

A decorative graphic in the top-left corner of the page consists of several thick, diagonal bars of various colors (yellow, grey, orange, red, blue, cyan, green, and pink) arranged in a layered, overlapping fashion, creating a dynamic, abstract pattern.

# OOP Design and Specification

**ThanhVu (Vu) Nguyen**

August 29, 2024 (latest version available on [nguyenthanhvuh.github.io/class-oo/oop.pdf](https://nguyenthanhvuh.github.io/class-oo/oop.pdf))

# Preface

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
1.1	Decomposition . . . . .	4
1.2	Abstraction . . . . .	4
<b>2</b>	<b>Procedural Abstraction</b>	<b>6</b>
2.1	Specifications . . . . .	7
2.1.1	Specifications of a Function . . . . .	7
<b>A</b>	<b>Lectures</b>	<b>9</b>

# Chapter 1

## Introduction

This book will guide you through the fundamentals of constructing high-quality software using a modern **object-oriented programming** (OOP) approach. We will use *Python* for demonstration, but the concepts can be applied to any object-oriented programming language. The goal is to develop programs that are reliable, efficient, and easy to understand, modify, and maintain.

### 1.1 Decomposition

As the size of a program increases, it becomes essential to *decompose* the program into smaller, independent programs (or functions or modules). This decomposition process allows for easier management of the program, especially when multiple developers are involved. This makes the program easier to understand and maintain.

Decomposition is the process of breaking a complex program into smaller, independent, more manageable programs, i.e., “divide and conquer”. It allows programmer to focus on one part of the problem at a time, without worrying about the rest of the program.

**Example** Fig. 1.1 shows a Python implementation of *Merge Sort*, a classic example of problem decomposition. It breaks the problem of sorting a list into simpler problems of sorting smaller lists and merging them.

### 1.2 Abstraction

*Abstraction* is key concept in OOP that allows programmers to hide the implementation details of a program and focus on the essential features. In an OOP language such as Python, you can abstract problems by creating functions, classes, and modules that hide the underlying implementation details.

```

def merge_sort(lst):
    if len(lst) <= 1:
        return lst

    mid = len(lst) // 2
    left = merge_sort(lst[:mid])
    right = merge_sort(lst[mid:])
    return merge(left, right)

def merge(left, right):
    result = []
    i = j = 0

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1

    result.extend(left[i:])
    result.extend(right[j:])
    return result

```

Fig. 1.1: Decomposition example: Mergesort

```

class Mammal:
    def __init__(self, name):
        self.name = name

    def speak(self): pass

class Dog(Mammal):
    def speak(self): return "Woof!"

class Cat(Mammal):
    def speak(self): return "Meow!"

```

Fig. 1.2: Decomposition example: Mergesort

**Example** Fig. 1.2 demonstrates an abstraction for different types of mammals. Mammals such as Dog and Cat share common behaviors such as making noise (speak). We can create a class ‘Mammal’ that defines these common behaviors, and then create subclasses Dog and Cat that inherit from Mammal and define their own unique behaviors.

## Chapter 2

# Procedural Abstraction

One common mechanism to *procedural abstraction*, which achieves abstraction is through the use of functions (procedures). By separating procedure definition and invocation, we make two important methods of abstraction: abstraction by parameterization and abstraction by specification.

**Abstraction by Parameterization** This allows you to generalize a function by using parameters. By abstracting away the specific data with *parameters*, the function becomes more versatile and can be reused in different situations. [Fig. 2.1](#) shows an example of abstract parameterization. The `cal_area` function calculates the area of a rectangle given its length and width, which are passed as parameters.

**Abstraction by Specification** This focuses on what the function does (e.g., sorting), instead of how it does it (e.g., using quicksort or mergesort algorithms). By defining a function's behavior through *specifications*, developers can implement the function in different ways as long as it fulfills the specifications. Similarly, the user can use the function without knowing the implementation details.

[Fig. 2.2](#) shows an example of abstraction by specification. The `sort_items` function sorts a list of items in ascending order; the user only needs to know it does sorting but does not need to know the sorting algorithm used.

```
def cal_area(length, width):  
    return length * width  
  
# can be used with different values for length and width.  
area1 = cal_area(5, 10)  
area2 = cal_area(7, 3)
```

Fig. 2.1: Example: Abstract Parameterization

```

def sort_items(items):
    """
    Sorts a list of items in ascending order.

    :pre: List of items to sort.
    :post: A new list of sorted items.
    """
    return sorted(items)

# The user of this function only needs to know that it sorts items,
# without needing to understand the sorting algorithm used.
sorted_list = sort_items([5, 3, 8, 1])

```

Fig. 2.2: Abstraction by Specification: sorting

## 2.1 Specifications

We define abstractions through specifications, which describe what the abstraction is intended to do rather than how it should be implemented. This allows specifications to be much more concise and easier to read than the corresponding code.

Specifications which can be written in either *formal* or *informal languages*. Formal specifications have the advantage of being precise and unambiguous. However, in practice, we often use informal specifications, describing the behavior of the abstraction in plain English (e.g., the `sorting` example in Fig. 2.2). Note that a specification is not a programming language or a program. Thus, our specifications won't be written in code (e.g., in Python or Java)

### 2.1.1 Specifications of a Function

The specification of a function consists of a *header* and a *description* of its behavior. The header gives the signature of the function, including its name, parameters, and return type. The description describes the function's behavior, including its preconditions and postconditions.

**Header** The header provides the *name* of the function, the number, order, and types of its *parameters* (inputs), and the type of its return value (output). For instance, the headers for the `sort_items` function in Fig. 2.2 and the `cal_area` function in Fig. 2.1 are as follows

```

def sort_items(items: list) -> list: ...

def calc_area(length: float, width: float) -> float: ...

```

**Preconditions and Postconditions** A typical function specification in an OOP language such as Python includes three main parts: the *preconditions* (also called

“requires”) and *Postconditions* (also called “effects”). Preconditions describe the conditions that must be true before the function is called. Typically these state the constraints or assumptions about the input parameters. Postconditions describe the conditions that will be true after the function is called. These typically state the expected results or outcomes of the function. The clauses are usually written as *comments* above the function definition, making them easily accessible within the code.

```
def sort_items(items: list) -> list:
    """
    Sorts a list of items in ascending order.

    :pre: List of sortable items.
    :post: A sorted permutation of the input list.
    """
    ...

def calc_area(length: float, width: float) -> float:
    """
    Calculates the area of a rectangle given its length and width.

    :pre: None
    :post: The area of the rectangle.
    """
    ...
```

## Precision



Appendix A

Lectures