# Klasy ImportedObjectBaseClass i ImportedObject

1. W klasie **ImportedObjectBaseClass**, dodałbym chroniona metodę zajmującą się czyszczeniem i korekcją wartości, dzięki temu uzyskamy możliwość większej kontroli w jaki sposób chcemy wykonywać tą operację dla poszczególnych klas z niej dziedziczących. Dodatkowo poprzez przeniesienie tego zadania do klasy przechowującej dane będziemy mogli w prosty sposób na bieżąco poprawiać dane przypisywane do poszczególnych pól.

2. W klasie **ImportedObject** usunął bym własność **Name**:

```
public string Name
{
  get;
  set;
}
```

   Przykrywa ona i tak już odziedziczone pole z klasy bazowej, co w tym przypadku nie jest konieczne, jeżeli taki efekty był zamierzony należałoby dodać słowo kluczowe **new** aby w poprawny sposób przykryć pole klasy bazowej.

3. Dodałbym pola prywatne odpowiadające własnością klas po to aby nie przypisywać wartości bezpośrednio do własności klasy. Pozwoli to na napisanie kodu wewnątrz getterów i seterów, który pozwoli sprawdzić czy dane wartości nie są **nullami**, oraz w łatwy sposób oczyścić dane podczas ich przypisywania.

# Klasa DataReader

1. Rozdzieliłbym klasę **DataReader** na dwie osobne klasy, **DataReader** oraz **DataPrinter**, po pierwsze będzie to bardziej poprawne podejście, zakładają że każda klasa powinna mieć jedno ściśle określone zadanie. Po drugie pozwoli to na wykorzystanie klasy **DataReader** do wczytania danych w momencie w którym będziemy chcieli zrobić z nimi coś innego niż wypisanie ich do konsoli.

2. Zamiast tworzyć kolejce typu IEnumerable, możemy stworzyć prywatne pole typu List przechowujące kolekcje obiektów typu ImportedObject wewnątrz klasy aby umożliwić prostsze wykonywanie operacji na kolekcji i zapobiec potencjalnemu rzutowaniu obiektów.

```
IEnumerable<ImportedObject> ImportedObjects;
```

   Na samym końcu możemy zwrócić kolekcję jako IEnumerable przy pomocy własności klasy.

3. Nie potrzebne dodanie pustego obiektu klasy **ImportedObject** podczas tworzenia nowej listy.

```
ImportedObjects = new List<ImportedObject>() { new
ImportedObject() };
```

4. Cały fragment kodu odpowiedzialny za wczytanie danych z pliku csv, można przenieść do osobnej prywatnej metody, która zwróci tablice stringów. Poprawi to czytelność kodu oraz późniejsze iterowanie po tablicy jest szybsze niż iterowanie po liście.

5. Główna pętla metody:

```
for (int i = 0; i <= importedLines.Count; i++)
```

   Możemy zacząć iteracje od 1 żeby pominąć pierwszy rekord tablicy zawierający nazwy kolumn z pliku csv. Dodatkowo warunek na zakończenie pętli powinien zostać zmieniony z „<=" na „<", ponieważ obecnie na końcu iteracji będziemy się odwoływać do nieistniejącego indeksu tablicy.

6. Przed stworzeniem nowego obiektu klasy **ImportedObject** i wypełnieniu go danymi można sprawdzić czy obecny rekord w pliku csv nie jest pusty aby nie tworzyć obiektów nie zawierających danych.

7. Wypełnianie obiektu klasy **ImportedObject** danymi:

```
importedObject.Type = values[0];

importedObject.Name = values[1];

importedObject.Schema = values[2];

importedObject.ParentName = values[3];

importedObject.ParentType = values[4];

importedObject.DataType = values[5];

importedObject.IsNullable = values[6];
```

   Jeżeli, którakolwiek z przetwarzanych linii nie będzie zawierała wszystkich danych program rzuci wyjątkiem **IndexOutOfRangeException**, ponieważ będzie się próbował odwołać do nie istniejącego indeksu tablicy. Aby temu zapobiec należy obłożyć ten fragment kodu klauzulą **try catch**, która złapie ten wyjątek i w przypadku jego wystąpienia będzie kontynuować pętle.

8. Poprzez stworzenie listy w klasie jako obiekt klasy **List** zamiast **IEnumerable**, możemy pozbyć się rzutowania tego obiektu za każdym razem kiedy chcemy dodać nowy obiekt do zbioru.

   ((List<ImportedObject>)ImportedObjects).Add(importedObject);

9. Możemy się całkowicie pozbyć pętli odpowiedzialnej za czyszczenie i korekcje danych, dzięki zmianą wprowadzonym do klas **ImportedObject** i **ImportedObjectBaseClass** operację te będą się wykonywać za każdym razem przed przypisaniem wartości do obiektu.

```
foreach (var importedObject in ImportedObjects)

{

  importedObject.Type = importedObject.Type.Trim().Replace(" ",
  "").Replace(Environment.NewLine, "").ToUpper();

  importedObject.Name = importedObject.Name.Trim().Replace(" ",
  "").Replace(Environment.NewLine, "");

  importedObject.Schema = importedObject.Schema.Trim().Replace(" ",
  "").Replace(Environment.NewLine, "");

  importedObject.ParentName =
  importedObject.ParentName.Trim().Replace(" ",
  "").Replace(Environment.NewLine, "");

  importedObject.ParentType =
  importedObject.ParentType.Trim().Replace(" ",
  "").Replace(Environment.NewLine, "");

}
```

Dzięki temu program zyska na optymalności przez to, że nie będzie musiał wykonywać dodatkowej iteracji po całym zbiorze danych.

10. Zamiast przechodzić dwoma zagnieżdżonymi pętlami po zbiorze, możemy napisać funkcję która będzie sprawdzać i korygować liczbę dzieci danego obiektu na bieżąco podczas wykonywania głównej pętli w metodzie.

    Jeżeli bardzo chcielibyśmy zostać przy tym rozwiązaniu to dzięki zmianie wewnętrznego pola klasy na typ List nie będziemy musieli rzutować go przy każdej iteracji pętli.

    ```
    var importedObject = ImportedObjects.ToArray()[i];
    ```

    Dodatkowo przy tak napisanym warunku sprawdzającym czy dany obiekt jest dzieckiem innego obiektu narażamy się na wystąpienie **NullReferenceException**, dlatego przed porównaniami powinniśmy sprawdzić czy któraś z wartości nie jest **nullem**.

    ```
    if (impObj.ParentType == importedObject.Type)

    {

      if (impObj.ParentName == importedObject.Name)

      {

        importedObject.NumberOfChildren = 1 +
        importedObject.NumberOfChildren;

      }

    }
    ```

11. Cały fragment kodu odpowiedzialny za wypisywanie danych do konsoli powinniśmy przenieść do osobnej klasy z powodów opisanych w podpunkcie 1.

12. Dodatkowo podczas wypisywania zamiast zagnieżdżać trzy pętle internujące po całym zbiorze danych, można te zbiory przed iteracją uszczuplić aby zamierały tylko dane które nas interesują w danym momencie i iterować po nich.

    W obecnym rozwiązaniu przy sprawdzaniach warunków wewnątrz pętli narażamy się na rzucenie przez program wyjątkiem **NullReferenceException** w przypadku kiedy któreś z porównywanych pól będzie miało wartość **null**.

    ```
    if (table.ParentType.ToUpper() == database.Type)
    ```

    ```
    if (column.ParentType.ToUpper() == table.Type)
    ```

    Można również przenieść wywołanie metody **ToUpper** dla pola **ParentType** do metody korygującej dane w klasach przechowujących dane, ponieważ wartość tego pola nigdy nie jest wyświetlana użytkownikowi.