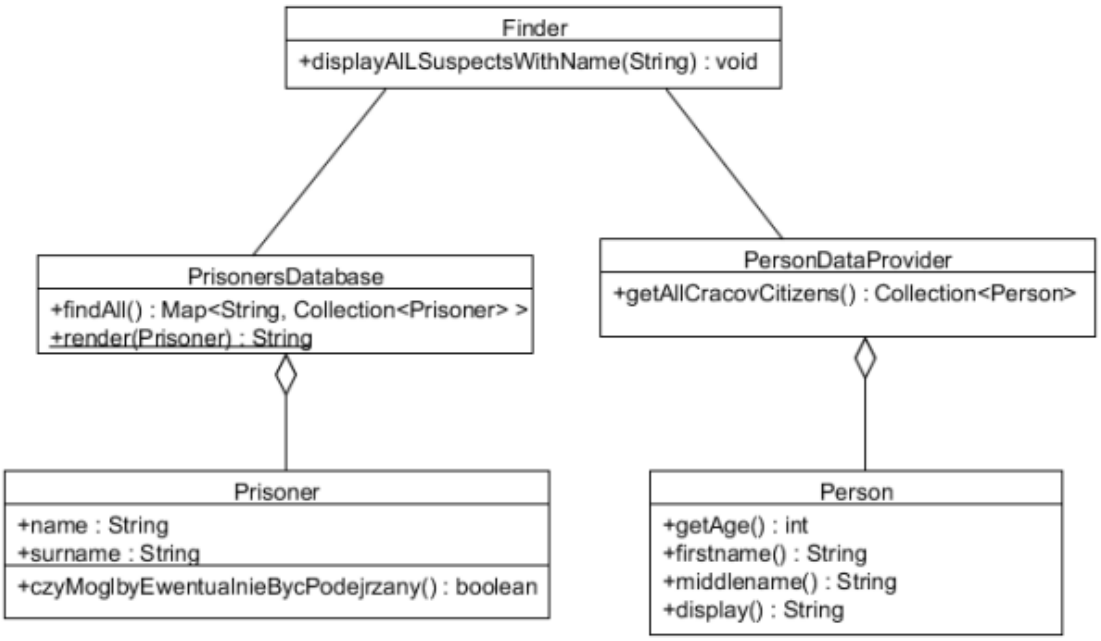
Refaktoryzacja

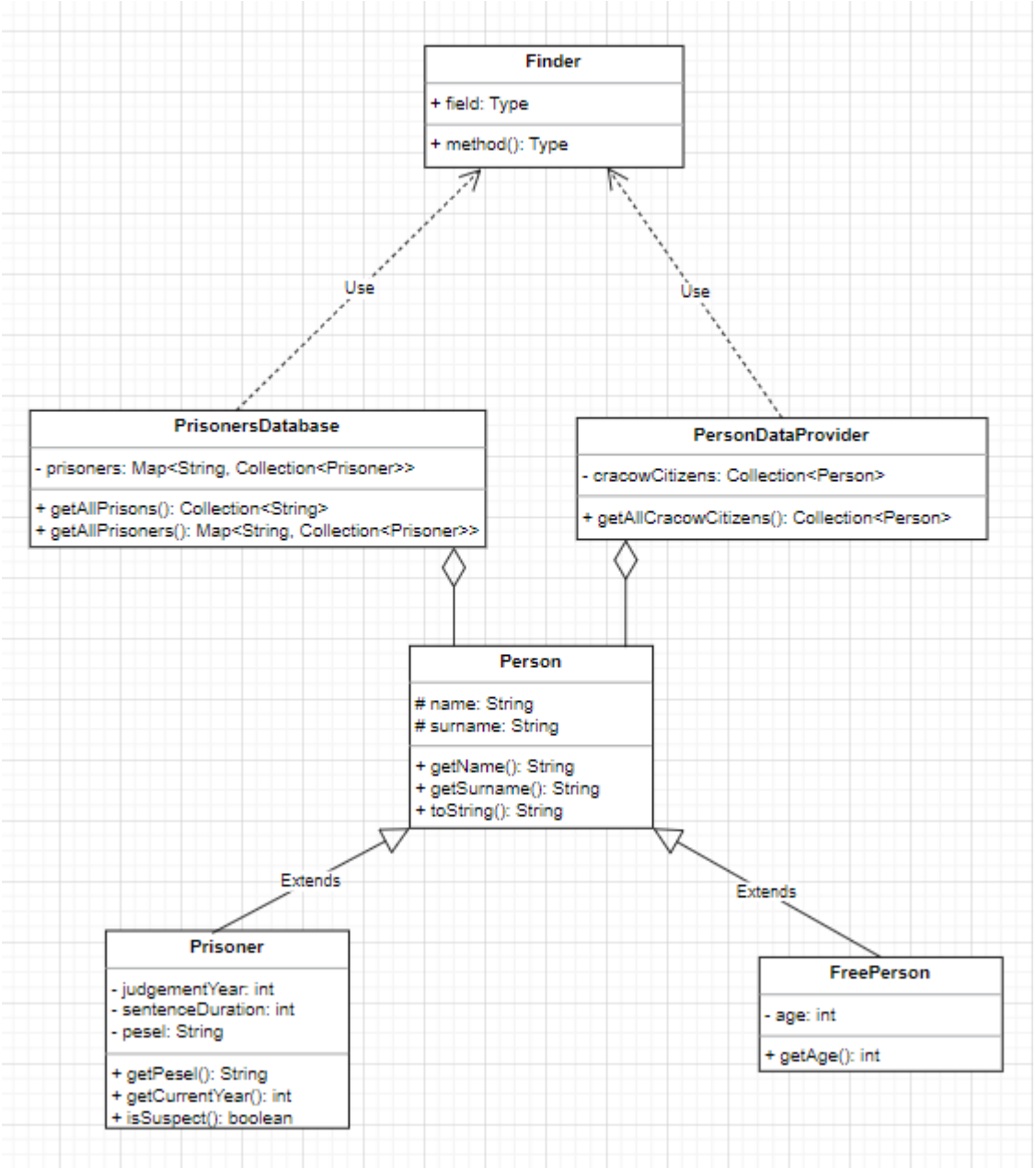
1. Narysuj diagram UML na podstawie analizy kodu. Porównaj otrzymany diagram z poniższym. Zaproponuj zmiany poprawiające jakość i czytelność kodu źródłowego (np. nazewnictwo i widoczność metod, hierarchia klas).

   Początkowy diagram UML:



   Proponowane zmiany:
   - zmiana nazw klas, atrybutów i metod na jeden język (najlepiej angielski),
   - zmiana dostępności atrybutów na prywatny,
   - wprowadzenie w diagramie nazw atrybutów oraz najpotrzebniejszych metod,
   - użycie camel case'a
   - generalizacja klas Prisoner oraz Person,
   - poprawa zależności między klasami.

   Początkowa wersja diagramu UML po wprowadzeniu zmian:



2. Wprowadź poprawę podstawowych błędów w kodzie, takich jak:

    1. publiczne pola zamiast metod dostępowych.

        - zmieniono pola klasy `Prisoner` na prywatne:

```
private final String name;
private final String surname;
```

- użyto getterów w miejscach, gdzie klasa `PrisonersDatabase` korzystała bezpośrednio z tych pól:

```
public static String render(Prisoner prisoner) {
    return prisoner.getName() + " " + prisoner.getSurname();
}
```

2. statyczne metody w złych miejscach.

- usunięto statyczną metodę z klasy `PrisonersDatabase`.

```
public static String render(Prisoner prisoner) {
    return prisoner.getName() + " " + prisoner.getSurname();
}
```

- dodano do klasy `Prisoner` metodę odpowiedzialną za wyświetlanie imienia i nazwiska osoby (nazwa identyczna jak w klasie `Person`):

```
public String display() {
    return name + " " + surname;
}
```

3. zbyt długie i enigmatyczne nazwy metod. Skorzystaj z narzędzi do refaktoryzacji udostępnianych przez środowisko programistyczne.

- zmieniono nazwę metody `czyMoglbyEwentualnieBycPodejrzany()` w klasie `Prisoner` na bardziej przystępną i trzymającą się konwencji angielskiej:

```
public boolean isJailedNow() {
    return judgementYear + sentenceDuration >= getCurrentYear();
}
```

Dodatkowe modyfikacje:

- zmiana nazw zmiennych tak, aby nie było literówek,
- zmiana nazw atrybutów w klasie `Person` z `firstname` na `name` oraz z `lastname` na `surname` i przy okazji zmiana nazw getterów,
- dodano “final” do atrybutów klasy `Person`,

- zmieniono nazwę klasy z `Person` na `CracowCitizen`.

3. Zaproponuj generalizację klas Person i Prisoner. Uzasadnij użycie interfejsu, klasy abstrakcyjnej lub interfejsu z metodami domyślnymi (Java 8). Zastanów się jakie metody można dodać do klasy Suspect, aby uogólnić klasę Finder.

   Do realizacji tego zadania wybiorę klasę abstrakcyjną, gdyż tylko ona może zawierać niestatyczne atrybuty.

- dodano klasę abstrakcyjną `Suspect`:

```
package pl.edu.agh.to.lab4;

public abstract class Suspect {

    protected final String name;
    protected final String surname;

    public Suspect(String name, String surname) {
        this.name = name;
        this.surname = surname;
    }

    public String getName() {
        return name;
    }

    public String getSurname() {
        return surname;
    }

    public String display() {
        return name + " " + surname;
    }
}
```

- powtarzające się atrybuty oraz metody usunięto z klas po niej dziedziczących (`Prisoner`, `CracowCitizen`)

- aby uogólnić klasę `Finder` do klasy `Suspect` dodano metodę abstrakcyjną:

```
public abstract boolean canBeAccused();
```

- jej implementacje pozostawiono klasom pochodnym:

  Klasa `Prisoner`:

```
public boolean isJailedNow() {
    return judgementYear + sentenceDuration >= getCurrentYear();
}


@Override
public boolean canBeAccused() {
    return !isJailedNow();
}
```

  Klasa `CracowCitizen`:

```
@Override
public boolean canBeAccused() {
    return this.age >= 18;
}
```

- dodano do klasy `Finder` prywatną metodę i od teraz za jej pomocą będziemy szukać nowych podejrzanych:

```
private boolean isAppropriateSuspect(String name, Suspect suspect) {
    return suspect.canBeAccused() && suspect.getName().equals(name);
}
```

- kolekcja, do której będziemy dodawać nowych podejrzanych, także uległa zmianie, ponieważ usunięto z metody `displayAllSuspectsWithName(String name)` dwie osobne kolekcje do przechowywania więźniów oraz pozostałych mieszkańców Krakowa. Zastąpiono je jedną uniwersalną kolekcją:

```
ArrayList<Suspect> suspects = new ArrayList<>();
```

Pozostałe zmiany:

- w klasie `Finder` zmieniono nazwy kilku zmiennych na bardziej zrozumiałe:

```
int numberOfSuspects = suspectedPrisoners.size() +
suspectedCracowCitizens.size();
System.out.println("Znalazlem " + numberOfSuspects + " pasujacych
podejrzanych!");

for (Prisoner prisoner : suspectedPrisoners) {
    System.out.println(prisoner.display());
}

for (CracowCitizen cracowCitizen : suspectedCracowCitizens) {
    System.out.println(cracowCitizen.display());
}
```

4. Kolejnym krokiem jest wprowadzenie generalizacji do klas dostarczających dane do systemu. Ważne jest w tym przypadku, że nie mamy wpływu na to jak one wyglądają (przyjmujemy, że implementacje tych klas są dostarczane z zewnątrz). Nie możemy więc zmienić metod występujących w ich interfejsach. Możemy jedynie metody dodać lub (najlepiej) udekorować wzorcem Dekorator.
   Dodatkowym problemem jest odmienna struktura danych zwracana przez obu dostawców. Jeden zwraca Collection<CracovCitizen>, drugi zaś Map<String, Collection<Prisoner». Ponadto, ponieważ zbiory danych są duże, nie chcemy tworzyć nowej kolekcji tylko po to by ją zwrócić. Rozwiązaniem może być Iterator. W przypadku PersonDataProvider dostarcza go interfejs Collection. Dla PrisonerDatabase musimy zaimplementować swój iterator, implementujący interfejs Iterator z Javy, który dostarczy interesujące nas elementy w tej samej formie.

- na samym początku zmieniono nazwy klas dostarczających dane do systemu:

  1. `PersonDataProvider` na `PeopleDataProvider`
  2. `PrisonersDatabase` na `PrisonersDataProvider`

- dodano interfejs `SuspectAggregate`, który będą implementować klasy dostarczające dane do systemu (`PrisonersDataProvider`, `PeopleDataProvider`)

```
package pl.edu.agh.to.lab4;

import java.util.Iterator;

public interface SuspectAggregate {
    Iterator<? extends Suspect> iterator();
```

```
}
```

- implementacja metody w klasie `PeopleDataProvider`:

```
@Override
public Iterator<? extends Suspect> iterator() {
    return this.cracowCitizens.iterator();
}
```

- implementacja metody w klasie `PrisonersDataProvider`:

```
@Override
public Iterator<Prisoner> iterator() {
    Iterator<Prisoner> prisonerIterator = prisoners.values().stream()
            .flatMap(Collection::stream)
            .collect(Collectors.toList())
            .iterator();
    return new FlatIterator(prisonerIterator);
```

Metoda ta zwraca iterator dla wszystkich więźniów znajdujących się w kolekcji `prisoners`.

- kod klasy `FlatIterator`:

```
package pl.edu.agh.to.lab4;

import java.util.Iterator;

public class FlatIterator implements Iterator<Prisoner> {
    private final Iterator<Prisoner> prisonerIterator;

    public FlatIterator(Iterator<Prisoner> prisonerIterator) {
        this.prisonerIterator = prisonerIterator;
    }

    @Override
    public boolean hasNext() {
        return prisonerIterator.hasNext();
    }

    @Override
```

```
    public Prisoner next() {
        return prisonerIterator.next();
    }
}
```

- klasa Finder wygląda teraz następująco:

```
package pl.edu.agh.to.lab4;

import java.util.ArrayList;
import java.util.Iterator;

public class Finder {
    private final SuspectAggregate allCracowCitizens;
    private final SuspectAggregate allPrisoners;

    public Finder(PeopleDataProvider peopleDataProvider,
PrisonersDataProvider prisonersDataProvider) {
        this.allCracowCitizens = peopleDataProvider;
        this.allPrisoners = prisonersDataProvider;
    }

    public void displayAllSuspectsWithName(String name) {
        ArrayList<Suspect> suspects = new ArrayList<>();

        Iterator<? extends Suspect> prisonersIterator =
allPrisoners.iterator();
        Iterator<? extends Suspect> cracowCitizensIterator =
allCracowCitizens.iterator();

        while (prisonersIterator.hasNext() && suspects.size() < 10) {
            Suspect suspect = prisonersIterator.next();
            if (isAppropriateSuspect(name, suspect))
suspects.add(suspect);
        }

        while (cracowCitizensIterator.hasNext() && suspects.size() < 10)
{
            Suspect suspect = cracowCitizensIterator.next();
            if (isAppropriateSuspect(name, suspect))
suspects.add(suspect);
```

```
        }

        int numberOfSuspects = suspects.size();
        System.out.println("Znalazlem " + numberOfSuspects + "
pasujacych podejrzanych!");

        suspects.forEach(suspect ->
System.out.println(suspect.display()));
    }

    private boolean isAppropriateSuspect(String name, Suspect suspect) {
        return suspect.canBeAccused() && suspect.getName().equals(name);
    }
}
```

Widzimy, iż poprzez dodanie wspólnego interfejsu kod klasy Finder uległ znacznemu ulepszeniu oraz klasy dostarczające dane do systemu uległy znacznej generalizacji.

5. Czy problem dodania nowego zbioru danych został rozwiązany? Czy wystarczy dostosować nową implementację do SuspectAggregate i wszystko będzie działać? Niestety, Finder nadal musi wiedzieć, ile jest zbiorów danych, więc trzeba zmienić jego implementację, aby go nie modyfikować przy nowych zbiorach danych. Obecnie, po tym że Finder jest połączony bezpośrednio z instancjami SuspectAggregate widać, że wewnątrz ciała klasy Finder są zagnieżdżone pętle. Należy więc dodać klasę pośrednią pomiędzy agregatami a finderem, której odpowiedzialnością będzie agregacja wszystkich osób ze wszystkich zbiorów danych i przekazanie ich do iteratora. Tym samym ściągamy z findera kolejną odpowiedzialność i jest tam coraz mniej kodu. W jaki sposób przekazać te dane? Iteratorem, który jednocześnie agreguje wszystkie instancje z całego systemu, wykorzystując wzorzec Composite.

- stworzono klasę `CompositeAggregate`, która agreguje wszystkie osoby ze wszystkich zbiorów danych w aplikacji i przekazuje je do iteratora:

```
package pl.edu.agh.to.lab4;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
import java.util.List;

public class CompositeAggregate implements SuspectAggregate {
```

```
    private final List<SuspectAggregate> suspectAggregates;

    public CompositeAggregate(List<SuspectAggregate> suspectAggregates)
{
        this.suspectAggregates = suspectAggregates;
    }

    @Override
    public Iterator<? extends Suspect> iterator() {
        Collection<Suspect> suspects = new ArrayList<>();
        for (SuspectAggregate suspectAggregate: suspectAggregates) {
            Iterator<? extends Suspect> suspectIterator =
suspectAggregate.iterator();
            while (suspectIterator.hasNext())
suspects.add(suspectIterator.next());
        }
        return suspects.iterator();
    }
}
```

- w klasie Finder zmieniono atrybuty oraz parametry konstruktora:

```
private final CompositeAggregate compositeAggregate;

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

Modyfikacji uległa także dalsza część kodu (mamy jedno wywołanie iteratora dla atrybutu klasy oraz jedną pętlę zamiast dwóch)

- kod klasy klienta:

```
package pl.edu.agh.to.lab4;

import java.util.ArrayList;

public class Application {
```

```
    public static void main(String[] args) {
        ArrayList<SuspectAggregate> suspectAggregates = new
ArrayList<>();

        suspectAggregates.add(new PrisonersDataProvider());
        suspectAggregates.add(new PeopleDataProvider());

        CompositeAggregate allSuspects = new
CompositeAggregate(suspectAggregates);
        Finder suspects = new Finder(allSuspects);
        suspects.displayAllSuspectsWithName("Janusz");
    }
}
```

- dostosowano także klasę testującą pod zmodyfikowany kod

6. Ostatnim krokiem jest wsparcie możliwości łatwego dodawania warunków wyszukiwania. Najprostsze byłoby dodawanie kolejnych metod szukających do findera, ale jest to rozwiązania bardzo mało rozszerzalne. Lepszym rozwiązaniem będzie zastosowanie strategii szukających (zwanych też predykatami). Mają bardzo prostą implementację i, co więcej, można je łączyć ze sobą z wykorzystaniem wzorca Composite, by otrzymać złożone kryteria wyszukiwania. Wprowadzając ten wzorzec proszę dodać wyszukiwanie po wieku.

- stworzono interfejs, który będą implementować wszystkie klasy wyszukujące podejrzanych po określonych kryteriach:

```
package pl.edu.agh.to.lab4;

public interface SearchStrategy {
    boolean filter(Suspect suspect);
}
```

- klasa wyszukująca podejrzanych po wieku:

```
package pl.edu.agh.to.lab4;

public class AgeSearchStrategy implements SearchStrategy {
```

```
    private final int age;

    public AgeSearchStrategy(int age) {
        this.age = age;
    }

    @Override
    public boolean filter(Suspect suspect) {
        return suspect.getAge() == age;
    }
}
```

- klasa wyszukująca podejrzanych po imieniu:

```
package pl.edu.agh.to.lab4;

public class NameSearchStrategy implements SearchStrategy {
    private final String name;

    public NameSearchStrategy(String name) {
        this.name = name;
    }

    @Override
    public boolean filter(Suspect suspect) {
        return suspect.getName().equals(this.name);
    }
}
```

- klasa łącząca wiele kryteriów wyszukiwania:

```
package pl.edu.agh.to.lab4;

import java.util.ArrayList;
import java.util.Collection;

public class CompositeSearchStrategy implements SearchStrategy {

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

    public void addStrategy(SearchStrategy searchStrategy) {
```

```
        searchStrategies.add(searchStrategy);
    }

    @Override
    public boolean filter(Suspect suspect) {
        for (SearchStrategy searchStrategy : searchStrategies)
            if (!searchStrategy.filter(suspect))
                return false;
        return true;
    }
}
```

- aby umożliwić wyszukiwanie podejrzanych po wieku do klasy `Suspect` dodano chronione pole `age` i podczas tworzenia bazy więźniów każdego uzupełniono o wiek:

```
protected int age;

public Suspect(String name, String surname, int age) {
    this.name = name;
    this.surname = surname;
    this.age = age;
}
```

- stworzono klasę `Student` dziedziczącą po klasie `Suspect`:

```
package pl.edu.agh.to.lab4;

public class Student extends Suspect {
    private final String index;

    public Student(String name, String surname, int age, String index) {
        super(name, surname, age);
        this.index = index;
    }
}
```

- oraz klasę dostarczającą bazę studentów do systemu:

```
package pl.edu.agh.to.lab4;
```

```java
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

public class StudentsDataProvider implements SuspectAggregate {

    private final Collection<Student> students = new ArrayList<>();

    public StudentsDataProvider() {
        students.add(new Student("Jan", "Student", 24, "305123"));
        students.add(new Student("Janusz", "Pilny", 30, "305124"));
        students.add(new Student("Janusz", "Len", 30, "305125"));
        students.add(new Student("Kasia", "Imprezowa", 21, "305126"));
        students.add(new Student("Piotr", "Nowak", 22, "305127"));
        students.add(new Student("Tomek", "Kwiatkowski", 23, "305128"));
        students.add(new Student("Janusz", "Kwiat", 24, "305129"));
        students.add(new Student("Alicja", "Czar", 25, "305133"));
        students.add(new Student("Janusz", "Ptak", 22, "305143"));
        students.add(new Student("Pawel", "Gawlowski", 21, "305153"));
        students.add(new Student("Krzysztof", "Kot", 20, "305163"));
    }

    public Collection<Student> getAllStudentsCitizens() {
        return this.students;
    }

    @Override
    public Iterator<? extends Suspect> iterator() {
        return students.iterator();
    }
}
```

- na koniec zmodyfikowano klasy testowe pod wprowadzoną refaktoryzację