CS 4530: Fundamentals of Software Engineering

Module 3: Code-Level Design Principles

Jonathan Bell, Adeel Bhutta, Mitch Wand
Khoury College of Computer Sciences

© 2022 Released under the CC BY-SA license

Learning Objectives for this Lesson

- By the end of this lesson you should be able to:
  - Describe the purpose of our design principles
  - List 5 code-level design principles with examples
  - Identify some violations of the principles and suggest ways to mitigate them

Use Design As a Way of Communicating Organization

- Software systems must be comprehensible by humans
- Which humans?
  - The other members of your team
  - The folks who will maintain and modify your system
  - Management
  - Your clients
  - and ...
  - You, a week from now or 6 weeks from now

Use Design to Control Complexity

- Software systems must be comprehensible by humans
- Why? Software needs to be maintainable
  - continuously adapted to a changing environment
  - Maintenance takes 50–80% of the cost
- Why? Software needs to be reusable
  - Economics: cheaper to reuse than rewrite!

Three Scales of Design

The Structural Scale

- key questions: what are the pieces? how do they fit together to form a coherent whole?

The Interaction Scale

- key questions: how do the pieces interact? how are they related?

The Code Scale

- key question: how can I make the actual code easy to test, understand, and modify?

Today’s topic: design principles at the code scale

Coupling is the biggest source of complexity at the code level

- Two pieces of code are *coupled* if a change in one demands a change in the other.
- A coupling represents an agreement between the two pieces of code.
  - They may agree on:
    - names
    - order (e.g. of arguments)
    - meaning (e.g. meaning of data)
    - algorithms
- The more two pieces of code are coupled, the harder they are to understand and modify: you have to understand both to understand either of them.

There's a fancy word for this: *connascence* (meaning "born together")

More coupling means less readability, less modifiability

Five general-purpose design principles

Five General Principles
1. Use Good Names
2. Make Your Data Mean Something
3. One Method/One Job
4. Don't Repeat Yourself
5. Don't Hardcode Things That Are Likely To Change

Principle 1. Use Good Names

- The name of a thing is a first clue to the reader about what the thing means.
  - often, it's the only clue ☹
- Use good names for
  - constants
  - variables
  - functions/methods
  - data types

Use Good Names for Variables and Types

```
var t : number
var l : number
```

```
var temp : number
var loc  : number
```

```
var temp : Temperature
var loc  : SensorLocation
```

Use Good Names for Functions and Methods

```
function checkLine (line) : boolean
```

```
function LineIsTooLong (line) : boolean
```

Use Good Names for Functions and Methods

- Use noun-like names for functions or methods that return values, e.g.

```
let c = new Circle(initRadius)
let a = c.diameter()
```

- not:

```
let a = c.calculateDiameter()
```

- Reserve verb-like names for functions or methods that perform actions, like

```
table1.addItem(student1,grade1)
```

Principle 2. Make Your Data Mean Something

- You need to do three things:
  1. Decide what part of the information in the "real world" needs to be represented as data
  2. Decide how that information needs to be represented as data
  3. Document how to interpret the data in your computer as information about the real world

Example:

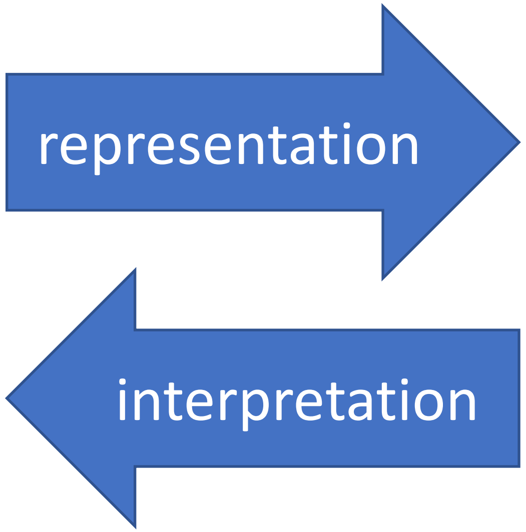
- Right now I am wearing a red shirt, and I've decided I need to represent that fact in my program.
- How should I represent that in my program?
- We need to decide:
  - how to represent shirts (including their color)
  - how to represent colors
  - how to represent **my** shirt

We need to write something like this:

```
type Shirt = {
    /** the color of the shirt */
    color: Color
}

type Color = { ... }

/** My shirt */
const myShirt: Shirt

myShirt.color = red
```

The Big Picture

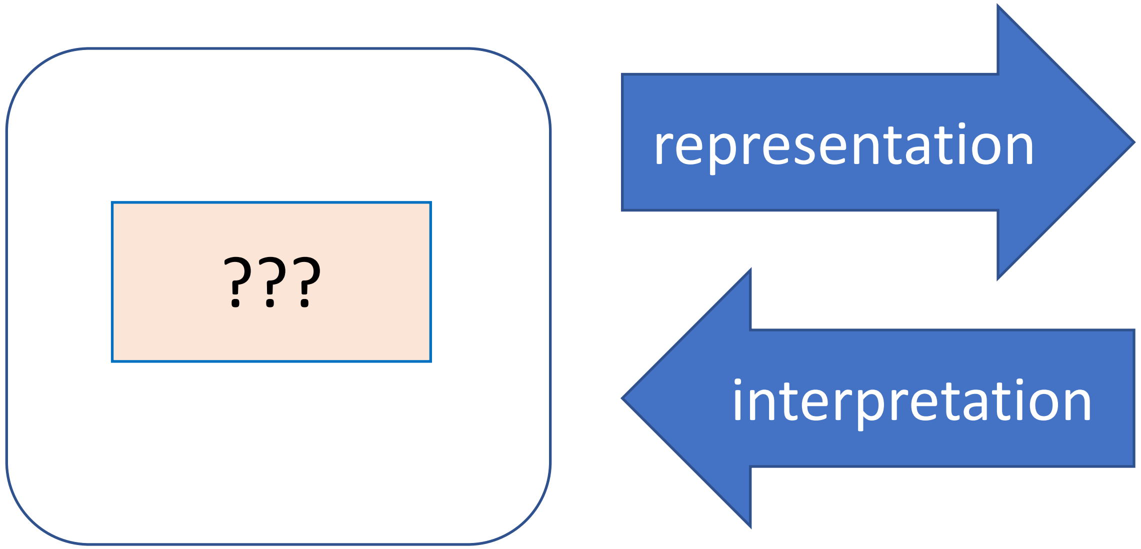
My shirt is red




```
type Shirt = {
    /** the color of the shirt */
    color: Color
}
type Color = { ... }

/** My shirt */
const myShirt: Shirt
myShirt.color = red
```

- How do we know that these are connected?
- Answer: we have to write it down.
- In our Typescript infrastructure, we do that with the comments.

Another Example: what do (x,y) mean?

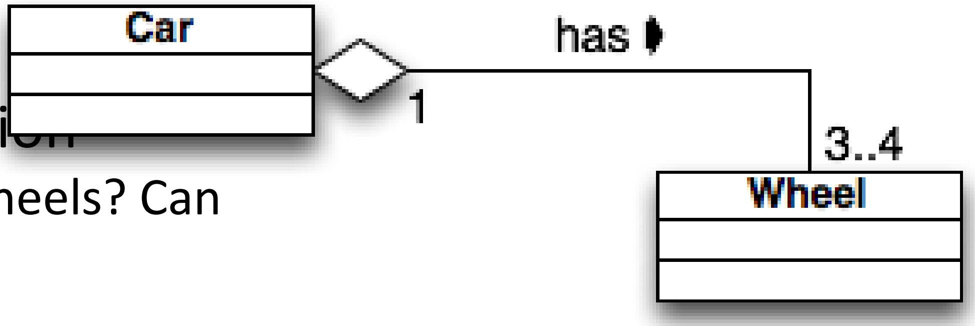



```
export interface BoundingBox {
    x: number;
    y: number;
    width: number;
    height: number;
};
```

- The center?
- Upper-left-hand corner?
- Does y grow in the up or down direction?
- And what about the units?
    - (Pixels? Scaled pixels? Something else?)

Another example: What does an object represent?

- What does an object of class Car represent?
  - a model of car (e.g. Dodge, Ford, Toyota)?
  - a particular car (my 2019 Toyota, VIN = 456789)?
- What does an object of class Wheel represent?
  - a model of tire? (Goodyear GoodGrips14)
  - a particular tire? (Goodyear GoodGrips14 SN = 345678)
- What does "has" represent?
  - depends on what Car and Wheel represent
  - this may affect the navigability of the association
    - (can you get from a car object to the associated wheels? Can you get from a wheel to the car that it's on?)



Principle 3: One Method/One Job

- Each class, and each method of that class, should have one job, and only one job
- If your method has more than one job, split it into 2 methods. Why?
  - You might want one part but not the other
  - It's easier to test a method that has only one job
- You call both of them if you need to.
  - or write a single method that calls them both
- Same thing for classes.

Principle 4: Don't Repeat Yourself

- If you have some quantity that you use more than once, give it a name and use the name.
- That way you only need to change it in one place!
- And of course you should use a good name
- If you have some task that you do in many places, make it into a procedure.
- If the tasks are slightly different, turn the differences into parameters.

A real example

```
function testequal <T> (testname: string, actualVal: T, correctVal: T) {
    test(testname,
        function () { expect(actualVal).toBe(correctVal) })
}

describe('tests for countOfLocalMorks', function () {
    testequal('empty crew',countOfLocalMorks(ship1),0)
    testequal('just Mork',countOfLocalMorks(ship2),1)
    testequal('just Mindy',countOfLocalMorks(ship3),0)
    testequal('two Morks',countOfLocalMorks(ship4),2)
    testequal('drone has no Morks',countOfLocalMorks(drone1),0)
})
```

Principle 5: Don't Hardcode Things That Are Likely To Change

- "No magic numbers" and "Don't Repeat Yourself" are already examples of this.
- General strategy: If there something that might change, give it a name
  - if it's not already a "thing", refactor to make it a "thing“
- Let’s look at a couple of examples.

Replace magic numbers with good names

- Replace magic numbers with good names

```
let salesprice = netPrice * 1.06
```

```
const salesTaxRate = 1.06
let salesPrice = netPrice * salesTaxRate
```

Example

- Imagine we are computing income tax in a state where there are four rates:
  - One on incomes less than \$10,000
  - One on incomes between \$10,000 and \$20,000
  - One on incomes between \$20,000 and \$50,000
  - One on incomes greater than \$50,000
- You might write something like

You might write something like

```
function grossTax(income: number): number {
    if ((0 <= income) && (income <= 10000)) { return 0 }
    else if ((10000 < income) && (income <= 20000))
    { return 0.10 * (income - 10000) }
    else if ((20000 < income) && (income <= 50000))
    { return 1000 + 0.20 * (income - 20000) }
    else { return 7000 + 0.25 * (income - 50000) }
}
```

- What might change?
  - The boundaries of the tax brackets might change
  - The number of brackets might change

So let's represent our data differently

```
// defines the tax bracket for income lower < income <= upper.
// if upper is null, then lower < income  (no upper bound)
type TaxBracket = {
    lower: number,
    upper: number | null,
    base : number
    rate : number
}

let brackets : TaxBracket[] = [
    {lower:0,     upper:10000, base:0,    rate:0},
    {lower:10000, upper:20000, base:0,    rate:0.10},
    {lower:20000, upper:50000, base:1000, rate:0.20},
    {lower:50000, upper: null, base:7000, rate:0.25}
]
```

And now it's easy to rewrite our function

```
// defines the incomes covered by a bracket
function isInBracket(income:number, bracket:TaxBracket) : boolean {
    if (bracket.upper == null)
    { return (bracket.lower <= income) }
    else
    { return ((bracket.lower <= income) && (income < bracket.upper))}
}

function income2bracket(income: number, brackets: Bracket[]): Bracket {
    return brackets.find(b0 => isInBracket(income, b0))
}

function taxByBracket(income:number,bracket:TaxBracket) : number {
    return bracket.base + bracket.rate * (income - bracket.lower)
}

function grossTax2 (income:number, brackets: TaxBracket[] ) : number {
    return taxByBracket(income,income2bracket(income,brackets))
}
```

Review: Learning Objectives for this Lesson

- You should now be able to:
  - Describe the purpose of our design principles
  - List 5 general design principles and illustrate their expression in code
  - Identify some violations of the principles and suggest ways to mitigate them

Additional Material

Examples of Design at the Structural Scale

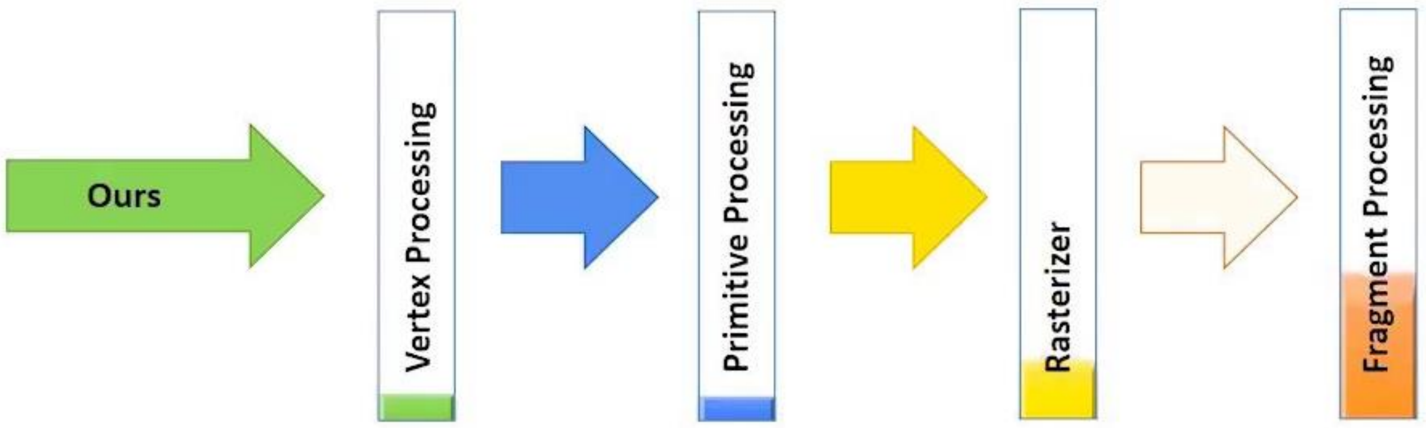
- Object-Oriented
- Pipeline
- Pipeline + Database
- Layered

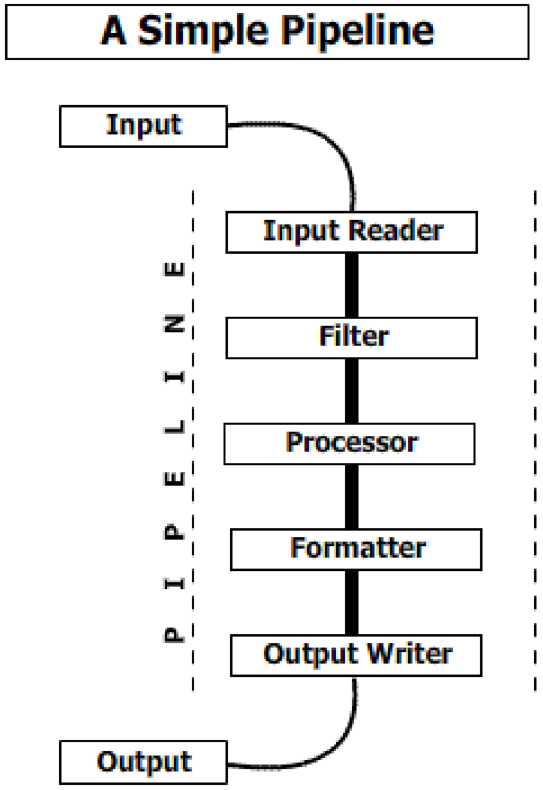
Object-Oriented Architecture

- The entities in the program correspond to entities in the real world.
- Example: a library system might have classes for
  - A holding (several books, eg: “7 copies of Moby-Dick”)
  - An individual item (“copy #3 of Moby-Dick”)
  - A card-holder (“Avery Fischer, library card #12345, …”)
  - A borrowing (“Avery Fischer borrowed copy #3 of Moby-Dick on 9/1/22”)

Pipeline Architecture

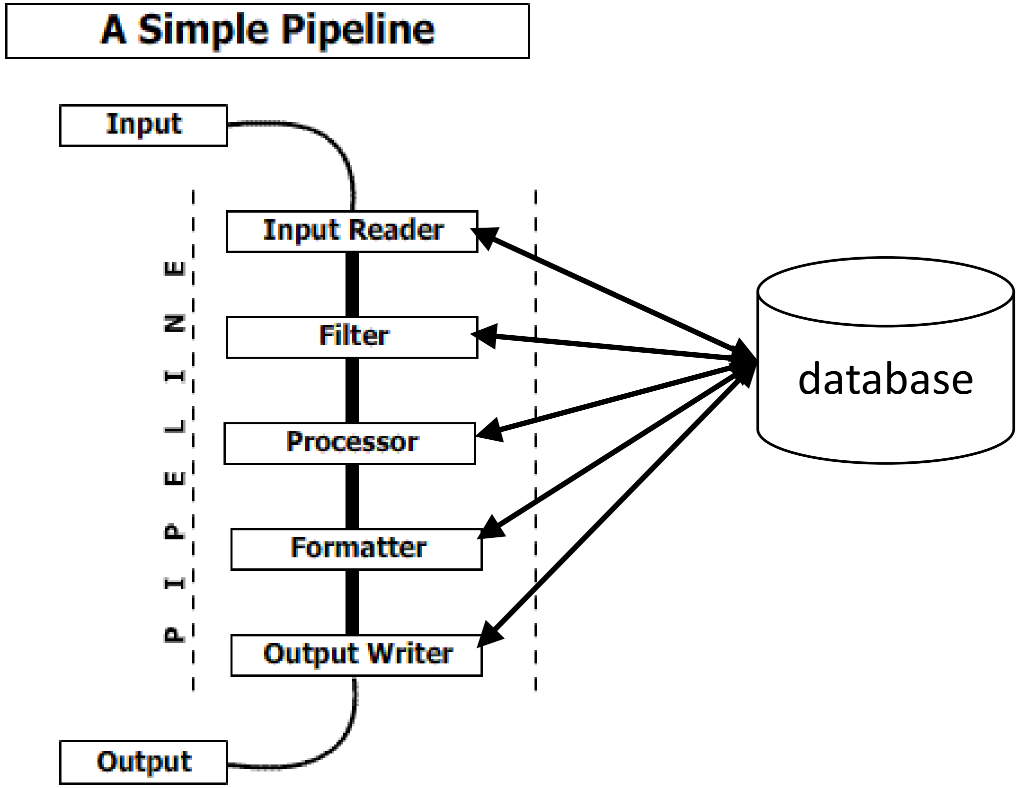
- The pieces correspond to stages in the transformation of data in the system
- Good for complex straight-line processes, e.g. image processing





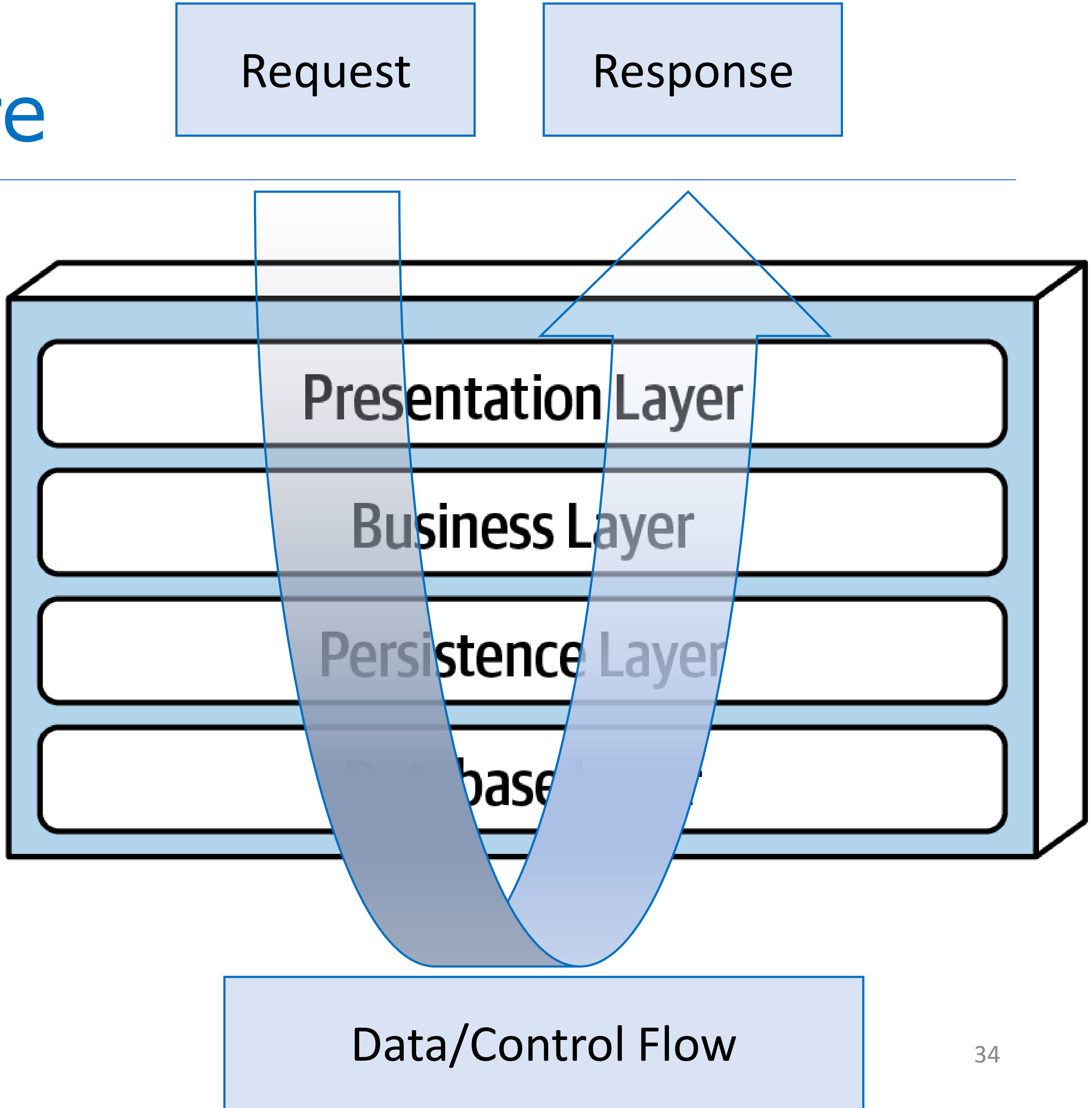
Pipeline + Database

- Stages in the pipeline share data through a database



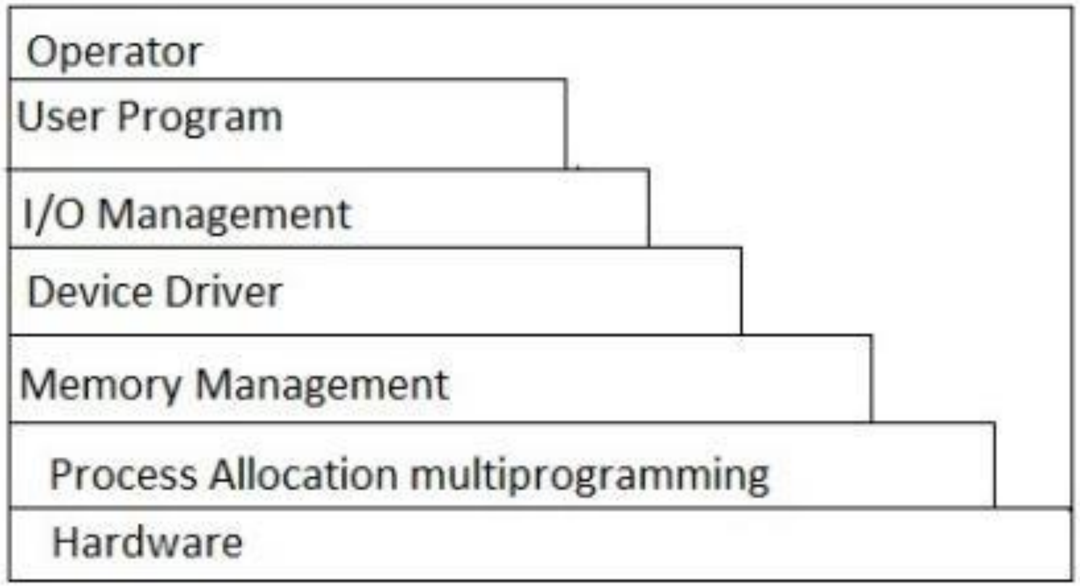
Layered Architecture

- The pieces correspond to level of concern.
- Each layer depends on services from the layer or layers below



Layered Architecture (contd)

- Typical organization for operating systems
- Layers communicate through procedure calls and callbacks (sometimes called "up-calls")



Design at the Interaction Scale

- Roughly what's typically called "Design Patterns"
- We'll talk about some OO Design Patterns in the next lecture.
- But we'll see interaction-scale patterns in many domains, not just OOP.