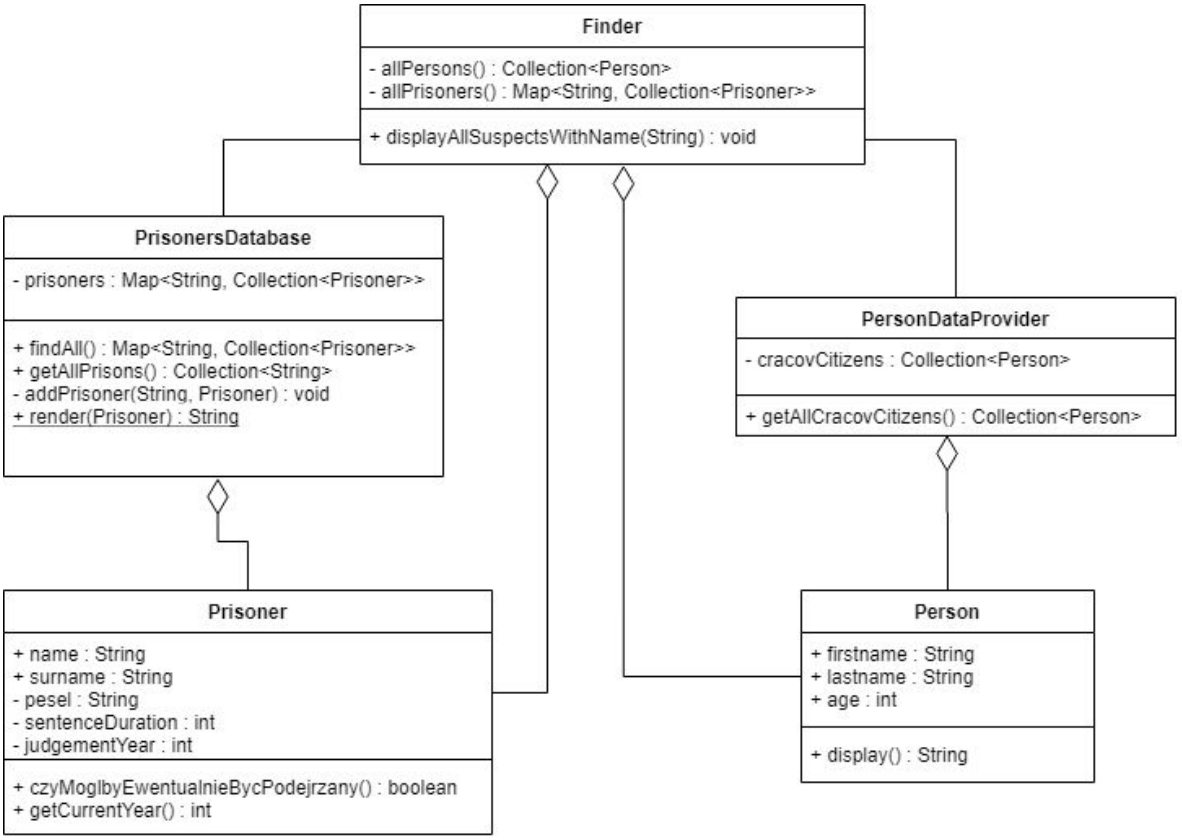
Projektowanie obiektowe
Raport - Refaktoryzacja

Autorzy:
Jakub Buziewicz
Szymon Rogus

Krok 1

Diagram na podstawie kodu:



Krok 2

Proponowane zmiany to:

1. Finder:
    - skasowanie pola “AllPersons“ oraz “allPrisoners“,
    - dodanie pola “personDataProvider” oraz “prisonersDatabase”,
    - zastąpienie starych konstruktorów jednym - zapamiętującym providerów podanych jako argumenty.
2. PrisonersDatabase:
    - zmienienie nazwy klasy na “PrisonerDataProvider”,
    - skasowanie metody “render”, ponieważ za to odpowiedzialna jest klasa “Prisoner”,
    - zmienienie nazwy metody “findAll” na “getPrisoners”,
    - zmienienie nazwy metody “getAllPrisons” na “getPrisons”,
    - przeniesienie dodawania przykładowych danych jako oddzielne metody “addExampleData”.
3. Prisoner:
    - ustawienie atrybutów na prywatne oraz dodanie getterów,
    - zmienienie nazwy pola “senteceDuration” na “sentenceTime”,
    - ustawienie na private metodę “getCurrentYear”,
    - zmienienie nazwy metody “czyMoglbyEwentualnieBycPodejrzany” na “isJailedNow”,
    - dodanie pola “age”, ponieważ nie każdy musi posiadać pesel,
    - dodanie metody “display”.
4. Person
    - zmienienie nazwy pola “firstname” na “name”,
    - zmienienie nazwy pola “lastname” na “surname”,
    - zmienienie nazwy getterów na “get<Atrybut>”.
5. PersonDataProvider
    - zmienienie nazwy klasy na “CitizenDataProvider”,
    - zmienienie nazwy gettera na “getCitizens”,
    - dodanie metody “addCitizen”,
    - przeniesienie dodawania przykładowych danych jako oddzielne metody “addExampleData” oraz wykorzystanie metody “addCitizen”, zamiast bezpośredniego dodawania do kolekcji.

Krok 3

Naszym wyborem tutaj jest klasa abstrakcyjna, ponieważ niektóre z metod możemy zdefiniować w klasie “Suspect”, ze względu na ich działanie, które jest takie samo. Jedną z takich metod jest metoda “display”. Dzięki przeniesieniu jej do klasy “Suspect”, możemy w klasie ”Finder” uogólnić metodę “displayAllSuspectsWithName”, w której podejrzani więźniowie oraz obywatele będą

trzymani na jednej liście. Warunek bycia podejrzanym przenieśliśmy do metod “canBeSuspect”, która jest metodą abstrakcyjną klasy Suspect.

Krok 4

W celu przeiterowania się po wszystkich elementach, będziemy potrzebować wzorca agregator. W tym celu stworzyliśmy interfejs “SuspectAggregate” oraz dodaliśmy metody “iterate”, która została zadeklarowana w interfejsie. W celu użycia takiej metody, potrzebujemy obudować klasy “PrisonerDataProvider” oraz “CitizenDataProvider” dekoratorem “Iterable<class name>”. Zapewnia on możliwość iterowania się bez zmiany DataProviderów. Dzięki temu metoda “findAllSuspectsWithName” w klasie “Finder” wygląda teraz dużo lepiej:

```
private ArrayList<Suspect> findAllSuspectsWithName(String name){
    ArrayList<Suspect> suspected = new ArrayList<~>();
    for (SuspectAggregate collection : collectionsToFind){
        for (Suspect suspect : collection) {
            if (suspect.canBeSuspect() && suspect.getName().equals(name))
                suspected.add(suspect);
            if (suspected.size() >= 10)
                return suspected;
        }
    }
    return suspected;
}
```

Krok 5

Aby odciążyć klasę “Finder” od odpowiedzialności modyfikowania nowych zbiorów danych, dodaliśmy nową klasę “CompositeAggregate”, która posiada kolekcję “suspectAggregates” - jest to kolekcja, która agreguje wszystkich podejrzanych (zarówno z klasy “CitizenDataProvider” jak i “PrisonerDataProvider”). W klasie tej umieszczona jest także metoda “findAllSuspectsWithName” - przeniesiona z klasy “Finder”. Używamy w niej iteratora aby dla zadanego parametru “name” zwrócić nową listę podejrzanych na podstawie dostępnej kolekcji agregatów:

```
public ArrayList<Suspect> findAllSuspectsWithName(String name){
    for (SuspectAggregate collection : this.suspectAggregates) {
        Iterator<Suspect> suspectedList = collection.iterator();

        while(suspectedList.hasNext()) {
            Suspect suspect = suspectedList.next();

            if (suspect.canBeSuspect() && suspect.getName().equals(name))
                this.suspectDataProvider.add(suspect);
            if (this.suspectDataProvider.size() >= 10)
                return this.suspectDataProvider;
        }
    }
    return this.suspectDataProvider;
}
```

Klasa “Finder” po dodaniu ”CompositeAggregate”:

```
private CompositeAggregate compositeAggregate;

public Finder(CompositeAggregate compositeAggregate) {
    this.compositeAggregate = compositeAggregate;
}

public void displayAllSuspectsWithName(String name) {
    ArrayList<Suspect> suspected = compositeAggregate.findAllSuspectsWithName(name);
    System.out.println("Znalazlem " + suspected.size() + " pasujacych podejrzanych!");
    for (Suspect s : suspected) {
        System.out.println(s.display());
    }
}
```

“Finder” korzysta z “CompositeAggregate” odpowiedzialnego za agregację oraz wyszukiwanie po agregatach.

Krok 6

Zastosowaliśmy wzorzec strategii do wybierania jak filtrujemy nasze dane. Stworzyliśmy interface “SearchStrategy”:

```
public interface SearchStrategy {

    Collection<Suspect> filter(Collection<Suspect> suspect);
}
```

Następnie dodaliśmy dwie klasy, które implementują ten interface - “AgeSearchStrategy” oraz “NameSearchStrategy”. Każda z nich filtruje kolekcję po zadanych warunkach. Następnie dodaliśmy klasę “CompositeSearchStrategy”, która agreguje strategie wyszukiwania:

```
public class CompositeSearchStrategy implements SearchStrategy {

    private Collection<SearchStrategy> searchStrategies = new ArrayList<>();
    private SearchStrategy actualStrategy;


    public void addStrategy(SearchStrategy newStrategy) {
        searchStrategies.add(newStrategy);
    }

    public void setStrategy(SearchStrategy strategy) {
        this.actualStrategy = strategy;
    }

    @Override
    public Collection<Suspect> filter(Collection<Suspect> suspect) {
        return actualStrategy.filter(suspect);
    }
}
```

Klasa ta umożliwia dodawanie strategii i ustawianie aktualnej. Wyszukiwanie odbywa się za pomocą aktualnie ustawionej strategii.

Klasa “Finder” korzysta teraz z dwóch kompozytów:

```java
public class Finder {

    private CompositeAggregate compositeAggregate;
    private CompositeSearchStrategy compositeSearchStrategy;

    public Finder(CompositeAggregate compositeAggregate, CompositeSearchStrategy searchStrategy) {
        this.compositeAggregate = compositeAggregate;
        this.compositeSearchStrategy = searchStrategy;
    }

    public void displayWithActualStrategy() {
        Collection<Suspect> suspected = compositeSearchStrategy.filter(compositeAggregate.getAllSuspects());
        for(Suspect suspect : suspected) {
            System.out.println(suspect.getName() + " " + suspect.getSurname());
        }
    }
}
```

Wyświetlanie osób podejrzanych następuje zgodnie z aktualną strategią przyjętą w “CompositeSearchStrategy”.

Opcjonalnie można stworzyć skomplikowane kryteria wyszukiwania poprzez nakładanie strategii. Zakładamy, że każda dodana strategia jest aktywna.
Tak wygląda wtedy klasa “CompositeSearchStrategy”:

```java
public class CompositeSearchStrategy implements SearchStrategy {

    private Collection<SearchStrategy> searchStrategies = new ArrayList<>();

    public void addStrategy(SearchStrategy newStrategy) { searchStrategies.add(newStrategy); }

    @Override
    public Collection<Suspect> filter(Collection<Suspect> suspect) {
        Collection<Suspect> suspectsFiltered = suspect.stream().filter(Suspect::canBeSuspect).collect(Collectors.toList());

        for(SearchStrategy searchStrategy : searchStrategies) {
            suspectsFiltered = searchStrategy.filter(suspectsFiltered);
        }

        return suspectsFiltered;
    }
}
```

Każda dodana strategia będzie uwzględniona podczas wyszukiwania w klasie Finder.

Osatecznie nasz diagram UML wygląda następująco: