
SYSC 3303 Real-Time Concurrent Systems

Review of the UML (Unified Modeling Language) Class Diagrams, Object Diagrams, Collaboration Diagrams

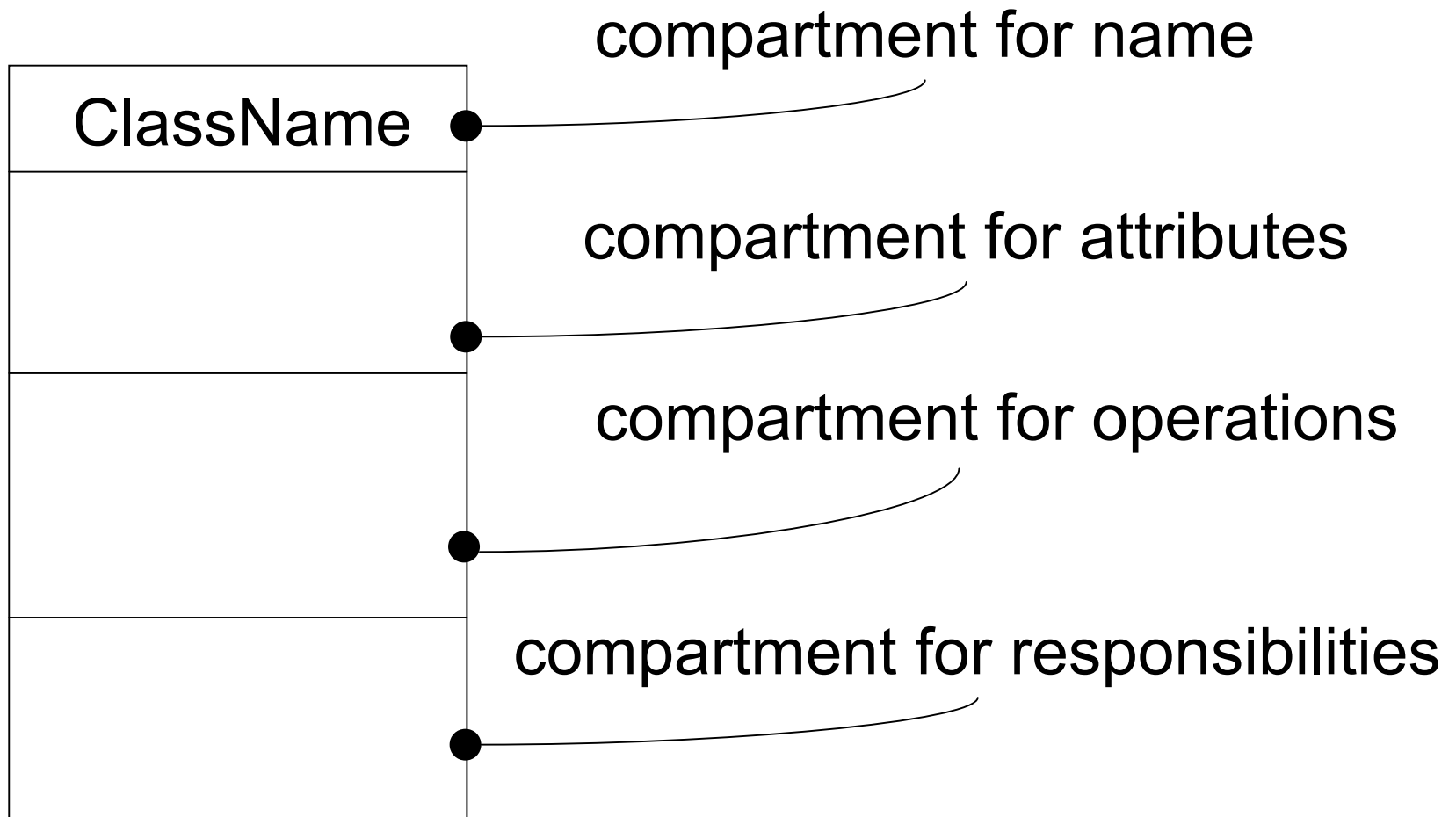
Modeling Concurrency with the UML and Recent Changes to the UML

- Copyright © 2001-2004 D.L. Bailey and 2007 L.S. Marshall, Systems and Computer Engineering, Carleton University
- revised July 9th, 2007

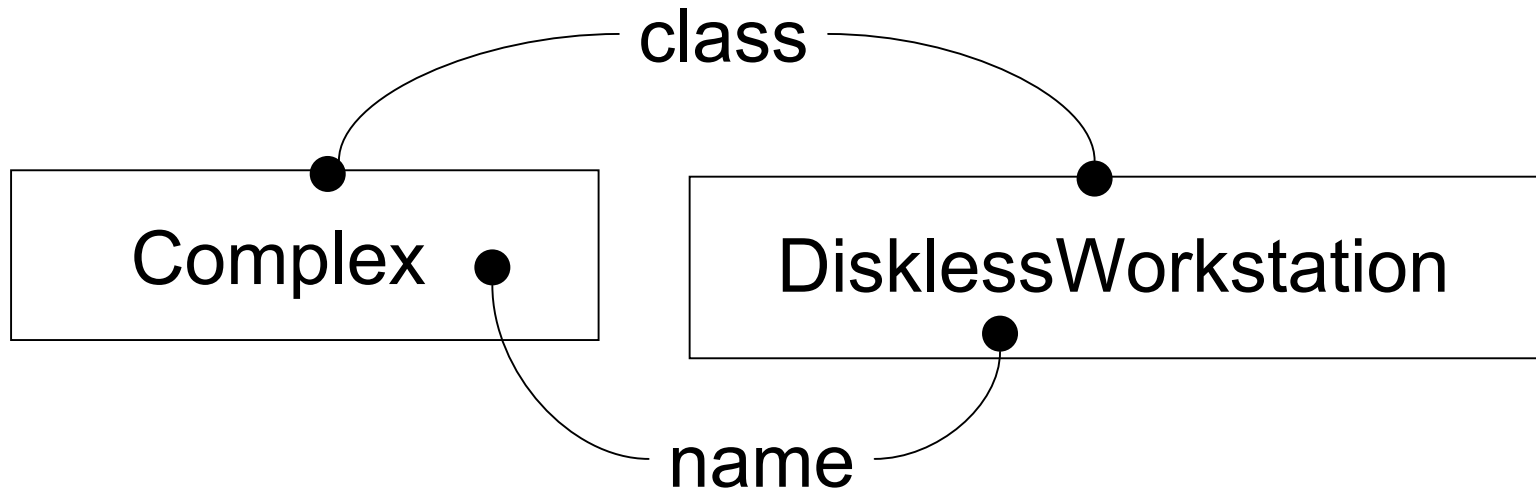
What's in This Set of Slides?

- Review of key elements of UML class diagrams, object diagrams, and collaboration diagrams
 - all students should be familiar with the UML from one or more of SYSC 1101, SYSC 2004, SYSC 3100, COMP 1006, COMP 3004, ...
- For more information, consult
 - *The Unified Modeling Language User Guide*, Grady Booch, James Rumbaugh, Ivar Jacobson, Addison-Wesley, 1999, ISBN 0-201-57168-4
 - *OMG Unified Modeling Language Specification, Version 1.4*
(<http://www.omg.org/technology/documents/formal/uml.htm>)

UML Notation for Classes

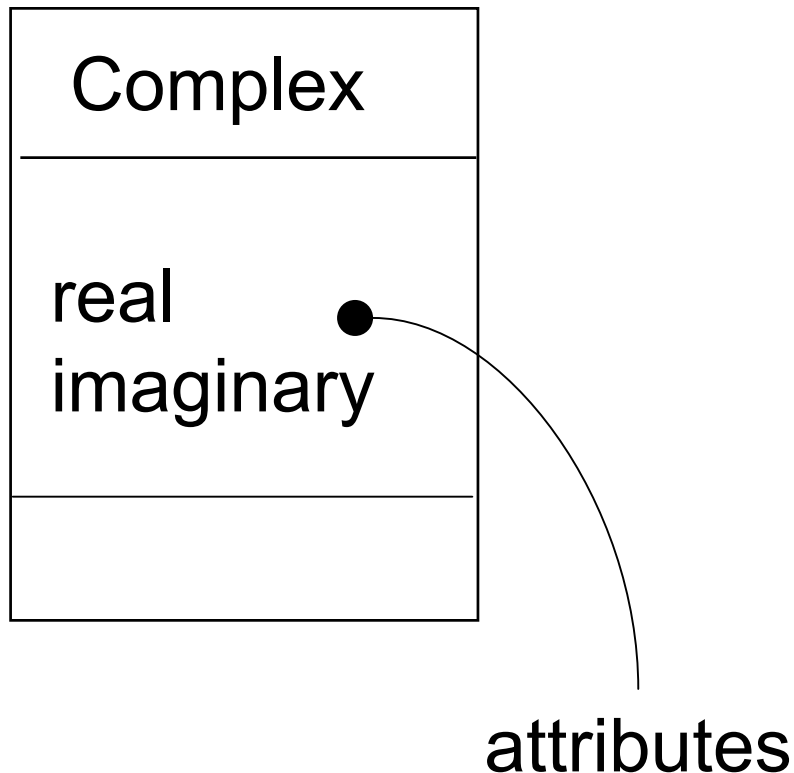


Class Names



- Every class has a name - usually a noun or short noun phrase drawn from the system being modeled
- Convention: capitalize the first letter of every word in a class name
- A class may be drawn showing only its name

Attributes

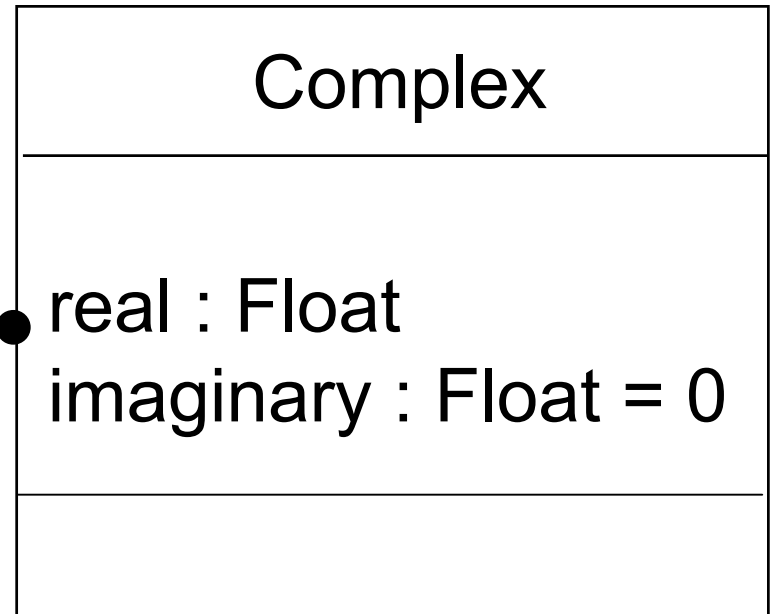


- An attribute name is usually a noun or short noun phrase
- Convention: capitalize the first letter of every word in the attribute except the first letter
- Attributes can be drawn showing only their names, as in this example

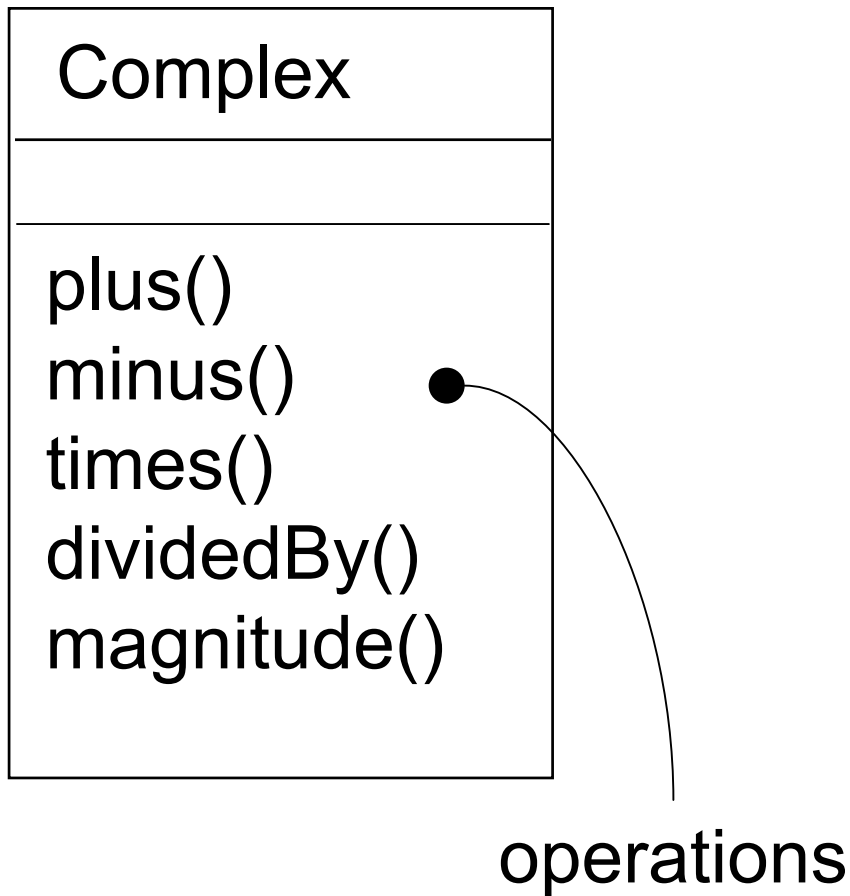
Attributes

- Can also specify an attribute's class and a default initial value

attributes

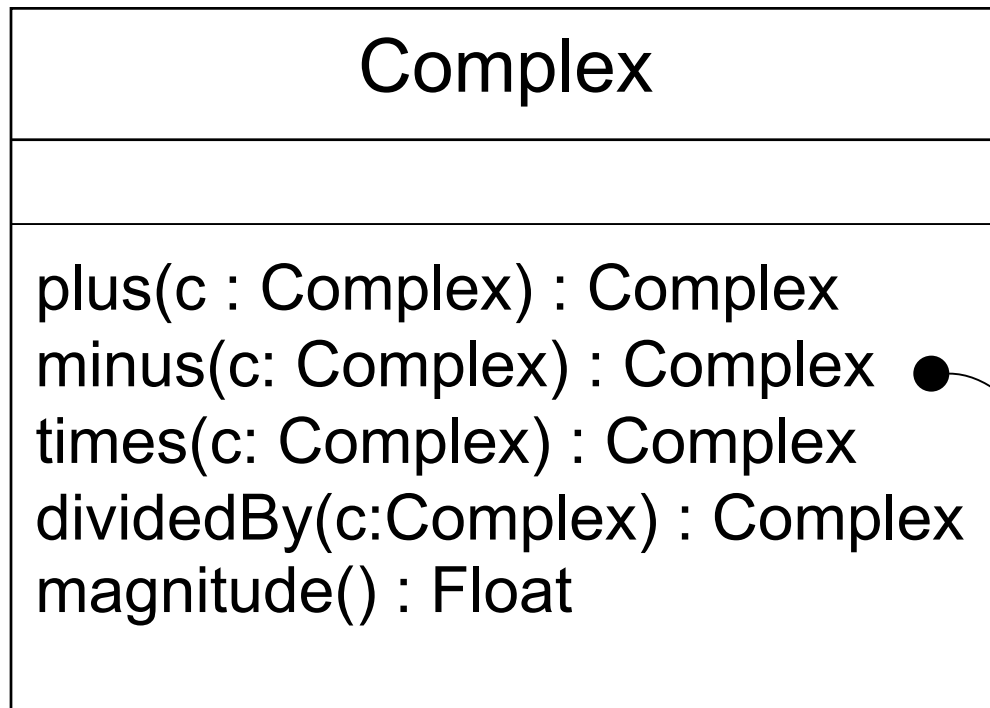


Operations



- An operation name is usually a verb or short verb phrase
- Convention: capitalize the first letter of every word in an operation except the first letter
- Operations can be drawn showing only their names, as in this example

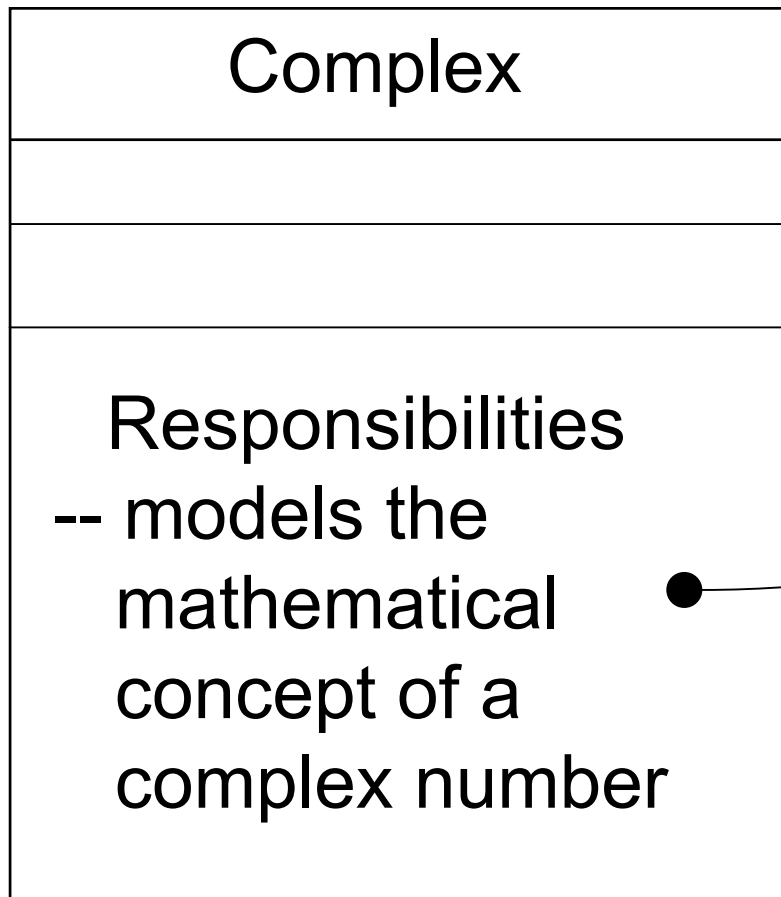
Operations



operations

- Can also specify an operation by stating its signature (name, type and default value (if any) of all parameters; and return type for functions)

Responsibilities



responsibilities

- Responsibilities are presented as free-form text

Additional Class Notation

- An *italicized* class name indicates that the class is abstract
- An *italicized* operation indicates that the operation is abstract
- An underlined attribute (operation) indicates that the attribute (operation) has classifier scope (in Java, these are equivalent to `static` variables and `static` methods)

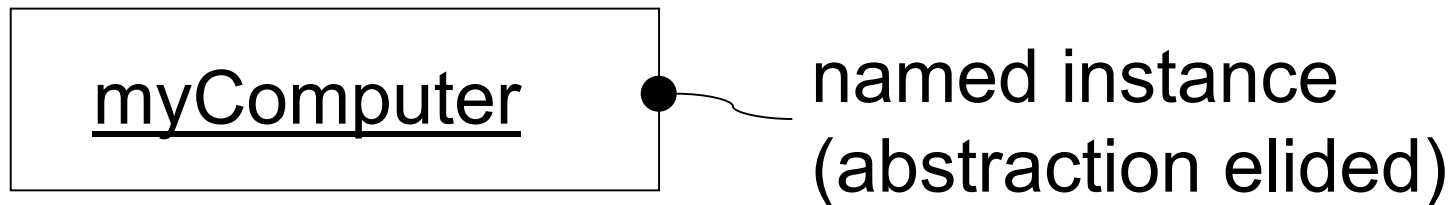
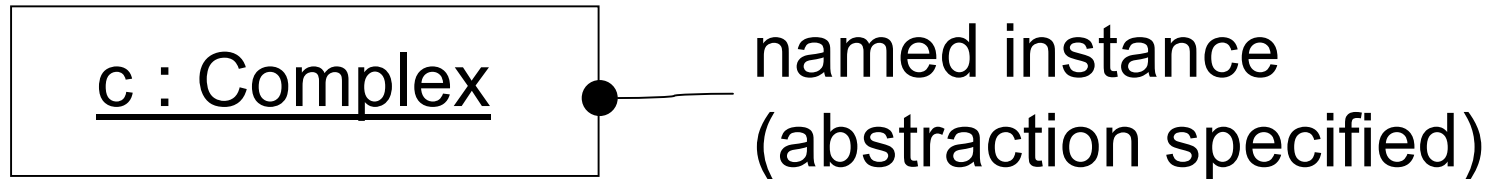
Additional Class Notation

- For attributes and operations:
 - **+** **name** means **name** has public visibility
 - **#** **name** means **name** has protected visibility
 - **-name** means **name** has private visibility
- Public/protected/private visibility in the UML doesn't mean quite the same thing that it does in Java (but it's close)

UML Notation for Objects

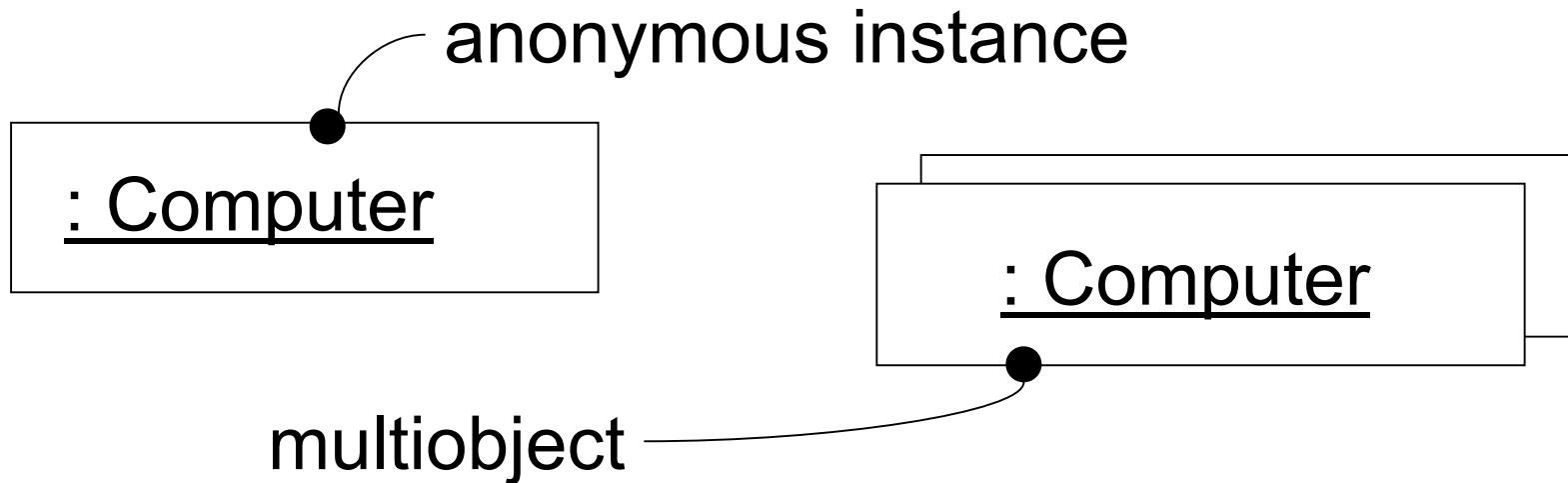
- Like a class, an object is drawn as a 2-D box
- To distinguish between a class and an instance, the instance name (and its class, if provided) is underlined
- Convention: capitalize the first letter of every word in an object name except the first letter; e.g., `c`, `myComputer`
- See examples on the next slide

UML Notation for Objects

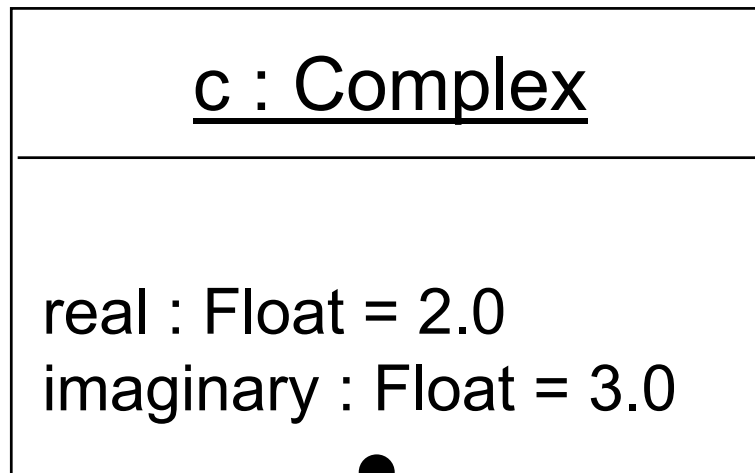


Anonymous Instances and Multiobjects

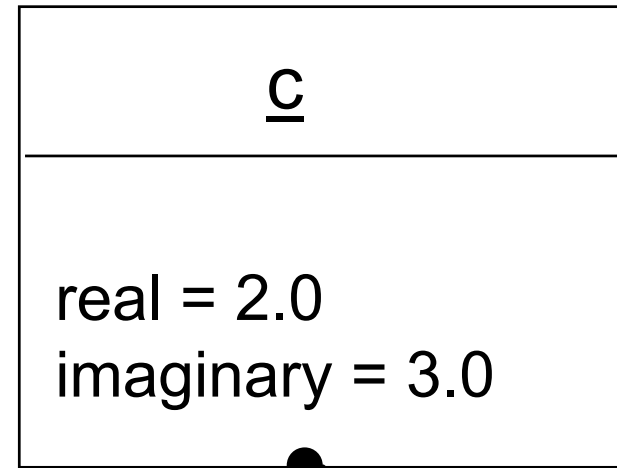
- We can have *anonymous instances* (only the class name is known)
- We can model collections of objects as *multiobjects* (collections of anonymous instances)



Depicting Attribute Values



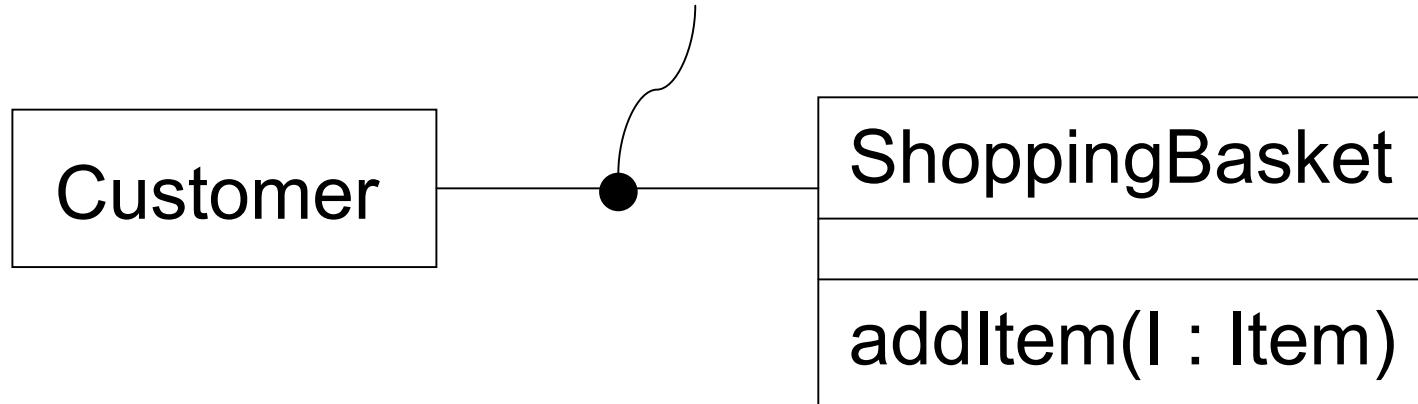
attribute type shown



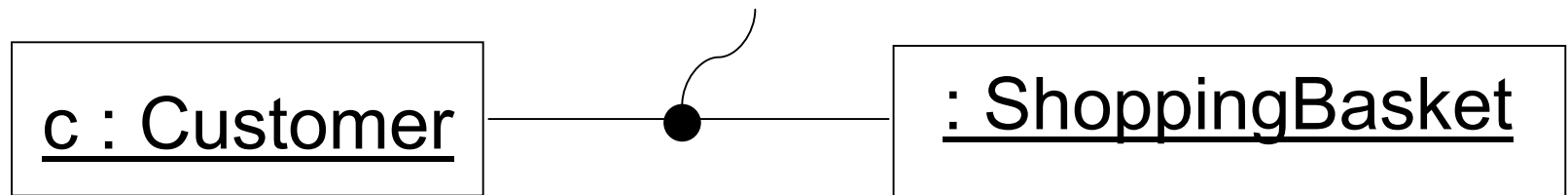
attribute type elided

Associations and Links

A class diagram depicts associations between classes



In the corresponding object diagram, a link is an instance of the association

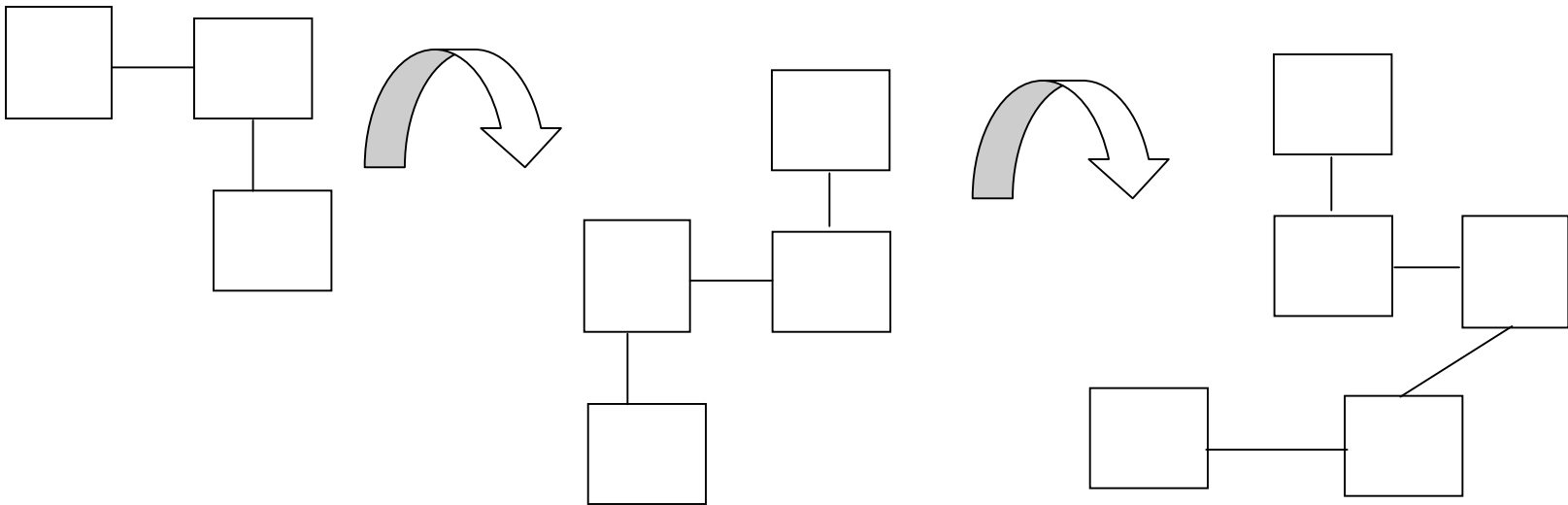


The Roles of Class Diagrams and Object Diagrams

- Class diagrams model the static view of a system (i.e., the structural aspects)
 - they don't indicate what objects exist at any particular instant
- Object diagrams also model the static view of a system
 - they provide a snapshot of the system, showing the objects that exist at an instant in time, their state (the values of their attributes) and the links between the objects

Object Diagrams for Modelling Behaviour?

- A series of object diagrams can help us visualize a system's structural dynamics over time...

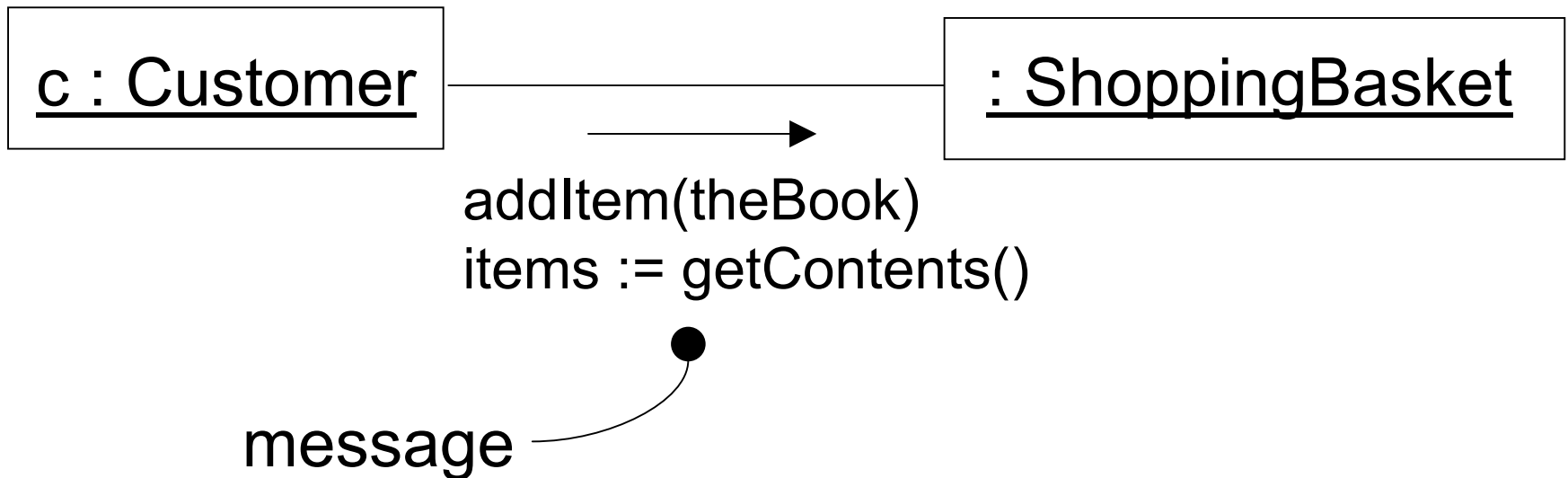


Object Diagrams for Modelling Behaviour?

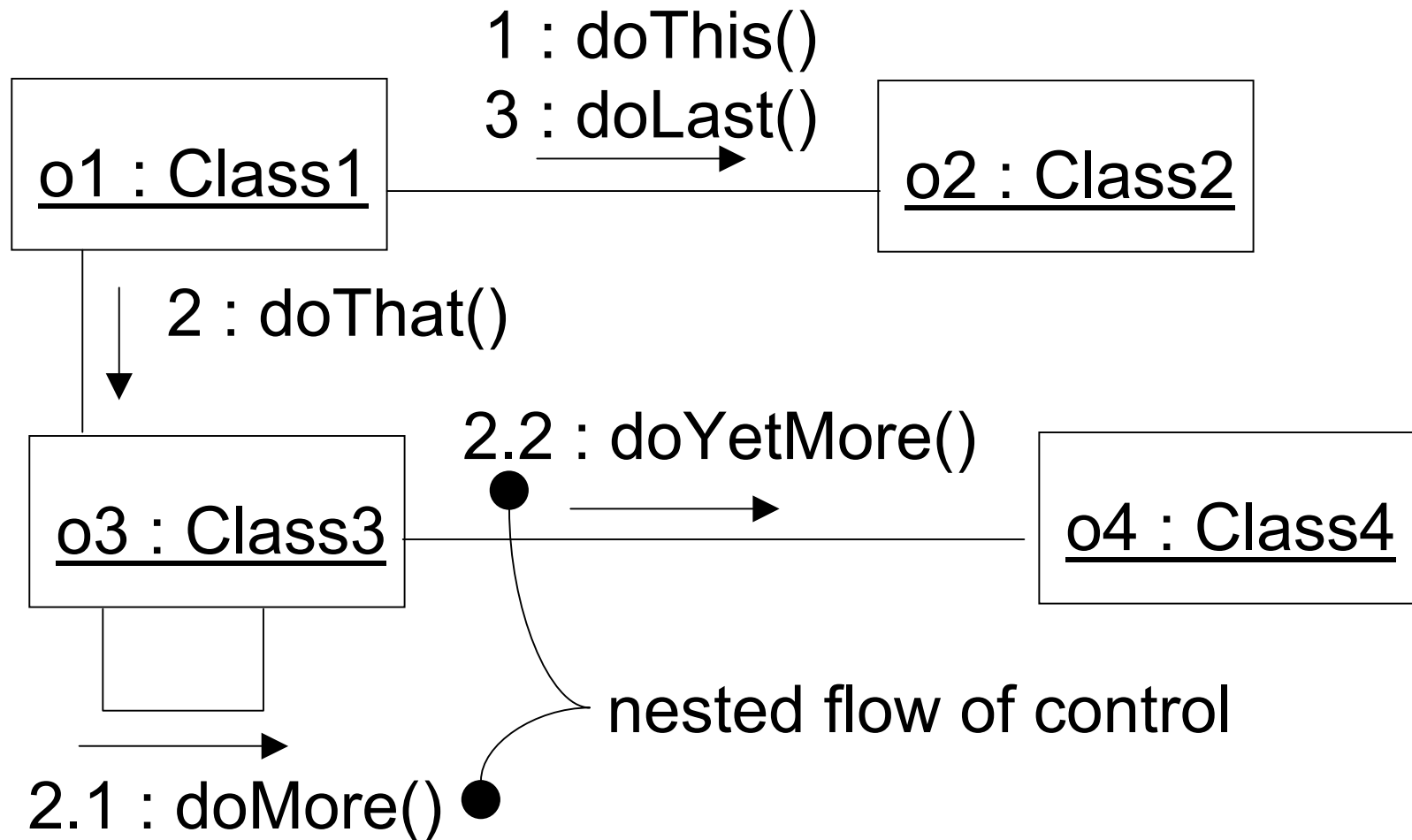
- ...but, they don't indicate the interactions between objects; e.g., the messages that are sent between objects and the order in which they are sent
- The UML provides two types of diagram for modelling interactions: collaboration diagrams (described next) and sequence diagrams (not covered in this set of slides)

Collaboration Diagram

- Messages are sent between objects along links
- Procedural flow of control is rendered by an arrow with a filled solid arrowhead

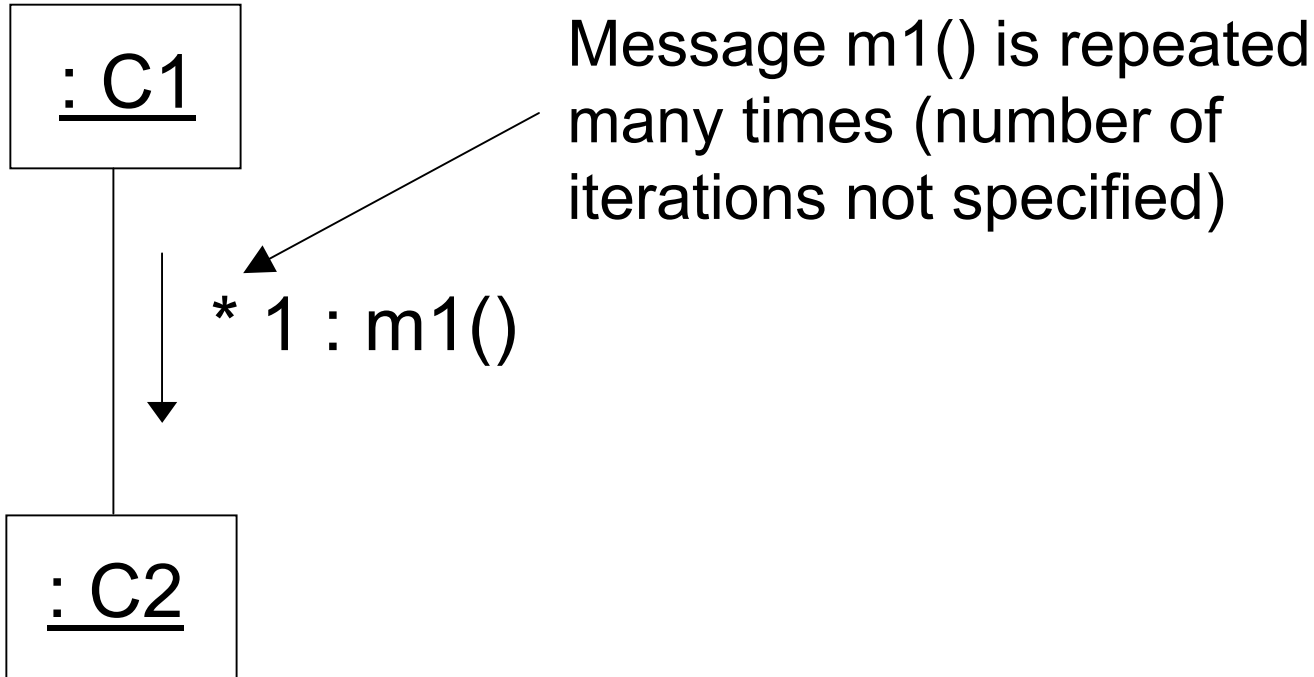


Message Sequencing in Collaboration Diagrams

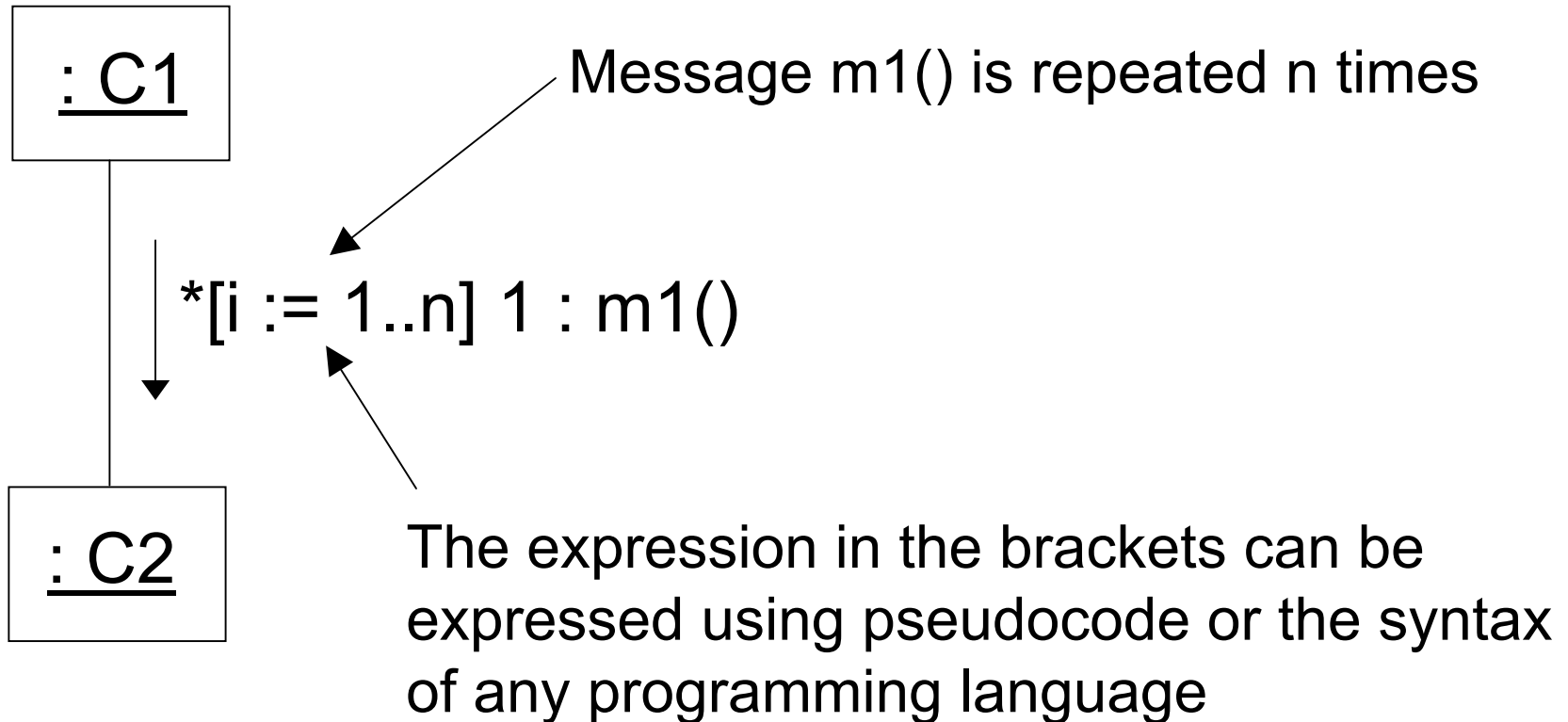


Modeling Iteration

- To model iteration, prefix the message sequence number with an iteration expression



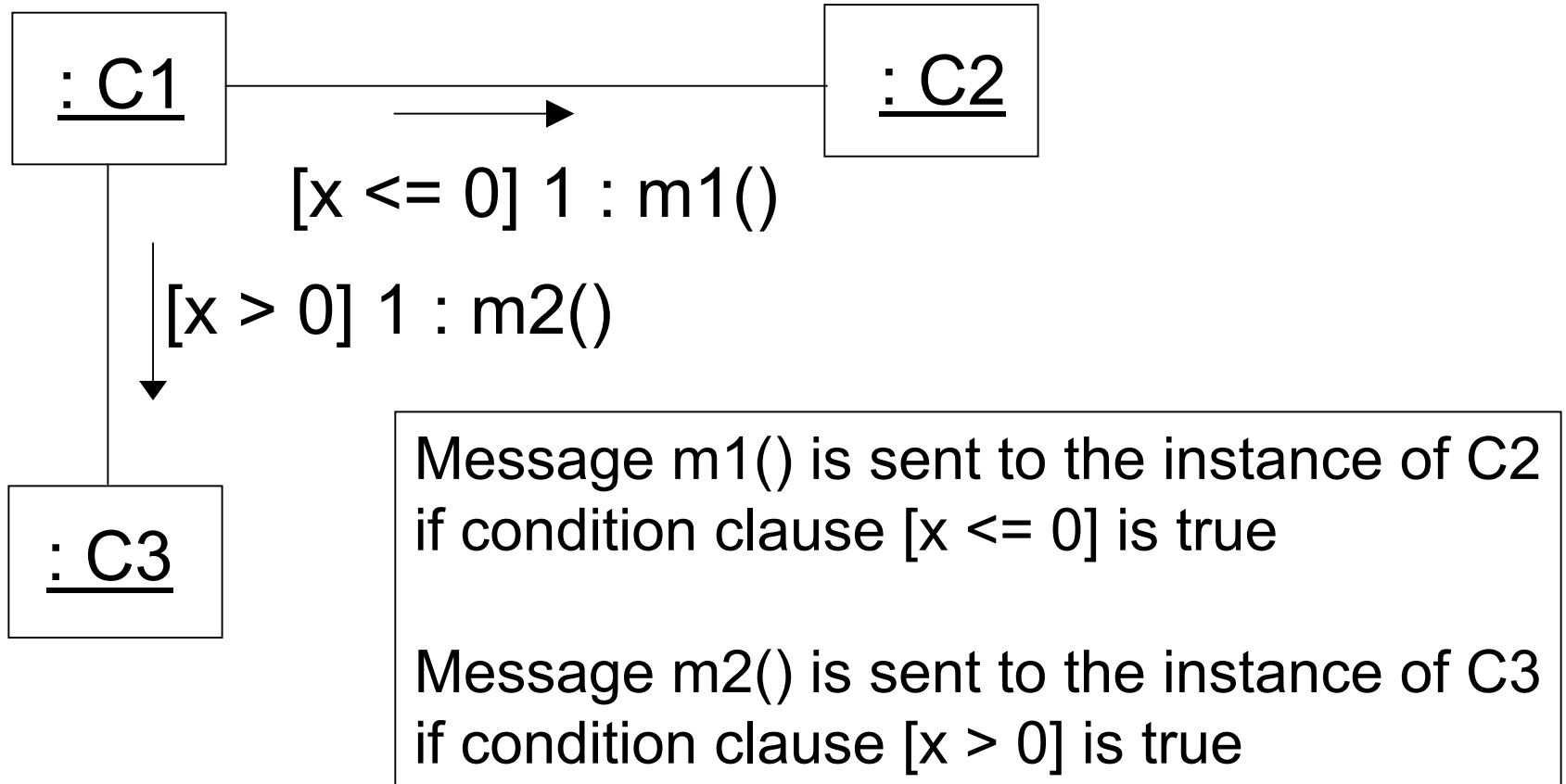
Modeling Iteration



Modeling Branching

- To model branching, prefix the message sequence number with a condition clause such as $[x > 0]$
- The message is sent only if the condition evaluates to true
- Alternate paths of a branch have the same sequence number but are distinguished by a nonoverlapping condition; e.g., $[x = 0]$, $[x < 0]$
- The expression in the brackets can be expressed using pseudocode or the syntax of any programming language

Modeling Branching



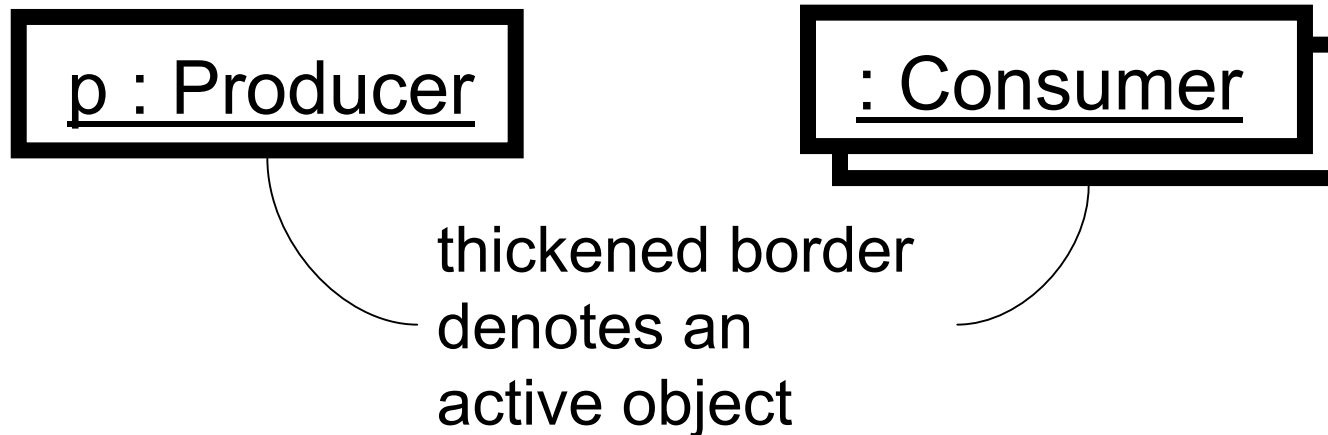
Limitations of Collaboration Diagrams

- Even for a single-threaded system, complicated sequences of iteration or branching can be difficult to represent in a single diagram
 - splitting the diagram into several diagrams may help
- For multithreaded systems, collaboration diagrams provide few visual clues as to where/when threads will block (e.g., by waiting in a synchronized method)
 - fortunately, the UML provides other techniques for modelling behaviour

UML Notation for Concurrency

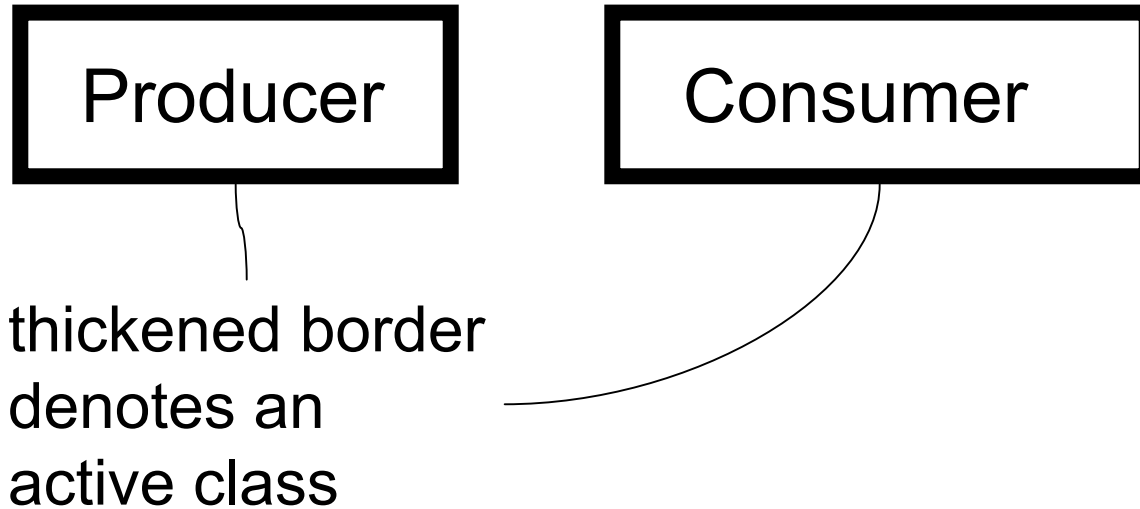
- The UML provides standard notation for modelling
 - active objects
 - active classes
 - object interaction
 - synchronization properties

Active Objects



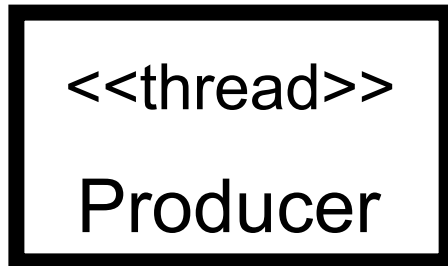
- An active object has an independent flow of control
- Active objects are instances of active classes

Active Classes



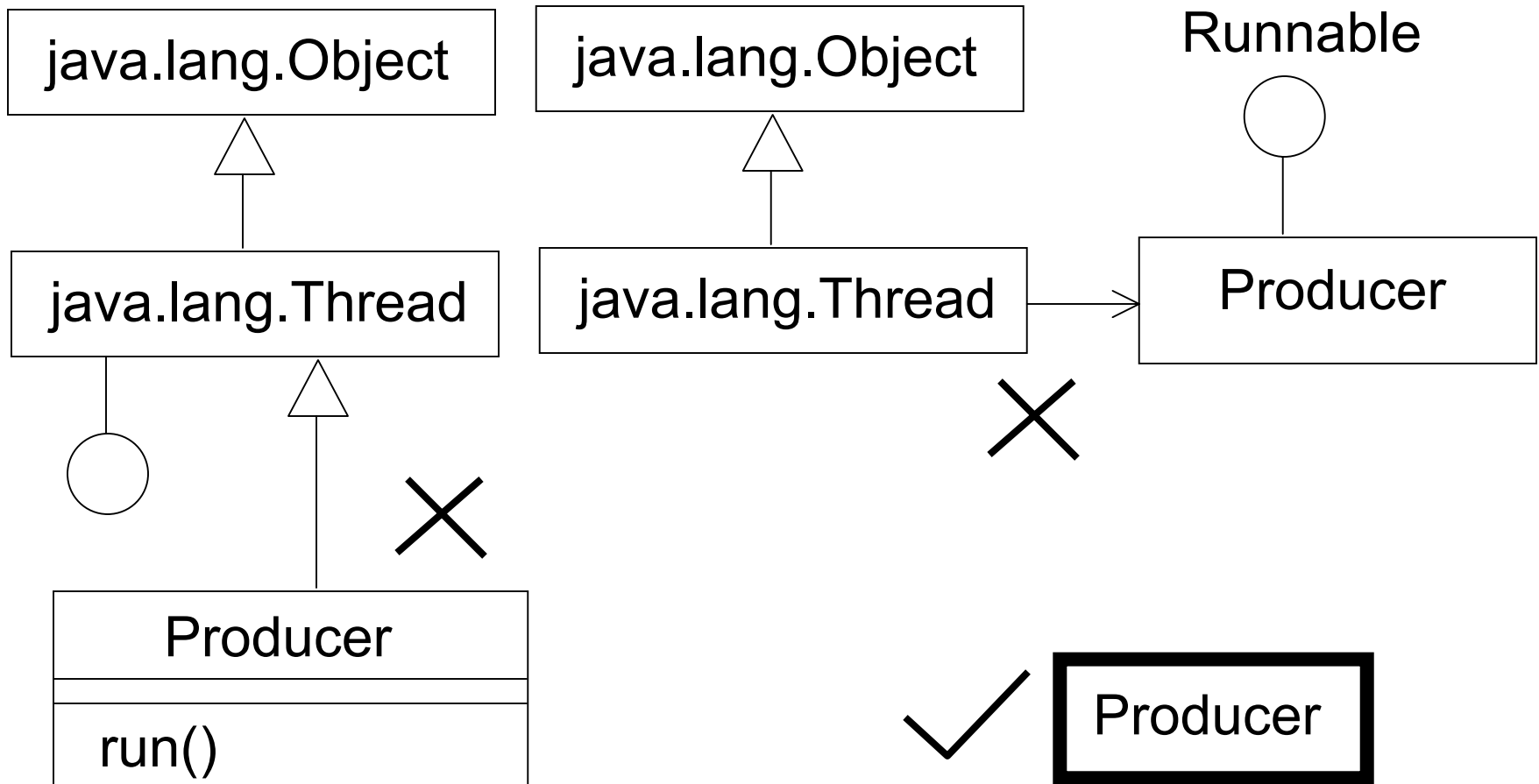
- Active classes can be modelled using the same UML mechanisms that are used with all classes

Stereotypes for Active Classes



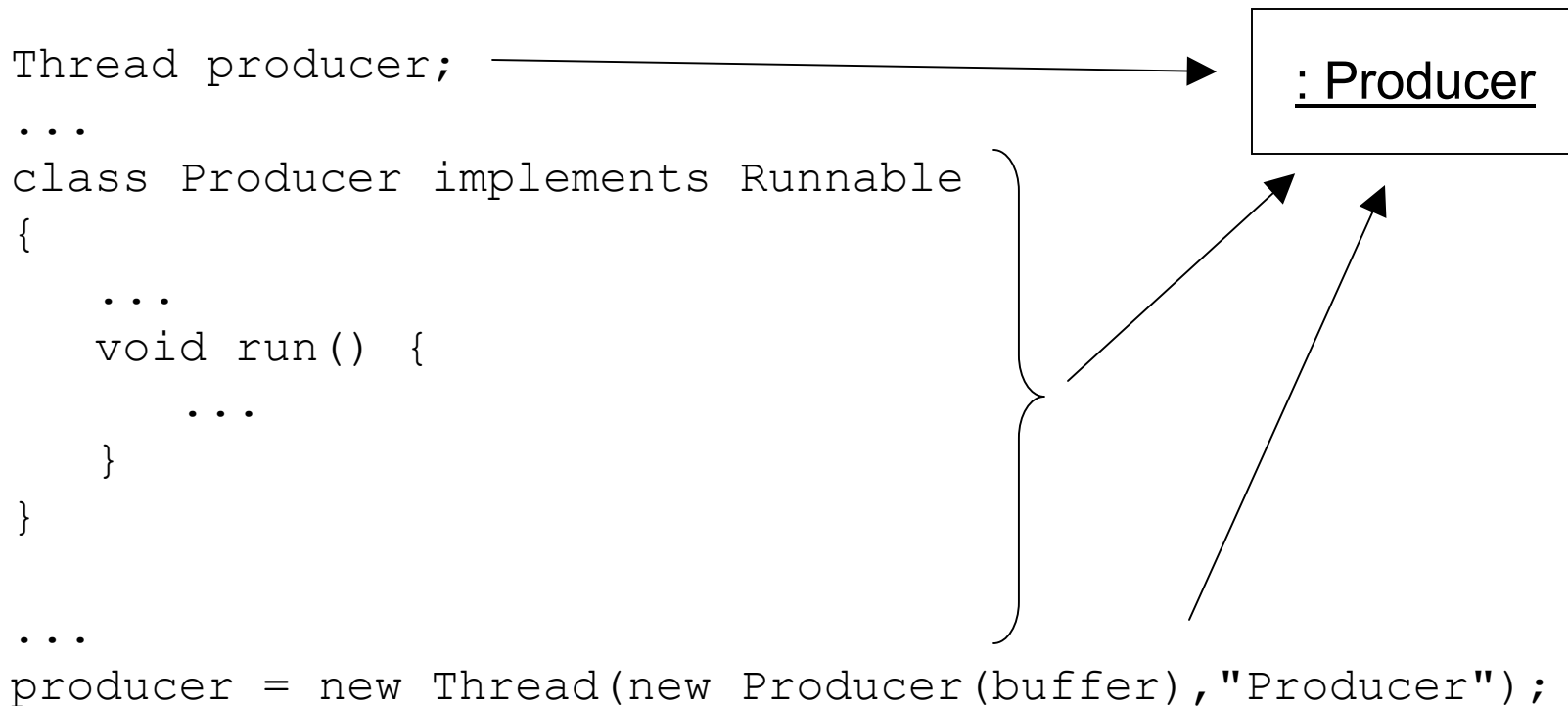
- A process is a heavyweight flow of control
 - executes concurrently with other processes
- A thread is a lightweight flow of control
 - executes concurrently with other threads contained in the same process

Avoid Cluttering Diagrams with Implementation Details



Icons Represent Concepts

- Icons represent concepts of things that exist only at run-time, not literal pieces of code

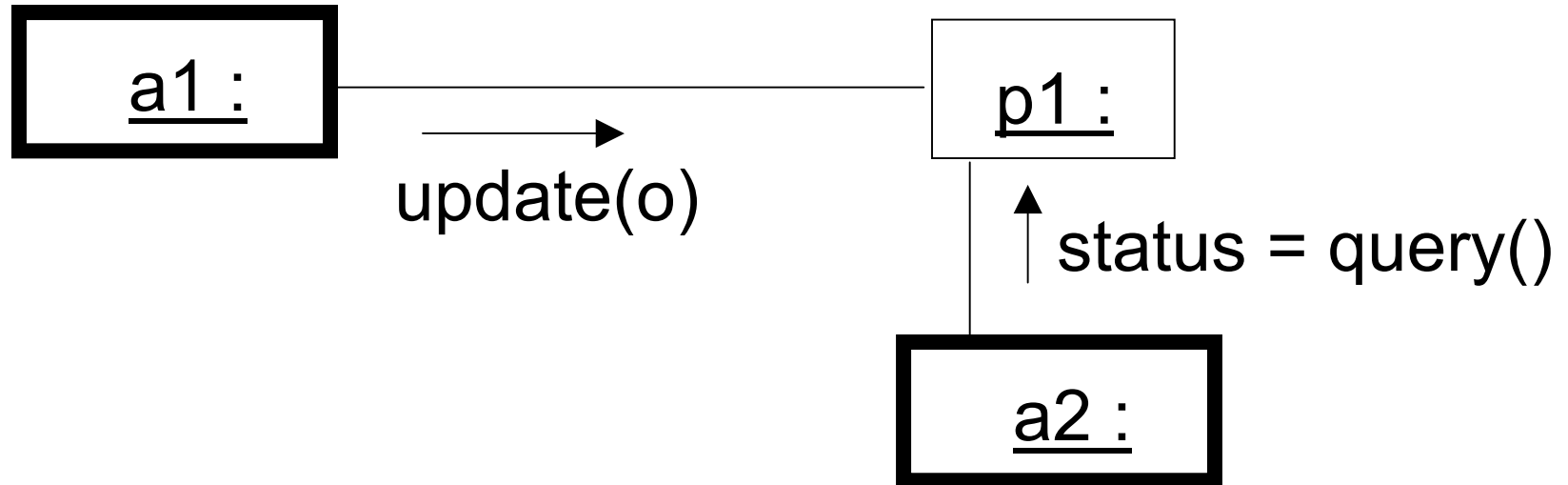


Message Passing Between Passive Objects



- This corresponds to invoking an operation (e.g., simple call/return method invocation)
- One flow of control through the two objects
 - update() is executed in the same context as the method in p1 that invokes it
- Implicitly, there is an active object "upstream" from p1 where the flow of control originates

Message Passing From Active to Passive Objects



- This corresponds to invoking an operation (e.g., simple call/return method invocation)
- Must consider the possibility of multiple flows of control executing in p1 concurrently; i.e., model the synchronization of the flows of control

Message Passing Between Active Objects



- The arrow denotes synchronous message passing between active objects (rendezvous semantics)
- Although this notation is useful when modelling concurrent systems whose threads/processes/tasks communicate by synchronous message passing, this form of communication is not directly supported by Java's threading model, so we won't consider it further

Message Sequencing

- When modelling concurrent systems, we must identify the flow of control (process or thread) that sends a particular message
- We do this by prefixing the message's sequence number with the name of the thread or process in which the sequence is rooted

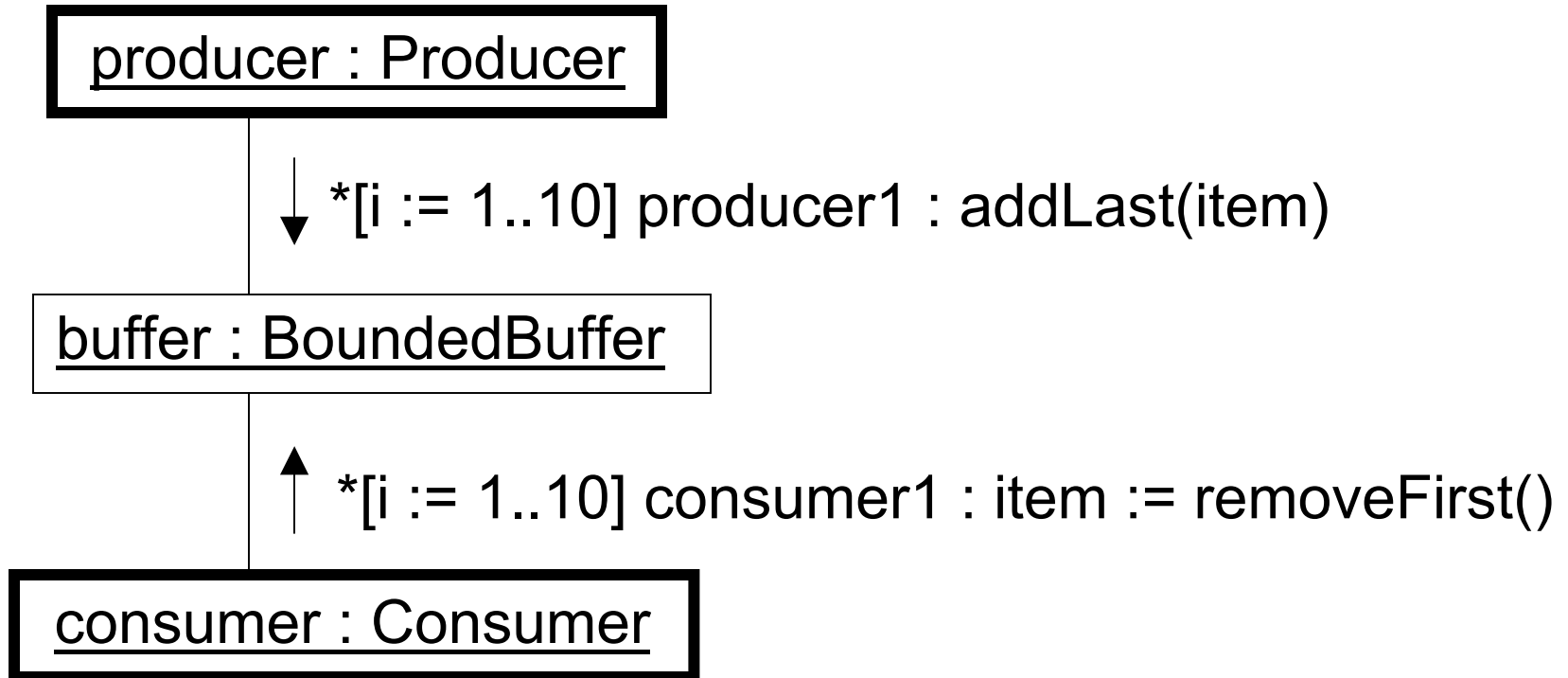
- **Examples:**

`producer1 : addLast(item)`

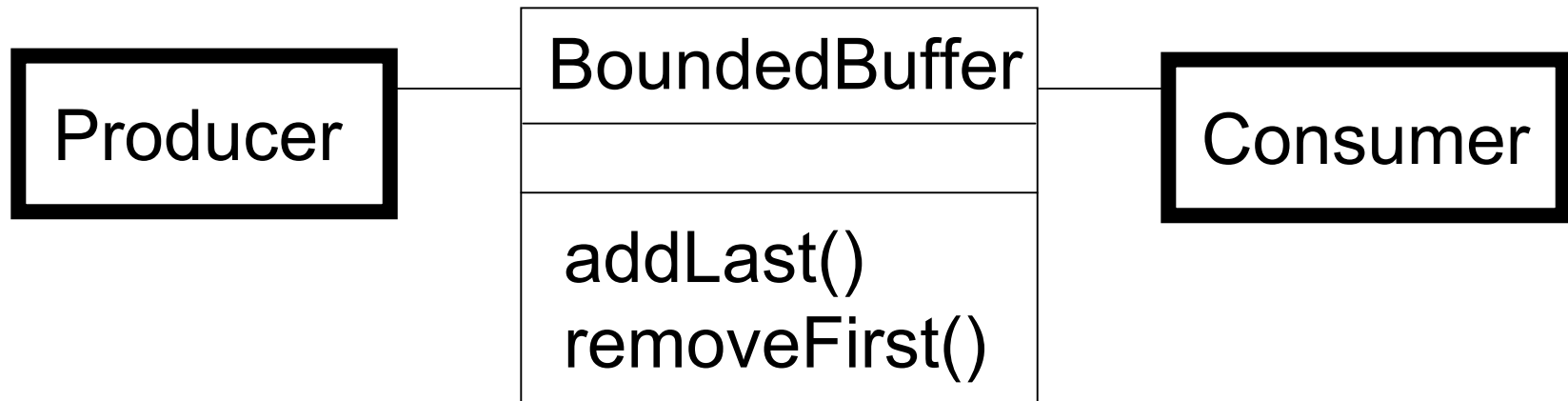
`consumer1 : item := removeFirst()`

- identifies the 1st messages in the sequences sent by the producer and consumer threads

Producer/Consumer/Bounded Buffer Collaboration Diagram



Synchronization Issues



- Classic mutex problem exists if multiple threads of control are simultaneously inside a passive object
 - interference may result if one thread invokes `addLast()` while another executes `removeFirst()` (or if two threads execute either method simultaneously), corrupting the object

Specifying Synchronization of Flows of Control

- The UML provides three ways of modelling the *synchronization properties* of operations defined in a class
- The quotations on the following slides are from:
"The Unified Modeling Language User Guide",
Grady Booch, James Rumbaugh, Ivar Jacobson,
Addison Wesley, 1999, pp. 129-130

Sequential Synchronization Property

- `{sequential}` - "Callers must coordinate outside the object so that only one flow is in the object at a time. In the presence of multiple flows of control, the semantics and integrity of the object cannot be guaranteed."
- Usually assumed to be the default property of operations, and if so, will not be rendered in class diagrams

Sequential Synchronization Property

- Mapping to Java: operations with the `{sequential}` synchronization property correspond to
 - those methods in conditionally thread-safe classes that must be protected by external synchronization
 - methods in thread-compatible classes

Guarded Synchronization Property

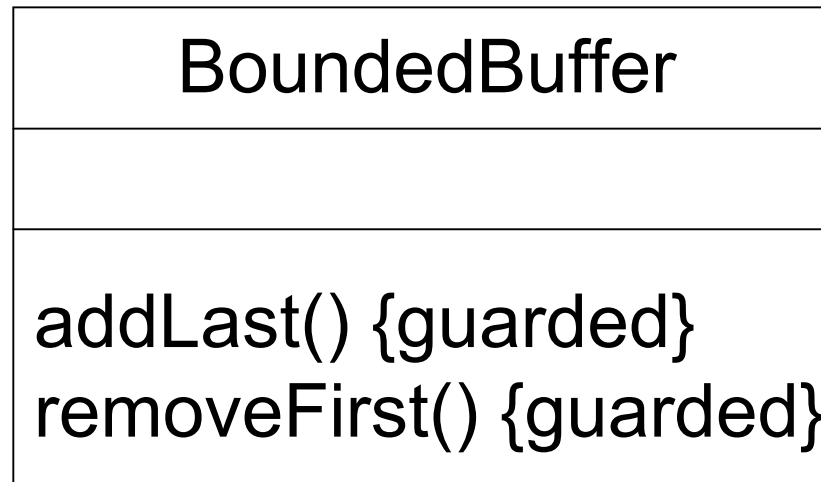
- `{guarded}` - "The semantics and integrity of the object is guaranteed in the presence of multiple flows of control by sequentializing all calls to all of the object's guarded operations. In effect, exactly one operation at a time can be invoked on the object, reducing this to sequential semantics."

Guarded Synchronization Property

- Mapping to Java: operations with the `{guarded}` synchronization property correspond to
 - methods in thread-safe classes
 - methods in conditionally thread-safe classes that do not require external synchronization

Rendering Synchronization Properties

- Synchronization properties attached to operations can be rendered in the UML by using constraint notation



Concurrent Synchronization Property

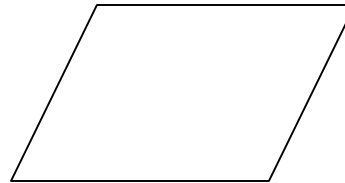
- `{ concurrent }` - "The semantics and integrity of the object is guaranteed in the presence of multiple flows of control by treating the operation as atomic. Multiple calls from concurrent flows of control may occur simultaneously to one object on any concurrent operation, and all may proceed concurrently with correct semantics; concurrent operations must be designed so that they perform correctly in the case of a concurrent sequential or guarded operation on the same object."

Concurrent Synchronization Property

- Mapping to Java: operations with the `{ concurrent }` synchronization property correspond to
 - methods in immutable classes
 - methods that modify the state of the object, but are designed to work even if the same method is executed simultaneously by another thread of control, or even if another concurrent, guarded, or sequential method is executed simultaneously by another thread of control
 - usually safer to implement these as guarded operations

Stereotypes for Modelling Concurrency

active object



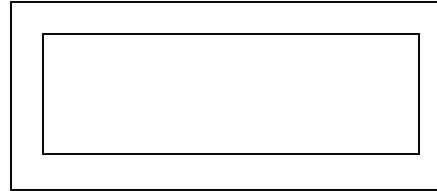
- Visually more distinct than a rectangle with a thickened border
- Easy to remember (parallelogram == parallelism)
 - Historical note: proposed by Buhr (Buhr diagrams in *System Design with Ada* (1984), Machine Charts in *Practical Visual Techniques in System Design* (1989)), and Booch ("Boochgrams" in *Software Components with Ada* (1987) and *Object-Oriented Design with Applications* (1991), early UML specs.)

Stereotypes for Modelling Concurrency

- Normally, object diagrams/collaboration diagrams don't indicate the synchronization properties of the object's operations
- We have to look at the class diagram to find this information
- Example:
 - in the class diagram, `BoundedBuffer` operations are rendered with the `{guarded}` constraint
 - the collaboration diagram for the producer/consumer/bounded buffer system does not depict these synchronization properties

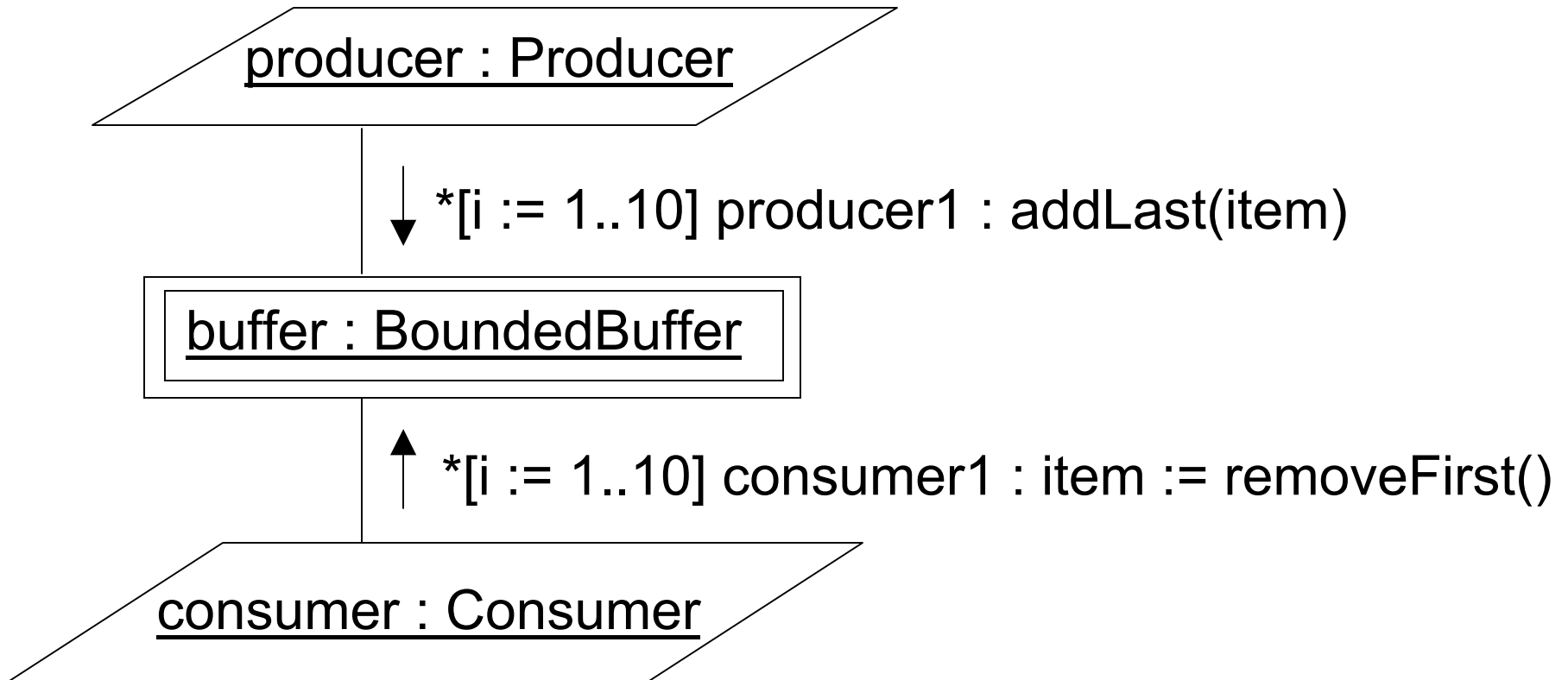
Stereotypes for Modelling Concurrency

monitor object



- A *monitor object* is one whose operations enforce mutual exclusion on the object (i.e., operations are guarded)
- If the monitor object has a private object that is used to enforce mutual exclusion (e.g., a mutex or a semaphore), the monitor object stereotype eliminates the need to draw that object (removes visual clutter)

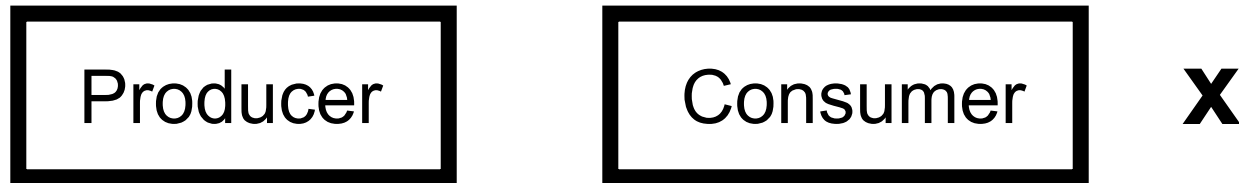
Revised Collaboration Diagram



Evolution of UML Notation

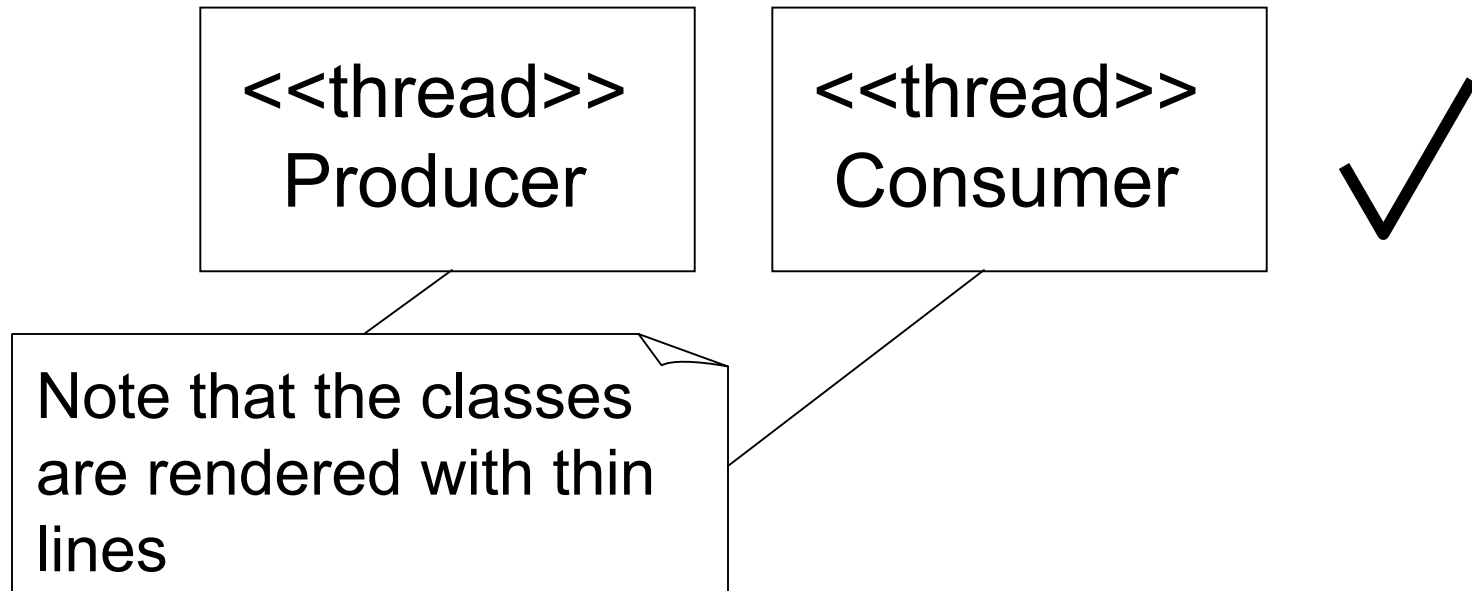
- Many books that use the UML (e.g., *The Unified Modeling Language User Guide*) are based on UML v1.3 (or earlier versions)
- There have been a number of changes between the Booch, Rumbaugh, Jacobson book and the UML v1.4 that are particularly relevant to this course

Active Classes



- The notation for active classes (a rectangle with thick lines) appears to no longer be part of the UML
- In fact, the term active class is not formally defined in the specification
 - if an instance of a class has its own thread of control "such a class is informally called an *active class*" (UML v1.4 Spec., Section 2.5.2.9)

Concurrency Stereotypes

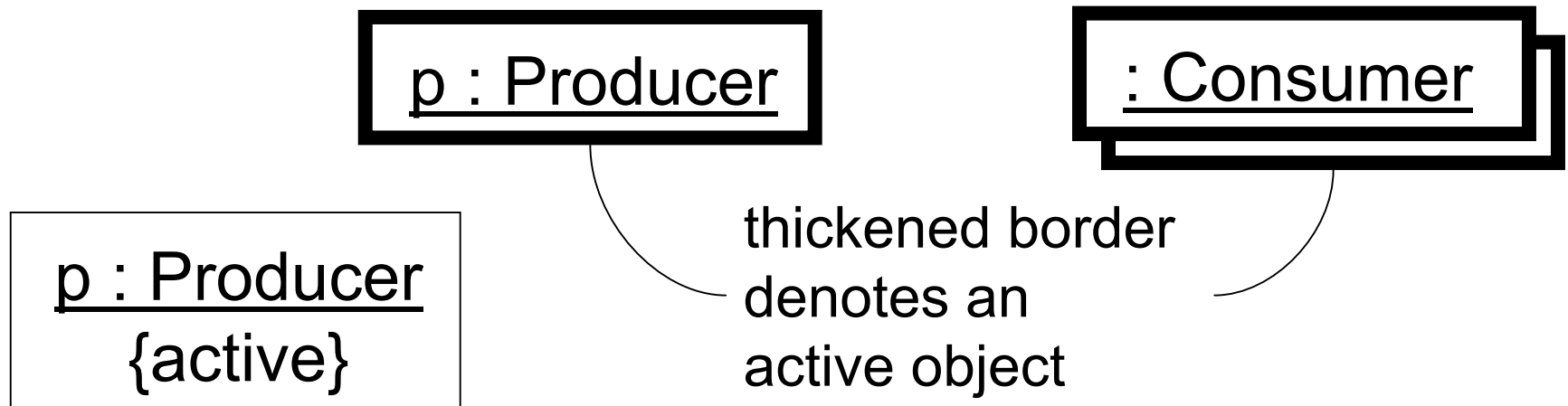


- Classes whose instances have their own threads of control are rendered with the <<thread>> and <<process>> stereotypes (UML v1.4 Spec., Section 2.5.2.10)

Synchronization Properties for Operations

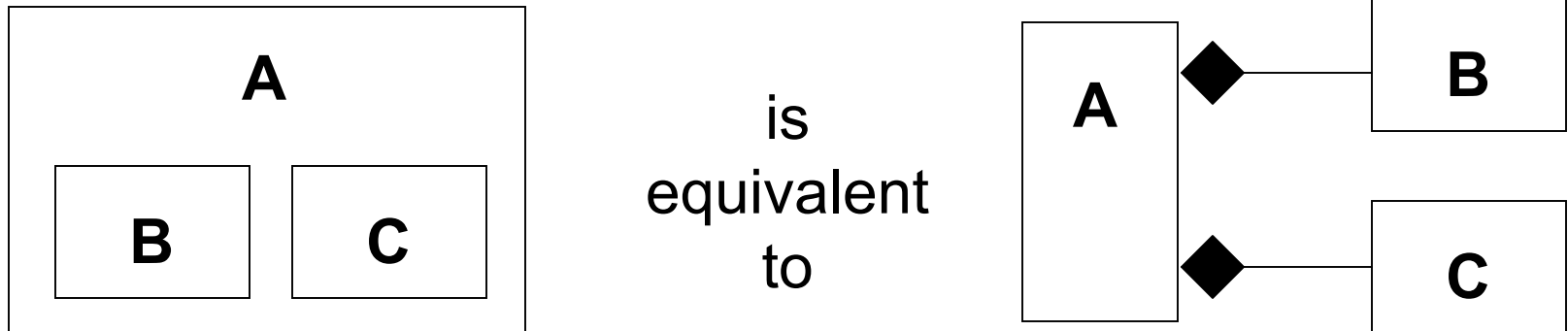
- The definitions of the `{sequential}`, `{guarded}`, and `{concurrent}` properties in the UML spec. are worded differently than the definitions in the User Guide; however, the definitions we quoted from the User Guide are consistent with the spec. (UML v1.4 Spec., Section 2.7.2.6)

Active Objects



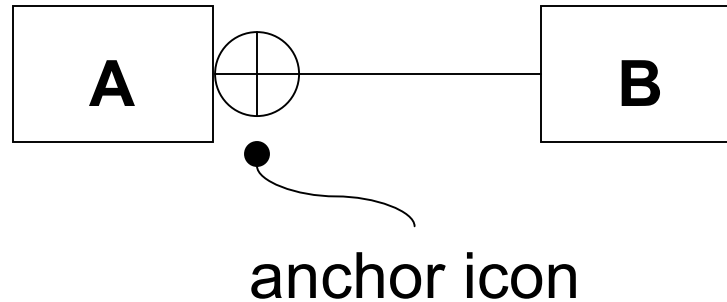
- An active object is rendered by drawing an object with a thickened border
- An active object can also be rendered using the `{active}` property
- Ref.: UML v1.4 spec., Section 3.71

Composition



- Drawing classes inside a class is a visual shorthand for rendering composition (“has-a” relationship)
- The left-hand diagram does not imply that, when programmed, classes B and C are nested inside class A
- The next slide describes new UML notation for nesting (not a concurrency issue, but of interest)

Nested Class Declarations



- The anchor line connecting class *A* and class *B* indicates that *B* is declared within *A*
- The anchor icon (a cross inside a circle) is placed beside the declaring class
- This construct is primarily used for implementation reasons and for information hiding
- Ref: UML v1.4 Spec., Section 3.27

Nested Class Declarations

- In Java, B could be
 - a nested top-level class or interface (i.e., a `static` class or an interface defined inside a top-level class)
 - any of the three types of inner classes (there is no such thing as an inner interface); i.e.,
 - a member class (a class defined as a member of an enclosing class, but defined without the `static` modifier)
 - a local class
 - an anonymous class