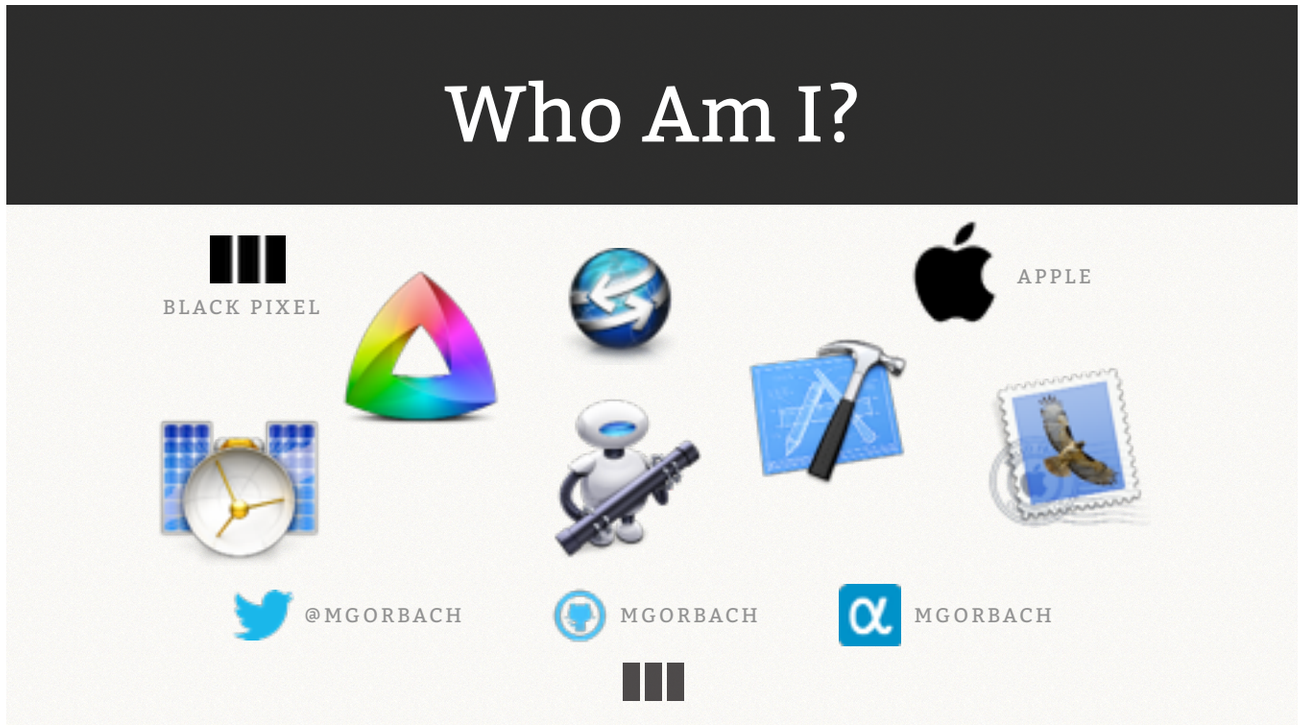
The Last 10%

Spit, Polish, and Why You Should Care
Or: A Short and Very Partial Guide to “Surprise and Delight”

CocoaConf 2013 Michael Gorbach

Good morning, everyone. I hope you’re enjoying the conference so far and thank you for coming out today.
I want to talk today about the last ten percent of the apps we build. What do I mean by that? The weasel word we use for this quality is “polish.” Apple has a phrase for this, too: “Surprise and Delight.” I know that’s vague. Rest assured, much of the point of this talk will be to make that concept a lot more concrete.

Let me start out by saying this, though: every product is different, and with that, the definition of “polish” changes, too. I don’t know what your product does, and so I can’t provide much help on the details of exactly how to make, for example, the best possible Underwater Basketweaving tracker. However, the pieces of software each of us in this room are building do have a tremendous amount in common. They live on Apple’s platforms, they interact with users with the mouse, the keyboard, their eyes, and their fingers. They interact with the external world, and, yes, they have bugs. It’s the polishing of these things, which we all have in common, that I want to talk about today. It is about getting many tiny little things right. It is hard work, but it can be rewarding.



Before I get into the talk, it's probably a good idea to introduce myself. I've been working on Mac apps since the Intel transition in 2007. My first app was a little tool called Macfusion, which let you mount web servers as volumes in your Finder. I wrote that little thing and open sourced it. The next day, I had real positive letters from my users, sitting right in front of me in my Inbox. I've been hooked ever since.

I've been very, very lucky to have worked at a few truly awesome companies, including Apple, and, more recently, Black Pixel. I've worked on Automator, Mail, and Xcode at Apple, and NetNewsWire and Kaleidoscope at Black Pixel.

One of perks of having spent time at companies like this is that I've worked with some really incredible visual and UX designers. I've learned a lot from them, but I've also learned very clearly that I am **not** them. Despite those limitations, I've been struggling to understand what it is that can make a piece of software on the Mac, or on the phone, rise above being "Just OK." I'm very much still figuring it out, but this talk is an attempt to lay out some of what I've learned.

With that over with, let's move on to today's plan.

The Plan

1. Why Polish?
2. Errors and Robustness
3. Performance and Responsiveness
4. Custom UI

I'm going to start today by talking about the significance of polish, and why you should bother spending time on it, even though it can be a real pain in the ass. After that, my goal is to talk about three broad areas where spending a little bit of additional time can make a big difference for the overall feel and quality of your app. Don't limit yourself to these areas, of course. They're just a small selection. I chose these three areas because they're common enough that they touch almost every app. I've also tried to base this talk around some concrete examples, so that hopefully everyone here gets at least a few specific things out of this that they can take and apply in their own work.

First, I'm going to talk about errors and robustness. I know this seems like the "eat your vegetables" section of the talk. And, yes, anyone that has worked with me will tell you that error handling has become a sort of pet peeve for me. But I hope by the time I'm done talking about this subject I'll be able to convince you that gracefully handling the problems that occur while people use your software matters more than you might think.

Second up is Performance and Responsiveness. In general, the speed at which our code runs is something we have an intuitive understanding of as engineers, but I think this is still a valuable subject to talk about. This is especially the case because user-visible responsiveness isn't the same thing as the performance that we talk about as engineers. I'm going to talk about the difference there, and how it's gotten me into trouble in the past.

Third, Custom UI. There are lots of technical discussions and talks on the subject of building custom interfaces. What I want to talk about today, though, is how custom interface affects the overall quality and feel of your app, in both positive and negative ways.

Polishing your work is difficult. It's probably an exaggeration to say that the last 10% of the app is 90% of the work, but ... maybe not by much. It's difficult, and when you're in the middle of it, it can be hard to see the rewards. I've got a tableView, it's editable, and it shows the user's data, right? Isn't that enough? If it basically works, shouldn't I just ship it and go from there?

Well, I don't think it's quite that simple. I may be naive, but I still really think that when you get the little things right, your customers do notice. Apple's certainly do. Maybe your product costs a little bit more, maybe it's got a smaller checklist of features. But a product that does a better job on the details feels different to use, emotionally speaking. There is a very real emotional reaction to quality. We experience it in physical things all the time, especially Apple's hardware. Remember the first time you held an iPhone 4? Virtual products have it too, although it's a little more subtle.

The most common argument I hear from people about this subject is that it's not worth it and no one will notice. The idea that polish is just for software nerds, or Apple fanboys, or other developers. I couldn't disagree more. Sure, those people will uniquely appreciate your hard work, but they aren't the ones you are sweating for.

Literature makes for a good metaphor here, I think. A great novel isn't just great for authors, or literary critics, or English professors. It just is a great novel. If you're an average reader, maybe you can't deconstruct the pieces of why a passage is evocative, or gut-wrenching, or beautiful. That's something only the writers or professors can do. But that doesn't make the passage any less beautiful to you. The quality that comes from polish is like a great writer's voice. It is expressed through hard work, and a million little decisions, but it is pervasive. Everyone who spends time with what you build will notice it, and it will make a difference. To bring it back to Apple's own catchphrase: "What do we want people to feel when they use our products?"

There are other reasons, too. Apple notices polish. When you go the extra mile, we as a community of developers notice it as well. The drive towards paying attention to the finest details is what sets apart our platform from the others. It forms a lot of what we have in common, and it is something we all respect.

http://www.homedsgn.com/2013/01/07/glass-houses-by-santambrogio-milano/glass-houses-05/

People in glass houses ...

Before I go on to try and make this more concrete, I want to talk about examples a bit.

While working on this talk, I kept finding that sometimes it's clearest to focus on what **not** to do, instead of the other way around. That means I'm going to be throwing some stones while I show my examples. My goal here is not to attack the people that wrote this software. These are all mistakes that pretty much every one of us could have made, and usually has made. The goal is, instead, to illustrate how we try to do better.

A number of the upcoming stones will be thrown at Apple itself. I do that for two reasons. First, Apple is a big punching bag. It's harder to hurt a Fortune 500 company's feelings. Second, Apple's software is familiar to all of us, and that gives us a common language to talk about these problems. So if it feels like Apple's being singled out here, that's why. It's not because they necessarily make any more, or any less, mistakes than any of us do.

Finally, I'll draw on some of the software I've worked on for examples as well. That's not say that I'm always doing things the right way. As an old tech lead of mine used to say, "I'm wrong most of the time!" I use these examples only because they are the examples, and the mistakes, that I know best. With that said, let's get into the specifics.

Errors and Robustness

So, I want to talk about errors. Things go wrong in your app all the time. External things, internal things, and um ... problems between the keyboard and the chair, too. I know, I know, things break, but they're corner cases, right? They're exactly the kind of thing that engineers can naturally be tempted to ignore. In fact, this whole part of the talk may seem like it's just not all that important. But before you go and get a coffee, let me try to explain why handling all these cases well, and thoroughly, is much more critical than it seems at first.

Who Cares?

You should. Robust apps build confidence.

- *"For the duration of the error condition, handling it is the app's entire reason to exist. Might as well do so properly."* ~ Nat Irons
- Make someone's crappy day a little bit better.
- Show compassion for your users and they will thank you for it.

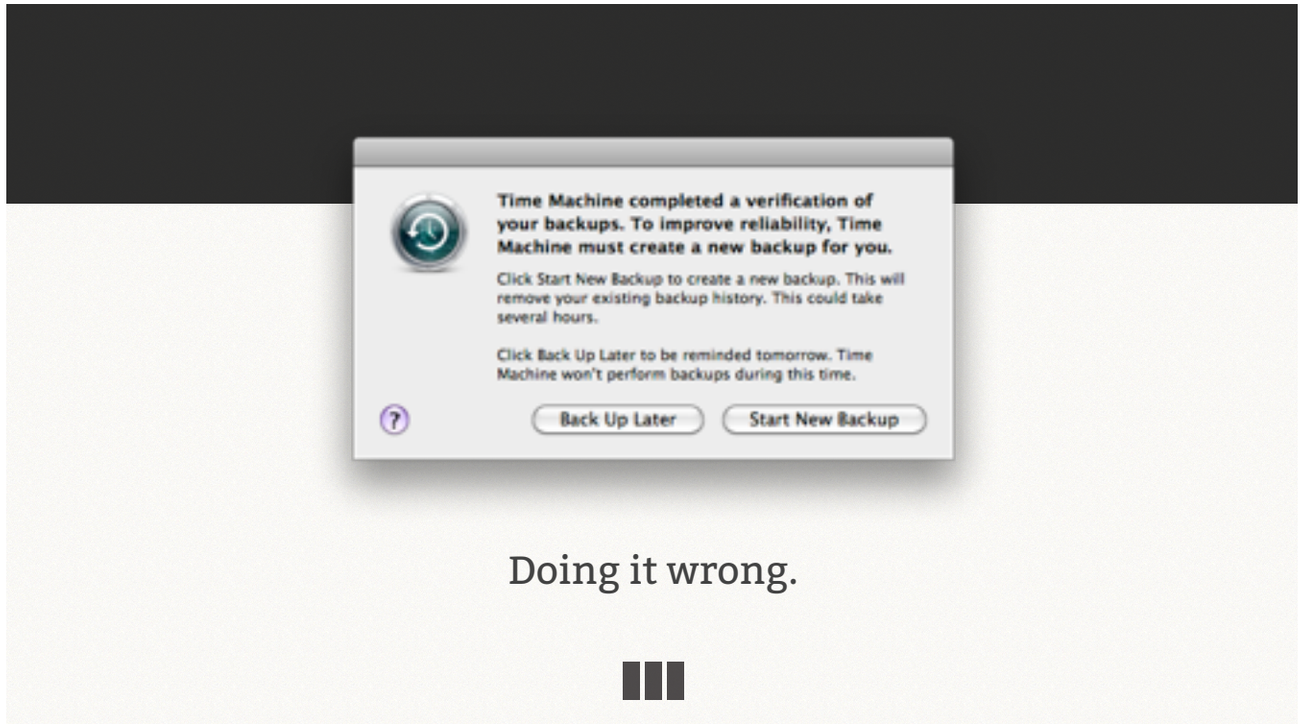
You should care about catching what goes wrong in your app. You should care about communicating it clearly and simply, and helping the user resolve the problem. Here's why.

First, robust apps build confidence. This is especially the case for the more meaty, productivity type of products. I'll give you an example from when I was working on Kaleidoscope. If you don't know the product, it's enough to know that Kaleidoscope is a diff tool, used for comparing source code and other documents, and moving changes back and forth. I found an issue where, depending on circumstances that had to do with network drives, or permissions, or encodings, sometimes Kaleidoscope would fail to save a file that the user had made changes to. We were being sloppy, and some API that was supposed to be writing to disk wasn't. The problem got logged in the Console somewhere, but was never shown to the user. Most of the time, if they hit save again, it would work properly, and it would always save when they tried to quit.

The problem, really, was a lack of thoroughness on our part. It turns out that the Apple API was giving us an error, and we were dropping it on the floor.

What's the harm here? We let our users down, and we lost their trust. Kaleidoscope is a professional tool, and it's so very critical that everyone who uses it can rely on it to do the right thing, or barring that, fail in a clear and consistent way. If this is an app that sometimes doesn't save, what else is it sometimes not doing, or doing wrong? Is it screwing up my source files? If I hit that issue, I might think maybe I don't want to come back to that app any more when I'm in a hurry and need to do a code review. **Maybe I can't rely on it**. That's why we went back and fixed the problem with Kaleidoscope. We caught the error, identified the specific error code, wrote some clear copy for the user, and presented that.

Our amazing QA engineer Nat Irons said something on Twitter I wanted to quote here. "For the duration of the error condition, handling it is the app's entire reason to exist. Might as well do so properly." When things break is an excellent time for us developers to show exactly how good we are at what we do. Even if it's ultimately user error, the fact that you chose to be helpful instead of leaving your users to fend for themselves will mean a lot. This is even more the case when the person behind the keyboard is frustrated and angry because their computer isn't behaving today. Show compassion to your users at that point, and they will thank you for it.



Who has seen this error message before?

I hope you haven't had to see this one ... but if you use Time Machine, you probably have.
I put this up here because this is a pretty common issue, to the point where this particular failure has almost become a sort of community joke. This isn't just a joke, though. I thought it'd be useful to look at exactly what's going wrong here, and maybe try to avoid some of these things in the future.

First, there's a larger robustness problem here. Time Machine is based on technologies like AFP, Sparse Disk Images, and HFS+. Those details don't really matter for this talk, but what does matter is that these technologies, especially when combined in the way that they have been by Apple to create Time Machine, have fundamental issues with robustness. When you add wireless networking into this mix, you've got a recipe for disaster. To me, it seems that the lesson here is this: The best way to handle errors is not to have to handle them at all. Design your software to be robust at its core, if at all possible. Maybe this error is just a case of bad technology choices leaking out to the poor, frustrated user.

Of course, choosing a very robust set of technologies isn't always possible. Sometimes error cases are fundamental and inevitable. That brings me to a second issue with this dialog. You have a prompt, asking you whether you want to **delete all your backup history completely**, or **just ignore the problem and totally stop backing up**. Neither of these are really acceptable options! And the funny thing is, there could be a third option here. If you look around online, you'll find articles talking about how most of the time, it's possible to fix this problem manually with some command line voodoo, and **preserve your old backups entirely**! Rather than working to resolve the problem that has arisen here, Apple is simply giving up and sticking you with the consequences. So there's another lesson here. We shouldn't be lazy. If a problem does occur with our software, no matter what the cause, we need to do everything within our power to fix it. Especially when it has to do with user data, as this issue does. Doing that is our job as developers. Doing that is why we get paid.

There's one final thing I wanted to discuss here. If you read the text, you'll see there are clarity issues with the copy here as well. I can see that, and I'm no copywriter. Do we really need to know, as the first sentence of the title says, that "Time Machine has completed a verification?" Is that relevant information

Be Thorough

- **NSError** is your friend. Consume where available, produce where appropriate.
- Any pieces of your software interacting with the external world are most likely to go wrong.
- Information about what went wrong, where, and how is valuable intelligence.

The simplest way to start doing a good job at dealing with errors is just to be thorough. The Cocoa APIs are full of NSError return values. They are there for a reason, and it's not just to make the method names longer. If there's an **NSError** return value, delegate method, or property, it's because there is a real possibility that whatever Apple API you are calling can fail. Apple's use of **NSError** is a good guideline for us, too. **Every operation in your app that can fail should have some path in the code to notify of that failure.** That doesn't mean the user needs to specifically know about every minor issue, but you probably do.

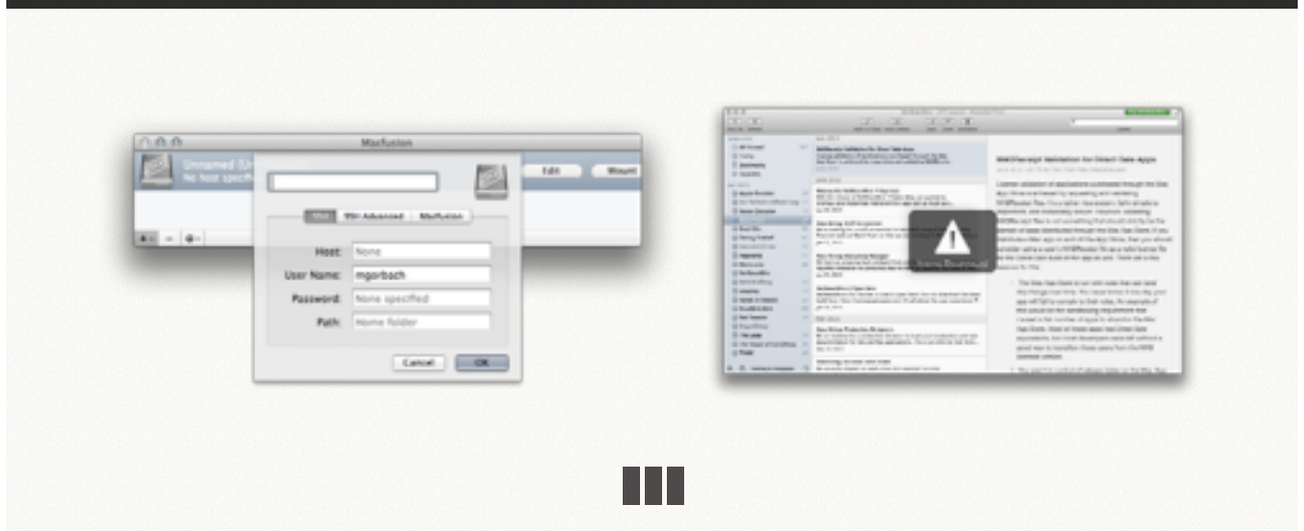
The most important areas to focus on for error handling and robustness are the ones where your app in any way touches the outside world. User input is the first big one. If you're letting the user enter anything at all, you can be sure that someone, somewhere, will enter it wrong. Let them know as clearly and as quickly as possible. Are you using the filesystem? Watch for permissions errors, paths being too long, and even network filesystems disappearing. Are you connecting to the network for anything at all? It can fail in a number of ways. All of them are important to know about, and most of them should be communicated to the user in different ways. The server giving a bad response isn't the same as the server being unavailable, which isn't the same as the user's internet connection being down entirely.

If we practice that sort of thoroughness, it can go a long way towards creating robust apps, and earning the trust of our customers.

There's another benefit, too. Information on what went wrong is a very valuable kind of intelligence. Most us have something like HockeyApp to track crashes in our apps, but that's only the first step. Just because your app didn't crash doesn't mean there wasn't a problem.

At the very least, log that information to the Console where it'll be accessible by you or your support folks if they're needed. If you can, and if the user allows it, communicate that information up to you. It's a goldmine for increasing the quality and robustness of your software. For example, If you're writing an RSS feed reader, don't wait for your app to crash to know that it's having trouble with a specific feed. Don't wait for the customer to write you a complaint email that it's not working, either. Instead, detect XML parser warnings earlier, and, with permission, of course, collect that information. Use it as feedback, so you can do a better job.

Present It Right: Modality



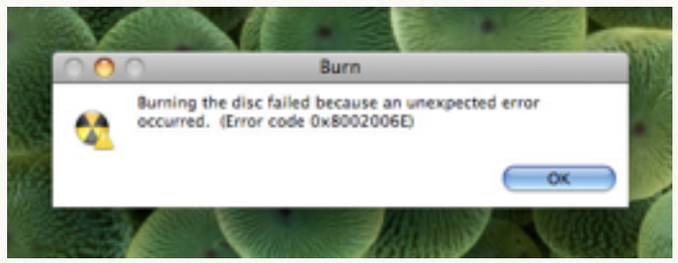
The next thing I want to talk about is error presentation. First off, not everything has be an error dialog or sheet. There are (at least) two other directions you can go, depending on your specific situation.

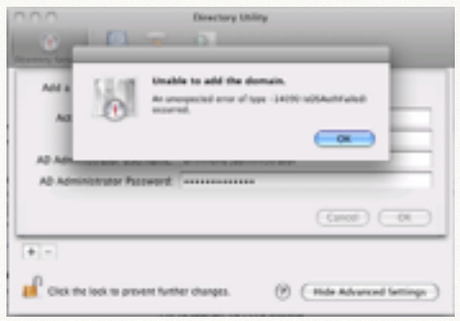
First, it's worth taking inspiration from the way web developers solve these sorts of problems. They have figured out that even if most of the time it's much easier to get some form data, wait until the user clicks the send button, send that data off, validate it, and finally show the resulting feedback or error, that's not necessarily the best experience for the user. Instead, they have in many cases started validating input in real time as it's entered. This makes for a very short feedback cycle.

That reminded me of one example where I realized that I dropped the ball on this, earlier in my career. The app on the left is my first one, called Macfusion. It lets you show the contents of web servers right inside your Finder. As you can see, we've got the typical form for entering some settings for the server. Click the OK button and Macfusion will try to connect, possibly figuring out that you've gotten the hostname wrong, or your username is incorrectly capitalized. It would then put up a sheet and let the user try again. Looking back, what I should have done here was to **validate as much as I could while the user was typing**. If the internet is down, for example, or the hostname is incorrect, I could figure that out immediately, without even checking their login details. The immediate, direct feedback makes a huge difference, and it would dramatically improve the interaction here.

There's one more direction I want to talk about here. Lots of actions in your app will be fundamentally secondary ones. These are the kinds of things that just don't affect the fundamental workflow, like sharing off a document, or sending a link to Instapaper, for example. Here's what happens when Instapaper sharing fails in NetNewsWire 4. Notice the bezel being used to show the problem. The key here is that, because we're dealing with a secondary sort of interaction, there's no need to block the user when something goes wrong. We are letting them know that it didn't succeed, including an explanation of what went wrong, but we're doing it in a way that's quite a bit less disruptive, because, hey, the user's trying to read a news article here.

Present It Right: Error Codes





So, what if you do have to use a regular plain old error dialog? Sometimes that's the only thing, and the right thing to do. If your app has been told to save something, and it can't, you need to stop the presses and let the user know that. It needs to be clear. There are a few guidelines I try to follow when looking at cases like this.

The first one is really simple. **Don't show error codes in your error copy!**

Here are two screenshots from Apple apps. Oh, right, 0x8002006E; obviously. I know exactly what's wrong here. There's Directory Utility, doing the same thing.

Present It Right: Error Codes

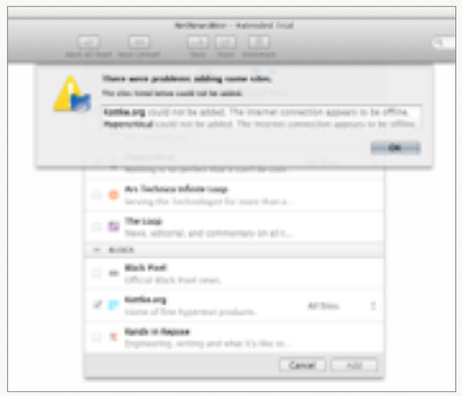
- Don’t show error codes in your copy.
  - (Most of) your users can’t read **MacErrors.h**. You can, though.
- Use **NSError’s** localized strings, or **GetMacOSStatusErrorString** and **GetMacOSStatusCommentString** if you have **OSError** codes.

Unless you are building a very special kind of app, with a very impressive base of users, they can’t look these up and they understand them. They can’t read MacErrors.h. Instead, this is exactly the kind of thing that makes normal people feel that computers are scary, bad, unpredictable things that are not their friends.

Error codes are very useful things. They’re useful because you can interpret and handle them without looking at strings. They’re useful for support, and they’re useful for debugging. By all means, store them, use them and analyze them, but, please, don’t scare people with them.

Instead, explain what went wrong using prose. How do you do that? Well, sometimes you can use Apple’s copy because it’ll already be in the NSError object. If you have an OSError code, Apple gives you functions to get readable strings out of those, too. Otherwise, the only way is to try and cause these issues to happen, understand what errors they return, and write your own prose.

Present It Right: More Tips



- Be specific!
- Use sheets when appropriate.
- Show multiple errors when appropriate.
- Use **- [NSResponder presentError:]** and **- [NSResponder willPresentError:]**

The second guideline is to be specific, as much as possible.

Here's a screenshot of NetNewsWire, showing what happens when you try to add some sites from the popular sites list, but something goes wrong. A few things to note here.

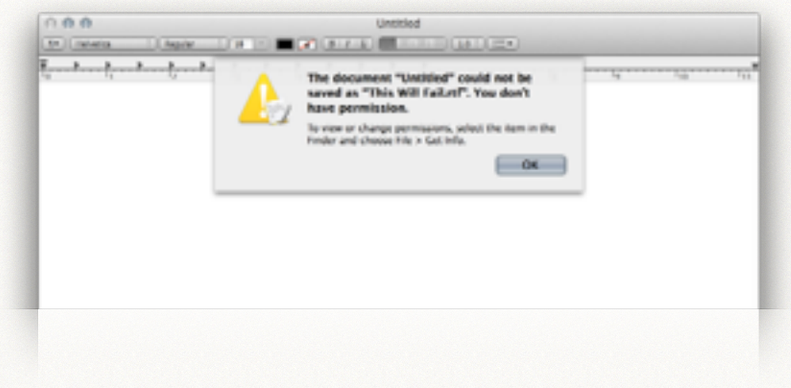
Number one, we're using sheets. Sheets are a bit of a pain in the ass API-wise because they're asynchronous, but I think they are worth it. The great thing about sheets is that they let you tie the problem that has happened much more specifically to where the user just clicked or typed. Using sheets also means that they don't disrupt any other windows or documents opened in your app, so you can have a modal error while being a little bit less of a pain in the ass.

Number two, we're clearly showing what user-visible concept actually had the problem associated with it. In this case, it's a site. We're giving the specific names of the sites that we tried, and failed, to add. This is much clearer than saying that something, somewhere has gone wrong, and is probably but not necessarily associated with what you just did.

Finally, we're showing multiple errors at once, since they've both occurred at the same time. This takes away any confusion about whether some things might have succeeded, but not others. That means the user doesn't need to go back and check what did and didn't go through.

To achieve the last two, Apple has given us a wonderful and flexible API called presentError:. This API is sort of unknown and underutilized, but it's wonderful because it lets you easily delegate presentation of errors using a chain, similar to the responder chain, so that the class that knows the most about the context of the problem can be responsible for displaying it. It makes this stuff really easy to do well. You've even got a method called willPresentError that lets you alter the error objects at each stage of the responder chain. That means each link can fill in the relevant information that it knows about the problem. I'd really recommend taking a look at this API, and getting used to using it. It makes this kind of handling really easy to implement.

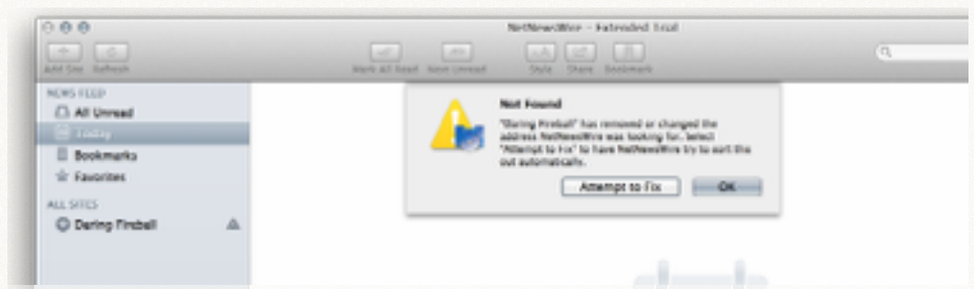
Present It Right: Apple's Copy



One last thing about writing this kind of copy. Sometimes, you don't have to! Apple often does a really, really good job of writing error copy, especially for things related to documents, files, and network activity. Use that to your advantage. It's wonderfully edited copy, and it's localized too, so you can save yourself some work doing that. Notice how they're doing a great job of using a sheet, and showing context about what exactly went wrong. They've even got a great suggestion for resolving the problem all written up for you. Try to use and re-use Apple's copy whenever you can. Of course, it's not always there, and when it's there, it's not always good. Sometimes, you'll end up needing to add more of your own information using the error presentation API I talked about earlier. I've found that a lot of the time I had to go and try these problem cases and see whether they are handled well or not.

OK, I know you're probably really, really tired of talking about errors by now, but before we get done with that, there's one more thing I want to mention.

Don’t Be Lazy



Like I said earlier when talking about the Time Machine error: Don’t be lazy. Sometimes you might find yourself writing an error dialog or sheet for a problem case, but if you spend some time really thinking about it, there’s something you can do to resolve it for the user.

I’ve found two scenarios when this comes up. If the resolution you have in mind is always appropriate to implement, meaning that there’s never a scenario where a user might **not** want you to do it, just make it happen right then. Don’t even bother the user about it. This might be the case, for example, if you detect some corruption in a database, and you know you can reliably fix it. It would be kind of nice if Apple did this and we never had to see that terrible Time Machine dialog from earlier. **If you find this scenario, just fix it.**

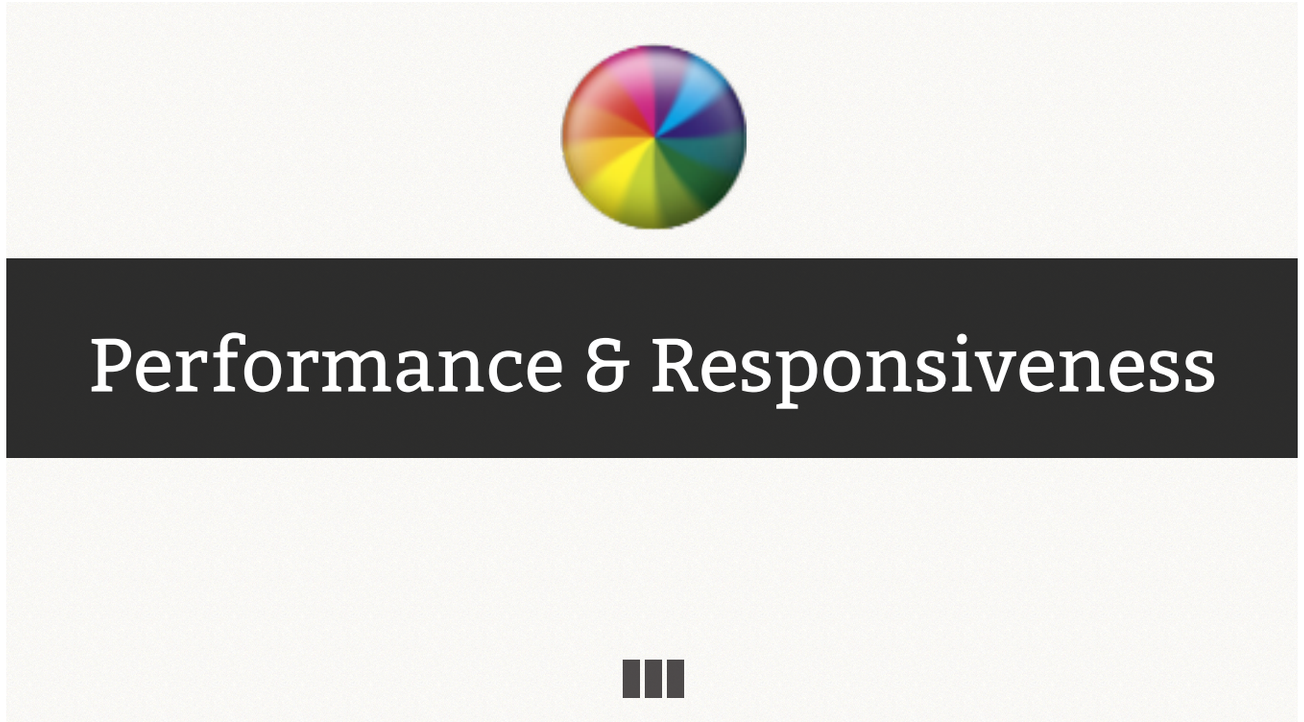
Sometimes, though, you have a good idea about how to fix the problem, but it’s not always obvious that you should do it without asking. Here’s an example of that, from NetNewsWire. Lets say the user added a site, and it used to work. Since then though, the site has moved its feed. The domain still exists, but we’re getting a 404 error when trying to refresh the feed. We could be lazy and just pass the problem on to the user. Sorry, something went wrong. Fix it however you want. But we can do better here. We know the original domain is still up, so we can do feed discovery on that domain. Much of the time, the feed discovery will find the new URL of the updated feed, and we will have fixed the problem. **What we’ve done here is effectively add a “Fix It” button to the error.**

Don’t Be Lazy: API

- Create an object implementing the **NSErrorRecoveryAttempting** protocol.
- Set that object as **NSError**’s **recoveryAttempter**, and set **localizedRecoveryOptions**.

There’s a great API for this on OS X, with a protocol called **NSErrorRecoveryAttempting**. It’s another one of my favorite underutilized Cocoa APIs. It lets you add any number of these “fix it” buttons to your error dialogs. The information on which buttons to show becomes part of the **NSError** object itself, and is passed around along with it. It’s not just text, though. The **NSError** object also contains a reference to whatever object is responsible for implementing each of the different fixes! This makes it really easy to structure your code so you can use fix it buttons to make error recovery suck less. Ever since I found this API, I’ve been looking for these buttons in error sheets.

To me, these sort of solutions are another example of **having compassion for the user**. Handling these sort of problems, easily and if possible automatically, is what they pay us for.

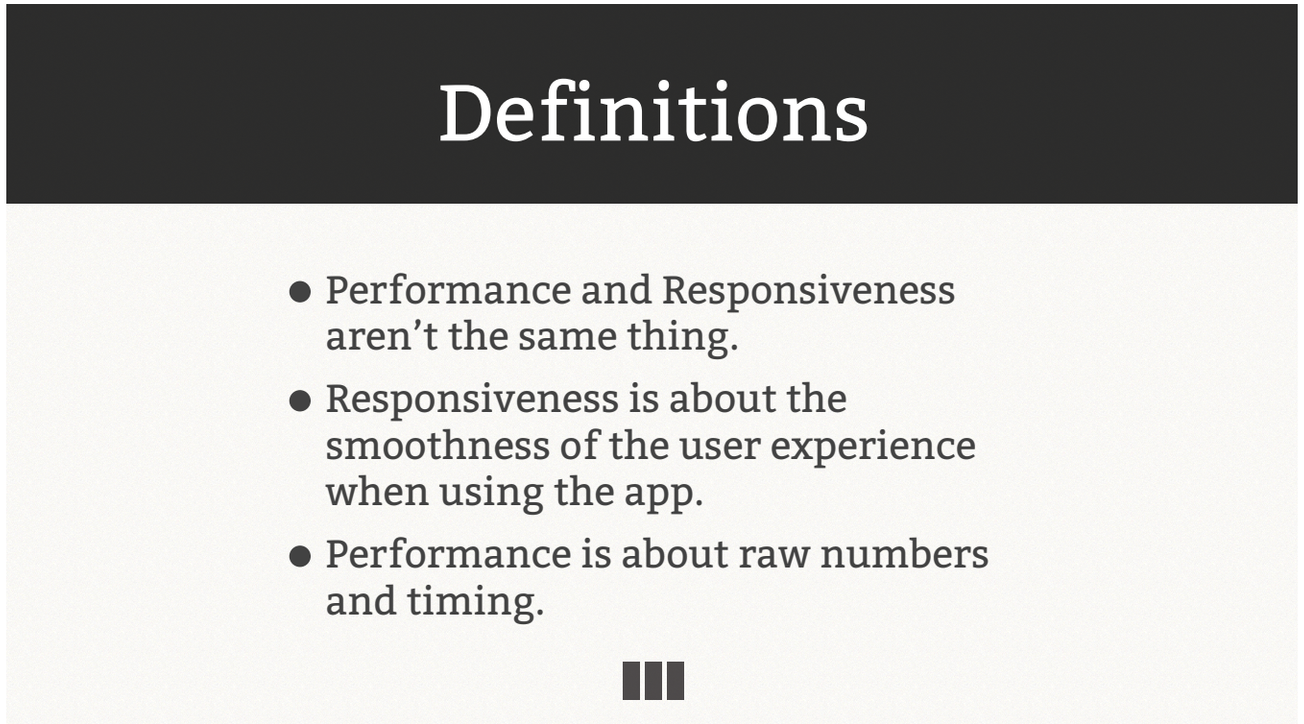


The next big area I want to cover is performance and responsiveness. The main reason I want to talk about this here is that **I think it's really hard to overestimate how much of an effect lack of responsiveness can have on the user experience of your app**. Emotionally speaking, a lack of responsiveness has a really incredible ability to break a user's flow, frustrate them, and undermine their confidence.

Maybe the closest example of this for the people in this room is Xcode. How many times have you been in the zone, typing away at code, when suddenly ... you see this guy here.

Now you're angry, you're out of the zone. You're not sure whether it will ever stop, or maybe you'll need to force quit it. You're not sure whether you're going to lose your work from the last few minutes or not. Eventually you develop a reflex to hit Command-S every 4 or 5 seconds just be sure, and so on, and so forth ...

It's more than that, though. If your app is consistently slow, to a noticeable extent, there is a psychological friction to using it. When Facebook made its iPhone app twice as fast last year, the amount of content people consumed with it **doubled**. And this was with no UI changes at all. If that's not an excellent example of why performance and responsiveness matter, I don't know what is.



The first thing I want to talk about here comes down to some definitions. This might be obvious, but I think it's important to note so that the rest of what I'm going to talk about makes sense.

Performance and responsiveness are two very different things. Responsiveness is about the feel of the app. It's about whether the user's flow is broken, and for how long. It's about whether the app is always ready to do what the user wants, when they want it. Performance is a much more quantifiable thing. It's about how fast NetNewsWire can import a 2000 feed XML file, for example.

Let me start by saying that these are are both very important. I'm certainly not suggesting that you use n squared algorithms or anything like that. That said, when you get complaints from users that an app "is slow" or "lags", most of the time they're talking about responsiveness, not raw performance. Meanwhile, as engineers, performance is what we tend to focus on. How can we optimize those CoreData queries? Where can we find a more efficient algorithm? These things are fun to work on, they're within our comfort zone, and, most importantly, they are wonderfully quantifiable things. Instruments tells us exactly what kind of progress we are making. I know I have often fallen into the trap of confusing these two, and working on trying to make my queries faster, when really it's the user experience that people are complaining about.

Most of us have already heard a ton about the tools we can use to improve our raw performance, and lots of people have given wonderful talks on this subject. That's why today, I'm going to focus mostly on how to make your app as responsive as possible.

Keep It Moving

- For any event that may take a significant period of time, have something moving on the screen.
  - Think about how the time of these operations scales.
  - If nothing is moving, the app is stuck.
- Use determinate progress, but only if you can use it accurately.

I would say that the number one most important part of responsiveness is just keeping your app moving.

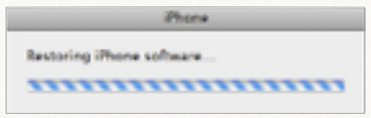
If your app is doing some user-requested work, and it might take any significant amount of time, you need some kind of an indicator of progress. By “any amount of time,” I’m really talking about any perceptible delay at all. Humans are excellent at being bothered by this kind of lag, even if it is just a second or two. Notice that I’m using words like “may” or “might.” Just because it seems really fast on your 32 Core Mac Pro monster development machine, doesn’t mean it’s going to be fast on my Mac Mini, especially when I’ve been a customer for years and have filled up a huge database. That’s why instead of thinking about seconds, I like to think about scaling. Does something get slower and slower depending on how many feeds the user has, or depending on how big the input file is? Then it certainly needs some kind of progress.

Progress can be a pain in the ass to implement. You’ve gotta use you blocks or delegates, you’ve gotta add new API in your app. Why bother doing it? It’s just a progress indicator, you’d think, right? Why is it so important? **It’s important because it provides reassurance, and it builds confidence**. Without that indicator, your app is stuck. As far as the user’s concerned, it’s frozen and they’re not sure whether it’ll come back in a bit, or crash, or lose the work they’ve been doing. Reassurance like this helps build trust with them, and makes them comfortable in your app.

I also wanted to talk about determinate progress. It’s always a nice thing to do, if you can do it. But unless you can do it reliably, don’t do it at all. If you see cases where your determinate progress doesn’t advance for a long period of time, drop it and do a regular spinner instead. Here’s an example from the App Store on the iPad. This Evernote update was sitting in this state for maybe 20 seconds. Probably it was copying files or something, but from my perspective, was it hung? Did I need to stop it and try again? Restart the iPad maybe? Even I got frustrated when I hit this. Reassurance that the gears really were turning would have helped me a lot here.

If You Can’t Perform, Cheat

- For user-visible responsiveness, cheating really does work!
- Make everything you can non-modal.
  - For secondary interactions, background them and show progress.
  - For primary interactions, do only what’s absolutely necessary, background the rest.



As I was talking about earlier, responsiveness and performance aren’t the same thing. With performance, it comes down to a numbers game and you end up having to optimize. Often times it happens that you can only win by using a smarter algorithm, or a different technology.

Responsiveness is different. In responsiveness, cheating is OK. In fact, cheating is one of the smartest things you can do to improve the feel of your app. **You don’t have to make it faster, you just have to make it feel faster**. I’m going to go over a few tricks I’ve found for doing that, along with some examples.

The first one is to use background and non-modality to your advantage. That means being very smart about where you’re blocking the user’s workflow, and where you’re not.

Some interactions in your app don’t have to be modal at all. These are often secondary things that the user is not necessarily waiting for. One example of this is importing OPML in NetNewsWire. We tried to make it faster, but it was still really slow. Instead, we cheated. We backgrounded the entire process so that the user didn't have to wait at all for it to finish. That took a lot of the pressure off in terms of frustrated users thinking it was too slow. Here’s another example, from iTunes. How many times have you been frustrated waiting for the iOS device to restore? It is really, really slow, but it would be so much less annoying to use if it spent that same time working in the background.

There are other interactions, though, that really are **primary**. Those are the cases where the user really is waiting on your app to finish doing something before they can move on with their workflow. **Even in these cases, if you can’t make it fast enough, you can still cheat**. Think very carefully about what you are doing, and when. Try to find a way to do only the absolute minimum amount of work, and make the rest happen in the background once you’ve unblocked your user.

We had an interesting example of this in NetNewsWire, while working on deletion. We found that it takes forever to delete even 200 feeds. You’d hit OK on the confirmation sheet and the app would hang for like twenty seconds. We could background the whole thing, but it would be confusing to have feeds

If You Can’t Perform, Cheat

- If you can’t make it faster …
  - Break up the work, fill it in gradually.
  - Do Less Work: Use proxies or promises.
  - Anticipate: Guess and start the work earlier, speculatively.
    - **NSCache** and **NSPurgeableData** can make this safe and easy to do.

There are three more useful techniques I want to mention for this sort of cheating. These might be pretty obvious; they’re all pretty well-known techniques from computer science. The reason I choose to mention them here, though, is that smart use of them really does have an effect on user experience. I know I personally have gone down rabbit holes when I’ve forgotten that my problem could be solved by just this sort of cheating.

First, consider breaking up your work into units of batches. Run each batch and fill in the results as they come in, as gradually and as smoothly as possible. Even if it might bother you a bit in a purely philosophical sense, the user doesn’t care that not all of the 2000 feeds are loaded in the tableView right now. They’re a human who has a very limited ability to keep many things in mind at once. Implement this right, and they’ll probably never notice the difference.

The second idea is a related one: proxies and promises. If you are having trouble making some interaction perform well enough, try and understand what the absolute minimum of information you need to show really is. Focus on that minimum amount. If the existing architecture of your app doesn’t match exactly what you need on a particular screen, you don’t necessarily need to rewrite it. Use a proxy, a promise object, or some other design that can hide the partially loaded nature of your model. CoreData does this all the time, right under our noses.

My favorite example of this comes from when I worked on Automator at Apple. Automator’s library consisted of hundreds of actions, each of which is associated with some code sitting inside a bundle in a different place on the disk. Especially back then, before we had SSDs, loading hundreds of these things was incredibly slow. Our solution was to pull a trick similar to CoreData. Automator loaded an absolutely minimal amount of information for each action from a centralized cache, and hid the fact that they were actually incomplete objects until the rest of the data was really needed. Only a few actions were in each workflow, so much of these bundles were never even loaded. It worked great, and again, no one was the wiser.

If the above two aren’t cutting it, there’s one other approach that can be useful. We design our apps with certain use cases in mind, and that leads to a structure for how people interact with it. This structure is something we keep in mind while architecting the code, and while testing it. It has another use though: It lets us anticipate.

Technical Bits

- Avoid blocking the main thread on disk IO: Use **NSOperationQueue** and **libdispatch.**
- Understand your database sizes.
- Tools
  - Network Link Conditioner.
  - Instruments: Time Profiler, Core Data Profiler.

Before I finish up with performance, I wanted to add a few more technical things. I'm going to start with two common issues that can easily cause responsiveness problems if you're not careful.

The first one has to do with disk IO. You should be very wary of accessing the disk with APIs like NSFileManager when you are doing it on the main thread. Unlike URL downloading, NSFileManager will by default block your app and can cause it to beachball. This is especially pernicious because most of us developers work on very modern, SSD-equipped Macs. It is both a blessing and curse of these machines that they make disk IO almost instant. The differences are really huge, as in orders of magnitude. It's surprisingly easy to write code that runs instantly on your machine, but takes 30 seconds on even a relatively modern Mac Mini without an SSD. Even forgetting the issue of the type of disk, lots of things can make an IO operation take an eternity, such as a very busy drive, mounted network shares, or even network-based home directories. Instead of developing on a slower machine, or keeping around non-SSD macs for testing, the way I've dealt with this in the past is to avoid disk IO on the main thread whenever possible. **NSOperationQueue** and **libdispatch** make this pretty easy most of the time, and it is well worth doing in order to keep your app stutter-free.

Next on the list has to do with user data. These days, many of our apps use CoreData in one way or another and deal with a single file containing all of the content the user has ever put into your app. One of the most important things you can do to ensure that your app stays responsive is to have a really clear idea of what this content looks like, and especially how big it is. If at all possible, use a feedback mechanism to communicate this size information to you, so you can build a histogram of how big these databases are in the real world. Look at that data, and choose an average size to target, as well as maximum size that you're willing to worry about. In my work on NetNewsWire, those numbers have really helped us stay focused on the user-visible aspects of responsiveness, as opposed to theoretical performance.

Keep in mind that when you're dealing with work that scales with the size of the database, the performance you're seeing on your test set may be very, very different than what your customers are seeing at home.

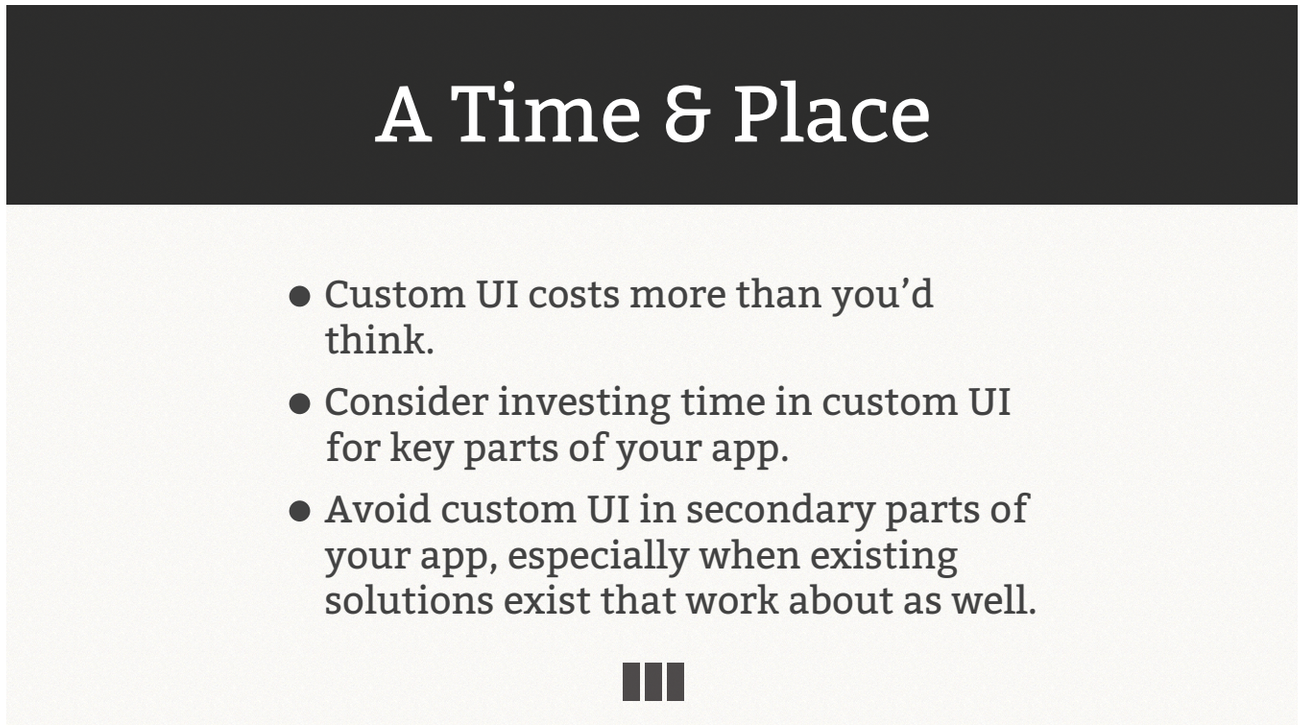
On the bright side, if you do manage to build an app that can able handle whatever your customers throw at it, that can be a real competitive advantage.

Custom Controls & UI

Everyone wants a beautiful app. We all want to stand out from the competition. And, well, the screenshot-based marketing that is facilitated by the app stores on both platforms doesn't help either. Because of all this, there's a pretty big temptation to create custom UI and controls.

By "custom", I mean that rather than customizing images or text in a standard way that is designed into the Cocoa frameworks, we do more of our own work. That usually means that you're doing your own drawing at a lower level, or implementing custom behavior in terms of how the controls interact with your users. All of the above, by the way, is much more likely in AppKit than in CocoaTouch, because of AppKit's rather frustrating lack of easy customizability in many areas. That said, I think a good guideline to use is that if you're writing lots of code for event handling, doing lots of custom drawing, and especially if you're re-implementing significant chunks of behavior already in Cocoa, that's custom UI.

Doing an excellent job at creating custom UI can lead to some really exceptional pieces of software, both in terms of visuals and in terms of function. But achieving that is actually very, very difficult. In fact, it's quite a bit harder than it appears at first. Meanwhile, attempting to pull off lots of custom UI and not doing a good job of it can be very damaging to a product. That's why I want to spend a few minutes talking about why doing custom UI well is so difficult, and how to a great job of it.



A good first step is to understand when and where custom UI is appropriate, and where it's best avoided.

Probably the most important thing I want to get across here is that custom UI costs more than it seems to. I'll get into some of the trickier bits a little bit later, but doing custom UI right is a very difficult job. If you want to do it right, introducing even a small custom control can quickly pile up in terms of time to spent re-implementing standard behavior, and worst of all, debugging the corner cases.

That's why I would really suggest that if you do custom UI, you focus on it for one or two areas of your app that are really key to the whole product. That way, it will be more likely that you'll invest the time and effort needed to do the job well, and when you do, it'll be something worth highlighting. It can even be a competitive advantage to your product in a crowded market.

Marco Arment talked about one such example recently on his podcast, when discussing his upcoming Podcast client, Overcast. He was considering doing some serious custom UI for his scrubber, including the ability to adjust the scrubbing speed in real time, zoom in for more accuracy, and more. That would be a significant time investment to build and maintain, but it is potentially worth doing so in a crowded market to create a showcase part of the product. If he was considering doing something really crazy like that for the Podcast finder interface, well, maybe that wouldn't be as worthwhile.

Again, whenever you create custom UI, keep in mind that in addition to the upfront cost of creating it, you will probably be paying a recurring cost for the maintenance of it as well. The more custom work you do, the more likely that it will break with new operating systems, and require updating for new designs from Apple. Thats why, if you can find an existing stock control that does what you want, or one that you can re-theme to do what you want in an easy and safe way, it's probably a good idea to use that instead.

Doing It Wrong

- Apple's UI controls, especially on the Mac, contain tons of subtle behaviors and interactions.
- The differences between the way stock controls act, and the way yours act, cause confusion.
- Either take the time to get the experience right, or use standard UI instead.

Apple ships lots of its own controls with iOS and Mac OS. Everything from buttons to tableViews to textViews. As engineers, we all know that these are "just code", no different in essence from our own code. As product designers, though, the situation looks a little different. Apple's NSTableView implementation, for example, includes twenty years of development, fine tuning not only how it draws and performs, but exactly how it interacts with the user in every possible corner case. So, it is "just code", like our code. But it is **a lot of code**, written by incredibly smart people, tuned and bug fixed over a very long period of time.

That means, first of all, that it's just plain difficult to duplicate. Not only is it a lot of work to implement all that, but it is even hard to work to figure out exactly **what** to implement. Just figuring out exactly how NSOutlineView acts during drag and drop, with all the corner cases, is a job in and of itself.

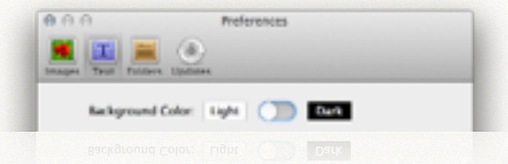
There's another thing to keep in mind about Apple's controls though. They are system controls. That means that every person who has used a Mac has used them a million times. In Mail, in Safari, in Messages, and Calendar. It's like the literature metaphor I used at the beginning of this presentation. Your typical Mac user may not know the exact details of every behavior of these controls, but they **do** notice them. They put all those tiny behaviors and interactions together, into a mental model of how the text fields, radio buttons, and tableViews are "supposed to" work. **What happens when they use your controls, which are really kind of similar to the Apple ones, but they look different, and act just slightly differently? What happens is an uncanny valley effect.**

It's a tableView, but the selection doesn't work quite right. It's a button, but it doesn't track the same way, or the alignment of text is a little bit different. Now, your app is somehow off, in a hard to identify way. It just doesn't feel right, and it doesn't fit in. And when it doesn't feel right, in the end, the impression it gives off is that the people who created it didn't really sweat the details.

That's why my suggestion is this: **If you're considering using custom UI, either take the time to get the experience right, or use standard UI instead.** There's nothing wrong with taking advantage of all the hard work that the engineers at Apple have done over the last twenty years.

Doing It Right

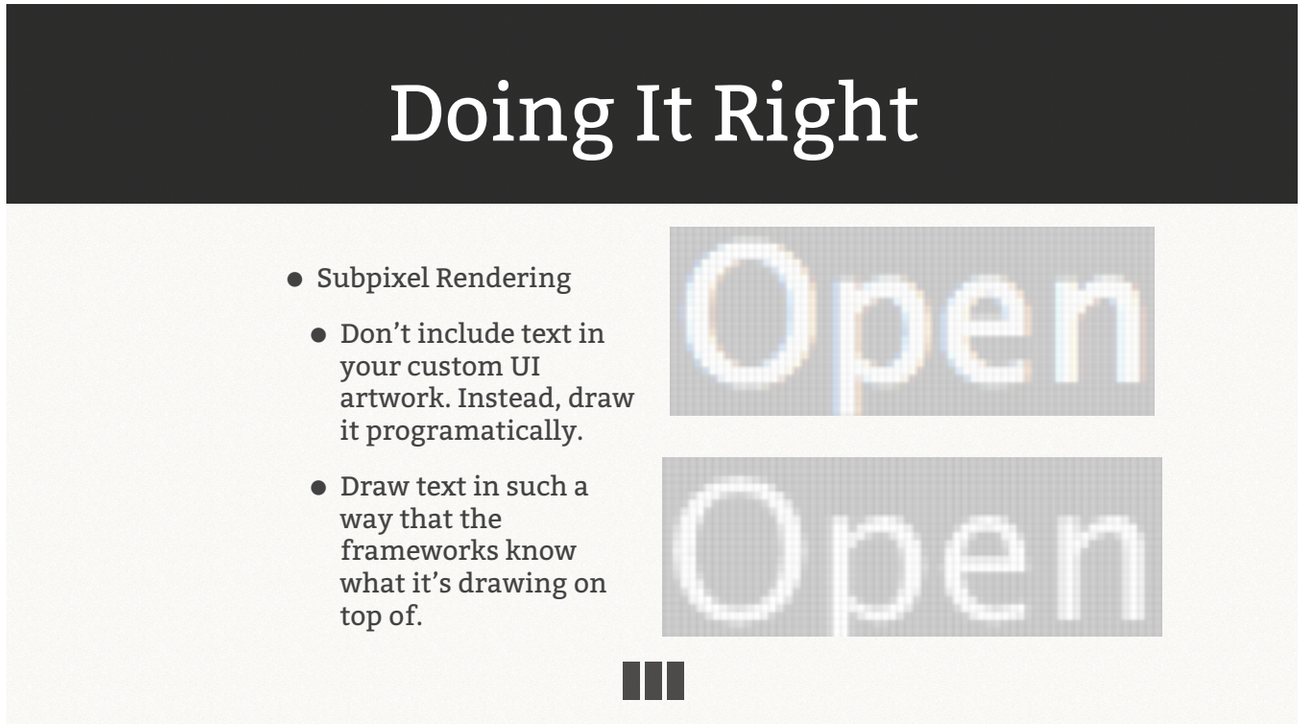
- Follow existing mental models.
  - Match the behaviors of the stock controls exactly, if you can.
  - If you can't, try to find existing examples in Apple's apps.



I talked about mental models earlier, and that's the subject of my first suggestion for custom controls. **If at all possible, find a mental model you can use that most users already have created.** Every time they see something that doesn't immediately make sense, something unfamiliar, it's more cognitive load. That particular screen becomes that much more intimidating and complex. The best place to take interaction models from is probably Apple's own controls. They're present everywhere, in every Apple and third party app on the platform. They're immediately familiar, and if you can match those behaviors, your controls will immediately make sense.

What if you can't find something to work off of within the set of standard controls? In that case, I'd go to Apple's own apps, inside and outside the operating systems themselves. These are not as pervasive as the standard controls, but they will still be very familiar to most people. The successful cases where Apple has done this have even been picked up by the community, including adoption in lots third party apps and open sourced versions of these controls.

Here's an example from Kaleidoscope. We wanted something a little bit clearer than the standard radio buttons to switch between background colors, and we wanted something visual. We found that the on / off switch for Time Machine backups matched this really well. Ever since Apple introduced this switch, this has become a common control. We used an implementation that matches it pretty closely in appearance, but matches it **very** closely in behavior. That includes things like the way you can slide the switch with the mouse, and the way it snaps to left and right positions, too.



I was saying earlier that Apple’s custom controls have twenty years of work behind them. That shows up in lots of ways, but there’s one specifically I wanted to focus on here because it’s subtle, but visual.

Apple’s controls always render text with subpixel rendering or subpixel antialiasing. That means that they will use the their knowledge of the physiology of your eye and the specifics of arrangement of pixels in your LCD monitor, plus their knowledge of the context and background into which the text is being drawn to make the text look smoother.

One of the easiest ways to test for this is to use something like Xscope and zoom into your text. If your subpixel antialiasing is working, you will see fringes of color, even on black or white text (see the top image; can you spot the blue and red?). If it isn’t working, text will appear thinner and more jagged, and you won’t see any color fringes.

What happens if you don’t have subpixel rendering? Of course, it’s not the end of the world. But this, like many of the other small things in this presentation, makes a difference. Your text will look just a little bit off, and just a little bit less smooth than the text in Apple’s standard controls.

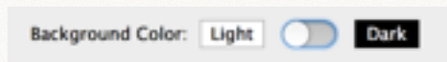
How do you make sure that your custom UI doesn’t break subpixel rendering? That’s sort of a tricky question, since it depends a lot on exactly how you’re using the frameworks. But there are two big things that tend to cause problems here.

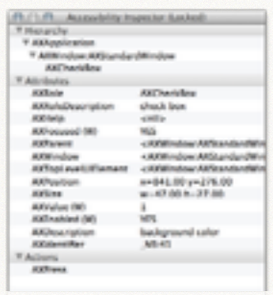
The first is when you are including text in your UI artwork itself. Doing that seems like a nice shortcut if you know exactly what text will be in there, and you know it won’t change. But if you do that, there’s no way for the system to do subpixel rendering, since the exact pixels are encoded in the image itself and that means they can’t be tweaked based on the exact LCD monitor being used.

The second problem has to do with the context into which you draw your text. This has been getting better and better with each release of OS X, but it’s still easy to get AppKit to stop doing subpixel rendering by using views layered on top of a transparent window, for example. The only way to be sure is to

Doing It Right

- Accessibility
  - **NSAccessibility.h** (on the Mac).
  - **- [NSObject accessibilityAttributeNames]** and **- [NSObject accessibilityAttributeValue:]**.





Continuing on the theme of what Apple's controls do for you pretty much automatically, there's also accessibility. Apple's got an incredible history of being one of the most accessible platforms around.

Every time you build an app that's accessible with tools like Voiceover, you make impossible things possible for your disabled customers. The accessibility API on the Mac is unfortunately not as strong as the one on iOS, but in both cases, the more you use Apple's frameworks to your advantage, the less you have to do.

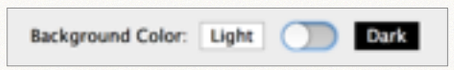
If you do custom controls, accessibility becomes something you need to worry about. You'll need to implement and test it yourself. **NSAccessibility.h** contains most of what you need to know to do a good job on this on OS X.

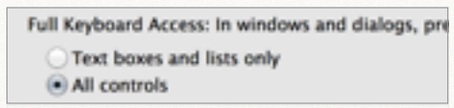
You're going to have to implement the methods from the **NSAccessibility** protocol, like these, to give accessibility the information it needs to know about your custom control.

Here's an example of what the accessibility results look like for the custom slider control in Kaleidoscope. As you can see, we've put in a specific description that voiceover can read, added some information about roles that conveys that this slider is essentially a checkbox, and even added a press action so a blind user can order Voiceover to activate the slider for her.

Doing It Right

- With Full Keyboard Access turned on, you app should be fully accessible.
  - **- [NSApplication isFullKeyboardAccessEnabled]**
  - The key view loop should cover all your controls, and be contiguous.
  - Use focus rings and focus states to make the current control clear.





Another thing to keep in mind when doing custom UI is keyboard control. Using your app via the keyboard is really important, both for power users that want to get things done as quickly as possible, and for impaired users that can't use the mouse successfully. On OS X, Apple has a setting called Full Keyboard Access that controls this. In one mode, you can tab through all the controls in a window. In another, you can only tab through the text boxes and various kinds of lists.

Of course, Apple's controls handle all of this automatically for the most part. Once again, when working on custom controls, this is something you're going to have to worry about yourself. You'll need to use that method on NSApplication to determine what kind of full keyboard access is enabled, and then adjust the key view loop as needed. You'll need to figure out ways to show that your control is active, like this custom focus ring being drawn by the slider control here.

There's probably more than can be added to this list, but I think I've gone for long enough on this that I hope I've conveyed a point to you: When you write custom UI, the farther you go down that route, the more you give up the benefits of the twenty years of incredible work that Apple has done to make all of us more productive.

There are huge benefits to doing this. You can create some awesome interactions that can be a big differentiator for your app. But, if you do go down this route, keep in mind that you're going to have to spend up front time and ongoing maintenance time learning and re-implementing a lot of the thorny bits that would otherwise have been handed to you.

I love custom controls in apps, so please go and build them. But please build them right. Your customers will thank you for it.

Conclusion

I'm pretty much out of time, so that's about all I've got for today. The one thing I wanted to leave you with is this:

There was a ton of stuff in this talk, and it's pretty much impossible to get all of it right and still, you know, ship. That shouldn't be a discouragement, though. I bring these things up not because you should get everything perfect in version 1.0, but because these are the sorts of things that we strive to get right. Polish isn't something you do all at once.

Polish is an ongoing process of improving the tiny details of your product. It's the drive to do that that defines us as Mac and iOS developers.

If we have a few minutes, I'll take some questions. I'll also be around during the conference, so please don't hesitate to come and say hi. Thanks for listening.