Programmazione Avanzata

Design Pattern: Template method

Il Design Pattern Template Method

- Il design pattern Template Method è un design pattern comportamentale che permette di definire i passi di un algoritmo demandando pero` alle sottoclassi il compito di definire alcuni di questi passi.
- Come esempio, consideriamo una classe AbstractWordCounter che fornisce due metodi:
  - can_count(*filename*) che restituisce True se la classe puo` contare le parole del file passato come argomento (basandosi sull'estensione del file)
  - count(*filename*) restituisce il numero di parole.
- La classe AbstractWordCounter ha due sottoclassi:
  - una per contare le parole di file di testo e l'altra per contare le parole di file HTML.

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 (esterno rispetto alla classi) itera su due oggetti classe (sottoclassi della classe atratta)
- Se una delle due classi puo` 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 e` in grado di contare le parole del file. il metodo restituisce implicitamente None per indicare che non e` 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)
```


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
```

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 e` un sottoinsieme di UTF-8).

re.compile(*pattern*) compila un'espressione regolare in un oggetto che puo` essere usato come pattern per fare il matching usando metodi quali match(), search(), finditer().

re.finditer(*pattern*) scandice string da sinistra a destra e restituisce un iteratore dei match (rispetto all'espressione regolare pattern)

```
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
```


Programmazione Avanzata

Design Pattern: Factory Method

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.

Factory Method Pattern: un'applicazione

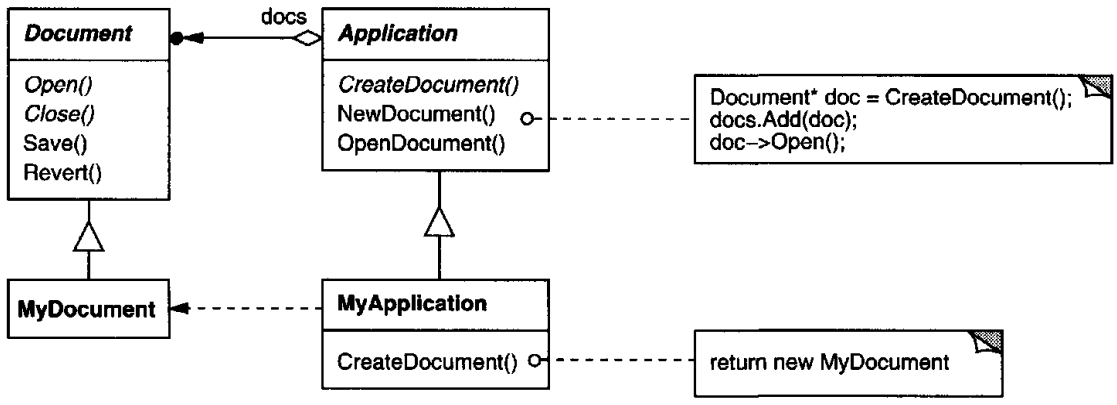
- **Esempio:** Consideriamo un framework per delle applicazioni ciascuna delle quali elabora documenti di diverso tipo.
  - Abbiamo bisogno di due astrazioni: la classe Application e la classe Document
    - La classe Application gestisce i documenti e li crea su richiesta dell'utente, ad esempio, quando 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.

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.

115

Factory Method Pattern: un'applicazione



116

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

Factory Method Pattern: un semplice esempio

```
class Pizza():
        def __init__(self):
            self._price = None

        def get_price(self):
            return self._price
```

Factory Method Pattern: un semplice esempio

```
class HamAndMushroomPizza(Pizza):
        def __init__(self):
            self._price = 8.5

class DeluxePizza(Pizza):
        def __init__(self):
            self._price = 10.5

class HawaiianPizza(Pizza):
        def __init__(self):
            self._price = 11.5
```


Factory Method Pattern: un semplice esempio

- PizzaFactory fornisce il metodo createPizza che è statico per cui può essere invocato quando non è stata ancora creata una pizza

```
class PizzaFactory:
    @staticmethod
    def create_pizza(pizza_type):
        if pizza_type == 'HamMushroom':
            return HamAndMushroomPizza()
        elif pizza_type == 'Deluxé:
            return DeluxePizza()
        elif pizza_type == 'Hawaiian':
            return HawaiianPizza()
```

Factory Method Pattern: un semplice esempio

```
if __name__ == '__main__':
    for pizza_type in ('HamMushroom', 'Deluxé, 'Hawaiian'):
       print('Price of {0} is {1}'.format(pizza_type,
       PizzaFactory.create_pizza(pizza_type).get_price())
```

- il tipo di pizza avrebbe potuto essere fornito dall'utente
- il tipo di pizza indicato dall'utente potrebbe essere stato inserito successivamente nel menu e la classe concreta corrispondente creata successivamente al main
- occorre modificare solo la factory

Factory Method Pattern: un semplice esempio

- Che cosa accade se vogliamo creare diversi tipi di negozi ciascuno dei quali vende pizze nello stile di una certà città
- Creiamo una classe astratta PizzaStore al cui interno c'è il metodo astratto **create_pizza**
- Dalla classe PizzaStore deriviamo NYPizzaStore, ChicagoPizzaStore e così via. Queste sottoclassi sovrascriveranno il metodo astratto. La decisione sul tipo di pizza da creare è presa dal metodo create_pizza della specifica sottoclasse.
  - analogamente a quanto accadeva nel framework per la gestione dei documenti
- PizzaStore avrà anche un metodo orderPizza() che invoca createPizza ma non ha idea su quale pizza verrà creata fino a che non verra` creata una classe concreta di PizzaStore

Factory Method Pattern: un semplice esempio

```
from abc import ABC, abstractmethod

class Pizza(ABC):

	@abstractmethod
	def prepare(self):
		pass

	def bake(self):
		print("baking pizza for 12min in 400 degrees..")

	def cut(self):
		print("cutting pizza in pieces")

	def box(self):
		print("putting pizza in box")
```


123

Factory Method Pattern: un semplice esempio

```
class NYStyleCheesePizza(Pizza):
	def prepare(self):
		print("preparing a New York style cheese pizza..")

class ChicagoStyleCheesePizza(Pizza):
	def prepare(self):
		print("preparing a Chicago style cheese pizza..")

class NYStyleGreekPizza(Pizza):
	def prepare(self):
		print("preparing a New York style greek pizza..")

class ChicagoStyleGreekPizza(Pizza):
	def prepare(self):
		print("preparing a Chicago style greek pizza..")
```


124

Factory Method Pattern: un semplice esempio

```
class PizzaStore(ABC):

      @abstractmethod
      def _createPizza(self, pizzaType: str) -> Pizza:
            pass

      def orderPizza(self, pizzaType):


            pizza = self._createPizza(pizzaType)

            pizza.prepare()
            pizza.bake()
            pizza.cut()
            pizza.box()
```


125

Factory Method Pattern: un semplice esempio

```
class NYPizzaStore(PizzaStore):

      def _createPizza(self, pizzaType: str) -> Pizza:

            pizza = None

            if pizzaType == 'Greek':
                  pizza = NYStyleGreekPizza()
            elif pizzaType == 'Cheese':
                  pizza = NYStyleCheesePizza()
            else:
                  print("No matching pizza found in the NY pizza store...")
            return pizza
```


126

Factory Method Pattern: un semplice esempio

```
class ChicagoPizzaStore(PizzaStore):

      def _createPizza(self, pizzaType: str) -> Pizza:

            pizza = None

            if pizzaType == 'Greek':
                  pizza = ChicagoStyleGreekPizza()
            elif pizzaType == 'Cheese':
                  pizza = ChicagoStyleCheesePizza()
            else:
                  print("No matching pizza found in the Chicago pizza store...")
            return pizza
```


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)
```

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)
```

129

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()
```


130

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()
```


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__ = ()
```

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}")
```


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}")
```

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)
```

135

Factory Method Pattern: un esempio

- Questa funzione factory 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()))
```


136