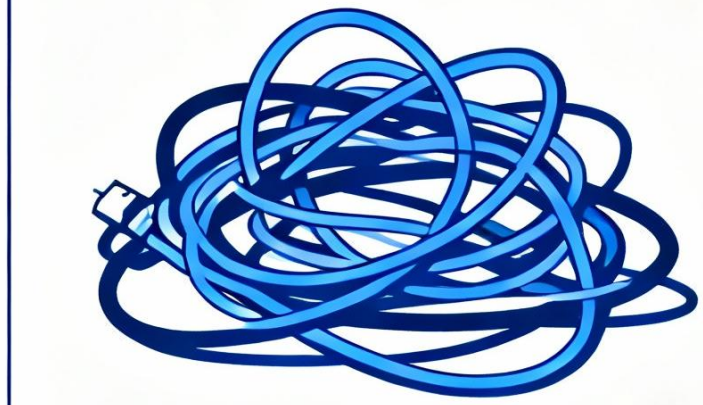

Course: Object Oriented Programming

# Software that Fits the World

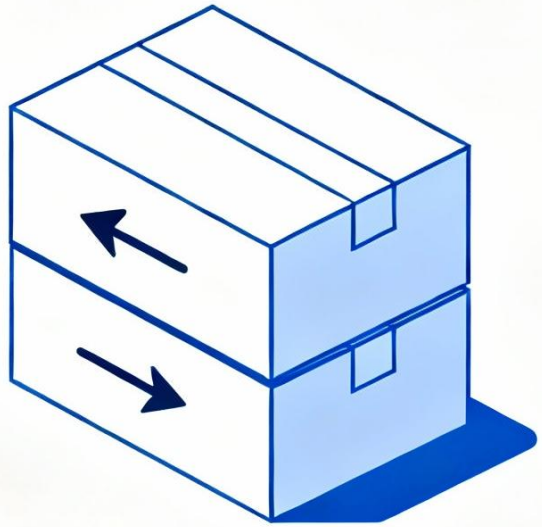
## An Introduction to Object Oriented Programming

Evis Plaku

# Same problem: a new paradigm



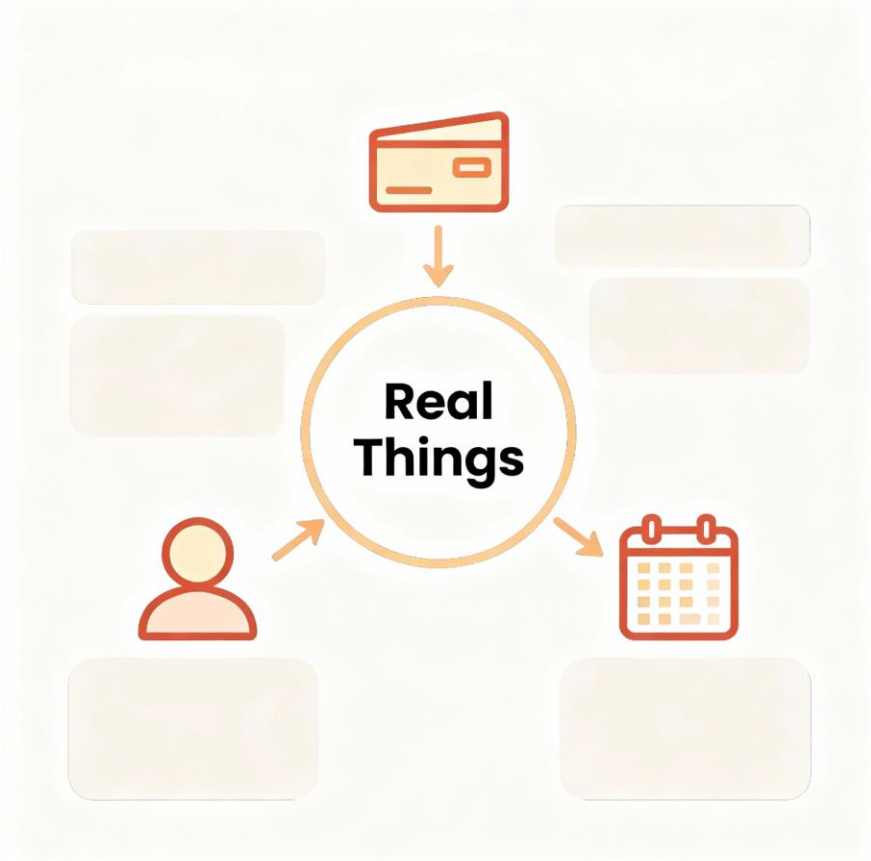
Procedural: steps everywhere



OOP: parts talk

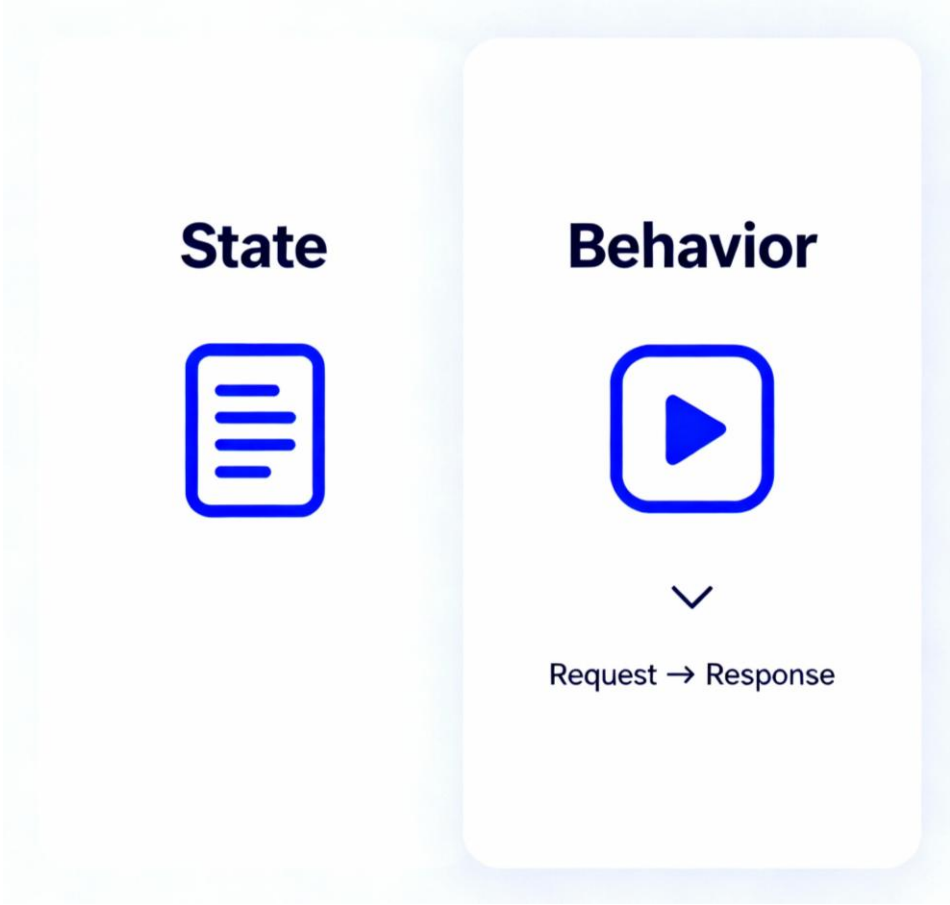
# Learn to model real problems with objects for clear, change-ready code

- We organize code around real things and their simple actions
- This helps manage complexity as features grow over the semester
- Expect practical ideas, plain language, and steady, hands-on progress



# It reduces complexity by grouping related data and actions together

- Objects reflect how people think: things with state and behavior
- Clear boundaries mean changes spread less and break fewer parts
- Shared patterns let teams reuse parts instead of rewriting code



# Turn everyday nouns and verbs into simple software building blocks

- Identify key things, their facts, and the actions they perform


- Keep each part focused on one role for clarity and change
- Let parts interact through small requests, like people cooperating

# Principles help with clear design

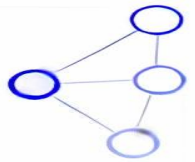
- Encapsulation hides internals, exposing safe, simple ways to interact
- Abstraction shows what matters, skipping distracting, low-level details
- Inheritance and polymorphism reduce duplication and simplify extensions

Abstraction

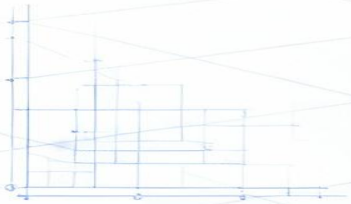
Encapsulation

Inheritance

Polymorphism



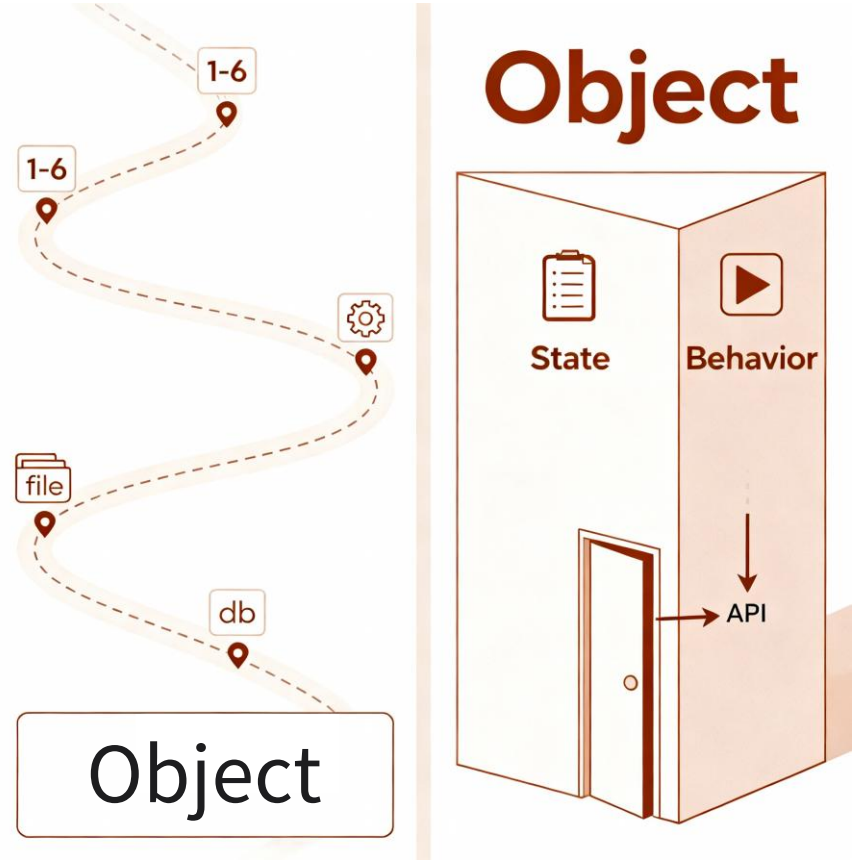
# How to think in OOP



UNIVERSITETI METROPOLITAN TIRANA

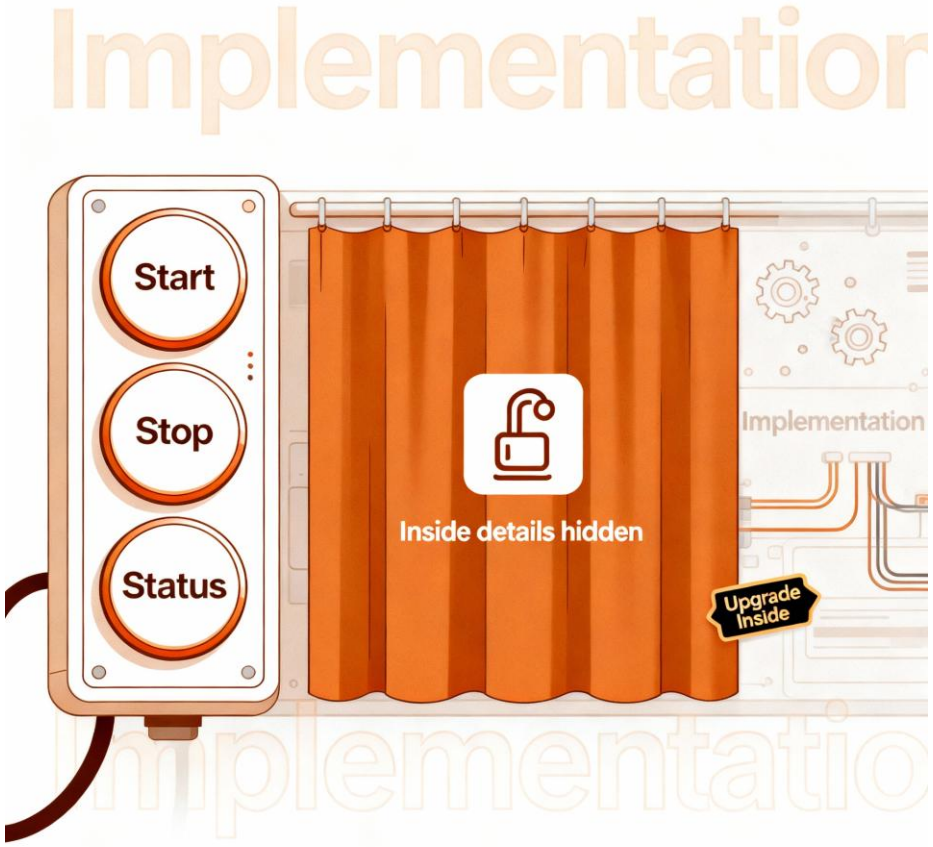
# Procedural suits small scripts; OOP scales better for complex systems

- Procedural code scatters data and logic across many separate steps
- OOP keeps related data and behavior in one clear place
- This reduces surprises when systems grow and requirements change



# Protect state by controlling how other parts can change it

- Hide details; present a small surface for safe interactions
- Validate inputs at the boundary to keep rules always true
- When internals change, outside code stays calm and stable

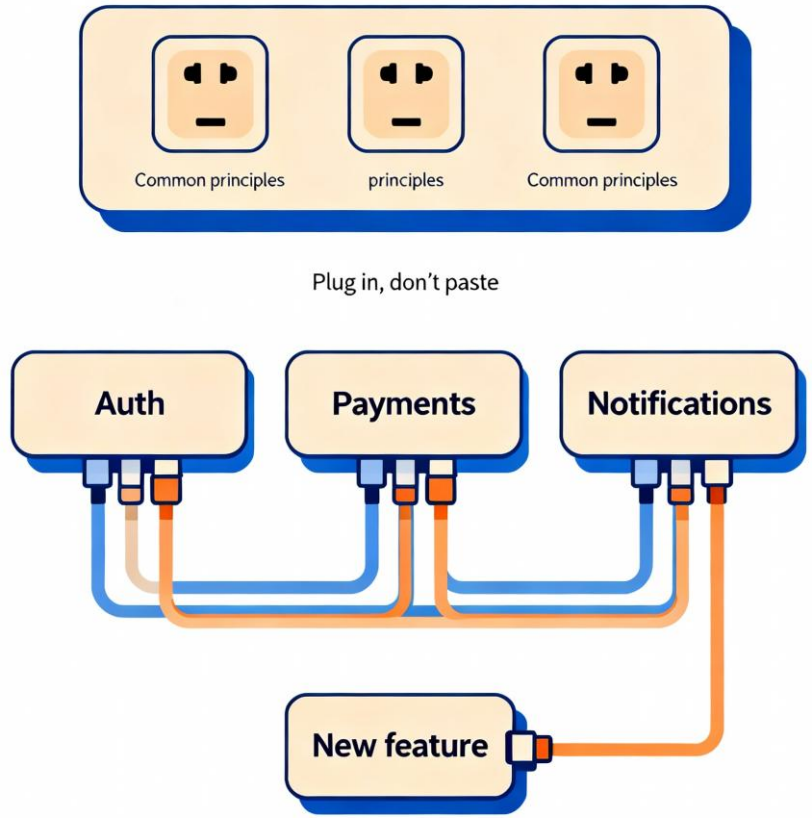


# Show the essence; avoid exposing unnecessary implementation details

- Name capabilities in plain terms users and developers both understand
- Keep interfaces simple; move complexity behind the curtain
- Stable abstractions make change cheaper and conversations easier

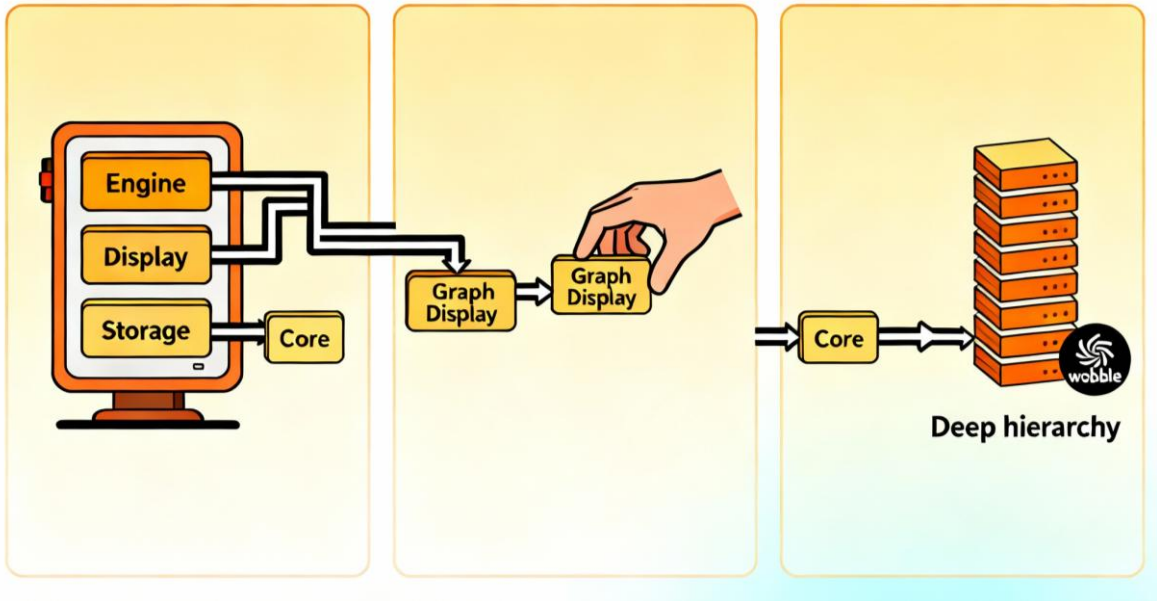
# Build once, reuse widely; extend behavior without breaking existing parts

- Design around common principles so new features fit well with existing ones
- Favor plug-in points over copy-paste code duplication
- Reuse speeds delivery and reduces bugs across teams



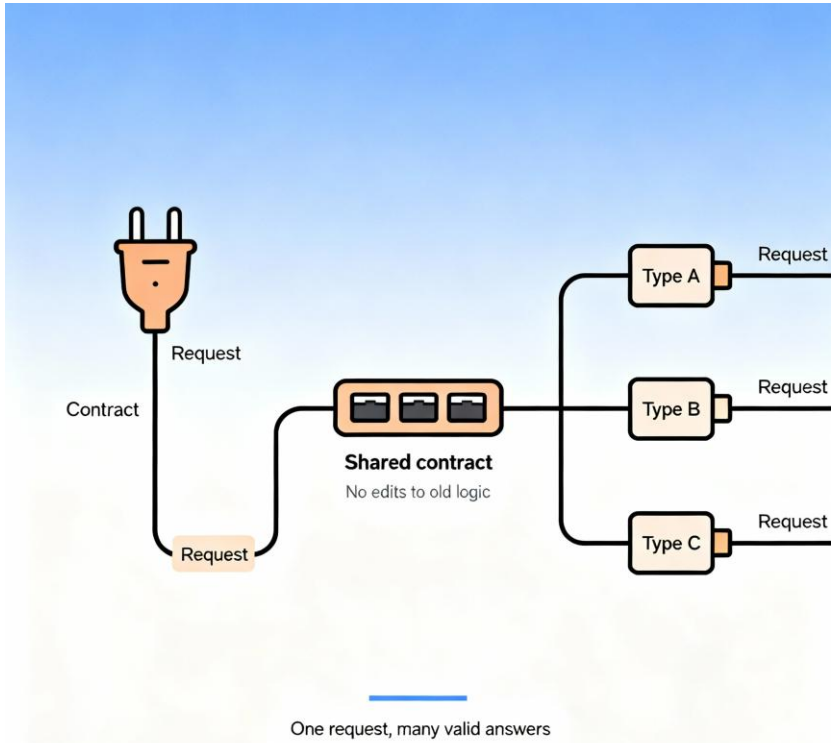
# Prefer small parts working together over deep, rigid class trees

- “**Has-a**” relationships compose features without tight coupling
- Swapping parts stays easy when behavior lives in collaborators
- Shallow structures adapt better than tall hierarchies



# Many types respond to one request, cutting complex if-else chains

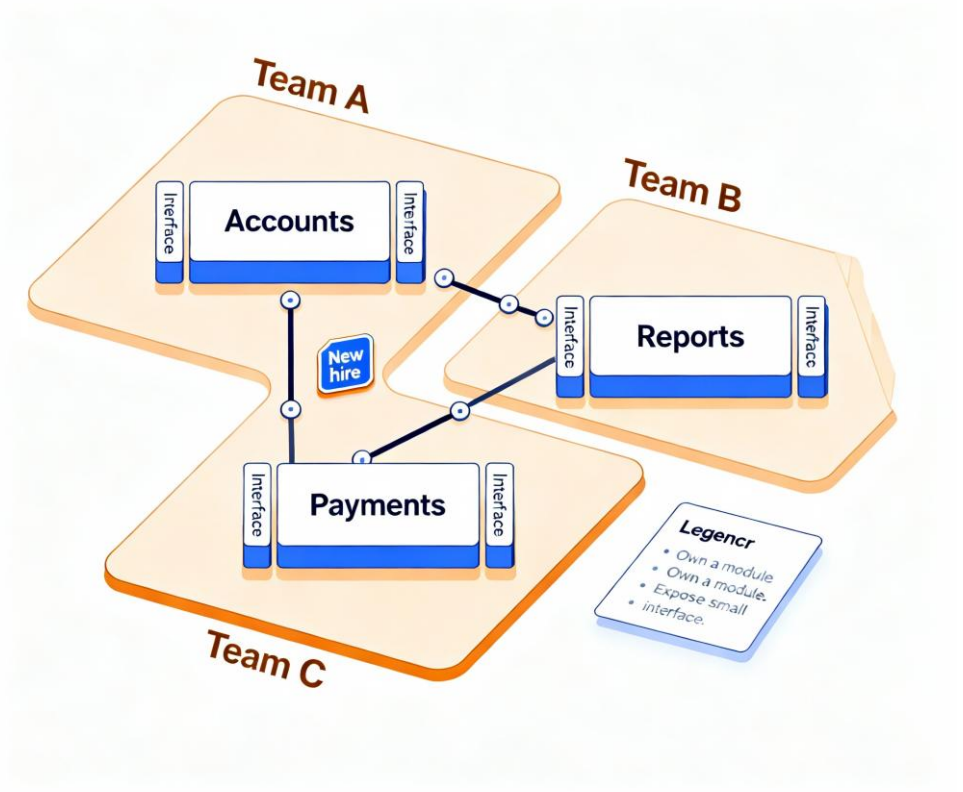
- Code targets shared contracts, not concrete case checks
- New types slot in without editing old decision logic
- Fewer branches mean clearer, safer, easier-to-test code



# Working at Scale

# Clear module boundaries allow parallel work and safer integration

- Teams own cohesive parts with stable interfaces between them
- Small surfaces reduce coordination and merge conflicts
- New members onboard faster with well-named, self-contained units




Own a module. Expose a small interface.

# A bank management domain to anchor ideas and connect weekly topics

- We will model entities, rules, and actions
- We will refine flows step by step as concepts deepen

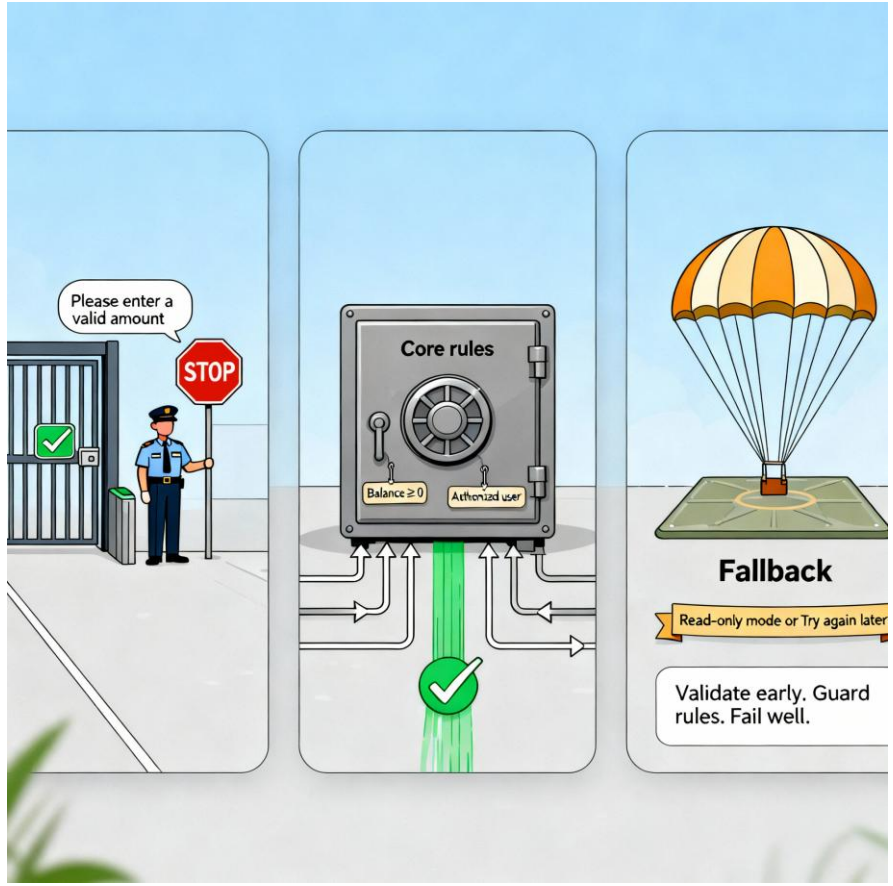
# Give each part one clear job; avoid “does everything” modules

- Cohesive responsibilities make code easier to read and test
- Split formatting, storage, and business rules cleanly
- Clear roles reduce hidden dependencies and surprises

UNIVERSITETI METROPOLITAN TIRANA

# Plan for errors; keep rules true, even when things fail

- Validate inputs early; fail fast  with clear messages
- Keep invariants central so they stay enforced everywhere
- Handle failures gracefully; prefer recovery over silent corruption



# Separate core logic from storage to keep designs portable

- Domain rules should not depend on storage details
- Map objects to files or databases through adapters
- This decoupling simplifies tests and future migrations

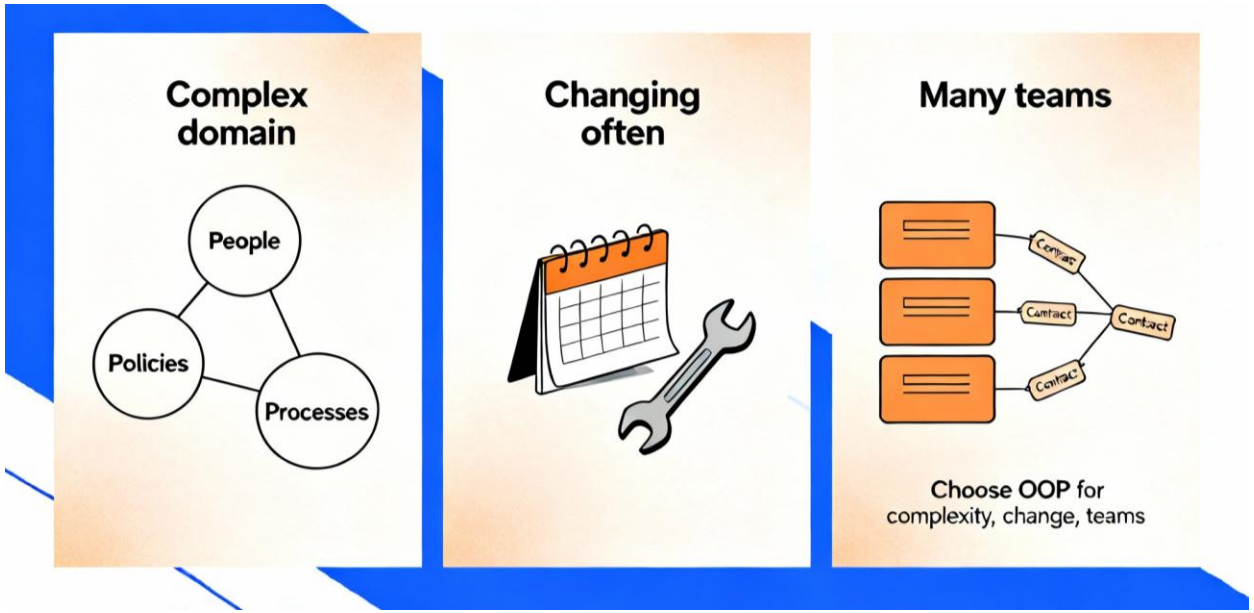
# Modeling, Choosing, and Quality

# Quick sketches align understanding before writing any code

- Draw key parts, simple links, and rough responsibilities
- Capture assumptions to revisit as details emerge
- Keep diagrams minimal; update as ideas change

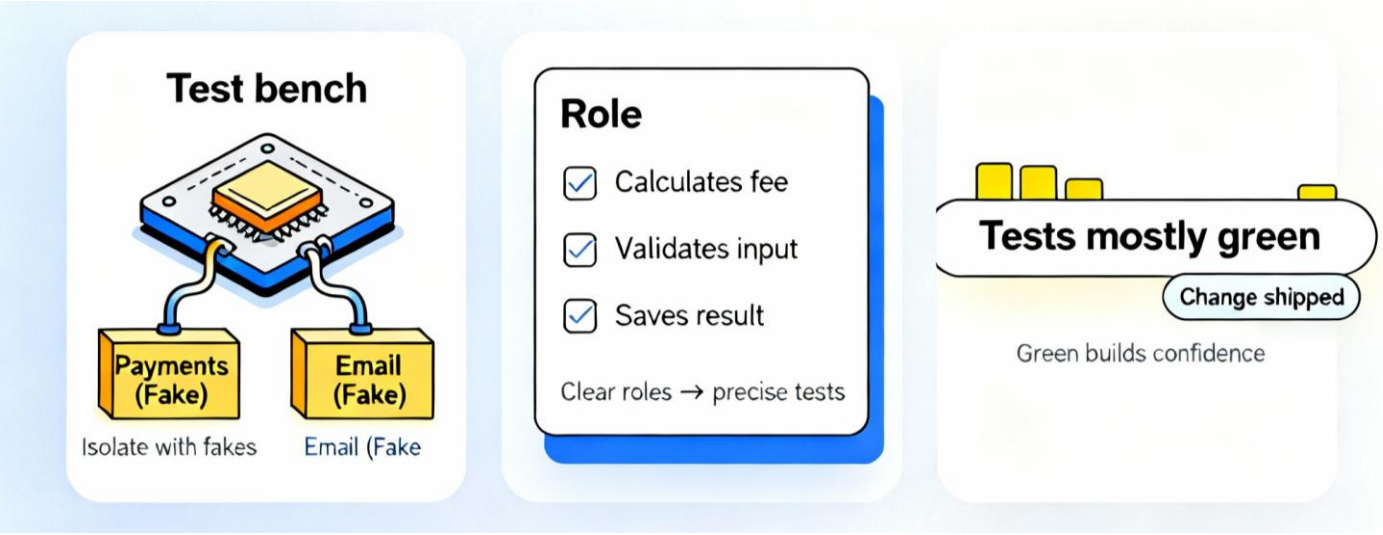
# Choose OOP for complex, evolving domains and long-lived systems

- Many interacting concepts benefit from clear models and boundaries
- Frequent change and multiple teams favor modular, contract-based designs



# Encapsulation and contracts make fast, focused tests possible

- Test parts in isolation with fake collaborators behind contracts

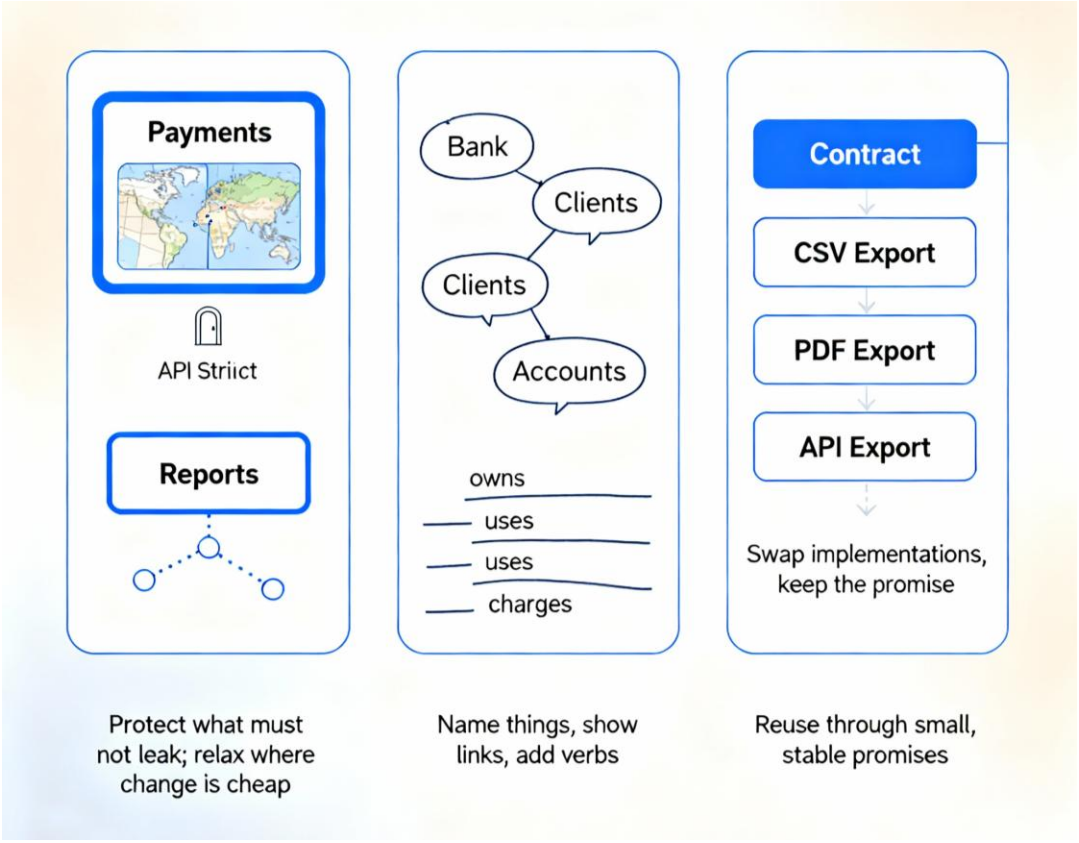



- Clear roles define precise, meaningful test cases
- Confidence grows when changes  keep tests green

UNIVERSITETI METROPOLITAN TIRANA

# Reflect together to connect ideas and surface trade-offs early

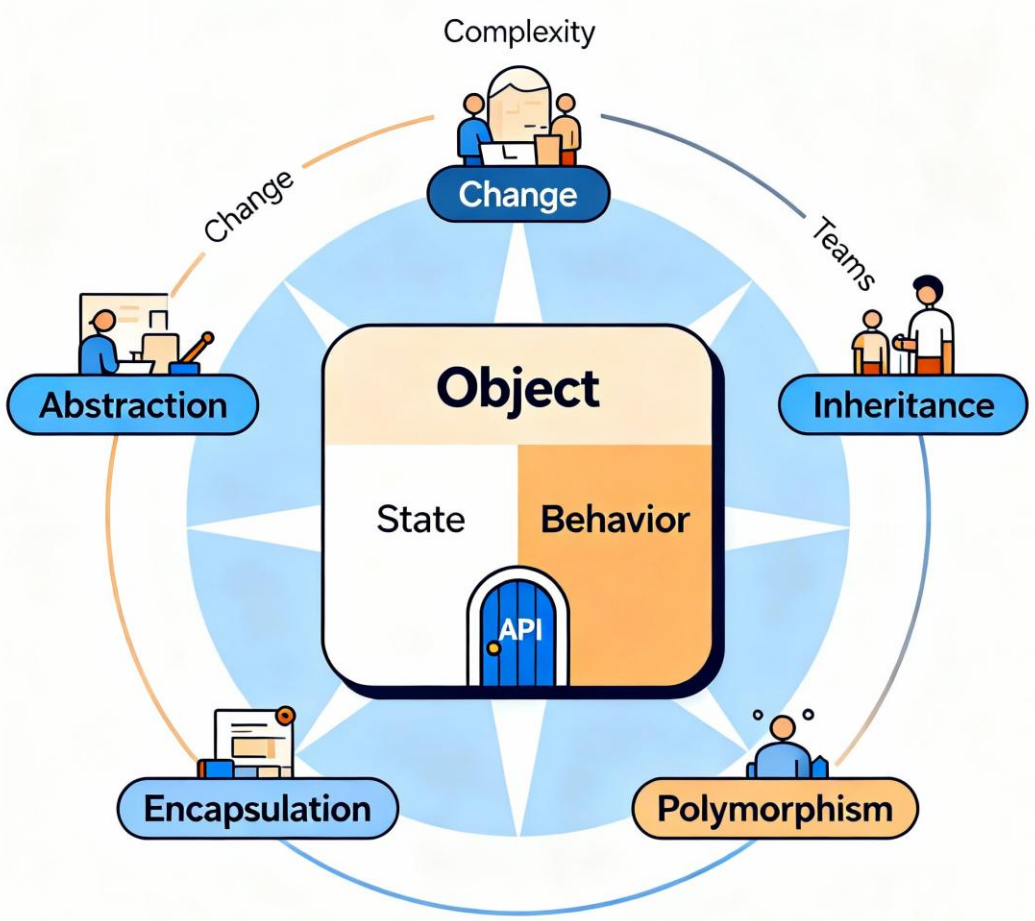
- Which parts deserve strict boundaries, and which can be flexible?
- How to define entities and their relations and actions?
- What contract would allow safe reuse across different features?



UNIVERSITETI METROPOLITAN TIRANA

# Objects help manage complexity, protect rules, and allow change

- Model real things and actions with small parts that cooperate
- Use the four pillars to keep designs clear, safe, and adaptable
- Prefer clear contracts, and strong boundaries for growth



# Learn what to do; hide how to do it

## That provides clarity in life and abstraction in OOP