Swift Types & Semantics

or: How I Learned to Stop Worrying and Love the Compiler

Swift Types & Semantics

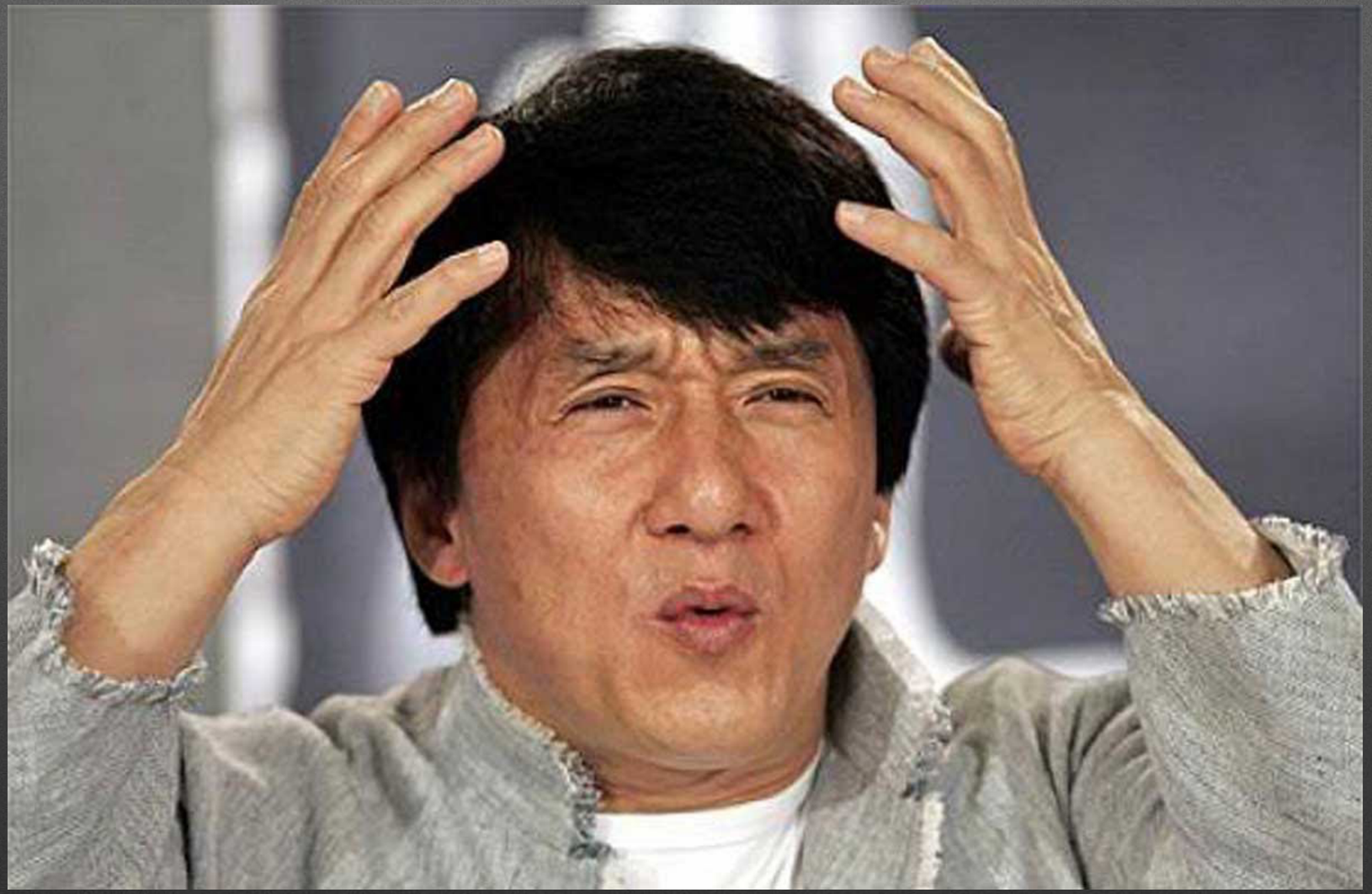
- Semantics
- Swift Tools for Semantics
- Use-cases
- Obvious limitations & downsides
- Interrogations

Semantics

Semantics is the linguistic and philosophical study of meaning, in language, **programming languages**, formal logics, and semiotics.

It is concerned with the relationship between **signifiers**—like words, phrases, signs, and symbols—and **what they stand for**, their denotation.

https://en.wikipedia.org/wiki/Semantics



Semantics in Swift: Clarity

Clarity at the point of use is your most important goal. Entities such as methods and properties are declared only once but used repeatedly. Design APIs to make those uses clear and concise. When evaluating a design, reading a declaration is seldom sufficient; always examine a use case to make sure it looks clear in context.

Clarity is more important than brevity. Although Swift code can be compact, it is a non-goal to enable the smallest possible code with the fewest characters. Brevity in Swift code, where it occurs, is a side-effect of the strong type system and features that naturally reduce boilerplate.

https://swift.org/documentation/api-design-guidelines/#fundamentals

Why semantics matter to me

- I need to understand your code
- I need you to understand my code
- I need to understand the code I’ve written in the past
- Code is not only for the computer but for humans too!

Swift Tools for Semantics

- Enums with associated values
  - Serves as an either-or-or-or…
- Generics and associated types
  - Allows various contexts of use for a single type
- Compiler-enforced type safety
  - Allows you to sit back and let the machine do the ugly checks for you

Let’s see some code!

Why go for semantic types?

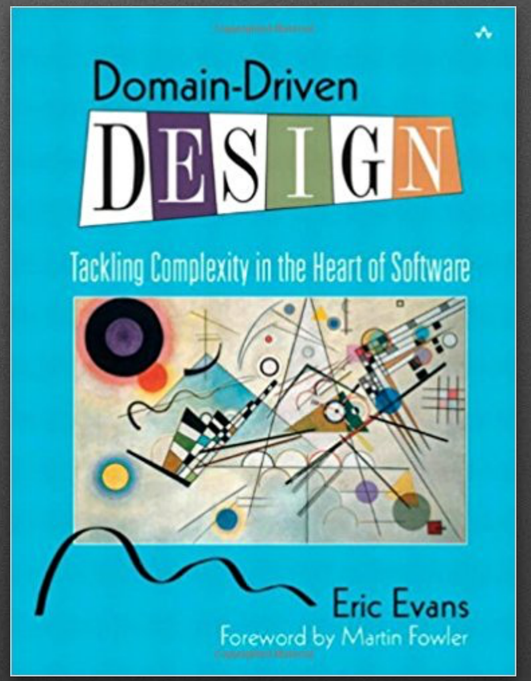
- **Better for humans!**
  - The code is the domain design
  - Moves away from “deducing the state from a bunch of state variables”
- **Better for quality!**
  - The compiler makes sure all cases are covered (unless you use default)
  - Only the useful state variables live within a given state “mode”
  - Eliminates the “impossible state” problem by design

Caveats

- Forces exhaustivity
- Extensibility can be tedious
- Enums are not externally extensible
- The compiler is smart but still slow (autocomplete lag, building times…)

Some Sources

Domain Modeling Made Functional - Scott Wlaschin



What about your experience?

In which contexts?
Model, View, other?…

View Controller
what about initial value?

Should we put reactive logic in didSet or KVO?

2-case Enum vs. Optional

Any other feedback?

Merci !

Twitter: @scalbatty
Github: scalbatty