Programmazione Avanzata

Design Pattern: Template Method

1

Il Design Pattern Template Method

- Il pattern Template Method e un design pattern comportamentale che permette di definire i passi di un algoritmo lasciando alle sottoclassi il compito di definire alcuni di questi passi.
- Vediamo un esempio in cui viene creata una classe astratta AbstractWordCounter class che fornisce due metodi.
  - il primo metodo, can_count(*filename*), restituisce un valore Booleano che indica se la classe può contare le parole del file dato in base all'estensione del file.
  - Il secondo metodo, count(*filename*), restituisce un conteggio di parole.
- Il codice comprende anche due sottoclassi, una che conta le parole in file di testo e l'altro per contare le parole in file HTML.

2

Il Design Pattern Template Method

- Tutti i metodi sono statici per cui non si ha mai a che fare con istanze della classe
- Il metodo count_words (esterna rispetto alla classi) itera su due oggetti classe (sottoclassi della classe atratta)
- Se una delle due classi può contare le parole nel file passato a count_words allora viene effettuato il conteggio e questo viene restituito dalla funzione.
- Se nessuna delle due classi è in grado di contare le parole del file, il metodo restituisce implicitamente None per indicare che non è stato in grado di effettuare il conteggio.

```
def count_words(filename):
    for wordCounter in (PlainTextWordCounter, HtmlWordCounter):
        if wordCounter.can_count(filename):
            return wordCounter.count(filename)
```


3

Il Design Pattern Template Method

- Di seguito sono mostrati due diversi codici per la classe astratta AbstractWordCounter.
- Questa classe fornisce i metodi che devono essere implementati nelle eventuali sottoclassi.

```
class AbstractWordCounter:

    @staticmethod
    def can_count(filename):
        raise NotImplementedError()

    @staticmethod
    def count(filename):
        raise NotImplementedError()
```

```
class AbstractWordCounter(
        metaclass=abc.ABCMeta):

    @staticmethod
    @abc.abstractmethod
    def can_count(filename):
        pass

    @staticmethod
    @abc.abstractmethod
    def count(filename):
        pass
```


4

Il Design Pattern Template Method

- Questa sottoclasse implementa il contatore per i file testuali e assume che i file con estensione .txt siamo codificati con UTF-8 (o 7-bit ASCII, che è un sottoinsieme di UTF-8).

re.finditer(*pattern*, *string*, *flags=0*) scandice string da sinistra a destra e restituisce i match (rispetto all'espressione regolare pattern nell'ordine in cui li trova.

```
class PlainTextWordCounter(AbstractWordCounter):

    @staticmethod
    def can_count(filename):
        return filename.lower().endswith(".txt")

    @staticmethod
    def count(filename):
        if not PlainTextWordCounter.can_count(filename):
            return 0
        regex = re.compile(r"\w+")
        total = 0
        with open(filename, encoding="utf-8") as file:
            for line in file:
                for _ in regex.finditer(line):
                    total += 1
        return total
```


5

Programmazione Avanzata

Design Pattern: Factory Method

6

Factory Method Pattern

- È un design pattern creazionale.
- Si usa quando vogliamo definire un'interfaccia o una classe astratta per creare degli oggetti e delegare le sue sottoclassi a decidere quale classe istanziare quando viene richiesto un oggetto.
  - Particolarmente utile quando una classe non può conoscere in anticipo la classe degli oggetti che deve creare.

7

Factory Method Pattern: un'applicazione

- **Esempio:** Consideriamo un framework per delle applicazioni ciascuna delle quali elabora documenti di diverso tipo.
  - Abbiamo bisogno di due astrazioni: la classe Application e la classe Document
    - La classe Application gestisce i documenti e li crea su richiesta dell'utente, ad esempio, quando l'utente seleziona Open o New dal menu.
  - Entrambe le classi sono astratte e occorre definire delle loro sottoclassi per poter realizzare le implementazioni relative a ciascuna applicazione
    - Ad esempio, per creare un'applicazione per disegnare, definiamo le classi DrawingApplication e DrawingDocument.
  - Definiamo un'interfaccia per creare un oggetto ma lasciamo alle sottoclassi decidere quali classi istanziare.

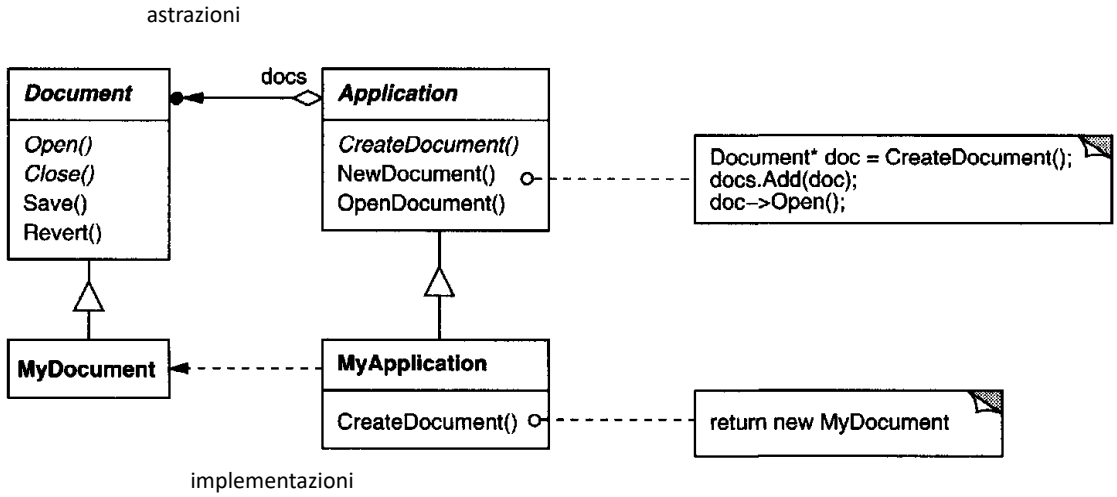
8

Factory Method Pattern: un'applicazione

- Poiché la particolare sottoclasse di Document da istanziare dipende dalla particolare applicazione, la classe Application non può fare previsioni riguardo alla sottoclasse di Document da istanziare
- La classe Application sa solo quando deve essere creato un nuovo documento ma non ne conosce il tipo.
- **Problema:** devono essere istanziate delle classi ma si conoscono solo delle classi astratte che non possono essere istanziate
- Il Factory method pattern risolve questo problema incapsulando l'informazione riguardo alla sottoclasse di Document da creare e sposta questa informazione all'esterno del framework.

9

Factory Method Pattern: un'applicazione



10

Factory Method Pattern: un'applicazione

- Le sottoclassi di Application ridefiniscono il metodo astratto CreateDocument per restituire la sottoclasse appropriata di Document
- Una volta istanziata, la sottoclasse di Application può creare istanze di Document per specifiche applicazioni senza dover conoscere le sottoclassi delle istanze create (CreateDocument)
- CreateDocument è detto factory method perché è responsabile della creazione degli oggetti

11

Factory Method Pattern: un esempio

Voglio creare una scacchiera per la dama ed una per gli scacchi

```
def main():
    checkers = CheckersBoard()
    print(checkers)

    chess = ChessBoard()
    print(chess)
```


12

Factory Method Pattern: un esempio

- la scacchiera è una lista di liste (righe) di stringhe di un singolo carattere
- __init__ Inizializza la scacchiera con tutte le posizioni vuote e poi invoca populate_board per inserire i pezzi del gioco
- populate_board è astratto

- La funzione console() restituisce una stringa che rappresenta il pezzo ricevuto in input sul colore di sfondo passato come secondo argomento.

```
BLACK, WHITE = ("BLACK", "WHITE")

class AbstractBoard:

    def __init__(self, rows, columns):
        self.board = [[None for _ in range(columns)] for _ in range(rows)]
        self.populate_board()

    def populate_board(self):
        raise NotImplementedError()

    def __str__(self):
        squares = []
        for y, row in enumerate(self.board):
            for x, piece in enumerate(row):
                square = console(piece, BLACK if (y + x) % 2 else WHITE)
                squares.append(square)
            squares.append("\n")
        return "".join(squares)
```

13

Factory Method Pattern: un esempio

- La classe per creare scacchiere per il gioco della dama

```
class CheckersBoard(AbstractBoard):

    def __init__(self):
        super().__init__(10, 10)

    def populate_board(self):
        for x in range(0, 9, 2):
            for row in range(4):
                column = x + ((row + 1) % 2)
                self.board[row][column] = BlackDraught()
                self.board[row + 6][column] = WhiteDraught()
```


14

Factory Method Pattern: un esempio

- La classe per scacchiere per il gioco degli scacchi

```
class ChessBoard(AbstractBoard):

    def __init__(self):
        super().__init__(8, 8)

    def populate_board(self):
        self.board[0][0] = BlackChessRook()
        self.board[0][1] = BlackChessKnight()
        ...
        self.board[7][7] = WhiteChessRook()
        for column in range(8):
            self.board[1][column] = BlackChessPawn()
            self.board[6][column] = WhiteChessPawn()
```

I metodi populate_board() di CheckersBoard e ChessBoard non sono dei factory method dal momento che le classi usate per creare i pezzi sono fissate nel codice.

15

Factory Method Pattern: un esempio

- La classe base per i pezzi
- Si è scelto di creare una classe che discende da str invece che usare direttamente str per poter facilmente testare se un oggetto z è un pezzo del gioco con isinstance(z,Piece)
- ponendo __slots__=() ci assicuriamo che gli oggetti di tipo Piece non abbiano variabili di istanza

```
class Piece(str):
    __slots__ = ()
```


16

Factory Method Pattern: un esempio

- La classe pedina nera e la classe re bianco
- le classi per gli altri pezzi sono create in modo analogo
  - Ognuna di queste classi è una sottoclasse immutabile di Piece che è sottoclasse di str
  - Inizializzata con la stringa di un unico carattere (il carattere Unicode che rappresenta il pezzo)

```
class BlackDraught(Piece):

    __slots__ = ()

    def __new__(Class):
        return super().__new__(Class, "\N{black draughts man}")

class WhiteChessKing(Piece):

    __slots__ = ()

    def __new__(Class):
        return super().__new__(Class, "\N{white chess king}")
```


17

Factory Method Pattern: un esempio

- Notiamo che qui la stringa che indica il pezzo è assegnata da __new__
- Il metodo __new__ non prende argomenti in quanto la stringa che rappresenta il pezzo è codificato all'interno del metodo.
  - TypeError: __new__() takes 1 positional argument but 2 were given
- Per i tipi che estendono tipi immutable, come str, l'inizializzazione è fatta da __new__.
  - https://docs.python.org/3/reference/datamodel.html : __new__() is intended mainly to allow subclasses of immutable types (like int, str, or tuple) to customize instance creation.

```
class BlackDraught(Piece):

    __slots__ = ()

    def __new__(Class):
        return super().__new__(Class, "\N{black draughts man}")

class WhiteChessKing(Piece):

    __slots__ = ()

    def __new__(Class):
        return super().__new__(Class, "\N{white chess king}")
```


18

Factory Method Pattern: un esempio

- Questa nuova versione del metodo CheckersBoard.populate_board() è un **factory method** in quanto dipende dalla factory function create_piece()
- Nella versione precedente il tipo di pezzo era indicato nel codice
- La funzione create_piece() restituisce un oggetto del tipo appropriato (ad esempio, BlackDraught o WhiteDraught) in base ai suoi argomenti.
- Il metodo ChessBoard.populate_board() viene anch'esso modificato in modo da usare la stessa funzione create_piece() invocata qui.

```
def populate_board(self):
    for x in range(0, 9, 2):
        for y in range(4):
            column = x + ((y + 1) % 2)
            for row, color in ((y, "black"), (y + 6, "white")):
                self.board[row][column] = create_piece("draught",
                        color)
```

19

Factory Method Pattern: un esempio

- La funzione factory create_piece usa la funzione built-in eval() per creare istanze della classe
- Ad esempio se gli argomenti sono "knight" and "black", la stringa valutata sarà "BlackChessKnight()".
- In generale è meglio non usare eval per eseguire il codice rappresentato da un'espressione perché è potenzialmente rischioso dal momento che permette di eseguire il codice rappresentato da una qualsiasi espressione

```
def create_piece(kind, color):
    if kind == "draught":
        return eval("{}{}()".format(color.title(), kind.title()))
    return eval("{}Chess{}()".format(color.title(), kind.title()))
```


20

Factory Method Pattern: un esempio

- Questa versione di CheckersBoard.populate_board() differisce da quella precedente in quanto il pezzo e il colore sono specificati da costanti e usa un nuovo factory create_piece().

```
DRAUGHT, PAWN, ROOK, KNIGHT, BISHOP, KING, QUEEN = ("DRAUGHT", "PAWN",
       "ROOK", "KNIGHT", "BISHOP", "KING", "QUEEN")

class CheckersBoard(AbstractBoard):
    ...

    def populate_board(self):
        for x in range(0, 9, 2):
            for y in range(4):
                column = x + ((y + 1) % 2)
                for row, color in ((y, BLACK), (y + 6, WHITE)):
                    self.board[row][column] = self.create_piece(DRAUGHT,
                            color)
```


Factory Method Pattern: un esempio

- Questa versione di create_piece() è un metodo di AbstractBoard che viene ereditato da CheckersBoard e ChessBoard.
- Prende in input due costanti kind e color e cerca nel dizionario __classForPiece la classe associata alla chiave (kind,color)
- La classe cosi` individuata viene quindi utilizzata per instanziare il pezzo desiderato.

```
class AbstractBoard:

    __classForPiece = {(DRAUGHT, BLACK): BlackDraught,
            (PAWN, BLACK): BlackChessPawn,
            ...
            (QUEEN, WHITE): WhiteChessQueen}
    ...
    def create_piece(self, kind, color):
        return AbstractBoard.__classForPiece[kind, color]()
```

Factory Method Pattern: un esempio

- L'ultima versione di create_piece comunque fa uso di informazioni riguardanti le sottoclassi di Piece che si trovano all'interno della classe AbstractBoard e in particolare nel dizionario __classForPiece di quella classe
- La versione di create_piece riportata in basso non usa il dizionario AbstractBoard. __classForPiece ma ricerca direttamente la sottoclasse da utilizzare nel dizionario restituito da globals()

```
def create_piece(kind, color):
    color = "White" if color == WHITE else "Black"
    name = {DRAUGHT: "Draught", PAWN: "ChessPawn", ROOK: "ChessRook",
            KNIGHT: "ChessKnight", BISHOP: "ChessBishop",
            KING: "ChessKing", QUEEN: "ChessQueen"}[kind]
    return globals()[color + name]()
```


Factory Method Pattern: un esempio

- Un modo di rendere piu` dinamiche le implementazioni fino ad ora viste è di aggiungere le sottoclassi dinamicamente:
  - Invece di codificare tutte le sottoclassi di Piece una ad una (staticamente), possiamo crearle dinamicamente con il seguente frammento di codice:

```
for code in itertools.chain((0x26C0, 0x26C2), range(0x2654, 0x2660)):
    char = chr(code)
    name = unicodedata.name(char).title().replace(" ", "")
    if name.endswith("sMan"):
        name = name[:-4]
    new = make_new_method(char)
    Class = type(name, (Piece,), dict(__slots__=(), __new__=new))
    setattr(sys.modules[__name__], name, Class) # Can be done better!
```

Factory Method Pattern: un esempio

- Questo frammento di codice scandisce tutti i numeri esadecimali associati alle pedine e agli scacchi e per ciascuno di essi
  - Salva in **char** la stringa che rappresenta il carattere Unicode ad esso associato.
  - Salva in **name** la stringa assegnata a **char** dopo aver trasformato in maiuscole le iniziali di tutte le parole al suo interno ed eliminato gli spazi
    - Ad esempio per code= 0x2654, setta char='♔' e name='WhiteChessKing'
  - Se name finisce con "sMan" (cioe` se è un pezzo della dama) cancella il suffisso 'sMan' da name
  - Invoca make_new_method(char) per creare una nuova funzione che viene memorizzata in **new**

```
for code in itertools.chain((0x26C0, 0x26C2), range(0x2654, 0x2660)):
    char = chr(code)
    name = unicodedata.name(char).title().replace(" ", "")
    if name.endswith("sMan"):
        name = name[:-4]
    new = make_new_method(char)
    Class = type(name, (Piece,), dict(__slots__=(), __new__=new))
    setattr(sys.modules[__name__], name, Class) # Can be done better!
```

la descrizione del corpo del for continua nella slide successiva

25

Factory Method Pattern: un esempio

- Memorizza in Class una nuova classe. Questa nuova classe è ottenuta invocando la funzione built-in **type** con i seguenti argomenti:
  - name : nome della classe
  - (Piece,): tupla delle classi base della classe
  - dizionario **dict**(__slots__=(), __new__=new)) : dizionario degli attributi della classe
    - in questo modo le istanze della classe non avranno __dict__ e sarrano create usando il metodo new
- aggiunge la classe Class al modulo corrente con setattr (l'attributo corrispondente avra` lo stesso nome della classe (name))

```
for code in itertools.chain((0x26C0, 0x26C2), range(0x2654, 0x2660)):
    char = chr(code)
    name = unicodedata.name(char).title().replace(" ", "")
    if name.endswith("sMan"):
        name = name[:-4]
    new = make_new_method(char)
    Class = type(name, (Piece,), dict(__slots__=(), __new__=new))
    setattr(sys.modules[__name__], name, Class) # Can be done better!
```


26

Factory Method Pattern: un esempio

- def make_new_method(char): # crea un nuovo metodo ogni volta

```
def new(Class): # Can't use super() or super(Piece, Class)
    return Piece.__new__(Class, char)
return new
```


Factory Method Pattern: un esempio

class **type**(*name, bases, dict, **kwds*)

- Cosa dice la documentazione:
- type invocato con tre argomenti restituisce un nuovo oggetto tipo (una nuova classe). Questa è essenzialmente una forma dinamica dello statement class.
- La stringa *name* string è il nome della classe e diventa il valore dell'attributo __name__
- La tupla *bases* contiene le classi base e diventa l'attributo __bases__; se è vuota viene aggiunta object come classe base
- Il dizionario *dict* dictionary contiene le definizioni degli attributi e dei metodi della classe.
- Questi due frammenti di codice creano la stessa classe:
  - **class X**:
    a = 1
  - X = type('X', (), dict(a=1))