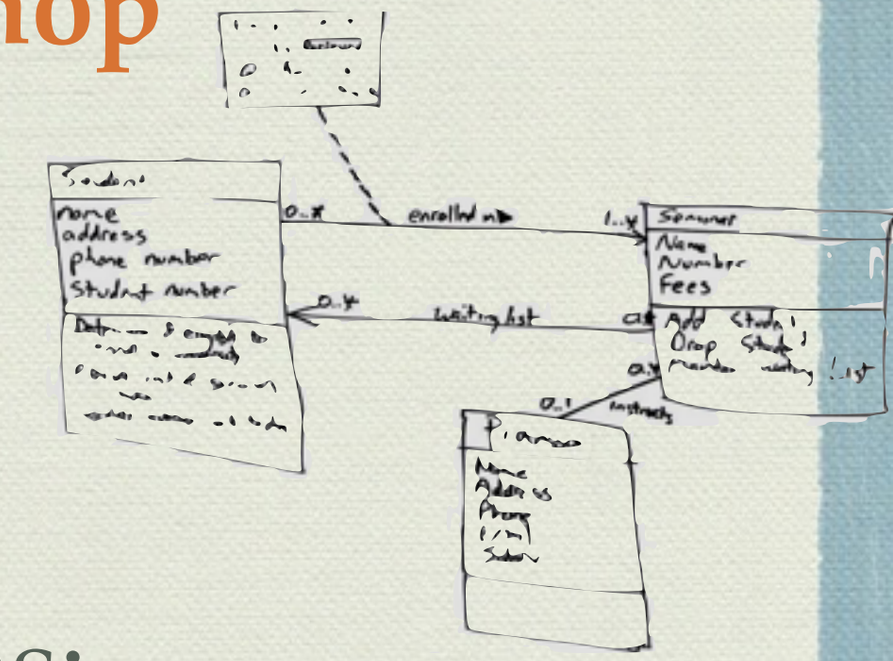
# Software Engineering Workshop




*Workshop 6*

## Modeling Ordered Interactions: Sequence Diagrams

*Slides prepared by Marwah Alaofi*

# Announcement

- Sequence diagram as well as the documentation are due next week.
- Every two/three students would create one sequence diagram.

# Something is Missing!

- Use cases allow your model to describe what your system must be able to do.
- Classes allow your model to describe the different types of parts that make up your system's structure.
- There's one large piece that's missing; with use cases and classes alone, you can't yet model how your system is actually going to do its job.
- This is where interaction diagrams, and specifically sequence diagrams, come into play.

# In today's workshop you will learn ..

- What is sequence diagrams
- Where to use sequence diagrams
- Graphical notations: participants, messages, lifelines and activation bars.

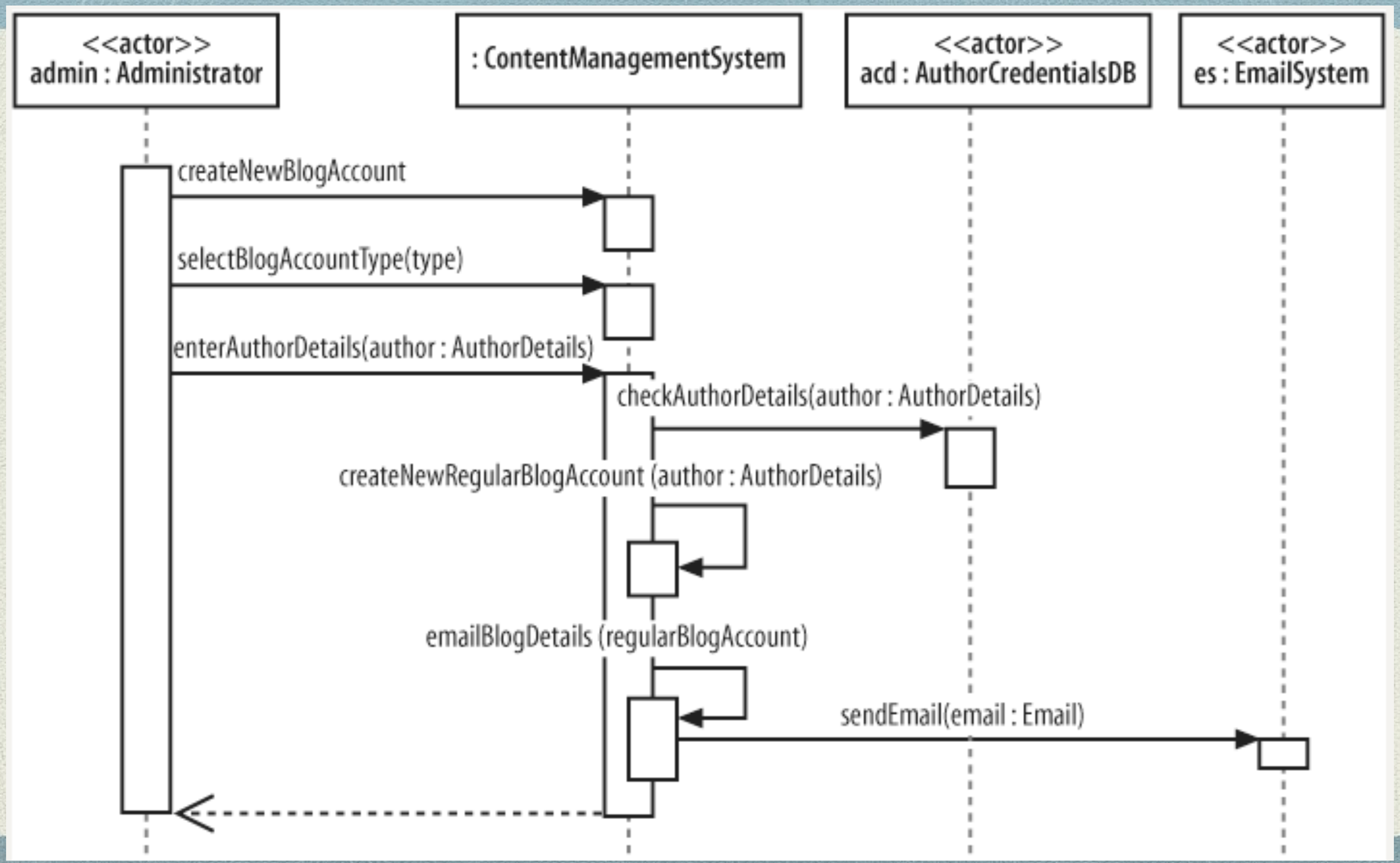
# Sequence Diagrams

- Sequence diagrams are all about capturing the order of interactions between parts of your system.
- Using a sequence diagram, you can describe which interactions will be triggered when a particular use case is executed and in what order those interactions will occur.
- Typically, a sequence diagram captures the behavior of a single scenario. It shows a number of example objects and the messages that are passed between these objects within the use case.

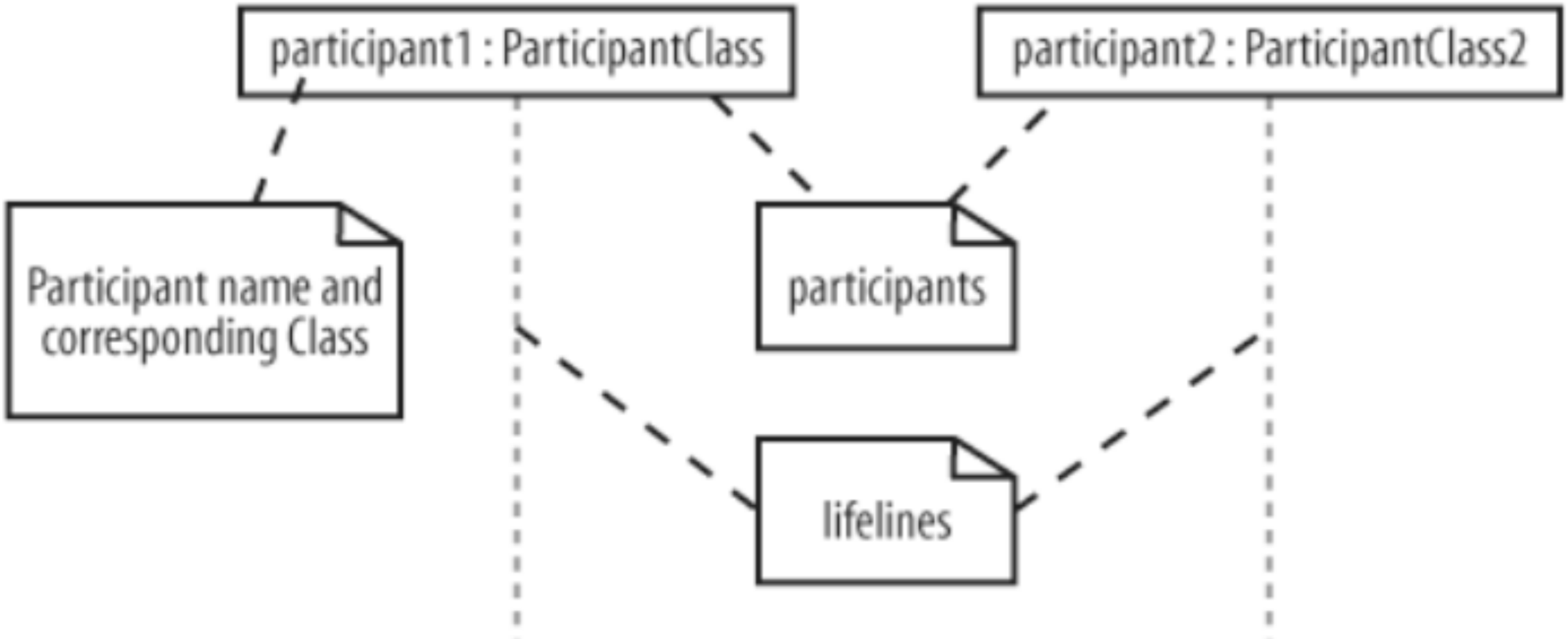
# When to Use Sequence Diagrams

- When you want to look at the behavior of several objects within a single use case.
- Sequence diagrams are good at showing collaborations among the objects.
- If you want to look at the behavior of a single object across many use cases, use a state diagram.
- If you want to look at behavior across many use cases or many threads, consider an activity diagram.

# How it looks like!



# 1. Participants in a Sequence Diagram



# 1.1 Participant Names

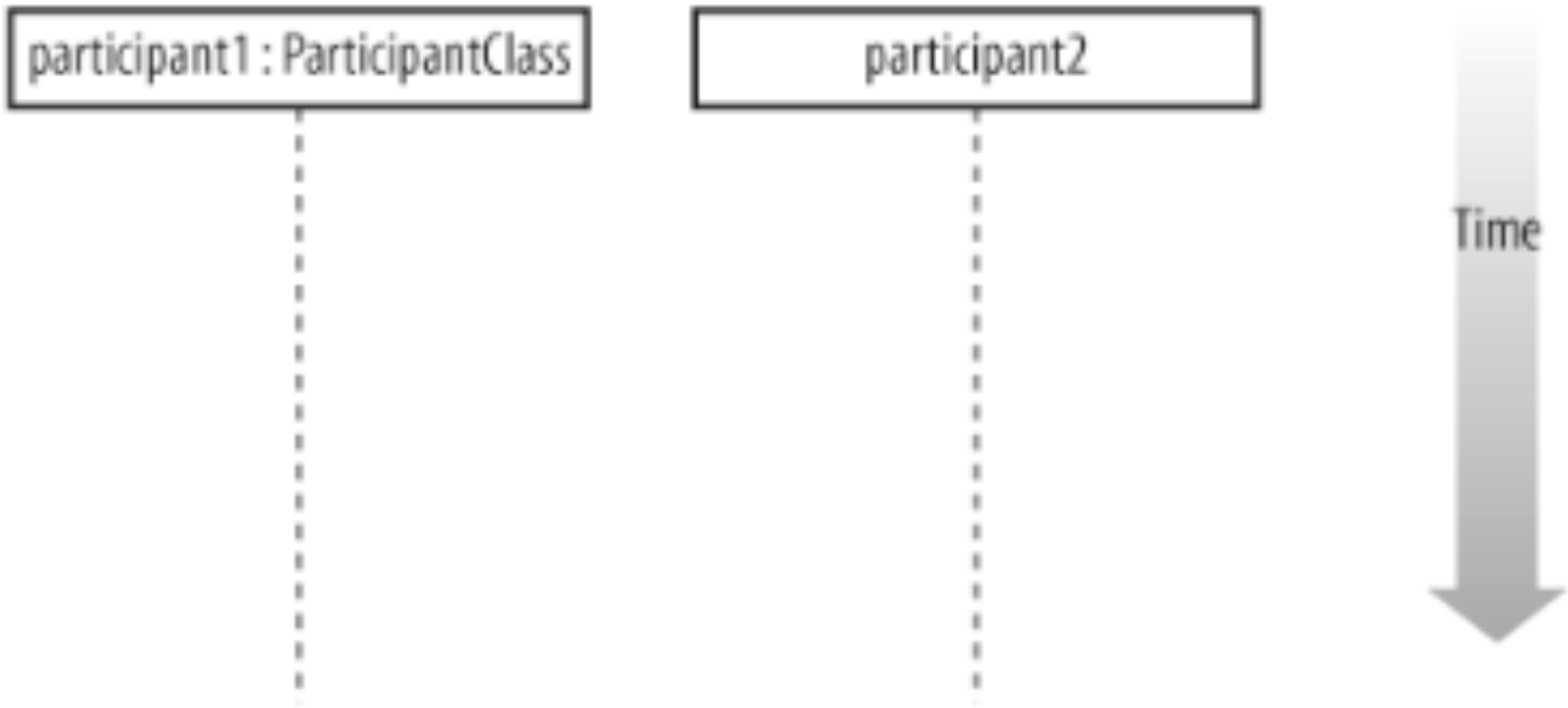
- Participants on a sequence diagram can be named in number of different ways, picking elements from the standard format:

```
name [selector] : class_name ref decomposition
```

- The elements of the format that you pick to use for a particular participant will depend on the information known about a participant at a given time, as explained

| | |
|---|---|
| admin | A part is named `admin`, but at this point in time the part has not been assigned a class. |
| : ContentManagementSystem | The class of the participant is `ContentManagementSystem`, but the part currently does not have its own name. |
| admin : Administrator | There is a part that has a name of `admin` and is of the class `Administrator`. |
| eventHandlers [2] : EventHandler | There is a part that is accessed within an array at element 2, and it is of the class `EventHandler`. |

# 2. Time in Sequence Diagrams

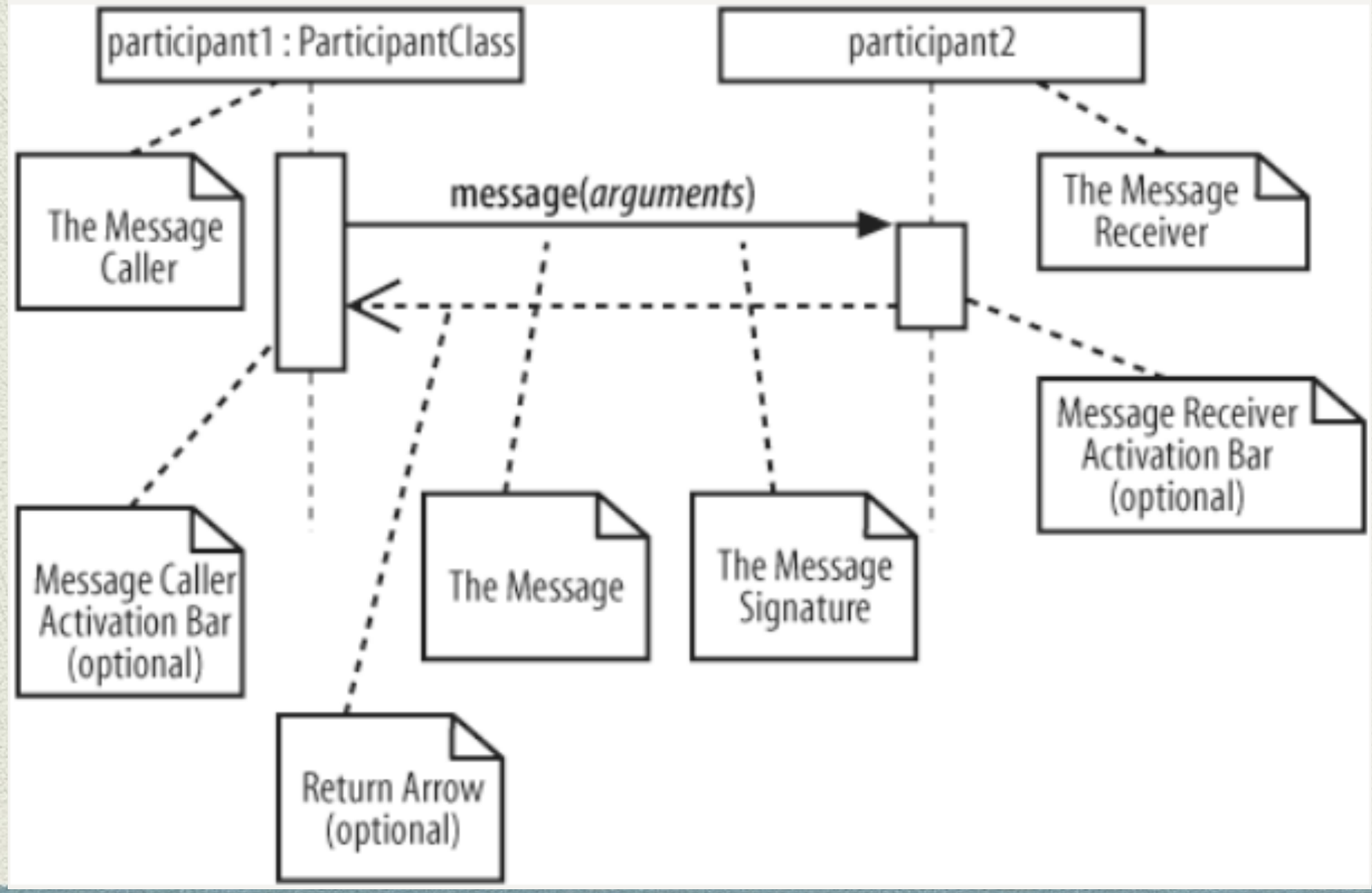


# 2. Time in Sequence Diagrams (Cont. )

- A sequence diagram describes the order in which the interactions take place, so time is an important factor.
- Time on a sequence diagram starts at the top of the page, just beneath the topmost participant heading, and then progresses down the page.
- The order that interactions are placed down the page on a sequence diagram indicates the order in which those interactions will take place in time.
- Time on a sequence diagram is all about ordering, not duration.

# 3. Messages in Sequence Diagrams

- An interaction in a sequence diagram occurs when one participant decides to send a message to another participant.



# 3. Messages in Sequence Diagrams (Cont.)

- Messages on a sequence diagram are specified using an arrow from the participant that wants to pass the message, the Message Caller, to the participant that is to receive the message, the Message Receiver.
- Messages can flow in whatever direction makes sense for the required interaction—from left to right, right to left, or even back to the Message Caller itself.
- Think of a message as an event that is passed from a Message Caller to get the Message Receiver to do something

# Message Signatures

- A message arrow comes with a description, or signature. The format for a message signature is:

```
attribute = signal_or_message_name (arguments) : return_type
```

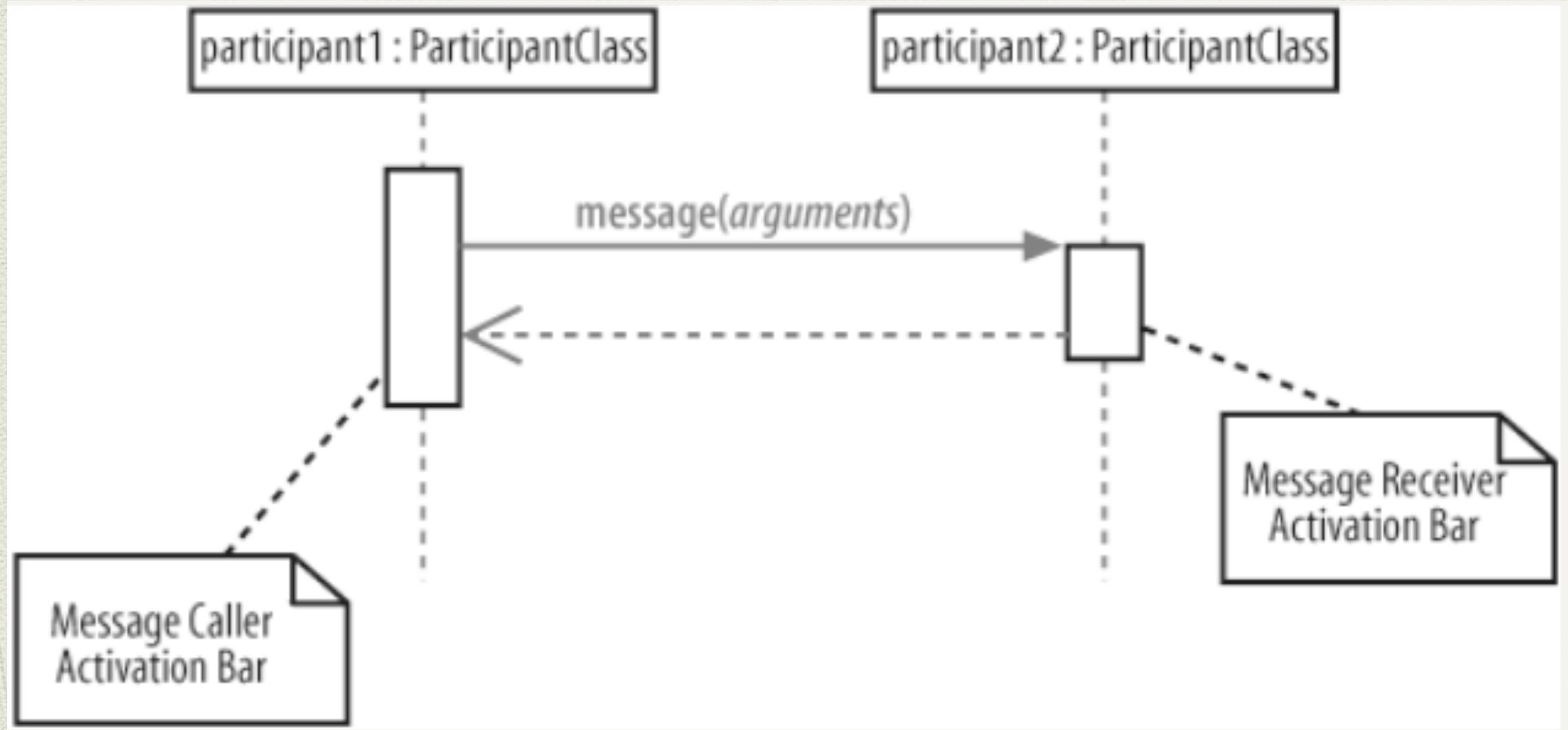
- You can specify any number of different arguments on a message, each separated using a comma. The format of an argument is:

```
<name>:<class>
```

# Message Signature – Examples

- doSomething( )
- doSomething(number1 : Number, number2 : Number)
- doSomething( ) : ReturnClass
- myVar = doSomething( ) : ReturnClass

# Activation Bars

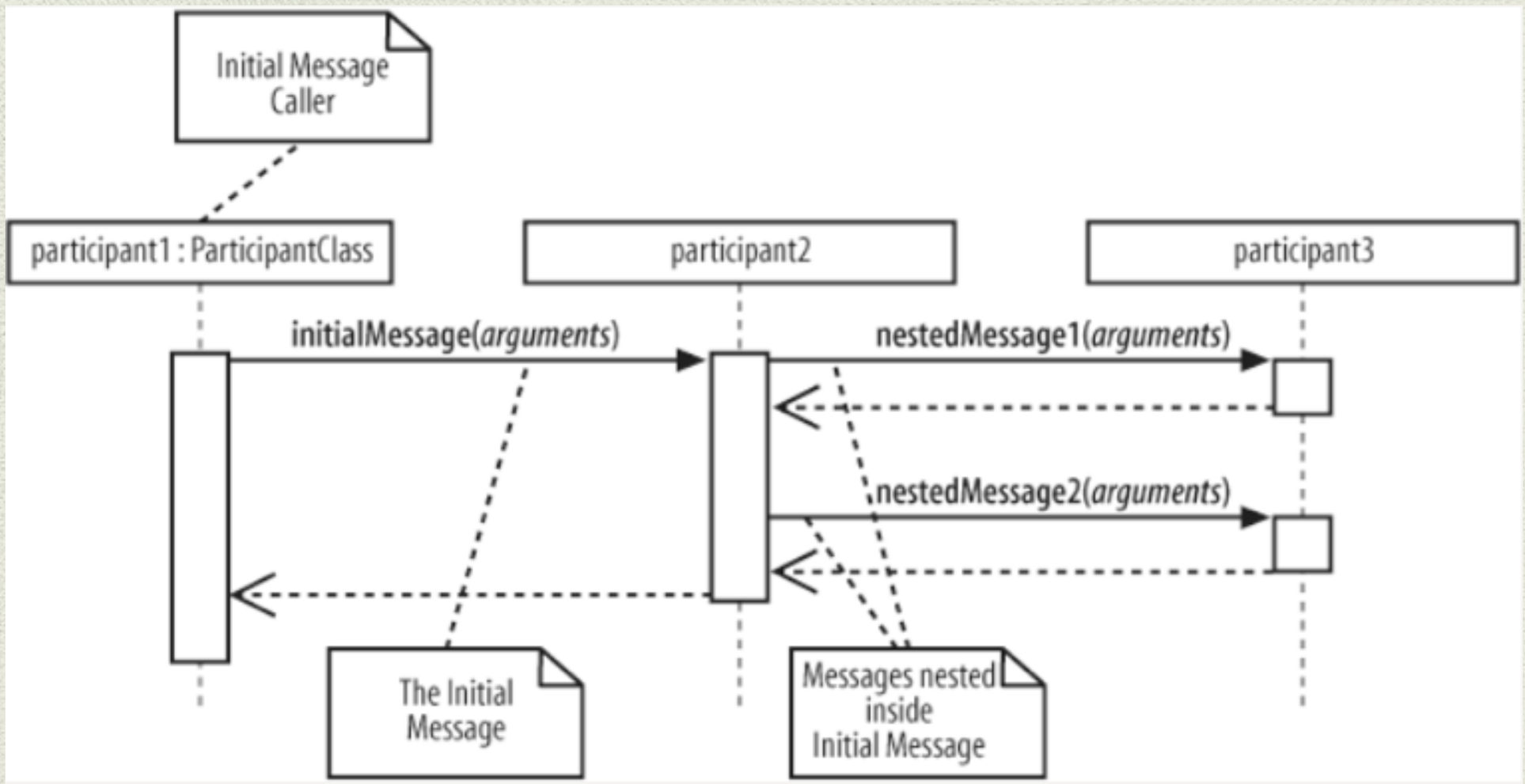


# Activation Bars (Cont.)

- When a message is passed to a participant it triggers, or invokes, the receiving participant into doing something; at this point, the receiving participant is said to be active. To show that a participant is active, i.e., doing something, you can use an activation bar.
- An activation bar can also be shown on the sending end. It indicates that the sending participant is busy while it sends the message.
- Activation bars are optional—they can clutter up a diagram.

# Nested Messages

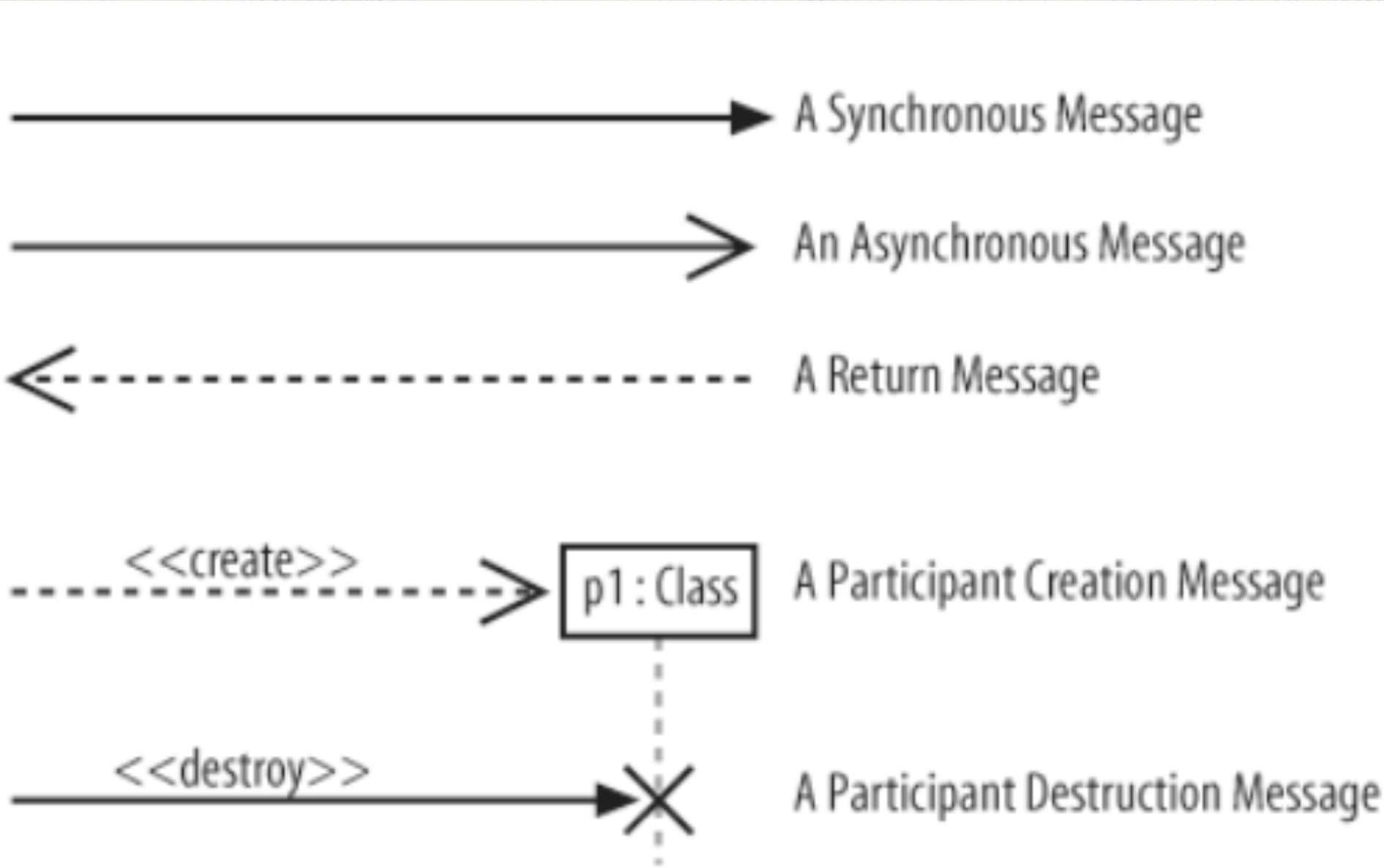
- When a message from one participant results in one or more messages being sent by the receiving participant, those resulting messages are said to be nested within the triggering message



# Message Arrows

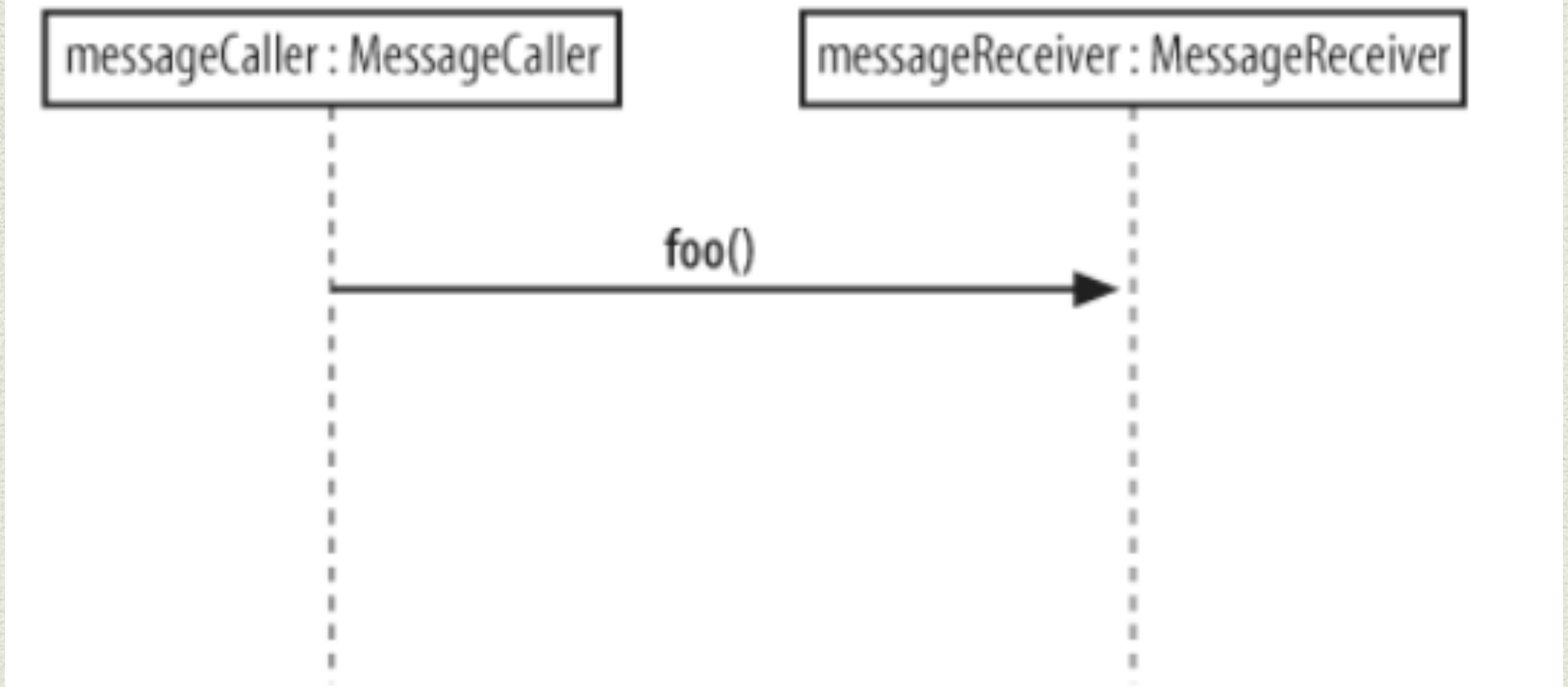
- The type of arrowhead that is on a message is also important when understanding what type of message is being passed.
- For example, the Message Caller may want to wait for a message to return before carrying on with its work—a synchronous message.
- Or it may wish to just send the message to the Message Receiver without waiting for any return as a form of "fire and forget" message —an asynchronous message.
- Sequence diagrams need to show these different types of message using various message arrows.

# Message Arrows (Cont.)



# Synchronous Messages

- A synchronous message is invoked when the Message Caller waits for the Message Receiver to return from the message invocation



```
public class MessageReceiver
{
   public void foo(  )
   {
      // Do something inside foo.
   }
}

public class MessageCaller
{
   private MessageReceiver messageReceiver;

   // Other Methods and Attributes of the class are declared here

   // The messageRecevier attribute is initialized elsewhere in
   // the class.

   public doSomething(String[] args)
   {
      // The MessageCaller invokes the foo(  ) method

       this.messageReceiver.foo(  ); // then waits for the method to return

      // before carrying on here with the rest of its work
   }
}
```

# Asynchronous Messages

- An asynchronous message is invoked by a Message Caller on a Message Receiver, but the Message Caller does not wait for the message invocation to return before carrying on with the rest of the interaction's steps.
- This means that the Message Caller will invoke a message on the Message Receiver and the Message Caller will be busy invoking further messages before the original message returns.

# Asynchronous Messages– Example

- If you are designing a piece of software with a user interface that supports the editing and printing of a set of documents. Your application offers a button for the user to print a document.

- Printing could take some time, so you want to show that after the print button is pressed and the document is printing, the user can go ahead and work with other things in the application. Here you need a new type of message arrow: the asynchronous message arrow.

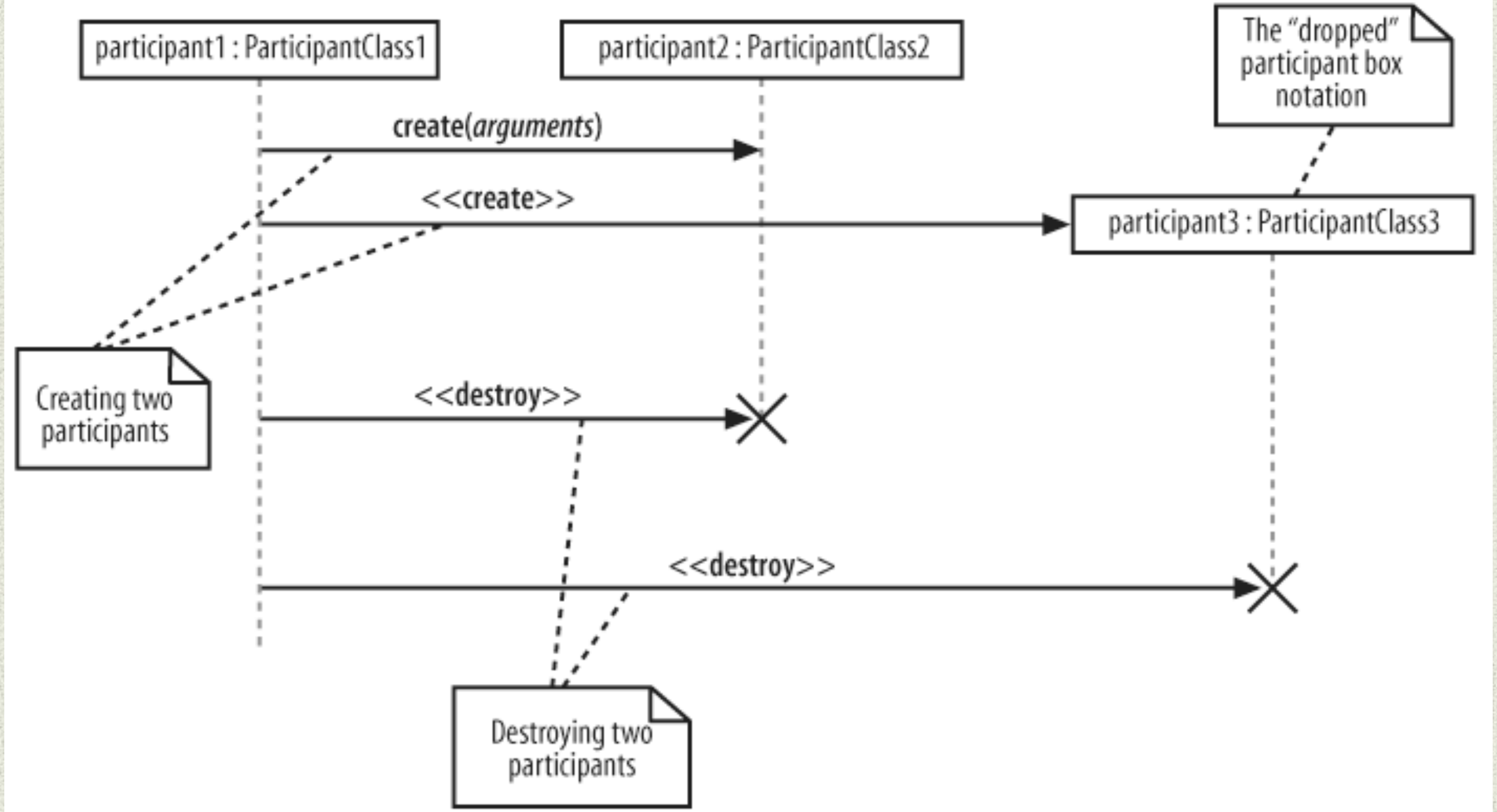
# The Return Message

- The return message is an optional piece of notation that you can use at the end of an activation bar to show that the control flow of the activation returns to the participant that passed the original message.
- In code, a return arrow is similar to reaching the end of a method or explicitly calling a return statement.
- You don't have to use return messages —sometimes they can really make your sequence diagram too busy and confusing.

# Participant Creation and Destruction Messages

- Participants do not necessarily live for the entire duration of a sequence diagram's interaction. Participants can be created and destroyed according to the messages that are being passed
- You don't have to use return messages —sometimes they can really make your sequence diagram too busy and confusing.
- You don't have to clutter up your sequence diagrams with a return arrow for every activation bar since there is an implied return arrow on any activation bars that are invoked using a synchronous message.

# Participant Creation and Destruction Messages



# Participant Creation and Destruction Messages (Cont.)

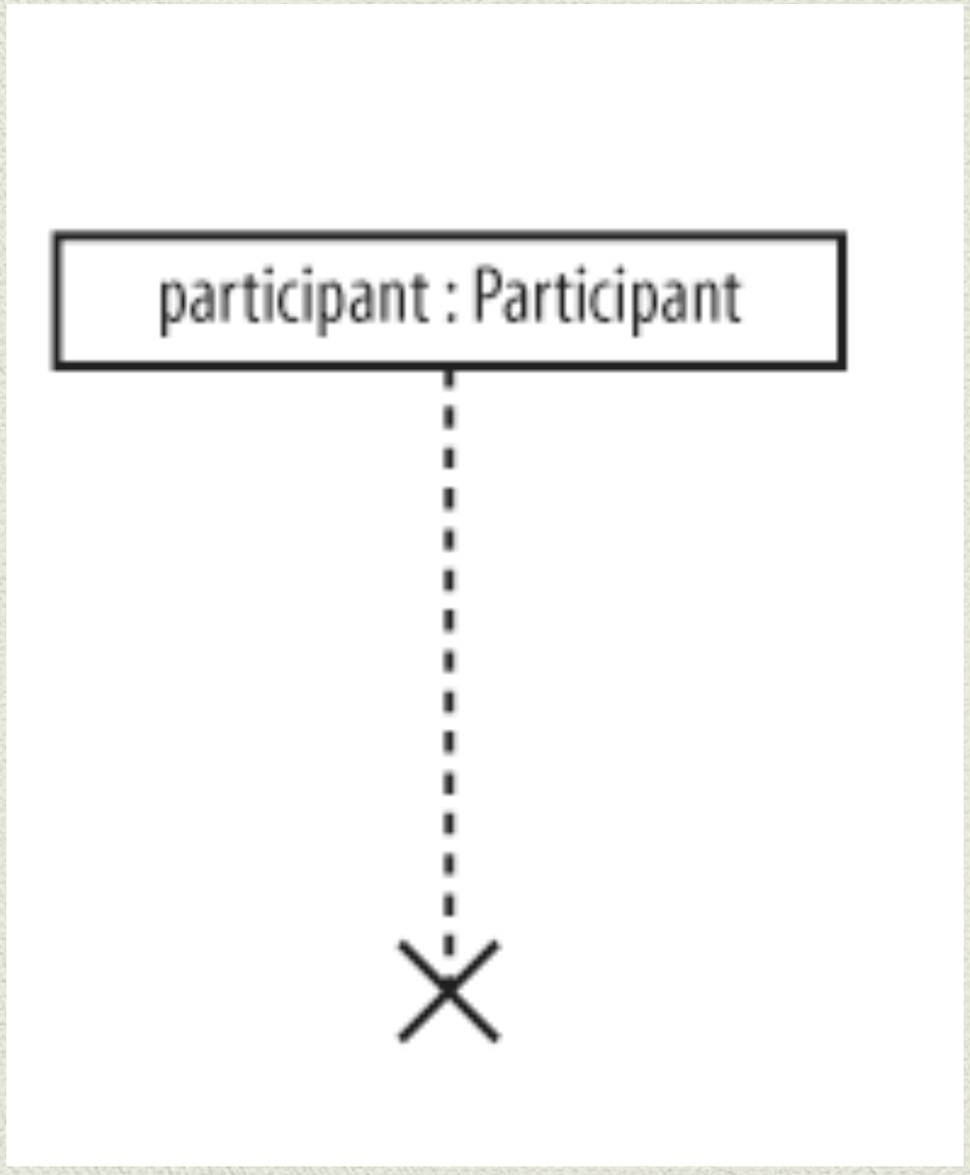
```
public class MessageReceiver {
    // Attributes and Methods of the MessageReceiver class
}
```

```
public class MessageCaller {

    // Other Methods and Attributes of the class are declared here

    public void doSomething(  ) {
        // The MessageReceiver object is created
        MessageReceiver messageReceiver = new MessageReceiver(  );
    }
}
```

# Participant Creation and Destruction Messages (Cont.)

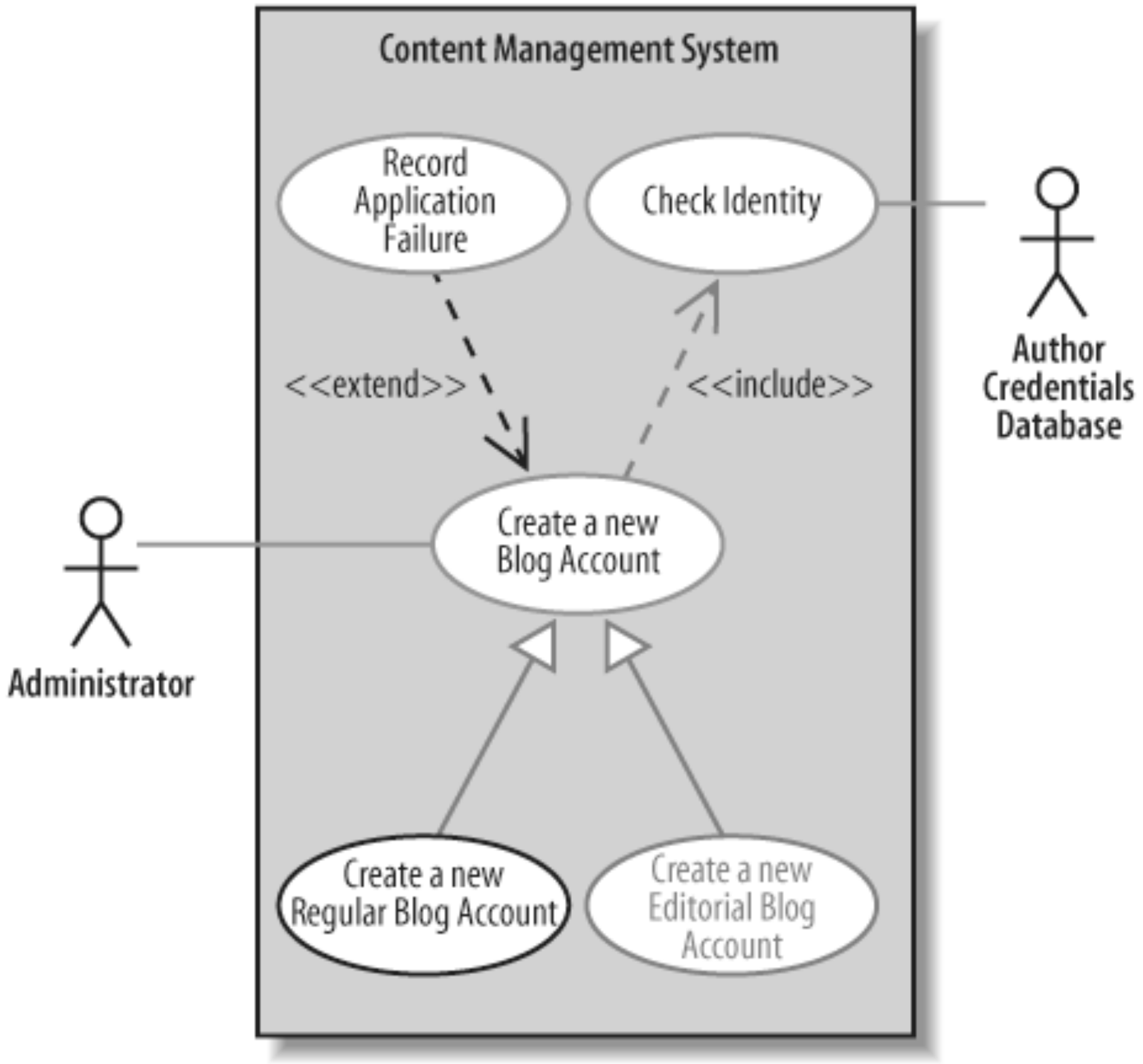
- With some implementation languages, such as Java, you will not have an explicit destroy method so it doesn't make sense to show one on your sequence diagrams.
- It is all handled implicitly by the Java garbage collector.
- In these cases, where another factor such as the garbage collector is involved, you can either leave the object as alive but unused or imply that it is no longer needed by using the destruction cross without an associated `destroy` method.

# Participant Creation and Destruction Messages (Cont.)



# Bringing a Use Case to Life with a Sequence Diagram

## Example


Content Management System
Record Application Failure
Check Identity
<<extend>>
<<include>>
Create a new Blog Account
Create a new Regular Blog Account
Create a new Editorial Blog Account
Author Credentials Database
Administrator

# The Scenario

1 The Administrator asks the system to create a new blog account.

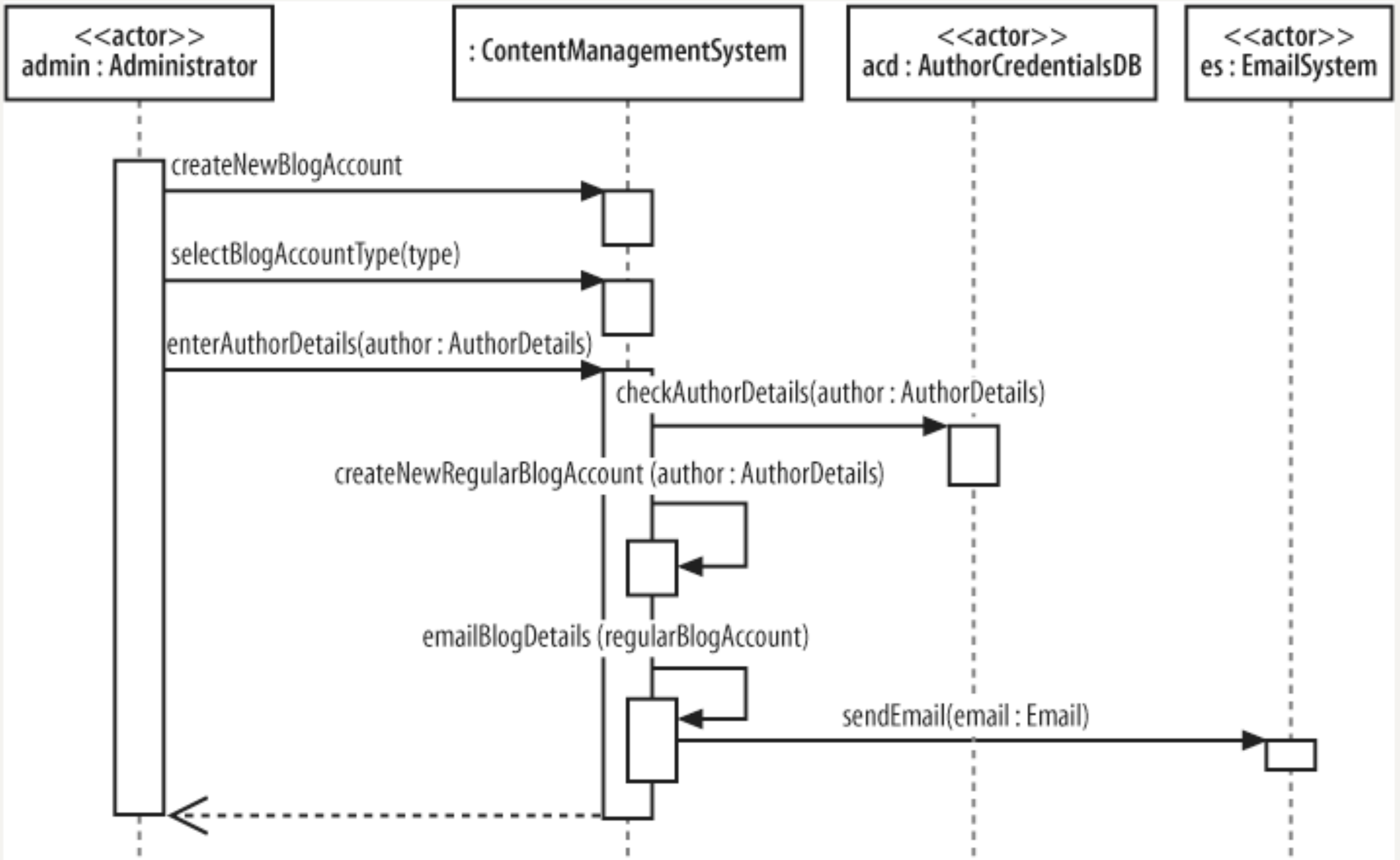
2 The Administrator selects the regular blog account type.

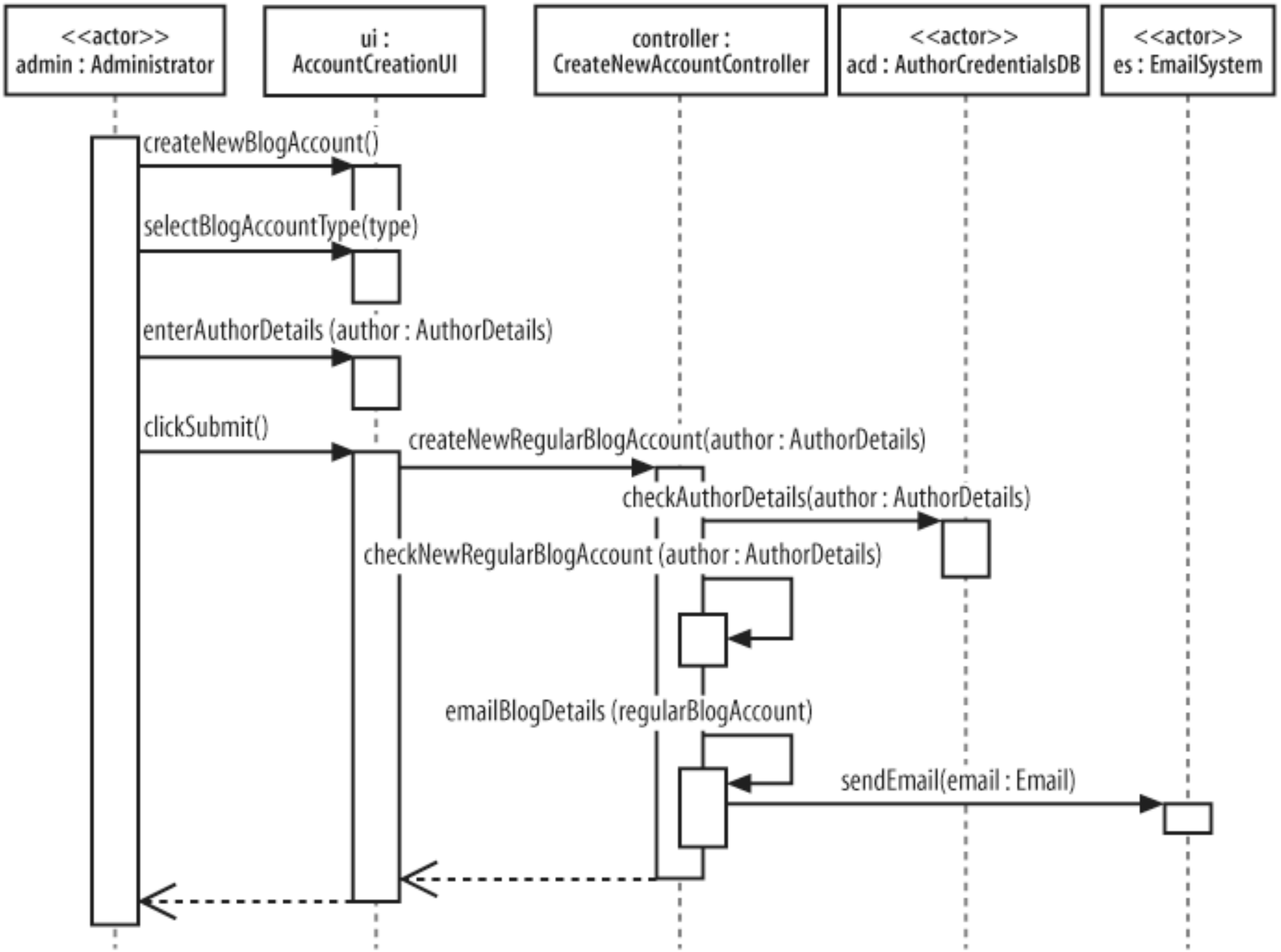
3 The Administrator enters the author's details.

4 The author's details are checked using the Author Credentials Database.

5 The new regular blog account is created.

6 A summary of the new blog account's details are emailed to the author.


<<actor>>
admin : Administrator
: ContentManagementSystem
<<actor>>
acd : AuthorCredentialsDB
<<actor>>
es : EmailSystem
createNewBlogAccount
selectBlogAccountType(type)
enterAuthorDetails(author : AuthorDetails)
checkAuthorDetails(author : AuthorDetails)
createNewRegularBlogAccount (author : AuthorDetails)
emailBlogDetails (regularBlogAccount)
sendEmail(email : Email)


<<actor>>
admin : Administrator
ui :
AccountCreationUI
controller :
CreateNewAccountController
<<actor>>
acd : AuthorCredentialsDB
<<actor>>
es : EmailSystem
createNewBlogAccount()
selectBlogAccountType(type)
enterAuthorDetails (author : AuthorDetails)
clickSubmit()
createNewRegularBlogAccount(author : AuthorDetails)
checkAuthorDetails(author : AuthorDetails)
checkNewRegularBlogAccount (author : AuthorDetails)
emailBlogDetails (regularBlogAccount)
sendEmail(email : Email)

# References


- Fowler, M. (2004). UML distilled: a brief guide to the standard object modeling language. Addison-Wesley Professional.
- Miles, R and Hamilton, K. (2006) Learning UML 2.0. Sebastopol: O'Reilly Media, Inc.
- Pender, T (2003). UML Bible. John Wiley & Sons, Inc., New York, NY.
- Pilone, D., & Pitman, N. (2006). *UML 2.0 in a nutshell*. O'Reilly.