Introducing Behavior as Data Pattern

When Data Becomes Behavior

Masoud Bahrami

mbahrami1990@gmail.com

Abstract

In my experience designing complex software systems, I frequently encountered challenges where changing a simple field value affected the overall behavior and logic of an object or module. Initially, these designs were data-driven, meaning those values were just stored as simple fields. Over time, as different behaviors were implemented based on those values, the code became scattered, complex, and difficult to maintain.

In this article, I introduce Behavior as Data Pattern. This pattern helps us decide when a simple field should be transformed into a behavioral model. I will also discuss key indicators and heuristics that help both developers and product owners recognize this need, accompanied by diverse real-world examples.

The Moment I Realized This Pattern Was Needed

In one project involving an online ordering system, during a meeting, the product manager said:

> *" When users want to settle their sales orders, they must select a settlement method. The available options include online payment, using their wallet balance, or a combination, wallet first, with the remaining amount paid online. Some sales are configured to be settled after delivery, while others require payment before delivery. Additionally, if a user chooses to settle the full amount using only their wallet, we offer a cashback reward: a percentage of the total order amount is returned to their wallet as a gift."*

A simple sentence but packed with information: the main status of the order, simultaneous temporary conditions, and even changes to order data. Initially, the status field was modeled as an *enum* with many scattered conditional checks around it. Initially, we modeled the order's status simply as an *enum*:

```
enum SettlementMethod {
  OnlinePayment,
  WalletOnly,
  WalletThenOnline,
  PreDelivery,
  PostDelivery,
  WalletWithCashback
}
```

The code was filled with conditionals like:

```
if (order.SettlementMethod == SettlementMethod.WalletThenOnline) {
  PayFromWallet(order);
  PayRemainingOnline(order);
} else if (order.SettlementMethod == SettlementMethod.OnlinePayment) {
  PayOnline(order);
}
 else if (order.SettlementMethod == SettlementMethod.WalletWithCashback) {
  PayFromWallet(order);
  ApplyCashback(order);
}

// More scattered conditionals handling various statuses...
```

As requirements grew, these scattered conditionals became unmanageable. Changes in the status field directly altered how the system behaved across multiple modules, leading to bugs and complexity.

These checks were scattered across payment services, billing modules, and even the order validation flow. As new combinations were added (like "*wallet, then online, but with delayed confirmation*"), this **enum-based** approach became fragile and hard to test.

I realized the Status field was no longer just data. It encoded complex behavior and temporary conditions simultaneously. This insight drove us to look for a design that encapsulated these behaviors more cleanly. In this case, I gradually realized the **SettlementMethod** field wasn't just a data flag anymore, it encoded complex, branching behaviors across the system. This was a classic case of data pretending to be behavior.

By moving from an *enum* to separate behavioral classes implementing shared interfaces, we achieved more maintainable and extensible code.

I gradually realized this data-centric model was insufficient because changing the status field directly influenced the system's behavior. I needed a design that cleanly encapsulated these behaviors in a maintainable and extensible way. This realization led me to **Behavior as Data Pattern**.

If you're curious about why mixing state and status in the enum caused issues in the first place, check out my State/Status Segregation Pattern. It explains how separating lifecycle state from conditional status leads to cleaner models.

Pattern Definition

The Behavior as Data pattern states:

If the value of a field affects the behavior of an object, then that field is not just data, it should be transformed into a behavioral model.

In other words, when fields such as *status*, *type*, or any other value directly determine the system's logic and operations, keeping them as primitive fields leads to complexity and maintainability problems. The solution is to introduce separate classes or interfaces modeling these distinct behaviors.

Indirection, Delegation, and the Philosophy Behind Behavior as Data

Let's start with a classic software principle knows as the theorem of software engineering:

"All problems in computer science can be solved by another level of indirection."
— David John Wheeler

Behavior as Data pattern follows this exact philosophy. Instead of hardcoding conditional logic directly inside the domain model, we introduce a structured and testable level of indirection, often in the form of delegation through interfaces or strategy objects.

Delegate + Indirection = Modular Behavior

This pattern leverages **delegation** to encapsulate behavior:

- Instead of **internal decision-making**, behavior is **delegated to external collaborators**
- This delegation is typically defined through **interfaces or polymorphic contracts**
- The added indirection gives us flexibility to **compose or configure behavior at runtime**

In essence, we **replace explicit logic with implicit wiring**, a powerful shift in how behavior is organized and extended.

It's Not Just About Ifs, It's About Behavioral Boundaries

Let's say it more clearly: **Behavior as Data** doesn't just move conditionals, it **challenges your modeling boundaries** and asks:

"Am I forcing fundamentally different behaviors into a single structure, violating cohesion and clarity?"

When the answer is yes, the problem isn't just conditional bloat, it's **a signal to refactor your design into behavior-centric models**.

Behavior as Data gives you a lens to recognize:

If a field's value determines complex and diverging behavior, then that field is not just data, it represents behavior and must be modeled as such.

Let's Look at a Few Examples

JournalEntryType = Sales | Purchase | Adjustment

In an accounting system:

- Sales creates **AR entries**, records revenue, and involves a customer.
- Purchase creates **AP entries**, involves a vendor, and records liabilities.
- Adjustment might be **internal only**, involving no third party.

These aren't just different *types*. They are different **accounting behaviors with different rules and interactions**.

Forcing them into one class means accumulating conditionals and tangled logic.

PaymentMethodType = Wallet | Bank | Installment | Combined

In a payment system:

- Wallet may need balance checks and locking funds.
- Bank may involve real-time gateway transactions.
- Installment needs schedules, contracts, and validation flows.
- Combined may mix multiple methods **atomically**.

These aren't just enum values. They are **distinct payment strategies** with different operational flows.

DocumentStatus = Draft | Submitted | Approved | Archived

In a document management system:

- Draft is editable.
- Submitted triggers workflows and approvals.

- Archived is locked and read-only.

These are not just states. They are **lifecycle stages**, with transitions, permissions, and context-specific behaviors.

Here, **State Pattern** becomes a concrete implementation of Behavior as Data.

Structure Over Scattering 

Behavior as Data moves conditional logics to **structured, explicit behavior modules**, where:

- Dependencies are better managed
- Behavior becomes **testable**, **composable**, and **extensible**
- The domain model transforms from "**a condition-checker**" to "**a behavior performer**"

This is exactly where **intentional design** starts to diverge from **accidental complexity**.

Behavior as Data is not just a pattern. It's a modeling insight. It helps you ask:

"Am I designing for what the system is — or just reacting to how it behaves now?"

When to Transform a Field into Behavior: Clues and Heuristics 

Determining whether a field should remain simple data or become behavior is not always obvious. Here are key heuristics to guide developers and product owners:

- **Does changing the value of this field alter the system's logic or behavior?** *If yes, consider behavioral modeling.*
- **Are there numerous scattered if/else or switch statements based on this field?** *A sign that behavior is entangled and needs refactoring.*
- **Is the set of possible field values likely to grow over time?** *Growing sets of values usually cause condition explosion.*

- **Is testing behavior complicated and tightly coupled to the field's value?** If tests require complex setup or many cases because of field values, consider behavioral modeling.
- **Does this field represent an important domain concept that triggers specific behaviors?** If it triggers domain-specific processes, notifications, or rules, model it as behavior.

Examples

1. Bank Account Currency

Before (Data-driven):

```
class BankAccount {
  String currency; // "USD", "EUR" // even if its type was an enum

  Money convertTo(String targetCurrency) {
    if (currency.equals("USD") && targetCurrency.equals("EUR")) { ... }
    else if (...) { ... }
  }
}
```

Problem: Currency conversion logic scattered and grows as currencies increase.

After (Behavior-driven):

```
interface Currency {
  Money convertTo(Money amount, Currency target);
}

class USD implements Currency { ... }
class EUR implements Currency { ... }

class BankAccount {
  Currency currency;

  Money convertTo(Currency target) {
    return currency.convertTo(this.balance, target);
```

```
  }
}
```

Now, currency is not just data but a behavioral model.

2. Tax Calculator

Before:

```
class Invoice {
  String country;

Decimal calculateTax() {
    if (country.equals("DE")) return amount * 0.19;
    if (country.equals("UK")) return amount * 0.2;
    ...
  }
}
```

After:

```
interface TaxPolicy {
  BigDecimal calculateTax(Decimal amount);
}

class GermanyTaxPolicy : TaxPolicy { ... }

class Invoice {
  TaxPolicy taxPolicy;

  Decimal calculateTax() {
    return taxPolicy.calculateTax(this.amount);
  }
}
```

Here, the country has become a behavioral model representing tax policy.

3. UI Button Style

Before:

```
<button class={isPrimary ? "primary" : "secondary"}>
```

After:

```
class Button {
  ButtonStyle style;

render() {
    return `<button class="${style.cssClass()}">`
  }
}
```

When a field influences rendering and behavior, it deserves behavioral modeling.

4. Airline Ticket Payment Methods

```
class PaymentMethod {
  string method; // "wallet", "bank", "installment", "combined"
}
```

Simple field is fine only if logic is trivial. When complex payment logic exists for each method:

```
interface PaymentStrategy {
  void pay(Order order);
}

class WalletPayment : PaymentStrategy { ... }
class BankPayment : PaymentStrategy { ... }
class InstallmentPayment : PaymentStrategy { ... }
class CombinedPayment : PaymentStrategy { ... }


class Order {
  PaymentStrategy paymentMethod;

  void pay() {
    paymentMethod.pay(this);
```

```
    }
}
```

Relation to State/Status Segregation Pattern (S3) 

This pattern, often discussed in complex systems and featured in my upcoming book *Language-Driven Design*, advocates separating status and behaviors tied to it. It complements Behavior as Data by providing a concrete approach to managing states and their associated behaviors, reducing conditional complexity.

Practical Advice for Teams

- Look for scattered conditionals during code reviews.
- Host modeling sessions with developers and product owners to clarify domain concepts.
- Document domain concepts explicitly to reflect their behavioral aspects.
- Use static analysis or complexity metrics to detect conditional explosion.

Refactoring Techniques: Applying Behavior as Data 

When you detect Behavior as Data, consider these steps to refactor:

1. Extract behavior logic into separate methods, classes or interfaces.
2. Replace conditional checks with polymorphism by delegating behavior to these components.
3. Swap the primitive field with an instance of the behavior model.
4. Write or update tests to cover the new behavioral abstractions.
5. Collaborate with product owners and domain experts to correctly model behaviors.

Behavior as Data's Refactoring Techniques by Example (you're right. It's a huge name for the section, but please be patient!!) 

Problem

In many systems, developers start by storing simple field values like *type*, *status*, or *mode* inside an object. Over time, these values begin influencing logic: conditional branches, business rules, rendering decisions, etc.

At first, it seems innocent.

But soon the code becomes tangled:

- Hard to add new types without breaking old ones.
- Behavior is scattered across conditionals.
- Testing becomes harder as logic depends on raw values.
- Developers duplicate logic or forget to handle all cases.

Let's see a concrete example.

Scenario: Journal Entry Posting (Simplified)

You're building a basic accounting module. A *JournalEntry* can be of different types, for example *Sales*, *Purchase*, or *Adjustment,* and each type needs to apply different logic to the accounting ledger.

Here's how it might start out:

```
enum JournalEntryType {
    Sales,
    Purchase,
    Adjustment
}

class JournalEntry {
    public JournalEntryType Type { get; set; }
    public decimal Amount { get; set; }

    public void PostTo(Ledger ledger) {
        switch (Type) {

            case JournalEntryType.Sales:
                ledger.Credit("AccountsReceivable", Amount);
                ledger.Debit("Revenue", Amount);
                break;
```

```
            case JournalEntryType.Purchase:
                ledger.Debit("AccountsPayable", Amount);
                ledger.Credit("Cash", Amount);
                break;

            case JournalEntryType.Adjustment:
                 ledger.Adjust("Suspense", Amount);
                break;
        }}}
```

This works, but it mixes data and behavior. What happens when you want to:

- Add *Refund* or *Transfer*?
- Change logic for *Purchase* without touching the rest?
- Reuse *Sales* behavior elsewhere.

You're forced to touch this method every time a classic code smell.

Refactoring to Behavior as Data

☑ Step 1: Extract the Behavior

Introduce a behavioral interface:

```
interface IPostingBehavior {
    void Post(Ledger ledger, decimal amount);
}

interface PostingBehavior {
        void post(Ledger ledger, BigDecimal amount);
}
```

☑ Step 2: Implement Specific Behaviors

```
class SalesPosting : IPostingBehavior {
    public void Post(Ledger ledger, decimal amount) {
        ledger.Credit("AccountsReceivable", amount);
        ledger.Debit("Revenue", amount);
    }
```

```
}

class PurchasePosting : IPostingBehavior {
    public void Post(Ledger ledger, decimal amount) {
        ledger.Debit("AccountsPayable", amount);
        ledger.Credit("Cash", amount);
    }
}

class AdjustmentPosting : IPostingBehavior {
    public void Post(Ledger ledger, decimal amount) {
        ledger.Adjust("Suspense", amount);
    }
}
```

☑ Step 3: Compose the Behavior into the Data

```
public class JournalEntry {
    public IPostingBehavior Behavior { get; set; }
    public decimal Amount { get; set; }
public void PostTo(Ledger ledger) {
        Behavior.Post(ledger, Amount);
    }}
```

Now the behavior is attached to the data in a clean, composable way.

☑ Step 4: Bridge the Enum to the Behavior (Optional)

If you still receive or store entry types as an enum, you can use a factory to map them:

```
static class PostingBehaviorFactory {
    public static IPostingBehavior From(JournalEntryType type) =>
        type switch {
            JournalEntryType.Sales => new SalesPosting(),
            JournalEntryType.Purchase => new PurchasePosting(),
            JournalEntryType.Adjustment => new AdjustmentPosting(),
            _ => throw new NotImplementedException()
        };}
```

Usage:

```
var entry = new JournalEntry(amount:1000,
    behavior: PostingBehaviorFactory.From(JournalEntryType.Sales));

entry.PostTo(ledger);
```

☑ Final Result: Cleaner & More Flexible

- Each behavior is isolated and testable.
- New behaviors don't affect existing code.
- No more nested if or switch.
- Data no longer drives behavior through raw values, it carries behavior.

Beyond Strategy: Other Ways to Implement Behavior as Data

While the Strategy pattern is often the go-to implementation for Behavior as Data, it's far from the only option. In many systems, especially those that are domain-heavy or involve varied lifecycles, other techniques can provide better alignment with the problem space or offer superior extensibility and clarity.

One of the most direct alternatives is subclassing. Instead of holding a field like Type or Category and branching logic based on it, you can represent each variant as a subclass. For instance, in a financial domain, invoices that behave differently depending on country-specific regulations can be represented by subclasses such as UKInvoice, GermanyInvoice, or USInvoice, all extending from a base Invoice class. This makes each invoice's behavior explicit and type-safe, without the need to rely on switch statements or delegated strategies.

Another powerful approach is the State pattern, which is especially useful when the field in question is a lifecycle state, such as Draft, Submitted, or Approved. In such cases, the pattern not only replaces branching logic with polymorphic behavior but also allows modeling of transitions between states. Each state is encapsulated as a separate class implementing the same interface, and the object itself maintains a reference to its current state, delegating behavior accordingly. This is ideal for workflows, document approval systems, or any process where the meaning and consequences of state transitions are important.

Functional languages or hybrid approaches often leverage patterns matching instead of full-blown polymorphism. For example, in C# 8 and beyond, switch expressions allow you to model behavior cleanly based on *enum* values or types. This doesn't provide the same level of encapsulation as OO

patterns, but it improves readability and can serve as an intermediate step toward full behavior modeling.

In enterprise systems, it's also common to externalize behavioral rules entirely using a rules engine. This is particularly effective when the behavior logic is owned by business stakeholders, changes frequently, or depends on a complex combination of conditions. Rather than writing conditional logic in code, you define rules declaratively, either through configuration or a DSL and the rules engine evaluates and applies them at runtime. This approach decouples domain logic from infrastructure and allows for more agile business updates.

Plugin registries or handler maps offer yet another dynamic solution. In such setups, each behavior (e.g., a payment processor or tax calculator) is registered in a registry or resolved through a dependency injection container. When the system encounters a field like *PaymentMethod* = *Bank*, it dynamically locates and executes the corresponding handler. This pattern scales well in modular architecture and is especially useful when the number of variants is not known at compile time.

The Visitor pattern is sometimes relevant too, particularly when you have a fixed set of variants but want to apply different behaviors over time, such as rendering, exporting, or validation. While it is less common for modeling a single behavioral field, it shines when behaviors must vary based on a domain type in multiple contexts.

Differentiation from Strategy and State Patterns

Behavior as Data is primarily a design heuristic rather than a strict structural pattern. Unlike well-defined patterns like Strategy or State, which prescribe clear structural solutions, this pattern guides *when* to apply such patterns. It prompts you to recognize when a data field should be replaced by a behavioral model, which you can then implement using Strategy, State, or other relevant patterns.

Beyond Strategy: Other Ways to Implement Behavior as Data

Examples

1. Subclassing

```
public abstract class Invoice {
    public abstract decimal CalculateTax();
}

public class UKInvoice : Invoice {
    public override decimal CalculateTax() => 0.20m;
```

```
}
```

2. State Pattern

```
public interface IFinancialTransactionState {
    void Post(FinancialTransaction transaction);
}
```

3. Pattern Matching (via switch)

```
public decimal Calculate(PaymentMethodType method) =>
    method switch {
        PaymentMethodType.Wallet => ...,
        _ => throw new NotImplementedException()
    };
```

4. Rules Engine

```
var policy = taxRules[region];

policy.Apply(order);
```

5. Plugin/Handler Registry

```
var handler = registry.Resolve(paymentMethod);

handler.Pay(order);
```

Summary Table

Implementation Approach	When to Use	Strengths
Strategy Pattern	Swappable behaviors in a clean OO model	Common and elegant in OOP
Subclasses	Entire object varies by behavior	Direct and clear
State Pattern	Lifecycle or workflow states	Handles transitions well
Pattern Matching	Discrete variants in FP-style code	Concise and readable
Rules Engine	Behavior is business-owned and dynamic	Externalizable, configurable
Plugin Registry	Extensible system with optional behaviors	Pluggable and dynamic
Visitor Pattern	Multiple behaviors on closed variants	Clean separation of concern

The Last Words

Recognizing when a field is not mere data, but a behavior trigger is crucial for designing flexible, testable, and maintainable software. The Behavior as Data pattern teaches us to separate behavior from passive data and transform fields that drive logic into distinct behavioral models. This reduces complexity, improves clarity, and prepares the system for future changes.

This pattern is a key part of the design philosophy explored in my upcoming book ***Language-Driven Design***.

Masoud Bahrami (https://MasoudBahrami.Com)