# Portal społecznościowy PastExplorer

Opis systemu, elementy specyfikacji, architektura

***Bartłomiej Hyży, Jakub Jaśkowiec, Michał Pieróg***
*Informatyka Stosowana, IV rok*

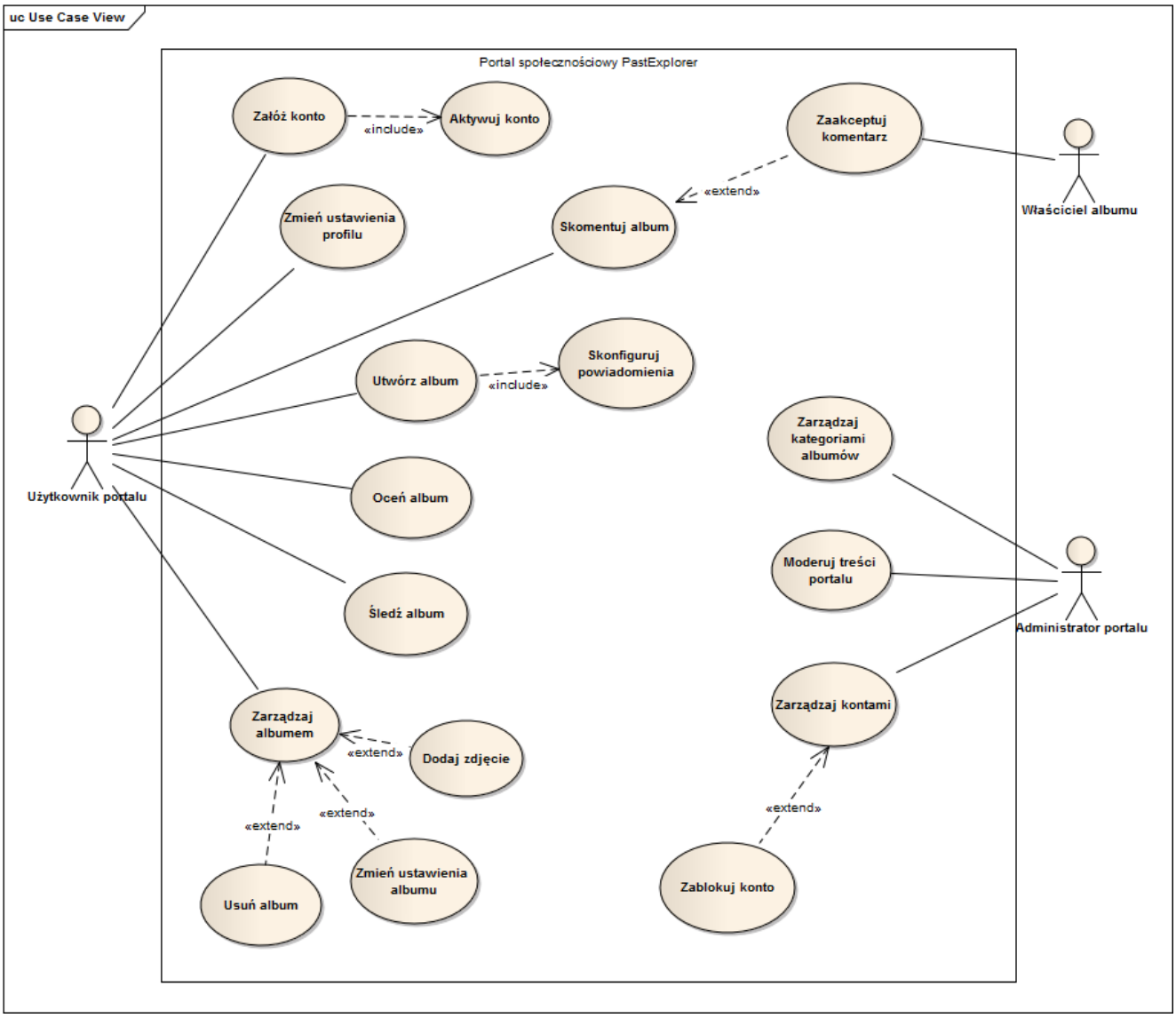
## *Opis systemu*

Aplikacja internetowa utworzona w ramach tego projektu pozwalać będzie za pomocą zdjęć umieszczanych przez użytkowników obserwować jak różnego rodzaju “rzeczy” (np. ludzie, budynki, krajobrazy) zmieniały się wraz z upływem czasu. Każdy użytkownik będzie mógł zakładać albumy będące kolekcjami zdjęć zgodnych tematycznie robionych w różnych odstępach czasu, np. codziennie, cotygodniowo, bądź corocznie.

Aplikacja będzie mocno nakierowana “społecznościowo”, tj. zawierać elementy pozwalające na tworzenie wokół serwisu społeczności użytkowników aktywnie uczestniczących w jego rozwoju, np. ocenianie i komentowanie albumów.

**Przykłady zastosowań:**

- codziennie robione zdjęcia rozwoju naszego dziecka
- cotygodniowo robione zdjęcia stanu budowy pewnego obiektu, np. stadionu piłkarskiego
- corocznie robione zdjęcia rynku w naszym mieście



## *Charakterystyka użytkowników*

Z portalu korzystać będą dwie grupy użytkowników:

- Zwykli użytkownicy –posiadać będą możliwość oglądania istniejących albumów zdjęć oraz ich ocenianie i komentowanie, a także tworzenie własnych; ponadto użytkownicy będą mogli śledzić dowolny album, tj. będą automatycznie powiadamiani o pojawianiu się nowych zdjęć
- Administratorzy – posiadać będą te same możliwości co zwykli użytkownicy, jednak dodatkowo będą odpowiedzialni za moderacją i nadzór nad działaniem serwisu; oprócz typowych zadań administracyjnych (zarządzanie kontami użytkowników, modyfikacja komentarzy, itp.) użytkownicy tego typu będą również mogli definiować kategorie, w ramach których tworzone będą mogły być albumy

## *Wymagania funkcjonalne*

Poniższa lista prezentuje funkcjonalności jakie posiadać będzie portal PastExplorer z podziałem na kategorie.

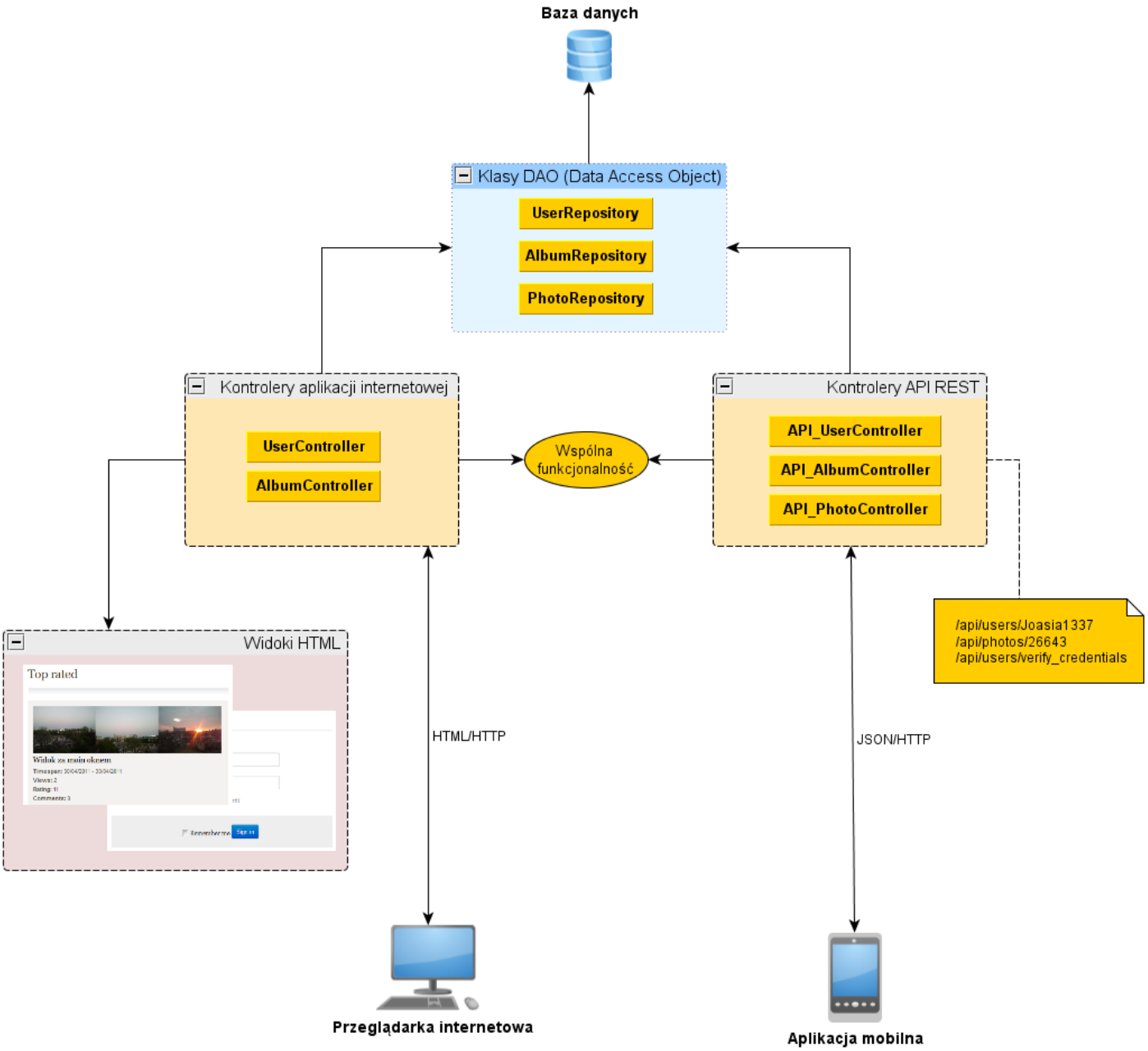
1. Profile użytkowników
    - rejestracja konta, aktywacja przez e-mail
    - logowanie/wylogowywanie
    - widok szczegółowy profilu, przeglądanie profilów innych użytkowników
    - ustawienia konta/profilu
        - dane dodatkowe (wiek, opis profilu)
        - ustawienia powiadomień
        - zmiana oraz odzyskiwanie hasła
        - usunięcie konta (wraz ze wszystkimi albumami, zdjęciami, komentarzami)
2. Albumy zdjęć
    - tworzenie nowego albumu
        - dane podstawowe (nazwa, opis, kategoria)
        - ustawienia powiadomień e-mail (odstęp czasowy pomiędzy kolejnymi powiadomieniami)
        - ustawienia poziomu prywatności
            - publiczny - dostępny dla wszystkich (również dla niezarejestrowanych)
            - prywatny - dostępny dla właściciela i wybranych osób:
                - o podanych nazwach użytkowników
                - znających hasło dostępu do albumu
        - ustawienia praw do komentowania albumu
            - automatycznie zezwalaj na komentarze
            - każdy komentarz wymaga autoryzacji
            - całkowite zablokowanie komentarzy
    - przeglądanie istniejących albumów
        - wyświetlanie informacji o albumie (nazwa, opis, ocena, komentarze, itp.)
        - przeglądarka zdjęć w albumie (atrakcyjna wizualizacja w formie „kliszy fotograficznej”)
    - wyszukiwanie albumów
    - komentowanie albumów
    - zarządzanie albumami
        - dodawanie zdjęć
        - usuwanie zdjęć
        - usunięcie całego albumu
        - zmiana ustawień albumu
3. Przeglądarka albumów
    - pasek czasu - zdjęcia wyświetlane w formie „kliszy”, przesuwane za pomocą suwaka, efektu zbliżony do filmu (obserwacje zmian obiektu wraz z upływem czasu)
    - przyciski szybkiej nawigacji (przewijanie w czasie dzień/tydzień/miesiąc/rok w przód lub w tył)
4. Elementy społecznościowe
    - ocenianie albumów (oceny dwustopniowe: podoba mi się / nie podoba mi się)
    - rankingi albumów (np. najlepiej ocenianie, najpopularniejsze, itp.)
    - śledzenie albumów (otrzymywanie poprzez e-mail powiadomień o nowych zdjęciach pojawiających się w śledzonym albumie)

5. Administracja
    - edycja/usuwanie komentarzy, zdjęć oraz kont użytkowników (profili)
    - edycja listy dostępnych kategorii albumów

## *Wymagania niefunkcjonalne*

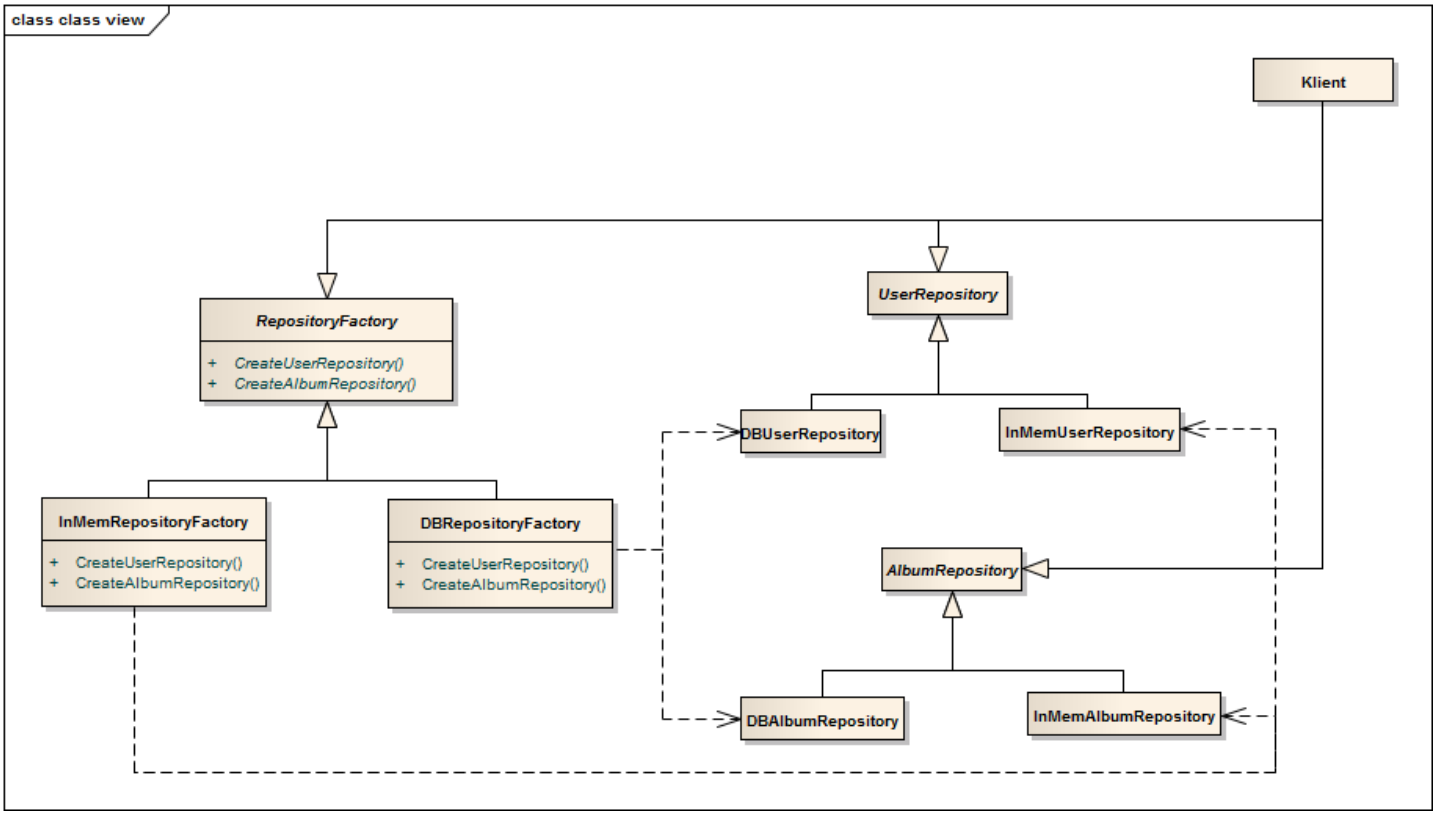
1. Zachowanie spójności pomiędzy albumami w bazie danych i zdjęciami należącymi do tych albumów, przechowywanymi oddzielnie w systemie plików
2. Łatwość użytkownia – portal powinien być intuicyjny w obsłudze i w działaniu przypominać powinien istniejące już, popularne portale społecznościowe, co ułatwi korzystanie z niego nowym użytkownikom

## *Architektura systemu*



*Wzorce projektowe*

## 1. Fabryka abstrakcyjna (Abstract Factory)




**Cel wzorca:** izolacja kodu aplikacji od rodziny klas odpowiedzialnych za zapewnienie dostępu do bazy danych, wymuszenie użycia klas dostępowych należących do jednej rodziny w celu zapewnienia spójności w użyciu magazynów danych.

Klasy abstrakcyjne *Repository definiują interfejs dostępu do tabel zawierających rekordy odpowiedniego modelu danych, np. użytkownika (UserRepository) czy albumu (AlbumRepository). Klasa abstrakcyjna RepositoryFactory definiuje interfejs fabryk pozwalających na tworzenie obiektów *Repository realizujących dostęp do danych w specyficzny, konkretny sposób. W projekcie zastosowane zostaną dwa sposoby dostępu do danych: komunikacja z motorem bazodanowym (DB*Repository) oraz przechowywanie danych w pamięci ulotnej (InMem*Repository). Drugi mechanizm dostępu uzasadniony jest koniecznością przeprowadzenia testów logiki biznesowej w odizolowaniu od warstwy bazodanowej, która może zakłócić te testy, przez co powinna zostać przetestowana osobno. Ponadto zastosowanie tego wzorca pozwoli na wygodniejszą pracę na zmieniającej się strukturze modeli obiektów bez konieczności synchronizacji schematu i zawartości bazy danych.

**Implementacja:**

*Definicja rodziny klas dostępu do bazy danych:*

```
public abstract class UserRepository {
      public User GetById(int id);
      // … inne operacje na bazie uzytkownikow …
}
public class DBUserRepository : UserRepository {
      // … implementacja operacji na uzytkownikach w bazie danych …
}
```

```
public class InMemUserRepository : UserRepository {
      // … implementacja operacji na uzytkownikach w pamieci operacyjnej …
}
```

*Definicja fabryki abstrakcyjnej i jej konkretnych implementacji (baza danych lub pamięć operacyjna):*

```
public interface RepositoryFactory {
      UserRepository CreateUserRepository();
      // …
}
public class DBRepositoryFactory : RepositoryFactory {
      public UserRepository CreateUserRepository {
            return new DBUserRepository();
      }
      // … metody fabryczne dla innych klas z rodziny DB*Repository …
}
public class InMemRepositoryFactory : RepositoryFactory {
      public UserRepository CreateUserRepository {
            return new InMemUserRepository();
      }
      // … metody fabryczne dla innych klas z rodziny InMem*Repository …
}
```

*Użycie fabryki do tworzenia obiektów dostępowych bazy danych:*

```
RepositoryFactory factory = RepositoryFactory.GetInstance();
UserRepository users = factory.CreateUserRepository();
// … użycie obiektu ‘users’ do operowania na bazie użytkowników
```

## 2. Singleton

**Cel wzorca:** zapewnienie, że pewna klasa posiada pojedynczą instancję, która dostępna jest w sposób globalny z różnych miejsc aplikacji.

W przypadku naszego systemu głównym zastosowaniem wzorca Singleton jest określenie pojedynczego miejsca, gdzie następuje tworzenie konkretnej implementacji fabryki abstrakcyjnej (opis w poprzednim punkcie) w zależności od aktualnych potrzeb. Wybór odpowiedniej implementacji może odbywać się poprzez analizę pliku konfiguracyjnego portalu, bądź poprzez makrodefinicje, które na etapie kompilacji projektu określą klasę (np. w zależności czy następuje budowanie wersji rozwojowej czy produkcyjnej), której instancja zostanie utworzona.

Zastosowanie wzorca Singleton w tym przypadku pozwala więc na uniknięcie sytuacji, gdy konieczne jest zawarcie decyzji o utworzenie konkretnej specjalizacji abstrakcyjnej fabryki w wielu miejscach, co bez wątpienia prowadzi do dużych problemów z utrzymaniem aplikacji.

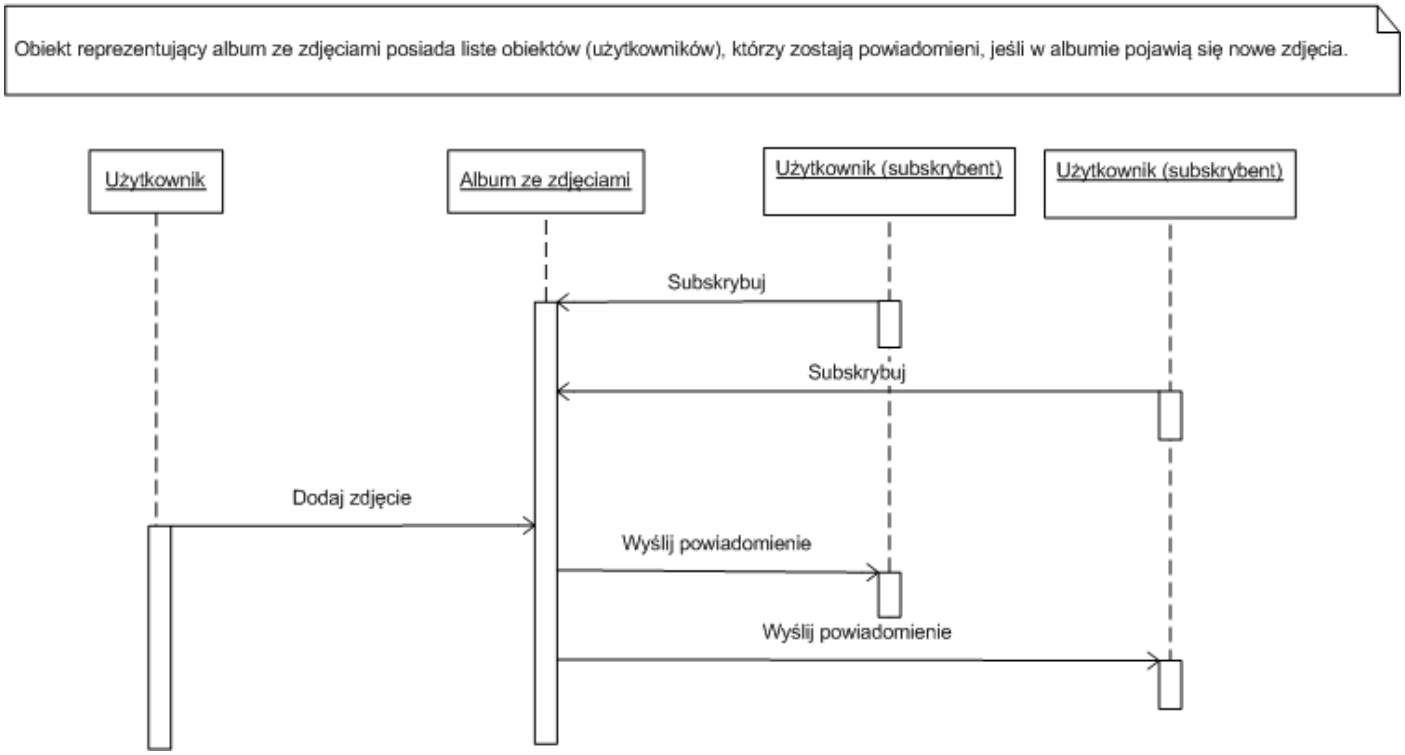
Dodatkowym powodem przemawiającym za użyciem wzorca Singleton jest chęć wprowadzenia bardziej optymalnego przydziału zasobów. W wielu przypadkach ograniczenie instancji klasy do pojedynczego egzemplarza nie jest w żaden sposób wymuszone, jednak wprowadzenie takiego ograniczenia może pomóc ograniczyć ilość alokacji kosztownego zasobu do minimum. W przypadku naszego systemu takim zasobem jest obiekt kontrolujący dostęp do bazy danych i odpowiedzialny m.in. za nawiązanie połączenia z nią.

**Implementacja:**

```
public abstract class RepositoryFactory {

      private static RepositoryFactory _instance;
```

```
    public static RepositoryFactory GetInstance() {
        if (_instance == null) {
            if (Config.GetValue("factory") == "database")
                _instance = new DBRepositoryFacyory();
            else if (Config.GetValue("factory") == "memory")
                _instance = new InMemRepositoryFactory();
        }
        return _instance;
    }

    // … inne operacje …
}
```

## 3. Obserwer


**Cel wzorca:**
Stworzenie zależności jeden-do-wielu pomiędzy obiektami, w przypadku zmiany stanu pierwszego obiektu obiekty zależne zostają automatycznie powiadomione.

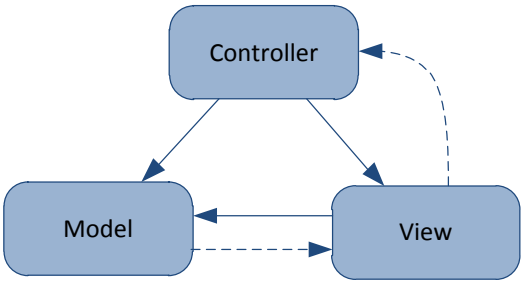
**Użycie wzorca w projekcie:**
Rezultatem użycia wzorca jest stworzenie mechanizmu pozwalającego na powiadamianie subskrybentów o zmianach w albumie.

Proces składa się z następujących elementów:
- Użytkownicy zgłaszają chęć subskrypcji albumu poprzez odpowiednią akcję na stronie;
- Obiekt reprezentujący album ze zdjęciami posiada listę użytkowników, którzy zgłosili chęć subskrypcji;

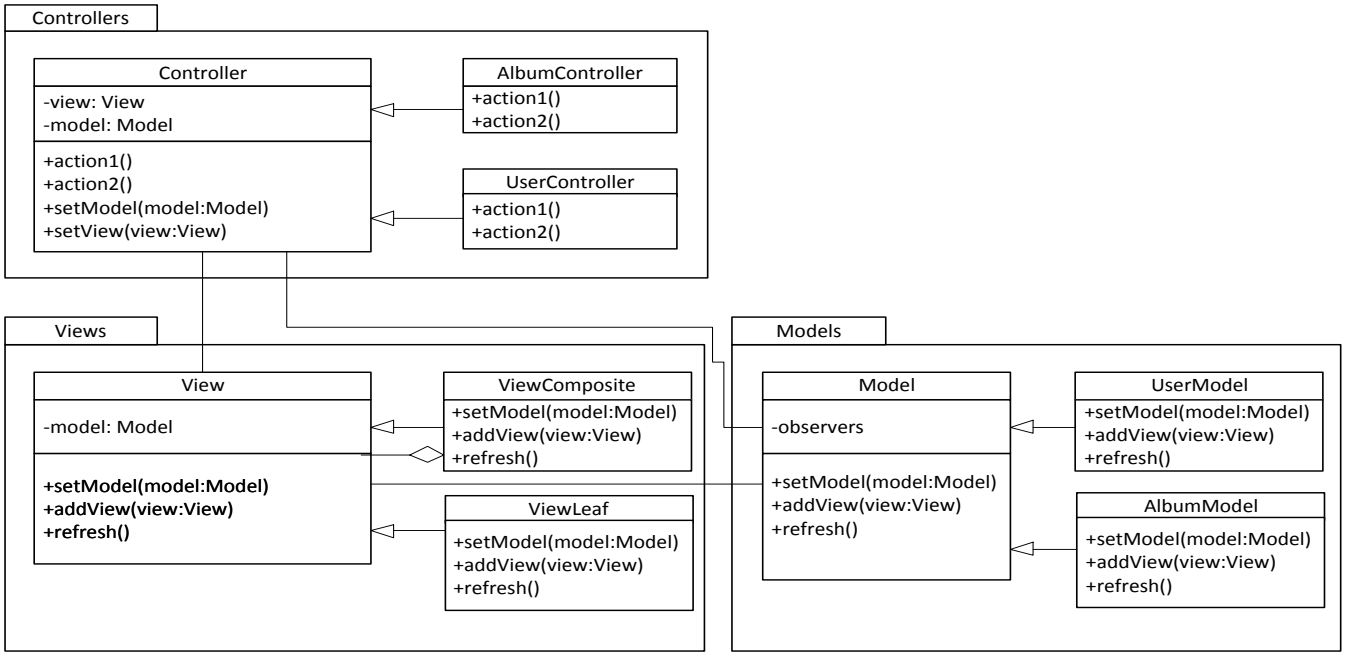
- W momencie, gdy do albumu dodane jest nowe zdjęcie, album przegląda listę swoich subskrybentów i wysyła do każdego z nich powiadomienie e-mail o nowym zdjęciu.

## 4. MVC (Model-View-Controller)

**Cel wzorca:**
pozwala na łatwy podział strukturalny aplikacji. Rozdziela warstwę logiki biznesowej od warstwy prezentacji oraz warstwy danych. Umożliwia równoległą implementację niezależnych warstw, przyśpieszając i ułatwiając tworzenie aplikacji. Zapewnia lepszą czytelność kodu i ułatwia rozwój i utrzymanie aplikacji.




W naszej aplikacji model jest pewną reprezentacją logiki biznesowej, stanowi on obiektową reprezentację danych z relacyjnej bazy danych.
Kontroler – przetwarza interakcje użytkownika, modyfikuje model, odświeża widok, przekazuje sterowanie do innego kontrolera.
Widok – stanowi graficzny interfejs użytkownika, prezentuje dane z modelu.
Widoki zbudowane są z wykorzystaniem wzorca kompozyt: mogą zawierać mniejsze powidoki. Ułatwia to tworzenie rozbudowanych interfejsów użytkownika i zapobiega duplikowaniu kodu dla różnych widoków.



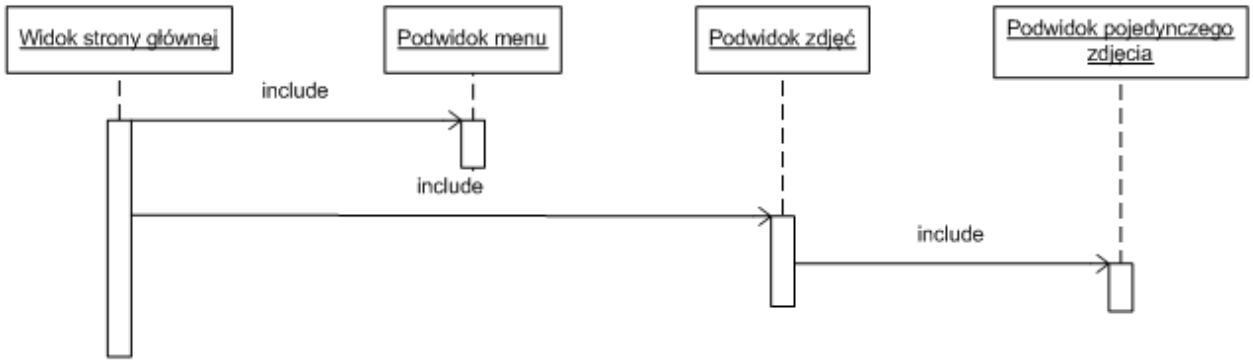

Przykład implementacji:

```
Namespace Controllers
{
      public class UserController : Controller
      {
            public ActionResult Index(){
```

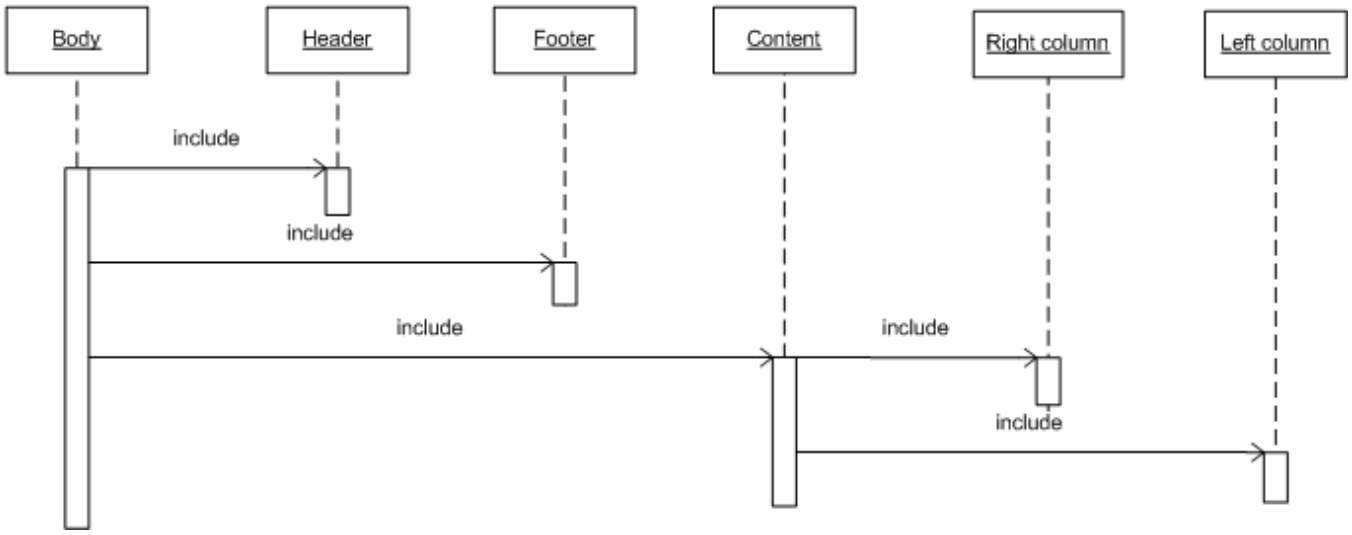
```
            //return View();
        }

        public ActionResult Create(){
            //return View();
        }

        public ActionResult Create(NewUserModel newUser){
            //Create model
        }

        public ActionResult SignIn(){
            //return View();
        }

        private void AuthenticateUser(string username, bool remember){
            //Authenticate
        }
    }
}

namespace Models
{
    public class NewUserModel
    {
        public virtual int? Id { get; set; }
        public virtual string Login { get; set; }
        public virtual string Password { get; set; }
        public virtual string Email { get; set; }
        public virtual string ActivationCode { get; set; }
        public virtual DateTime? DateOfBirth { get; set; }
        public virtual string About { get; set; }
        public virtual bool NotifyComment { get; set; }
        public virtual bool NotifyPhoto { get; set; }
        public virtual bool NotifySubscription { get; set; }
    }
}
```

## 5. Kompozyt


Przykład 1.:
Widoki aplikacji zaprojektowane są przy pomocy wzorca projektowego kompozyt. Zorganizowane są w strukturę drzewiastą i hierarchiczną.




Przykład 2.:
Struktura CSS zorganizowana jest w podobny sposób.




**Cel wzorca:**
Zbudowanie struktury obiektów w formie drzewa, które zawiera zarówno kompozycje obiektów jak i indywidualne obiekty jako wierzchołki.

Przy użyciu wzorca projektowego kompozyt możemy wykonywać te same operacje na grupach obiektów jak i na indywidualnych obiektach.

**Użycie wzorca w projekcie:**
Ten wzorzec zostanie użyty w dwóch przypadkach:

- Podczas renderowania widoków
Widok główny (przypisany do akcji kontrolera) posiada wywołania podwidoków (nazwanych partialami)

- Podczas budowy struktury strony w html/css
Każda strona składa się z elementów div, oznaczonych konkretnymi klasami. Umożliwia to łatwą stylizację elementów za pomocą języka CSS.

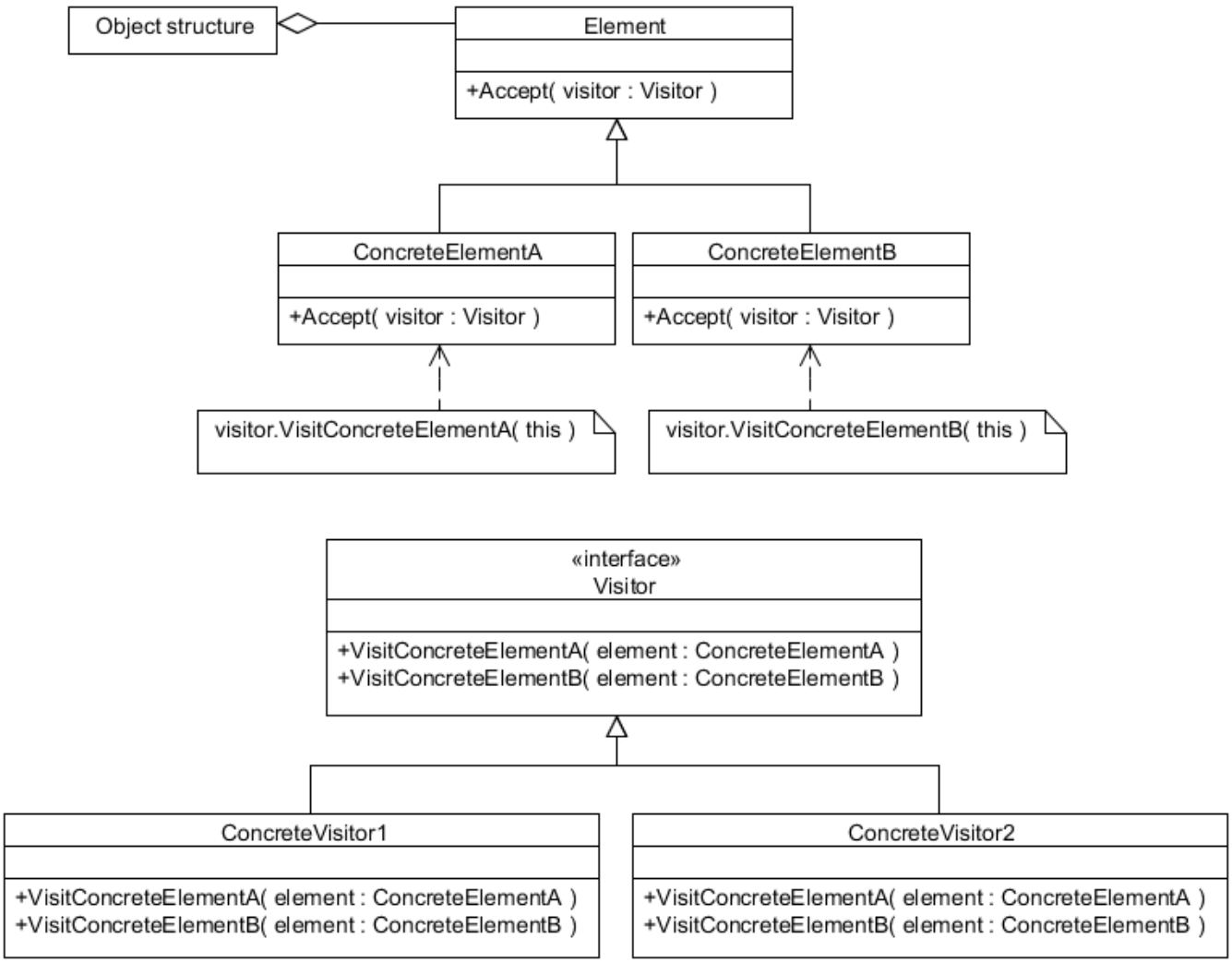
Przykład implementacji:

```
<body>
    <div id="wrapper">
        <div id="header">
           ...
        </div>
        <!-- end #header -->
        <div id="menu">
           ...
        </div>
        <!-- end #menu -->
        <div id="page">
           ...
        </div>
        <!-- end #page -->
    </div>
    <div id="footer">
      ...
    <!-- end #footer -->
</body>
```

## 6. Visitor

**Cel wzorca:**

Umożliwia odseparowanie algorytmu od struktury obiektów na których operuje. Dodaje nowe operacje do istniejących obiektów bez konieczności ich modyfikowania. Odwiedzając poszczególne elementy struktury obiektów wykonywane są odpowiednie metody.

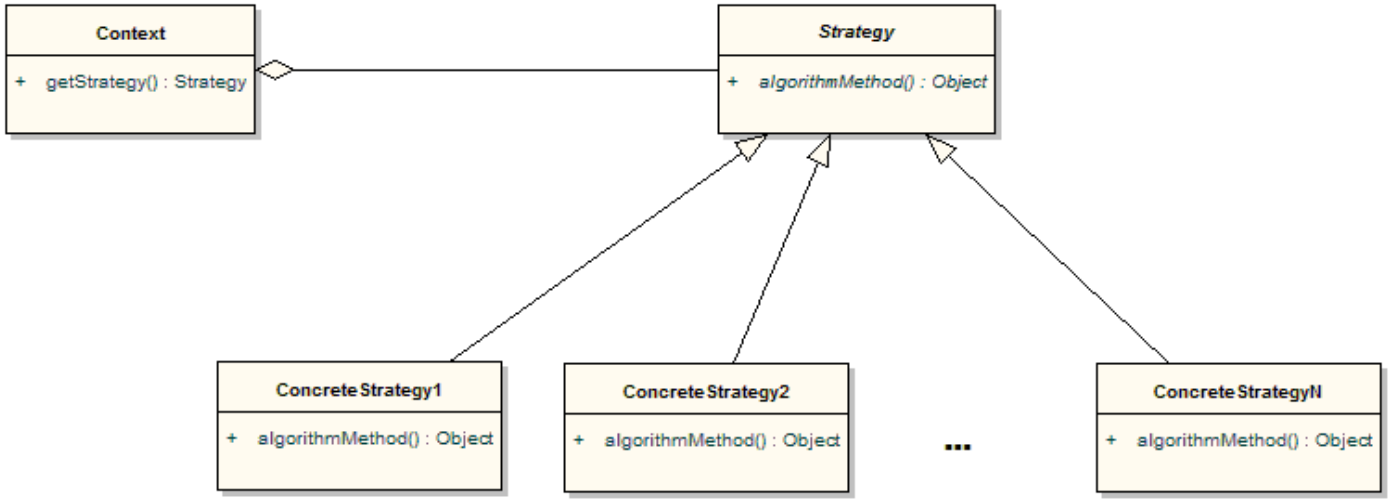



**Wykorzystanie w projekcie:**

Odwiedzający będzie przechodził po strukturze modelu i dokonywał jego walidacji. W zależności od potrzeb wykorzystane zostaną różne rodzaje Odwiedzającego, co zapewni dostosowanie rodzaju walidacji do wymagań.

## 7. Strategy

**Cel wzorca:**
Strategia umożliwia wybór algorytmu w trakcie wykonania programu. Definiuje rodzinę algorytmów i poprzez ich inkapsulację umożliwia ich wymienne stosowanie. Umożliwia różnicowanie algorytmu niezależnie od klienta, który go używa.




**Wykorzystanie w projekcie:**

Strategia zostanie wykorzystana przy autoryzacji użytkowników. W zależności od roli użytkownika zostanie dostarczona strategia udzielania dostępu do zasobów. Dostęp do zasobu będzie zależał od roli jaką użytkownik posiada w systemie.

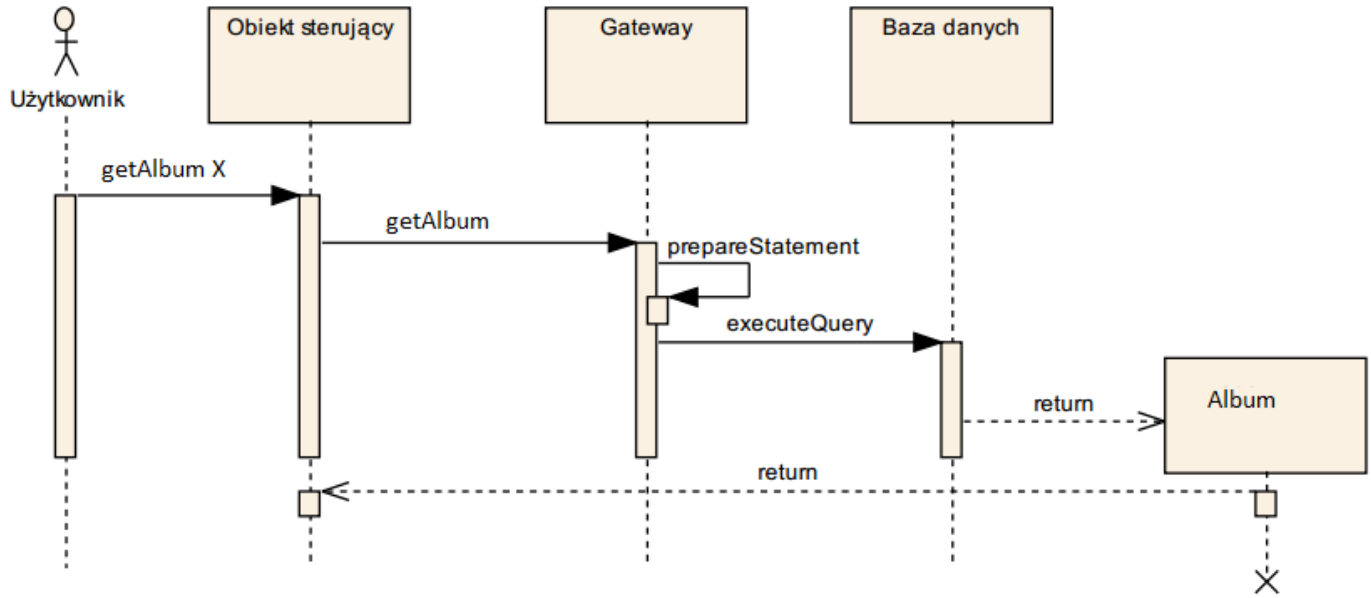
## 8. Brama danych wiersza – Row Data Gateway

**Cel wzorca:**

Obiekt służy jako Gateway (brama) do jednego rekordu w bazie danych. Istnieje jedna instancja na jeden wiersz.

**Wykorzystanie w projekcie:**

Opisany wzorzec został użyty podczas mapowania obiektów bazy danych na obiekty języka C#. Dzięki temu rozwiązaniu ułatwione zostało manipulowanie obiektami bazodanowymi.



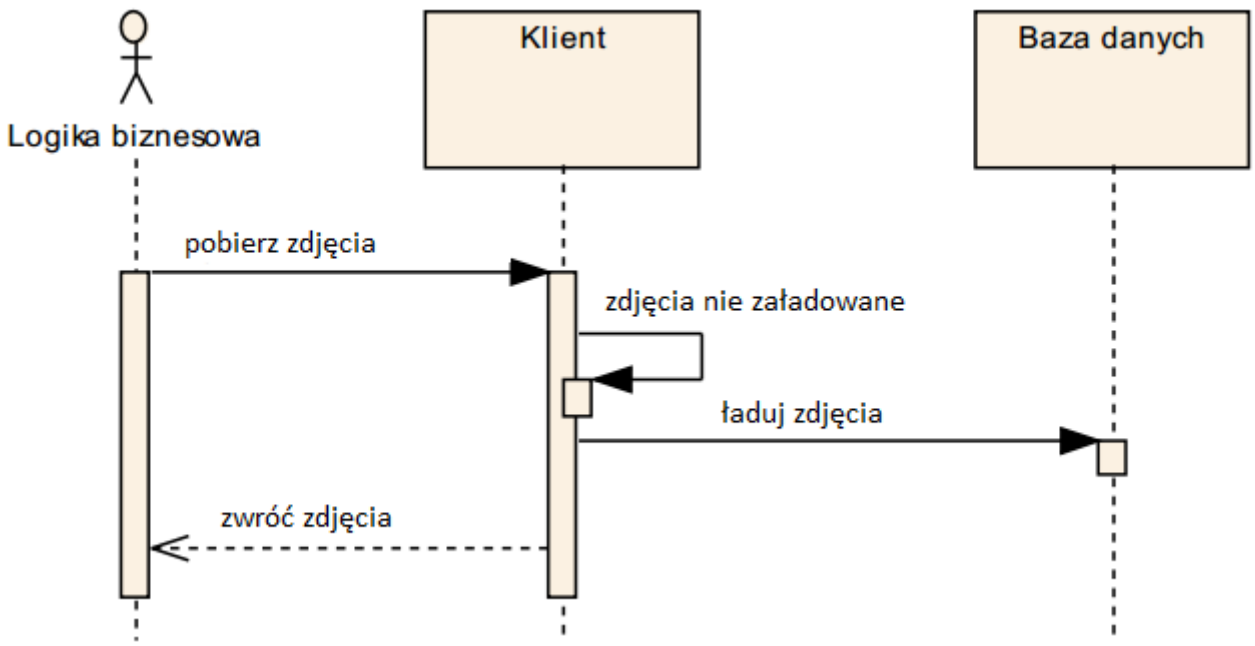
## 9. Opóźniona inicjalizacja - Lazy Load

**Cel wzorca:**

Lazy Load polega na utworzeniu pustego odwołania do obiektu, rzeczywisty odczyt z bazy danych następuje w momencie pobierania danych. Dzięki użyciu tego mechanizmu, możliwe jest: ograniczenie liczby obiektów przechowywanych w pamięci, uniknięcie złożonych zapytań SQL generujących złożone struktury danych, optymalizacja dostępu do bazy danych. Wywołanie następuje jedynie w przypadku konieczności ładowania dodatkowych danych.

**Wykorzystanie w projekcie:**

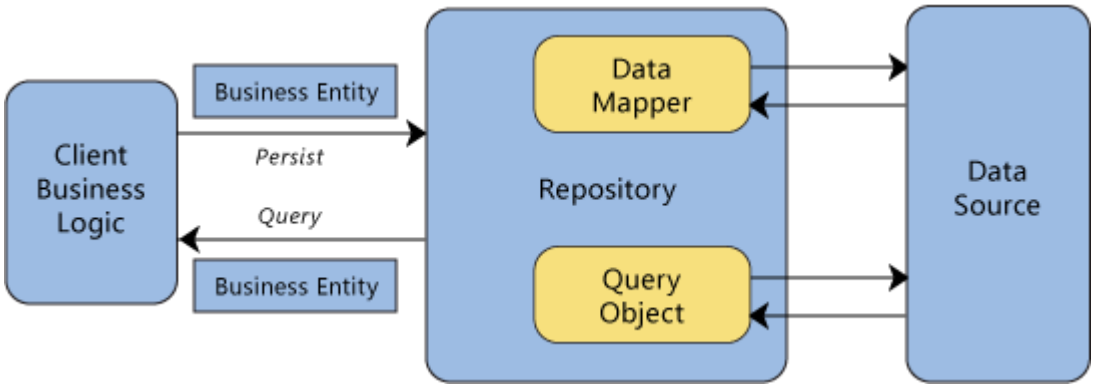
Wykorzystany w projekcie framework umożliwiający mapowanie obiektowo-relacyjne (NHibernate) został skonfigurowany tak, aby Lazy Load było możliwe.
W ten sposób ładowane są m.in. obiekty powiązane z albumem, takie jak: zdjęcia albumu, użytkownicy albumu czy kategorie albumu.




## 10. Repozytorium

**Cel wzorca:**

Repozytorium stanowi ogniwo łączące warstwę dziedziny (logikę biznesową) oraz warstwę odwzorowania danych. Wzorzec zastosowaliśmy z następujących powodów:
- dostęp do źródła danych odbywa się z wielu miejsc i chcemy rozwiązania dostarczającego centralnie zarządzany dostęp do danych, z ustandaryzowanymi regułami dostępu i logiką,
- chcemy poprawić użyteczność i czytelność kodu poprzez oddzielenie logiki biznesowej oraz kodu odpowiedzialnego za interakcję z bazą danych




**Wykorzystanie w projekcie:**

Stworzone zostały repozytoria dla albumu, fotografii oraz użytkowników.
Fragment kodu repozytorium użytkowników:

```
public class UserRepository : DataRepository<UserModel, Int32?>
{
	// inne metody

	public UserModel GetByUsername(string username)
	{
		using ( var session = GetSession() )
		{
			return session.CreateQuery( "from UserModel where Login =
:login" ).SetParameter( "login", username ).UniqueResult<UserModel>();
		}
	}

	// inne metody
}
```

## 11. Rekord aktywny (Active Record)

**Cel wzorca:**

Rekord aktywny to obiekt, który zawiera dane (z wiersza tabeli) oraz metody operujące na danych. Przy użyciu tego wzorca następuje powiązanie logiki biznesowej z operacjami na bazie danych. Wzorzec podobny do wzorca Gateway, uzupełniony o zachowania obiektów. Użycie tego wzorca umożliwia łatwiejsze manipulowanie obiektami.

**Wykorzystanie w projekcie:**

Prezentowane przykłady kodu pochodzą z modelu AlbumModel. Znajdują się w nim atrybuty rekordu wraz z walidacją, metody operujące na danych oraz statyczne metody wyszukujące dane.

```
public class AlbumModel : AbstractDataModel<AlbumModel>
    {

        // przykładowy atrybut z walidacją
         [Required]
         [StringLength(255, MinimumLength = 3)]
         public virtual string Name { get; set; }

        // przykład statycznej metody wyszukującej dane
         public static UserModel[] FindUsersByLogins(string[] logins)
         {
             UserRepository users = new UserRepository();
             UserModel[] userList = new UserModel[logins.Length];
             for (int i = 0; i < logins.Length; i++)
             {
                 UserModel user = users.GetByUsername(logins[i]);
                 if (user == null)
                     return null;
                 userList[i] = user;
             }
             return userList;
         }

         // przykład metody operującej na danych
         public virtual bool CreateTrustedUser(UserModel user)
         {
             using (var session = SessionProvider.SessionFactory.OpenSession())
             {
                 using (var transaction = session.BeginTransaction())
                 {
                     IQuery query = session.CreateSQLQuery(string.Format("insert
into trustedusers (album_id,user_id) values ({0}, {1})", Id, user.Id));
                     query.ExecuteUpdate();
                     transaction.Commit();
                 }
             }
             return true;
         }
     }
```

# *Diagram klas: Warstwa dostępu do bazy danych*

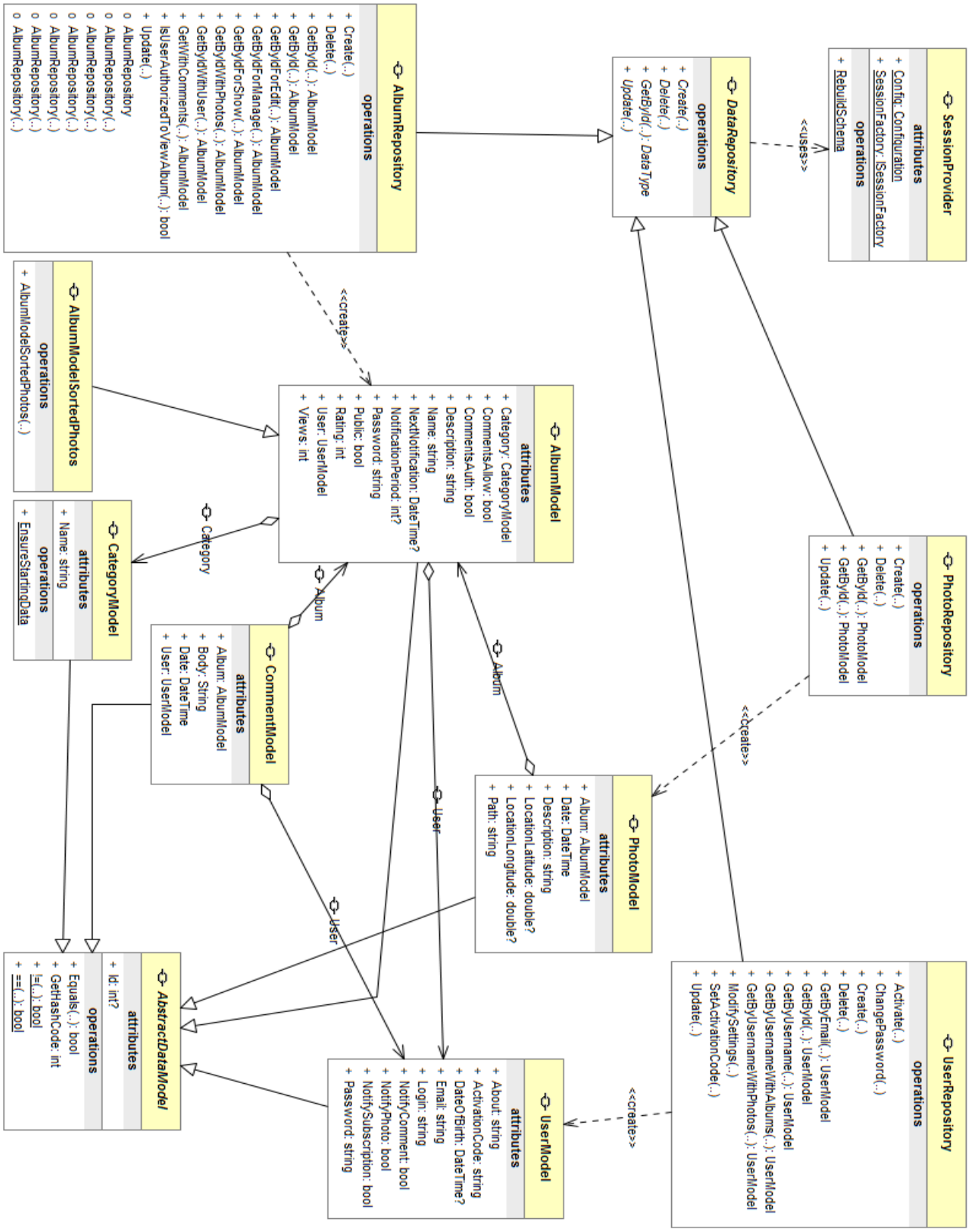
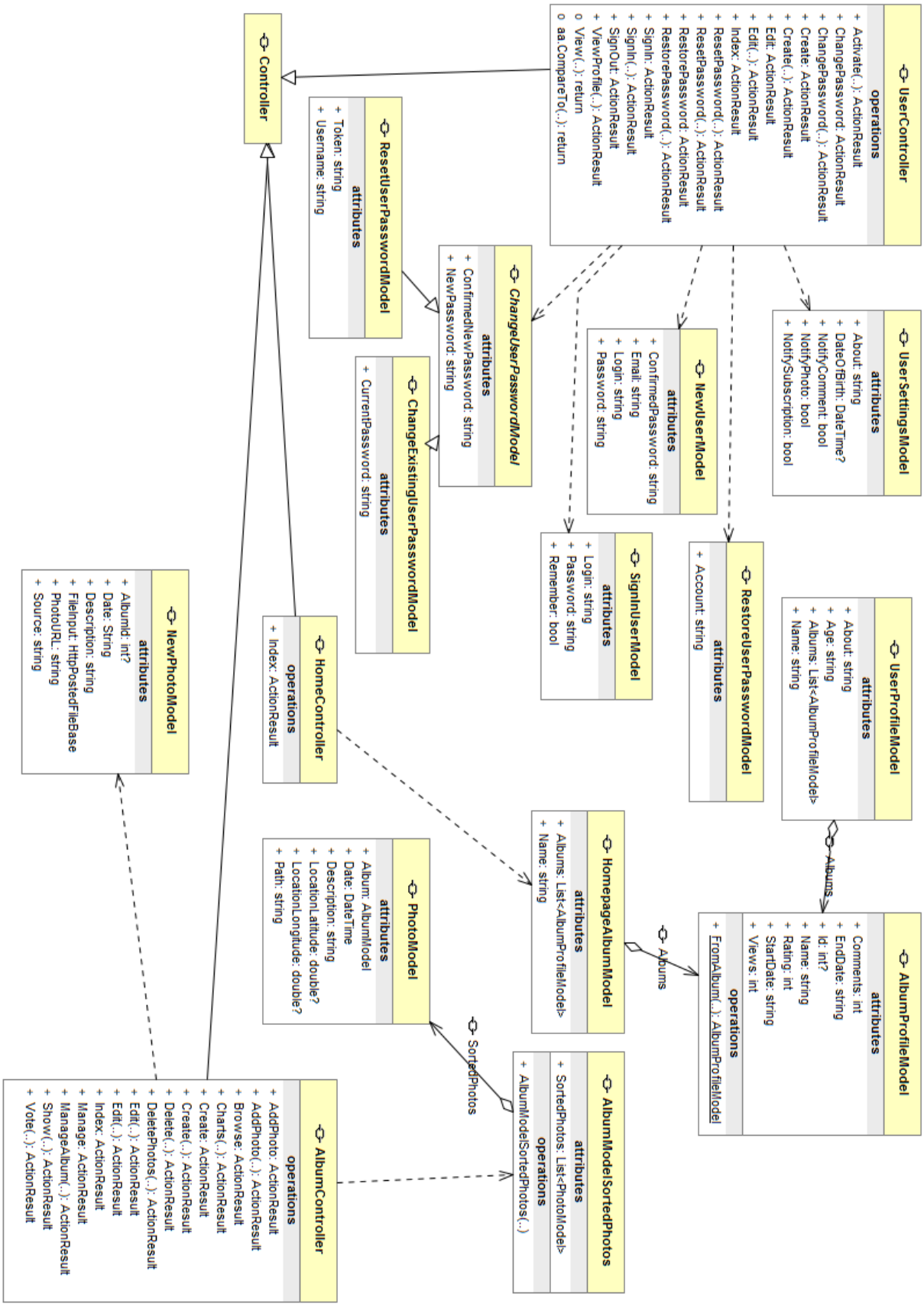
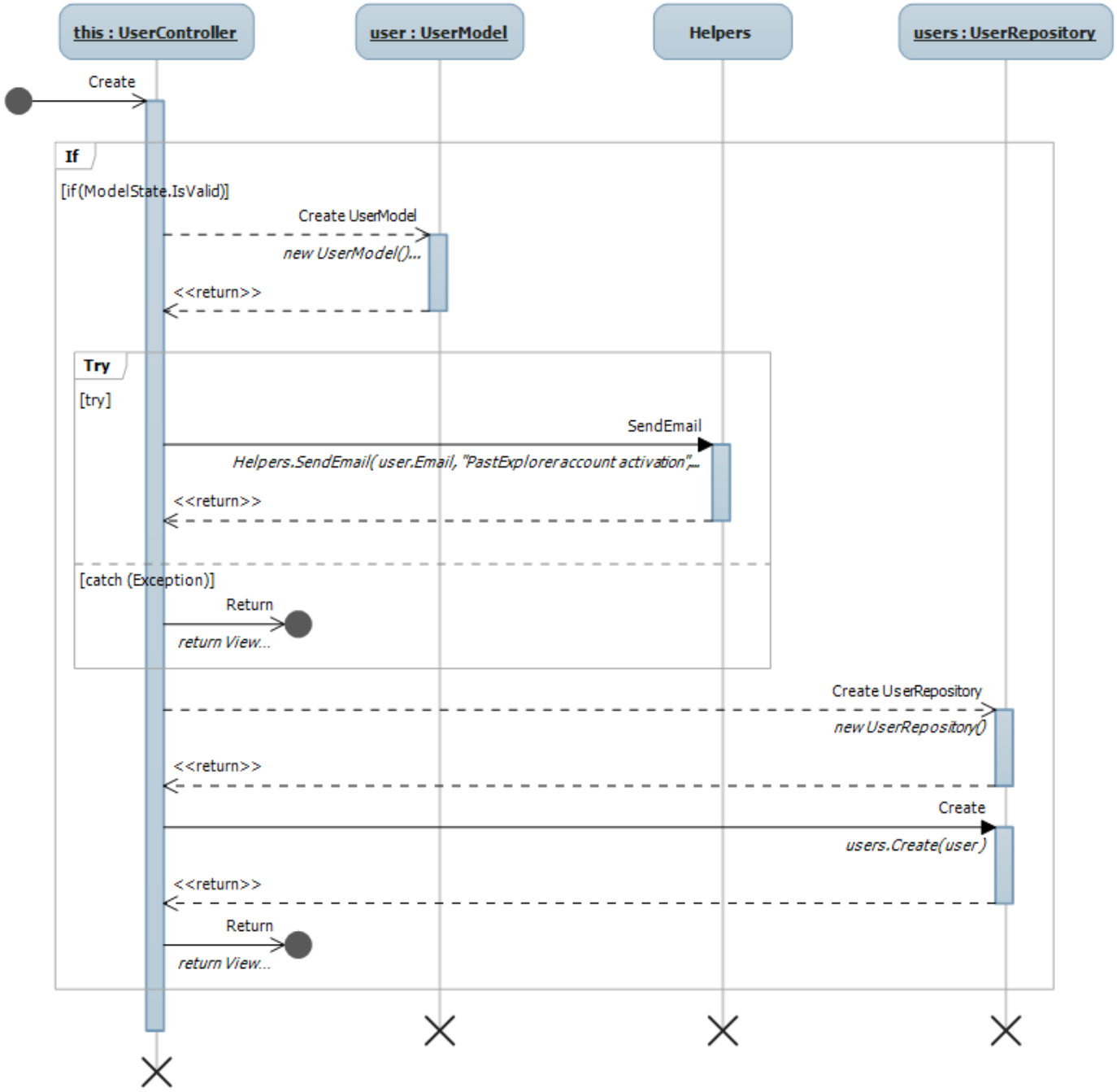


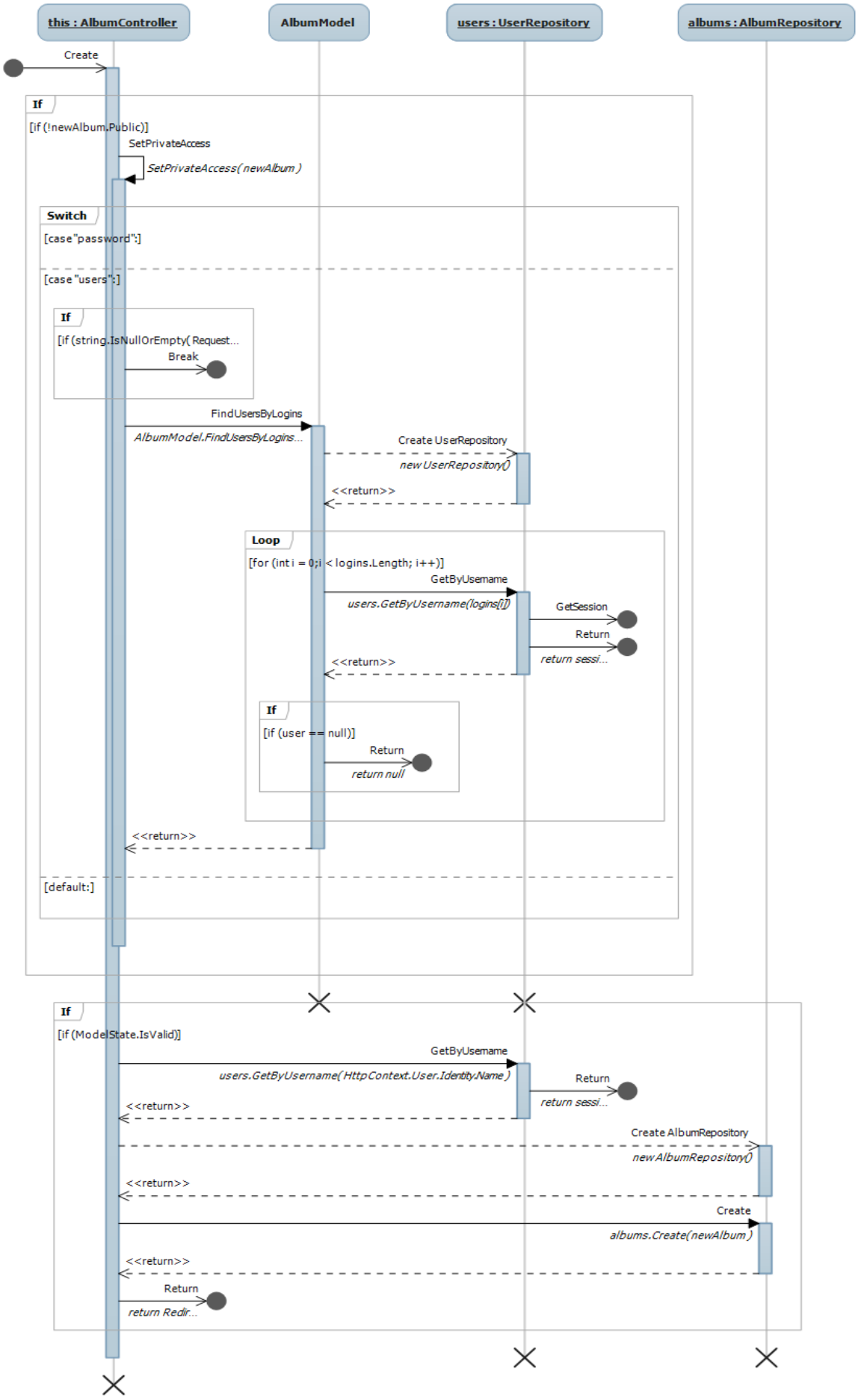
***Diagram klas: Warstwa logiki***



## *Diagram sekwencji: Tworzenie konta*



# *Diagram sekwencji: Tworzenie albumu*



## *Webservice (REST API)*

W związku z tym, że implementacja właściwego serwisu internetowego obsługiwanego za pomocą przeglądarki internetowej zmierza ku końcowi, postanowiliśmy w ramach projektu wykonać aplikację mobilną, która pozwoli wykonywać część czynności dostępnych z poziomu serwisu za pośrednictwem smartfona z systemem Android.

Aby było to możliwe, konieczna była implementacja interfejsu dostępowego do funkcjonalności serwisu. Dotychczasowa implementacja dobrze nadawała się dla użytkownika łączącego się z portalem za pomocą przeglądarki internetowej. Rezultaty jego akcji zwracane były w postaci kodu HTML renderowanego przez przeglądarkę w sposób atrakcyjny wizualnie. Taka reprezentacja jest jednak mało efektywna i ergonomiczna dla aplikacji mobilnej.

Zdecydowaliśmy się więc na implementację interfejsu dostępowego dla aplikacji mobilnej w oparciu o model REST (ang. *Representational State Transfer*). Aktualnie zyskuje on coraz większą popularność i zaczyna wypierać bardziej standardowe rozwiązania oparte o SOAP/XML/WSDL, które charakteryzują się stosunkowo dużą złożonością konfiguracji i formalizmem.

**Funkcjonalność aplikacji mobilnej:**

- autentykacja użytkownika poprzez mechanizm HTTP Basic Authentication (HTTPS?)
- wyświetlanie listy albumów użytkownika
- wyświetlanie zdjęć w albumie użytkownika
- robienie zdjęcia z poprzednimi zdjęciami jako przezroczystymi makietami, wysyłanie do serwisu
- dołączanie danych geolokalizacyjnych do zdjęć

**Zasoby obsługiwane przez REST API:**

- użytkownicy (/api/users)
- albumy (/api/albums)
- zdjęcia (/api/photos)

**Pełny listing aktualnie zaimplementowanego API wraz z przykładowymi wywołaniami:**

- zwracanie informacji o użytkowniku

**`GET /api/users/JanekKowalski`**

```
{
    "ok": true,
    "data": {
        "id": 3,
        "username": "JanekKowalski",
        "date_of_birth": {
            "day": 1,
            "month": 3,
            "year": 1989
        },
        "about": "Jestem z Krakowa. Lubię jeździć na rowerze.",
        "albums": [
            "http://localhost:3518/api/albums/5",
            "http://localhost:3518/api/albums/18"
        ]
    }
}
```

- zwracanie informacji o albumie

**`GET /api/albums/5`**

```
{
    "ok": true,
    "data": {
        "id": 5,
        "name": "Moja twarz",
        "description": "Jak zmieniałem się w czasie",
        "category": "People",
        "owner": "JanekKowalski",
        "is_public": true,
        "rating": 10,
        "views": 1234,
        "photos": [
            "http://localhost:3518/api/photos/1",
            "http://localhost:3518/api/photos/2",
            "http://localhost:3518/api/photos/3",
            "http://localhost:3518/api/photos/4",
            "http://localhost:3518/api/photos/5",
            "http://localhost:3518/api/photos/6",
            "http://localhost:3518/api/photos/7"
        ],
        "comments": []
    }
}
```


- zwracanie informacji o zdjęciu

**`GET /api/photos/2`**


```
{
    "ok": true,
    "data": {
        "id": 2,
        "album": "http://localhost:3518/api/albums/5",
        "date": {
            "day": 30,
            "month": 4,
            "year": 2011
        },
        "description": "Oto ja",
        "image": "http://localhost:3518/Static/photos/photo_2012051022450267.jpg",
        "thumbnail": "http://localhost:3518/Static/photos/photo_2012051022450267_mini.jpg",
        "latitude": 25.21356,
        "longitude": 34.12357
    }
}
```


- wysyłanie zdjęcia

**`POST /api/photos`**
**`(... HTTP Body: zdjęcie, metadane ...)`**

## *Autentykacja użytkownika (REST API)*

Istnieje wiele rozwiązań problemu weryfikacji, czy użytkownik jest tym, za kogo się podaje (autentykacja) i czy posiada dostęp do określonego zasobu (autoryzacja). Przed przystąpieniem do implementacji API REST analizie poddane zostało pare rozwiązań, ostatecznie jednak zdecydowaliśmy się na wbudowany w HTTP mechanizm Basic Authentication z następujących powodów:

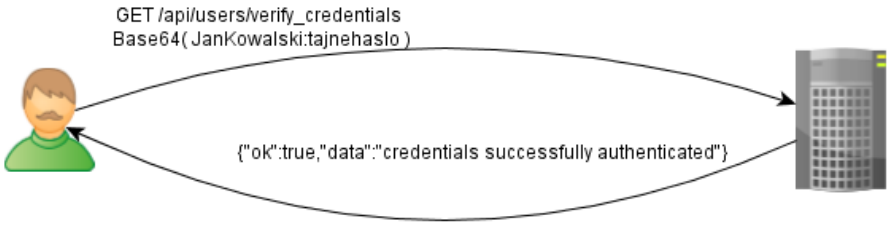
- prostota, autentykacja polega na dodaniu jednego nagłówka do zapytania HTTP z zakodowanymi w Base64 danymi użytkownika (login + hasło)
- brak konieczności przetrzymywania po stronie serwera żadnego stanu zalogowania użytkownika, jak w przypadku sesji, co naruszało by postulamy modelu REST
- jest to mechanizm wbudowany we wszystkie przeglądarki – duża uniwersalność

Metoda ta jednak posiada pewne wady:

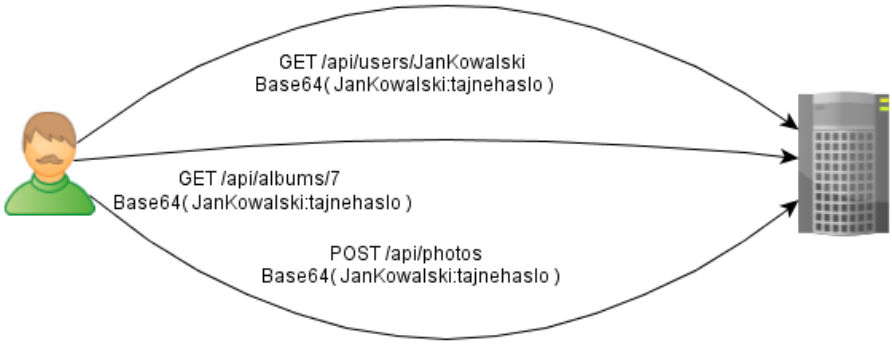
- dane użytkownika (w tym hasło) przesyłane są w zapytaniu HTTP tekstem jawnym, stąd konieczność komunikacji przez HTTPS (szyfrowanie SSL/TLS)
- dane użytkownika przesyłane są wraz z każdym zapytaniem, co może stwarzać niepotrzebne zagrożenie; bezpieczniejszym rozwiązaniem może być modyfikacja mechanizmu, polegająca na przesyłaniu skrótu hasła obliczonego funkcją hashująca, oraz odpowiednia interpretacja tego skrótu po stronie serwera (tracimy jednak w ten sposób możliwość współpracy z przeglądarkami internetowymi, które nie potrafią obsłużyć takiego mechanizmu; w naszym przypadku nie jest to problemem, z uwagi na to, że API przeznaczone jest głównie dla aplikacji mobilnej, którą sami zaimplementujemy)

**Autentykacja użytkownika odbywa się dwustopniowo:**

1. Weryfikacja danych logowania wprowadzonych przez użytkownika




2. Autentykacja przy każdym zapytaniu



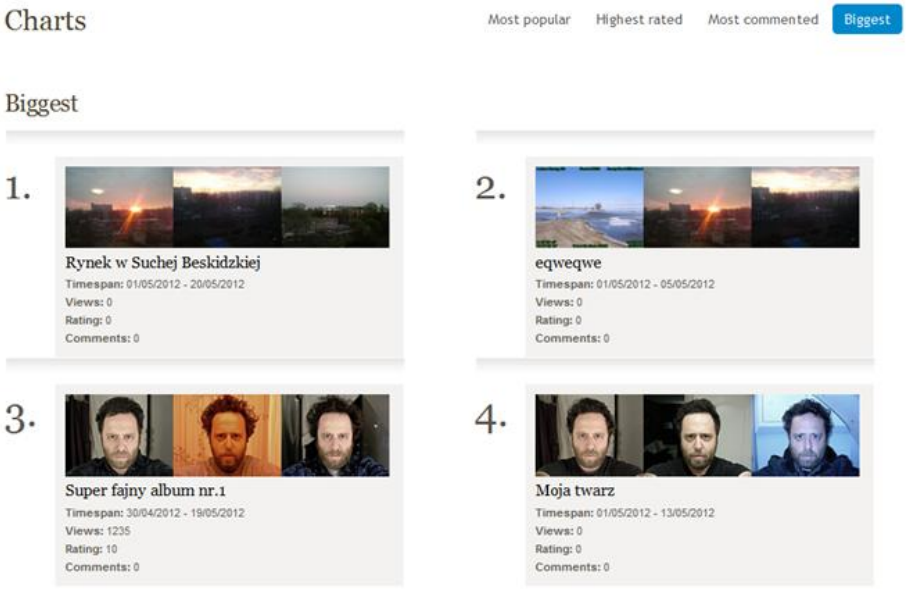

W obu krokach uwierzytelnianie przeprowadzane jest w ten sam sposób. W pierwszym kroku następuje jedynie weryfikacja danych logowania wprowadzonych przez użytkownika, dzięki czemu aplikacja mobilna może od razu wykryć, że wprowadzone zostały nieprawidłowe dane, poinformować o tym i zablokować wykonywanie zapytań do momentu podania poprawnych danych.

Dodatkowo autentykacja odbywa przy każdym zapytaniu, gdyż nie można zakładać, że wstępna weryfikacja danych logowania zostanie w ogóle przeprowadzona. Powoduje to pewien narzut wydajnościowy, jednak jest on naszym zdaniem zdecydowanie akceptowalny, biorąc pod uwagę prostotę takiego mechanizmu oraz jego przenośność.

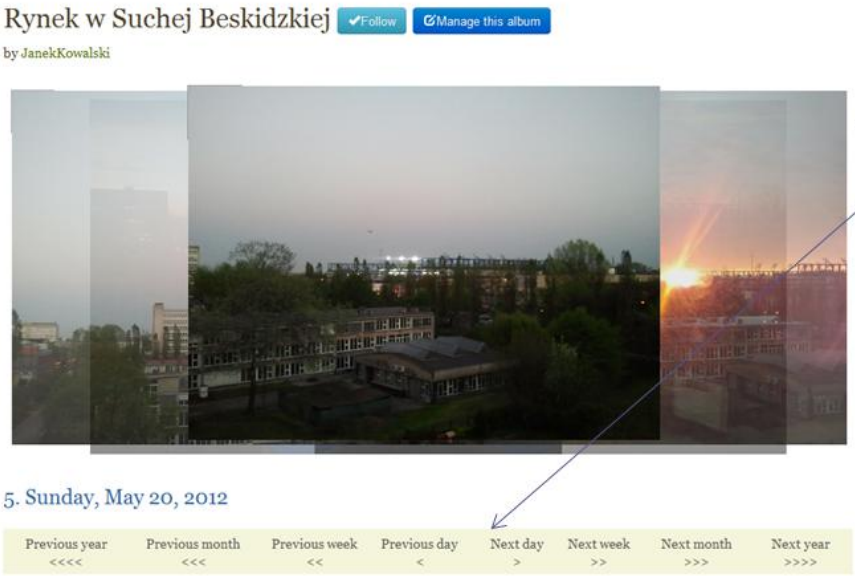
***Prezentacja projektu (zrzuty ekranu)***

- Rankingi albumów

- najbardziej popularne
- najlepiej oceniane
- najczęściej komentowane
- największe (liczba zdjęć)




- Przeglądanie zdjęć




- nawigacja: przeciąganie lub przeskoki czasowe (miesiąc później, itp.)
- technologia: JavaScript (jQuery) zamiast Silverlight
- Możliwość oddania głosu na album

- Zmiana ustawień konta/profilu