7. Modeling Ordered Interactions: Sequence Diagrams

Shaoning Zeng, http://zsn.cc

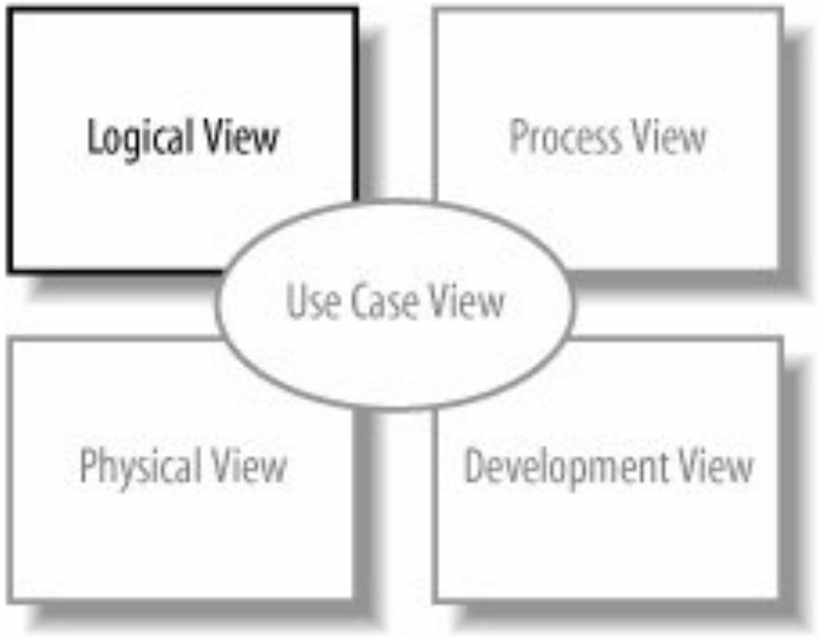
7. Modeling Ordered Interactions: Sequence Diagrams

- 7.1. Participants in a Sequence Diagram 参与者
- 7.2. Time 时间
- 7.3. Events, Signals, and Messages 事件、信号与消息
- 7.4. Activation Bars 存活条
- 7.5. Nested Messages
- 7.6. Message Arrows
- 7.7. Bringing a Use Case to Life with a Sequence Diagram
- 7.8. Managing Complex Interactions with Sequence Fragments

7. Modeling Ordered Interactions: Sequence Diagrams

- 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.
- What to model how your system is actually going to its job. 系统如何完成某个交互?
  - This is where interaction diagrams ,
  - and specifically sequence diagrams, come into play.

Figure 7-1. The Logical View of your model contains the abstract descriptions of your system's parts, including the interactions between those parts



7. Modeling Ordered Interactions: Sequence Diagrams

- Sequence diagrams are an important member of the group known as interaction diagrams. 是一种交互图
  - Interaction diagrams model important runtime interactions between the parts that make up your system and form part of the logical view of your model 运行时交互
- Sequence diagrams are not alone in this group;
  - they work alongside communication diagrams (see Chapter 8)
  - and timing diagrams (see Chapter 9) to help you accurately model how the parts that make up your system interact.

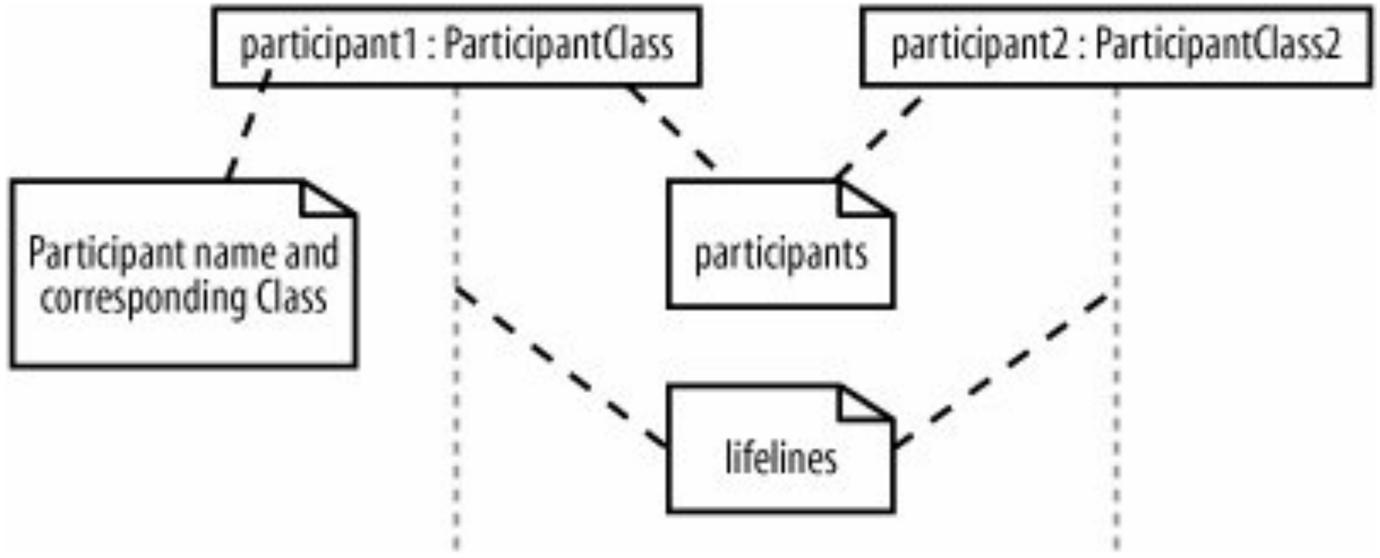
7. Modeling Ordered Interactions: Sequence Diagrams

- Sequence diagrams are all about capturing the order of interactions between parts of your system. 交互的顺序
  - Using a sequence diagram, you can describe which interactions will be triggered when a particular use case is executed 用例执行过程中触发了哪些交互
  - and in what order those interactions will occur. 强调顺序
- Sequence diagrams show plenty of other information about an interaction, but their forté is the simple and effective way in which they communicate the order of events within an interaction. 事件顺序

7.1. Participants in a Sequence Diagram

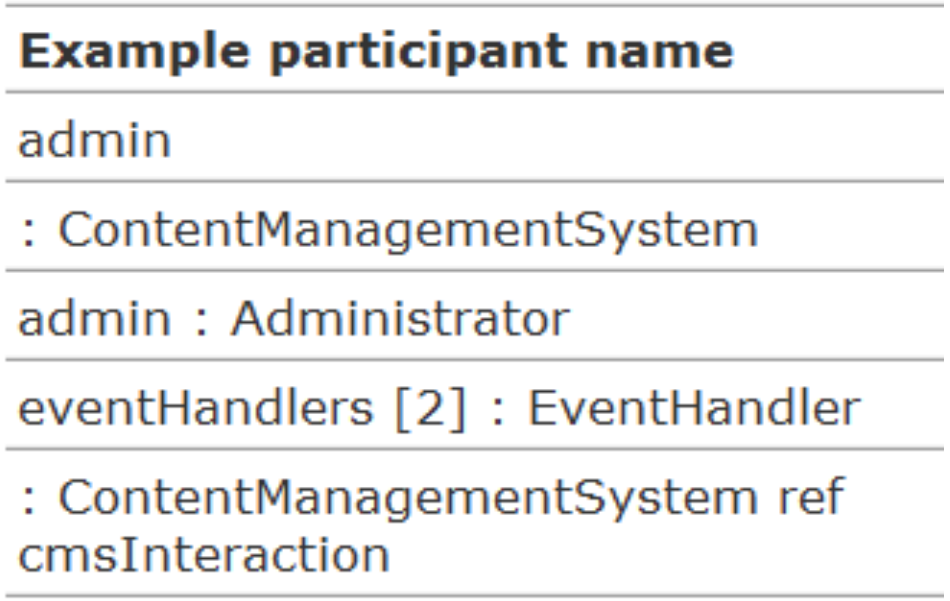
- A sequence diagram is made up of a collection of participants - the parts of your system that interact with each other during the sequence. 参与交互的对象
- Where a participant is placed on a sequence diagram is important. 参与者的位置很重要
  - Regardless of where a participant is placed vertically, participants are always arranged horizontally with no two participants overlapping each other 水平排列，不重叠
- Each participant has a corresponding lifeline(生命线) running down the page.
  - A participant's lifeline simply states that the part exists at that point in the sequence and is only really interesting when a part is created and/or deleted during a sequence
    生命线表示该时刻对象真正存在

Figure 7-2. At its simplest, a sequence diagram is made up of one or more participants only one participant would be a very strange sequence diagram, but it would be perfectly legal UML


participant1 : ParticipantClass
participant2 : ParticipantClass2
Participant name and corresponding Class
participants
lifelines

Participant Names

- *name* [selector] : *class_name ref decomposition*



Example participant name
admin
: ContentManagementSystem
admin : Administrator
eventHandlers [2] : EventHandler
: ContentManagementSystem ref cmsInteraction

7.2. Time 时间

- A sequence diagram describes the order in which the interactions take place, so time is an important factor. 描述了交互发生的顺序 – 时间
- Time on a sequence diagram starts at the top of the page, just beneath the topmost participant heading, and then progresses down the page. 时间顺序 - 从上到下
  - 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.

7.2. Time

- Time on a sequence diagram is all about ordering, not duration. 时间仅表示先后顺序，不表示时间跨度
  - Although the time at which an interaction occurs is indicated on a sequence diagram by where it is placed vertically on the diagram, how much of the vertical space the interaction takes up has nothing to do with the duration of time that the interaction will take. 垂直方向位置与时间跨度无关
  - Sequence diagrams are first about the ordering of the interactions between participants; more detailed timing information is better shown on timing diagrams (see Chapter 9).

7.3. Events, Signals, and Messages

- An event is any point in an interaction where something occurs, as shown on Figure 7-4.
- Events are the building blocks for signals and messages.
  - Signals and messages are really different names for the same concept: a signal is the terminology often used by system designers, while software designers often prefer messages.
- In terms of sequence diagrams, signals and messages act and look the same, so we'll stick to using the term "messages" in this book. 事件 = 消息 = 信号

Figure 7-4. Probably the most common examples of events are when a message or signal is sent or received

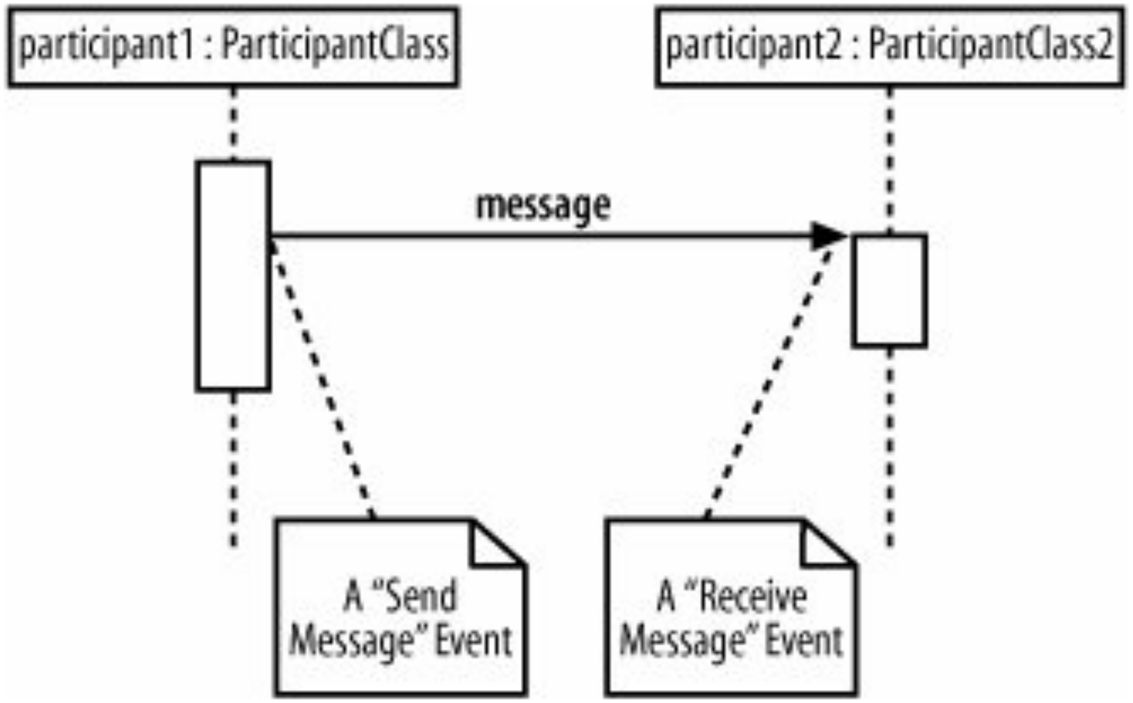
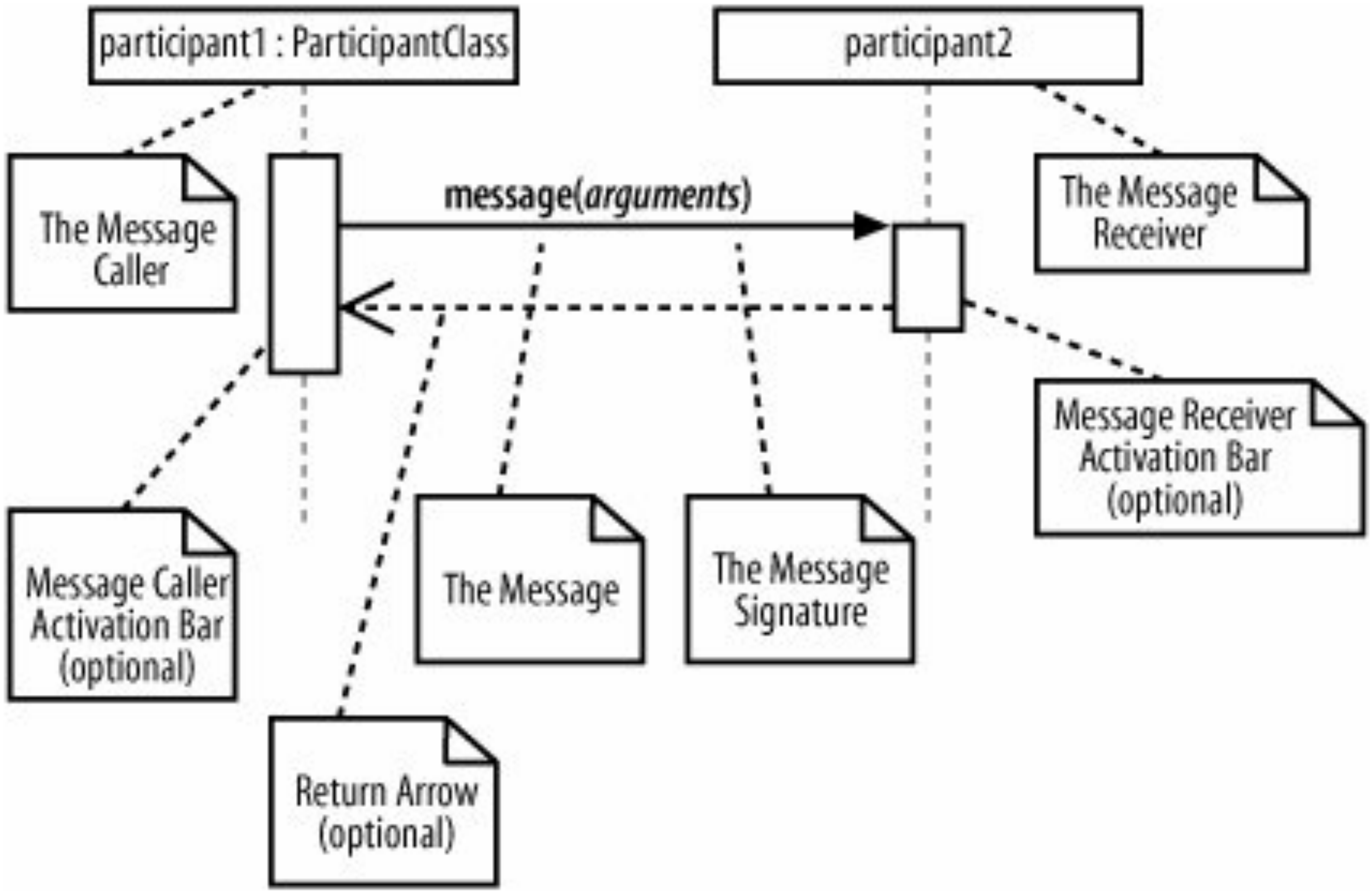
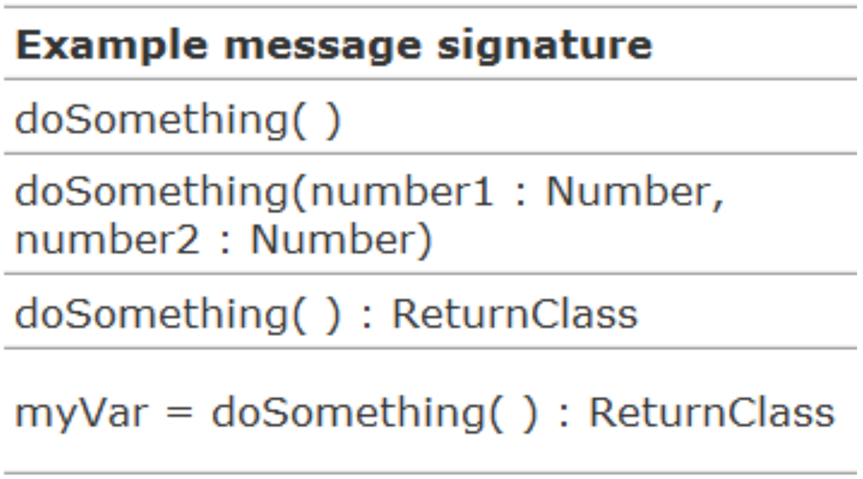

participant1 : ParticipantClass
participant2 : ParticipantClass2
message
A "Send Message" Event
A "Receive Message" Event

Figure 7-5. Interactions on a sequence diagram are shown as messages between participants



Message Signatures 消息签名

- 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>***

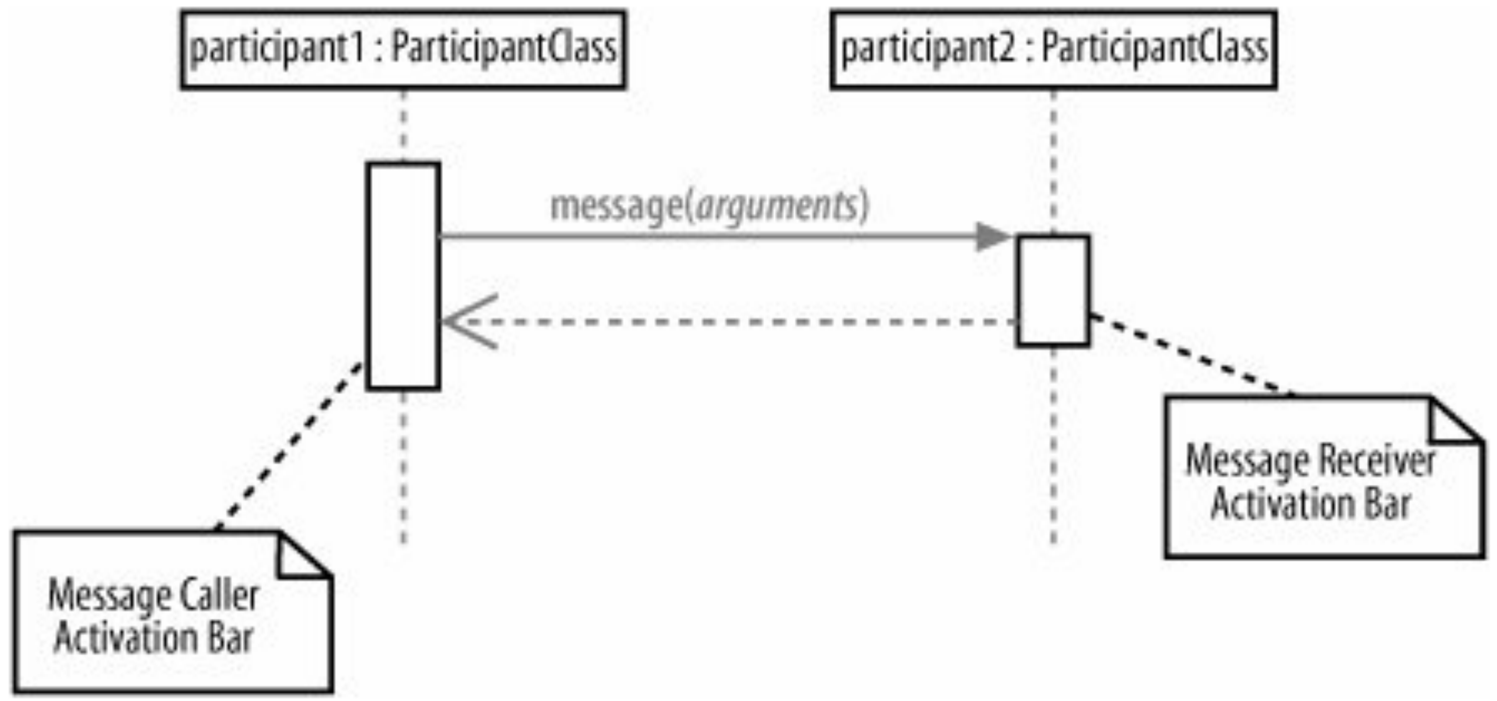


Example message signature
doSomething( )
doSomething(number1 : Number, number2 : Number)
doSomething( ) : ReturnClass
myVar = doSomething( ) : ReturnClass

7.4. Activation Bars 存活条

- 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, as shown in Figure 7-6.
- An activation bar can be shown on the sending and receiving ends of a message. 显示了发送端和接收端
  - It indicates that the sending participant is busy while it sends the message
  - and the receiving participant is busy after the message has been received

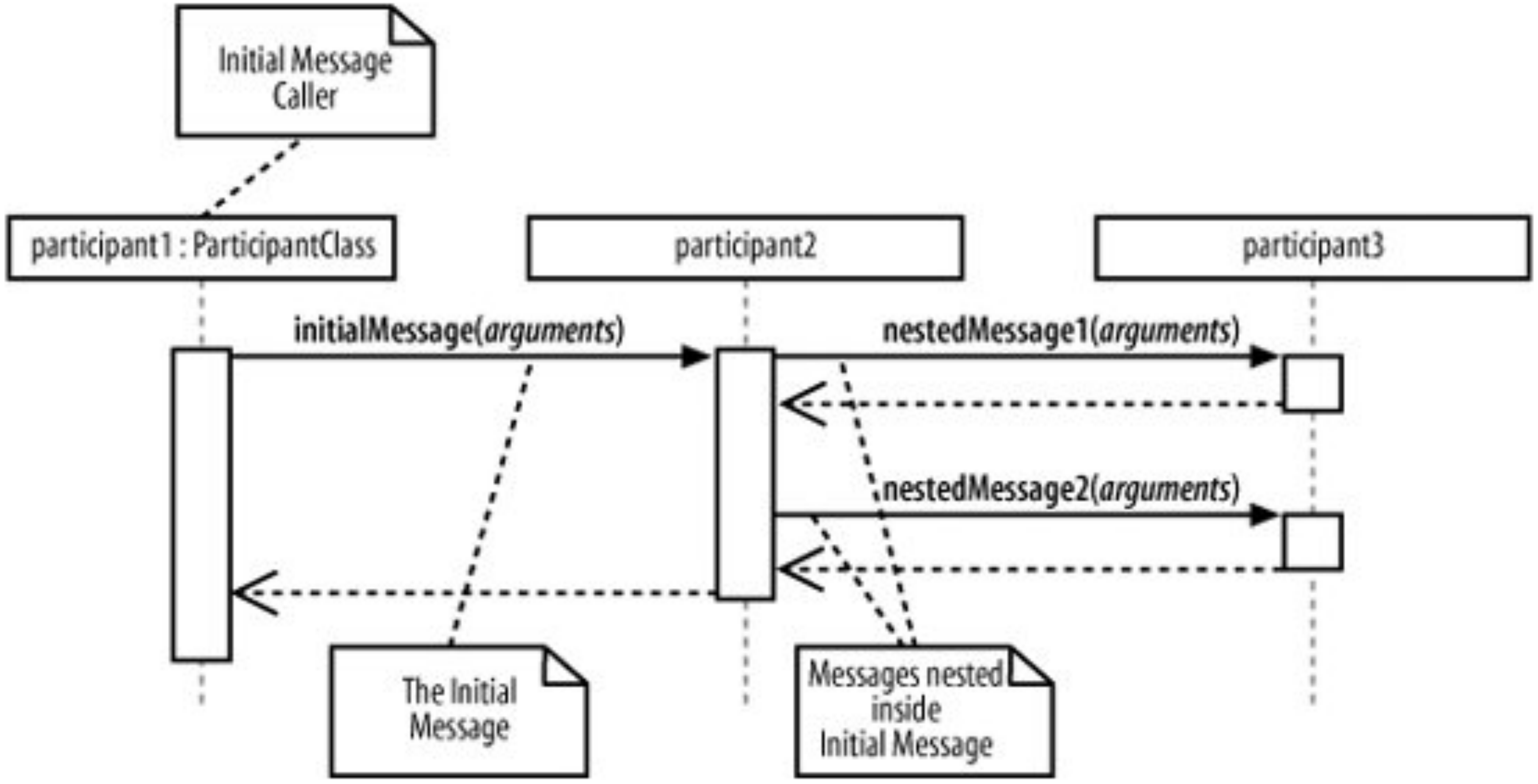
Figure 7-6. Activation bars show that a participant is busy doing something for a period of time


participant1 : ParticipantClass
participant2 : ParticipantClass
message(*arguments*)
Message Receiver Activation Bar
Message Caller Activation Bar

7.5. 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, as shown in Figure 7-7.
  一条消息导致多条消息发生

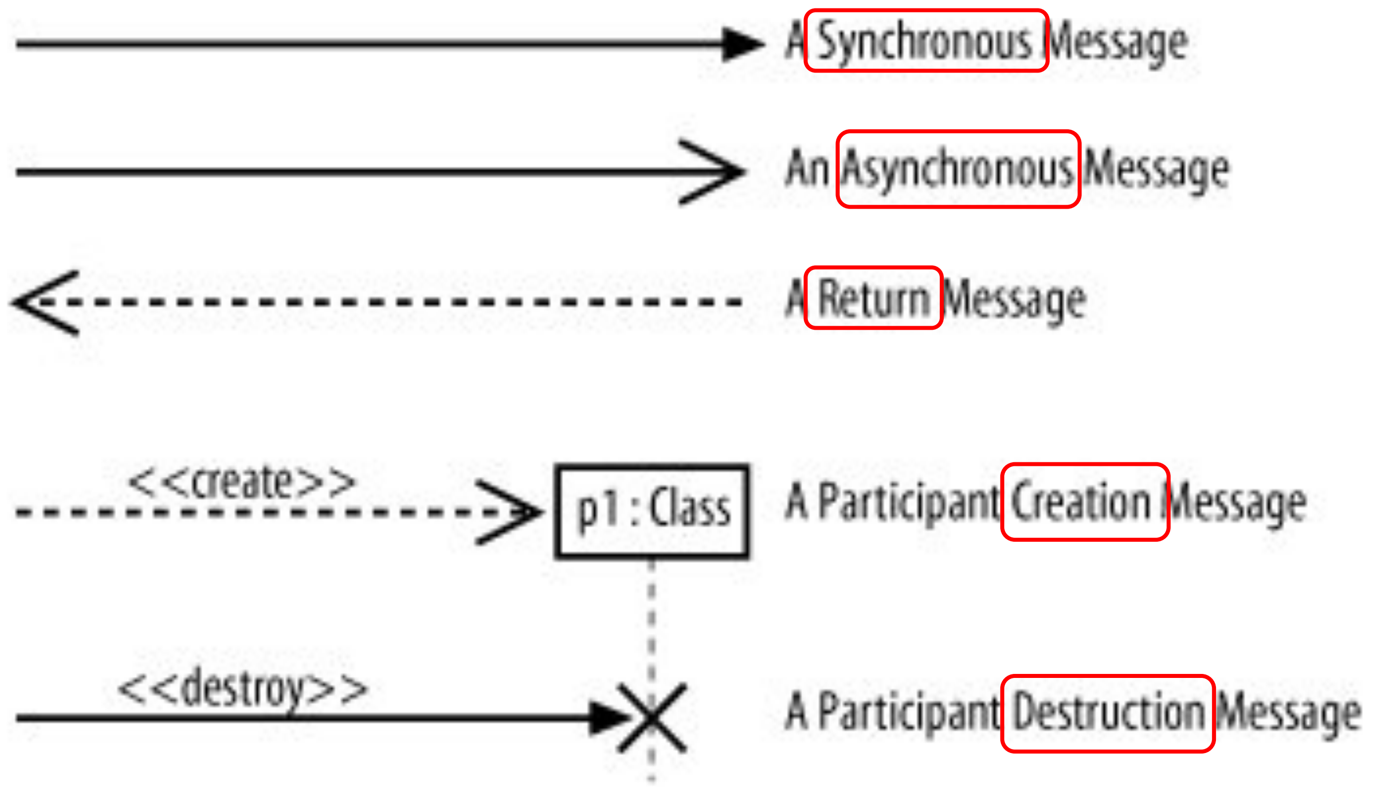
Figure 7-7. Two nested messages are invoked when an initial message is received


Initial Message Caller
participant1 : ParticipantClass
participant2
participant3
initialMessage(*arguments*)
nestedMessage1(*arguments*)
nestedMessage2(*arguments*)
The Initial Message
Messages nested inside Initial Message

7.6. Message Arrows 消息箭头

- The type of arrowhead that is on a message is also important when understanding what type of message is being passed. 消息的箭头也非常重要
  - 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 , as shown in Figure 7-8. 不同的箭头表示不同的含义

Figure 7-8. There are **five** main types of message arrow for use on sequence diagram, and each has its own meaning



7.6.1. Synchronous Messages

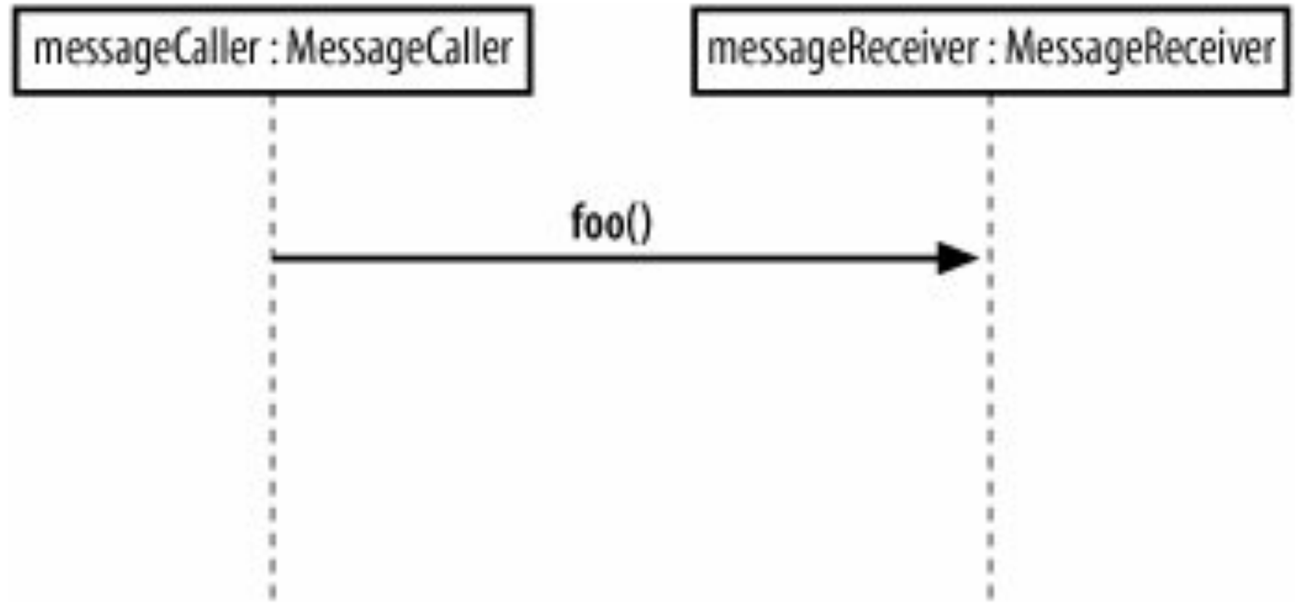

messageCaller : MessageCaller
messageReceiver : MessageReceiver
foo()

Figure 7-9. The messageCaller participant makes a single synchronous message invocation on the messageReceiver participant

Example 7-1. The messageCaller object makes a regular Java **method call** to the foo( ) method on the messageReceiver object and then **waits for** the messageReceiver.foo( ) method to **return** before carrying on with any further steps in the interaction

```
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
   }
}
```

7.6.2. Asynchronous Messages

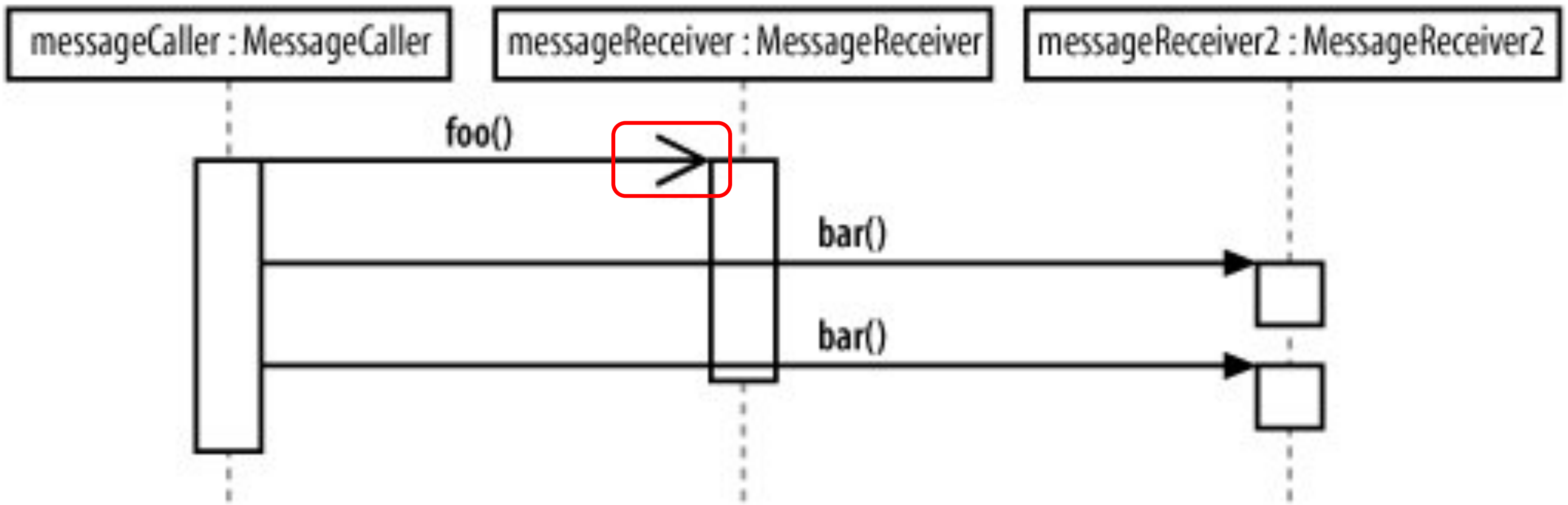
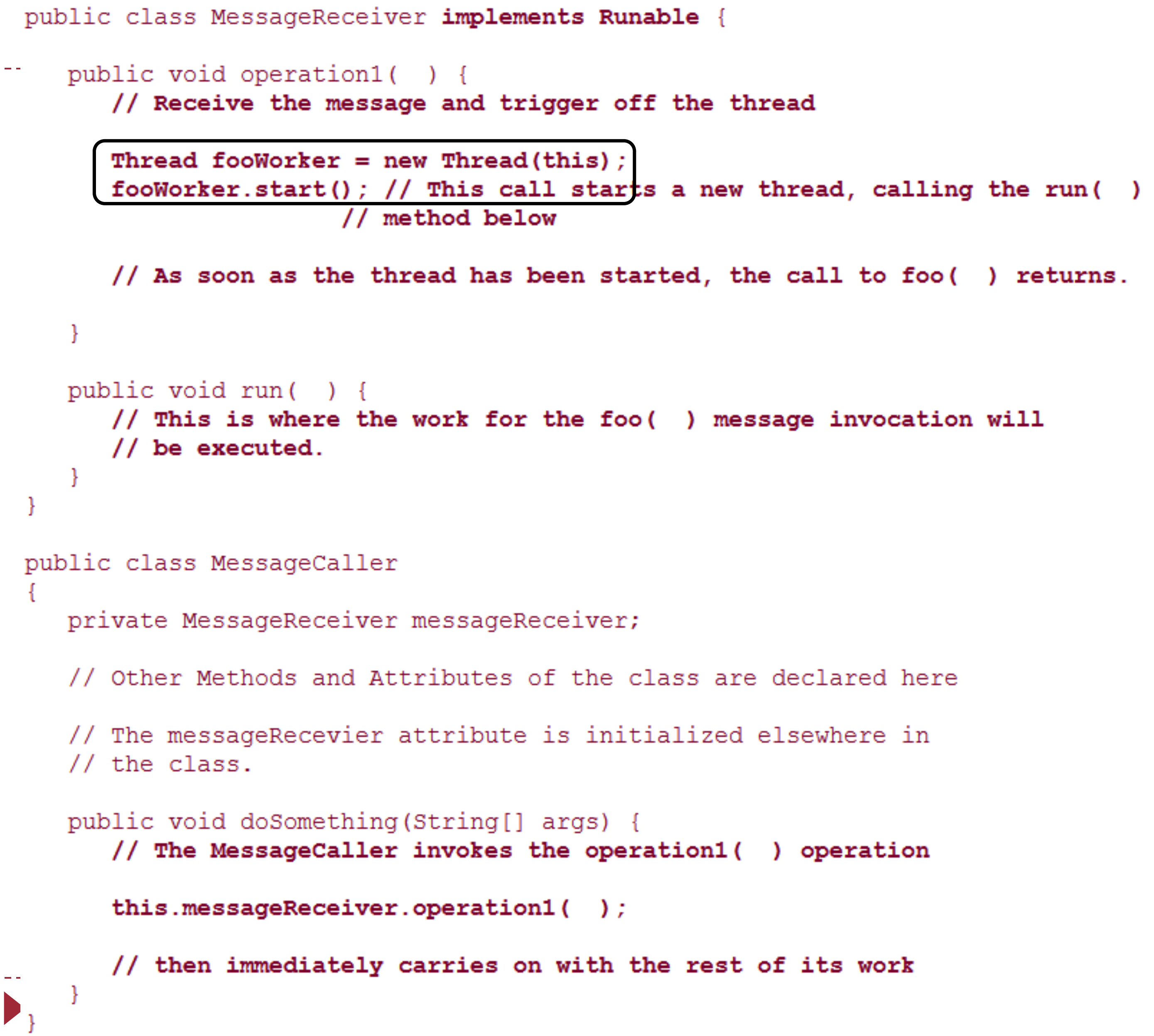

messageCaller : MessageCaller
messageReceiver : MessageReceiver
messageReceiver2 : MessageReceiver2
foo()
bar()
bar()

Figure 7-10. While the foo( ) message is being worked on by the messageReceiver object, the messageCaller object has carried on with the interaction by executing further synchronous messages on another object

Example 7-2. The operation1( ) **asynchronous**



```
public class MessageReceiver implements Runable {

   public void operation1(  ) {
      // Receive the message and trigger off the thread

      Thread fooWorker = new Thread(this);
      fooWorker.start(); // This call starts a new thread, calling the run(  )
                      // method below

      // As soon as the thread has been started, the call to foo(  ) returns.

   }

   public void run(  ) {
      // This is where the work for the foo(  ) message invocation will
      // be executed.
   }
}

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 void doSomething(String[] args) {
      // The MessageCaller invokes the operation1(  ) operation

      this.messageReceiver.operation1(  );

      // then immediately carries on with the rest of its work
   }
}
```

nessage

nediately

geCaller

7.6.3. The Return Message 返回消息

- 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. 可选的
  - In code, a return arrow is similar to reaching the end of a method or explicitly calling a return statement. 返回语句
- You don‘t have to use return messages sometimes they can really make your sequence diagram too busy and confusing. 不一定要添加返回消息
  - 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.

7.6.4. Participant Creation and Destruction Messages

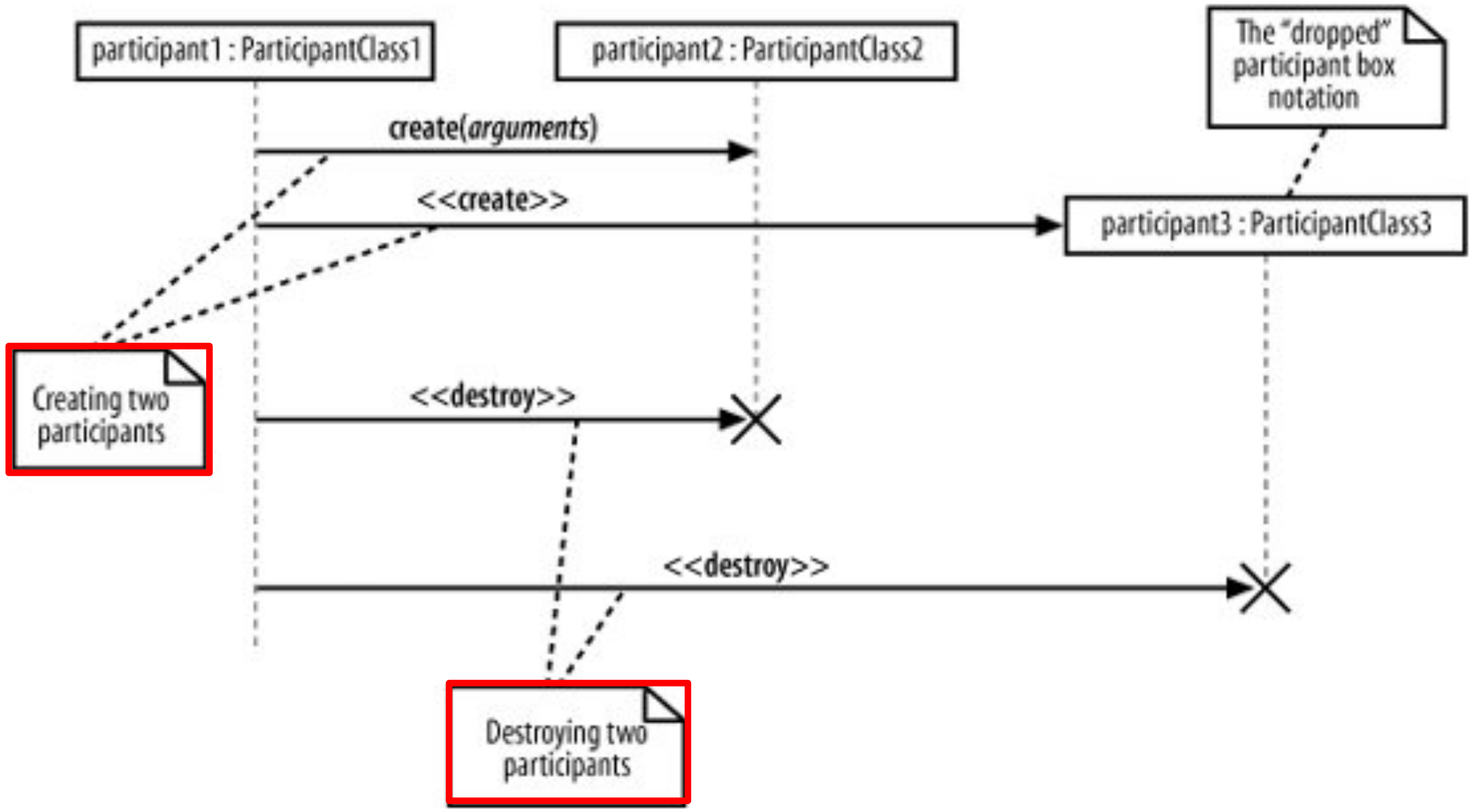

participant1 : ParticipantClass1
participant2 : ParticipantClass2
The "dropped" participant box notation
create(*arguments*)
<<create>>
participant3 : ParticipantClass3
Creating two participants
<<destroy>>
<<destroy>>
Destroying two participants

Figure 7-11. Both participant2 and participant3 are created throughout the course of this sequence diagram

Example 7-3. The MessageCaller creates a new MessageReceiver object simply by using the **new** keyword

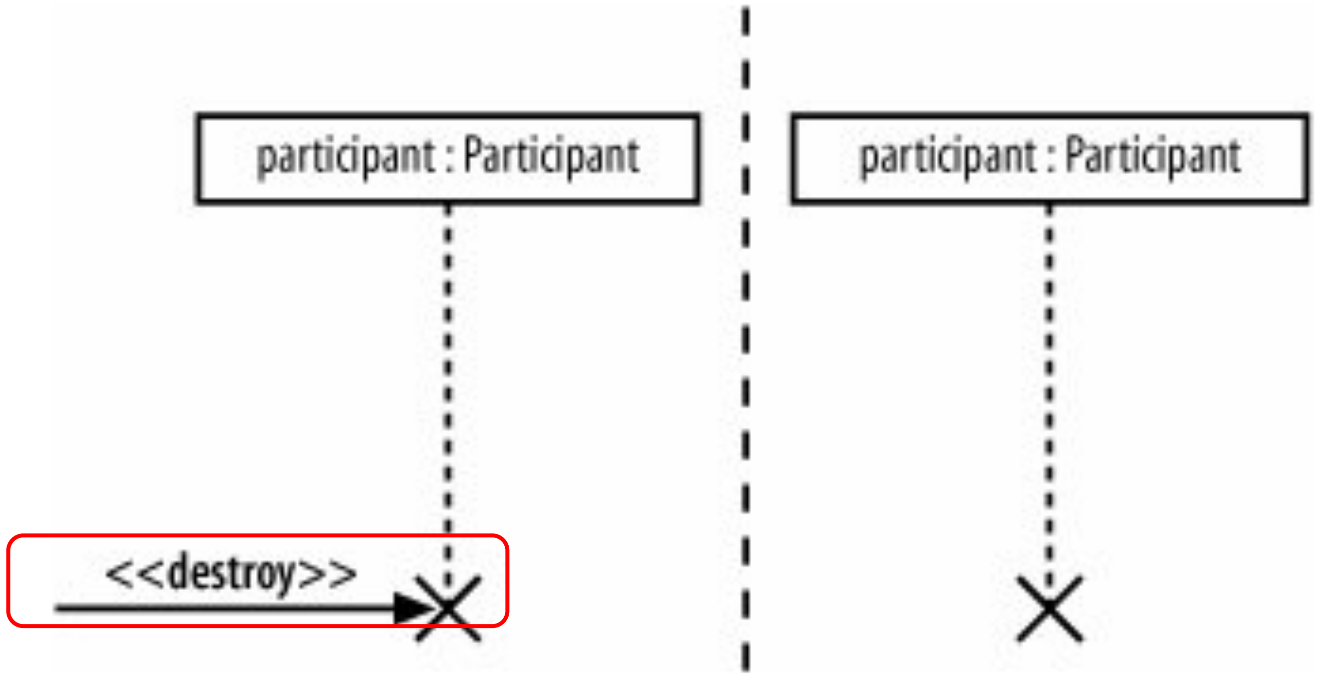
```
public class MessageReceiver {
   // Attributes and Methods of the MessageReceiver class
}

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(  );
   }
}
```

Figure 7-12. Using an explicit destroy message or implying that a participant has been discarded using just a destruction cross



7.7. Bringing a Use Case to Life with a Sequence Diagram 使用顺序图描述用例的交互

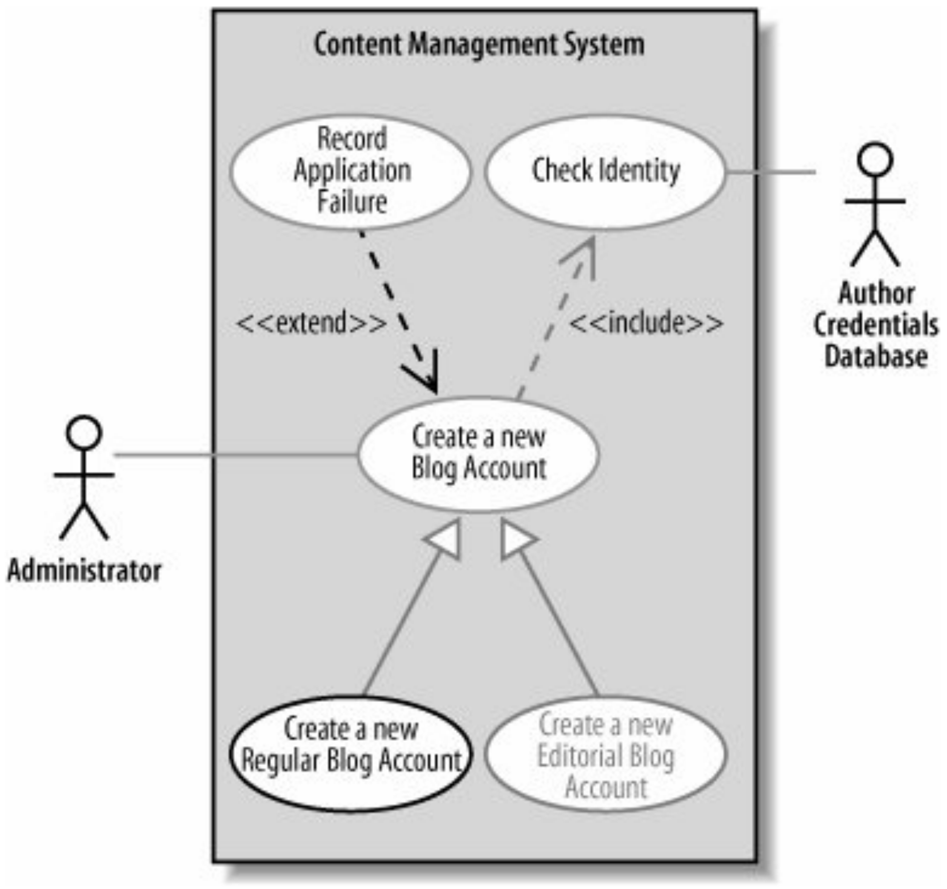
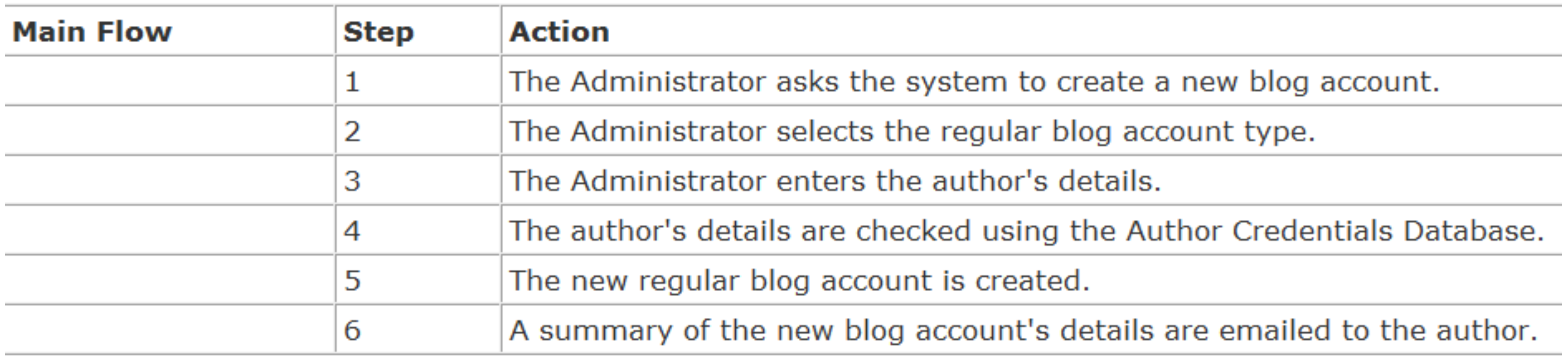

Content Management System
Record Application Failure
Check Identity
<<extend>>
<<include>>
Author Credentials Database
Administrator
Create a new Blog Account
Create a new Regular Blog Account
Create a new Editorial Blog Account

Figure 7-13. The Create a new Regular Blog Account use case diagram

7.7.1. A Top-Level Sequence Diagram

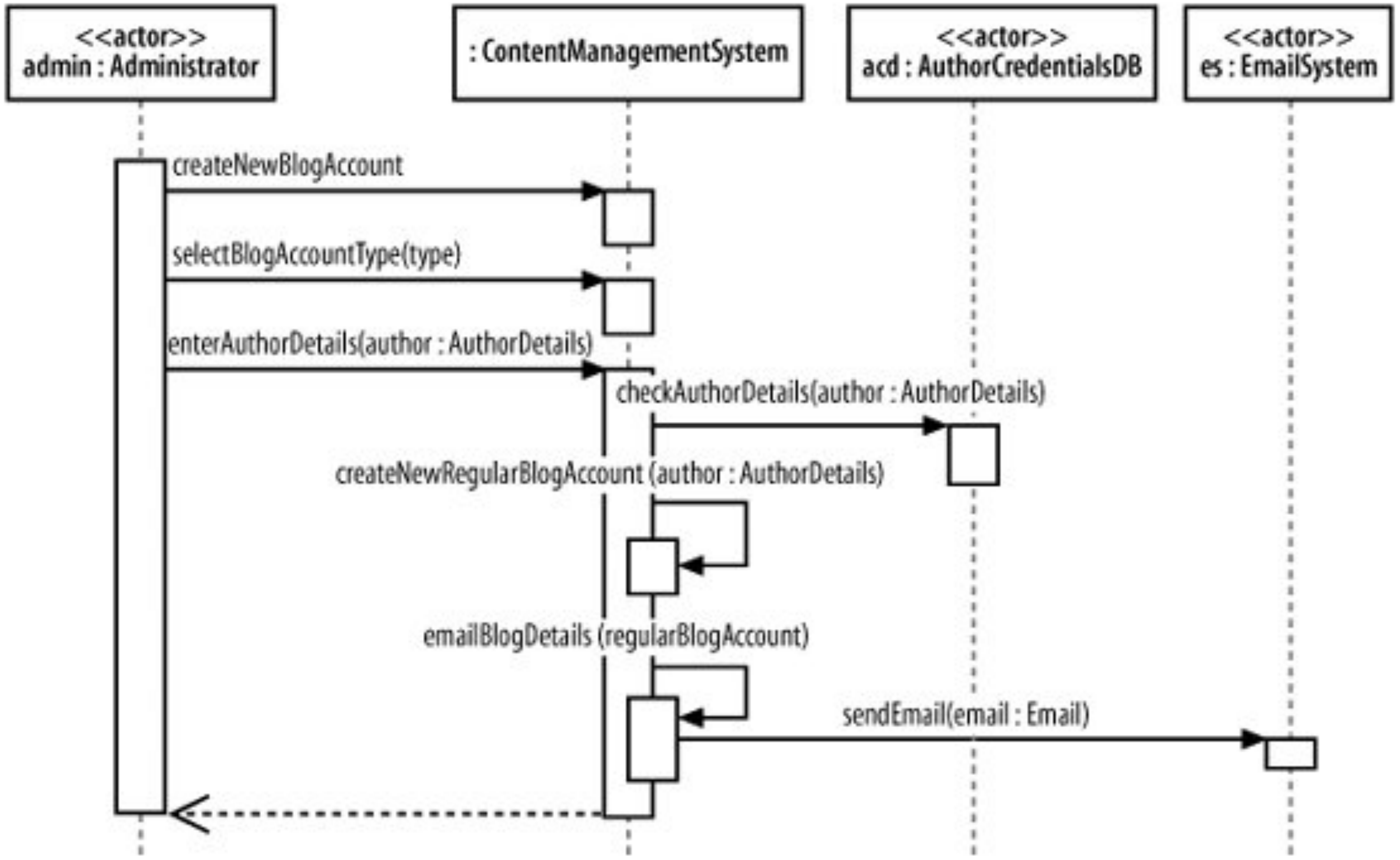
- Before you can specify what types of interaction are going to occur when a use case executes, you need a more detailed description of what the use case does. If you've already completed a use case description, you already have a good reference for this detailed information.



Main Flow	Step	Action
	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.

Table 7-3. Most of the detailed information that you will need to start constructing a sequence diagram for a use case should already be available as the Main Flow within the use case's description

Figure 7-14. This sequence diagram shows the actors that interact with your system and your system is shown simply as a single part in the sequence 描述参与者与系统的交互


<<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)

7.7.2. Breaking an Interaction into Separate Participants

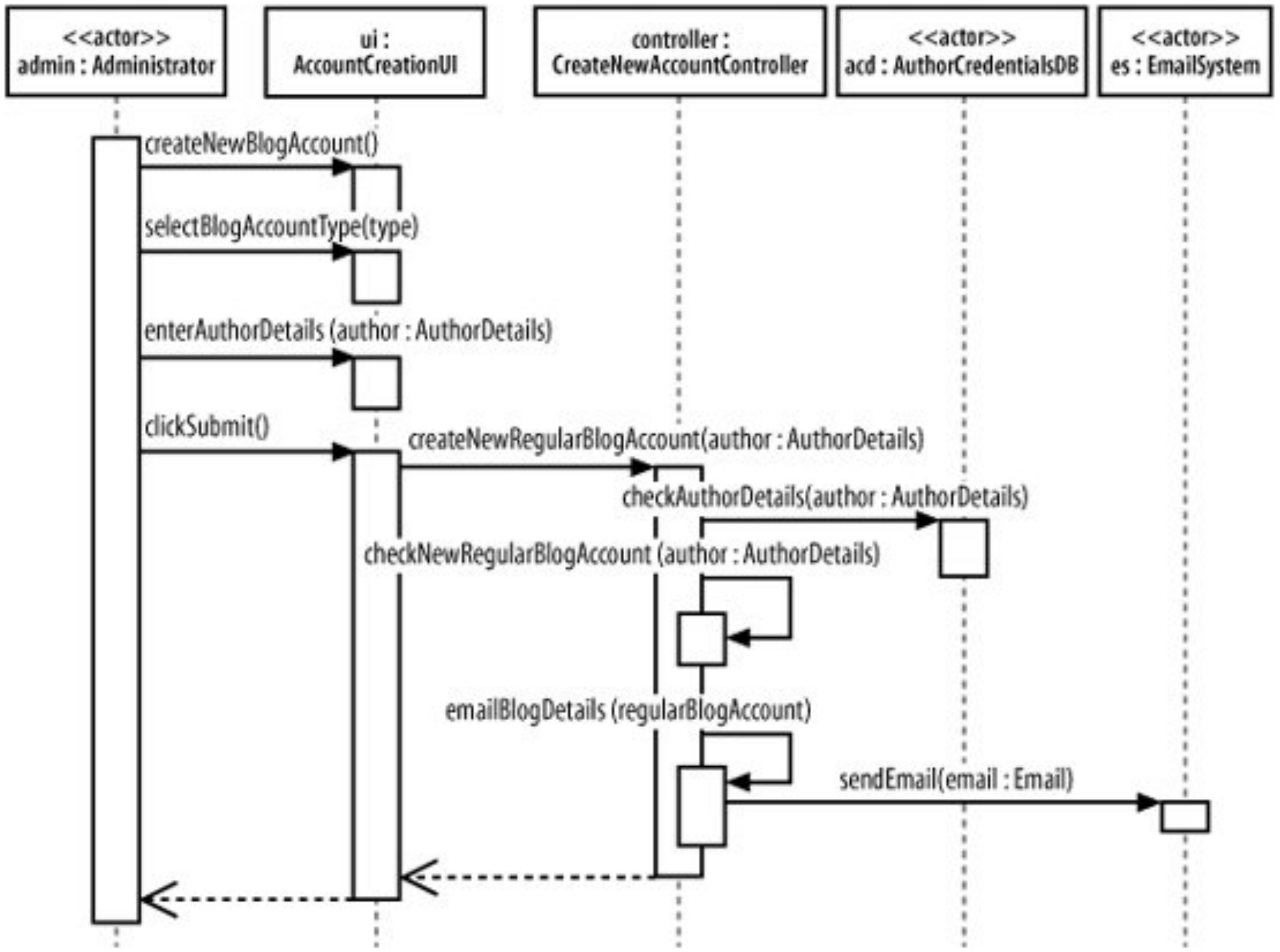

<<actor>> admin : Administrator
ui : AccountCreationUI
controller : CreateNewAccountController
<<actor>> acd : AuthorCredentialsDB
<<actor>> es : EmailSystem
createNewBlogAccount()
selectBlogAccountType(type)
enterAuthorDetails (author : AuthorDetails)
clickSubmit()
createNewRegularBlogAccount(author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

Figure 7-15. Adding more detail about the internals of your system

7.7.3. Applying Participant Creation

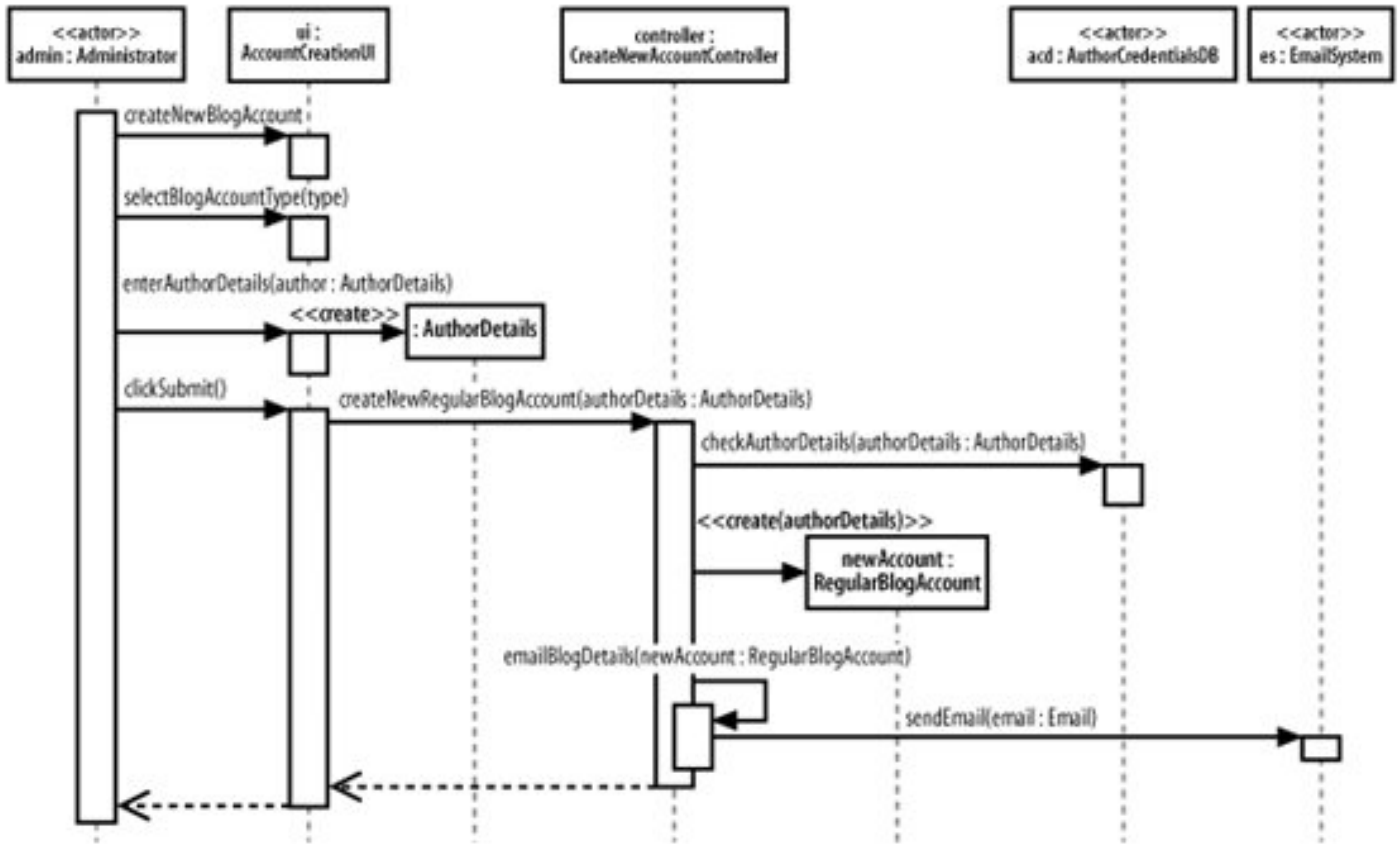

<<actor>> admin : Administrator
ui : AccountCreationUI
controller : CreateNewAccountController
<<actor>> acd : AuthorCredentialsDB
<<actor>> es : EmailSystem
createNewBlogAccount
selectBlogAccountType(type)
enterAuthorDetails(author : AuthorDetails)
<<create>>
: AuthorDetails
clickSubmit()
createNewRegularBlogAccount(authorDetails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

Figure 7-16. Showing the lifelines of your sequence diagram's participants

7.7.4. Applying Participant Deletion

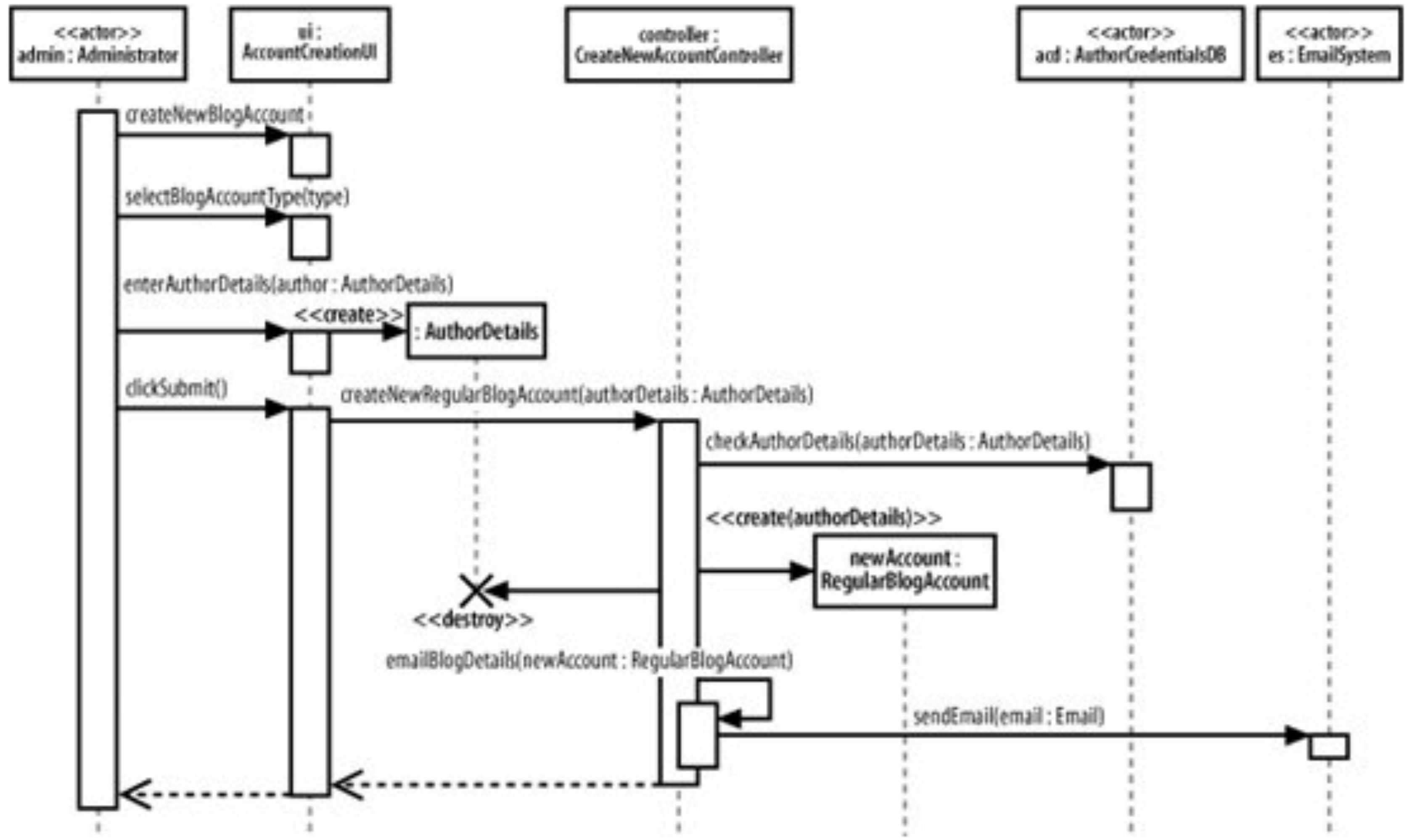
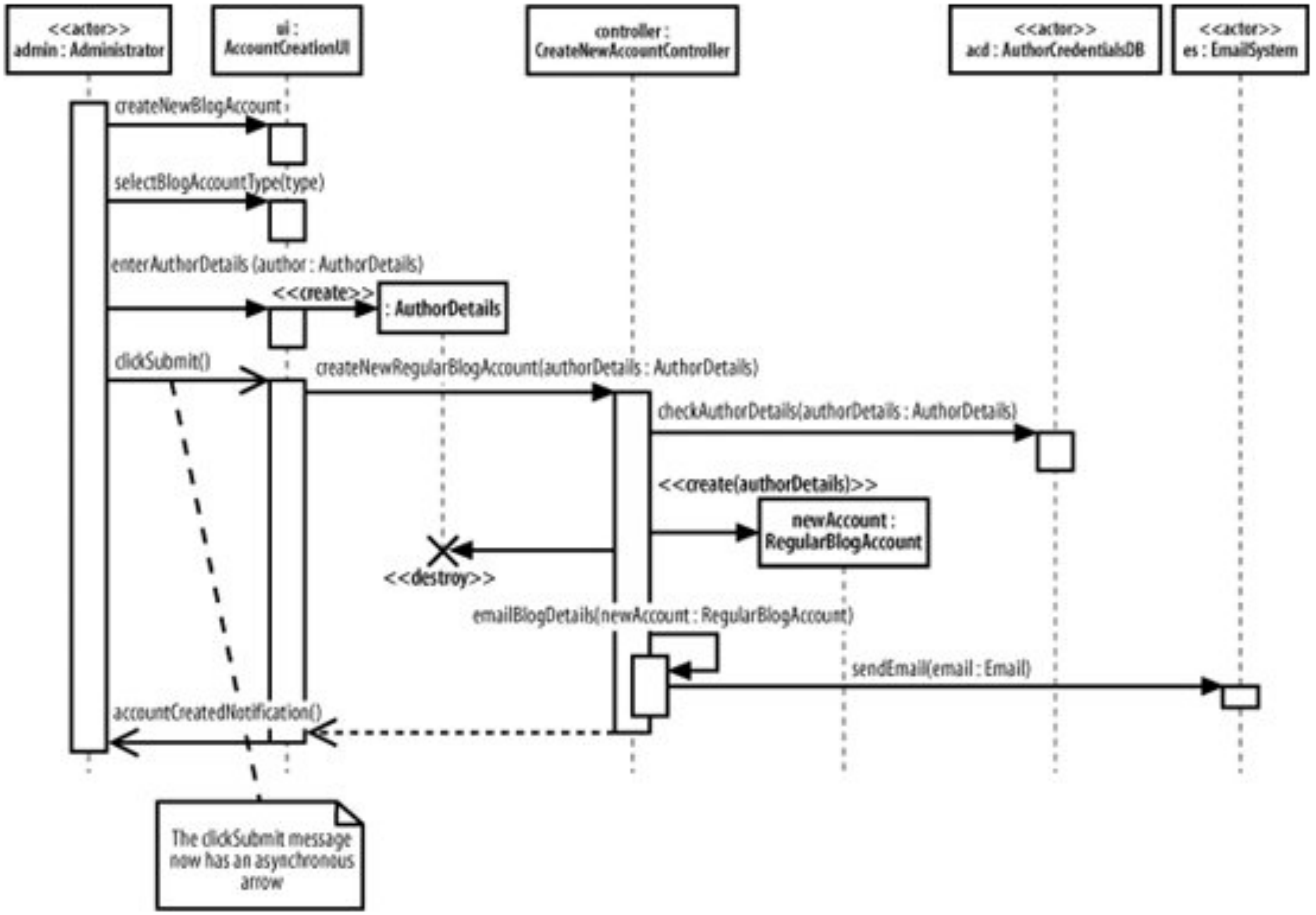

<<actor>>
admin : Administrator
ui : AccountCreationUI
controller : CreateNewAccountController
<<actor>>
acd : AuthorCredentialsDB
<<actor>>
es : EmailSystem
createNewBlogAccount
selectBlogAccountType(type)
enterAuthorDetails(author : AuthorDetails)
<<create>>
: AuthorDetails
clickSubmit()
createNewRegularBlogAccount(authorDetails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

Figure 7-17. Showing that a participant is discarded using the destruction cross

7.7.5. Applying Asynchronous Messages



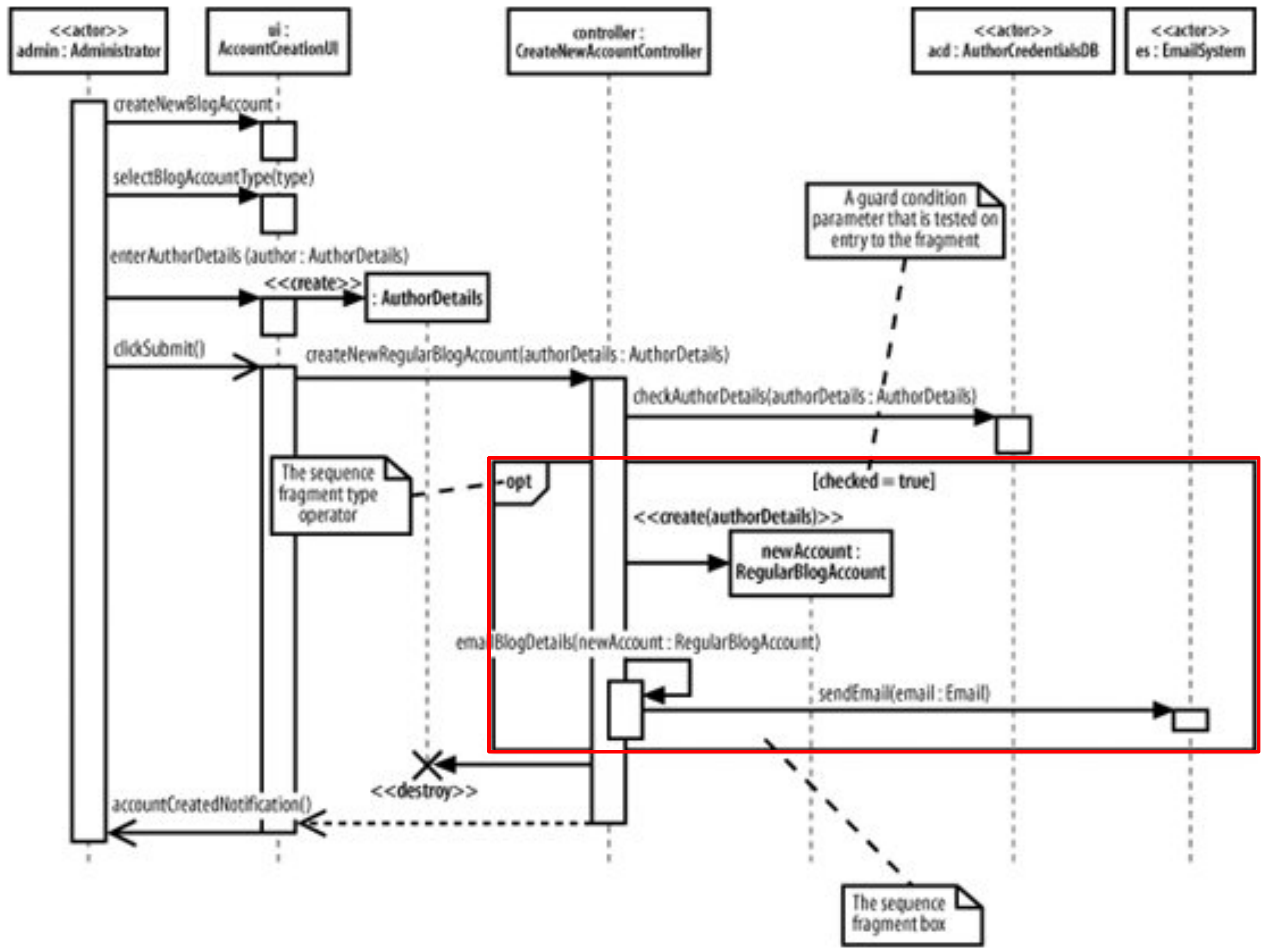
<<actor>> admin : Administrator
ui : AccountCreationUI
controller : CreateNewAccountController
<<actor>> acd : AuthorCredentialsDB
<<actor>> es : EmailSystem
createNewBlogAccount
selectBlogAccountType(type)
enterAuthorDetails (author : AuthorDetails)
<<create>>
: AuthorDetails
clickSubmit()
createNewRegularBlogAccount(authorDetails : AuthorDetails)
checkAuthorDetails(authorDetails : AuthorDetails)
<<create(authorDetails)>>
newAccount : RegularBlogAccount
<<destroy>>
emailBlogDetails(newAccount : RegularBlogAccount)
sendEmail(email : Email)
accountCreatedNotification()
The clickSubmit message now has an asynchronous arrow

Figure 7-18. The clickSubmit( ) message will currently produce some irregular behavior when the admin creates a new account

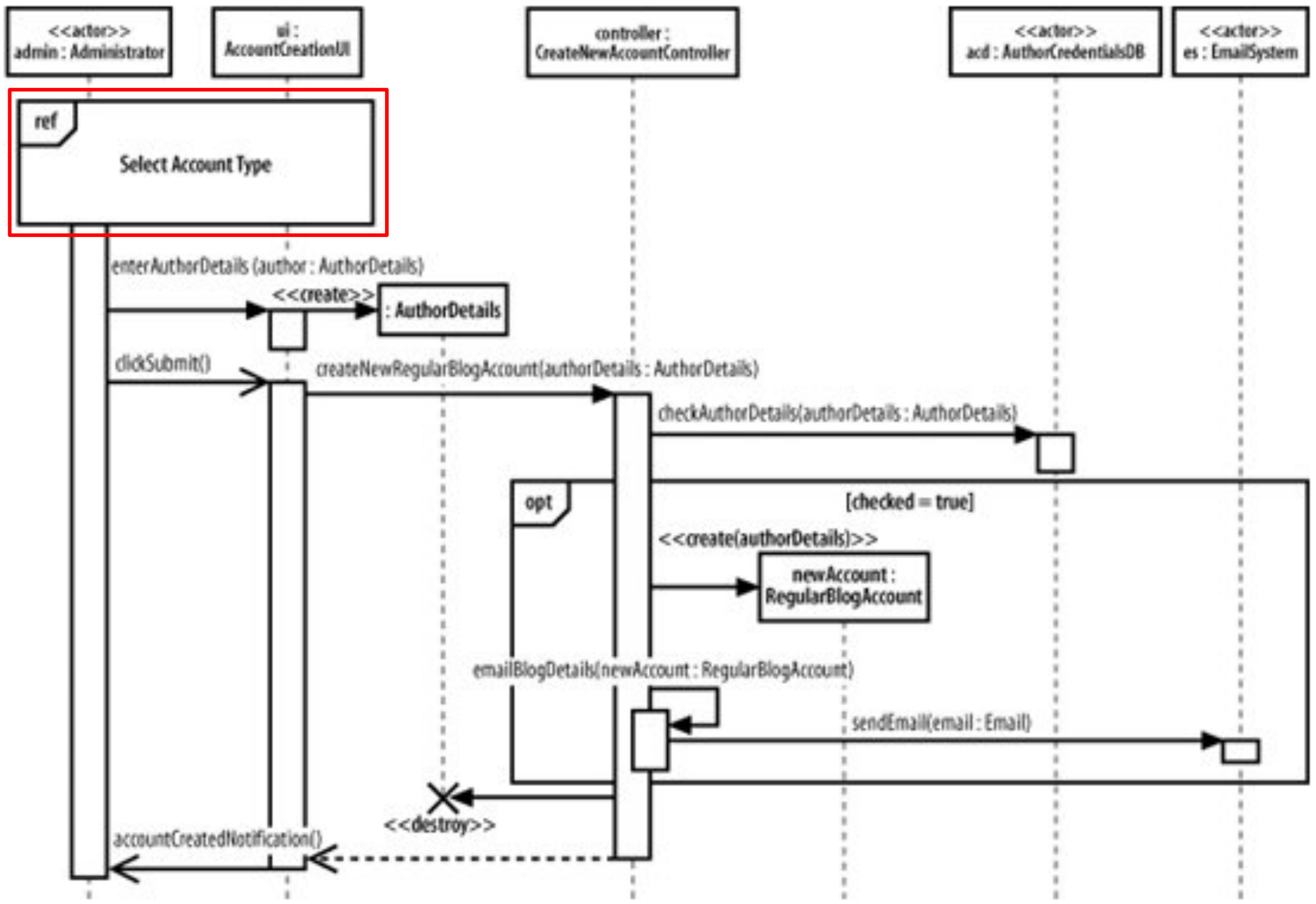
7.8. Managing Complex Interactions with Sequence Fragments 顺序图片断

- A sequence fragment is represented as a box that encloses a portion of the interactions within a sequence diagram, as shown in Figure 7-19.
- A sequence fragment's box overlaps the region of the sequence diagram where the fragment's interactions take place.
- A fragment box can contain any number of interactions and, for large complex interactions, further nested fragments as well.
- The top left corner of the fragment box contains an operator. The fragment operator indicates which type of fragment this is.

Figure 7-19. A sequence fragment located as part of a larger sequence diagram, with notes to indicate the fragment box, any parameters, and its operator

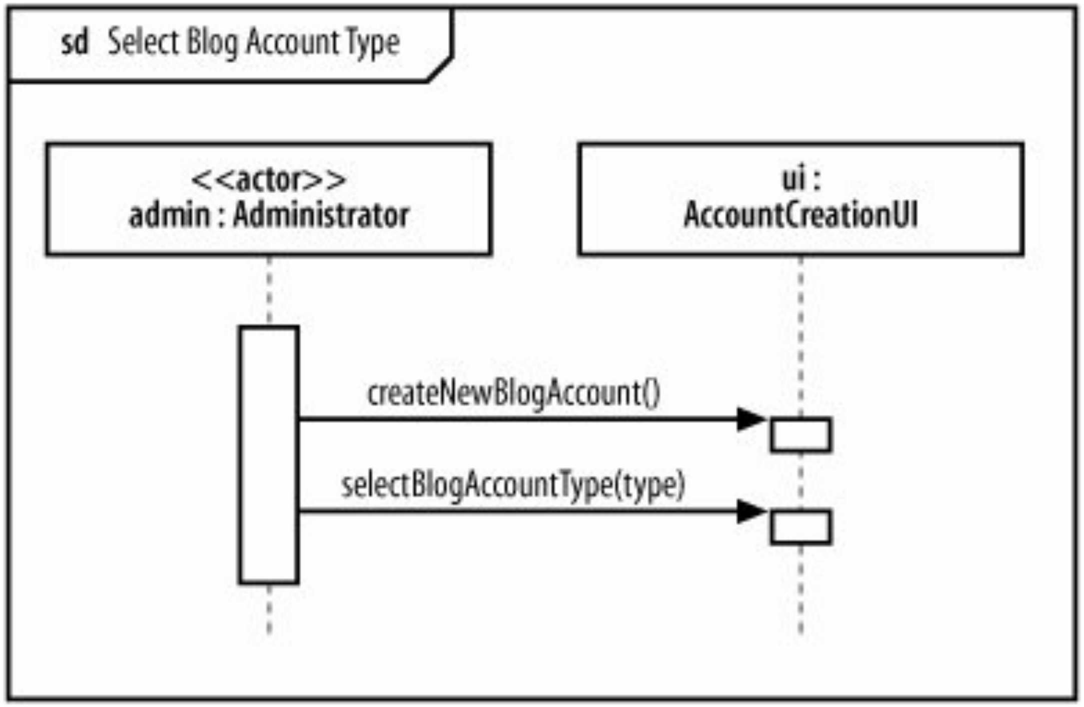

<<actor>> admin : Administrator
ui : AccountCreationUI
controller : CreateNewAccountController
<<actor>> acd : AuthorCredentialsDB
<<actor>> es : EmailSystem
createNewBlogAccount
selectBlogAccountType(type)
enterAuthorDetails (author : AuthorDetails)
<<create>>
: AuthorDetails
clickSubmit()
createNewRegularBlogAccount(authorDetails : AuthorDetails)
checkAuthorDetails(authorDetails : AuthorDetails)
A guard condition parameter that is tested on entry to the fragment
The sequence fragment type operator
opt
[checked = true]
<<create(authorDetails)>>
newAccount : RegularBlogAccount
emailBlogDetails(newAccount : RegularBlogAccount)
sendEmail(email : Email)
<<destroy>>
accountCreatedNotification()
The sequence fragment box

7.8.1. Using a Sequence Fragment: The **ref** Fragment



<<actor>> admin : Administrator
ui : AccountCreationUI
controller : CreateNewAccountController
<<actor>> acd : AuthorCredentialsDB
<<actor>> es : EmailSystem
ref
Select Account Type
enterAuthorDetails (author : AuthorDetails)
<<create>>
: AuthorDetails
clickSubmit()
createNewRegularBlogAccount(authorDetails : AuthorDetails)
checkAuthorDetails(authorDetails : AuthorDetails)
opt
[checked = true]
<<create(authorDetails)>>
newAccount : RegularBlogAccount
emailBlogDetails(newAccount : RegularBlogAccount)
sendEmail(email : Email)
<<destroy>>
accountCreatedNotification()

Figure 7-21. A referenced sequence diagram that contains the new account selection interactions



7.8.2. A Brief Overview of UML 2.0's Fragment Types

ref	assert	loop
break	alt	opt
neg	par	region

Type	Parameters
ref	None
assert	None
loop	min times, max times, [guard_condition]
break	None
alt	[guard_condition1] ... [guard_condition2] ... [else]
opt	[guard_condition]
neg	None
par	None
region	None

Summary

- 6. Bringing Your Classes to Life: Object Diagrams
  - 6.1. Object Instances
  - 6.2. Links
  - 6.3. Binding Class Templates
- 7. Modeling Ordered Interactions: Sequence Diagrams
  - 7.1. Participants in a Sequence Diagram
  - 7.2. Time
  - 7.3. Events, Signals, and Messages
  - 7.4. Activation Bars
  - 7.5. Nested Messages
  - 7.6. Message Arrows
  - 7.7. Bringing a Use Case to Life with a Sequence Diagram
  - 7.8. Managing Complex Interactions with Sequence Fragments

Next ……

- 8. Focusing on Interaction Links: Communication Diagrams 会话图
  - 8.1. Participants, Links, and Messages 参与者、链接与消息
  - 8.2. Fleshing out an Interaction with a Communication Diagram
  - 8.3. Communication Diagrams Versus Sequence Diagrams

See you …