Program Design

Modularity and reusability

Complexity

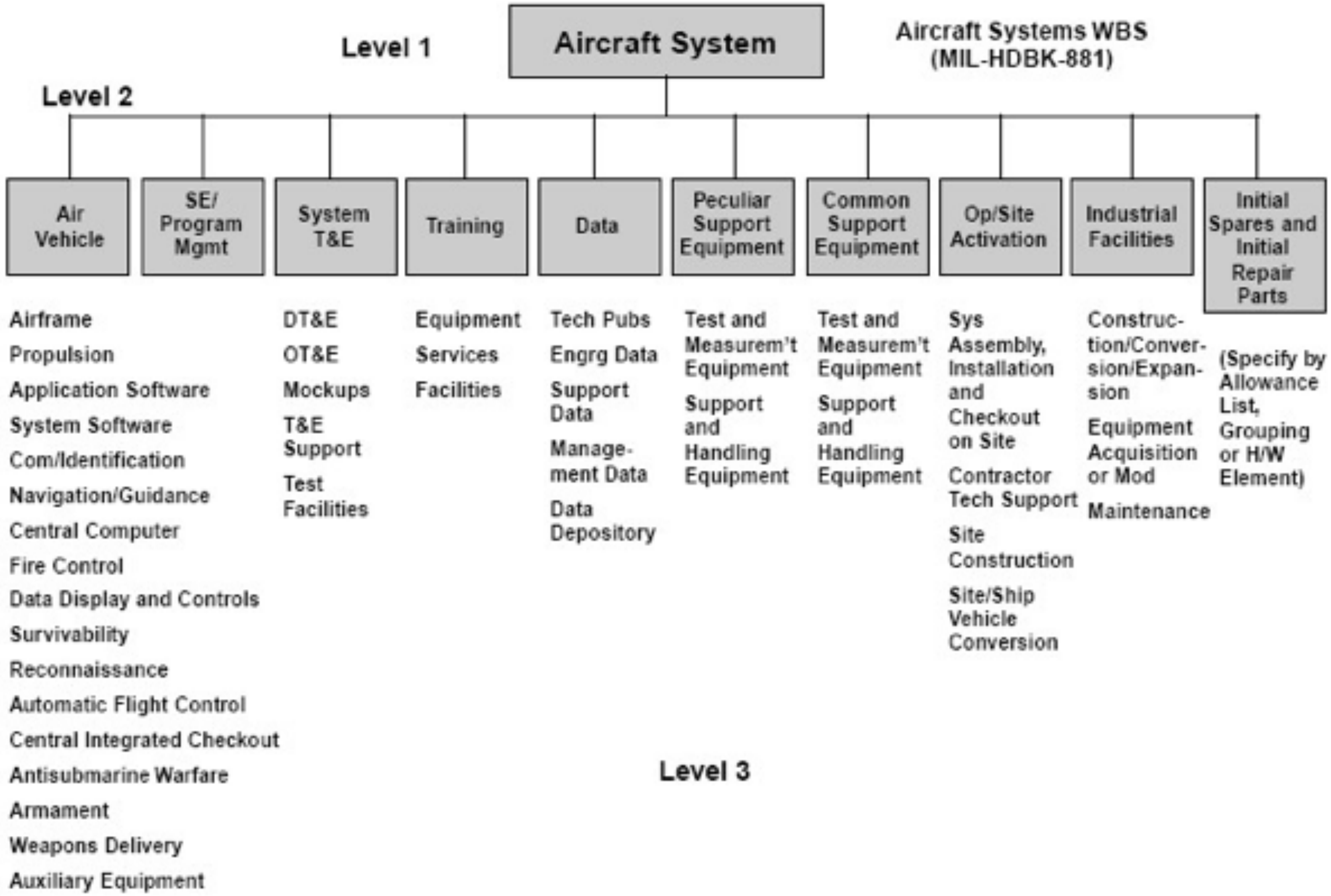
In Engineering and Programming

Project Sizes in University

- Course assignment

- Course project/FYP

Project Size at Work

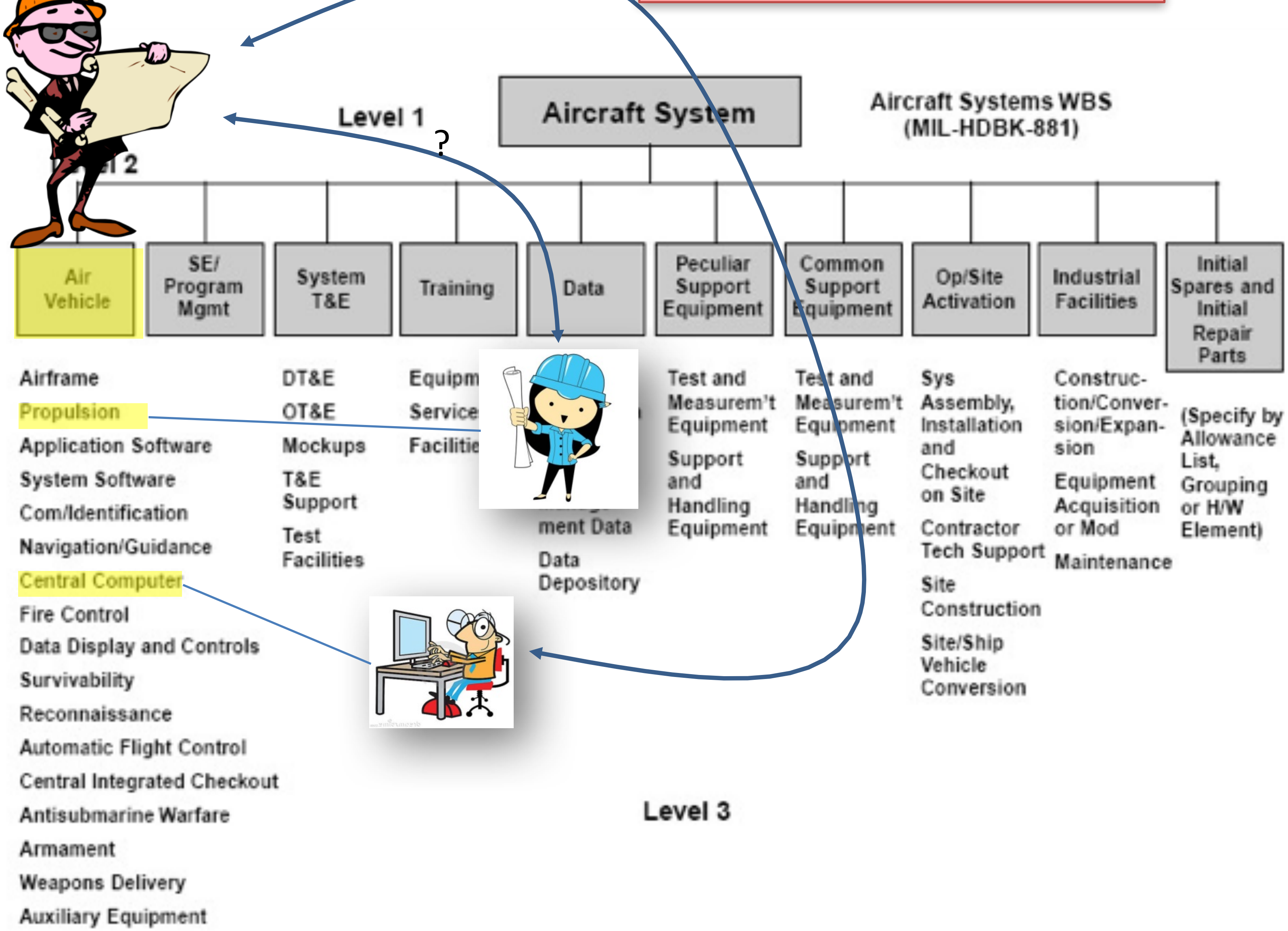

Level 1
Aircraft System
Aircraft Systems WBS
(MIL-HDBK-881)
Level 2
Air Vehicle
SE/ Program Mgmt
System T&E
Training
Data
Peculiar Support Equipment
Common Support Equipment
Op/Site Activation
Industrial Facilities
Initial Spares and Initial Repair Parts
Airframe
Propulsion
Application Software
System Software
Com/Identification
Navigation/Guidance
Central Computer
Fire Control
Data Display and Controls
Survivability
Reconnaissance
Automatic Flight Control
Central Integrated Checkout
Antisubmarine Warfare
Armament
Weapons Delivery
Auxiliary Equipment
DT&E
OT&E
Mockups
T&E Support
Test Facilities
Equipment
Services
Facilities
Tech Pubs
Engrg Data
Support Data
Management Data
Data Depository
Test and Measurem't Equipment
Support and Handling Equipment
Test and Measurem't Equipment
Support and Handling Equipment
Sys Assembly, Installation and Checkout on Site
Contractor Tech Support
Site Construction
Site/Ship Vehicle Conversion
Construction/Conversion/Expansion
Equipment Acquisition or Mod
Maintenance
(Specify by Allowance List, Grouping or H/W Element)
Level 3

Managing Complexity

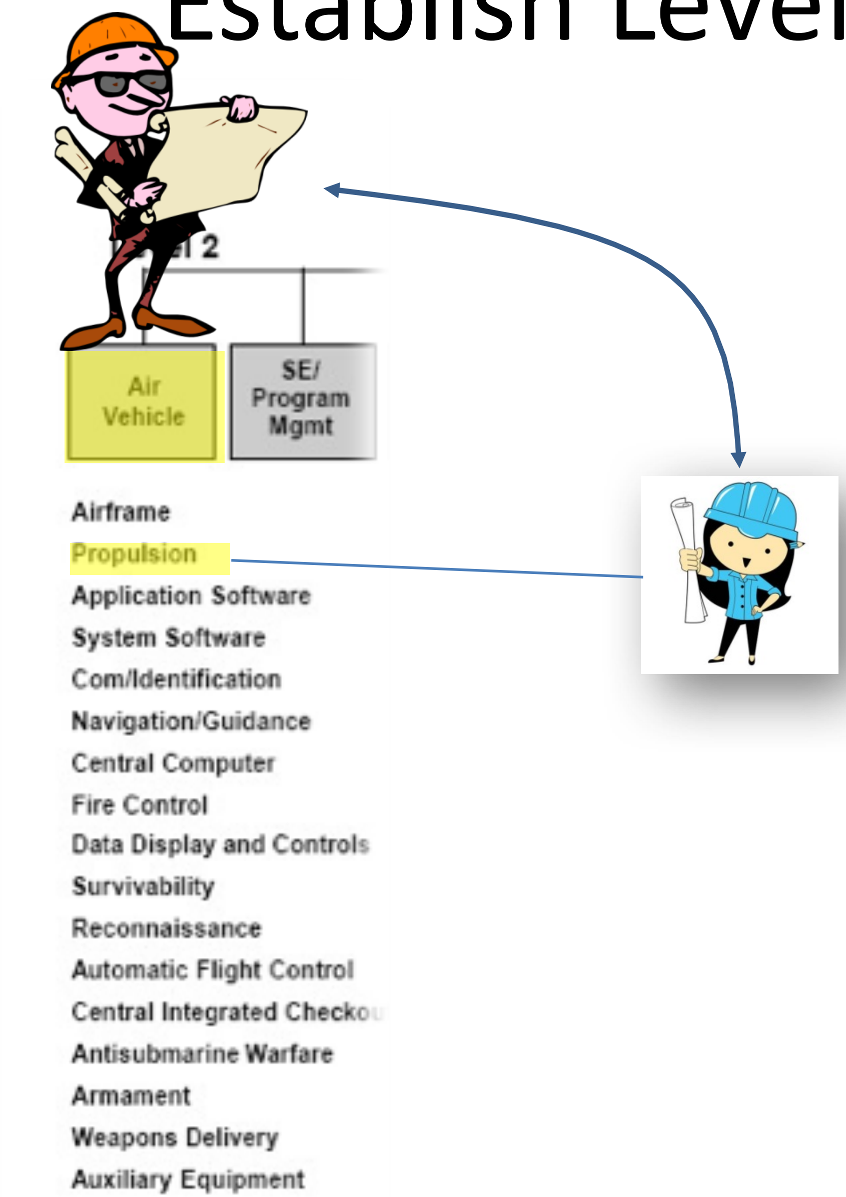
Abstraction

- A technique to manage complexity
- Establishing a level of complexity on which a person interacts with the system
- Suppressing the more complex details below the current level.

How they work together?



Establish Level of Complexity

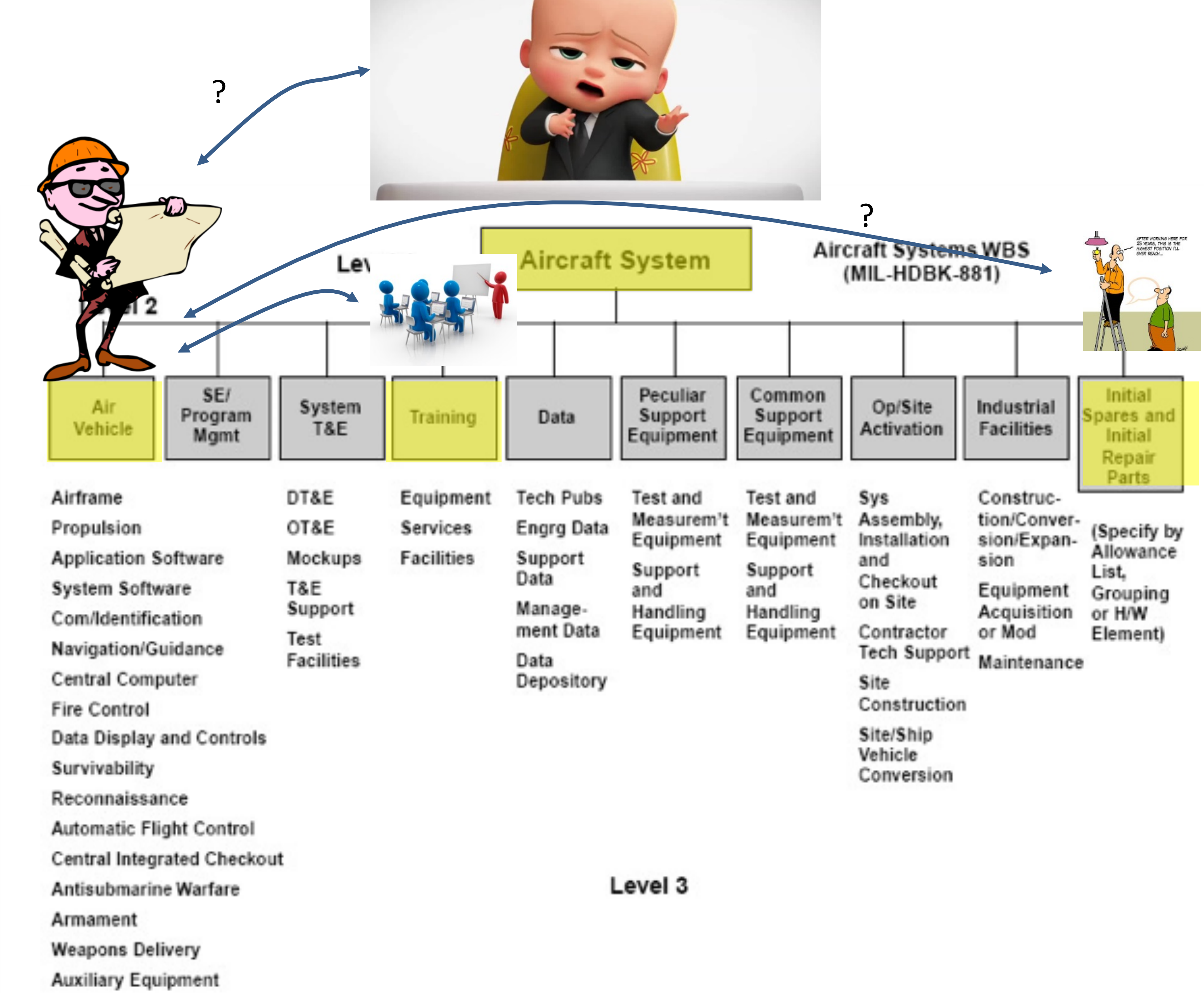


- The chief engineer want to know
  - How much fuel does the propulsion engine uses
  - How much force can the engine provide

The level of complexity that the chief engineer wants to know

- But NOT
  - ~~How exactly the engine works~~
  - ~~How many parts are there in the engine~~

The level of complexity suppressed from the chief

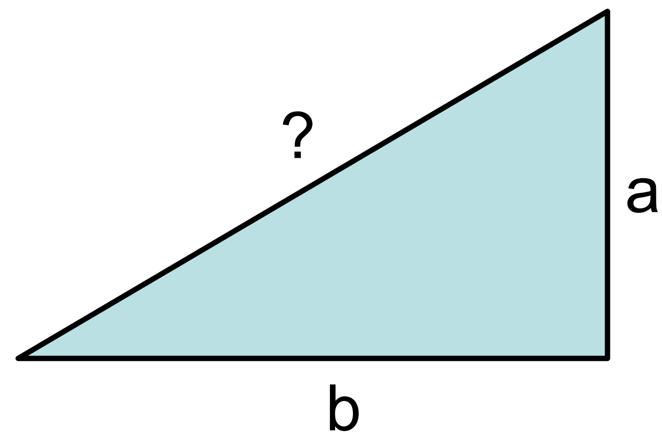

?
?
Aircraft System
Aircraft Systems WBS
(MIL-HDBK-881)
Air Vehicle
SE/ Program Mgmt
System T&E
Training
Data
Peculiar Support Equipment
Common Support Equipment
Op/Site Activation
Industrial Facilities
Initial Spares and Initial Repair Parts
Airframe
Propulsion
Application Software
System Software
Com/Identification
Navigation/Guidance
Central Computer
Fire Control
Data Display and Controls
Survivability
Reconnaissance
Automatic Flight Control
Central Integrated Checkout
Antisubmarine Warfare
Armament
Weapons Delivery
Auxiliary Equipment
DT&E
OT&E
Mockups
T&E Support
Test Facilities
Equipment
Services
Facilities
Tech Pubs
Engrg Data
Support Data
Manage-ment Data
Data Depository
Test and Measurem't Equipment
Support and Handling Equipment
Test and Measurem't Equipment
Support and Handling Equipment
Sys Assembly, Installation and Checkout on Site
Contractor Tech Support
Site Construction
Site/Ship Vehicle Conversion
Construc-tion/Conver-sion/Expan-sion
Equipment Acquisition or Mod
Maintenance
(Specify by Allowance List, Grouping or H/W Element)
Level 3

Compare:

```
def hypotenuse(a, b):
    return sqrt(sum_of_squares(a, b))

def sum_of_squares(x, y):
    return square(x) + square(y)

def square(x):
    return x * x
```

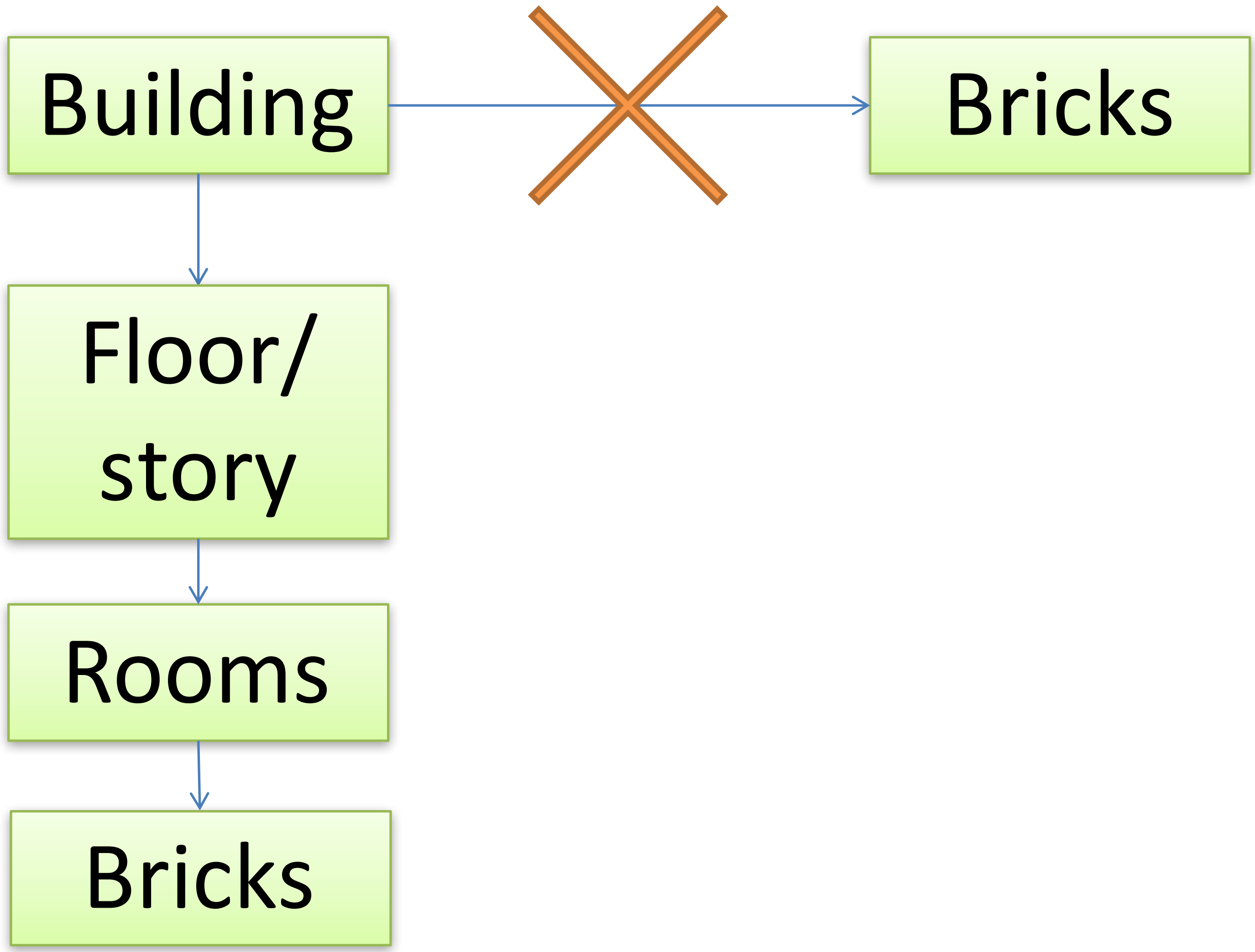


Versus:

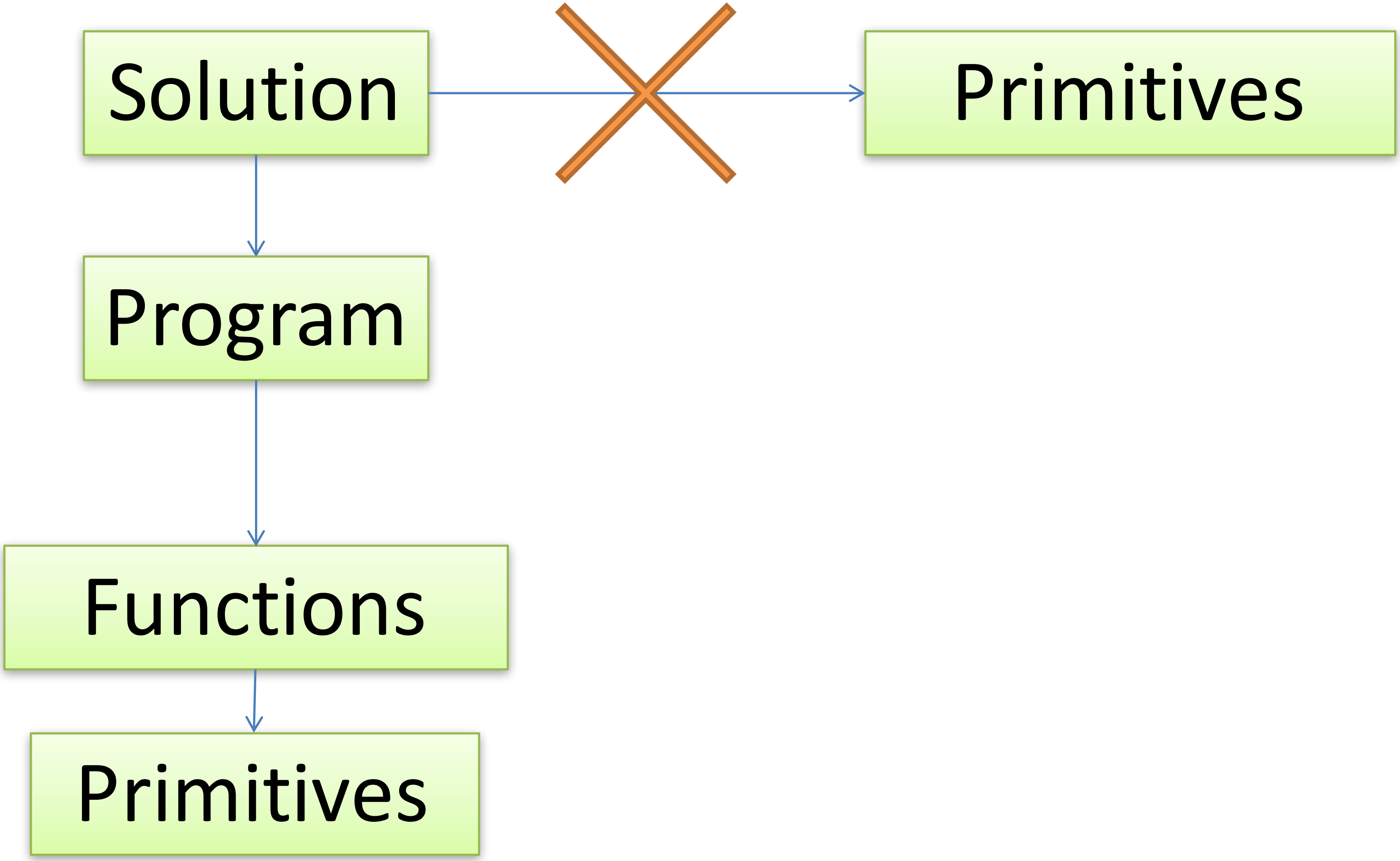
```
def hypotenuse(a, b):
    return sqrt((a*a) + (b*b))
```

What Makes a Good Abstraction?

1. Makes it more natural to think about tasks and subtasks



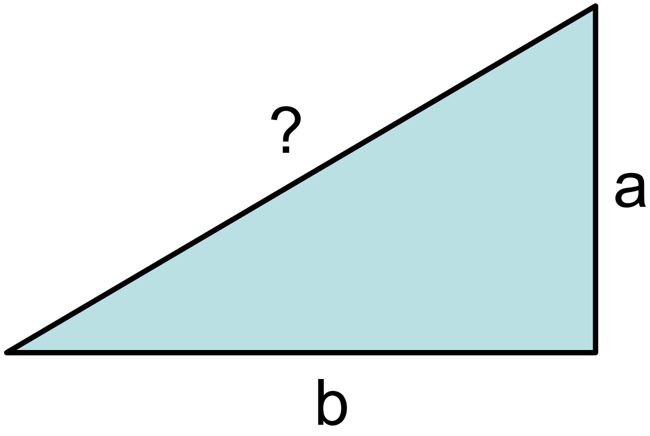
1. Makes it more natural to think about tasks and subtasks



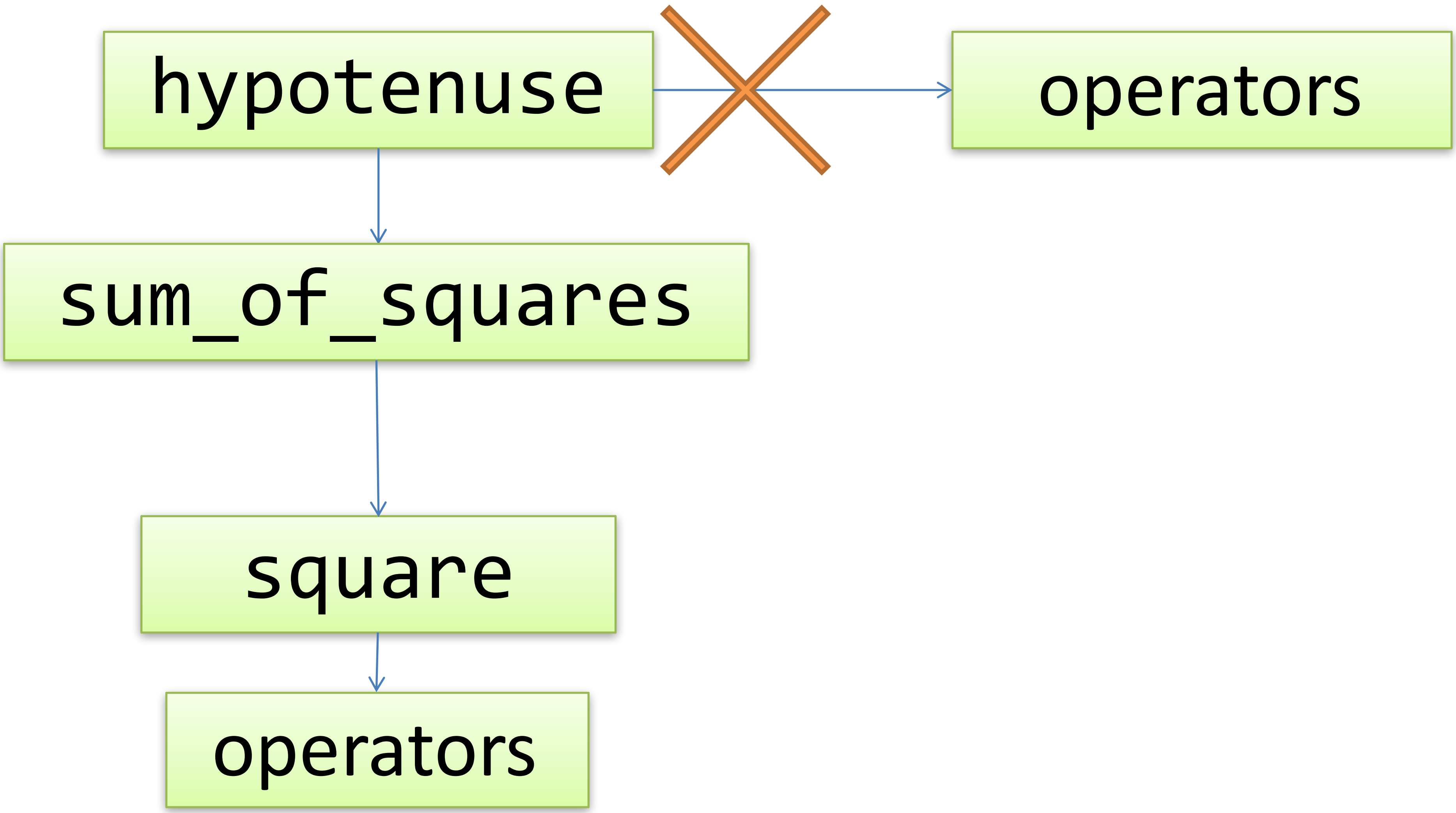
```
def hypotenuse(a, b):
    return sqrt(sum_of_squares(a, b))

def sum_of_squares(x, y):
    return square(x) + square(y)

def square(x):
    return x * x
```



1. Makes it more natural to think about tasks and subtasks



2. Makes programs easier to understand

```
def hypotenuse(a, b):
    return sqrt(sum_of_squares(a, b))

def sum_of_squares(x, y):
    return square(x) + square(y)

def square(x):
    return x * x
```

3. Captures common patterns

- E.g. computing binomial coefficient

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

- You won't write code like

```
def bc(n,k):
    a = code for computing 1x2x3…x n
    b = code for computing 1x2x3…x k
    c = code for computing 1x2x3…x(n-k)
    return a / (b * c)
```

Common Patterns

3. Captures common patterns

- E.g. computing binomial coefficient

$$\binom{n}{k} = \frac{n!}{k!\,(n-k)!}$$

- Write a function `factorial(n)` then

```
def bc(n,k):
    a = factorial(n)
    b = factorial(k)
    c = factorial(n-k)
    return a / (b * c)
```

4. Allows for code reuse

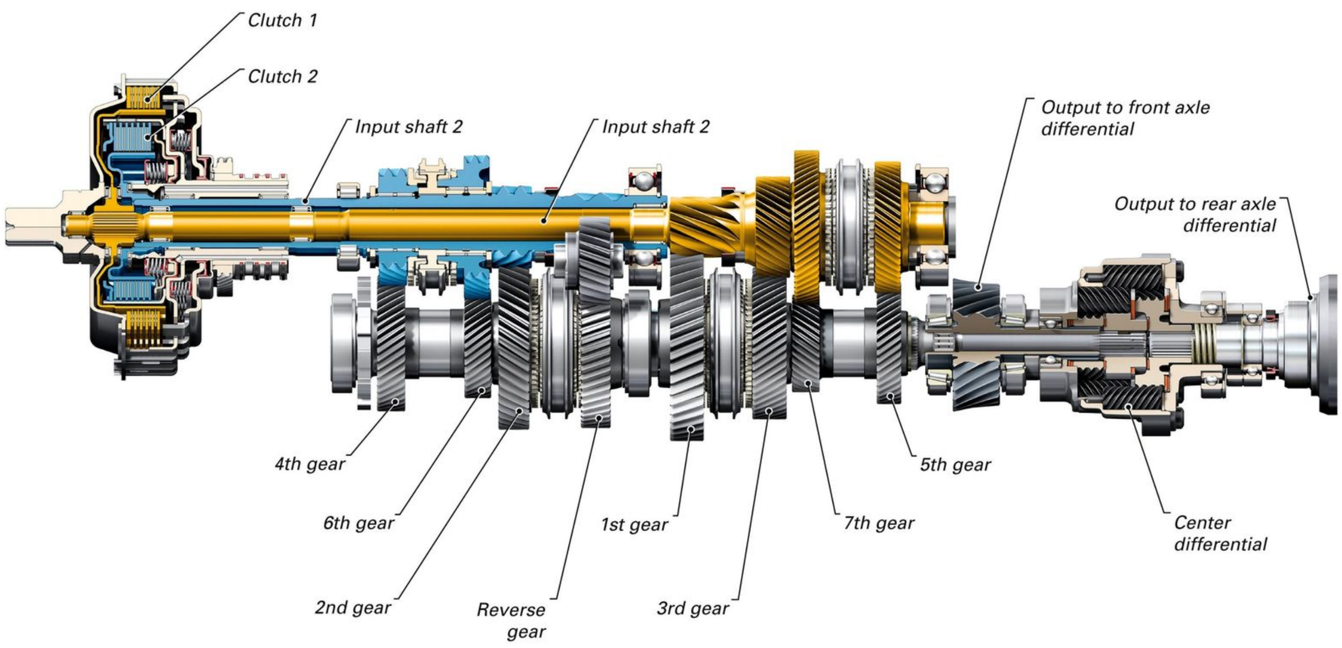
- Function `square()` used in `sum_of_squares()`.
- `square()` can also be used in calculating area of circle.

```
pi = 3.14159
def circle_area_from_radius(r):
  return pi * square(r)

def circle_area_from_diameter(d):
  return circle_area_from_radius(d/2)
```

5. Hides irrelevant details

- The structure of a gear box maybe interesting to some fans but not everyone who drives



Made of?

How doe

rk?

6. Separates specification from implementation

- Specification: WHAT IT DOES
  - E.g the function cos(x) compute the cosine of x

- Implementation: HOW IT DOES

$$\cos(x) = \cancel{\sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!}}$$

Different ways of implementing square(x)

```
def square(x):
        return x * x


def square(x):
        return x ** 2


def square(x):
    return exp(double(log(x)))
def double(x): return x + x
```

7. Makes debugging (fixing errors) easier

```
def hypotenuse(a, b):
    return sqrt((a + a) * (b + b))
```

Where is/are the bugs?

```
def hypotenuse(a,b):
    return sqrt(sum_of_squares(a,b))

def sum_of_squares(x, y):
    return square(x) * square(y)

def square(x):
    return x + x
```

Good Abstraction?

1. Makes it more natural to think about tasks and subtasks
2. Makes programs easier to understand
3. Captures common patterns
4. Allows for code reuse
5. Hides irrelevant details
6. Separates specification from implementation
7. Makes debugging easier

Program Design
Top-down Approach

pretend you have whatever you need

Sequence of Writing

1

```
def hypotenuse(a, b):
    return sqrt(sum_of_squares(a, b))
```

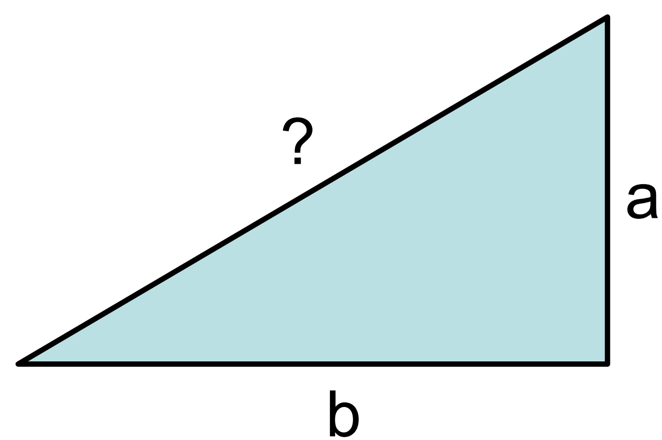
2

```
def sum_of_squares(x, y):
    return square(x) + square(y)
```

3

```
def square(x):
    return x * x
```

pretend you have
whatever you need



Another Example

- NTUC Comfort, the largest taxi operator in Singapore, determines the taxi fare based on distance traveled as follows:

Basic fare	Normal
Flag-Down (inclusive of 1st km or less)	\$3.00-\$3.40
Every 400m thereafter or less up to 10km	\$0.22
Every 350 metres thereafter or less after 10 km	\$0.22
Every 45 secs of waiting or less	\$0.22

Problem: Write a Python function that computes the taxi fare from the distance traveled.

How do we start?

Formulate the Problem

- We need a name!
  - Pick a meaningful one
  - Definitely not “foo”

Function

Formulate the Problem

- What are the
  - Input?
  - Output?

Formulate the Problem

- How exactly should we design the function?
  1. Try a few simple examples.
  2. Strategize step by step.
  3. Write it down and refine.

Solution

• What to call the function?	`taxi_fare`
• What data are required?	`distance`
• Where to get data?	function argument
• What is the result?	`fare`

Try a few simple examples

- e.g.#1: distance = 800 m, fare = \$3.00
- e.g.#2: distance = 3300 m,
  - fare = \$3.00 +roundup(2300/400) × \$0.22 = \$4.32
- e.g.#3: distance = 14500 m,
  - fare = \$3.00 +roundup(9000/400) × \$0.22 + roundup(4500/350)× \$0.22 = \$10.92

Basic fare	Normal
Flag-Down (inclusive of 1st km or less)	$\boxed{\$3.00}$-\$3.40
Every 400m thereafter or less up to 10km	\$0.22
Every 350 metres thereafter or less after 10 km	\$0.22
Every 45 secs of waiting or less	\$0.22

Pseudocode

- Case 1: distance <= 1000
  - fare = $3.00

- Case 2: 1000 < distance <= 10,000
  - fare = $3.00 + $0.22 * roundup((distance – 1000)/ 400)

- Case 3: distance > 10,000
  - fare = $3.00 +roundup(9000/400) + $0.22 * roundup((distance – 10,000)/ 350)

- Note: the Python function ceil rounds up its argument. math.ceil(1.5) = 2

Pseudocode (Refined)

- Case 1: distance <= 1000
  - fare = $3.00
- Case 2: 1000 < distance <= 10,000
  - fare = $3.00 + $0.22 * roundup((distance – 1000)/ 400)
- Case 3: distance > 10,000
  - fare = $**8.06** + $0.22 * roundup((distance – 10,000)/ 350)
- Note: the Python function ceil rounds up its argument. math.ceil(1.5) = 2

Solution

```
def taxi_fare(distance): # distance in metres
    if distance <= 1000:
        return 3.0
    elif distance <= 10000:
        return 3.0 +
            (0.22*ceil((distance-1000)/400))
    else:
        return 8.06 +
            (0.22*ceil((distance-10000)/350))

# check: taxi_fare(3300) = 4.32
```

Coping with Changes

- What if starting fare is increased to $3.20?
- What if 385 m is increased to 400 m?

Avoid Magic Numbers

- It is a terrible idea to hardcode constants (magic numbers):
  – Hard to make changes in future

- Define abstractions to hide them!

```
stage1 = 1000
stage2 = 10000
start_fare = 3.0
increment = 0.22
block1 = 400
block2 = 350

def taxi_fare(distance): # distance in metres
      if distance <= stage1:
            return start_fare
      elif distance <= stage2:
            return start_fare + (increment *
                   ceil((distance - stage1) /
block1))
      else:
            return taxi_fare(stage2) +
                  (increment * ceil((distance -
stage2) / block2))
```

Better to make them all CAPS for "normal" convention

How to I Manage My Own Code?

Let’s say I wrote some cool code

```
def square(x):
    return x * x

def singHappyBirthdayTo(name):
    print('Happy birthday To You!')
    print('Happy birthday To You!')
    print('Happy birthday to ' + name + '~')
    print('Happy birthday to You!!!')
```

- And I save it to a file called

my_cool_package.py

Then I can use it for another file

- Another file:

Same as the file name but without ".py"

```
import my_cool_package
from math import pi

def circle_area_by_radius(r):
    return pi * my_cool_package.square(r)


print(circle_area_by_radius(10))
```

Or

- Another file:

```
import my_cool_package as mcp
from math import pi

def circle_area_by_radius(r):
    return pi * mcp.square(r)


print(circle_area_by_radius(10))
```

Or

- Another file

```
from my_cool_package import square

def squared_sum(a,b):
    return square(a) + square(b)


print(squared_sum(3,4))
```

Or

- Another file

But in general, it’s not a good habit to name a variable/function so short that you cannot understand what it does

```
from my_cool_package import square as sq

def squared_sum(a,b):
    return sq(a) + sq(b)


print(squared_sum(3,4))

```