Opracował: Mateusz Zubrzycki, 272568

Instrukcja

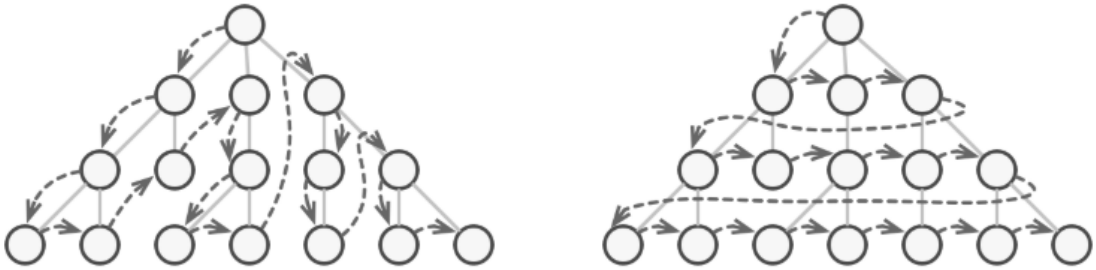
Wzorzec:

Iterator - behawioralny wzorzec projektowy, pozwalający sekwencyjnie przechodzić od elementu do elementu jakiegoś zbioru bez konieczności eksponowania jego formy (lista, stos, drzewo, itp.).

Problem:

W programowaniu często spotykamy się z potrzebą przeglądania elementów kolekcji (np. listy, stosu, zbioru), ale bez ujawniania jej wewnętrznej reprezentacji. Trudność pojawia się, gdy chcemy zapewnić jednolity sposób przeglądania różnych struktur danych bez duplikowania logiki przechodzenia przez elementy.

Problemem jest zatem **zapewnienie zunifikowanego mechanizmu iterowania** po elementach kolekcji, niezależnie od jej typu wewnętrznego. Potrzebujemy rozwiązania, które pozwoli „klientowi” kolekcji na uzyskiwanie kolejnych elementów, nie naruszając zasady hermetyzacji.



Rozwiązania:

Główną ideą wzorca **Iterator** jest ekstrakcja zadań związanych z przechodzeniem przez elementy kolekcji do osobnego obiektu zwanego ***iteratorem***.

Oprócz implementowania samego algorytmu, obiekt iteratora hermetyzuje wszystkie szczegóły sposobu przechodzenia przez kolejne elementy, jak bieżąca pozycja, czy ilość pozostałych elementów. Dzięki temu wiele iteratorów może jednocześnie przeglądać tę samą kolekcję, niezależnie od siebie.

Zazwyczaj, iteratory udostępniają jedną główną metodę pobierającą elementy kolekcji. Klient może wywoływać ją raz za razem aż przestanie ona zwracać kolejne obiekty, co oznacza osiągnięcie końca zbioru.

Wszystkie iteratory muszą implementować ten sam interfejs. Czyni to kod klienta kompatybilnym z dowolną kolekcją czy algorytmem przechodzenia o ile istnieje odpowiedni iterator. Jeśli potrzebny jest specjalny sposób przeglądania kolekcji, można stworzyć nową klasę iteratora, bez konieczności dokonywania zmian w kolekcji lub w kodzie klienta.

Konsekwencje:

Zalety:

- Umożliwia przechodzenie przez kolekcje bez ujawniania ich implementacji.
- Upraszcza kod klienta – iterowanie jest zunifikowane.
- Pozwala na wiele iteratorów działających jednocześnie na tej samej kolekcji.
- Ułatwia dodawanie nowych sposobów iteracji (np. od końca, z filtrem).

Wady:

- Wprowadza dodatkowe klasy, co zwiększa złożoność projektu.
- Może być nieoptymalne dla bardzo prostych kolekcji (np. jednorazowe przejście przez listę).
- Jeśli iterator ma modyfikować kolekcję, trzeba zadbać o bezpieczeństwo współbieżności i spójność danych.

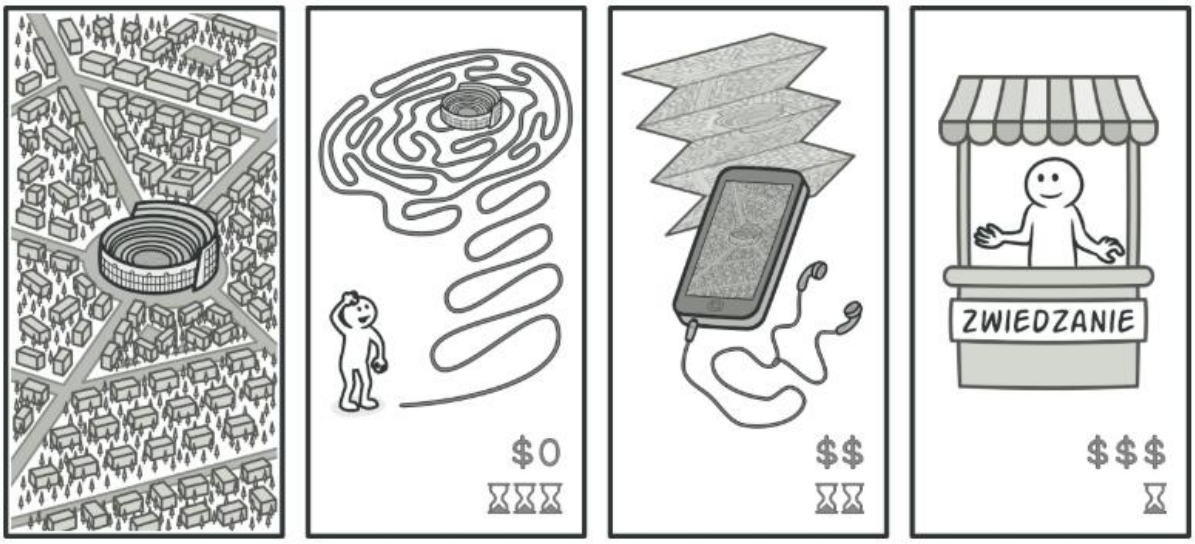
Przykład:

Zamierzasz odwiedzić Rzym na parę dni i zwiedzić wszystkie najważniejsze miejsca i atrakcje turystyczne. Ale na miejscu łatwo zmarnować sporo czasu chodząc w kółko, nie mogąc znaleźć nawet Koloseum.

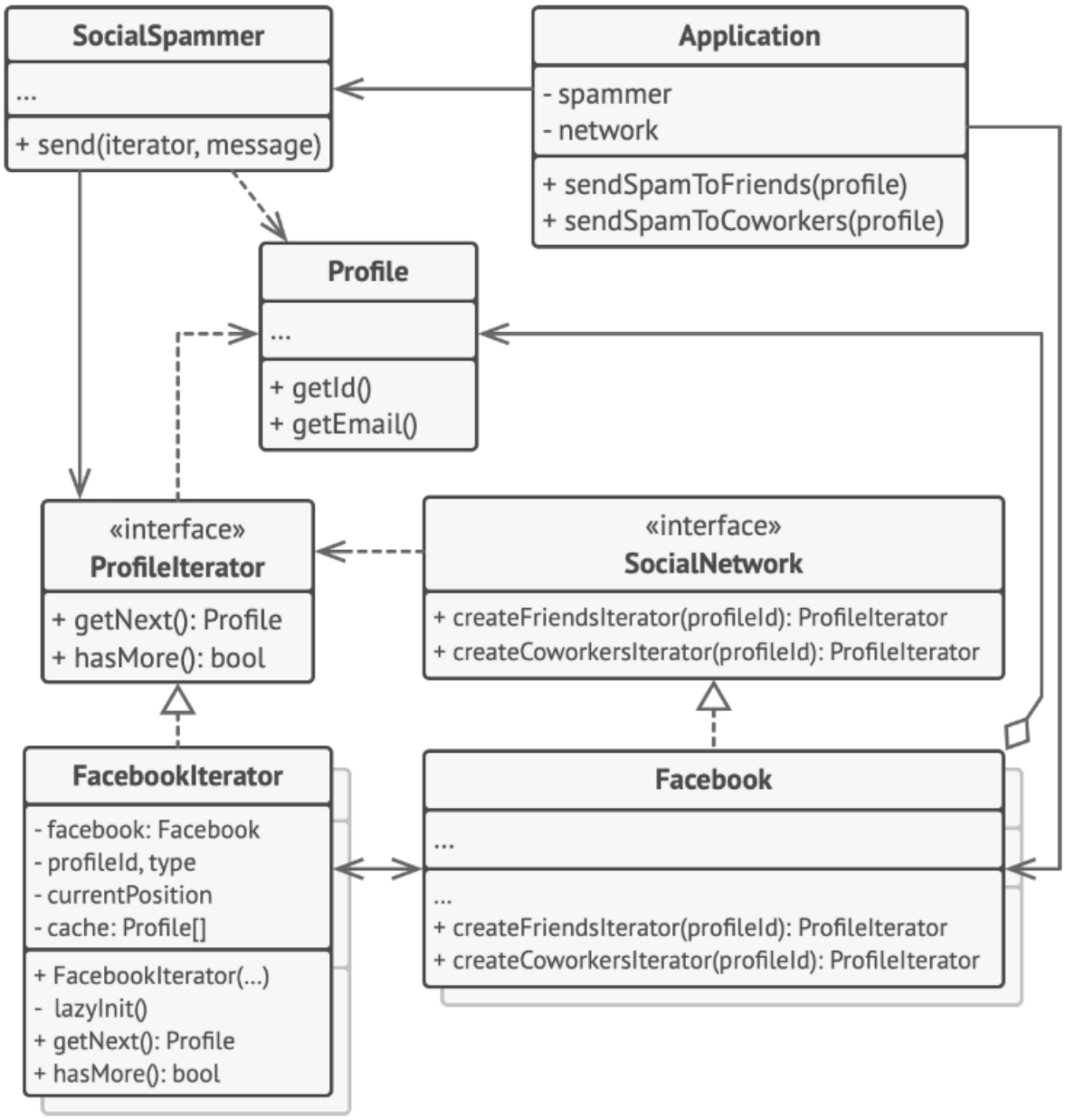
Z drugiej strony, można kupić aplikację wirtualnego przewodnika na smartfon i nawigować z jej pomocą. Sprytne i niedrogie, a dodatkowo można zatrzymać się przy dowolnym miejscu na ile się chce.

Trzecia alternatywa to poświęcenie części swojego wycieczkowego budżetu na wynajęcie przewodnika który zna miasto jak własną kieszeń. Przewodnik mógłby dostosować trasę wycieczki do twoich preferencji, pokazać wszystko co najciekawsze i opowiedzieć wiele ciekawych historii. Byłoby miło, ale również i drożej.

Wszystkie te opcje — losowo obrane kierunki, smartfonowy przewodnik i wynajęty przewodnik — stanowią iteratory pozwalające na dostęp do wielkiej kolekcji widoków i atrakcji Rzymu.



Pseudokod:


SocialSpammer
...
+ send(iterator, message)
Application
- spammer
- network
+ sendSpamToFriends(profile)
+ sendSpamToCoworkers(profile)
Profile
...
+ getId()
+ getEmail()
«interface»
ProfileIterator
+ getNext(): Profile
+ hasMore(): bool
«interface»
SocialNetwork
+ createFriendsIterator(profileId): ProfileIterator
+ createCoworkersIterator(profileId): ProfileIterator
FacebookIterator
- facebook: Facebook
- profileId, type
- currentPosition
- cache: Profile[]
+ FacebookIterator(...)
- lazyInit()
+ getNext(): Profile
+ hasMore(): bool
Facebook
...
...
+ createFriendsIterator(profileId): ProfileIterator
+ createCoworkersIterator(profileId): ProfileIterator