UML, the Unified Modeling Language

William H. Mitchell (whm)
Mitchell Software Engineering (.com)

What is UML?

"The Unified Modeling Language (UML) is a language for specifying, visualizing, constructing, and documenting the artifacts of software systems, as well as for business modeling and other non-software systems".— OMG UML Specification

"UML is a graphical notation for modeling various aspects of software systems." — whm

Why use UML?

Two questions, really:

Why use a graphical notation of any sort?

- Facilitates construction of models that in turn can be used to:
  - Reason about system behavior
  - Present proposed designs to others
  - Document key elements of design for future understanding

Which graphical notation should be used?

- UML has become the de-facto standard for modeling object-oriented systems.
- UML is extensible and method-independent.
- UML is not perfect, but it's good enough.

The Origins of UML

Object-oriented programming reached the mainstream of programming in the late 1980's and early 1990's.

The rise in popularity of object-oriented programming was accompanied by a profusion of object-oriented analysis and design methods, each with its own graphical notation.

Three OOA/D gurus, and their methods, rose to prominence

> Grady Booch — The Booch Method
> James Rumbaugh, et al. — Object Modeling Technique
> Ivar Jacsobson — Objectory

In 1994, Booch and Rumbaugh, then both at Rational, started working on a unification of their methods. A first draft of their Unified Method was released in October 1995.

In 1996, (+/-) Jacobson joined Booch and Rumbaugh at Rational; the name UML was coined.

In 1997 the Object Management Group (OMG) accepted UML as an open and industry standard visual modeling language for object-oriented systems.

Current version of UML is 1.4.

Official specification is available at www.omg.org (566 pages)

UML Diagram Types

There are several types of UML diagrams:

Use-case Diagram
: Shows actors, use-cases, and the relationships between them.

Class Diagram
: Shows relationships between classes and pertinent information about classes themselves.

Object Diagram
: Shows a configuration of objects at an instant in time.

Interaction Diagrams
: Show an interaction between a group of collaborating objects. Two types: Collaboration diagram and sequence diagram

Package Diagram
: Shows system structure at the library/package level.

State Diagram
: Describes behavior of instances of a class in terms of states, stimuli, and transitions.

Activity Diagram
: Very similar to a flowchart—shows actions and decision points, but with the ability to accommodate concurrency.

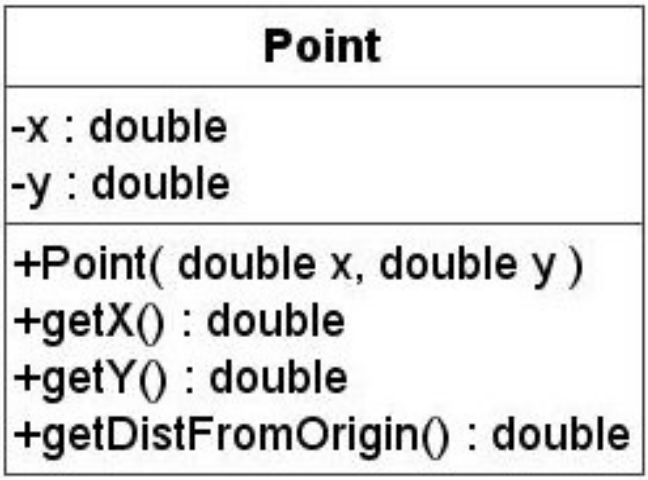
Deployment Diagram
: Shows configuration of hardware and software in a distributed system.

Class diagram basics

Consider a simple class to represent a point on a Cartesian plane:

```
class Point {
   private double x, y;

   Point(double x, double y) { this.x = x; this.y = y; }
   public double getX( ) { return x; }
   public double getY( ) { return y; }

   public double getDistFromOrigin( ) { ... }
   }
```

The corresponding UML class diagram:



Three *compartments* are shown: class name, attributes, operations.

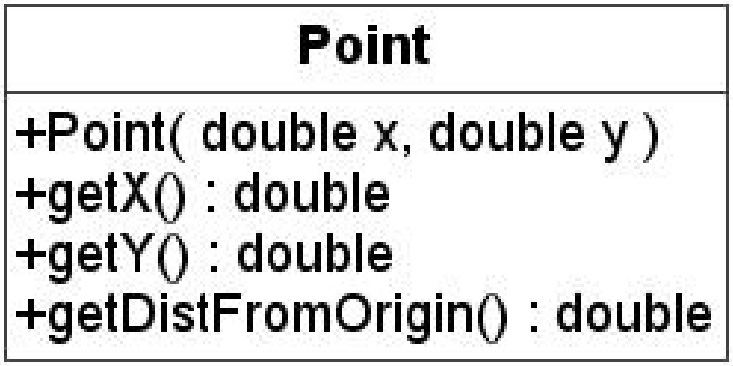
Member visibility is indicated with + (public) and - (private); these are called *visibility adornments*.

Class diagram basics, continued

UML permits the detail of a diagram to vary based on the intended use. The compartments with attributes and/or operations can be omitted.

Here is a less detailed diagram for the Point class:

Here's a version with only the operations shown:

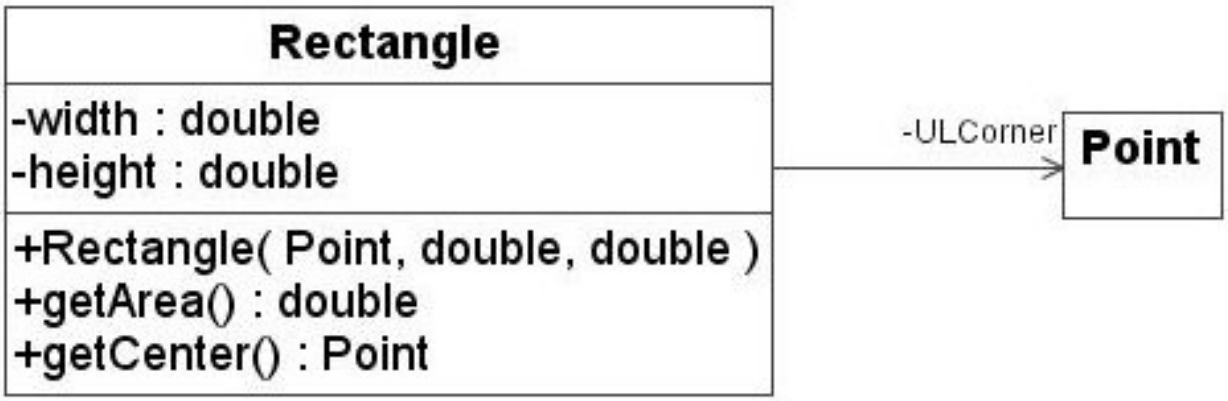


Class diagram basics, continued

Here is a class that represents rectangles using a point, a width, and a height:

```
class Rectangle {
    private double width, height;
    private Point ULCorner;

    public Rectangle(Point ULC, double w, double h) { ... }
    public double getArea() { ... }
    public Point getCenter() { ... }
    }
```

A class diagram showing the relationship between Rectangle and Point:



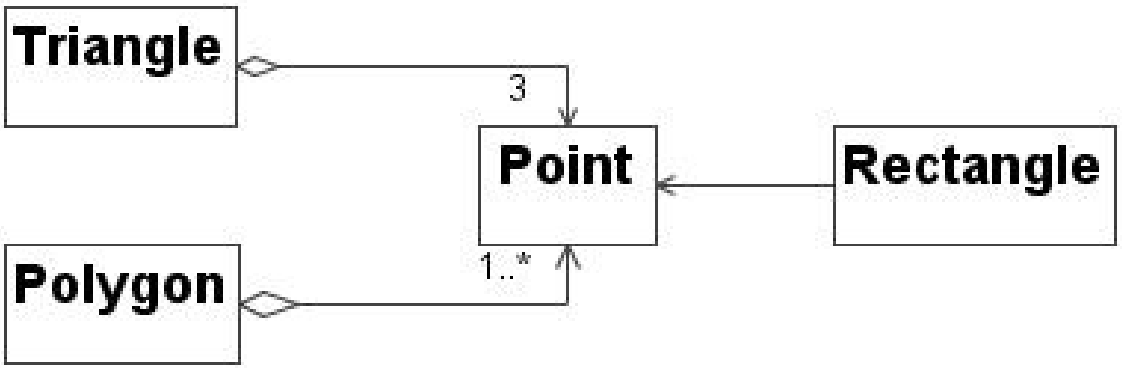
The link between Rectangle and Point indicates:

- There is a field in Rectangle that references a Point.
- Point has no knowledge of Rectangle. (We'd expect no references to Rectangle in the source for Point.)
- Point's *role* with respect to Rectangle is that of "ULCorner".

Class diagrams—aggregation

When a class may reference several instances of another class, the link between the two classes is shown with a diamond on the end of the aggregate class.

Here is a class diagram showing the relationships between Triangle, Polygon, Rectangle, and Point classes:



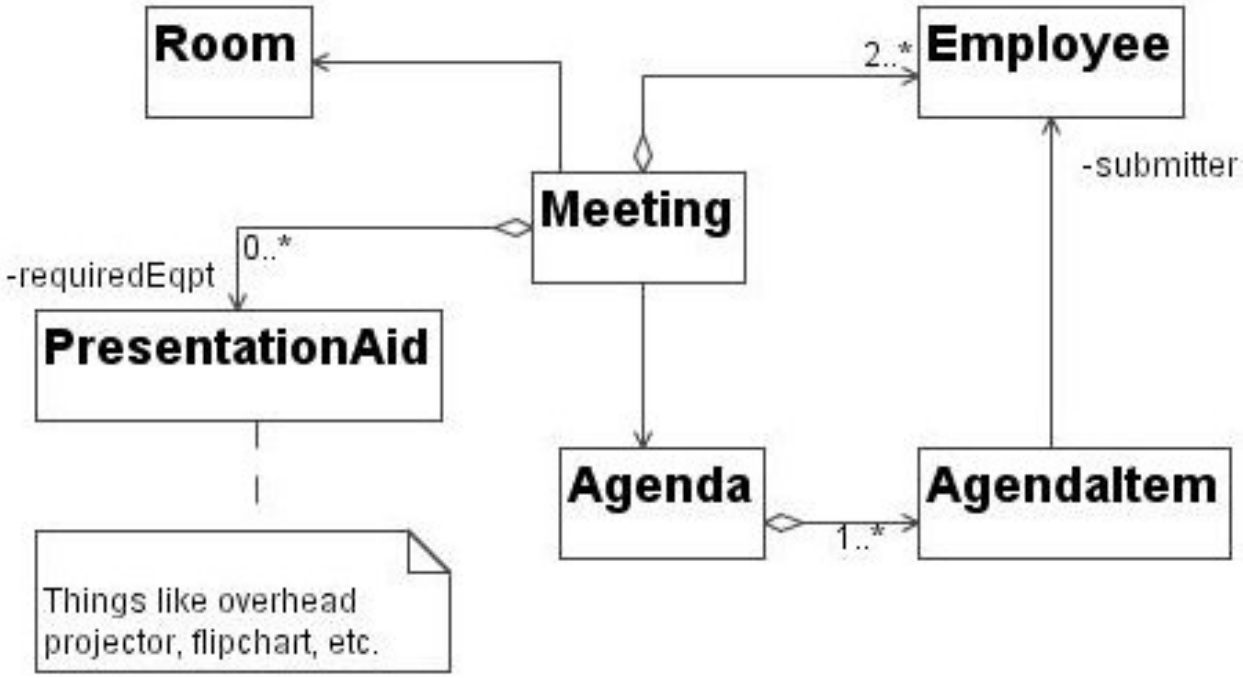
It can be seen that:

- An instance of Triangle references three instances of Point.
- An instance of Polygon references at least one, and a potentially unlimited number of Point instances.
- Aside from Triangle, Polygon, and Rectangle knowing of Point, no classes have any knowledge of any other classes.

Note that "3" and "1..*" are *multiplicity* specifications.

Aggregation, continued

Another example of aggregation:



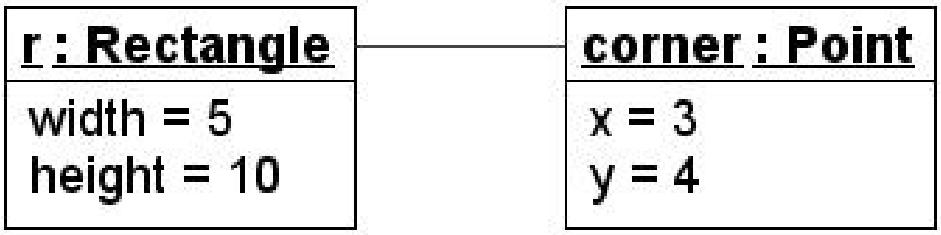
Object diagrams

An *object diagram* shows a configuration of objects at a point in time.

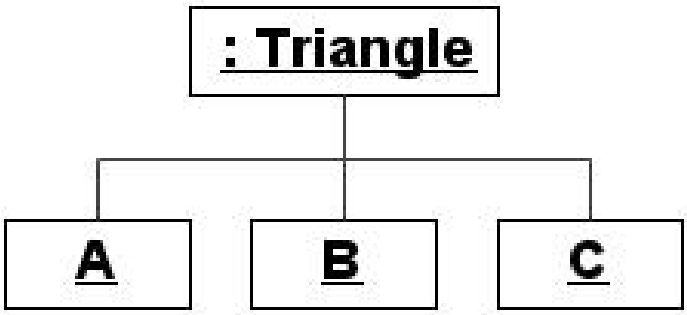
After execution of this code,

```
Point corner = new Point(3,4);
Rectangle r = new Rectangle(corner, 5, 10);
```

the situation can be depicted with this object diagram:



If desired, various elements may be omitted:



Note that an underlined name in the name compartment is the indication that an object is at hand.

Collaboration diagrams

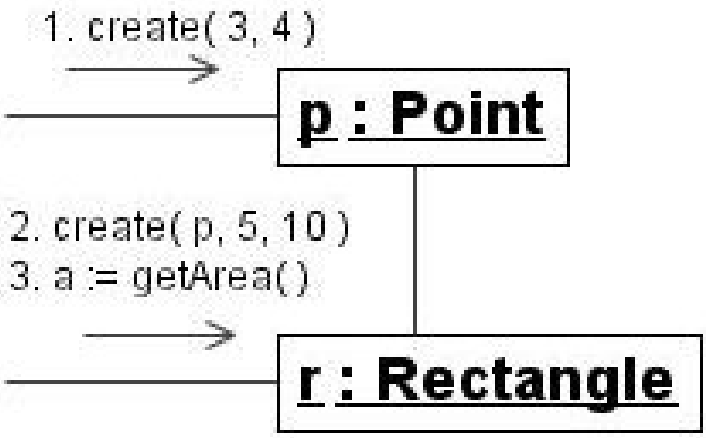
Interaction diagrams show a series of method invocations among a group of objects.

One type of interaction diagram is a *collaboration diagram*, which is essentially an object diagram augmented with method invocations.

Consider the following code:

```
Point p = new Point(3,4);
Rectangle r = new Rectangle(p, 5, 10);
double a = r.getArea();
```

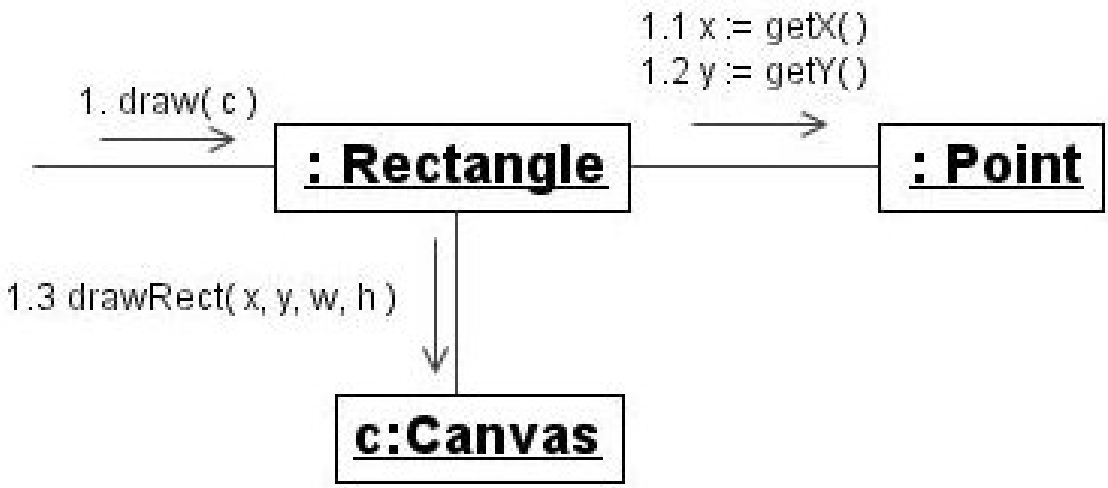
Here is the corresponding collaboration diagram:



Note that a constructor call is depicted by sending a create( ) message to an object that in fact comes into existence as a result of the constructor call.

Collaboration diagrams, continued

Imagine a `draw(Canvas)` method for Rectangle and a `drawRect(x, y, w, h)` method in Canvas. Here is a collaboration diagram that shows the method invocations for drawing a Rectangle:



The corresponding code:

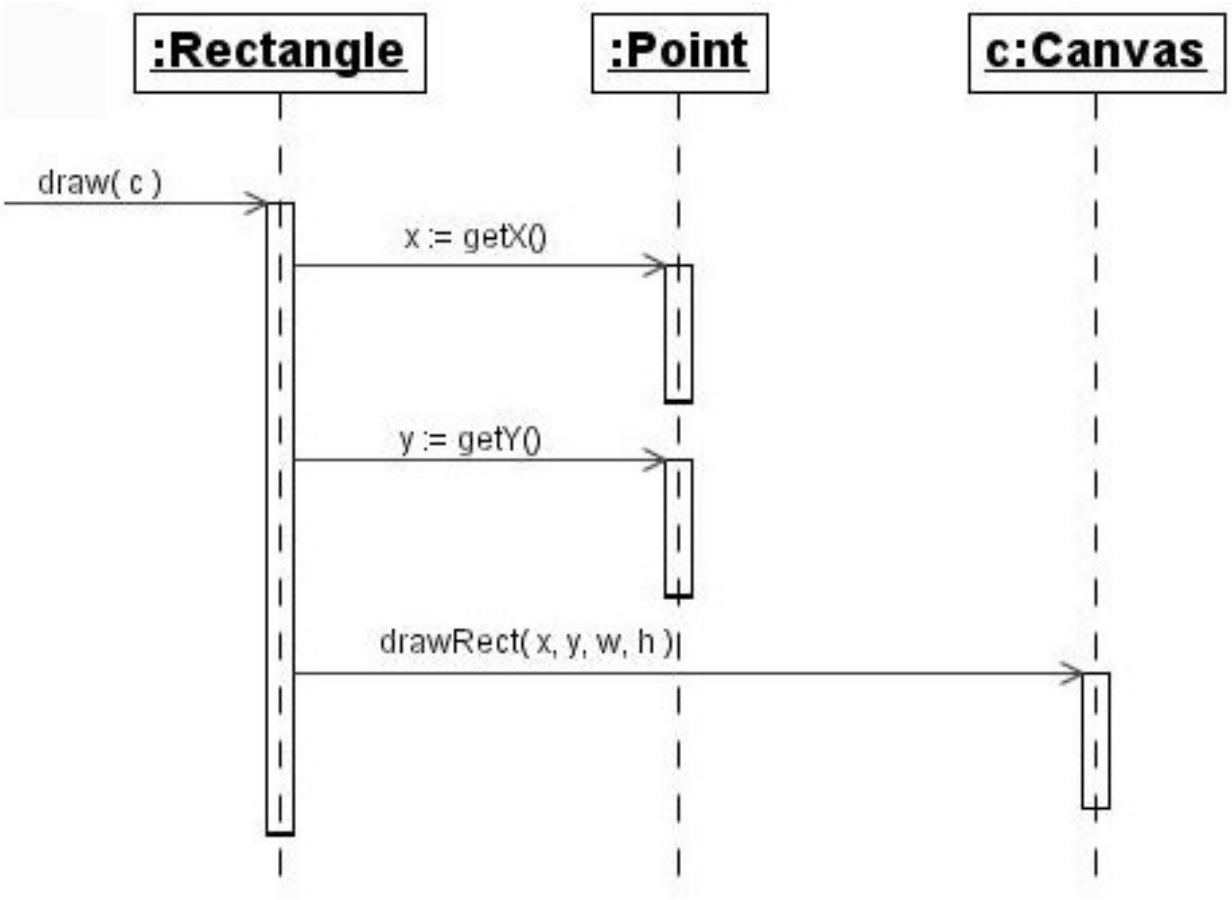
```
class Rectangle {
   ...
   public void draw(Canvas c) {
      double x = ULCorner.getX();
      double y = ULCorner.getY();

      c.drawRect(x, y, width, height);
      }
   ...
   }
```

Sequence diagrams

The other type of UML interaction diagram is the *sequence diagram*.

A sequence diagram presents the same information shown on a collaboration diagram but in a different format.

Here is a sequence diagram for the rectangle drawing scenario:



The dashed vertical lines are *lifelines*.

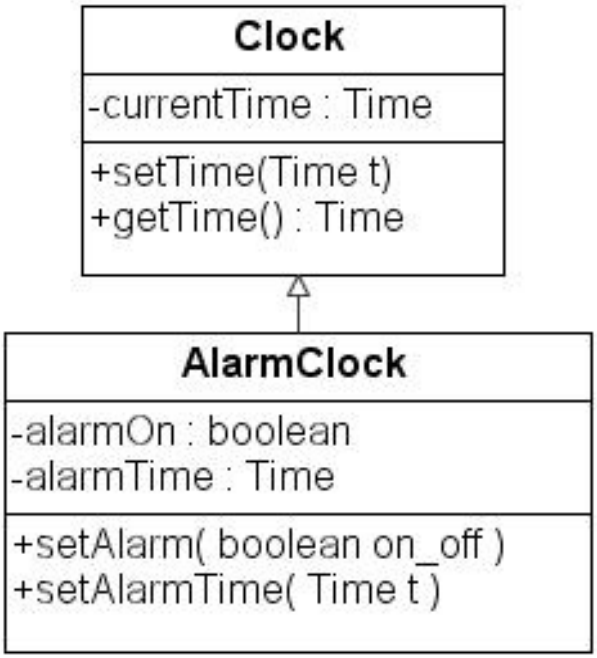
The vertical boxes on the lifelines are *activations*.

Class diagrams—inheritance

A simple example of inheritance in Java:

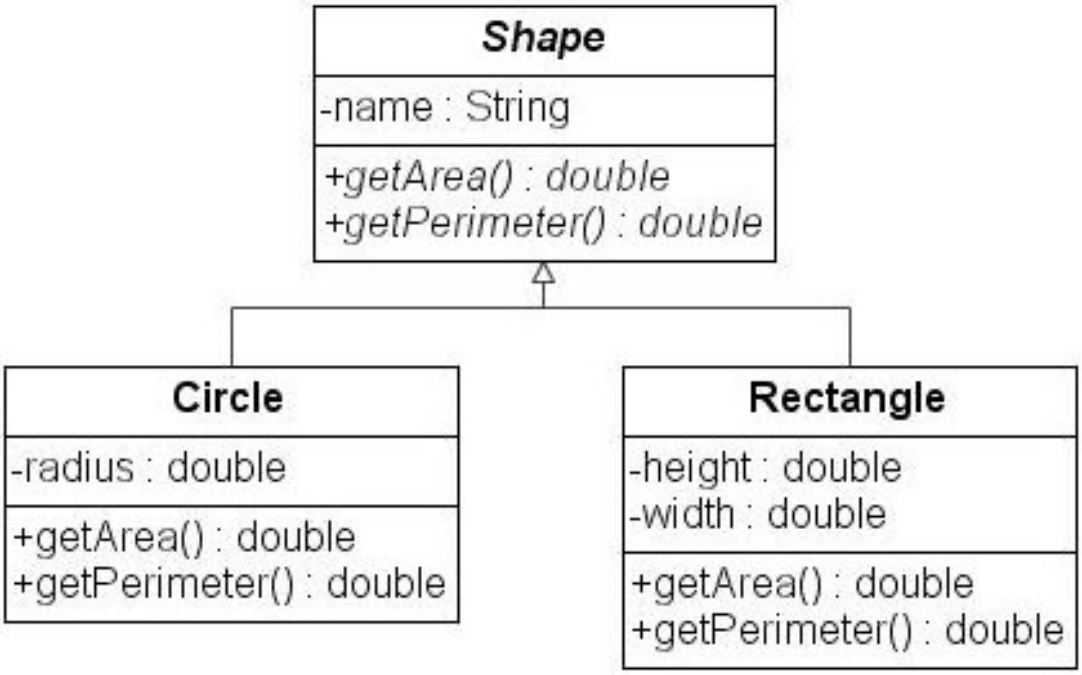
```
class Clock {
    private Time currentTime;
    public void setTime(Time t) { ... }
    public Time getTime() { ... }
    }

class AlarmClock extends Clock {
    private Time alarmTime;
    private boolean alarmOn;
    public void setAlarmTime(Time t) { ... }
    public void setAlarm(boolean on_off) { ... }
    }
```

Expressed in UML:



Inheritance, continued

Abstract classes and methods are indicated as being such by italicizing the name of the class or method:



The code: (Rectangle not shown)

```
abstract class Shape {
    private String name;
    Shape(String name) { ... }
    public abstract double getArea();
    public abstract double getPerimeter();
    }

class Circle extends Shape {
    private double radius;
    Circle(String name, double radius) { ... }
    public double getArea() { ... }
    public double getPerimeter() { ... }
    }
```

If drawing by hand, add text such as "{abstract}" or just "{a}" following the class or method name to indicate an abstract member.

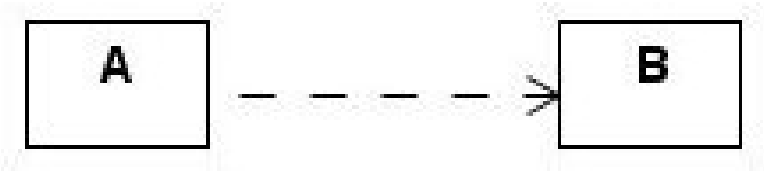
Details on dependency and association

If a change in a class B may affect a class A, then it is said that A depends on B.

In Java, creating an instance of a class, or passing a class instance as a parameter creates dependency:

```
class A {
   void f( ) {
      ...
      B b = new B();
      ...use methods in B...
      }

   void g(B b) {
      ...use methods in B...
      }
   }
```

Class A depends on class B:



In Booch's notation, it would be said that A uses-a B.

Two typical implications of class A depending on class B:

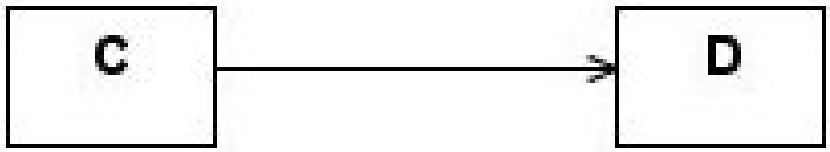
- A definition of B, such as a C++ header file or a Java class file, is required to compile A.
- A change in B requires A to be rebuilt.

Details on dependency and association, cont.

An association is unidirectional if a class C has an attribute of class type D, but D makes no use of C:

```
class C {
   ...
   private D d1;
   }

class D {
   ...no usage of C...
   }
```

This diagram shows that there is navigation (visibility) from C to D, but not from D to C:



Association implies dependency.

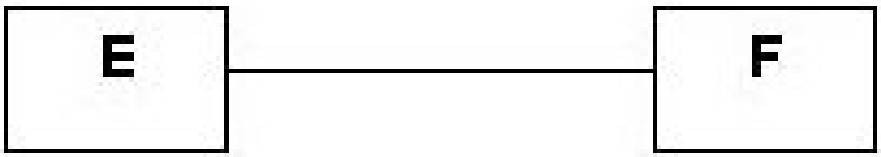
In Booch's notation it would be said that C has-a D.

Details on dependency and association, cont.

If two classes have attributes referencing objects of each other's type, a bidirectional association exists:

```
class E {
    ...
    private F f1;
    }

class F {
    ...
    private E e1;
    }
```

A bidirectional association, with navigability from each class to the other, is shown by a line with no arrowheads:



Example: InfoMagic

Imagine a simple command line tool, called InfoMagic, to manage "notes"—textual information about topics of interest to the user. Notes are stored in a filesystem-like hierarchy where folders may contain any number of notes and any number of folders.

Users may enter one of several commands at the prompt:

`new folder <name>`
: Make a new folder in the current folder.

`new note`
: Create a new note and add it to the list of notes in the current folder.

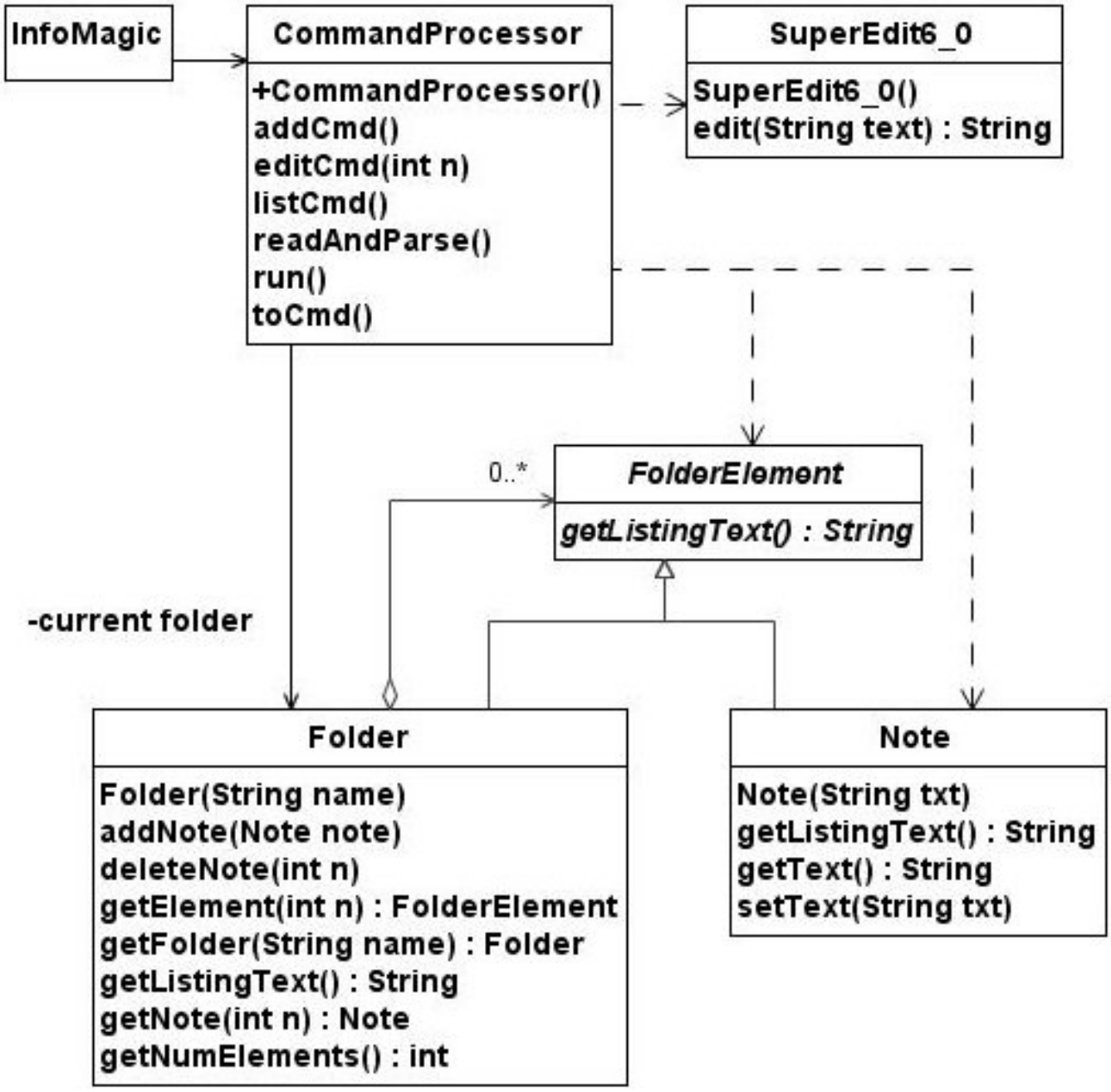
`list`
: Display a list of notes and folders in the current folder. Notes are displayed with a sequence number but that number is not associated with the note itself.

`edit <note#>`
: Edit the text of the specified note using a text editor.

`delete <note#>`
: Delete the specified note

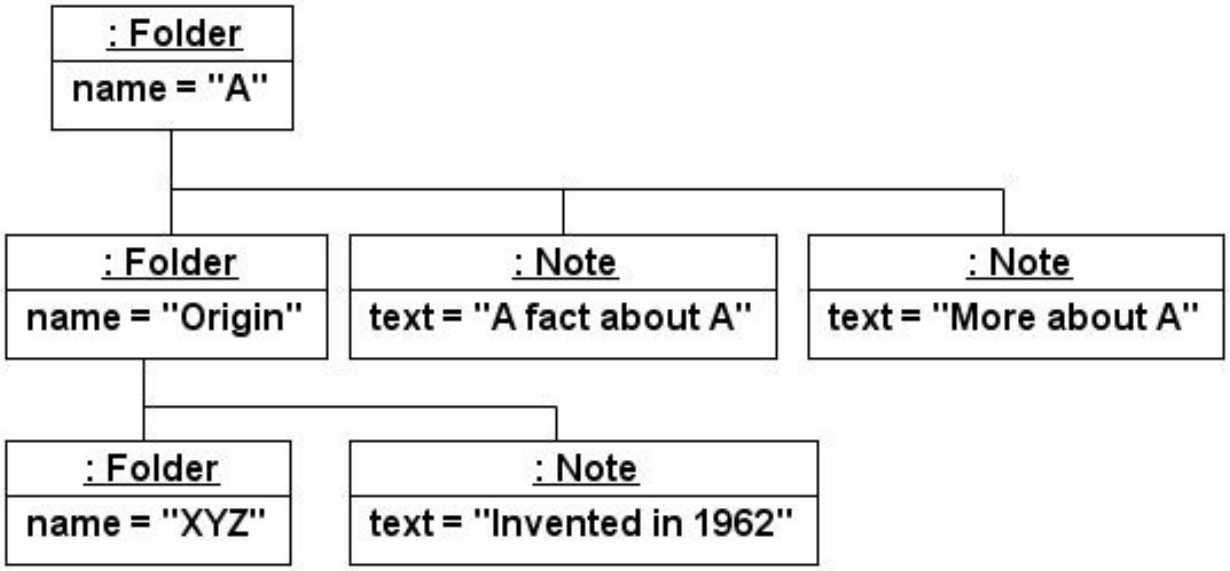
`to <folder>[/<folder>/...]`
: Change to a new folder using a UNIX-style path specification

InfoMagic: Class diagram

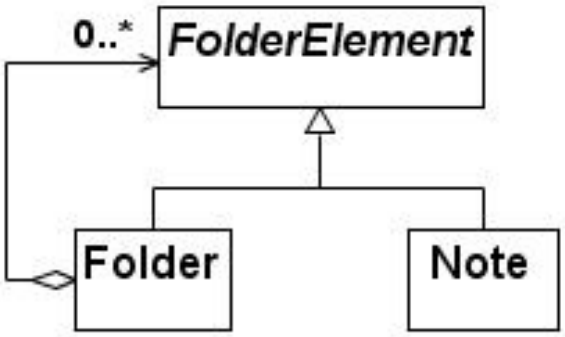


InfoMagic: Object diagram

Here is an object diagram that shows a potential collection of Folders and Notes:

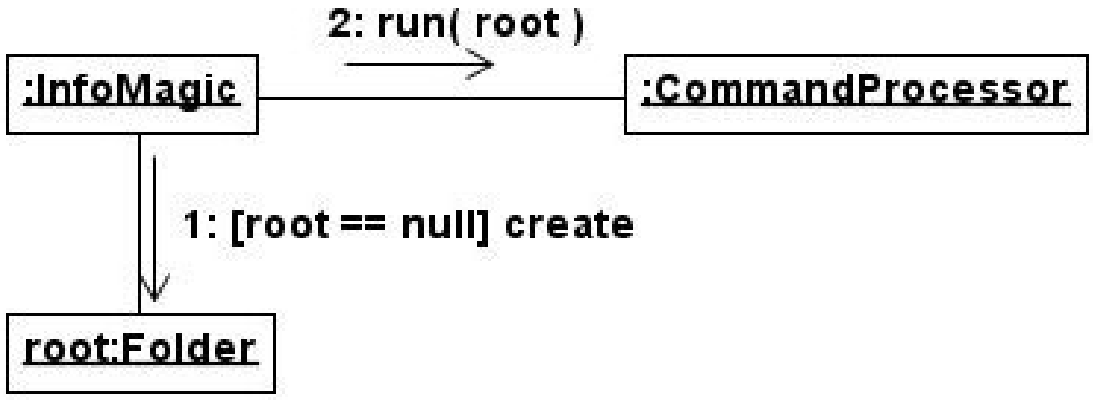


Recall this relationship:

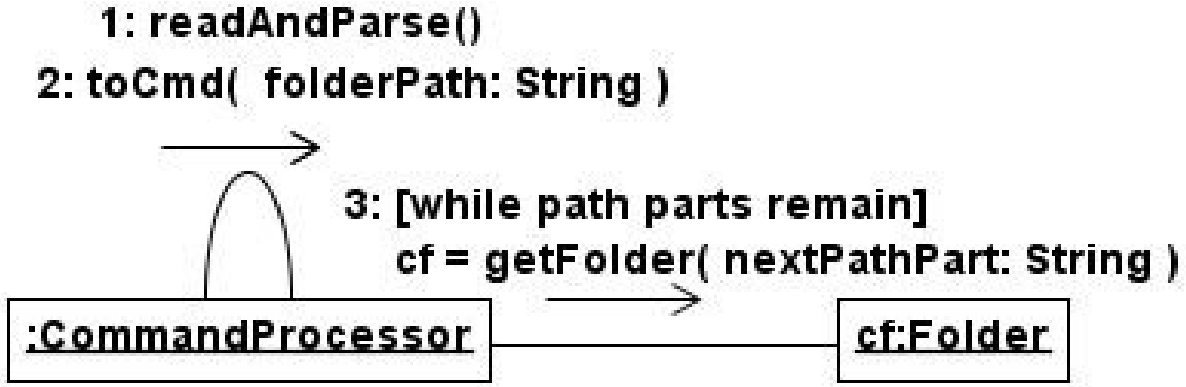


InfoMagic: Interaction diagrams

Collaboration diagram for "Program start-up":

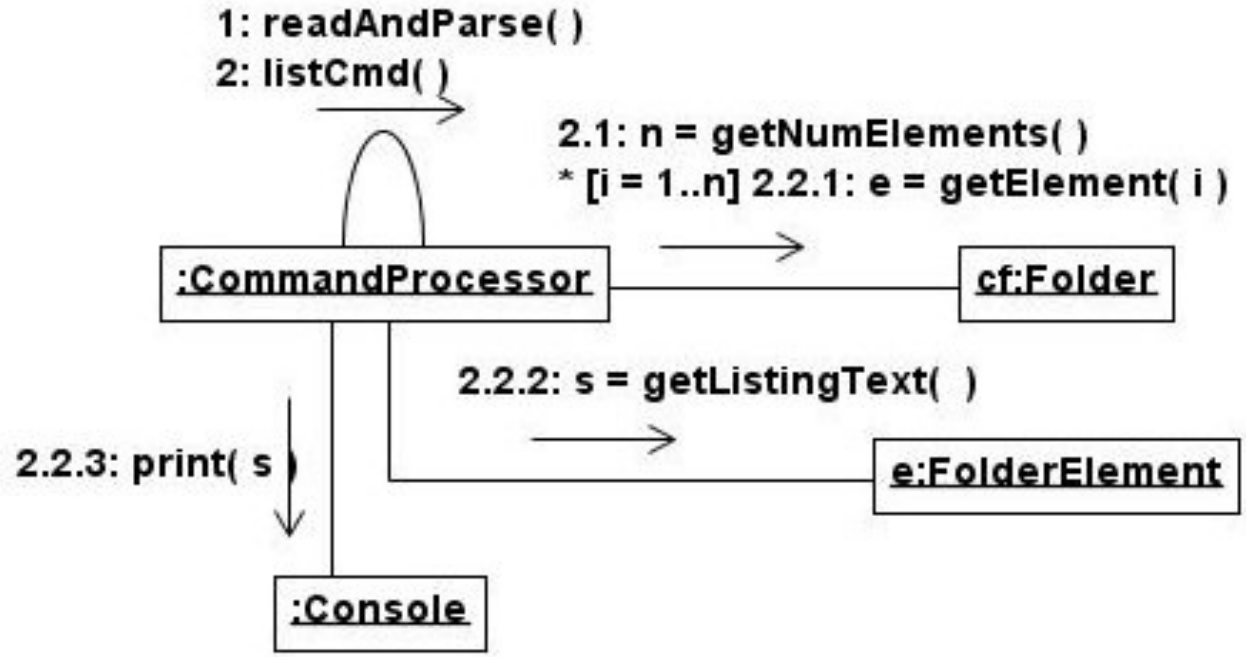


Collaboration diagram for "`to folder/folder/...`":

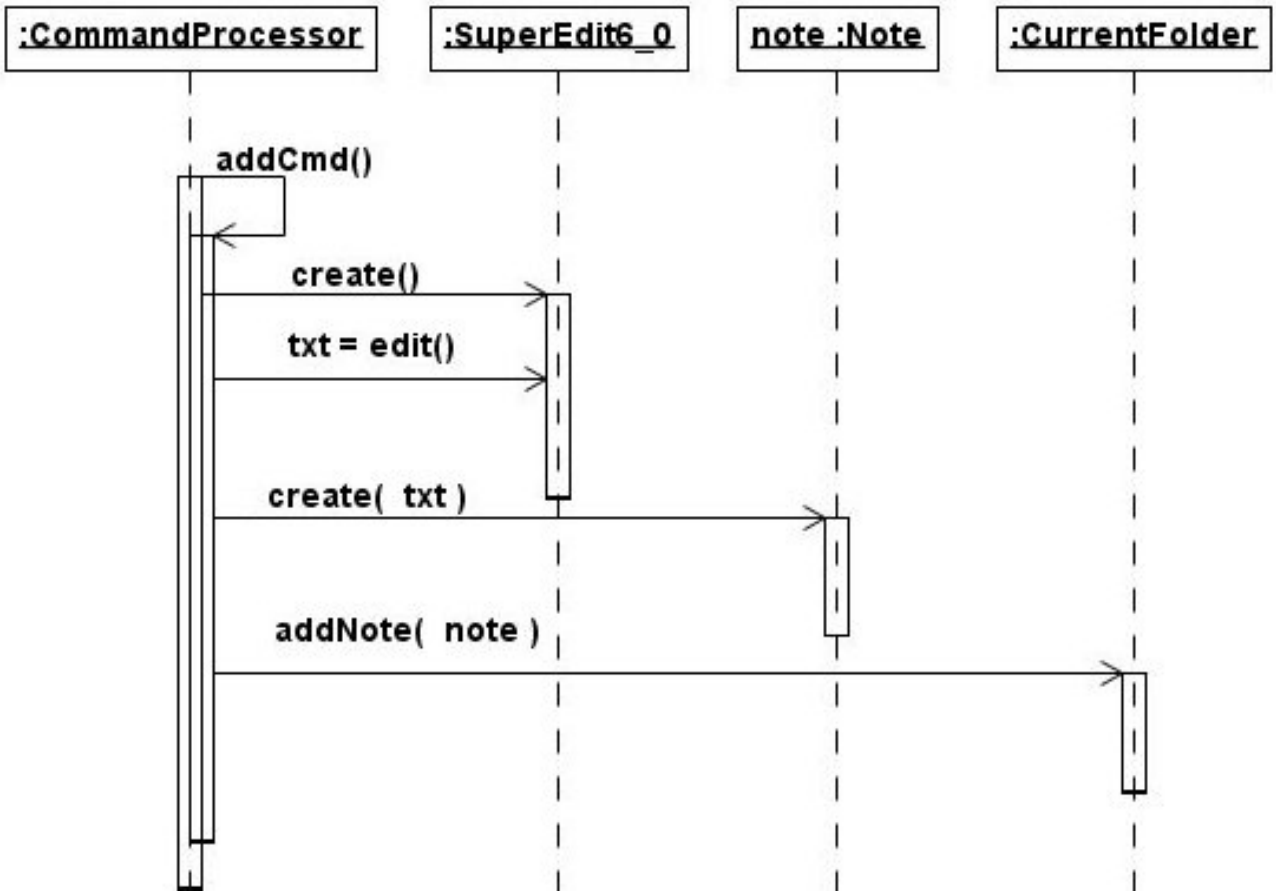


InfoMagic: interaction diagrams, continued

Collaboration diagram for "`list`":

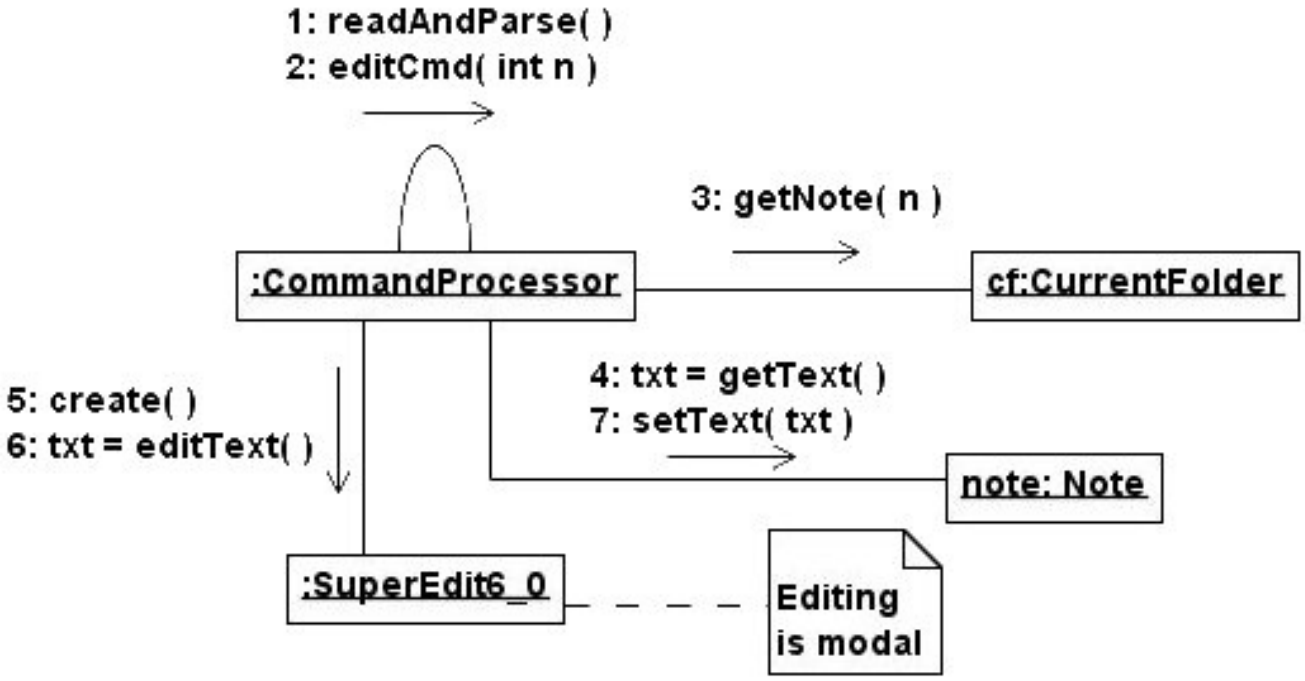


Sequence diagram for "`add`":



InfoMagic: interaction diagrams, continued

Collaboration diagram for "`edit`":



O-O design with UML—"Elevator Training"

Here's a featherweight object-oriented design method based on UML:

1. For each system operation identify pertinent objects and devise an interaction between them, sketching it as a collaboration diagram.

2. Derive a class diagram from the set of collaboration diagrams.

3. Analyze the relationships shown in the class diagram and refine the design, repeating steps 1, 2, and 3, as necessary.

4. Code it!

Effective use of UML

- Take time to understand the notation but don't make a career of it.
- Keep in mind the purpose of a particular diagram and use appropriate detail for that purpose.
- Beware of arguments about notional choice, such as aggregation versus composition, that are ultimately inconsequential.
- Don't fill design documents with unnecessary diagrams.
- Don't get carried away with drawing tools.
- Customers usually aren't too interested in a beautiful set of diagrams. What they want is good software.

Recommended reading

If you're going to buy just one book on UML...

> *UML and the Unified Process*, by Jim Arlow and Ila Neustadt. 2002. Published by Addison-Wesley. ISBN 0-201-77060-1.

A pretty good reference but a little disappointing considering the authors...

> *The Unified Modeling Language User Guide*, by Grady Booch et al. 1999. Published by Addison-Wesley. ISBN 0-201-57168-4.

Concise, but a little bit pricey for the size...

> *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, 2nd Edition, by Martin Fowler and Kendall Scott. 1999. Published by Addison-Wesley. ISBN 0-201-65783-X

A great book on object-oriented design that makes extensive use of UML...

> *Applying UML and Patterns*, by Craig Larman. 2002. Published by Prentice Hall PTR. ISBN 0-13-092529-1