# Odwzorowanie obiektów na format CSV

## Podstawowe analogie między plikami csv a relacyjnymi bazami danych:

* Kolumny i atrybuty- pierwsza linia pliku csv określa kolumny całego pliku. Specyfikacja formatu określa, że jest to element opcjonalny. Jednak, dla tego projektu, założono, że jest on wymagany. Pozwoli on określić powiązanie między atrybutami rekordów a obiektów. Wiązanie będzie odbywało się wyłącznie przez nazwę, nie będzie możliwości wykonania go przez pozycję.
* Rekord- odpowiednikiem rekordu jest linijka pliku csv, która jest zakończona znakiem końca wiersza. Jest on odwzorowywany na pojedynczy obiekt (jeśli nie uwzględniamy relacji).
* Tabela- odpowiednikiem tabeli jest zbiór linii pliku csv, czyli pojedynczy plik csv. Przyjęte jest założenie, że w jednym pliku znajdują się encje jednego typu (zawierający te same atrybuty). W świecie obiektowym jest ona reprezentowana przez klasę. Specyfikacja formatu csv wymaga, żeby liczba atrybutów w obrębie pliku była stała, co jest zgodne z tabelą relacyjnej bazy danych.
* Relacja- analogicznie do baz danych między plikami csv można określić relacje: jeden-do-jednego, jeden-do-wielu, wiele-do-jednego oraz wiele-do-wielu. Tak jak w przypadku tabel relacyjnej bazy danych, ostatnia relacja wymaga pliku pośredniczącego.

## Wartości null

Wartość null może być odwzorowywana na przykład: na tekst „null” lub podobny lub pusty ciąg znaków. Wybrano drugą możliwość. Przykładowo:

„ciąg”; 7;; „znaków”;3.14 CRL

Ma wartość null w 3. kolumnie.

## Embeded value

Cześć obiektów może być na tyle mała i prosta w inicjowaniu, że nie opłaca się zapisywać ich jako osobnych plików csv. Problem zostanie rozwiązany przez konwertery, które może rejestrować użytkownik. Dla typów prostych i niektórych klas języka Java, biblioteka będzie oferowała gotowe, nadpisywalne konwertery. Ich interface jest prosty: obejmuje metodę do przekształcenia ciągu znaków na obiekt i odwrotną. Nie będą obsługiwały one jednak dziedziczenia- dla każdej klasy trzeba zarejestrować konwerter.

## Kolekcje i tablice typów niebędących encjami

Dla uproszczenia projektu, nie zdecydowano się na obsługę typów generycznych. Klasy interface’u Collection będą służyły jednie dla określania relacji. Mapy nie będą obsługiwane, a tablice będą interpretowane jako embeded value (bez względu, czy zawierają one klasy encji, czy nie).

## Klucze główne

Klucze główne encji są niezbędne w przypadku, gdy posiadają one relacje. W językach obiektowych, obiekty są rozróżniane na podstawie miejsca w pamięci. Przy zapisywaniu ich do pliku csv ta informacja jest tracona. Sposobem, aby ją zachować jest zastosowanie kluczy, które mogą jednoznacznie identyfikować obiekt. W projekcie założono, że użytkownik ma możliwość określenia pola, które będzie pełniło rolę klucza głównego. Jednakże, będzie na nim spoczywać odpowiedzialność, aby zapewnić jego unikatowość. Dla ułatwienia projektu, nie zamierzam implementować obsługi kluczy złożonych. Ponadto, biblioteka będzie zgłaszać brak klucza obcego w encji lub encji bazowych, wtedy, kiedy będzie to konieczne np. przy relacjach. Typ klucza głównego będzie traktowany jak embeded value.

**Dziedziczenie a klucze główne. ?????**

## Dziedziczenie

Dziedziczenie jest dość złożonym problemem, który mocno rzutuje na dalsze elementy projektu. Dlatego zdecydowano się, że prace nad jego odwzorowywaniem zostaną podjęte przed obsłużeniem relacji.

W przypadku plików csv, analogicznie jak w przypadku relacyjnych baz danych, można wyróżnić trzy podejścia obsługi dziedziczenia:

1. Przechowywać całą hierarchię dziedziczenia w jednym pliku.
2. Przechowywać w osobnych plikach każdą konkretną klasę w całości.
3. Przechowywać hierarchię dziedziczenia rozdzieloną na osobne, powiązane ze sobą pliki.

Pierwsze rozwiązanie wymaga określenia, jakiego typu jest wiersz pliku. Można utworzyć dodatkową kolumnę, która może zawierać nazwę lub kod klasy. Dla projektu, dla uproszczenia, przyjęto pierwsze rozwiązanie. Zostanie zapewniony jednak odpowiedni interface, aby można było używać kodów klasy. Rozwiązanie prowadzi, do powstawania jednego, co jest wygodne, ale mocno rozbudowanego pliku csv, który posiada wiele kolumn o pustych wartościach. Jednakże, ze względu na ich zastosowania, gdzie dane chcemy trzymać w jak najmniejszej liczbie plików, takie podejście może wydawać jest najbardziej odpowiednie. Wadą takiego rozwiązania, oprócz nadmiarowości, jest ograniczenie czytelności plików dla człowieka.

Drugie podejście zakłada odwzorowanie każdej nieabstrakcyjnej klasy na osobny plik. Nie wymaga to stosowania identyfikatorów klasy, gdyż są ich wskazywanie odbywa się na podstawie nazwy pliku. Stosowanie tego rozwiązania prowadzi do postawania wielu plików, ale o mniejszej ilości kolumn. Ponadto, w przypadku braku występowania relacji, można je traktować jako niezależną całość oraz są one czytelniejsze dla człowieka.

Trzecie rozwiązanie zakłada odwzorowanie hierarchii dziedziczenia jako osobnych plików, które są ze sobą powiązane opcjonalną relacją jeden-do-jednego. Obiekty, w postaci wierszy plików csv, mogą być związane poprzez dzielenie klucza głównego, lub przy pomocy kluczy obcych. Stosowanie tego rozwiązania prowadzi do powstawia, wielu, pozbawionych redundancji plików. Są one jednak nieczytelne dla człowieka. Ponadto wymagane jest wykonywane złączeń. W przypadku relacyjnych baz danych wykonuje się je przy pomocy SQL, co jest zdecydowanie łatwiejsze niż w przypadku plików. Naiwne podejście wymagałoby analizowania wartości w postaci surowej i pobranie z wielu plików odpowiednich kolumn (ponieważ nie można utworzyć „pół” instancji klasy). Szybsze podejścia są skomplikowane i wciąż mniej wydajne niż pozostałe (tak jak w przypadku relacyjnych baz danych).

W projekcie, do realizowania dziedziczenia wybrano podejście pierwsze.

Ze względu na specyfikę projektu (biblioteka ma jedynie zapisać/odczytywać dane) problem dziadzienia jest ograniczony, co nie oznacza, że strywializowany. Rozwijanie hierarchii dziedziczenia jest ograniczone przez dane wejściowe użytkownika oraz koniecznych do zrealizowania relacji. Jednakże, w celu ograniczenia tworzenia instancji należy opracować mechanizm rejestrowania odwzorowanych (z pliku na obiekt) encji oraz ich wyszukiwania z uwzględnieniem dziedziczenia.

Kolejnym założeniem w projekcie dotyczącym dziedziczenia jest to, że strategia oraz metadane się nie dziedziczą. Oznacza to, że odwzorowywanie relacji dziedziczenia zatrzyma się na pierwszej, nieoznaczonej klasie. Będzie można to wykonać przy pomocy odpowiedniej adnotacji.

## Klucze obce:

Do utrwalenia asocjacji obiektów w plikach csv konieczne są klucze obce. Dla relacji jeden-do-jednego, odpowiednikiem atrybutu obiektu, w którym znajduje się referencja do innego obiektu, będzie kolumna z id tego obiektu. Odwracanie zależności dla relacji wiele-do-jednego, będzie polegało na oznaczeniu zbioru po stronie „wiele” odpowiednią adnotacją ze wskazaniem nazwy atrybutu drugiej strony relacji, wskazany atrybut będzie odpowiadał na kolumnie klucza obcego, w dozorowanym pliku. Nazwy plików w relacji będą rozpoznawane po nazwach encji (określona na podstawie nazwy klasy lub wskazana w adnotacji) w kodzie programu. Dodatkowo kolumna klucza w pliku csv będzie posiadała nazwę według schematu: [*nazwa kolumny@nazwa pliku, gdzie znajduje się druga strona relacji*]. Klucze obce będą wykorzystywane przy relacjach jeden-do-jednego oraz wiele-do-wielu.

## Asocjacje:

W projekcie przyjęto, że korzystający z niej użytkownik musi wskazywać obie strony relacji, przy pomocy adnotowania odpowiednych pól encji. Pozwoli to bibliotece sprawdzać spójność danych przy zapisie danych, co pozwoli uniknąć niespodziewanego jej działania, przy ponownym ich wczytywaniu. Zapewnienie takiego mechanizmu jest konieczne, ponieważ definicja formatu, oraz narzędzia do obsługi plików csv, w odróżnieniu od SZRBD, nie zapewnią takich mechanizmów.

**Jeden do jednego**

Oznaczanie tej asocjacji odbywa się przy pomocy adnotacji *@CsvOneToOne*. Posiada ona jeden wymagany argument typu string, który wskazuje na nazwę pola, będącego drugą stroną relacji. Drugi argument typu boolean jest opcjonalny i wskazuje, że dane pole ma być odwzorowany na kolumnę z kluczem obcym. Wymagane jest, aby jedna strona relacji posiadała jego wartość ustawioną na „true”. W przypadku jeśli oznaczane obu stron nie byłoby obowiązkowe, klucz musiałby znaleźć w obu plikach csv. Przykładowo: obiekt klasy A i obiekt klasy B są związane relacją jeden do jednego. Klucz obcy do pliku B.csv zapisywany jest jako kolumna w pliku A.csv oraz klasa B nie posiada odpowiednio oznaczonego pola klasy A. Jeśli nie korzystamy z dodatkowych plików metadanych (a założeniem projektu jest brak takich plików), przy wczytywaniu pliku B.csv do listy obiektów typu B tracimy informację o związanych obiektach typu A. Wymaganie oznaczanie obu stron relacji pozwoli również uniknąć sytuacji powstawania niepoprawnej relacji jeden do wielu, gdyż możliwe będzie sprawdzenie referencji obu relacji.

**Jeden do wielu / wiele do jednego**

Obie relacje są wzajemnie dla siebie drugą strona, dlatego będą traktowane jako jedna relacja. Koniec „wiele” relacji będzie odpowiadał polu, będącym kolekcją encji oznaczoną adnotacją *@CsvManyToOne*, natomiast „jeden” będzie polem typu encji oznaczonym adnotacją *@CsvOneToMany.* Obie adnotacje posiadają obowiązkowy parametr typu String, wskazujący na pole, które jest drugą stroną relacji. Projekt zakłada odwzorowanie ten relacji umieszczanie klucza obcego jako kolumny w pliku encji po stronie „jeden”.

**Wiele do wielu**

Relacja wiele do wielu po stronie obiektowej odpowiada dwóm listom oznaczonymi adnotacjami *@CsvManyToMany*. Adnotacja ta, podobnie jak poprzednie posiada obowiązkowy argument typu String, wskazujący na pole, które jest drugą stroną relacji. Odwzorowując tę relację na pliki można zastosować plik łączący, posiadający klucze obce do obu końców relacji. W tym projekcie, aby ograniczyć liczbę plików wynikowych, zastosowano inne podejście. Strona relacji jest odwzorowana na jedną kolumnę, której wiersze są listą kluczy obcych, oddzielonych separatorem pliku np. *"[1;2;3;6;8]"*. Nazwa kolumny posiada takie same zasady nazewnictwa jak kolumny zwykłych kluczy obcych.

## Warstwy abstrakcji:

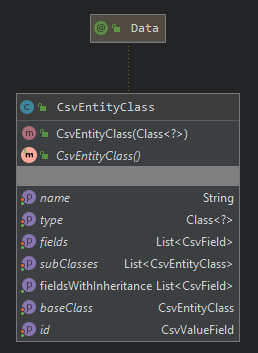
## Etapy odwzorowywania z obiektów na dane w formacie csv

## Etapy odwzorowywania z danych w formacie csv na obiekty

## Szczegółowy opis etapów

**Budowa hierarchii dziedziczenia encji**

Informacje o klasie encji są przechowywane o obiektach klasy:



**Uzupełnianie struktury klasy encji:**

Uzupełnianie kontekstu odbywa się dla każdego klasy obiektu wejściowego w następujący sposób:

**Wejścia:** klasa obiektu, obiekt, kontekst rozwiązywania klas encji

**Krok 1.** Jeżeli przekazany obiekt klasy jest równy null lub klasa nie jest oznaczona jako encja lub klasa jest typu bazowego, to zwróć null, jeżeli nie, kontynuuj.

**Krok 2.** Sprawdź czy klasa obiektu nie jest zarejestrowana w kontekście, jeżeli tak, zwróć obiekt z kontekstu, jeśli nie kontynuuj.

**Krok 3.** Utwórz obiekt typu *CsvEntityClass* oraz przypisz go do zmiennej *csvEntityClass*.

**Krok 4.** Uzupełnij pola obiektu *csvEntityClass* (analiza pól klasy encji).

**Krok 5.** Wywołaj rekurencyjnie funkcję dla klasy bazowej obiektu wejściowego, zwróconą obiekt przypisz do zmiennej *baseClassEntity*.

**Krok 6.** Jeżeli *baseClassEntity* jest różne od null to:

**Krok 6.1.** Ustaw w *csvEntityClass* jako klasę bazową obiekt *baseClassEntity*.

**Krok 6.2.** Dodaj obiekt *csvEntityClass* do listy klas pochodnych obiektu *baseClassEntity*.

**Krok 7.** Dodaj obiekt *csvEntityClass* do kontekstu rozwiązywania klas encji.

**Krok 8.** Zwróć obiekt *csvEntityClass*.

**Analiza pól klasy encji:**