How to know if your software developer is trying to kill you

Your guide to navigating your start-up through the software development wilderness.

Illustrations by Jenny Palmer

The rest by Ben Liebert

Dedicated to my Jenny, who has listened patiently each night for ten years while I blab on about this stuff.

Who is this book for?

This book is for people who want to get some software developed, but have no idea how to get started, nor the processes involved. By the end of this book you'll be able to:

1. confidently select a software developer
2. assess the price your developer is charging
3. understand the basics of software development
4. ensure your application is future-proofed and adaptable
5. "speak geek" and actually understand what your developer is talking to you about

So, let's get started...

How to choose a developer

The single most important decision you make now is selecting the person that converts your idea into actual software. Everything you say or show your developer will be filtered through their own experience, preferences and expectations of how software should behave, so you need to find a good one right from the start.

Unfortunately, and as with most service industries like lawyers or accountants, you often don't realize you've made a bad choice until something goes wrong - and that could be years down the track. The key to mitigating this is to ask the right questions and look for clues early in the relationship.

Availability

If your website goes down in the middle of the night, or a customer has an urgent feature request, you need your developer to be ready and available.

Now, I'm not talking about when they go on holiday or are away sick - these are predictable and part of life. I'm talking about the times when they already have a 12-hour day behind them and your site goes down as they're walking out the door.

Promptness and availability are the types of things that are best observed over time, and by then it's too late to change (cheaply). But there are still some helpful things to look for early on:

- when you ask for your initial meeting request, are they quick to get back to you?
- if they are late on something, do they advise you early or simply fail to deliver?
- are you able to get their mobile number or just an office number?

By virtue of them being the gatekeeper between you and your product, your developer should understand that being highly available is simply part of the job.

But don't abuse their generosity. If you're pulling them out of family dinners with something you'd like to have - or worse, something that is urgent but late because you were disorganized - then you might be looking for a replacement soon.

Communication

As I mentioned earlier, every idea you have for your product will be filtered through your developer, so it is imperative that you can communicate well with them.

Communication is much more than simply conveying an idea.

A good developer won't just listen to what you're saying - they will also listen to what you're *trying* to say and incorporate this into your planning. You won't need to speak geek because they'll interpret your non-geek requirements accordingly. For example, they may recognize that two of your requirements are competing with each other, or that two could be rolled into one to save money.

Ideally your developer will also have an appreciation of the non-technical parts of your business, such as sales, and advise on how your product can be built to complement these. Remember that they've been building software for a long time and have seen a lot of successes and failures so their experience is a valuable resource. Make sure you listen when they speak up – good communication is a two-way street.

Finally, it helps if you have a personal rapport with the developer. It's really important that you remain *partners* in the project and don't devolve into adversaries. Clear and open communication is key to this.

The best way I know to establish a developer's communication level is to.... communicate with them! Organize a face-to-face meeting and make sure you have a chat about non-technical things too.

Country / region

Exchange rate differences between countries can save you a great deal of money when purchasing products or services, and this includes software development. For this reason, many western companies outsource their software development offshore.

The price offered by these offshore companies can be very appealing – I often see quotes at 30% of their western counterparts – however you must remember that this only represents *value* if the product is delivered to an equivalent level of quality and timeliness.

It is my experience that this is rarely the case. Specifically, communication becomes a major problem when dealing with offshore companies:

- the client and developers are usually in different time zones, so turnaround time on communication is massively slowed

- the developers do not always speak the same language as the client – certainly not to a comfortable conversational level. Many times the offshore company *does* have a fully conversant project manager, but this puts you a level removed from the developers themselves

To mitigate this, you must be prepared to write extremely detailed instructions in a manner that the developers can understand, which ironically requires a skill level akin to an actual developer. This is why the companies that get the most value out of offshore development are other development companies that can out-source parts of their development process.

My recommendation is to start with a local developer until you get an understanding of the development process and then slowly out-source parts of the project as you grow.

Word of mouth

As with most service industries, sometimes the best indicator of a reliable supplier is a good reference from a friend or colleague that you trust. If you don't know anybody with the first-hand experience that you need, it's perfectly reasonable to ask prospective developers for examples of previous work or references from other clients.

Keep in mind though that different software projects require different types of developers. If a friend has recommended somebody, you still have to make sure they are suited to the type of software you are building.

Another good place to get a reference is from other developers. Just as a builder can usually recommend a plumber or an electrician, developers usually have a network of people and companies they trust. If they are not suited to your job, perhaps they know somebody who is.

Price

The price of your developer is one of the first and most tangible things you can ascertain about them. Obviously you want to keep your costs low, but here's a few things to keep in mind:

- a low hourly rate means nothing if they take twice as long to complete the project
- a higher hourly rate generally means the developer is more experienced, which means they are better equipped to handle that one in a hundred disaster that could tie up a junior for days
- your choice of platform will be a large influence on price. Platforms with high licensing fees or development tools tend to have their costs passed on

to you (don't worry, I talk more about platform choice later)
- don't overlook the value of personality and communication. Be happy to pay a bit more for someone you can enjoy working with

Finally, people often concern themselves too much with the up-front and short-term costs of a software developer. Trust me - a talented and reliable developer, who prices themselves accordingly, will *save you money in the long run* with an extensible architecture and software that your customers love.

Is your developer charging you a fair price?

Before you can establish if your developer priced your project fairly, you need to know their intrinsic *value*. I have provided a useful tool for you to calculate value, without alienating or trivializing the multitude of factors that make a developer "good".

There are no hard-and-fast rules around this, however I've found you can get a pretty good idea based on the following:

First, find the baseline price

The baseline price is the market hourly rate when you strip away all the characteristics of the individual developer. It's

pretty easy to work out the baseline price simply by searching the internet for developers that:

- build using the platform/language of your choice (I talk more about these later)
- develop in the country of your choosing (remember that different countries charge different software development rates)
- are quoting a price based on the services you need. This is important because there are different levels of development services with different price points (I speak more on the different levels of development later in the book)

Make some phone calls and find out the two most expensive *hourly rates*, average them and then halve the result – this is your baseline price. The reason we average the top *two* rates is to reduce the impact in the event you stumble across some super-famous developer that you'd never reasonably employ anyway.

Next, factor in the developer-specific attributes

We halved the baseline price above because the second half of a developer's value can be inferred by the factors in the table below. For each factor, I have indicated the importance by placing a percentage against it. All you need

to do is select an appropriate percentage from the right-hand column and add that to your baseline price…

Aspect	% baseline
Experience An experienced developer will charge more by the hour, but remember that your *overall* costs shouldn't change by much because: • they will take fewer hours to build your product • their work is less likely to result in bugs that need fixing • their design choices early on will save you money when extending the system later	Add 2% for every year experience, up to a maximum of 20%
Reputation If a developer has a good reputation, they are more likely to be getting work via word of mouth and, consciously or not, will slowly raise their price to suit demand. From your point of view, a good reputation gives you assurance that they can do what they say they can do. You can determine reputation by searching for them online or by calling prior clients of theirs. The types and sizes of past clients can also be a helpful indication of their value - big name clients are less price-sensitive and tend to put more emphasis on reliability and experience.	You cannot find anything about these guys: 0% You have a personal recommendation from somebody: 10% 80% or more of the feedback you can find about them is positive: 20% As above, plus you recognize (and trust) the people/companies that provided the feedback: 30%

Aspect	**% baseline**
Office & overheads An office can be a good indication of the size and type of a developer's business. It costs money for floor space, which will ultimately be reflected in their rate. That doesn't mean that somebody working from home is necessarily passing those savings on to you. However, in my experience, the location of a developer has a measurable correlation to a developer's worth.	Works out of their mum's spare bedroom: 0% Home office: 5% Modest office: 10% Really flash office: 20%
Staff Managing staff (or sub-contractors) takes time, and chances are this time is being charged against your project. Of course there are benefits to having multiple developers working on your project (e.g. cost savings by using junior developers when appropriate), but don't forget there's an overhead attached to that flexibility.	Sole contractor: 0% Works with just their cat: 1% Less than 5 staff: 10% 5 or more staff: 15%
Non-tangibles Aspects like responsiveness and communication vital to your working relationship and should be considered when evaluating the 'value' of a developer.	This leaves 15% for you to assign based on your own personal interaction with the developers. You can scale this however you like, but if you're not sure what to do (or haven't met them), just assign the full amount to everybody you are evaluating.

You'll notice that the percentage column sums to 100%. This means that if your developer excels in everything, the final rate will come out equal to the most expensive developer you found when researching your baseline price. Clever huh?

As you evaluate developers, you can use this handy table to get a feel for their value. I've thrown in a couple of examples so you can see how the maths works.

	Rate	Base price	E	R	O	S	NT	Value
ACME 1	$80/hr	$50/hr	10%	5%	0%	10%	10%	$67.5/hr
ACME 2	$70/hr	$50/hr	20%	20%	20%	10%	10%	$90/hr

Sweat equity

Sweat equity is a colloquial term for getting paid in shares or interest in your company, instead of cash. A typical example is that a developer may be offered 5% of your company if they work for half their hourly rate.

If done fairly and openly, sweat equity is a pretty good way to get a good developer at an affordable price, especially when you are unsure what revenue your product may generate. It also means that your developer is “invested” in the success of your project and more likely to go the extra mile.

Just keep the drawbacks in mind too:

- your developer has other clients. If you are paying them a lower hourly rate, they may be inclined to go for the short-term benefits of the other work first
- an invested developer will want more say in how the product evolves. Even if their ideas are good, beware of having too many opinions and styles contributing to your product. Mutual interests can be hindered by competing ideas, so I recommend that you outline responsibilities from the start of your contract so that everybody knows their role
- the more they reduce their rate, the more sensitive your developer will become to drawn out deadlines

and scope-creep. Every extra hour they work means more money lost (as opportunity cost working for other clients)

- if you replace the developer, you may have unwittingly given away 5% of your company to somebody who is no longer involved

---

A good tip is to pay equity as a share of annual revenue, for every year that they work for you. This means you're only exposed for the next 12 months if the relationship sours

---

Price escalation

Beware that some developers will purposefully low-ball their initial prices on the expectation that they can slowly increase them as your relationship becomes more entrenched (to be fair, this is common to all service industries, not just software development).

Most developers I've worked with are fair and ethical, but here are a few ways to keep on top of things:

- as your product evolves, the time it takes to maintain it will also increase and with it your development charges. This is natural and to be expected. But if

their output suddenly halves, they might be taking advantage now that you're an entrenched client

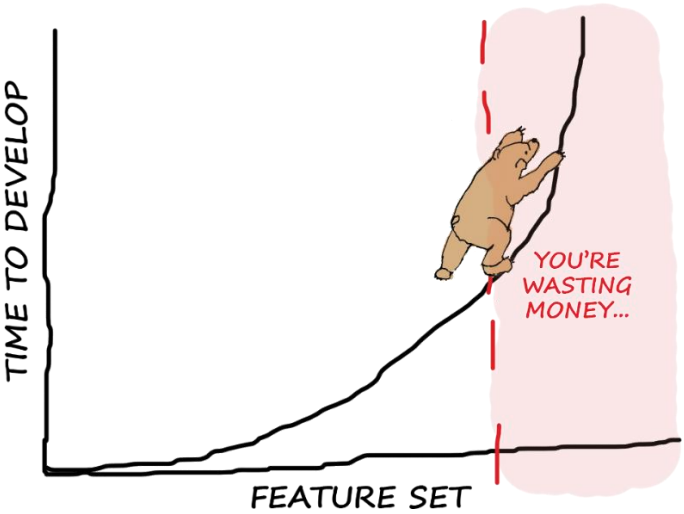
- if your developer is working by the hour, it is reasonable to ask for a breakdown of how that time was spent. Most developers use time-tracking software so this should not be a problem. Get into the habit from day one so that it doesn't feel accusatory if you begin later
- if you have access to the *source code*, you are technically able to track how much code is being written. Disclaimer: code is about *quality* not *quantity*, so you're looking for averages and changes over time
- don't over-react on the first suspicion of overcharging. You'd be amazed at how some things which appear simple to you may in fact be extremely technical under-the-hood. But the converse is true also and you'll probably find cost savings in another part of the system if you look for them
- above all, *communicate* with your developer – remember, you're on the same team

Pragmatism

The time it takes to develop a software feature generally increases exponentially as more features are added. Or, another way of looking at it is the good old fashioned 80/20

rule - the last 20% of your software feature will take as long to build as the first 80% did.

If yours is a multi-million-dollar company, you can probably afford to push features to 100%. But when you are starting out you need to get the best value for money you can - and this means knowing how far to push a feature.



Unfortunately for you, it's not always apparent to non-developers where this line begins and ends so you need to rely on your developer for guidance. In some cases, this may mean building code which the developer considers to be of lower quality (temporarily bruising their ego), but knows is in the best interests of the business. This is what I mean by a pragmatic developer.

Unlike many of the other attributes in this chapter, you don't need to know how to recognise a pragmatic developer - you can usually just make one. Tell them from the start that you value pragmatism and want them to view the software from both a technical and business perspective.

Other skills and interests

I've always found that developers with skills and interests outside of computers are more creative and easier to work with because they have more varied influences in their life, and potentially more in common with their non-technical clients. See if you can glean a little about these aspects of their life when you are speaking with them.

Within the technical sphere, there is also plenty of scope for variety. Ask how your developer keeps up with tech trends and/or participates in the tech community. This could be as simple as subscribing to industry newsletters or attending monthly meetups.

Activities like attending tech meetings or contributing to an open-source project show a developer's enthusiasm for their craft beyond their weekly pay-check and are a really good sign.

Changing developers

Hope for the best, prepare for the worst. If you are unable to continue with your developer for any reason, you need to be able to salvage what you can and hand it over to the next person as quickly and cheaply as possible. To do this, there are two things that you must possess *before* you need them – source code and documentation.

Get a copy of the source code

Source code refers to the programming your developer has done to build your software. It is just a set of files stored on a computer - the same as any other collection of computer files.

While in the development process, these files are on the developer's computer - if he or she disappears, and their computer goes with them, your software is lost. Forever.

A common solution to this is something called *source control*, which is a piece of software that the developer uses to track the work they do. One of the features of source control systems is that they allow multiple people to view the files. So, ask if your developer is using source control and if so, simply ask for your own login. You can then download the code whenever you like.

Failing this, there are more manual systems like popular file-sharing sites or even just asking them to email the source code to you once a month. Whatever mechanism you choose, make sure you do it from day one and *don't let it lapse*.

Documentation

Once your new developer has the source code, they need to know how it works. A lot of software is self-evident and easy to follow even for an inexperienced developer, but it can very quickly get complicated. Your new developer will be able to work things out eventually, but it would be a lot quicker if the software was *explained* to them.

This is where documentation comes in. As I'm sure you've guessed, this is simply a case of your developers annotating their code as they go along. The level of documentation ranges from massive multi-page files with working examples,

down to simple one-line comments within the source code itself.

For this purpose, you definitely don't need the former (it's expensive) - just ask that your developer maintains good coding practice and comments on their work as they go along. Your new developer will be very grateful for it.

Changing software developers is not an easy process, but there's no benefit in pretending it will never happen to you. You'll make life a lot easier for yourself if you take these simple precautions early on.

Platform selection (the tech behind your software)

Did you know that humans around the world speak approximately 6,500 different languages? While each can do pretty much the same thing, they have their individual strengths and weaknesses. For example, some are easier to learn, but less expressive and others can convey just about anything you like, but it takes a full minute to get it out. In some cases, the language is spoken by millions of people and in others it is spoken by just a few dozen.

Similarly, within software development, one is provided with a bewildering range of development language options. And

just like spoken languages, they each have their strengths and weaknesses which need to be considered in the context of your own software's requirements.

It would be reckless for me to suggest a development platform without knowing what you want to build, but there are a number of considerations common to all software products that can help you decide. If you are technically minded and like a bit of internet-searching, you may wish to work through this chapter on your own and then find a developer to suit. But for most readers, I expect you will take this chapter to your developer and work through it with them.

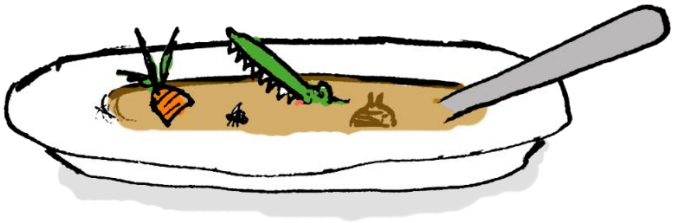
On that note, your platform selection and developer selection usually go hand in hand - so make sure you read these chapters together before you make any decisions.

Open-source software

Bruce is a chef who is famous for his delicious tomato soup. People flock to his restaurant from miles out of town and naturally he keeps the recipe a closely guarded secret, lest another chef cut in on his business. In this respect, we would call Bruce's recipe "closed-source", meaning that

while we can see the end result, we cannot see how it is made.

Later in life, Bruce decides he's had enough of the restaurant business and retires. Without a restaurant to serve it in, Bruce releases his soup recipe so that other chefs may serve it, or people may even make it at home themselves. Bruce has essentially "open-sourced" his recipe.



In a similar way, a lot of modern software is open-source, meaning that anybody (usually another developer) can see exactly how the platform was built. This initially seems like a ridiculous idea (if everybody knows Bruce's recipe, they will make the soup themselves and his sales will plummet). However the issue is not so cut-and-dried - open-source software has some compelling benefits for its vendors:

- open-source software fosters a community of trust and transparency amongst its developers

- other developers (many better than the vendor themselves) will spot and fix bugs and contribute them back to the community

Note also that while it is possible for someone with technical knowledge to copy and use the software, if they do so their version will become incompatible from the ‘base’ version and they will forego any the aforementioned updates or bug-fixes from the community and/or vendor. So this is actually lower-risk than you might initially think.

However, open-source software also suffers from a variety of ailments:

- you need a strong governing body to keep the software updates on track and in a consistent direction for the software’s evolution
- because the software is contributed to by a wide variety of developers, there is no single accountable entity to address problems when things go wrong
- in-fighting and personal or political differences amongst members of the open-source community have brought many open-source projects to a standstill

*To clarify, when talking about open-source here I do not mean that you open-source **your** software product (that will be just for you). Instead I'm talking about the platform that your developer uses to build your software.*

Closed-source

Closed-source is simply the opposite of open-source. The providers of closed-source software do not release the source code and therefore nobody can see how it works. Other developers who use the software must accept it "as-is" and are unable to copy or modify it in any way. The drawbacks to this are obvious:

- any bugs in the closed-source code must be tolerated until the provider releases an update (having said that, most bugs can be worked-around by your developer, and critical bugs are usually fixed very quickly)
- if the software does not work in the way your developer wants, it is not possible for them to adapt it

The main benefit of closed-source software is its value to the provider. Many thousands of hours have gone into producing it and the provider wants to protect their

investment so they will work hard to keep it in good shape. Closed-source software is often *purchased* by your developer, which gives the provider a further incentive to make sure it is of high quality.

The open vs closed debate has very passionate advocates on both sides, but I think it's actually a minor point in the greater scheme of your product:

- open-source is usually free (or at least cheaper), but any money you save is minimal compared to other costs in your business
- open-source is touted as having a stronger and more giving community but in my experience there is nothing lacking in any of the 'paid' communities. Developers in general are very willing to share their time and expertise, regardless of the platform
- there are other much more important considerations in selecting a platform, like speed and developer support
- most systems use a variety of platforms and languages, so you will probably end up using a bit of both anyway

Ultimately I suggest you simply ignore whether a platform is open- or closed-source and evaluate them on their other merits

Speed of development

People always enquire about my hourly rate but not once have I been asked how *fast* at developing I am. The choice of development platform greatly impacts how fast your developer can churn out code, so if someone's charging 150% more than the last guy but they're operating at twice the speed, you're getting a bargain.

In general, you'll get better speed out of:

- more popular platforms. There is better community support and examples that your developer can leverage
- newer platforms. Software platforms that have been around for decades are obligated to remain compatible with the software products that originally used them, making it harder for the platform to incorporate modern time-saving features. The documentation for older systems can also get quite large, increasing the time it takes for your developer to find what they need

- specialized platforms. For example, if a platform has been designed to specifically build websites, it's probably going to be quicker than using a platform designed to build mobile phone applications. The flip-side of this is that if you want to support both web and mobile later, you'll probably need a different developer

How many developers are using the platform?

In general, the more obscure the platform or language, the fewer developers you'll find that can use it. This can manifest badly in a few ways:

- if your developer decides to quit or gets hit by a bus, you may not be able to find a replacement, meaning all your existing code will be very difficult (or impossible) to upgrade
- due to simple supply-and-demand, you may find yourself paying pretty high rates to your developer. If the language lends itself to rapid development, then this may be justifiably offset, but it's something to keep in mind
- fewer developer options simply means you can be less fussy about other considerations, such as the geographic region they live in

- a low uptake or small developer pool reduces the impetus on the software provider to maintain and incorporate modern programming features. Upgrades can slowly grind to a standstill, leaving you with a product built on dead technology

---

A good way to check developer support for a particular platform is to search for it on a job-hunting website. The more hits you get, the more developers are using that platform.

Community support

Loosely related to developer support, this refers to the amount of technical documentation, blog posts and forums that your developer can find *from other developers* online. No developer knows everything, so a helpful, knowledgeable community of like-minded people is an essential part of doing a good job.

One of the measures of platform ‘popularity’ is actually the amount of community support online, so yeah - more popular platforms have better community support.

To check for community support, jump on the internet and search for your platform plus words like "community", "blog" or "forum". You'll soon get a feel for how big a community is behind each platform.

Platform vendor support

If you build houses for a living, you will have a collection of tools to assist you, such as a skill saw and a hammer. In a similar way, a platform provider gives your developer the tools they need to build your software. If there is a bug or usability issue with the platform, who can your developer turn to for help?

This may be a rare case, but if your site goes down in the middle of the night before your ad-campaign is due to run, it's nice for your developer to be able to pick up the phone and talk to somebody.

The main thing to note here is that if your developer has not paid for their software (e.g. it is open-source), the concept of a 'vendor' is a moot point. Their only help available will be from the community of other developers or a third-party 'support service' (if you have the foresight to sign up to one earlier).

Further to this, many modern software applications use some kind of centralized server to host their software in the cloud and in these cases, you will also need support from the *platform provider* if your site goes down. Some providers have additional support plans which you can pay for (ahead of time), but it's definitely worth checking out their forums and contact information as part of your selection process (by the way - if you can't easily find their forums or contact information, that is a *big* red flag).

A slave to fashion

If you keep an eye on tech news, you'll probably have a few platform names swirling around your head which are being touted as having "massive developer uptake" or being the "next big thing".

Unless you're capable of stripping away the hype and assessing things objectively, I strongly advise you *not* to use these platforms. The history of software development is littered with the carcases of once-fashionable platforms or paradigms that fell by the wayside for a variety of reasons (or sometimes apparently no reason at all).

Remember that in most cases, these platforms are simply doing exactly the same thing as existing platforms, and if it is truly revolutionary then the other players will catch up pretty

quickly. Any savings or hype you get from using a fashionable new platform will be greatly offset if you have to re-write your software from scratch in a couple of years' time.

My recommendation here is to use a platform that is a minimum of three years old and at least at "version one".

Hidden costs

Up-front costs of development are easy enough to identify, however your ongoing development may have a few nasty surprises, so don't forget to ask about:

- licensing of *supporting* software. This refers not to the platform you choose (I speak more on platform selection below), but the software such as IDEs and development databases which your developer needs in order to do their jobs. In most cases, your developer will already have what they need, but it doesn't hurt to ask
- per-developer licensing. Some software licenses are charged by the product, in which case you're fine. But others charge per-developer, so if you expand your team later, your licensing costs will expand too.
- site hosting fees. Unless your software product is installed on people's computers or mobile devices, it is likely that you will host it on a connected server or in the cloud. This will incur hosting fees. In general, open-source software tends to be lower cost because it doesn't have a license component built in and there is a 'free' ethos throughout the community
- high-traffic hosting. Many cloud-based hosts charge according to how much activity you run through your servers, so if your software platform communicates in

a particularly verbose fashion you may incur higher charges. A similar example is using an in-memory cache in front of your database - this takes up space and traffic and you *will* be charged for it, so factor this into your decision

As you can see, there are a lot of moving parts here and it is actually *really* tough to accurately predict your charges. Many hosting companies and platform vendors purposefully complicate their pricing tables in order to highlight their feature line-items ("only 1 cent an hour!"), which does nothing to help. Furthermore, price-based competition means your projections are constantly changing underfoot anyway.

Please don't give your developer too hard a time if they can't give you an exact figure. Generally, if you're using a popular cloud host, you're probably doing okay.

A (not so) quick note on architecture

This is not a technical book and you may not be a technical person. However, you will soon have a technical *product* and you need to understand what you're getting. The decisions initially made as part of your overall architecture will:

- impact the quality and speed of development of your product for the rest of its life
- facilitate the introduction of new product features - including those that you can't even envisage now
- cost you massively to undo

Each of these sections is worthy of a book on its own, but I will keep it brief here. Instead, my intention is that *you* can

make sure that your software developer knows what they're talking about. And if they don't, then I recommend you get a new developer.

Automatic testing

Automatic testing means writing software that tests other software. Yes, that sounds like an eternal spiral (who tests the tests??!!) but in practice it's a damn good idea. To help explain why it's important, here's a non-technical example:

Bruce is a master criminal and every weekend (he still has a day job during the week) he drives across country to his lair. Bruce knows that he needs exactly half a tank of gas in his car in order to make the journey, so at the beginning of every journey he glances at the fuel gauge to make sure he has enough.

After a particularly evil and lucrative scheme, Bruce decides to reward himself with a brand new Pollut-o-matic exhaust system. However, he doesn't realize that the extra weight added by the Pollut-o-matic means he now needs a full tank of gas every weekend and he ends up stranded in the middle of nowhere.

Obviously what Bruce should have done is re-test his entire car after the upgrade. This would have immediately shown the shortfall in fuel and he could have adjusted it

accordingly. The problem is that running tests like that is boring and time-consuming, but if he ran this test automatically after every change then he could be assured his car always behaved as expected.

That’s pretty much the essence of automatic testing - identifying how the critical parts of your system should run and then *automatically* testing them to make sure they aren’t adversely affected as the system evolves.


TESTOMATIC 2000

The main kinds of automatic testing are unit testing, integration testing and UI testing, but I'm not going to go into their details here. However there are two things to take away:

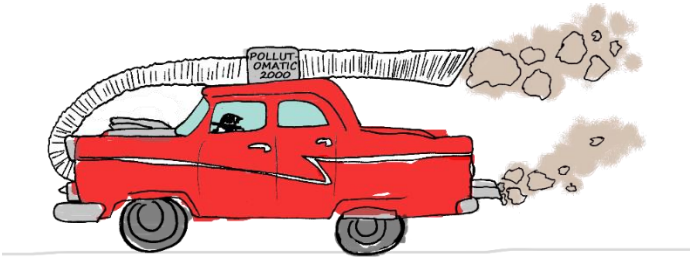
- a good developer will *always* mention automatic testing
- you shouldn't do it

---

*Yes, you read that correctly - I'm advising you **not** to conduct automatic tests. At least not yet.*

Automatic testing is an essential part of a robust software product, but it also adds between 30-50% to your development costs. When you're prototyping or market-testing an idea, that kind of overhead may mean there's no product left to test at the end of it all. So how about you do this instead...

- *listen* and agree when your software developer brings it up
- develop your architecture *in anticipation of* automatic testing
- just don't actually build the tests (yet)



If you follow this plan, it will take a *lot* of discipline (and money) later to retro-fit your tests, but it's really important that you do. Tests are boring and expensive, but for that 1% of cases where it saves your system from going down, tests are your best friend.

Remember also that you don't need to test the entire application - just the critical paths like logging in.

Extensibility (expecting the unexpected)

As the needs of your customers change, so too will your product features. *Extensibility* is a rough measure of how easy it is to change or modify the product without impacting the existing functionality. And it's important because it directly impacts:

- your ongoing development costs
- your developer's ability to roll out new features quickly and easily

- the risk of breaking old functionality as you roll out new features (this is called a *regressive* bug, by the way)

Your developer will know the architectural patterns to improve extensibility (you can impress them with phrases like *dependency injection* and *inversion of control* if you like) but only *you* can identify the short vs medium-term functionality, so before you go into any in-depth discussions about software development you need to have a list of:

- the features you need before your product can be used
- the features you expect to have in the medium-term (say, six months)
- the features you expect to see in the long-term (2-3 years away)

Your developer will build the short-term features directly into the product, but your requirements for the medium- and long-term will impact how they design the overall architecture, so it's really essential that you discuss this with them.

An important exception to this rule is if you are building a *proof of concept*. In this case you may forego extensibility in favour of extremely fast development - after all, there's no point being able to easily extend a system that nobody wants.

A proof of concept usually has no sound architectural underpinning, so you'll throw it away and restart afterwards (I discuss it more in detail later in the book).


16

The user experience

User experience (or UX) is a term developers use to describe how your customers interact with your software. I'm sure you've used a website or an app before and been bamboozled or frustrated – this is an example of bad UX. Conversely, as well as simply delivering functionality to your users, good UX will go that extra step to make your software a delight to use.

When selecting a developer, it is crucial that they place a high value on UX. Many developers are highly technical and therefore take great pride in the technical details of their work. They will often dismiss UX as just "pretty fluff" which a designer just tacks on at the end.

This is a regrettable attitude. All the clever algorithms and technical wizardry that your developer builds are presented to the user through the UX. It is a vital and necessary part of building beautiful software.

Your requirements will vary, but to give you an idea of quick UX wins, you might:

- automatically focus the first textbox when a form loads

- allow the user to press the <Enter> key to submit a form, instead of having to click a “Submit” button with their mouse
- avoid technical jargon. If you have a contact form on your site, the button should say “Contact us”, not “Submit”
- guide the user through the site by highlighting the *single* button or action on the page which you expect them to need next
- if a page has plenty of text, make sure it is broken up and logically laid out.
- use colour and layout consistently and judiciously. You should be able to squint your eyes at a site and still get an idea of the important sections
- if your software is web-based, make sure you test its behaviour on a variety of screen sizes

As you can see, a lot of UX is logical. But it is easily overlooked in many applications – make sure yours isn’t one of them.

---

*Don’t forget the **experience** part of UX. Many people think it begins and ends with how your software looks, but that is only one part of the experience. Equally important are things like speed, accuracy, anticipation, relevance & follow-up*

To get a good UX you need a specialist, and this is most often *not* the nerd that develops the main software for you. However, your developer is still heavily involved in the design because:

- they dictate the constraints that the designer must work within. The extent to which your designer appreciates these constraints can massively impact the overall price and user-experience of your system.
- depending on what *platform* you choose, your developer will often be the one integrating the design back into the software. If your designer and developer have a complementary understanding, then you are more likely to get a final product that actually looks like the original designs

---

For these reasons I recommend obtaining your software developer first and then the designer. Often your developer can recommend a designer that they have worked with previously, which is a big help.

Deciding what to build. And when.

Envisaging the end goal for your product is pretty straight forward, but working out the steps to get there can be technically challenging for the developer and an exercise in dispassionate patience for the client.

The worst thing you can possibly do is build your entire software product before you release to market:

- it will cost you a lot of money to build
- it will cost you even more money to un-build the features you no longer want
- trust me, many of the features you build will not be used by your customers
- and trust me again - your customers will come up with other features that you haven't thought of yet

For these reasons, I recommend you develop your product in three initial stages:

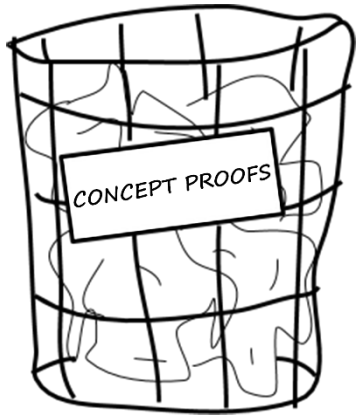
1. a proof of concept
2. a minimum viable product
3. version one

Let's look at each in a little more detail below.

Proof of concept

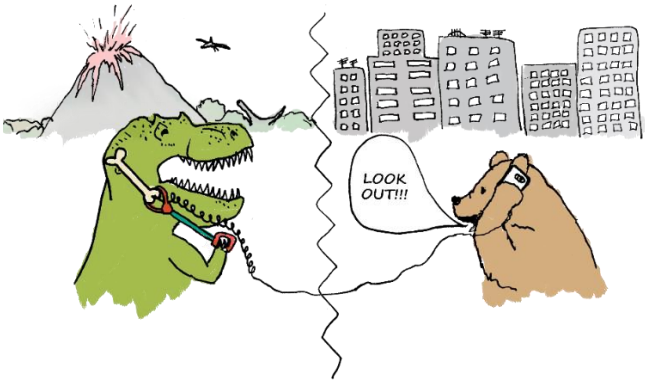
With respect to software development, a *proof of concept* is intended to make sure that your product is technically possible – a little like the software equivalent of market validation.

Building a proof of concept essentially involves writing the absolute minimum amount of code required to test your theory and padding out the rest with the cheapest solution you can find.



For example, if your idea is to make a telephone that can call through time, you may need to build a simple piece of code that can send binary transmissions through

wormholes. If you need some supporting infrastructure such as a login form, you could fake it with a PowerPoint presentation (i.e. you don't necessarily need a developer)



.

For the most part, a proof of concept is a *throwaway system* because you're building it as quickly and cheaply as possible with no thought to the future. For this reason, you may also break some of my guidelines above about developer selection by getting a cheap junior developer if possible, because you don't necessarily intend to continue with them.

Remember, you may not need a proof of concept if you are confident there are no technical challenges. It's always worth running it past your developer first though.

Minimum Viable Product (MVP)

The MVP may well be the toughest software release you ever do. It consists of the absolute minimum amount of functionality you need in order to get your product in front of your customers. Most clients really struggle with this because they feel that either:

- feature XYZ is a ‘showcase feature’ and will really set the product apart. I advise you to think long and hard about these types of features because they are usually expensive to build and quite often your customers will not find them as important or impressive as you think they are
- releasing an unfinished product will give potential competitors a whiff of what you’re trying to do. I can’t comment on the ruthlessness or perceptiveness of your competitors, but I will point out that first-mover advantage is of little use to you if you don’t have the funds to maintain it because you’ve blown them all on your full-featured system

---

My overarching observation is simply this: every piece of functionality costs you money, so don’t build it until you know you need it.

Here some practical examples of getting your product down to MVP:

- maybe you don’t need to make people log in to your system - just let everybody do everything on the site
- you don’t need a password-reminder system. Just have the page send you an email and you can go into the database and generate a new one yourself and email the customer back
- you don’t need a contact form on your site - just include an email address instead
- if your product has a back-end system which you use to populate the database (e.g. registering new products in your e-commerce system) then you probably don’t need to build any forms to help you manage it – just get your developer to insert the information directly into the database as required (it’s usually a pretty quick and simple job)
- don’t build a system to serve 100,000 concurrent users if you’ve only got 100 to start with
- don’t worry about a flashy landing page describing what you do or displaying an introduction video

Your options will differ depending on your exact product, but the point is to be imaginative and ruthless otherwise your project is vulnerable to "death by a thousand features".

MVP architecture

Because your MVP will serve as the base for version one, the one place where you should *not* skimp is your software architecture. Your developer will have good tricks for "stubbing out" or "mocking" functionality which you don't need for your MVP. This means that they can prepare your system for expansion later without actually implementing it now – giving you the best of both worlds.

Remember that any short-cuts you make on your architecture during the MVP will cost you doubly later on because you'll have to pay to undo them as your system evolves.

Scope creep

Sarah has decided to build a bicycle. Because she is highly intelligent & responsible, she has taken the time to read this

book and already established that her minimum viable product is simply to get a user from A to B.

"No problem", she thinks to herself as she grabs a hammer, nails a couple of wheels to a plank of wood, straps her cat (her reluctant test pilot) to the top and pushes him in a straight line. "Job done!"

…but then she thinks, "well perhaps it should have brakes", so she adds brakes. "Oh and what about steering?" - so she adds this too. Before she knows it…she's built a whole bike. There's even a tiny fire-retardant suit for her cat. It's pretty cool, don't get me wrong, but it's much more than she needed and (just between you and me) much more than she could afford.

In the software industry, this is referred to as scope creep and it is a cancerous mind-set that must be excised at its first sign.

Scope creep is responsible for blowing budgets, exceeding timeframes, disappointing customers, creating bugs, polluting your product vision and turning developers into nervous wrecks. It is prevalent in every phase of your development - not just the MVP.

To mitigate scope creep, just ask these questions of yourself every time you consider a new feature:

- what problem is this feature solving?
- will this feature get me more money or customers?
- what is the worst thing that will happen if this feature is not present?
- is there a cheap alternative or workaround that I can build instead?
- are there other, more important features that will need to be deferred in favour of this one?
- how does this feature contribute to each of my product, business and overall strategy goals?

Technical debt

When you take out a loan from the bank, you are committing your future-self to pay for something which you want now. In a similar way, *technical debt* means taking a shortcut with your software now, on the understanding that you'll fix it up later.

Examples of technical debt might be:

- not encrypting your passwords in the database
- writing your website for desktop only, without considering how it will look on a mobile phone
- omitting a caching server in favour of direct database access
- copying entire sections of your software to other parts of the system, instead of adjusting the original code to accept multiple use cases

Technical debt is not necessarily an evil thing. Just like a loan, it can be used to your advantage - for example, to quickly develop and test a new function on your beta testers.

But also like a loan, if you don't stay on top of it, it will spiral to unmanageable proportions. I have seen systems where developers shaved a day off their development time, only to pay for it in *months* of development years later as they tried

to untangle everything. Remember that technical debt is usually at the architecture level, so it underpins all subsequent development as well.

A way to keep on top of technical debt is to simply maintain a list of it with your developer and commit regular blocks of time (e.g. the first week of every month) to go back and redress any issues. The earlier you fix things up, the cheaper it will be in the long run.

The development process

Once you've selected your developer and decided on your platform and architecture, the development process begins. This chapter will guide you on what to expect and how to manage things as they proceed.

Your development methodology

In the early days of software development, the prevailing project management paradigm was: do your market research, design the system, get it developed, pass it on to testing, then put it out to market.

This became known as *waterfall development* because each stage was completed in its entirety before moving to the next. If you are technically minded, you may be tempted to

use this approach yourself – intending that by the time you chat with software developers you've already checked your market and designed the system (just hand them the spec, and go!)

However, as technologies, communication and user expectations have evolved, most software development has changed to what is known as an *agile* methodology. Using this approach, the client (you), developers and end users all work together in iterative cycles to design the product.

Most people's concern with agile development is that you are putting an un-finished or un-tested product in front of your customers. However in practice this is rarely the case (you can still test and conduct market research within each of these stages) and it is offset by many other benefits:

- the contribution of real-world use cases provided by your customers
- your developers and users will appreciate the participatory culture
- your software developers have seen a lot of businesses start and fail, so they can often advise in other non-technical aspects of your business
- if something goes wrong, your agile processes will easily facilitate rapid deployment of bug fixes and product updates

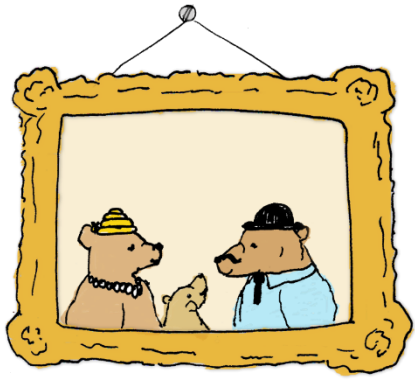
- you will be more responsive to customer feedback and changing competitor trends

User-driven development

Agile development also opens the door to another modern paradigm - *user-driven development*. This simply means that as well as designing the system based on your own expert knowledge, you constantly obtain feedback from your users and integrate it into the next release cycle.

I cannot emphasise enough how valuable this resource is. Your user-base is a pool of real-time market researchers & product testers with a vested interest in the success of your product. And they're usually free!

Good product and process design can easily include these people into your development lifecycle, so make the most of them.



An easy start for your user base is your friends and family. They know you and your limitations and

they are nosy enough to be prepared to work for free. Hopefully you have the kind of friends that will give you honest feedback if the product is not working correctly (if they don't do this, there's not much point using them actually).

After you exhaust your friend and family network, you need to quickly expand your beta testers to include:

- people that don't know you personally (they are more likely to give honest, un-biased feedback)
- people that are actual end users of the final system. There's little point building a professional accounting product and then getting your ten-year-old niece to use it.

There's a bit of a lead-time in getting beta testers, so I recommend you get started before you have a product to show them.

Strategies to attract testers are up to you, but things that I've seen work include:

- have some kind of 'pre-signup' form which gives benefits to early adopters of your product

- allow your beta testers to invite a limited number of their own friends to also be beta testers - create a sense of opportunity and privilege by purposefully keeping this number low
- identify specific customers you'd like to have and invite them in an advisory capacity - many will be flattered and willing to participate if it means influencing the product design in a direction of their choosing

Deployments & upgrades

Once your software is developed or has a new feature added, the job is not done - it still has to be deployed to the customer. If your software is on the web, this means copying it to a server or to the cloud. If your software is a mobile app, it means deploying through an App Store. Many software products are, in fact, over multiple platforms, so you may end up making multiple deployments.

People underestimate how much effort goes into deployments, but trust me - it's significant. What's more, it's the first experience your customer has of the new features so it is critical to get it right. Large software companies have dedicated 'ops' and infrastructure teams to manage this

stuff, but in your case it's probably going to fall to your developer.

The purpose of this section is not to provide technical assistance, but instead to give you a little bit of warning that this is a legitimate and necessary part of the software development process. So when your developer starts talking about "continuous integration", "rollbacks" and "integrated testing", don't be angry at the extra costs, be grateful that they know what they're talking about (check out the geek-speak appendix at the end of this book for details on each of these).

Depending on your requirements, your deployment processes may be as simple as a file-copy to the server, or more complicated than the software itself. Discuss your options and requirements with your developer to find the right balance, and be prepared to add it to your budget.

There are different levels of development

The car-wash down the road from me has a big sign outside offering their different services. For $20 they'll pretty much spray your car with a hose. For another $20, you'll get it waxed and for another $20 they'll clean the inside too. However, regardless of which service you choose, I'll bet you would still describe your car as being "washed".

In a similar way, people think of their software as being "developed", but within that there are a multitude of services which your developer may or may not be doing.

This ambiguity might lead to problems for you later if there is a disparity between what was provided and your implicit expectations.

Most developers don't have a helpful sign like my carwash does, so it is up to *you* to find out beforehand exactly what services they consider to constitute "software development".

I've helpfully listed the most common services in the table below:

Service	Do you need it?
Assistance with establishing your software requirements	Depending on your level of technical ability, you may already have a complete specification ready to hand to the developer. Otherwise you will need your developer to assist you with this
Building your software	Um, of course
Building an extensible architecture	Yes, unless you're doing a proof of concept
Documentation	Prudent, especially if you are expecting to switch developers or take the project in-house later. Documentation *can* be retrospectively added at very little overhead, but this is no good to you if your relationship sours suddenly
Automated deployment processes	If you plan to release regularly (e.g. at least once a week) then this will save you a lot of money and stress
Automated tests	Yes, unless you are in the very early stages of a product (e.g. MVP) or you are prepared to manually test every time you release the software

Service	Do you need it?
Design & user experience	Depends on your audience. If people's first impression of the software is what they *see*, then yes, you probably want design

This table can also explain why different developers will give you vastly different quotes for your software. Some will take your requirements at face value and just deliver what you asked for, while others will implicitly know that you need additional services and include them in the quote. Neither mechanism is wrong - just be aware of it and talk to them.

To conclude...

The most important thing you can do when embarking on your software development project is to *be prepared.* Take the time to understand the different things that go into your project and don't let yourself be overwhelmed or intimidated by technical jargon.

Software development is a specialized field, but that doesn't mean it should be inaccessible to non-technical people. Finding a developer that shares your vision and that you can communicate with is key to bridging this gap.

Have fun, and I wish you the best of luck. If you have any questions or want to share your successes or failures, come find me at www.blackballsoftware.com.

Appendix 1 - how to speak geek

If your developer is starting to talk a little over your head, use this handy translation guide to make sure you're on the same page…

Geek	Human
Abstraction	Taking a specialized concept and identifying the properties common with similar specialized concepts. For example, a bicycle and a car are specialized types of vehicles, or an employee is a specialized type of human. By abstracting common properties (e.g. “top speed” or “first name”), a developer is able to re-use code more effectively because updates need only be made on the single *abstracted* concept, not the multitude of specialized instances. Ultimately, proper use of abstraction makes software code easier to maintain and quicker to update, which in turn saves you time and money
Algorithm	Just a specific piece of code that follows a set of rules to deliver a desired result. Algorithms can be extremely sophisticated (like calculating the required trajectory to break a rocket out of earth’s atmosphere) or extremely simple (like calculating the tax on an invoice)

Geek	Human
API	Stands for Application Programming Interface. Just think of it as a way for two software programs to share data. Pretty common on modern websites.
Assembly	The aforementioned zeroes and ones are compiled down into a single file, which is called an assembly - it's just a file on disk. At the end of the day it *is* your software
Brownfield	Building a piece of software on top of an existing piece of software. This is much harder to do because you have to work within the limitations of the existing architecture. Opposite of *greenfield* development.
Bug	A problem with the software

Geek	Human
Caching	Sometimes your software needs to perform large calculations that require too much time to be delivered when the user requests them. If that is the case, we store the result in a *cache*, so that subsequent requests are lightning-fast. Caching is also used to reduce the use of your servers by storing the results of resource-heavy calculations so they don't have to be recalculated every time the user wants to view them
Carbon-based interface	A facetious description used by developers to describe the humans that use the software e.g. "we're experiencing a problem with the carbon-based interface"
Cloud	Just a fancy name for the internet
Coffee	The fuel your developer requires to generate computer code

Geek	Human
Compile	Software is written over a number of different files in a language that is human-readable. But a computer needs to read zeroes and ones, so when we are ready to use the software we have to convert it - and this conversion is called *compiling* or *building*
Content delivery network (CDN)	Makes your website quicker for everybody to use by making copies in places which are geographically closer to your customers.
Continuous integration	A set of processes which allow small changes to the software to be automatically *integrated* into your main software product. Benefits include quick discrete deployments, quicker detection of regressive bugs and faster feedback loops

Geek	Human
Cookie	When you browse the internet, as well as the stuff you see, other information is also transferred which the developer may need to know. For some reason beyond my understanding, this information is called a *cookie* – you can just think of it as a few letters and numbers at the end of every page. A common use-case is storing whether a person is logged in or not, otherwise you'd have to sign in again every time you switched pages!
Cross-site scripting	Also referred to as XSS, this refers to a method by which a hacker can cause code from another website to run within your own. From there, they can do lots of nasty stuff like read your users' passwords or redirect them to another site altogether
Database	A big file (or collection of files) that store information, such as user logins

Geek	Human
Database (relational)	This is the ‘classic’ database style, where as well as storing data, the database lets you define the relationships between the different types of data. For example, if you had a list of vehicles and a list of people, a relational database would also let you define a relationship about who owns which vehicle
Database (non-relational)	Also known as flat-file or NoSQL databases, a non-relational database forgoes the ‘relationship’ aspect of a database for raw speed. This works because a relational database has to ‘join’ the various datasets when we query them, but a non-relational database just stores the vehicle and person information in a single datastore (no join required). The trade-off here is that the same information is often stored in multiple places, so they tend to be bigger and less real-time

Geek	Human
FTP	File Transfer Protocol is a way of copying files between two computers. When you are ready for your website to go live, a developer will often ‘FTP’ their assemblies to the server.
Greenfield	Building a piece of software from scratch. Opposite of *brownfield* development.
Hotfix	Updating a component software (usually a bug) without having to redeploy the entire piece of software, and without impacting your customers
IDE	Integrated Development Environment. Just a fancy name for the software a developer uses to write code – similar to how one might use Microsoft Word to write a letter.

Geek	Human
Integration testing	Much the same as unit testing, except that we test how the various function points *interact*. For example, if your unit tests were 1) submitting username & password; and 2) accessing the login table in the database, then you may have an integration test which ensured that a user with valid credentials could log in
Minification	Re-compiling your source code so that the file size is smaller and transmits over the internet faster
Plug-in	A piece of software written by somebody else, which your developer can use to fulfil a requirement of your site. Generally pretty good because they avoid your developer having to reinvent the wheel

Geek	Human
Responsive design	Building a piece of software so that it looks good and functions well no matter how big or small the user's computer is. A typical example is a website that runs on both desktop and a mobile phone. In the old days we just built software to run on desktop-size monitors, but these days people access from all kinds of devices so it is much more of an issue
Rollbacks	Usually done when a software update has gone horribly wrong, this basically means reverting to the previous version. In some cases, while an upgrade may take only a few minutes, a rollback may take many hours because you have to manually undo both the mistakes and also any activity which users did on the new version
Server	Just like your normal computer, but it's got a cable plugged into it so that it can be accessed over the internet. We use servers to store things like websites and databases - things which need to be always available to people on the internet.

Geek	Human
Skin	The way your software looks - colours, fonts etc
Source code	The files that your developer writes which are *compiled* into the *assemblies*
SQL	Often pronounced “sequel”, it stands for Structured Query Language and is a programming language developers use to communicate with a database
SSL / TSL / HTTPS	All variations of the technology for encrypting communication between software and the people using it (ie. your customers). Limits the ability of hackers to understand data in the event that they are able to intercept it. A common use case is encrypting passwords when someone signs in to your website

Geek	Human
Sunlight	A naturally occurring warning-system which lets your developer know that it's time to go to bed. The disappearance of sunlight each evening signifies that they'll stop getting client phone calls and the real work can now begin.
Thumbnail	A small copy of an image. Used on websites because of their smaller file size makes them quicker to transfer over the internet
UI / GUI / UX	User Interface or (Graphical User Interface – they're interchangeable) are the bits of your software that people can see & interact with (like buttons). UX is a bit more of a modern term and stands for User Experience. It covers things like how long your software takes to load or if there are nice animations/transitions (like fading in a login form) which make the software a pleasure to use

Geek	Human
UI testing	UI (user interface) is the bits of your software that the user sees and interacts with, such as buttons or page transitions. So UI testing is just making sure these all work nicely.
Unit testing	Breaking your software down into discrete function points (units) and then writing other software to use them in isolation. This means that if a developer later modifies a unit to accommodate a new use case, your test will ensure that it still behaves as expected for the *old* use cases.
Version control	Taking copies of your source code on a regular basis so that you can revert back to previous 'versions' at any time

Appendix 2 – one more thing...

This book has deliberately avoided recommending specific vendors, technologies or even architectural methodologies. The tech environment is changing so fast that anything put to print will be out of date within six months.

However I am constantly reviewing my craft and I usually have a few strong recommendations for anybody that cares to listen. If you'd like to hear our current thinking, you are always welcome to get in touch - www.blackballsoftware.com

Appendix 3 – notes

Now that you're equipped with the right questions to ask prospective developers, you'll need a place to record their answers. Well, look no further as I have helpfully left a few blank pages where you can write notes and compare developers right within the context of this book…

Developer name	
Hourly rate	
Baseline rate (refer calculator above)	
Platform	
Notes	

Developer name	

Hourly rate	
Baseline rate (refer calculator above)	
Platform	
Notes	

Developer name	
Hourly rate	
Baseline rate (refer calculator above)	
Platform	
Notes	

Developer name	

Hourly rate	
Baseline rate (refer calculator above)	
Platform	
Notes	

Developer name	
Hourly rate	

Baseline rate (refer calculator above)	
Platform	
Notes	

Developer name	
Hourly rate	

Baseline rate (refer calculator above)	
Platform	
Notes