Esempio di strutturazione del codice mediante l'impiego di Design Pattern

November 28, 2022

1 Introduzione

Il progetto "Esempio di strutturazione del codice" mostra alcune tecniche di programmazione e di strutturazione del codice che migliorano le sue caratteristiche di:

- *generalità della soluzione*: il codice risolve una classe più generale di problemi;
- *menutenibilità*: il codice può essere modificato per: (1) aggiungere nuove funzionalità, (2) migliorarne le prestazioni o correggerne errori con uno sforzo contenuto, minimizzando il numero e l'entità delle modifiche al codice esistente;
- *leggibilità*: è più facile comprendere la logica che ha originato il codice e verificarne la correttezza.

Il problema che si vuole risolvere è quello di acquisire i record relativi a un'agenda e memorizzarli in un file JSON sotto forma di lista di tabelle. Per semplicità si assume che ogni record sia formato da un nome, un cognome, un numero di telefono e un indirizzo di posta elettronica (quest'ipotesi non toglie generalità alle soluzioni proposte).

Il progetto parte da una soluzione banale e molto limitata al problema posto, e sviluppa con commit successivi versioni della soluzione più ricche dal punto di vista funzionale e con migliori caratteristiche di manutenibilità e leggibilità. Usando il comando:

```
git diff <commit successivo> <commit precedente>
```

si può apprezzare le modifiche apportate da ogni versione della soluzione rispetto alle precedenti.

Di seguito si commentano i commit corrispondenti alle versioni successive prodotte, evidenziando come esse migliorano le caratteristiche sopra citate.

2 Soluzione banale (v1.0)

In questa versione i record vengono immessi manualmente dallo standard input e il loro numero è noto a priori. Si tratta di una soluzione banale con funzionalità limitate.

3 Prima generalizzazione (v1.1)

In questa versione i record vengono immessi manualmente dallo standard input, ma il loro numero non è noto a priori. La fine della lista di record viene segnalata da opportuni valori in input. È una soluzione leggermente più generale, ma non cambia l'organizzazione del codice.

4 Introduzione delle function (v1.2)

In questa versione vengono introdotti dei sottoprogrammi per decomporre la soluzione in parti ciascuna delle quali ha un compito ben definito. In particolare, viene definita un'interfaccia per la funzione che acquisisce un record e che resterà invariata in tutte le soluzioni successive. La semantica di tale interfaccia è definita dal seguente codice e in particolare attraverso il commento che segue la prima riga:

[ ]:
```
def get_record():
    """
    acquisisce un record e lo restituisce sotto forma di dizionario
    restituisce un dizionario vuoto per indicare che non vi sono altri
    record da acquisire

    Returns
    -------
    dict
        il record sotto forma di dizionario
        restituisce un dizionario vuoto per indicare che non vi sono altri
        record da acquisire
    """
```

Questa volta non cambiano le funzionalità offerte dalla soluzione che quindi non è più generale, ma cambia modo sostanziale la struttura del codice. L'introduzione dei sottoprogrammi e il conseguente isolamento di funzionalià specifiche con interfacce ben definite migliora la manutenibilità e la leggibilità dal codice come si può apprezzare confrontando il codice della precedente versione:

[ ]:
```
# acquisisce i record
agenda = []
while True:
    nome = input('Nome: ')
    if nome == 'stop':  # non vi sono altri record da acquisire
        break
    record = {}
    record['nome'] = nome
    record['cognome'] = input('Cognome: ')
    record['telefono'] = input('Telefono: ')
    record['email'] = input('Email: ')
    agenda.append(record)

# visualizza i record acquisiti
for record in agenda:
    print('-----------')
    for key in record:
        print(f'{key}: {record[key]}')
print('-----------')

# salva i record acquisiti in formato json
```

```
fout = open('../temp/agenda.json', 'w')
dump(agenda, fout, indent=3)
fout.close()
```

con il programma principale della versione `v1.2`:

[ ]:
```
if __name__ == "__main__":
    # acquisisce i record
    agenda = []
    while True:
        record = get_record()
        if record == {}:  # non vi sono altri record da acquisire
            break
        agenda.append(record)

    display_records(agenda)

    save_json(agenda, '../temp/agenda.json')
```

5 Migliorare la leggibilità (v1.3)

In questa versione viene introdotto un generatore (`records`) per miglioraere la leggibilità del programma principale incapsulando nel generatore la logica di ripetizione con cui vengono acquisiti più record.

Con l'introduzione del generatore il programma principale diventa infatti:

[ ]:
```
if __name__ == "__main__":
    # acquisisce la lista di record
    agenda = [record for record in records(get_record)]

    display_records(agenda)

    save_json(agenda, '../temp/agenda.json')
```

Vedremo che questa modifica faciliterà l'aggiunta di nuove funzionalità nelle versioni successive.

6 Introduzione di una classe (v1.4)

In questa versione viene introdotta una classe che incapsula l'acquisizione di un record. Non cambia la generalità della soluzione, ma la strutturazione del codice. In particolare, l'introduzione della classe `Record_Stdin_Reader` permette di introdurre sorgenti di dati differenziate nelle successive versioni.

Si noti che un risultato simile si sarebbe potuto fare anche usando le function (una function diversa per ogni sorgente), ma con minori opportunità di sfruttare il polimorfismo nelle versioni successive (si veda la versione v3.0 in cui viene impiegato il design pattern `Strategy`), ma anche maggiore

difficoltà di implementazione per mantenere la semantica dell'interfaccia dei `get_record` (si veda le considerazioni fatte sulla versione v2.0).

7 Introduzione di una seconda classe (v2.0)

Introduzione di una seconda classe che acquisisce i record da una sorgente diversa (un file JSON).

Notare come la seconda classe implementa la semantica del metodo get_record:

- quando l'oggetto viene creato il file JSON viene letto interamente memorizzando nell'oggetto della classe `Record_JSONfile_Reader` la lista dei record;
- per restituire un record diverso ad ogni chiamata del metodo `get_record`, viene mantenuto un indice che identifica il prossimo record che deve essere restituito.

8 Decomposizione della soluzione in più file (v2.1)

Le classi che implementano l'acquisizione di record vengono isolate in un modul e importate nello script principale.

Con l'occasione, è stata introdotta una terza classe che acquisisce i record da una sorgente diversa (una GUI Qt).

La GUI è generata usando QtDesigner e gestita da programma tramite il package `PySide6`. Il codice generato tramite QtDesigner e l'applicazione `pyside6-uic` è mantenuto in una sottodirectory dedicata (vedi documentazione di `PySide6` per maggiori dettagli).

Per interagire con la GUI è stata introdotta la classe `Record_Dialog` sottoclasse della classe `QDialog`, fornita dal modulo `QtWidgets` del package `PySide6`, e della classe `Ui_GetRecord`, generata dall'applicazione `pyside6-uic`.

9 Introduzione del pattern `strategy` (v3.0)

Introduzione del pattern `strategy` per rendere trasparente al client l'algoritmo di acquisizione della lista di record e allo stesso tempo rendere tale algoritmo (cioé l'uso combinato di generator e comprehension) indipendente dalla natura della sorgente dei dati.

Il pattern strategy ha richiesto l'introduzione di una classe astratta `Record_Reader`.

10 Introduzione del pattern `factory` (v3.1)

Oltre al pattern strategy, viene inoltre introdotto un pattern `factor` semplificato realizzato tramite un metodo statico della classe `Record_Reader` con l'obiettivo di eliminare la dipendenza del client dalle classi concrete che acquisiscono record.

Si noti che in questo modo nello script principale non occorre più importare le singole classi concrete che implementano l'acquisizione di record da sorgenti diverse e pertanto l'introduzione di ulteriori lettori di record non comportano modifiche allo script principale, ma solo alla factory che deve prevedere ulteriori casi.

11 Introduzione del pattern `decorator` (v4.0)

Viene introdotto il pattern `decorator` per fare in modo che i nomi e i cognomi vengano acquisiti con il primo carattere maiuscolo e i rimanenti minuscoli.

Anche in questo caso si usa una factory per eliminare la dipendenza del client dalle classi concrete di decorator.

Si noti come l'introduzione di decorator non richiede alcuna modifica al file `record_readers.py`, né alla classe `context` che implementa l'algoritmo di acquisizione di una lista di record da una stessa sorgente, ma solo allo script principale (il client), che deve creare il decoratore e collegarlo allo oggetto da decorare:

[ ]:
```
# client
if __name__ == "__main__":
    # acquisisce la lista di record da un file JSON
    context = Context(
        Record_Reader.create_instance(source_type='json file',
                                      fname='../data/agenda_old.json'))
    agenda = context.get_list_of_records()
    # acquisisce la lista di record da una GUI Qt assicurandosi che nomi
    # e cognomi siano stringhe di lettere minuscole che iniziano con maiuscole
    source = Record_Reader.create_instance(source_type='Qt dialog')
    context.source = \
        Record_Reader_Decorator.create_instance(source, \
                                                decorator_type='capitalize')
    agenda += context.get_list_of_records()

    display_records(agenda)

    save_json(agenda, '../temp/agenda.json')
```

12 Introduzione del file `requirements.txt`

Infine, poiché il progetto nella sua versione finale richiede il package `PySide6` che non è standard, è stato generato il file `requirements.txt` che elenca i package da installare:

```
PySide6==6.3.2
PySide6-Addons==6.3.2
PySide6-Essentials==6.3.2
shiboken6==6.3.2
```

Nel caso si voglia eseguire il codice con un interprete per il quale non sono stati installati i package `PySide6` sarà sufficiente eseguire il comando:

```
pip install -r requirements.txt
```