The Perfect IT Workplace

Shlomi Fish `<shlomif@shlomifish.org>`

Revision History

Revision 1841	30 April 2008	shlomif

Forked the template from a previous work and began working on it.

Revision 5762	02 July 2008	shlomif

The first revision as publicised on Reddit.com, by pkrumins.

Revision 5977	26 July 2008	shlomif

Forked the revision 1 version and started working on revision 2. Fixed many typos. Added the motivation for automated tests. Added more text to the design section.

Revision 6036	02 August 2008	shlomif

Started adding the essential tools section with more information, instead of just the knowledge in the Ether.

Revision 6048	04 August 2008	shlomif

Added the "Building, Configuration and Packaging" section.

Revision 6201	14 November 2008	shlomif

Added the Maturity, Mind-share and Distance from the Mainstream sub-section to "Choosing a Technology".

Revision 2319 (svn)	27 February 2009	shlomif

Added missing id attributes to footnotes, so they won't be randomly generated.

Table of Contents

Introduction

I originally wrote "The End of Info-Tech Slavery" [http://www.shlomifish.org/philosophy/computers/software-management/end-of-it-slavery/] a few years ago, shortly after I was fired from a job I liked and for which I worked about three months. Some people who commented on the article found it of good value. However many people who commented on it have voiced some criticism.

One of the comments I received said that I kept giving bad examples for conditions I disliked from previous workplaces, instead of positive elements I would have liked to see or have seen. As much as I like liquidating negatives, I agree that I should also mention some good things that a workplace needs to adopt.

Another commenter told me in private, that he felt that the "End of IT Slavery" and other essays of mine reflected the fact that I had a "primadonna attitude" instead of a more positive "team-player" attitude. While this may be deduced from my "The End of IT Slavery", I found that as a tech worker, I tended to just get my job done as I told, try to be as productive as I can be, try to adjust to the surroundings, and even be a little of what Israelis call "rosh qatan" (= "small-head") [http://www.joelonsoftware.com/items/2004/12/06.html]

I can see why people would think so after reading the "End of IT Slavery". However, my intention in the article was to guide employers (and employees) as to how to best treat their employees, and give them good conditions, so they'll be happier and more productive. Going over the original article, I can see that I indeed had a very ranting tone, and had given bad examples for what not to do, exclusively.

So this time, I'm going to try again and hopefully be more successful. This article aims to cover the elements that great programmers [http://www.paulgraham.com/gh.html] would love to see in a perfect

workplace. By implementing as many of them as possible, or at least giving them a serious thought, you can make such potential employees want to work for you, love to work for you, want to stay, and be happy as long as they are working for you. Furthermore, you will know better than to irrationally fire perfectly good employees.

I'm not trying to be original in this article, because an essayist (or any kind of artist) does not get credit for coming up with perfectly original concepts or ideas (as I note later in a different context). What I'm trying to do is summarise everything I've learned, judged and concluded, so far, based on the many sources I've read. I may be just a dwarf standing on the shoulders of giant, but hopefully I'll still be able to see farther than most.

There is a lot of material on good software management out there, and not everybody took the time to read all the important one. This material can sometimes be contradictory, and sometimes false. Nevertheless, I decided to integrate it here into what I've perceived and concluded to be the best choices.

Sources from Which the Advice was Taken

This is a non-exhaustive list of major sources of influence.

1. Eric S. Raymond's Homepage [http://www.catb.org/esr/] and especially his the *Cathedral and the Bazaar* series [http://www.catb.org/esr/writings/cathedral-bazaar/] and his "How to become a Hacker" [http://www.catb.org/~esr/faqs/hacker-howto.html] document.

2. The wonderful *Joel on Software* site [http://www.joelonsoftware.com/] by Joel Spolsky.

3. Paul Graham's essays [http://www.paulgraham.com/] - tend to be interesting, but often suffering from some problems.

4. Extreme Programming [http://www.extremeprogramming.org/] - seems a bit too rigorous, but still has many good ideas.

5. Damian Conway's excellent book Perl Best Practices [http://www.oreilly.com/catalog/perlbp/]

The Elements of a Perfect Workplace

Hire the Best Developers

Joel on Software [http://www.joelonsoftware.com/] and others have written about the importance of having very good developers. See for example his "Guerilla Guide to Interviewing" [http://www.joelonsoftware.com/articles/GuerrillaInterviewing3.html]. Bad programmers will create code that's buggy, insecure, hard-to-maintain, etc. The so-called "Medium-level techs" will probably not, but will be heavily under-productive in comparison to star programmers. (And according to Brooks' Law [http://en.wikipedia.org/wiki/Brooks'_law], you can't effectively replace one good programmer with many worse ones.)

As I noted there are many aspects that qualify someone as a good programmer [http://discuss.joelonsoftware.com/default.asp?joel.3.197867.7]. However, it does not change the fact that hiring a large number of mediocre programmers will never be as effective as hiring one or two good or very good programmers.

Openness

One of the most important trends in the software world recently was towards "openness": open-source, open-content, open-specifications, etc. Most good programmers you can find will find the open source

(also known as "Free Software") movement very appealing, and share some of its ideals. I don't propose you should make your software open-source, although, of course, this is often a good business model. However, there are other aspects of openness that you should follow if you want to attract and keep good programmers.

First of all, don't over protect your code. Make it open-source if possible, give your contractors and consultants (including outsource ones) access to it, and allow your programmers to show parts of it to their online friends to get some advice. You should make as much of it as possible public, under open source licences [http://en.wikipedia.org/wiki/Open_source_license] to allow it to be used more often, increase its quality, and grow a culture around it. From my experience, working on "shrinkwrap" software that is used by end-users in the wild, and open-source software in particular, is the best way to increase the quality of the software [http://www.shlomifish.org/philosophy/computers/high-quality-software/] as it requires the most discipline to work on and support.

Openness does not end at that. Another important aspect is to allow your programmers to tell their friends and family about what they do. Some defence-related companies seem to think keeping everything confidential is a good idea, but that will cause your employees to feel unnecessarily trapped and unable to get help. So if you want to attract the best programmers, make sure they can tell other people of what they do, and what problems they are facing. (I'm not advocating complete transparency, if it's not appropriate, but rather not being overly secretive.)

Finally, you should avoid vendor lock-in: use standard or documented protocols and specifications, take Joel Spolsky's advice of letting your users go back [http://www.joelonsoftware.com/articles/fog0000000052.html], and make sure your users have control of their data and can access it, back it up and convert it to different format.

A good example for this is the photo-sharing site Flickr [http://flickr.com/], which has published full APIs for its service, and even allowed these APIs to be used by competing site.

Great Working Conditions

You should make sure your employees have great working conditions.

Kitchen

There should be a kitchen with a lot of food, hot and cold beverages, etc.

Comfortable Offices

I don't necessarily agree that your workers should have spacious offices with doors that close, but they should still be comfortable enough. At one of my workplace, I constantly had to move to let other people out of their seats, and this was unbearable. So don't do that.

In one job that I fondly remember, every employee had their own spacious cube, with a desk to put a computer and a place to put a few chairs. This was much more like it.

The Best Tools That Money Can Buy

This cannot be stressed enough. As Joel Spolsky notes [http://www.joelonsoftware.com/articles/fog0000000043.html] (based on Steve McConnell) in item No. 9 of the Joel Test, you need to "use the best tools that money can buy".

If you buy old, broken and/or barely functioning hardware, you'll spend a lot of time debugging the problems there, which will waste a lot of precious time. And you may lose a lot of reputation and customers due to down-time. **Relying on reliable, high-end hardware** is a much better idea.

I've been to two workplaces that gave me an old computer with a 40 GB hard-disk. It wasn't enough at all. At one place, we've reached the limit of this hard-disk due to several large source code checkouts, and as a result needed a bigger hard-disk. And the only hard-disks the lab had were 80 GB ones, which were bought because they were the cheapest (per unit, not per capacity). Please, **buy large enough hard-disks**.

At the same workplace, I was given a computer with a read-only CD-ROM drive. It was not even a DVD reader I brought a DVD of audio files from home, and could not read it. In this day and age, read/write DVD drives are the standard, and are ultra-cheap.

Sometimes you'll need **several computers**, or a decent **virtual machine emulator**, to run alternative operating systems on the same machine.

Make sure your workers have a **high-quality screen**. They need to look at it most of the day, and they want it to look nice. A decent 19# LCD screen nowadays is very cheap, and well worth the added productivity.

You also need a state-of-the-art version control system [http://better-scm.shlomifish.org/] , of which there are currently several high-quality open-source alternatives. Some very costly version control systems have a bad reputation for being extreme troublemakers, while the modern open-source alternatives "just work".

If there's a good commercial software application that your employees like to use and can recommend, don't hesitate to buy it. You'll also find O'Reilly Safari [http://safari.oreilly.com/] Licences to be a good idea so your employees can easily look up and read information in many books on-line.

Sane Working Hours

Give your employees sane working hours as Evan Robinson writes in "Why Crunch Mode Doesn't Work: 6 Lessons" [http://www.igda.org/articles/erobinson_crunch.php]. I.e: 40 hours work-weeks or less. While sometimes asking them to stay late to finish a deadline is acceptable, the so-called "crunch mode" is under-effective and a recipe for disaster.

This feature was covered on Slashdot [http://developers.slashdot.org/story/05/06/08/1646234/why-crunch-mode-doesnt-work] where there are many useful comments and later on a similar article ("Why We Have to Go Back to a 40-Hour Work Week to Keep Our Sanity") [http://www.alternet.org/visions/154518/why_we_have_to_go_back_to_a_40-hour_work_week_to_keep_our_sanity/?page=entire] was published in the site *AlterNet*, and also featured on Slashdot (with many comments) [http://developers.slashdot.org/story/12/03/16/1252220/bring-back-the-40-hour-work-week?sdsrc=popbyskid].

This cannot be stressed enough. The more people work, the less their hourly throughput is going to be, until they end up at negative throughput. As a result, overworking workers is dangerous because you're risking introducing bugs and destroying existing work. Furthermore, your workers are going to be much less happy than if they worked less, which is also something you'd like to avoid.

Location, Location, Location

Find a good place which is close enough for most people to get to. Make sure your programmers can commute to you easily. Even pay for taxi-cabs out of your own expense, or ask other employees to give them a ride, if there isn't good public transportation. The money you spend on the cabs is very small compared to the one you would lose by lost productivity, and by frustrations of travelling.

A Lot of Paid Vacation

Give your workers a lot of paid vacation. Humans are not machines - they need rest and relaxation. The 14 annual vacation days mandated by the Israeli government are a joke. You should give them much more than that. Joel Spolsky's company - Fog Creek Software [http://www.fogcreek.com/] gives 6 weeks of paid vacation, which is much more reasonable.

Avoid Over-Strict Firewalls

Some organisations put their intranets behind firewalls that block access to almost all services. It is not uncommon to see only the HTTP and HTTPS ports open for free access.

However, there are many other Internet services that star programmers need in order to be productive. Among them are:

1. Internet Relay Chat [http://en.wikipedia.org/wiki/Internet_Relay_Chat] and other forms of Instant Messaging [http://en.wikipedia.org/wiki/Instant_messaging]. This allows star programmers to discuss problems and share solutions interactively with their peers, or just to take a break from work and chat.

2. SSH - Secure Shell [http://www.openssh.org/] - allows access to remote computers over the network.

3. BitTorrent [http://en.wikipedia.org/wiki/BitTorrent_(protocol)] - allows downloading some content that is otherwise not available on the Web, or that its download can be resumed faster.

4. rsync [http://samba.anu.edu.au/rsync/] - allows for faster incremental downloading and mirroring.

And naturally, malware can easily propagate and survive using HTTP and HTTPS alone, and there are ways that clueful workers can overcome such restrictions.

At one of my workplaces, I was able to chat on the IRC, connect using IM, ssh to everywhere I wanted, etc. without any restriction. It was a liberating feeling and I felt at home there. At a more recent one, I needed to connect to a remote host, and invoke port forwarding in order to talk on the IRC which was annoying and error prone.

So make sure your firewall is not over-zealous and does not prevent legitimate uses.

About the Salary

Competitive Salary? Yes, it's important, but not absolutely everything. Great programmers won't work for you for free, but they'll find many other conditions much more tempting than an over-blown salary. See what Eric Raymond wrote about it in "Homesteading the Noosphere" [http://www.catb.org/~esr/writings/cathedral-bazaar/homesteading/ar01s19.html], quoting many studies:

> Psychologist Theresa Amabile of Brandeis University, cautiously summarizing the results of a 1984 study of motivation and reward, observed "It may be that commissioned work will, in general, be less creative than work that is done out of pure interest.". Amabile goes on to observe that "The more complex the activity, the more it's hurt by extrinsic reward." Interestingly, the studies suggest that flat salaries don't demotivate, but piecework rates and bonuses do.

> Thus, it may be economically smart to give performance bonuses to people who flip burgers or dug ditches, but it's probably smarter to decouple salary from performance in a programming shop and let people choose their own projects (both trends that the open-source world takes to their logical conclusions). Indeed, these results suggest that the only time it is a good idea to reward performance in programming is when the programmer is so motivated that he or she would have worked without the reward!

Conditions: Conclusion

One final note: you might think to yourself: **"how can I afford all that?"**. Well, the answer is that the cost of all these advantages is very small in comparison to your general operation. And they will pay themselves much more as time goes by in the happiness, productivity and loyalty of your workers.

The last thing you want is to lose a competent developer. And all these things will better make sure that he or she will stay with you.

Pair Programming

When doing Pair programming [http://en.wikipedia.org/wiki/Pair_programming], there are two programmers sitting next to a single computer screen and keyboard, with one of them actually writing the code, and the other one observing and helping. At first glance, it seems like it's a waste of resources: won't they be under-productive? The answer is a "no".

First of all, pair programming increases run-time code review, as the "passive" programmer observes what the active programmer writes, gives him advice and answers his question. Code-review is always a good thing. CodeReview

Secondly, pair programming causes both programmers to have more discipline, and they are less likely to procrastinate as the active programmer will bore the watching one.

Pair programming is also more fun, because people feel more comfortable working in pairs than alone.

Finally, pair programming is more productive, because it was shown that it yields more output than two individual programmers.

I had had a very good experience working in pairs back when I was an undergraduate student at the Technion [http://www.technion.ac.il/] and can highly recommend it.

Give Your Employees the Freedom to Do What They Want

If your employee is responsible, he'll care about his job to be productive, even if you don't watch him. If he isn't, then no amount of watching will make him responsible.

You shouldn't constantly monitor your employees. Instead, give them time to clear their mind and do non-coding-related activities such as:

1. Play computer games.

2. Browse the web.

3. Watch videos or listen to netcasts.

4. Read articles or books.

5. Work on open-source software or their own personal projects.

6. Chat with their co-workers or on-line peers.

7. Take walks in the neighbourhood, do sports, go to clubs and other activities.

These tasks and others don't interfere with the work much, and actually contribute to productivity. And along with pair programming, you can be more certain your programmers will work. Don't judge your

---

CodeReview Pair programming is still not a substitute for code reviews by people who didn't directly write the code. It was noted at a blog entry I recall reading somewhere but can no longer find. **TODO: Write more.**

employees by how they spend their time at work - instead judge them by the overall progress they do in the long run.

Moreover, some tasks that your employees do at their free time, may eventually bring your company publicity and esteem. Part of the reason Joel Spolsky's Fog Creek Software company [http://www.fogcreek.com/] is so well-known is because of the Joel on Software [http://www.joelonsoftware.com/] blog, which is one of the most-read weblogs about software management. This in turn resulted in a lot of publicity to the products of Fog Creek.

If your employees are productive, you should not really be concerned what they are doing on their work time. [EricSink] Google allows its employees work on non-work-related-tasks 20% of the time (while Google still owns the intellectual rights to their creations). I would go a step further by saying you simply should allow your employees to do as much non-work-related leisure as they feel they need to on their working hours. Tell them you expect long-term results, not 100% productivity-of-time.

Phrase Your Job Ads in a Unique and Smart Way

"3 Years Experience in Perl - Must", "20 Years Experience in PHP - Advantage", "Team Player", "Independent Thinker". **Boring**.

Your job ad need to stand out. Take this job ad that was posted to the Israeli Ruby mailing list [http://groups.google.com/group/israelrb/browse_frm/thread/bb11cf1c3ed8e221] for an excellent example:

> We are looking for a developer who:
>
> - Want a full time, salaried position.
>
> - Want to work in a fun, young workplace.
>
> - Want to work in an environment that allows them to use (just about) whatever tools they wish to get the job done.
>
> - Knows and love Ruby.
>
> - Ideally have experience with PHP and Python.
>
> - Want to work on large scale Rails/Merb projects that have nothing to do with "the Social Web".
>
> - Want to drown a puppy every time they hear phrases like "the Social Web".

See? I also want to write sites that are not necessarily "Social Web", and I also got tired of the "Social Web" and its association with Ruby and other similar technologies. So this ad has caught my attention.

As Joel on Software notes about experience [http://www.joelonsoftware.com/articles/LordPalmerston.html], the reason companies want people with a lot of experience in a certain niche, is because they tend to overcome problems they encounter more easily and so can help themselves and their co-workers with their problem more easily. [LinuxSystemsProgramming]

However, as long as you have someone with enough experience, then bright people without a lot of experience can still prove to be useful and very productive. So don't demand too much from them. As of

---

[EricSink] Eric Sink wrote a wonderful entry about the tolerance of his former boss [http://www.ericsink.com/entries/scrabble_1994.html] on his weblog. (Here's an update [http://www.ericsink.com/entries/ethics.html].)

[LinuxSystemsProgramming] For example, I have a lot of experience working on Linux and developing for it, with Perl and doing mostly Algorithms and Text Processing in C. Recently, however I got a job as a developer for a Linux Server-side C/C++ application (dealing with sockets, processes, threads, Unicode and other such advanced problems). While I made a lot of progress, I noticed that I often got stuck on many problems, for which I had no idea how to easily resolve. In this case, my co-worker who had more experience than me in this area, could often help me more.

2008, there aren't enough clueful developers in Israel for all the workplaces (and I think that's also the case in most other locations around the world), and you can't afford to be too picky.

Another tip I can offer to look for employees is to use specialised job boards like the the Joel on Software job-board [http://jobs.joelonsoftware.com/], and jobs.perl.org [http://jobs.perl.org/], the DailyWTF "Non-WTF" board [http://jobs.thedailywtf.com/1001/browse.aspx] which highly-qualified and niche people follow. Just make sure to phrase your ad in a non-boring way, as I noted earlier.

Read a Lot of Software Management Material

It's impossible to put all the good advice I found regarding software management and software engineering on-line and off-line, in one document. So I urge you as a manager or an employee to read as much different material you can find. There's a lot of good advice out there. Socrates said: "I know that I do not know.", and he was right.

While it's easy to dismiss other people as "idiots" or what they say as being "stupid", one should realise that even incorrect conclusions or reasoning can bring useful insights, and give some food for thought. Sometimes, I find some useful insights in rehearsed things or short blog entries.

If you're a manager or team-leader, you should make sure that you're not much more clue-less than your programmers are, but that has often been the case for me and others.

Listen to Your Developers

Which brings us to this: you obviously cannot get every aspect of software management right at first. Your programmers may eventually become unhappy, and you should make sure they can tell you how they feel, and to take note of what they say. Otherwise, you're risking under-productivity or downright clinical depression [http://en.wikipedia.org/wiki/Clinical_depression].

Software Engineering Best Practices

Functional Specs

The first advice I can give is to **write functional specs** [http://www.joelonsoftware.com/articles/fog0000000036.html]. In the link, Joel Spolsky explains the motivation and method of writing functional spec. A functional spec describes how the end-user will use and interact with the program. It is not a technical spec which explains how the inner code will work.

From my experience, functional specs are fun to write, reveal many problems in the initial design, and make you think how the program will work on the inside.

Here are three examples for functional specs I wrote:

1. Functional Spec for a Freelancers Board [http://svn.berlios.de/svnroot/repos/web-cpan/App-Freelancers-Board-Yonathan/trunk/docs/functional-spec/App-Freelancer-Board-Yonathan-functional-spec.txt] (with a Biblical theme).

2. Functional Spec for a better classification of CPAN Modules [http://svn.berlios.de/svnroot/repos/web-cpan/CPAN-Module-Classification/trunk/docs/functional-spec-for-CPAN-Classification-Proposal.txt] (with a theme of FOSS world celebrities).

3. Functional Spec for a Windows package management system. [http://svn.berlios.de/svnroot/repos/winapt/winapt/trunk/docs/design/functional-spec.txt] (featuring characters from Ozy and Millie [http://www.ozyandmillie.org/]).

Automated Tests

You should write automated tests such as unit tests, integration tests, system tests, that will run automatically on the code and yield an answer if all of them are successful, or if any of them fail.

There are many good practices for automated testing such as having daily builds or reaching a 100% test coverage. Here are some resources to get you started:

- The Perl Quality Assurance project [http://qa.perl.org/], also see their wiki.

- The Wikipedia page about Software Testing. [http://en.wikipedia.org/wiki/Software_testing]

Why are automated tests important? For several important reasons:

- They provide a certain kind of executable and machine and human-understandable specification for the program, for new features and for bug reports. By writing a test such as `add(5,6) == 11` both humans and machines can tell that that's what the "add()" function is supposed to do.

- A test suite with a good test coverage can better ascertain that doing refactoring, adding new features, removing them, or fixing bugs, won't break anything.

- When the program breaks, then, without automated tests, one cannot easily determine what exactly is wrong there. On the other hand if you have a lot of tests, with good coverage, you can see which tests fail, and what exactly went wrong.

In short, writing automated tests gives more confidence in the code and in changing it.

Note that having automated tests is not a substitute for having dedicated software testers [http://www.joelonsoftware.com/articles/fog0000000067.html] (and vice versa). By all means, they are both necessary for any serious operation.

Design and Planning

Most people agree that designing a program, planning it and thinking about it is a good idea. Extreme Programming suggests to have one design meeting every week, which sounds reasonable.

Joel Spolsky gives the motivation for designing software in one of his articles [http://www.joelonsoftware.com/articles/NothingIsSimple.html], which is a recommended read. He also voices an opinion about over-formal design (with lots of UML diagrams, etc.) that can actually be a disadvantage to a project's success.

Refactoring

Refactoring [http://www.refactoring.com/] is the process of improving the internal quality of the code (from "a big mess" to "squeaky-clean and modular code"), while not changing its external behaviour. This is done for mostly functional and bug-free code, but one which is not as elegant as it could be, so it can be better managed.

"Joel on Software" features two excellent article about the motivation for refactoring: "Things You Should Never Do, Part I" [http://www.joelonsoftware.com/articles/fog0000000069.html] (why rewriting functional code from scratch is a bad idea), and "Rub a dub dub" [http://www.joelonsoftware.com/articles/fog0000000348.html] (how and why to do refactoring).

There are many other resources for that on-line, along with many refactoring patterns.

There are many types of refactoring: grand refactoring sessions (= what Joel describes), continuous refactoring (refactoring as you go), "just-in-time refactoring" (refactoring enough to achieve a certain task), etc.

But refactoring is important, makes development faster in the long run (and even in the short-run), and can prevent the code from deteriorating into an ugly, non-functional mess that would be hard to salvage.

Software Engineering Methods to Avoid

There are a few software engineering methods that I find pointless, so I'd like to briefly point them out.

The first is the **"Huge-design-up-front"**, where an "architect" or even the entire team spend a very long time writing an extremely comprehensive and detailed technical document that specifies in detail how the software will work.

The problem with this approach is that it is a huge waste of time and also it is impossible to design a large project top-down like that. A better way is to involve the entire team in a good design session (a week long at most) while writing some functional specs, diagrams and some other documents, and then to write it incrementally.

A similar fallacy is the **"Mountains of documentation fallacy"** of having superfluous commenting, "Literate programming" (see the Wikipedia article [http://en.wikipedia.org/wiki/Literate_programming]), etc. The problem with this approach is that the extra documentation is often redundant if the code is well written and factored out [ExtractMethod]

Some documentation (especially API documentation such as Perl's POD [http://en.wikipedia.org/wiki/Plain_Old_Documentation] or Doxygen [http://www.doxygen.org/]) is good, and you shouldn't feel too guilty about writing a comment for an interesting trick, but too much documentation slows things down, and doesn't help with the design process, and eventually may turn out to be misleading or harmful.

Something else I consider a bad idea is Overengineering or YAGNI ("You Ain't Gonna Need It") [http://en.wikipedia.org/wiki/You_Ain't_Gonna_Need_It]. The basic idea is not to implement too many features at once, or over-complicate the design of the code, and instead focus on getting the necessary functionality working.

YAGNI, put aside, I still believe that some forward-planning is good.

[ExtractMethod] A good example is that instead of having the following pseudo-code:

```
function create_employee(params)
{
    let emp = Employee.new();

    emp.set_birth_year(params[year]);

    emp.set_experience_years(params[exp_amount]);

    emp.set_education(params[education]);

    ### Calculate the salary:
    emp.set_salary(emp.calc_education_factor(emp.get_education()) * emp.get_experience_years());

    ### More stuff snipped.

    return emp;
}
```

Then it would be a good idea to extract the complex `emp.set_salary()` call into a simple `emp.calculate_salary()` method. This method extraction [http://c2.com/cgi/wiki?ExtractMethod] will make the intention of the code self-documenting (as the method will have a meaningful name) and much more robust for future changes than adding a comment.

And this is just a small example.

Maintaining Two Codebases

The code-with-accompanied-documentation is somewhat a sub-case of a more general fallacy: the belief that a company can simultaneously maintain two different code-bases (say two code-bases implementing the same functionality in two different languages), while keeping them both synchronised.

The Extreme Programming experts have been warning against it, and it also makes sense that it is hard to do given that programmers, by nature, lack the appropriate discipline to keep maintaining two or more code-bases at once.

One possible solution to this problem has been illustrated by Fog Creek software [http://www.joelonsoftware.com/articles/FogBugzIII.html]. What they did was implement a compiler from a common language, to several target languages. Naturally, this compiler also needs to be maintained and extended, but it is less work and less error-prone than maintaining several different code-bases.

Don't Over-Specialise

Many workplaces seem to think that it is a good idea to make sure every developer is limited to a certain aspect of the development process, and will specialise exclusively in it. Taken to extreme, however, such a trend is very dangerous.

Programmers need to have a solid understanding of the entire product stuck in their head. Good programmers can also benefit from a change of scenery. And naturally, reviewing someone else's code [http://en.wikipedia.org/wiki/Code_review], and criticising it, is also always beneficial.

Therefore, you should make sure that your workers don't over-specialise. Let them dedicate some time to work on what they find interesting, and let them review and criticise other people's work.

Once, before I started my undergraduate studies, I worked s a developer of a software-based modem (so-called "Winmodems"). I was impressed from the atmosphere there, and how professional and clueful the managers were. One of the things I liked about the job was that while I was initially hired as a tester, I eventually was appreciated for my eclectic knowledge of different aspects of sound, games, modems, Internet, and other fields. So, I was given more diverse things to do there, involving many aspects of our operation.

You should follow suit and involve your star developers in as many aspects of your development.

Listen to Your Customers or Users

Joel on Software gives a lot of good advice on having a good customer service [http://www.joelonsoftware.com/articles/customerservice.html]. You should listen to what your customers (or users) say. Don't dismiss them. Don't give them annoying canned responses. Open your bug-tracker to the world as a web-service. There are many web-based, and gratis bug-trackers, such as Bugzilla [http://www.bugzilla.org/], and you should use one of them for better interaction with your customers.

As Joel indicates, a bug report, email, or a query usually indicate a problem in the application: a bug, a missing feature, or something that's not clear enough. Such problems need to be fixed in the code.

Furthermore, if one or more of your customers are requesting a feature, and it seems to be important enough, give a priority to implementing it. As a developer from Mercury Interactive (Now part of HP) noted [http://tech.groups.yahoo.com/group/hackers-il/message/2751] usually the features requested by the customers, are the ones that will yield the largest amount of newer customers, not-to-mention will allow keeping customers, if they are paying for upgrades or as a "software-as-a-service" model.

Joel Spolsky also notes that good customer service is part of the key to a small ISV's success [http://www.inc.com/magazine/20080401/how-hard-could-it-be-fire-and-motion.html?partner=fogcreek].

"Good Poets Borrow. Great Poets Steal."

Originality and "innovation" is not as important as people think it is. It is perfectly OK to create a product in a niche which has a lot of existing competition [http://www.joelonsoftware.com/articles/fog0000000056.html], it's OK to build on other people's efforts, and it's OK to "borrow" or "steal" (see the quote at the top) ideas from your users, competitors and friends.

You're not getting credit for originality - you're getting credit for high-quality products (software, services, support, etc. - whatever you do) and especially for revenue and profits. See Joel's "Converting Capital into Software That Works" for more information [http://www.joelonsoftware.com/articles/fog0000000074.html]

Another common fallacy is that ideas are the hardest part of the production process. However, ideas are very cheap, and creative people have far too many good ideas. It's actually harder to develop the idea into a product, to mass-produce it, and then to mass-distribute it. Edison was not the first to come up with the Electrical Lightbulb, or to develop a working prototype, but he was the one to have put up all the effort in making it so popular and prevalent. Therefore, he is rightly credited as its inventor.

Similarly, in the software world, you can often see competing products displacing more established competition, or sometimes a market where there isn't a clear winner (e.g: window managers and desktop environments for Unix [http://xwinman.org/], Bug trackers, text editors, etc.)

Even making rounder wheels is a good way to earn a living and to get respect.[IdeasSameTime]

Study Psychology

Your co-workers are people. As such they have thoughts, feelings, emotions, and desires, which guide them and affect them. Learning how people's psychology works, and how to motivate people and help them if they are feeling de-motivated - will help an employer be more effective, and make his employees more productive.

This is also the case for your customers, users, associates and friends, who you will surely have to interact with , make sure they are happy most of the time, and deal with their problems effectively.

Here are a few good resources on Psychology:

• Feeling Good: The New Mood Therapy [http://www.amazon.com/exec/obidos/ASIN/0380810336/ref=nosim/shlomifishhom-20/] - the best book I've read about Psychology. A self-help book of Cognitive-behavioural therapy [http://en.wikipedia.org/wiki/Cognitive_behavioral_therapy], this books explains what causes Clinical depressions, and other negative mood swings, and why people behave the way they do.

It is a stark anti-thesis to Freudian psychology, which makes little sense and is completely not helpful.

• The Neo-Tech Discovery [http://www.shlomifish.org/philosophy/philosophy/guide-to-neo-tech/] - an off-shoot of Ayn Rand's Objectivism, Neo-Tech is the best idea-system I've encountered yet. Note that it is easy to both dismiss Neo-Tech as a stupid cult, or to hugely misunderstand it at first. So when reading Neo-Tech go over the material (preferably without skipping, but possibly while taking breaks), and then let it sink for a while.

[IdeasSameTime] Barry Jones said [http://www.shlomifish.org/humour/fortunes/shlomif-fav.html#same-ideas-as-everybody-else]

> If you have the same ideas as everybody else, but have them one week earlier than everyone else - then you will be hailed as a visionary. But if you have them five years earlier, you will be named a lunatic.

This illustrates that as technology progresses, people tend to get the same ideas at roughly the same time, when they are mature and the natural logical step forward. It's unlikely you can avoid it.

Note that as of May 2008, the old hyperlinks to the Neo-Tech site lead to redirects or PHP errors. You can still find the old pages on the web archive [http://www.archive.org/], and I hope they'll get fixed. In any case, it may be a good idea to order "The Neo-Tech Discovery" book. (It is not available in book stores).

- The Design of Everyday Things [http://www.amazon.com/exec/obidos/ASIN/0465067107/ref=nosim/shlomifishhom-20/] - this book is a must read to everyone doing user-interface design. It explains how frustrating it is to use many everyday objects (and, by inflection, software and software devices), and how to design them right.

- Helplessness: On Depression, Development and Death [http://www.amazon.com/exec/obidos/ASIN/071672328X/ref=nosim/shlomifishhom-20/] - I haven't read this book yet, but it was recommended on the *Joel on Software* book reviews [http://www.joelonsoftware.com/navLinks/fog0000000262.html], which I found to be of value. It is also written from the Cognitive-Behavioural Psychology viewpoint.

Naturally, becoming a more psychologically-capable person is a process. No one is fully friendly, tactful, helpful, and considerate at all times. But one must always aspire to be more.

For example, at one of the workplaces I had, whenever I told my boss that I didn't finish what he asked for, but instead found out something else of importance or made some other progress he said "So what you're saying is that you didn't achieve anything.". This is a classical case of the **"All or Nothing Thinking"** cognitive error, which is given in "Feeling Good" as a possible primary cause of depression.

This attitude caused me to think and feel that nothing I could do, would possibly please him, and that he was demotivating on purpose. This was one of the reasons that caused me to assume a variation on clinical depression called "Hypomania" (= below-Mania) [http://en.wikipedia.org/wiki/Hypomania], which in turn made me under-productive.

Read Success and Failure Stories

One of the many quotes featured at the signature of a friend of mine is "Learn from mistakes of others; you won't live long enough to make them all yourself". It is important to read success and failure stories of what people did, what mistakes they performed and what are their conclusions.

I believe we can learn more from the failures of ourselves and others, than we can learn from successes. That's because in a success, almost every aspect was done right, while in a failure one can look back and say where things went wrong.

You can find many such stories on the Internet, and sometimes you'll hear about them in news sites, blogs, and various forums.

Essential Organisational Software to Have

A Wiki to Prevent the "Knowledge in the Ether" Anti-Pattern

One of the anti-patterns I encountered in previous workplaces is that of the "Knowledge in the Ether" [http://use.perl.org/~Shlomi+Fish/journal/33013]. Quoting from my original post:

> One anti-pattern I noted in a previous workplace was what I called "Knowledge in the Ether": most of the instructions for getting the application up and running, and a lot of the collective knowledge were not written down in a collective place. TDDPirate said he was familiar with it and called it the "Oral Torah" [http://en.wikipedia.org/wiki/Oral_Torah] syndrome.

> The solution for this is simple: set up a wiki for the company, and instruct people to write a note there whenever they need to explain to someone how to do it (or give a link to a previous note), **instead of** guiding him how to do it. While this requires some discipline and getting used to, it can also be done after the fact.
>
> Obviously the amount of knowledge in people's head and in the Ether can never be completely eliminated. But it should be kept down to a minimum.

There was also a comment there that said that if one encounters a repetitive task, they should write a script to do that. And it's a good idea to keep good README files, comments, etc. and to use a standard building procedure, or prepare distribution-level packages for the program.

Nevertheless, a workplace should encourage its employees to note down every useful knowledge and procedure. My friend once told me that in his previous workplace they kept a knowledge-base as a group of Microsoft Word documents stored inside a Visual SourceSafe repository. As a result, people felt it was too complicated to update and maintain them. Eventually, he set up an instance of MediaWiki [http://www.mediawiki.org/] there, which proved to be much more convenient, accessible and quick.

A Bug-Tracker

Another essential program to use is a good web-based **bug-tracker** [http://en.wikipedia.org/wiki/Bugtracker]. Joel Spolsky has an article about the motivation and strategy for bug-tracking [http://www.joelonsoftware.com/articles/fog0000000029.html], which is well worth reading.

The reason you need it to be web-based is so that it would be accessible from all operating systems, so it can be exposed to the outside and so you wouldn't need to install a client on all machines. Microsoft's internal bug tracking system is not web-based (for legacy reasons) and they've been criticised for the fact that Internet Explorer, unlike the Mozilla-based browsers, does not have a publicly available bug-tracker.

Furthermore, in a previous workplace of mine, I was stationed on a Linux workstation (which was the deployment platform and what I like working on), and found that I couldn't interact with our bug-tracker, which only had a windows client. Its more recent version already had a web-interface, but they wouldn't upgrade for fear of breaking the other people's workflow. This indicates another problem in not having a standards-compliant web-interface that can run on most browsers.

Version Control Software

A good team should have a good version control server and clients available to every developer. I originally started the "Better SCM" site [http://better-scm.shlomifish.org/] as a way to concentrate the material for the various alternative available, after I realised CVS has become inadequate for most needs.

My rule of the thumb for choosing a version control system is the following: "When in doubt - use Subversion [http://better-scm.shlomifish.org/subversion/]". See Ben Collins-Sussman's article "Version Control and 'the 80%'" [http://blog.red-bean.com/sussman/?p=79] for some of the motivation for this statement. Another one is that from my impression most of the new breed "Distributed Version Control Systems" tend to have catches for the unwary and other things that don't exactly work as expected, and sometimes have poor portability to Microsoft Windows - a fact which is often a deal-breaker.

Building, Configuration and Packaging

In this day and age, you'd probably expect the new system to be packaged for the target platforms (of which there can be many), to be capable of being configured according to the constraints of the platform and according to various limitations and to properly be able to be built incrementally and tested.

If you're not convinced, refer to the Robert Jones' OnLAMP.com article on "Planning for Disaster Recovery on LAMP Systems" [http://www.onlamp.com/pub/a/onlamp/2004/04/08/

disaster_recovery.html], and What Joel Spolsky says in the Joel Test [http://www.joelonsoftware.com/articles/fog0000000043.html] about "Making a build in one step", and about having daily builds.

So you need a good building, configuration and packaging system. I covered such systems in an earlier article [http://www.shlomifish.org/philosophy/computers/high-quality-software/rev2/#easy_to_compile_and_install], and also maintain a list of links to some prominent ones and to other more comprehensive lists [http://www.shlomifish.org/open-source/resources/software-tools/], so you have a high-quality selection.

But you need to use something, and preferably something common, standardised, well-tested and high-quality.

Choose Your Technology Carefully

So which technology should you choose? I'm not going to give a direct answer, and often there are many factors that make one more appropriate in certain cases than others. Joel Spolsky attempted to answer this question [http://www.joelonsoftware.com/items/2006/09/01.html], but ended up falling into many common misconceptions and prejudices (while still making a few good points). He also focused exclusively on web-development.

I won't repeat this mistake here, but instead give some general guidelines.

Is it Open-source and/or Portable?

One huge advantage to a technology is that it will be open-source and available on several operating systems, including most UNIXes and Windows 32-bit, and on many CPU architectures. While, often you'll encounter many platform-specific bugs or libraries or programs that work properly only on certain configurations (which is expected), you should still have the choice of being able to rely on the technology being present.

For example, Microsoft's Visual Basic [http://en.wikipedia.org/wiki/Visual_Basic] was a closed-source and Windows 32-bit and x86 specific programming language, which had proven to be very popular. However, Microsoft decided to discontinue it completely, and as a result, many of the applications developed for it can no longer be maintained into the future, and important bugs in Visual Basic will not get fixed.

Expect some Unexpected Problems

Eric Raymond's "Tale of J. Random Newbie" [http://www.faqs.org/docs/artu/ch16s01.html] is illustrative of the many problems programmers encounter in highly-proprietary environments. However, even if you're using proven, mature, well-documented and functional technology, you are likely to encounter bugs.

I like Perl a lot, and have recently found some bugs or even crashes in Perl code, which I've been trying to isolate and report. Perl is otherwise very good and reliable, but advanced users, or even intermediate ones are likely to discover many edge cases.

You should make sure you have the capability to isolate, report and possibly send a fix to such a technological problem, and get a prompt fix. This often depends on having good support from the technology vendor or developer.

The Technology's Power

Various technologies and development platforms [http://www.joelonsoftware.com/articles/Platforms.html], vary in their power and cost. As opposed to the possible misconception of "If you want

something good, you'll have to pay more.", often cheaper, or even gratis (or open-source) solutions are more powerful (not to mention more reliable and faster) than their more costly counterparts.

Often, however, the really good solutions will be an overkill. Most experts I've talked with agreed that the Oracle Database [http://www.oracle.com/database/index.html] is the most powerful on the market and "years ahead of anything else out there". However, most run-of-the-mill web applications won't make use of even a small percentage of its advanced features, enough to justify using it instead of, say, the open-source and gratis PostgreSQL [http://www.postgresql.org/].

For many problem domains, Oracle would make the most sense, but it's still not very useful in the general case.

Habits

It makes sense to choose a technology which you know very well [http://www.joelonsoftware.com/articles/LordPalmerston.html], so you can overcome problems more quickly. However, smart programmers can learn different technologies very quickly. As Spolsky notes [http://www.joelonsoftware.com/articles/ThePerilsofJavaSchools.html]:

> The recruiters-who-use-grep, by the way, are ridiculed here, and for good reason. I have never met anyone who can do Scheme, Haskell, and C pointers who can't pick up Java in two days, and create better Java code than people with five years of experience in Java, but try explaining that to the average HR drone.

If you expect that you will hire the best developers, as I pointed earlier, then they should probably be able to pick up something new fairly quickly. If, on the other hand, you're going to hire bad programmers, then you're likely to be screwed - even if they have extensive experience with their native technology.

So you should choose the best technology - not the one for which you expect to find the most "experts".

Reliability and Reputation

Often a technology will get a bad reputation as being "quirky", "hard-to-get-right", "buggy", "unreliable", etc. This for example, has been the case against MySQL [http://www.shlomifish.org/open-source/anti/mysql/], against PHP [http://www.google.com/search?q=php%20sucks], against Sendmail, and against other technologies. From my experience, such common criticisms often have a lot of substance, and should be evaluated, especially if there are equally affordable technologies, but such of a better reputation.

Maturity, Mindshare and Distance from the Mainstream

Maturity, Mind-share and distance from the Mainstream are three factors that, like it or not, also affect the choice of language. Immature technologies tend to have many non-discoverable, or well-publicised quirks and bugs. Technologies without a good mind-share have fewer people to consult about problems one encounter. And the less mainstream a technology is, the less it tends to be polished or to have essential features, tools and APIs, that developers of more mainstream technologies take for granted.

As a result, such factors should be weighed in, when choosing a technology.

Case Study

I once went to a job interview in downtown Tel-Aviv, which went very well. Then I received a phone call from them that they want me to come and install Fedora Linux on the computer. I did that, and next thing I knew, I was given other tasks.

I was instructed to write a mail-processing framework in PHP. Now since I know and love Perl, I know there are many fine modules for doing that on CPAN (= The Comprehensive Perl Archive Network) [http://

perl-begin.org/topics/cpan/], but there was little of substantial quality for PHP. At a certain time I needed to register at a site, to download the latest version of a PHP library, that was open-source software, because the download required authentication . There didn't seem to be anything better.

I was told that they would prefer to write everything in PHP, because they expected to hire only PHP programmers, possibly without any Perl experience.

That wasn't all. They also decided against using Postfix [http://www.postfix.org/], which is a modern, high-performance and open-source SMTP server, and instead preferred the old Sendmail SMTP server which has a far worse reputation, and an arcane configuration system.

Furthermore, they also decided to use MySQL instead of PostgreSQL, and we ran into a PHP misbehaviour (which was fixable given a lot of PHP know-how) that made us have to restart the connection to the server after every request. Both MySQL and Sendmail were chosen due to political reasons.

With my PHP code barely working and prone to many errors, I decided to quit after about a month, out of being appalled by the bad code craftsmanship I had achieved there. Make sure you don't repeat such a mistake.

Provide an Excellent Customer Support

Joel Spolsky wrote a feature titled "Seven steps to remarkable customer service" [http://www.joelonsoftware.com/articles/customerservice.html], which is very insightful and inspiring. Among other things, Spolsky says there that every support request needs to be fixed twice: first of all by providing a fix or a workaround for the customer, and secondly by implementing a permanent fix in the product, the web-site or whatever, to prevent your establishment from ever getting such requests again.

In addition to what Spolsky says there, I should note that you should provide help in as many channels as possible: from telephone to E-mail to IRC (Internet Relay Chat) [http://en.wikipedia.org/wiki/Internet_Relay_Chat] to providing your own web forums and mailing lists for help, etc. In order to prevent over-engineering and unnecessary forward-planning I would recommend implementing them when there turns out to be a demand for them, and to allow your establishment to grow organically. As Eric Raymond notes in "Necessary Preconditions for the Bazaar Style" [http://catb.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/ar01s10.html] (part of his the "Cathedral and the Bazaar" essay) it is necessary for the first released version of a successful open-source project to be good enough for people to run it, play with it, comment on it and find it useful. Similarly, when creating a commercial product, one should create customer service channels incrementally as they are needed.

A particularly bad example of an awful tech support incident is the suspension of my former Google AdSense account [http://community.livejournal.com/shlomif_hsite/11327.html]. On April, 2009, I was informed that my Google AdSense account was disabled because it caused a "significant risk to Google's AdWords advertisers", and from then on I could not log in to my account to get help. After I filled and submitted an appeal form, I received no reply after a month. Another appeal I sent did not get any response either. I next tried the Google AdSense support forum and received some replies from people who were not Google employees, that did not seem like a likely cause for an account termination, and no reply from an AdSense employee. After I talked with a few Google employees I know trying to reach someone in AdSense, one of them told me they did not know how to contact anyone in AdSense so they'll likely going to be of little assistance. (Which is a big red flag in itself, indicating that Google has become too disparate and inefficient.)

A somewhat better, but still bad incident was with the ShareThis.com button [http://sharethis.com/] that I installed in my site. After they started using Google Analytics in their re-usable widget (which I didn't want), I wrote a support request and was replied with a laconic E-mail that only said: "At this time, there is no way to disable our widget from accessing Google Analytics.". The problem with this is that they are

likely to receive the same complaint from other users like me, and so will have to reply times and again, instead of fixing the underlying problem, like Spolsky suggested. I myself have decided to move away from ShareThis and others are likely to follow suit.

And for a change, I was provided a good support by Github [http://github.com/], which is a popular software-development hub. At one point, I got an announcement that my own copy of a formerly public version control repository was deleted, because the now private repository has been deleted as well. I did not want it and so went on the github IRC channel, and asked for help. I was told it was probably a bug and was asked to file a support request, which I did [http://support.github.com/discussions/repos/2359-public-fork-of-a-formerly-public-repository-was-deleted-without-my-consent]. There it was acknowledged that this was a bug, and I was offered a workaround that worked.

Weekly Reports and Other Ways to Keep Track of Your Employees

Often your employees will run into problems and may be too shy or not have enough initiative to communicate them to you. If kept unchecked, this may lead to frustration, feelings of being trapped or worse. Here are some effective ways I found to avoid such issues, based on my experience from some of my previous workplaces.

The first is to ask every employer to submit **a weekly report**: of what they have accomplished that week, what they plan to finish the next week, which problems they have run into, and what help they think they need. This report can be sent to their manager or supervisor, but they should feel free to send a carbon copy to their peers.

Another method that I came up with, is to have an interactive multi-way Internet-based channel/"chat-room" where all the team's employees can talk and chat. This can be implemented using a dedicated IRC (= Internet Relay Chat) [http://perl-begin.org/irc/], using the more modern XMPP protocol [http://xmpp.org/], and possibly using such dedicated proprietary solutions such as the now disbanded "IBM Workplace" [http://en.wikipedia.org/wiki/IBM_Workplace]. By chatting online about their work (and other stuff) and keeping logs of the conversations, one can get a feel for what the employees are feeling.

Finally, it is a good idea to try to ask your employees once in a while what they feel or if they've run into any problems. Often, a different person can provide a fresh perspective on how to things and come up with creative solutions for how to solve their problems.

Naturally, I'm not suggesting going to extremes such as monitoring your employees every move (as Joel Spolsky notes in his "Two Stories" (about Microsoft and Juno) essay [http://www.joelonsoftware.com/articles/TwoStories.html]), much less installing keystroke loggers [http://en.wikipedia.org/wiki/Keystroke_logging] or otherwise violating your employees' privacy. But it would be a good idea to keep abreast of the problems they run into and see how you can solve them.

Conclusion

Not all the possible advice on how to manage a workplace was covered here. I refer you to the links at the beginning and inside the document for more reading. You should read them and be enlightened.

The workplace I'm talking about is a workplace that aims to employ great programmers to work on great projects, which your developers will love to do. It's not meant for a sweat-shop or one of those companies working on all these low-quality applications for internal-use, especially those that are developed by or for large organisations.

If you want to hire the top developers, you should aim for a top environment and very good conditions.

I don't intend this document to be dogma, but rather to provoke clear thinking of the matter. A lot of bright programmers I've talked with about the vision of a perfect workplace, have claimed that this is an non-achievable fantasy, and that all employers are much more clueless than that. However, while perfection can never be achieved, I've seen my share of good workplaces, and benevolent and clueful employers, and I think many of them are enlightened enough to improve according to my suggestions.

It's not necessary for a programmer to be a wage slave - they need to be treated much better than that, if their employer wants them to be productive and happy. I hope this essay explained how.

Thanks

Thanks to firespeaker [http://jnw.name/] for going over an early version of this article and providing some corrections.