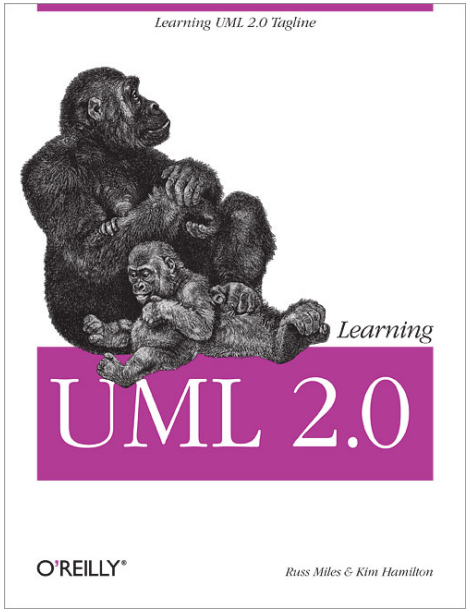
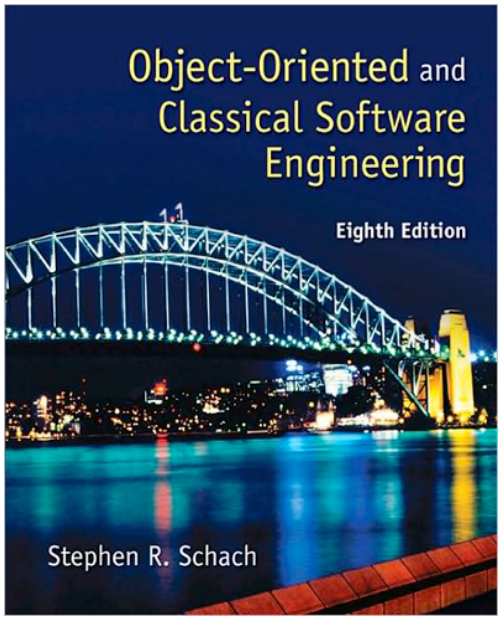
CMPE 202

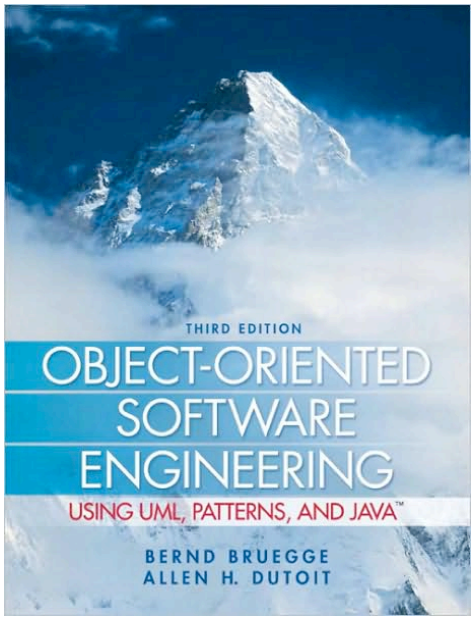
Introduction to UML

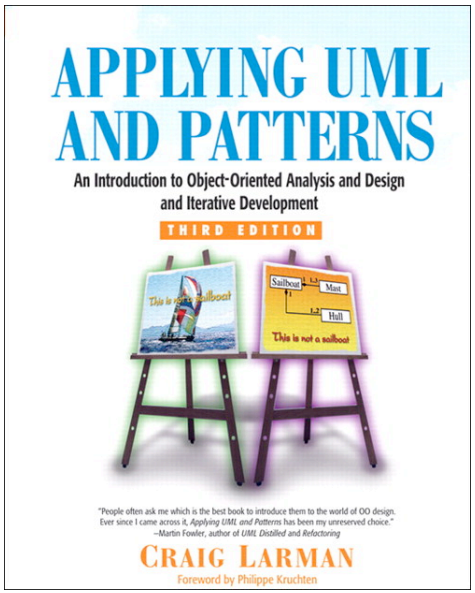
Sequence Diagrams

Resources for this talk









Sequence Diagrams

Why Sequence Diagrams?

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. There's one large piece that's missing from this jigsaw; with use cases and classes alone, you can't yet model *how* your system is actually going to its job. This is where interaction diagrams, and specifically sequence diagrams, come into play.

Participants

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. Where a participant is placed on a sequence diagram is important. Regardless of where a participant is placed vertically, participants are always arranged horizontally with no two participants overlapping each other, as shown in Figure 7-2.

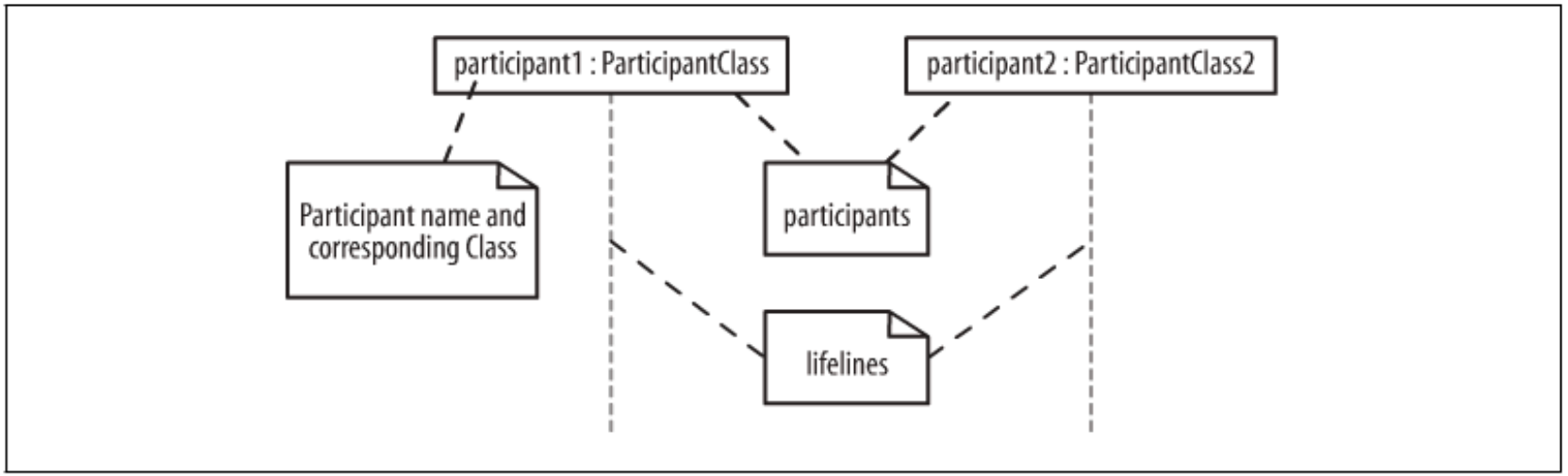


Figure 7-2. At its simplest, a sequence diagram is made up of one or more participants—only one participant would be a very strange sequence diagram, but it would be perfectly legal UML

Participant Names

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
```

Table 7-1. How to understand the components of a participant's name

Example participant name	Description
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`.
: ContentManagementSystem ref cmsInteraction	The participant is of the class `ContentManagementSystem`, and there is another interaction diagram called `cmsInteraction` that shows how the participant works internally (see "A Brief Overview of UML 2.0's Fragment Types," later in this chapter).

Time in Sequence Diagrams

Time

A sequence diagram describes the order in which the interactions take place, so time is an important factor. How time relates to a sequence diagram is shown in Figure 7-3.

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

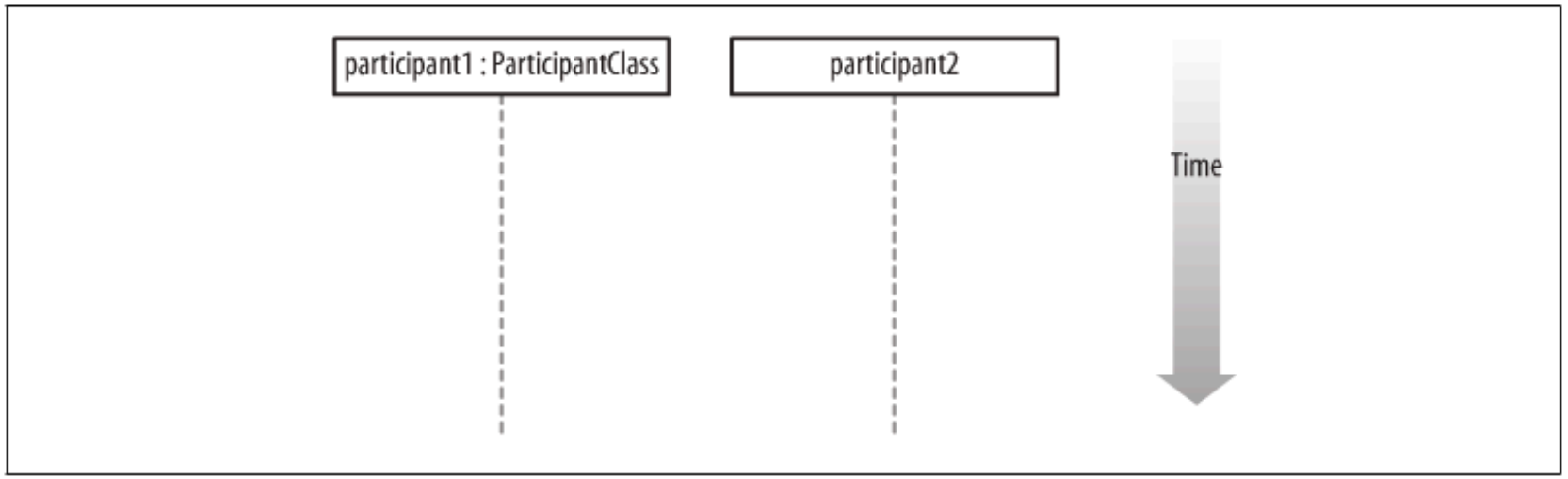


Figure 7-3. Time runs down the page on a sequence diagram in keeping with the participant lifeline

Messages

Events, Signals, and Messages

The smallest part of an interaction is an event. An *event* is any point in an interaction where something occurs, as shown on Figure 7-4.

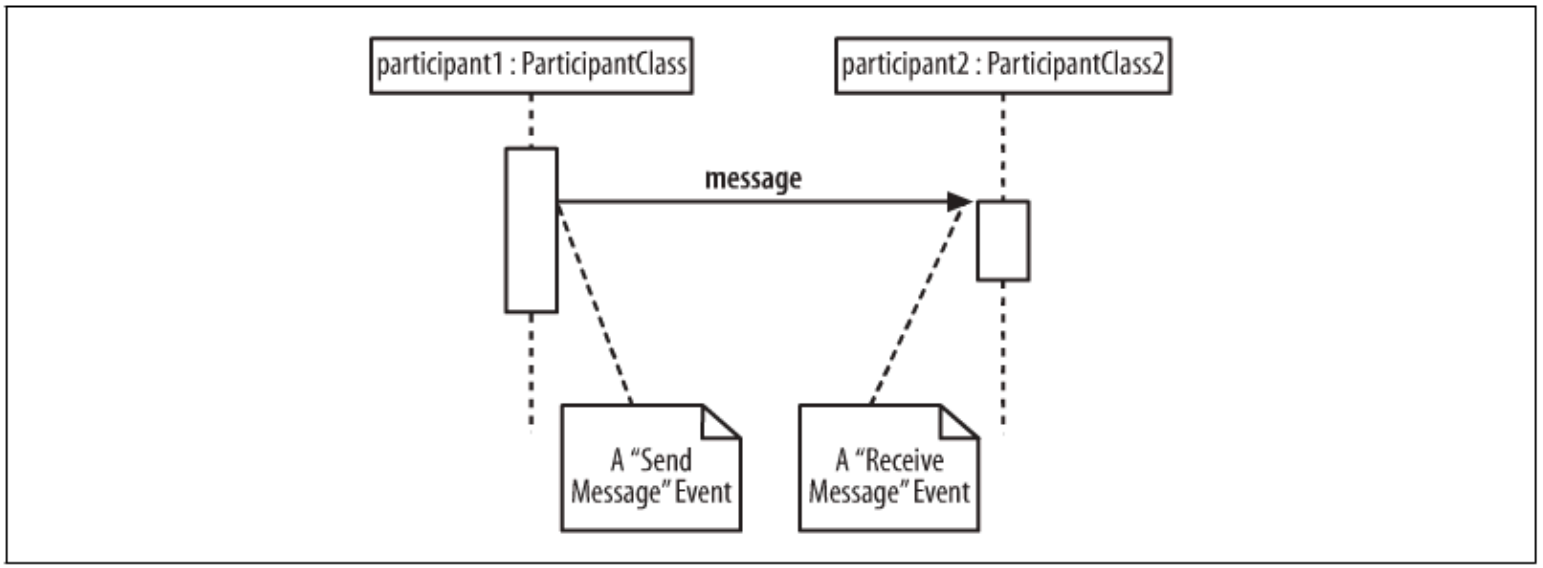


Figure 7-4. Probably the most common examples of events are when a message or signal is sent or received

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.

Interactions

An interaction in a sequence diagram occurs when one participant decides to send a message to another participant, as shown in Figure 7-5.

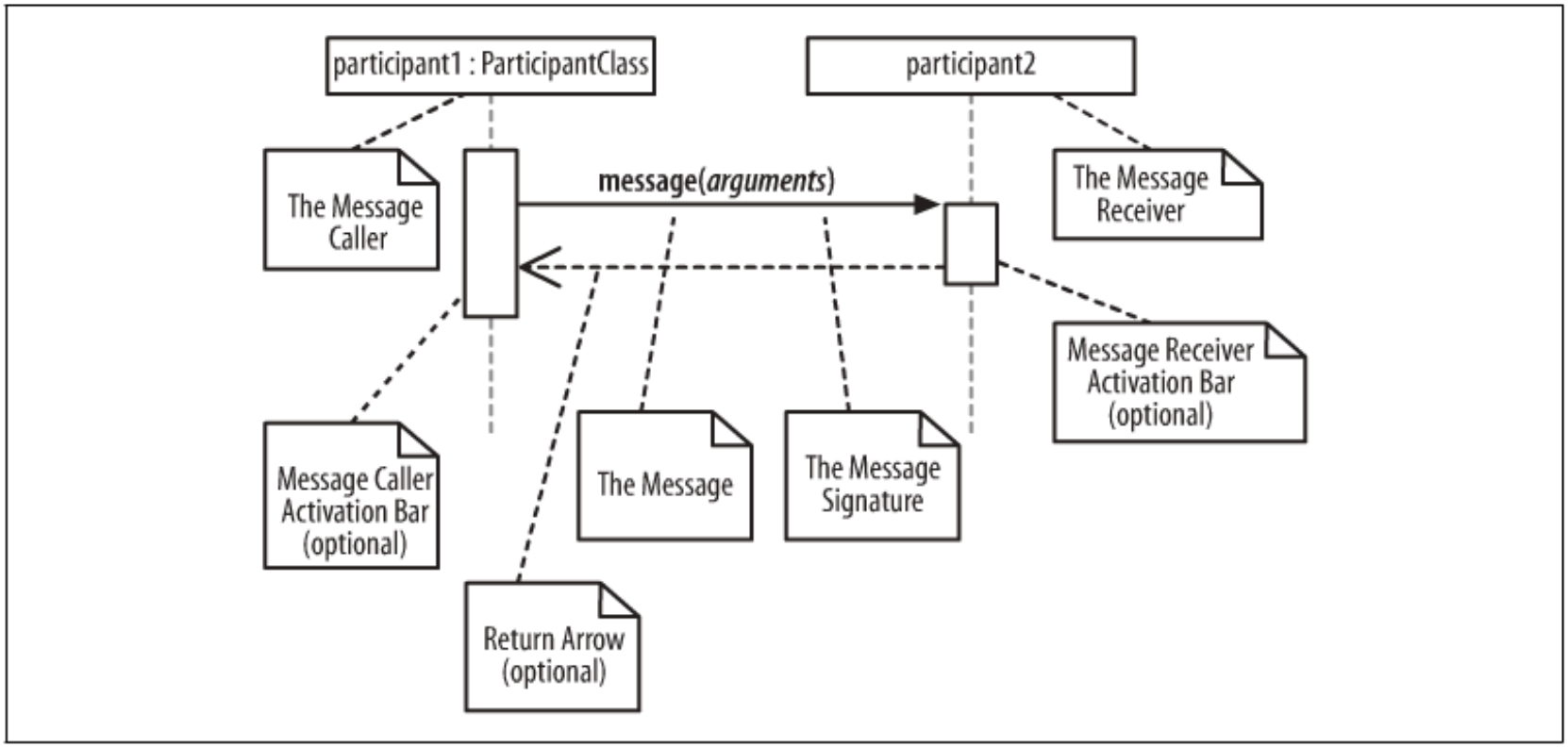


Figure 7-5. Interactions on a sequence diagram are shown as messages between participants

Message Signatures

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>
```

The elements of the format that you use for a particular message will depend on the information known about a particular message at any given time, as explained in Table 7-2.

Table 7-2. How to understand the components of a message's signature

Example message signature	Description
doSomething( )	The message's name is `doSomething`, but no further information is known about it.
doSomething(number1 : Number, number2 : Number)	The message's name is `doSomething`, and it takes two arguments, `number1` and `number2`, which are both of class `Number`.
doSomething( ) : ReturnClass	The message's name is `doSomething`; it takes no arguments and returns an object of class `ReturnClass`.
myVar = doSomething( ) : ReturnClass	The message's name is `doSomething`; it takes no arguments, and it returns an object of class `ReturnClass` that is assigned to the `myVar` attribute of the message caller.

Activation Bars

Activation Bars

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.

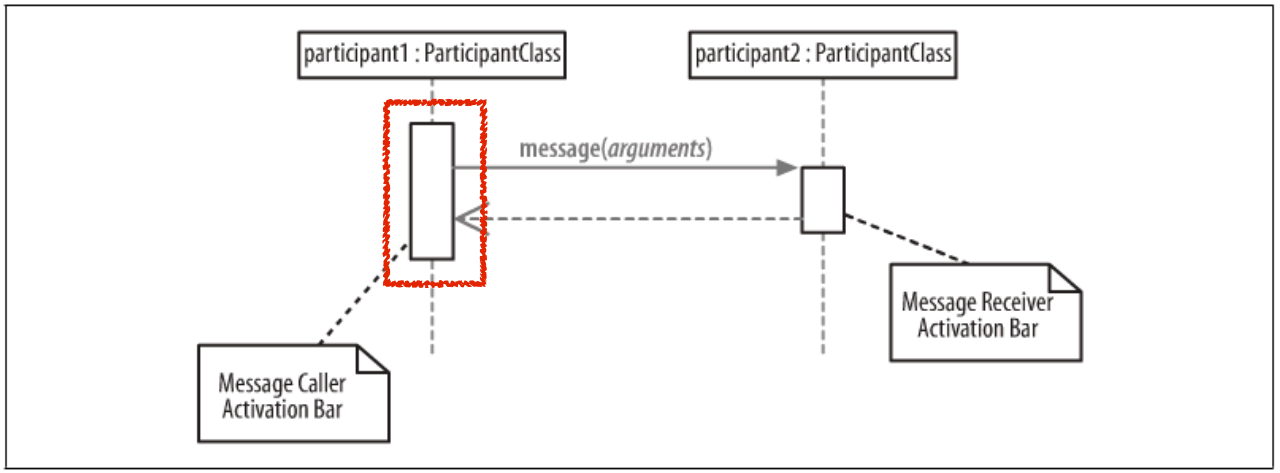


Figure 7-6. Activation bars show that a participant is busy doing something for a period of time

An activation bar can be shown on the sending and receiving ends of a message. 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

Activation bars are optional—they can clutter up a diagram.

Nested Messages

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.

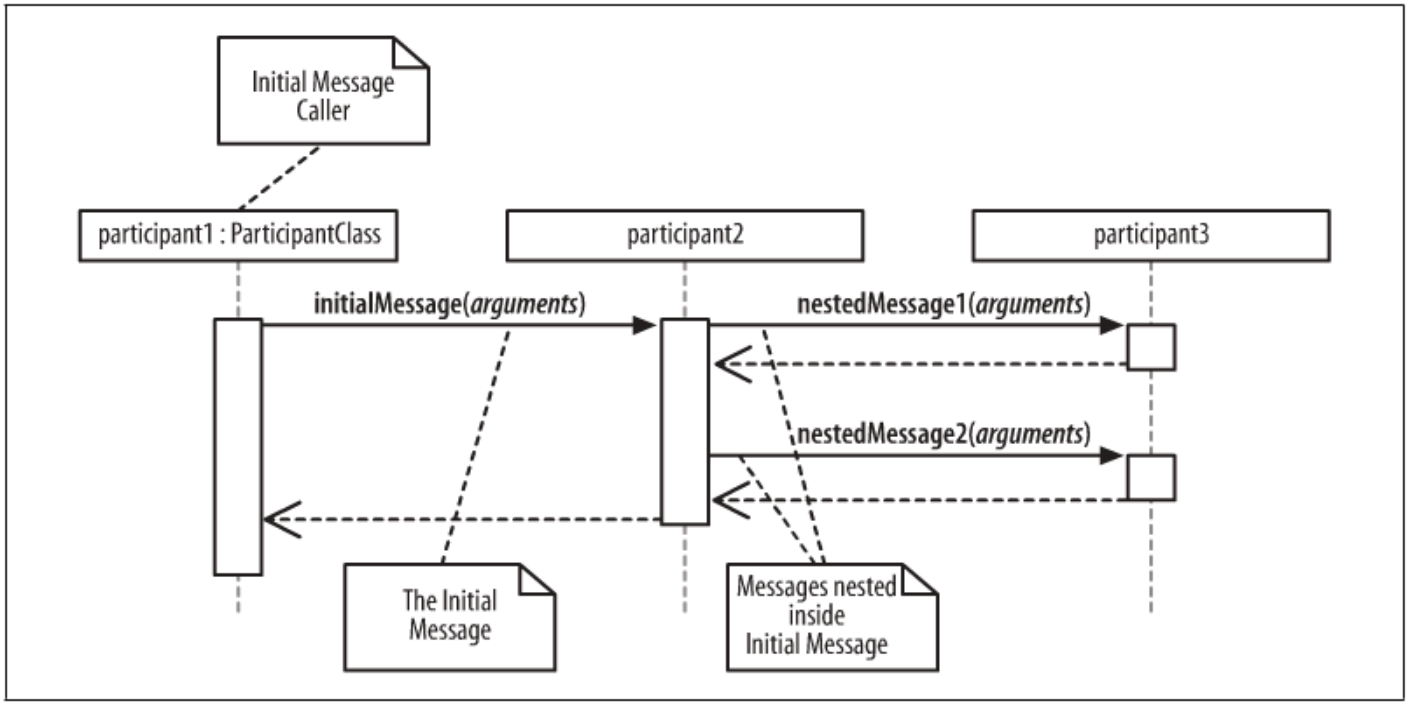


Figure 7-7. Two nested messages are invoked when an initial message is received

Types of Messages

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, as shown in Figure 7-8.

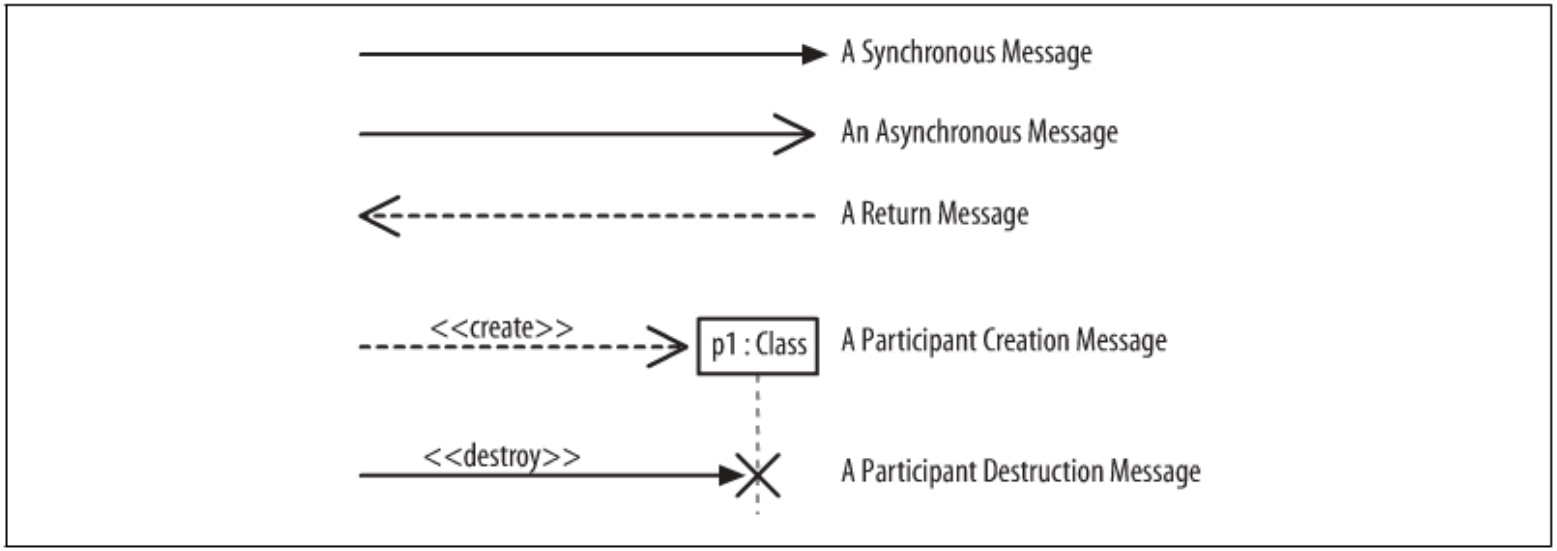


Figure 7-8. There are five main types of message arrow for use on sequence diagram, and each has its own meaning

Realizing Use Cases

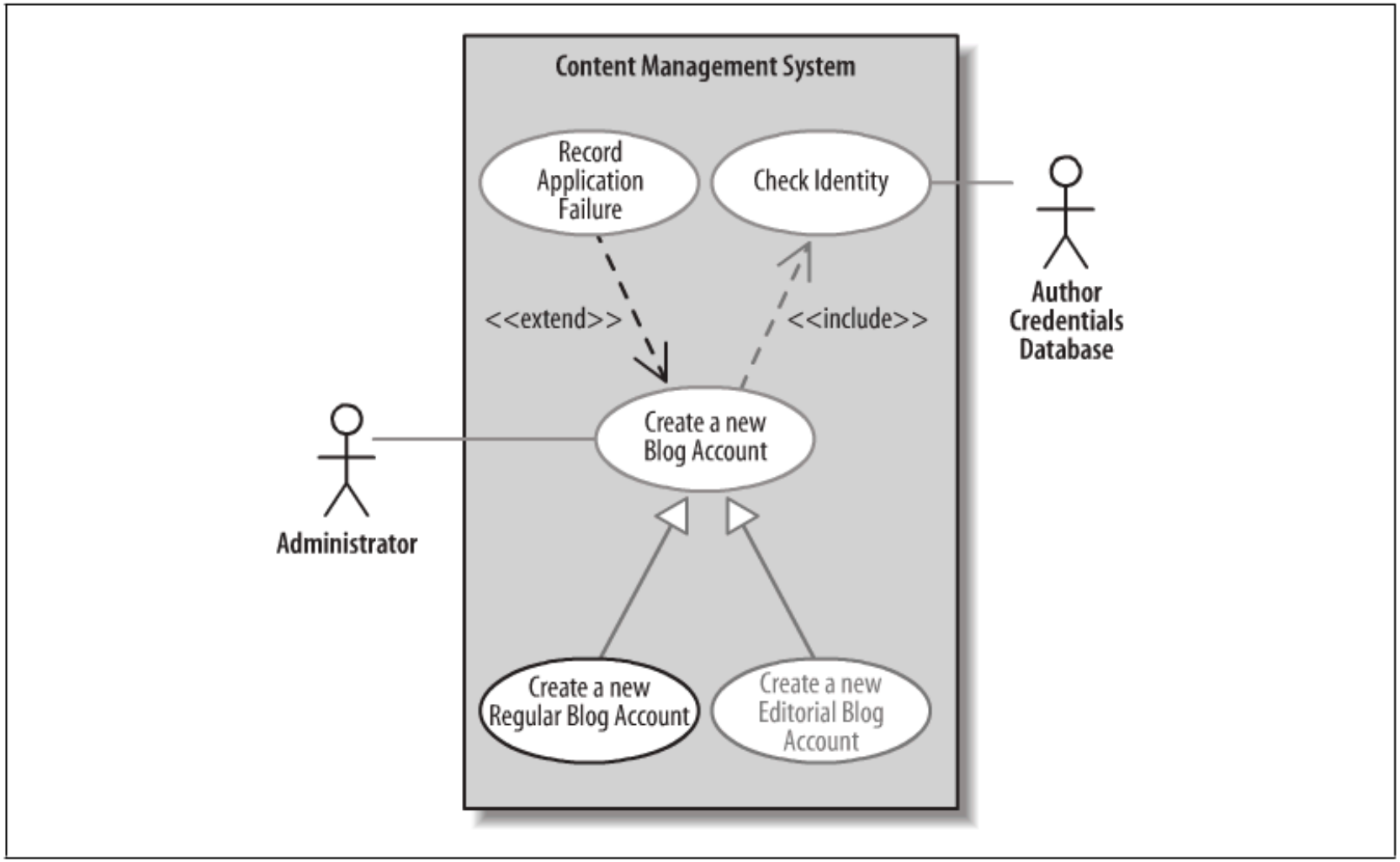


Figure 7-13. The Create a new Regular Blog Account use case diagram

Top-Level Sequence Diagram

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.

Table 7-3 shows the steps that occur in the `Create a new Regular Blog Account` use case according to its detailed description.

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

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 only shows the Main Flow—that is the steps that would occur without worrying about any extensions—but this is a good enough starting point for creating a top-level sequence diagram, as shown in Figure 7-14.

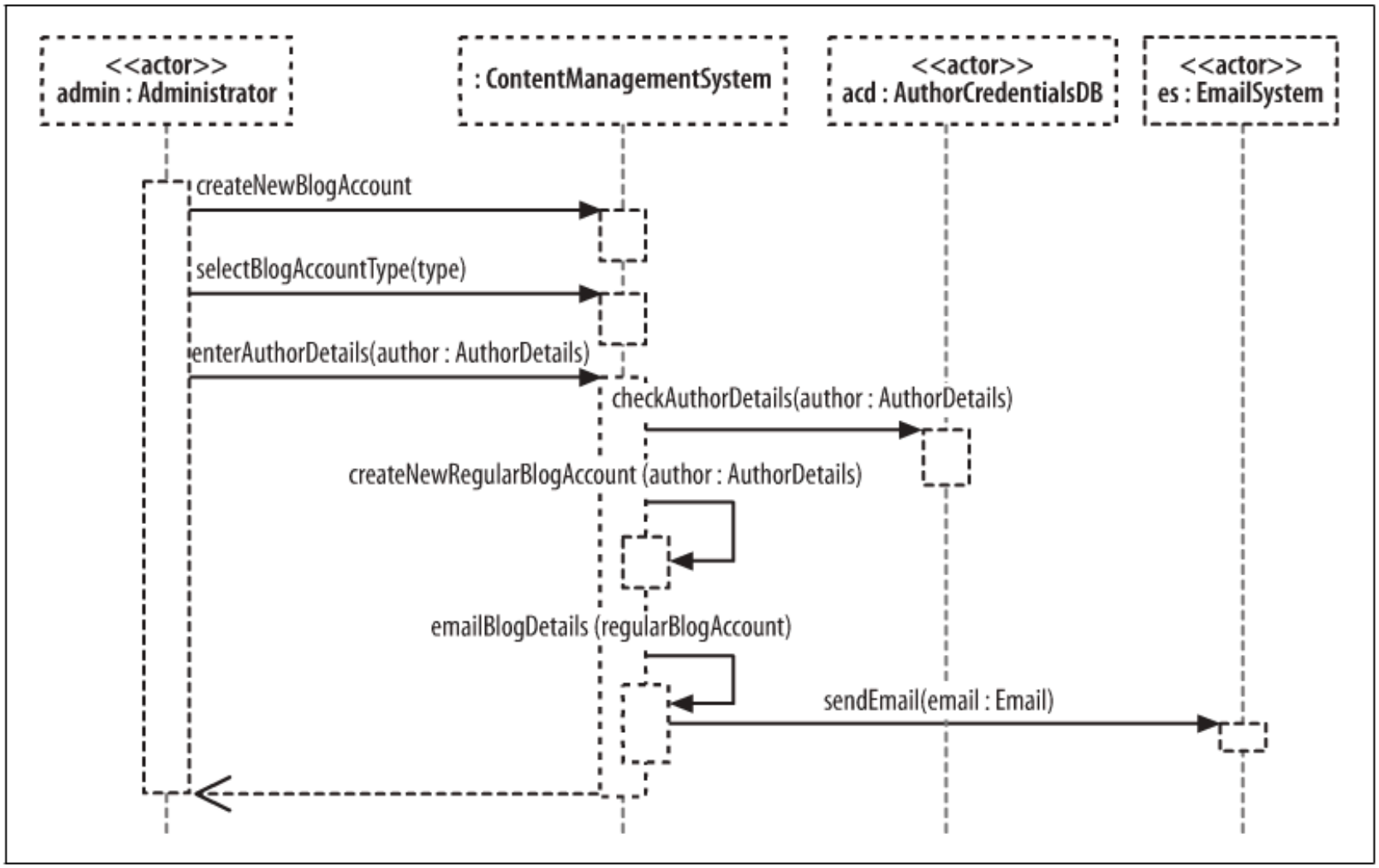


Figure 7-14. This sequence diagram shows the actors that interact with your system and your system is shown simply as a single part in the sequence

Participants Collaborating

Breaking an Interaction into Separate Participants

At this point, Figure 7-14 shows only the interactions that must happen between the external actors and your system because that is the level at which the use case description's steps were written. On the sequence diagram, your system is represented as a single participant, the `ContentManagementSystem`; however, unless you intend on implementing your content management system as a single monolithic piece of code (generally not a good idea!), it's time to break apart `ContentManagementSystem` to expose the pieces that go inside, as shown in Figure 7-15.

More Details - UI & Controller

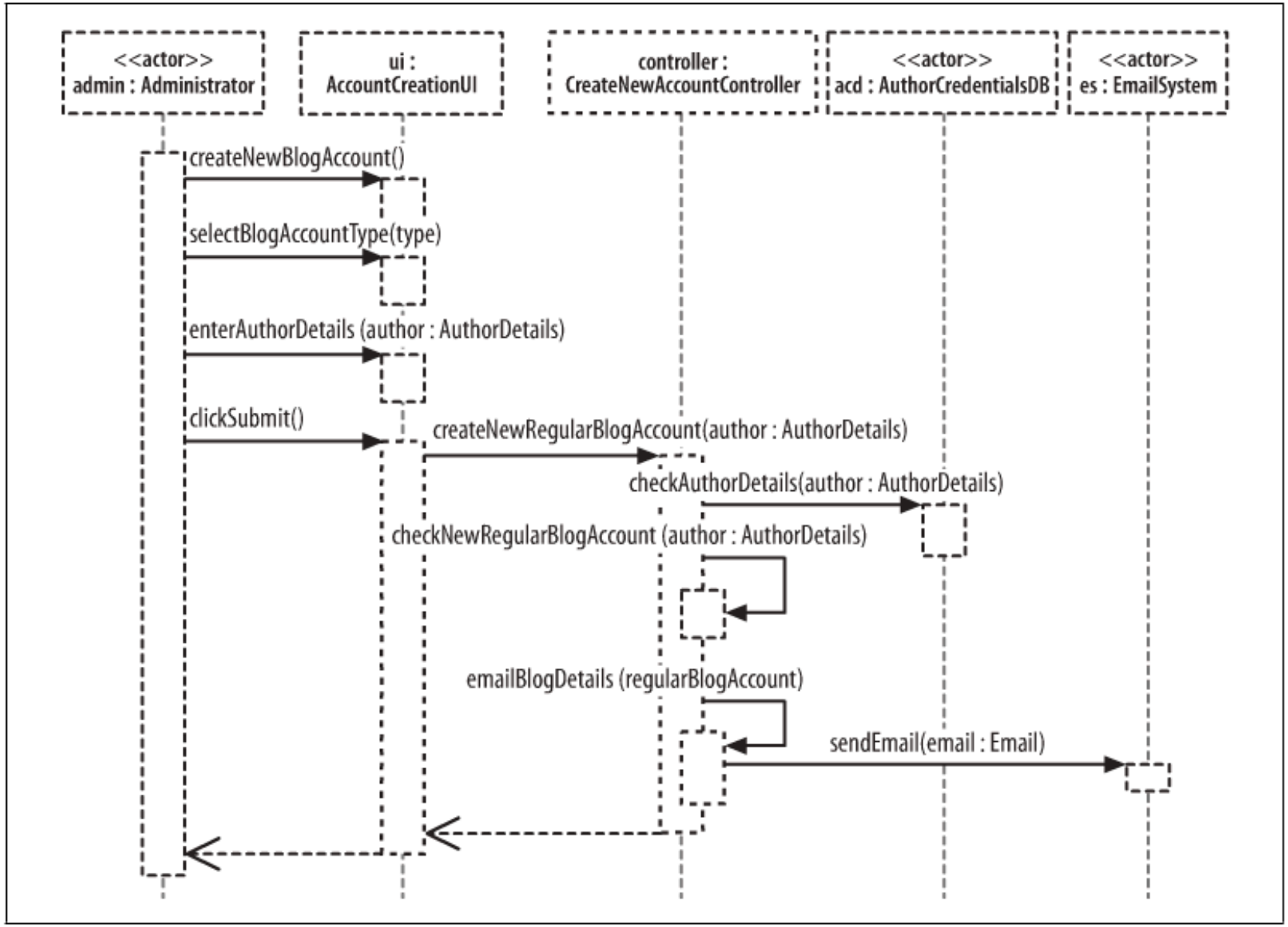


Figure 7-15. Adding more detail about the internals of your system

Applying Participant Creation

Something critical is missing from the sequence diagram shown in Figure 7-15. The title of the use case in which the sequence diagram is operating is `Create a new Regular Blog Account`, but where is the actual *creation* of the blog account? Figure 7-16 adds the missing pieces to the model to show the actual creation of a regular blog account.

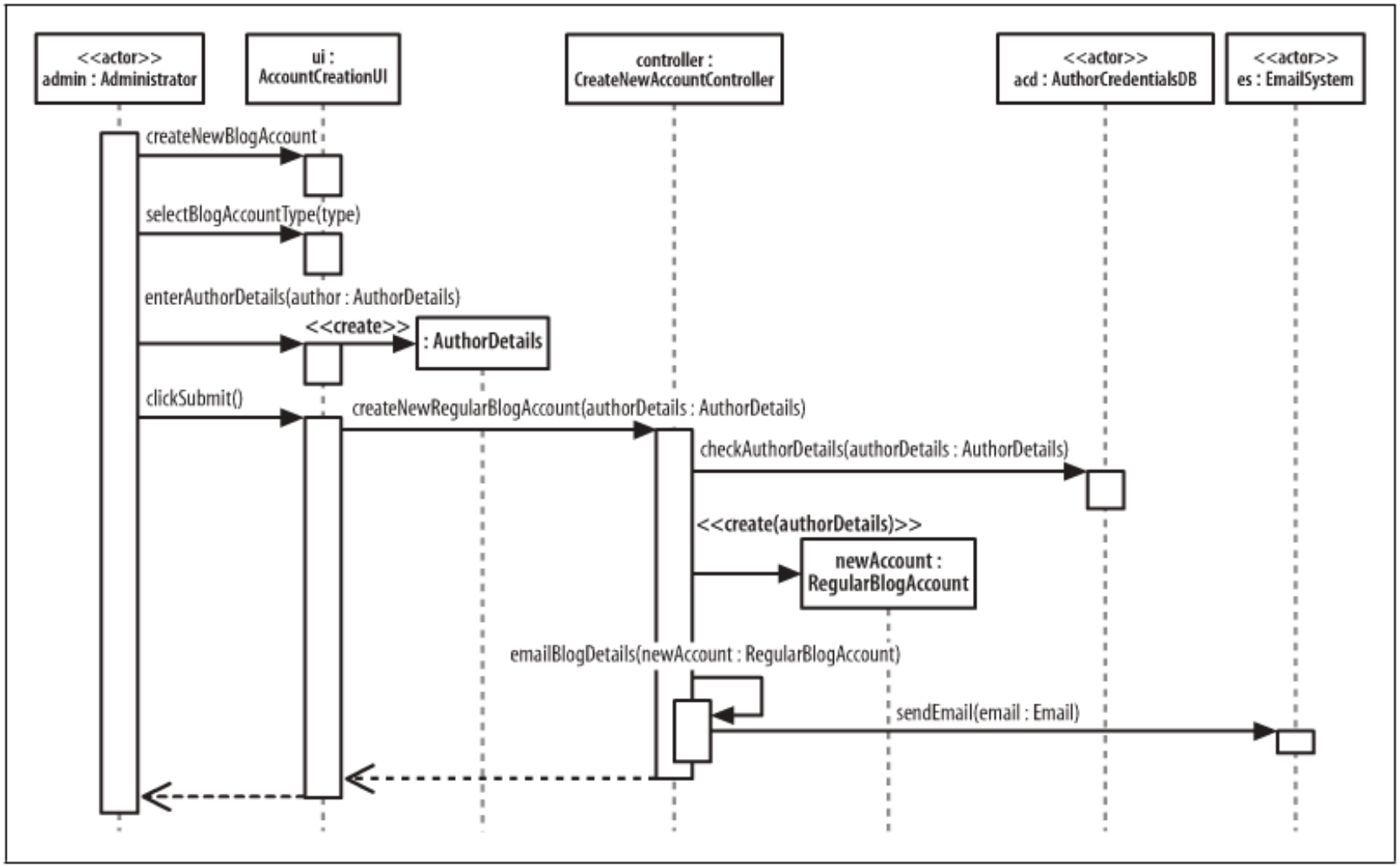


Figure 7-16. Showing the lifelines of your sequence diagram's participants

Applying Participant Deletion

Let's say that the `authorDetails:AuthorDetails` participant is no longer required once the `newAccount:RegularBlogAccount` has been created. To show that the `authorDetails:AuthorDetails` participant is discarded at this point, you can use an explicit `destroy` message connected to the destruction cross, as shown in Figure 7-17.

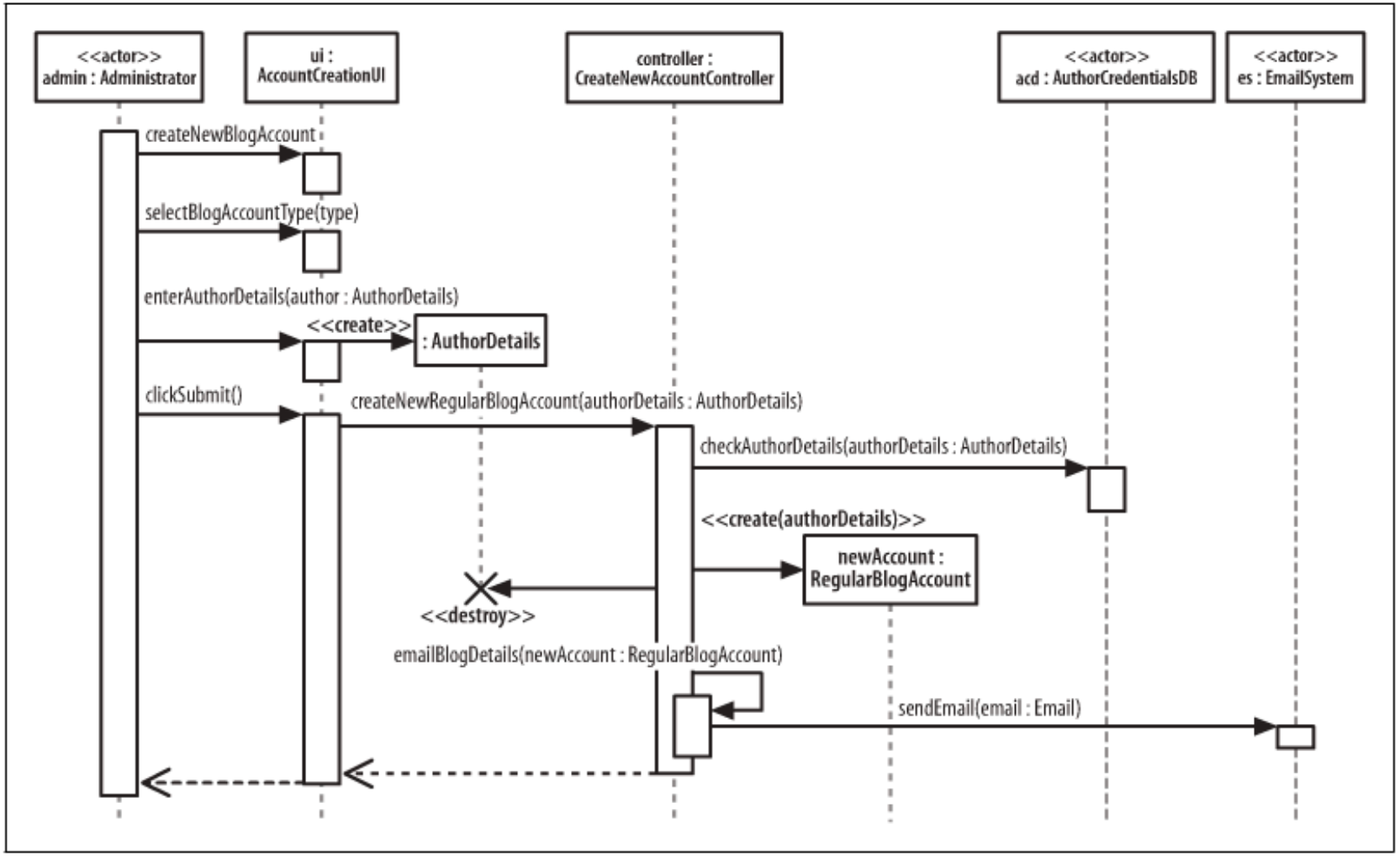


Figure 7-17. Showing that a participant is discarded using the destruction cross

Applying Asynchronous Messages

So far, all of the messages on our example sequence diagram have been synchronous; they are executed one after the other in order, and nothing happens concurrently. However, there is at least one message in the example sequence that is a great candidate for being an asynchronous message, as shown in Figure 7-18.

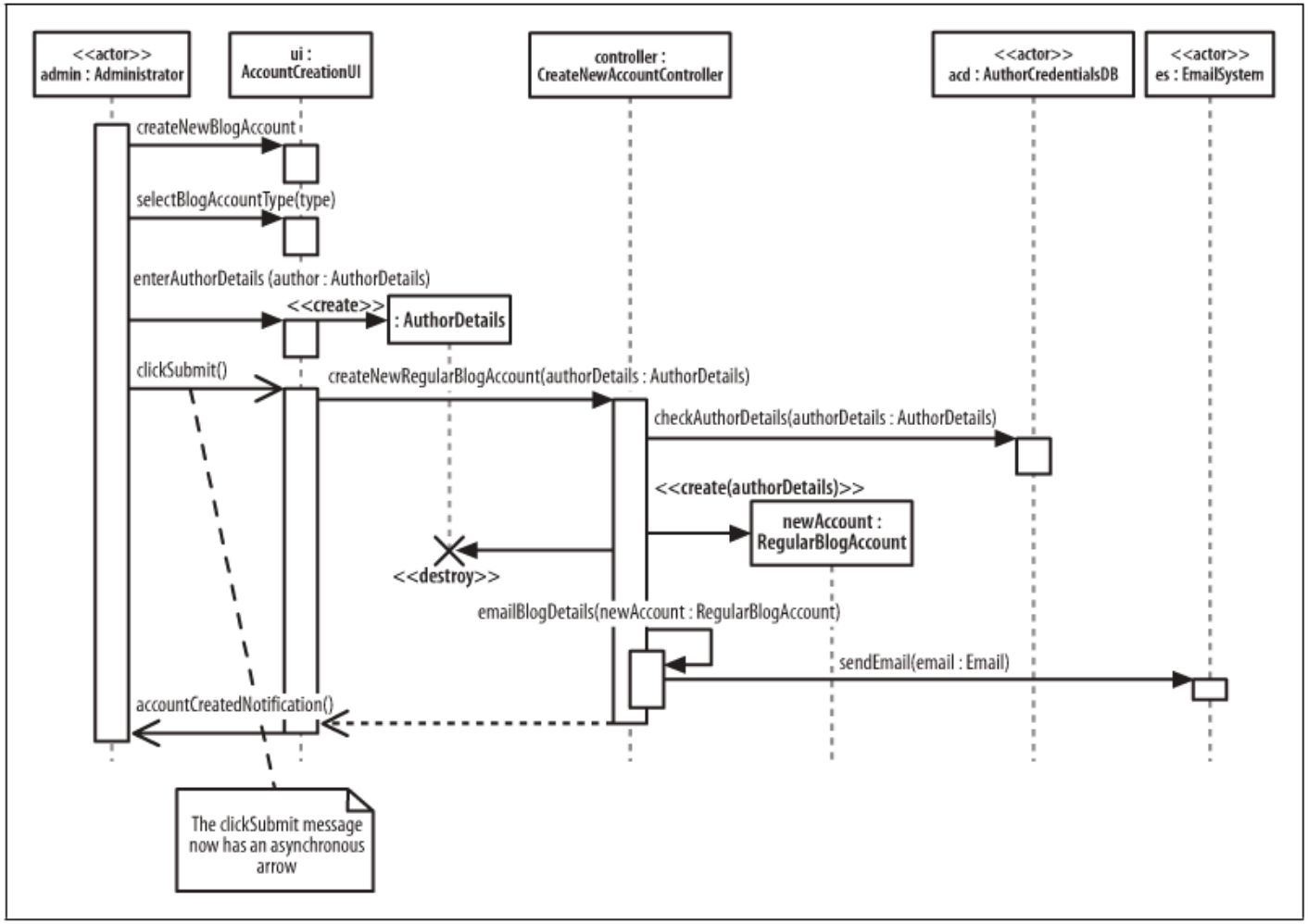
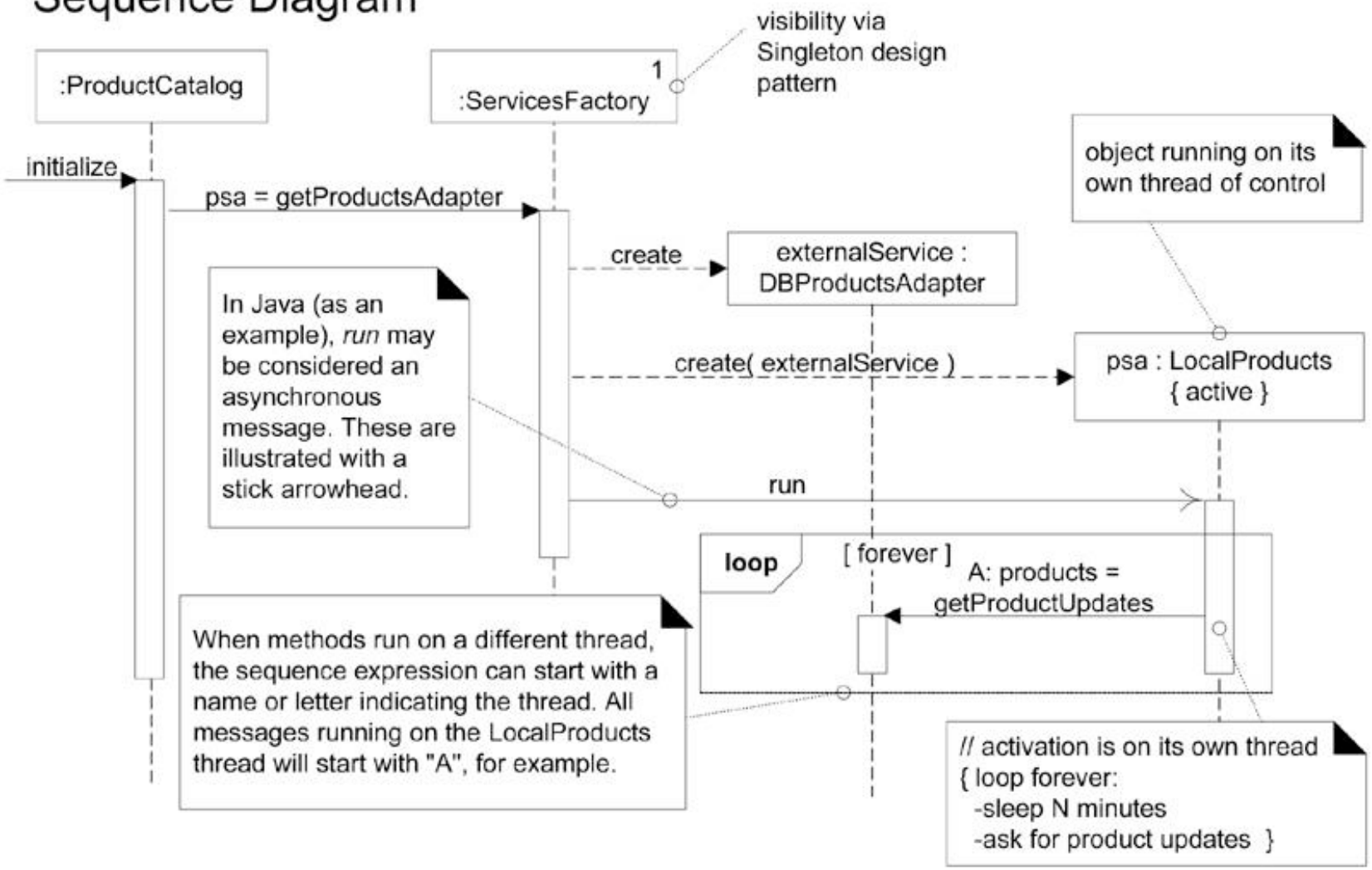


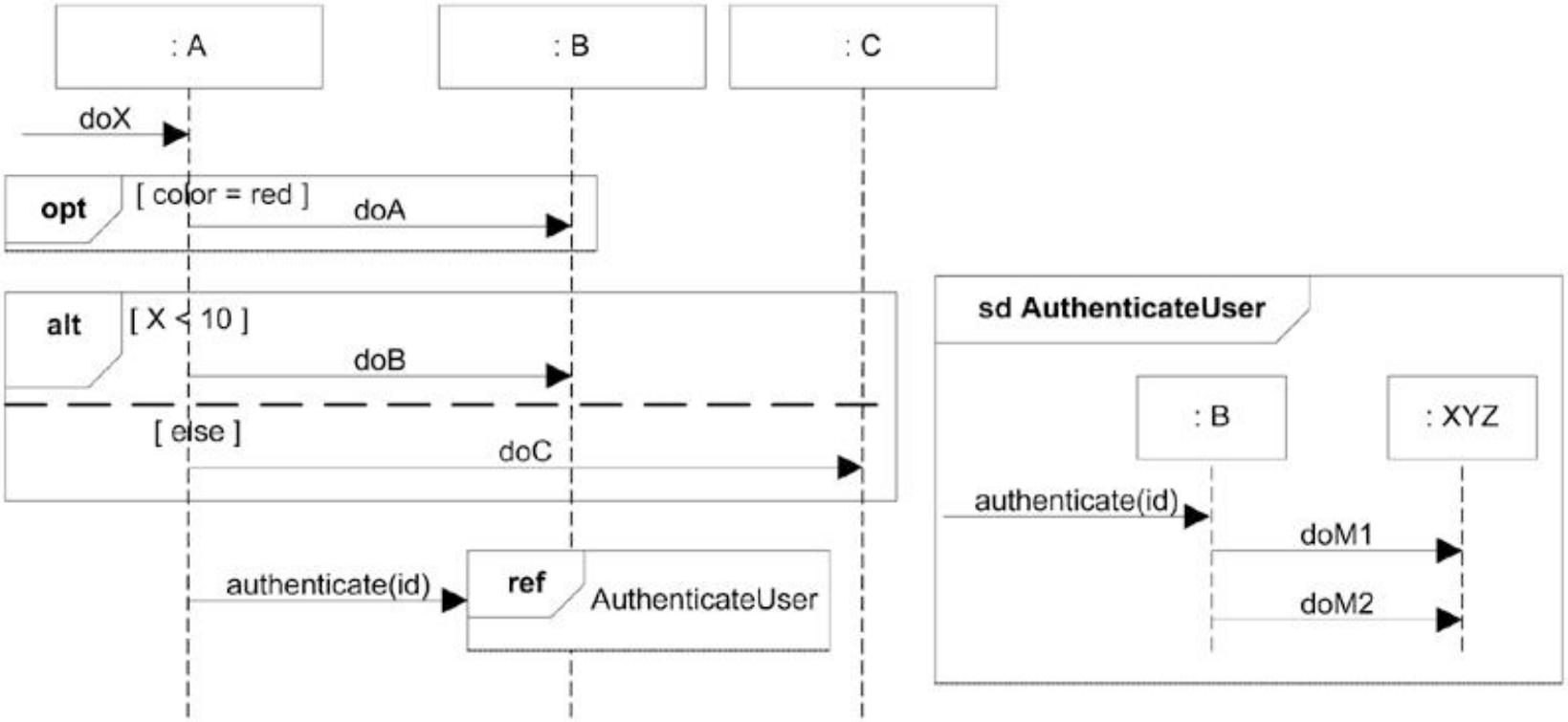
Figure 7-18. The clickSubmit( ) message will currently produce some irregular behavior when the admin creates a new account

Interaction Diagrams

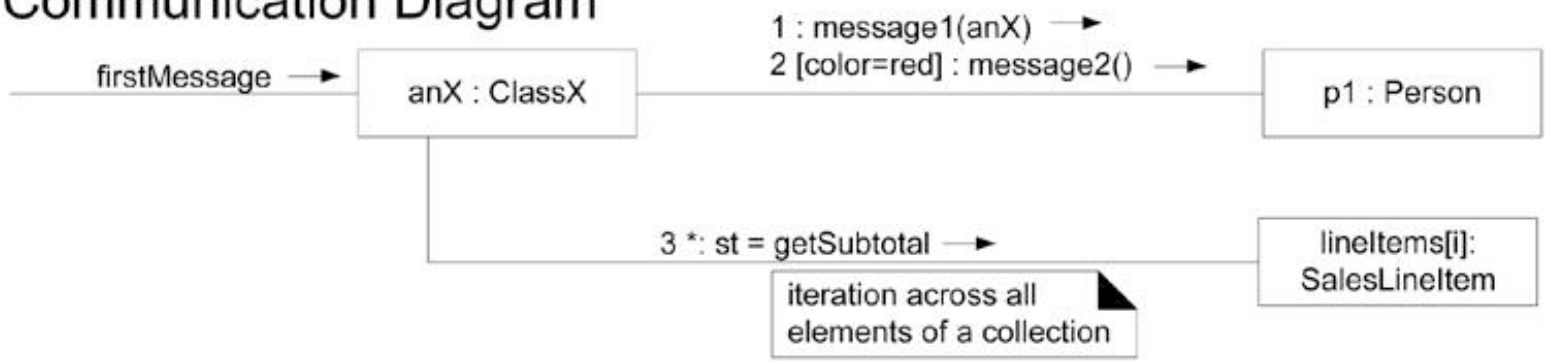
Quick Reference

Sequence Diagram





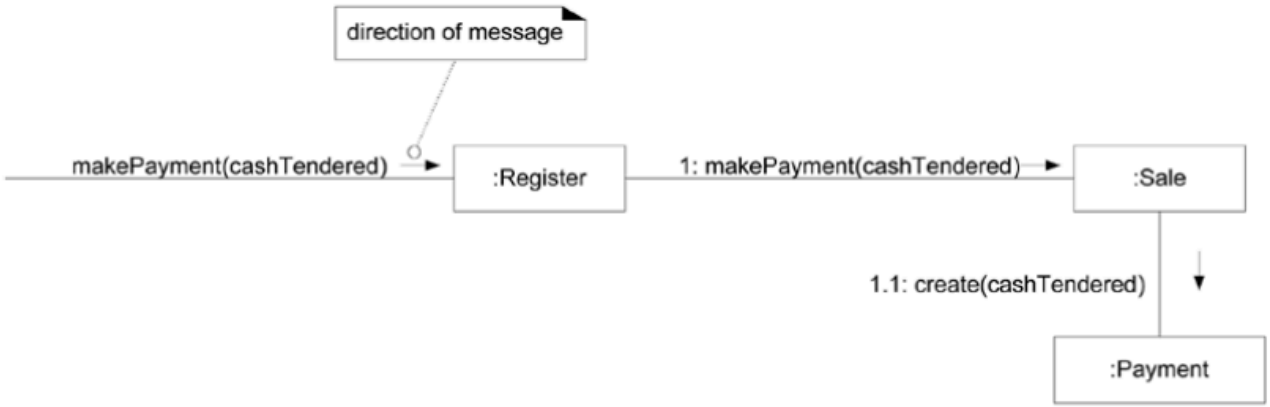
Communication Diagram



Type	Strengths	Weaknesses
sequence	clearly shows sequence or time ordering of messages large set of detailed notation options	forced to extend to the right when adding new objects; consumes horizontal space
communication	space economical—flexibility to add new objects in two dimensions	more difficult to see sequence of messages fewer notation options

Example Sequence Diagram: makePayment

Example Communication Diagram: makePayment



The following table summarizes some common frame operators:

Frame Operator	Meaning
alt	Alternative fragment for mutual exclusion conditional logic expressed in the guards.
loop	Loop fragment while guard is true. Can also write *loop(n)* to indicate looping n times. There is discussion that the specification will be enhanced to define a *FOR* loop, such as *loop(i, 1, 10)*
opt	Optional fragment that executes if guard is true.
par	Parallel fragments that execute in parallel.
region	Critical region within which only one thread can run.

Figure 15.13. A conditional message.

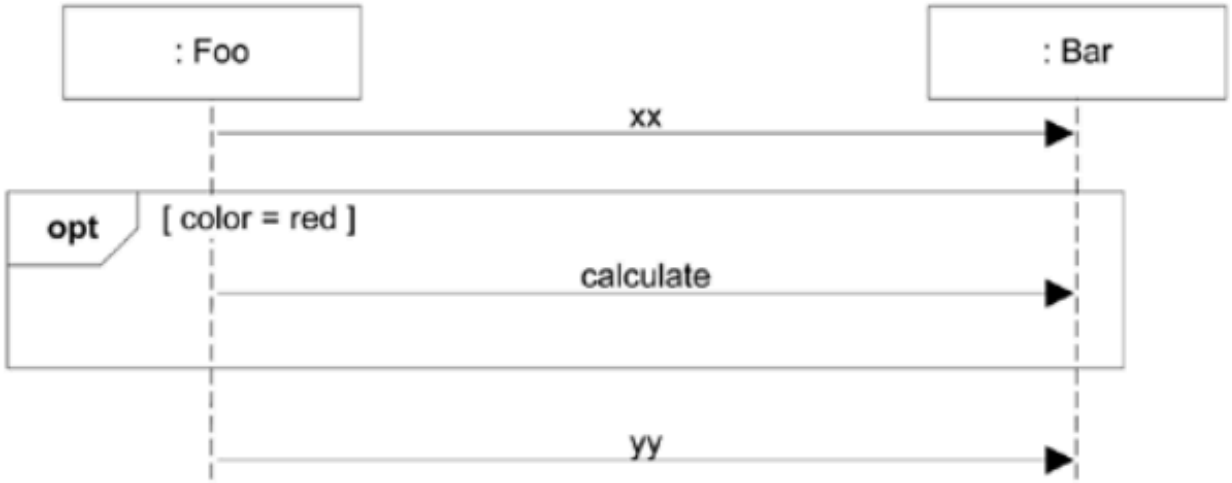


Figure 15.15. Mutually exclusive conditional messages.

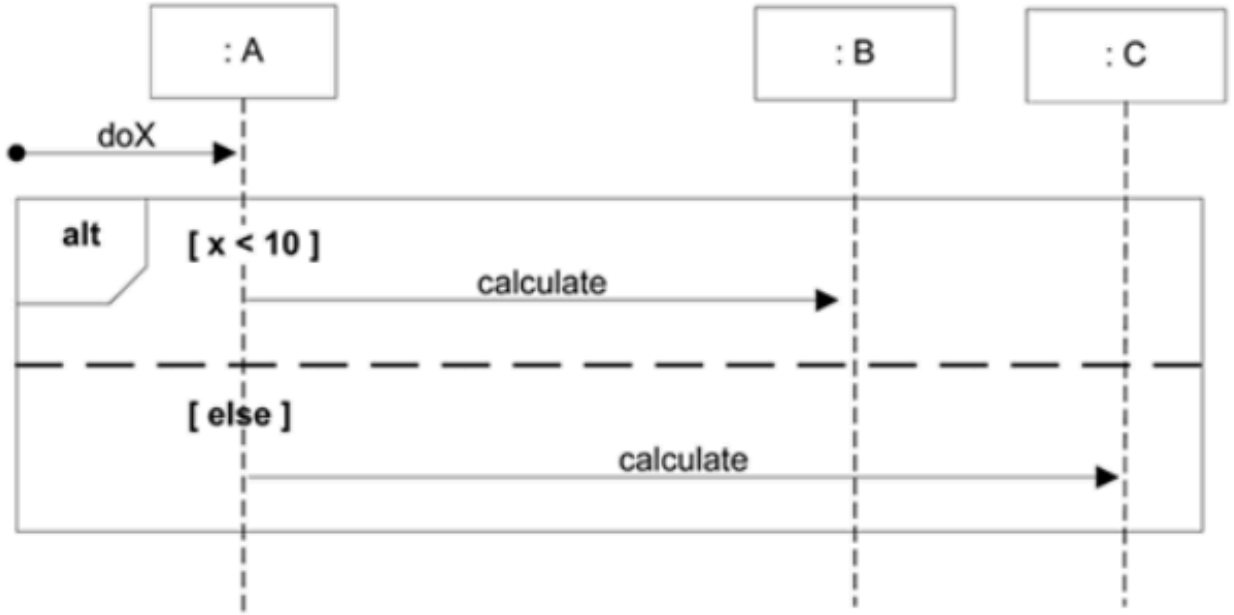


Figure 15.16. Iteration over a collection using relatively explicit notation.

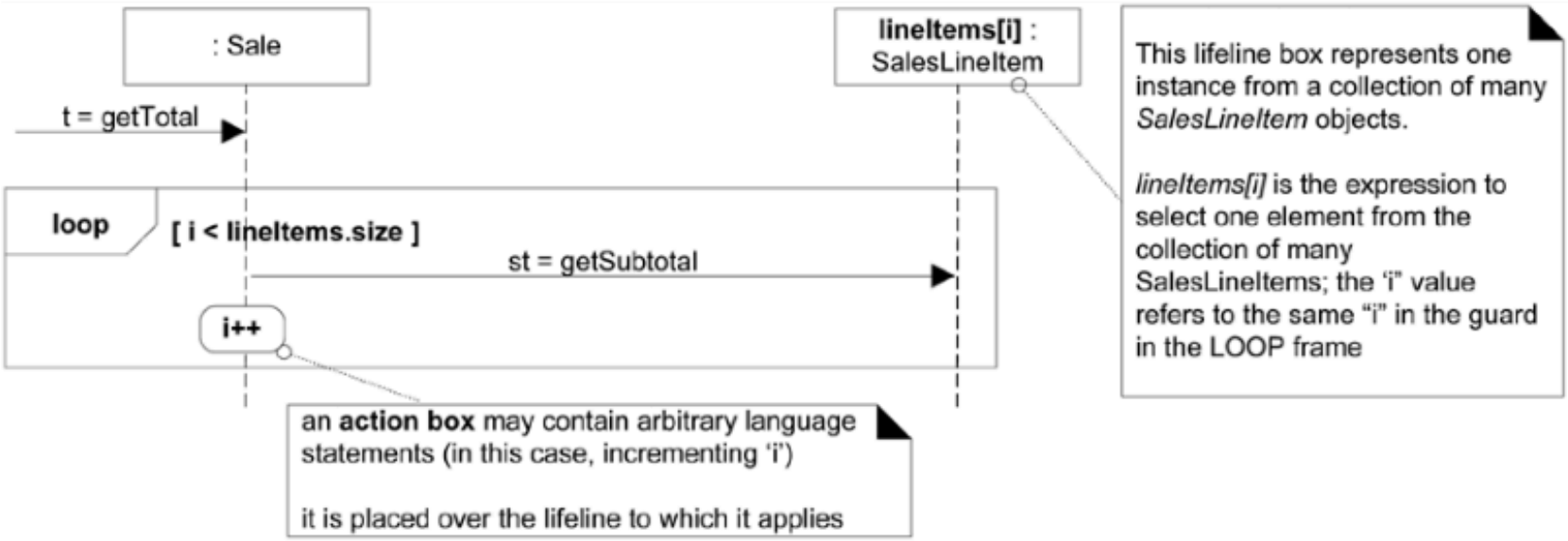


Figure 15.17. Iteration over a collection leaving things more implicit.

Figure 15.18. Nesting of frames.

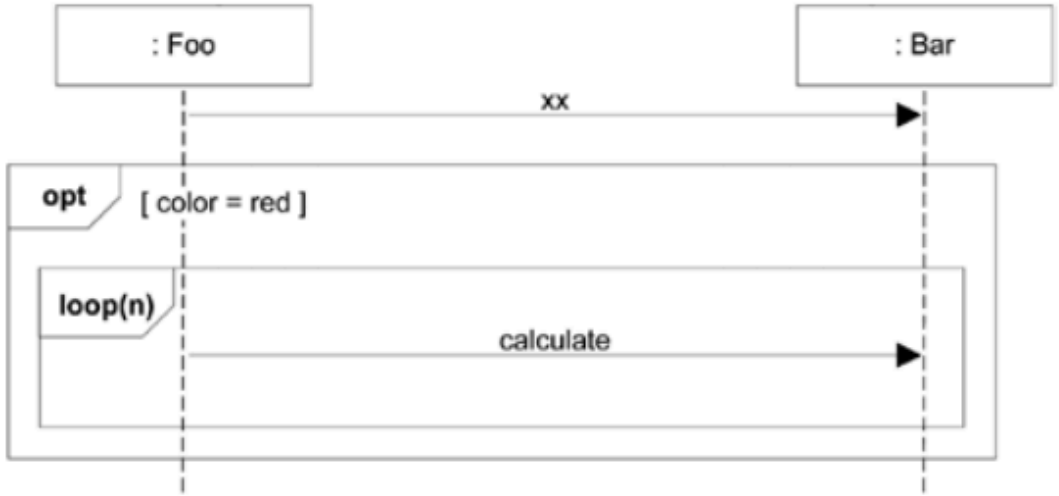


Figure 15.19. Example interaction occurrence, *sd* and *ref* frames.

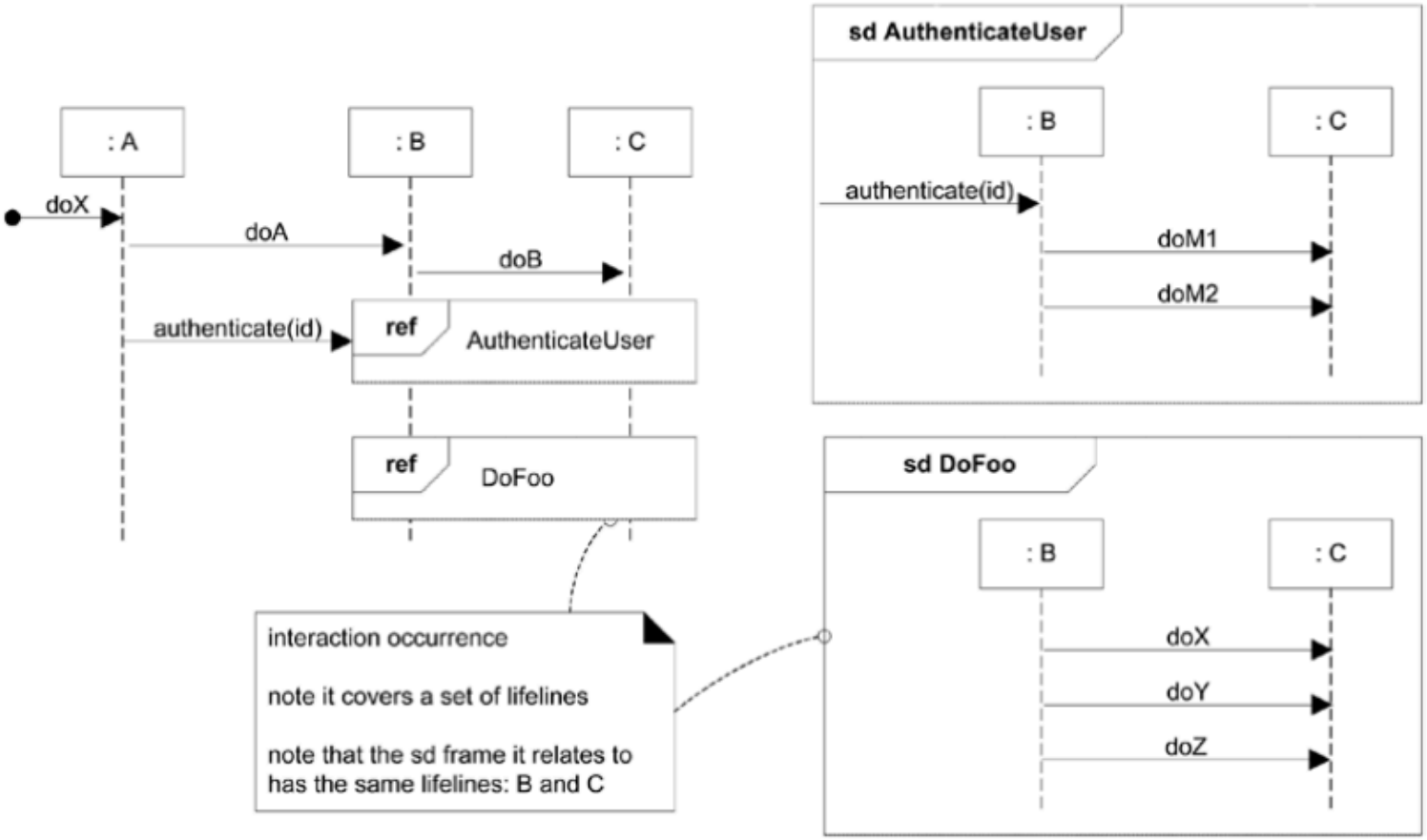


Figure 15.28. Complex sequence numbering.

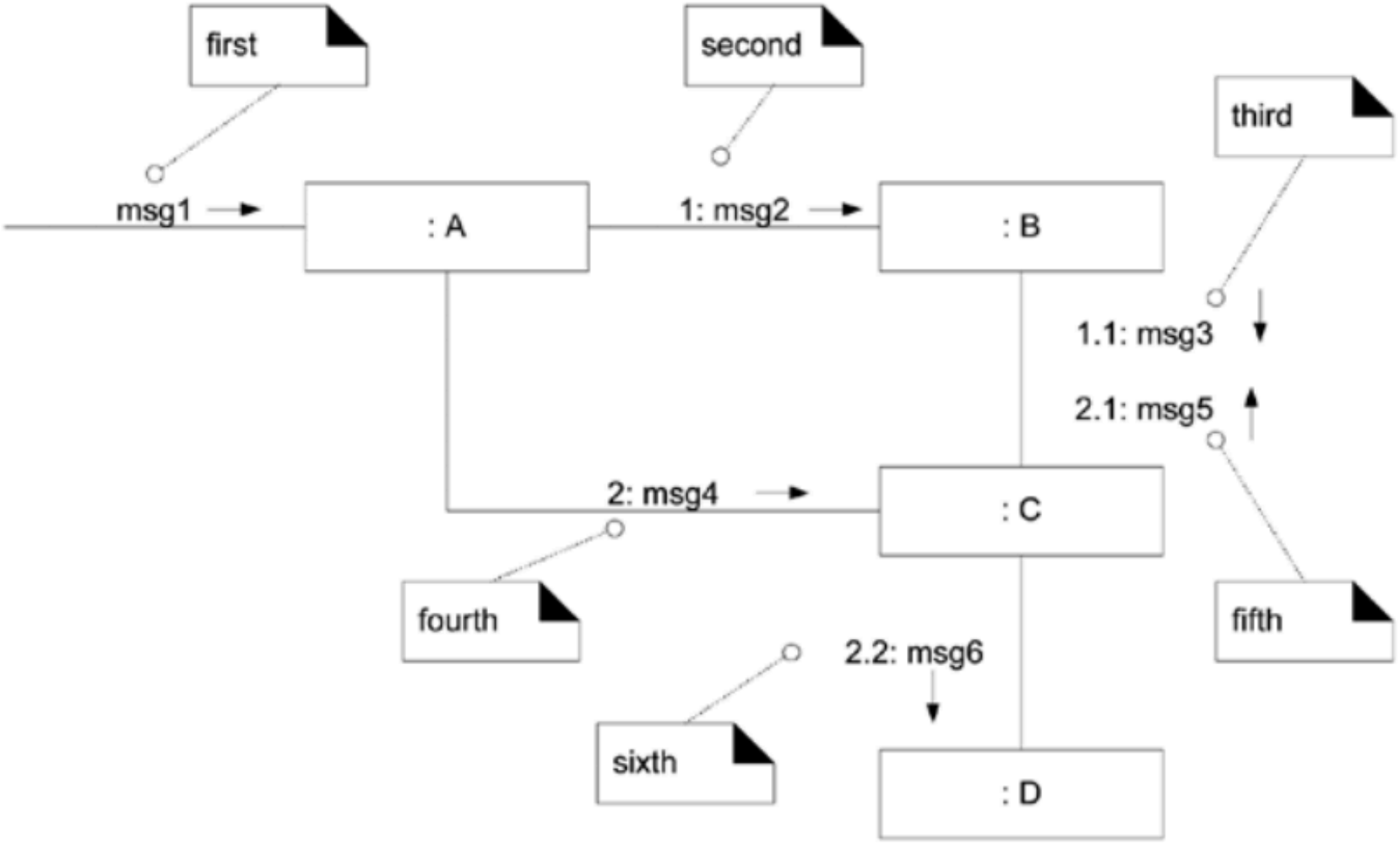


Figure 15.30. Mutually exclusive messages.

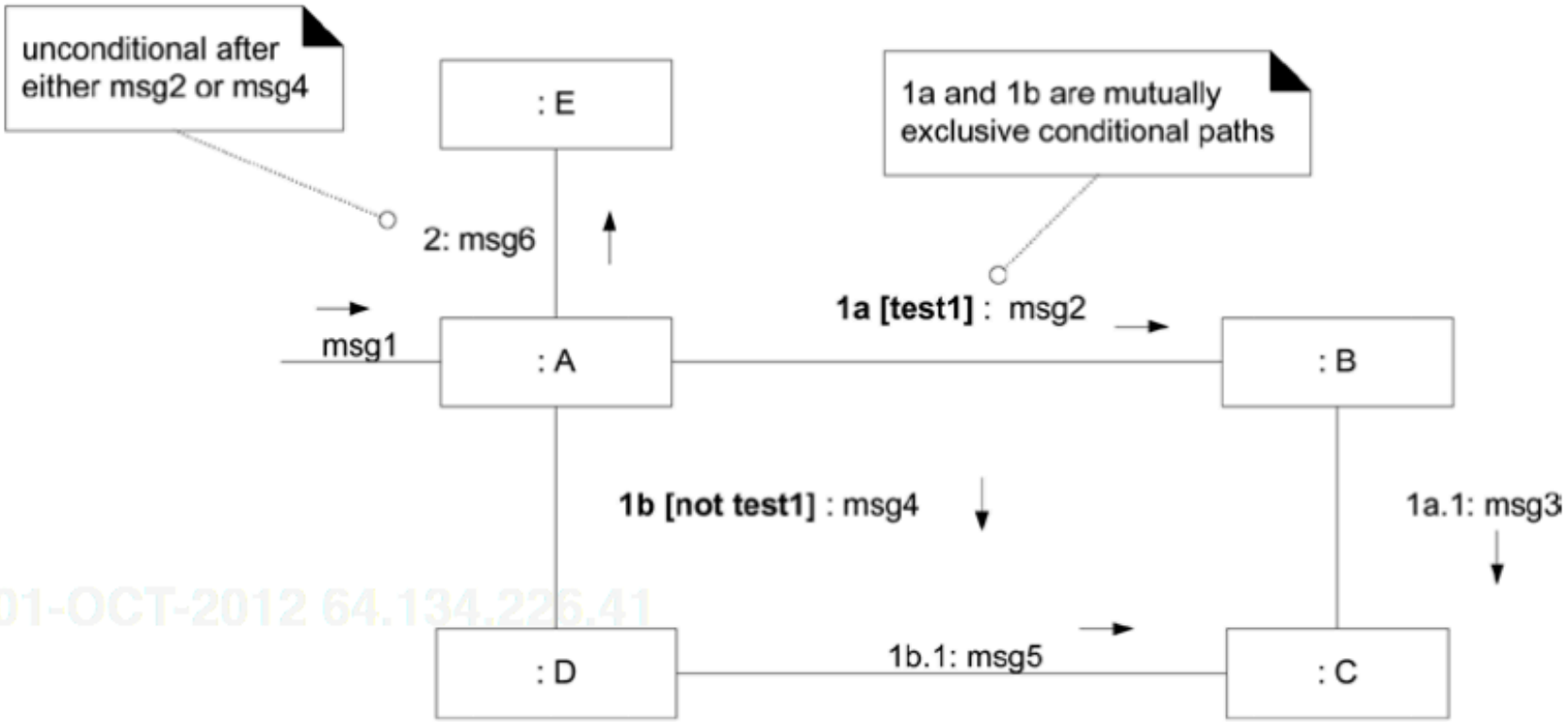


Figure 15.32. Iteration over a collection.

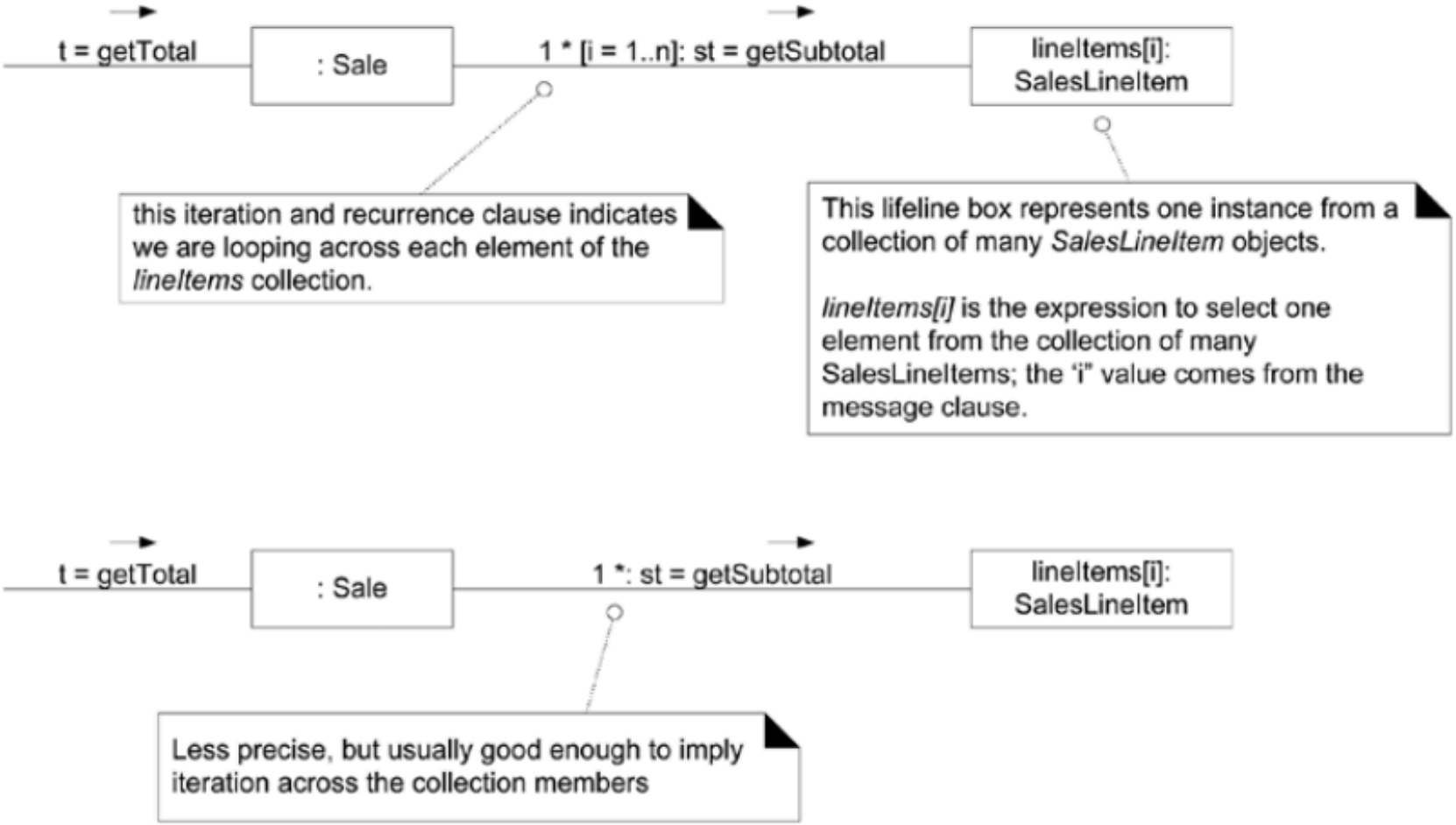
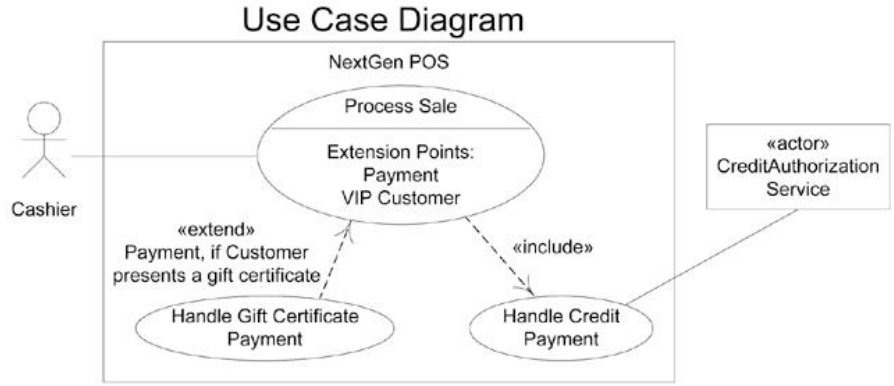
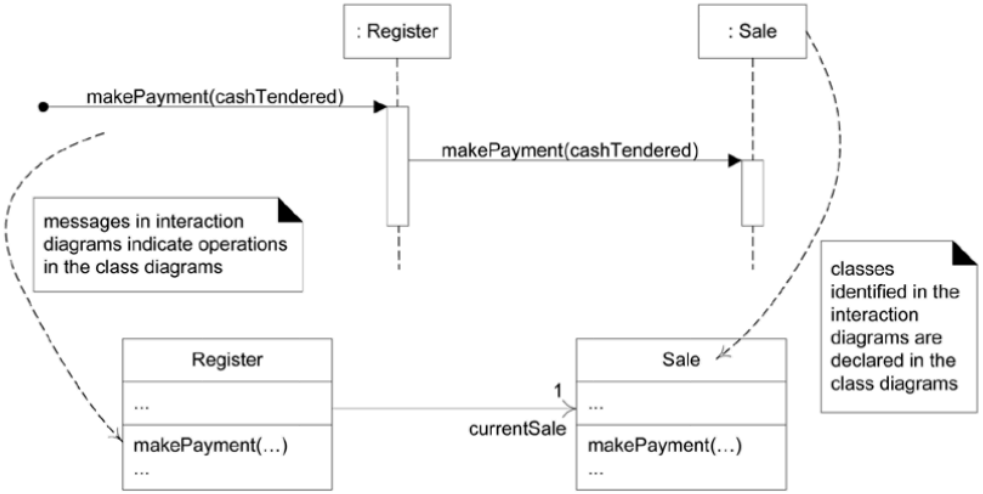
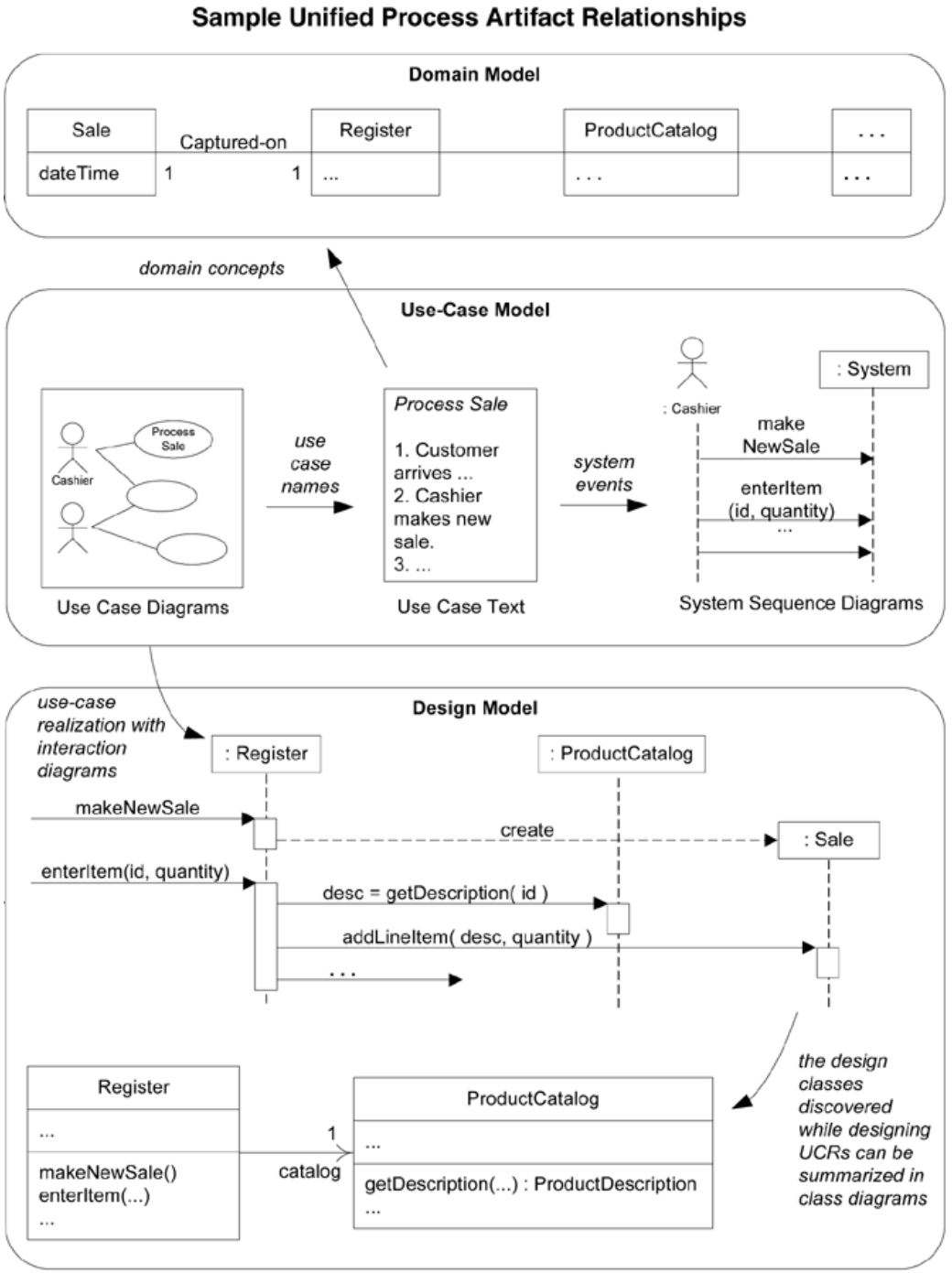


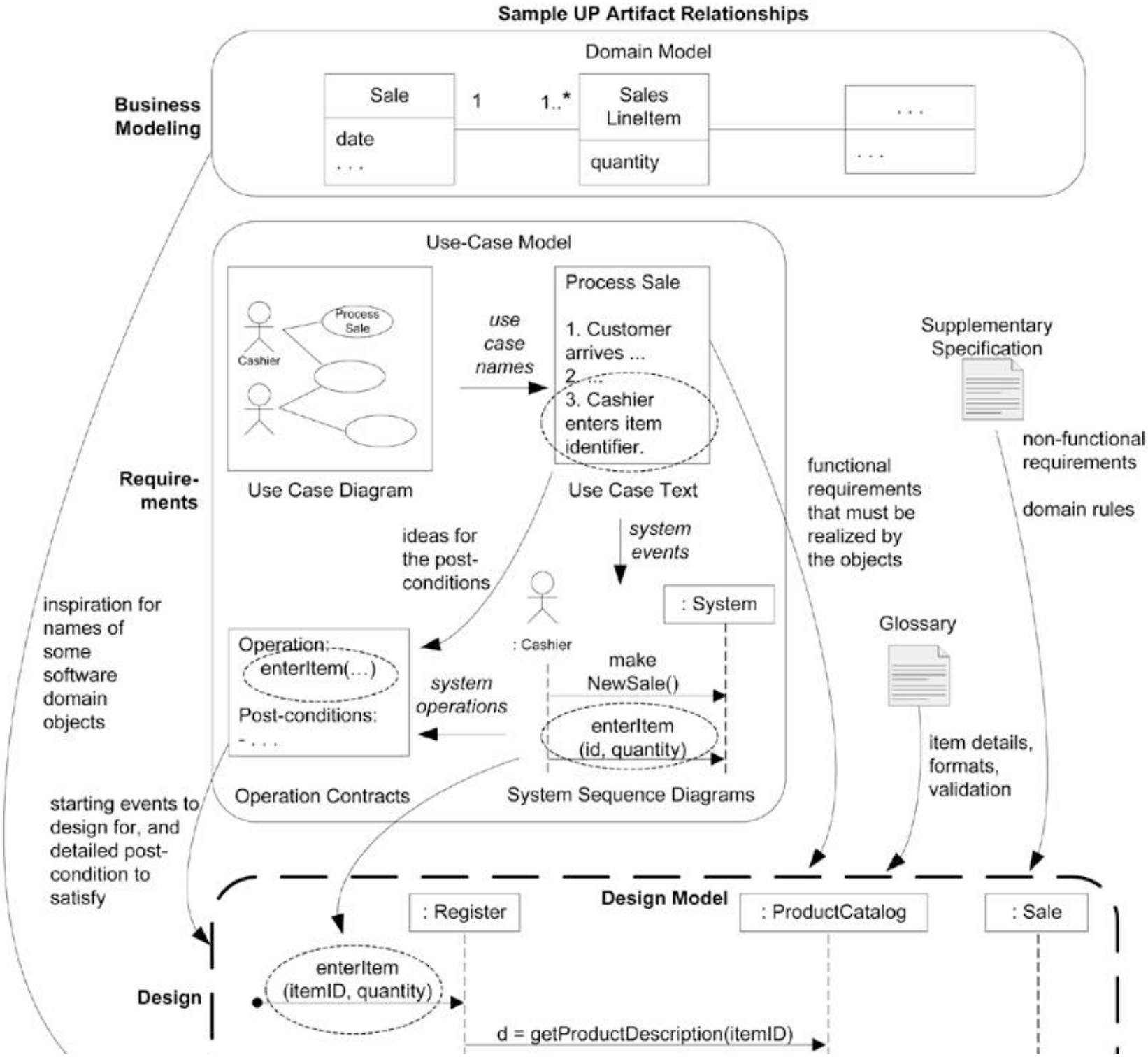
Diagram Relationships

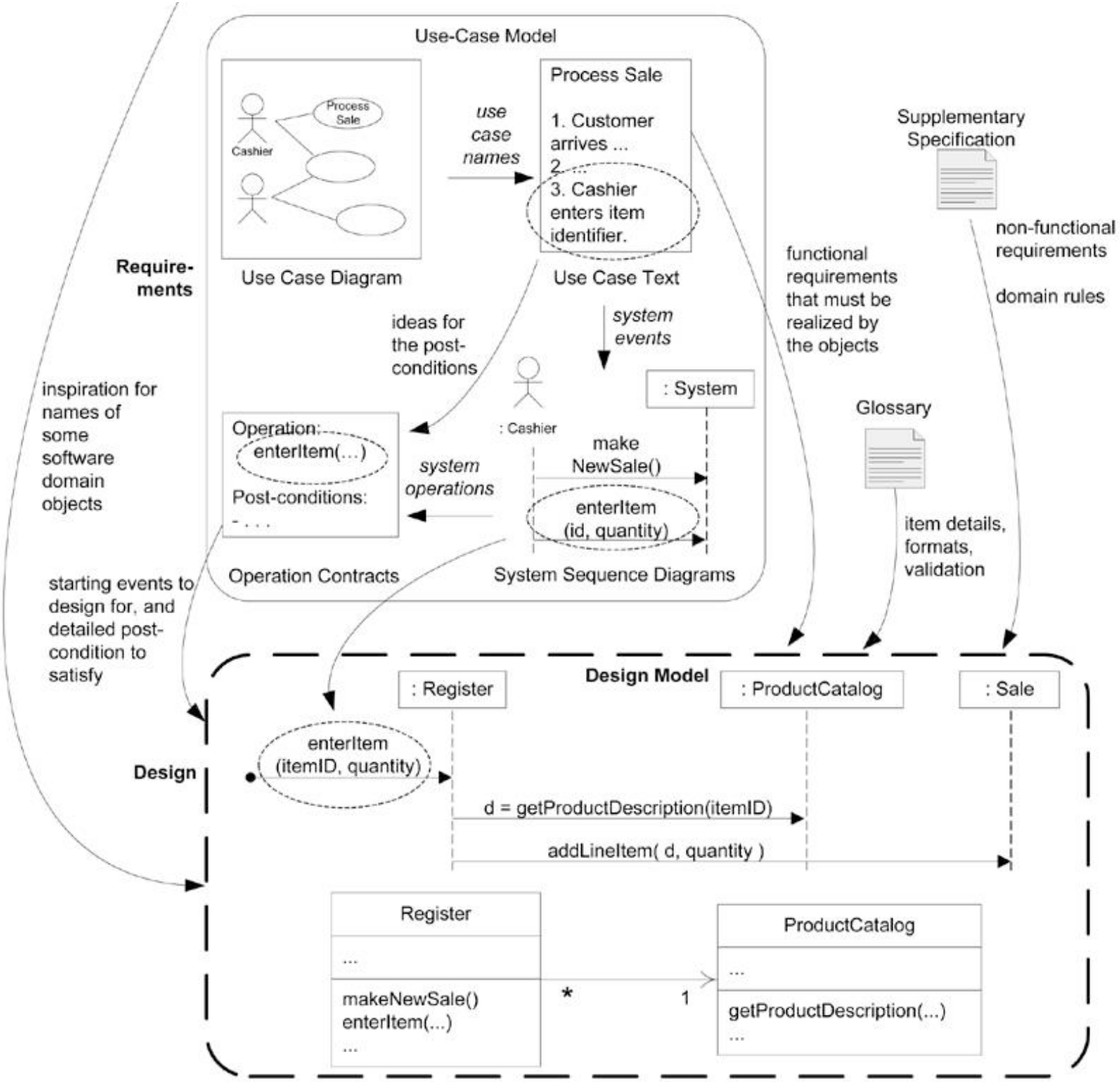
Quick Reference


Use Case Diagram
NextGen POS
Process Sale
Extension Points:
Payment
VIP Customer
Cashier
«extend»
Payment, if Customer
presents a gift certificate
«include»
Handle Gift Certificate
Payment
Handle Credit
Payment
«actor»
CreditAuthorization
Service


: Register
: Sale
makePayment(cashTendered)
makePayment(cashTendered)
messages in interaction diagrams indicate operations in the class diagrams
classes identified in the interaction diagrams are declared in the class diagrams
Register
...
makePayment(...)
...
1
currentSale
Sale
...
makePayment(...)
...


Sample Unified Process Artifact Relationships
Domain Model
Sale
dateTime
Captured-on
1
1
Register
...
ProductCatalog
...
domain concepts
Use-Case Model
Process Sale
Cashier
use case names
Process Sale
1. Customer arrives ...
2. Cashier makes new sale.
3. ...
system events
: Cashier
: System
make NewSale
enterItem (id, quantity)
...
Use Case Diagrams
Use Case Text
System Sequence Diagrams
use-case realization with interaction diagrams
Design Model
: Register
: ProductCatalog
makeNewSale
create
: Sale
enterItem(id, quantity)
desc = getDescription(id)
addLineItem(desc, quantity)
...
Register
...
makeNewSale()
enterItem(...)
...
1
catalog
ProductCatalog
...
getDescription(...) : ProductDescription
...
the design classes discovered while designing UCRs can be summarized in class diagrams


Sample UP Artifact Relationships
Business Modeling
Domain Model
Sale
date
. . .
1
1..*
Sales LineItem
quantity
. . .
. . .
Requirements
Use-Case Model
Process Sale
Cashier
use case names
Process Sale
1. Customer arrives ...
2. ...
3. Cashier enters item identifier.
Use Case Diagram
Use Case Text
ideas for the post-conditions
system events
: System
: Cashier
Operation:
enterItem(...)
Post-conditions:
- . . .
system operations
make NewSale()
enterItem (id, quantity)
Operation Contracts
System Sequence Diagrams
inspiration for names of some software domain objects
starting events to design for, and detailed post-condition to satisfy
Supplementary Specification
non-functional requirements
domain rules
functional requirements that must be realized by the objects
Glossary
item details, formats, validation
Design Model
Design
: Register
: ProductCatalog
: Sale
enterItem (itemID, quantity)
d = getProductDescription(itemID)


Use-Case Model
Process Sale
Cashier
Process Sale
use case names
1. Customer arrives ...
2. ...
3. Cashier enters item identifier.
Use Case Diagram
Use Case Text
Require-ments
Supplementary Specification
non-functional requirements
domain rules
functional requirements that must be realized by the objects
ideas for the post-conditions
system events
: System
: Cashier
Operation:
enterItem(...)
Post-conditions:
- . . .
system operations
make NewSale()
enterItem (id, quantity)
Glossary
item details, formats, validation
inspiration for names of some software domain objects
Operation Contracts
System Sequence Diagrams
starting events to design for, and detailed post-condition to satisfy
Design Model
Design
: Register
: ProductCatalog
: Sale
enterItem (itemID, quantity)
d = getProductDescription(itemID)
addLineItem(d, quantity)
Register
...
makeNewSale()
enterItem(...)
...
*
1
ProductCatalog
...
getProductDescription(...)
...