Przedmiot: Wzorce Projektowe

Lab 5 – DDD

1. Pobrać z platformy e-learningowej solucję *DDD.CarRental*. W skład solucji wchodzą 3 projekty:
    - projekt typu Console (.NET Core) o nazwie *DDD.CarRental.ConsoleTest* – zawiera tzw. *Composition root* w oparciu o kontener DI (ang. *Dependency Injection*) oraz implementację prostego scenariusza testującego działanie całej aplikacji
    - projekt typu Class Library (.NET Core) o nazwie *DDD.CarRental.Core* – zawiera kod apliakcji *CarRental* w oparciu o koncepcję DDD oraz CQS (ang. *Command Query Separation*)
    - projekt typu Class Library (.Net.Core) o nazwie *DDD.SheredKernel* – zawiera klasy i interfejsy wspólne dla wszystkich projektów opartych o koncepcję DDD

    W solucji wykorzystywane są następujące pakiety:

    - Dapper
    - Microsoft.Data.Sqlite.Core
    - Microsoft.EntityFrameworkCore.Sqlite
    - Microsoft.Extensions.DependencyInjection
    - Microsoft.Extensions.Logging.Debug

2. Warstwę aplikacji zaimplementować zgodnie z podejściem CQS (ang. *Command Query Separation*).

    Uwagi:
    - W odróżnieniu od zadania z CQRS (ang. *Command Query Resposibility Segregation*), gdzie wykorzystywane było podejście oparte o *Transaction Script* i w efekcie *Command/Query Handler* zawierał logikę aplikacyjną i biznesową, w bieżącym projekcie należy zastosować podejście *Domain Model + Command/CommandHandler + Query/QueryHandler*, gdzie *Command/CommandHandler + Query/QueryHandler* zawierają tylko logikę aplikacji, logika biznesowa znajduje się w klasach *Domain Model* lub serwisach domenowych.

    a. Zaimplementować następujące operacje zmieniające stan systemu:
        i. Utwórz samochód – tworzy obiekt typu *Car* i zapisuje zmiany. Do tworzenia obiektów można zaprojektować fabrykę (zob. *StartVisit*), można też tworzyć tradycyjne (zob. *CreatePlayer*).
        ii. Utwórz kierowcę – tworzy obiekt typu *Driver* i zapisuje zmiany.
        iii. Wynajmij samochód – tworzy obiekt typu *Rental*, wprowadza dane które na tym etapie są dostępne (*Started*). Przy pomocy fabryki tworzy odpowiednią politykę darmowych minut. Rejestruje politykę w obiekcie *Driver* albo *Rental*. Ustawia status samochodu na „Wypożyczony” i zapisuje zmiany.
        iv. Zwróć samochód – zmienia status samochodu na „*Wolny*”. Odczytuje położenie samochodu z pewnego serwisu w warstwie infrastruktury.

Aktualizuje położenie samochodu (*CurrentPosition*). Oblicza kwotę do zapłaty w obiekcie *Rental* wg prostej formuły: *liczba minut wypożyczenia * cena za minutę*. Oblicza darmowe minuty na podstawie polityki darmowych minut. Dopisuje darmowe minuty do konta kierowcy.

b. Zaimplementować następujące operacje odczytujące stan systemu:
   i. Pobierz dane o wszystkich samochodach (*Car + CurrentPosition*)
   ii. Pobierz dane o wszystkich kierowcach (*Driver*)
   iii. Pobierz dane o wszystkich wynajmach (*Rental*)
   iv. Pobierz dane o konkretnym samochodzie
   v. Pobierz dane o konkretnym kierowcy
   vi. Pobierz dane o konkretnym wynajmie

Uwagi:

- Dane pobierane i zwracane przez serwis powinny mieć postać lekkich obiektów transferowych (*DTO*). Dla każdego obiektu biznesowego stworzyć obiekt typu *DTO*. Do mapowania obiektów biznesowych na transferowe warto stworzyć własne mappery lub – jeśli ktoś ma doświadczenie – wykorzystać bibliotekę np. *AutoMapper*.

3. W warstwie logiki biznesowej (DomainModelLayer→Models) utworzyć klasy modelu zgodnie z propozycją podziału na agregaty z rys. 2 lub wg własnego pomysłu.

a. Zaimplementować następujące klasy typu *Aggregate Root*:
   - *Car*
   - *Rental*
   - *Driver*

b. Zaimplementować następujące klasy typu *Value Object*:
   - *Distance* – klasa przechowuje odległość (Value) oraz jednostkę w jakich wyrażona jest odległość (Unit). Klasa potrafi porównywać odległości, dodawać, odejmować i przeliczać na inne jednostki (np. km na mile i odwrotnie)
   - *Position* – Klasa przechowuje położenie na przestrzeni XY oraz jednostkę jakiej używamy do wyrażenia położenia w przestrzeni (Unit). Klasa potrafi obliczyć odległość pomiędzy bieżącym położeniem a przekazanym przez argument zwracając wynik w postaci *Distance*. Sygnatura może wyglądać następująco:
     Distance CalculateDistance(Position d)

4. W warstwie infrastruktury (InfrastructureLayer) zaimplementować:
   a. kontekst bazy - *CarRentalDbContext*,
   b. repozytoria dla wszystkich agregatów
   c. jednostkę pracy (ang. *Unit of Work*).

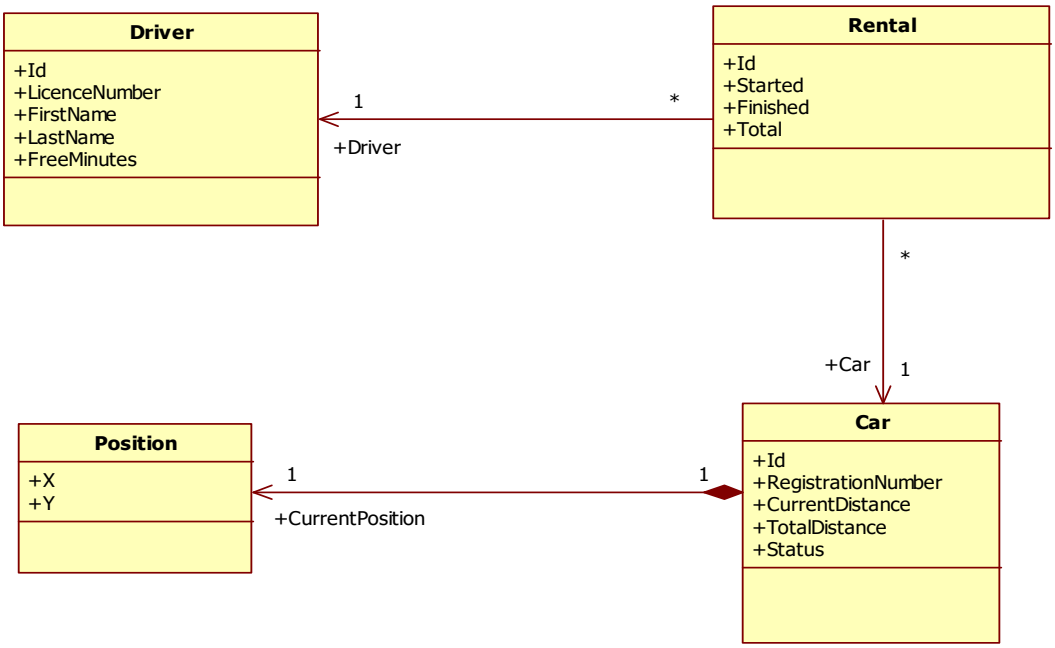
d. Usługę do odczytywania bieżącego położenia, np. *PositionService* – usługa winna generować losowo położenie samochodu.

Uwagi:

- Model klas z warstwy logiki biznesowej realizowanej zgodnie z koncepcja DDD ma wiele specyficznych cech, m.in. agregaty nie zawierają referencji do innych agregatów, klasy typu *ValueObject* nie posiadają identyfikatora, itp. Wszytko to sprawia, że mapowanie do modelu relacyjnego nie jest trywialne i nie udaje się tego dokonać jedynie przy pomocy konwencji. Aby właściwie skonfigurować model relacyjny warto skorzystać z tzw. *Fluent API*. Przykład wykorzystania *Fluent API* znajduje się w projekcie *EscapeRoom* (zob. klasy w katalogu *EntityConfiguration*).

5. W projekcie *DDD.CarRental.ConsoleTest* przygotować scenariusz testowe typu *End-To-End* w (zob. klasa *TestSuit* w projekcie *EscapeRoom*).

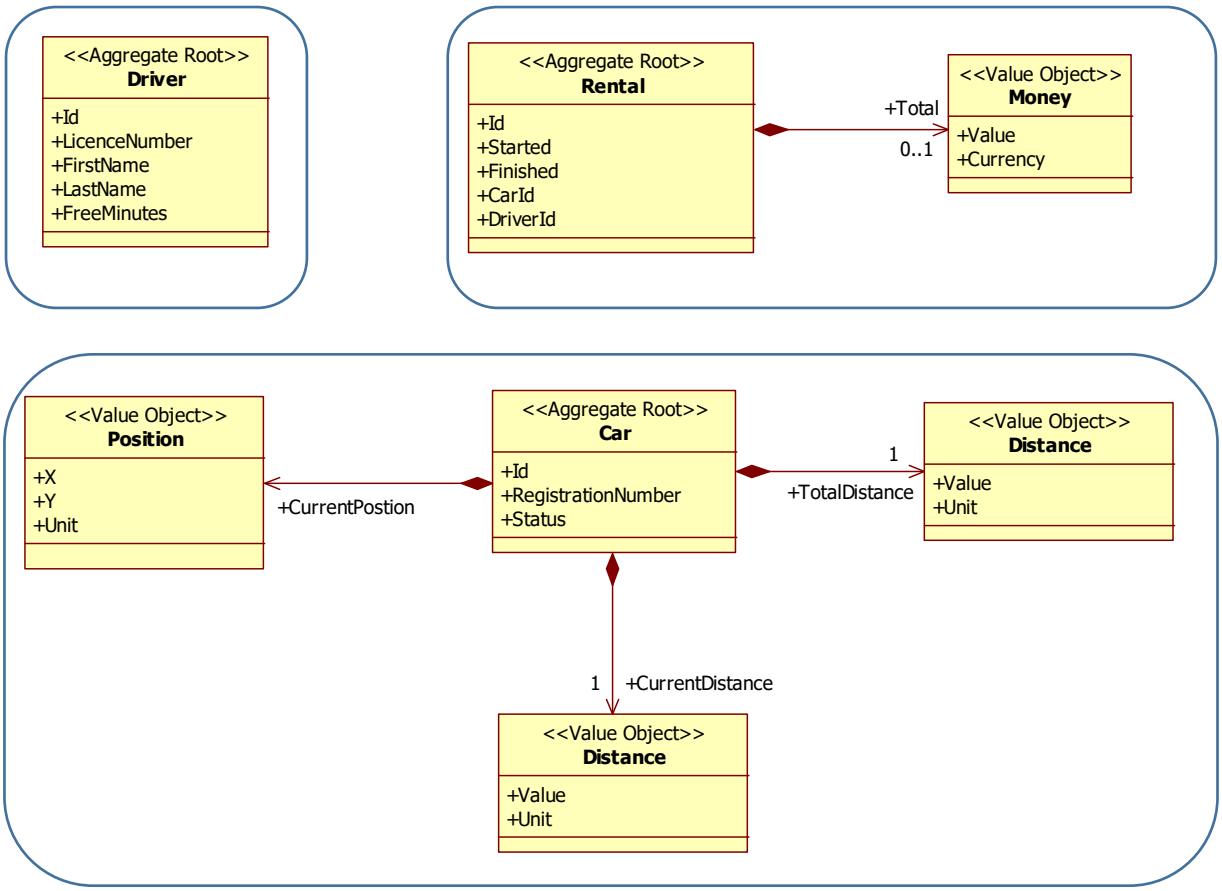
6. Rozbudować projekt CarRental o nową funkcjonalność. Implementacja nowej funkcjonalności musi wiązać się z nowym agregatem, w skład którego wchodzi minimum 2 klasy.



Rysunek 1: *Klasyczny model domeny systemu CarRental*

Uwagi:

- W klasycznym modelu domeny związki pomiędzy obiektami realizowane są przy pomocy referencji (np. obiekt *Rental* posiada referencje do obiektów *Driver* i *Car*)



Rysunek 2: *Propozycja modelu domeny dla systemu CarRental w stylu DDD*

Uwagi:

- W modelu domeny w stylu DDD związki wewnątrz agregatu realizowane są przy pomocy referencji (np. *Car* i *Position*), a związki pomiędzy agregatami - przy pomocy identyfikatorów (np. *Rental* posiada *CarId* i *DriverId*)

Znaczenie atrybutów:

Car

- RegistrationNumber – numer rejestracyjny samochodu
- CurrentDistance – bieżący przebieg, tj. od ostatniego ładowania
- TotalDistance – całkowity przebieg
- CurrentPosition – aktualne położenie samochodu
- Status – aktualny status samochodu (0 – wolny, 1, zarezerwowany, 2 – wypożyczony)

Distance

- Value – odległość
- Unit – nazwa jednostka w które wyrażona jest odległość

Position

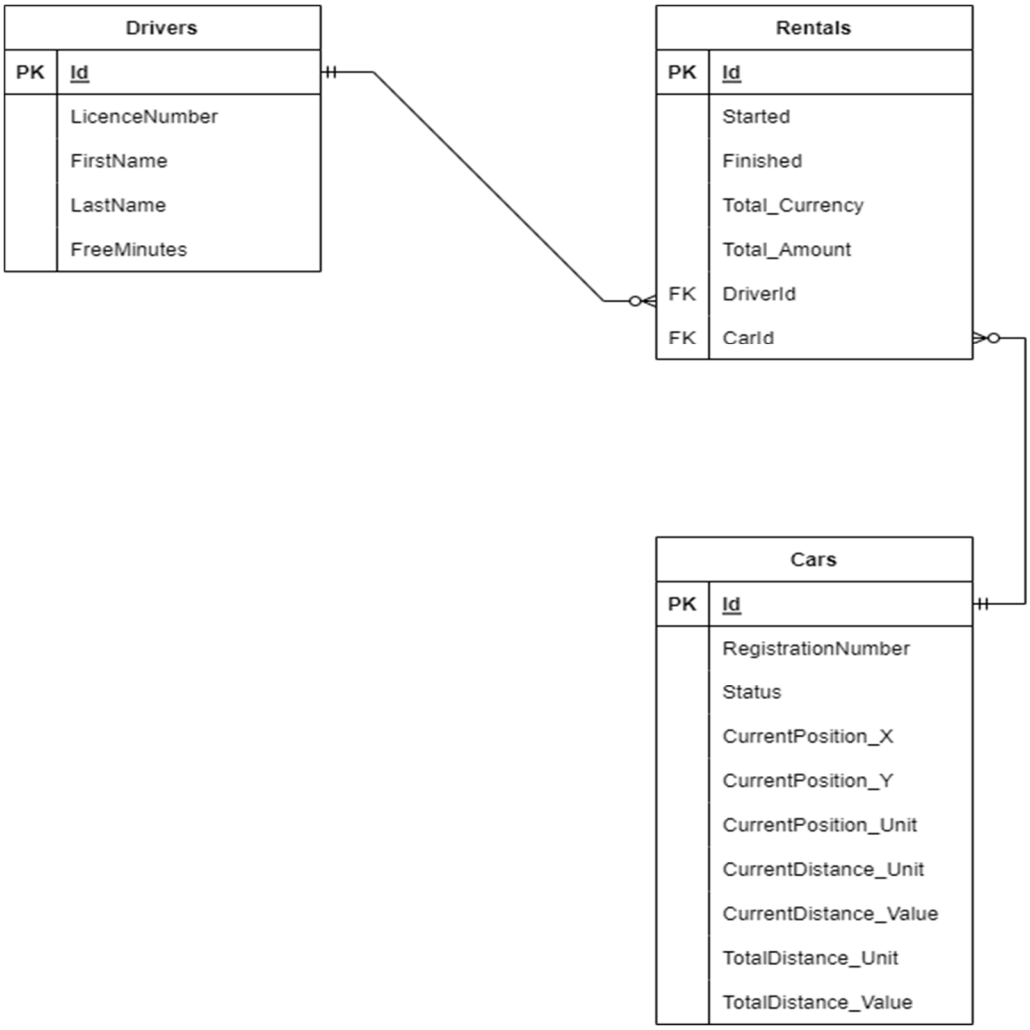
- X, Y – współrzędna X i Y położenia (przyjmujemy kartezjański układ współrzędnych, chyba że ktoś chce urealnić problem, to może przejść na położenie GPS)
- Unit – nazwa jednostki, w której wyrażone jest położenie

Driver

- LicenceNumber – numer prawa jazdy
- FirstName – imię kierowcy
- LastName – nazwisko kierowcy
- FreeMinutes – wolne minuty do wykorzystania w przyszłości

Rental

- Started – data i czas wypożyczenia samochodu
- Finished – data i czas zwrotu samochodu
- Total – całkowity koszt wypożyczenia
- CarId – identyfikator wynajmowanego samochodu
- DriverId – identyfikator wynajmującego kierowcy



Rysunek 3: *Propozycja modelu relacyjnego dla systemu CarRental*

Uwagi:

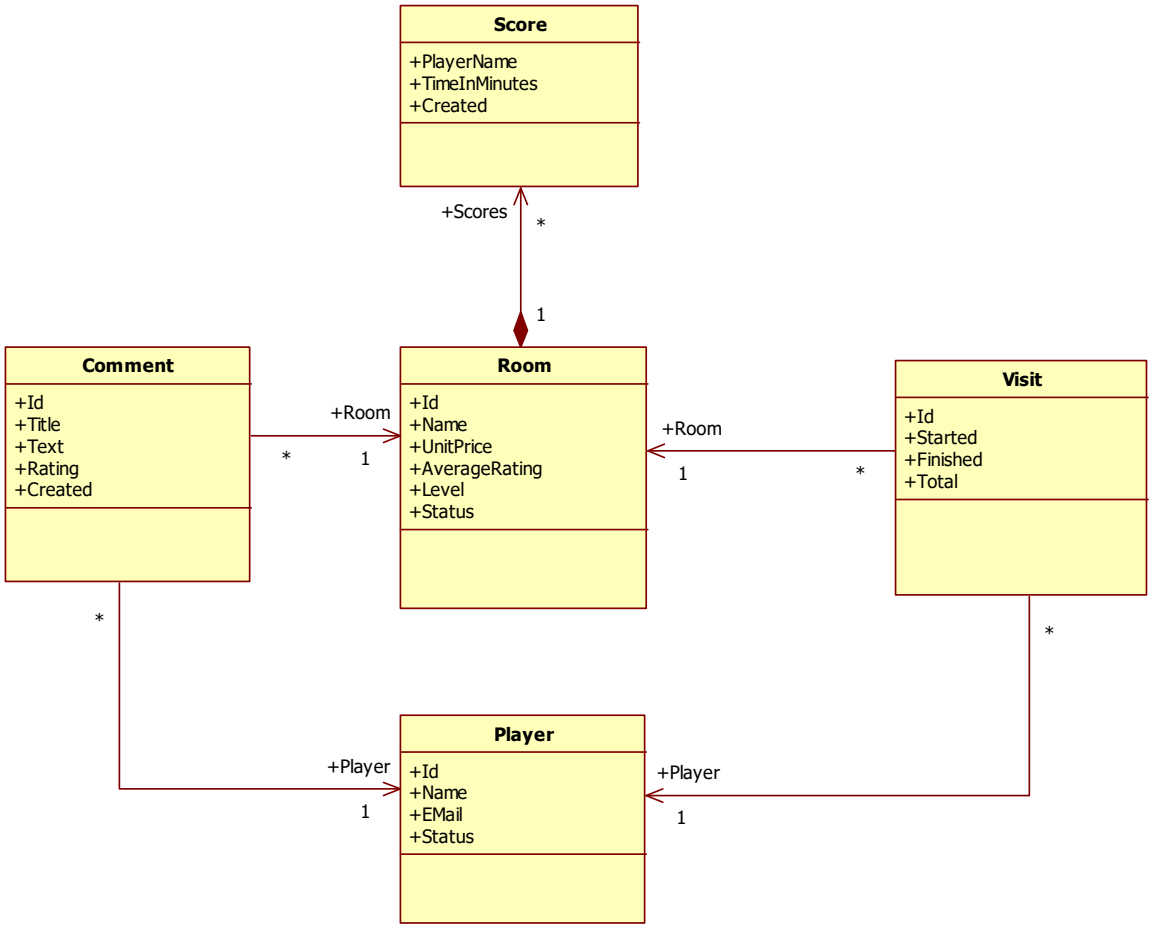
- W modelu relacyjnym związki pomiędzy tabelami są realizowane przy pomocy pary *klucz główny-klucz obcy*.
- Obiekty powiązane z korzeniem agregatu o krotności 1 zwykle mapowane są na dodatkowe pola w tabeli odpowiadającej agregatowi (np. klasy *Rental* i *Money* są mapowane na tabelę *Rentals* zawierającą również pola związane z *Money*, tj. *Total_Currency* i *Total_Amount*). Do tego rodzaju mapowania w EF służy operacja *OwnsOnes* (zob. np. klasa *RoomConfiguration* w projekcie *EscapeRoom*)

Przykład:

Poniżej znajduje się krótki opis dziedziny z przykładowego systemu EscapeRoom + klasyczny model domeny + model domeny zgodny z koncepcją DDD.

Opis dziedziny:

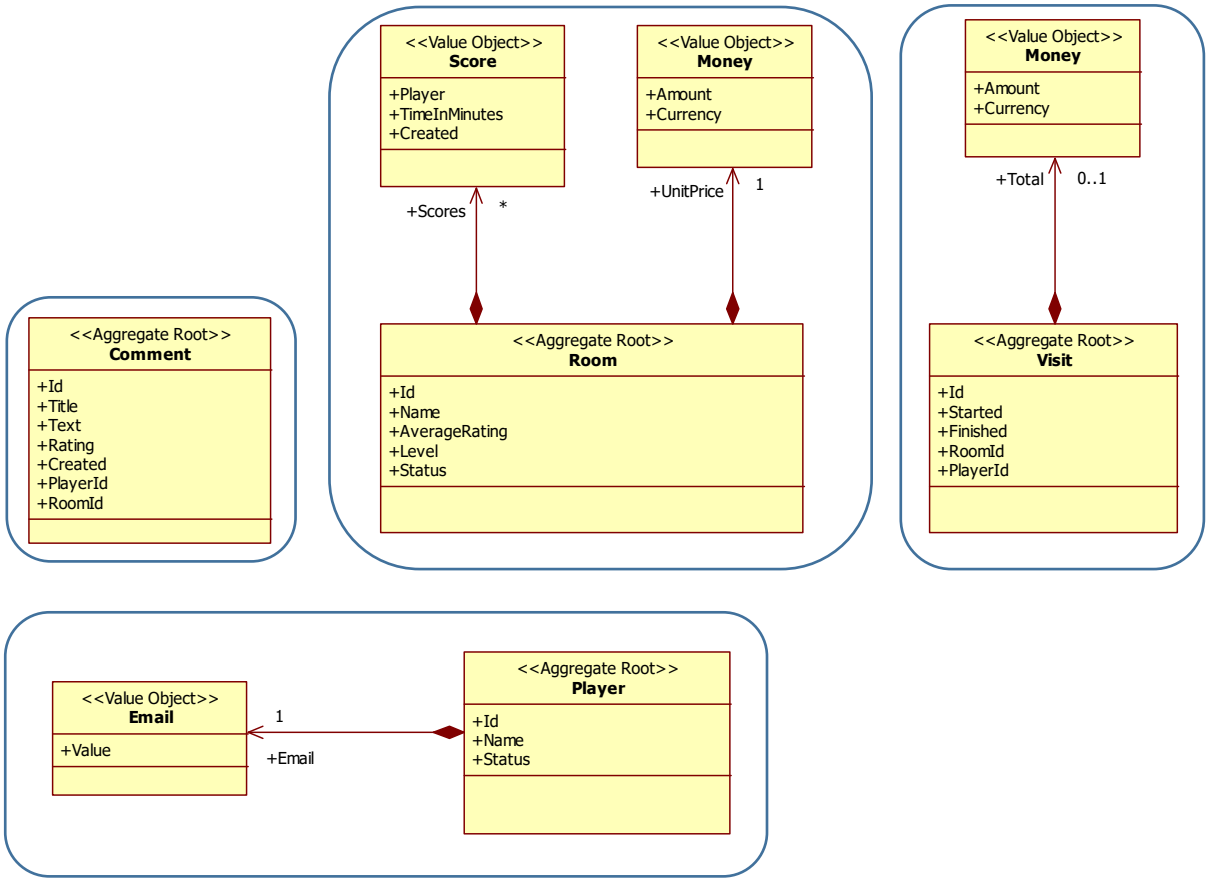
System EscapeRoom służy do obsługi pokoi zagadek. System rejestruje wizyty graczy w pokojach zagadek. Gracz może dodać komentarz po wizycie w pokoju. Pokój przechowuje informacje o 3 najlepszych wynikach. Po zakończeniu wizyty obliczana jest kwota do zapłaty. Firma stosuje polityki obliczania rabatów w zależności od klienta. Kwota do zapłaty jest zmniejszana o wielkość udzielonego rabatu.



Rysunek 4: *Klasyczny model domeny systemu EscapeRoom*

Uwagi:

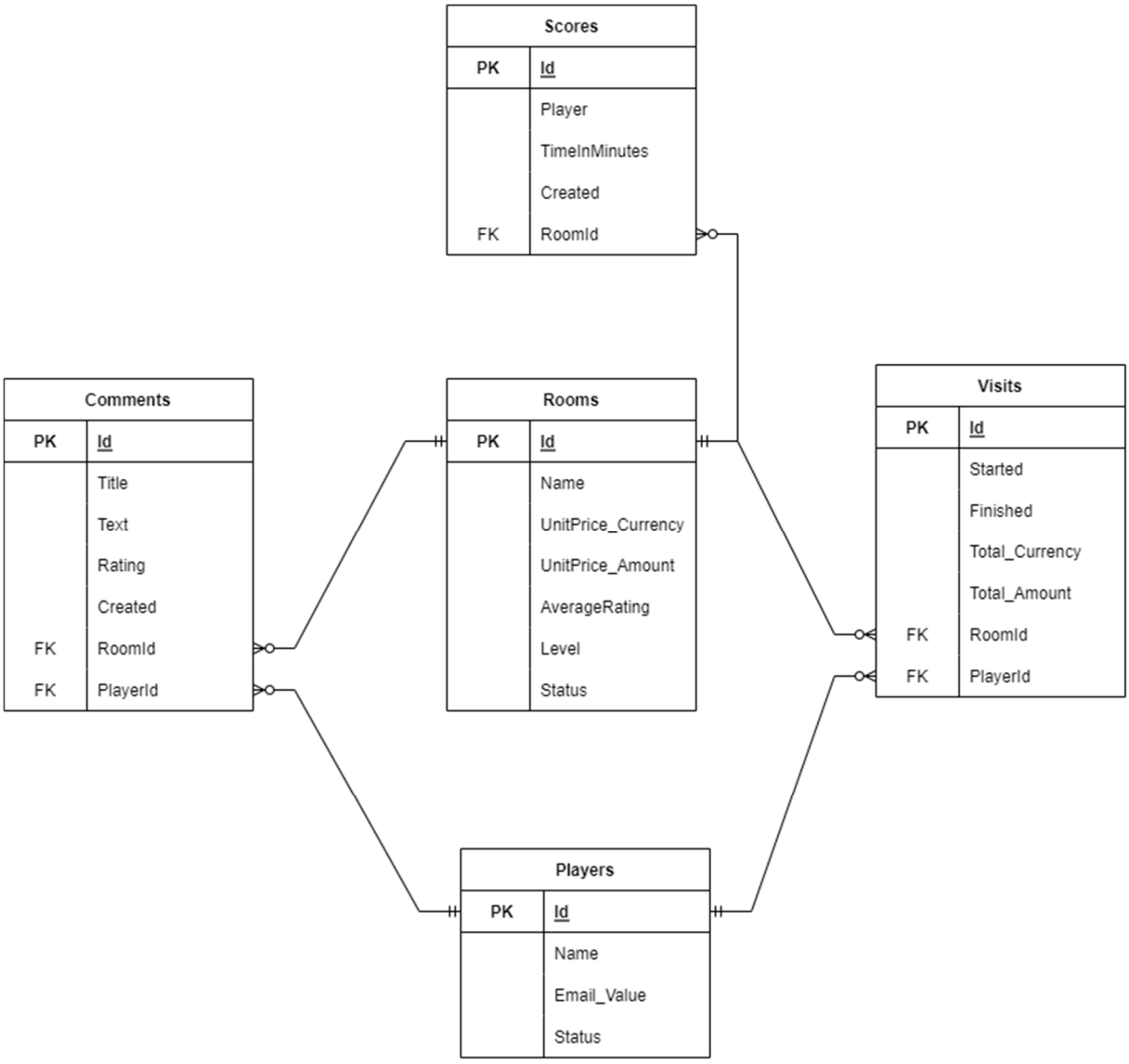
- W klasycznym modelu domeny wszystkie związki pomiędzy obiektami realizowane są przy pomocy referencji (np. obiekt *Visit* posiada referencje do obiektów *Room* i *Player*)



Rysunek 5: *Model domeny systemu EscapeRoom w stylu DDD*

Uwagi:

- W modelu domeny w stylu DDD związki wewnątrz agregatu realizowane są przy pomocy referencji (np. *Player* i *Email*), a związki pomiędzy agregatami - przy pomocy identyfikatorów (np. *Visit* posiada *RoomId* i *PlayerId*)



Rysunek 6: *Model relacyjny systemu EscapeRoom*

Uwagi:

- W modelu relacyjnym związki pomiędzy tabelami są realizowane przy pomocy pary *klucz główny-klucz obcy*.
- Obiekty powiązane z korzeniem agregatu o krotności 1 zwykle mapowane są na dodatkowe pola w tabeli odpowiadającej agregatowi (np. klasy *Room* i Money są mapowane na tabelę *Rooms* zawierającą również pola związane z *Money*, tj. *UnitPrice_Currency* i *UnitPrice_Amount*). Do tego rodzaju mapowania w Entity Framework służy operacja *OwnsOnes* (zob. np. klasa *RoomConfiguration* w projekcie *EscapeRoom*)
- Obiekty powiązane z korzeniem agregatu o krotności wiele (*) mapowane są na dodatkowe tabele (np. klasy *Room* i *Score* są mapowane na tabele *Rooms i Scores*