You need to build a software application. Now what?

By Frank Zinghini, CEO
Applied Visions

Introduction

It's time to build a software application. Your customers—or your bottom line—are demanding it. Now what? You're not the only one in this position now. Software has become the most dominant player in our world, and even companies that have never created software are finding that they must, in order to keep their products relevant.

So. Now what? Good question—and I guarantee you will have more questions as you go through the development process.

How you address the issues you encounter will determine the success or failure of your software application.

I could have called this guide "How to develop great applications in spite of yourself" because, like most things in life, we can be our own worst enemies when it comes to developing applications. Fortunately, it is possible to develop successfully while avoiding the most common mistakes. That pathway to success, from a manager's perspective, is what I offer in this guide.

You'll notice that we put a lot of emphasis on "your customer" in these pages. We do that when we design applications, too. Customers matter: If they don't like using your product, they won't want it. So software usability is a big part of this process. Usability can be the difference between your product becoming a revenue-generating success, or a profit-sucking disaster.

Creating revenue-generating applications is what we live for at our company, and doing this for countless clients over the years has given us some wisdom. I want to pass that wisdom on.

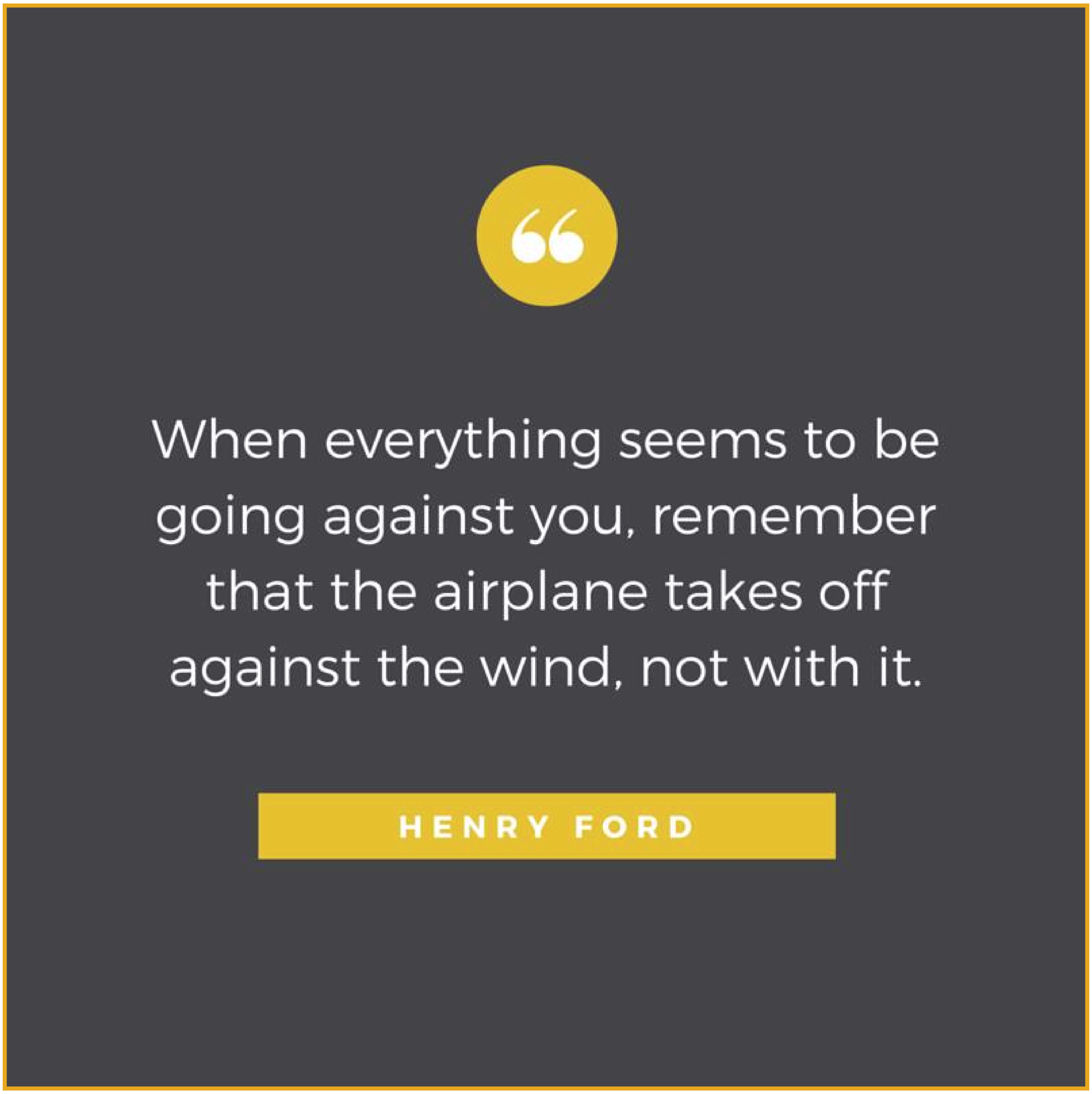
And, I want your application to be a resounding success.

Sincerely,

[illegible]

Frank Zinghini
CEO
Applied Visions

Contents



When everything seems to be going against you, remember that the airplane takes off against the wind, not with it.

HENRY FORD

First, a few management truths.

1. **You will never be finished. If you're lucky.** If you succeed, people will want to keep buying and upgrading your application. They will come up with new ideas and new requirements. To keep them from abandoning your application, and to keep competitors at bay, you'll need to keep meeting those requirements. Plus, technology changes constantly. The only way to keep selling your product is to stay in continual upgrade/update mode.

2. **You will be faced with technical decisions that could have a serious impact on your application.** If you are a software expert, great. You'll be able to help your team make wise technology decisions. But if not, you will need to follow a few basic rules.

    a. **Don't take for granted that your programmers know best.** Developers often go with what they know, because it's safer to stay within the bounds of their own experience and preferences. But sometimes they may choose tools because they're "cool" and will look good on their resumes. "Coolness" and "longevity" are often in opposition. If you're creating a small, short-life application, no big deal. But if your goal is to create a robust and lasting application, you'll want a development environment that is proven, and will be around for at least several years.

    b. **Engage someone who can help you make these decisions, and who doesn't have a vested interest in the outcome.** Programmers can make a very convincing argument for a particular technology, while conveniently omitting that another just-as-viable (or even better) technology exists. It helps to have a trusted advisor weigh in on the choices. Because your programmers will have to buy into the decision, the person should also be someone whom they respect, and who can convince them that what's good for the business is also good for them.

    c. **Don't take this lightly.** I can't tell you how many times a manager has come to us after an application has been fully developed and out in the marketplace for some time, only to learn that it isn't able to handle the demands that success has placed on it. Or that the underlying language isn't supported anymore, so the entire application needs to be redeveloped. Or they just can't find people who know that language, so they can't get anything done. Every decision you make has ripple effects downstream. Don't take any of this lightly.

Who are your customers, and what do they want from your product?

If they don’t like it, your product won’t sell. Everything you do to create your product will be wasted if you don’t start first with your customer.

Too many companies skip this part; they “think” they know what the customer wants, often because they decided to create the application to solve a problem that they had themselves. They build their own desires and opinions into the product, only to find out that customers think differently about all sorts of things relating to that product.

No matter how smart you are, or how much you think you know about your customers, when you take your idea or a beta version out to your customers, they will always surprise you. What you learn from them at that stage will not only help you serve them better, but will open your eyes to other opportunities for your product.

You probably already think you know your customers. Maybe they’re already using your other products, and you know the problems they’re trying to solve with this new software product. So far, this all makes sense.

But without real data, you will be disappointed when you try to sell it to them. What you thought was important to them—so important that it would be the reason they would buy your product—wasn’t that important to them after all. But other things are, things that would have caused you to design the product differently.

Don’t guess. It’s very expensive.

Ask.

Notes from the field: Don’t forget their bosses

We built a product for visualizing information security data, for use by analysts who oversee the security of large networks. We put a lot of effort into sophisticated visualizations that let analysts interact with the data to see what’s going on, and to find and close security holes.

When we beta tested it at the security operations center for a very large, important network, we found that users would work with the tool for a while, then taper off and go back to their old tools.

When we asked them why, we learned that while they really liked the visualizations, and were using them effectively, they were frustrated because they needed to report status to their superiors every day and couldn’t figure out how to get the visual data into their daily briefings without all kinds of extra work.

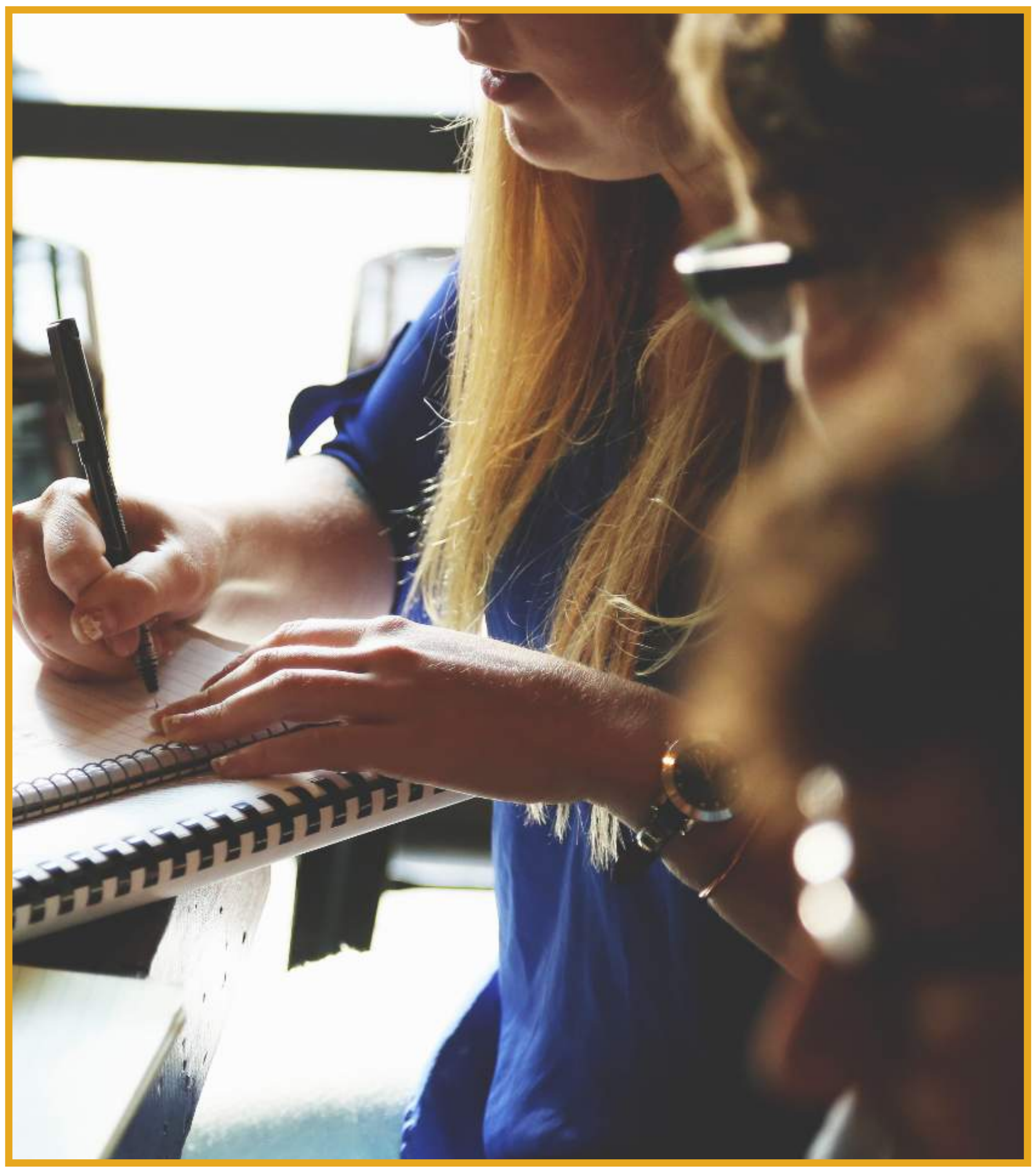
We went back and added a feature that automatically creates a PowerPoint file from the visualizations, using a template they create just once.

They loved it.

Now we are always sure to ask users questions about what they need to do with the data.

LEARN TO SPEAK FROM THE CUSTOMER'S VIEW
RANJAN ACHARYA

What do you need to know about your customers before you can sell them your application?



1. What the customer is doing now to solve the problem, and how they feel about it. If they're ambivalent, because their current method (including "doing nothing") is good enough, it will be harder to sell your solution.

2. What else the customer has looked at, and how they felt about the solutions they found.

3. What competitors have promised them, and what they believed or didn't believe about what they were told.

4. What they might pay for your product, and how they want to pay. Yes, you can ask about this. You need to know if they insist on up-front licensing, or would be open to a subscription model.

5. How they prioritize the features and functions. Which of them are essential, which ones are "nice to have," which ones are "standard" and "assumed," and which ones are not really necessary.

6. Which of those "must have" features are things they really use, and which ones are just "checklist" items. You'll be surprised how often those "must have" features never get used. We separate features into two categories: "buying criteria" and "using criteria."

7. How they feel about these types of applications in general—in terms of larger trends. Do they still feel that this type of application is critical, or is interest in this waning? Are people finding other ways to solve the problem, or is the problem less critical than it used to be?

8. What other applications—similar to yours—excited them, and why.

9. The things that would make your customer say, "I have to have that," or, "I'll never buy that."

How can you make sure they'll like your software?

As you develop your application, countless decisions will need to be made. Those decisions will determine if your product is a success out in the marketplace. That success will be driven more by the usability of the product than by any other factor.

Today's customers have become quite accustomed to trying out an application before they commit to paying for it. In just a few seconds, they will decide if it will do what they hope it will do, and if the application is pleasing to use. They won't have the patience to read a manual or let you train them. They won't think twice about rejecting it. They will assume that someone else must be doing it right; they just have to keep looking.

Your revenue will suffer if you leave usability decisions to your programmers, or if you let these decisions be made "by accident" as an almost thoughtless part of the development process. Or if you don't give usability the attention and focus it deserves.

You really don't want to develop an application, take it to market, and only *then* find that it is unusable, and that you have to go back to the beginning. Making changes to a program in response to user rejection, after it was supposed to be "finished," is no small undertaking. It is also disheartening, embarrassing, and expensive.

Here's how to make sure they'll like your application:

1. **Put a key stakeholder in charge of usability.** Someone who knows the customer, and who represents the customer's interests. That person could be you, the team leader, or a product manager. It should be someone who is a part of all the decisions being made, who knows how the customer thinks, and who can represent the customer to the team.

2. **Involve a usability expert in the development process.** This can be outsourced, if your budget doesn't allow hiring someone full-time, but make sure it's someone who really understands human factors. Adding usability expertise to the team will ensure that your application makes sense to your users, and makes them happy.

3. **Talk to actual users.** Find out who they are, the problems they're trying to solve, what they want get from your application, what else they've tried to do to solve their problems (and how that worked out for them), and how this product would fit into their life or work. Don't depend on surveys for this; the survey itself will be biased toward your own assumptions (which are always slightly-to-terribly wrong), and users often tell you what they want, which is usually different than what they need. Interview at least seven customers of any given type; this is a sufficient sample size to be able to trust the data.

4. **Organize your findings into categories.** Then analyze your results, and describe the issues that all of these customers have in common. This will provide a roadmap for development, and will result in customer-pleasing decisions.

5. **Mock up the application.** Use anything from PowerPoint to an interactive prototyping tool such as Invision. Take it back to those same customers and get their feedback. Record their comments, organize the findings, then analyze and summarize the results. Again, this will provide a roadmap for development that will result in customer-pleasing decisions.

6. **Once the application is up and running, get it into your users' hands early and often.** Don't wait for a "beta release" (unless you want to be like Google and just call everything "beta," which is fine, too). Start with a small set of users and grow this group as your application matures. Repeat the process of soliciting and analyzing feedback.

7. **After the product is released, keep gathering user suggestions.** Make it easy for users to provide feedback. Keep track of suggestions, build those changes into your roadmap, and make sure someone gets back to the them (and all other users) when new releases come out with the changes. Make your revision history visible to the public. Future revenue depends on how you respond to suggestions, because buyers considering your application will research reviews and discussion groups to determine how well you respond to customer feedback.

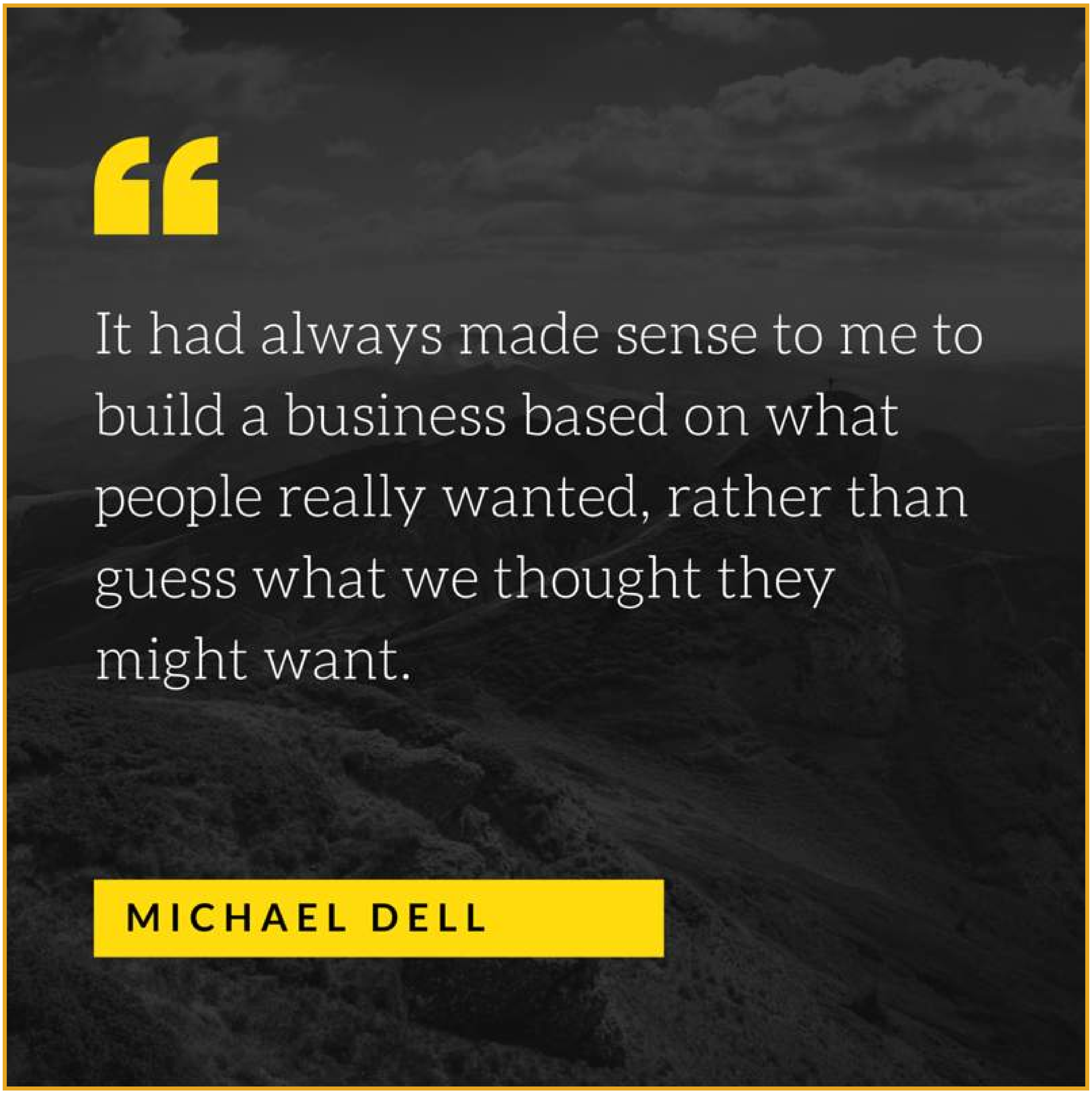
If you're thinking this sounds like a lot of work, just remember how much harder (and more expensive) it is to make these changes *after* you've released your product and have gotten bad reviews or angry support calls.

Also, if you're worried about bothering your customers with all of this up-front work, don't be. In our experience, people like being heard. They get excited when a feature comes out that they see as a response to their input. They gain a sense of "ownership" of that feature, and they will tell others how responsive you have been.

It's really not hard to find people willing to help with this process. Chances are you've got a group of "friendly customers" that you can tap into—people who already like your products and your company, and who would be thrilled to help shape your next product. If you're new to a market, there are all sorts of resources to help—just Google "find beta testers." You'll be amazed.

Are you considering all of your customers? We've been using the words "customer" and "user" somewhat interchangeably, but in reality they are not always the same. The person who makes the buying decision is your customer, whether or not they ever use the application. And in many situations they are not the ones to use it. We have created complex, B2B applications that are used by one group in the enterprise (the marketing department, for example) but acquired by a different group (the IT department).

This certainly complicates sales and marketing. It also complicates development, because you have to satisfy different stakeholders with a different set of requirements, desires, and "gotchas."



“It had always made sense to me to build a business based on what people really wanted, rather than guess what we thought they might want.

MICHAEL DELL

What are the commercial considerations?

Why are you doing this in the first place?

Sometimes developing an application just sneaks up on you. Your customers need you to solve a particular problem, and you find yourself creating a solution for them that involves software.

You give that to your customers, thinking "that's done," and the next thing you know they want changes, or they have ideas for other things that the solution can do for them.

As you dutifully satisfy those requirements to keep your customers happy, it dawns on you that you're creating something that delivers real value, and maybe it can become a product that you might be able to sell to other like-minded customers. At this point, it's time to shift from "accidental" to "intentional" product development.

On the other hand, you may have indeed set out to develop your application as a product from the start.

In either case, you're now intending to take an application to market, and it's time to ask—and answer—these questions.

What kind of application is it?

There are always structural decisions to be made that should be driven by value to the customer, but often are driven instead by developer prejudice. You'll hear from your dev team about the cloud, browsers, mobile web and mobile native, tablet and handheld, and desktop. The overarching concern should be how your customers will interact with your product to get value from it. Here are some general observations that may help you in your specific situation, starting with what the choices are.

Desktop There are times when an application demands the full attention of the machine it's running on. These applications are generally compute-intensive, highly interactive programs that perform specialized functions or interact with specialized hardware, such as computer-aided design, music or video editing, or machine controllers. These are applications that you still "install" onto a specific computer—your workstation or your laptop—that take full advantage of the capabilities of that machine.

Browser Nowadays your thought process should be, “Is there any reason why my application should *not* be browser-based?” There are two main reasons for choosing a browser interface. There’s no need to install anything on the user’s computer, so distribution and maintenance of the application is greatly simplified, which is really nice. But more importantly, if all that’s needed to run your application is a browser, then it can be used on any computer, from anywhere in the world. Welcome to the cloud.

Mobile Tablets, phablets, smartphones, and watches. The world of mobile devices is growing, but they share common issues. Your choice of which devices to support depends on how your user wants to interact with your product. If it doesn’t make sense to squeeze your user experience down to the size of a phone, then tablet-only it is. Can you specify a particular device, or can your customers choose their own?

So what will it be? Before you even consider building a desktop application, it’s wise to ask yourself: Why not cloud? Why not mobile? As of May 2015, Google confirmed that there are more searches being conducted via mobile devices than desktop computers. Global PC sales peaked in 2011 and have been falling ever since. It’s an increasingly mobile world.

Even if your product is obviously not a mobile product, don’t assume that your users don’t still want a mobile component. There is a growing expectation that all apps should span all of the devices in the user’s life (“Why can’t I view the results of my data analysis on my iPad as I ride the train home at night?”).

Mobile: native vs. web

The mobile web experience can feel, well, very “webby.” If you want your users to have a smooth, pleasing user experience on a mobile device, then you should seriously consider native apps. If your product involves using the camera or other hardware features of the device, then you are definitely going native.

You also have to consider an important characteristic of the mobile experience: you can’t rely on a steady internet connection (“can you hear me now?”). If you want your customers to keep using your product as the train enters the tunnel, then plan on a native app, and design it to do things such as cache data locally so it can handle sporadic connectivity.

There are indeed times when a mobile web interface is “good enough,” and it’s even possible to realize your mobile strategy “for free” by building a web interface that automatically adjusts itself to the size of the device it’s displayed on (known as “responsive design”). But do not underestimate your customer’s desire for a smooth user experience. If they get tired of watching the little spinner while waiting for your app to load, they will find a competitor with a native app. Guaranteed.

Speaking of devices, there's a whole world of hurt around deciding which devices to support. Android, iOS, or Windows Mobile? All of them? How many different flavors of Android (there are many)? Do you build separate apps in each platform's own language, or do you pursue a single-codebase, cross-platform strategy? The impact of this decision on the development, testing, and support effort is profound.

As always, before you make a final decision on the right platform(s) for your application, study your market, and interview some potential customers. Understand their needs for mobility. Find out if there are any barriers in the customer's mind that you may not have anticipated, such as privacy or security concerns.

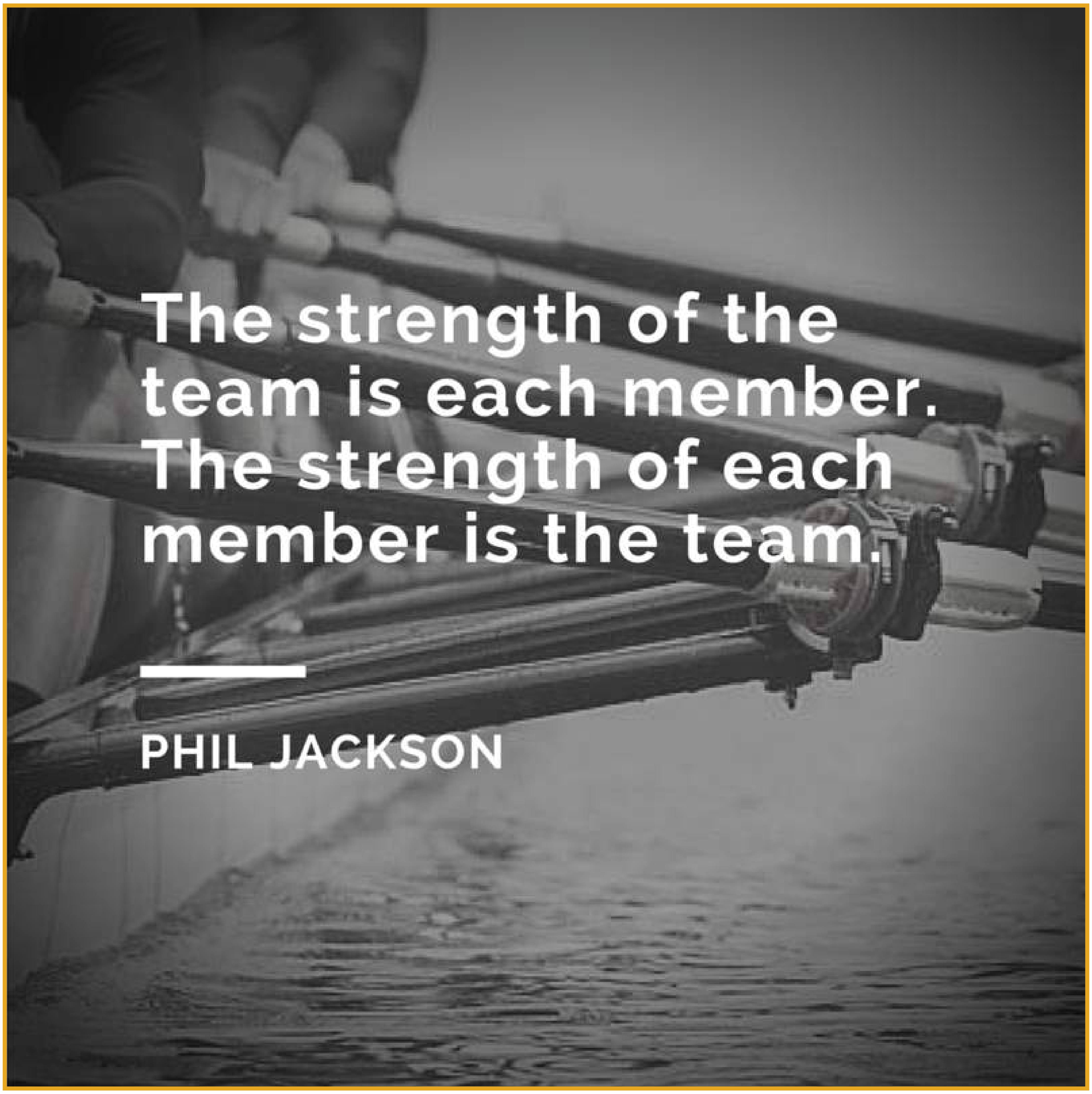
What's your revenue model?

Pricing is one of the most challenging aspects of developing a commercial application. Not just how much you charge, but how you charge for it. Should you charge a one-time license fee, or a monthly subscription? If it's a multi-user application, should the subscription be based on the number of users, number of projects, or some other metric (number of reports, volume of data)? Ask your customers what makes sense to them, what price they would consider fair, what other applications have they used that are similar to yours, and how they were priced.

Of course you must also do competitive research to see the price of comparable applications.

There's a growing practice of monetizing applications by offering a free version in which ads are displayed. There are even advertising services that will source the ads for you, so you don't have to worry about selling ad space. But the more important question is, will users even be interested in the app if there are ads in it? Or will they resent you for it?

Pricing software products is a complex undertaking, and outside the scope of this guide. Just remember what I said earlier: if you're lucky, you will never be done, so make sure that your pricing can support an ongoing development effort.


The strength of the team is each member. The strength of each member is the team.
PHIL JACKSON

What kind of team will you need?

So you've worked out what you need to do, and now it's time to build the team to do it.

What you do now will have a profound effect on the success of your development. For the sake of this discussion, I'll focus on the composition of the team, and not on recruiting specific people; that's a whole different topic.

First, let's start with that word "team." One person is not a team, so don't think that all you need to do is "find a really good programmer" to build your application. No one can develop an application well in isolation.

Sure, if you need a very small, targeted app, you can probably do it with one programmer (although you'll still have to work out usability; your nose-to-the-code programmers are not usually good at that).

Any serious application development effort requires a multidisciplinary team. How big a team, and how many disciplines, is determined by the scale and nature of the project.

There are some basic team composition ideas that we can work with. In talking about this it helps to avoid actual job titles; those are always loaded with preconceptions and emotions that get in the way of rational thought. Save the titles for when you do the recruiting.

It's difficult to talk about the team without talking about the process, but that's also a larger discussion and best left to other guides. All you need to know is that the process best suited to customer-facing applications is called "Agile" (in particular, a flavor of it called "Agile Scrum," perhaps so named because developing applications is as much fun as having two dozen rugby players fall on you).

Agile and Scrum

The essence of Agile is that a smallish team of talented developers, working closely with users (or their stand-in), can execute a series of small (two- to three-week) development cycles (called "sprints"). The goal is for each sprint to end in a demonstration of something meaningful, adjusting assumptions and requirements along the way based on user feedback, and eventually converging to a good final result. (Sadly, as often happens, time and commerce have morphed this simple philosophy into a sort of dogma, replete with high priests [consultants], scripture [books and blogs], and speaking in tongues [jargon]. Don't let that scare you off.)

This team must be close-knit, because they will be working very hard together to keep up the pace of those sprints. They should meet daily but briefly (Scrum calls for 15-minute "stand-up" meetings, because sitting down leads to settling in, which wastes time). Being in the same room, at least part of the time, helps a lot, as does being in the same time zone.

Don't be tempted to put yourself on this team. You may think you know your customer better than anyone, or perhaps you've done development and think you can lead this effort. But you're probably involved in so many other things in your business that you will have a hard time being a productive participant in this effort. There is no "U" in Team. The best teams include:

User Representative This person represents the needs and wants of your user. They know how users think, what they care about, and what frustrates them. They "negotiate" with the development team, helping to set priorities based on their understanding of what's important to the user/customer. Ideally this person should come from the "business" side—such as a product manager or business analyst.

Designer As we've discussed, your team needs this on some level, depending upon the nature of the application. It might be a single user-experience designer, or you may need a human factors expert if things are complicated. But be careful. There's a difference between "design" and "art." There are "user interface designers" who are able to create beautiful-looking screens, but who do not understand the usability issues that inform proper design.

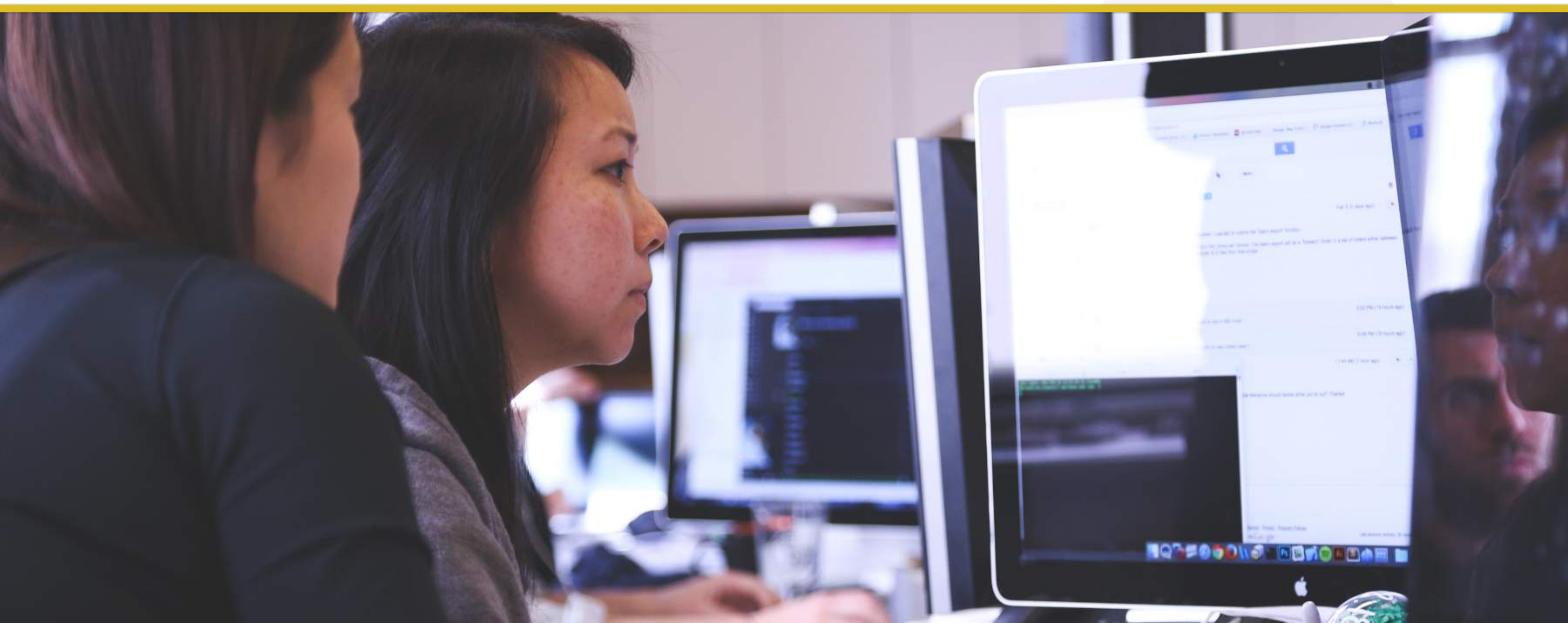
Leader You may hear from Agile followers that there are no leaders anymore, in the traditional sense. Everyone is supposed to work together equally (and interchangeably) toward a common goal. But in development, as in most walks of life, every group of people united by some mission longs for a leader. Someone to make decisions, to inspire, and to motivate. Just remember that there is a difference between leading and managing; developers like to be led, but they do not appreciate being managed.

The leader is most often a developer, because that's where leadership is most often needed: choosing programming languages, debating the relative merits of different frameworks, and so on. You'll want an alpha male or female, who is respected and can lead the team with grace and wisdom. But the leader must also have a strong feel for the whole picture, including business goals; the customer's perspective and desires; a realistic sense of what the team can accomplish; and an appreciation for how similar applications have been developed by competitors.

Developers These are your builders. They crank out code all day, and are happy to do it. There are subtle differences between developers, programmers, and coders, but overall they are the ones cutting the stones and building your cathedral. They come in many different forms, and while we're not going to go into the details of recruiting, it helps to at least understand the three main types of developers.

> **There are the plodders.** They've learned one tool or language (maybe two) and they are inclined to stick with that tool going forward. Yes, they can learn new languages and skills, but mostly they will only do that if it's a job requirement. They do yeoman's work. They can get bogged down and often fail to ask for help, because they often don't interact successfully with other human beings.
>
> **There are rock stars;** brilliant developers who can tear through code, often to the admiration and envy of their peers. Avoid them. Their resumes will be incredibly impressive, but when you put them to work, you won't get what you need out of them. You will get what *they* think you need. That said, if you're up against an impossible deadline and you have solid design for usability and a strong leader to guide things, rock stars can be useful. Especially for challenging backend code that doesn't face the user.
>
> **In between are the innovators.** They are not afraid to try new things—a very important trait in a successful developer. While they are open to experimentation, they also have a sense of appropriateness; they do not make arbitrary choices. Innovators recognize and deal with roadblocks. ("Wait. This isn't working. We need to recognize this and discuss alternatives.") The innovator will look for another way to solve the problem, working with, and at times guiding, the rest of the team. Most leaders are innovators; when you find one, don't let go.

Testers A tenet of Agile is that every sprint produces a working release that can be shown to users, so your team needs testers. Many organizations have a "quality assurance department" that waits for the developers to say it's all done before they go in and test things. In Agile, it's different; the testers are an integral part of the team. At the start of the sprint, they listen to what the team says they will be building, and they figure out how they will test it. They share that with the team—the "acceptance criteria" of the work—so the developers know what it means to be "done." They test early and often, and they communicate constantly with the developers.



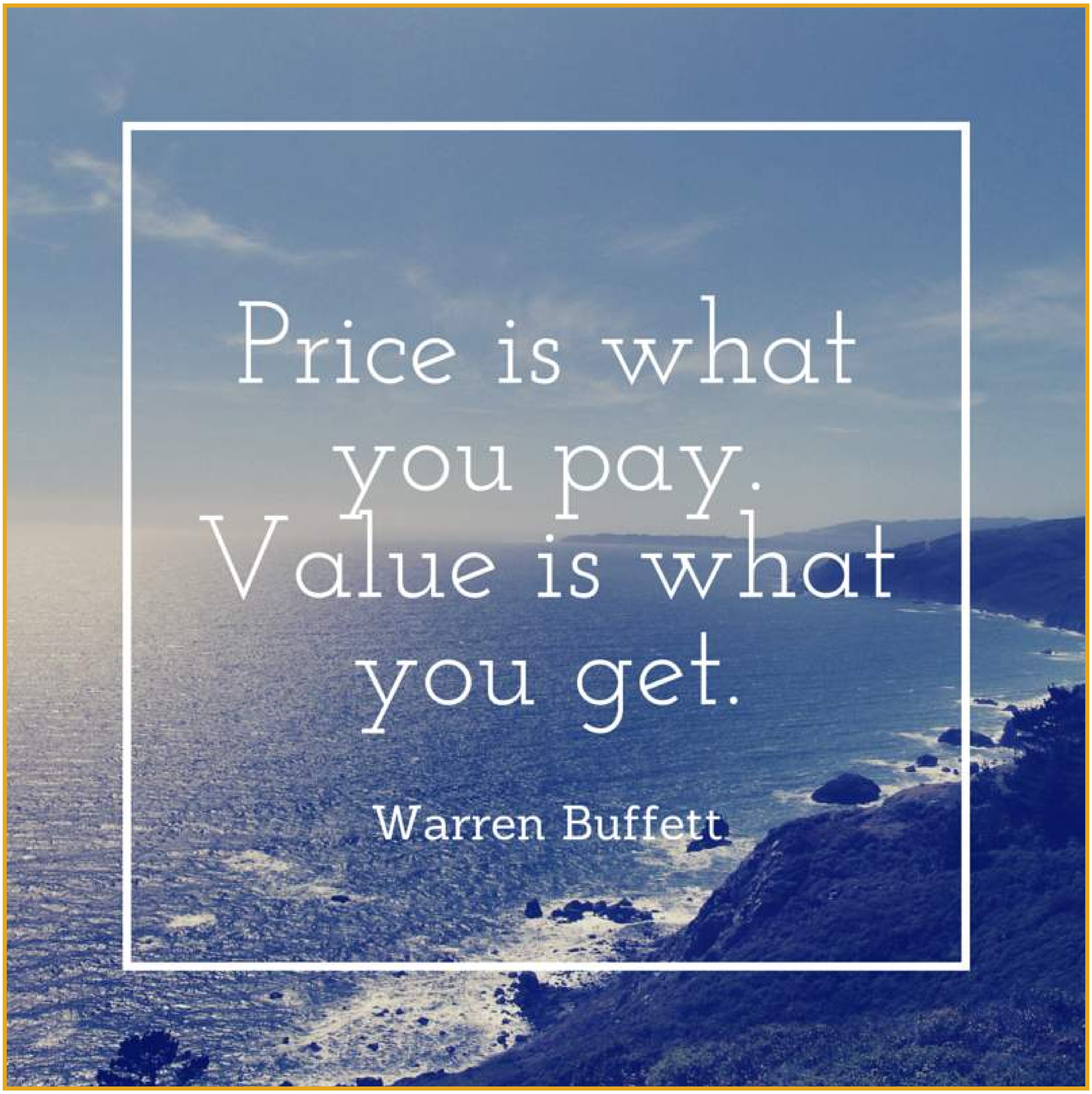
Playing well with others

We touched briefly on how some people are better fit for a team than others, but it bears a closer look. You can spend a lot of time studying team dynamics; you probably should (and you probably have). But for now let's consider the qualities that you should look for in everyone you add to your development team:

Communication. Good teamwork requires clear, unambiguous communication. That includes speaking and writing. Email, messaging, and issue-tracking systems are the most common channels, but team meetings are also important. Phones are often involved. So the ability to communicate verbally is a critical skill, and is particularly important for those who need to communicate outside the team with users, customers, or the company's stakeholders.

Empathy. Something often lacking in developers, and especially in rock star developers, is an appreciation for other people's issues, concerns, and feelings. That's been an issue since the first cavemen got together to discuss making a fire, but it is magnified by the fast pace, unpredictability, and high pressure of application development.

Collaboration and humility. This is the willingness to leave your ego home, accept criticism, be open to change, and forego your own needs for the good of the team. This is what makes a "team player."


Price is what you pay. Value is what you get.
Warren Buffett

How much will it cost?

How much ya got?

Seriously, and you're going to hate this, development cost is more often a question of what you are willing to spend than what it will actually cost. That's because the cost of doing everything that you actually want is always way more than what you are willing to spend. So start thinking in terms of "what can I get for what have?"

You can get a rough idea of the overall cost. First work out all the numbers in a wee-hours spreadsheet. Make your most educated guesses about the people needed, technical tools, outside vendors, and business services; all of it. Show it to others, get their input. Make adjustments. Work on it until you think you have it all as buttoned-up as it can be.

Then triple it. In my experience, this will be closer to your real cost.

Why triple? On top of the actual costs of the development work, the additional costs come from:

One part "oops." There are going to be things that go wrong. You may select a particular technology as the basic development environment, and then halfway through the development cycle, you realize that it can't handle some important aspect of your application.

This happens to almost everyone. You're lucky if it happens to you *before* you take the product to market; most people find out after it's been out there for a little while and the technology just can't keep up with demand, or customers reject the application because of a limitation imposed by the technology. In our "try before you buy" software economy, customers find out very quickly if they want to pay for the app they've just downloaded or obtained as a trial.

One part "what?!" Unanticipated costs can come from any quarter. The lead engineer you hired, who was a great fit, might be recruited away and the only viable candidates are 25% more expensive. Plus it will take time for the new lead to come up to speed. The integration between your application and another could turn out to be far more difficult than you first anticipated, adding months to your development cycle. You can't know these things before you get started; they'll pop up as you go.

Now that you've got some kind of budget in mind, there's one last thing you need to know. You will never get everything you want for that. Software development just doesn't work that way.

Yours will *not* be the first software project in history to come in exactly on time and on budget. Getting to the end of a development project is an exercise in compromise; the strength of Agile is it recognizes that. The key to being "Agile" is recognizing that you may have to give in on one thing to gain something else.

The Golden Triangle

It's a bit of a cliche, but it still applies: "Better, Cheaper, Faster: Pick any two." It's every bit as true in software development as it is in the pizzeria or other local business where you last saw that old sign.

The best way to deal with the reality that something's got to give is to start with the tightest constraint.

- If budget is the most critical issue to you, adjust the time and feature expectations accordingly.
- If you must have certain features or the product won't sell, make that your top priority and be willing to put in the time and money necessary to achieve that end.
- If you're pressed for time, for whatever reason, you'll have to pay extra (for more and better people) or cut some of the features to get the product out there, and be ready to make adjustments to the product minute it goes out into the market. (And pray that you can move quickly enough to make those adjustments before a bad reputation kills your market chances altogether.)

Cutting corners

If cost is a driver, one thing you may consider is whether to have the application developed offshore. We all know that the labor costs offshore are much lower than they are in the U.S., and there are plenty of talented programmers in other countries. Given that we are based in the States I am obviously biased, but what I can tell you is that we are often asked to re-develop an application that was originally taken offshore.

This is not a reflection on the technical skills of the offshore team, but rather on the appropriateness of the team to the particular application being developed. We're speaking here of applications that touch your customers, build your reputation, and generate your revenue. The teamwork and communication discussed here are critical to that, and are not well-served by remote teams in other countries.

When will it be finished?

Remember what we said earlier: a software application of any value is *never* finished. If you're lucky. So what you're really asking is, "When can I ship something?" That's an important distinction, because it's a reminder that, to some extent, it's under your control.

Most applications have some minimal set of capabilities that must be completed before they can be at all useful (you'll hear developers speak of "minimal viable product," or "MVP" because giving things acronyms makes them sound better). But beyond that, you have choices. Just as you can dial back features to save money, you can do the same to get the thing out the door.

Be wary of development by contract: if you promise features to your customers before knowing what can be done by when, you may be headed for disappointment. Development is a matter of trust and compromise, not contract.

Different stakeholders will argue (passionately, if they are good at their jobs) for particular features. As the manager in charge, it may fall on you to decide what is best for the business. I suggested earlier that you stay out of the day-to-day work of the team, even if you think you know the customer better than anyone. That's true. But it is here, at these significant decision points, that your understanding of the customer, the market, and the business will come to bear on the success of the project. These are the decisions that need your input and direction.

How's it going?

Now that you've accepted your role in determining when it can ship, you'll want to know how to tell if the team is making progress towards that goal.

You rely on your team to report progress, of course, but if you've ever been the recipient of "progress reports" that detail "percent complete" as they color in the milestones on a Gantt chart, you've probably been frustrated. It's fairly typical for developers to be "85% finished" within a week or two, and remain at that reported 85% for another month.

Here again, Agile helps. By agreeing to build your product in short sprints, each of which produces something that can be demonstrated, the team is making its progress apparent. You'll have the list of everything that the team agreed should be in the application (they call them "stories"), and you can see that list being worked off in two- to three-week increments.

Of course there will be status reports (you'll see "burn-down charts" and hear of "story velocity"), but just seeing those demonstrations every few weeks will help you get a feeling for how it's going.

Your role in all this is to listen to the team, respond when they need your help, and watch to make sure they keep their eye on the big picture.

The requests for help will take many forms. You may be called on to resolve those feature conflicts between the developers and the user representatives ("they say they want X but that will take a month; if we do Y instead we can do it in a week"). Or you may be asked for resources (tools, people, money). Because you have constant visibility into their progress, those requests will make sense to you.

The downside of maintaining a focus on the needs of the user is that sometimes the "backend" things, such as integration with external systems, application security, or keeping audit trails, can be overlooked. Don't let your developers be dismissive about these issues, which are always more complicated in reality than they are in theory.

"Saving that for the end" is not a good strategy. Integration complications are one of the main reasons that application development takes longer than anticipated. These factors should be factored into the development strategy right from the start.

I hope you have found this guide helpful in some way. I'm always happy to talk about this process, whether you are on the edge of the pool about to jump in, or already deep in the middle of it. Feel free to email me at fzinghini@avi.com.

Frank Zinghini is founder and CEO of Applied Visions, Inc., which creates custom software applications for the cloud, mobile, desktops, and the Internet of Things (IoT). The company specializes in building applications that meet revenue and usability goals as much as they meet technical requirements, and has earned a reputation for applications that are visually exceptional and exceptionally usable. Since its founding in 1987, Applied Visions has spun off two other businesses: Secure Decisions, which brings innovative visualization to national security applications; and Code Dx, which sells a software vulnerability management product that helps developers create secure applications.

6 Bayview Avenue
Northport, NY 11768

631-759-3901 - fzinghini@avi.com - www.avi.com