Tipos de datos abstractos

Jorge Mario Londoño Peláez & Varias AI

January 31, 2025

1 Concepto de Tipos de Datos Abstractos

En esta sección, exploraremos los conceptos fundamentales relacionados con los Tipo de Dato Abstracto (TDA) (Abstract Data Type (ADT)). Es crucial entender cómo estos se relacionan con las representaciones de datos a bajo nivel y los tipos de datos primitivos.

- **Representaciones de bajo nivel**: En el nivel más básico, los datos se representan como secuencias de bits (datos binarios). Estas representaciones son directamente interpretadas por el hardware de la computadora.
- **Tipos de datos primitivos (o concretos)**: Son los tipos de datos básicos que los lenguajes de programación ofrecen directamente, y que tienen un soporte nativo en el hardware. Ejemplos comunes incluyen `int` (enteros), `float` (números de punto flotante), `char` (caracteres) y `bool` (booleanos).
- **Tipos de datos abstractos**: Los ADT son modelos de datos que se definen en términos de las operaciones que se pueden realizar sobre ellos, en lugar de cómo se almacenan en la memoria. Un ADT encapsula la representación de los datos y las operaciones permitidas, ofreciendo una interfaz clara y bien definida. Los ADT se construyen utilizando tipos de datos primitivos u otros ADT más simples, permitiendo modelar los datos de una manera más cercana a la lógica del problema que se está resolviendo.

2 Beneficios de los ADT

Los ADT ofrecen varios beneficios importantes en el desarrollo de software, que mejoran la calidad, la mantenibilidad y la reutilización del código:

- **Alto nivel de abstracción**: Los ADT permiten modelar los datos de una manera que se asemeja mucho a la forma en que se conciben en el problema real. Esto facilita la comprensión del problema y el diseño de la solución, ya que se trabaja con conceptos más cercanos al dominio del problema.
- **Modularidad y Reutilización**: Los ADT fomentan la creación de componentes de software reutilizables. Una vez que se define un ADT, se puede utilizar en diferentes partes de un programa o incluso en diferentes proyectos, lo que reduce la duplicación de código y acelera el desarrollo.

- **Encapsulación y Ocultamiento de la Información**: Los ADT encapsulan la representación interna de los datos y exponen una interfaz bien definida para interactuar con ellos. Esto oculta los detalles de implementación y protege los datos de manipulaciones incorrectas, lo que ayuda a mantener la consistencia y la integridad de los datos.
- **Facilidad de Mantenimiento**: Al separar la interfaz de la implementación, los ADT facilitan el mantenimiento y la evolución del software. Los cambios en la implementación de un ADT no afectan al resto del código, siempre y cuando la interfaz se mantenga igual.

3 Implementación de TDA en la Programación Orientada a Objetos (POO)

En la Programación Orientada a Objetos (POO) (Object Oriented Programming (OOP)), los ADT se implementan utilizando clases y objetos. Esta sección detalla cómo se lleva a cabo esta implementación:

- **Representación del Estado del TDA**: El estado de un ADT se representa mediante las variables de instancia (también conocidas como atributos o campos) dentro de una clase. Estas variables almacenan los datos que definen el estado específico de cada objeto del ADT.
- **Interfaz del ADT (Application Programming Interface (API))**: La interfaz de un ADT se define a través de los métodos públicos de la clase. Estos métodos son las operaciones que se pueden realizar sobre los objetos del ADT, y permiten interactuar con los datos de manera controlada y segura.
- **Clases como Implementación de TDA**: En OOP, un ADT se implementa como una clase. La estructura del ADT se define por los tipos de datos de las variables de instancia, mientras que el comportamiento se define por los métodos de la clase.
- **Objetos como Instancias del ADT**: Los datos del tipo del ADT se representan como instancias de la clase, es decir, objetos. Cada objeto es una realización concreta del ADT, con su propio estado específico.

4 Ejemplos básicos de ADT

A continuación, se presentan algunos ejemplos básicos de ADT implementados en Python, mostrando cómo se definen las clases y sus métodos para encapsular los datos y las operaciones:

- **ADT Fecha**: Representa una fecha con día, mes y año.

```python
class Fecha:
    def __init__(self, dia, mes, anno):
        self.dia = dia
        self.mes = mes
        self.anno = anno

    def __str__(self):
        return f"{self.dia}/{self.mes}/{self.anno}"
```

- **ADT Contador**: Representa un contador con un nombre y un valor entero.

```python
class Contador:
    def __init__(self, nombre, valor=0):
        self.nombre = nombre
        self.valor = valor

    def incrementar(self):
        self.valor += 1

    def __str__(self):
        return f"{self.nombre}: {self.valor}"
```

- **ADT Punto**: Representa un punto en un plano cartesiano con coordenadas x e y.

```python
class Punto:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def mover(self, dx, dy):
        self.x += dx
        self.y += dy

    def __str__(self):
        return f"({self.x}, {self.y})"
```

5 Prueba unitaria de las operaciones de un ADT

Las pruebas unitarias son una parte fundamental del desarrollo de software, ya que permiten verificar que cada componente del programa (en este caso, las operaciones de un ADT) funciona correctamente de forma aislada. En esta sección, explicaremos cómo implementar pruebas unitarias sencillas utilizando sentencias `assert` en Python, sin necesidad de librerías especializadas.

Las pruebas unitarias se basan en la idea de crear casos de prueba que ejerciten diferentes caminos de ejecución del código y verificar que el comportamiento observado coincide con el comportamiento esperado. Cada caso de prueba debe probar una funcionalidad específica del ADT.

A continuación, se muestra un ejemplo de cómo implementar una prueba unitaria para el método `mover` del ADT `Punto`:

```python
# Prueba unitaria del metodo mover
p = Punto(1, 2)
assert str(p) == "(1, 2)" # verificar el estado inicial
p.mover(3, 4)
assert str(p) == "(4, 6)" # verificar el estado despues de mover
print("Prueba unitaria del metodo mover: OK")
```

En este ejemplo, primero se crea un objeto `Punto` con coordenadas iniciales (1, 2). Luego, se utiliza una sentencia `assert` para verificar que la representación en cadena del punto es la esperada. A continuación, se llama al método `mover` para desplazar el punto y se utiliza otra sentencia `assert` para verificar que las coordenadas del punto se han actualizado correctamente. Si alguna de las aserciones falla, el programa se detendrá con un error. Si todas las aserciones pasan, se imprimirá un mensaje indicando que la prueba unitaria ha sido exitosa.

Este enfoque sencillo permite verificar el comportamiento de las operaciones de un ADT de manera efectiva y sin necesidad de herramientas complejas. Es importante crear suficientes casos de prueba para cubrir todos los posibles escenarios y asegurar la calidad del código.

5.1 Uso de la herencia

La herencia es un concepto fundamental en la POO que permite crear nuevas clases basadas en clases existentes. Esto promueve la reutilización de código y la creación de jerarquías de clases. En el contexto de los ADT, la herencia se puede utilizar de dos maneras principales: herencia de interfaces y herencia de implementación.

5.1.1 Tipos de herencia

En la POO, la herencia se manifiesta principalmente de dos formas:

- **Herencia de interfaces:** Se enfoca en heredar la estructura (métodos) de una clase abstracta o interfaz, sin heredar la implementación.
- **Herencia de implementación:** Se hereda tanto la estructura como la implementación de una clase base.

5.1.2 La interface como la definición del API del ADT

Una interfaz define el API de un ADT, especificando las operaciones que se pueden realizar sobre los datos, sin revelar cómo se implementan estas operaciones. En esencia, una interfaz es un contrato que las clases deben cumplir.

El tipo de una interfaz se puede utilizar como un tipo de datos, lo que permite el polimorfismo. Esto significa que diferentes clases pueden implementar la misma interfaz, y el código cliente puede interactuar con estas clases a través de la interfaz, sin conocer la implementación específica. Esto facilita la creación de sistemas flexibles y extensibles.

5.2 La herencia de implementacion

En algunos casos, es útil heredar la implementación de una clase base, además de su interfaz. Esto permite reutilizar el código de la clase base y evitar la duplicación de código.

En Java, todas las clases heredan implícitamente de la clase `Object`. Esta clase proporciona métodos como `equals`, `hashCode` y `toString`, que son fundamentales para el comportamiento de los ADT.

- `equals`: Permite comparar dos objetos para determinar si son iguales. Es importante sobrescribir este método para que la comparación se base en el contenido del objeto (el valor del ADT), en lugar de comparar sus referencias.
- `toString`: Devuelve una representación en cadena del objeto. Es útil para la depuración y la visualización de los datos. Idealmente todo ADT debe proporcionar la forma de representar textualmente el valor del ADT para por ejemplo visualizarlo fácilmente.
- `hashCode`: Devuelve un código hash para el objeto. Este método es importante para el uso de ADT en estructuras de datos como las tablas asociativas.

5.2.1 Ejemplos

Ejemplo 1: ADT Punto2D Este ejemplo define un ADT `Punto2D` que representa un punto en un plano bidimensional. La interfaz `Punto2D` define los métodos `getX()`, `getY()` y `distancia()`. Se proporcionan dos implementaciones: `Punto2DCartesiano` y `Punto2DPolar`.

```
interface Punto2D {
    double getX();
    double getY();
    double distancia(Punto2D otro);
}
```

Listing 1: Interfaz Punto2D

```
class Punto2DCartesiano implements Punto2D {
    private double x;
    private double y;

    public Punto2DCartesiano(double x, double y) {
        this.x = x;
        this.y = y;
    }

    @Override
    public double getX() {
        return x;
    }

    @Override
    public double getY() {
        return y;
    }

    @Override
    public double distancia(Punto2D otro) {
        double dx = this.getX() - otro.getX();
        double dy = this.getY() - otro.getY();
        return Math.sqrt(dx * dx + dy * dy);
    }
}
```

Listing 2: Implementación cartesiana de Punto2D

```
class Punto2DPolar implements Punto2D {
    private double radio;
    private double angulo;

    public Punto2DPolar(double radio, double angulo) {
        this.radio = radio;
        this.angulo = angulo;
    }


    @Override
    public double getX() {
        return radio * Math.cos(angulo);
    }

    @Override
    public double getY() {
        return radio * Math.sin(angulo);
    }

   @Override
    public double distancia(Punto2D otro) {
```

```
        double dx = this.getX() - otro.getX();
        double dy = this.getY() - otro.getY();
        return Math.sqrt(dx * dx + dy * dy);
    }
}
```

Listing 3: Implementación polar de Punto2D

Ejemplo 2: Implementación de `equals` Este ejemplo muestra cómo implementar el método `equals` para la clase `Punto2D`. La implementación debe comparar los puntos con una tolerancia para tener en cuenta la precisión limitada de los números de punto flotante.

```
    @Override
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null || !(obj instanceof Punto2D)) return false;
        Punto2D other = (Punto2D) obj;
        return distancia(other) < TOL;
    }

    static final double TOL = 1E-12;
```

Listing 4: Implementación de equals en Punto2D

Ejemplo 3: Implementación de `toString` Este ejemplo muestra cómo implementar el método `toString` para las clases `Punto2DCartesiano` y `Punto2DPolar`.

```
    @Override
    public String toString() {
        return "(" + x + ", " + y + ")";
    }
```

Listing 5: Implementación de toString en Punto2DCartesiano

```
    @Override
    public String toString() {
       return "(" + radio + "@" + angulo + ")";
    }
```

Listing 6: Implementación de toString en Punto2DPolar

Ejemplo 4: Prueba unitaria Este ejemplo muestra cómo realizar una prueba unitaria para verificar el correcto funcionamiento del método `equals`.

```
public static void main(String[] args) {
    Punto2D p1 = new Punto2DCartesiano(1, 1);
    Punto2D p2 = new Punto2DCartesiano(1.00001, 1.00001);
    Punto2D p3 = new Punto2DCartesiano(2, 2);
    Punto2D p4 = new Punto2DPolar(Math.sqrt(2), Math.PI / 4);


    assert p1.equals(p2) : "p1 debe ser igual a p2";
    assert !p1.equals(p3) : "p1 no debe ser igual a p3";
    assert p1.equals(p4) : "p1 debe ser igual a p4";

    System.out.println("Pruebas unitarias pasaron");
}
```

Listing 7: Prueba unitaria de equals

5.2.2 Métodos especiales en Python

Los métodos especiales, también conocidos como métodos mágicos o dunder methods (por sus nombres que comienzan y terminan con doble guión bajo), son métodos que permiten personalizar el comportamiento de los objetos en Python. Estos métodos son invocados automáticamente por el intérprete en ciertas situaciones, como cuando se crea un objeto, se imprime, se compara, etc.

En la implementación de TDAs, los métodos especiales son muy útiles para definir cómo se comportan los objetos de nuestro TDA en diferentes contextos.

5.3 Métodos Especiales Comunes

- `__init__`: El constructor de la clase. Se llama cuando se crea un nuevo objeto de la clase.
- `__str__`: Define cómo se representa un objeto como una cadena de texto. Se llama cuando se usa la función `str()` o `print()` sobre el objeto.
- `__eq__`: Define cómo se comparan dos objetos para determinar si son iguales. Se llama cuando se usa el operador `==`.
- `__hash__`: Define el valor hash de un objeto. Es necesario para usar objetos como claves en diccionarios o en conjuntos. Cuando se sobre-escribe el método hash, es importante mantener la consistencia con el `equals`: Si dos objetos son iguales, sus hashes deben ser los mismos.

5.4 Ejemplo en Python

A continuación, se muestra un ejemplo de cómo usar estos métodos especiales en el ADT `Punto`:

```
class Punto:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __str__(self):
        return f"({self.x}, {self.y})"

    def __eq__(self, otro_punto):
        if otro_punto is None:
            return False
        if isinstance(otro_punto, Punto):
            return self.x == otro_punto.x and self.y == otro_punto.y
        return False

    def __hash__(self):
        return hash((self.x, self.y))
```

En este ejemplo, hemos definido cómo se crea un `Punto`, cómo se representa como una cadena, cómo se compara con otro `Punto` y cómo se calcula su valor hash. Esto nos permite usar los objetos `Punto` de manera más natural y consistente.