Overview of Object-Oriented Technology & System Analysis and Design with UML

Yu-Chuan Lin

Prof. Min-Hsiung Hung

Intelligent Manufacturing Research Center (iMRC)
Department of Computer Science and Information Engineering
Chinese Culture University

November 13, 2019

Outline

I. Introduction to Object-Oriented Programming & Technology

II. System Analysis and Design with Unified Modeling Language (UML)

III. Example Papers containing UML Diagrams

I. Introduction
Object-Oriented Programming

Object-Oriented Programming

- **Object-Oriented Programming (OOP)** is a programming paradigm based on the concept of "objects", which can contain data and code: data in the form of fields (often known as attributes or properties), and code, in the form of procedures (often known as methods).
- A feature of objects is that an object's own procedures can access and often modify the data fields of itself (objects have a notion of this or self). In OOP, computer programs are designed by making them out of objects that interact with one another

Source: Wikipedia

CUP

Source: ikea

Class

```
int cupSize = 50cm * 50cm;
bool withHandler = true;
String cupName = "my cup";
public Array function
cupManufacture(int cupSize, bool
withHandler, String cupName){
array materialList = [cupSize,
withHandler, cupName];
return materialList;
}
```

```
Class Cup {
    String name;
    int size;
    bool withHandler;
    // contructor
    public function __constructor(String name, int
size, bool withHandler){
        this->name = name;
        this->size = size;
        this-> withHandler = withHandler;
    }
    public Array function manufacture(int size, bool
withHandler, String name){
        array materialList = [size, withHandler, name];
        return materialList;
    }
}
//New a cup in the Controller
Class Controller {
    Cup myTeaCup = new Cup("Tea", "500", true);
}
```

Source: Chun Yeung

Inheritance


```
Class Cup {
    public String name;
    public int size;
    public bool withHandler;
    public Array function manufacture(int size, bool withHandler,
    String name)
    {
        array materialList = [size, withHandler, name];
        return materialList;
    }
}
```

```
Class TeaCup extends Cup {

    public bool forTeaOnly;
    public String country;
}
```

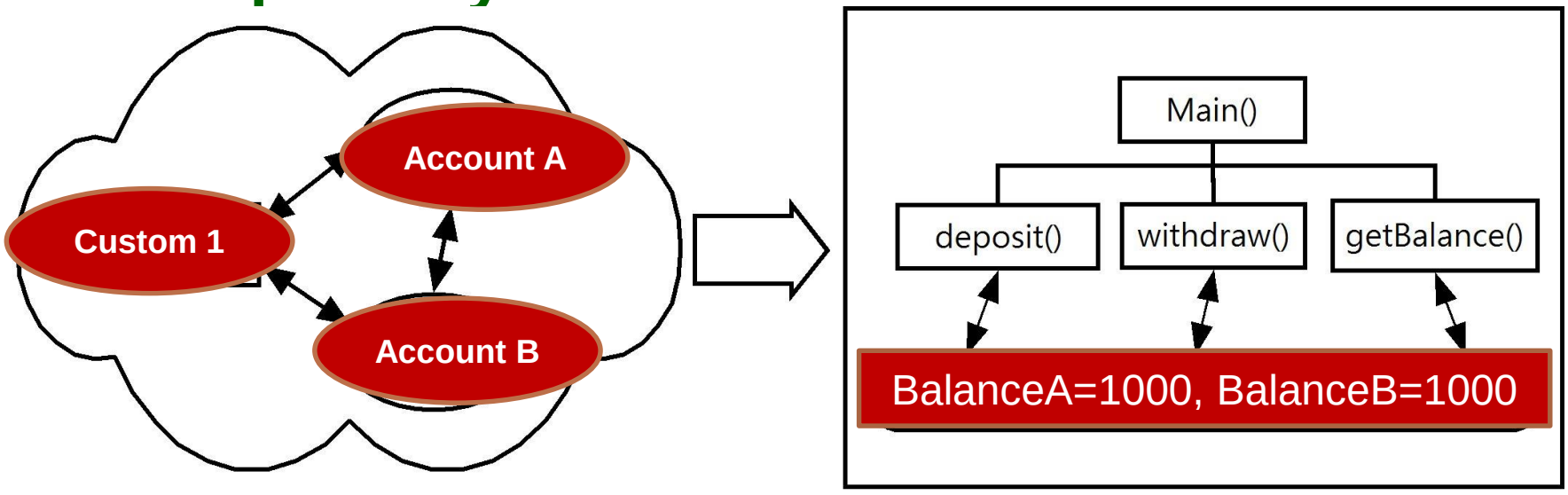
Source: Chun Yeung

Key Elements of Object-Oriented Systems

- ***Classes*** **-- template to define objects**
- ***Instances*** **-- specific examples of class members**
- ***Objects*** **– instances of a class, building block of the system**
- ***Attributes*** **-- describe data aspects of the object**
- ***Methods*** **-- the processes the object can perform**
- ***Messages*** **-- instructions sent to or received from other objects**

Traditional Application Programming

- Adopt procedure-oriented approach.
- Focus on procedures.
- Data and methods are considered separately:

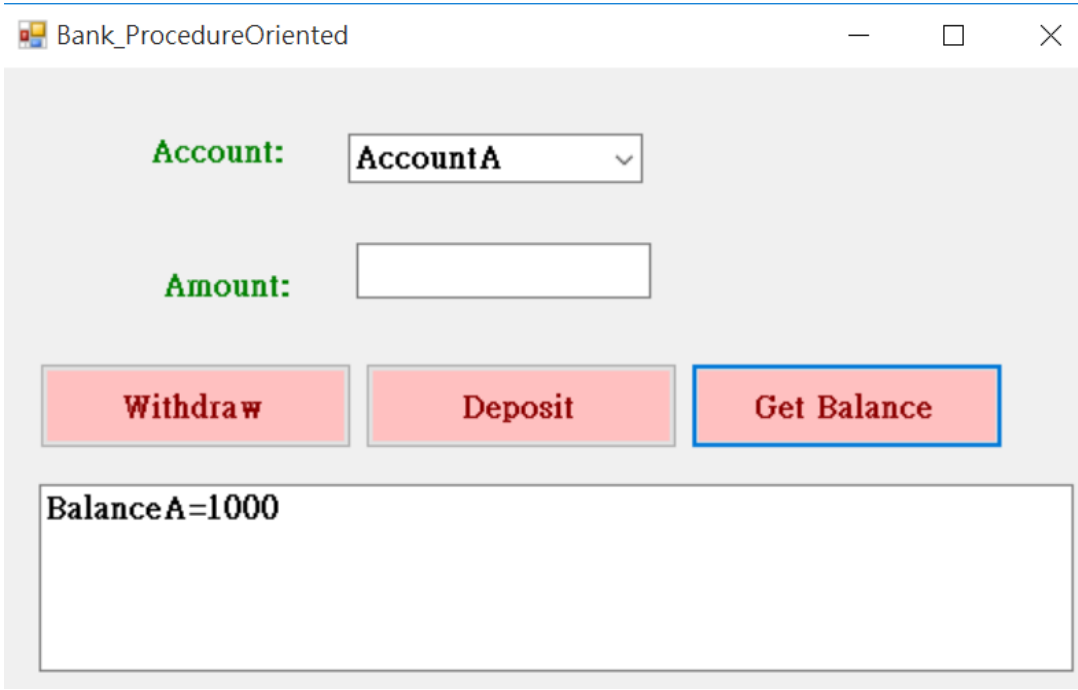


In Real World

In Software

Procedure-Oriented Example Code in C#

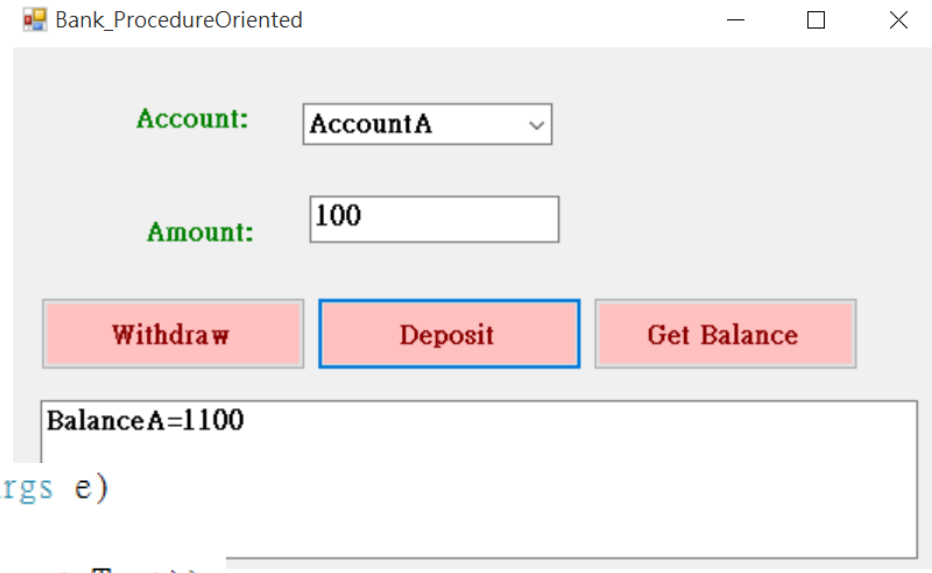

```
int BalanceA = 1000, BalanceB = 1000;

void deposit(int account, int amount)
{
    if (account == 0) BalanceA += amount;
    else BalanceB += amount;
}
//
void withdraw(int account, int amount)
{
    if (account == 0) BalanceA -= amount;
    else BalanceB -= amount;
}
//
int getBalance(int account)
{
    if (account == 0) return BalanceA;
    else return BalanceB;
}
```



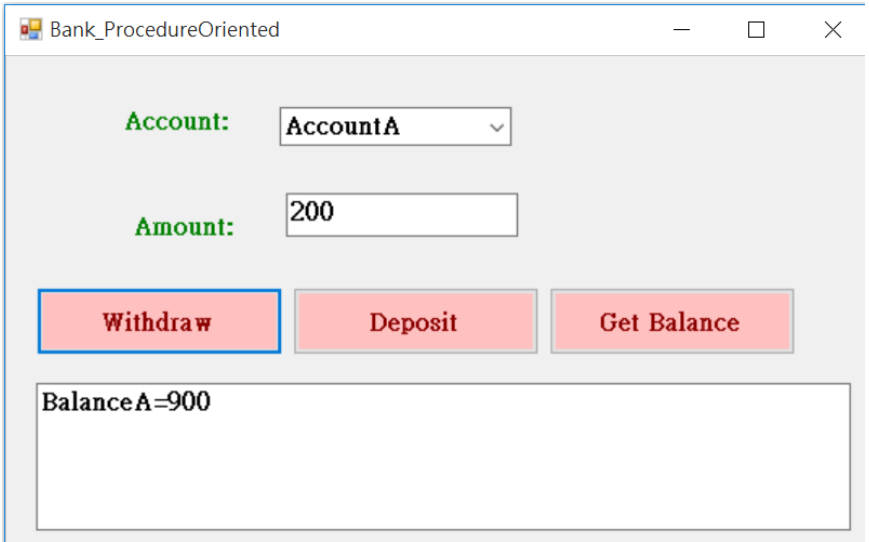

```
private void btnGetBalance_Click(object sender, EventArgs e)
{
    if (cbAccount.SelectedIndex == 0)
        tbResult.Text = "BalanceA=" + getBalance(cbAccount.SelectedIndex).ToString();
    else
        tbResult.Text = "BalanceB=" + getBalance(cbAccount.SelectedIndex).ToString();
}
```

Procedure-Oriented Example Code in C#


```
private void btnDeposit_Click(object sender, EventArgs e)
{
    deposit(cbAccount.SelectedIndex, int.Parse(tbAmount.Text));
    if (cbAccount.SelectedIndex == 0)
        tbResult.Text = "BalanceA=" + BalanceA;
    else
        tbResult.Text = "BalanceB=" + BalanceB;
}
```

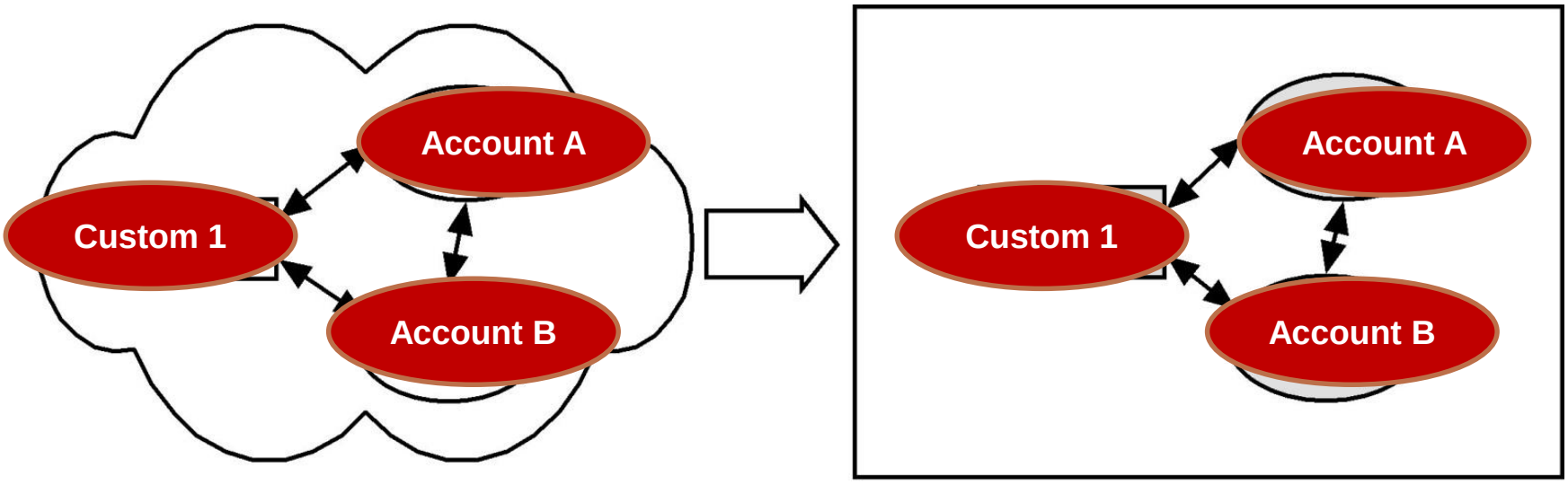



```
private void btnWithdraw_Click(object sender, EventArgs e)
{
    withdraw(cbAccount.SelectedIndex, int.Parse(tbAmount.Text));
    if (cbAccount.SelectedIndex == 0)
        tbResult.Text = "BalanceA=" + BalanceA;
    else
        tbResult.Text = "BalanceB=" + BalanceB;
}
```



Object-Oriented Application Programming

- We are in the world consisting of objects.
- Encapsulate data and methods into a container, i.e. class.
- System is composed of objects and messages among objects.



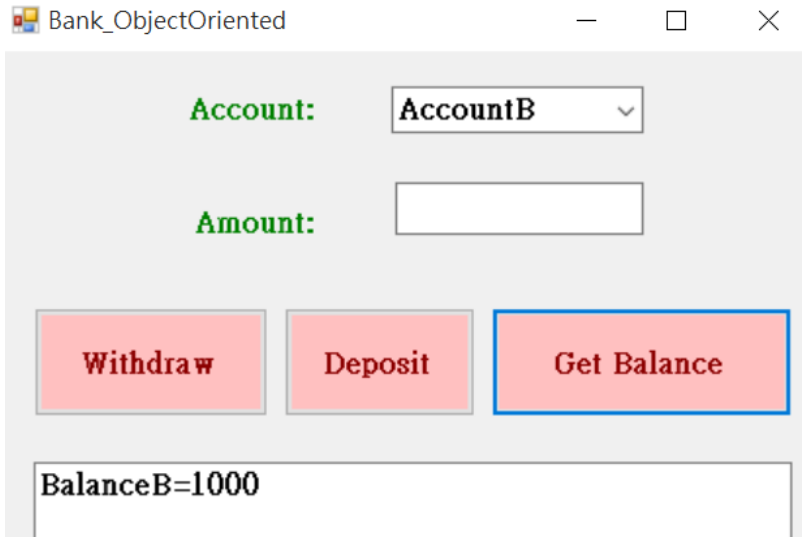
In Real World

In Software

Object-Oriented Example Code in C#

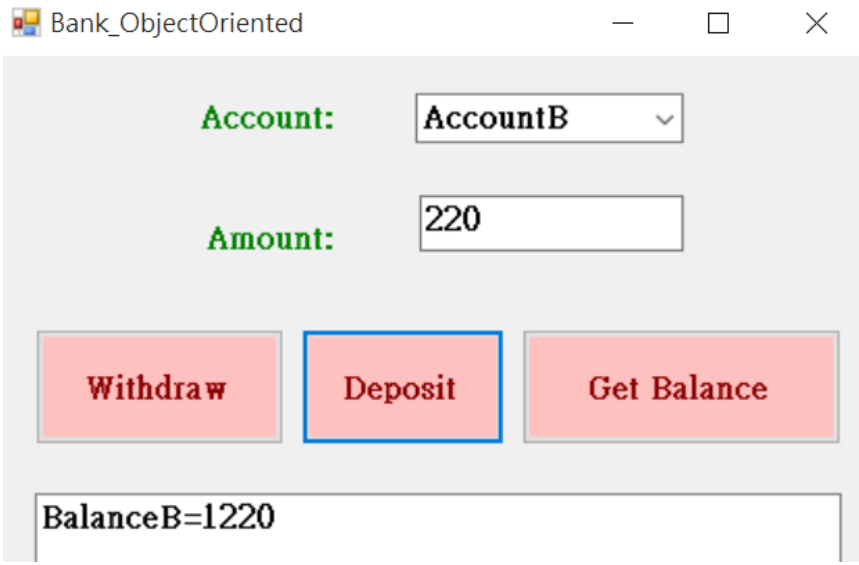
```
class Account
{
    public int Balance;
    public Account(int amount)
    {
        Balance = amount;
    }
    //
    public void deposit(int amount)
    {
        Balance += amount;
    }
    //
    public void withdraw(int amount)
    {
        Balance -= amount;
    }
    //
    public int getBalance(int account)
    {
        return Balance;
    }
}
```

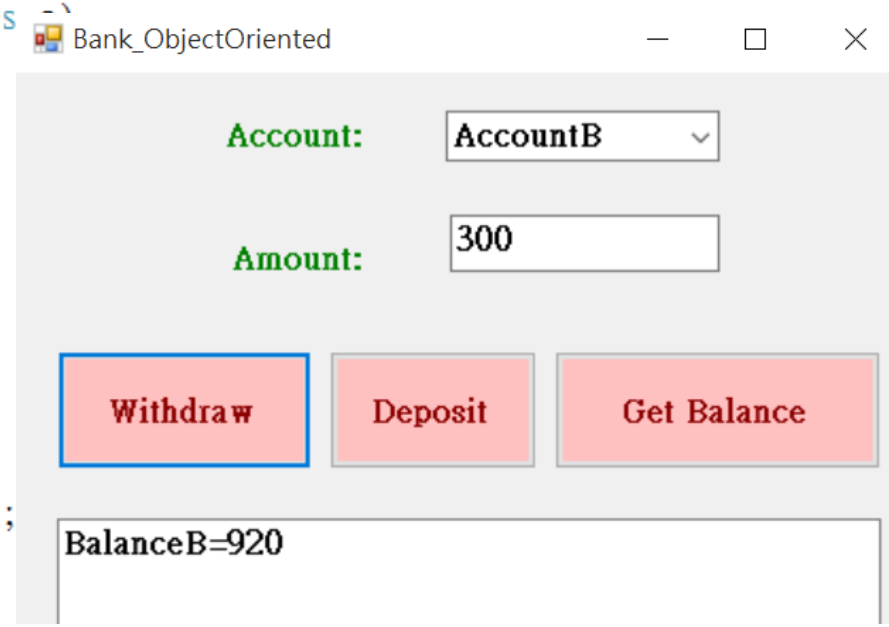
```
Account accountA, accountB;
private void Form1_Load(object sender, EventArgs e)
{
    accountA = new Account(1000);
    accountB = new Account(1000);
}
private void btnGetBalance_Click(object sender, EventArgs e)
{
    if (cbAccount.SelectedIndex == 0)
    {
        tbResult.Text = "BalanceA=" + accountA.Balance;
    }
    else
    {
        tbResult.Text = "BalanceB=" + accountB.Balance;
    }
}
```



Object-Oriented Example Code in C#


```
private void btnDeposit_Click(object sender, EventArgs e)
{
    if (cbAccount.SelectedIndex == 0)
    {
        accountA.deposit(int.Parse(tbAmount.Text));
        tbResult.Text = "BalanceA=" + accountA.Balance;
    }
    else
    {
        accountB.deposit(int.Parse(tbAmount.Text));
        tbResult.Text = "BalanceB=" + accountB.Balance;
    }
}
private void btnWithdraw_Click(object sender, EventArgs e)
{
    if (cbAccount.SelectedIndex == 0)
    {
        accountA.withdraw(int.Parse(tbAmount.Text));
        tbResult.Text = "BalanceA=" +accountA.Balance;
    }
    else
    {
        accountB.withdraw(int.Parse(tbAmount.Text));
        tbResult.Text = "BalanceB=" + accountB.Balance;
    }
}
```





Three Pillars of Object-Oriented Programming

- **Encapsulation:**
  - **Classes and Objects**
- **Inheritance:**
  - **Generalization and Specialization**
- **Polymorphism:**
  - **Dynamic Binding (i.e., Binding at runtime)**

Encapsulation

- The concept of encapsulation revolves around the notion that an object's internal data should not be directly accessible from an object instance.

- Rather, if the caller wants to alter the state of an object, the user does so indirectly using accessor (i.e., "getter") and mutator (i.e., "setter") methods.

Class

- **The purpose of a class is to declare a collection of methods, operations and attributes that fully describe the structure and behaviour of objects.**

➢ **Structure:**

what an object *knows* information that it holds

➢ **Behaviour: what an object *can do***

Class

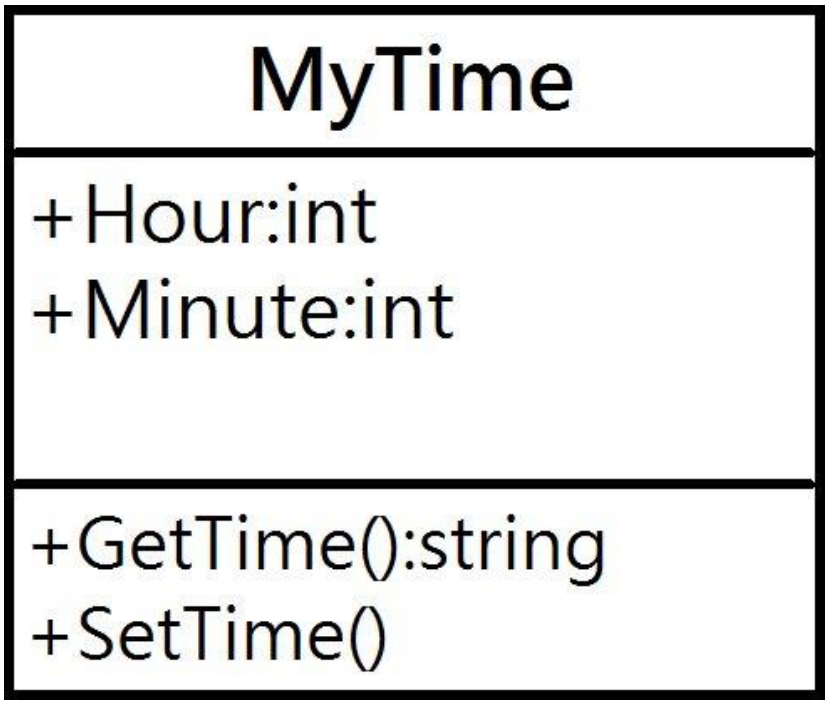
- Each class contains two types of members:
  - Data Member
  - Method Member

- Syntax for Class Declaration:

```
Access_modifier class class_name
{
  fields、properties、methods
}
```

Example of Declaring Class


```
public class MyTime
{
   public int Hour;
   public int Minute;
   public string GetTime()
   {
      string str;
      str = Hour + ":" + Minute;
      return str;
   }
   public void SetTime(int h, int m)
   {
      Hour = h;
      Minute = m;
   }
}
```



Objects

- An object is:

  "an abstraction of something in a problem domain, reflecting the capabilities of the system to

  - keep information about it,
  - interact with it,
  - or both."

Objects

- Objects have state, behaviour and identity
  - **State:**

    the condition of an object at any moment, affecting how it can behave
  - **Behaviour:**

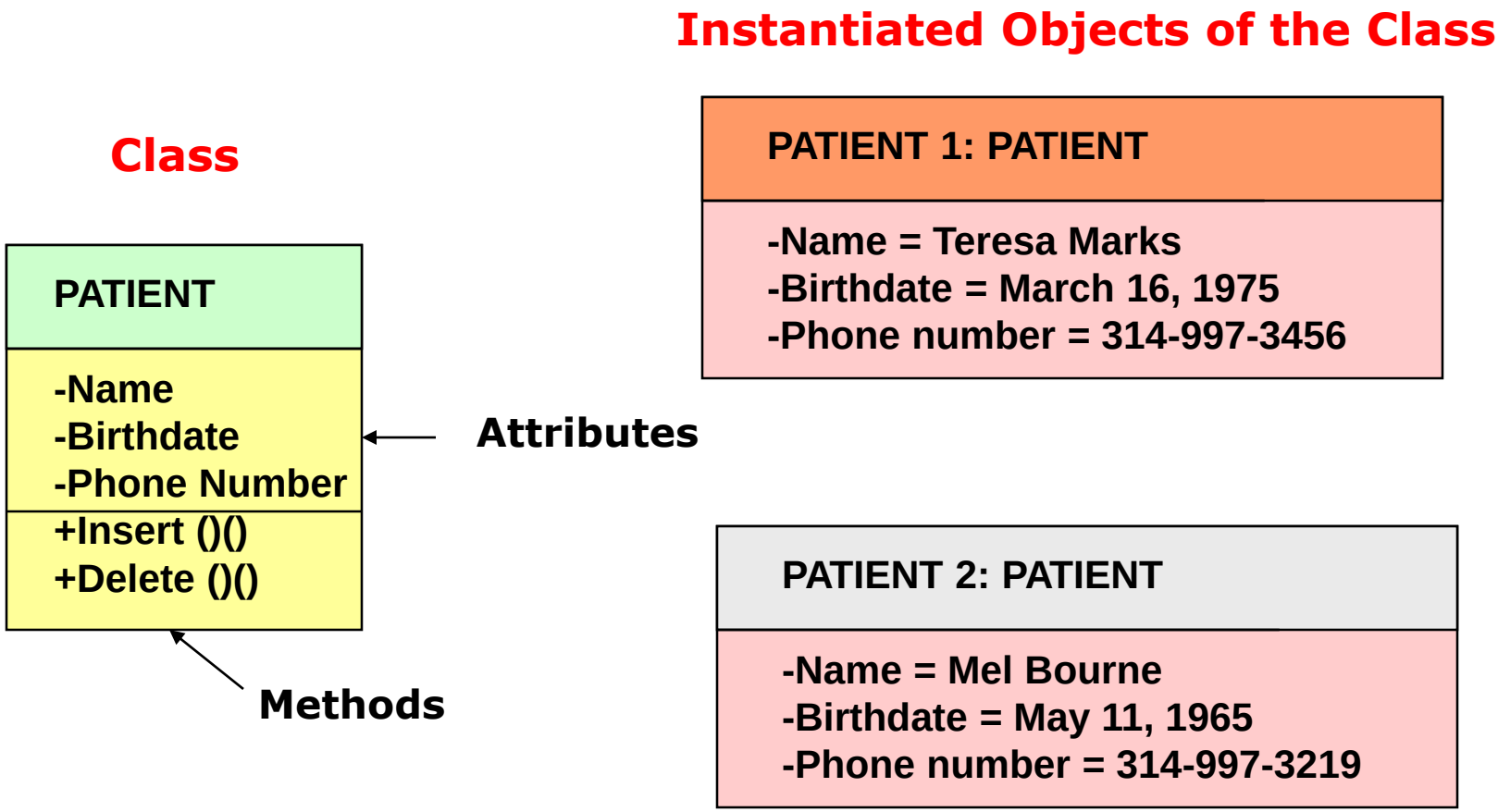
    what an object can do, how it can respond to events and stimulation
  - **Identity:**

    each object is unique

Object V.S. Class

- All objects are instances of some class
- Class is a description of a set of objects with similar
  - attributes,
  - operations,
  - methods,
  - relationships and semantics.

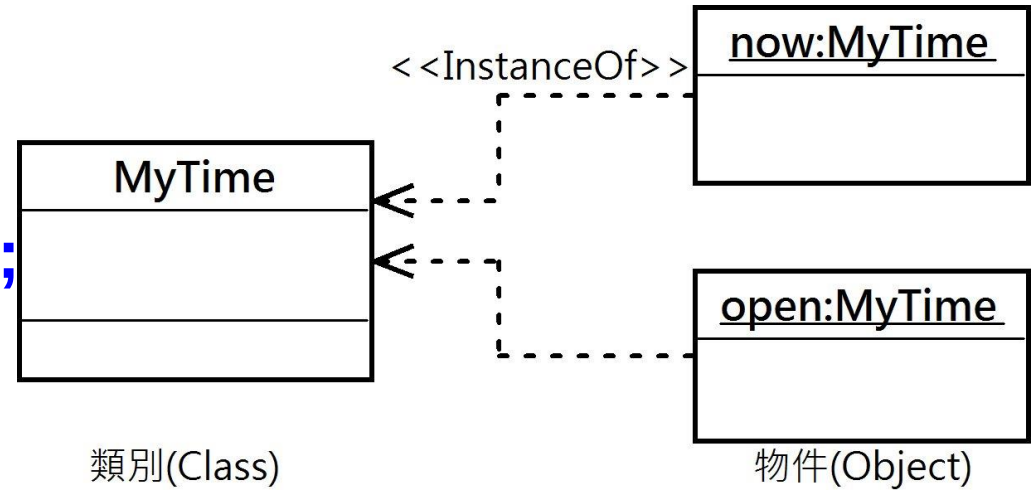
A Class and Its Objects



Example of Creating Object using Class

- **Example of creating objects using class:**

```
MyTime now, open;
now = new MyTime();
open = new MyTime();
```



- **Set objects' properties:**

```
open.Hour = 10;
open.Minute = 30;
```

- **Call an object's method:**

```
lblOutput.Text = "Open time is: " + open.GetTime();
```

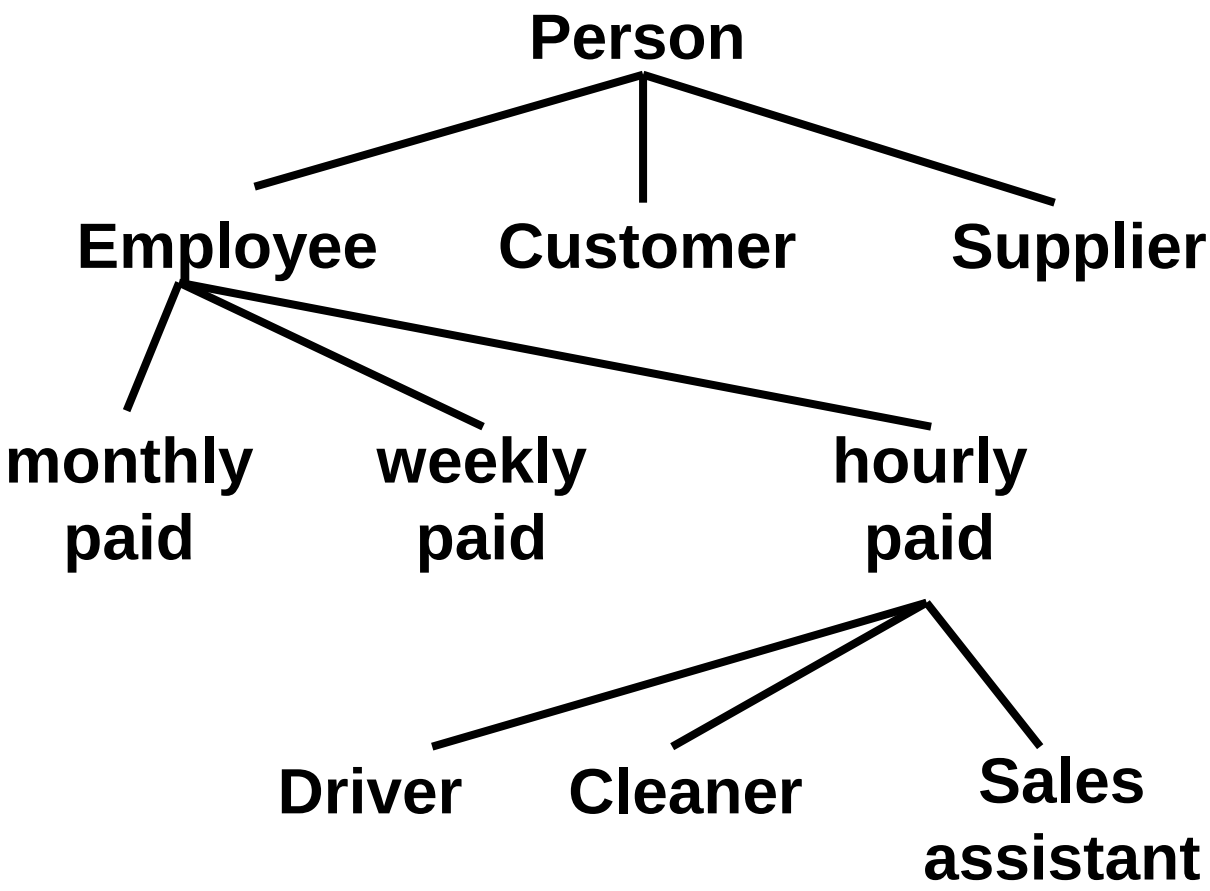
Inheritance (Class Hierarchy)

- The basic idea behind classical inheritance is that new classes can be created using existing classes as a starting point.
- Inheritance is the aspect of OOP that facilitates code reuse.

Generalization and Specialization

- Classification is hierarchic in nature
- For example, a person may be an employee, a customer, a supplier of a service
- An employee may be paid monthly, weekly or hourly
- An hourly paid employee may be a driver, a cleaner, a sales assistant

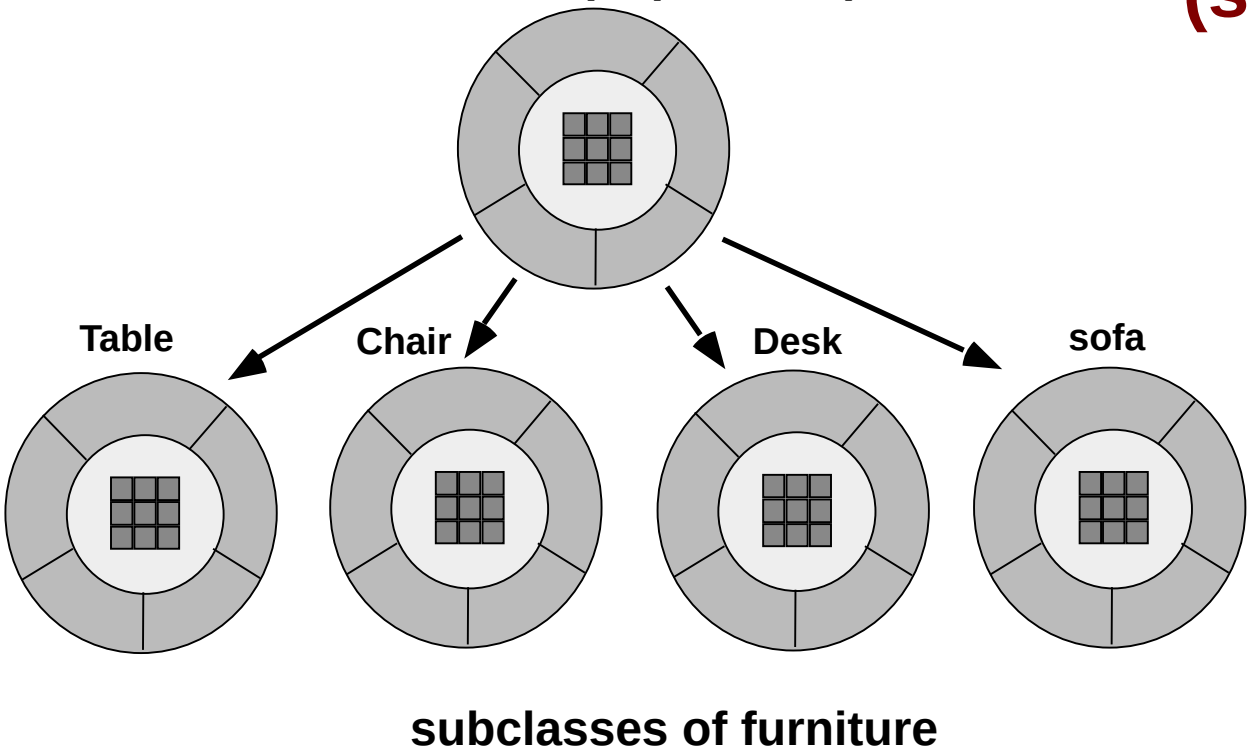
Class Hierarchy Example (1/2)



More general (superclasses)

More specialized (subclasses)

Class Hierarchy Example (2/2)



More general (superclasses)

More specialized (subclasses)

Generalization and Specialization

- **More general bits of description are *abstracted out* from specialized classes:**

General (superclass)

Person
Name Date of birth Gender Title

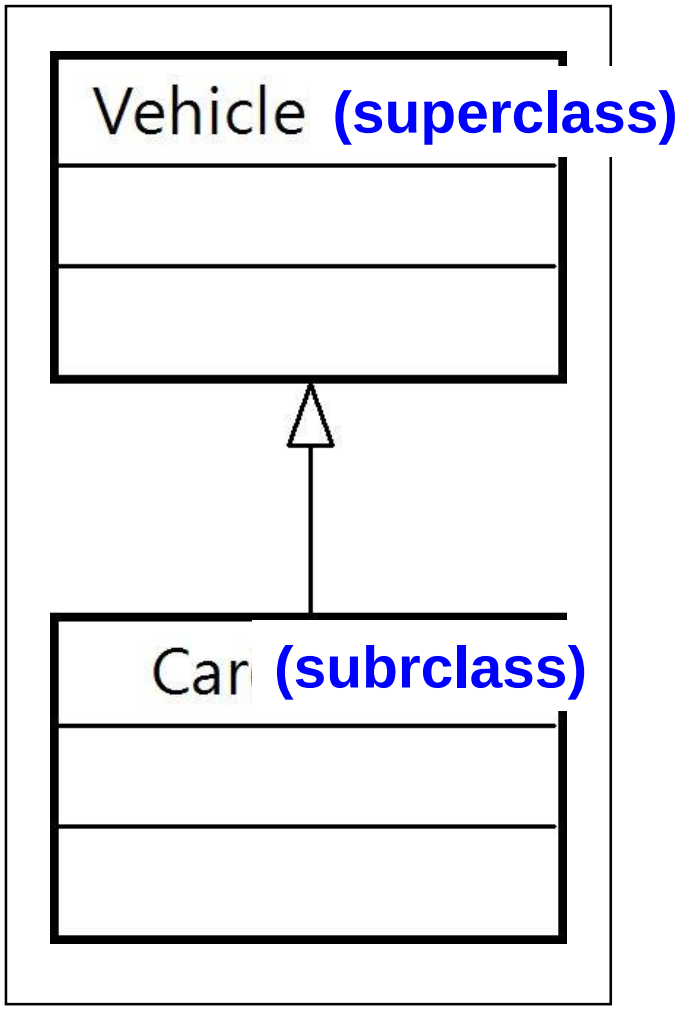
Specialized (subclass)

HourlyPaidDriver
StartDate StandardRate OvertimeRate LicenceType

Class's Inheritance

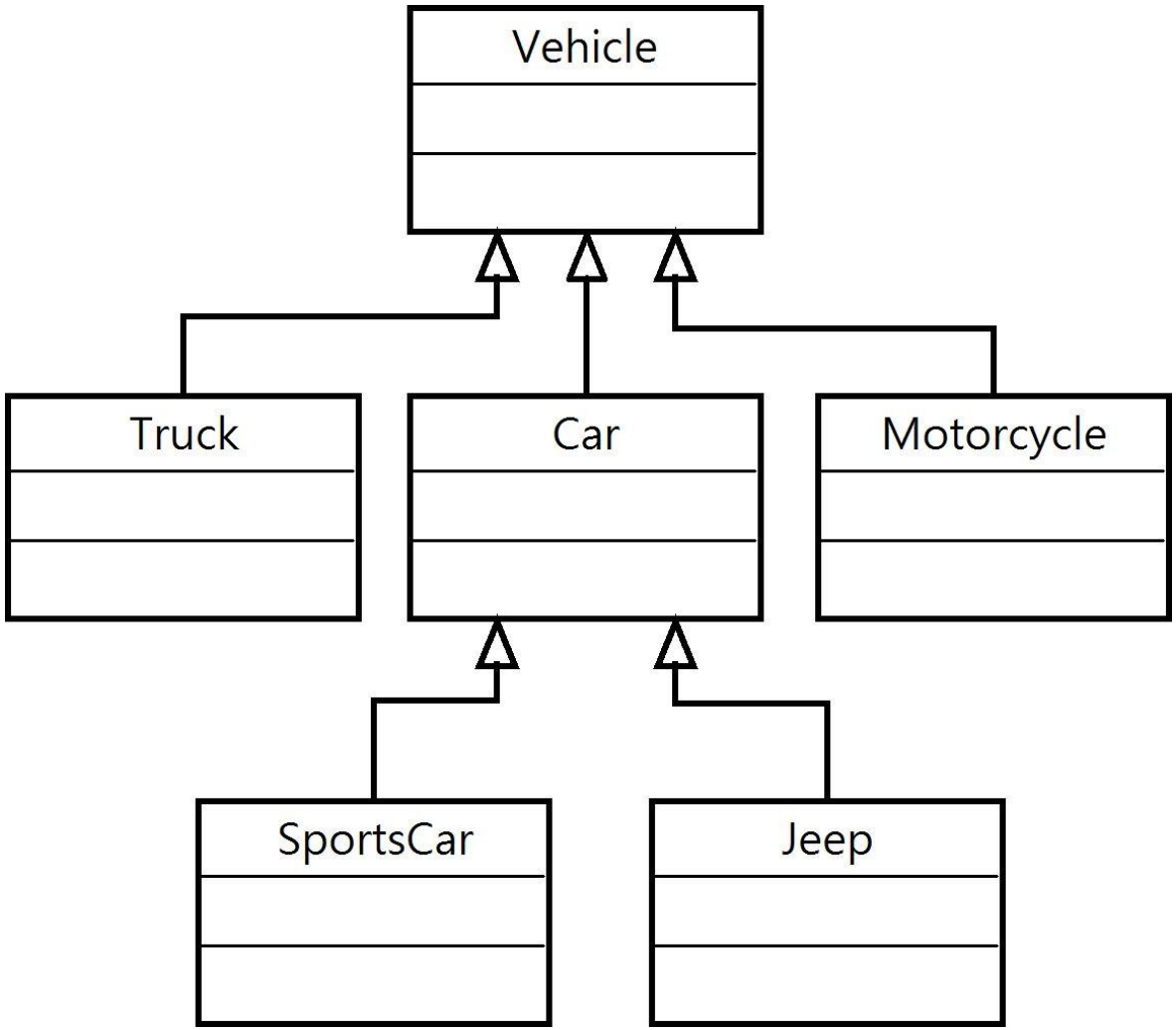
- **Vehicle:**
  Superclass or Base Class

- **Car:**
  Subclass or Derived Class



Sibling Classes

- Truck, Car, Motorcycle are sibling classes：



Inheritance

- The whole description of a superclass applies to all its subclasses, including:
  - Information structure
  - Behaviour
- Often known loosely as *inheritance*
- Actually, inheritance is the facility in an O-O language that implements generalization /specialization

Syntax of Inheritance in C#

- **Syntax of Inheritance :**

```
Access_modifier  class class_name: base_class
{
   // additional properties
   // additional methods
}
```

Example of Inheritance Coding (1/2)

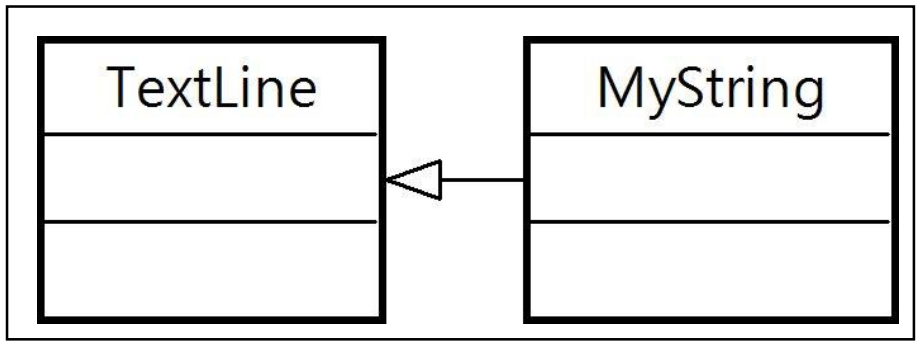
- TextLine is the base class:

```
public class TextLine {
   public string Line;
   public TextLine(string text) { Line = text; }
   public string GetWord() {  }
   public virtual string GetText() {  }
}
```

Example of Inheritance Coding (2/2)

- MyString is a subclass of the TextLine class:

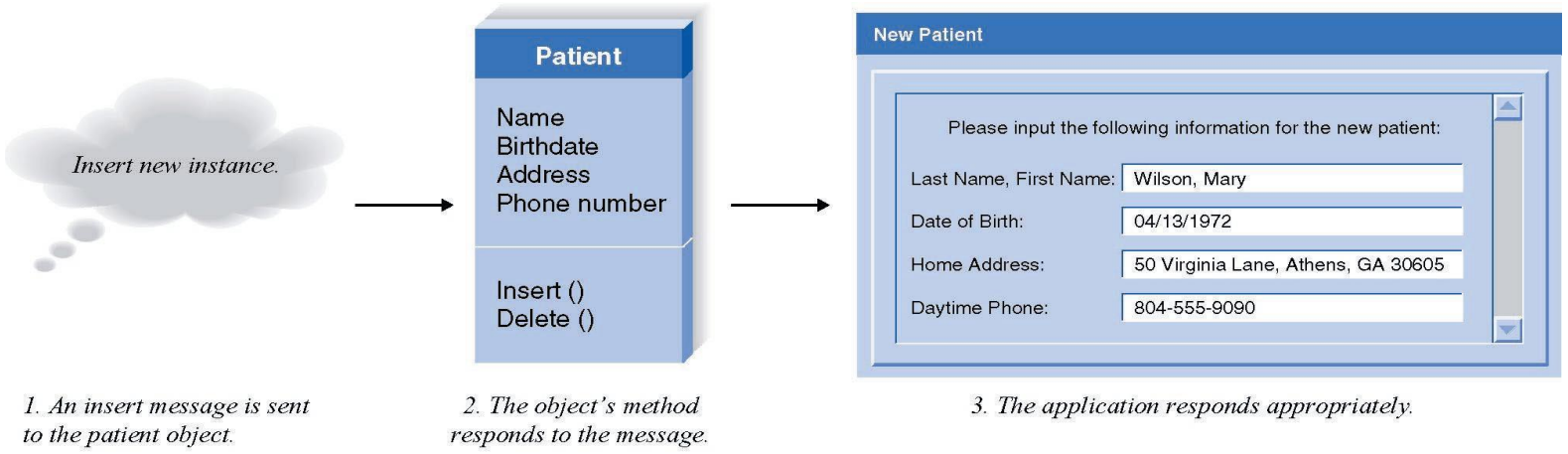
```
public class MyString : TextLine
{
   public MyString() : base() { }
   public long Length {  }
   public int InStr(string str) {  }
}
```



Polymorphism

- Polymorphism allows one message to be sent to objects of different classes
- Sending object need not know what kind of object will receive the message
- Each receiving object knows how to respond appropriately
- Polymorphism can achieve **dynamic binding**, determining which methods should be called at runtime, instead of compiling time.

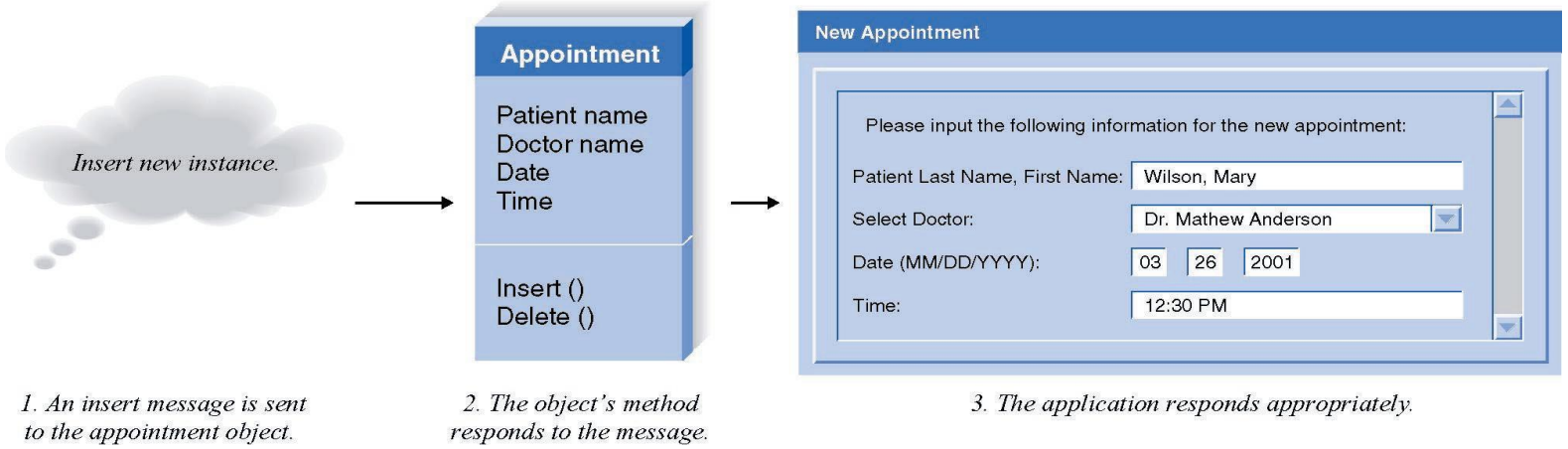
Polymorphism



1. An insert message is sent to the patient object.

2. The object's method responds to the message.

3. The application responds appropriately.



1. An insert message is sent to the appointment object.

2. The object's method responds to the message.

3. The application responds appropriately.

Example of Polymorphism Coding


```
abstract class Shape {
        public abstract void f();
}

class Triangle extends Shape {
    public void f() {  System.out.println("Triangle!");   }
}

class Rectangle extends Shape {
    public void f() {  System.out.println("Rectangle!");    }
}

class Circle extends Shape {
    public void f() {  System.out.println("Circle!");   }
}

class poly {
    public static void main (String[] args) {
            Shape[] s = new Shape[] { new Triangle(), new Rectangle(), new Circle() };

            for (int i=0; i<s.length; i++)
                    s[i].f();
    }   }
```

Results:

Triangle!
Rectangle!
Circle!

Implementation of Polymorphism by Abstract Class

- Declare an abstract base class：

```
abstract class Shape {
   public abstract double Area();
}
```

Declare subclasses

- Declare subclasses to inherit the base class:

```
class Circle : Shape {
   ……..
   public override double Area() {
      return (3.1415 * r * r);
   }
}
class Rectangle : Shape {
   ……..
   public override double Area() {
      return (height * width);
   }
}
```

Polymorphism of Method's Behaviors

- **Create objects of subclasses:**

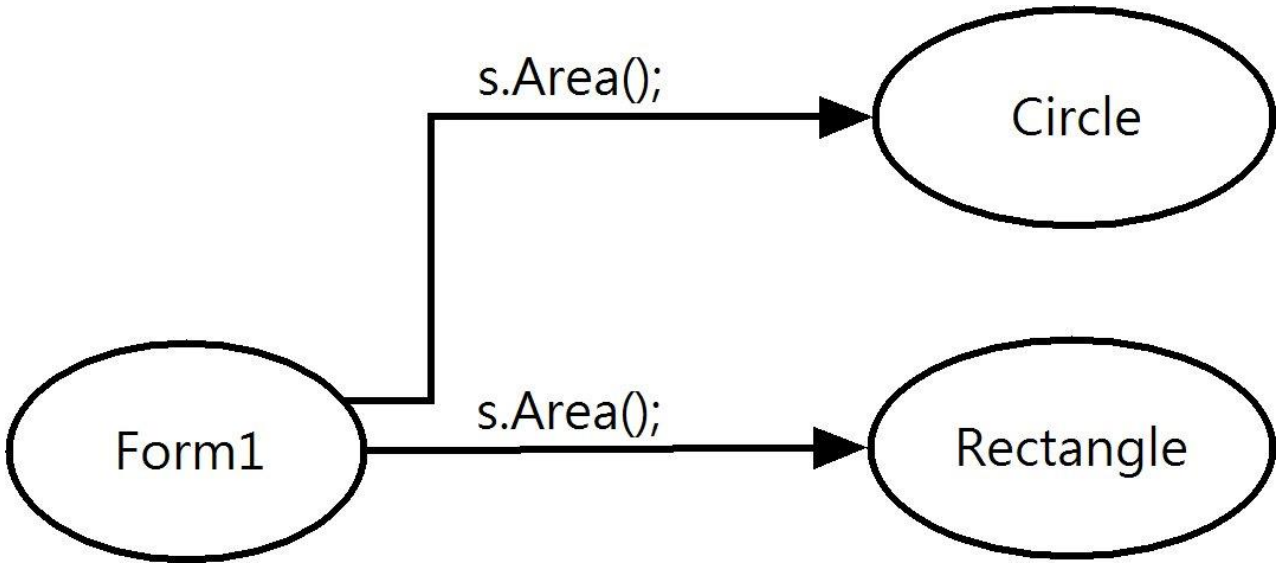
```
Circle c = new Circle();

Rectangle r = new Rectangle();
```

- **Declare base class's variable to reference the objects created by subclasses:**

```
Shape s;

s = c;

lblOutput.Text += "Area: " + s.Area() + "\n";

s = r;

lblOutput.Text += "Area: " + s.Area() + "\n";
```

Illustration of Dynamic Binding

- **Passing messages to different objects at runtime:**



Object-Oriented Programming Language

- Java :
- C++ :
- C# :
- JavaScript
- Node.js
- Python

II. System Analysis and Design with UML (Unified Modeling Language)

What is a Model

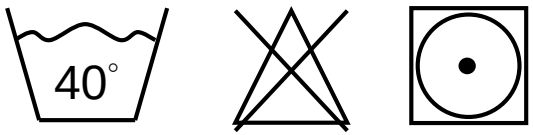
- Like a map, a model represents something else
- A useful model has the right level of detail and represents only what is important for the task in hand
- Many things can be modelled: bridges, traffic flow, buildings, economic policy

Why Use a Model?

- A model is quicker and easier to build
- A model can be used in a simulation
- A model can evolve as we learn
- We can choose which details to include in a model
- A model can represent real or imaginary things from any domain

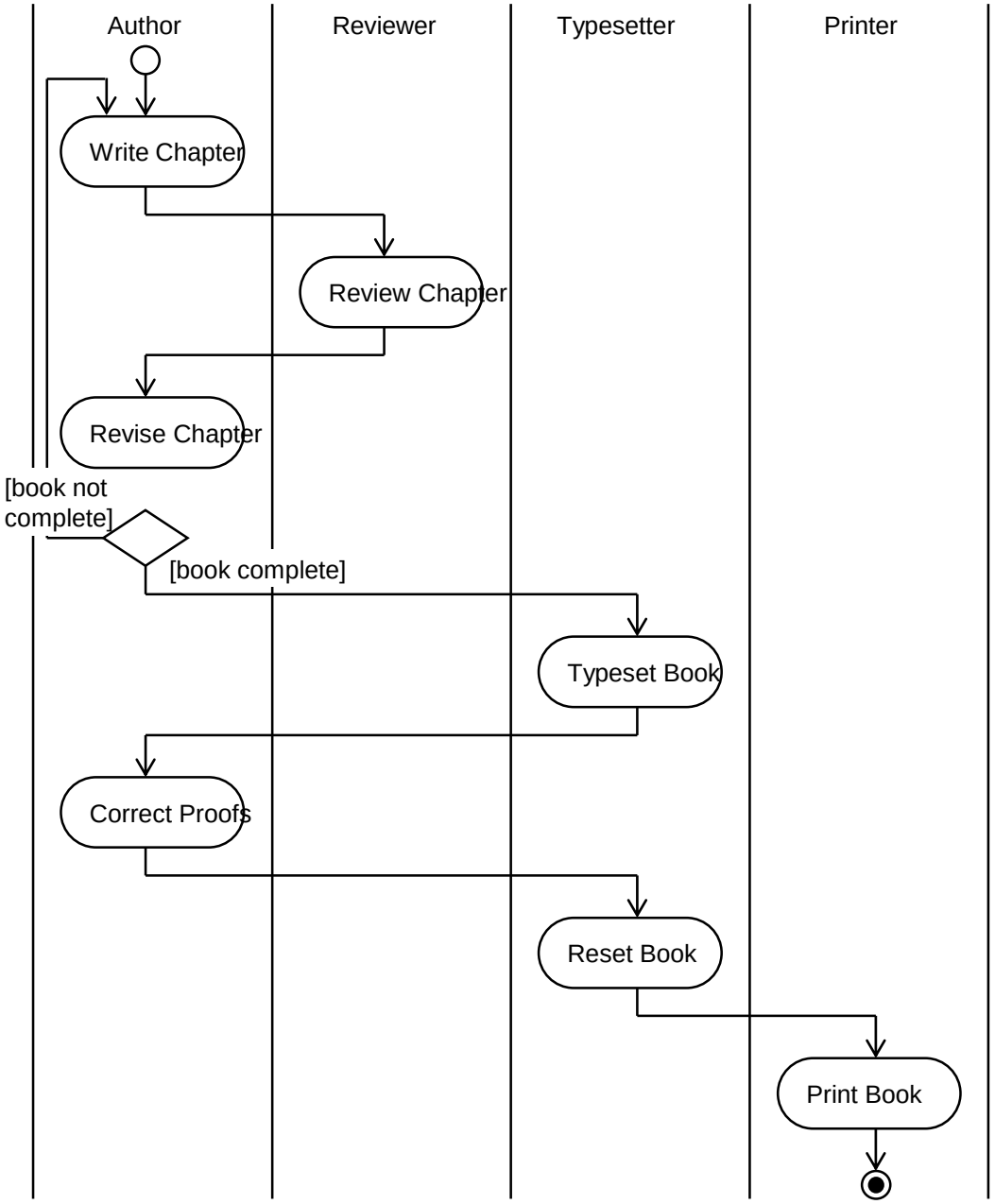
What is a Diagram?

- Abstract shapes are used to represent things or actions from the real world
- Diagrams follow rules or standards
- The standards make sure that different people will interpret the diagram in the same way

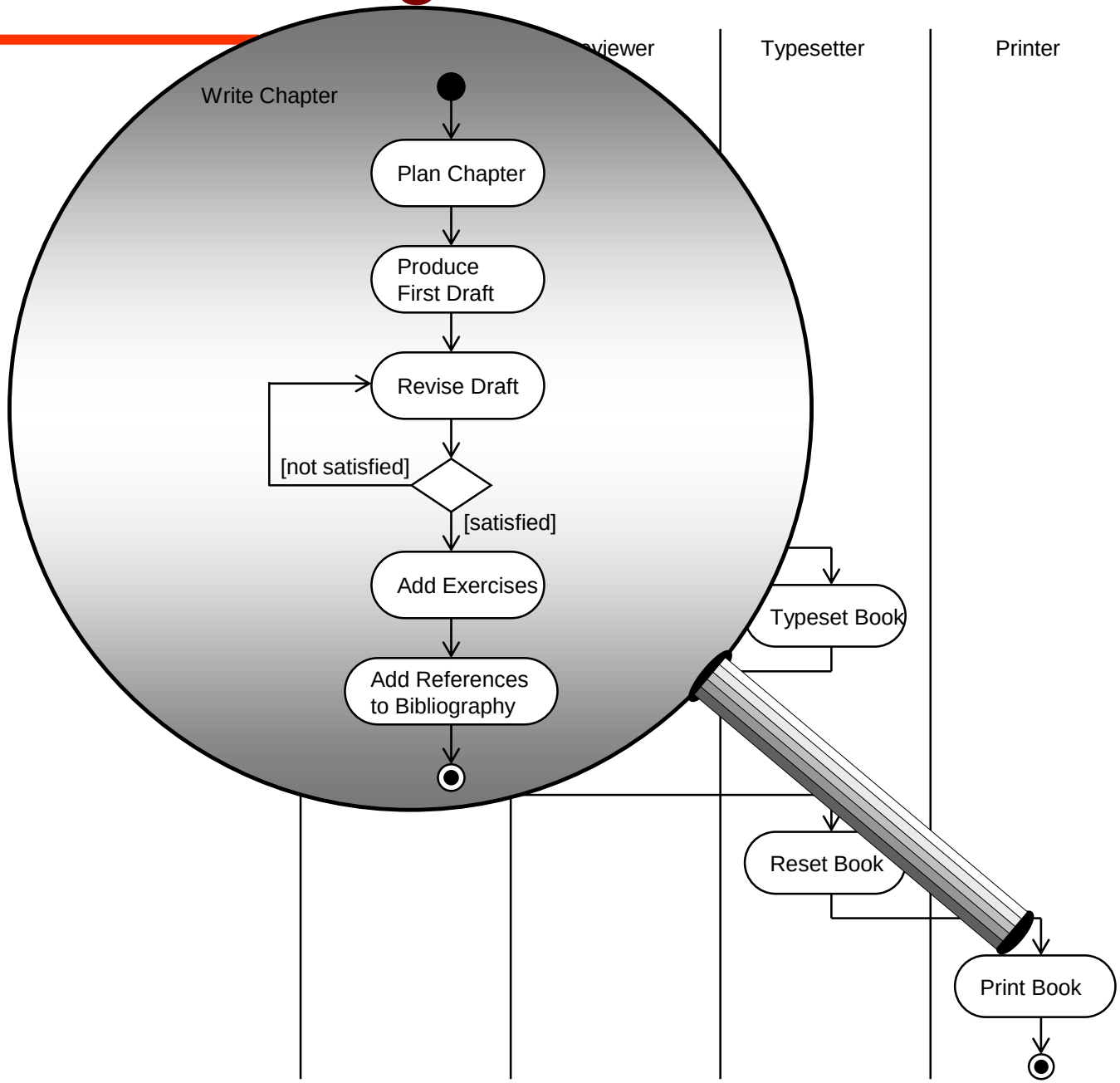


An Example of a Diagram

- An activity diagram of the tasks involved in producing a book.



Hiding Detail

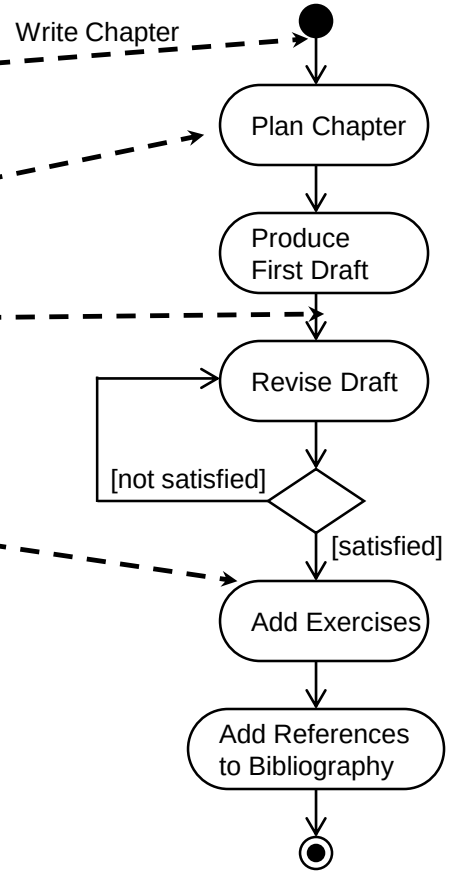


Object-Oriented Analysis and Design

- Uses Unified Modeling Language (UML) for diagramming
  - Use-case Driven
  - Architecture Centric
  - Iterative and Incremental

Diagrams in UML

- UML diagrams consist of:
  - icons
  - two-dimensional symbols
  - paths
  - Strings
- UML diagrams are defined in the UML specification.



Diagrams V.S. Models

- A diagram illustrates some aspect of a system.
- A model provides a complete view of a system at a particular stage and from a particular perspective.
- A model may consist of a single diagram, but most consist of many related diagrams and supporting data and documentation.

Examples of Models

- Requirements Model
  - complete view of requirements
  - may include other models, such as a Use Case Model
  - includes textual description as well as sets of diagrams

Examples of Models

- **Behavioural Model**
  - shows how the system responds to events in the outside world and the passage of time
  - an initial model may just use Communication Diagrams
  - a later model will include Sequence Diagrams and Statecharts

Models in UML

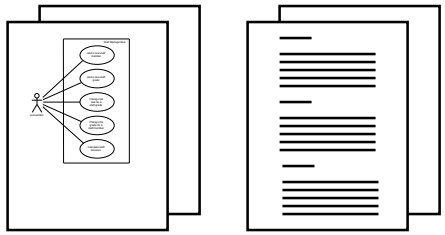
- Different models present different views of the system, for example (4+1 Views):
  - Use Case View
  - Design View
  - Process View
  - Implementation View
  - Deployment View

Developing Models

- During the life of a project using an iterative life cycle, models change along the dimensions of:
  - abstraction—they become more concrete
  - formality—they become more formally specified
  - level of detail—additional detail is added as understanding improves

Development of the Use Case Model

Iteration 1
Obvious use cases.
Simple use case descriptions.



Iteration 2
Additional use cases.
Simple use case descriptions.
Prototypes.

Iteration 3
Structured use cases.
Structured use case descriptions.
Prototypes.

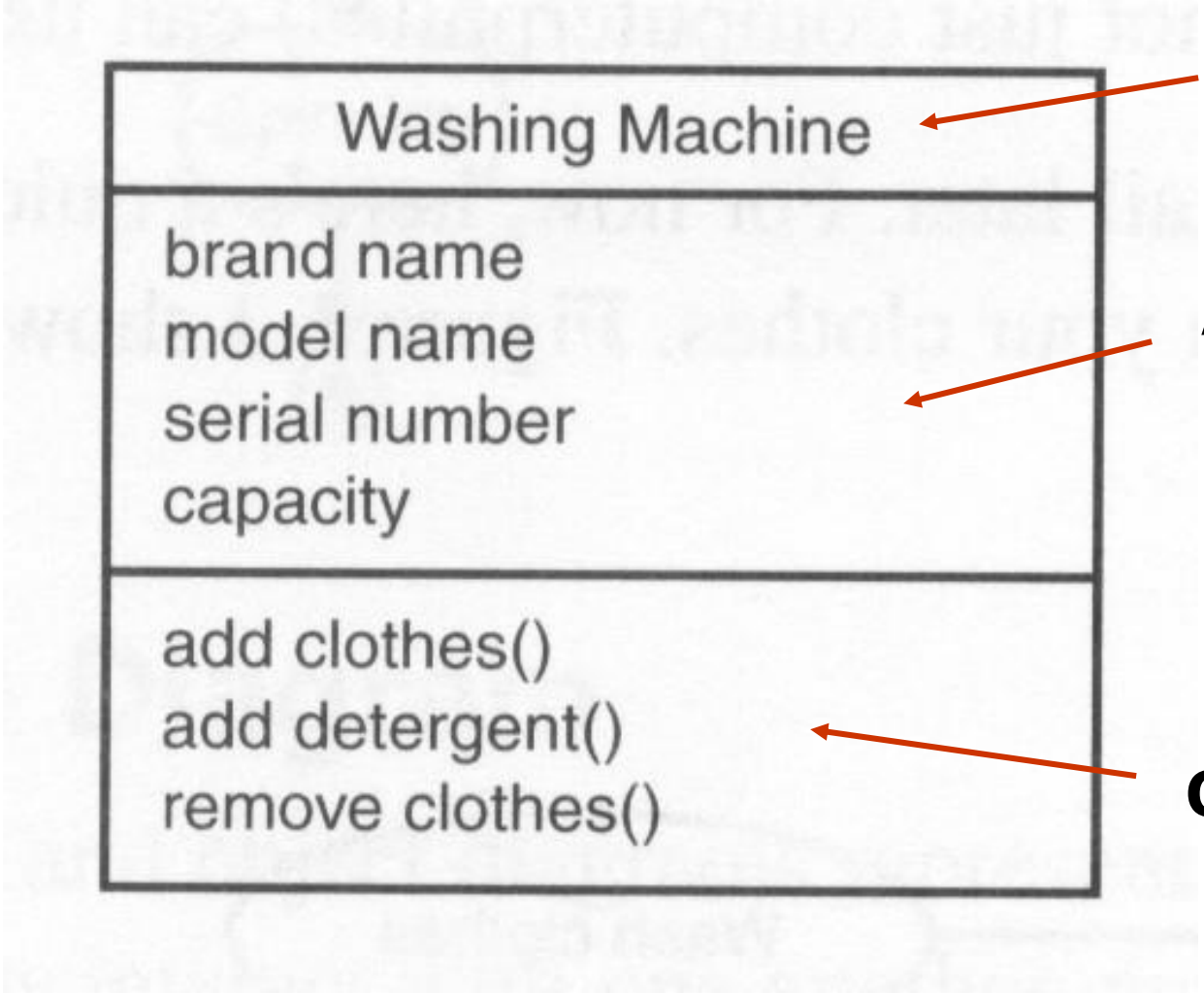
Introducing the Unified Modeling Language (UML)

- **Why the UML is necessary?**

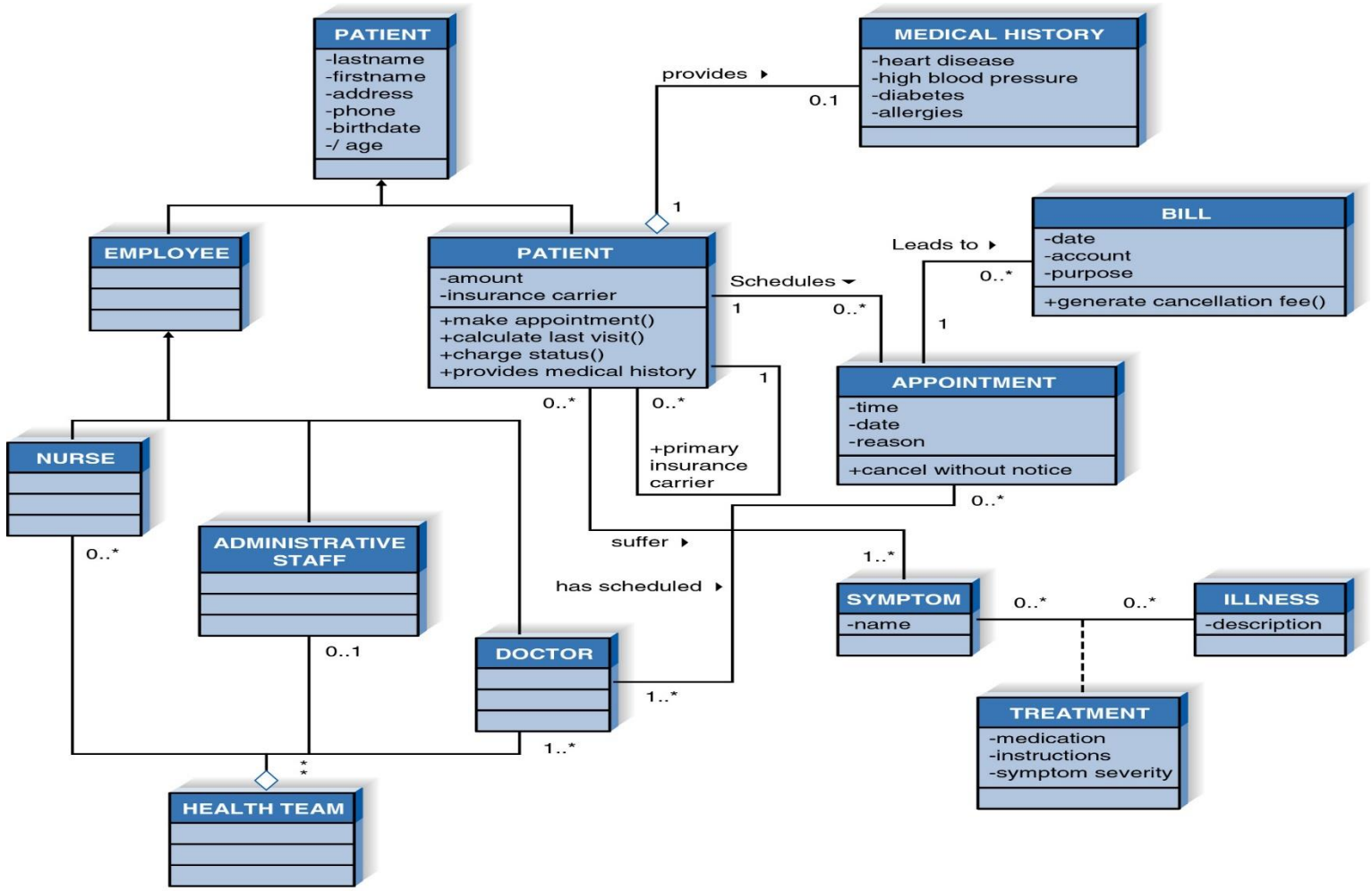
- **How the UML came to be?**

- **The diagrams of the UML**

The UML Class Icon

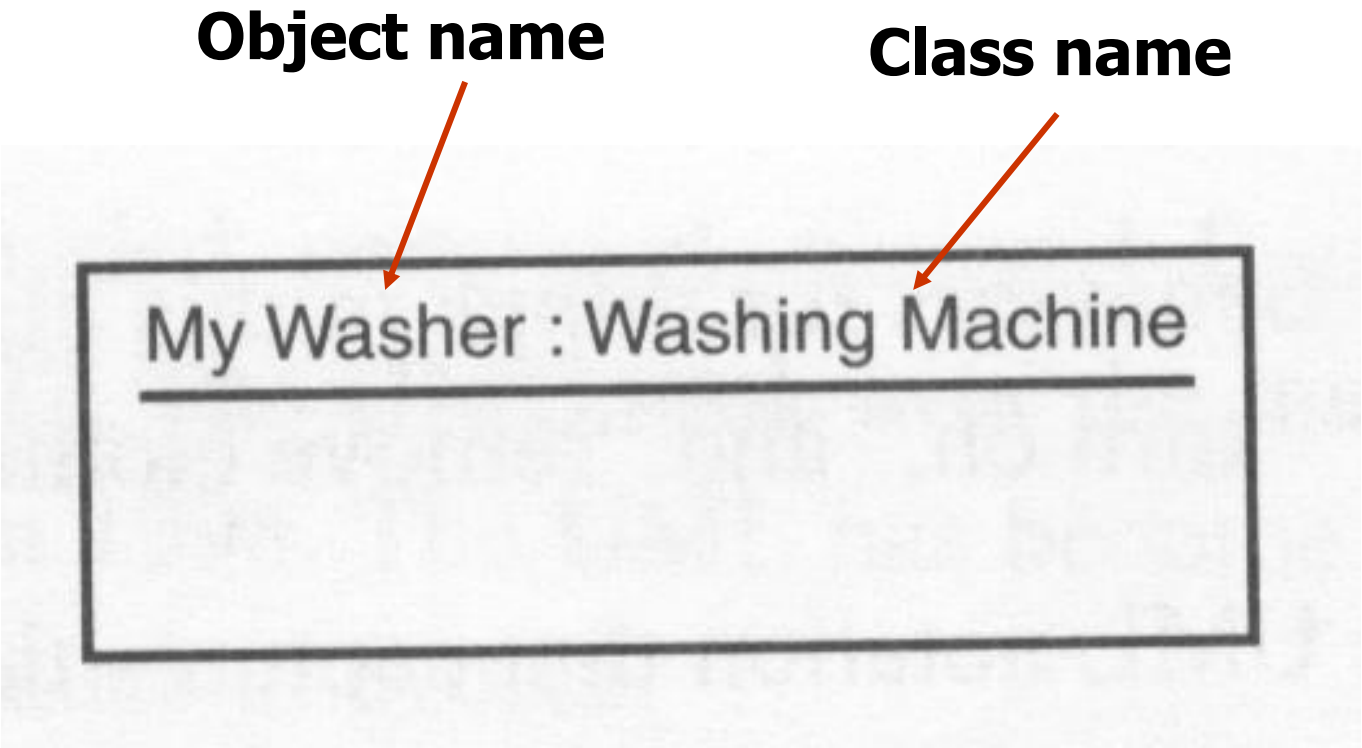


Example Class Diagram



Dennis: SAD
Fig: 7-2 W-26 100% of size
Fine Line Illustrations (516) 501-0400

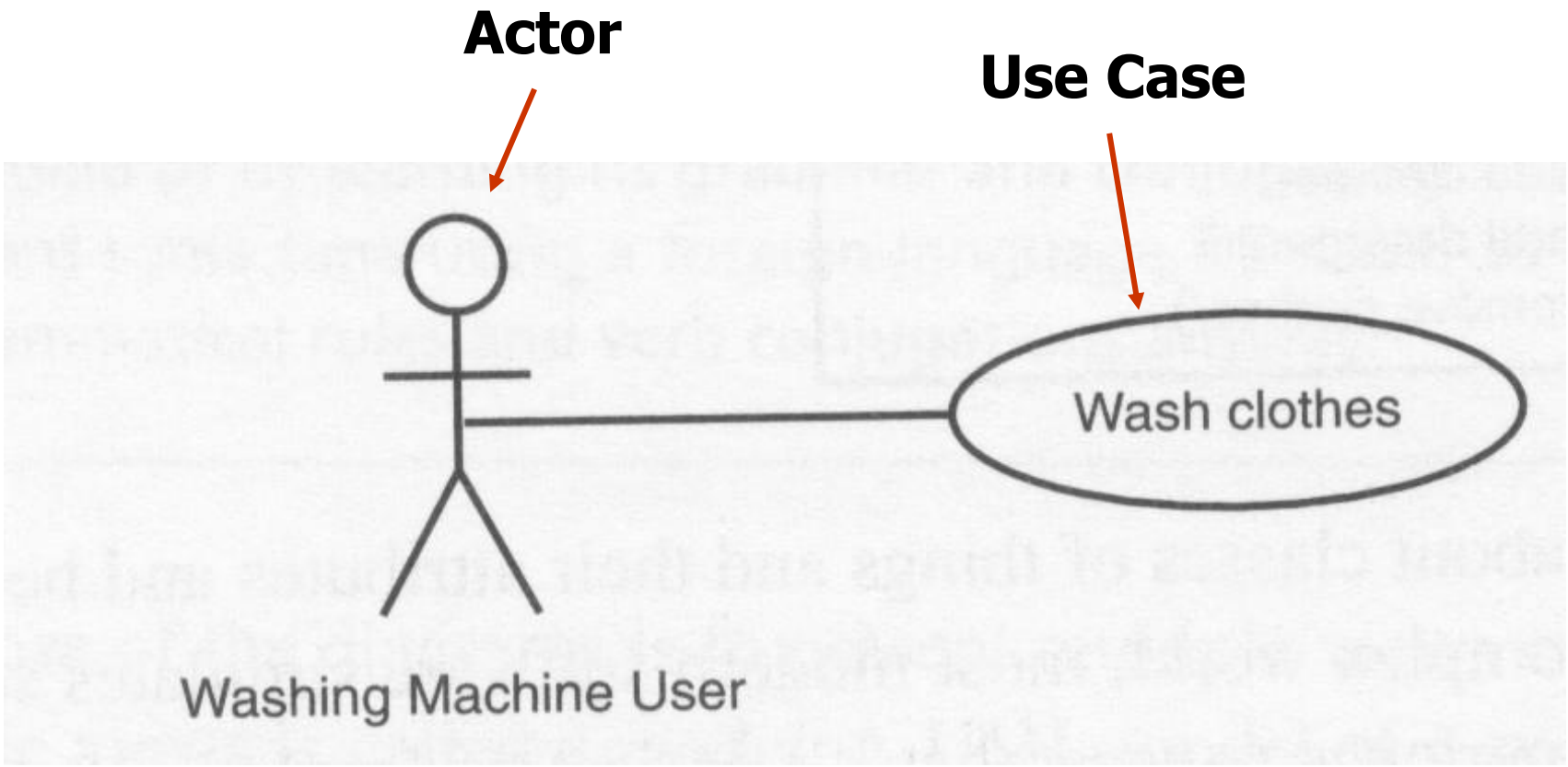
The UML Object Icon



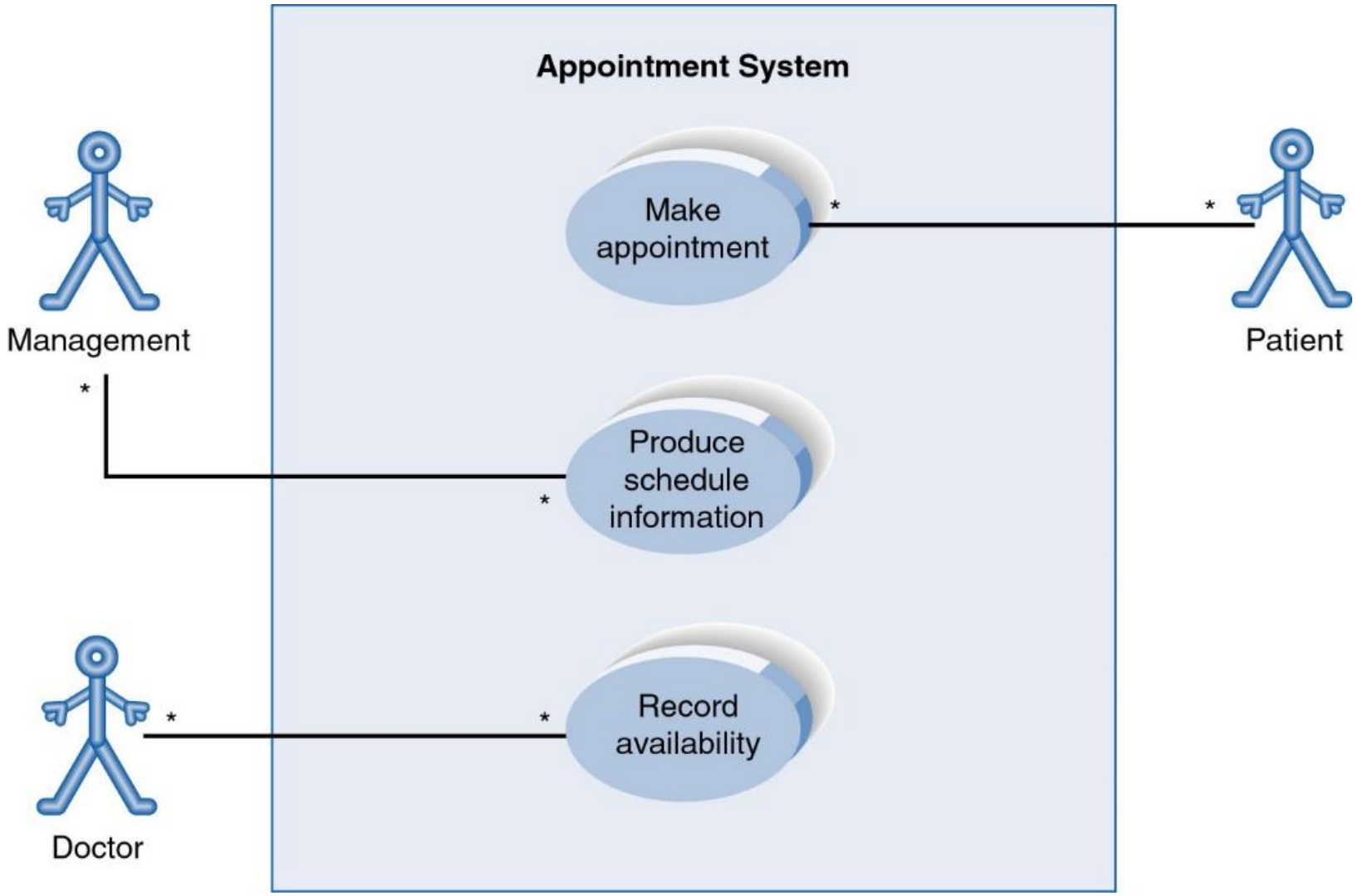
- **An object diagram shows the objects and their relationships with one another.**

The UML Use Case Diagram

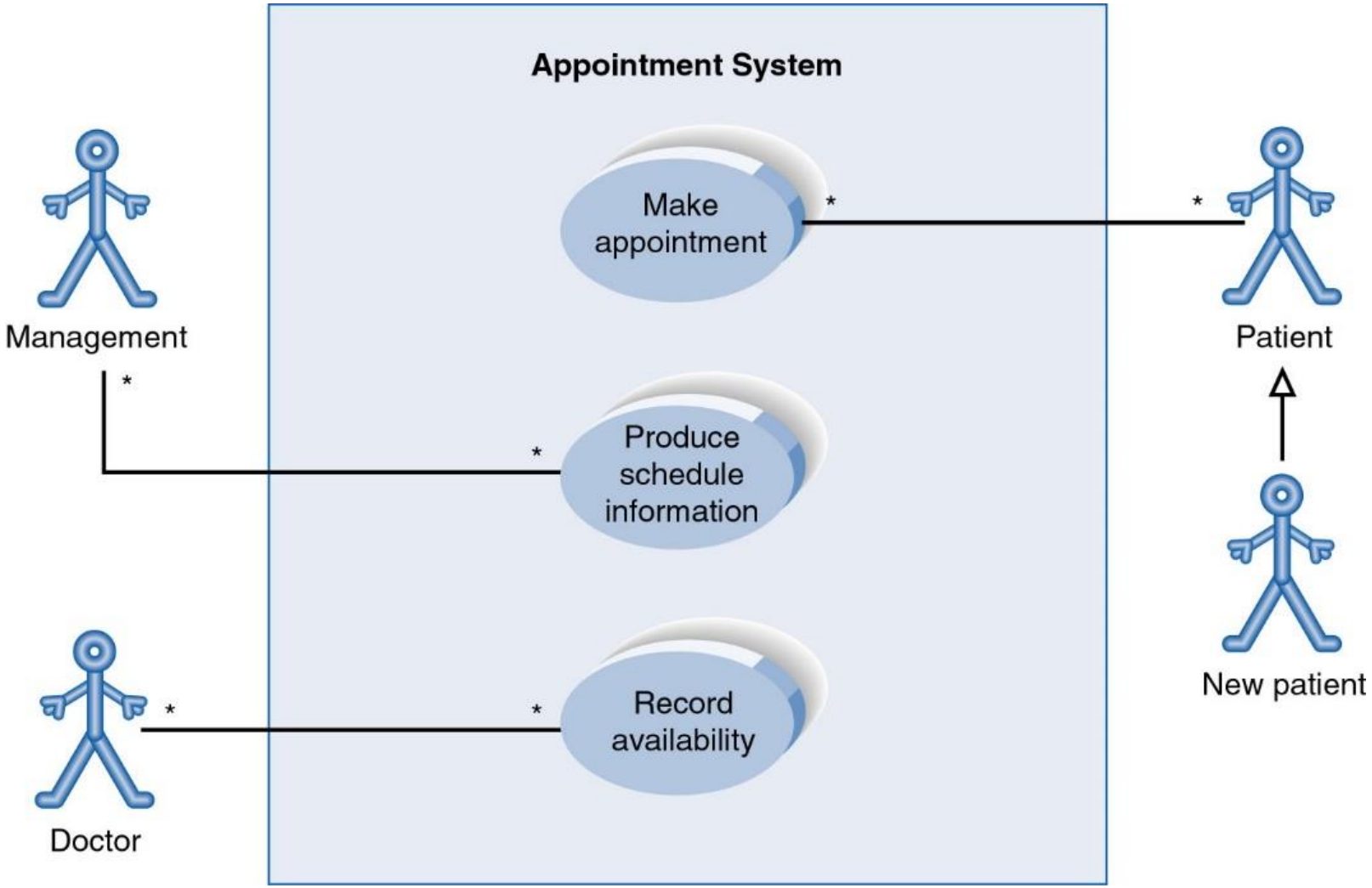
- **A use case is a collection of scenarios about system use .**



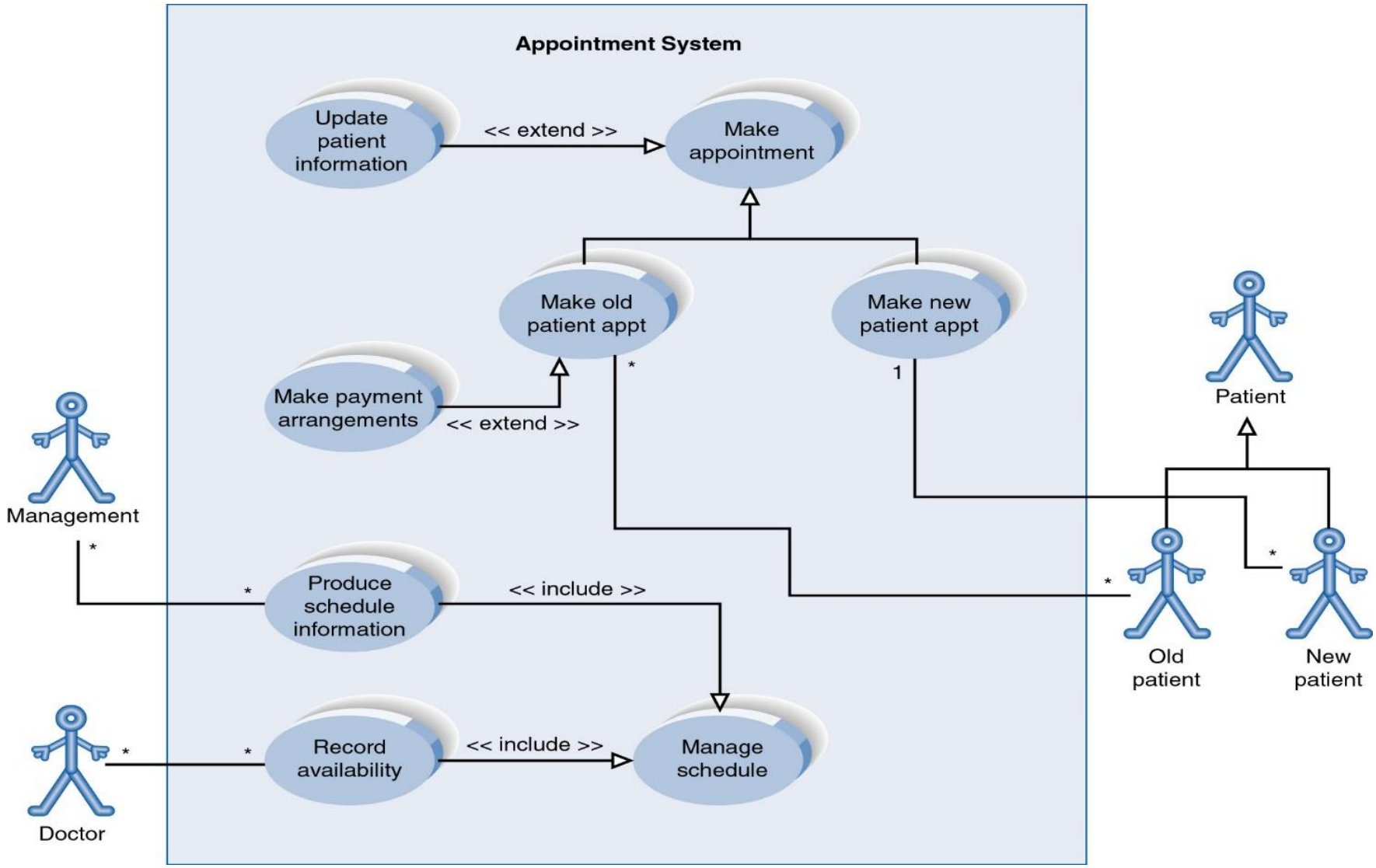
The Use-Case Diagram for Appointment System



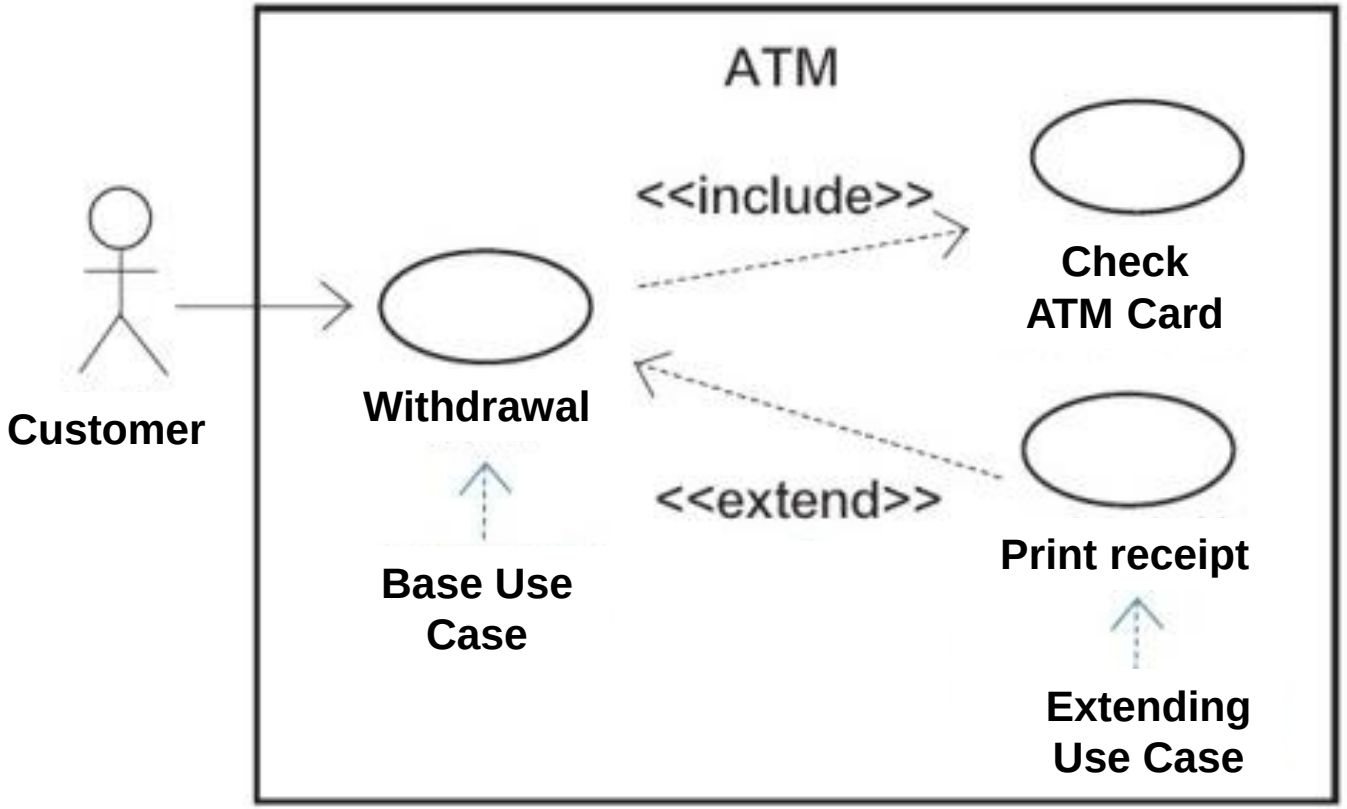
Use-Case Diagram with Specialized Actor



Extend and Include Relationships



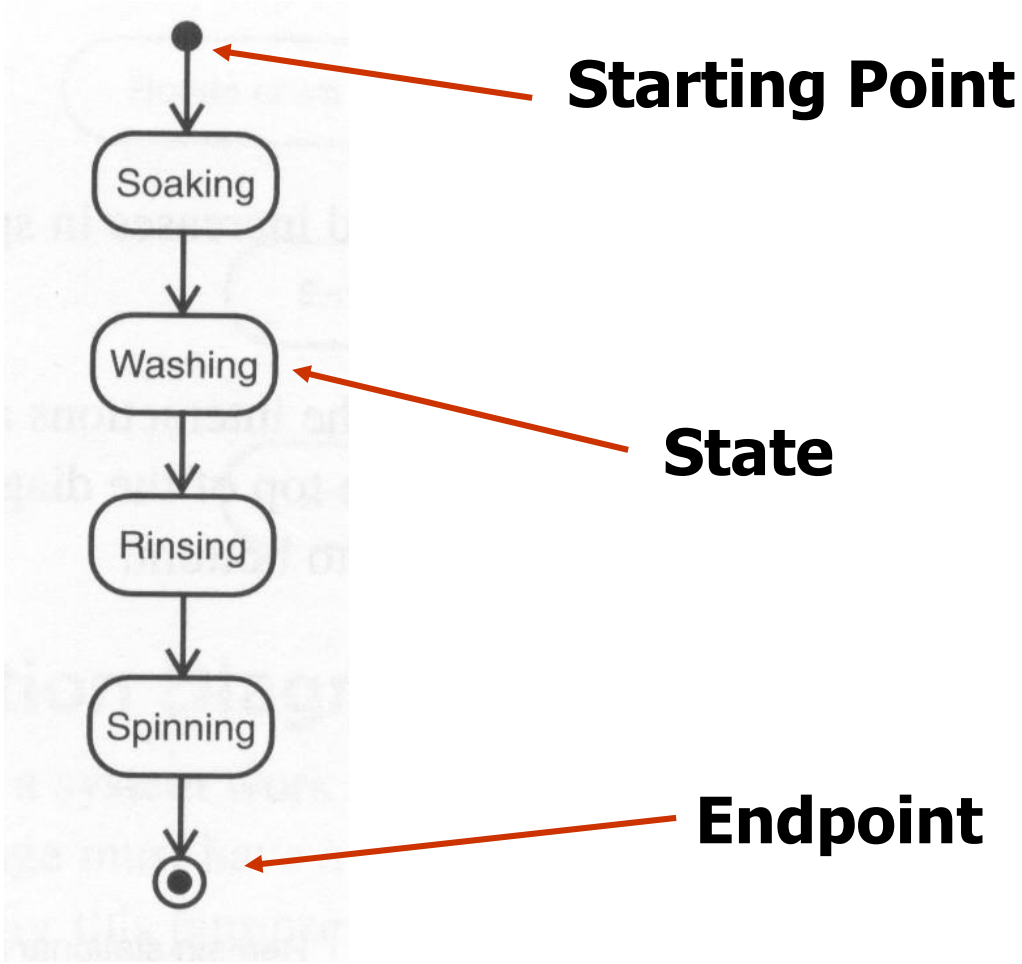
Extend and Include Relationships



Source: iThome

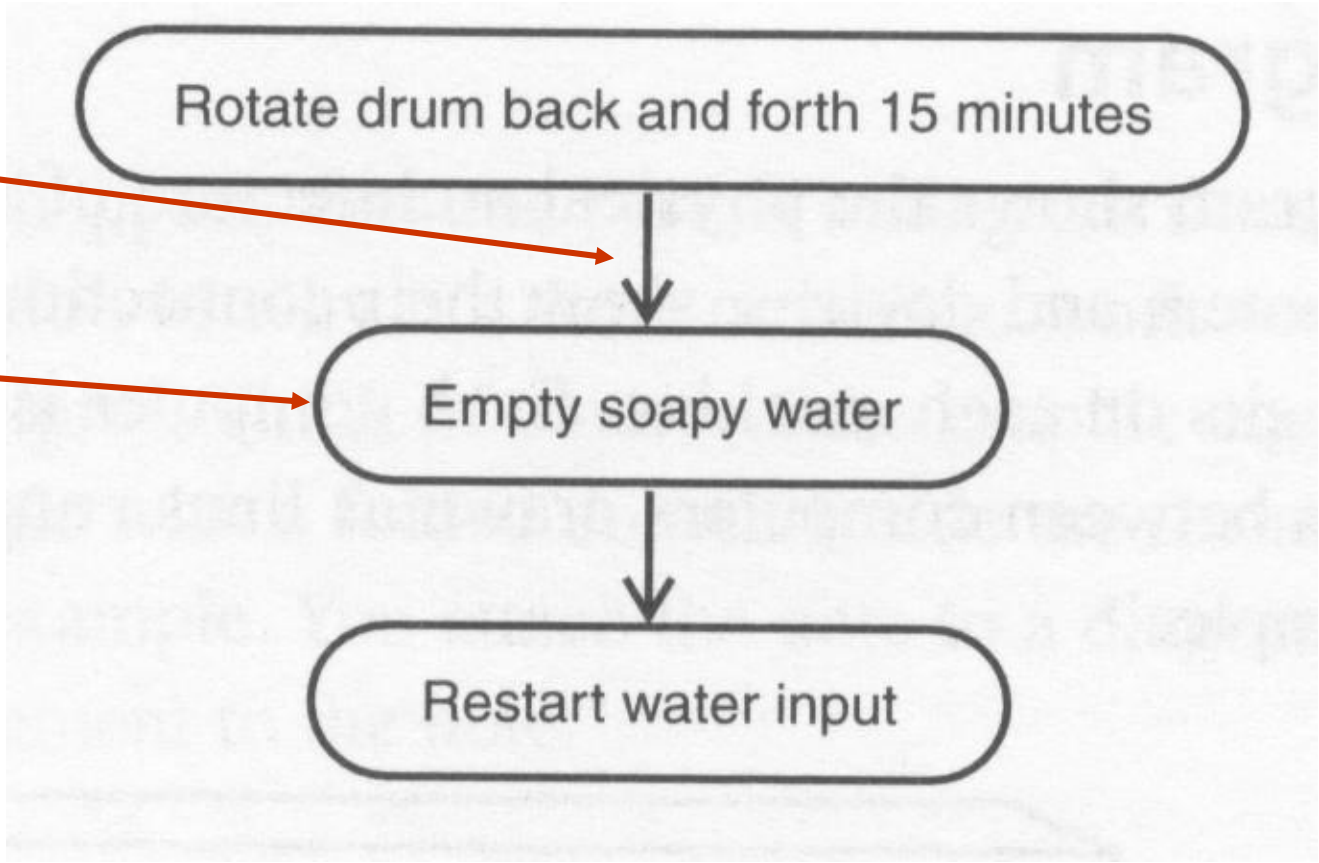
The UML State Diagram

- To show the states of a single object.



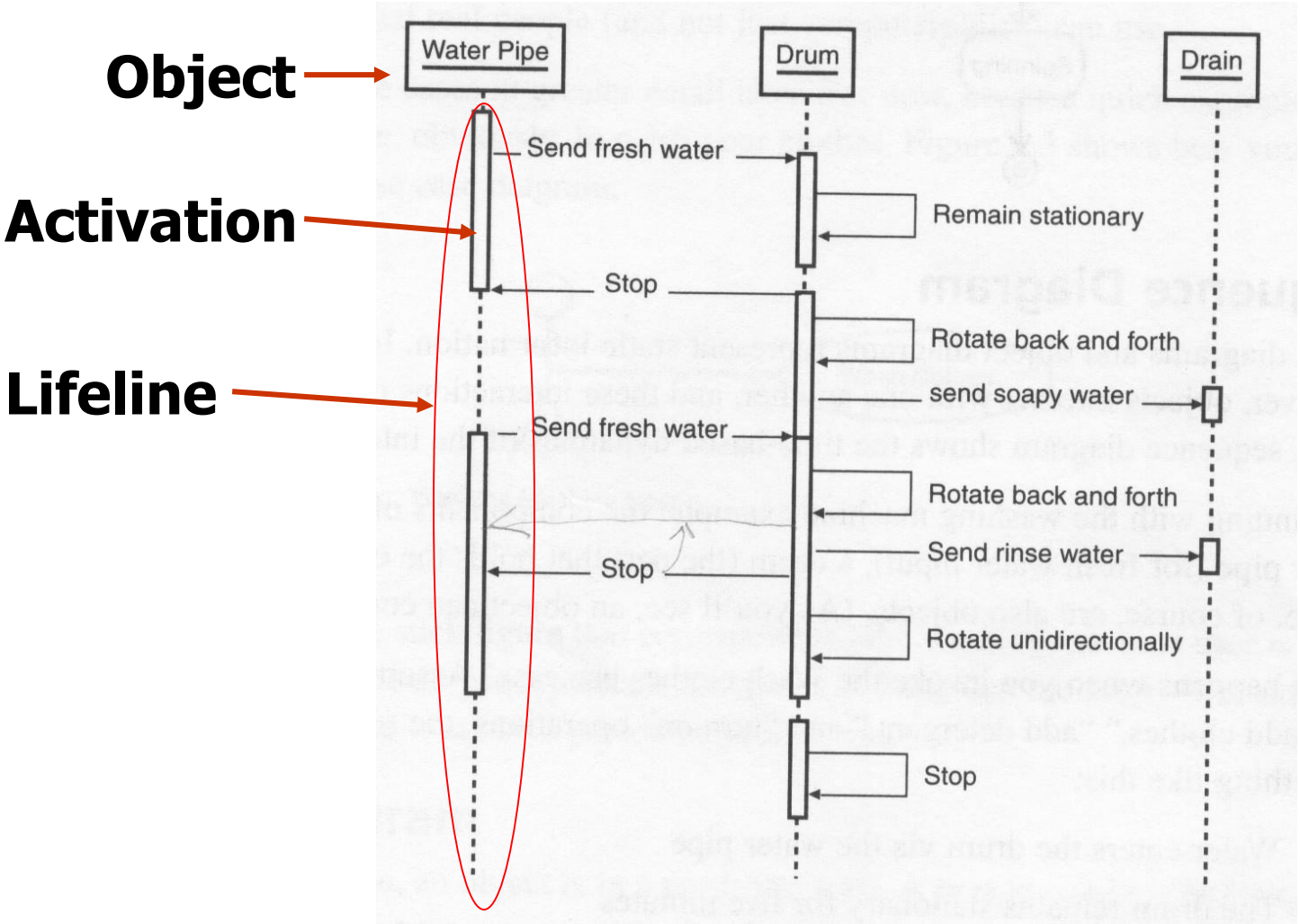
The UML Activity Diagram

- Like flowcharts, an activity diagram shows steps, decision points, and branches in a business or an operation.

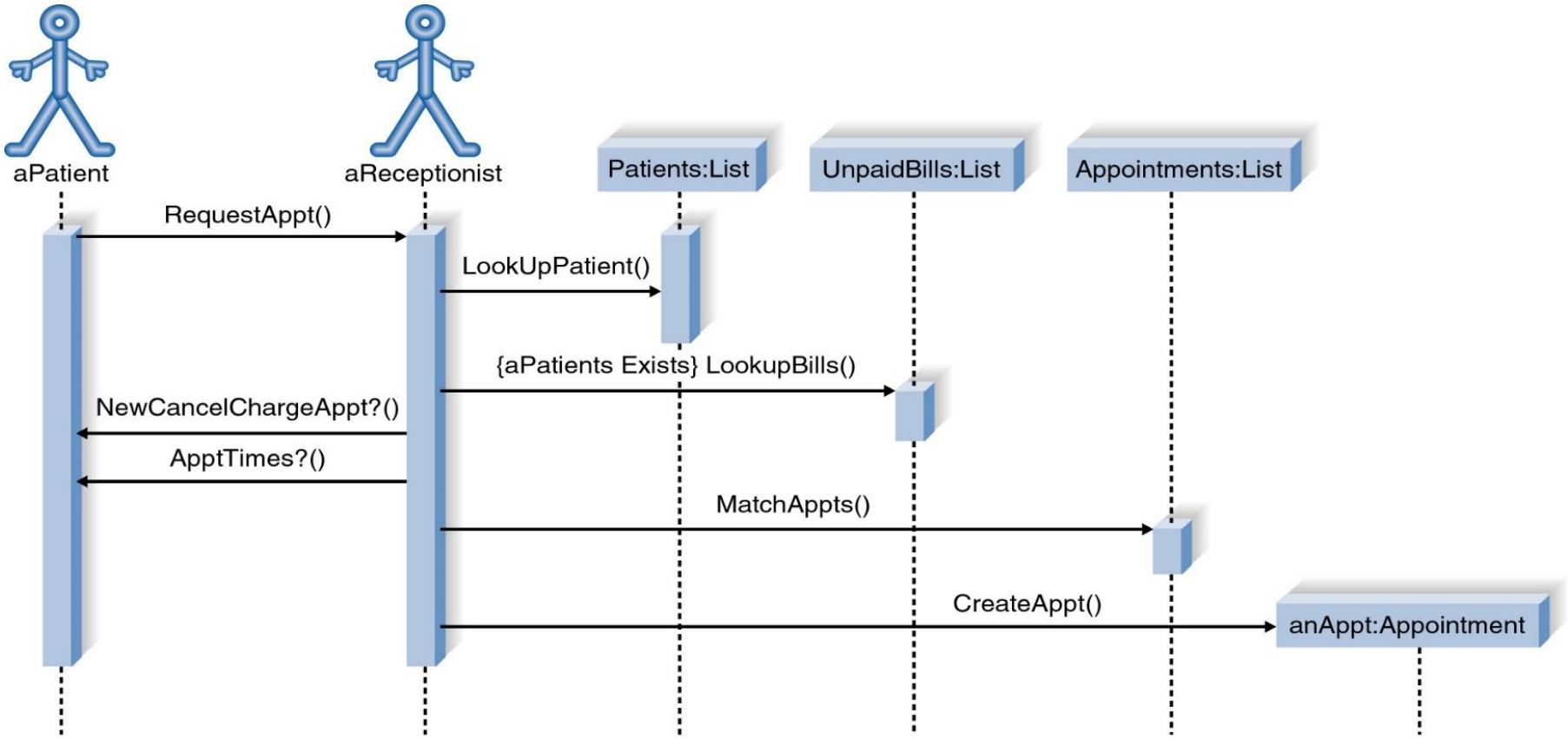


The UML Sequence Diagram

- To show how objects interact (according to time)



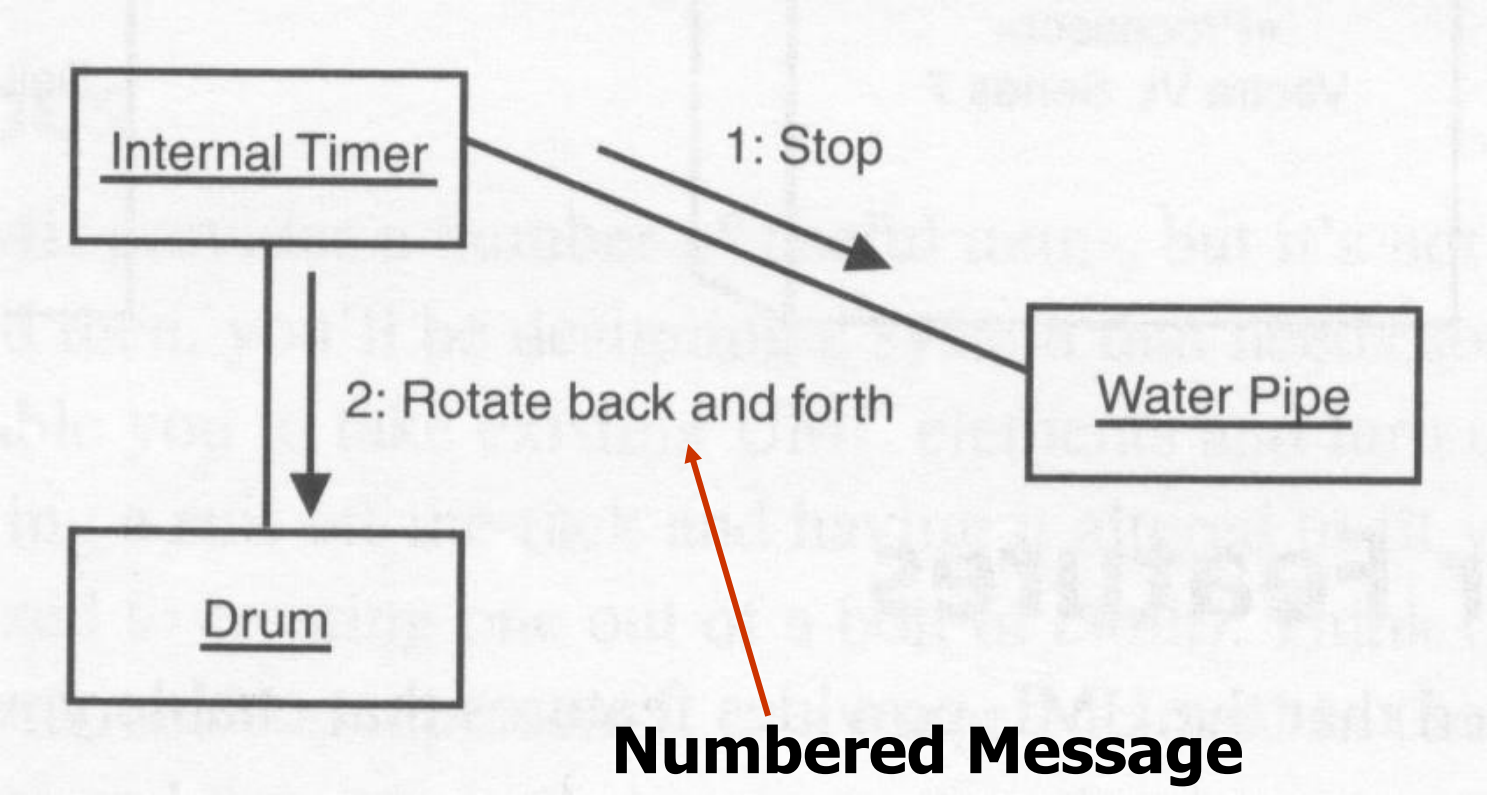
Example Sequence Diagram



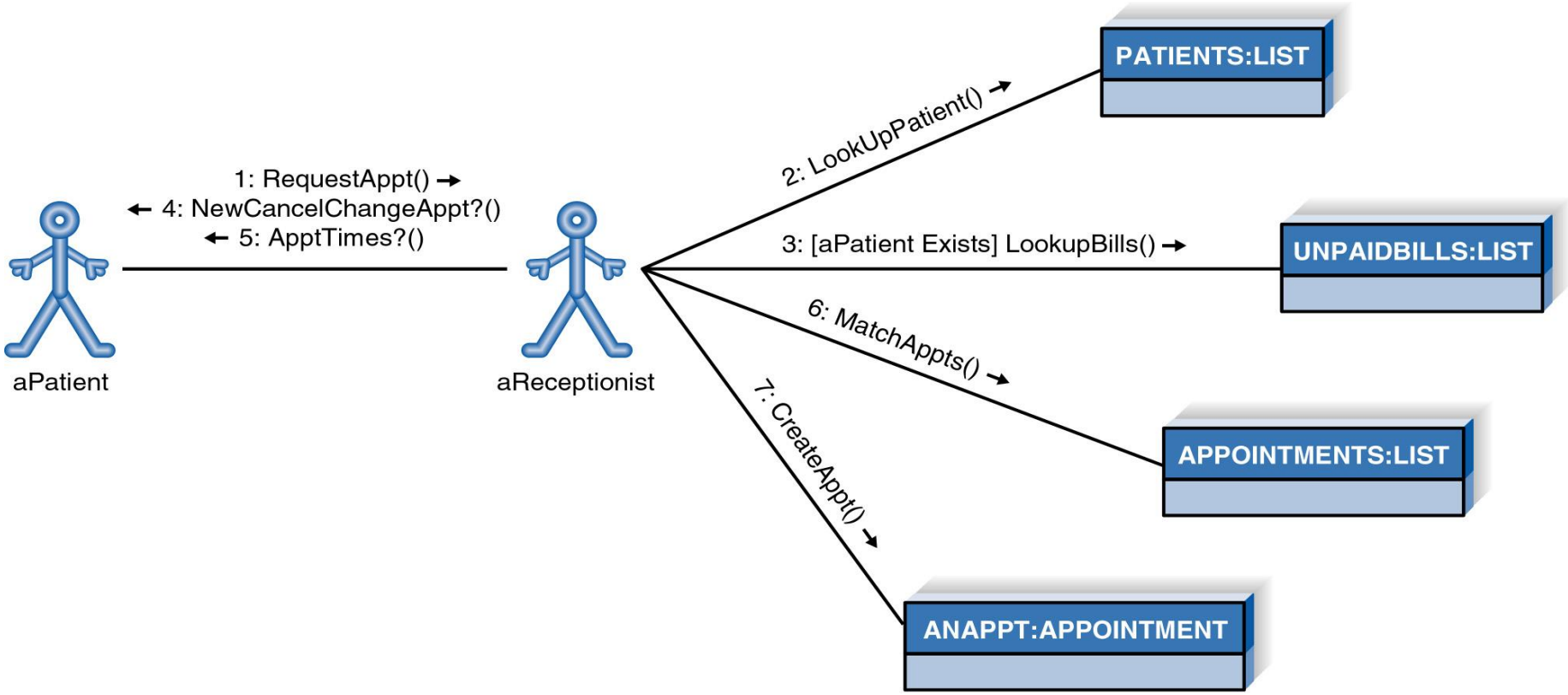
Dennis: SAD
Fig: 8-1 W-30 100% of size
Fine Line Illustrations (516) 501-0400

The UML Communication Diagram

- To show how objects interact (according to space)
- Semantically equivalent to the sequence diagram



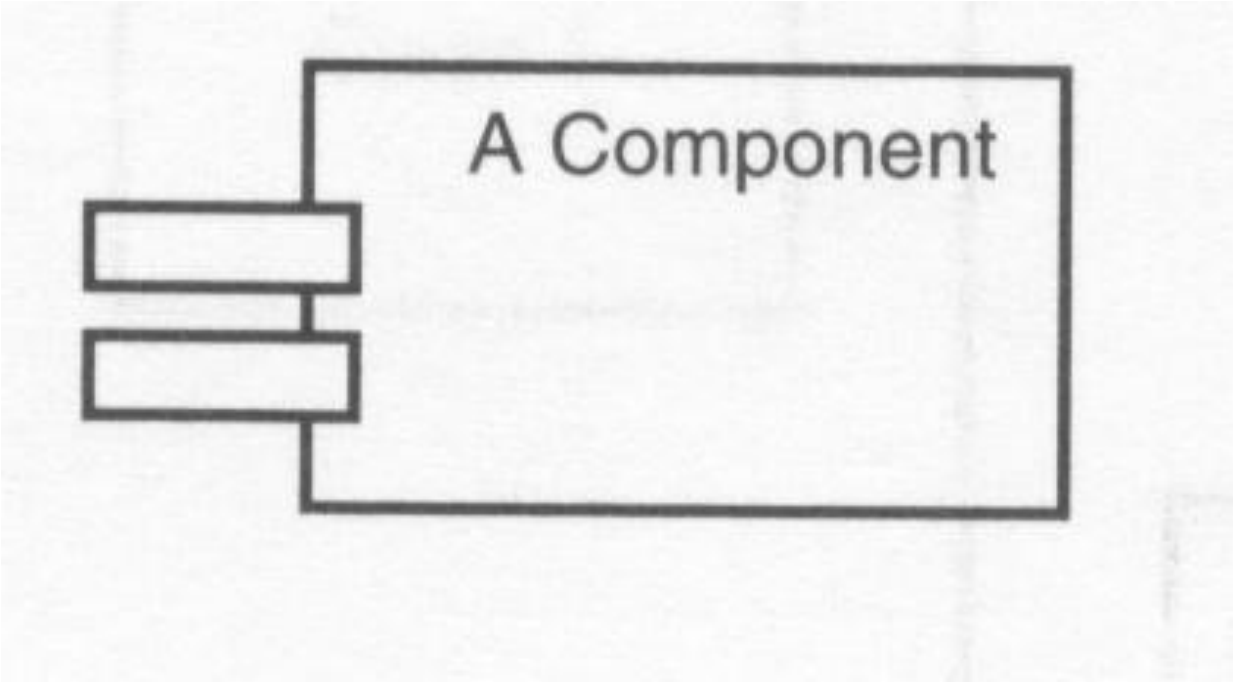
Example Communication Diagram



Dennis: SAD
Fig: 8-6 W-33 100% of size
Fine Line Illustrations (516) 501-0400

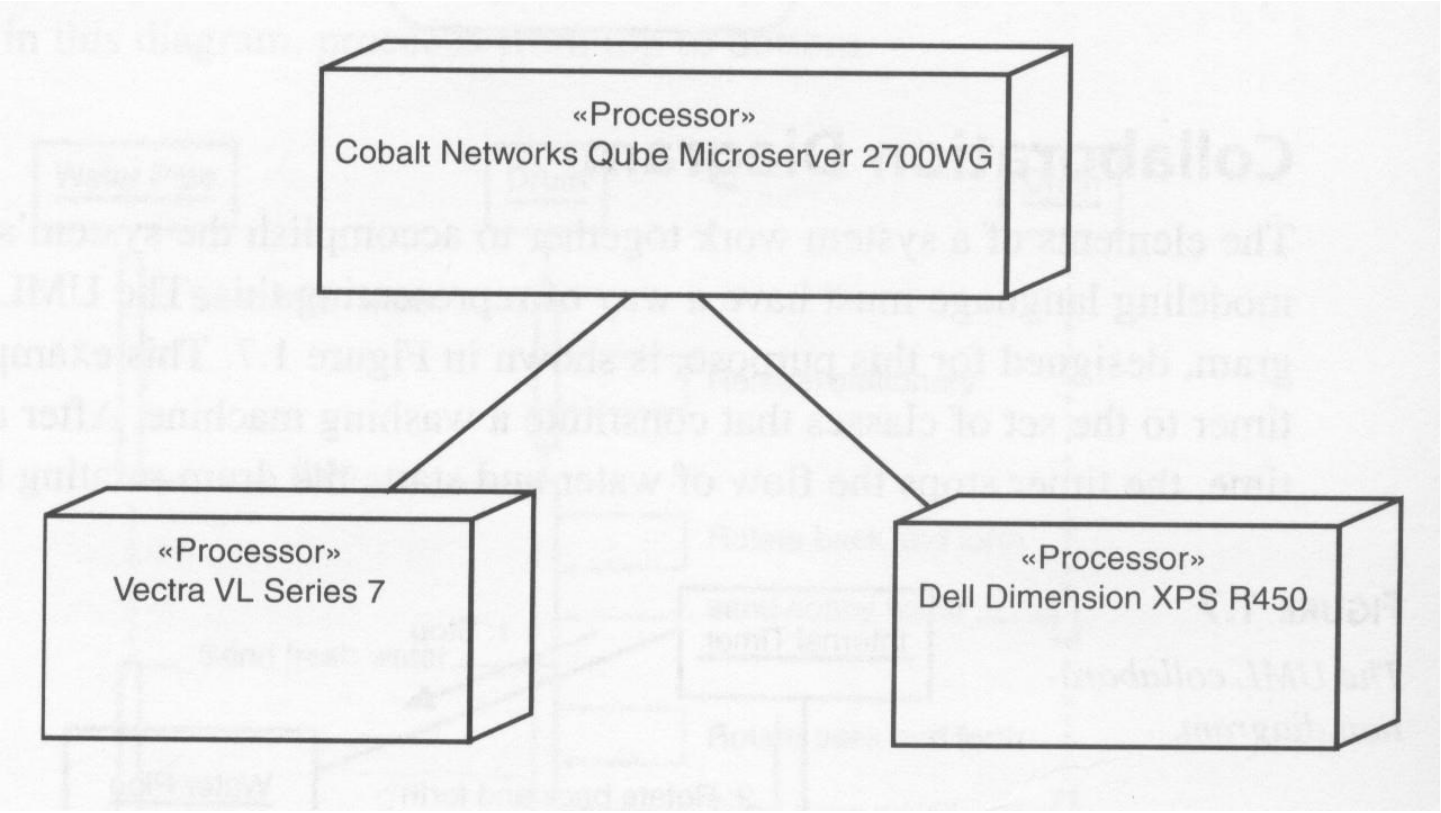
The UML Component Diagram

- To represent software components



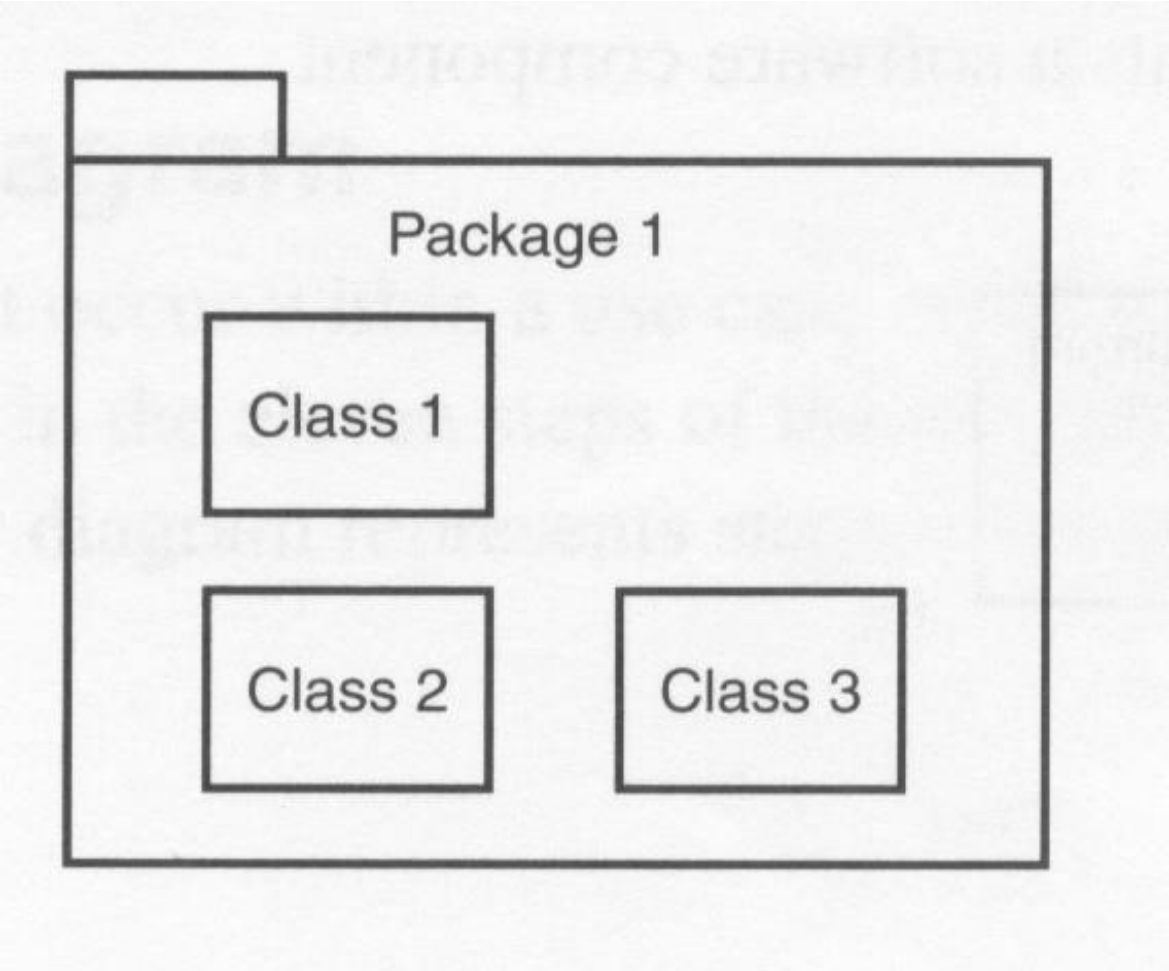
The UML Deployment Diagram

- To show the physical architecture of a computer-based system



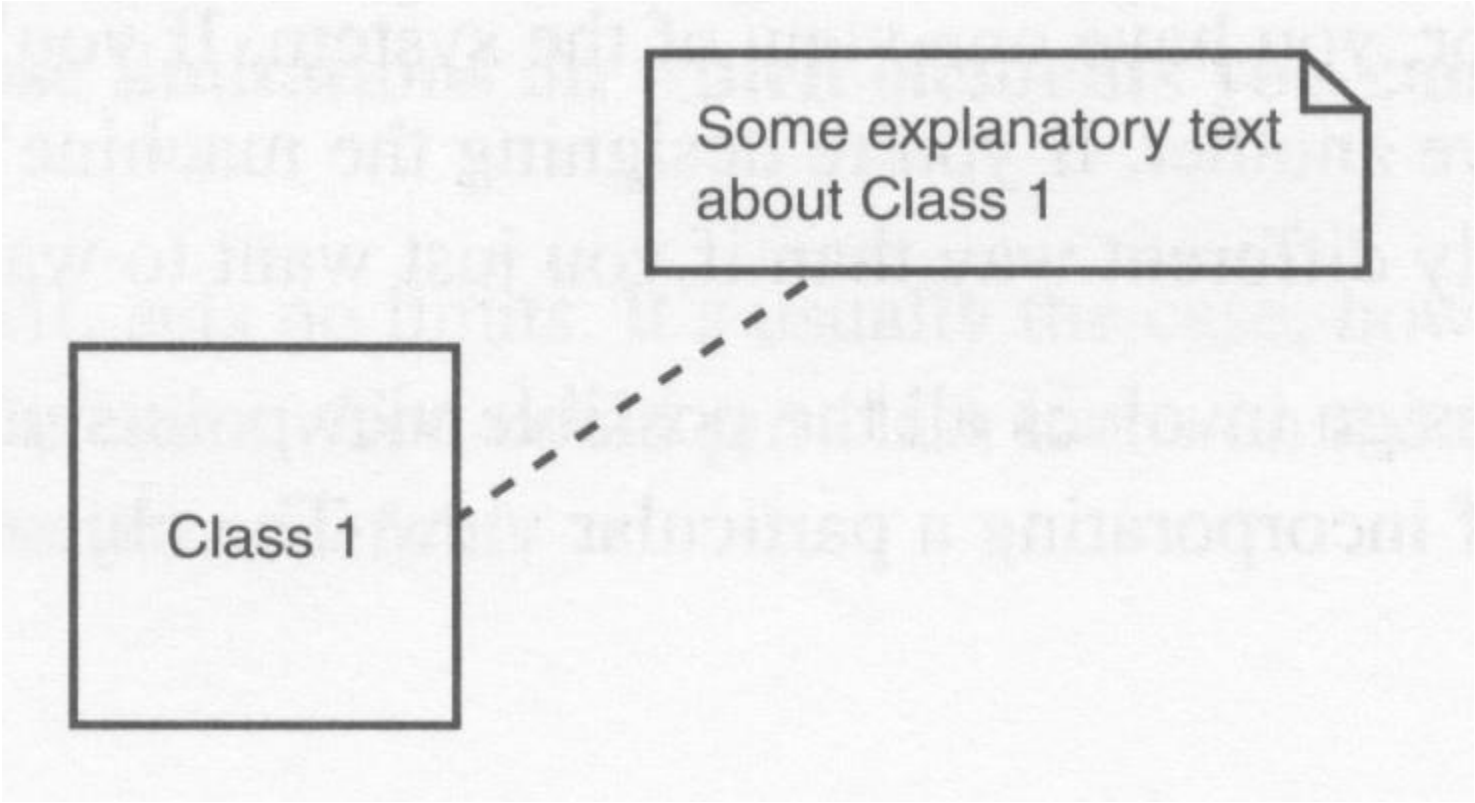
The UML Package

- To group the elements of a diagram.



Notes

- **In any diagram, you can add explanatory comments by attaching a note.**



- **Why it's important to use a number of different types of diagrams?**

Software Development Life Cycle

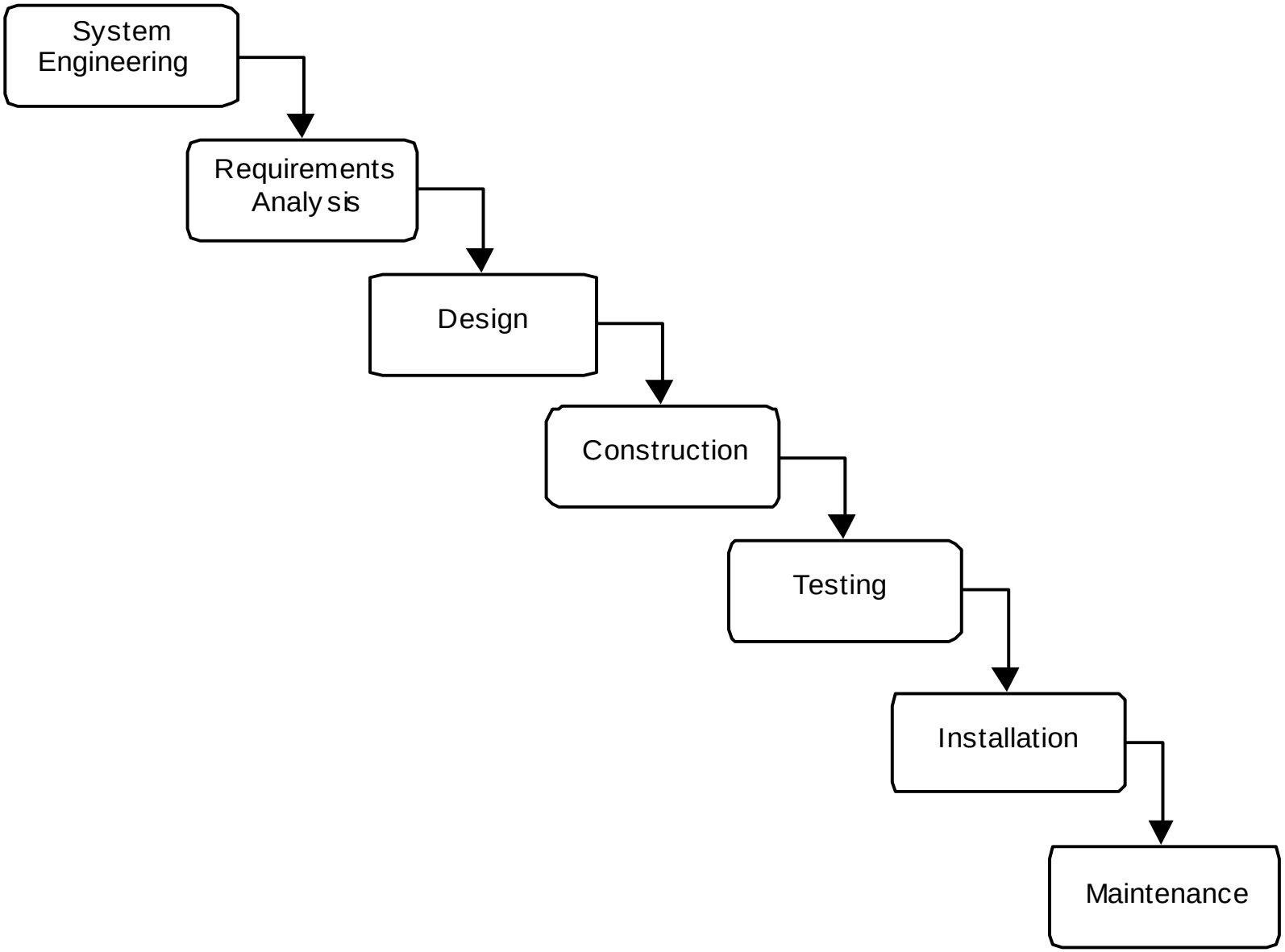
Based on Alan Dennis, Barbara Wixom, and David Tegarden

© 2005
John Wiley & Sons, Inc.

Waterfall Life Cycle

- The traditional life cycle (TLC) for information systems development is also known as the waterfall life cycle model
  - So called because of the difficulty of returning to an earlier phase
- The model shown here is one of several more or less equivalent alternatives
  - Typical deliverables are shown for each phase

Traditional Waterfall Life Cycle



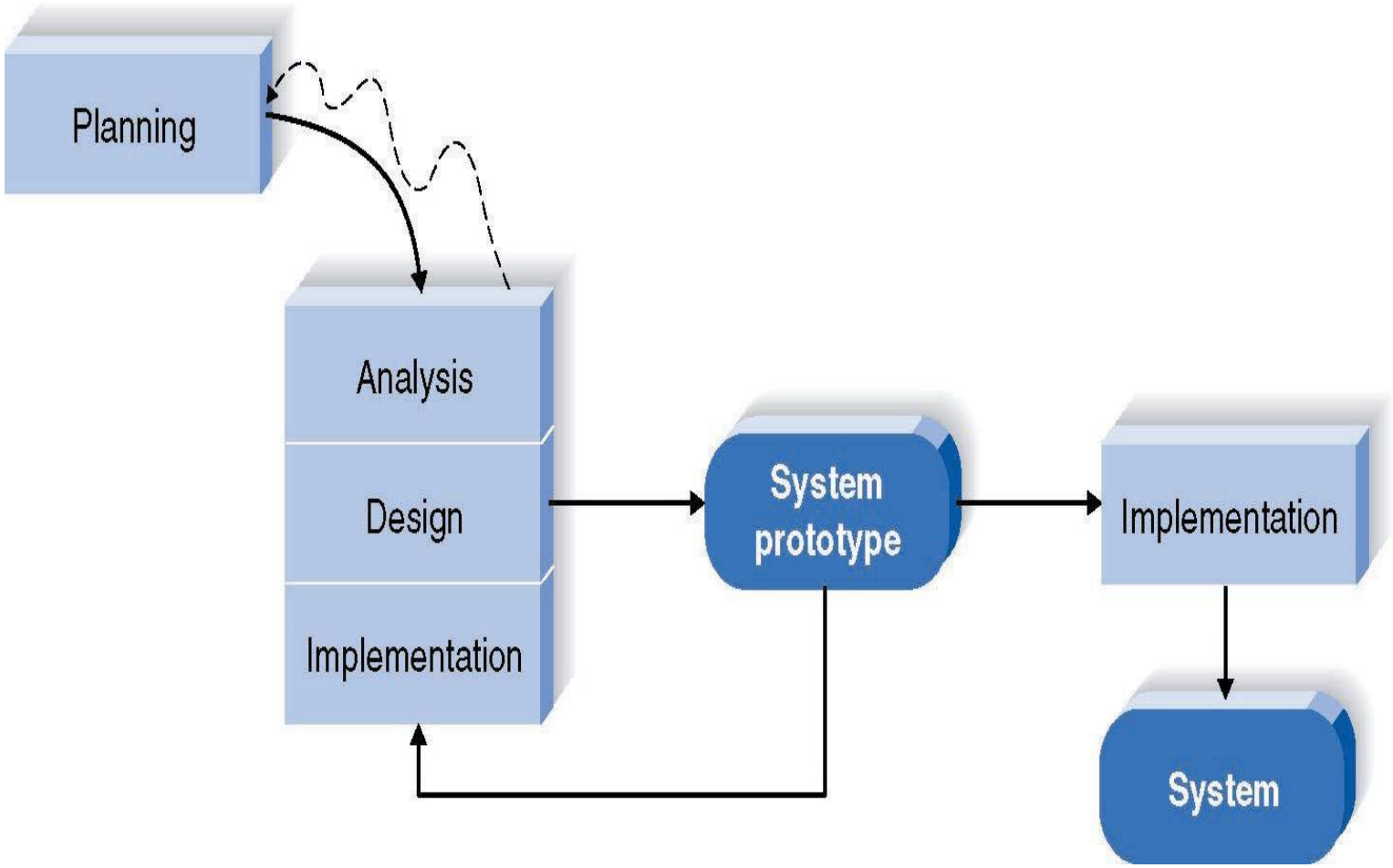
Pros and Cons of the Waterfall Method

Pros	Cons
Identifies systems requirements long before programming begins	Design must be specified on paper before programming begins
	Long time between system proposal and delivery of new system

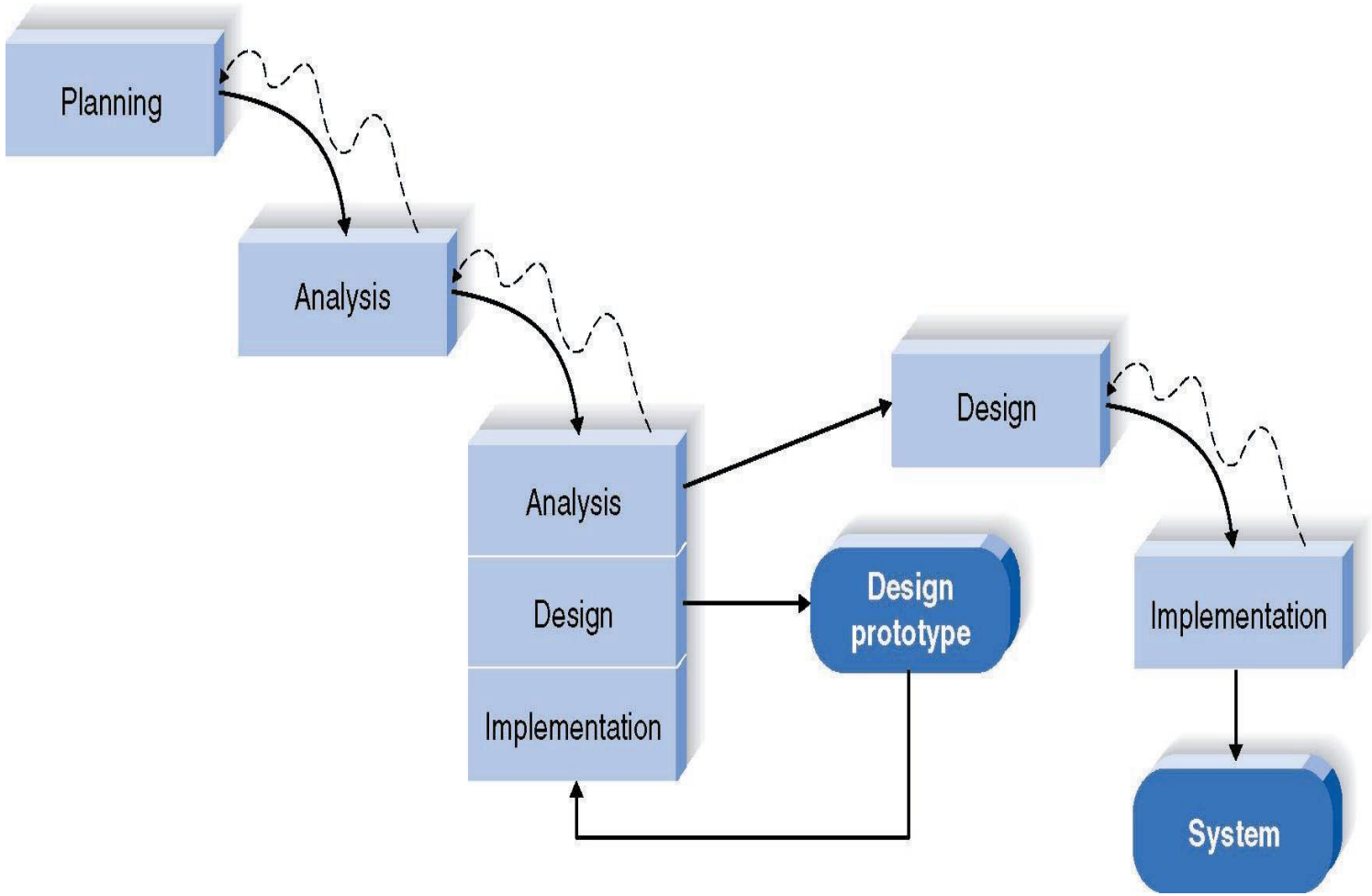
Rapid Application Development Categories

- Prototyping
  - System prototyping
- Throw-away prototyping
  - Design prototyping

How Prototyping Works



Throwaway Prototyping



Prototyping—Advantages

- Early demonstrations of system functionality help identify any misunderstandings between developer and client
- Client requirements that have been missed are identified
- Difficulties in the interface can be identified
- The feasibility and usefulness of the system can be tested, even though, by its very nature, the prototype is incomplete

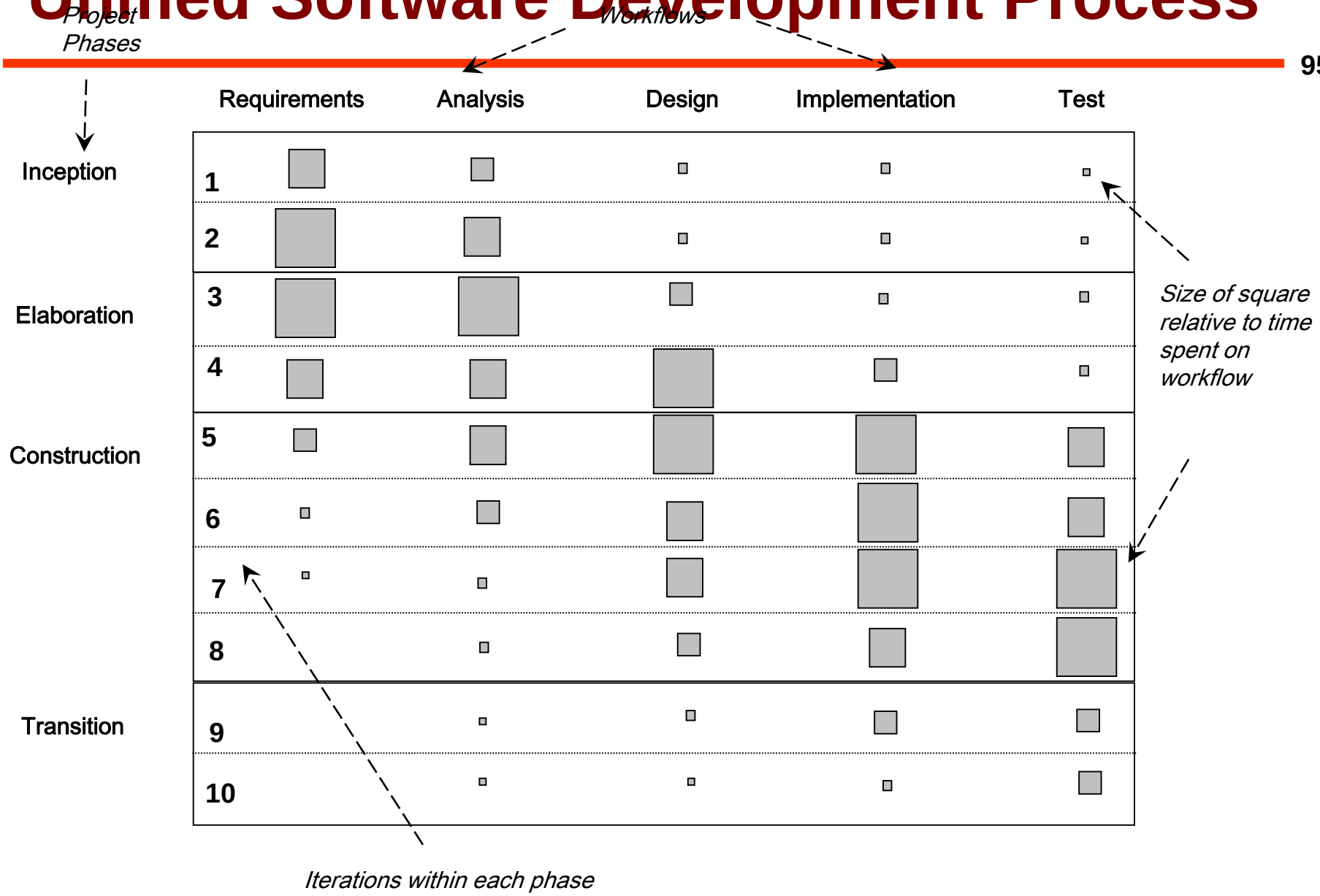
Prototyping—Problems

- The client may perceive the prototype as part of the final system
- The prototype may divert attention from functional to solely interface issues
- Prototyping requires significant user involvement
- Managing the prototyping life cycle requires careful decision making

Unified Software Development Process

- Captures many elements of best practice
- Main phases
  - *Inception* is concerned with determining the scope and purpose of the project
  - *Elaboration* focuses requirements capture and determining the structure of the system
  - *Construction's* main aim is to build the software system
  - *Transition* deals with product installation and rollout

Unified Software Development Process



UML Tools

- Microsoft Visio
- IBM Rational Software Architect
- Visual Paradigm
- Sparx Systems Enterprise Architect
- JUDE
- Software Ideas Modeler (SIM)

  https://www.softwareideas.net/

Example Papers containing UML Diagrams

[1] H.-C. Huang, Y.-C. Lin, **M.-H. Hung***, C.-C. Tu, and F.-T. Cheng, “Development of Cloud-based Automatic Virtual Metrology System for Semiconductor Industry,” *Robotics and Computer-Integrated Manufacturing*, Vol. 34, pp. 30-43, Aug. 2015. [SCI]

[2] C.-C. Chen*, Y.-C. Lin, M.-H. Hung, C.-Y. Lin, Y.-J. Tsai, and F.-T. Cheng, “A Novel Cloud Manufacturing Framework with Auto-Scaling Capability for Machine Tool Industry,” *International Journal of Computer-Integrated Manufacturing*, Vol. 29, No. 7, pp. 786–804, 2016. [SCI]

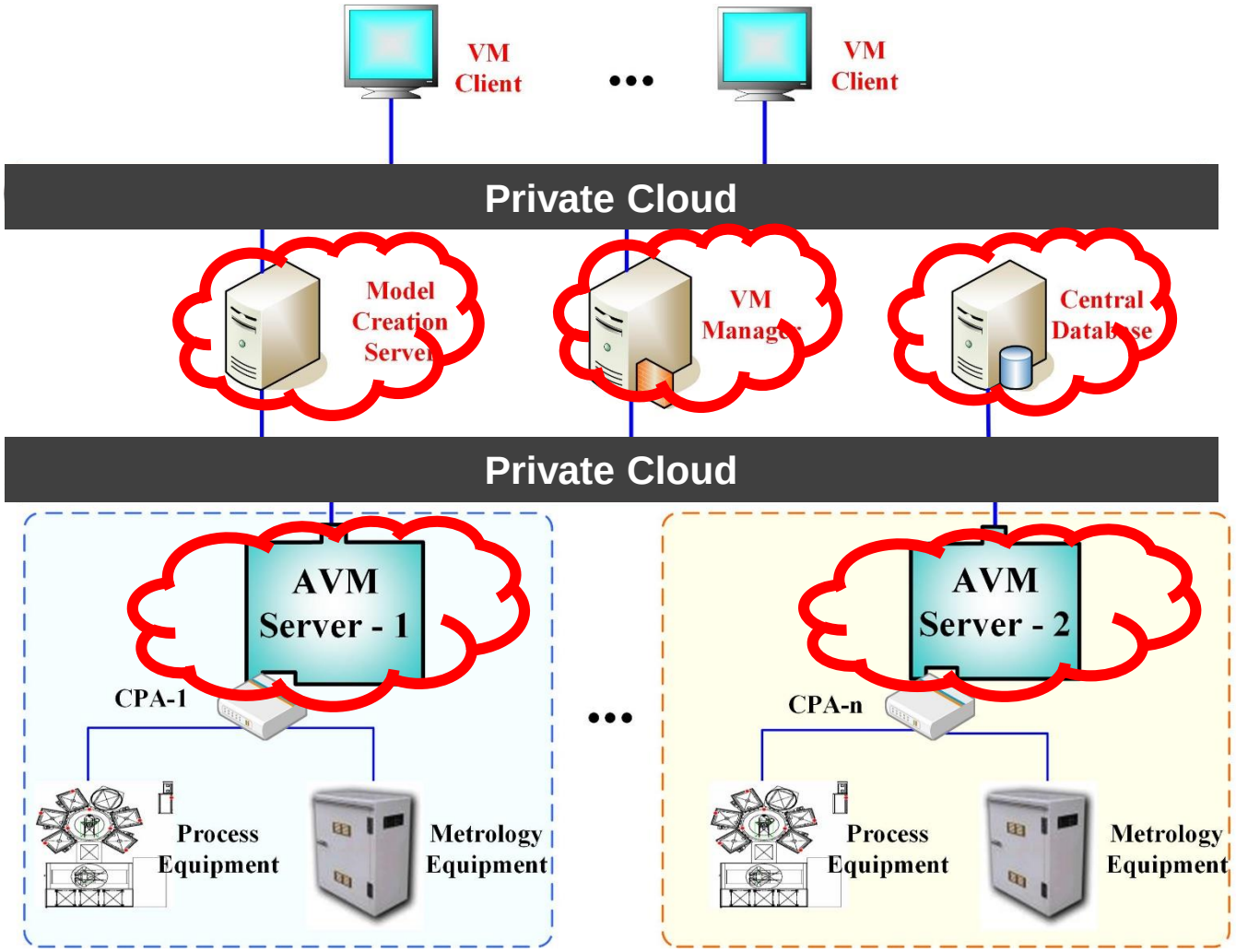
Some Limitations of the Existing AVM System

- **Existing AVM systems have several limitations in a factory-wide deployment with a great number of equipment:**
  - **Incurring high hardware cost, occupying a great volume of shop-floor space, and needing complex management efforts.**
  - **Deployment of VM servers needs to be conducted manually, which would be cumbersome and prone to error.**
  - **The AVM system cannot automatically increase or decrease the number of VM servers on demand, usually resulting in over-provisioning (having idle VM servers) or under-provisioning (having insufficient VM servers).**

Objectives of this Work

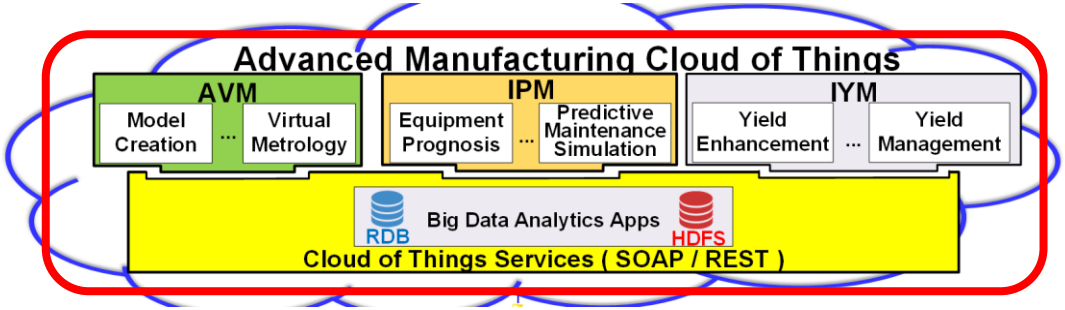
- Propose an approach of building cloud-based AVM systems, which could
  - effectively remedy the above-mentioned shortcomings of existing AVM systems in plant-wide deployment and in model creation functionality
  - keep the reworking efforts as little as possible to transform a AVM system into a cloud-based one.

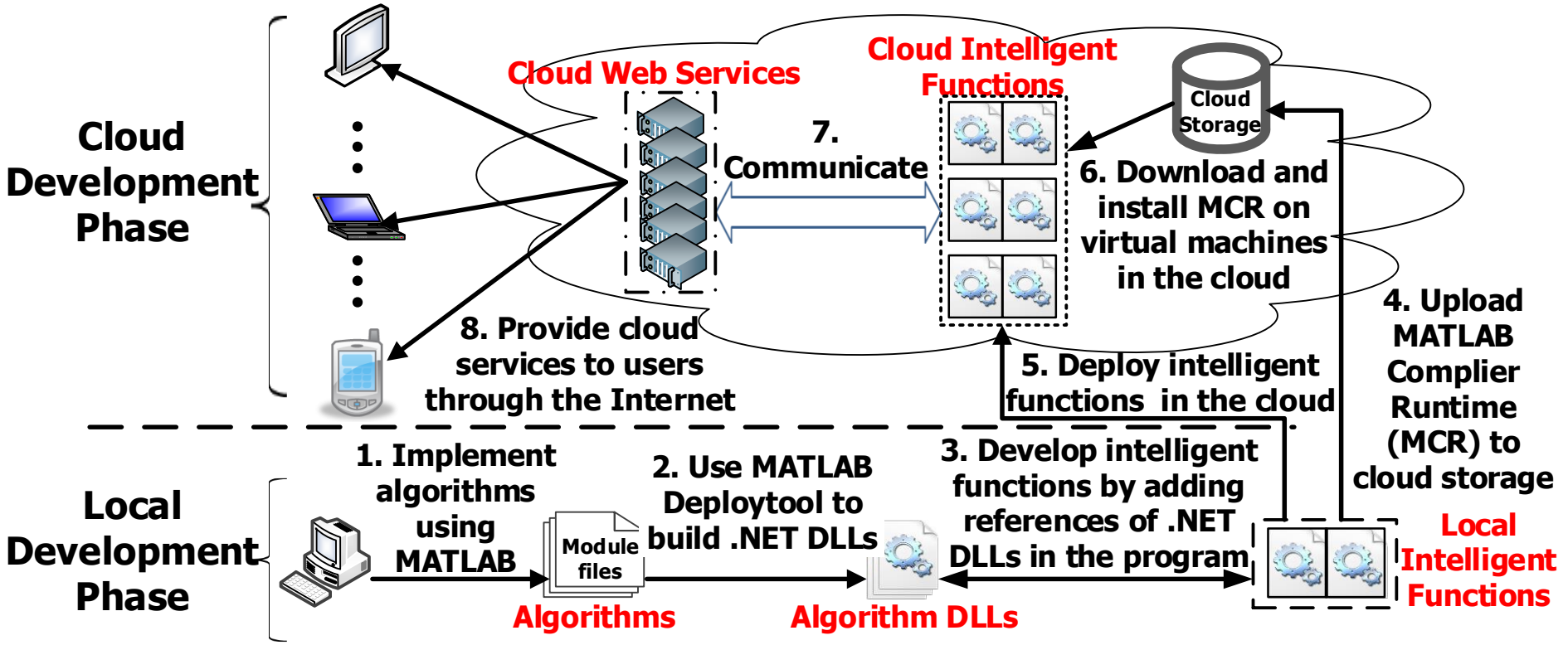
Cloud-based AVM System



- H.-C. Huang, Y.-C. Lin, M.-H. Hung, C.-C. Tu, and F.-T. Cheng, "Development of Cloud-based Automatic Virtual Metrology System for Semiconductor Industry," *Robotics and Computer-Integrated Manufacturing*, Vol. 34, pp. 30-43, Feb. 2015.

Generic Procedure for Building Intelligent Manufacturing Cloud Services

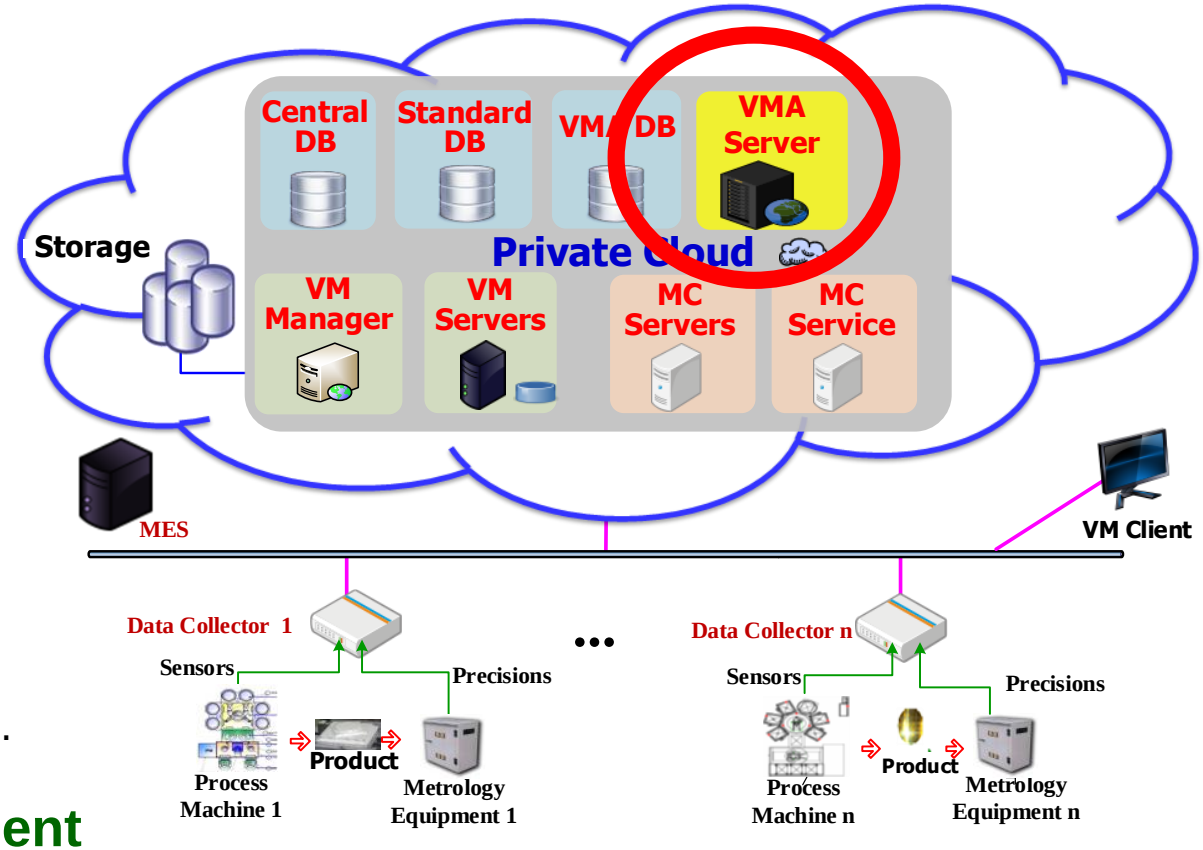




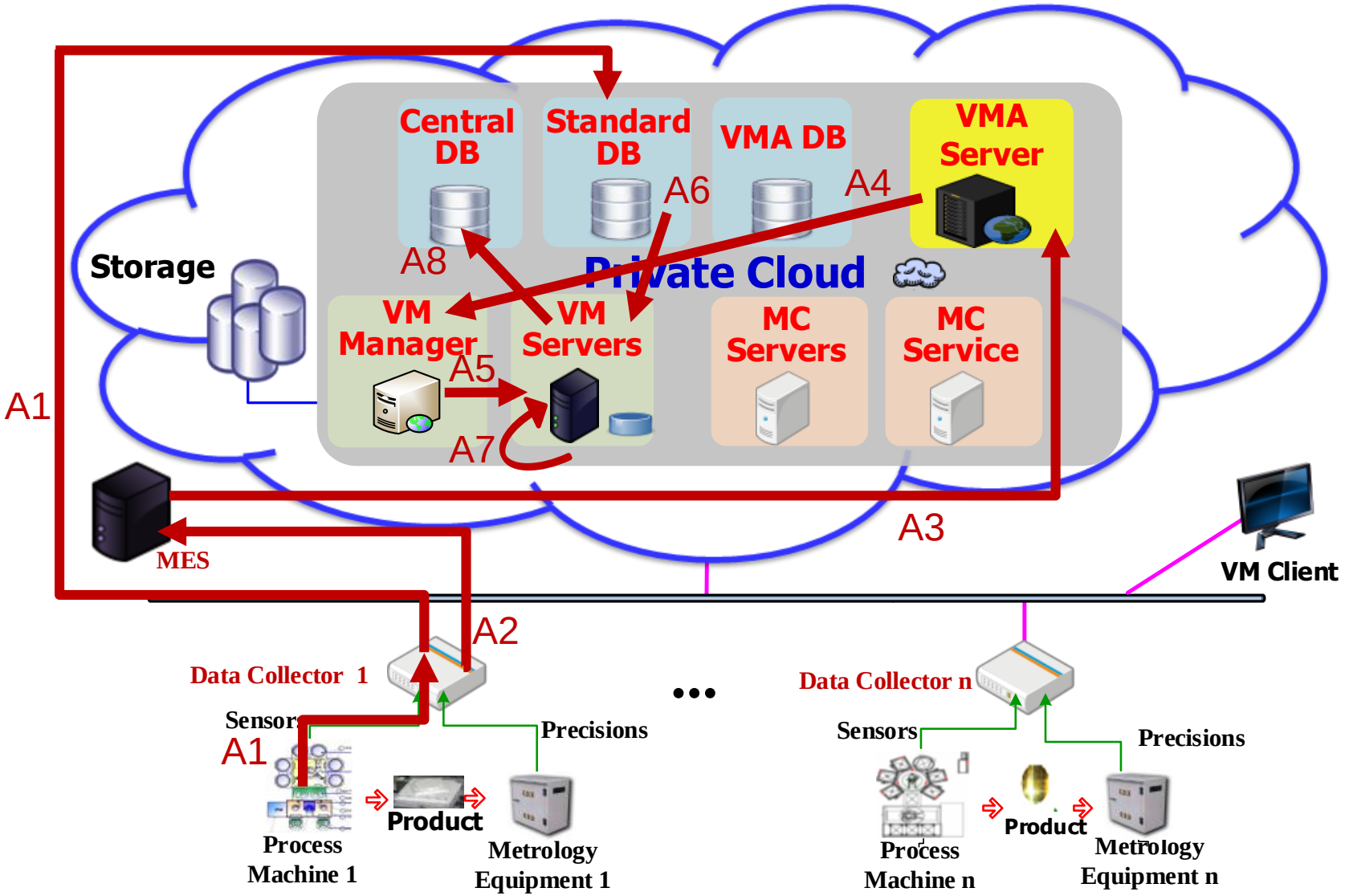
Architecture of the cloud-based AVM system

1. **Create a private cloud environment**
   VMware vSphere and its associated software, are used to create a private cloud environment.
2. **Virtualizing all servers**
   Leaving the original implementation codes of each server unchanged.
3. **Design an Virtual Machine Administrator (VMA)**
   Host and perform the major designed functional mechanisms.

- **Automatic-Deployment**
- **Automatic-Scaling**
- **Automatic-Serving**



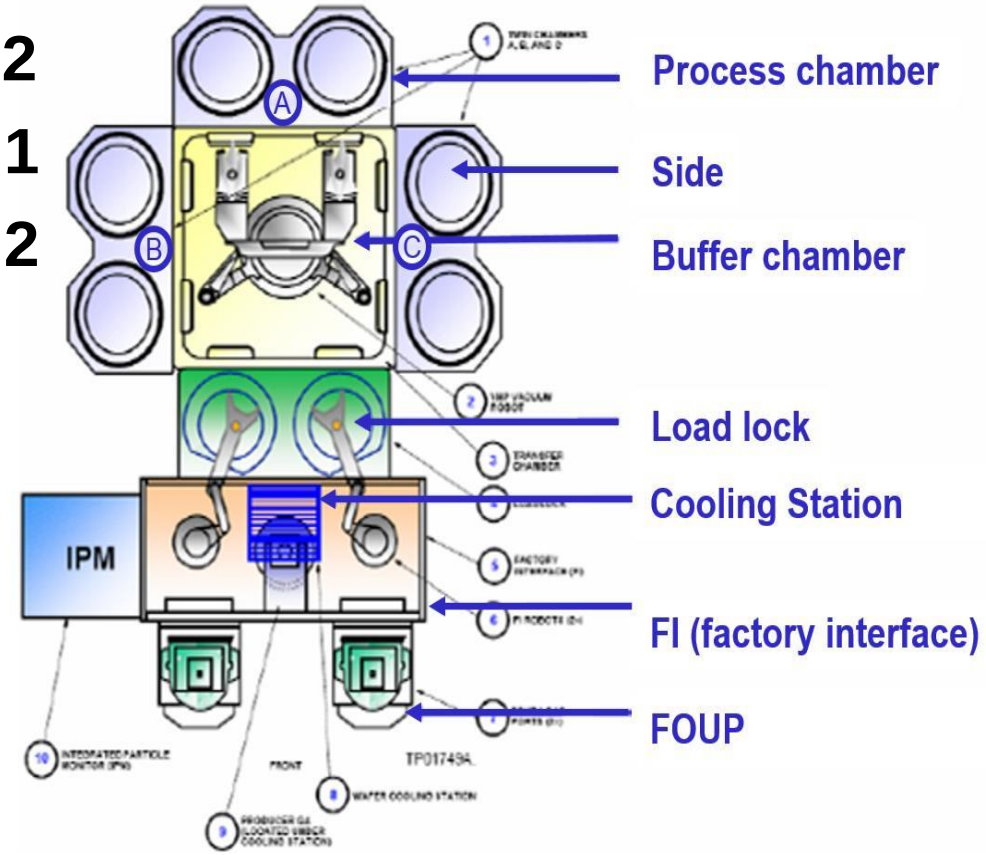
Operation Flow



Terminologies (1/2)

Combination Information of Equipment

- A01, Chamber A, Side 1
- A01, Chamber A, Side 2
- A01, Chamber B, Side 1
- A01, Chamber B, Side 2



Architecture of the ULKCVD Equipment A01

Terminologies (2/2)

- **Template Virtual Machine**

  A virtual machine installed with the VM server software, but lacking the combination information of equipment, prediction model, and IP.

- **Spare Virtual Machine**

  When a cloned virtual machine is turned on and assigned with an IP, it is called a spare virtual machine, which is ready to be configured become a virtual VM server.。

- **VM Server**

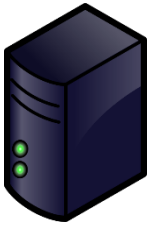
  When the spare virtual machine is equipped with prediction models and the combination information of equipment, it becomes a VM server.

Template Virtual Machine

[Clone]

Spare Virtual Machine

[prediction model+ combination information+IP]



VM Server

Schema Design of VMA Database

- **CombinationDEF Table:**
  Define the combination information.

Case	UMC	UMC	UMC	FATEK	FATEK
FieldName	Field1	Field2	Field3	Field1	Field2
Def	DEP_EQPID	DEP_CHAMBER	DEP_SIDE	STAGE	STEP

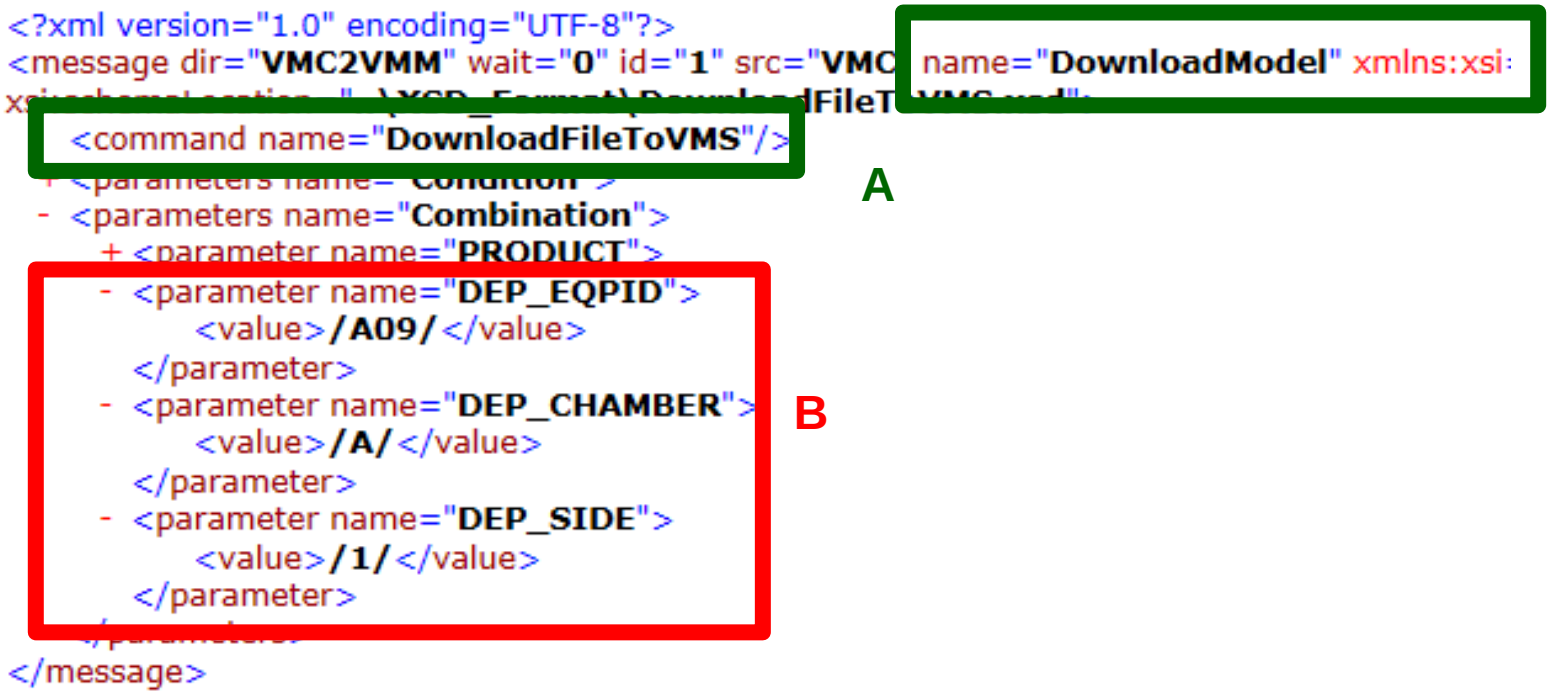
- **VMList Table: Store the statuses of virtual machines.**

IP	192.168.0.15	192.168.0.16	192.168.0.17	192.168.0.18
VMName	UMC-VMS-15	UMC-VMS-16	FATEK-VMS-17	ScaleOutVM
VMStatus	On	Off	On	On
LastestUsageTime	2013-08-12 17:21:47	2013-08-12 17:21:50	2014-02-12 14:15:32	
IPUsed	true	true	true	true
CombinationStatus	true	true	true	False
Field 1	A01	A01	OP1	
Field 2	A	A	2	
Field 3	1	2		
Case	UMC	UMC	FATEK	

active VM server VM server turned off spare virtual machine

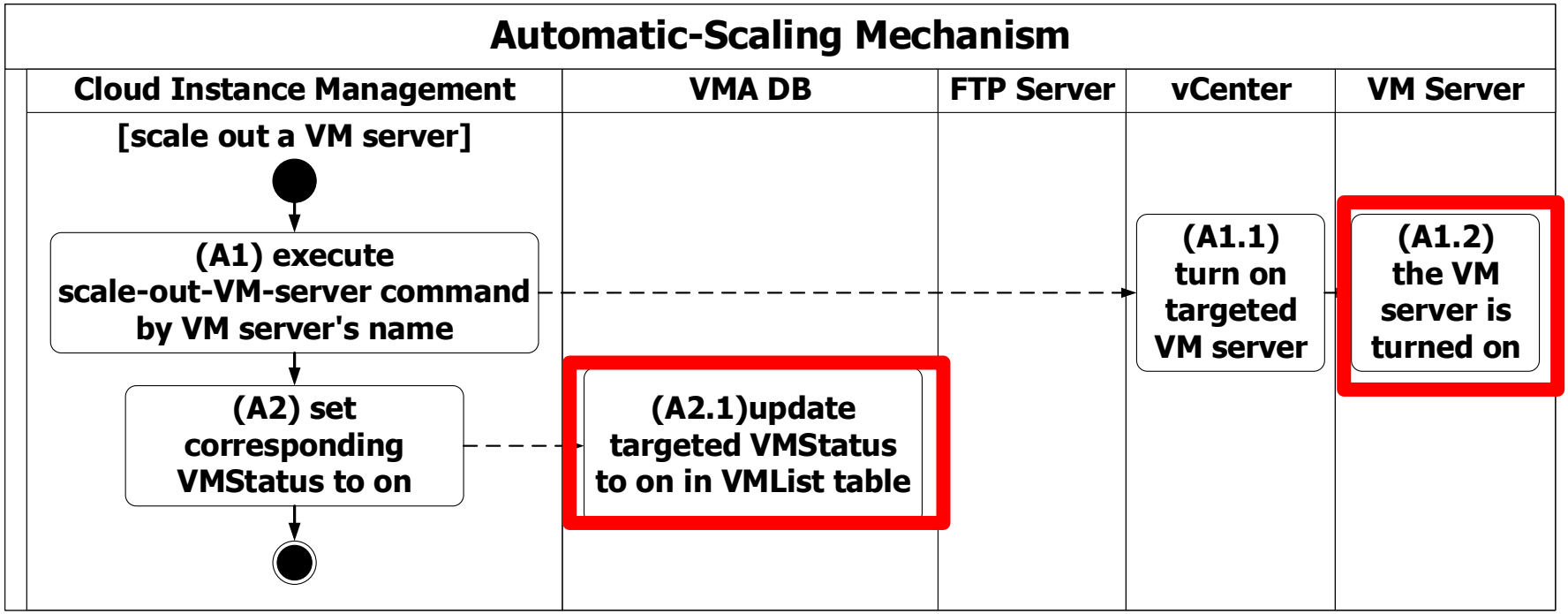
Design of MES Command

- By parsing and extracting the information from the DEP_EQPID, DEP_CHAMBER, and DEP_SID elements, the VMA server is able to determine the combination for setting the VM server to serve the requested VM task.



Design of Automatic-Scaling Mechanism (1/3)

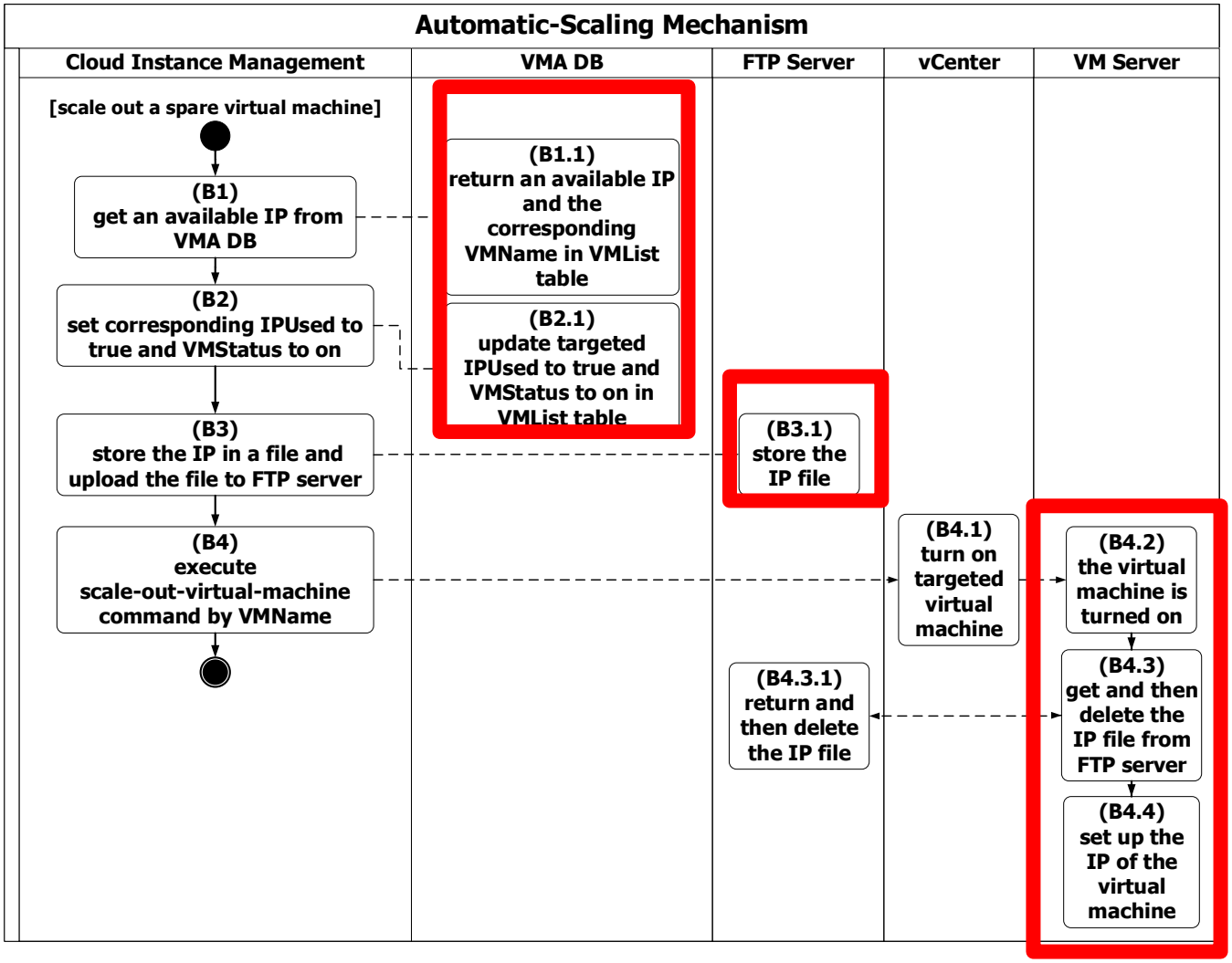
- **Scale out a VM server:**
  to turn on a VM server



Design of Automatic-Scaling Mechanism(2/3)

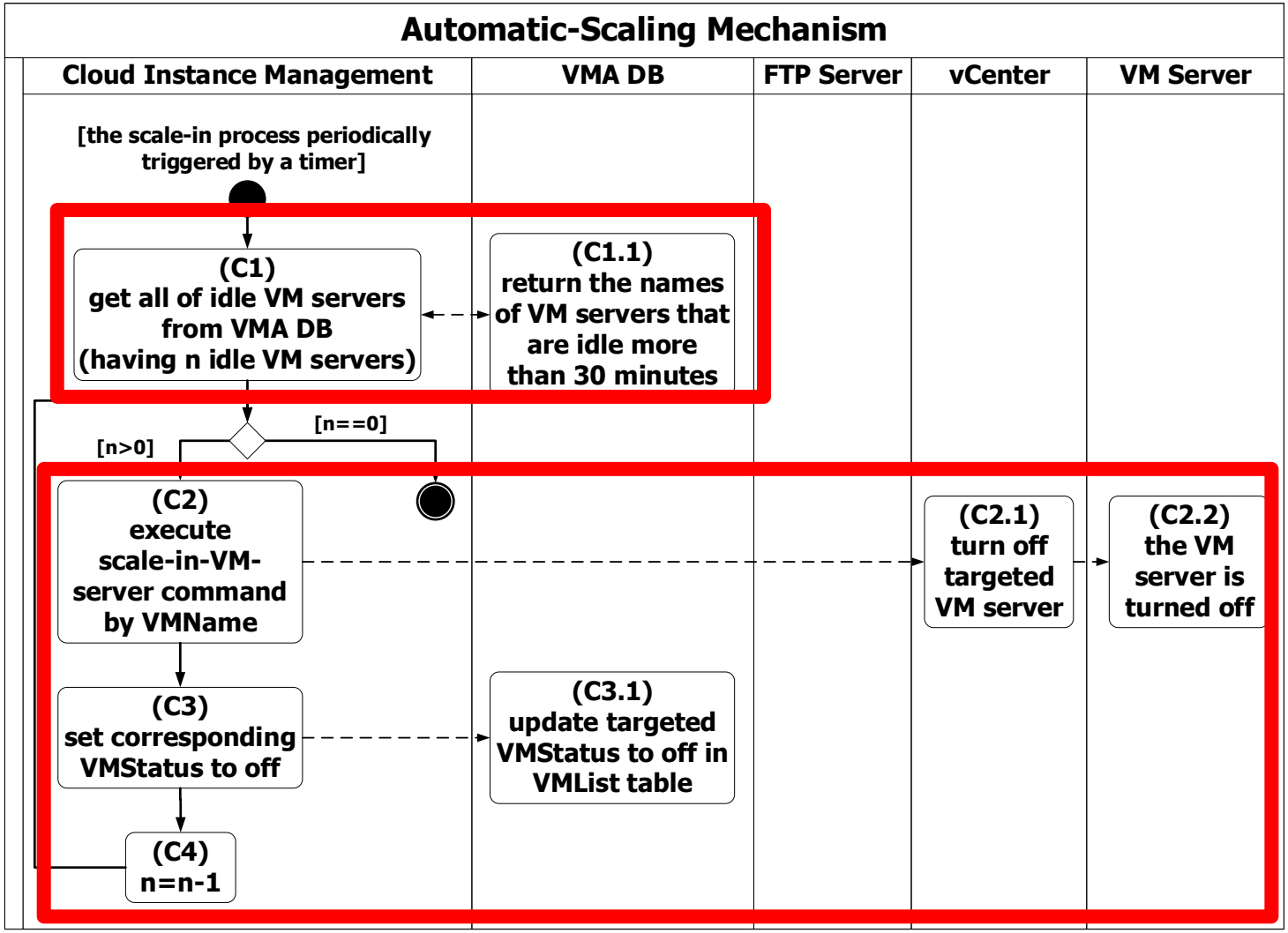
- Scale Out a Spare Virtual Machine:

  to generate a new spare virtual machine



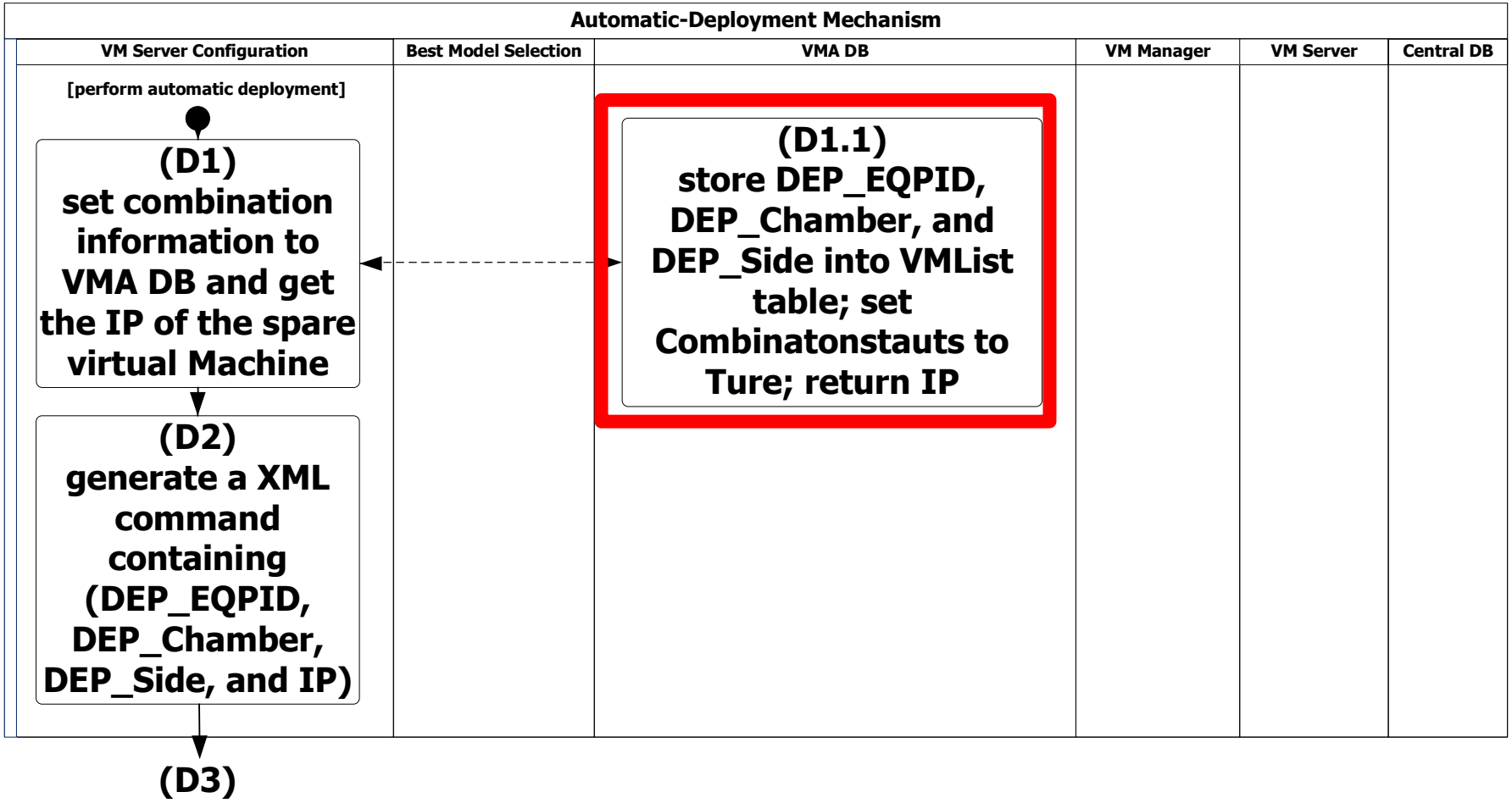
Design of Automatic-Scaling Mechanism(3/3)

- Scale in a VM server:
  to turn off a VM server (periodically triggered by a timer)

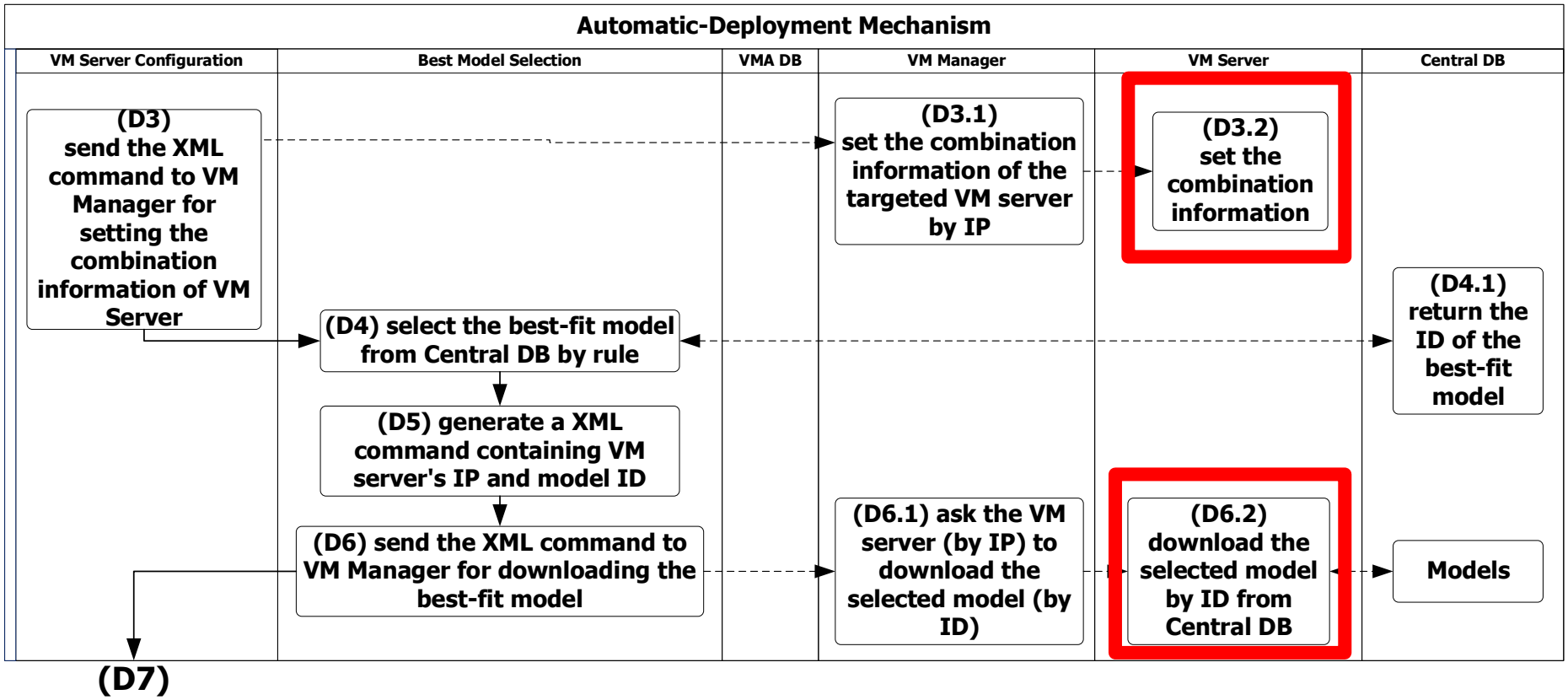


Design of Automatic-Deployment Mechanism(1/3)

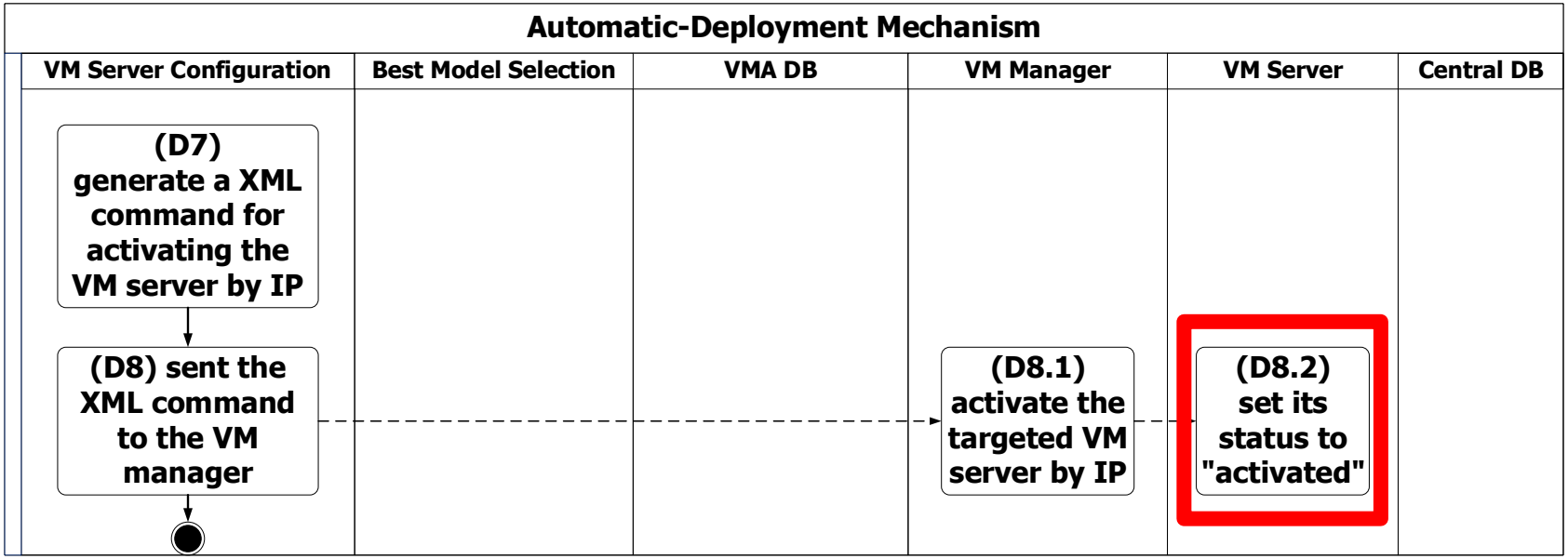
- This mechanism is to automatically transform a spare virtual machine into an active VM server:



Design of Automatic-Deployment Mechanism(2/3)

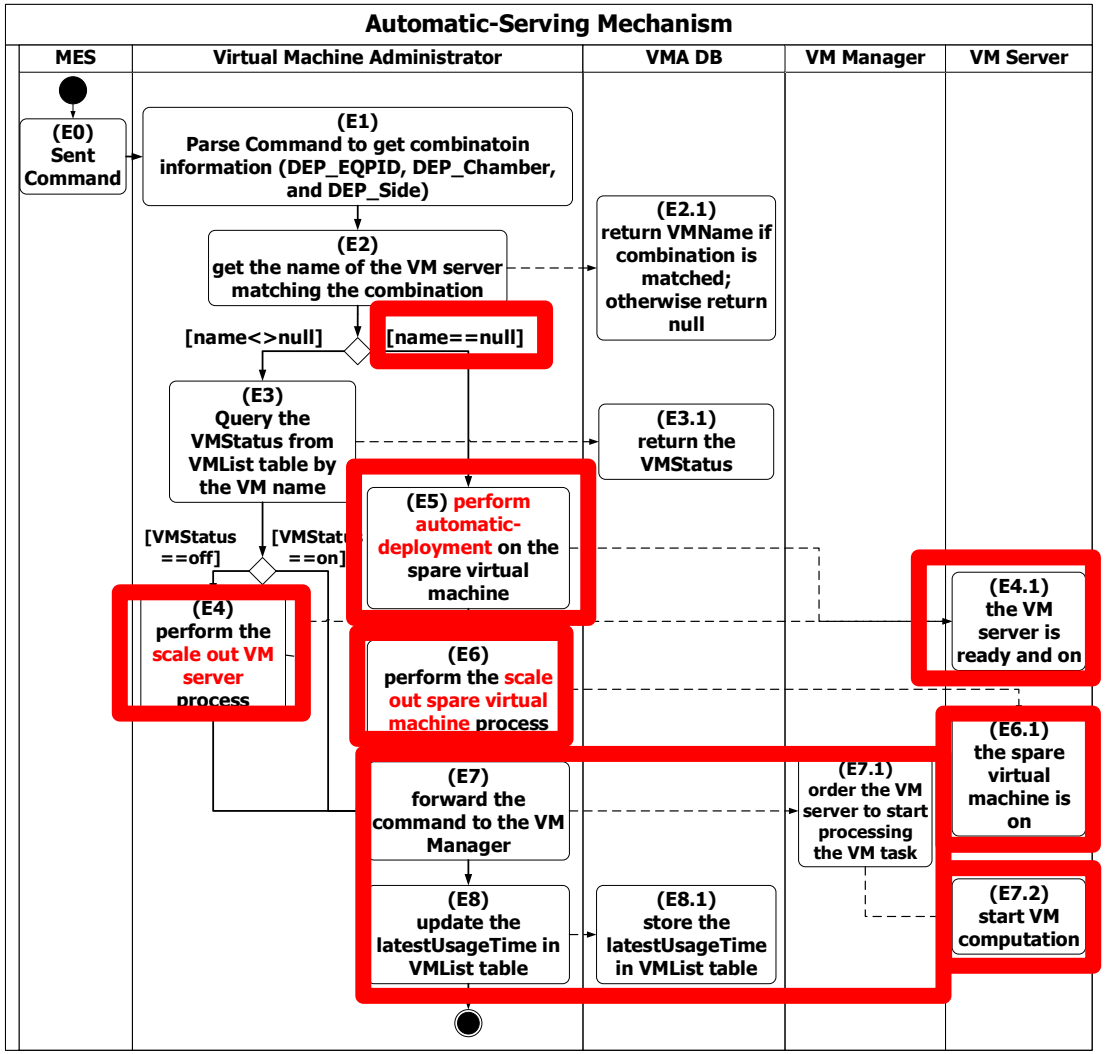


Design of Automatic-Deployment Mechanism(3/3)

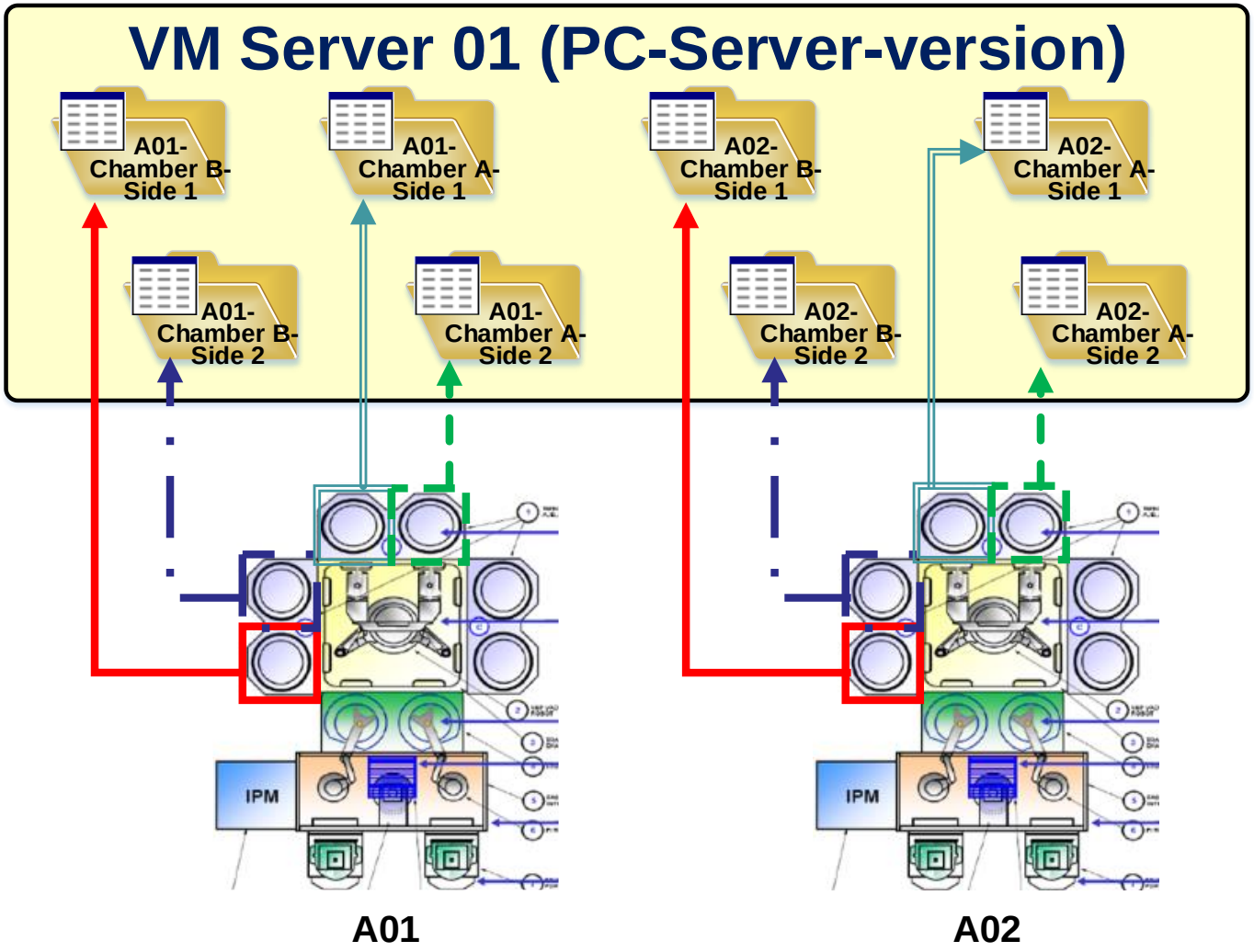


Design of Automatic-Serving Mechanism

- This mechanism is to automatically have a VM server to serve a requested VM task:



Testing environment of PC-Server-version AVM system (1/2)



Testing environment of PC-Server-version AVM system (2/2)

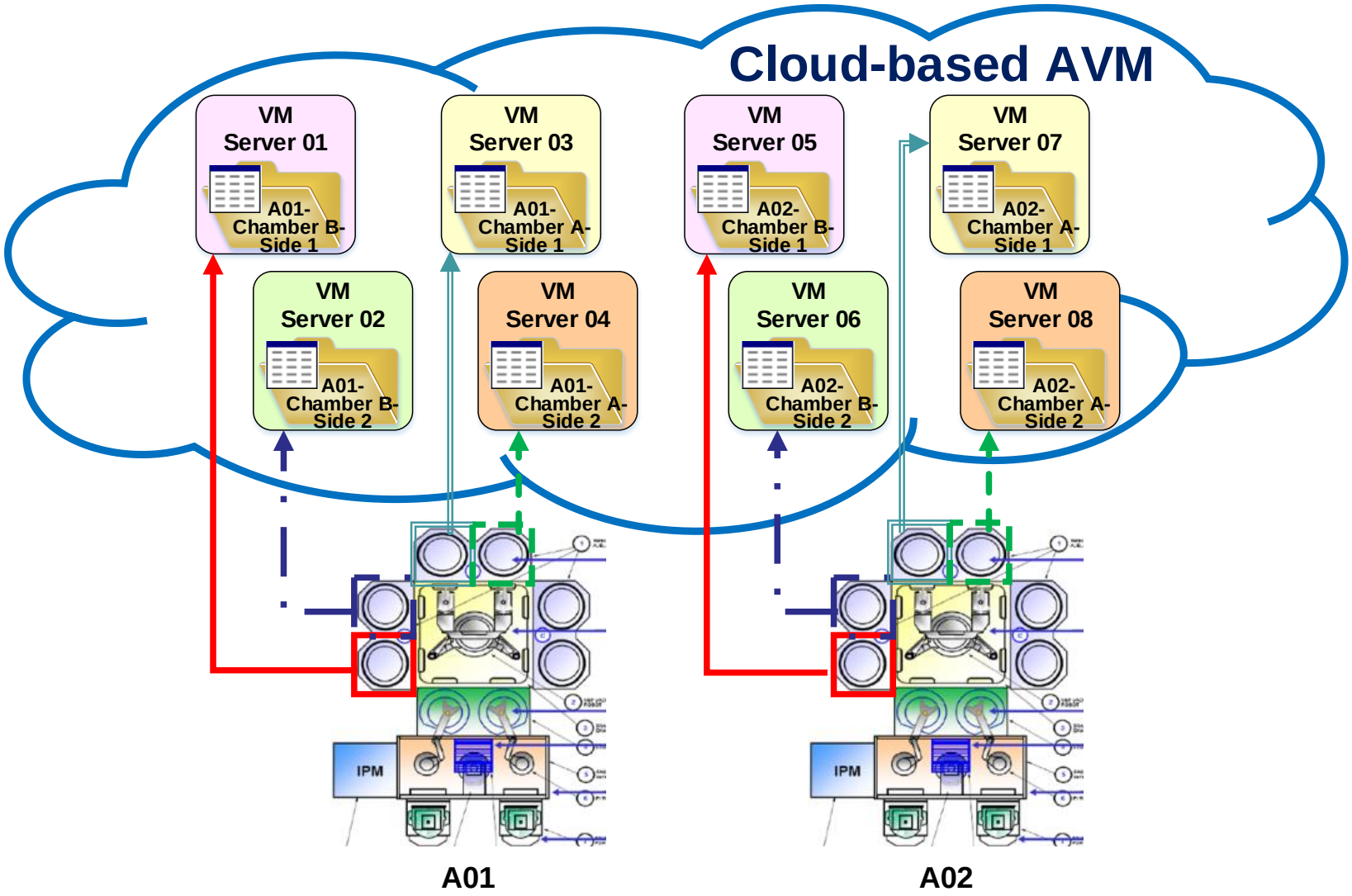
- Physical Server Specifications:

Computer	PC 1	PC 2
Software	VM Manager	VM Server
OS	Windows 7	Windows 7
CPU	Intel® Core™ i5-3450	Intel® Core™ i5-3450
Clock Speed	3.1 GHz	3.1 GHz
RAM	4 GB	4 GB

- Test Data Information:

DEP-EQPID	DEP-Chamber	DEP-Side	Time Period	Operation VM Server	Number of workpiece
A01	A	1	2012/10/15 09:30 ~ 2012/10/15 10:17	VM Server 01	10
A01	A	2	2012/10/15 09:00 ~ 2012/10/15 10:16		10
A01	B	1	2012/10/15 16:30 ~ 2012/10/15 16:40		10
A01	B	2	2012/10/15 09:00 ~ 2012/10/15 11:50		10
A02	A	1	2012/10/15 13:00 ~ 2012/10/15 16:20		10
A02	A	2	2012/10/15 09:00 ~ 2012/10/15 13:18		10
A02	B	1	2012/10/15 14:00 ~ 2012/10/15 17:20		10
A02	B	2	2012/10/15 09:00 ~ 2012/10/15 13:20		10
					Total 80

Testing environment of the cloud-based AVM system (1/2)



Testing environment of the cloud-based AVM system (2/2)

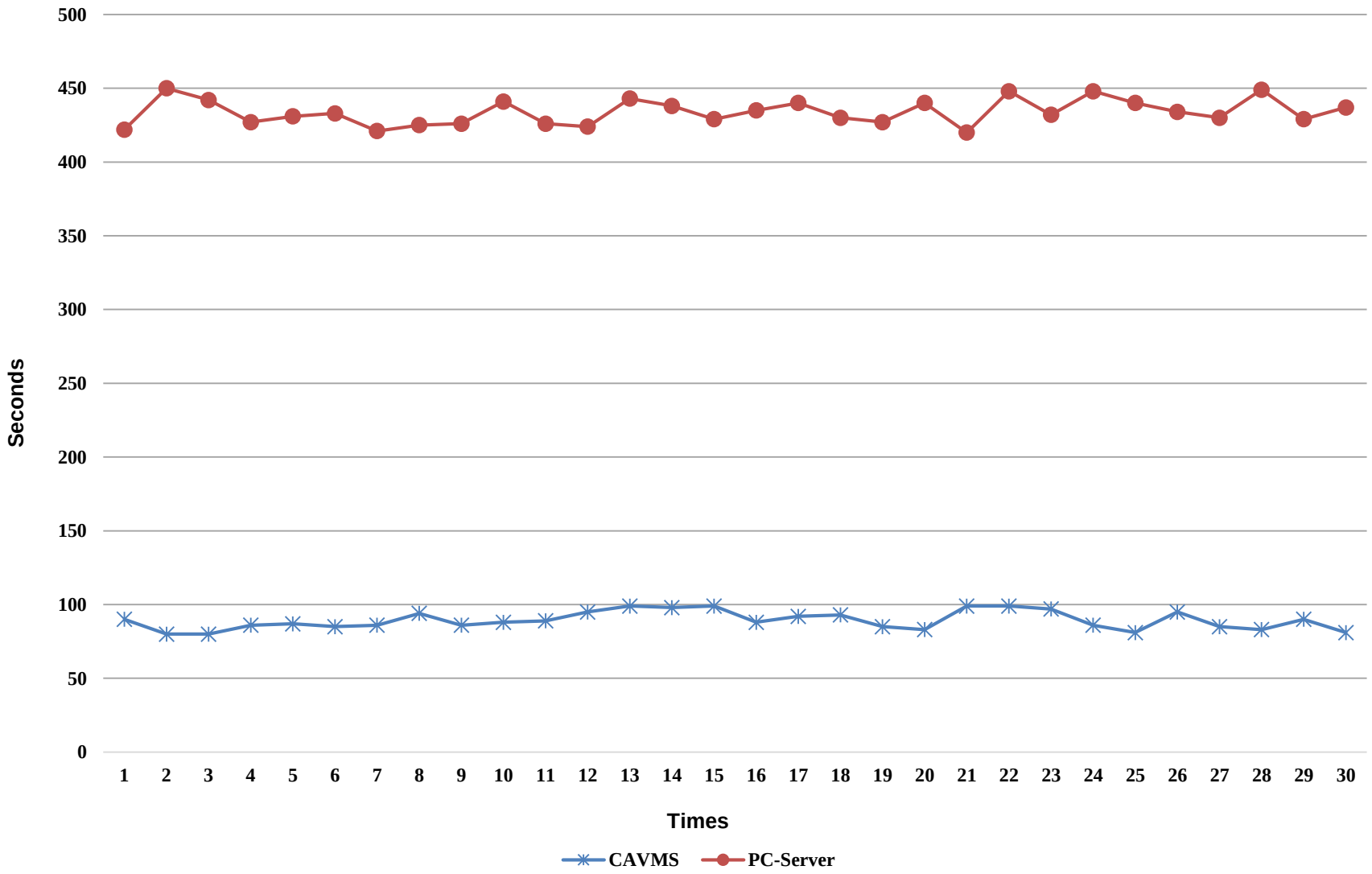
- Virtual Machine Specifications :

Computer	Virtual Machine
OS	Windows 7
CPU	Intel® Xeon® E5-2625
Clock Speed	2.0 GHz
RAM	4 GB

- Test Data Information :

Test Data Information					
DEP-EQPID	DEP-Chamber	DEP-Side	Time Period	Operation VM Server	Number of workpiece
A01	A	1	2012/10/15 09:30 ~ 2012/10/15 10:17	VM Server 01	10
A01	A	2	2012/10/15 09:00 ~ 2012/10/15 10:16	VM Server 02	10
A01	B	1	2012/10/15 16:30 ~ 2012/10/15 16:40	VM Server 03	10
A01	B	2	2012/10/15 09:00 ~ 2012/10/15 11:50	VM Server 04	10
A02	A	1	2012/10/15 13:00 ~ 2012/10/15 16:20	VM Server 05	10
A02	A	2	2012/10/15 09:00 ~ 2012/10/15 13:18	VM Server 06	10
A02	B	1	2012/10/15 14:00 ~ 2012/10/15 17:20	VM Server 07	10
A02	B	2	2012/10/15 09:00 ~ 2012/10/15 13:20	VM Server 08	10
					Total 80

Comparison of the execution time of predicting the production quality of 80 wafers



Comparison of Prediction Accuracy

- VM results of the PC-Server-based AVM system:

NN	MAPE (%)	Standard Deviation
Phase I	**0.3954**	**15.0091**
Phase II	**0.2894**	**14.6837**

- VM results of the cloud-based AVM system:

NN	MAPE (%)	Standard Deviation
Phase I	**0.3954**	**15.0091**
Phase II	**0.2894**	**14.6837**

References

1. Joseph Schmuller, Sams Teach Yourself UML in 24 Hours, Third Edition, Sams Publishing, March 2004.
2. Bennett, McRobb and Farmer, Object Oriented Systems Analysis and Design Using UML, (2nd Edition), McGraw Hill, 2002.
3. Alan Dennis, Barbara Wixom, and David Tegarden, Systems Analysis and Design With UML 2.0-An Object-Oriented Approach, Second Edition John Wiley & Sons, Inc., 2005.
4. Min-Hsiung Hung, Fan-Tien Cheng, and Sze-Chien Yeh, "Development of a Web-Services-Based e-Diagnostics Framework for Semiconductor Manufacturing Industry," *IEEE Transactions on Semiconductor Manufacturing*, Vol. 17, No. 5, pp. 122-135, February 2005.
5. e-Diagnostics Guidelines and Guidebook, ISMI Web Site: URL: http://ismi.sematech.org/emanufacturing/ediagguide.htm/