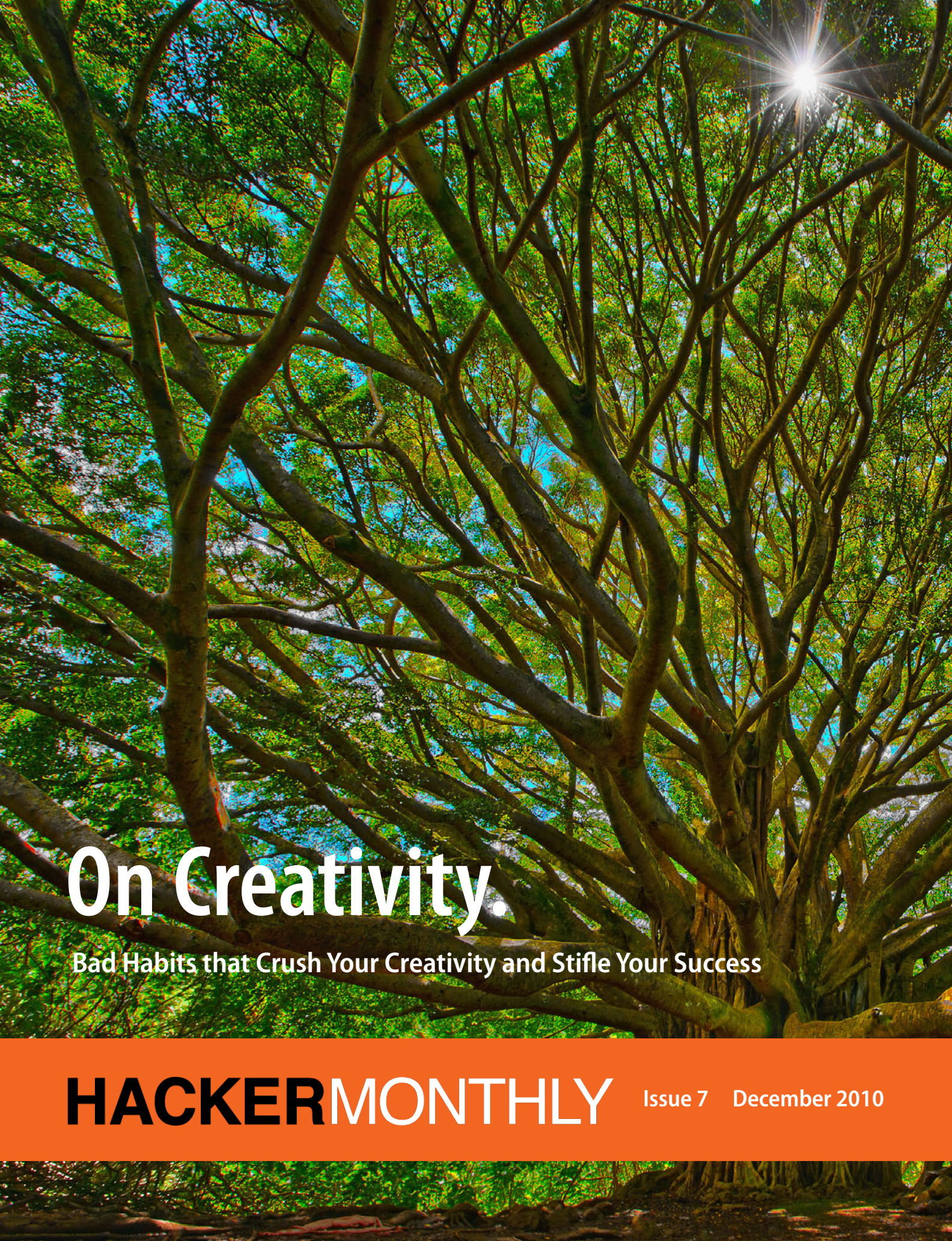


On Creativity.

Bad Habits that Crush Your Creativity and Stifle Your Success

HACKERMONTHLY Issue 7 December 2010

Curator
Lim Cheng Soon

Proofreader
Ricky de Laveaga

Illustrator
Jaime G. Wong

Printer
MagCloud

E-book Conversion
Fifobooks.com

Contributors

ARTICLES
Jake Seliger
Randy Kepple
Dean Rieck
Nathan Marz
Steve Blank
Oliver Reichenstein
Kent Healy
Bradley Wright
Salvatore Sanfilippo
Marijn Haverbeke
Phil Cryer
Paul Querna
Luke Palmer
Fredrik Johansson

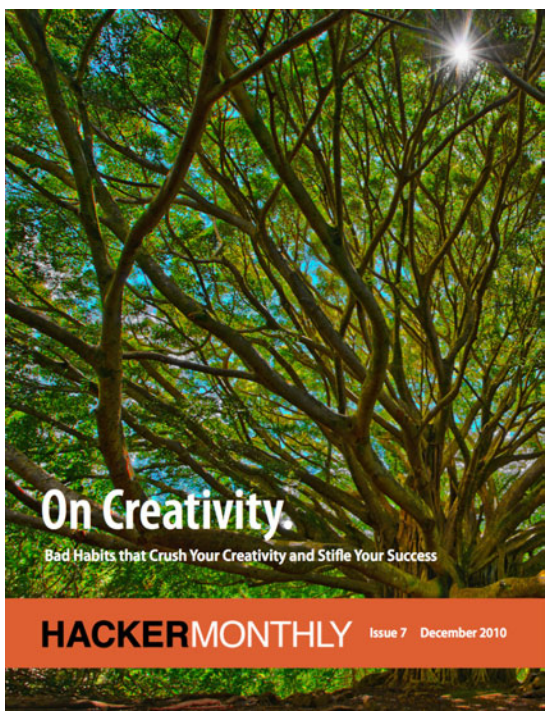
COMMENTARIES
Ed Weissman
Catherine Darrow
Sahil Lavingia
Mahmud Mohamed
Patrick McKenzie
Michael F Booth
Philip Hofstetter
Wes Felter
Juan Pablo
Elben Shira
Leon Paternoster

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

Advertising
ads@hackermonthly.com

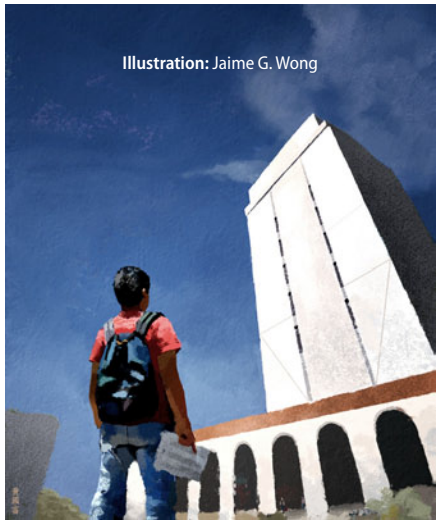
Contact
contact@hackermonthly.com

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



Cover Photo: Randy Kepple (http://randykepple.com)

Contents



FEATURES

STARTUP

SPECIAL

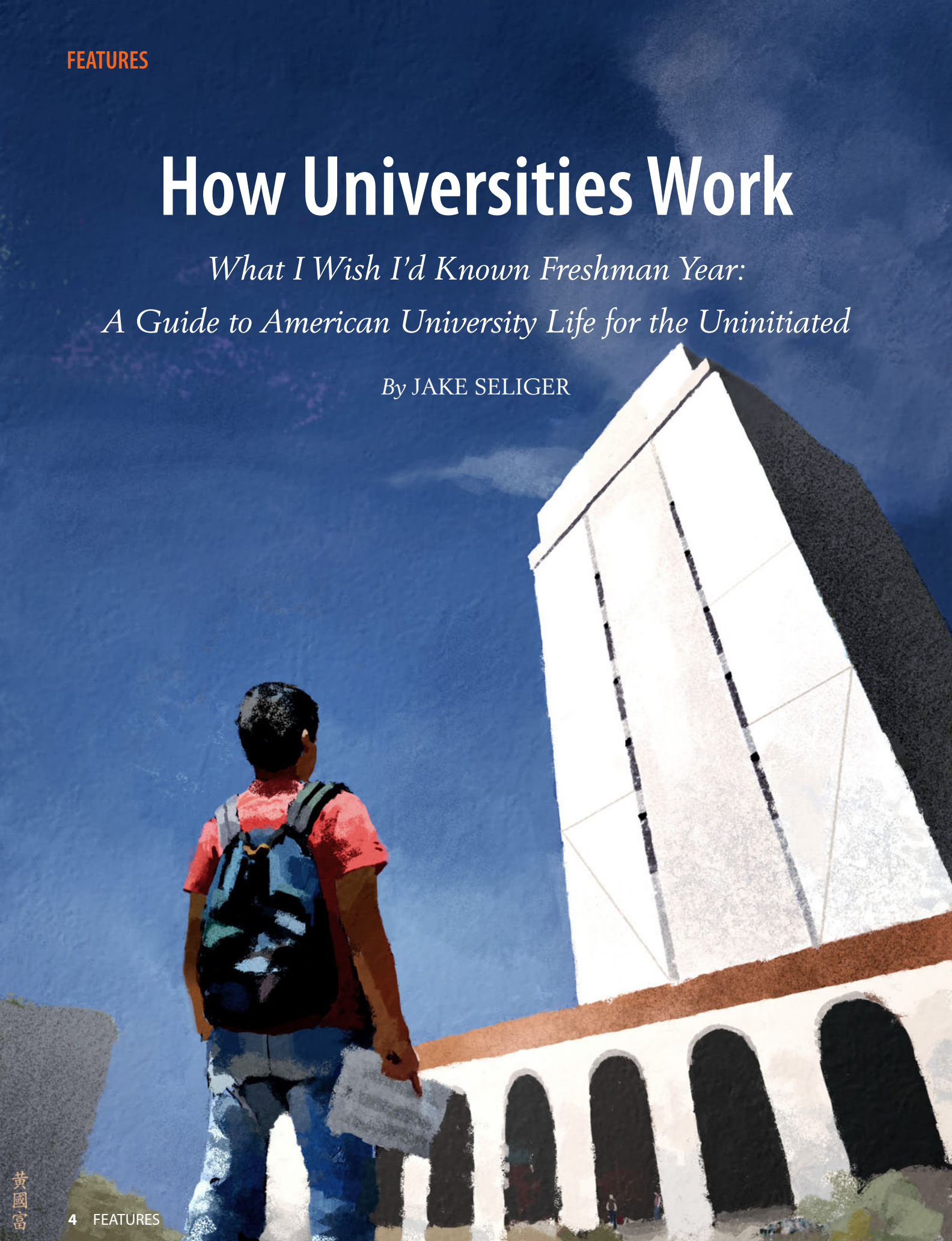
PROGRAMMING

For links to the posts on Hacker News, visit *http://hackermonthly.com/issue-7*. All articles and comments are reprinted with permission of their original author.

How Universities Work

What I Wish I'd Known Freshman Year: A Guide to American University Life for the Uninitiated

By JAKE SELIGER



黃國富

Fellow graduate students sometimes express shock at how little many undergraduates know about the structure and purpose of universities. It's not astonishing to me: I didn't understand the basic facts of academic life or the hierarchies and incentives universities present to faculty and students when I walked into Clark University at age 18. I learned most of what's expressed here through osmosis, implication, inference, discussion with professors, and random reading over seven years. Although most of it seems obvious now, as a freshman I was like a medieval peasant who conceived of the earth as the center of the universe; Copernicus' heliocentric revolution hadn't reached me, and the much more accurate view of the universe discovered by later thinkers wasn't even a glimmer to me. Consequently, I'm writing this document to explain, as clearly and concisely as I can, how universities work and how you, a freshman or sophomore, can thrive in them.

The biggest difference between a university and a high school is that universities are designed to create new knowledge, while high schools are designed to disseminate existing knowledge. That means universities give you far greater autonomy and in turn expect far more from you in terms of intellectual curiosity, personal interest, and maturity.

Degrees

This section might make your eyes glaze over, but it's important for understanding how universities work. If you're a freshman in college, you've probably just received your high school diploma. Congratulations: you're now probably working toward your B.A. (bachelor of arts) or B.S. (bachelor of science), which will probably take four years. If you earn that, you'll have received your undergraduate degree.

From your B.A./B.S., if you wish to, you'll be able to go on to professional degrees like law (J.D.), medicine (M.D.), or business (M.B.A.), or to further academic degrees, which usually come in the form of an M.A., or Master's Degree. An M.A. usually takes one to two years after a B.A. After or concurrently with an M.A., one can pursue a Ph.D., or Doctor of Philosophy degree, which usually takes four to ten years after a B.A.

The M.A. and Ph.D. are known as research degrees, meaning that they are conferred for performing original research on a specific topic (remember: universities exist to create new knowledge). Professional degrees are designed to give their holder the knowledge necessary to be a professional: a lawyer, a doctor, or a business administrator.

Many if not most people who earn Ph.D.s ultimately hope to become a professor, as described in the next section. The goal of someone earning a Ph.D. is essentially to become the foremost expert in a particular and narrow subject.

Professors, Adjuncts, and Graduate Students

There are two to three main groups—one could even call them species—you'll interact with in a university: professors, adjunct professors, and graduate students.

Professors almost always have a Ph.D. Many will have written important books and articles in their field of expertise. They can be divided into two important classes: those with tenure—a word you'll increasingly hear as you move through the university system—and those without. "Tenure," as defined by the New Oxford American Dictionary that comes with Mac OS X 10.6, is "guaranteed permanent employment, esp. as a teacher or professor, after a probationary period." It means that the university can't fire the professor, who in turn has proven him or herself through the publication of those aforementioned books and papers along with a commitment to teaching. This professor will probably spend her career at the university she's presently at.

Those without tenure but hoping to achieve it are on the "tenure track," which means that, sometime between three and six years after they're hired, a committee composed of their peers in the department will, along with university administrators and others, decide whether to offer tenure. Many professors on the tenure track are working feverishly on books and articles meant for publication. Without those publications, they will be denied tenure and fired from their position.

Adjuncts, sometimes called adjunct professors, usually have at least an M.A. and often have a Ph.D. They do not have tenure and are not on the "tenure track" that could lead to tenure. They usually teach more classes than tenured or tenure-track professors, and they also have less job security. Usually, but not always, adjuncts teach lower-level classes. They are not expected to do research as a condition of staying at the university.

Graduate Students (like me, as of this writing) have earned a B.A. or equivalent and are working towards either an M.A. or a Ph.D. From the time they begin, most graduate students will spend another two to eight years in school. They take a set number of small, advanced classes followed by tests and/or the writing of a dissertation, which is an article or book-length project designed to show mastery in their field.

Many—also like me—teach or help teach classes as part of their contract with the university. In my case, I teach two classes most semesters, usually consisting of English 101, 102, or 109 for the University of Arizona. As such, I take and teach classes. In return, the university doesn't charge me tuition and pays me a small stipend. Most graduate students who teach you ultimately want to become professors. To get a job as a professor, they need to show excellence in research—usually by writing articles and/or books—as well as in teaching.

For all three groups, much of their professional lives revolve around tenure, which brings additional job security, income, and prestige.

Illustration: Jaime G. Wong

> **“Most professors are interested in their students to the extent that students are interested in the subject being taught.”**

Two Masters

Most graduate students and non-tenured professors serve two masters: teaching and research. As an undergraduate, you primarily see their teaching side, and your instructors might seem like another version of high school teachers. For some instructors, however, teaching is not their primary duty and interest; rather, they primarily want to conduct original research, which usually takes the form of writing articles (also sometimes called “papers”) and books. The papers you are assigned for many classes in part help you prepare for more advanced writing and research.

Graduate students and professors feel constant tension between their teaching and their research/writing responsibilities. Good ones try to balance the two. For most graduate students and professors, however, published research leads to career advancement, better jobs, and, ultimately, tenure. Many of your instructors will have stronger incentives to work on research than teaching. This doesn’t mean they will shirk teaching, and most teach creatively and diligently, as they should. But it’s nonetheless wise to understand the two masters most of your instructors face.

Interacting with Professors, Adjuncts, and Graduate Students

To earn tenure (or work towards earning tenure), many professors and grad students spend long periods of time intensely studying a subject, most often through reading. They expect you to read the assigned material and have some background in reading more generally; if you don’t, expect a difficult time in universities.

Professors and your other instructors have devoted or are devoting much of their lives to their subjects. As you might imagine, having someone say that they find a subject boring, worthless, or irrelevant often irritates professors, adjuncts, and graduate students, since if those people found their subject boring, worthless, or irrelevant, they wouldn’t have spent or be planning to spend their lives studying it. Most make their subject their lives and vice-versa. They could earn more money in other professions but choose not to pursue those professions, but they are often excited by knowledge itself and want to find others who share that excitement. If you say or imply their classes are worthless, you’ve said or implied that their entire lives are worthless. Most people do not like to think that their lives are worthless.

Professors can sometimes seem aloof or demanding. This is partially due to the demands placed on them (see “Two Masters,” above). Being aloof or demanding doesn’t mean a professor doesn’t like you. Most professors are interested in their students to the extent that students are interested in the subject being taught. In this sense, professors often try to stir students’ interest in a subject, but actively hostile/uninterested students will often find their instructors uninterested in them. Motivated and interested students often inspire the same in their professors.

To be sure, there are exceptions: some professors will be hostile or uninterested regardless of how much effort a student shows, and some will be martyrs who try to reach even the most distant, disgruntled student. But most professors are in the middle, looking for students who are engaged and focusing on those students.

Nearly all your instructors have passed through the trials and tests they’re giving you: if they hadn’t done so, and excelled, they wouldn’t be teaching you. Thus, few are impressed when you allocate time poorly, try to cram before tests, appear hungover in class, and show up late to or miss class repeatedly. On the other hand, many will cut slack for diligent students who show promise.

One reason professors don’t think much of student excuses is because many students have different priorities than professors. As undergraduates, most professors were part of the “academic culture” on campus, to use Murray Sperber’s term; in contrast, many undergraduates are part of the collegiate (interested in the Greek system, parties, and football games) or vocational (interested in job training) cultures. The academic culture, according to Sperber, “[has a] minimal understanding of, and sympathy for, the majority of their undergraduate students” at big public schools. I think he’s too harsh, but the principle is accurate: if you aren’t in school to learn and develop your intellect—and most students in most schools aren’t, as Sperber shows—you probably won’t understand your professors and their motivations. But they will understand yours. Academics are a disproportionately small percentage of the student population at most schools but an extraordinary large proportion of grad students and professors.

> **“Virtually everything you learn in universities and in life is the substance or application of two abilities: math and reading/writing.”**

Requirements for Undergraduates

You can only graduate from a university if you pick a major and fulfill its requirements. Clark called its undergraduate requirements “Perspectives,” while the University of Arizona calls them “Gen Eds” or “General Education Requirements.” There is no way to avoid filling requirements, and most requirements demand that you spend a certain amount of time with your rear end in a seat at a certain number of classes. Fulfill as many requirements as possible as soon as you realize those requirements exist, assuming you want to graduate on time.

You’ll often be assigned an “academic advisor,” whose job it is to help keep you on track to graduate and to help you pick courses. Don’t be afraid of this person: he or she will often help you or point you to people who can help you. At bigger schools, your advisor will often seem harried or uninterested, but even if that person is, you should remember that he or she is still a valuable resource. And if you can’t get help from your counselor, find the requirements of potential majors or all majors and work toward checking them off, because you won’t be able to get out of them.

I tried and found that there is virtually no negotiating with requirements, even if some are or seem silly. For example, Clark required that students take “science perspective.” In studying my schedule and options, I figured that astronomy was the easiest way out. Considering how useless astronomy looked, I decided to petition the Dean of Students to be excused from it so I could take better classes, arguing that I’d taken real science classes in high school and that I could be more productively engaged elsewhere. The answer came quickly: “no.”

Astronomy consisted of tasks like memorizing the lengths of planets from the sun, what the Kuiper Belt is, and the like. Tests asked things like the size of each planet—in other words, to regurgitate facts that one can find in two seconds on Google, which is how I found out what the Kuiper Belt is again. The professor teaching it no longer appeared to have a firm grasp of his mental faculties. At least it was relatively easy: the only worse thing would’ve been having to take, say, chemistry, or a real science class.

That astronomy class was probably the most useless I took, and Clark’s tuition at that time was something like $22,000. I received a scholarship toward tuition, room, and board, so my tuition was probably closer to $16,000, or $8,000 per semester. Undergrads took four classes, so the useless astronomy class cost around $2,000. Would I have rather taken another English class, or Computer Science, or a myriad of other subjects? You bet. But I couldn’t, and if I didn’t take some kind of science class, I wouldn’t have been able to graduate, no matter the uselessness of the class.

What should I major in?

I have a theory that virtually everything you learn in universities and in life is the substance or application of two (or three, depending on how you wish to count) abilities: math and reading/writing. Regardless of what you major in, work on building those two skills.

In the liberal arts, that most often means philosophy, English, and history; other majors vary by university, but those requiring a lot of reading and writing are almost always better than those that don’t. In the hard sciences and economics you’ll be left to develop your reading and writing skills on your own. And this does apply to you, whether you realize it or not. As software company founder and rich guy Joel Spolsky wrote:

> *Even on the small scale, when you look at any programming organization, the programmers with the most power and influence are the ones who can write and speak in English clearly, convincingly, and comfortably. Also it helps to be tall, but you can’t do anything about that.*

The difference between a tolerable programmer and a great programmer is not how many programming languages they know, and it’s not whether they prefer Python or Java. It’s whether they can communicate their ideas. By persuading other people, they get leverage.

So if you want leverage, learn how to write. And if liberal arts majors don’t want to be bamboozled by statistics, they better learn some math.

In short, I have no idea what you should major in. But you probably shouldn’t major in business, communication, sociology, or criminal justice, all of which are worthy subjects that, for most undergraduates, are sufficiently watered down that you’re unlikely to challenge yourself much. Odds are that you’ll even make more money as a philosophy major than a business management major.

> **"Try to learn something about everything and everything about something."**
>
> — Thomas Huxley

Paul Graham wrote:

> *Thomas Huxley said "Try to learn something about everything and everything about something." Most universities aim at this ideal.*
> *But what's everything? To me it means, all that people learn in the course of working honestly on hard problems. All such work tends to be related, in that ideas and techniques from one field can often be transplanted successfully to others. Even others that seem quite distant. For example, I write essays the same way I write software: I sit down and blow out a lame version 1 as fast as I can type, then spend several weeks rewriting it.*

The reality is that your specific major probably doesn't matter nearly as much as your tenacity, ability to learn, and the consistent application of that ability to learn to specific problems. One way people—friends, employers, graduate schools, colleagues, etc.—measure this is by measuring the way you speak and write, which together are a proxy for how much and how deeply you've read.

A great deal of college is about teaching you how to learn, and reading is probably the fastest way to learn. Once you've mastered the art of reading, you'll be set for life, provided you keep exercising the skills you develop at a university. Keep that in mind as you search for majors: those that assign more reading, more writing, and more math are probably more worthwhile than those that don't.

Many people have many opinions about what you should major in, and most of them are probably wrong, this one included. As I said previously, it probably doesn't matter in the long run, so don't worry much about what to major in—worry about finding something you're passionate about and something you love. In Prelude to Mathematics, W.W. Sawyer wrote: "An activity engaged in purely for its consequences, without any pleasure for the activity itself, is likely to be poorly executed" (16 – 17). If possible, find something to major in which you enjoy for itself, or which you can learn to enjoy for itself.

How do I get an A?

One thing you shouldn't do is say that all you want to do is get an A: as stated above, most professors are completely and utterly invested in their subject. When you ask how you get an "A," they're likely to be annoyed because you're indicating you don't care about learning, which is the best way to earn an A. Instead, you care about the badge. It's like asking how you become poet laureate, as Ebenezer Cooke does in The Sot-Weed Factor: the question itself is wrong, because the right question is how you become a poet, and the laureateship will follow (Barth 73). If you ask professors how to get an A, they'll also tell you what you already know: work hard at the class, show up, read the book(s) and related materials, form study groups, and the like.

Another grad student in English said that she's almost relieved when students say they just want to get an A, because it means she doesn't have to worry about them or their grade. Paradoxically, when you say that you just want an A/B/C, you lower the probability that you'll actually get it.

To get that A/B/C, demonstrate that you're interested in the material, do all the reading, and show up to class every day. Go to the professor's office hours to ask intelligent questions—like whether you're on the right track regarding a paper—or what you could've done better on a quiz. By doing so, you're showing that you're interested in doing better, rather than saying you are. Novelists have a saying: "show, don't tell," which means that you should show what a character is thinking and why they are acting in a certain way rather than telling the reader. Readers are smart and will figure it out for themselves. Your professors will be able to figure out in a million ways whether you're interested in a subject, and when you ask how you get an A, they'll know you aren't.

Oh, and don't fear the library—it's the big place with the books. If you conduct research with books, your professors will be impressed. And learn to use the online journals. If you don't know what this means, ask a librarian, who will assist you. They very seldom bite and are there to help, and most schools also conduct library help sessions at the beginning of each year. Indeed, almost everyone at

> **“It took me a long time to express clearly what I was doing, but eventually I realized that one way to deal with a difficult problem is to change the question.”**
>
> — Paul Krugman

a university is there to help you learn; you just need to a) want to learn and b) ask. Many students never get to point a, and of those who do, more should get to point b.

Reflection

I wrote this now because I'm old enough to, I think, have some perspective on universities while still being young enough to remember the shock and bewilderment of the first semester of my freshmen year. This document reflects my academic training and preoccupation: it contains allusions and references to other work and is structured in such a way that you can skip easily from section to section. As a trade-off for its detail, however, weaker or uninterested students might lose interest in it before they come to the end, which is unfortunate because it describes the world they will largely be inhabiting for somewhere between one week and six if not more years.

Anecdotes from my own academic experience are included because discovering facts about the incentives in university life didn't occur all at once for me. No one gave me a document like this: I was expected to either already know or understand most of what you just read, and as a result, I spent years drawing a mental map of universities. The professors and graduate students had spent long enough in the university atmosphere that they knew how universities were structured with the thoroughness you know your native language. I've written this in the hope that it will better explain to you (in the plural sense) what I've explained to many individuals.

My natural impetus is to remember when I have to repeat the same things over and over again, consider how I might convey all the things I've said to a large number of people, and then write those things down so that they might be read, which is a vastly more efficient information transfer mechanism than speech. Nonetheless, I realize that this document and my explanations are probably not perfect, so if you've read this to the best of your ability and still have questions, don't be afraid to ask them. One thing universities should inculcate is inquisitiveness, and I hope I do so as a teacher and as a person.

When you ask questions, you're not only helping yourself discover something: you're helping the person you're asking better understand the subject at hand and the nature of what they're trying to say. By asking me questions about this document, you might help me ultimately improve it, and ultimately help those who read it in the future. If there is one cultural advantage universities should impart more than any other, it is the ability to ask questions about even the most fundamental things; confusion and uncertainty are often the sources of new knowledge.

As Paul Krugman, who won the 2008 Nobel Prize for Economics, said of his own research (which led him to the prize):

> *The models I wrote down that winter and spring were incomplete, if one demanded of them that they specify exactly who produced what. And yet they told meaningful stories. It took me a long time to express clearly what I was doing, but eventually I realized that one way to deal with a difficult problem is to change the question — in particular by shifting levels.*

He also has a section called “question the question,” in which he recursively asks himself whether the question he has asked is the right one. For him, as for many people, questions are at the center of the learning universe, and if you learn to ask them promiscuously and then seek the answers, whether from me, your other professors, or from books, you'll be better equipped to find the answers, do well in college, and do well in life. One challenge is often learning enough to be able to formulate the right questions, and with this in mind, I hope you know how to ask important questions about the institution you're attending. ■

Jake Seliger writes at *http://jseliger.com* and *http://blog.seliger.com*. He's a graduate student in English Literature at the University of Arizona and works as a consultant at Seliger + Associates Grant Writing [www.seliger.com].

Reprinted with permission of the original author.
First appeared in *http://hn.my/university/*.

Google tracks you. We don't.

Bad Habits that Crush Your Creativity and Stifle Your Success

By RANDY KEPPLE *and* DEAN RIECK

> **"There are no days in life so memorable as those which vibrated to some stroke of the imagination."**
>
> — Ralph Waldo Emerson

STANDING IN FRONT of this massive banyan tree on Maui, I was inspired to try something new. I had a vision in my mind's eye of this tree. And Maui will definitely vibrate the imagination of a creative artist. But such vibrational creativity can be elusive more often than not.

Artistry and creativity are two words that work hand in hand. Artistry is defined as an expression of creative skill. Creativity is defined as the creation of artistic work using the imagination or original ideas.

As a artist, the hardest block to overcome is the beginning. Finding inspiration. The imagination can get paralyzed by fear. Trying to create something original. Something that is authentic, yet unique enough to be recognized as original.

I was fortunate to attend a fantastic workshop on Maui this year as a teacher and a participant: The Tropical Island Boot Camp hosted by Randy Jay Braun on Maui. As someone who primarily specializes in photographing people, it was inspiring to push myself to see in new ways and try my hand at expressing my vision in a different way.

At the end of the workshop exploring creative techniques and sharing a life changing experience with my new family, it made me think about the process of creativity, something I actually think about quite a bit. Why is it so hard to break through the barriers of creative block? Could I be doing this to myself? In my ongoing series on the artistic process, I'd like to share with you an article about breaking through developed habits that crush creativity.

Photo: Randy Kepple, *http://randykepple.com.*

It's a myth that only highly intelligent people are creative. In fact, research shows that once you get beyond an I.Q. of about 120, which is just a little above average, intelligence and creativity are not at all related.

That means that even if you're no smarter than most people, you still have the potential to wield amazing creative powers.

So why are so few people highly creative?

Because there are bad habits people learn as they grow up which crush the creative pathways in the brain. And like all bad habits, they can be broken if you are willing to work at it.

Here are eight of the very worst bad habits that could be holding you back every day:

1 Creating and evaluating at the same time

You can't drive a car in first gear and reverse at the same time. Likewise, you shouldn't try to use different types of thinking simultaneously. You'll strip your mental gears.

Creating means generating new ideas, visualizing, looking ahead, considering the possibilities. Evaluating means analyzing and judging, picking apart ideas and sorting them into piles of good and bad, useful and useless.

Most people evaluate too soon and too often, and therefore create less. In order to create more and better ideas, you must separate creation from evaluation, coming up with lots of ideas first, then judging their worth later.

2 The Expert Syndrome

This a big problem in any field where there are lots of gurus who tell you their secrets of success. It's wise to listen, but unwise to follow without question.

Some of the most successful people in the world did what others told them would never work. They knew something about their own idea that even the gurus didn't know.

Every path to success is different.

3 Fear of failure

Most people remember baseball legend Babe Ruth as one of the great hitters of all time, with a career record of 714 home runs. However, he was also a master of the strike out. That's because he always swung for home runs, not singles or doubles. Ruth either succeeded big or failed spectacularly.

No one wants to make mistakes or fail. But if you try too hard to avoid failure, you'll also avoid success.

It has been said that to increase your success rate, you should aim to make more mistakes. In other words, take more chances and you'll succeed more often. Those few really great ideas you come up with will more than compensate for all the dumb mistakes you make.

> **"The brain is a wonderful organ. It starts the moment you get up and doesn't stop until you get into the office."**
>
> — Robert Frost

4 Fear of ambiguity

Most people like things to make sense.

Unfortunately, life is not neat and tidy. There are some things you'll never understand and some problems you'll never solve.

I once had a client who sold a product by direct mail. His order form broke every rule in the book. But it worked better than any other order form he had ever tried.

Why? I don't know.

What I do know is that most great creative ideas emerge from a swirl of chaos. You must develop a part of yourself that is comfortable with mess and confusion. You should become comfortable with things that work even when you don't understand why.

5 Lack of confidence

A certain level of uncertainty accompanies every creative act. A small measure of self-doubt is healthy.

However, you must have confidence in your abilities in order to create and carry out effective solutions to problems.

Much of this comes from experience, but confidence also comes from familiarity with how creativity works.

When you understand that ideas often seem crazy at first, that failure is just a learning experience, and that nothing is impossible, you are on your way to becoming more confident and more creative.

Instead of dividing the world into the possible and impossible, divide it into what you've tried and what you haven't tried. There are a million pathways to success.

6 Discouragement from other people

Even if you have a wide-open mind and the ability to see what's possible, most people around you will not. They will tell you in various and often subtle ways to conform, be sensible, and not rock the boat.

Ignore them. The path to every victory is paved with predictions of failure. And once you have a big win under your belt, all the naysayers will silence their noise and see you for what you are — a creative force to be reckoned with.

7 Being overwhelmed by information

It's called "analysis paralysis," the condition of spending so much time thinking about a problem and cramming your brain with so much information that you lose the ability to act.

It's been said that information is to the brain what food is to the body. True enough. But just as you can overeat, you can also overthink.

Every successful person I've ever met has the ability to know when to stop collecting information and start taking action. Many subscribe to the "ready – fire – aim" philosophy of business success, knowing that acting on a good plan today is better than waiting for a perfect plan tomorrow.

8 Being trapped by false limits

Ask a writer for a great idea, and you'll get a solution that involves words. Ask a designer for a great idea, and you'll get a solution that involves visuals. Ask a blogger for a great idea, and you'll get a solution that involves a blog.

We're all a product of our experience. But the limitations we have are self-imposed. They are false limits. Only when you force yourself to look past what you know and feel comfortable with can you come up with the breakthrough ideas you're looking for.

Be open to anything. Step outside your comfort zone. Consider how those in unrelated areas do what they do. What seems impossible today may seem surprisingly doable tomorrow.

If you recognize some of these problems in yourself, don't fret. In fact, rejoice! Knowing what's holding you back is the first step toward breaking down the barriers of creativity.

Reprinted with permission of original author. First appeared in *http://www.directcreative.com/bad-habits-that-kill-creativity.html.*

I CAN TELL YOU from personal experience that this article is spot on. It's important to let go of old habits of thinking and doing and place yourself in a situation where you can fail. A moment of seeing something and deciding that you are going to challenge yourself to do something different this time.

Creative inspiration came to me standing in front of this massive banyan tree. The lighting was bright and dark all at the same time. Randy Jay Braun is a master at creating HDR (High Dynamic Range) landscape panoramics of Hawaii. Something that was foreign to me. I was inspired to try the HDR technique with this tree. A regular exposure would not be able to capture the incredible dynamic range in this scene. This image is the result of 9 separate exposures combined with HDR Pro in Photoshop CS5.

There is much more to share from the Maui workshop. I was even inspired to really go crazy and create an HDR portrait of Randy Jay Braun. So tell me, which habit do you relate to most from this article? What techniques have you developed to break past self-inflicted barriers to creativity? ■

Randy Kepple is a professional photographer and armchair philosopher based out of the Pacific Northwest. Randy specializes in the art of photographing people. Visit the Randy Kepple Photographs website [http://randykepple.com] for more information on the art and business of image making from Randy Kepple.

Dean Rieck is one of America's top direct marketing copywriters [http://www.directcreative.com/] and author of Dazzle Your Clients and Double Your Income [http://www.procopytips.com/dazzle-your-clients] a free report for writers.

Commentary

By ED WEISSMAN (edw519)

9 Fatigue

Self-help guru Tony Robbins starts every one of his programs covering diet and exercise because he has figured out that if you don't feel well, you probably won't do well.

Of course, I think football coach Vince Lombardi said it best, "Fatigue makes cowards of us all."

By CATHERINE DARROW (Dove)

I CLICKED THROUGH AND wound up disappointed. There's merit to the suggestions, no doubt, but I was hoping for something more like this:

1. *Consuming stupid entertainment*
2. *Staying up too late*
3. *Eating crappy food*
4. *Not ever getting fresh air*
5. *Starving your muse*

. . .

The self-confidence and intellectual exploration stuff I already know. If I didn't, I wouldn't be creative in the first place.

(Expanding on 5. Starving your muse...)

I first heard the expression in an article about resolving writer's block. While that particular article is specific to role-playing games, the idea has more general application and I've heard other writers refer to it.

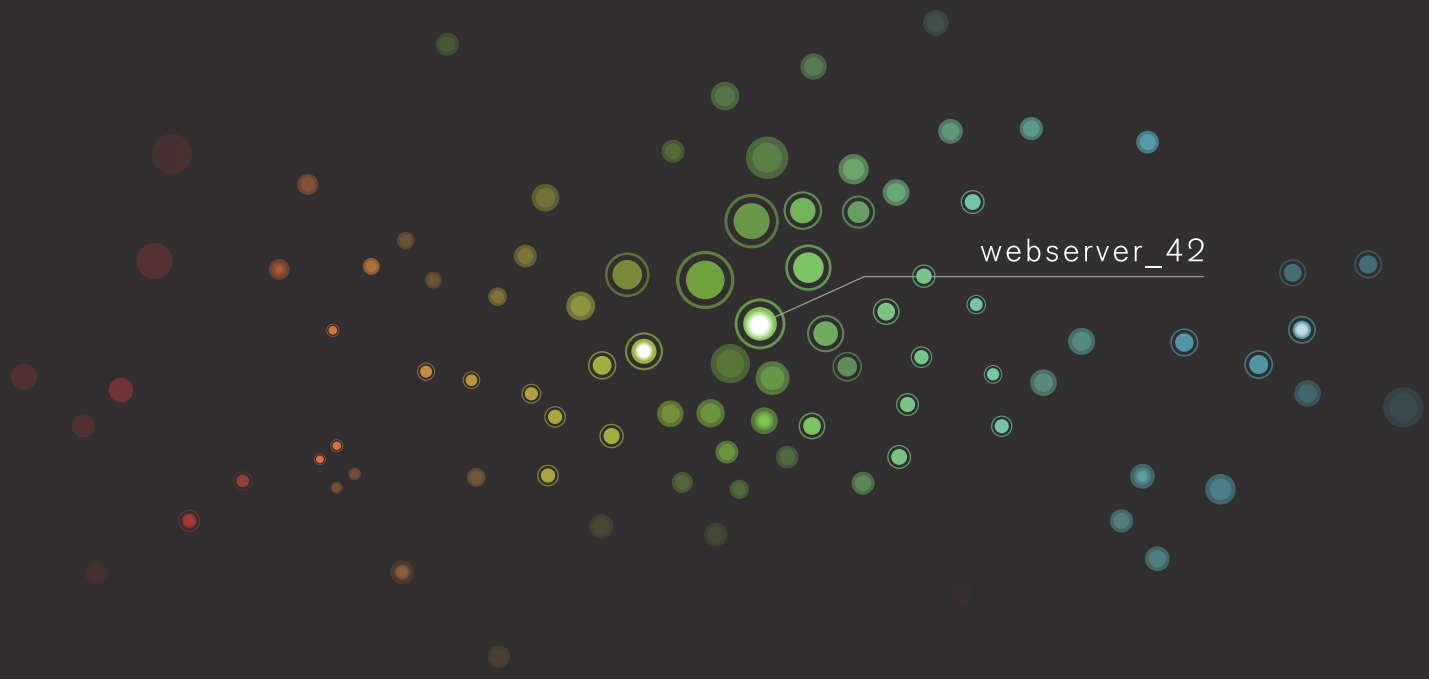
The basic idea is that you can't always be creating. You're never as original as you think you are; your output depends on your input. If you are a storywriter, you need to remember to read for pleasure. Seek new experiences, consume the things that are innovative or interesting or just plain cool in your field of choice. If you keep your muse well-fed on interesting ideas, she'll be ready to provide you with new ideas when you need them.

It applies even in a technical context. Even if you are forced to work in Java or Ada or on a horrific enterprise application, you should be playing in Haskell during your free time, reading papers on interesting algorithms, doing recreational mathematics, that sort of thing. The ideas that will come to you when facing your work are much improved by play.

My own creativity-killing bad habit is to starve my muse. Either to drown myself so completely in the act of creation that I run out of ideas, or to intellectually consume crap rather than good stuff.

These are your servers

These are your servers on Cloudkick



Any questions?

cloudkick.com
415.779.5425

support for 8 clouds + dedicated hardware

cloudkick

the best way to manage the cloud

How to Get a Job At a Kick-Ass Startup

By NATHAN MARZ

When I finished college, I was incredibly naive when it came to finding a great job. I knew that I wanted to work at a small startup but didn't know how to find that great opportunity. I didn't know what questions to ask to evaluate a company, and I didn't know how I should present myself during the recruitment process.

Now I'm a few years out of college and I have that kick-ass job I was looking for. My dual experiences of looking for a job and being on the other side recruiting programmers have taught me quite a bit about what it takes to get a great job at a kick-ass startup.

Here are my tips, from preparing for the job search process to finding great startups to applying and getting the job.

Preparing for the job search

❶ **Make a list of the qualities you're looking for in a job. Be explicit and specific.**
What are you looking for? Coworkers that are really smart that you can learn from? Coworkers that you can socialize with? Flexibility in how/when you work? Write these qualities down.

❷ **Prepare questions that will measure a company against each item in your list.** Stay away from bullshit questions like "What do you dislike about working here?" Bullshit questions will get bullshit answers. Your questions should be specific and help you gauge the company against the qualities you wrote on your list.

For example, if I was curious about how much flexibility employees have to work at home, I would ask:

> *How often do you work at home?*
> *What's the company's policy on working from home?*
> *What would happen if you worked from home for a week?*

When I'm interviewing someone, I like it a lot when the candidate comes prepared with a list of questions. It shows the candidate is on top of things.

❸ **Maximize your personal brand.**
Evaluating a programmer's skill is hard. You need to make it easy for the startup to see that you're a superstar. So make a website and list your side projects there. Link to your Twitter and GitHub accounts. Write some blog posts that showcase your technical ability. You need to develop a personal brand, and you need to do so long before you ever send in a resume. If you don't already have a personal website or blog, make one now. Frankly, developing your personal brand is something you should be doing on a regular basis anyway.

Finding interesting startups

❶ **Look at the portfolio companies of respected investors.**
Let investors filter for you! Go to the website of investors to see their portfolio companies. Looking at the portfolio of seed stage investors like Y Combinator is a great way to find early-stage opportunities.

❷ **Look at the Hacker News threads that list who's hiring.**
This is better than looking at a job board. The companies advertising on the Hacker News threads at least pay attention to the hacker community.

❸ **Let companies find you. Make a public presence. Interact on Hacker News and Twitter. Make or contribute to open source projects. Blog. Make it easy to contact you.**
Hiring is one of the biggest problems at startups. Startups use every channel they can find to source good candidates, including reaching out directly to interesting people they come across. Most of the inbound messages you'll get will be from uninteresting companies, but every now and then an interesting opportunity will come your way.

> “Don’t describe yourself. Instead, describe amazing things you’ve done.”

❹ **Forget recruiters.**
Recruiters tend to be annoying. Plus, a ton of high quality startups refuse to deal with them.

❺ **Invest in your network.**
Your network will lead to interesting and unexpected opportunities. Interact with people on Twitter. Send cold emails to founders of companies and ask if they want to grab coffee. If you’ve made even a minimal investment into your personal brand, founders will be ecstatic to meet you and build a relationship with you.

Evaluating a startup

❶ **The people are much more important than what the company is working on currently.**
An early-stage startup is likely to change the direction of the company at some point. That’s the nature of startups. You should find what they’re working on interesting, but I find that candidates obsess way too much with the product and market of a startup when asking questions.

It’s much more important to focus on the people in the startup. Are they a strong team that executes well? Are they creative? How do they interact with each other? How are decisions made? Would you like to work with these people?

❷ **Observe the working conditions. They reveal a lot about the company’s philosophies towards its employees.**
You’re looking for top notch monitors, chairs, desks, and computers. Look at how much space each programmer has and if the environment is quiet or not.

A top notch work environment is a good investment for maximizing the productivity of programmers and keeping them happy and healthy. Anything less than a top notch work environment is an indication that the company is overly focused on keeping costs low and is cheap with its employees.

❸ **Is the company founded by hackers or business guys?**
Hackers are much more likely to understand what it takes to make a great environment for programmers. Not to say that business guys can’t make a great work environment, it’s just less likely.

❹ **Are they using pressure tactics on you?**
If a company uses pressure tactics on you to get you to accept an offer, it’s a huge red flag. Just imagine how the company will treat you as an employee if they’re willing to manipulate you into accepting their offer.

❺ **Do they move the process forward quickly?**
By “moving the process forward quickly,” I mean answering emails within a few hours. Moving the process forward quickly is a sign that the startup is on top of things.

❻ **Is there hierarchy? Do people give themselves titles?**
This is a big red flag. It’s a sign that the company is filled with big egos or people who think startups are smaller versions of big companies. Startups should be very flat and anyone in the organization should be able to talk to the CEO.

❼ **Do your research on the company. Read the company’s blog. Read the blogs of the employees.**
Startups are a collection of personalities. Do your research and try to figure out if you’d like to work with the people there.

Getting the job

❶ **Don’t describe yourself. Instead, describe amazing things you’ve done.**
The biggest mistake you can make in a cover letter is using an empty phrase like “motivated self-starter.” Believe it or not, everyone describes themselves as an amazing person. Even if you’re amazing, describing yourself as such is meaningless.

Instead, you need to describe amazing things you've done. Focus on problems you've solved as opposed to solutions you've built. You have to be concise and to the point as people have short attention spans when reading cover letters.

❷ **Links, links, links, links, links.**
You'll only get the job if the company is convinced that you'll build amazing things for them. The best way to persuade them of this is to show them amazing things you've built in the past! Links are gold. Link to your open source and side projects.

Remember, it's hard for a startup to evaluate the skill of a programmer. Technical questions can be very inaccurate and incorrectly filter good programmers. So you need to make it easy for the startup to see that you're a superstar, and the best way to do this is to link them to amazing things you've built.

If you don't have any links to show off, you need to remedy that.

❸ **Be yourself.**
Stay away from formal, cookie-cutter cover letters. Do not start off a cover letter with something like "Dear Hiring Manager." Formal cover letters make you sound like a drone, and startups don't hire drones. They hire creative people who get things done.

❹ **Examples are your friends in tech questions.**
When you're stuck on a tech question, work through a few examples. More often than not this will guide you much closer to the solution. I'm shocked at how many people don't use this technique.

❺ **For tech questions, get a correct answer first. Then figure out how to make it faster or simpler.**
A mistake I see a lot of people make is try to get a perfect answer on the first try. A lot of times they're searching for an O(1) solution where none exists. It's better to just get something working first, and then figure out how to optimize or refactor it.

When you get an offer

You have all the leverage when you get an offer. If a kick-ass startup gives you an offer, they consider you to be a rare individual. Negotiate with that in mind.

The best company will give you time to make the best decision for yourself, because they are confident they are a great place to work. They will be aggressive in selling you on the opportunity, but they won't pressure you.

When you accept an offer from that kick-ass startup, congratulations. Get ready for a fun ride. ■

Nathan Marz is a programmer and blogger living in San Francisco. Nathan is the Lead Engineer at BackType and the author of Cascalog, an open-source project for processing data on Hadoop using the Clojure programming language.

Reprinted with permission of the original author.
First appeared in *http://hn.my/startupjob/*.

Commentary

By SAHIL LAVINGIA (sahillavingia)

CAUTION: ONLY WORKS if you're a kick-ass programmer.

Strive to be that and getting a job becomes magnitudes easier. Sweet, successful side projects are the staple of a kick-ass guy. Glad I got some under my belt.

By MAHMUD MOHAMED (mahmud)

NOT JUST KICK-ASS programmers. I have consulted with a lady who ran a small, posh web-shop; she met me at the door and hushed me to tip-toe past a bunch of interns fiddling with photoshop and drupal. Those kids got more respect working for school credit, doing nothing but theming, than most of us get in higher positions with other companies. She also made it a point to "take them to the ATM" on Fridays as well.

OTOH, if you have never seen competent interns with a modicum of responsibility, well, they're a sight to behold. They subcontract for bigger shops and get no credit for their work, but their stuff looks like shrink-wrapped orgasms dipped in pixel-perfect honey. Really awesome crew.

You Negotiate Commodities, But You Seize Opportunities

By STEVE BLANK

It took losing something important to understand the difference between a commodity and an opportunity. Along the way I also learned yet another way entrepreneurs see the world differently from their investors.

Advisory Board

In the early days of Rocket Science I realized that we needed high-level advice on multiple fronts; technology, game development, video game distribution, etc. At one of our initial board meetings we had agreed on the general principle of an advisory board and put together an overall stock budget to compensate advisors.

One of the first potential advisors I reached out to was someone who 10 years earlier tried to hire me as the VP of Marketing of his new division at Sun Microsystems. For lots of reasons that never worked out, but I liked him so much that the following year I tried to hire him as the VP of Engineering of Ardent. (He was having too much fun at Sun and turned me down.)

Now a decade later, we caught up over lunch and I found that he was in the middle of taking a new job inside his company and had some time on his hands. Chatting with him just reinforced my earlier opinion that he was an extraordinary combination of sheer technical talent, great business and common sense and a level-headed decision maker. I knew he would bring immense value to me and the company.

Over the next week we exchanged emails over advisory board stock. I made him an offer and he countered with one I thought was still reasonable (but I didn't tell him that). The timing was perfect, my board meeting was in two days. I could get him the stock he asked for approved at my board meeting and then reply.

Death by Spreadsheet

I was so excited to break the news to the board that I put this new advisor on as the first agenda item. Even back then the advisor was a well-known name in Silicon Valley. The conversation went great and everyone agreed he'd teach us a lot – until one of the board members asked, "How much stock do we have to give him?" I threw out the number of shares I had offered and he had requested, naively thinking everyone would see what a no-brainer this was. Instead what I got was, "Wait a minute. He's asking for one-third of our advisory board stock budget. We had agreed we were going to get 5 to 6 advisors with that amount of stock." At first I wasn't sure I was hearing this correctly. The advisor was a world-class guy, in my judgment he was worth more than all the other advisors I was going to get.

Then the other VC's piled on. "You need to live on the budget we gave you. Go back to him and offer him less stock."

As a first-time CEO getting beaten up my board I thought this wasn't a fight worth having. (I couldn't have been more wrong.) So I agreed to go back to my potential advisor and tell him the best I could do was my first offer.

I was about to get a few lessons that have lasted for a long time.

Thanks But No Thanks

Putting my best marketing spin on it, I sent our potential advisor a message that essentially said, "I'm not sure I can meet your request, but here's another offer." I dressed it up as best as I could, making some of the other terms more palatable, but it still wasn't what he asked for.

> **"If it's one-of-a-kind that give you an advantage, it's an opportunity."**

I guess I shouldn't have been surprised when he sent me a very polite note back that said, "Thanks but no thanks. I'm now getting more involved in my new job as CTO and I'm too busy to go back and forth negotiating this." But I was crushed. I knew my company had just lost something important. Something that I couldn't just go out and replace. And I realized I screwed up in at least two major ways.

You Negotiate Commodities, But You Seize Opportunities

I hadn't just lost a potential advisor, I had lost an irreplaceable opportunity. We didn't lose him just over a stock offer. We lost him because we had treated him as a commodity – something that was readily available from multiple sources, something for which you could negotiate a price.

In reality what I had in front of me was an opportunity - a favorable combination of circumstances that rarely occurs and if seized upon would have given me an advantage.

You treat commodities and opportunities radically differently.

Founding CEO's are supposed to search for a repeatable business model, not just blindly execute their original plan. That requires you to identify opportunities and seize the day. Opportunities are not just about sales, marketing or product. In this case it was about a resource I had in my hands and let go of.

I had acted like an employee, not as a founder and certainly not as the CEO of a startup. I had let my board tell me that the opportunity I saw was a commodity that could be managed by a spreadsheet. And I didn't stand up for what I had believed in.

It would never happen again.

Lessons Learned

- Great entrepreneurs see opportunities before others do.
- Ask, "Is it a commodity or an opportunity?"
- If it's one-of-a-kind that give you an advantage, it's an opportunity.
- Grab opportunities with both hands and don't let go.
- It's better to beg for forgiveness than ask for permission.
- Carpe Diem ■

Steve Blank is a retired serial entrepreneur and the author of Four Steps to the Epiphany [http://www.amazon.com/Four-Steps-Epiphany-Steven-Blank/dp/0976470705]. Today he teaches entrepreneurship to both undergraduate and graduate students at U.C. Berkeley, Stanford University and the Columbia University/Berkeley Joint Executive MBA program. He also blogs about entrepreneurship at *www.steveblank.com*.

Reprinted with permission of the original author.
First appeared in *http://hn.my/opportunities/*.

Commentary

By PATRICK MCKENZIE (patio11)

Over and over in business you'll see people avoid decisions they don't deeply understand (e.g. the average VC knows nothing about gaming, the average PHB knows nothing about databases, the average techie knows nothing about your market), and to paper over their ignorance and demonstrate they are in control and providing value they'll suggest a change to something they think they understand (advisor shares, the design of the front page, your pricing relative to a bowl of ramen).

This rarely works well, particularly when the two decisions are in fact related. One coping mechanism is being able to ignore advice (if you haven't taken their money, you can probably ignore their advice). Another is having a list of knobs people can twirl which are off the critical path (salaryman survival skill #1: distract the boss with rearranging a Gantt chart which can't kill anyone).

Scalable PHP Hosting, made easy with the CatN Platform.

Instantly deploy your apps on our super cluster. No code changes necessary. SSH Access, Cron jobs, Git & SVN support all as standard.

from £5 per month

www.catn.com

Web Design is 95% Typography

By OLIVER REICHENSTEIN

95% of the information on the web is written language. It is only logical to say that a web designer should get good training in the main discipline of shaping written information, in other words: Typography.

Information design is typography

Back in 1969, Emil Ruder, a famous Swiss typographer, wrote on behalf of his contemporary print materials what we could easily say about our contemporary websites:

> *Today we are inundated with such an immense flood of printed matter that the value of the individual work has depreciated, for our harassed contemporaries simply cannot take everything that is printed today. It is the typographer's task to divide up and organize and interpret this mass of printed matter in such a way that the reader will have a good chance of finding what is of interest to him.*

With some imagination (replace print with online) this sounds like the job description of an information designer. It is the information designer's task "to divide up and organize and interpret this mass of printed matter in such a way that the reader will have a good chance of finding what is of interest to him."

Macro-typography (overall text-structure) in contrast to micro typography (detailed aspects of type and spacing) covers many aspects of what we nowadays call "information design." So to speak, information designers nowadays do the job that typographers did 30 years ago:

> *Typography has one plain duty before it and that is to convey information in writing. No argument or consideration can absolve typography from this duty. A printed work which cannot be read becomes a product without purpose.*

Optimizing typography is optimizing readability, accessibility, usability(!), overall graphic balance. Organizing blocks of text and combining them with pictures, isn't that what graphic designers, usability specialists, information architects do? So why is it such a neglected topic?

Too few fonts? Resolution too low?

The main—usually whiny—argument against typographical discipline online is that there are so few fonts available. The second argument is that the screen resolution is too low, which makes it hard to read pixelated or anti-aliased fonts in the first place.

The argument that we do not have enough fonts at our disposition is as good as irrelevant: During the Italian renaissance the typographer had one font to work with, and yet this period produced some of the most beautiful typographical work:

The typographer shouldn't care too much what kind of fonts he has at his disposal. Actually the choice of fonts shouldn't be his major concern. He should use what is available at the time and use it the best he can.

Choosing a typeface is not typography

The second argument is not much better. In the beginning of printing the quality of printed letters was way worse than what we see on the screen nowadays. More importantly, if handled professionally, screen fonts are pretty readable.

Information design is not about the use of good typefaces, it is about the use of good typography. Which is a huge difference. Anyone can use typefaces, some can choose good typefaces, but only few master typography.

Treat text as a user interface

Yes, it is annoying how different browsers and platforms render fonts, and yes, the resolution issue makes it hard to stay focused for more than five minutes. But, well, it is part of a web designer's job to make sure that texts are easy and nice to read on all major browsers and platforms. Correct leading, word and letter spacing, active white space, and dosed use of color help readability. But that's not quite it. A great web designer knows how to work with text not just as content, he treats "text as a user interface." Have a look at Khoi Vinh's website, and you'll probably understand what that means:

Slightly more famous examples of unornamental websites that treat text as interface are: Google, eBay, craigslist, YouTube, Flickr, Digg, reddit, Delicious. Being a hard to dispute necessity, treating text as a user interface is the only parameter for success. Successful websites manage to create a simple interface AND a strong identity at the same time. But that's another subject. ■

Oliver Reichenstein (@iA on Twitter) is an interface designer and founder of Information Architects Inc. He has lived in Japan since 2003.

Reprinted with permission of the original author.
First appeared in *http://hn.my/95typography/*.

Commentary

By ELBEN SHIRA (elbenshira)

WHAT'S UP WITH these typophiles? They walk around thinking they're the most important member of the club.

The truth is, design is complicated because it is not (yet) a science. Typography is important, but it is not king. What we really need to do is get inside the user's head and model their thought process. This is probably impossible, so we should do the next best thing: learn empathy. Be the user. Not some one-trick pony.

By LEON PATERNOSTER (leonpaternoster)

THE CONTEXT OF this statement is important, I think. Back in 2006 web sites were all about Flash, widgets, fancy graphics etc. All Oliver was saying is that these things are unimportant if your text is unreadable.

And by typography he means a lot more than whether it's Georgia or Helvetica.

Why Most People Don't Succeed

— *How You Can Be the Exception*

By KENT HEALY

The burn: Psychological burnout due to overlooking the immeasurable sources of drive.

The diagnosis: If you ignored the fact that your car required an oil change, what would happen? (No, this is not a trick question.)

The vehicle's functions would be utterly undermined leading to complete engine failure. However, it's not just cars that require tune-ups. Ultimately, just about everything requires some extra attention. We wouldn't wash a car once and expect it to be clean forever. We wouldn't go to the gym for one workout and expect to be fit for life. And we certainly wouldn't ingest vitamins once and expect our bodies to be eternally nourished.

This is all common sense.

But why then, are so many people unpleasantly surprised when they feel unsatisfied or don't perform at their full potential? Not surprisingly, like a car, our dirty laundry, or our computer, we too, need tune-ups. But sadly, **it seems to be human nature to wait until something is not working in our lives before we change our priorities.**

Although this concept does not only apply to our physical health, I thought I would share part of a conversation I had recently with a doctor who confirmed this idea. "The big problem I see," he said, "is the number of people who do not consistently maintain their health and ignore the many amber alerts indicating that their behavior needs to change."

The doctor continued, "Most patients look at professional help purely as a last resort; meaning once the pain gets unbearable, they finally come in. Sometimes I can help, but other times, it's God's business at that point. People are not very proactive when it comes to their personal lives. I don't understand it. What wait? Why risk it?"

On some level, most of us expect our personal life to de-frag itself, to watch the wrinkles and flaws simply iron themselves out. We can easily see how this strategy has worked out. It certainly explains the alarming rate of depression, overload, and chronic health problems in society today.

Panic is a strategy for fire stations:

Why then, do we operate our lives like fire stations; passively waiting for disaster to strike before taking reactive measures? Why experience heartache before taking a step back to consider adapting our approach? Here is my three-word-theory: Maintenance is boring. We don't even enjoy taking our car in for a tune-up let alone consistently confronting our own personal baggage.

It is far more pleasurable to pander to our immediate desires. There is also a thrill in creating/doing something new. But the same cannot be said about maintenance.

Maintenance requires discipline, routine, and brutal self-honesty – not words we commonly associate to pleasure. I will be the first to admit the challenge of exercising regularly, adhering to core values, eating healthy, honoring commitments, and engaging in personal reflection and evaluations. It's difficult – as are most things worth doing.

The inordinate reward:

But in every challenge there lies an antithetical reward, an often unintended opportunity. Why? One reason is because the majority opts to avoid confrontation. Thus the obvious consequence is fewer people who follow through with acts of maintenance – the behavior needed to perform at their peak. The not so obvious consequence is the disproportionate reward for the few who do master maintenance.

The reason is simple: Most people simply don't stay in the game long enough to win it. Instead, they run out of steam or choose to settle. Therefore, the abundance that exists is distributed generously to those who do what the majority is simply unwilling to do. I am reminded of a quote from my days in self-help: "Successful people are successful because they are willing to do what unsuccessful people are unwilling to do." So simple. So true.

Reprinted with permission of the original author.
First appeared in *http://hn.my/succeed/*.

Life is not a zero sum game. But stagnation and lazy habits certainly create vivid impressions of lack and deprivation that people mistaken for absolute universal laws. But fortunately, there is enough [enter your definition of success] to go around. (I can hear the pessimist reader cringing: "Enough of 'what-exactly' to go around? Happiness? How do you measure that anyway?" And there in lies a costly misconception…)

The modern metrics dilemma:

While some outcomes of personal maintenance are clearly visible (savings account balance, weight, appearance, sales figures, etc.), many are not. Sometimes to a fault, we place an exorbitant amount of attention on measurable metrics assuming what is most important can be measured.

> *In our dogged pursuit of what is quantifiable we often neglect what is not. Maintenance loses much of its glory due the numerous immeasurable, overlooked, and undervalued rewards.*

Perhaps, Einstein said it best, "Not everything that matters can be measured and not everything that can be measured matters." Without concocting a rigorous study (which most of us will never organize for ourselves), it is difficult to measure personal satisfaction, peace of mind, elation, engagement, etc.

"Big deal. Gimme results!" the pessimist exclaims.

Blinded by outcome, we are quick to overlook the root causes of such outcomes. It's often the immeasurable factors that fuel the behavior required to produce the measurable results. An absence of satisfaction and passion begets results only in the interim. If success is a combination of process, experience, and outcome then sustenance is imperative. But caught up in the modern allure of immediate, quantifiable results, we burn out frequently, quit regularly, and rarely experience notable success.

Long-distance goals cannot be achieved without maintenance (ask any marathon runner). Daily disciplines enable long-term performance and uncommon results. In fact, the very nature of the word "maintenance" embodies a consistent commitment to the long-term… otherwise each action is merely anomalous – and like I've always said, the only difference between "luck" and "skill" is consistency.

Insightful Questions & Actions:

Actions:

- Get honest about your current situation. Rate the following areas of your life on a scale of 1-10: Physical Health, grades, job performance, personal happiness, relationships, financial situation, etc.
- Then follow up with the question: What would it take to make this area a 10?
- Set reminders in your calendar/on your phone to increase consistent follow through.
- Form an accountability partnership with a friend or small group to review and critique progress and process.
- Identify the times you performed at your best and deconstruct the routine that enabled the result. What form of daily maintenance aided your performance?
- Schedule time with yourself away from distractions. (If you can set an appointment with the auto mechanic or your hairdresser, you can schedule an appointment with yourself.) During this time you may wish to address the questions below or create your own. Record your thoughts for future reference.

Questions:

- What top performer/s (athlete, business magnate, etc.) do I admire most? What routines might they use to maintain their edge?
- What are the consequences of neglecting maintenance?
- What unforeseen rewards might stem from a commitment to consistent follow through in the area of ___ [your desired activity]?
- How have I formed new habits in the past? What process works best for me?
- What new routines could I instigate that may ease the process of maintaining constructive behavior?
- At what time should I schedule my next personal tune-up? ■

Kent Healy is an author, speaker, columnist, real estate investor, entrepreneur, a student of life, graphic designer, and an advocate of applied sciences in the realm of personal lifestyle. He blogs at *http://dontgetburnedblog.com/*.

Commentary

By ED WEISSMAN (edw519)

"Maintenance is boring."

Just a few things I don't think are boring:

- sinking my teeth into some fresh sweet melon
- a late afternoon jog in the woods
- hanging out with friends and family
- an ice cold beer at the football game
- an all-you-can-eat salad bar
- curling up with SO (even if it is a chick flick)
- a hot shower, freshly brushed teeth, and a warm bathrobe
- a happy dance after a new program runs the first time

If you think of the things you need to do to live well as "maintenance," they would seem boring, and you won't want to do them.

But if you think of them as "living," you'll embrace them and never give the concept of "maintenance" a second thought.

How to Set Up Your Own Private Git Server on Linux

By BRADLEY WRIGHT

One of the things I'm attempting to achieve this year is simplifying my life somewhat. Given how much of my life revolves around technology, a large part of this will be consolidating the various services I consume (and often pay for). The mention of payment is important, as up until now I've been paying the awesome GitHub for their basic plan.

I don't have many private repositories with them, and all of them are strictly private code (this blog: Amanda's blog templates and styles; and some other bits) which don't require collaborators. For this reason, paying money to GitHub (awesome though they may be) seemed wasteful.

So I decided to move all my private repositories to my own server. This is how I did it.

Set up the server

These instructions were performed on a Debian 5 "Lenny" box, so assume them to be the same on Ubuntu. Substitute the package installation commands as required if you're on an alternative distribution.

First, if you haven't done so already, add your public key to the server:

```
ssh myuser@server.com mkdir .ssh
scp ~/.ssh/id_rsa.pub myuser@server.com:.ssh/
authorized_keys
```

Now we can SSH into our server and install Git:

```
ssh myserver.com
sudo apt-get update
sudo apt-get install git-core
```

…and that's it.

Adding a user

If you intend to share these repositories with any collaborators, at this point you'll either:

- Want to install something like Gitosis (outside the scope of this article); or
- Add a "shared" Git user.

We'll be following the latter option. So, add a Git user:

```
sudo adduser git
```

Now you'll need to add your public key to the Git user's authorized_keys:

```
sudo mkdir /home/git/.ssh
sudo cp ~/.ssh/authorized_keys /home/git/.ssh/
sudo chown -R git:git /home/git/.ssh
sudo chmod 700 !$
sudo chmod 600 /home/git/.ssh/*
```

Now you'll be able to authenticate as the Git user via SSH. Test it out:

```
ssh git@myserver.com
```

Add your repositories

If you're to not share the repositories, and just want to access them for yourself (like I did, since I have no collaborators), you'd do the following as yourself. Otherwise, do it as the Git user we added above.

If using the Git user, log in as them:

```
login git
```

Now we can create our repositories:

```
mkdir myrepo.git
cd !$
git --bare init
```

The last steps creates an empty repository. We're assuming you already have a local repository that you just want to push to a remote server.

Repeat that last step for each remote Git repository you want.

Log out of the server as the remaining operations will be completed on your local machine.

Configure your development machine

First, we add the remotes to your local machine. If you've already defined a remote named origin (for example, if you followed GitHub's instructions), you'll want to delete the remote first:

```
git remote rm origin
```

Now we can add our new remote:

```
git remote add origin git@server.com:myrepo.git
git push origin master
```

And that's it. You'll probably also want to make sure you add a default merge and remote:

```
git config branch.master.remote origin && git config branch.master.
merge refs/heads/master
```

And that's all. Now you can push/pull from origin as much as you like, and it'll be stored remotely on your own myserver.com remote repository.

Bonus points: Make SSH more secure

This has been extensively covered by the excellent Slicehost tutorial, but just to recap:
Edit the SSH config:

```
sudo vi /etc/ssh/sshd_config
```

And change the following values:

```
Port 2207
...
PermitRootLogin no
...
AllowUsers myuser git
...
PasswordAuthentication no
```

Where *2207* is a port of your choosing. Make sure to add this to your Git remote:

```
git remote add origin ssh://git@myserver.com:2207/~/myrepo.git
```

Based in London, Brad is a front end developer and Python hacker at social betting startup Smarkets.

Reprinted with permission of the original author. First appeared in *http://hn.my/privategit/*.

Commentary

By MICHAEL F BOOTH (mechanical_fish)

> *"I decided to move all my private repositories to my own server."*

When you do this, make sure that the server has continuous backups. Also, make sure you still have an offsite backup.

Once you figure out what these things are worth, you may realize that you should probably just keep paying GitHub.

By PHILIP HOFSTETTER (pilif)

The backups aren't as important as each git repo is a full blown clone. If your local repo is destroyed, you still have the server copy. If your server blows up, you still have the local copy.

There are many other good reasons for a service like GitHub, like the excellent collaboration features, the really good repository and history browser or the good bugtracker.

If you don't need those (small team, working alone) but are concerned about uploading your intellectual property to a third party server in a potentially foreign country (depending on your location), then quickly setting up Gitosis / Gitweb / Redmine might be enough for you.

In my personal case, I would really love to use GitHub even for my small team, but I'm too concerned about the legal issues to go ahead with that (and the local installation is plain too expensive).

What's Wrong With 2006 Programming?

By SALVATORE SANFILIPPO

Redis 2.0 introduced a new feature called Virtual Memory. The idea is that some applications using Redis may not access the whole dataset with the same frequency. In extreme cases only a little percentage of hot spot data is used often, while the rest is mostly idle and touched very rarely. For instance imagine a Redis instance holding User objects: the most active users will hit this subset of records continuously, while a large percentage of users will access the site a few times a month, and another large subset of users completely forgot about this web service at all.

Since Redis is memory backed the idea was to transfer rarely accessed data on disk, to reload swapped data when needed (that is when a client will try to access it). The actual implementation of Redis Virtual Memory is completely done in user space: we try to approximate an LRU algorithm, encode data that should be swapped, write it on disk, and reload if needed, decode, managing pages in the swap file, and so forth. It's a non trivial piece of code but it is working well.

Still almost every week I receive a mail, a blog message, a tweet, or I happen to read an article pointing me to this article written by the Varnish guy (edit: that is, the well known developer Poul-Henning Kamp). The article will tell you how silly is to implement your caching layer on top of the one already provided by the operating system. The idea is that you should just write things into an mmap()ed file or alike, and let the OS swap/load things for you.

If you know Redis you already know that we actually try hard to use the operating system smartness to do complex things in a simpler ways. For instance our persistence engine is completely based on fork() copy-on-write semantics of modern kernels, but for Redis Virtual Memory using the OS is not a good solution, and it's time to explain in details why it is not.

OS paging is blocking as hell

The first huge problem with this approach is how badly blocking it is. What happens is that when you try accessing a memory page that is swap on disk the CPU will raise an exception, asking the kernel to retrieve the page from the swap file and transfer it in a physical memory page. In the meantime the process is completely blocked.

What this means? That if we have two clients, C1 and C2, and...

- C1 is trying to access a key that was stored into a page that the OS transferred on disk.
- C2 is trying to access a key that is fully in memory. A recently used one.
- C1 sends the query one millisecond before C2.
- Because C1 will touch a page that is swapped on disk, the process will be halted, and will wait the disk I/O needed to bring the page back into memory.
- In the meanwhile everything is stopped. Even if C2 was going to read something in memory it gets serialized and will be served after C1.

One very important goal in Redis VM (and I guess this should be a primary goal of every system with a low latency semantics) is to be able to serve keys that are in memory as fast as usually. Clients performing a query against a rarely used page will instead pay the latency penalty, without effects for other clients.

This is already a show stopper and just because of this it should not be worth continuing with the rest of the article, but well, while I'm at it it's a good exercise I guess.

The granularity is 4k pages

The kernel is able to swap/load 4k pages. For a page to be idle from the point of view of the kernel and its LRU algorithm, what is needed is that there are no memory accesses in the whole page for some time.

Redis is an in-memory data structures server, this means that our values are often things like lists, hash tables, balanced trees, and so forth. This data structures are created incrementally with commands, often in a long time. For instance a Redis list may be composed of 10k elements storing the timeline of a twitter user, accumulated in the course of six months. So every element of the list is a Redis object. Redis objects get shared, cached, and so forth: there is no good locality in such a data structure obviously.

Multiply this for all the keys you have in memory and try visualizing it in your mind: These are a lot of small objects. What happens is simple to explain, every single page of 4k will have a mix of many different values. For a page to be swapped on disk by the OS it requires that all contained objects should belong to rarely used keys. In practical terms the OS will not be able to swap a single page at all even if just 10% of the dataset is used.

Oh but this is since you are lame! Store related objects nearby...

The whole Redis semantics of being single threaded, fast, and very versatile in the data structures provided, is up to the fact that we use the good and old data structures implemented with something that is able to provide good performance even with bad locality (compared to a disk) that is: memory.

Handling these data structures with very good locality is as hard as implementing these data structures well on disk. If we could do this, it would be a much better strategy to use the inverse design: store everything on disk and use the kernel disk cache to take the hot spot in memory. Persistence and VM solved in a single pass, a no brainer.

Actually in Redis 2.2 we try to "compact" our data in memory, and in this way we obtained huge space savings. Many datasets in Redis 2.2 takes just 20% of the space that was required in 2.0. This is five times more space efficient than before. But where is the trick? That we can do this only for small lists, sets, and hashes, where O(N) algorithms are as fast as O(1) algorithms because of cache locality.

I think I already showed my point, but there are more good reasons to implement paging at application level, especially in the case of Redis.

Optimal representation on disk and on memory are very different

Many data structures are designed to be able to provide specific time complexity performances. For instance an hash table provides an element lookup time of O(1) in the average case. In a similar way a balanced tree is designed so that it's possible to update a Redis sorted set score in O(log(N)).

For this to be possible, there is a need to waste memory because you have meta data of many kinds: pointers, allocations overheads, informations per every node for augmented data structures (like our skip list implementation), and so forth. The representation of data is optimized for interacting with this data.

On the other side when values are swapped they are idle. For storage the best representation can be completely different. For instance a hash table holding name of fruits in memory can be represented on disk as a trivial comma separated string of values: "orange,apple,...."

The OS has zero knowledge of what's written in a page. Instead with application level paging we know what we are doing, and can serialize the data in the VM in the smarter way. This means from 5 to 10 times less disk I/O compared to the work performed by the kernel in the same conditions!

Aging algorithm can't be changed

And finally... what value to swap on disk? What value to take in memory?

Again, the kernel will use a simple LRU algorithm, where the granularity is the page. Redis can do much better, for instance LRU is not always the best algorithm when accessing data in a "circular" way, one record after the other and then again. Also the current Redis algorithm takes into account the size of a given value. If it's small it's not worth transferring if the age is exactly like another value that is bigger, and things like this. In Redis 2.2 we plan to provide different swapping algorithms so that people can pick what works better for a given dataset.

I think the Varnish article is not bad at all, the real problem is that an article is not enough to provide a deep understanding of the specific implementation of a different system. I hope this article provided a counter-case for the Varnish approach that can be used when it is sensible to use it. And the other way around. ■

Salvatore Sanfilippo aka antirez is an Italian computer programmer. He is currently the lead developer of Redis and works for VMware. In the past he focused on security and programming languages.

Reprinted with permission of the original author. First appeared in *http://hn.my/2006programming/*.

Commentary

By WES FELTER (wmf)

JUST TO AMPLIFY his point, if you want your program to take page faults as PHK suggests, it has to be multithreaded. If you choose event-driven concurrency you can't afford to take page faults in mmap() or read(). When you make the threads vs. events decision you're implicitly making a bunch of related decisions about I/O and scheduling as well; a hybrid approach (like using events and mmap) won't work well.

Bouncing Beholder

By MARIJN HAVERBEKE

My winning JS1K entry — [http://marijnhaverbeke.nl/js1k/] a JavaScript platform game that fits in 1024 bytes.

This is the code:

```
c=document.body.children[0];h=t=150;L=w=c.
width=800;u=D=50;H=[];R=Math.random;for($ in C=c.
getContext('2d'))C[$[J=X=Y=0]+($[6]||'')]=C[$];setInte
rval("if(D)for(x=405,i=y=I=0;i<1e4;)L=\H[i++]=i<9|L<w
&R()<.3?w:R()*u+80|0;$=++t%99-u;$=$*$/8+20;y+=Y;x+=y-
H[(x+X)/u|0]>9?0:X;j=H[o=\x/u|0];Y=y<j|Y<0?Y+1:(y=j,J?-
10:0);with(C){A=function(c,x,y,r){r&&a(x,y,r,0,7,0);fillSty
le=c.P\?c:'#'+'ceff99ff78f86eeaaffffd45333'.substr(c*3,3)
;f();ba()};for(D=Z=0;Z<21;Z++){Z<7&&A(Z%6,w/\2,235,Z?250-
15*Z:w);i=o-5+Z;S=x-i*u;B=S>9&S<41;ta(u-S,0);G=cL(0
,T=H[i],0,T+9);T%6||(A(2,25,T-7\,5),y^j||B&&(H[i]-
=.1,I++));G.P=G.addColorStop;G.P(0,i%7?'#7e3':(i^o||y^T|
|(y=H[i]+=$/99),\'#c7a'\));G.P(1,'#ca6');i%4&&A(6,t/2%20
0,9,i%2?27:33);m(-6,h);qt(-6,T,3,T);l(47,T);qt(56,T,56,\
h);A(G);i%3?0:T<w?(A(G,33,T-15,10),fc(31,T-7,4,9)):(A(7,25
,$,9),A(G,25,$,5),fc(24,$,2,h),D=B&y\>$-9?1:D);ta(S-u,0)}
A(6,u,y-9,11);A(5,M=u+X*.7,Q=y-9+Y/5,8);A(8,M,Q,5);f
x(I+'¢',5,15)}D=y>h?1:D",u);onkeydown=onkeyup=function(e)
{E=e.type[5]?4:0;e=e.keyCode;J=e^38?J:E;X=e^37?e^39?X:E:-E}
```

Why?

I've heard people wax poetic about programming old, limited-memory machines. I wouldn't know anything about those — at the time they were current, I was writing rudimentary number-guessing games in BASIC. But doing this competition entry gave me a taste of what they might be talking about.

In typical 21st-century programming, the machine limits one has to deal with are wide and fuzzy. Program size is rarely an issue, so like painters working on an infinite canvas, we often don't know when to stop. When a program has to fit in a tightly limited space, the experience is different. You program by carefully refining every single expression, chipping away at your code until it reflects your vision as well as it can.

In terms of productivity, this is an awful way of coding. But it certainly is fun. Not to mention that it gives me an excuse to use every kind of weird hack I can think of.

How?

For a start, of course, there are the tiny local tricks that save a few bytes here and there, which adds up to at least a hundred bytes on the whole program. `|0` truncates, `&&` or `?:` can replace `if` (sometimes), `&` can replace `&&` (sometimes), you can reuse initializers (`J=X=Y=0`), a `with` statement can shorten object access, etc.

Compression algorithms, such as Google's Closure Compiler and UglifyJS, and various `eval/replace` hacks suggested for the JS1K contest, don't really do much on properly hand-compressed code. In fact, they all ended up making the code bigger...

The tiny size required me to design the program in a "holistic," highly un-modular way, meaning every single aspect of the program could influence every other one. There was an issue causing the clouds to be drawn incorrectly for negative X coordinates. To work around this would have required quite a few extra characters (I was using `x|0` where I actually needed `Math.floor(x)`). Instead, I made the playing field start at 400 and put empty space at the start to prevent the player from seeing any negative X coordinates. Problem solved.

Mechanized Abbreviation

The coolest hack in this program is probably the mechanized abbreviation of the canvas context methods. Method names like `quadraticCurveTo`, `createLinearGradient` are nice and explicit, but those two taken together already eat 3.5% of the bytes available — when only referenced once! I needed to use them, but I wanted to avoid spelling them.

Turns out I can get away with that. At the start of the program there is a `for/in` loop that goes over the properties of the canvas context, and adds a new property, with a shorter name, for each of them. It took some experimenting to find an abbreviation algorithm that doesn't have clashes on any of the methods we use — I ended up using the first letter of the name plus the the 7th letter, if any. So `lineTo` becomes `l`, and `quadraticCurveTo` becomes `qt`. I can then use these short names to actually access the methods — without ever having written out the full name.

This does, of course, not work for properties like `fillStyle`. You can copy those, but the copies won't do anything.

Functions As a Scarce Resource

Functions are hugely useful for factoring out pieces of shared functionality, and thus shortening code. Unfortunately, the word "function" is 8 characters, and the minimal overhead for a function definition something like 14 bytes, 20 if you actually want to return something.

Thus, a function has to be really, really useful before it pays off to define it.

The program started off with five functions, which has since been reduced to two. In one of these places, I have little choice — `window.onkeydown` only takes function values. I'm using the same function for `onkeydown` and `onkeyup`, which turned out to be more efficient anyway. The checks for which key is pressed or released are also repeated in both. To check whether an event is a keydown or a keyup, I used `e.type[5]`, where `e` is the event object. If this is a keyup event, the type of the event does not have a 6th character, so that this evaluates to a falsy value.

The other function used is the one called `A`. This rolls three pieces of functionality into one (saving me two function keywords). It takes a `fillstyle` as its first argument, and an optional `x`, `y`, and `radius` after that. If the optional arguments are provided, it starts by drawing a circle. Then it sets the `fillStyle` of the canvas context to the provided style, or — if the style is not a gradient — it uses it as an index into a string of colors. After this, it calls (the abbreviated versions of) `fill()` and `beginPath()` on the canvas context. Note that, because a canvas context is specified to start with an empty path, it is safe to start drawing before the first call to `beginPath`, and thus `beginPath`, though it is usually done before one starts drawing, can be made part of our "after-drawing routine."

This function is used in three different ways. Obviously, it is used to draw colored circles (the game contains a lot of circles). But code that has drawn a path in some other way (the ground blocks) can also call it to just assign a `fillStyle` and fill the path. Finally, code that just wants to set the `fillStyle` can use it for that — as long as no path is in the process of being drawn. Now that's reusability. The program uses this function in ten different places.

The World

The game world is divided (along the x axis) into 50-pixel-wide units. When starting a game (or at game-over time), an array is initialized containing a randomized height-map. The gaps work mostly the same as the other positions, their height is just off the bottom of the canvas. The generating algorithm takes some care to not introduce gaps of more than one unit, since those would be unjumpable. This `heightmap` array (plus the player's position, speed, and a time counter for animation) represents pretty much the whole game state.

So how does the game know where the coins are, if it is not explicitly keeping state for them? Every block whose random height is divisible by 6 gets a coin, and when the player collects the coin, .1 is subtracted from the height, and the coin no longer shows up.

Apart from block height, block's x-coordinates can also be used to add distinctive features. Every third block gets a decorative tree, if it is visible. If it is invisible, it gets a (stylized) Piranha Plant. Every seventh block is purple/sinky. This produces a relatively nice random world, without requiring involved data structures or lots of code.

Physics

The "physics" in this game are coded in an entirely ad-hoc and special-cased way. Player movement needs to be restricted in two ways — you can't walk through the sides of blocks, and you shouldn't fall through the top. The first is handled by simply cancelling horizontal movement whenever it would take the player more than nine (the higher 1-byte number...) pixels below the ground, and the second is simply a direct check against the height array. If the player is below or on the ground, his y position is set to ground level, and his vertical speed is set to zero, unless the up arrow is pressed, in which case it is set to minus ten (minus is up). In the other case, where the player is above the ground, one is added to the vertical speed, creating a gravity effect.

Collision detection is also handled case-by-case. The most involved case is collision with the plants, which takes some 20 characters. The "is the player near the middle of this block" part of the test is reused to determine whether a coin is being picked up.

Code

Below follows a somewhat expanded, formatted, lightly commented version of the code. The interval code was made a function (it is a string the compressed version) to conveniently allow newlines inside of it.

```
canvas=document.body.children[0];
screen_height=time=150;
last_height=screen_width=canvas.width=800;
unit=dead=50;
heights=[];

// The abbreviation loop, initializing the variable needed
by the key-handlers on the side.
for(prop in context=canvas.getContext('2d'))
 context[prop[jump=speed_x=speed_y=0]+(prop[6]||'')]=conte
 xt[prop];

setInterval(function(){
 if(dead)
   // initialize the player position, score, and heightmap
   for(x=405,i=y=score=0;i<1e4;)
     // (screen_width is reused as the off-the-screen
     // height of gap blocks)

     // a block can be a gap if its index is <9, or if the
     // last block was no gap. after this test, a random
     // number is compared to .3 to determine whether an
     // actual gap is generated, or regular random height.
     last_height=heights[i++]=
       i<9|last_height<screen_width&Math.
       random()<.3?screen_width:Math.random()*unit+80|0;

 // silly formula to create parabolic movement based on
 // the time
 plant_pos=++time%99-unit;plant_pos=plant_pos*plant_
 pos/8+20;


 y+=speed_y;
 // only move horizontally if that doesn't take us deep
 // underground (x/unit|0 fetches the index of the block
 // below an x coordinate)
 x+=y-heights[(x+speed_x)/unit|0]>9?0:speed_x;
 // compute final player height index, and ground level
 // under it
 ground=heights[player_index=x/unit|0];
 // adjust y and speed_y based on whether we are on the
 // ground or not
 speed_y=y<ground|speed_y<0?speed_y+1:(y=ground,jump?-10:0);

// we'll need the context a lot
 with(context){
   A=function(color,x,y,radius){
     // a is the abbreviated form of arc
     radius&&a(x,y,radius,0,7,0);
     // if color is not a gradient object (we set a P
     // property in gradient objects), it is an index into
     // a set of colors
     fillStyle=color.P?color:'#'+'ceff99ff78f86eeaaff
     ffd45333'.substr(color*3,3);
     // f for fill, ba for beginPath
     f(); ba();
   };

   // now loop over visible, or close to visible, blocks,
   // and draw them and their clouds
   for(dead=i=0;i<21;i++){
     // this loop is reused for drawing the background/
     // rainbow, which consists of seven concentric
     // circles. there's no good reason why interleaving
     // clearing the screen with drawing the screen's
     // contents should work, but in this case it does
     i<7&&A(i%6,screen_width/2,235,i?250-15*i:screen_width);

     // we start drawing 5 units in front of the player
     // (first four will be off-screen, needed just for
     // clouds)
     height_index=player_index-5+i;

     scroll_pos=x-height_index*unit;
     // since player screen position is fixed, we can use
     // scroll position for collision detection.
     // this variable indicates whether the player is in
     // the 'middle' of the current block
     player_in_middle=scroll_pos>9&scroll_pos<41;

     // ta for translate. move to start of block to make
     // other drawing commands shorter
     ta(unit-scroll_pos,0);
     // cL for createLinearGradient, for the ground/grass
     // gradient
     gradient=cL(0,height=heights[height_index],0,height+9);
     // if height is divisible by 6, there's a coin here.
     // draw it. if the player is standing on the ground,
     // in the middle of this unit, pick up the coin
     height%6||(A(2,25,height-7,5),y^ground||player_in_
     middle&&(heights[height_index]-=.1,score++));
```

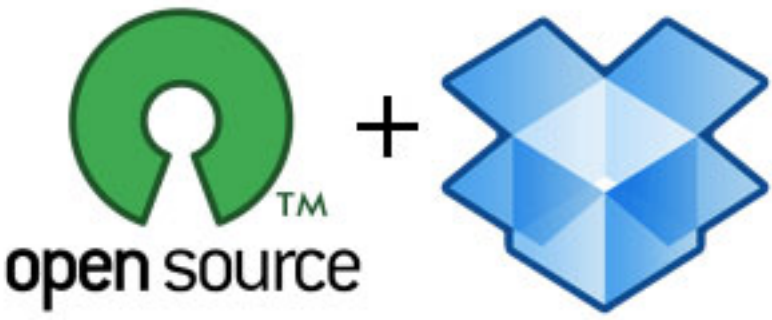
```
      // abbreviate, since we need this twice (and use it
      // again to test whether a value passed to A is a
      // gradient)
      gradient.P=gradient.addColorStop;
      // this implements sinky terrain---when the index is
      // divisible by 7, we use a different color,
      // and do the sinking if the player is standing here
      gradient.P(0,height_index%7?'#5e1':(height_
      index^player_index||y^height||

(y=heights[height_index]+=plant_pos/99),'#a59'));
      // brown earth color for the bottom of the gradient
      gradient.P(1,'#b93');

      // this draws the clouds
      height_index%4&&A(6,time/2%200,9,height_index%2?27:33);

      // draws the terrain block. m is moveTo, qt is
      // quadraticCurveTo, l is lineTo
      m(-6,screen_height);qt(-6,height,3,height);l(47,heigh
      t);qt(56,height,56,screen_height);A(gradient);

      // draw deco trees or piranha plant (height==screen_
      // width for gap blocks), check for collision with
      // plant
      height_index%3?0:height<screen_width
        ?(A(gradient,33,height-15,10),fc(31,height-7,4,9))
        :(A(7,25,plant_pos,9),A(3,25,plant_pos,5),
        fc(24,plant_pos,2,screen_height),
          dead=player_in_middle&y>plant_pos-9?1:dead);

      // undo block-local translation
      ta(scroll_pos-unit,0)
    }

    // draws the player, using the speed to adjust the
    // position of the iris
    A(6,unit,y-9,11);
    A(5,iris_x=unit+speed_x*.7,iris_y=y-9+speed_y/5,8);
    A(8,iris_x,iris_y,5);

    // color is already dark from eye pupil, draw score
    // with this color
    fx(score+'¢',5,15)
  }

  // check whether the player has fallen off the screen
  dead=y>screen_height?1:dead
},unit);
```

```
onkeydown=onkeyup=function(e){
  // if this is a keydown event, new_val gets the value 4,
  // otherwise 0
  new_val=e.type[5]?4:0;
  e=e.keyCode;

  // give jump a truthy value if up was pressed, falsy if
  // up was released
  jump=e^38?jump:new_val;

  // similar for speed_x, inverting new_val if left is
  // pressed
  speed_x=e^37?e^39?speed_x:new_val:-new_val
}
```

Marijn Haverbeke is a programming language enthusiast and polyglot. He's worked his way from trivial BASIC games on the Commodore, through a C++ phase, to the present where he mostly hacks on database systems and web APIs in dynamic languages. He's about to publish his first book — Eloquent JavaScript: A Modern Introduction To Programming [http://eloquentjavascript.net].

Reprinted with permission of the original author.
First appeared in *http://hn.my/js1k/*.

Build an Open Source Dropbox Clone



By PHIL CRYER

First off, if you haven't tried Dropbox, you should check it out; sync all of your computers via the Dropbox servers, their basic free service gives you 2Gigs of space and works cross-platform (Windows, Mac and Linux). I use it daily at home and work, just having a live backup of my main data for my work system, my home netbook, and any other computer I need to login to is a huge win. Plus, I have various 'shared' folders that distribute certain data to specific users and co-workers to whom I've granted access. This means work details can be updated and automatically distributed to the folks I want to review or use the data immediately. I recommend everyone try it out to see how useful it is, as it's turned into a game changer for me. So when Dropbox made headlines that they were supporting Linux, and releasing the client as open source, it got hopes up that users would be able to run their own, private Dropbox systems. In the end, it was only the client that was open source; the server would remain proprietary. While slightly disappointing, this is fine because it's a company trying to make money. I don't fault Dropbox for this, it's just that a free, portable service like that would be a killer app.

Meanwhile at work I'm working on a solution to sync large data clusters online and the project manager described it as the need for 'Dropbox on steroids'. Before I had thought it was more complicated, but after thinking about it, I realized he was right. Look, Dropbox is a great idea, but it obviously is just a melding of something similar to rsync, with something watching for file changes to initiate the sync, along with an easy- to-use front end. From there I just started looking at ways this could work, and there are more than a few; here's how I made it work.

Linux now includes inotify, which is a kernel subsystem that provides file system event notification. From there all it took was to find an application that listens to inotify and then kicks off a command when it hears of a change. I tried a few different applications like inocron, inosync and iwatch, before going with lsyncd. While all of them could work, lsyncd seemed to be the most mature, simple to configure and fast. Lsyncd uses inotify to watch a specified directory for any new, edited or removed files or directories, and then calls rsync to take care of business. So let's get started in making our own open source Dropbox clone with Debian GNU/Linux (Squeeze)

Ladies and gentlemen, start your ~~engines~~ servers!

First, you need two servers: one being the server and the other the client. (You could do this on one host if you wanted to see how it works for a proof of concept).

Install OpenSSH client and server

First you'll need to install OpenSSH on both the Client and Server. On the remote system:

```
apt-get install openssh-server
```

On the local box it's more than likely that the client is installed, but just in case:

```
apt-get install openssh-client
```

Configure SSH for Password-less Logins

You'll need to configure SSH to use password-less logins between the two hosts you want to use, as this is how rsync will pass the files back and forth. I've previously written a HOWTO on this topic, so we'll crib from there.

First, generate an SSH public key:

```
ssh-keygen -N '' -f ~/.ssh/id_dsa
```

You shouldn't have a key stored there yet, but if you do it will prompt you and ask if you want to overwrite it; make sure you overwrite it.

Enter passphrase (empty for no passphrase):

```
<Enter>
```

Enter same passphrase again:

```
<Enter>
```

We're not using pass phrases so the logins between the systems can be automated. This should only be done for scripts or applications that need this functionality, it is not for logging into servers lazily, and **it should never be done as root!**

Now, replace REMOTE_SERVER with the hostname or IP that you're going to call when you SSH to it, and copy the key over to the server:

```
ssh-copy-id REMOTE_SERVER
```

Note that if you have an older system you may not have ssh-copy-id installed, so you can do it the old way by piping the output of your key over SSH (which is good to know how to do anyway):

```
cat ~/.ssh/id_rsa.pub | ssh REMOTE_SERVER 'cat - >> ~/.ssh/
authorized_keys2'
```

Lastly, we need to set the permissions on the key file to a sane level:

```
ssh REMOTE_SERVER 'chmod 700 .ssh'
```

Now, give it a go to see if it worked:

```
ssh REMOTE_SERVER
```

You should be dropped to a prompt on the remote server without being prompted for a password. If not you may need to redo your .ssh directory, so on both servers:

```
mv ~/.ssh ~/.ssh-old
```

and goto 10

Install rsync and lsyncd

Next up is to install rsync and lsyncd. rsync is a basic command and should already be installed (you don't need to run it on the server, just the client on both systems), but to make sure you have it, and install lsyncd at the same time:

```
apt-get install rsync lsyncd
```

Note that before Squeeze there was no official Debian package, but it's simple to build from source and install if you need to. First off, if you don't have build essentials you'll need them, as well as libxml2-dev to build the lsyncd source. Installing those is as simple as:

```
apt-get install libxml2-dev build-essential
```

Now we'll download the lsyncd code, uncompress it and build it:

```
wget http://lsyncd.googlecode.com/files/lsyncd-1.39.tar.gz
tar -zxf lsyncd-1.39.tar.gz
cd lsyncd-1.39
./configure
make; make install
```

This install does not install the configuration file, so we'll do that manually now:

```
cp lsyncd.conf.xml /etc/
```

Configure lsyncd

Next we need to edit the configuration file now located in /etc The file is a simple, well- documented XML file, and mine ended up like so – just be sure to change the source and target hosts and paths to work with your systems:

```
<lsyncd version="1.39">
  <settings>
    <logfile filename="/var/log/lsyncd"/>
    <!--Specify the rsync (or other) binary to call-->
    <binary filename="/usr/bin/rsync"/>
    <pidfile filename="/var/run/lsyncd.pid"/>
    <callopts>
      <option text="-lt%r"/>
      <option text="--delete"/>
      <exclude -file/>
    <source />
    <destination />
    </callopts>
  </settings>
  <directory>
    <source path="/var/www/sync_test"/>
    <target path="desthost::module/"/> </directory>
</lsyncd>
```

Launch lsyncd in debug for testing

We're ready to give it a go, may as well run it in debug for fun and to learn how lsyncd does what it does:

```
lsyncd --conf /etc/lsyncd.conf.xml --debug
```

Watch for errors, if none are found, continue.

Add files and watch them sync

Now we just need to copy some files into this directory on the source box:

```
/var/www/sync_test
```

And again, watch for any errors on the screen, if these come back as a failed connection it'll be an SSH/key issue; common, and not too difficult to solve. From here add some directories and watch how they're queued up, and then take a look at them on the remote box: from this point out it "just works." Now give it more to do by adding files and directories, and then the logging for errors while they sync. As it stands the system uses the source system as the preferred environment, so any files that change, or are added or removed, will be processed on the remote system. This is analogous to how Dropbox works, you can use multiple sources (your laptop, your desktop, etc) and their server serves as the remote system, keeping all the clients in line.

Reprinted with permission of the original author.
First appeared in *http://hn.my/dbclone/*.

Conclusion

You should now have a basic, working Dropbox style setup for your own personal use. I had this running and used it to sync my netbook back to my home server, and then have my work desktop sync to my home server, so both the netbook and the desktop would stay in sync without me doing anything besides putting files in the specified folder. For my week long test I ran a directory alongside my Dropbox directory just to see how they both acted, and I didn't have any failures along the way.

Epilogue

This article is an updated version of one that originally appeared on *http://fak3r.com/* in September 2009 under the title "HOWTO build your own Dropbox clone." In the year since it's publication I've received a great deal of interest in my idea, and have continuously thought of ways to improve upon it. In the configuration file for lsyncd it has a line that reads, "Specify the rsync (or other) binary to call," and this is the kind of flexibility I needed. Today I'm utilizing Unison to handle the syncing for the project, and besides having many attractive features, it's the right solution to do true two-way syncing. This fits the one-to-many Dropbox model better than rsync does. The project has now been released as open source under the name lipsync, and is available here: *https://github.com/philcryer/lipsync* Take it, try it out and improve upon it. If you have troubles ping me on my blog, contact me via GitHub or email; I'm happy to help. Thanks. ■

Phil Cryer is a husband, father, artist, music lover, hacker, open source technologist and civil liberties activist. He currently works as a senior systems engineer currently building a global, distributed, storage network. He hold a bachelor's degree in fine arts, and believes that imagination is more important than knowledge. He can be reached at *http://philcryer.com*.

Mobile Notifications for Everything

Impossibly quick push notifications

for your web app, service or weekend project.

Cheaper than SMS. Painless integration. No mobile app coding!

Check out our API and start building today!

https://api.notifo.com

Java Trap, 2010 Edition

By PAUL QUERNA

As a member of the Apache Software Foundation, my views on open source tend to gravitate towards more liberal licenses, like the Apache License (v2.0), BSD, or MIT licenses. I strongly believe in enabling companies to take open source software and do whatever they wish to do with it, placing as little restrictions as feasible under current laws. I believe that better communities for software development are enabled by these liberal licensing situations. Rather than creating a single power with significantly more rights, as seen in the "open core" movement, liberal open source development encourages real, dedicated and sustainable contributions, made by companies with business models other than selling support and 'enterprise features'.

I have to be honest — I am not a huge fan of Java the language — I would rather write code in Python, Javascript, C, C++, or heck maybe even PHP, but I find myself surrounded by Java everywhere. Java and the JVM today are core to many components we are using to build Cloudkick, and there are no viable alternatives.

Today IBM announced they are shifting their focus, and will be developing on top of the OpenJDK. This comes in addition to the Oracle lawsuit against Google over Android. Oracle is good at big company politics, and at extracting value — I'm sure they will extract every penny out of Sun's husk.

While Sun, now Oracle, has licensed the OpenJDK itself under the GPL, the licensing of the TCK has been a problem for more than 5 years. Other blog posts go into far more detail about this, and I encourage you to understand all the details about the story of the TCK, Apache, and Sun — but it isn't what I want to focus on.

I consider myself an open source advocate, though in a far different manner than someone like Richard Stallman, creator of the GNU Project. Richard's views and my own don't often align around many topics, but the increasing turmoil in the Java world has changed some beliefs I have about software platforms and licensing.

More than 6 years ago, "Free but Shackled – The Java Trap" was published by Richard. While I don't agree with the moral arguments about the freedom of software, I now believe that the Java platform is a trap.

Richard speaks about the Free World, and many other GNU priorities in this excerpt, but I believe the core point is the most important. If your code depends on a platform, you are at the mercy of that platforms licensing and development:

> *This problem can occur in any kind of software, in any language. For instance, a free program that only runs on Microsoft Windows is clearly useless in the Free World. But software that runs on GNU/Linux can also be useless if it depends on other nonfree software. In the past, Motif (before we had LessTif) and Qt (before its developers made it free software) were major causes of this problem. Most 3D video cards work fully only with nonfree drivers, which also cause this problem. But the major source of this problem today is Java, because people who write free software often feel Java is sexy. Blinded by their attraction to the language, they overlook the issue of dependencies and fall into the Java Trap.*

When you build software in Java and the JVM, you are being locked into only running it on a platform controlled by a single company — Oracle. Oracle is working to maintain this platform control by refusing to remove the field of use clauses in the TCK, effectively preventing Apache Harmony from ever

> **“When I am picking a platform to build upon, I want to know it will be around regardless of the whims of a single company.”**

being able to ship a real release. The lawsuit against Google also confirms the fear of Oracle using their control of the platform aggressively.

The problem is not so much about Oracle controlling their code. As I said above, I believe in the rights of a company to do as these choose — but at the same time, if they choose to be bad stewards of this, I will choose not to use their platform. Most importantly in the Java world, is that stranglehold being placed upon 3rd party implementations. Oracle could close source the OpenJDK for all I care, but what offends me most is their desire to squash alternative implementations.

Consider some alternatives to Java, which all have multiple implementations now:

- Python: CPython, but also has PyPy, IronPython, and Jython.
- Ruby: MRI, but also JRuby, MacRuby,
- Javascript: v8 (node.js), Spidermonkey, whatever-safari-is-calling-their-JS-engine-now.
- C/C++: Clang and GCC
- C#: CLI and Mono

These multiple implementations of the languages are creating innovation on their respective platforms. They are all for the most part driven by diverse communities, mostly under liberal licenses. Communities built around common goals and beliefs, rather than arcane licensing policies trying to protect a company's mobile market. In Java you will only be given one choice, the choice that Larry and Oracle give you. Any attempts to build an alternative implementation will be made exceedingly difficult.

When I am picking a platform to build upon, I want to know it will be around regardless of the whims of a single company. I want to know there is a diverse community behind it. I want people to be experimenting with new ways to build a VM to make the platform even better.

This is why I must ask, how can anyone pick Java and the JVM on which to build their company's future? I know Oracle and IBM — they will pump millions into the continued development of the platform, but it's not a platform I want to be using. Big companies throwing around development like this don't create the values I find essential in picking a platform. Oracle is going to control the future of Java. I don't know what will happen to the Java Community Process, but I lack any faith in it continuing.

Take a hard look at your development, why are you using Java? Are you building upon a platform where open experimentation is encouraged, and not feared? It is impossible for a business to pivot and abandon Java in a day, but after the events of the last few months, I will seek to use alternatives wherever possible.

Is your platform free, or is it a trap? ■

Paul Querna is the Chief Architect at Cloudkick, a Y-Combinator funded start-up. Cloudkick specializes in portability and openness between cloud providers. 1,000s of companies use Cloudkick to manage their infrastructure on Amazon EC2, Rackspace Cloud, GoGrid, etc. Paul has participated in many open source projects and is a committer to the Apache HTTP Server and Apache Libcloud, the open source library for developers to build portable cloud applications. Paul also previously served as VP of Infrastructure for the Apache Software Foundation.

Reprinted with permission of the original author.
First appeared in *http://hn.my/javatrap/*.

IDEWTF

By LUKE PALMER

Photoshop, Premiere, Maya, AutoCAD, ProTools, Finale, Reason, InDesign. These are the state-of-the art tools for creators. They are all rich, powerful, versatile programs that strive to work at the artist's level of thought.

And the people who write this amazing software get to use… Visual Studio? Eclipse? Emacs? At least I've heard IntelliJ IDEA is great, I've never used it. But when contrasting these to the tools above something seems missing. Like a library full of features that bring programs to the coder's level of abstraction. Languages are attempting this now, but languages are written in text and IDEs are basically text editors with a few extra features. Photographers have clone (a tool that lets them erase sections of images replacing it with something that looks convincingly the background in that area), we have refactor-rename that doesn't even respect alpha conversion (avoiding name conflicts with your new name).

Why do we even have to worry about alpha conversion? That's like a composer worrying about MIDI patch numbers! We still denote identity with a string? Premiere wouldn't have that — it would link directly to the clip in question. Who cares if you have named something else the same thing? My friends have no trouble distinguishing me from another Luke who walked in the room.

And files? We have libraries, namespaces, modules, classes, and functions to organize our code. Files are almost entirely orthogonal and not-almost entirely meaningless. Kudos to CodeBubbles for noticing and removing that tumor. But, that's just a guy at a university, so naturally we won't get to use that for real for quite some time.

What's up with import statements? That's just some junk that comes with representing programs as text. Eclipse has surpassed those… sort of… but we're not all Java programmers. Why can't I just type the class name and then pick the one I want from a list?

All the state-of-the-art creative programs have multiple views: more than one way to see your creation to get a better handle on it. Maya has isometric, wireframe, flat-shaded, full light…. We have the class hierarchy view. Oh boy. Why can't I look at

```
while (!queue.empty()) {
  Production p = queue.pop();
  predict(p);
  if (p.star.is_terminal) { scan(p); }
  complete(p);
}
```

click a little [+] next to predict(p) and see it right there inline, with its argument replaced by p? Oh, that's how that works, cool, [-]. Instead we go to its definition, where we see it next to the functions we happened to define near it, about which we care nothing. Then we substitute the arguments in our heads, fathom loop invariants and fail to see how they are violated, and spend the next 5 minutes wondering if p is mutated in this call chain.

How come I can still have syntax errors? How come it is ever possible to have a syntax error in a program? Shouldn't the IDE at least be helping me to have a valid program all the time? Finale doesn't let you write a measure with the wrong number of beats and then complain when you push play. It just fixes it for you — "oh look, you need a rest there."

"My indentation's wrong. Oops, rename missed that one. Oh right, need to import Data.List. Ugh, namespace pollution. Fine, looks like I need to copy and paste again because abstracting will be a pain. I hate how you can only have one class per file and how it discourages small classes. Shit, that mFoo/foo accessor pattern again… weren't get/set supposed to do away with the need for accessors? Fuck, looks like this virtual method needs another parameter — give me fifteen minutes."

Do we not hear ourselves?! Software developers, the masters that create the masters' tools, are touching up Avatar with MS Paint. Shouldn't we be sculpting effortlessly a masterpiece with a beautiful dynamic interface while robots bring us platters of Mountain Dew? We're wasting our time with spelling errors while the 3D artist in the back is putting finishing touches on his city.

W. T. F. ■

Luke Palmer is an indie game developer for Hubris Arts by day, a Haskell fanatic by night. He is best known for his research in functional reactive programming — a way to write games and other interactive applications in purely functional style. He is currently researching ways to automatically extract safely reusable code, in order to build a search engine for code snippets.

Reprinted with permission of the original author.
First appeared in *http://hn.my/idewtf/*.

SEEING IS BELIEVING

RUBY ON RAILS 3 TUTORIAL SCREENCAST SERIES

by Michael Hartl, author of Rails Tutorial and RailsSpace

"My former company (CD Baby) was one of the first to loudly switch to Ruby on Rails, and then even more loudly switch back to PHP… This book by Michael Hartl came so highly recommended that I had to try it, and Ruby on Rails Tutorial is what I used to switch back to Rails again… Though I've worked my way through many Rails books, this is the one that finally made me 'get' it."

—From the foreword by DEREK SIVERS

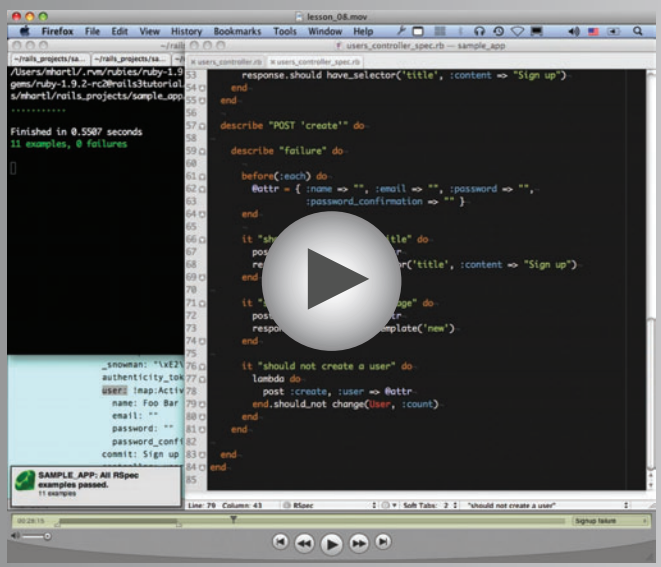
"I got review access to all of the material a week ago and can confirm that, yes, these screencasts are awesome… If you basically want to be able to look 'over the shoulder' of an experienced Rails developer and see how a Rails development environment is set up and how multiple apps are built, there's nothing that can beat this. This isn't a set of 'build a blog in 15 minutes' videos—it's a complete course that could kick off a new career for you with Rails 3.0."

—PETER COOPER, Ruby Inside

GET A FREE LESSON, see testimonials, and read the book in PDF and HTML formats at http://RailsTutorial.org

RECEIVE 10% OFF
any combination of products
ENTER COUPON CODE
"hackermonthly"
valid now through 12.31.10

BUILT FROM THE GROUND UP FOR RUBY ON RAILS 3



- Learn to make real, industrial-strength web applications
- 12 individual lessons totaling more than 15 hours
- Dovetails with, but goes beyond the Rails Tutorial book

Rails Tutorial
railstutorial.org

The DUCT TAPE Architect

By FREDRIK JOHANSSON

I HAVE BEEN DOING software architectural work for a long time now, and as it turns out, the 'right way' of solving things may not always be the best way. Below are two anecdotes from life in the trenches.

The Case of the Database Bottleneck

I was visiting a company where we discussed their current system design and what problems they were experiencing. Their system had a respectful peak of 7,000 concurrent users and it turns out that at those peak times they started to hit the limit of their database which was running as a single entity.

We discussed the regular slew of database scaling solutions such as sharding, dedicated reader nodes etc. and some pros and cons with each solution.

As it turned out however, they solved it in their own way. "Yeah, we solved it," they came back to me when I asked them about it. "We bought an SSD drive which replaced the old hard-drive. It is much faster now." they said matter-of-factly.

Naturally I scoffed at this and thought for myself that they have only bought themselves a little bit of time and, in at best, they could grow by 50-100% but then it would be the same issues all over! Rookies! Surely they did not understand the beauty of unlimited, linear scaling with sharding?

Within less than a year, they had grown about 20% and were then bought up by a bigger player in the industry. As is custom, their system was erased from the face of the earth in favor of the larger one. They never hit the limit of the SSD.

Later on I also did some calculations, if they had grown by 100%, they would have become one of the top 5 actors in the market and their profits would have been through the roof. Their development budget would have been completely different by then.

So when looking back, in this case, it actually seems like the SSD solution was the right thing to do. They only needed to buy some more time for the deal to come through.

The Case of the Missing Scheduler

Another case occurred when I was reviewing a large gaming network that was running cash games as well as tournaments. There where many tournaments running on a daily basis and most of them were re-occurring events, such as The Daily Lunch Tournament etc. Almost every gaming network I know has a scheduling option for tournaments. An administrator would enter a tournament template and then say something like 'run every day at 12 AM' for instance. You would also be able to create a future tournament and say 'start this tournament on October 10 at 18 AM'. Then the system would then create and start the tournament as specified.

This network did not have that.

Instead, they had about 10 employees in Indonesia who would work in shift and manually create each tournament and then manually click 'start' to start them. Nuts! This must surely be fixed!

So we started a discussion and I don't remember my exact word, but they were something like: "This is insane! Surely we should be able to implement a simple scheduler in the system?"

To which they replied something like: "Sure. But we have made an estimate on the time it would take us, and the cost of the developers on US salaries to implement this corresponds to about 7 years of the Indonesian guys doing this manually."

Yikes.

"Besides, do you want to be the guy who calls them up and tell them and their families that they are losing their jobs? And for what? Saving a buck after 7 years? We have a choke-full backlog to work on anyway."

Hmmm. Maybe it would not be worth cutting other features out in order to prioritize a feature that would cause 10 people their jobs and not save any money for a long time. Could this be? What kind of socialist development company was this?

As you might have guessed by now, by being able to dedicate their developers to other things rather than make the Indonesians redundant they were able to dish out new feature that actually attracted new players. Which turned out to be very successful for the owners in the end.

Summary

Am I advocating that you shouldn't care about scalability (just buy SSD's!) or never automate tasks because there's cheap labour to be found? Am I advocating quick hacks and avoiding solid engineering principles? Of course not.

But sometimes it is good to try and raise the view a bit and try to see what *actually* needs to be solved. As engineers we do sometimes get stuck on implementing the 'right thing' and lose sight of reality as it comes. I know I do. ■

Fredrik Johansson is the founder and CEO of Cubeia Ltd, a premium software provider providing scalable and robust solutions for the online gaming industry. Fredrik has experience from working with architectural challenges on multiple high volume multiplayer installations. Additional information about Fredrik and Cubeia can be found at *www.cubeia.com*.

Reprinted with permission of the original author.
First appeared in *http://hn.my/ducttape/*.

Commentary

By JUAN PABLO (jpablo)

I would choose the SSD every time over a method that requires doing a lot of consulting and engineering of the current system like sharding.

Why spend a lot of time and work when a simple hardware upgrade will work ?

And you are deluding yourself if you think that the sharding model you are going to implement is not "only buying you time" and you will have to do additional engineering over time if you keep growing.

Dream. Design. Print.

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

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

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

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

25% Off the First Issue You Publish

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

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

MAGCLOUD