⚠️ Clear simulator settings

⚠️ Hide macOS menu bar

⚠️ Quit all macOS apps

⚠️ Turn on presentation focus mode

⚠️ Pre-warm Scrumdinger tests

⚠️ Open Xcode projects

Control your dependencies

don't let them control you

Hello, today we are going to talk about dependencies. This includes defining what dependencies are, showing why they can be problematic, and then describing how to take control of them rather than letting them control you.

Brandon Williams

brandon@pointfree.co

@mbrandonw (@hachyderm.io)

Stephen Celis

stephen@pointfree.co

@stephencelis (@hachyderm.io)

But first a little bit of information about myself. Here is my email, Twitter and Mastodon handle in case you want to get in touch or have questions.

Also, this talk is joint work with my collaborator Stephen Celis, and so here is his information too. You can reach out to either one of us.

Point-Free

www.pointfree.co

@pointfreeco (@hachyderm.io)

And if you weren't aware, Stephen and I run a site called Point-Free, where we go deep into topics like the one I am about to discuss today, and a whole bunch more. I encourage you to check it out if anything in today's talk catches your eye.

What is a dependency?

Let’s start with a seemingly simple question: What is a dependency?

This is a surprisingly tricky question to answer, and I can't claim to have the most universal definition, but for the purposes of this talk we will define dependency as the following…

What is a dependency?

> *The types and functions in your application that need to interact with outside systems that you do not control.*

Dependencies are the types and functions in our application that need to interact with outside systems we do not control.

Now that definition may seem a little nebulous, so let's look at some very concrete examples…

Examples of dependencies

Network API clients

```
let users = try await apiClient.fetchUsers()
```

Network API clients are a "dependency" because they make network requests in order to load data from external servers. Typically those servers are not things under your control, such as if you are hitting the Stripe API for payment processing, or Mastodon API to load new posts.

Examples of dependencies

Location managers

```
let manager = CLLocationManager()

await manager.requestWhenInUseAuthorization()

let location = try await manager.requestLocation()
```

Location managers are another example of a dependency.

If you have ever used Core Location in your app, then you were using a dependency. Not only do you interact with Apple's frameworks for fetching location data, which you do not control, but those APIs interact with the GPS instruments in your device in order to triangulate coordinates. That is a major outside system interacting with satellites orbiting the earth.
Examples of dependencies

File systems

```
var data = try Date(contentsOf: URL(…))

data.append(…)

try data.write(to: URL(…))
```

If you have ever saved to or loaded from the disk, then you were using a dependency. The file system is another massive external system because it exists outside your code base. Any application can read and write to it without your knowledge, and so it is not something your application can control.

This also extends to user defaults too. Anytime you read or write to user defaults you are indeed interacting with the file system, and so that is a dependency.
Examples of dependencies

Firebase

```
import FirebaseDatabase

let ref = Database.database().reference()
self.ref
    .child("users")
    .child(user.uid)
    .setValue(["username": username])
```

Firebase is another **massive** dependency. It's kind of a *mega* dependency, because it acts as a database, acts as a network API client, handles analytics, crash reporting, logging, remote config, authentication, push notifications, storage, and more! All of the code that powers these features is written by Google and hosted on Google's servers, and so is fully out of your control.
Examples of dependencies

Clocks and schedulers

```
```

```
try await Task.sleep(for: .seconds(1))


DispatchQueue.main.schedule(after: .now + 1) {
  …
}
```

If you have ever used `Task.sleep`, or `clock.sleep`, or a Combine `Scheduler`, or even just a dispatch queue in general, then you were again using a dependency. Those objects speak with the outside world in order to execute work at a later time, and we have no ability to tell those processes to do things in a different manner.
Examples of dependencies

`Date()` and `UUID()`

```
let user = User(
  id: UUID(),
  name: "Blob",
  createdAt: Date()
)
```

And then seemingly innocuous things such as `Date` and `UUID` initializers can also be considered dependencies. After all, when you ask Foundation for the current date it reaches out into the real world to get that information.

The same is true for `UUID`s. Each time you construct one you get a completely random sequences of hex digits, and the process that concocts that value is an external system that we do not control
Examples of dependencies

…and *a lot* more

And this is only scratching the surface. There are many, many, *many* more examples of dependencies, and your code today is probably full of them.

The problem with uncontrolled dependencies

So, if you believe everything I have said so far, then dependencies are seemingly ubiquitous in application development. They are literally everywhere, and if this is the first time you are hearing about “dependencies” then you may wonder what the big deal is.

You've been coding for years, apparently using dependencies left and right, yet you've never stopped to think about it and you get along just fine. So, why should you care?

Well, I'm here to tell you there are a lot of problems that occur when you have uncontrolled dependencies. You probably have these problems in your code base right now, but you may not know that dependencies are to blame for it.

And you may have also come up with all types of work arounds for the problem, or just learned to adapt and live with the problems.
The problem with dependencies

Many are slow to compile

```
import Firebase
```

Build succeeded 86.9 seconds

Perhaps the biggest problem with dependencies is that very often they are extremely slow compile. It takes nearly 90 seconds to compile a fresh project with Firebase added to it on my M1 MacBook Pro. That is nearly 90 seconds just to compile Firebase, not including all the code that needs to be built for your application.

This means every time you do a clean on your project you incur a minimum 90 second cost to rebuild. Or if you change branches to work on a different feature, again you are most likely going to incur that 90 second cost again. This time really starts to add up.

Now this does not apply to Apple's frameworks because those frameworks are all pre-compiled and included automatically. There is no compile-time penalty to using Core Location. It's just ambiently always there. This downside mostly applies to 3rd party frameworks, such as Firebase, or web socket libraries, and more.
The problem with dependencies

Some make it annoying to run your code

Another problem with dependencies is that they can just make your code more annoying to work with. Because they reach out to uncontrolled external systems you are at the mercy of those systems in order to see how your code even runs.

I have a demo for this. I'm going to switch over to the simulator, where I have an app running that has lots of demos that we will be playing around with in this talk. The first demo is this "Countdown" feature.

– Demo feature in simulator
– Mention that confetti layering is not right
– Show the code and fix it, show you have to wait 10 seconds again
– Previews are supposed to help with things like this
– Previews don't help
– The red flag is `Task.sleep`
– That's a global, uncontrolled dependency
– We are forced to wait real time to pass
– We can hardcode a smaller `countdown`

– But that's also bad, test values in prod code

So, we are now seeing in very concrete terms that by not controlling this dependency, the `Task.sleep`, we are letting it control us. It is forcing us to either wait for real world time to pass to test this feature, or to sprinkle in hacks. Neither of which is good.

The problem with dependencies

Some do not work in Xcode previews

Ok, so already it's seeming pretty bad to not control your dependencies, but things get far, far worse.

In the previous example we just considered, at least the feature worked in the preview. It was just annoying to use the preview because we had to wait for real world time to pass in order to see the feature's behavior.

But there are many dependencies that simply do not work in Xcode previews, and that completely ruins your ability to iterate on designs, functionality, fix bugs and more.

In fact, *most* of the 1st party frameworks that Apple gives us access to simply do not work in previews, including Core Location, Core Motion, speech recognizers, Store Kit, the Contacts and Address Book frameworks, Game Center, and a whole bunch more.

If you indiscreetly sprinkle direct access to those frameworks in your code, you have a good chance at just completely ruining your ability to quickly iterate on your features in previews.

I have a demo to show exactly how this can happen. Let's go back to the simulator, and this time let's go into the "Location Demo".

– Run location demo in simulator to show how it mostly works just fine
– Walk through code to show it is reasonable
– Show that feature does not work well in preview
– Core Location just doesn't work in previews
– Can't show authorization alert
– So we are stuck previewing only very basic functionality in the view
– Can't see how the feature reacts to the location manager
– We are forced to run on simulator
– But even that isn't ideal since once you grant permission you can't un-grant it to get the authorization alert again.
– You have to delete / re-install app

There are even some dependencies that don't work in *simulators*. For example, Core Motion has some interesting APIs for reading accelerometer and gyroscope events from the device, but none of that works in the simulator. You have no choice but to actually run the app on your device if you want to play around with that behavior.

We are again seeing that by not controlling our dependencies we are allowing them to control us. We simply are not able to use Xcode previews if we use certain frameworks directly in our code.
The problem with dependencies

Some *break* Xcode previews
So, it's a pretty big bummer that previews become less helpful as you start to use certain dependencies in your code base.

But things get much, *much* worse. Some dependencies just completely break previews entirely. To demonstrate this lets head back over to the simulator and take a look at the "Contacts demo."

–
Run demo in simulator to show how it works
–
Walk through the code to show it is reasonable
–
Show how previews are again mostly non-functional
–
Modularizing apps is increasingly popular in the iOS community
–
It's a great way to decouple and isolate features, and speed up compile times
–
But if we copy and paste contacts demo to an SPM module we will see preview crashes
–
This is because of missing info plist entry
–
Some may know of a workaround
–
We can create a new "core" view that holds only data and use it from the view that interacts with dependency
–
Show that and show the preview works
–
Sounds better in theory than works in practice, for a few reasons

First of all, most views are not just simple, inert representations of data. Most views have behavior, such as buttons to tap, textfields to enter text into, gestures to perform, and more.

In order to support that we are going to need to bloat this seemingly simple "core view" with all kinds of callback handlers and communication mechanisms so that it can communicate back and forth with the real view that can actually interact with the contacts framework. Doing so can be quite complicated to get right.

And second, we are maintaining a bunch of additional code just to work around the fact that we have an uncontrolled dependency in our codebase. All code is a liability in that it gives you new ways to get something wrong.

Third, this view is only a mere shadow of the feature. A mirage. We are cramming data into it to make something appear in the preview, but the code path that is actually responsible for getting the data is not being exercised at all in the preview, and therefore we have no choice but to still run the full application in the simulator if we *truly* want to see how this screen behaves when it interacts with the contacts dependency.

And finally, the code needed to maintain all of these little inert views that only hold onto data vastly outweighs the amount of code needed to control your dependencies.

If you have 20 views that all use a single dependency then you will need to maintain 20 core views just to avoid the dependency. Or you could control the dependency a single time and make use of it in as many views as you want.
The problem with dependencies

Accidental interaction with “live” dependencies
We’ve so far covered a lot of very in-your-face and pernicious problems when using dependencies without abandon in your code. Next I want to talk about a much more subtle problem that can remain hidden from you for a long time until it finally bites you in the butt.

And that is accidentally using “live” dependencies when you don’t mean to. To explore this, lets head back to the simulator yet again to check out the “Analytics demo".

– Run analytics demo in simulator, open logs and filter to show tracking
– Walkthrough code to show it is reasonable
– Run in preview, click around
– Open logs and show that we were secretly tracking live analytics data even though we are just testing the feature

Over the course of working on this feature for an afternoon I can *easily* refresh this preview hundreds of times. Each preview refresh tracks a new analytic event, and that is going to muddy up our data. We can no longer confidently make business decisions based on this data because it now holds events that didn’t actually come from our end users and instead came from our developers building new features.
The problem with dependencies

Difficult or impossible to unit test and UI test
And if *all* of that wasn’t bad enough, it gets worse.

I saved this one for last because testing isn’t exactly a top priority in the iOS community. I think that is a little unfortunate, and I would highly encourage everyone to really understand what testing brings to a codebase and consider what it would take to

make your code testable.

Tests are a great barometer for the health of your code, but I'm not going to get deep into testing, that is a talk for another time.

Instead I want to look at yet another demo to show off this problem, but this time we are going to look at a code sample from Apple.

Apple has this great tutorial called “Scrumdinger” which builds a pretty complex application from scratch. It deals with multiple forms of navigation and complex side effects, such as persistence, timers and speech recognizers. It’s pretty cool.

But also it isn’t built with testing in mind. It's pretty much impossible to write tests for the app because the vast majority of logic is trapped in views, and the code that isn't reaches out to live, uncontrolled dependencies.

But, one way to get *some* test coverage on an application regardless of the manner in which it was built is UI tests. UI tests allow you to literally boot up your app in the simulator and emulate a script of user actions on the screen so that you can assert on what happens.

– Open scrum dinger Xcode and launch in simulator

Well, I gave this a shot in Apple’s Scrumdinger app by writing a test to exercise the flow of the user tapping the “+” button, filling in some details, hitting “Add”, and then asserting that the root list has a single row with the details that were just entered into the form.

So, let’s run the test… and it passes!

But, let’s run the test again… and now it fails!

What’s going on?

Well, one of the features of this application is that it persists data to disk so that next time you launch you have all of your meetings from last time. That’s a great feature to have, but it’s also making testing complicated because now when we run this test it is loading up data from *previous* runs of the test. Our dependency on the global file system is bleeding over from test to test.

So we have no choice but to actually *weaken* what we are testing here. We can’t test that when we go through this user flow a single item was added to the list that matches “Engineering”, but rather we can only test that there is at *at least* one item with “Engineering”.

So, if we ever accidentally introduce a bug that causes more than one item to be added to the list our test will happily pass and we will be none the wiser unless we happen to actually load up the app and witness this behavior.

Death by 1,000 paper cuts

So we have seen over and over again that seemingly reasonable code can lead to some very unfortunate results. Everyone here probably has at least one version of these problems in your code base today.

Maybe it’s a preview that doesn’t work very well due to complex behavior and so you just run it in the simulator.

Or maybe you have a feature that uses a dependency that doesn’t work in the simulator and so you are constantly running it on your actual device just to play around with its most basic behavior.

Or maybe you have an application that takes a long time to build so you just make sure to never accidentally clean the project, or you keep multiple versions of your repo checked out so that you don’t have to switch branches.

And over time you’ve just grown to understand that these are the peculiarities of your application and learn to deal with it. You may have some tricks you employ to get around certain annoyances, and for the most part you are able to be productive throughout the day.

And that may be OK for you, but this does not scale at all, especially when you are on a team with multiple people because *then* you need to distribute your little bag of tricks that you employ daily so that everyone on the team can use them too and not be blocked by these quirks.

What I’m trying to say here is that uncontrolled dependencies can make it really, *really* annoying to work in a code base. Each of these in isolation is maybe not a huge deal, but it’s easy to have lots of uncontrolled dependencies that cause more and more problems, and the problems start compounding on each other.

What can we do about it?
So, that was just a lot of time spent on negativity. We saw problem after problem with seemingly reasonable code, so now we will devote time to some positivity. What can we do to fix the problem?

In short:

Don't reach out to code you don't own and can't control
Well, in short: stop reaching out to code that you don't own and can't control from the outside. This includes interacting directly with the 3rd party APIs, such as Core Location, URL Session, file systems, clocks, schedulers, date and UUID initializers, and more.

All of those things talk to external systems and so the moment you interact with one of those APIs directly in your feature code you are susceptible to the vagaries of the outside world.

Let's see how controlling our dependencies can fix a lot of the problems we just saw in our demos.
Controlling dependencies

Time

```
let clock: any Clock<Duration>
try await clock.sleep(for: .seconds(1))
```

Let's start with the "Countdown" demo. It's main problem is that we had to wait for real time to pass in order to see the confetti animation, and it was that way because we reached out to the global, uncontrolled `Task.sleep` method.

What if instead we passed in an explicit clock so that we can control time. By default we will use a `ContinuousClock`, but in the preview we can provide an "immediate" clock that squashes all of time into a single instant. When you tell it to sleep it just ignores you and doesn't suspend at all.

– Open simulator to show that there are "controlled" versions of all the demos we just explored
– Each demo works exactly the same as before, but the code has been slightly tweaked to control dependencies
– Open code for countdown demo
– Mention that code is identical to before except for 2 changes:
– -> a clock has been added as an instance variable
– -> and initializer has been provided so that any clock can be passed in
– By default a live continuous clock is used
– But in previews we can use something else, like an immediate clock
– The standard library doesn't ship an immediate clock, but we do in our swift-clocks open source library.
– Now the feature counts down immediately
– No need to wait for real time to pass
– And we didn't need to hack around and change production values with debug values
– So we accomplished this by adding just a few lines of code, and we still get to use the `sleep` method exactly how we wanted to naively, it just now goes through a clock instead of `Task`

Alright, so this is looking really promising. We've seen the very basics of controlling a dependency:

– Add the dependency as an instance variable to the type that wants to use it
– Then actually use that dependency instead of reaching to the global, uncontrollable dependency.

This turned out to be really simple in this case, but I don't want to lead you to believe it's always this easy. We are lucky here because we happen to want to control a dependency that is already abstracted with a protocol, `Clock`, which means it easy to add an `any Clock` to the view and then substitute any number of implementations at runtime.

Controlling dependencies

Analytics

```
let analytics: any Analytics = LiveAnalytics()

analytics.track("Authorization granted")
```

Most of the time we have to put in a little bit of extra work to actually control our dependencies.

Take for example the analytics demo we showed off before. It had the problem of using an uncontrolled dependency for tracking analytics events, and that meant that we accidentally tracked real events when running the app in the preview. That was really bad

The fix is to take back control over this dependency rather than letting it control us, but this time we have to do a bit more work.

- – Jump to the controlled analytics demo
- – Show that we have an Analytics protocol at the top
  - – Mention that this is not how Stephen and I usually do things, but we are keeping it simple for the talk
- – Observable object now takes an analytics client
- – Exposes initializer that defaults to a "live" version
- – And then we use the client instead of reaching out to the global, uncontrollable singleton
- – At the bottom of the file there are two conformances
  - – a live conformance that uses `URLSession`
  - – a loop conformance that does not
- – We can use the noop conformance in previews so that we can play around with the feature to our heart's content without worrying about accidentally tracking analytics

So that's great!

But, even this dependency wasn't so hard to control. The analytics client has a very simple interface, just a single `track` endpoint, and so it was very straightforward to slap a protocol in front of it and then pass it explicitly to the model.

Controlling dependencies

Apple Frameworks

This is not always the case, and things get more complicated as the dependency gets more complicated.

Take the location demo from earlier. Interacting with a `CLLocationManager` is a lot more complicated than that analytics client. It has multiple methods we need to call, and it has a delegate that feeds us a stream of events. It takes more work to control this dependency, but you only have to do it once and then can immediately use it in as many views as you want.

So, let's go back to the demos and see what the location demo looks like with this controlled dependency.

–

Open controlled location demo

– Show the protocol at the top and how it's a lot more complicate

  – instead of a delegate that you can override we expose a stream of delegate actions you can subscribe to

–

Observable object holds onto a location client instead of reaching out to CLLocationManager directly

–

Initializer allows passing in a different location client at runtime

–

Then we use the location client in the model's code instead of using CLLocationManager

– At the bottom we have two conformances:

  – A live conformance that actually interacts with CLLocationManager

  – A mock conformance that does not touch CLLocationManager and instead we can configure it's auth status and location

–

Then in previews we can use a mock location client to simulate the user's location being in LA

– We can even change the auth status to denied to see what happens when the user has previously denied

  – Right now the feature doesn't do anything with that info, but if it did we would be able to see it

–

Open the contacts demo to show it works the same

–

If we moved this code over to the SPM module like we did earlier it would no longer crash because in the preview we are no longer touching the contacts framework

–

We've completely removed it from the equation

Dependencies controlled

So, things are looking pretty great!

We have defined what dependencies are, we have seen how they can cause all types of problems in real code, and we have seemingly fixed those problems.

You just need to identify your dependencies, make them controllable by putting an interface in front of them so that you have the freedom to swap out live and mock implementations. When running your app in the simulator or on device you will probably want to use the live implementation, and then in previews and tests you will probably want to use a mock implementation.

Safety & Ergonomics

Well, it all sounds great, but unfortunately things are more complicated than they seem at first.

If what we have discussed so far is all I told you about dependencies and you went out and tried wrangling in the dependencies in your application, you would run into many safety & ergonomics problems, and it would probably be such a pain that you would give up

Safety

```
init(
  analytics: any Analytics = LiveAnalytics(),
  location: any LocationClient = LiveLocationClient()
) {
  // ...
}
```

The first problem is safety.

Because we didn't want to break existing code we provided a default when we wanted to pass in a dependency. This is extremely ergonomic to use in practice because we just don't have to worry about these arguments unless we want to actually control dependencies.

So, that seems good, but also severely hinders the safety of this code.

First, in a more real world application, you will have many objects that need dependencies, and they will nest inside each other. By supplying defaults to these arguments we make it possible, and very easy, to accidentally supply explicit dependencies to one part, but then not another part. That will mean part of your application is using the default, live dependencies and some other part is suddenly using modified or controlled dependencies. That can cause very strange, subtle bugs and is most likely not what you want.

So that's bad, but also since it is possible to create an object without supplying any dependencies, and more importantly, even possible to create an object while being completely *ignorant* of the dependencies that *could* be supplied, it is still easy for us to accidentally reach out to live dependencies when we don't mean to.

– Go back to code of controlled analytics demo
– Show that commenting out NoopAnalytics causes analytics to be tracked again
– It's on us to pass all dependencies
– If analytics is added at a later time we may have many previews that haven't been updated and will be tracking live analytics.

Ergonomics

```
init(
  analytics: any Analytics,
  locationClient: any LocationClient
) {
  // ...
}
```

So, safety is a really big issue with default arguments, so we may want to require all dependencies to be passed in explicitly.

But that brings up all new problems of its own. The simple act of adding a dependency to a feature will reverberate throughout the entire application.

Suppose this little location demo was many layers deep in the application. Maybe you have to switch to a tab, drill down to a screen, open up a sheet, and then drill down again to finally get to this feature. Well, if we add a single dependency to the location feature we are going to have to also add that same dependency to every single feature that touches the location feature, as well as every feature that touches a feature that touches the location feature, and on and on.

Adding a dependency has turned into a viral event in which you are now updating lots of objects in your code to take a dependency just so that you can pass it down the line, and the object doesn't necessarily even have any need for that dependency.

*Defaults are **ergonomic**, but not **safe**.*

*Requirements are **safe**, but not **ergonomic**.*

What we are seeing is the unfortunate dichotomy between safety and ergonomics. Often things that are safe are not ergonomic, and things that are ergonomic are not safe. You have to give up a little of one to get some of the other, and it's a real bummer!

“Dependency injection” library

This is what leads people to adopt what is known as a "dependency injection" library. The sole purpose of a dependency injection library is to make it possible to give our objects the dependencies it needs to do its job, but in a manner that is safer than an initializer with default arguments and more ergonomic than an initializer with non-default arguments.

Now, it's not really possible to offer full safety and full ergonomics at the same time, and so every library has to make tradeoffs.

And the stance the library takes on safety versus ergonomics will dictate what other kinds of features the library offer you for overriding and controlling dependencies in interesting ways.

– Cleanse
– Factory
– Needle
– Swinject
– Weaver
– ...

There are a decent number of dependency injection frameworks out there in the Swift community. Here are just a few, and each of these has their own trade offs based on what they choose to prioritize.

If you decide to control your dependencies, and I hope I have somewhat convinced you that it is worthwhile doing, then you may want to research all the various options out there and see how they can help you solve the problems you are having with your dependencies.

github.com/pointfreeco/

swift-dependencies

And it just so happens that Stephen and I maintain a dependency library. I am in no way suggesting you use our library. We have made certain trade offs and decisions that are a top priority for us but maybe don't make a lot of sense for you.

In particular, we wanted to fully embrace structured programming and structured concurrency when overriding dependencies, and so we use Swift's new `@TaskLocal` machinery to power the dependencies under the hood. This brings a lot of power when used correctly, but also completely prevents certain features that other dependency libraries have.

So, let's quickly see how our dependencies library can help improve upon the safety and ergonomics of the demos we have been exploring.

–

Go back to simulator to show there are even more demos down below.

–

These demos work exactly as the previous versions we've seen, but now dependencies are controlled using our library

–

First is the countdown demo. This code is *identical* to the original version, except with 1 new line added and 1 line changed

–

Now we mark our dependency with `@Dependency`

–

And that simple act makes an initializer not required

–

When you register a dependency with our library you get to specify the default value used, and in this cause it's a continuous clock.

–

That means the preview down below uses a live continuous clock and we have to wait for time to pass

–

But we can wrap the creation of the view in `withDependencies` to control its clock from the outside

–

We will override the clock to be an immediate clock

–

And now it works like before

–

So our dependencies library provides the tools that allow you to skip the explicit initializer but also still allow you to override dependencies.

–

So, that sounds great for ergonomics, but what about safety?

–

Isn't this going to have problems where we accidentally use live dependencies in previews?

–

Well, the library has a tool for that too

–

Let's hop over to the location demo

–

Here we are using `@Dependency` twice, once for the analytics client and once for the location client

–

Down in the preview we will see we are construct the observable object without overriding dependencies

–

Yet when we run in the preview it is skipping the analytics event. And even the location features work

–

And if we run in the simulator it is definitely tracking live analytics

–

How does that work?

–

Well, when you register your dependencies with the library you get to specify the default, live version that is used when running in the simulator and on device

–

But you also get to specify the version of the dependency that is used in previews, and for that we have the analytics dependency use the loop version and the location client use the mock version.

–

That means we don't have to do anything special in our preview.

–

And if you add analytics to an existing feature, then it will automatically get the loop version in previews, meaning you never have to worry again about accidentally tracking live analytics in previews.

–

So, this is showing how we try to balance safety versus ergonomics in our library

–

It is ergonomic because we don't have to implement explicit initializers in our types that require all dependencies to be provided

–

But it is also ergonomic because each dependency has the ability to specify a safe version of it to use in previews

–

There's even a way to carve out a dependency to use for tests, but we won't have time to explore that.

–

Finally, let's show what it takes to add a new dependency to a type

–

Our dependency library comes with many controllable dependencies out of the box

–

Let's add `@Dependency(\.date)` to the observable object.

–

That's all it takes. There is no other code to fix.

This is just the beginning

So, I hope that this talk has helped you understand what dependencies are, why they can be so pernicious in a code base, and what you can do to start controlling them.

However, I must warn that although we have covered a lot in this talk, it's really only just the beginning. We've discussed enough for you to be dangerous, but there's is still so much to learn to truly master the topic.

Advanced topics

– Designing dependencies
– Overriding dependencies
– Testing

Here are just a few topics that are important to dive into to fully master dependencies.

First there's "designing dependencies". By this I mean the techniques that guide one in creating the interfaces and wrapper types that you put in front of a dependency in order to make it controllable. We showed off a few examples of this, such as the `Analytics` `LocationClient`, but we didn't actually build those types from scratch in this talk.

There is a lot to know about how those clients were designed, and how to make them maximally flexible for testing and previews. It is also important to know when it is appropriate to model something as a dependency and when it is not. Not *everything* should be a dependency, and on the flip side, some things should be dependencies that don't at first seem like dependencies.

Next there is the idea of overriding dependencies. We have already seen a glimpse of this for previews where we used the `withDependencies` tool to override dependencies just for the preview. But there are so many more cool things we can do.

For example, suppose you have an onboarding experience that teaches a person how to use a particular feature of your application. It would be great if you could allow them use the actual real feature but without worrying about accidentally write data to disk, or sending data to the server, or tracking analytics that aren't meant to be tracked. And so our library also provides tools that allow you to accomplish this. You can construct just one small part of your application that runs in an altered execution context, and it will not affect the rest of the application. That is incredibly powerful.

And finally there is testing. We didn't directly show how our dependencies library enables testing, but it isn't too different from what we did in previews. You use the `withDependencies` tool to set up your dependencies, and then execute your test in that controlled environment.

But the library goes a step beyond too. If you access a dependency in a testing context that you have *not* explicitly overridden, then you get a testing failure letting you know it is not OK to use live dependencies in tests. This prevents you from accidentally tracking analytics, writing data to disk, or any number of things while testing. Using live dependencies can bleed into other tests causing mystifying failures, or can cause your tests to become slow and flakey. And so it's good for the library to prevent you from doing this.

It also means that if you suddenly start to use a new dependency in a feature, then you will be notified when running your test suite that there is more to assert on. It's incredibly powerful.

github.com/pointfreeco/

swift-dependencies

And there is even more to discuss beyond everything discuss so far, and so I highly encourage you to check out our dependencies project. There is a ton of documentation, and we even have a full application built using the library. We took Apple's Scrumdinger app and rebuilt it.

We controlled every dependency in that application, and that unlocked all new powers from Xcode previews, and made testing very easy.

But also I want to highly encourage you to check out the other dependency libraries out there in the community. We know that ours is not going to be for everyone. We take a pretty drastic stance by building it on top of Swift's `@TaskLocal` machinery, and in some ways that severely limits it, but in other ways also gives it power.

You may not want those limitations and/or powers, and so it will be good to see what the other dependency libraries have to offer.

Thank you

Brandon Williams

– brandon@pointfree.co

– @mbrandonw

– www.pointfree.co