

Challenges of using Xcode at scale

Introduction

- Pedro Piñera
- Berlin-based
- Open Source and developer tooling passionate
- Creator of Tuist and co-founder and CEO of Tuist GmbH



Tuist

Xcode project management

(We fight Xcode intricacies, so that you don't have to)

For my parents: we help developers stay productive

Development stack

1. Concepts

Provide a **common language** across the platforms

2. Xcode projects and workspaces

Provide an API to **describe** the project

3. Xcode build system

Provide a tool to **compile** the project

4. Xcode

Provide a tool to **interact** with the project

Targets

Unit of encapsulation of various source files with a well-defined public interface

In the early days of Xcode

1 project 

1 app (macOS) 

1 target 

But the environment changed...

Targets

Unit of encapsulation of various source files with a well-defined public interface

In the early days of Xcode


1 project 

1 app (macOS) 

1 target 

These days...

1 workspace 

Multiple projects 

Multiple platforms (   )

Multiple products (app, extensions)

Multiple targets per project

Multiple external targets (packages)

Concepts, Xcode projects and workspaces, build system, and editor need to evolve too...

And they had to make great decisions quickly

Convenience vs. scalability

- **Convenience:**

- Sensible defaults
- Build-time resolution of implicitness

- **Scalability:**

- Explicitness
- Predictability

Challenges

1. Unmaintainable dependency graphs
2. Implicit dependencies
3. Unreliable SwiftUI previews
4. Slow compilation of Swift Macros
5. Non-optimizable workflows
6. Slow Swift Package Manager integration
7. Mergable mess

1. Unmaintainable dependency graphs

- New targets in the graph might have upstream effects (e.g. embedding into products)
- The graph is not visible in the UI
- The graph is implicitly codified into the project files

What can I do about it

- Make it explicit through tools like Tuist, SPM, or Bazel
- Keep graphs simple and shallow

2. Implicit dependencies

- Two types:
 1. Opted-in through the "Find implicit dependencies" scheme option
 2. Accidental due to shared built products directory
- Non-declared dependencies can resolve implicitly (fine at small scale)
- But can lead to build issues and headaches at scale

What can I do about it

- Disable "Find implicit dependencies from schemes"
- Use Tuist and enable "force explicit dependencies".

3. Unreliable SwiftUI previews

- SwiftUI previews might not work.
 - Even if the app compiles

What can I do about it

- Ensure scripts' input and output files are accurate
 - Or just avoid them
- Prefer dynamic products over static ones
- Use module-scoped example apps

4. Slow compilation of Swift Macros

- Swift Macros might have a deep dependency tree (mostly SwiftSyntax)
- Can add a lot of compilation time to clean builds

What can I do about it

- Precompile Swift Macros
 - And use build settings and phases to integrate them
- Avoid the current "Macro-all-the-things" trends
 - They have a cost not many people talk about
 - A strong dependency with a build system not designed for scale

5. Non-optimizable workflows

- Xcode's editor and build system are strongly coupled
- You can't optimize compilation steps

What can I do about it

- Replace the build system with Bazel (costly)
- Use Tuist and Tuist Cloud

6. Slow Swift Package Manager integration

- You have no control over when the SPM graph is resolved
- The resolution might be slow and happen at any time (e.g. when adding a file)
- SPM's design principles didn't account for project management usage
 - And that's becoming apparent as companies use it for that.
- The fact that you can use it for project management, doesn't mean it's designed for it.

Metric	.xcodeproj	SPM	Change
Open project (cold)	35s	1min 23s	+137%
Open project (warm)	26s	75s	+188%
Indexing	45 min	45 min	-
Create file	7s	16s	+128%
Delete file	~0	15s	+inf.
Move file	~0	14s	+inf.
Rename file	~0	15s	+inf.
Clean build	6.3 mins	16 mins	+154%
Incremental build (level 1)	1 min	1.9 mins	+90%
Incremental build (level 2)	4 mins	6.2 mins	+55%
Incremental build (level 3)	7.2 mins	11.77 mins	+63%

What can I do about it

- Use it only for integrating external dependencies
- Use Tuist's XcodeProj-based integration.

7. Mergable mess

- Slow compilation or slow build time
 - Dynamic frameworks are easier to integrate but impact build times negatively
 - Static frameworks are harder to integrate but increase build times
- Apple's response
 - Some configuration-based build-time magic
 - But...
 - The result is more unpredictable
 - The graph is harder to reason about

What can I do about it

- Use Tuist to statically switch between static and dynamic

Demo time

Summing it up

1. Broadly used != Being suitable for scale (e.g. Swift Macros)
2. Optimizations require explicitness. Apple leans on implicitness to provide convenience
3. More implicit configuration resolved at build time == More unpredictable behaviours
4. Swift Package Manager is a package manager, not a solution for project management at scale
5. Making Xcode work at scale with little cost is feasible, we do it at Tuist

고마워요