> Simon Walker
> June 7th, 2025

# Most important three of,

> - abstraction
> - encapsulation
> - polymorphism
> - coupling
> - cohesion
> - delegation

Such a difficult first question for the semester.

## The minimum for a coherent mathematical system:

**1. Abstraction**
Fundamental to math and logic as a whole. A system without abstraction in the general sense, immediately becomes too complex to grow. All science relies on abstraction; this is surely the most critical in an absolute sense.

**2. Encapsulation**
Now that we have a system minimally expressive, the next priority is organization, legibility, and extensibility. The ability to bundle logic/state is necessary.

**3. Cohesion**
Cohesion must be beneath encapsulation because cohesion could be thought of the degree to which a set of fields/methods were properly organized; it implies encapsulation. Strong cohesion is understandable code.

Abstraction simplifies and encapsulation/cohesion protect/organize.

## And,

**4. Coupling, delegation, & polymorphism**
Low coupling is more of a by-product of good abstraction & encapsulation. And the other two are techniques to implement the former.

# For example,

**Bad example**. This example has minimal abstraction.

```
water_level = 0
is_on = False

# Making coffee requires knowing all implementation details
if water_level < 200:
    print("Error: Not enough water")
    return

if not is_on:
    print("Powering on machine")
    is_on = True

print("Heating water...")
print("Brewing coffee...")
water_level -= 200
print("Enjoy your coffee!")
```

One could imagine even more basic, without variables having names, as that is an extremely primitive notion of "abstraction".

**Better example**. Reuse and clarity increase when we abstract logic into methods.

```
water_level = 0
is_on = False

def make_coffee():
    if not _check_water():
        print("Please add water first")
        return
    _power_on()
    _heat_water()
    _brew()
    print("Enjoy your coffee!")

def add_water(amount):
    if amount > 0:
        water_level += amount
        print(f"Added water. Current level: {water_level}ml")
    else:
        print("Invalid water amount")

# Implementation details hidden from user

def _power_on():
    if not is_on:
        is_on = True
        print("Machine powered on")

def _check_water():
    return water_level >= 200

def _heat_water():
    print("Heating water...")

def _brew():
    global water_level
    print("Brewing coffee...")
    water_level -= 200

# Usage
add_water(250)
make_coffee()
```

**Good example**. Encapsulated state and cohesive organization.

```
class CoffeeMachine:
    def __init__(self):
```

```python
        # Encapsulated state
        self._water_level = 0
        self._is_on = False

    # Cohesive public interface
    def make_coffee(self):
        if not self._check_water():
            print("Please add water first")
            return
        self._power_on()
        self._heat_water()
        self._brew()
        print("Enjoy your coffee!")

    def add_water(self, amount):
        if amount > 0:
            self._water_level += amount
            print(f"Added water. Current level: {self._water_level}ml")
        else:
            print("Invalid water amount")

    # Encapsulated implementation details
    def _power_on(self):
        if not self._is_on:
            self._is_on = True
            print("Machine powered on")

    def _check_water(self):
        return self._water_level >= 200

    def _heat_water(self):
        print("Heating water...")

    def _brew(self):
        print("Brewing coffee...")
        self._water_level -= 200

# Usage
machine = CoffeeMachine()
machine.add_water(250)  # state is protected
machine.make_coffee()
```