Using Objects

Welcome back to CS 2100!

Prof. Rasika Bhalerao

Poll: What gets printed?

```python
class Cat:
    def __init__(self, name: str, human: str):
        self.name = name
        self.human = human
        print(name * 3)

mini: Cat = Cat('Mini', 'Rasika')
```

1. `Mini`
2. `<__main__.Cat object at 0x10d380200>`
3. `MiniMiniMini`
4. (Nothing)

State and aliasing

When a variable holds an object, it holds a *reference* to that object. That object is stored somewhere in the memory of the computer, and the variable knows where to access it.

Multiple variables can hold references to the same object.

Alias: a second reference to an existing object (same place in computer's memory)

Copy: another object that looks exactly the same (different place in computer's memory)

```
original: List[int] = [1, 2, 3]

alias: List[int] = original
copy: List[int] = original.copy()

original[2] = 90

print(alias) # [1, 2, 90]
print(copy) # [1, 2, 3]
```

Passing mutable objects as arguments

If we pass an object as an argument to a function, an *alias* is created, not a copy.

```python
def sum_with_bad_manners(in_list: List[int]) -> int:
    sum: int = 0
    while (len(in_list) > 0):
        sum += in_list.pop()
    return sum


my_list: List[int] = [1, 2, 3, 4]
print(f'Sum: {sum_with_bad_manners(my_list)}') # Sum: 10
print(my_list) # []
```

Good manners: Functions should leave their args unchanged (unless they clearly state otherwise in the documentation).

```python
class Shirt:
    def __init__(self, size: int):
        self.size = size

def main() -> None:
    six_hundred: int = 600
    subtract_one(six_hundred)
    print(six_hundred)                          # 600

    shirt: Shirt = Shirt(600)
    shrink_shirt(shirt)
    print(shirt.size)                           # 599

def subtract_one(num: int) -> None:
    num -= 1

def shrink_shirt(shirt: Shirt) -> None:
    shirt.size -= 1
```

Poll: Which of these makes it print 4?

```python
def main() -> None:
    shirt: Shirt = Shirt(600)
    modify_shirt(shirt)
    print(shirt.size)

if __name__ == '__main__':
    main()
```

1.

```python
def modify_shirt(shirt: Shirt) -> None:
    shirt.size = 4
```

2.

```python
def modify_shirt(shirt: Shirt) -> None:
    shirt = Shirt(4)
```

3.

```python
def modify_shirt(shirt: Shirt) -> Shirt:
    return Shirt(4)
```

The `__str__()` function

Every class has a `__str__()` method.

We can overwrite it with our own `__str__()` method.

When we print an object, it implicitly calls the `__str__()` method.

The default `__str__()` method is not helpful.

```
mini = Cat('Mini', 'Rasika')
print(mini) # <__main__.Cat object at 0x1095ca790
```

The `__str__()` function

Why is it different when we print a list (which is also an object)?
We usually write our own `__str__()` method that returns a more helpful `str` for that class.

```python
class Cat:
    ...
    def __str__(self):
        return f'{self.name} meows to {self.human}'


mini = Cat('Mini', 'Rasika')
print(mini) # Mini meows to Rasika
```

Poll: What is printed?

```python
class Cat:
    def __init__(self, name: str, human: str):
        self.name = name
        self.human = human

    def __str__(self) -> str:
        print('MUAHAHAHAHHA')
        return self.name

mini: Cat = Cat('Mini', 'Rasika')
print(mini)
```

1. `Mini`
2. `MUAHAHAHAHHA` // `Mini`
3. `<__main__.Cat object at 0x10d380200>`
4. `MUAHAHAHAHHA` // `<__main__.Cat object at 0x10d380200>`

Abstraction

We organize lines of code into functions, functions into classes, classes into...

"Absraction": the prodecure of grouping more granular things into less granular groups

Benefits of abstraction we already saw in functions:

- reuse code without redundancy
- hide implementation details
- break down a problem into smaller, more manageable pieces

The same applies to the idea of putting methods into classes.

Abstraction

Hard to debug or modify:

```
length: int = 5
width: int = 3

print(get_area_of_rectangle(
    length, width))

length = 6
width = 4

print(get_perimeter_of_rectangle(
    length, width))
```

Put relevant perimeter and area methods into a class for each shape:

```
table: Rectangle = Rectangle(5, 3)
print(table.area())

print(Rectangle(6, 4).perimeter())

chair: Square = Square(4)
print(chair.area())
```

Benefits of abstraction:

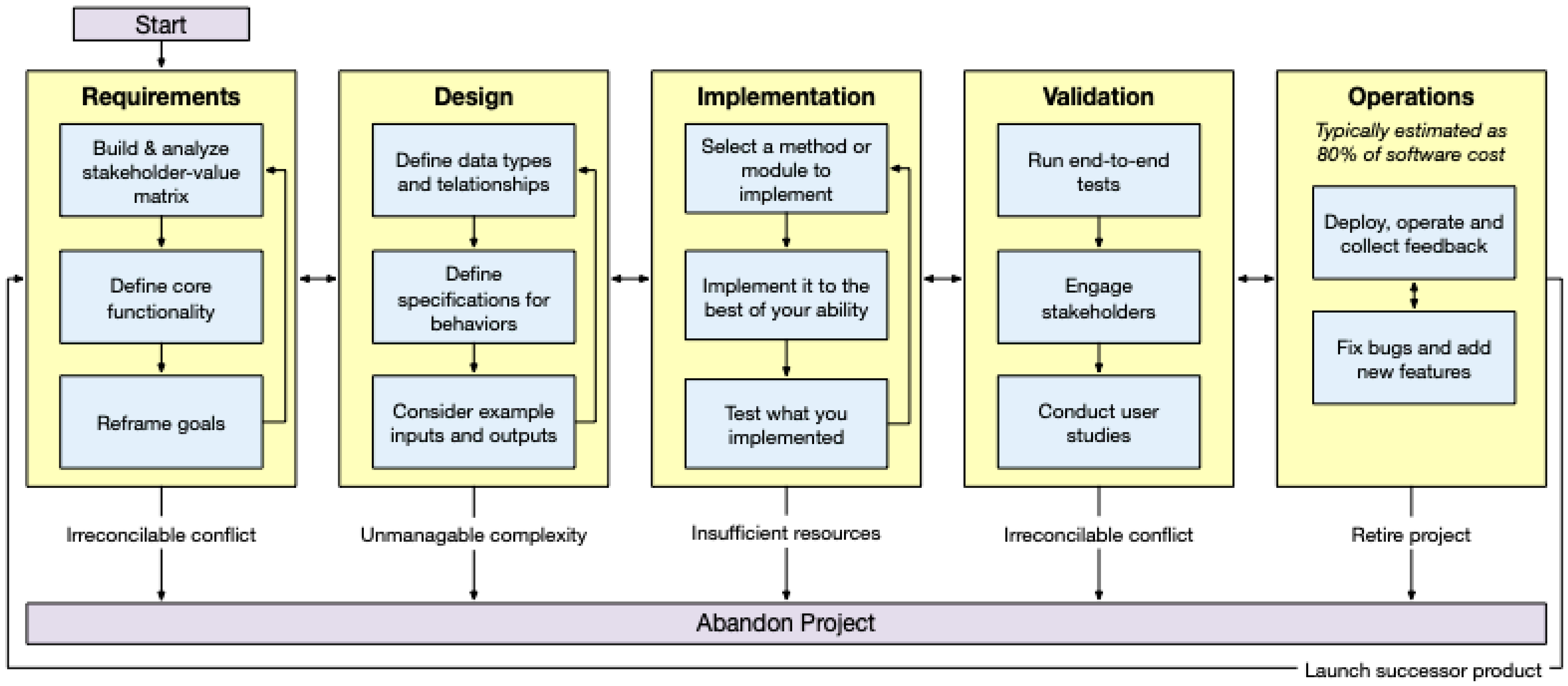
- We can use the same code multiple times without re-writing it
- It's easier to read
- It's easier to maintain and adapt the code later on
- It's easier to test behaviors in isolation
- Each variable, function, and class has a single, clear responsibility

The Single Responsbility Principle

A core principle for writing code is that every component of our code must have a single purpose. This makes our code easier to read, test, and maintain.

Component	Does not Follow Single Responsibility Principle
Variable	`age_or_nonexistent: int = -1 # negative if nonexistent`
Function	`def read_file_compute_average_print_score(filename: str) -> None:`
Class	`class FileManagerAndOutputFormatter`

The Program Design and Implementation Process



Start
Requirements
Build & analyze stakeholder-value matrix
Define core functionality
Reframe goals
Design
Define data types and telationships
Define specifications for behaviors
Consider example inputs and outputs
Implementation
Select a method or module to implement
Implement it to the best of your ability
Test what you implemented
Validation
Run end-to-end tests
Engage stakeholders
Conduct user studies
Operations
Typically estimated as 80% of software cost
Deploy, operate and collect feedback
Fix bugs and add new features
Irreconcilable conflict
Unmanagable complexity
Insufficient resources
Irreconcilable conflict
Retire project
Abandon Project
Launch successor product

Will revisit multiple times this semester, deeper in CS 3100

Open-ended poll:

What would happen if we directly jumped to the Implementation phase, skipping the Requirements and Design phases? I.e., what would happen if we started by writing code without first designing the project and planning out its pieces?

Poll:

1. What is your main takeaway from today?
2. What would you like to revisit next time?