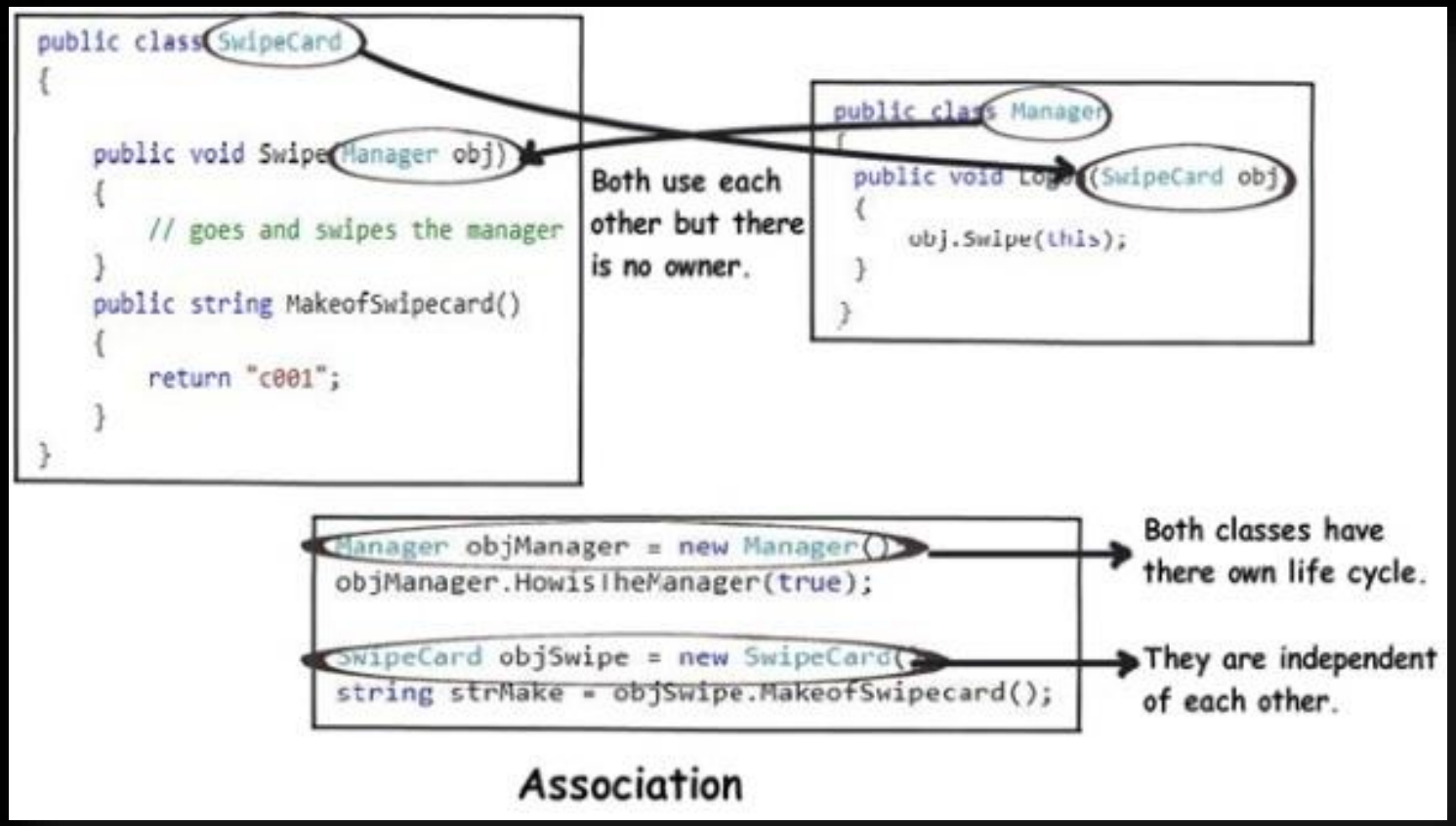
ASSOCIATIONS

- AN ASSOCIATION IMPLIES TWO MODEL ELEMENTS HAVE A RELATIONSHIP
  - USUALLY IMPLEMENTED AS AN INSTANCE VARIABLE IN ONE CLASS



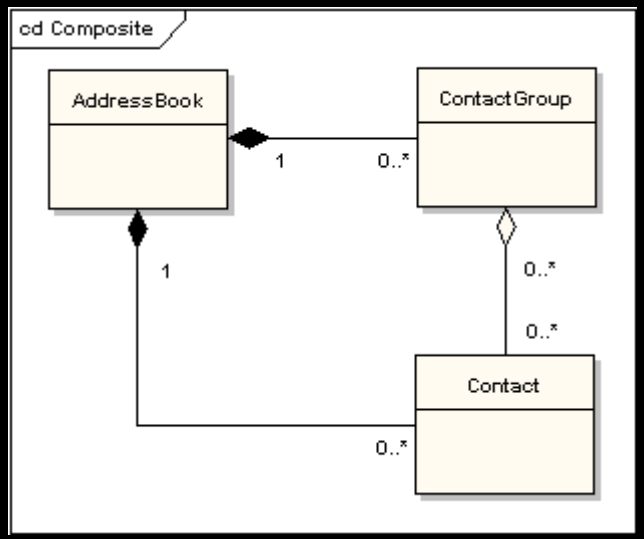
- THIS CONNECTOR MAY INCLUDE:
  - NAMED ROLES AT EACH END, CARDINALITY, DIRECTION AND CONSTRAINTS
- WHEN CODE IS GENERATED FOR CLASS DIAGRAMS, NAMED ASSOCIATION ENDS BECOME INSTANCE VARIABLES IN THE TARGET CLASS
  - EXAMPLE: "PLAYSFOR" WILL BECOME AN INSTANCE VARIABLE IN THE "PLAYER" CLASS

ASSOCIATION EXAMPLE



AGGREGATIONS

- TO DEPICT ELEMENTS WHICH ARE MADE UP OF SMALLER COMPONENTS
  - SHOWN BY A **WHITE DIAMOND-SHAPED** ***ARROWHEAD*** POINTING TOWARDS THE TARGET OR PARENT CLASS.



AGGREGATION EXAMPLE

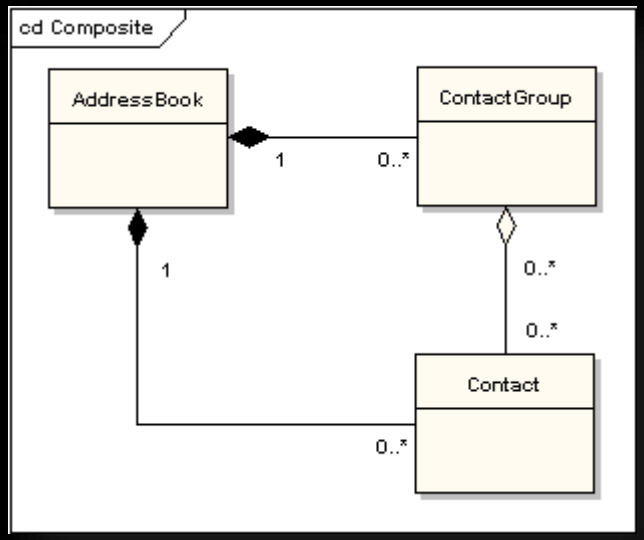
http://www.learncpp.com/cpp-tutorial/103-aggregation/

```
1)  class Teacher {
2)  private:
3)      string m_strName;
4)  public:
5)      Teacher(string strName)        : m_strName(strName)    {    }
6)      string GetName() { return m_strName; }
7)  };
8)
9)  class Department {
10) private:
11)     Teacher *m_pcTeacher; // This dept holds only one teacher
12)  public:
13)     Department(Teacher *pcTeacher=NULL)        : m_pcTeacher(pcTeacher)    {    }
14) };

15) int main()
16) {
17)     // Create a teacher outside the scope of the Department
18)     Teacher *pTeacher = new Teacher("Bob"); // create a teacher
19)     {
20)         // Create a department and use the constructor parameter to pass the teacher to it.
21)         Department cDept(pTeacher);
22)      } // cDept goes out of scope here and is destroyed
23)      // pTeacher still exists here because cDept did not destroy it
24)     delete pTeacher;
25) }
```

COMPOSITION

- A STRONGER FORM OF AGGREGATION – A COMPOSITE AGGREGATION
  - SHOWN BY A BLACK DIAMOND-SHAPED ARROWHEAD



- IF THE PARENT OF A COMPOSITE AGGREGATION IS DELETED, USUALLY ALL OF ITS PARTS ARE DELETED WITH IT
  - COMPOSITION IS USED FOR OBJECTS THAT HAVE A HAS-A RELATIONSHIP TO EACH OTHER

COMPOSITION EXAMPLE

```
1)  #include "CPU.h"
2)  #include "Motherboard.h"
3)  #include "RAM.h"
4)
5)  class PersonalComputer
6)  {
7)  private:
8)      CPU m_cCPU;
9)      Motherboard m_cMotherboard;
10)     RAM m_cRAM;
11) };
```

http://www.learncpp.com/cpp-tutorial/102-composition/

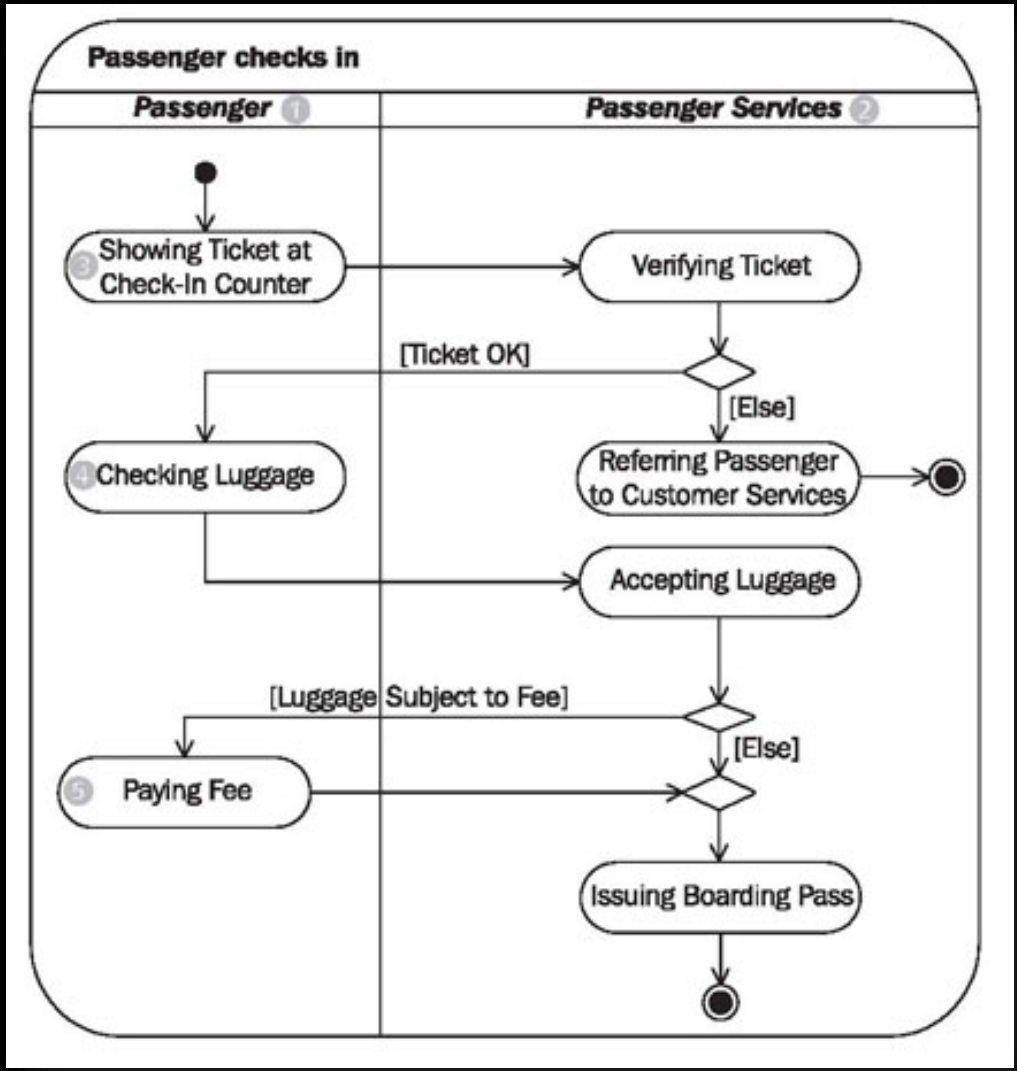
```
12) PersonalComputer::PersonalComputer(int nCPUSpeed, char *strMotherboardModel,
    int nRAMSize)
13)     : m_cCPU(nCPUSpeed), m_cMotherboard(strMotherboardModel),
          m_cRAM(nRAMSize)
14) {
15) }
```

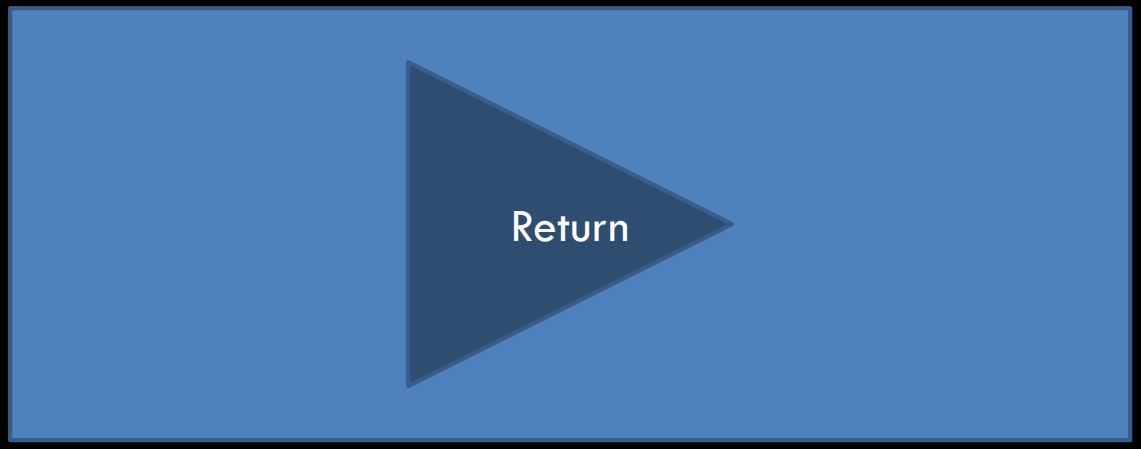
THE DIFFERENCES BETWEEN COMPOSITION AND AGGREGATION

- COMPOSITIONS:
  - TYPICALLY USE **NORMAL MEMBER VARIABLES**
  - CAN USE POINTER VALUES IF THE COMPOSITION CLASS AUTOMATICALLY HANDLES ALLOCATION/DEALLOCATION
  - RESPONSIBLE FOR CREATION/DESTRUCTION OF SUBCLASSES
- AGGREGATIONS:
  - TYPICALLY USE **POINTER VARIABLES** THAT POINT TO AN OBJECT THAT LIVES OUTSIDE THE SCOPE OF THE AGGREGATE CLASS
  - CAN USE REFERENCE VALUES THAT POINT TO AN OBJECT THAT LIVES OUTSIDE THE SCOPE OF THE AGGREGATE CLASS
  - NOT RESPONSIBLE FOR CREATING/DESTROYING SUBCLASSES

ACTIVITY DIAGRAMS

- HTTPS://SOURCEMAKING.COM/UML/MODELING-BUSINESS-SYSTEMS/EXTERNAL-VIEW/ACTIVITY-DIAGRAMS




Return

LEARNING OBJECTIVES

- PATTERNS
  - ADAPTER PATTERN
  - MODEL-VIEW-CONTROLLER PATTERN
  - SORTING PATTERN AND IT'S EFFICIENCY
  - PATTERN FORMALISM
- UML
  - HISTORY OF UML
  - UML CLASS DIAGRAMS
  - CLASS INTERACTIONS

INTRODUCTION

- PATTERNS AND UML
  - SOFTWARE DESIGN TOOLS
  - PROGRAMMING-LANGUAGE INDEPENDENT
    - ASSUMING OBJECT-ORIENTED-CAPABLE
- PATTERN
  - LIKE "ORDINARY" PATTERN IN OTHER CONTEXTS
    - AN "OUTLINE" OF SOFTWARE TASK
  - CAN RESULT IN DIFFERENT CODE IN DIFFERENT BUT SIMILAR TASKS
- UML
  - GRAPHICAL LANGUAGE FOR OOP DESIGN

PATTERNS

- PATTERNS ARE DESIGN PRINCIPLES
  - APPLY ACROSS VARIETY OF SOFTWARE *APPLICATIONS*
  - MUST ALSO APPLY ACROSS VARIETY OF *SITUATIONS*
  - MUST MAKE ASSUMPTIONS ABOUT APPLICATION DOMAIN
- EXAMPLE:
  ITERATOR PATTERN APPLIES TO CONTAINERS OF ALMOST ANY KIND

PATTERN EXAMPLE: ITERATORS

- RECALL ITERATORS
- ITERATOR PATTERN APPLIES TO CONTAINERS OF ALMOST ANY KIND
- 1ST DESCRIBED AS "ABSTRACT"
  - AS WAYS OF CYCLING THRU ANY DATA IN ANY CONTAINER
- THEN GAVE SPECIFIC APPLICATIONS
  - SUCH AS LIST ITERATOR, CONSTANT LIST ITERATOR, REVERSE LIST ITERATOR, ETC.

CONSIDER NO PATTERNS

- ITERATORS
  - IMAGINE HUGE AMOUNT OF DETAIL IF ALL CONTAINER ITERATORS PRESENTED SEPARATELY!
  - IF EACH HAD DIFFERENT NAMES FOR BEGIN(), END()
  - TO MAKE "SENSE" OF IT, LEARNERS MIGHT MAKE PATTERN THEMSELVES!
- UNTIL PATTERN DEVELOPED, ALL WERE DIFFERENT
  - "SEEMED" SIMILAR, BUT NOT ORGANIZED
- CONSIDER CONTAINERS AS WELL
  - SAME ISSUES!

ADAPTER PATTERN

- TRANSFORMS ONE CLASS INTO DIFFERENT CLASS
  - WITH NO CHANGES TO UNDERLYING CLASS
  - ONLY "ADDING" TO INTERFACE
- RECALL STACK AND QUEUE TEMPLATE CLASSES
  - BOTH CAN CHOOSE UNDERLYING CLASS USED TO STORE DATA:
    STACK<VECTOR<INT>> -- INT STACK UNDER VECTOR
    STACK<LIST<INT>> -- INT STACK UNDERLYING LIST
  - ALL CASES UNDERLYING CLASS NOT CHANGED
    - ONLY INTERFACE IS ADDED

ADAPTER PATTERN INTERFACE

- HOW TO ADD INTERFACE?
  - IMPLEMENTATION DETAIL
  - NOT PART OF PATTERN
- BUT… TWO WAYS:
  - EXAMPLE: FOR STACK ADAPTER:
    - UNDERLYING CONTAINER CLASS COULD BE MEMBER VARIABLE OF STACK CLASS
    - OR STACK CLASS COULD BE DERIVED CLASS OF UNDERLYING CONTAINER CLASS

MODEL-VIEW-CONTROLLER PATTERN

- WAY OF DIVIDING I/O TASK OUT
  - MODEL PART: HEART OF APPLICATION
  - VIEW PART: OUTPUT
    - DISPLAYS PICTURE OF MODEL'S STATE
  - CONTROLLER PART: INPUT
    - RELAYS COMMANDS FROM USER TO MODEL
- A DIVIDE AND CONQUER STRATEGY
  - ONE BIG TASK → THREE SMALLER TASKS
    - EACH WITH WELL-DEFINED RESPONSIBILITIES

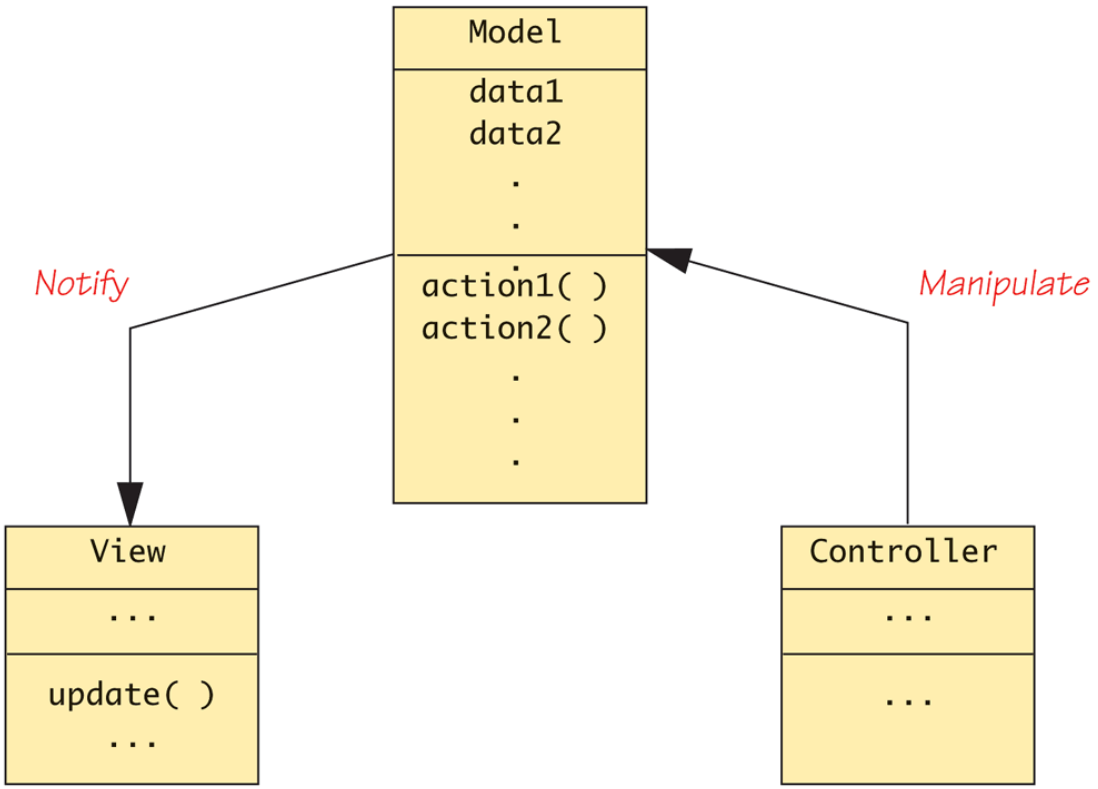
MODEL-VIEW-CONTROLLER PATTERN

- ANY APPLICATION CAN FIT
- BUT PARTICULARLY SUITED TO GUI DESIGN PROJECTS
  - WHERE VIEW CAN ACTUALLY BE VISUALIZATION OF STATE OF MODEL

DISPLAY 20.1

MODEL-VIEW-CONTROLLER PATTERN

Display 20.1 **Model-View-Controller Pattern**



A SORTING PATTERN EXAMPLE

- SIMILAR PATTERN AMONG "MOST-EFFICIENT" SORTING ALGORITHMS:
  - RECURSIVE
  - DIVIDE LIST INTO SMALLER LISTS
  - THEN RECURSIVELY SORT SMALLER LISTS
  - RECOMBINE TWO SORTED LISTS OBTAINING ONE FINAL SORTED LIST

SORTING PATTERN

- CLEARLY A DIVIDE-AND-CONQUER STRATEGY
- HEART OF PATTERN:
  INT SPLITPT = SPLIT(A, BEGIN, END);
  SORT(A, BEGIN, SPLITPT);
  SORT(A, SPLITPT, END);
  JOIN(A, BEGIN, SPLITPT, END);
- NOTE NO DETAILS ON HOW SPLIT AND JOIN ARE DEFINED
  - DIFFERENT DEFINITIONS WILL YIELD DIFFERENT SORTING ALGORITHMS

FUNCTION SPLIT

- REARRANGES ELEMENTS
  - IN INTERVAL [BEGIN, END]
- DIVIDES INTERVAL AT SPLIT POINT, *SPLITPT*
- TWO NEW INTERVALS THEN SORTED
  - [BEGIN, SPLITPT) – FIRST HALF
  - [SPLITPT, END) – SECOND HALF
- NO DETAILS IN PATTERN
  - NOTHING ABOUT HOW REARRANGE AND DIVIDE TAKES PLACE

FUNCTION JOIN

- COMBINES TWO SORTED INTERVALS
  - PRODUCES FINAL SORTED VERSION
- AGAIN, NO DETAILS
  - JOIN FUNCTION COULD PERFORM MANY WAYS

SAMPLE REALIZATION OF SORTING PATTERN: MERGESORT

- SIMPLEST "REALIZATION" OF SORTING PATTERN IS MERGESORT
- DEFINITION OF SPLIT VERY SIMPLE
  - JUST DIVIDES ARRAY INTO TWO INTERVALS
  - NO REARRANGING OF ELEMENTS
- DEFINITION OF JOIN COMPLEX!
  - MUST SORT SUBINTERVALS
  - THEN MERGE, COPYING TO TEMPORARY ARRAY

MERGESORT'S JOIN FUNCTION

- SEQUENCE:
  - COMPARE SMALLEST ELEMENTS IN EACH INTERVAL
  - SMALLER OF TWO → NEXT POSITION IN TEMPORARY ARRAY
    - REPEATED UNTIL THROUGH BOTH INTERVALS
  - RESULT IS FINAL SORTED ARRAY

SORT PATTERN COMPLEXITY

- TRADE-OFF BETWEEN SPLIT AND JOIN
  - EITHER CAN BE SIMPLE AT EXPENSE OF OTHER
  - E.G., IN MERGESORT, SPLIT FUNCTION SIMPLE AT EXPENSE OF COMPLICATED JOIN FUNCTION
  - COULD VARY IN OTHER ALGORITHMS
- COMES DOWN TO "WHO DOES WORK?"

CONSIDER QUICKSORT

- COMPLEXITY SWITCH
  - JOIN FUNCTION SIMPLE, SPLIT FUNCTION COMPLEX
- LIBRARY FILES
  - INCLUDE FILES "MERGESORT.CPP", "QUICKSORT.CPP" BOTH GIVE TWO DIFFERENT REALIZATIONS OF SAME SORT PATTERN
  - PROVIDE SAME INPUT AND OUTPUT!

QUICKSORT REALIZATION

- A SOPHISTICATED SPLIT FUNCTION
  - ARBITRARY VALUE CHOSEN, CALLED "SPLITTING VALUE"
  - ARRAY ELEMENTS REARRANGED "AROUND" SPLITTING VALUE
    - THOSE LESS THAN IN FRONT, GREATER THAN IN BACK
    - SPLITTING VALUE ESSENTIALLY "DIVIDES" ARRAY
  - TWO "SIDES" THEN SORTED RECURSIVELY
- FINALLY COMBINED WITH JOIN
  - WHICH DOES NOTHING!

SORTING PATTERN EFFICIENCY

- MOST EFFICIENT REALIZATIONS "DIVIDE" LIST INTO TWO CHUNKS
  - SUCH AS HALF AND HALF
  - INEFFICIENT IF DIVIDED INTO "FEW" AND "REST"
- MERGESORT: O(N LOG N)
- QUICKSORT:
  - WORST CASE: $O(N^2)$ (IF SPLIT UNEVEN)
  - AVERAGE CASE: O(N LOG N)
    - IN PRACTICE, ONE OF BEST SORT ALGORITHMS

PRAGMATICS AND PATTERNS

- PATTERNS ARE GUIDES, NOT REQUIREMENTS
  - NOT COMPELLED TO FOLLOW ALL FINE DETAILS
  - CAN TAKE "LIBERTIES" AND ADJUST FOR PARTICULAR NEEDS
    - LIKE EFFICIENCY ISSUES
- PATTERN FORMALISM
  - STANDARD TECHNIQUES EXIST FOR USING PATTERNS
  - PLACE OF PATTERNS IN SOFWARE DESIGN PROCESS NOT YET CLEAR
    - IS CLEAR THAT MANY BASIC PATTERNS ARE USEFUL

UML

- UNIFIED MODELING LANGUAGE
- ATTEMPT TO PRODUCE "HUMAN-ORIENTED" WAYS OF REPRESENTING PROGRAMS
  - LIKE PSEUDOCODE: THINK OF PROBLEM, WITHOUT DETAILS OF LANGUAGE
- PSEUDOCODE VERY STANDARD, VERY USED
  - BUT IT'S A LINEAR, ALGEBRAIC REPRESENTATION
- PREFER "GRAPHICAL" REPRESENTATION
  - ENTER UML

UML DESIGN

- DESIGNED TO REFLECT/BE USED WITH OBJECT-ORIENTED PROGRAMMING PHILOSOPHY
- A PROMISING EFFORT!
- MANY COMPANIES HAVE ADOPTED UML FORMALISM IN SOFTWARE DESIGN PROCESS

HISTORY OF UML

- DEVELOPED WITH OOP
- DIFFERENT GROUPS DEVELOPED OWN GRAPHICAL REPRESENTATIONS FOR OOP DESIGN
- 1996:
  - BOOCH, JACOBSEN, RUMBAUGH RELEASED EARLY VERSION OF UML
  - INTENDED TO "BRING TOGETHER" VARIOUS OTHER REPRESENTATIONS TO PRODUCE STANDARD FOR ALL OBJECT-ORIENTED DESIGN

UML LATELY

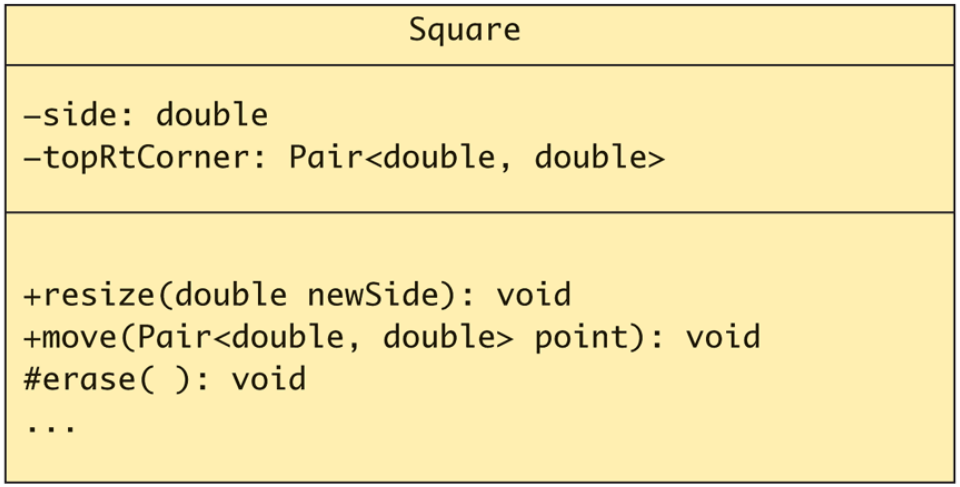
- SINCE 1996:
  - DEVELOPED AND REVISED WITH FEEDBACK FROM OOP COMMUNITY
- TODAY:
  - UML STANDARD MAINTAINED AND CERTIFIED BY OBJECT MANAGEMENT GROUP (OMG)
    - NON-PROFIT ORGANIZATION THAT PROMOTES USE OF OBJECT-ORIENTED TECHNIQUES

UML CLASS DIAGRAMS

- AS CLASSES ARE CENTRAL TO OOP…
- CLASS DIAGRAM IS SIMPLEST OF UML GRAPHICAL REPRESENTATIONS TO USE
  - THREE-SECTIONED BOX CONTAINS:
    - CLASS NAME
    - DATA SPECIFICATIONS
    - ACTIONS (CLASS MEMBER FUNCTIONS)

CLASS DIAGRAMS EXAMPLE: **DISPLAY 20.6** A UML CLASS DIAGRAM

Display 20.6 A UML Class Diagram



CLASS DIAGRAMS EXAMPLE NOTES

- DATA SECTION:
  - + SIGN INDICATES PUBLIC MEMBER
  - - SIGN INDICATES PRIVATE MEMBER
  - # INDICATES PROTECTED MEMBER
  - OUR EXAMPLE: BOTH PRIVATE (TYPICAL IN OOP)
- ACTIONS:
  - SAME +, -, # FOR PUBLIC, PRIVATE, PROTECTED
- NEED NOT PROVIDE ALL DETAILS
  - MISSING MEMBERS INDICATED WITH ELLIPSIS (…)

CLASS INTERACTIONS

- CLASS DIAGRAMS ALONE OF LITTLE VALUE
  - JUST REPEAT OF CLASS INTERFACE, OFTEN "LESS"
- MUST SHOW HOW OBJECTS OF VARIOUS CLASSES INTERACT
  - ANNOTATED ARROWS SHOW INFORMATION FLOW BETWEEN CLASS OBJECTS
    - RECALL MODEL-VIEW-CONTROLLER PATTERN
  - ANNOTATIONS ALSO FOR CLASS GROUPINGS INTO LIBRARY-LIKE AGGREGATES
    - SUCH AS FOR INHERITANCE

MORE CLASS INTERACTIONS

- UML IS EXTENSIBLE
  - IF YOUR NEEDS NOT IN UML, ADD THEM TO UML!
- FRAMEWORK EXISTS FOR THIS PURPOSE
  - PRESCRIBED STANDARD FOR ADDITIONS
  - ENSURES DIFFERENT SOFTWARE DEVELOPERS UNDERSTAND EACH OTHER'S UML

SUMMARY

- PATTERNS ARE DESIGN PRINCIPLES
  - APPLY ACROSS VARIETY OF SOFTWARE APPLICATIONS
- PATTERN CAN PROVIDE FRAMEWORK FOR COMPARING RELATED ALGORITHMS" EFFICIENCY
- UNIFIED MODELING LANGUAGE (UML)
  - GRAPHICAL REPRESENTATION LANGUAGE
  - DESIGNED FOR OBJECT-ORIENTED SOFTWARE DESIGN
- UML IS ONE FORMALISM USED TO EXPRESS PATTERNS