Extracted from:

iOS Recipes

Tips and Tricks for Awesome iPhone and iPad Apps

The Pragmatic Bookshelf

Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

iOS Recipes

Tips and Tricks for Awesome iPhone and iPad Apps

Matt Drance
Paul Warren

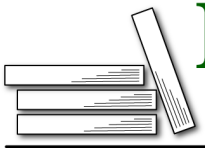
Edited by Jill Steinberg

iOS Recipes

Tips and Tricks for Awesome iPhone and iPad Apps

Matt Drance
Paul Warren

The Pragmatic Bookshelf
Dallas, Texas • Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

The team that produced this book includes:

Jill Steinberg (editor)
Potomac Indexing, LLC (indexer)
Kim Wimpsett (copyeditor)
David J Kelly (typesetter)
Janet Furlow (producer)
Juliet Benda (rights)
Ellie Callahan (support)

Printed in the United States of America.
ISBN-13: 978-1-934356-74-6
Printed on acid-free paper.
Book version: P1.0—July 2011

Recipe 15

Simplify Table Cell Production

Problem

UIKit provides an efficient reuse mechanism for table view cells, keeping overhead low and minimizing costly allocations that slow down scrolling. Although this mechanism works well to curb resource consumption, it tends to be verbose, repetitive, and, most of all, error prone. This common pattern begs for a solution that minimizes controller code and maximizes reuse across multiple views or even applications.

Solution

A basic UITableView layout, as seen in the iPod and Contacts applications, is simple enough to re-create without causing too many headaches: the cells all use the same boilerplate UITableViewCellStyle. Once we venture outside of this comfort zone, however, our code can get messy rather quickly. Consider a custom cell with two images and a text label. Our -tableView:cellForRowAtIndexPath: method may start off like this:

```
static NSString *CellID = @"CustomCell";

UITableViewCell *cell = [tableView
                         dequeueReusableCellWithIdentifier:CellID];
if (cell == nil) {
  cell = [[[UITableViewCell alloc]
                   initWithStyle:UITableViewCellStyleDefault
                reuseIdentifier:CellID]
          autorelease];
  UIImage *rainbow = [UIImage imageNamed:@"rainbow.png"];
  UIImageView *mainImageView = [[UIImageView alloc] initWithImage:rainbow];
  UIImageView *otherImageView = [[UIImageView alloc] initWithImage:rainbow];
  CGRect iconFrame = (CGRect) { { 12.0, 4.0 }, rainbow.size };
  mainImageView.frame = iconFrame;
  iconFrame.origin.x = CGRectGetMaxX(iconFrame) + 9.0;
  altImageView.frame = iconFrame;

  [cell.contentView addSubview:mainImageView];
  [cell.contentView addSubview:otherImageView];
  UILabel *label = [[UILabel alloc] initWithFrame:labelFrame];
  [cell.contentView addSubview:label];
```

```
  [mainIcon release];
  [otherIcon release];
  [label release];
}

return cell;
```

Note we haven't even configured the cell yet! When reusing a cell, how do we get at those now-anonymous subviews that were added during creation? We have two options: set tag literals on the subviews, which we then use to fish them back out at reuse time; or write a UITableViewCell subclass with explicit properties. Going the subclass route is much more attractive because it does the following:

- Defines a contract (properties) for accessing the subviews
- Avoids the danger of tag collisions in the cell hierarchy (multiple subviews with the same tag)
- Decouples the cell's layout from the view controller, enabling code reuse across views and projects

By using a subclass, we get a number of other opportunities to simplify the table-building process. Every table view data source inevitably has the same cell dequeue/alloc code in it. This code is not just redundant; it's also fragile: a misspelled cell identifier, a single instead of a double equals in our nil check—subtle errors lead to performance hits and wasted debugging time. If we didn't have to constantly copy and paste this redundant code, or even look at it, our routine for building table views would be much less tedious.

Enter PRPSmartTableViewCell: a foundational subclass of UITableViewCell that eliminates clutter in our table view controllers and prevents costly bugs in our scattered cell boilerplate. The class's primary task is to abstract away that boilerplate so that, ideally, we never have to worry about it again. The class has a special initializer method and two convenience methods, which we'll explore next.

Download SmarterTableCells/Classes/PRPSmartTableViewCell.h

```
@interface PRPSmartTableViewCell : UITableViewCell {}

+ (id)cellForTableView:(UITableView *)tableView;
+ (NSString *)cellIdentifier;

- (id)initWithCellIdentifier:(NSString *)cellID;

@end
```

The +cellForTableView: class method handles cell reuse for a table view that's passed by the caller—our table view controller.

Download SmarterTableCells/Classes/PRPSmartTableViewCell.m

```
+ (id)cellForTableView:(UITableView *)tableView {
    NSString *cellID = [self cellIdentifier];
    UITableViewCell *cell = [tableView
                             dequeueReusableCellWithIdentifier:cellID];
    if (cell == nil) {
        cell = [[[self alloc] initWithCellIdentifier:cellID] autorelease];
    }
    return cell;
}
```

This code should look familiar: it's nearly identical to the reuse code you've surely written dozens (if not hundreds) of times as an iOS developer. Note, however, that the cell identifier string is obtained from another class method: +cellIdentifier. This method uses the cell's class name as the identifier by default, even for subclasses of PRPSmartTableViewCell you write. Now, whenever we decide to write a custom cell class, we're guaranteed a unique cell identifier for free. Note that the identifier is not marked static as you've seen in most sample code, so there is some extra allocation going on in the default implementation. If you find this to be a problem, you can always override (or edit) +cellIdentifier to change its behavior.

Download SmarterTableCells/Classes/PRPSmartTableViewCell.m

```
+ (NSString *)cellIdentifier {
    return NSStringFromClass([self class]);
}
```

Finally, we use a new designated initializer, -initWithCellIdentifier:, to set up the cell and its layout. This is where we'd put the verbose layout code that would otherwise live in our controller.

Download SmarterTableCells/Classes/PRPSmartTableViewCell.m

```
- (id)initWithCellIdentifier:(NSString *)cellID {
    return [self initWithStyle:UITableViewCellStyleSubtitle
               reuseIdentifier:cellID];
}
```

With this new pattern, here's how we'd write and use our table cell subclass:

1. Create a subclass of PRPSmartTableViewCell.
2. Override -initWithCellIdentifier:.
3. Call +cellForTableView: from our table view controller.

Now let's take a look at our table view controller code for producing a custom PRPSmartTableViewCell:

Download SmarterTableCells/Classes/PRPRainbowTableViewController.m

```
- (UITableViewCell *)tableView:(UITableView *)tableView
         cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    PRPDoubleRainbowCell *cell = [PRPDoubleRainbowCell
                                  cellForTableView:tableView];
    cell.mainLabel.text = [self.quotes objectAtIndex:indexPath.row];
    return cell;
}
```

The controller code is significantly reduced and much more readable—it now contains only the customization of the cell for that particular view. All the cell's characteristic logic and layout is hidden away in the cell class, allowing it to be easily reused anywhere else in this or another project. If you were planning to write a UITableViewCell subclass, this additional code could save you a lot of work in the long run. If you're writing a basic table view with one of the standard cell types, it could be overkill.

This pattern pays especially large dividends when you're writing a heavily customized table view with assorted types of cells. We'll explore this further in Recipe 18, *Organize Complex Table Views*, on page ?.

You can also easily extend this pattern to use custom cells created in Interface Builder, as you'll see in the next recipe.