Introduction to
Object-Oriented Analysis/Design/Programming

Prof. Lin Weiguo

Oct.2013

Static Modeling using the Unified Modeling Language (UML)

Material based on
[Booch99, Rambaugh99, Jacobson99, Fowler97, Brown99]

Objects

- The purpose of class modeling is to describe objects.
- An object is a concept, abstraction or thing that has meaning for a domain/application.
- Some objects have real world counterparts while others are conceptual entities.
- The choice of objects depends on judgment and the nature of problem.
- All objects have identity and are distinguishable.

Classes

- An **object** is an **instance** – occurrence – **of a class**
- A class describes a group of objects with the same properties (attributes), behavior (operations), kinds of relationships, and semantics.
- The objects in a class share a common semantic purpose, above and beyond the requirement of common attributes and behavior.
- By grouping objects onto classes we abstract a problem.

UML representation of classes/objects:

- UML: Unified Modeling Language (OMG Standard): O.O Visual Modeling language

- Class/object representation

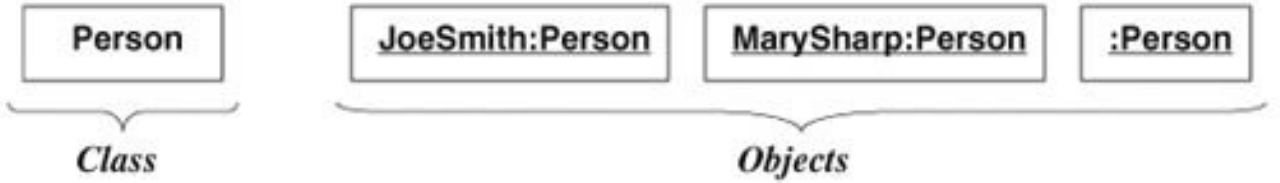
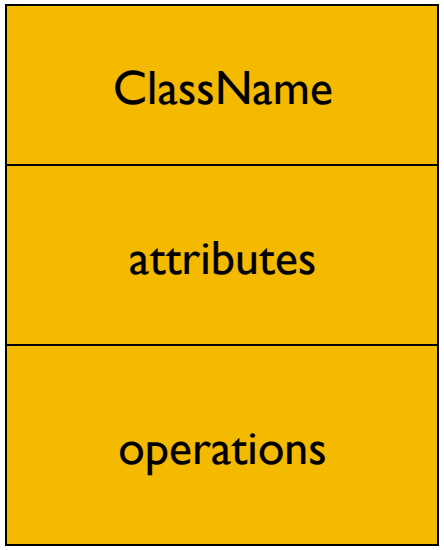


Figure 3.1 A class and objects. Objects and classes are the focus of class modeling.

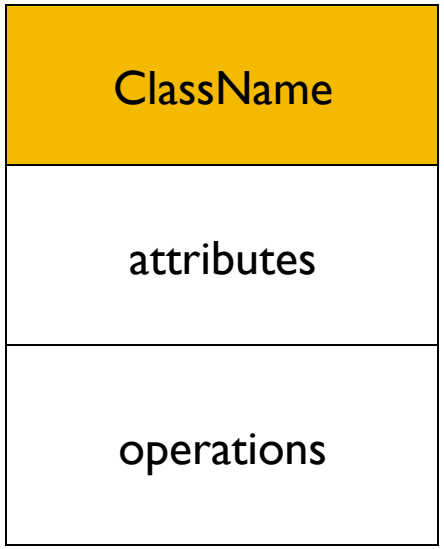
Classes



A *class* is a description of a set of objects that share the same attributes, operations, relationships, and semantics.

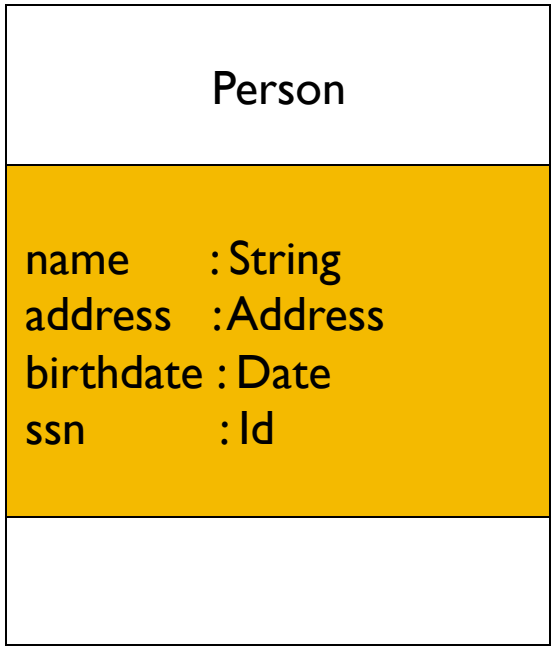
Graphically, a class is rendered as a rectangle, usually including its name, attributes, and operations in separate, designated compartments.

Class Names



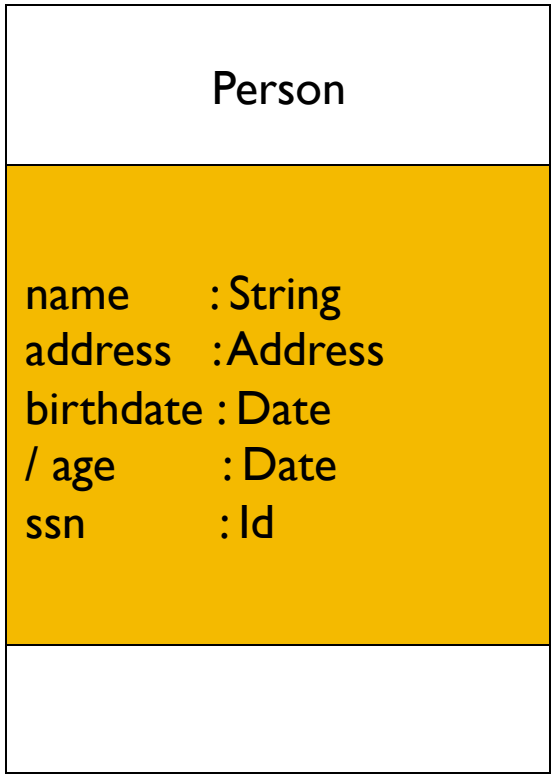
The name of the class is the only required tag in the graphical representation of a class. It always appears in the top-most compartment.

Class Attributes



An *attribute* is a named property of a class that describes the object being modeled. In the class diagram, attributes appear in the second compartment just below the name-compartment.

Class Attributes (Cont'd)



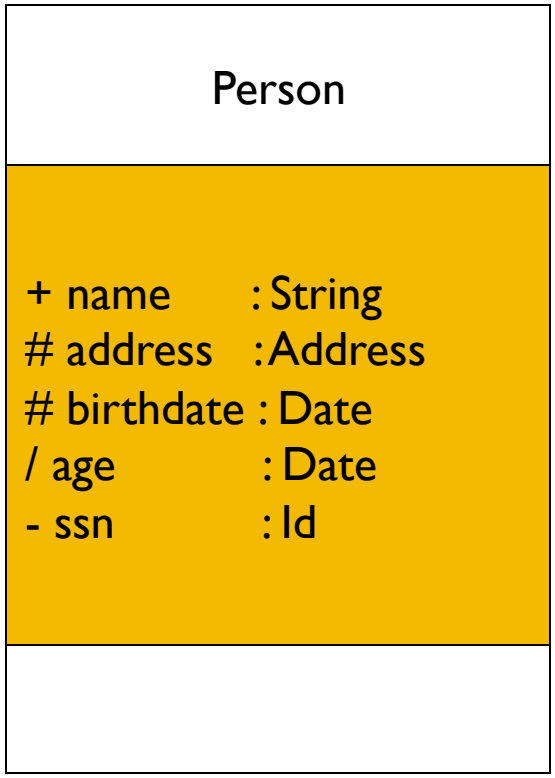
Attributes are usually listed in the form:

attributeName : Type

A *derived* attribute is one that can be computed from other attributes, but doesn't actually exist. For example, a Person's age can be computed from his birth date. A derived attribute is designated by a preceding '/' as in:

/ age : Date

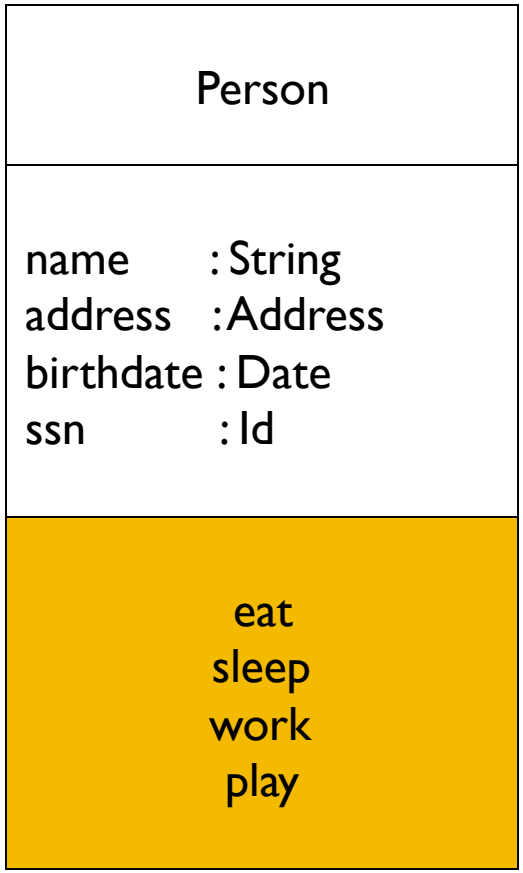
Class Attributes (Cont'd)



Attributes can be:

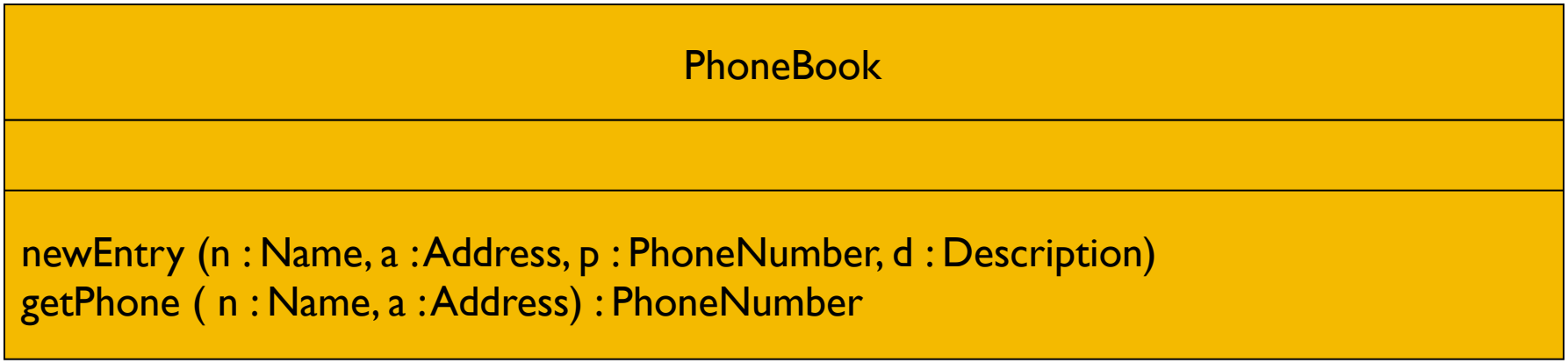
+ public
# protected
- private
/ derived

Class Operations



Operations describe the class behavior and appear in the third compartment.

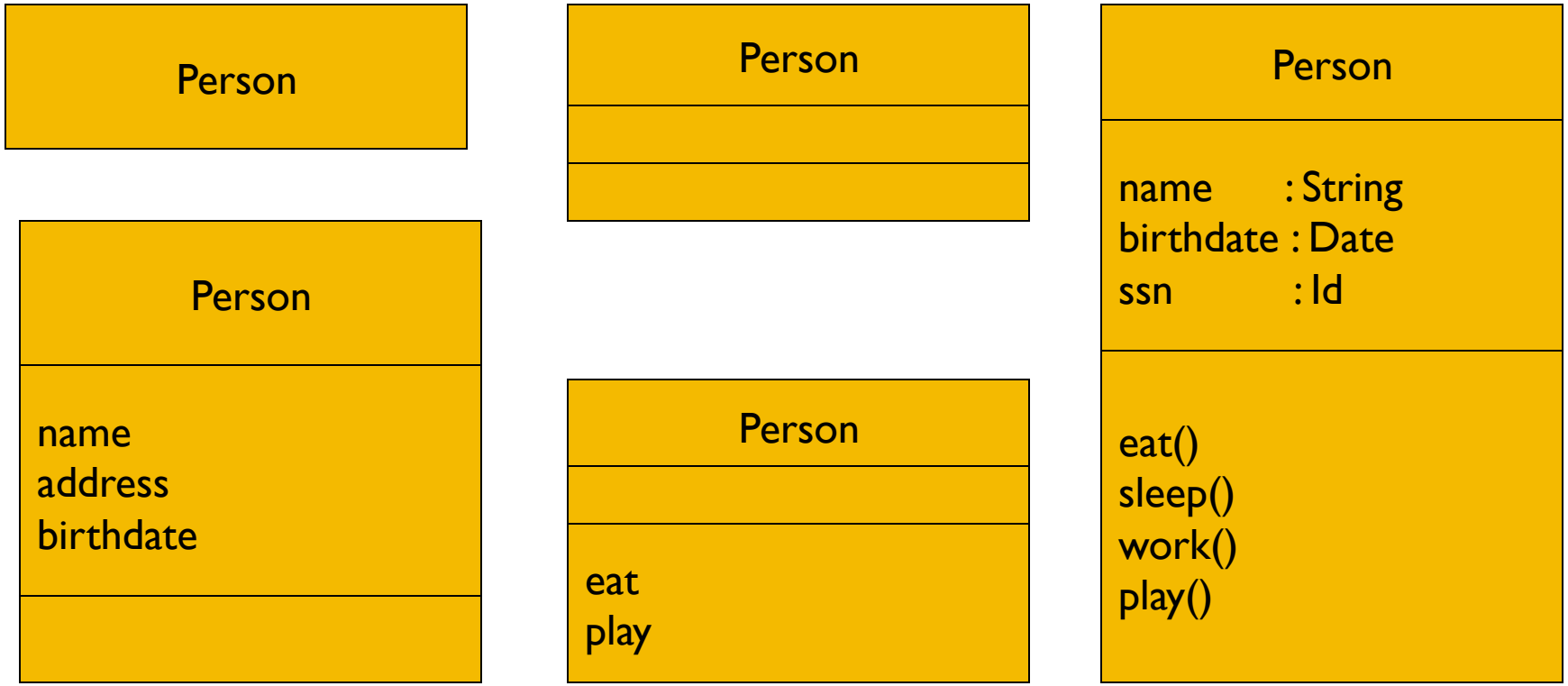
Class Operations (Cont'd)



You can specify an operation by stating its signature: listing the name, type, and default value of all parameters, and, in the case of functions, a return type.

Depicting Classes

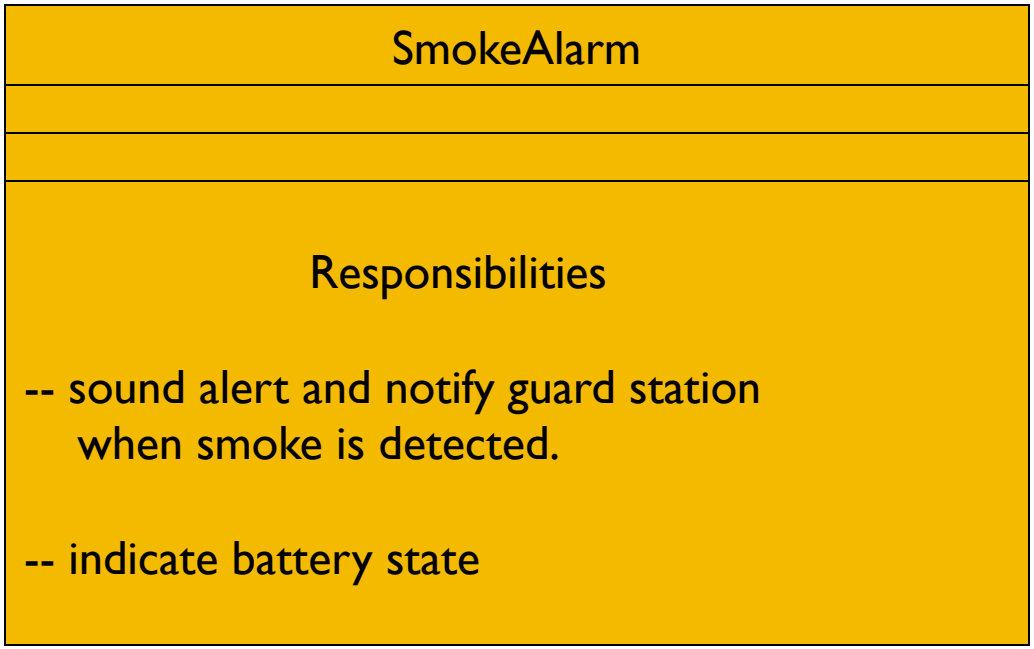
When drawing a class, you needn't show attributes and operation in every diagram.



Class Responsibilities

A class may also include its responsibilities in a class diagram.

A responsibility is a contract or obligation of a class to perform a particular service.



Relationships

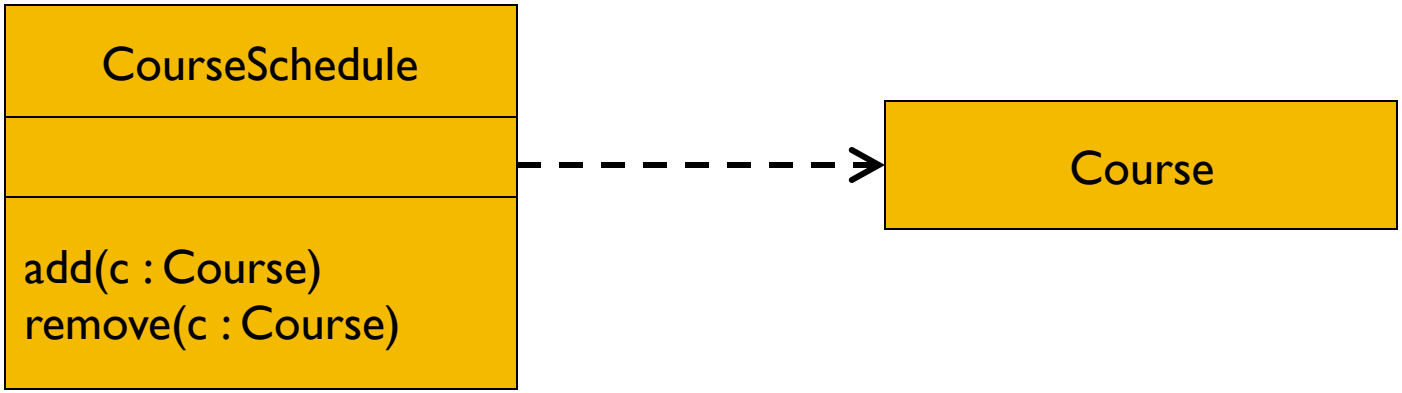
In UML, object interconnections (logical or physical), are modeled as relationships.

There are three kinds of relationships in UML:

- Dependencies
- Generalizations
- Associations

Dependency Relationships

A *dependency* indicates a semantic relationship between two or more elements. The dependency from *CourseSchedule* to *Course* exists because *Course* is used in both the **add** and **remove** operations of *CourseSchedule*.



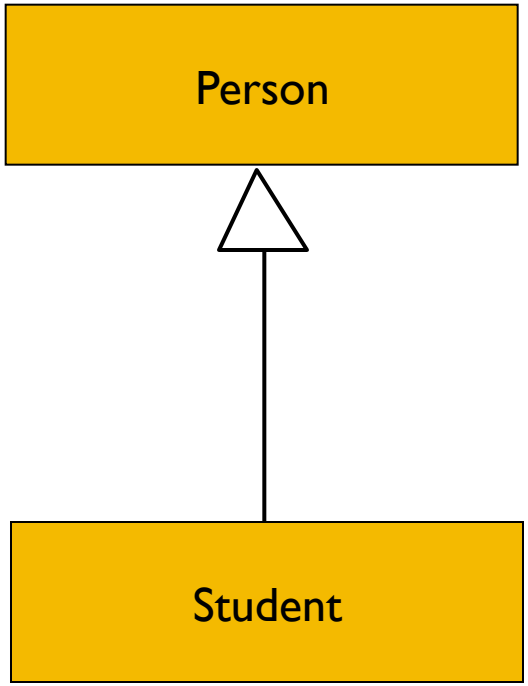
依赖(Dependency)

【依赖关系】：是一种使用的关系，即一个类的实现需要另一个类的协助，所以要尽量不使用双向的互相依赖.

【代码表现】：局部变量、方法的参数或者对静态方法的调用

【箭头及指向】：带箭头的虚线，指向被使用者

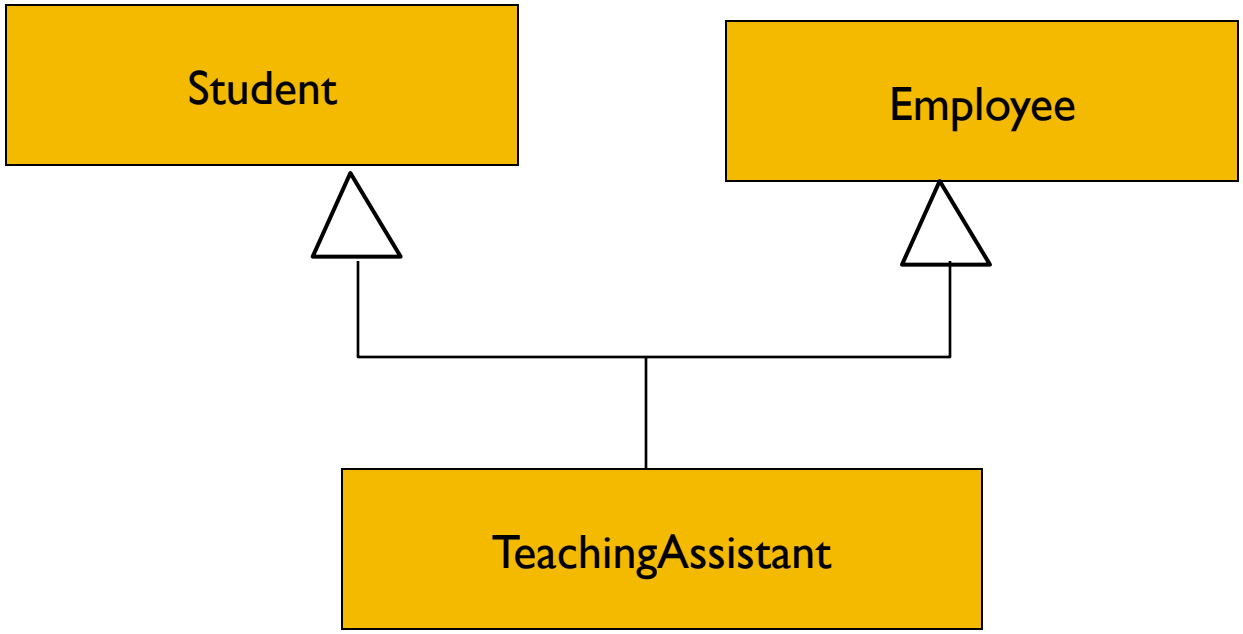
Generalization Relationships



A *generalization* connects a subclass to its superclass. It denotes an inheritance of attributes and behavior from the superclass to the subclass and indicates a specialization in the subclass of the more general superclass.

Generalization Relationships (Cont'd)

UML permits a class to inherit from multiple superclasses, although some programming languages (e.g., Java) do not permit multiple inheritance.



泛化（Generalization）

- 【泛化关系】：是一种继承关系，表示一般与特殊的关系，它指定了子类如何特化父类的所有特征和行为. 例如：老虎是动物的一种，即有老虎的特性也有动物的共性.
- 【箭头指向】：带三角箭头的实线，箭头指向父类

Association Relationships

If two classes in a model need to communicate with each other, there must be link between them.

An *association* denotes that link.

Association Relationships (Cont'd)

We can indicate the *multiplicity* of an association by adding *multiplicity adornments* to the line denoting the association.

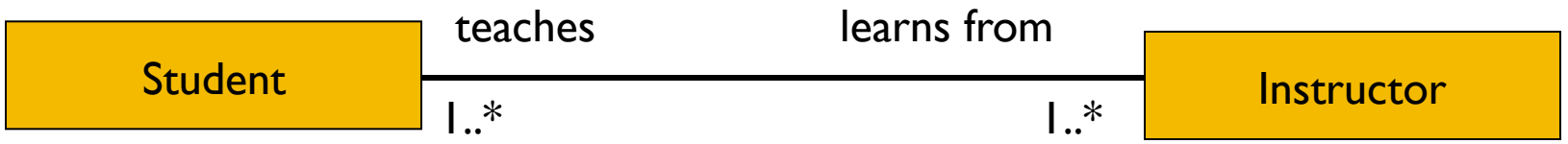
The example indicates that a *Student* has one or more *Instructors*:

Association Relationships (Cont'd)

The example indicates that every *Instructor* has one or more *Students*:

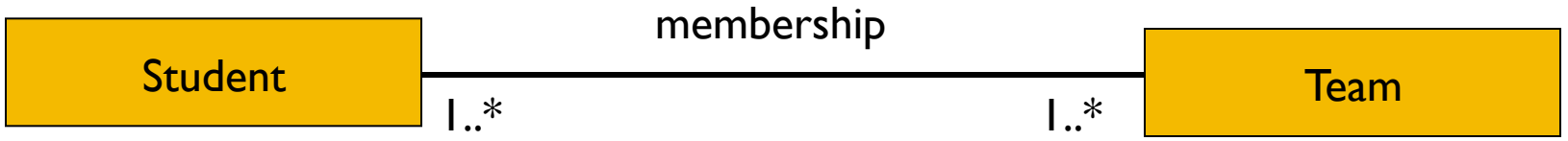
Association Relationships (Cont'd)

We can also indicate the behavior of an object in an association (*i.e.*, the *role* of an object) using *rolenames*.



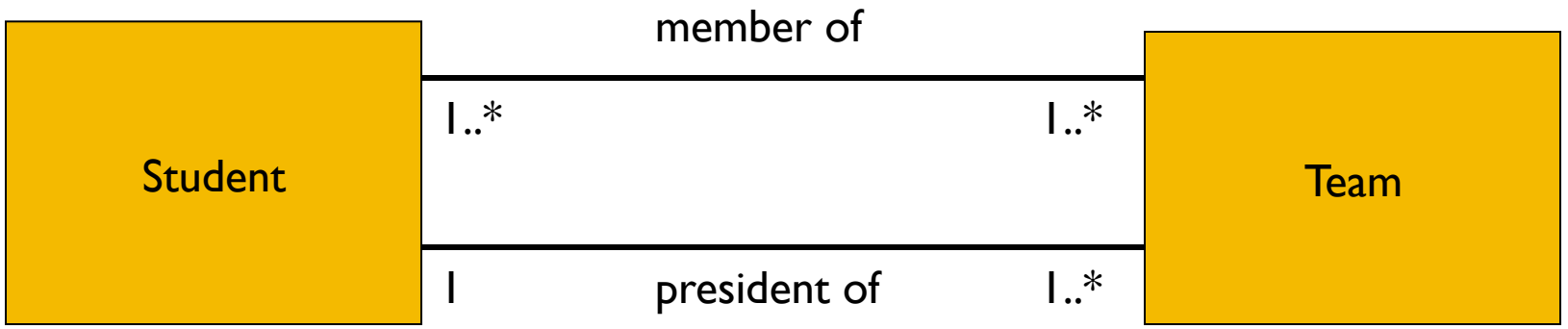
Association Relationships (Cont'd)

We can also name the association.



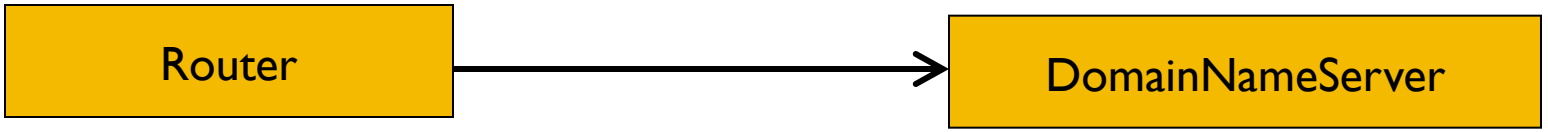
Association Relationships (Cont'd)

We can specify dual associations.



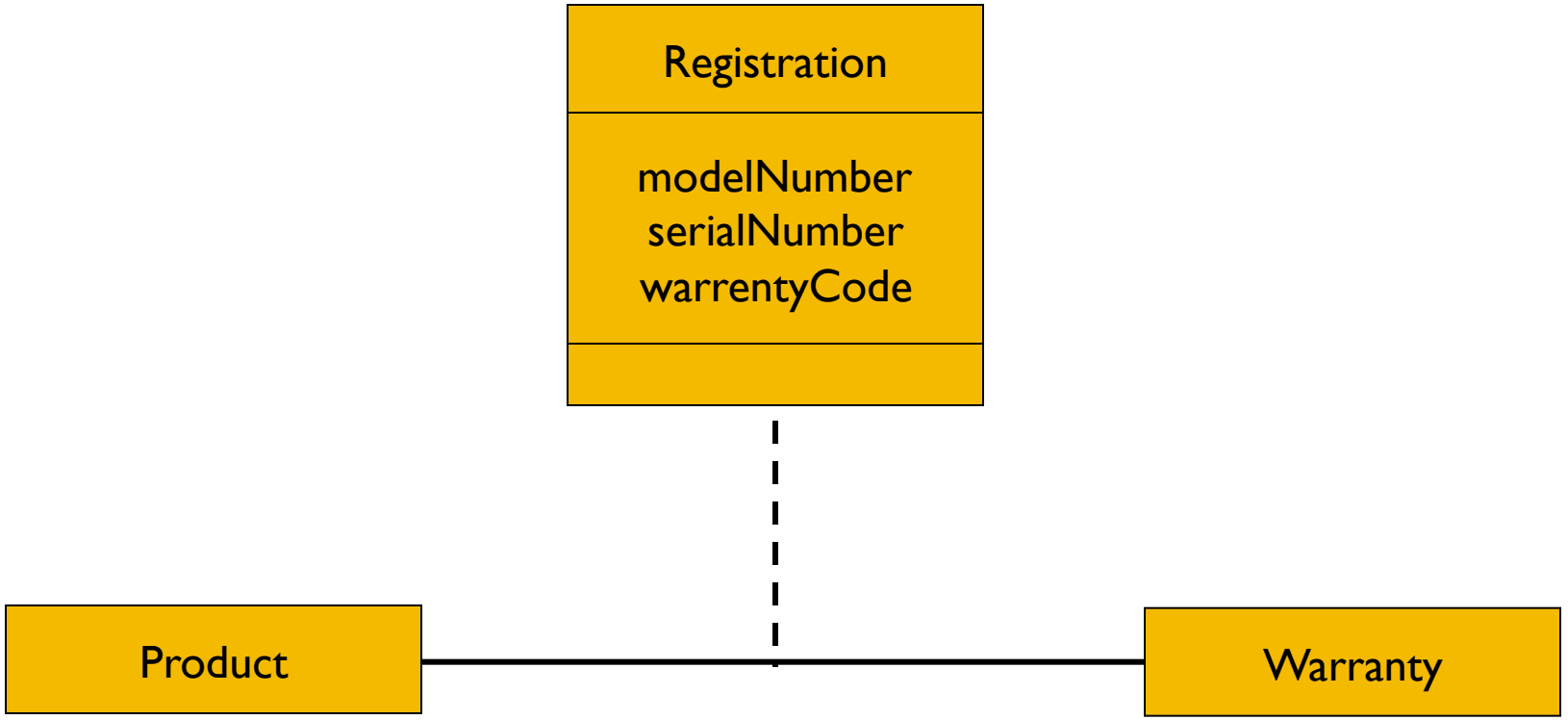
Association Relationships (Cont'd)

We can constrain the association relationship by defining the *navigability* of the association. Here, a *Router* object requests services from a *DNS* object by sending messages to (invoking the operations of) the server. The direction of the association indicates that the server has no knowledge of the *Router*.



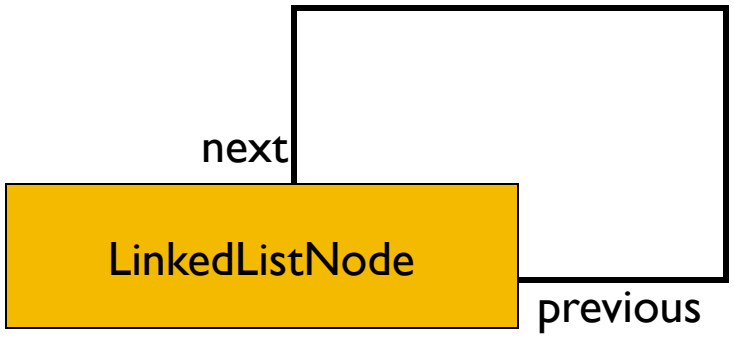
Association Relationships (Cont'd)

Associations can also be objects themselves, called *link classes* or an *association classes*.



Association Relationships (Cont'd)

A class can have a *self association*.



关联（Association)

【关联关系】：是一种拥有的关系，它使一个类知道另一个类的属性和方法；如：老师与学生，丈夫与妻子

关联可以是双向的，也可以是单向的。双向的关联可以有两个箭头或者没有箭头，单向的关联有一个箭头。

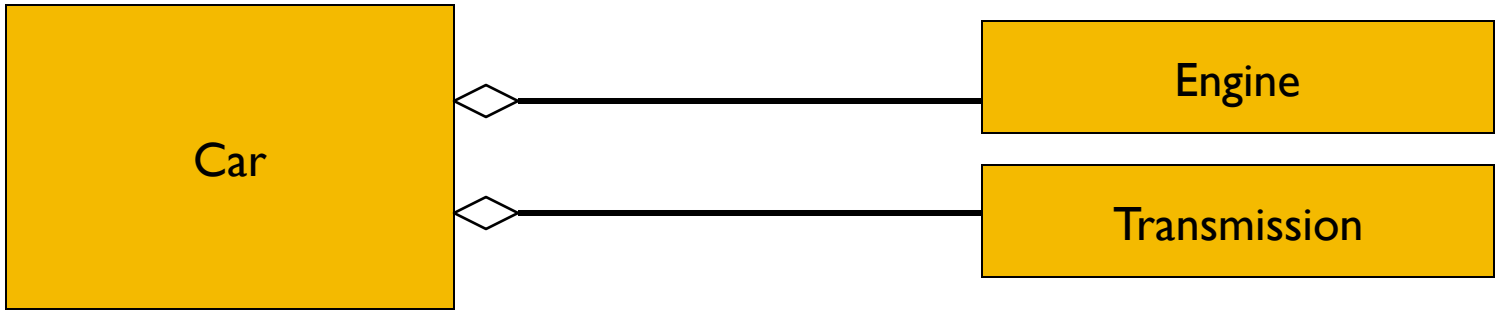
【代码体现】：成员变量

【箭头及指向】：带普通箭头的实心线，指向被拥有者

Association Relationships (Cont'd)

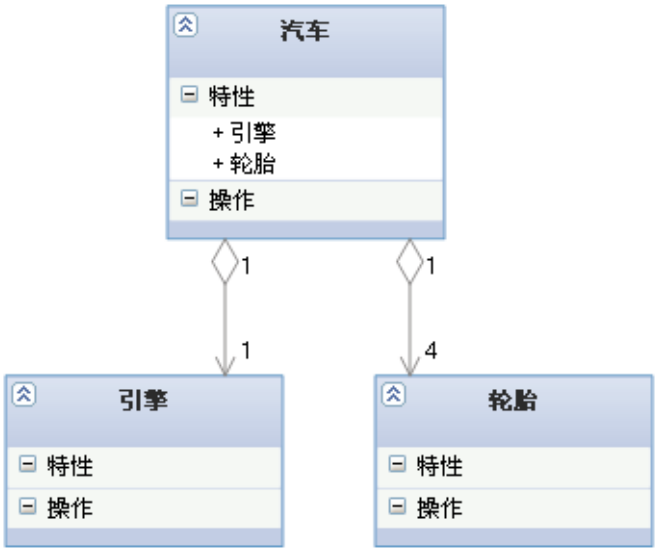
We can model objects that contain other objects by way of special associations called *aggregations* and *compositions*.

An *aggregation* specifies a whole-part relationship between an aggregate (a whole) and a constituent part, where the part can exist independently from the aggregate. Aggregations are denoted by a hollow-diamond adornment on the association.



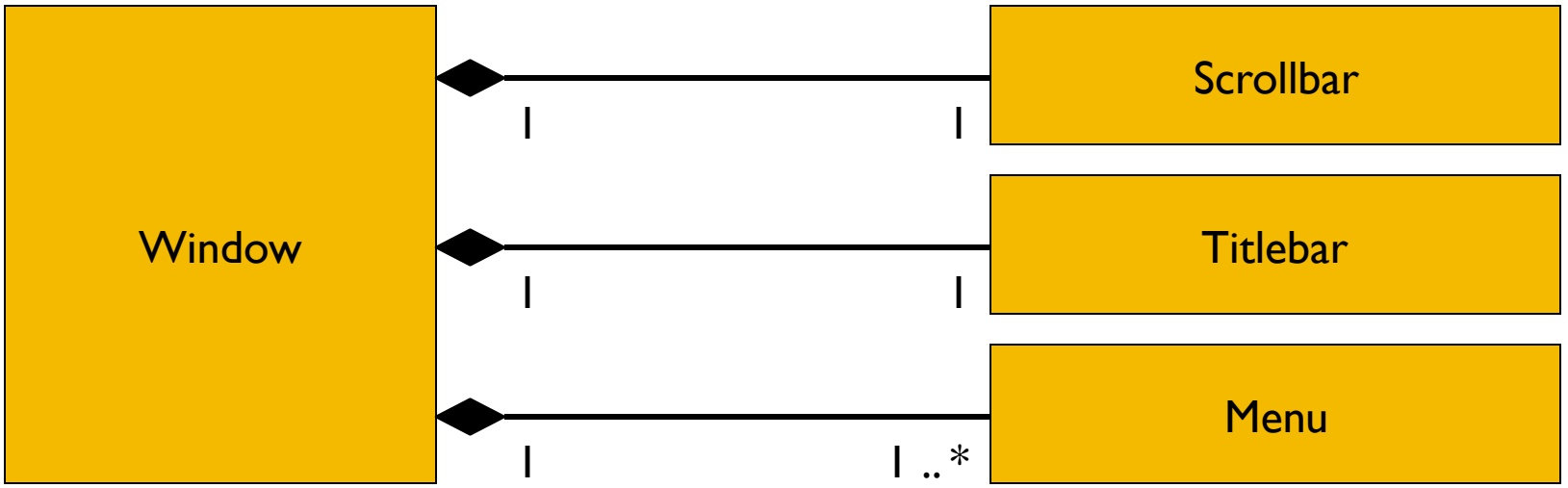
聚合（Aggregation）

【聚合关系】：是整体与部分的关系，且部分可以离开整体而单独存在.
如车和轮胎是整体和部分的关系，轮胎离开车仍然可以存在.
聚合关系是关联关系的一种，是强的关联关系；关联和聚合在语法上无法区分，必须考察具体的逻辑关系。
【代码体现】：成员变量
【箭头及指向】：带空心菱形的实心线，菱形指向整体



Association Relationships (Cont'd)

A *composition* indicates a strong ownership and coincident lifetime of parts by the whole (*i*.e., they live and die as a whole). Compositions are denoted by a filled-diamond adornment on the association.



组合(Composition)

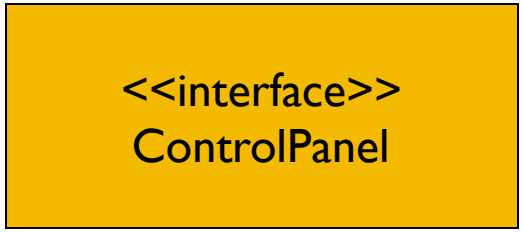
【组合关系】：是整体与部分的关系，但部分不能离开整体而单独存在．如公司和部门是整体和部分的关系，没有公司就不存在部门．

组合关系是关联关系的一种，是比聚合关系还要强的关系，它要求普通的聚合关系中代表整体的对象负责代表部分的对象的生命周期

【代码体现】：成员变量

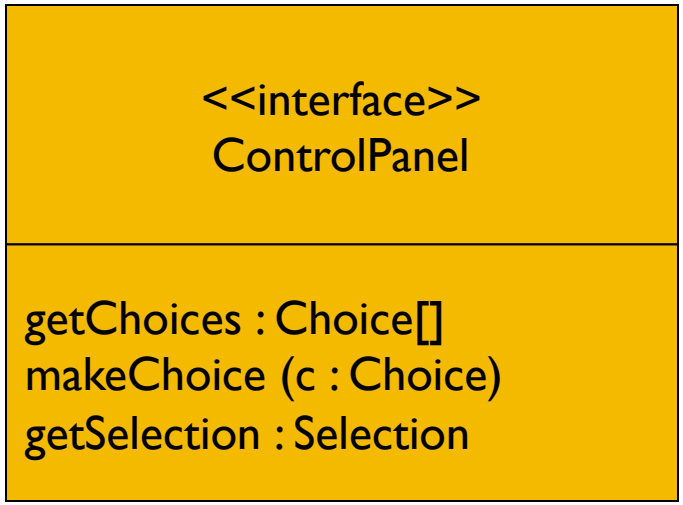
【箭头及指向】：带实心菱形的实线，菱形指向整体

Interfaces



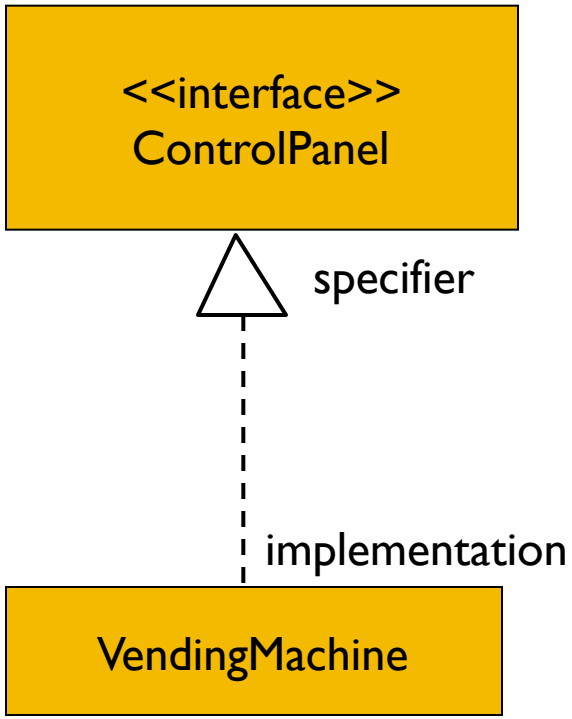
An *interface* is a named set of operations that specifies the behavior of objects without showing their inner structure. It can be rendered in the model by a one- or two-compartment rectangle, with the *stereotype* <<interface>> above the interface name.

Interface Services



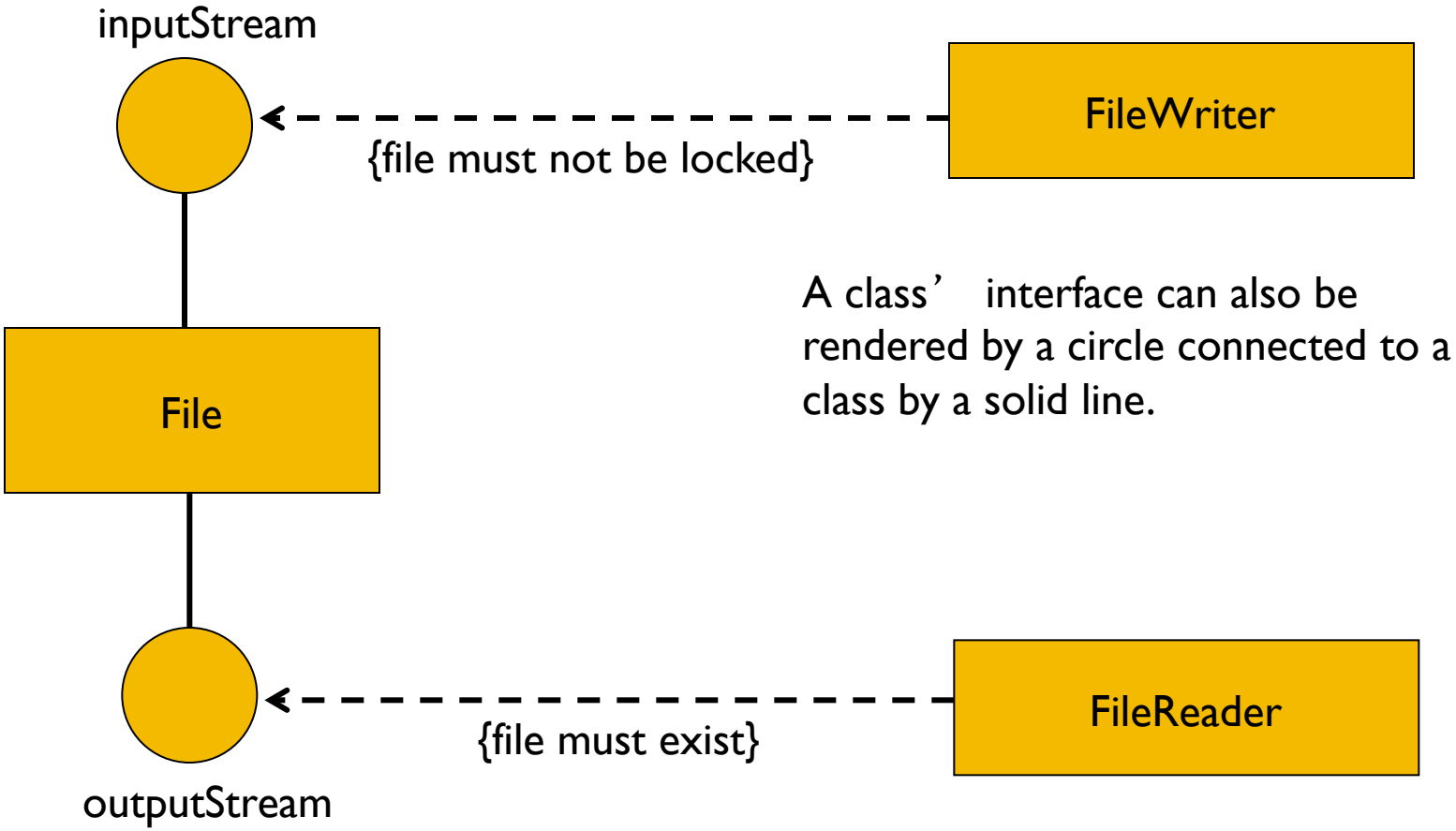
Interfaces do not get instantiated. They have no attributes or state. Rather, they specify the services offered by a related class.

Interface Realization Relationship



A *realization* relationship connects a class with an interface that supplies its behavioral specification. It is rendered by a dashed line with a hollow triangle towards the specifier.

Interfaces

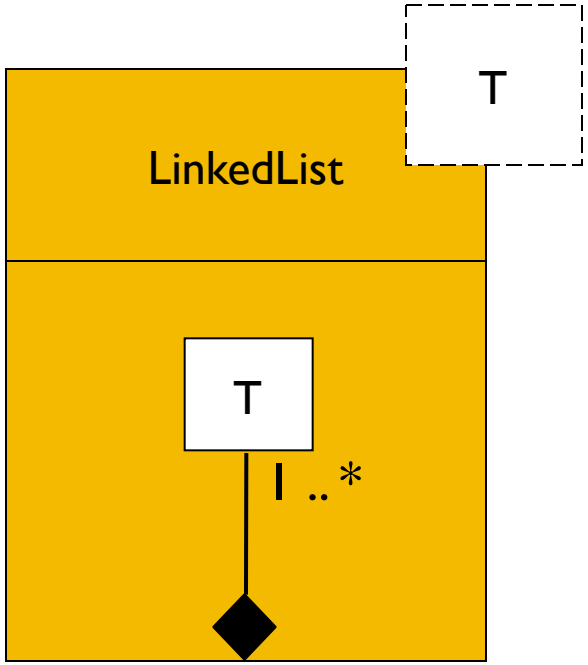

inputStream
FileWriter
{file must not be locked}
A class' interface can also be rendered by a circle connected to a class by a solid line.
File
FileReader
{file must exist}
outputStream

实现（Realization）

【实现关系】：是一种类与接口的关系，表示类是接口所有特征和行为的实现.

【箭头指向】：带三角箭头的虚线，箭头指向接口

Parameterized Class

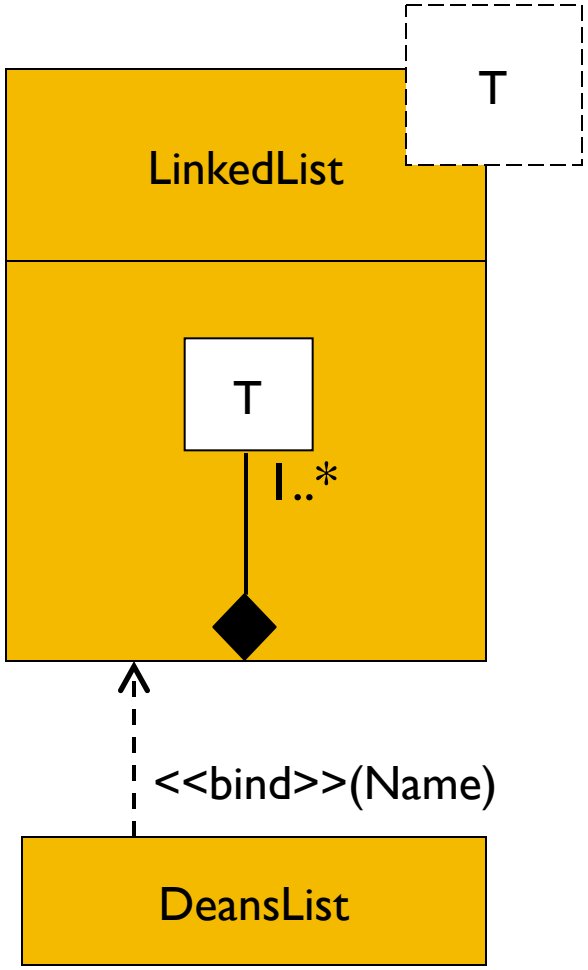


A *parameterized class* or *template* defines a family of potential elements.

To use it, the parameter must be bound.

A *template* is rendered by a small dashed rectangle superimposed on the upper-right corner of the class rectangle. The dashed rectangle contains a list of formal parameters for the class.

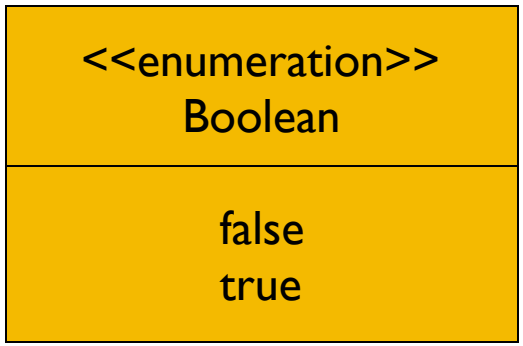
Parameterized Class (Cont'd)



Binding is done with the <<bind>> stereotype and a parameter to supply to the template. These are adornments to the dashed arrow denoting the realization relationship.

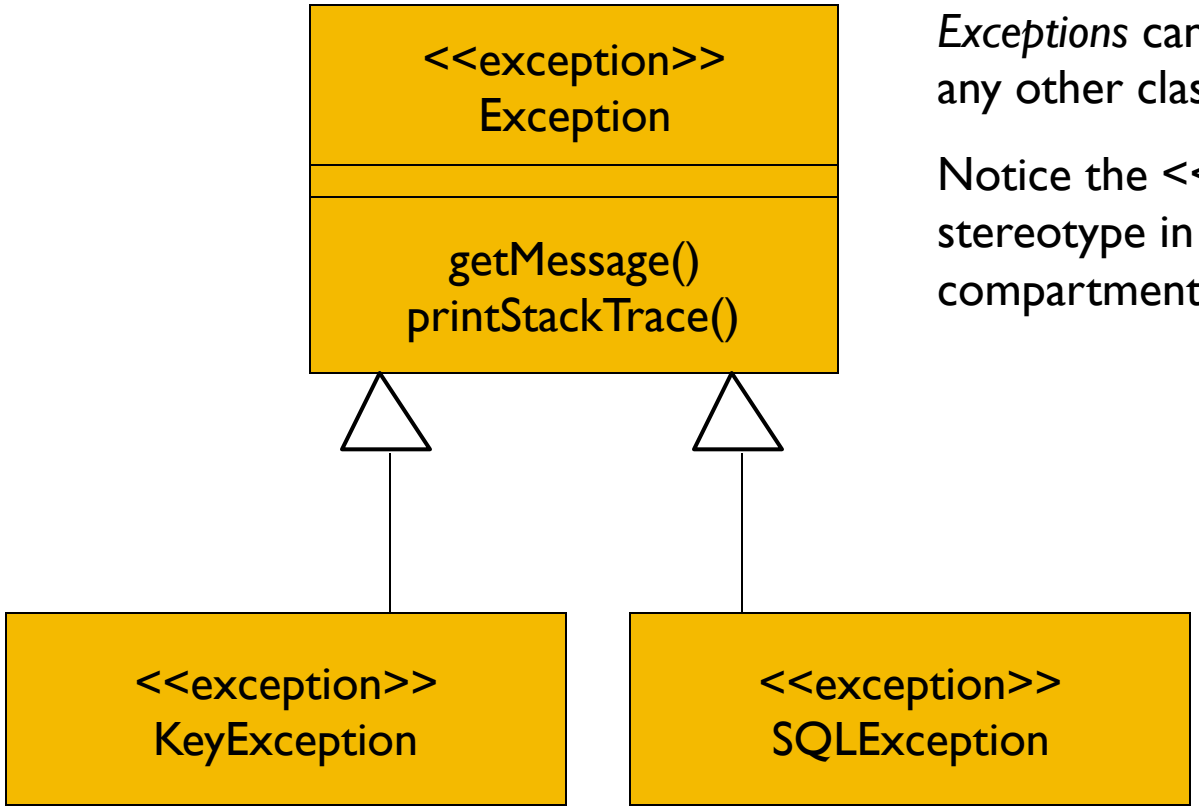
Here we create a linked-list of names for the Dean's List.

Enumeration



An *enumeration* is a user-defined data type that consists of a name and an ordered list of enumeration literals.

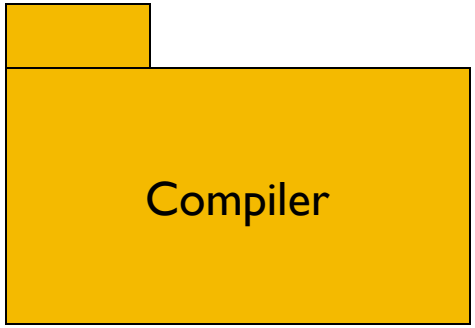
Exceptions



Exceptions can be modeled just like any other class.

Notice the <<exception>> stereotype in the name compartment.

Packages



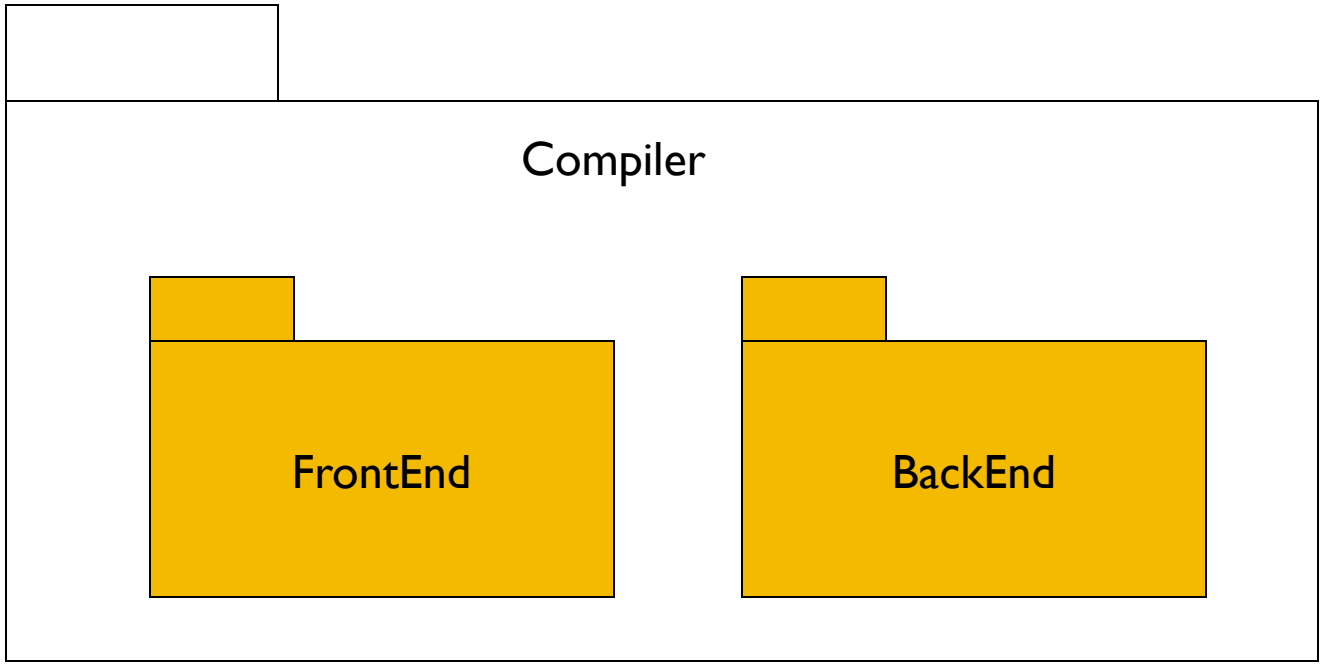
A *package* is a container-like element for organizing other elements into groups.

A package can contain classes and other packages and diagrams.

Packages can be used to provide controlled access between classes in different packages.

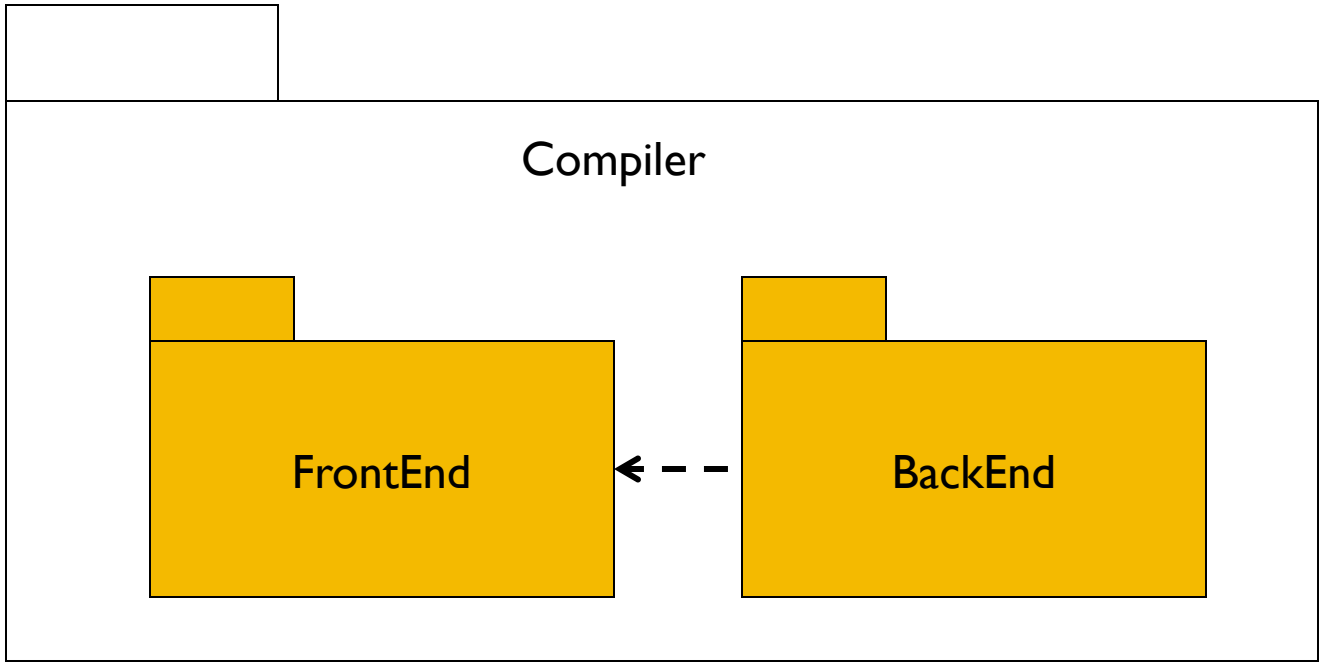
Packages (Cont'd)

Classes in the *FrontEnd* package and classes in the *BackEnd* package cannot access each other in this diagram.

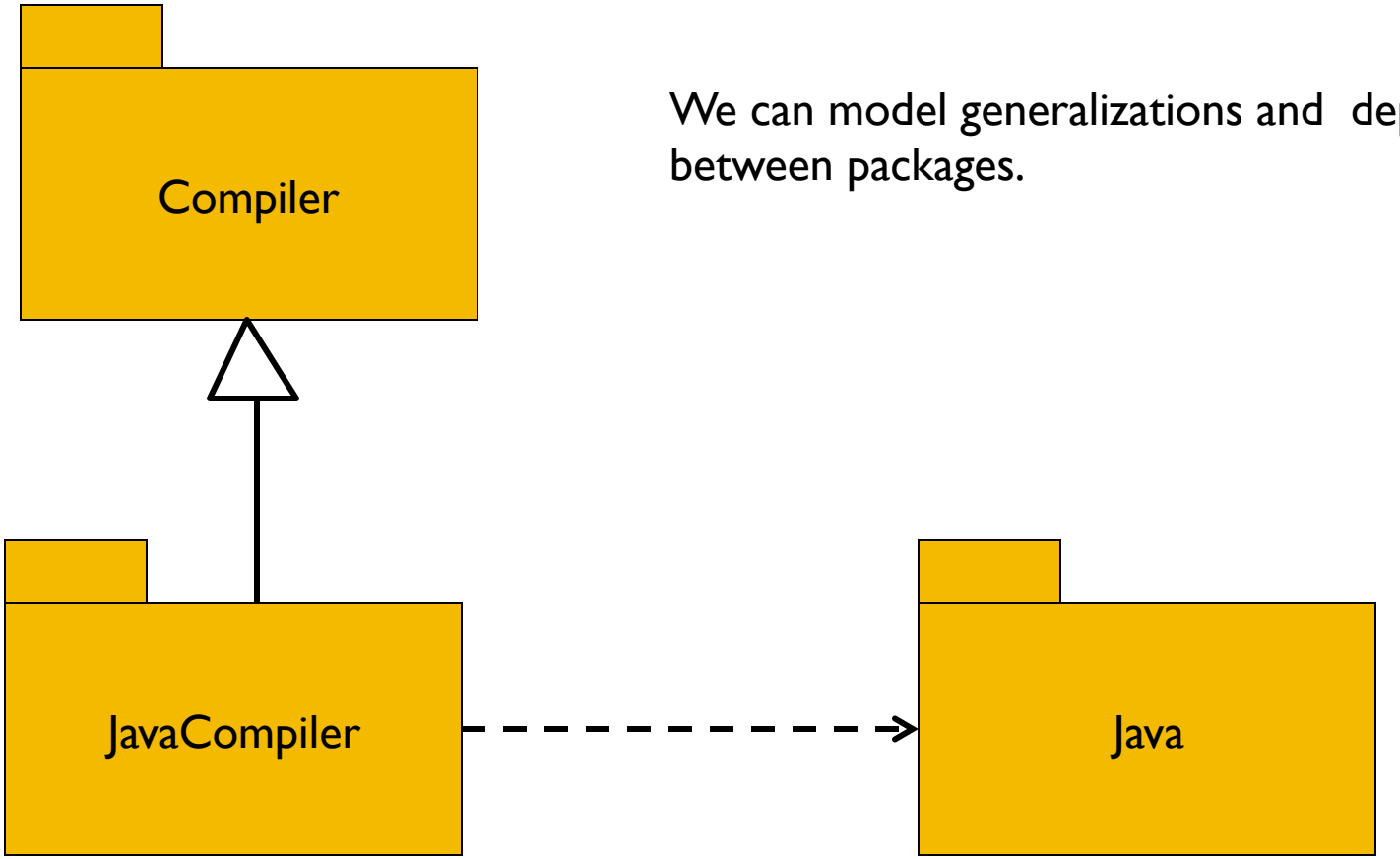


Packages (Cont'd)

Classes in the *BackEnd* package now have access to the classes in the *FrontEnd* package.



Packages (Cont'd)



We can model generalizations and dependencies between packages.

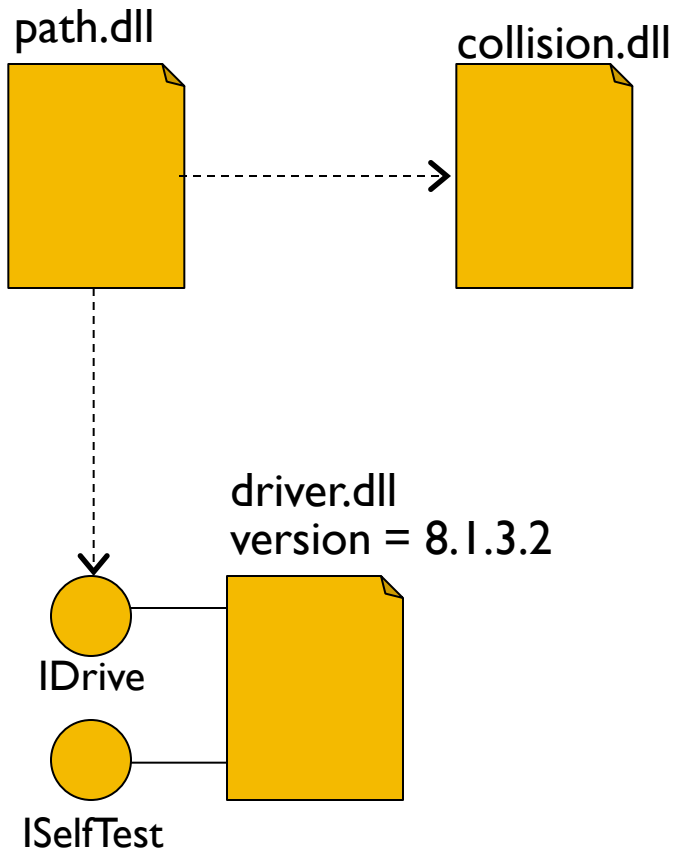
Component Diagram

Component diagrams are one of the two kinds of diagrams found in modeling the physical aspects of an object-oriented system. They show the organization and dependencies between a set of components.

Use component diagrams to model the ***static implementation view*** of a system. This involves modeling the physical things that reside on a node, such as executables, libraries, tables, files, and documents.

- The UML User Guide, Booch et. al., 1999

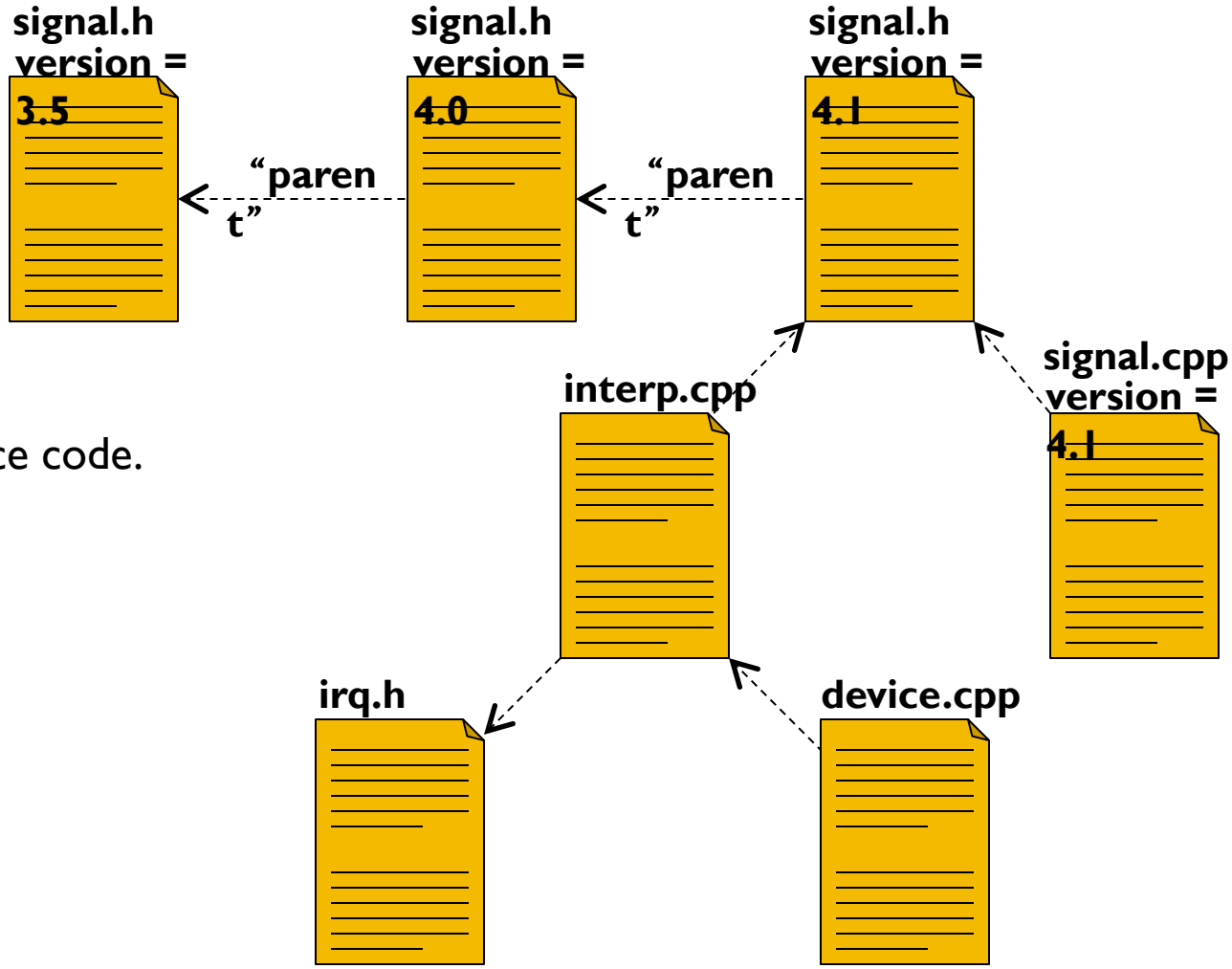
Component Diagram



Here’s an example of a component model of an executable release.

[Booch,99]

Component Diagram



Modeling source code.

[Booch, 99]

Deployment Diagram

Deployment diagrams are one of the two kinds of diagrams found in modeling the physical aspects of an object-oriented system. They show the configuration of ***run-time processing*** nodes and the components that live on them.

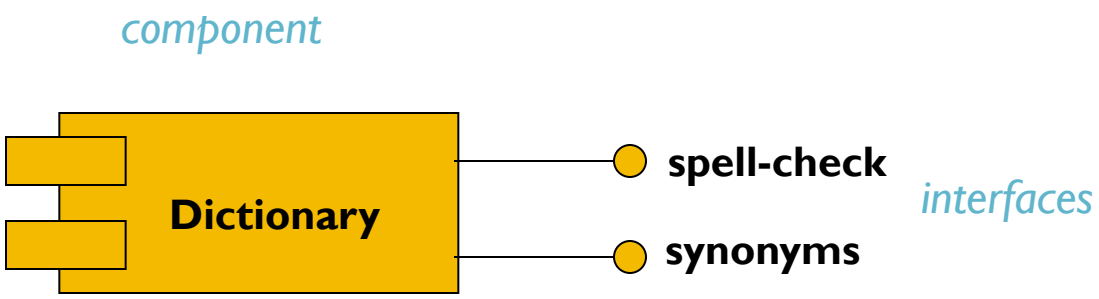
Use deployment diagrams to model the ***static deployment view*** of a system. This involves modeling the topology of the hardware on which the system executes.

- *The UML User Guide, [Booch,99]*

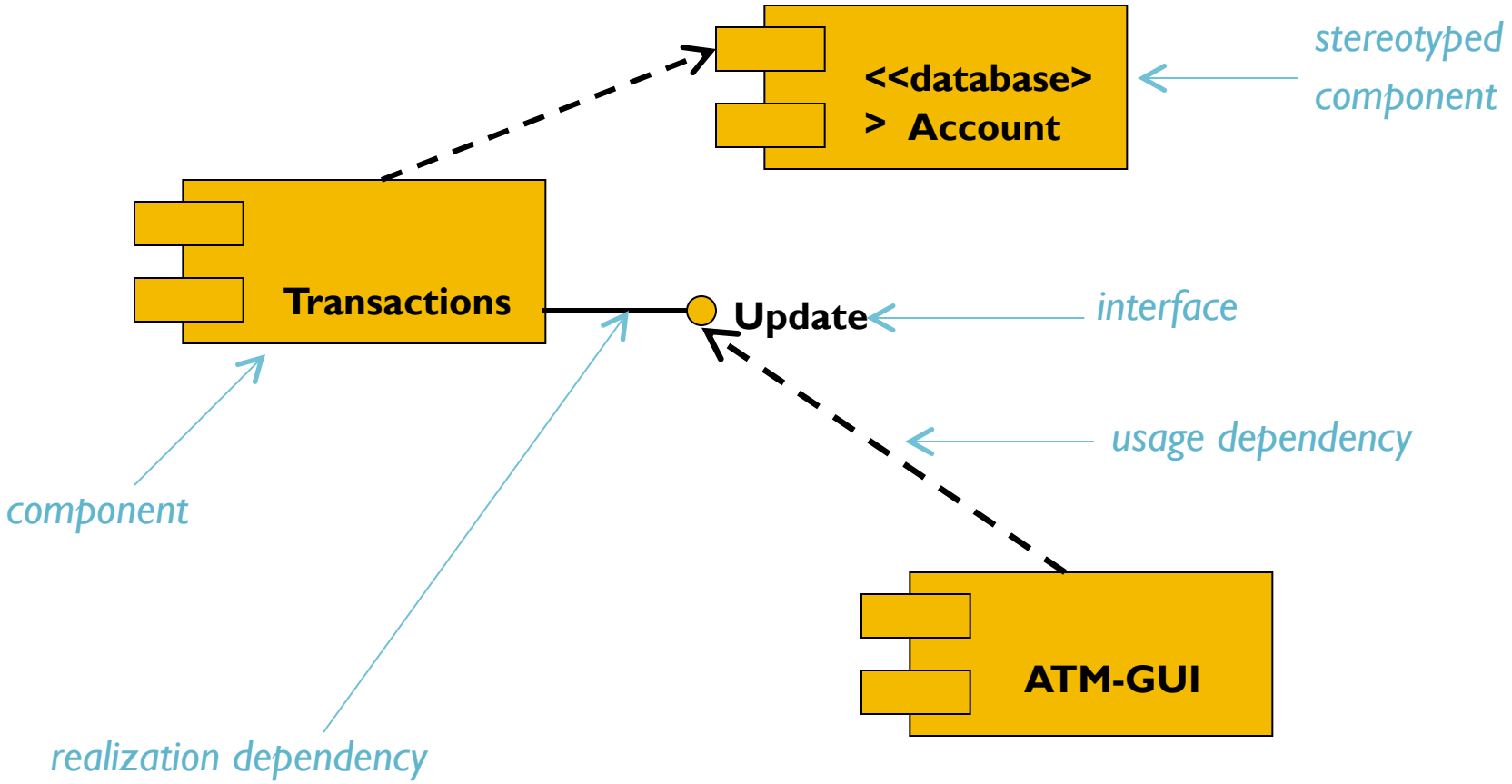
Deployment Diagram

A component is a physical unit of implementation with well-defined interfaces that is intended to be used as a replaceable part of a system. Well designed components do not depend directly on other components, but rather on interfaces that components support.

- *The UML Reference Manual, [Rumbaugh,99]*

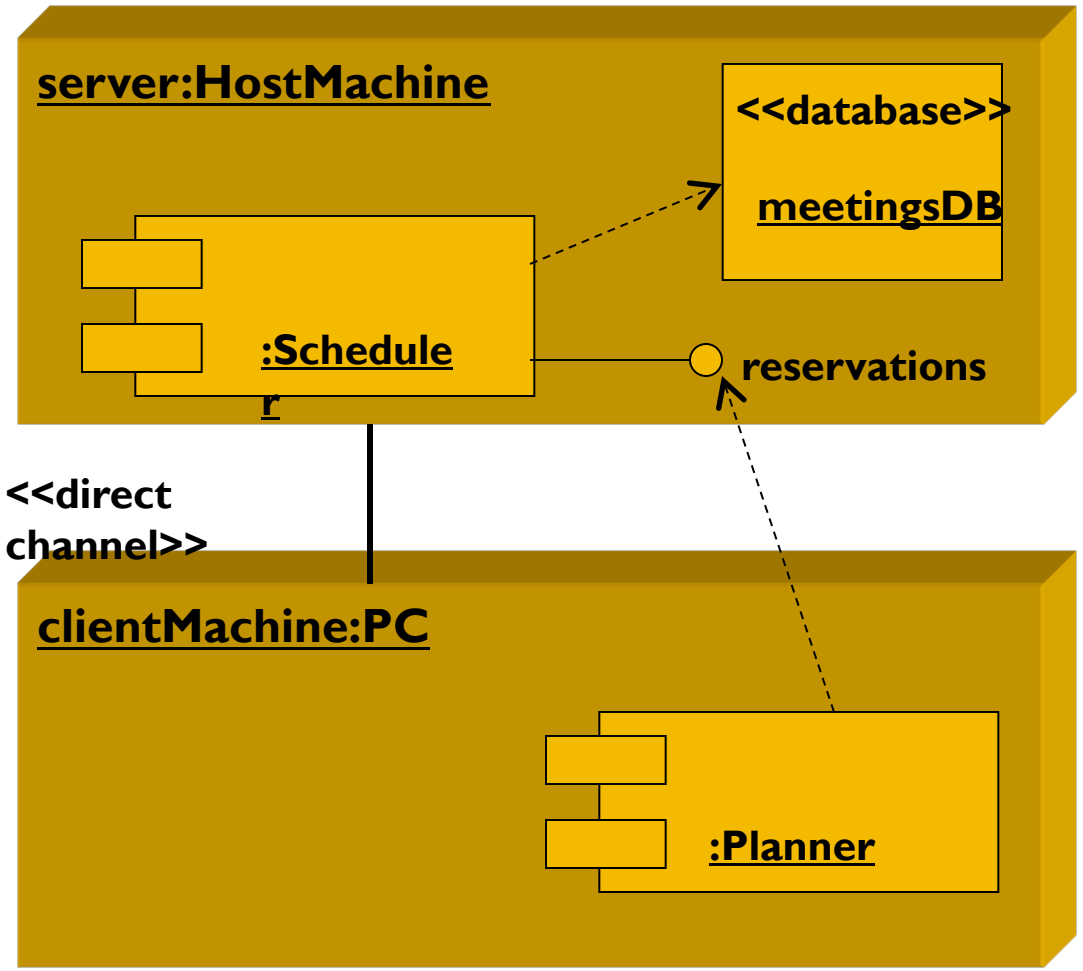


Deployment Diagram



[Rumbaugh,99]

Deployment Diagram



Deployment diagram of a client-server system.

[Rumbaugh,99]

Dynamic Modeling using the Unified Modeling Language (UML)

Use Case

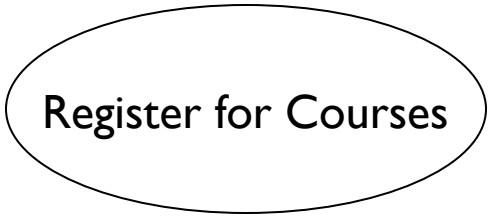
"A *use case* specifies the behavior of a system or a part of a system, and is a description of a set of sequences of actions, including variants, that a system performs to yield an observable result of value to an actor."

- *The UML User Guide, [Booch,99]*

"An *actor* is an idealization of an external person, process, or thing interacting with a system, subsystem, or class. An actor characterizes the interactions that outside users may have with the system."

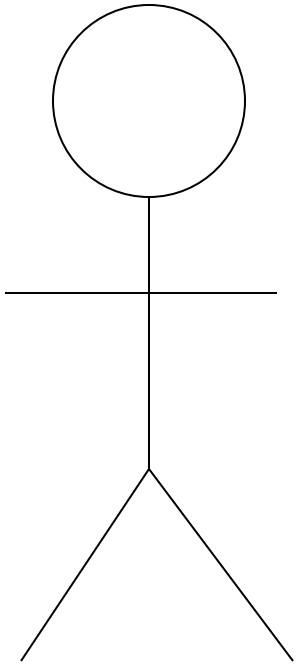
- *The UML Reference Manual, [Rumbaugh,99]*

Use Case (Cont'd)



A use case is rendered as an ellipse in a use case diagram. A use case is always labeled with its name.

Use Case (Cont'd)

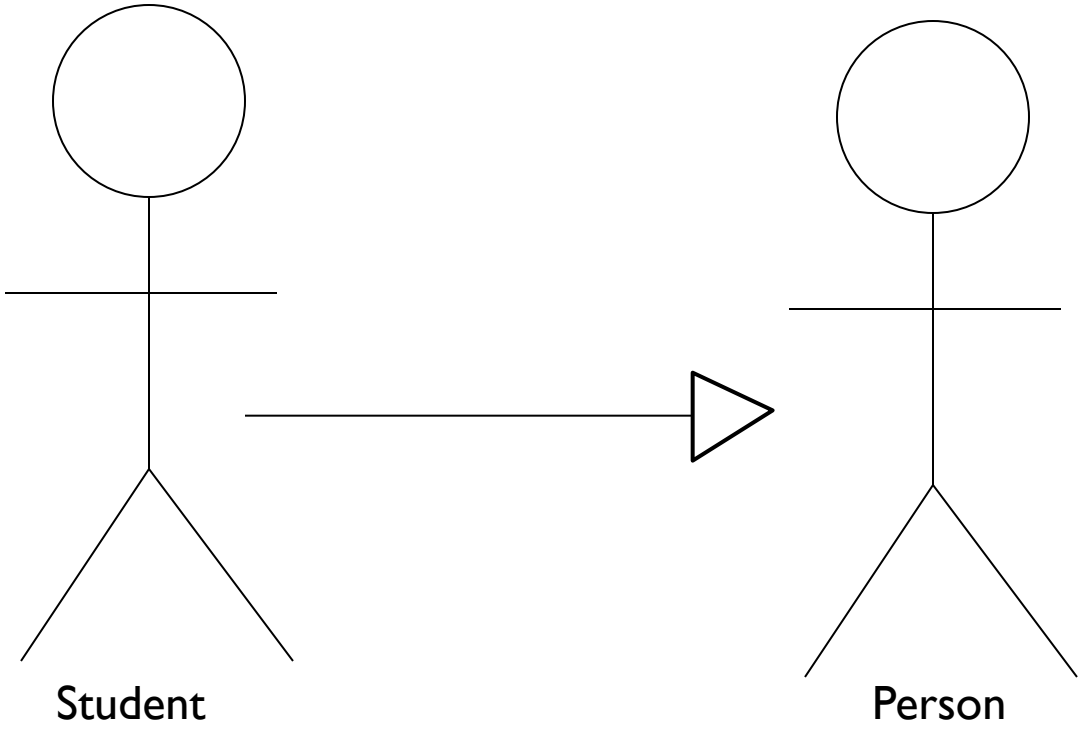


Student

An actor is rendered as a stick figure in a use case diagram. Each actor participates in one or more use cases.

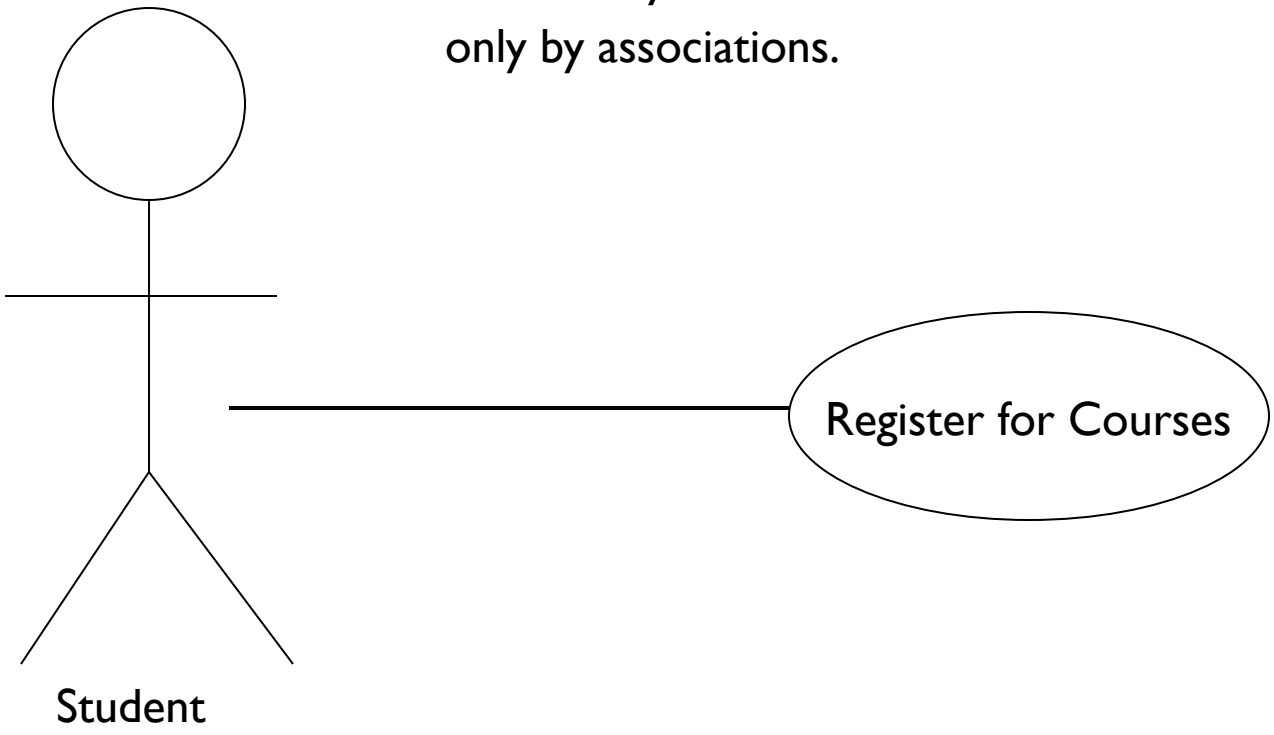
Use Case (Cont'd)

Actors can participate in a generalization relation with other actors.



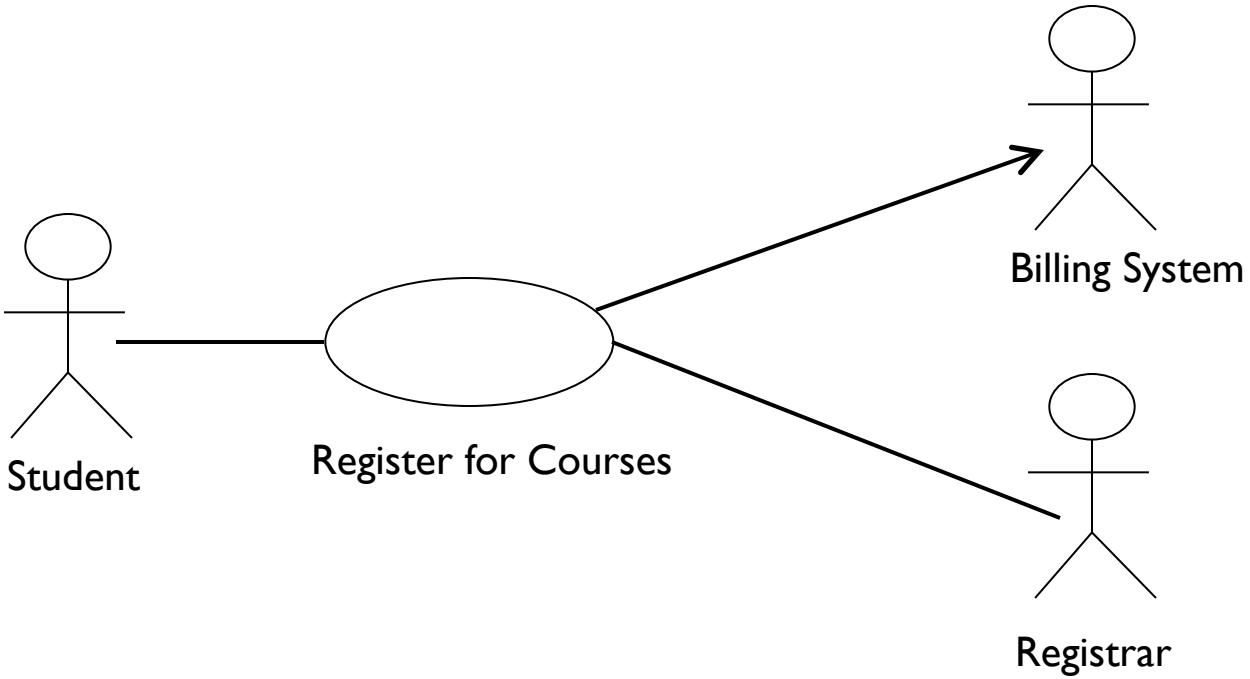
Use Case (Cont'd)

Actors may be connected to use cases only by associations.



Use Case (Cont'd)

Here we have a *Student* interacting with the *Registrar* and the *Billing System* via a "*Register for Courses*" use case.



State Machine

"The state machine view describes the dynamic behavior of objects over time by modeling the lifecycles of objects of each class. Each object is treated as an isolated entity that communicates with the rest of the world by detecting events and responding to them. Events represent the kinds of changes that objects can detect... Anything that can affect an object can be characterized as an event."

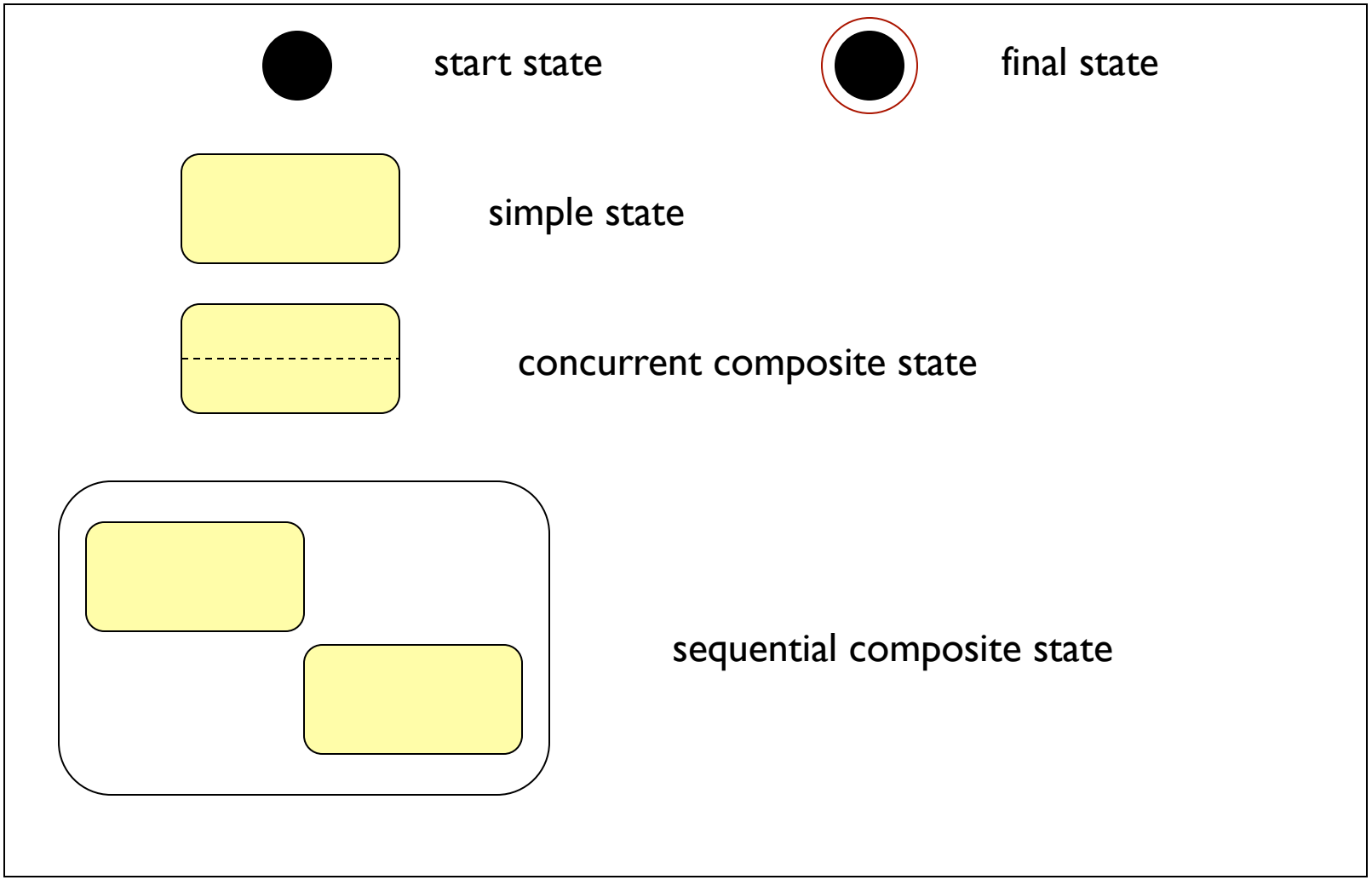
- *The UML Reference Manual, [Rumbaugh,99]*

State Machine

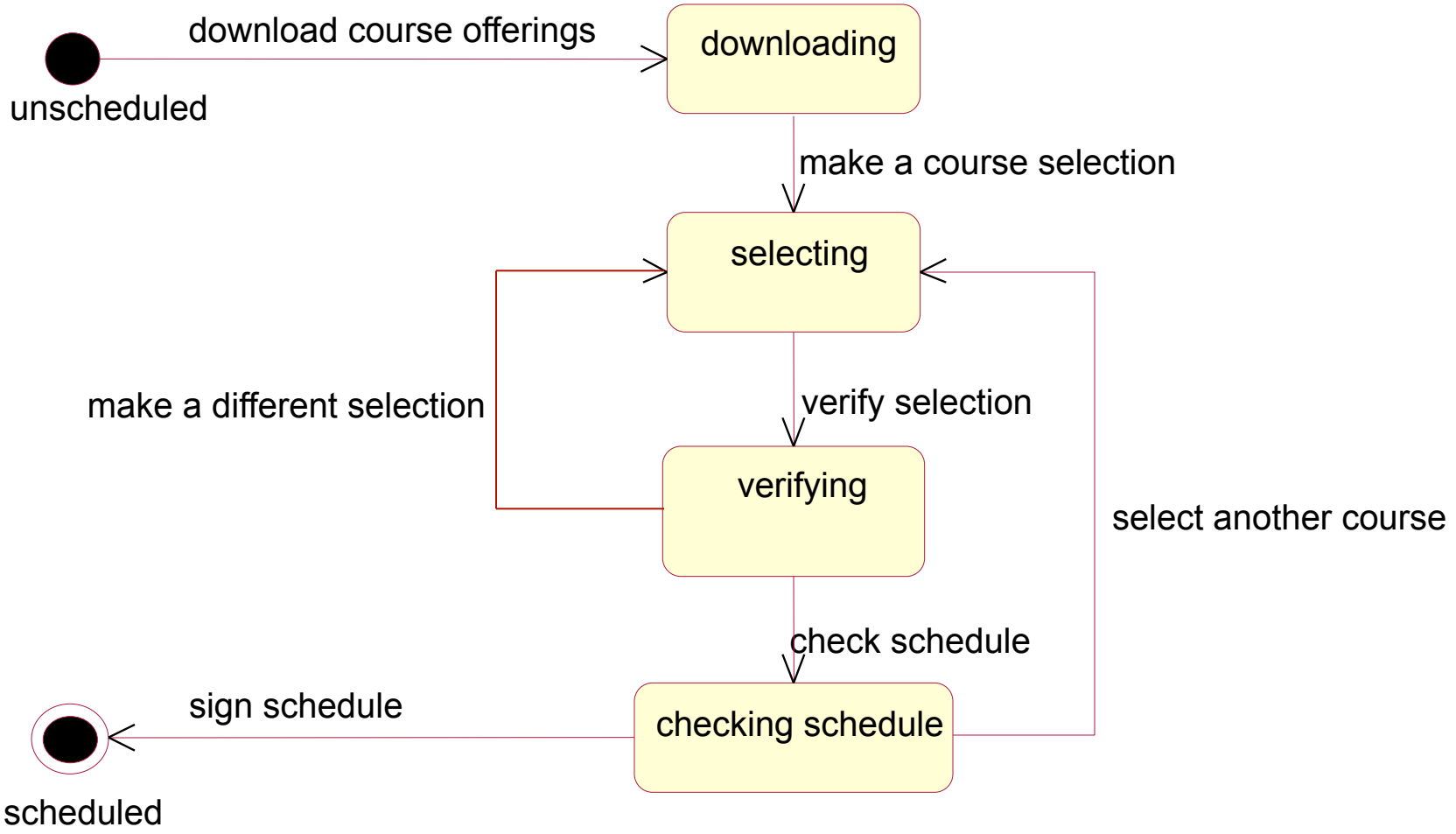
An object must be in some specific state at any given time during its lifecycle. An object transitions from one state to another as the result of some event that affects it. You may create a state diagram for any class, collaboration, operation, or use case in a UML model .

There can be only one start state in a state diagram, but there may be many intermediate and final states.

State Machine



State Machine



Sequence Diagram

A *sequence diagram* is an interaction diagram that emphasizes the time ordering of messages. It shows a set of objects and the messages sent and received by those objects.

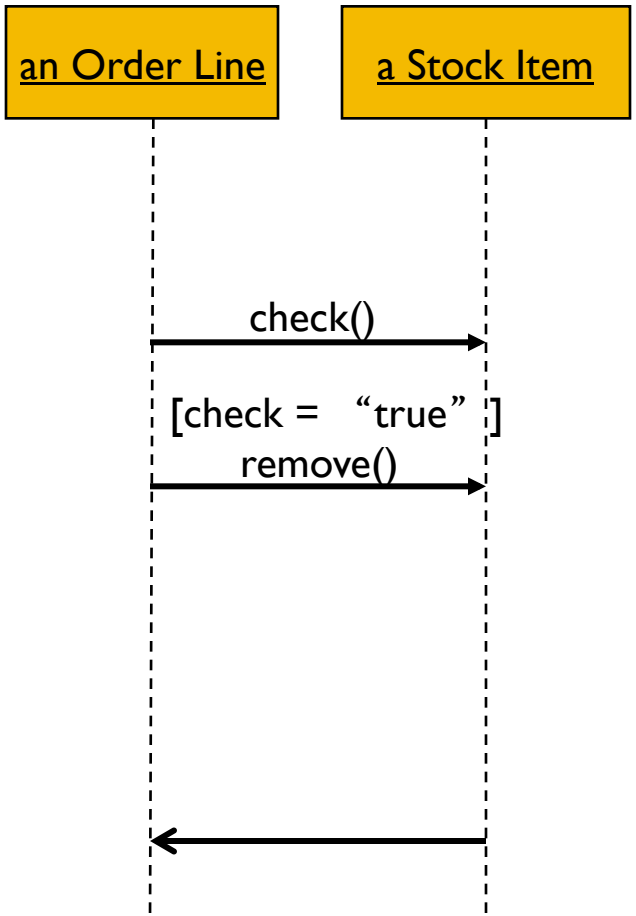
Graphically, a sequence diagram is a table that shows objects arranged along the X axis and messages, ordered in increasing time, along the Y axis.

- *The UML User Guide, [Booch,99]*

Sequence Diagram

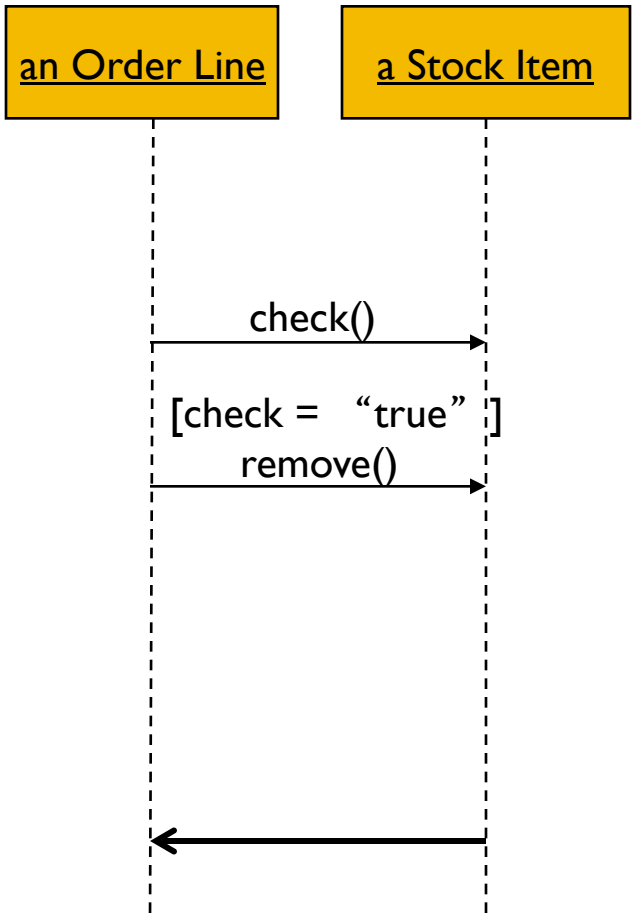
An object in a sequence diagram is rendered as a box with a dashed line descending from it. The line is called the *object lifeline*, and it represents the existence of an object over a period of time.

Sequence Diagram



Messages are rendered as horizontal arrows being passed from object to object as time advances down the object lifelines. Conditions (such as [check = "true"]) indicate when a message gets passed.

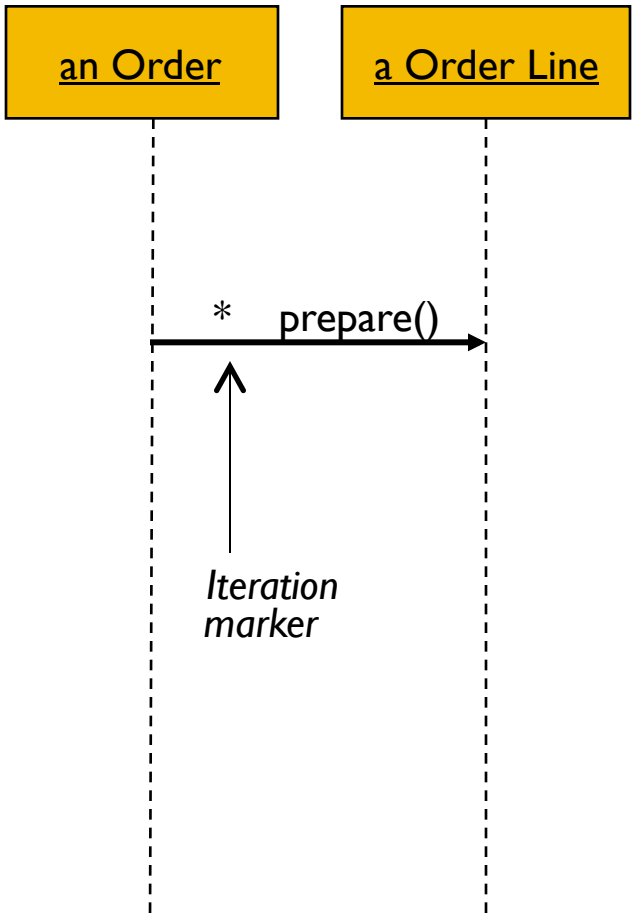
Sequence Diagram



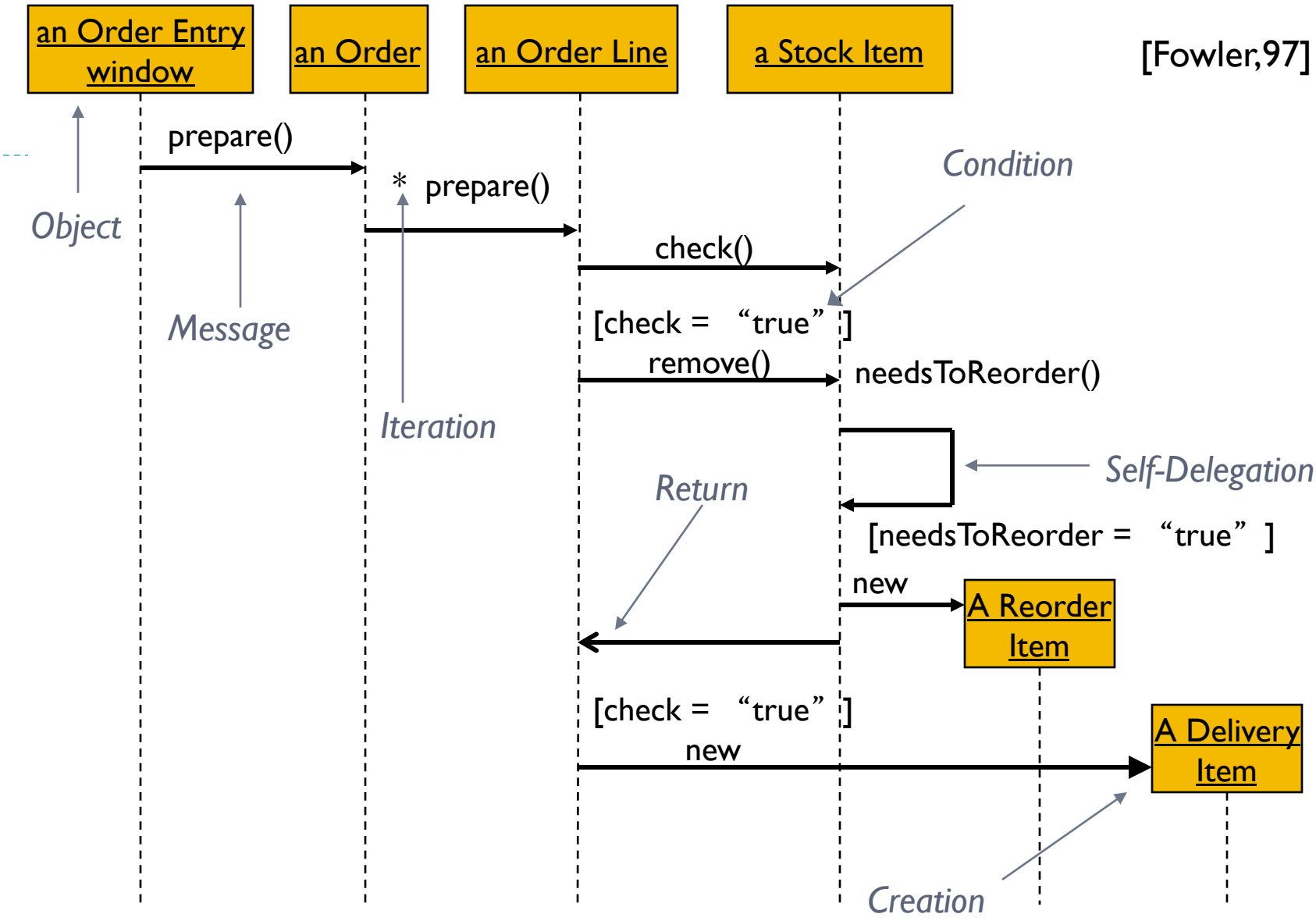
Notice that the bottom arrow is different. The arrow head is not solid, and there is no accompanying message.

This arrow indicates a **return** from a previous message, not a new message.

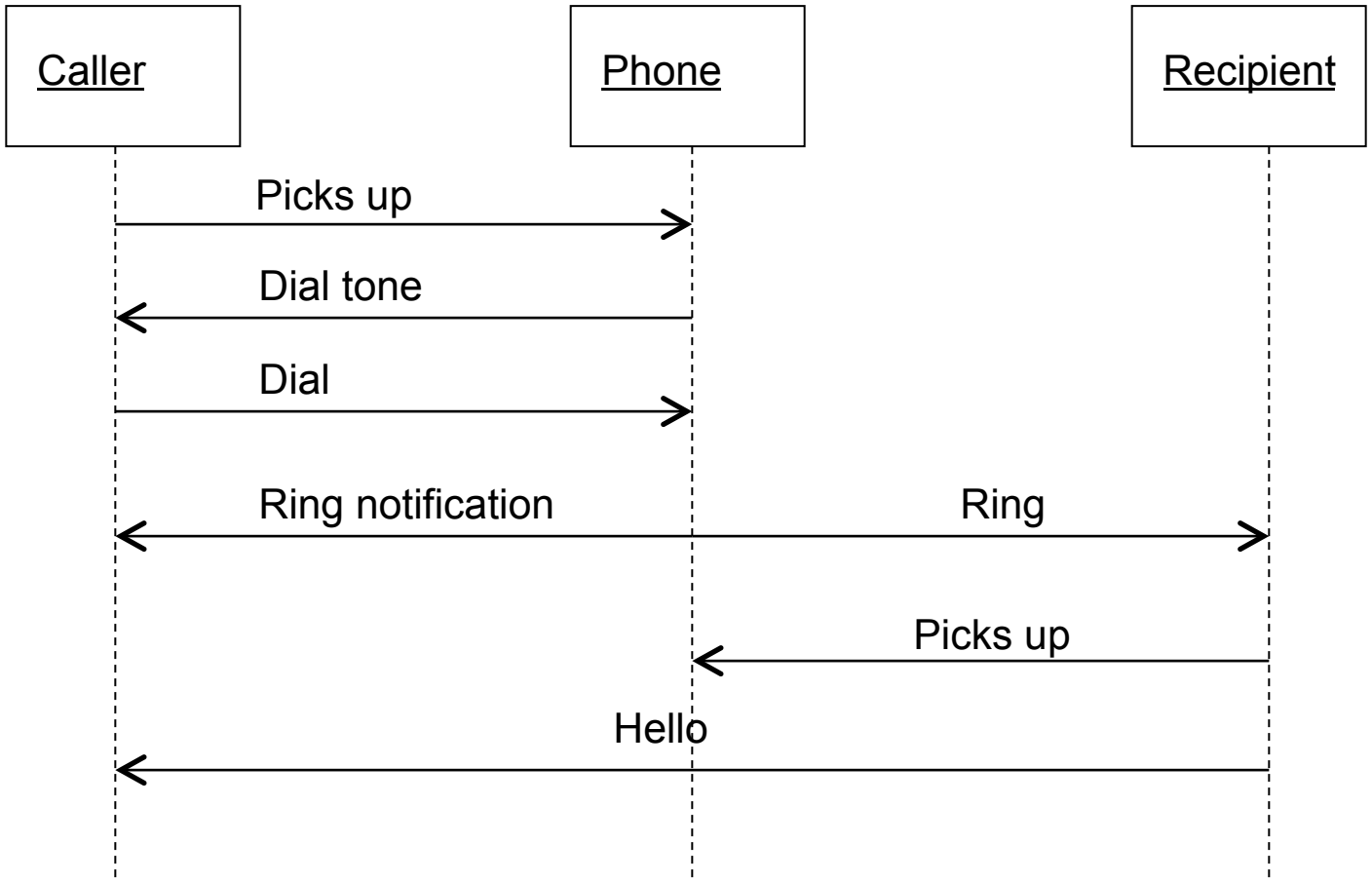
Sequence Diagram



An iteration marker, such as * (as shown), or *[i = 1..n] , indicates that a message will be repeated as indicated.


an Order Entry window
an Order
an Order Line
a Stock Item
[Fowler,97]
prepare()
* prepare()
check()
[check = "true"]
remove()
needsToReorder()
[needsToReorder = "true"]
new
A Reorder Item
[check = "true"]
new
A Delivery Item
Object
Message
Iteration
Condition
Self-Delegation
Return
Creation

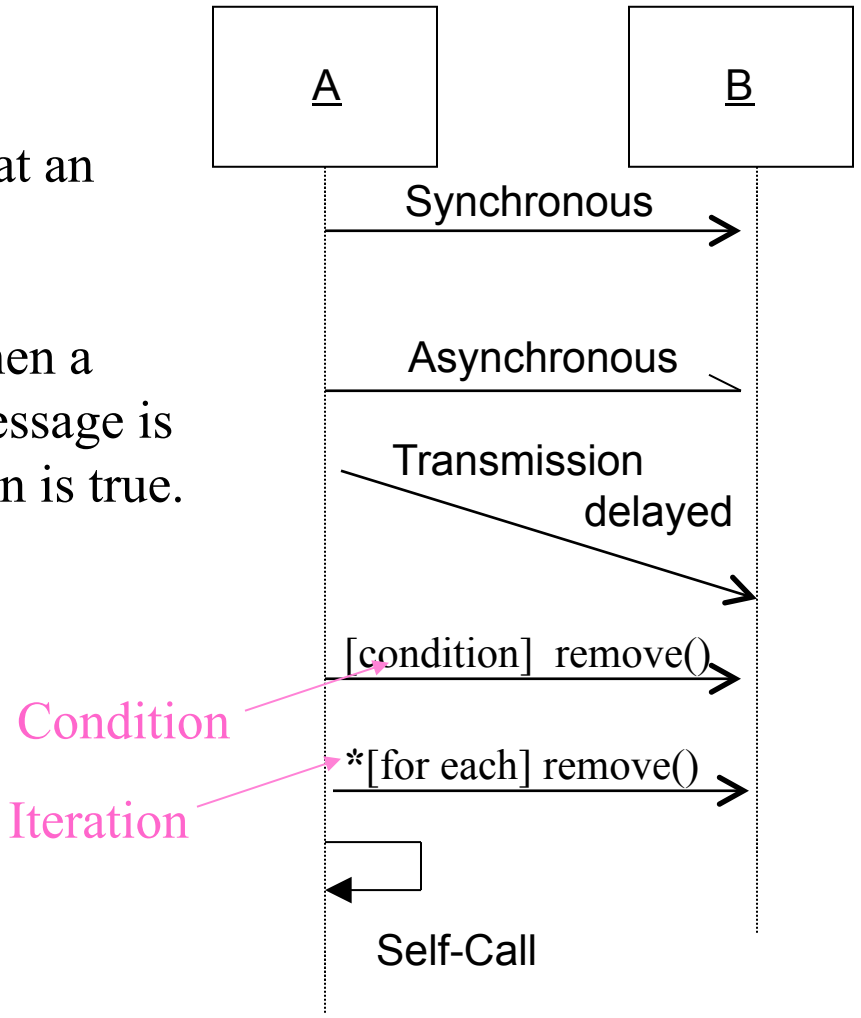
make a phone call



Sequence Diagram:Object interaction

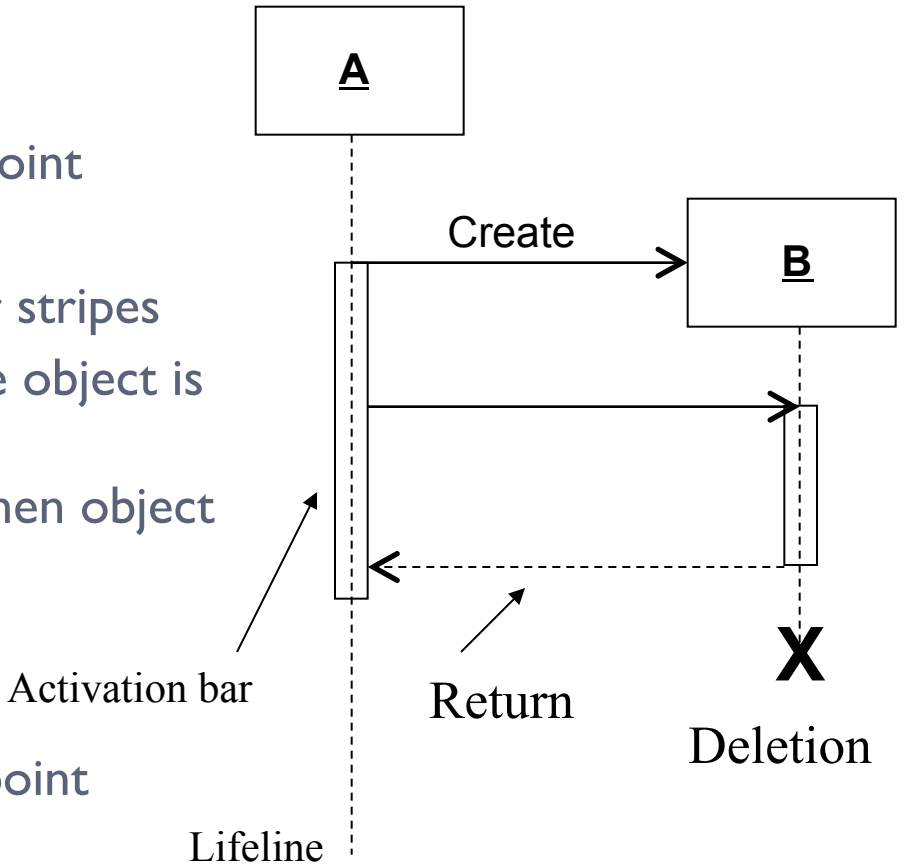
Self-Call: A message that an Object sends to itself.

Condition: indicates when a message is sent. The message is sent only if the condition is true.



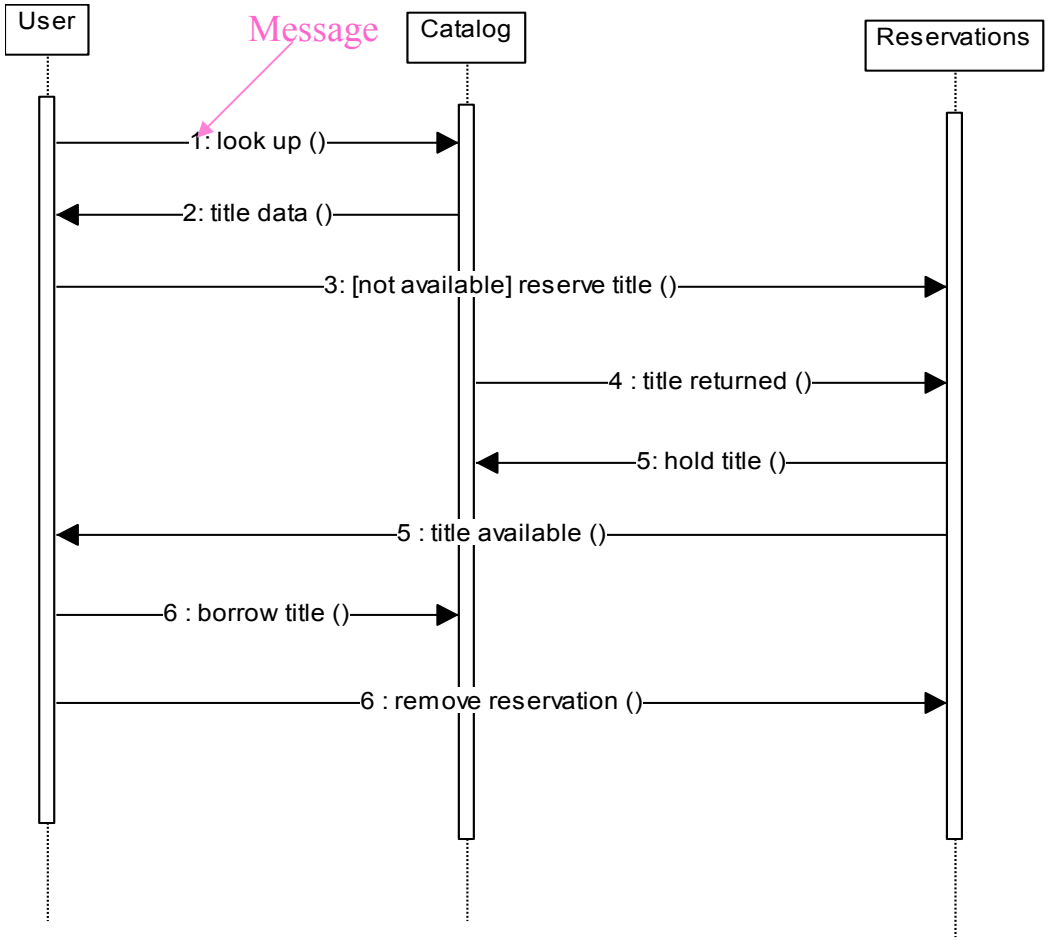
Sequence Diagrams – Object Life Spans

- Creation
  - Create message
  - Object life starts at that point
- Activation
  - Symbolized by rectangular stripes
  - Place on the lifeline where object is activated.
  - Rectangle also denotes when object is deactivated.
- Deletion
  - Placing an 'X' on lifeline
  - Object's life ends at that point



Sequence Diagrams

- Sequence diagrams demonstrate the behavior of objects in a use case by describing the objects and the messages they pass.
- The horizontal dimension shows the objects participating in the interaction.
- The vertical arrangement of messages indicates their order.
- The labels may contain the seq. # to indicate concurrency.



Sequence Diagram(序列图)

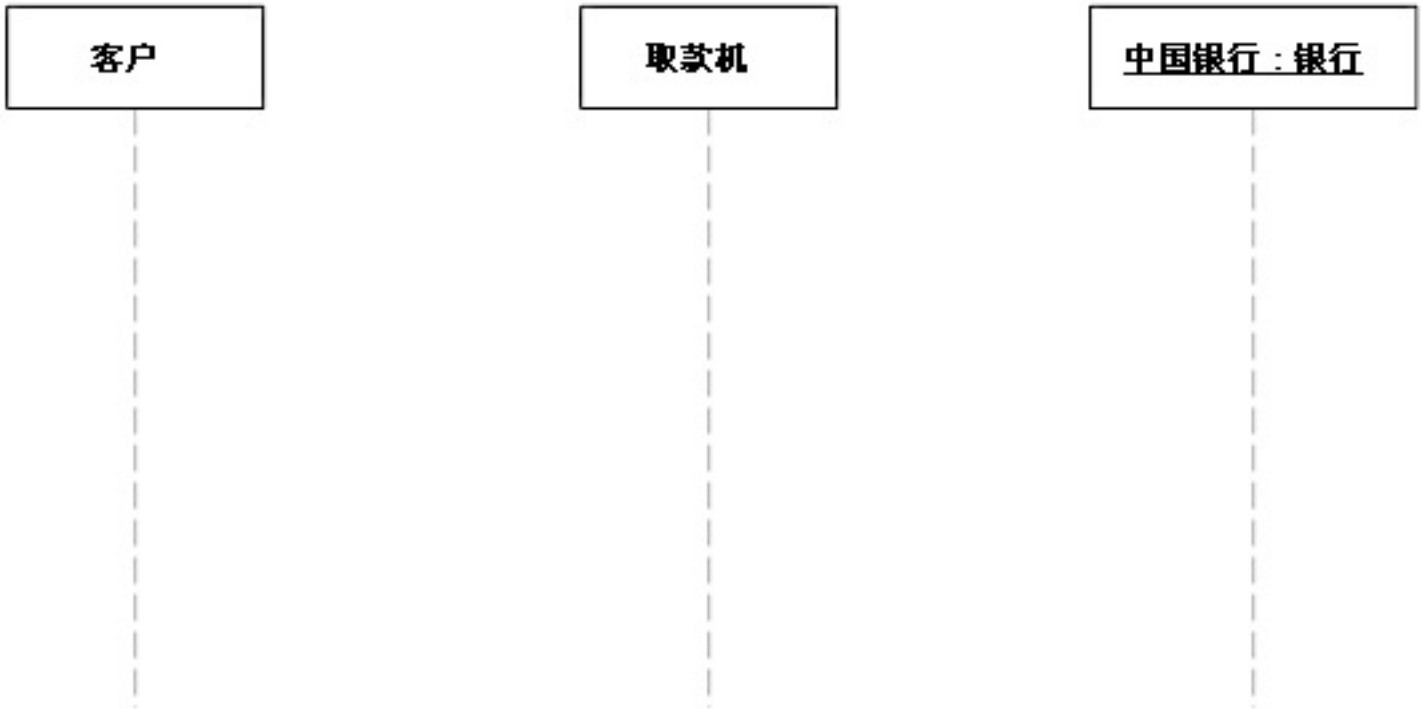
序列图主要用于展示对象之间交互的顺序。

序列图将交互关系表示为一个二维图。纵向是时间轴，时间沿竖线向下延伸。横向轴代表了在协作中各独立对象的类元角色。类元角色用生命线表示。当对象存在时，角色用一条虚线表示，当对象的过程处于激活状态时，生命线是一个双道线。

消息用从一个对象的生命线到另一个对象生命线的箭头表示。箭头以时间顺序在图中从上到下排列。

生命线：

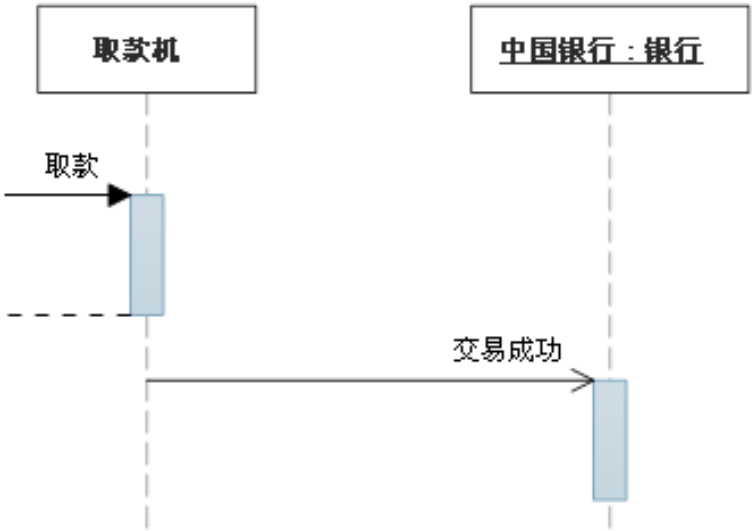
生命线名称可带下划线。当使用下划线时，意味着序列图中的生命线代表一个类的特定实例。



同步消息 & 异步消息

同步消息:发送人在它继续之前，将等待同步消息响应。

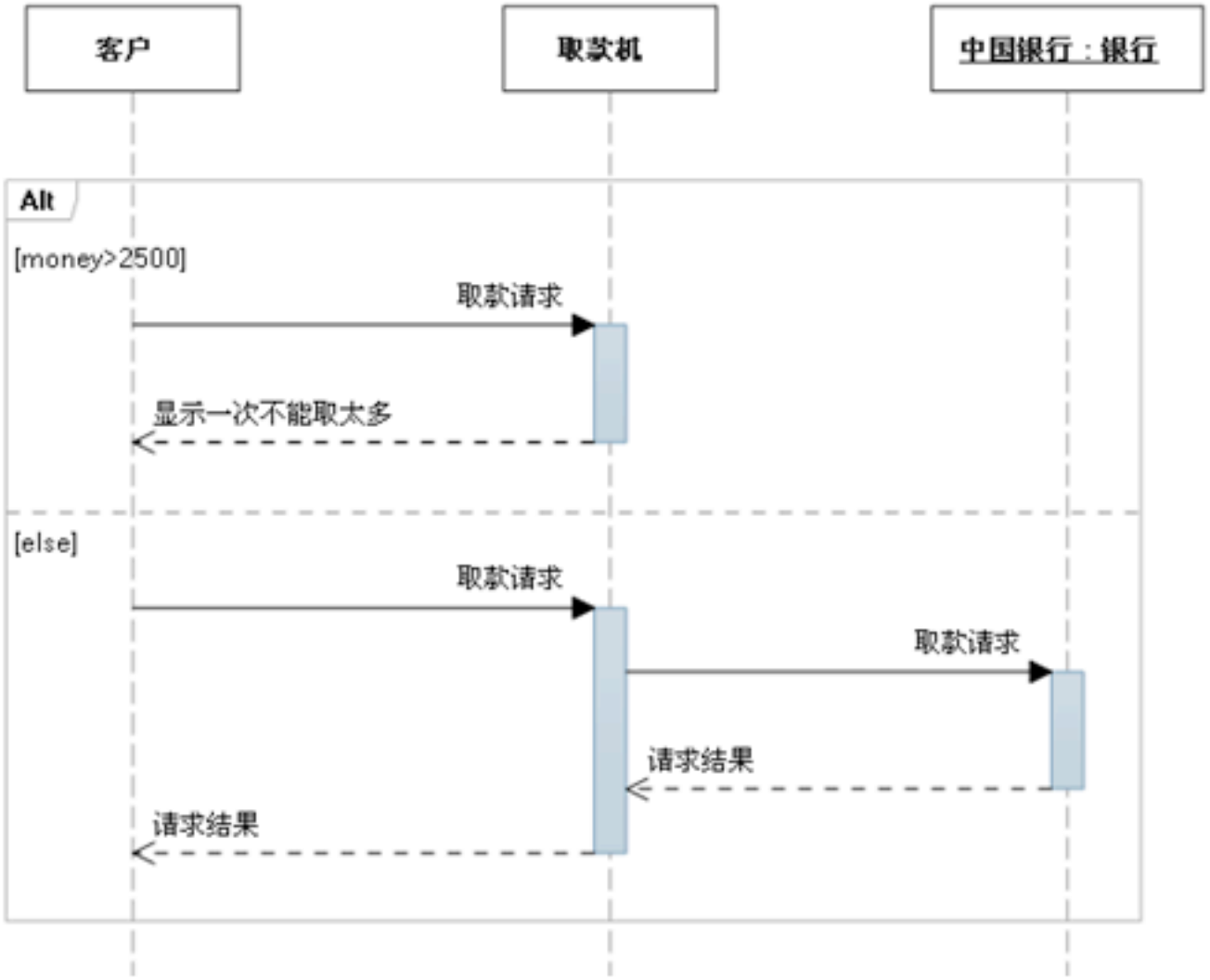
异步消息:在发送方继续之前，无需等待响应的消息。



组合片段

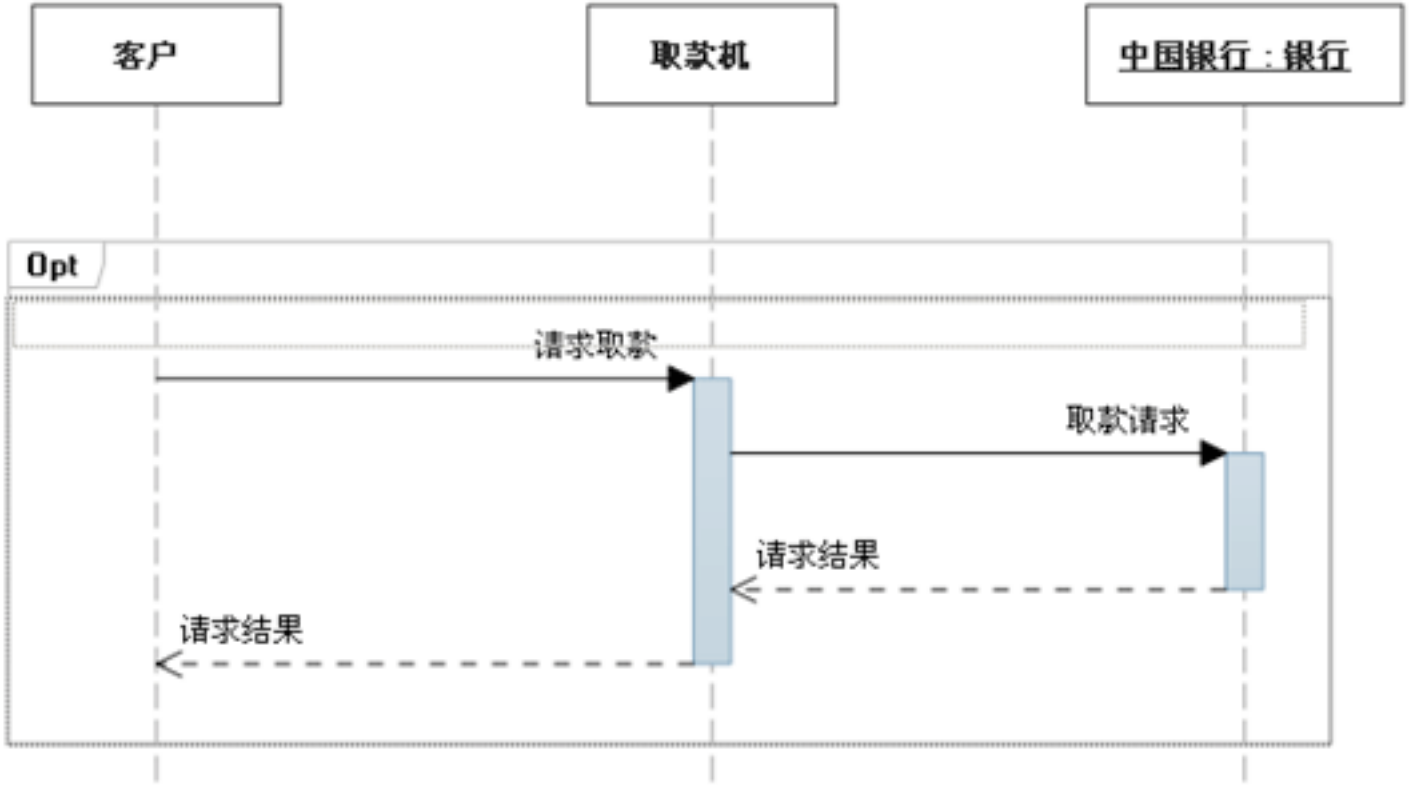
组合片段用来解决交互执行的条件及方式。它允许在序列图中直接表示逻辑组件，用于通过指定条件或子进程的应用区域，为任何生命线的任何部分定义特殊条件和子进程。

抉择（Alt）


客户
取款机
中国银行：银行
Alt
[money>2500]
取款请求
显示一次不能取太多
[else]
取款请求
取款请求
请求结果
请求结果

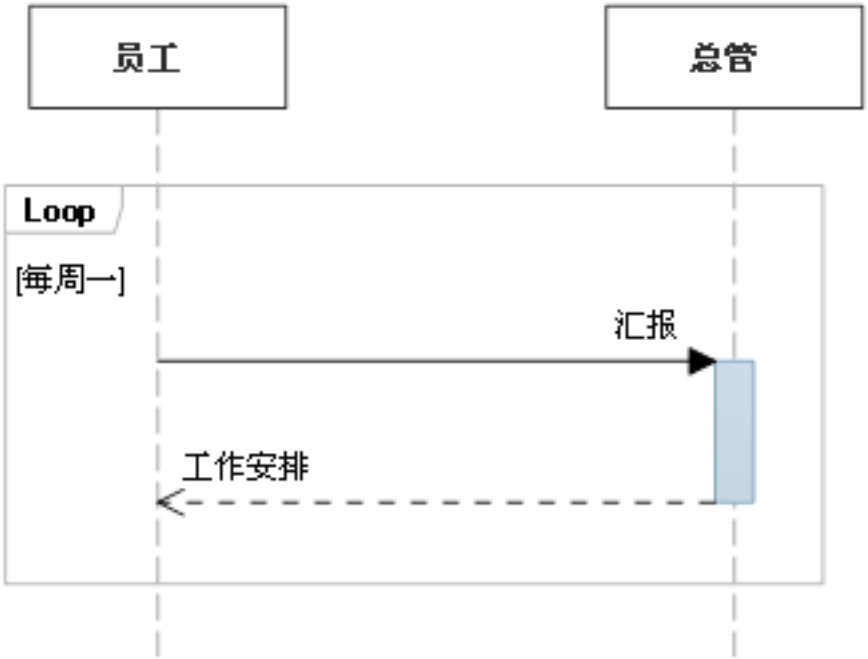
选项（Opt）

包含一个可能发生或不发生的序列

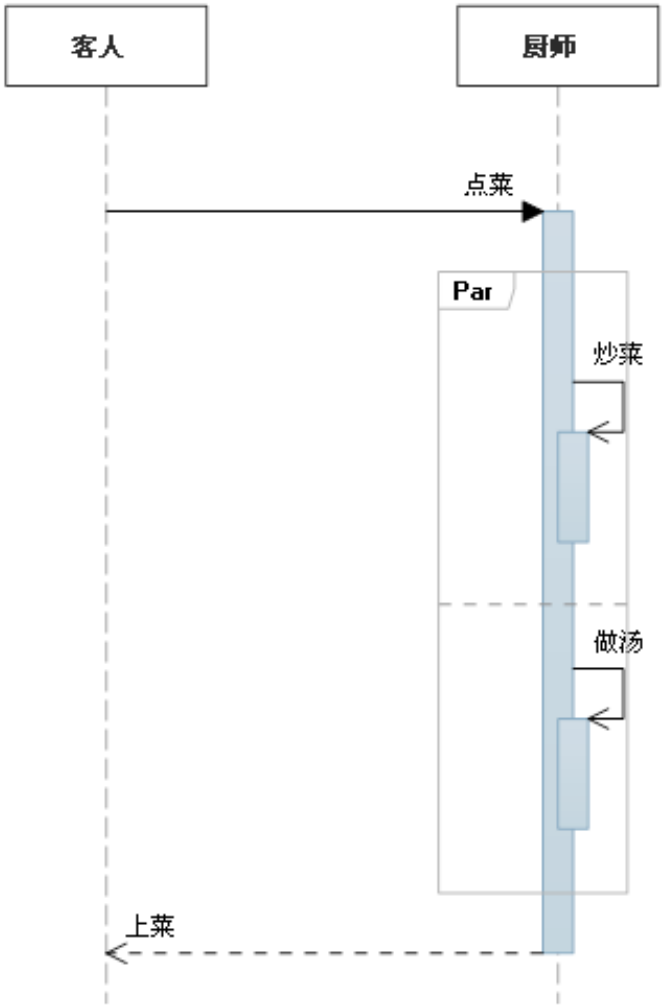


循环（Loop）

片段重复一定次数。 可以在临界中指示片段重复的条件。



并行（Par）

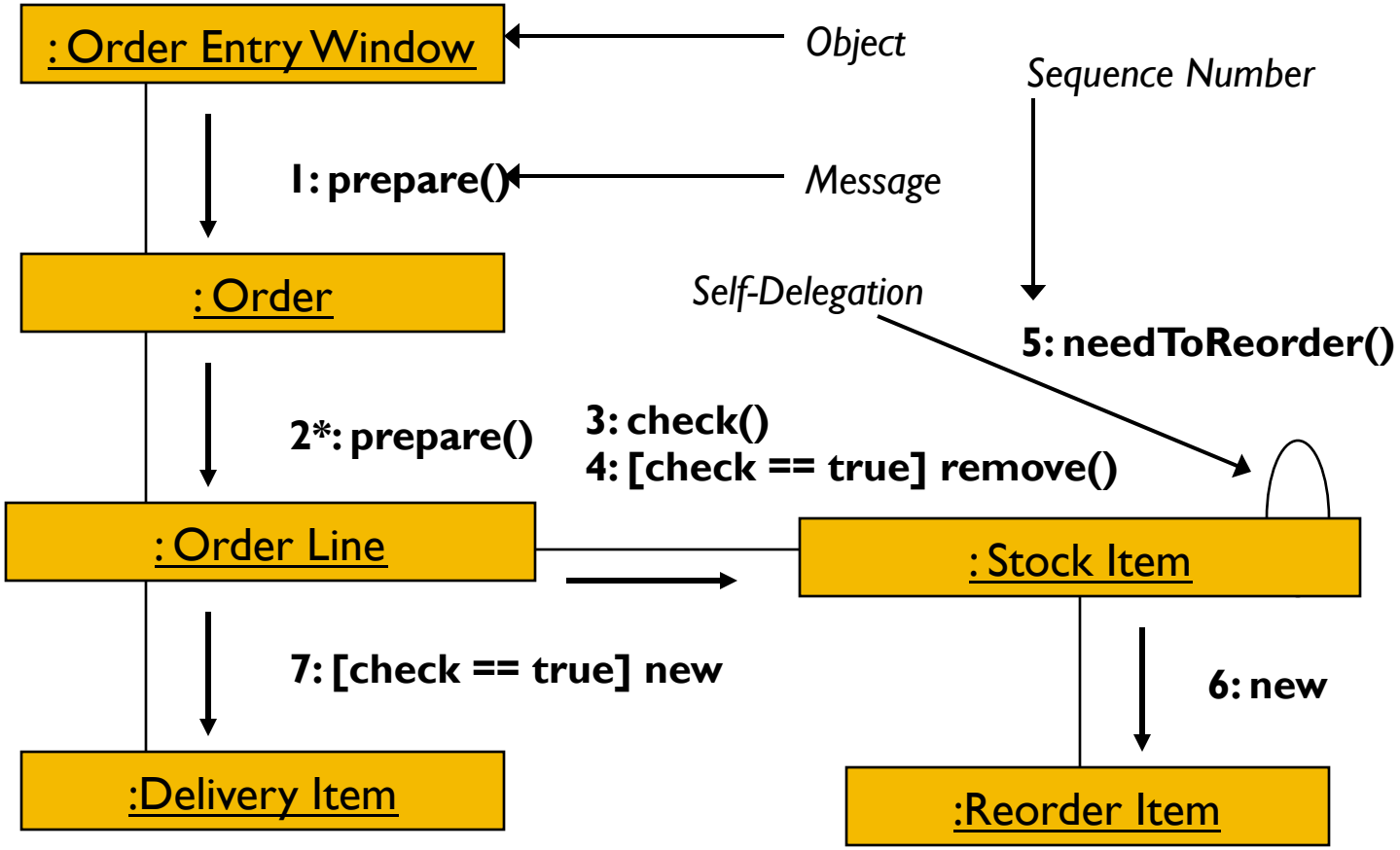


Collaboration Diagram

A collaboration diagram emphasizes the relationship of the objects that participate in an interaction. Unlike a sequence diagram, you don't have to show the lifeline of an object explicitly in a collaboration diagram. The sequence of events are indicated by sequence numbers preceding messages.

Object identifiers are of the form *objectName : className,* and either the objectName or the className can be omitted, and the placement of the colon indicates either an objectName: , or a :className.

Collaboration Diagram



[Fowler,97]

Collaboration Diagram Sequence Diagram

Both a collaboration diagram and a sequence diagram derive from the same information in the UML's metamodel, so you can take a diagram in one form and convert it into the other. They are semantically equivalent.

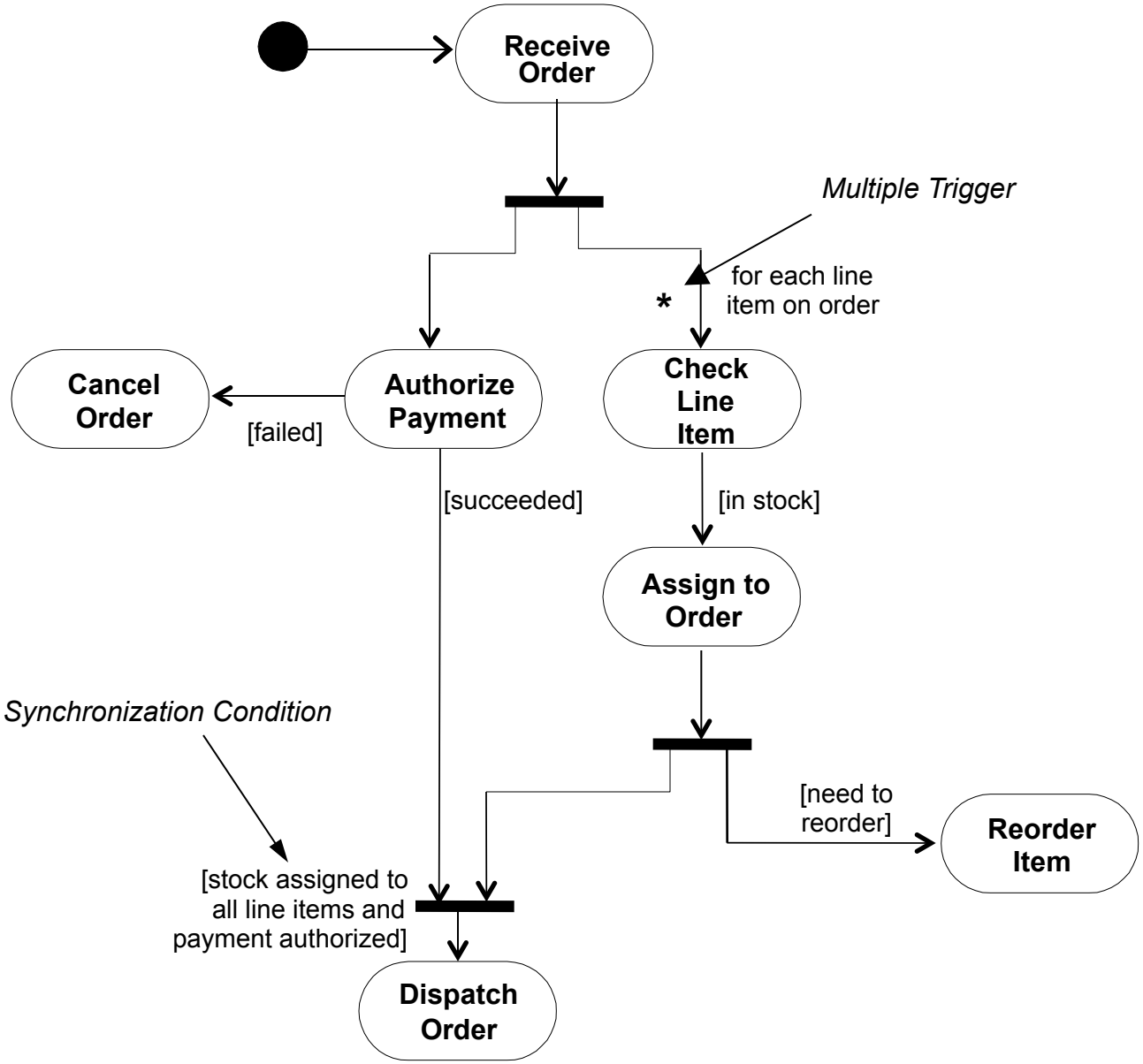
Activity Diagram

An activity diagram is essentially a flowchart, showing the flow of control from activity to activity.

Use activity diagrams to specify, construct, and document the dynamics of a society of objects, or to model the flow of control of an operation. Whereas interaction diagrams emphasize the flow of control from object to object, activity diagrams emphasize the flow of control from activity to activity. ***An activity is an ongoing non-atomic execution within a state machine.***

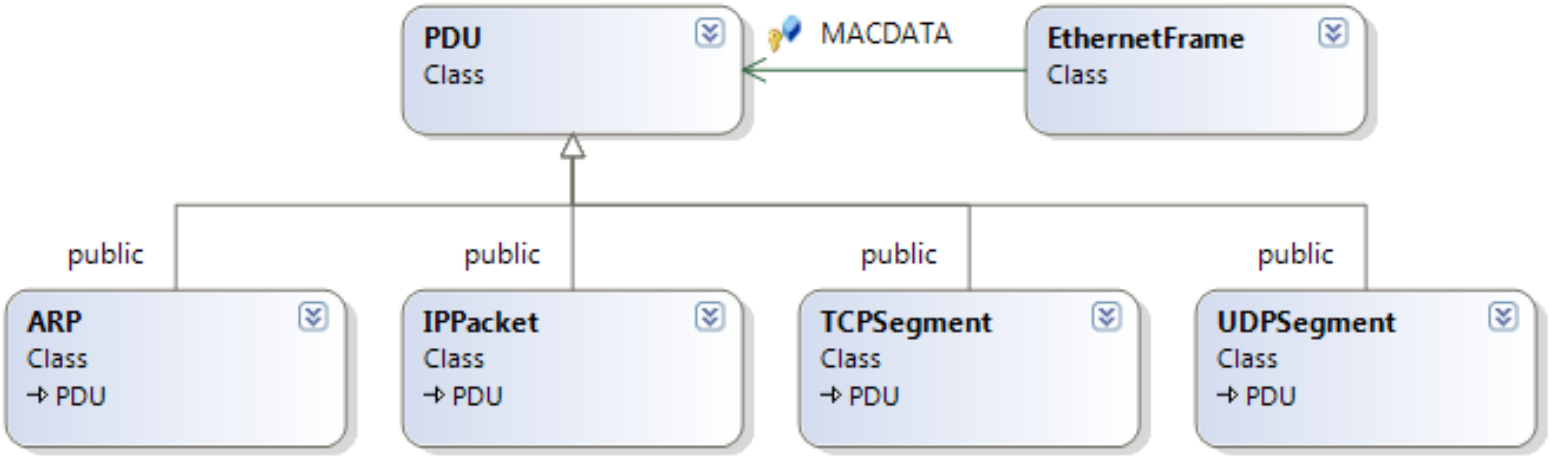
- *The UML User Guide, [Booch,99]*

[Fowler,97]

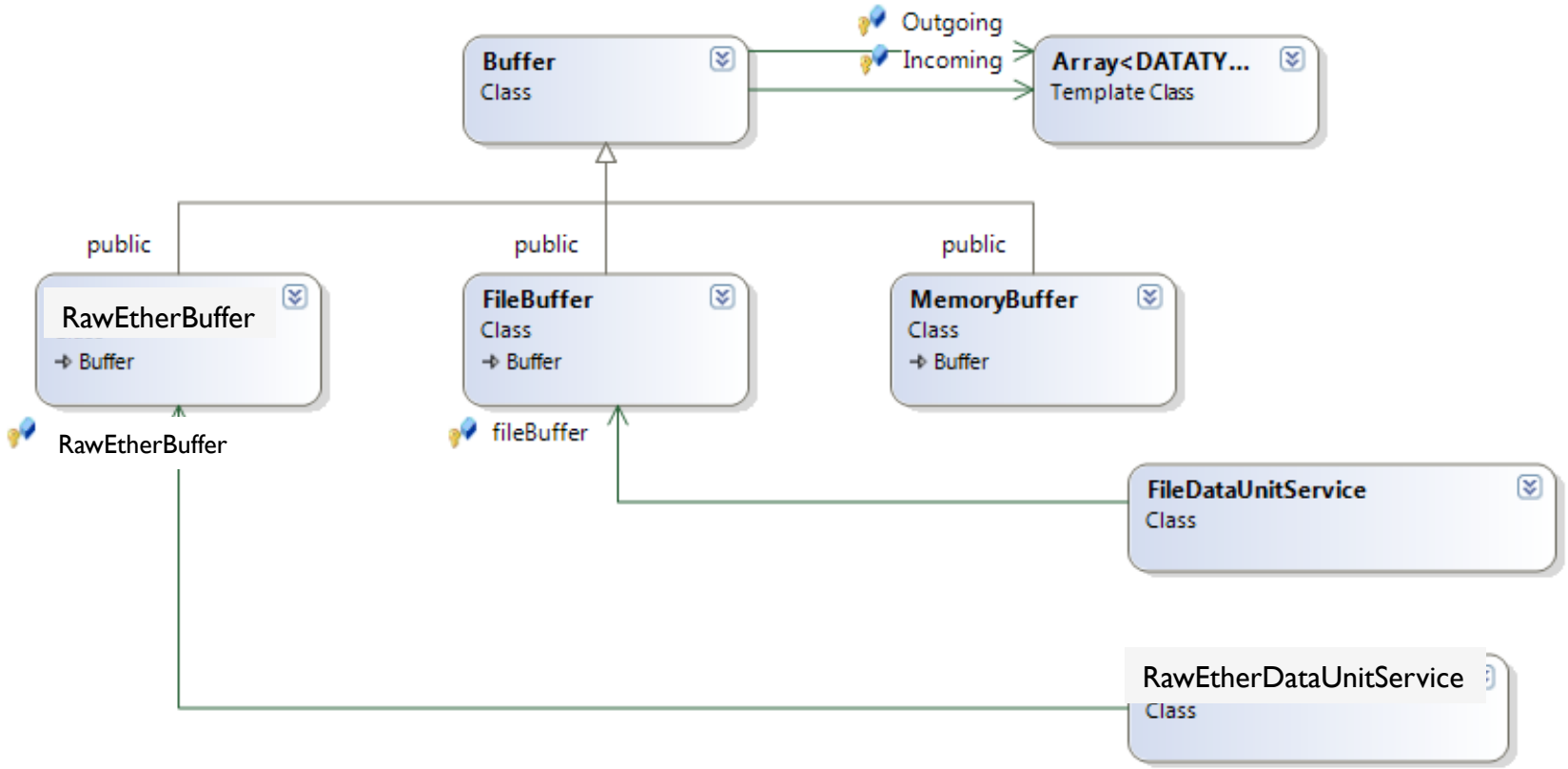

Receive Order
Multiple Trigger
for each line item on order
*
Cancel Order
Authorize Payment
Check Line Item
[failed]
[succeeded]
[in stock]
Assign to Order
Synchronization Condition
[need to reorder]
Reorder Item
[stock assigned to all line items and payment authorized]
Dispatch Order

Frame Parser

Frame and PDU



Buffer


Outgoing
Incoming
Buffer
Class
Array<DATATY...
Template Class
public
public
public
RawEtherBuffer
Buffer
FileBuffer
Class
Buffer
MemoryBuffer
Class
Buffer
RawEtherBuffer
fileBuffer
FileDataUnitService
Class
RawEtherDataUnitService
Class

CFileBuffer

```
class FileBuffer :          public Buffer
{

protected:
            CFile       *pFile;
            BOOL        EndOfFile;
public:
            FileBuffer(void){pFile=NULL; BufferType=FILEBUFFER;EndOfFile=FALSE;}
            FileBuffer(CFile *pf){pFile=pf; BufferType=FILEBUFFER;}
            ~FileBuffer(void){;}

            void SetFile(CFile* pf){pFile=pf; EndOfFile=FALSE;}

            virtual int     ReadBinary(BYTE *Data, UINT Count);

            virtual BOOL SendBinary(BYTE *Data, UINT Count);
};
```

EthernetFrame

```
//RFC 894  - A Standard for the Transmission of IP Datagrams over Ethernet Networks
class EthernetFrame
{
protected:
        BYTE Preamble[7];//The Preamble field is a 7-octet field,10101010,0xAA

        BYTE SFD;//StartFrameDelimiter:The SFD field is the sequence 10101011 0xAB, It immediately follows the preamble pattern.A
        MAC frame starts immediately after the SFD.

        BYTE DestAddress[6];//The Destination Address field shall specify the destination addressee(s) for which the MAC frame is
        intended.
        BYTE SourceAddress[6];//The Source Address field shall identify the station from which the MAC frame was initiated.
        //Each address field shall be 48 bits in length.
        //Each octet of each address field shall be transmitted least significant bit first.

        WORD EtherType;//BYTE EtherType[2];

        PDU * MACDATA;//MAC Client Data,  46 to 1500 or 1504 or 1982 octects (including PAD)

        int  MACDATALength;//Length of MACDATA(no padding), range 46, 1500 (including PAD), we just only deal with basic frame
        int   PADLength;//Length of PAD, range:0,46

        DWORD FCS;//Frame Check Sum Sequence
        CRC32 crc32;//for CRC32 Calculation
        ….
```

EthernetFrame

```
public:
        EthernetFrame(void);
        EthernetFrame(BYTE *m_DestAddress,BYTE *m_SourceAddress,WORD m_EtherType=ETHERTYPE_IPV4);
        ~EthernetFrame(void);

        //Access methods
        Void SetDestAddress(BYTE *m_DestAddress);
        const BYTE *GetDestAddress(void) const{return this->DestAddress;}
        Void SetSourceAddress(BYTE *m_SourceAddress);
        const BYTE *GetSourceAddress(void) const{return this->SourceAddress;}
        Void SetEtherType(WORD m_EtherType);
        WORD GetEtherType() const{return this->EtherType;}
        BOOL SetMACDATA(PDU * DATA);
        const PDU *GetMACDATA() const{return this->MACDATA;}
        UINT GetMACDATALength() const{return this->MACDATALength;}
        UINT GetPADLength() const{ return this->PADLength;} //We can not get the PAD Length if we read a fake frame.
        DWORD GetFCS() const{return this->FCS;}
        DWORD ComputeFCS();
        //Input/Output methods
        BOOL Write(Buffer &Link);//Write out Whole Frame: from preamble to FCS
        BOOL FindNextEtherFrame(Buffer &Link);//Find the position of next Etherframe in the incoming buffer;
        BOOL Read(Buffer &Link);//read whole frame
        BOOL ReadDynamic(Buffer&Link);//called by Read(), read different frame payload;IP or ARP

        BOOL WriteInnerFrame(Buffer &Link);//Write out inner frame: from Destination to MACDATA, not
        including preamble,SFD and FCS
};
```

PDU

```
class PDU          //interface for Protocol Data Unit
{
        public:
        Virtual int GetTotalLength() const= 0;
        Virtual void SetTotalLength(int length)= 0;
        Virtual int GetHeaderLength() const = 0;
        Virtual void SetHeaderLength(int length)= 0;

        Virtual void PrintHeader() const= 0;
        Virtual void Print() const= 0;

        virtual BOOL Write(Buffer &link)= 0;
        Virtual BOOL Read(Buffer &Link)= 0;

};
```

```
class ARP : public PDU
{
        protected:
          //http://www.iana.org/assignments/arp-parameters/
           static const WORD HRD_ETHERNET = 1;          // Ethernet
           static const WORD HRD_LOOPBACK = 772;        // Loopback interface
           static const WORD HRD_IEEE80211 = 801;       // IEEE 802.11
           static const WORD HRD_NONE = 0xFFFE;         // Zero header device

           static const WORD PRO_IP = 0x0800;

           // Opcodes.
           static const WORD OP_REQUEST= 1;//ARP REQUEST
           static const WORD OP_REPLY= 2;//ARP REPLY
           static const WORD OP_RREQUEST= 3;//RARP request Reverse
           static const WORD OP_RREPLY= 4;//RARP reply Reverse

           // Constants from RFC 3927 - Dynamic Configuration of IPv4 Link-Local Addresses
           static const int PROBE_WAIT = 1;//1 second   (initial random delay)
           static const int PROBE_NUM = 3;//3    (number of probe packets)
           static const int PROBE_MIN = 1;//1 second   (minimum delay till repeated probe)
           static const int PROBE_MAX = 2;//2 seconds  (maximum delay till repeated probe)

           static const int ANNOUNCE_WAIT = 2;//2 seconds  (delay before announcing)
           static const int ANNOUNCE_NUM = 2;//2    (number of announcement packets)
           static const int ANNOUNCE_INTERVAL = 2;//2 seconds  (time between announcement packets)
           static const int MAX_CONFLICTS = 10;//10          (max conflicts before rate limiting)
           static const int RATE_LIMIT_INTERVAL = 60;//60 seconds  (delay between successive attempts)
           static const int DEFEND_INTERVAL = 10;//10 seconds  (minimum interval between defensive ARPs).

          WORD     HardwareType;//hrd: hardware type.
          WORD     ProtocolType;//pro: protocol type
          BYTE     HardwareAddrLen;//hln:  length of each hardware address.
          BYTE     ProtocolAddrLen;//pln: length of each protocol address.
          WORD     OpCode;//op:opcode

          BYTE     SenderHardwareAddr[6];//sha:hardware address of sender of this packet.
          DWORD SenderProtocolAddr;//protocol address of sender of this packet.
          BYTE     TargetHardwareAddr[6];//tha:hardware address of target of this packet.
          DWORD TargetProtocolAddr;//protocol address of target.
```

ARP

```
public:
        ARP(void);
        ~ARP(void);

        Virtual int GetTotalLength() const {return 28;}//Total Length 28 bytes, 18 bytes padding
        needed for a Etherent frame
        Virtual void SetTotalLength(int length) {;}//nop
        Virtual int GetHeaderLength() const {return 8;}//consider 8 bytes
        Virtual void SetHeaderLength(int length){;}

        Virtual void PrintHeader() const{;}
        Virtual void Print() const{;}

        virtual BOOL Write(Buffer &link){return TRUE;}
        Virtual BOOL Read(Buffer &Link){return TRUE;}

};
```

FileDataUnitService

```
#include "FileBuffer.h"
#include "EthernetFrame.h"

//Protocol Data Unit (Frame) file I/O service class
class FileDataUnitService
{
protected:
		FileBuffer	fileBuffer;

		CFile frameFile;

public:

		FileDataUnitService(void);
		~FileDataUnitService(void);

		BOOL OpenForReading(const TCHAR* inFileName);
		BOOL OpenForWriting(const TCHAR* outFileName);
		BOOL IsFileOpen() const {return !((HANDLE)frameFile == CFile::hFileNull);}
		BOOL Close();


		BOOL Read(EthernetFrame &frame);
		BOOL Write(EthernetFrame &frame);
		BOOL WriteTesting(const BYTE * Data,UINT DataLength);		//test file writing only
};
```

References

1. [Booch2005] , Grady Booch, James Rumbaugh, Ivar Jacobson, The Unified Modeling Language User Guide,2nd ed, Addison Wesley, 2005
2. [Rambaugh2005] Rumbaugh, James, Ivar Jacobson, Grady Booch, The Unified Modeling Language Reference Manual, 2nd ed, Addison Wesley, 2005
3. [Jacobson99] Jacobson, Ivar, Grady Booch, James Rumbaugh, The Unified Software Development Process, Addison Wesley, 1999
4. [Fowler, 2003] Fowler, Martin, Kendall Scott, UML Distilled: Applying the Standard Object Modeling Language,3rd ed, Addison Wesley, 2003.
5. [Blaha,2004] Michael R. Blaha, James R Rumbaugh, Object-Oriented Modeling and Design with UML (2nd Edition), Prentice Hall,2004
6. [Erich94] Erich Gamma , Richard Helm, Ralph Johnson, John Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1994