MOBILE ISSUE

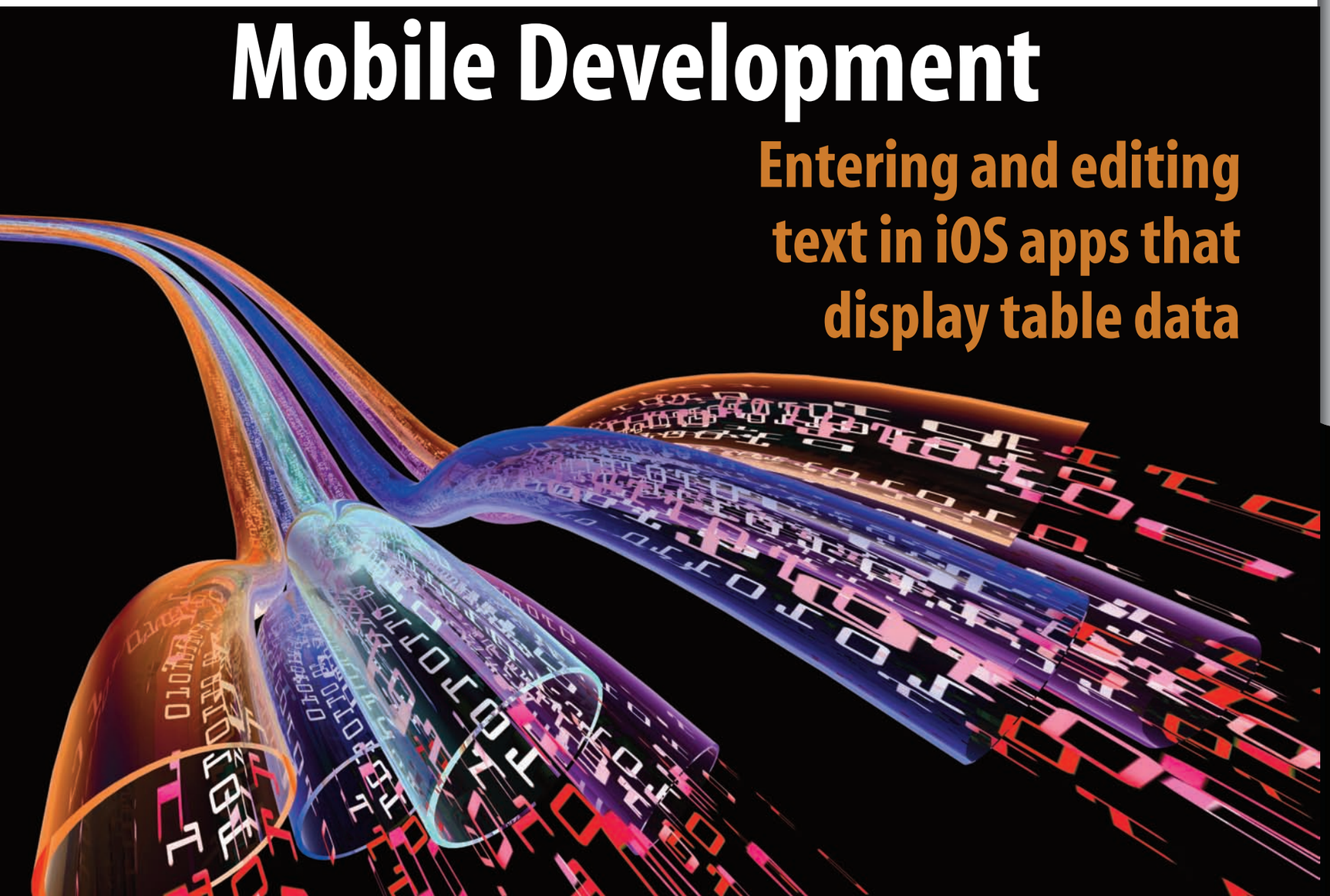
Dr. Dobb's Journal

MAY 2011

Next

Mobile Development

Entering and editing text in iOS apps that display table data



ALSO INSIDE

Java Meets Objective-C >>

Open, Extensible, and Standards-Compliant: What does your project really need? >>

From the Vault: Ternary Trees >>

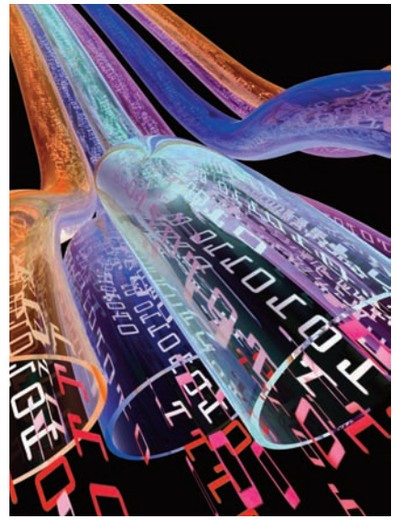
UBM TechWeb

www.drdobbs.com

Dr. Dobb's Journal

CONTENTS

May 2011



COVER STORY

10 Making Two-Way Tables in iOS

By Tom Thompson

Although iOS has several Cocoa Touch classes that specialize in displaying text, the class of choice for creating tables that present a two-way flow of information is *UITextField*. This class is designed to capture and edit small amounts of text that the app then processes, usually immediately.

6 Open, Extensible, Standards-Compliant: Are You Sure That's What You Want?

By Mike Rozlog

Many software projects needlessly saddle themselves with requirements that they think they need, but which frequently just add to overhead and delay. Openness, extensibility, and standards-compliance should be more than check-box items. Add them only to the extent your project needs them.

21 Java Meets Objective-C

By Genadiy Shteyman

For Java programmers, learning Objective-C is not as hard as you might expect. You just need to know how Java maps to the language and where the gotchas are. This is all explained using a small social networking app (in both languages) as an example.

32 From the Vault: Ternary Search Trees

By Jon Bentley and Bob Sedgewick

This seminal article from 1998 by two luminary algorithm specialists presents a new data structure, ternary search trees, which combines the time efficiency of digital tries with the space efficiency of binary search trees. The resulting structure is faster than hashing for many typical search problems, and supports a broader range of useful problems and operations.

3 Letters

By you

From the changing face and shape of documents, to the cost of bad requirements, to RIM's potential problems, Dobb's readers had lots to say.

40 Links

Snapshots of the most interesting articles on drdobbs.com: Recasting pointers safely at runtime, authenticating images, teaching C++ badly, and the Language of the Month.

41 Editorial and Business Contacts

More on DrDobbs.com

Agile at 10: What We Believe (Scott Ambler)

To mark the 10th anniversary of the drafting of the Agile Manifesto, prominent agilists returned to Snowbird, Utah, to determine where the agile community now stands and to examine Agile's core concepts. Scott Ambler takes you inside the meeting, outlines what was discussed and agreed upon, and elucidates what it means to be Agile after a decade of effort.
http://drdobbs.com/architecture-and-design/2293011280

Qt Application Development for Symbian

Qt is a cross-platform native development framework for all major operating systems, including Linux, Windows, Mac, Symbian, and Maemo. This article explores developing smartphone applications with Qt for Symbian OS and deploying them on Nokia Ovi Store.
http://drdobbs.com/mobility/229301197

Quartz 2.0 for Java

Quartz is a open-source job scheduler for Java developers, and can be used to schedule tasks that range from simple to complex.
http://drdobbs.com/blogs/java/2294013123

Android, Airplanes, and Assembler: IDEOne

Al Williams describes his experiences with a nifty little tool from the Android market: IDEOne.
http://drdobbs.com/blogs/embedded-systems/229400478

Data-Centric Architecture: A Model for the Era of Big Data

As businesses struggle to handle large volumes of data rapidly, technologies used in industry and by the masters of the universe on Wall Street are suddenly becoming fashionable...again.
http://drdobbs.com/architecture-and-design/229301018

IN THIS ISSUE

Guest Editorial >>
Two-Way Tables in iOS >>
Java Meets Objective-C >>
Ternary Search Trees >>
Letters >>
Links >>
Table of Contents >>

Mailbag

Readers sound off on the changing face and shape of documents, the cost of bad requirements, and RIM's Playbook

Below The RIM

In response to "Note to RIM: All Things to All Developers Will Not Succeed" (http://drdobbs.com/mobility/229400918), which cast doubts on RIM's strategy for attracting developers to its Playbook tablet and Blackberry phones:

I work in the public sector and the one advantage that RIM has over all their competitors is the fact that their devices meet the stringent GCSX security standards — basically it's the only device that the government trusts to handle data securely. When a Playbook is paired with a Blackberry, this security is built into the tablet, too. I'm really looking forward to seeing what we can develop with the Blackberry and I for one can't wait to get hold of a Playbook — I think the smaller size gives it an advantage over the iPad. I will admit that we're not going to use RIM's own developer toolkit (I had a quick go with this on Eclipse and found it difficult to come to grips with to say the least); we'll be using a toolkit provided by a third party. The thought of having to write code in C++ (I'm a VB.NET developer by trade) fills me with dread!

—Mark Perry

The Evolving Shape of Documents

In response to "The Fast Evolving Shape of Documents" (http://drdobbs.com/tools/229400040)

In response to your editorial, "The Fast Evolving Shape of Documents," which argued that the fast changing formats of documents will have dramatic effects on our usage and on programming:

The key item is that the historical use of static monochromatic text is no longer necessary. It is technically possible for documents to include animation, multiple colors, and even three dimensions.

Since software applications are dynamic and fast moving, I should think that animated software design methods would improve software quality and reliability.

Ordinary news documents and journals could also expand by including some animated segments. This already occurs with some of the ads that pop up when you access journals, but I'm speaking of editorial content and not extraneous marketing material.

Incidentally, if you analyze the costs of large software application development, the costs in rank order are:

IN THIS ISSUE

Guest Editorial >>
Two-Way Tables in iOS >>
Java Meets Objective-C >>
Ternary Search Trees >>
Letters >>
Links >>
Table of Contents >>

1 Finding and fixing bugs
2 Producing paper documents
3 Code development
4 Meetings and communications
5 Project management

It is a known fact that requirements, design, and document bugs outnumber coding bugs. Some of these could be eliminated with more sophisticated animated design representations.

Here is the distribution of defect origins:

1 Requirements bugs = 1 per function point
2 Design bugs = 1.25 per function point
3 Coding bugs = 1.75 per function point
4 User document bugs =0.6 per function point
5 Bad fixes or secondary bugs =0.4 per function point

Total = 5.0 bugs per function point.

As you can see, 3.4 bugs out of 5.0 are non-code defects that originate in documents.

—Capers Jones

To make electronic forms of documents an adequate, perhaps superior replacement for printed documents, we need three levels of independent abstractions for documents: the document itself, how the document is rendered on a particular media or device, and the markup for that document. A document should be able to be rendered through a transformation that is a function of a device and display parameters. This leads to the notion that page numbers simply are not ideal for electronic documents. Also, I'd like the markup to reference, but be separate from, the book. I want my markup to be durable, but one thing that this could also give us is multiple markups for the same document. Of course, this markup needs to be rendered also. So, the pipeline is Document->Document+Markup->Display Device. Markup also needs to go back from Display-Control to portable markup.

—Geoffrey Pascoe

You wrote:

- "The browser itself, of course, is a page-oriented device." I don't agree — I think of it more as a continuous scroll(ing piece of paper), and in some cases, a browser (with funky CSS stylesheets) will even reformat pages in separate tabs, which is more "folder like" than page like.
- "Such that, the concept of, say, starting to read at page 145 is now insufficient" True, this is where the Bible got this right years (centuries) ago…it is quite easy to refer to "chapter and verse" in any Bible and know what you are talking about regardless of how big the print is or what kind of paper (even ePaper) it is "printed" on. Maybe this is what we need for the future of eBooks!
- "The orientation of documents has changed…" And of course, with iPads, iPhones, Androids, etc., the orientation can change dynamically when the user rotates the device. This means that the content needs to readjust automatically to the orientation of the device.

FYI, when I was reading a book with Google Books' eReader, one of the options was to show the book in "scanned mode"; which would

IN THIS ISSUE

Guest Editorial >>
Two-Way Tables in iOS >>
Java Meets Objective-C >>
Ternary Search Trees >>
Letters >>
Links >>
Table of Contents >>

preserve the pagination of the original (hardcover, I assume) book — another interesting thought on preserving page references.

—John Silva

Andrew Binstock responds: "My comment about browsers might need more clarification. When reading long articles, most sites still send you to a new link to read page 2, page 3, etc. The metaphor is still principally the page."

Your editorial resonated deeply with me. I'm a former Technical Communicator (a.k.a. Technical Writer) for a software company. As a Content Manager and Knowledge Management system consultant for the same software company, I'm still amazed at how we hold on to the "book" paradigm when it comes to documentation. This includes our product documentation group.

My working "manifesto" on this topic:

- Content must be decoupled from the delivery medium.
- Content must be delivered in its own context, and not (necessarily) as a part of a narrative.
- If the design calls for a PDF output, documents must be delivered in landscape orientation. This was a tough sell for me (as a Tech Communications instructor) as recently as five years ago. It made sense to me then (projector screens and flat panel monitors), and it makes sense to me now for the same reasons you mention (iPad). It's also my belief that our giant flat-panel TV screens will be a common document delivery vehicle.
- As a consumer of documentation and other content, the fewer formatting barriers, the better. In other words, just get me to the content without having to think about color, typeface, bells, or whistles. This is especially true for software project artifacts. It's tough enough for the engineers to write clearly, but that's another manifesto.

—Michael Boor
Wonderware Developer Network

COMING SOON TO DRDOBBS.COM

The next installment of our series on Cloud Programming written by Allen Holub

Dennis Shasha on Redundancy and Reliability

Language of the Month: Groovy ++

Plus another installment new feature, Project of the Month:

The Best of the Little-Known Open Source Programming Tools

Have a correction or a thoughtful opinion on Dr. Dobb's content? Let us know! Write to Andrew Binstock at alb@drdobbs.com. Letters chosen for publication may be edited for clarity and brevity. All letters become property of Dr. Dobb's.

IN THIS ISSUE

Guest Editorial >>
Two-Way Tables in iOS >>
Java Meets Objective-C >>
Ternary Search Trees >>
Letters >>
Links >>
Table of Contents >>

Extensible, Open, Standards-Compliant: Are You Sure That's What You Want?

By Michael Rozlog

This month, I invited Mike Rozlog of Embarcadero Technologies to give us an ISV perspective on business programming — and on the problems he sees at customer sites. The result is an excellent argument for the extreme programming practice of YAGNI (You ain't gonna need it). — Ed.

Over the past few years I've to come to the realization that almost any technology can solve most problems given enough time and money. This is not to say that a technology choice is not important, it's just lower in the order of issues facing today's developers. For example, I would not want to use my high-speed computing team writing in Assembler to create a website in PHP: Not because they could not do it, but because it is not what they are good at.

However, I'm beginning to think that the real barrier to getting things done in our profession is the tendency to add three requirements to projects: making them extensible, open, and standard. Let's look at these in greater detail.

Extensible

Project definitions usually require that the application be extensible to other applications and be able to interact with other applications' data. The problem with using this word in the project is that it is very hard to define. It could mean that I may have to allow for new modules to be added without disrupting the existing application, or it could mean that end users can extend the current delivered content or application in some way. When most teams at ISVs see the word "extensible" in the project plan, they see that their timeline and cost has just grown substantially. There are very few applications that are truly extensible. Sure, a few come to mind — the Eclipse project, Delphi/C++Builder, and Visual Studio — but as you see, most of these are development software. There are others that allow for open APIs to be used to expand the usability of software.

However, in a general business software application, where does extensibility come in? Who tests the extensibility? When is extensibility added? I would submit that adding the concept of "extensibility" to an application should require a new project just for that feature. Sure, you could put in code that could help with extensibility in the future, but good programming practice states we don't put anything into the code that is not going to be used because it adds overhead, maintenance, and complexity to an application that may or may not ever use that feature. [*YAGNI —Ed.*] So where extensibility is specified, add only the needed functionality and only when it is really needed. This minimal, late approach significantly reduces the amount of work and rework a team needs to do for this nebulous requirement at best.

IN THIS ISSUE

Guest Editorial >>
Two-Way Tables in iOS >>
Java Meets Objective-C >>
Ternary Search Trees >>
Letters >>
Links >>
Table of Contents >>

An example illustrating the real cost of writing extensible code can be found in "How (not) to write Factorial in Java" (http://chaosinmotion.com/blog/?p=622): A project requirement that simply states that a feature needs to be extensible, or worse yet, that the application needs to be extensible, fails to specify or anticipate what developers might do to make it happen.

Open

What is an open architecture? Does it help you get the application done quicker, better, and more correctly? If the answer is "no," then who cares if the application is open? For the majority of applications out in the world today, most of the ones that are built on an "open" architecture don't use it enough to deliver that openness. As an industry, we try to adopt these things that claim to be open in the hopes of easier or better integration in the future. However, that point rarely ever arrives; and when it does, the openness is often of little use.

Like the argument for "extensibility," the concept of "open" does not need to be addressed until there is a clear and defined need — for example, if the application must be open to accepting a web service that transmits secure customer information for logging by another system. If, as developers, we worry about a requirement like this when we start development, even though that data is not ready to be emitted by our system, then we end up wasting a lot of time building unneeded infrastructure that most likely will have to be modified when the requirement becomes real and an update to the project is needed. All too often, we see the words "open" in a project and it makes the development and business teams do a lot more busy work trying to define an architecture that may some day in the future need to be open, but at this time, is not needed.

IN THIS ISSUE

Guest Editorial >>
Two-Way Tables in iOS >>
Java Meets Objective-C >>
Ternary Search Trees >>
Letters >>
Links >>
Table of Contents >>

There is enough evidence that states requirements are the key to both project success and failure. [*See Capers Jones's letter in the Letters section for more on this topic. –Ed.*] Infotech Research stated in 2005: "Flawed requirements trigger 70% of project failures." Just this one non-functional requirement — openness — can significantly reduce the chances of success for the project.

Standard

The requirements for standards-compliance is the cause of an incalculable amount of rework. There are people standing around the water cooler, executives setting edicts, and teams figuring out the right features from a smorgasbord of accepted, developing, or proposed standards must be used in the next big project. Why? We all know by now that standards come and go in our industry.

Remember when it was important to support EJB 1.0, 1.1, 1.2; or back in 1998 when it was SOAP 1.0; or was that XML-RPC? The question that must be asked over and over again is, "what did adhering to those specifications do for your business?" I would surmise not much, because if you were using SOAP or XML-RPC as a protocol, you were still doing maintenance projects to get things working with each integration partner. In some cases, your development team worked very hard to be compliant with that 1.1 standard. And if you were 1.1 compliant, and the other integration partner was 1.1 compliant, it may have saved a few minutes of work; but in actuality, the difference between the "bad" 1.0 and "good" 1.1 back then was very minimal. It was mostly marketing hype and FUD (Fear, Uncertainty, Doubt) put out by the submitting vendor.

I knew of many companies that ripped out software that worked just to be on the latest standard. In actuality, these companies did everything needed to integrate a partner or system, but they still needed to go through, at a very minimum, a full regression testing cycle to ensure that things were working correctly. So even though both sides of the equation were compliant with the relevant standard, they still had to do a full project to verify the integration.

So developers spent countless hours making sure to support this "standard" and that "standard," and at the end of the day, the work to integrate partners was still a job that needed a full project and testing cycle to ensure the business activity was working correctly. Chasing after these constantly evolving specifications and standards is causing a lot of churn — much of it wasted if the application does not have the specific need.

This churn costs all of us real money. Estimates using the 1995 *Chaos Report* indicated that companies would spent about US $81 billion on failed or canceled projects in 1995. Ten years later, the *Chaos Report* from 2005 showed that we as an industry had become worse, not better; so the number adjusted for inflation and a higher failure rate mean we are losing more investment dollars on almost every single project today.

What Can Be Done?

Beyond keeping an eye out for these three attributes in project definitions and discussions, the key is to bring back "focus" on the important things in a project. One of the questions I ask when training developers is, "what is the most important thing the system you create has to do?" The answers I get back from students is rather telling (shown here with my responses):

- Your software needs to be future proof — *really?*
- Your software must be easy to read — *ok, but is that the most important?*

IN THIS ISSUE

Guest Editorial >>
Two-Way Tables in iOS >>
Java Meets Objective-C >>
Ternary Search Trees >>
Letters >>
Links >>
Table of Contents >>

- Your software must be maintainable — *ok, again but is that the most important?*
- The software must have good documentation — *nice to have…*
- The software must run fast — *nice to have…*
- It must be developed using Agile methods — *really?*
- It needs to run on the latest "x" framework — *really?*
- It must support both white and black box testing — *really, really nice to have!*

The list goes on, and on, and on, and on…and few people can just state that the software being developed must work. That is all, technically nothing more and nothing less. If I develop my latest project to be the most extensible, the most open, and the most standard software ever created by man and it can not calculate 2 + 2 correctly, none of the other answers matter.

Once we have software that works, I'm in favor of the other things. Of course I want to refactor my software for maintainability, of course I want my software optimized, I would love for the software I create to be easily maintained, and I always love working on a agile team — but if the team cannot make the product work, nothing else matters. However, far too many people focus on those other tasks before they have a workable solution.

So in my opinion, words do matter and they matter at all levels. Watch for executive management talking about the dreaded three requirements. Help educate them on how the software you are developing is going to work and save time, money, or personnel for the company. Help the team understand what the main goal of the application is; it needs to work first, then you can do the refactoring and add all the bells and whistles.

And Then…

With all the work we have done as an industry, the latest statistics show that most of the world still runs on COBOL. Studies show that 90% of financial institutions use COBOL for their majority of processing, despite it being a 50-year old language. There are some 200 billion lines of COBOL still running the show, day in and day out. So how important is it that the next big application run the latest language, use the latest standards, and connect to all kinds of endpoints?

If nothing else, it should point out that always moving to the next language or setting this language or that language as a standard really does not hold a lot of water. Think about how many of you reading this now work for a company that has project standards that must be adhered to (Java, .NET, etc.), but the underlying part of your business is still running COBOL and it is getting the job done.

The sad fact is that most companies could likely do "whatever the requirement was" in the initial technology. That may be COBOL for company X, but it might have been C for company Y, and it may have been Delphi for company Z. Keep in mind that languages do become extinct over time. Things like PL/1 and RPG may still be used in some corner in a basement, but for the most part, those languages are dead. Then again, many languages over the past couple of decades have been pronounced dead, but here they are, alive and kicking. Look at Objective-C, created back in 1980. Did you think back in 1998 when you were worried about the latest SOAP standard that, in 2011, you would be looking at writing your next piece of software in Objective-C, I doubt it!

— Mike Rozlog worked for many years at Borland and CodeGear. Today, he is the Director of Delphi Solutions at Embarcadero Technologies.

IN THIS ISSUE

Guest Editorial >>
Two-Way Tables in iOS >>
Java Meets Objective-C >>
Ternary Search Trees >>
Letters >>
Links >>
Table of Contents >>

Making Two-Way Tables in iOS

Although iOS has several Cocoa Touch classes that specialize in displaying text, the class of choice for making two-way tables is *UITextField*

By Tom Thompson

In a previous *DDJ* article, "Displaying Tabular Data on iPhones" (http://drdobbs.com/mobility/229218541), I discussed how to use the *UITableView* class in Apple's iOS SDK to generate and present tables. Such tables can effectively display a lot of information on small screens, particularly the iPod Touch and iPhone. For some apps, simply presenting the data — such as stock market quotes, weather reports, price information, or an itinerary — is sufficient. However, for many business apps this is not enough: The table must implement a two-way flow of information. For example, a part-ordering app not only has to display what is in inventory, but also allow a user to enter a request that orders a part (or parts). This article helps close the communications loop by showing you how to enter and edit text in the table.

Although iOS has several Cocoa Touch classes that specialize in displaying text, the class of choice for our purpose is *UITextField*. This class is designed to capture and edit small amounts of text that the app then processes, usually immediately. *UITextField* does this by presenting a text string within an editable on-screen area or field. Visually, the field is perceived as editable because it is surrounded by a border — unlike text that belongs to a static label. When a user taps on the field, the *UITextField* first presents a mechanism (typically a virtual keyboard) that lets them modify the text. When the user completes editing the text, she taps the Return key. The keyboard disappears, and the *UITextField* instance then sends action messages to a delegate object for a response. In our case, when the keyboard is dismissed, the *UITableView*'s delegate method writes the modified text back into the data model. The data model is typically a custom class that is responsible for storing and updating the information that the table displays. From there, to continue with the parts ordering example, the revised data might be sent over the air to a parts database server, where the appropriate scripts are launched and complete the transaction. Let's take a closer look at *UITextField* to see how we'll put it to use for entering text.

UITextField Overview

A brief tour of the class taxonomy should help you understand the capabilities of *UITextField* and what methods you can expect to use. *UITextField*'s ancestry and related classes are shown in Figure 1. Like any iOS class whose purpose is to display content on-screen, *UITextField* ul-

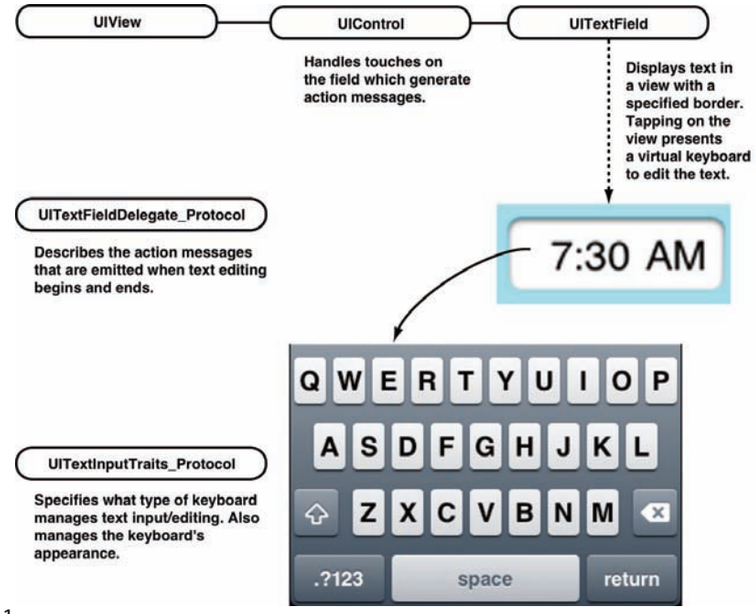


Figure 1.

timately inherits from the *UIView* superclass. However, its immediate superclass is *UIControl*, because a *UITextField* object has to interact with the user. Specifically, *UITextField* issues action messages in response to touches within its view that initiate or end a text editing session.

Since *UITextField*'s input field is actually a view, the class has the usual Objective-C properties that it inherits from *UIView*. For example, there are properties that define the view's background color, its frame size, the alpha channel value, and its position within its superview. *UITextField* also adds properties that describe a background image and border style for the view.

Because it displays text, *UITextField* has properties that specify the text's typeface (or font) and point size, its alignment within the field, its color, and — of course — the text string itself. This last property, text, has read and write characteristics, and we'll use this feature to display and update the table's display of content.

There are also two ancillary protocols that refine *UITextField*'s behavior and action messages in response to user input. Recall that a protocol defines a set of methods that implement specific capabilities for the class. The *UITextInputTraits_Protocol* defines behaviors that are associated with keyboard input. It can specify the type of virtual keyboard used (such as whether it is standard keyboard or a numeric keypad), and how the text appears on-screen (the text can be displayed as bullet symbols when secure input is requested). It also describes the string displayed in the Return key. For example, "Go" might be presented in the key when an URL is entered, or "Search" if the input string is being used to conduct a search. The *UITextFieldDelegate_Protocol* specifies the action messages that the *UITextField* object sends to a delegate object. Many of these messages signal that text editing has begun or has ended on the field. The former messages can be used to invoke a custom keyboard or object (such as a picker) for text entry, while the latter are used to initiate processing on the edited text. Other messages allow access to the edited text, and can modify the behavior of the text display or of the Return key. As an example, the field can clear the text string when it is tapped on, a desirable behavior if you expect the user to enter text that's different from that being displayed.

UITextField's capabilities therefore make it ideal for building an input/editing mechanism into a table. First, the object displays text, so it can be co-opted to display the table's content. Second, the border around the field informs the user that they can tap and change this content. Finally, when the user completes entering new content and taps Return, we can use the resulting action messages to store the revised content into the table's data model. Finally, we send an update

message to the table object so that it reads the modified data and updates its on-screen content. So, now let's put what we've learned into writing some code.

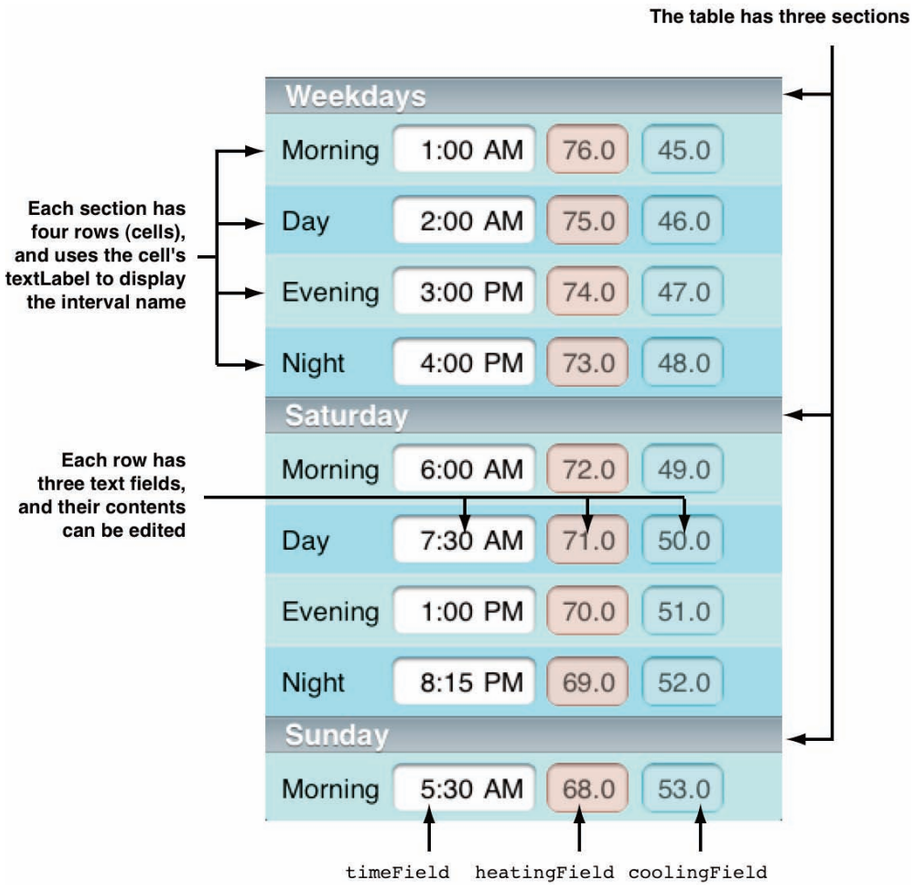


Figure 2..

Building the Two-Way Table

Let's proceed step-by-step to assemble a working two-way table interface. That means first making a table, then wiring in some instances of *UITextField* into the table's cells. Then we complete the connection to the data model through the action messages generated by *UITextField*. For this example, we'll assume a hypothetical table that displays thermostat controls. The table will display days of the weeks as sections. Each day (section) has four intervals that divide the day, with a time specified for each interval, and the heating and cooling values that are used for this interval. See Figure 2 for the planned table design with text fields and their variable names.

Fire up Xcode. Choose File > New Project, and select "Window-based application" for the iOS application template. Name the project "TestTemp." This gives us the bare-minimum code scaffold and a MainWindow.xib file for the app. Like I did in the iOS tables article, I'll use a *UITableViewController* class to manage both the table display and user responses. So, select File > New File and add a *UIViewController* subclass with the option for *UITableViewController* subclass set. Name the file "TableTempControl." Don't set the "With XIB for user interface" option: We'll add any UI objects required into the MainWindow.xib file.

Speaking of that .xib file, open it with Interface Builder (IB) now. In IB, choose Tools > Library to bring up the Library window, then drag and drop *UITableViewController* class icon into the MainWindow.xib *window*. This adds the GUI support for the table. Now we have to connect it to the *UITableViewController* code in the project . This is done by adding the declaration and the property for *TableTempControl* to the TestTempAppDelegate.h file. Most of the details on how this is done can be found in the article "Displaying Tabular Data on iPhones" (http://drdobbs.com/mobility/229218541), so I'll skip the details here.

Once you are done, you should be able to build and run the project, and see the app present a table. The table presents empty rows for now, since it doesn't have any content to display.

The lack of table content lets us segue into making the class that implements the table's data model. In Xcode, choose File > New and select Objective-C class. Use the default "Subclass of NSObject" setting, and name the class "TableData." Add the code from Listing One into the TableData.h and TableData.m files for this class.

Listing One

```
#import "TableData.h"

@implementation TableData

// TableData.h
//
#import <Foundation/Foundation.h>


@interface TableData : NSObject {

    NSMutableArray *timeOfDays;
    NSMutableArray *heatStr;
    NSMutableArray *coolStr;
}

@property (nonatomic, retain) NSMutableArray *timeOfDays;
@property (nonatomic, retain) NSMutableArray *heatStr;
@property (nonatomic, retain) NSMutableArray *coolStr;

- (void) initTableData;


@end


// TableData.m
//
@synthesize timeOfDays;
@synthesize heatStr;
@synthesize coolStr;

-(void) initTableData {

    timeOfDays = [[NSMutableArray alloc] init];
     heatStr = [[NSMutableArray alloc] initWithObjects:@"76.0",
@"75.0", @"74.0", @"73.0", @"72.0", @"71.0", @"70.0", @"69.0",
@"68.0", @"67.0", @"66.0", @"65.0", nil];
     coolStr = [[NSMutableArray alloc] initWithObjects:@"45.0",
@"46.0", @"47.0", @"48.0", @"49.0", @"50.0", @"51.0", @"52.0",
@"53.0", @"54.0", @"55.0", @"56.0", nil];

}

-(void)dealloc {

    [coolStr release];
    [heatStr release];
    [timeOfDays release];

    [super dealloc];

}

@end
```

From the variable names, you can see that the class stores arrays of data values related to time (*timeOfDays*), heating (*heatStr*), and cooling (*coolStr*). Note also that the arrays are mutable ones, as the data will be subject to change. We have initialized both the heating and cooling arrays with data. We'll deal with the time array later.

Now let's add code to the *TableTempControl* class to utilize the data model just constructed. Listing Two shows the modifications made to this class's header file.

Listing Two

```
//
//  TableTempControl.h

#import <UIKit/UIKit.h>

#import "TableData.h"

@interface TableTempControl : UITableViewController {

    UITableViewCell *settingsCell;

    NSDateFormatter    *dateFormatter;
```

IN THIS ISSUE

Guest Editorial >>
Two-Way Tables in iOS >>
Java Meets Objective-C >>
Ternary Search Trees >>
Letters >>
Links >>
Table of Contents >>

```
    NSMutableArray    *timeOfDays;

    UITextField *timeField;
    UITextField *heatingField;
    UITextField *coolingField;

    TableData *settings;    // Object to store table data
     NSMutableArray *listOfSections; //  Content  for  section
                                     //  headers (titles)
    NSMutableArray *listOfRows; // Content for cell textfields

}

@property (nonatomic, retain) IBOutlet UITableViewCell *settingsCell;

@property (nonatomic, retain) NSDateFormatter *dateFormatter;

@property (nonatomic, retain) IBOutlet UITextField *timeField;
@property (nonatomic, retain) IBOutlet UITextField *heatingField;
@property (nonatomic, retain) IBOutlet UITextField *coolingField;

@end
```

Here, you can see the where *UITextField* objects are declared. Note that their property definitions include *IBOutlet*, which makes these objects appear in IB. There's also the declaration for the instance of *UITableViewCell*, *settingsCell*, which we'll use to make the table's content display. Another declaration defines the *TableData* instance for the data model, *settings*. Finally we declare an *NSDateFormatter* object to manage the time value. We could use strings to represent the time, but we might as well leverage the capabilities of a ready-made class for displaying dates and time. The arrays *listOfSections* and *listOfRows* will contain information about the table's section and rows. Moving on to the TableTempControl.m file, we add the following methods (Listing Three):

Listing Three

```
- (void) setUpTables {

    NSDictionary *tableSectionsDict;
    NSDictionary *currentWeekdayArrayDict;
    NSDictionary *currentSaturdayArrayDict;
    NSDictionary *currentSundayArrayDict;

    NSArray *sectionNamesArray = [NSArray arrayWithObjects:
       @"Weekdays", @"Saturday", @"Sunday", nil];
    tableSectionsDict = [NSDictionary dictionaryWithObject:
       sectionNamesArray forKey:@"Sections"];

// Add dictionary entry for content for weekdays section (section 0)
     NSArray *currentWeekdayArray = [NSArray arrayWithObjects:
       @"Morning", @"Day", @"Evening", @"Night", nil];
              currentWeekdayArrayDict    =    [NSDictionary
dictionaryWithObject:currentWeekdayArray forKey:@"Content"];

    // Add dictionary entry for content rows in Saturday section
    // (section 1)  —  copy content elements from weekdays array
    NSArray *currentSaturdayArray = [NSArray arrayWithArray:
      currentWeekdayArray];
    currentSaturdayArrayDict =
        [NSDictionary dictionaryWithObject:
         currentSaturdayArray forKey:@"Content"];

    // Add dictionary entry for content rows in Sunday section
    // (section 2)
    NSArray *currentSundayArray =
     [NSArray arrayWithArray:currentWeekdayArray];
       currentSundayArrayDict = [NSDictionary dictionaryWithObject:
       currentSundayArray forKey:@"Content"];

    // Set up lists that determine how the tables appear.
    // This list specifies the section headers.
    [listOfSections addObject:tableSectionsDict];

    // Lists of row content
    [listOfRows addObject:currentWeekdayArrayDict];
    [listOfRows addObject:currentSaturdayArrayDict];
    [listOfRows addObject:currentSundayArrayDict];

}

// Generate time schedule from schedule data in timeStr.
// timeStr data must conform to NSDate's short style format.
- (void)setUpSchedule {

     NSArray *timeStr = [[NSArray alloc] initWithObjects:@"1:00
AM", @"2:00 AM", @"3:00 PM", @"4:00 PM", @"6:00 AM", @"7:30 AM",
@"1:00 PM", @"8:15 PM", @"5:30 AM", @"9:45 AM", @"4:15 PM",
@"11:45 PM", nil];

    // Set up formatter -- no date display, short time format
```

```
    dateFormatter = [[NSDateFormatter alloc] init];
    [dateFormatter setTimeStyle:NSDateFormatterShortStyle];
    [dateFormatter setDateStyle:NSDateFormatterNoStyle];

    // Convert time strings into NSDate objects and store in array
    for (int i = 0; i < 12; i++) {

        [settings.timeOfDays addObject:
          [dateFormatter dateFromString:[timeStr objectAtIndex:i]]];

    } // end for
}
```

The *setUpTables* method generates the header and certain row information, which is stored in *NSArray* objects. It then builds the structure of arrays that hold the table's content. This arrangement factors the static data (the section headers and content for the cell's *textLabel*) apart from the dynamic data in the *TableData* class. The *setUpSchedule* method uses the *NSDateFormatterNoStyle* constant to configure the date data to display time values only. Next, in *viewDidLoad*, you add code to create an instance of the *TableData* and message its initialization method. Finally, you invoke the *setUpTables* and *setUpSchedule* to populate the data model. For the details, see the example program. When you build and run the app, you'll see the table broken into three sections: Weekdays, Saturday, and Sunday. The rows (cells) are empty, though. However, that's expected because there's only stub code for *cellForRowAtIndexPath*, which is responsible for creating the cell and updating its content. Let's correct that now.

Go into IB and create an empty object. Save it as TempCell.xib, and let IB add it to the project. Click on the File's *Owner* element in the TempCell.xib window and use the Identity Tool inspector (Tools > Identity Inspector) to change the *Class* option from *NSObject* to *TableTemp - Control*. This specifies that our table controller is the delegate object for our table cells. From the Library window, drag and drop a *UITableViewCell* into the TempCell.xib window. Control-click on the *Table View Cell* element in the window, and from the window that appears, click on the New Referencing Outlet choice. Drag a connection to the File's *Owner* element in the window. Choose *settingsCell* from the pop-up menu that appears. What you have just done is forge the links that specify the delegate (*TableTempControl*) for any message from the cell, and that the instance of the cell is referenced by *settingsCell*. This makes the cell visible to our code.

Let's start displaying some content in the cell. In *TableTempControl* class, locate the *cellForRowAtIndexPath* method, and modify its code like so (Listing Four):

Listing Four

```
// Customize the appearance of table view cells.
- (UITableViewCell *)tableView:(UITableView *)tableView
      cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell =
      [tableView dequeueReusableCellWithIdentifier:CellIdentifier];

    if (cell == nil) {
        [[NSBundle mainBundle] loadNibNamed:@"TempCell"
           owner:self options:nil];
        cell = settingsCell;
        settingsCell = nil;
    }

    // Fetch content for row label (day of week)
    NSDictionary *dictionary =
      [listOfRows objectAtIndex:indexPath.section];
    NSArray *array = [dictionary objectForKey:@"Content"];
    NSString *cellValue = [array objectAtIndex:indexPath.row];

    //Set text in label
    cell.textLabel.text = cellValue;

    return cell;
}
```

IN THIS ISSUE

Guest Editorial >>
Two-Way Tables in iOS >>
Java Meets Objective-C >>
Ternary Search Trees >>
Letters >>
Links >>
Table of Contents >>

Notice how if cell is *nil*, this method generates a new instance of it by loading the nib file TempCell.xib. Otherwise, the table will reuse existing instances of the cell for faster scrolling efficiency. The *listOfRows* array is first accessed to get the section array, and the row value references the content from the appropriate array. The text string obtained is copied into the text property for the cell's *textLabel*. Build and run the project. If the app dies horribly, you have botched completing a link in IB or have misspelled the nib file name. Make any corrections, and try again. If all goes well, you now have the phrases "Morning," "Day." "Evening," and "Night" repeating in the table's rows. Now the text fields have to be added.

Go back into IB, and drag the icon for a *UITextField* from the Library window and drop it into the TempCell.xib window. Do this two more times. Use IB to arrange the fields and their sizes similar to that in the figure. Or, you can cheat and look at the frame sizes and positions in the example project. In the TempCell.xib window, click on the disclosure triangle adjacent to the *Table View Cell* element to reveal the three *Round Style Text Field* elements you just added. Control-click on each, and first drag a connection from the delegate to the File's *Owner* element. Control-Click on each of them again and from New Referencing Outlet, drag a connection to the File's *Owner*. In the pop-up menu that appears, choose *timeField*, *heatingField*, and *coolingField* for each field, respectively. That should tie the text fields in the cell's UI to our code. Now revise *cellForRowAtIndexPath* as follows in Listing Five:

Listing Five

```
// Customize the appearance of table view cells.
- (UITableViewCell *)tableView:(UITableView *)tableView
    cellForRowAtIndexPath:(NSIndexPath *)indexPath {

    static NSString *CellIdentifier = @"Cell";

    UITableViewCell *cell =
       [tableView dequeueReusableCellWithIdentifier:CellIdentifier];

    if (cell == nil) {
      [[NSBundle mainBundle] loadNibNamed:@"TempCell"
          owner:self options:nil];
       cell = settingsCell;
       settingsCell = nil;

       cell.textLabel.backgroundColor = [UIColor clearColor];

       timeField.font =
         [UIFont systemFontOfSize:[UIFont labelFontSize]];
       heatingField.font =
         [UIFont systemFontOfSize:[UIFont labelFontSize]];
       coolingField.font =
         [UIFont systemFontOfSize:[UIFont labelFontSize]];

    }

    // Configure the cell's contents with regards to section and row
    // Fetch content for row label (day of week)
    NSDictionary *dictionary =
      [listOfRows objectAtIndex:indexPath.section];
    NSArray *array = [dictionary objectForKey:@"Content"];
    NSString *cellValue = [array objectAtIndex:indexPath.row];

    // Set text in cell label
    cell.textLabel.text = cellValue;

    // Set text in editable fields
    timeField.text = [NSString stringWithFormat:@"%@",
      [self.dateFormatter stringFromDate:[settings.timeOfDays
       objectAtIndex:((indexPath.section * 4) + indexPath.row)]]];
    heatingField.text = [NSString stringWithFormat:@"%@",
        [settings.heatStr objectAtIndex:((indexPath.section * 4) +
            indexPath.row)]];
    coolingField.text = [NSString stringWithFormat:@"%@",
        [settings.coolStr objectAtIndex:((indexPath.section * 4) +
            indexPath.row)]];

    return cell;
}
```

Study the code that extracts the relevant values from the arrays and places them into the text fields in the cell. The section value is used to

IN THIS ISSUE

Guest Editorial >>
Two-Way Tables in iOS >>
Java Meets Objective-C >>
Ternary Search Trees >>
Letters >>
Links >>
Table of Contents >>

calculate which object in the *timeOfDays*, *heatStr*, and *coolStr* arrays to access. The string fetched from the array is then placed in the text property of the corresponding text field. Items of interest are that we adjust the typeface size to ensure that text fits, and that the background color for the cell's *textLabel* is clear. The modification to the label's *backgroundcolor* property is required for the text fields to appear in the content area of the cell. I lost a few hours tracking down and resolving that quirk. Build and compile the app, and you should see the time and temperatures appear in the table.

But we're not done yet: If you click on a text field, the virtual keyboard appears and you can edit the text. However, it doesn't disappear when you click on the Return key. What is happening is that we have failed to designate a delegate for any *UITextField* action messages, and the message we want to catch is the one that informs us that the editing is done.

Making the *TableTempControl* class the delegate for the action messages is straightforward. Go to the *TableTempControl*'s header file, and revise the interface line to appear as so:

```
@interface TableTempControl :
   UITableViewController <UITextFieldDelegate> {
```

Now a method is required to handle the action message generated when the user taps the Return key to signal the editing is done. This message is handled, per the *UITextFieldDelegate_Protocol*, by the method *textFieldShouldReturn*. Quickly we can write:

```
- (BOOL)textFieldShouldReturn:(UITextField *)textField {

  [textField resignFirstResponder];
  return YES;

}
```

When a text field is tapped on, iOS makes the virtual keyboard that appears the first responder. That is, the keyboard object gets the first

crack at handling any subsequent taps on the screen. This flow of events through the view hierarchy remains in force until the text field resigns its first responder status. The code above does that by sending a *resignFirstResponder* message to text field instance referenced by this method's *textField* argument. When you tap the Return key now, the keyboard is dismissed.

However, we are still not done: If you scroll the table up and down so that the edited fields disappear off-screen briefly, when they reappear they display the old values. The reason is that while we've altered the string in the text field, we haven't stored the change in the data model yet. Let's look into doing that.

The Chicken and the Egg Problem

The solution seems simple enough: Just write the edited value from the text field back into its proper element in the array. However, the problem becomes apparent after a little thought: How do you know which array and which element to update? *UITextField* carries a lot of information about itself, but not the section or row of the cell that you tapped on. Nor does the cell have any information about which text field was tapped. How can we find this critical information?

One way to deal with the problem is, when *textFieldShouldReturn* executes, work your way up the view hierarchy using superview until you get the *view* class of the cell, and have it obtain the reference to the table. Knowing the cell, you can message the table to get its *indexPath*. You then use the *indexPath* to extract the row and column information:

```
UIView *view = (UIView *)[textField superview];
UITableViewCell *cell = (UITableViewCell *)[view superview];
UITableView *table = (UITableView *)[cell superview];
NSIndexPath *textFieldIndexPath =
  [table indexPathForCell:cell];
NSUInteger cellSection = [textFieldIndexPath section];
NSUInteger cellRow =[textFieldIndexPath row];
```

Then you can compare the *textField* object passed to this method and compare it to our date, cooling, and heating text field objects to find out which object was tapped. However, that is an awful lot of work to get this information. Fortunately, you can use the text field's view tag to expedite the process. When the cell is created or updated, you encode the section, row, and view information into the view's tag. When the *textFieldShouldReturn* method is invoked, just decode the value in tag of the *UITextField*'s view to get the section and row information.

Let's look at the cell update code in *cellForRowAtIndexPath* in Listing Six to see how the section, row, and field information are encoded into the tag value:

Listing Six

```
#define TIME_FIELD    1
#define HEATING_FIELD 4
#define COOLING_FIELD 2

// Configure the cell's contents with regards to section and row
    // Fetch content for row label (day of week)
    NSDictionary *dictionary =
      [listOfRows objectAtIndex:indexPath.section];
    NSArray *array = [dictionary objectForKey:@"Content"];
    NSString *cellValue = [array objectAtIndex:indexPath.row];

    // Set text in cell label
    cell.textLabel.text = cellValue;

// Set up time field display. Encode section and row into tag value.
    timeField.tag = (indexPath.section * 100) + (indexPath.row *
      10) + TIME_FIELD;

    // Drop in time content from timeOfDays array
    timeField.text = [NSString stringWithFormat:@"%@",
     [self.dateFormatter stringFromDate:[settings.timeOfDays
      objectAtIndex:((indexPath.section * 4) + indexPath.row)]]];

// Set up heating field display's tag.
```

IN THIS ISSUE

Guest Editorial >>
Two-Way Tables in iOS >>
Java Meets Objective-C >>
Ternary Search Trees >>
Letters >>
Links >>
Table of Contents >>

```
// Drop in content from heatStr array.
    heatingField.tag = (indexPath.section * 100) +
      (indexPath.row * 10) + HEATING_FIELD;
    heatingField.text = [NSString stringWithFormat:@"%@",
      [settings.heatStr objectAtIndex:((indexPath.section * 4) +
         indexPath.row)]];

    // Set up tag for cooling field.
    // Insert content from coolStr array.
    coolingField.tag = (indexPath.section * 100) +
      (indexPath.row * 10) + COOLING_FIELD;
    coolingField.text = [NSString stringWithFormat:@"%@",
     [settings.coolStr objectAtIndex:((indexPath.section * 4) +
         indexPath.row)]];

    return cell;
}
```

Here we encode the section information by multiplying its value by 100, while the row value is multiplied by ten, and the field value is just a single digit. These values are summed together to form the tag value.

When *textFieldShouldReturn* executes now, determining the section, row and field information of the object that was tapped just takes a few compare operations (Listing Seven):

Listing Seven

```
- (BOOL)textFieldShouldReturn:(UITextField *)textField {

    // Extract section, row, and tag to determine which text
    // field that was tapped
    NSUInteger row = (textField.tag % 100) / 10;
    NSUInteger section = (textField.tag / 100);
    NSUInteger inputFieldTag = (textField.tag - (section * 100) -
      (row * 10));

    // Handle specific field selection, based on textField's tag
    if (inputFieldTag == HEATING_FIELD) {
        [settings.heatStr replaceObjectAtIndex:((section * 4) +
           row) withObject:textField.text];

    }

    if (inputFieldTag == COOLING_FIELD) {
        [settings.coolStr replaceObjectAtIndex:((section * 4) +
          row) withObject:textField.text];
    }

    if (inputFieldTag == TIME_FIELD) {
        [settings.timeOfDays replaceObjectAtIndex:((section * 4)
           + row) withObject:[dateFormatter dateFromString:
           textField.text]];
    }

    // Update row with modified field. First get indexPath to row.
    // Will only update one row.
    NSUInteger fieldRowIndex[] = {section, row};
    NSIndexPath *rowIndexPath =
      [[NSIndexPath alloc] initWithIndexes:fieldRowIndex length:2];
    NSArray *rowArray =
      [[NSArray alloc] initWithObjects:rowIndexPath, nil];

    [self.tableView reloadRowsAtIndexPaths:rowArray
      withRowAnimation:UITableViewRowAnimationNone];
    // Do the update.

    // Release objects used t make index path
    [rowArray release];
    [rowIndexPath release];

    // Signal that editing is done, release first responder
    [textField resignFirstResponder];

    return YES;
}
```

With the section and row values in hand, it's easy to calculate the array index, then use *NSMutableArray*'s *replaceObjectAtIndex* method to write the new value into the correct element in the array. The other point of interest is that we use the section and row information to create an *indexPath*. This *indexPath* is used to specify the exact row that needs refreshing when invoking *reloadRowsAtIndexPaths*. For small tables, invoking *reloadData* to refresh the entire contents of the table is suitable. However, for large, complex tables you should use *reloadRowsAtIndexPaths* to keep the overhead required to perform the update to a minimum. You shouldn't see the update occur, which is fine: Such operations should be transparent to the user. If you want to confirm that the refresh takes place, mod-

ify the animation argument in the *reloadRowsAtIndexPaths* method and see what happens. This completes the construction of the two-way table. What you do with the updated data in the table's data model is up to you.

Final Touches (No Pun)

When entering text into *timeField*, if you enter anything other than a time consisting of digits, a colon, and AM or PM, the *dateFormatter* tosses an exception. You need to add code to do sanity checks on the edited text, or restrict what can be entered. One way to do this is to use the *UITextInputTraits_Protocol*'s *keyboardType* property to specify a keypad for numeric input only. This works for entering temperatures, but still does not solve the problem for *timeField*, because it requires some non-numeric characters. Alternatively, you can use the *UITextFieldDelegate_Protocol*'s *textFieldShouldBeginEditing* method to present a date picker rather than a keyboard. How to do that is the subject of an entire article in itself.

The iOS interface is a colorful one, and we can leverage that to make the table UI more useful. For example, we can make the *heatingField* view red and the *coolingField* view blue to help remind users as to which temperature they are changing.

To do this, go back into IB, open TempCell.xib, and select the *heatingField* object. Use the Attributes Inspector to change the background color of the view. If you run the app however, all you get is a colored rectangular border around the field. For whatever reason, adding color to a rounded text field is fraught with gotchas. First, to get color within the proper field, adjust the view's alpha channel to 0.75. This value strikes a reasonable balance between color intensity in the view while keeping the displayed text from becoming too faint. You still have that rectangular border, however. To fix this, add the Quartz Core libraries to the app by inserting the following header file into the TableTempControl.m file.

```
#import <QuartzCore/QuartzCore.h>
```

We will use the Quartz layer feature to trim the colored corners. Go to the file's *cellForRowAtIndexPath* method and revise the cell creation code to match that in Listing Eight:

Listing Eight

```
if (cell == nil) {
     [[NSBundle mainBundle] loadNibNamed:@"TempCell"
        owner:self options:nil];
     cell = settingsCell;
     settingsCell = nil;

     cell.textLabel.backgroundColor = [UIColor clearColor];

     timeField.font = [UIFont systemFontOfSize:
       [UIFont labelFontSize]];
     heatingField.font = [UIFont systemFontOfSize:
       [UIFont labelFontSize]];
     coolingField.font = [UIFont systemFontOfSize:
       [UIFont labelFontSize]];

     [[heatingField layer] setCornerRadius:8.0f];
     [[heatingField layer] setMasksToBounds:YES];
     [[coolingField layer] setCornerRadius:8.0f];
     [[coolingField layer] setMasksToBounds:YES];

 }
```

When you build the project and run the app, it should resemble that in Figure 2.

Now that you know what's required to write a two-way table, go forth and write business apps.

Assoicated code and files for this article can be downloaded at http://i.cmpnet.com/ddj/images/article/2011/0411/UITextField.zip.

— Tom Thompson is head of Proactive Support for embedded products at Freescale Semiconductor.

Java Meets Objective-C

While Java and Objective C might seem worlds apart, the programming languages and the platform libraries have a lot in common. An introductory application demonstrates this

By Genadiy Shteyman

I recently made the transition from writing enterprise Java applications to using Objective-C. I found that the transition would have been a lot easier had I the benefit of being able to find an article like this one.

So here I aim to help Java programmers quickly grasp the main features of Objective–C development. I will use a social networking application as an example to illustrate setting up a development environment for both languages. I'll cover creating basic objects, comparing how the MVC design pattern works in both languages, and show you how data is stored and accessed in both languages.

iPhone Development: Where to Begin

To start developing iPhone apps, a Mac computer is required. The latest Mac OS X version 10.6 typically includes a copy of the Xcode IDE and suite of tools for developing iPhone software using Objective-C (see Figure 1). In November 2010. Apple released the long-awaited SDK 4.2, which included a rich set of frameworks and features to build interactive iPhone apps. Xcode also contains a Simulator, which allows you to run your program and see how it would look on the device.

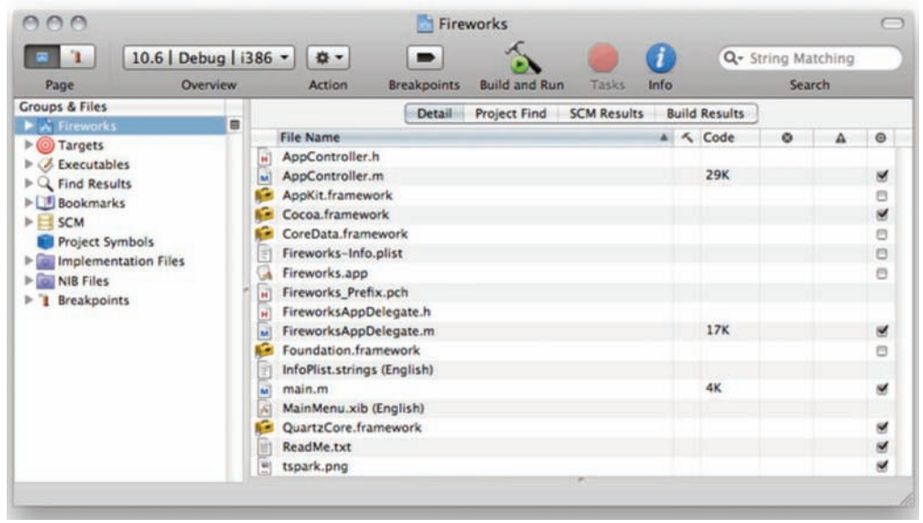


Figure 1: Xcode Development Environment, Project View.

Objective–C is the main development language for iPhone apps. Luckily for Java developers, it is fully object-oriented and utilizes the same concepts as all other OO languages — inheritance, polymorphism, and encapsulation. To define a class (called "module" or .m file in Objective-C), you first have to define an interface (that is a header or .h file), and include it in your class implementation.

As an example, let's look at a social networking application, which would allow you to stay in touch with your friends. A friend's profile will be stored in memory as *FriendProfile* object. It'll contain four fields: your friend's name, city, country, and contact number; as in Listing One.

Listing One

```
//declare FriendProfile.h interface file

#import <UIKit/UIKit.h>

@interface FriendProfile: NSObject
{
}

@property (nonatomic, retain) NSString * name;
@property (nonatomic, retain) NSString * country;
@property (nonatomic, retain) NSString * city;
@property (nonatomic, retain) NSString * phoneNbr;
@end

//FriendProfile.m implementation file
#import "FriendProfile.h"

@implementation FriendProfile

@synthesize name;
@synthesize country;
@synthesize city;
@synthesize phoneNbr;

@end
```

In this example, interface *FriendProfile: NSObject* means that we define an interface called *FriendProfile*. It inherits its functionality from the base class called *NSObject* (colon notation specifies super class). *NSObject* is a root class that most Objective-C classes inherit from, similar to the *Object* class in Java. Next, we allocate in memory necessary variables of *NSString* type (equivalent to Java's *String* class type) to store our friend's data fields. The implementation class will then use *@synthesize* keyword to automatically create getter and setter methods.

You can create the *FriendProfile* object as follows:

```
FriendProfile * profile = [[FriendProfile alloc] init];
```

Here, *alloc* and *init* act like the *new* keyword in Java, allocating *FriendProfile* object in memory. Next, you can assign some values to object fields.

```
[profile setName:@"Albert"];
[profile setCountry:@"USA"];
[profile setCity:@"Houston"];
[profile setPhoneNbr:@"123-456-789"];
```

Or, you can simply write:

```
profile.name = @"Albert";
profile.country = @"USA";
profile.city = @"Houston;
profile.phoneNbr = @"123-456-789";
```

To fully understand Objective-C syntax and features, I would recommend reading this excellent language reference on Apple's developer site at http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/ObjectiveC/Introduction/introObjectiveC.html.

From Java

In Java, the *FriendProfile* class would be very similar to its Objective-C counterpart, see Listing Two:

Listing Two

```
public class FriendProfile {
```

IN THIS ISSUE

Guest Editorial >>
Two-Way Tables in iOS >>
Java Meets Objective-C >>
Ternary Search Trees >>
Letters >>
Links >>
Table of Contents >>

```
//no direct access to variables, only via getters and setters
private String name;
private String country;
private String city;
private String phoneNbr;

//declare public constructor
public FriendProfile(){}

//define getters and setters
public String getName() {
      return name;
}

pubilc void setName(){
      this.name = name;
}
public String getCountry() {
      return country;
}

     pubilc void setCountry(){
      this.country = country;
}


public String getCity(){
      return  city;
}

public void setCity(){
      this.city = city;
}

public String getPhoneNbr(){
      return  phoneNbr;
}

public void setPhoneNbr(){
      this.phoneNbr = phoneNbr;
}

}
```

Listing Two provides analogous fields, but getters and setters must be written out explicitly. Now let us see how we would add a new friend to our contact list (see Listing Three).

Listing Three

```
//import standard HTTP servlet packages
import javax.servlet.*;
import javax.servlet.http.*;

import java.io.*;

//import our classes
import com.model.ProfileManager;
import com.vo.FriendProfile;


public class FriendlyServletController extends HttpServlet
     implements Servlet
{

      protected void doPost(HttpServletRequest request,
                            HttpServletResponse response)
          throws ServletException, IOException
      {
            //set reponse type as HTML
            response.setContentType("text/html");

       String name       = request.getParameter("name");
       String country    = request.getParameter("country");
       String city       = request.getParameter("city");
       String phoneNbr = request.getParameter("phoneNbr");


      //normally you have to validate browser-originated requests
       validateParameters(name, country, city, phoneNbr);

            FriendProfile newProfile = new FriendProfile();
        newProfile.setName(name);
        newProfile.setCountry(country);
            newProfile.setCity(city);
            newProfile.setPhoneNbr(phoneNbr);
   ProfileManager().getInstance().addToContacts(newProfile);
   response.getWriter().write("Your friend is added to contacts");
      }
}
```

In this example, the *FriendlyServletController* class extends behavior from *HTTPServlet*, which is the Java server-side component class that processes requests coming from the browser. When a user logs in to your website and decides to add a friend, he would specify the data

from the friend in HTML form fields. When a user submits the form, the Servlet receives and validates the request parameters and creates an object of *FriendProfile* type to store the data in memory.

The *ProfileManager* class stores your *FriendProfile* object in the database.

MVC on the iPhone

The Model-View-Controller (MVC) design pattern frequently used in Java Web apps is present in iPhone development as well. If you look up the *UIViewController* class definition in the iOS Reference Library (http://developer.apple.com), you will see this: "*UIViewController* class provides the fundamental view-management model for iPhone applications…You use each instance of *UIViewController* to manage a view hierarchy." *UIViewController* is essentially a controller component that invokes business logic and updates a client's view.

When you create an object of *UIViewController* type in Xcode, you can choose to create it with an XIB file. This is a special type of Xcode file

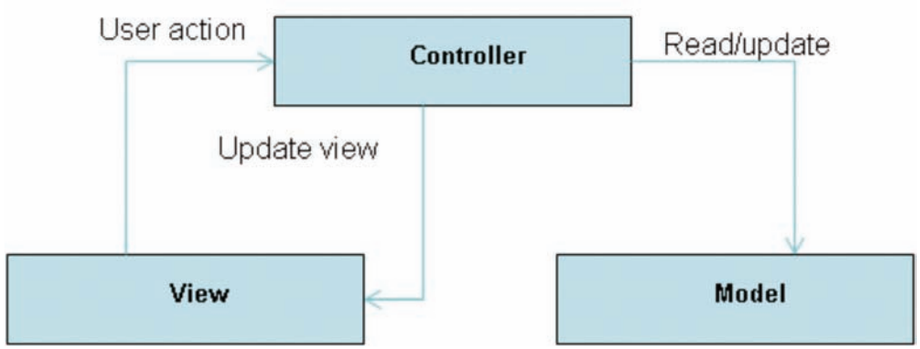


Figure 2: Model-View-Controller Design Pattern.

IN THIS ISSUE

Guest Editorial >>
Two-Way Tables in iOS >>
Java Meets Objective-C >>
Ternary Search Trees >>
Letters >>
Links >>
Table of Contents >>

that defines the graphical user interface, or view, which can contain various controls such as buttons, table views, labels, etc.

Going back to our social networking example, suppose you have added several of your friends to the contact list. Now you would like to tap on a specific friend entry and see his detailed information. This functionality is achieved by defining our controller class as in Listing Four.

Listing Four

```
// FriendProfileViewController.h
#import "FriendProfile.h"

// define our custom controller to inherit from UIViewController
class
@interface FriendProfileViewController: UIViewController
{
}
//this property will hold our friend info retrieved from database
@property (nonatomic, retain) FriendProfile * friendProfile;
...


// FriendProfileViewController.m
@implementation FriendProfileViewController

@synthesize friendProfile;

...
//allocate and initialize controller object, typically in another
controller or
//UIApplicationDelegate object at the begging of your program:


FriendProfileViewController *profileController = [[FriendProfile-
ViewController alloc] initWithNibName:
@" FriendProfileViewController " bundle:nil];

//add new view to the parent view and make it appear on the screen
[window addSubview: profileController.view];
[window makeKeyAndVisible];
```

In this code, we create an instance of *FriendProfileViewController* and initialize it with our custom *View Bundle*, which is responsible for showing friend's information.

Alloc and *initWithNibName* are methods used to instantiate the controller class. They are equivalent to Java's *new* keyword.

The *Model* kicks in when the view is actually loaded. Every controller has a few lifecycle methods inherited from the parent class. *UIViewController*. *ViewdidLoad* method is one of them. It's responsible for additional setup, after the view is loaded. It calls *model* to bring a friend's information from the database and update the view. In the simplest case, our view will contain a series of labels, or objects of *UILabel* type, which can be set to any text (such as a friend's name or phone number) at application runtime. The user immediately sees the friend's information updated.

MVC in Java

Next, I will illustrate how to show a friend's details on a browser's screen using Java on the backend. Let's slightly modify our *FriendlyServletController*, which, in fact, is our *Controller*; see Listing Five:

Listing Five

```
public class FriendlyServletController extends HttpServlet
     implements Servlet
{

    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response)
          throws ServletException, IOException
    {
      response.setContentType("text/html");

//receive action requested by user
      String action = request.getParameter("requestedAction");

      if (action!=null && action.equals("showFriendProfile")){
//look up your friend details by name
          String name = request.getParameter("name");

    FriendProfile newProfile = new FriendProfile ();
    newProfile.setName(name);
```

IN THIS ISSUE

Guest Editorial >>
Two-Way Tables in iOS >>
Java Meets Objective-C >>
Ternary Search Trees >>
Letters >>
Links >>
Table of Contents >>

```
FriendProfile friendProfile =
    ProfileManager().getInstance().lookupContact(newProfile);

    //store friend's information and invoke the View
    request.setAttribute("retrievedProfile", friendProfile);
    request.getRequestDispatcher.forward("DisplayFriend.jsp");
}
       ...
```

In this example, *FriendlyServletController* receives the HTTP form-originated request for specific action called "*showFriendProfile*." Our model is a *ProfileManager* object that is responsible for looking up specific records from the database by friend name. A database record comes back to the *Controller* in the form of a *FriendProfile* object, which is stored and forwarded to our *View*, a *DisplayFriend.jsp* page. This view will show information that pertains to our specific friend.

iPhone: Database Access

Complex applications contain storage of some kind, generally a database. Apple recommends accessing the database via the Cocoa API framework called "Core Data." The Core Data framework directly interfaces with the SQLite database, (which, in our example, is running on your mobile device). Core Data hides the complexity of dealing with SQL. Instead, it provides the very handy *NSManagedObject* interface, which allows you to directly manipulate an entity object's instance fields. These fields will automatically be stored in the database. Another convenience of the Core Data Framework is that database table creation (as well as adding relations and constraints to the tables) can all be done within the Core Data User Interface provided by XCode.

Let's go back to our social networking app and show how to fetch a friend's profile from the database. We will use SQLite and the Core Data API, but first we have to slightly modify our *FriendProfile* class, as in Listing Six:

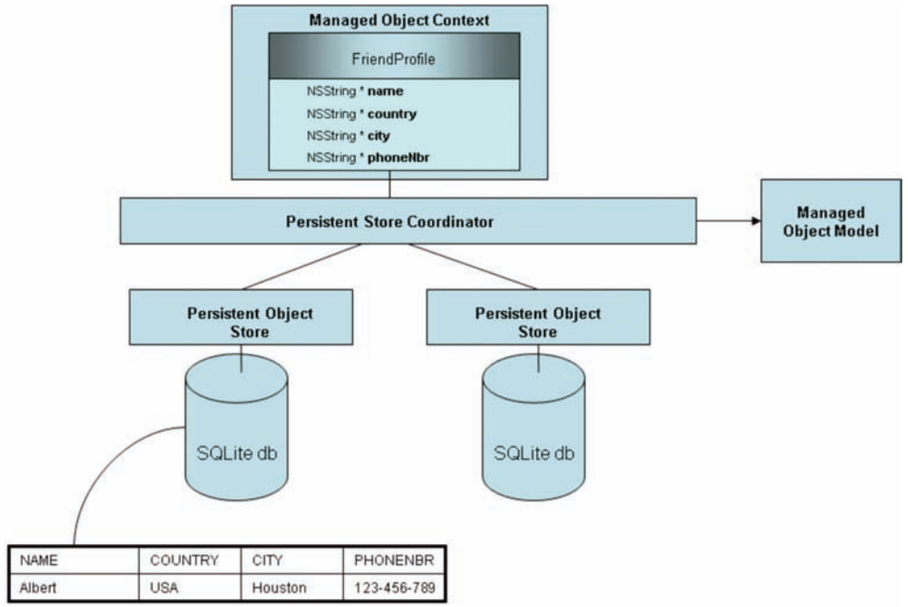


Figure 3. Core Data Stack Diagram.

Listing Six

```
//FriendProfile.h interface file
//modify our FriendProfile object to be CoreData Compliant

#import <UIKit/UIKit.h>

//need to import Core Data framework
#import <CoreData/CoreData.h>

//extend from NSManagedObject in order to inherit Managed Object
//behavior

@interface FriendProfile: NSManagedObject
{
}
@property (nonatomic, retain) NSString * name;
@property (nonatomic, retain) NSString * country;
@property (nonatomic, retain) NSString * city;
@property (nonatomic, retain) NSString * phoneNbr;
@end
```

IN THIS ISSUE

Guest Editorial >>
Two-Way Tables in iOS >>
Java Meets Objective-C >>
Ternary Search Trees >>
Letters >>
Links >>
Table of Contents >>

```
//FriendProfile.m implementation file
//must import header file
#import "FriendProfile.h"

@implementation FriendProfile
@dynamic name;
@dynamic country;
@dynamic city;
@dynamic phoneNbr;
@end
```

The difference between the *FriendProfile* class definition here and the one in Listing One is that here, I've included the Core Data framework's header file. Also, our class now extends from *NSManagedObject*, which gives it all the basic behavior required of a Core Data object.

Accessors, used in Core Data's *NSManagedObject* class, are created dynamically at runtime. If you want to declare and use properties of the *FriendProfile* class, but want to avoid warnings about methods missing at compile time, you can use the *@dynamic* directive, which is shown in the implementation class, instead of *@synthesize*.

Using the *NSManagedObject* API is a bit complex, but it becomes intuitive once you understand it. Listing Seven is an example method that fetches a friend's profile from a database table, called FRIENDPROFILE. The table contains four columns: NAME, COUNTRY, CITY, and PHONENBR.

Listing Seven

```
- (NSManagedObject *) getFriendProfileObjectbyName:(NSString
*)name
{
//create sort descriptors to specify preferred sort order on
//table's "name" column
NSSortDescriptor *sortDescriptor1 = [[NSSortDescriptor alloc]
initWithKey:@"name" ascending:YES];
NSArray *sortDescriptors = [[NSArray alloc]
           initWithObjects:sortDescriptor1,nil];

//specify our filter condition being friend's name
NSPredicate *predicate =
[NSPredicate predicateWithFormat:@"name == %d", name];

//store our friend's profile(s) in array object
NSMutableArray *friends = [[NSMutableArray alloc] init];

//fetch our friend's profile from database table
friends = [[DatabaseController
fetchMutableManagedObjectsForEntity:@"FriendProfile" withPredi-
cate:predicate andSortDescriptors:sortDescriptors];

[sortDescriptor1 release];
[sortDescriptors release];
[predicate release];

if ([friends count] > 0) {
    return [friends objectAtIndex:0];
}
return nil;
}
```

Our *getFriendProfileObjectbyName* method receives the friend's name as an argument. By using the Core Data API, we specify which table is queried as well as the predicates and sorting requirements. As a result, this SQL statement executes behind the scenes:

```
SQL>select * from FriendProfile where name = "Albert";
```

fetchMutableManagedObjectsForEntity, called on class *DatabaseController,* is a custom method that contains a lot of "boilerplate" code (unfortunately, such code exists in this framework) to properly initialize *ManagedObjectContext*, etc. You can easily copy this standard code from the reference at http://developer.apple.com/library/mac/#DOCUMENTATION/DataManagement/Conceptual/CoreDataSnippets/Articles/fetching.html.

Once you fetch the *FriendProfile*, you can retrieve its properties in this manner:

```
NSString* name =        FriendProfile.name;
NSString* country =     FriendProfile.country;
NSString* city =        FriendProfile.city;
NSString* phoneNbr =    FriendProfile.phoneNbr;
```

Overall, Core Data is a useful feature. It allows you to graphically define entities (tables) and relations; and generates corresponding Managed Objects dynamically. Also, there is no need to deal with complex SQL statements. On the negative side, there is a significant amount of "boilerplate" code, which you have to carefully place and test.

Java: Database Access

There are multiple Java persistence frameworks. In my opinion, Hibernate is the Java framework that most closely resembles the Core Data API. Hibernate follows an Object-Relational Mapping (ORM) paradigm. That is, it allows you to persist object data into a relational database by simply setting fields on the object, which is directly mapped to a particular database table. The mapping can be done using either XML files or metadata annotations introduced in Java 5. Listing Eight shows an example of the former.

Listing Eight

```
<hibernate-mapping>
  <class name="com.vo.FriendProfile" table=" FRIENPROFILE ">
  <property name="name">
     <column name="NAME" />
  </property>

  <property name="country">
    <column name="COUNTRY"/>
  </property>

<property name="city">
    <column name="CITY"/>
  </property>
  <property name="phoneNbr">
    <column name="PHONENBR"/>
  </property>
 </class>
</hibernate-mapping>
```

IN THIS ISSUE

Guest Editorial >>
Two-Way Tables in iOS >>
Java Meets Objective-C >>
Ternary Search Trees >>
Letters >>
Links >>
Table of Contents >>

In this example, the *FriendProfile* object from Listing Two is mapped to a database table with the same name, which is a conventional way of dealing with data mapping. Four object fields are directly mapped to four table columns with the same names. This mapping allows Hibernate to generate proper SQL behind the covers.

Another configuration file, called hibernate.cfg.xml, contains database connection configuration details, such as database URL, drivers, login, and password; see Listing Nine.

Listing Nine

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;
import org.hibernate.Transaction;
import org.hibernate.cfg.Configuration;

...

Session session = null;
Transaction tx = null;
FriendProfile profile = null;


try{
    // Read Hibernate configuration file
    SessionFactory sessionFactory = new
          Configuration().configure().buildSessionFactory();

    // A specific session object is obtained
    session = sessionFactory.openSession();

    // A new database transaction is started
    tx = session.beginTransaction();

    // FriendProfile object is fetched from the store
    profile = (FriendProfile) session.get
             (FriendProfile.class, new String("Albert"));

    tx.commit();
    session.close();
}
catch(Exception e){}

return profile;
```

In Listing Nine, we have imported all the necessary Hibernate libraries. Then we created a Hibernate Session and started the transaction. Next, we programmatically retrieved the *FriendProfile* object by simply issuing a *get* method on the *Session* object, and passing the expected object type and query filter — your friend's name. The same query, which we saw in the Core Data example, will run here as well, and return back a *FriendProfile* object with all profile fields filled in.

Conclusion

Despite differences in language syntax and underlying runtime platforms, iPhone development with Objective-C and Web Application Development using Java share common traits:

- Both languages are object-oriented
- Both languages utilize the same Design Patterns, e.g., MVC
- Both languages use similar Database Access techniques , e.g., ORM

However, there are things that Java developers have to watch for in Objective-C:

- **Object creation:** Java objects are created at runtime using the *new* keyword. There is no explicit memory allocation that a Java programmer has to worry about. In Objective-C, an object can be created using one of three keywords: *alloc*, *new*, or *copy*. Each one of these, similar to the *retain* method, increments the *retain* count of the object. *Retain* count shows how many pointers to the object exist, and whether it can be reclaimed by the Memory Manager.

IN THIS ISSUE

Guest Editorial >>
Two-Way Tables in iOS >>
Java Meets Objective-C >>
Ternary Search Trees >>
Letters >>
Links >>
Table of Contents >>

- **Object destruction:** Java memory management is greatly simplified with garbage collecting. Java objects that reference other objects are organized as an object graph sitting in JVM's heap memory. The moment an object becomes unreachable (not referenced by other objects in a graph), it becomes a candidate for garbage collection, usually after its reference is set to null, or it falls out of its method scope.

  Objective-C has a memory manager, not a garbage collector. If you allocate an object using one of the four aforementioned techniques, you must release the object by calling the *release* method on this object.

  Calling the *release* method decreases *retain* count. When the count reaches zero, the object is sent a *dealloc* method; at which point, the object can release any additional memory and call *dealloc* on its super class. Failure to release or auto release an object causes memory leaks and unpredictable behavior in the future.

- **Over-release and premature deallocation:** Java programmers are immune to these errors thanks to garbage collection. Objective-C programmers have to remember not to release more than they retain. Over-releasing method calls on already deallocated objects can cause the application to crash.

What is clear from these examples is that Objective-C and Java have many syntactic and semantic elements in common. Moreover, the way in which problems are solved tend to use similar components. If you approach Objective-C with this perspective and bear in mind the differences highlighted in this article (and in the "Additional Differences" section), you're likely to find the transition fairly smooth.

Additional Differences

iPhone vs. Java comparison:

1. Objective-C uses the *nil* keyword to specify that an object reference does not point to any object. This is equivalent to Java's *null*. However, the subtle difference is that Objective-C allows you to call methods on *nil* objects, simply returning *nil*. Java will throw a *NullPointerException* that can halt program execution.

2. Objective-C uses the *self* keyword to refer to fields and method within the object *[self fetchLatestData:dataLocation];* whereas Java would use the *self* keyword for the same thing.

3. Objective-C uses distinct data type called *id* to declare a general type of object. This comes handy when you do not know at compile time what type of object comes back from a method, so we can receive *id*, and cast it to necessary type:

```
- (void) categoryButtonPressed:(id)sender
{
      NSUInteger whichButton = ((UIButton*)sender).tag;
}
```

   Java utilizes the *Object* type for the same thing.

4. The Objective-C method that receives multiple arguments is broken into multiple parts:

```
- (NSString*) getListOfFriends: (NSString*) name
forCity:(NSString*) city
withPhoto: (UIImage*) photo
```

IN THIS ISSUE

Guest Editorial >>
Two-Way Tables in iOS >>
Java Meets Objective-C >>
Ternary Search Trees >>
Letters >>
Links >>
Table of Contents >>

The actual method name is *getCategory:forPublication:withImage*: and each argument is specified after the colon. Java simplifies the method signature syntax:

```
public String getListOfFriends (name, city, photo);
```

In Java, a method name must be a single word, followed by a list of arguments separated by commas.

Useful Resources

1. A Tour of Xcode: http://developer.apple.com/library/ios/#documentation/DeveloperTools/Conceptual/A_Tour_of_Xcode/000-Introduction/qt_intro.html#//apple_ref/doc/uid/TP30000890-CH204-TPXREF101
2. Objective-C Programming Language: http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/ObjectiveC/Introduction/introObjectiveC.html
3. NSManagedObject Class Reference: http://developer.apple.com/library/mac/#documentation/Cocoa/Reference/CoreDataFramework/Classes/NSManagedObject_Class/Reference/NSManagedObject.html
4. Wikipedia Core Data entry: http://en.wikipedia.org/wiki/Core_Data
5. Memory Management Programming Guide: http://developer.apple.com/library/ios/#documentation/Cocoa/Conceptual/MemoryMgmt/MemoryMgmt.html#//apple_ref/doc/uid/10000011i

— Genadiy Shteyman has a B.S. in Computer Science and M.S. in Management of Information Systems. He is a vice president in the private bank division at a major financial institution.

IN THIS ISSUE

Guest Editorial >>
Two-Way Tables in iOS >>
Java Meets Objective-C >>
Ternary Search Trees >>
Letters >>
Links >>
Table of Contents >>

From the Vault

Ternary Search Trees

Back in 1998, **Dr. Dobb's Journal** published the work of noted computer scientists Jon Bentley and Bob Sedgewick as they posed the question "When you have to store a set of strings, what data structure do you use?" Their suggested answer advised you to start with ternary search trees, which combine the time efficiency of digital tries with the space efficiency of binary search trees. This article, pulled from our vault, discusses the use of ternary search trees.

— DDJ

By Jon Bentley and Bob Sedgewick

When you have to store a set of strings, what data structure should you use? You could use hash tables, which sprinkle the strings throughout an array. Access is fast, but information about relative order is lost. Another option is the use of binary search trees, which store strings in order, and are fairly fast. Or you could use digital search tries, which are lightning fast, but use lots of space.

In this article, we examine ternary search trees, which combine the time efficiency of digital tries with the space efficiency of binary search trees. The resulting structure is faster than hashing for many typical search problems, and supports a broader range of useful problems and operations. Ternary searches are faster than hashing and more powerful, too.

We described the theory of ternary search trees at a symposium in 1997 (see "Fast Algorithms for Sorting and Searching Strings," by J.L. Bentley and R. Sedgewick, *Proceedings of the 8th Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1997, http://portal.acm.org/citation.cfm?id=314161). In this article, we'll concentrate on what the data structure offers working programmers. *Algorithms in C*, Third Edition, by Robert Sedgewick (Addison-Wesley, 1998, http://is.gd/F1dldJ) provides yet another view of ternary search trees. For more information (including all the code in this article and the driver software), refer to http://drdobbs.com/database/articleID/184410528/sourcecodeID/30201315 or www.cs.princeton.edu/~rs/strings.

A Grove of Trees

Figure 1 is a binary search tree that represents 12 common two-letter words. For every node, all nodes down the left child have smaller values,

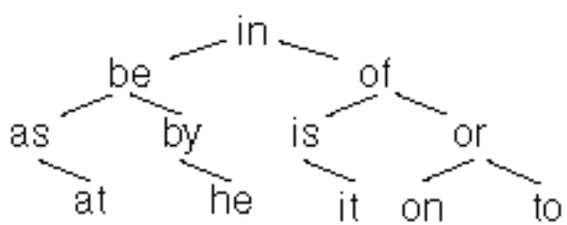


Figure 1: A binary search tree for 12 words.

while all nodes down the right child have greater values. A search starts at the root. To find the word "on," for instance, we compare it to "in" and take the right branch. We take the right branch at "of" and the left branch at "or," and then arrive at "on." Every comparison could access each character of both words.

Digital search tries store strings character by character. Figure 2 is a tree that represents the same set of 12 words; each input word is shown beneath the node that represents it. (Two-letter words lead to prettier pictures; all structures that we'll see can store variable-length words.) In a tree representing words of lowercase letters, each node has 26-way branching (though most branches are empty, and not shown in Figure 2). Searches are very fast: A search for "is" starts at the root, takes the "i" branch, then the "s" branch, and ends at the desired node. At every node, we access an array element (one of 26), test for null, and take a branch. Unfortunately, search tries have exorbitant space requirements: Nodes with 26-way branching typically occupy 104 bytes, and 256-way nodes consume a kilobyte. Eight nodes representing the 34,000-character Unicode Standard would together require more than a megabyte!

Ternary search trees combine attributes of binary search trees and digital search tries. Like tries, they proceed character by character. Like binary search trees, they are space efficient, though each node has three children, rather than two. A search compares the current character in the search string with the character at the node. If the search character is less, the search goes to the left child; if the search character is greater, the search goes to the right child. When the search character is equal, though, the search goes to the middle child, and proceeds to the next character in the search string.

Figure 3 is a balanced ternary search tree for the same set of 12 words. The low and high pointers are shown as solid lines, while equal pointers are shown as dashed lines. Each input word is shown beneath its terminal node. A search for the word "is" starts at the root, proceeds down the equal child to the node with value "s," and stops there after two compar-

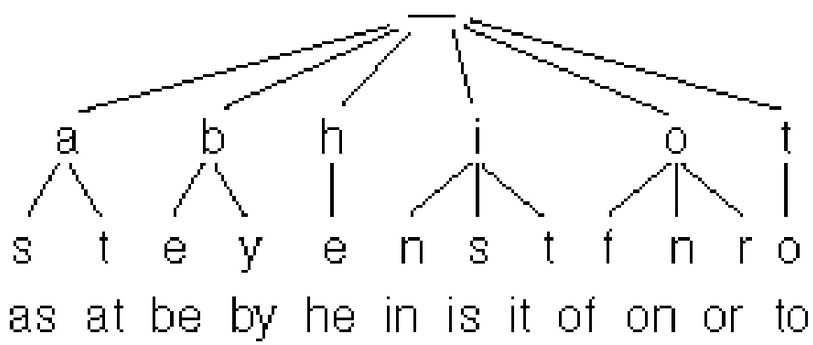


Figure 2: A digital search trie for 12 words.

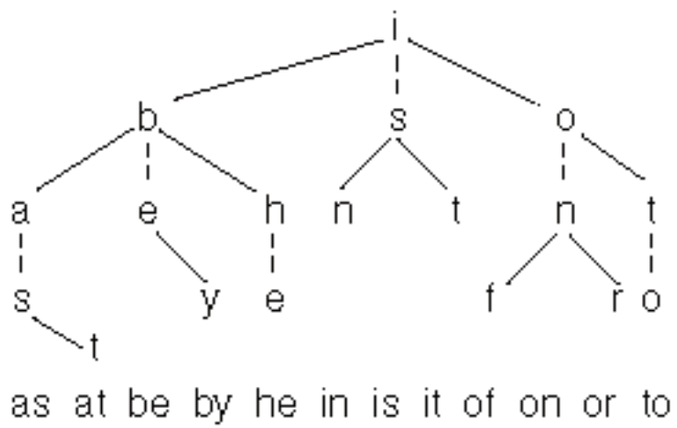


Figure 3: A ternary search tree for 12 words.

isons. A search for "ax" makes three comparisons to the first letter ("a") and two comparisons to the second letter ("x") before reporting that the word is not in the tree.

The idea behind ternary search trees dates back at least as far as 1964. In "Randomized Binary Searching with Tree Structures" (*Communications of the ACM*, March 1964, http://portal.acm.org/citation.cfm?id=363987), H.A. Clampett sketched a primitive version of the structure. Computer scientists have proven many theorems about the trees; for instance, searching for a string of length *k* in a ternary search tree with *n* strings will require at most *O(log n+k)* comparisons.

The C Structure

As far as we can tell, previous authors have viewed ternary search trees as a theoretical structure for proving theorems. We were delighted to find that the trees also lead to practical computer programs. Although we've chosen to illustrate the data structure in C, we could have just as easily chosen C++, Java, or other languages.

Each node in a ternary search tree is represented by the following structure:

```
typedef struct tnode *Tptr;
typedef struct tnode {
   char splitchar;
   Tptr lokid, eqkid, hikid;
} Tnode;
```

The value stored at the node is *splitchar*, and the three pointers represent the three children. The root of the tree is declared to be *Tptr* root. We will represent every character in each string, including the null character that terminates it.

Membership Searching

We begin with a recursive version of the search function. It returns 1 if string *s* is in the subtree rooted at *p*, and 0 otherwise; it is originally called as *rsearch(root, s)*:

```
int rsearch(Tptr p, char *s)

{   if (!p) return 0;
    if (*s < p->splitchar)
         return rsearch(p->lokid, s);
    else if (*s > p->splitchar)
        return rsearch(p->hikid, s);
    else {
        if (*s == 0) return 1;
        return rsearch(p->eqkid, ++s);
   }
}
```

The first *if* returns 0 if the search has run off the end of the tree. The next two *if* statements take the low and high branches as appropriate. The final *else* branch returns 1 if the current character is the end-of-string character 0, and otherwise moves to the next input character and to the equal branch.

Some programmers might feel more comfortable with the iterative version of the search function (which has only one argument):

```
int search(char *s)
{   Tptr p;
    p = root;
    while (p) {
       if (*s < p->splitchar)
           p = p->lokid;
       else if (*s == p->splitchar) {
          if (*s++ == 0)
              return 1;
          p = p->eqkid;
      } else
          p = p->hikid;
    }
    return 0;
}
```

This is substantially faster on some compilers, and we'll use it in later experiments. On most optimizing compilers, though, the runtime of the recursive version is within a few percent of the iterative version.

Both versions of the search use a pattern that we will see repeatedly: If the search character is less, go to *lokid*; if it is greater, go to *hikid*; if it is equal, go to the next character and *eqkid*. The following code illustrates that shorter is not always cleaner, simpler, faster, or better:

```
int rsearch2(Tptr p, char *s)

{ return (!p ? 0 : (
        *s == p->splitchar
      ? (*s ? rsearch2(p->eqkid, ++s) : 1)
      : (rsearch2(*s < p->splitchar ?
              p->lokid : p->hikid, s))
    ));
}
```

Inserting a New String

The *insert* function inserts a new string into the tree, and does nothing if it is already present. We insert the string *s* with the code *root = insert(root, s);*. The first *if* statement detects running off the end of new node, initializes it, and falls through to the standard case. Subsequent code takes the appropriate branch, but branches to *eqkid* only if characters remain in the string.

```
Tptr insert(Tptr p, char *s)
{ if (p == 0) {
       p = (Tptr) malloc(sizeof(Tnode));
       p->splitchar = *s;
       p->lokid = p->eqkid = p->hikid = 0;
  }
  if (*s < p->splitchar)
       p->lokid = insert(p->lokid, s);
  else if (*s == p->splitchar) {
       if (*s != 0)
           p->eqkid = insert(p->eqkid, ++s);
  } else
       p->hikid = insert(p->hikid, s);
  return p;
}
```

This code is short but subtle, and worth careful study. The resulting tree stores the strings themselves, but no other information. Real symbol tables store additional data with each string. To illustrate this problem, we'll store a pointer to every string in the tree; this data will be used by later search algorithms. We could add a new info field to each node, but that would be wasteful. Instead, we'll exploit the fact that a node with a null *splitchar* cannot have an *eqkid*, and store the data in that field. We could make a union that contains the two possible pointers, but that would be syntactically clumsy (we would have to refer to the union at each reference). We will instead use the sleazy hack of coercing a string into a *Tptr*. The code inside **s == p->splitchar test*, therefore, becomes the following:

```
if (*s == 0)
    p->eqkid = (Tptr) insertstr;
else
    p->eqkid = insert(p->eqkid, ++s);
```

Faster Insertion Functions

We can improve insertion in two different ways. We'll start by tuning the insertion code, and consider different orders in which we can insert the nodes into the tree.

A major cost of the *insert* function is calling *malloc* to create each node. The function *insert2* in the demo program (at http://drdobbs.com/database/articleID/184410528/sourcecodeID/30201315) uses the ancient C technique of allocating a buffer of nodes and dealing them out as needed. Profiling shows that this eliminates most of the time spent in storage allocation. We measured the time to insert the 234,936 words in a dictionary file (one word per line, for a total of 2,486,813 characters). Table 1 lists the runtimes in seconds on three 200-MHz machines, with the compilers optimizing code for speed. The insert3 function (also available at http://drdobbs.com/database/articleID/184410528/source-

codeID/30201315) uses other common techniques to reduce the runtime even further: transforming recursion to iteration, keeping a pointer to a pointer, reordering tests, saving a difference in a comparison, and splitting the single loop into two loops. These changes are beneficial on all machines, and substantial on the SPARC.

Better Insertion Orders

In what order should you insert the nodes into a tree? No matter in what order you insert the nodes, you end up with the same digital search trie — the data structure is totally insensitive to insertion order. Binary search trees are at the opposite end of the spectrum: If you insert the nodes in a good order (middle element first), you end up with a balanced tree. If you insert the nodes in sorted order, the result is a long skinny tree that is very costly to build and search. Fortunately, if you insert the nodes in random order, a binary search tree is usually close to balanced.

Ternary search trees fall between these two extremes. You can build a completely balanced tree by inserting the median element of the input set, then recursively inserting all lesser elements and greater elements. A simpler approach first sorts the input set. The recursive build function inserts the middle string of its subarray, then recursively builds the left and right subarrays. We use this method in our experiments; it is fast and produces fairly well-balanced trees. The cost of inserting all words in a dictionary with function insert3 is never more than about 10 percent greater than searching for all words. D.D. Sleator and R.E. Tarjan describe theoretical balancing algorithms for ternary search trees in "Self-Adjusting Binary Search Trees" (*Journal of the ACM*, July 1985; http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.95.1380&rep=rep1&type=pdf).

Machine	inert1	insert2	insert3
UltraSPARC-2	3.95	2.75	0.49
MIPS R4000	2.40	1.40	1.00
Pentium Pro	7.91	1.10	0.71

Table 1: Runtimes of insertion functions.

Comparison to Other Data Structures

Symbol tables are typically represented by hash tables. We conducted a simple experiment to compare ternary search trees to that classic data structure; the complete code is available at http://drdobbs.com/database/articleID/184410528/sourcecodeID/30201315.

To represent *n* strings, our hash code uses a chained table of size *tabsize=n*. The hash function, from Section 6.6 of B. Kernighan and D. Ritchie's *The C Programming Language*, Second Edition (Prentice Hall, 1988, http://is.gd/mW08qQ), is reasonably efficient and produces good spread:

```
int hashfunc(char *s)
{   unsigned n = 0;
        for ( ; *s; s++)
                n = 31 * n + *s;
        return n % tabsize;
}
```

The body of the search function is as follows:

```
for (p = tab[hashfunc(s)]; p; p = p->next)
    if (strcmp(s, p->str) == 0)
       return 1;
return 0;
```

For fair timing, we replaced the string comparison function *strcmp* with inline code (so the hash and tree search functions used the same coding style). We used the same dictionary to compare the performance of hashing and ternary trees. We first built the tree by inserting each word in turn

(middle element first, then recurring). Finally, we searched for each element word in the input file. Table 2 presents the times in seconds for the two data structures across three machines. For both building and (successful) searching, the two structures have comparable search times.

Ternary search trees are usually noticeably faster than hashing for unsuccessful searches. Ternary trees can discover mismatches after examining only a few characters, while hashing always processes the entire key. On some data sets with very long keys and mismatches in the first few characters, ternary trees took less than one-fifth the time of hashing.

Functions *insert3* and search combine to yield a time-efficient symbol table. On one dictionary described at our website (http://www.cs.princeton.edu/~rs/strings/), ternary search trees used about three times the space of hashing. An alternative representation of ternary search trees is more space efficient: When a subtree contains a single string, we store a pointer to the string itself (and each node stores three bits telling whether its children point to nodes or strings). This leads to code that is less efficient, but it reduces the number of tree nodes close to the space used by hashing.

We analyzed the worst-case performance of several aspects of ternary search trees in our theoretical paper. In "The Analysis of Hybrid Trie Structures" (*Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, January 1998, see http://portal.acm.org/citation.cfm?id=314613&picked=prox&preflayout=flat), J. Clement, P. Flajolet, and B. Valle analyze the expected performance of a broad class of hybrid trees that includes ternary trees, and describe extensive experiments to support their theory.

Machine	Build		Search	
	Hash	TST	Hash	TST
UltraSPARC-2	0.53	0.49	0.58	0.52
MIPS R4000	0.88	1.00	1.00	0.91
Penitum Pro	0.93	0.71	0.55	0.64

Table 2: Rutimes of data structures.

Other Operations on Ternary Trees

Most standard techniques on binary search trees can be applied immediately to their ternary cousins. For instance, we can print the strings in the tree in sorted order with a recursive traversal:

```
void traverse(Tptr p)
{    if (!p) return;
     traverse(p->lokid);
     if (p->splitchar)
         traverse(p->eqkid);
     else
         printf("%s/n", (char *) p->eqkid);
     traverse(p->hikid);
}
```

Simple recursive searches can find the predecessor or successor of a given element or list all items in a given range. If we add a *count* field to every node, we can quickly count the elements in a given range, count how many words begin with a given substring, or select the *m*th largest element. Most of these operations require logarithmic time in a ternary tree, but linear time in a hash table.

Partial-Match Searching

We turn next to the more subtle problem of "partial-match" or "crossword puzzle" searching: A query string may contain both regular letters and the "don't care" character ".". Searching the dictionary for the pattern ".u.u.u" matches the single word auhuhu, while the pattern ".a.a.a" matches 94 words, including banana, casaba, and pajama. (That pattern does not match abracadabra, though. The entire word must match the pattern; we do not look for patterns in the middle of longer words.)

This venerable problem has been studied in many papers, such as in

"The World's Fastest Scrabble Program," by A.W. Appel and G.J. Jacobson (*Communications of the ACM*, May 1988, http://portal.acm.org/citation.cfm?id=42420). In "Partial-Match Retrieval Algorithms," (*SIAM Journal on Computing,* 5, 1976, http://epubs.siam.org/sicomp/resource/1/smjcat/v5/i1/p19_s1), R.L. Rivest presents an algorithm for partial-match searching in digital tries: Take the single given branch if a letter is specified, for a *don't-care* character, recursively search all branches. Function *pmsearch* implements Rivest's method in ternary search trees. The function puts pointers to matching words in *srcharr[0..srchtop-1]*. It is called, for instance, by:

```
srchtop = 0;
pmsearch(root, ".a.a.a");
```

The following is the complete search code:

```
void pmsearch(Tptr p, char *s)
{    if  (!p) return;
     nodecnt++;
     if (*s == '.' || *s < p->splitchar)
        pmsearch(p->lokid, s);
     if (*s == '.' || *s == p->splitchar)
        if (p->splitchar && *s)
            pmsearch(p->eqkid, s+1);
     if (*s == 0 && p->splitchar == 0)
            srcharr[srchtop++] =
                (char *) p->eqkid;
     if (*s == '.' || *s > p->splitchar)
            pmsearch(p->hikid, s);
}
```

Function *pmsearch* has five *if* statements. The second and fifth *if* statements are symmetric; they recursively search the *lokid* (or *hikid*) when the search character is the don't care "." or when the search string is less (or greater) than the *splitchar*. The third *if* statement recursively searches the *eqkid* if both the *splitchar* and *string* are nonnull. The fourth *if* statement detects a match to the query and adds the pointer to the complete word (stored in *eqkid* by our sleazy hack).

Rivest states that partial-match search in a trie requires "time about $O(n^{(k-s)/k})$ to respond to a query word with *s* letters specified, given a file of *n k*-letter words." Ternary search trees can be viewed as an implementation of his tries (with binary trees implementing multiway branching), so we expected his results to apply immediately to our program. Our experiments, however, led to a surprise: Unspecified positions at the front of the query word are dramatically more costly than unspecified characters at the end of the word. (Rivest suggested shuffling the characters in a word to avoid such problems.) Table 3 gives the flavor of our experiments. The first line says that the search visited 18 nodes to find the single match to the pattern "banana." The tree contains a total of 1,026,033 nodes; searches with some of the first letters specified visit just a tiny fraction of those nodes.

Near-Neighbor Searching

We finally examine the problem of "near-neighbor searching" in a set of

Pattern	Matches	Nodes
banana	1	18
ban...	33	352
.a.a.a	94	5821
...ana	38	24,069
xy.....	7	250
.....xy	16	441,583
television	1	21
tele.....	27	639
t.l.v.s..n	1	254
....vision	6	73,786

Table 3: Partial-match searching.

strings: We are to find all words in the dictionary that are within a given Hamming distance of a query word. For instance, a search for all words within distance two of *Dobbs* finds *Debby*, *hobby*, and 14 other words.

Function nearsearch performs a near-neighbor search in a ternary search tree. Its three arguments are a tree node, string, and distance. The first *if* statement returns *if* the node is null or the distance is negative. The second and fourth *if* statements are symmetric: They search the appropriate child if the distance is positive or if the query character is on the appropriate side of *splitchar*. The third *if* statement either checks for a match or recursively searches the middle child.

```
void nearsearch(Tptr p, char *s, int d)
{   if (!p || d < 0) return;
    if (d > 0 || *s < p->splitchar)
        nearsearch(p->lokid, s, d);
    if (p->splitchar == 0) {
       if ((int) strlen(s) <= d)
          srcharr[srchtop++] = (char *) p->eqkid;
    } else
       nearsearch(p->eqkid, *s ? s+1:s,
          (*s == p->splitchar) ? d:d-1);
    if (d > 0 || *s > p->splitchar)
        nearsearch(p->hikid, s, d);
}
```

Our website (http://www.cs.princeton.edu/~rs/strings/) contains data on the performance of this search algorithm. It is quite efficient when searching for near neighbors, but searching for distant neighbors grows more expensive. A simple probabilistic model accurately predicts its runtime on real data.

Conclusion

The primary challenge in implementing digital search tries is to avoid using excessive memory for trie nodes that are nearly empty. Ternary search trees may be viewed as a trie implementation that gracefully adapts to handle this case, at the cost of slightly more work for full nodes. Ternary search trees combine the best of two worlds: the low space overhead of binary search trees and the character-based time efficiency of digital search tries.

Ternary search trees have been used for several years to represent English dictionaries in a commercial optical character recognition (OCR) system built at Bell Labs. The trees were faster than hashing for the task, and they gracefully handle the 34,000-character set of the Unicode Standard. The designers have also experimented with using partial-match for word lookup: Replace letters with low probability of recognition with the "don't care" character.

Ternary search trees are efficient and easy to implement. They offer substantial advantages over both binary search trees and digital search tries. We feel that they are superior to hashing in many applications for the following reasons:

- Ternary trees do not incur extra overhead for insertion or successful searches.
- Ternary trees are usually substantially faster than hashing for unsuccessful searches.
- Ternary trees gracefully grow and shrink; hash tables need to be rebuilt after large size changes.
- Ternary trees support advanced searches, such as partial-match and near-neighbor search.
- Ternary trees support many other operations, such as traversal to report items in sorted order.

— At the time of publication, Jon was a Member of Technical Staff at Bell Labs, and Bob the William O. Baker Professor of Computer Science at Princeton University.

IN THIS ISSUE

Guest Editorial >>
Two-Way Tables in iOS >>
Java Meets Objective-C >>
Ternary Search Trees >>
Letters >>
Links >>
Table of Contents >>

This Month on DrDobbs.com

Interesting items posted on www.drdobbs.com over the past month that you may have missed

LANGUAGE OF THE MONTH: MIRAH

What happens when you take Ruby syntax, add static typing, and run it on the JVM? Charles Nutter, the designer of JRuby, shows us.
http://drdobbs.com/java/229400307

HOW TO SECURE AND AUTHENTICATE IMAGES USING WATERMARKS

Cocoa applications use the *NSImage* class as the primary container for image data on MacOS X. This class supports most image formats such as JPEG, PNG, and TIFF. It can load data from a local file or read it from a network session. The class is hardware-agnostic — it can adapt its image data to suit the display device's native resolution and color model. But the *NSImage* class is unable to authenticate its image data. This article teaches you how to give *NSImage* the ability to support watermarks, explains how a watermark affects image quality, and shows how well it stands up to alterations.
http://drdobbs.com/security/229400454

CLOUD FOUNDRY: VMWARE'S CLOUD GAMBIT

Andrew Binstock overviews VMware's Cloud Foundry, which consists of several components of cloud infrastructure.
http://drdobbs.com/high-performance-computing/229401463

HOW TO TEACH C++ BADLY

If you want to learn how to confuse beginning C++ programmers, these suggestions are a fine place to start.
http://drdobbs.com/blogs/cpp/229401490

KEEP AGILE GOING

Jim Highsmith, one of the signers of the Agile Manifesto, discusses what Agile needs to do to keep going.
http://drdobbs.com/architecture-and-design/229301126

TWISTING THE RTTI SYSTEM FOR SAFE DYNAMIC CASTS OF VOID* IN C++

This article introduces a safe mechanism to dynamically cast *void** to typed pointers at runtime.
http://drdobbs.com/open-source/229200212

NOTE TO RIM: ALL THINGS TO ALL DEVELOPERS WILL NOT SUCCEED

RIM's support for Java, Android, and C/C++ on the Blackberry Playbook is welcome, but less important to developers than creating a product buyers want.
http://drdobbs.com/mobility/229400918

IN THIS ISSUE

Guest Editorial >>
Two-Way Tables in iOS >>
Java Meets Objective-C >>
Ternary Search Trees >>
Letters >>
Links >>
Table of Contents >>

Dr. Dobb's

Jon Erickson Editor in Chief, Dr. Dobb's
jerickson@drdobbs.com

Andrew Binstock Executive Editor, Dr. Dobb's
alb@drdobbs.com

Deirdre Blake Managing Editor, Dr. Dobb's
dblake@techweb.com

Amy Stephens Copyeditor, Dr. Dobb's
astephens@techweb.com

Sean Coady Webmaster, Dr. Dobb's
scoady@techweb.com

DR DOBB'S
UBM TECHWEB
303 Second Street,
Suite 900, South Tower
San Francisco, CA 94107
1-415-947-6000

IN THIS ISSUE

Guest Editorial >>
Two-Way Tables in iOS >>
Java Meets Objective-C >>
Ternary Search Trees >>
Letters >>
Links >>
Table of Contents >>

Dr.Dobb's Business Contacts

DR. DOBB'S

Sales Director, Michele Hurabiell
(415) 378-3540, mhurabiell@techweb.com

Account Executive, Shaina Guttman
(212) 600-3106, sguttman@techweb.com

INFORMATIONWEEK BUSINESS TECHNOLOGY NETWORK

CMO, Scott Vaughan
(949) 223-3662, svaughan@techweb.com

VP of Group Sales, InformationWeek Business Technology Network, Martha Schwartz
(212) 600-3015, mschwartz@techweb.com

Sales Assistant, Group Sales, Kelly Glass
(212) 600-3327, kglass@techweb.com

Publisher's Assistant, Esther Rodriguez
(949) 223-3656, erodriguez@techweb.com

SALES CONTACTS—WEST

Western U.S. (Pacific and Mountain states) and Western Canada (British Columbia, Alberta)

Western Regional Director, Matt Stovall
(415) 947-6245, mstovall@techweb.com

District Sales Manager, Rachel Calderon
(516) 562-5338, rcalderon@techweb.com

Inside Sales Manager, Vesna Beso
(415) 947-6104, vbeso@techweb.com

Sales Assistant, Ian Doyle
(415) 947-6105, idoyle@techweb.com

Strategic Accounts

Account Director, Sandra Kupiec
(415) 947-6922, skupiec@techweb.com

Account Manager, Shoshana Freisinger
(415) 947-6349, sfreisinger@techweb.com

Sales Assistant, Matthew Cohen-Meyer
(415) 947-6214, mmeyer@techweb.com

SALES CONTACTS—EAST

Midwest, South, Northeast U.S. and Eastern Canada (Saskatchewan, Ontario, Quebec, New Brunswick)

District Manager, Jenny Hanna
(516) 562-5116, jhanna@techweb.com

District Manager, Michael Greenhut
(516) 562-5044, mgreenhut@techweb.com

Account Manager, Cori Gordon
(516) 562-5181, cgordon@techweb.com

Inside Sales Manager East, Ray Capitelli
(212) 600-3045, rcapitelli@techweb.com

Sales Assistant, Elyse Cowen
(212) 600-3051, ecowen@techweb.com

Strategic Accounts

District Manager, Mary Hyland
(516) 562-5120, mhyland@techweb.com

Account Manager, Tara Bradeen
(212) 600-3387, tbradeen@techweb.com

Account Manager, Jennifer Gambino
(516) 562-5651, jgambino@techweb.com

Sales Assistant, Kathleen Jurina
(212) 600-3170, kjurina@techweb.com

Dr. Dobb's

Sales Director, Michele Hurabiell
(415) 378-3540, mhurabiell@techweb.com

Account Executive, Shaina Guttman
(212) 600 3106, sguttman@techweb.com

MARKETING

VP, Marketing, Winnie Ng-Schuchman
(631) 406-6507, wng@techweb.com

Director of Marketing, Sherbrooke Balser
(949) 223-3605, sbalser@techweb.com

Marketing Manager, Monique Luttrell
(949) 223-3609, mluttrell@techweb.com

AUDIENCE DEVELOPMENT

Director, Karen McAleer
(516) 562-7833, kmcaleer@techweb.com

BUSINESS OFFICE

General Manager, Marian Dujmovits

United Business Media LLC
600 Community Drive
Manhasset, N.Y. 11030 (516) 562-5000

UBM TECHWEB

Tony L. Uphoff CEO

John Dennehy CFO

David Michael CIO

Bob Evans Sr. VP and Global CIO Director

Joseph Braue Sr. VP, Light Reading Communications Network

Scott Vaughan CMO

Ed Grossman Executive Vice President, InformationWeek Business Technology Network

John Ecke VP and Group Publisher, Financial Technology Network, InformationWeek Government, InformationWeek Healthcare

Martha Schwartz VP, Group Sales, InformationWeek Business Technology Network

Beth Rivera Senior VP, Human Resources

David Berlind Chief Content Officer, TechWeb, and Editor in Chief, TechWeb.com

Fritz Nelson VP, Editorial Director, InformationWeek Business Technology Network, and Executive Producer, TechWeb TV

UNITED BUSINESS MEDIA LLC

Pat Nohilly Sr. VP, Strategic Development and Business Admin.

Marie Myers Sr. VP, Manufacturing