Lecture04: Design Engineering

EGCI341:WEEK 05

Outline

- Introduction to Design Model
- Design Steps
- Quality Attributes
- Level of Abstraction
- Design Classes
- Pattern-Based Software Design
- Frameworks

Design Model

- Design model is the place where customer requirements, business needs, and technical considerations come together in the formation of a product or system

Design creates a representation or model of the software

- ***Design model*** (Unlike analysis model) provides detail about:
  1. Software data structures
  2. Architecture
  3. Interfaces
  4. Components that are necessary to implement the system

Design Model (cont.)

- Design allows a software engineer to model the system or product

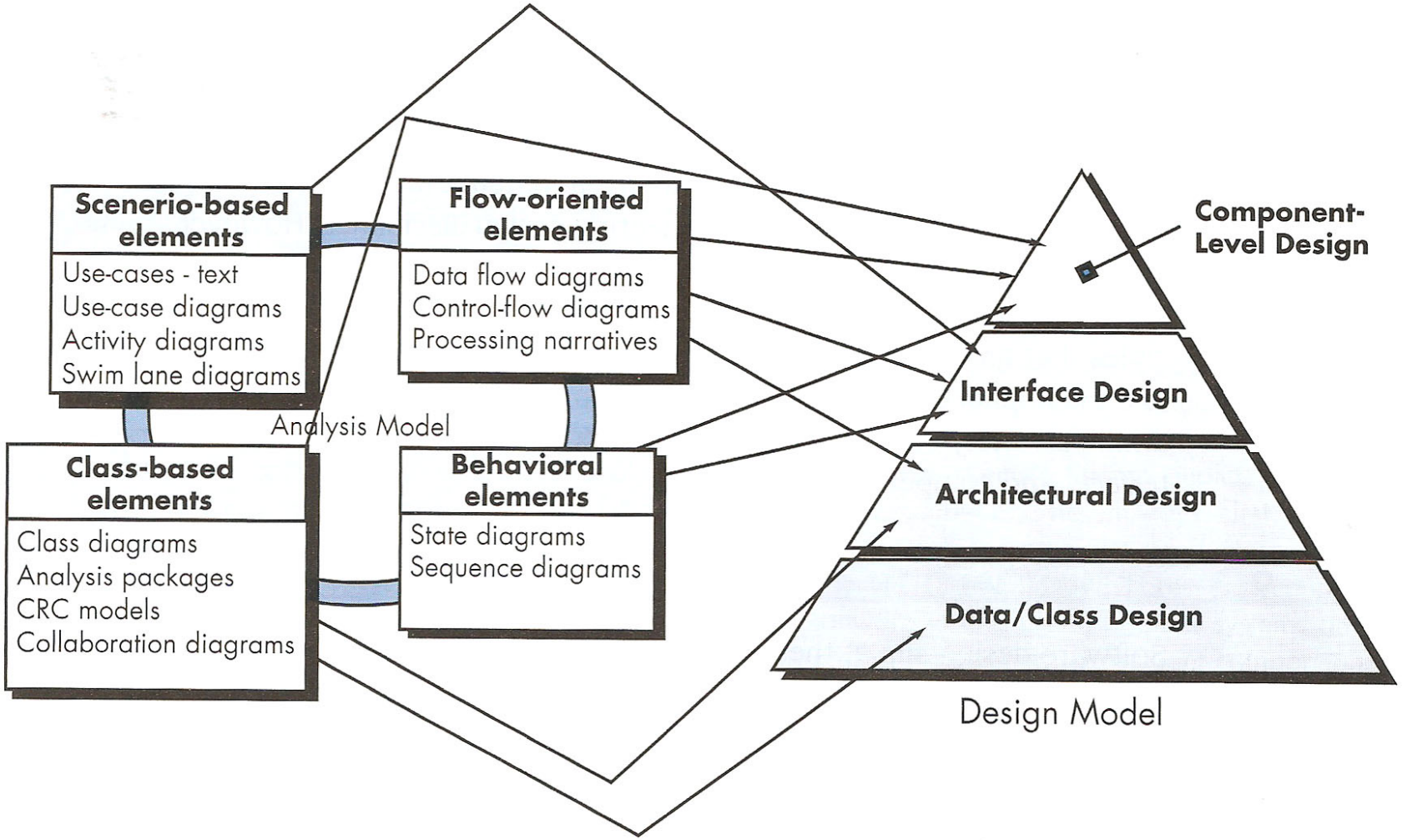
- This model can be assessed for software quality before:
  - Code is generated
  - Tests are conducted
  - End-users become involved in large numbers

- Design is the place where software quality is established

What are the steps?

- Design depicts the software in a number of different ways:
  1. *Architecture* of the system or product
  2. *Interfaces* that connect the software to:
     - End-users
     - Other systems and devices
     - Components
  3. *Software components* that are used to construct the system

Translating the analysis model into the design model [1]



Guide for the evaluation of a good design

- The design must implement *all of the explicit requirements* contained in the *analysis model*
- It must accommodate *all of the implicit requirements* desired by the *customer*
- The design must be *a readable, understandable* guide for those who:
  - Generate code
  - Test
  - Support the software
- The design should provide:
  - A complete picture of the software addressing the data, functional, and behavioral domains from an implementation perspective

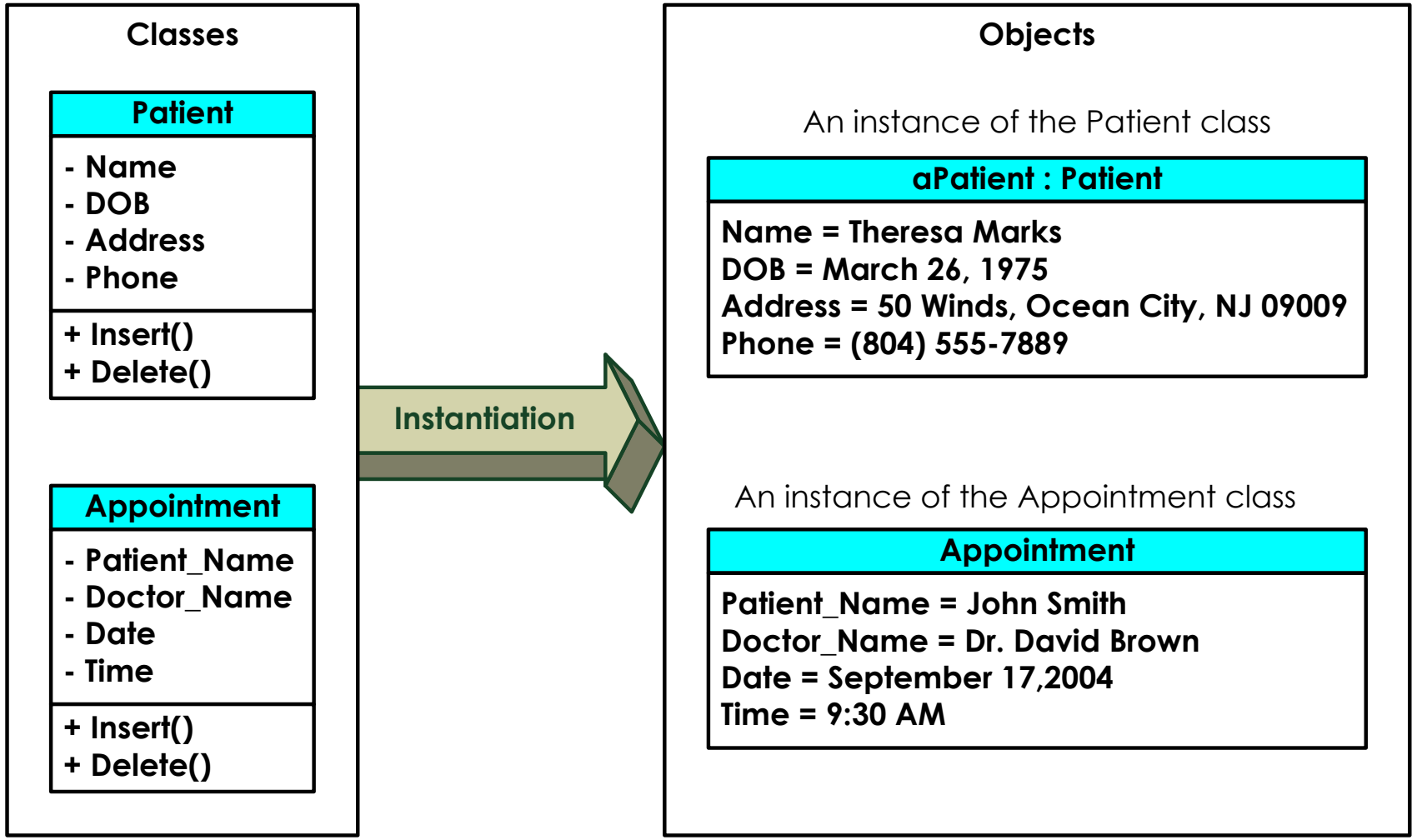
Class-Based Modeling

- If you look around a room,
  - There is a set of physical objects that can be easily identified, classified, and defined (in terms of attributes and operations)
- Problem of a software:
  - Classes (and objects) may be more difficult to define
- A class is the general template
  - Use to define and create specific instances or objects
- Every object is associated with a class

Write the class and object of computer lab's room

Class	*Object*

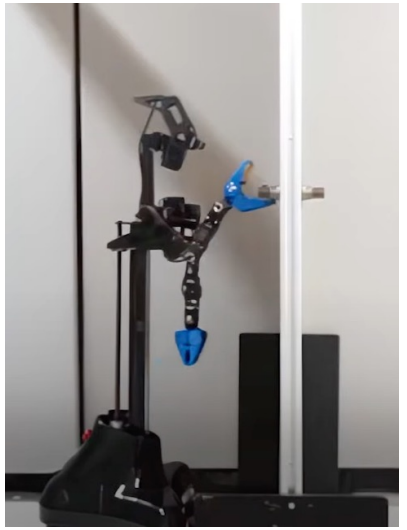
Classes and Objects



Level of Abstraction

Procedural Abstraction refers to a sequence of instructions that have a specific function

- Name of procedural abstraction implies the functions
- Example of a procedural abstraction
  - The word ***Open*** for "a door"
  - ***Open*** implies a long sequence of procedural steps



Data Abstraction is a named collection of data that describes a data object

- Context of the procedural abstraction ***open***,
  - Define a ***data abstraction*** called ***door***
  - Encompass *a set of attributes* that describe the ***door***

Reference: 1. A screenshot from the video of a robot learning how to open a door released by researchers in Japan. Door-opening is a difficult task for robots. HITACHI, LTD.
2. HTTPS://WWW.YOUTUBE.COM/WATCH?V=URMDJP-LUVO

Level of Abstraction (practice)

Write 2 levels of abstraction for both procedural and data abstraction

Procedural Abstraction
High Level:

Low Level:

Data Abstraction
High:

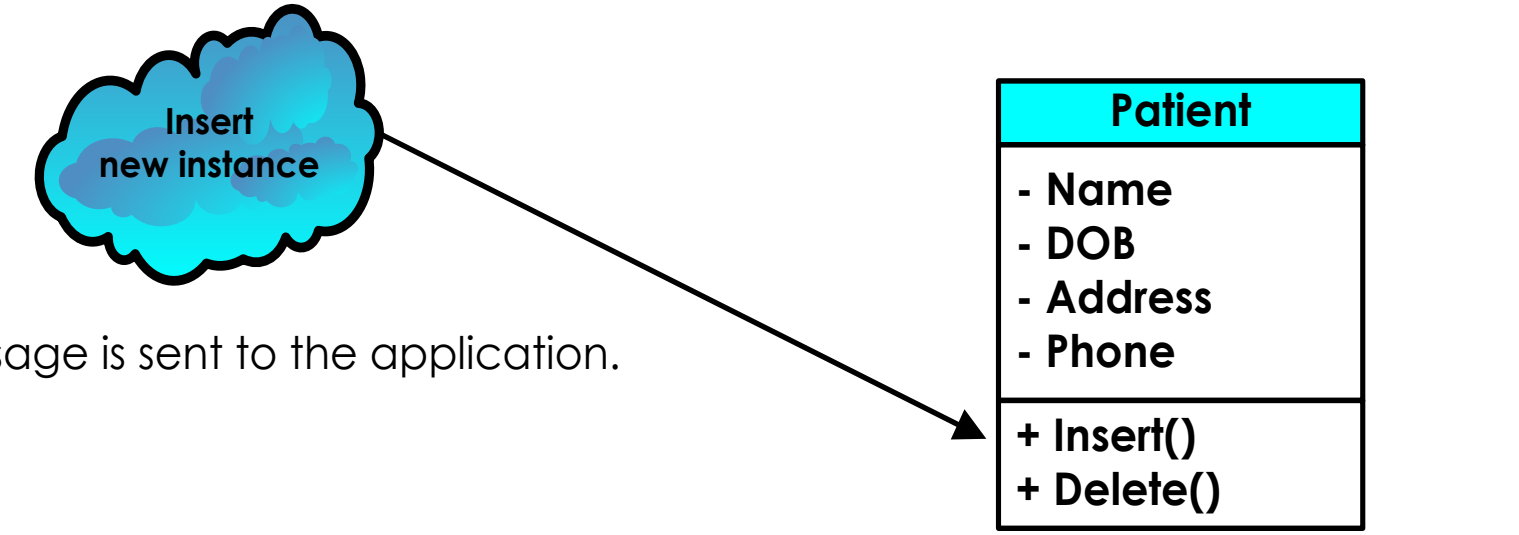
Low Level:

Methods and Messages

Objects communicate by message passing

- Messages are information sent to objects to trigger methods
- Messages can be implemented by procedure calls
  - Name of the service requested by the calling object
    - Name: procedure name
    - Information: parameter list
- Methods implement an object's behavior
  - A method is nothing more than an action that an object can perform

Methods and Messages (Cont.)



A message is sent to the application.

The object's insert method will respond to the message and insert a new patient instance.

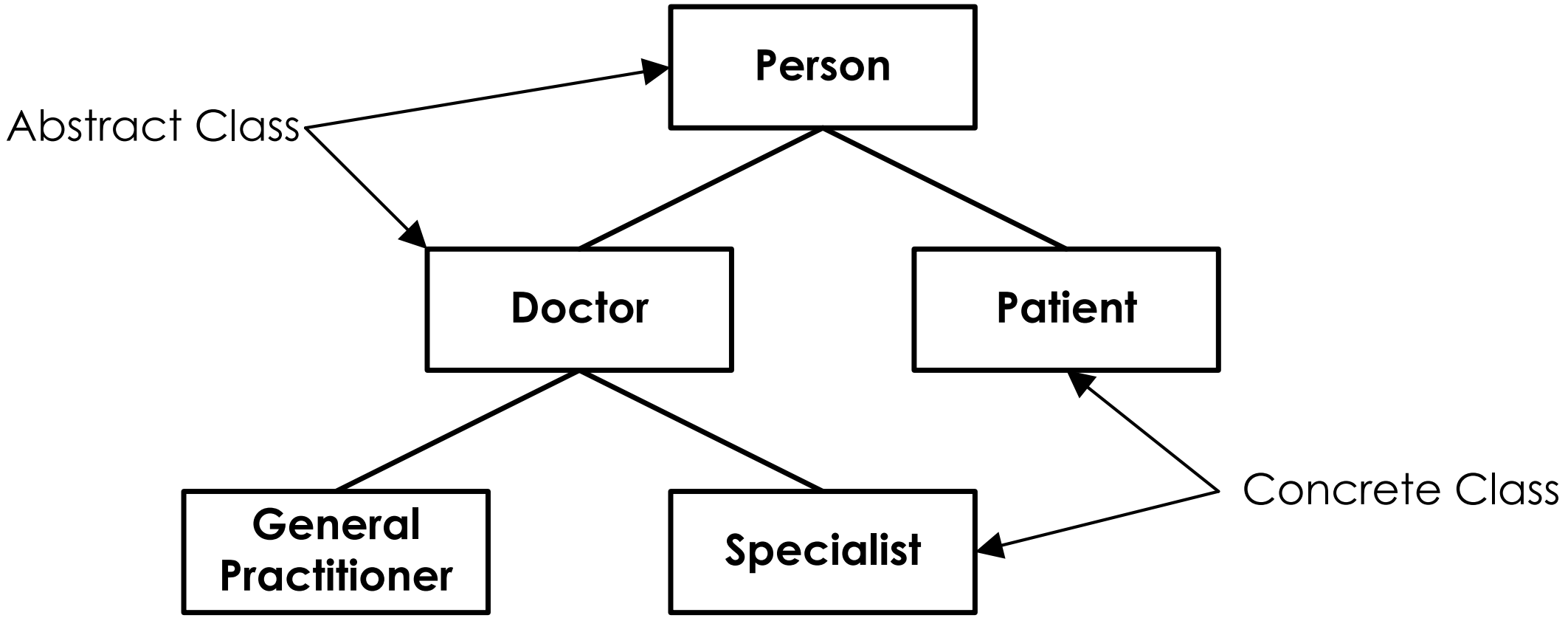
Message examples

```
// Call a method associated with a buffer
// object that returns the next value
//   in the buffer

                v = circularBuffer.Get () ;


// Call the method associated with a
// thermostat object that sets the
// temperature to be maintained

                thermostat.setTemp (20) ;
```

Generalization and Inheritance

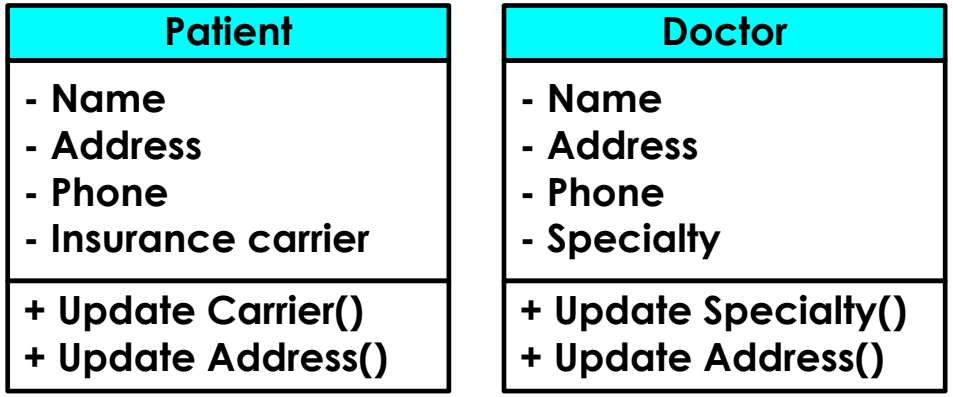
- Objects are members of classes that define attribute types and operations
- Classes may be arranged in a class hierarchy
  - One class (a superclass) is a generalization of one or more other classes (subclasses)
- A subclass inherits the attributes and operations from its super class and may add new methods or attributes
- Generalization is implemented as inheritance in OO programming languages

Inheritance

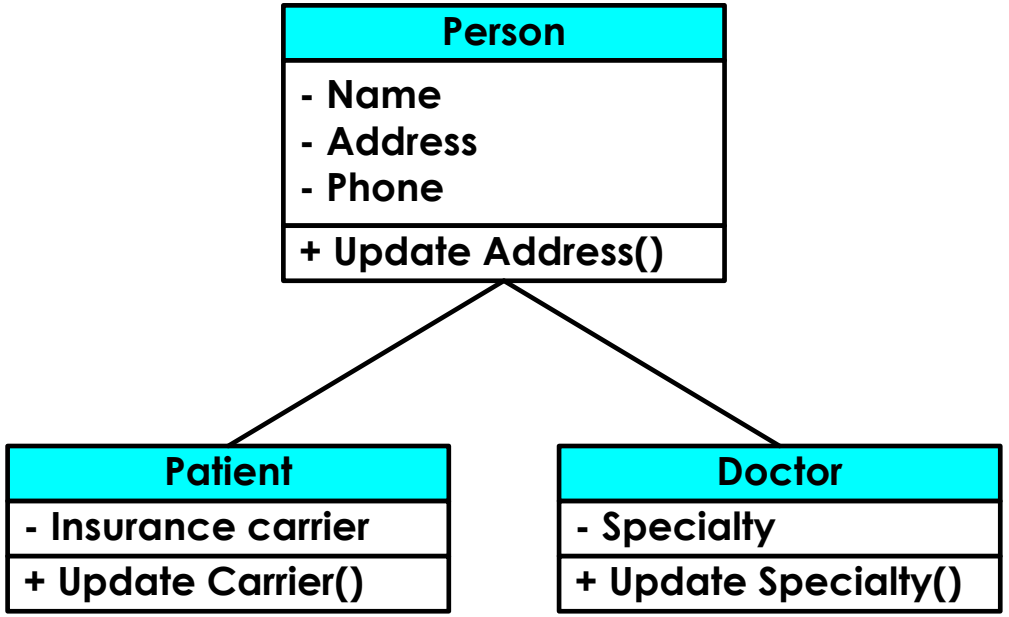


Inheritance (Cont.)

Without Inheritance



With Inheritance

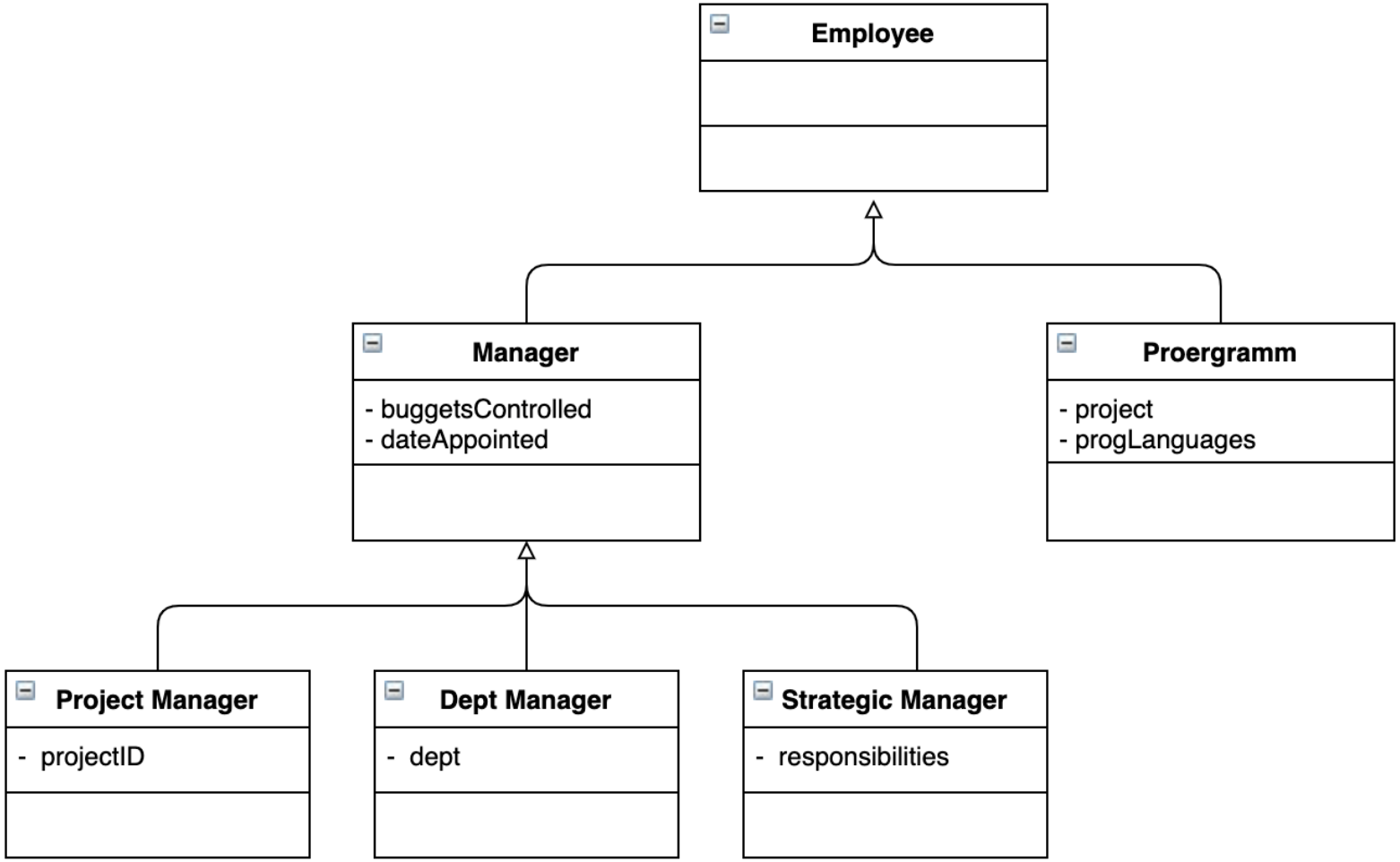


Write the objects of computers: Desktop and Tablet PC

Without Inheritance	*With Inheritance*

Write the objects of Computer, Desktop, and Tablet PC

A generalisation hierarchy



Advantages of Inheritance

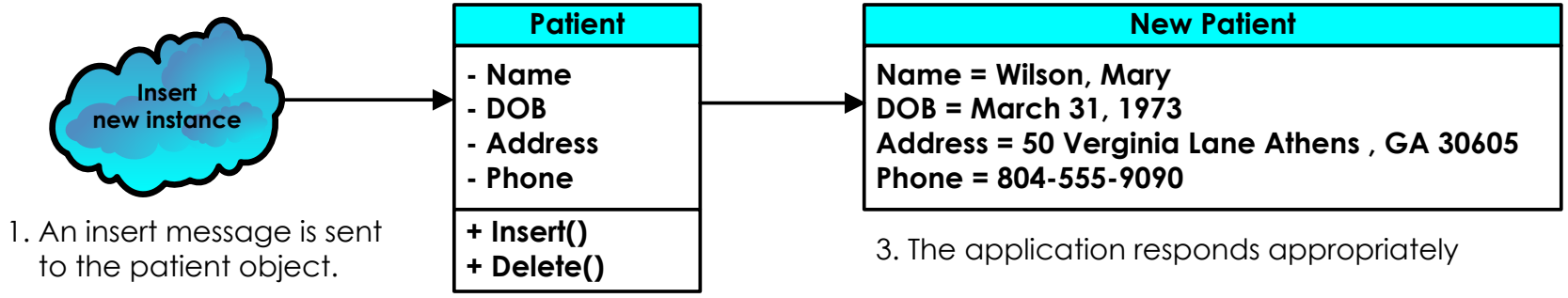
- It is an *abstraction mechanism* which may be used to classify entities
- It is a *reuse mechanism* at both design and programming level
- The inheritance graph is a source of organizational knowledge about domains and systems

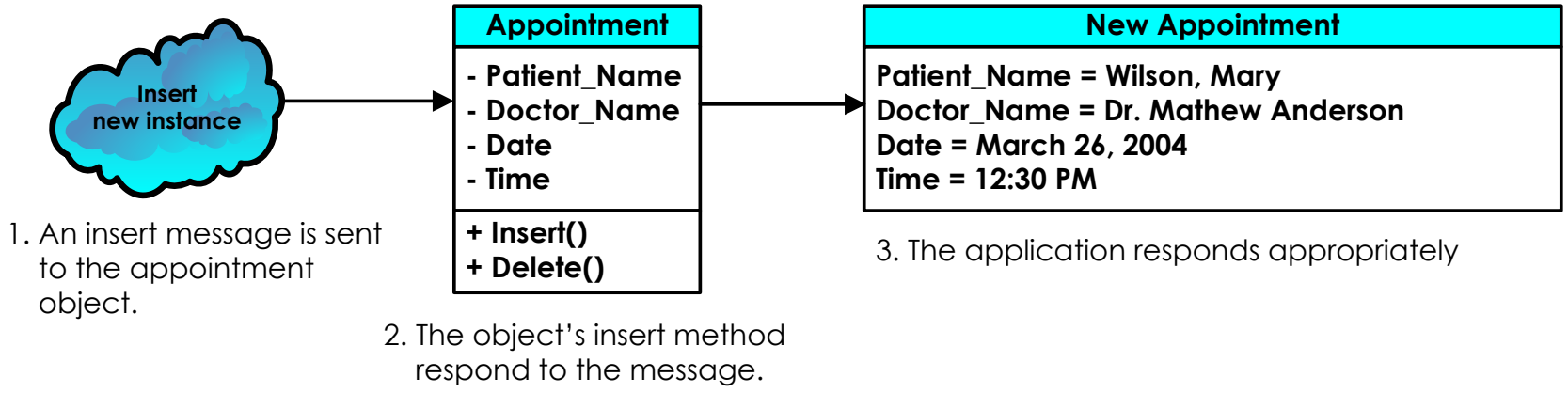
Problems with Inheritance

- Classes are not self-contained
  - They cannot be understood without reference to their super-classes
- Designers have a tendency to reuse the inheritance graph created during analysis
  - Lead to significant inefficiency
- Inheritance graphs of analysis, design and implementation have different functions and should be separately maintained

Polymorphism & Dynamic Binding

Polymorphism means that the same message can be interpreted differently by different classes of object.





Class Diagrams

Class diagram is a static model that shows the classes and the relationships among classes that remain constant in the system over time.

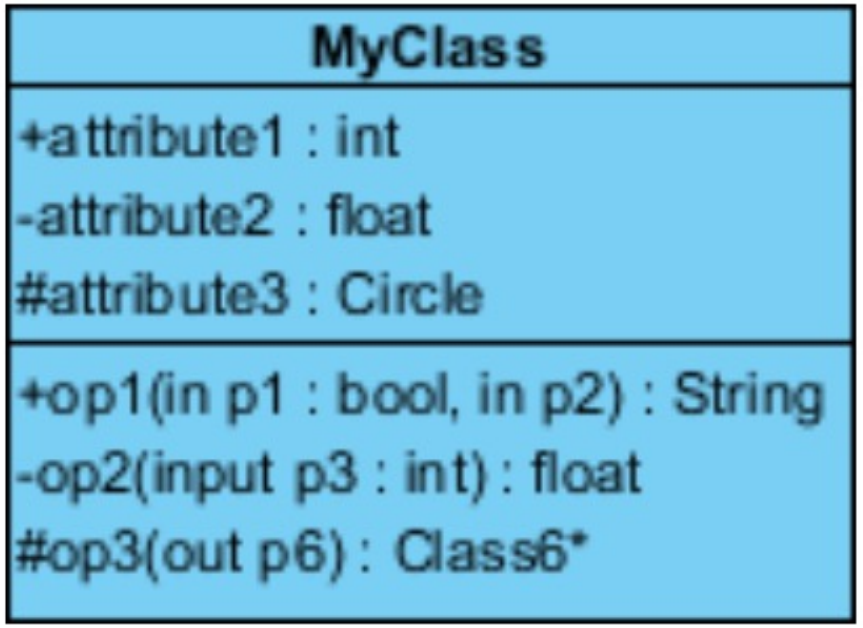
Elements of Class Diagram

Attributes are properties of the class about which we want to capture information

- Derived: slash (/)
- Public: (+)
- Protected: (#)
- Private: (-)

Operations are functions or actions that a class can perform

Class Visibility Example



In the example above:

- attribute1 and op1 of MyClassName are public
- attribute3 and op3 are protected.
- attribute2 and op2 are private.

Access for each of these visibility types is shown below for members of different classes.

Access Right	public (+)	private (-)	protected (#)	Package (~)
Members of the same class	yes	yes	yes	yes
Members of derived classes	yes	no	yes	yes
Members of any other class	yes	no	no	in same package

Operations

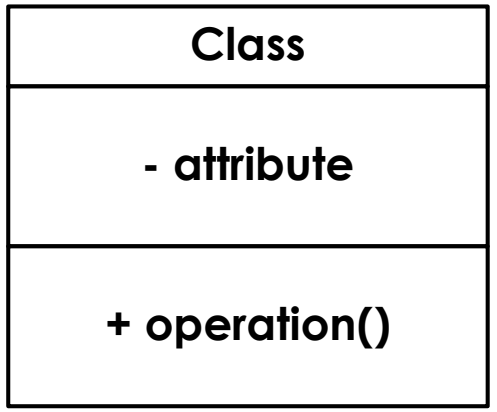
A class can contain three kinds of operations:

- Constructor Operation (create)
- Query Operation (read)
- Update Operation (update or delete)

Class Diagram Syntax

A CLASS

- Represent a kind of person, place, or thing about which the system will need to capture and store information
- Has a name typed in bold and centered in its top compartment
- Has a list of attributes in its middle compartment
- Has a list of operations in its bottom compartment



Class Diagram Syntax (Cont.)

AN ATTRIBUTE

- Represent the properties that describe the state of an object
- Can be derived from other attributes and show by placing a slash before the attribute's name
- The visibility of an attribute can be public (+), protect (#), or private (-)

attribute name
-/ derived private attribute

Class Diagram Syntax (Cont.)

AN OPERATION

- Represent the actions or functions that a class can perform
- Include parentheses that may contain parameters or information needed to perform the operation

operation name()

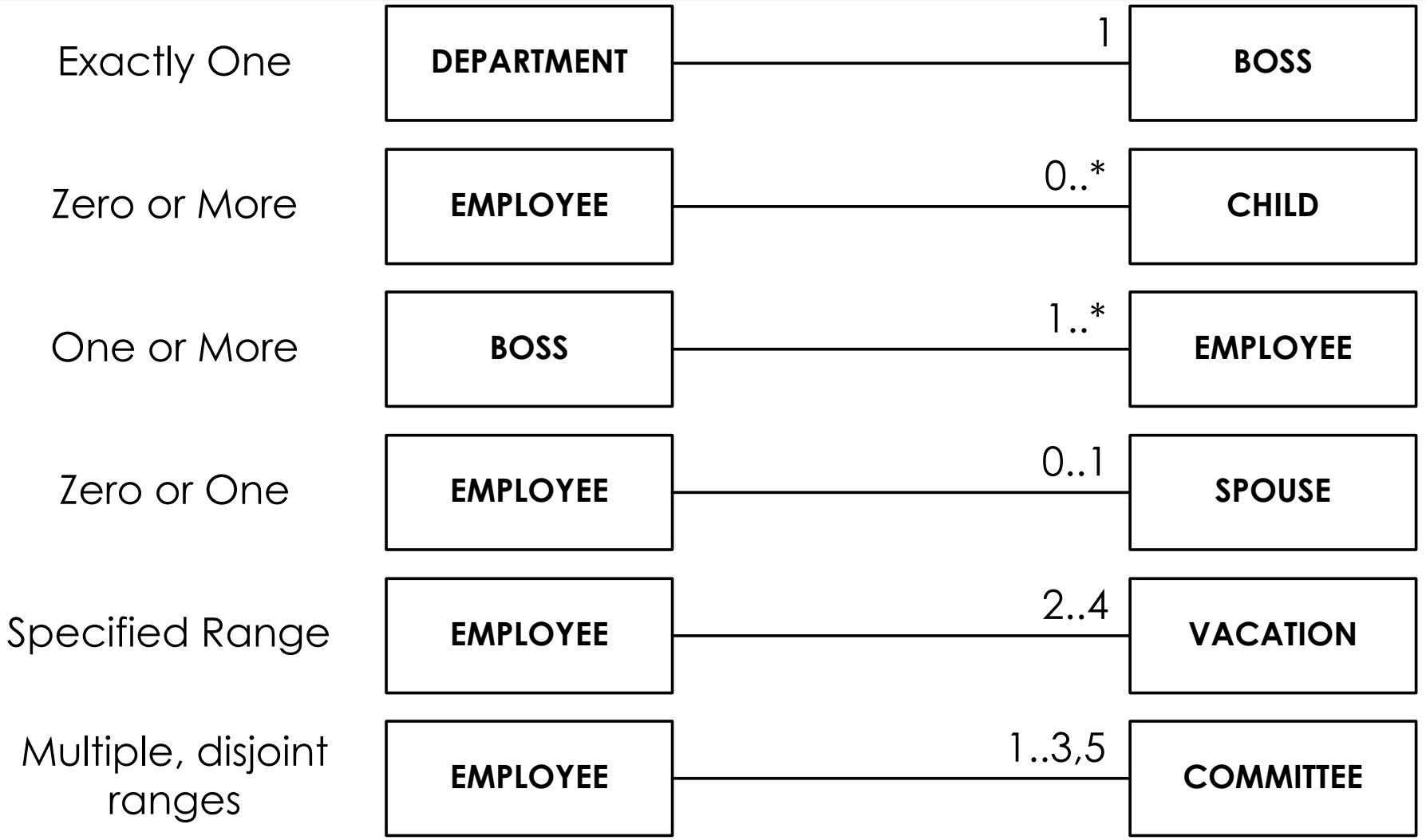
Class Diagram Syntax (Cont.)

AN ASSOCIATION

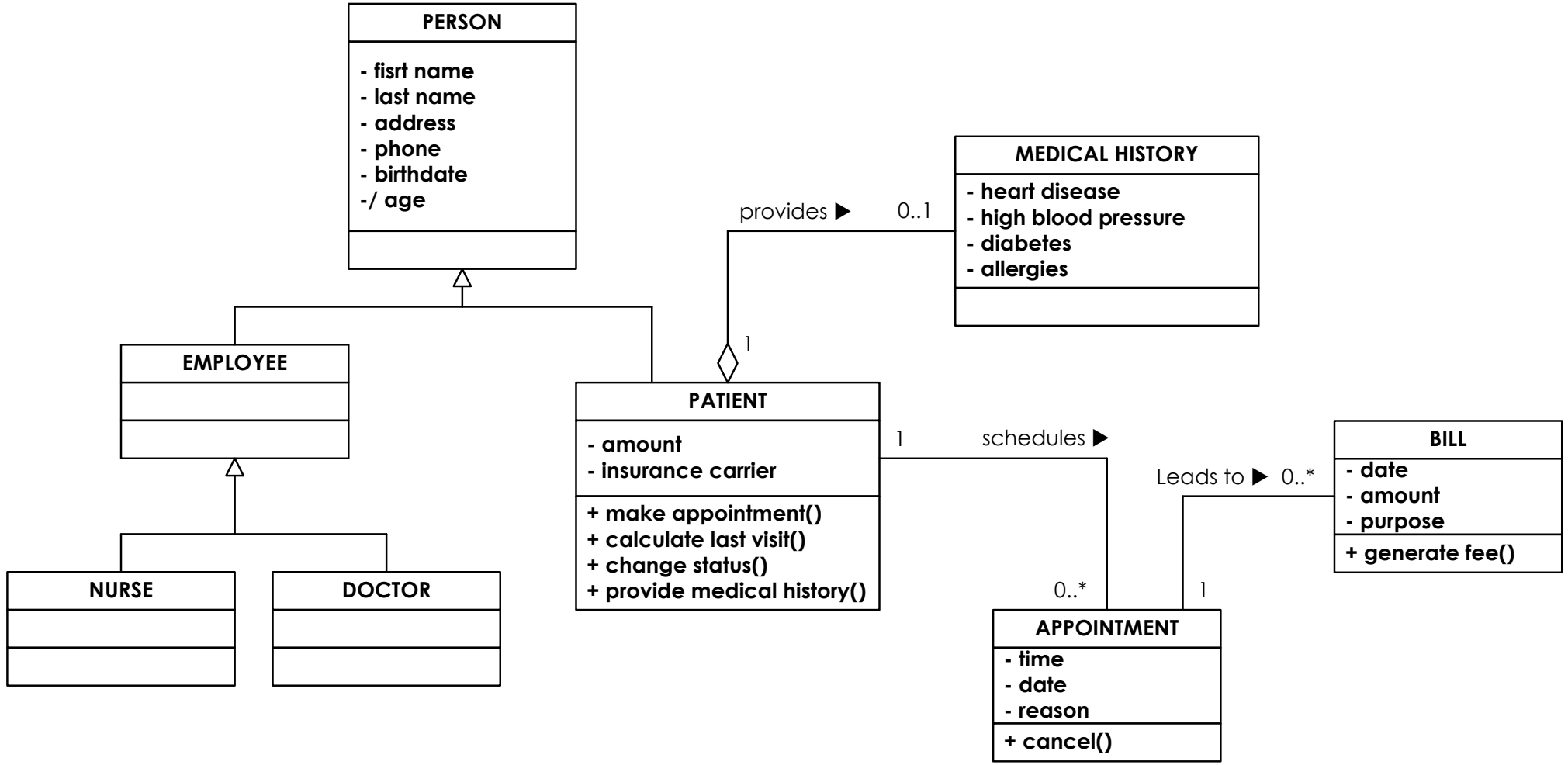
- Represent a relationship between multiple classes, or a class and itself
- It is labeled by using a verb phrase or a role name, whichever better represents the relationship
- Exist between one and more classes
- Contains multiplicity symbols, which represent the minimum and maximum number that a class instance can be associated with the related class instance

1..* Verb phase 0..1

Multiplicity



Example : Class Diagram

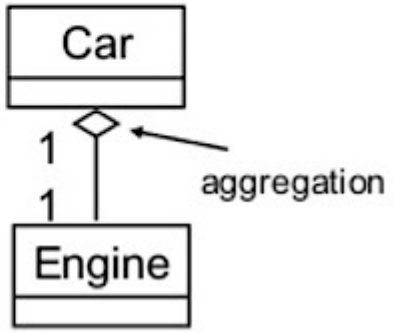


Aggregation and Composition Relationship

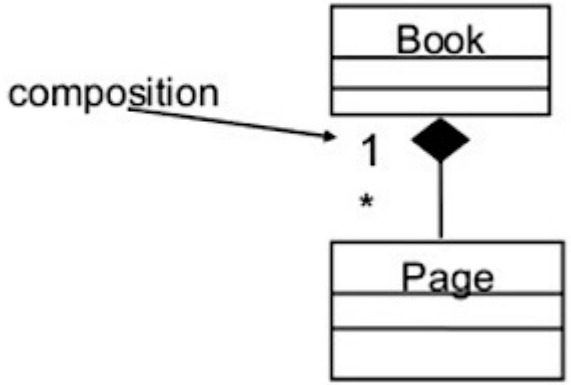
Association types

- **aggregation**: "is part of" / "has a"
  - symbolized by a clear white diamond

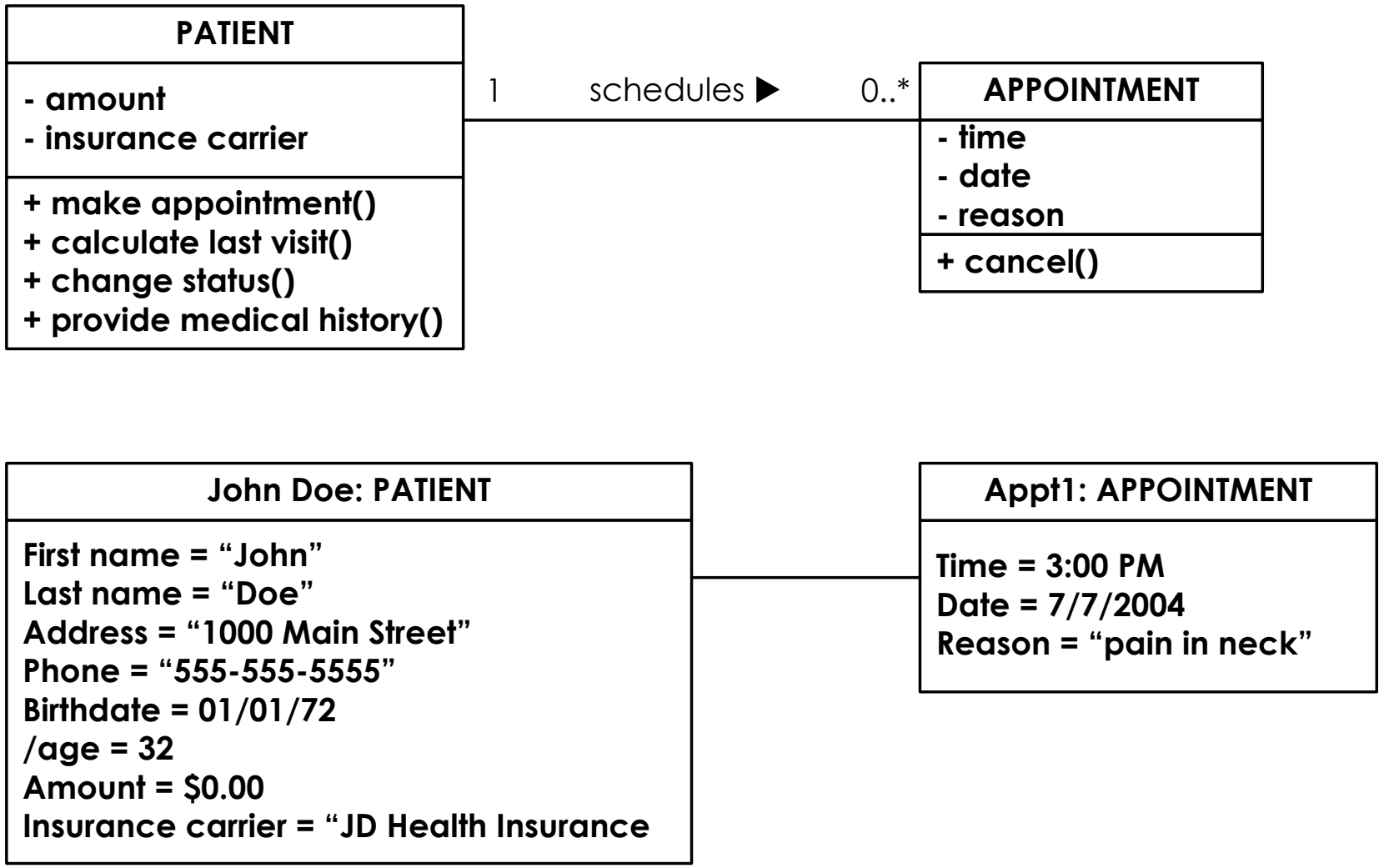
  * refers to the formation of a particular class as a result of one class being aggregated or built as a collection.



- **composition**: "is entirely made of"
  - stronger version of aggregation
  - the parts live and die with the whole
  - symbolized by a black diamond



Example : Object Diagrams



Example : Write the Medical History Object and Bill Object

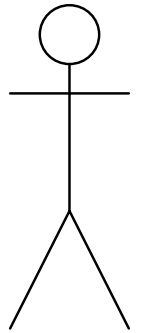
Sequence Diagram

- Sequence diagrams is a behavioral model to describe the internal dynamic aspects of an information system that supports the business processes in an organization
- Sequence diagrams are one of two types of interaction diagrams
- They illustrate the objects that participate in a use case and the messages that pass between them over time for one use case

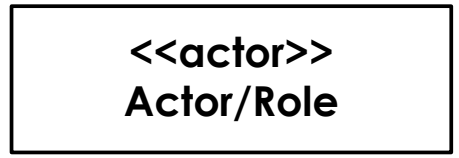
Sequence Diagram Syntax

AN ACTOR

- Person or system that derives benefit from and is external to the system
- Stick figure (default) or if a non-human actor is involved, as a rectangle with <<actor>> in it (alternative)
- Participate in a sequence by sending and/or receiving messages
- Place across the top of the diagram



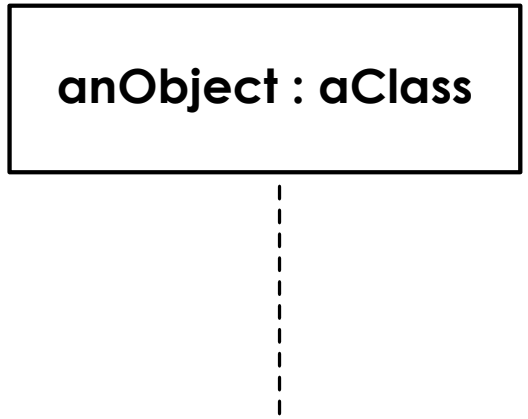
anActor



Sequence Diagram Syntax (Cont.)

AN OBJECT

- Participate in a sequence by sending and/or receiving messages
- Place across the top of the diagram



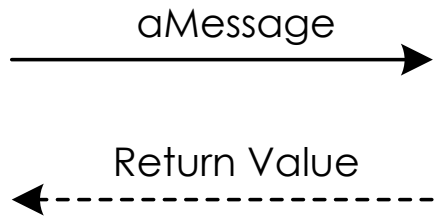
Sequence Diagram Syntax (Cont.)

AN EXECUTION OCCURANCE

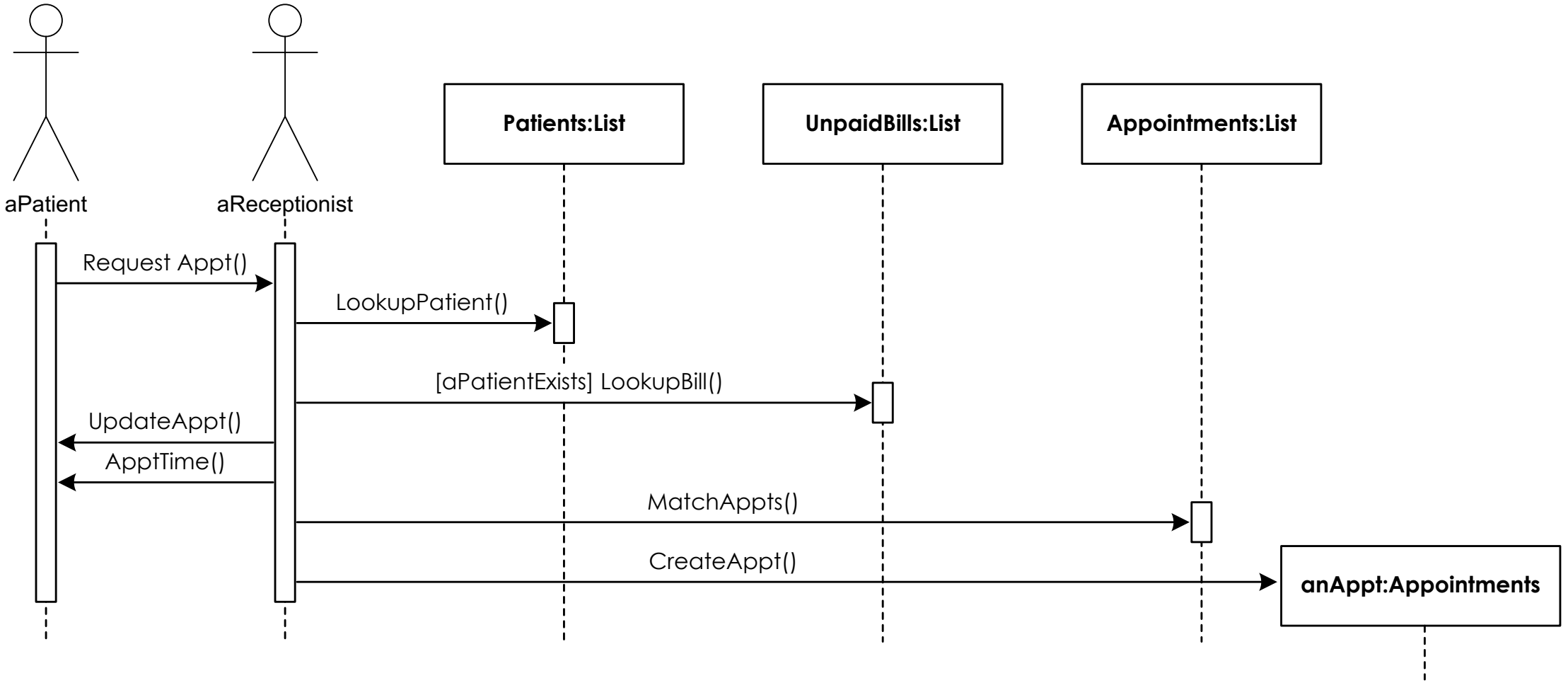
- Long narrow rectangle placed atop a lifeline
- Denotes when an object is sending or receiving message

A MESSAGE

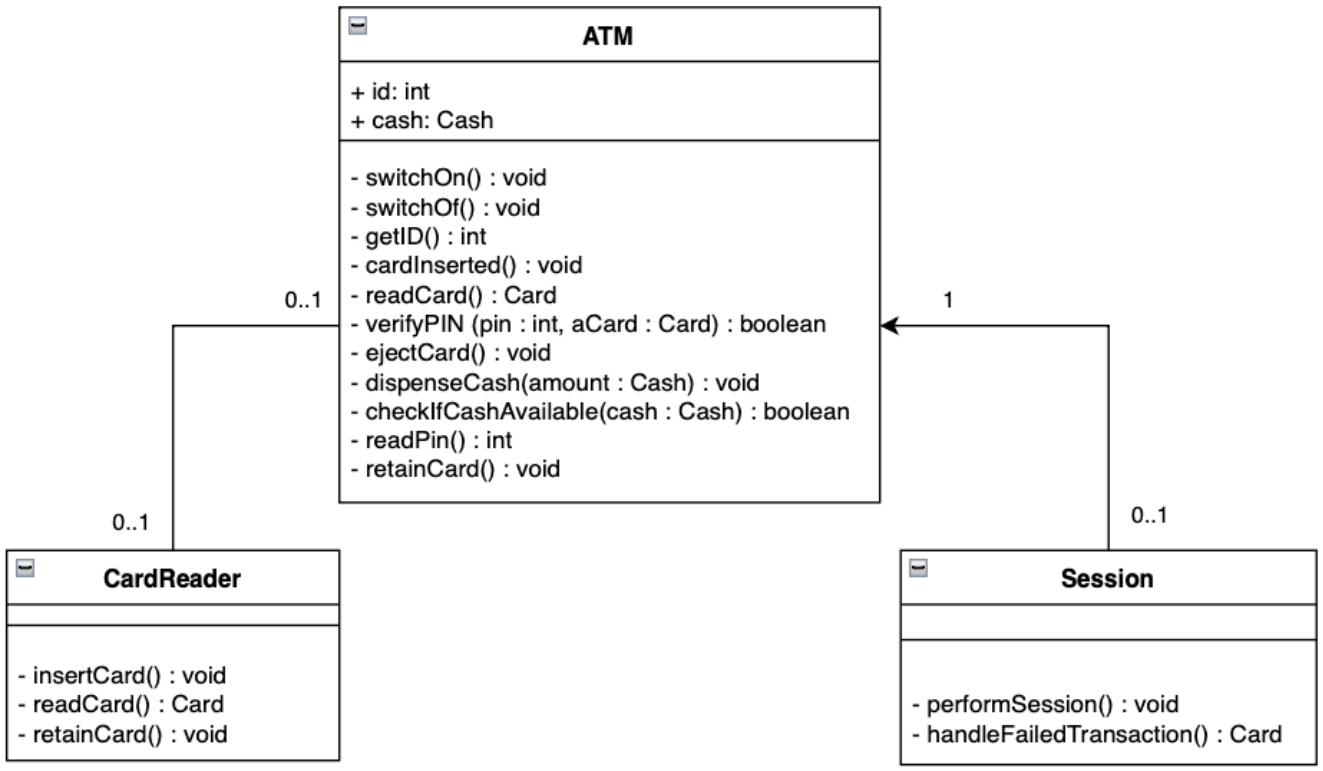
- Convey information from one object to another one
- An operation call is labeled with the message being sent and a solid arrow, which a return is labeled with the value being returned and show as a dash arrow

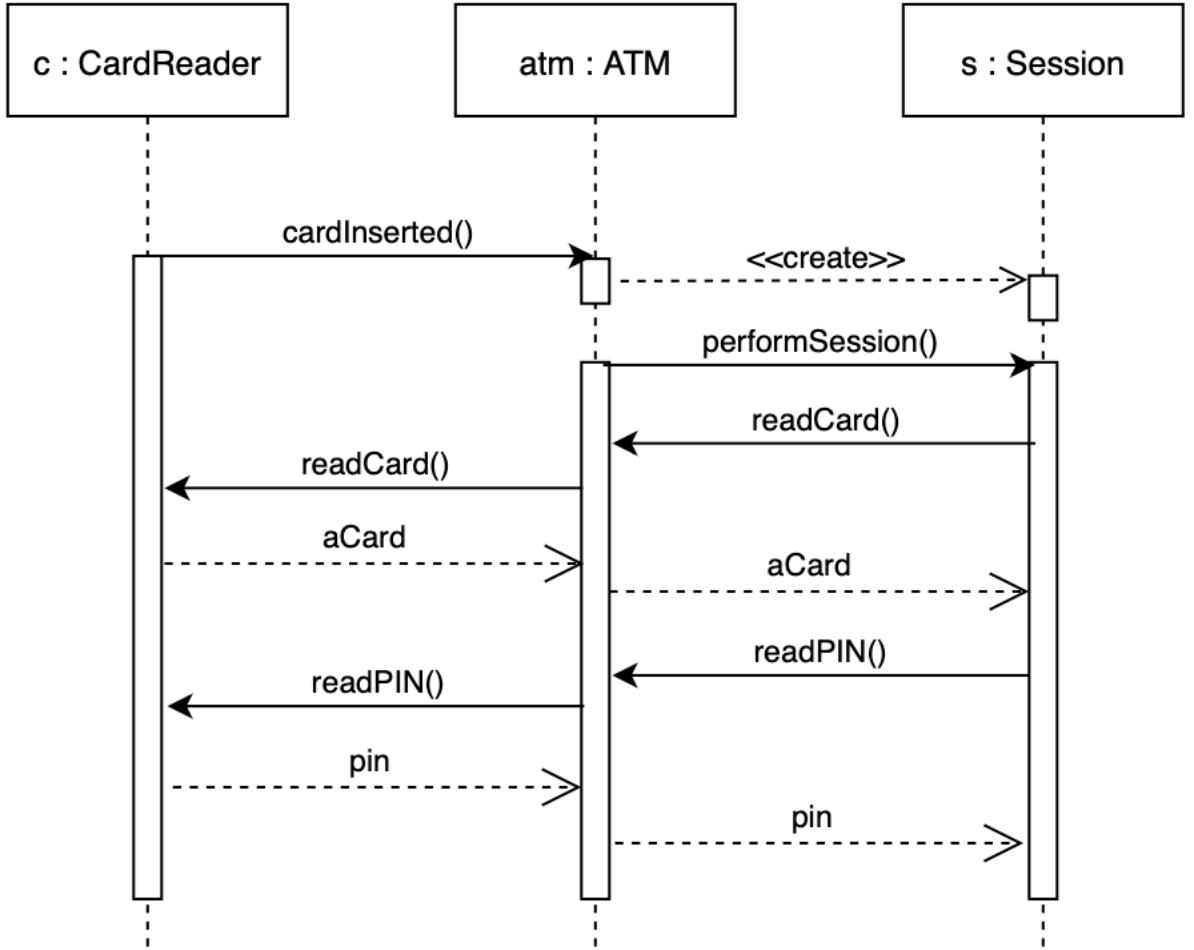


Example: Sequence Diagram

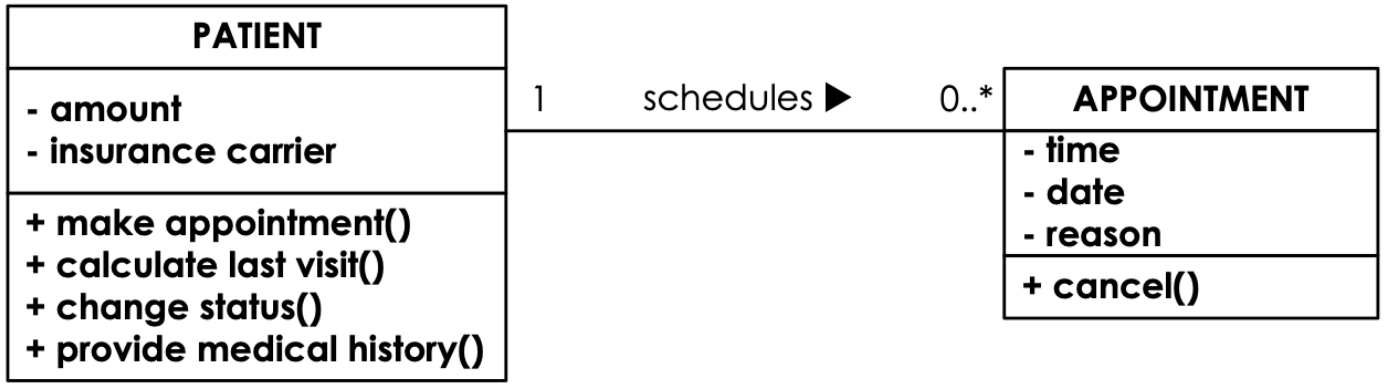


Example : Relationship between Class and Sequence Diagram





Practice : Draw the Sequence Diagram of “Patient make appointment”



Draw diagram here

Software Architecture

- The structure or organization of program components (modules)
- The manner in which these components' interaction
- The structure of data that are used by the components

Software architecture indirect refers to:

- Overall structure of the software
- The ways in which that structure provides conceptual integrity for a system

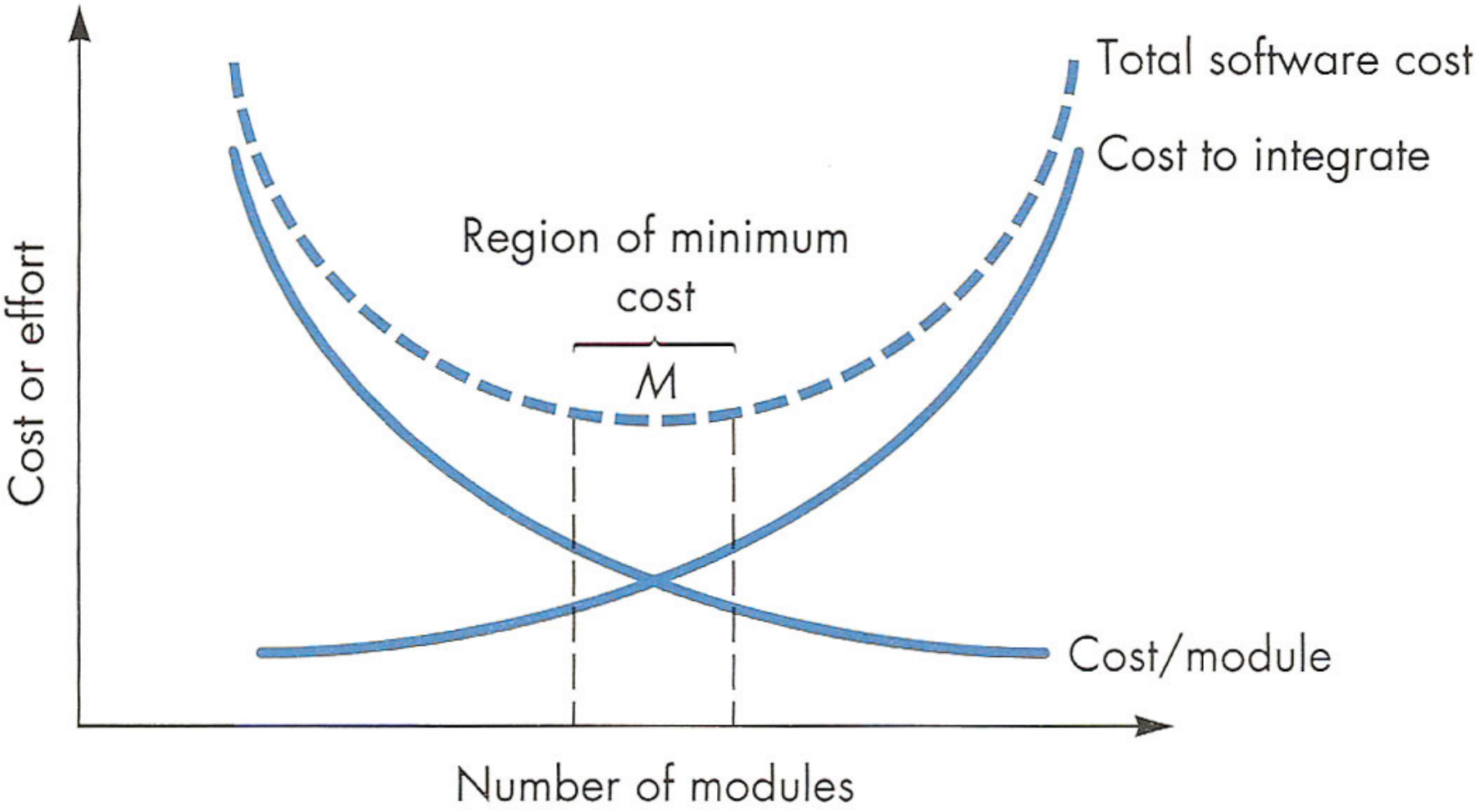
Patterns

- Each design pattern is to provide a description that enables a designer to determine:
  - Whether the pattern is applicable to the current work
  - Whether the pattern can be reused
  - Whether the pattern can serve as a guide for developing a software

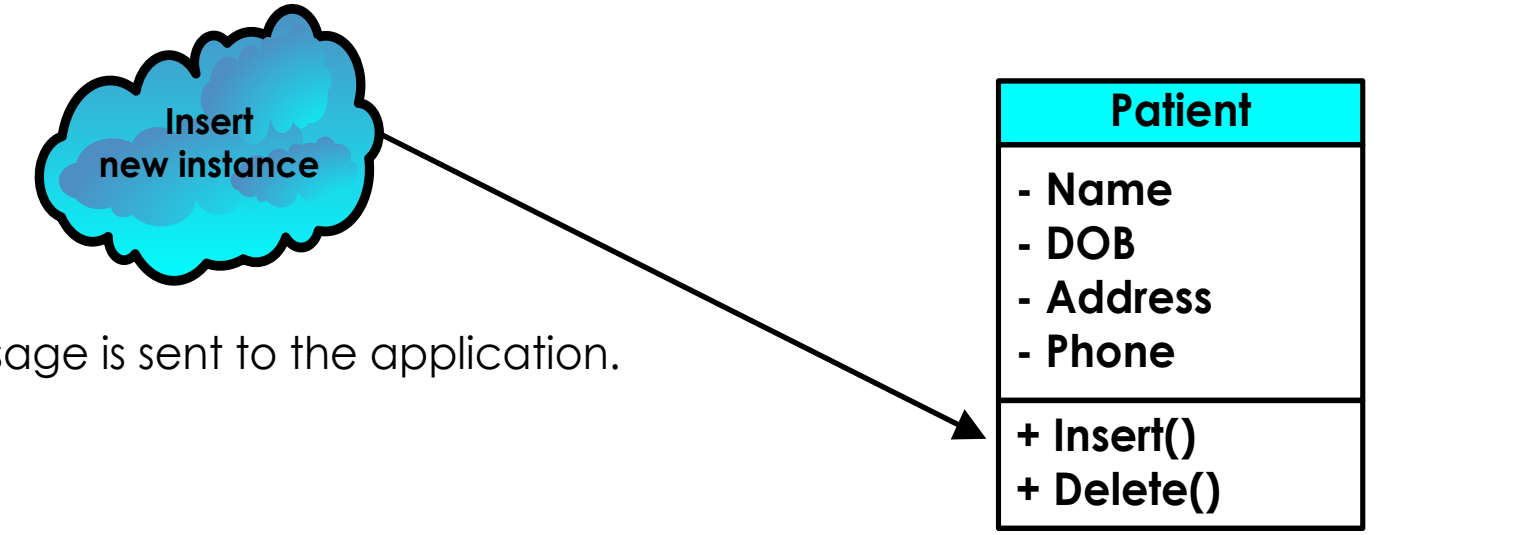
Modularity

- Software is divided into separately components
- Modules are integrated to satisfy problem requirements

Modularity and Software Cost [1]



Methods and Messages (Cont.)



A message is sent to the application.

The object's insert method will respond to the message and insert a new patient instance.

Information Hiding

The intent of information hiding is:

- To hide the detail of data structures and procedural processing behind a module interface

Encapsulation & Information Hiding (cont.)

```
// an example in Java
public class Employee {
   private BigDecimal salary = new BigDecimal(50000.00);
   public BigDecimal getSalary() {
      return salary;
   }
   public static void main() {
      Employee e = new Employee();
      BigDecimal sal = e.getSalary();
   }
}
```

Encapsulation & Information Hiding (cont.)

```
// an example in Java
//Class is the encapsulation
public class Employee {
    private BigDecimal salary = new BigDecimal(50000.00);
    // Information hiding from outside due to the private variable
    public BigDecimal getSalary() {
        return salary;
    }
    public static void main() {
        Employee e = new Employee();
        BigDecimal sal = e.getSalary();
    }
```

Encapsulation & Information Hiding (cont.)

- Encapsulation in non-object-oriented languages
- Ex. A structure can be declared in the public API (i.e., the header file) for a set of functions
  - Operate on an item of data containing data members that are not accessible to clients of the API:

```
// Header file "api.h"

struct Entity;  // Opaque structure with hidden members

// API functions that operate on 'Entity' objects

extern struct Entity * open_entity(int id);

extern int process_entity(struct Entity *info);

extern void close_entity(struct Entity *info);
```

Encapsulation & Information Hiding (cont.)

- Clients call the API functions to allocate, operate on, and de-allocate objects.
- The contents of this type are known and accessible only to the implementation of the API functions;
  - Clients cannot directly access its contents

```
// Implementation file "api.c"
#include "api.h"    // Complete definition of
                       the 'Entity' object
struct Entity {
   int ent_id;    // ID number
   char ent_name[20]; // Name
    ... and other members ...
};
```

```
// API function implementations
struct Entity * open_entity(int id) { ... }
int process_entity(struct Entity *info) { ... }
void close_entity(struct Entity *info) { ... }
```

Functional Independence

The concept of functional independence is a direct consequence of *modularity* and the *concepts of abstraction* and *information hiding*

Functional Independence is assessed by using 2 qualitative criteria: *cohesion* and *coupling*

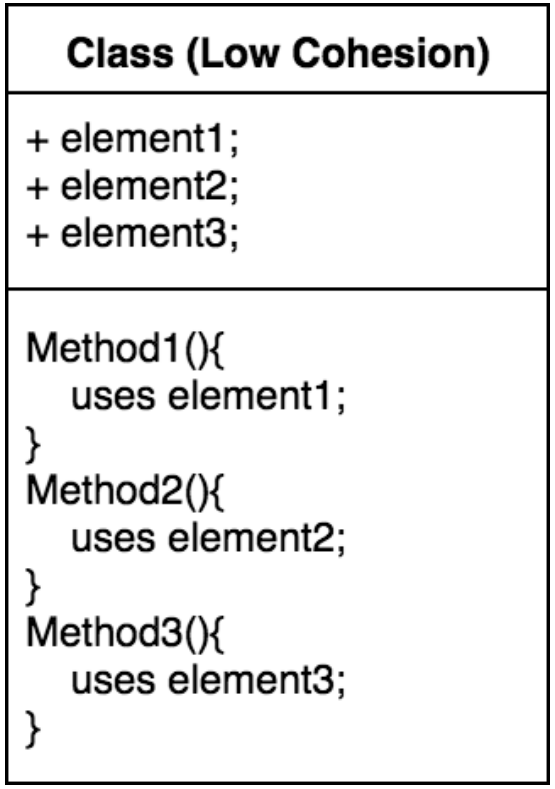
- **Cohesion** is an indication of the relative functional strength of a module
- **Coupling** is an indication of the relative interdependence among module

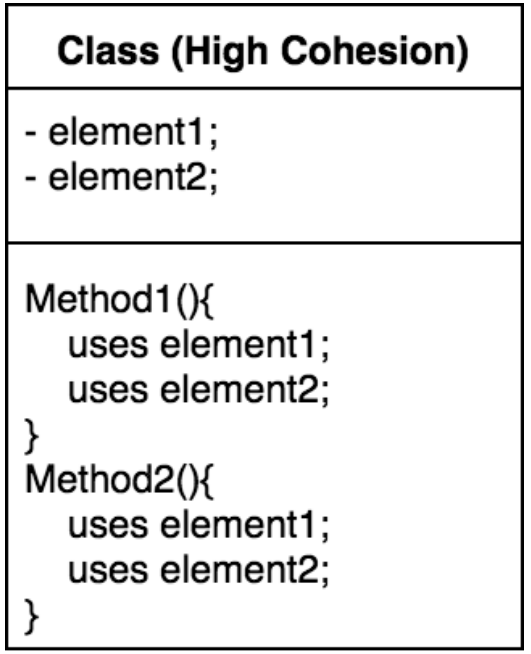
Cohesion

- The degree to which the elements of a certain module belong together
- A measure of how strongly related each piece of functionality expressed by the source code of a software module

Cohesion

- The degree to which the elements of a certain module belong together
- A measure of how strongly related each piece of functionality expressed by the source code of a software module





Cohesion

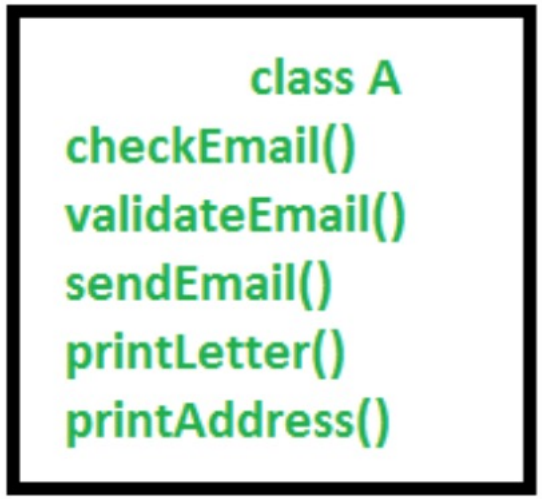
There are 3 cases where methods are low in “cohesiveness”:

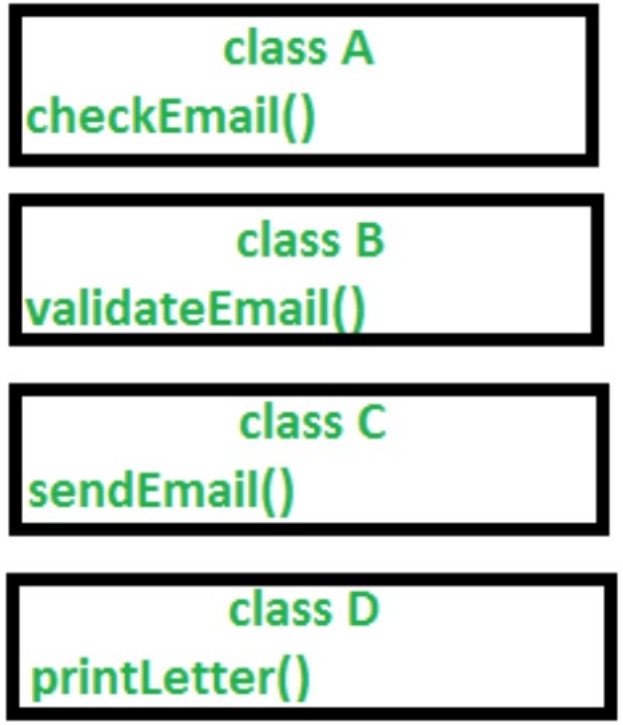
- Case 1: Method is not related to the class at all
- Case 2: Utility Class
- Case 3: Hidden objects and subclasses

Reference: https://thebojan.ninja/2015/04/08/high-cohesion-loose-coupling/

```
1:    class HighCohesion
2:    {
3:      private int _elementA;
4:      private int _elementB;
5:
6:      public int MethodA()
7:      {
8:        var returnValue = SomeOtherMethod(_elementA);
9:        return SomeVeryOtherMethod(_elementB);
10:     }
11:
12:     public void PrintValues()
13:     {
14:       Console.WriteLine(_elementA);
15:       Console.WriteLine(_elementB);
16:     }
17:   }
```

Which one is high cohesion?





Coupling

- **Coupling** is how much one component knows about the inner workings or inner elements of another one, i.e. how much knowledge it has of the other component.
- **Loose coupling** is a method of interconnecting the components in a system or network so that those components, depend on each other to the least extent practically possible.
- **Tight coupling** is where components are so tied to one another, that you cannot possibly change the one without changing the other.

Reference: https://thebojan.ninja/2015/04/08/high-cohesion-loose-coupling/

Coupling

Tight Coupling

Class 1
+ elementA;

MethodA(){
if(elementA)
return uses Class2().***elementB***;
}
MethodC(){
uses Class2().***MethodB()***;
}

Class 2
+ ***elementB***;

MethodB(){
....
}

- Tight coupling is where components are so tied to one another, that you cannot possibly change the one without changing the other.

Reference: https://thebojan.ninja/2015/04/08/high-cohesion-loose-coupling/

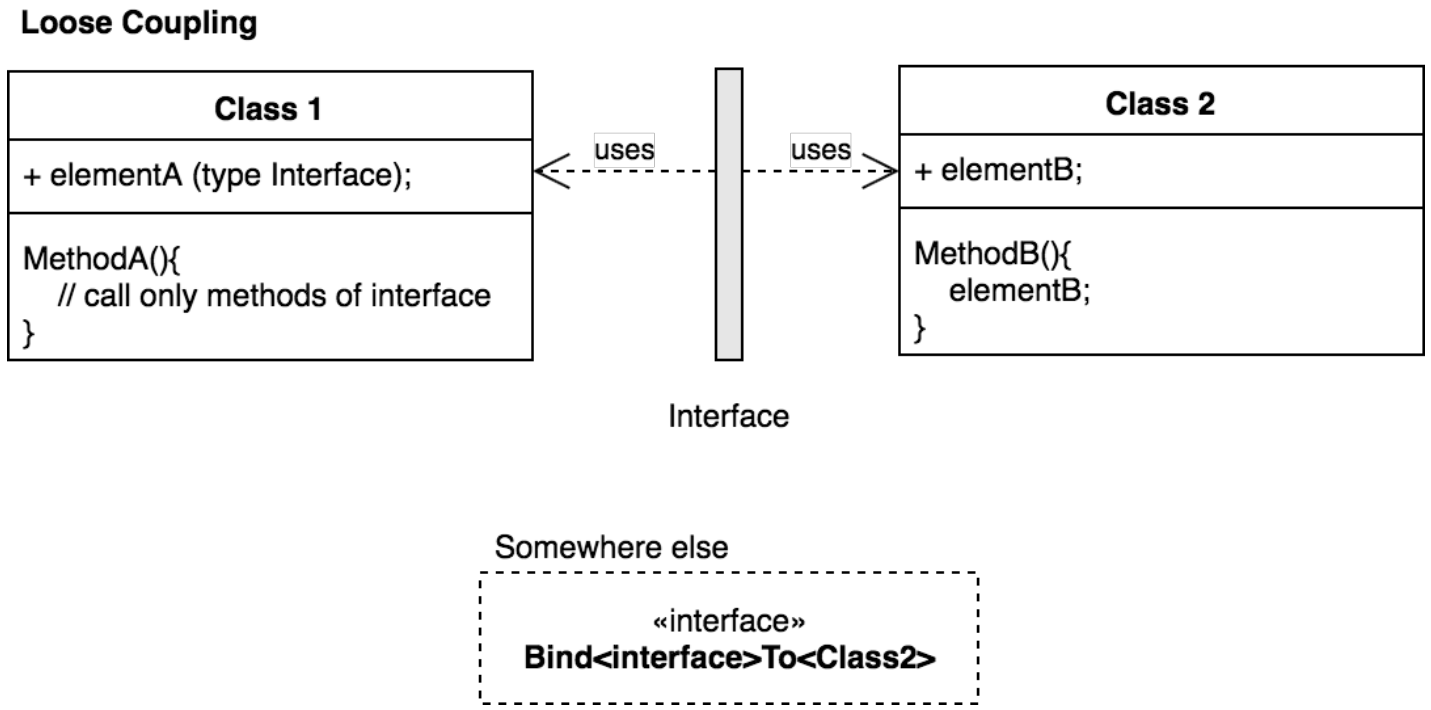
```
1:    class ClassA
2:    {
3:      private bool _elementA;
4:
5:      public int MethodA()
6:      {
7:        if (_elementA)
8:          return new ClassB()._elementB;
9:
10:       return 0;
11:     }
12:
13:     public void PrintValues()
14:     {
15:       new ClassB().MethodB();
16:     }
17:   }
18:
19:   class ClassB
20:   {
21:     public int _elementB;
22:
23:     public void MethodB()
24:     {
25:       Console.WriteLine(_elementB);
26:     }
27:   }
```

High or Low Coupling

Reference: https://thebojan.ninja/2015/04/08/high-cohesion-loose-coupling/

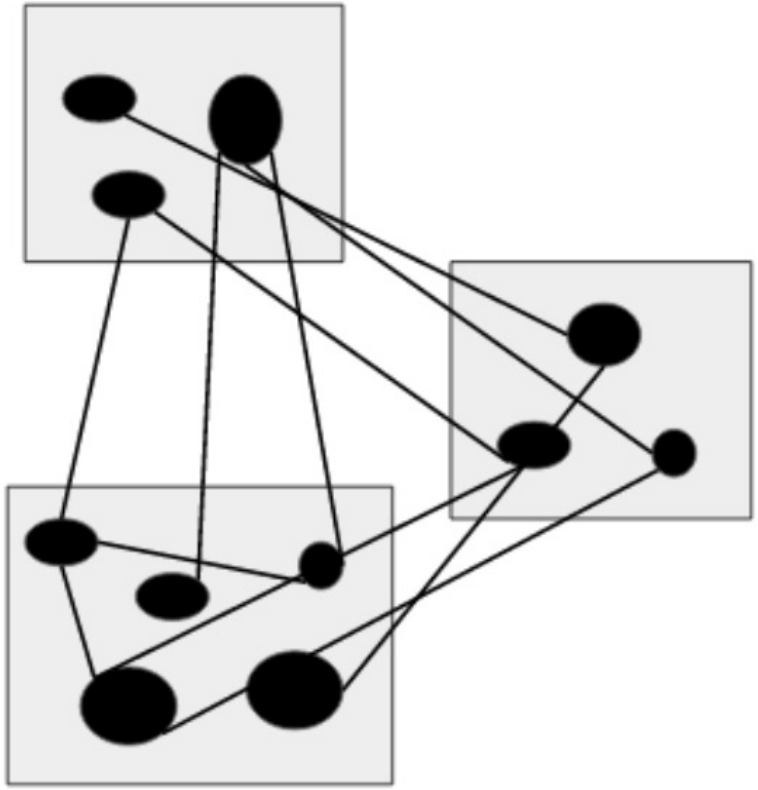
Coupling

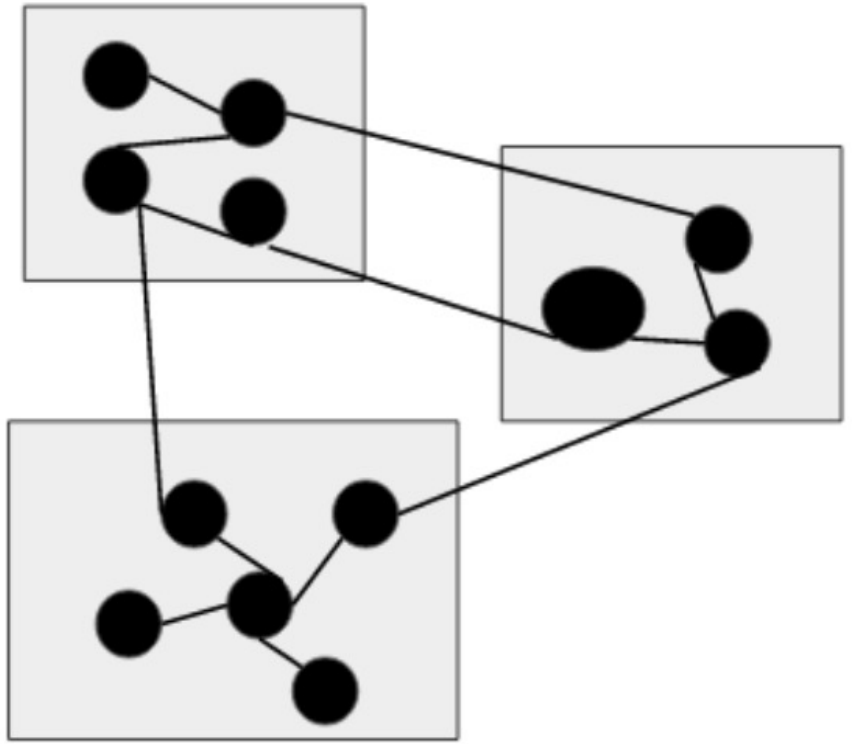
Loose coupling is a method of interconnecting the components in a system or network so that those components, depend on each other to the least extent practically possible.




```
1:  class ClassA
2:    {
3:      private readonly ISomeInterface _interfaceImpl;
4:
5:      public ClassA(ISomeInterface interfaceImpl)
6:      {
7:        _interfaceImpl = interfaceImpl;
8:      }
9:
10:      public int MethodA()
11:      {
12:        return _interfaceImpl.MethodB();
13:      }
14:
15:      public void PrintValues()
16:      {
17:        _interfaceImpl.PrintValues();
18:      }
19:    }
20:
21:    interface ISomeInterface
22:    {
23:      int MethodB();
24:      void PrintValues();
25:    }
26:
27:    class ClassB : ISomeInterface
28:    {
29:      private int _elementB = 2 + 2;
30:
31:      public int MethodB()
32:      {
33:        return _elementB;
34:      }
35:
36:      public void PrintValues()
37:      {
38:        Console.WriteLine(_elementB);
39:      }
40:    }
```

Which one is a good functional dependencies? (Cohesion and Coupling)





Design Classes

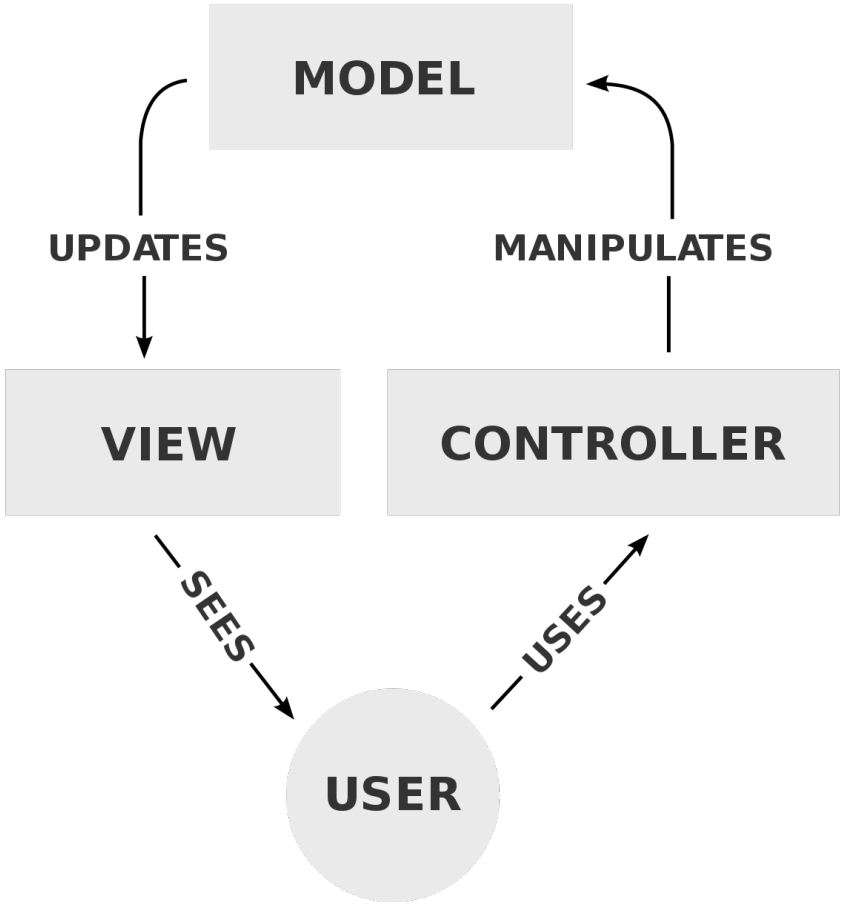
- Software team must define a set of design classes
- A set of design classes that refine the analysis classes by providing design details
- Design details:
  - Implement the classes
  - Create a new set of design classes that implement a software infrastructure to support the business solution

Design Classes (Cont.)

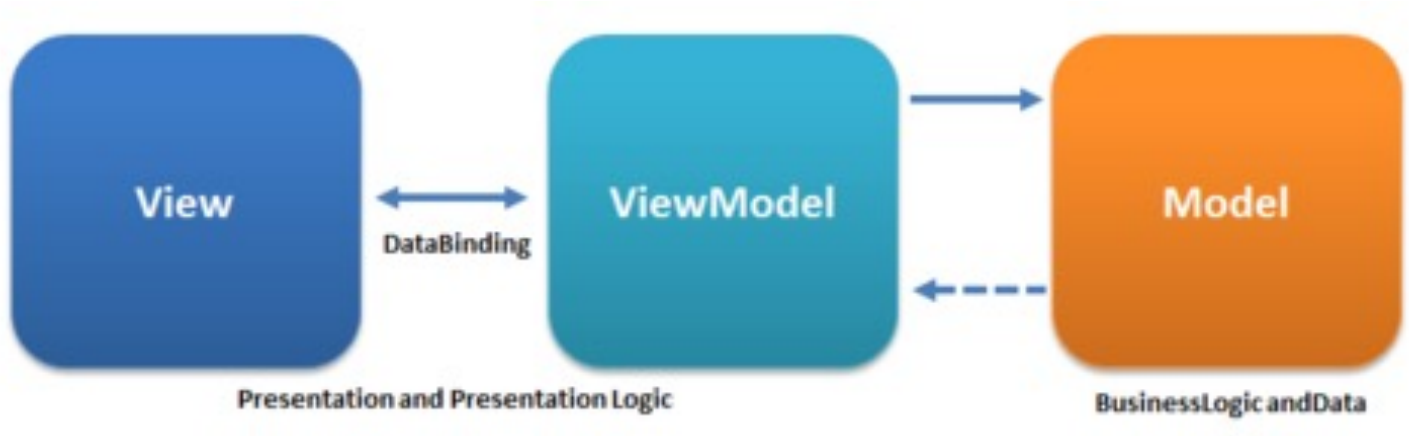
Five different types of design classes:

- User Interface Classes
- Business Domain Classes
- Process Classes
- Persistent Classes (Data Management)
- System Classes

Model-View-Control vs Model-View-View Model



MVC [1]



MVVM [2]

Reference: [1] https://en.wikipedia.org/wiki/Model–view–controller
[2] https://en.wikipedia.org/wiki/Model–view–viewmodel

Difference Between MVC and MVVM

The key differences between the MVC pattern and the MVVM pattern:

- MVC is a software architectural pattern that allows us to separate an app into three key logical components, Model, View, and Controller. In contrast, the MVVM pattern makes it possible to build the user interface of an app separately from the application logic.
- The entry point to the app in MVC is the controller, whereas the entry point to the app in MVVM is the View.
- In MVC, the Controller handles the user input, whereas, in MVVM, the view takes the user input.
- In MVC, there are ‘one to many’ relationships between Controller and View. On the other hand, MVVM has ‘one to many’ relationships between View and View Model.

Reference: https://www.sencha.com/blog/how-to-implement-mvvm-architecture-in-ext-js-web-application/

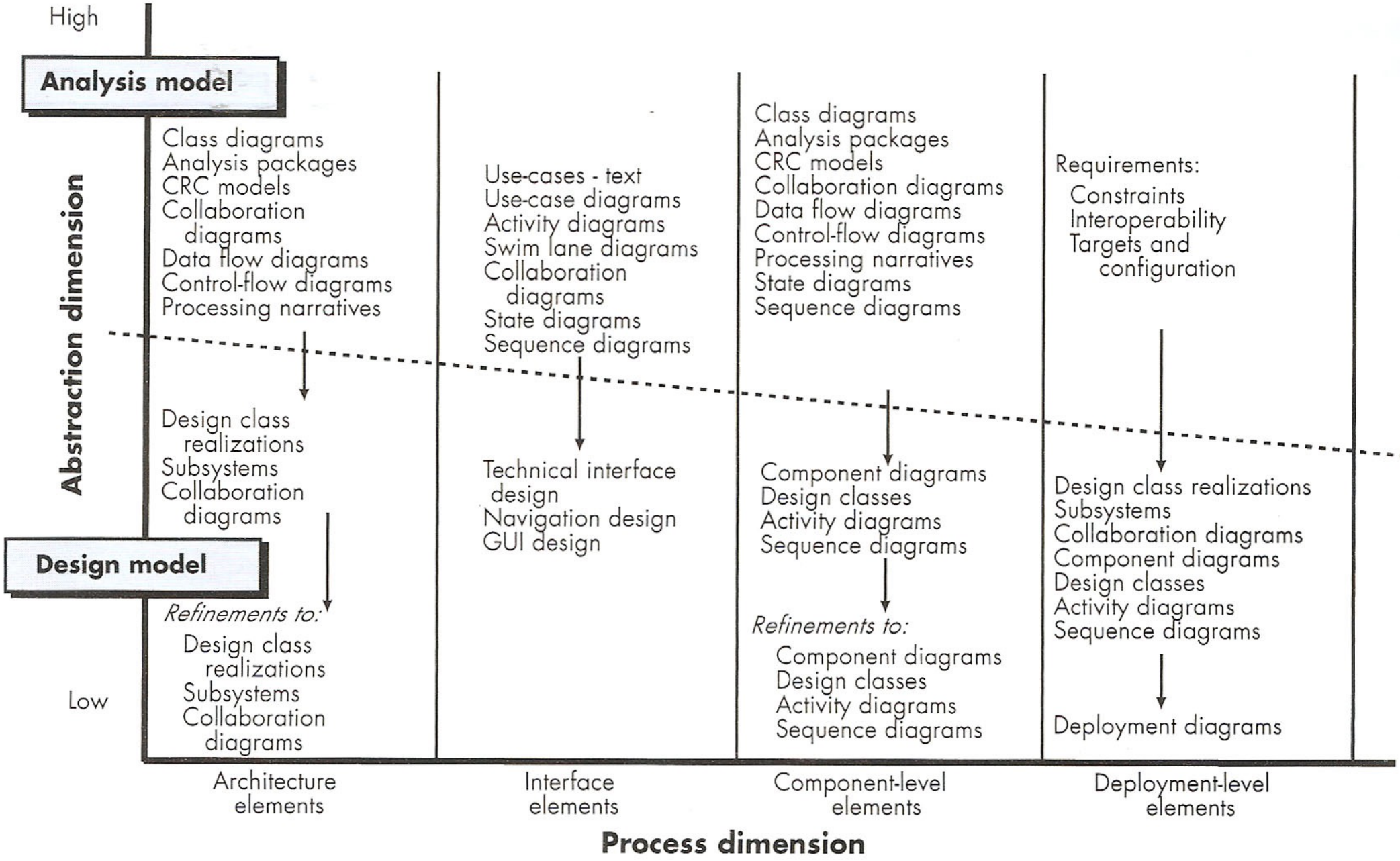
What's a "Well-Formed" design class?

Arlow and Neustadt suggest that each design class be reviewed to ensure that it is "*well-formed*"

They define *4 characteristics of a well-formed design class*:

1. Complete and Sufficient
2. Primitiveness (or Originality)
3. High Cohesion
4. Low Coupling

Design Model [1]



Data Design Elements

- **Data design** (sometimes referred to as data architecting)
- Creates a model of data and/or information that is represented as a *high level of abstraction*

Architectural Design Elements

- Architectural design for software = Floor plan of a house
- The architectural model is derived from *3 sources*
  1. **Information** about the application domain for the software to be built
  2. Specific analysis model **elements**
     - Such as data flow diagrams or analysis classes, their relationships and collaborations for the problem at hand
  3. Availability of architectural **patterns and styles**

Interface Design Elements

- Interface design for software = A set of detailed drawings (and specifications) for the doors, windows, and external utilities of a house
- There are *3 important elements* of interface design:
  1. **User Interface** (UI)
  2. **External Interfaces** to other systems, devices, networks, or other procedures or consumers of information
  3. **Internal Interfaces** between various design components

Pattern-Based Software Design

- Best designers in any fields have a weir ability to see patterns that:
  - Characterize a problem
  - Corresponding patterns that can be combined to create a solution
- A software engineer should look for every opportunity to reuse existing design patterns rather than creating new ones

Pattern-Based Software Design (cont.)

- Design patterns can be used throughout software design
- Once the analysis model has been developed:
  - Designer can examine a detailed representation of the problem to be solved and the constraints that are imposed by the problem
- The problem description is examined at various levels of abstraction to determine if it is responsive to one or more of the following types of design patterns:
  - Architectural Patterns
  - Design Patterns
  - Idioms (Coding Patterns)

References

1. Ian Sommerville, Software Engineering 10th Edition, Pearson, April 2015

Any Questions?

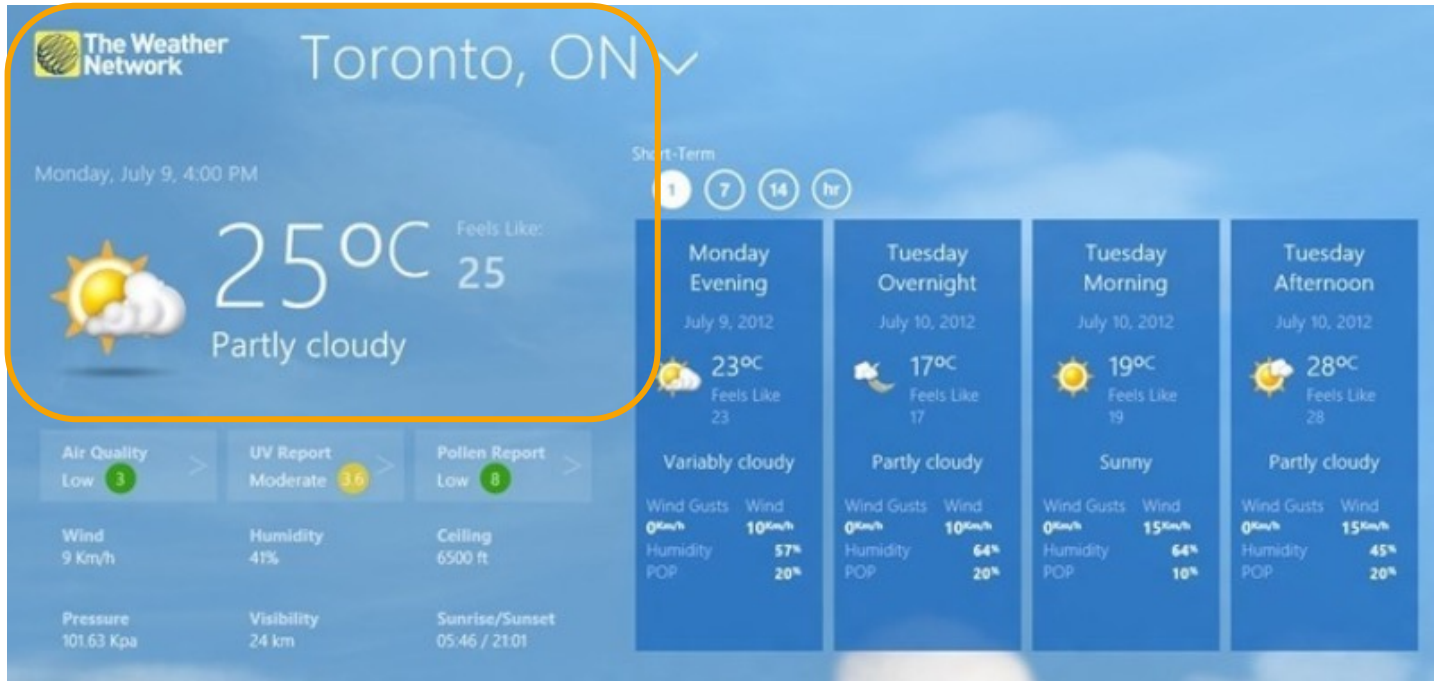
:O)

Thank you

Practice: Software Architecture Design (1)

Design class components and sequence diagram for the application below

- The weather report website
  - Assume that the weather data are ready in the database, the application can retrieve the current weather information according to the city name and date. These information can be reported on the website (online).



Practice: Software Architecture Design (2)

Design the class components and sequence diagram for the application below

- Currency Exchange Mobile Application
  - Assume that the exchange rate can be retrieved from currency web service (online). The application converts the money by submitting the amount of money and currency from both sides