OOP Design and Specification

ThanhVu (Vu) Nguyen

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

Preface

Contents

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.

```python
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

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

    def speak(self): pass
```

```python
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. Mamals such as Dog and Cat share common behaviors such as making noise (speak). We can create a class 'Mamal" that defines these common behaviors, and then create subclasses `Dog` and `Cat` that inherit from `Mamal` 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 mergsort 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.

```python
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

```python
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

Appendix A

Lectures