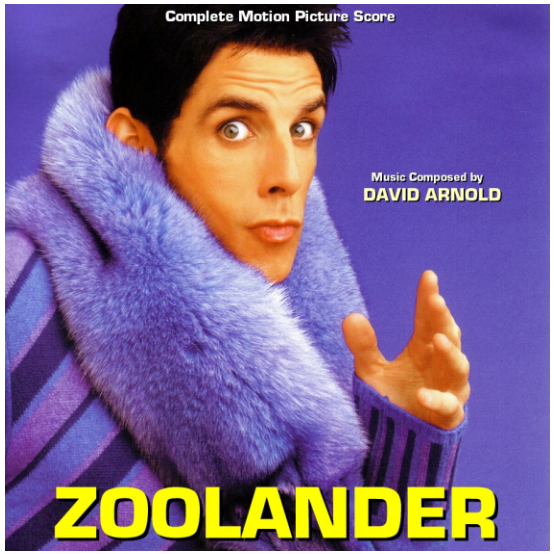
10 Things You Should Know About UML

Paul Nguyen

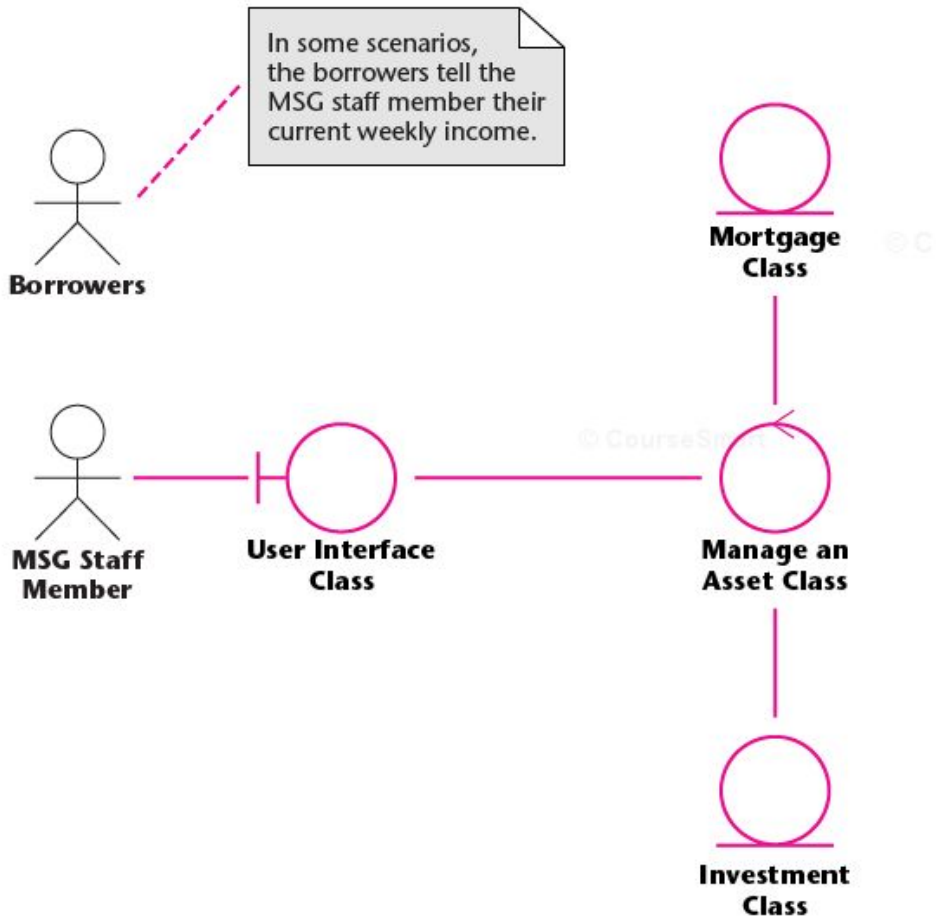
Do you “Model”?

by 2006 Gartner estimated that more than 10 million IT professionals used UML, and by 2008 over 70% of software development organisations worldwide were using it.



Example - Manage Asset Use Case

A class diagram showing the classes that realize the `Manage an Asset` use case of the MSG Foundation case study.

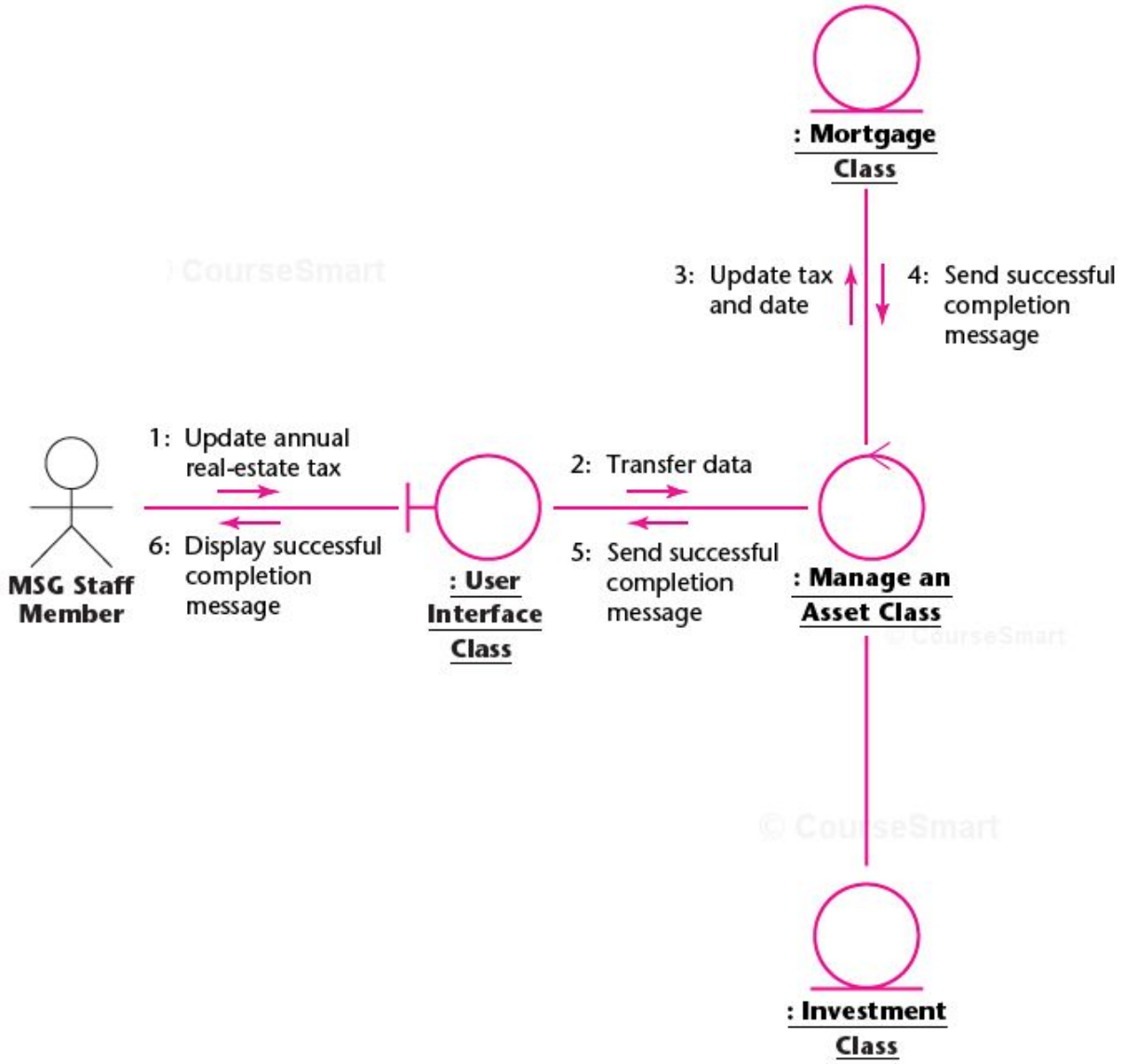


A scenario of the `Manage an Asset` use case.

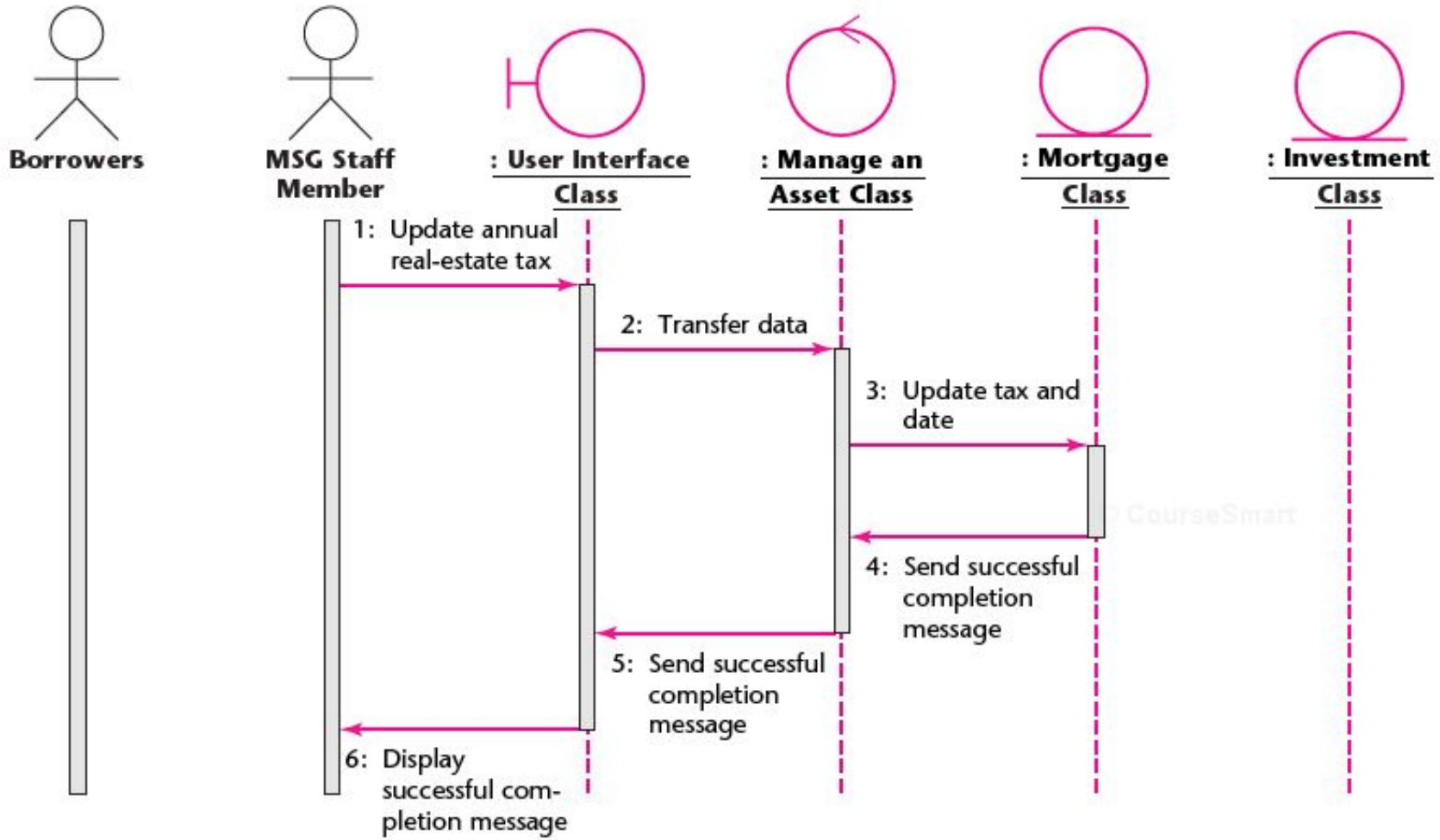
An MSG Foundation staff member wants to update the annual real-estate tax on a home for which the Foundation has provided a mortgage.

1. The staff member enters the new value of the annual real-estate tax.
2. The information system updates the date on which the annual real-estate tax was last changed.

Example - Manage Asset Use Case

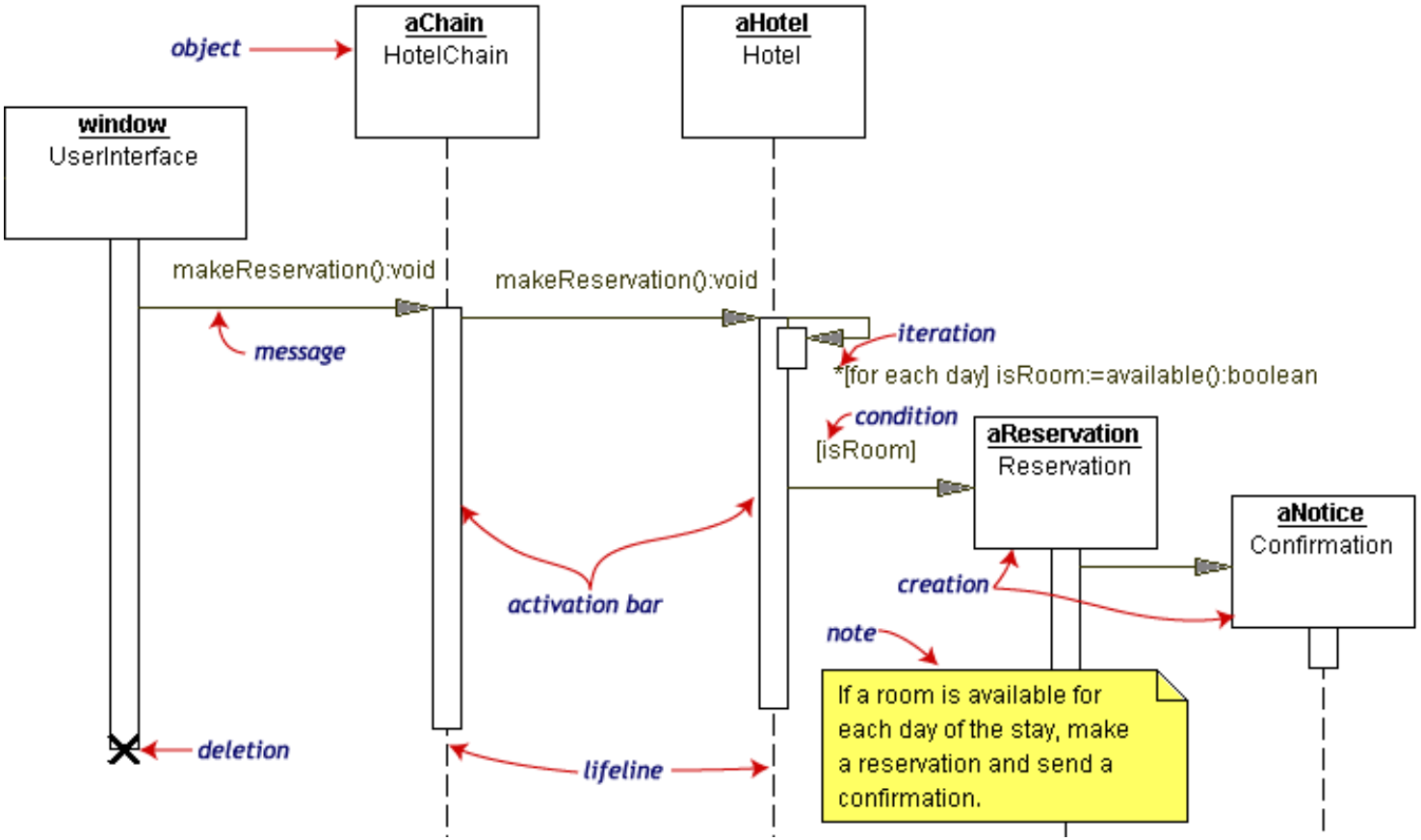


Example - Manage Asset Use Case



Thing #6

- The Sequence Diagram
  - Good at showing “sequential” flow… Not good at concurrency



http://dn.codegear.com/article/31863

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.

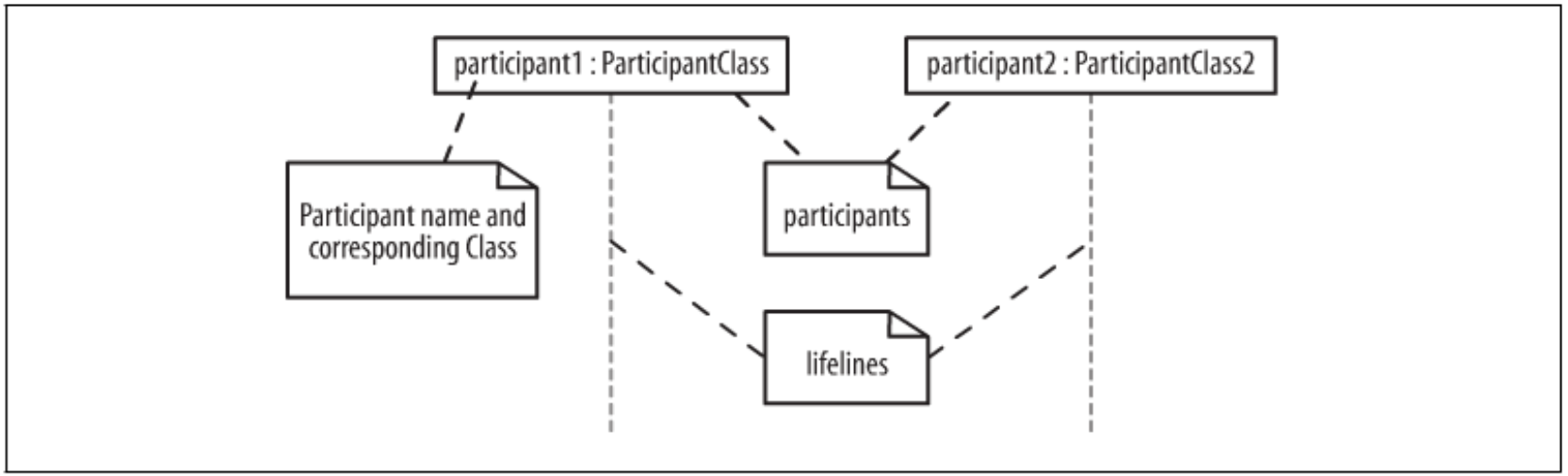


Figure 7-2. At its simplest, a sequence diagram is made up of one or more participants—only one participant would be a very strange sequence diagram, but it would be perfectly legal UML

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

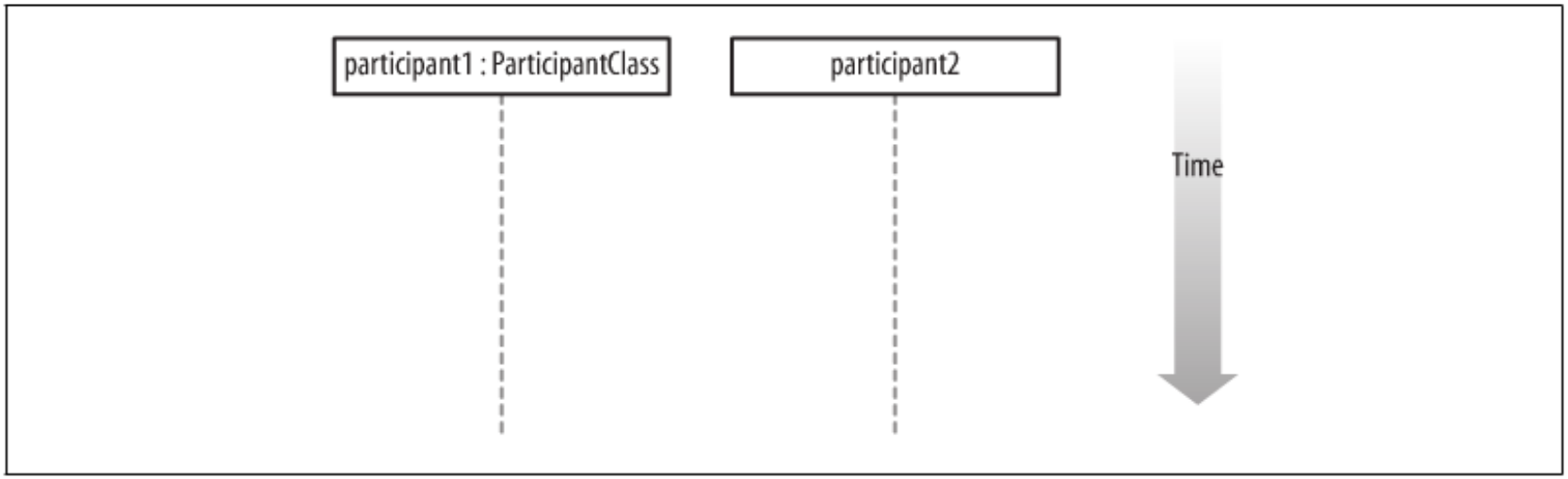


Figure 7-3. Time runs down the page on a sequence diagram in keeping with the participant lifeline

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.

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.

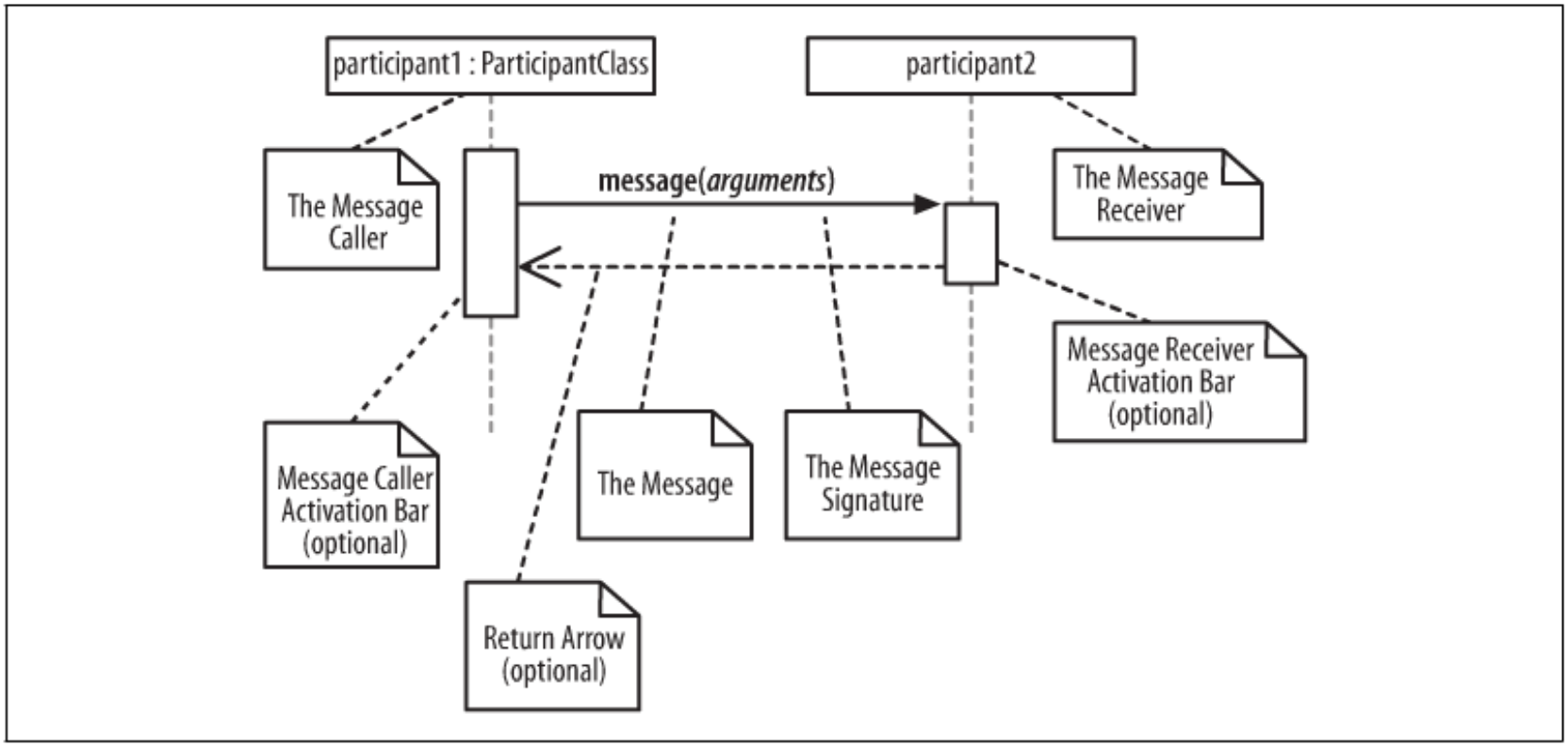


Figure 7-5. Interactions on a sequence diagram are shown as messages between participants

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.

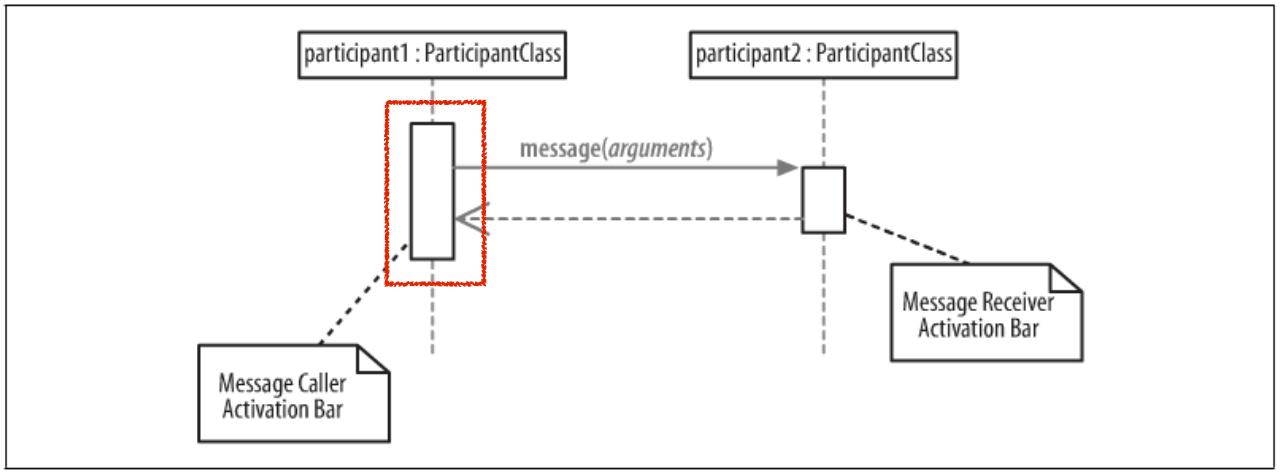


Figure 7-6. Activation bars show that a participant is busy doing something for a period of time

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.

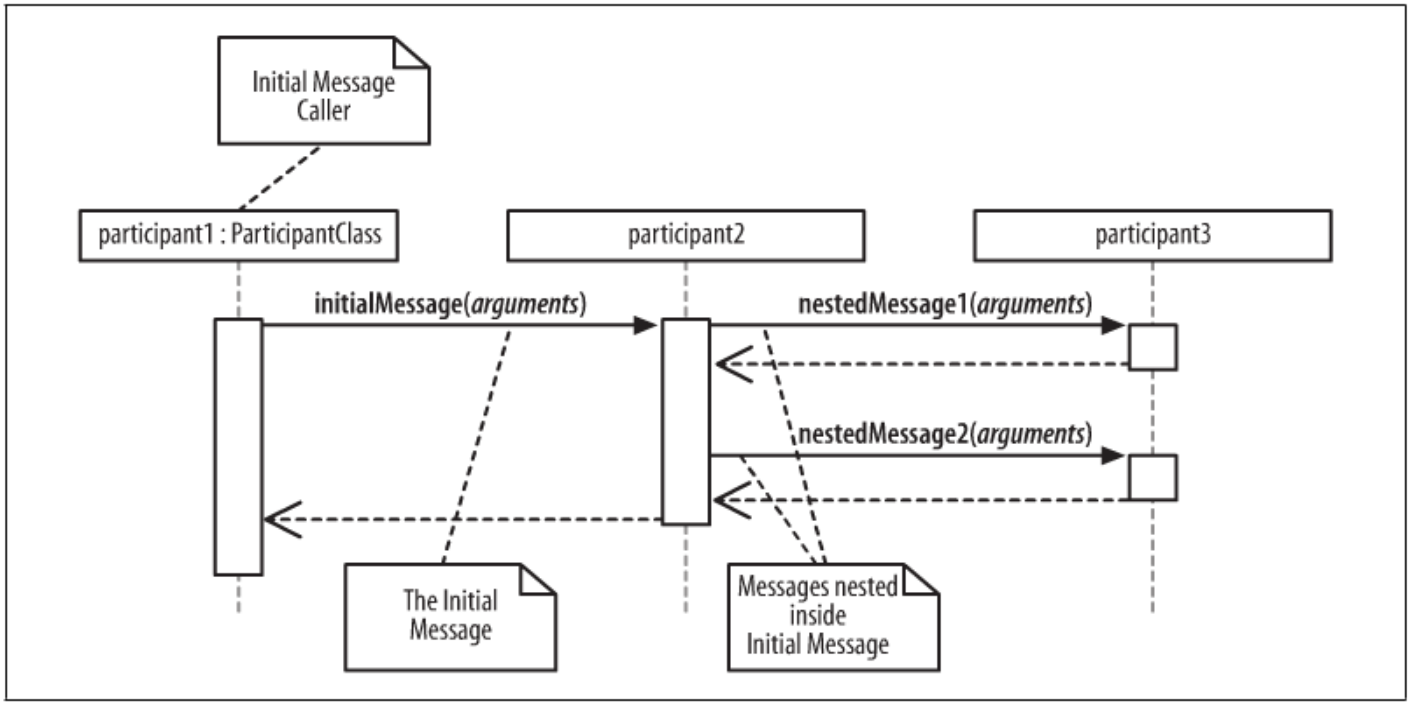


Figure 7-7. Two nested messages are invoked when an initial message is received

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.

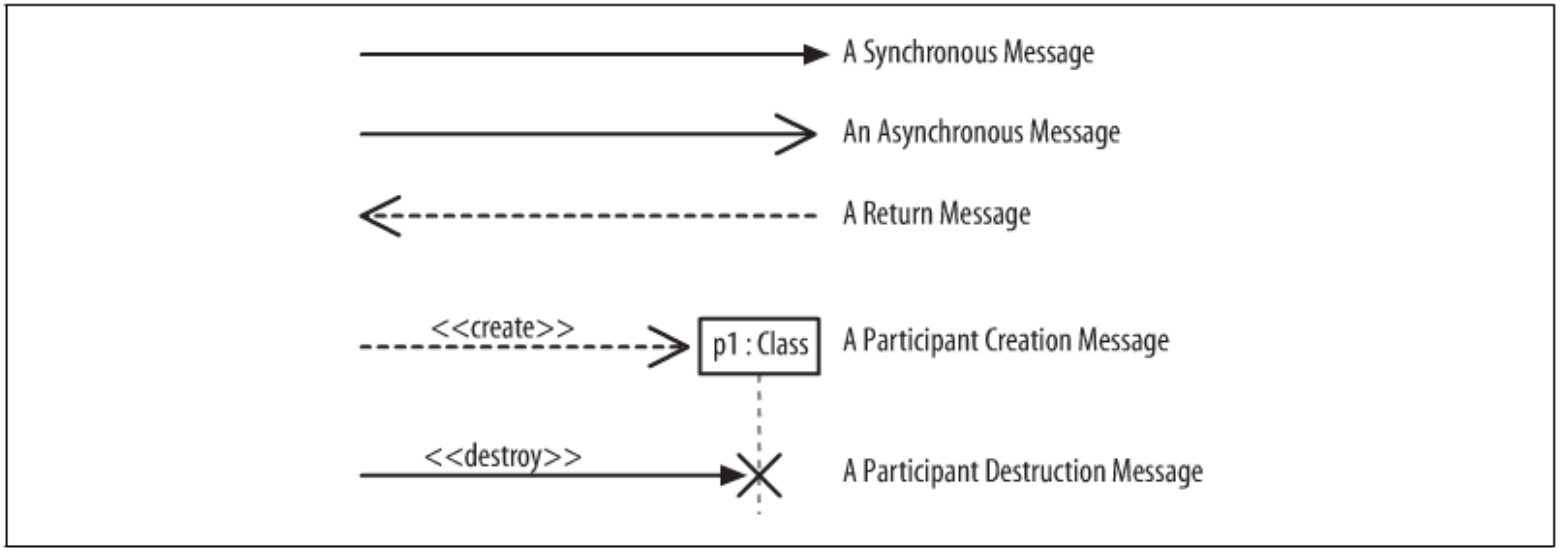


Figure 7-8. There are five main types of message arrow for use on sequence diagram, and each has its own meaning

Realizing Use Cases

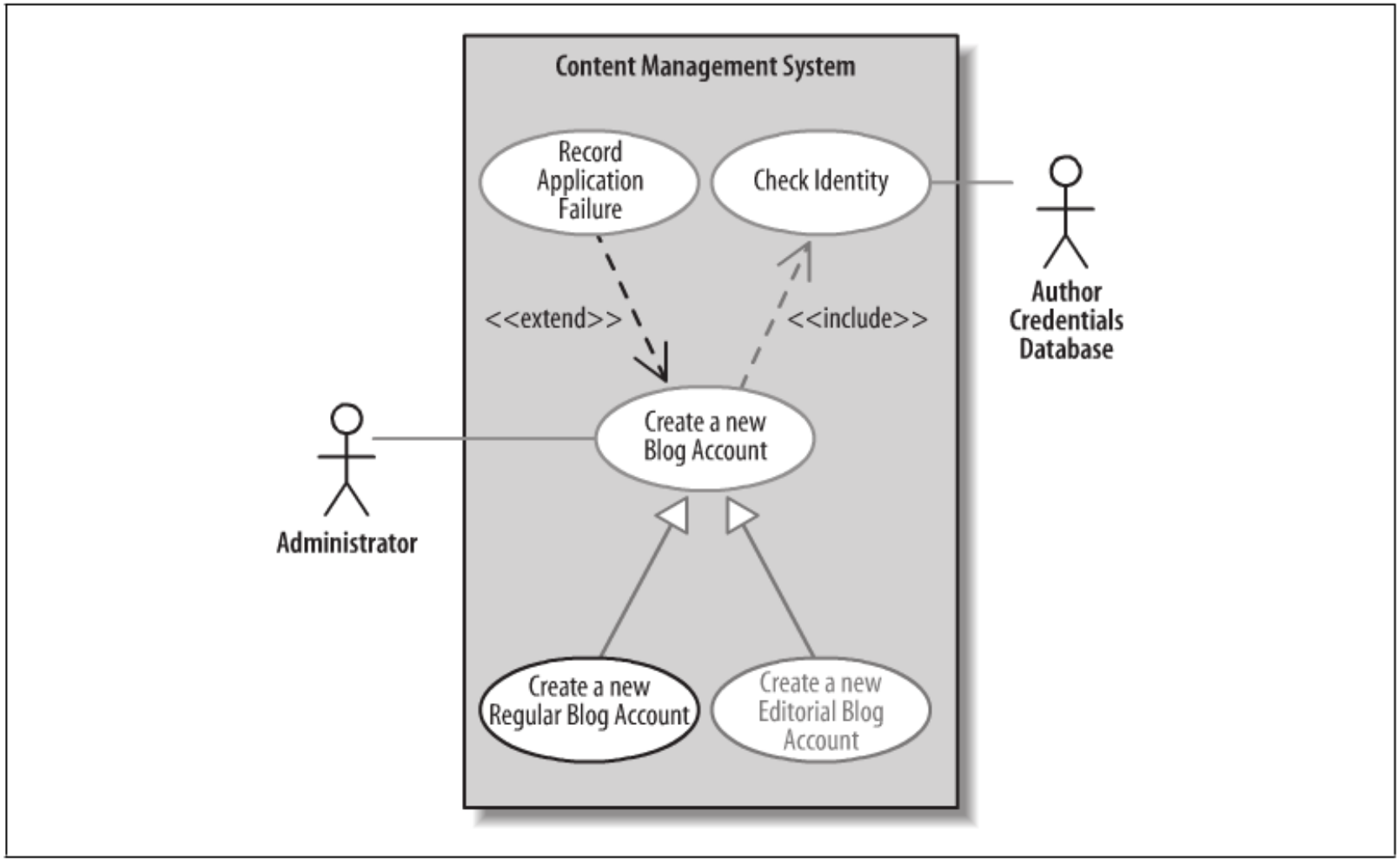


Figure 7-13. The Create a new Regular Blog Account use case diagram

Top-Level Sequence Diagram

A Top-Level Sequence Diagram

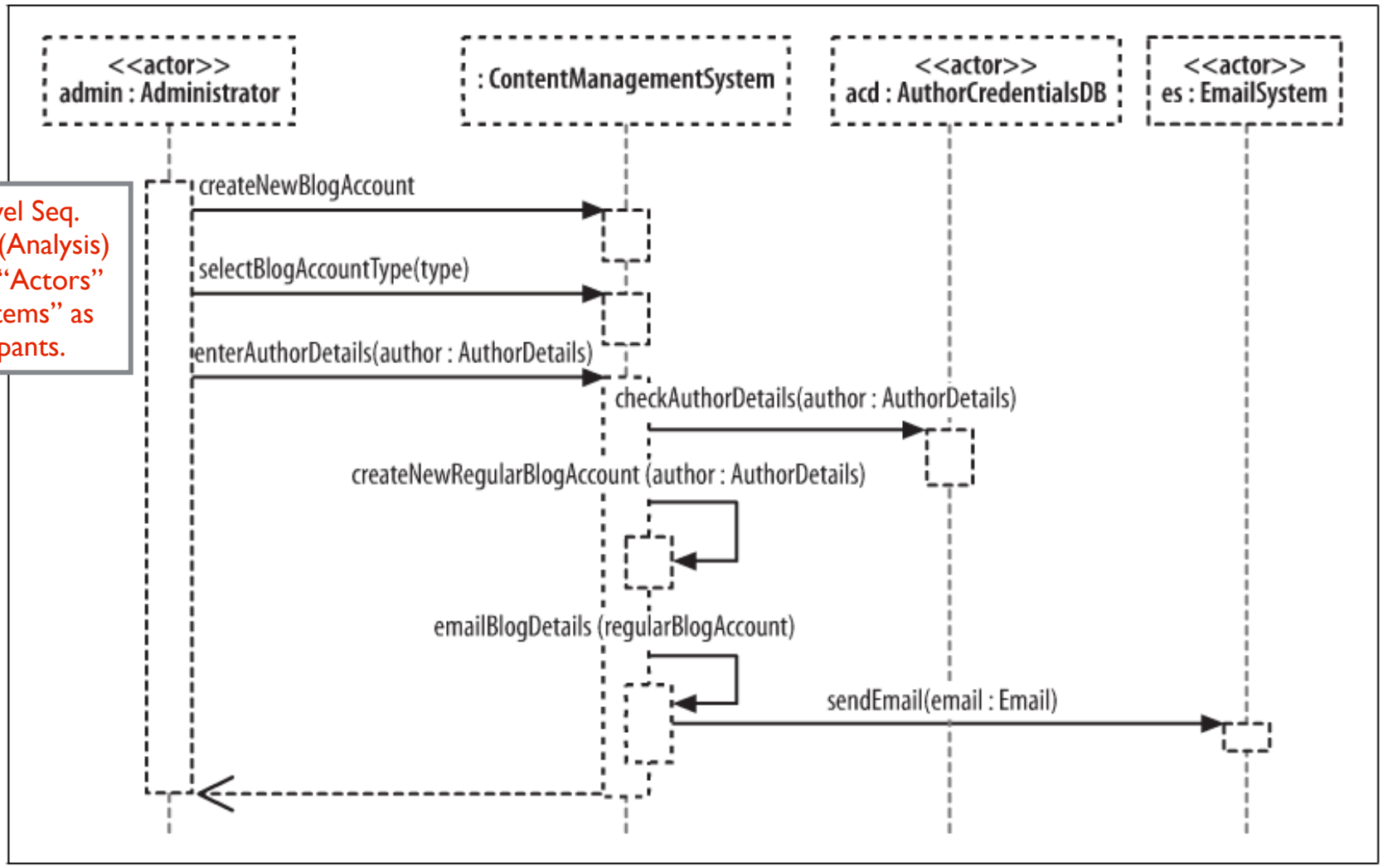
Before you can specify what types of interaction are going to occur when a use case executes, you need a more detailed description of what the use case does. If you've already completed a use case description, you already have a good reference for this detailed information.

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.



Top Level Seq. Diagrams (Analysis) only have "Actors" and "Systems" as Participants.

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

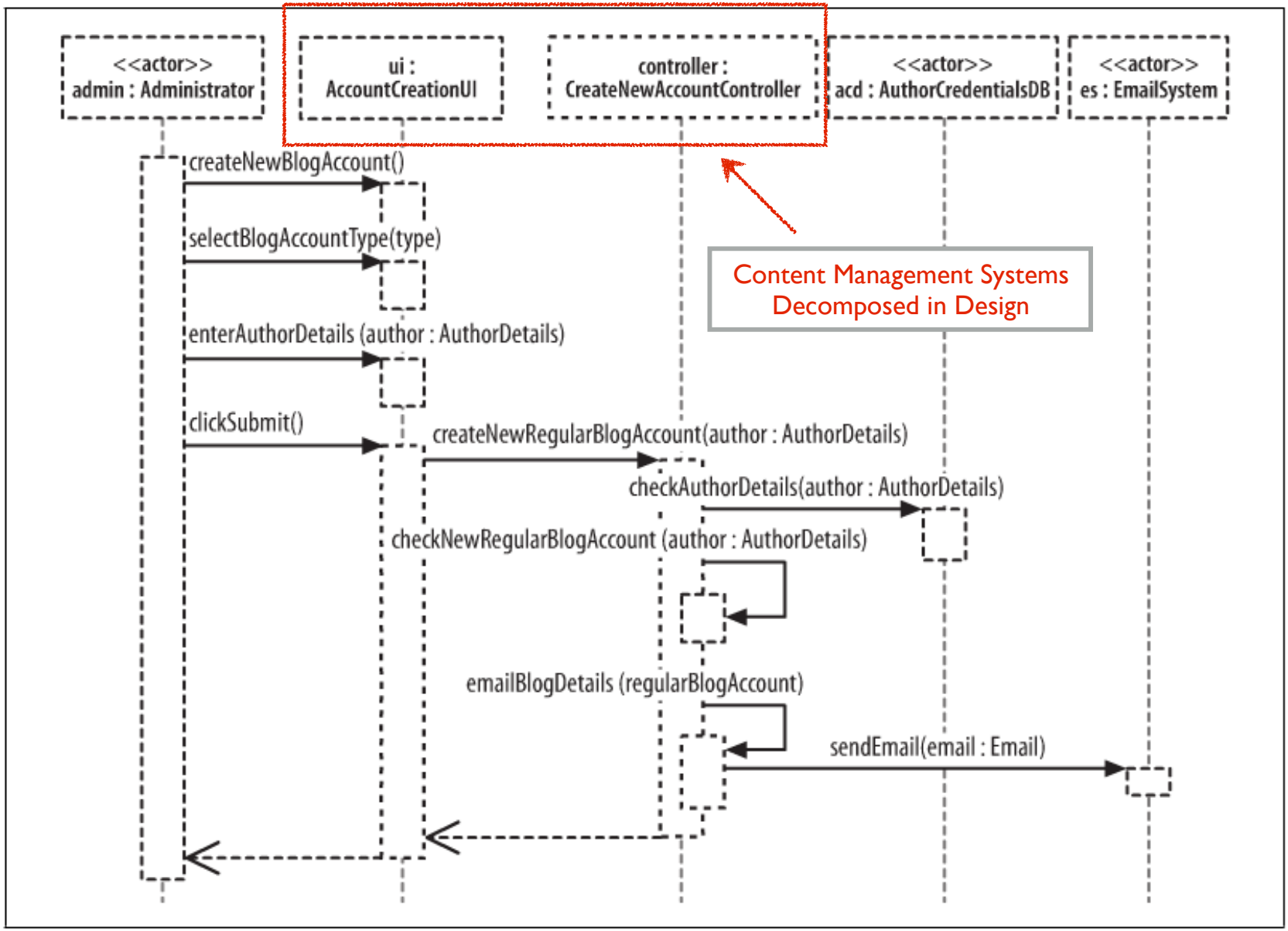
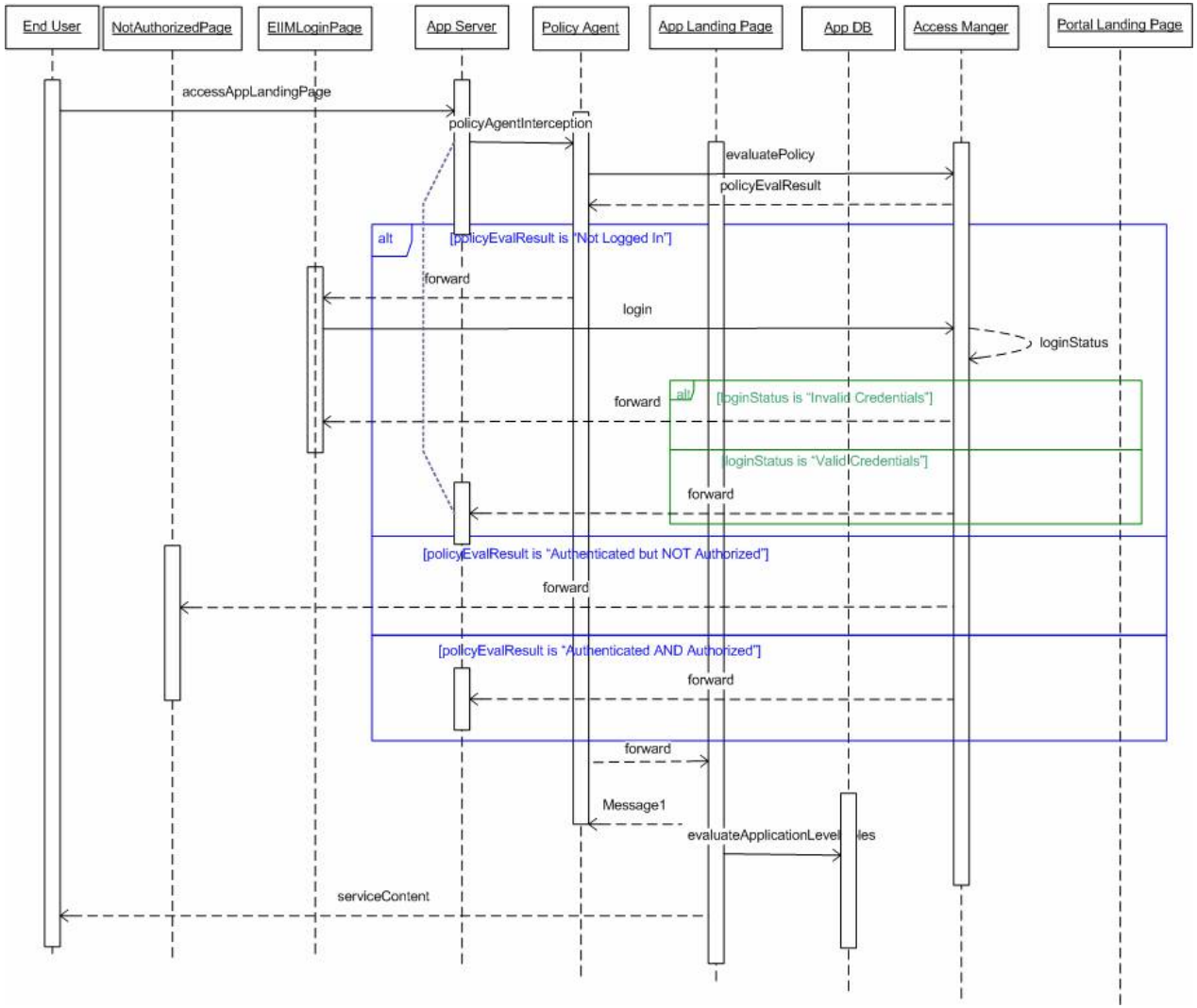


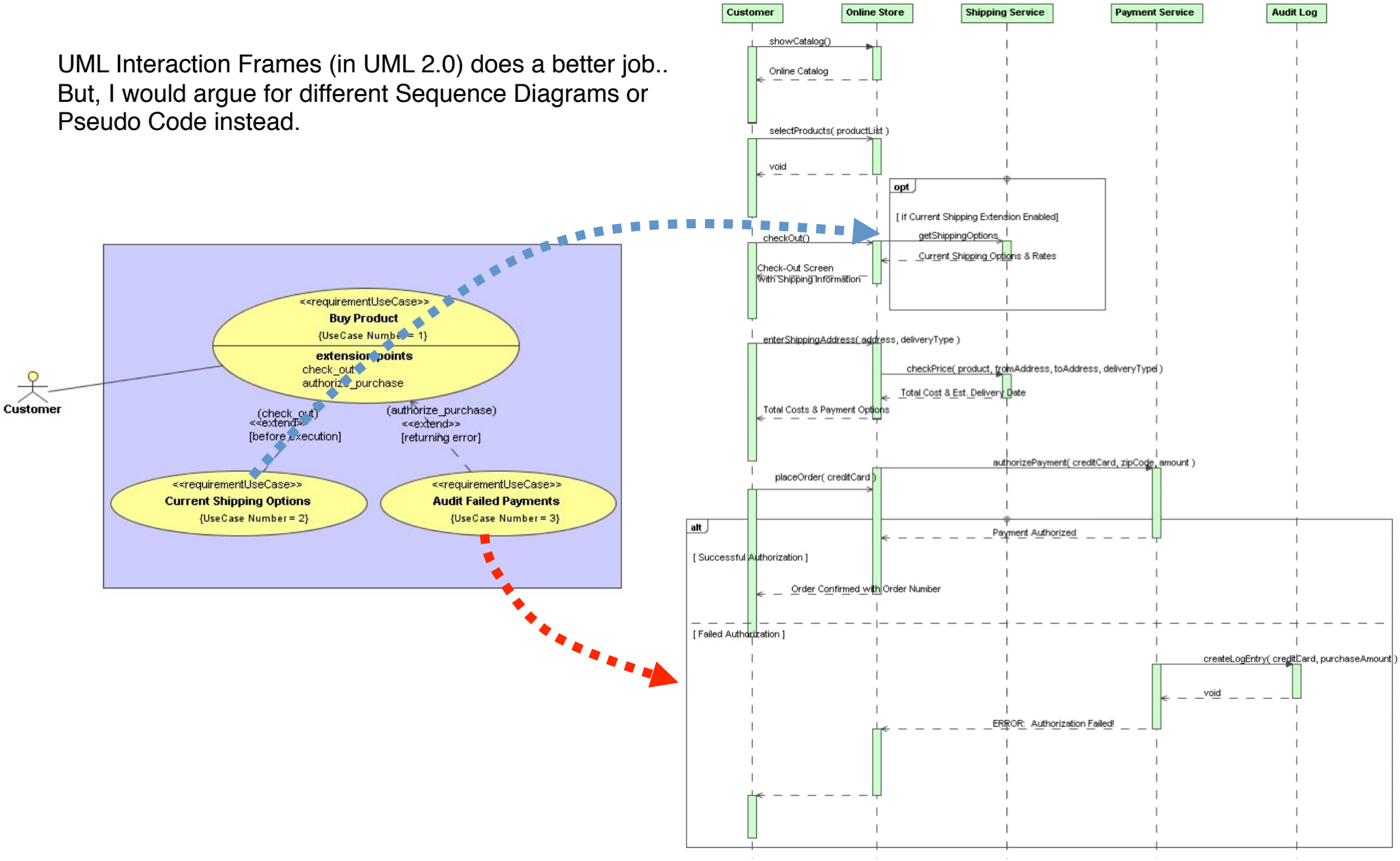
Figure 7-15. Adding more detail about the internals of your system

When Sequence Diagrams go “wrong”!

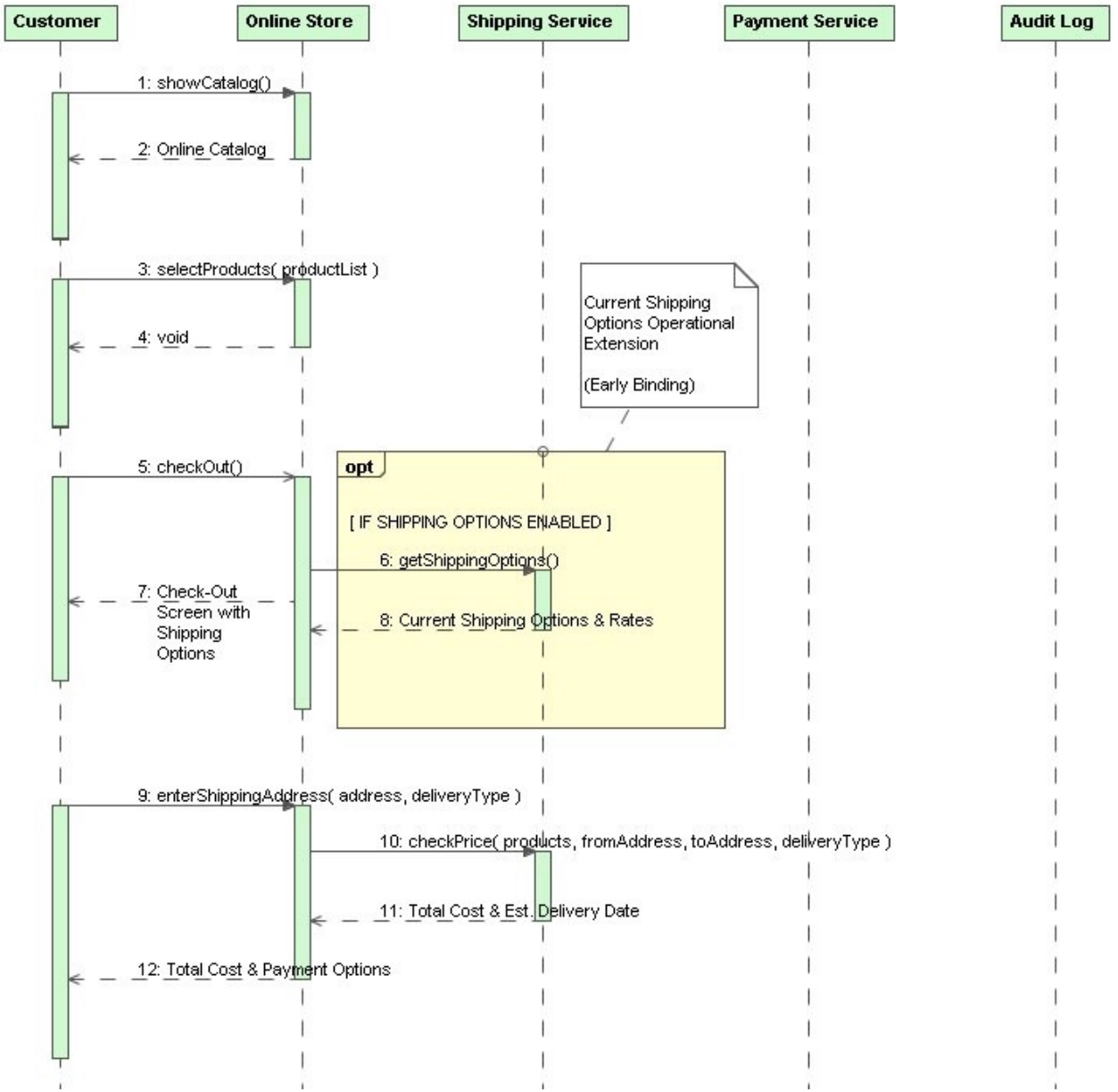


UML 2.0 Interaction Frame (with sequence fragments)

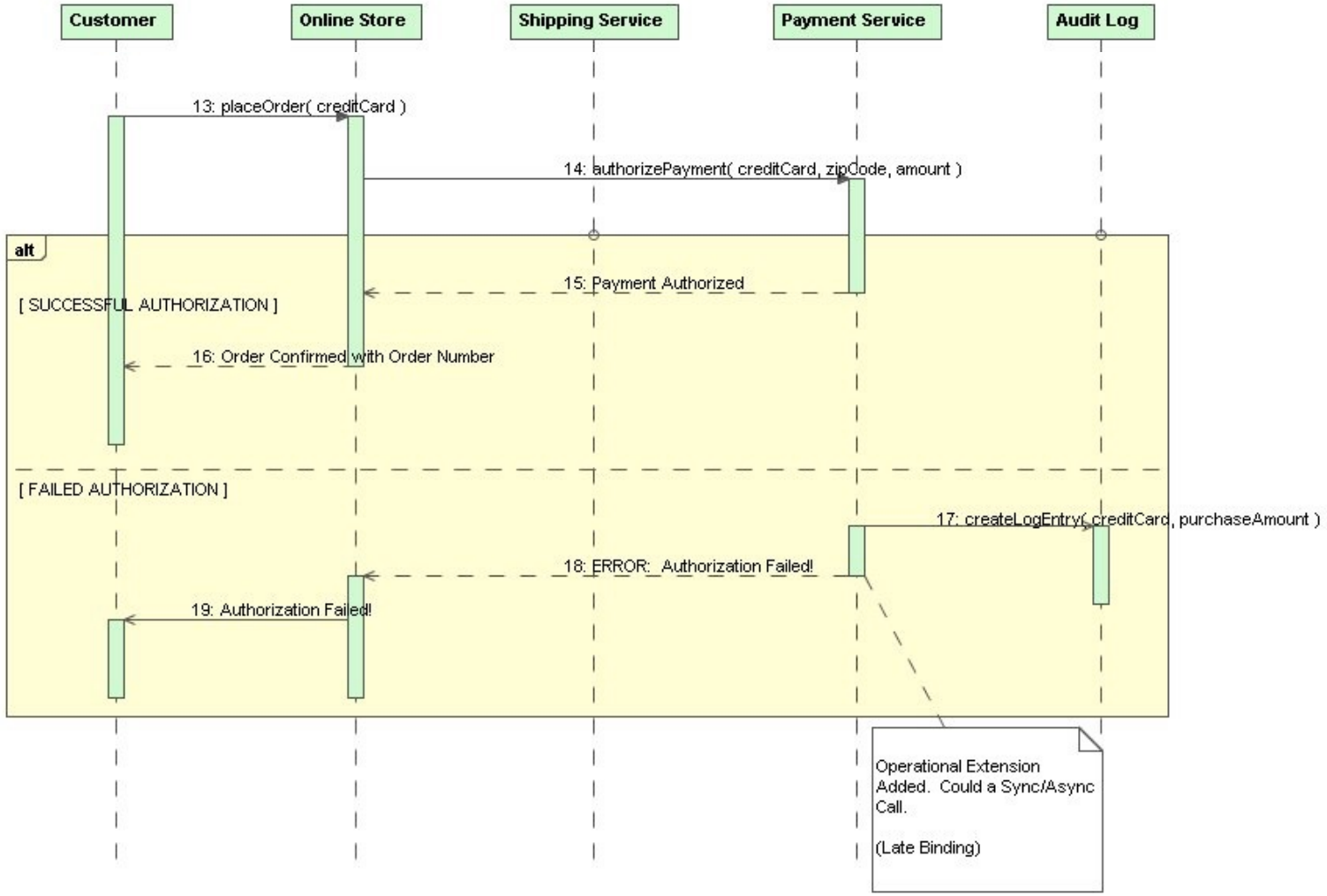
UML Interaction Frames (in UML 2.0) does a better job..
But, I would argue for different Sequence Diagrams or Pseudo Code instead.



UML 2.0 Interaction Frame ("opt" fragment example)

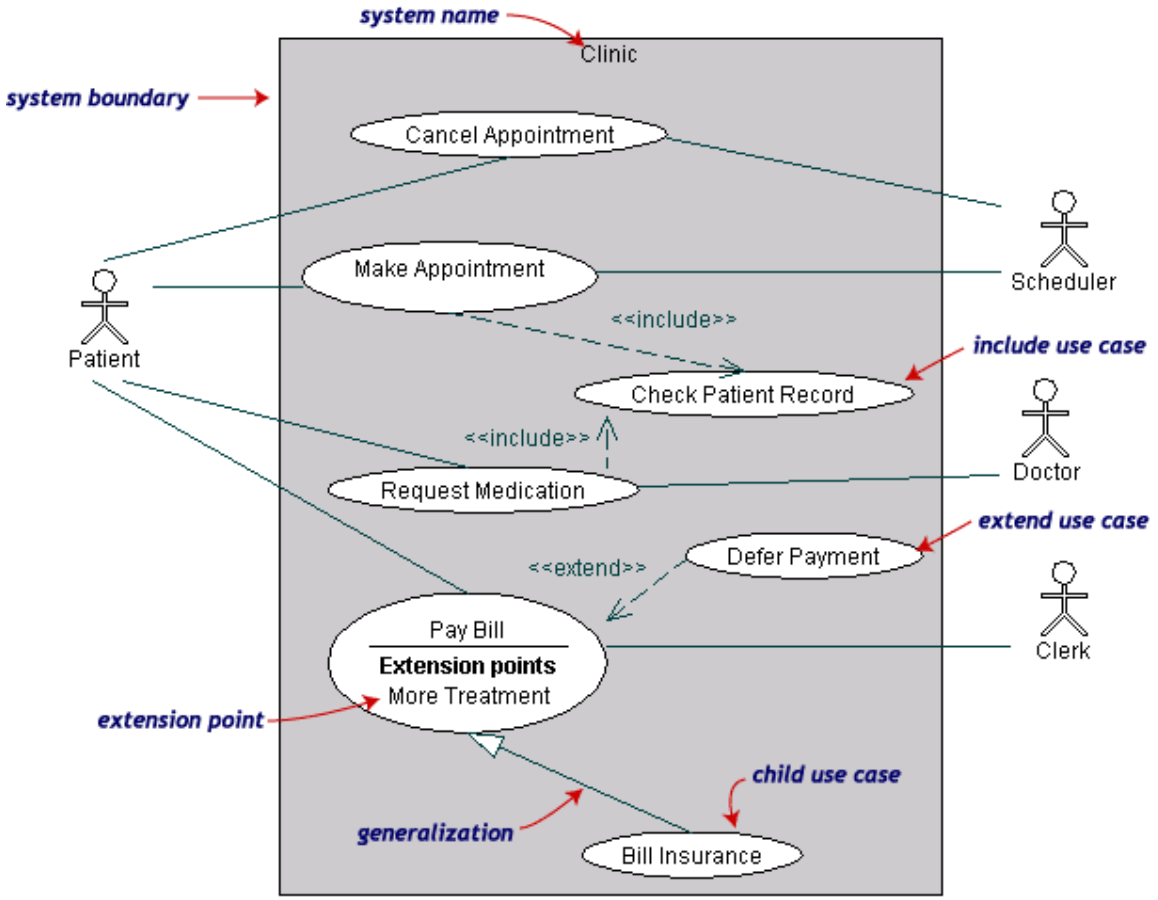


UML 2.0 Interaction Frame



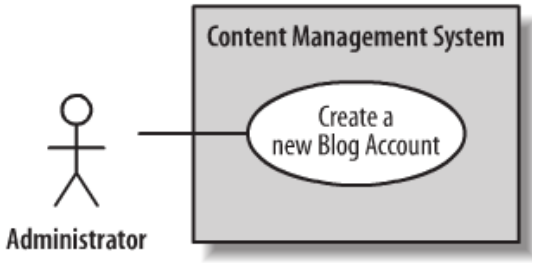
Thing #7

- The Use Case Model



http://dn.codegear.com/article/31863

Use Case Example



Use case name	Create a new Blog Account
Related Requirements	Requirement A.1.
Goal In Context	A new or existing author requests a new blog account from the `Administrator`.
Preconditions	The system is limited to recognized authors and so the author needs to have appropriate proof of identity.
Successful End Condition	A new blog account is created for the author.
Failed End Condition	The application for a new blog account is rejected.
Primary Actors	Administrator.
Secondary Actors	Author Credentials Database.
Trigger	The `Administrator` asks the CMS to create `a new blog account`.

	Step	Action
Main Flow	1	The `Administrator` asks the system to create a new blog account.
	2	The `Administrator` selects an account type.
	3	The `Administrator` enters the author's details.
	4	The author's details are verified using the Author Credentials Database.
	5	The new blog account is created.
	6	A summary of the new blog account's details are emailed to the author.
Extensions	**Step**	**Branching Action**
	4.1	The Author Credentials `Database` does not verify the author's details.
	4.2	The author's new blog account application is rejected.

Use Case Descriptions

Use case description detail	What the detail means and why it is useful
Related Requirements	Some indication as to which requirements this use case partially or completely fulfills.
Goal In Context	The use case's place within the system and why this use case is important.
Preconditions	What needs to happen before the use case can be executed.
Successful End Condition	What the system's condition should be if the use case executes successfully.
Failed End Condition	What the system's condition should be if the use case fails to execute successfully.
Primary Actors	The main actors that participate in the use case. Often includes the actors that trigger or directly receive information from a use case's execution.
Secondary Actors	Actors that participate but are not the main players in a use case's execution.
Trigger	The event triggered by an actor that causes the use case to execute.
Main Flow	The place to describe each of the important steps in a use case's normal execution.
Extensions	A description of any alternative steps from the ones described in the Main Flow.

Use Case Example (cont.)

Use case name	Create a new Blog Account
Related Requirements	Requirement A.1.
Goal In Context	A new or existing author requests a new blog account from the `Administrator`.
Preconditions	The system is limited to recognized authors and so the author needs to have appropriate proof of identity.
Successful End Condition	A new blog account is created for the author.
Failed End Condition	The application for a new blog account is rejected.
Primary Actors	Administrator.
Secondary Actors	Author Credentials Database.
Trigger	The `Administrator` asks the CMS to create `a new blog account`.

Main Flow	Step	Action
	1	The `Administrator` asks the system to create a new blog account.
	2	The `Administrator` selects an account type.
	3	The `Administrator` enters the author's details.
	4	The author's details are verified using the Author Credentials Database.
	5	The new blog account is created.
	6	A summary of the new blog account details are emailed to the author.
Extensions	**Step**	**Branching Action**
	4.1	The Author Credentials `Database` does not verify the author's details.
	4.2	The author's new blog account application is rejected.

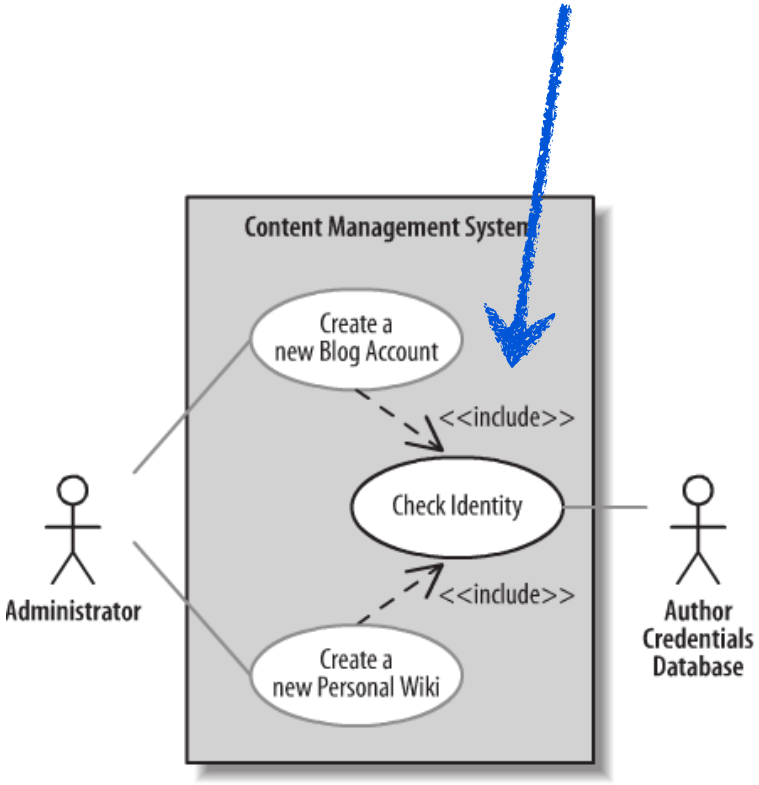
Use case name	Create a new Personal Wiki
Related Requirements	Requirement A.2.
Goal In Context	A new or existing author requests a new personal Wiki from the Administrator.
Preconditions	The author has appropriate proof of identity.
Successful End Condition	A new personal Wiki is created for the author.
Failed End Condition	The application for a new personal Wiki is rejected.
Primary Actors	Administrator.
Secondary Actors	Author Credentials Database.
Trigger	The Administrator asks the CMS to create a new personal Wiki.

Main Flow	Step	Action
	1	The Administrator asks the system to create a new personal Wiki.
	2	The Administrator enters the author's details.
	3	The author's details are verified using the Author Credentials Database.
	4	The new personal Wiki is created.
	5	A summary of the new personal Wiki's details are emailed to the author.
Extensions	**Step**	**Branching Action**
	3.1	The Author Credentials Database does not verify the author's details.
	3.2	The author's new personal Wiki application is rejected.

Why bother with all this hassle with reuse between use cases? Why not just have two use cases and maintain the similar steps separately? All this reuse has two important benefits:

- Reuse using `<<include>>` removes the need for tedious cut-and-paste operations between use case descriptions, since updates are made in only one place instead of every use case.
- The `<<include>>` relationship gives you a good indication at system design time that the implementation of `Check Identity` will need to be a reusable part of your system.

Move to "include" Use Case



Use Case Example (with <<include>>)

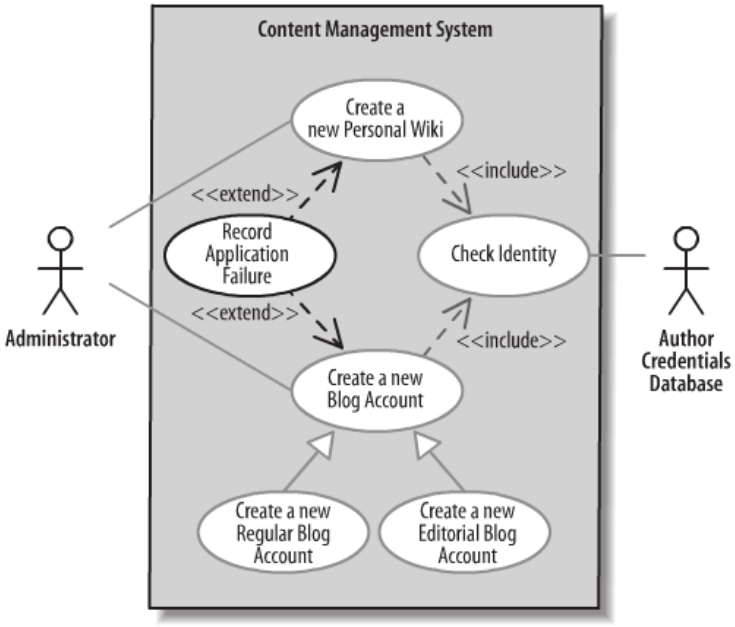
Use case name	Create a new Blog Account	
Related Requirements	Requirement A.1.	
Goal In Context	A new or existing author requests a new blog account from the Administrator.	
Preconditions	The author has appropriate proof of identity.	
Successful End Condition	A new blog account is created for the author.	
Failed End Condition	The application for a new blog account is rejected.	
Primary Actors	Administrator	
Secondary Actors	**None**	
Trigger	The Administrator asks the CMS to create a new blog account.	
Included Cases	**Check Identity**	
Main Flow	**Step**	**Action**
	1	The Administrator asks the system to create a new blog account.
	2	The Administrator selects an account type.
	3	The Administrator enters the author's details.
	4	The author's details are checked.
	include::Check Identity	
	5	The new account is created.
	6	A summary of the new blog account's details are emailed to the author.

Use case name	Create a new Personal Wiki	
Related Requirements	Requirement A.2	
Goal In Context	A new or existing author requests a new personal Wiki from the Administrator.	
Preconditions	The author has appropriate proof of identity.	
Successful End Condition	A new personal Wiki is created for the author.	
Failed End Condition	The application for a new personal Wiki is rejected.	
Primary Actors	Administrator	
Secondary Actors	**None**	
Trigger	The Administrator asks the CMS to create a new personal Wiki.	
Included Cases	**Check Identity**	
Main Flow	**Step**	**Action**
	1	The Administrator asks the system to create a new personal Wiki.
	2	The Administrator enters the author's details.
	3	The author's details are checked.
	include::Check Identity	
	5	The new personal Wiki is created.
	6	A summary of the new personal Wiki's details are emailed to the author.

Included Use Case

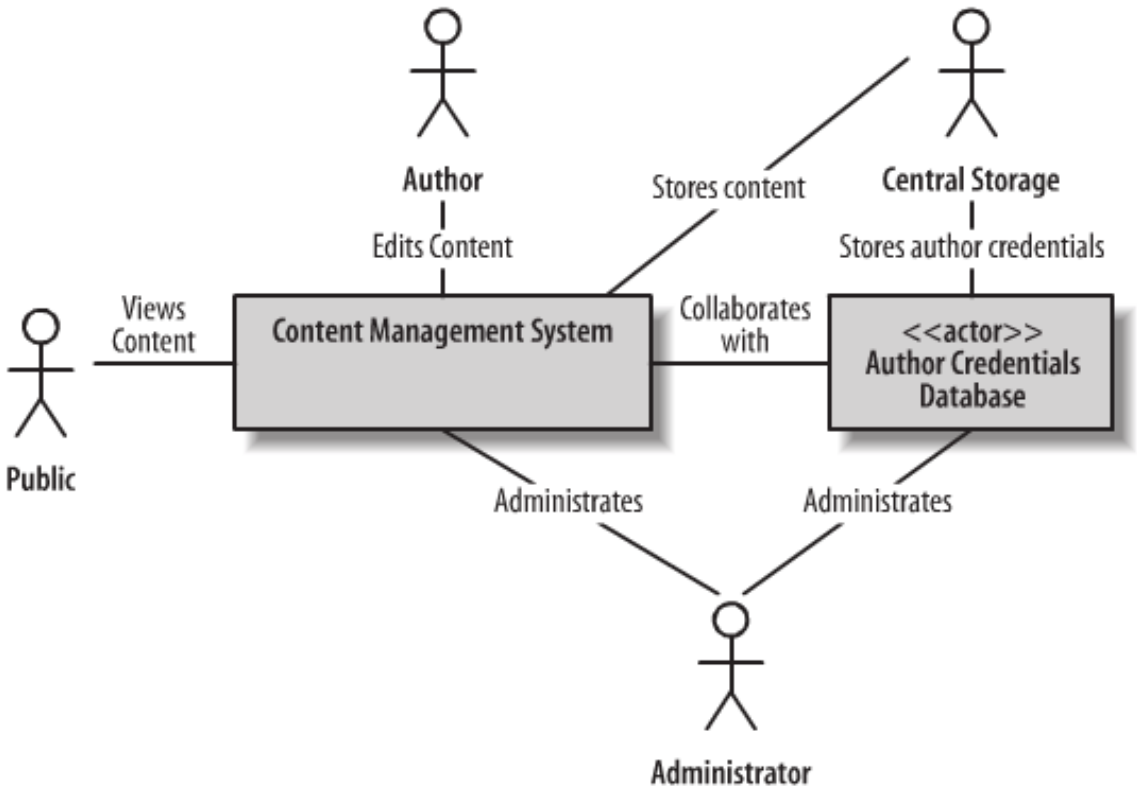
Use case name	**Check Identity**	
Related Requirements	Requirement A.1, Requirement A.2.	
Goal In Context	An author's details need to be checked and verified as accurate.	
Preconditions	The author being checked has appropriate proof of identity.	
Successful End Condition	The details are verified.	
Failed End Condition	The details are not verified.	
Primary Actors	Author Credentials Database.	
Secondary Actors	None.	
Trigger	An author's credentials are provided to the system for verification.	
Main Flow	**Step**	**Action**
	1	The details are provided to the system.
	2	The Author Credentials Database verifies the details.
	3	The details are returned as verified by the Author Credentials Database.
Extensions	**Step**	**Branching Action**
	2.1	The Author Credentials Database does not verify the details.
	2.2	The details are returned as unverified.

Use Case Extension Example



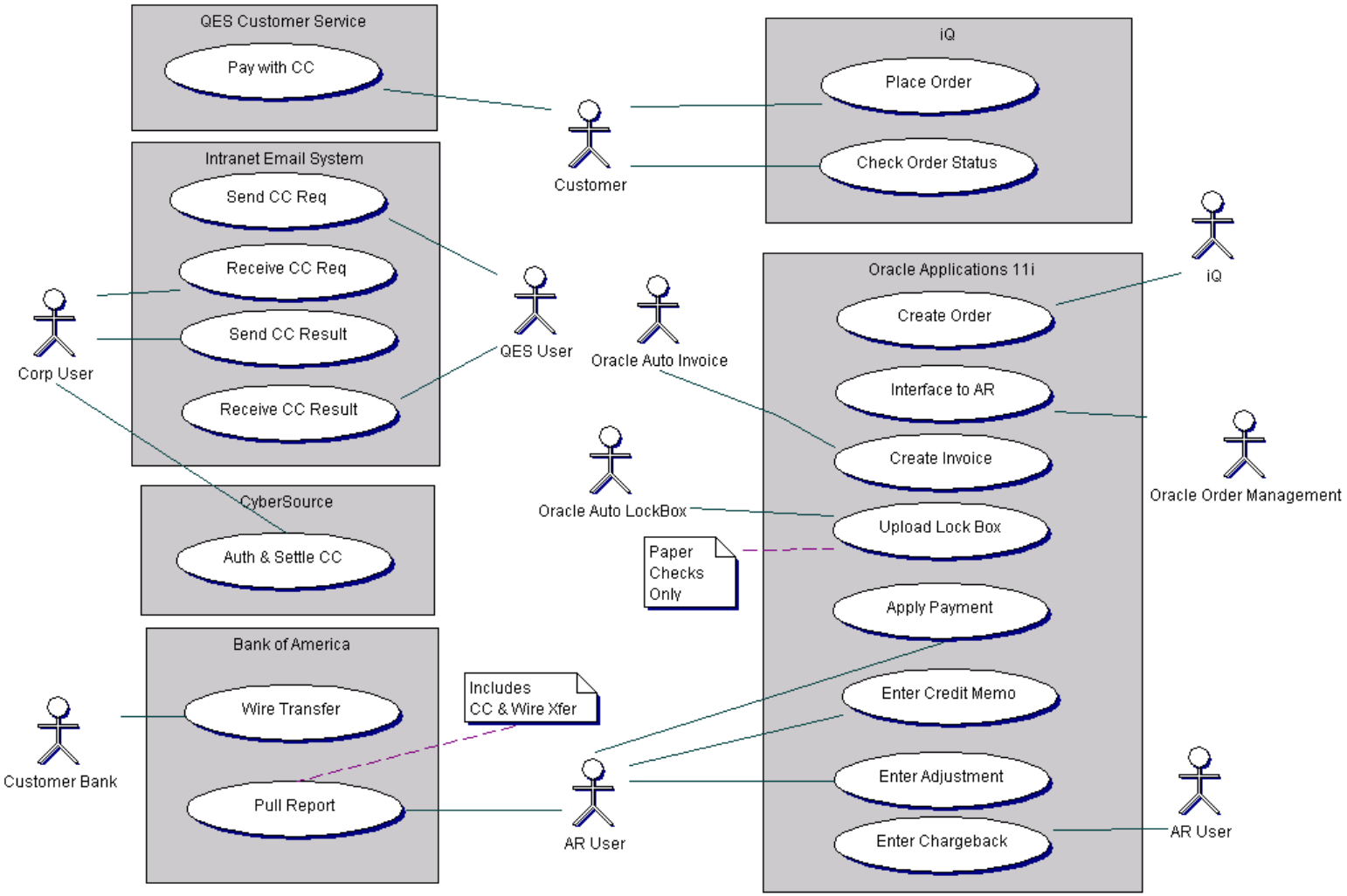
Use case name	Create a new Blog Account	
Related Requirements	Requirement A.1.	
Goal In Context	A new or existing author requests a new blog account from the Administrator.	
Preconditions	The author has appropriate proof of identity.	
Successful End Condition	A new blog account is created for the author.	
Failed End Condition	The application for a new blog account is rejected.	
Primary Actors	Administrator.	
Secondary Actors	None .	
Trigger	The Administrator asks the CMS to create a new blog account.	
Included Cases	Check Identity	
Main Flow	**Step**	**Action**
	1	The Administrator asks the system to create a new blog account.
	2	The Administrator selects an account type.
	3	The Administrator enters the author's details.
	4	The author's details are checked.
	include::Check Identity	
	5	The new account is created.
	6	A summary of the new blog account's details are emailed to the author.
Extensions	**Step**	**Branching Action**
	4.1	The author is not allowed to create a new blog.
	4.2	The blog account application is rejected.
	4.3	**The application rejection is recorded as part of the author's history.**

Use Case Overview Diagram

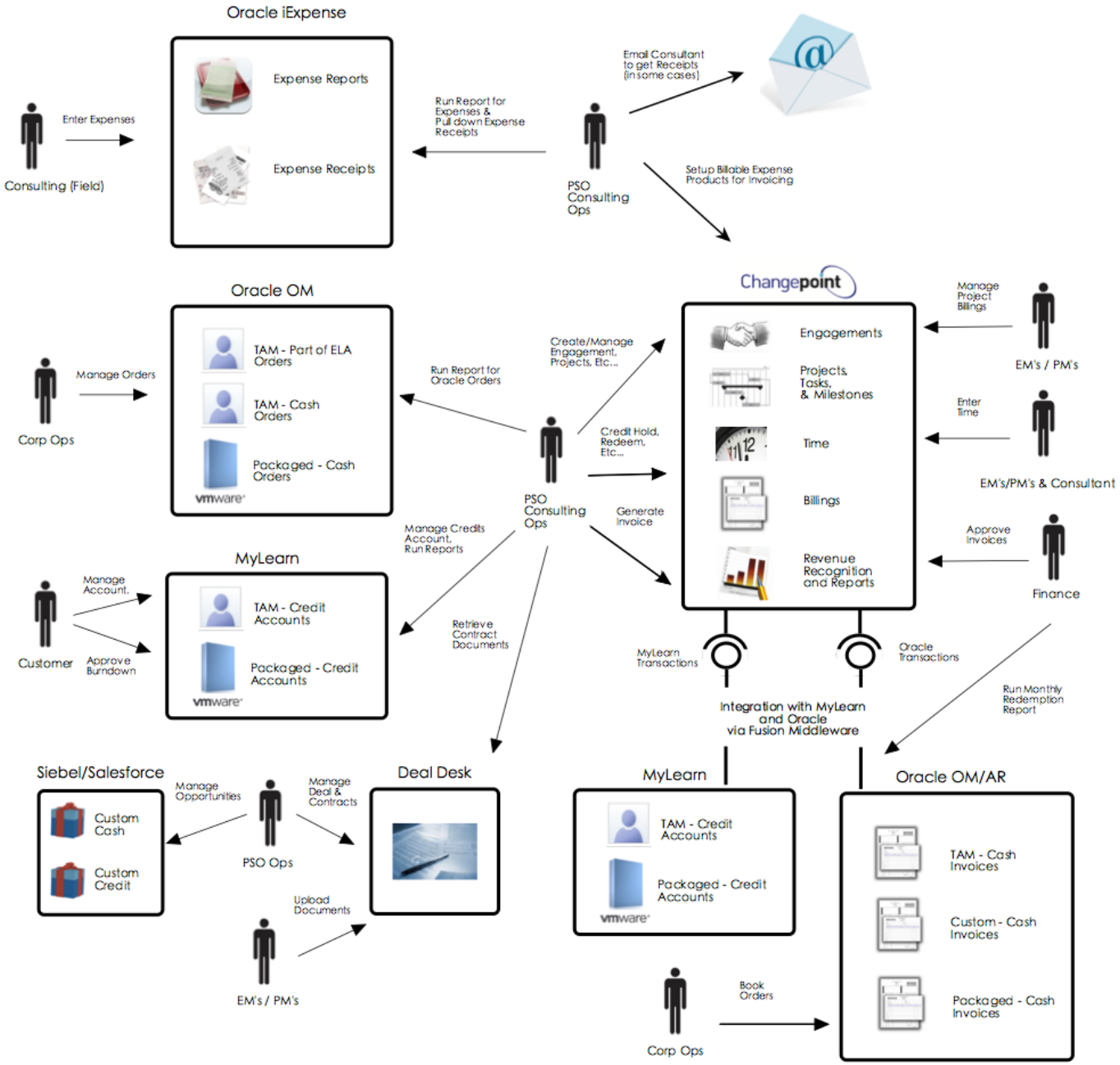


The "big picture" view of Systems and Users

“Big Picture” Use Case Models

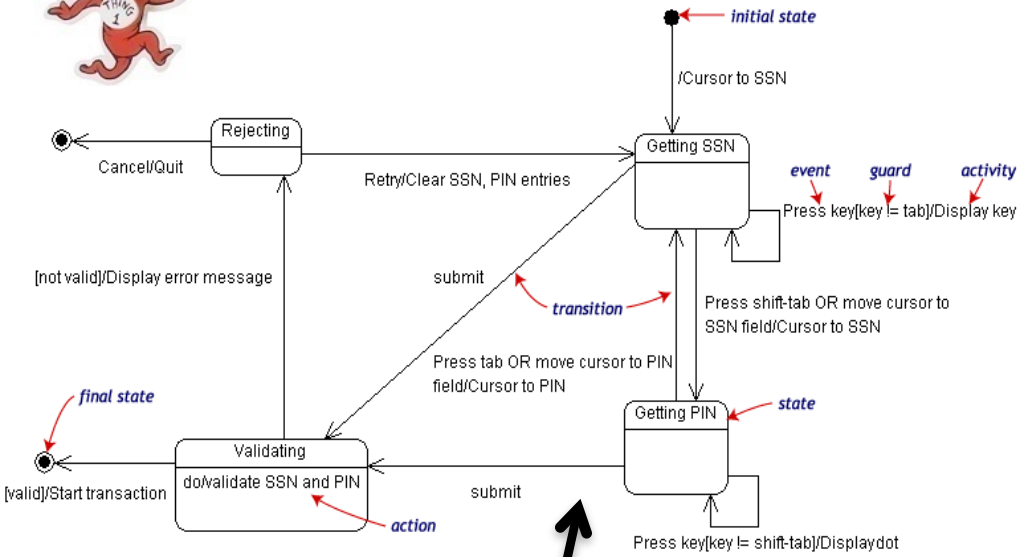


Use Case Overview (Paul's Approach)



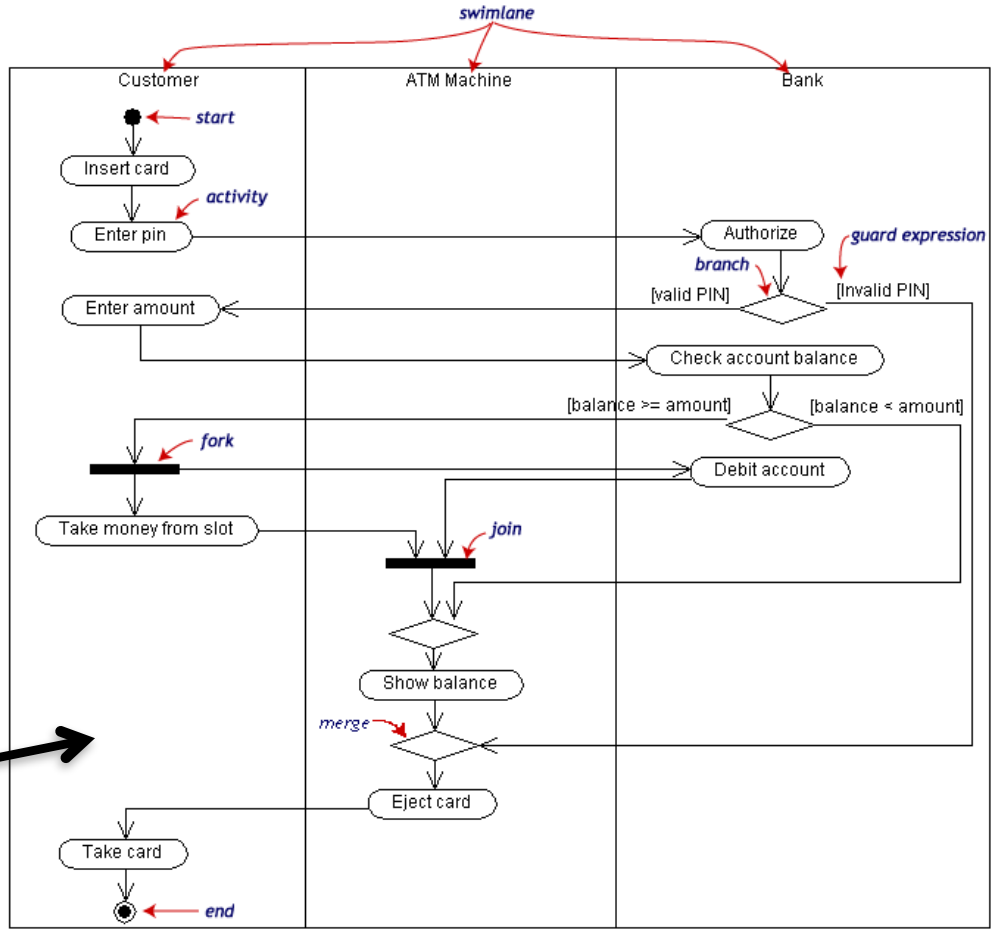
Thing #8

- State & Activity Diagrams
  - Better at showing "concurrency"… But, not good at "sequencing"



States and State Transitions of an "Object"

A "Process" model showing object responsibilities (i.e. activity) typically In a single use case.



http://dn.codegear.com/article/31863

State Diagram

State Machines

State machine diagrams are heavily used in special niches of software and hardware systems, including the following:

- Real-time/mission-critical systems, such as heart monitoring software
- Dedicated devices whose behavior is defined in terms of state, such as ATMs
- First-person shooter games, such as *Doom* or *Half-Life*

To reflect these common uses, this chapter will deviate from the CMS example used throughout the rest of this book.

Most of this chapter focuses on behavioral state machines, which can show states, transitions, and behavior (inside states and along transitions). There's another type of state machine called a *protocol state machine* that doesn't model behavior but is useful for modeling protocols such as network communication protocols. Protocol state machines are discussed briefly at the end of the chapter.

Two kinds of State Machines

States, Events & Transitions

Essentials

Let's look at the key elements of state diagrams using a simple example. Figure 14-2 shows a state diagram modeling a light. When you lift the light switch, the light turns on. When you lower the light switch, the light turns off.

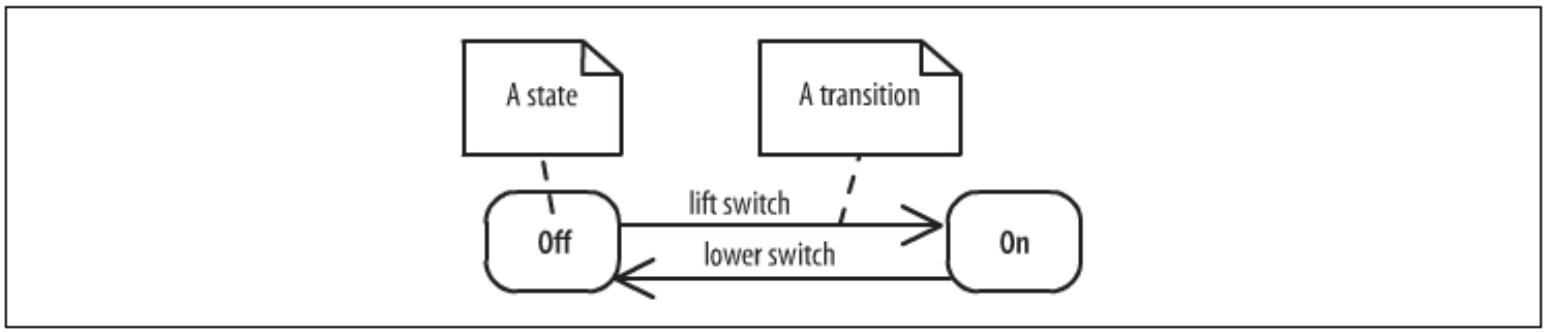


Figure 14-2. The fundamental elements of a state diagram: states and transitions between states

A state diagram consists of *states*, drawn as rounded rectangles, and *transitions*, drawn as arrows connecting the states. A transition represents a change of state or how to get from one state to the next. A state is *active* when entered through a transition, and it becomes *inactive* when exited through a transition.

The event causing the state change, or *trigger*, is written along the transition arrow. The light in Figure 14-2 has two states: `Off` and `On`. It changes state when the `lift switch` or `lower switch` triggers occur.

State Transition Table vs. State Diagram

Table 14-1. Table view of light states and transitions—not UML notation

State/Trigger	Light switch lifted	Light switch lowered
`Off`	`On`	-
`On`	-	`Off`

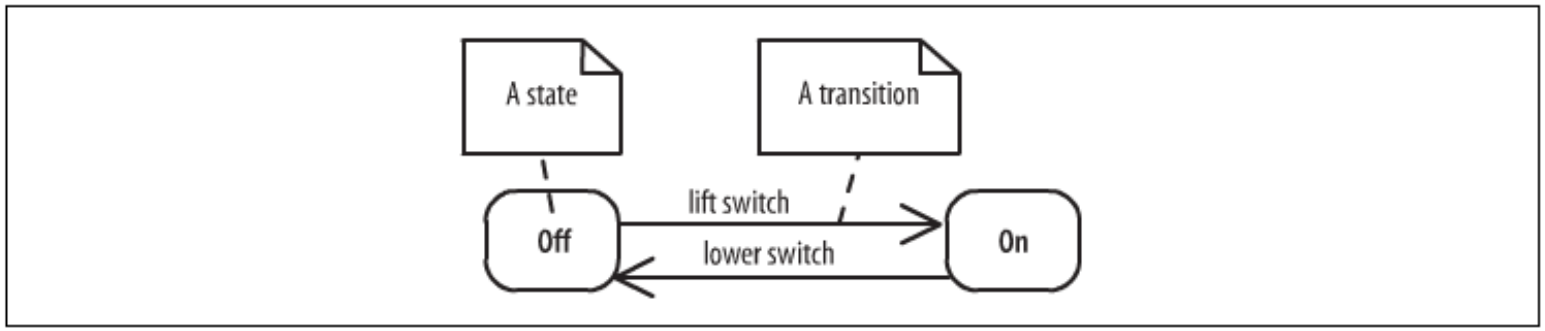


Figure 14-2. The fundamental elements of a state diagram: states and transitions between states

Pseudostates

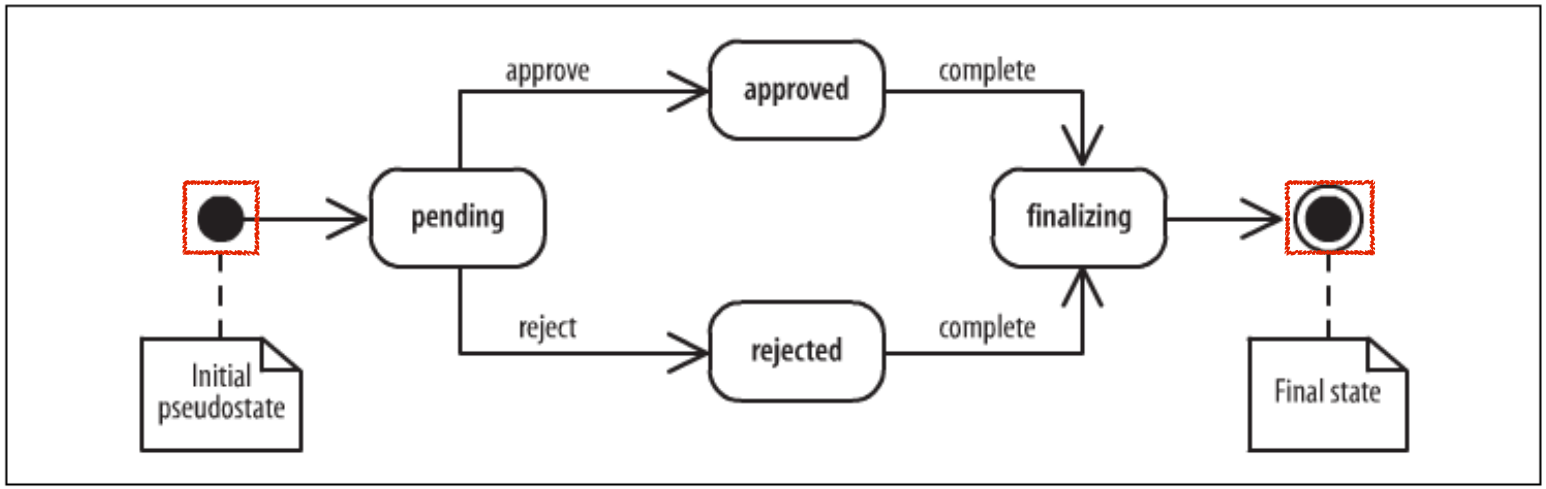


Figure 14-3. Initial pseudostate and final states in an AccountApplication state diagram

Active States & Do Behavior

States

A *state* is a condition of being at a certain time. A state can be a passive quality, such as `On` and `Off` for the light object. A state can also be an active quality, or something that an object is doing. For example, a coffeemaker has the state `Brewing` during which it is brewing coffee. A state is drawn as a rounded rectangle with the name of the state in the center, as shown in Figure 14-4.

If the state is a "doing" state, you can write the behavior inside the state, as shown in Figure 14-5.

Figure 14-4. A rectangle with rounded corners and the name in the center is the most common way to draw a state

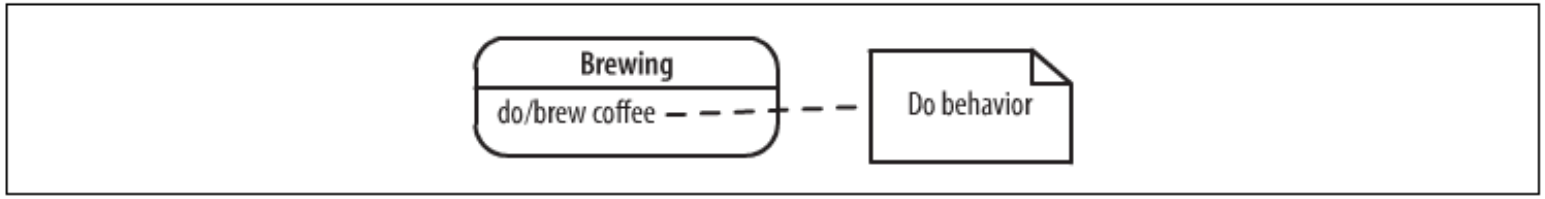


Figure 14-5. Showing the behavior details of a "doing" state

`Do` behavior, written as `do/`*`behavior`*, is behavior that happens as long as the state is active. For example, the coffeemaker in Figure 14-5 does the behavior `brew coffee` while in the `Brewing` state.

State Transitions

Transitions

A transition, shown with an arrow, represents a change of states from a *source state* to a *target state*. A *transition description*, written along the arrow, describes the circumstances causing the state change to occur.

The previous state diagrams in this chapter had fairly simple transition descriptions because they consisted only of triggers. For example, the light in Figure 14-2 changed state in response to the triggers `lift switch` and `lower switch`. But transition descriptions can be more complex. The full notation for transition descriptions is *`trigger[guard] / behavior`*, where each element is optional, as shown in Figure 14-6. This section defines each of these elements, and then in "Transition Variations" we'll show how these elements interact to model different types of state changes.

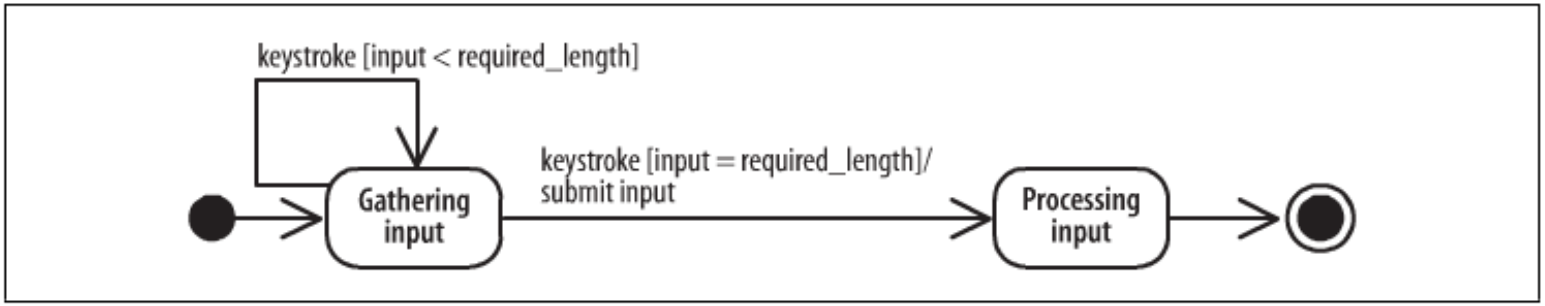


Figure 14-6. This input processing state diagram models features a trigger, guard, and transition behavior along one of its transitions

Entry/Exit Behavior

Internal Behavior

Internal behavior is any behavior that happens while the object is in a state. You've already seen `do` behavior, which is behavior that is ongoing while the state is active. Internal behavior is a more general concept that also includes entry and exit behavior.

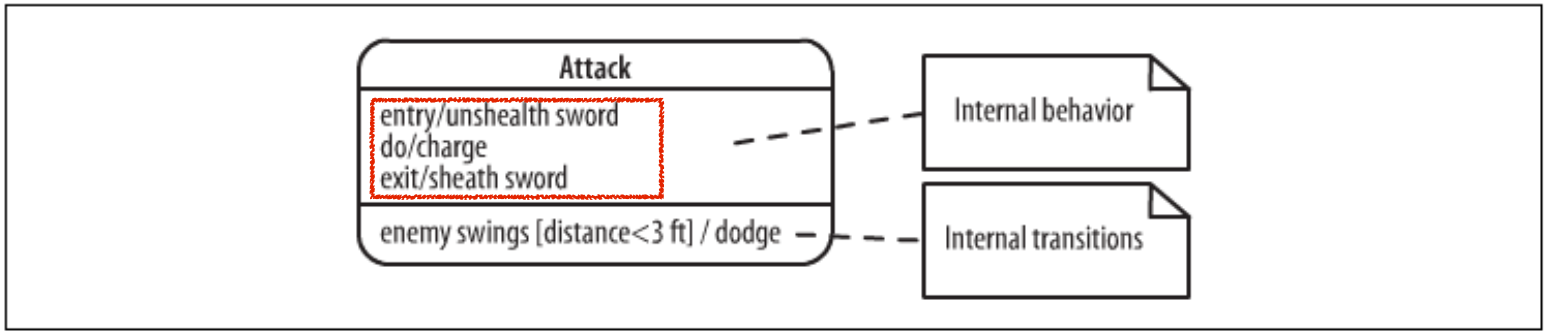


Figure 14-14. Internal behavior and transitions of the Attack state

Internal behavior is written as *`label / behavior`*. The label indicates when the behavior executes—in other words, events or circumstances causing the behavior. There are three special labels: `entry`, `exit`, and `do`.

Entry behavior happens as soon as the state becomes active and is written as `entry/` *`behavior`*. Exit behavior happens immediately before the state becomes inactive and is written as `exit/`*`behavior`*.

Internal Transitions

Internal Transitions

An *internal* transition is a transition that causes a reaction within a state, but doesn't cause the object to change states. An internal transition is different from a self transition (see Figure 14-11) because self transitions cause entry and exit behavior to occur whereas internal transitions don't.

Internal transitions are written as *`trigger [guard] / behavior`*, and they are listed inside a state. In Figure 14-16, the `Attack` has an internal transition: when an opponent swings his weapon and is less than three feet away, the troll dodges.

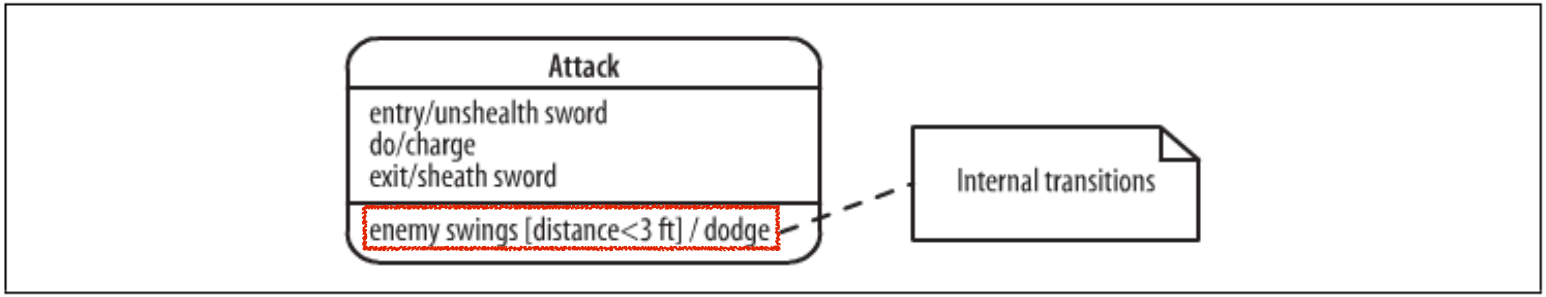


Figure 14-16. The bottom compartment shows internal transitions

Protocol State Machines

Protocol State Machines

Protocol state machines are a special kind of state machine focusing on how a protocol, such as a communication protocol (e.g., TCP), works. The main difference between protocol state machines and behavioral state machines, which we've focused on previously, is that protocol state machines don't show behavior along transitions or inside states. Instead, they focus on showing a legal sequence of events and resulting states. Protocol state machines are drawn in a tabbed rectangle with the name of the state machine in the tab followed by {protocol}, as shown in Figure 14-22.

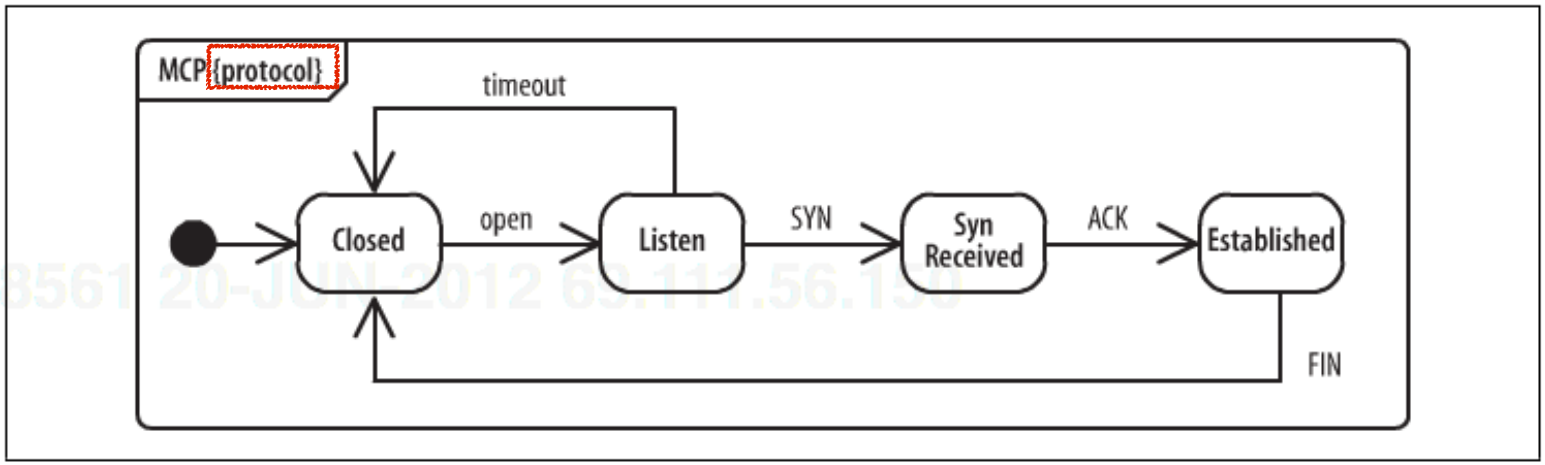


Figure 14-22. Protocol state machine modeling the receiver side of a simplified communication protocol called My Communication Protocol (MCP)

Payment State Model

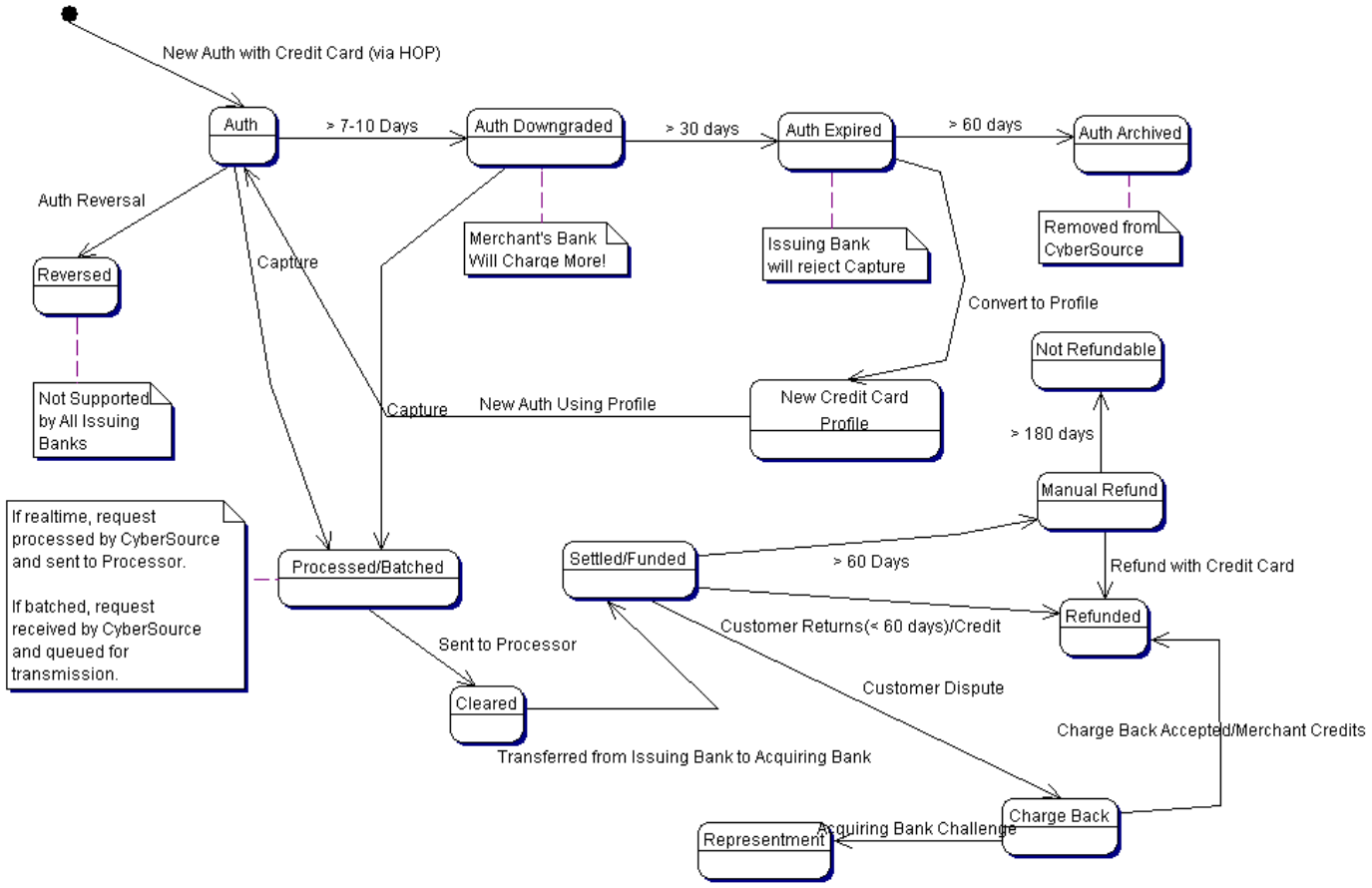
event_type

(credit card, bank transfer, or direct debit transactions)

Type of event that occurred for the transaction:

- `Chargeback`: The customer did not authorize the transaction. For more information, see processor_message.
- `Correction`: The payment or refund was corrected, or the bank was unable to credit the customer's account; the value is either positive or negative.
- `Failed`: The account was invalid or disabled. For more details about the decline, see processor_message.
- `Other`: The processor reported an unanticipated event.
- `Payment`: The payment was received by the processor's bank; the value is always positive.
- `Refund`: The payment was returned; the value is always negative. For more details about the refund, see processor_message.
- `Reversal`: A payment was reversed. For more details about the reversal, see processor_message.
- `Settled`: The transaction has been settled: the payment has been received, or the refund has been given to the customer.
- `Settled Unmatched`: A bank transfer payment has been received but cannot be matched to the original request.

Alpha (20)



Activity Diagrams

Activity Diagram - Basic Elements

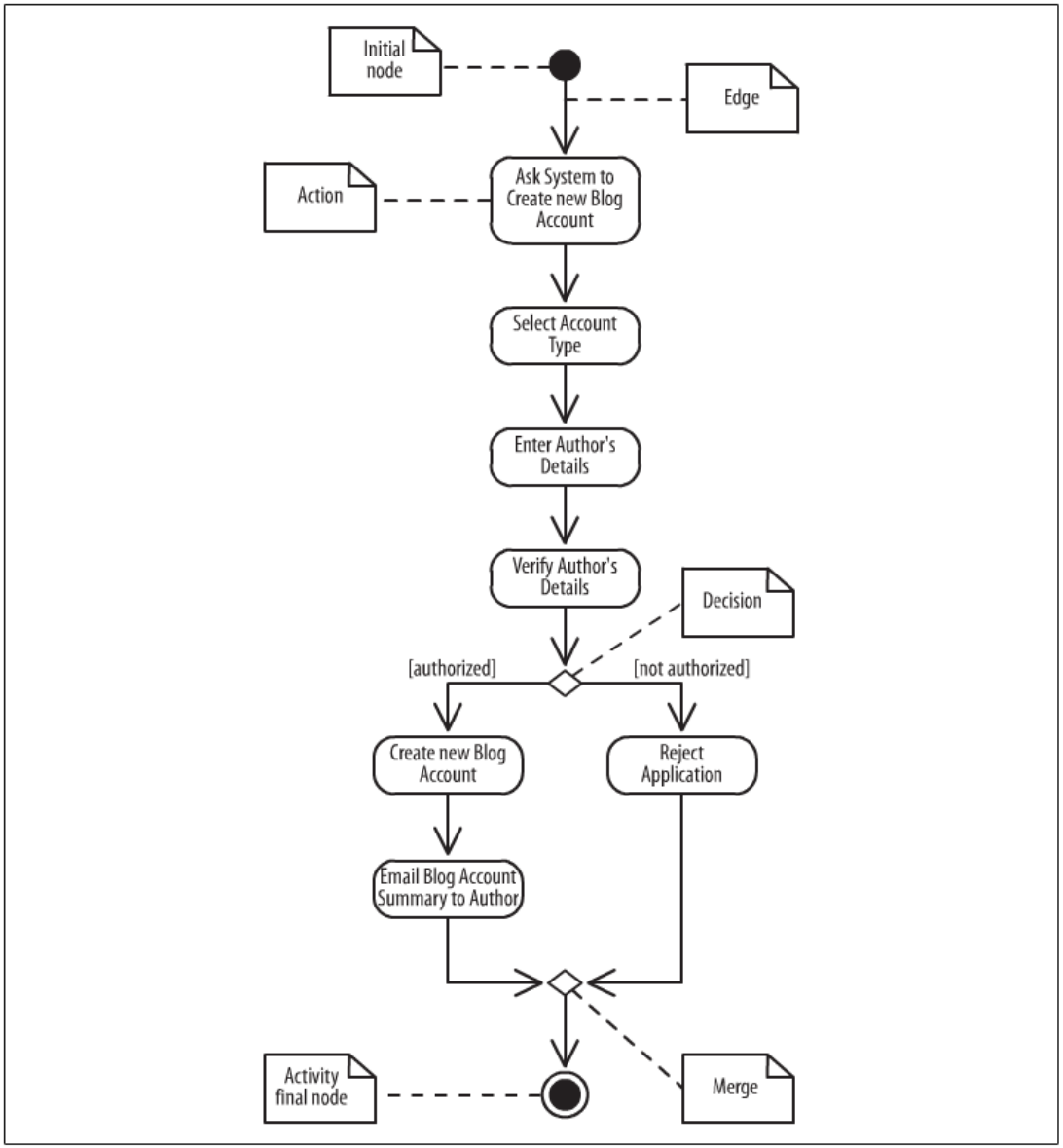


Figure 3-2. Activity diagrams model dynamic behavior with a focus on processes; the basic elements of activity diagrams are shown in this blog account creation process

Activity Diagram - Actions

The actions in this simple car-washing activity are shown in Figure 3-3.

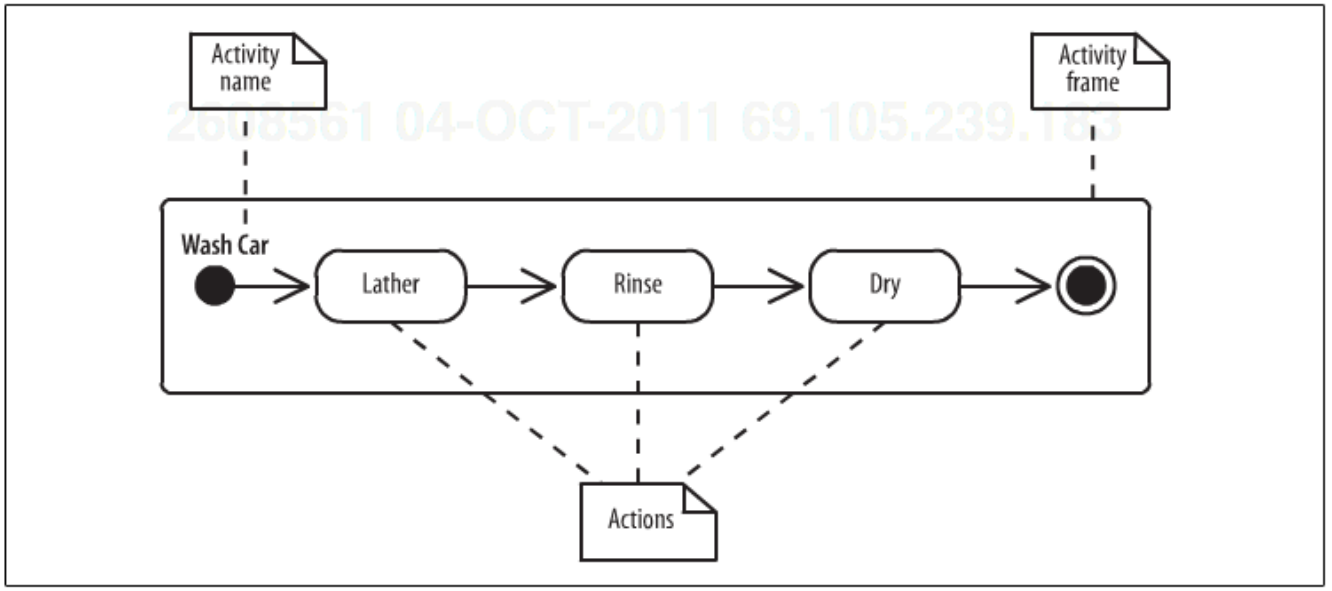


Figure 3-3. Capturing the three actions—Lather, Rinse, and Dry—that make up washing a car in an activity diagram

The activity frame is optional and is often left out of an activity diagram, as shown in the alternative `Wash Car` activity in Figure 3-4.

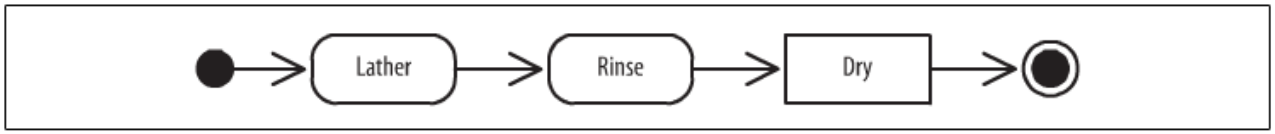


Figure 3-4. The activity frame can be omitted

Activity Diagram - Decisions

Decisions are used when you want to execute a different sequence of actions depending on a condition. Decisions are drawn as diamond-shaped nodes with one incoming edge and multiple outgoing edges, as shown in Figure 3-5.

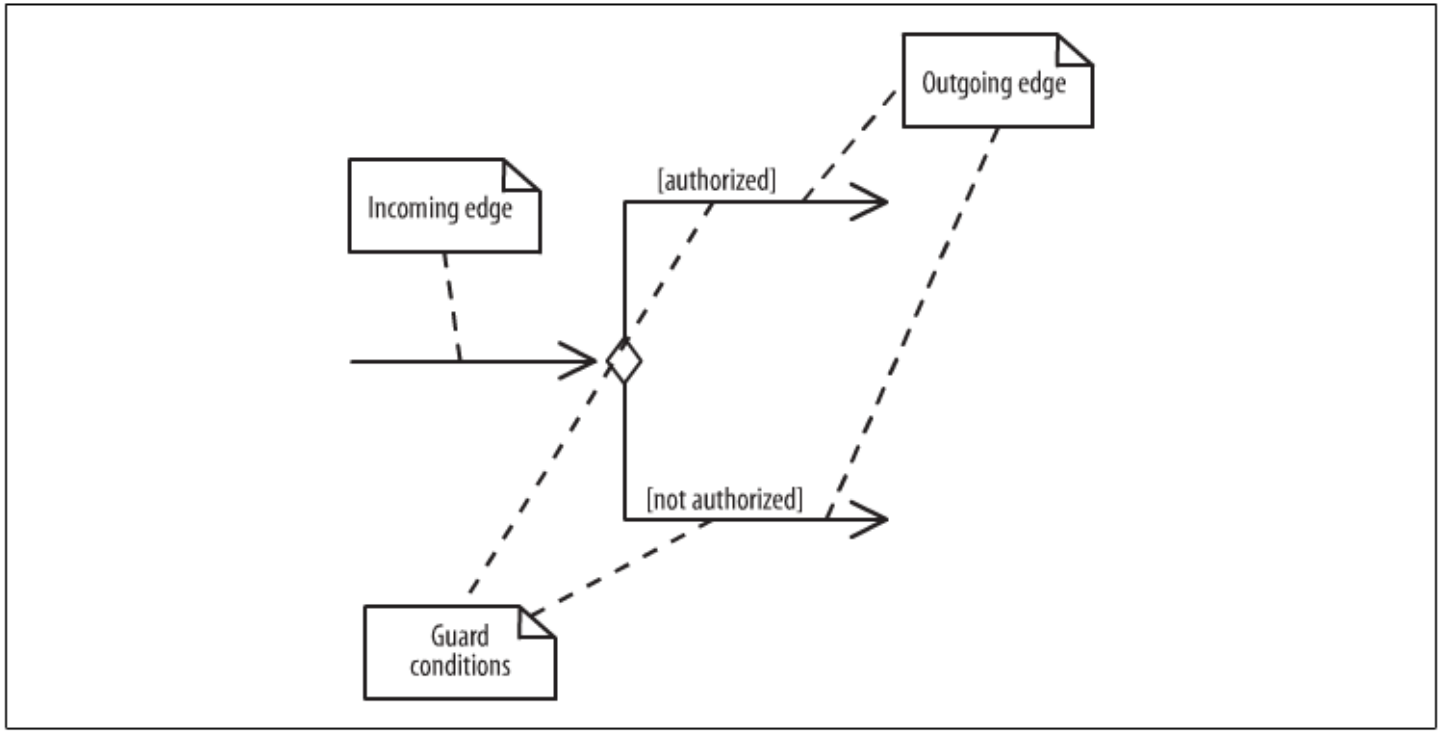


Figure 3-5. Only one edge is followed after a decision node

Each branched edge contains a *guard condition* written in brackets. Guard conditions determine which edge is taken after a decision node.

Activity Diagram - Merges

The branched flows join together at a *merge* node, which marks the end of the conditional behavior started at the decision node. Merges are also shown with diamond-shaped nodes, but they have multiple incoming edges and one outgoing edge, as shown in Figure 3-6.

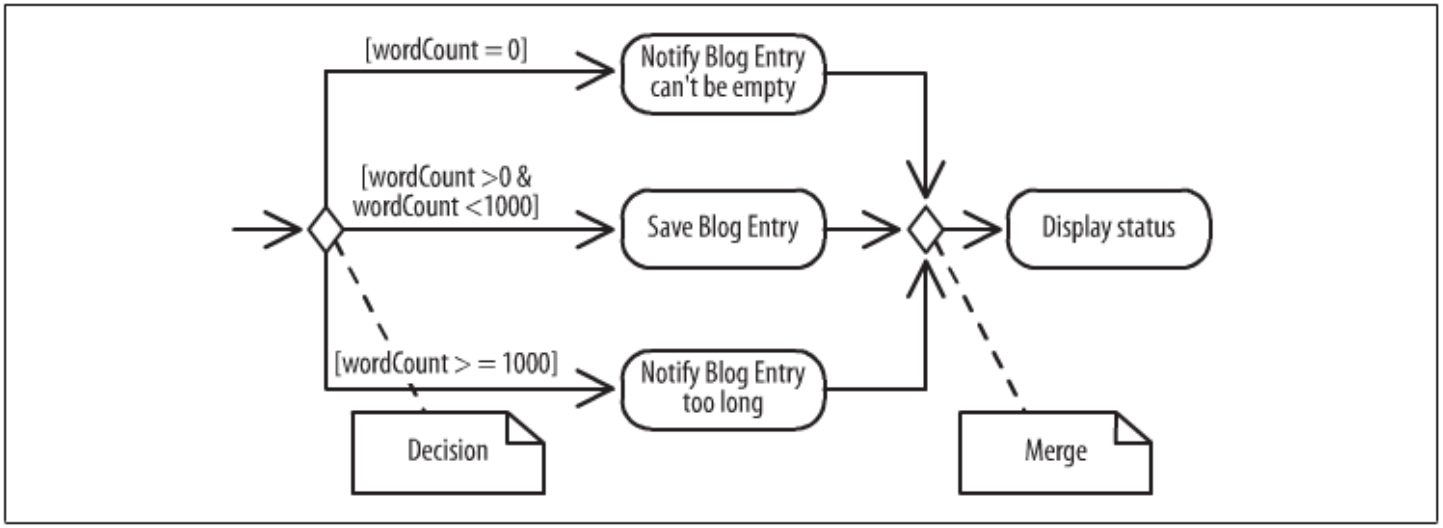


Figure 3-6. If the input value of age is 1200, then the Notify Blog Entry too long action is performed

Activity diagrams are clearest if the guards at decision nodes are complete and mutually exclusive. Figure 3-7 shows a situation in which the paths are not mutually exclusive.

Activity Diagram - Invalid Guards

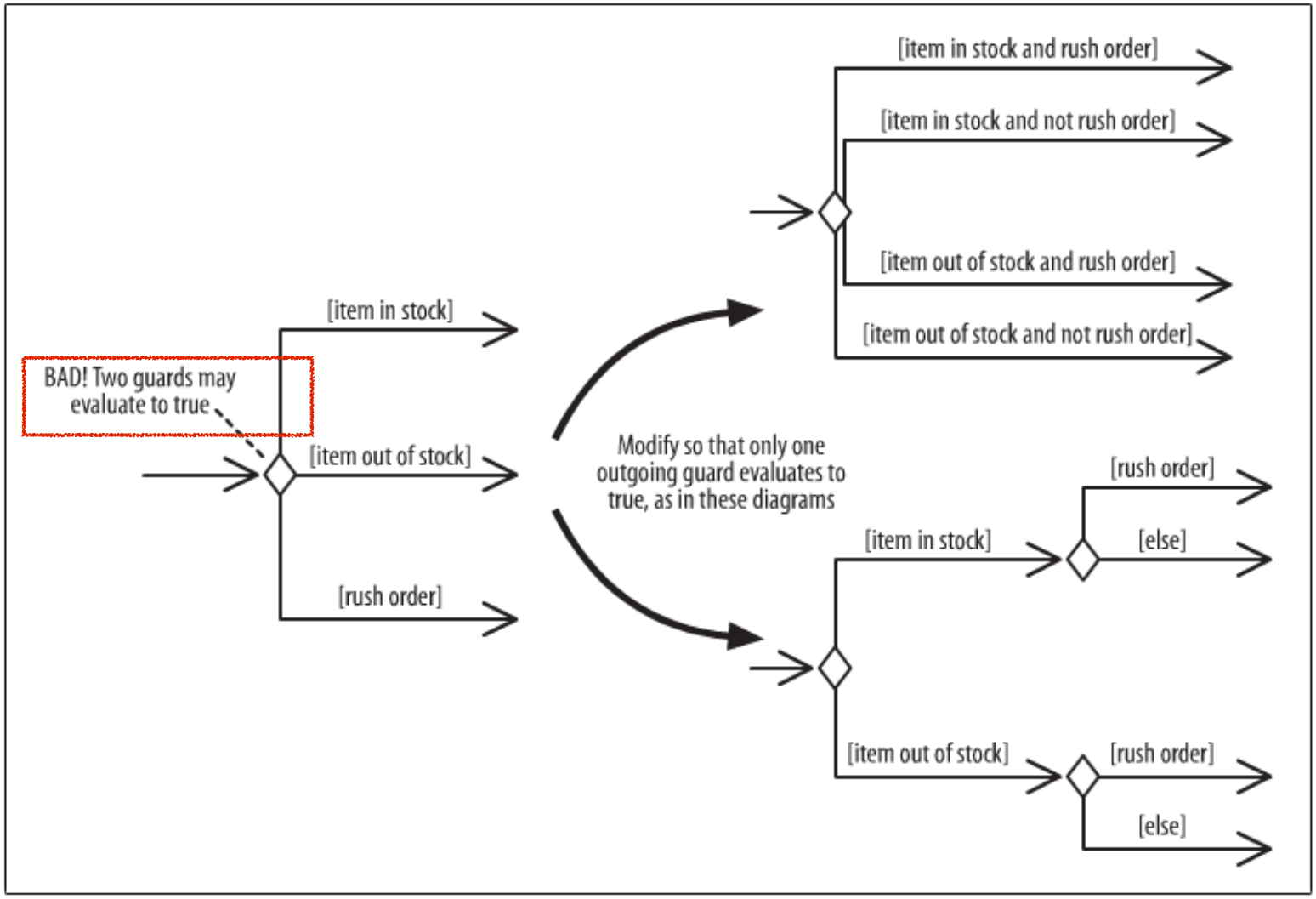


Figure 3-7. Beware of diagrams where multiple guards evaluate to true

Activity Diagram - UML 1.x to 2.0

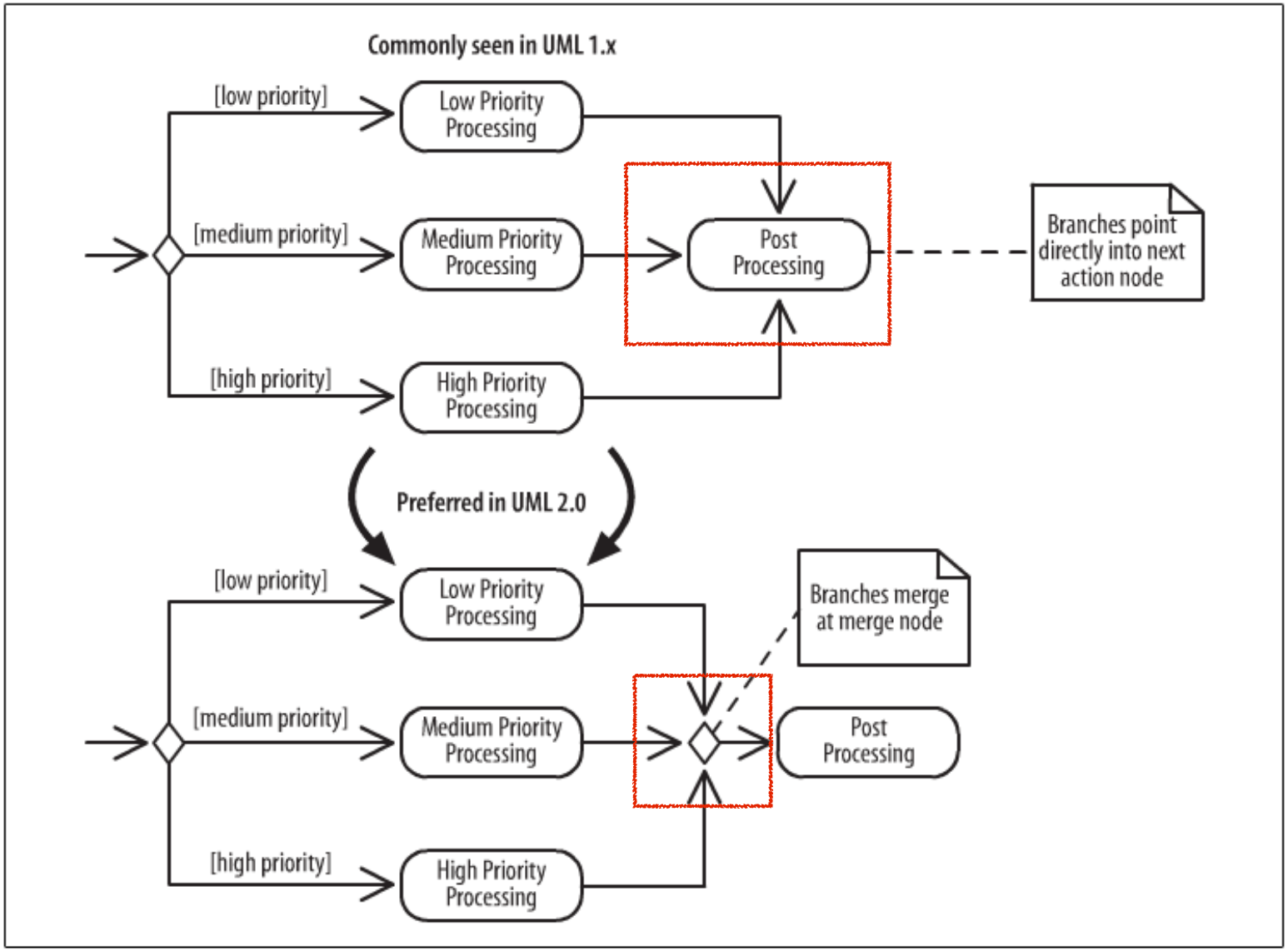


Figure 3-8. In UML 2.0, it's better to be as clear as possible and to show merge nodes

Activity Diagram - Forks & Joins

Figure 3-10 completes the activity diagram for the computer assembly workflow.

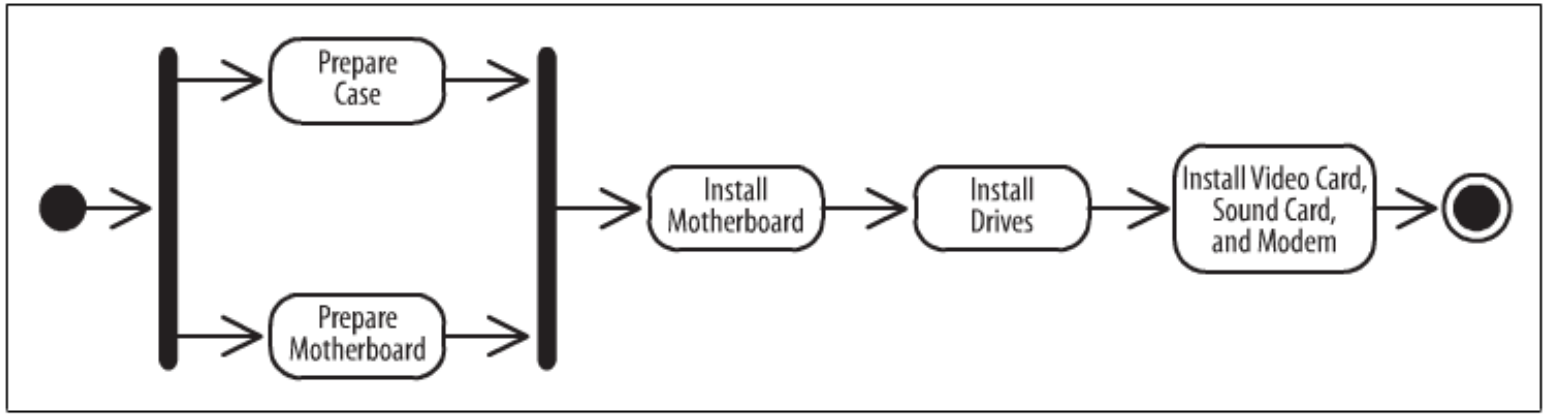


Figure 3-10. The computer assembly workflow demonstrates how forks and joins work in a complete activity diagram

When actions occur in parallel, it doesn't necessarily mean they will finish at the same time. In fact, one task will most likely finish before the other. However, the join prevents the flow from continuing past the join until all incoming flows are complete. For example, in Figure 3-10 the action immediately after the join—`Install Motherboard`—executes only after both the `Prepare Case` and `Prepare Motherboard` actions finish.

Activity Diagram - Timers

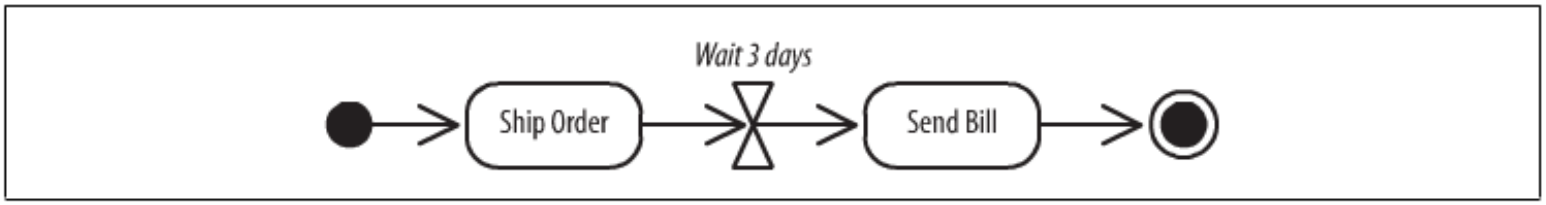


Figure 3-11. A time event with an incoming edge represents a timeout

A time event with no incoming flows is a *recurring* time event, meaning it's activated with the frequency in the text next to the hourglass. In Figure 3-12, the progress bar is updated every second.

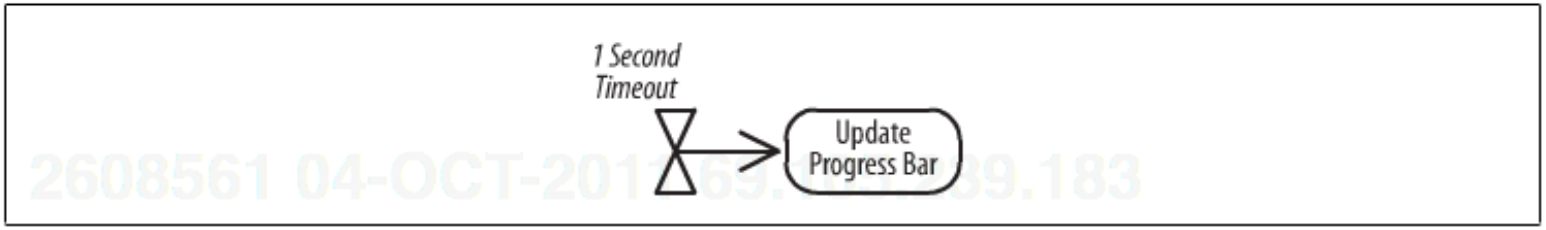


Figure 3-12. A time event with no incoming flows models a repeating time event

Notice that there is no initial node in Figure 3-12; a time event is an alternate way to start an activity. Use this notation to model an activity that is launched periodically.

Activity Diagram - Call Activity

Figure 3-13 shows the computer assembly workflow from Figure 3-10, but the `Prepare Motherboard` action now has an upside-down pitchfork symbol indicating that it is a *call activity* node. A call activity node calls the activity corresponding to its node name. This is similar to calling a software procedure.

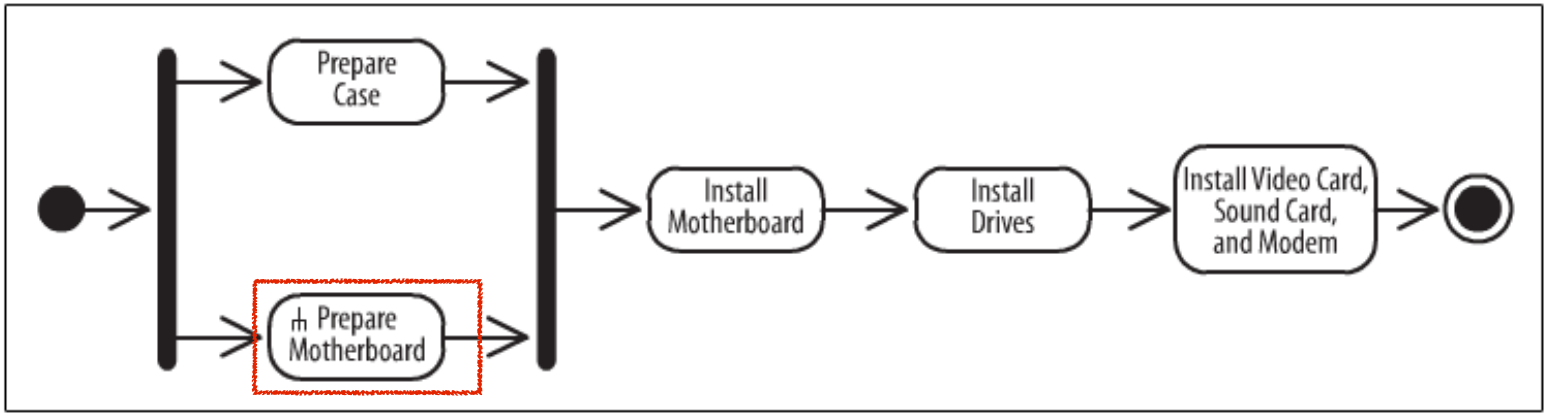


Figure 3-13. Rather than cluttering up the top-level diagram with details of the Prepare Motherboard action, details are provided in another activity diagram

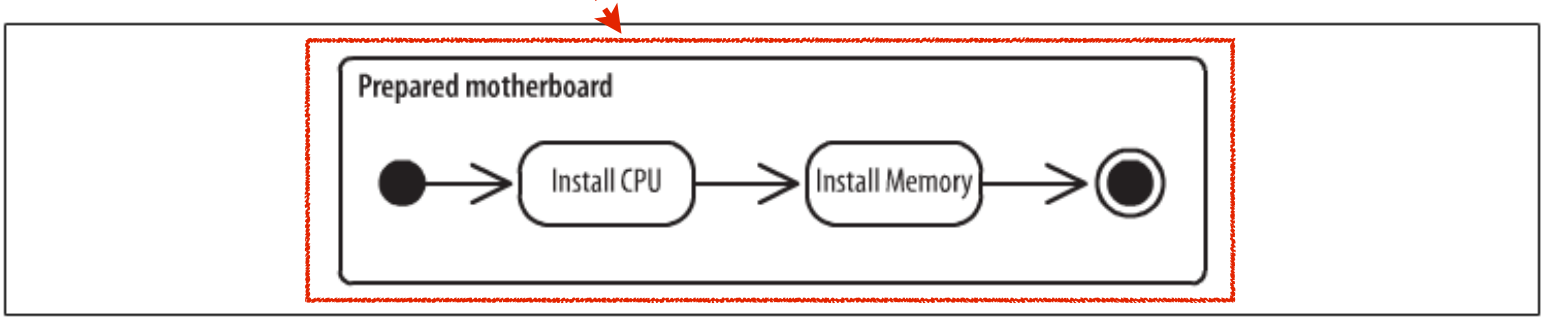


Figure 3-14. The Prepare Motherboard activity elaborates on the motherboard preparation process

Activity Diagram - Sending Objects

An object node is drawn with a rectangle, as shown in the order approval process in Figure 3-15. The `Order` object node draws attention to the fact that the `Order` object flows from the `Receive Order Request` action to the `Approve Payment` action.

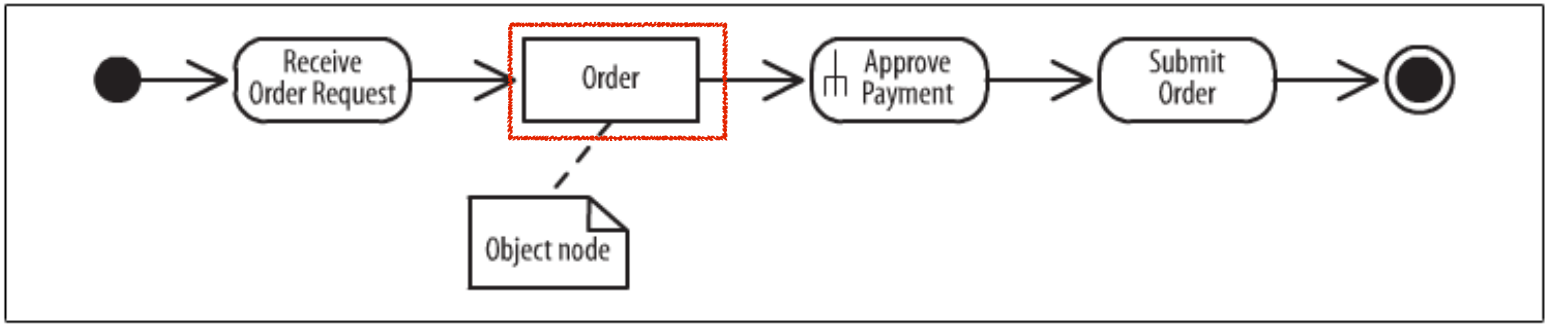


Figure 3-15. The Order object node emphasizes that it is important data in this activity and shows which actions interact with it

Activity Diagram - I/O Pins

An *input pin* means that the specified object is input to an action. An *output pin* means that the specified object is output from an action. In Figure 3-16, an `Order` object is input to the `Approve Payment` action and an `Order` object is output from the `Receive Order Request` action.

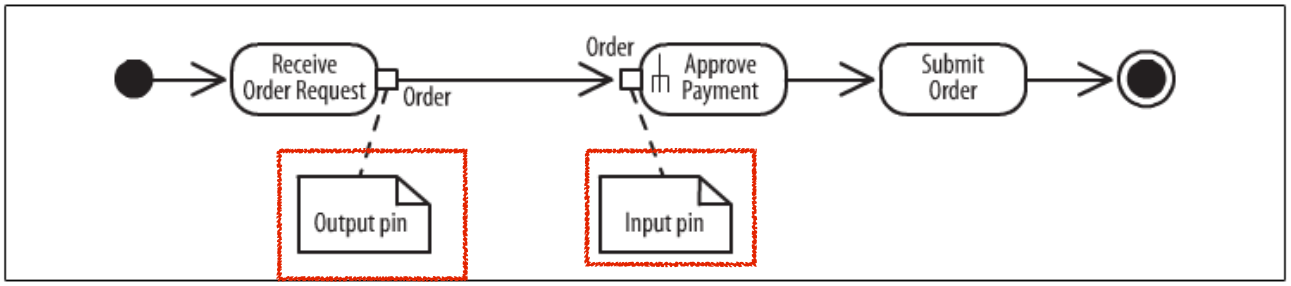


Figure 3-16. Pins in this change request approval process allow finer-grained specification of input and output parameters

Figure 3-17 specifies that the `Approve Payment` action requires the `Cost` object as input and shows how this data is obtained from the `Order` object using the transformation specified in a note.

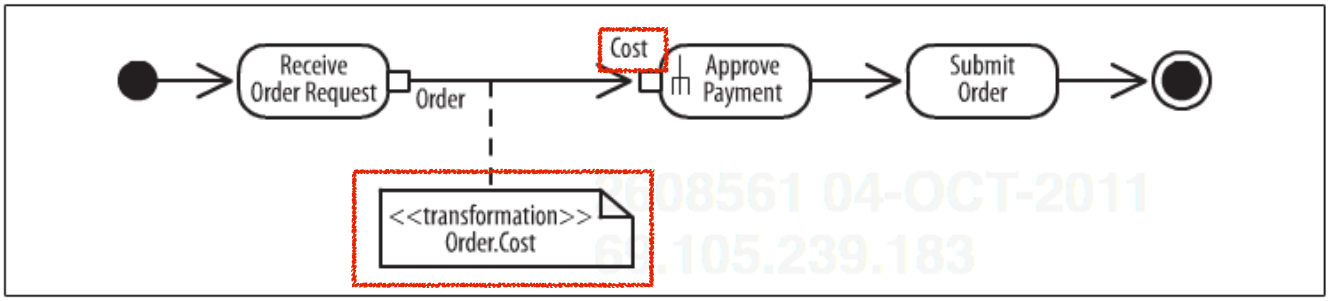


Figure 3-17. Transformations show where input parameters come from

Activity Diagram - Object State Changes

You can also show an object changing state as it flows through an activity. Figure 3-18 shows that the `Order` object's state is `pending` before `Approve Payment` and changes to `approved` afterward. The state is shown in brackets.

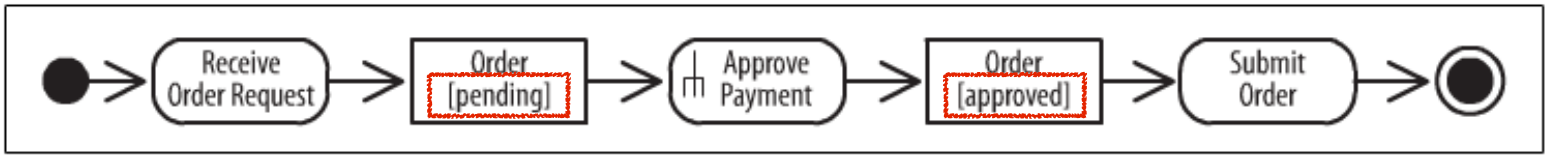


Figure 3-18. The focus of this diagram is the change of state of the Order object throughout the order approval process

Activity Diagram - Signals

Activities may involve interactions with external people, systems, or processes. For example, when authorizing a credit card payment, you need to verify the card by interacting with an approval service provided by the credit card company.

In activity diagrams, *signals* represent interactions with external participants. Signals are messages that can be sent or received, as in the following examples:

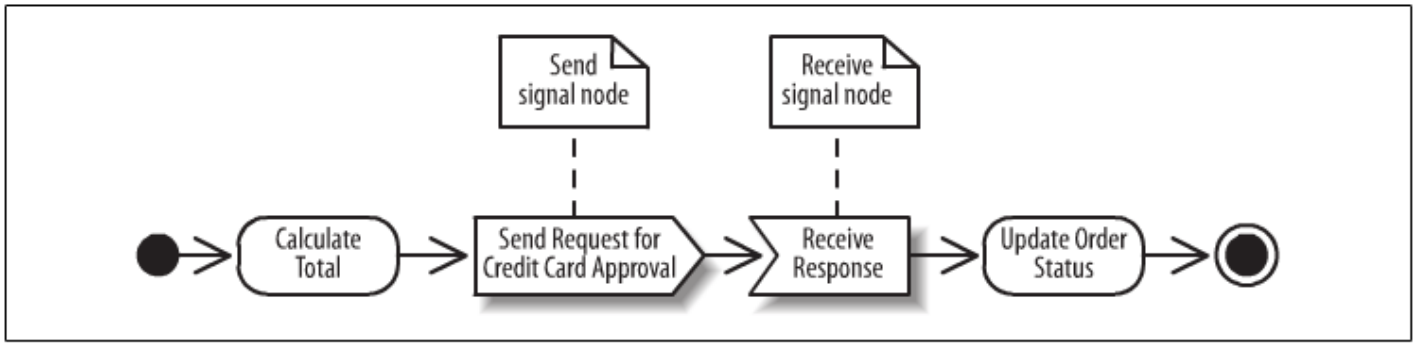


Figure 3-20. Send and receive signal nodes show interactions with external participants

When you see a receive signal node with no incoming flows, it means that the node is *always* waiting for a signal when its containing activity is active. In the case of Figure 3-21, the activity is launched every time an account request signal is received.

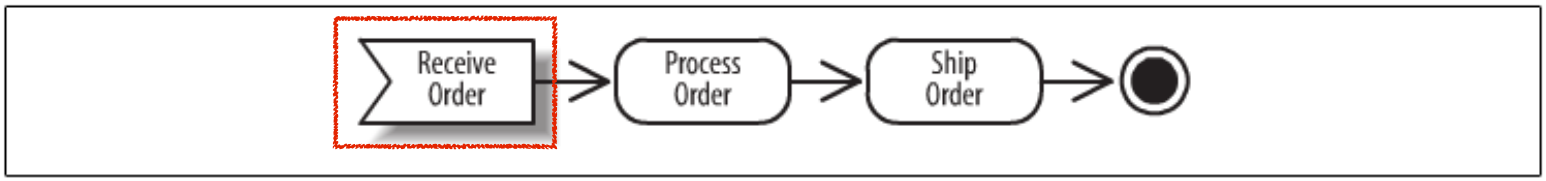


Figure 3-21. Starting an activity with a receive signal node: the receive signal node replaces the usual initial node

Activity Diagram - Interrupts

Draw an interruption region with a dashed, rounded rectangle surrounding the actions that can be interrupted along with the event that can cause the interruption. The interrupting event is followed by a line that looks like a lightning bolt. Figure 3-22 extends Figure 3-21 to account for the possibility that an order might be canceled.

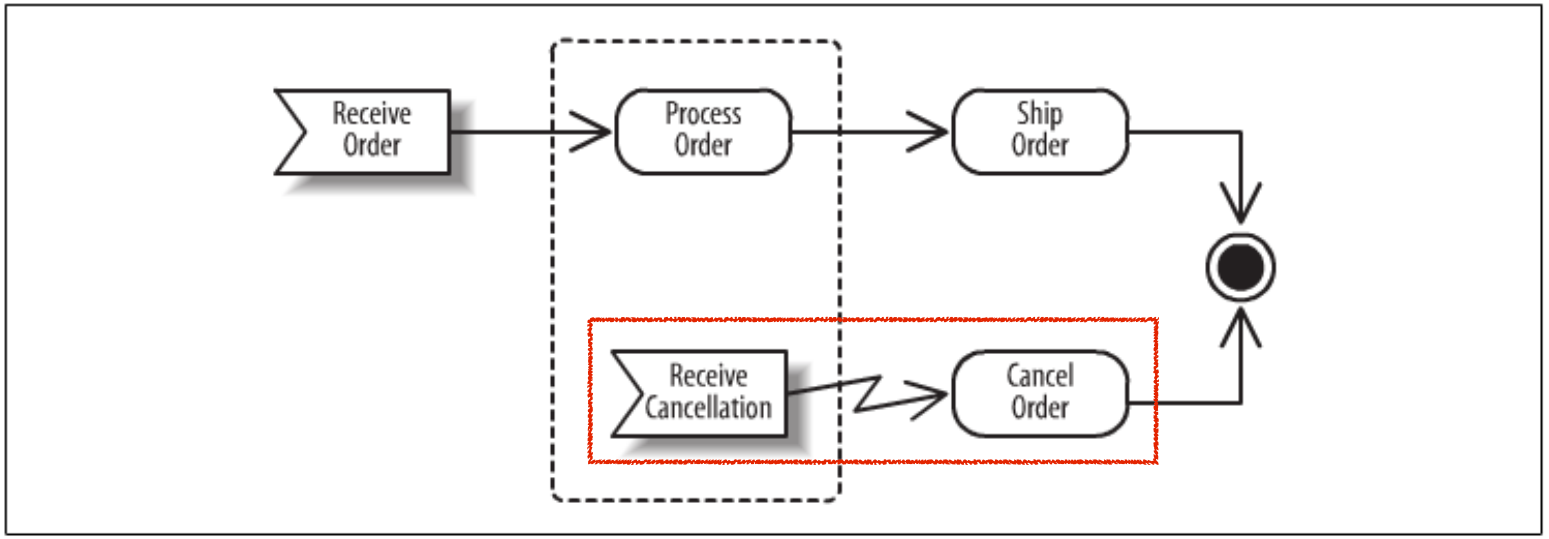


Figure 3-22. Interruption region showing a process that can be interrupted

Activity Diagram

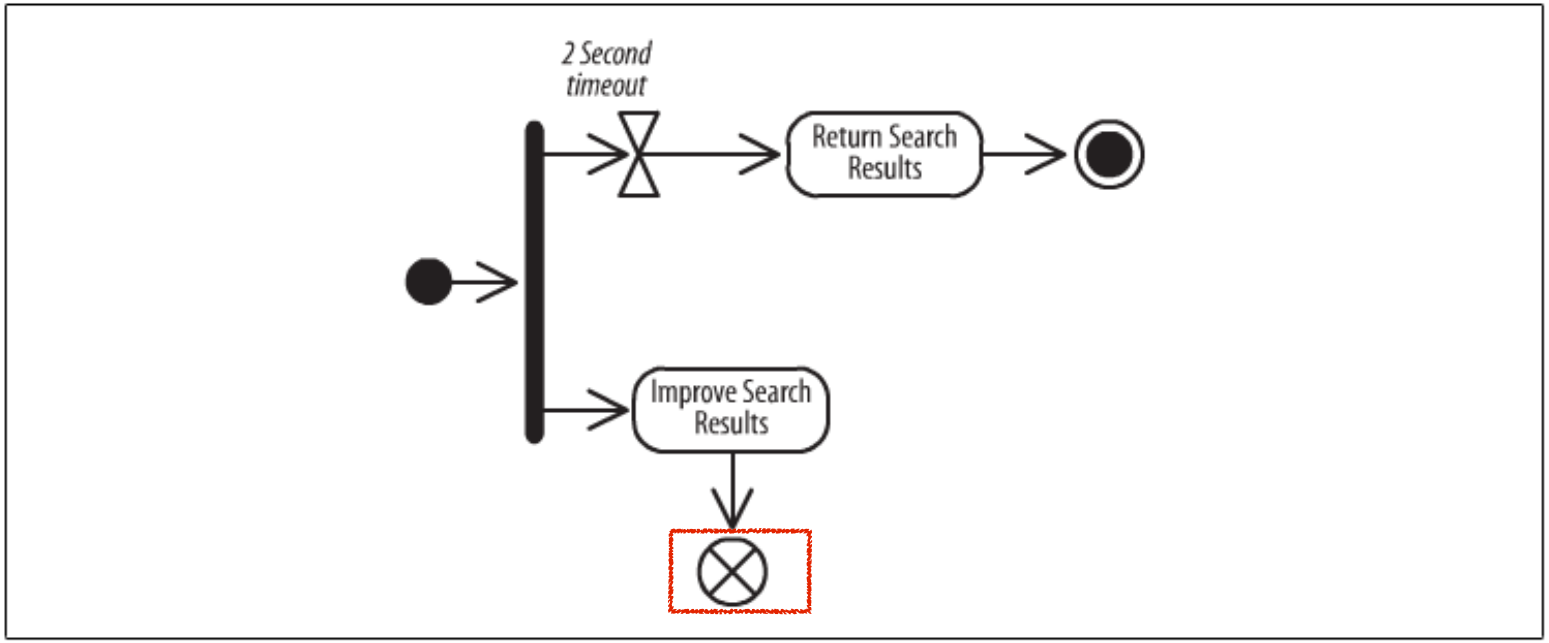


Figure 3-23. A flow final node terminates only its own path—not the whole activity

A new feature of UML 2.0 is the ability to show that a flow dies without ending the whole activity. A *flow final* node terminates its own path—not the whole activity. It is shown as a circle with an X through it, as in Figure 3-23.

Activity Diagram - Swimlanes

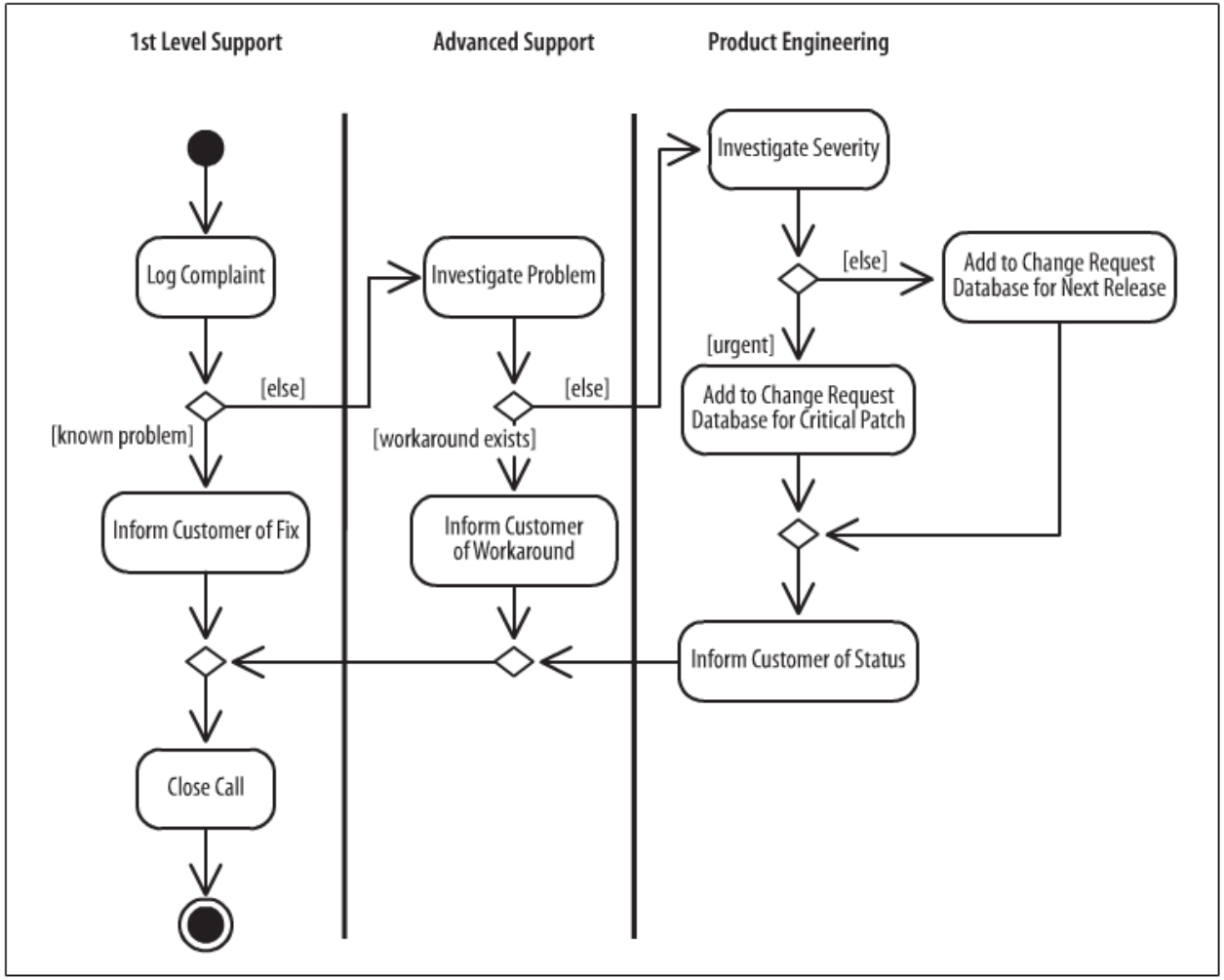


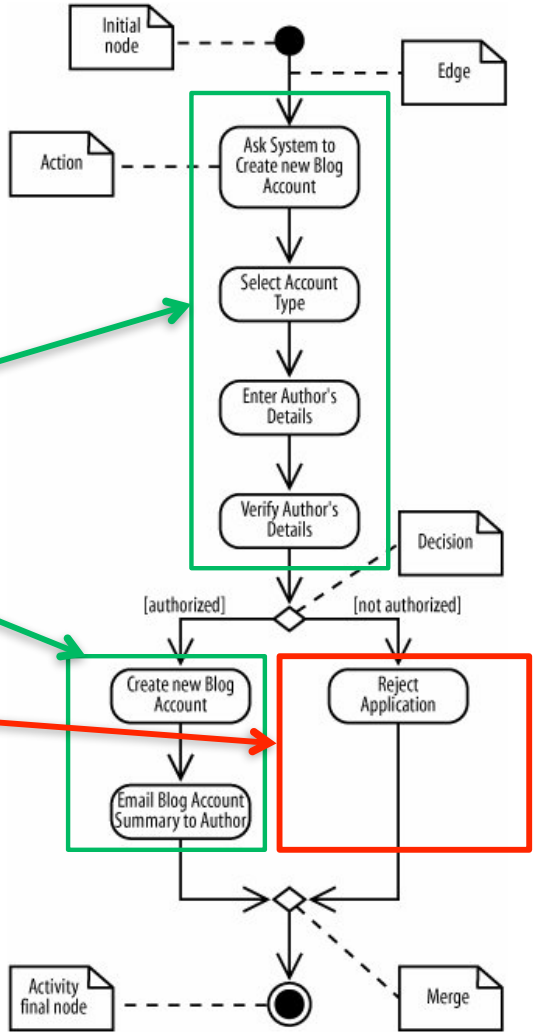
Figure 3-24. Partitions help organize this activity diagram by clarifying responsible parties

You can also show responsibility by using *annotations*. Notice that there are no swimlanes; instead, the name of the responsible party is put in parentheses in the node, shown in Figure 3-25. This notation typically makes your diagram more compact, but it shows the participants less clearly than swimlanes.

Relationship with Use Cases

Modeling Use Cases with Activity Diagrams

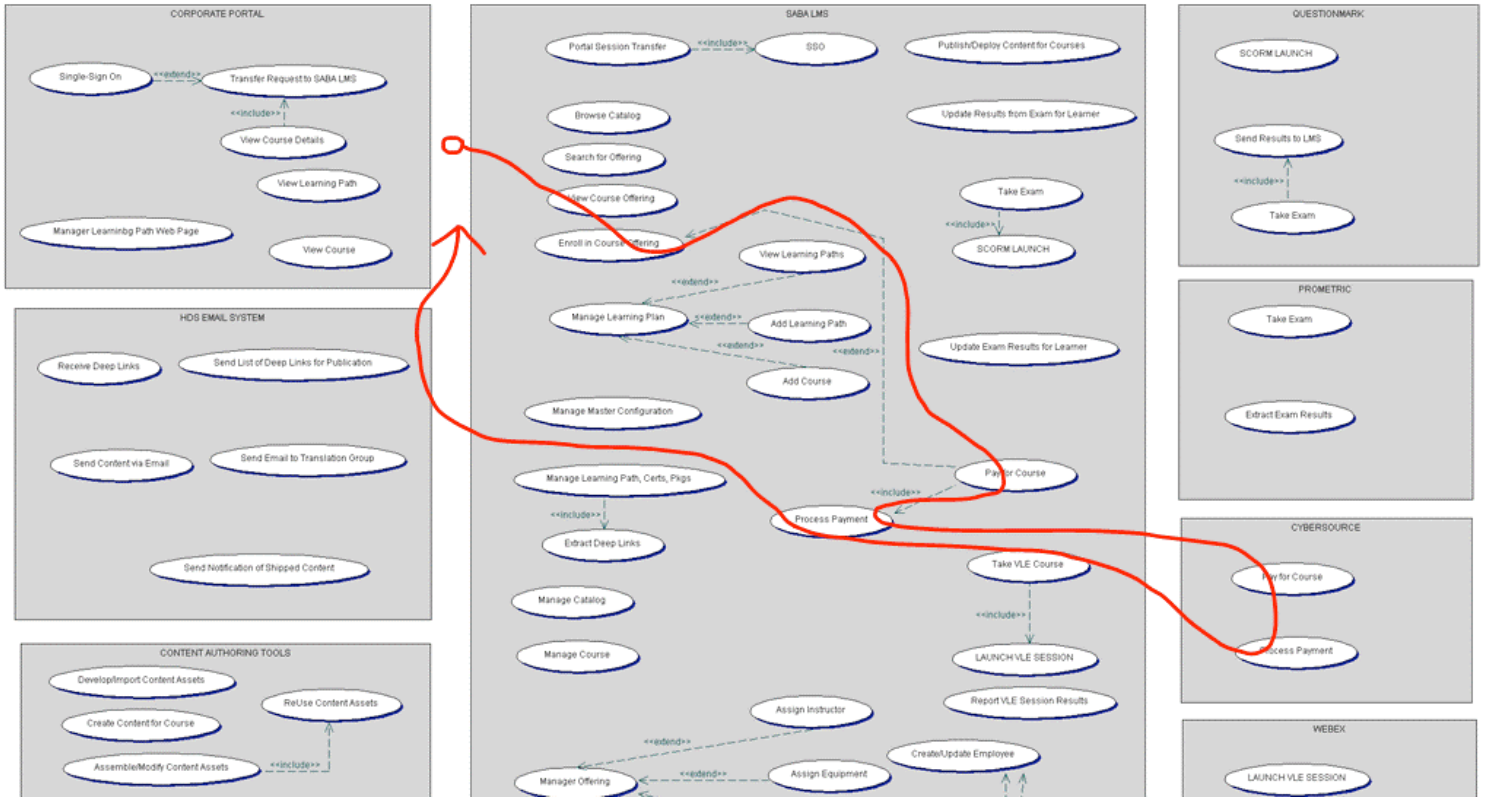
Use case name	Create a new Blog Account	
Related Requirements	Requirement A.1.	
Goal In Context	A new or existing author requests a new blog account from the Administrator.	
Preconditions	The system is limited to recognized authors, and so the author needs to have appropriate proof of identity.	
Successful End Condition	A new blog account is created for the author.	
Failed End Condition	The application for a new blog account is rejected.	
Primary Actors	Administrator.	
Secondary Actors	Author Credentials Database.	
Trigger	The Administrator asks the Content Management System to create a new blog account.	
Main Flow	Step	Action
	1	The Administrator asks the system to create a new blog account.
	2	The Administrator selects an account type.
	3	The Administrator enters the author's details.
	4	The author's details are verified using the Author Credentials Database.
	5	The new blog account is created.
	6	A summary of the new blog account's details are emailed to the author.
Extensions	Step	Branching Action
	4.1	The Author Credentials Database does not verify the author's details.
	4.2	The author's new blog account application is rejected.



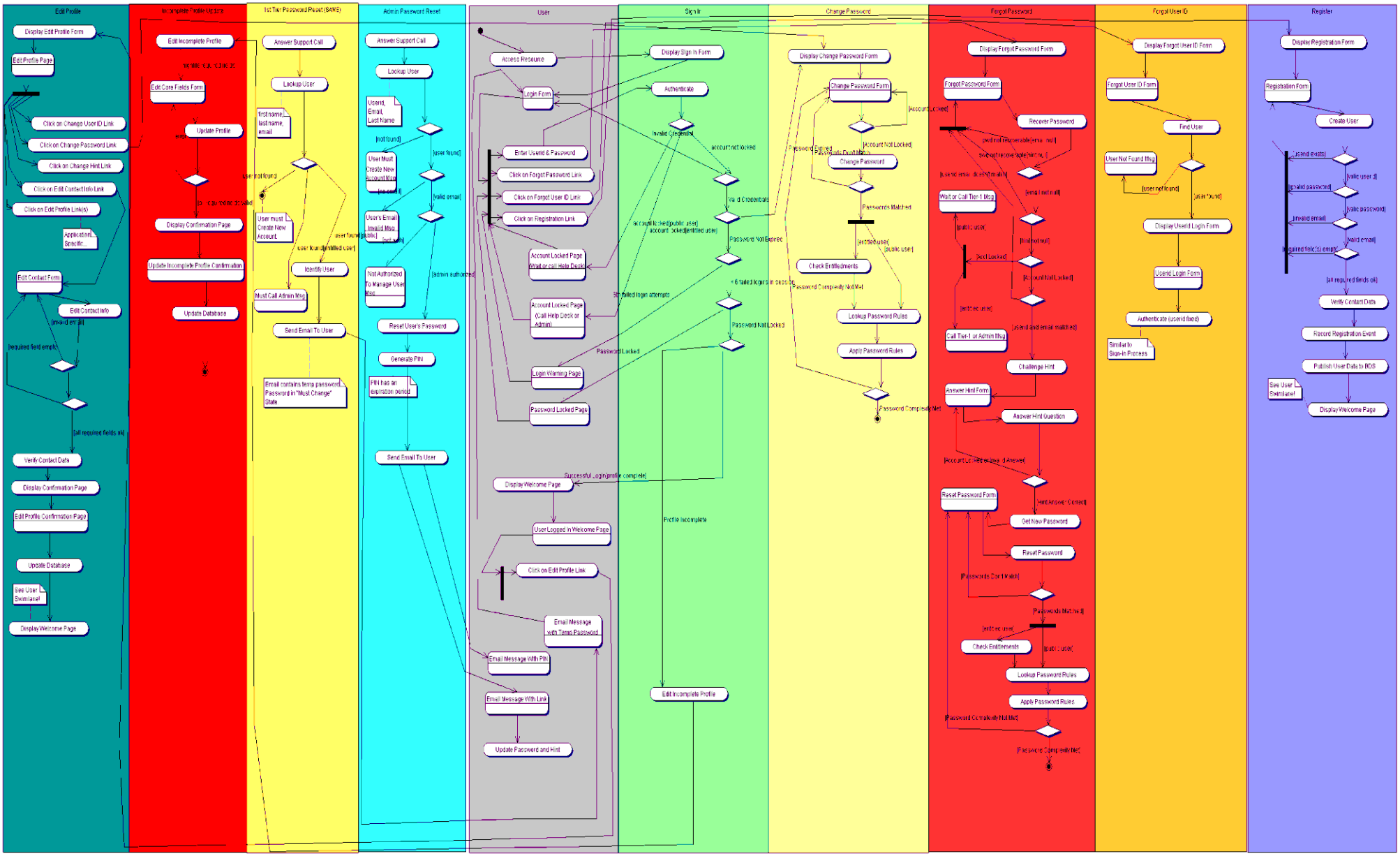
Learning UML 2.0, Hamilton & Miles, O'Reilly 2006

Linking Use Cases in a Business

Use Case Name	Primary Actor	System	Brief Description
View Course Offering	Learner	Saba LMS	View Course Offering Details
Enroll in Course Offering	Learner	Saba LMS	Enrolling in a Course Offering (ILT, VLE, etc...)
Pay for Course	Learner	Saba LMS	Extends Enroll in Course Offering when payment (different types) is required
Process Payment	Saba LMS System	Saba LMS	Included in Pay for Course as transfer point to Cybersource
Pay for Course	Learner	CyberSource	Learner Enters Credentials, Credit Card, Etc... to reserve and confirm enrollment
Process Payment	Saba LMS System	CyberSource	Saba Receives Payment Processing Results from Payment Session Transfer Extends Process Payment when confirmation of payment successful from CyberSource
Send Enrollment Email Notification	Saba LMS System	HDS Email Server	
Receive Confirmation of Enrollment Email Message	Learner	Email Client	Learner checks email client and reads the enrollment confirmation email message

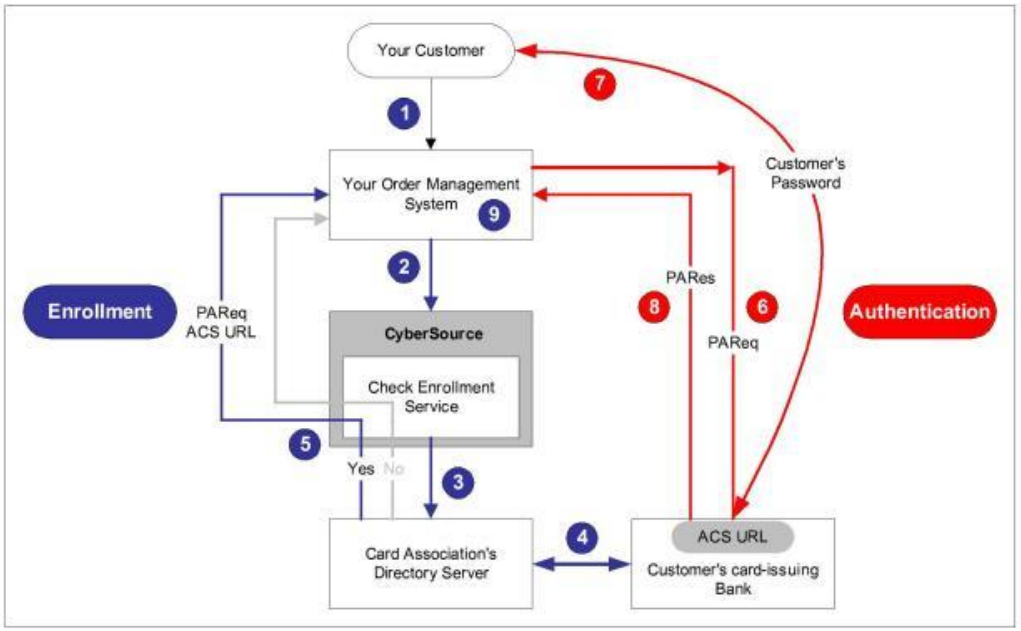


Activity Diagrams with Multiple Use Cases

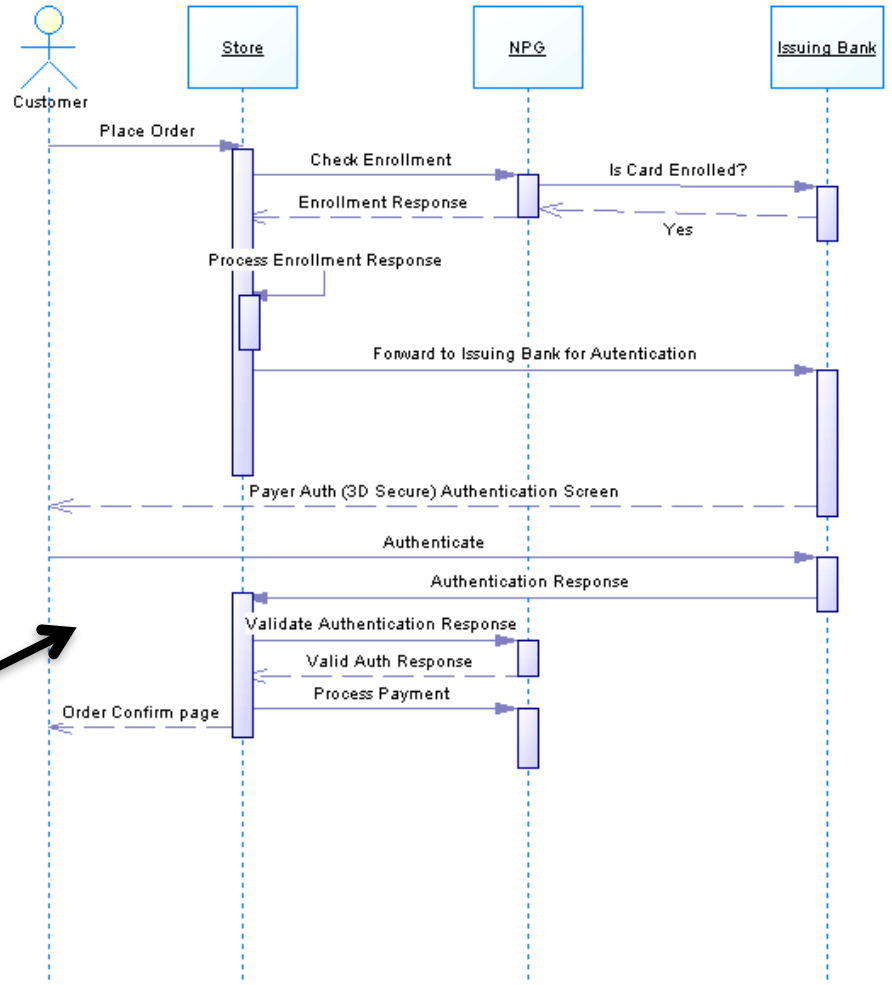


Activity Diagrams with “sequences” super-imposed

Activity Diagram with “sequence” labels



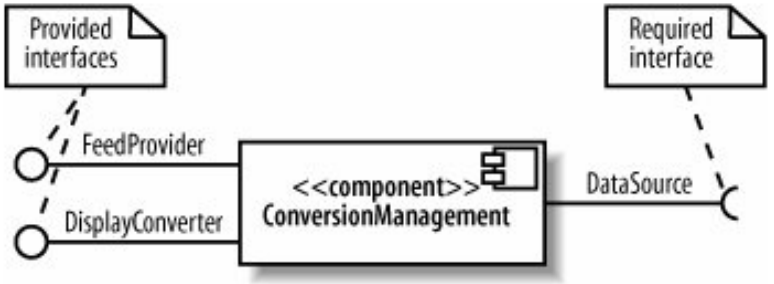
Sequence Diagram Alternative

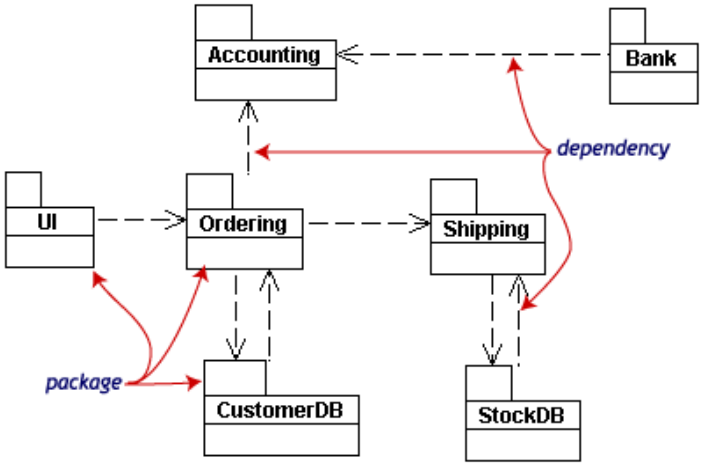


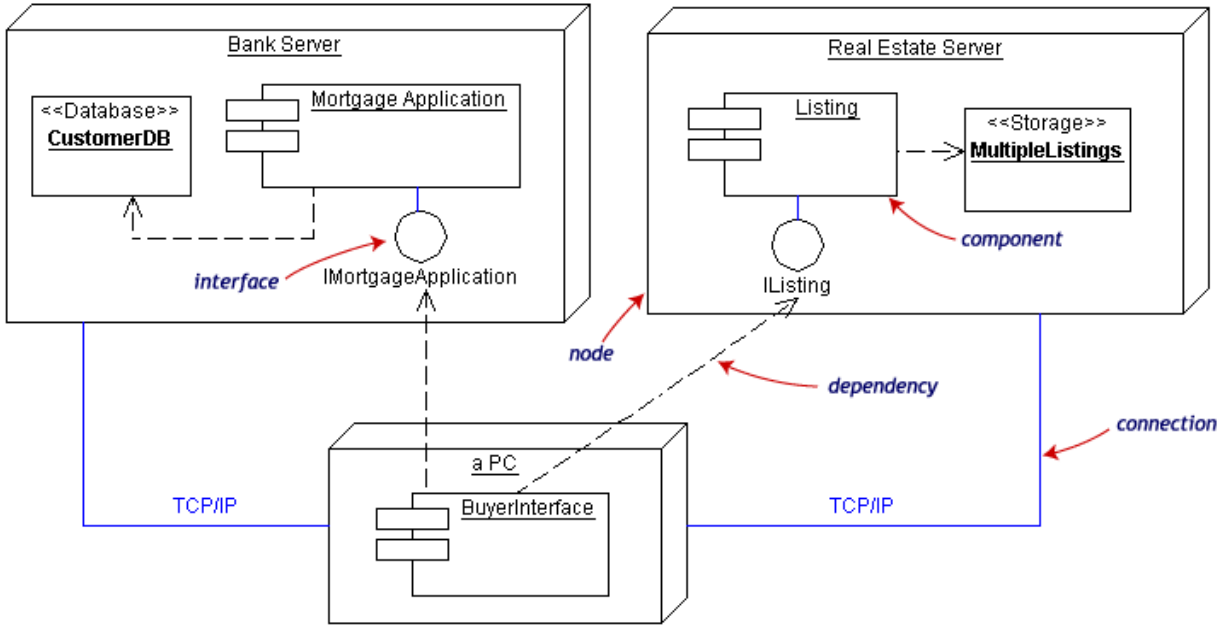
Probably better to use a Sequence Diagram!

Thing #9

The Package, Component & Deployment Diagrams



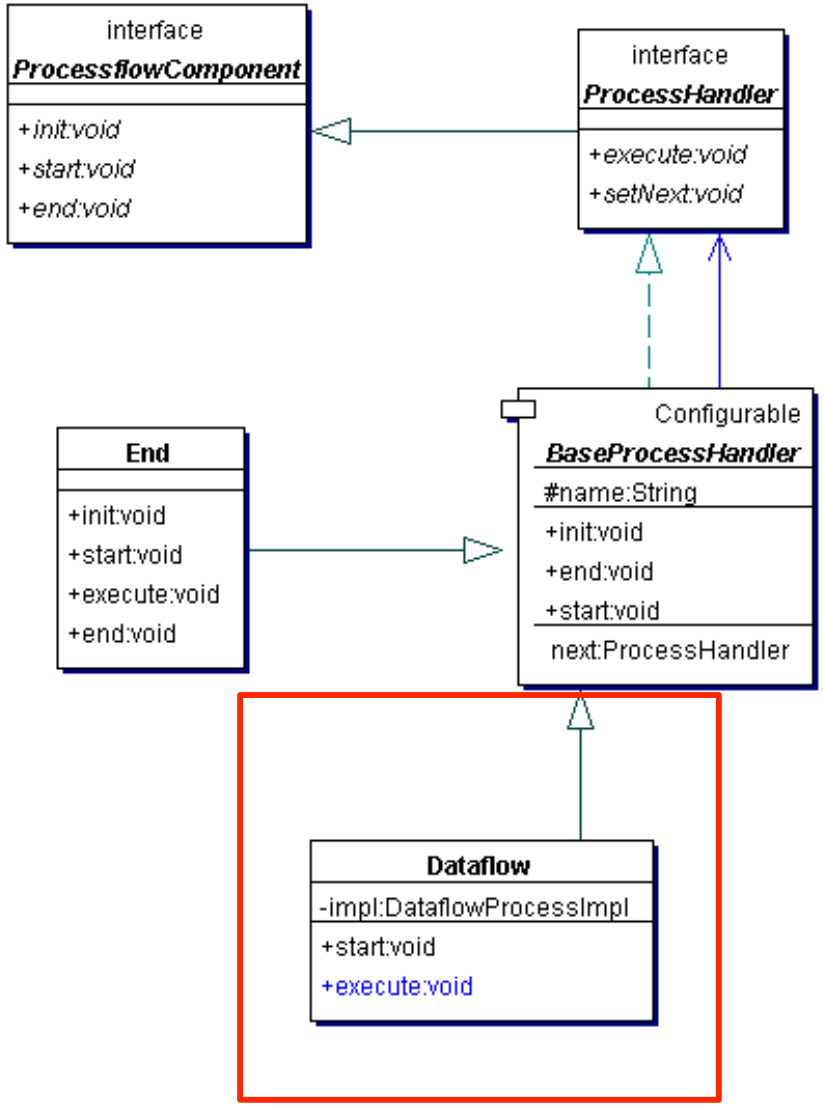




http://dn.codegear.com/article/31863

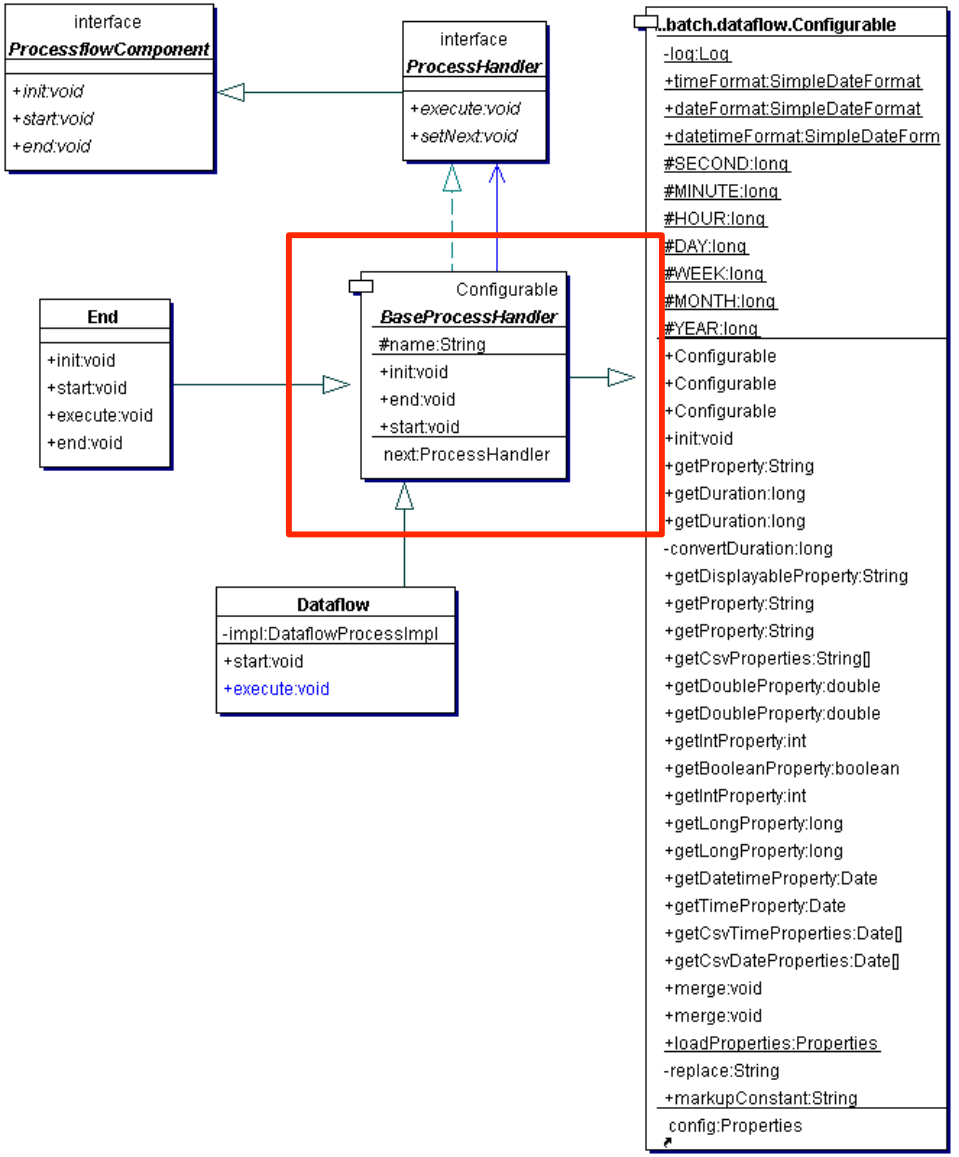
Code-to-Model Relationship

```
package com.cybersource.commons.batch.processflow;

import java.util.Properties;

import com.cybersource.commons.batch.dataflow.DataflowProcessImpl;
import com.cybersource.commons.batch.exception.BatchException;


public class Dataflow extends BaseProcessHandler {
   private DataflowProcessImpl impl;

   public void start(  ) throws BatchException {
      Properties submitConfig = loadProperties(
            getProperty( name + "property" ),
            getConfig(  )
          );
      impl = new DataflowProcessImpl( null, getConfig(  ) );
   }

   public void execute(  ) throws BatchException {
      impl.run(  );
      next.execute(  );
   }
}
```



Code-to-Model Relationship Example (cont.)

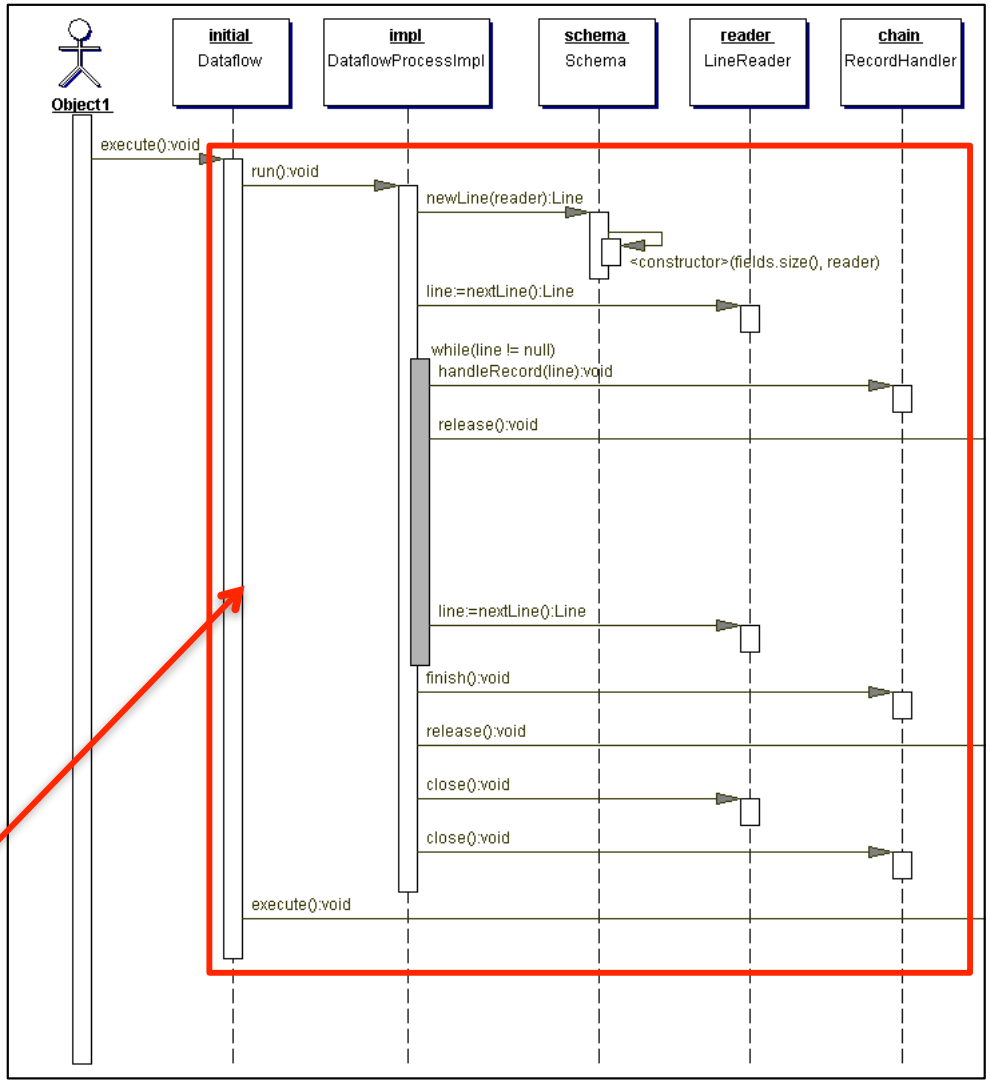
```
package com.cybersource.commons.batch.processflow;

import java.util.Properties;
import com.cybersource.commons.batch.dataflow.Configurable;
import com.cybersource.commons.batch.exception.BatchException;

public abstract class BaseProcessHandler extends Configurable implements ProcessHandler {

    protected String name;
    protected ProcessHandler next = new End();

    public void init( String name, Properties config ) throws BatchException {
        super.init( config );
        this.name = name;
    }

    public void setNext(ProcessHandler handler) {
        this.next = handler;
    }

    public void end() {
        next.end();
    }

    public void start() throws BatchException {
        next.start();
    }
}
```



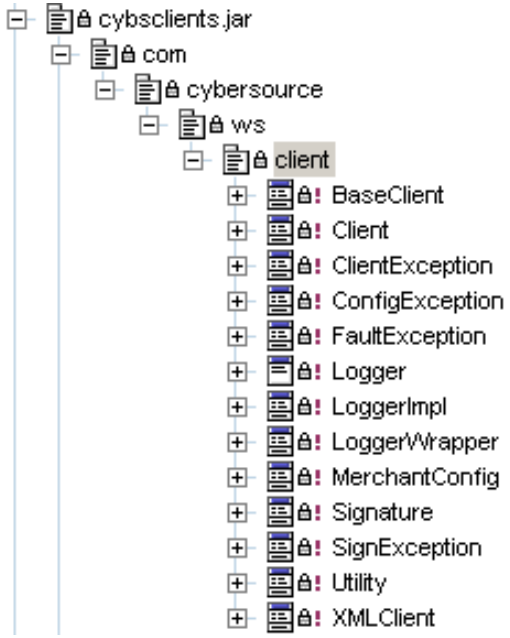
Code-to-Model Relationship Example (cont.)

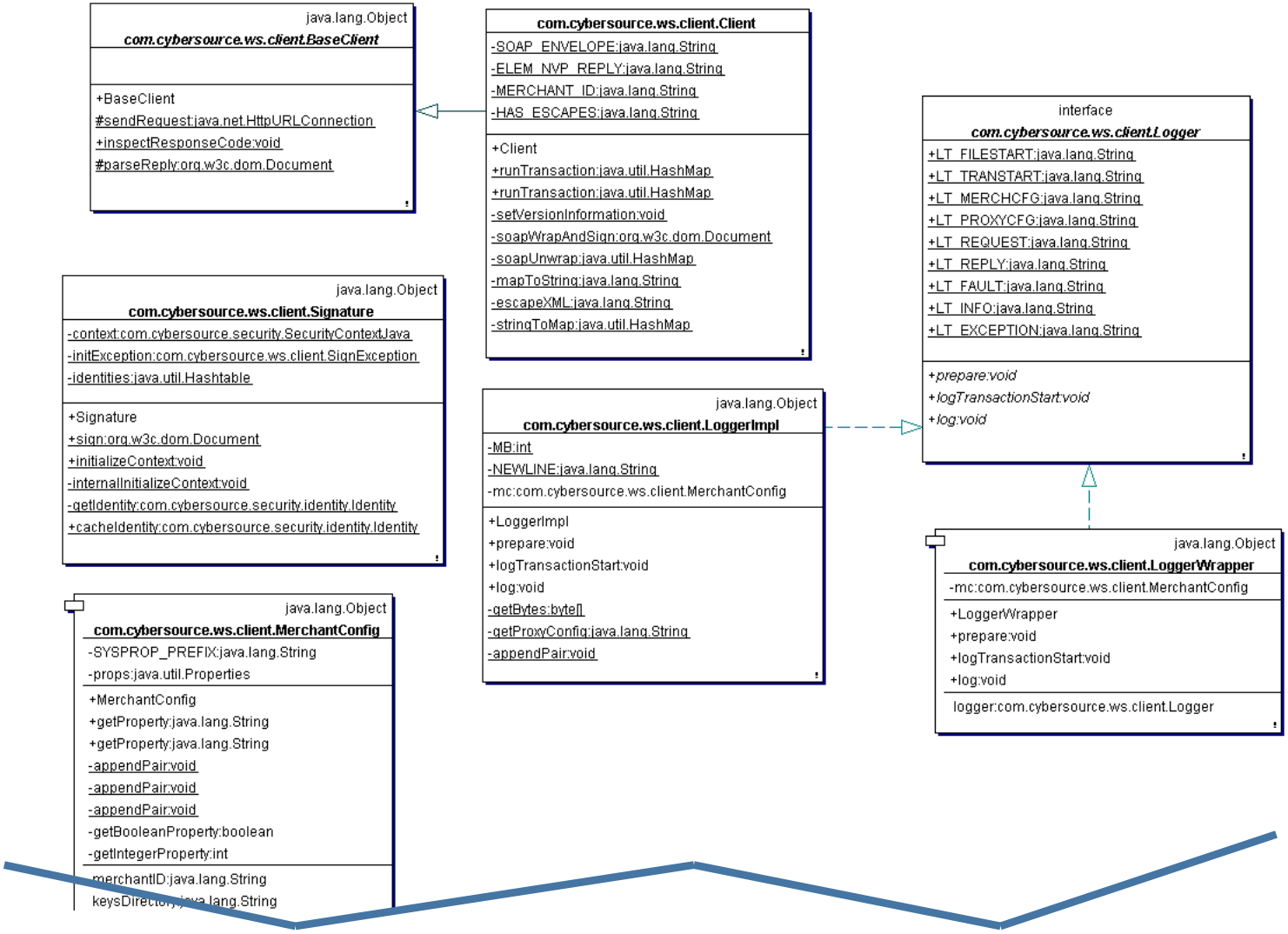
```
package com.cybersource.commons.batch.processflow;

import java.util.Properties;

import com.cybersource.commons.batch.dataflow.DataflowProcessImpl;
import com.cybersource.commons.batch.exception.BatchException;


public class Dataflow extends BaseProcessHandler {
   private DataflowProcessImpl impl;

   public void start(  ) throws BatchException {
      Properties submitConfig = loadProperties(
            getProperty( name + "property" ),
            getConfig(  )
          );
      impl = new DataflowProcessImpl( null, getConfig(  ) );
   }

   public void execute(  ) throws BatchException {
      impl.run(  );
      next.execute(  );
   }
}
```

```
public void run() {
   try {
      this.lastLine = schema.newLine(reader);
      Line line = reader.nextLine();
      while(line != null) {
         chain.handleRecord(line);
         lastLine.release();
         lastLine = line;
         line = reader.nextLine();
      }
      chain.finish();
      lastLine.release();
   } catch(Throwable e) {
      errors.add(e);
      e.printStackTrace();
   }
   try {
      reader.close();
      chain.close();
   } catch(Throwable e) {
      errors.add(e);
      e.printStackTrace();
   }
}
```



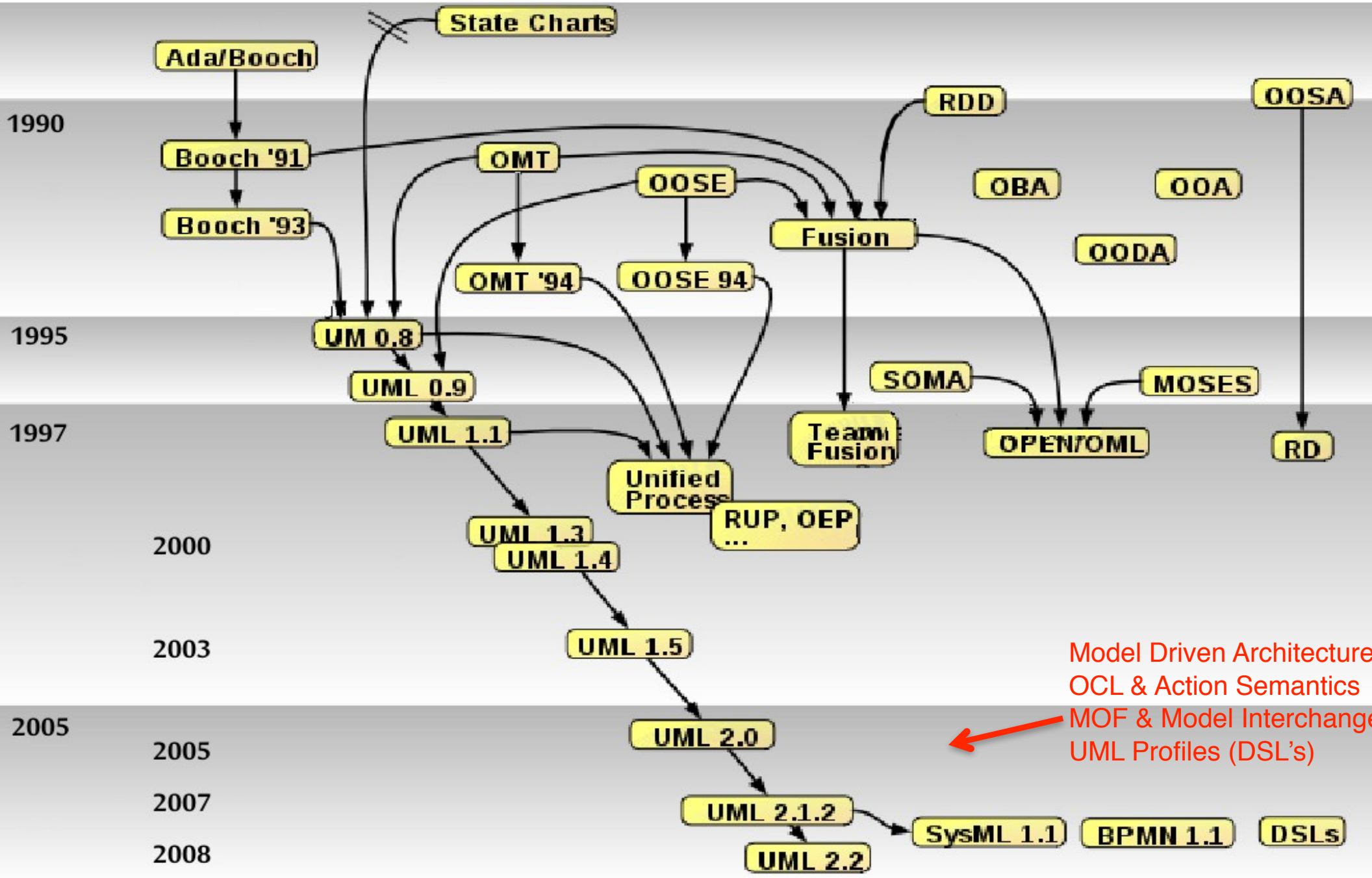
Java JAR “Binaries”-to-Model





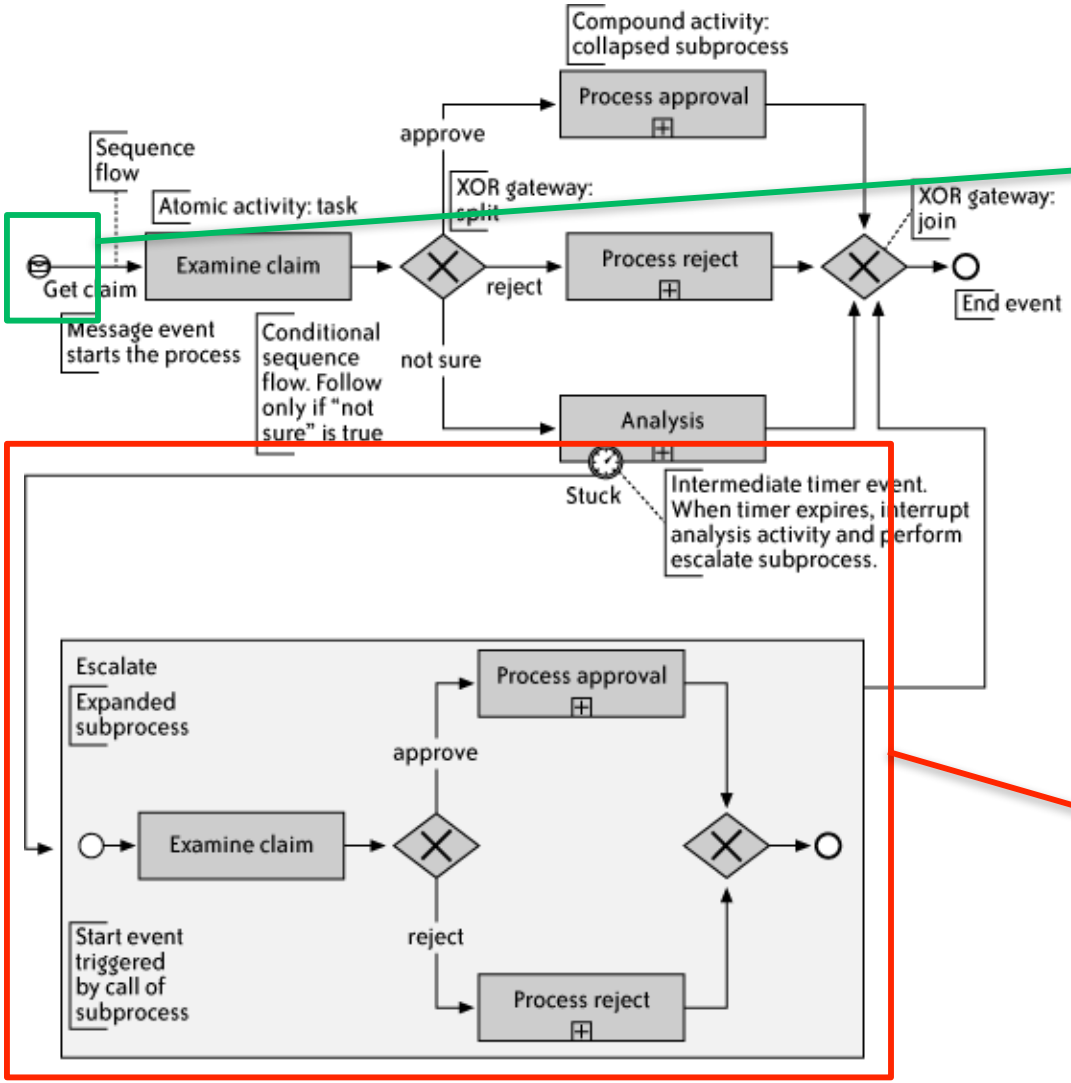
Thing #10

The Future of UML?



http://en.wikipedia.org/wiki/Unified_Modeling_Language

BPMN & BPEL




```
<sequence>

   <!-- We start with a receive activity: get the initiate message. Will
      correlate on claim set defined earlier
   -->
   <receive partnerLink="client" portType="tns:InsuranceClaim"
      operation="initiate" variable="initiateMsg" createInstance="yes"
      name="initiateEvent">
      <correlations>
         <correlation set="claim" initiate="yes"/>
      </correlations>
   </receive>

   <!-- Let an agent evaluate it. Call worklist partner to do this -->
   <invoke name="evalClaim" partnerLink="worklist"  portType="task:TaskManager"
     operation="evalClaim" inputVariable="initiateMsg"/>

   <!-- Get either the response or a timeout -->
   <pick name="analyzePick">
      <onMessage partnerLink="worklist" portType="task:TaskManagerCallback"
         operation="onTaskResult" variable="taskResponse">
         <!-- From response extract status and set to variable 'status' -->
         <assign name="setStatus">
            <copy>
               <from variable="taskResponse" part="payload"
                  query="/tns:taskMessage/tns:result="/>
               <to variable="status"/>
            </copy>
         </assign>
      </onMessage>
      <!-- Timeout! 10 days have passed. Escalate -->
      <onAlarm for="PT10D">
         <sequence>
            <!-- Call partner service to escalate -->
            <invoke name="evalClaim" partnerLink="worklist"
               portType="task:TaskManager" operation="escalateClaim"
               inputVariable="initiateMsg"/>
            <!-- Get the escalation response -->
            <receive name="receiveTaskResult" partnerLink="worklist"
               portType="task:TaskManagerCallback"
               operation="onTaskResult" variable="taskResponse"/>
          <!-- From response extract status and set to variable 'status' -->
            <assign name="setStatus">
               <copy>
                  <from variable="taskResponse" part="payload"
                     query="/tns:taskMessage/tns:result="/>
                  <to variable="status"/>
               </copy>
            </assign>
         </sequence>
      </onAlarm>
   </pick>
```

Essential Business Process Modeling, Havey, O'Reilly 2005

http://www.bpmn.org

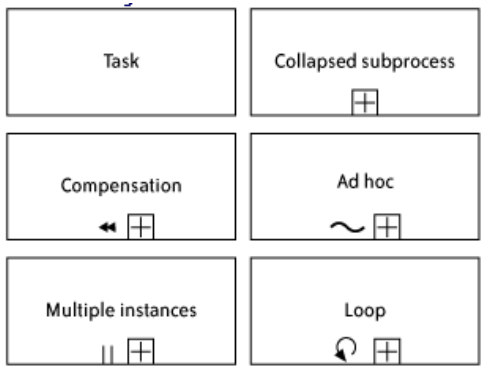
(see also) http://www.itposter.net/itPosters/bpmn/bpmn.htm

BPMN Notation (in brief)

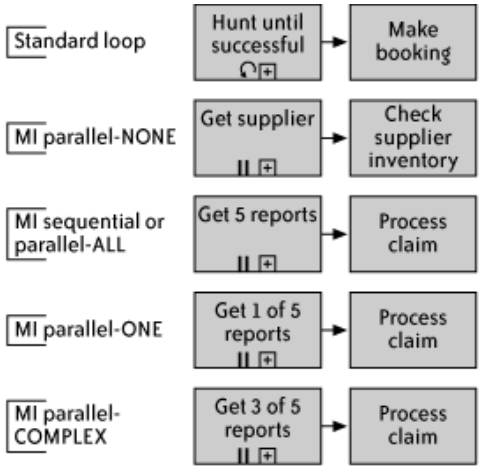
BPMN Events

Start	Intermediate	End	Name
○	◎	●	Basic
✉	✉	✉	Message
◷	◷		Timer
▤	▤		Rule
	Ⓝ	Ⓝ	Exception
	⊗	⊗	Cancellation
	◀◀	◀◀	Compensation
→	→	→	Link
✸	✸	✸	Multiple
		◑	Termination

BPMN Activities



BPMN Loops



BPMN Gateways (i.e. control structures)

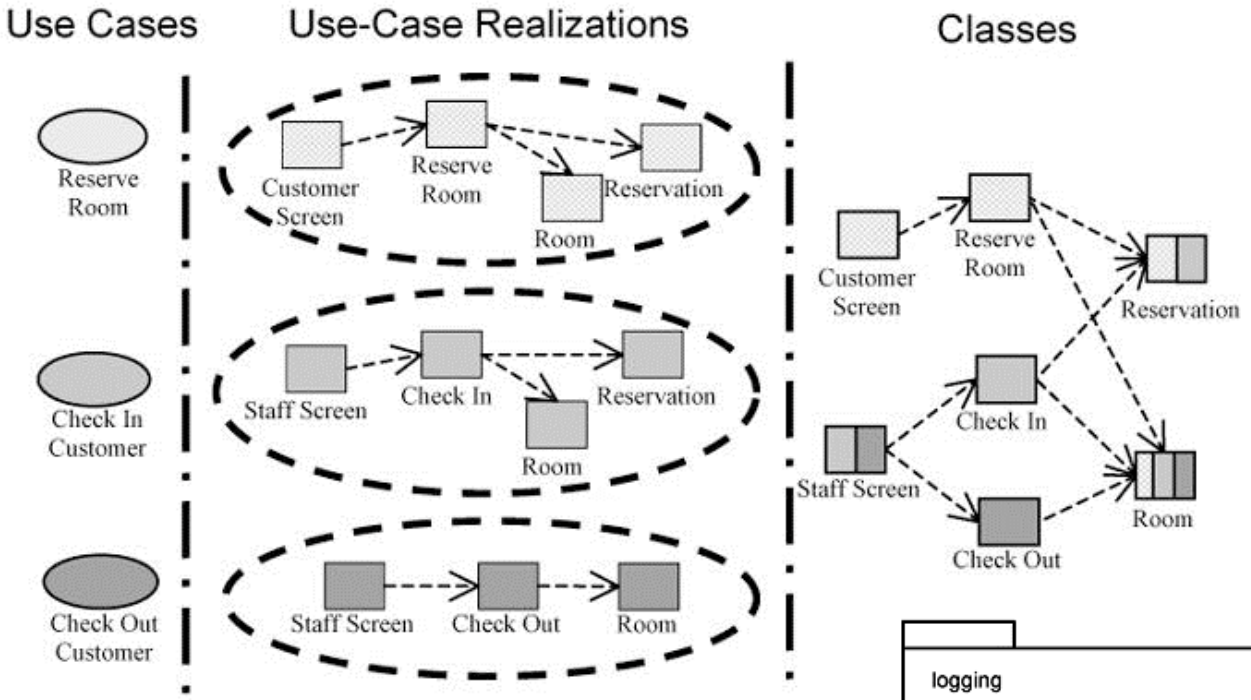
Symbol	Name
◇	Exclusive OR
◇ ✕	Exclusive OR
◇ ✸	Exclusive OR (Event-based)
◇ ○	Exclusive OR
◇ ✳	Complex
◇ +	Parallel

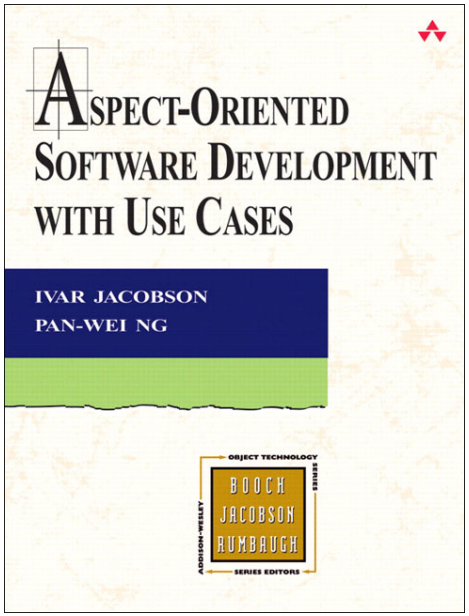
Essential Business Process Modeling, Havey, O'Reilly 2005

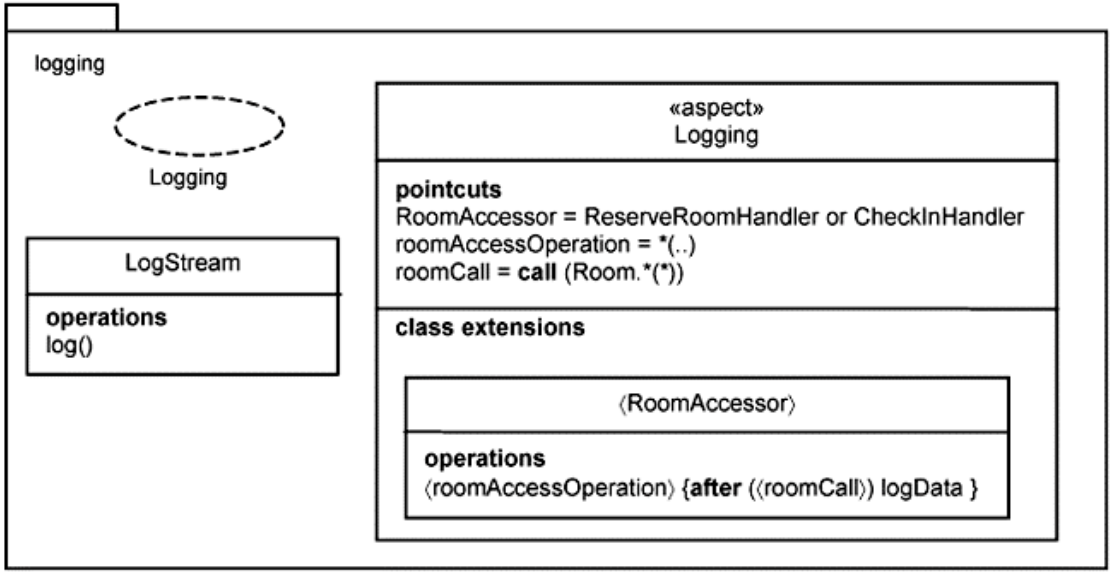
http://www.bpmn.org

(see also) http://www.itposter.net/itPosters/bpmn/bpmn.htm

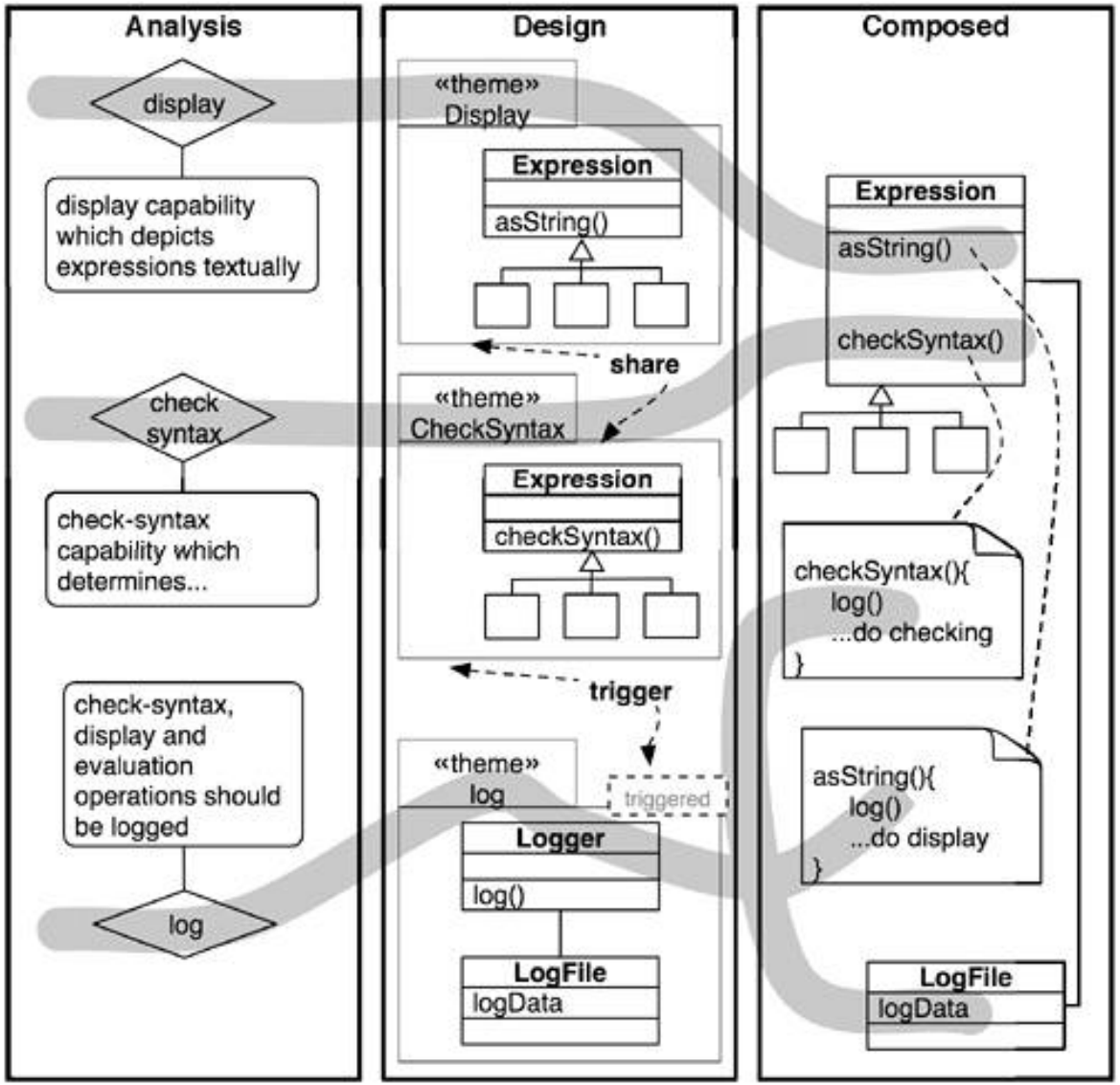
Aspect-Oriented Use Cases

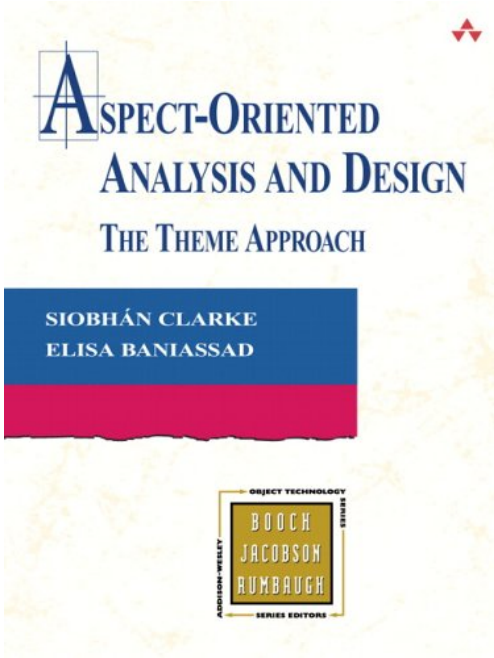






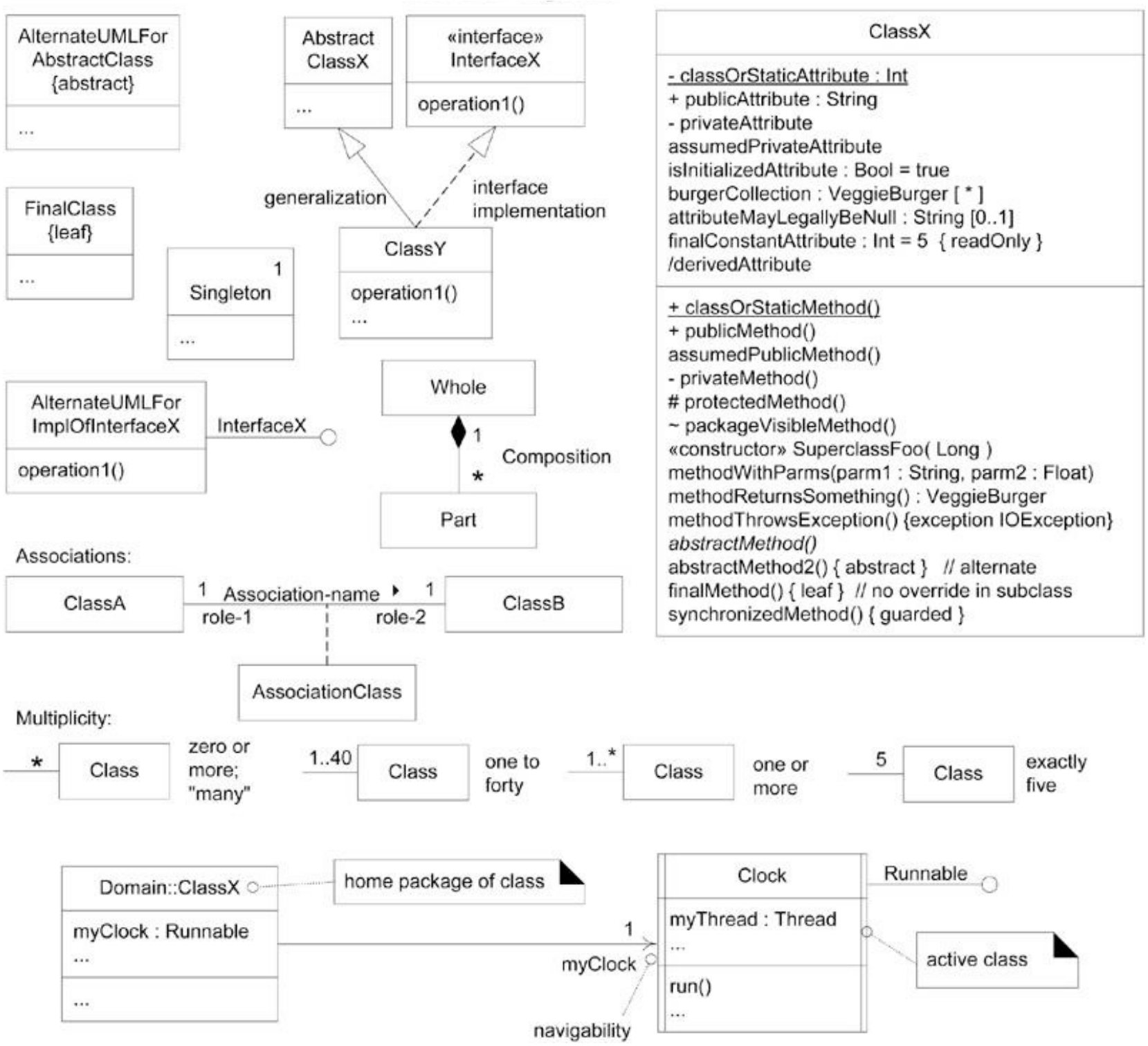
Aspect-Oriented Design (Theme

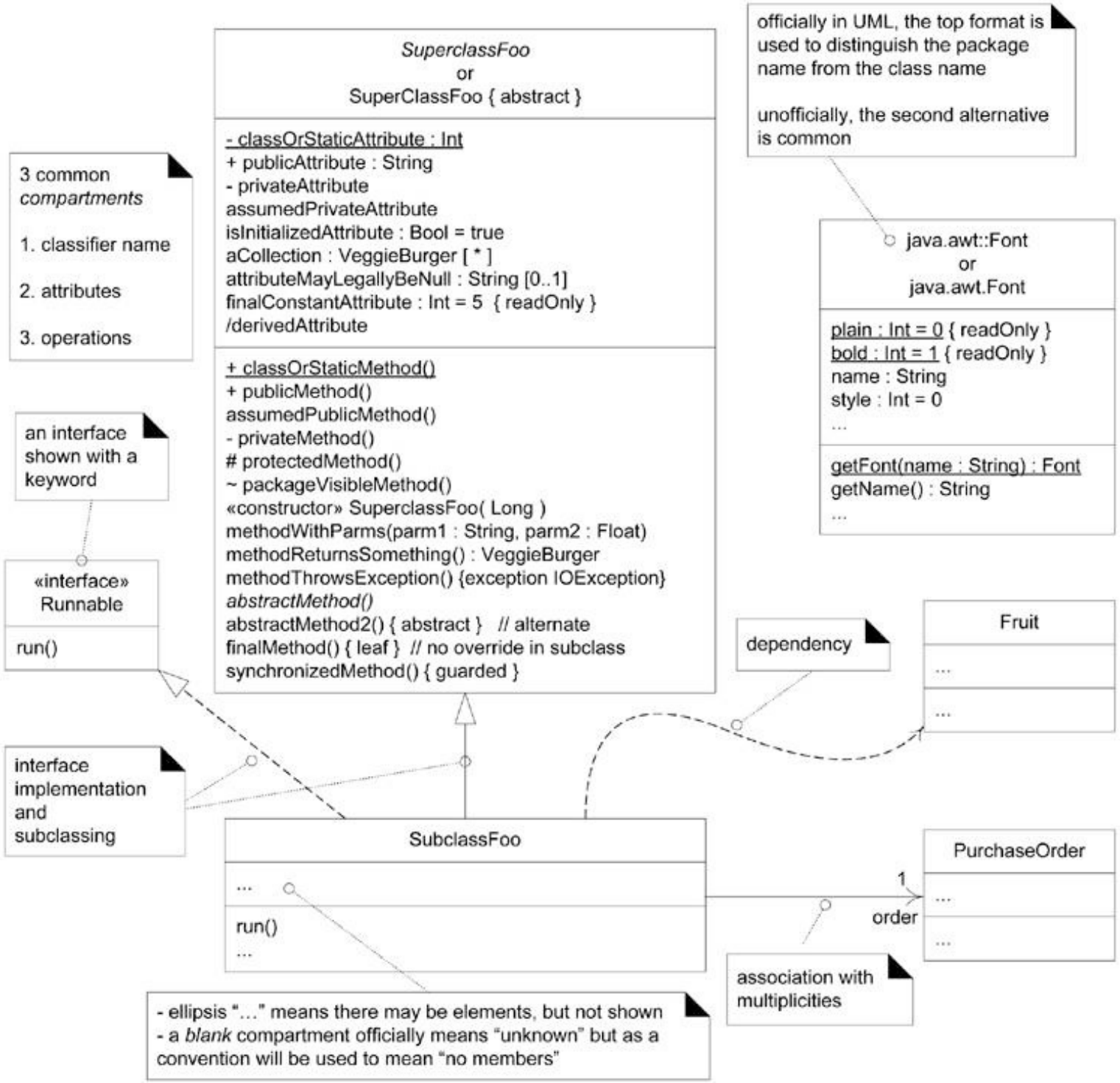




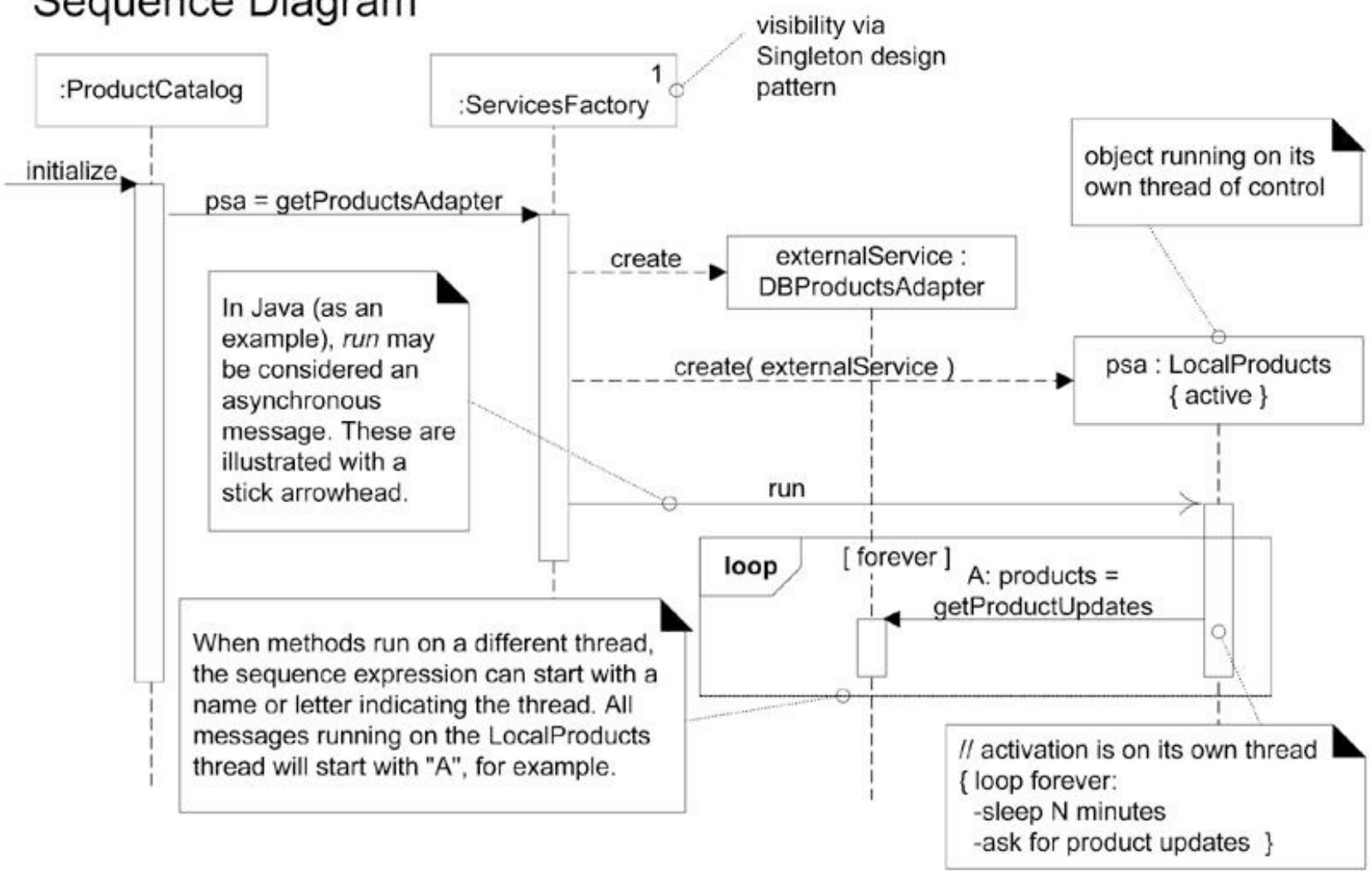
UML Diagrams

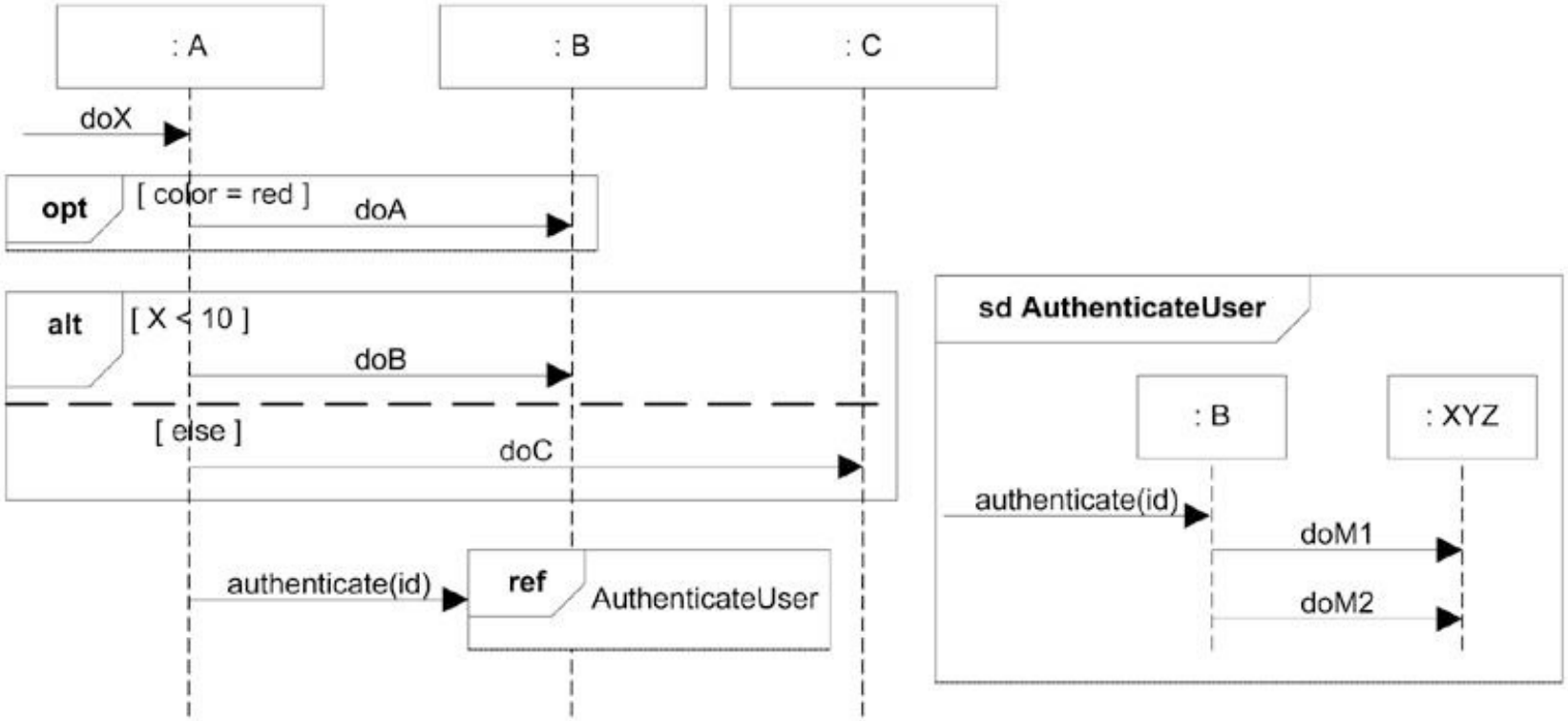
Quick Reference


AlternateUMLFor
AbstractClass
{abstract}
...
Abstract
ClassX
...
«interface»
InterfaceX
operation1()
generalization
interface
implementation
FinalClass
{leaf}
...
1
Singleton
...
ClassY
operation1()
...
ClassX
- classOrStaticAttribute : Int
+ publicAttribute : String
- privateAttribute
assumedPrivateAttribute
isInitializedAttribute : Bool = true
burgerCollection : VeggieBurger [*]
attributeMayLegallyBeNull : String [0..1]
finalConstantAttribute : Int = 5 { readOnly }
/derivedAttribute
+ classOrStaticMethod()
+ publicMethod()
assumedPublicMethod()
- privateMethod()
protectedMethod()
~ packageVisibleMethod()
«constructor» SuperclassFoo(Long)
methodWithParms(parm1 : String, parm2 : Float)
methodReturnsSomething() : VeggieBurger
methodThrowsException() {exception IOException}
abstractMethod()
abstractMethod2() { abstract } // alternate
finalMethod() { leaf } // no override in subclass
synchronizedMethod() { guarded }
AlternateUMLFor
ImplOfInterfaceX
operation1()
InterfaceX
Whole
1
Composition
*
Part
Associations:
ClassA
1
Association-name
1
ClassB
role-1
role-2
AssociationClass
Multiplicity:
*
Class
zero or
more;
"many"
1..40
Class
one to
forty
1..*
Class
one or
more
5
Class
exactly
five
Domain::ClassX
home package of class
myClock : Runnable
...
...
1
myClock
navigability
Clock
myThread : Thread
...
run()
...
Runnable
active class

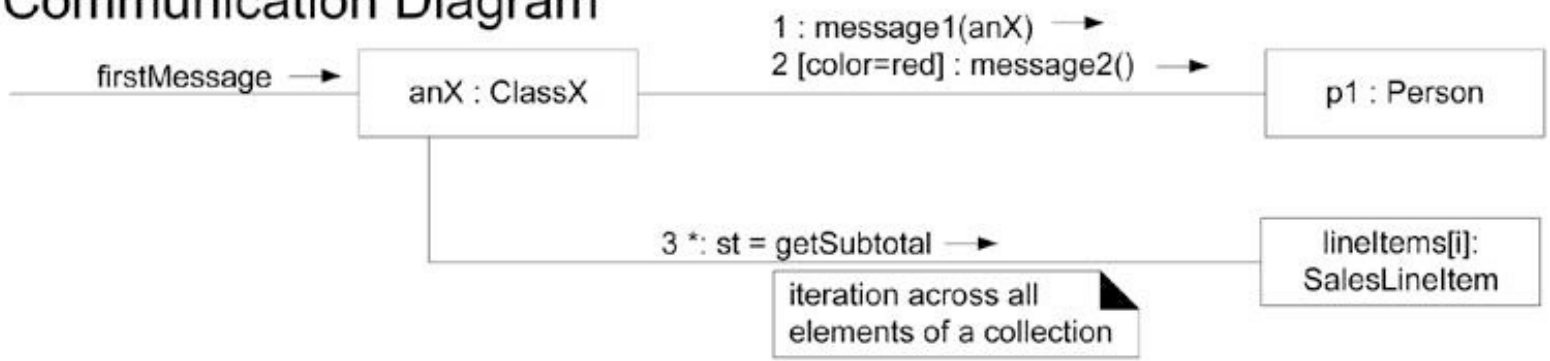

SuperclassFoo
or
SuperClassFoo { abstract }
- classOrStaticAttribute : Int
+ publicAttribute : String
- privateAttribute
assumedPrivateAttribute
isInitializedAttribute : Bool = true
aCollection : VeggieBurger [*]
attributeMayLegallyBeNull : String [0..1]
finalConstantAttribute : Int = 5 { readOnly }
/derivedAttribute
+ classOrStaticMethod()
+ publicMethod()
assumedPublicMethod()
- privateMethod()
protectedMethod()
~ packageVisibleMethod()
«constructor» SuperclassFoo(Long)
methodWithParms(parm1 : String, parm2 : Float)
methodReturnsSomething() : VeggieBurger
methodThrowsException() {exception IOException}
abstractMethod()
abstractMethod2() { abstract } // alternate
finalMethod() { leaf } // no override in subclass
synchronizedMethod() { guarded }
3 common compartments
1. classifier name
2. attributes
3. operations
officially in UML, the top format is used to distinguish the package name from the class name
unofficially, the second alternative is common
java.awt::Font
or
java.awt.Font
plain : Int = 0 { readOnly }
bold : Int = 1 { readOnly }
name : String
style : Int = 0
...
getFont(name : String) : Font
getName() : String
...
an interface shown with a keyword
«interface»
Runnable
run()
dependency
Fruit
...
...
interface implementation and subclassing
SubclassFoo
...
run()
...
PurchaseOrder
...
...
1
order
association with multiplicities
- ellipsis "..." means there may be elements, but not shown
- a blank compartment officially means "unknown" but as a convention will be used to mean "no members"

Sequence Diagram





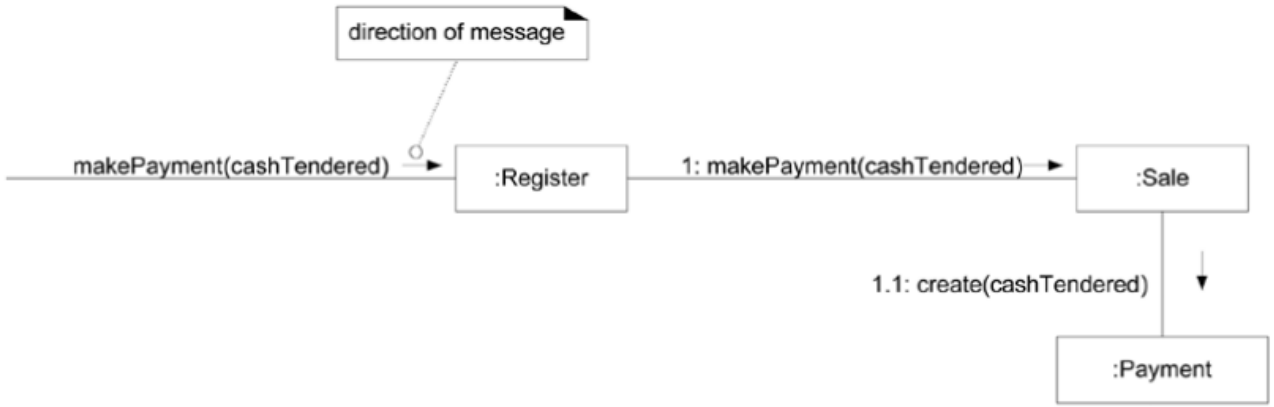
Communication Diagram



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.

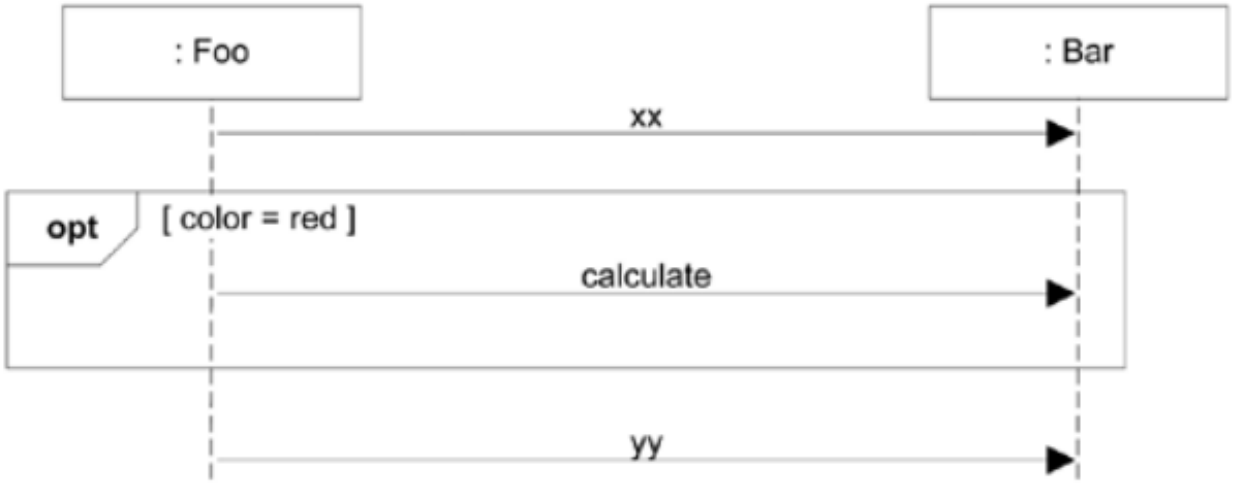


Figure 15.15. Mutually exclusive conditional messages.

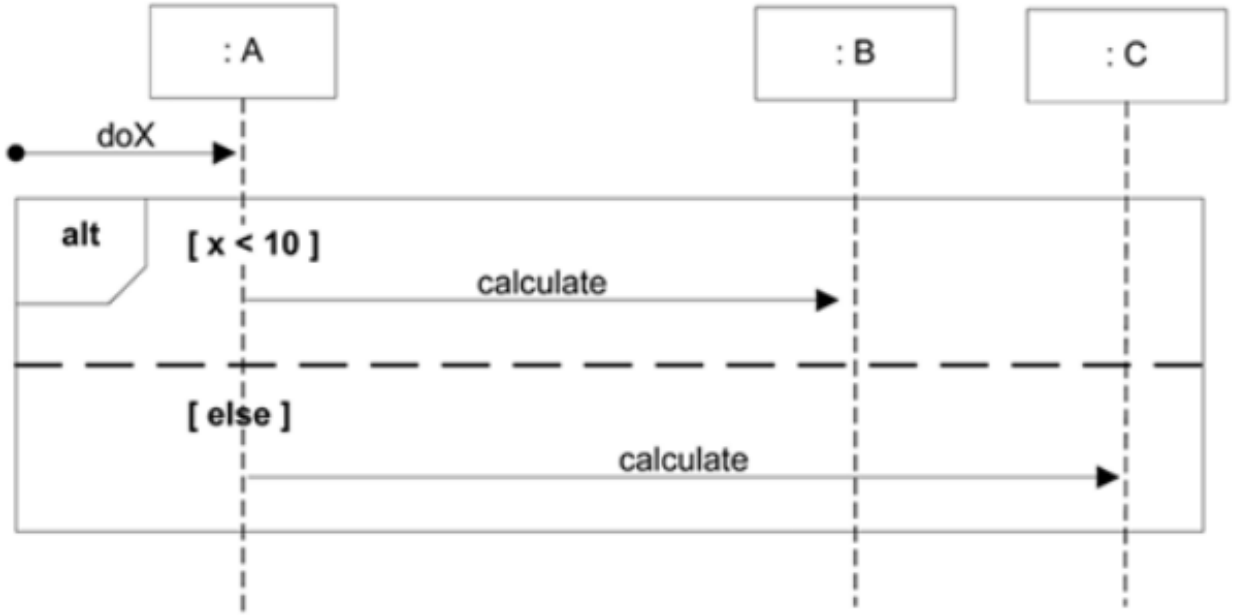


Figure 15.16. Iteration over a collection using relatively explicit notation.

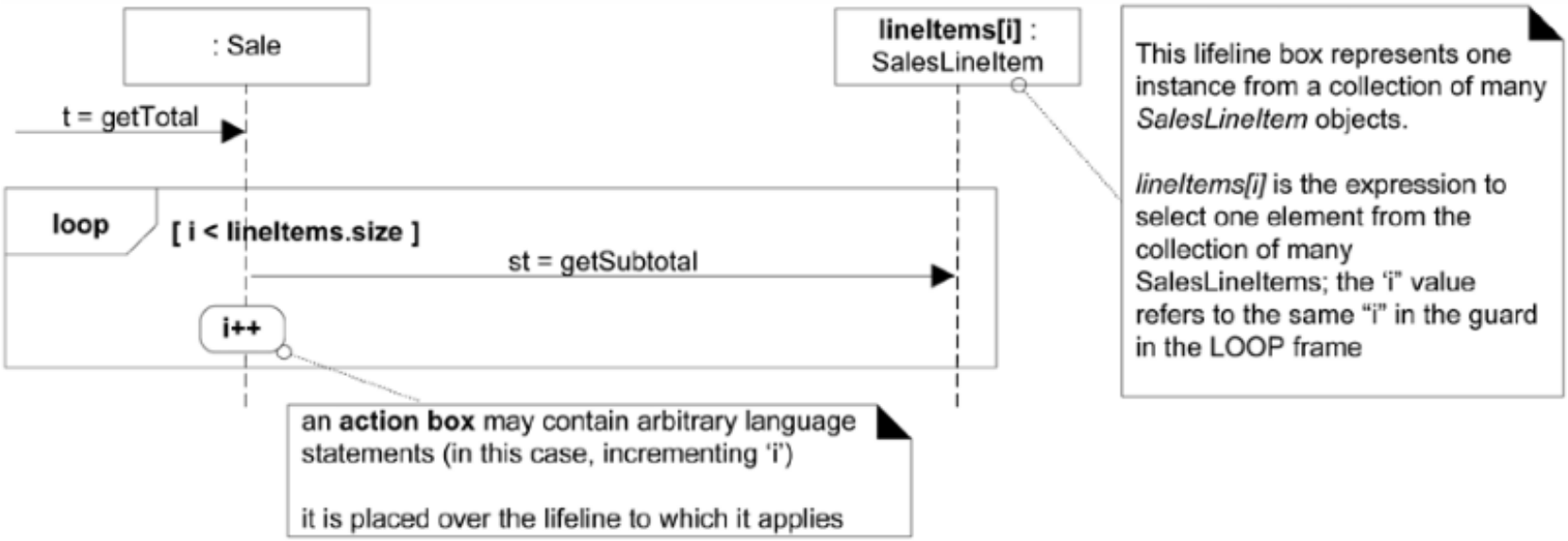


Figure 15.17. Iteration over a collection leaving things more implicit.

Figure 15.18. Nesting of frames.

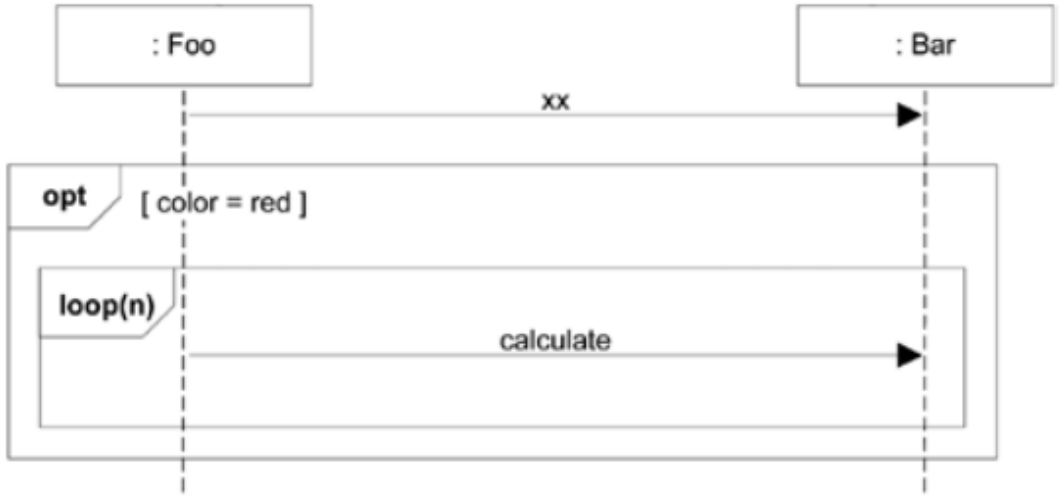


Figure 15.19. Example interaction occurrence, *sd* and *ref* frames.

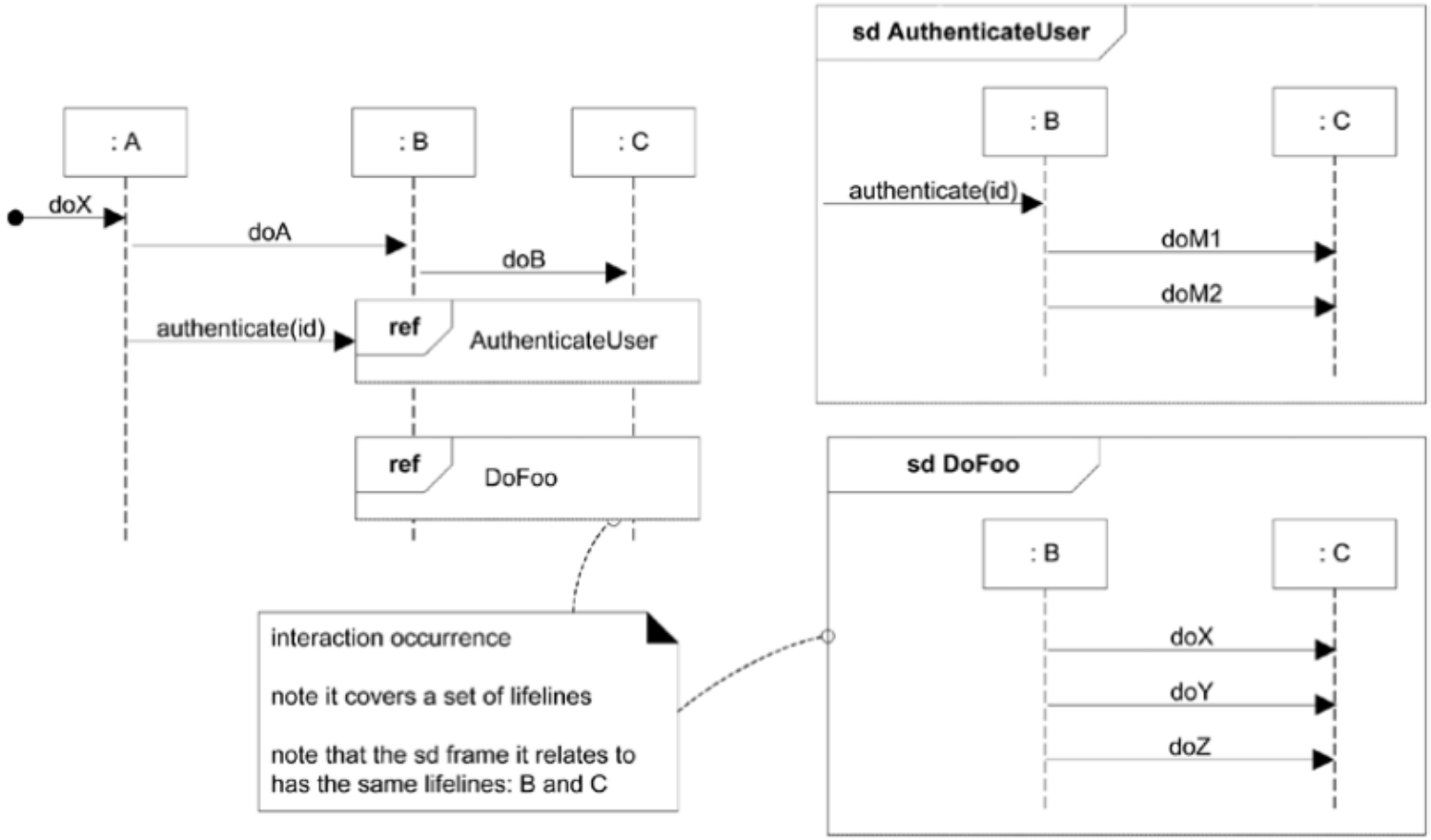


Figure 15.28. Complex sequence numbering.

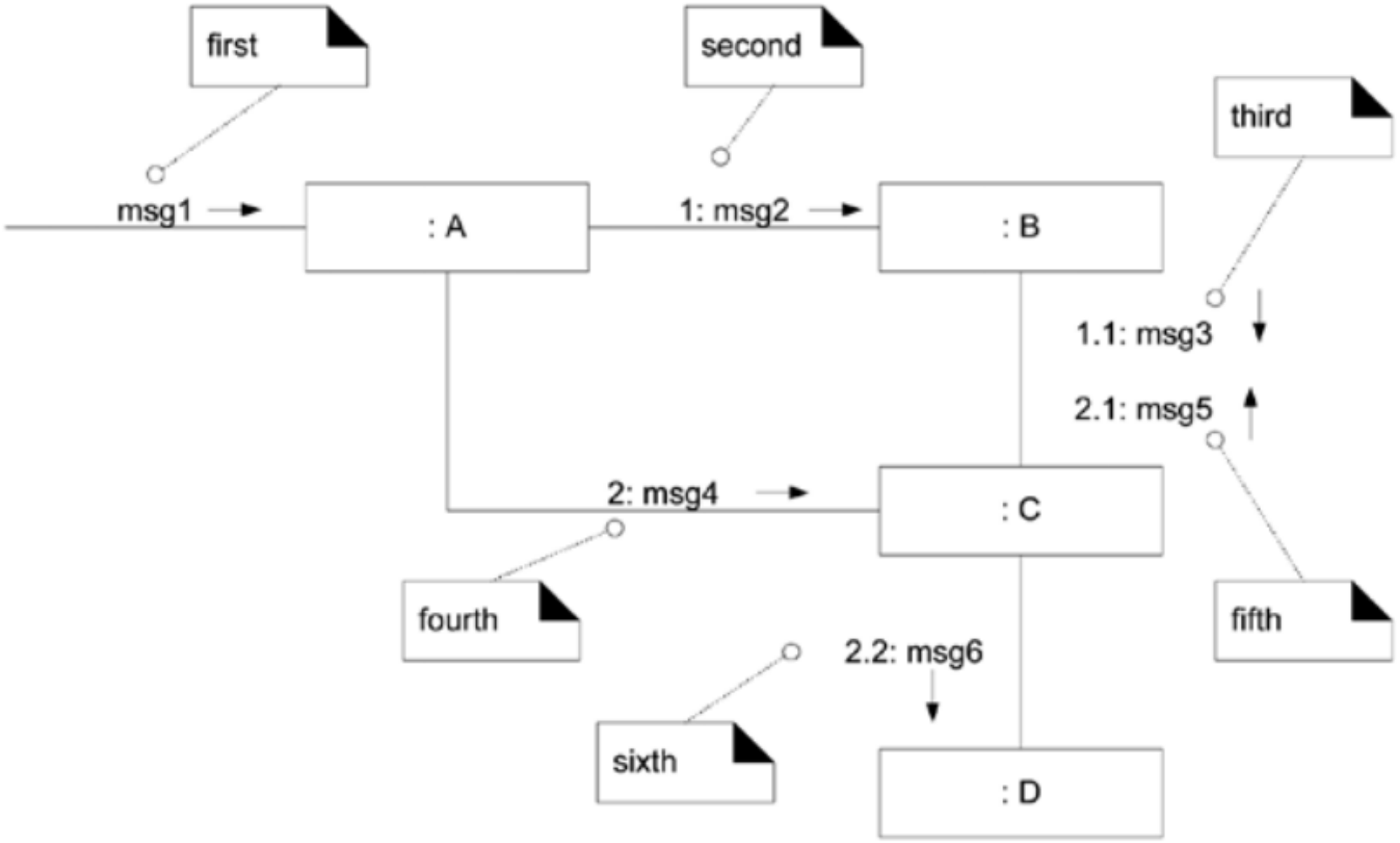


Figure 15.30. Mutually exclusive messages.

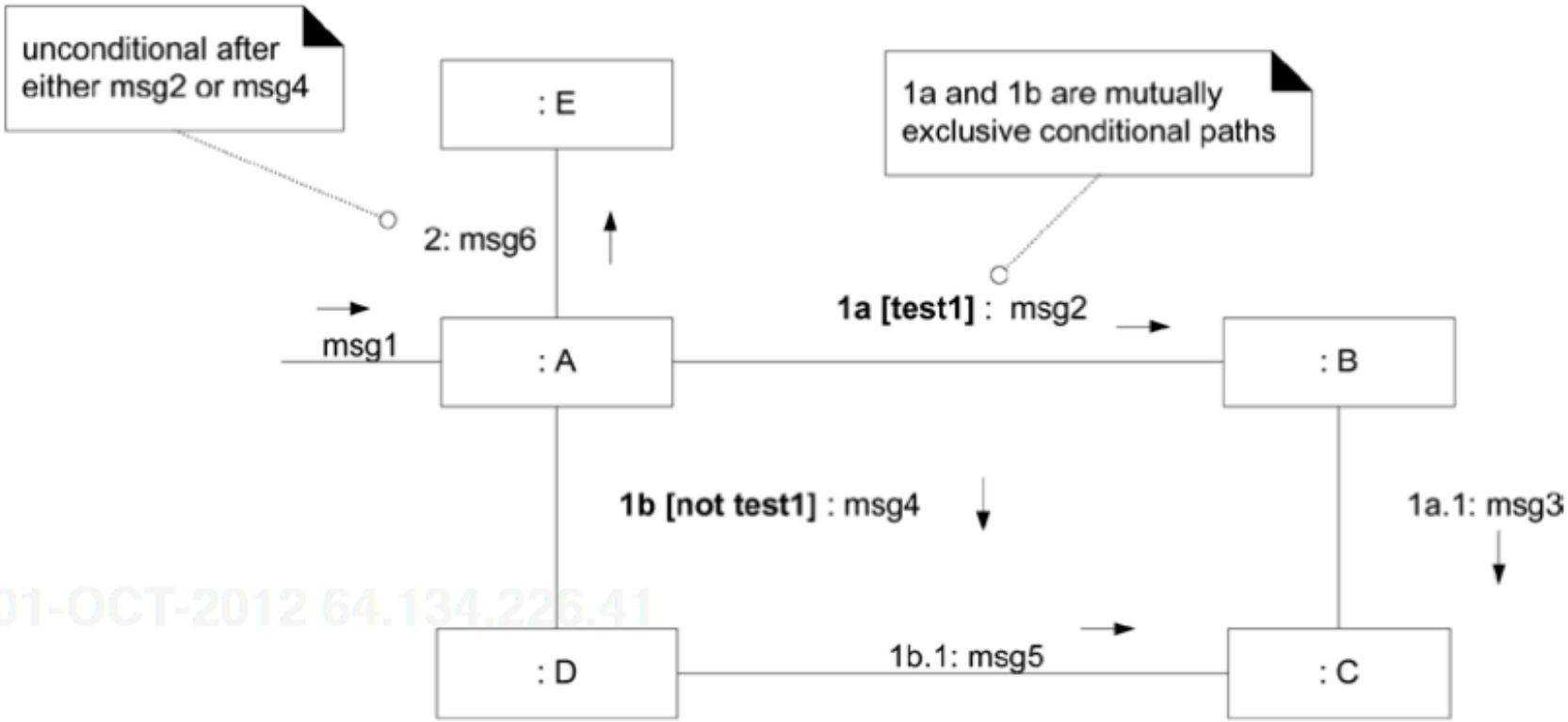
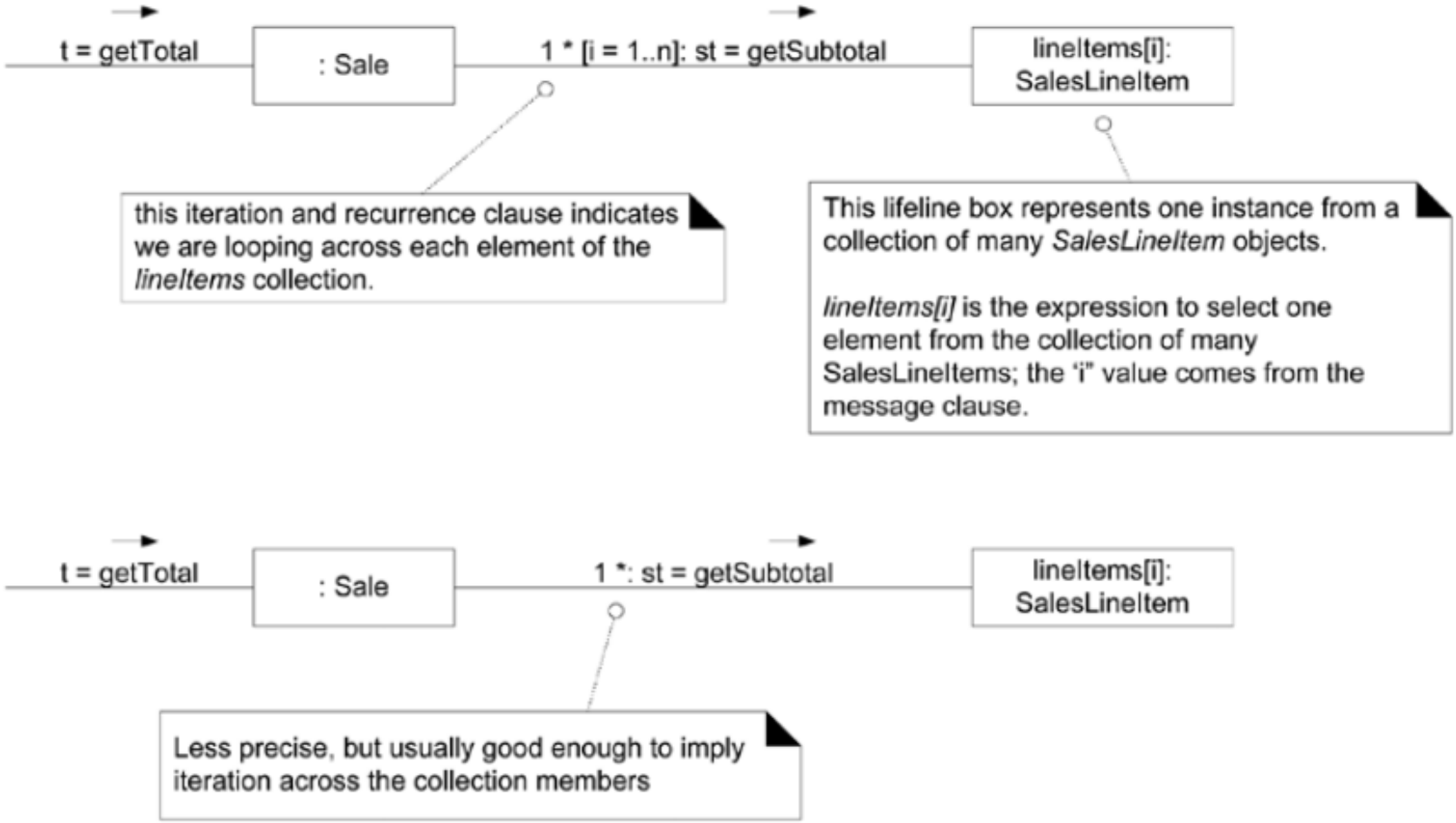
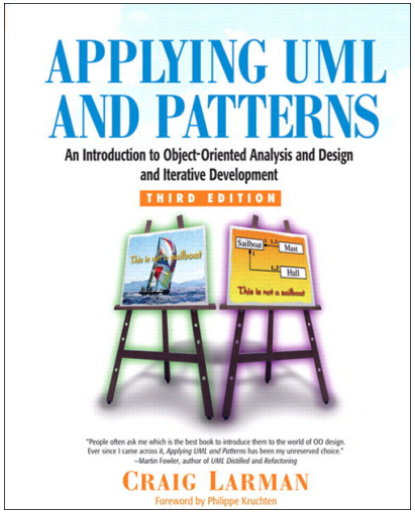
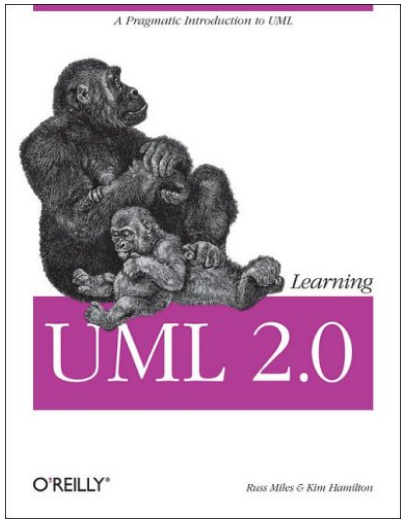


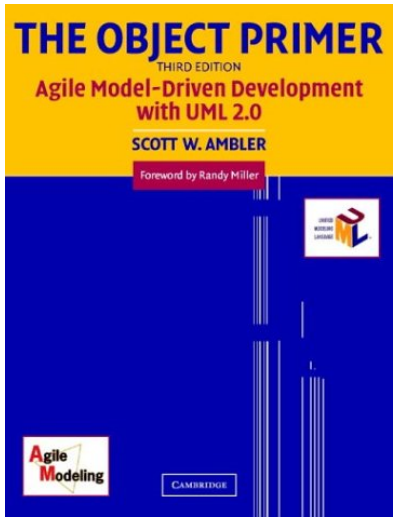
Figure 15.32. Iteration over a collection.

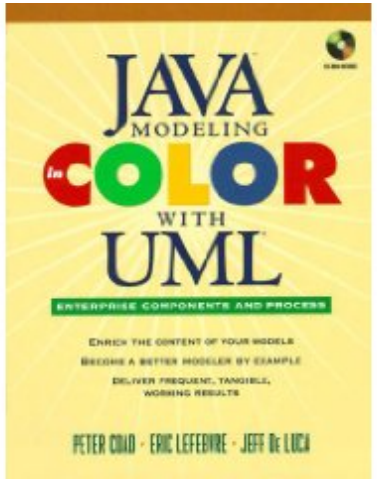


For Further Reading









Coad Book is out-of-print, but chapter one is available for download at: http://dn.codegear.com/article/29871