Inside the Bracket

[what reallyHappens];

Mark Dalrymple

CocoaHeads / Pittsburgh June 2013
CocoaHeads / Atlanta November 2013

http://borkware.com/cocoaconf

Day-in, Day-out

```
- (void) drawRect: (CGRect) rect {
    [[NSColor darkGrayColor] setStroke];

    for (NSString *countryCode in g_countryPaths) {
        NSBezierPath *path = [g_countryPaths objectForKey: countryCode];

        // Ask the delegate.
        NSColor *fillColor = [self.delegate worldMap: self
                                 colorForCountryCode: countryCode];

        if (fillColor == nil) fillColor = [NSColor whiteColor];

        [fillColor setFill];
        [path fill];

        [path stroke];
    }

} // drawRect
```

Day-in, Day-out

```
- (void) drawRect: (CGRect) rect {
    [[NSColor darkGrayColor] setStroke];

    for (NSString *countryCode in g_countryPaths) {
        NSBezierPath *path = [g_countryPaths objectForKey: countryCode];

        // Ask the delegate.
        NSColor *fillColor = [self.delegate worldMap: self
                                 colorForCountryCode: countryCode];

        if (fillColor == nil) fillColor = [NSColor whiteColor];

        [fillColor setFill];
        [path fill];

        [path stroke];
    }

} // drawRect
```

Why

It's all Indirection

It's all Indirection

Any problem in computing
can be solved with an
additional layer of indirection

Indirection

- Loops are indirection

```
NSLog (@"The numbers from 1 to 5:");
NSLog (@"1");
NSLog (@"2");
NSLog (@"3");
NSLog (@"4");
NSLog (@"5");
```

Indirection

- Loops are indirection

```
NSLog (@"The numbers from 1 to 5:");
NSLog (@"1");
NSLog (@"2");
NSLog (@"3");
NSLog (@"4");
NSLog (@"5");
```

$\longrightarrow$

```
NSLog (@"The numbers from 1 to 10:");
NSLog (@"1");
NSLog (@"2");
NSLog (@"3");
NSLog (@"4");
NSLog (@"5");
NSLog (@"6");
NSLog (@"7");
NSLog (@"8");
NSLog (@"9");
NSLog (@"10");
```

Indirection

- Loops are indirection

```
NSLog (@"The numbers from 1 to 5:");
NSLog (@"1");
NSLog (@"2");
NSLog (@"3");
NSLog (@"4");
NSLog (@"5");
```

```
NSLog (@"The numbers from 1 to 10:");
NSLog (@"1");
NSLog (@"2");
NSLog (@"3");
NSLog (@"4");
NSLog (@"5");
NSLog (@"6");
NSLog (@"7");
NSLog (@"8");
NSLog (@"9");
NSLog (@"10");
```

```
NSLog (@"The numbers from 1 to 5:");
int i;
for (i = 1; i <= 5; i++) {
    NSLog (@"%d\n", i);
}
```

Indirection

- Loops are indirection

```
NSLog (@"The numbers from 1 to 5:");
NSLog (@"1");
NSLog (@"2");
NSLog (@"3");
NSLog (@"4");
NSLog (@"5");
```

```
NSLog (@"The numbers from 1 to 10:");
NSLog (@"1");
NSLog (@"2");
NSLog (@"3");
NSLog (@"4");
NSLog (@"5");
NSLog (@"6");
NSLog (@"7");
NSLog (@"8");
NSLog (@"9");
NSLog (@"10");
```

```
NSLog (@"The numbers from 1 to 5:");
int i;
for (i = 1; i <= 5; i++) {
    NSLog (@"%d\n", i);
}
```

```
NSLog (@"The numbers from 1 to 10:");
int i;
for (i = 1; i <= 10; i++) {
    NSLog (@"%d\n", i);
}
```

Indirection

- Variables are indirection

```
NSLog (@"The numbers from 1 to 5:");
int i;
for (i = 1; i <= 5; i++) {
    NSLog (@"%d\n", i);
}
```

$\searrow$

```
NSLog (@"The numbers from 1 to 10:");
int i;
for (i = 1; i <= 10; i++) {
    NSLog (@"%d\n", i);
}
```

Indirection

- Variables are indirection

```
NSLog (@"The numbers from 1 to 5:");
int i;
for (i = 1; i <= 5; i++) {
    NSLog (@"%d\n", i);
}
```

```
NSLog (@"The numbers from 1 to 10:");
int i;
for (i = 1; i <= 10; i++) {
    NSLog (@"%d\n", i);
}
```

```
int count = 5;
NSLog (@"The numbers from 1 to %d:", count);

int i;
for (i = 1; i <= count; i++) {
    NSLog (@"%d\n", i);
}
```

Indirection

- Variables are indirection

```
NSLog (@"The numbers from 1 to 5:");
int i;
for (i = 1; i <= 5; i++) {
    NSLog (@"%d\n", i);
}
```

```
NSLog (@"The numbers from 1 to 10:");
int i;
for (i = 1; i <= 10; i++) {
    NSLog (@"%d\n", i);
}
```

```
int count = 5;
NSLog (@"The numbers from 1 to %d:", count);

int i;
for (i = 1; i <= count; i++) {
    NSLog (@"%d\n", i);
}
```

```
int count = 10;
NSLog (@"The numbers from 1 to %d:", count);

int i;
for (i = 1; i <= count; i++) {
    NSLog (@"%d\n", i);
}
```

Indirection

- Files are indirection

Hard-coding words:

```
const char *words[4] = {
    "aardvark", "abacus",
    "allude",   "zygote" };
```

Indirection

- Files are indirection

Hard-coding words:

```
const char *words[4] = {
    "aardvark", "abacus",
    "allude",   "zygote" };
```

Read them from a file

```
FILE *wordFile =
    fopen ("/tmp/words.txt", "r");
```

Indirection

- Files are indirection

Hard-coding words:

```
const char *words[4] = {
    "aardvark", "abacus",
    "allude",   "zygote" };
```

Read them from a file

```
FILE *wordFile =
    fopen ("/tmp/words.txt", "r");
```

Get file name from program argument

```
int main (int argc, const char *argv[] {
   FILE *wordFile =
       fopen (argv[1], "r");
```

It's an open / closed case

It's an open / closed case

Robust code should be
open to extension
but closed to modification

Open/Closed Principle

I do some stuff, like loop to draw a set of views

Open/Closed Principle

I do some stuff, like loop to draw a set of views

I should be able to draw new kinds of views

Open/Closed Principle

I do some stuff, like loop to draw a set of views

I should be able to draw new kinds of views

Without changing the loop

Open/Closed Principle

I do some stuff, like loop to draw a set of views

I should be able to draw new kinds of views open!

Without changing the loop

Open/Closed Principle

I do some stuff, like loop to draw a set of views

I should be able to draw new kinds of views open!

Without changing the loop closed!

Drawing Views

```
typedef struct View {
    ViewKind kind;
    Rect   bounds;
} View;
```

```
typedef enum {
    kButtonView,
    kSliderView,
    kPonyView
} ViewKind;
```

Drawing Views

```
typedef struct View {
    ViewKind kind;
    Rect   bounds;
} View;
```

```
typedef enum {
    kButtonView,
    kSliderView,
    kPonyView
} ViewKind;
```

```
void DrawViews (View *views[], int count) {

    for (int i = 0; i < count; i++) {
        View *view = views[i];

        switch (view->kind) {
          case kButtonView:
            printf ("Drawing a button!\n");
            ButtonDraw (view);
            break;

          case kSliderView:
            printf ("Drawing a slider!\n");
            SliderDraw (view);
            break;

          case kPonyView:
            printf ("OMG PONIES!\n");
            PonyDraw (view);
            break;
        }
    }
```

Wouldn't It Be Nice?

```
void DrawViews (View *views, int count) {

    for (int i = 0; i < count; i++) {
        View *view = views[i];
        YoViewDrawYourself (view);
    }

} // DrawViews
```

Back to Indirection

Let's add a layer of indirection!

Back to Indirection

Let's add a layer of
indirection!

Instead of calling a function directly
let's look-over-there for what function to call

Function Pointers!

```
typedef void (*DrawCallback) (View *view);

typedef bool (*HitTestCallback) (View *view, Point mouseClick);

typedef char * (*DebugDescriptionCallback) (View *view);
```

Function Pointers!

```
typedef void (*DrawCallback) (View *view);

typedef bool (*HitTestCallback) (View *view, Point mouseClick);

typedef char * (*DebugDescriptionCallback) (View *view);



static void ButtonDraw (View *view) {
    printf ("Drawing a button!\n");
}
```

Function Pointers!

```
typedef void (*DrawCallback) (View *view);

typedef bool (*HitTestCallback) (View *view, Point mouseClick);

typedef char * (*DebugDescriptionCallback) (View *view);



static void ButtonDraw (View *view) {
    printf ("Drawing a button!\n");
}


DrawCallback drawer = ButtonDraw;
```

Function Pointers!

```
typedef void (*DrawCallback) (View *view);

typedef bool (*HitTestCallback) (View *view, Point mouseClick);

typedef char * (*DebugDescriptionCallback) (View *view);



static void ButtonDraw (View *view) {
    printf ("Drawing a button!\n");
}


DrawCallback drawer = ButtonDraw;



drawer (view);
```

Function Pointers!

```
typedef void (*DrawCallback) (View *view);

typedef bool (*HitTestCallback) (View *view, Point mouseClick);

typedef char * (*DebugDescriptionCallback) (View *view);



static void ButtonDraw (View *view) {
    printf ("Drawing a button!\n");
}


DrawCallback drawer = ButtonDraw;


drawer (view);
```

no parens!

Function Pointers!

```
drawer (view);
```

Function Pointers!

```
drawer (view);


drawer = ImageViewDraw;
drawer (view);

drawer = SliderDraw;
drawer (view);
```

So, Let's build a jump table

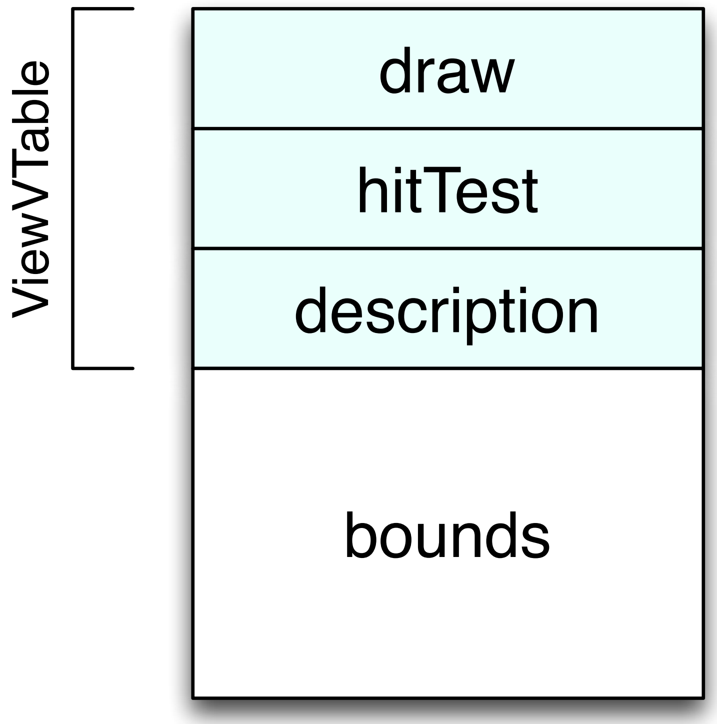
```
typedef struct ViewVTable {
    DrawCallback             draw;
    HitTestCallback          hitTest;
    DebugDescriptionCallback description;
} ViewVTable;
```

So, Let's build a jump table

```
typedef struct ViewVTable {
    DrawCallback             draw;
    HitTestCallback          hitTest;
    DebugDescriptionCallback description;
} ViewVTable;
```

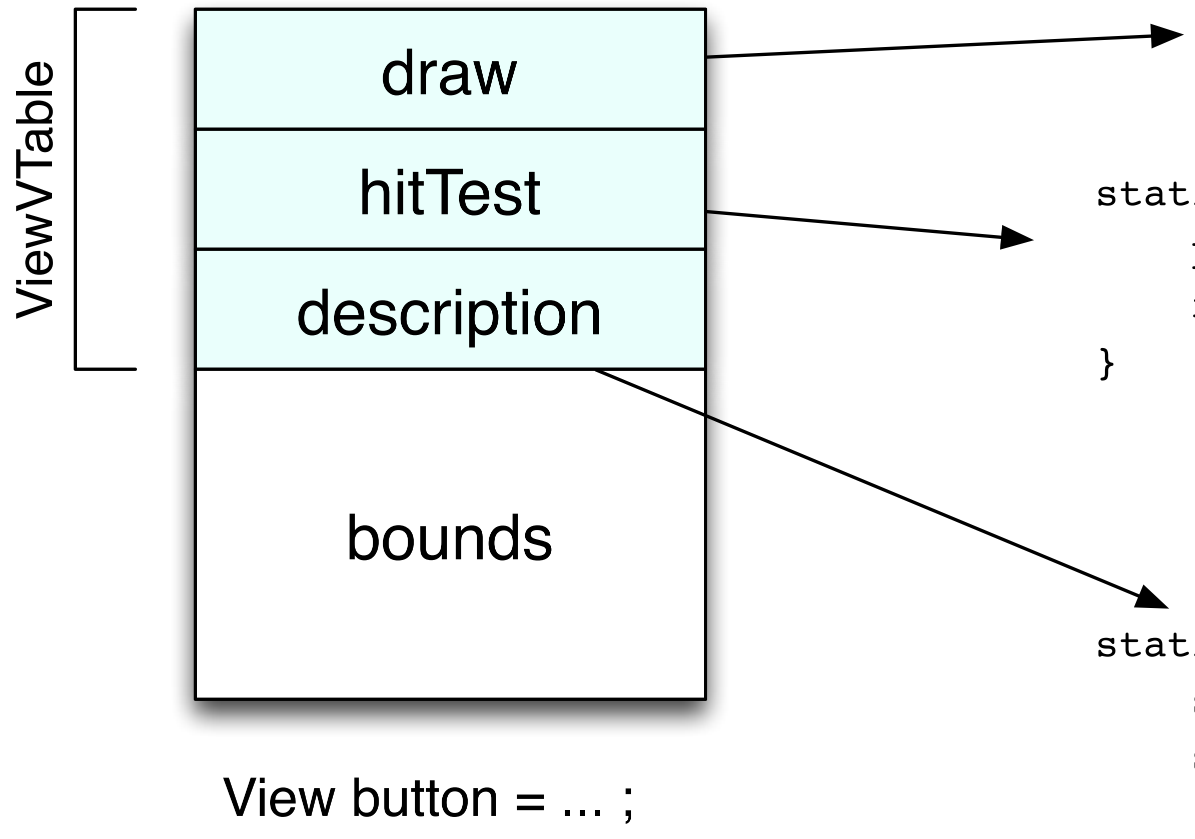
```
typedef struct View {
    ViewVTable vtable;
    Rect       bounds;
} View;
```

The New View Review



View button = ... ;

The New View Review



```
static void ButtonDraw (View *view) {
    printf ("Drawing a button!\n");
}
```

```
static bool ButtonHitTest (View *view, Point point) {
    printf ("Hit testing a button!\n");
    return false;
}
```

```
static char * ButtonDebugDescription (View *view) {
    static char s_unsafeBuffer[1024];
    snprintf (s_unsafeBuffer,
              sizeof(s_unsafeBuffer),
              "Button at %p", view);
    return s_unsafeBuffer;
}
```

Let's use it!

```
View button;

button.vtable.draw = ButtonDraw;
button.vtable.hitTest = ButtonHitTest;
button.vtable.description = ButtonDebugDescription;

button.bounds = (Rect) { 0.0, 0.0, 100.0, 200.0 };
```

Let's use it!

```
View button;

button.vtable.draw = ButtonDraw;
button.vtable.hitTest = ButtonHitTest;
button.vtable.description = ButtonDebugDescription;

button.bounds = (Rect) { 0.0, 0.0, 100.0, 200.0 };
```

```
void DrawViews (View *views[], int count) {
    for (int i = 0; i < count; i++) {
        View *view = views[i];
        printf ("drawing %s\n",
                view->vtable.description(view));
        view->vtable.draw (view);
    }
} // DrawViews
```

Let's use it!

```
View button;

button.vtable.draw = ButtonDraw;
button.vtable.hitTest = ButtonHitTest;
button.vtable.description = ButtonDebugDescription;

button.bounds = (Rect) { 0.0, 0.0, 100.0, 200.0 };
```

```
void DrawViews (View *views[], int count) {
     for (int i = 0; i < count; i++) {
          View *view = views[i];
          printf ("drawing %s\n",
                    view->vtable.description(view));
          view->vtable.draw (view);
     }
} // DrawViews
```

open!

Let's use it!

```
View button;

button.vtable.draw = ButtonDraw;
button.vtable.hitTest = ButtonHitTest;
button.vtable.description = ButtonDebugDescription;

button.bounds = (Rect) { 0.0, 0.0, 100.0, 200.0 };
```

```
void DrawViews (View *views[], int count) {
    for (int i = 0; i < count; i++) {
        View *view = views[i];
        printf ("drawing %s\n",
                view->vtable.description(view));
        view->vtable.draw (view);
    }
} // DrawViews
```

open!

closed!

Let's use it!

```
View button;

button.vtable.draw = ButtonDraw;
button.vtable.hitTest = ButtonHitTest;
button.vtable.description = ButtonDebugDescription;

button.bounds = (Rect) { 0.0, 0.0, 100.0, 200.0 };
```

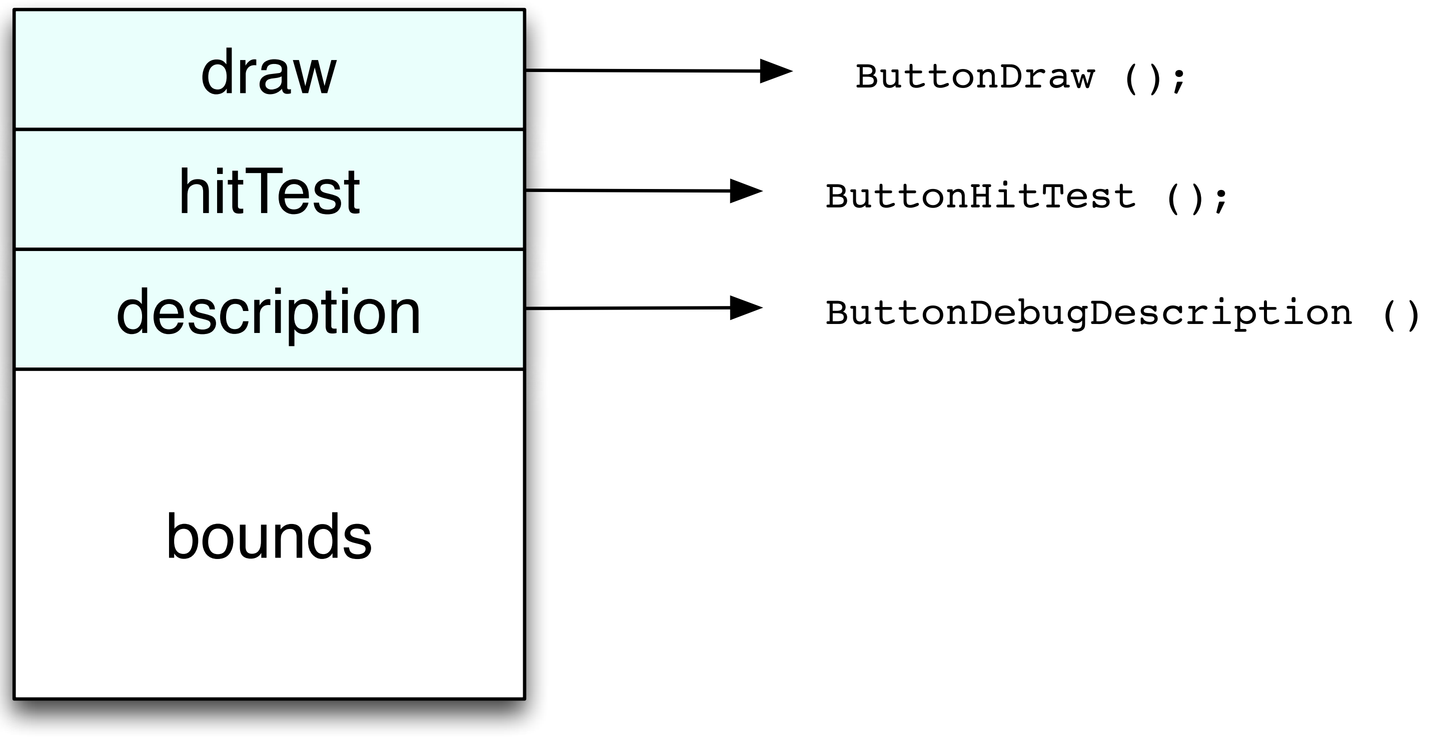
```
void DrawViews (View *views[], int count) {
    for (int i = 0; i < count; i++) {
        View *view = views[i];
        printf ("drawing %s\n",
                view->vtable.description(view));
        view->vtable.draw (view);
    }
} // DrawViews
```

open!

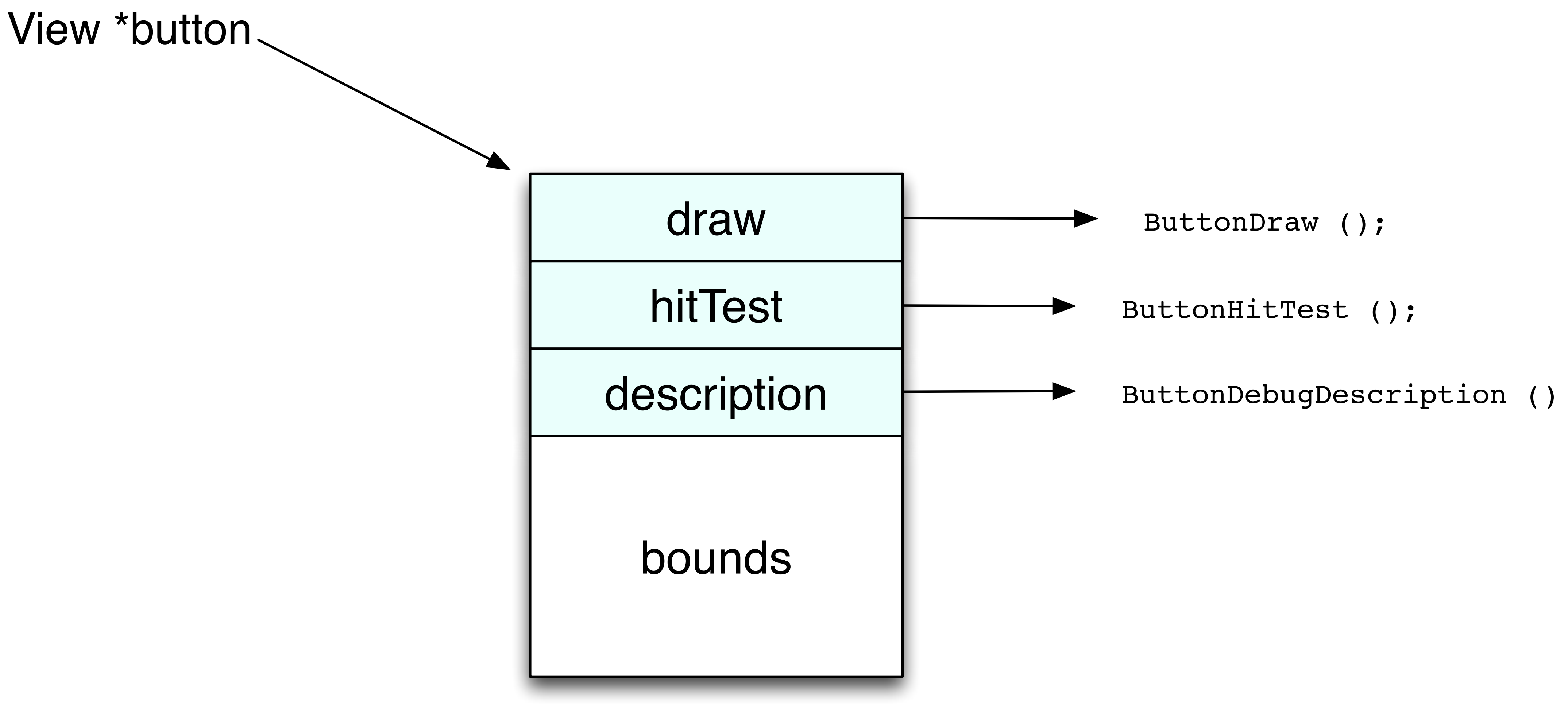
closed!

Polymorphism!

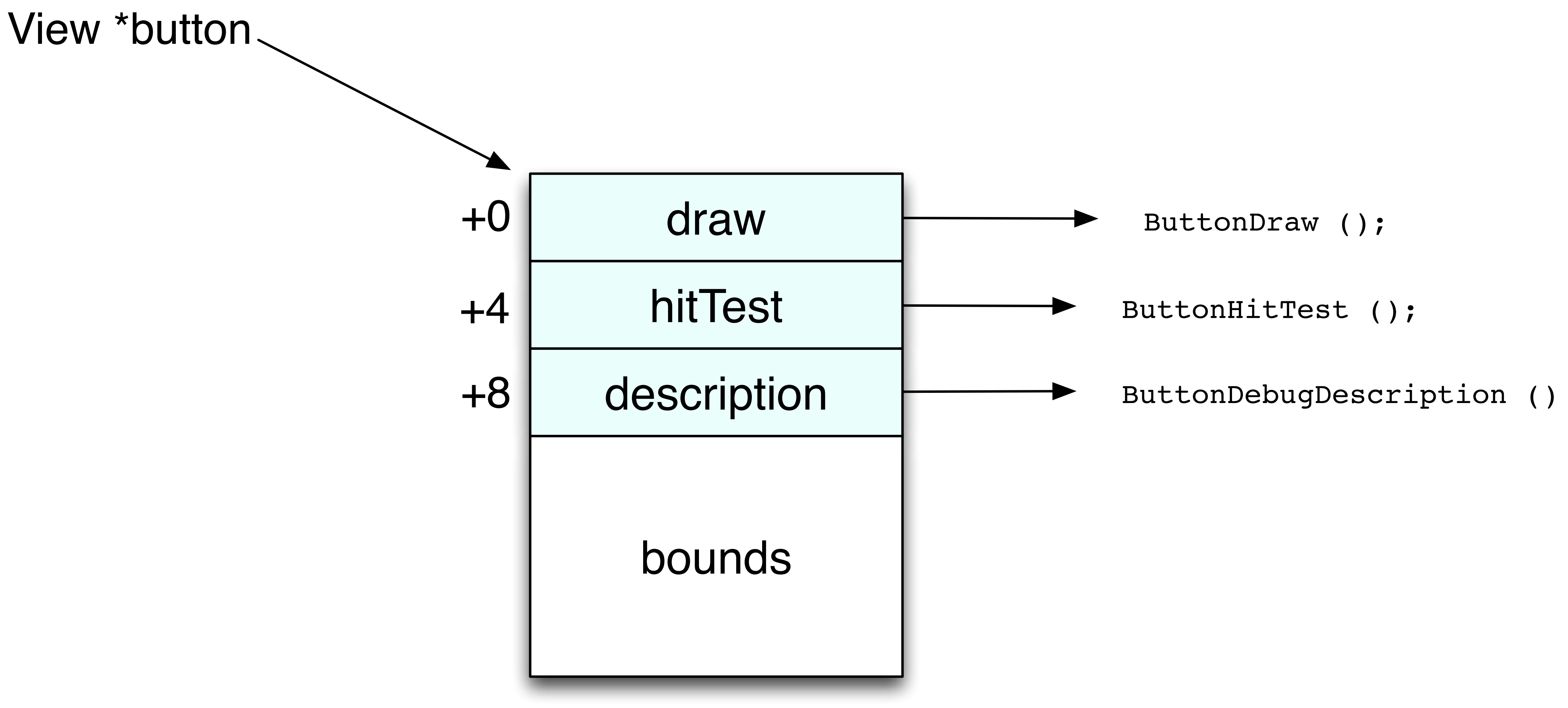
What You Just Saw



What You Just Saw



What You Just Saw



Make it Flexible

Make it Flexible

Let's add a layer of indirection!

Make it Flexible

Let's add a layer of
indirection!

Instead of pointer + offset
let's look up function to call ... by **name**

Wouldn't It Be Nice?

Instead of

```
drawFunction = someView.vtable.draw;
drawFunction (bounds);
```

Wouldn't It Be Nice?

Instead of

```
drawFunction = someView.vtable.draw;
drawFunction (bounds);
```

How about

```
drawFunction = someView.dictionary.GetFunctionPointerForName("draw");
drawFunction (bounds);
```

Wouldn't It Be Nice?

Instead of

```
drawFunction = someView.vtable.draw;
drawFunction (bounds);
```

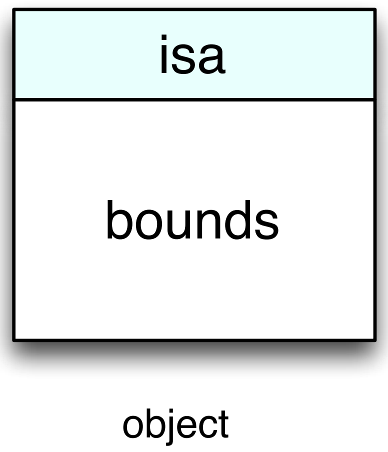
How about

```
drawFunction = someView.dictionary.GetFunctionPointerForName("draw");
drawFunction (bounds);
```

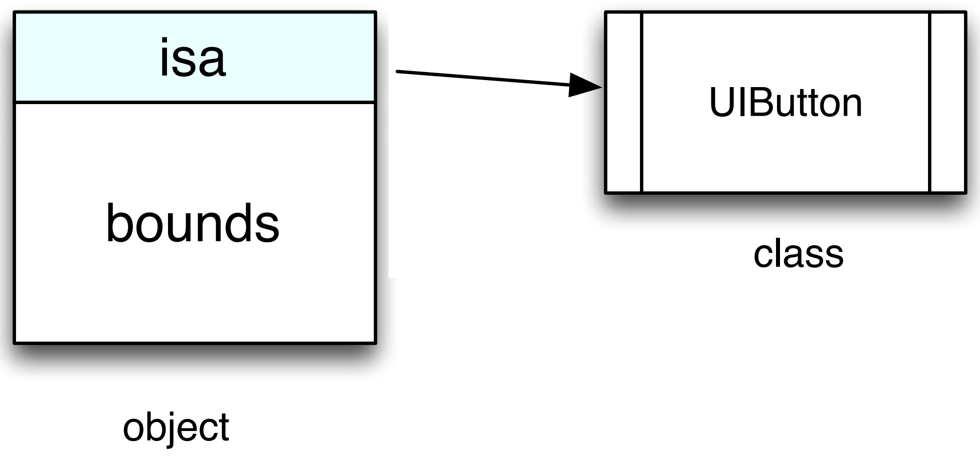
Add Some Fancy Syntax

```
[someView draw];
```

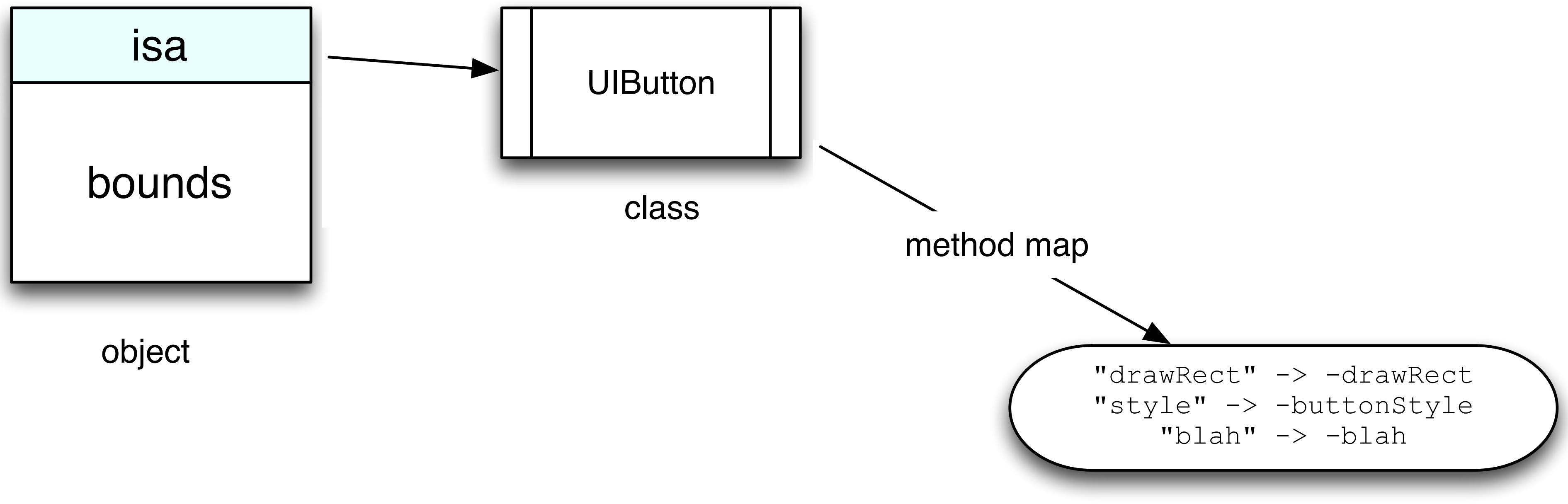
isa bell ringing?



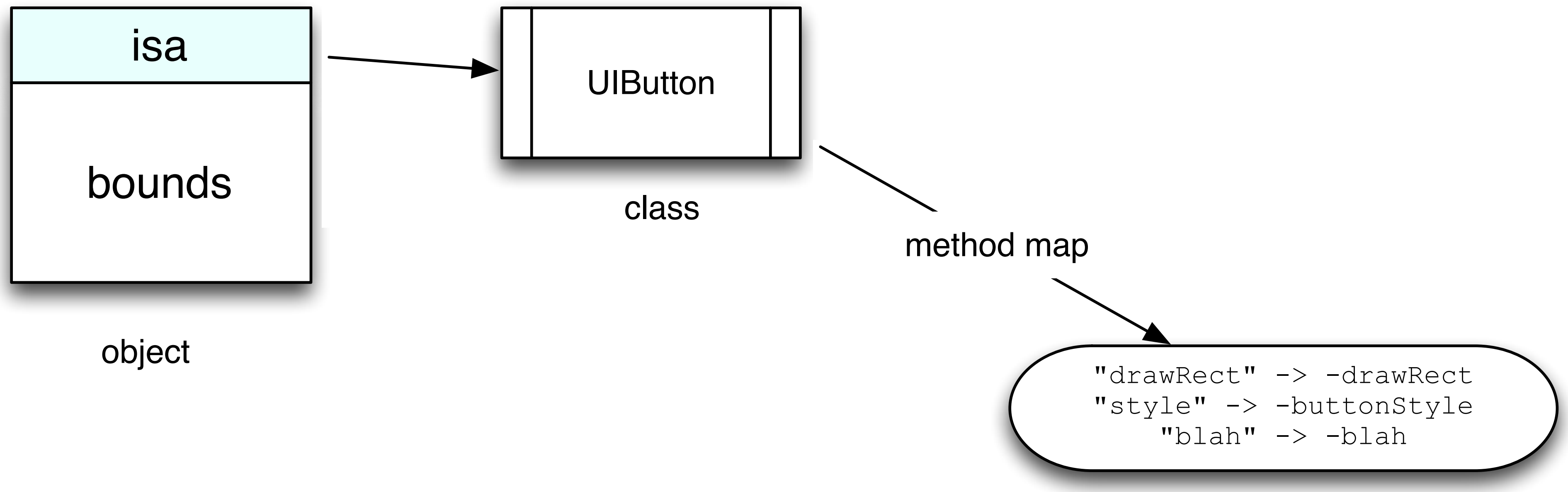
isa bell ringing?



isa bell ringing?

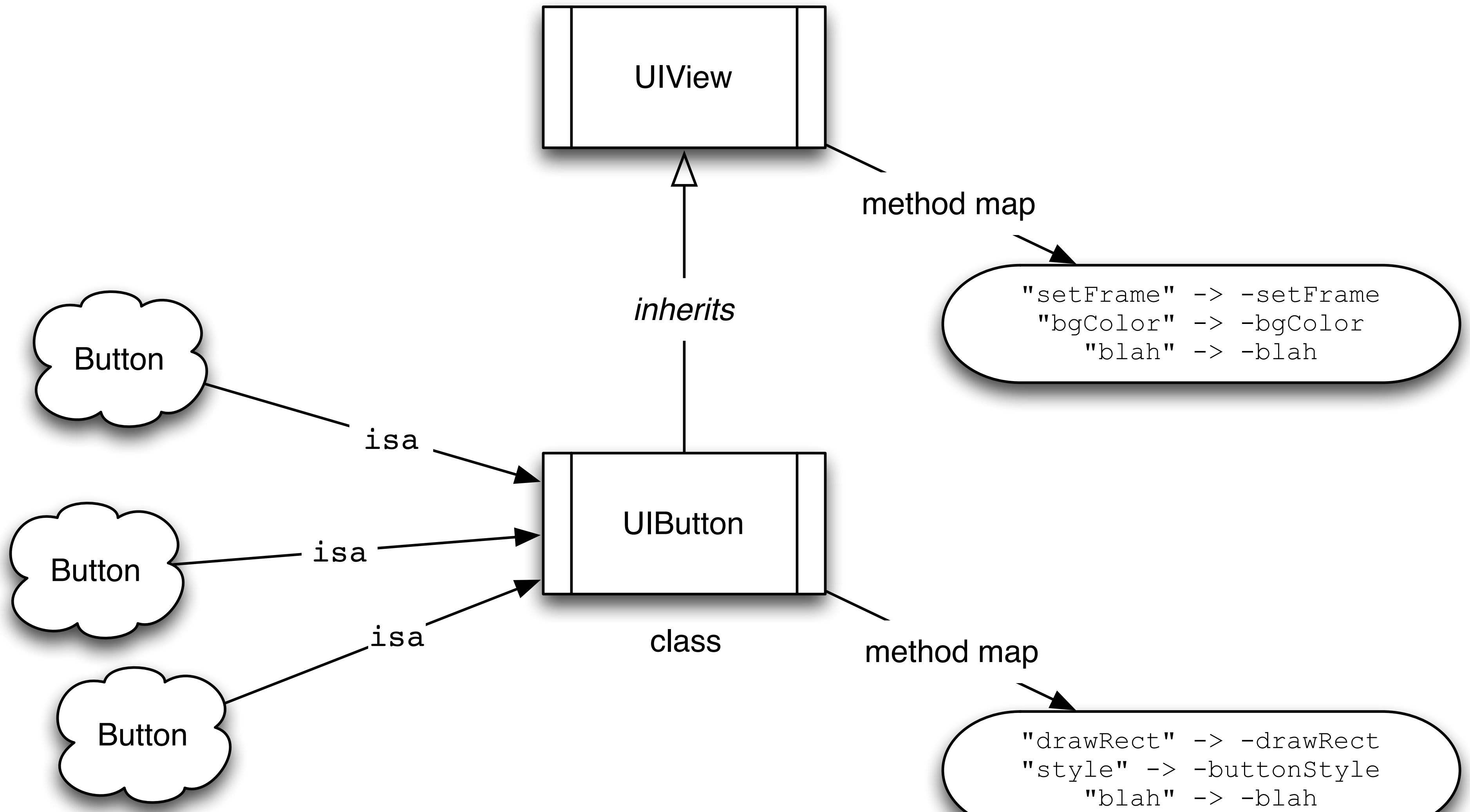


isa bell ringing?

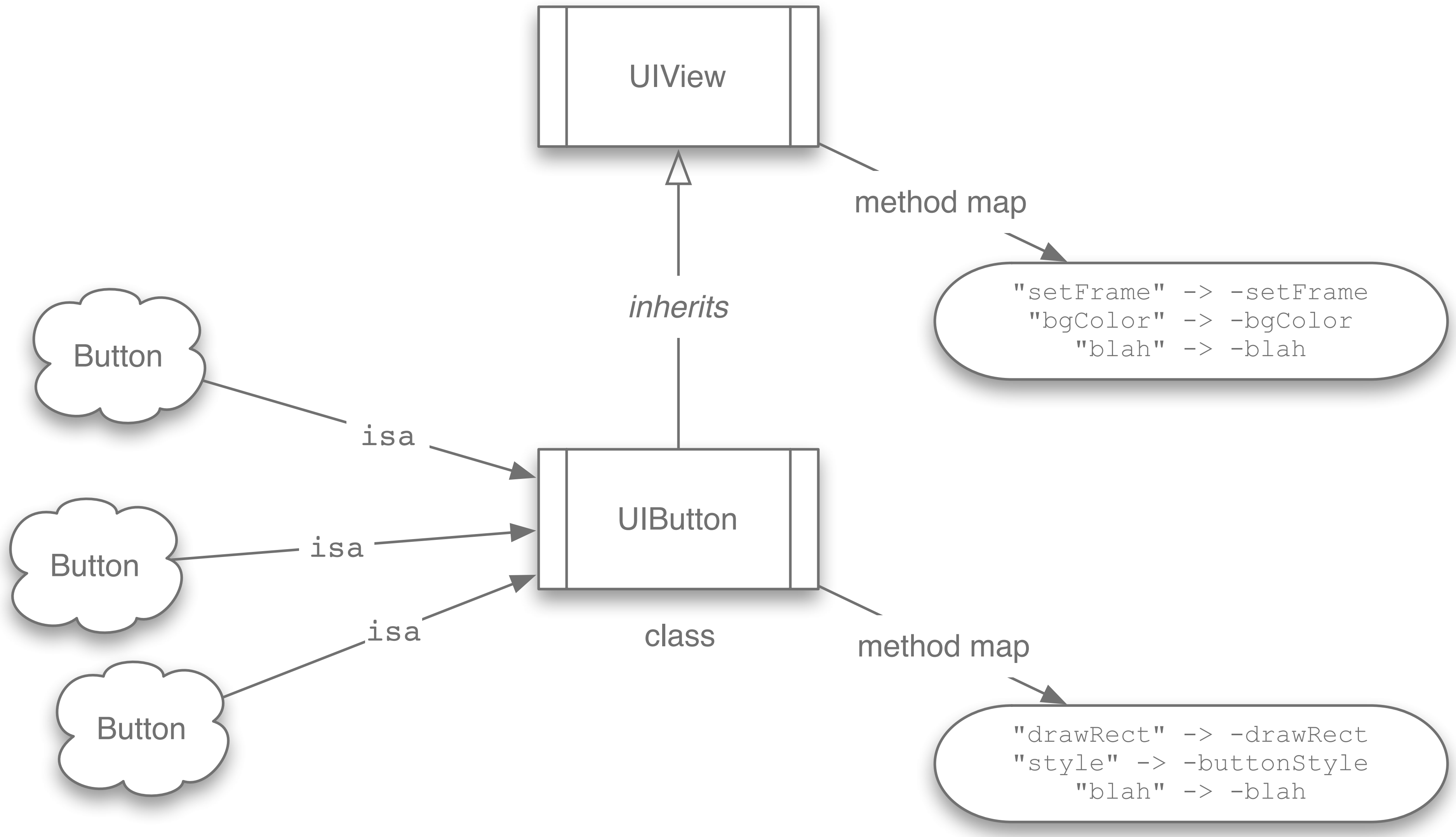


```
[someView drawRect];
```

Inheritance



Inheritance



```
"setFrame" -> -setFrame
 "bgColor" -> -bgColor
    "blah" -> -blah
```

```
"drawRect" -> -drawRect
"style" -> -buttonStyle
    "blah" -> -blah
```

```
[someView setFrame];
```

Wrap up

Wrap up

- Polymorphism gives you flexibility

Wrap up

- Polymorphism gives you flexibility
  - Central to the Open/Closed principle

Wrap up

- Polymorphism gives you flexibility
  - Central to the Open/Closed principle
- It's all indirection

Wrap up

- Polymorphism gives you flexibility
  - Central to the Open/Closed principle
- It's all indirection
- Objective-C maps names to function pointers

Wrap up

- Polymorphism gives you flexibility
  - Central to the Open/Closed principle
- It's all indirection
- Objective-C maps names to function pointers
  - At run-time!