OOP Design and Specification

ThanhVu (Vu) Nguyen

September 10, 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.

```
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 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 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 `target` item is found in a list of sorted `items`. The user only needs to provide a sorted list and a target, 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

```python
def exists(items:List[int], target:int) -> bool:
    """
    Find an item in a list of sorted items.

    Pre: List of sorted items
    Post: True if the target is found, False otherwise.
    """
    ...

# The user only needs to know that this function checks for the existence of an item
#  in a sorted list, without needing to understand the search algorithm/implementation.
```

Fig. 2.2: Abstraction by Specification

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

```python
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 Weak Pre-conditions

For pre-conditions, we want as weak a constraint as possible to make the function more versatile, allowing it to handle a larger class of inputs. A condition is weaker than another if it is implied by the other or having less constraints than the other. For example, the condition $x \leq 5$ 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). The *weakest* precondition is *True*, which indicates no constraints on the input.

2.2.2 Strong Post-conditions

In contrast, for post-conditions, we want as strong a condition as possible to ensure that the function behaves as expected. A condition is stronger than another if it implies the other or that its constraints are a strict subset of the other. For example, the condition $x \leq 10$ is stronger than $x \leq 5$ or that the input list is sorted is stronger than the list is not sorted.

2.2.3 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, especially when the function is used publicly or in a library where the user may not know the input constraints. Partial functions can be used when the function is used internally, e.g., a helper or auxiliary function and the caller is knowledgeable and can ensure its preconditions are satisfied.

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.

Turning Partial Functions into Total Functions It is often possible to turn a partial function into a total function in two steps. First, we move preconditions into postconditions and specify the expected behavior when the precondition is not satisfied. Second, we modify the function to handle the cases when the preconditions are not satisfied. For example, the `exists` function in Fig. 2.2 is turned into the total function shown in Fig. 2.3.

2.2.4 No implementation details

The specification should not include any implementation details, such as the algorithm used or the data structures employed. This allows the function to be imple-

```python
def exists(items: List[int], target: int) -> bool:
    """
    Find an item in a list of sorted items.

    Pre: True
    Post: If the input items are not sorted, raise an exception.
          Return True if the item is found, False otherwise.


    """

    if not is_sorted(items):
        raise Exception(...)
```

Fig. 2.3: Total Specification for the program in Fig. 2.2

mented in different ways as long as it satisfies the specification. For example, the `exists` function in Fig. 2.2 does not specify the search algorithm used to find the item in the list.

Some common examples to avoid include the mentioning of specific data structures (e.g., arrays), algorithms (e.g., quicksort or mergesort). Also avoid specifications mentioning indices because this implies the use of arrays.

Chapter 3

Data Abstraction

This chapter focuses on *abstract data type* (ADT), a foundation of OOP and key concept in programming that allows developers to separate how data is implemented from how it behaves. Through ADT, programmers can create new data types relevant to their application. These ADTs consist of objects and associated operations.

An ADT has two main components: *parameterization* and *specification*. Parameterization involves using parameters for flexibility, while specification means including operations as part of the data type, which abstracts away the underlying data representation. This ensures that even if the data structure changes, programs that rely on it remain unaffected, as they only interact with the operations rather than the data's internal structure.

By abstracting data, developers can postpone decisions about data structures until they fully understand how the data will be used, leading to more efficient programs. ADT is also beneficial during program maintenance, as changes to the data structure affect only the type's implementation, not the modules using it.

3.1 Specifications of an ADT

As with functions (§2, the specification for an ADT defines its behavior without being tied to a specific implementation. The specification explains what the operations on the data type do, allowing users to interact with objects only via methods, rather than accessing the internal representation.

Structure of an ADT In a modern OOP language such as Python or Java, data abstractions are defined using *classes*. Each class defines a name for the data type, along with its constructors and methods.

Fig. 3.1 shows a class template in Python, which consists of three main parts. The *overview* describes the abstract data type in terms of well-understood concepts, like mathematical models or real-world entities. For example, a stack could be described using mathematical sequences. It also indicates whether the objects of this type are

```
class DataType:
    """
    OVERVIEW: A brief description of the data type and its objects.
    """

    def __init__(self, ...):
        """
        Constructor to initialize a new object.
        """

    def method1(self, ...):
        """
        Method to perform an operation on the object.
        """
```

Fig. 3.1: Abstract Data Type template

mutable (their state can change) or immutable (their state cannot change). The *Constructor* initializes a new object, setting up any initial state required for the instance. Finally, *methods* define operations users can perform on the objects. These methods allow users to interact with the object without needing to know its internal representation. In Python, `self` is used to refer to the object itself, similar to `this` in Java or C++.

Note that as with procedural specification (§2), the specifications of constructors and methods of an ADT do not include implementation details. They only describe what the operation does, not how it is done. Moreover, they are written in plain English as code comment.

3.1.1 Example: `IntSet` ADT

Fig. 3.2 gives a specification for an `IntSet` ADT, which represents unbounded set of integers. The ADT is *mutable*, i.e., its state can change, and includes a constructor to initialize an empty set, and methods to insert, remove, check membership, get the size, and choose an element from the set.

3.2 Implementing ADT

To implement an ADT, we first choose a *representation* (**rep**) for its objects, then design constructors to initialize it correctly, and methods to interact with and modify the rep. For example, we can use a `list` (or vector) as the rep of `IntSet` in Fig. 3.2.

3.3 Reasoning

Repr Invariant The *representation invariant* (repr-inv) is a condition that must be true for all objects of a class. It is a key part of the specification of an ADT,

```python
class IntSet:
    """
    OVERVIEW: IntSets are unbounded, mutable sets of integers.
    This implementation uses a list to store the elements, ensuring no duplicates.
    """
    def __init__(self):
        """
        Constructor
        EFFECTS: Initializes this to be an empty set.
        """
        self.els = []  # the representation (list)

    def insert(self, x: int):
        """
        MODIFIES: self
        EFFECTS: Adds x to the elements of this set if not already present.
        """
        if self.get_index(x) < 0:
            self.els.append(x)

    def remove(self, x: int):
        """
        MODIFIES: self
        EFFECTS: Removes x from this set if it exists.
        """
        i = self.get_index(x)
        if i < 0:
            return
        self.els[i] = self.els[-1]  # Replace with the last element
        self.els.pop()  # Remove the last element

    def is_in(self, x: int) -> bool:
        """
        EFFECTS: Returns True if x is in this set, otherwise False.
        """
        return self.get_index(x) >= 0

    def get_index(self, x: int) -> int:
        """
        EFFECTS: If x is in this set, returns the its index, else returns -1.
        """
        for i, element in enumerate(self.els):
            if x == element:
                return i
        return -1

    def size(self) -> int:
        """
        EFFECTS: Returns the number of elements in this set (its cardinality).
        """
        return len(self.els)

    def choose(self) -> int:
        """
        EFFECTS: If this set is empty, raises an Exception.
        Otherwise, returns an arbitrary element of this set.
        """
        if len(self.els) == 0:
            raise Exception(...)
        return self.els[-1]  # Returns the last element arbitrarily
```

Fig. 3.2: The IntSet ADT

ensuring that the object's internal representation is consistent with the ADT's behavior. For example, the representation invariant for the `IntSet` ADT in Fig. 3.2 is that all elements in the set are unique.

3.4 Mutability

3.4.1 In-class Exercise: Immutable Queue

Consider the mutable `Queue` implementation in Fig. 3.3.

1. Rewrite Queue to be *immutable*. Keep the representation variables `elements` and `size`.
2. Do the right thing with `enQueue()`.
3. Do the right thing with `deQueue()`.

```python
class Queue:
    """
    A generic Queue implementation using a list.
    """

    def __init__(self):
        """
        Constructor
        Initializes an empty queue.
        """
        self.elements = []
        self.size = 0

    def enqueue(self, e):
        """
        MODIFIES: self
        EFFECTS: Adds element e to the end of the queue.
        """
        self.elements.append(e)
        self.size += 1

    def dequeue(self):
        """
        MODIFIES: self
        EFFECTS: Removes and returns the element at the front of the queue.
        If the queue is empty, raises an IllegalStateException.
        """
        if self.size == 0:
            raise Exception(...)

        result = self.elements.pop(0)  # Removes and returns the first element
        self.size -= 1
        return result

    def is_empty(self):
        """
        EFFECTS: Returns True if the queue is empty, False otherwise.
        """
        return self.size == 0
```

Fig. 3.3: Mutable Queue

Appendix A

In-Class exercises

Appendix B

Lectures

B.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

B.2 Module 2: