# CS 4530: Fundamentals of Software Engineering

# Module 04 Writing Readable Code: Code-Level Design Principles

Adeel Bhutta, Mitch Wand

Khoury College of Computer Sciences

## Learning Objectives for this Lesson

- By the end of this lesson you should be able to:
  - Describe the purpose of our best practices for code-level design
  - List 5 principles for designing readable code, with examples
  - Identify some violations of the practices 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 Achieve Non-Functional Qualities

- Readability – ease of reading code/understanding it
- Maintainability – ease of fixing defects
- Extensibility – ease of adding new components without changing existing ones
- Configurability – ease of changing behavior by end-users
- Testability – ease of writing tests
- Scalability – ease of improving performance with more computing resources

## 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!
- Consider design costs as an investment in reducing future costs

# Developers spend more time understanding code than writing it

- Study methodology:
  - Instrument 18 developer's IDEs for 740 development sessions
  - Record time spent in different tasks
- Conclusion: 70% of time spent in understanding-related tasks
- Other studies have shown similar statistics

Figure 3 summarizes the average distribution of activities of the developers and their sessions in our dataset.

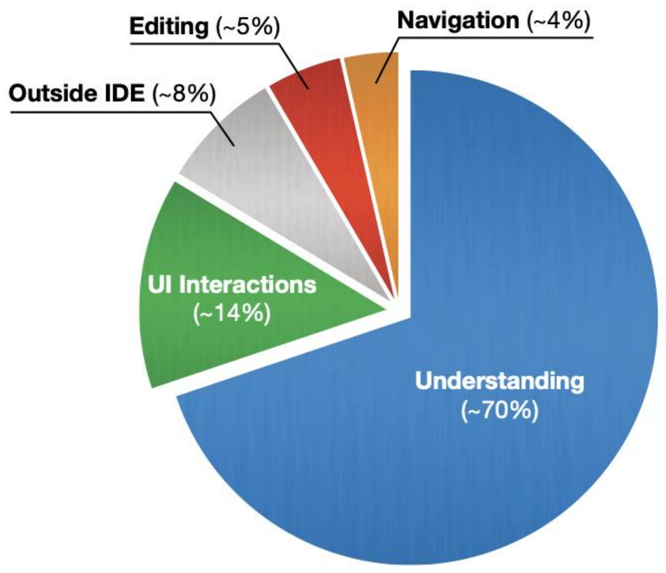


Fig. 3. How do developers spend their time?

Program understanding is as expected the dominant activity, but as we see our analysis attributes to it even more importance than what the common knowledge suggests, reaching a value of roughly 70%. This is a strong point in favor of the research

"I know what you did last summer: an investigation of how developers spend their time" Minelli, Mocci and Lanza. ICPC 2015

# 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

### The Structural Scale

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

### The Interaction Scale

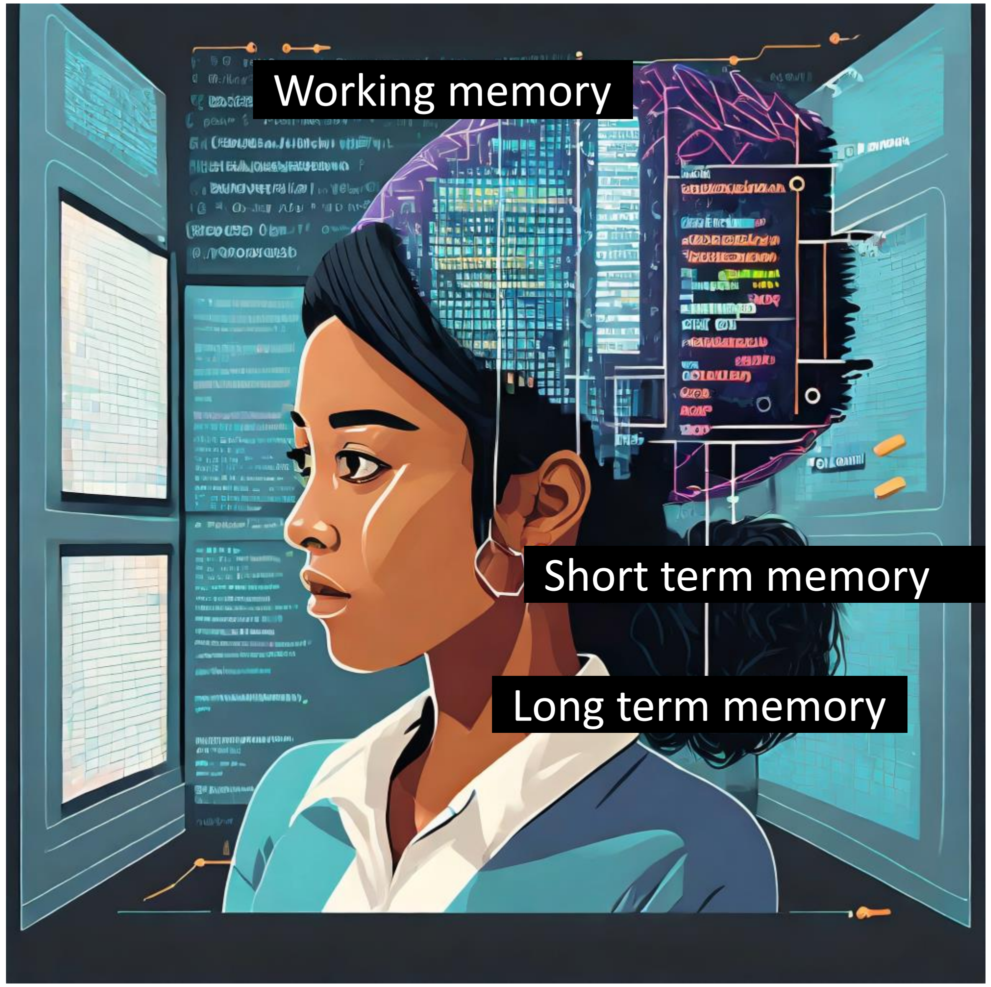
- key questions: how do the pieces interact? how are they related?

### The Code Scale

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

# Readability and Maintainability are Human Factors

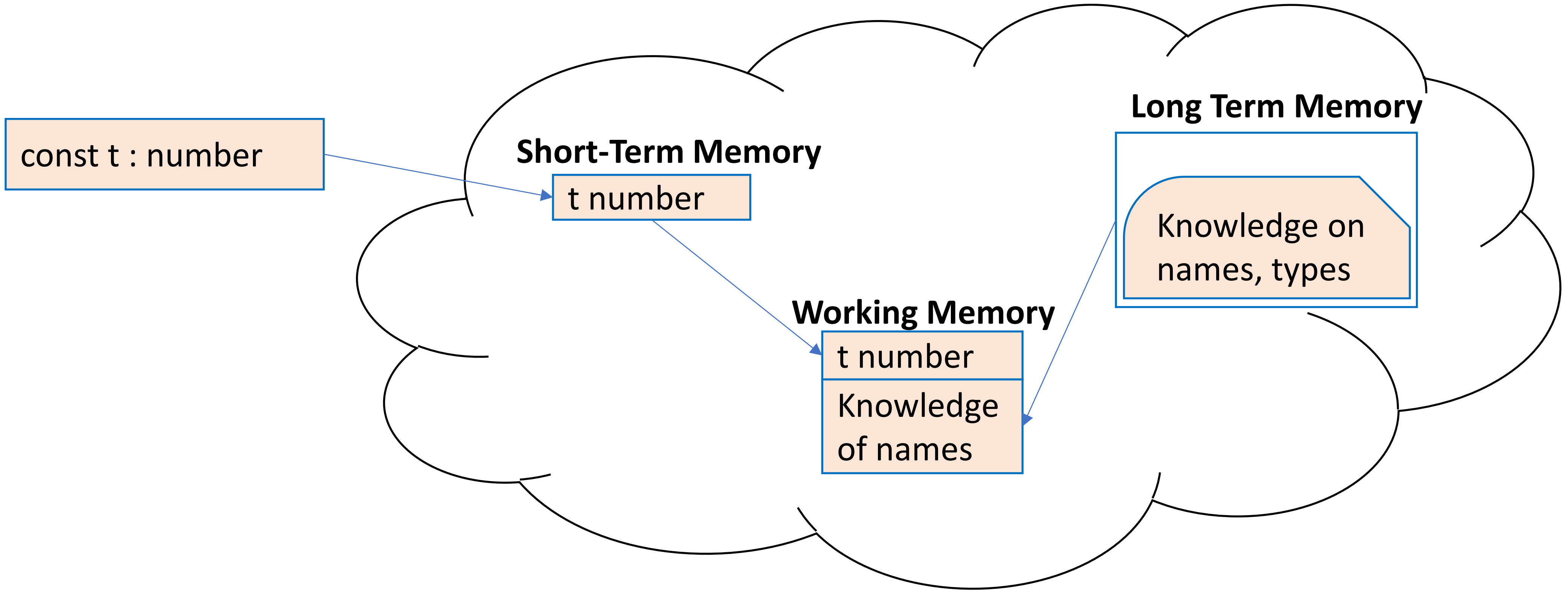
- Understanding code requires a developer to:
  - Access their long-term memory (e.g. for language syntax, API documentation)
  - Access their short-term memory (e.g. for tracking keywords, variable names, etc)
  - Manipulate new thoughts, ideas, etc in working memory
- Short-term memory capacity: 7 ± 2 items ("chunks")
- Long-term memory: unlimited capacity, much slower
- Working memory: where short-term and long-term memory interact to make decisions.



## Short-term memory capacity: 7 ± 2 chunks

- A chunk might be anything that has meaning
- What counts as a chunk is relative to concepts you already have
- The *number* of chunks in STM is fixed, but chunks might be different size.
- Example:
  - 6, 1, 7, 3, 7, 3, 2, 4, 6, 2   --10 chunks
  - 617-373-2462   -- 3 chunks

## Cognitive Load Example



https://theelearningcoach.com/learning/what-is-cognitive-load/

## Simple things can improve readability

```
function foo
    (x: number, y: number,
     shouldIncrement: boolean)
    : number {
    if (shouldIncrement)
      x++;
      x *= 2;
    x += y;
    return x;
  }
```

```
function foo
    (x: number, y: number,
     shouldIncrement: boolean)
    : number {
    if (shouldIncrement) {
      x++;
    }
    x *= 2; // correct indentation
    x += y;
    return x;
  }
```

## Five general-purpose design principles for understandability and maintainability

| 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
- Good names support chunking

## Use Good Names for Variables and Types

```
const t : number
const l : number
```

```
const temp : number
const loc  : number
```

```
const temperature      : number
const sensorLocation   : number
```

```
const temperature      : Temperature
const sensorLocation   : 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.

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

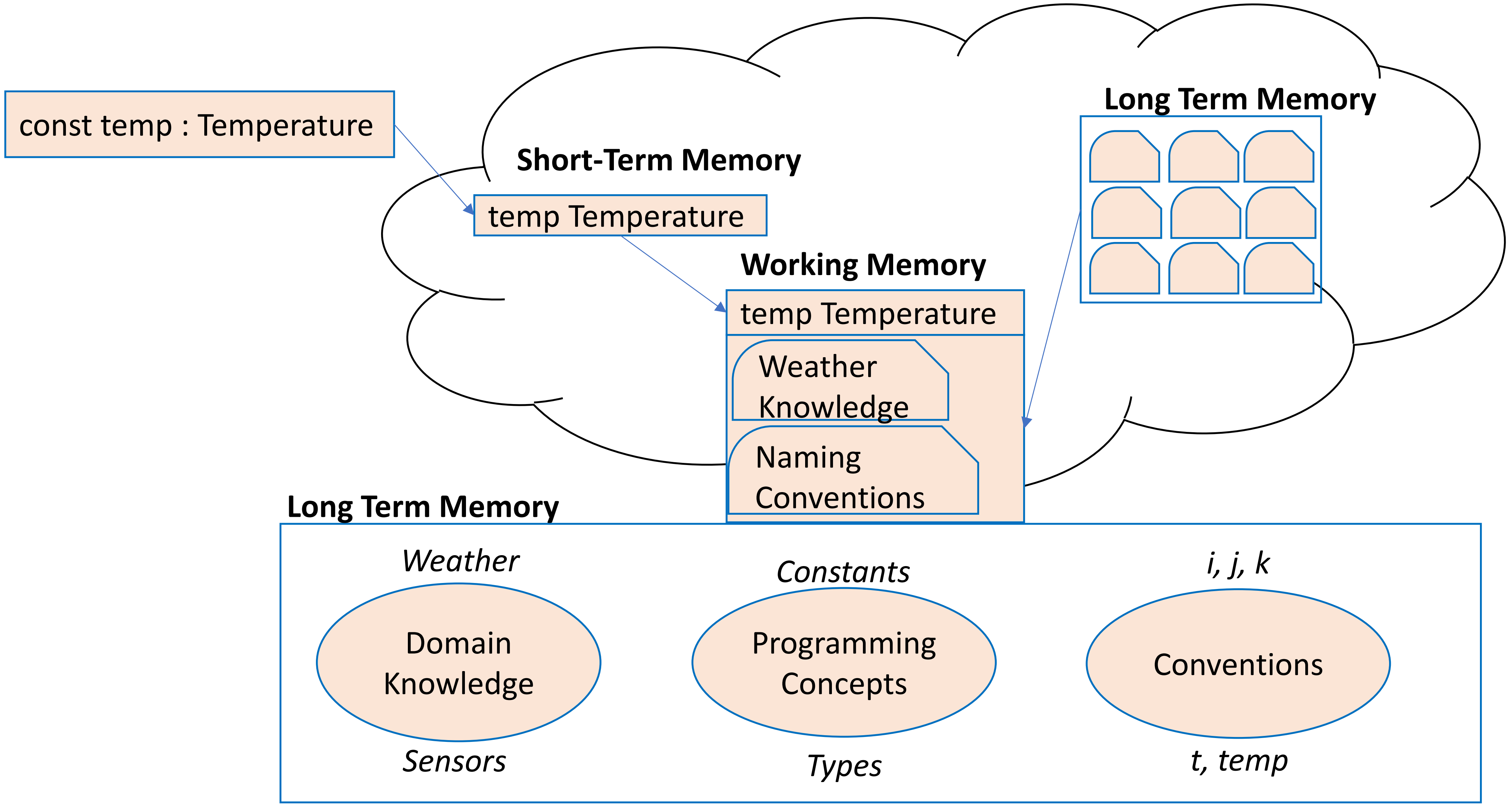
- not:

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

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

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

# Good Names Draw on Broad Long-Term Memory Knowledge



## Standardizing Names is More Important than the Standard Itself

- Using a naming convention *consistently* in a codebase improves understanding
- Consider standardizing:
  - snake_case vs camelCase vs PascalCase
  - Abbreviations in names vs words written out fully
  - Maximum name length
- For this course, we have some naming policies in our style guidelines
- We've covered some of these on the preceding slides.

## 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: Temperature Sensor

- We want to track the temperature of sensors in different locations
- How should we represent our sensor and its location?
- We need to decide:
  - How to represent sensors (including their temperature)
  - How to represent temperature and locations
  - How to represent a specific sensor, like one in the bathroom

## One Way of Representing a Temperature Sensor

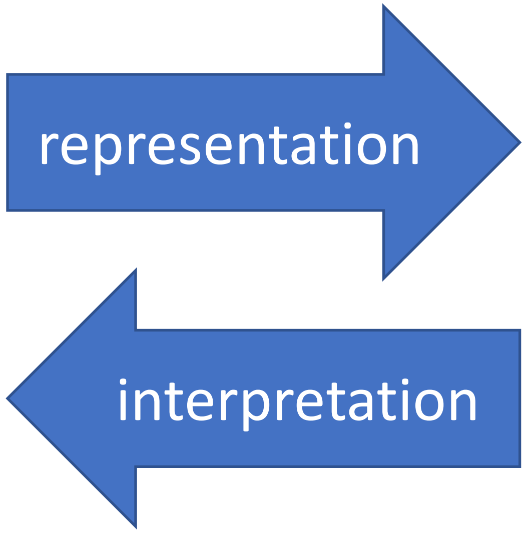
```
type Sensor = {
    location: SensorLocation
    currentTemperature?: Temperature
}

type Temperature = {
    degreesFahrenheit: number
}

type SensorLocation = "Basement" | "Bathroom" | "Kitchen";

let bathroomTemperatureSensor: Sensor = {
    location: "Bathroom"
}
```

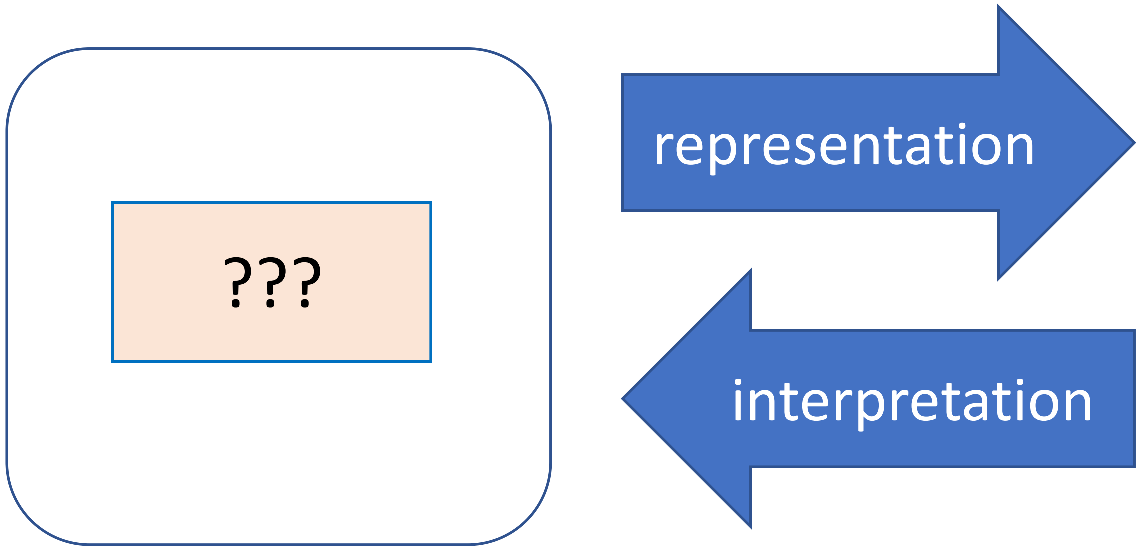
## The Big Picture



```
let bathroomTemperatureSensor: Sensor =
{
    location: "Bathroom"
}
```

- How do we know that these are connected?
- Answer: we have to write it down.
- In Typescript, we do that with names, types and comments.

## Another Example: what does (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?)

## 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.

## One Method/One Job Allows for Better In-Memory Chunking

- Recall: we have limited capacity in our short-term memory
- We get to remember “chunks”
- Splitting long methods into smaller ones with good names helps us hold more code in our short-term memory

## 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, () => { 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)
})
```

## A better example

```
function testLocalMorks(testName: string, ship: Ship, expected: number) {
    test(testName, () => {
        expect(countOfLocalMorks(ship)).toBe(expected)
    })
}

const testData = [
    { testname: 'emptyCrew', ship: ship1, expected: 0 },
    { testname: 'justMork', ship: ship2, expected: 1 },
    { testname: 'justMindy', ship: ship3, expected: 0 },
    { testname: 'twoMorks', ship: ship4, expected: 2 },
    { testname: 'droneNoMorks', ship: drone1, expected: 0 }
]

describe('tests for countOfLocalMorks', () => {
    testData.forEach(({ testname, ship, expected }) => {
        testLocalMorks(testname, ship, expected)
    })
})
```

## Beware of clones

- Terminology: “Code Clone” – a copy (or near-miss copy) of code within a codebase
- Clones are created when DRY is violated

```
int foo(int j){
  if(j<0)
        return j;
  else
        return j++;
}
```

```
int goo(int j){
  if(j<0)
        return j;
  else
        return j--;
}
```

```
int getOrOffset(int j,
boolean increment){
  if(j<0)
        return j;
  else {
        if(increment)
             return j++;
        else
             return j--;
  }
}
```

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

- 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“
  - Making it a “thing” makes it easier to understand!
- Let’s look at a couple of examples.

## A "thing" is something that may be used in many places.

- Replace magic numbers with good names

```
const salesPrice = netPrice * 1.06
```

```
const salesTaxRate = 1.06
const 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

```
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

## The tax bracket might be a "thing"

- We see there are four of them, but there might be more.
- They are likely to be used in many places
- A tax bracket has a lower bound, an (optional) upper bound, and a rate.
- Each of these may change, but they always represent the same thing

## So we can represent a tax bracket like this:

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

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

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

## And we'll also need to represent a tax table

```
// represents a tax table as an array of brackets.
// INVARIANT: ???
type TaxTable = Bracket[]


/** INVARIANT:
 * 1. the brackets are a non-overlapping partition of the positive real numbers
 * 2. the tax associated with the upper bound of a bracket is
 *    the same as the tax associated with the lower bound
 *    of the next higher bracket.
 */

/** NOTE:
 * because the brackets are a non-overlapping partition of the positive real numbers,
 * the brackets need not be sorted by lower bound.
 */
```

## And now it's easy to rewrite our function

```
// because the brackets are a non-overlapping partition of the
positive real numbers,
// there will be a unique bracket for each income, so the "as
Bracket" cast is safe. */
function income2bracket(income: number, table: TaxTable): Bracket {
    return table.find(b0 => isInBracket(income, b0)) as Bracket
}

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

## Review: Learning Objectives for this Lesson

- You should now be able to:
  - Describe the purpose of our best practices for code-level design
  - List 5 principles for designing readable code, with examples
  - Identify some violations of the practices and suggest ways to mitigate them