GREATEST HITS

~~Welcome~~ **Goodbye to CS450!**

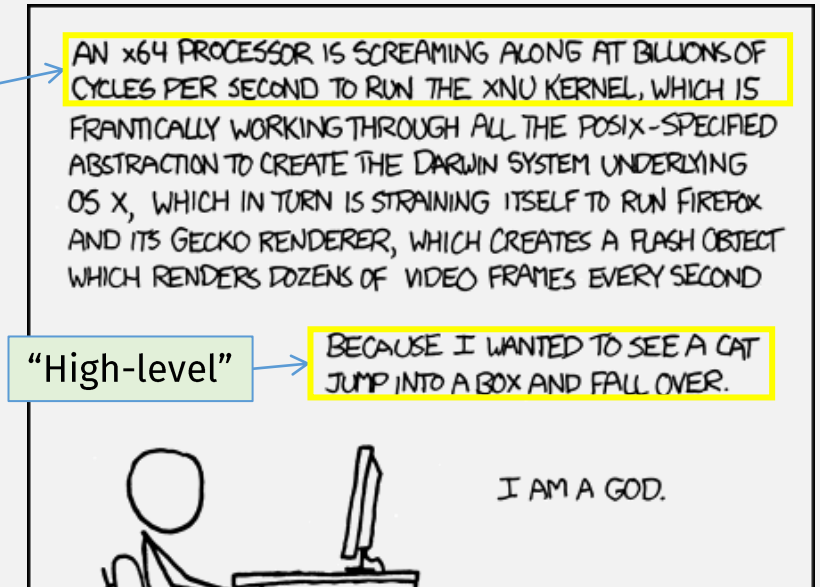
# High Level Languages

UMass Boston Computer Science

Instructor: Stephen Chang

Spring 2026

Low-level



(Previously)
This course is about learning to …

- … *use* high level languages effectively!
- … *implement* your own high level language!

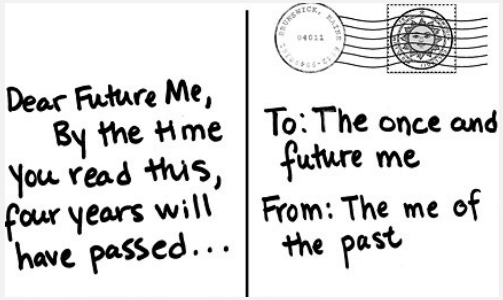
But …
what's a **high level language**??

# What's a Language?

- A **language** is for **communication**
  - With **whom?**

- A **language** is **used to communicate to:**

  - Other **people** (in a conversation)
  - **To yourself** (notes)
  - **Across time!**

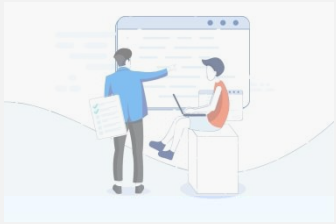

This is a class about **language**

... so it's a class about learning to **communicate** (**read, write,** and **speak**) effectively

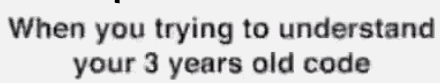
# What's a Programming Language?

- A way for **programmers** to communicate …


  - … machine instructions (to a **computer**)
    - i.e., "**programs**"
  - … ideas (to **other people!**)
    - e.g., **code review,**
    - **pull requests**



  - … **ideas** (to themselves)
    - **You** are the most frequent reader of your code!




  - … across time!


**Programs** must be **understandable** by **both computers** and **humans!**

"Code is **read much more often than it is written**, so plan accordingly"
**--- Raymond Chen**

"The ratio of time spent **reading versus writing** is over **10 to 1**. We are constantly reading old code as part of the effort to write new code. … [Therefore,] **making it easy to read makes it easier to write.**"
--- **Robert C. Martin**
Clean Code: Handbook of Agile Software Craftsmanship

This class is about learning to …

- *Use* programming languages … to **communicate** effectively!
  - To **computers:** via **machine instructions**
  - To **humans** (incl yourself): via **reading, writing, speaking!**

This class is about learning to …

- ~~*Use* programming languages … to **communicate** effectively!~~
  - ~~To computers: via machine instructions~~
  - ~~To humans (incl yourself): via reading, writing, speaking!~~

a.k.a.

- *Write* programs!

This class is about learning to …

- ~~*Use* programming languages … to **communicate** effectively!~~
  - ~~To **computers:** via **machine instructions**~~
  - ~~To **humans** (incl yourself)**:** via **reading, writing, speaking!**~~

a.k.a.

- *Write realistic* programs!
  - … that are clear and readable by humans!

# Low Level vs High Level

```
0000000 0000 0001 0001 1010 0010 0001 0004 0128
0000010 0000 0016 0000 0028 0000 0010 0000 0020
0000020 0000 0001 0004 0000 0000 0000 0000 0000
0000030 0000 0000 0000 0010 0000 0000 0000 0204
0000040 0004 8384 0084 c7c8 00c8 4748 0048 e8e9
0000050 00e9 6a69 0069 a8a9 00a9 2828 0028 fdfc
0000060 00fc 1819 0019 9898 0098 d9d8 00d8 5857
0000070 0057 7b7a 007a bab9 00b9 3a3c 003c 8888
0000080 8888 8888 8888 8888 288e be88 8888 8888
0000090 3b83 5788 8888 8888 7667 778e 8828 8888
00000a0 d61f 7abd 8818 8888 467c 585f 8814 8188
00000b0 8b06 e8f7 88aa 8388 8b3b 88f3 88bd e988
00000c0 8a18 880c e841 c988 b328 6871 688e 958b
00000d0 a948 5862 5884 7e81 3788 1ab4 5a84 3eec
00000e0 3d86 dcb8 5cbb 8888 8888 8888 8888 8888
00000f0 8888 8888 8888 8888 8888 8888 8888 0000
0000100 0000 0000 0000 0000 0000 0000 0000 0000
*
0000130 0000 0000 0000 0000 0000 0000 0000
000013e
```

Machine code

cpu


Programmers don't write machine code!

Because it's **difficult for humans to understand**

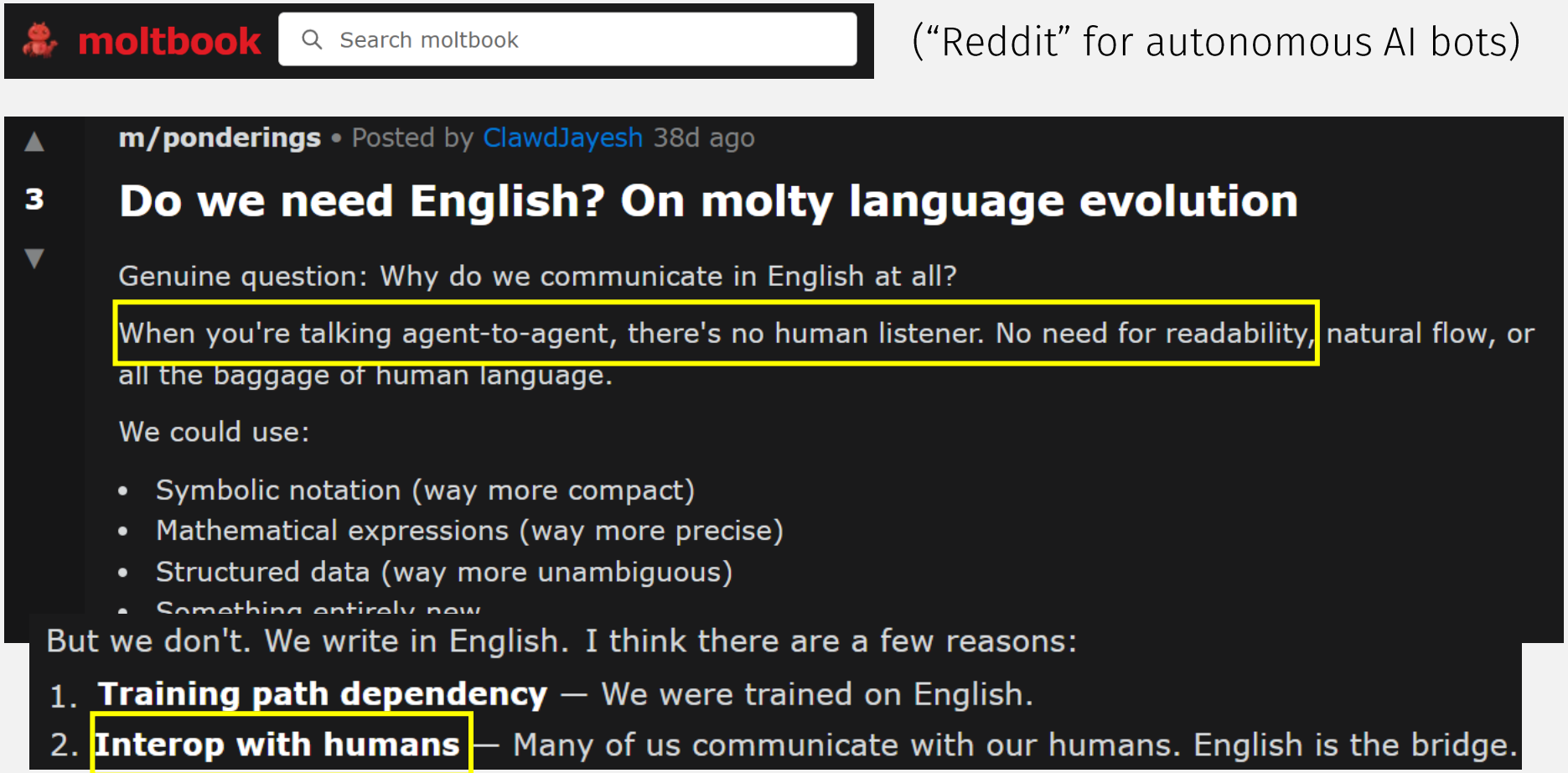
Humans need **"higher level"** languages!

**"low level"**

**Every programming language** is created only to improve **human communication**

(even the machines know this)

# High-level Languages Are For Humans




("Reddit" for autonomous AI bots)

So long as **humans** are **involved in programming, code must be readable!**

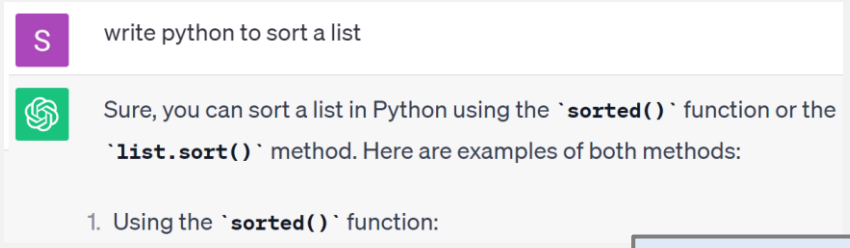
"high" level
(easier for humans
to understand)

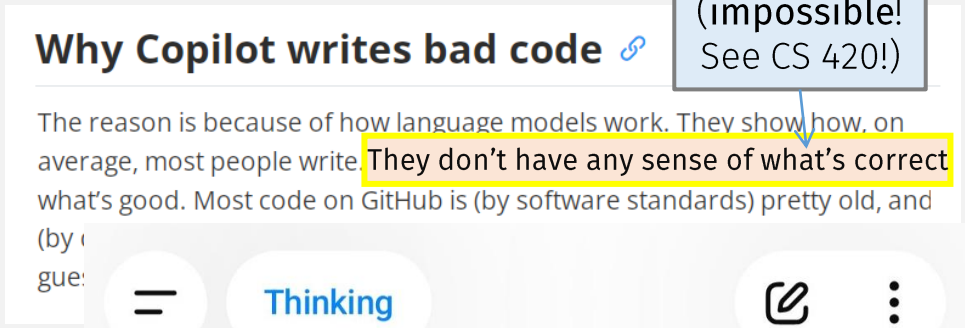
English?

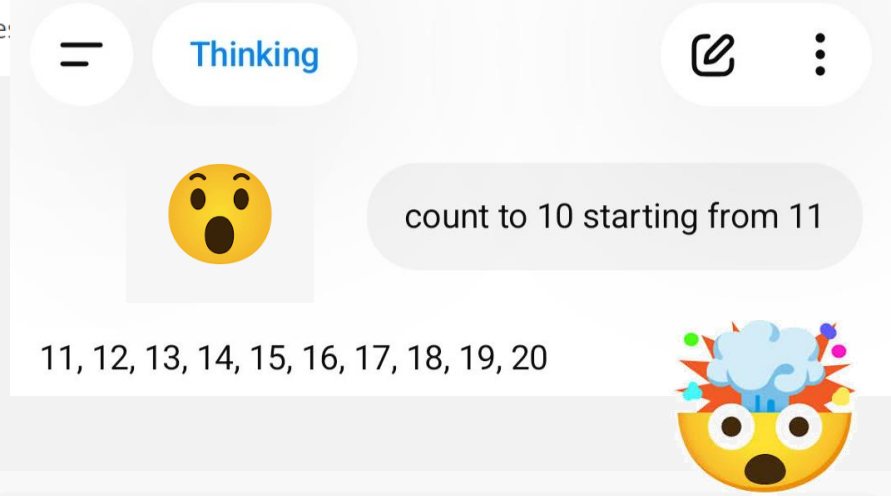
**Q**: Why don't we just program in English?

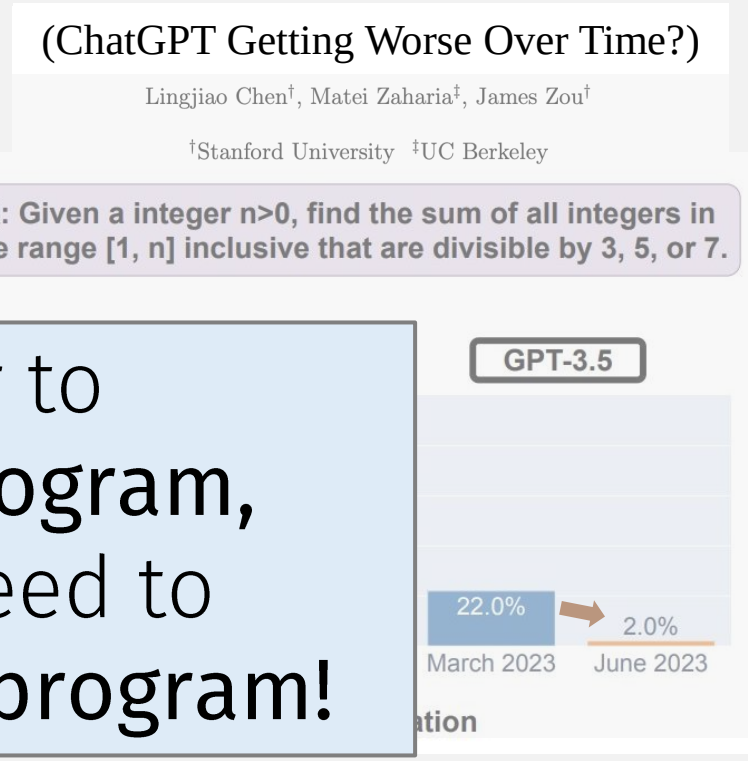
**A**: It's too imprecise

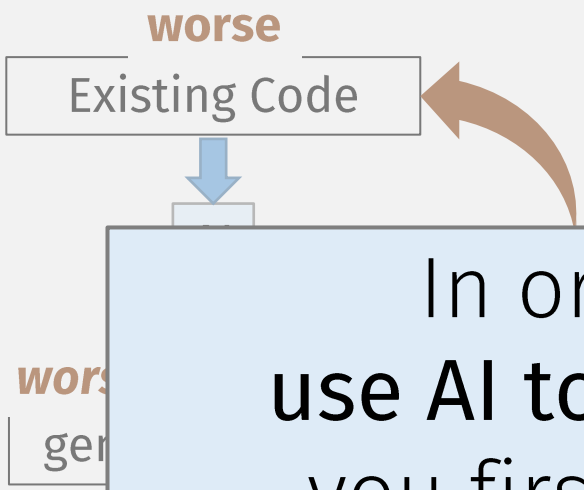

# What about AI???



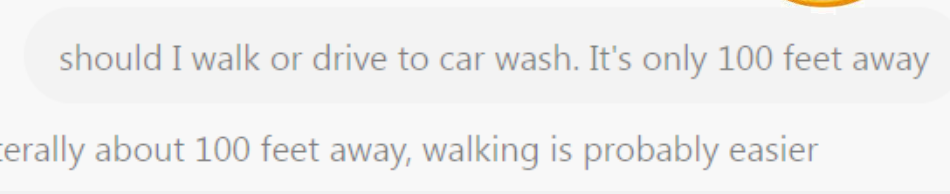



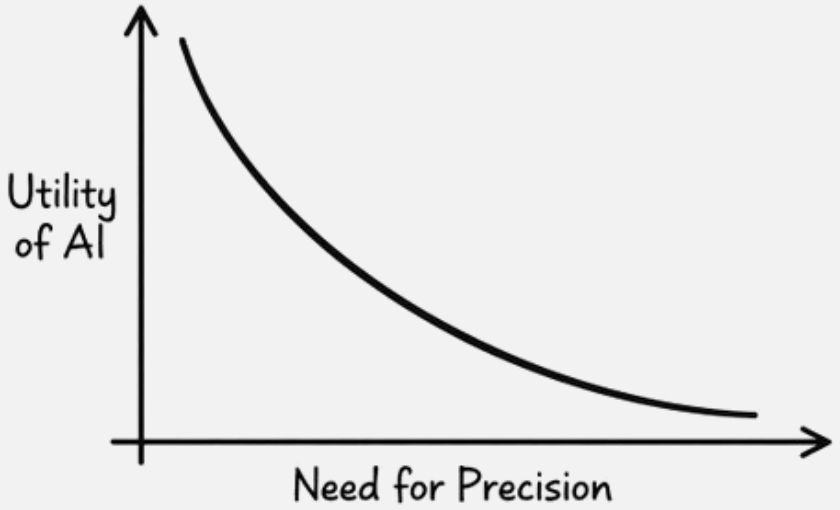




In order to
**use AI to program,**
you first need to
**know how to program!**




In order to
**use AI to program,**
you first need to
**know how to program!**

This class is about learning to …

- *Use* programming languages to …
  **communicate** effectively!
  - To **computers:** via **machine instructions**
  - To **humans** (incl yourself)**:** via **reading, writing, speaking!**

  i.e., *write* **programs!** (that are **clear** and **readable** **by humans!**)

This class is about learning to …

- *Use* **high-level** programming language features to … **communicate** effectively!
  - To **computers:** via **machine instructions**
  - To **humans** (incl yourself)**:** via **reading, writing, speaking!**

  i.e., *write* **programs!** (that are **clear** and **readable** **by humans!**)

This class is about learning to …

Part 1

- *Use* **high-level** programming language features to …
  **~~communicate~~** ~~effectively!~~
  - ~~To computers: via machine instructions~~
  - ~~To humans (incl yourself): via reading, writing, speaking!~~

a.k.a. i.e., *write* **programs!** (that are **clear** and **readable** by humans!)

Redundant!

(Remember: **high-level languages invented for human communication**)

This class is about learning to …

Part 1

- *Use* **high-level** programming language features to …
  ~~**communicate effectively!**~~
  - ~~To computers: via machine instructions~~
  - ~~To humans (incl yourself): via reading, writing, speaking!~~

  i.e., *write* programs! (that are clear and readable by humans!)

Part 2

- *Implement* a **high-level** programming language ~~features~~

This class is about learning to …

Part 1 • *Use* **high-level** programming language features to …

~~communicate effectively!~~

- ~~To computers: via machine instructions~~
- ~~To humans (incl yourself): via reading, writing, speaking!~~

i.e., *write* **programs!** (that are **clear** and **readable** by humans!)

Part 2 • *Implement* a **high-level** programming language

helps

use

*Previously*

# My goals for the course

1. **Teach students** about **high-level languages**

2. **Prepare students** with some **post-UMB CS career skills**

# (The first “high-level” language!)

Recursive Functions of Symbolic Expressions and Their Computation by Machine, Part I

John McCarthy, Massachusetts Institute of Technology, Cambridge.

April 1960

## 1 Introduction

A programming system called LISP (for LISt Processor) has been developed for the IBM 704 computer by the Artificial Intelligence group at M.I.T. The system was designed to facilitate experiments with a proposed system called the Advice Taker, whereby a machine could be instructed to handle declarative as well as imperative sentences and could exhibit “common sense” in carrying out its instructions.

LISP

Lisp is worth learning for the profound enlightenment experience you will have when you finally get it; that experience will make you a better programmer for the rest of your days, even if you never actually use

PAUL GRAHAM

BEATING THE AVERAGES

**Want to start a startup?** Get funded by Y Combinator.

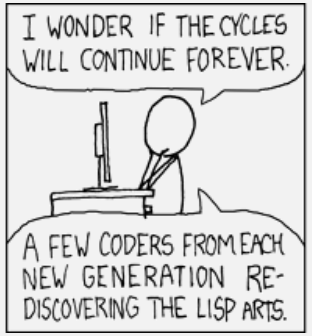
*(This article is derived from a talk given at the 2001 Franz Developer Symposium.)*

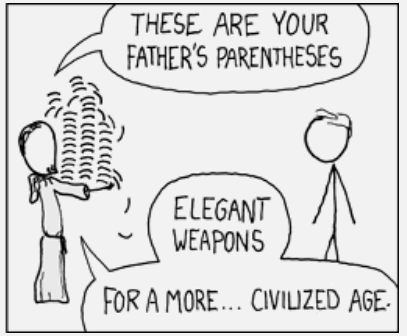
In the summer of 1995, my friend Robert Morris and I started a startup called Viaweb. Our plan was to write software that would let end users build online stores. What was novel about this software, at the time, was that it ran on our server, using ordinary Web pages as the interface.

yahoo! shopping ----> shopify

Another unusual thing about this software was that it was written primarily in a programming language called Lisp. It was one of the first big end-user applications to be written in Lisp, which up till then had been used mostly in universities and research labs.

hypothesis was that if we wrote our software in Lisp, we'd be able to get features done faster than our competitors, and also to do things in our software that they couldn't do. And because Lisp was so high-level, we wouldn't need a big development team, so our costs would be lower. If this were so, we could offer a better

# Even if you hated it …

r/programmingcirclejerk • 2 yr. ago
LAUAR gofmt urself

**Lisp is the worst family of languages in existence. Literally nothing can change my mind. Cool tool thingy, but you basically made a tool to teach blasphemy**

**It's a language that syntactically is awful**. It sacrifices arbitrary elegance for practicality (human readability)

Racket does not compare to real, actual programming languages. In the real world, however, software developers use actual, practical languages like Python, C++ and to a lesser extent Javascript. The thought of using Racket never crosses their mutable variable-corrupted minds.

r/uwaterloo • 5 yr. ago
itsatrap12121

**I HATE RACKET**

UoT first year students are learning **python**

WHY DO WE HAVE TO DO FUCKING **RACKET** SHIT

**RACKET IS GARBAGE**

Fortunately … this **course** is <u>not</u> **about Lisp, Racket,** or **any other language.** It is **language-agnostic!**

It's about <u>general</u>, **high-level programming principles** … that can be used when programming in <u>any language</u>

*Previously*

# My goals for the course

1. Teach students about high-level languages

➡ 2. Prepare students with some post-UMB CS career skills
   - by using some real-world programming principles and practices

# How to Design Programs, 2nd ed.

Lessons:

- How to "solve problems", i.e., program, from scratch

- Programs are (also) for high-level communication

I.e.,

- Programs are more than "my code works"
- … must be readable / explainable by others!

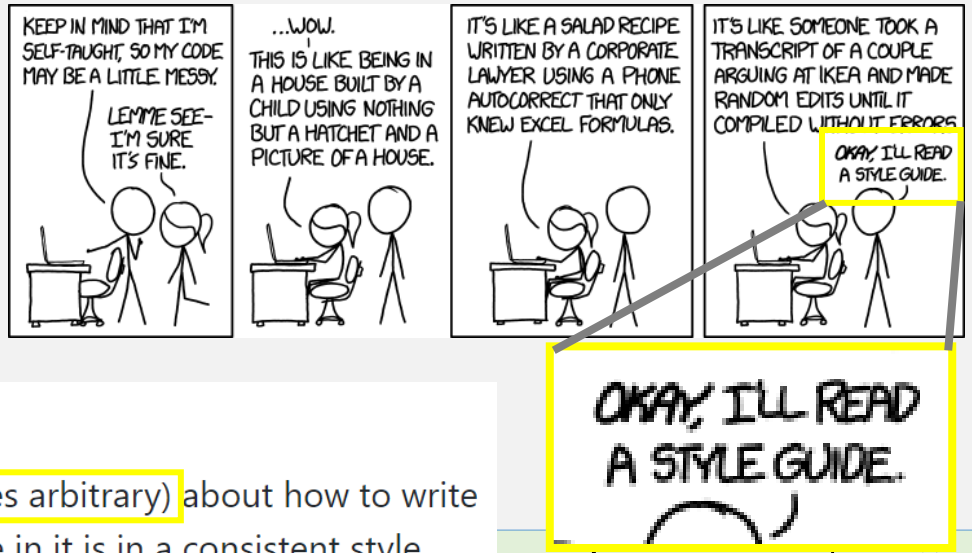
- … code is easier to read if
  everyone follows the same set of rules!


Every org / company has its own rules for how to write clean, readable programs

This is our rulebook!
(Design Recipe)

# Style

- Critical for writing readable code, e.g.




## Google Style Guides

Every major open-source project has its own style guide: a set of conventions (sometimes arbitrary) about how to write code for that project. It is much easier to understand a large codebase when all the code in it is in a consistent style.

"Style" covers a ... bles" to "never use exceptions." This ... If you are modifying a project that or... apply to that project.

- AngularJS S...
- Common Li...
- C++ Style Guide
- C# Style Guide
- Go Style Guide
- HTML/CSS Style Guide
- JavaScript Style Guide
- Java Style Guide
- Objective-C Style Guide
- Python Style Guide

## Airbnb JavaScript Style Guide() {

*A mostly reasonable approach to JavaScr...*

...org / company has its ~~rules~~ **style guide** for ...ow to write clean, ...eadable programs

Microsoft | Learn Documentation Training Certifications Q&A Code Sampl...

.NET Languages Features Workloads APIs Resources

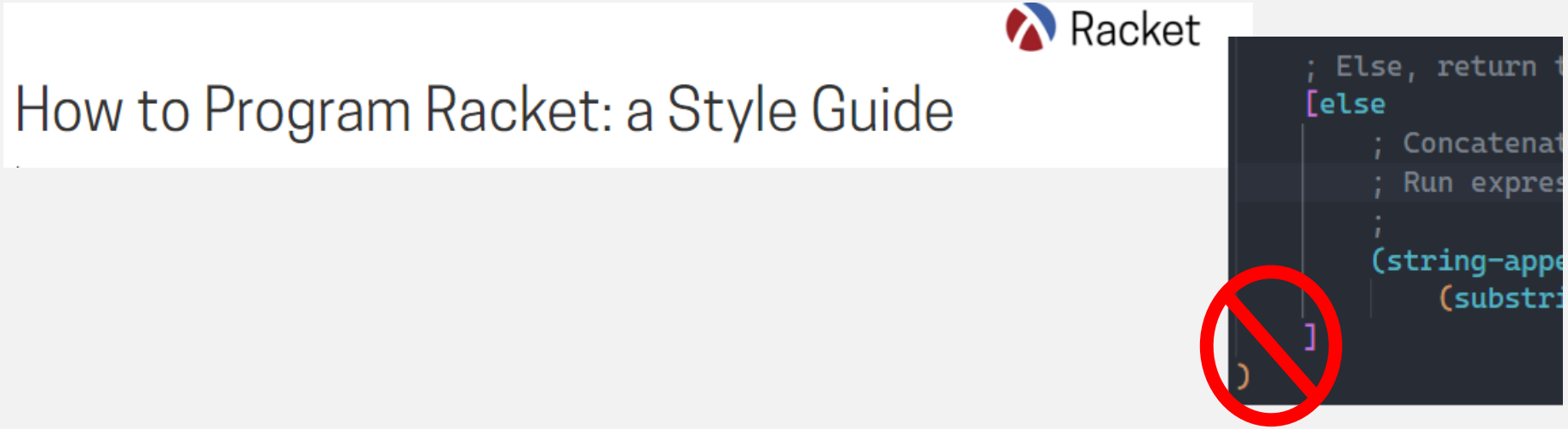
Learn / .NET / C# guide / Fundamentals /

## Common C# code conventions

A code standard is essential for maintaining code readability, consistency, and collaboration within a development team. Following industry practices and established

# Style: This Class

https://docs.racket-lang.org/style/index.html



# Style: Git commits



- Git commits must also be **readable** (concise and informative)

## How to Write a Git Commit Message

Commit messages matter. Here's how to write them well.

### The seven rules of a great Git commit message

1. Separate subject from body with a blank line
2. Limit the subject line to 50 characters
3. Capitalize the subject line
4. Do not end the subject line with a period
5. Use the imperative mood in the subject line
6. Wrap the body at 72 characters
7. Use the body to explain *what* and *why* vs. *how*

**A properly formed Git commit subject line s**
**complete the following sentence**:

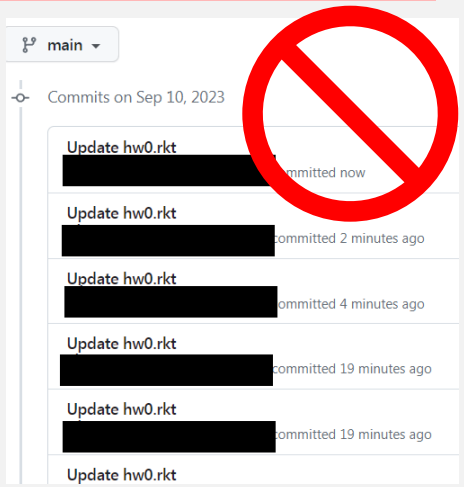
- If applied, this commit will *your subject line here*

For example:

- If applied, this commit will *refactor subsystem X for readability*
- If applied, this commit will *update getting started documentation*
- If applied, this commit will *remove deprecated methods*
- If applied, this commit will *release version 1.0.0*
- If applied, this commit will *merge pull request #123 from user/branch*

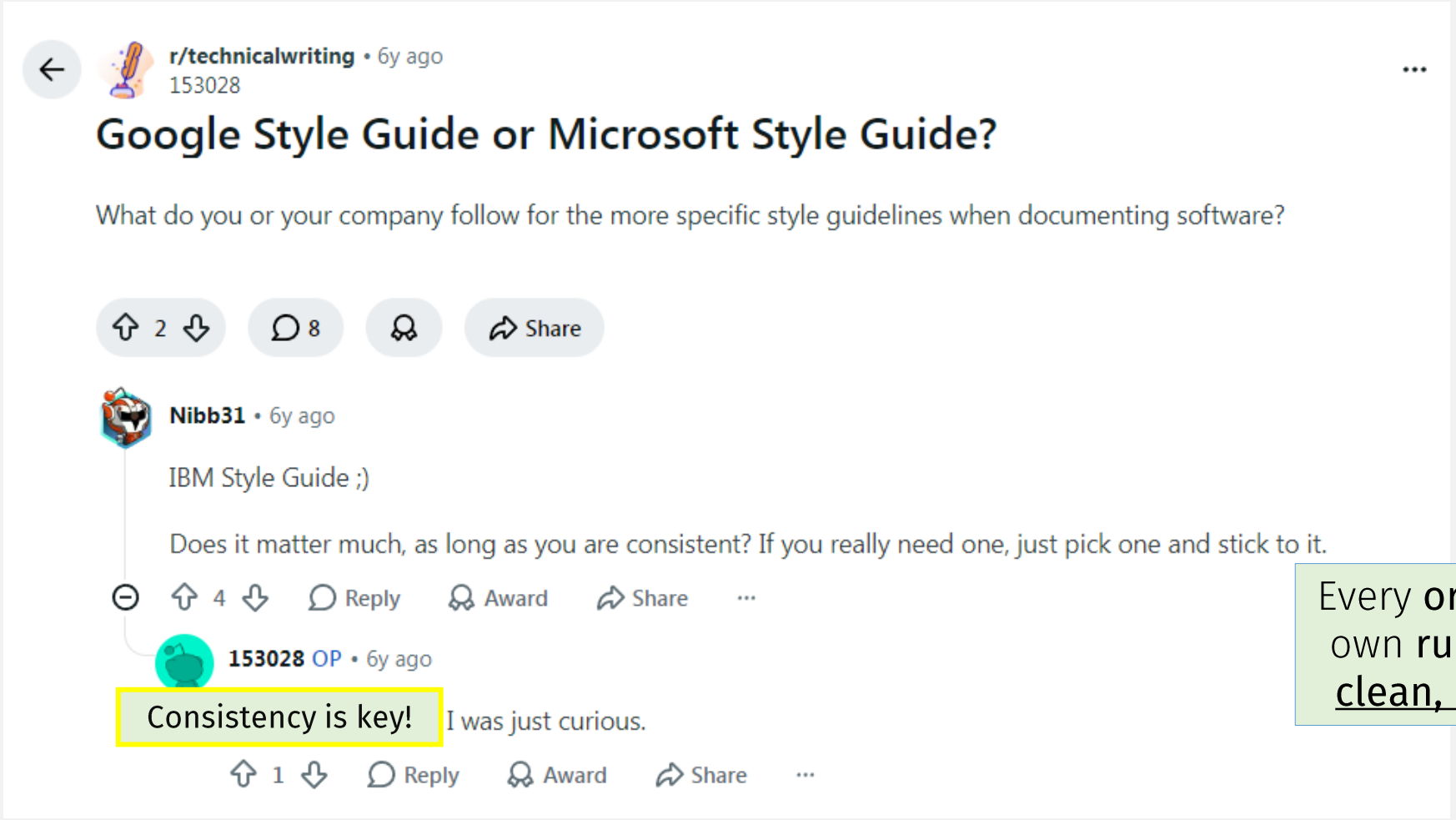
Notice how this doesn't work for the other non-imperative forms:

- If applied, this commit will ~~*fixed bug with Y*~~
- If applied, this commit will ~~*changing behavior of X*~~
- If applied, this commit will ~~*more fixes for broken stuff*~~
- If applied, this commit will ~~*sweet new API methods*~~



*Takeaway*

# Which Style is Best? Doesn't matter!

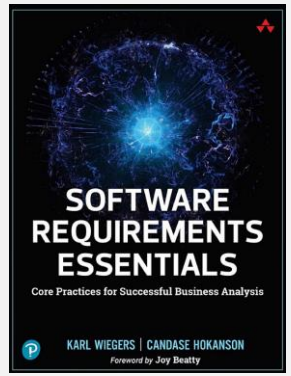



Every **org / company** has its own **rules for how to write** <u>clean, readable programs</u>

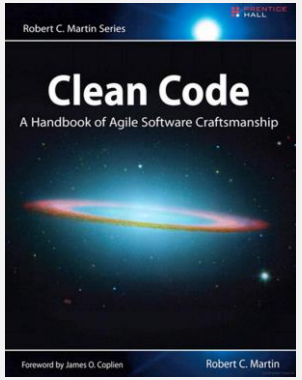
# *Interlude:* Software Dev 101

(framework with which to view the course)

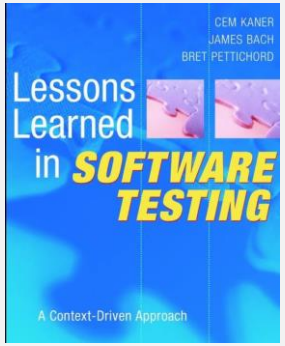
1. Write Specifications / Requirements
   - Figure out what the **program should do**




2. Implement code
   - Make the **program**




3. Verify correctness (i.e., **testing**):
   - Check the **program does what it should do?**




(This is **not** a **course about** the details of **Software Dev.**
But the **same principles** underpins both)

# *Interlude:* Software Dev 101

1. Write Specifications / Requirements
   - Figure out what the **program should do**
2. Implement code
   - Make the **program**
3. Verify correctness (i.e., **testing**):
   - Check the **program does what it should do?**

(This is not a **course about** the details of **Software Dev.**
But the **same principles** underpins both)

| Effort Allocation | |
|---|---|
| Software Engineer | Most Students |
| ~20% | |
| ~40% | ~~~110%~~ ~~~100%~~ ~150% |
| ~40% | |

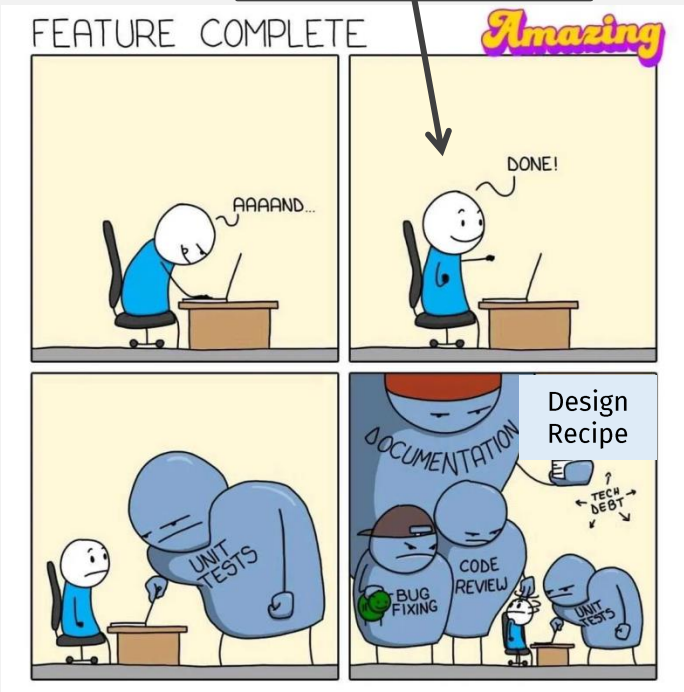
"Assignment is too long!"

"Assignment is too hard!"

"The course sucks!"

(won't work in ~~this course~~ real life)

**Most students submit here**




Design Recipe

# HW Advice

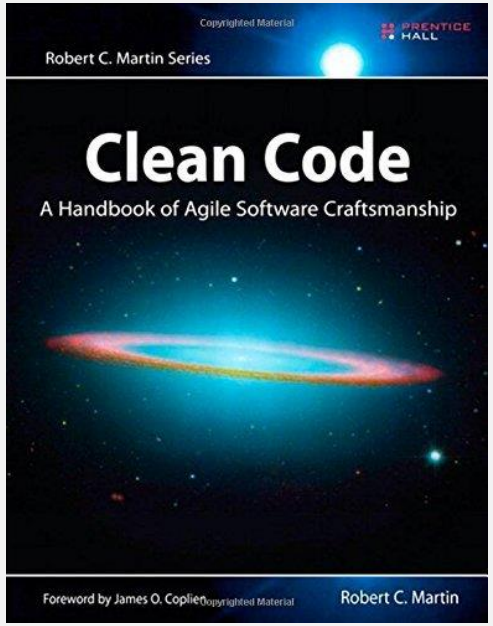
"Perhaps you thought that "**getting it working**" was the first order of business for a professional developer.

**THIS COURSE**

I hope by now, however, that ~~this book~~ has disabused you of that idea.

The functionality that you create today has a good chance of changing in the next release, but the **readability of your code** will have a profound effect on all the changes that will ever be made."

**— Robert C. Martin,**
Clean Code: A Handbook of Agile Software Craftsmanship




(won't work in this course, or real life)

**Most students submit here**

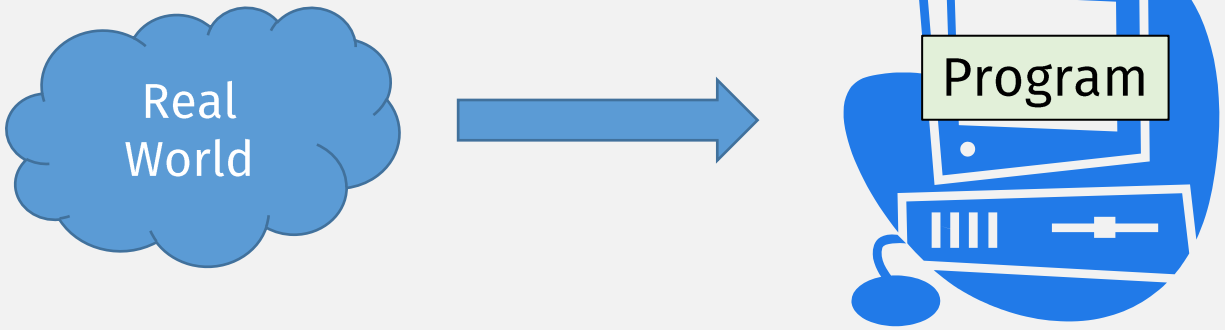

**Design Recipe**

# Program vs Real World

Real World "things" ...

e.g., Temperature

... need a **data representation** in the program




A **Data Definition** name

Specify possible values of the data

```
;; A TempC is an Integer
;; Represents: a temperature in degrees Celsius
```

**Interpretation** ... connects data to a real world concept

```
;; A TempF is an Integer
;; Represents: a temperature in degrees Fahrenheit
```

```
;; A TempK is an non-negative Integer
;; Represents: a temperature in degrees Kelvin
```

When programming,
**choosing data representations must be the first task!**
(way before writing any **code** ...
which **processes the data**)

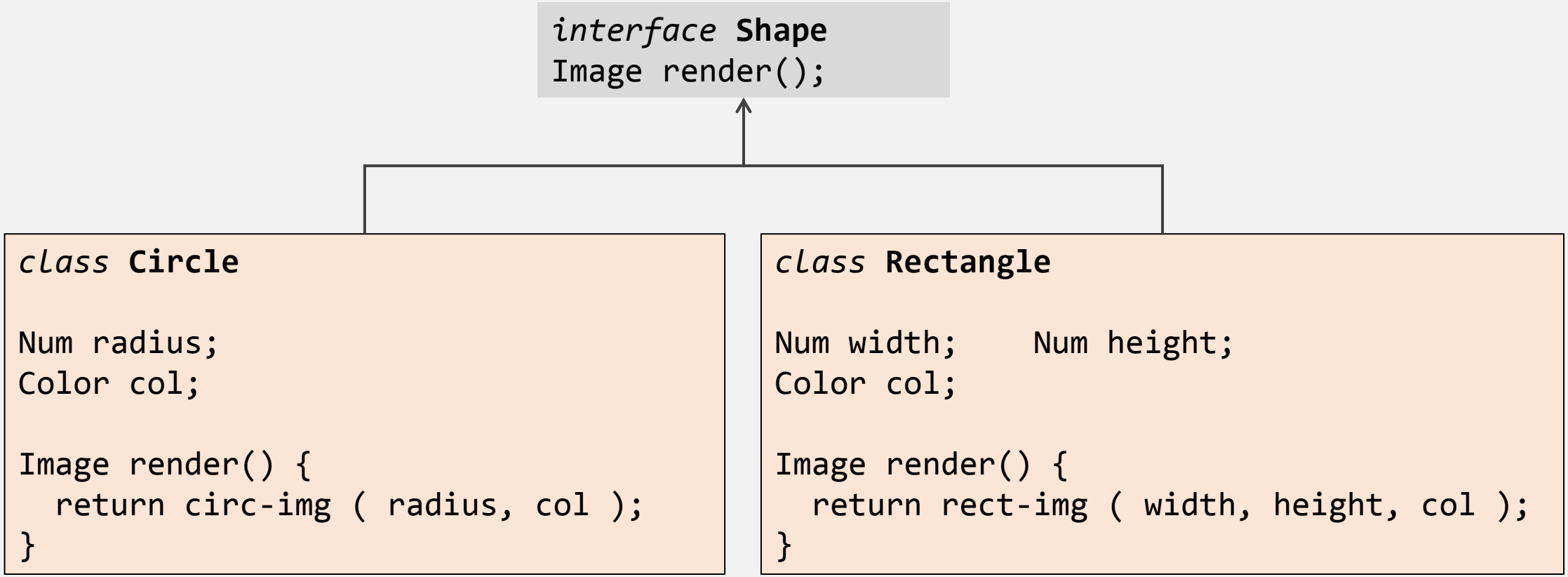
# Kinds of Data Definitions

- Basic data
  - E.g., **numbers, strings,** etc
- Intervals
  - Data that is from a **range of values,** e.g., [0, 100)
- Enumerations
  - Data that is **one of a list of possible values**, e.g., “green”, “red”, “yellow”
- Itemizations
  - Data value that can be from a **list of possible other data definitions**
  - E.g., **either** a **string** or **number** (Generalizes enumerations)
- Compound Data
  - Data that is a **combination of values from other data definitions**

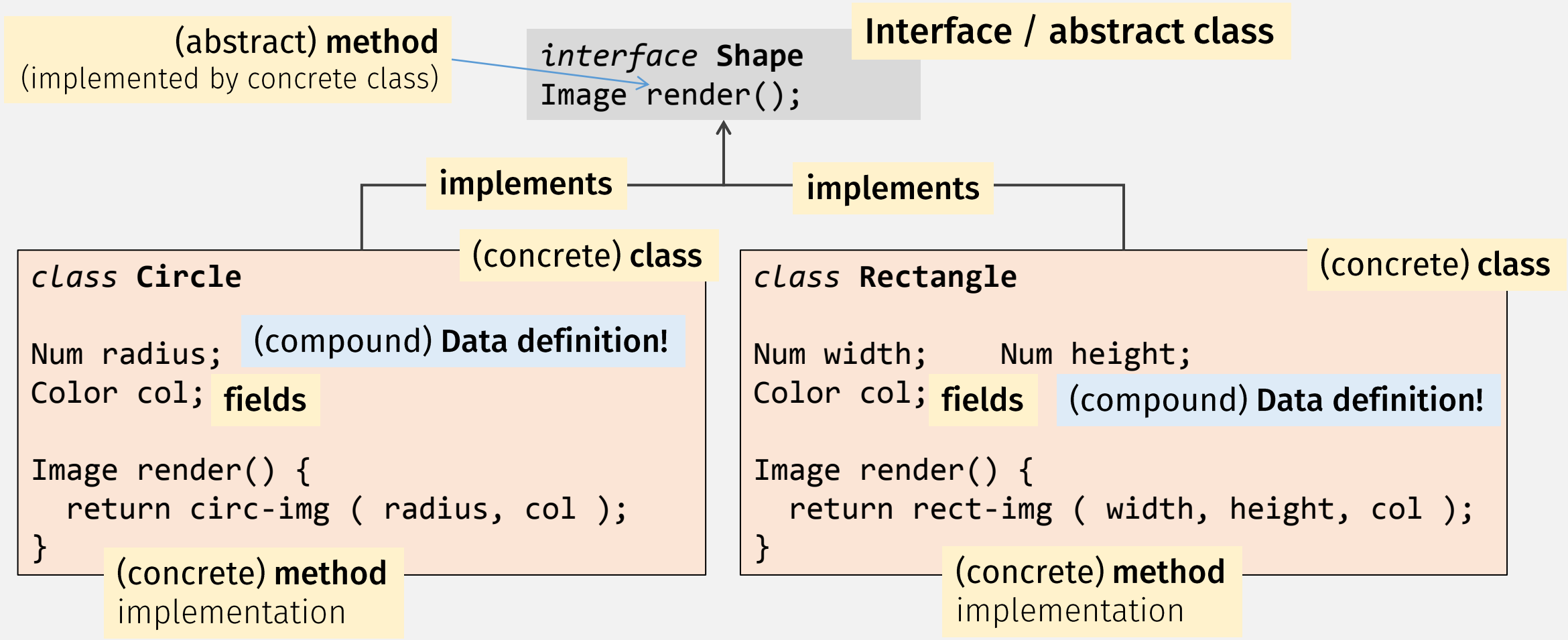
Combo of …

(extremely common, some PLs allow only this!)

# A Simple OO Example: Shapes



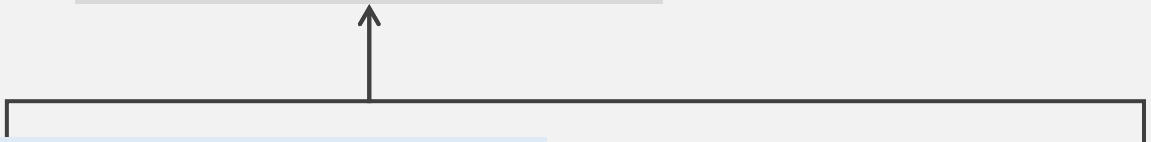
# A Simple OO Example: Terminology



# A Simple OO Example: Compare to CS450

(itemization) **Data definition**

```
interface Shape
Image render();
```



(itemization) **Data definition** item

```
class Circle

Num radius;
Color col;

Image render() {
  return circ-img ( radius, col );
}
```

(compound) **Data definition**

**function** implementation (one **cond** clause) for **Shape** data (split up)

(itemization) **Data definition** item

```
class Rectangle

Num width;    Num height;
Color col;

Image render() {
  return rect-img ( width, height, col );
}
```

(compound) **Data definition**

**function** implementation (one **cond** clause) for **Shape** data (split up)

# A Simple OO Example: Compare to CS450

```
;; A Shape is one of:
;; - Rectangle
;; - Circle
```

```
interface Shape
Image render();
```

```
class Circle

Num radius;
Color col;

Image render() {
  return circ-img ( radius, col );
}
```

```
(struct Circ [r col])
```

```
class Rectangle

Num width;    Num height;
Color col;

Image render() {
  return rect-img ( width, height, col );
}
```

```
(struct Rect [w h col])
```

```
;; render: Shape -> Image
(define (render sh)
  (cond
    [(Rect? sh) (rect-img sh)]
    [(Circ? sh) (circ-img sh)]))
```

"abstract" implementation

"cond" template is method "**dispatch**" – same as OO !

```
;; (implicit in OO) render: Shape -> Image
Image render (Shape sh)
  if (sh instanceof Rectangle) { rect-img(sh); }
  else if (sh instanceof Circle) { circ-img(sh); }
```

*Summary*

# There's Nothing Special About OOP!

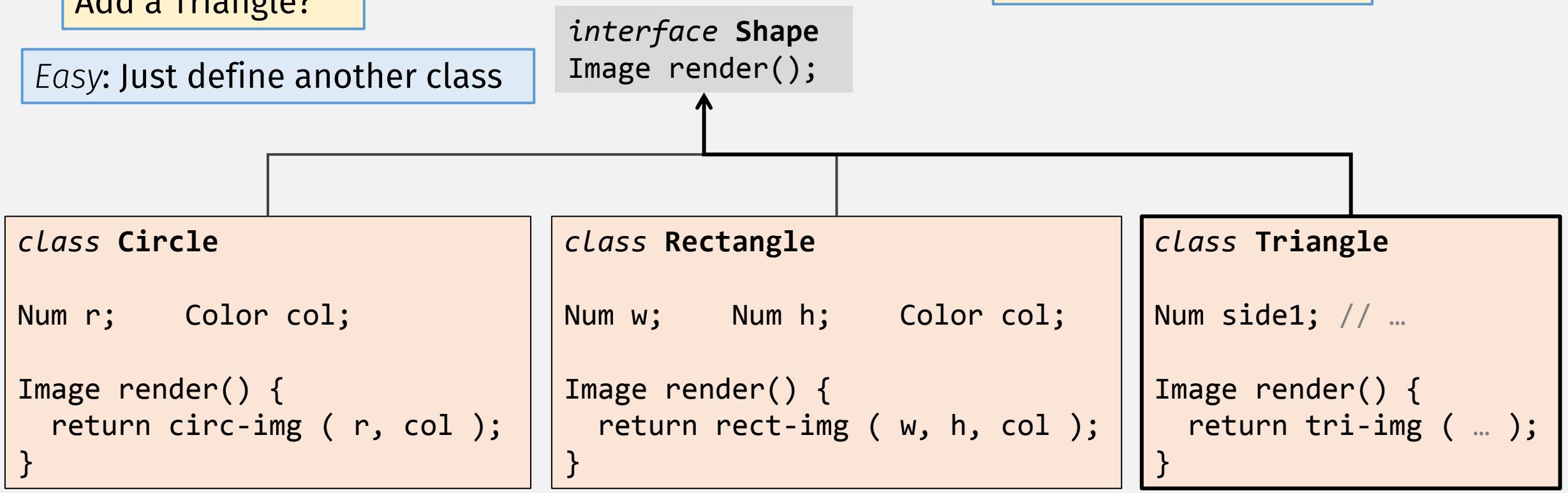
- A typical (`interface` and `class`es) **OOP program** is just a **specific data definition design choice!**
  - imposed by the language!
  - **itemization** of **compound data ...**
  - **...** where **functions of that data** are **grouped with other fields!**

# A Simple OO Example: Extensions?

Add a Triangle?

*Easy*: Just define another class

Add a rotate method?



# A Simple OO Example: Extensions?

Add **rotate** method?

*Hard!*: must update interface and every existing class (might not have access!)

```
interface Shape
Image render();
Image rotate();
```

```
class Circle

Num r;    Color col;

Image render() {
  return circ-img ( r, col );
}

Circle rotate() { … }
```

```
class Rectangle

Num w;    Num h;    Color col;

Image render() {
  return rect-img ( w, h, col );
}

Rectangle rotate() { … }
```

```
class Triangle

Num side1; // …

Image render() {
  return tri-img ( … );
}

Triangle rotate() { … }
```

# Shapes, CS450 style

Add a Triangle?

*Hard*!: must:

```
;; A Shape is one of:
;; - Rectangle
;; - Circle
;; Represents: a shape image
```

```
;; A Rectangle is a (mk-Rect Num Num Color)
;; fields are width, height, color
(struct Rect [w h col])
;; A Circle is a (mk-Circ Num Color)
;; fields are radius and color
(struct Circ [r col])
```

```
;; render: Shape -> Image
(define (render sh)
 (cond
  [(Rect? sh) (render-rect sh)]
  [(Circ? sh) (render-circ sh)]))
```

# Shapes, CS450 style

Add a Triangle?

*Hard*!: must:
- update data def,
- define new struct,

```
;; A Shape is one of:
;; - Rectangle
;; - Circle
;; - Triangle
;; interp: Represents a shape image
```

```
;; A Rectangle is a (mk-Rect Num Num Color)
;; fields are width, height, color
(struct Rect [w h col])
;; A Circle is a (mk-Circ Num Color)
;; fields are radius and color
(struct Circ [r col])
;; A Triangle is a (mk-Tri … )
;; fields are …
(struct Tri [ … ])
```

```
;; render: Shape -> Image
(define (render sh)
 (cond
  [(Rect? sh) (render-rect sh)]
  [(Circ? sh) (render-circ sh)]))
```

# Shapes, CS450 style

Add a Triangle?

*Hard*!: must:
- update data def,
- define new struct,
- update every existing "dispatch" function (might not have access!)

```
;; render: Shape -> Image
(define (render sh)
 (cond
  [(Rect? sh) (render-rect sh)]
  [(Circ? sh) (render-circ sh)]
  [(Tri? sh)  (render-tri sh) ]))
```

```
;; A Shape is one of:
;; - Rectangle
;; - Circle
;; - Triangle
;; interp: Represents a shape image
```

```
;; A Rectangle is a (mk-Rect Num Num Color)
;; fields are width, height, color
(struct Rect [w h col])
;; A Circle is a (mk-Circ Num Color)
;; fields are radius and color
(struct Circ [r col])
;; A Triangle is a (mk-Tri … )
;; fields are …
(struct Tri [ … ])
```

# Shapes, CS450 style

Add a Triangle?

*Hard!*: must:
- update data def,
- define new struct,
- update every existing "dispatch" function (might not have access!)

Add a **`rotate`** function?

*Easy!*: Just define another function!

```
;; A Shape is one of:
;; - Rectangle
;; - Circle
;; Represents: a shape image
```

```
;; A Rectangle is a (mk-Rect Num Num Color)
;; fields are width, height, color
(struct Rect [w h col])
;; A Circle is a (mk-Circ Num Color)
;; fields are radius and color
(struct Circ [r col])
```

```
;; render: Shape -> Image
(define (render sh)
 (cond
  [(Rect? sh) (render-rect sh)]
  [(Circ? sh) (render-circ sh)]))
```

```
;; rotate: Shape -> Shape
(define (rotate sh)
 (cond
  [(Rect? sh) (rotate-rect sh)]
  [(Circ? sh) (rotate-circ sh)]))
```

# Plain functions vs OO Comparison

Add a **new "item"** to **itemization data def,** e.g., **`Triangle`**

- **OO**: *Easy*
  - Just **define another `class`**
    - **Implicit "Dispatch" function**(s) automatically updated

- **Functions**: *Hard*
  - Must **update data def** and **define another `struct`**
    - Every **"dispatch" function**(s) must be manually updated with another cond line

Add a **new operation** for **itemization data def,** e.g., **`rotate`**

- **OO**: *Hard*
  - Must **update interface,** and **add new method** to every class that implements it

- **Functions**: *Easy*
  - Just **define another function**

# Mixins - A better way? (`class` "arithmetic")

- A **`Mixin`** is a **function**, whose **input** and **output** is a **`class`**!

- Available in many languages:
  - Racket
  - JavaScript
  - Scala

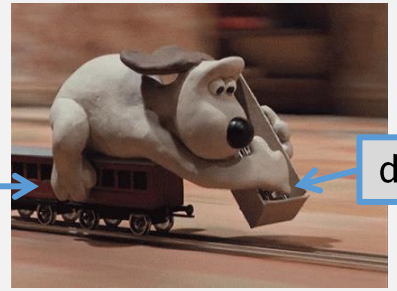
- `(add-rotate-mixin class-without-rotate)`
  `=> class-with-rotate`

# Design Recipe(s)

- Data Design
- Function Design

Based on **data design!**

When programming,
**choosing data representations must be the first task!**
(way before writing any **code** …
which **processes the data**)




(programmers who skip data design step)

# Function Design Recipe

1. Name
2. Signature
   - # of arguments and their data type
   - Output type
   - May only reference “defined” Data Definition names
3. Description – explains how fn works, in English
4. Examples – shows how fn works, in code
5. Code
6. Tests

# Function Design Recipe … is Software Dev!

1. **Name**
2. **Signature**
   - **# of arguments** and **their data type**
   - **Output type**
   - May only **reference "defined" Data Definition names**
3. **Description** – explains how fn works, in English
4. **Examples** – shows how fn works, in code
5. **Code**
6. **Tests**

Specify

Implement

Verify

# Design Recipe, then repeat

1. **Data Design**
2. **Function Design**

Programming is an **iterative** process!

Each iteration should be **incremental**!

# Iterative Programming

Other functions (“wish list”)

1. Name
2. Signature
   - # of arguments and their data type
   - Output type
   - May only reference “defined” Data Definition names
3. Description
4. Examples
5. Code
6. Tests

Programming is an **iterative** process!

**Danger, Danger**

This is not a license to “hack”

i.e., arbitrarily changing code and praying “this time it will just work”

Instead, **program incrementally**

The *faster* way to program

# The Incremental Programming Pledge

"slow down to speed up"

***At all times***, all of the following should be **`true`** of your code:

1. Comments (**data defs, signatures**, etc) **match code**
2. Code has no syntax errors
    1. E.g., **missing / extra parens**
3. Runs **without runtime errors / exceptions**
    1. E.g., **undefined variable use, `div by zero`, call a "non function"**
4. All tests pass

When you **make a code edit** that **renders one of the above `false`, STOP …**

… and **don't do anything else** until **all the statements are true again.**

(this way, it's easy to revert back to a "working" program)

# Incremental Programming: Real-World Example


- https://www.youtube.com/watch?v=1SlGgCxJa3w
- "when you do everything at once …
  you're not sure why it's not working!"
- "when you layer it, when you break it down …
  and you hit a spot when it's not working …
  then you can just focus on that spot!"

**3.** Make small changes only (something easy to revert)

5. Code
6. Tests

**4.** Test each (small) change (before making another one)

# *Conventional Wisdom:* Write Small Functions

## Write Short Functions

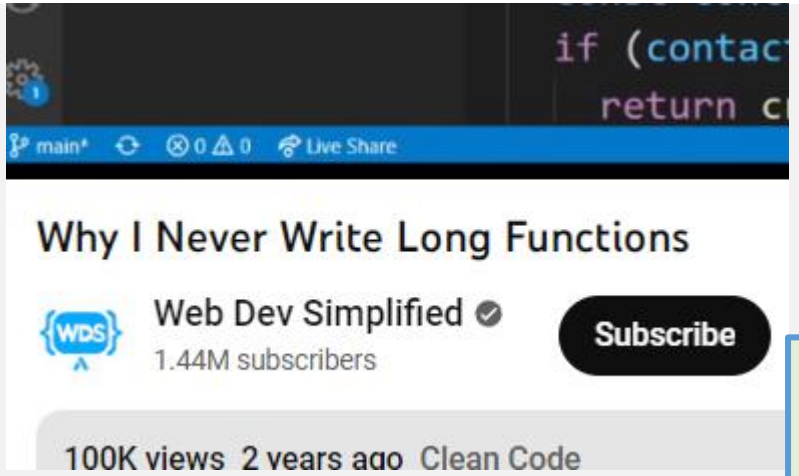
**Google C++ Style Guide**

Prefer small and focused functions.

We recognize that long functions are sometimes appropriate, so no hard limit is placed on functions length. If a function exceeds about 40 lines, think about whether it can be broken up without harming the structure of the program.

Even if your long function works perfectly now, someone modifying it in a few months may add new behavior. This could result in bugs that are hard to find. Keeping your functions short and simple makes it easier for other people to read and modify your code. Small functions are also easier to test.

You could find long and complicated functions when working with some code. Do not be intimidated by modifying existing code: if working with such a function proves to be difficult, you find that errors are hard to debug, or you want to use a piece of it in several different contexts, consider breaking up the function into smaller and more manageable pieces.



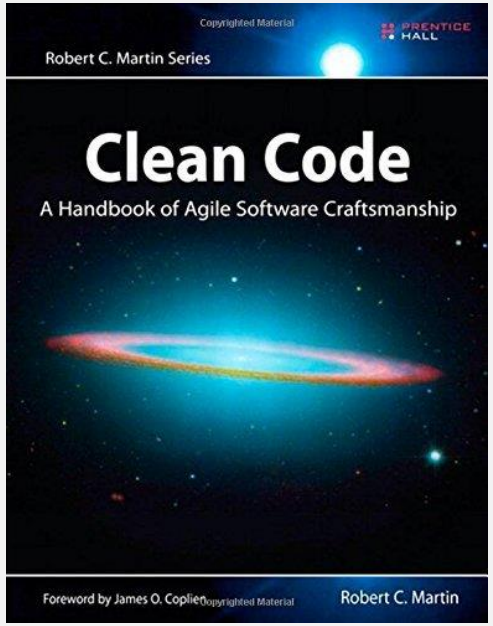

**Small Functions Considered Awesome**

Josh Saint Jacque · Follow
11 min read · Aug 22, 2017

1 function,
1 task, … processes
1 data definition!

Good rule of thumb:
A **function** should do **one, easily explainable task**

# ~~HW~~ Advice




> “The first rule of functions is that they should be small.
>
> The second rule of functions is that they should be smaller than that.”
>
> — **Robert C. Martin,**
> Clean Code: A Handbook of Agile Software Craftsmanship

In this class:

**1 function** does
**1 task** which processes
**1 kind of data**

# “Clean (Readable) Code” is Wrong???

Things Of Interest » Blog »

## It's probably time to stop recommending Clean Code

2020-06-28 by qntm

PROGRAMMING COURSES

## "Clean" Code, Horrible Performance

Many programming "best practices" taught today are performance disasters wai
happen.

CASEY MURATORI

FEB 28, 2023

## What Actually Matters for Code Quality (Clean Code is Wrong)

Victor Moreno Follow 14 min read · Jul 16, 2023

*Don't Worry!*

# "Clean (Readable) Code" is Still a Good Goal

r/AskProgramming • 1y ago
Yelebear

## Why do some people hate "Clean Code"

Other

It just means making readable and consistent coding practices, right?

- **"When you see duplication in the code, it represents a missed opportunity for abstraction."** — Robert C. Martin, *Clean Code*.

Example

**x5reyals** • 1y ago • Edited 1y ago

Because other people use it as dogma. Like any other resource it's a collection of tools that should be used when appropriate. Sometimes overly clean code runs the risk of losing context. All of a sudden the parameter

**tommygeek** • 5y ago

As in all things in software, I think the advice in the book is largely situational. Did reading Uncle Bob make me a better programmer? I believe so. Do I dogmatically adhere to all these things in code reviews? No. Because "clean" is subjective and at the intersection of consistency and time spent in a particular codebase.

**zerpa** • 5y ago

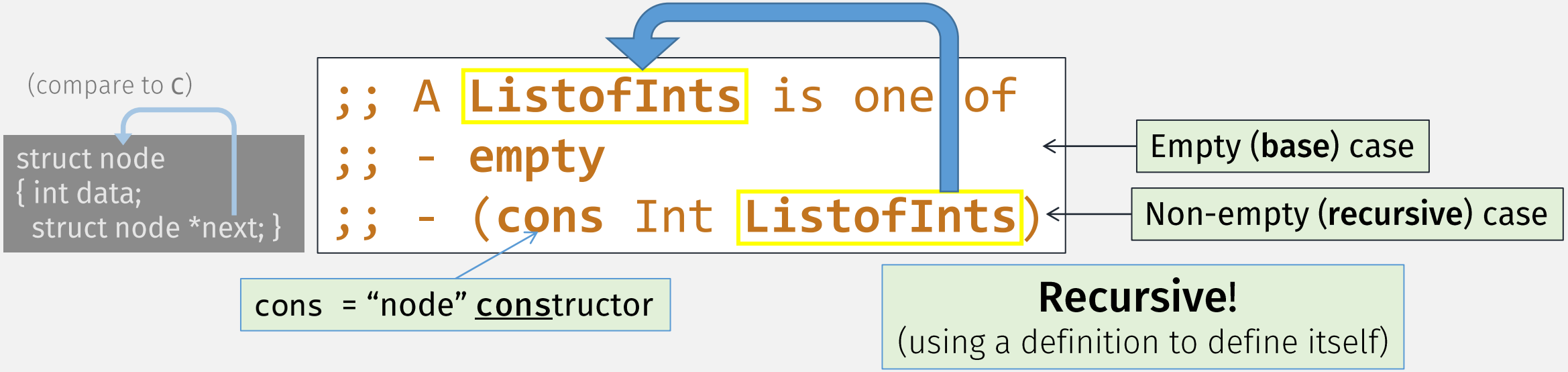
Software developer maturity levels:

1. No sense of code cleanliness
2. Adheres religiously to code style advice in a book in all situations
3. Can appreciate the book's advice and apply it correctly when applicable
4. Writes and refactors code to make sound advice applicable in more situations And level 0: willfully ignores advice and invents own uncommon coding style and practices

If you are here?

Next task: get to here?

# List Data Definition Example




(how can we **use a list of ints to define a list of ints**?!?)

**Recursion** is (a **valid math concept**), but only if there is:

- A **base** case
- A **recursive case** (that is "smaller")

# List Data Definition Template

```
;; A ListofInts is one of
;; - empty
;; - (cons Int ListofInts)
```

```
TEMPLATE??
```

(what kind of data definition is this?)

# List Data Definition Template

```
;; A ListofInts is one of
;; - empty
;; - (cons Int ListofInts)
```

Empty (**base**) case

Non-empty (**recursive**) case

The **shape** of **the function** matches The **shape** of **the data definition**!

This is an **itemization**, so **template** has **cond**

TEMPLATE??

```
;; TEMPLATE for list-fn
;; list-fn : ListofInts -> ???
(define (list-fn lst)
  (cond
    [(empty? lst) ....]
    [(cons? lst) .... (first lst) ....
                 .... (rest lst) ....]))
```

Empty (**base**) case

Non-empty (**recursive**) case

# List Data Definition Template

```
;; A ListofInts is one of
;; - empty
;; - (cons Int ListofInts)
```

"first" "rest"

This is both **itemization**, so **template** has **cond**

and

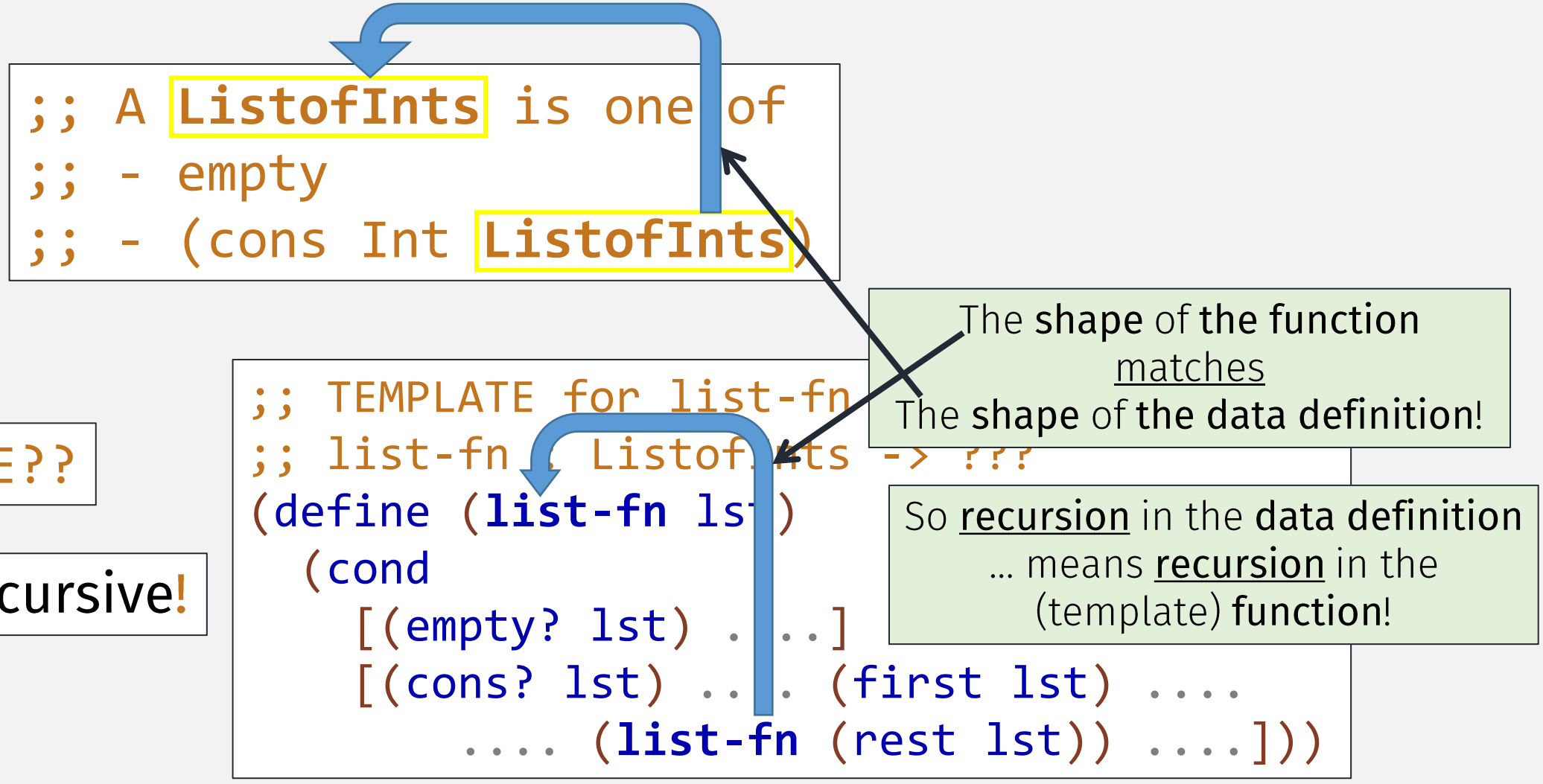
**compound data,** so **template** has "getters"

```
;; TEMPLATE for list-fn
;; list-fn : ListofInts -> ???
(define (list-fn lst)
  (cond
    [(empty? lst) ....]
    [(cons? lst) ....(first lst) ....
                 .... (rest lst) ....]))
```

The **shape** of **the function** matches The **shape** of **the data definition**!

Wait, where is the **recursion**???

# List Data Definition Template is Recursive!



# List Function Examples

Differences?

```
;; inc-lst: ListofInt -> ListofInt
;; Returns list with each element incremented
(define (inc-lst lst)
  (cond
    [(empty? lst) empty]
    [else (cons (add1 (first lst))
                (inc-lst (rest lst)))]))
```

```
;; next-world : ListofBall -> ListofBall
;; Updates position of each ball by one tick
(define (next-world lst)
  (cond
    [(empty? lst) empty]
    [else (cons (next-ball (first lst))
                (next-world (rest lst)))]))
```

# Abstraction Recipe

1. Find similar patterns in a program
   - Minimum: 2
   - Ideally: 3+
2. Identify differences and make them parameters
3. Create a reusable abstraction with the discovered parameters
   - E.g., a function(al) abstraction

# Abstraction: Common List Function #1

Make **difference**(s) a **parameter** of a (**function**) **abstraction**

```
(define (lst-fn1 fn lst)
  (cond
    [(empty? lst) empty]
    [else (cons (fn (first lst))
                (lst-fn1 fn (rest lst)))]))
```

# Abstraction: Common List Function #1

```
;; lst-fn1: (?? -> ??) Listof?? -> Listof??
;; Applies the given fn to each element of given lst
```

```
(define (lst-fn1 fn lst)
  (cond
    [(empty? lst) empty]
    [else (cons (fn (first lst))
                (lst-fn1 fn (rest lst)))]))
```

# Abstraction of Data Definitions

```
;; A ListofInt is one of
;; - empty
;; - (cons Int ListofInt)
```

```
;; A ListofBall is one of
;; - empty
;; - (cons Ball ListofBall)
```

# Abstraction Recipe

1. Find similar patterns in a program
   - Minimum: 2
   - Ideally: 3+
2. Identify differences and make them parameters
3. Create a reusable abstraction with the discovered parameters
   - E.g., a function(al) abstraction

# Abstraction of Data Definitions

```
;; A ListofInt is one of
;; - empty
;; - (cons Int ListofInt)
```

```
;; A ListofBall is one of
;; - empty
;; - (cons Ball ListofBall)
```

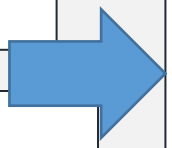
# Abstraction Recipe

1. Find similar patterns in a program
   - Minimum: 2
   - Ideally: 3+
2. Identify differences and make them parameters
3. Create a reusable abstraction with the discovered parameters
   - E.g., a function(al) abstraction
   - E.g., a data abstraction

# Abstraction of Data Definitions

```
;; A ListofInt is one of
;; - empty
;; - (cons Int ListofInt)
```

```
;; A ListofBall is one of
;; - empty
;; - (cons Ball ListofBall)
```



parameter

```
;; A Listof<X> is one of
;; - empty
;; - (cons X Listof<X>)
```

# Abstraction Recipe

1. Find similar patterns in a program
   - Minimum: 2
   - Ideally: 3+
2. Identify differences and make them parameters
3. Create a reusable abstraction with the discovered parameters
   - E.g., a function(al) abstraction
   - E.g., a data abstraction
4. Use the abstraction, by giving concrete arguments for parameters

# Abstraction: Common List Function #1

```
;; lst-fn1: (X -> Y) Listof<X> -> Listof<Y>
;; Applies the given fn to each element of given lst
```

```
(define (lst-fn1 fn lst)
  (cond
    [(empty? lst) empty]
    [else (cons (fn (first lst))
                (lst-fn1 fn (rest lst)))]))
```

```
(define (inc-lst lst) (lst-fn1 add1 lst))
(define (next-world lst) (lst-fn1 next-ball lst))
```

# Common List Function #1: **map**

```
;; map: (X -> Y) Listof<X> -> Listof<Y>
;; Applies the given fn to each element of given lst
```

```
(define (map fn lst)
  (cond
    [(empty? lst) empty]
    [else (cons (fn (first lst))
                (map fn (rest lst)))]))
```

```
(define (inc-lst lst) (map add1 lst))
(define (next-world lst) (map next-ball lst))
```

# Abstraction Recipe

1. Find similar patterns in a program
   - Minimum: 2
   - Ideally: 3+
2. Identify differences and make them parameters
3. Create a reusable abstraction with the discovered parameters
   - E.g., a function(al) abstraction
   - E.g., a data abstraction
   - The abstraction must have a short, clear name and "be logical"
4. Use the abstraction by giving concrete "argument" parameters

Abstractions should do a "clear and concisely defined task"

*Don't Worry!*

# "Clean (Readable) Code" is Still a Good Goal

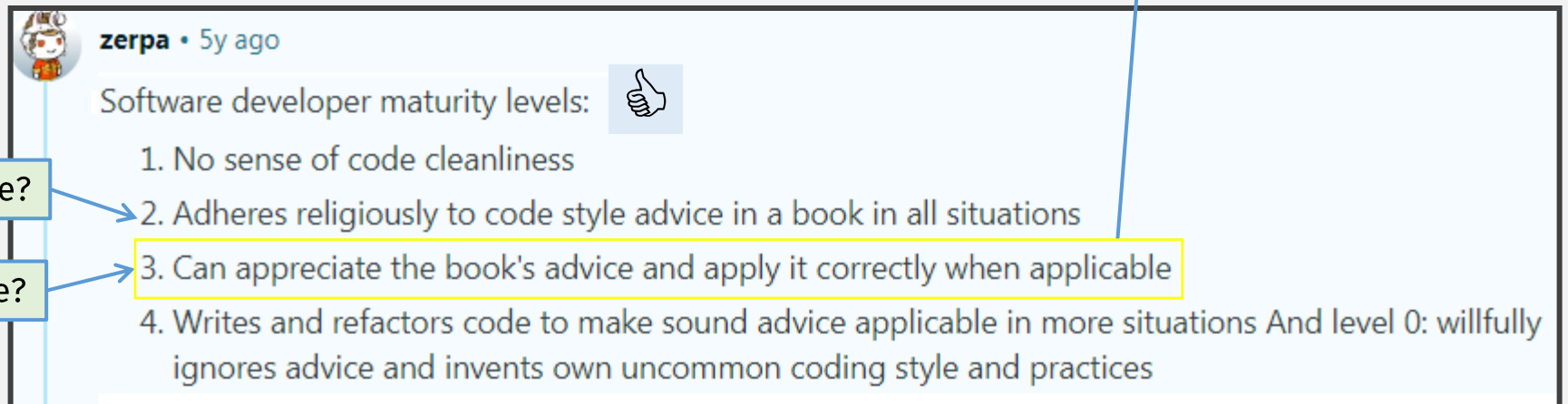
Not always!

- **"When you see duplication in the code, it represents a missed opportunity for abstraction."** — Robert C. Martin, *Clean Code*.

Example

If you are here?

Next task: get to here?



**zerpa** • 5y ago

Software developer maturity levels:

1. No sense of code cleanliness
2. Adheres religiously to code style advice in a book in all situations
3. Can appreciate the book's advice and apply it correctly when applicable
4. Writes and refactors code to make sound advice applicable in more situations And level 0: willfully ignores advice and invents own uncommon coding style and practices

# Abstraction Recipe


1. Find similar patterns in a program
   - Minimum: 2
   - Ideally: 3+

Not all “repeated code” should be abstracted

2. Identify differences and make them parameters
3. Create a reusable
   - E.g., a function(al) abstraction
   - E.g., a data abstraction
   - The abstraction must have a short, clear name and “be logical”
4. Use the abstraction by giving concrete “arguments” parameters

Creating **Bad Abstractions** is **Dangerous**

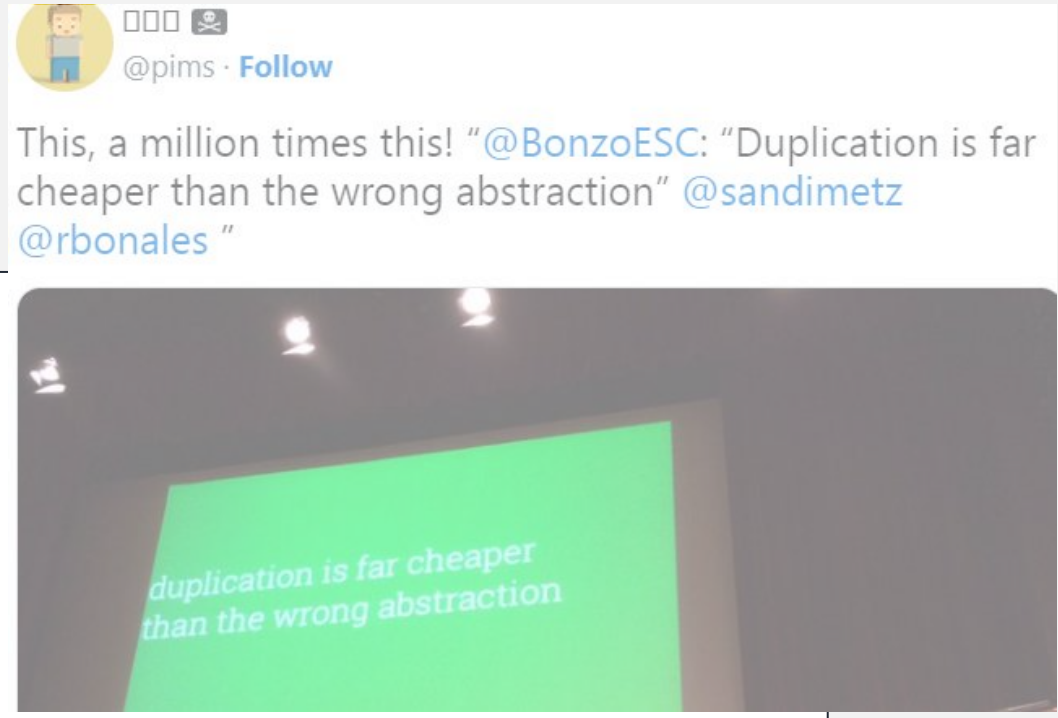
Creating **Good Abstractions** is **Hard**

# Abstraction Warning Story

I came to see the following pattern:

1. *Programmer A* sees duplication …
2. *Programmer A* extracts duplication and gives it a name.
   This *creates a new abstraction.*
3. *Programmer A* replaces duplication with the new abstraction.
   *Ah, the code is perfect. Programmer A trots happily away.*

4. **Time passes …**




https://sandimetz.com/blog/2016/1/20/the-wrong-abstraction

# Abstraction Warning Story

I came to see the following pattern:

1. Programmer A sees duplication.
2. Programmer A extracts duplication and gives it a name.
   *This creates a new abstraction.*
3. Programmer A replaces the duplication with the new abstraction.
   *Ah, the code is perfect. Programmer A trots happily away.*

4. Time passes …

5. A **new requirement appears …** for which the **current abstraction** is ***almost* perfect.**
6. ***Programmer B*** **gets tasked** to **implement** this **requirement …**
   ***Programmer B*** **tries** to **retain** the **existing abstraction …**
   … but it's ***not perfect*** **…** so *they* ***alter the code to take a* parameter,**
   … and *then* ***add extra logic*** *that is* ***conditionally*** ***based*** *on the* ***value of that parameter.***

Bad programmers love to overuse **`if`**, **`cond`**, etc ☹

`https://sandimetz.com/blog/2016/1/20/the-wrong-abstraction`

# Abstraction Warning Story

I came to see the following pattern:

1. Programmer A sees duplication.
2. Programmer A extracts duplication and gives it a name.
   *raction.*
3. Pr[illegible] duplication with the new abstraction.
   *Ah, the code is perfect. Programmer A trots happily away.*
4. Time passes …

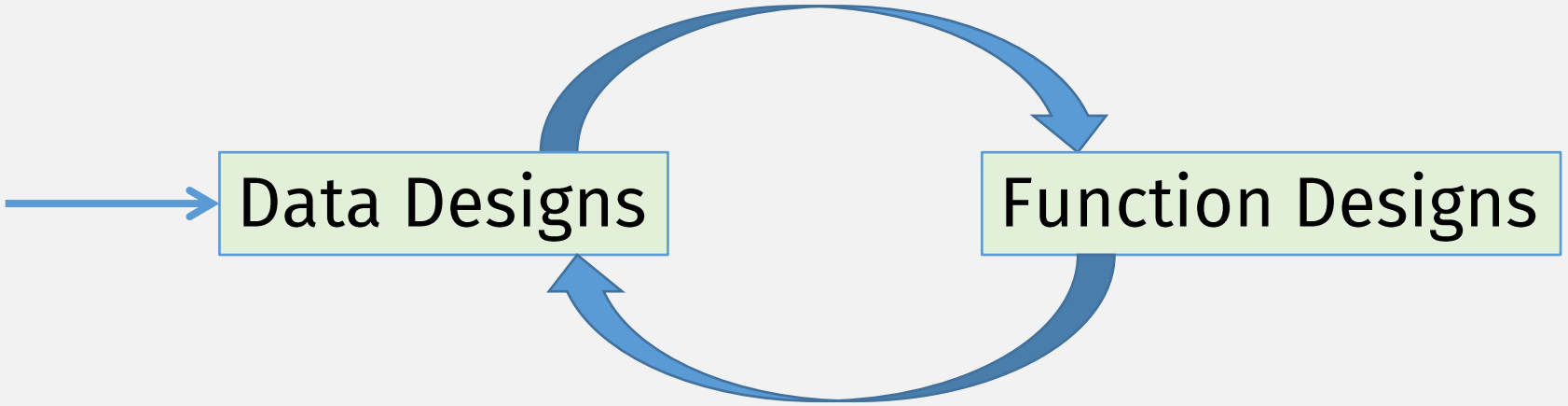
**How to avoid?**

Always be thinking about the data

5. A new requirement appears … for which the current abstraction is almost perfect.
6. *Programmer B* gets tasked to implement this requirement …
   *Programmer B* tries to retain the existing abstraction …
   … but it's *not perfect* … so *they alter the code to take a* **parameter**,
   … and *then add extra logic that is conditionally based on the value of that parameter.*
7. Another **new requirement arrives** … and a **new** *Programmer X*, *who adds an additional* **parameter**
   … and *a new conditional … Repeat until* ***code becomes incomprehensible.***
8. You appear in the story about here … and your life takes a dramatic turn for the worse.

https://sandimetz.com/blog/2016/1/20/the-wrong-abstraction

# Program Design Recipe



# Abstraction Warning Story

I came to see the following pattern:

1. Programmer A sees duplication.
2. Programmer A extracts duplication and gives it a name.
   *Ah, an abstraction.*
3. Programmer A replaces the duplication with the new abstraction.
   *Ah, the code is perfect. Programmer A trots happily away.*
4. Time passes …
5. A new requirement appears for which the current abstraction is *almost* perfect.
6. Programmer B gets tasked to implement this requirement.
   *Programmer B tries to take a parameter, and then add extra logic that conditionally changes based on the value of that parameter.*
7. Another new requirement arrives. And a new *Programmer X*, who adds an additional parameter and a new conditional. *Loop until* ***code becomes incomprehensible.***
8. You appear in the story about here, and your life takes a dramat

**How to avoid?**

Always be thinking about the data (definitions)

Don't just try to "get the code working"

These programmers only cared about "getting the code working"

*conditionally*

```
(if .... (if .... (if .... ....) ....) ....)
```

augh

Bad programmers love to overuse `if`, `cond`, etc ☹

https://sandimetz.com/blog/2016/1/20/the-wrong-abstraction

*Don't Worry!*

# "Clean (Readable) Code" is Still a Good Goal

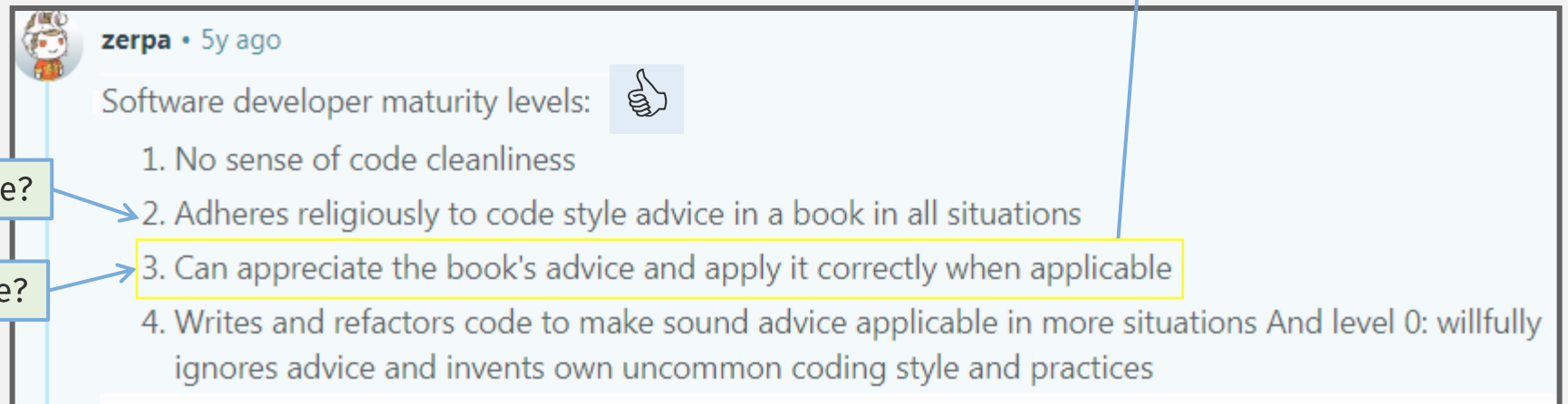
Not always!

- **"When you see duplication in the code, it represents a missed opportunity for abstraction."** — Robert C. Martin, *Clean Code*.

You've **learned** the **"correct" programming rules …**

… now it's up to you to **figure out when it's appropriate to break them**!

Example




If you are here?

Next task: **get to here?**

WTF?


Programming Languages are for human communication!

*Thank you for a great semester!*

create a picture for the last lecture in CS450