# Analiza Obiektowa

# PLAN:

1. AO & PROJEKTOWANIE
2. MODEL KONCEPCYJNY
3. MODELE STRUKTURALNE
4. PRZYKŁAD 1 AO
5. PRZYKŁAD 2 AO
6. KAHOOT

# AO & PROJEKTOWANIE

# Zaczynamy

## Analiza obiektowa (AO):

- podświetlanie **obiektów**
- określić ich podstawowe **właściwości**
- opisać **zachowanie** (polecenia, które może wykonać)

Co jest objektom?

**Obiektem** można nazwać coś, co ma wyraźne granice oraz posiada *stan* i *zachowanie*.

## Stan określa zachowanie :

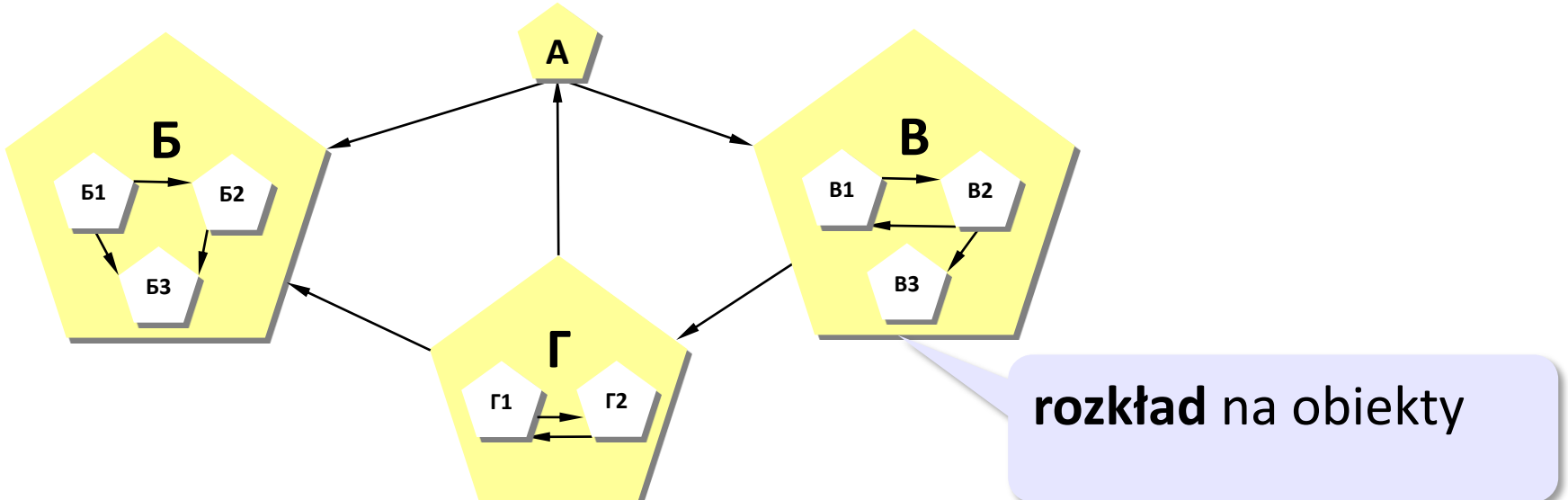
- osoba leżąca nie będzie skakać
- rozładowany pistolet nie wystrzeli

**Klasa** - jest to zbiór obiektów, które mają wspólną strukturę i wspólne zachowanie.

# Używanie obiektów

**Program - zestaw obiektów (modeli), z których każdy ma własne właściwości i zachowanie, ale jego wewnętrzny układ jest ukryty przed innymi obiektami.**

! Konieczne jest „podzielenie” zadania na obiekty!

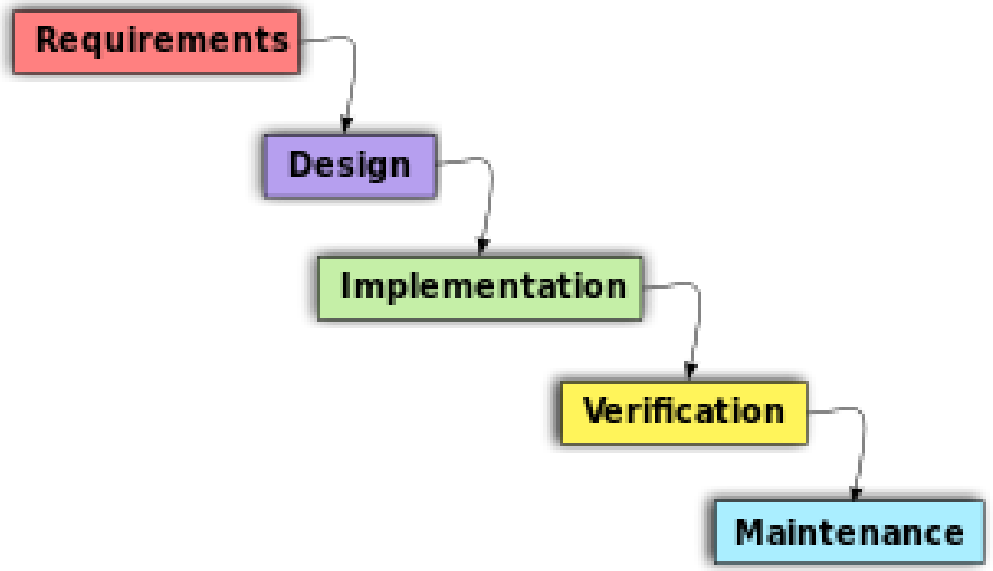



**rozkład** na obiekty

# Analiza obiektowa

Jest to ustrukturyzowana metoda analizy, projektowania systemu poprzez zastosowanie koncepcji zorientowanych obiektowo i opracowanie zestawu graficznych modeli systemu podczas cyklu życia oprogramowania.

**AO w cyklu życia oprogramowania:**



Rozróżnienie między **wymaganiami** a **projektem** jest często opisywane jako „**co** Vs **jak**”.

W zakresie wymagań programiści współpracują z użytkownikami i ekspertami dziedzinowymi, aby zdefiniować, co system ma robić. Szczegóły implementacji powinny być w większości lub całkowicie ignorowane na tym etapie.

Celem fazy wymagań jest stworzenie modelu systemu niezależnie od ograniczeń, takich jak odpowiednia technologia. Zazwyczaj odbywa się to za pomocą przypadków użycia i abstrakcyjnej definicji najważniejszych obiektów przy użyciu modelu koncepcyjnego.

Faza projektowania udoskonala model analizy i stosuje potrzebną technologię oraz inne ograniczenia wdrożeniowe.

Skupia się na opisie obiektów, ich atrybutów, zachowania i interakcji. Model projektowy powinien zawierać wszystkie wymagane szczegóły, aby programiści mogli zaimplementować projekt w kodzie.

# Analiza obiektowa (wymagania)

W analizie obiektowej, my …

**Uzyskujemy wymagania:** określamy, co oprogramowanie musi robić i jaki problem próbuje rozwiązać.

**Określamy wymagania:** opisujemy wymagania, zazwyczaj używając przypadków użycia (i scenariuszy) lub historyjek użytkownika.

**Tworzymy model koncepcyjny:** zidentyfikujemy ważne obiekty, doprecyzuj je oraz zdefiniuj ich relacje i zachowanie, a następnie narysuj je na prostym diagramie.

# Projektowanie obiektowe

W fazie projektowania opisujemy te obiekty (tworząc diagram klas z diagramu koncepcyjnego - zwykle mapując model koncepcyjny na diagram klas), ich atrybuty, zachowanie i interakcje.

Dane wejściowe do projektowania obiektowego są dostarczane przez wyniki analizy obiektowej.

W projektowaniu obiektowym, my …

**Opisujemy klasy** i ich relacje za pomocą diagramu klas.

**opisujemy interakcję** między obiektami za pomocą diagramu sekwencji.

**Stosujemy zasady** projektowania oprogramowania i wzorce projektowe.

# Modelowanie systemowe

Modelowanie systemu to proces opracowywania modeli systemu, przy czym każdy model reprezentuje inną perspektywę tego systemu.

Modele są zwykle oparte na notacji graficznej, która prawie zawsze opiera się na notacji w Unified Modeling Language (UML). Inne modele systemu, takie jak model matematyczny; szczegółowy opis systemu.

## Różne perspektywy:

**Zewnętrzne**, gdzie modeluje się kontekst lub środowisko systemu.

**Interakcja**, gdzie modeluje się interakcję między komponentami systemu lub między systemem a innymi systemami.

**Strukturalne**, gdzie modeluje się organizację systemu lub strukturę danych przetwarzanych przez system.

**Behawioralne**, gdzie modelowane jest dynamiczne zachowanie systemu i jego reakcja na zdarzenia.

# Zunifikowany język modelowania (UML)

Zunifikowany język modelowania stał się standardowym językiem modelowania obiektowego.

Najczęściej używane diagramy to:

**Diagram przypadków użycia (use-case)**: Pokazuje interakcję między systemem a jego środowiskiem (użytkownikami lub systemami) w określonej sytuacji.

**Diagram klas**: Pokazuje różne obiekty, ich relacje, zachowania i atrybuty.

**Diagram sekwencji**: Pokazuje interakcje między różnymi obiektami w systemie oraz między aktorami a obiektami w systemie.

**Diagram maszyny stanów**: Pokazuje, w jaki sposób system reaguje na zdarzenia zewnętrzne i wewnętrzne.

**Diagram aktywności**: Pokazuje przepływ danych między procesami w systemie.

Diagramy można tworzyć na papierze lub na tablicy, przynajmniej na początkowych etapach projektu.

# Lista narzędzi Unified Modeling Language

**Narzędzia UML** to aplikacje obsługujące niektóre funkcje zunifikowanego języka modelowania (Unified Modeling Language).

| Name | Creator | Platform / OS | First public release | Latest stable release | Open source | Software license | Programming language used |
|---|---|---|---|---|---|---|---|
| ArgoUML | Tigris.org | Cross-platform (Java) | 1998-04 | 2011-12-15[1] | Yes | EPL | Java, C++ (as module) |
| Astah | ChangeVision, Inc.[2] | Cross-platform (Java) | 2009-10-19 | 2023-06-28 (v9.1)[3] | No | Commercial. Free education edition, subscription model | Java |
| ATL | Obeo, INRIA Free software community | Cross-platform (Java) | Unknown | 2020-06-22 (4.2.1)[4] | Yes | EPL | Java |
| Micro Focus Together | Micro Focus | Cross-platform (Java) | Unknown | 2008 (v12.6)[5] | No | Commercial | Unknown |
| BOUML | Bruno Pagès | Cross-platform | 2005-02-26 | 2021-01-14 | No | Free from v7.0, Commercial starting from v5.0 up to v6.12,GPL before v5.0[6] | C++/Qt and Java ("plug-out") |
| Cacoo | Nulab | Windows 7+, Mac OS X | 2010-10 | Website frequently updated | No | Commercial, Free edition available | HTML5 |
| CaseComplete | Serlio Software | Windows | 2004 | 2020 (v15) | No | Commercial | C# |
| ConceptDraw PRO | CS Odessa | Windows, macOS | 1993 | 2017-11-07 (v11)[7] | No | Commercial | Unknown |
| Creately | Cinergix | Windows, Mac OS X | 2008 | Website frequently updated | No | Commercial, Free edition available | HTML5 |
| Dia | Alexander Larsson/GNOME Office | Cross-platform (GTK+) | 1998-08-31[8] | 2014-09-05 (v0.97.3)[9] | Yes | GPL | C |
| Diagrams.net previously Draw.io[10] | JGraph Ltd.[11] | Windows, Linux, macOS, Chrome | 2016-09-06[12] | Website frequently updated [13] | Yes | Free, Apache v2 | Javascript, Java |
| Eclipse UML2 Tools[14] | Eclipse Foundation | Cross-platform (Java) | 2007 | 2020-03-02 (v5.5.1)[15] | Yes | EPL[16] | Java |
| Edraw Max | Edrawsoft | Windows, Linux, macOS | 2004 | 2020-07-30 (v10.0.5)[17] | No | Commercial | C++ |

# MODEL KONCEPTUALNY

Identyfikacja najważniejszych obiektów, ich relacji i interakcji między nimi.

# Model konceptualny

Gdy mamy już kilka przypadków użycia lub historyjek użytkownika, następną rzeczą, którą możemy zrobić, jest stworzenie modelu koncepcyjnego naszego systemu. Oznacza to po prostu identyfikację najważniejszych obiektów.

**Proces**

Zidentyfikujemy więc te obiekty, zaczniemy je doprecyzowywać, a następnie narysujemy je na prostym diagramie. Możemy również pokazać relacje i interakcje między nimi.

**Wskazówka**

Stworzenie prostego modelu koncepcyjnego dla większości aplikacji nie jest i nie powinno być długim procesem. Kilka godzin poświęconych na to jest zwykle więcej niż wystarczające.

# 1. Identyfikacja obiektów

To, co robimy, to rozpoczęcie zbierania naszych przypadków użycia, historyjek użytkownika i wszelkich innych pisemnych wymagań razem.

Use case:

Customer adds items to the shopping cart. User will then enter their payment details, delivery address, and email address for the order to be fulfilled by the system.

The system processes the payment and sends a confirmation to the customer's email with the order #, status, and items.
Customer can also view their order details.

Obiekty będą miały formę *rzeczowników*. Są to obiekty kandydujące, niektóre z nich będą rzeczywistymi obiektami w systemie, a reszta nie, jak zobaczysz później.

Po prostu identyfikujesz obiekty, nie analizujesz ich ani nie oceniasz. Nie martwimy się też o to, że któregoś z nich zabraknie, przy pierwszym obrocie możesz przegapić niektóre obiekty, co jest normalne.

# 2. Udoskonalanie obiektów

Po utworzeniu obiektów kandydujących, zaczynasz je udoskonalać i wybierać rzeczywiste obiekty, które znajdą się w systemie.

Usuń wszelkie **duplikaty.** Możemy znaleźć te same obiekty o różnych nazwach, ale w rzeczywistości oznaczają one to samo.

Konieczne może być **połączenie** niektórych obiektów, a nawet **podzielenie** ich na inne.

Zamiast tego można zidentyfikować **atrybut** jako obiekt.

Zamiast tego można zidentyfikować **zachowanie** jako obiekt.

***Atrybut*** jest właściwością lub cechą obiektu. Na przykład, gdy mówimy „Samochód jest czerwony”, czerwony jest tutaj właściwością samochodu, który jest rzeczywistym obiektem.

***Zachowanie*** jest czymś, co obiekt może zrobić (odpowiedzialność). Na przykład, kiedy mówimy: „Ptak potrafi latać”, latanie jest tutaj zachowaniem, podczas gdy ptak jest rzeczywistym obiektem.

## 3. Rysowanie obiektów

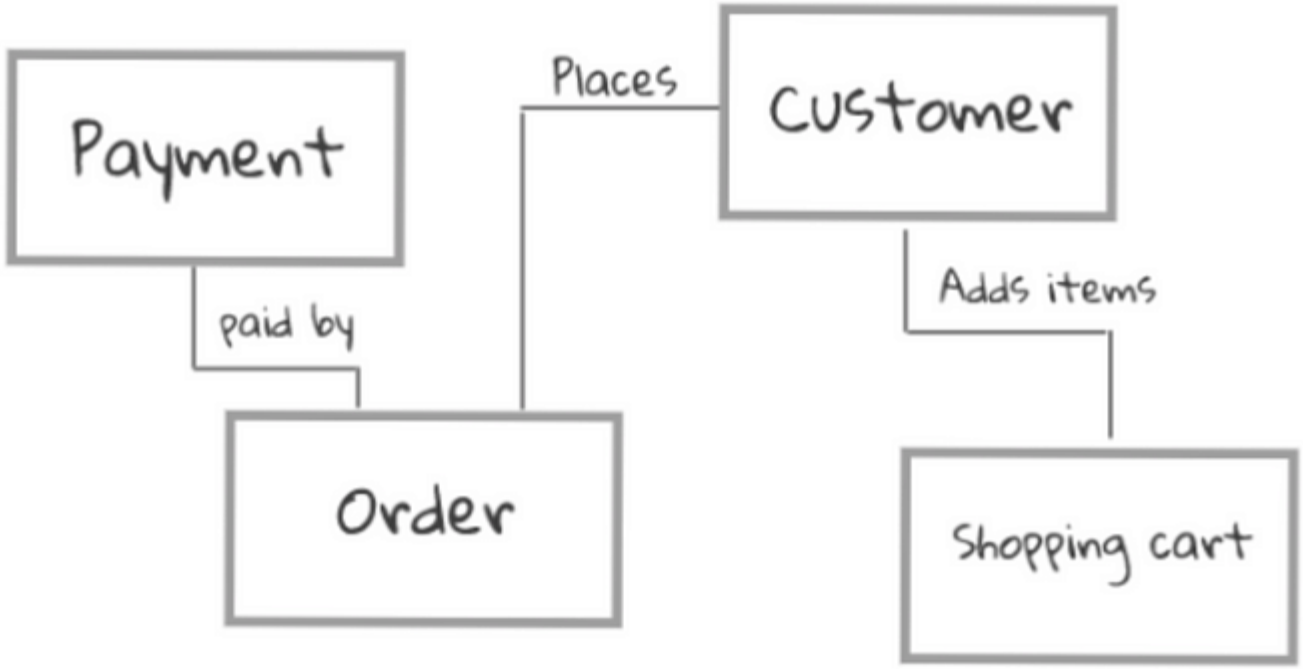
Teraz należy narysować model koncepcyjny za pomocą ołówka i kartki papieru, zaznaczając wszystkie obiekty.

Istnieją pewne narzędzia, których możesz użyć, ale na razie ołówek i kartka papieru są więcej niż wystarczające.

## 3. Identyfikacja relacji między obiektami

Zacznij wskazywać relacje między obiektami. To bardzo oczywiste, że te obiekty będą ze sobą współdziałać. Na przykład klient może złożyć zamówienie, student może zapisać się na kurs, administrator może zaktualizować post i tak dalej.

Narysowanie linii między obiektami i napisanie czasowników relacji wystarczy, aby oznaczyć istnienie relacji.



# 4. Identyfikacja zachowań obiektów

Zachowania to rzeczy (czasowniki), które obiekt może zrobić, lub, innymi słowy, obowiązki obiektu, które staną się metodami w naszej klasie obiektów.

Możemy więc wrócić do przypadku użycia lub historii użytkownika i poszukać czasowników i fraz czasownikowych, aby wybrać obowiązki.

Takie rzeczy jak *dodawanie* pozycji, *wprowadzanie* szczegółów płatności i adresu, *realizacja* zamówienia, *przetwarzanie* płatności,*wysyłanie* wiadomości e-mail ze szczegółami zamówienia i *przeglądanie* szczegółów zamówienia.

Use case:

Customer adds items to the shopping cart. User will then enter their payment details, delivery address, and email address for the order to be fulfilled by the system.

The system processes the payment and sends a confirmation to the customer's email with the order #, status, and items. Customer can also view their order details.

Add (shopping cart) items

Enter payment details, delivery address, and email address

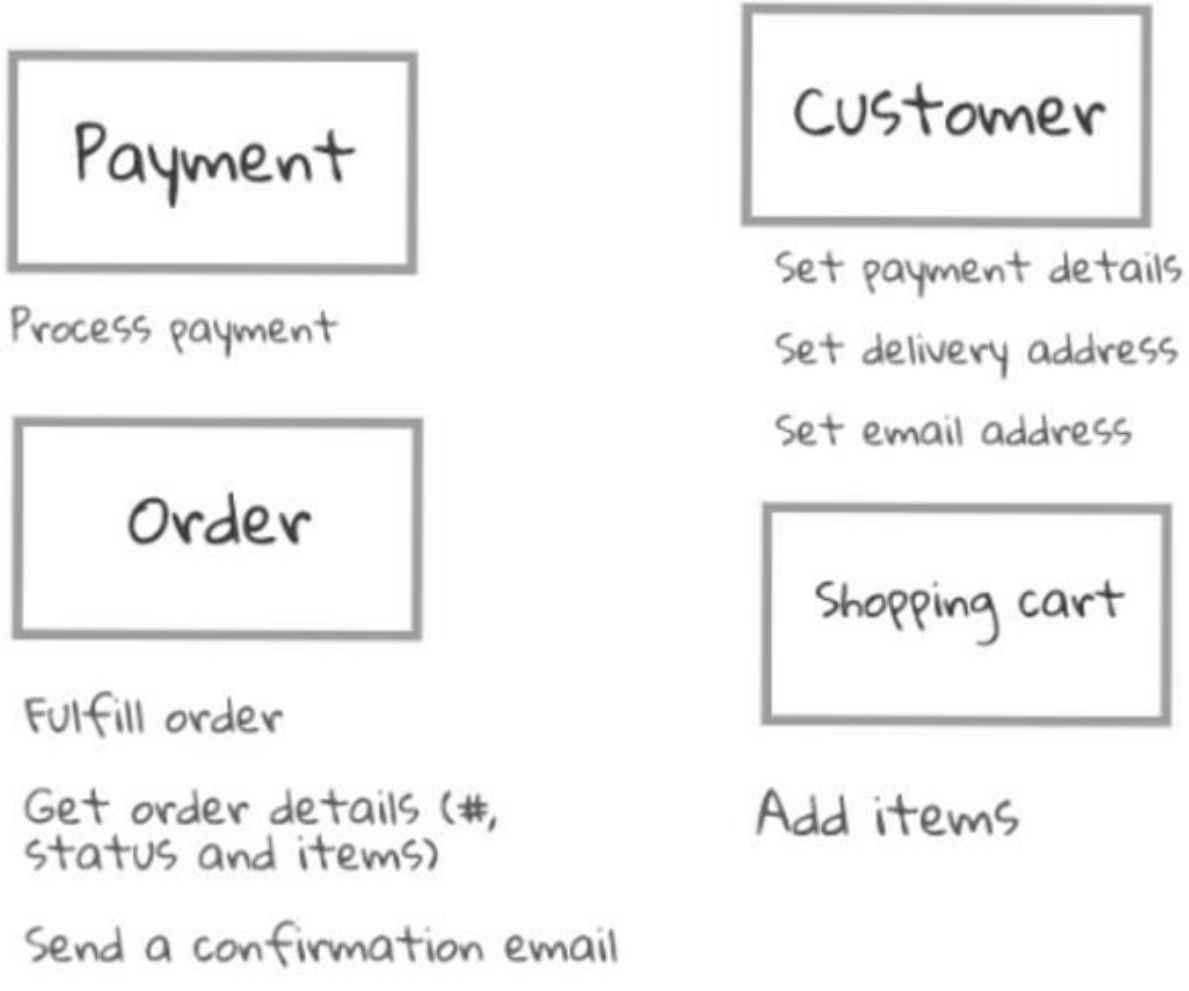
Process the payment

Fulfill the order

Send a confirmation email

View order details

Nie wszystkie z nich staną się zachowaniami, niektóre zostaną połączone, niektóre trzeba będzie rozdzielić, a niektóre po prostu nie będą potrzebne lub zostaną zastąpione czymś innym, ale są one dobrym punktem wyjścia.



# Karty Współpracownika Odpowiedzialności Klasowej (CRC)

Class Responsibility Collaborator (CRC) to kolejna technika organizowania obiektów.

**Karty klasy-odpowiedzialności-współpracy (CRC)** są narzędziem burzy mózgów wykorzystywanym w projektowaniu oprogramowania obiektowego. Zostały one pierwotnie zaproponowane przez Warda Cunninghama i Kenta Becka jako narzędzie dydaktyczne, ale są również popularne wśród projektantów-ekspertów i zalecane przez praktyków programowania ekstremalnego.

Zapisujesz każdy obiekt na kartce papieru, z jego powiązanymi zachowaniami po jednej stronie, a po drugiej stronie znajdują się inne obiekty (współpracownicy), które mają związek z tym obiektem.

Są małe, łatwe w użyciu, można je przesuwać, aby pokazać związek, można je łatwo modyfikować, jeśli wystąpi błąd.

# MODELE STRUKTURALNE

Modelowanie organizacji systemu.

Modele strukturalne oprogramowania przedstawiają organizację systemu pod względem komponentów, które go tworzą i ich relacji. Skupimy się na diagramie klas do modelowania struktury klas w systemie oprogramowania.

## Diagram klas

Do stworzenia diagramu klas wykorzystamy informacje o naszych obiektach z modelu pojęciowego.

### Konwencja nazewnictwa

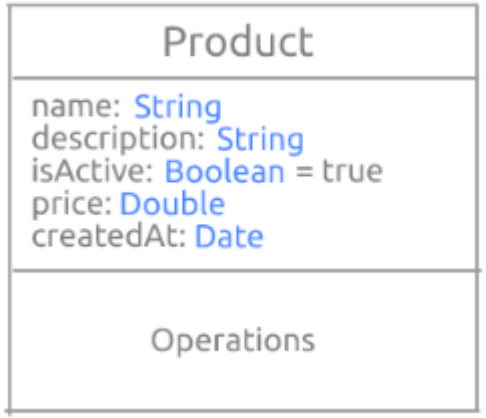
Zaczynasz od sposobu nazywania swoich klas. Zazwyczaj nazwa klasy jest w liczbie pojedynczej i pisana wielką literą, jak na przykład „BankAccount”.

W przypadku atrybutów i metod są one pisane wielbłądzią literą; mała pierwsza litera, a wszystkie inne słowa zaczynają się wielką literą, jak „getProductDetails”.

Niektórzy ludzie wolą używać wielkich pierwszych liter dla prywatnych atrybutów i metod, ale nie ma to znaczenia, chyba że używasz jednego sposobu nazewnictwa, a twój kod jest przejrzysty, łatwy do odczytania i utrzymania.

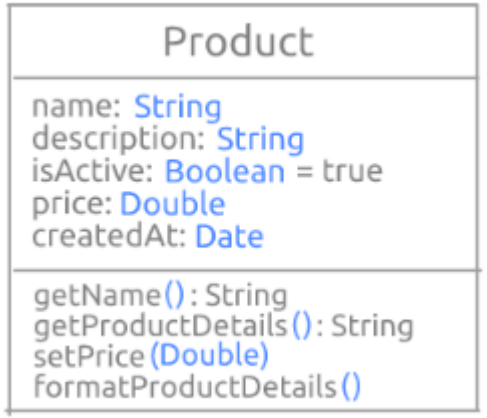
## Atrybuty

Następnie wypisujemy atrybuty, ich typy danych i wartości domyślne, jeśli istnieją.




## Metody

Wpisz metody, ich parametry, jeśli istnieją, w nawiasach i ich typ zwracany.

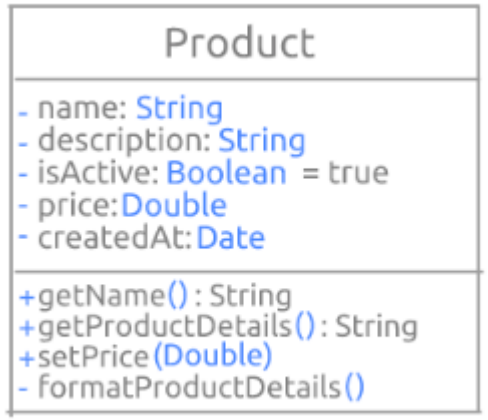


# Public, Private and Protected

Są one nazywane modyfikatorami dostępu. Służą one do kontrolowania dostępu do członków obiektu.

Dobrą praktyką jest utrzymywanie wszystkich członków (atrybutów i metod) jako prywatnych, chyba że muszą one zostać ujawnione.

Używamy więc znaków do oznaczenia modyfikatora dostępu każdego członka w klasie.



**Public** (‘+’): Mówi się, że można go zobaczyć i uruchomić w dowolnej części aplikacji.

**Private** (‘-’): Mówi, że można go uruchomić tylko wewnątrz jego klasy.

**Protected** (‘#’): Mówi, że można go wywołać tylko wewnątrz jego klasy, a także wewnątrz klas podrzędnych.

Niektórzy programiści mają tendencję do utrzymywania swoich atrybutów jako prywatnych, nawet jeśli wiedzą, że inna klasa musi je uruchomić. Pozwalają one na pobieranie i ustawianie wartości tych atrybutów za pomocą tak zwanych *Getters* i *Setters*. Są to metody umożliwiające pobieranie i ustawianie atrybutów bez zezwalania zewnętrznemu kodowi na bezpośredni wpływ na nie.

# Constructor and Destructor

Za każdym razem, gdy tworzysz obiekt, musisz go skonstruować, co oznacza przypisanie wszystkim zmiennym instancji ich prawidłowych wartości. Daje to pewność, że obiekt będzie w prawidłowym stanie po utworzeniu.

Konstruktor to metoda o tej samej nazwie co klasa. Jest ona wywoływana zaraz po utworzeniu obiektu. Można mieć wiele konstruktorów.

*Posiadanie dwóch lub więcej metod o tej samej nazwie, ale z różnymi parametrami, zarówno pod względem liczby parametrów, ich kolejności, jak i typu danych, nazywane jest „przeciążaniem metod”.*

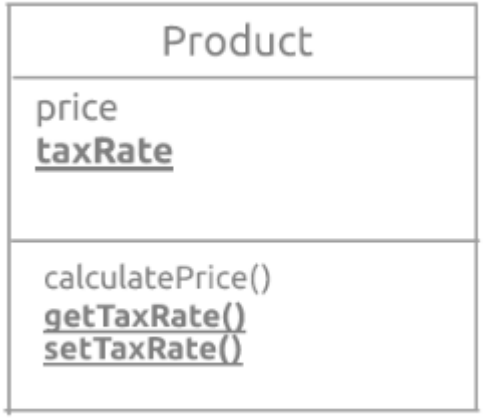
W Javie nie ma destruktora, ponieważ jest to język odśmiecania, więc nie można wiedzieć, kiedy obiekt zostanie zniszczony przez odśmiecanie. Podczas gdy w C++ ta koncepcja ma zastosowanie i mamy destruktor

# Instance Members Vs Static Members

Być może potrzebujemy elementu członkowskiego, który będzie taki sam dla wszystkich instancjonowanych obiektów.

W takim przypadku możemy utworzyć *statyczny* element członkowski, który będzie współdzielony przez całą klasę. Tak więc, będzie jedna i tylko jedna kopia tego elementu.

Członkowie statyczni są pokazani z podkreśleniem, aby odróżnić je od członków instancii.




Dostęp do członka(member) statycznego uzyskuje się poprzez nazwę klasy, a nie instancję obiektu, a także można uzyskać do niego dostęp, nawet jeśli żaden obiekt nie został jeszcze utworzony.

Teraz, jedna ważna rzecz, którą należy wiedzieć, nie można wywołać metod instancji lub atrybutów wewnątrz metody statycznej.

Powinno to mieć dla ciebie sens, ponieważ metody statyczne nie są powiązane z żadnym obiektem, więc te metody instancji lub atrybuty nie należą do żadnego obiektu, gdy próbujemy je wywołać z wnętrza metody statycznej.

```
public class Product {
   private double price;
   private static double taxRate = 0.1;  // 10% tax rate on price

   public static double getTaxRate() {
      // this.price = 5123.0;         // WRONG!
      return taxRate;
    }

   public double calculatePrice() {
      return this.price + (this.price * Product.taxRate);
   }
}
```

# Rodzaje związków

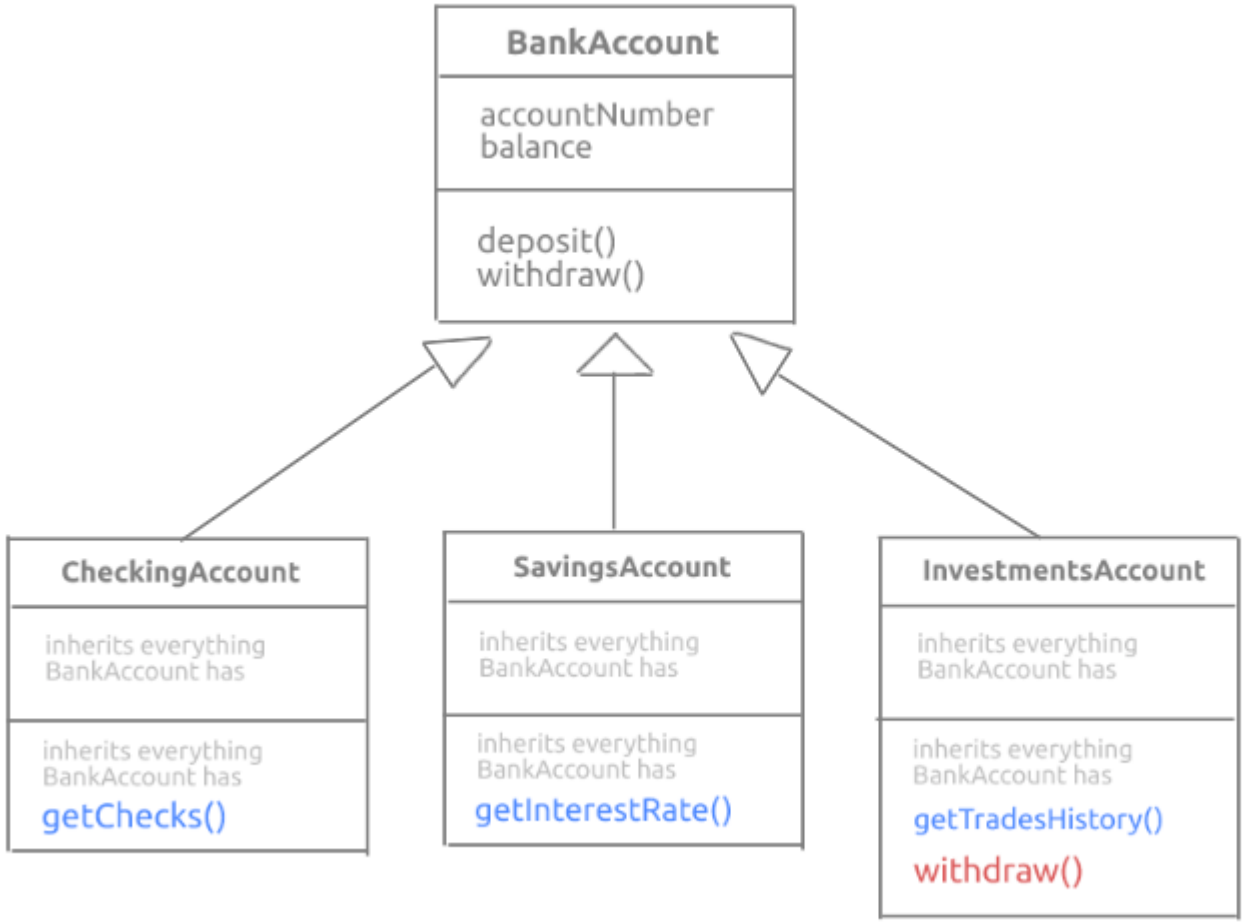
## Inheritance(Dziedziczenie)

Kiedy mówimy „uczeń jest osobą”, „pracownik jest osobą”, „samochód jest pojazdem”, „autobus jest pojazdem” itd. Wskazuje to, że uczeń i pracownik mają pewne wspólne atrybuty i zachowania, to samo dotyczy samochodu i autobusu..

Dziedziczenie zapewnia ogólną *nadklasę* , klasę abstrakcyjną, która ma wspólne atrybuty i metody dla wszystkich *podklas*.

W ten sposób unikamy duplikatów, ponieważ nie musimy pisać wspólnych atrybutów lub metod. Wystarczy napisać je tylko raz w nadklasie, a automatycznie będą one miały wszystkie atrybuty i metody z ich nadklasy.

# Inheritance




**Overriding**

Możemy dodać pewne metody lub atrybuty, które są specyficzne dla klasy podrzędnej.

Coś, na co należy uważać, jeśli chcemy przepisać odziedziczone metody w klasie podrzędnej, robimy coś, co nazywa się **nadpisywaniem(overriding)**.

*Nadpisywanie to zastępowanie lub dodawanie dodatkowego zachowania do odziedziczonego.*

Nie jest to dobra praktyka w tym przypadku, chyba że masz abstrakcyjną metodę w superklasie. Metody abstrakcyjne zostaną wyjaśnione później.

Kolejną rzeczą jest unikanie sytuacji, w której masz wiele poziomów dziedziczenia - Depth of Inheritance (DIT).

# Association, Aggregation and Composition (Asocjacja, agregacja i kompozycja)

Identyfikowaliśmy relacje między obiektami, rysując linię między dowolnymi dwoma obiektami w naszym modelu koncepcyjnym.

Może to oznaczać relację **asocjacyjną**. Oznacza to, że obiekt musi wiedzieć o innym obiekcie lub wchodzić z nim w interakcje. Może to być na przykład sytuacja, w której obiekt *klienta* musi wywołać metodę *koszyka na zakupy*.

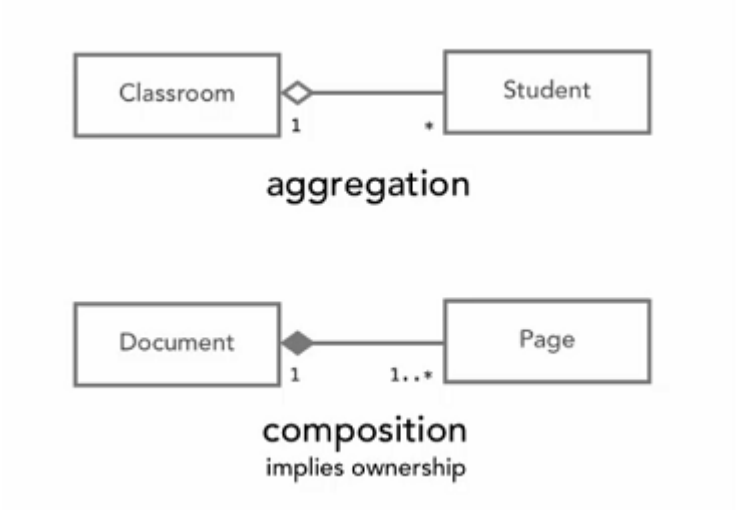
```
public class Customer {
  private ShoppingCart myCart;
  // ...
  public int items(){
      return myCart.getItemCount();
  }
}

public class ShoppingCart {
  private ArrayList items;
  // ...
  public void addItem(Item t) {
      items.add(t);
  }
  public int getItemCount() {
      return items.size();
  }
}
```

Pokazuje to, że obiekt *klienta* używa obiektu *koszyka* , aby uzyskać liczbę przedmiotów w koszyku klienta, a obiekt *koszyka* musi użyć obiektu *item* , aby dodać przedmioty do koszyka.

Relacja **agregacji** jest również asocjacją. Opisuje jednak, że obiekt może być wykonany lub zbudowany z innych obiektów.

Na przykład „samochód ma silnik”, „klasa ma uczniów” itp. Odnosi się do relacji „HAS-A”, a nie „IS-A”, jak w przypadku dziedziczenia. Nie możemy więc powiedzieć, że „samochód JEST siedzeniem”.

**Kompozycja** jest tym samym, co agregacja, ale implikuje pojęcie własności. Jest to więc bardziej specyficzna forma agregacji.



Na przykład, jeśli mamy klasę *dokumentu* i klasę *strony* , a obiekt *dokumentu* składa się z obiektów *strony* .

Jeśli w dowolnym momencie obiekt *dokumentu* nie został jeszcze utworzony; nie istnieje, obiekt *strony* również nie powinien istnieć.

Podobnie, jeśli zniszczyliśmy obiekt *dokumentu* , to powiązane z nim *obiekty stron* również powinny zostać zniszczone. Nie może on istnieć logicznie, jeśli obiekt *dokumentu* nie istnieje.

Kiedy używać agregacji, a kiedy kompozycji? Użyj agregacji, gdy obiekty mogą istnieć niezależnie, i użyj kompozycji, jeśli te obiekty nie mogą logicznie istnieć bez obiektu, który jest z nich utworzony.

# Abstract Class
# (Klasa abstrakcyjna)

Klasa abstrakcyjna to klasa, która nie może być instancjonowana; nie można tworzyć z niej obiektów. Istnieje tylko po to, by być dziedziczoną.

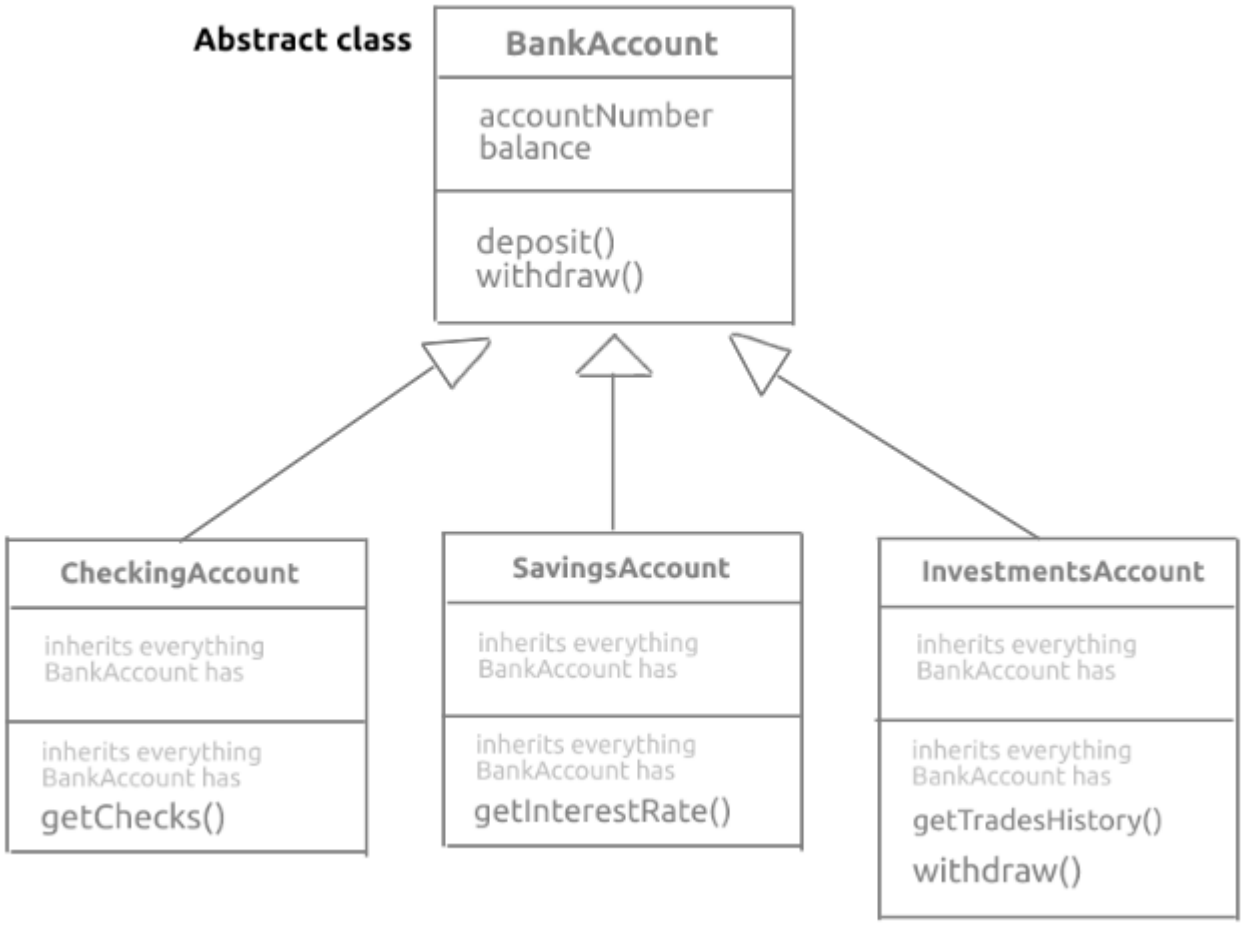
Istnieje po to, aby zapewnić klasę generyczną, która ma pewne wspólne atrybuty i metody wspólne dla wszystkich dziedziczonych klas, a ty nigdy nie utworzysz instancji obiektu z tej klasy. Z drugiej strony, klasa **konkretna** może być instancjonowana.

Oznaczamy klasę jako abstrakcyjną poprzez użycie słowa kluczowego „abstract” w deklaracji klasy.

```
abstract public class BankAccount {
   // your code goes here
}
```

W klasie abstrakcyjnej metody i atrybuty nie muszą być statyczne, więc mogą być dostępne dla każdego obiektu klasy, która dziedziczy z tej klasy abstrakcyjnej.

# An abstract class



# Abstract Methods
# (Metody abstrakcyjne)

Metoda abstrakcyjna to metoda, która jest zadeklarowana bez implementacji i musi znajdować się wewnątrz klasy abstrakcyjnej.

Wszystkie podklasy muszą nadpisywać metody abstrakcyjne. Jest to przypadek, w którym nadpisywanie odziedziczonych metod jest nie tylko dozwolone, ale i obowiązkowe.

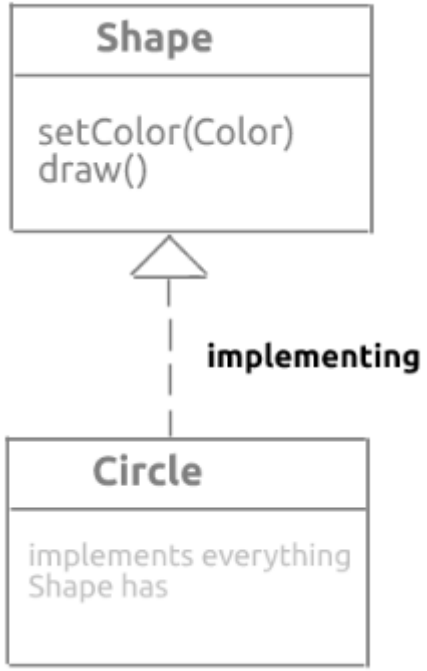
```
abstract public class BankAccount {
   abstract public void withdraw();
}
```

W klasie abstrakcyjnej możemy mieć również metody nieabstrakcyjne, które są współdzielone przez wszystkie podklasy.

# Interfaces (Interfejsy)

Jest to klasa abstrakcyjna niejawnie (nie piszesz słowa kluczowego „abstract)”, ale składa się z listy metod.

Istnieje również po to, by być dziedziczoną i nadpisywaną przez podklasy.




```
public interface Shape {
  public void setColor(Color color);
  public void draw();
}
```

W tym miejscu należy zadać ważne pytanie: czego powinniśmy użyć w sytuacji dziedziczenia, konkretnej klasy czy interfejsu? Cóż, to zależy od tego, co chcemy zrobić:

Jeśli wspólne metody i atrybuty są współdzielone między wszystkimi podklasami, potrzebujesz klasy konkretnej, więc piszesz kod tylko raz, a podklasy będą go miały domyślnie.

Jeśli, z drugiej strony, istnieją wspólne metody między podklasami, ale każda podklasa ma swoją własną implementację, potrzebujemy interfejsu.

Prawdą jest, że wielu programistów woli używać interfejsu, aby zapewnić formalną listę metod do obsługi, niż dziedziczyć z konkretnej klasy i zajmować się wieloma wstępnie dostarczonymi funkcjami, które mogą, ale nie muszą być poprawne.

# Przykład 1 AO

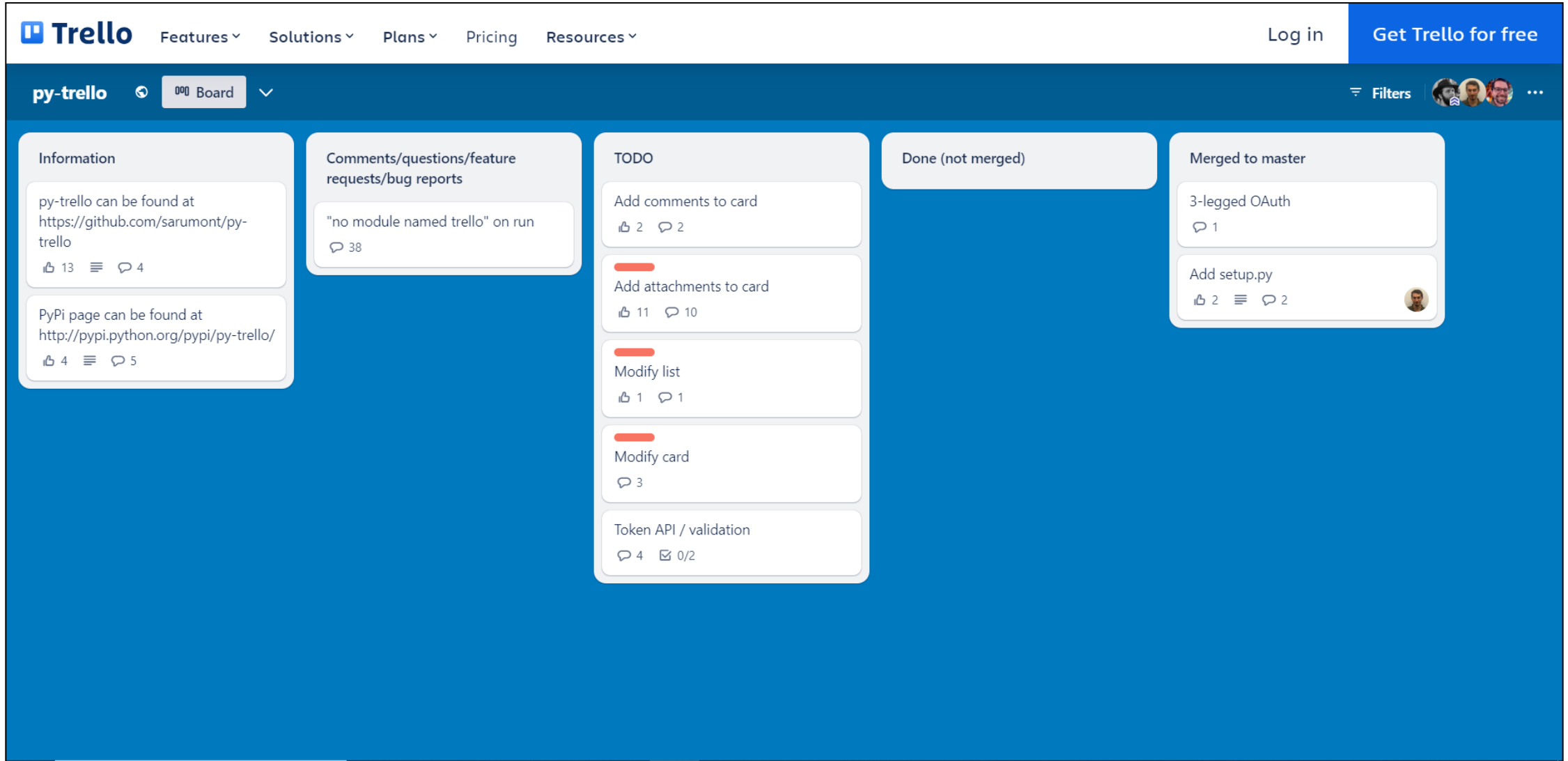
# INTELLIGENT MODULE OF BROWSER MANAGEMENT SYSTEM OF IT-PROJECTS TRELLO
# (INTELIGENTNY MODUŁ PRZEGLĄDARKOWEGO SYSTEMU ZARZĄDZANIA PROJEKTAMI IT TRELLO)

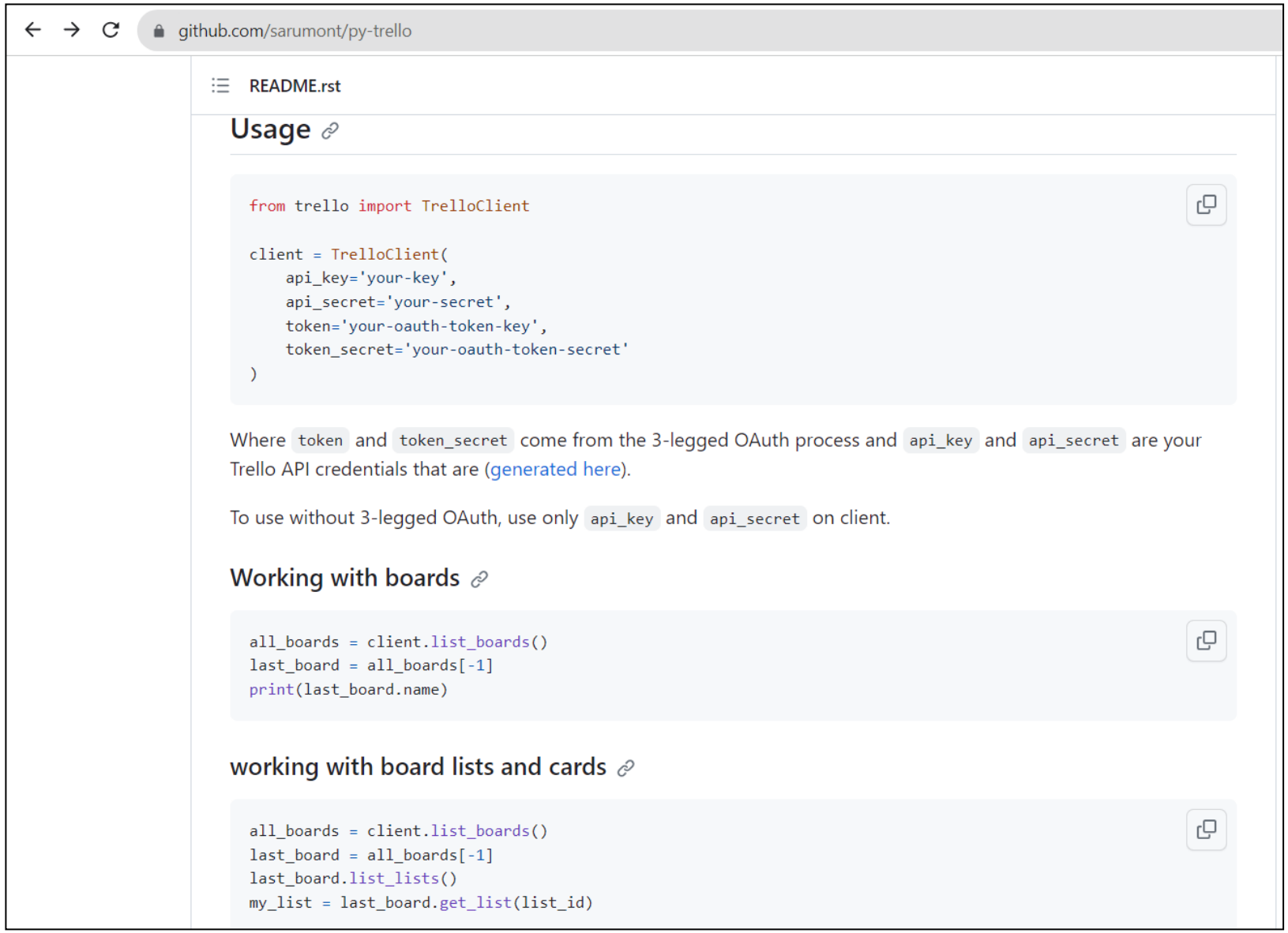
**Celem** pracy jest rozszerzenie funkcjonalności przeglądarkowego systemu zarządzania projektami IT Trello poprzez opracowanie inteligentnego modułu dla API Trello.

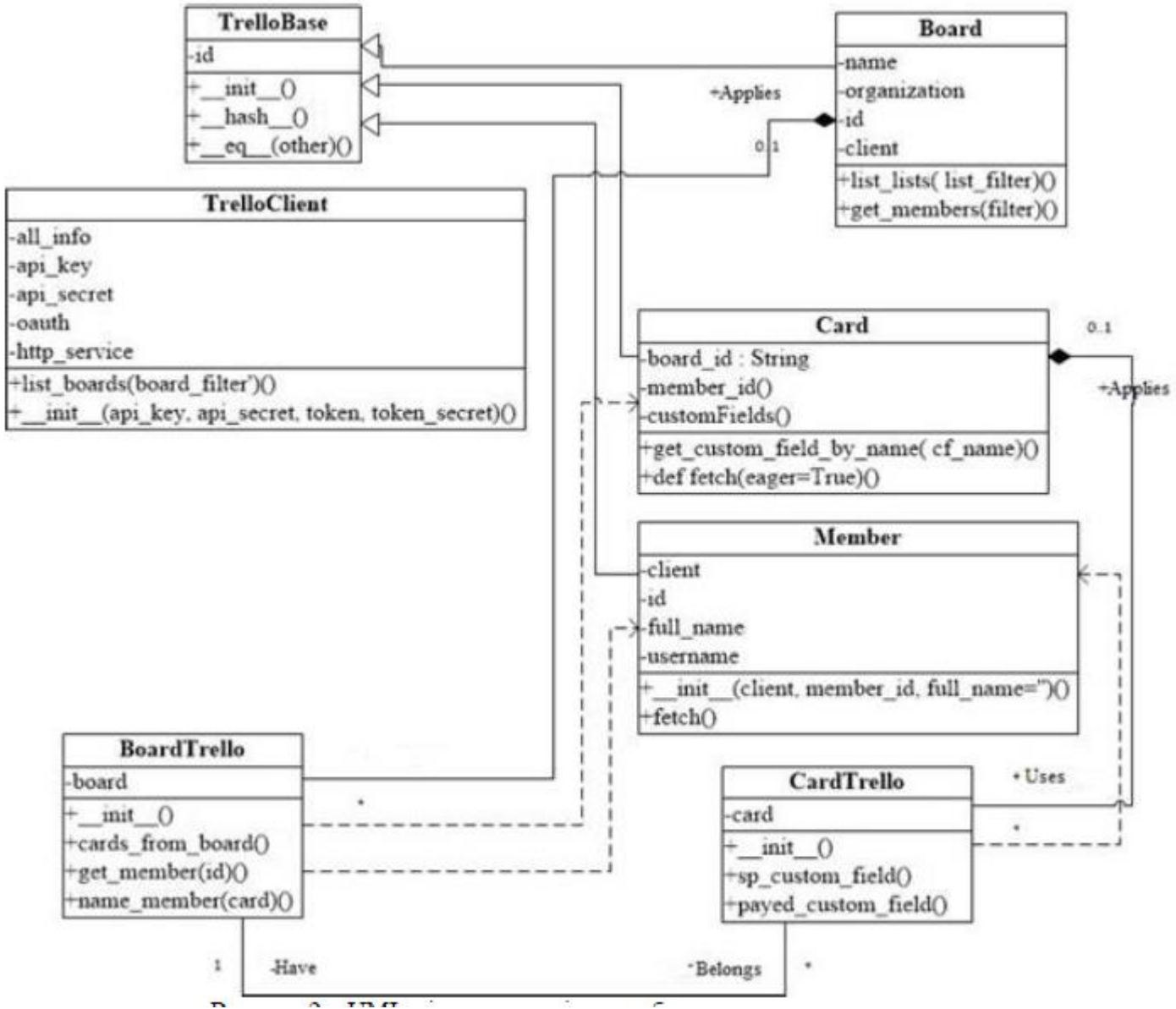
**Zadania**:

1) wybór danych z tablic, list i kart Trello;
2) agregacja otrzymanych danych;
3) dane wyjściowe w postaci tabel w pliku w formacie Microsoft Excel;
4) rozwój inteligentnego modułu dla API Trello;
5) zatwierdzanie wyników rozwoju w działaniach zespołu IT programistów Sinai RnD.

# TRELLO Board:




github.com/sarumont/py-trello
README.rst
Usage
from trello import TrelloClient
client = TrelloClient(
api_key='your-key',
api_secret='your-secret',
token='your-oauth-token-key',
token_secret='your-oauth-token-secret'
)
Where token and token_secret come from the 3-legged OAuth process and api_key and api_secret are your Trello API credentials that are (generated here).
To use without 3-legged OAuth, use only api_key and api_secret on client.
Working with boards
all_boards = client.list_boards()
last_board = all_boards[-1]
print(last_board.name)
working with board lists and cards
all_boards = client.list_boards()
last_board = all_boards[-1]
last_board.list_lists()
my_list = last_board.get_list(list_id)


TrelloBase
-id
+__init__()
+__hash__()
+__eq__(other)()
Board
-name
-organization
-id
-client
+list_lists( list_filter)()
+get_members(filter)()
+Applies
0..1
TrelloClient
-all_info
-api_key
-api_secret
-oauth
-http_service
+list_boards(board_filter')()
+__init__(api_key, api_secret, token, token_secret)()
Card
-board_id : String
-member_id()
-customFields()
+get_custom_field_by_name( cf_name)()
+def fetch(eager=True)()
0..1
+Applies
Member
-client
-id
-full_name
-username
+__init__(client, member_id, full_name='')()
+fetch()
BoardTrello
-board
+__init__()
+cards_from_board()
+get_member(id)()
+name_member(card)()
CardTrello
-card
+__init__()
+sp_custom_field()
+payed_custom_field()
+Uses
1
-Have
*Belongs
*

Możliwą opcją rozwoju modułu oprogramowania jest wykorzystanie imitacji klas *Board* i *Card*. Wówczas istnieje potrzeba stworzenia wrappera dla klasy *TrelloClient*.

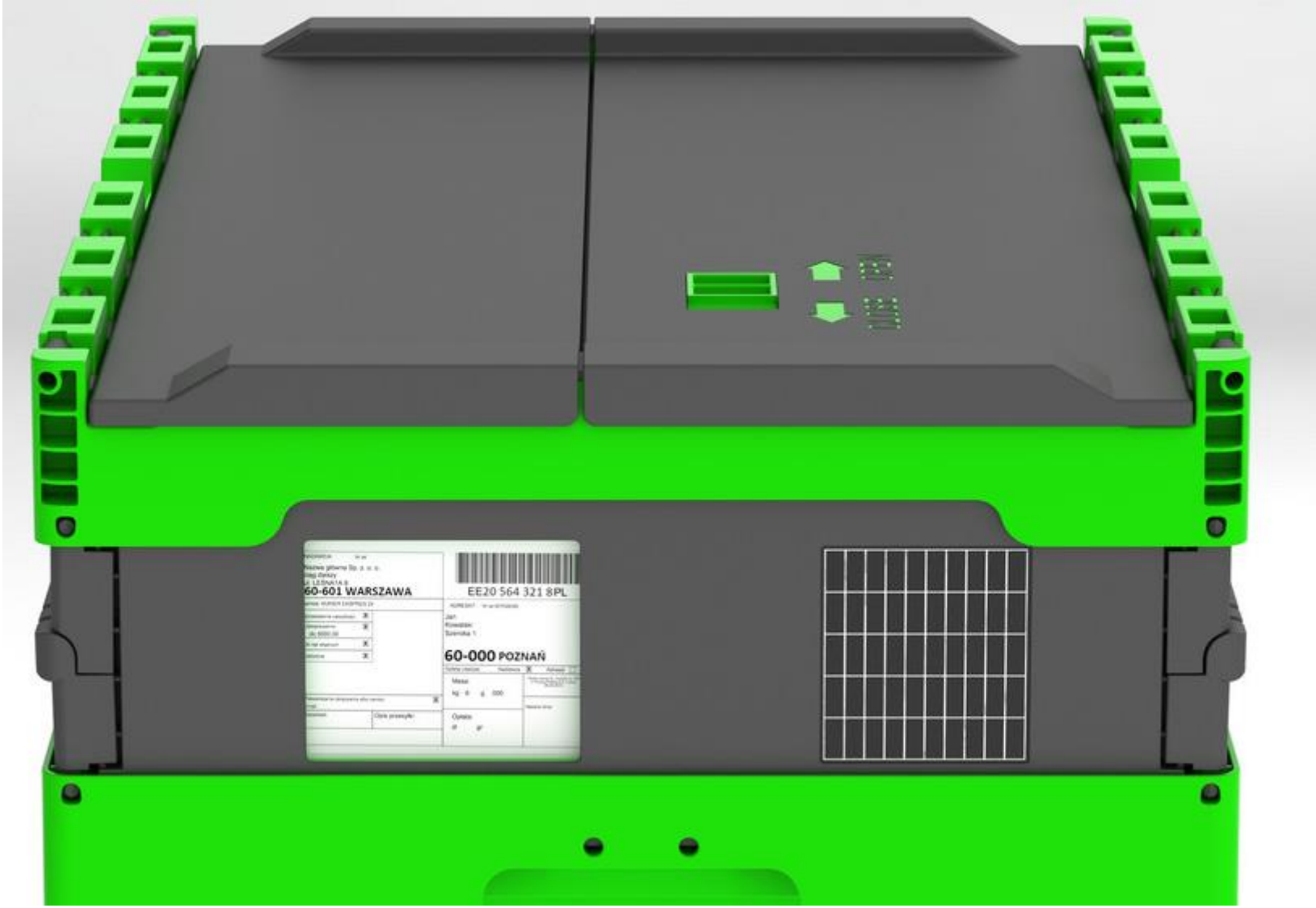
Bardziej optymalną opcją jest użycie obiektów Board i Card w treści programu, które będą instancjami odpowiednio klas *BoardTrello* i *CardTrello*.

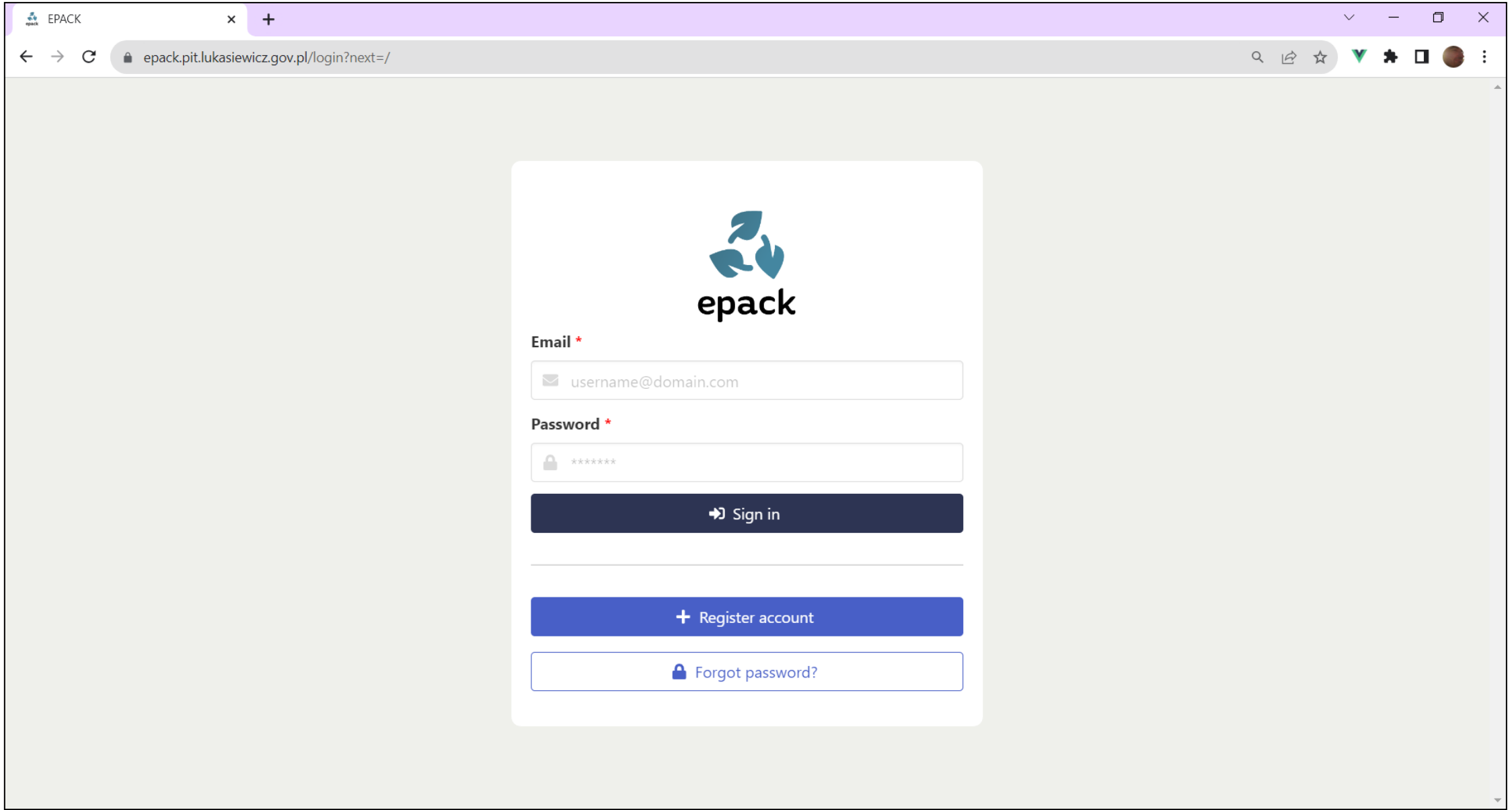
Z rysunku widać, że klasa *TrelloClient* jest jakby oddzielona od wszystkich innych i nie jest z nimi powiązana, ale jest to klasa zawierająca pola i metody uwierzytelniania klienta systemu zarządzania *Trello*. Nie jest ona bezpośrednio powiązana ze wszystkimi klasami, ale bez niej praca żadnej z nich nie jest możliwa. Klasy *Board*, *Card* i *Member* dziedziczą po klasie *TrelloBase*. Zatem każda zmiana w klasie *TrelloBase* spowoduje konieczność zmiany wszystkich klas, które po niej dziedziczą. Relacja klasy *BoardTrello* do klasy *Board*, podobnie jak klasy *CardTrello* do klasy Card, jest kompozycyjna, ponieważ *BoardTrello* jest częścią całej *Board*, a *CardTrello* jest tylko częścią całej Card. Jednocześnie klasa *BoardTrello* zależy od klas *Card* i *Member*, a klasa *CardTrello* zależy od klasy *Member*. Zależność ta jest pokazana na rysunku linią przerywaną. W głównej części programu tworzone są dwa obiekty - obiekt Board, który jest instancją klasy *BoardTrello*, oraz obiekt Card, który jest instancją klasy *CardTrello*. Pomiędzy tymi dwoma obiektami w ciele programu nawiązywana jest ścisła komunikacja. Dlatego relacja między klasami *BoardTrello* i *CardTrello* jest asocjacją. Mówiąc dokładniej, relacja między klasami *BoardTrello* i *CardTrello* jest asocjacją typu N-array, ponieważ jeden obiekt board odpowiada jednemu zestawowi obiektów card. Dlatego na rys. 2 klasy *BoardTrello* i *CardTrello* są połączone linią ciągłą, a ich moc jest równa 1..*.

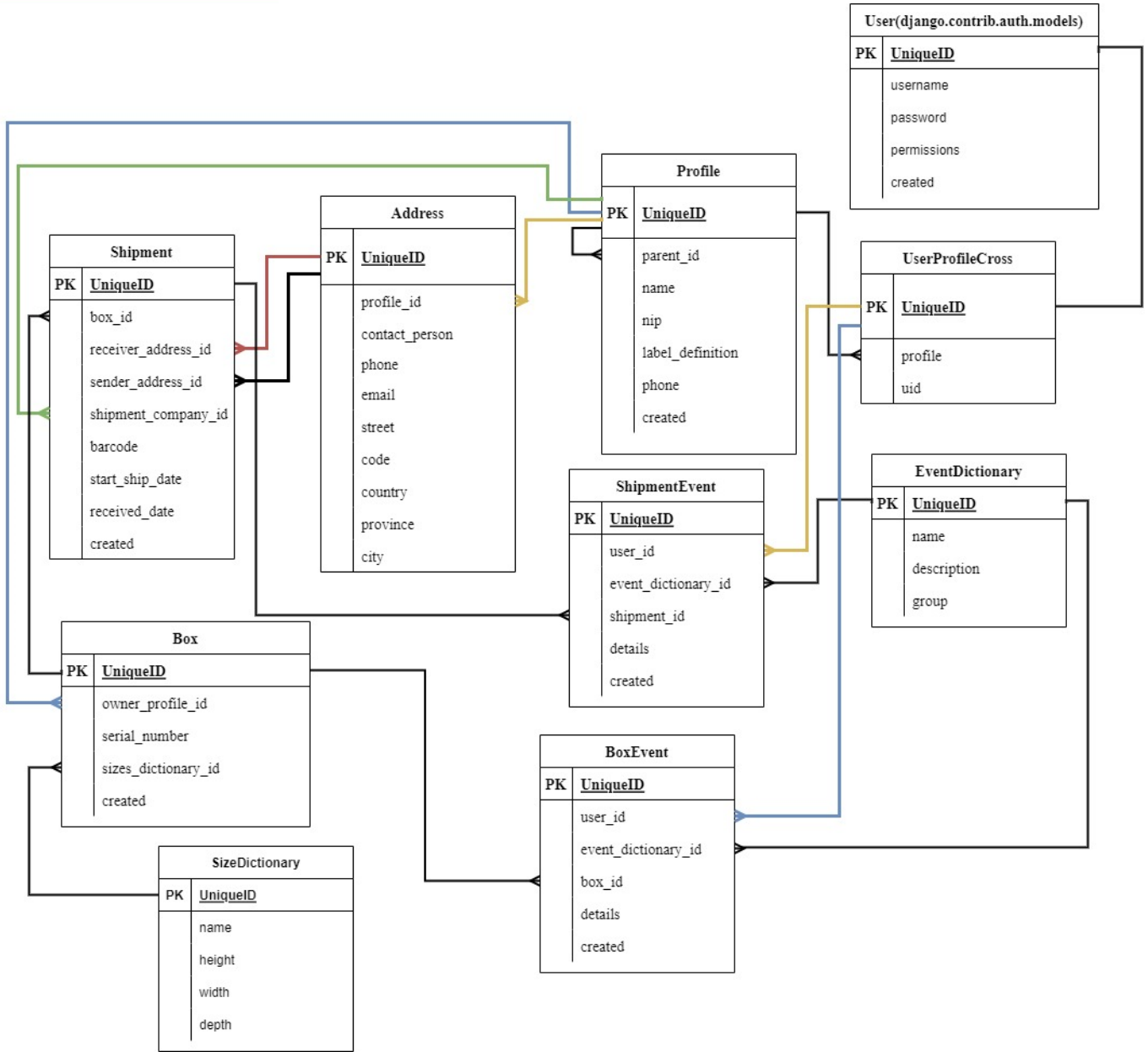
# Przykład 2 AO

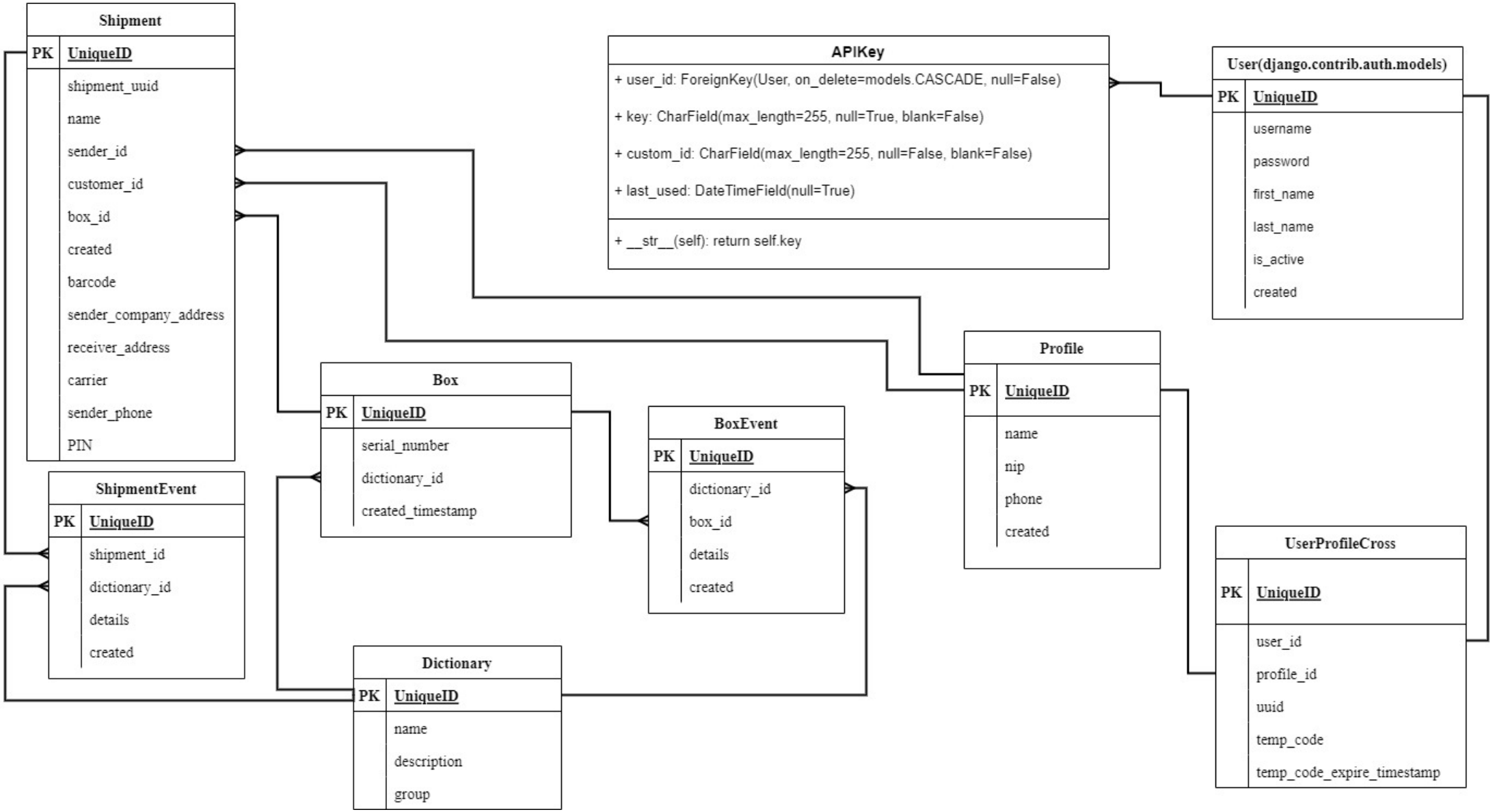
# Moje doświadczenie w Polsce:

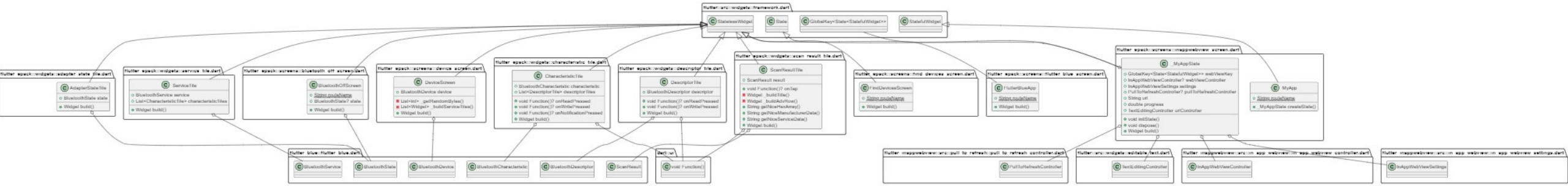
## e-Pack – Inteligentne opakowanie zwrotne dla branży e-commerce




EPACK
epack.pit.lukasiewicz.gov.pl/login?next=/
epack
Email *
username@domain.com
Password *
*******
Sign in
Register account
Forgot password?


User(django.contrib.auth.models)
PK UniqueID
username
password
permissions
created
Profile
PK UniqueID
parent_id
name
nip
label_definition
phone
created
Address
PK UniqueID
profile_id
contact_person
phone
email
street
code
country
province
city
Shipment
PK UniqueID
box_id
receiver_address_id
sender_address_id
shipment_company_id
barcode
start_ship_date
received_date
created
UserProfileCross
PK UniqueID
profile
uid
EventDictionary
PK UniqueID
name
description
group
ShipmentEvent
PK UniqueID
user_id
event_dictionary_id
shipment_id
details
created
Box
PK UniqueID
owner_profile_id
serial_number
sizes_dictionary_id
created
BoxEvent
PK UniqueID
user_id
event_dictionary_id
box_id
details
created
SizeDictionary
PK UniqueID
name
height
width
depth


Shipment
PK UniqueID
shipment_uuid
name
sender_id
customer_id
box_id
created
barcode
sender_company_address
receiver_address
carrier
sender_phone
PIN
APIKey
+ user_id: ForeignKey(User, on_delete=models.CASCADE, null=False)
+ key: CharField(max_length=255, null=True, blank=False)
+ custom_id: CharField(max_length=255, null=False, blank=False)
+ last_used: DateTimeField(null=True)
+ __str__(self): return self.key
User(django.contrib.auth.models)
PK UniqueID
username
password
first_name
last_name
is_active
created
Box
PK UniqueID
serial_number
dictionary_id
created_timestamp
BoxEvent
PK UniqueID
dictionary_id
box_id
details
created
Profile
PK UniqueID
name
nip
phone
created
ShipmentEvent
PK UniqueID
shipment_id
dictionary_id
details
created
Dictionary
PK UniqueID
name
description
group
UserProfileCross
PK UniqueID
user_id
profile_id
uuid
temp_code
temp_code_expire_timestamp


StatelessWidget
State
GlobalKey<State<StatefulWidget>>
StatefulWidget
AdapterStateTile
ServiceTile
BluetoothOffScreen
DeviceScreen
CharacteristicTile
DescriptorTile
ScanResultTile
FindDevicesScreen
FlutterBlueApp
MyAppState
MyApp
BluetoothService
BluetoothState
BluetoothDevice
BluetoothCharacteristic
BluetoothDescriptor
ScanResult
PullToRefreshController
TextEditingController
InAppWebViewController
InAppWebViewSettings

# UŻYCIE ANALIZA I PROJEKTOWANIE OBIEKTOWE W SWOICH PROJEKTACH!

# KAHOOT

# Dziękuję za uwagę!