Comp-304 : Collaboration Diagrams
Lecture 11

Alexandre Denault
Original notes by Hans Vangheluwe
Computer Science
McGill University
Fall 2006

Dynamic

- Class diagrams are composed of static entities.
  - Classes, packages, etc
- Interaction diagrams describe the behavior of an application.
  - They are dynamic in nature.
  - Thus, they are composed of dynamic entities : objects.
- We will focus on two kinds of interaction diagrams :
  - Collaboration Diagrams
  - Sequence Diagrams

Behavior

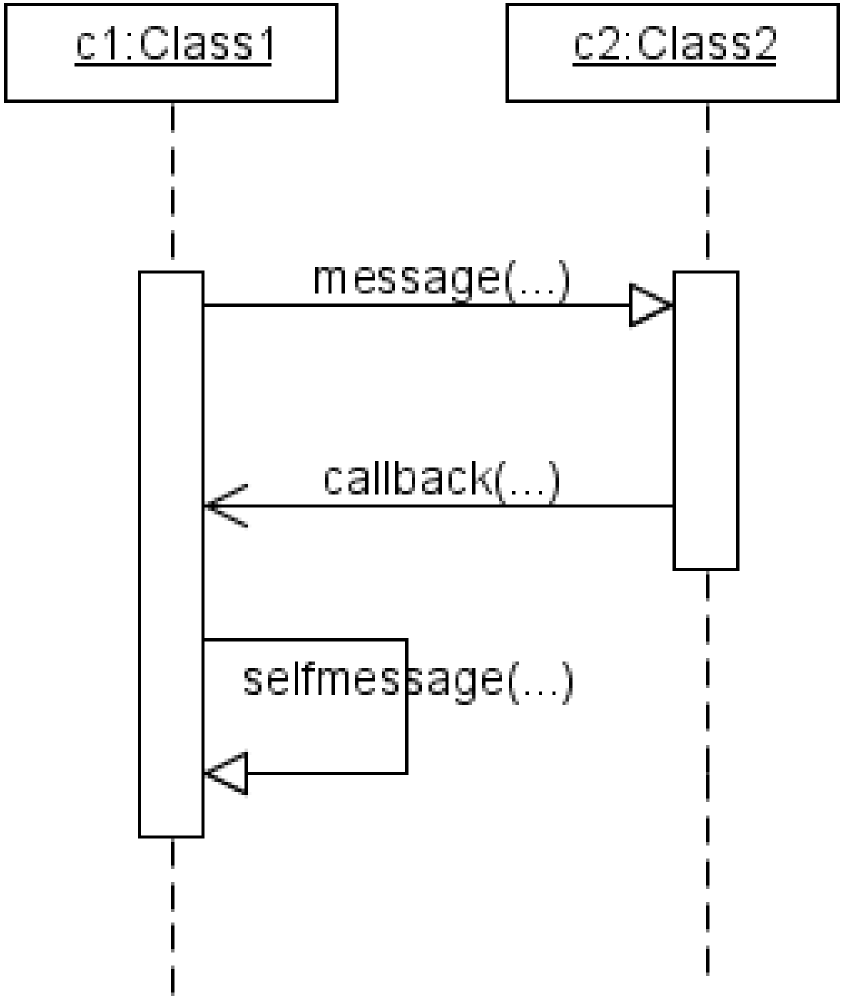
- Object Interaction diagrams depict dynamic, run-time behavior
  - communication between objects via messages
  - sequence of transactions in a dialog between a user and a system
  - one trace of behavior is ideally one use case
- With interaction diagram, we introduce the notion of time.

Collaboration Diagrams

- Collaboration diagrams represent objects in a system an their associations.
- They are composed of three elements:
  - Objects
  - Associations
  - Messages

Sequence Diagrams

- Sequence diagrams illustrate the sequence of actions that occur in a system.
- They are composed of 2 elements
  - Object
  - Messages



Sequence VS Collaboration

- Both diagrams are illustrate interaction.
  - Sequence is used to illustrate temporal interactions.
  - Collaboration is better suited to display the association between the objects.
- Given enough information, a sequence diagram can be converted into a collaboration diagrams (and vice-versa).

Use Case

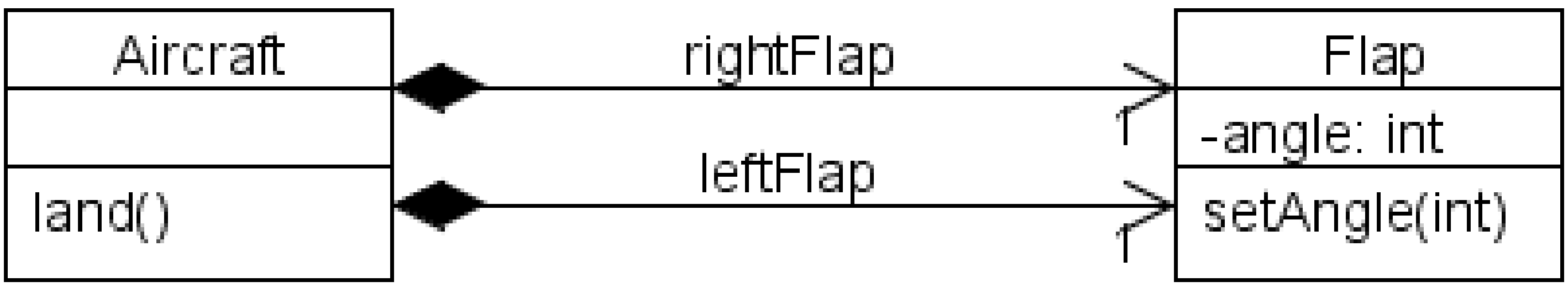
- Use case is a technique for capturing functional requirements of systems and systems-of-systems.
- Each use case focuses on describing how to achieve a goal or task.
- For software, multiple use cases are often necessary to fully describe the functionality of that software.

Trace of Behavior

- Imagine you go to the ATM
  - System : waiting for card
  - User: insert card
  - System : ask for pin
  - User: enter pin
  - System : verify pin
  - ...
- Once you have traversed a use case, you can figure out how many objects are created and what messages are passed between them

Setting up our example

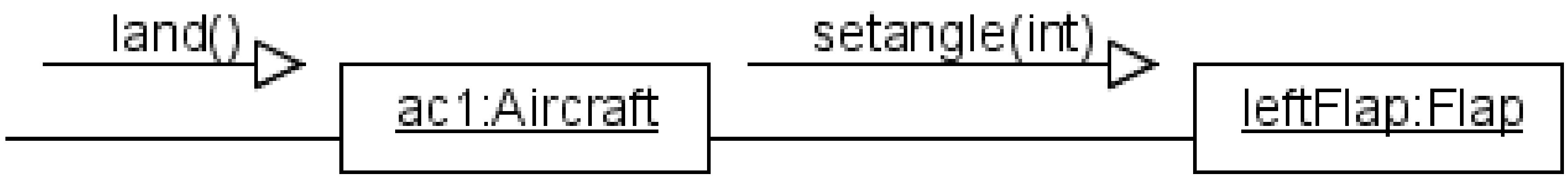
- Consider the following class diagram.



- Suppose some external call to an instance of Aircraft executes land(), a public method.
- In turn, land() executes setangle() of some instance of Flap.

Collaboration Diagram

- Remember, we are depicting the interaction between instances, not classes.
- ac1 has a reference to a Flap named leftFlap.
- In the code of the method land(), there is a call leftFlap.setangle(int)



A few things to note

- To depict a message, we draw a small arrow from the sender object to the target object, this shows the direction of communication
- With the arrow is the operation name we desire to execute, along with all arguments
- The arrow is parallel to a line, which depicts there is a link between the objects (<u>usually</u> by a class association, but not necessarily)

No association?

- If objects aren't linked by association, then how could they be linked?
- Suppose o1 sent o2 a reference to itself. So, o2 may refer to o1 (via the reference) even though there is no association between the classes of o1 and o2.
- This is known as a dynamic reference.
- This reference also allows a target object to “callback” a sender object.

Callback

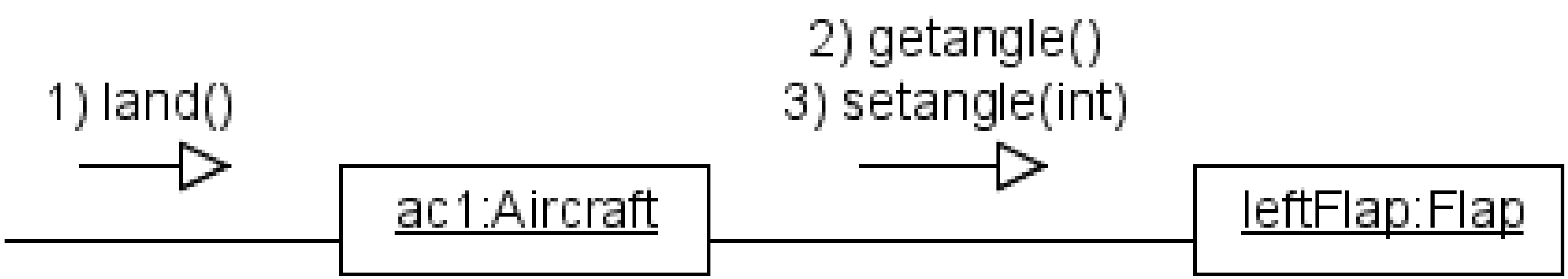
- The more formal description of callback is executable code that is passed as an argument to other code.
- However, the term callback is also used when a reference is passed to achieve the same thing.
- Callbacks are often used in asynchronous messaging.
- A piece of code or a reference is assigned to do something when a specific event occurs.
    - i.e. Swing and an ActionListener

Original Execution

- The following diagram is exactly the same as the previous diagram, except that it shows which objectName.ClassName.methodName(args) was originally executed.
- Useful if we want to see exactly where the execution started.

Order

- Suppose we wanted to verify the angle first, then set it, how do we depict the order in which the calls should be made.
- We simply need to add numbers to the messages to show the sequence of the calls.

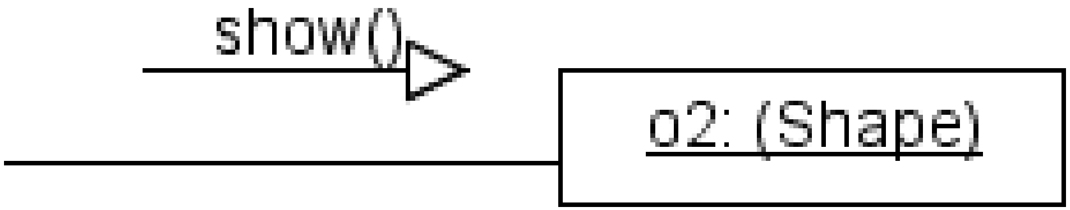


Polymorphism

- Polymorphism is a problem in object interaction.
- Suppose we want to send the message show() to a Shape object.
  - That could be an instance of Triangle, Rectangle or Square at run-time.
  - How do we depict this in a collaboration diagram?
- Usually, we are certain that o1 sends a message to o2
- Also, suppose that Triangle, Rectangle and Square are subclasses of Shape.

Polymorphism (cont.)

- Make the target object's class the lowest class in the inheritance hierarchy that is a superclass of all the classes to which the target object may belong to.
- Put the superclass name in parenthesis to show that it will be evaluated at run-time.
- This is a form of substitutability.

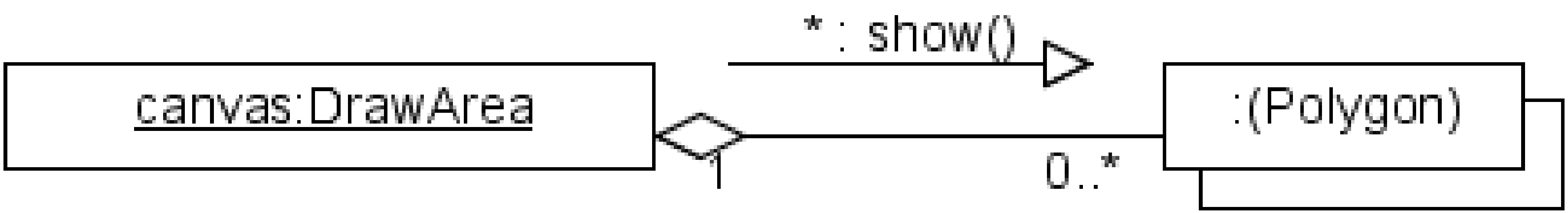


Iterated Messages

- Suppose we have an object DrawArea which has a shapes array of Polygons (Triangles, Rectangles and Squares) that belong to its area.
- We want to repeatedly send the message show() to all the constituent objects (Polygons) of the aggregate object (DrawArea).
- Iterator Pattern (a design pattern) can be used as a traversal method.

Iterated Messages (cont.)

- Notice the aggregate connector.
- show() message is called many times (the *)
- DrawArea may have 0 or more Polygons in its array named shapes.
- Target object is unnamed and double boxed to show multiplicity.



Referring to Self

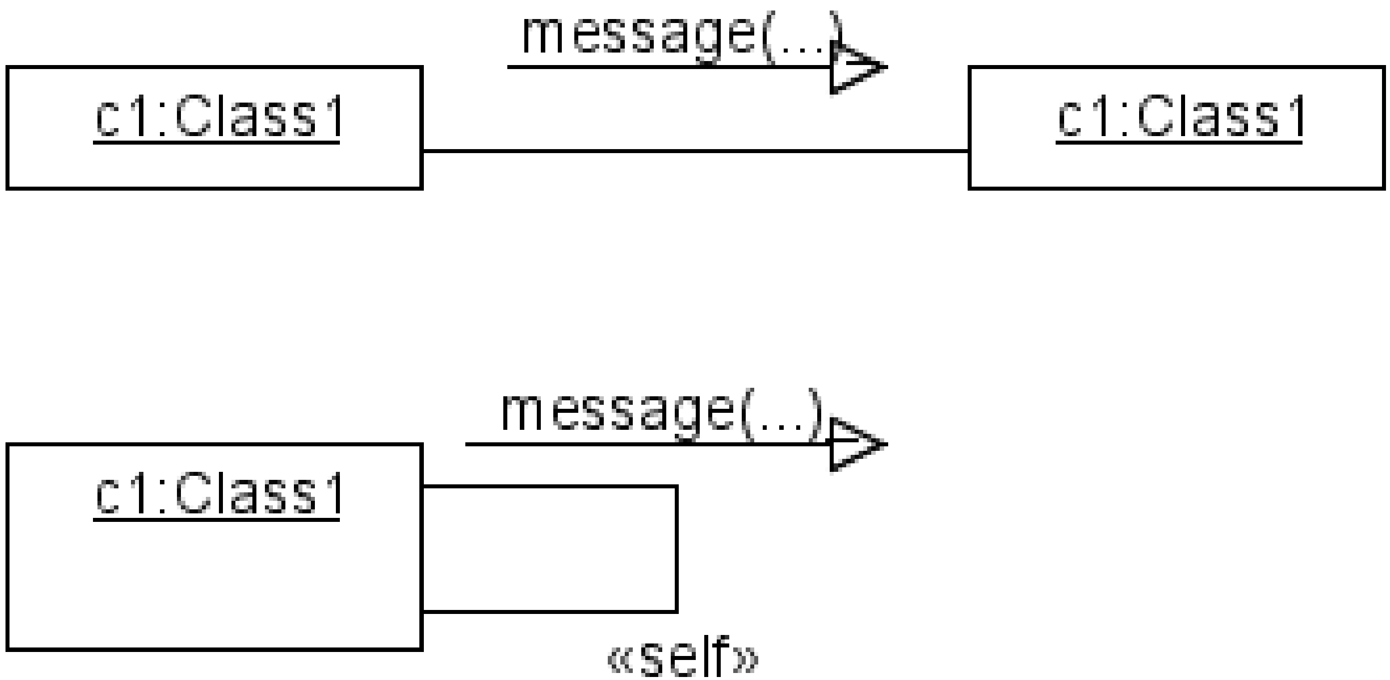
- When an object refers to self, it is referring to its own object handle.
  - In Python, we also use the keyword *self*
  - In Java, C++ and PHP, we use the keyword *this*
  - *In Visual Basic, we use the* keyword *me*
- This is useful to
  - Pass the target object a reference to the sender object (for callbacks)
  - Send a message to itself

Passing a reference to self

- In message, just add self as an argument.

Sending a message to self

- There are two ways to depict this.



Why send a message to self?

- Think of it as implementation / information hiding.
  - We don't want to show how a variable is stored or manipulated.
    - get/set (accessor/mutator) methods
- It may sound weird, but we might want to hide implementation details from methods within the same class (especially if those methods are public).