# Writing Correct Code Without Straining Your Brain

Michael Mossey

# What do I mean by 'correct code'?

In a strict sense,

- Every input produces correct output
- Every execution path through the code is expected and proper
- Every state the program enters into is valid

Note: full coverage testing doesn't check every path

# A practical compromise

- Check as many cases as we can
- Structure and document code to make that easier
- Promote 'understanding as a whole'

# What we will cover:

The writing and inspecting processes (not testing)

"Fact 37" in "Facts and Fallacies of Software Engineering" by Robert Glass:

*Rigorous inspection can remove up to 90% of errors from a software product before the first test case is run.*

(and beneficial tradeoff of time: saves testing and debugging time)

Act I: checking cases

Act II: mindset that promotes understanding as a whole

# Act I: checking cases

# Mathematical proofs

Given that x & y are real, prove x^2 + y^2 >= 0

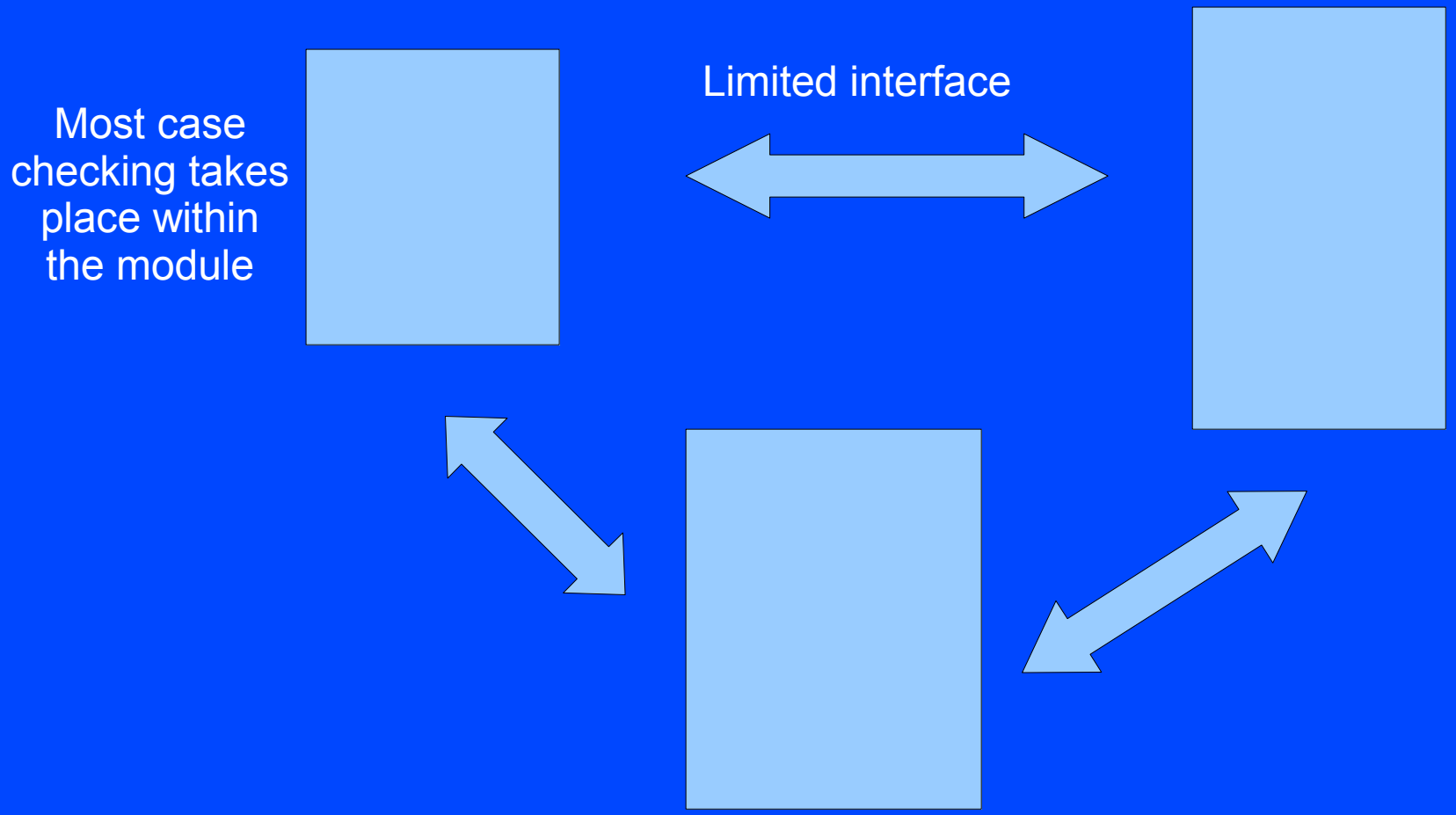
Tool 1: assumptions or invariants

X & Y are real; not complex

Tool 2: cases

- X < 0, Y < 0
- X >=0, Y < 0
- X < 0, Y >= 0
- X >= 0, Y >= 0

# Strategy: divide and conquer



# A function as one unit

- Check how it relates to the rest of the world
- Check how it handles cases within itself

# Relating to the rest of the world:

Chapter 6 of C++ FAQS:

A function specification is an external advertisement of what a function does.

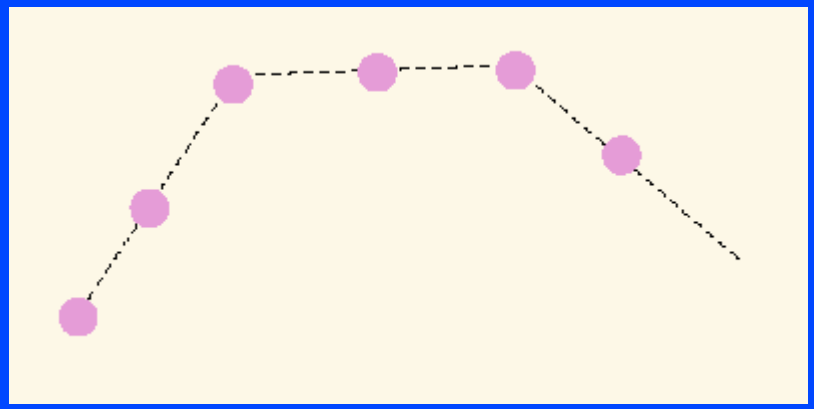
Consists of:

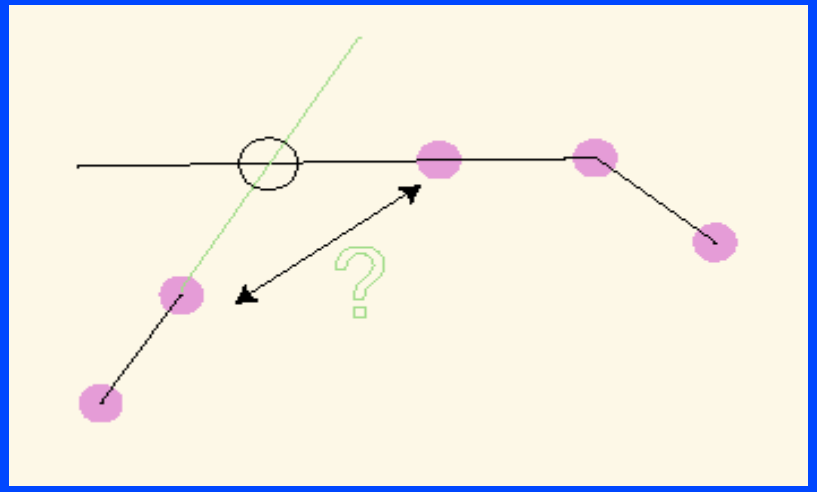
- Requirements
- Promises

# Inside a function

- Making cases explicit
- Loop invariants

# Making cases explicit: transmitter history and filling gaps





# Loop invariants

- help reduce number of cases to check

# The class as a unit

- in every member function, check that you've considered all class state during function processing, and properly set all state before exit (a good reason to have smaller classes)

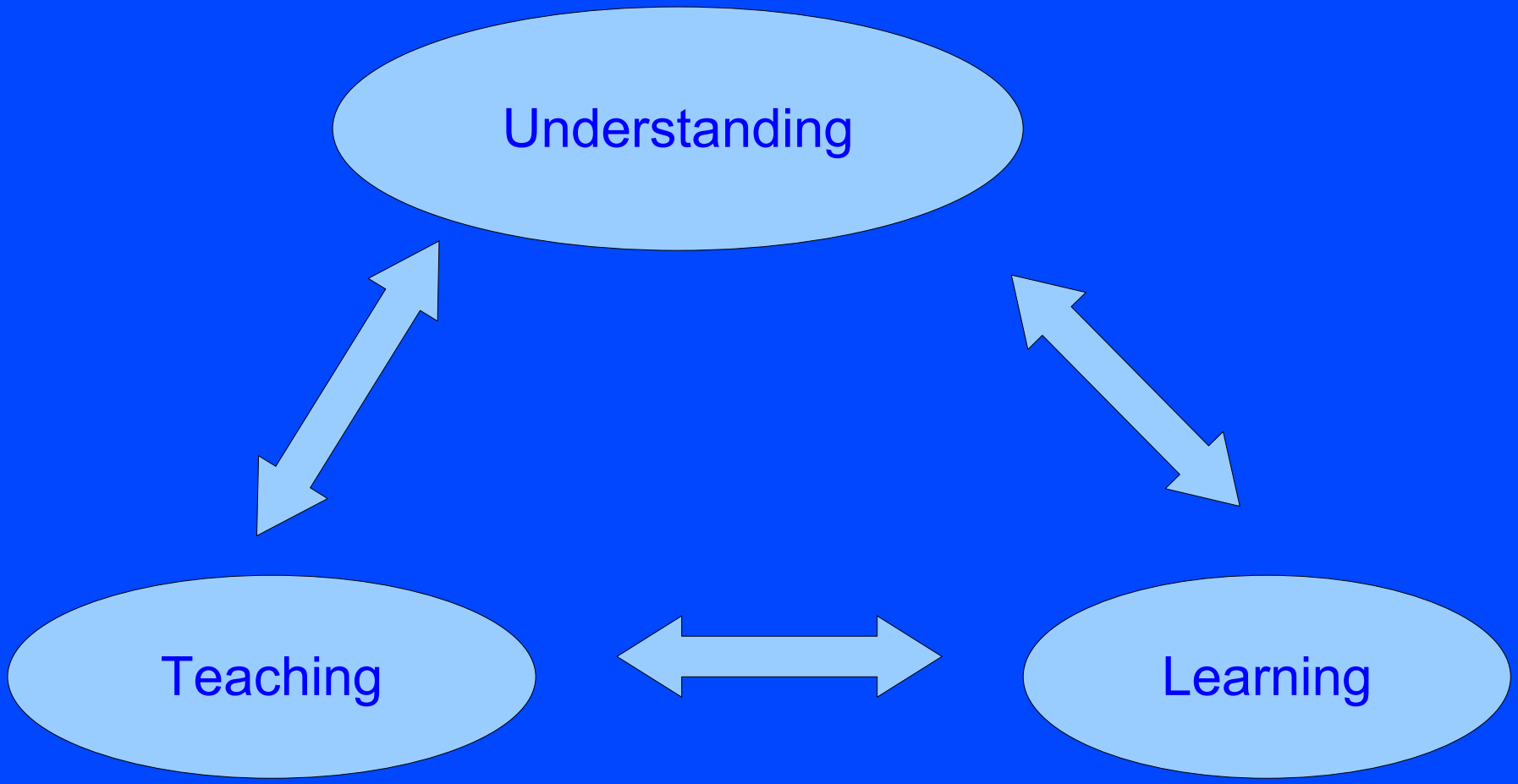
- class invariants (Chapter 10, C++ FAQS)

# Refactoring into smaller functions

- example from Nido

# Act II: the mindset

# The trinity



# Tennis lesson (Timothy Gallwey)

*The moral: start with the right perspective at the high level, and the details fit in easily.*

# Code as teacher

Code structure and documentation teaches the reader.

(Help the reviewer and the maintainer, not to mention yourself six months later.)

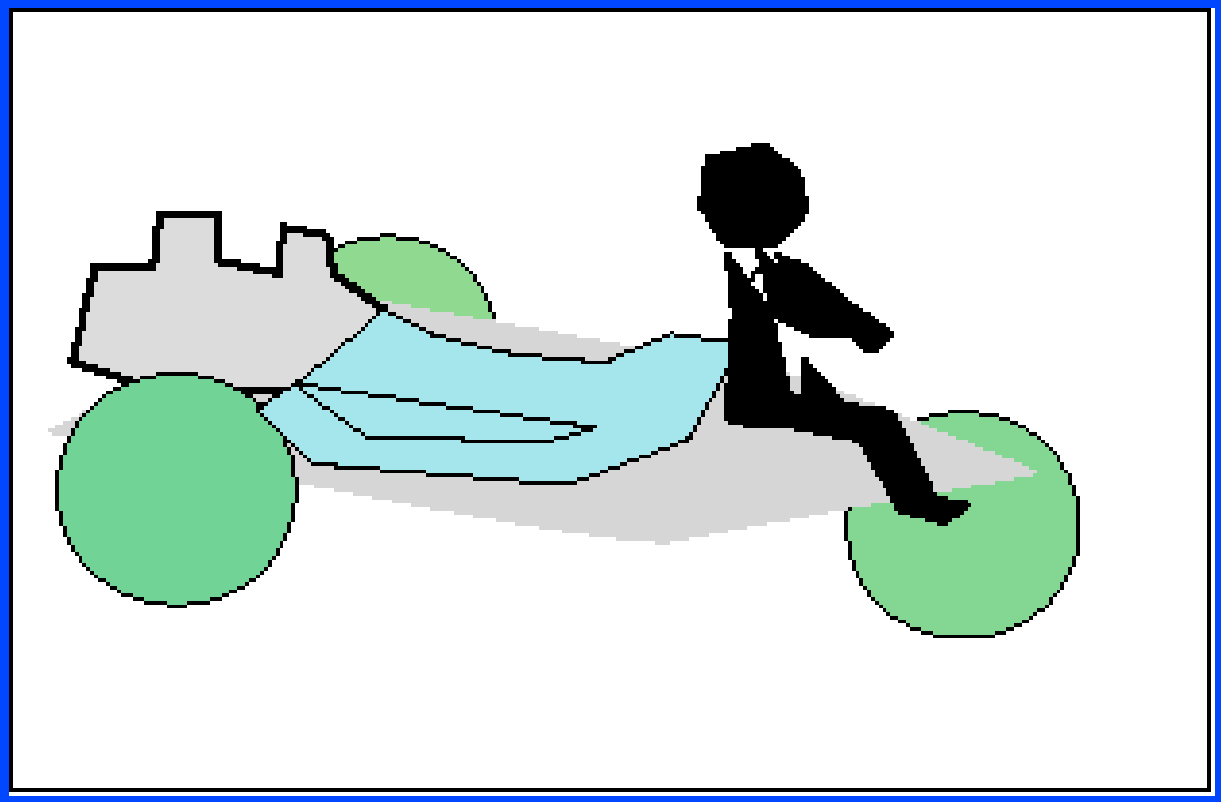
# Teaching example one

(teaching about vehicles like cars and trucks, a domain we already know something about.)

I'm going to describe a vehicle:

> The purpose of this vehicle is to carry heavy objects. It runs on wheels, and it has three of them. At the front of the vehicle sits the driver; behind him is the platform for holding the objects, and behind that is the engine.

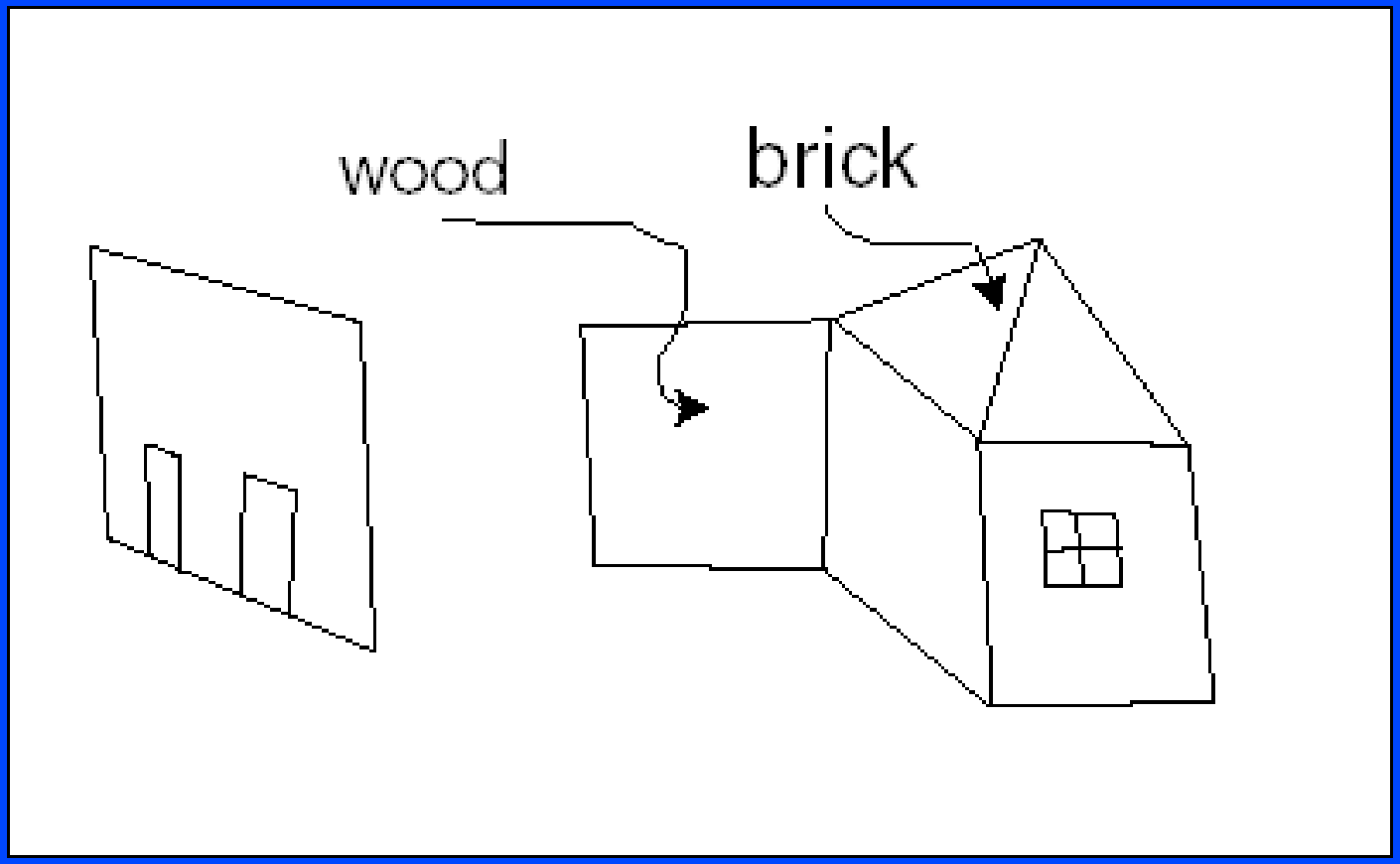
# What it looks like



# Teaching example two

I'm going to describe a building structure. It has four walls, two doors, and a window. It has a roof. It is made of brick and wood.

# What it looks like



# Two examples of class documentation

Example 1:

class FooOrganizer :
"This class is a container of pointers to Foo objects. This class organizes the Foo objects by size."

Example 2:

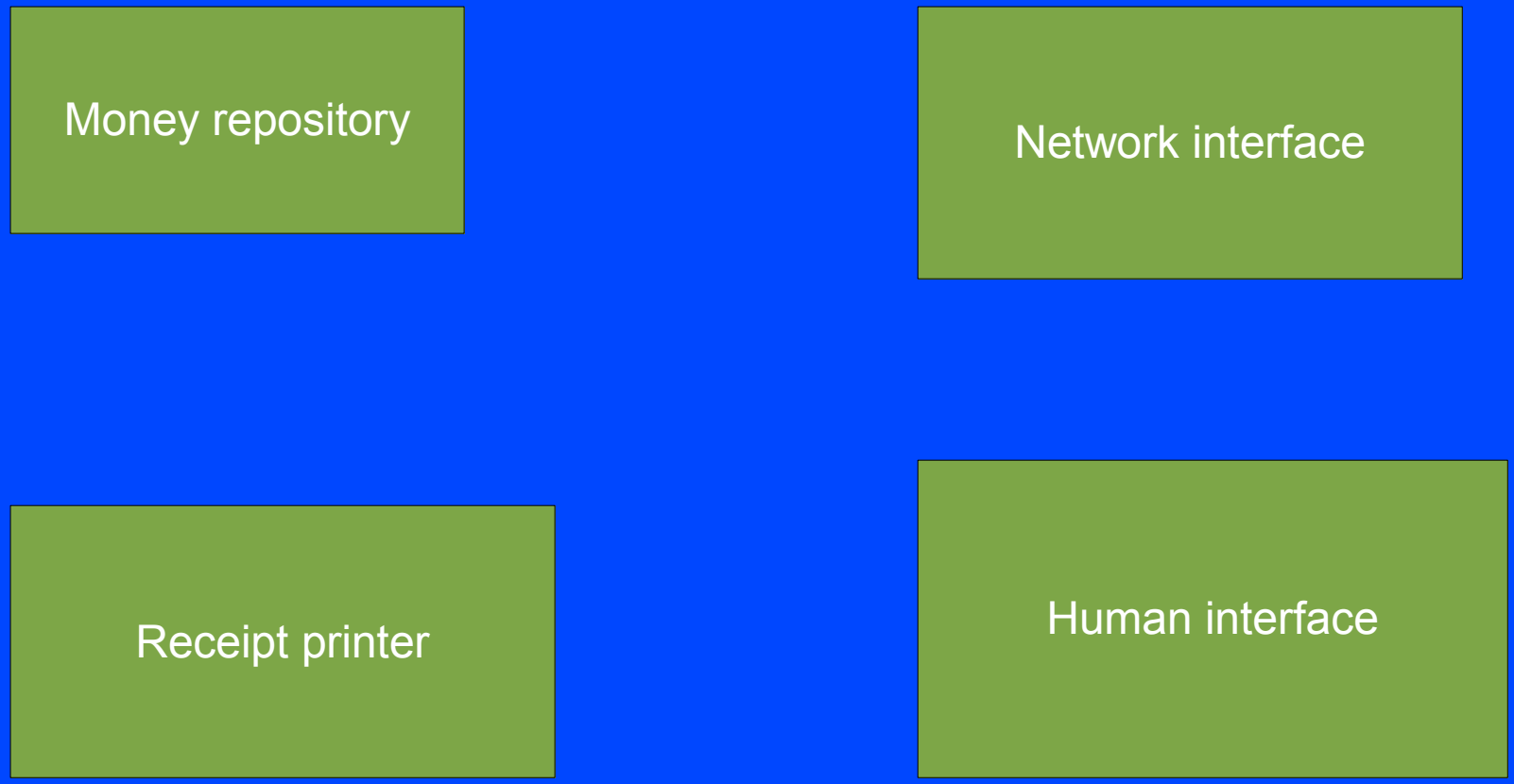
class FooSet :
"Call this with the size, it will return the nearest Foo object"

# Zen: “Beginner's Mind”

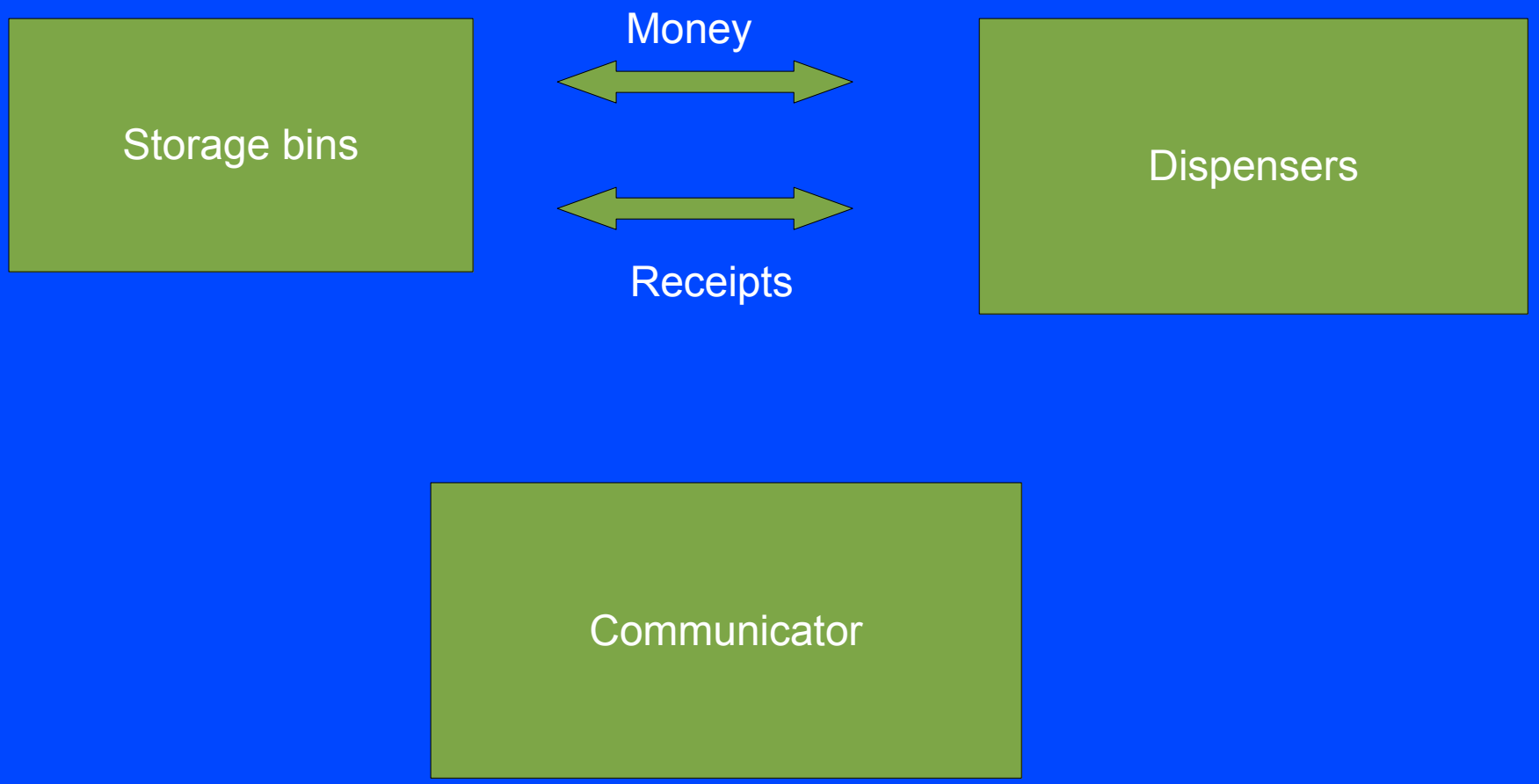
- write the manual first

- write the test cases first

# Design of an ATM machine



# Another ATM machine



# Recap:

- case checking, analogous to mathematical proving

    (known conditions, making cases explicit, specifications)

- refactoring into smaller functions

- mindset

    - think of your code as a teacher

    - find the right initial mindset