OOP Design and Specification

ThanhVu (Vu) Nguyen

September 5, 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 a 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 `exists` method return true if the item is found in a list of sorted items. The user only needs to provide a sorted list, but does not need to know what algorithm is used or implemented to determine if the item exists in the list.

```
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 exists(items):
    """
    Find an item in a list of sorted items.

    Pre: List of sorted items
    Post: True if the item is found, False otherwise.
    """
    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 exists(items: list) -> bool: ...

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

Note that in a language like Java, the header also provides *exceptions* that the function may throw.

Preconditions and Postconditions A typical function specification in an OOP language such as Python includes: *Preconditions* (also called the "requires" clause) and *Postconditions* (also called the "effects" clause). Preconditions describe the conditions that must be true before the function is called. Typically these state the constraints or assumptions about the input parameters. If there are no preconditions, the clause is often written as `None`.

Postconditions, under the assumption that the preconditions are satisfied, describe the conditions that will be true after the function is called. These typically state the expected results or outcomes of the function. Moreover, they often describe the relationship between the inputs and outputs.

The clauses are usually written as *comments* above the function definition, making them easily accessible within the code.

```
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.
    """
    ...
```

For example, the specification of the `calc_area` function in Fig. 2.1 has (i) no preconditions and (ii) the postcondition that the function returns the area of a rectangle given its length and width. Similarly, the `exists` function in Fig. 2.2 has the specification that given a list of sorted items (precondition), it returns true if the item is found in the list, and false otherwise (postcondition). Note how the specification is written in plain English, making it easy to understand for both developers and users of the function.

Modifies Another common clause in a function specification is *modifies*, which describes the inputs that the function modifies. This is particularly useful for functions that modify their input parameters.

```
def add_to_list(input_list, value):
"""
Adds a value to the input list.

Pre: None
Post: Value is added to the input list.
Modifies: the input list
"""
...
```

2.1.2 In-class Exercise

See IC1-B for in-class exercises on specifications.

2.2 Designing Specifications

2.2.1 Weaker Preconditions

A condition is weaker than another if it is implied by the other or having less constraints than the other. For example, the condition that the input $x \leq 5$ is weaker than $x \leq 10$ or that the input list is not sorted is weaker than the list is sorted (which is weaker than the list that is both sorted and has no duplicates). A function with a weaker precondition is preferred because it can handle a larger class of inputs (e.g., can handle both sorted and unsorted lists). The weakest precondition is *None*, which means the function can be called with any input.

Total vs Partial Functions A function is *total* if it is defined for all legal inputs; otherwise, it is *partial*. Thus a function with no precondition is total, while a function with the strongest possible precondition is partial. Total functions are preferred because they can be used in more situations.

The `calc_area` function in Fig. 2.1 and `add_to_list` function in Fig. 2.2 are total because they can be called with any input. The `exists` function in Fig. 2.2 is partial because it only works with sorted lists.

Appendix A

Lectures

A.1 Module 1: Overview

- Syllabus (no cheating)
- Overview
  - Decomposition (§1.1)
  - Abstraction (§1.2)
  - Abstraction by Parameterization (Fig. 2.1)
  - Abstraction by Specification (Fig. 2.2)
  - Specifications (§2.1)

- Break 1 (25 mins): 5 min break, 20 min IC1-A exercise
- Correctness Overview
  - Ideally: Satisfies preconditions and postconditions
  - 4 scenarios to consider
    1. Precondition is satisfied, postcondition is satisfied: correct
    2. Precondition is satisfied, but postcondition is not: incorrect
    3. Precondition not satisfied, but postcondition is: correct
    4. Precondition not satisfied, but postcondition is not: correct
  - Preconditions are the responsibility of the caller (the client)
  - Postconditions are the responsibility of the developer (the supplier, i.e., you)

  - For many non-OOP programs, this is relatively straightforward and can be checked automatically. Things become quite thorny when dealing with OOPs (e.g., inheritence)

- Break 2 (25 mins) : 5 min break, 20 min IC1-B exercise

A.2 Module 2: