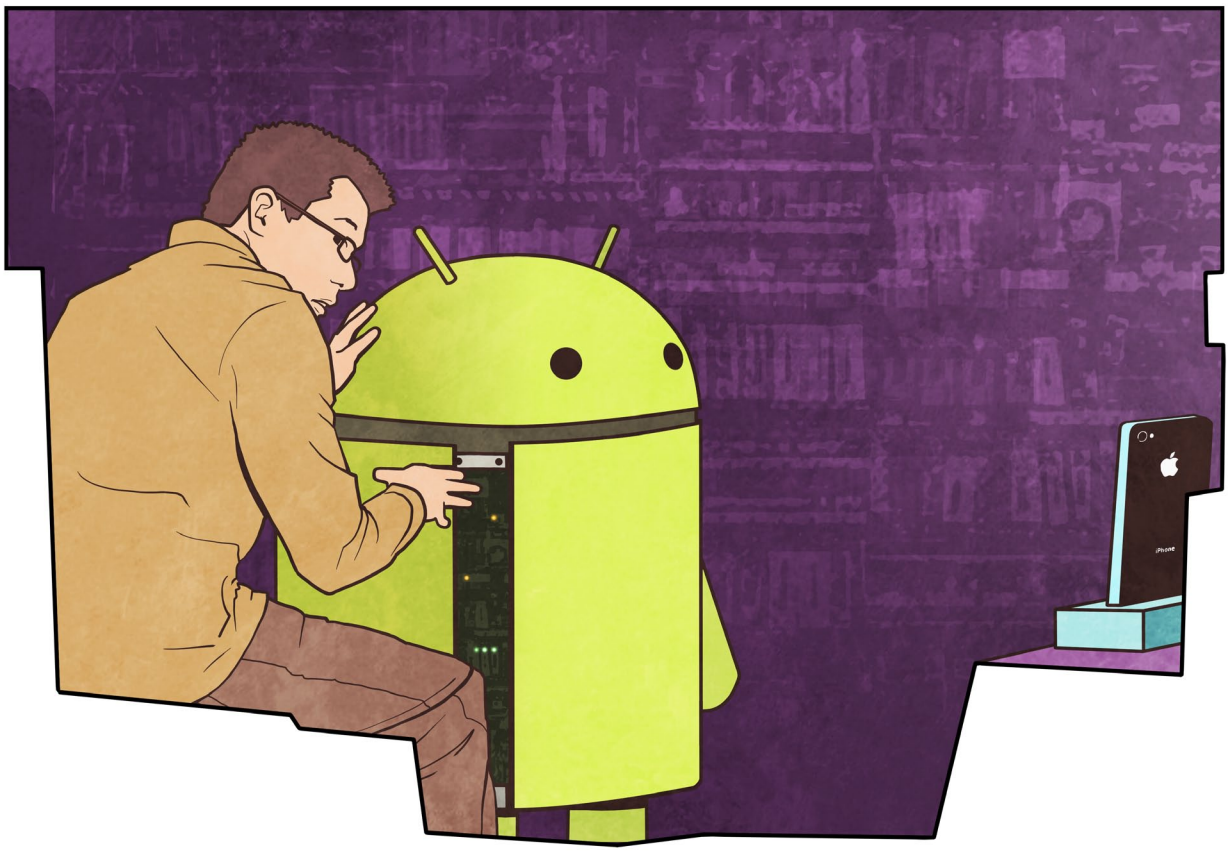


An iOS Developer Takes on Android

by Nick Farina

HACKERMONTHLY

Issue 19
Dec 2011

Which design had 60% more signups?

TRUSTED BY OVER
13,000
USERS

Microsoft · AMD · GROUPON · foursquare · LOVEFILM.com · AWeber COMMUNICATIONS

Answer:
http://bit.ly/a-or-b

Visual Website Optimizer
World's easiest A/B testing tool

HARVEST

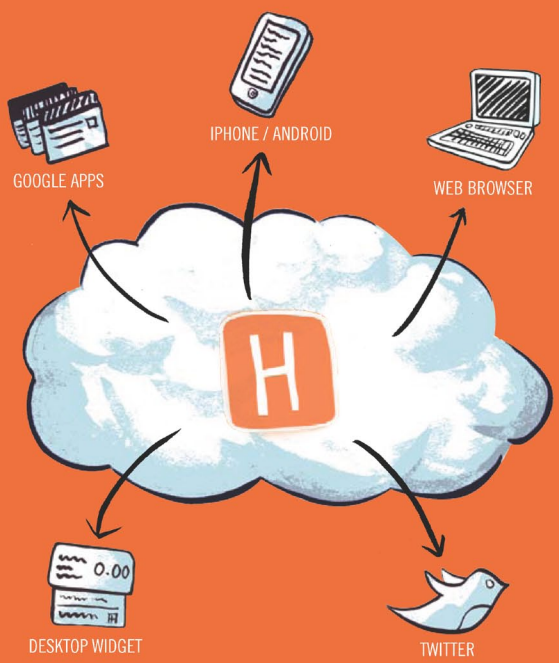
TIMESHEETS INVOICES REPORTS

Track time anywhere, and invoice your clients with ease.

Harvest is available wherever your work takes you. Whether you are working from home, on-site, or through a flight. Harvest keeps a handle on your billable time so you can invoice accurately. Visit us and learn more about how Harvest can help you work better today.

Why Harvest?

- Convenient and accessible, anywhere you go.
- Get paid twice as fast when you send web invoices.
- Trusted by small businesses in over 100 countries.
- Ability to tailor to your needs with full API.
- Fast and friendly customer support.



Learn more at **www.getHarvest.com/hackers**

Curator
Lim Cheng Soon

Contributors
Nick Farina
Reginald Braithwaite
Clint Watson
Gabriel Weinberg
Jared Carroll
Rich Jones
Russ Cox
Kristian Storm

Illustrator
Jaime G. Wong

Proofreader
Emily Griffin

Printer
MagCloud

HACKER MONTHLY is the print magazine version of Hacker News — *news.ycombinator.com*, a social news website wildly popular among programmers and startup founders. The submission guidelines state that content can be "anything that gratifies one's intellectual curiosity." Every month, we select from the top voted articles on Hacker News and print them in magazine format. For more, visit *hackermonthly.com*.

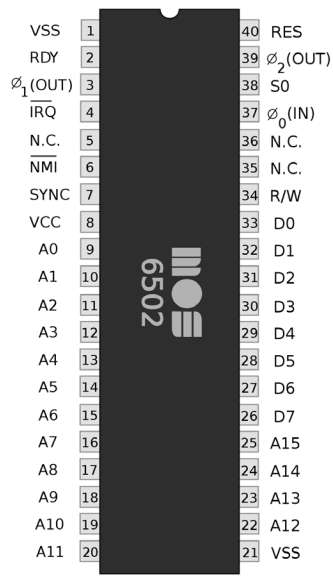
Advertising
ads@hackermonthly.com

Contact
contact@hackermonthly.com

Published by
Netizens Media
46, Taylor Road,
11600 Penang,
Malaysia.

Cover Illustration: Jaime G. Wong

Contents



MOS Technology 6502 CPU pinout diagram, by Bill Bertram

An iOS Developer Takes on Android

By NICK FARINA

Recently, we released the Android version of Meridian, our platform for building location-based apps.

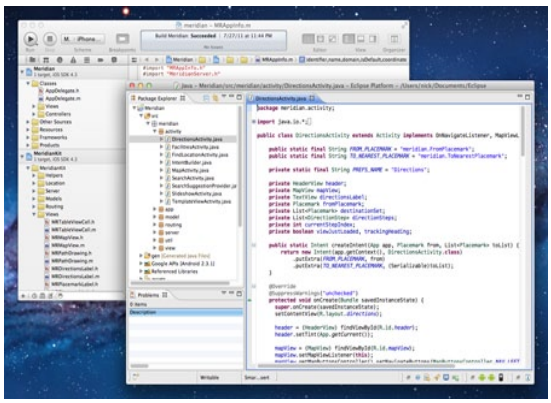
We didn't use one of these "Cross Platform!" tools like Titanium. We wrote it, from scratch, in Java, like you do in Android.

We decided it was important to keep the native stuff native, and to respect each platform's conventions as much as possible. Some conventions are easy to follow, like putting our tabs on the top. Other conventions go deep into the Android Way, like handling `Intents`, closing old `Activities`, implementing Search Providers, and being strict about references to help the garbage collector.

Now, our platform leverages HTML5 (buzzword, sorry) in many places for branding and content display, so we got a fair amount of UI for free. But there was much platform code written in Objective-C that needed translation into Java, such as map navigation, directions, and location switching.

So, we rolled up our sleeves, downloaded the Android SDK, and got to work.

Development Environment



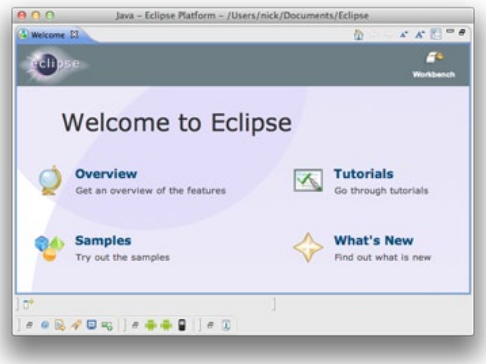
Apple has made it pretty easy to start writing iOS apps. Of course, Step One is "Buy a Mac." Easy! Then just

download the free Xcode Installer from the Mac App Store, and start writing code when it's done.

Android is a bit more involved. You can download the SDK easily, but to actually start writing code, you'll want to setup Eclipse and install Google's ADT Plugin.

If you want to waste a week or so playing around and not getting work done, you could explore the many tempting alternatives to writing Java in Eclipse. You could download Netbeans, or write in Scala, or finally start learning VIM.

But let's assume you are on a deadline and want to do things the way that Google endorses. The first thing you'll need to do is accept the reality of Eclipse.

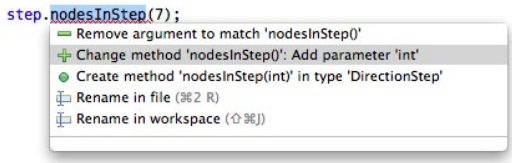


You're going to just hate Eclipse. You're going to hate it with the heat of a thousand suns. It's going to feel slow and bloated and it won't taste like real food.

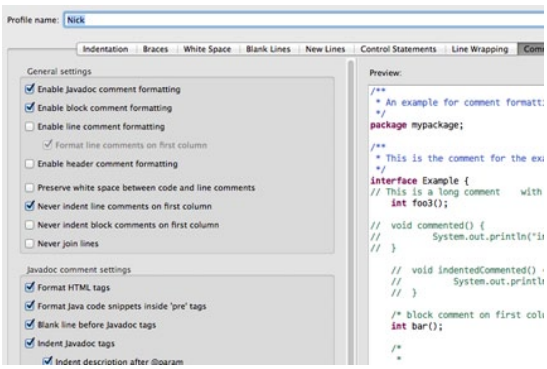
Eclipse is a world unto itself. It's the IDE to end IDEs. Consequently, it has many abstract-sounding concepts you'll have to learn. There are *Workspaces*, and *Perspectives*, and *Run Configurations*. And Eclipse itself is just an empty shell of sorts; all non-trivial functionality is provided via a complex network of interdependent Plugins, similar to Linux distributions. Come to think of it…

Of course, IDE weirdness isn't unique to Eclipse; Xcode was pretty damn weird at first, too, and it's getting more meta with each release (Schemes, anyone?).

The upside is, after acclimating to Eclipse, you'll enjoy some seriously amazing, productivity-boosting code completion, refactoring, and automatic fixing. It'll basically write your code for you.



A great way to get comfortable with Eclipse is to spend a couple hours, and I'm being dead serious, tweaking the hundreds of options and checkboxes and fiddly things in the Preferences section.



Again, being serious here, I felt a lot more comfortable and familiar and happy with Eclipse after getting to know it this way. Does this mean that other programs should expose every conceivable preference imaginable? Jesus, no. Are you crazy?

The Java Language

Java is a high level programming language. It's unproductive to have an opinion about it. Instead, consider how Android uses Java.

This [hn.my/async] is how you do something on a background thread. This [hn.my/uievents] is how you listen for events (it's actually just like a `delegate` in ObjC). This [hn.my/activities] explains the lifecycle of `Activities`, which are exactly like `UIViewControllers` in Cocoa.

Overall, the Android frameworks are very well designed and consistent, and the API works harmoniously with the Java language. It's actually similar enough in the fundamentals that our app has almost the exact same class structure on Android as on iOS.

And the code ended up looking strikingly similar as well. Here's a snippet of ObjC from our app that draws an arrowhead, followed by the Java version.

```
void CGContextDrawArrowhead(CGContextRef c, NSArray *nodes) {

    if ([nodes count] < 2) return;

    CGContextSetLineCap(c, kCGLineCapRound);
    CGContextSetLineJoin(c, kCGLineJoinMiter);

    MRRoutingNode *lastNode = nodes.lastObject;
    MRRoutingNode *penultimateNode = [nodes objectAtIndex:nodes.count-2];

    CGPoint startPoint = penultimateNode.point;
    CGPoint endPoint = lastNode.point;

    float length = 9.0f;
    float width = 5.0f;

    double r = atan2(endPoint.y - startPoint.y,
                     endPoint.x - startPoint.x);
```

```
static void drawArrowhead(Canvas canvas, List<RoutingNode> nodes) {

        if (nodes.size() < 2) return;

        paint.setStrokeCap(Cap.BUTT);
        paint.setStrokeJoin(Join.MITER);

        RoutingNode lastNode = Coll.last(nodes);
        RoutingNode penultimateNode = nodes.get(nodes.size()-2);

        PointF startPoint = penultimateNode.point();
        PointF endPoint = lastNode.point();

        float length = 9.0f;
        float width = 5.0f;

        double r = Math.atan2(endPoint.y - startPoint.y,
                              endPoint.x - startPoint.x);
```

Freaky, right? A lot of our source is like that.

Debugging

Now that you've written some code, you'll want to try running it. In Apple's world, we have the iOS "Simulator."

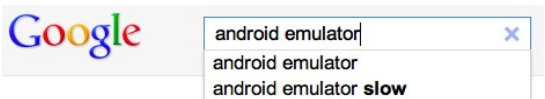
It's called a Simulator because it's phony (ha!). It's not the real iPhone OS. When you run your app "in the simulator," Xcode compiles your app into a desktop application, and runs it natively on your Mac. If you look in Activity Monitor, you'll see your app right there, running alongside Mail and iCal and iTunes.

The downside to the Simulator is that it doesn't exactly replicate the iOS environment. And your code isn't the same either; it must be compiled for x86/64 instead of ARM, so there's no guarantee things will work the same on an iOS device.

The upside to the Simulator is that it's not an "Emulator." What's an Emulator?

Behold, the Android Emulator. Its goal is noble: to run your app on the complete unaltered Android OS inside a virtual machine that executes ARM instructions.

What is the primary quality of the Android Emulator?



I thought the iPhone Simulator was a terrible idea when I first heard of it, until I remembered that we built the exact same thing for Windows Mobile back in the day, because the Windows Mobile Emulator was so goddamn slow.

It takes the Android Emulator about 2 minutes to boot up on my perfectly modern machine. But what really hurts is the edit/debug cycle. Every time I change a bit of Java and need to rerun the app, it takes about 30 seconds to redeploy and start up in the Emulator. Compare that to 5 seconds on the iOS Simulator. It may not sound like much but remember you'll be doing this hundreds of times throughout your day.

Fortunately, it turns out to be much quicker to deploy and boot up your app on a physical device over USB. So trust me, just go out and buy an Android phone. Better: buy a few of them to test against fragmentation.

Not that fragmentation is unique to Android; it's just exaggerated a lot in the media. We need quite a few iOS devices in our lab, too. One tiny unexpected OS or device difference can bring your app crashing down on any platform.

UI Design Tools

For laying out widgets on iOS, we have Interface Builder.

Your experience with Interface Builder may differ from mine, but I've learned to use it very sparingly. It's great when you want to create static layouts with precision. But for dynamic content, especially `UITableViews`, it tends to make things more complex than they would be in code. Especially when you come back after a while and forget all the little dependencies between your `Controller` and your `XIB`.

On Android, you can create UI layouts in XML.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout
  xmlns:android="http://schemas.android.com/apk/res/android"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  android:background="#a4bbcd" android:orientation="vertical">

    <!-- Flexible space -->
    <View android:layout_width="wrap_content"
          android:layout_height="wrap_content"
          android:background="@drawable/blue_gradient"
          android:layout_weight="1"/>
```

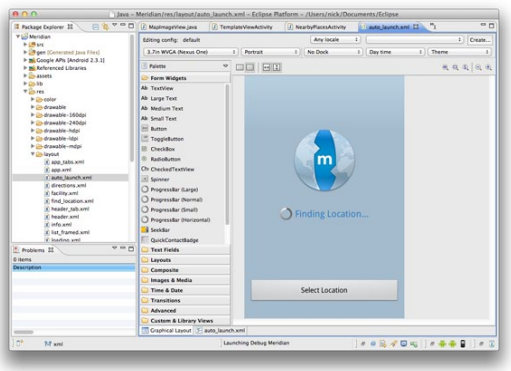
It's a bit like HTML, except it's not HTML. It has a basic styling system that's kind of like CSS, except it's not CSS.

Lots of well-meaning developers in positions of power have tried to reinvent HTML and CSS over the years. Mozilla created XUL for cross-platform UI in Firefox. Adobe created MXML for Flash. Microsoft created XAML for Windows.

I myself am guilty of creating my own XML-based layout system for Flash called Bent, back when I had too much free time. So I can tell you that inventing these systems is the most fun you will ever have as a developer. It feels like you are creating the One True Framework, and once it's complete everyone will embrace it and get excited about it and learn it and build on it and hold you up and sing epic poems about your genius!

But the reality is, it's not HTML and CSS and so it's another thick layer of stuff that you have to learn and understand and fight with when things don't work like you expect.

On the plus side, you can preview your XML at design-time in a nice visual editor, much like Interface Builder:

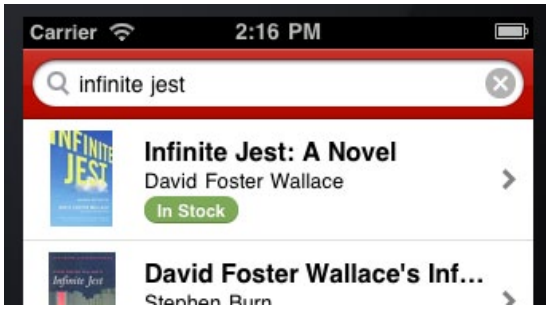


Which is pretty grand.

Now, technically you could write everything in Java, just like on iOS you can eschew Interface Builder entirely and write only Objective-C. But you'll find that when you scour the Internet for how to do a thing on Android, you'll end up needing to understand all these XML formats anyway just to understand code samples on the web.

But it's not really that bad, because you also get…

A Real Box Model



Here's a list item on iOS that represents a search result.

And here's just a snippet of the ObjC that renders that item:

```
CGFloat x = MARGIN + SEARCH_RESULT_THUMB_SIZE + MARGIN;
CGFloat y = MARGIN + 5 + 5;
CGFloat w = self.frame.size.width - MARGIN - SEARCH_RESULT_THUMB_SIZE - MARGIN - 30;
// the 30 is to account for disclosure

if (hasDecorator) y -= DECORATOR_HEIGHT/2;

// the title is allowed to take up two lines, then the whole thing is vertically centered
CGSize titleSize = [title sizeWithFont:titleFont
                     constrainedToSize:CGSizeMake(w,titleFont.leading * maxTitleLines)
                         lineBreakMode:UILineBreakModeTailTruncation];

if (titleSize.height > titleFont.leading)
    y -= 10; // two lines? move things up a bit

if (!subtitle) y += 7;
```

Talk about a bag of hurt. You could create the initial layout in Interface Builder, of course, but then you'd have to kiss that silky smooth scrolling goodbye. Maybe in the future when iOS devices are blazing fast.

As humans, we don't typically think "The title should be positioned at 30 pixels by 40 pixels, with a maximum height of 35 pixels." Instead, we think "The title should be above the subtitle, and to the right of the thumbnail image, and have at most 2 lines."

Android has a system of layout containers (similar to HTML) that let you describe where content should be, relative to everything else. Here's a similar snippet from the same search result in Android:

```
<LinearLayout
    android:layout_width="fill_parent"
    android:layout_height="wrap_content"
    android:layout_gravity="center"
    android:orientation="vertical">

    <TextView android:id="@+id/title"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:layout_marginBottom="-2sp"
        android:text="Infinite Jest"
        android:textSize="18sp"
        android:textColor="#ffffff"
        android:lines="2"/>
```

Now, you'll have to study up on all these layout containers like `FrameLayout`, `LinearLayout`, `RelativeLayout`, and learn all their quirks, but in the end you'll get a very natural, flexible UI layout system.

The best part is that it's zero-effort-easy to make layouts that automatically resize for portrait/landscape device orientations and varying screen sizes. This is in stark contrast to the absolutely primitive springs and struts system in Cocoa/iOS.

But here's the thing about the simplistic layout system in iOS that I just mocked, it turns out to be a reasonable compromise when you consider…

Animation

The thing to realize about Android is that it used to look like this:

It was conceived and designed during the pre-iPhone days of Blackberry and Windows Mobile, and the influence of those platforms goes very deep into the Android OS.

For instance. The rendering system, that is, the method by which UI widgets like menus and buttons and such get painted on the screen, is primarily software-based.

What does that mean? Let's take the screenshot above as an example. If you pressed the Down key, you would expect the "Homepage" entry to be selected instead of "Go to." So you press the Down key. This causes an "invalidate," meaning, "please repaint the screen." So the screen is cleared, then:

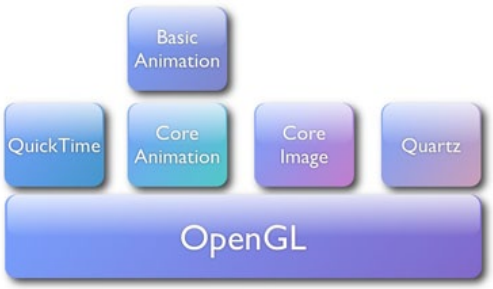
1. The OS redraws the status bar at the top
2. The WebView redraws the Google.com website
3. The Menu draws its translucent black background and border
4. All the menu text is drawn
5. The blue gradient highlight is drawn over "Homepage."

This all happens very quickly, and you only ever see the final result, so it looks like just a few pixels have changed, but in fact the whole screen must be reconsidered and redrawn.

If this sounds familiar, it's because this is the basic method used in GDI, the rendering system introduced with Microsoft Windows 1.0. That sounds damning, but really most GUIs operated this way.

Until the iPhone came along…

When you're using an iPhone, you're



playing a hardware-accelerated 3D game. You know, the kind of 3D where everything is made out of hundreds of little triangles.

When you flick through your list of friends in the Contacts app, you're causing those triangles to move around. And there's a "camera," just like a 3D shooter, but the camera is fixed above the Contacts app's virtual surface and so it appears 2D.

Which is a long way of saying that everything on iOS is drawn using OpenGL. This is why animation on iOS is so hopelessly fast. You may have noticed that `-drawRect` is not called for each frame of an animation. It's called once, then you draw your lines and circles and text onto an OpenGL surface (which you didn't even realize), then Core Animation moves these surfaces around like pulling on the strings of a marionette. All the final compositing for each frame is done in hardware by the GPU.

Android seems to have made the decision early on that they wouldn't force their manufacturing partners to include a GPU. This decision made total sense back in the pre-iPhone days, but now it's causing pain, as even the new hardware acceleration in Android 3.0 is limited by the original software-based compositing system.

So here's the catch with the wonderful flexible layout system in Android: You must be very careful. If you animate certain kinds of properties, you can easily force the CPU to do all that fancy, expensive layout on each animation frame. And the CPU is very busy right now parsing some JSON from a web API or something, OK?

Meridian For Android

All in all, it took us about 4 months of development time to build and release Meridian for Android.

When we first released Meridian, our number one piece of feedback was "Will you make an Android version please?" Except often without the "please."

And it turns out Android is the third platform for Meridian. The first was Windows Mobile, if you can believe that.

We started building what we now call "Meridian," back before the iPhone existed. At the time, Windows Mobile was the only mobile platform capable of delivering the experience we wanted.

So you could say I've got a rather long view of things now. There will always be new platforms and new paradigms to learn. The best we can do is to understand where each one came from, and to embrace the positives and overcome the negatives as quickly as possible so we can ship some awesome features before everything changes again. ■

Nick is the CTO and cofounder of Meridian [meridianapps.com], the location-based software platform. He also cofounded Spotlight Mobile, creator of award-winning apps for brands like Nike, Vogue, and Barnes & Noble.

Reprinted with permission of the original author. First appeared in *hn.my/iosandroid* (nfarina.com)

I Make Dreams Come True

By REGINALD BRAITHWAITE

Photo: *flickr.com/photos/jsorbie/2953147267*

> **"Visions are worth fighting for. Why spend your life making someone else's dreams?"**
>
> — Orson Welles in the film "Ed Wood"

Recently, I read "Don't Call Yourself a Programmer." [hn.my/callprogm] Right away, I'm going to say that in my nearly thirty years of experience turning software into money, just about everything Patrick says about the business of software and how to get ahead is absolutely true. Sure, you can quibble with something or other. After all, getting ahead in business is a huge subject and this is an essay. Something has to be over-generalized or insufficiently explained. That's the nature of trying to cram so many bits of information into a fixed container.

"Don't Call Yourself a Programmer" contains a lot of the stuff I would want to tell a young Mr. Braithwaite. There are some other things I'd want to tell myself. One of them is, I'm not in the "programming" business, or the "adding value to the economy" business. I program and add value, but my motivation

> **“Now that I’ve dreamt in color, I don’t want to wake up and live in black and white.”**

for programming and for finding ways to add value is to finance my real business: **the “making dreams come true” business.**

Consider movies. Sure, there are incredible documentaries. And moving, gritty, realistic dramas. But many great movies have a dream-like quality to them. I don’t mean the movies depict dreaming. But movies like Amelie [hn.my/amelie] have a dreamy feeling, something that connects with us in a deep way, something that takes us out of where we are and places us in the past, future, or parallel reality. Something that breaks the laws of physics or society.

Such movies aren’t just about what the filmmakers were dreaming; they connect with us because they’re a realization of our dreams. Filmmakers make dreams come true. Just like I do. And I don’t just mean that I make my dreams come true. I write software that makes its users’ dreams come true.

The Dream Feeling

Software (like films) realizes dreams with a visual look and feel, a new programming notation, or perhaps a library that makes you think about doing an old thing in a new way. Great software and great programming languages have a dream-like quality. Learning to use them has this strange feeling of needing to relearn everything you know while feeling natural.

I want that from the software I build. If it doesn’t change the way users think, what damn good is it? When someone uses my software or programs with one of my libraries, I want “Aha!” I want, “Of course it works this way, why did everyone screw this up before?” I want, “Now that you’ve given it to me, you can have it back when you pry it from my cold, dead fingers.”

As a user, I had this experience looking at a Macintosh in 1985. The GUI seems obvious today, but to someone who hadn't seen The Mother of all Demos [hn.my/demo], or hadn't followed what PARC was up to, the illusion of direct manipulation and WYSIWYG was like being in a dream. It had felt so different from everything I'd seen before, but somehow connected deeply with how things ought to be. It wasn't alien.

Speaking of drawing things, when I started to learn object-oriented programming, I bought a book that walked me through writing a vector drawing program step-by-step using Object Pascal. I won't say that Objects were an "Aha!" for me. I grokked the style and my response leaned towards, "Oh, fine, that's how we'll do it." Polymorphism was more of a "Neat-o" than an "I must be dreaming." To this day, I'm reserved about OOP. But something else blew my mind: the book explained how to implement an Undo Stack using what is now called the Command Pattern.

When I figured out what was going on, my mind immediately told me that I wasn't in Kansas any more. Here I was thinking about programs as a collection of data types and some procedures and functions operating on those data types. OOP reorganized the data types and functions, but it wasn't a huge leap of thought. You still operated with data types and functions. But Commands were something new: Now the idea of changing the data was a piece of data itself. To this day, the Command Pattern feels very related to the essential idea of Lisp's equivalence between programs and data, to the idea of a programmable programming language, to the idea of functions operating on functions and not just on data. It felt like the basic laws of physics were disrupted, like someone had shown Newton the two slit experiment or general relativity.

Did I just talk about the Command Pattern? I went to the first Startup School. Naturally, I could tell you about seeing Woz in person or some such, but do you know what blew my mind? Seeing a crowd of people use SubEthaEdit to take notes. Collaborative editing was another dream-like feature, something which made me feel like I was in an alternate reality where software and operating systems worked on an entirely different basis. Collaborative editing takes the Command Pattern and runs it through an algorithm called Operational Transformation and out comes magic: people around the world editing a document like a spreadsheet or a program in real time.

Now that I have seen Operational Transformation, I can't help thinking that all of the CRUD software I'm writing is Doing It Wrong, that nearly everything ought to be live and in real time and collaborative. That every web program should be manipulating commands or edits or whatever you want to call them, that undo and collaboration and a list of who did what should all be basic and baked into the architecture. Once again, everything is different, yet somehow feels like it's always been this way.

And now that I've dreamt in color, I don't want to wake up and live in black and white.

The Business of Dreams

I strongly suspect that most people will either agree or disagree with my feelings at a visceral level. It's a little like invention: you probably have some deep-seated attraction to the idea of inventing new things. You may also have a perfectly healthy aversion to going down the rabbit hole when you could be getting things done with the tools and processes that already exist.

So I'm not going to tell you that you are going to be taller, more attractive, or make more money if you try to make dreams come true. I can tell you that there is money to be made in the dreams business. You know how I don't want to wake up from my dreams? Other people are just the same. Do you want to give up your GUI and do everything on the command line? Or do you think phones should be tethered to a twisted pair of copper wires?

Maybe a few people with Emacs think so. Congratulations, retro-grouches. Everyone else in the world is addicted to GUIs and watching a movie on a tablet and collaboratively editing a spreadsheet. Once you show someone a dream and let them live in it for a moment, they don't want to leave. Dreams are habit-forming. Dreaming is addictive. And if you make dreams, your business supplies dreams to people who never want to give them up.

That's a powerful business model. Use it wisely. ■

Reginald is a software developer and development lead with Unspace Interactive. He writes code and words about code in homoiconic. Follow him on Twitter @raganwald

Reprinted with permission of the original author.
First appeared in *hn.my/dreams*

How My Lifestyle Business Became a Startup

By CLINT WATSON

I CHUCKLED WHEN I read this in a recent article by Gabriel Weinberg of DuckDuckGo fame [hn.my/ddgtimes], "Things take time. We've been live now for about 3 years. They don't need to take that long of course, but don't expect results overnight."

Things take time indeed. Three years? I've been at this for over 10 years now. Are we even a "startup?" At this point, I've got to own up to the fact that FASO is an ongoing concern that happens to be entering a startup-like phase later in life. Maybe we're not truly a startup, in the sense of being new and not-yet-profitable (we're old and profitable), but the challenges and growth we face today have changed us from a lifestyle business into a startup.

The missive below outlines how FASO went from a sideline project, almost a hobby, to the full-blown "startup" we are today. Our path was very different from what you normally read about in the current angel/VC-fueled environment. Perhaps our story will give hope to other weekend warriors who don't have the luxury of quitting their job, moving to the valley, and working on their startup full-time.

Starting sometime in mid-2000, our growth looked like this:

Year	Number of Paying Users
2000	ideas, planning
2001	early coding, first few customers
2002	35
2003	100
2004	170
	[started full time]
2005	350

> **“The idea was wasn’t to build a start-up and exit. The idea was to make a great living doing something I loved.”**

Year	Number of Paying Users
2006	683
2007	1,130
2008	1,978
2009	2,656
2010	3,561
2011	4,382
Today	5,637 active users 4,608 paying users + 1,029 active free users

*All figures are mid-year numbers to coincide with number of full years since launch.

A Short History of My Long Startup

I had an early background in computers, but had taken a 16-year detour to follow a career in one of my other passions: the visual arts. During those 16 years, I ran and later owned an art gallery. I was able to use some of my early hacker skills while in the gallery business and had cobbled together a nice, dynamic, functional website to showcase art for the gallery. This was back when such things were still a novelty, at least for smaller concerns. However, by the time the idea for FASO started to percolate in my mind, my coding skills were, to understate the issue, rusty. Nevertheless, I laid down the first code that eventually became our web application in February of 2001, at 32 years old — already ancient for a tech startup founder. Joel said that good software takes 10 years, but Joel’s a great programmer. I’m not nearly as good as him. Plus, FASO was part-time for the first 5 years, so I’m going to allow us a few more years on achieving “good” status with our software.

I started FASO, not because I was seriously planning a startup but because I wanted to solve a friend’s problem. My friend, an artist, wanted a nice website to display his art, and so he asked me for help. “Make something people want” is what Paul Graham says.

And I inadvertently stumbled onto that. My friend had a problem that he wanted solved, so I decided to build the solution that had been percolating in my mind.

Turns out, a few other artists wanted the same solution. After a few months, I had given my solution away to a few more artists. In fact, I had to give it away: I had no idea how to easily collect money online anyway. One day, I ended up on the phone with a guy who, of all things, ran a credit card gateway and, as luck would have it, he had an ASP library. So I set up an account with him and was able to accept credit cards! Honestly, this was all a bit simpler in those days because it predated PCI. (Not that we weren't careful anyway — we were. Today, we process payments via Braintree Payment Solutions' excellent API, which handles all the difficult compliance stuff for you.)

Probably six months after setting up my friend's site, I landed my first cash-paying customer (she's still a customer), even though I still had no signup process. I didn't even have a front-end website (other than a page where the users could log in). I would simply meet an artist, in real life, sell them on the idea and then go manually add their record to the database, after which point they could log in.

For the first 5 years, while I ran this thing as a part-time "nights and weekends" project, it was pretty stress-free; I just hacked on it when I had time. Once I went full-time, I spent the next couple of years learning how to do things properly, refactoring code, migrating onto more serious platforms, etc. I thought I could just build a nice little lifestyle business. The idea was wasn't to build a startup and exit. The idea was to make a great living doing something I loved.

However, once we reached a certain point, I looked up and realized that we were now a startup.

Becoming a "Real" Startup

As your customer load grows, as you face scaling decisions, as you attract competitors, all of the "normal" things you read about startup life start to happen. Serious competitors, especially, change the game. Like Paul Graham said, "your competitors decide how hard you work. And they pretty much all make the same decision: as hard as you possibly can." So much for my lifestyle business. That's okay; I still love what I do. Startups are fun.

So now, after 10 years, I find myself running a startup and working harder than I ever did at the beginning. But I always did tend to do things a little backwards.

How to Follow the Slow, Safe Road

I suspect many people want to start a startup, but already have obligations: a mortgage, a family, etc. Risking it all is fine when you're in college, but, if you can't risk it all and move to the valley, then perhaps there is another path.

I think the trick to starting slow is to pick the right industry or niche. You can't compete slowly with Facebook, Twitter, Google, or even Y-Combinator startups. They're just going to move way too fast. However, I still think our long road to success can be followed by others, as long as you solve a real problem (especially if it's currently unsolved or poorly solved) for real users who are willing to pay. Look in niches outside of tech for those (please don't build another URL shortener, Twitter client, or, for Pete's sake, a photo-sharing app).

It helps tremendously to have experience in your niche's industry. I see would-be founders on HN saying that it seems like most of the good ideas have been done. That's because they're not looking outside of the narrow world of tech. Really, how many bug trackers does the world need? Look somewhere else.

Go get a job in another industry that interests you. You'll be amazed at how many problems are unsolved. Watch how we solve problems in tech, and then ask how you can mimic those solutions specifically for your niche.

Don't worry about losing too much time, starting a few years later isn't the end of the world and may just give you a huge competitive advantage.

For example, at 32, I had run or owned an art gallery for 12 years. That's a huge competitive advantage in terms of knowledge, experience, and industry contacts that can't easily be duplicated by our competitors. So, yeah, while it took us several years to stand on our own two legs — our extended search for our MVP — we now have several advantages. Including the fact that VC's don't own 80% of the company. They own 0%. I own 100% (I've had one offer to purchase it. The offer was a lot of money...but not enough).

"Things take time," Gabriel said. He wasn't kidding. We're the poster boys for that motto. Fortunately for Gabriel, he's much further along at 3 years than we are at 10. Kudos to him.

With any luck, you'll get there in a lot less time than a decade. But even if it takes that long, don't worry, it's a fun ride. ■

Clint is the single founder and lead developer of FineArtStudioOnline [faso.com], a web company (founded in 2001) that creates web applications for visual artists. His company's (and his personal) core belief is "Sharing Art Enriches Life." Before FASO, Clint ran (and was part-owner) of an art gallery for about 16 years.

Reprinted with permission of the original author.
First appeared in *hn.my/lifestyle* (faso.com)

What I Learned From Raising Venture Capital

By GABRIEL WEINBERG

I'm new at this

I recently raised venture capital for the first time. I'm going to relate what happened below and highlight my mistakes/takeaways in the bold section titles. However, please bear in mind my lack of experience in these matters and that this is written from the entrepreneur's perspective.

Background

DuckDuckGo raised a series A round from Union Square Ventures and a handful of awesome angel investors. While we did talk to over 30 VC firms, I realized our path ended successfully and had been relatively quick and painless. So first off, I want to say I sympathize with everyone who struggles with the funding process.

The funding process is not to be taken lightly

Even though our round went relatively smoothly, it was still a massive time sink. It was the top idea in my mind, and it pretty much consumed my life for four months. In other words, it seems like you should commit to being all-in, all-consumed for a while, or you might as well not do it.

This reality is especially troublesome for two reasons. First, you're distracted, and so your business suffers. Second, every VC expects you to keep making forward progress (as if you're just talking to just them and have the rest of the time to work on the business — yeah right).

Save up good news for the middle of the process

We did not do this, but it ended up working out that way anyway, and so I saw the value in it first hand. I started raising right after releasing dontbubble.us, which is demarcated by C in our traffic graph. Then in the middle of the process, we got picked as one of TIME's top 50 websites for 2011 (annotated as D in the graph). It was nice recognition (and traffic growth) at just the right time.

A similar (also unplanned) thing happened when selling my last business. In the middle of that process, we had released a feature that really exploded our user growth. I don't think it was completely random that these things happened at the right time, however. I generally try to operate in such a way as to maximize my luck.

People would talk to me because of traction and track record

I waited three and half years before seeking funding, much longer than most people would. I basically waited until we had significant (not huge, but significant) traction in a large market. Traction trumps everything.

I'm not suggesting it's great to wait, and I realize most people cannot wait for a variety of reasons. I'm just highlighting that we did and it clearly helped get VCs talking to us.

We probably could have raised earlier

If you checked out our graph, it's nice, but looking back, it is unclear to me how different things would have been if I had tried to raise after reaching 1,000,000 direct searches a month instead of 7,000,000. When talking to people, for most, it didn't really seem that it would have made a huge difference, though there is of course no control.

However, the business difference in those two numbers is large. Granted they're both still very small numbers when considering the search market as whole, but moving orders of magnitude is really de-risking the business a great deal.

If you try to raise between significant milestones, unless you can show other reasons why you're killing it (the nice graph in our case), you're risking getting hit on valuation (or lack of funding) because your momentum is unclear and/or people can't perceive the real progress your making. On the other hand, after just reaching a milestone people care about, your momentum is large and people generally extrapolate what you're doing in your favor.

The subtext here is that it has to be a milestone people care about and not just one that you care about, even if you have great reasons to care.

There are a lot of VCs

Let's get to some numbers. I talked to 31 VC firms, 5 seed funds, and about 14 angels. These numbers do not include firms or people that I never actually connected with on the phone, on Skype video, or in person. There were countless more I didn't get to or weren't on my radar.

In the end we got funded by a big name, but I met plenty of lesser names from whom I would have gladly been funded. In a second I'd choose someone I like/respect/trust/think is a good fit at a no-name firm vs. someone I don't at a big name firm.

Your VC is essentially buying in as a co-founder

Would you chose a co-founder that sucks? Of course not willingly. It is a major reason startups die.

I treated the VC decision in a similar way. They're going to be with me long-term. They have a significant equity stake and other significant terms.

But I'm not just thinking negative here. Like a great co-founder, they can also help me in strategic ways at the right time.

I realize a lot of people don't think they have choices, but that's a bit of a fallacy. You are making choices by deciding who to talk to in the first place.

It's good to know people ahead of time

I only knew a handful of VCs coming into the process, but I wish I had known more. I just never made a concerted effort to meet them.

There is conflicting advice in the blogosphere about whether to take VC "informational interviews" or not. But from my perspective, people take you much more seriously if you are a known quantity. You see firms backing the same entrepreneur again and again. And more generally I think people do invest (at least more easily) in lines, not dots.

To blast or not to blast

I also got conflicting advice on how to seek VC intros. You can either a) get intros in concentric circles based on who you think is best (first tier, second, etc.), or b) try to get introduced all at once to everyone you want.

The idea behind the former is that you give the best people a sense you're coming to them somewhat exclusively because of fit, which they like and will be more inclined to listen/fund, while the idea behind the latter is to get people moving along faster because you're a "hot" deal.

Don't pitch ideal VCs first because you'll mess up

Ultimately, I decided to blast, but not after messing up some early pitches with people I really wanted to work with.

Quite frankly, I didn't have my pitch down and I'm not sure you can get it down without giving it to some real VCs and seeing what resonates and what falls flat.

Get your story straight

Another reason I messed up my early pitches is I was a bit wishy-washy on how much I wanted to raise and what I would do with the funds. Bad idea. People like confidence. Perhaps it is a good idea to have those exploratory meetings ahead of time, but I learned they should not be combined with a pitch because it is just communicating either you're not serious or you're not ready.

It's awkward to pitch people you know, but give them the real pitch

Yet another reason I messed up my early pitches is I treated people I knew too casually. I should have given them the real pitch. You can still ask them to introduce you to others that may be good fits after the fact, which is one of the reasons I wanted to talk to them about it as well.

But by changing the pitch into a friendly conversation, I was again sending a signal that they shouldn't take me seriously. And they didn't.

Use AngelList

I got significant value out of AngelList. Most importantly, I got high quality inbound requests from people I would have never connected with otherwise.

I also think we were perceived as somewhat hot by the VCs who hang out on AngelList because our startup was getting a lot of followers/introductions and going out on a lot of the associated emails.

Intro channel breakdown

Of the connections made, I broke them down into categories of how I got the initial introduction:

- I knew them and reached out: 4 VCs, 2 seed funds, 4 angels.
- Suggested intro from someone: 14 VCs, 2 seed funds, 5 angels
- AngelList inbound: 9 VCs, 1 seed fund, 5 angels
- AngelList outbound: 2 VCs
- Cold inbound: 2 VCs

Of the VCs, I broke them down into categories of how far I got with them:

- Talked with non-partners: 9
- Talked with a partner: 12
- Talked with multiple partners: 7
- Got a terms sheet: 3

Of the VCs I got to the multiple partners (MP) / terms sheet (TS) level, the original categories broke down like:

- I knew them and reached out: 2 MP, 1TS.
- Suggested intro from someone: 3MP, 1TS
- AngelList inbound: 1MP, 1TS
- AngelList outbound: 0
- Cold inbound: 1MP

Getting to the right partner initially really matters

Pretty soon after that first meeting, you need a partner to convince their other partners this is a deal the partnership should move forward with. Not one of the partners I talked to handed the deal off to another partner to manage, which says to me you have to figure out a priori which partner at a given firm would be most likely to get excited about your deal. Otherwise it is likely to be dead on arrival.

Often the right partner is unclear without talking to someone else

Like I said, when I started raising this round, I only knew a few VCs personally. I also thought that they may not want to fund or be a right fit for DuckDuckGo. Sure, I had heard of a bunch of VC firms, but when I went to their websites and looked at their team pages, I may have heard of one of their 2-10 partners. For all I knew, this guy I've never heard of would be the best fit for us.

So what I did was try to find the right partners by:

1. Asking the VCs I knew
2. Asking the angels I knew
3. Researching related investments

Having a good network to get intros really matters

As you can see from the intro categories, #1 and #2 yielded most of my introductions. If I didn't know these people, my job would have been much, much harder. Not only would it have been harder to meet people, but I wouldn't have known the right people to target.

A warm intro >> cold intro

Obvious.

Principal intro is OK if not right partner intro

Less obvious, and I think there is conflicting advice here, too. Some say don't bother if you can't get a partner intro. Partner intros always trump non-partner intros.

First of all, my funding stands as a counter-example, because I was introduced to Christina at USV, and that led to my funding.

But more broadly, if you don't know the right partner or don't have a partner intro, and you get introduced to a well-respected principal/associate/

analyst and they get excited about what you're doing, they can help a) determine the right partner; and b) rope them into the next meeting with good framing.

I never walked through my 6 slides

I have no idea how my demeanor or pitch varied from the countless ones VCs get. All I know is what I did. I had a six-slide deck that I would send beforehand.

Then I would spend my time telling my story, starting well before DuckDuckGo (briefly), and then taking them through what had happened until now, why I was raising money now, and what my future plans were.

I got very few No's

Like people say, most people just stop emailing you. They don't say no. I just kept moving forward, so beyond a thank you note after a meeting, I didn't really press people. Ultimately I wanted to see who was excited about me, so I figure the least they could do is follow-up like they said they were going to do.

Some firms did follow-up promptly with no's and specific reasons, and I was really grateful for that. They were:

- Andreessen Horowitz
- Matrix Partners
- SV Angel
- Redpoint
- Mergenthaler
- Greylock
- Charles River Ventures
- DFJ

Having multiple term sheets really matters

I realize most companies can't get multiple term sheets. But if you can, they can really be used as a forcing function to move people along, and that's what happened in my case.

It's a double-edged sword though. It moved people to say yes or no more quickly. As for the no's, sometimes people felt things were moving too fast for them and they couldn't catch up.

Blasting matters to align with timelines

The process takes time, but once you get the term sheet, there is more pressure to move quickly. To get multiple term sheets, you need to get people somewhat aligned in the process, and the best way to do that is to start them all at the same time.

I did not do that very well for a couple of reasons.

1. I didn't blast right away
2. I tried this in July/August

VCs really do take vacations in August

Some of my initial meetings were delayed a month or more because of vacation schedules. Ultimately I have no regrets of course, but it should give you pause to raise in August if you are trying to align everyone time-wise.

Some partners I really liked

I met a lot of people I really liked in this process. Here are a few (besides everyone at USV of course).

- Todd Hixon and John Backus at New Atlantic Ventures
- Michael Dearing at Harrison Metal
- Saar Gur at Charles River Ventures
- Dan Beldy at Steamboat

And I already knew these partners but can't speak more highly of them:

- Antonio Rodriguez at Matrix
- Gilman Louie at Alsop Louie

I did not push for a bidding war

I did have multiple term sheets at the same time, and that did work as a forcing function. But I did not try to actively use it to create some kind of bidding war. Instead, I went in with terms in mind that I thought were fair for everyone, and I tried to stick to those terms.

I focused on what value the VC would bring

I really did try to get the right VC in the deal, and so I tried to focus on figuring out how individual firms and people could be particularly useful to DuckDuckGo over the life the company.

I did reference checks

I talked to a bunch of portfolio companies that had various VCs on their board. I asked them all sorts of questions about that relationship and their experiences with the particular partner and firm in general. This information was very interesting. Entrepreneurs really do stick together, and I found people would speak freely — even if I didn't know them very well (or at all) ahead of time.

I didn't travel much

I'm sure I could have gotten more interest and higher terms if I did a huge road show. But I hate travelling, so I did most of this entire raise via Skype video and phone. I did make two trips to New York and one to DC. And a few VCs came down to Philadelphia to meet me.

I'm really not sure how much that turned people off or not. I'm sure it seemed different. I got a sense for most West-coast VCs it was a big turn off and that location does matter, even though they do invest in New York a decent amount.

Having a mentor really helped

Someone who has been there, done that is great. I actually talked to people and picked their collective brains, but my uncle (again) proved invaluable throughout the process. I was talking to him constantly. I suggest you find someone like that. If you've previously raised an angel round, it's probably one of those angels.

Congratulations

I got more congratulations from this funding than from any previous DuckDuckGo milestone. I understand why, and I thank you, everyone! But it still feels wrong somehow.

Conclusion

I realize that's a lot to take in, and I don't expect anyone to make it this far. I wanted to write this for people in the process. ■

Gabriel Weinberg is the founder of DuckDuckGo, a search engine. He is also an active angel investor, based out of Valley Forge, PA.

Reprinted with permission of the original author.
First appeared in *hn.my/raisefund* (gabrielweinberg.com)

Vim Text Objects: The Definitive Guide

By JARED CARROLL

To edit efficiently in Vim, you have to edit beyond individual characters. Instead, edit by word, sentence, and paragraph. In Vim, these higher-level contexts are called text objects.

Vim provides text objects for both plaintext and common programming language constructs. You can also define new text objects using Vim script.

Learning these text objects can take your Vim editing to a whole new level of precision and speed.

Structure of an Editing Command

In Vim, editing commands have the following structure:

```
<number><command><text object or motion>
```

- The **number** is used to perform the command over multiple text objects or motions, e.g., backward three words, forward two paragraphs. The number is optional and can appear either before or after the command.
- The **command** is an operation, e.g., change, delete (cut), or yank (copy). The command is also optional, but without it, you only have a motion command, not an edit command
- The **text object** or **motion** can either be a text construct, e.g., a word, a sentence, a paragraph, or a motion, e.g., forward a line, back one page, end of the line.
- An **editing command** is a command plus a text object or motion, e.g., delete this word, change the next sentence, copy this paragraph.

Plaintext Text Objects

Vim provides text objects for the three building blocks of plaintext: words, sentences and paragraphs.

Words

- `aw` → a word (includes surrounding white space)
- `iw` → inner word (does not include surrounding white space)

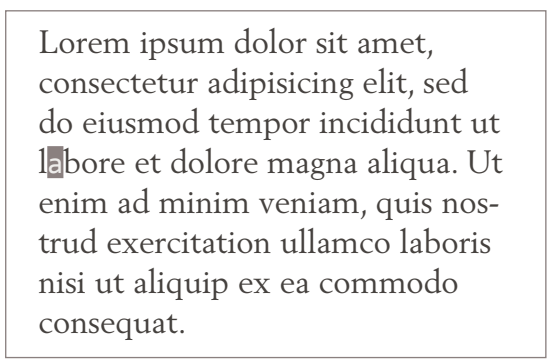
`daw`

Text objects beginning with `a` include the surrounding white space in the text object, those starting with `i` do not. This convention is followed by all text objects.

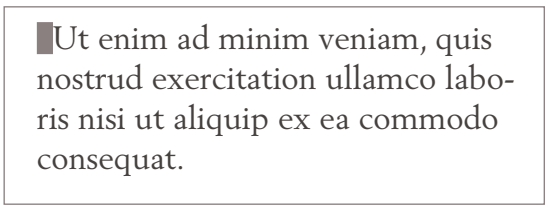
The motion `w` may seem similar to the text object `aw`. The difference is in the allowed cursor position. For example, to delete a word using `dw` the cursor must be at the start of the word, any other position would delete only part of the word; however, `daw` allows the cursor to be at any position in the word.

Sentences

- `as` → a sentence
- `is` → inner sentence



`cis`

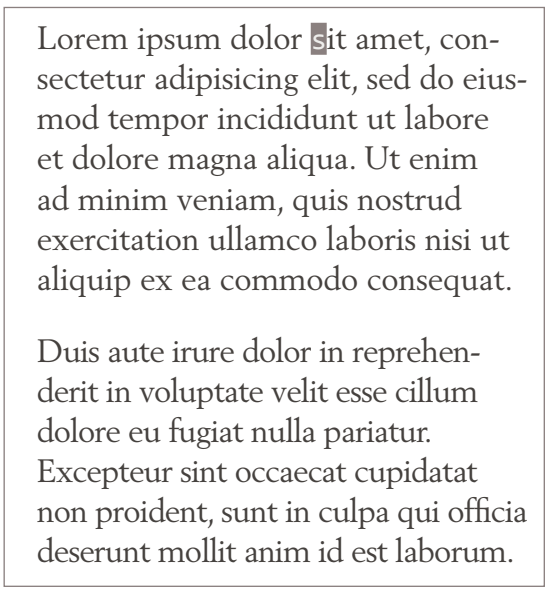


Notice how the "inner" text object does not include the trailing white space.

Like `aw`, as offers the same cursor position advantage over its motion counterparts `( )`, forward and backward a sentence. To operate on the entire previous sentence, `(` requires the cursor to be at the end of the sentence; to operate on the entire next sentence, `)` requires your cursor to be at the start of the sentence.

Paragraphs

- `ap` → a paragraph
- `ip` → inner paragraph



dap

Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in culpa qui officia deserunt mollit anim id est laborum.

Again, `ap` and `ip` provide the same cursor position advantage that Vim's sentence and word text objects provide: your cursor can be anywhere within the paragraph in order to operate on it.

Motion Commands vs. Text Objects Commands

A command using a motion, e.g., `cw`, operates from the current cursor position. A command using a text-object, e.g., `ciw`, operates on the whole object regardless of the cursor position. We saw this behavior in each of the various plaintext text objects. Although this requires one more character, it saves you the time and effort of moving the cursor into the "right" position.

Programming Language Text Objects

Vim provides several text objects based on common programming language constructs.

Strings

- `a"` → a double quoted string
- `i"` → inner double quoted string
- `a'` → a single quoted string
- `i'` → inner single quoted string
- `a`` → a back quoted string
- `i`` → inner back quoted string

```
puts 'Hello "world"'
```

ci"

```
puts 'Hello ""'
```

Notice that the cursor was not even within the double-quoted phrase ("world"); the command defaulted to changing the first double-quoted phrase in the line.

```
puts 'Hello "world"'
```

ci'

```
puts ''
```

Current line searches offer an alternative way to delete a quoted phrase. Continuing with the previous example, placing the cursor on the first `'` and executing `ct'` would delete the contents of the single quoted string and place us in insert mode. However, this is less flexible than using a text object, because it requires the cursor to be on the opening `'`.

A search pattern `/'` could also be used, but it, too, requires the cursor to be on the opening `'`. It also deletes the closing `'`.

It's best to use search commands for searching and not editing.

Parentheses

- `a)` → a parenthesized block
- `i)` → inner parenthesized block

```
Project.all(:conditions => {
:published => true })
```

`da)`

```
Project.all
```

Both of these text objects are also available as `ab` and `ib`; however, I find these less intuitive than using the version that includes a parenthesis character.

The `%` motion is another way to match a pair of parentheses. Entering `%` on an opening parenthesis will move the cursor to the closing parenthesis. Combined with `a` command, this can provide the same functionality as `a)`, e.g., `c%` is equivalent to `ca)`. However, the disadvantage to using `%` is that the cursor must be on the opening or closing parenthesis; with `a)` the cursor can be anywhere on or the parenthesized phrase. There is also no way to replicate `i)` using `%`.

Brackets

- `a]` → a bracketed block
- `i]` → inner bracketed block

```
(defn sum [x y]
  (+ x y))
```

`di]`

```
(defn sum [
  (+ x y))
```

The `%` movement can also be with `[ ]`. However, it has the same limited flexibility when using it with `( )`.

Braces

- `a}` → a brace block
- `i}` → inner brace block

```
puts "Name: #{user.name}"
```

`ci}`

```
puts "Name: #{}"
```

Both of these text objects are also available as `aB` and `iB`; however, I find these less intuitive than using the version that includes a brace character.

Again, the `%` movement can also be with `{ }`. However, it has the same limited flexibility when using it with `( )` or `[ ]`.

Markup Language Tags

- `at` → a tag block
- `it` → inner tag block

```
<h2>Sample Title</h2>
```

`cit`

```
<h2></h2>
```

Notice that the cursor was not even within the `<h2>`. This is a very efficient way to quickly replace tag content.

- `a>` → a single tag
- `i>` → inner single tag

```
<div id="content"></div>
```

`di>`

```
<></div>
```

This text object can be used to quickly operate on a single tag and its attributes.

Vim Scripts Providing Additional Text Objects

Using Vim script, it's possible to create new text objects. Here's a few of my favorite scripts that introduce new programming language text objects.

CamelCaseMotion

CamelCaseMotion provides a text object to move by words within a camel or snake-cased word.

- `i,w` → inner camel or snake-cased word

```
BeanFactoryTransactionAttrib-
uteSourceAdvisor
```

`ci,w`

```
FactoryTransactionAttribute-
SourceAdvisor
```

VimTextObj

VimTextObj provides a text object for function arguments.

- `aa` → an argument
- `ia` → inner argument

```
foo(42, bar(5), 'hello');
```

`cia`

```
foo(42, , 'hello');
```

Indent Object

Indent Object provides a text object based on indentation level. This script is aimed at programming languages that use significant whitespace to delimit code blocks, e.g., Python, CoffeeScript, because its text object does not include the line after the last line of the indentation level.

- `ai` → the current indentation level and the line above
- `ii` → the current indentation level excluding the line above

```
def foo():
  if 3 > 5:
    return True
  return "foo"
```

`dai`

```
def foo():
  return "foo"
```

Ruby Block

Ruby Block provides a text object based on a Ruby block, i.e., any expression that is closed with the end keyword.

- `ar` → a Ruby block
- `ir` → inner Ruby block

```
hash.each do |key, value|
  puts key
  puts value
end
```

`cir`

```
hash.each do |key, value|

end
```

Vi Command Line Editing

If you use Vi command line editing in your shell, enabled with `set -o vi` in bash and `bindkey -v` in zsh, Vim's text objects are not available. Text objects were introduced by Vim, but shell command line editing is based on Vi.

Precision Editing

Vim's text objects provide an incredible level of precision. The key is to try to always edit by text objects. Editing by motions e.g., by part of a line, to the next occurrence of a character, is tedious, clumsy, and slow. Instead of correcting a misspelling character by character, change the entire word and re-type it.

Don't be discouraged by the large number of text objects. Their conventions make them intuitive and easy to learn. After some practice, like every other Vim command, they'll quickly become just another muscle memory. ■

Jared Carroll is a programmer currently practicing agile development at Carbon Five in Los Angeles, CA. He has been coding in a variety of programming languages at start ups and consultancies for the past 6 years. His main interests are ergonomics and productivity.

Reprinted with permission of the original author.
First appeared in *hn.my/textobjects* (carbonfive.com)

Python for the Web

By RICH JONES

Python is the best language in the world for interacting with the web, and I'm going to show you why.

This article will give an extremely high-level overview of how to use Python for the web. There are many ways you can interact with the web using Python, and this article will cover all of them. This includes Python web scraping, interacting with APIs (Application Programming Interfaces), and running your own Python web site using Python server software. There are many ways to do all these things in Python, but I'm going to show you how to do it the right way using the most modern techniques.

Interacting with Websites and APIs Using Python

The single best package for interacting with the web using Python is "Requests" by Kenneth Reitz. I really cannot stress what a good library this is. I use it every single day of my life and I absolutely love it. It is the reason that Python is the best language for the web.

First, you'll need to install it. The best way to do this is using "pip," the Python package manager. If you don't have pip, read this article [hn.my/pip] and follow the instructions, or, if you are on Windows, look at this post on Stack Overflow [hn.my/pipwin].

Once you have pip installed, run:

```
pip install requests
```

And now you have Requests installed! You may need to run this as "sudo" if you're on Linux or OSX. Now let's look at a few examples.

The two methods you'll need the most are GET and POST. GET does exactly what it says, it gets a web page. POST is similar, only it sends information to a web page.

First let's take a look at GET. Let's say we want to grab all of Gun.io's front page.

```
import requests

r = requests.get('http://gun.io')
print r.content
```

That's it! In only three lines of Python, you can grab a whole webpage and print it to the screen. Awesome!

Now let's look at a slightly more complicated example. Let's try a case where we have to use a username and password.

```
import requests

r = requests.get('https://api.
github.com', auth=('YOURUSERNAME',
'PASSWORD'))
print r
```

Here, YOURUSERNAME and YOURPASSWORD will be sent as login credentials to the server.

Now, let's try a POST request to send some data TO the server. This is for the case where there is a form, and you want to use Python to fill in the values.

```
import requests
url = 'https://testexample.com/
form'
data={'title': 'RoboCop',
'description': 'The best movie
ever.'}
r = requests.post(url, data=data)
print r
```

This code will send the values "RoboCop" and "The best movie ever" for the fields "title" and "description," respectively. You can use the "auth" parameter from the previous example if you are posting to a password-protected form.

Processing JSON in Python

Many times you interact with an API in Python, you will be given a response in a form called JSON, or JavaScript Object Notation. JSON is almost identical to the Python dictionary format. The best way to interact with JSON in Python is by using the "simplejson" Python library, which you can find documentation for here [hn.my/simplejson]. Again, use pip to install it like so:

```
pip install simplejson
```

Let's take a look at an example.

```
import requests
import simplejson

r = requests.get('https://github.
com/timeline.json')
c = r.content
j = simplejson.loads(c)

for item in j:
    print item['repository']
['name']
```

This code will get a list of recent events from GitHub, in JSON format, and parse that JSON using Python. As the resulting object (in this example, "j") is a Python dictionary, we can loop over it and print the information it contains. This code will then print out the name of each repository for each item in the response.

Scraping the Web Using Python

Unfortunately, we can't always interact with the web in a nice format like JSON. Most of the time, websites only return HTML, the kind that your browser turns into the nice-looking webpages you see on your screen. In this case, we have to do what's called "scraping," taking that ugly HTML and turning it into usable data for our Python program.

The best way to do this is by using a Python package called LXML. If I had to describe LXML, I would call it shitty and awesome. LXML is extremely fast and very capable, but it also has a confusing interface and some difficult-to-read docs. It is certainly the best tool for the job, but it is not without fault.

Let's say there is a webpage that has a value you want to get into your Python program. You know from looking at the source of the webpage that the value you want is inside an element which has a specific "id" attribute. Let's use LXML to get that value.

First, install it using pip:

```
pip install lxml
```

Okay, now let's try it.

```
import requests
import lxml
from lxml import html

r = requests.get('http://gun.io')
tree = lxml.html.fromstring(r.
content)
elements = tree.
get_element_by_id('frontsubtext')
for el in elements:
    print el.text_content()
```

This code uses Requests (from before) to get our webpage. Then, it uses the HTML parser in LXML to get the "tree" of parsed HTML elements. The next line calls the "Get Element By Id" function to return a list of all elements which have the id value of "frontsubtext." Then, we iterate over the items in that list, and print the text content of each element. Ta-da!

Python Web Sites

The other side of using Python on the web is for making websites. The best way to do that is to use a web "framework" called Django [djangoproject.com].

Now, Django can be tricky. Django isn't the fastest or the easiest way to get your Python code executing on the web, but Django has the largest community and the most documentation available, so it's the best thing to learn

in the long run. This is going to be a very, very brief introduction to Django; I'm just going to teach you how to get your Python code to return a result to an HTML web page.

So, let's get started!

First things first: install Django using pip. This should be easy by now!

```
sudo pip install django
```

Okay, now you've got Django installed on your system. Let's make a new Django project. Let's say we want a website which returns an uppercase version of a string we pass, and we're going to call it UppercaseMaker. So, call this to make a new Django project:

```
django-admin.py startproject
UppercaseMaker
```

Then, go into the directory it made:

```
cd UppercaseMaker
```

You'll see some files in there, like settings.py and urls.py. We'll get back to those in a second. Now that you're in the folder, you'll need to make a new "application." In Django, applications are where the actual work is done. Let's make one called "upper."

```
django-admin.py startapp upper
```

For this application to be activated in our Django project, we'll need to edit settings.py and add it to the list of INSTALLED_APPS. So, you need to change the settings.py (at around line 111) so that it looks like this:

```
INSTALLED_APPS = (
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.sites',
    'django.contrib.messages',
    # Uncomment the next line to
enable the      # admin:
    # 'django.contrib.admin',
    # Uncomment the next line to
enable admin
    # documentation:
    # 'django.contrib.admindocs',
     'UppercaseMaker.upper',
)
```

While you're here, you should also change the TEMPLATE_DIRS variable so that it looks like this:

```
import os
TEMPLATE_DIRS = (
    os.path.join(os.path.dir-
name(__file__), 'templates'),
)
```

This will make it so when Django needs to render templates, it will look in the "templates" directory of your project's folder.

Now in your project directory, you'll see that there is a directory called "upper." Let's go in and take a look. You'll see there's a file called "views.py" — that's where the magic happens. Let's put some code in it.

```
from django.shortcuts import
render_to_response

def home(request, input="No input
supplied"):
    output = input.upper()
    return render_to_
response('home.html', {'output':
output})
```

So this is defining a function called "home," which takes two parameters, "request," which contains information about the request which was sent to the server (information about the user, their browser, etc), and a string called "input," which defaults to "No input supplied." The next line is pretty obvious: it takes the input string, puts it in uppercase, and makes a variable called "output."

Then we pass that in a dictionary to a Django function called "render_to_response," which takes a template file and a dictionary of variables and makes it into the nice HTML you see as the final website. We haven't looked at the template file yet, so let's do that now.

Go back to the project directory and make a new folder called "templates" and inside it, make a file called "home.html," and put this in it:

```
<html>
  <head><title>{{output}}</
title></head>
    <body>
      Your output is: {{output}}.
    </body>
</html>
```

This is an extremely simple HTML page which takes whatever value we put in the "output" variable from our views.py and puts it on the screen wherever we wrap it with double curly braces.

Only one thing left now! Let's take a look at urls.py in the project folder. Put this in it:

```
from django.conf.urls.defaults
import patterns, include, url

urlpatterns = patterns('',
    url(r'^(?P<input>[^/]+)$',
'UppercaseMaker.upper.views.
home'),

)
```

This says, for the empty path (the blank space in between the "^" and the "("), call the function "UppercaseMaker.upper.views.home" and pass it the trailing value and call it "input." So, when somebody visits "http://www.ourwebsite.com/test" , the value "test" is sent to our "home" function from before, made uppercase, and printed to the screen.

Now, you can try this for yourself by running this command from your project's directory.

```
python manage.py runserver
```

Then, in your web broswer, go to the URL "http://localhost:8000/test", and you should see the output, "Your output is: TEST." on the screen.

Hooray! You're now executing your own Python code as a website. Pretty cool!

I've made this example as a git repository, so if you want to have your own copy of this example to play with, execute

```
git clone https://Gunio@github.
com/Gunio/UppercaseMaker.git
```

Conclusion

So there you have it: a very high-level introduction to the major ways you'll be interacting with the web using Python. This guide is by no means meant to be exhaustive, but hopefully you are now on the right path.

The key take-aways are: to make HTTP requests, use the Requests library. To parse JSON, use simplejson. To parse HTML, use LXML. And to serve your own Python websites, use Django. Other guides will tell you to use things like "urllib2" and "BeautifulSoup" — don't waste your time! Those packages and the tutorials which recommend them are now outdated. Requests and LXML are the best tools for the job. Django is still the best Python web framework, but make sure that any tutorials you are reading are compatible with the version of Django you are using, as the project can change quite quickly, and there are lots of outdated Django tutorials on the web. ■

Rich Jones is a traveling hacker and transparency activist. He is the director of the OpenWatch police monitoring project. His new startup is *Gun.io*, a crowd-funding platform for hackers to hire each other for small tasks.

Reprinted with permission of the original author.
First appeared in *hn.my/pythonweb* (gun.io)

The MOS 6502 and the Best Layout Guy in the World

By RUSS COX

The MOS 6502 was ubiquitous in its day. The 6502 and its slight variants were at the heart of the Apple II, the Atari 2600, the BBC Micro, the Commodore 64, and the Nintendo Entertainment System, among others. It's amazing to think that all five — each a very influential system in its own right — were built around the same chip.

The 6502 was the brainchild of Chuck Peddle, at the time an engineer with Motorola. Peddle was one of the engineers who worked on the Motorola 6800, and one of his jobs was to, well, peddle the 6800 to customers. The customers loved everything about it except its $300 price tag. Peddle tried to convince management at Motorola to create a lower-cost microprocessor, but Motorola didn't want to have such a chip cut into their not insubstantial profits from the 6800, and they told him in no uncertain terms that they wouldn't build such a chip. In response, Peddle and a handful of other 6800 engineers left Motorola and built one themselves. It was the MOS Technology 6502 and sold for $25. Even though both the 6800 and 6502 had a clock rate of 1 MHz, the 6502 had a minimal instruction pipeline that overlapped the fetch of the next instruction with the execution of the current one when possible, giving it a significant performance boost. And of course it sold for ten times less. So it ended up everywhere.

The story of the 6502 makes up the first chapter of Brian Bagnall's On the Edge: the Spectacular Rise and Fall of Commodore. My favorite part of the description of the development of the 6502 is the actual chip layout. These days, you can't design and lay out a computer chip without a computer. An Intel Core 2 chip has hundreds of millions of transistors. The 6502 had 3,510, and an engineer — a person, not a computer — had to draw each one by hand to lay out the chip. Mainly it was a single engineer, Bill Mensch.

But it gets better. Once the layout was completed and double-checked — a process that meant months using a ruler! — it still had to be converted into a Rubylith photomask that would etch the right patterns onto the silicon. The photomask for the 6502 was the size of a large table — large enough that the engineers crawled around on top of it to perform the job of cutting the layout out of the mask, all the while being careful to wear clean socks with no holes, so that stray toenails didn't insert traces in the mask where they didn't belong.

The most amazing part about the whole process is that they got the 6502 right in one try. Quoting On the Edge:

> *Bil Herd summarizes the situation. "No chip worked the first time," he states emphatically. "No chip. It took seven or nine revs [revisions], or if someone was real good, they would get it in five or six."*

Normally, a large number of flaws originate from the layout design. After all, there are six layers (and six masks) that have to align with each other perfectly. Imagine designing a town with every conceivable layer of infrastructure placed one on top of another. Plumbing is the lowest layer, followed by the subway system, underground walkways, buildings, overhead walkways, and finally telephone wires. These different layers have to connect with each other perfectly; otherwise, the town will not function. The massive complexity of such a system makes it likely that human errors will creep into the design.

After fabricating a run of chips and probing them, the layout engineers usually have to make changes to their original design, and the process repeats from the Rubylith down. "Each run is a couple of hundred thousand [dollars]," says Herd.

Implausibly, the engineers detected no errors in Bill Mensch's layout. "He built seven different chips without ever having an error," says Peddle with disbelief in his voice. "Almost all done by hand. When I tell people that, they don't believe me, but it's true. This guy is a unique person. He is the best layout guy in the world."

The first chapter of On the Edge is posted on Bagnall's web site [hn.my/bagnall]. The chapter says that Peddle "created a concept called pipelining," which could be interpreted as saying that the 6502 was the first pipelined processor and that Peddle invented it. Does anyone know?

Fast forward thirty years. Computers are now old enough that there can be significant interest in maintaining the history of these old, venerable designs. The actual paper designs of the 6502 are long gone.

A team of three people — Greg James, Barry Silverman, and Brian Silverman — accumulated a bunch of 6502 chips, applied sulfuric acid to them to strip the casing and expose the actual chips, used a high-resolution photomicroscope to scan the chips, applied computer graphics techniques to build a vector representation of the chip, and finally derived from the vector form what amounts to the circuit diagram of the chip: a list of all 3,510 transistors with inputs, outputs, and what they're connected to. Combining that with a fairly generic (and, as these things go, trivial) "transistor circuit" simulator written in JavaScript and some HTML5 goodness, they created an animated 6502 web page that lets you watch the voltages race around the chip as it executes. For more, see their web site *visual6502.org*.

Oh, and it actually works. They applied the same technique to build the transistor map for an Atari 10444D TIA chip, which connected the 6502 to the television in the original Atari 2600, and then they simulated both chips together and were able to run actual Atari 2600 games. So the transistor map is probably (very close to) correct. More impressively, they got to that point without debugging. Their SIGGRAPH 2010 abstract explains that there were only 8 errors in the map of 20,000 components, and all the errors were spotted during the vectorization process. History repeats itself.

Michael Steil took the circuit information and started looking closely at how the chip did what it did. At last week's Chaos Communication Congress 27C3 conference, he gave a 50-minute talk that introduces the 6502 architecture and then uses the circuit diagram to explain various details and undocumented features of the chip. The original announcement is at Steil's blog [hn.my/steil]. There is a version of the talk on YouTube [hn.my/27c3], and in the blog comments you'll find links to higher-resolution copies. It's well worth watching in any form, and it's fun to see how much Peddle, Mensch, and the rest of the 6502 team packed into that tiny number of transistors. ■

Russ Cox wrote his first programs on a Commodore 64 powered by a MOS 6502. Today, he works at Google on Go. His blog is *research.swtch.com*

Reprinted with permission of the original author.
First appeared in *hn.my/6502* (swtch.com)
Photo of MOS 6502 Processor taken by Dirk Oppelt.

P-III Autopsy

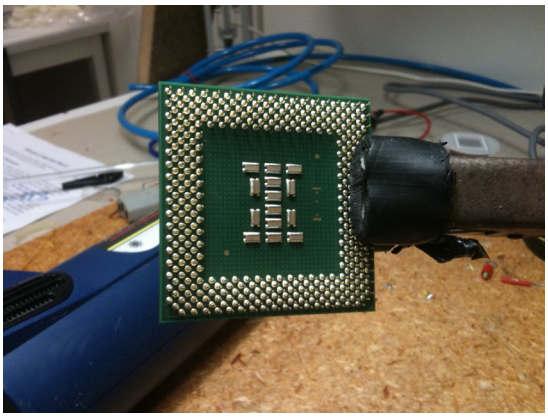
By KRISTIAN STORM

For teaching a course I needed to take a closer look at a CPU. I asked around and got my hands on an old P-III Coppermine that was about to get thrown out. I'll start with a disclaimer: I know virtually nothing about CPUs, so if I claim something to be true, it probably isn't.

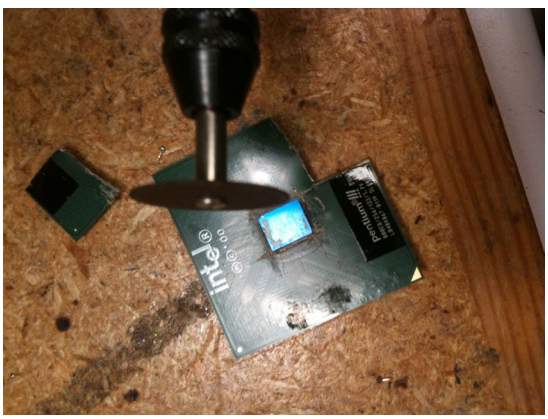
The first challenge is to get the actual silicon processor chip off of the plastic bonding board. In the picture beside, the blue thing you see is the back side of the processor chip. When the processor is finished, it is turned upside down and bonded to the green circuit board. This allows the metal pads on the silicon chip and the pads on the circuit board to join, creating a connection (this is one of those claims...). I believe that the CPU at that stage is heated up in order to melt the joints and thereby solder them together.

I figured I should be able to remove the chip by heating it. I first tried using a heat gun, but that just made some bad smelling fumes. I instead turned to brute force and used a power-saw to cut out the part containing the actual chip. Using pliers I managed to get a few pieces off of the board and got the rest off by using a scalpel.

The blue part in the middle is the actual Si chip. I needed to remove it in order to further inspect the CPU.



Back side of the circuit board containing the CPU. Each pin you see (give or take) should be connected to a pad on the silicon chip.



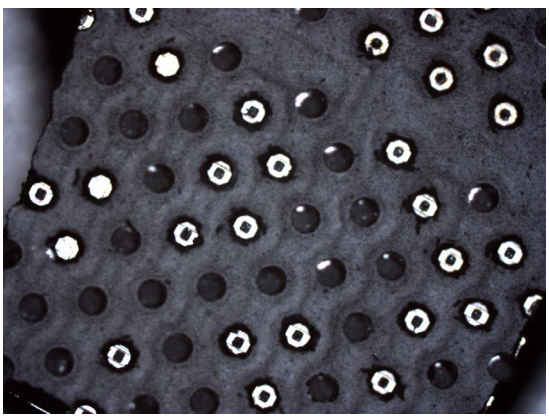
A saw comes in handy sometimes....

Below you can see the result. On the bottom side of the piece that came off, you can see all the connector pads that were previously connected to the pins on the backside of the circuit board.

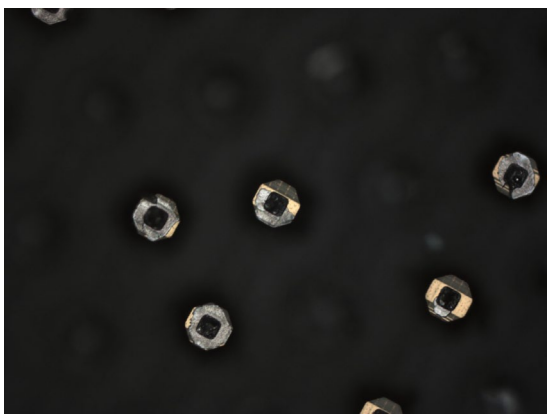
A piece of the processor chip came off.

This is the piece that came off. It's been flipped so that the side you see was originally facing towards the circuit board. Each little dot contains a metal pad that connects the interior of the chip to the leads on the board.

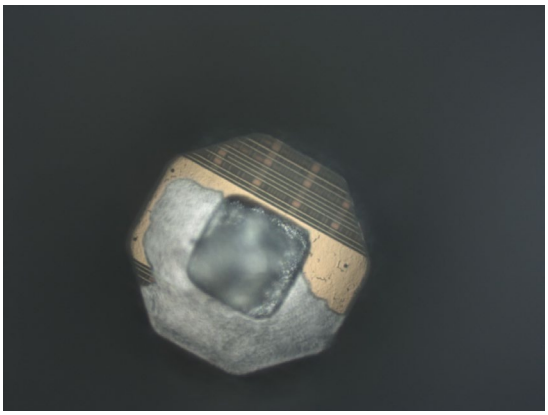
Now the interesting part begins. I looked at the piece above in an optical microscope. The picture below shows an enhanced version of the little dots in the above picture.



This is a piece of the processor. The side you're seeing was once facing towards the plastic circuit board. Each little hole is a metal pad connecting the interior of the chip to a macroscopic lead on the circuit board.



Looking closer, we start seeing some structure inside the holes.



A processor contains many layers of metal leads in order to connect the transistors at the surface of the silicon chip into useful units. The metal layers are clearly visible through the small holes in the chip.

Furthermore, by changing the focus of the microscope, we can see multiple layers within each hole.

Focus is on upper layer.

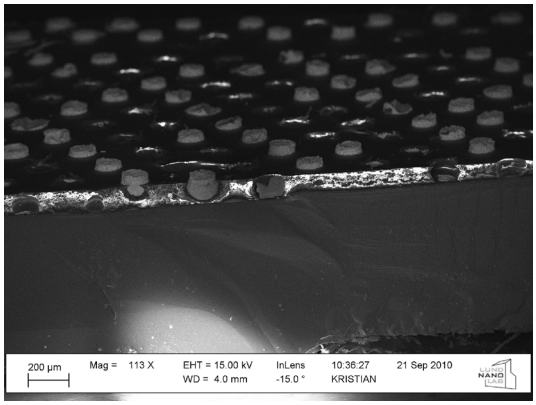
Focus is on bottom layer.

Focus is on middle layer.

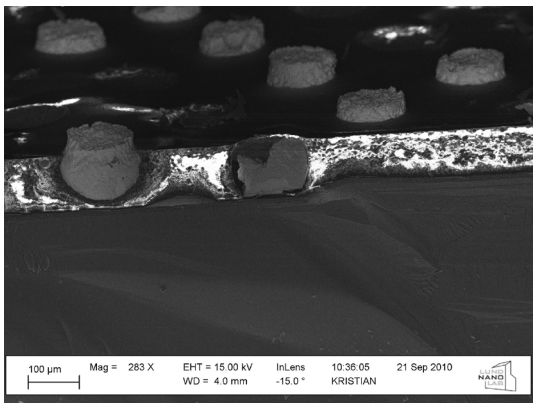
In a CPU there are multiple layers of sandwiched metal leads going down to the transistors at the bottom (at the surface of the silicon wafer). I believe what we're seeing is simply those different layers.

Since optical microscopy doesn't show very much detail, I decided to load the chip into a scanning electron microscope (SEM).

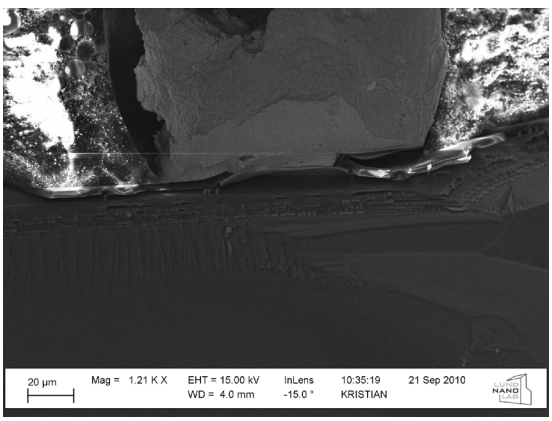
What I did was to cleave the chip into smaller pieces. This way I can peek from the side of the chip and get some cross-sectional images. Below is a series of images that shows a zoom-in on the surface of the Si chip. For some reason I had a lot of trouble getting a good focus. I'm not sure what kind of Si is used for these CPUs, but if it's a non-conducting silicon, the electron beam in the SEM can charge the material, making it difficult to focus. Another possibility is that all the plastic encapsulation material caused charging effects.



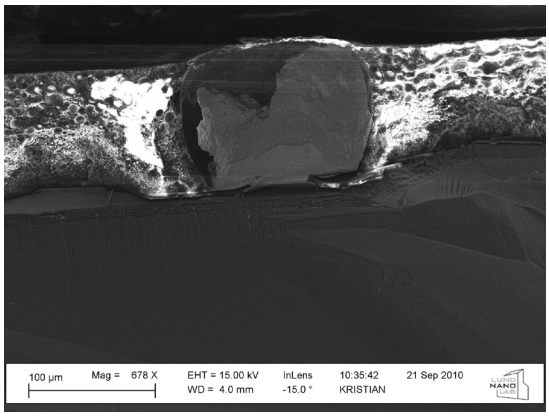
You're looking at the processor chip from the side. In the top part of the image you see the metal pads that were once connected to leads on the circuit board when the chip was bonded face down.



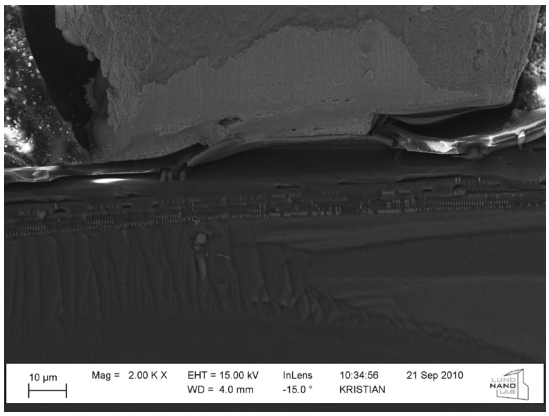
We still don't see a lot. The light stuff between the metal pads is probably some kind of polymer used to fill up the space.

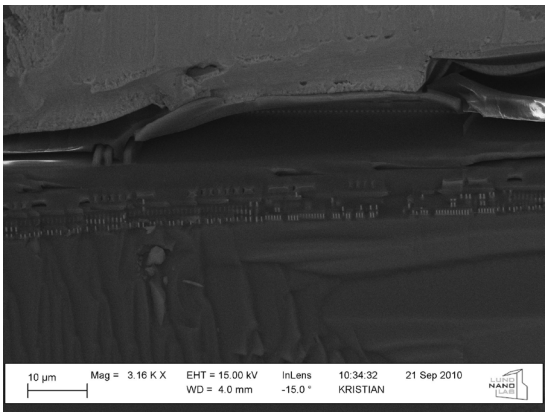


We start to get more detail at the surface of the silicon chip (bottom part of the image. The Si starts somewhere at the bottom of the large metal blob). The texture in the polymer filler (next to the metal blob) could be due to something being mixed into the polymer to increase its thermal conductivity.

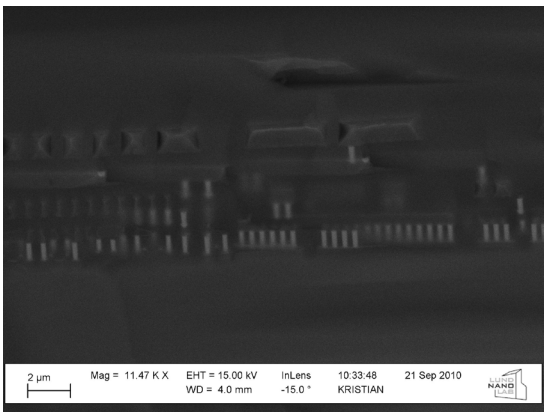


We start seeing something below the metal blob. The vertical lines that are barely visible would be the multiple layers of metal leads. To the right you see the same structure, but from another angle, since the cut changes direction.

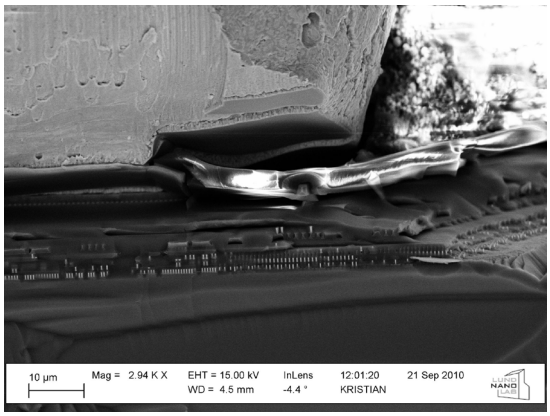




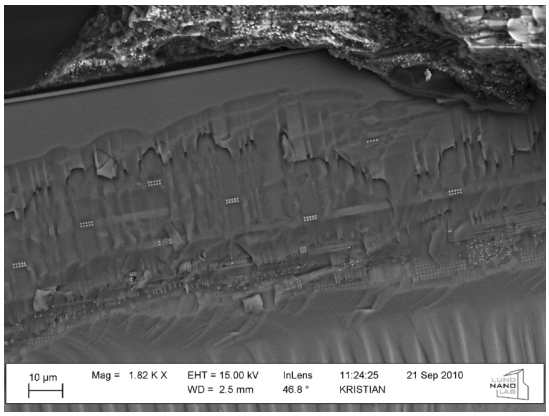
At this point they are clearly visible. I count about six layers of metal leads visible in the image.



The feature size of the lowest metal layer is around 200-250nm. Since the P-III started out at 250nm process but developed into 180nm (according to Wikipedia), the transistor layer must be fairly close to the lowest visible metal layer in the image.



Just a nice overview.



This is not a cross-sectional image. It is taken from the top (along the sample normal, for those of you who speak science). I accidentally chipped the processor, and this is how it looks. Several of the metal layers are visible, and as we go down in the image, we go down through the metal layers. I would guess that the bright spots you see are vias connecting leads lying in different layers. ■

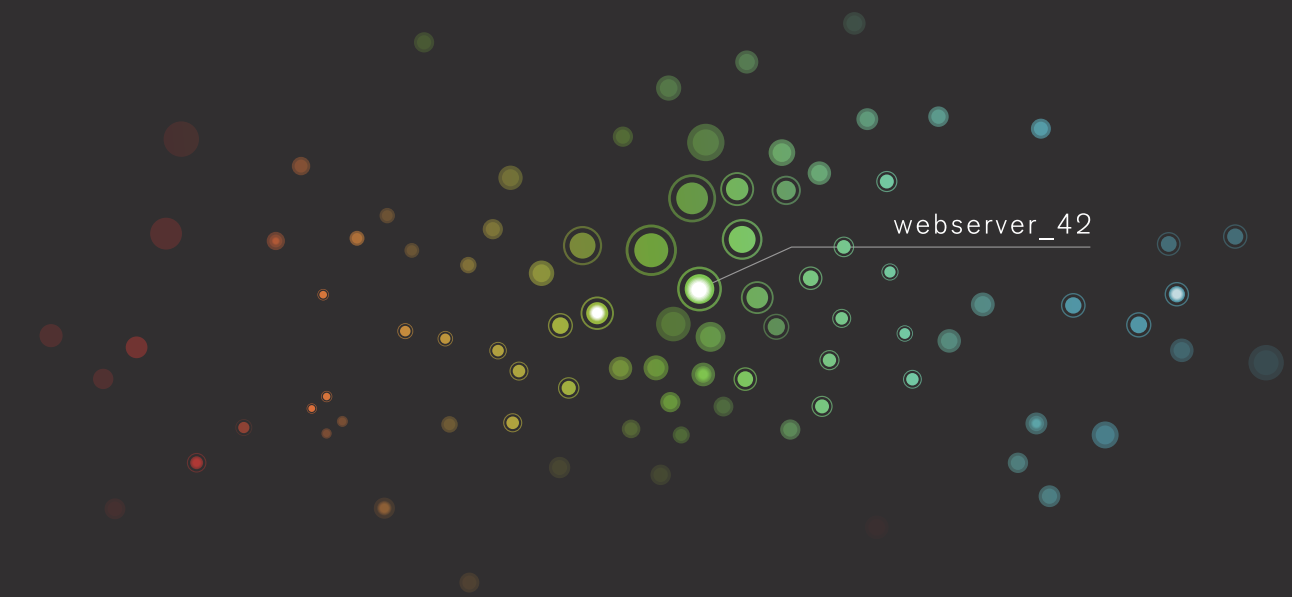
Kristian is currently part-time doing a PhD in solid state physics and nano-devices at Lund University in Sweden developing nanowire devices, and part-time working at a company developing the first generation nanowire LED technology. As a hobby, he enjoys playing around with robotics, building various remote-controlled and autonomous aerial vehicles.

Reprinted with permission of the original author.
First appeared in *hn.my/p3* (sciencystuff.com)


Trying to read your customers' minds?
Our simple engagement tools help you understand your customers, prioritize feedback, and give great customer support even faster. Spend more time building a product your customers will love!
uservoice
Get 50% off your first 3 months* with the code mindreader at UserVoice.com.
* Offer good for new accounts if used before 12/31/2011.

These are your servers

These are your servers on Cloudkick



Any questions?

cloudkick.com
415.779.5425

support for 8 clouds + dedicated hardware

cloudkick

the best way to manage the cloud

Dream. Design. Print.

MagCloud, the revolutionary new self-publishing web service by HP, is changing the way ideas, stories, and images find their way into peoples' hands in a printed magazine format.

HP MagCloud capitalizes on the digital revolution, creating a web-based marketplace where traditional media companies, upstart magazine publishers, students, photographers, designers, and businesses can affordably turn their targeted content into print and digital magazine formats.

Simply upload a PDF of your content, set your selling price, and HP MagCloud takes care of the rest—processing payments, printing magazines on demand, and shipping orders to locations around the world. All magazine formatted publications are printed to order using HP Indigo technology, so they not only look fantastic but there's no waste or overruns, reducing the impact on the environment.

Become part of the future of magazine publishing today at www.magcloud.com.

25% Off the First Issue You Publish

Enter promo code **HACKER** when you set your magazine price during the publishing process.

Coupon code valid through February 28, 2011.
Please contact promo@magcloud.com with any questions.

MAGCLOUD