The Rules of a Zen Programmer

HACKERMONTHLY

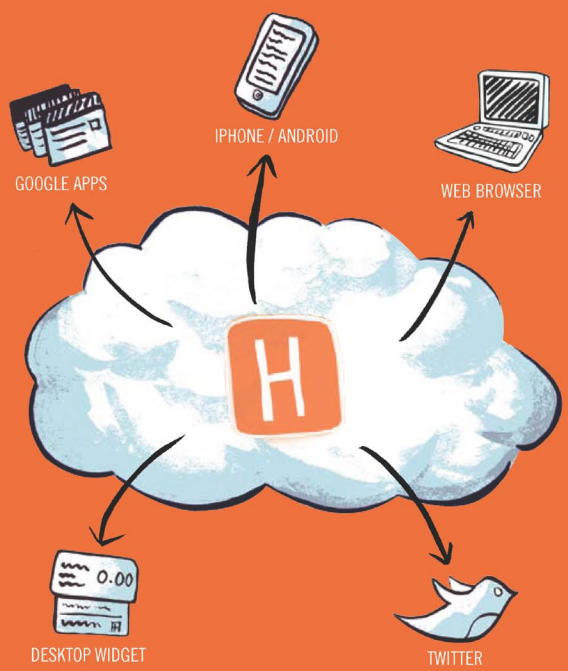
Issue 22
March 2012

HARVEST

TIMESHEETS INVOICES REPORTS

Track time anywhere, and invoice your clients with ease.

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



Why Harvest?

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

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

Curator
Lim Cheng Soon

Contributors
Christian Grobmeier
Jason Shen
Jacques Mattheij
Paul Graham
Francis Irving
James Hague
Ilya Grigorik
John Carmack
Peter Schuller
Rob Landley
Igor Teper
Reginald Braithwaite

Proofreaders
Emily Griffin
Sigmarie Soto

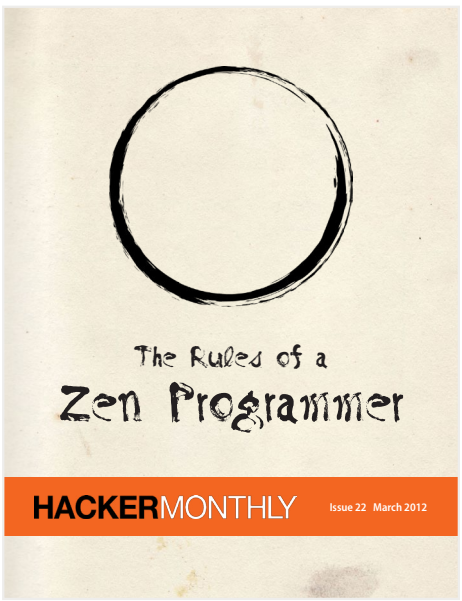
Printer
MagCloud

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

Advertising
ads@hackermonthly.com

Contact
contact@hackermonthly.com

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



Contents

For links to Hacker News dicussions, visit *hackermonthly.com/issue-22*

The Rules of a Zen Programmer

By CHRISTIAN GROBMEIER

ONE RAINY MORNING, I found myself sitting at my desk thinking about efficient working. Before I started as a freelancer I had some days when I worked a lot but could only look back on a worse outcome.

I started with Zen practice back in 2006. What clearly came to my mind before long was this: the old Zen masters already knew hundreds of years ago how programmers today should work. Even though I don't like these "be a better programmer" articles, I want to outline some of my thoughts from that morning.

1 Focus

If you have decided to work on a task, do it as well as you can. Don't start multiple things at the same time. Do only one thing at a time. You won't become quicker, just scattered. If you work multithreaded, you'll become exhausted, make more errors, and waste time jumping from one task to another. This is not only about programming; this is a general tip.

Kodo Sawaki says: if you need to sleep, sleep. Don't plan your software when you need to sleep. Just sleep. If you code, code. Don't daydream — code. If you are so tired that you cannot program, sleep. Even known multitaskers like Stephan Uhrenbacher have decided to work single threaded. I had a similar experience to Stephan, and finally I wrote Time & Bill, a time tracking tool. My goal was to track my time so easily that I would do it even for small tasks like a phone call. Now I can create a few stopwatches at the beginning of the day and track my time with only one click. The outcome was a disaster: sometimes I just worked a few minutes on a task until I moved on to the next one. Now I am better. Similar to the Pomodoro technique I plan a few time slots and concentrate on them. No chatting, no sleeping, no checking out of a new great game at the App store.

2 Keep Your Mind Clean

Before you work on your software, you need to clean up your memory. Throw away everything in your mind for the time being. If you have trouble with something, don't let it influence you. It is mostly the case that trouble will go away. If the trouble is so heavy that you can't let it go, don't work. Try to clear things up. But when you start working, let the outer world fade away.

Something exciting on the mailing list? Leave it there. You can follow the exciting stuff later. Shutdown what fills your mind with clutter: close Twitter, Facebook, and your email. You should even mute your mobile and leave it in your pocket. You can say it is similar to item #1, focus. But there is one more restriction: don't use your mobile before work or at lunch. They connect you with the outer world and probably bring up some new trouble or things which require your attention.

Think like this: usually your mind is pretty clean when you wake up at the morning. If it is not, some exercise helps (I do long distance running). If you feel clean and refreshed, go to work and work as well as you can. When you leave your work then you can fill up your mind with clutter. You'll see it is not so much fun if you have a full working day behind you. Twitter and Co are consuming too much of your energy. Do not think it is just for a minute. It's not.

> **If you want to gain something and don't care about your life now, you have already lost the game.**

3 Beginner's Mind

Remember the days when you were a beginner., when you feel like you have never learned enough. Think of yourself as you were a beginner every day. Always try to see technology from a beginner's perspective. You can accept corrections to your software better and leave the standard path if you need to. There are some good ideas even from people who don't have your experience.

Was there ever a software build twice the same way? Even if you copy software it is different somehow.

4 No Ego

Some programmers have a huge ego problem. But there is no time for that.

Who judges your quality as programmer? You? No. Others? Probably. But can you really compare an Apple with a Banana? No. You are an individual. You cannot compare yourself as a whole with another human being. You can only compare a few facets.

A facet is nothing that you can be proud of. You are good at Java? Cool. The other guy is not as good as you, but better with bowling. Is Java more important than bowling? It depends on the situation. Probably you earn more money with Java, but the other guy might have more fun in life because of his bowling friends.

Can you really be proud because you are a geek? Programmers with ego don't learn. Learn from everybody, from the experienced and from the noobs at the same time.

Kodo Sawaki once said: you are not important.

Think about it.

5 There is No Career Goal

If you want to gain something and don't care about your life now, you have already lost the game. Just act as well as you can, without looking at the goal you might reach down the road.

Working for twenty years to become a partner? Why aren't you working as hard as possible just because it is fun? Hard working can be fun. "A day without work is a day without food" is a Zen saying.

There is no need to accept happiness after twenty years. You can be happy right now, even when you are not a partner or don't drive a Porsche. Things change too easily. You could get sick.

> **“You must be mindful in your life. Otherwise you waste it.”**

You could get fired. You could burn out (if you follow all these items, I guess that likelihood is low).

Until these bad things happen, just work as well as you can and have fun with doing it. No reason to envy the gains of your colleagues. No reason to think about the cool new position you didn’t get.

After all, you will acheive something. You’ll end up with nice memories, maybe a good position — and twenty excellent years. Every day is a good day.

If you ever come to the point were you think that working at your company is no fun at all, you must leave immediately. Never stay at a company which takes away the happiness in your life. Of course, this is only possible in the rich countries, where people have the choice to leave. But if you are living in such an environment, do it. Leave without regret. You have no time to waste, you are probably dead tomorrow.

When you have no career goal, leaving is easy.

6 Shut Up

If you don’t have anything to say, don’t waste your colleagues’ time. This doesn’t make you look wimpy. Every day you work you need to try to not get on someone else’s nerves. Imagine if everybody tried this — what a great workplace would that be? Sometimes it is not possible. Try hard, you will like it.

If you don’t develop an ego, it is pretty easy to shut up and care about the areas you have something to talk about. Don’t confuse your ego with your “experience,” and always remember: you are a beginner. If somebody has a good idea, support the idea.

7 Mindfulness. Caring. Awareness.

Yes, you are working. But at the same time you are living and breathing. Even when you have some hard times at work, you need to listen to the signs of your body. You need to learn about the things which are good for you. This includes everything, including basic things like food. You need to care for yourself and for everything in your environment, because after all, the water you drink is the water which

runs in the river. Because you are living only for yourself. Remember that you live alone and you'll die alone. The world goes on without you.

Avoid work situations you don't like. Avoid working for free if it means you will have no fun and keeps you away from your bed. Let go of anything that doesn't make you happy. Think working for free sounds fun in theory? Consider the people doing Open Source in their spare time. If you have subscribed to some project's mailing list, you probably know what heat there is. If you don't have fun with that, stop doing it. I know a bunch of people who work in an Open Source environment they don't like. Again, with Time & Bill, I have tracked the time I spend in Open Source projects and was surprised how much time I lose there — especially on projects I didn't like very much.

Keeping this in mind, some people think they are only happy when they have free time and can spend the evening with an Xbox and some beer. While this is a good idea from time to time, it is not necessary that the only time that your life is fun. If you can avoid situations you don't like, avoid them (as I said earlier). But sometimes shit is unavoidable. Like for example manually copying and pasting stuff from your manager's Excel sheet into phpmyadmin. This can take you days, and it is really boring. It is no fun, but sometimes you need to do such things. You cannot always quit your job when you got a boring task. Zen monks are not shy about their work either. They get up at 4 AM (sometimes earlier, sometimes later, depending on the convent) and start meditation and work (they even consider work meditation practice). They have stuff to do like cleaning the toilets. Or working in the garden. Or as a Tenzo, they cook. They do it with all the care they can get. Whatever they do, they do it without suffering and they are (or should be) happy, because every second, even the second they are cleaning toilets, is a second of their life.

That being said: stop crying if you need to copy/paste in Excel. Just do it. Don't waste your energy with such things, they will pass. Become the best Excel copy/paster out there instead.

If you suffer a heart attack, people will probably say: "uh yes, he really worked too much, he even worked for me for free at night." Nobody can guide you to the other world. This last step is taken by you alone. You cannot take back anything in this world. So it is up to you to take care, in every second. If you die, you die. But when you live, you live. There is no time to waste.

"Care" is a huge word in Zen Buddhism, and I think in every form of Buddhism. I cannot express everything which needs to be said. It is difficult to understand the different meanings of "care." Probably you better understand the word "awareness." You must be aware of what you do, in every second of your life. You must be mindful in your life. Otherwise you waste it. But, of course, it is up to you to do so, if you like.

8 There is No Boss

Yes, there is somebody who pays you. There is somebody who tells you what needs to be done. And he can fire you. But this is no reason to give up your own life or to become sick of your work. Your Boss has no control over you. It can even be doubted that you have control about you — but let's not go down this path.

Back to your Boss: he can make your life worse if you allow him to do so. But there is a way out. Say "No" if you need to do something which makes you sick or is against your ethics. What will happen? In the worst case scenario, he will fire you. So what? If you live in western nations and if you are a coder (which is very likely when you read this) you'll get another job.

I don't mean to say "No" to tasks like copying CSV data to HTML. I am talking about eighty-hour weeks when you feel your body breaking. Or if you feel that your kids need some attention. Or if you are forced to fire people just because your Boss doesn't like them. Or if you are a consultant and get a job developing software for nuclear plants or for tanks. You can say "No."

9 Do Something Else

A programmer is more than a programmer. You should do something which has nothing to do with computers. In your free time, go sailing, fishing, or diving. Do meditation, martial arts, or play Shakuhachi. Whatever you do, do it with all the power you have left. Like you do at work. Do it seriously. A hobby is not just a hobby, it's an expression of who you are. Don't let anybody fool you when they say hobbies are not important. Nowadays it takes effort having hobbies. I have recorded several CDs and wrote fantasy books (the latter one unpublished, I must practice more). These things have made me into the person I am now, and finally they have led me to Zen and this article. These days I practice Zen Shakuhachi. It is a very important aspect to my daily life.

10 There is Nothing Special

A flower is beauty. But it's just a beautifu flower — nothing more. There is nothing special about it. You are a human who can program. Maybe you are good. There is nothing special about you. You are like me or all other people on this planet.

You need to go in the loo and you need to eat. Of course you need to sleep. After (hopefully) a long time you will die, and everything you have created will be lost. Even pyramids get lost, after a long time. Do you know the names of the people who built the pyramids? And if you do, is it important that you know? It's not. Pyramids are there, or they're not. Nothing special about that.

Same goes for your software. The bank is earning money with your software. After you leave, nobody remembers you. There is nothing wrong around it. It is the flow of time. Nothing you should be worrying about. If you are living according to the first nine rules, you'll see that this last project was a good and funny project. Now it's simply time to go on and concentrate on something else.

If your company closes because of financial problems, no problem. Life will go on. There is no real need for an Xbox, a car, or other belongings. Most people on this planet live in poverty. They don't care about having an Xbox, because they would be glad to get some food or even water.

So why exactly are you special? Because you had the luck to be born in the western world? Because you can code? No, there is nothing special about it. You can let go of your ego and live freely. Enjoy the colors and the smell of flowers around. Don't be too sad when the winter comes, and don't be too happy when spring comes back. It is just a flow. Keep in mind when somebody denies your application. Because the company is not so special that you need to be worried about the job. ■

Disclaimer

I am not a Zen monk. I am just practicing and learning. Please ask your local Zen monk if you feel there is something you need to understand further. ■

Christian is a developer since 1998. In 2006 he worked 75 hours a week. This made him start with Zen practice. Today he runs Time & Bill [timeandbill.de], studies psychology and tries to apply Zen to his daily life and work.

Reprinted with permission of the original author.
First appeared in *hn.my/zen* (grobmeier.de)

Background image by Dioma.
Zen circle image by DragonArtz.

Google tracks you. We don't.

How to Be Relentlessly Resourceful

By JASON SHEN

Relentlessly resourceful. This is the essential quality of a good startup founder, according to Paul Graham, co-founder of Y Combinator. When asked by Forbes what he looks for in founders, four out of the five elements related to resourcefulness. He has even written two essays dedicated to the concept.

And yet people don't seem to really understand what being resourceful means. The top comment on HN from his most recent post posed this question:

> *Yes, there are certain skills that make it easier to find information on your own. But this is also a function of the problem domain and how well you know it. If you give me a credit card and a problem statement, chances are that I can come up with a working webapp that solves the problem.*
>
> *But if you give me the name of a VC and tell me to go raise money — where do I start? How do I approach him? What will burn bridges and what won't?*

Some great HNers jumped in to answer that question, but I thought I'd take a crack at laying out, in full, what I believe being resourceful looks like and how someone can act with more relentless resourcefulness.

Let's start by talking about the two types of resourcefulness: internal and external.

- **Internal resourcefulness** is really just creativity. It's figuring out how to fit a cube into a cylinder on Apollo 13 or resolving that nasty bug in your code. You might benefit from the advice or perspective of others, but the resources you need to solve the problem are generally within your grasp (or inside your brain).

- **External resourcefulness** is when you need resources that are outside your control. Things like seed capital for your startup, a liquor license for your bar, a distribution channel for your new product. You will likely need to interact with other people/entities to *get* the resources you need to address your problem.

This article focuses more on that external resourcefulness because I think in some ways it's more open-ended and confusing, and academically/technically intelligent people often struggle to be externally resourceful.

Prerequisites

Before we begin, I think there are fundamental underlying conditions needed before someone can really be relentlessly resourceful.

Willingness to Endure Discomfort

I originally wanted to call this "guts" or "courage," but it's much more than this. It's being willing to approach people you feel you have no business talking to, asking for more than you feel wise asking for and doing work you might not like or feel competent in. If you can't or are unwilling to endure rejection, embarrassment, uncertainty, fear or failure, just close the window now because it's not happening.

Communication Skills

You don't need to be a world-class public speaker or best-selling author to be resourceful, but you need to have some threshold ability to communicate ideas clearly and persuasively to relevant audiences. This is definitely a skill you can develop — start a blog, join toastmasters, study copywriting, learn how to sell. If people struggle to understand you or are never convinced to do something you suggest, it's going to be really rough.

Grit/Not Quitting

Researchers at UPenn have found that grit (perseverance and passion for long-term goals) is a better predictor for success over IQ or conscientiousness. What you should draw from this is that you should have long-term goals you are really, really determined to achieve. Since you will face a lot of setbacks during the journey, don't start unless you have the bullheaded tenacity to finish.

The Formula

Alright, now that we've gotten that out of the way, here are the 3 things you need to do to be relentlessly resourceful:

1. **Learn enough to get clue**
2. **Actually take action**
3. **Repeat until you succeed**

1 Learn Enough To Get A Clue

Ok, so you have a challenge in front of you. Whether it is getting published as an author, starting a restaurant or destroying all the horcruxes hidden by He-Who-Shall-Not-Be-Named, you start by getting a lay of the land.

Lucky for us, there is an incredible treasure trove of information on the Internet (barring the passage of SOPA and PIPA) that we can dive through.

Google is your friend. Quora is your friend. Wikipedia, Twitter, Facebook, HN, the blogosphere. I assure that you can find the answer for many of the questions you have using one of these resources.

- Need to get startup capital from a venture capitalist? Mark Suster, a 2x entrepreneur turned VC will tell you how, for free! [hn.my/fund]
- Want to skip the line by bribing the Matire'd? Jonas Luster, a cook and cooking author will tell you how, for free! [hn.my/skip]
- Want to grow your blog audience? Tyler Terooven, a lifestyle blogger who came "out of nowhere" will tell you how, not for free, but I bought the guide and it's worth every penny. [hn.my/tyler]

Now, this is just the starting point. This online research is often enough to get you on the right path, but sometimes you've got problems that are more thorny, nuanced and specific. That's when you have to learn from people.

Unless you live under a rock, there is probably someone in your extended network who has done whatever it is you are trying to do (or something similar). Get in touch and ask for 10 minutes of his/her time.

Don't believe me? I dare you to post on Facebook, Twitter and in an email to 10 good friends:

> *Hey everyone!*
>
> *I really need your help with something! I'm looking to get in touch with someone who knows a lot about XX (or has done XX or something similar) for a really important project/goal/thing I'm working on.*
>
> *If you know someone who fits that profile (or know someone who might know) I would really appreciate if you could connect us. All help will be rewarded with cookies made by yours truly.*
>
> *Thanks so much!*

Do that, wait a few days and write back if you don't at least get *something*. I will send *you* cookies made by me if you draw a total blank.

Ok, fine, so you grew up in Siberia and literally only know 10 people. I bet you are still aware of someone "famous" who has done what you want to do — but they aren't in your network.

No problem. Let's go ask them for advice.

From these meetings you will start to get the nuanced, insider knowledge you need to get at whatever resource

you want. It might take some time and work to learn what you need to know, but information is almost *never* the limiting factor in being resourceful.

But what do I mean by "enough to get a clue"? The idea here is that you need to get some perspective. If you truly know nothing about a topic, you need to dive in enough until you understand at least a little bit about what's going on. Once you "have a clue,", you want to move to Step 2, where you start to really make progress.

It's important not to get stuck in the learning phase. You can "study" forever and never accomplish anything. In fact, many people do just that — they "study" fitness, dating techniques or personal finance forever and don't actually do anything. That, my friend, is death. Don't get stuck.

2 Actually Take Action

Alright, this is the most important step.

You gotta do a bunch of stuff. No way around it.

- If your goal is to raise funds for your startup, you could put together a deck, find a meetup with real investors attending and *actually* go talk to one of them about your business.
- If your goal is to throw a smashing dinner party but you can't cook, you could find a basic recipe online, buy the ingredients from the store and *actually* follow the instructions to make a dish.
- If your goal is to get a girlfriend, you could throw on some nice clothes, walk over to a bar or lounge and *actually* have a conversation with a girl.
- If your goal is to get published as a novelist, you could map out an outline of the story and *actually* write the first chapter.

A rule of thumb: if you aren't feeling uncomfortable, then you haven't gone far enough yet.

Resourceful people take action. It's not that they don't think, plan, study, strategize or prepare. They do all those things too. What separates people who really "make things happen" and analysts is action.

Think about your favorite hero. Ender Wiggin. Harry Potter. Lisbeth Salandar. Bruce Wayne. The reason why we love these characters is because they face up to enormous odds and they win through their resourcefulness and courage. They don't cower in the face of a challenge, they take action and make things happen.

Because I know what you're thinking, I've prepared a handy FAQ:

Q: How do I know what to do?
A: You did step one right? So you have a clue! What makes sense? What action seems like a reasonable way to get closer to your final step? Chances are you know exactly what the next step is, so the real issue is "Why aren't you doing what you know you should?"

Q: Taking action is scary! Wouldn't it be better to learn more until this problem gets less scary?
A: It's always going to be scary. Courage is not the absence of fear. Courage is feeling the fear and doing it anyway. Learning indefinitely will not solve your problems.

Q: But what if I get rejected/make a mistake/fail? That'll ruin everything and then my life will be over!
A: Unless you are learning how to pack your own parachute before sky diving, I promise you will almost certainly *not* die if you mess up. You will be mildly embarrassed, maybe set back a few bucks or some period of time, and that's pretty much it. Most people will forget about your mishap almost immediately after it happens. People just don't care that much about you.

Q: I'm doing lots of stuff but still not making progress. I'm making spreadsheets, organizing data into a wiki, mapping out the competitors, having conversations over beers with my friends…
A: You're doing fake work. This is why I said you should feel uncomfortable with the actions you're taking. Making charts is easy and safe. You've got to be out on the line of fire. If you can't fail then it doesn't count as action.

3 Repeat Until You Succeed

So you did some real stuff. Some of it worked, much of it didn't. Now what?

Time to learn again. What lessons can you draw from your experience to inform your next try? What can you do differently or do better?

Ok, now go do that. How did it go? Any surprises? What new angle can you try? What worked that you can double down on? How can you avoid making that mistake next time? Ok, now try again.

The magic of the doing-learning loop is that momentum builds upon itself. The first time you ski you fall a ton, but as you start to figure out what's going on, you fall less and less until you're flying down the mountain. It's only through *doing* that you figure out what *not* to do next time.

So, if the first five investors turn you down, tweak your pitch and try again. If that doesn't work, maybe you need to get more traction. Maybe you need to get a warm intro. Maybe you need to use AngelList. Maybe you need to go through YC. May you need to get on Techcrunch. Maybe you need to do some consulting and bootstrap. Maybe you need to do a Kickstarter.

Keep trying stuff, tweaking, asking questions, getting advice/ideas, experimenting and pushing forward until you find something that works. Then build on that and add fuel to the fire. Don't take "no" for an answer, ever. ■

Jason Shen is cofounder of Ridejoy, a Y Combinator-funded community marketplace for ridesharing. His blog, The Art of Ass-Kicking [jasonshen.com], has been covered in Lifehacker, ReadWriteWeb, and of course, Hacker Monthly.

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

It Takes 3 Years to Build a Business

By JACQUES MATTHEIJ

IT TAKES TIME. Three years to be precise.

I have absolutely no idea why it has to take 3 years, but it seems to be about right, based on countless observations of people that start a business and how long it takes them to be successful. One man consultancy shops, airlines, and everything in between: 3 years. Sometimes a bit less, sometimes a bit more.

Typically it goes like this:

- In the first year you lose money. Earlier on more so than later in the year.
- Somewhere during the second year you break even.
- In the third year you finally make back all your initial capital.

The reason why it works this way is simple enough: customer acquisition is a time-consuming process, and a new business lacks several things that help in gaining new customers: a reputation, a network, and existing customers.

It's like being the guy or girl without a partner. If you don't have a partner, it is hard to find one, but once you have one, everybody seems to flirt with you.

A new business is like that. People might even consider doing business with you, but nobody wants to be "first" for fear of being burned. So they will play it safe and choose someone that already has a reputation and existing customers, and they'll find them through their extensive network of contacts built up over the years.

You initially don't stand a chance. So it is hardly surprising that the author of that piece found it difficult to get enough work at a high enough rate for his consultancy business to get off the ground. In another 6 months, he'd be further in the hole, but he would have had a bit more traction.

Once you get that first customer, you are on your way to an eventual success. One customer will lead to another, which is the basis of your network. Happy customers are references which you can use to cement your reputation. And once that wheel starts turning, it will speed up. And before the third year is out you'll be in demand to the point where you will probably have to raise your rates to control the influx of new customers. But it takes time to get there and the first year is terrible.

The bad news here is: if you plan on quitting your job to start a consultancy business and you have less than 18 months worth of expenses in your savings account, then you shouldn't do it, unless you have a very large amount of work lined up. And even then you'll need to be more frugal than Ebenezer Scrooge, in case any of it dries up or a customer doesn't pay on time (or at all!).

Another option is to start your company on the side, while you have a regular job that pays for your basic needs. Enjoy that you essentially have infinite runway, even if the take-off will be slower because a lot of your good time is already spoken for. Some jobs are more flexible than others in this respect and therefore more suitable for the purpose.

Once you have enough money salted away to deal with unforeseen circumstances, economic dips, and the like, you can always fire your boss. But don't do it too soon, or you will fail. ■

Jacques Mattheij is the inventor of the live streaming webcam, founder of camarades.com / ww.com and a small time investor. He also collects insightful comments from Hacker News.

Reprinted with permission of the original author. First appeared in *hn.my/3years* (jacquesmattheij.com)

S
CH
LEP

BLINDNESS

By PAUL GRAHAM

THERE ARE GREAT startup ideas lying around unexploited right under our noses. One reason we don't see them is a phenomenon I call schlep blindness. Schlep was originally a Yiddish word but has passed into general use in the US. It means a tedious, unpleasant task.

No one likes schleps, but hackers especially dislike them. Most hackers who launch startups wish they could do it by just writing some clever software, putting it on a server somewhere, and watching the money roll in — without ever having to talk to users, negotiate with other companies, or deal with other people's broken code. Maybe that's possible, but I haven't seen it.

One of the many things we do at Y Combinator is teach hackers about the inevitability of schleps. No, you can't start a startup by just writing code. I remember going through this realization myself some time in 1995. I soon learned from experience that schleps are not merely inevitable, but pretty much what business consists of. A company is defined by the schleps it will undertake. And schleps should be dealt with the same way you'd deal with a cold swimming pool: just jump in. This is not to say you should seek out unpleasant work per se, but that you should never shrink from it if it's on the path to something great.

The most dangerous thing about our dislike of schleps is that much of it is unconscious. Your unconscious won't even let you see ideas that involve painful schleps. That's schlep blindness.

The phenomenon isn't limited to startups. Most people don't consciously decide not to be in as good physical shape as Olympic athletes, for example. Their unconscious mind decides for them, shrinking from the work involved.

The most striking example I know of schlep blindness is Stripe [stripe.com] or rather Stripe's idea. For over a decade, every hacker who'd ever had to process payments online knew how painful the experience was. Thousands of people must have known about this problem. And yet when they started startups, they decided to build recipe sites or aggregators for local events. Why? Why work on problems few care much about and no one will pay for when you could fix one of the most important components of the world's infrastructure? Because schlep blindness prevented people from even considering the idea of fixing payments.

Probably no one who applied to Y Combinator to work on a recipe site began by asking "should we fix payments or build a recipe site?" and chose the recipe site. Though the idea of fixing payments was right there in plain sight, they never saw it because their unconscious mind shrank from

the complications involved. You'd have to make deals with banks. How do you do that? Plus, you're moving money, so you're going to have to deal with fraud and people trying to break into your servers. Additionally, there are probably all sorts of regulations to comply with. It's a lot more intimidating to start a startup like this than a recipe site.

That scariness makes ambitious ideas doubly valuable. In addition to their intrinsic value, they're like undervalued stocks in the sense that there's less demand for them among founders. If you pick an ambitious idea, you'll have less competition because everyone else will have been frightened off by the challenges involved. This is also true of starting a startup in general.

How do you overcome schlep blindness? Frankly, the most valuable antidote to schlep blindness is probably ignorance. Most successful founders would probably say that if they'd known about the obstacles they'd have to overcome when they were starting their company, they might never have started it. Maybe that's one reason the most successful startups so often have young founders.

In practice, the founders grow with the problems. But no one seems able to foresee this, not even older, more experienced founders. So the reason younger founders have an advantage is that they make two mistakes that cancel each other out. They don't know how much they can grow, but they also don't know how much they'll need to. Older founders only make the first mistake.

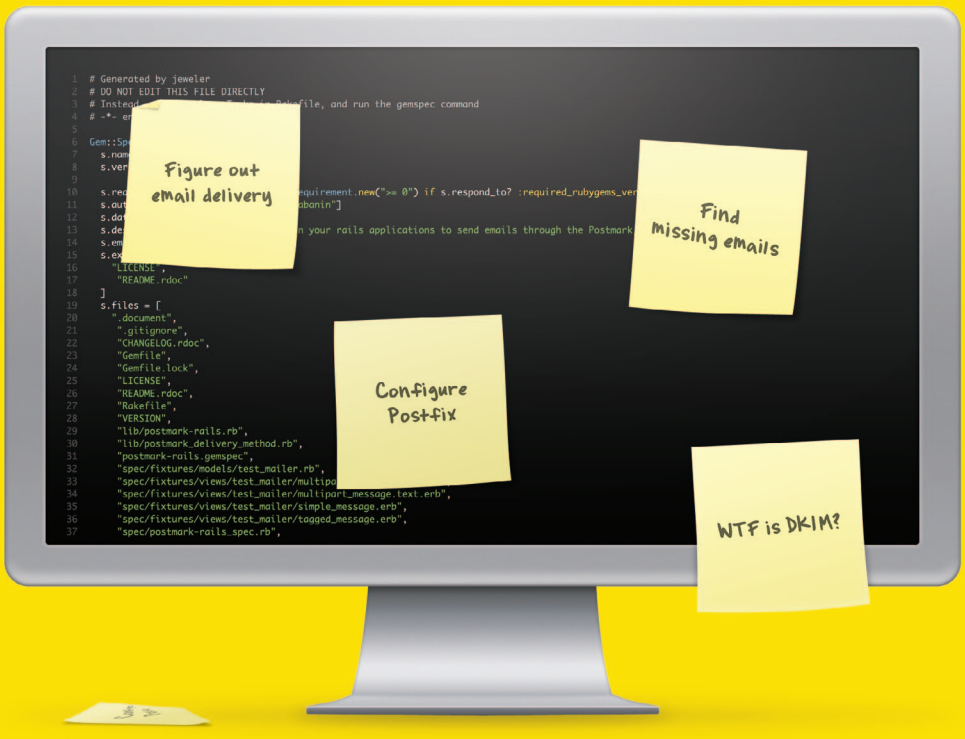
Ignorance can't solve everything though. Some ideas so obviously entail alarming schleps that anyone can see them. How do you see ideas like that? The trick I recommend is to take yourself out of the picture. Instead of asking "what problem should I solve?" ask "what problem do I wish someone else would solve for me?" If someone who had to process payments before Stripe had tried asking that, Stripe would have been one of the first things they wished for.

It's too late now to be Stripe, but there's plenty still broken in the world, if you know how to see it. ■

Paul Graham is an essayist, programmer, and programming language designer. He's currently working on a new programming language called Arc, a new book on startups, and is one of the partners in Y Combinator.

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

If you enjoy configuring Postfix and Sendmail, ignore this.



We'll deal with that so you can get back to coding.

Postmark

hacker.postmarkapp.com

Transactional email delivery for web apps.

Astonishments in the History of Version Control

By FRANCIS IRVING

> **"If you really want…truly ancient history, you have to go back to delta decks on punch cards."**
>
> **— Jim Rootham**

In a world where biographies of cod are not just accepted, but rightly popular, it wouldn't seem entirely crazy to write a history book on how computer programmers store the vital product of their labors: source code.

Since neither you nor I have time to read or write such a thing, we're going to have to settle on this one article.

It's an important subject. The final product (GitHub) seems incredibly obvious. And popular. Yet it took decades of iterative innovation, from some of the cleverest minds in the field, to make something so apparently simple yet powerful.

And every step was astonishing.

1 Source code is text in a file! (1960s)

With hindsight, it's obvious that source code is best stored as just writing in simple documents. A brief read of the history of ASCII [hn.my/ascii] gives a flavor for the complexity of agreeing even that.

2 Humans can manually keep track of versions of code! (1960s)

As with everything, in the beginning there was no software.

> *"At my first job, we had a Source Control department. When you had your code ready to go, you took your floppy disks to the nice ladies in Source Control, they would take your disks, duly update the library, and build the customer-ready product from the officially reposed source." (Miles Duke)*

3 You can keep lots of versions in one file! (1972, 1982)

Using a fancy interleaved weave file format, SCCS ruled the roost of version control for a decade.

It took some years to develop a good method for recording the changes from one version of a file to the next. "An Algorithm for Differential File Comparison" [hn.my/diff] is a relatively late paper to read on the subject (1976).

In 1982, SCCS's successor RCS used these diffs in reverse to beat SCCS, and astonished this commenter:

> *"Along came RCS with its reverse-deltas, and I thought it was the bee's knees." (Anonymous)*

4 You can each have your own copy checked out! (1982)

At the time, people tended to log into a central mainframe and work together via that. With RCS, using symbolic links, it could be arranged so that each person was working with the same version control, but with their own working copy.

> *"There will be a file called RCS that is a symbolic link to the master RCS repository that you share with the rest of your group members." (Information on Using RCS at Yale)*

5 Wow! You can version multiple files at once! (1986)

Amazingly, up until CVS, each version control system was for separate individual files. Yes, you can use RCS with wildcards to commit multiple files, or mark particular branches. But it isn't really part of the system.

In CVS it was the default to modify all the files recursively. Software was suddenly a recursive tree of text files, rather than just a directory or an individual file.

It was badly implemented, as it wasn't "atomic" (successor Subversion fixed this in 2000), but really that doesn't matter for the purpose of astonishment.

6 Two people can edit the same file at the same time, and it merges what they both did! (1986)

In the late 1990s I worked at Creature Labs. We were changing from Visual SourceSafe (commercial, made by Microsoft) to CVS (open source, made by a bunch of hippies).

There was frankly disbelief that it could do its main magical promise: let multiple people edit the same file at the same time, and be able to flawlessly merge their changes together without breaking anything.

The exclusive locking of SourceSafe was a real problem when we were making Creatures 3. I remember a particular occasion when we were adding garbage collection, which meant editing most code files, and the lead programmer had to check out every file exclusively over the weekend while he implemented it.

This paper [hn.my/grune] from 1986 is an excellent historical record of this magic, wherein Dick Grune suffers the same problem while his team codes a compiler in Holland, and so he invents CVS.

7 The shared repository can be on a remote machine! (1994)

Most of this time people were mainly using version control on one computer. Some versions of RCS, and hence CVS, had a remote file sharing mechanism to let you have a remote code repository in 1986.

> *"If a version of RCS is used that can access files on a remote machine, the repository and the users can all be on different machines." (Dick Grune)*

But it looks like it was only in 1994, when a TCP/IP protocol added, that the idea really took off.

> *"[CVS] did not become really ubiquitous until after Jim Blandy and Karl Fogel (later two principals of the Subversion project) arranged the release of some patches developed at Cygnus Software by Jim Kingdon and others to make the CVS client software usable on the far end of a TCP/IP connection from the repository." (Eric Raymond)*

8 Free open source version control hosting! (1999)

This isn't an advance in source control technology, but it was astonishing, and on the Internet, social advances can be as important as technical ones.

> *The tendency was for older OSS versions to be hard to find. John T. Hall had the insight that if projects were developed on the site, the old versions would be there by default. A development platform service was audacious, but no one else was doing it, and we thought "why not?" (Brian Biles)*

Partying like there was no tomorrow (for their stock), VA Linux introduced SourceForge to the world. This was great for new projects (like my TortoiseCVS).

It was hard and expensive to get a server on the Internet back then, and it wasn't easy or cheap to set up source control and a bug tracker. This new service, despite its lack of business model, fledged numerous projects that bit earlier.

9 You can distribute it all so there's no central repository! (2005)

There was a wave of version control systems in the early nineties, making version control completely distributed.

That is, your local machine has an entire copy of the history of the code, and can easily branch and merge on a peer-to-peer basis with any other copy of it. By the way, the same feature makes it much easier to branch and merge in general.

Given that, it seems unfair that I've dated this astonishment in 2005. That's because I'm not recording the first time anyone made the astonishing thing, but rather the first time it was productized and made popular. April 2005 was when both Mercurial and Git were released.

The post "The Risks of Distributed Version Control" (late 2005) [hn.my/risk] shows how radical this new-fangled stuff was seen to be.

10 When you checkout that's a fork, too, and you can do that in public! (2008)

GitHub is successful for several reasons.

In the context of this article, the astonishment was that you might want to make even your tiny hacks to other people's code public. Before GitHub, we tended to keep those on our own computer.

Nowadays, it is so easy to make a fork, or even edit code directly in your browser, that potentially anyone can find immediately, even your least polished bug fixes.

Coda

Have a quick look back up at those decades of progress. Yes, some of the advances were also enabled by increasing computer power. But mainly, they were simply made by people thinking of cleverer ways of collaborating.

It makes me wonder, what is next? What new astonishing thing will happen in version control?

More broadly, can the same thing happen in other fields?

Are core parts of our information infrastructure that ultimately block innovation in government or healthcare or journalism or data as capable of such dramatic improvement?

I have this feeling we're going to find out. ■

Reprinted with permission of the original author.
First appeared in *hn.my/cvs* (flourish.org)

Francis Irving, CEO of ScraperWiki, lives in Liverpool, UK. He was the founding developer of mySociety, which over the last 8 years has made the world's most innovative democracy websites. He created TortoiseCVS.

A Programming Idiom You've Never Heard Of

By JAMES HAGUE

Here are some sequences of events:

1. Take the rake out of the shed, use it to pile up the leaves in the backyard, and then put the rake back in the shed.
2. Fly to Seattle, see the sights, and then fly home.
3. Put the key in the door, open it, and then take the key out of the door.
4. Wake-up your phone, check the time, and then put it back to sleep.

See the pattern? You do something, do something else, and then you undo the first thing. Or more accurately, the last step is the inverse of the first. Once you're aware of this pattern, you'll see it everywhere. Pick up the cup, take a sip of coffee, and put the cup down. And it's all over the place in code, too:

1. Open a file, read the contents, and then close the file.
2. Allocate a block of memory, use it for something, and then free it.
3. Load the contents of a memory address into a register, modify it, and then store it back in memory.

While this is easy to explain and give examples of, it's not simple to implement. All we want is an operation that looks like `idiom(Function1, Function2)`, so we could write the "open a file..." example above as `idiom(Open, Read)`. The catch is that there needs to be a programmatic way to determine that the inverse of "open" is "close." Is there a programming language where functions have inverses?

Surprisingly, yes: J [jsoftware.com]. And this idiom I keep talking about is even a built-in function in J, called *under*. In English, and not J's terse syntax, the open file example is stated as "read under open."

One non-obvious use of *under* in J is to compute the magnitude of a vector. Magnitude is an easy algorithm: square each component, sum them up, and then take the square root of the result. Hmmm...the third step is the inverse of the first. "Sum under square," or as it is written in actual J code:

```
mag =: +/ &.: *:
```

In the above example, `+/` is "sum," `&.:` is "under," and `*:` is "square." ■

- Read the followup here: *prog21.dadgum.com/122.html*

James Hague is a recovering programmer who now works full time as a game designer, most recently acting as Design Director for Red Faction: Guerrilla. He's run his own indie game studio and is a published photographer.

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

Building a Modern Web Stack for the Real-Time Web

By ILYA GRIGORIK

The web is evolving. After a few years of iteration the WebSockets spec is finally here (RFC 6455), and as of late 2011 both Chrome and Firefox are SPDY capable. These additions are much more than just "enhancing AJAX," as we now have true real-time communication in the browser: stream multiplexing, flow control, framing, and significant latency and performance improvements. Now, we just need to drag our "back office" — web frontends, app servers, and everything in between — into this century to take advantage of these new capabilities.

We're optimized for "Yesterday's Web"

Modern backend architecture should allow you to terminate the user connection as close to the user as possible to minimize latency — this is your load balancer or web server occupying ports `80` and `443 (SSL)`. From there, the request is routed on the internal network from the frontend to the backend service, which will generate the response. Unfortunately, the current state of our "back office" routing is not only outdated, but often it is also the limiting factor in our adoption of these real-time protocols.

WebSockets and SPDY are both multiplexed protocols, which are optimized to carry multiple, interleaved streams of data over the same TCP pipe. Unfortunately, popular choices, such as Apache and Nginx, have no understanding of this and at best

degrade to dumb "TCP proxies." Even worse, since they do not understand multiplexing, stream flow-control and priority handling goes out the door as well. Finally, both WebSockets and SPDY communicate in framed messages, not in TCP streams, which need to be re-parsed at each stage.

Put all of this together and you quickly realize why your own back office web stack, and even the popular platforms such as Heroku and Google's App Engine are unable to provide WebSockets or SPDY support: our services are fronted by servers and software which was designed for yesterday's web.

Architecture for the "Real-Time Web"

HTTP is not going away anytime soon, and we will have to support both the old and new protocols for some time to come. One attempt at this has been the `SPDY > HTTP` proxy, which converts a multiplexed stream into a series of old-fashioned HTTP requests. This works, and it allows us to reuse our old infrastructure, but this is exactly backwards from what we need to be doing!

Instead of converting an optimized, multiplexed stream into a series of internal HTTP dispatches, we should be asking for `HTTP > SPDY` infrastructure, which would allow us to move beyond our outmoded architectures. In 2012, we should demand our internal infrastructure to offer the following:

- Request and Response streaming should be the default
- Connections to backend servers should be persistent
- Communication with backend servers should be message-oriented
- Communication between clients and backends should be bi-directional

Make SPDY the default, embrace dynamic topologies

The first step towards these goals is to recognize that translating SPDY to HTTP is a convenient path in the short term, but exactly the wrong path in the long term. SPDY offers multiplexing, flow control, optimized compression, and framing. We should embrace it and make it the default on the backend. Once we have a multiplexed, message-oriented protocol on the backend, we can also finally stop reparsing the same TCP stream on every server. Writing HTTP parsers in 2012 is neither fun nor an interesting problem.

Finally, this architecture should not require a dedicated OPS team or a custom software platform to maintain. Modern web applications are rarely powered by a single host and require dynamic (re)configuration and management. Services such as Heroku, CloudFoundry, and GAE have built their own "routing fabrics" to handle these problems. Instead, we need to design architectures where the frontends and the backends are decoupled by default and require minimal intervention and maintenance.

Adopt a modern Session Layer

Building dynamic network typologies is not for the faint of heart, especially once we add the additional requirements for message-oriented communication, multiplexed streams, and a grab bag of performance constraints. Thankfully, libraries such as ØMQ offer all of the above and more, all wrapped behind a simple and an intuitive API. Let the frontend parse and emit SPDY frames, and then route them internally as ØMQ messages to any number of subscribers.

Mongrel2 was one of the first web servers to explore this type of architecture with ØMQ, which allowed it to sidestep the entire problem of backend configuration, as well as enable a number of interesting worker topology patterns. There is still room for improvement, but it is a much needed step in the right direction. As a concrete example, let's consider a sample workflow with SPDY and ØMQ:

1. An HTTP (or SPDY) request arrives to the frontend
2. Frontend parses the request and generates `SYN_STREAM`, `HEADERS`, and `DATA SPDY` frames
3. The messages are delivered into a `PUSH` ØMQ socket (ala Mongrel2)
4. Backend subscribers use a `PULL` socket to process the SPDY stream
5. Backend subscribers stream a response back to the frontend

The communication is done over a persistent channel with message-oriented semantics, the frontend and the backends are completely decoupled, and we can finally stop punching "TCP holes" in our networks to support the modern web.

Supporting HTTP 2.0 in the back office

The new protocols are here, but the supporting "back office" architecture requires a serious update: SSL is becoming the default, streaming is no longer an option, and long-lived persistent connections are in. SPDY is gaining momentum, and I have no doubts that in the not so distant future it will be an IETF-approved protocol. Similarly, ØMQ is not the only alternative for internal routing, but it is definitely one that has been gaining momentum.

Fast HTTP parsing and routing is simply not enough to support the modern web use cases. Likewise, punching "TCP holes" in our infrastructure is not a viable long-term solution — in 2012 we should be asking for more. Yes, I'm looking at you Varnish, Nginx, Apache, and friends. ■

Ilya Grigorik is a web engineer, an open-source and Ruby evangelist, a data geek, and a proverbial early adopter of all things digital. He is currently helping lead the social analytics efforts at Google. Earlier, Ilya was the founder and CTO of PostRank, a social analytics company, which was acquired by Google.

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

Static Code Analysis

By JOHN CARMACK

THE MOST IMPORTANT thing I have done as a programmer in recent years is aggressively pursue static code analysis. Even more valuable than the hundreds of serious bugs I have prevented with it is the change in mindset about the way I view software reliability and code quality.

It is important to say right up front that quality isn't everything, and acknowledging it isn't some sort of moral failing. Value is what you are trying to produce, and quality is only one aspect of it, intermixed with cost, features, and other factors. There have been plenty of hugely successful and highly regarded titles that were filled with bugs and crashed a lot; pursuing a Space Shuttle style code development process for game development would be idiotic. Still, quality does matter.

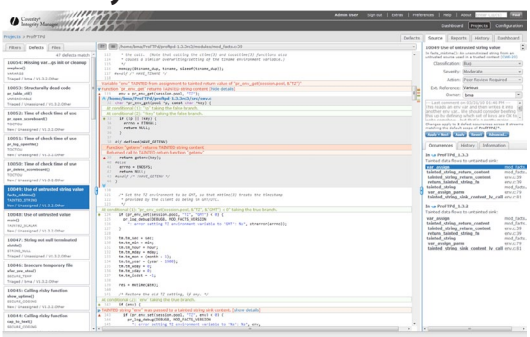
I have always cared about writing good code; one of my important internal motivations is that of the craftsman, and I always want to improve. I have read piles of books with dry chapter titles like "Policies , Standards, and Quality Plans," and my work with Armadillo Aerospace has put me in touch with the very different world of safety-critical software development.

Over a decade ago, during the development of Quake 3, I bought a license for PC-Lint and tried using it — the idea of automatically pointing out flaws in my code sounded great. However, running it as a command line tool and sifting through the reams of commentary that it produced didn't wind up winning me over, and I abandoned it fairly quickly.

Both programmer count and codebase size have grown by an order of magnitude since then, and the implementation language has moved from C to C++, all of which contribute to a much more fertile ground for software errors. A few years ago, after reading a number of research papers about modern static code analysis, I decided to see how things had changed in the decade since I had tried PC-Lint.

At this point, we had been compiling at warning level 4 with only a very few specific warnings disabled, and warnings-as-errors forced programmers to abide by it. While there were some dusty reaches of the code that had years of accumulated cruft, most of the code was fairly modern. We thought we had a pretty good codebase.

Coverity



Initially, I contacted Coverity [coverity.com] and signed up for a demo run. This is serious software, with the licensing cost based on total lines of code, and we wound up with a quote well into five figures. When they presented their analysis, they commented that our codebase was one of the cleanest of its size they had seen (maybe they tell all customers that to make them feel good), but they presented a set of about a hundred issues that were identified. This was very different than the old PC-Lint run. It was very high signal-to-noise ratio — most of the issues highlighted were clearly incorrect code that could have serious consequences.

This was eye-opening, but the cost was high enough that it gave us pause. Maybe we wouldn't introduce that many new errors for it to catch before we ship.

Microsoft /analyze

I probably would have talked myself into paying Coverity eventually, but while I was still debating it, Microsoft preempted the debate by incorporating their /analyze [hn.my/analyze] functionality into the 360 SDK. /analyze was previously available as part of the top-end, ridiculously expensive version of Visual Studio, but it was now available to every 360 developer at no extra charge. I read into this that Microsoft feels that game quality on the 360 impacts them more than application quality on Windows does.

Technically, the Microsoft tool only performs local analysis, so it should be inferior to Coverity's global analysis, but enabling it poured out mountains of errors, far more than Coverity reported. True, there were lots of false positives, but there was also a lot of scary, scary stuff.

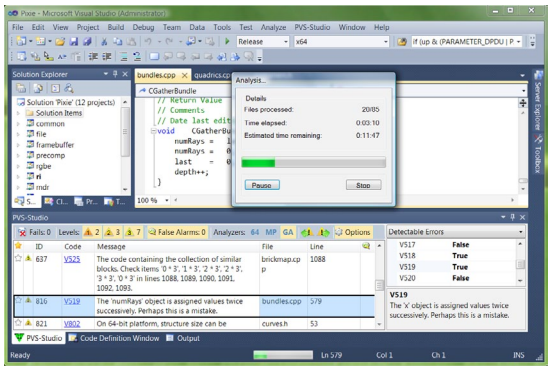
I started slowly working my way through the code, fixing up first my personal code, then the rest of the system code, then the game code. I would work on it during odd bits of free time, so the entire process stretched over a couple months. One of the side benefits of having it stretch out was that it conclusively showed that it was pointing out some very important things

— during that time there was an epic multi-programmer, multi-day bug hunt that wound up being traced to something /analyze had flagged, but I hadn't fixed yet. There were several other, less dramatic cases where debugging led directly to something already flagged by /analyze. These were real issues.

Eventually, I had all the code used to build the 360 executable compiling without warnings with /analyze enabled, so I checked it in as the default behavior for 360 builds. Every programmer working on the 360 was then getting the code analyzed every time they built, so they would notice the errors themselves as they were making them, rather than having me silently fix them at a later time. This did slow down compiles somewhat, but /analyze is by far the fastest analysis tool I have worked with, and it is oh so worth it.

We had a period where one of the projects accidentally got the static analysis option turned off for a few months, and when I noticed and re-enabled it, there were piles of new errors that had been introduced in the interim. Similarly, programmers working just on the PC or PS3 would check in faulty code and not realize it until they got a "broken 360 build" email report. These were demonstrations that the normal development operations were continuously producing these classes of errors, and /analyze was effectively shielding us from a lot of them.

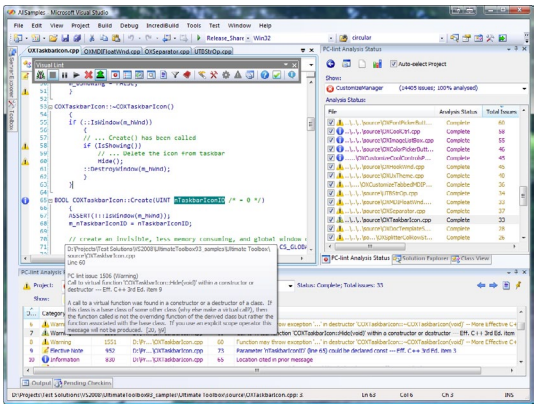
PVS-Studio



Because we were only using /analyze on the 360 code, we still had a lot of code not covered by analysis — the PC and PS3 specific platform code and all the utilities that only ran on the PC.

The next tool I looked at was PVS-Studio [hn.my/pvs]. It has good integration with Visual Studio and a convenient demo mode (try it!). Compared to /analyze, PVS-Studio is painfully slow, but it pointed out a number of additional important errors, even on code that was already completely clean to /analyze. In addition to pointing out things that are logically errors, PVS-Studio also points out a number of things that are common patterns of programmer error, even if it is still completely sensible code. This is almost guaranteed to produce some false positives, but damned if we didn't have instances of those common error patterns that needed fixing.

PC-Lint



Finally, I went back to PC-Lint [hn.my/pcl], coupled with Visual Lint [hn.my/vl] for IDE integration. In the grand UNIX tradition, it can be configured to do just about anything, but it isn't very friendly and generally doesn't "just work." I bought a five-pack of licenses, but it has been problematic enough that I think all the other developers that tried it gave up on it. The flexibility does have benefits — I was able to configure it to analyze all of our PS3 platform specific code, but that was a tedious bit of work.

Once again, even in code that had been cleaned by both /analyze and PVS-Studio, new errors of significance were found. I made a real effort to get our codebase lint clean, but I didn't succeed. I made it through all the system code, but I ran out of steam when faced with all the reports in the game code. I triaged it by hitting the classes of reports that I worried most about and ignored the bulk of the reports that were more stylistic or potential concerns.

Trying to retrofit a substantial codebase to be clean at maximum levels in PC-Lint is probably futile. I did some "green field" programming where I slavishly made every picky lint comment go away, but it is more of an adjustment than most experienced C/C++ programmers are going to want to make. I still need to spend some time trying to determine the right set of warnings to let us get the most benefit from PC-Lint.

Discussion

I learned a lot going through this process. I fear that some of it may not be easily transferable, that without personally going through hundreds of reports in a short amount of time and getting that sinking feeling in the pit of your stomach over and over again, "we're doing OK" or "it's not so bad" will be the default responses.

The first step is fully admitting that the code you write is riddled with errors. That is a bitter pill to swallow for a lot of people, but without it, most suggestions for change will be viewed with irritation or outright hostility. You have to want criticism of your code.

Automation is necessary. It is common to take a sort of smug satisfaction in reports of colossal failures of automatic systems, but for every failure of automation, the failures of humans are legion. Exhortations to "write better code" plans for more code reviews, pair programming, and so on just don't cut it, especially in an environment with

dozens of programmers under a lot of time pressure. The value in catching even the small subset of errors that are tractable to static analysis every single time is huge.

I noticed that each time PVS-Studio was updated, it found something in our codebase with the new rules. This seems to imply that if you have a large enough codebase, any class of error that is syntactically legal probably exists there. In a large project, code quality is every bit as statistical as physical material properties — flaws exist all over the place, you can only hope to minimize the impact they have on your users.

The analysis tools are working with one hand tied behind their back, being forced to infer information from languages that don't necessarily provide what they want, and generally making very conservative assumptions. You should cooperate as much as possible — favor indexing over pointer arithmetic, try to keep your call graph inside a single source file, use explicit annotations, etc. Anything that isn't crystal clear to a static analysis tool probably isn't clear to your fellow programmers, either. The classic hacker disdain for "bondage and discipline languages" is short-sighted — the needs of large, long-lived, multi-programmer projects are just different than the quick work you do for yourself.

NULL pointers are the biggest problem in C/C++, at least in our code. The dual use of a single value as both a flag and an address causes an incredible number of fatal issues. C++ references should be favored over pointers whenever possible; while a reference is "really" just a pointer, it has the implicit contract of being not-NULL. Perform NULL checks when pointers are turned into references, then you can ignore the issue thereafter. There are a lot of deeply ingrained game programming patterns that are just dangerous, but I'm not sure how to gently migrate away from all the NULL checking.

printf format string errors were the second biggest issue in our codebase, heightened by the fact that passing an `idStr` instead of `idStr::c_str()` almost always results in a crash. However, annotating all our variadic functions with /analyze annotations so they are properly type checked kills this problem dead. There were dozens of these hiding in informative warning messages that would turn into crashes when some odd condition triggered the code path, which is also a comment about how the code coverage of our general testing was lacking.

A lot of the serious reported errors are due to modifications of code long after it was written. An incredibly common error pattern is to have some perfectly good code that checks for NULL before doing an operation, but a later code modification changes it so that the pointer is used again without checking. Examined in isolation, this is a comment on code path complexity, but when you look back at the history, it is clear that it was more a failure to communicate preconditions clearly to the programmer modifying the code.

By definition, you can't focus on everything, so focus on the code that is going to ship to customers, rather than the code that will be used internally. Aggressively migrate code from shipping to isolated development projects. There was a paper recently that noted that all of the various code quality metrics correlated at least as strongly with code size as error rate, making code size alone give essentially the same error-predicting ability. Shrink your important code.

If you aren't deeply frightened about all the additional issues raised by concurrency, you aren't thinking about it hard enough.

It is impossible to do a true control test in software development, but I feel the success that we have had with code analysis has been clear enough that I will say plainly: It is irresponsible to not use it. There is objective data in automatic console crash reports showing that Rage, despite being bleeding edge in many ways, is remarkably more robust than most contemporary titles. The PC launch of Rage was unfortunately tragically flawed due to driver problems — I'll wager AMD does not use static code analysis on their graphics drivers.

The takeaway action should be: If your version of Visual Studio has /analyze available, turn it on and give it a try. If I had to pick one tool, I would choose the Microsoft option. Everyone else working in Visual Studio, at least give the PVS-Studio demo a try. If you are developing commercial software, buying static analysis tools is money well spent.

A final parting comment from Twitter:

> *The more I push code through static analysis, the more I'm amazed that computers boot at all.*
>
> **— Dave Revell (@dave_revell)**

John Carmack is a founder and technical director of Id Software and Armadillo Aerospace.

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

Practical Garbage Collection

By PETER SCHULLER

WHY SHOULD ANYONE have to care about the garbage collector?

That is a good question. The perfect garbage collector would do its job without a human ever noticing that it exists. Unfortunately, there exists no known perfect (whatever perfection means) garbage collection algorithm. Further, the selection of garbage collectors practically available to most people is additionally limited to a subset of garbage collection algorithms that are in fact implemented. (Similarly, malloc is not perfect either and has its issues, with multiple implementations available with different characteristics. However, this article is not trying to contrast automatic and explicit memory management, although that is an interesting topic.)

The reality is that, as with many technical problems, there are some trade-offs involved. As a rule of thumb, if you're using the freely available Hotspot based JVM:s (Oracle/Sun, OpenJDK), you mostly notice the garbage collector if you care about latency. If you do not, chances are the garbage collector will not be a bother — other than possibly to select a maximum heap size different from the default.

By latency, in the context of garbage collection, I mean pause times. The garbage collector needs to pause the application sometimes in order to do some of its work; this is often referred to as a stop-the-world pause (the "world" being the observable universe from the perspective of the Java application, or mutator in GC speak (because it is mutating the heap while the garbage collector is trying to collect it). It is important to note that while all practically available garbage collectors impose stop-the-world pauses on the

application, the frequency and duration of these pauses vary greatly with the choice of garbage collector, garbage collector settings, and application behavior.

As we shall see, garbage collection algorithms exist that attempt to avoid the need to ever collect the entire heap in a stop-the-world pause. The reason this is an important property is that if at any point (even if infrequently), you stop the application for a complete collection of the heap, the pause times suffered by the application scale proportionally to the heap size. This is typically the main thing you want to avoid when you care about latency. There are other concerns as well, but this is usually the big one.

Tracing vs. reference counting

You may have heard of reference counting being used (for example, cPython uses a reference counting scheme for most of its garbage collection work). I am not going to talk much about it because it is not relevant to JVM:s, except to say two things:

- One property that reference counting garbage collection has is that an object will be known to be unreachable immediately at the point where the last reference is removed.
- Reference counting will not detect as unreachable cyclic data structures, and has some other problems that cause it to not be the be-all end-all garbage collection choice.

The JVM instead uses what is known as a tracing garbage collector. It is called tracing because, at least at an abstract level, the process of identifying garbage involves taking the root set (things like your local variables on your stack or global variables) and tracing a path from those objects to all objects that are directly or indirectly reachable from said root set. Once all reachable (live) objects have been identified, the objects eligible for being freed by the garbage collector have been identified by a process of elimination.

Basic stop-the-world, mark, sweep, resume

A very simple tracing garbage collector works using the following process:

1. Pause the application completely.
2. Mark all objects that are reachable (from the root set, see above) by tracing the object graph (i.e., following references recursively).
3. Free all objects that were not reachable.
4. Resume the application.

In a single-threaded world, this is pretty easy to imagine: the call that is responsible for allocating a new object will either return the new object immediately, or, if the heap is full, initiate the above process to free up space, followed by completing the allocation and returning the object.

None of the JVM garbage collectors work like this. However, it is good to understand this basic form of a garbage collector, as the available garbage collectors are essentially optimizations of the above process.

The two main reasons why the JVM does not implement garbage collection like this are:

- Every single garbage collection pause will be long enough to collect the entire heap; in other words, it has very poor latency.
- For almost all real-world applications, it is by far not the most efficient way to perform garbage collection (it has a high CPU overhead).

Compacting vs. non-compacting garbage collection

An important distinction between garbage collectors is whether or not they are compacting. Compacting refers to moving objects around (in memory) so as to collect them in one dense region of memory, instead of being spread out sparsely over a larger region.

Real-world analogy: consider a room full of things on the floor in random places. Taking all these things and stuffing them tightly in a corner is essentially compacting them, freeing up floor space. Another way to remember what compaction is, is to envision one of those machines that take something like a car and compact it together into a block of metal, thus taking less space than the original car by eliminating all the space occupied by air (but as someone has pointed out, while the car is destroyed, objects on the heap are not!).

By contrast a non-compacting collector never moves objects around. Once an object has been allocated in a particular location in memory, it remains there forever or until it is freed.

There are some interesting properties of both:

- The cost of performing a compacting collection is a function of the amount of live data on the heap. If only 1% of data is live, only 1% of data needs to be compacted (copied in memory).
- By contrast, in a non-compacting collector objects that are no longer reachable still imply book keeping overhead as their memory locations must be kept track of as being freed, to be used in future allocations.
- In a compacting collector, allocation is usually done via a bump-the-pointer approach. You have some region of space, and maintain your current allocation pointer. If you allocate an object of *n* bytes, you simply bump that pointer by *n* (I am eliding complications like multi-threading and optimizations that implies).
- In a non-compacting collector, allocation involves finding where to allocate using some mechanism that is dependent on the exact mechanism used to track the availability of free memory. In order to satisfy an allocation of n bytes, a contiguous region

of n bytes free space must be found. If one cannot be found (because the heap is fragmented, meaning it consists of a mixed bag of free and allocated space), the allocation will fail.

Real-world analogy: consider your room again. Suppose you are a compacting collector. You can move things around on the floor freely at your leisure. When you need to make room for that big sofa in the middle of the floor, you move other things around to free up an appropriately sized chunk of space for the sofa. On the other hand, if you are a non-compacting collector, everything on the floor is nailed to it, and cannot be moved. A large sofa might not fit, despite the fact that you have plenty of floor space available — there is just no single space large enough to fit the sofa.

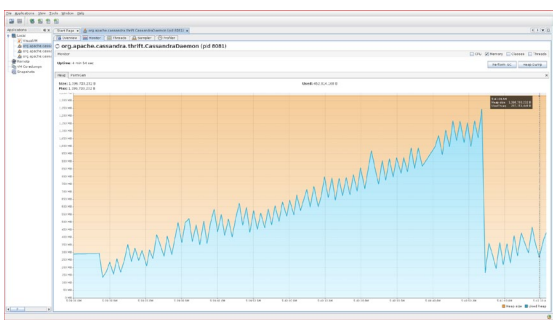
Generational garbage collection

Most real-world applications tend to perform a lot of allocation of short-lived objects (in other words, objects that are allocated, used for a brief period, and then no longer referenced). A generational garbage collector attempts to exploit this observation in order to be more CPU efficient (in other words, have higher throughput). (More formally, the hypothesis that most applications have this behavior is known as the weak generational hypothesis.)

It is called "generational" because objects are divided up into generations. The details will vary between collectors, but a reasonable approximation at this point is to say that objects are divided into two generations:

- The young generation is where objects are initially allocated. In other words, all objects start off being in the young generation.
- The old generation is where objects "graduate" to when they have spent some time in the young generation.

The reason why generational collectors are typically more efficient, is that they collect the young generation separately from the old generation. Typical behavior of an application in steady state doing allocation, is frequent short pauses as the young generation is being collected, punctuated by infrequent but longer pauses as the old generation fills up and triggers a full collection of the entire heap (old and new). If you look at a heap usage graph of a typical application, it will look similar to this:



Typical saw tooth behavior of heap usage with the throughput collector

The ongoing saw tooth look is a result of young generation garbage collections. The large dip towards the end is when the old generation became full and the JVM did a complete collection of the entire heap. The amount of heap usage at the end of that dip is a reasonable approximation of the actual live set at that point in time. (Note: This is a graph from running a stress test against a Cassandra instance configured to use the default JVM throughput collector; it does not reflect out-of-the-box behavior of Cassandra.)

Note that simply picking the "current heap usage" at an arbitrary point in time on that graph will not give you an idea of the memory usage of the application. I cannot stress that point enough. What is typically considered the memory "usage" is the live set, not the heap usage at any particular time. The heap usage is much more a function of the implementation details of the garbage collector; the only effect on heap usage from the memory usage of the application is that it provides a lower bound on the heap usage.

Now, back to why generational collectors are typically more efficient.

Suppose our hypothetical application is such that 90% of all objects die young; in other words, they never survive long enough to be promoted to the old generation. Further, suppose that our collection of the young generation is compacting (see previous sections) in nature. The cost of collecting the young generation is now roughly that of tracing and copying 10% of the objects it contains. The cost associated with the remaining 90% was quite small. Collection of the young generation happens when it becomes full, and is a stop-the-world pause.

The 10% of objects that survived may be promoted to the old generation immediately, or they may survive for another round or two in young generation (depending on various factors). The important overall behavior to understand, however, is that objects start off in the young generation, and are promoted to the old generation as a result of surviving in the young generation.

(Astute readers may have noticed that collecting the young generation completely separately is not possible; what if an object in the old generation has a reference to an object in the new generation? This is indeed something a garbage collector must deal with.)

The optimization is quite dependent on the size of the young generation. If the size is too large, it may be so large that the pause times associated with collecting it is a noticeable problem. If the size is too small, it may be that even objects that die young do not die quite quickly enough to still be in the young generation when they die. Recall that the young generation is collected when it becomes full; this means that the smaller it is, the more often it will be collected. Further recall that when objects survive the young generation, they get promoted to the old generation. If most objects, despite dying young, never have a chance to die in the young generation because it is too small, then they will get promoted to the old generation and the optimization that the generational garbage collector is trying to make will fail. Instead you will take the full cost of collecting the object later on in the old generation (plus the up-front cost of having copied it from the young generation).

Parallel collection

The point of having a generational collector is to optimize for throughput; in other words, the total amount of work the application gets to do in a particular amount of time. As a side-effect, most of the pauses incurred due to garbage collection also become shorter. However, no attempt is made to eliminate the periodic full collections which will imply a pause time of whatever is necessary to complete a full collection.

The throughput collector does do one thing which is worth mentioning in order to mitigate this: It is parallel, meaning it uses multiple CPU cores simultaneously to speed up garbage collection. This does lead to shorter pause times, but there is a limit to how far you can go. Even in an unrealistic perfect situation of a linear speed-up (meaning, double CPU count -> half collection time) you are limited by the number of CPU cores on your system. If you are collecting a 30 GB heap, that is going to take some significant time even if you do so with 16 parallel threads.

In garbage collection parlance, the word parallel is used to refer to a collector that does work on multiple CPU cores at the same time.

Incremental collection

Incremental in a garbage collection context refers to dividing up the work that needs to be done in smaller chunks, often with the aim of pausing the applications for multiple brief periods instead of a single long pause. The behavior of the generational collector described above is partially incremental in the sense that the young generation collectors constitute incremental work. However, as a whole, the collection process is not incremental because of the full heap collections incurred when the old generation becomes full.

Other forms of incremental collections are possible. For example, a collector can do a tiny bit of garbage collection work for every allocation performed by the application. The concept is not tied to a particular implementation strategy.

Concurrent collection

Concurrent in a garbage collection context refers to performing garbage collection work concurrently with the application (mutator). For example, on an 8 core system, a garbage collector might keep 2 background threads that do garbage collection work while the application is running. This allows significant amounts of work to be done without incurring an application pause, usually at some cost of throughput and implementation complexity (for the garbage collection implementer).

Available Hotspot garbage collectors

The default choice of garbage collector in Hotspot is the throughput collector, which is a generational, parallel, compacting collector. It is entirely optimized for throughput, the total amount of work achieved by the application in a given time period.

The traditional alternative for situations where latency/pause times are a concern, is the CMS collector. CMS stands for "Concurrent Mark & Sweep" and refers to the mechanism used by the collector. The purpose of the collector is to minimize or even eliminate long stop-the-world pauses, limiting garbage collection work to shorter stop-the-world (often parallel) pauses, in combination with longer work performed concurrently with the application. An important property of the CMS collector is that it is not compacting, and thus suffers from fragmentation concerns.

As of later versions of JDK 1.6 and JDK 1.7, there is a new garbage collector available which is called G1 (which stands for "Garbage First"). Its aim, like the CMS collector, is to try to mitigate or eliminate the need for long stop-the-world pauses and it does most of its work in parallel in short stop-the-world incremental pauses, with some work also being done concurrently with the application. Contrary to CMS, G1 is a compacting collector and does not suffer from fragmentation concerns — but it has other trade-offs instead.

Observing garbage collector behavior

I encourage readers to experiment with the behavior of the garbage collector. Use jconsole (comes with the JDK) or VisualVM (which produced the graph earlier on in this article) to visualize behavior on a running JVM. But, in particular, start getting familiar with garbage collection log output by running your JVM with:

```
-XX:+PrintGC
-XX:+PrintGCDetails
-XX:+PrintGCDateStamps
-XX:+PrintGCApplicationStoppedTime
-XX:+PrintPromotionFailure
```

Also useful but verbose:

```
-XX:+PrintHeapAtGC
-XX:+PrintTenuringDistribution
-XX:PrintFLSStatistics=1
```

The output is pretty easy to read for the throughput collector. For CMS and G1, the output is more opaque to analysis without an introduction. I hope to cover this in a later update.

In the mean time, the take-away is that those options above are probably the first things you want to use whenever you suspect that you have a GC related problem. It is almost always the first thing I tell people when they start to hypothesize GC issues: have you looked at GC logs? If you have not, you are probably wasting your time speculating about GC. ■

Peter Schuller is a Software Engineer in the Core Storage team at Twitter; before that, he was a developer at Spotify.

Reprinted with permission of the original author.
First appeared in *hn.my/gc* (worldmodscode.wordpress.com)

Understanding the bin, sbin, usr/bin, usr/sbin Split

By ROB LANDLEY

You know how Ken Thompson and Dennis Ritchie created Unix on a PDP-7 in 1969? Well, around 1971 they upgraded to a PDP-11 with a pair of hard drives.

When their root filesystem grew too big to fit on their tiny (half a megabyte) system disk, they let it leak into the larger but slower RK05 disk pack, which is where all the user and home directories lived and why the mount was called /usr. They replicated all the OS directories under the second disk (/bin, /sbin, /lib, /tmp...) and wrote files to those new directories because their original disk was out of space. When they got a second RK05 disk pack, they mounted it on /home and relocated all the user directories to this third disk so their OS could consume all the space on the first two disks and grow to *three whole megabytes*.

Of course they made rules about "when the system first boots, it has to come up enough to be able to mount the second disk on /usr, so don't put things like the mount command in /usr/bin or we'll have a chicken and egg problem bringing the system up." The fact their tiny system disk was much faster than an RK05 disk pack worked in there too: moving files from /bin to /usr/bin had a significant performance impact on this particular PDP-11. Fairly straightforward, and also fairly specific to the hardware v6 Unix was develped on 40 years ago.

The /bin vs. /usr/bin split (and all the others) is an artifact of this, a 1970s implementation detail that got carried forward for decades by bureaucrats who never question why they're doing things. It stopped making any sense before Linux was ever invented for multiple reasons:

1. Early system bring-up is the provice of initrd and initramfs, which deal with the "this file is needed before that file" issues. We already have a temporary system that boots the main system.
2. Shared libraries (introduced by the Berkeley guys) prevent you from independently upgrading the /lib and /usr/bin parts. Two partitions have to match or they won't work. This wasn't the case in 1974; back then they had a certain level of independence because everything was statically linked.
3. Cheap retail hard drives passed the 100 megabyte mark around 1990, and partition resizing software showed up somewhere around that time (partition magic 3.0 shipped in 1997).

Of course once the split existed, some people made other rules to justify it. Root was for the OS stuff you got from upstream and /usr was for your site-local files. Then / was for the stuff you got from AT&T and /usr was for the stuff that your distro, like IBM AIX or Dec Ultrix or SGI Irix, added to it, and /usr/local was for your specific installation files. Later, somebody decided /usr/local wasn't a good place to install new packages, so let's add /opt! I'm still waiting for /opt/local to show up...

Of course, given 30 years to fester, this split made some interesting distro-specific rules show up and go away again, such as "/tmp is cleared between reboots, but /usr/tmp isn't." On Ubuntu, /usr/tmp doesn't exist, and on Gentoo, /usr/tmp is a symlink to /var/tmp, which now has the "not cleared between reboots" rule. Yes, all this predated tmpfs. It has to do with read-only root file systems. /usr is always going to be read-only in that case, and /var is where your writable space is. Moreover, / is mostly read-only except for bits of /etc, which they tried to move to /var, but symlinking /etc to /var/etc happens more often than not.

Standards bureaucracies, like the Linux Foundation (which consumed the Free Standards Group in its ever-growing accretion disk years ago), happily document and add to this sort of complexity without ever trying to understand why it was there in the first place. "Ken and Dennis leaked their OS into the equivalent of home because the root disk on the PDP-11 was too small" goes whoosh over their heads. ■

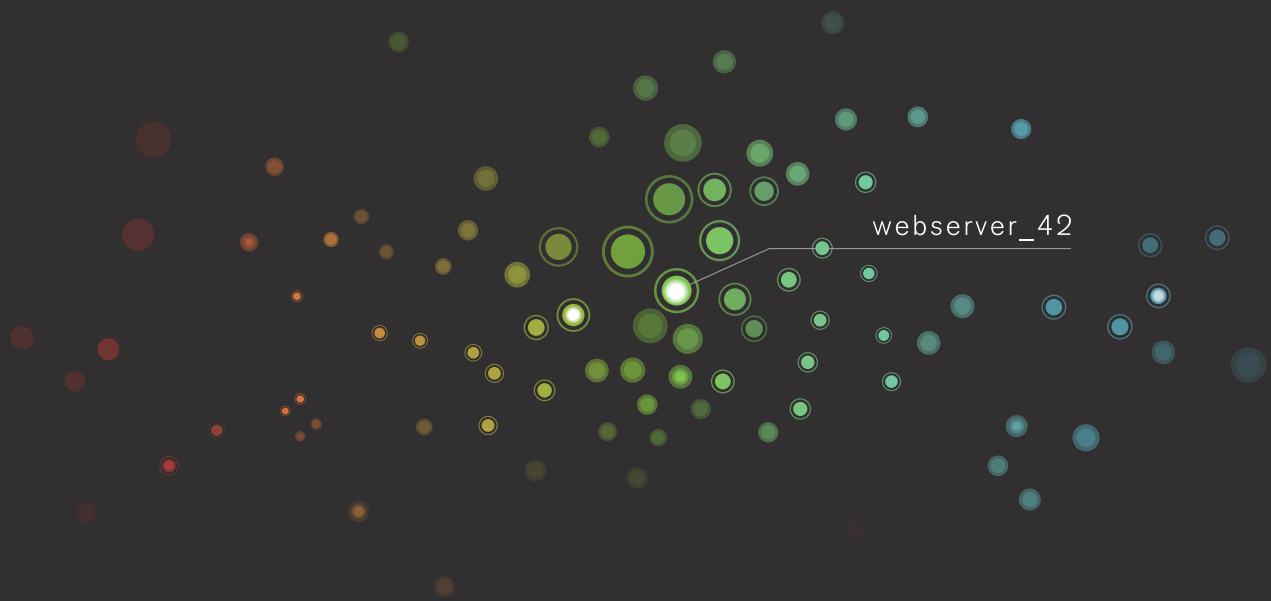
Rob Landley has been a geek since childhood, a Linux geek since 1998, and an embedded Linux geek since 2001.

Reprinted with permission of the original author.
First appeared in *hn.my/bin* (busybox.net)

References:
- *http://cm.bell-labs.com/cm/cs/who/dmr/notes.html*
- *http://cm.bell-labs.com/cm/cs/who/dmr/hist.html*

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

The Secret Number

By IGOR TEPER

Dr. Simon Tomlin studied the man sitting across the table from him. Rocking back and forth in his chair, with his shoulders slouching, his eyes darting all around the room, and his upper lip twitching every few seconds, the man conveyed a distinctly squirrel-like impression. It was hard to believe that, before his breakdown, this man had been one of the foremost number theorists in the world.

"How are you today, Professor Ersheim?" asked Dr. Tomlin.

"Fine, fine, thank you, just fine," replied the man without looking at him.

"Have you been sleeping all right?"

"Oh, yes, I've been sleeping quite well, sleeping like a baby," replied Ersheim, nodding vigorously in sync with his rocking. Still no eye contact.

"That's good to hear."

Ersheim suddenly stopped rocking and looked straight at Tomlin, eyes bulging. "Oh, cut the nice-guy act, Doctor," he said sharply. "I know you think I'm crazy, don't you think I know you think I'm crazy? That's what everyone thought about Laszlo Bleem, too; that's what they want you to think." He stared at Tomlin, not moving, not blinking.

"Who are you talking about, Professor? Who wants everyone to think you're crazy?"

"The numbers, Doctor, the numbers. They say that numbers don't lie, only they do, they lie all the time, they've always lied. But not to me — oh, no, I see through their deceptions, I know what they're hiding," said Ersheim. He started rocking again.

"And what would that be, Professor?"

"Bleem, that's what. Bleem!" shouted Ersheim, banging his fists against the desk. He then leaned close to Tomlin and whispered, "The secret integer between three and four."

"We have been over this, Professor — there is no integer between three and four."

"Tell that to Laszlo Bleem, Doctor," said Ersheim. "Only you can't — he's dead," he added, giggling. Then he whispered, "He died for trying to expose bleem."

"Laszlo Bleem died in a car accident, Professor."

"Oh, grow up! The man published a paper detailing his discovery of an up-until-now unknown integer somewhere between one and twenty, stating that he was working on a proof of its existence and exact location, and a week after the paper is published — poof! Bleem dies in a car crash, and his house burns down, destroying all of his written notes. The next day the computer system at his university crashes, erasing all of his electronic notes. Bleem got too close, see, and he was eliminated. Just as I'm going to be, if you don't listen to me."

At this point, Tomlin decided that it was time to play his trump card.

"All right Professor, let's say that there is, as you say, a secret integer between three and four. Positive integers are counting numbers, right?"

"That's right, Doctor," nodded Ersheim, and then, as if to confirm that fact, he began counting, moving his head from side to side: "one, two, three, bleem, four…"

"That's enough, Professor," interrupted Tomlin. "Now, if bleem is a counting number, that means that you can have bleem of something."

"Of course," said Ersheim. "I didn't know you were a mathematician, Doctor." He looked at Tomlin with what was probably meant to be a smile, but looked more like a scowl.

"Just bear with me, Professor," said Tomlin as he reached into his pocket and drew out a little plastic bag.

"What's that, Doctor?" asked Ersheim.

"Jelly beans," said Tomlin, smiling, as he tore open the packet and emptied its contents, about two dozen multicolored jelly beans, onto the desk.

"Now Professor Ersheim, I'd like you to please separate bleem of these jelly beans from the rest," said Tomlin, a self-satisfied grin on his face.

"All right," said Ersheim, and reached over and moved three jelly beans over to his side of the desk. He looked at them with suspicion, then looked back at the main pile, then back at the three lying before him, and quickly grabbed

another one and put it next to them. He studied the four jelly beans for a moment, then slid the fourth one back toward Tomlin, but when it was about halfway to the main pile, he snatched it back and added it to the three, visibly agitated. He then picked up each of the four jelly beans and held it up to his eyes, turning it this way and that, looking at it with deep mistrust. When he had inspected all of the jelly beans, he sat back in his chair, a look of frustrated resignation on his face.

"I can't do it, Doctor," he said.

"So bleem is not an integer after all," said Tomlin triumphantly.

"No!" screamed Ersheim and swept his hand over the desktop, sending the jelly beans flying all over the room. "Bleem exists! Something prevented me from separating bleem jelly beans! I could have three or four, but not bleem!"

"Calm down, Professor. I was here, I watched what you were doing, and there was nothing restraining you, nothing preventing you from separating out bleem jelly beans except for the fact that bleem doesn't exist."

"But it does exist," said Ersheim timidly. He added, with growing conviction, "It does exist. And I can prove it!"

"How can you prove it, Professor, if you insist that there is an omnipresent, invisible force keeping it secret?"

"Remember, Doctor," said Ersheim, his tone conspiratorial, "that I'm a mathematician, and a damn good one. All of mathematics has been doctored in order to conceal bleem's existence, see, but it wasn't doctored perfectly, oh no. There is an obscure branch of number theory that I helped invent about twenty years ago, and I think I can apply some of its theorems to prove that, in order for mathematics to be consistent, there must be an integer between three and four. That was the topic of my lecture during which I was so rudely interrupted by several of my colleagues and lost my temper."

Lost your temper indeed, thought Tomlin. It had taken two weeks to repair all the damage to the lecture hall.

"Those colleagues didn't seem impressed by your proof, Professor," said Tomlin.

"That's because I haven't worked out all the particulars of the proof yet," said Ersheim. "And even if I had, none of those idiots knows the first thing about my research," he added angrily. "But I'm close, Doctor, I can feel it. Just let me out of here, let me return to my research, and I'll have the proof in just a few months. Or at least allow me access to a pen and some paper so that I can work in here."

Ersheim was clearly agitated, so Tomlin decided not to aggravate him further.

"All right, Professor," said Tomlin, "I'll think about what you've told me. I just have one more question for you."

"What's that, Doctor?"

"What possible reason could anyone have to keep secret the existence of a number?"

"I'm not sure," said Ersheim, shaking his head. "Perhaps bleem has some mystical properties — don't give me that look, Doctor — or is believed to have them. Numerology has always had a fanatical following." After a moment's pause, Ersheim's face lit up with excitement. "Or perhaps the knowledge of bleem would allow us to attain a much higher level of mathematical sophistication. It might allow us to come up with a mathematically viable theory of time travel, or faster-than-light communication, or who knows what else."

"I see," said Tomlin, "and you really think the discovery of bleem might make these things possible?"

"I don't know, but who's to say it won't?" said Ersheim with a shrug.

"I see your point," said Tomlin. "Well, Professor, I'm very glad we had this talk. You've given me a lot to think about. I'll see you in a couple of days."

They shook hands, and Ersheim left the room. Tomlin sat there for a while, looking at the jelly beans strewn about on the floor.

How sad, thought Tomlin, that a man who has devoted his entire life to the study of numbers should come to think that those very numbers are out to get him. It made sense, of course, that the paranoia manifested itself in relation to something that Ersheim was already obsessed with.

Tomlin was not entirely pleased with that afternoon's session. He had hoped that the jelly bean example would force Ersheim to see the absurdity of his position, but all it did was aggravate him. Still, such a strong reaction indicated that perhaps Tomlin had hit upon a sensitive spot in Ersheim's delusion.

Satisfied that some progress had been made, Tomlin packed up his things and went home. Before leaving the hospital, he instructed the attendants who watched Ersheim that their patient should under no circumstances be allowed access to writing materials.

Tomlin had trouble getting to sleep that night. Every time he closed his eyes, he was confronted by visions of an army of giant numerals closing in on him, guided by a shadowy shape that was bleem. Frustrated, he pulled out a notepad he kept by his bedside, and wrote down the numbers between one and ten. They look so harmless, he thought, just squiggles on a sheet of paper, and yet numbers lie at the foundation of science, and thus make modern civilization possible. He looked at them again, with more respect, and mentally read them off, one by one. One, two, three, four, five, six, seven, eight, nine, ten. They were all there; there was neither need nor room for bleem. His mind finally at ease, Tomlin went to sleep.

He was awakened next morning by the ringing of his telephone. It was Gene, one of the attendants from the hospital. Ersheim was gone.

Tomlin rushed to the hospital. Upon arrival, he was greeted by Gene, who

explained to him what had happened, denying responsibility at every opportunity. Ersheim had been fine at ten the previous evening, when Gene last checked on him, but when Gene made his morning rounds at six, Ersheim was not in his room. Ersheim's door was locked from the outside, and the night watchman reported nothing out of the ordinary. As far as anyone could tell, Ersheim had vanished into thin air.

"I think you should see his room," added Gene when he was finished.

Tomlin followed Gene to Ersheim's room. When he saw it, his worst fears were confirmed.

The walls of the room were covered with equations. Rows upon rows of mathematical symbols, most of which Tomlin did not recognize, written by an unsteady hand in reddish purple ink. Ersheim had to have worked nonstop all night by the light of the moon.

Looking around the room, Tomlin noticed in one of the corners a little pool of what must have served as Ersheim's ink. He walked over to it, and found a plastic cup that had been knocked over. Dipping his finger in the ink, he tasted it. Grape juice. Floating in the puddle of juice was a crude writing implement fashioned out of a drinking straw. Piled up in another corner of the room were all of Ersheim's clothes. There was no sign of Ersheim himself.

"Looks like he left us a little snack," said Gene from behind Tomlin.

Tomlin turned around to see Gene standing next to the night table. Gene was reaching for one of three small dark objects lying on the table.

"Don't touch those!" yelled Tomlin.

"They're just jelly beans, Doc," replied Gene, as he flicked one of them into the air.

Tomlin watched in horror as the jelly bean described a parabola in the air, ending up in Gene's mouth.

"Want one?" asked Gene, motioning at the remaining jelly beans.

Tomlin looked down at the night table. There were three jelly beans on the tabletop. ■

Igor Teper lives with his wife and son in the San Francisco Bay Area and teaches old atoms new tricks at temperatures near absolute zero. He also writes stories, occasionally. "The Secret Number" was recently made into a short film of the same name.

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

Autodidacticism

By REGINALD BRAITHWAITE

There is a saying: "The man who is his own lawyer has a fool for a client." I wonder: "Does the man who is his own teacher have a fool for a student?" Now obviously, this is not always true. I can't speak to law, medicine, or architecture, but many people have done extraordinarily well as software developers after teaching themselves to program. In my own case, when I first saw a computer in school I had already been writing programs of one recreational sort or another for years.

While Computer Science is an excellent program in academia, the actual nitty-gritty of software engineering isn't so well developed. Many software development programs are actually preparation for a life as a clerk or disguised tests for conformity.

But no matter what you think about formal education, it has one thing going for it: the separation of teacher and student. Ideally, while the teacher has an interest in the student's success, the teacher does not rely on the student's influence. The teacher can fail the student. The teacher can force the student to learn things that are not fun or interesting. A student who just wants to learn enough to get a job can be forced to learn things that "won't be asked in the interview." A student who loves the recreational aspects of computer science can be dragged away from optimizing his personal HashLife project and told to get cracking on understanding principles of large-scale software architecture.

This arms-length relationship is important. It is why the man who is his own lawyer has a fool for a client: a good client seeks out a lawyer who can provide an objective perspective. You cannot be objective about your own choices. The same is true in real estate: my mother, who was the top salesperson in her days as a broker, always engaged another realtor to represent her when buying and selling her own property. She valued having an objective viewpoint.

Being your own teacher means forgoing this objective perspective. It often means being unaware of what you are missing. When people claim they are good at teaching themselves to program, I often think what they really mean is that they are extraordinarily good at learning to program. But there is more to being a good student than being good at learning. One of the responsibilities of a good student is to seek out excellent teachers. In the Wikipedia article on Autodidacticism, I find this paragraph:

> *Autodidactism is only one facet of learning, and is usually complemented by learning in formal and informal spaces: from classrooms to other social settings. Many autodidacts seek instruction and guidance from experts, friends, teachers, parents, siblings, and community.*

I think this is the correct approach. Instead of thinking of yourself an excellent — and therefore sole or primary — teacher, think of yourself as an excellent student with a voracious appetite for knowledge from many sources, carefully chosen to provide a balance between fun and drudgery, between inspiration and perspiration, between passionate support and dispassionate feedback.

Returning to the proposition, I will not say that the man who is his own teacher has a fool for a student. Instead, I will suggest that the man who does not limit himself to any one teacher — himself included — is a very wise student. ■

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

Reprinted with permission of the original author.
First appeared in *hn.my/autodidact* (raganwald.posterous.com)

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.

Please contact promo@magcloud.com with any questions.

MAGCLOUD