Rigor and Reasoning in Research Software (R3Sw)

Abstraction, Decomposition & Specification

Alper Altuntas

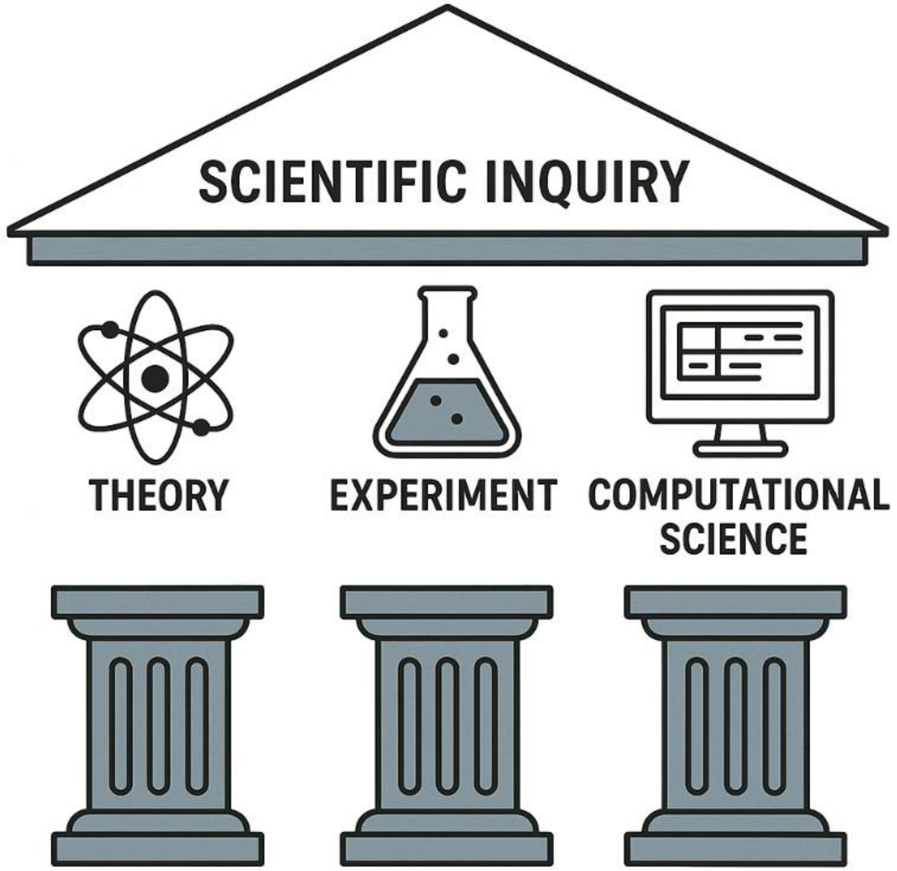
NSF National Center for Atmospheric Research (NCAR)

Nov 5-7, 2025

This work was supported by the Better Scientific Software Fellowship Program, a collaborative effort of the U.S. Department of Energy (DOE), Office of Advanced Scientific Computing Research via ANL under Contract DE-AC02-06CH11357 and the National Nuclear Security Administration Advanced Simulation and Computing Program via LLNL under Contract DE-AC52-07NA27344; and by the National Science Foundation (NSF) via SHI under Grant No. 2435328.

This material is based upon work supported by the National Center for Atmospheric Research, which is a major facility sponsored by the National Science Foundation under Cooperative Agreement No. 1852977.

Why this matters?



“Computational science now constitutes the third pillar of scientific inquiry.”
(Presidential Information Technology Advisory Committee Report, 2005)

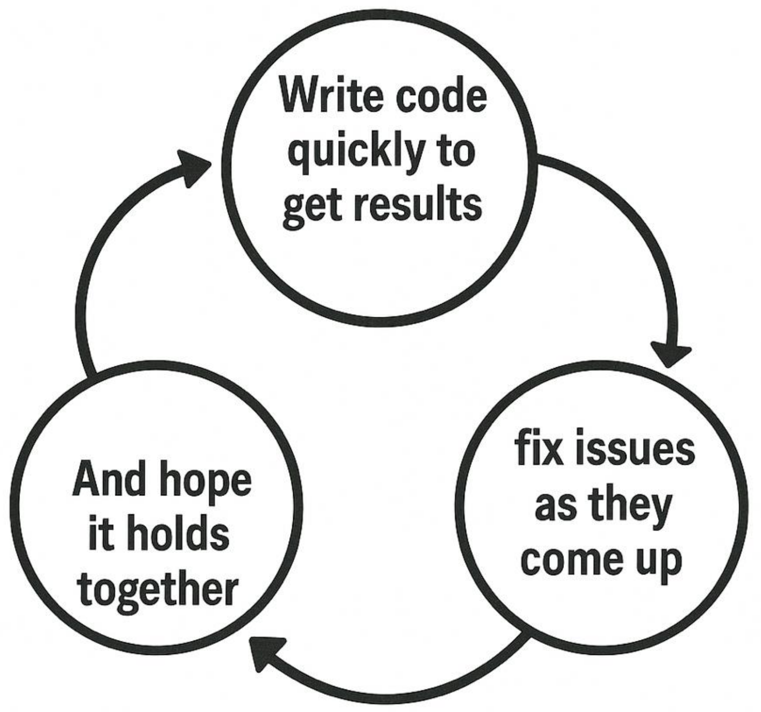
The Goal

- Explore how to make scientific software more reliable, understandable, and robust.
- We’ll aim to do so without losing the creativity and speed that make research exciting.

The code-and-fix style

- write code quickly to get results,
- fix issues as they appear,
- and hope it holds together.

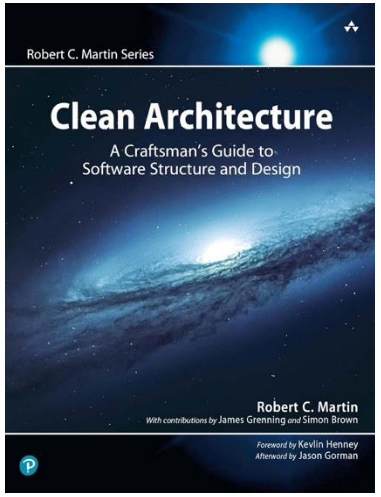
The result? Code that works… most of the time.
But is fragile, hard to test, and tricky to extend.

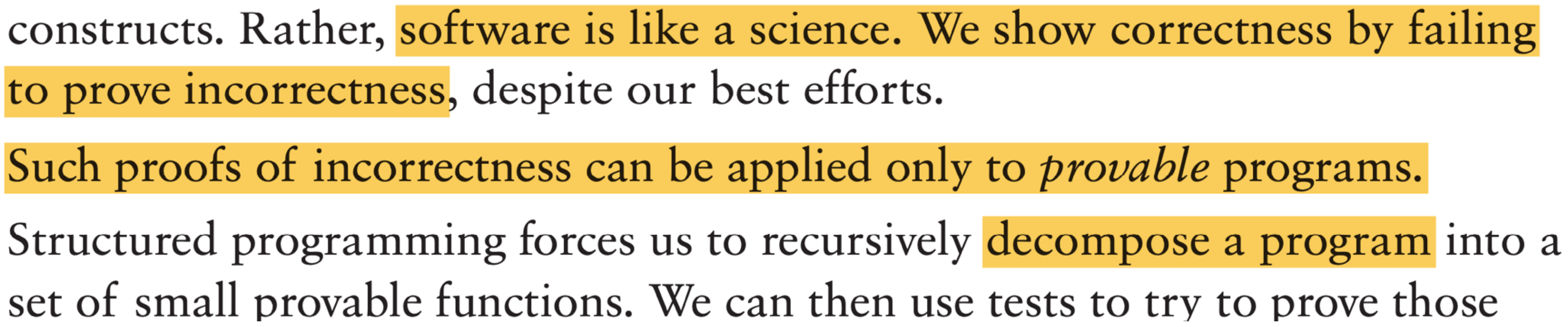

Write code quickly to get results
fix issues as they come up
And hope it holds together

The code-and-fix style

A scientific mindset

- A theory is scientific if it’s ***refutable***.
- Similarly, a software can be ***scientific***, if its testable / verifiable.
- So, we’ll go over practical techniques to:
  - **hypothesize**: specify intent
  - **refute**: test / verification
  - **refine** code & understanding


Robert C. Martin Series
Clean Architecture
A Craftsman's Guide to Software Structure and Design
Robert C. Martin
With contributions by James Grenning and Simon Brown
Foreword by Kevlin Henney
Afterword by Jason Gorman


constructs. Rather, software is like a science. We show correctness by failing to prove incorrectness, despite our best efforts.

Such proofs of incorrectness can be applied only to *provable* programs.

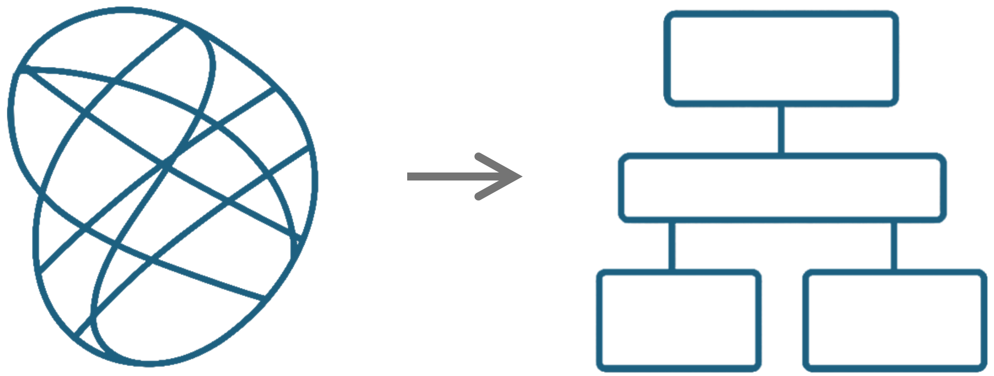
Structured programming forces us to recursively decompose a program into a set of small provable functions. We can then use tests to try to prove those

Robert C. Martin
Agile Manifesto, SOLID principles

"Provable" here doesn't refer to a mathematical proof, but to scientific reasoning: something that can be falsified through experiments, or testing.

Decomposition

- **Based on abstractions**, i.e., the key structures, relations, features, etc.
- **Decompose the code**, into modules, classes, methods, functions, ...
- Such that each part
  - has an **isolated, clear purpose**.
  - exposes **well-defined** interfaces.
  - is **refutable**, i.e., testable.



Decomposition is the central software design task. (J. Ousterhout)

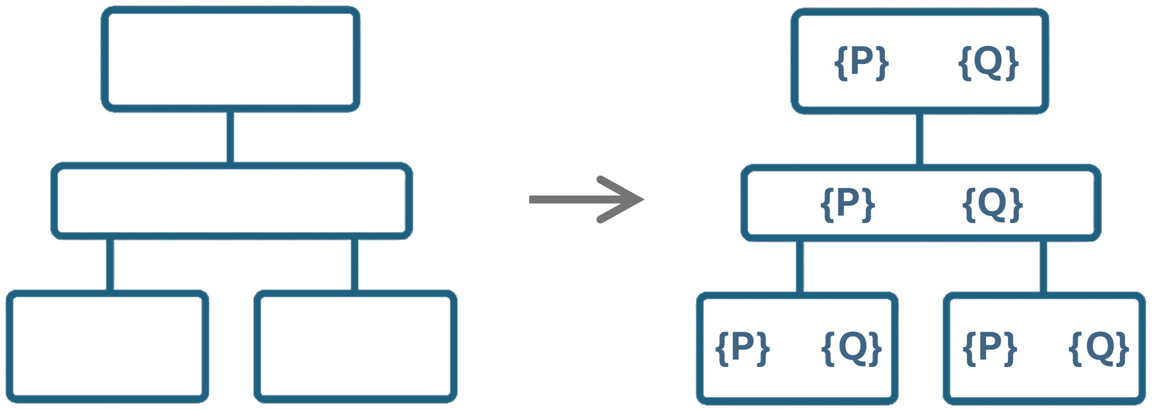
- One we can’t capture fully in a one-day tutorial, or in a few slides.
- Here, we focus on a few key principles for decomposing a ***large, monolithic*** routine into smaller functions that facilitate ***testing and reasoning***:
  - **Single purpose**: each procedure should do one thing and do it clearly.
  - **General**: avoid hard-coded values or hidden assumptions
  - **Explicit, narrow interfaces:** small, coherent APIs that express intent
  - **Dependency injection**: pass in policies or strategies, don’t grab globals.

Specification

Once we have modular pieces, we need to say what each one should do:

- A ***specification*** is a precise, ***refutable*** statement that defines:
  - What properties, conditions, and constraints are expected to hold.
  - What behaviors and outputs should result from given inputs.

- Levels of specification:
  - System-level.
  - Module/class/**function level**.



What to specify?

- **Assumptions**: background facts we take as given (e.g., valid b.c.)
- **Contracts**: function preconditions and postconditions.
- **Properties**: logical, computational, or physical/scientific.

What to specify?

- **Assumptions**: background facts we take as given (e.g., valid b.c.)
- **Contracts**: function preconditions and postconditions.
- **Properties**: logical, computational, or physical/scientific.

```python
def div(x, y):
    res = x / y
    return res
```

$\xrightarrow{\textit{Contract Specification}}$

```python
def div(x: float, y: float) -> float:
    assert y != 0           # P
    res = x / y             # code
    assert res * y == x     # Q
    return res
```

How to specify?

In this tutorial, our main focus is function-level specification. As such, we use a lightweight, practical style of specification that combines:

- **Type annotations**
- **Assertions**

Both pytest and hypothesis libraries rely on assertions.

Why specify?

- **Testing**: Reasoning artifacts (e.g., assertions) ease testing efforts.
- **Change**: To safely evolve or replace implementations.
- **Reasoning and Understanding**: To think rigorously about intent.

  “Writing is Nature's way of showing you how sloppy your thinking is.” (Dick Guindon)

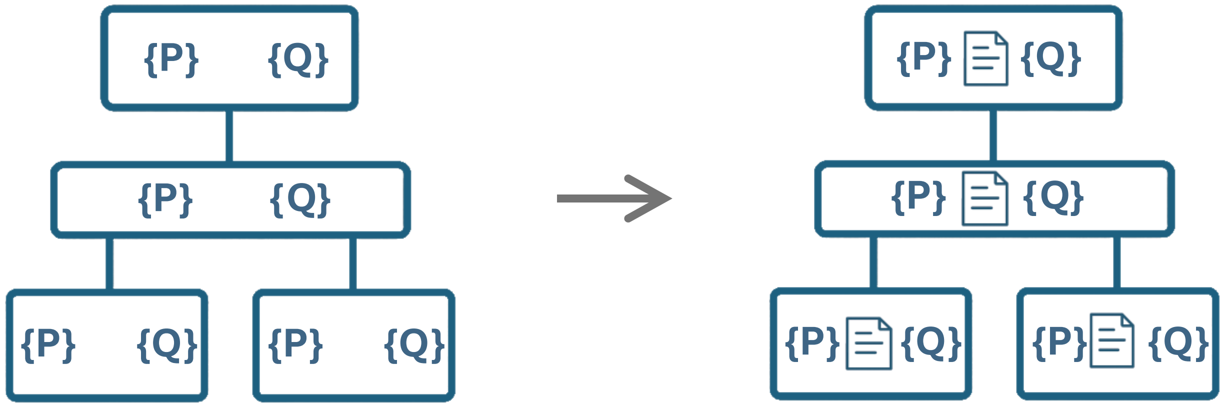
Implementation

With abstractions identified, pieces decomposed, and specifications written:

- implement code that satisfies them.

This shifts our mindset:

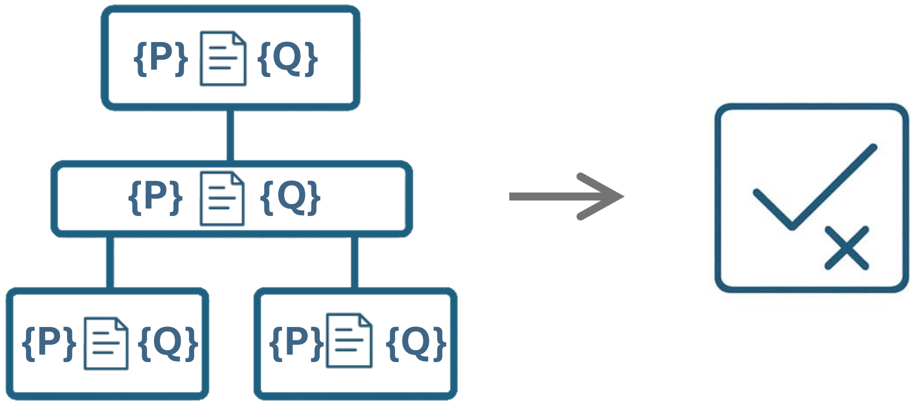
- Instead of starting with lines of code and hoping they work, we start with clear expectations and then fill in implementations that must meet them.



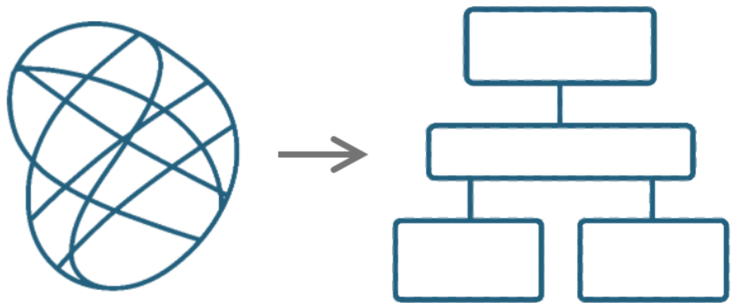
Testing & verification (i.e., refutation)

This is the final step. The majority of this tutorial is devoted to practical testing and verification techniques. We'll cover:

- **Unit testing**: *for this input, expect this output*
- **Property-based testing**: *for all inputs satisfying preconditions, expect this relation to hold*
- **Bounded Formal Verification**: *prove that properties hold for all possible inputs*
- **Functional "off-offline" testing:** *real-world applications*

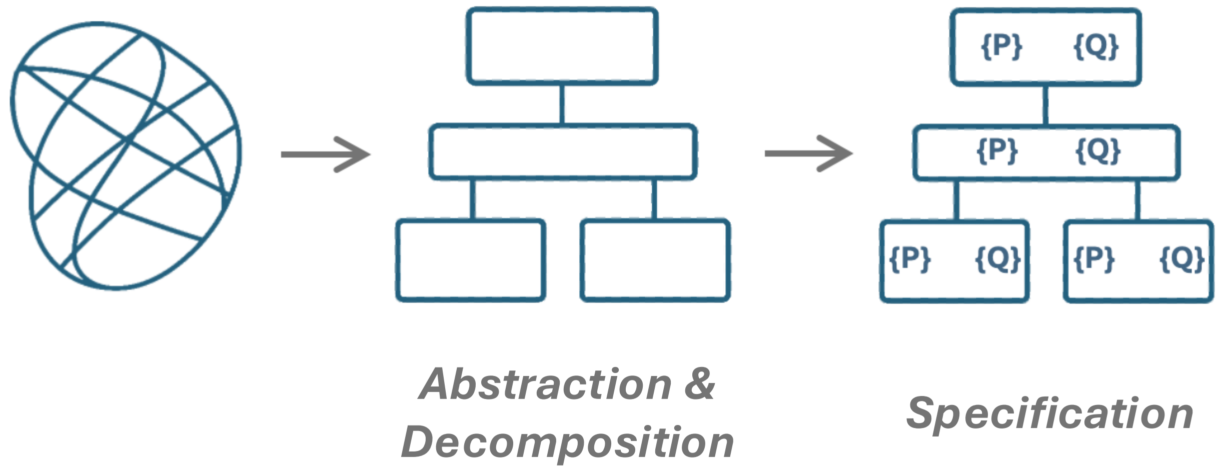


In summary

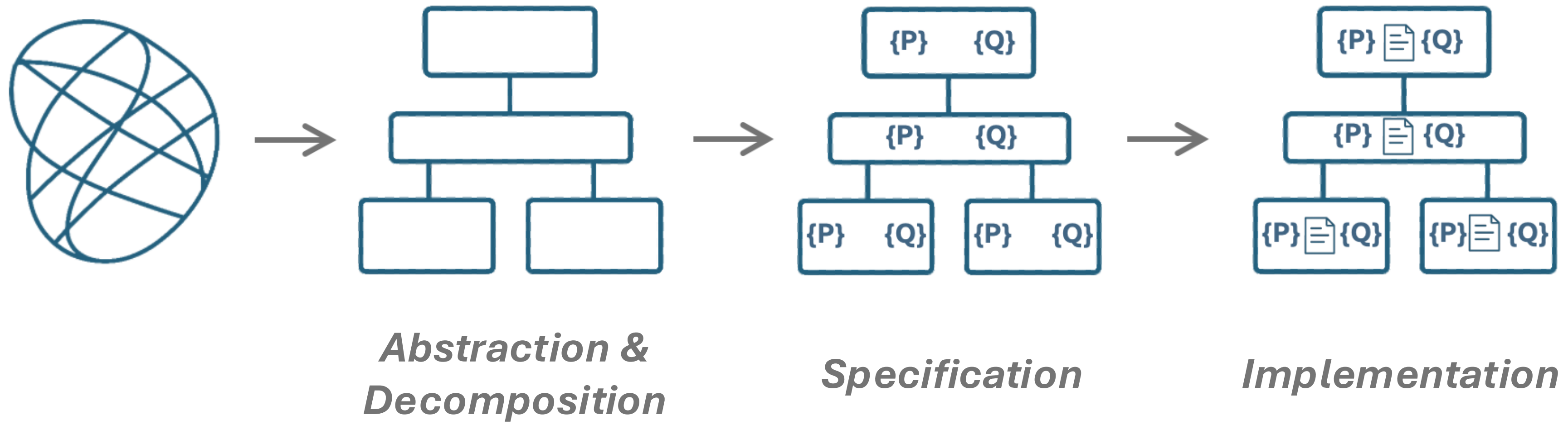


Abstraction & Decomposition

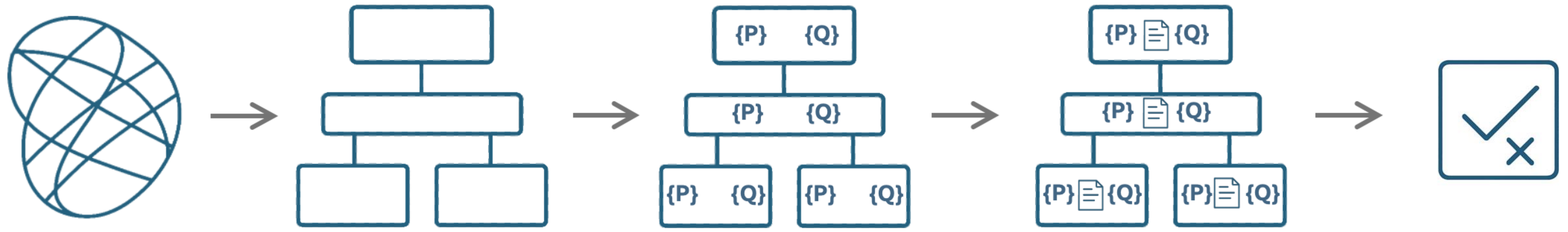
In summary



In summary


{P}
{Q}
Abstraction &
Decomposition
Specification
Implementation

In summary



Abstraction & Decomposition

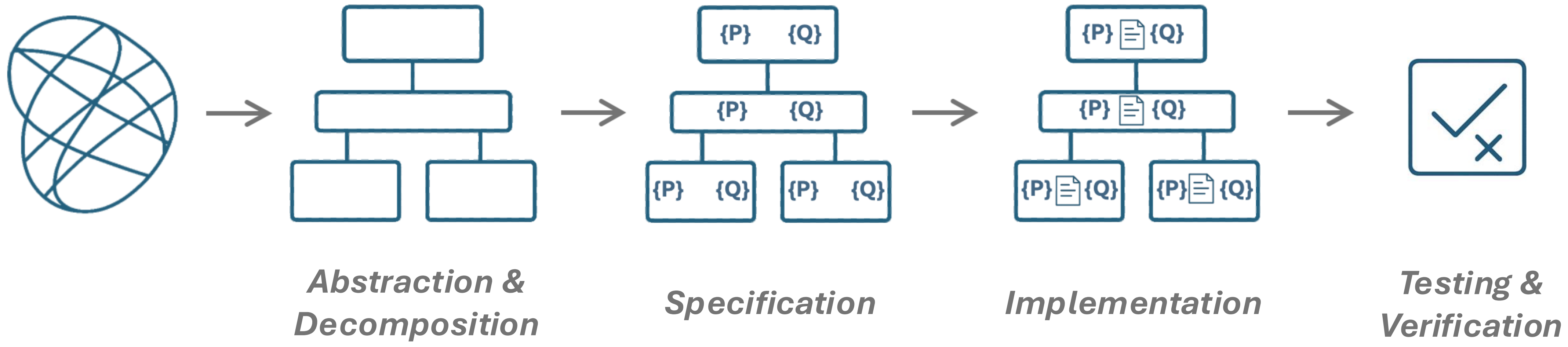
Specification

Implementation

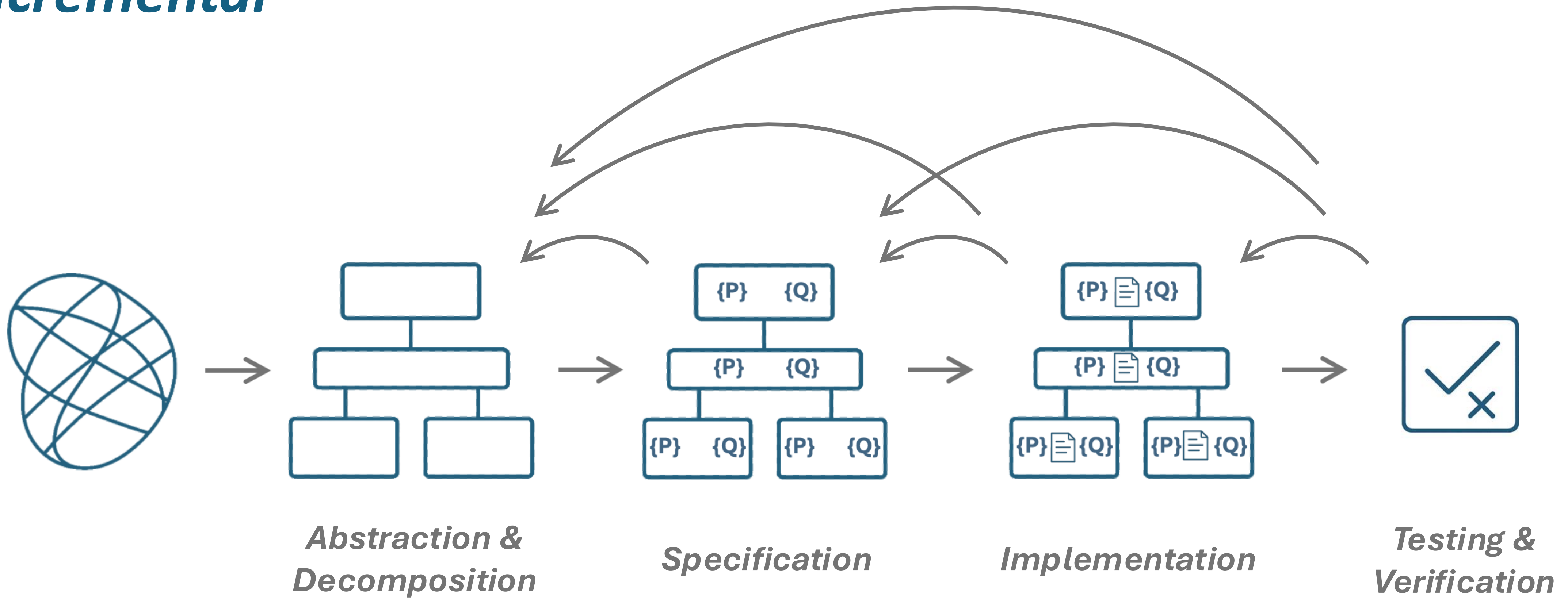
Testing & Verification

Waterfall?

“The initial design for a system or component is almost never the best one; experience inevitably shows better ways to do things.” (J. Ousterhout, 2022)



~~Waterfall~~
Incremental



~~Waterfall~~

Incremental

To enable incremental design & development:

- **Tight feedback loops:** Later work (impl/tests) can challenge and refine earlier choices.
- **Move in small slices:** specify just enough, implement a bit, test and revise.

How this works:

- **Clear abstractions** → easy to change parts without ripple.
- **Clear, stable interfaces** → refactor internals freely.
- **Specs/contracts** → instant oracles
- **Fast unit/property tests** → quick checks

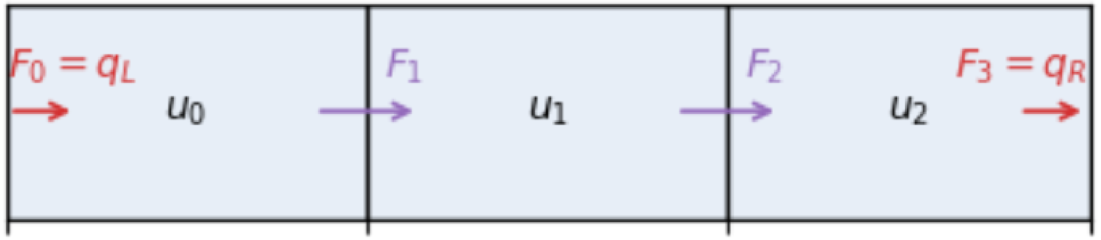
A running example

A running example: The 1-D Heat equation

$$\frac{\partial u}{\partial t} = \kappa \frac{\partial^2 u}{\partial x^2}$$

A running example: The 1-D Heat equation

$$\frac{\partial u}{\partial t} = \kappa \frac{\partial^2 u}{\partial x^2}$$



$$F_i = -\kappa \frac{u_i - u_{i-1}}{\Delta x}$$

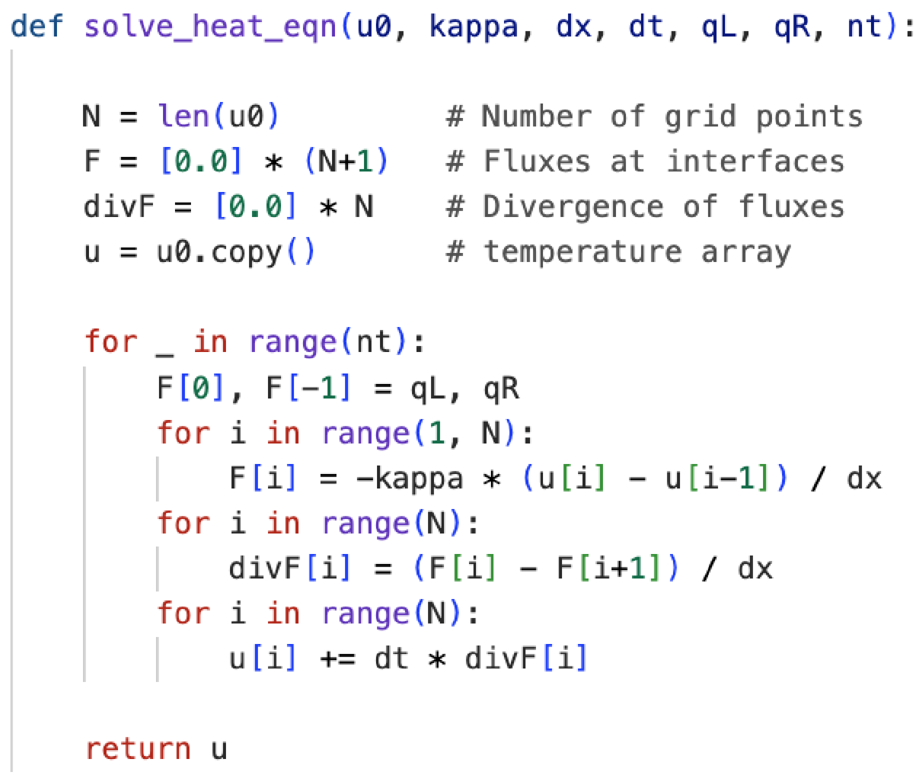
$$div(F_i) = \frac{F_i - F_{i+1}}{\Delta x}$$

$$u_i^{n+1} = u_i^n + \Delta t \cdot div(F_i)$$

A quick implementation

```python
def solve_heat_eqn(u0, kappa, dx, dt, qL, qR, nt):

    N = len(u0)           # Number of grid points
    F = [0.0] * (N+1)     # Fluxes at interfaces
    divF = [0.0] * N      # Divergence of fluxes
    u = u0.copy()         # temperature array

    for _ in range(nt):
        F[0], F[-1] = qL, qR
        for i in range(1, N):
            F[i] = -kappa * (u[i] - u[i-1]) / dx
        for i in range(N):
            divF[i] = (F[i] - F[i+1]) / dx
        for i in range(N):
            u[i] += dt * divF[i]

    return u
```

A quick implementation



```python
def solve_heat_eqn(u0, kappa, dx, dt, qL, qR, nt):

    N = len(u0)           # Number of grid points
    F = [0.0] * (N+1)     # Fluxes at interfaces
    divF = [0.0] * N      # Divergence of fluxes
    u = u0.copy()         # temperature array

    for _ in range(nt):
        F[0], F[-1] = qL, qR
        for i in range(1, N):
            F[i] = -kappa * (u[i] - u[i-1]) / dx
        for i in range(N):
            divF[i] = (F[i] - F[i+1]) / dx
        for i in range(N):
            u[i] += dt * divF[i]

    return u
```

Quite straightforward and manageable, but allows us to observe and generalize some issues regarding testing/reasoning:

- **Unclear abstractions**: data structures and relations not explicit.
- **No reasoning artifacts**: no assertions, property specifications, or contracts.
- **Mixed concerns**: one function handling multiple responsibilities.
- **Opaque state**: hard to inspect intermediate results.
- **No reuse or substitution**: computations tightly coupled.

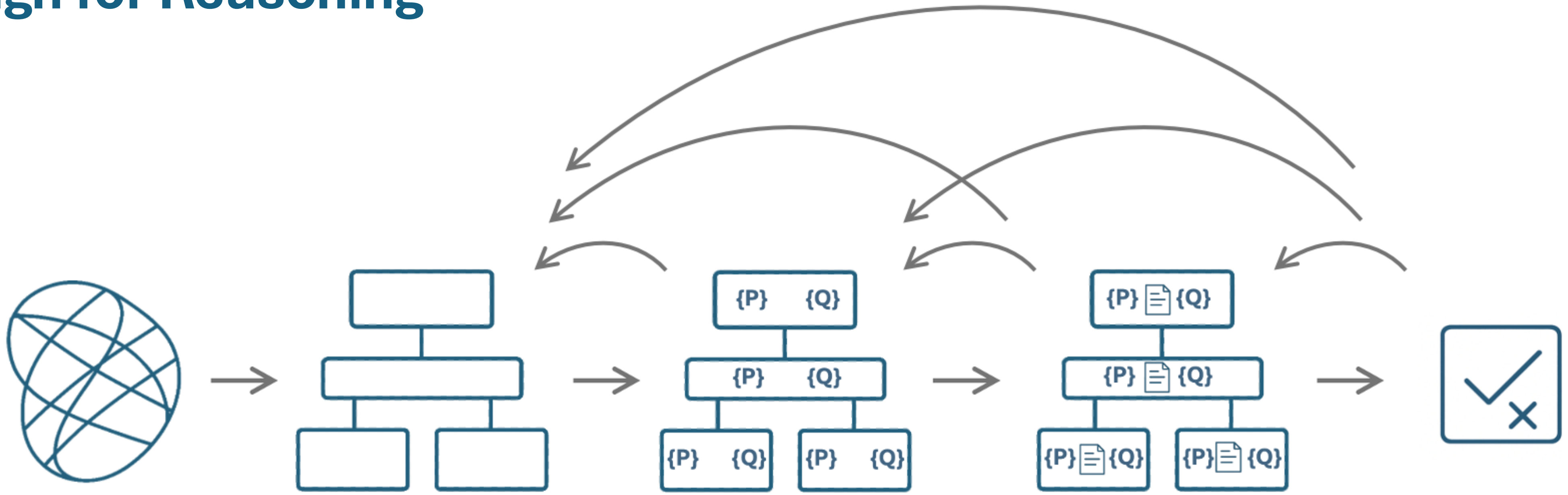
Monolithic Code - Observations

For only ~10 lines of code, these issues may not seem significant.

Our aim is to discuss strategies and techniques generalizable to real applications:

- The same patterns/issues appear in routines with 100s of lines.
- There, these design flaws become real barriers to understanding and testing.

Design for Reasoning



What we had:

- monolithic
- mixed concerns
- hard to test or extend
- lack explicit reasoning

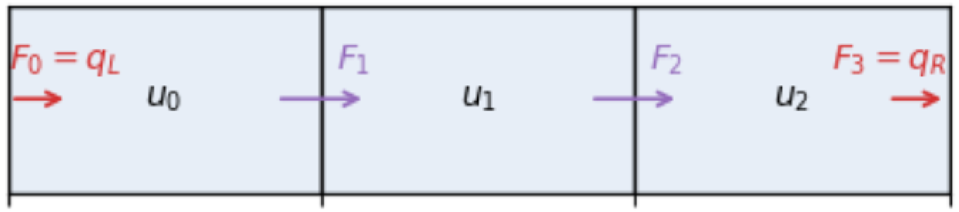
What we want:

- separation of concerns
- explicit expectations
- A structure we can *understand, test, extend*

Abstraction – Deciding What Matters

Goal: Identify the key concepts that define the system.

- Focus on *what* we represent, not *how* we compute it.



Concepts in Heat Equation Solver:

- **Mesh** → domain geometry. A collection of cells.
- **Cell fields** → quantities stored at cell centers (or cell avg.d)
- **Face fields** → quantities incident on cell interfaces
- **Computations**

Abstraction – Deciding What Matters

```python
from dataclasses import dataclass

@dataclass
class Mesh:
    """Uniform 1-D mesh."""

    dx: float  # cell size
    N: int     # number of cells

    def cell_field(self) -> vec:
        return [0.0] * self.N

    def face_field(self) -> vec:
        return [0.0] * (self.N + 1)
```

Abstraction – Deciding What Matters

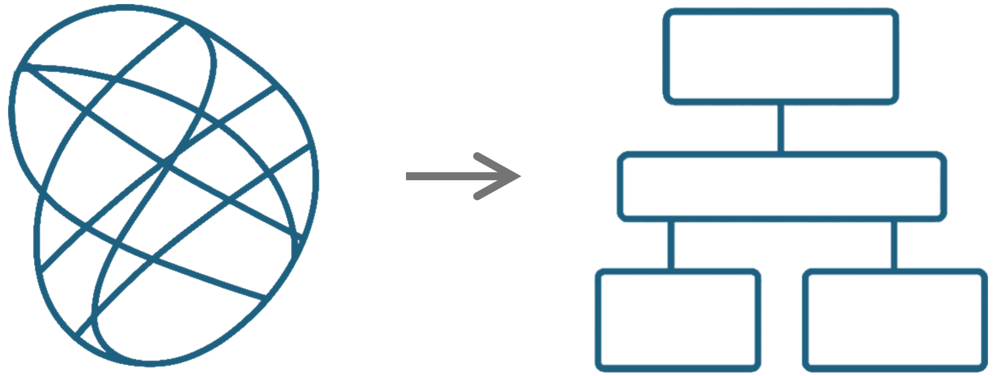
```
from dataclasses import dataclass

@dataclass
class Mesh:
    """Uniform 1-D mesh."""

    dx: float  # cell size
    N: int     # number of cells

    def cell_field(self) -> vec:
        return [0.0] * self.N

    def face_field(self) -> vec:
        return [0.0] * (self.N + 1)
```

Note that our approach is mainly based on procedural abstraction. We won't rely much on OOP features beyond this simple data class.

Decomposition – Organizing the code

Goal: Split computation along abstractions.

- Each procedure should have a **single, clear purpose.**
- Procedures should be **general.** Avoid hard-coded assumptions.
- Keep them **simple but composable.**



Decomposition – The core API

```python
# Procedures:

def apply_bc(f_out: vec, bc: vec) -> None:
    """Apply BCs by overriding first and last face quantities (f_out)."""
    ...

def diffusive_flux(f_out: vec, c: vec, kappa: float, dx: float) -> None:
    """Given a cell field (c), compute the diffusive flux (f_out)."""
    ...

def divergence(c_out: vec, f: vec, dx: float) -> None:
    """Compute the divergence of face quantities (f) and store in (c_out)."""
    ...

def step_heat_eqn(u_inout: vec, kappa: float, dt: float, mesh: Mesh, bc: vec):
    """Advance cell field u by one time step using explicit Euler method."""
    ...

def solve_heat_eqn(u0: vec, kappa: float, dt: float, nt: int, dx: float, bc: vec) -> vec:
    """Orchestrate nt steps over cell field u."""
    ...
```

Decomposition – The core API

```python
# Procedures:

def apply_bc(f_out: vec, bc: vec) -> None:
    """Apply BCs by overriding first and last face quantities (f_out)."""
    ...

def diffusive_flux(f_out: vec, c: vec, kappa: float, dx: float) -> None:
    """Given a cell field (c), compute the diffusive flux (f_out)."""
    ...

def divergence(c_out: vec, f: vec, dx: float) -> None:
    """Compute the divergence of face quantities (f) and
    ...

def step_heat_eqn(u_inout: vec, kappa: float, dt: float,
    """Advance cell field u by one time step using explic
    ...

def solve_heat_eqn(u0: vec, kappa: float, dt: float, nt:
    """Orchestrate nt steps over cell field u."""
    ...
```

Argument naming convention:

- c: cell field
- f: face field
- _out suffix: output argument
- _inout suffix: input/output argument

These general argument names make the procedures reusable (in line with *abstraction by parameterization*.)

Program to an interface, not an implementation

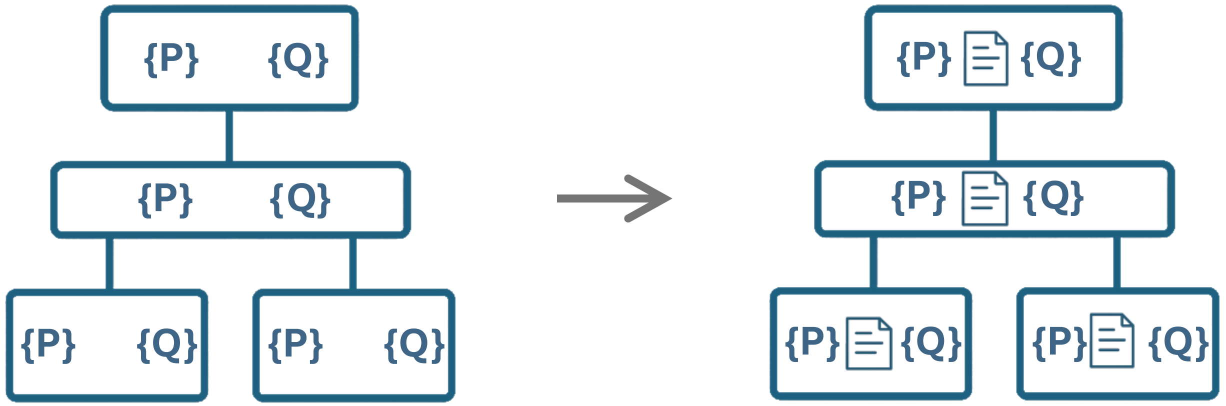
- Gamma et al. *Design Patterns*, 1995

Design each function around what it promises to do, not how it does it.
By depending only on clear, minimal interfaces (inputs, outputs, and contracts) other parts of the code remain unaffected when internals change.

Specification

In addition to type annotations in core API, we can specify contracts:

```python
def divergence(c_out: vec, f: vec, dx: float) -> None:
    """Compute the divergence of face quantities (f) and store in (c_out)."""
    assert len(c_out) == len(f) - 1, "Size mismatch"
    assert dx > 0, "Non-positive dx"
    ...
```



Specification

Beyond contracts and structural correctness, we can express:

- logical, computational, or physical/scientific properties

For the heat equation solver, examples include:

- **Telescoping:** sum of divergences equals boundary flux difference
- **Conservation:** total heat remains constant
- **Symmetry:** symmetric initial conditions stay symmetric
- **Monotonicity:** No new extrema

Specifying the Telescoping Property

The sum of the divergence over all cells equals the net flux through the boundaries.

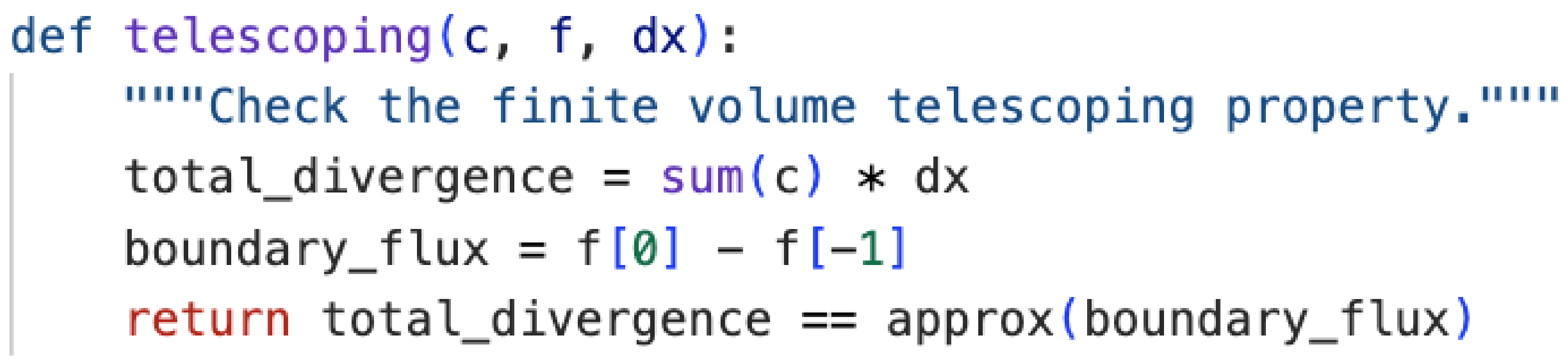
$$\sum_{i=0}^{N-1} (\nabla \cdot F)_i = F_0 - F_N$$

Specifying the Telescoping Property

The sum of the divergence over all cells equals the net flux through the boundaries.

$$\sum_{i=0}^{N-1}(\nabla \cdot F)_i = F_0 - F_N$$

Isolation of concerns and having well defined APIs facilitate specification and testing of such properties:

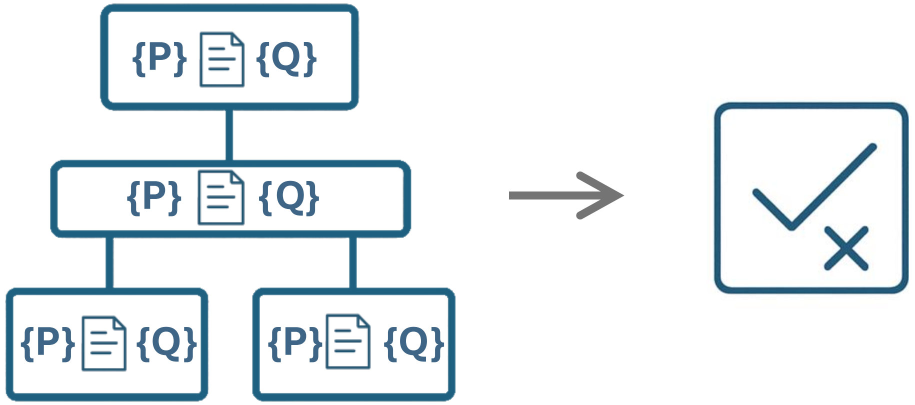


```python
def telescoping(c, f, dx):
    """Check the finite volume telescoping property."""
    total_divergence = sum(c) * dx
    boundary_flux = f[0] - f[-1]
    return total_divergence == approx(boundary_flux)
```

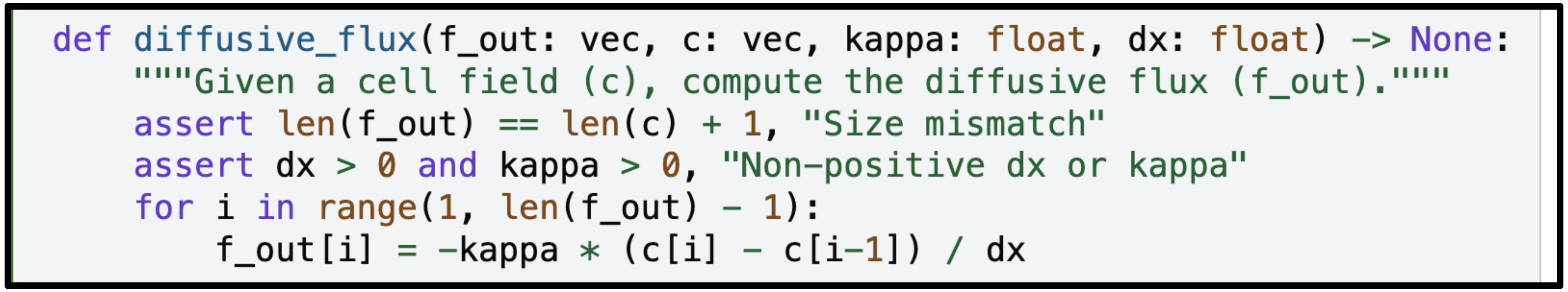
Implementation

With abstractions identified, pieces decomposed, and specifications written:

- implement code that satisfies them.



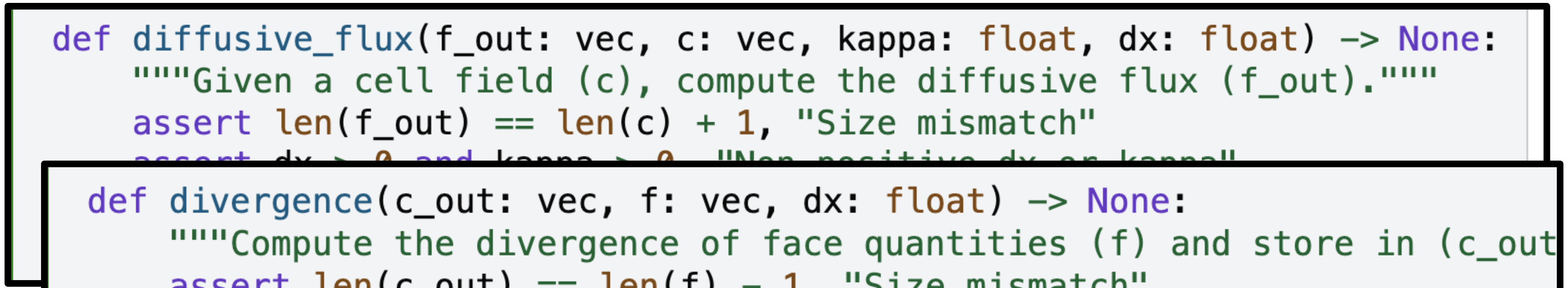
Implementation



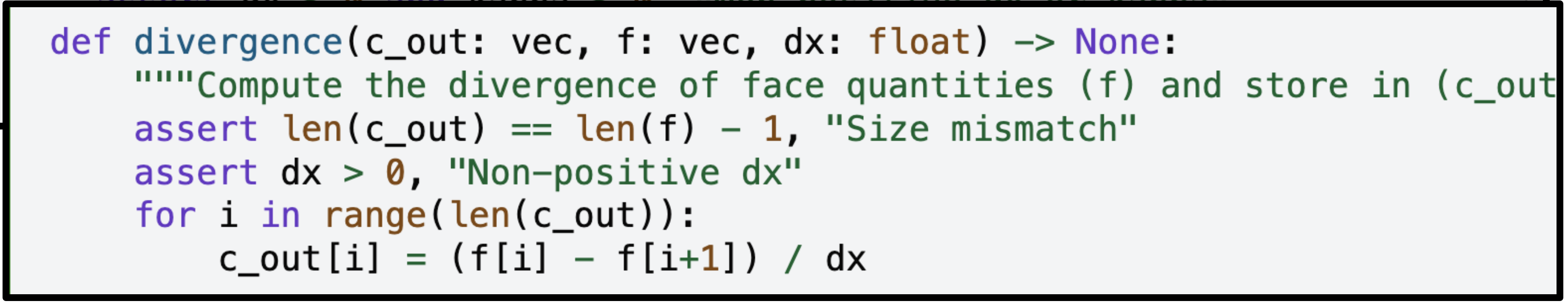
```python
def diffusive_flux(f_out: vec, c: vec, kappa: float, dx: float) -> None:
    """Given a cell field (c), compute the diffusive flux (f_out)."""
    assert len(f_out) == len(c) + 1, "Size mismatch"
    assert dx > 0 and kappa > 0, "Non-positive dx or kappa"
    for i in range(1, len(f_out) - 1):
        f_out[i] = -kappa * (c[i] - c[i-1]) / dx
```

$$F_i = -\kappa \frac{u_i - u_{i-1}}{\Delta x}$$

Implementation



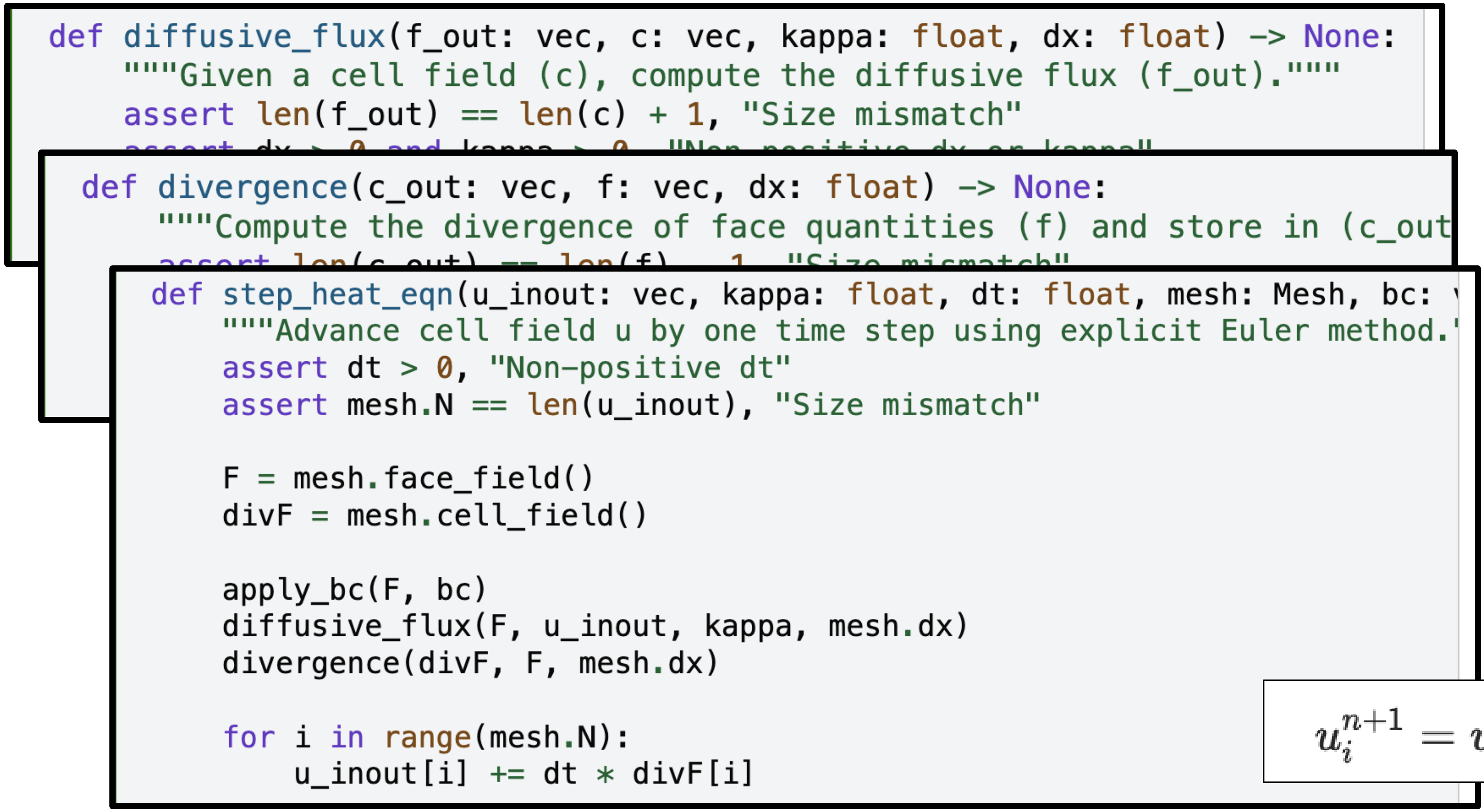
```
def diffusive_flux(f_out: vec, c: vec, kappa: float, dx: float) -> None:
    """Given a cell field (c), compute the diffusive flux (f_out)."""
    assert len(f_out) == len(c) + 1, "Size mismatch"
    assert dx > 0 and kappa > 0, "Non-positive dx or kappa"
```



```
def divergence(c_out: vec, f: vec, dx: float) -> None:
    """Compute the divergence of face quantities (f) and store in (c_out
    assert len(c_out) == len(f) - 1, "Size mismatch"
    assert dx > 0, "Non-positive dx"
    for i in range(len(c_out)):
        c_out[i] = (f[i] - f[i+1]) / dx
```

$$div(F_i) = \frac{F_i - F_{i+1}}{\Delta x}$$

Implementation



```
def diffusive_flux(f_out: vec, c: vec, kappa: float, dx: float) -> None:
    """Given a cell field (c), compute the diffusive flux (f_out)."""
    assert len(f_out) == len(c) + 1, "Size mismatch"
    assert dx > 0 and kappa > 0, "Non-positive dx or kappa"
```

```
def divergence(c_out: vec, f: vec, dx: float) -> None:
    """Compute the divergence of face quantities (f) and store in (c_out
    assert len(c_out) == len(f) - 1, "Size mismatch"
```

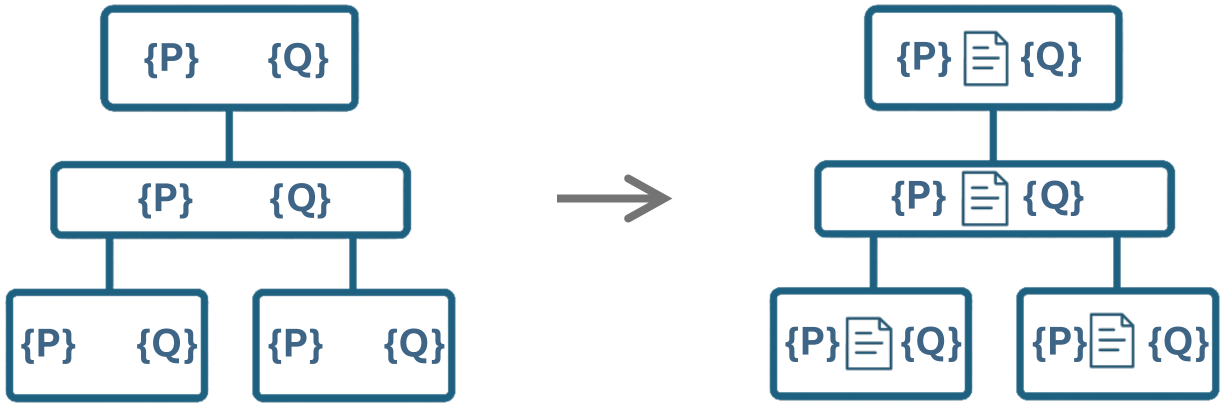
```
def step_heat_eqn(u_inout: vec, kappa: float, dt: float, mesh: Mesh, bc:
    """Advance cell field u by one time step using explicit Euler method.
    assert dt > 0, "Non-positive dt"
    assert mesh.N == len(u_inout), "Size mismatch"

    F = mesh.face_field()
    divF = mesh.cell_field()

    apply_bc(F, bc)
    diffusive_flux(F, u_inout, kappa, mesh.dx)
    divergence(divF, F, mesh.dx)

    for i in range(mesh.N):
        u_inout[i] += dt * divF[i]
```

$$u_i^{n+1} = u_i^n + \Delta t \cdot div(F_i)$$

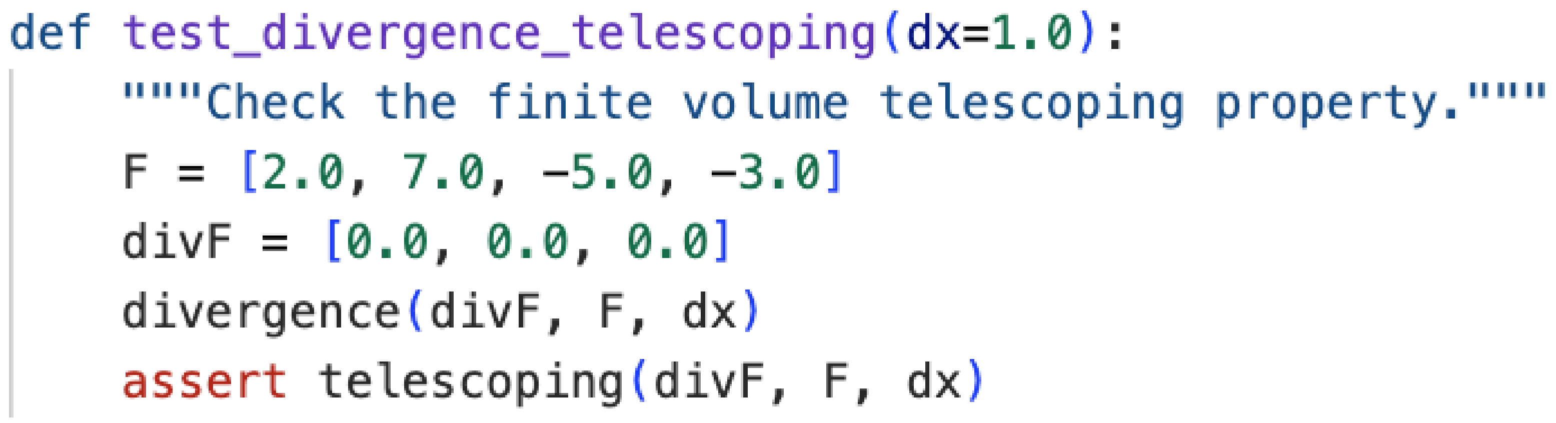
Testing

In the remainder of this tutorial, we’ll cover several testing and verification techniques in detail with and with hands-on exercises:

- Unit Testing
- Property-base testing
- Bounded formal verification
- Functional Testing



Unit Testing



```python
def test_divergence_telescoping(dx=1.0):
    """Check the finite volume telescoping property."""
    F = [2.0, 7.0, -5.0, -3.0]
    divF = [0.0, 0.0, 0.0]
    divergence(divF, F, dx)
    assert telescoping(divF, F, dx)
```

Manish Venumuddula
(NSF NCAR)

Unit Testing

```python
def test_divergence_telescoping(dx=1.0):
    """Check the finite volume telescoping property."""
    F = [2.0, 7.0, -5.0, -3.0]
    divF = [0.0, 0.0, 0.0]
    divergence(divF, F, dx)
    assert telescoping(divF, F, dx)
```

```
○ (r3sw) [cgdm-flax: ~]$ pytest test_divergence.py
```

Unit Testing

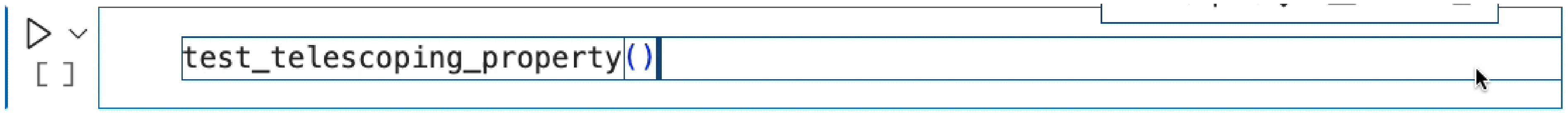
- Unit tests are practical and useful.
- They are great for quick, focused feedback.
- But they only test the examples we think to write.

Property-Based Testing

- For all inputs satisfying preconditions, check specified relations/properties.
- Auto-generate many random inputs: explore cases we might never think to test.
- The randomness is *smart*:
  - guided by strategies and constraints.
  - target edge cases.
  - shrink counterexamples.

Deepak Cherian
(Earthmover)

Property-Based Testing

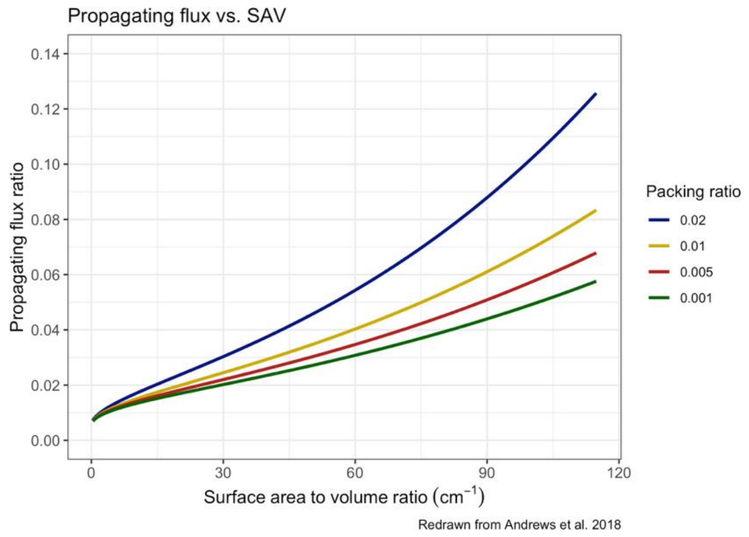


```
[ ]  test_telescoping_property()
```

Functional Testing

- Apply this mindset to a Python version of a production code that simulates wildfire behavior (translated from Fortran).
- Explore **scientific representations**:
  - using synthetic fuels and comparing to observational data.

Adrianna Foster
(NSF NCAR)
"Functional testing"



low load dry climate grass

dwarf conifer with understory

moderate load conifer litter

“Testing can be used to show the presence of bugs,
but never to show their absence.”

- Edsger W. Dijkstra

Bounded Formal Verification

- *Symbolically* check whether properties hold for all possible inputs within specified ranges.
- Less practical, but still achievable when we have the right abstractions and decompositions.

```python
def prove_telescoping(N):
    dx = Real('dx')
    f = [Real(f'f{i}') for i in range(N+1)]
    c = [Real(f'c{i}') for i in range(N)]

    s = Solver()
    s.add(dx > 0)                       # Preconditions: physical constraints
    s.add(divergence(c, f, dx))         # code: divergence
    s.add(Not(telescoping(c, f, dx)))# Postcondition: telescoping property
    return s.check() == unsat
```

Bounded Formal Verification

By reasoning symbolically:

- we move from **testing some cases** to **proving properties for all cases** within a defined scope.

This gives **scope completeness**:

- The solver explores ***every*** combination of inputs allowed by the specification.

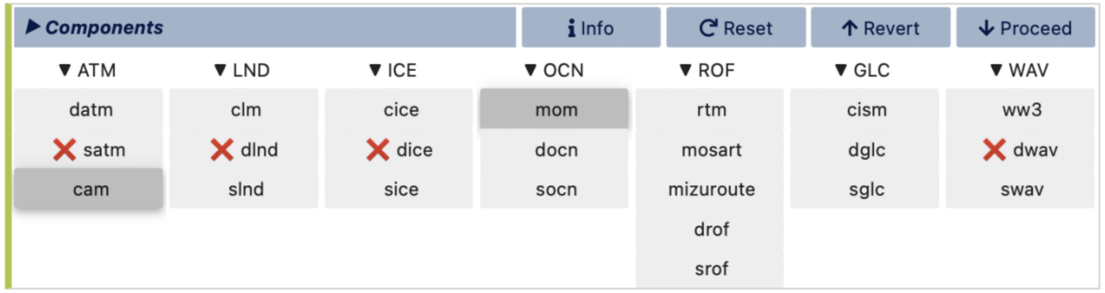
And thanks to **the small scope hypothesis:**

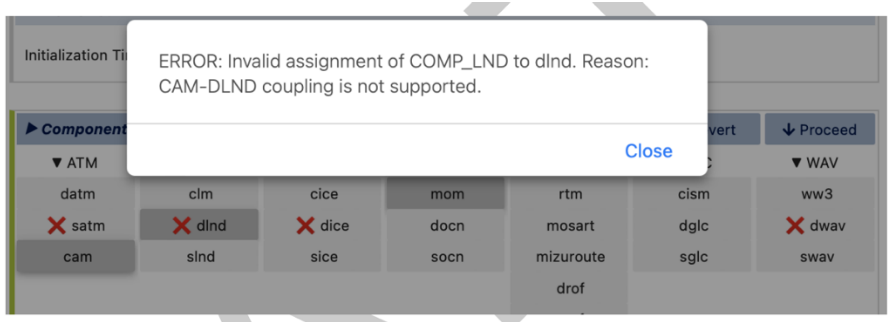
- A high proportion of bugs can be found by checking the program for ***all*** inputs within some small scope. (D. Jackson, 2006)

The Z3 Solver

We'll introduce the Z3 Solver, a popular formal methods tool used across many domains.

- **Bug finding & verification**: powers symbolic execution and static-analysis tools
- **Constraint solving**: finds configurations or schedules that satisfy complex interdependent rules
- **Compilers & optimizers**: LLVM-based tools to check that optimized code.
- **Tool development**:
  - We've used Z3 in **visualCaseGen**, a custom CESM case configurator: Z3 validates setting in real time by analyzing dependencies, detecting conflicts, and explaining incompatibilities.




ERROR: Invalid assignment of COMP_LND to dlnd. Reason: CAM-DLND coupling is not supported.

Formal Reasoning and AI

Tomorrow’s Keynote: “Lean into Verifiable Intelligence”

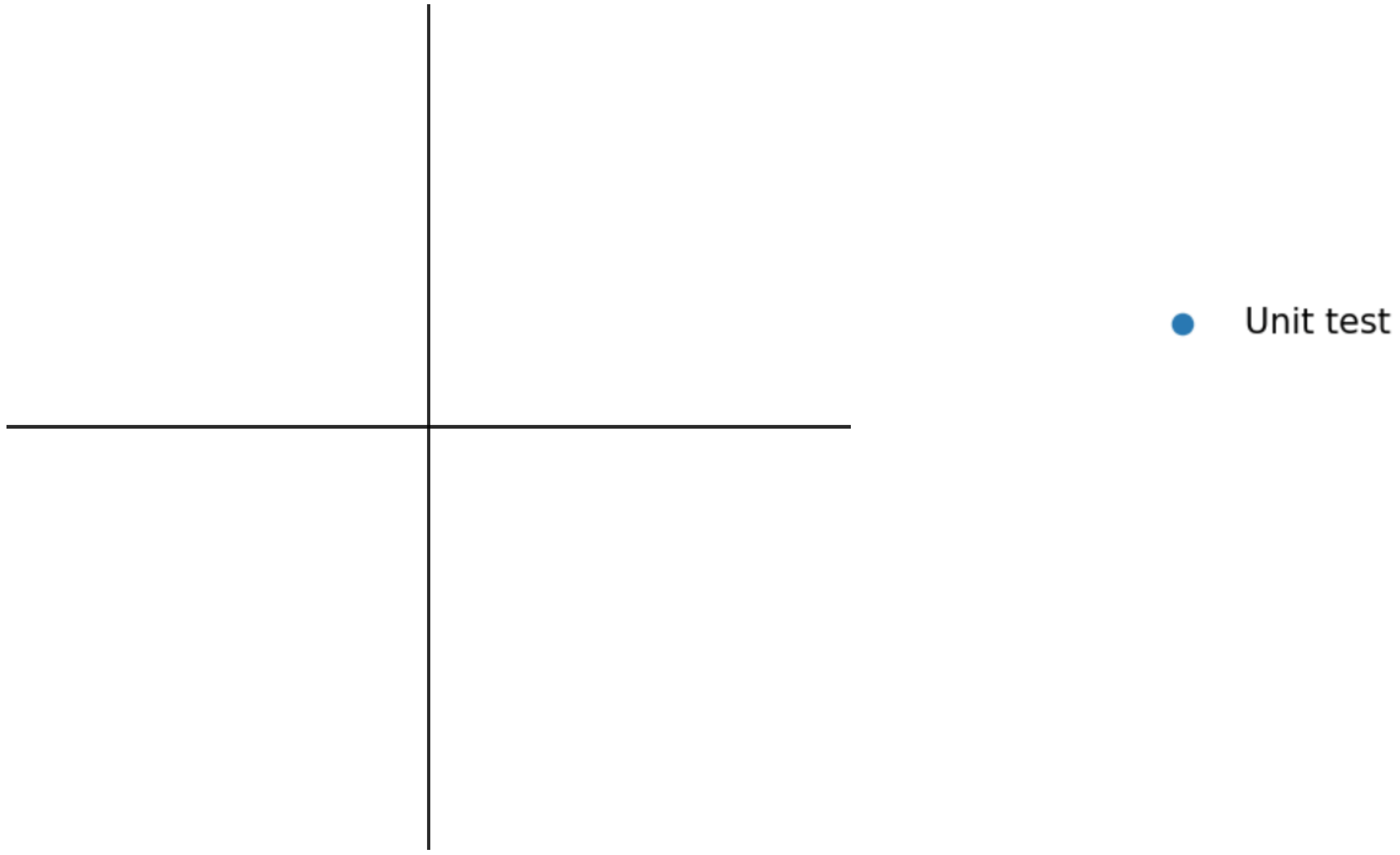
- A talk about Lean, another formal verification tool, and how AI and formal reasoning are coming together to make systems both smarter and more trustworthy.

Soonho Kong
(Amazon)

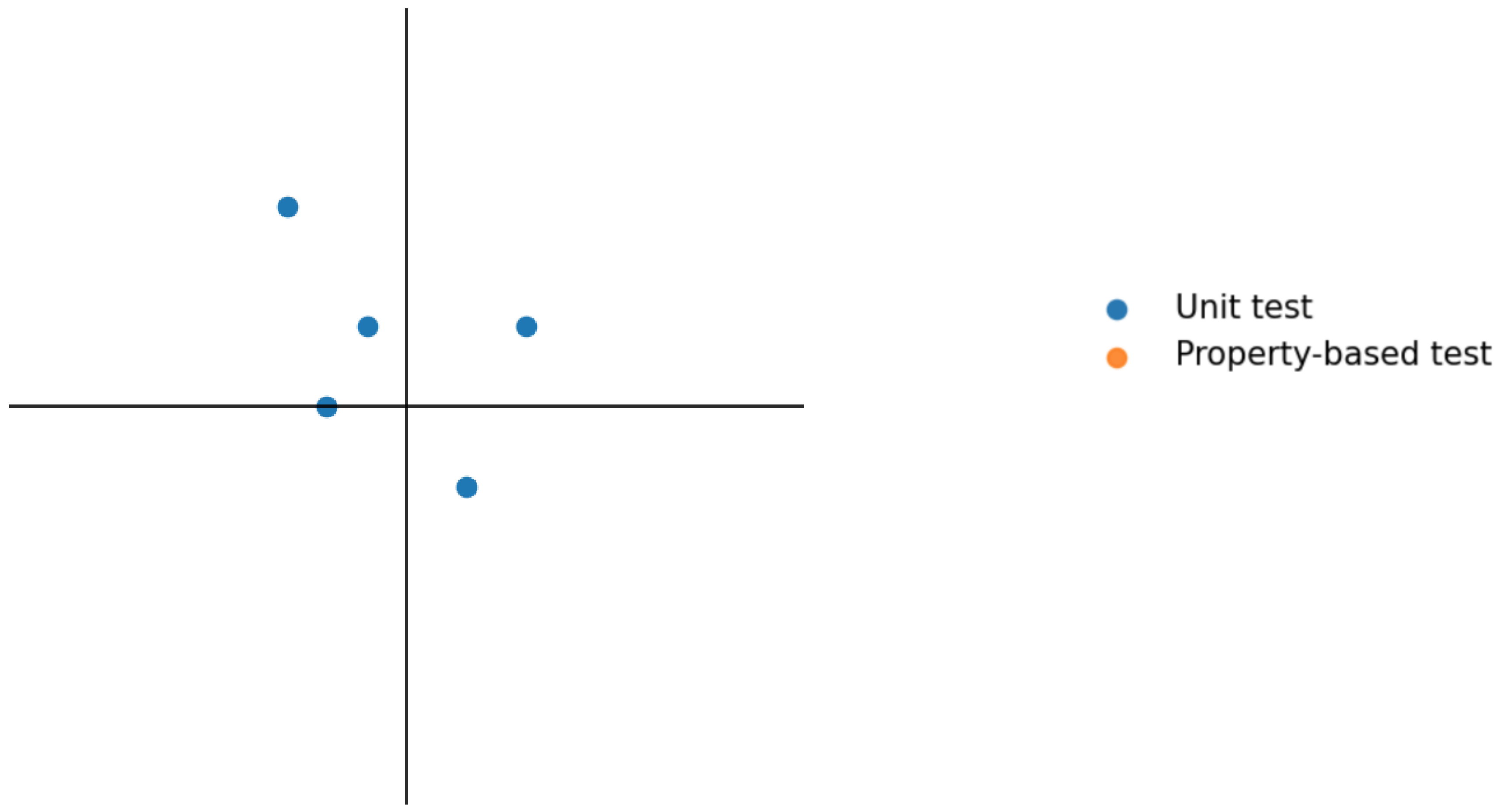
LEAN THEOREM PROVER

Final Words

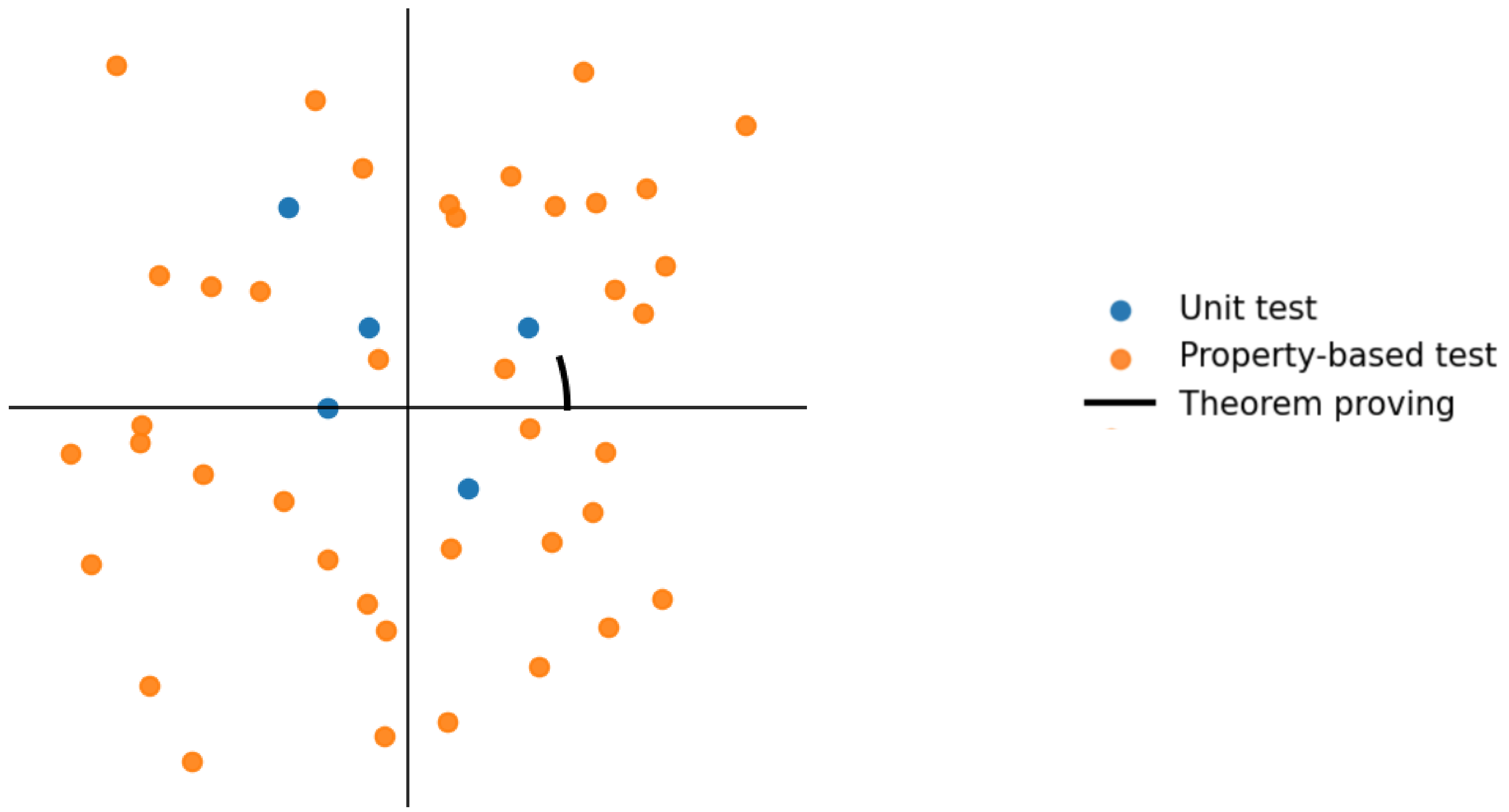
Coverage and Problem Sizes



Coverage and Problem Sizes



Coverage and Problem Sizes



No One-Size-Fits-All Solution

Each technique brings its own tradeoffs in effort, scope, and assurance.

- **Unit testing**: fast, focused, and essential
- **Property-based testing**: broad and exploratory
- **Formal Verification**: deep and exhaustive, but less practical

Choose based on your project’s needs and practical constraints.

Beyond Tools: A Scientific Mindset

- Start by thinking about the abstractions and properties your code should hold.
- Hypothesize about these properties through reasoning and specification.
- Attempt to refute your hypothesis through tests / verification.
- Incrementally refine your code and your understanding based on the results.

Does the extra work slow us down?

- Yes, for one-off applications.
- But for code that lives beyond a few versions, the effort pays back:
    - It enables quick inquiries, changes, and exploration of new ideas.

Thanks!

altuntas@ucar.edu

BSSw fellowship

Link to survey:

https://www.surveymonkey.com/r/RFQYLG6