Parnas Tables in Practice: From Specification to Swift

Let the Compiler Prove Your Software Quality

Mark H Pavlidis — tacow — 2025.11.12

Who's been pulled off feature work this week for a bug?

Who’s been pulled off feature work for a bug?

- Show of hands: "Who's been pulled off feature work this week for a bug?”
- How many times THIS WEEK has someone been interrupted by a production bug that could have been caught at compile time?
- DID YOU KNOW YOU COULD CATCH SOME CLASS AT COMPILE!?
- For those of you old enough to remember the transition from Objective-C (it will never die) to Swift — we learned type safety eliminated a class of bugs by using the type system
- **Today I'll show you how to eliminate entire classes of logic bugs**

Improving Software Quality

Using Formal Methods That Compilers Can Verify

Motivation — Who Benefits?

Formal Methods Approach

Using Parnas Tables (Tabular Notation)

From Specification to Swift

Demo — Compiler Verification

Summary

Motivation — Who Benefits?

Motivation — Who Benefits?

- Improving software quality means improving quality of life for:

- **Customers:** Less frustration using the product — less churn
- **Technical Support:** Fewer tickets to handle
- **Developer Developer Developers:** Less bugs — distractions, more time for features, deep focus to do hard tasks
- **QA:** Can find deeper issues before shipping
- **Managers:** Increased velocity - more output, fewer developers needed

The Limits of Testing
TDD or not TDD

- Testing proves presence of bugs, not absence
- Testing frameworks in Xcode have evolved significantly
    - Many still “hold it wrong” by not testing boundary cases
- TDD was developed as way to bridge the gap of incomplete specifications by non-domain experts
- Fundamental limitation: state space explosion 💥
- The opportunity: Compiler Verification
    - What if domain experts could read and validate the specification
    - What if the compiler could prove correctness at compile time?
    - Do we even need tests if the compiler can verify correctness?

Improving Software Quality

Practical Formal Methods Approach

- Decomposition principles (aka Information Hiding)
- Formal specifications using tabular notation
- State machines for modelling behaviour
- Mapping specifications directly to Swift code
- Compile-time verification of completeness and disjointness

Decomposition
Information Hiding

- Decompose by ***"likely changes"*** not data flow
- Based on 1972 paper by David L Parnas “On the Criteria To Be Used in Decomposing Systems into Modules”
- Information Hiding **is not** putting a Single Responsibility in a module
- Changes are localized to a single module
- Interfaces remain stable
- Hidden information can still be complex business logic

What Are Formal Methods?

Discrete Math!

- Define software systems precisely
- Reason about correctness
- Prove properties hold
- Formal Specification Tools: Z, VDM, TLA+, TLC
- Informal Comments (PRD) ➡ **Semi-Formal (Tables)** ➡ Formal (Proofs)

The Challenge

Traditional math specs are unreadable

Access Control Rules

$$\forall\, u \in \text{Users},\ r \in \text{Resources},\ a \in \text{Actions}:$$

$$\text{allowed}(u, r, a) \iff (\text{role}(u) = \text{admin}) \lor (\text{role}(u) = \text{user} \land \text{type}(r) \in \{\text{public}, \text{shared}\} \land a = \text{read}) \lor$$

$$(\text{role}(u) = \text{user} \land \text{owner}(r) = u \land a \in \{\text{read}, \text{write}, \text{delete}\}) \lor (\text{role}(u) = \text{guest} \land \text{type}(r) = \text{public} \land a = \text{read})$$

Problems

- Few people can read this fluently
- Hard to verify completeness (did we cover all cases?)
- Hard to verify disjointness (do cases overlap?)
- Doesn't translate obviously to code or domain expert language

The Solution

Parnas Tables (Tabular Notation)

User Role	Resource Type	Action	Owner?	Result
Admin	* (any, DC)	*	*	✓ Allow
User	Public	Read	*	✓ Allow
User	Shared	Read	*	✓ Allow
User	*	Read	Yes	✓ Allow
User	*	Write	Yes	✓ Allow
User	*	Delete	Yes	✓ Allow
User	Public	Write	No	✗ Deny
User	Public	Delete	No	✗ Deny
User	Shared	Write	No	✗ Deny
User	Shared	Delete	No	✗ Deny
User	Private	*	No	✗ Deny
Guest	Public	Read	*	✓ Allow
Guest	Private	Read	*	✗ Deny
Guest	*	Write	*	✗ Deny
Guest	*	Delete	*	✗ Deny

Why Completeness Matters

Missing Cases == Bugs

User Role	Resource Type	Action	Owner?	Result
Admin	*	*	*	✓ Allow
User	Public	Read	*	✓ Allow
Guest	Public	Read	*	✓ Allow

- Completeness = Every possible input combination has a defined output
- **Missing Cases:**
  - What if User tries to Write to Public resource?
  - What if User tries to access Shared resource?
  - What if Guest tries to Write?
- **Without completeness:** Runtime error, undefined behaviour, or worse security vulnerability!

Why Disjointness Matters

Non-deterministic == Bugs

User Role	Resource Type	Action	Owner?	Result
Admin	*	*	*	✓ Allow
*	Public	Write	*	✗ Deny
User	*	Read	Yes	✓ Allow

- Disjointness = No input should match multiple rules
- **Problem Case: (User, Public, Write, true)**
  - Matches row 2: ✗ Deny
  - Matches row 3: ✓ Allow
- **Without disjointness:** Non-deterministic behaviour, hard to reason about

Why Complete and Disjoint Matters

Where do the “requirements” come from

- On the previous slides are examples specifications embodied as tables
- Typically from a product manager in the form of PRD or user stories
  - Often they are incomplete and/or inconsistent
  - Rarely (if ever) define the boundary conditions precisely
- PM may not be a domain expert, developers typically are not:
  - So who’s going to define what the “truth” is for these cases?
  - Can the domain expert understand your code or test cases?

Fix This Table

A group exercise

Temperature (°C)	Humidity (%)	Action
< 0	*	Heat + Humidify
0-20	40-60	Heat
20-25	40-60	OFF
>25	>60	Cool + Dehumidify

Issues

1. **Incomplete:** What about 0-20°C with <40% or >60% humidity?
2. **Incomplete:** What about 20+°C with <40% or >60% humidity?
3. **Incomplete:** What about >25°C with ≤60% humidity?
4. **Non-disjoint:** Temperature boundaries (Is 20 in second or third? Is 25 in third or fourth?)

Fixed Table

Complete and Disjoint

Temperature (°C)	Humidity (%)	Action
< 0	*	Heat + Humidify
[0-20)	< 40	Heat + Humidify
[0-20)	[40, 60]	Heat
[0-20)	> 60	**Heat**
[20-25)	< 40	Humidify
[20-25)	[40, 60]	OFF
[20-25)	> 60	Dehumidify
≥ 25	≤ 60	Cool
≥ 25	> 60	Cool + Dehumidify

- Every temperature/humidity combination is covered (complete)
- No overlapping ranges (disjoint)
- Clear boundary definitions using [inclusive, exclusive)
- Realistic HVAC logic: cold air needs humidity, hot air gets dehumidified naturally by cooling
- Verifiable by a domain expert in a format they can understand

Mapping to Swift

Basic Structure

```swift
enum UserRole {
    case admin, user, guest
}

enum ResourceType {
    case publicResource, sharedResource, privateResource
}

enum Action {
    case read, write, delete
}

enum AccessResult {
    case allow, deny

    var symbol: String { ••• }
}

// Complete and exhaustive implementation
func checkAccess(
    role: UserRole,
    resource: ResourceType,
    action: Action,
    isOwner: Bool
) -> AccessResult {
    switch (role, resource, action, isOwner) { ••• }
}
```

Mapping to Swift

Pattern Matching

1:1 mapping from table to code

```
switch (role, resource, action, isOwner) {
// Admin can do anything
case (.admin, _, _, _):
    return .allow

// User access to public resources
case (.user, .publicResource, .read, _):
    return .allow

// User access to shared resources
case (.user, .sharedResource, .read, _):
    return .allow

// User access to owned resources (full control)
case (.user, _, .read, true):
    return .allow
case (.user, _, .write, true):
    return .allow
case (.user, _, .delete, true):
    return .allow

// Guest access
case (.guest, .publicResource, .read, _):
    return .allow

default:
    return .deny
```

Mapping to Swift

Compiler Enforces Completeness

```
) -> AccessResult {
    switch (role, resource, action, isOwner) {
    // Admin can do anything
    case (.admin, _, _, _):
        return .allow

    // All other cases denied (explicit enumeration for exhaustiveness)
    case (.user, .publicResource, .write, _),
         (.user, .publicResource, .delete, _),
         (.user, .sharedResource, .write, _),
         (.user, .sharedResource, .delete, _),
         (.user, .privateResource, .read, false),
         (.user, .privateResource, .write, false),
         (.user, .privateResource, .delete, false),
         (.guest, .publicResource, .write, _),
         (.guest, .publicResource, .delete, _),
         (.guest, .sharedResource, _, _),
         (.guest, .privateResource, _, _):
        return .deny
    }
```

Switch must be exhaustive

Demo

[TODO ad link to gh playground]

When to Use This Approach

In Summary

- Great For ✅:
    - Complex business rules with multiple conditions
    - Access control / permissions systems
    - State machines (workflow, protocols)
    - Configuration validation
    - Rule engines
    - Code that changes frequently (new cases added)

The Spectrum of Verification

In Summary

- Balanced between runtime tests and rigorous formal proofs
- Tabular notation + Swift's type system strikes a balance
- Practical for everyday development
- Compiler does the verification
- Catches bugs and missing specifications early
- Complete coverage!
- No theorem proving or graduate-level Discrete Math required! 🎉

Wrap Up
In Summary

With Parnas Tables (Tabular Notation) we have a technique that:

- **Domain experts, developers, and QA can read and understand !!!**
- Bridges domain experts and developers, developers and QA
- Developers can implement exactly in Swift (and other languages that support case tuples)
- Swift Compiler can verify `switch` completely
- Eliminates entire bug classes with completeness & disjointness
- Less testing, more confidence: proofs >> tests

Parnas Tables in Practice: From Specification to Swift

Let the Compiler Prove Your Software Quality

I’m available to coach your team on these methods

mhp@flixel.com

Mark H Pavlidis — tacow — 2025.11.12