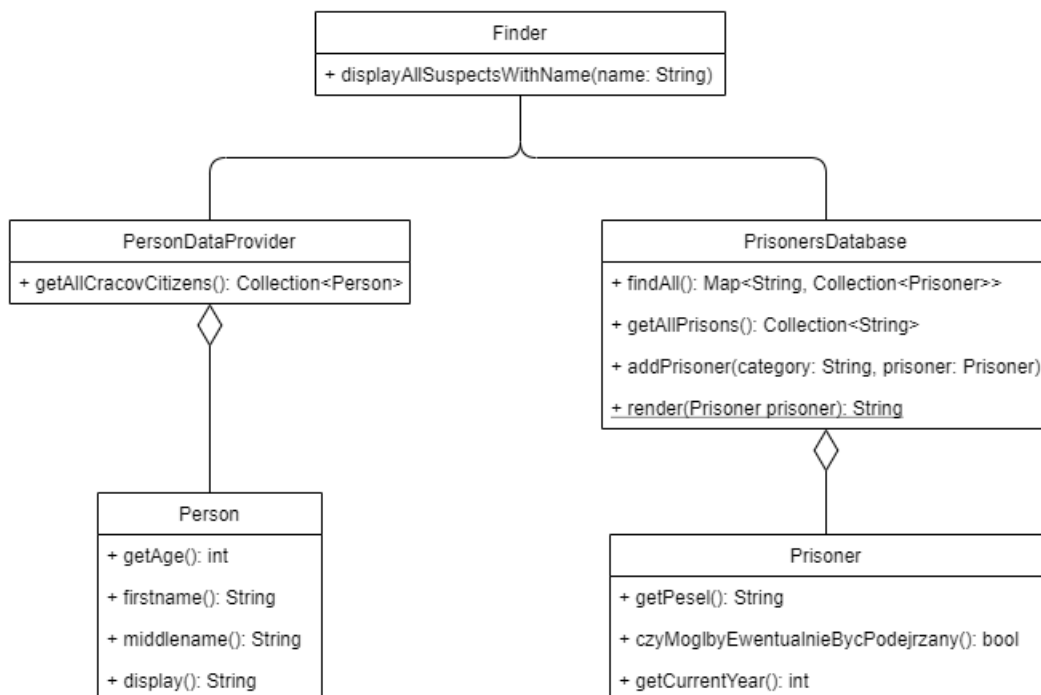


# Projektowanie Obiektowe

Albert Gierlach

## Krok 1

Na podstawie obecnego kodu stworzyłem diagram UML. Ujednoliciłem diagramy klas – każda posiada tylko zestaw metod, bez pól publicznych



Rysunek 1: Diagram UML

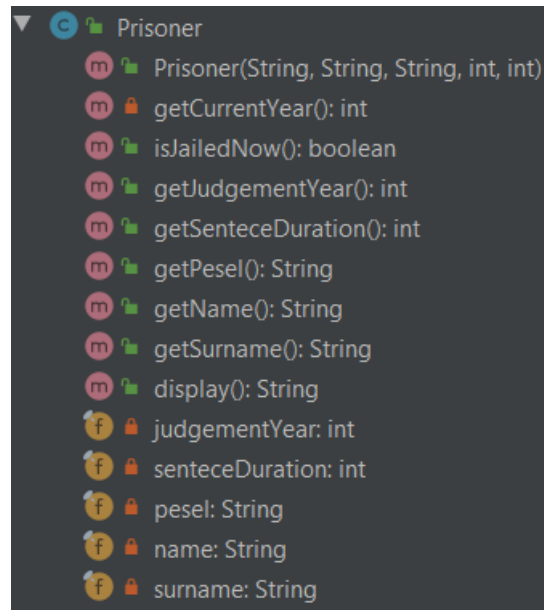
Zmiany poprawiające czytelność i jakość kodu to:

- wydzielenie wspólnej części klas Person i Prisoner (lub Prisoner mógłby dziedziczyć z Person)
- zmiana nazw metod na angielskie (lub polskie, bardziej chodzi o spójność i konsekwentne stosowanie, chociaż preferowane są angielskie nazwy)
- w klasie Person zamiast metody display można przeciążyć metodę toString()
- zmienne wewnątrz klas powinny być prywatne

Osobiście uważam, że długość nazw metod jest w porządku, metody powinny mieć nazwy mówiące o tym co wykonuje dana metoda. W obecnym czasie środowiska programistyczne pozwalają na dopełnianie nazw, więc pisanie długich nazw metod nie jest tak uciążliwe.

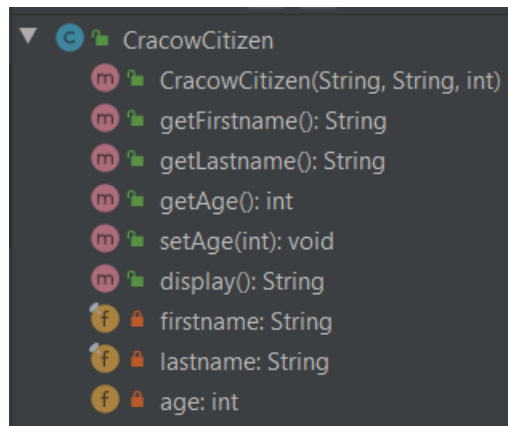
## Krok 2

Klasę Prisoner poprawiłem w następujący sposób: zmieniłem zmienne klasowe na prywatne, dodałem metody dostępowe, zmieniłem nazwy niektórych metod.



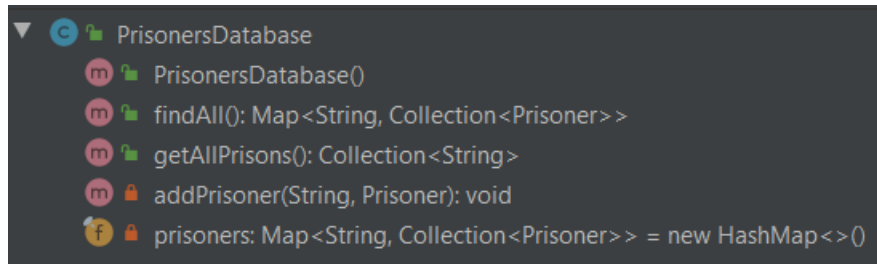
Rysunek 2: Klasa Prisoner

Klasa Person została przemianowana na CracowCitizen:



Rysunek 3: Klasa CracowCitizen

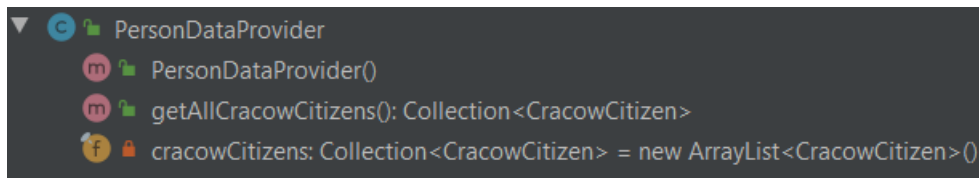
Klasa PrisonersDatabase pozostała prawie bez zmian – usunąłem tylko metodę statyczną pobierającą dane Prisonera. Zamiast tego możemy użyć metody display() na obiekcie typu Prisoner.



```
PrisonersDatabase
  PrisonersDatabase()
  findAll(): Map<String, Collection<Prisoner>>
  getAllPrisons(): Collection<String>
  addPrisoner(String, Prisoner): void
  prisoners: Map<String, Collection<Prisoner>> = new HashMap<>()
```

Rysunek 4: Klasa PrisonersDatabase

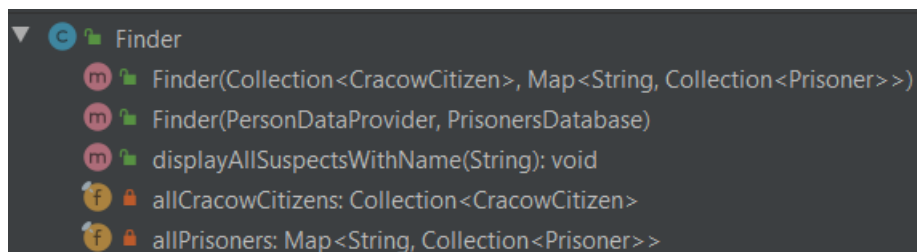
Klasa PersonDataProvider nie została zmieniona w wielkim stopniu, jedynie dostosowałem zmienione nazwy typów.



```
PersonDataProvider
  PersonDataProvider()
  getAllCracowCitizens(): Collection<CracowCitizen>
  cracowCitizens: Collection<CracowCitizen> = new ArrayList<CracowCitizen>()
```

Rysunek 5: Klasa PersonDataProvider

Klasa Finder pozostała bez zmian.

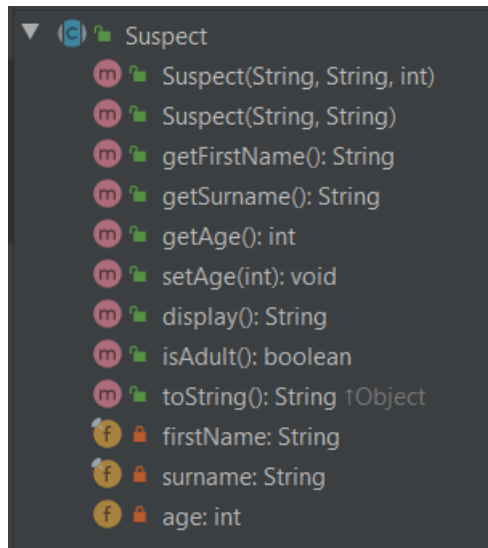


```
Finder
  Finder(Collection<CracowCitizen>, Map<String, Collection<Prisoner>>)
  Finder(PersonDataProvider, PrisonersDatabase)
  displayAllSuspectsWithName(String): void
  allCracowCitizens: Collection<CracowCitizen>
  allPrisoners: Map<String, Collection<Prisoner>>
```

Rysunek 6: Klasa Finder

### Krok 3

Dokonałem generalizacji klasy Prisoner i CracowCitizen. Wydzieliłem wspólne części do klasy abstrakcyjnej Suspect. Klasy prezentują się następująco:



Rysunek 7: Klasa Suspect

```
public class CracowCitizen extends Suspect {  
    public CracowCitizen(String firstname, String lastname, int age) {  
        super(firstname, lastname, age);  
    }  
}
```

Rysunek 8: Klasa CracowCitizen

```

public class Prisoner extends Suspect{
    private final int judgementYear;
    private final int senteceDuration;
    private final String pesel;

    public Prisoner(String name, String surname, String pesel, int judgementYear, int sentenceDuration) {
        super(name, surname);
        this.pesel = pesel;
        this.judgementYear = judgementYear;
        this.senteceDuration = sentenceDuration;
    }

    private int getCurrentYear() {
        return Calendar.getInstance().get(Calendar.YEAR);
    }

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

    public int getJudgementYear() {
        return judgementYear;
    }

    public int getSenteceDuration() {
        return senteceDuration;
    }

    public String getPesel() {
        return pesel;
    }
}

```

Rysunek 9: Klasa Prisoner

Do klasy Suspect dodałem metodę sprawdzającą czy dana osoba może być podejrzana. W klasach dziedziczących odpowiednio zaimplementowałem tą metodę. Klasy wyglądają teraz następująco:

```

public abstract class Suspect {
    private final String firstName;
    private final String surname;
    private int age;

    public Suspect(String f, String l, int age){
        this.firstName = f;
        this.surname = l;
        this.age = age;
    }

    public Suspect(String f, String l){
        this(f, l, age: 0);
    }

    public boolean canBeAccused(String n){
        return this.firstName.equals(n);
    }
}

```

Rysunek 10: Metoda canBeAccused w klasie Suspect

```

public class CracowCitizen extends Suspect {
    public CracowCitizen(String firstname, String lastname, int age) {
        super(firstname, lastname, age);
    }

    @Override
    public boolean canBeAccused(String n) {
        return super.canBeAccused(n) && isAdult();
    }
}

```

Rysunek 11: Metoda canBeAccused w klasie CracowCitizen

```

public class Prisoner extends Suspect{
    private final int judgementYear;
    private final int senceDuration;
    private final String pesel;

    public Prisoner(String name, String surname, String pesel, i
        super(name, surname);
        this.pesel = pesel;
        this.judgementYear = judgementYear;
        this.senceDuration = sentenceDuration;
    }

    @Override
    public boolean canBeAccussed(String n) {
        return super.canBeAccussed(n) && !isJailedNow();
    }
}

```

Rysunek 12: Metoda canBeAccused w klasie Prisoner

Uproszczona metoda klasy Finder wygląda teraz następująco:

```

public void displayAllSuspectsWithName(String name) {
    ArrayList<Prisoner> suspectedPrisoners = new ArrayList<>();
    ArrayList<CracowCitizen> suspectedCracowCitizens = new ArrayList<>();

    for (Collection<Prisoner> prisonerCollection : allPrisoners.values()) {
        for (Prisoner prisoner : prisonerCollection) {
            if (prisoner.canBeAccussed(name)) {
                suspectedPrisoners.add(prisoner);
            }
            if (suspectedPrisoners.size() >= 10) {
                break;
            }
        }
        if (suspectedPrisoners.size() >= 10) {
            break;
        }
    }

    if (suspectedPrisoners.size() < 10) {
        for (CracowCitizen cracowCitizen : allCracowCitizens) {
            if (cracowCitizen.canBeAccussed(name)) {
                suspectedCracowCitizens.add(cracowCitizen);
            }
            if (suspectedPrisoners.size() + suspectedCracowCitizens.size() >= 10) {
                break;
            }
        }
    }
}

```

Rysunek 13: Zmieniona metoda klasy Finder

## Krok 4

Na początku stworzyłem uogólnioną klasę FlatIterator. Klasa ta ma za zadanie przechowywać iteratory do kolejnych kolekcji przechowywanych w mapie Map<String, Collection<Prisoner>> prisoners. Strategia iterowania polega na przechodzeniu przez kolejne iteratory podkolekcji aż do wyczerpania elementów. Klasa prezentuje się następująco:

```
public class FlatIterator<Type> implements Iterator<Type> {
    private final Iterator<Iterator<Type>> suspectIterators;
    private Iterator<Type> currentIterator;

    public FlatIterator(Collection<Iterator<Type>> suspectCollection) {
        this.suspectIterators = suspectCollection.iterator();
        if(!suspectCollection.isEmpty()){
            this.currentIterator = suspectIterators.next();
        }
    }

    @Override
    public boolean hasNext() {
        if(currentIterator == null){
            return false;
        }

        if(currentIterator.hasNext()){
            return true;
        }

        if(suspectIterators.hasNext()){
            return (currentIterator = suspectIterators.next()).hasNext();
        }

        return false;
    }

    @Override
    public Type next() {
        return currentIterator.next();
    }
}
```

Rysunek 14: Klasa FlatIterator



Później stworzyłem oraz zaimplementowałem interfejs SuspectAggregate:

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

Rysunek 15: Interfejs SuspectAggregate

```
public class PrisonersDatabase implements SuspectAggregate {  
  
    private final Map<String, Collection<Prisoner>> prisoners = new HashMap<>();  
  
    public PrisonersDatabase() {...}  
  
    public Map<String, Collection<Prisoner>> findAll() {...}  
  
    @Override  
    public Iterator<? extends Suspect> iterator() {  
        return new FlatIterator<>(prisoners.values().stream().map(Collection::iterator).collect(Collectors.toList()));  
    };  
}
```

Rysunek 16: Implementacja interfejsu SuspectAggregate w klasie PrisonersDatabase

```
public class PersonDataProvider implements SuspectAggregate {  
  
    private final Collection<CracowCitizen> cracowCitizens = new ArrayList<>();  
  
    public PersonDataProvider() {...}  
  
    public Collection<CracowCitizen> getAllCracowCitizens() {...}  
  
    @Override  
    public Iterator<? extends Suspect> iterator() {  
        return cracowCitizens.iterator();  
    }  
}
```

Rysunek 17: Implementacja interfejsu SuspectAggregate w klasie PersonDataProvider

## Krok 5

Stworzyłem klasę CompositeAggregate – wykorzystuje ona podobną implementację iteratora co klasa Prisoner.

```
public class CompositeAggregate implements SuspectAggregate{
    private final List<SuspectAggregate> aggregateList;

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

    @Override
    public Iterator<? extends Suspect> iterator() {
        return new FlatIterator<Suspect>() {
            @Override
            public Iterator<Suspect> iterator() {
                return aggregateList.stream().map(a -> a.iterator()).collect(Collectors.toList());
            }
        };
    }
}
```

Rysunek 18: Klasa agregująca źródła danych

Później zmieniłem implementację konstruktorów klasy Finder tak, aby korzystały z CompositeAggregate:

```
public class Finder {
    private final CompositeAggregate compositeAggregate;

    public Finder(Collection<CracowCitizen> allCracowCitizens, Map<String, Collection<Prisoner>> allPrisoners) {
        SuspectAggregate p = () -> new FlatIterator<>() {
            @Override
            public Iterator<Suspect> iterator() {
                return allPrisoners.values().stream().map(Collection::iterator).collect(Collectors.toList());
            }
        };

        this.compositeAggregate = new CompositeAggregate(List.of(p, allCracowCitizens::iterator));
    }

    public Finder(PersonDataProvider personDataProvider, PrisonersDatabase prisonersDatabase) {
        this.compositeAggregate = new CompositeAggregate(List.of(personDataProvider, prisonersDatabase));
    }
}
```

Rysunek 19: Klasa Finder korzystająca z funkcjonalności CompositeAggregate

Oraz w celu przetestowania zrefaktoryzowałem metodę displayAllSuspectsWithName i przetestowałem zmiany.

```

public void displayAllSuspectsWithName(String name) {
    var res = new ArrayList<Suspect>();
    var iterator = compositeAggregate.iterator();
    while(iterator.hasNext()){
        Suspect s = iterator.next();
        if(s.canBeAccused(name)){
            res.add(s);
        }

        if(res.size() >= 10){
            break;
        }
    }

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

    for (Suspect n : res) {
        System.out.println(n.display());
    }
}

```

Rysunek 20: Uproszczona metoda szukająca w klasie Finder

```

"C:\Program Files\Java\jdk-13\bin\java
Znalazlem 5 pasujacych podejrzanych!
Janusz Krakowski
Janusz Programista
Janusz Złowieszczy
Janusz Podejrzany
Janusz Zamkniety

```

Rysunek 21: Dane wynikowe po wprowadzonych zmianach

## Krok 6

Na początku stworzyłem interfejs `SearchStrategy`, a następnie klasę `CompositeSearchStrategy`, która będzie łączyć złożone kryteria wyszukiwania.

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

Rysunek 22: Interfejs `SearchStrategy`

```
public class CompositeSearchStrategy implements SearchStrategy {  
    List<SearchStrategy> searchStrategies;  
  
    public CompositeSearchStrategy() {  
        this.searchStrategies = new LinkedList<>();  
    }  
  
    public void addStrategy(SearchStrategy strategy){  
        this.searchStrategies.add(strategy);  
    }  
  
    @Override  
    public boolean filter(Suspect s) {  
        return searchStrategies  
            .stream()  
            .allMatch(searchStrategy -> searchStrategy.filter(s));  
    }  
}
```

Rysunek 23: Klasa agregująca strategie poszukiwań

Następnie stworzyłem trzy kryteria wyszukiwania – `AgeSearchStrategy`, `NameSearchStrategy`, `NotInJailSearchStrategy`.

```
public class NotInJailSearchStrategy implements SearchStrategy {  
    @Override  
    public boolean filter(Suspect s) {  
        boolean isPrisoner = (s instanceof Prisoner);  
        return !isPrisoner || !((Prisoner) s).isJailedNow();  
    }  
}
```

Rysunek 24: Strategia wyszukiwania osób niebędących w więzieniu

```

public class NameSearchStrategy implements SearchStrategy{
    private final String name;

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

    @Override
    public boolean filter(Suspect s) {
        return s.getFirstName().equals(name);
    }
}

```

Rysunek 25: Strategia wyszukiwania osób po imieniu

```

public class AgeSearchStrategy implements SearchStrategy {
    private final int age;

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

    @Override
    public boolean filter(Suspect s) {
        boolean isCitizen = (s instanceof CracowCitizen);
        return !isCitizen || s.getAge() >= age;
    }
}

```

Rysunek 26: Strategia wyszukiwania osób z wiekiem większym od podanego

Metoda szukająca została uproszczona do takiego stanu:

```

public void displayAllSuspectsWithName(String name) {
    var strategy = new CompositeSearchStrategy();
    strategy.addStrategy(new NameSearchStrategy(name));
    strategy.addStrategy(new AgeSearchStrategy(18));
    strategy.addStrategy(new NotInJailSearchStrategy());

    var res = new ArrayList<Suspect>();
    var iterator = compositeAggregate.iterator();
    while(iterator.hasNext()){
        Suspect s = iterator.next();
        if(strategy.filter(s)){
            res.add(s);
            if(res.size() > 10){
                break;
            }
        }
    }

    System.out.println("Znalazłem " + res.size() + " pasujących podejrzanych!");
    res.forEach(s -> System.out.println(s.display()));
}

```

Rysunek 27: Uproszczona metoda szukająca osób o zadanych kryteriach

Jak widać metoda szukająca uległa znacznemu uproszczeniu. Po uruchomieniu otrzymałem poprawne dane:

```

Znalazłem 5 pasujących podejrzanych!
Janusz Krakowski
Janusz Programista
Janusz Złowieszczy
Janusz Podejrzany
Janusz Zamknięty

```

Rysunek 28: Wynik działania programu po wprowadzonych zmianach

Testy także pokazują poprawność implementacji:

All in lab4 x			Tests passed: 7 of 7 tests – 49 ms	
✓	<default package>	49 ms	"C:\Program Files\Ja	
▶	✓ FinderTest	48 ms	Process finished wit	
▶	✓ PrisonerDatabaseTest	1 ms		
▼	✓ PrisonerTest	0 ms		
✓	testPrisonerHasBeenReleasedFromJail	0 ms		
✓	testPrisonerIsInJail	0 ms		

Rysunek 29: Wyniki testów po wprowadzonych zmianach