2. Abstracción

Programación II

Grado en Inteligencia Artificial

M. Alonso, M. Cabrero y E. Hernández

fic facultade de informática da coruña

UNIVERSIDADE DA CORUÑA

Contenidos

Definición

¿Qué es la abstracción según la RAE?

abstraer

Del lat. *abstrahĕre* 'arrastrar lejos', 'apartar, separar'.

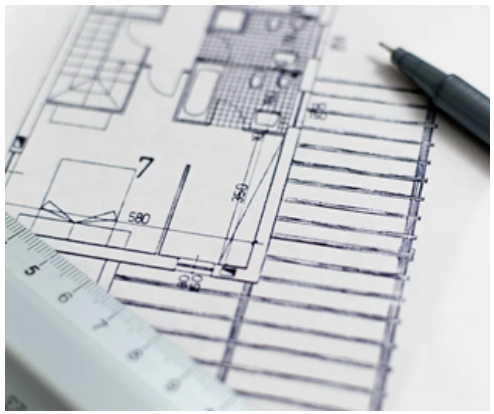
1. Separar por medio de una operación intelectual un rasgo o una cualidad de algo para analizarlos aisladamente o considerarlos en su pura esencia o noción. *Pensar es olvidar diferencias, es generalizar, abstraer.*

2. Hacer caso omiso de algo, o dejarlo a un lado. *Centremos la atención en lo esencial abstrayendo de consideraciones marginales.*

La abstracción es humana

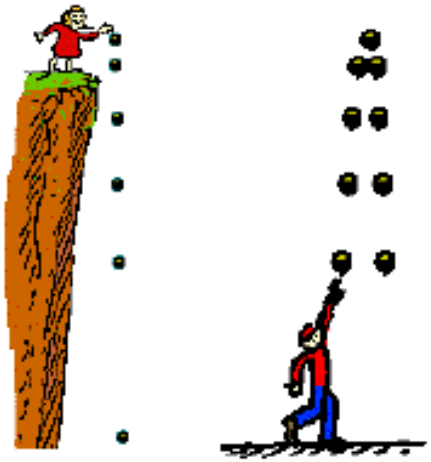
- Los seres humanos aplicamos este concepto para enfrentamos con problemas del mundo real.
- Limitamos la cantidad de información—en un principio—a la estrictamente relevante.
- Nos quedamos con una parte de la realidad, una simplificación, un subconjunto de propiedades y características de los objetos reales.

Ejemplos de abstracción



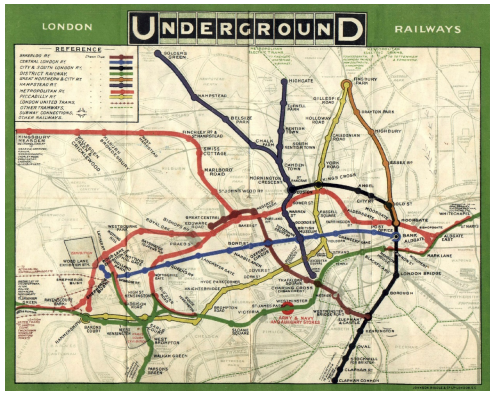
Un plano de una casa es la abstracción de una casa física que nos permite centrar la atención en **elementos estructurales** sin preocuparnos de **detalles accidentales** (p.ej. color de la cocina) que afectan al aspecto final pero no a las relaciones entre las distintas partes de la casa (paredes, electricidad, calefacción, ventanas, ...).

Ejemplos de abstracción

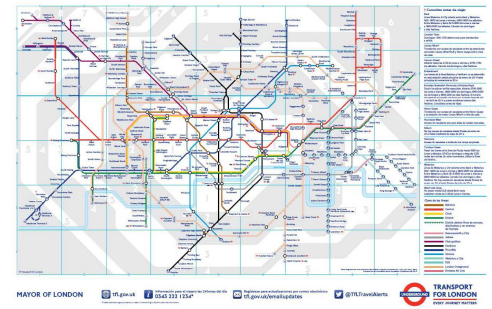


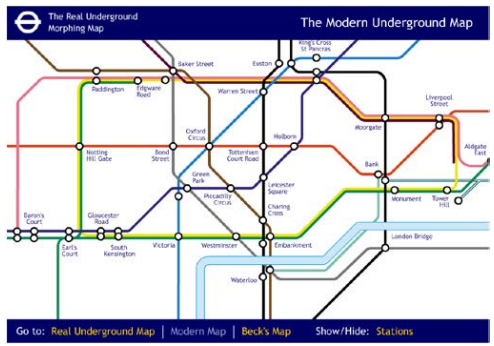
Cuando se estudia el efecto de la gravedad sobre un objeto que cae se ignoran ciertos detalles accidentales como el color o el sabor del objeto (poco importa de qué variedad era la manzana que golpeó a Newton en la cabeza).

Ejemplos de abstracción



Mapa del metro





El plano del metro de Londres fue publicado en 1908. Seguía fielmente la geografía pues se respetaron todas las curvas y la distancia relativa entre estaciones. Con el tiempo, del mapa fueron eliminados ciertos detalles relacionados con la superficie, la distancia o la orientación de las líneas, en favor de una representación más abstracta que sólo mostrara la conectividad entre estaciones.

La abstracción en programación

- En general, el problema fundamental de la programación es gestionar la complejidad asociada a cualquier problema a resolver por el ordenador.
- Para ello son necesarios modos de limitar el número de detalles a considerar.
- El proceso de ignorar algunos detalles mientras se concentra en aquellos relevantes para el problema es la **abstracción**.
- El desarrollo de software efectivo es un ejercicio de construcción de las abstracciones apropiadas.

Tipos de abstracción

- Abstracción **funcional**: Uso de un método o función sabiendo qué hace pero ignorando cómo lo lleva a cabo.
  - P.ej., para obtener la raíz cuadrada de un número usamos la función correspondiente de Python `sqrt`.
- Abstracción **de datos**: Separación entre las propiedades de un tipo de dato (valores que puede tomar y operaciones) de la implementación del tipo de dato.
  - P.ej., para utilizar los *strings* de Python desconocemos cómo se representan internamente o cómo están implementadas las operaciones.

Abstracción funcional

Abstracción funcional (o procedural)

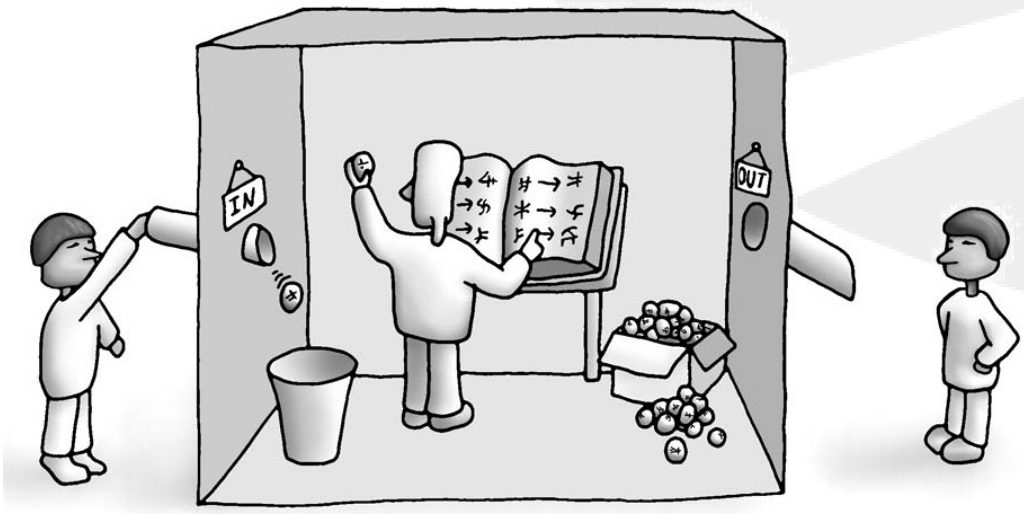
- El código de la función implementa el servicio y el programa que la usa ejerce como *cliente*.
- Entre el implementador y el cliente existe una **interfaz**, una barrera de abstracción que separa dos vistas:
  - La del cliente que usa la función sin conocimiento de cómo trabaja (i.e. no sabe *qué hay dentro de la caja*).
  - La del implementador de la función que no se preocupa en cómo será usada sino de ofrecer una descripción cuidadosa de lo que hace (i.e. la especificación).
- La interfaz se formaliza en *una especie de contrato*: la especificación.

La especificación como contrato

- Incluirá el nombre de la función, una descripción precisa de su cometido, los tipos de los parámetros que necesita y el tipo de retorno (si hay).
- Podrá ser informal pero lo suficientemente completa y no ambigua para que cliente e implementador no tengan dudas trabajando sólo con ella.
- Si se mantiene la especificación como una barrera firme, tanto el código del cliente como la implementación pueden cambiar sin afectar al programa. Se conoce como *independencia de la implementación*.

La especificación como contrato

- El experimento de la habitación china: por un extremo el individuo A (programa) introduce (input) textos escritos en chino y por el otro el individuo B (programa) recibe (output) la traducción al inglés.
- La implementación no es más que una persona con un diccionario chino-inglés.
- Si se mantiene la especificación podemos cambiar de editorial (implementación) sin afectar a la salida.



La especificación precisa, por favor

- Puede surgir un desastre real cuando las especificaciones no están claramente explicadas y cumplidas.
- Un ejemplo: la misión Mars Climate Orbiter de la NASA en 1999 se estrelló debido a un error en las mediciones: se estaba dando información a un módulo en unidades del sistema anglosajón, pero se esperaba en unidades métricas. Se perdieron 125 millones de dólares.

Precondiciones y Poscondiciones

- La especificación de una función/método se escribe, de forma breve y precisa, en términos de *precondiciones* y *poscondiciones*.
- Una **precondición** es una declaración de lo que se supone que es cierto sobre el estado del cálculo en el momento en que se llama a la función o método.
  - Establece cualquier suposición que haga la implementación (especialmente sobre parámetros).
  - Tiene que ver con el estado del objeto sobre el que se ejecuta el método. P. ej. antes de borrar un item de una colección, éste debe pertenecer a ella.
- Una **poscondición** es una declaración sobre lo que es verdadero después de que la función/método finalice.
  - Normalmente acompañan a métodos que modifican el estado interno del objeto.

Precondiciones y Poscondiciones

```
def sqrt(x):
        """Calcula la raíz cuadrada de x

        pre: x es un entero o un flotante y x>=0
        pos: devuelve la raíz cuadrada no negativa de x"""
```

- La precondición y la poscondición juntas describen la función como un **contrato**: si el cliente cumple la condición previa a la llamada a la función, entonces la implementación asegura la condición posterior a la ejecución.

Especificaciones en el código. ¿Cómo?

- Comentarios regulares (#): Reservados a los implementadores.
- *docstrings*:
  - Expresiones de cadena en la parte superior de un módulo o inmediatamente después de un encabezado de función o de clase.
  - Destinados a los programadores del cliente.
  - PyDoc genera documentación automáticamente.
  - Uso con el sistema de ayuda interno de Python.

```
>>> help("sqrt")
Help on function sqrt in module __main__:

sqrt(x)
    Calcula la raíz cuadrada de x

    pre: x es un entero o un flotante y x>=0
    pos: devuelve la raíz cuadrada no negativa de x
```

¿Y si no se cumplen las especificaciones?

- Ignorarlo: bloqueo o (*abort*) del programa, resultados imprevistos
- Detectar el error:
  - *Imprimir un mensaje*: el cliente no sabe qué ha pasado.
  - *Señalizar*: devolver un resultado imposible (`sqrt` devuelve `-1`), variable global con código de error...⇒ estructuras de decisión complejas para gestión del error.
  - *Excepciones*: no se comprueba explícitamente; cuando se produce, se "llama" al código del cliente que manejará la situación.
  - *Aserciones*: verificar antes y lanzar una excepción (programación defensiva)⇒ sobrecarga de ciclos CPU.

Ejemplo de excepción

```
def sqrt(x):
        if type(x) not in (type(1), type(1.0)):
                raise TypeError('number expected')
        if x < 0:
                raise ValueError('math domain error')

        # Calcula raíz cuadrada aquí
        ...
```

```
>>> try:
...     sqrt (-1)
... except ValueError:
...     print('Oh, lo siento.')
...
Oh, lo siento.
>>>
```

```
>>> try:
...     sqrt ("hola")
... except TypeError:
...     print('No es un número.')
...
No es un número.
```

Ejemplo de aserción

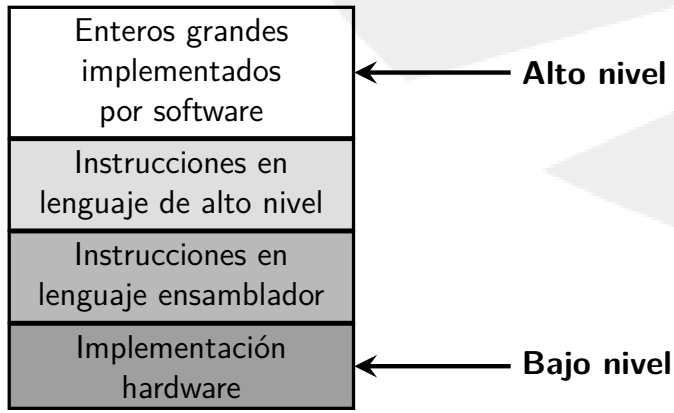
```
def sqrt(x):
        assert type(x) in (type(1), type(1.0)) and (x >= 0)
        ...
```

```
>>> sqrt(-1)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 2, in sqrt
AssertionError
```

Abstracción de datos

Abstracción de datos

- Recordemos que es la separación entre las propiedades de un tipo de datos (valores que puede tomar y operaciones) de la implementación del tipo de dato
- ¿Cómo representar valores enteros en un ordenador y hacer operaciones sobre esos valores?
- Niveles de abstracción para representar el tipo de dato `int` desde el nivel más bajo (binario) hasta lenguajes de alto nivel.



Abstracción de datos

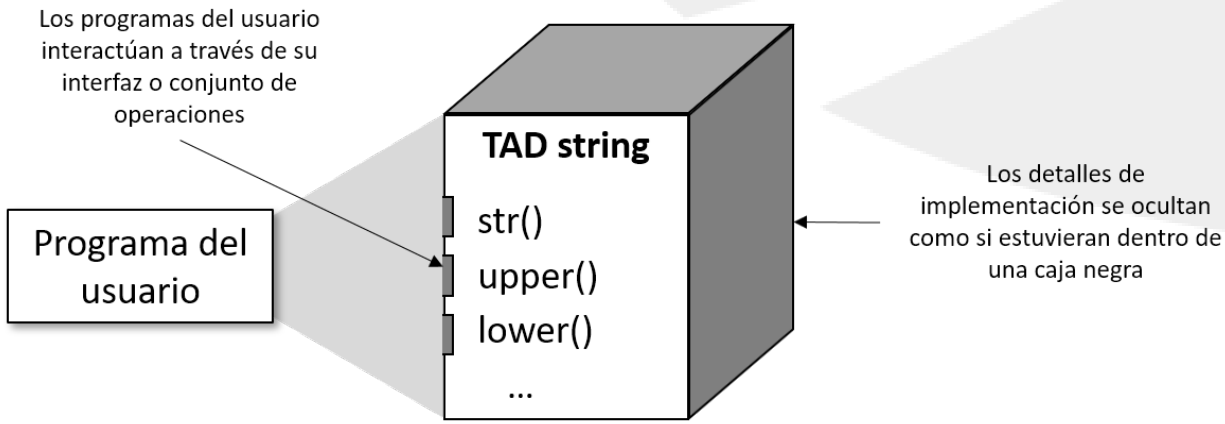
- El tipo predefinido `float` representa decimales en el rango $-1{,}79 * 10^{308}$ a $1{,}8 * 10^{308}$ y puede usarse con operaciones como la suma, resta, multiplicación, ...
- Conociendo tan sólo esto podemos escribir programas "clientes" del tipo de dato.
- Se desconoce la implementación subyacente, la representación de todos los posibles valores así como un conjunto de funciones que manipulen dicha representación.
- Los tipos predefinidos (`string`, `float`, `list`, ...) tienen una representación y un conjunto de operaciones propias.

Conclusión

Tipo de dato = conjunto de valores + operaciones

Tipo Abstracto de Datos (TAD)

- Es un tipo de datos **definido por el usuario** que "especifica" un conjunto de valores de datos y una colección de operaciones bien definidas para ejecutarse sobre dichos valores.
- Se definen independientemente de la implementación (¡céntrate en cómo usarlo en lugar de cómo realizarlo!)
- La interacción con el nuevo tipo es a través de la interfaz o conjunto de operaciones para lo que se ocultan los detalles de implementación



Tipos de operaciones de un TAD

La aproximación clásica es:

- **Generadoras**: Crean una nueva instancia del tipo (puede o no partir de otras instancias anteriores). Usándolas es posible construir todos los valores del tipo. P.ej. crear una lista o añadir elementos.
- **Modificadoras**: Igual que las generadoras pero no pertenecen al "conjunto mínimo" que permite construir los valores del tipo. P.ej. eliminar un elemento de la lista o concatenar dos listas.
- **Observadoras**: Permiten acceder a aspectos del tipo (codificados en otros tipos), pero no modifica el tipo al que observa. P.ej. número de elementos de la lista.

Tipos de operaciones de un TAD

Una aproximación más moderna es:

- **Constructoras**:
  - **Creadoras**: Crean e inicializan nuevas instancias del TAD sin usar instancias del TAD como argumentos de entrada. P.ej. crear una lista.
  - **Productoras**: Crean nuevas instancias del TAD a partir de otras instancias del TAD. P. ej. la concatenación de dos listas crea una lista nueva.
- **Mutadoras**: Modifican los contenidos de una instancia (estado de un objeto) del TAD. P. ej. añadir o borrar un elemento a la lista.

Tipos de operaciones de un TAD

- **Observadoras (o "accesoras")**: Acceden a datos contenidos en una instancia del TAD y devuelven resultados de tipos diferentes. P. ej. el tamaño de una lista devuelve un tipo `int`
- **Iteradoras**: Un tipo de observadoras que procesan de manera secuencial componentes de datos individuales de la instancia del TAD.

Ventajas de un TAD

- Permiten centrarnos en resolver el problema en lugar de atascarnos en los detalles de implementación.
- Reducen los errores lógicos causados por usar mal las estructuras de almacenamiento y los tipos de datos porque no hay acceso a la implementación.
- Facilitan el cambio de implementación del TAD sin tener que modificar el código del programa cliente que lo usa.
- Permiten que varios equipos trabajen independientemente en módulos de un proyecto complejo siempre que respeten la definición del TAD.
- Favorecen la reutilización del código.

Diseñando un TAD: Ejemplo 1

Especificación del TAD Carta: Valores

- Una carta de la baraja francesa es un par de valores, uno correspondiente al valor de la carta y otro correspondiente al palo.
- Los valores de las cartas son: 'as', 'dos', 'tres', 'cuatro', 'cinco', 'seis', 'siete', 'ocho', 'nueve', 'diez', 'jota', 'reina' y 'rey'.
- Los palos de las cartas son: 'corazones' ♡, 'diamantes' ♢, 'picas' ♠ y 'treboles' ♣

Especificación del TAD Carta
Operaciones Constructoras Creadoras

- **crear(valor, palo) $\rightarrow$ Carta**

  Objetivo: Crea una carta del valor y palo dados

  Entradas: Un valor y un palo

  Salidas: Un nueva carta

  Precondición: valor en el rango (1,13) y palo en 'cdpt'

Especificación del TAD Carta
Operaciones Observadoras

- **palo()** $\rightarrow$ `caracter`

  Objetivo Devuelve el palo de la carta
  Entradas Una carta (*self*)
  Salidas Un valor de palo
  Poscondición El valor de palo es un carácter del conjunto cdpt

- **valor()** $\rightarrow$ `entero`

  Objetivo Devuelve el valor de la carta
  Entradas Una carta (*self*)
  Salidas Un valor numérico
  Poscondición El valor de la carta está en el rango (1-13)

Especificación del TAD Carta
Operaciones Observadoras

- **nombrePalo()** $\rightarrow$ `cadena`
  - Objetivo: Devuelve el nombre del palo de la carta
  - Entradas: Una carta (*self*)
  - Salidas: Un valor de ('corazones', 'diamantes', 'picas', 'tréboles"
- **nombreValor()** $\rightarrow$ `cadena`
  - Objetivo: Devuelve el nombre del valor de la carta
  - Entradas: Una carta (*self*)
  - Salidas: Un valor de ('as', 'dos', ..., 'jota', 'reina', 'rey')
- **toString()** $\rightarrow$ `cadena`
  - Objetivo: Devuelve la representación de la carta
  - Entradas: Una carta (*self*)
  - Salidas: Una cadena con el nombre de la carta

Implementación del TAD Carta

- Decidir cómo representar en el ordenador la `Carta` y sus componentes `palo` y `número`. P.ej. usando un tipo `tupla` el As de tréboles sería `(1, 't')`
- Codificar cada operación de `Carta` como una función
- Colocar todas las operaciones en un mismo fichero (módulo)

Implementación del TAD Carta

```
# TADCarta.py implementación TAD Carta con funciones

_PALOS = 'cdpt'
_NOMBRES_PALOS = ['corazones', 'diamantes', 'picas', 'tréboles']
_VALORES = range(1,14)
_NOMBRES_VALORES = [ 'As', 'Dos', 'Tres', 'Cuatro', 'Cinco', 'Seis', \
                     'Siete', 'Ocho', 'Nueve', 'Diez', 'Jota', 'Reina', 'Rey' ]
def create(valor, palo):
    assert valor in _VALORES and palo in _PALOS
    return (valor, palo)

def valor(carta):                    def palo(carta):
    return carta[0]                      return carta[1]

def nombrePalo(carta) :
    index = _PALOS.index( palo(carta) )
    return _NOMBRES_PALOS [index]

def nombreValor(carta) :
    index = _VALORES.index( valor(carta) )
    return _NOMBRES_VALORES[index]

def toString(carta) :
    return nombreValor(carta) + ' de ' + nombrePalo (carta)
```

Implementación del TAD Carta: Comentarios

- Se usa `assert` para asegurar que se cumplen las precondiciones.
- Los nombres con mayúsculas son constantes. Comienzan con un guión bajo para que el cliente no pueda acceder a ellos cuando importe el módulo.
- Las operaciones `palo` y `valor` simplemente desempaquetan la parte apropiada de la tupla `carta`.
  - ¿No podría el cliente acceder directamente con `miCarta[0]`?
  - El cliente *podría* pero *no debería*, puesto que el TAD desacopla al cliente de la implementación.
  - Si se permite el acceso, cualquier cambio posterior en la implementación deshabilita el código del cliente.
  - Los clientes deberían usar el TAD sólo a través de las operaciones proporcionadas.

Uso del TAD Carta

```
# test_TADCarta.py
# Programa de prueba del TAD Carta
import TADCarta

def printAll():
    for p in 'cdpt':
        for v in range(1, 14):
            carta = TADCarta.create(v,p)
            print (TADCarta.toString(carta))

if __name__ == '__main__':
    printAll()
```

Salida por pantalla

```
As  de corazones
Dos  de corazones
Tres  de corazones
...
Jota  de tréboles
Reina  de tréboles
Rey  de tréboles
```

Implementación del TAD Carta con objetos

- Usando Python como lenguaje de implementación podemos implementar el TAD Carta con objetos a partir de **la misma especificación**
- Los objetos combinan datos y operaciones:
  - Las variables instancia almacenan los datos
  - Los métodos corresponden a las operaciones

Implementación del TAD con objetos

```
class Carta:
    PALOS = 'cdpt'
    NOMBRES_PALOS = ['corazones','diamantes','picas','tréboles']
    VALORES = range(1,14)
    NOMBRES_VALORES = [ ' As ' , ' Dos ' , ' Tres ' , ' Cuatro ' , \
                        ' Cinco ' , ' Seis ' ,' Siete ' , ' Ocho ' , \
                        ' Nueve ' , ' Diez ' , ' Jota ' , \
                        ' Reina ' , ' Rey ' ]
    def __init__(self, valor, palo):
        assert valor in self.VALORES and palo in self.PALOS
        self.valor_num = valor
        self.palo_char = palo

    def palo(self):
        return self.palo_char

    def valor(self):
        return self.valor_num

    def nombrePalo(self):
        return self.NOMBRES_PALOS[self.PALOS.index(self.palo_char)]

    def nombreValor(self):
        return self.NOMBRES_VALORES[self.VALORES.index(self.valor_num)]

    def __str__(self):
        return '{0} de {1}'.format(self.nombreValor(), self.nombrePalo())
```

Uso del TAD Carta con objetos

```
# test_TADClaseCarta.py
# Programa de prueba del TAD ClaseCarta
# Antes import TADCarta
from TADClaseCarta import Carta

def printAll():
    for palo in 'cdpt':
        for valor in range(1, 14):
            # Antes carta = TADCarta.create(v,p)
            carta = Carta (valor, palo)
            print (carta)

if __name__ == '__main__':
    printAll()
```

¿Y si cambiamos la representación del TAD?

- La independencia entre la especificación y la implementación de un TAD se manifiesta en que el acceso a los datos tiene lugar a partir de métodos (`palo` y `valor`)
- Es posible cambiar la representación concreta sin afectar al código del cliente.
- Representación alternativa:
  - Un entero correlativo (entre 0 y 51) y las cartas se ordenan por valor y por palo (corazones, diamantes, picas y tréboles).
  - La primera carta en el mazo sería el as de corazones y la última el rey de tréboles.

Cambiando la representación del TAD

```
class Carta:
    PALOS = 'cdpt'
    NOMBRES_PALOS = ['corazones','diamantes','picas','tréboles']
    VALORES = range(1,14)
    NOMBRES_VALORES = [ ' As ' , ' Dos ' , ' Tres ' , ' Cuatro ' , ' Cinco ' , \
                        ' Seis ' ,' Siete ' , ' Ocho ' , ' Nueve ' ,\
                        ' Diez ' , ' Jota ' , ' Reina ' , ' Rey ' ]

    def __init__(self, valor, palo):
        assert valor in self.VALORES and palo in self.PALOS
        self.num_carta = self.PALOS.index(palo)*13 + (valor - 1)

    def palo(self):
        return self.PALOS[self.num_carta // 13]

    def valor(self):
        return self.num_carta % 13

    def nombrePalo(self):
        return self.NOMBRES_PALOS[self.num_carta // 13]

    def nombreValor(self):
        return self.NOMBRES_VALORES[self.num_carta % 13]

    def __str__(self):
        return '{0} de {1}'.format(self.nombreValor(), self.nombrePalo())
```

Diseñando un TAD: Ejemplo 2

Especificación del TAD *DataSet*: Valores

- Un *dataset* es una colección de números.
- La colección no está ordenada y puede incluir elementos repetidos.

Especificación del TAD *DataSet*: Operaciones

- Se necesitan métodos que añadan elementos al dataset y que devuelvan el valor mínimo y máximo.
- Se deben incluir operaciones para calcular simples estadísticas como media y desviación estándar.

Especificación del TAD *DataSet*
Operaciones Constructoras Creadoras

- **crear()** $\rightarrow$ **Dataset**

  Objetivo Crear un dataset vacío

  Salidas Un nuevo dataset

  Poscondición El dataset no contiene datos

Especificación del TAD *DataSet*
Operaciones Mutadoras

- **add(e)**

  Objetivo Añadir el elemento *e* al dataset

  Entradas Un dataset (*self*) y un elemento *e*

  Salidas El elemento *e* añadido al dataset

Especificación del TAD *DataSet*
Operaciones Observadoras

- **min()** $\rightarrow$ **`real`**
  - Objetivo: Devolver el valor mínimo del dataset
  - Entradas: Un dataset (*self*)
  - Salidas: Un valor `real`
  - Precondición: El tamaño del dataset es mayor que 0
- **max()** $\rightarrow$ **`real`**
  - Objetivo: Devolver el valor máximo del dataset
  - Entradas: Un dataset (*self*)
  - Salidas: Un valor `real`
  - Precondición: El tamaño del dataset es mayor que 0
- **tamaño()** $\rightarrow$ **`entero`**
  - Objetivo: Devolver el número de elementos del dataset
  - Entradas: Un dataset (*self*)
  - Salidas: Un valor `entero`

Especificación del TAD *DataSet*
Operaciones Observadoras

- **media()** $\rightarrow$ `real`

  Objetivo Devolver el valor medio del dataset

  Entradas Un dataset (*self*)

  Salidas Un valor `real`

  Precondición El tamaño del dataset es mayor que 0

- **desv_estandar()** $\rightarrow$ `real`

  Objetivo Devolver el valor de desviación estándar del dataset

  Entradas Un dataset (*self*)

  Salidas Un valor `real`

  Precondición El tamaño del dataset es mayor que 1

Implementación del TAD *DataSet*

- La nota numérica debería soportarse en un tipo predefinido `float`.
- Para calcular las estadísticas se necesita almacenar un conjunto de valores, podría ser, usando uno de los tipos colección predefinidos (una lista).
- Cada uno de los métodos estadísticos iterará la estructura para realizar los cálculos.

Implementación del TAD *DataSet*

```python
#TADClaseDataset.py
from math import sqrt, pow
class Dataset:
    def __init__(self):
        """Constructor"""
        self._data = []
        self._tamaño = 0

    def add(self, x):
        self._data.append(x)
        self._tamaño += 1

    def min(self):
        assert self._tamaño > 0
        return(min(self._data))
```

Implementación del TAD *DataSet*

```
def max(self):
    assert self._tamaño > 0
    return(max(self._data))

def tamaño(self):
    return(len(self._data))

def media(self):
    assert self._tamaño > 0
    return(sum(self._data)/self._tamaño)

def desv_estandar(self):
    assert self._tamaño > 1
    diferencias = 0
    _media = self.media()
    for i in self._data:
        diferencias += pow((i - _media), 2)
    return(sqrt(diferencias / (self._tamaño - 1)))
```

Implementación del TAD *DataSet* versión 2

- Representación alternativa:
  - Los métodos no necesitan conocer específicamente los valores.
  - `max` y `min` sólo requieren el mayor y el menor valor añadido.
  - la `media` sólo necesita la suma de valores y su número.
  - la desviación estándar se puede calcular con la formulación "shortcut": suma de cuadrados y suma de valores.

Implementación del TAD *DataSet* versión 2

```python
#TADClaseDatasetv2.py
from math import sqrt, pow
class Dataset:
    def __init__(self):
        """Constructor"""
        self._max = self._min = None
        self._tamaño = self._suma = self._suma_cuadrados = 0

    def add(self, x):
        if self._max == None:
            self._min = self._max = x
        elif x > self._max:
            self._max = x
        elif x < self._min:
            self._min = x
        self._tamaño += 1
        self._suma += x
        self._suma_cuadrados += pow(x, 2)
```

Implementación del TAD *DataSet* versión 2

```
def min(self):
    assert self._tamaño > 0
    return(self._min)

def max(self):
    assert self._tamaño > 0
    return(self._max)

def tamaño(self):
    return(self._tamaño)

def media(self):
    assert self._tamaño > 0
    return(self._suma/self._tamaño)

def desv_estandar(self):
    assert self._tamaño > 1
    return(sqrt((self._suma_cuadrados - (pow(self._suma,2)/self._tamaño))
```

Uso del TAD *DataSet*

```
# test_TADClaseDataset.py
# Programa de prueba del TAD Dataset
from TADClaseDataset import Dataset
#from TADClaseDatasetv2 import Dataset     <----------------

def main():
    print ('Programa para cálcular el min, max, media y desviación
    ↪  típica de un dataset')
    data = Dataset()
    while True :
        xStr = input ('Introducir un número (<Entrar> para
        ↪  terminar): ')
        if xStr == '':
            break
        data.add(float(xStr))
    print ('Resumen de ', data.tamaño(), ' notas.')
    print ('Mínimo: ' , data.min())
    print ('Máximo: ' , data.max())
    print ('Media: ' , data.media())
    print ('Desviación típ.: ' , data.desv_estandar())
if __name__ == '__main__':
    main ()
```

Diseñando un TAD: Ejemplo 3

Especificación del TAD *Bolsa*: Valores

- Una ***bolsa*** es una colección de un número arbitrario de elementos (todos del mismo tipo) en la que el orden (posición) no es relevante.
- Los elementos no tienen por qué ser necesariamente distintos. Es decir, pueden existir varias ocurrencias del mismo elemento.
- Las operaciones de acceso se limitan a añadir o eliminar ítems individuales, determinar si un ítem está en la bolsa, y recorrer la colección de ítems.
- Otras operaciones: unión, diferencia e intersección (constructoras productoras) y multiplicidad o número de ocurrencias (observadora).

Especificación del TAD *Bolsa*
Operaciones Constructoras Creadoras

- **crear()** $\rightarrow$ **Bolsa**

  Objetivo Crear una bolsa vacía

  Salidas Una nueva bolsa

  Poscondición La bolsa no contiene elementos

Especificación del TAD *Bolsa*
Operaciones Mutadoras

- **add(e)**

  Objetivo Añadir el elemento *e* a la bolsa
  Entradas Una bolsa (*self*) y un elemento *e*
  Salidas El elemento *e* añadido a la bolsa

- **remove(e) → e**

  Objetivo Eliminar una ocurrencia del elemento en la bolsa
  Entradas Una bolsa (*self*)
  Salidas La bolsa sin una ocurrencia del elemento *e*
  Precondición Al menos hay una ocurrencia del elemento
  Poscondición El número de ocurrencias del elemento disminuye en uno

Especificación del TAD *Bolsa*
Operaciones Observadoras

- **len()** → `integer`
  - Objetivo: Devolver el número de elementos de la bolsa
  - Entradas: Una bolsa (*self*)
  - Salidas: Un valor `entero`
- **contains(e)** → `booleano`
  - Objetivo: Determina si el elemento está en la bolsa
  - Entradas: Una bolsa (*self*)
  - Salidas: Un valor `booleano`
  - Poscondición: `True` si encuentra, `False` en caso contrario
- **iter()** → `iterator`
  - Objetivo: Devolver un iterador para recorrer la bolsa
  - Entradas: Una bolsa (*self*)
  - Salidas: Un iterador

Implementación del TAD *Bolsa*

```
#TADBolsa.py
#Implementa el contenedor TAD Bolsa usando una lista.
class Bolsa:
    def __init__( self ):
        """Constructor"""
        self._theItems = list()

    def add( self, item ):
        self._theItems.append( item )

    def __len__( self ):
        return len( self._theItems )

    def __contains__( self, item ):
        return item in self._theItems

    def __iter__( self ):
        return _BolsaIterador( self._theItems )

    def remove( self, item ):
        assert item in self._theItems, "El ítem debe estar en la lista."
        ndx = self._theItems.index( item )
        return self._theItems.pop( ndx )
```

Implementación del TAD *Bolsa*

```
#TADBolsa.py

#Iterador para el TAD Bolsa iplementado como una lista.
class _BolsaIterador :
    def __init__(self, theList):
        self._bagItems = theList
        self._curItem = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self._curItem < len(self._bagItems):
            item = self._bagItems[self._curItem]
            self._curItem += 1
            return item
        else:
            raise StopIteration
```

Uso del TAD *Bolsa*

```
from TADBolsa import Bolsa
def main():
    print('Implementación del carro de la ')
    print(' compra usando un TAD Bolsa.')

    bolsa = Bolsa()

    while True:
        articulo = input('Articulo (. fin): ')
        if articulo =='.':
            break
        unidades = int(input('Unidades: '))
        precio = float(input('Precio: ' ))
        for i in range(0,unidades):
            bolsa.add((articulo, precio))

    print('\nDevolución------')
    articulo = input('Articulo: ')
    precio = float(input('Precio: ' ))
    bolsa.remove((articulo, precio))

    print('\nEn caja---------')
    for item in bolsa:
        print('Articulo: {0}\t{1}'.format(item[0],
        ↪  item[1]))
        total += item[1]

    print('TOTAL: {0:5.2f}'.format(total))

if __name__ == '__main__':
    main ()
```

```
Implementación de un carro de la
 compra usando un TAD Bolsa.
Articulo (. fin): Leche
Unidades: 4
Precio: 0.6
Articulo (. fin): Pan
Unidades: 2
Precio: 0.5
Articulo (. fin): .

Devolución------
Articulo: Leche
Precio: 0.60

En caja---------
Articulo: Leche 0.6
Articulo: Leche 0.6
Articulo: Leche 0.6
Articulo: Pan   0.5
Articulo: Pan   0.5
TOTAL:  2.80
```

Bibliografía

Bibliografía

- David M. Reed and John Zelle. Data structures and algorithms using Python and C++. Franklin, Beedle & Associates incorporated, 2009.
- Rance Necaise. Data structures and algorithms using Python. John Wiley & Sons, INC., 2011.

2. Abstracción

Programación II

Grado en Inteligencia Artificial

M. Alonso, M. Cabrero y E. Hernández

fic facultade de informática da coruña

UNIVERSIDADE DA CORUÑA