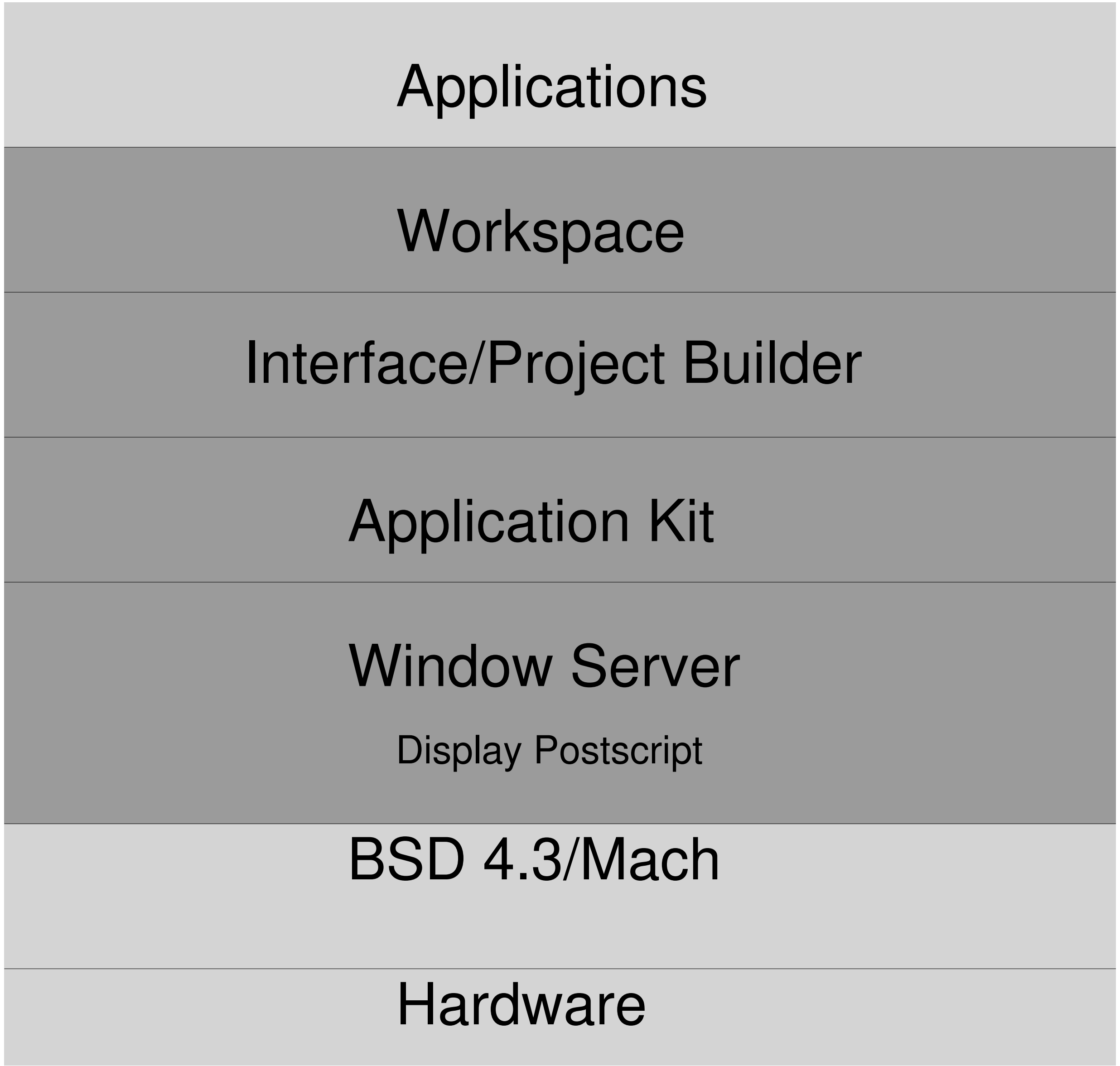
The Application Kit

NeXTSTEP

NeXTSTEP is NeXT's application development and user environment, consisting of the ***Workspace Manager***, ***Interface Builder (& Project Builder)***, ***Application Kit***, and ***Window Server***.

NextStep


Applications
Workspace
Interface/Project Builder
Application Kit
Window Server
Display Postscript
BSD 4.3/Mach
Hardware

What is the Application Kit?

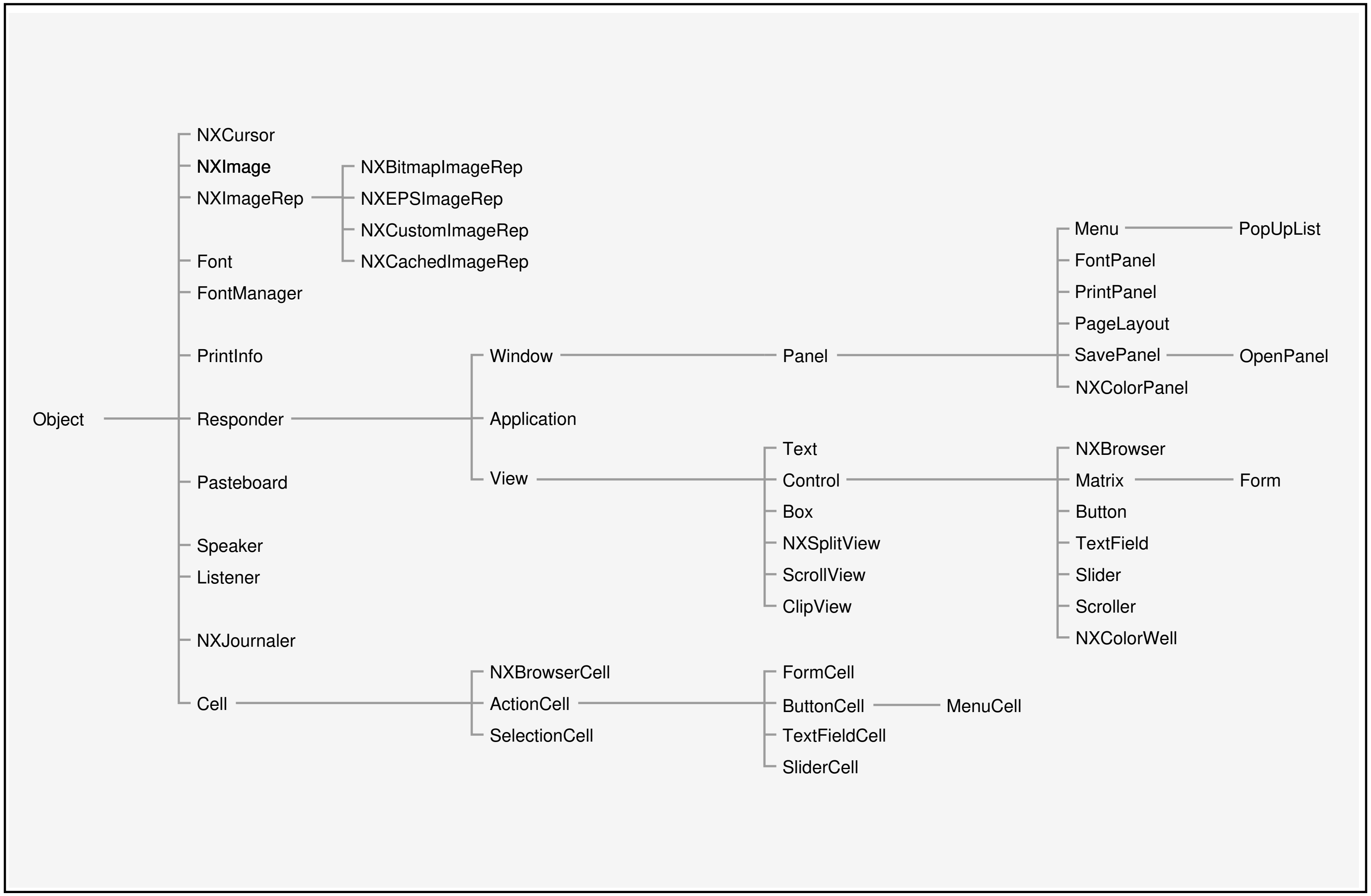
The Application Kit:

Provides a ***basic program structure*** for applications that draw on the screen and respond to events.

Implements the ***NeXT user interface*** and relieves you of the more tedious programming tasks.

Provides a rich ***library of objects*** (classes) for getting user input and a uniform way of interacting with those objects.

Application Kit Inheritance Hierarchy



Using AppKit Classes

You can make use of AppKit classes four ways:

1. You can create ***objects (instances)*** belonging to the classes.

2. You can define ***subclasses*** of AppKit classes, then create instances of those for your application.

3. You can define a ***category*** that extends the original definition of a class.

4. You can define ***delegate*** objects that act on behalf of objects that inherit from the AppKit.

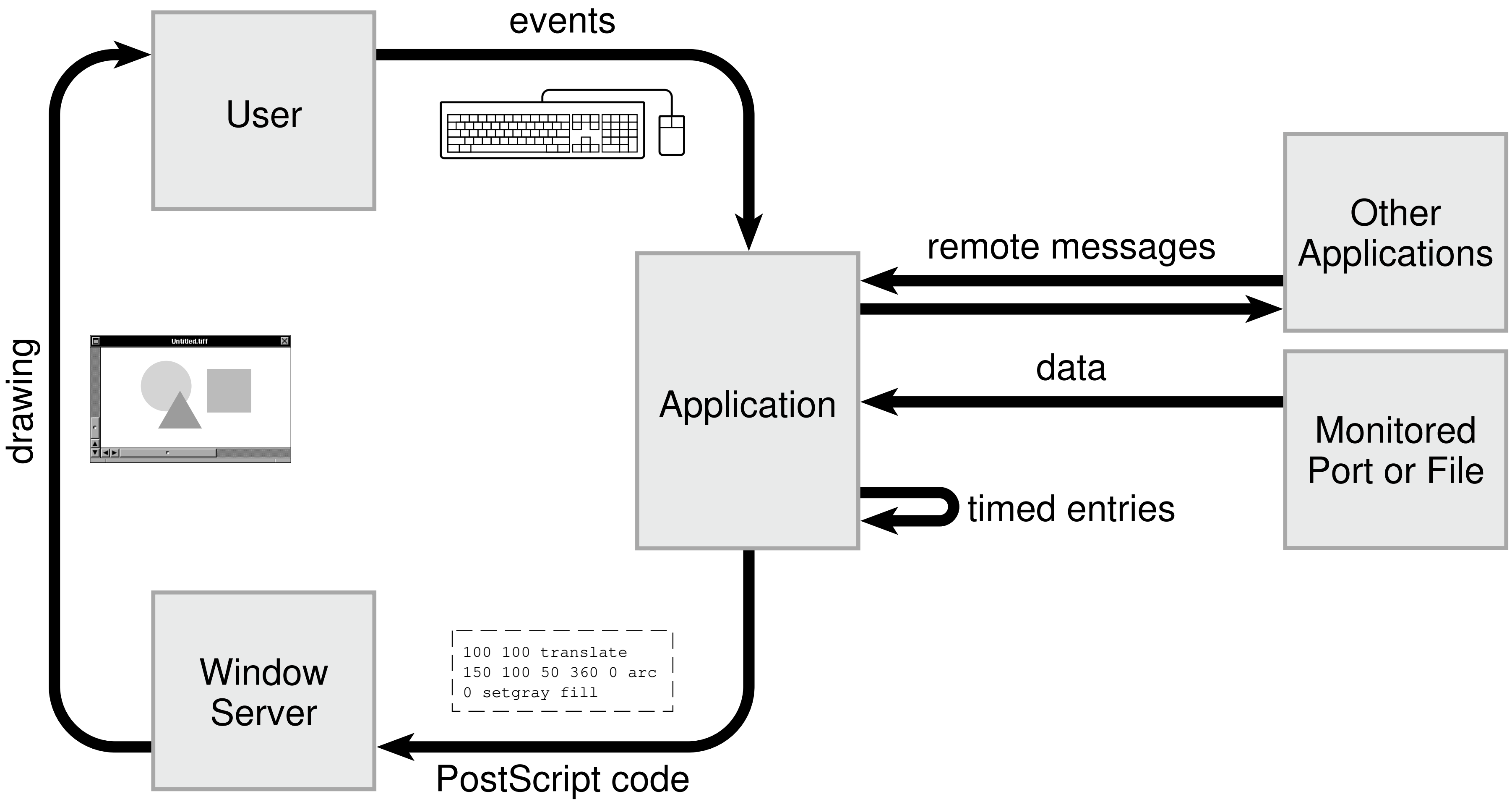
Structure of an Application

The NeXT interface is ***event-driven***.

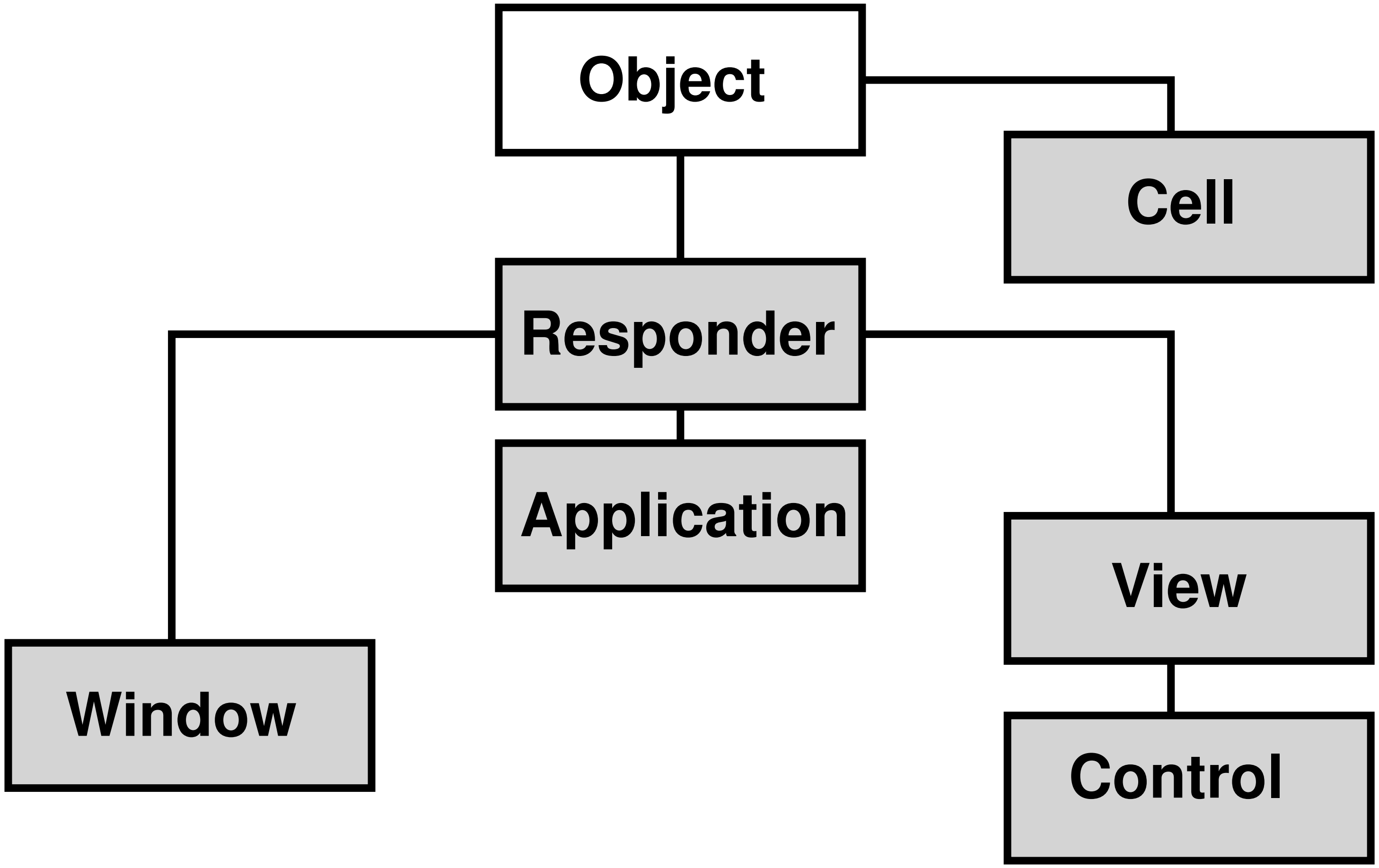
The Application Kit provides the ***main event loop*** and automatically dispatches events to the appropriate object.

Every application has one ***Application object*** that gets events from the ***Window Server*** and oversees the application's ***windows***.

The Event Cycle



Principal Application Kit Objects



Responder

An ***abstract superclass*** for classes that respond to keyboard and mouse events.

Responders participate in a linked-list of event-handling objects called a ***responder chain***.

If an object in the chain can't handle a message indicating an event, the message is passed on to its ***next responder***.

The ***Responder class*** defines the elements essential to the responder chain, a **`nextResponder`** instance variable, and the methods for passing messages from one object to another.

View

The ***View class*** is an ***abstract superclass*** which provides a structure for drawing on the screen and for handling mouse and keyboard events.

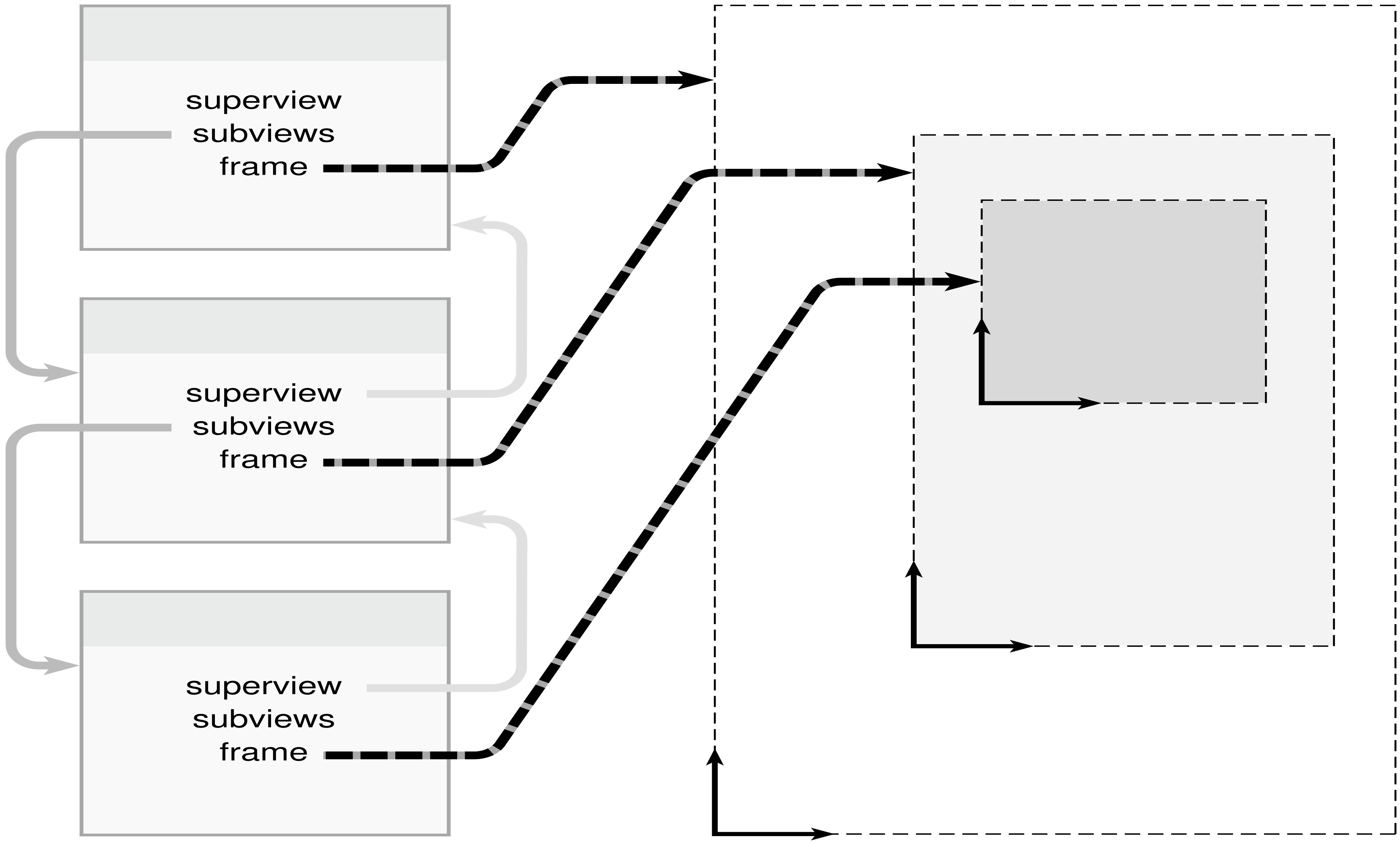
All ***graphical objects*** inherit from View.

The View's ***frame rectangle*** defines the boundaries of the View, the tablet on which it can draw.

Every View object is associated with a particular ***window***.

Views within a window are linked together in a ***view hierarchy***.

View Hierarchy

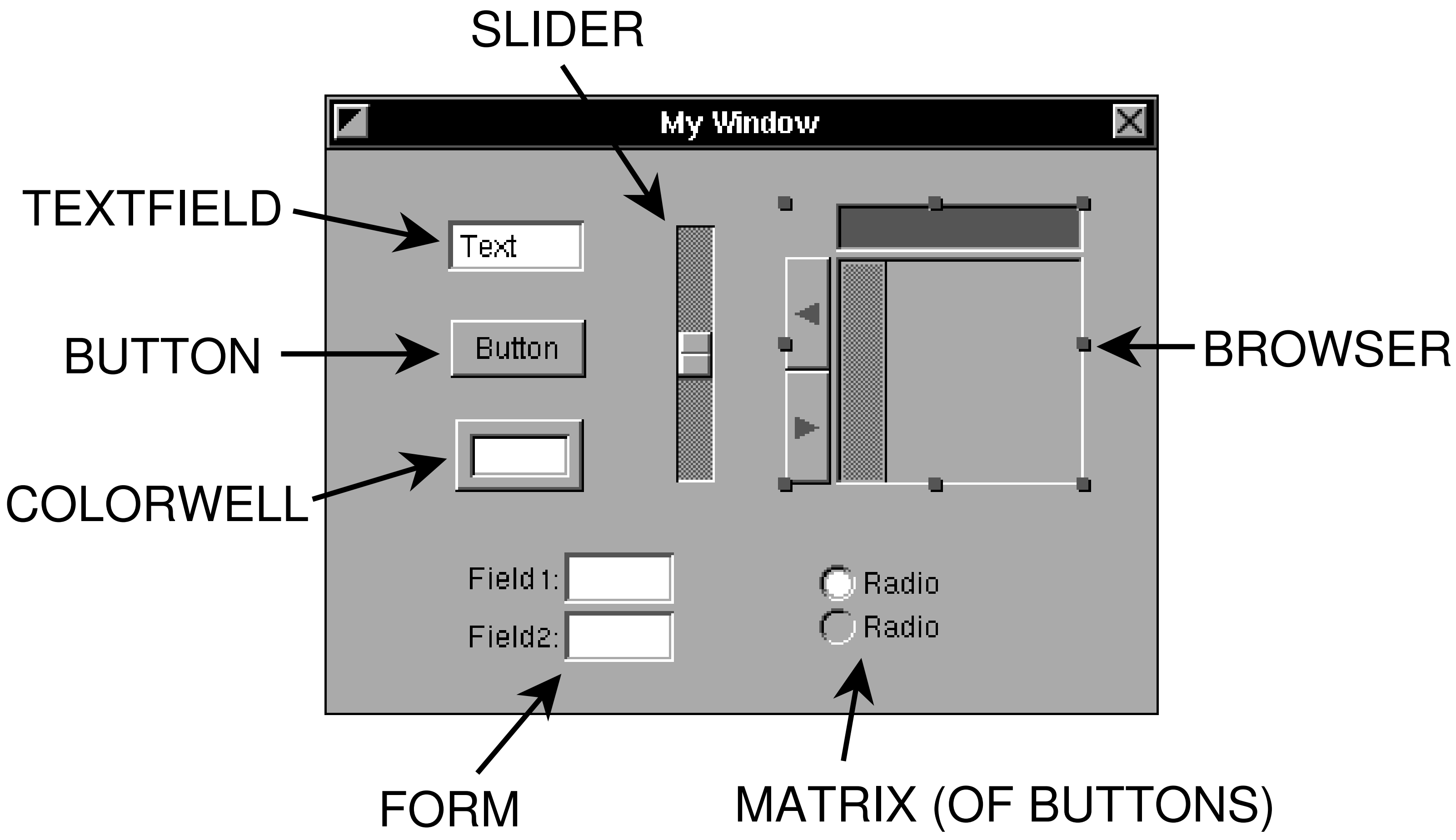


Control

A subclass of View that provides an ***abstract superclass*** for objects that receive mouse and keyboard events and translate them into application-specific messages for other objects.

A Control’s job is to interpret the user’s ***mouse and keyboard actions*** and ask another object to respond to them.

Control Examples:



Cell

A Cell is an object that can draw within a View and handle events that are passed to it from the View.

Cells are a place for putting much of the work associated with a View.

Most Controls are built around Cells. The Control (a View) provides the rectangle for drawing and receives the mouse events. The Cell does the actual work of drawing in the View and responding to the events.

For example, Buttons have ButtonCells, Sliders have SliderCells. Or there can be a Matrix of ButtonCells.

Window

Every window is managed by a ***Window object***.

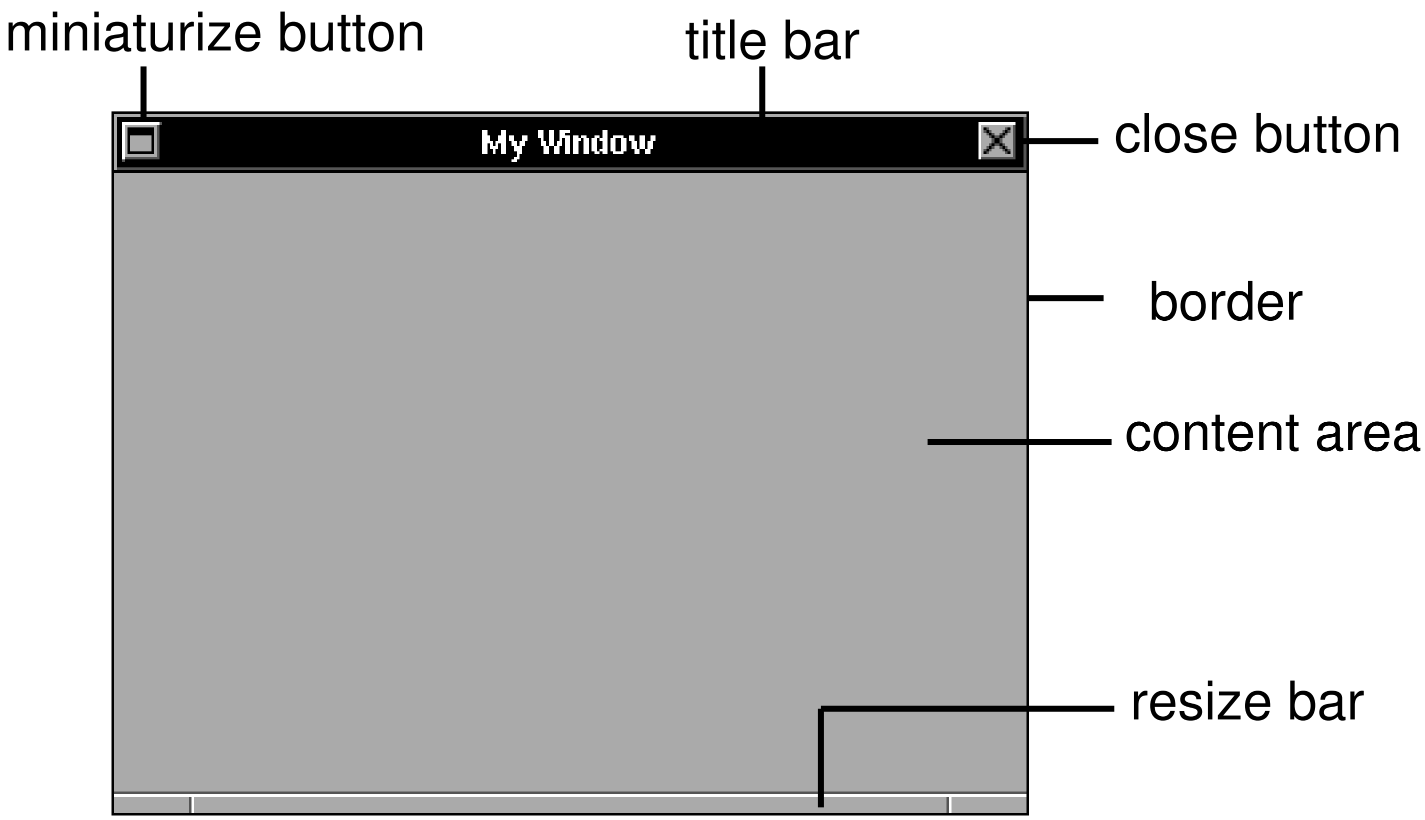
Every Window has a view hierarchy with at least two views: a ***frame view*** and a ***content view***.

The frame view fills the frame rectangle and draws the Window's border, title bar, and resize bar.

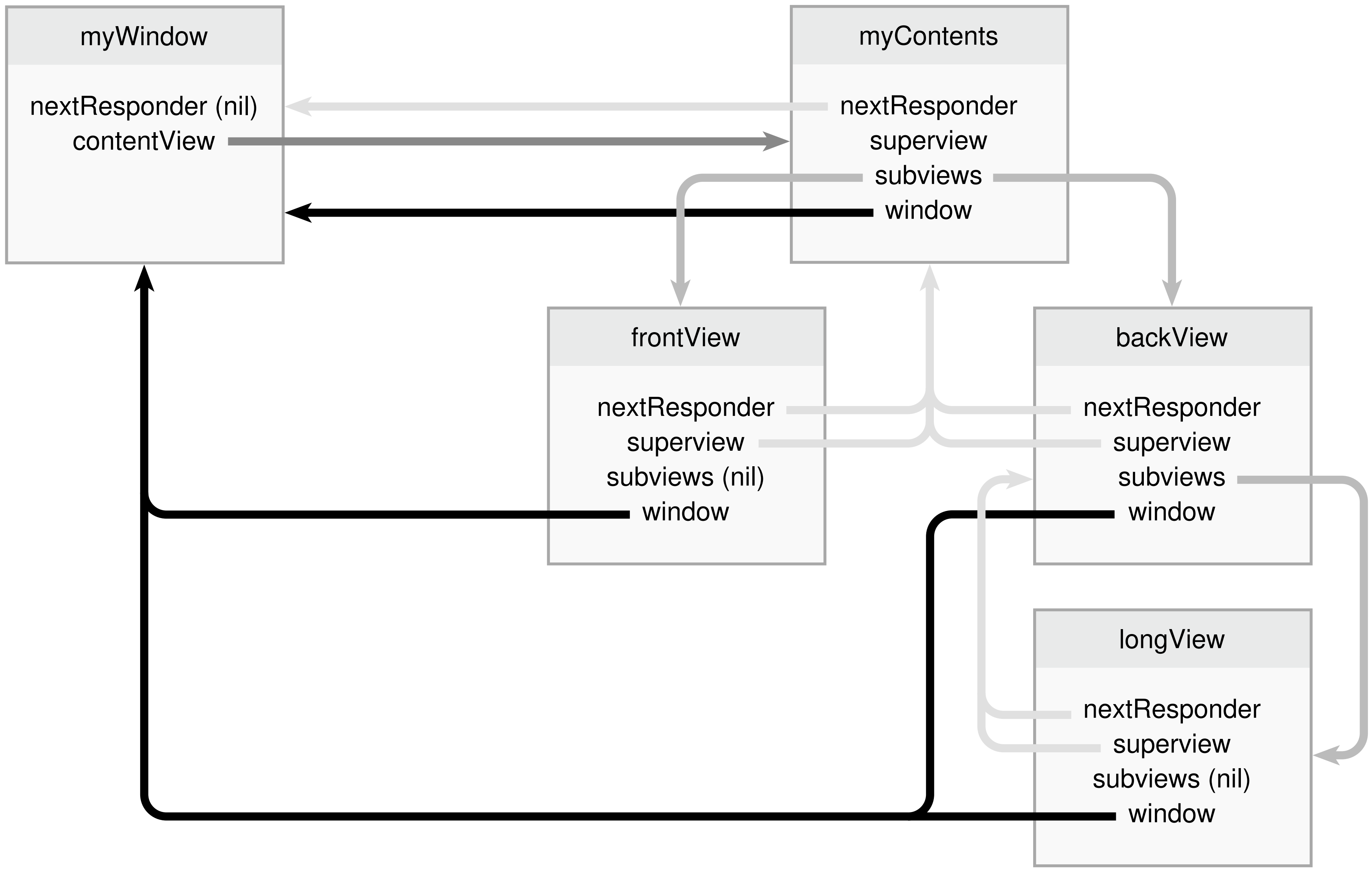
The content view is a subview of the frame view and is the highest level view for drawing or installing other subviews.

The Window oversees all of its Views.

Here is a typical window:



Application Kit



Application

Every program has one ***Application object*** which:

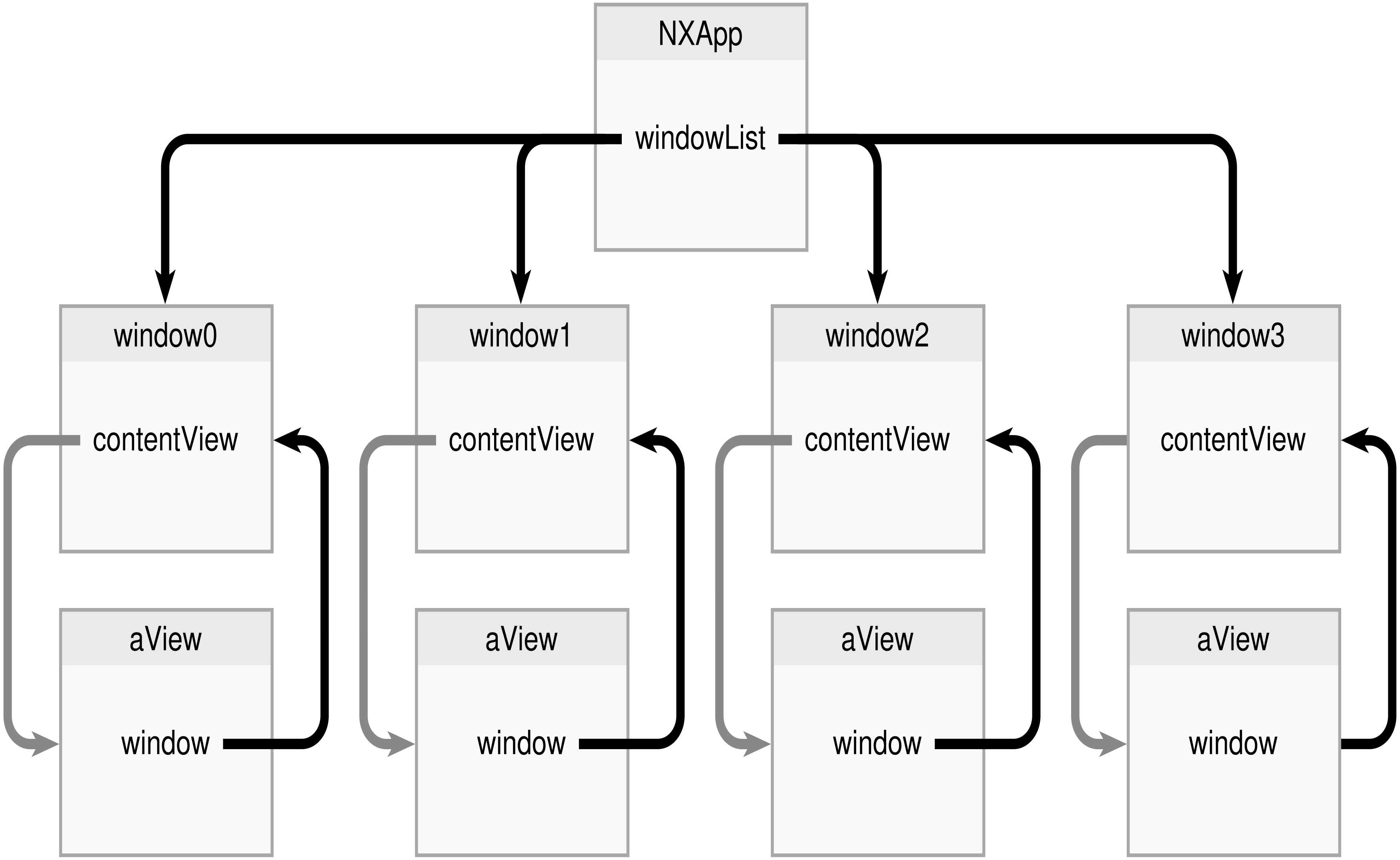
Receives events from the ***Window Server*** and distributes them to other objects.

Manages the application’s Windows.

Keeps global information that’s shared by other objects.

NXApp is the global variable which points to your Application object.

Application Kit



Events

Events are things like ***mouseUp, mouseDown, mouseDragged,*** etc.

Events are ***dispatched as messages***.

Usually, one object (e.g., a control) responds to the event, then notifies another object that the event occurred.

Two types of messages get sent depending on the object that receives the initial message: ***action*** and ***notification***.

The Event Process

1. The Window Server sends mouse, keyboard, and machine events to the Application object.

2. Within the application, the AppKit dispatches event messages to the appropriate object.

3. The object responds.

The Event Process: A Closer Look

Window Server sends all events to the Application object.

- The Application object handles ***machine events*** directly (e.g., power off).

- If a ***window event***, the Application object sends a ***window event message*** to the appropriate window (e.g., close window).

- Otherwise, it sends an event message to the appropriate window for ***dispatch***.

The Event Process: A Closer Look

(continued)

The window dispatches ***mouse event messages*** to the appropriate view in the Window.

`mouseDown:` to the deepest View underneath the mouse.

`mouseUp:` or **`mouseDragged:`** to the view which initially received **`mouseDown:`**.

`mouseEntered:` or **`mouseExited:`** to the object which "owns" the appropriate tracking rectangle.

keyboard and **`mouseMoved:`** event messages are dispatched to the window's ***firstResponder***.

How Objects Respond

- Do nothing but pass event message on to its nextResponder.

  By default, nextResponder is the object's superview.

  This is the default behavior for Views.

- Perform object-specific action.

  E.g., Text objects display characters corresponding to keystrokes.

How Objects Respond *(continued)*

- Begin a ***modal loop***

E.g., a Button object highlights on a **`mouseDown:`** *message and enters a modal loop waiting for a* ***mouseUp*** *event.*

- Respond to the event and send a message to another object notifying it of the event.

Two notification paradigms are used here:

*1. **Target-Action** is used by Controls.*

*2. **Delegation-Notification** is used by Window, Application, and Text.*

Target-Action

The ***control*** object gets an event message and handles the event;

e.g., after a **`mouseDown:`** *message a Button object highlights and waits for a mouseUp event.*

When handling is complete, the control notifies a target object by invoking an action method owned by the target.

In effect, a control object receiving an *event message* translates it into an *action message* to a target object.

Target-Action *(continued)*

Action messages to the target from the sender are always of the special form:

```
[target actionMethodName:self]
```

The target object must have an action method of the form:

```
-actionMethodName:sender{
        <code to take action>
        return self;
}
```

(**`sender`** will contain the **`id`** of the sending object.)

Examples of Action Messages

The Application Kit defines a number of action messages to which specific objects in the kit will respond. A few examples are below. These objects ***respond to*** these action messages:

Application

hide:	Hide application’s windows
unhide:	Unhide application’s windows
terminate:	Terminates the application

Button

performClick:	Simulates clicking button

Control & Cell

takeIntValueFrom:	Get value from sender & set own value
takeStringValueFrom:	Get value from sender & set own value
takeFloatValueFrom:	Get value from sender & set own value
takeDoubleValueFrom:	Get value from sender & set own value

Text

cut:	Cut selection & put in pasteboard
copy:	Copy selectiont to pasteboard
paste:	Paste contents of pasteboard
delete:	Delete selection, not to pasteboard

View

printPSCode:	Print view and its subviews

Window

orderFront:	Brings window to front
orderOut:	Remove the window from the screen
performClose:	Simulate clicking the close button
printPSCode:	Prints all of the view within the window

Of course, you can send these from your own custom object. For example, order your info panel to the front:

```
[myInfoPanel orderFront:self]
```

Action Methods

When a target object's method responds to an action message, it must take some action. This action might be any of:

1. **Simple Action** - Simply do something.

2. **Query the Control** - Ask the control for information, then do something with it.

3. **Query a Matrix** - The control is in a matrix which must be queried for more information.

Action Methods *(continued)*

1. Simple Action

In the simplest case, as when responding to a button, the method simply takes an action without communicating with the *sending* control object:

```
-clear:sender  //msg from a button
{
    [display setFloatValue:0.0];
    return self;
}
```

Action Methods *(continued)*

2. Query the Control

Sometimes it is necessary to query the **sender** for more information. For example, a method responding to a message from a slider might need to ask the slider for its value:

```
-takeVolumeFrom:sender //msg from a slider
{
    volume=[sender floatValue];
    return self;
}
```

The **`id`** of the sending control (slider) is in **`sender.`**

Action Methods *(continued)*

3. Query a Matrix

When the control is a **Matrix**, it can be asked for the **`id`** of the ***cell*** in the matrix which initiated the message. That cell can then be asked for its ***tag*** (a label integer). For example, a keypad matrix of buttons 0 to 9, where the tags of the buttoncells are also set to 0 to 9:

```
-digit:sender  //msg from a button matrix
{
    nextDigit=[[sender selectedCell] tag];
           //or:  =[sender selectedTag];
    <do something with it>
    return self;
}
```

Action Methods *(continued)*

3. Query a Matrix *(continued)*

Form is a subclass of **Matrix**. The **FormCell**s in the Form are individually selectable. For example, the Form is asked for the **index** in the Form where data was entered, then the data at that index is obtained:

```
-dataIn:sender  //msg from a form
{
     index=[sender selectedIndex];
     dataEntered=[sender intValueAt:index];
     <do something with it>
     return self;
}
```

Delegation and Notification

An object in AppKit classes such as Window, Text, and Application will *"delegate"* some of its responsibility for handling an event to another object called its ***delegate*** by *"notifying"* the delegate that the sender's state has either changed or is about to change.

A delegate can choose to:

Ignore the notification.

Do additional processing in response.

Possibly block the change that resulted in the sender sending the notification.

Delegation and Notification
(continued)

Why use delegates?

Extend default handling of events by AppKit objects without subclassing.

Centralizes custom behavior in one place (i.e., the delegate).

Delegates can serve multiple clients and thus provide a natural way of keeping track of the status of things.

Delegation and Notification

(continued)

Delegation-notification is <u>similar</u> to the target-action paradigm in that...

The delegate, like the target, is an object and must be explicitly set.

An object can have only one delegate at a time.

A notification message is sent in response to some event or change.

Delegation and Notification
(continued)

Delegation-notification is <u>different</u> from the target-action paradigm in that...

Notification messages are pre-defined by the sender’s class definition. The delegate implements methods only for those it chooses to respond to.

An object may send a different notification message depending on what has occurred.

If no delegate is set, no notification messages are sent.

Delegation and Notification
(continued)

Notification messages are only sent if...

The delegate has been set in IB or by message:

```
[myWindow setDelegate:myDelegate]
```

and the delegate has implemented a method of the same name:

```
/*in class implementation for
myDelegate*/
-windowWillClose:(id)theWindow {
...}
```

Delegation and Notification

(continued)

A delegate need only implement methods for those notification messages to which it cares to respond.

When a delegate has not implemented the corresponding method for a particular notification message, the sender does not send the message.

For many AppKit notification messages, the delegate can block a proposed change by returning **nil**. If the delegate does not implement the corresponding method, the change will occur by default.

Example:

The code in the object sending the notification message (in this example, the Window object) would look something like this:

```
if(delegate &&[delegate
respondsTo:@selector(windowWillClose:)])
    [delegate windowWillClose:self];
```

For Window’s **`windowWillClose:`** notification message, if the delegate returns **nil**, the window will not close (the code above does not reflect that).

Examples of Notification Messages

Several AppKit objects have pre-defined notification messages for changes in state for which custom behavior may be desired. A few examples are below. These objects ***send*** these messages to their delegate:

Application

appDidInit:
appDidAwake:
appDidBecomeActive:
appDidHide:

Window

windowWillClose:
windowWillResize:toSize:
windowDidResize:
windowDidMove:
windowDidExpose:
windowDidBecomeKey:
windowDidBecomeMain:
windowDidMiniaturize:
windowDidUpdate:
windowWillMiniaturize:toMiniwindow:

Text

textWillChange:
textWillResize:
textWillEnd:
textDidResize:oldBounds:invalid:
textDidChange:
textDidEnd:endChar:
text:isEmpty:
textDidRead:paperSize:
textWillConvert:fromFont:toFont:
textWillFinishReadingRichText:stream:atPosition
textWillSetSel:toFont:
textWillStartReadingRichText:
textWillWrite:paperSize:

First Responder

keyDown events are dispatched to the frontmost window willing to accept keyDown events:

Determined by window’s ***eventMask***.

Default of a titled window is to accept ***keyDown*** events.

Within a window, keyDown events are dispatched to the window’s firstResponder.

First Responder (continued)

Every window has a firstResponder.

Must have ***Responder*** as an ancestor class.

It responds to **`keyDown:`** message when its window is the ***keyWindow***.

The firstResponder in the keyWindow is automatically sent all action messages for whom no specific target was specified.

If the firstResponder cannot take action, the events are passed up a ***responder chain***.

First Responder (continued)

Two ways to become a firstResponder

1. On ***mouseDown*** event, the window asks a view object if it wants to become firstResponder via:

```
if([myView acceptsFirstResponder])...;
```

Default View method returns NO; override default if you want the view to respond to keyboard messages.

```
-(BOOL)acceptsFirstResponder
      {return YES;}
```

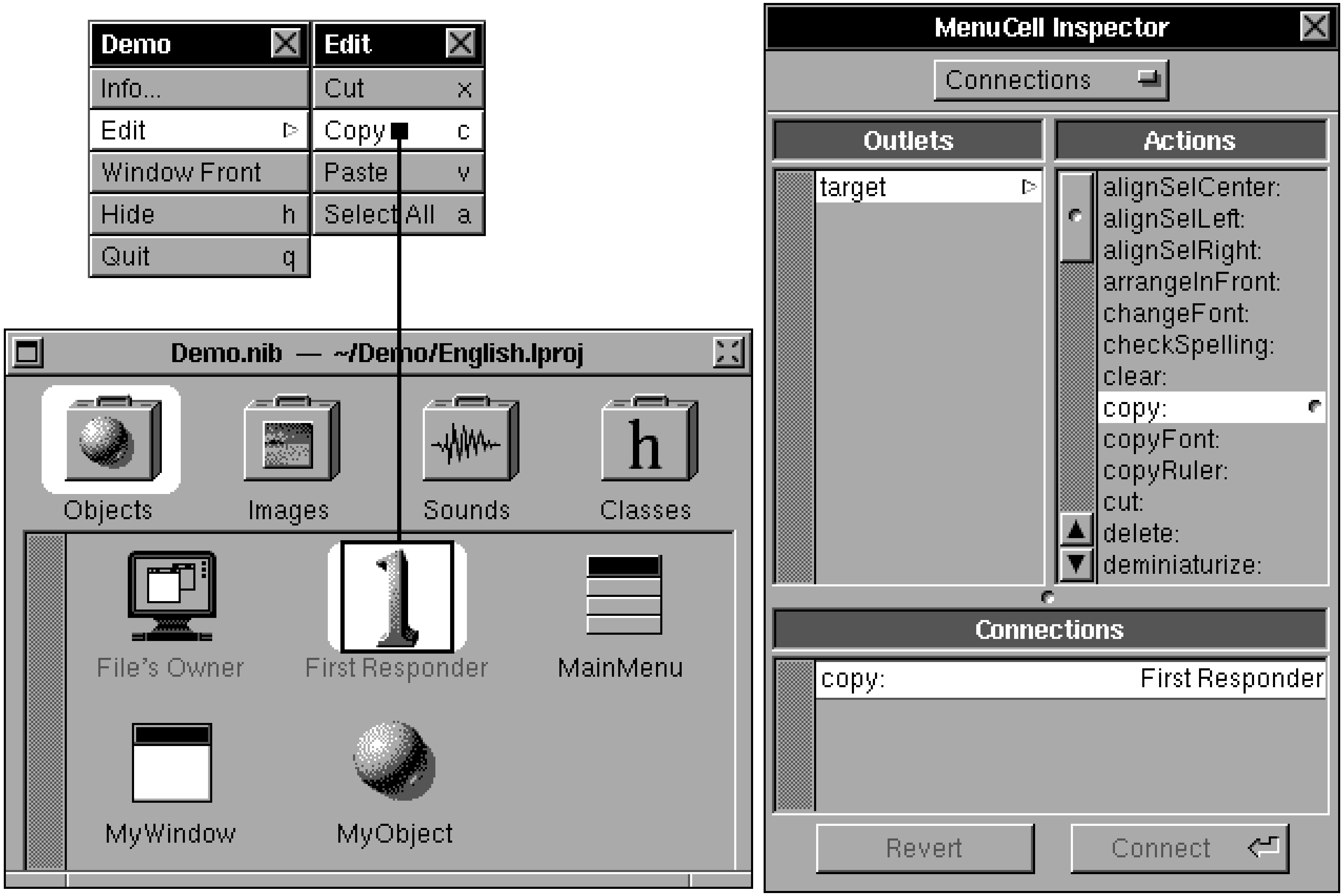
First Responder (continued)

2. Explicitly declare an object to be firstResponder.

```
[window makeFirstResponder:self];
```

Objects which are typically firstResponders are Text and TextField.

In Interface Builder, an interface object can connect to the firstResponder icon in the File Window:



Speaker/Listener

Communication protocol between different processes.

Allows for fast data transfer.

Transparent on network, i.e., processes on different machines can communicate seamlessly.

Resources

NeXT documentation manuals.

NeXTSTEP Applications Programming: A Hands-On Approach, by Simson Garfinkel and Michael K. Mahoney, Spring-Verlag, 1992. The best book anywhere on developing applications under NeXTSTEP.