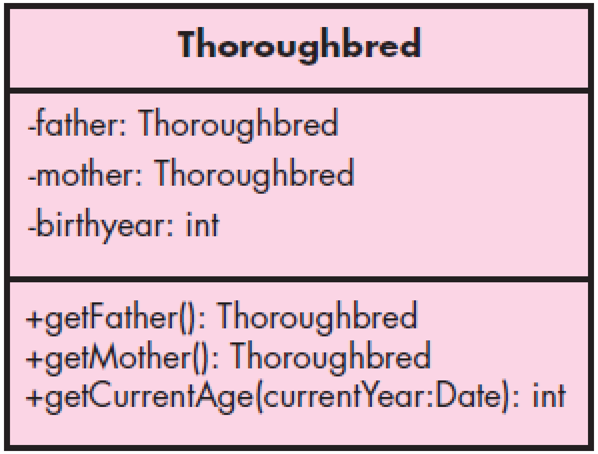
An Introduction to UML

Unified Modeling Language

- A standard language for writing software blueprints.
  - To visualize, specify, construct, and document the artifacts of a software-intensive system
  - To create UML diagrams to help software developers build the software.
- History
  - Grady Booch, Jim Rumbaugh, and Ivar Jacobson developed UML in the mid-1990s
  - In 1997, UML 1.0 was submitted to the Object Management Group
  - UML 1.0 was revised to UML 1.1 and adopted later that year
  - The current standard is UML 2.0 and is now an ISO standard

- UML 2.0 provides 13 different diagrams for use in software modeling
  - Class
  - Use case
  - Sequence
  - Communication
  - Activity
  - State
  - Deployment

A class diagram



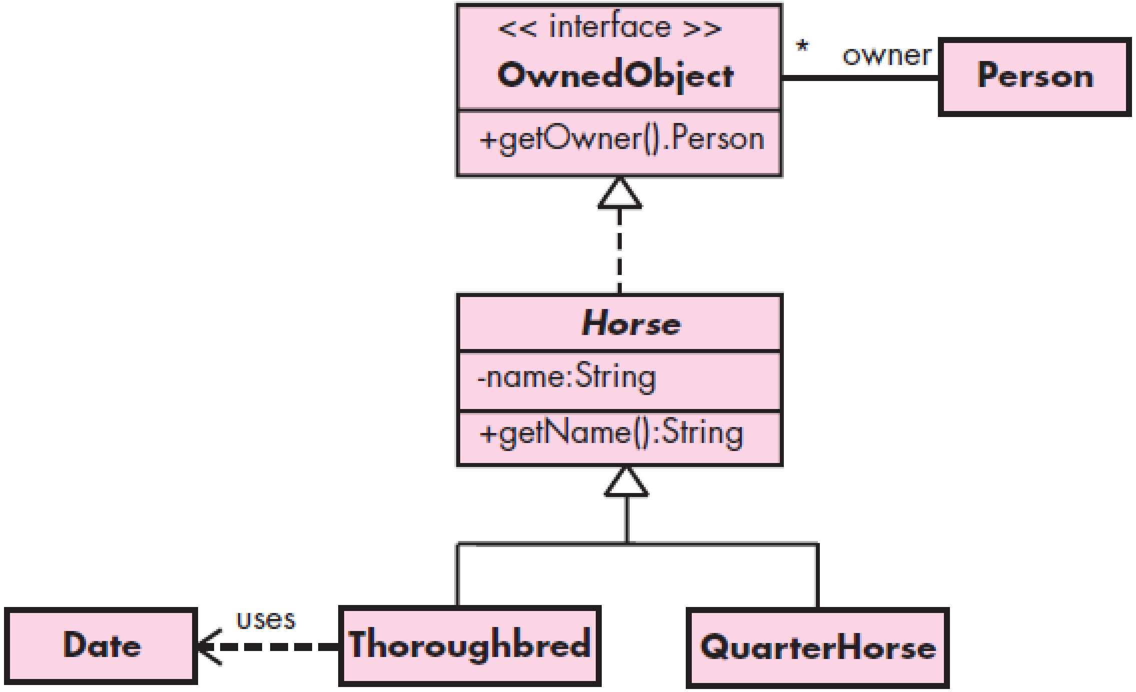
Thoroughbred

-father: Thoroughbred
-mother: Thoroughbred
-birthyear: int

+getFather(): Thoroughbred
+getMother(): Thoroughbred
+getCurrentAge(currentYear:Date): int

- To model classes, including their attributes, operations, and their relationships and associations with other classes
- The visibility is indicated by a preceding –, #, ~, or +
- An abstract class or abstract method is indicated by the use of italics for the name
- An interface is indicated by adding the phrase «interface» (called a *stereotype*) above the name.

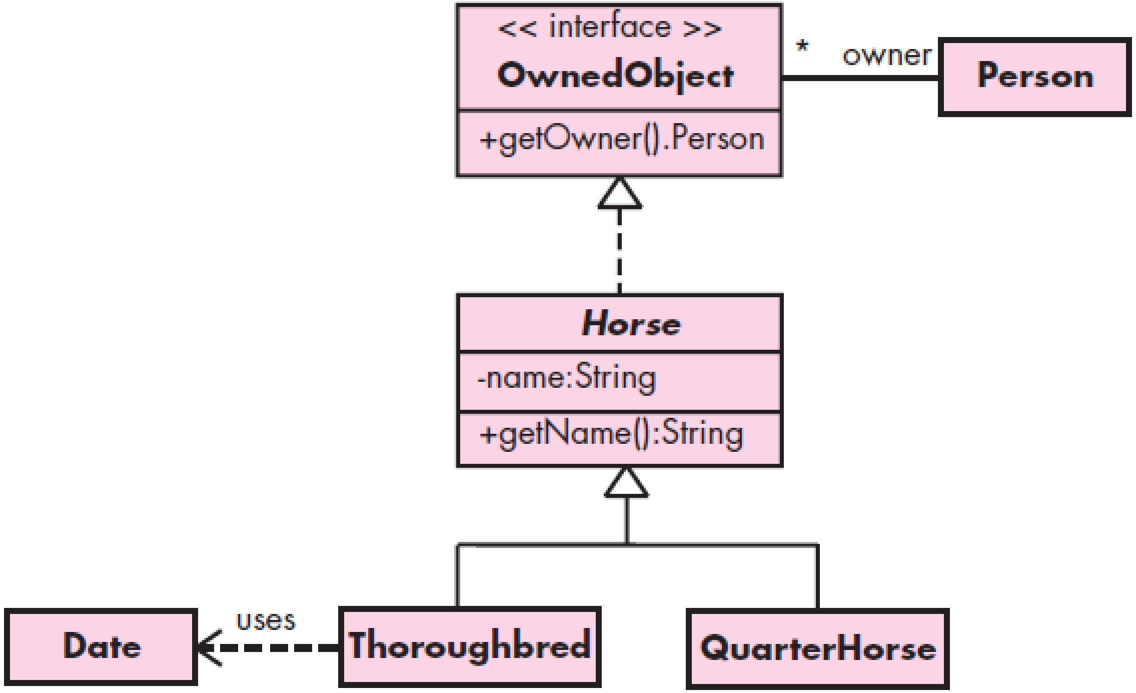
A class diagram



A class diagram regarding horses

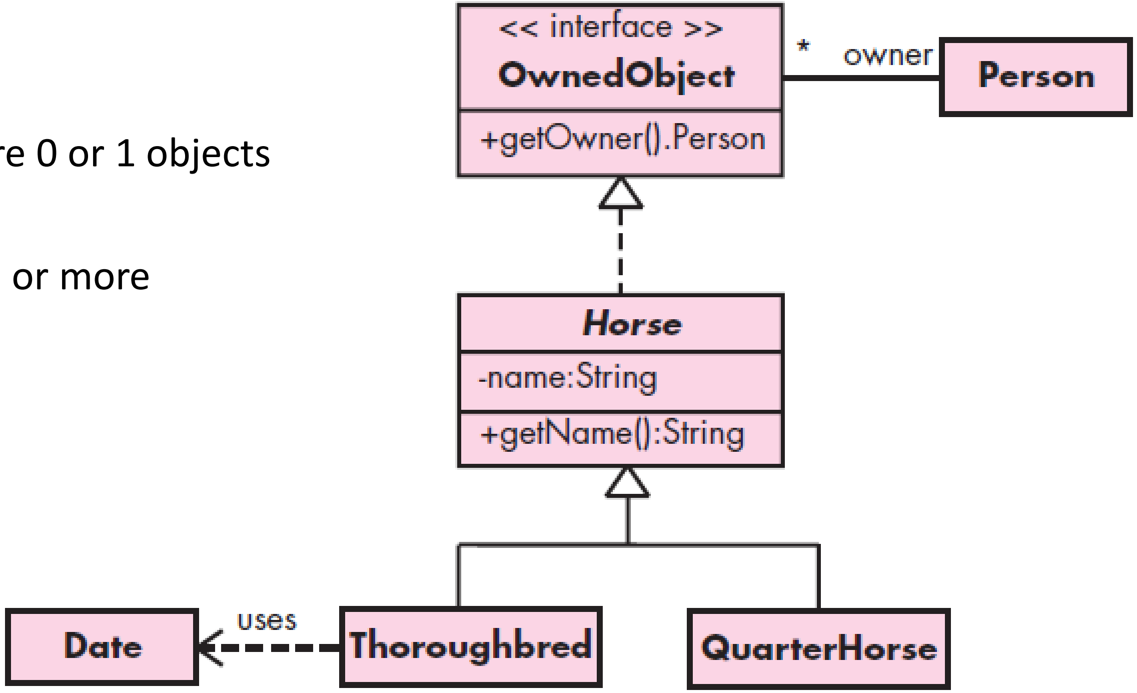
A class diagram

- a fourth section at the bottom of the class box can be used to list the responsibilities of the class
- *Generalization*: a class that is a subclass of another class
- *Realization*: indicates implementation of an interface



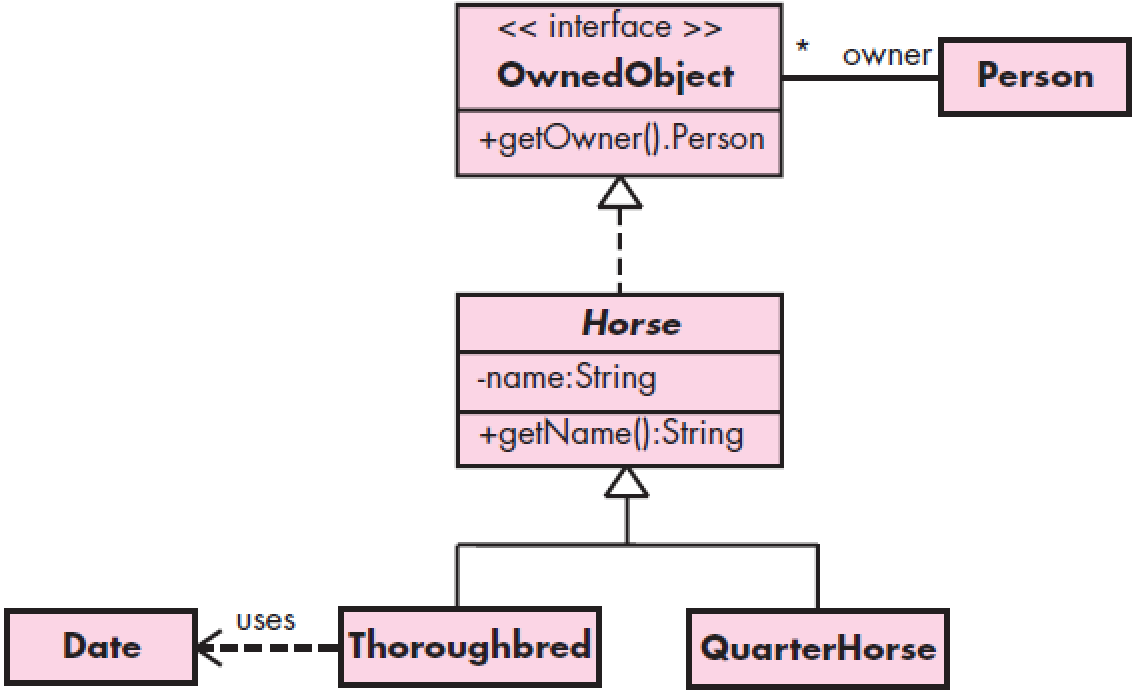
A class diagram

- *Association* between two classes means that there is a structural relationship between them.
  - Label, as can each of its ends, to indicate the role of each class in the association.
  - Arrows on either or both ends of an association line indicate navigability
  - Multiplicity
    - 0..1 means that there are 0 or 1 objects
    - 1..* means one or more
    - 0..* or just * means zero or more



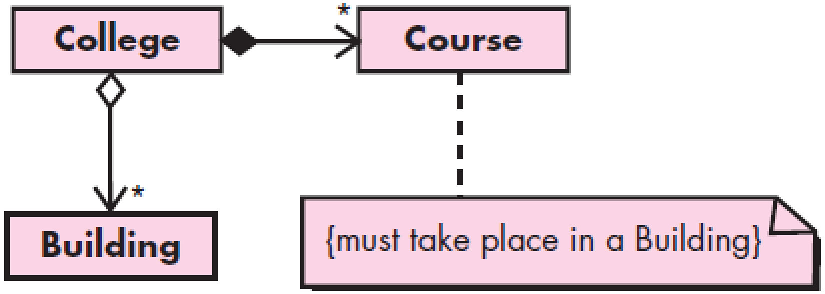
A class diagram

- An association might also connect a class with itself, using a loop.
- *Dependency:* One class depends on another if changes to the second class might require changes to the first class



A class diagram

- An *aggregation* is a special kind of association indicated by a hollow diamond on one end of the icon
  - An *aggregation* is a special kind of association It indicates a “whole/part” relationship
  - A *composition* is an aggregation indicating strong ownership of the parts. In a composition, the parts live and die with the owner
- *Note* contain comments about the role of a class or constraints that all objects of that class must satisfy.

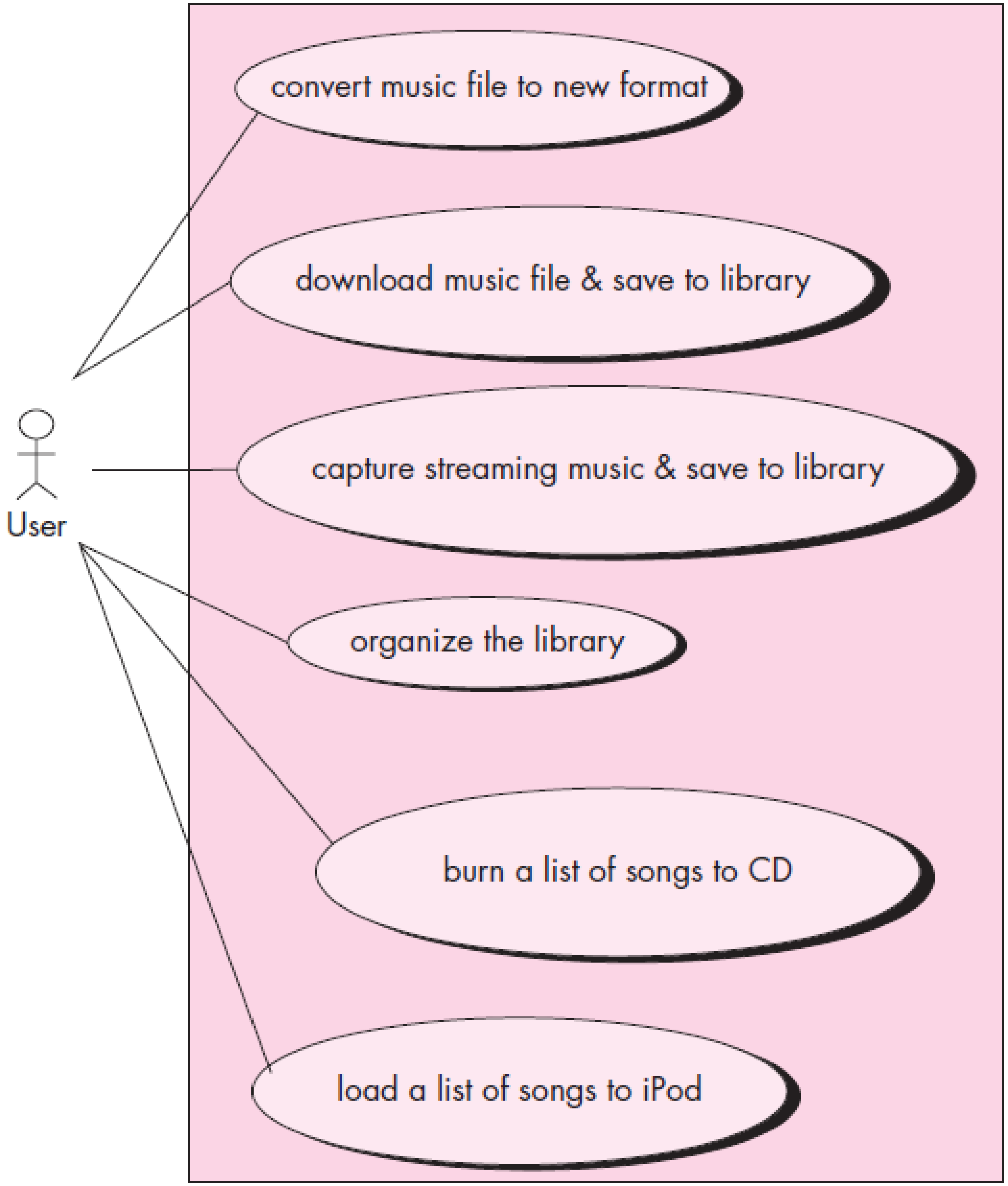


USE-CASE DIAGRAMS

- Use cases (Chapters 5 and 6) and the UML *use-case diagram* help you determine the functionality and features of the software from the user's perspective.

USE-CASE DIAGRAMS

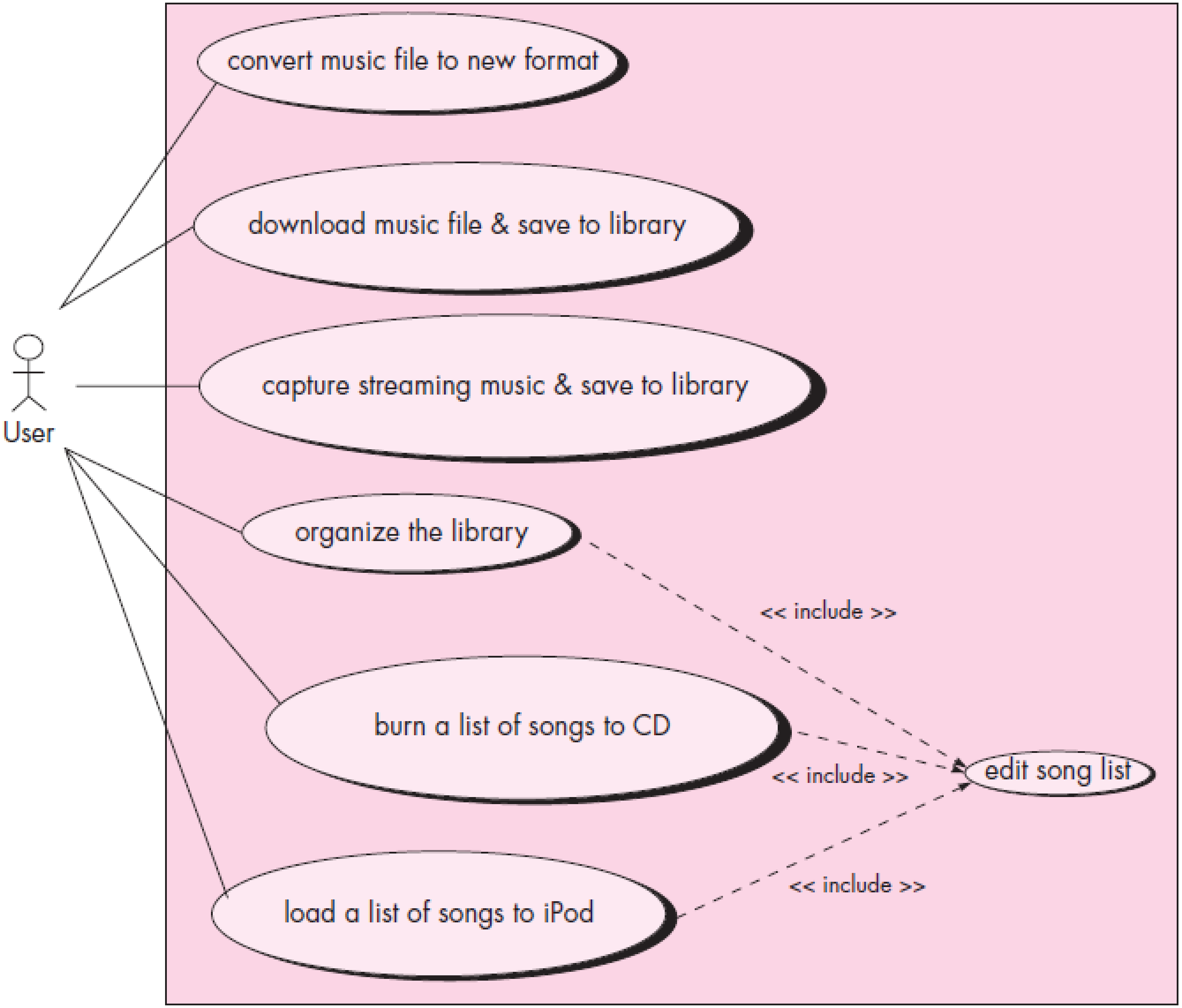
- A *use case* describes how a user interacts with the system by defining the steps required to accomplish a specific goal
- Variations in the sequence of steps describe various scenarios
- A UML *use-case diagram* is an overview of all the use cases and how they are related. It provides a big picture of the functionality of the system


convert music file to new format
download music file & save to library
capture streaming music & save to library
User
organize the library
burn a list of songs to CD
load a list of songs to iPod

A use-case diagram for the music system

USE-CASE DIAGRAMS

- The stick figure represents an *actor* that is associated with one category of user (or other interaction element).
- The *actors* are connected by lines to the use cases that they carry out.
- To avoid duplication in use cases, it is usually better to create a new use case representing the duplicated activity, and then let the other uses cases include this new use case as one of their steps.

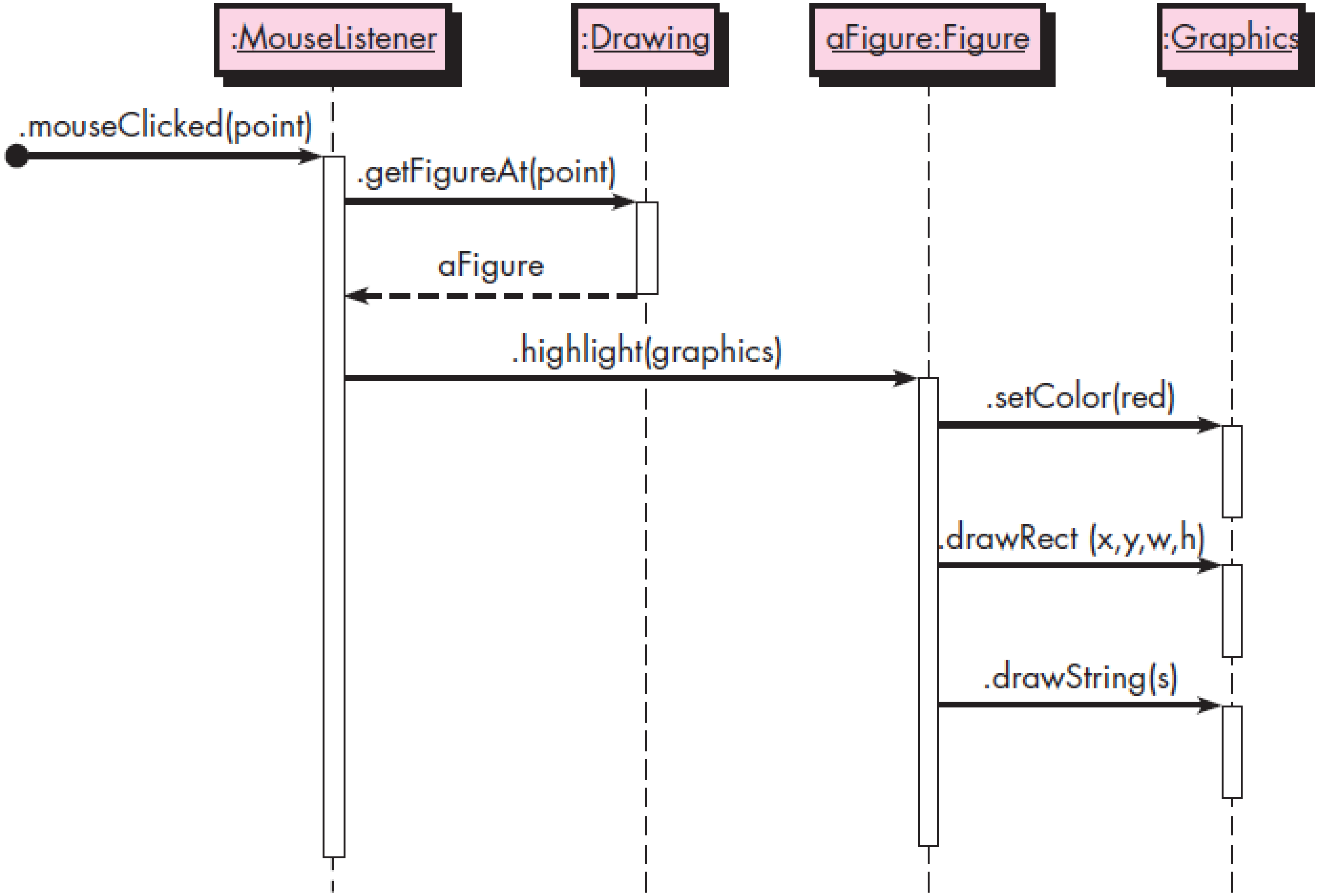

convert music file to new format
download music file & save to library
capture streaming music & save to library
User
organize the library
<< include >>
burn a list of songs to CD
<< include >>
edit song list
load a list of songs to iPod
<< include >>

A use-case diagram with included use cases

SEQUENCE DIAGRAMS

- a *sequence diagram* is used to show the dynamic communications between objects during execution of a task.
- It shows the temporal order in which messages are sent between the objects to accomplish that task.
- One might use a sequence diagram to show the interactions in one use case or in one scenario of a software system.

The diagram shows the steps involved in highlighting a figure in a drawing when it has been clicked.



A sample sequence diagram

SEQUENCE DIAGRAMS

- When an object is executing a method you can optionally display a white bar, called an *activation bar*.
- show the return from a method call with a dashed arrow and an optional label
- A black circle with an arrow coming from it indicates a *found message* whose source is unknown or irrelevant.

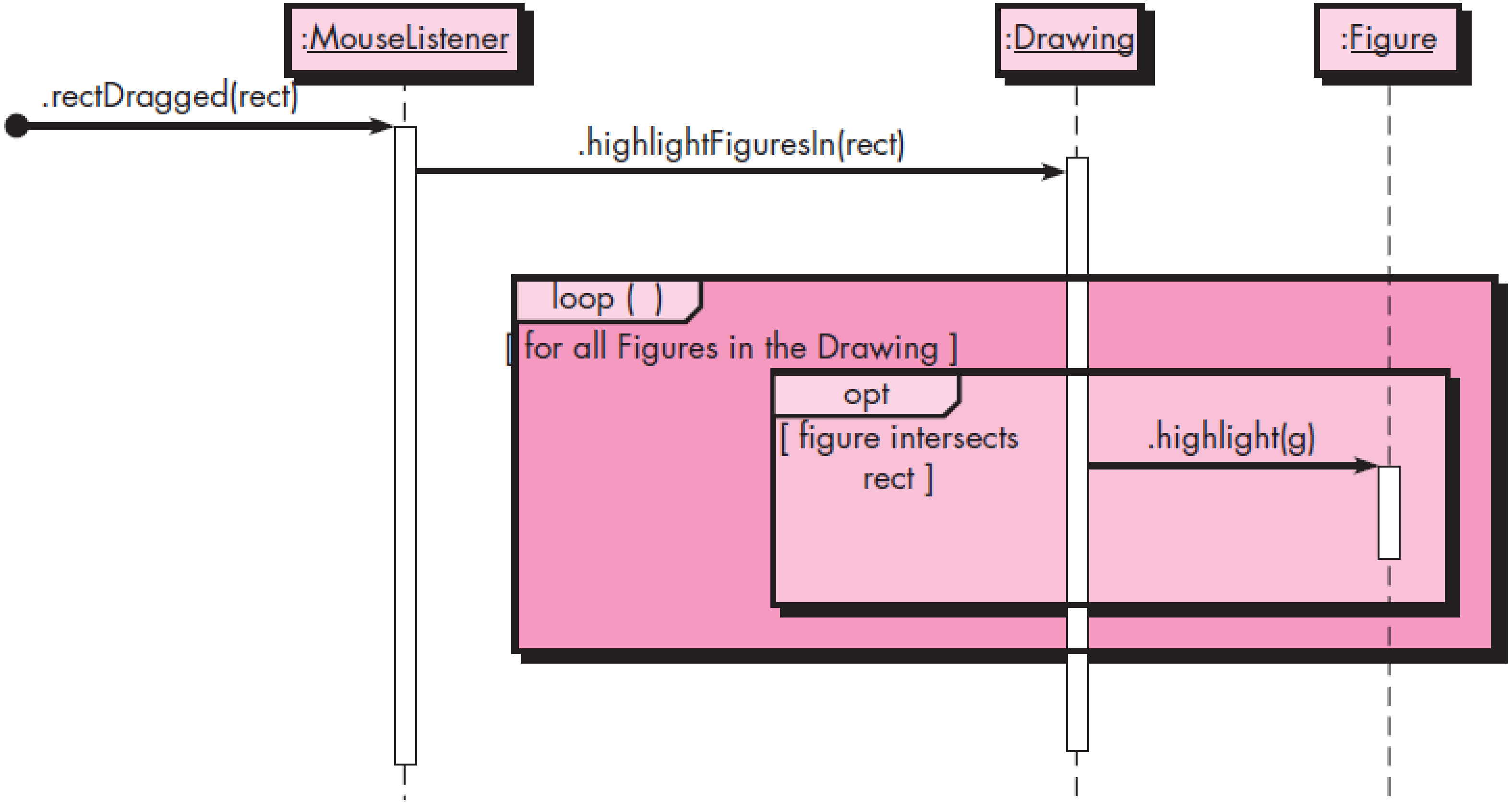
SEQUENCE DIAGRAMS

- Each box in the row at the top of the diagram usually corresponds to an object, although it is possible to have the boxes model other things, such as classes.
- Below each box there is a dashed line called the *lifeline* of the object.
- The vertical axis in the sequence diagram corresponds to time, with time increasing as you move downward.
- A sequence diagram shows method calls using horizontal arrows from the *caller* to the *callee,* labeled with the method name and optionally including its parameters, their types, and the return type.

SEQUENCE DIAGRAMS

- If you insist on including loops, conditionals, and other control structures in a sequence diagram, you can use *interaction frames*,
- which are rectangles that surround parts of the diagram and that are labeled with the type of control structures they represent.
- The phrases in square brackets are called *guards*, which are Boolean conditions

showing the process involved in highlighting all figures inside a given rectangle.

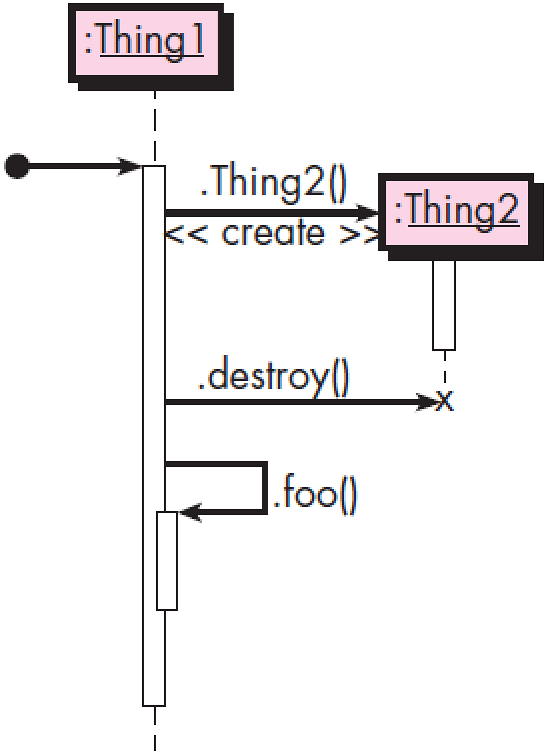

:MouseListener
:Drawing
:Figure
.rectDragged(rect)
.highlightFiguresIn(rect)
loop ( )
[for all Figures in the Drawing]
opt
[figure intersects rect]
.highlight(g)

A sequence diagram with two interaction frames

SEQUENCE DIAGRAMS

- You can distinguish between synchronous and asynchronous messages
- You can show an object sending itself a message
- You can show object creation by drawing an arrow appropriately labeled (for example, with a «create» label)
- You can show object destruction by a big X at the end of the object’s lifeline.

SEQUENCE DIAGRAMS

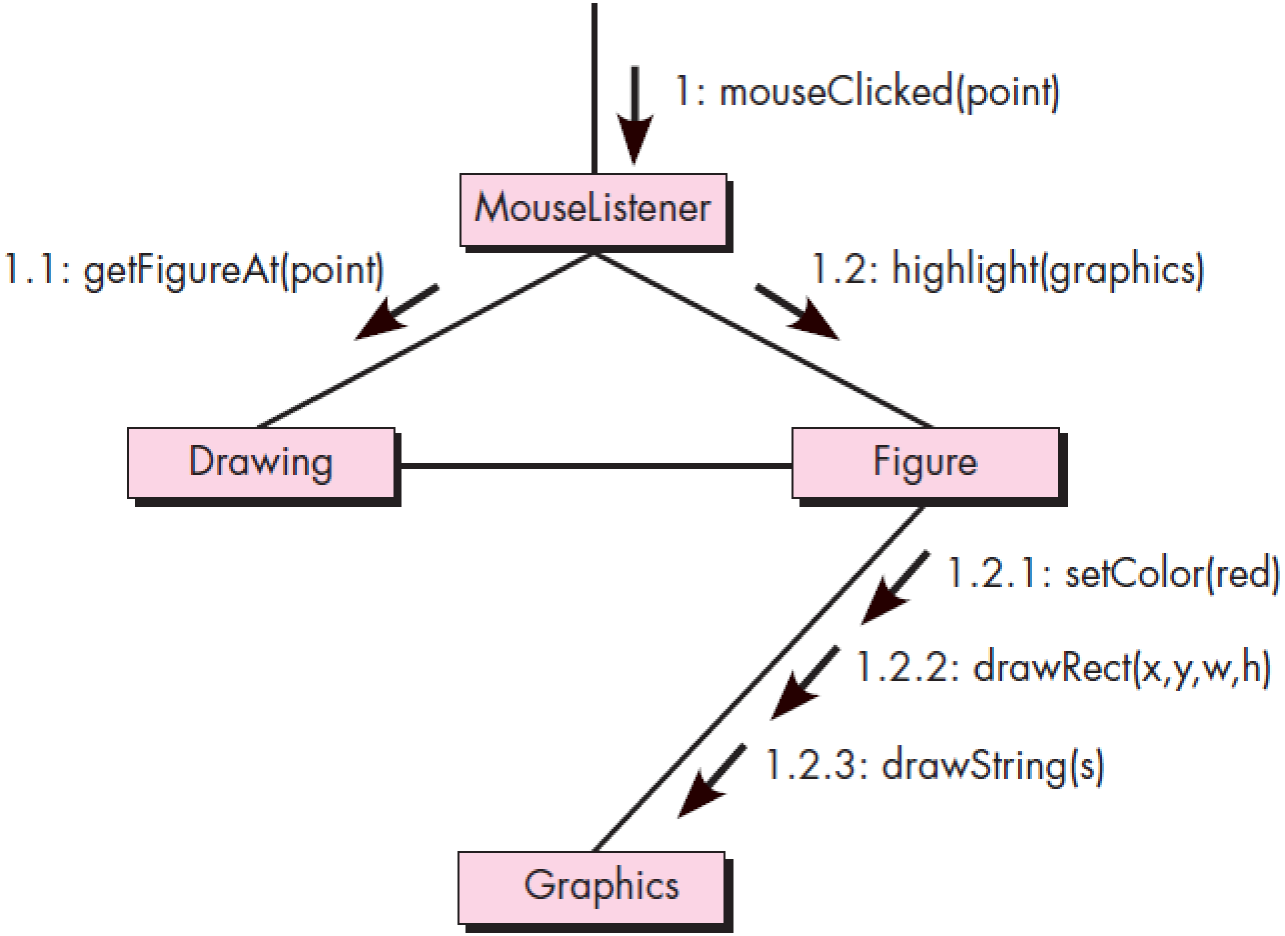


Creation, destruction, and loops in sequence diagrams

COMMUNICATION DIAGRAMS

- The UML *communication diagram* (called a “collaboration diagram” in UML 1.X) provides another indication of the temporal order of the communications but emphasizes the relationships among the objects and classes instead of the temporal order.

COMMUNICATION DIAGRAMS



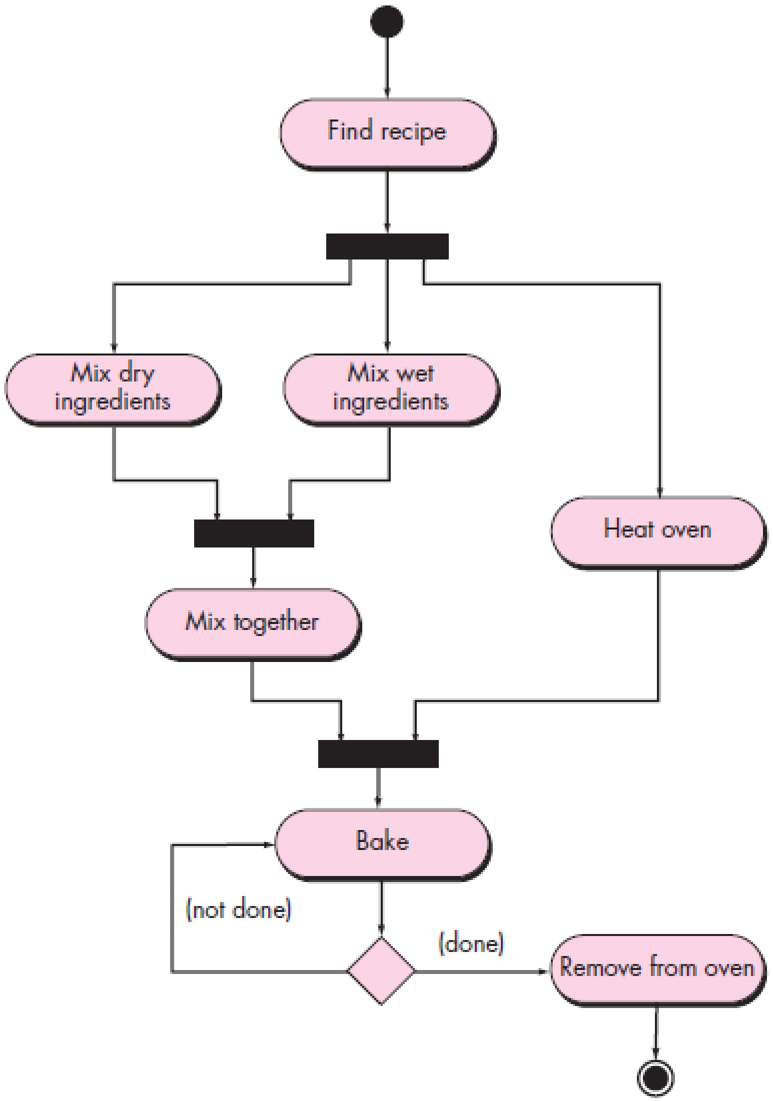
A UML communication diagram

COMMUNICATION DIAGRAMS

- There are many optional features that can be added to the arrow labels.
  - An incoming arrow could be labeled A1: mouseClicked(point). indicating an execution thread, A.

ACTIVITY DIAGRAMS

- A UML *activity diagram* depicts the dynamic behavior of a system or part of a system through the flow of control between actions that the system performs.
- an *action* node, represented by a rounded rectangle, which corresponds to a task performed by the software system.
- *Arrows* from one action node to another indicate the flow of control.
- *A solid black dot* forms the *initial node* that indicates the starting point of the activity.
- A black dot surrounded by a black circle is the *final node* indicating the end of the activity.



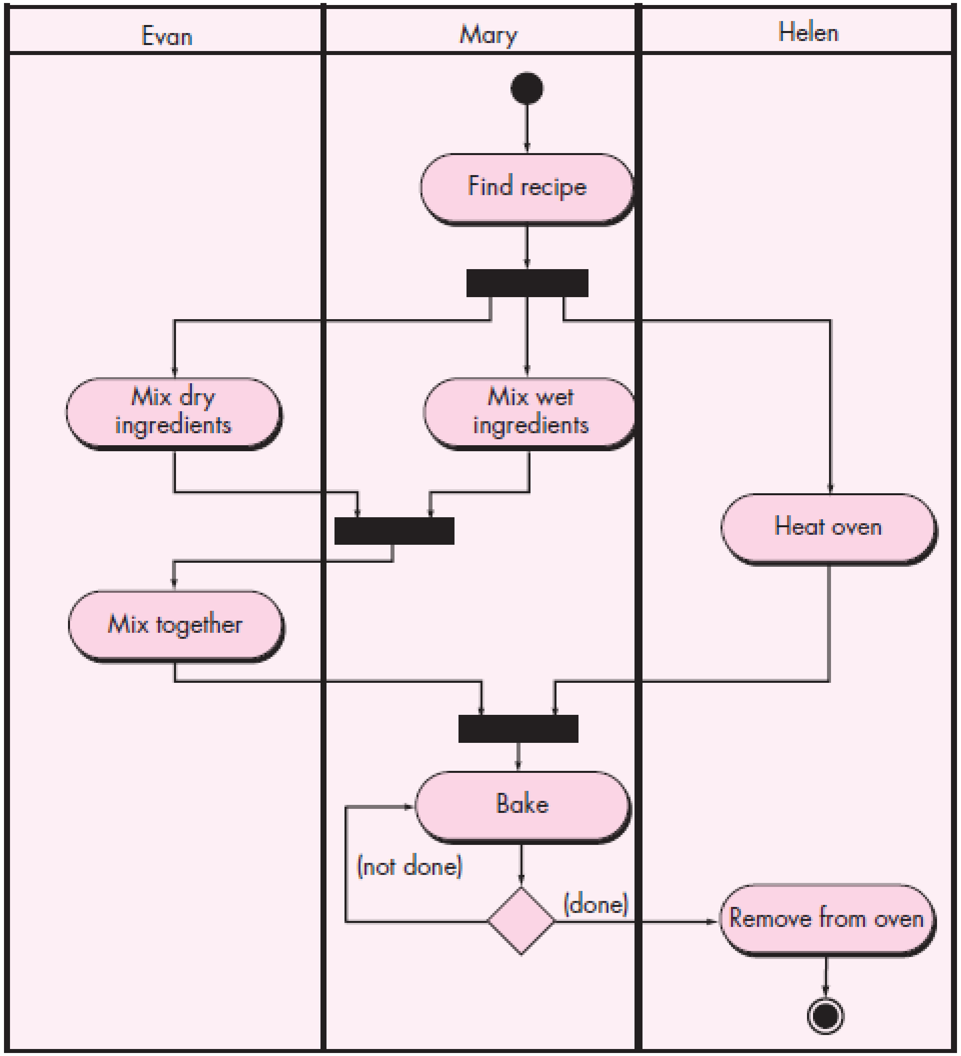
A UML activity diagram showing how to bake a cake

ACTIVITY DIAGRAMS

- A *fork* represents the separation of activities into two or more concurrent activities.
- A *join* is a way of synchronizing concurrent flows of control.
- A *decision* node corresponds to a branch in the flow of control based on a condition. Each outgoing arrow is labeled with a guard (a condition inside square brackets).

ACTIVITY DIAGRAMS

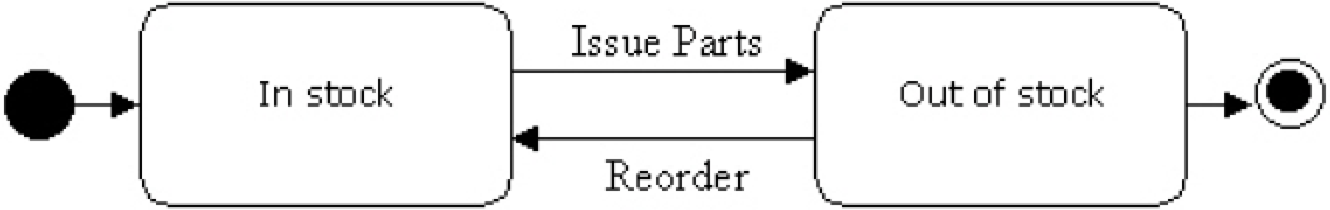
- if you do want to indicate how the actions are divided among the participants, you can decorate the activity diagram with *swimlanes*

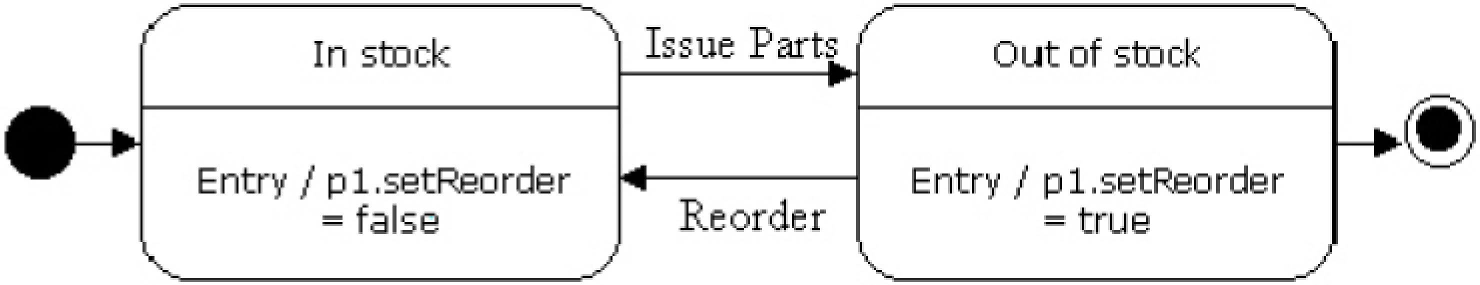


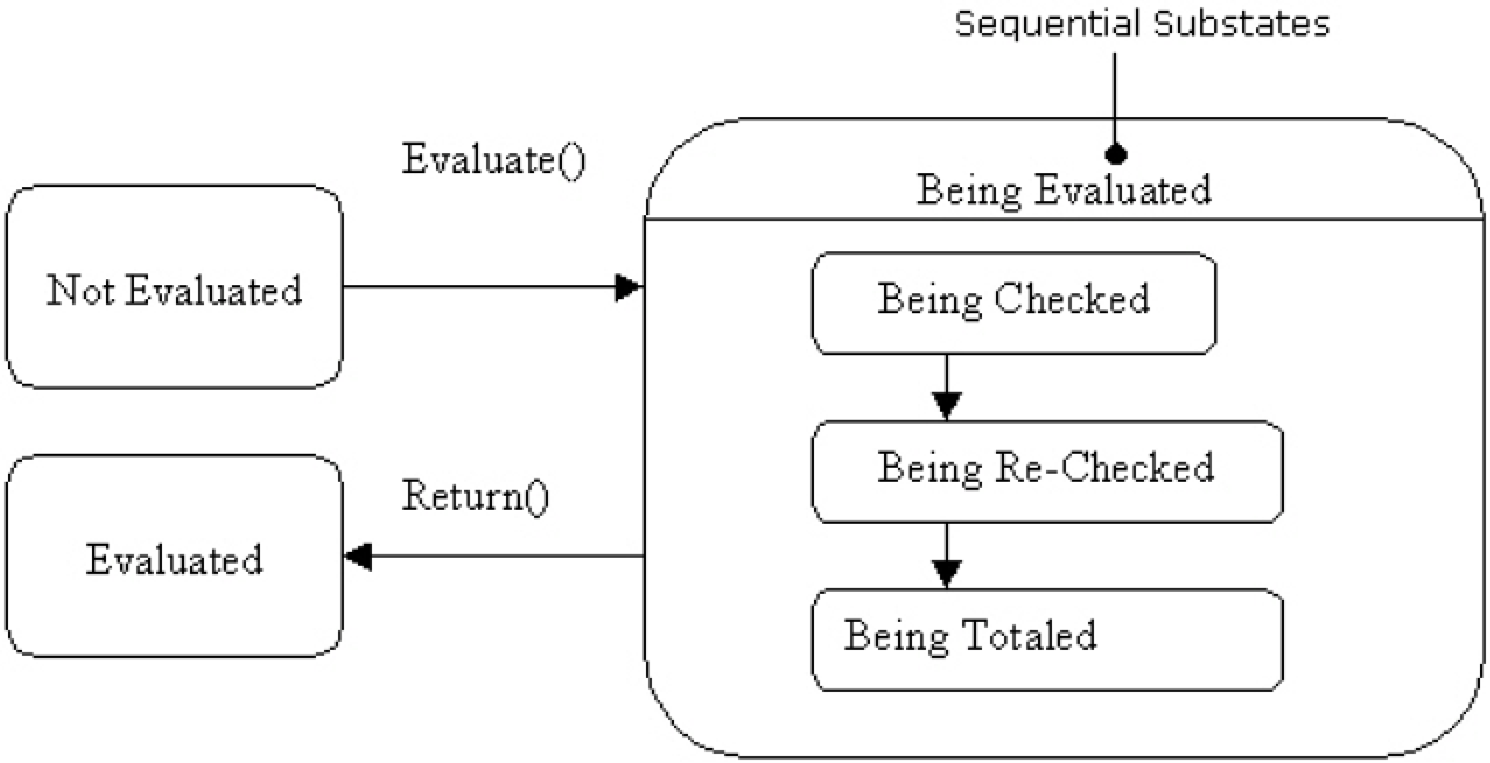
The cakebaking activity diagram with swimlanes added

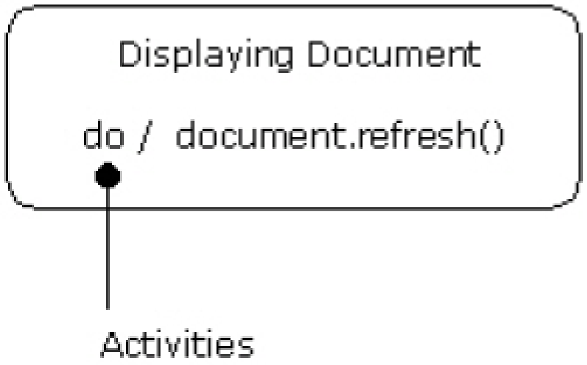
STATE DIAGRAMS

- The behavior of an object at a particular point in time often depends on the state of the object, that is, the values of its variables at that time.


In stock
Issue Parts
Reorder
Out of stock


In stock
Entry / p1.setReorder = false
Issue Parts
Reorder
Out of stock
Entry / p1.setReorder = true

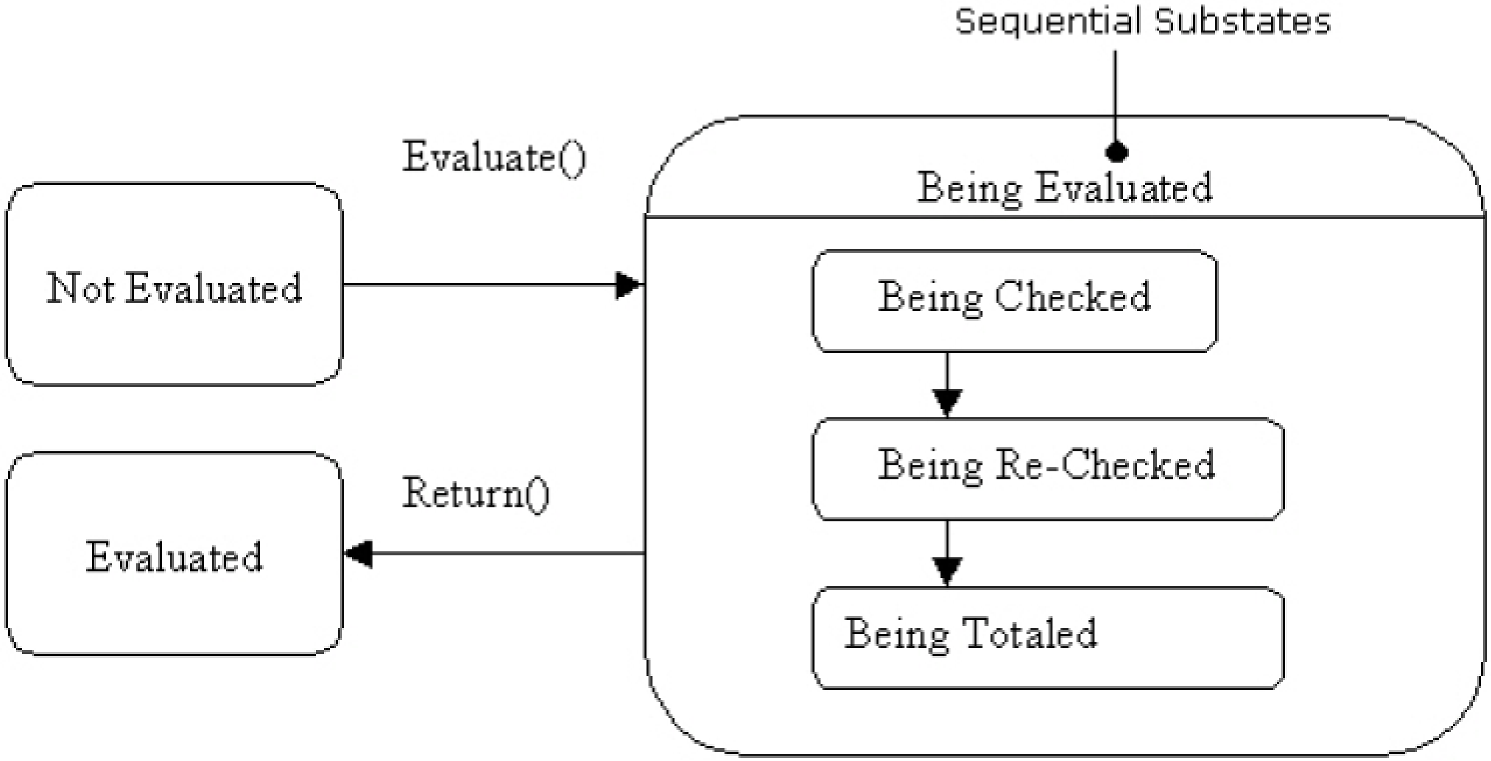

Sequential Substates
Evaluate()
Being Evaluated
Not Evaluated
Being Checked
Being Re-Checked
Return()
Evaluated
Being Totaled

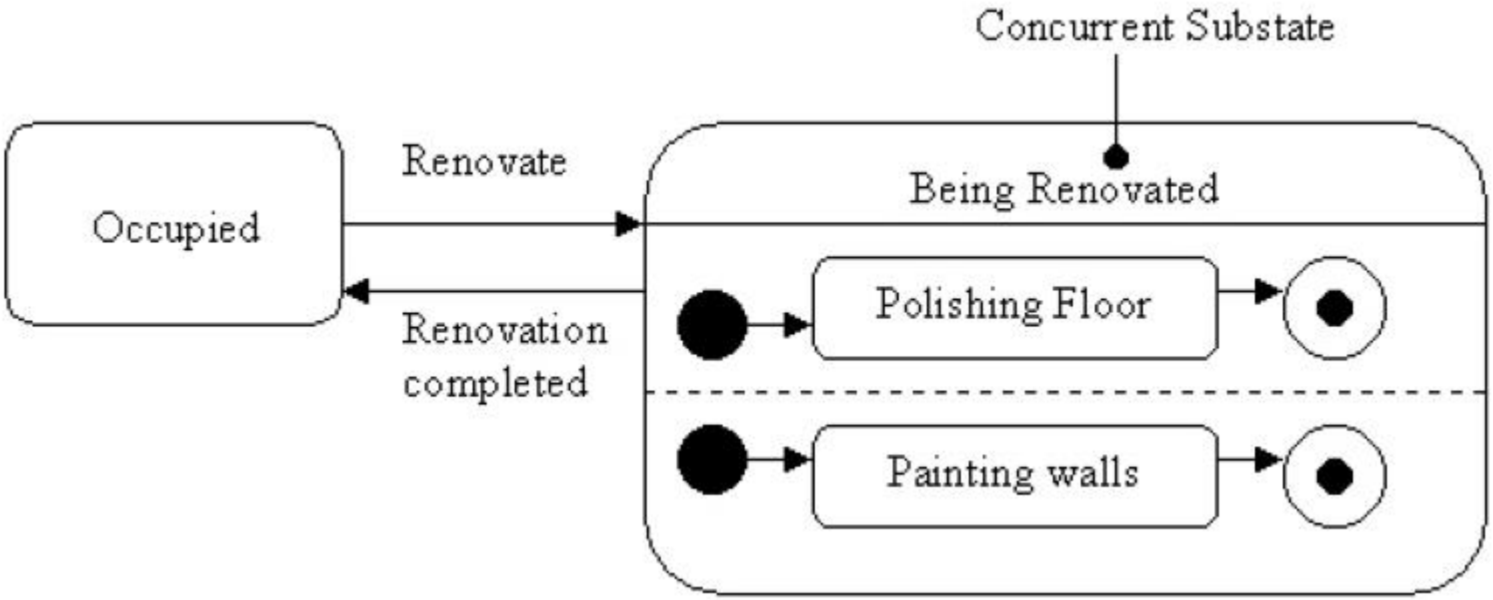


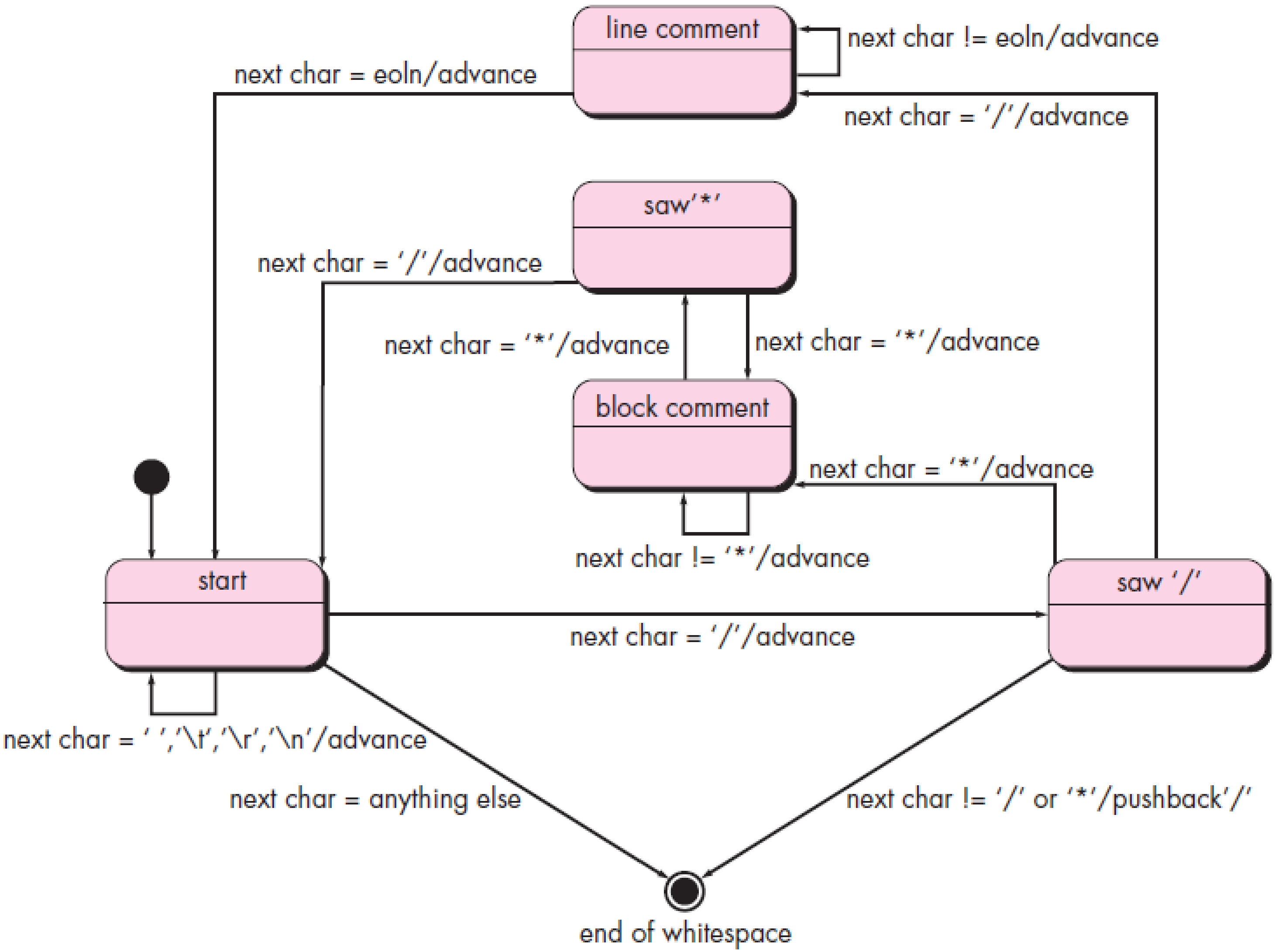
Displaying Document

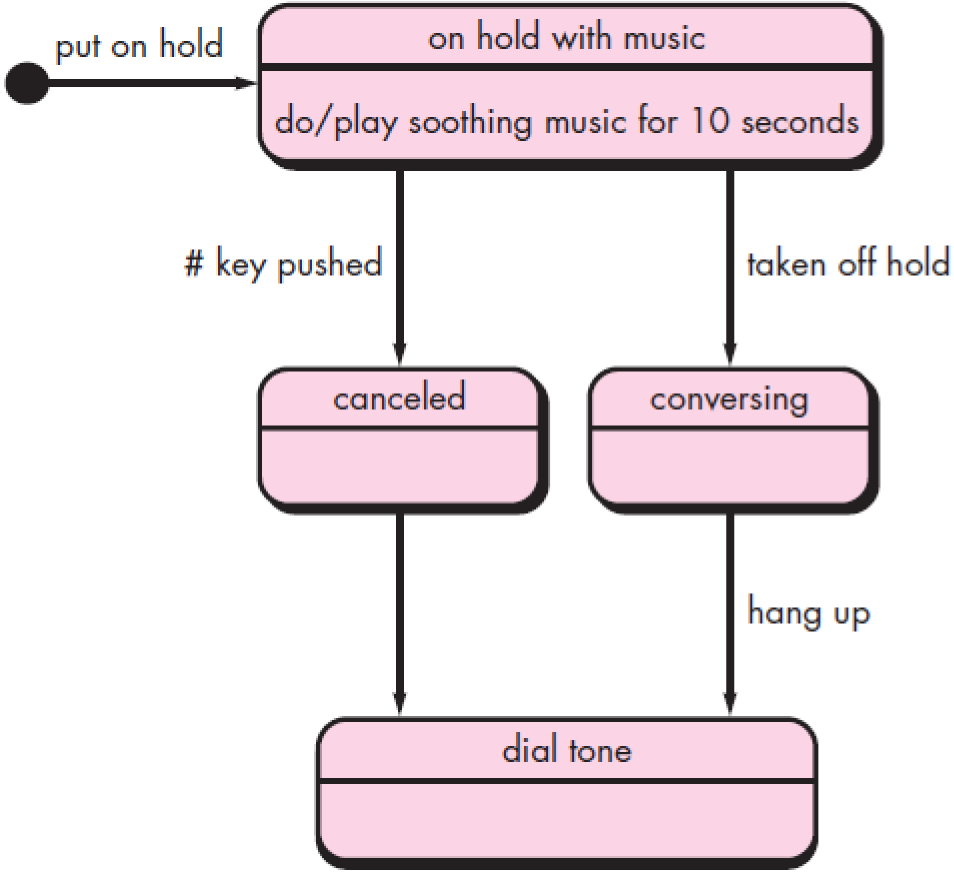
do / document.refresh()

Activities


Sequential Substates
Evaluate()
Being Evaluated
Not Evaluated
Being Checked
Being Re-Checked
Return()
Evaluated
Being Totaled


Concurrent Substate
Occupied
Renovate
Renovation completed
Being Renovated
Polishing Floor
Painting walls


line comment
next char != eoln/advance
next char = eoln/advance
next char = ‘/’/advance
saw’*’
next char = ‘/’/advance
next char = ‘*’/advance
next char = ‘*’/advance
block comment
next char = ‘*’/advance
next char != ‘*’/advance
start
saw ‘/’
next char = ‘/’/advance
next char = ‘ ’,’\t’,’\r’,’\n’/advance
next char = anything else
next char != ‘/’ or ‘*’/pushback’/’
end of whitespace


put on hold
on hold with music
do/play soothing music for 10 seconds
# key pushed
taken off hold
canceled
conversing
hang up
dial tone