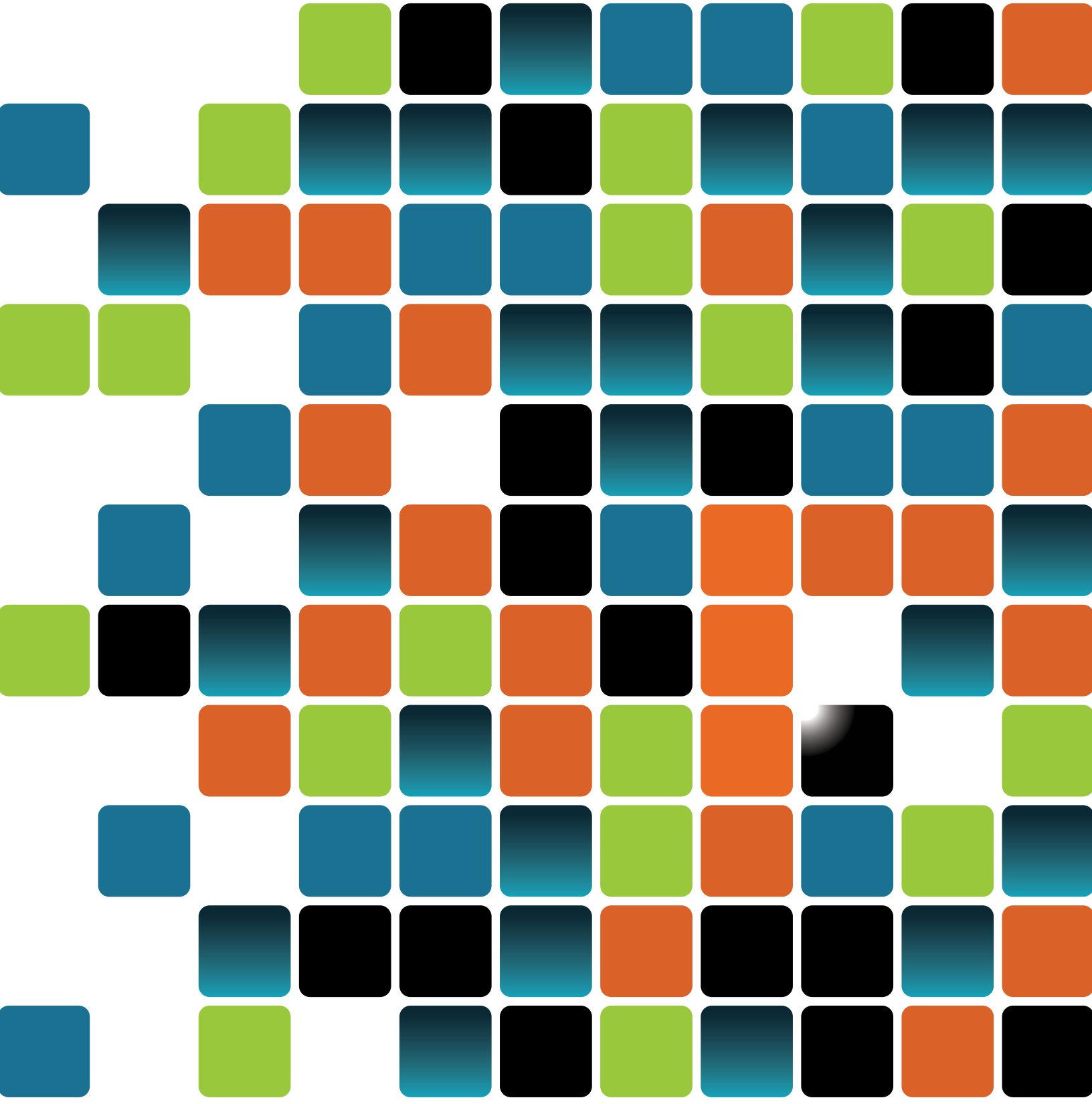
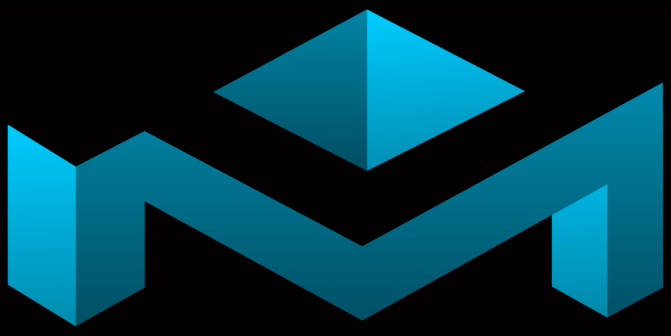
MASHED CODE MAGAZINE

Volume One, Issue One 2011





ProtectedMethod

Ruby. Rails. ProtectedMethod.

We Build Awesome Web Apps.

www.protectedmethod.com

Letter from the Curator

Nick Watts, Creator and Curator

Welcome to the first issue of *Mashed Code Magazine*. This magazine is a unique endeavor in the world of Software Development conferences. It is a freely distributed magazine that accompanies and complements the CodeMash conference. Every piece of content in *Mashed Code Magazine* is authored by people who are either speaking at or attending the conference. Our goal is to provide the same type of technical content that CodeMash offers but in a package that you can take home with you. CodeMash has the greatest community surrounding it of any development conference, so it only makes sense that *Mashed Code Magazine* has been designed and written by your peers in that community.

Not only is the creation of *Mashed Code Magazine* unique, its content is unique too. In fact, we hope that you find *Mashed Code Magazine* to be different from any other developer magazine that you have read – online or in print. There are a variety of topics covered in the magazine, from the history of CodeMash to machine learning to how to better use your brain. These topics – and this is the cool part – are then presented in a variety of ways. We have a mix of interviews, prose, technical articles including code, short "Lightning Articles" and visuals. Despite the variety in presentation, almost all of the content is technical and focused towards adding to the tremendous value that CodeMash already provides.

There are, of course, many people behind the creation of *Mashed Code Magazine*. The first thanks goes to the authors whom have given their time freely and openly. We are thankful that so many people were willing to take a risk on something new. If you enjoy an article, please make a point to say thanks to the author during the conference. We are no less grateful to the organizers of CodeMash – especially Jason Gilmore and Jim Holmes. The CodeMash organizers have been both accepting of introducing the magazine to the CodeMash community and accommodating in offering much needed help and advice throughout the publishing process. Equally as important has been the volunteer staff: Matt Darby, Jason Gilmore, Jason Gurik, Michael Letterle and Sara Smith. These volunteers have supported this project whole-heartedly and steadfastly well beyond what was expected of them.

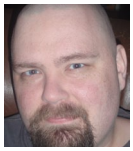
We hope that you enjoy the magazine and learn something from it. If you want to give us your opinion of the magazine, we've got a feedback form setup at http://www.mashedcodemagazine.com. Just click on the survey link to let us know how we did this year. To keep up with news about the magazine you can follow us on Twitter @mashedcodemag.

Enjoy the conference!

Nick Watts, Creator and Curator
Nick can be contacted on Twitter @thewonggei or via email at nick.a.watts@gmail.com.







Staff Page

Nick Watts - Creator and Curator

nick.a.watts@gmail.com • @thewonggei

Nick is a lifelong student of computer science, music, and literature. In pursuit of those interests, Nick works by day as a Senior Programmer/Analyst at Ohio Mutual Insurance Group, moonlights as the Technical Editor for *No Fluff Just Stuff*, *the Magazine*, reviews books for The Pragmatic Bookshelf and listens to a fascinating collection of music while performing all of his duties. Nick has specialized in working with web services and mastering various testing techniques and tools. Nick blogs frequently at http://thewonggei.wordpress.com.

Jason Gilmore - CodeMash Co-Founder

W. Jason Gilmore is the author of six books, including the bestselling “Beginning PHP and MySQL, Fourth Edition” (Apress, 2010), “Easy PHP Websites with the Zend Framework” (W.J. Gilmore, LLC, 2009), and “Easy PayPal with PHP” (W.J. Gilmore, LLC, 2009). He has been published more than 150 times within the likes of Developer.com, JSMag, and Linux Magazine. Over the years, Jason has instructed hundreds of developers in the United States and Europe.

Jason is co-founder of the popular CodeMash Conference (http://www.codemash.org), and was a member of the 2008 MySQL conference speaker selection board.

Sara Michelle Smith - Graphic Designer

Sara Smith is a Graphic Artist and also the Art Director at MW Defense Systems in Lumberton, NC. She designs 2 monthly magazines, GroovyMag and JSMag in her spare time and works on various other freelance projects for several companies in the Fayetteville area. She has an Associates Degree in Advertising & Graphic Design and is always eager for new opportunities to use her creativity. When she is not creating, she enjoys music, the arts, and collecting spiders.

Matt Darby, MS - Partner, Protected Method

matt@protectedmethod.com • @mattdarby • www.protectedmethod.com

Matt is a co-founding Partner at Protected Method and holds a Masters Degree in Computer Science along with 13+ years experience in web development and IT management. He is a frequent open source contributor and a Rails Core Contributor.

Matt loves developing comprehensive solutions that address anything from building servers to developing and maintaining complex web applications. Ruby and Ruby on Rails are his current tools of choice.

Jason R. Gurik - Editor

@JasonRGurik

Jason R. Gurik began his career as a Cobol programmer, eventually working his way into the more exciting and challenging object oriented languages. Today as a Senior Programmer/Analyst for the Ohio Mutual Insurance Group, Jason supports applications in both .Net and Java. Education and continued learning have always been important to Jason. He recently earned his Masters Degree in Business Administration and has aspirations to work his way into an IT leadership position in the future.

Michael Letterle - Bit Samurai

@mletterle • http://blog.prokrams.com/

A contributor to IronRuby, Michael Letterle is an avid technologist and passionate about programming. He was awarded the Microsoft MVP Award for his work on IronRuby. When not hacking on Ruby, he is a lead developer at PreEmptive Solutions, and gets paid to work on the Dotfuscator and Runtime Intelligence Platforms. He has been an active member of the .NET community for the past 5 years, speaking at regional conferences such as CodeMash and eRubyCon. Michael also likes to try and play the guitar and his Xbox when he’s not enjoying time with his wife and four year old daughter.

Table of Contents

Holmes on CodeMash

By Jim Holmes

I'm writing this article while sitting in my recliner just a few short days after CodeMash 2011 has sold out. We committee members are still a little numb because CodeMash 2011 sold out in a staggering 3.5 days – and that over a weekend!

The speed at which CodeMash 2011 sold out is a testament to many things, but mainly the great culture that's built up around the conference. That culture has come from a lot of hard work by the organizers, but is also directly attributable to the amazing attendees who return to the conference year after year.

Getting Started

CodeMash grew out of a dinner meeting between some of the Heartland community's most passionate members. Drew Robbins, Chris Judd, Dave Donaldson, Jason Gilmore, and Brian Prince all hailed from different domains in the community, and all of them were wondering how to get those different domains talking together in order to spread common problem solving concepts and get people thinking about new ways to create better software.

CodeMash Then

That dinner led to a small group of folks coming together to plan and execute the first CodeMash. The initial organizing committee included Drew Robbins, Jason Gilmore, Brian Prince, Jason Follas, Josh Holmes, Dianne Marsh, Darrell Hawley, and Jim Holmes (no relation to Josh). All the members of the committee had experience putting on local and regional developer events.

From the outset we knew we wanted CodeMash to be a quirky, fun conference; finding the right venue was really important to us. I don't remember who came up with the idea of looking to the Kalahari as a place to host the event, but it quickly became obvious that having several hundred pasty white geeks playing around at an indoor water park was a perfect fit with the funky tone we were looking to pull off.

We had a strong common vision for the conference: build a list of kickass sessions to attract attendees to get outside their comfort zone. The selection committee solicited submissions from a wide range of speakers in different domains and technologies. Talks on Java, Ruby, .NET, Python, and other platforms all came pouring in.
Planning of the first CodeMash, held in January 2006, proceeded with plenty of bumps and setbacks, but everyone's hard work paid off in spades. Somewhere around 250 attendees, speakers, and staff gathered at the Kalahari and took part in a small, but amazing conference. Attendees came away energized and motivated to go look at things in a different way.

(Tangentially, organizers Brian Prince and Josh Holmes both came away with no hair, having challenged the audience to write 500 blog posts about CodeMash during the conference.)

The success of that first conference was due in no small part to the attendees' mindset. The collaborative, not combative, environment was a critical part of the organizers' vision, and attendees totally got what we were trying to accomplish. Discussions were passionate and engaged, but always productive.

CodeMash Now

Fast forward to October, 2010. In five years we've seen tremendous changes in the size of the conference, but we've managed to keep the same open, collaborative, and passionate environment despite having grown to over triple the size of the first conference. We'd have grown still larger last year and this, but we've maxed out the venue – we can't fit any more geeks in the conference center!

Along the way we've picked up more attendees, more sessions, more families, the KidzMash sessions (targeted to the attendees' children), the CodeMash Families group (helping family members meet up), the PreCompiler workshops, and a host of other small things. The membership of the planning committee has changed a bit over the years, but we remain a tiny, tight group of folks focused on creating a great conference centered around world-class content.

The tremendous growth we've seen has been ***without a single dollar spent on advertizing***. Nada. Zilch. Nil. Everyone who's learned about CodeMash has learned about it by word of mouth. Blogs, Twitter, and good old conversations between developers passionate about learning drove our growth.

Why it's successful

What makes CodeMash so successful? Three things. First, to morph a phrase from the real estate world, "Content, content, content!" The content selection committee has the daunting task of wading through hundreds of submissions – over 500 for the 2011 event. We've been fortunate to have great submissions from influencers in the Heartland region to industry leaders. We've had visionary keynoters Neal Ford, Mary Poppendieck, and Venkat Subramaniam to name a few. Content is king. Period.

Secondly, we've had an amazing group of attendees. We have the cream of the crop attending this conference, and they're here because they totally get the central vision of CodeMash: folks from different domains engaging and sharing knowledge.

Finally, the playful nature of CodeMash is unique to any conference, anywhere. More and more attendees understand that not only *can* they bring their families, they ***should*** bring them. Attendees hang out together in the water park – we even had an open space session on networking for geeks that took place in the Lazy River. Playful geeks socialize in the Game Room with boards, charts, and dice. The CodeMash Jam Session fills a room every year with music and chatter. All these small metaphorical corners add up to a large batch of awesomesauce that slathers the conference in goodness.

CodeMash Next

What's in the future for CodeMash? The first three years we were agonizing, seriously agonizing, over whether we should cap the conference at 400, then 500. We've hit the physical limits of the venue for the next two years, although a rumored expansion may give us room for thousands. Does CodeMash keep its awesometastic nature if it gets that big? Other conferences have started out as something really special and lost their luster when they've grown too big. At the end of the day, our single, driving goal is to have a great conference for the attendees, not to grow to "X" number or "Y" dollars.

What will CodeMash look like in 2012 or 2013? No clue. CodeMash has a life of its own, and for that I'm incredibly fortunate. I work with an amazing team of folks who do all the hard work putting on the conference. I rub elbows with world-renowned speakers who put forth tremendous sessions that change peoples' lives. I'm surrounded by the best crowd of attendees whose energy and enthusiasm lift me up and keep me rolling year after year.

We'll figure it out as we go. At the end of the day I just write checks, run telephone conferences, and herd cats. To paraphrase Sam Malone's last line in *Cheers* "I'm the luckiest guy alive."

Code Retreat

A World-Wide Phenomenon that was Spawned at Codemash

By Corey Haines
Twitter: @coreyhaines, @coderetreat

Code Retreat focuses on getting back to the basics and studying the fundamentals of software development. It has happened in at least 10 different countries around the world, including places like Romania, Spain, Australia and Belgium. The goals of Code Retreat are simple: by retreating from the pressures of getting things done, we allow ourselves the freedom to practice writing perfect code. Who would have thought that such a powerful, world-wide phenomenon would have its roots at an indoor water park in Sandusky, Ohio. But, for those of us who have watched Codemash grow from a small regional conference in 2006 to a conference with a national draw in 2011, it isn't surprising; it was probably inevitable.

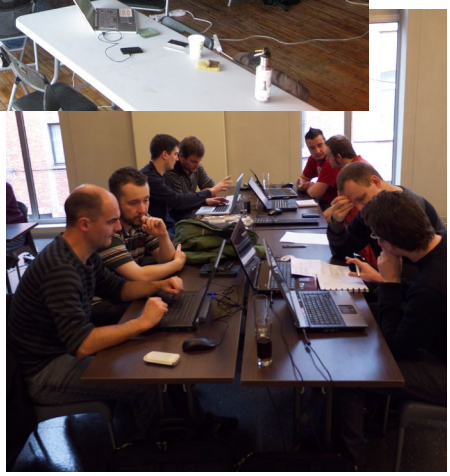
At 2009 Codemash, I was at the beginning of my pair-programming tour. I had returned from my first 3-week trip around Illinois, Indiana, Ohio and Michigan, programming with people in exchange for room and board. In December of 2008, I had attended the Software Craftsmanship Summit in Libertyville, IL, the event that eventually spawned the Software Craftsmanship Manifesto. One of the core ideas related to Software Craftsmanship is that of developer competence and accurate self-assessment. The idea of code katas and coding dojos had been around for a while, but we were looking for new complementary ideas, as well.

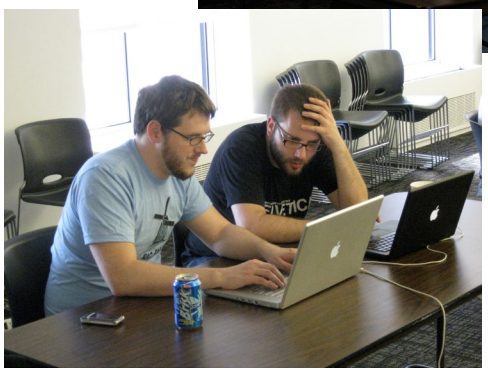
Gary Bernhardt, another long-time Codemash alumni, and I had chatted a bit about the idea of pair-programming retreats: longer activities that would focus on pair-programming as a technique for transferring knowledge and skills while working on an actual project. Around the same time, Patrick Wilson-Welsh and Nayan Hajratwala were contemplating the idea of a code chautauqua, a retreat to get away from the pressures of deadlines and to focus on skills. As commonly happens at Codemash, although our ideas had different focuses, they began to cross-pollinate. We started thinking about details, and decided the best way to figure it out was to organize something. We weren't sure exactly of the format, but the idea of spending a day focused on practice, retreating from the pressure of 'getting it done,' would be the goal. In the end, we decided upon the name 'Code Retreat.'

The first retreat was planned for January 24th, 2009, in Ann Arbor, Michigan. We just needed to figure out the format. In order to satisfy the need to 'retreat from getting things done,' it was important to have a good structure; we wanted to keep people from falling into the usual trap of cutting corners while pushing to the end goal. To achieve this, the idea was to have short sessions where the code would be deleted at the end. Patrick had an existing Java-applet-based version of Conway's Game of Life, so we figured we could use that problem. We didn't want it to be Java-specific, so we spent the morning doing Java and the afternoon doing Ruby. Tapping into our existing connections in the area, we were able to attract such people as Ron Jeffries, Chet Hendrickson and Bill Wake. We even convinced J.B. Rainsberger to fly in from Dauphin, Manitoba, to attend. With an experienced line-up like this, the day was bound to yield something worthwhile. Having a general idea of a problem and a format, we decided to let the actual specifics of the day evolve as we discovered new things during the activities.

After the first Code Retreat, 2009 saw more events happen. Slowly, the format evolved into what we have today, as outlined at http://www.coderetreat.com: 45-minute sessions, deleting the code, long lunch, emphasis on the basics of the 4 rules of simple design. While speaking at the Agile Eastern European conference in Bucharest, Romania in May of 2009, I had introduced the idea of Code Retreat to Alex Bolboaca and Maria Diaconu, who began to host their own retreats. While the general format through 2009 was to focus on a single language, the community in Romania began doing language-agnostic retreats, allowing each pair to choose what language to focus on. In February of 2010, as part of my 2010 Code Retreat tour, I had the opportunity to go back and see how the format had evolved there, bringing back this idea of language agnosticism into the retreats that I have run. Since then, Code Retreats have focused even more finely on practicing the basics, independent of the language that is used.

In May of 2010, we held our first weekend-long Code Retreat in Floyd, VA, sponsored by companies like





EntryWay Software and Engine Yard. This was taking the idea of retreat one step forward: 40+ people gathered in a small one-stoplight town in Virginia, staying from Friday night until Sunday afternoon. The location attracted two great minds of the Agile community: Mike Hill and George Dinwiddie. One of the great things about Code Retreat is the easy mixing of people at all different experience levels. On Saturday, we did a standard Code Retreat then retired to Apple Ridge Lodge for an evening of beer from the local brewery and brick-oven pizzas cooked on the spot by the local pizza company. The success of this event led to immediate planning for Code Retreat Floyd, 2011!

An important role during a Code Retreat is that of the facilitator. This is a leader who guides the teams in exploration of their design, pointing out possible improvements for each pair to explore. This guidance is often in the form of questions about a pair's code, pointing out duplication and naming issues. This helps the pair pull away from the depths of the code and to look at their current design as a reflection on their current skills. We get better by pushing ourselves past our general limits, trying things that may or may not work. After all, on average, in 22.5 minutes, the code will disappear. Alongside me, others have focused on fine-tuning their own skills in this role, people such as Gary Bernhardt, Enrique Comba and Alex Bolboaca. While anyone can facilitate, having the experience of many Code Retreats under your belt helps you adjust the approach to the teams at hand.

Nowadays, it is common to see a Code Retreat happening almost every weekend somewhere in the world. The Romanian community has thrived, and the community in Spain is stepping up to organize their own retreats on a regular basis. The future promises to be even more exciting. The frequency of regular retreats is increasing, and plans are underway for Code Retreat Floyd 2011. One of the most exciting plans for 2011 is International Day of Code Retreat. This is a global event with a goal of having 32 hours straight where a Code Retreat is happening somewhere in the world. Starting just to the west of the International Date Line, Code Retreats will begin to ripple through the time zones, starting at 8.30am. As a retreat ends in one time zone, the participants can teleconference to another event that is just starting. Pass-offs will interweave themselves around the globe, finally finishing just to the east of the Date Line. How many retreats will happen that day? It remains to be seen.

It is fitting that the initial Code Retreat idea was spawned at Codemash. This conference strives to bring people together from different siloed communities, rising above language differences and sharing what is common among all of us: the fundamentals of software development. Code Retreat focuses on a specific area of this transcendence by eliminating all the differences and focusing on the essentials: improving your skill by practicing the basics. And it all started at an indoor water park in a little town called Sandusky.

Interview with Ethan Dicks

By Jason Gilmore

Ethan Dicks is the founder of the Central Ohio Reprap and Makerbot User's Group. His CodeMash session, "3D Printing with Open Source on a Makerbot", introduces attendees to the power of 3D printing. During his session Ethan will offer several demonstrations using his Makerbot Cupcake CNC 3D printer, so be sure to swing by his session for an amazing glimpse into the future of product development.

CodeMash cofounder Jason Gilmore recently sat down with Ethan and talked about the rapid prototyping movement, self-replication, and the DIY movement.

WJG:
We are really looking forward to your RepRap session and demonstration at the upcoming CodeMash. For the folks who aren't familiar with this fascinating machine, could you introduce the RepRap and tell us how you became involved with the local RepRap community?

ED:
The RepRap project, short for Replicating Rapid-prototyper, is the brainchild of Dr. Adrian Bowyer at Bath University. The technology it uses, fused filament deposition, is not a new way to create solid objects, but being able to do so for much less than the cost of a new car is. An outgrowth of the RepRap project is Makerbot Industries. They started by selling ready-to-assemble circuit boards for RepRap machines and produced their own derivative printer, the Cupcake CNC, commonly just called a Makerbot. As Makerbots were starting to be sold, a friend of mine in Columbus approached me about 3D printing and that conversation turned into a group project to build a 3D printer of our own with plans to make a printer for each member of the group. We are still working towards that goal, but we have nearly all the parts for one RepRap built or purchased and have a substantial amount gathered for the second. From this core, I reached out last December to other RepRap and Makerbot owners in the area and started a user group. We've been meeting every month since at The Columbus Idea Foundry.

WJG:
The RepRap's level of sophistication continues to grow seemingly by the month. What is the most complex object you've seen created so far?

ED:
Out of all the complex and detailed items, I'd probably have to point to the printed Makerbot (http://www.thingiverse.com/thing:3285) as being the most complex. It's around 150 individual printed parts glued and bolted together to make a working 3D printer (the original relies on several square feet of laser-cut plywood for its structure).

WJG:
One of the primary goals of the RepRap project is to create a machine which can self-replicate. How close is the project to this goal being a reality?

ED:
The first RepRap machine to make a copy of itself was a "Darwin". The announcement of that in early 2008 was what got me started participating in the international community. The first Darwins were machined by hand or on CNC milling machines or printed on simpler wood-and-metal CNC extrusion machines nicknamed "RepStraps". With the rise of RepRaps, RepStraps and Makerbots, the current RepRap model, the Mendel, was designed from the ground up to use as many printed parts as possible. Not including nuts and bolts, over 50% of a Mendel is already printable by another Mendel, but there are some experimental designs that use fewer bearings and fewer metal fasteners and have a higher percentage of printed components.

WJG:
I understand you're involved with the Columbus Idea Foundry. These so-called Hackerspaces are really starting to gather steam across the country. Why do you think this is?

ED:
While one hacker working alone can produce amazing things, the potential of groups of hackers, each with

their own background and skills, pooling their talent and resources is more amazing. In the case of the Columbus Idea Foundry, many of the early members were well versed in welding and metalworking as well as carpentry and various other shop skills; electronics and programming wasn't as well represented. As the Idea Foundry has grown, we've picked up individuals both with new skills and skills already represented, making it easier for groups to form to tackle specific projects as their own free time permits.

WJG:
Other than the RepRap, what other interesting projects is the Columbus Idea Foundry working on these days?

ED:
The Idea Foundry has just moved to newer, larger quarters with room for individual artists to set up their own workspaces. The build out on the first batch is nearing completion. Tangential to that, one of the members has purchased a 60W Epilog laser cutter/engraver for his own work that the rest of us can book time on. Once that's fully checked-out and operational, I expect many amazing things to come from the combination of a laser cutter, a Makerbot, a Shopbot, and metal welding/casting resources all under the same roof.

WJG:
Is it possible for an entrepreneur to hire a Columbus Idea Foundry to build a product prototype? If so, what is the process for doing so?

ED:
The process would begin by contacting the head of the Columbus Idea Foundry, Alex Bandar <alex.bandar@yahoo.com> to arrange for a time to meet in person, tour the shop and discuss the details.

Refactoring Your Wetware:

Rewire Your Brain to Get More Great Ideas

By Andy Hunt

Portions of this article adapted from "*Pragmatic Thinking & Learning*", by Andy Hunt. Published by Pragmatic Bookshelf (www.PragProg.com) and reproduced here with permission.

Your brain is a funny thing. It's a horribly complicated, squishy bit of stuff that has a hard time understanding itself. In fact, the more we understand about the brain the more incredible it is that we understand anything at all.

But it's important that we do try to understand how the brain works—and specifically, for you to understand how your brain works. That's where software development actually occurs. Not in an IDE, not on a whiteboard, certainly not in a 500-page "design document." It all comes from the brain—your brain. So it makes some sense to take a break from learning new programming languages, libraries and tools, and take some time to understand how your brain works, and maybe even try some tweaks to rewire it. It's actually pretty easy to do.

In software development, we often advise against creating self-modifying code (robust languages such as Erlang don't even have modifiable variables). This prohibition is for your own good: self-modifying code can be unpredictable at best, and devilishly hard to debug and work with.

So it's a little disconcerting to realize that your brain, an incredible learning machine, is itself a self-modifying machine. It can be unpredictable at best, and devilishly hard to debug and work with.

Among other things, this means that the thoughts you think, how you think them, and what you believe about the brain's capacities, physically affects the "wiring" of the brain itself.

Now that might sound like wishful thinking laced with a liberal sprinkling of faerie dust, but in fact it's just an example of the importance of context: your thoughts are inextricably linked with your brain, and each affects the other. Your brain isn't an object, it's a system.

Context and Systems Thinking

In John Steinbeck's *The Sea of Cortez*, the author writes:

> *"The man with his pickled fish has set down one truth and has recorded in his experience many lies. The fish is not that color, that texture, that dead, nor does he smell that way."*

In other words, taking something out of context to further study it can be misleading. Our world is *tightly coupled*, to use the object-oriented notion. The context of one thing of interest overlaps with the next, such that everything is ultimately interconnected: the physical world, social systems, your innermost thoughts, the unrelenting logic of the computer—everything forms one immense, interconnected system of reality. Nothing exists in isolation; everything is part of the system and so part of a larger context.

Because of that inconvenient fact of reality, small things can have unexpectedly large effects. That disproportionate effect is the hallmark of nonlinear systems, which includes the real world and can also include the artificial worlds you create when you write software.

But back to your brain: small things that you think are so subtle or inconsequential that they couldn't *possibly* make a difference can actually make a huge difference. For instance, thinking a thought to yourself versus speaking it out loud, or writing a sentence on a piece of paper versus typing it into an editor on the computer. Abstractly, these things should be perfectly equivalent.

But they aren't.

These kinds of activities utilize very different pathways in the brain—pathways that are affected by your very thoughts and how you think them. Your thoughts are not disconnected from the rest of the brain machinery or your body, or your team or your company; it's all *connected*.

In his seminal book *The Fifth Discipline*, Peter Senge popularized the term *systems thinking* to describe a different approach of viewing the world. In systems thinking, one tries to envision an object as a connection point of several systems, rather than as a discrete object unto itself.

For instance, you might consider a tree to be a single, discrete object sitting on the visible ground. That's how you draw a tree in grade school.

But in fact, a tree is a connection of at least two major systems: the processing cycle of leaves and air, and the cycle of roots and earth. It's not static; it's not isolated. And even more interesting, you'll rarely be a simple observer of a system. More likely, you'll be part of it, whether you know it or not. Remember Heisenberg and his quantum uncertainty principle? In a more general sense the *observer effect* suggests that you can't observe a system without altering it. It's all connected.

So to recap: everything is connected to everything else, and your brain is a self-modifying machine. That leads to...

DIY Brain Surgery and Neuroplasticity

You can physically rewire your brain. Want more capability in some area? You can wire yourself that way. You can repurpose areas of the brain to perform different functions. You can dedicate more neurons and interconnections to specific skills. You can build the brain you want.

Before you get carried away, put away the Dremel tool and dental pick; there's an easier way to do brain surgery. No tools required.

Until recently, it was believed that brain capacity and internal "wiring" was fixed from birth. That is, certain localized areas of the brain were specialized to perform certain functions according to a fixed map. One patch of cortical real estate was devoted to processing visual input, another to taste, and so on. This also meant that the capacity for whatever abilities and intelligence you were born with were largely fixed and that no additional training or development would get you past some fixed maximum.

Fortunately for us and the rest of the race, it turns out that isn't true. But as cool as this idea of self-wiring brain is, the reason that the researchers got it wrong is almost more interesting.

In a discovery that turned the field on its ear, Dr. Elizabeth Gould discovered *neurogenesis*—the continued birth of new brain cells throughout adulthood. The reason researchers had never witnessed neurogenesis previously was because of the environment of their test subjects: they had taken the research subjects out of their natural habitat, and studied them in the lab. They quite literally took them out of context.

For decades, scientists were misled because an artificial environment (sterile lab cages) created artificial data. Once again, context is key. What they really observed was that if you're a lab animal stuck in a cage, you will never grow new neurons (if you're a programmer stuck in a drab cubicle, you probably will never grow new neurons either.)

On the other hand, in a rich environment with things to learn, observe, and interact with, you will grow plenty of new neurons and new connections between them.

So it turns out that the human brain is wonderfully plastic. In fact, it's so plastic and adaptable that researchers have been able to teach a blind man to see with his tongue. [1] They took a video camera chip and wired its output to the patient's tongue in a small 16x16-pixel arrangement. His brain circuits rearranged themselves to perform visual processing based on the neural input from his tongue, and the patient was able to see well enough to drive around cones in a parking lot! Also notice that the input device isn't particularly high resolution: a mere 256 pixels. But the brain fills in enough details that even this sort of low-res input is enough.

Neuroplasticity (the plastic nature of the brain) also means that the maximum amount you can learn, or the number of skills you attain, is not fixed. There is no upper limit—as long as you believe that is true.

According to Stanford University research psychologist Carol Dweck[2] students who believed they could not increase their intelligence in fact couldn't. Those who believed in the plasticity of their brains increased their

abilities easily. In other words, thinking makes it so. What you think about the brain's capacities physically affects the "wiring" of the brain itself. That's a pretty profound observation. Just *thinking* that your brain has more capacity for learning increases your brain's capacity for learning.

Cortical Competition

And it's not just your beliefs that can rewire your brain; there is always an ongoing competition for cortical real estate in your brain based on your ongoing experiences.

Hebb's rule (named after Dr. Donald Hebb), tells us that "Neurons that fire together wire together. Neurons that fire apart wire apart." That is, every memory you repeatedly access, and every experience you practice, strengthens the connections between those sets of neurons.

Skills and abilities that you constantly use and constantly practice will begin to dominate, and more of your brain will become wired for those purposes.

At the same time, lesser-used skills will lose ground. "Use it or lose it" is the rule in this case, because your brain will dedicate more resources to whatever you are doing the most.

Perhaps this is why musicians practice scales incessantly; it's sort of like refreshing dynamic RAM. Want to be a better coder? Practice coding more. Engage in deliberate, focused practice. Want to learn a foreign language? Immerse yourself in it. Speak it all the time. Think in it. Your brain will soon catch on and adapt itself to better facilitate this new usage.

Want more great ideas? Better solutions to thorny debugging or architectural problems? You can use this cortical-rewiring to help generate and capture more great ideas and flashes of insight.

Generating Great Ideas

The trick is to first capture *all* ideas, in order to get more of them.

Once you start keeping track of ideas, *you'll* get more of them. Your brain will begin to reorganize itself to better support this activity.

Everyone–no matter their education, economic status, day job, or age–has good ideas. But out of this large number of people with good ideas, far fewer bother to keep track of them. Of those, even fewer ever bother to act on those ideas. Fewer still then have the resources to make a good idea a success. You have to start by at least keeping track of good ideas, and once you start, you'll get more.

But answers and insights pop up independently of your conscious activities, and not always at a convenient time. You may well get that million-dollar idea when you are nowhere near your computer.

In fact, you're probably much more likely to get that great idea precisely *because* you are away from the computer. Typing on the keyboard tends to activate mental processing modes that block creativity and invention, not encourage it. You may have experienced this phenomenon yourself. Have you ever been stuck on a hard problem, and giving up for a moment, walked away to go to the restroom, or out to lunch, or home for the day, only to have the solution pop into your head as you're halfway down the hall? That's your brain's subtle mechanisms at work, showing you your flash of insight when you least expect it.

That means you need to be ready to capture any insight or idea twenty-four hours a day, seven days a week, no matter what else you might be involved in. Here are a couple of methods you might find useful:

- **Pen and notepad** I carry around a Fisher Space Pen and small notepad. The pen is great; it's the kind that can write even upside down in a boiling toilet, should that need arise. Folks also recommend the Zebra T3 series; see http://www.jetpens.com for both a pen and Mechanical pencil version. You can get light-up pens with LEDs for thoughts from the middle of the night. The notepad is a cheap 69-cent affair from the grocery store—skinny, not spiral bound, like an oversize book of matches. I can carry these with me almost everywhere.

- **Index cards** Some folks prefer having separate cards to make notes on. That way you can more easily toss out the dead ends and stick the very important ones on your desk blotter, corkboard, refrigerator, and so on.

- **Mobile Devices** You can use your Apple iThingy (Phone/Pod/Pad/Touch) or Palm OS or Pocket PC device along with note-taking software or a simple Wiki. I really enjoy having ubiquitous access to my personal Wiki, using something like DropBox (http://www.dropbox.com) for cloud storage.

- **Voice memos** You can use your cell phone or iThingy to record voice memos. This technique is especially handy if you have a long commute, where it might be awkward to try to take notes while driving. Some voicemail services now offer voice-to-text (called *visual voicemail)*, which can be emailed to you along with the audio file of your message. This leads to a really cool workflow: just call your voicemail hands-free from wherever you are, leave yourself a message. When you get back to your computer, just copy and paste the text from your email into your to-do list, your source code, your blog, or whatever. Pretty slick.

- **Pocket Mod** The free Flash application available at http://www.pocketmod.com cleverly prints a small booklet using a regular, single-sided piece of paper. You can select ruled pages, tables, to-do lists, music staves, and all sorts of other templates. A sheet of paper and one of those stubby pencils from miniature golf, and you've got yourself a dirt cheap, disposable PDA.

- **Notebook** For larger thoughts and wanderings, I carry a Moleskine notebook. (http://www.moleskine.com). These come in a variety of sizes and styles, ruled or not, thicker or thinner paper. There's a certain mystique to these notebooks, which have been favored by well-known artists and writers for more than 200 years, including van Gogh, Picasso, Ernest Hemingway, and even your humble author. The makers of Moleskine call it a "reservoir of ideas and feelings, a battery that stores discoveries and perceptions, and whose energy can be tapped over time". I find that there's something about the heavyweight, cream-colored, unlined pages that invites invention. Because it feels more permanent than the cheap disposable notepad, I noticed a tendency to not write in it until a thought had gelled for a while, so I wouldn't fill it up prematurely. That's bad, so I started making sure I always had a backup Moleskine at the ready. That made a big difference, and now I'm more likely to jot down my mental wonderings.

The important part is to use something that you *always* have with you.

Whether it's paper, a cell phone, an MP3 player, or a PDA doesn't matter, as long as you always have it.

Use one of these simple mechanisms consistently, and you'll begin to rewire your brain to become more creative and inventive. And since everything is a mass of inter-connected systems, it will begin to affect other things around you as well: you'll find yourself writing better code and designing better systems.

Andy Hunt is a programmer turned consultant, author and publisher. He authored the best-selling book "The Pragmatic Programmer" and six others, was one of the 17 founders of the Agile Alliance, and co-founded the Pragmatic Bookshelf, publishing award-winning and critically acclaimed books for software developers.

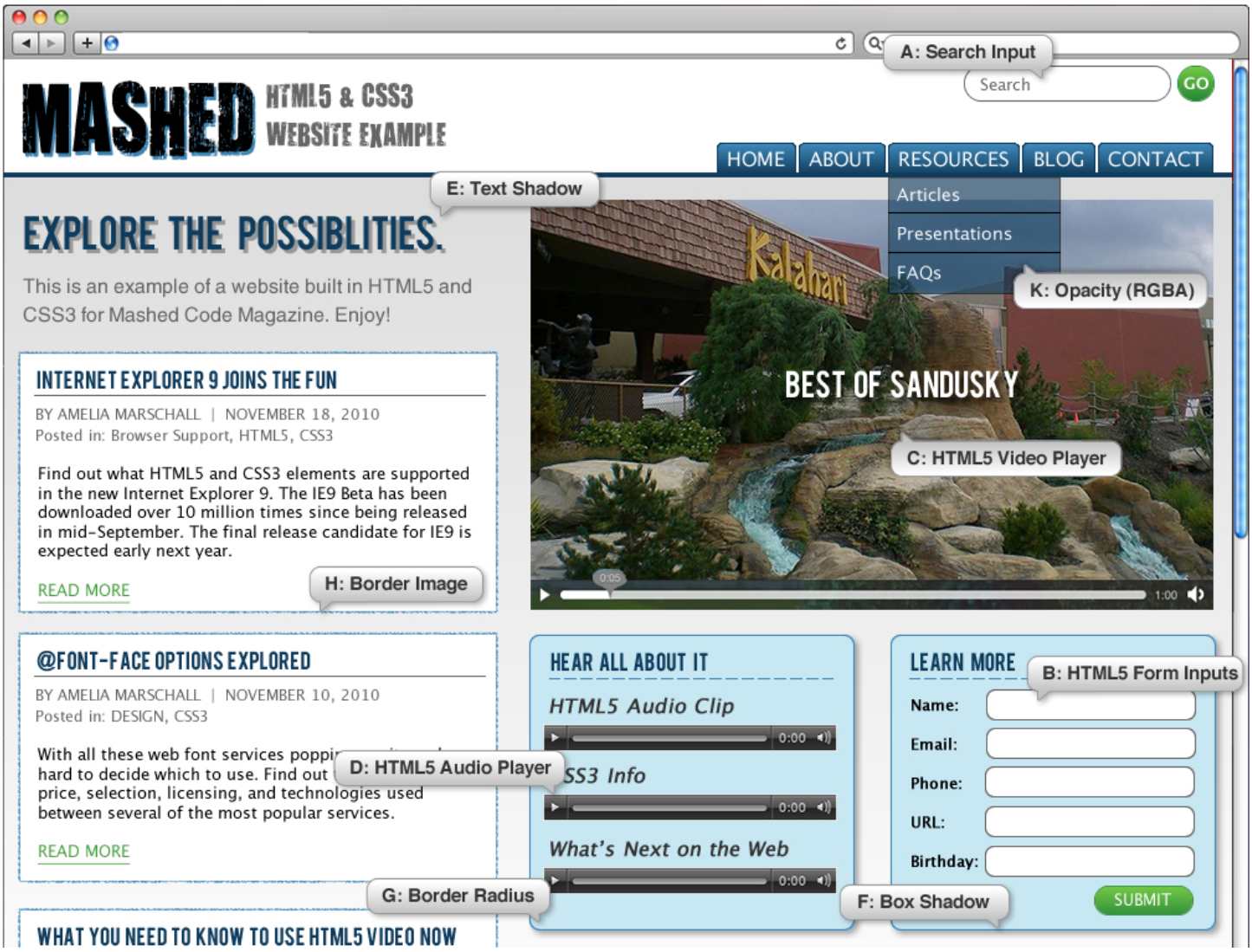
References:

[1] Doidge, Norman. *The Brain That Changes Itself*. Viking, New York, 2007.

[2] Dweck, Carol S. *Mindset: The New Psychology of Success*. Ballantine Books, New York, 2008.

Using HTML5 and CSS3 Today

By Amelia Marschall
amelia@gravityworksdesign.com
@MimiAmelia



HTML5 and CSS3 are the latest buzzwords in the web development world, and it's really quite easy to start using their elements in your websites today. All you need to use HTML5 is the new, compact Doctype: `<!DOCTYPE html>`. *In CSS3, many of the new styles can enhance your designs without decreasing functionality in older browsers. Let your website degrade gracefully, and use browser fall backs when necessary. Most of all, experiment and enjoy these simpler techniques for styling your website.*

Amelia Marschall is a Partner and Creative Director at Gravity Works Design & Development in Lansing, Michigan. She holds a Bachelors degree from Northern Michigan University in Graphic Design and Marketing and has over five years of experience ranging from website design, logos, branding, and print media. She is continually exploring the latest techniques in website design, and has spoken about the topic in the local development community. When she is not designing or building websites, she enjoys swimming, skiing, pottery, and competing in triathlons.

A: Search Input

Instead of a regular text input, search boxes can now be marked up as `type="search"`. Browsers that recognize this type will add search specific styling and features, such as an "x" button to clear the content in Safari. Browsers that don't support the search type will still render the field as `type="text"`. So you can use `type="search"` today.

```
<input type="search" placeholder="Search">
```

B: HTML5 Form Inputs

There are 13 new input types in HTML5. Browser support is not yet widespread, but if a type isn't recognized, it will still be rendered as a text box. Supporting browsers can do special things with the types, such as the modified keyboard on the iPhone when entering an email or URL, or a native date picker. Browsers are also starting to add automatic client-side validation for HTML5 input types.

```
<input type="email">
<input type="tel">
```

```
<input type="url">
<input type="date">
```

C: HTML5 Video Player

The `<video>` element now allows you to play a video directly on a webpage, without any plug-in. All of the latest browser versions provide native video players. But, they can't all decide on a codec. You can link to source video files, but you must include both OGG and MP4 (H.264) file formats. You can also include a fall back player for better browser support. Available attributes include: `width, height, autoplay, loop, controls, preload`.

```
<video width="640" height="360" controls>
   <source src="myvideo.mp4" type="video/mp4" />
   <source src="__VIDEO__.OGV" type="video/ogg" />
   <!-- include flash fall back here -->
</video>
```

D: HTML5 Audio Player

The latest browsers also provide native audio players. Again, you must include two of the three possible codecs for all browser support: OGG, MP3, and WAV. You can also include a fall back player for older browser versions.

```
<audio controls preload="auto" autobuffer>
  <source src="elvis.mp3" />
  <source src="elvis.ogg" />
  <!-- include flash fall back here -->
</audio>
```

E: Text Shadow

For the `text-shadow` CSS element, enter the properties in order of x-offset, y-offset, blur amount, and color. You can add multiple shadows by comma separating multiple lists of these properties within the `text-shadow` element.

```
.shadow1 { text-shadow: 5px 5px 0px #999; }
```

F: Box Shadow

Just like text-shadow, enter the properties for the `box-shadow` element in order from x-offset, y-offset, blur, and color. Multiple shadows are an option, as well as an internal box shadow by adding `inset` to the property list.

```
.box1 { box-shadow: 4px 4px 5px #aaa; }
```

G: Border Radius

Finally - rounded corners without images! And it works with borders and background colors. You may also target a single corner with `border-radius-topright`, etc.

```
.box1 { border-radius: 5px; }
```

H: Border Image

`Border-image` repeats and resizes a single image to create a seamless border. Define the image source, pixel amount to crop each corner, and whether the middle sections should be `repeat, round` (repeated, ending evenly), or `stretch` (the middle slice is stretched to fill the space.) `Border-width` should equal the corner sizes.

```
.borderbox { border: solid transparent; border-
width: 20px; border-image: url(line-edge.png) 20 20
20 20 repeat repeat; }
```

I: Opacity (RGBA)

There are two options here: `Opacity` will apply transparency to the selected element, as well as any other elements within it. Which means text inside the defined element might be opaque when you don't want it to be. `Background: RGBA` adds an alpha amount to a color definition. So only the defined color will be transparent.

```
.opaque1 { background:#000; opacity:0.5; }   OR
.opaque2 { background: rgba(0,0,0, 0.5; }
```

Note: Some browsers require a specific prefix on new CSS elements (-moz, -webkit, -o, -ms). For a good summary of HTML5 and CSS3 browser support, visit www.caniuse.com.

A Brief Introduction to Machine Learning

By Seth Juárez, DevExpress

Rutherford D. Roger once quipped: "We are drowning in information and starving for knowledge." This apt description of our current information conundrum has indeed stifled many a business model, seized decision-making, and shrunk many a bottom line. It turns out that the knowledge needed to ensure a robust business model, a rapid decision pace, and a raging bottom line already lies hidden in the information we store in our systems on a daily basis. The tools needed for this type of complex analysis has often come under the "Business Intelligence" umbrella that quite often fall short because of the inability of its users to ask the right questions. Although it is indeed useful to use OLAP, and surely reporting is an important, if not essential, tool in making business decisions, data mining has taken the proverbial "back-seat" on the road to business success. In order to get our information to drive our success, we simply need to learn to ask the right questions. Machine Learning is a field that has systematized the types of questions that can be asked of our information. Additionally, Machine Learning as a field has taken the additional step of creating compact models that provide great insight into the true meaning of the information we have painstakingly gathered as organizations. This introduction to Machine Learning will cover a brief overview of its two general fields, walk through a clustering algorithm, and finally show the practical implications of said algorithm in a simple environment.

Machine Learning

When software engineers are tasked with problems, our general approach is to come up with a series of steps involved in completing the task effectively and in a timely manner. Often, after formulating such algorithms, we go through the effort of refining, refactoring, and reassessing our work only to realize that there are additional edge cases for which we must account. There are certain classes of tasks that seem to have an infinite amount of edge cases. Take, for example, the task of figuring out whether a particular loan applicant will become delinquent. This task becomes extremely difficult given the amount of information that exists on loan applicants. A simpler example would be the task of grouping loan applications. How should they be organized? While grouping them by delinquency might seem the most plausible approach, we might miss the fact that some who are not delinquent are actually more closely aligned to those who are delinquent. This insight might help us tackle future problems as well. These two examples illustrate the two general fields in the Machine Learning discipline:

1. Can I predict the future? (Supervised Learning)

2. How is my information organized? (Unsupervised Learning)

Instead of producing algorithms to solve these problems, we turn software engineering upside-down for a moment and change the "human writing code" portion of the task to simply "human supplying data," and then allow the machine to come up with the appropriate algorithm (of sorts). While this seems like an implausible thing to do, the fields of mathematics, probability, statistics, algorithmic complexity, and information theory have converged in such a way so as to make this not only a possibility but a practicality in many large organizations.

Consider the following examples of successes that have used these approaches:

1. Amazon suggests appropriate products to purchase in the future based on previous purchases

2. Netflix compiles a list of movies and shows to watch in the future based on previous viewing patterns

3. Twitter suggests people who you should follow

4. Chemistry.com has a personality test to assist in finding that perfect date

These kinds of applications of machine learning still answer

the two basic questions listed previously but have been adapted to different domains quite successfully.

Supervised Learning

Supervised learning deals primarily with the prediction of future examples based on previous ones. Let's develop the loan application example a little further. Consider the following table of delinquency information as well as the information gathered during their application process:

Delinquent	Funds	Salary	Debts
Yes	$5,000	$110,000	$15,000
No	$10,000	$100,000	$1,200
Yes	$2,000	$65,000	$12,000
No	$6,000	$40,000	$8,000
?	$8,000	$75,000	$3,000

Loan Application Data, Table 1

Given the last example, what can we expect of their future delinquency? While a thorough use of the process of deduction might serve us well with only four examples, this mechanism breaks down when considering even 100 examples. In this case, a different approach is required.

In this setting, each row is called an *Example* (**x**). The columns (Funds, Salary, and Debts) are *Features* of each example. The Delinquent column is called a *Label* (**y**). A labeled dataset is a collection of (**x**, **y**) pairs. Given this dataset, the task of supervised learning is to construct a model for which given a new **x**, we can predict **y**. In other words, can the system correctly Classify new examples given previous examples. In this illustration, the range of answers the Label could take was either {**Yes**, **No**}. The number of answers that could be assigned to the target label dictates the classification type. Binary classification deals with a target label that takes only two values. When learning a label that can take only a finite set of values, the task is called Multi-Class Classification. Finally, when the target label can take an infinite set of values (like integers or doubles) the task is called Regression (See Figure 1).

What values can a target label take?

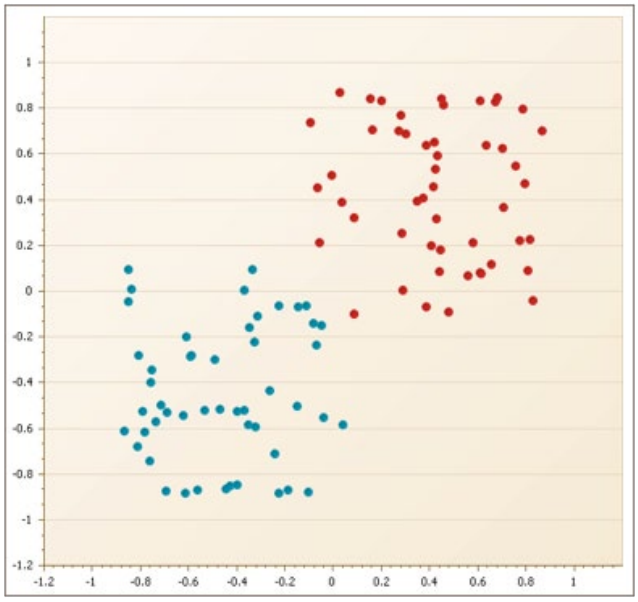
1. **Binary Classification**:
   {Yes, No}, {0, 1}, {Red, Blue}
2. **Multi-Class Classification**:
   {A, B, C, D, F}, {1, 2, 3}, {Red, Blue, Green}
3. **Regression**:
   Integers, Doubles, Decimals

Supervised Learning Types, Figure 1

In the case of classification, visualizing the task at hand is not too difficult if we think about the notion of class separability.

Binary Classification

Figure 2 describes a simple case of a two featured (x and y axis) labeled ({red, blue}) dataset and the corresponding orientation in a graph.

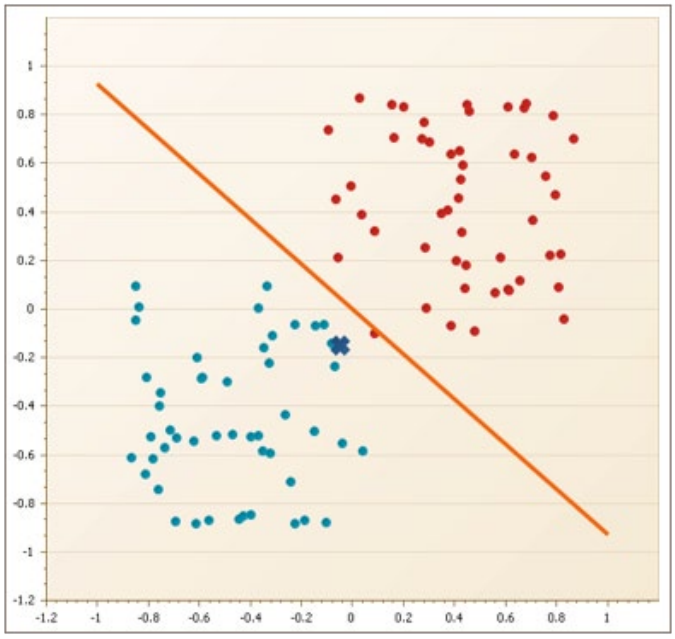


Two Featured Labeled Dataset in Euclidian Space, Figure 2

Given a new point in this space, how hard would it be to "eye-ball" the appropriate label? Intuitively we can visually detect the interaction of each of the two classes.

The first and easiest thing to do is simply draw a line through the data that best separates the classes (shown in Figure 3). When a new point is introduced, return the appropriate color based on the orientation of said point. This is called the *decision boundary*. The simplest algorithm that achieves this goal is called the Perceptron Algorithm. This algorithm simply initializes a line, then iterates through each point, and moves

the line in order for each point to be correctly situated with respect to the same.



Linear Separator, Figure 3

Another approach is to see how "close" it is to the other points. This simple approach is called kNN – or K Nearest Neighbors. In other words, pick the k closest points and have them each vote on the correct class for the introduced point. If we were to map out the class boundaries using this approach, notice that there would be certain points in this space where picking k neighbors would result in a tie. If the point were moved slightly in either direction, then a definite answer would arise. The point at which the tie exists is, again, called the decision boundary. This separator, unlike the one described earlier, would be non-linear and would offer greater decision power but at the cost of higher complexity. This interaction between model complexity and predictive power is indeed a balancing act. The linear model described only stores weights for each feature in the data representation. The kNN strategy, however, needs to store all the examples in order to produce the strongest prediction.

Multi-Class Classification

Using a simple binary classification model, one could envisage an approach to bootstrapping the simpler classifier to handle multiple classes. Notice that the constructed binary classifiers have the ability to separate two classes. With multi-class classification, multiple binary classifiers could be set up to vote on the appropriate class using one of two approaches: one-vs-all or all-vs-all. Let's change the earlier example of loan applications to have three classes instead of two. In other words, the label can take any of the following values: {Delinquent, Unknown, Not Delinquent}. In the one-vs-all approach, we set up three binary classifiers:

1. Delinquent or something else {Unknown, Not Delinquent}

2. Unknown or something else {Delinquent, Not Delinquent}

3. Not Delinquent or something else {Delinquent, Unknown}

Using these three classifiers, we find the one with the best result and choose the corresponding class. The all-vs-all approach would create a classifier for each pairing and return the best result. Notice that the cardinality (or size) of values that the label can take greatly increase the number of classifiers needed in the all-vs-all approach. For k classes, one would need $(k¦2) = k!/(2\ (k-2)!)$ different classifiers to do the job whereas in the one-vs-all approach we would need only k.

Regression

When trying to learn a label that can take a value from an infinite set, the best approach is to generate a curve that best represents the points. A well know regression type is linear regression. This approach draws the best line through the given points. The term best in this case is the line that minimizes the difference between the actual values of the label at a given point and the predicted value. While this indeed is the best mathematical approach to fitting a line to data, linear regressions fall short when data is non-linear.

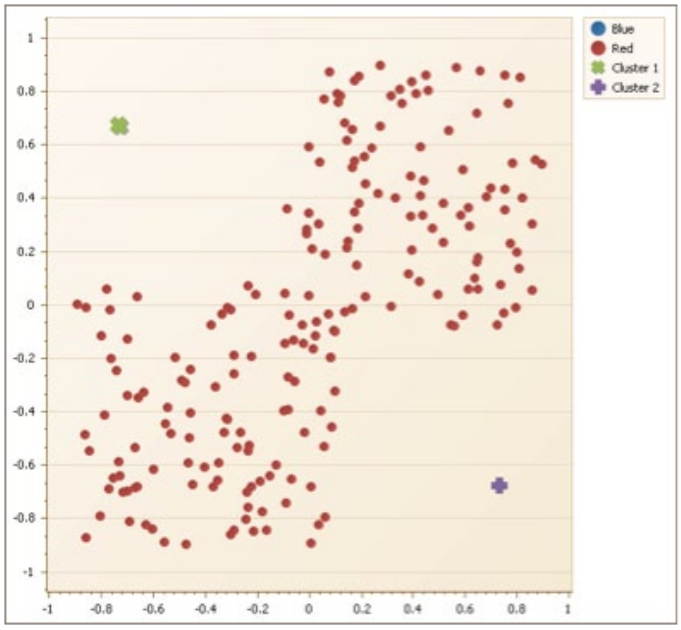
Unsupervised Learning

Unsupervised learning deals primarily with learning the shape of underlying information. In this particular setting, there are no labels provided as the goal of the learning task is unclear. The intent of this approach is to learn the grouping, ontological structure, or distribution of underlying data. In short, the goal is to learn the shape of data under investigation.

Grouping

The first and simplest question regarding the shape of data is how it can be grouped. Consider the task of placing employees on different floors of a building. The human resources department thinks it prudent to divide the employees by how similar they are to each other in order to

minimize conflict. Notice that in this example, we are asking the computer to learn a specific grouping given a set of unlabeled examples. The key to performing this task is by assigning a similarity measure between employees in order to ascertain the best grouping. In essence, if Sally is similar to Jim and not similar to Edith, then Sally and Jim should be on the same floor in the absence of Edith. A simple algorithm to perform this task is called K-Means.

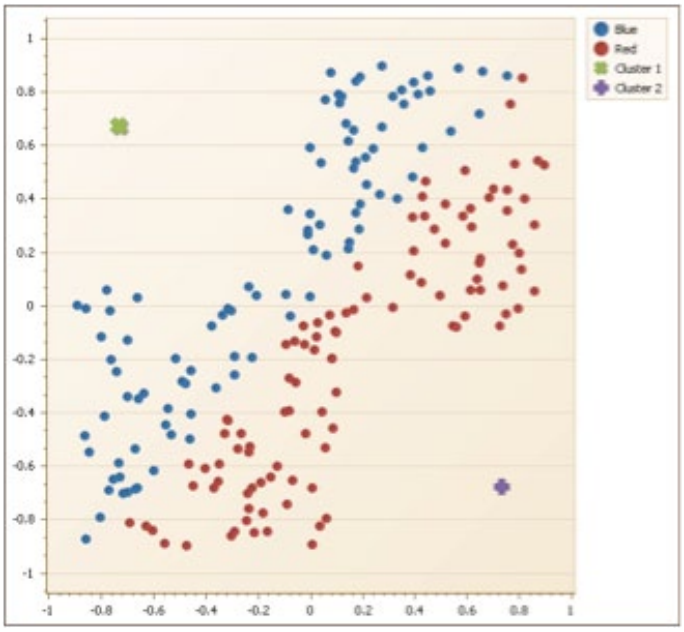


Clustering Data, Figure 4

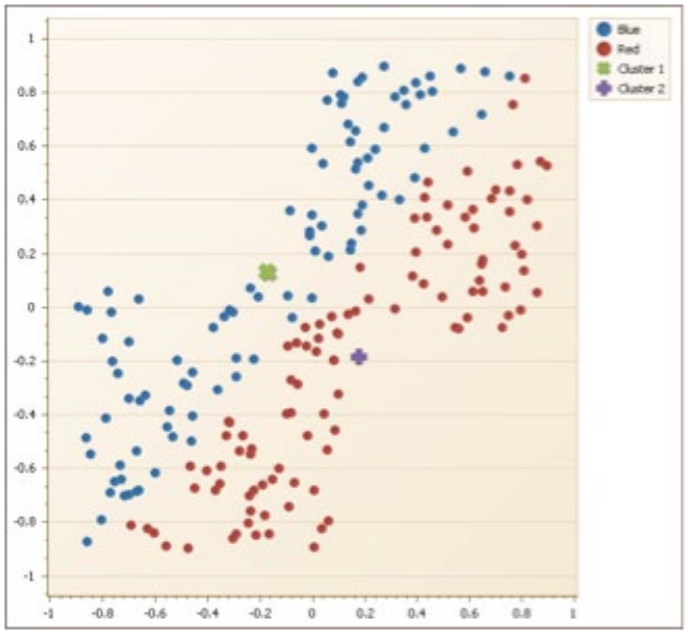
Consider data as arranged in Figure 4. The goal of K-Means is to create a grouping based upon relative distances between points. In this algorithm K represents the number of desired groups. In our case, we will fix K to be 2. The first step is to pick two centers. Several strategies have been introduced to pick initial centers, since bad center initialization can lead to a longer run time. Once the centers have been chosen, the algorithm repeats two steps until convergence:

1. Assign each point to the closest center

2. Update the centers to the average of all assigned points



Assignment Step, Figure 5



Update Step, Figure 6

Figures 5 and 6 show the corresponding steps in one iteration. An apt analogy for this process is two indentations in the ground and two marbles. The ground indentations in this case represent the relative mass of the data. In other words, imagine pulling the real data centers from behind into a bowl. In our case there would be two. Now start two marbles (as shown in Figure 5) and let them go. As time progresses, the marbles will eventually reach equilibrium at the appropriate centers in these bowl-shaped ground indentations.

Ontology

In computer science and information science, an ontology is a formal representation of knowledge as a set of concepts within a domain, and the relationships between those concepts[1] . Is there then an optimal arrangement of data given its individual relationships? Consider the task of arranging the following things:

{ Car, Fish, Dog, Canoe, Bike, Bear, Clam, Raft }

At a glance, this particular group of things might be hierarchically arranged as described in Figure 7. At the root level, the system has a node for all things in the group. The second level could be perceived as vehicles versus animals. The third level for the vehicles might be land vehicles versus water vehicles. The third level for animals could be land and water animals. While this particular hierarchy of things could be thought of as grouping, the flavor is still a bit distinct. In this case the explicit division into a predefined number of groups is absent. This particular task is called hierarchical clustering. Hierarchical clustering can be implemented in an agglomerative or agglutinative fashion. Agglomerative, or bottom-up, clustering starts with each element in a group of its own and grows each group based on similarity. Agglutinative, or top-down, starts with a single group and recursively break the group down into sub-groups based again on similarity. In both cases, each grouping becomes a single item by combining its elements into a representation that can subsequently be compared to other items. In the example above, {Car, Bike} were combined into the "Land Vehicle" element and {Canoe, Raft} were combined into the "Water Vehicle" element. When combining {Land Vehicle, Water Vehicle} we simply get Vehicle. The same kind of divisions can again be considered on the animal side of the tree. The manner in which elements are combined and then compared is called linkage. If these were points in the Euclidean space, we could measure the distance between groupings of points in one of three ways[2]:

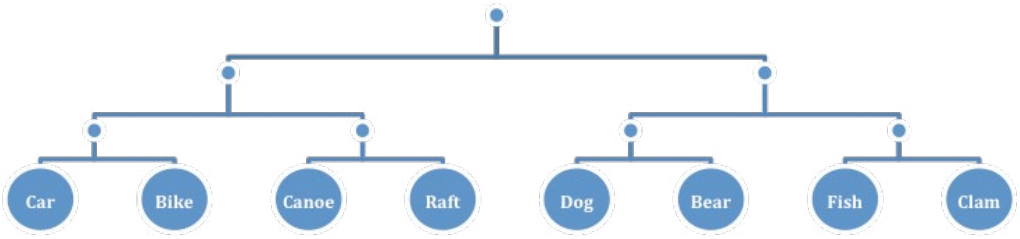
1. Take average distance between all points in each group (Mean or Average Link)

2. Take the minimum distance between all points in each group (Minimum or Single Linkage)

3. Take the maximum distance between all points in each group (Maximum or Complete Linkage)

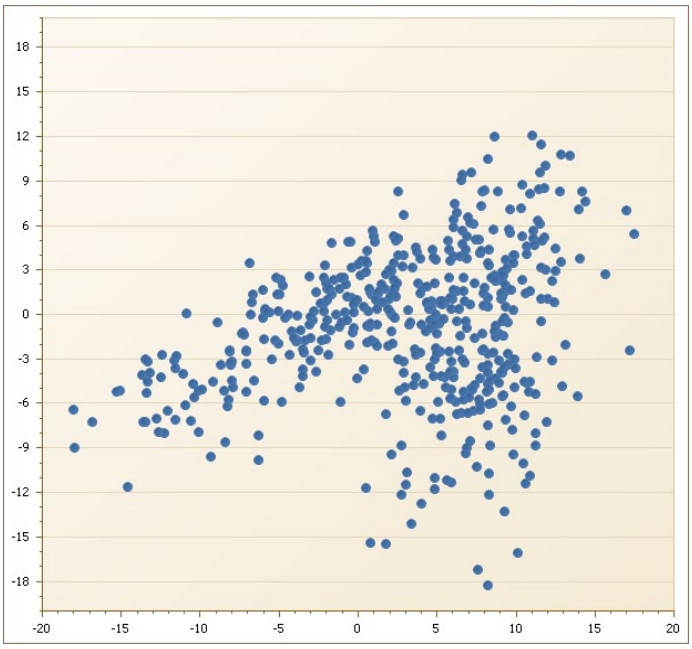
Statistics

Consider the arrangement of points in Figure 8. Let's assume that each point is the location of a bee at a certain time[3] . Although the positions of each bee indeed seem to be random, the swarm of bees itself seem to have some inherent pattern.

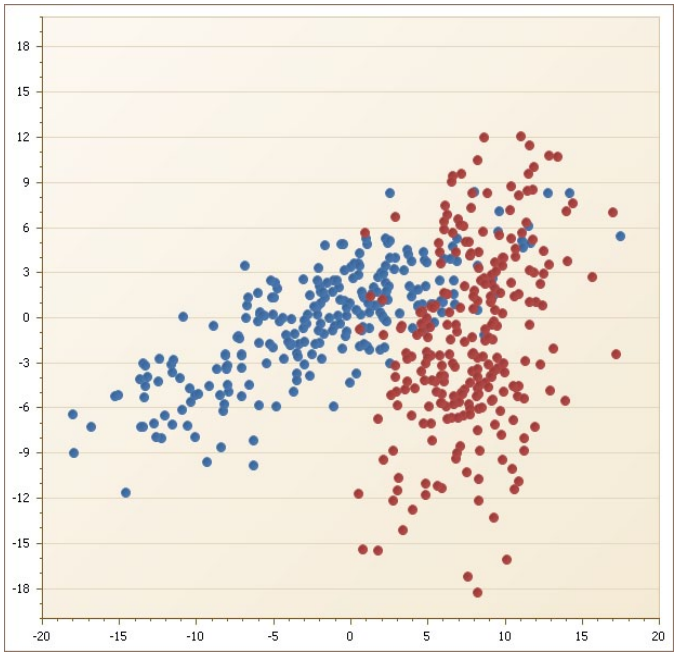


Hierarchical Data Structure

Figure 7



Randomly Generated Points, Figure 8



Randomly Generated Points by Distribution, Figure 9

It seems as though the bees are swarming in a V-typed shape with each cluster of bees centered at (-1, 0.5) and (8, 1) and tapering off in each direction. While this might seem like a truly improbable guess, these points were modeled using a mixture of normally distributed points as shown in Figure 9.

This particular model infers the underlying statistical representation of the data generated according to Gaussian[4] distributions. In this case we have a model that contains a mixture of two multivariate Gaussians centered at (8, 1) and (-1, 0.5), as discussed previously. This model is called a Gaussian Mixture Model (or GMM) and is extremely useful for finding the general statistics of the data in question. Continuing with the example, this swarm of bees might be used to detect hive locations. Although our data did not itself have this information, the statistical location of each bee in the swarm of bees could indicate the location of their respective hives. Consider a random sample of lake water that shows toxicity levels. A Gaussian Mixture Model could also be used to find the statistical distribution of toxicity levels in order to find the origin of pollutants.

The general algorithm for inferring this model is similar to the assignment-update step described for the K-Means algorithm. It is in fact an instance of a set of algorithms which uses the Expectation Maximization (EM) approach. As previously discussed, the K-Means algorithm chose initial cluster centers (the model parameters) and then maximized the expected cluster membership based on the distance of every point with respect to the centers (expectation step). Once membership was tabulated, the centers were then recalculated to maximize appropriate membership (maximization step). In the GMM algorithm, the EM approach calculates the center as well as a covariance[5] matrix. In this case the probabilistic model is fixed and the parameters are re-estimated until they reach equilibrium.

Representing Data

The aforementioned approaches rely heavily on a mathematical representation of seemingly un-mathematical things. The examples discussed included loan applications, employees, bees, etc. For a moment, let's discuss the mathematical representation of points along with a standard method for computing similarity and distance.

Vectors

A standard approach to looking at points is through the use of charts. Since our first brushes with math, we have been trained to plot (x, y) points in a graph and then subsequently forced to interpret the meaning of such data. This two-dimensional space becomes more interesting when adding a third dimension. What happens when more dimensions are added? Mathematicians invented a compact representation of n-dimensional space which allows practitioners to think in terms of a single dimension while working in many: this concept is called a Vector. Vectors are simply arrays of numbers with a predefined length. This length corresponds to the dimensionality of the space. In the case of Machine Learning, our dimensionality corresponds to the number of features in our examples. This is not too foreign a thought when considering Employees for example. Employees have many dimensions to them: Name, Age, Salary, etc. Vectors can also be thought of as points in space. Lastly, vectors can also be thought of as directions. Consider the vector (1, 1). When placing the point on a chart, one can draw a line from (0, 0) over to (1, 1). The general direction of this line is 45° from zero stopping at the point (1, 1). The key to Machine Learning is the conversion from that three-dimensional space into an actual numerical vector. Once the transformation is made, one can rely on simple geometry to do the rest.

Separators

As discussed previously, the Perceptron Algorithm creates a line by ensuring that each point, as it iterates, is correctly positioned with respect to the same. The manner in which it does so relies upon simple vector arithmetic called the Dot Product. Simply put, the dot product of two equally sized vectors is the sum of the products of each position of the vector:

$$dot(x,y) = \sum_i x_i y_i = ||x|| \times ||y|| \cos \Theta$$

The second portion of the formula above describes the relationship between the dot product and the corresponding angle between the two vectors. The double bar notation above signifies the length of the vector from itself to the origin. This final relationship provides the greatest insight into this notion of separability. In the last case, the cosine of an angle is always 0 at 90°. That would suggest that regardless of the magnitude of any two vectors, their dot product is 0 whenever the angle between them is 90°. The simplest example is using the following two vectors: (0, 1) and (1, 0). These represent the x-axis and y-axis respectively and are definitely 90° apart. Computing their dot product, as expected, will produce 0. In mathematical parlance, any two vectors that are orthogonal[6] have a dot product of zero.

The more interesting phenomena is what happens when the angle becomes smaller and larger than 90°. When the vectors are pointed in the same general direction (i.e. less than 90°), the dot product is positive. When they are pointed in opposite directions (i.e. more than 90°), the dot product is negative. In other words, a single vector can be used to represent a boundary. Looking back at Figure 3, the line can be represented by a single vector where the boundary becomes every point at which the dot product is zero. Additionally, every point at which the dot product is negative becomes the red class and every point at which the dot product is positive becomes the red class. This new boundary vector (let's call it W) then becomes the class separator. All the Perceptron Algorithm does is adjust this vector every time it encounters an incorrectly classified point by nudging this vector W (also called the weight vector) in a direction that corrects the mistake.

Similarity

Although there are many measures of similarity, we will restrict our discussion to what is called the L_2 norm[7] or Euclidian distance. Every vector is endowed with a notion of length[8]. Consider the vector (3, 4). When drawing a line from (0, 0) to (3, 4), we can create a triangle that is three units wide and 4 units high. The hypotenuse (or length of the vector itself) would be the square root of the squares of each element according to the Pythagorean Theorem; in this case 5. In other words the vector (3, 4) has a length of 5. This can be extended to the distance between two vectors by the notion of vector subtraction. When one subtracts two vectors, the resulting vector represents the direction from the end-tip of one vector to the end-tip of the other. This magnitude (or length) of this new vector is exactly the distance between the original two vectors. This is generally represented by the double bar notation: $||x||_2$ and is calculated in the general case with:

$\sum_i \sqrt{x_i^2}$

The distance between two vectors (denoted by[9] $||x - y||$) then becomes:

$\sum_i \sqrt{(x_i - y_i)^2}$

Although there are many other similarity measures, this is the measure that is most often used in the K-Means algorithm. During each step, the distance between each point and the centers is calculated. The center with the shortest distance to the current point wins out and is subsequently assigned.

Objects

While these mathematical notions used to create class separation as well as compute distances between points is extremely useful when creating charts, it is not nearly as intuitive when considering everyday objects. Because of this particular difficulty, I decided to create a library that eases the object to vector conversion[10] . It turns out that there exists a corresponding numerical representation for almost all base .NET data types. These include Boolean (convert to 0, 1 or -1, 1), Byte (0 – 255), Char (0 – 65535), Date (which has numerical representation from 0 time), Decimal, Double, Integer, Long, Short, and Single. When aggregating these data types into an object, one can create a feature for each property in the object and place the corresponding numerical value into a slot in the corresponding vector. The problem arises in the case of strings. Although this is not as straightforward as the other conversions, there are existing strategies to ameliorating the problem. The conversion depends on the type of value contained in the string: single-concept or multi-concept. The single concept strings are things like names, objects (tree, frog, cat, boat), and places. Multi-concept strings are several concepts amalgamated into a long string. Examples include blog posts, summaries, movie descriptions, etc. While the strategies for converting these things into strings are the same, the granularities differ a great deal. In the case of single-concept strings, a character vector can be created that consists of each letter in the string and the counts of each character in the word (see Table 2).

Single-Concept	Corresponding Vector
Cat	{ a: 1, c: 1, d: 0, g: 0, i: 0, o: 0, r: 0, t: 0} [1, 1, 0, 0, 0, 0, 0, 0]
Dog	{ a: 0, c: 0, d: 1, g: 1, i: 0, o: 1, r: 0, t: 0} [0, 0, 1, 1, 0, 1, 0, 0]
Cart	{ a: 1, c: 1, d: 0, g: 0, i: 0, o: 0, r: 1, t: 1} [1, 1, 0, 0, 0, 0, 1, 1]
Dig	{ a: 0, c: 0, d: 1, g: 1, i: 1, o: 0, r: 0, t: 0} [0, 0, 1, 1, 1, 0, 0, 0]

Single-Concept String Vectors, Table 2

It is important to note that the counts are over all examples provided to the learning system. In the case of multi-concept strings, the same kind of vector can be created with a wider granularity. Each word in the string now becomes a spot in the vector with the corresponding value being its count (see Table 3).

Multi-Concept	Corresponding Vector
The red car	{ the: 1, red: 1, car: 1, dog: 0, goes: 0, up: 0} [1, 1, 1, 0, 0, 0]
The red dog	{ the: 1, red: 1, car: 0, dog: 1, goes: 0, up: 0} [1, 1, 0, 1, 0, 0]
Car goes up	{ the: 0, red: 0, car: 1, dog: 0, goes: 1, up: 1} [0, 0, 1, 0, 1, 1]
Dog goes up	{ the: 0, red: 0, car: 0, dog: 1, goes: 1, up: 1} [0, 0, 0, 1, 1, 1]

Multi-Concept Word Vectors, Table 3

Note that in each case, the words or phrases that were most similar were also the most similar when it came to the construction of the corresponding vectors. Note the vector similarity, for example, between The red car and The red dog. These were only different in one position. Additionally, notice that the dot product between The red dog and Car goes up is exactly 0 which means these two phrases are completely orthogonal (perpendicular) to each other!

Code

As discussed earlier, the library I created simplifies this conversion. Consider Code Example 1:

```
public class Student
{
    [Feature]
    public string Name { get; set; }

    [Feature]
    public double GPA { get; set; }

    [Feature]
    public int Age { get; set; }

    [Feature]
    public bool Tall { get; set; }


    [Feature]
    public int Friends { get; set; }

    [Label]
    public bool Good { get; set; }
}
```

Student Class Definition, Code Example 1

In this particular class, we have defined several features and a label. This class has been designed to learn whether a Student is good or not based upon their Name, GPA, Age,

Tallness, and Number of Friends. Once the Feature and Label attributes are used to decorate the class properties, a collection of student objects can then be used to learn a model which can correctly identify the Good label given a new student object as show in Code Example 2:

```
Student[] data = StudentData.GetData();
IModel<Student> model = new
PerceptronModel<Student>();
IPredict<Student> classifier = model.
Generate(data);
...Snip...
classifier.Predict(unknownStudent);
```

Perceptron Model for Students, Code Example 2

Based upon an array of students, the system creates a Matrix (Vectors stacked) for the examples and then runs the perceptron algorithm to create the associated vector W (or separator) in order to predict on new student objects. When the classifier is given an unknown student, it takes the responsibility of filling in the predicted value of Good.

Consider a class that stores blog data aggregated from a feed:

```
public class Feed
{
    public string Source { get; set; }

    [StringFeature(SplitType=StringType.Word)]
    public string Content { get; set; }

    public int Cluster { get; set; }
}
```

Feed Class used in K-Means, Code Example 3

In this case, we wish to create a grouping of blog entries aggregated from a feed. In this case we have what seems to be only one feature. As discussed before, this feature creates a word dictionary in the background (since the Split Type is set to Word) and creates the appropriate word vectors for each example. In this case, the K-Means algorithm is executed in the following way:

```
Feed[] feedData = FeedData.GetData();
ICluster<Feed> kmeans = new KMeans<Feed>();
int[] grouping = kmeans.Generate(feedData, 2);
```

K-Means on Feed Data, Code Example 4

In this case the grouping is returned as an array of group assignments for each item in the feed array.

Conclusion

While the topic of Machine Learning appears to be daunting, the field is quite simple when viewed in its rudiments. We have seen how to predict between two things and a class of things (finite or infinite). We have also discovered some techniques that can assist in finding the general shape of our data whether or not it be a clustering, a hierarchy, or general statistics of our data. The following simple questions can be answered automatically using the described techniques:

1. Can I predict the future based on the past?

2. Is there inherent shape to my data that I am not exploiting?

Answering these questions correctly can certainly assist in crafting business models as well as boost the speed of decision making, ultimately allowing you to maximally profit from your data.

Seth Juarez has a Master's Degree in Computer Science and is also currently pursuing a PhD in the same. His field of research is in Artificial Intelligence, specifically in the realm of Machine Learning. He is a Technical Evangelist for DevExpress where he specializes in data analysis in conjunction with their reporting toolset. When he is not working in that area, he devotes his time to an open source Machine Learning Library specifically for .NET that is intended to simplify the use of popular supervised and unsupervised learning models.

---

[1]See http://en.wikipedia.org/wiki/Ontology_(information_science)
[2]There exist other linkage methods, but these three are the most common.
[3]These points, in fact, do not represent anything other than randomly generated points according to a mixture of 2 multivariate Gaussian distributions rotated about the x-axis using the standard rotation matrix. The bee analogy is for clarification purposes.
[4]Also called Normal distribution
[5]In regular Gaussian Distributions there are two parameters, the mean and standard deviation. The mean is the center of the distribution whereas the standard deviation represents the width of the distribution (same value as the variance squared). In multivariate Gaussian distributions, the mean is also a parameter as well as the covariance matrix. This measure is similar to the standard deviation in that it is a measure of how points vary between each other.
[6]Orthogonal is a mathematical way of saying that two vectors are perpendicular or 90º apart.
[7]There is a generalized L_p norm which corresponds to $||x||_p = \sum_i(|x_i|^p)^{1/p}$
[8]This is true only of a special form of vector spaces referred to as Hilbert spaces.
[9]When the subscript is omitted, it is assumed to be the L_2 norm or Euclidian Distance.
[10]See http://machine.codeplex.com

Improve Your Java with Groovy

By Kenneth Kousen
ken.kousen@kousenit.com http://www.kousenit.com

Groovy (http://codehaus.org/) was never intended to replace Java. It enhances and extends Java. Adding Groovy to your Java will make your life as a developer easier in many ways.

- Groovy adds *closures, builders, and metaprogramming* capabilities to Java
- Groovy *compiles to Java bytecodes*, so it runs wherever you have a Java Virtual Machine
- Groovy is *easy for Java developers to learn and use*

Ant Maven Gradle

- Ant is simple, but low-level. Ivy helps with dependencies
- Maven works well for projects designed for it
- *Gradle* does everything Ant and Maven do, and is *easy to customize, configure, and write*

JUnit GroovyTestCase Spock

- JUnit made automated testing a reality
- JGroovyTestCase is built into the Groovy libraries, extends JUnit TestCase, and adds helpful capabilities
- JSpock is a clean, *easy-to-write DSL for tests*

Groovy works with Java but also goes beyond it. Groovy closures treat functions as first-class objects, leading to much simpler code. Groovy's metaprogramming capabilities produce builders, domain specific languages, and more. In my presentation on *Improving Your Java with Groovy*, we'll look at many ways Groovy can help simplify your life, complete with lots of code examples.

Mobile Interaction

By Christopher M. Judd

One of the things that makes mobile development uniquely different from web or desktop application development is the access to input devices and sensors beyond the keyboard and mouse. Most mobile devices include a microphone, accelerometer, camera, GPS, and multi-touch screen. When used correctly, these features can add another dimension to the end user experience.

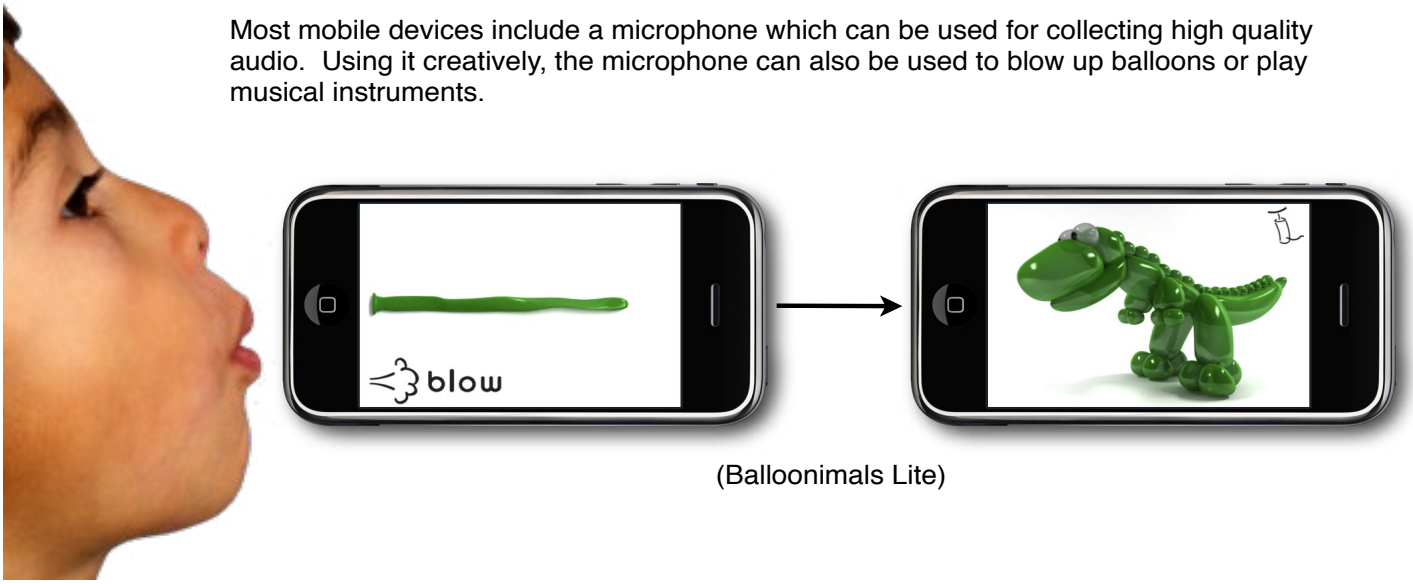
Accelerometer & Gyroscopes

Devices include both accelerometers & gyroscopes which means they know where they are in space and whether they are moving. This can be used by games to drive a car or defeat Sith Lords. However, even business applications have taken advantage of this to provide Etch a Sketch like delete or clear functionality.

(Lightsaber)

Microphone

Most mobile devices include a microphone which can be used for collecting high quality audio. Using it creatively, the microphone can also be used to blow up balloons or play musical instruments.



(Balloonimals Lite)

Multi-Touch Screens

Multi-touch screens enable our devices to morph into anything we want them to be. Our devices can be a piano one minute and become a game pad the next. Our creativity is not limited by physical buttons, and applications mimic their real life counterparts. For example, numerous book reading applications simulate page turning.

(Virtuoso)

Camera & Video

Mobile cameras can be used to capture the moment. They can also be used to capture ISBNs or QR codes which reduce the amount of information that needs to be typed. More interestingly, the camera can also be used to overlay data on the real world to create augmented reality.

(RedLaser)

GPS & Compass

(GPSLite)

Today's devices know exactly where they are by using the GPS and embedded compasses. This means your application can also be aware of where it is being used. This can be exploited to help users and developers. Users can use the device to find the closest gas station and developers can gather analytics of where the application is being used.

Ruby, Unix and Open Source Philosophy

By Matt Yoho

By and large, Ruby programmers feel they embody a distinct culture. They are absolutely obsessed with testing, be it via Test Driven Development (TDD) or Behavior Driven Development (BDD). Practitioners in the Ruby community have contributed greatly to the BDD movement in particular, with tools such as RSpec and Cucumber, which have gained some notice and use from non-Rubyists. They have a history of creating clever and playful hacks and somewhat absurdist inside jokes and memes; thank a Rubyist if you've ever chuckled at the phrase "chunky bacon". They have a reputation, both within the Ruby community and outside of it, for becoming enamored with the shiniest, newest toys, such as the swath of emergent NoSQL tools. And they sometimes give the impression, thanks in large part to the increasingly popular and historically brash Rails web application framework, that they've cockily set out to change the workings of the information technology world.

But for all the rapid change, vocal enthusiasm and novel output that Ruby developers may be known for, at its best, many of the core principles and practices of Ruby can be traced to knowledge that the larger Unix community has practiced and refined for decades. Unix has a complex history, and the intricate threads of its lineage and progeny are beyond the scope of this article. For the purpose of discussion here, we'll take Unix to refer to any effectively POSIX-compliant system, which includes the various flavors of GNU/Linux as well as the contemporary BSD-based systems, including Mac OS X. But to understand how these older ideas have influenced Ruby, we'll have to look first at some history.

A Brief History

Understanding the genesis of Ruby and the interrelationship with Unix it has had since birth involves looking at least briefly into the history of Unix and its development landscape. The first version of Unix was written at Bell Labs, debuting in its earliest form in 1969, as a relatively lightweight timesharing and multitasking (or multiprocessing) operating system. It was initially put to use as a text processing system for the preparation of patents. In 1972, it was reimplemented primarily in C, a very low level language that was created in part for the purpose of that implementation. Command-line driven, users interacted with the system via the shell, the utility responsible for command interpretation. A few variants on shell scripting languages (such as sh, ksh, csh, or the more-recent bash and zsh) came into being over time to ease this interaction using abstractions and to allow for complex batch processing. Being of a higher level of abstraction than C code and the underlying system APIs of Unix, shell scripts could serve as glue code, quickly tying together more fundamental system utilities and low-level programs. The relative ease of implementing solutions in this way, along with early Unix design decisions to make utilities narrow of purpose and simple to tie together, lead to an early adoption in Unix culture of rapidly prototyping systems by leveraging higher-level languages.

In 1987, Perl emerged as a general purpose Unix scripting language meant to make scripting easier. It rapidly gained widespread use in the early days of web programming due in part to its strengths in text processing. It also made much of the low-level Unix C API available in a friendly, dynamic wrapper, making it highly effective as a rapid development language. But it was criticized for having many rough corners and for having poor maintainability as a project scaled. In its early days, it lacked viable object-oriented features when object-orientation was becoming the dominant programming paradigm.

In February of 1993, Yukihiro Matsumoto, also known as Matz, decided to create a new scripting language to supersede Perl. Six years later, in his forward to the first edition of *Programming Ruby*, the first English-language publication on Ruby, Matz stated that his original goal in creating the language was to have a more powerful and object-oriented scripting language than Perl. (http://ruby-doc.org/docs/ProgrammingRuby/html/foreword.html) The name Ruby is itself an oblique reference to Perl, via the latter's homophony with the "pearl". Although Ruby's features, type system, and object model were strongly

influenced by Smalltalk, its syntax and, perhaps more importantly, its level of integration with the underlying POSIX system were adopted from Perl.

The Ruby standard library includes a slew of POSIX system calls, such as `Kernel::fork`, `Socket::socketpair`, and `IO::pipe`. It inherits from Perl's great utility for leveraging the underlying system APIs to quickly tie together complex systems. (This is one of the many reasons that JRuby, the Ruby implementation on the JVM, is so impressive: for implementing features, such as forking, that don't map well to the underlying VM.) For example, it's possible to write full-featured networked servers using TCP or Unix sockets entirely in Ruby code. And with the use of Ruby's extension API for providing Ruby libraries implemented in native C, it's possible to replace performance-critical sections of an application with faster versions of the code, which has led to many effective server applications written in Ruby. To see a discussion of one such server application, read Ryan Tomayko's article *I like Unicorn because it's Unix*, an excellent post discussing the Unicorn web application server. (http://tomayko.com/writings/unicorn-is-unix) In it he discusses how Unicorn, written almost entirely in pure Ruby, uses the Unix system fundamentals `fork` and `select` to take a traditional multiprocessing (pre-forking) approach to servicing web requests, employing techniques such as relying on the underlying system kernel for a simple form of request scheduling.

Documentation Conventions

In Ruby, there is a documentation convention for distinguishing between class (or module) methods and instance methods. Class methods are indicated by using a period between the class name and method name, like so:

`Class.class_method` or by using a double colon: `Class::class_method`.

Instance methods are indicated by using a hash: `Class#instance_method`

In addition to Unix API wrappers in its standard library, Ruby has several ways of direct syntactical support for interacting with POSIX shells. The backtick operator, ``, can be used to wrap a command to be executed in a sub shell and return the result of that command. This allows a Ruby script to easily defer some complex or specialized processing to a system utility or external application. The following is some example code that shells out to the system to perform MIME type verification on a given file.

```
module MimeChecker
  ACCEPTABLE_FILE_TYPES = %w(image/png application/pdf)

  def self.accepted?(file)
    ACCEPTABLE_FILE_TYPES.include?(mime_type(file))
  end

  def self.mime_type(file)
    mime_type = `file --brief --mime "#{file.path}"`
    mime_type.strip!
    mime_type.gsub!(/;.*$/,'')
    mime_type
  end
end

file = File.open('funny_cat.jpg')
MimeChecker.accepted?(file)
#=> false
file = File.open('funny_cat.png')
MimeChecker.accepted?(file)
#=> true
```

Listing 1: The MimeChecker module

This example shows the definition of a Ruby module `MimeChecker` with a module method, `MimeChecker#accepted?` that expects a file object and returns true or false indicating whether or not the file is of an acceptable MIME type. The most important work done in verifying the MIME type in the above example is handled by the system utility `file`, which is invoked and passed parameters inside the backtick operators. The rest of the code is just scrubbing and checking whether or not a parsed-out string is included in a collection of known types.

Imagine, though, if instead of using the already available `file` utility, that functionality were implemented in Ruby instead. It would clearly result in more code, more effort to maintain that code, and would almost surely be less performant than a native utility that has been scrutinized potentially for decades. This also illustrates an important principle popularized by Unix culture that is sometimes phrased as "small pieces, loosely joined". Or as Doug McIlroy, one of the founders of Unix and the creator of Unix pipes, phrased it, quoted in *A Quarter Century of Unix*, "This is the Unix philosophy: Write programs that do one thing and do it well."

The Influence of Unix Philosophy

Eric S. Raymond, a prominent Unix programmer and historian referred to this principle as the Rule of Modularity in his book, *The Art of Unix Programming*, and describes it as "Write simple parts connected by clean interfaces." He

identified several other principles that had emerged over time, such as the Rule of Simplicity, which states "Design for simplicity; add complexity only where you must." (It might also be recognized as the pithier, though slightly more abrasive, "KISS: Keep it simple, stupid.") Other rules include the Rule of Least Surprise, which dictates that interfaces should always do the least surprising thing; the Rule of Economy, which states that programmer time should be conserved in favor of less-expensive machine time (and implies the value of high-level languages); and the Rule of Generation, which suggests that one should write programs that write programs whenever possible. Rubyists, like developers across most languages, are willing to embrace all of these principles, but perhaps more than any other developers this side of Lispers, they relish the opportunity to accomplish this last rule via metaprogramming.

An illustration of the Rule of Simplicity and of Ruby's facility as a glue language is the following code, extracted from a system written to rapidly prototype near real-time generation of screenshots of arbitrary web pages. Ruby was used to write the screenshot daemon (the term for a long-running background service in the Unix world), providing wrapper code to coordinate the various components in the system, which included a minimal X11 windowing environment and `wmctrl`, a command line utility for manipulating X windows..

While the daemon was running, it was possible that up to a dozen browser windows could be open at once, each loading a different page to be captured. The `wmctrl` utility was used to give the correct browser window focus, allowing its image to be captured. The following code uses another of Ruby's syntax elements for invoking shell commands, `%x{}` and allows manipulation of the windows currently open within the daemon.

```
# WindowList represents currently open windows
# Window allows operations on a GUI window
# Both rely on the wmctrl utility for actual work
class WindowList
  class << self
    # Format of wmctrl output:
    # 0x0040003d  0 10563  ey02-s00576 Mozilla Firefox
    HANDLE     = /(0x[a-f0-9]{8})/
    PID        = /\d\s+(\d+)/
    SLICE      = /[a-z0-9\-_]*/
    DATA_REGEX = /^#{HANDLE}  #{PID}  #{SLICE} (.*)$/i

    def windows(for_pid = nil)
      windows = []
      data.each_line do |line|
        md = line.chomp.match DATA_REGEX
        unless for_pid && (md[2].to_i != for_pid.to_i)
          windows << Window.new(md[1], md[2], md[3])
        end
      end
      windows
    end

    def window(id)
      windows.find { |w| w.id == id }
    end

    private
    def data
      # %x{} is an alternative shell command syntax
      %x{wmctrl -lp}
    end
  end

  class Window
    attr_reader :id, :pid, :name
    def initialize(id, pid, name)
      @id, @pid, @name = id, pid, name
    end

    def refresh!
      @name = WindowList.window(@id).name
      self
    end

    def close!
      # %x{} is an alternative shell command syntax
      %x{wmctrl -i -c #{@id}}
    end

    def focus!
      %x{wmctrl -i -a #{@id}}
    end
  end
end
```

Listing 2: A wmctrl wrapper

Although there is some noise in the regular expression used for parsing out window information, the example illustrates using the output obtained by the running command "`wmctrl -lp`" to capture a list of open windows and represent them in a collection of Window proxy objects. It is then possible to give focus to an individual window or close one, by again calling the `wmctrl` shell utility with appropriate arguments. This functionality displays another truism acknowledged by the Unix approach at work: text is a universal format.

Code examples can ably illustrate something like the KISS principle, but the Rule of Least Surprise, particularly at the language level, is a subtler thing. There is a phenomenon reported by Ruby programmers that Ruby APIs tend somehow to be highly intuitive. Make an assumption about what a method's name and its signature should be, without consulting the API documentation, and one commonly turns out to be correct. There is undoubtedly some selection bias at work here. But it seems likely that the language does promote this sort of "intuition-driven" approach. Matz is widely quoted as saying he designed Ruby to be "natural, not simple" and to reflect the way that a programmer thinks about a problem. Because of this, Ruby probably reflects

the way that Matz thinks about problems! And while it seems plausible that many Rubyists share similar mental approaches to Matz and to each other, one of the beautiful aspects of Ruby is the freedom it gives you to express things in different ways.

One convention that illustrates this expressiveness, and is enabled by Ruby, is the practice of using punctuation in certain method names. It is common to see "truthy" methods – ones that return true or false – have a question mark attributed to their name, such as `String#blank?`. Similarly, methods that end with an exclamation point tend to indicate one of two things: that an exception may be raised or that the method mutates the state of the object that receives it. There is nothing in the Ruby interpreter that enforces these conventions; they are purely idioms. While not perfect or foolproof, they provide one of the sources of esthetic convenience that serves Ruby's stated goal of being a pleasure to use.

When it comes to the Rule of Economy, a Ruby developer often must exercise restraint so as to not take it too far. Metaprogramming is the tool of choice for a Ruby developer desiring to save programmer time by applying some classic hacker laziness, and as a tool it is very powerful. It's also very fun, and the combined affect can be seductive, particularly to new Rubyists. In practice, it is a technique best-reserved for rare use. But for a slightly contrived example, one could desire a class whose instance would respond to any method of the form `#say_hello_world`, where the message name would by `say_` followed by a series of words separated by underscores, and the resulting output should be those following words formed into a proper sentence, with capitalization and appropriate spacing. To a Rubyist, this looks like a job for metaprogramming, which is, roughly speaking, the practice of writing code that writes code.

```
class Greetings

  def method_missing(name, *args, &block)
    message = name.to_s

    # Verify the method matches the say_*
    # pattern we're looking for
    if message =~ /^say_\w+/

      message.sub!(/say_/, '')
      message.capitalize!
      message.gsub!(/_/, ' ')

      # Here we construct a string
      # corresponding to a method definition
      # using Ruby's string interpolation
      method_body = <<-EVAL
        def #{name}
          "#{message}"
        end
      EVAL

      # class_eval will evaluate the string
      # inside the Greetings class, defining
      # the method called 'name'
      self.class.class_eval method_body

      # send lets us dynamically call this
      # newly-defined method
      send(name)
    else
      super
    end
  end

end
```

Listing 3: Dynamic method generation

Here, Ruby's built-in support for programmatic method definition is combined with the ability of its objects to handle messages that they don't already understand. `Object#method_missing` is a special method that is invoked whenever the method called on an instance is not already defined. By applying a combination of these features, any methods that are called on (or in terms more idiomatic to Ruby, any messages that are received by) an instance of `Greetings` that correspond to the `say_*` pattern will be converted into actual, defined methods that return the appropriate string. This is a naive example of a portion of Ruby's facility at metaprogramming.

Open Source and Free Software

Today, Unix and POSIX have become associated for many with the Free Software or Open Source Software movements. Though Open Source Software and Free Software are collectively referred to as "OSS", the two are not really synonymous. The colloquialism "free as in beer or free as in speech" is meant to illustrate the difference between the two, and it refers to distinctions in the software licenses, the legal documents describing the terms of use for software packages and libraries, that are associated with each type of OSS project.

In the OSS world, there are restrictive licenses, such as the set of GPL licenses[1] and there are permissive licenses that impose virtually no restrictions on the end product. The latter include licenses like the MIT license and the BSD license. Ironically, the original Unix system as created by Bell Labs was proprietary and closed source. Fortunately, Bell Labs shared versions of the source with several universities, such as Berkeley, which spent several years

writing their own version of Unix, BSD Unix, which was free of proprietary code and some of the first software placed under the BSD license. Although the BSD family of Unix systems (the descendants of which today include NetBSD, FreeBSD, and consequently Mac OS X) had existed prior to it, it wasn't until the release of the GNU/Linux that an Open Source version of Unix really came into widespread use. In spite of this, works such as Eric S. Raymond's *The Cathedral and the Bazaar* have championed OSS as a means of enabling robust and successful commercial software systems by leveraging the power of distributed scrutiny and leveraging diversity of ideas.

Matz had this to say about the differences in OSS approaches in an August 2000 interview with Japan, Inc., "There are various merits, but for me the biggest is freedom. I prefer the term 'free software' more than open source." The Ruby language itself is dual licensed and the GPL and the somewhat less-restrictive Ruby license.

The Rails framework is licensed under the MIT license, and reflects the larger trend in the Ruby community to use permissive licenses. The majority of Ruby libraries, particularly those produced since the debut of Rails, are written to be used and modified freely, reflecting a seemingly laissez faire attitude toward code sharing. It is perhaps because of this aspect of Ruby culture that a success, and general boon to OSS, such as the GitHub source code repository service could spring forth from it and grow in a short time well beyond its borders.

The Essential Nature of Good Documentation

Dennis Ritchie, the author of C and one of the original Unix engineers, identified the value of good documentation as enabling the success of Unix in his introduction to *Advanced Programming in the Unix Environment*:

It is important that papers and writings about the Unix system were always encouraged, even while the software of the system was proprietary. In fact, I would claim that central reason for [Unix's] longevity has been that it attracted remarkably talented writers to explain its beauties and mysteries.

In addition to its widely-recognized man pages, which provide online documentation for its system calls and utilities, Unix has a long tradition of academic papers and seminal books being published about it. Books such as *The Art of Unix Programming, The C Programming Language*, and the *Unix Network Programming* series are still widely regarded as classics. Ruby is no different in its need for, and its possession of, good documentation. Since the publication of *Programming Ruby*, the first comprehensive English-language discussion of Ruby, in 2001, more high-quality traditional-style discussion of the language and its techniques have been published.

One obvious advantage that the Ruby community has over the early Unix pioneers is the ubiquity of the internet (coupled, of course, with the relative power of home computers and software.) Rubyists and Ruby enthusiasts have embraced the practice of screen casting, or recording video and audio clips and publishing them on the web, producing video tutorials that may run anywhere between five minutes to an hour in length. PeepCode is a successful commercial venture that produces and sells high-quality screen casts and PDF guides on various technical topics but with a slant toward Ruby and Rails (http://peepcode.com/). Railscasts, by Ryan Bates, is another edifying and prolific screen cast series focusing on Ruby and Rails related topics that is entirely free (http://railscasts.com/).

Blogging as can be found across most subcultures of programming is also a common practice among Rubyists. Geoffrey Grosenbach, the operator of PeepCode, also provides a beautiful blog series in which every post is unique and is laid out in an online e-zine-style format (http://blog.peepcode.com/). In true hacker fashion, he dedicates a lengthy tutorial post describing the software stack he has custom-built to produce his distinctive posts and encourages others to build their own version. Yehuda Katz, one of the Rails Core team members responsible for the release of Rails 3, blogs profusely, illuminating Ruby and Rails internals or best practices (http://yehudakatz.com/). The roll call of helpful Ruby bloggers could go on for pages.

Ruby also has many options for documenting actual code. RDoc, the de facto standard for Ruby code documentation, is generated and made available any time a Ruby library is installed. A competing documentation tool, YARD, provides slightly different options for its output and prettier formatting by default. Ryan Tomayko produced two different tools, both highly useful for documenting libraries and utilities.

Ronn is a Ruby tool for producing roff (a text formatting tool dating back to the earliest releases of Unix at Bell Labs) output, which can then be used to generate man pages or HTML (https://github.com/rtomayko/ronn). Rocco is a Ruby port of Docco, which generates pretty inline documentation that's ideal for describing the usage of an API (https://github.com/rtomayko/rocco).

Freedom in the Ruby Language

David Heinemeier Hansson, the original creator of Rails, championed the Rubyist's ability to reopen classes and add or modify functionality, a practice known as monkey-patching or duck-punching (a reference to Ruby's duck-typed nature), during his keynote at RubyConf 2010. Because of Ruby's dynamic and open nature, it's very straightforward to add behavior to preexisting classes. This contrasts sharply with the concepts of `sealed` in `C#` or `final` in Java, two languages much more interested in restricting programmer options. He referred to an instance where, in the ActiveSupport library (one of the sub-libraries of Rails) he added a method largely to spite a commenter on the Reddit message boards. Out of the box, Ruby's `Array` class has two instance methods, `Array#first` and `Array#last`, that allow one to bypass the accessor syntax to get at the first and last elements in the array respectively. This is done purely for convenience and esthetics, demonstrating a small amount of Ruby's programmer-friendly attitude. Hansson decided to go further toward this by adding several more convenience methods, such as `Array#second`, `Array#third`, and so forth. When a forum commenter took exception to this, decrying it as unnecessary, Hansson responded by introducing a new convenience method, `Array#forty_two`, in the next version of the library.

```
# From ActiveSupport, a part of the Rails framework.
# This demonstrates Ruby's open classes by adding
# methods to a core class, Array.
class Array
  def second
    self[1]
  end

  def third
    self[2]
  end

  def fourth
    self[3]
  end

  def fifth
    self[4]
  end

  # etc.
  # Equal to <tt>self[41]</tt>.
  # Also known as accessing "the reddit".
  def forty_two
    self[41]
  end
end
```

Listing 4: `ActiveSupport` **extensions to** `Array`

Although there are multiple ways to take advantage of Ruby's open classes, the above is probably the most straightforward way of doing so: simply re-declare the class – `Array` in this case – and start adding additional methods. In the context of an instance method on a Ruby class, `self` refers to the particular instance on which the method has been called.

Hansson claimed during his keynote that he had never once actually used the `forty_two` method. To him, the wonderful part of its existence was that it illustrated the freedom and flexibility that Ruby gives developers over the language itself. This freedom has inspired all manner of ingenious hackery, only a very small fraction of which was created initially to spite forum denizens.

Hackery and Expression

Hacking, despite its relaxed contemporary meaning, roughly equivalent to working on enjoyable code, has a more specific classical meaning that harkens back to the early days of the MIT AI Research labs. A hack, in this early sense, must include a special degree of cleverness or ingenuity and must inspire a sort of delight in its perpetrator and its audience. It may have, but does not require, a degree of utility beyond the hack itself. The history of prank back and forth between Caltech and MIT is rife with apocryphal examples. Steven Levy's *Hackers* is a fascinating examination of early hacker culture and its impact on the computer age.

One example of the Ruby hacking spirit, and its infectiousness, is the annual Rails Rumble (http://railsrumble.com/), a 48-hour competition where teams of up to four developers and designers compete to build from scratch the best open-source Rails application possible. Started in 2007 and inspired by the similar Rails Day event which ran the two previous years, this past year's (2010) contest saw 500 teams compete. The Rails Rumble has inspired similar competitions in Ruby's sister communities, such as the Django Dash for Python developers and the Node Knockout for Node.js JavaScript developers.

Another wonderful example of clever hacking for its own sake in the Ruby world is Chris Wanstrath's URLDSL module, the entirety of which can be found in a GitHub gist here: http://ozmm.org/posts/urls_in_ruby.html

In less than 100 lines of actual Ruby code, Wanstrath exploited the dynamism of Ruby and the ability to modify the behavior of Ruby's core classes (in this case, both `Integer` and `Symbol`) to allow one to make a web request and capture the result simply by typing out the qualified URI to the page. The following example shows retrieving the JSON data representing a user's recent activity on GitHub and extracting out their username from the latest action they have taken.

```
require 'urldsl'
require 'json'

URLDSL do
  activities    = https://github.com/mattyoho.json
  latest_action = JSON.parse(url.to_s).first

  puts latest_action['actor']
end

#=> 'mattyoho'
```

Listing 5: The URLDSL module in action

The result of the puts call will print out the username, which in this case is 'mattyoho'. Although the practicality of this hack is relatively low, the novelty is quite high and what it reveals about the malleability and expressiveness of the language (not to mention the cleverness of the hacker) is impressive. As such, it should be considered a hack in the classic sense.

Ruby's patron saint of creative expression and delightful hackery has been a coder and teacher known as Why The Luck Stiff, or _why for short. Though he spoke at conferences, often about the value of teaching Ruby and programming to others, he guarded his true identity closely and was known primarily through his prolific amount of work, which always seemed to come from a slightly off-kilter perspective on the world. He created the original version of Try Ruby! (http://tryruby.org/), a site that lets visitors play with the Ruby language in their browser. He wrote and freely published the classic introductory Ruby programming book *Why's Poignant Guide to Ruby* (now hosted at http://mislav.uniqpath.com/poignant-guide/) that introduced the world to the concept of chunky bacon, and recorded an accompanying album of songs (ostensibly) about Ruby that was freely released alongside the book. He created the hpricot Ruby library, which provided possibly the most pleasing XML parsing API to date but has since been superseded by the API-compatible Nokogiri library. He created the Shoes framework (http://shoesrb.com/), a small graphical framework for Ruby, and Hackity Hack (http://hacketyhack.heroku.com/), a companion project with the aim of introducing children and newcomers to programming.

His works revealed a significant amount of technical ability, but that was not their most important or appreciated aspect. (He was, not surprisingly, a prominent exception to the Ruby's ubiquitous TDD practice.) What was beloved about _why and his work was the celebratory joy for the act of creation they expressed. _why has, unfortunately, retired from the Ruby community and removed his persona from the world as well as he was able. His absence has left a strangely-shaped hole in Ruby culture (Matz publicly asked him to come back, stating that he was missed, during the keynote of RubyConf 2010) but the spirit of playfulness and hacking are fundamental to the Ruby community and thus live on.

Conclusion

The Ruby community and the technology it produces are a product of many cultural and technical influences and it inherits powerful legacies established by the Unix and OSS cultures. By acknowledging, absorbing, and iterating on the dominant principles of the practitioners that have come before, Rubyists make themselves valuable members of that larger community and increase their chances of producing high-quality, long-lived, and useful software.

Matt Yoho is a developer and agile enthusiast with a love for Ruby and the web who works for EdgeCase, LLC in Columbus, OH. He is a supporter of the Software Craftsmanship movement and is the coordinator of the apprenticeship program at EdgeCase. A teacher, trainer, and speaker when possible, he likes comic books, karaoke, Free Software, and sweet potato fries. He is one fairly hep cat.

[1]Licenses such as those in GPL family are referred to as viral licenses because they restrict what you can and cannot do with your product if it is built using GPL code by requiring any products that are built and distributed with that code by licensed under the same or similar terms.

Planning for Success

By Jon Kruger
http://jonkruger.com/blog

We all want to succeed at something, whether it's your career, a sport, being a good parent, or whatever else you're working at. But do you have a plan for succeeding? The fact of the matter is that if you just hope that you're going to succeed without doing something to make sure that you actually do succeed, it's unlikely that you will achieve your goals. Most people do not succeed accidentally; they do it through many small decisions and actions over the course of time that leads them to success.

First, think about your personal mission statement in life. What are your priorities, and what is most important to you? Consider everything — your family, God, friends, activities, work, etc.

Next, think about your goals. Do you want to learn how to do mobile development? Run a half-marathon? Lose 20 pounds? Brainstorm and write down a bunch of ideas, then evaluate them all and figure out which ones are most important. Think of your life and career like a business. What adds the most value to your life at this point in time? You can have different goals for different areas of your life (e.g. your marriage, your kids, your career). Write them down and share them with someone else so that they can help you stay on track.

Now you have to execute. When you have to make a decision about what to do with your time, make the decision based on your goals and priorities, not because of your feelings, what others might think, or what others say you should do.

This is what I do to make sure that I'm making the most of my life and it's really helped me keep my priorities straight and be a better husband, father, and software developer.

soapUI: The Worst Name a Great Tool Ever Had

By Nick Watts
http://thewonggei.wordpress.com/tag/soapui/

In Software, sight of the word "soap" connotes the acronym SOAP, which, to many, means web services. So it's perfectly natural for you to assume that soapUI is a tool that deals with just SOAP services. I'm happy to say that you're wrong. soapUI (http://www.soapui.org) is a misnomer and is actually a sophisticated functional testing tool that works with a laundry list of distributed technologies like SOAP and RESTful web services, HTTP, JDBC, AMF and JMS.

The bulk of the value in soapUI functional tests originates from the tool's powerful assertion capabilities. Built-in assertions are used to make common verifications that make sense for the technology under test. More detailed and customized assertions can be created with XPath or XQuery expressions. Even finer grained control can be harnessed through scripting. soapUI allows the test author to write test setup and teardown scripts and test assertions in either JavaScript or Groovy. Based on context, different helpful objects are automatically made available to the scripts to allow access to the request and response messages as well as soapUI's test harness. Tests can be executed against either the real service or mock services, which soapUI will help you setup and configure.

soapUI functional tests are easy to automate. soapUI includes a command line utility that allows flexible running of test scripts. The command line utility can be integrated with JUnit and can be configured to output test result reports

that match the JUnit output format. The compatibility with JUnit makes it easy for you to add soapUI test runs into your Continuous Integration server. soapUI is also compatible with Maven 1.x and 2.x.

Even though soapUI is poorly named, it's a great tool for writing functional tests against an assortment of distributed services.

Introduction to Regular Expressions

By Matt Casto
http://google.com/profiles/mattcasto

Have you ever needed to find a particular combination of words in a long document, possibly to replace it with something different? How about matching user input with a complex list of possible values like an email address? Sure, you could write code in any language to read a string of text character by character, but that code could be horribly inefficient and, likely, rife with bugs. Regular Expressions, or Regex, is a language tailored for this task, and is one of those tools that every developer should have in their toolbox. You can do your job without regular expressions, but knowing when and how to use them will make you a much more efficient and marketable developer.

There's something of a stigma attached to Regular Expressions and makes some people hesitant to learn about it. It's considered a write-only language – it's much more difficult to read a Regex someone else wrote than it is to write it yourself. Also, there's the famous quote, "Some people, when confronted with a problem, think 'I know, I'll use regular expressions.' Now they have two problems." While those points have merit, they shouldn't dissuade you from learning what Regular Expressions are, and the extent of their capabilities.

Regular Expressions were created in the 1950's using a mathematic notation called regular sets, but was popularized by the "ed" and "grep" programs in Unix. With Perl came the Regex libraries, and these days all major programming languages have Regular Expression libraries based on these roots.

There are plenty of resources available online for learning Regular Expressions. You can get advice from the RegexAdvice forums (http://regexadvice.com/Forums/) or search for specific Regular Expression solutions at RegExLib.com (http://regexlib.com/). Whether you're starting from scratch, or want to go beyond novice, "Mastering Regular Expressions" by Jeffrey Friedl (http://oreilly.com/catalog/9780596528126/) won't steer you wrong.

A Slideshare presentation of the above can be found at http://www.slideshare.net/mattcasto/introduction-to-regular-expressions-1879191.

http://xkcd.com/208/

Introduction to Android Development

By Jason Farrell

Excerpts from this article taken from http://bit.ly/GKmFI

I was watching a foreign cartoon recently that was released in 1997. One of the characters in the episode was being harassed by another character. The method of harassment was a large floating rotary phone following the character wherever she went. The origin of this entertainment was Japan, which is perhaps one of the leading mobile technology nations on the planet. At this time, cell phones were just starting to make their way into mainstream culture.

Fast forward to today, cell phone adoption is extremely widespread with most people carrying at least one device. Most of this adoption is in the form of "dumb" devices where the primary, if not singular function, is to make and receive phone calls. However, the development of "smart" devices is outpacing most other technology areas, while the rise of popular services on the Internet has helped bring these devices to the forefront of our lives. Furthermore, the level of maturity has risen dramatically, which has assisted with making the devices easier to use and thus hastened their adoption. As these phones steadily become the norm the possibilities of what we can accomplish with them has become immense.

In June of 2007, Apple released its first iPhone. Based largely on the technology proven through the iPod, the iPhone introduced a new breed of smartphone. Rather than being geared for a specific function, the iPhone gravitated towards allowing personalization, mostly through "apps". The result was the creation of a whole new segment of the cellular device market: the consumer smartphone market. Subsequently, the market exploded to the point that last year it sold 172,373,100 units worldwide (http://tcrn.ch/99dBk4). The personalization aspect introduced by Apple caused people to begin looking at smartphones not as just phones, but as tools to improve and help manage their lives.

In April of 2009 Google entered the market by releasing the first version of their Android operating system, codenamed Cupcake. By the end of the same year, two additional versions of Android (Donut and Éclair) had been released and its market share had soared to 4% worldwide, placing them 6th among major operating systems. Figure 1 illustrates the sum total of market share for all operating systems worldwide at the end of 2009.

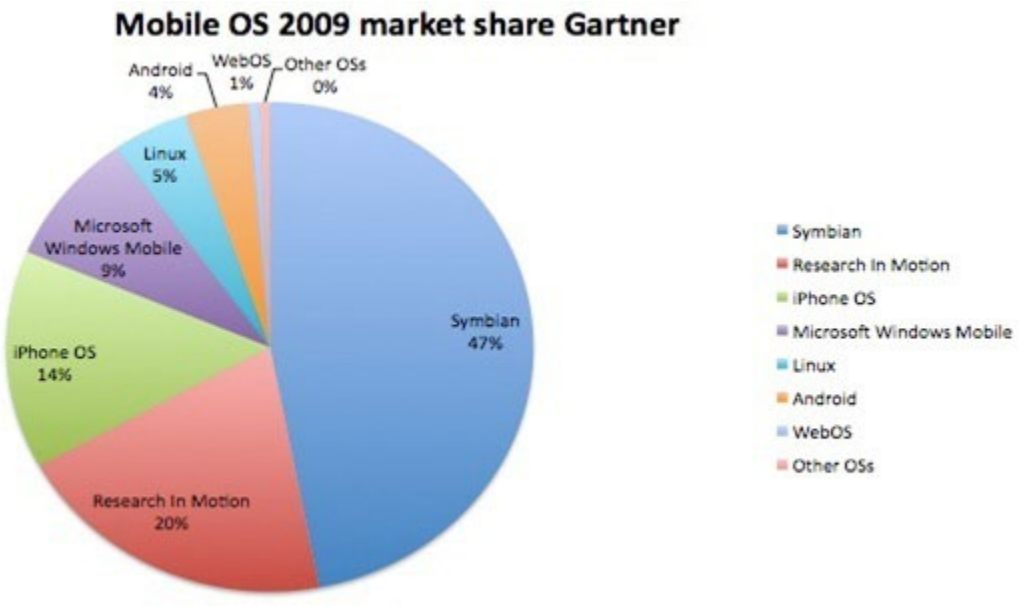


Figure 1

Fueled by sales of the Droid, Android's market share swelled to 9% by February of 2010 and by July had reached 13% of the world market, adding 160,000 new users daily. In the United States, the first quarter of 2010 saw Android sell more phones than any other manufacturer, including Apple. Then, despite the release of the iPhone 4 and backed by the releases of the Droid 2 and Droidx, Android took the crown in the second quarter of 2010 (http://bit.ly/ay3AAq) leading many to believe that it will soon surpass Apple in worldwide market share.

The Versions of Android

As of November 2010 four versions of Android have been released with two known versions waiting in the wings. Each version has built on top of the others, expanding the features offered and supported by the operating system. The addition of these new elements has allowed Android to continue to gain market share by adding to the experience offered by the platform.

Android 1.5 – Cupcake

Released: 4/30/2009

Notable features:

- Smart Soft Input Device (SIP – On Screen Keyboard)
- Home Screen Widgets
- Live Folders
- Video Recording
- Picasa Integration
- Voice Search
- Car/Stereo Bluetooth

The first release of Android added the basic feature set for a smartphone and supported the basic hooks to leverage them through the mobile OS.

Android 1.6 – Donut

Released: 9/15/2009

Notable Features:

- Quick Search Box
- VPN Support
- Gesture Support
- CDMA Support
- OpenCore2 Media
- WVGA Resolution
- Text-to-Speech

Donut's main offering was support for the new WVGA resolution standard. Donut also included support for gestures which gave it the ability to seriously compete with iPhone UI. CDMA support was also added, which opened Android to other non-GSM carriers and allowed it to have better penetration in America.

Android 2.0/2.1 – Éclair

Released: 10/26/2009

Notable Features:

- Updated Graphical Interface
- MSFT Exchange Support
- HTML5
- Digital Zoom
- Bluetooth 2.1
- Multiple Account Support

Éclair is often considered to be the "turning point" for Android. It launched on the immensely successful Milestone from Motorola, known as The Droid. The Droid is often credited with singlehandedly reviving Verizon's smartphone offerings and making Android a legitimate alternative to the iPhone.

Android 2.2 – Frozen Yogurt (Froyo)

Released: 5/20/2010

Notable Features:

- Chrome JS Engine Upgrade
- Voice Dialing
- Flash 10
- Integrated Calendar
- Tethering and Hotspot
- Multilingual Keyboard

Froyo added support for Adobe Flash, something which set Android apart from the iPhone. Support for natively tethering or creating a mobile hot spot was also added. Currently, Froyo runs the majority of Google phones in the world (http://engt.co/fPSPW9).

With these releases Android has closed much of the feature and experience gap that existed between itself and other mobile platforms, mainly iOS; however, even Google acknowledges that they have a way to go to surpass the iPhone in particular, especially with respect to the UI. The following upcoming releases of Android will likely address these shortcomings.

Android 2.3 – Gingerbread

Expected Release: December 2010

Notable Features:

- Built-in video chat
- Integrated Google Apps
- Updated User Interface and experience

The time might finally be at hand for Android to take its final step to match the iPhone: the UI. Recently Matias Duarte defected from Palm to Google; it was Matias who was responsible for the UI used by the Pre: WebOS. It is believed that he was hired by Google to help Android undergo a major facelift and provide a clean familiar UI that can compete with the iPhone.

Android Futures – HoneyComb

Expected Release: 2011

Nothing is presently known about this release outside of the codename.

Programming for Android

Building on existing technologies, Google's goal was to provide a cross platform means for the development of Android applications through the use of existing tools like Eclipse and Java to support a modern developer experience. Eclipse (http://www.eclipse.org) is a tool that is already widely used for development in a variety of non-Microsoft technologies. It features a plugin model which allows for other development environments to be supported by the core program. For the purposes of developing Android apps the Android Developer Tool plugin is used[1] .

To begin, first download the latest Eclipse IDE (at the time of this writing the current version was 3.2 Helios). Once it is installed, use the Android Developer Portal (http://bit.ly/m3rcP) and follow the instructions for installing the Android Developer Tool (ADT). Once it is installed, you will need to choose what versions of Android you will support. The best way to do this is to launch the Android SDK and Android

Figure 2

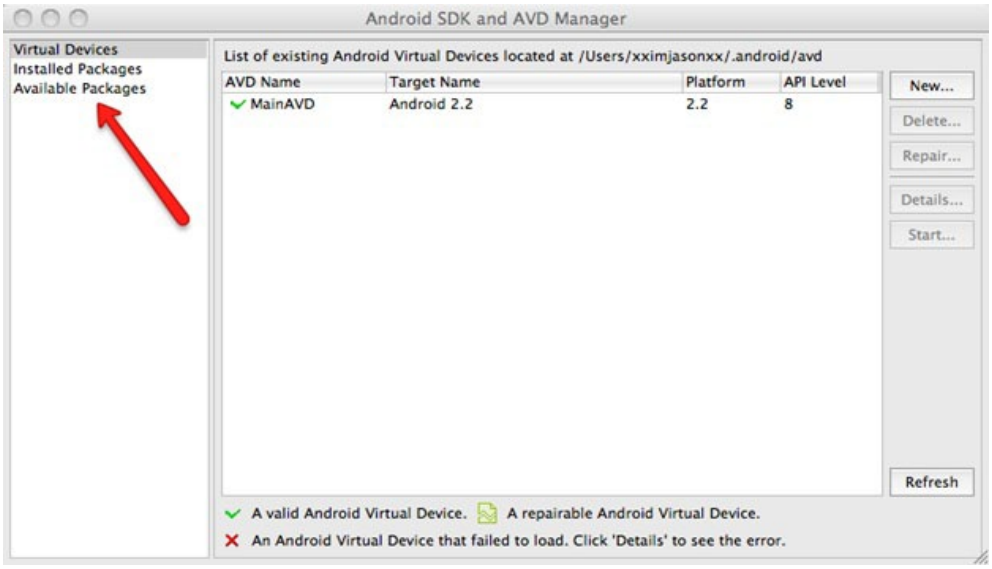


Figure 3

Virtual Device (AVD) Manager from within Eclipse; click the button as shown in the Figure 2. This will display the screen shown in Figure 3.

The first thing you will need to do is install an Android runtime. From the dialog expand "Available Packages". From this window you will be able to select the various Android runtimes. Installing will bake them right into Eclipse. Select only the 2.2 release (API Level 8).

Once the installation is complete you will need to create a virtual device to emulate the Android OS. The one shown in Figure 3 emulates a device running the Froyo (2.2) release. You can configure multiple device profiles to test different hardware, resolutions, and OS versions.

To create your first Android program, simply open Eclipse and choose File -> New Project and pick 'Android Project' from the list shown in Figure 4. Once you complete this step the 'New Android Project' dialog will appear. You will need to fill in the following fields:

- **Project Name** – The name of the project.

- **Build Target** – The targeted version of Android you want to develop for. Bear in mind that choosing a newer OS release will preclude older devices and thus limit the audience which will make for fewer downloads. In addition, the lower versions may not offer all of the same libraries as those with more recent releases.

- **Application Name** – This is used for aesthetics, it can be whatever you want. Generally it should match the project name.

- **Package Name** – The root namespace of the project. You have to put something in here as it will be examined when you deploy to the marketplace.

- **Activity** – An activity is the container for an Android application. Applications are made up of multiple activities, each with a responsibility. The name can be whatever you like, but it should describe your application. *The checkbox next to the field label asks whether Eclipse should create the Activity for you; make sure this box is checked.*

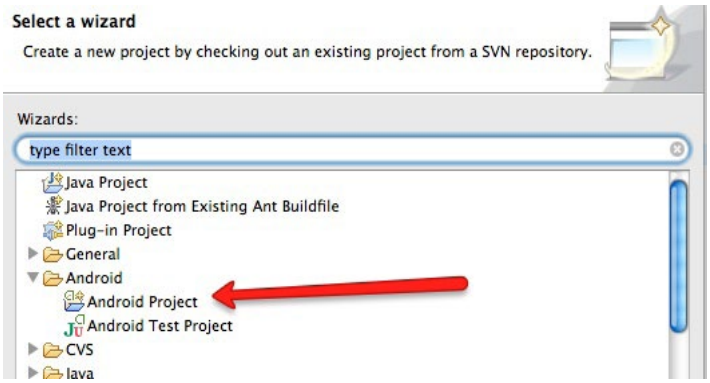


Figure 4

Hitting next will bring up another dialog requesting whether to create a test project or not. This step is optional, for our purposes we will not use a test project.

Once you hit finish, you can hit the play button within the IDE and run your application. Note that the first time you run the application the emulator will have to boot its copy of Android, so be prepared to wait. For subsequent runs, it is recommended you do NOT close the emulator, so as to speed development.

Figure 5

Once Android has booted, Eclipse will automatically start your root activity and you will see a variation of the common Hello World test application in the emulator window (shown in Figure 5). Congratulations, you have successfully created your first Android app. Now let's explore the various pieces.

If you look at Eclipse, you notice the project has a host of folders and files that were created for you.

The *res* (or resource) folder contains references to your application's resources. These items are surfaced via the generated **R** class; the source of which is located in the *gen* folder of the project.

Open the *main.xml* file from the *res/layout* path in the project. Look over this file and take note that the text property for the TextView control, which controls what the user sees, currently is set to a localized string. Change it to your name. The file should look like Figure 6. The first thing to understand is a nomenclature difference to those coming from the Windows programming world. In Windows we say *control*, in Android we say view. The two are, for their part, analogous. In our sample we have two views: LinearLayout and TextView, however, we would more commonly refer to LinearLayout as a *ViewGroup*, or a view which can contain multiple views. Every layout MUST have a ViewGroup as its root element.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.
com/apk/res/android"
        android:orientation="vertical" android:layout_
width="fill_parent"
    android:layout_height="fill_parent">
        <TextView android:layout_width="fill_parent"
android:layout_height="wrap_content"
                android:text="Hello, Jason" />
</LinearLayout>
```

Figure 6

The next thing to understand is the layout width and height attributes. For every view (including ViewGroups) these must be defined so as to denote the amount of space the element will take up. For both of these attributes there are three legal values: *fill_parent, wrap_content*, or a fixed width measurement such as 24px or 12dip. Both *wrap_content* and *fill_parent* serve to create relative screen layouts that will respect the size of the screen.

- *wrap_content* – take only as much space as is required by the views contents.

- *fill_parent* – take as much space as is available.

It is important to exercise caution when using *fill_parent*. A common mistake made by beginners is to use it improperly causing elements to not appear on the screen. *fill_parent* will take up as much space as is available to the view at the time it is rendered. This means if you intend to place elements next to each other, the second item will not appear because all of the space has been taken by the first. The key is to carefully consider the layout you are creating and take notice of the order in which you are specifying views.

The Android Manifest

The *AndroidManifest.xml* (located at the project root) file fulfills a very critical purpose: specifying to the Android runtime what your applications needs are to execute. For example, when the user downloads your application off the Android Marketplace they will be asked to confirm the permissions the application needs; these are driven off of the *AndroidManifest.xml file*: a sample is provided in Figure 7.

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/
apk/res/android"
      package="com.example.AndroidContactPick"
      android:versionCode="1"
      android:versionName="1.0">
    <application android:icon="@drawable/icon"
android:label="@string/app_name">
        <activity android:name=".ContactPick"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.
action.MAIN" />
                <category android:name="android.intent.
category.LAUNCHER" />
            </intent-filter>
        </activity>
    </application>

        <uses-permission android:name="android.
permission.READ_CONTACTS"/>
        <uses-permission android:name="android.
permission.CALL_PHONE" />
        <uses-permission android:name="android.
permission.CAMERA" />
        <uses-permission android:name="android.
permission.ACCESS_FINE_LOCATION" />
</manifest>
```

Figure 7

The upper portion of the file specifies the Activities included in the application or the number of independent screens the application can expect to be present, the lower portion specifies what permissions the application will need to run. In this case we will need the ability to use the phone to make calls, access to the contact list, access to the camera, and access to the GPS unit.

Views and Activities

We've seen how Android separates out the view into layouts and surfaces them through the res folder. But a display we cannot interact with programmatically will not win us any fans on the Marketplace; we need to be able to talk to our views on the layout. To do this, we associate a layout with an Activity and then reference the views in the layout. The Activity source can be found in the src folder. The source I am using is shown in Figure 8.

```
package com.example.HelloWorld;

import android.app.Activity;
import android.os.Bundle;

public class HelloWorld extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);
    }
}
```

Figure 8

The *onCreate* method in your Activity class is called as soon as the Activity is ready to execute. The call to the parent *onCreate* method is required; an exception will be raised if you fail to perform this action. Next, the call to *setContentView* will associate an Activity with a layout; be sure to call this before you attempt to interact with the layout.

The parameter to *setContentView* is *R.layout.main*. This reference comes from our *layout* file being surfaced through the inner class *layout* located in the generated *R* class. This happens because *layout* is a special folder that *res* expects There are many of these special folders, all of which will cause an inner class to be generated within the *R* class by the name of the folder. Finally, *main* is the name of the item, minus its extension.

Once *setContentView* is called you can use the built-in function *findViewById* to access and modify the views in your layout. To do this, we will need to make one change to our layout, note the updated code in Figure 9.

```
package com.example.HelloWorld;

import android.app.Activity;
import android.os.Bundle;
import android.widget.TextView;

public class HelloWorld extends Activity {
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.main);

        TextView myTextView = (TextView)findViewById(R.
id.lblText);
        myTextView.setText("This is a Test");
    }
}
```

Figure 9

By adding the *id* attribute to our *TextView* we gain the ability to identify and reference the view instance, through the use of *R.id.lblText*. Again we surface a value through the *R* class via an inner class, *id* in this case. But we didn't create a folder, so how did Eclipse know to create the inner class? The answer lies in the syntax we used to define the id in the main.xml file: @+id/lblText (Figure 10). By utilizing the '+' in this code we tell Android we intend to interact with this view from code, Eclipse in turn will automatically update our R class with the appropriate value; yes values can be duplicated and reused across layouts.

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.
com/apk/res/android"
        android:orientation="vertical" android:layout_
width="fill_parent"
    android:layout_height="fill_parent">
        <TextView android:layout_width="fill_parent"
android:layout_height="wrap_content"
                android:text="Hello, Jason"
android:id="@+id/lblText" />
</LinearLayout>
```

Figure 10

At this point, run your application and you should see the emulator once again, this time with the defined string in the viewport window.

Closing Remarks

Apple's iPhone really took the entire smartphone market by storm and raised the bar for all manufacturers; a bar that many could not reach for much of the time the iPhone has been out. This is changing. Taking the lead from the iPhone, other manufacturers shifted their focus to providing end users with a more enriching experience and thus many have closed the gap with Apple. Leading the charge in this respect is Android, which is predicted to eclipse iPhone and RIM by the end of 2012 (http://bit.ly/biF5Fd). The popularity of mobile devices cannot be underestimated, a study conducted by IBM found that mobile development will surpass all other forms of development by 2015 (http://bit.ly/9Q4CPJ).

With any mobile platform, the developer experience is a huge factor in fueling demand, especially when multiple options are available. It is the developers who create the apps which heavily influence the demand, and if one platform is easier to program against then another, developers will more often choose that platform. This is why the choice of using Java gave Google such a huge advantage; the developer experience with Java and Eclipse already existed and the tools could be easily acquired and used; in contrast to Apple and Objective-C.

We saw it with Microsoft throughout the 90s where the lack of substantial competition led to complacency. Competition is needed to bring out the best in people and companies. The iPhone helped push a stagnant market forward and helped it evolve into what it is now. Now Android is helping to push the market again, to an end that can only beneficial to consumers.

Jason Farrell is a technologist focused on community who specializes in web and mobile application development. Jason has been doing web development professionally for 4 years and has used a variety of tools and frameworks. Jason started programming Android for CodeMash last year and since then has created several presentations and spoken at a variety of events on Android.

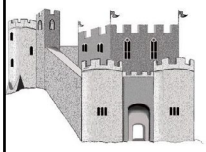
When he is not programming Jason enjoys sports and movies and loves traveling around the world. Recently he spent two weeks backpacking across Japan. He is currently in the process of planning a return trip next year to climb Mt. Fuji.

[1]It is possible to develop Android applications without using Eclipse, but it requires more manual actions be taken. This article will not cover this approach and will assume Eclipse is to be used for development.

A Clojure Fairy Tale

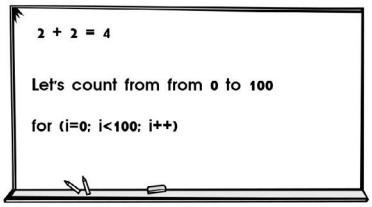
By Carin Meier
Twitter: @carinmeier

Once upon a time, there was a kingdom ruled by a good king and queen. After many years, a princess was born. There was much excitement and rejoicing when the name was announced:



```
king=>(println "Announcing the birth
of Princess Chloe!")

Announcing the birth of Princess Chloe!
```

The princess grew up into a lovely smart child. However, she had a hard time at princess boarding school, where all the other imperative kids made fun of her lisp and called her lazy.




```
princess-chole=> (+ 2 2)
4

princess-chole=>(range 100)
(0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17
```

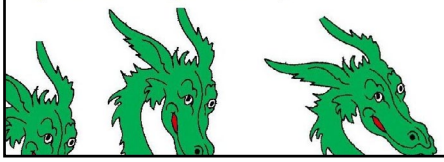
She took solace and strength from the unfailing love of her mother and father

Unfortunately, on the eve of her 18th birthday, they were killed in a tragic scheduling collision of the royal promenade and the wargames of the palace division of Teddy Bear Berserkers.

The evil wizard decided now was the time to strike to destroy the kingdom. He would unleash his most terrible creation... The infinite headed Hydra!

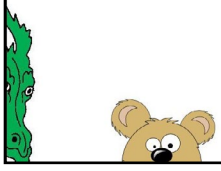
Muhahaha!

```
wizard=> (defn hydra [num-heads]
  (take num-heads (repeat "hydra-head")))

wizard=> (hydra 5)
("hydra-head" "hydra-head" "hydra-head"
"hydra-head" "hydra-head")
```



The kingdom's Berserker Teddy Bears put up a valiant fight, but the finite troops could not keep up. The kingdom would be destroyed! The princess had to do something ...

```
princess-chloe=>
 (defn defeat-hydra [num-heads]
  (replace {"hydra-head" "flower"} (hydra
num-heads)))
```



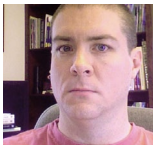
```
princess-chloe=> (defeat-hydra 5)
("flower" "flower" "flower" "flower"
"flower")

princess-chloe=> (defeat-hydra 10)
("flower" "flower" "flower" "flower"
"flower" "flower" "flower" "flower"
"flower" "flower")
```

After a few rounds, the hydra disappeared and the wizard was forced to retreat back to his tower of evil.
Everyone cheered and had tea.

```
narrator=> (def the_end
(repeat "They all lived happily ever after."))
```

** Infinitely, of course...*

Web Development AntiPatterns



By Mike Doel

Rule number 1 in writing an article like this is to make sure you grab your reader's attention right away and draw them in. So, let me just say then that you, dear reader, are looking REALLY good right now. You been working out? Lost a few pounds? Yeah? It shows - all this hard work you've been doing. There's just one thing. I know that sleeveless shirt you put on probably looked good in the mirror - it does indeed show off your new and improved guns quite splendidly. The only thing is that we're going to be outside shoveling snow, and well, you might come to regret that decision sooner or later. Perhaps this coat might be of assistance.

You now know what an AntiPattern is. It's not just a simple error born out of carelessness. Those types of mistakes are typically found quickly and rectified. An AntiPattern is something much more dangerous. It is a solution to a problem that initially appears to bring benefit but reveals itself with further inspection and experience to be a source of trouble. Furthermore, to be a true AntiPattern, it needs to be accompanied by a refactored solution that is better suited for the situation.

Writing about AntiPatterns in web development is something of a challenge. You see, when you stop to think about it, the number of different stumbling blocks that developers encounter is overwhelming. How could I possibly do justice to them all in a single article? The short answer is I couldn't. And so, in the spirit of sunlight being the best disinfectant, I'm going to write about the errors I've made (and continue to make) in the hope that doing so will help serve as a reminder to both of us to avoid these pitfalls in the future. So, with that as prolog, let's dive in and take a look at a short survey of six AntiPatterns - two each from the most common web architecture in use - the Model View Controller (MVC) framework.

The examples in this piece use Ruby and Rails and come directly from my project - www.resortgems.com, a site that allows people planning a vacation to rent timeshare properties directly from owners who are not interested in using their timeshare that year. Because Ruby is a fairly expressive language, the examples should be readable even if you are not a practitioner of the language. But rest assured, the AntiPatterns and refactored solutions demonstrated here are not unique to that platform. Look carefully and you might even recognize some of your own code.

Model AntiPattern 1 – The God Class

Our first AntiPattern is the God Class. Put simply, a God Class is one that is omniscient - it knows about everything, or nearly everything in the system. Consider the code in Listing 1. It is not uncommon at all to find yourself working on a project with one or two classes like this that have bloated beyond all sense of reason. God classes violate several of the basic tenets of good object oriented design, including the Single Responsibility Principle1 which states:

> **There should never be more than one reason for a class to change**

```
class Resort
       def self.grouped_by_state .... end
       def self.grouped_by_country ... end
       def address= ... end
       def available_listings ... end
       def recently_rented_listings ... end
       def units_matching(bedrooms,sleeps,view)
... end
       def allows_check_in_on?(day) ... end
       def check_in_days ... end
       def multi_branded? ... end
       def amenities ... end
       ...
end
```

Listing 1

In this case, the Resort class needed to change to reflect new requirements in geography, ownership, branding, and more. Of course, nobody sets out to create objects this way. Like most of the AntiPatterns we'll cover here, this is one that comes about as a result of skipping one of the key steps in the development process - refactoring. It's not enough to just make your code work and pass your tests (assuming you've written some). Once the initial code is written and working according to specification, stop and consider how

the code could be written better.
The solution to the God Class problem is to refactor it into multiple classes or modules - one for each responsibility that the class started out with (see Listing 2). The benefits of doing so should be fairly obvious. The resulting code is more readable, easier to test, and easier to maintain.

```
# resort.rb
class Resort
       include Locatable
       include Rentable
       ...
end

# locatable.rb
module Locatable
       def self.included(base)
              base.extend(ClassMethods)
       end
       module ClassMethods
              def grouped_by_state ... end
              def grouped_by_country ... end
       end
       def address= ... end
end

#rentable.rb
module Rentable
       def available_listings ... end
       def recently_rented_listings ... end
end
```

Listing 2

There are several clues to look for that can tip you off to God Classes:

- a class with many accessor methods
- a class with methods that operate over only a subset of the instance variables
- a class that churns frequently in your source code control system

Model AntiPattern 2 – Database Abuse

By now, there's a good chance that you've left behind the practice of doing direct database access and instead started to use an ORM (Object Relational Mapping) layer like Hibernate, ADO.net, or ActiveRecord. And why not? The advantages in abstraction that come with these components no doubt make you much more productive than you were without them. But, there's no free lunch. In particular, it's fairly easy to write code that produces the correct results, but does so in a highly inefficient manner. Consider the implementation of the `largest_listing` method in Listing 3. If you're not familiar with Ruby/Rails, the largest_listing method uses the associations built into the `ActiveRecord` component to first find all `Listings` associated with the Resort and then takes the result and sorts them by bedroom count. Because the number of bedrooms is in the `Unit` record associated with the `Listing` (i.e. and not in the `Listing` record itself), a new query is needed for each listing to find its bedroom count.

```
class Listing
       belongs_to :unit
       delegate :bedrooms, :to => :unit
end

class Resort
       has_many :listings

       def largest_listing
              listings.sort_by(&:bedrooms).last
       end
end
```

Listing 3

This fragment demonstrates an "N+1" problem - so named because of the number of queries it generates - one to get the set of rental listings associated with a `Resort` and one for each Unit associated with the `Listing`. Compare that with the implementation in Listing 4 which uses a technique known as "eager loading" to reduce the number of queries to two. In this case, a total of two queries is needed. The first retrieves all of the `Listings` associated with the `Resort`. The second gets all of the `Unit` records associated with any of those `Listings`. The performance improvement can be profound in cases where the number of associated rows grows rapidly. But it comes at a price - broken encapsulation. Now, the `Resort` class has direct knowledge that `Listing` uses the `Unit` class to keep track of how many bedrooms are associated with it. We've just taken a step towards making our `Resort` class a God Class.

```
def largest_listing
       listings.includes(:unit).sort_
by(&:bedrooms).last
end
```

Listing 4

Can we avoid the database abuse without breaking encapsulation? As it turns out, we can as shown in Listing 5. The eager loading is pushed back into the `Listing` class where it belongs. Beyond being better encapsulated, this has enabled yet another efficiency gain - putting the

responsibility of ordering the rows on the database instead of in far less efficient application code.

```
class Listing
       belongs_to :unit
       delegate :bedrooms, :to => :unit
       named_scope :sorted_by_size, {
        :includes => :unit,
        :order => ‚Äòunit.bedrooms,'
       }
end

class Resort
       def largest_listing
              listings.sorted_by_size.last
       end
end
```

Listing 5

The lesson here is two-fold. First, even though an ORM makes it easy to blithely ignore the mechanics behind the construction of SQL statements, you still need to be aware of how the ORM layer executes the query so that you can avoid killing your database. And second, don't replace one AntiPattern with another when refactoring!

View AntiPattern 1 – Logic in View Templates

These days, most web applications are developed using the Model View Controller (MVC) paradigm. Many of the primary web development frameworks in place (Ruby on Rails, Django, Spring MVC, etc.) have this pattern as their foundational architecture. But, it takes discipline and skill to apply the pattern appropriately. One of the fundamental mistakes frequently made is to put logic in the wrong places.

In a properly constructed MVC application, the view layer should be home only to concerns of presentation. And yet, it's not uncommon at all to find other types of logic intermixed with the view templates. In Listing 6, part of a form where an administrator conListings a resort object to have one or more secondary brands, the set of brands to use is queried directly from the second line of the example. Such logic belongs in the Controller, not the View.

```
%span.label Secondary Brands:
- Brand.all.each do |brand|
  %label
    = check_box_tag 'resort[brand_ids][]',
brand.id,
@resort.secondary_brands.include?(brand)
    = brand.name
```

Listing 6

Listing 7 is yet another example of the AntiPattern. In this case, we are looking at a part of a form where a timeshare owner is describing the attributes (bedrooms, bathrooms, view, etc.) of the unit they want to list for rental. The code snippet includes a branch with a non-trivial condition to determine how the form should render the choices the user will make. While this logic is rightfully part of the presentation layer, it really doesn't belong in the View template itself. A better choice is to put code like this in a View helper that is referenced by the View as in Listing 8.

```
.section
  = render 'resorts/thumbnail_view'
  - if @manually_specify_unit || @listing.
resort_pending?
    = render 'manually_specify_unit'
  - else
    = render 'choose_unit_from_dropdown'
```

Listing 7

```
# Helper code ...
def choices_manually_specified?
       @manually_specify_unit || @listing.resort_
pending?
end


# Template code ...
  .section
    = render 'resorts/thumbnail_view'
    - if choices_manually_specified?
      = render 'manually_specify_unit'
    - else
      = render 'choose_unit_from_dropdown'
```

Listing 8

View AntiPattern 2 – The Golden Hammer

There's an old adage that says "when all you have is a hammer, every problem looks like a nail". This same issue can affect your code as well. Too often, we fall back on the familiar techniques and develop solutions that are similar to ways we've done them in the past. This can be especially true when you are new to a language or framework.

Consider the two choices for sorting collections shown in Listing 9. Shortly after I learned Ruby, I learned how to sort collections using the first technique. Having learned sorting, I moved on to learn other things. And while there's nothing incorrect in the logic, it's clearly more verbose than the second version. I had a hammer that allowed me to sort collections one way and used that hammer to turn every

sorting problem into a nail.

```
def least_expensive_listing_1
       @listings.sort{|x,y| x.price <=> y.price
}.first
end

def least_expensive_listing_2
       @listings.sort_by(&:price).first
end
```

Listing 9

If trying to grasp all of the idiomatic solutions to a new language is difficult, the problem is even more challenging when a large web application framework is brought into the picture. Whether its Rails or various parts of J2EE or ASP.NET, the framework you're using to help you build your application is no doubt very large and full featured. Make sure to learn its capabilities - especially with respect to the View layer where the framework can greatly simplify your work and help encourage best practices (e.g. avoiding SQL injection, cross-site scripting, and the like).

Controller AntiPattern 1 – The Fat Controller

Like it's cousin AntiPattern above (Logic in View Templates), the Fat Controller AntiPattern is another case where logic that should be in one place (the Model layer) is instead included in the Controller layer. How often have you found yourself writing a controller similar to the one in Listing 10? The controller action, used to see recent signups, is the one setting the rules for what is considered a recent signup. Such logic belongs in the model as shown in Listing 11. Doing it this way yields several benefits including easier testing of domain logic and the ability to reuse the definition of "recent" in other contexts.

```
def show_recent_signups
       @users = User.find(:conditions =>
                [,'created_at > (?),', 7.days.
ago],
      :order => ,'created_at DESC,',
      :limit => 100)
       render :index
end
```

Listing 10

```
# controller
def show_recent_signups
       @users = User.recent
       render :index
end


# model
class User
       named_scope :recent, lambda {
    :conditions => [,'created_at > (?),',
7.days.ago],
    :order => :created_at,
    :limit => 100)
       }
end
```

Listing 11

Another common instance of this AntiPattern is to send mail directly from within the controller action as is done in Listing 12. Beyond clouding up the controller with things that need not concern it, consider also what happens if there is a problem with the mail gateway. You never want to leave your web user waiting for off-host connections to take place. The refactored solution in this case is to move the mail sending logic into the model layer, but not the model directly, as a two step process shown in Listing 13:

1. An Observer object tied to the creation of the Rental object creates a marker that indicates that the mail needs to be sent. In this case, we're using the `Delayed::Job gem to send_later` a message to the observer object that will send the mail.

2. A separate process or thread watches for those markers and independently sends the mail (in this case through the `Observer`).

```
def create
       @rental = Rental.new(params[:rental])
       @rental.owner = current_user
       if @rental.save
              Mailer.send_rental_confirmation(@rental)
              Mailer.send_staff_alert_on_rental(@rental)
              redirect_to @rental
       else
              render :new
       end

end
```

Listing 12

```
def create
       @rental = Rental.new(params[:rental])
       @rental.owner = current_user
       if @rental.save
              redirect_to @rental
       else
              render :new
```

```
        end
end

class RentalObserver < ActiveRecord::Observer
        def after_create(rental)
                send_later(deliver_mail,rental)
        end

        def deliver_mail(rental)
                Mailer.send_rental_confirmation(rental)
                Mailer.send_staff_alert_on_
rental(rental)
        end
end
```

Listing 13

Controller AntiPattern 2 – The Boat Anchor

One more AntiPattern for the road. Or perhaps, one more for the water park in this case. Many people are familiar with the YAGNI acronym. If you're not, it stands for "You Ain't Gonna Need It" and it is advice that suggests never to build anything until you absolutely have no other option. Much of software complexity comes from people over-engineering solutions that scratch the "we might need this some day" itch.

YAGNI has a related cousin - YDNIA, or "You Don't Need It Anymore". Everyone has had the experience of working with customers who change their mind about requirements, sometimes only to reverse course and go back to what they initially asked for. If it happens too often, you might develop a defense mechanism of leaving old algorithms around (perhaps commented out) to make it easier to reverse course if/when those changes come. This is what's called a boat anchor - dead weight you keep dragging around in case you might need it. YDNIA. And even if you do, that's what source code control systems are for. So, get in there and exorcise that dead code that nobody is using and pick up speed in the process. And yes, that goes for the commented out code too.

Where to Find More

This article really just begins to skim the surface of AntiPatterns. While there is no definitive reference to AntiPatterns to match the Gang of Four book for patterns, there are several sources you can consult to learn more. Ward Cunningham has collected a large catalog of AntiPatterns and documented them on his site at: http://c2.com/cgi/wiki?AntiPatternsCatalog

In addition, a nice description of several generalized AntiPatterns is available at:

http://sourcemaking.com/antipatterns/software-development-antipatterns

Finally, there are several books on the topic. The initial effort in the field is *AntiPatterns Refactoring Software, Architectures, and Projects in Crisis* by William Brown et. al. And for those who have appreciated the Ruby nature of the example code here, be sure to check out the new *Rails AntiPatterns: Best Practice Ruby on Rails Refactoring* by Chad Pytel and Tammer Saleh.

Mike Doel is a co-founder and chief technical officer of VacationView - the company behind www.resortgems.com. He has a wife and two kids who are very much looking forward to the water park and he's not afraid to admit his coding mistakes in public.

Why Ruby on Rails?

By Matt Darby

CodeMash has established itself as a great place for programmers of different worlds to meet up and share ideas (and drinks!). My short plea for attention is to promote the glory that is Ruby on Rails.

It sometimes seems that even after five years since Ruby on Rails' creation, it's still treated like the new kid on the block and doesn't garner the full respect it deserves from those that use older, more established languages or frameworks. Yeah, I'm looking right at you Java.

Why did I choose to specialize in Rails? Why did I decide to base (and bet) my consultancy, Protected Method on it? Rails solves difficult, repetitive problems elegantly and reliably. This is due in large part to the elegance of the Ruby language. If you haven't looked into Ruby yet, I implore you to implement a blog in Rails. It will change the way you look at programming—it just might become fun again. You'll be reminded why you chose this career. Ruby lets you solve problems instead of forcing you to wrangle your language into submission. Matt Yoho has an amazing article in this magazine (*Ruby, Unix, and Open Source Philosophy*) that highlights some great examples.

I wrote a blog post a few years back when I "got it" (http://www.matt-darby.com/why_rails). What I wrote then still holds true:

So what does Ruby on Rails 'feel' like? It feels like cheating. It feels like starting a marathon and then taking a taxi to the finish line. With Rails I would honestly say that I'm coding at 5% of my normal workload. Much like the 1950's promise that technology would slow down our lives, the actual consequence is that I have more time to do better work. I'm not just filling 5% of my development time, I've freed up 95% to conquer other aspects of coding, such as Behavior Driven Development, requirements analysis, and web design.

A Look at the Azure Labs

By Michael S. Collier
http://www.michaelscollier.com/

The Windows Azure Platform is relatively young, with new features being added to the platform every few months. As a way to test out potential new features, Microsoft makes many features available in a "labs" environment. Microsoft currently provides two labs – SQL Azure Labs and Windows Azure AppFabric Labs.

SQL Azure Labs (available at http://www.sqlazurelabs.com) provides an early look into the SQL Azure OData Service and SQL Azure Data Sync CTP1. The SQL Azure OData Service provides a way to enable OData functionality on a SQL Azure instance with the click of a few buttons. SQL Azure Data Sync CTP1 allows for the bi-directional synchronization of SQL Azure databases. SQL Azure Labs also recently provided a look at Project Houston, which was a Silverlight-based tool for managing SQL Azure instances. Of late much of the functionality in Project Houston was rolled into the new Silverlight-based Windows Azure Management Portal.

The Windows Azure AppFabric Labs (available at https://portal.appfabriclabs.com) provides a preview of several exciting features, such as AppFabric Service Bus version 2.0, AppFabric Caching, and AppFabric Access

Control. AppFabric Service Bus version 2.0 offers an early view into the APIs and functionality planned for the next incarnation of the AppFabric Service Bus. AppFabric Caching brings distributed in-memory caching to the cloud, without additional infrastructure complexities. Finally, AppFabric Access Control provides an easy way to bring federated identity management, via providers such as Facebook and Google, to applications without the challenges involved with working with multiple identity providers.

The labs are a great way for users to provide feedback on Microsoft's ideas for the future of the Windows Azure platform. If you have a great idea for SQL Azure or Windows Azure AppFabric, be sure to submit your idea at www.MyGreatSqlAzureIdea.com or www.MyGreatWindowsAzureIdea.com.
For more information, please visit my blog at www.MichaelSCollier.com.

How Does Massage Benefit Programmers?

By Dana Watts, RN, LMT

Whether you work in a corporate cubicle or a home office, you are using tools that affect your well being. Your constant use of devices like your keyboard and mouse can cause Repetitive Strain Injuries (RSIs), which are defined as sprains or small tears, resulting in irritation or inflammation of muscles, tendons, or nerves. Repetitive Strain Injuries occur most often by doing repetitive motions or by holding an awkward position for an extended period of time. Desk jobs, like yours, increase strain and tightness of muscle groups, increase fatigue, and decrease circulation.

Common Symptoms of RSI include numbness, tingling, pain, loss of strength, decreased joint movement and decreased coordination. At first these symptoms can be treated with rest, but soon will persist even after activity is stopped due to progression of damage to the tissues. Specialists may prescribe rest, hot and cold therapy, stretching exercises, Massage Therapy, and in extreme cases, surgery.

Curing RSI is not always easy, which makes prevention the first line of defense. Frequent rest periods are an easy, yet effective, method of prevention. Ideally, persons should rest hands from repetitive motions or stationary positions about every twenty minutes. Typing break software, such as MacBreakZ (http://www.publicspace.net/MacBreakZ/) and RSIGuard (http://www.rsiguard.com/products/for_individ.htm), have interactive reminders to take rest periods and do stretching exercises while working. Ergonomic work spaces are also a very powerful tool, but many employers fail to provide them.

Massage Therapists play an important role in alleviating or preventing symptoms. Massage Therapy is now offered as part of the benefit package through many employers. Massage helps to increase circulation, which brings fresh oxygen and nutrients to your tissues, as well as carrying away waste products. Massage decreases pain and tension in over used muscle groups and also decreases stress hormones which play an active part in muscle tightness.

The next time you find yourself with aches and pains at your desk, remember what massage can do for you!

See these references for related information:
http://www.massagetherapyfoundation.org/public.html
http://www6.miami.edu/touch-research/
http://www.bls.gov/opub/cwc/archive/spring1999brief2.pdf

Scala: Too Hard for the Average Developer

By Dianne Marsh
http://srtsolutions.com/blogs/diannemarsh

Scala has gotten a lot of press in the past few years. Some people love it for its support of functional programming. Others think it represents the next generation for Java development. The Scala 2.8 release makes the language even more modern and powerful, with constructs such as named and typed arguments. But recently, some people have expressed concern that it may be too complex for the average Java developer.

Let's talk about the tradeoffs. If you spend the time to learn Scala, you reduce the amount of boilerplate code that you need to write. You also get to use powerful constructs, such as for expressions and match expressions. And the new collections library offers consistency between collections that significantly reduces the cognitive load. IDE and tooling support has improved as well.

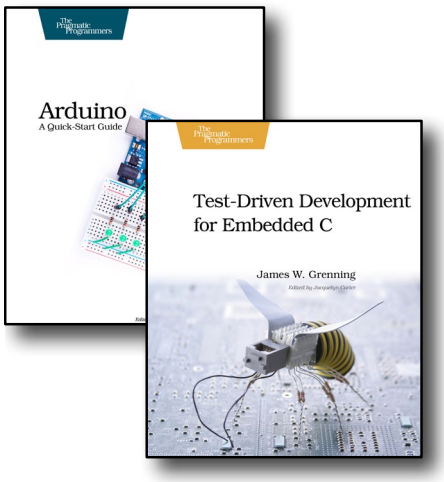
So why is there so much discomfort around Scala?

- It's *different* than Java. Java's longevity means that some professional software developers with a decade of experience have only worked in a single language. They haven't had to make that transition, and perhaps it's a little intimidating. Java isn't simple either; it's just a known entity.
- There are some aspects of Scala that are complex. Implicit conversions are very powerful, and their inclusion helps to support the conciseness of the language. But most Scala developers will not need to use them, and perhaps should be cautioned against doing so. The same is true for C# programmers, who have a similar construct, and have similarly misused it.
- The terseness of the language can appear complex to an untrained eye. This has always been the case in programming languages, and Scala is no exception. I remember the controversy around using the ternary operator, and yet that's become a construct that most programmers now recognize. Coding standards likely exist for your Java team, and it's reasonable to re-evaluate them for any new language.

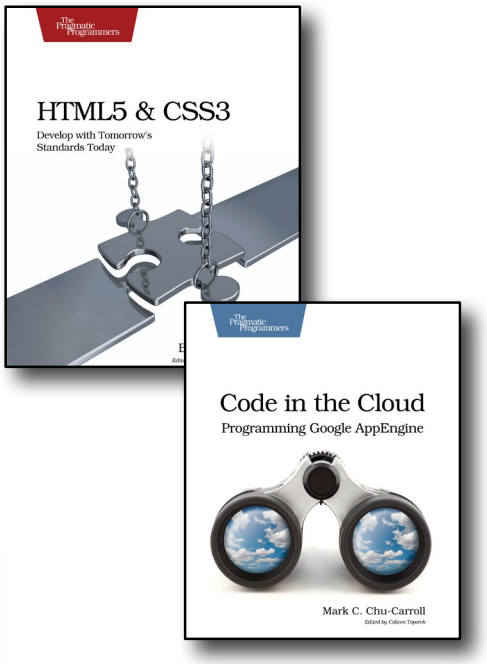
So to the complexity concern, I say, "Nonsense." Like any other language, there is syntax to learn. By adding in functional constructs, Scala also offers a new paradigm. But there's a lot to love. We've chosen a profession that embraces change, and it's not realistic to think that a single language will satisfy our needs throughout our careers. As for the "average developer" litmus test, we'll have to leave that to you to decide!

The Pragmatic Bookshelf

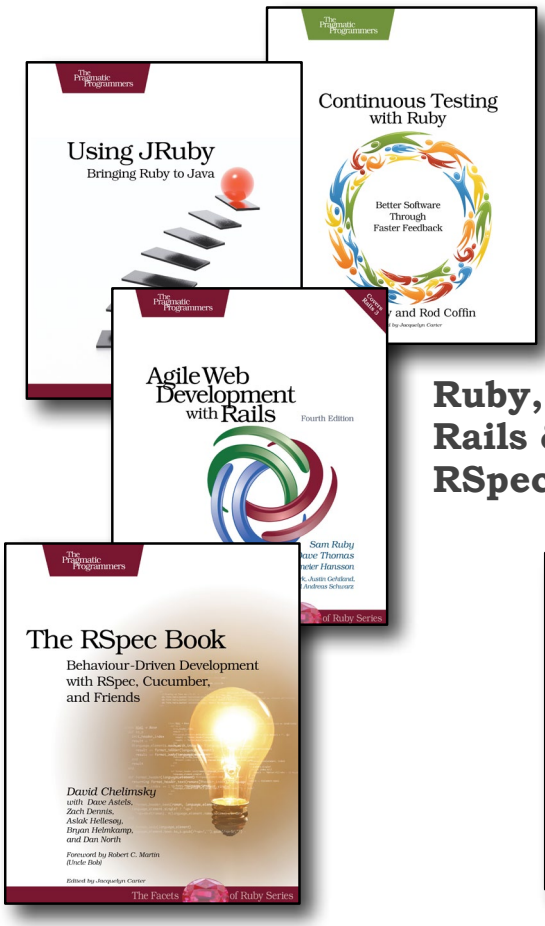
Keeping You At the Top of Your Game.



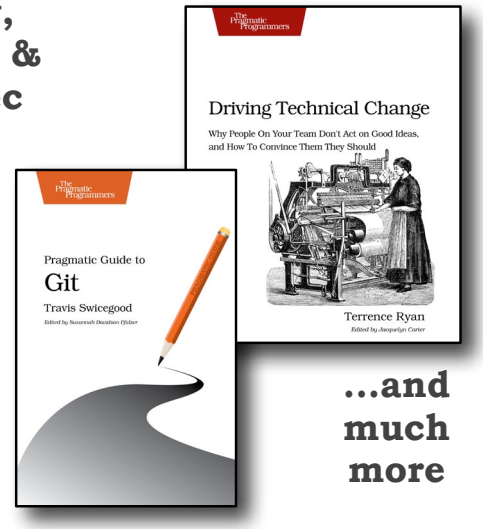
Arduino & Embedded



Beyond Web 2.0



Ruby, Rails & RSpec



...and much more

www.PragProg.com

Pragmatic Bookshelf