23 APRIL 2018 · MARC PALMER

A new idea: Feature Driven Development

What do I mean by Feature Driven Development?

In short:

> Including information about the Features and Actions of your application in the code itself, and using this information remove boilerplate code and ease debugging

I do not mean this definition of FDD which has some similar ideas but is a *little bit* OTT.

With this information in your code, you can add all kinds of powerful capabilities. Debugging and diagnosing failures becomes easier because you can clearly identify what the user was doing, and it brings more clarity to the internal architecture of your applications. This helps everyone who works on the product use the same terminology.

Why is this useful?

> *Customer:* "The app crashed. I'm going to give you a 1-star review"
>
> *Developer:* "I'm sorry about that. Please tell us what you were doing in the run up to it crashing so that we can reproduce it and help you"

This is a ridiculous situation, and yet this is reality for most of us.

I don't know about you but I've been programming for over 30 years and for a long time I have felt embarrassed by the fact almost nobody in our industry knows what the user is doing in their apps, especially at the point when something goes wrong. *Surely* we can do a lot better than a stack trace and an impenetrable mess of a 100MB debug log.

Some apps do use analytics, but even then you are unlikely to be logging every single action the user performs. There's a tendency to limit analytics to "commercially useful" events only. Development and QA builds of apps often do not log analytics events to avoid polluting the data. Analytics are also a bit creepy, especially through third party services.

I finally decided to do something about this for my apps when I realised the other benefits I would get if my code had better awareness of what the user was doing on a functional level. Like most developers I need to add feature flagging, analytics, in-app purchases, deep linking URLs, crash post-mortem logging, as well as iOS/macOS

Handoff and Siri Suggestions support. All of these can be largely automated if you have Action(s) as a first-class concept.

Feature Driven Development is the least ugly name I could think of [1] for designing apps around their features and the high level actions that users perform when interacting with these features. This idea is not specific to any one platform or a specific kind of app, but seems to lend itself particularly well to mobile and desktop apps. I have an implementation of these ideas for Apple platforms released as an Open Source project called Flint.

The core concepts

When creating or updating apps we typically think of adding new features, but you probably cannot look at your code and see any sign of this information from the design process — at most you might have a new "feature flag" that you added to selectively enable this new functionality.

To address this problem we need to make a Feature a first-class object in your code and have it describe whether or not it is enabled, as well as the actions available.

There are four basic building blocks in FDD: `Feature`, `Action`, `Input` and `Presenter`.

What do we mean by Features and Actions?

Features are the things people can do with your app, at whatever granularity is appropriate for you. Composing Features into hierarchies allows you to group concepts together and achieve reporting and "toggling" at a high level, e.g. disabling an entire drawing feature rather than every drawing tool feature within that feature.

Actions are the things that your program actually *does* for the user to use a feature. Invariably even for a trivial feature there is more than one Action required.

Features are what we design and sell. Actions are what we "perform" to make those features happen in code and track with FDD tools (analytics, logging and so on).

Examples:

- "Document Editing" is a high level feature. It may have many subfeatures, such as "Insert Picture", "Spell check".
- "Rename file" is a feature that would have at least two actions usually; "Show rename input screen" and "Perform file rename", and probably a "Cancel" or "Dismiss" action.
- "Post Tweet" is a feature that would include multiple actions such as "Show Compose UI", "Save As Draft", "Cancel", and "Post". It might have a subfeature "Attach Media" with actions "Show Media Library", "Attach Selected Media", "Dismiss".

There are no hard and fast rules, you do what works for you in terms of how you split up features. A general rule that seems to work is:

> If you need multiple show/dismiss actions, you probably need to split those out into sub-features

Putting it into practice

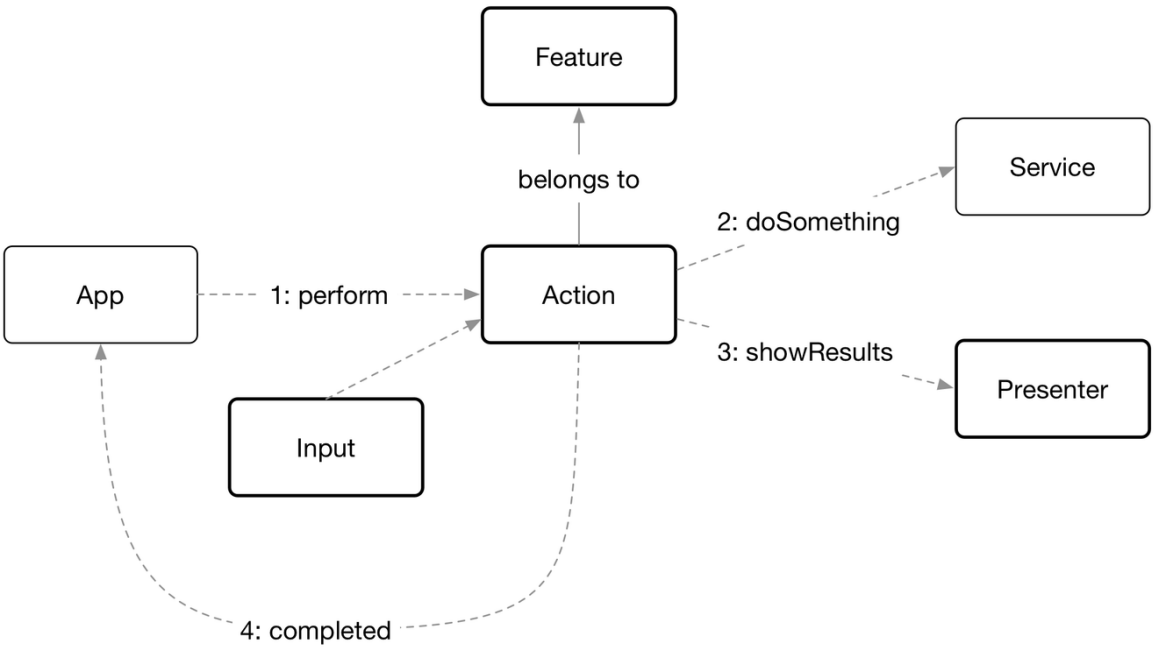
How you build these is up to you, but they are very simple concepts. When your app needs to invoke an action it must first verify that the action's feature is currently available, and then pass in an input and a presenter to the action itself. This feature availability step can be avoided if you know the feature is always available, but this depends on your implementation of FDD.

The definition of Features and Actions themselves is relatively trivial, and it will be covered briefly later. Performing actions is the most important idea, and here's a simple expression of this in pseudo-code:

```
if documentEditFeature is enable
  perform duplicateShapeAction w
    the current shape as input a
    the current view controller
```

There you have a conditional feature with a clear declaration of intent in the code to perform the action. It couldn't be simpler, conceptually.

- The `Input` is whatever type your action requires, usually collapsing several pieces of information into a single wrapper type e.g. a `struct` or `class`.
- The `Presenter` is any type you define that the action can use to provide output. Using interface or protocol types for these will allow you to decouple your actions from all UI and makes your app's primary actions very easy to unit test without bringing up a UI. Doing this will also make your actions cross-platform.



Performing an action

Programmers reading this will instantly see that the pseudo-code above lends itself very

well to automating tasks like reporting analytics, tracking what the user was doing for debugging, and choke points for features that users must pay extra to enjoy.

Beyond your development team, the product designers, visual designers, QA and marketing teams all need to communicate about these features. Too often there is a lack of consistent terminology, with issue trackers containing requests that are not expressed in terms of which user-facing feature is actually affected, ad hoc business-driven analytics changes and QA failure reports based on test scripts disconnected from the product design.

The utopian intention is that FDD helps your teams use the same terminology of Features and Actions through the whole "stack" required for the production of quality apps. When a discussion is being had about a bug relating to "opening files" what you should actually end up with is a bug that is clearly related to the "Open File Action" of the "Document Management Feature". Your test scripts would use the same language… and through FDD your logs will also use the same language. You start your product design process with Features and Actions in mind and it all flows from there.

As programmers we are used to working at a lower level than this — we try to modularise our code into reusable and decoupled components that meet the requirements in a way that we *hope* will be maintainable and minimises the opportunity

for bugs. This approach does not replace any of that, but enables a range of modern capabilities in a natural way, with just a little extra code, and most importantly, a small shift in mindset where you tease out the actions required to implement each feature in advance, and a presenter interface to abstract the UI.

While the FDD principle is simple, it can be quite tricky to put into practice in an elegant way depending on your chosen language's features and your desired balance of compile time vs. runtime safety.

I think it's well worth going for the extra compile-time safety if you can. You don't want your app to crash or freak out at run time because it called an action from a Feature that isn't enabled because someone forgot to include a check for its availability. Of course this is not a new problem, but FDD enables you to reduce the chances of it happening.

What FDD gives you

This is the really exciting part. Aside from having coherence with your product design terminology, if your code has an expression of the high level product design structure — the Features and Actions — this empowers you to more easily implement things that will be familiar to most developers of modern apps, including:

- Feature flagging (e.g. enabling new features only in certain internal builds,

or for certain users)
- Tracking Analytics for user actions
- In-App Purchases and Subscriptions that enable specific features
- Deep Linking from URLs into the app's content
- Automation workflows
- Operating system integrations (e.g. Handoff, Spotlight Search)
- Provision for A/B testing (supplying different variants of features to different user cohorts)

Beyond those, it adds exciting new possibilities for:

- Debug logging and crash reporting that maintains a clear link to user actions
- Logging of user action "Timelines", showing everything the user did in sequence
- Improved or even automated test plans
- User behaviour analysis (how users use your apps)
- Easier unit testing of features by decoupling UI from the code that performs actions
- Advanced features that are aware of each user's usage patterns. Imagine a persistent timeline of what the user has done in your app, being able to query that at runtime to suggest next actions or apply machine learning to it

Of course how much of this you get and how much effort is required depends on how you implement FDD. The aforementioned Flint framework for Apple platforms achieves all

of the above and you just write a few lines of Swift to describe and configure your Features and Actions.

Without Features and Actions expressed in your code it is much harder to do these things and often leads to tangled code or teams deciding not to bother with some of them at all.

What it looks like to apply FDD to an application

Let's use an imaginary app as an example: it lets you store favourite sequences of Emoji and recall them, so that they can be easily pasted into any app. It has cloud syncing and some preferences such as remembering your desired skin tone.

The list of features and actions might look like this:

- Feature: "Favourites"
  - Action: show
  - Action: dismiss
  - Action: createNew
  - Action: delete
  - Action: select
  - Action: save
- Feature: "Preferences"
  - Action: show
  - Action: dismiss
  - Action: changeSetting
- Feature: "Authentication"
  - Action: showLogin
  - Action: dismissLogin

- Action: attemptLogin
- Action: showSignup
- Action: dismissSignup
- Action: showTermsOfService
- Action: logout

You can see how this already gives a good clear picture of what the app does, as well as what UI wireframes are required and what features need testing and how to describe test scripts. It can even extend into how you categorise issues in your defect tracking systems, as well as making it much easier to see gaps in specifications.

Essentially you need to take everything that can happen as a result of user interaction and express it as an action on a feature. This might sound like a lot of work, but in fact it is a small amount of information. Prior to adding this information, a lot of these actions exist in your code but it is not obvious where they are or where the possible entry points lie.

What should a Feature look like in code?

How you define your features is entirely up to you and your environment. Perhaps it will be loosely typed, with dynamic class lookups based on a configuration file, or a class per feature that defines all the properties statically at compile time.

Either way, the properties that are generally required include:

id — A unique ID for the feature, so that it can be easily identified in logs

name — A name for the feature, for display to developers and QA, and maybe users depending on your app (In-App Purchases, user feature toggles)

description — A description of the feature, for display to developers and QA, and maybe users depending on your app

isAvailable — Whether or not the Feature is currently available

availability — Indication of how availability of this feature is determined – e.g. runtime check required, purchase required, based on a user preference or setting

actions — The list of available Action(s) in this feature

variation — Which A/B testing variant is currently in effect

The above is just a guide and in fact due to the static nature of the framework's design Flint has some divergence from the above, to leverage Swift language features for compile-time safety.

For example Flint separates features that are not always available into a separate type `ConditionalFeature`. It has availability properties only on that type, and this enables it to selectively supply functions that are selectively available on non-conditional or conditional features. This makes it impossible to perform actions of a

conditional feature without first checking if the feature is available.

Flint also does not have an `actions` property on `Feature` and instead uses a simple domain-specific-language to declare actions in such a way as to have compile-time type safety on Action input and presenter types when they are invoked.

What do Action(s) look like?

The `Action` part of FDD is event lighter than the `Feature`. You just need something that serves as a conduit between your UI and your components and services that provide the business logic.

Typically you'll need properties and functions such as:

id — A unique ID for the action, so that it can be easily identified in logs

name — A name for the action, for use in debug UIs and QA scripts

description — A description of the action, for use in debug UIs and QA scripts

inputType — The type of input the action requires

presenterType — The type of presenter the action requires

analyticsID — The analytics event ID to report to the analytics back end, optional as some actions do not require analytics events

perform(input, presenter, completion) — The function that actually performs the action

Action implementations are typically simple pieces of logic that stitch together whatever components you have in your code for business logic, taking an input and something that can present the result of the action, where necessary.

In some languages with support for generics you may define `inputType` and `presenterType` as types, as is the case in Flint so it is not strictly a property — in Swift it is expressed as an `associatedtype` in Swift terminology. This enables compile-time checks to ensure it is impossible to perform actions with the wrong input or presenter types, and gives better autocomplete in IDEs.

Often action implementations are only a handful of lines long, something like this in pseudo-code:

```
if input is valid then
  call service to do the work, p
  call presenter to show results
  call completion handler indica
else
  call completion handler indica
```

Contextual Logging, your new superpower

Let's be up front about this. Re-inventing logging seems like the most egregious kind of folly.

However, when you have formalised the dispatch of actions in your application, and you have contextual information about which feature the action belongs, clear inputs and the presenter used, you can do very powerful things. It would be wrong to not use this information.

At the very least your standard logging can now include the feature and action when writing log output. It could also include a textual description of the inputs. What's more, you can filter — at runtime or as part of a post-mortem investigation — all of the logging output by specific features.

This is far more powerful than classic logging scopes because the "topic path"[2] of each log entry is hierarchical. Features can be grouped and are hence hierarchical, and they related directly to user intentions, so retaining this information with log entries provides for hierarchical filtering[3]. Logging is normally scoped based on divisions between your code's subsystems, and filtering logs to tie up with user activity is often impossible.

Think about this for a moment. Using these techniques you can get logging *all through your app's code stack* that pertains to a specific action a user took. Not just a time-window spanning the start and end of the action.

If you have something like a network subsystem that makes multiple concurrent requests based on user actions and background activity, if you implement contextual logging you will be able to tease out the network logging that applies to a single specific action the user took, no matter how much time passed from start to end.

Here's an example of what logging may look like with information about the feature and action that triggered it. This is simplified output from a Flint demo app that has logging enabled to track the start and completion of all actions, and has an "ActivitiesFeature" that automatically publishes all qualifying successfully completed actions to the operating system for, among other things, search indexing:

```
16:56:02.687 DEBUG FlintDemo.App
16:56:02.704 DEBUG FlintDemo.App
16:56:07.067 DEBUG FlintDemo.App
16:56:07.076 DEBUG FlintDemo.App
16:56:08.192 DEBUG FlintDemo.App
16:56:08.199 DEBUG FlintDemo.App
16:56:08.261 DEBUG FlintCore.Fli
16:56:08.261 DEBUG FlintCore.Fli
16:56:08.261 DEBUG FlintCore.Fli
16:56:08.262 DEBUG FlintCore.Fli
```

Your actions will often call other subsystems that need to perform logging. By passing a context-specific logger instance to your subsystems, they will not lose the information about the current user activity.

Instead of having subsystems rely on their own singleton logger or log scope, you pass in the logger to use for requests. Language and/or platform tricks can make this less invasive through the use of thread-local variables or similar.

By way of example, Flint uses this new superpower to provide its own feature called "Focus" whereby the developer can call a function to add specific features to the "focus areas", and **all logging app-wide is silenced apart from logging that results from contextual loggers for features or sub-features that are currently focused**. This can be changed at runtime to streamline debugging and silence the deafening noise of the full spectrum of your app's logging.

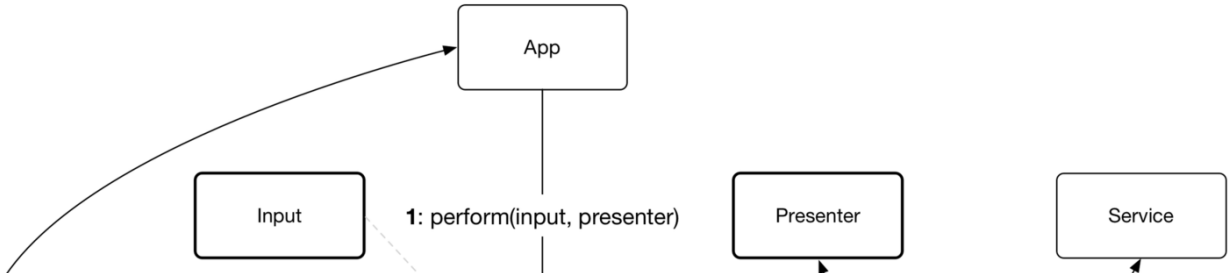
Related concepts for your own implementations

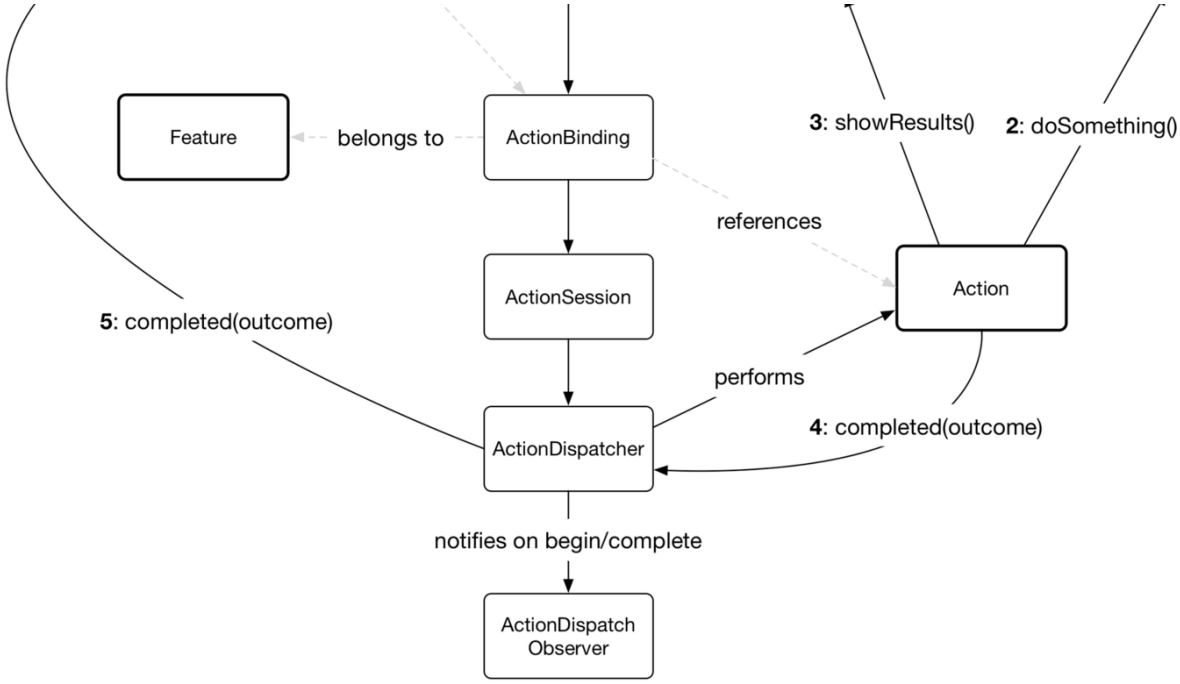
We've covered the basic building blocks: Features, Actions, Inputs and Presenters.

There are some other useful concepts that came up while building Flint, which you may find helpful if putting together your own FDD framework:

- Feature Group — having a special kind of Feature that can have sub-features enables you to build a hierarchy of functionality. This in means you can easily disable a whole graph of features if the user has not made a specific in-app purchase.

- Action Session — this is used to invoke actions and provide a grouping so they can be identified clearly in logs, as well as enforcing threading contracts. You can imagine a multi-window app would have one session per open document or window, and one or more background sessions.
- Action Dispatcher — sessions use a dispatcher to actually invoke the action. It provides a single place to observe all invocations across all sessions, for automatic logging and action-related side-effects.
- Action Context — these hold the contextual logger, input and convenience functions and are passed to actions when they are performed.
- Contextual Logger — this is a trivial logging interface that privately stores the feature and action information for each action dispatch, as well as textual representations of the action input and other useful information. Everything needed for the underlying logging implementation to implement Contextual Logging. It is passed to actions so they can perform context-specific logging and pass it forward to other subsystems.





Performing an action in Flint

Libraries and Frameworks you can use

If you have an implementation you wish to list here, let us know.

- Flint — An FDD framework for Apple platforms, written in Swift

Questions?

Feel free to e-mail me or join our Flint Slack which has an `#otherplatforms` channel for those who wish to discuss the non-Flint broader ideas expressed here.

UPDATED: Added definitions of Feature and Action, link to the OTT version of "FDD"

[1] : Yes, really. “Feature Oriented Development” is even worse

[2] : This borrows somewhat from various message queue or pubsub backend servers' concept of topics

[3] : Notably Apple recently overhauled their OSes’ logging APIs to introduce hierarchical context, but sadly this is not currently useful from Swift. So close and yet… failed to join this up to wider ideas of actions within apps!

NEXT