Politechnika Świętokrzyska
Wydział Elektrotechniki, Automatyki i Informatyki

Zespół:
Prusicki Jakub

Narzędzia odwzorowania mapowania obiektowo - relacyjnego

Projekt z Technologii Obiektowych
na studiach stacjonarnych
o kierunku **Informatyka**
Stopień **II**

Opiekun projektu:
dr inż. Mariusz Bedla

Kielce, 2022

SPIS TREŚCI

1. Wstęp teoretyczny.

1.1. Adnotacje.

„Adnotacje w języku Java to mechanizm metadanych, który pozwala programistom dodawać informacje o programie do kodu źródłowego. Adnotacje są zazwyczaj określane przez symbol “@”, po którym następuje nazwa adnotacji. Adnotacje mogą być stosowane do klas, metod, zmiennych, parametrów i pakietów. Adnotacje są konstrukcją, która pozwala na przekazywanie dodatkowych informacji na temat kodu. Informacje te mogą być wykorzystane później w kilku miejscach. Mówi się, że adnotacje służą do przekazywania metadanych. Innymi słowy przekazują one dane o kodzie źródłowym.

Wraz z Java 1.5 zostały wprowadzone adnotacje. Adnotacjami nazywamy dodatkowe informacje (metadane) umieszczone w kodzie programu źródłowego, które nie wpływają na działanie programu. Wielu początkujących programistów nie jest w stanie dostrzec ich zastosowania. Dopiero po zapoznaniu się z nimi i dłuższym programowaniu widać ich sens. Mam nadzieję, że po przeczytaniu tego artykułu ujrzysz możliwe zastosowania i potencjał ich wykorzystania. Możliwe wykorzystania:

- Metaprogramowanie
- Tworzenie własnych wskazówek
- Sterowanie komunikatami generowanymi przez kompilator

@Table

Adnotacja @Table pozwala na zmianę nazwy odwzorowanej tabeli. Domyślnie nazwa jest taka sama jak nazwa klasy. W Java dobrą praktyką jest zapisywanie nazwy klasy w liczbie pojedynczej z wielkiej litery. Natomiast w bazie danych tabele zapisuje się małymi znakami w liczbie mnogiej. Aby było to możliwe wykorzystujemy adnotacje @Table.

```
1  @Entity
2  @Table(name = "books")
3  public class Book {
4
5  }
```

Rys. 1.1 Przykład adnotacji @Table.

@Column

Analogicznie do @Table działa adnotacja @Column – jednak tyczy się ona kolumn. Konwencją w Java jest tworzenie nazw zmiennych zapisywanych notacją camelCase. W bazie danych nazwy zmiennych wielowyrazowe rozdzielane są znakiem podkreślenia/podłogi -> „_”.

```java
@Entity
@Table(name = "books")
public class Book {
    @Column(name = "publish_date")
    private LocalDate publishDate;
}
```

Rys. 1.2 Przykład adnotacji @Column.

@Enumerated

Domyślnie typ enum jest zapisywany w bazie danych w postaci liczbowej – według kolejności wartości zapisanej w tym typie. Adnotacja *@Enumerated* pozwala tym sterować. Możemy wówczas ustawić, że wartości będą zapisywane w postaci ciągu tekstowego.

```java
@Entity
@Table(name = "books")
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String title;
    private String author;
    @Column(name = "publish_date")
    private LocalDate publishDate;

    @Enumerated(EnumType.ORDINAL) // default
    private BookType bookType;

    // geters + setters
}
```

Rys. 1.3 Przykład adnotacji @Enumerated.

Oczywiście trzeba poważnie rozważyć czy stosowanie typu enum w postaci łańcucha znaków w bazie danych ma sens. Wszystko zależy od potrzeb systemu. Trzeba pamiętać, że zapisywanie tekstowej wartości typu wyliczeniowego do kolumny zamiast liczby wiąże się ze znacznie zwiększona ilością wykorzystywanej przestrzeni dyskowej przez bazę danych. Zwłaszcza jeśli mamy w tabeli dużo rekordów''[1].

1.2. ORM.

„ORM to akronim od *Object-Relational Mapping*, a narzędzia ORM pozwalają na mapowanie obiektów w języku programowania na odpowiednie zapytania SQL i operacje na bazach danych. Oto kilka popularnych narzędzi ORM:

1. ***Hibernate*** - popularne narzędzie ORM dla języka Java, które oferuje obsługę różnych baz danych i ułatwia operacje na bazach danych.

2. ***Entity Framework*** - narzędzie ORM dla platformy .NET, które umożliwia pracę z bazami danych SQL Server, MySQL, Oracle, PostgreSQL i inne.

3. ***Django ORM*** - narzędzie ORM dla frameworka webowego Django w języku Python, które ułatwia interakcję z bazami danych SQL.

4. ***Sequelize*** - narzędzie ORM dla języka JavaScript, które umożliwia łatwe mapowanie obiektów na relacyjne bazy danych w środowisku Node.js.

5. ***SQLAlchemy*** - narzędzie ORM dla języka Python, które oferuje wiele zaawansowanych funkcji, takich jak automatyczne tworzenie tabel, obsługę transakcji i wiele innych frameworków i bibliotek ,które można wykorzystać w zależności od języka programowania i wymagań projektowych.

Myśląc o technologii ORM jak o pewnego rodzaju pomoście już nie tylko pomiędzy kolejnymi warstwami aplikacji, lecz także między aplikacją a bazą danych, można dodać do wniosku, że skoro istnieje interfejs, to powinna nastąpić znaczna redukcja kodu. Jak najbardziej można się zgodzić z tak postawionym sformułowaniem.

Zastosowanie technologii ORM, gwarantuje implementację podstawowych operacji, takich jak: dodawanie, odczyt, modyfikacja, usuwanie nazywanymi operacjami

typu CRUD (ang. create, read, update, delete). Bez wątpienia fakt ten wpływa na przyspieszenie procedury tworzenia kodu przez programistę, zwalniając go tym samym z obowiązku tworzenia procedur składowanych odpowiedzialnych za wspomniane operacje. W tym przypadku zastosowania gotowego rozwiązania nie wyklucza możliwości użycia niestandardowej procedury dla konkretnych warunków, przy czym istnieje także możliwości implementacji odpowiedniej funkcji z poziomu samej aplikacji. Wszystkie te cechy gwarantują elastyczność i dostosowanie rozwiązania do ustalonych wymagań. Podczas procesu projektowania, a później tworzenia relacyjnej bazy danych moĪna kierować się różnego rodzaju przesłankami, które mają wpływ na końcową postać schematu bazy danych. Czasem będzie to minimalizacja czasu dostępu do danych, innym razem oszczędność pamięci fizycznej lub w skrajnych przypadkach będą to błędy na poziomie projektowym. Jednak bez względu na przyczynę taki model rzadko w pełni odzwierciedla logikę, jaką będzie musiała realizować aplikacja. W takich przypadkach zastosowanie technologii *ORM* pozwoli utworzyć taki model biznesowy, jaki w rzeczywistości jest wymagany, co także usprawni dalszy proces rozwoju systemu oraz traktowania go jako zgodnego z założeniami. Ta cecha jest szczególnie istotna także z punktu widzenia procesu wytwarzania oprogramowania przy wzięciu pod uwagę faktu, że w pracy nad projektem zaangażowani są zarówno projektanci, mający na celu utworzenie jak najbardziej wydajnej struktury bazy danych, jak i programiści, którzy skupiają się na samej logice systemu''[2].

1.3. HIBERNATE.

„Hibernate to framework umożliwiający mapowanie obiektowo-relacyjne (ORM) w języku Java. Hibernate umożliwia programistom pracę na poziomie obiektów, a jednocześnie automatycznie mapuje je na poziom relacyjnej bazy danych. Aby skorzystać z Hibernate'a, należy najpierw skonfigurować jego ustawienia. Można to zrobić przy użyciu pliku konfiguracyjnego hibernate.cfg.xml, który definiuje m.in. sposób połączenia z bazą danych oraz mapowanie klas Java na tabele w bazie danych. W Hibernate'ie klasy Java nazywane są encjami, a ich pola odpowiadają kolumnom w tabelach bazy danych. Hibernate umożliwia także tworzenie zapytań przy użyciu języka HQL (Hibernate Query Language) oraz kryteriów. Hibernate posiada także wiele narzędzi umożliwiających m.in. tworzenie tabel w bazie danych na podstawie encji, generowanie kodu na podstawie istniejącej bazy danych, czy też narzędzia do zarządzania transakcjami. Do pracy z Hibernate'em często wykorzystuje się także dodatkowe narzędzia i biblioteki, takie jak

Spring Framework czy JPA (Java Persistence API), które umożliwiają łatwiejszą konfigurację oraz szybsze wystartowanie samego projektu.

Korzyści

- ➢ programista nie musi tworzyć struktury bazy danych dla swojej aplikacji.
- ➢ programista, nie musi wykonywać ręcznych zapytań do bazy danych;
- ➢ znacznie przyspiesza proces tworzenia aplikacji;
- ➢ czysty, schludny kod.

Wady

- ➢ jest dodatkową warstwą na JDBC i generuje on opóźnienie w czasie dostępu
- ➢ generowanie nadmiarowych zapytań – np. znany problem „hibernate n+1”.

Jak wygląda realność rozwiązania ?

➢ Obecnie większość firm wybiera stosowanie tego frameworka, ponieważ decydującym czynnikiem jest skrócenie czasu tworzenia rozwiązań informatycznych. Dodatkowo całe rozwiązanie programistyczne jest bardziej znacznie czytelne niż korzystanie z JDBC (które wymaga wielu klas szablonowych).

➢ Obecnie stosowana techniką jest korzystanie z Hibernate w miejscach gdzie tylko to jest możliwe, natomiast w przypadku gdzie ważny jest czas dostępu korzystać z JDBC.

➢ Dobrą praktyką jest staranie się by korzystać tylko z jednego rozwiązania albo ORM albo JDBC. Aplikacje tworzone w ten sposób są znacznie łatwiejsze w utrzymaniu[3].

2. Rodzaje mapowań.

2.1. Single Table Inheritance.

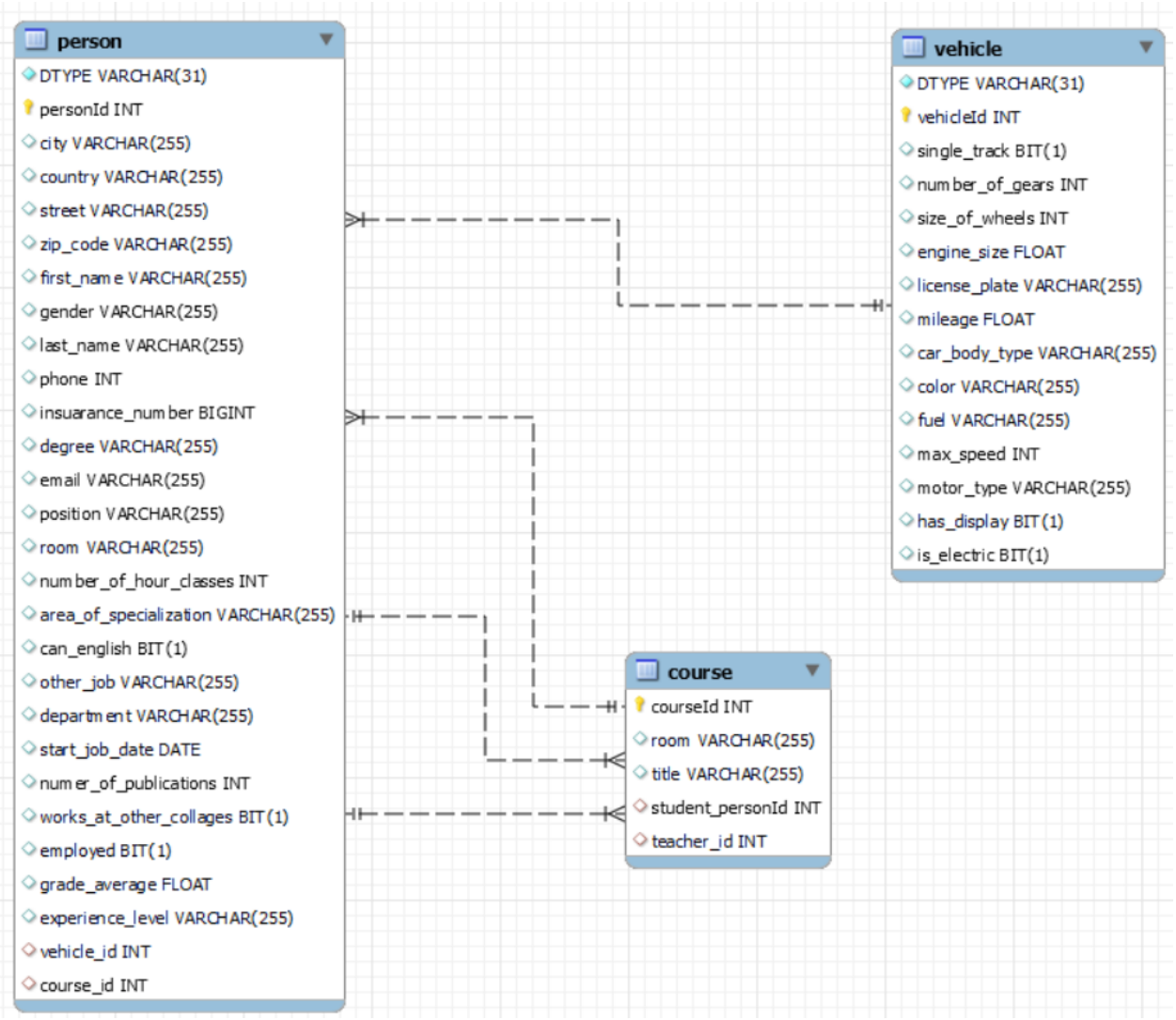
„Strategia pojedynczej tabeli tworzy jedną tabelę dla każdej hierarchii klas. JPA domyślnie wybiera tę strategię, jeśli nie określimy jej jawnie ... Możemy zdefiniować strategię, której chcemy użyć, dodając adnotację @Inheritance do superklasy”[3]:

```
@Entity
@Inheritance(strategy = InheritanceType.SINGLE_TABLE)
public class MyProduct {
    @Id
    private long productId;
    private String name;

    // constructor, getters, setters
}
```

Rys. 2.1 Przykład strategii Single Table Inheritance[3].

Single Table Inheritance (STI) to jedna z technik używanych w dziedzinie mapowania obiektowo-relacyjnego (ORM) w programowaniu obiektowym. Polega na przechowywaniu danych różnych klas dziedziczących w jednej tabeli w bazie danych, z dodatkowym polem identyfikującym typ obiektu. W STI, każda klasa dziedzicząca reprezentowana jest przez wiersze w tej samej tabeli. Pola specyficzne dla danej klasy są przechowywane w kolumnach tabeli, a pole identyfikujące typ obiektu wskazuje, do jakiej klasy należy dany wiersz. Dzięki temu, w jednej tabeli można przechowywać obiekty różnych klas, co eliminuje potrzebę tworzenia osobnych tabel dla każdej klasy. Główną zaletą STI jest prostota struktury bazy danych i możliwość wykonywania zapytań dotyczących wielu typów obiektów jednocześnie. Jednakże, może to prowadzić do niejednoznaczności i redundancji w przypadku, gdy klasy dziedziczące mają bardzo różne zestawy pól i zachowań. Ponadto, dodanie nowych pól do jednej klasy może wymagać zmian w strukturze całej tabeli. Mimo pewnych ograniczeń, Single Table Inheritance jest popularnym podejściem w przypadkach, gdy dziedziczące klasy mają podobne struktury i współdzielą wiele pól. Umożliwia ono efektywne przechowywanie i manipulację danymi związanych z różnymi typami obiektów w jednej tabeli[4].



Rys. 2.2 Diagram dla strategii Single Table Inheritance.

Cechą charakterystyczną tej strategii jest duża liczba wartości **Null**. Poniższe tabele przedstawiają tę sytuację na podstawie zrealizowanego projektu.

number_of_all_persons	insuarance_number_nulls	degree_nulls	position_nulls	room_nulls	number_of_hour_classes_nulls	area_of_specialization_nulls	can_english_nulls	other_job_nulls	department_nulls
2000	500	1000	1000	1000	1500	1833	1844	1844	1000

number_of_all_persons	department_nulls	start_job_date_nulls	numer_of_publications_nulls	works_at_other_collages_nulls	employed_nulls	grade_average_nulls	experience_level_nulls	vehicle_id_nulls
2000	1000	1500	1823	1823	1500	1500	1500	0

number_of_all_vehicles	number_of_gears_nulls	size_of_wheels_nulls	engine_size_nulls	license_plate_nulls	mileage_nulls	carBodyType_nulls	color_nulls	fuel_nulls	max_speed_nulls	motorType_nulls	has_display_nulls	is_electric_nulls
2000	1519	1519	978	978	978	1460	1460	1460	1518	1518	1503	1503

2.2. Joined Table Inheritance.

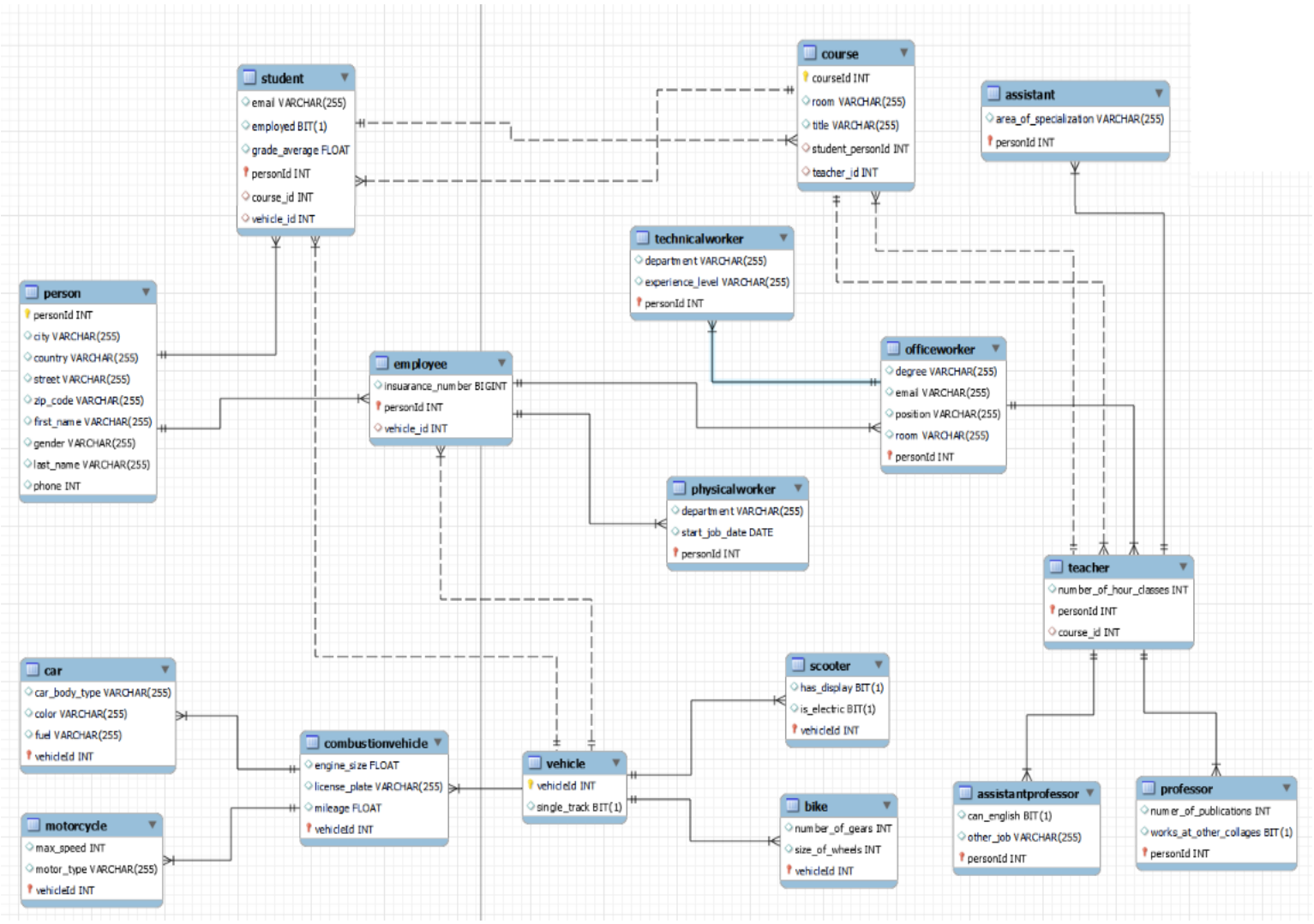
..Korzystając z tej strategii, każda klasa w hierarchii jest odwzorowana na swoją tabelę. Jedyną kolumną, która wielokrotnie pojawia się we wszystkich tabelach, jest identyfikator, który zostanie użyty do ich połączenia w razie potrzeby”[3].

```java
@Entity
@Inheritance(strategy = InheritanceType.JOINED)
public class Animal {
    @Id
    private long animalId;
    private String species;

    // constructor, getters, setters
}
```

Rys. 2.2 Diagram dla strategii Single Table Inheritance[3].

„Joined Table Inheritance (JTI) to technika mapowania obiektowo-relacyjnego, która umożliwia reprezentację hierarchii dziedziczenia klas w bazie danych poprzez tworzenie oddzielnych tabel dla każdej klasy. Każda tabela zawiera specyficzne pola i dane związane z daną klasą, co pozwala na precyzyjne odwzorowanie struktury obiektów. W JTI relacje między klasami są nawiązywane poprzez klucze obce, które umożliwiają powiązanie rekordów między tabelami. Dzięki temu możliwe jest odwzorowanie zarówno relacji jeden do jeden, jak i jeden do wielu pomiędzy klasami dziedziczącymi. Każda tabela prezentuje tylko te pola, które są specyficzne dla danej klasy, co prowadzi do uniknięcia redundancji danych. Zaletą JTI jest większa elastyczność struktury bazy danych, ponieważ każda klasa może mieć swoje własne zestawy pól, a zmiany w jednej klasie nie wpływają na strukturę pozostałych klas. Umożliwia to łatwe dodawanie nowych klas do hierarchii dziedziczenia bez konieczności modyfikowania istniejących tabel. Jednak korzystanie z JTI może prowadzić do bardziej skomplikowanych zapytań, ponieważ często wymaga łączenia tabel w celu pobrania danych z różnych klas. To może wpływać na wydajność systemu, zwłaszcza w przypadku dużych ilości danych. Warto zwrócić uwagę na optymalizację zapytań i indeksowanie odpowiednich kolumn, aby zoptymalizować działanie bazy danych”[4].



Rys. 2.3 Diagram dla strategii JOIN Table Inheritance[3].

„Joined Table Inheritance (JTI) charakteryzuje się tym, że każda klasa w hierarchii dziedziczenia ma swoją własną tabelę w bazie danych. To umożliwia precyzyjne odwzorowanie struktury obiektów i uniknięcie redundancji danych. Relacje między klasami są nawiązywane za pomocą kluczy obcych, co umożliwia **elastyczne zarządzanie powiązanymi danymi**”[4].

2.3. Table per Class .

„Strategia Tabela według klas odwzorowuje każdy podmiot na jego tabelę, która zawiera wszystkie właściwości podmiotu, w tym te odziedziczone.. Wynikowy schemat jest podobny do schematu używającego @MappedSuperclass. Ale Tabela według klasy rzeczywiście zdefiniuje jednostki dla klas nadrzędnych, umożliwiając w rezultacie skojarzenia i zapytania polimorficzne.. Aby skorzystać z tej strategii, musimy tylko dodać adnotację @Inheritance do klasy podstawowej”[3]:

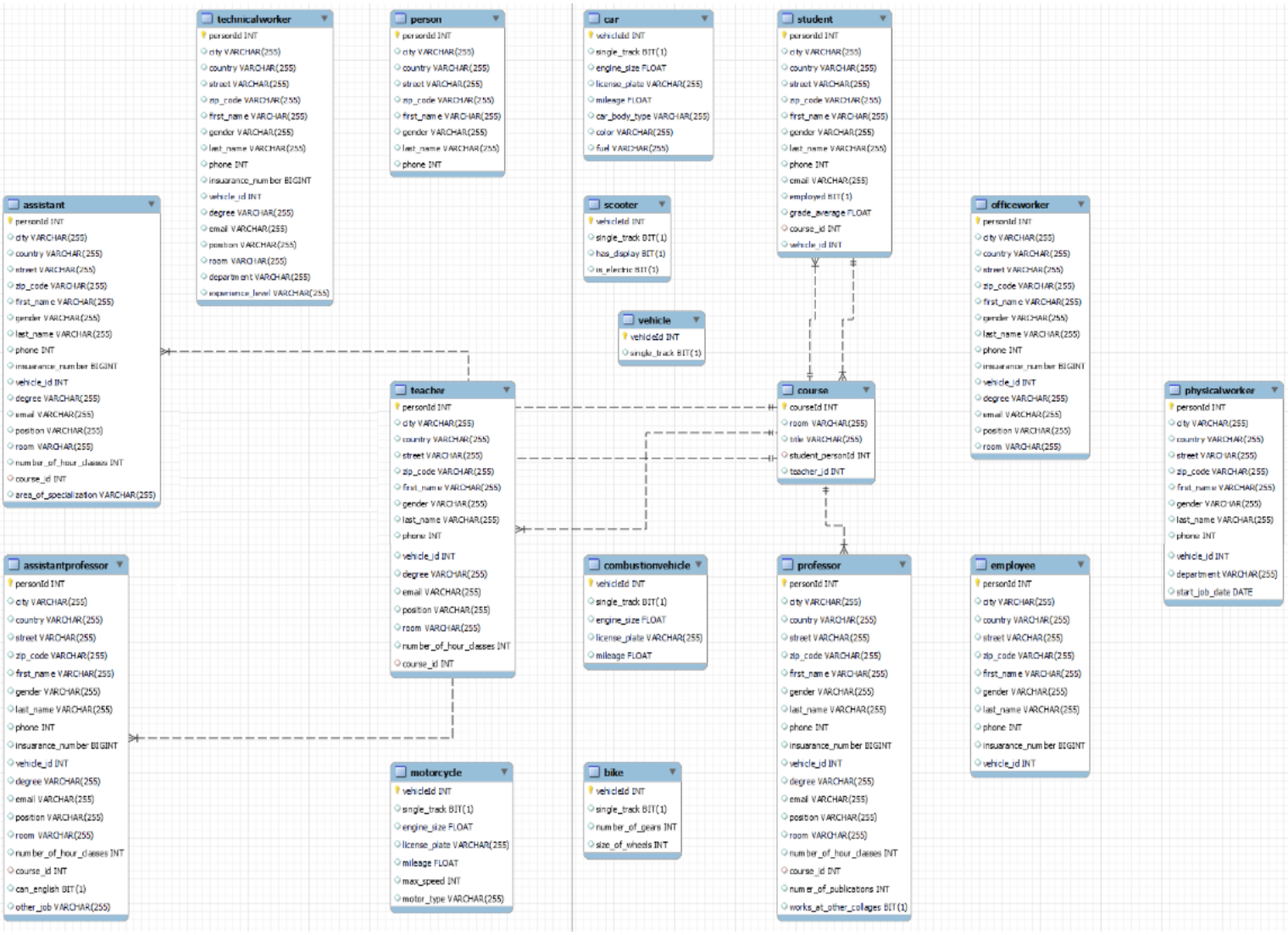
```
@Entity
@Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
public class Vehicle {
    @Id
    private long vehicleId;

    private String manufacturer;

    // standard constructor, getters, setters
}
```

Rys. 2.3 Diagram dla strategii Single Table Inheritance[3].

„Table per Class (TPC) to strategia mapowania obiektowo-relacyjnego, w której każda klasa w hierarchii dziedziczenia jest reprezentowana przez oddzielną tabelę w bazie danych. Każda tabela zawiera zarówno pola specyficzne dla danej klasy, jak i dziedziczone z klas nadrzędnych. Dzięki temu podejściu można bezpośrednio odwzorować strukturę klas w schemacie bazy danych, co zapewnia integralność danych i zachowanie hierarchii klas. W przypadku TPC, zapytanie danych dla konkretnej klasy jest proste, ponieważ polega na bezpośrednim dostępie do odpowiadającej tabeli. Jednak, gdy potrzebne jest pobranie danych z kilku klas jednocześnie, wymaga to łączenia tabel, co może skomplikować i spowolnić zapytania. Jedną z zalet TPC jest **unikanie redundancji** poprzez przechowywanie tylko odpowiednich pól w każdej tabeli. Ponadto, umożliwia efektywne zapytanie i indeksowanie danych specyficznych dla każdej klasy. Z drugiej strony, dodanie nowych klas do hierarchii może wymagać tworzenia dodatkowych tabel, co może zwiększyć złożoność schematu bazy danych”[4].



Rys. 2.3 Diagram dla strategii Table Per Class Inheritance[3].

„Ogólnie rzecz biorąc, Table per Class łączy w sobie zachowanie projektu obiektowego i elastyczną, wydajną strukturę bazy danych, ale wymaga ostrożnego rozważenia kompromisów i wpływu na wydajność zapytań” [4].

2.4. Porównanie mapowań.

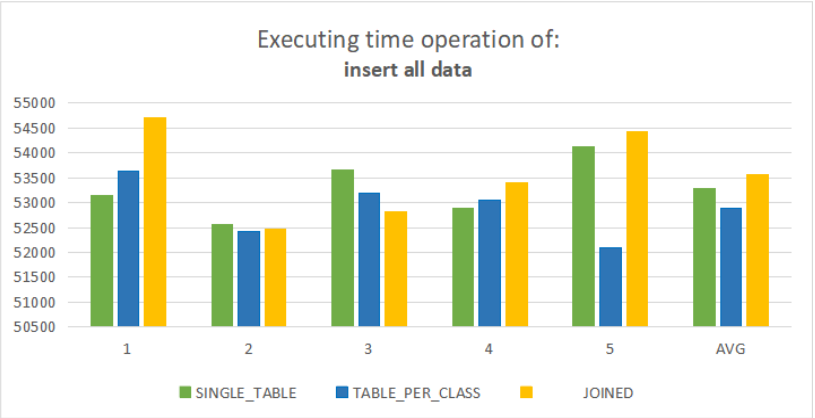
Już do tej pory można było zaobserwować pewne cechy charakterystyczne dla konkretnych rodzajów dziedziczeń. Przykładowo dla Single Table Inheritance była to mniejsza liczba tabeli, jednak z dużą zawartością wartości NULL. Natomiast Joined Table Inheritance pozwalał na elastyczne zarządzanie powiązanymi danymi,a Table per Class było unikanie redundancji, dzięki przechowywaniu tylko odpowiednich pól.

Kolejnym elementem porównawczym może być złożoność logiki oraz czytelność diagramów, które pozwalają budować i rozwijać dany projekt. W porównaniu do pozostałych dwóch strategii diagram STI jest bardzo prosty, jednak warto wspomnieć jakim kosztem zyskujemy tą prostotę logiki oraz czytelności. Poniżej przedstawiono porównanie miejsca zajmowanego na dysku **takiej samej** ilości danych dla różnych rodzajów mapowań.

zużyte miejsce w pamięci (kb)		
SINGLE_TABLE	**TABLE_PER_CLASS**	**JOINED**
702	497	546

Rys. 2.5 Tabela porównawcza danych eksportowych.

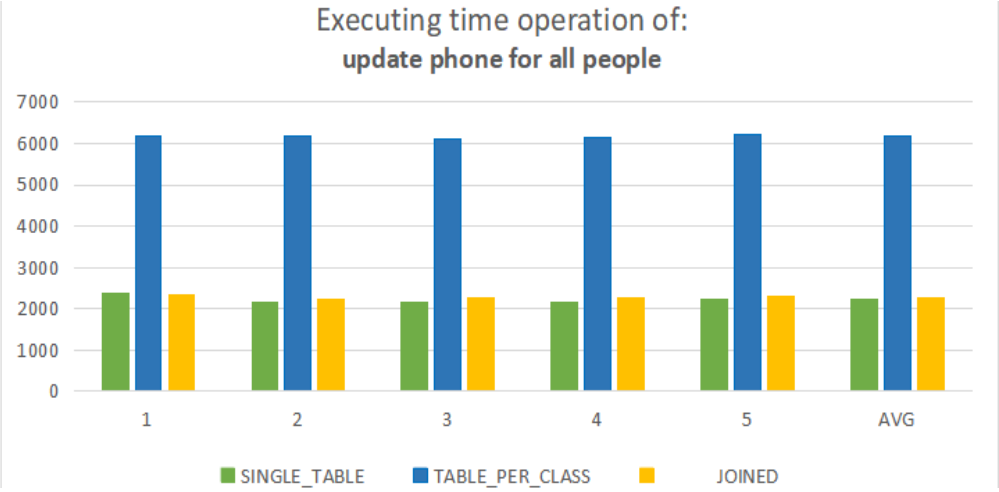
W następnych krokach zostaną przedstawione porównania wyników czasu dla różnych operacji wykonywanych za pomocą przedstawionych już wcześniej strategii. Dzięki takim działaniom można określić, która ze strategii będzie najbardziej optymalna dla danego projektu.



Rys. 2.3 Wykres dla zapisu wszystkich danych.

	SINGLE_TABLE	TABLE_PER_CLASS	JOINED
1	53144	53634	54702
2	52571	52427	52466
3	53659	53198	52825
4	52898	53053	53409
5	54120	52093	54421
AVG	**53278,4**	**52881**	**53564,6**

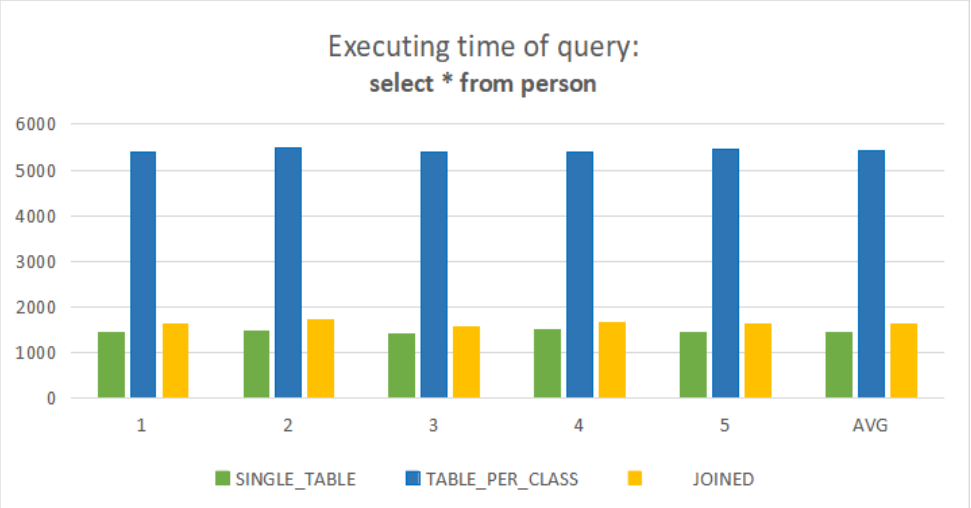
Rys. 2.5 Tabela porównawcza dla operacji zapisu wszystkich danych.



Rys. 2.3 Wykres dla aktualizacji numeru telefonu wszystkich osób w bazie.

	SINGLE_TABLE	TABLE_PER_CLASS	JOINED
1	2371	6189	2353
2	2147	6188	2227
3	2156	6098	2251
4	2167	6128	2247
5	2241	6212	2300
AVG	**2216,4**	**6163**	**2275,6**

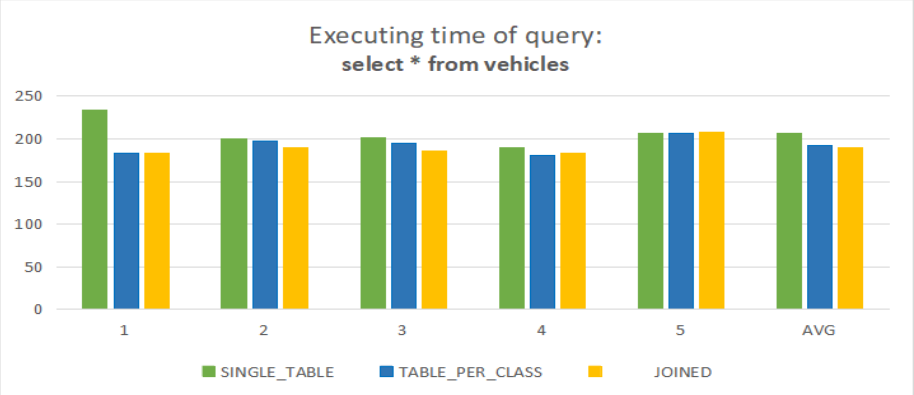
Rys. 2.5 Tabela porównawcza dla aktualizacji numeru telefonu wszystkich osób w bazie.



Rys. 2.3 Wykres dla odczytania wszystkich osób w bazie .

	SINGLE_TABLE	TABLE_PER_CLASS	JOINED
1	1432	5379	1629
2	1464	5472	1709
3	1396	5374	1562
4	1505	5382	1661
5	1457	5444	1617
AVG	**1450,8**	**5410,2**	**1635,6**

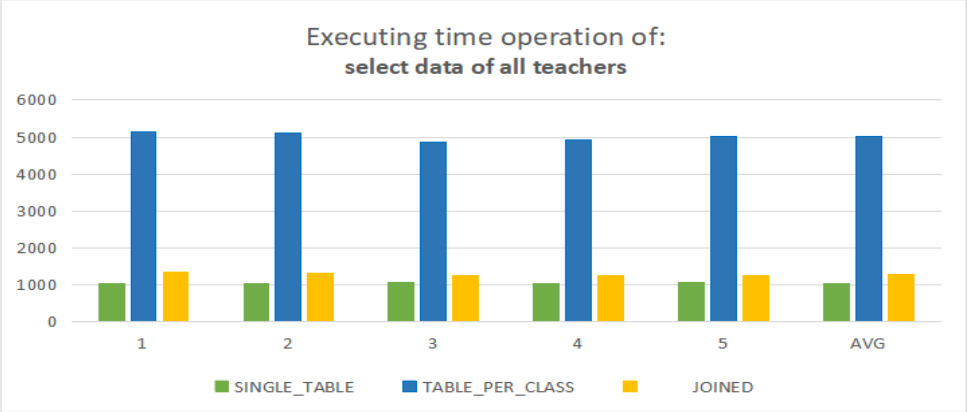
Rys. 2.5 Tabela porównawcza dla odczytania wszystkich osób w bazie.



Rys. 2.3 Wykres dla odczytania wszystkich samochodów z bazy.

	SINGLE_TABLE	TABLE_PER_CLASS	JOINED
1	233	183	183
2	200	198	189
3	201	195	186
4	189	181	183
5	206	206	208
AVG	**205,8**	**192,6**	**189,8**

Rys. 2.5 Tabela porównawcza dla odczytania wszystkich samochodów z bazy.



Rys. 2.3 Wykres dla odczytania wszystkich nauczycieli z bazy.

	SINGLE_TABLE	TABLE_PER_CLASS	JOINED
1	1032	5134	1341
2	1049	5116	1314
3	1068	4874	1247
4	1026	4917	1256
5	1077	5020	1253
AVG	**1050,4**	**5012,2**	**1282,2**

Rys. 2.3 Wykres dla odczytania wszystkich nauczycieli z bazy.

```
try {
        session.beginTransaction();
        Collection<Person> theVehicles = new ArrayList<>();
        long startTime = System.currentTimeMillis();
        theVehicles.addAll(session.createQuery("from Person where numberOfHourClasses is not null").list());
        //theVehicles.addAll(session.createQuery("from Student").list());
        //theVehicles.addAll(session.createQuery("from Professor").list());

        session.getTransaction().commit();
        long endTime = System.currentTimeMillis();
        System.out.println(endTime - startTime + "ms");

        System.out.println("\n Done !");
        System.out.println("Size of person list: " + theVehicles.size());
```

Rys. 2.5 Zapytanie odczytu wszystkich nauczycieli.

LITERATURA

[1] Najważniejsze adnotacje hibernate

https://bykowski.pl/hibernate-6-najprzydatniejsze-adnotacje-przy-modelowaniu-klas-encji/

[2] Opis hibernate.

https://bykowski.pl/czym-jest-hibernate/

[3] Opisy strategii mapowania dziedziczenia.

https://www.baeldung.com/hibernate-inheritance

[4] Opisy charakterystyki strategii mapowania dziedziczenia

(pytania były budowane na zasadzie "opis *$nazwaStrategiiDziedziczenia*").

https://chat.openai.com