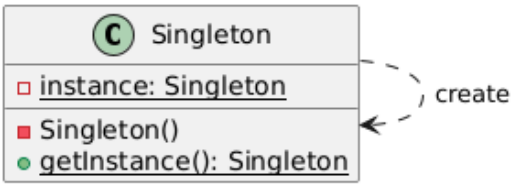
Singleton to sposób projektowania oprogramowania, który zapewnia, że dana klasa może mieć tylko jedną instancję (czyli tylko jeden obiekt tej klasy), i że istnieje globalny punkt dostępu do tego obiektu. Singleton jest szczególnie użyteczny, gdy potrzebujemy kontrolować dostęp do zasobów, które powinny być unikalne w systemie.

Najczęstsze zastosowania wzorca Singleton:
- W celu zapewnienia dostępu do wspólnych danych w całej aplikacji, np. do danych konfiguracyjnych.
- W celu ładowania i buforowania cennych zasobów tylko raz oraz zapewniania globalnego dostępu dla wszystkich i zoptymalizowania wydajności.
- Do utworzenia egzemplarza rejestratora dziennikowego aplikacji, ponieważ zazwyczaj potrzebny jest tylko jeden.
- Do implementacji algorytmów biznesowych, które powinny być jednolite w całym systemie (np. zasady naliczania punktów lojalnościowych).

Diagram klas przedstawiony na rysunku pokazuje, że Singleton składa się z jednej klasy zawierającej odniesienie do jedynego swojego egzemplarza oraz kontrolującej jego tworzenie i wykorzystanie przez jedną metodę dostępową.



1) Implementacja podstawowa

Singleton zapewnia tylko jeden obiekt danego typu, więc pierwszą czynnością podczas implementacji, powinno być przejęcie kontroli nad tworzeniem obiektów klasy. Jest to bardzo łatwe, ponieważ wystarczy w tym celu tylko ukryć konstruktor.

Następnie należy zdefiniować metodę tworzącą jedyny egzemplarz lub zwracającą go, jeśli został już utworzony. Ponieważ egzemplarz klasy *Singleton* początkowo nie istnieje, tworząca go metoda musi być statyczna, aby można ją było wywoływać przy użyciu nazwy klasy, np. *Singleton.getInstance()*.

```
public class Singleton {
    private static Singleton instance;

    private Singleton () {
        // Prywatny konstruktor zapobiega tworzeniu nowych obiektów spoza klasy
    }

    public static Singleton getInstance() {
        if (instance == null) { // 1 – problem wielowątkowości
            instance = new Singleton();
        }
        return instance;
    }
}
```

W miejscu komentarza numer 1 na listingu znajduje się kod sprawdzający, czy singleton już istnieje i tworzący go w razie negatywnego wyniku testu. Jeśli natomiast test wypadnie pozytywnie, zamiast tworzyć nowy obiekt, zwracamy ten, który został utworzony w poprzednim wywołaniu metody *getInstance()*. Każde kolejne wywołanie tej metody powinno mieć taki sam skutek w postaci zwrócenia wcześniej utworzonego egzemplarza obiektu klasy Singleton.

2) Problem wielowątkowości

Problem może się pojawić w aplikacjach wielowątkowych. W miejscu komentarza numer 1 na listingu, dwa wątki mogą sprawdzić warunek jednocześnie i oba utworzą nową instancję, co złamie zasadę Singletona. Aby rozwiązać ten problem można wykonać synchronizację z podwójnym sprawdzeniem oraz słowem kluczowym *volatile*.

```
public final class Singleton {
    private static volatile Singleton instance;

    private Singleton () {
        // Prywatny konstruktor zapobiega tworzeniu nowych obiektów spoza klasy
    }

    public static Singleton getInstance() {
        if (instance == null) {
          synchronized (Singleton.class) {
            if (instance == null) instance = new Singleton();
          }
       }
        return instance;
    }
}
```

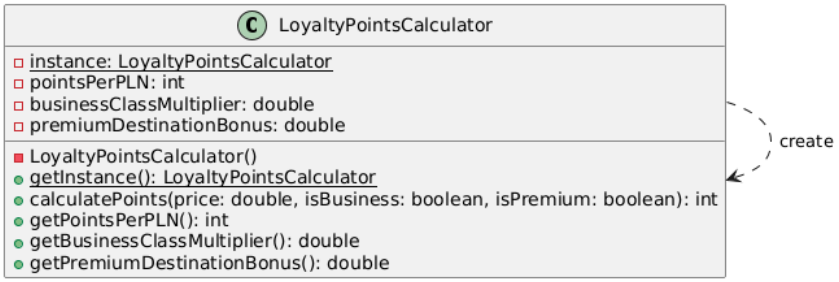
Słowo kluczowe *volatile* zapewnia, że:

- wszystkie wątki widzą najbardziej aktualną wartość zmiennej
- kompilator i procesor nie mogą zmienić kolejności instrukcji przy tworzeniu obiektu
- wszystkie operacje przed zapisem do zmiennej *volatile* są widoczne dla wątków czytających tę zmienną

Bez *volatile* jeden wątek może zobaczyć częściowo skonstruowany obiekt, co prowadzi do błędów.

Przykład użycia wzorca Singleton w systemie biura podróży:

W biurze podróży mamy program lojalnościowy, który nalicza punkty za każdą rezerwację. Chcemy, aby zasady naliczania punktów były jednolite w całym systemie i obliczane przez jeden algorytm. Możemy użyć wzorca Singleton, aby zagwarantować, że wszystkie rezerwacje w systemie używają tych samych zasad naliczania punktów lojalnościowych.



Poniżej został przedstawiony kod implementujący klasę *LoyaltyPointsCalculator*, która zarządza obliczaniem punktów lojalnościowych w biurze podróży w sposób zgodny ze wzorcem projektowym Singleton. Oznacza to, że w aplikacji może istnieć tylko jedna instancja tej klasy, a dostęp do niej uzyskujemy przez metodę *getInstance()*.

- Prywatne pole *instance* przechowuje jedyną instancję klasy *LoyaltyPointsCalculator*.
- Pole *pointsPerPLN* określa liczbę punktów lojalnościowych przyznanych za każdą wydaną 1zł.
- Pole *businessClassMultiplier* to mnożnik dla klasy biznesowej.
- Pole *premiumDestinationBonus* to dodatkowy bonus dla tzw. „kierunków premium".
- Konstruktor *LoyaltyPointsCalculator()* jest prywatny, aby uniemożliwić tworzenie obiektów tej klasy spoza niej.
- Metoda *getInstance()* zwraca jedyną instancję *LoyaltyPointsCalculator*. Jeśli instancja jeszcze nie istnieje, jest tworzona.
- Metoda *calculatePoints()* to główna metoda licząca punkty lojalnościowe.
- Wszystkie pola przechowujące zasady naliczania punktów są *final* - oznacza to, że są **niemutowalne** i nie mogą być zmienione po utworzeniu obiektu. Niemutowalny Singleton nie ma problemów z bezpieczeństwem wątkowym i jest łatwy w testowaniu.

```
public final class LoyaltyPointsCalculator {
    private static volatile LoyaltyPointsCalculator instance;

    // FINAL - zasady nie zmieniają się podczas działania systemu
    private final int pointsPerPLN;
    private final double businessClassMultiplier;
    private final double premiumDestinationBonus;

    private LoyaltyPointsCalculator() {
        // Zasady naliczania punktów - ładowane przy tworzeniu
        this.pointsPerPLN = 10;
        this.businessClassMultiplier = 1.5;
        this.premiumDestinationBonus = 1.2;
    }

    public static LoyaltyPointsCalculator getInstance() {
        if (instance == null) {
            synchronized (LoyaltyPointsCalculator.class) {
                if (instance == null) {
                    instance = new LoyaltyPointsCalculator();
                }
            }
        }
        return instance;
    }

    public int calculatePoints(double bookingPrice, boolean isBusinessClass, boolean isPremiumDestination) {
        double points = bookingPrice * pointsPerPLN;

        if (isBusinessClass) {
            points *= businessClassMultiplier;
        }

        if (isPremiumDestination) {
            points *= premiumDestinationBonus;
        }

        return (int) Math.round(points);
    }

    // Tylko gettery, brak setterów (niemutowalny stan)
    public int getPointsPerPLN () { return pointsPerPLN; }

    public double getBusinessClassMultiplier() { return businessClassMultiplier; }

    public double getPremiumDestinationBonus() { return premiumDestinationBonus; }
}
```

Przykładowy test:

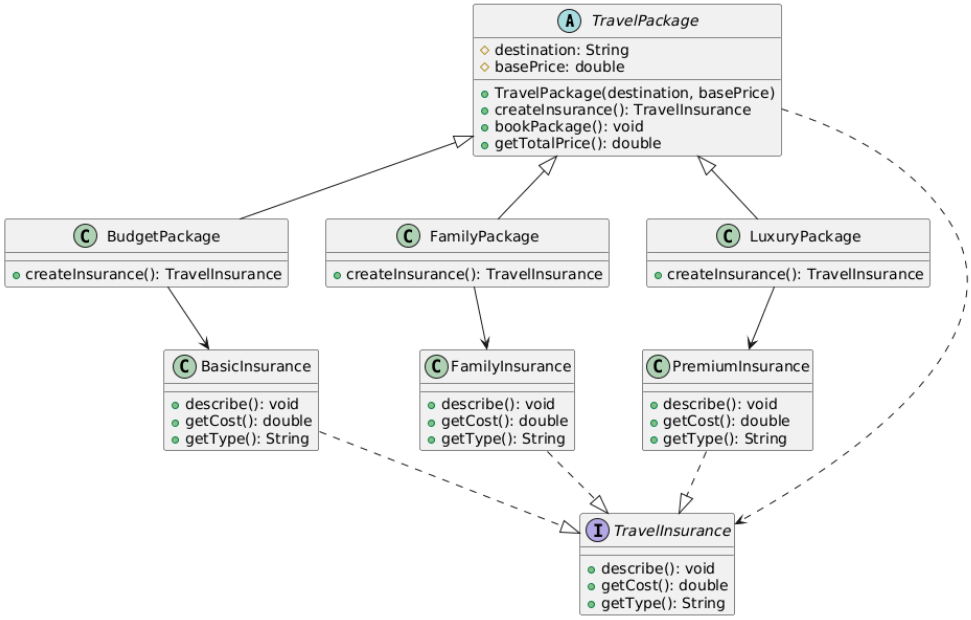
```
public class TestLoyaltyPointsCalculator {
    private LoyaltyPointsCalculator calculator;

    @BeforeEach
    public void setUp() {
        calculator = LoyaltyPointsCalculator.getInstance();
    }

    @Test
    public void testSingletonInstance() {
        LoyaltyPointsCalculator instance1 = LoyaltyPointsCalculator.getInstance();
        LoyaltyPointsCalculator instance2 = LoyaltyPointsCalculator.getInstance();

        assertSame(instance1, instance2, "Obie referencje powinny wskazywać na ten sam obiekt");
    }

    @Test
    public void testCalculatePointsBasic() {
        // Rezerwacja za 100zł, klasa ekonomiczna, standardowy kierunek
        int points = calculator.calculatePoints(100.0, false, false);

        // 100 * 10 punktów za zł = 1000 punktów
        assertEquals(1000, points, "Za rezerwację 100zł powinno być 1000 punktów");
    }
}
```

Metoda fabrykująca to wzorzec projektowy, który polega na tworzeniu obiektów przy użyciu specjalnej metody zwanej metodą fabrykującą. Zamiast bezpośrednio tworzyć obiekty za pomocą *new*, tworzymy je za pośrednictwem tej metody. Pozwala to na decydowanie, jaki typ obiektu zostanie utworzony w trakcie działania programu, co zwiększa elastyczność i skalowalność kodu.

Metody fabrykującej używamy:

- gdy w danej klasie nie można z góry ustalić klasy obiektów, które trzeba utworzyć.
- kiedy nie chcemy, aby kod bezpośrednio tworzył obiekty za pomocą *new*, tylko przez metodę fabrykującą, która może decydować o typie zwracanego obiektu.
- gdy chcemy, aby dodanie nowych typów obiektów wymagało minimalnych zmian w istniejącym kodzie.

Przykład użycia wzorca metody fabrykującej w systemie biura podróży:
System biura podróży oferuje trzy typy pakietów wycieczek, z których każdy automatycznie dołącza odpowiedni typ ubezpieczenia dostosowany do charakteru wyjazdu.



Interfejs *TravelInsurance* reprezentuje wspólny szablon wszystkich typów ubezpieczeń w biurze podróży. Konkretne klasy *BasicInsurance*, *FamilyInsurance* i *PremiumInsurance* implementują ten interfejs, oznacza to, że reprezentują trzy różne rodzaje ubezpieczeń: podstawowe, rodzinne i premium, ale z zewnątrz widzimy je zawsze przez wspólny typ *TravelInsurance*.

Klasa *TravelPackage* reprezentuje ogólny pakiet wycieczki. W tej klasie zdefiniowana jest metoda fabrykująca *createInsurance(),* która ma zwracać obiekt typu *TravelInsurance*, oraz metody biznesowe. Z *TravelPackage* dziedziczą trzy konkretne klasy: *BudgetPackage*, *FamilyPackage* i *LuxuryPackage*. Każda z nich nadpisuje metodę fabrykującą *createInsurance()* i w jej wnętrzu tworzy odpowiedni typ ubezpieczenia. *BudgetPackage* tworzy *BasicInsurance*, *FamilyPackage* tworzy *FamilyInsurance*, a *LuxuryPackage* tworzy *PremiumInsurance*. Kod, który rezerwuje wycieczkę, pracuje na poziomie abstrakcji *TravelPackage* i *TravelInsurance*: wywołuje np. *bookPackage()* lub *createInsurance(),* ale nie musi wiedzieć, czy pod spodem powstaje ubezpieczenie podstawowe, rodzinne czy premium. Dzięki temu zasada metody fabrykującej jest spełniona, decyzja o tym, jaki dokładnie obiekt ubezpieczenia utworzyć, jest ukryta w konkretnych klasach pakietów.

```
public interface TravelInsurance {
    void describe();
    double getCost();
    String getType();
}
```

```
public class BasicInsurance implements TravelInsurance {

    @Override
    public void describe() { System.out.println("=== Podstawowe Ubezpieczenie Podróżne ==="); }

    @Override
    public double getCost() { return 50.0; }

    @Override
    public String getType() { return "Basic"; }

    @Override
    public String toString() { return "BasicInsurance{cost=" + getCost() + "}"; }
}
```

```
public class PremiumInsurance implements TravelInsurance {

    @Override
    public void describe() { System.out.println("=== Premium Ubezpieczenie Podróżne ==="); }

    @Override
    public double getCost() { return 150.0; }

    @Override
    public String getType() { return "Premium"; }

    @Override
    public String toString() { return "PremiumInsurance{cost=" + getCost() + "}"; }
}
```

```
public class FamilyInsurance implements TravelInsurance {

    @Override
    public void describe() { System.out.println("=== Rodzinne Ubezpieczenie Podróżne ==="); }

    @Override
    public double getCost() { return 200.0; }

    @Override
    public String getType() { return "Family"; }

    @Override
    public String toString() { return "FamilyInsurance{cost=" + getCost() + "}"; }
}
```

```
public abstract class TravelPackage {
    protected String destination;
    protected double basePrice;

    public TravelPackage(String destination, double basePrice) {
        this.destination = destination;
        this.basePrice = basePrice;
    }

    //METODA FABRYKUJĄCA (Factory Method)
    public abstract TravelInsurance createInsurance();

    public void bookPackage() {
        // Wywołanie metody fabrykującej
        TravelInsurance insurance = createInsurance();
        insurance.describe();
    }

    public double getTotalPrice() { return basePrice + createInsurance().getCost(); }

    public String getDestination() { return destination; }

    public double getBasePrice() { return basePrice; }
}
```

```
public class BudgetPackage extends TravelPackage {

    public BudgetPackage(String destination, double basePrice) { super(destination, basePrice); }

    @Override
    public TravelInsurance createInsurance() { return new BasicInsurance(); }

    @Override
    public String toString() {
        return "BudgetPackage{" +
                "destination='" + destination + '\'' +
                ", basePrice=" + basePrice +
                ", insurance=" + createInsurance().getType() +
                ", totalPrice=" + getTotalPrice() +
                '}';
    }
}
```

```
public class LuxuryPackage extends TravelPackage {

    public LuxuryPackage(String destination, double basePrice) { super(destination, basePrice); }

    @Override
    public TravelInsurance createInsurance() { return new PremiumInsurance(); }

    @Override
    public String toString() {
        return "LuxuryPackage{" +
                "destination='" + destination + '\'' +
                ", basePrice=" + basePrice +
                ", insurance=" + createInsurance().getType() +
                ", totalPrice=" + getTotalPrice() +
                '}';
    }
}
```

```
public class FamilyPackage extends TravelPackage {

    public FamilyPackage(String destination, double basePrice) { super(destination, basePrice); }

    @Override
    public TravelInsurance createInsurance() { return new FamilyInsurance(); }

    @Override
    public String toString() {
        return "FamilyPackage{" +
                "destination='" + destination + '\'' +
                ", basePrice=" + basePrice +
                ", insurance=" + createInsurance().getType() +
                ", totalPrice=" + getTotalPrice() +
                '}';
    }
}
```

```
// Przykład użycia wzorca Factory Method w systemie biura podróży.
public class TravelAgencyDemo {

    public static void main(String[] args) {

        // Tworzenie różnych pakietów
        TravelPackage budget = new BudgetPackage("Barcelona", 500.0);
        TravelPackage luxury = new LuxuryPackage("Malediwy", 3000.0);
        TravelPackage family = new FamilyPackage("Paryż", 2000.0);

        // Rezerwacja pakietów
        budget.bookPackage();
        luxury.bookPackage();
        family.bookPackage();
    }
}
```

Fabryka abstrakcyjna pozwala na tworzenie rodzin powiązanych obiektów bez określania ich konkretnych klas. Fabryka abstrakcyjna udostępnia interfejs do tworzenia grupy powiązanych produktów, takich jak zestawy obiektów, które mogą współpracować ze sobą.

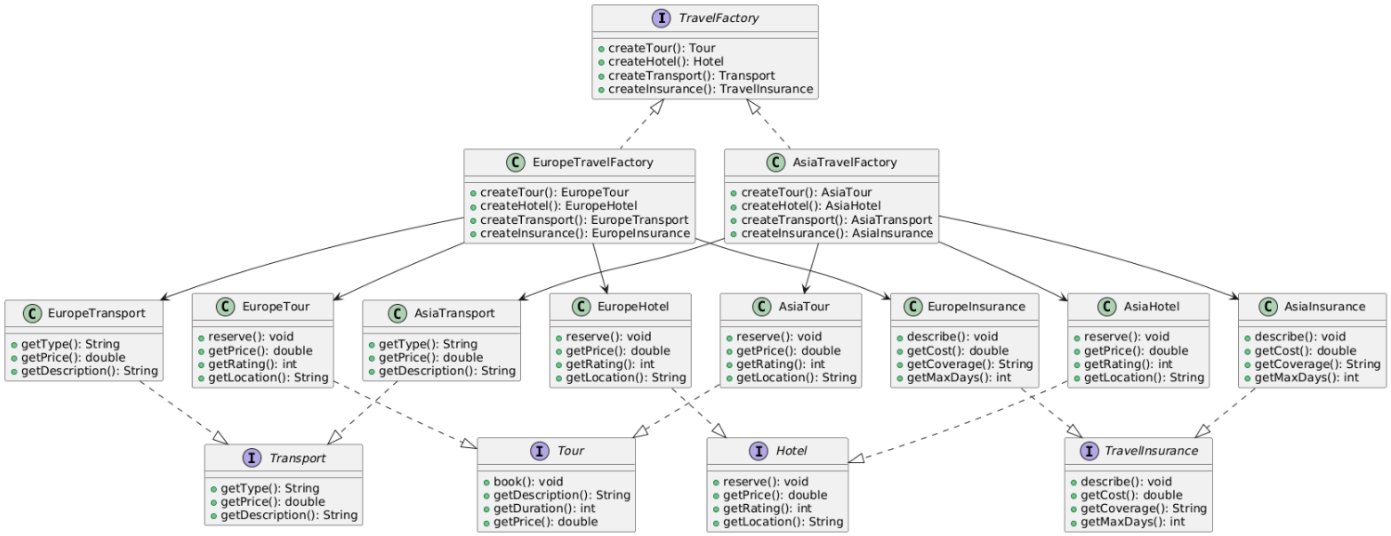
Fabryki abstrakcyjnej używamy:

- Kiedy system musi tworzyć zestawy powiązanych obiektów, które różnią się w zależności od kontekstu, ale w ramach danej rodziny produktów współpracują ze sobą.
- Gdy potrzebujemy uniezależnić kod klienta od szczegółów tworzenia różnych typów produktów.

- Gdy różne zestawy obiektów są stosowane razem i zmiana jednego elementu wymaga zmian w innych (np. zmiana jednego elementu wycieczki wpływa na pozostałe).

Przykład użycia wzorca fabryki abstrakcyjnej w systemie biura podróży:
Biuro podróży oferuje kompleksowe pakiety wakacyjne do różnych regionów świata. Każdy region wymaga innego zestawu usług, dostosowanych do specyfiki lokalnej. Przyjmijmy, że biuro podróży ma oferty wycieczek do miast w Europie i Azji. Dla każdego kontynentu, można stworzyć zestaw obiektów związanych z wycieczką, hotelem, transportem oraz ubezpieczeniem. Dzięki temu pakiet wakacyjny będzie dopasowany do danego kontynentu.

Interfejsy *Tour*, *TravelInsurance, Hotel, Transport* definiują wspólne operacje, które implementują klasy konkretnych produktów (dla Europy i Azji). Fabryka abstrakcyjna *TravelFactory* zapewnia metody tworzące obiekty w rodzinie produktów (wycieczkę, hotel, transport i ubezpieczenie). Konkretny obiekt fabryczny *EuropeTravelFactory* i *AsiaTravelFactory* – tworzą komplet obiektów dla danego regionu (Europa lub Azja), gwarantując, że produkty są ze sobą zgodne. Fabryka abstrakcyjna jest przydatna, gdy system wymaga grupowania powiązanych obiektów, tutaj tworzy spójny zestaw usług wakacyjnych dla różnych regionów.




```
public interface Tour {
    void book();
    String getDescription();
    int getDuration();
    double getPrice();
}
```

```
public interface Hotel {
    void reserve();
    double getPrice();
    int getRating();
    String getLocation();
}
```

```
public interface Transport {
    String getType();
    double getPrice();
    String getDescription();
}
```

```
public interface TravelInsurance {
    void describe();
    double getCost();
    String getCoverage();
    int getMaxDays();
}
```

```
public class EuropeTour implements Tour {
    private final String name;
    private final String route;
    private final int duration;
    private final double price;

    public EuropeTour() {
        this.name = "European Grand Tour";
        this.route = "Paryż → Amsterdam → Berlin → Praga → Wiedeń";
        this.duration = 14;
        this.price = 1200.0;
    }

    @Override
    public void book() { System.out.println("REZERWACJA WYCIECZKI PO EUROPIE"); }

    @Override
    public String getDescription() { return "Odkryj piękno Europy!" }

    @Override
    public int getDuration() { return duration; }

    @Override
    public double getPrice() { return price; }

    @Override
    public String toString() {
        return "EuropeTour{" +
                "name='" + name + '\'' +
                ", duration=" + duration + " days" +
                ", price=" + price +
                '}';
    }
}
```

```
public class EuropeHotel implements Hotel {
    private final String name;
    private final String location;
    private final int rating;
    private final double pricePerNight;
    private final String mealPlan;

    public EuropeHotel() {
        this.name = "European Boutique Hotels";
        this.location = "Centra miast - maksymalnie 15 min do głównych atrakcji";
        this.rating = 4;
        this.pricePerNight = 120.0;
        this.mealPlan = "Śniadania (BB)";
    }

    @Override
    public void reserve() { System.out.println("REZERWACJA HOTELU W EUROPIE"); }

    @Override
    public double getPrice() { return pricePerNight; }

    @Override
    public int getRating() { return rating; }

    @Override
    public String getLocation() { return location; }

    @Override
    public String toString() {
        return "EuropeHotel{" +
                "name='" + name + '\'' +
                ", rating=" + rating + " stars" +
                ", pricePerNight=" + pricePerNight +
                '}';
    }
}
```

```
public class EuropeTransport implements Transport {
    private final String type;
    private final String details;
    private final double price;

    public EuropeTransport() {
        this.type = "Pociąg";
        this.details = "komfortowe pociągi wysokiej prędkości.";
        this.price = 80.0;
    }

    @Override
    public String getType() { return type; }

    @Override
    public double getPrice() { return price; }

    @Override
    public String getDescription() { return "Transport Europa";}

    @Override
    public String toString() {
        return "EuropeTransport{" +
                "type='" + type + '\'' +
                ", price=" + price +
                '}';
    }
}
```

```
public class EuropeInsurance implements TravelInsurance {
    private final String name;
    private final String coverage;
    private final int maxDays;
    private final double cost;

    public EuropeInsurance() {
        this.name = "Insurance";
        this.coverage = "Strefa";
        this.maxDays = 30;
        this.cost = 40.0;
    }

    @Override
    public void describe() { System.out.println("Ochrona medyczna"); }

    @Override
    public double getCost() { return cost; }

    @Override
    public String getCoverage() { return coverage; }

    @Override
    public int getMaxDays() { return maxDays; }

    @Override
    public String toString() {
        return "EuropeInsurance{" +
                "coverage='" + coverage + '\'' +
                ", maxDays=" + maxDays +
                ", cost=" + cost +
                '}';
    }
}
// ANALOGICZNIE DLA AZJI
```

```
public interface TravelFactory {
    Tour createTour();
    Hotel createHotel();
    Transport createTransport();
    TravelInsurance createInsurance();
}
```

```
public class EuropeTravelFactory implements TravelFactory {

    @Override
    public Tour createTour() { return new EuropeTour(); }

    @Override
    public Hotel createHotel() { return new EuropeHotel(); }

    @Override
    public Transport createTransport() { return new EuropeTransport(); }

    @Override
    public TravelInsurance createInsurance() { return new EuropeInsurance(); }

    @Override
    public String toString() {
        return "EuropeTravelFactory - fabryka pakietów europejskich";
    }
}
```

Budowniczy to wzorzec projektowy, który oddziela proces tworzenia skomplikowanego obiektu od jego reprezentacji. Pozwala na tworzenie różnych reprezentacji tego samego obiektu przy użyciu tego samego procesu budowania. Jest szczególnie przydatny, gdy obiekt ma wiele opcjonalnych parametrów lub skomplikowaną strukturę.

Budowniczego używamy:

- kiedy konstruktor klasy ma zbyt wiele parametrów, co utrudnia czytelność i utrzymanie kodu.
- gdy proces tworzenia obiektu wymaga wielu kroków i różnych konfiguracji.
- gdy ten sam proces budowania ma generować różne warianty obiektów.

Przykład użycia wzorca budowniczy w systemie biura podróży:

W biurze podróży tworzymy kompleksowe pakiety wakacyjne, które są wysoce konfigurowalne i zawierają wiele elementów:

- Obowiązkowe: miejsce docelowe, liczba dni
- Opcjonalne: typ hotelu, rodzaj wyżywienia, środek transportu, przewodnik, ubezpieczenie, transfer z/do lotniska, lista atrakcji turystycznych, wielkość grupy.

Bez użycia wzorca Builder musielibyśmy stworzyć konstruktor z wieloma parametrami.

- Klasa *TravelDirector* zarządza procesem budowy. Jest odpowiedzialna za konfigurację wycieczek, korzystając z budowniczego.
- Klasa *TripBuilder* jest konkretną klasą odpowiedzialną za krok po kroku budowanie obiektu Trip.
- Klasa *Trip* reprezentuje kompleksowy pakiet wycieczki, który jest wynikiem procesu budowania.

```
public class Trip {
    // Pola wymagane
    private final String city;
    private final int days;

    // Pola opcjonalne
    private final String hotelType;
    private final String mealType;
    private final String transportType;
    private final boolean guide;
    private final boolean insurance;
    private final boolean transfer;
    private final List<String> activities;
    private final int groupSize;

    Trip(String city, int days, String hotelType, String mealType, String transportType, boolean guide, boolean
insurance, boolean transfer, List<String> activities, int groupSize) {

        this.city = city;
        this.days = days;
        this.hotelType = hotelType;
        this.mealType = mealType;
        this.transportType = transportType;
        this.guide = guide;
        this.insurance = insurance;
        this.transfer = transfer;
        this.activities = new ArrayList<>(activities);
        this.groupSize = groupSize;
    }

    // Gettery

    public double getTotalCost() {
        double cost = 0.0;

        return cost;
    }
}
```

```
public class TripBuilder {
    private String city;
    private int days;

    private String hotelType = "THREE_STAR";
    private String mealType = "BREAKFAST";
    private String transportType = "BUS";
    private boolean guide = false;
    private boolean insurance = false;
    private boolean transfer = false;
    private List<String> activities = new ArrayList<>();
    private int groupSize = 1;

    public TripBuilder(String city, int days) {
        this.city = city;
        this.days = days;
    }

    public TripBuilder setHotelType(String hotelType) {
        this.hotelType = hotelType;
        return this;
    }
    // pozostałe setter

    public TripBuilder reset() {
        this.hotelType = "THREE_STAR";
        this.mealType = "BREAKFAST";
        this.transportType = "BUS";
        this.guide = false;
        this.insurance = false;
        this.transfer = false;
        this.activities = new ArrayList<>();
        this.groupSize = 1;
        return this;
    }

    public Trip build() {
        validateBeforeBuild();

        return new Trip(city, days, hotelType, mealType, transportType, guide, insurance, transfer, activities,
groupSize);
    }
```

```
  private void validateBeforeBuild() {
        if (city == null || city.trim().isEmpty()) {
            throw new IllegalStateException("Miasto nie może być puste");
        }

        if (days <= 0) {
            throw new IllegalStateException("Liczba dni musi być większa niż 0 (podano: " + days + ")");
        }

        if (days > 365) {
            throw new IllegalStateException("Liczba dni nie może przekraczać 365 (podano: " + days + ")");
        }

        // Walidacja groupSize
        if (groupSize <= 0) {
            throw new IllegalStateException("Wielkość grupy musi być większa niż 0 (podano: " + groupSize + ")");
        }

        if (groupSize > 50) {
            throw new IllegalStateException("Wielkość grupy przekracza 50 osób (podano: " + groupSize + ")");
        }

        if (hotelType == null) {
            throw new IllegalStateException("Typ hotelu nie może być null");
        }

        if (mealType == null) {
            throw new IllegalStateException("Typ wyżywienia nie może być null");
        }

        if (transportType == null) {
            throw new IllegalStateException("Typ transportu nie może być null");
        }
    }

    @Override
    public String toString() {
        return "TripBuilder{" +
                "city='" + city + '\'' + ", days=" + days +
                ", hotelType=" + hotelType + ", mealType=" + mealType +
                ", transportType=" + transportType + ", guide=" + guide +
                ", insurance=" + insurance + ", transfer=" + transfer +
                ", activities=" + activities.size() + ", groupSize=" + groupSize + '}';
    }
}
```

```
public class TravelDirector {
    private TripBuilder builder;

    public TravelDirector(TripBuilder builder) { this.builder = builder; }

    public TripBuilder getBuilder() { return builder; }

    public void setBuilder(TripBuilder builder) { this.builder = builder; }

    public Trip buildBudgetTrip(String city, int days) {
        return builder
                .reset() // Reset do wartości domyślnych
                .setHotelType("THREE_STAR")
                .setMealType("BREAKFAST")
                .setTransportType("BUS")
                .setGuide(false)
                .setInsurance(false)
                .setTransfer(false)
                .setGroupSize(30)
                .build();
    }

    public Trip buildStandardTrip(String city, int days) {
        boolean needsGuide = isPopularDestination(city);
        return builder
                .reset()
                .setHotelType("FOUR_STAR")
                .setMealType("HALF_BOARD")
                .setTransportType("TRAIN")
                .setGuide(needsGuide)
                .setInsurance(true)
                .setTransfer(false)
                .setGroupSize(20)
                .build();
    }
```

```
    public Trip buildLuxuryTrip(String city, int days) {
        TripBuilder luxuryBuilder = builder
                .reset()
                .setHotelType("FIVE_STAR")
                .setMealType("FULL_BOARD")
                .setTransportType("FLIGHT")
                .setGuide(true)
                .setInsurance(true)
                .setTransfer(true)
                .setGroupSize(10);

        addPremiumActivities(luxuryBuilder, city);
        return luxuryBuilder.build();
    }

    public Trip buildAllInclusiveTrip(String city, int days) {
        TripBuilder aiBuilder = builder
                .reset()
                .setHotelType("RESORT")
                .setMealType("ALL_INCLUSIVE")
                .setTransportType("FLIGHT")
                .setGuide(true)
                .setInsurance(true)
                .setTransfer(true)
                .setGroupSize(15);

        addAllActivities(aiBuilder, city);
        return aiBuilder.build();
    }

    public Trip buildCustomTrip() { return builder.build(); }

    private boolean isPopularDestination(String city) {
        String cityLower = city.toLowerCase();
        return cityLower.contains("pary") || cityLower.contains("lond") || // Paryż lub Londyn

    }

    private void addPremiumActivities(TripBuilder builder, String city) {
        String cityLower = city.toLowerCase();

        if (cityLower.contains("pary")) {
            builder.addActivity("Wieża Eiffla - wejście na szczyt")
                   .addActivity("Wersalski Pałac - zwiedzanie");
        } else if (cityLower.contains("lond")) {
            builder.addActivity("London Eye - VIP kabina")
                   .addActivity("British Museum - ekskluzywna wycieczka");
        }
    }

    private void addAllActivities(TripBuilder builder, String city) {
        addPremiumActivities(builder, city);

        builder.addActivity("Wycieczka kulinarna")
               .addActivity("Wieczorne zwiedzanie");
    }
}
```

```
public class BuilderPatternDemo {
    public static void main(String[] args) {
        Trip customTrip = new TripBuilder("Paryż", 7)
                .setHotelType("BOUTIQUE")
                .setMealType("HALF_BOARD")
                .setTransportType("TRAIN")
                .setGuide(true)
                .setInsurance(true)
                .addActivity("Wieża Eiffla")
                .setGroupSize(5)
                .build();
        }
        //LUB
        TripBuilder builder = new TripBuilder("Barcelona", 10);
        TravelDirector director = new TravelDirector(builder);
        Trip budgetTrip = director.buildBudgetTrip("Barcelona", 5);
}
```

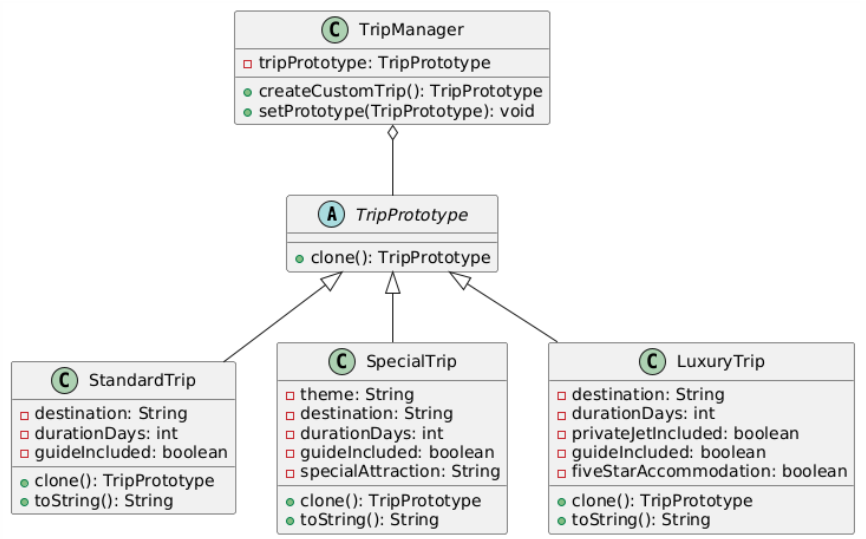
Prototyp pozwala na tworzenie nowych obiektów poprzez klonowanie już istniejących instancji, zamiast ich ręcznego konstruowania. Obiekt nazywany "prototypem" zawiera pełną informację o sobie i może tworzyć swoje kopie. Dzięki temu, zamiast tworzyć nowy obiekt od zera, można skopiować istniejący, co jest szybsze i bardziej efektywne.

Prototypu używamy:

- gdy chcemy stworzyć kopię istniejącego obiektu, który już ma wszystkie właściwości ustawione, zamiast go konfigurować od początku.
- kiedy chcemy mieć pewność, że nowo stworzony obiekt będzie identyczny lub bardzo podobny do istniejącego.

Przykład użycia wzorca prototypu w systemie biura podróży:

W biurze podróży możesz mieć pewne ustalone plany podróży, jak np. wycieczki dla rodzin, par czy osób zainteresowanych aktywnym wypoczynkiem. Użycie wzorca Prototyp pozwala na łatwe klonowanie tych planów i personalizowanie ich dla każdego klienta.




```
// Klasa bazowa dla prototypów wycieczek
public abstract class TripPrototype {

    // Metoda abstrakcyjna do klonowania obiektu
    public abstract TripPrototype clone();
}
```

```
// Klasa konkretna reprezentująca standardową wycieczkę
public class StandardTrip extends TripPrototype {
    private String destination;
    private int durationDays;
    private boolean guideIncluded;

    // Konstruktor
    public StandardTrip(String destination, int durationDays, boolean guideIncluded) {
        this.destination = destination;
        this.durationDays = durationDays;
        this.guideIncluded = guideIncluded;
    }

    // Implementacja metody clone() - klonowanie obiektu
    @Override
    public TripPrototype clone() {
        return new StandardTrip(destination, durationDays, guideIncluded);
    }

    @Override
    public String toString() {
        return "Standard Trip [Destination: " + destination + ", Duration: " + durationDays + " days, Guide
Included: " + guideIncluded + "]";
    }
}
```

```
// Klasa konkretna reprezentująca tematyczną wycieczkę
public class SpecialTrip extends TripPrototype {
    private String theme;
    private String destination;
    private int durationDays;
    private boolean guideIncluded;
    private String specialAttraction;

    public SpecialTrip(String theme, String destination, int durationDays, boolean guideIncluded, String
specialAttraction) {
        this.theme = theme;
        this.destination = destination;
        this.durationDays = durationDays;
        this.guideIncluded = guideIncluded;
        this.specialAttraction = specialAttraction;
    }

    // Implementacja metody clone() - klonowanie obiektu
    @Override
    public TripPrototype clone() {
        return new SpecialTrip(theme, destination, durationDays, guideIncluded, specialAttraction);
    }

    @Override
    public String toString() {
        return "Special Trip [Theme: " + theme + ", Destination: " + destination + ", Duration: " +
durationDays + " days, Guide Included: " + guideIncluded + ", Special Attraction: " + specialAttraction + "]";
    }
}
```

```
// Klasa konkretna reprezentująca luksusową wycieczkę
public class LuxuryTrip extends TripPrototype {
    private String destination;
    private int durationDays;
    private boolean privateJetIncluded;
    private boolean guideIncluded;
    private boolean fiveStarAccommodation;

    // Konstruktor
    public LuxuryTrip(String destination, int durationDays, boolean privateJetIncluded, boolean guideIncluded,
boolean fiveStarAccommodation) {
        this.destination = destination;
        this.durationDays = durationDays;
        this.privateJetIncluded = privateJetIncluded;
        this.guideIncluded = guideIncluded;
        this.fiveStarAccommodation = fiveStarAccommodation;
    }

    // Implementacja metody clone() - klonowanie obiektu
    @Override
    public TripPrototype clone() {
        return new LuxuryTrip(destination, durationDays, privateJetIncluded, guideIncluded,
fiveStarAccommodation);
    }

    @Override
    public String toString() {
        return "Luxury Trip [Destination: " + destination + ", Duration: " + durationDays + " days, Private
Jet: " + privateJetIncluded + ", Guide Included: " + guideIncluded + ", 5-Star Accommodation: " +
fiveStarAccommodation + "]";
    }
}
```

```
// Klasa zarządzająca prototypami i ich klonowaniem
public class TripManager {
    private TripPrototype tripPrototype;

    public TripManager(TripPrototype tripPrototype) {
        this.tripPrototype = tripPrototype;
    }

    // Tworzenie nowej wycieczki na podstawie prototypu
    public TripPrototype createCustomTrip() {
        return tripPrototype.clone();
    }

    // Zmiana aktualnego prototypu
    public void setPrototype(TripPrototype newPrototype) {
        this.tripPrototype = newPrototype;
    }
}
```

- *TripPrototype* to klasa bazowa, która zawiera metodę *clone()*, służącą do klonowania obiektów wycieczek. Klasa ta jest abstrakcyjna, a konkretne dziedziczą po niej.
- *StandardTrip*, *SpecialTrip* i *LuxuryTrip* to klasy reprezentująca wycieczkę standardową, tematyczną i luksusową, implementujące metodę *clone()*.
- *TripManager* to klasa zarządzająca prototypami i ich klonowaniem. Pozwala na tworzenie kopii wycieczek oraz zmianę aktualnie używanego prototypu.

Źródła:

- „*Wzorce projektowe. Elementy oprogramowania obiektowego wielokrotnego użytku.*”, E.Gamma, R.Helm, R.Johnson, J.Vlissides, Helion, 2010
- „*Java EE. Zaawansowane wzorce projektowe*.” , M.Yener, A.Theedom, Helion, 2015