OOP Design and Specification

ThanhVu (Vu) Nguyen

September 16, 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)
```

```python
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. 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

Procedural abstraction is a fundamental concept in programming that allows developers to create functions (methods) that hide the implementation details of a program. By abstracting away the details, developers can focus on the essential features of the program, making it easier to understand, modify, and maintain.

By separating procedure definition and invocation, we make two important methods of abstraction: abstraction by parameterization and abstraction by specification.

Abstraction by Parameterization This generalizes a function by using *parameters*. This allows the function to be used with different input values, making it more versatile and reusable. 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.

```
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

Abstraction by Specification This specifies on what the function does (e.g., sorting), instead of how it does it (e.g., using quicksort or mergsort algorithms, implemented in C). 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.

```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.
# They don't need to know the search algorithm or 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.

```python
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.

```python
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: User Equality

This exercise touches on some thorny issues with inheritance. There is a lot going on in this example, but it is a good exercise to understand the subtleties of inheritance.

1. First, look at the Javadoc to understand the behaviors `equals()` (while the specification is for Java, the idea is the same in Python).

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

    def __eq__(self, other):
        if not isinstance(other, User):
            return False
        return self.name == other.name
```

Fig. 2.3: User class

```python
class SpecialUser(User):
    """Don't do this until you've done with User"""

    def __init__(self, name, id):
        super().__init__(name)
        self.id = id

    def __eq__(self, other):
        if not isinstance(other, SpecialUser):
            return False
        return super().__eq__(other) and self.id == other.id
```

Fig. 2.4: SpecialUser class

- Specifically, read carefully the *symmetric*, *reflexive*, and *transitive* properties of `equals()`.
- Ignore *consistency*, which requires that if two objects are equal, they remain equal.

2. For the `User` class in Fig. 2.3, does `equals()` satisfy the three equivalence relation properties? If not, what is the problem?
    - Come up with several concrete test cases (e.g., create various `User` instances) to check the properties.
    - If there is a problem, show the test case that demonstrates the problem.
    - Explain why the problem occurs and come up with a fix.
3. So the same analysis for the `SpecialUser` class in Fig. 2.4.

2.2 Designing Specifications

When designing specifications, it is important to consider several factors to ensure that the function is well-defined and can be used effectively. These factors include the *strength* of the pre- and post-conditions, whether the function is *total* or *partial*, and the *avoiding implementation details* in the specification.

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 functions `calc_area` function in Fig. 2.1 and `add_to_list` 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, e.g., throws an `Exception`. Second, we modify the function to satisfy the new specification, i.e., handling 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. 2.5.

2.2.4 In-class Exercise: Partial and Total Specifications for `tail`

Consider the following code:

```
def tail(my_list):
    result = my_list.copy()
```

```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. 2.5: Total Specification for the program in Fig. 2.2

```python
    result.pop(0)
    return result
```

- What does the implementation of `tail` do in each of the following cases? You might want to see the Python document for `pop`. How do you know: Running the code or reading Python document?
  - `list = None`
  - `list = []`
  - `list = [1]`
  - `list = [1, 2, 3]`
- Write a *partial specification* for `tail`
- Rewrite the specification to be *total*. Use *exceptions* as needed.

2.2.5 No implementation details

The specification should not include any implementation details, such as the algorithm used or the data structures employed. This improves flexibility as it allows the function to be implemented 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, indices), algorithms (e.g., quicksort or mergesort), and exceptions (e.g., related to IndexError). Instead, focus on the behavior of the function, such as the input and output relationships, the constraints on the input, and the expected results. Also avoid specifications mentioning indices because this implies the use of arrays.

2.3 Exercise

2.3.1 Specification for Sorting

Write the specification for the generic `ascending_sort` method below. The specification should include preconditions and postconditions.

```python
def ascending_sort(my_list):
    # REQUIRES/PRE:
    # EFFECTS/POST:
    ...
```

2.3.2 Specification of Binary Search

Come up with the specification for a *binary search* implementation whose header is given below. Remember for precondition you want something as *weak* as possible and for postcondition as *strong* as possible. Note that binary search returns the *location* (an non-neg integer) of the `target` value if found, and returns -1 if `target` is not found.

```python
def binary_search(arr: List[int], target: int) -> int:
    """
    PRE/REQUIRES:
    POST/EFFECTS:
    """
    ...
```

2.3.3 Loan Calculator

Consider a function that calculates the number of months needed to pay off a loan of a given size at a fixed *annual* interest rate and a fixed *monthly* payment. For example, a $100,000 loan at an 8% annual rate would take 166 months to discharge at a monthly payment of $1,000, and 141 months to discharge at a monthly payment of $1,100. (In both cases, the final payment is smaller than the others; we round 165.34 up to 166 and 140.20 up to 141.) Continuing the example, the loan would never be paid off at a monthly payment of $100, since the principal would grow rather than shrink.

- Implementation satisfying given specification: Define a Python class called ‘Loan‘. In that class, write a function that satisfies the following specification:

  ```python
  class Loan:
      @staticmethod
      def months(principal: int, rate: float, payment: int) -> int:
          """
          Calculate the number of months required to pay off a loan.

          param principal: Amount of the initial principal (in dollars)
          param rate: Annual interest rate (e.g., 0.08 for 8%)
          param payment: Amount of the monthly payment (in dollars)
  ```

```
        Requires/Pre: principal, rate, and payment all positive and
        payment is sufficiently large to drive the principal to zero.
        Effects/Post: return the number of months required to pay off the principal
        """
```

- Note that the precondition is quite strong, which makes implementing the method easy. The key step in your calculation is to change the principal on each iteration with the following formula (which amounts to monthly compounding):

```
new_principal = old_principal * (1 + monthly_interest_rate) - payment
```

- To make sure you understand the point about preconditions, your code is required to be *minimal*. Specifically, if it is possible to delete parts of your implementation and still have it satisfy the requirements, you'll earn less than full credit.

- *Total* specification: Now change the specification to *total* in which the postcondition handles violations of the preconditions using *exceptions*. In addition, provide a new implementation `month` that satisfies the new specification.

2.3.4 Partial and Total Functions

1. Write the *partial* specifications for the below two functions.
2. Modify the specifications to make the functions *total*.
3. Modify the implementations of the two functions to satisfy the total specification.

Recall that specifications do not deal with types (which are taken care by the function signature and enforced by the type system of compiler/interpreter). In other words, you do not need to worry about types here and can assume conditions about types are satisfied.

```
def divide(a:float, b:float) -> float:
    """
    PRE:
    POST:
    """
    return a / b

def get_average(numbers: list[float]) -> float:
    """
    PRE:
    POST:
    """
    total = sum(numbers)
    return divide(total, len(numbers))
```

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.

3.1 Specifications of an ADT

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

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.

```python
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

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. The overview can also indicate whether the objects of this type are *mutable* (their state can change) or *immutable*. 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 the specification for an `IntSet` ADT, which represents unbounded set of integers. `IntSet` 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. `IntSet` is also mutable, as it allows elements to be added or removed. *mutator* `insert` and `remmove` are mutator methods and have a `MODIFIES` clause. In contrast, `is_in`, `size`, and `choose` are *observer* methods that do not modify the object.

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. We could use other data structures, such as a `set` or `dict`, as the rep, but a list is a simple choice for demonstration.

To aid understanding and reasoning of the rep of an ADT, we use two key concepts: *representation invariant* and *abstraction function*.

3.2.1 Representation Invariant (Rep-Inv)

Because the rep might not be necessarily related to the ADT itself (e.g., the list has different properties compared to a set), we need to ensure that our use of the rep is consistent with the ADT's behavior. To do this, we use *representation invariant* (**rep-inv**) to specify the constraints for the rep of the ADT to capture its behavior.

For example, the rep-inv for a stack is that the last element added is the first to be removed and the rep-inv for a binary search tree is that the left child is less than the parent, and the right child is greater. The rep-inv for our `IntSet` ADT in Fig. 3.2 is that all elements in the list are unique.

```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) -> None:
        """
        MODIFIES: self
        EFFECTS: Adds x to the elements of this set if not already present.
        """
        if not self.is_in(x): self.els.append(x)

    def remove(self, x: int) -> int:
        """
        MODIFIES: self
        EFFECTS: Removes x from this set if it exists. Also returns
        the index of x in the list.
        """
        i = self.find_idx(x)
        if i != -1:
            # Remove the element at index i
            self.els = self.els[:i] + self.els[i+1:]
        return i

    def is_in(self, x: int) -> (bool, int):
        """
        EFFECTS: If x is in this set, return True. Otherwise False.
        """
        return True if find_index(x) != -1 else False

    def find_idx(self, x:int)->int:
        """
        EFFECTS: If x is in this set, return its index. Otherwise 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

    def __str__(self) -> str:
        """
        Abstract function (AF) that returns a string representation of this set.
        EFFECTS: Returns a string representation of this set.
        """
        return str(self.els)
```

Fig. 3.2: The IntSet ADT

```
# Rep-inv:
# els is not null, only contains integers and has no duplicates.
```

The rep-inv must be preserved by all methods (more precisely, *mutator* methods). It must hold true before and after the method is called. The rep-inv might be violated temporarily during the method execution, but it must be restored before the method returns. For `IntSet` Notice that the mutator `insert` method ensures that the element is not already in the list before adding it.

The rep-inv is decided by the designer and specified in the ADT documentation as part of the specification (just like pre/post conditions) so that it is ensured at the end of each method (like the postcondition). Moreover, because rep-inv is so important, it is not only documented in comments but also checked at runtime. This is done by invoking a `repOK`, discussed later, method at the start and end of each method.

3.2.2 Abstraction Function (AF)

It is difficult to understand the ADT by looking at the rep directly. For example, we might not be able to visualize or reason about a binary tree or a graph ADT when using list as the rep. To aid understanding, *abstraction function* (**AF**) provides a mapping between the rep and the ADT. Specifically, the AF maps from a *concrete state* (i.e., the `else els`) to an *abstract state* (i.e., the set). AF is also a *many-to-one* mapping, as multiple concrete states can map to the same abstract state, e.g., the list `[1, 2, 3]` and `[3, 2, 1]` both map to the same set `{1, 2, 3}`.

Just as with rep-inv, the AF is documented in the class specification. Modern OOP languages often provide methods implementing the AF, in particular developer overrides the `__str__` method in Python and `toString` in Java to return a string representation of the object. For example, the `__str__` method in Fig. 3.2 returns a string representation of the set.

3.2.3 In-Class Excercise: Stack

In this exercise, you will implement a `Stack` ADT. A stack is a common data structure that follows the Last-In-First-Out (LIFO) principle. You will:

1. Choose a Representation (rep) for the stack.
2. Define a Representation invariant (rep-inv)
3. Write a `repOK` method
4. Provide the specifications of basic stack operations (`push`, `pop`, `is_empty`) and implement these methods accordingly.
5. Define an Abstraction Function (AF)

6. Implement `__str__()` to return a string representation of the stack based on the AF

3.3 Mutability vs. Immutability

An ADT can be either mutable or immutable, depending on whether their objects' values can change over time. An ADT should be immutable if the objects it models naturally have unchanging values, such as mathematical objects like integers, polynomials (Polys), or complex numbers. On the other hand, an ADT should be mutable if it models real-world entities that undergo changes, such as an automobile in a simulation, which might be running or stopped, or contain passengers, or if the ADT models data storage, like arrays or sets.

Immutability is beneficial because it offers greater safety and allows sharing of subparts without the risk of unexpected changes. Moreover, immutability can simplify the design by ensuring the object's state is fixed once created. However, immutable objects can be less efficient, as creating a new object for each change can be costly in terms of memory and time.

Converting from mutable to immutable Given a mutable ADT, it is possible to convert it to an immutable one by ensuring that the rep is not modified by any method. This can be achieved by making the rep private and only allowing read-only access to it. In Python, this can be done by using the `@property` decorator to create read-only properties. For example, the `els` list in Fig. 3.2 can be made read-only by defining a property method `elements` that returns a copy of the list.

```
class IntSet:
    def __init__(self):
        self.__els = []   # Private rep
    @property
    def self.els(self):
        return self.__els
```

Moreover, we need to convert mutator methods into observer methods, which make a copy of the rep, modify it, and return the modified rep object.

```
def insert_immutable(self, x: int) -> IntSet:
    new_set = self.els.copy()
    if not self.is_in(x):
        new_set = new_set.append(x)
    return new_set
```

If the mutator returns a value v, then our new method returns a tuple consisting of (i) the new rep object and the return the value v.

```
def remove_immutable(self, x: int): -> (IntSet, int):
    i = self.find_idx(x)
    new_set = self.els.copy()
    if i != -1:
        # Remove the element at index i
        new_set = self.els[:i] + self.els[i+1:]
    return (new_set, i)
```

If you do not want to return multiple values (e.g., like in Java), then you can create two methods, one for returning the value and the other for returning the new rep object. For example, a mutator `pop` method of a `Stack` would result into two methods: `pop2` returns the top element and `pop3` returns the new stack with the top element removed.

Finally, it is important that while it is possible to convert a mutable ADT to an immutable one as shown, mutability or immutability should be the property of the ADT type itself, not its implementation. Thus, it should be decided at the design stage and documented in the ADT specification.

3.3.1 In-class Exercise: Immutable Queue

Consider the mutable `Queue` implementation in Fig. 3.3. Rewrite `Queue` to be *immutable*. Keep the representation variables `elements` and `size`.

3.4 Exercise

3.4.1 Polynomial ADT

Use the Poly ADT in Fig. 3.4 to answer the following question. Use the `Stack` ADT in Fig. B.1 as an example.

1. Write an Overview that describes what `Poly` does. You must provide some examples to demonstrate (e.g., Poly(2,3) means what?).
2. Provide the specifications for all methods in the ADT.
3. Write the **rep** used in this code. Describe how this rep represents `Poly2`.
4. Provide the **rep-inv** for the `Poly2` ADT. Note, this would be the constraints over the rep variable(s).
5. Write a RepOK method that checks the rep-inv.
6. Describe the AF in this code. Use `__str__` to help.

```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

```python
class Poly:
    def __init__(self, c=0, n=0):
        if n < 0:
            raise ValueError("Poly(int, int) constructor: n must be >= 0")
        self.trms = {}
        if c != 0:
            self.trms[n] = c

    def degree(self):
        if len(self.trms) > 0:
            return next(reversed(self.trms.keys()))
        return 0

    def coeff(self, d):
        if d < 0:
            raise ValueError("Poly.coeff: d must be >= 0")
        return self.trms.get(d, 0)

    def sub(self, q):
        if q is None:
            raise ValueError("Poly.sub: q is None")
        return self.add(q.minus())

    def minus(self):
        result = Poly()
        for n, c in self.trms.items():
            result.trms[n] = -c
        return result

    def add(self, q):
        if q is None:
            raise ValueError("Poly.add: q is None")

        non_zero = set(self.trms.keys()).union(q.trms.keys())
        result = Poly()
        for n in non_zero:
            new_coeff = self.coeff(n) + q.coeff(n)
            if new_coeff != 0:
                result.trms[n] = new_coeff
        return result

    def mul(self, q):
        if q is None:
            raise ValueError("Poly.mul: q is None")

        result = Poly()
        for n1, c1 in self.trms.items():
            for n2, c2 in q.trms.items():
                result = result.add(Poly(c1 * c2, n1 + n2))
        return result

    def __str__(self):
        r = "Poly:"
        if len(self.trms) == 0:
            r += " 0"
        for n, c in self.trms.items():
            if c < 0:
                r += f" - {-c}x^{n}"
            else:
                r += f" + {c}x^{n}"
        return r
```

Fig. 3.4: Polynomial ADT

Appendix A

Miscs

Appendix B

Answers

B.1 ADT

B.1.1 Stack ADT

```
class Stack:
"""
OVERVIEW: Stack is a mutable ADT that represents a collection of elements in LIFO.
AF(c) = the sequence of elements in the stack in sorted order from bottom to top.
rep-inv:
    1. elements is a list (could be empty list, which represents and empty stack).
    2. The top of the stack is always the last element in the list.
"""

def __init__(self):
    """
    Constructor
    EFFECTS: Initializes an empty stack.
    MODIFIES: self
    """
    self.elements = []

def repOK(self):
    """
    EFFECTS: Returns True if the rep-invariant holds, otherwise False.
    The invariant checks:
    1. elements is a list.
    2. If the stack is non-empty, the top of the stack is the last element in the list.
    """
    # Check that elements is a list
    if not isinstance(self.elements, list):
        return False

    # If the stack is not empty, ensure that the top is the last element in the list.
    # This is implicitly guaranteed by the use of `list.append` for push and `list.pop` for
    # so no further explicit check is needed for the "top as last element."
    return True

def push(self, value):
    """
    MODIFIES: self
    EFFECTS: Adds value to the top of the stack.
    """
    self.elements.append(value)

def pop(self):
    """
    MODIFIES: self
    EFFECTS: Removes and returns the top element from the stack.
    Raises an exception if the stack is empty.
    """
    if self.is_empty():
        raise Exception("Stack is empty")
    return self.elements.pop()

def is_empty(self):
    """
    EFFECTS: Returns True if the stack is empty, otherwise False.
    """
    return len(self.elements) == 0

def __str__(self):
    """
    EFFECTS: Returns a string representation of the stack,
             showing the elements from bottom to top.
    """
    # The abstraction function maps the list of elements to a stack view
    return f"Stack({self.elements})"
```

Fig. B.1: Stack ADT