Co-funded by the European Union

Project Nr. 2022-1-FR01-KA220-HED-000086863

lightcode

Strengthening the Digital Transformation of Higher Education Through Low-Code

# 12. Data Validation and Consistency

Paris-Dauphine University – PSL

Funded by the European Union. Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Education and Culture Executive Agency (EACEA). Neither the European Union nor EACEA can be held responsible for them.

Erasmus+ Project
lightcode

Co-funded by the European Union

## PROJECT'S COORDINATOR

**Paris Dauphine University**
France

## PROJECT'S PARTNERS

**University of Macedonia** Greece

**University of Niš**
Serbia

**Karmic Software Research**
France

**REACH Innovation**
Austria

University of Zagreb
Faculty of Economics & Business

**University of Zagreb**
Croatia

**Symplexis**
Greece

*symplexis.eu*

## TABLE OF CONTENTS

OVERVIEW 4

STRUCTURE OF THE MODULE 8

DATA VALIDATION AND CONSISTENCY, STEP BY STEP 10

CONCLUSION 25

REFERENCES 27

# OVERVIEW

## Introduction

In the low-code world, **data drives everything**. Whether configuring an object, filling out a form, triggering an action, or displaying a chart—every meaningful interaction begins with data and depends on its reliability. But just because data exists doesn't mean it's correct.

This chapter introduces the foundational principles of **data validation and consistency** within the LightCode platform.

Unlike traditional coding environments, LightCode does not expose low-level type-checkers or formal schema languages. Instead, it offers a **pedagogical approach where validation emerges at multiple levels**—from data types and object structures, to method-level assertions and embedded test scenarios. Validation is not limited to static checks; it also evolves alongside system behavior and user interaction.

You'll begin by exploring the importance of basic data types—strings, numbers, booleans, lists—and how these are **implied by the UI components** you choose. You'll then build on prior chapters by connecting this idea to **object classes and the attributes** they define. From there, you'll examine how LightCode uses simple contracts (e.g. required fields, allowed values, range constraints) to prevent invalid configurations from entering your system.

But structure is only part of the story. Real-world consistency also involves **behavior and state**. This chapter introduces the concept of **progressive validation**, where rules vary depending on the state of an object. You'll explore how **methods with embedded assertions** enforce logic when it matters most, and how **test methods** turn expected behavior into executable documentation.

Rather than treating the database as the single source of truth, **LightCode models consistency at the application layer**—closer to users, logic, and learning. It aligns with modern practices like Domain-Driven Design and hexagonal architecture, where domain rules live near the behavior they govern.

Ultimately, data validation isn't just about avoiding mistakes—it's about building systems you can trust. And in LightCode, learners don't just follow rules; they **design them, test them, and evolve them**. That's what makes this chapter a crucial step in your low-code journey: it builds confidence through clarity, and teaches that **trust itself can be modeled**.

## Why is this Module Important?

Low-code platforms promise speed, accessibility, and creativity—but none of these matter if the **data is wrong**. Whether it's a payment mistakenly processed before an order is confirmed, a string used where a number is expected, or two related objects drifting out of sync, **small inconsistencies can lead to confusing bugs or even serious consequences** in real-world apps.

In LightCode, data validation is not a technical afterthought—it's a **pedagogical building block**. It helps learners understand the importance of **precision, reliability, and timing**, even in systems that seem simple. Validation isn't just about what's allowed—it's also about **when** and **why**.

Rather than enforcing a rigid schema from the start, LightCode adopts a **progressive model of consistency**. Rules can change as an object evolves, and validations can reflect the current state of the system. This reflects how most apps work in the real world: data comes in incomplete, users act in unpredictable ways, and software must guide rather than punish.

This module also makes a clear distinction between **consistency and persistence**. Validation in LightCode happens **at the domain level**—close to the logic and transitions users work with—rather than in the database schema. This mirrors modern practices where teams design domain models independently from storage, enabling iteration and autonomy.

Moreover, this chapter connects logic and data in a way that supports both educational and practical goals. It reinforces why certain inputs are invalid, how to declare expectations explicitly, and how to use validation as a **friendly feedback mechanism**—not just a blocker. This is especially relevant for future "citizen developers," who may not write traditional code but will still be responsible for designing robust, safe tools.

Finally, validation is a matter of trust. When users interact with your app, they expect it to behave sensibly. They assume that totals are correct, that buttons appear only when needed, and that information flows predictably. This chapter teaches how to uphold that trust—not by adding complexity, but by making good design habits **visible, teachable, and testable**.

## Learning Outcomes

By the end of this module, you will be able to:

- **Identify and describe** the basic data types used in LightCode (string, number, boolean, list, date), along with their default icons and typical interface behaviors.
- **Recognize** how UI widgets (e.g. checkboxes, picklists) are automatically generated from declared types and how they support implicit type-level validation in forms and blocks.

- **Define** structured data models using object classes and attributes, understanding how these structures drive interface generation and constrain input without custom logic.
- **Distinguish** between structural consistency (valid types and required presence) and behavioral consistency (valid transitions, logical flow, and object state coherence).
- **Apply** validation logic through state-dependent methods, using Python assert statements to enforce domain-specific rules at the right time.
- **Write** and interpret embedded test methods to simulate behaviors and verify system correctness before user interaction.
- **Understand** how validation is distinct from persistence, and how consistency can be modeled and tested without relying on database schemas.
- **Evaluate** and resolve validation issues by interpreting visual feedback, error messages, and structured test output in the LightCode platform.
- **Appreciate** the value of progressive validation—building consistency step by step—as a teachable and scalable design pattern for low-code apps.

## Prerequisites

This chapter builds on concepts introduced in earlier modules. To fully engage with the material, learners should have:

- **Completed Chapter 8: Let's Explore the LightCode Platform**, with a basic ability to navigate the interface, inspect objects, and test methods.

- **Completed Chapter 10: Data Management**, including the use of data grids, basic data types, and in-memory modifications.
- **Completed Chapter 11: Automating Processes**, with an understanding of object states, transitions, and the use of methods to encode logic.
- **Basic familiarity with YAML**, as used in LightCode to define objects, attributes, and simple test scenarios.
- A **functional understanding of UI blocks**, such as forms, picklists, and buttons, and how they reflect underlying data types.

No prior knowledge of formal data validation frameworks or programming languages is required. An open, inquisitive mindset—and a readiness to experiment and reflect—will be the most valuable tools in this chapter.

## STRUCTURE OF THE MODULE

This module is designed to take you from intuitive recognition of valid data (what looks right) to explicit, testable validation rules that support reliable system behavior.

Each section builds on the previous, combining visual modeling, state-based thinking, and embedded feedback mechanisms.

By the end of this module, you'll not only understand **what consistency is**—you'll be able to model it, test it, and evolve it over time.

Here's how the journey unfolds:

1. **Understanding Types and Implied Validation**. Learn how data types drive the structure and behavior of forms, and how validation can

emerge naturally from class attributes. Discover how the platform enforces constraints through UI components—without extra code.

2. **From Structure to Behavior: States, Picklists, and Available Actions**. Explore how objects evolve over time via states and transitions. Understand how picklists, buttons, and visibility are all controlled by declared logic—ensuring behavioral consistency.

3. **Validation Through Transitions: Assertions Inside Methods**. Go deeper into method logic using assertions to enforce contextual rules. See how validation happens during actions, not just before them—and how the system responds when something goes wrong.

4. **Validation Through Tests and Examples**. Shift from reactive to proactive. Use embedded test methods to simulate workflows, confirm behavior, and detect issues before deployment. Tests become living documentation of what your objects are supposed to do.

5. **Surfacing and Resolving Inconsistencies**. Learn how LightCode helps detect structural and syntactic issues—like bad references, misnamed fields, or broken dependencies—and guides you in resolving them through meaningful error messages and halting execution where needed.

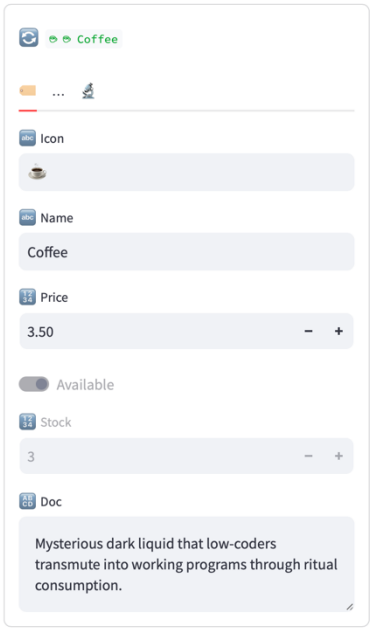
6. **Consistency Isn't Perfection: Staying Coherent While Moving Fast**. Wrap up with a modern take on validation. Explore why consistency doesn't mean rigidity, how it supports iteration and learning, and why persistence is no longer the only place where truth lives.

# DATA VALIDATION AND CONSISTENCY, STEP BY STEP

## 1. Understanding Types and Implied Validation

In LightCode, validation doesn't start with code—it starts with modeling. Each object is a self-contained contract, where data types and semantics guide both the user experience and the system's integrity. A single class declaration becomes a source of truth for form generation, field behavior, and interactive constraints.

Let's start with what the user sees: a clean, expressive form showing a catalog item—**Coffee**—ready to be selected, priced, and maybe… consumed.




At first glance, the form looks simple. But every field, every icon, and every visual constraint is derived from one thing: **the data model**. This form was not hand-coded.

It emerged automatically from the following YAML declaration.

```yaml
attributes:
  - !Attribute {name: name, type: str, icon: "🔤", default: Unknown}
  - !Attribute {name: price, type: float, default: 0.0}
  - !Attribute {name: available, type: bool, icon: "🔘", default: True, read_only: True}
  - !Attribute {name: stock, type: int, icon: "📦", default: 3, read_only: True}
```

And here's what it enforces—not through programming, but by structure alone:

| Attribute | Type | UI Element | Constraint |
|---|---|---|---|
| **icon** | str | 🔤 Text input | Emoji. Inherited |
| **name** | str | 🔤 Text input | Editable, initialized to default if empty |
| **price** | float | 🔢 Number input | Enforces numeric, decimal allowed |
| **available** | bool | Boolean | Read-only, reflects backend logic |
| **stock** | int | Numeric badge | Read-only, no user interaction |
| **doc** | str | 🔠 Multiline text field | Editable but semantically informative. Inherited |

Everything here is **implied validation**. The user cannot type "banana" into price because the field only accepts floats. They cannot toggle available because it's read-only. Even the stock and doc fields signal clearly what is editable, what is fixed, and what is informational.

There are no validators. No conditionals. No user-defined logic. Instead, **the form expresses the model**.

And the doc string? It's not just a joke:

*Mysterious dark liquid that low-coders transmute into working programs through ritual consumption.*

Even documentation is data. It's stored, structured, and surfaced automatically to the user interface.

This is what we mean by **data-driven design** or **model-driven design**. When the structure is sound, the interface is intuitive—and consistency emerges by default, not by enforcement.

Now, since the model is **trapped between two worlds**—on one side, end-user graphic conventions, and on the other, Python syntax—**LightCode performs smooth translations** to keep things natural.

For example, model attributes and method names are written in **snake_case**, as per Python conventions. But when they appear in the user interface, they are automatically translated: underscores become spaces, and names are rendered in a human-readable way. place_order becomes "Place order". order_id appears as "Order id".

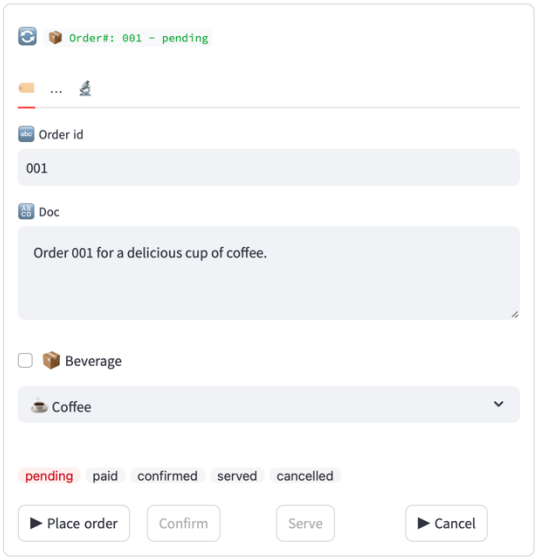
This lets designers and builders **speak Python to the system** and **natural language to the user**—without duplicating effort or writing glue code. The semantics remain direct and expressive, while the burden to customize every label, button, or action is eliminated.

It's another layer of **implicit consistency**: one definition, interpreted appropriately for multiple roles—human, machine, and interface. Experts call it the single source of truth. And now you can not only understand them but also speak their language.

> *Did you notice any inconsistencies with the image? We intentionally inserted easter eggs. Do you see discrepancies between icons and types? How can you explain and fix them? If you don't yet, stay tuned!*

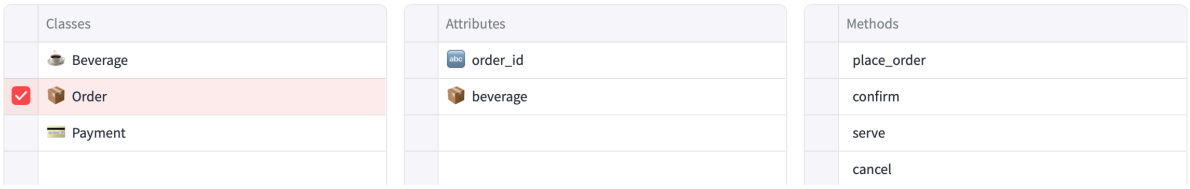
In the next section, we'll see how this model not only produces a static form, but also **shapes interactions** and acts as a semantic specification for downstream processes like ordering, validating, or automating delivery.

## 2. From Structure to Behavior: States, Picklists, and Available Actions

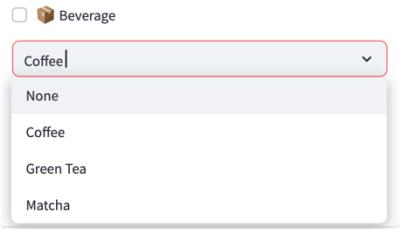



While simple objects define what can be stored and shown, other objects define **what can happen**. These are the active objects—those that carry both *data* and *behavior*. In LightCode, behavior emerges from declared **states** and **methods**, with validation built into the way these transitions are defined.

Let's take the example of an Order object. It carries a reference to a Beverage, a status, and a few possible actions. Nothing in this object was manually wired to buttons or logic gates. Instead, it was **declared once** and interpreted by the platform to generate both the interface and its behavioral constraints. This matches exactly what's described in the model—declared once, reflected everywhere:




It won't surprise you either to see beverage is a pick list for end users, offering all available beverages as a choice, including None (the python convention for no value or no choice in our case).

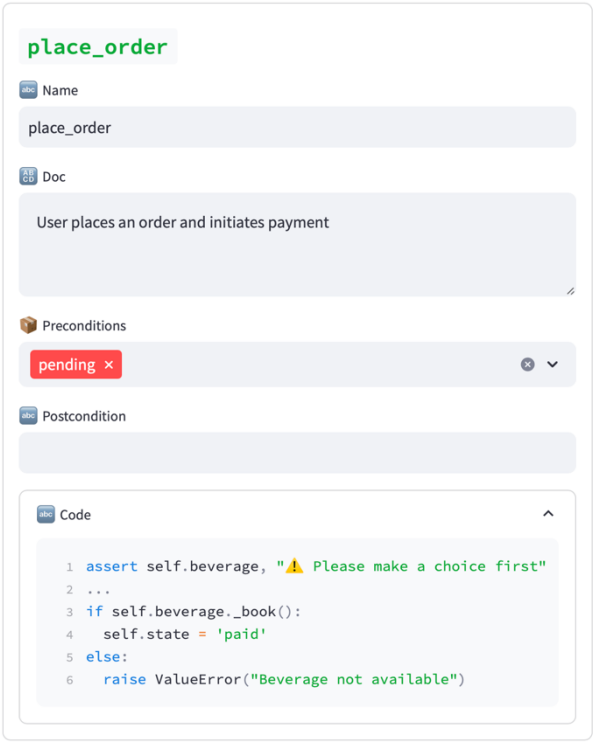



This sets **consistency** between user interface, model and executable code, so builders like you won't have to manage it over and over again through multiple iterations real life problems are nowadays solved: agile.

## 3. Validation Through Transitions: Assertions Inside Methods

Not all consistency can be encoded in the form. Some constraints are about behavior—things that must be true before a transition can happen. In LightCode, we don't handle this with if-else chains or popup logic scattered across the UI. We handle it **where it belongs**: inside the method.

Take this example: a user wants to place an order. But should the system proceed if no beverage is selected? Of course not. That would make no sense. And it shouldn't be up to every interface or workflow to remember that rule.




Here's how the method is defined.

Let's unpack what happens here.

- The **assertion** on line 1 checks that a beverage has been selected. If not, the method exits early with a clear, readable message.
- The next check (line 3) calls a method on the beverage itself—_book()—which likely handles internal availability logic. If it returns True, the order is moved to the paid state.
- If the booking fails, we raise a ValueError, stopping the process and surfacing a message like "Beverage not available".

This method does three things at once:

1. **Guards the transition** with a contextual rule.

2. **Delegates responsibility** to the linked object (beverage).

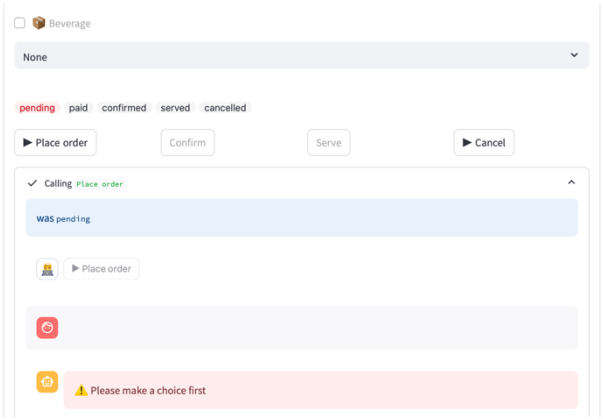
3. **Maintains consistency** by preventing illegal state changes.

Crucially, these checks are not global rules. They're **local, scoped to the transition itself**, and they only apply when the user clicks "Place order". No need to write the same validation in five different places.

And because the method declares its precondition (pending), the LightCode interface **only shows the action** when it's valid to use. If the order is already paid or cancelled, the button disappears or is disabled. No guesswork. No exceptions leaking through.

This is validation done right: lightweight, expressive, and safely embedded where it matters.

When something goes wrong—like a user clicking "Place order" without selecting a beverage—LightCode does not silently fail, nor does it crash. Instead, it gives **a transparent trace of what happened**, where it stopped, and *why*.




Here's what happens:

- The system logs the attempted method call (Place order),
- Shows the current state (pending),
- Identifies the actor (the user or test),
- And then halts with a clear assertion error: ⚠ *Please make a choice first*

This trace is not hidden in logs. It's visible directly in the interface, so both end-users and builders can understand what rule was violated—without needing a debugger.

This is where LightCode stands out: **validation isn't silent, abstract, or buried—it's conversational**. The platform doesn't just enforce consistency—it explains it, like a good teammate would.

And because the method doesn't proceed, no side effects occur. The state doesn't change. The beverage isn't booked. The order remains pending, safely awaiting a valid choice.

In LightCode, **validation is a dialogue between model and user**, where assertions act as checkpoints—and their failure is not a crash, but a teaching moment.

In the next section, we'll go further by turning these expectations into **testable examples**. Instead of waiting for someone to click the wrong button, we'll write tests that **simulate behaviors** and verify the system stays consistent—before anything goes live.

> 💡 *In standard Python, assert statements can be disabled with optimization flags (e.g. python -O). But in LightCode, they are **never skipped**. We treat them as **first-class validation expressions**, not optional debug hints. This is both a technical design and a philosophical nod to **Bertrand Meyer**, who introduced the idea of "Design by Contract"—where preconditions, postconditions, and invariants are core parts of the system's correctness.*

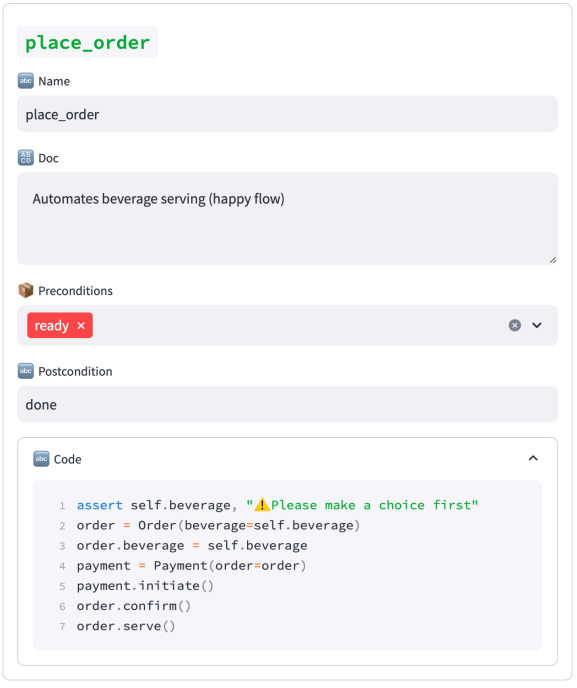
## 4. Validation Through Tests and Examples

So far, we've discussed validation that happens at runtime—when users interact with the UI or invoke methods manually. But what if you want to **document and verify the expected behavior ahead of time**, before any user touches the system?

LightCode supports this by letting you define test-like behaviors using **embedded test methods**, such as _test_*, written directly in Python.

These are not separate files or external frameworks—they're part of the model. They're visible, versioned, and executable like everything else.

Let's take the example from the **Assistant** class and see what happens when executing embedded tests.





What we see here is a full test cycle that runs directly within the platform:

- First, LightCode checks if class names are not python keywords. This is an additional kind of consistency that avoids name collisions.

- The embedded _test_place_order method creates a fresh Assistant object (called fixture), assigns it a Beverage, and runs the full flow, including calling place_order.
- Assertions inside the test check that the state has transitioned to 'done' and that the beverage stock decreased by one.

The platform's test runner provides a structured, readable output:

- Module and class names are clearly shown.
- Each test method's result is displayed individually.
- Assertion results are color-coded with ✅ or ⚠️ depending on success.

No IDE integration is required. No custom tooling.

Just embedded, declarative tests that double as **living documentation** of your expected behavior.

More importantly:

- If a transition fails due to a missing precondition, you'll see it immediately.
- If an assumption about state or resource usage doesn't hold, you can revise it early—before exposing it to end users.

These tests are not theoretical—they're **the first line of confidence** that your models **do what they say, and say what they do**.

> 💡 *Please not the test features depend on the plugins are installed on your server. Please check with your instructor about variants and options.*

In the next section, we'll push further: what happens when **validation fails silently?** When objects drift apart? When one model evolves, but another still depends on its former structure?

We'll explore how LightCode helps you **surface and resolve inconsistencies**—across classes, transitions, and imported modules.

## 5. Surfacing and Resolving Inconsistencies

Validation isn't only about business logic. Sometimes, what breaks your system is much simpler:

- A forgotten comma.
- A mistyped field name.
- A circular reference.
- An outdated dependency.

LightCode helps detect these issues immediately, by surfacing inconsistencies that could otherwise remain hidden in larger systems. Whether it's a typo in a method, a broken import, or a wrong attribute name, the platform **stops execution and shows you exactly where the problem is**.

Take the following example from the Assistant class. A test method named _test_more includes a basic placeholder:

assert True, "⚠️ Test method to be customized ...

But something is off—an unclosed string literal. When the module is run in test mode, here's what happens:


Instead of guessing what went wrong, you're told:

- The exact file (swift_sip),
- The class (Assistant),
- The method (_test_more),
- And the precise issue:

SyntaxError: unterminated string literal (detected at line 1, offset 14)

Even better, **the system refuses to run the rest of the tests**, reminding you that the module must be valid before behavior can be trusted.

This is LightCode's validation model at its broadest:

- **Runtime behavior** is constrained by assertions and transitions.
- **Static correctness** is enforced by YAML structure and Python syntax.
- **Interdependency consistency** is checked at the module level—no unlinked references, no missing imports, no silent failures.

As a result:

- Builders get early feedback.
- Teachers get teachable moments.
- And teams get a shared understanding that correctness includes *everything*, from logic to layout to lifecycle.

In the final section, we'll tie it all together: what does consistency actually *mean* in LightCode—and how do we balance it with flexibility, iteration, and continuous improvement?

## 6. Consistency Isn't Perfection: Staying Coherent While Moving Fast

In the world of low-code, perfection is not the goal—**progress with coherence is**.

LightCode doesn't enforce correctness as an absolute law. It promotes **structural and behavioral consistency** through:

- Data types that shape what users can enter,
- Methods that declare what transitions are allowed,
- Assertions that describe what must be true *right now*,
- And test methods that ensure those truths hold up over time.

But LightCode also accepts the reality of change:

- Maybe the structure evolves.
- Maybe states shift.
- Maybe you fix a typo and forget a test.

- Maybe you rename an attribute in a shared module, and someone else is still using the old one.

And that's okay. The platform is designed not just to enforce, but to **help you detect, adapt, and recover**:

- Validation catches typos and logic errors early.
- Interface constraints guide users instead of blocking them.
- Precondition mismatches, assertion failures, and syntax errors are shown where they happen, in plain language.

### A Note for the Academically Inclined

In classical database theory, **validation is absolute**: a field is required, or it's not. A constraint holds, or the record is rejected.

But low-code operates in a different environment—one where **learning happens through user behavior**, not just static schemas.

That's why LightCode introduces **progressive consistency**, using **state machines** and **method-level assertions** to validate data *when it matters*, not all at once. Users can start small, evolve their inputs, and let the system adapt around them.

This makes forms more accessible, workflows more forgiving, and user behavior part of the learning cycle.

## And What About Persistence?

A classic objection: if validation isn't tightly coupled to storage, **how do we ensure long-term data integrity?**

LightCode follows a **modern separation of concerns** inspired by Domain-Driven Design (DDD):

- **Consistency** is enforced at the domain level—inside entities and aggregates—using the kind of declarative logic we've explored.
- **Persistence**, by contrast, is considered **infrastructure**, not the primary source of truth.

In distributed systems and hexagonal architectures, persistence is often *delayed, partial, or decoupled*. Different modules may use different databases. Teams evolve domains independently. The app logic lives outside the database—not in triggers, constraints, or stored procedures.

LightCode aligns with this view:

- It lets you test behavior *before* wiring up storage.
- It lets teams iterate on logic *without breaking contracts*.
- It models the world—not the table.

**In short**: Validation is about what makes sense. Persistence is about where and when you keep it. They're different—and keeping them distinct makes your system more resilient, testable, and adaptable.

## A Teaching Philosophy, Not Just a Technical One

Ultimately, LightCode's approach to validation is a pedagogical choice.

It's designed for **learners who aren't full-time developers**, for **teams that grow by iteration**, and for **systems that are living things, not frozen diagrams**.

Consistency, here, is not perfection. It's clarity. It's structure with room to breathe. And above all, it's something you can test, teach, and trust.

## CONCLUSION

This chapter began with a simple observation: **everything is data**. Whether visible in a form or hidden in a method call, data is at the heart of every object, every transition, and every outcome in LightCode. But correctness isn't just about storing data—it's about **ensuring that data remains meaningful, structured, and coherent** as it flows through the system.

We explored how LightCode supports validation not through static enforcement, but through **progressive layers of consistency**:

- Starting with **data types**, which shape how users interact with fields,
- Then through **class structures**, which define not only what data exists, but what is editable, what is computed, and what is inherited,
- Through **states and transitions**, which organize behavior over time,
- Through **assertions**, which enforce domain logic exactly when and where it matters,
- And finally, through **embedded test methods**, which turn every object into its own living, verifiable specification.

Along the way, we challenged classical assumptions from database theory and instead aligned with modern practices from **Domain-Driven Design**, **hexagonal architecture**, and **agile development**. In LightCode, validation is not a rigid checklist—it's a dynamic, contextual conversation between builder, model, and user.

We also saw that **persistence is not the primary concern** in this architecture. The domain model—not the database schema—defines what is valid. This empowers teams to iterate, learners to explore, and systems to remain adaptable as complexity grows.

In the end, LightCode's approach to validation is neither defensive nor dogmatic. It is:

- **Declarative**, but not rigid.
- **Opinionated**, but progressive.
- **Teachable**, testable, and truly low-code.

Consistency isn't something you enforce once. It's something you evolve.

And this chapter has shown that in LightCode, **you don't need to choose between structure and flexibility**. You can have both—if you design it right.

# REFERENCES


- **Evans, E. (2003). Domain-Driven Design: Tackling Complexity in the Heart of Software**

  *A foundational text introducing the idea that business rules and data validation should live in the domain model, not the database. Core inspiration for using aggregates and progressive consistency.*

- **Fowler, M. (2002). Patterns of Enterprise Application Architecture**

  *Chapter 3 ("Domain Logic Patterns") is especially useful to understand why separating validation from persistence improves flexibility and testability.*

- **Vernon, V. (2013). Implementing Domain-Driven Design**

  *A practical follow-up to Evans' work, with examples of how aggregates encapsulate validation logic in real applications. Highlights the value of state-aware rules.*

- **The Python Language Reference.** https://docs.python.org/3/reference/

  *Particularly sections on assert and exceptions. Useful for understanding how LightCode test methods and method-level guards align with native Python behavior.*

- **The Agile Manifesto** (2001). https://agilemanifesto.org

  *Not a technical source, but a cultural one. Validates the LightCode philosophy that software evolves through feedback, and that validation is often better learned through iteration than enforced upfront.*