

# Projekt Bazy Danych 2

## Dokumentacja aplikacji

Adrian Mrzygłód

Aplikacja desktopowa do zarządzania biblioteką gier. Do napisania aplikacji użyto:

Java

JPA

Apache Derby

JavaFX

Funkcje aplikacji:

- Można założyć konto,
- Można kupować kody(klucze) do gier na platformie,
- Można aktywować grę za pomocą kodów,
- Można doładować konto,
- Można zapisać się na turnieje w grach, które się posiada,

Dodatkowo użytkownik firmowy może tworzyć zapisy na turnieje, które organizuje.

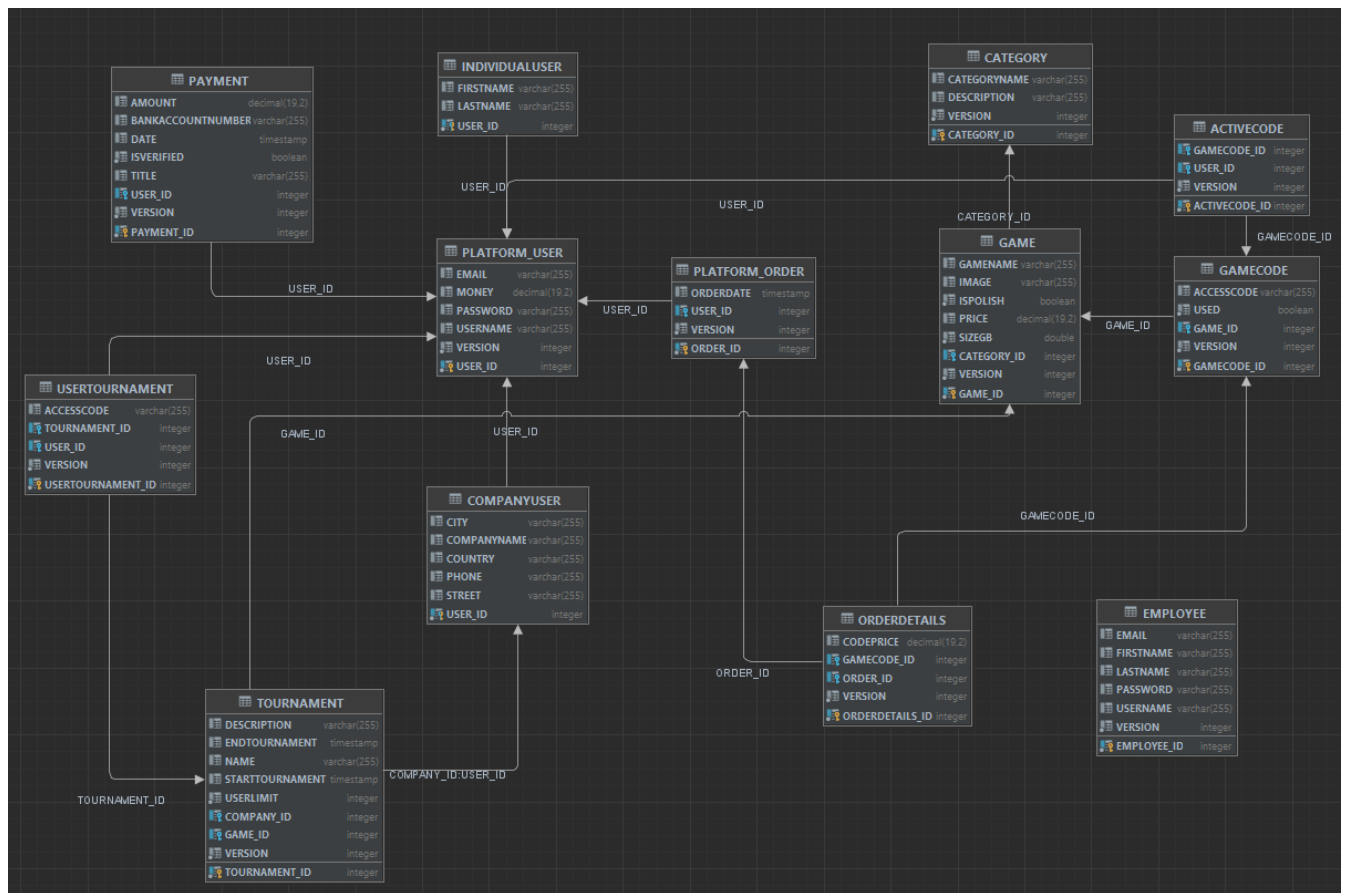
Pracownik może:

- Dodać grę, kod oraz kategorię,
- Zatwierdzić wpłatę użytkownika.

# Baza Danych

W aplikacji użyto bazy danych Apache Derby. Tabele były mapowane na podstawie klas za pomocą JPA.

Diagram:



# Tabele:

## 1.Category

Tabela jest generowana na podstawie klasy Category:

```
@Entity
public class Category {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int Category_ID;
    private String categoryName;

    private String description;

    @OneToMany(mappedBy = "category")
    private List<Game> games;

    @Version
    @Column(name = "version", nullable = false, columnDefinition =
"INTEGER DEFAULT 0")
    private int version;

    public Category() {
    }
    public Category(String categoryName, String description) {

        this.categoryName=categoryName;
        this.description=description;
        this.games= new ArrayList<>();
    }
    public String getCategoryName() {
        return categoryName;
    }

    public int getCategoryID() {
        return Category_ID;
    }

    public String getDescription() {
        return description;
    }

    public List<Game> getGames() {
        return games;
    }

    public void addGameToCategory(Game game) {
        this.games.add(game);
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Category category)) return false;
        return Category_ID == category.Category_ID;
    }
}
```

```
@Override
public int hashCode() {
    return Objects.hash(Category_ID);
}
```

Kod:

```
create table CATEGORY
(
    CATEGORY_ID    INTEGER            not null
                  primary key,
    CATEGORYNAME   VARCHAR(255),
    DESCRIPTION    VARCHAR(255),
    VERSION        INTEGER default 0 not null
);
```

Klasa zawiera informacje o kategoriach gier, czyli nazwa oraz opis.

Mapowanie:

```
@OneToMany(mappedBy = "category")
private List<Game> games;
```

Powoduje, że istnieje relacja One-To-Many między tabelami Category a Game.

## 2.Game

Tabela jest generowana na podstawie klasy Game:

```
@Entity
public class Game {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int Game_ID;
    private String gameName;
    private String image;
    private BigDecimal price;
    private BigDecimal sizeGB;
    private boolean isPolish;

    @OneToMany(mappedBy = "game")
    private List<GameCode> gameCodes;
    @ManyToOne
    @JoinColumn(name = "Category_ID")
    private Category category;
    @OneToMany(mappedBy = "game")
    private List<Tournament> tournaments;

    @Version
    @Column(name = "version", nullable = false, columnDefinition =
"INTEGER DEFAULT 0")
    private int version;

    public Game(){
    }

    public Game(String gameName,Category category,String image, BigDecimal
price, BigDecimal sizeGB, boolean isPolish) {
        this.gameName = gameName;
        this.category = category;
        category.addGameToCategory(this);
        this.image=image;
        this.price = price;
        this.sizeGB = sizeGB;
        this.isPolish = isPolish;
        this.gameCodes= new ArrayList<GameCode>();
        this.tournaments= new ArrayList<Tournament>();
    }

    public String getGameName() {
        return gameName;
    }

    public Category getCategory() {
        return category;
    }

    public BigDecimal getPrice() {
        return price;
    }

    public BigDecimal getSizeGB() {
        return sizeGB;
    }
}
```

```

    public boolean getIsPolish() {
        return isPolish;
    }

    public String getIsPolishString(){
        return Boolean.toString(this.isPolish);
    }

    public void setIsPolish(boolean isPolish) {
        this.isPolish = isPolish;
    }

    public String getImage() {
        return image;
    }

    public int getGame_ID() {
        return Game_ID;
    }

    public List<GameCode> getGameCodes() {
        return gameCodes;
    }

    public void addGameCode(GameCode gameCode) {
        this.gameCodes.add(gameCode);
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (!(o instanceof Game game)) return false;
        return getGame_ID() == game.getGame_ID();
    }

    @Override
    public int hashCode() {
        return Objects.hash(getGame_ID());
    }
}

```

Kod:

```

create table GAME
(
    GAME_ID      INTEGER          not null
                primary key,
    GAMENAME     VARCHAR(255),
    IMAGE        VARCHAR(255),
    ISPOLISH     BOOLEAN          not null,
    PRICE        DECIMAL(19, 2),
    SIZEGB       DOUBLE          not null,
    CATEGORY_ID  INTEGER
                constraint FKGCX2LIG7O3NBPO3JI8DKRBO2
                references CATEGORY,
    VERSION      INTEGER default 0 not null
);

```

Klasa zawiera informacje o grach, czyli nazwa gry, nazwa zdjęcia, cena, rozmiar w GB, a także czy jest Polska wersja językowa. Klasa nie posiada pola ilości gier, ponieważ zależy ona od ilości kodów do gier, a ilość kodów można pobrać za pomocą zapytania sql.

Mapowanie:

```
@OneToMany(mappedBy = "game")  
private List<GameCode> gameCodes;
```

Powoduje, że istnieje relacja One-To-Many między tabelami Game a GameCode.

Jedna gra posiada wiele kodów dostępowych. Są one unikalne.

Mapowanie:

```
@ManyToOne  
@JoinColumn(name = "Category_ID")  
private Category category;
```

Powoduje że istnieje relacja One-To-Many między tabelami Category a Game. Jedna gra może posiadać jedną kategorię. Wiele gier może należeć do jednej kategorii.

Mapowanie:

```
@OneToMany(mappedBy = "game")  
private List<Tournament> tournaments;
```

Powoduje, że istnieje relacja One-To-Many między tabelami Game i Tournament. Aby zorganizować turniej należy wskazać w jaką grę się gra. Dlatego jedna gra może mieć wiele turniejów, które są rozgrywane w tej grze.



### 3. GAMECODE

Tabela jest generowana na podstawie klasy GAMECODE:

```
@Entity
public class GameCode {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int gameCode_ID;

    private String accessCode;
    private boolean used;

    @ManyToOne
    @JoinColumn(name="game_id")
    private Game game;

    @OneToOne(mappedBy = "gameCode")
    private OrderDetails orderDetails;

    @OneToOne(mappedBy = "gameCode")
    private ActiveCode activeCode;

    @Version
    @Column(name = "version", nullable = false, columnDefinition =
"INTEGER DEFAULT 0")
    private int version;

    public GameCode() {
    }

    public GameCode(String accessCode, Game game) {
        this.accessCode = accessCode;
        this.used = false;
        this.game = game;
        game.addGameCode(this);
    }

    public String getAccessCode() {
        return accessCode;
    }

    public boolean isUsed() {
        return used;
    }

    public Game getGame() {
        return game;
    }

    public void setUsed(boolean used) {
        this.used = used;
    }

    public int getGameCode_ID() {
        return gameCode_ID;
    }

    public OrderDetails getOrderDetails() {
```

```

        return orderDetails;
    }

    public void setOrderDetails(OrderDetails orderDetails) {
        this.orderDetails = orderDetails;
    }

    public ActiveCode getActiveCode() {
        return activeCode;
    }

    public void setActiveCode(ActiveCode activeCode) {
        this.activeCode = activeCode;
        this.used=true;
    }
}

```

Kod:

```

create table GAMECODE
(
    GAMECODE_ID INTEGER          not null
        primary key,
    ACCESSCODE  VARCHAR(255),
    USED        BOOLEAN          not null,
    GAME_ID     INTEGER
        constraint FKSJAPDB6SL6C7OTRSGH5W5RSD3
        references GAME,
    VERSION     INTEGER default 0 not null
);

```

Klasa zawiera informacje o kodach do gier, czyli kod dostępowy do gry oraz informację czy kod został użyty.

Mapowanie:

```

@ManyToOne
@JoinColumn(name="game_id")
private Game game;

```

Powoduje, że istnieje relacja One-To-Many między tabelami Game a GameCode. Jedna gra posiada wiele kodów dostępowych. Wiele kodów dostępowych może należeć do jednej gry. Kody dostępowe są unikalne.

Mapowanie:

```

@OneToOne(mappedBy = "gameCode")
private OrderDetails orderDetails;

```

Powoduje, że istnieje relacja One-To-One między tabelami OrderDetails a GameCode. Jeden obiekt szczegółów zamówienia posiada jeden kod dostępowy. Jeden kod dostępowy może należeć do jednego szczegółu zamówienia. Ta relacja służy do przypisania kodu do zamówienia poprzez tabelę OrderDetails. Po przypisaniu obiektu OrderDetails kod jest kupiony.

Mapowanie:

```
@OneToOne(mappedBy = "gameCode")  
private ActiveCode activeCode;
```

Powoduje, że istnieje relacja One-To-One między tabelami ActiveCode a GameCode. Jeden obiekt aktywnego kodu posiada jeden kod dostępowy. Jeden kod dostępowy może należeć do jednego obiektu aktywnego kodu. Ta relacja służy do aktywowania kodu dla użytkownika poprzez tabelę ActiveCode. Po przypisaniu obiektu ActiveCode kod jest aktywowany, zmieniana jest też wartość pola used na true.

## 4. ActiveCode

Tabela jest generowana na podstawie klasy ActiveCode:

```
@Entity
public class ActiveCode {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int ActiveCode_ID;

    @OneToOne
    @JoinColumn(name = "GameCode_ID")
    private GameCode gameCode;

    @ManyToOne
    @JoinColumn(name="user_id")
    private PlatformUser user;

    @Version
    @Column(name = "version", nullable = false, columnDefinition =
"INTEGER DEFAULT 0")
    private int version;

    public ActiveCode() {
    }

    public ActiveCode(PlatformUser platformUser, GameCode gameCode) {
        this.gameCode=gameCode;
        this.gameCode.setActiveCode(this);
        this.user=platformUser;
        this.user.addActiveCode(this);
    }

    public int getActiveCode_ID() {
        return ActiveCode_ID;
    }

    public GameCode getGameCode() {
        return gameCode;
    }

    public PlatformUser getUser() {
        return user;
    }

}
```

Kod:

```
create table ACTIVECODE
(
    ACTIVECODE_ID INTEGER          not null
        primary key,
    GAMECODE_ID   INTEGER
        constraint FKFPFAYFKHQKFTM1E9FSQFHFXXKQ
        references GAMECODE,
    USER_ID       INTEGER
        constraint FK8YO35B17CLLIQ9RDP3IABQBHR
```

```

        references PLATFORM_USER,
VERSION          INTEGER default 0 not null
);

```

Klasa służy do aktywowania kodu do gry. Gdy użytkownik aktywuje kod, to tworzony jest obiekt klasy ActiveCode do którego przypisywane są obiekty klas PlatformUser oraz GameCode. Po stworzeniu tej obiektu tej klasy kod gry zostaje aktywowany dla danego użytkownika.

Mapowanie:

```

@OneToOne
@JoinColumn(name = "GameCode_ID")
private GameCode gameCode;

```

Powoduje, że istnieje relacja One-To-One między tabelami ActiveCode a GameCode. Jeden obiekt aktywnego kodu posiada jeden kod dostępowy. Jeden kod dostępowy może należeć do jednego obiektu aktywnego kodu. Ta relacja służy do aktywowania kodu dla użytkownika poprzez tabele ActiveCode.

Mapowanie:

```

@ManyToOne
@JoinColumn(name="user_id")
private PlatformUser user;

```

Powoduje, że istnieje relacja One-To-Many między tabelami PlatformUser a ActiveCode. Użytkownik może mieć wiele aktywnych kodów do gier. Wiele aktywnych kodów może mieć jednego użytkownika. Użytkownik może aktywować kod jeśli nie był aktywowany, był kupiony, oraz użytkownik nie ma aktywnej danej gry, którą kod aktywuje (czyli wśród obiektów ActiveCode które posiada użytkownik nie ma kodu, który aktywuje daną grę).

## 5.PLATFORM\_USER

Tabela jest generowana na podstawie klasy PlatformUser:

```
@Entity
@Table(name = "PLATFORM_USER")
@Inheritance(strategy = InheritanceType.JOINED)
public class PlatformUser {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int User_ID;
    private String username;
    private String password;
    private String email;
    private BigDecimal money;

    @OneToMany
    @JoinColumn(name = "USER_ID")
    private List<PlatformOrder> platformOrders;

    @OneToMany
    @JoinColumn(name = "USER_ID")
    private List<Payment> payments;

    @OneToMany(mappedBy = "user")
    private List<ActiveCode> activeCodes;

    @OneToMany(mappedBy = "user")
    private List<UserTournament> userTournaments;

    @Version
    @Column(name = "version", nullable = false, columnDefinition =
"INTEGER DEFAULT 0")
    private int version;

    public PlatformUser(String username, String password, String email) {
        this.username = username;
        this.password = password;
        this.email = email;
        this.money=BigDecimal.valueOf(0);
        this.platformOrders = new ArrayList<PlatformOrder>();
        this.activeCodes= new ArrayList<ActiveCode>();
        this.userTournaments= new ArrayList<UserTournament>();
        this.payments= new ArrayList<Payment>();
    }

    public PlatformUser() {
    }

    public String getUsername() {
        return username;
    }

    public String getPassword() {
        return password;
    }

    public String getEmail() {
        return email;
    }
}
```

```

    public BigDecimal getMoney() {
        return money;
    }

    public void setMoney(BigDecimal money) {
        this.money = money;
    }

    public void updateMoney(BigDecimal cash) {
        this.money = money.add(cash);
    }

    public boolean canBuy(BigDecimal cash) {
        return this.money.subtract(cash).compareTo(BigDecimal.valueOf(0)) > 0
? true : false;
    }

    public void addOrder(PlatformOrder platformOrder) {
        this.platformOrders.add(platformOrder);
    }

    public void addActiveCode(ActiveCode activeCode) {
        this.activeCodes.add(activeCode);
    }

    public int getUser_ID() {
        return User_ID;
    }

    public List<PlatformOrder> getPlatformOrders() {
        return platformOrders;
    }

    public List<ActiveCode> getActiveCodes() {
        return activeCodes;
    }

    public List<UserTournament> getUserTournaments() {
        return userTournaments;
    }

    public void addUserTournament(UserTournament userTournament) {
        this.getUserTournaments().add(userTournament);
    }

    public void addPaymentToUser(Payment payment) {
        this.payments.add(payment);
    }

    public List<Payment> getPayments() {
        return payments;
    }
}

```

Kod:

```

create table PLATFORM_USER
(
    USER_ID INTEGER not null
        primary key,
    EMAIL VARCHAR(255),

```

```

    MONEY    DECIMAL(19, 2),
    PASSWORD VARCHAR(255),
    USERNAME VARCHAR(255),
    VERSION  INTEGER default 0 not null
);

```

Klasa zawiera informacje o użytkownikach aplikacji, czyli nazwę użytkownika, hasło, email, ilość pieniędzy. Adnotacja `@Table` służy do nazwania tabeli inaczej niż nazwa klasy, zaś `@Inheritance` wskazuje, że jest to klasa podstawowa modelująca dziedziczenie. Jest to klasa którą rozszerzają klasy `IndividualUser` oraz `CompanyUser`. W kontekście schematu bazy danych klasy dziedziczące to oddzielne tabele.

Mapowanie:

```

@OneToMany
@JoinColumn(name = "USER_ID")
private List<PlatformOrder> platformOrders;

```

Powoduje, że istnieje relacja One-To-Many między tabelami `PlatformUser` a `PlatformOrder`. Jeden użytkownik może zrobić wiele zamówień przy zakupie kodów do gier.

Mapowanie:

```

@OneToMany
@JoinColumn(name = "USER_ID")
private List<Payment> payments;

```

Powoduje, że istnieje relacja One-To-Many między tabelami `PlatformUser` a `Payment`. Jeden użytkownik może wykonać wiele wpłat, aby zasilić swoje konto pieniędzmi, z których może później kupować kody do gier.

Mapowanie:

```

@OneToMany(mappedBy = "user")
private List<ActiveCode> activeCodes;

```

Powoduje, że istnieje relacja One-To-Many między tabelami `PlatformUser` a `ActiveCode`. Użytkownik może mieć wiele aktywnych kodów do gier. Użytkownik aktywując kod do gry dodaje do swojego konta obiekt `ActiveCode`, który zawiera obiekt `GameCode`, czyli kod do gry który ma zostać aktywowany. Użytkownik może aktywować kod jeśli nie był aktywowany, był kupiony, oraz użytkownik nie ma aktywnej danej gry, którą kod aktywuje (czyli wśród obiektów `ActiveCode` które posiada użytkownik nie ma kodu, który aktywuje daną grę). Użytkownik może aktywować dla swojego konta każdą grę, ale tylko raz (jednym kodem).

Mapowanie:

```

@OneToMany(mappedBy = "user")
private List<UserTournament> userTournaments;

```

Powoduje, że istnieje relacja One-To-Many między tabelami `PlatformUser` a `UserTournament`. Tabela `UserTournament` jest tak naprawdę tabelą łącznikową relacji Many-To-Many między użytkownikiem a turniejem, jednak zawiera dodatkowo kod dostępu do turnieju. Jeden użytkownik może brać udział w wielu turniejach (może mieć wiele obiektów `UserTournament`). Użytkownik może brać udział w turnieju pod warunkiem że posiada aktywowaną grę, w którą rozgrywany jest turniej. Po zapisaniu dostaje kod do turnieju.



## 6. INDIVIDUALUSER

Tabela jest generowana na podstawie klasy IndividualUser:

```
@Entity
public class IndividualUser extends PlatformUser{

    private String firstname;
    private String lastname;

    public IndividualUser(){
    }

    public IndividualUser(String username, String password, String email,
String firstname, String lastname) {
        super(username, password, email);
        this.firstname = firstname;
        this.lastname = lastname;
    }

    public String getFirstname() {
        return firstname;
    }

    public void setFirstname(String firstname) {
        this.firstname = firstname;
    }

    public String getLastName() {
        return lastname;
    }

    public void setLastName(String lastname) {
        this.lastname = lastname;
    }
}
```

Kod:

```
create table INDIVIDUALUSER
(
    FIRSTNAME VARCHAR(255),
    LASTNAME  VARCHAR(255),
    USER_ID   INTEGER not null
        primary key
        constraint FKBNSVO3DGYLN3SOCQPGJDR3THL
            references PLATFORM_USER
);
```

Klasa rozszerza klasę PLATFORM\_USER, zawiera informacje o użytkownikach indywidualnych, takie jak imię i nazwisko. W kontekście schematu bazy danych jest to tabela z kluczem głównym oraz obcym, który jest kluczem głównym tabeli PLATFORM\_USER.

## 7. COMPANYUSER

Tabela jest generowana na podstawie klasy COMPANYUSER:

```
@Entity
public class CompanyUser extends PlatformUser{

    private String companyName;
    private String country;
    private String city;
    private String street;
    private String phone;

    @OneToMany(mappedBy = "companyUser")
    private List<Tournament> tournaments;

    public CompanyUser() {
    }

    public CompanyUser(String username, String password, String email,
String companyName, String city, String street, String phone,String
country) {
        super(username, password, email);
        this.companyName = companyName;
        this.country=country;
        this.city = city;
        this.street = street;
        this.phone = phone;
        this.tournaments= new ArrayList<Tournament>();
    }

    public String getCompanyName() {
        return companyName;
    }

    public void setCompanyName(String companyName) {
        this.companyName = companyName;
    }

    public String getCity() {
        return city;
    }

    public void setCity(String city) {
        this.city = city;
    }

    public String getStreet() {
        return street;
    }

    public void setStreet(String street) {
        this.street = street;
    }

    public String getPhone() {
        return phone;
    }

    public void setPhone(String phone) {
        this.phone = phone;
    }
}
```

```

    }

    public List<Tournament> getTournaments() {
        return tournaments;
    }

    public void addTournament(Tournament tournament){
        this.tournaments.add(tournament);
    }

    public String getCountry() {
        return country;
    }

    public void setCountry(String country) {
        this.country = country;
    }
}

```

Kod:

```

create table COMPANYUSER
(
    CITY          VARCHAR(255),
    COMPANYNAME   VARCHAR(255),
    COUNTRY       VARCHAR(255),
    PHONE         VARCHAR(255),
    STREET        VARCHAR(255),
    USER_ID       INTEGER not null
                primary key
                constraint FK8IRBLIO1MS0SNY8JIJFEEK7MJ
                references PLATFORM_USER
);

```

Klasa rozszerza klasę PLATFORM\_USER, zawiera informacje o użytkownikach firmowych, takie jak nazwa firmy, kraj, miasto, ulica oraz telefon. W kontekście schematu bazy danych jest to tabela z kluczem głównym oraz obcym, który jest kluczem głównym tabeli PLATFORM\_USER.

Mapowanie:

```

@OneToMany(mappedBy = "companyUser")
private List<Tournament> tournaments;

```

Powoduje, że istnieje relacja One-To-Many między tabelami CompanyUser a Tournament. Tylko użytkownik firmowy może stworzyć turniej. Firma może stworzyć wiele turniejów.

## 8. PAYMENT

Tabela jest generowana na podstawie klasy PAYMENT:

```
@Entity
public class Payment {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int payment_ID;

    @ManyToOne
    @JoinColumn(name="user_id")
    private PlatformUser platformUser;

    private BigDecimal amount;

    private String bankAccountNumber;

    private Timestamp date;
    private String title;
    private boolean isVerified;

    @Version
    @Column(name = "version", nullable = false, columnDefinition =
"INTEGER DEFAULT 0")
    private int version;

    public Payment() {
    }

    public Payment(BigDecimal amount, String bankAccountNumber, Timestamp
date, String title) {
        this.amount = amount;
        this.bankAccountNumber = bankAccountNumber;
        this.date = date;
        this.title = title;
        this.isVerified=false;
    }

    public int getPayment_ID() {
        return payment_ID;
    }

    public BigDecimal getAmount() {
        return amount;
    }

    public String getBankAccountNumber() {
        return bankAccountNumber;
    }

    public Timestamp getDate() {
        return date;
    }

    public String getTitle() {
        return title;
    }
}
```

```

    public boolean isVerified() {
        return isVerified;
    }

    public void setVerified(boolean verified) {
        isVerified = verified;
    }

    public PlatformUser getPlatformUser() {
        return platformUser;
    }

    public void setPlatformUser(PlatformUser platformUser) {
        this.platformUser = platformUser;
    }
}

```

Kod:

```

create table PAYMENT
(
    PAYMENT_ID          INTEGER          not null
        primary key,
    AMOUNT              DECIMAL(19, 2),
    BANKACCOUNTNUMBER   VARCHAR(255),
    DATE               TIMESTAMP,
    ISVERIFIED          BOOLEAN          not null,
    TITLE               VARCHAR(255),
    USER_ID            INTEGER
        constraint FKEWSN75PS3D68CWDGACKUY5S5L
        references PLATFORM_USER,
    VERSION             INTEGER default 0 not null
);

```

Klasa zawiera informacje o doładowaniach kont użytkowników, czyli wartość przelewu, numer konta bankowego, data wpłaty, tytuł przelewu, oraz informacje czy zweryfikowano przelew. Wpłata zostanie przypisana do konta użytkownika w momencie gdy pracownik zweryfikuje wpłatę.

Mapowanie:

```

@ManyToOne
@JoinColumn(name="user_id")
private PlatformUser platformUser;

```

Powoduje, że istnieje relacja One-To-Many między tabelami PlatformUser a Payment. Jeden użytkownik może wykonać wiele wpłat. Wiele wpłat może być przypisane do jednego użytkownika.

## 9. PlatformOrder

Tabela jest generowana na podstawie klasy PlatformOrder:

```
@Entity
@Table(name = "PLATFORM_ORDER")
public class PlatformOrder {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int Order_ID;
    @OneToMany(mappedBy = "order", cascade = CascadeType.PERSIST)
    private List<OrderDetails> orderDetails;

    @ManyToOne
    @JoinColumn(name="user_id")
    private PlatformUser platformUser;

    private Timestamp orderDate;

    @Version
    @Column(name = "version", nullable = false, columnDefinition =
"INTEGER DEFAULT 0")
    private int version;

    @PrePersist
    protected void onCreate() {
        this.orderDate = new Timestamp(System.currentTimeMillis());
    }

    public PlatformOrder(){
    }

    public PlatformOrder(PlatformUser platformUser){
        this.platformUser = platformUser;
        platformUser.addOrder(this);
        this.orderDetails = new ArrayList<OrderDetails>();
    }
    public void addDetailsToOrder(OrderDetails details){
        this.orderDetails.add(details);
    }

    public int getOrder_ID() {
        return Order_ID;
    }

    public Timestamp getOrderDate() {
        return orderDate;
    }

    public List<OrderDetails> getOrderDetails() {
        return orderDetails;
    }
}
```

Kod:

```
create table PLATFORM_ORDER
(
    ORDER_ID INTEGER          not null
        primary key,
```

```
ORDERDATE TIMESTAMP,
USER_ID INTEGER
    constraint FKQJ5MDVURTEECHV4BIOJ37CG2T
        references PLATFORM_USER,
VERSION INTEGER default 0 not null
);
```

Klasa zawiera informacje o zamówieniach, takie jak data zamówienia. Służy do składania zamówień przez użytkownika. Użytkownik w momencie zakupu tworzy zamówienie, do zamówienia dołączane są kody gier za pomocą tabeli OrderDetails, a kwota na jego koncie zmniejsza się o cenę kodów.

Mapowanie:

```
@OneToMany(mappedBy = "order", cascade = CascadeType.PERSIST)
private List<OrderDetails> orderDetails;
```

Powoduje, że istnieje relacja One-To-Many między tabelami PlatformOrder a OrderDetails. Jedno zamówienie może mieć wiele szczegółów zamówienia, zaś wiele szczegółów zamówienia może należeć do jednego zamówienia. Relacja ta służy do przyłączania kodów do gier do zamówienia za pomocą tabeli OrderDetails.

Mapowanie:

```
@ManyToOne
@JoinColumn(name="user_id")
private PlatformUser platformUser;
```

Powoduje, że istnieje relacja One-To-Many między tabelami PlatformUser a PlatformOrder. Jeden użytkownik może zrobić wiele zamówień przy zakupie kodów do gier. Wiele zamówień może należeć do jednego użytkownika.

## 10. OrderDetails

Tabela jest generowana na podstawie klasy OrderDetails:

```
@Entity
public class OrderDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int OrderDetails_ID;

    private BigDecimal codePrice;

    @OneToOne
    @JoinColumn(name = "GameCode_ID")
    private GameCode gameCode;

    @ManyToOne
    @JoinColumn(name="order_id")
    private PlatformOrder order;

    @Version
    @Column(name = "version", nullable = false, columnDefinition =
"INTEGER DEFAULT 0")
    private int version;

    public OrderDetails() {
    }

    public OrderDetails(BigDecimal codePrice, PlatformOrder order, GameCode
gameCode) {
        this.codePrice=codePrice;
        this.gameCode=gameCode;
        gameCode.setOrderDetails(this);
        this.order = order;
        this.order.addDetailsToOrder(this);
    }

    public int getOrderDetails_ID() {
        return OrderDetails_ID;
    }

    public BigDecimal getCodePrice() {
        return codePrice;
    }

    public GameCode getGameCode() {
        return gameCode;
    }

    public PlatformOrder getOrder() {
        return order;
    }
}
```



Kod:

```
create table ORDERDETAILS
(
    ORDERDETAILS_ID INTEGER          not null
        primary key,
    CODEPRICE        DECIMAL(19, 2),
    GAMECODE_ID      INTEGER
        constraint FK3QBWM0IH4NL1ALYVE7LO7FM1T
        references GAMECODE,
    ORDER_ID         INTEGER
        constraint FKQ8LOKVG4PFL4GDKX6YL57E771
        references PLATFORM_ORDER,
    VERSION          INTEGER default 0 not null
);
```

Klasa zawiera informacje o szczegółach zamówienia, takie jak cena kodu. Służy do przyłączenia kodu do zamówienia. Po przypisaniu tego obiektu do zamówienia kod zostaje kupiony.

Mapowanie:

```
@OneToOne
@JoinColumn(name = "GameCode_ID")
private GameCode gameCode;
```

Powoduje, że istnieje relacja One-To-One między tabelami OrderDetails a GameCode. Jeden obiekt szczegółów zamówienia posiada jeden kod dostępowy. Jeden kod dostępowy może należeć do jednego szczegółu zamówienia. Ta relacja służy do przypisania kodu do zamówienia poprzez tabelę OrderDetails.

Mapowanie:

```
@ManyToOne
@JoinColumn(name="order_id")
private PlatformOrder order;
```

Powoduje, że istnieje relacja One-To-Many między tabelami PlatformOrder a OrderDetails. Jedno zamówienie może mieć wiele szczegółów zamówienia, zaś wiele szczegółów zamówienia może należeć do jednego zamówienia. Relacja ta służy do przyłączania kodów do gier do zamówienia za pomocą tabeli OrderDetails.

## 11. Tournament

Tabela jest generowana na podstawie klasy Tournament:

```
@Entity
public class Tournament {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int Tournament_ID;

    private String name;
    private Timestamp startTournament;
    private Timestamp endTournament;
    private String description;
    private int userLimit;

    @OneToMany(mappedBy = "tournament")
    private List<UserTournament> userTournaments;

    @ManyToOne
    @JoinColumn(name = "company_id")
    private CompanyUser companyUser;

    @ManyToOne
    @JoinColumn(name="game_id")
    private Game game;

    @Version
    @Column(name = "version", nullable = false, columnDefinition =
"INTEGER DEFAULT 0")
    private int version;

    public Tournament() {
    }

    public Tournament(String name, Timestamp startTournament, Timestamp
endTournament, String description, int userLimit, CompanyUser companyUser,
Game game) {
        this.name=name;
        this.startTournament = startTournament;
        this.endTournament = endTournament;
        this.description = description;
        this.userLimit = userLimit;
        this.companyUser = companyUser;
        this.game = game;
        this.userTournaments= new ArrayList<UserTournament>();
    }

    public Timestamp getStartTournament() {
        return startTournament;
    }

    public void setStartTournament(Timestamp startTournament) {
        this.startTournament = startTournament;
    }

    public Timestamp getEndTournament() {
        return endTournament;
    }
}
```

```

public void setEndTournament(Timestamp endTournament) {
    this.endTournament = endTournament;
}

public List<UserTournament> getUserTournaments() {
    return userTournaments;
}

public int getNumberOfPlayers(){
    return this.userTournaments.size();
}

public CompanyUser getCompanyUser() {
    return companyUser;
}

public Game getGame() {
    return game;
}

public int getTournament_ID() {
    return Tournament_ID;
}

public String getName() {
    return name;
}

public String getDescription() {
    return description;
}

public int getUserLimit() {
    return userLimit;
}

public void addUserTournament(UserTournament userTournament){
    this.userTournaments.add(userTournament);
}
}

```

Kod:

```

create table TOURNAMENT
(
    TOURNAMENT_ID    INTEGER                not null
                    primary key,
    DESCRIPTION      VARCHAR(255),
    ENDTOURNAMENT    TIMESTAMP,
    NAME             VARCHAR(255),
    STARTTOURNAMENT  TIMESTAMP,
    USERLIMIT        INTEGER                not null,
    COMPANY_ID       INTEGER
                    constraint FK7Q4AT9BHO1I8X7YNGWET0Y4VM
                    references COMPANYUSER,
    GAME_ID          INTEGER

```

```

        constraint FK3301SQ6PNIV0CP0L2UVNA5HQ8
        references GAME,
        VERSION          INTEGER default 0 not null
    );

```

Klasa zawiera informacje o turniejach, czyli nazwę turnieju, datę rozpoczęcia i zakończenia, opis, limit użytkowników biorących udział w turnieju. Służy do obsługi turniejów. Turniej może stworzyć użytkownik firmowy, brać udział może każdy użytkownik pod warunkiem odpowiedniej liczby miejsc i posiadania gry, w którą turniej jest rozgrywany.

Mapowanie:

```

@OneToMany(mappedBy = "tournament")
private List<UserTournament> userTournaments;

```

Powoduje, że istnieje relacja One-To-Many między tabelami Tournament a UserTournament. Tabela UserTournament jest tak naprawdę tabelą łącznikową relacji Many-To-Many między użytkownikiem a turniejem, jednak zawiera dodatkowo kod dostępowy do turnieju. W jednym turnieju może brać udział wielu użytkowników (może mieć wiele obiektów UserTournament). Do turnieju można dodawać użytkowników, pod warunkiem że są wolne miejsca.

Mapowanie:

```

@ManyToOne
@JoinColumn(name = "company_id")
private CompanyUser companyUser;

```

Powoduje, że istnieje relacja One-To-Many między tabelami CompanyUser a Tournament. Tylko użytkownik firmowy może stworzyć turniej. Firma może stworzyć wiele turniejów. Wiele turniejów może być stworzonych przez jedną firmę. Relacja służy do przypisywania do firmy turniejów które stworzyła.

Mapowanie:

```

@ManyToOne
@JoinColumn(name="game_id")
private Game game;

```

Powoduje, że istnieje relacja One-To-Many między tabelami Game i Tournament. Aby zorganizować turniej należy wskazać w jaką grę się gra. Dlatego jedna gra może mieć wiele turniejów, które są rozgrywane w tej grze. Wiele turniejów może być rozgrywane w jednej grze. Relacja służy do przypisania grze turniejów, które są w niej organizowane i grane.

## 12. USERTOURNAMENT

Tabela jest generowana na podstawie klasy USERTOURNAMENT:

```
@Entity
public class UserTournament {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int UserTournament_ID;

    @ManyToOne
    @JoinColumn(name="tournament_id")
    private Tournament tournament;

    @ManyToOne
    @JoinColumn(name="user_id")
    private PlatformUser user;

    private String accessCode;

    @Version
    @Column(name = "version", nullable = false, columnDefinition =
"INTEGER DEFAULT 0")
    private int version;

    public UserTournament() {
    }

    public UserTournament(Tournament tournament, PlatformUser user, String
accessCode) {
        this.tournament=tournament;
        this.user=user;
        this.accessCode=accessCode;
    }

    public Tournament getTournament() {
        return tournament;
    }

    public void setTournament(Tournament tournament) {
        this.tournament = tournament;
    }

    public PlatformUser getUser() {
        return user;
    }

    public void setUser(PlatformUser user) {
        this.user = user;
    }

    public String getAccessCode() {
        return accessCode;
    }

    public void setAccessCode(String accessCode) {
        this.accessCode = accessCode;
    }
}
```

Kod:

```
create table USERTOURNAMENT
(
    USERTOURNAMENT_ID INTEGER          not null
        primary key,
    ACCESSCODE         VARCHAR(255),
    TOURNAMENT_ID      INTEGER
        constraint FKQAOWNRUTVWPU0U8SKB4C2QA0P
        references TOURNAMENT,
    USER_ID            INTEGER
        constraint FKHFT3IV5VXXU3N1HM7MUO2RT44
        references PLATFORM_USER,
    VERSION            INTEGER default 0 not null
);
```

Klasa tworzy tabelę łącznikową relacji Many-To-Many między tabelami Tournament i PlatformUser, , jednak zawiera dodatkowo kod dostępowy do turnieju. Wiele użytkowników może mieć wiele turniejów. Stworzenie obiektu USERTOURNAMENT oznacza dodanie użytkownika do turnieju, dodatkowo generowany jest kod dla użytkownika dostępowy do turnieju.

Mapowanie:

```
@ManyToOne
@JoinColumn(name="tournament_id")
private Tournament tournament;
```

Powoduje, że istnieje relacja One-To-Many między tabelami Tournament a UserTournament. W jednym turnieju może brać udział wielu użytkowników (może mieć wiele obiektów UserTournament). Wiele użytkowników (obiektów UserTournament) może być przypisane do jednego turnieju. Do turnieju można dodawać użytkowników, pod warunkiem że są wolne miejsca.

Mapowanie:

```
@ManyToOne
@JoinColumn(name="user_id")
private PlatformUser user;
```

Powoduje, że istnieje relacja One-To-Many między tabelami PlatformUser a UserTournament. Jeden użytkownik może brać udział w wielu turniejach (może mieć wiele obiektów UserTournament). Wiele turniejów (obiektów UserTournament) może być przypisane do jednego użytkownika. Użytkownik może brać udział w turnieju pod warunkiem że posiada aktywowaną grę, w którą rozgrywany jest turniej. Po zapisaniu dostaje kod do turnieju.

## 13. Employee

Tabela jest generowana na podstawie klasy Employee:

```
@Entity
public class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int Employee_ID;
    private String username;
    private String password;
    private String firstname;
    private String lastname;
    private String email;

    @Version
    @Column(name = "version", nullable = false, columnDefinition =
"INTEGER DEFAULT 0")
    private int version;

    public Employee(String username, String password, String firstname,
String lastname, String email) {
        this.username = username;
        this.password = password;
        this.firstname = firstname;
        this.lastname = lastname;
        this.email = email;
    }

    public Employee() {

    }

    public String getUsername() {
        return username;
    }

    public String getPassword() {
        return password;
    }

    public String getFirstname() {
        return firstname;
    }

    public String getLastname() {
        return lastname;
    }

    public String getEmail() {
        return email;
    }

}
```

Kod:

```
create table EMPLOYEE
(
    EMPLOYEE_ID INTEGER          not null
        primary key,
    EMAIL        VARCHAR(255),
    FIRSTNAME    VARCHAR(255),
    LASTNAME     VARCHAR(255),
```

```
PASSWORD    VARCHAR(255),  
USERNAME    VARCHAR(255),  
VERSION     INTEGER default 0 not null  
) ;
```

Klasa zawiera informacje o pracownikach, czyli nazwę użytkownika, hasło, imię, nazwisko, email. Klasa służy do zamodelowania pracownika, który może dodawać gry, kody do gier, kategorie, a także weryfikować wpłaty od użytkowników.

Wszystkie klasy oraz tabele mają pole version oznaczone adnotacją @Version. Pole to jest automatycznie inkrementowane przez JPA przy każdej aktualizacji encji. W ten sposób można kontrolować, czy encja została zmodyfikowana przez innego użytkownika między pobraniem i zapisaniem. W ten sposób, jeśli encja została zmodyfikowana przez innego użytkownika między pobraniem i zapisaniem, zostanie zgłoszony wyjątek.



# Aplikacja

Aplikacja składa się z głównego pakietu w `src/main/java` czyli `com.example.project`. W nim trzymane są pakiety:

- Controllers- kontrolery interfejsu JavaFX
- Logic- część logiczna aplikacji, a w niej:
  - DatabaseClasses- klasy mapowane do tabel
  - Filters- filtry
  - MainController- jest w nim główna klasa kontrolująca aplikację
- Providers- klasa obsługująca połączenie między aplikacją a bazą danych

W pakiecie głównym jest klasa `Main` służąca do uruchomienia aplikacji.

Innym pakietem jest też pakiet `resources/com.example.project`, w którym są:

- pakiet `data`, w którym są zdjęcia gier
- pakiet `META-INF`, w którym jest plik konfiguracyjny `jpa.persistence.xml`
- klasy `.fxml`, definiujące okna graficzne

# Operacje w aplikacji

## 1. Rejestrowanie użytkownika

Można stworzyć użytkownika indywidualnego lub firmowego.

Tworzony jest użytkownik indywidualny:

GameNet

# GAMENET

## Rejestracja

☒ Indywidualna ☐ Firmowa

Nazwa użytkownika:

User

Hasło:

...

Powtórz hasło:

...

Imię:

T

Nazwisko:

N

Email:

tn@example.com

Powrót

Zarejestruj się

	USER_ID	EMAIL	MONEY	PASSWORD	USERNAME	VERSION
1	1	a@o2.pl	9071.20	a	Nowy	0
2	217	game1111@o2.pl	7744.49	b	GameDev	1
3	425	a@o2.pl	9789.20	a	AM	0
4	525	new@o2.pl	9699.20	a	Nowy2	0
5	1132	tn@example.com	10000.00	abc	User	0

	FIRSTNAME	LASTNAME	USER_ID
1	A	M	1
2	a	m	425
3	N	A	525
4	T	N	1132

Tworzony jest użytkownik firmowy:

GameNet

# GAMENET

## Rejestracja

☐ Indywidualna☒ Firmowa

Nazwa użytkownika:

Firma

Hasło:

...

Powtórz hasło:

...

Nazwa firmy:

Nowa Firma

Miasto:

San Francisco

Ulica:

River 11

Telefon:

111 111 111

Kraj:

USA

Email:

newcompany@example.com

Powrót

Zarejestruj się

	CITY	COMPANYNAME	COUNTRY	PHONE	STREET	USER_ID
1	San Francisco	GameDevelopment	USA	111 222 333	High	217
2	San Francisco	Nowa Firma	USA	111 111 111	River 11	1306

	USER_ID	EMAIL	MONEY	PASSWORD	USERNAME	VERSION
1	1	a@o2.pl	9071.20	a	Nowy	0
2	217	game1111@o2.pl	5953.11	b	GameDev	5
3	425	a@o2.pl	9789.20	a	AM	0
4	525	new@o2.pl	9699.20	a	Nowy2	0
5	1132	tn@example.com	6191.42	abc	User	3
6	1306	newcompany@example.com	10000.00	abc	Firma	0

Do dodawania do bazy danych użytkownika indywidualnego użyto funkcji:

```
public void addIndividualUser(IndividualUser individualUser) {
    final EntityManager manager = getManager();
    EntityTransaction etx = manager.getTransaction();
    try {
        etx.begin();

        manager.persist(individualUser);

        etx.commit();

    } catch (Exception exception) {
        if(etx.isActive()) {
            etx.rollback();
        }
    }
    finally {
        manager.close();
        setEntityManager();
    }
}
```

Podobna funkcja służy do dodania użytkownika firmowego:

```
public void addCompanyUser(CompanyUser companyUser) {
    final EntityManager manager = getManager();
    EntityTransaction etx = manager.getTransaction();
    try {
        etx.begin();

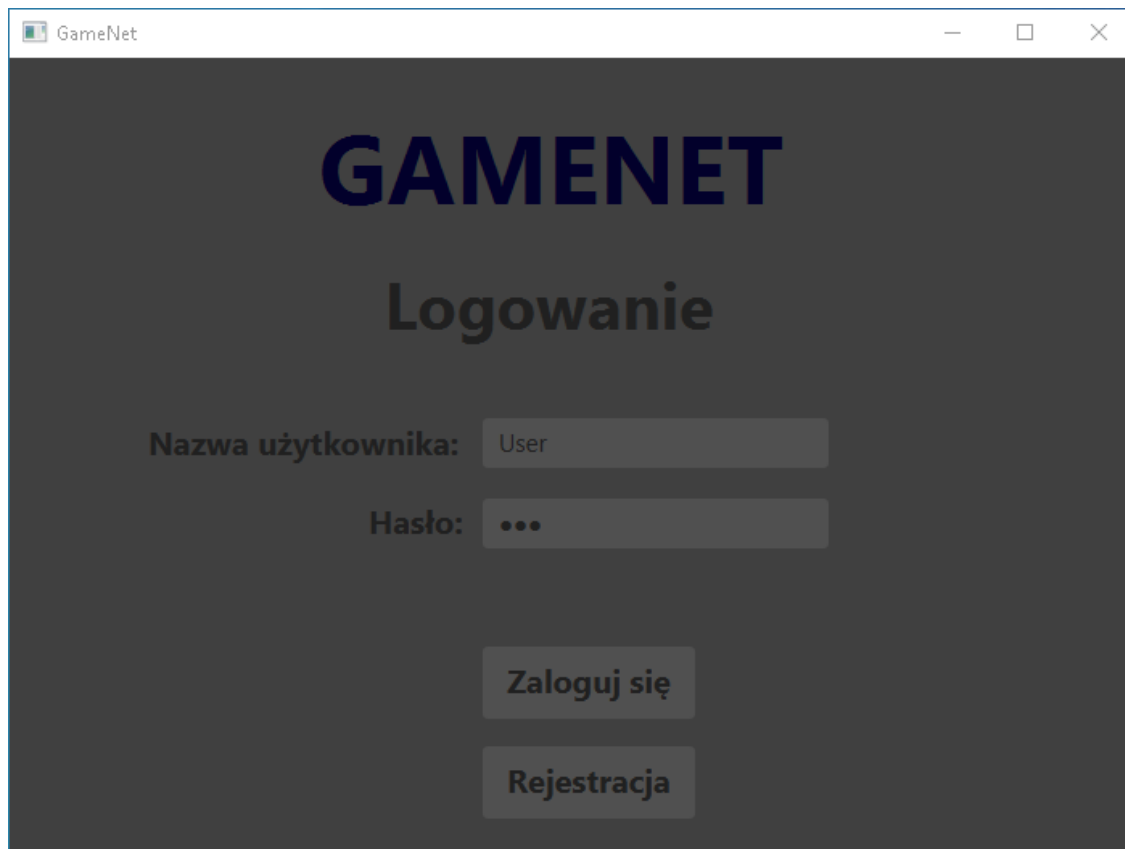
        manager.persist(companyUser);

        etx.commit();

    } catch (Exception exception) {
        if(etx.isActive()) {
            etx.rollback();
        }
    }
    finally {
        manager.close();
        setEntityManager();
    }
}
```

## 2. Logowanie użytkownika

Można się zalogować jako użytkownik indywidualny, firmowy lub pracownik:



The screenshot shows a web application window titled "GameNet". The main heading is "GAMENET" in large blue letters, followed by "Logowanie" in large white letters. Below the heading, there are two input fields: "Nazwa użytkownika:" with the text "User" and "Hasło:" with three dots. Below the fields are two buttons: "Zaloguj się" and "Rejestracja".

Do zalogowania korzysta się z funkcji:

```
public PlatformUser getUsersByName(String username){
    final EntityManager manager = getManager();

    Query query = manager.createQuery("SELECT u from PlatformUser u
where u.username= :username");
    query.setParameter("username",username);
    List<PlatformUser> platformUsers = query.getResultList();

    PlatformUser user=platformUsers.size()==0 ? null :
platformUsers.get(0);

    return user;
}
```

```
public Employee getEmployee(String username){
    final EntityManager manager = getManager();

    Query query = manager.createQuery("SELECT e from Employee e
where e.username= :username");
    query.setParameter("username",username);
    List<Employee> employees = query.getResultList();
```

```

Employee user=employees.size()==0 ? null : employees.get(0);

return user;
}

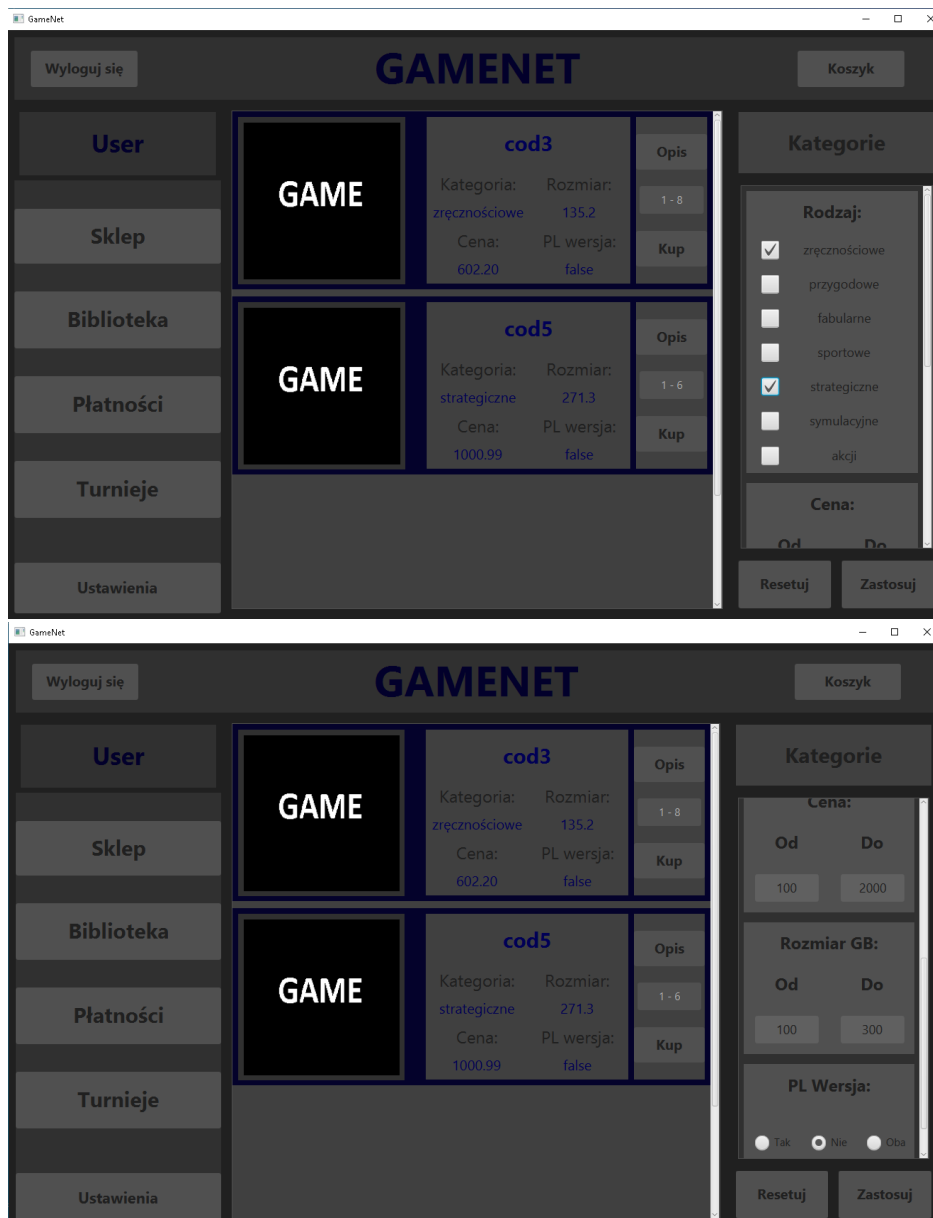
```

Pierwsza funkcja pobiera użytkownika z bazy danych, zaś druga pracownika. Weryfikuje się za ich pomocą czy użytkownik istnieje.

### 3. Filtrowanie i zakup gier przez użytkownika

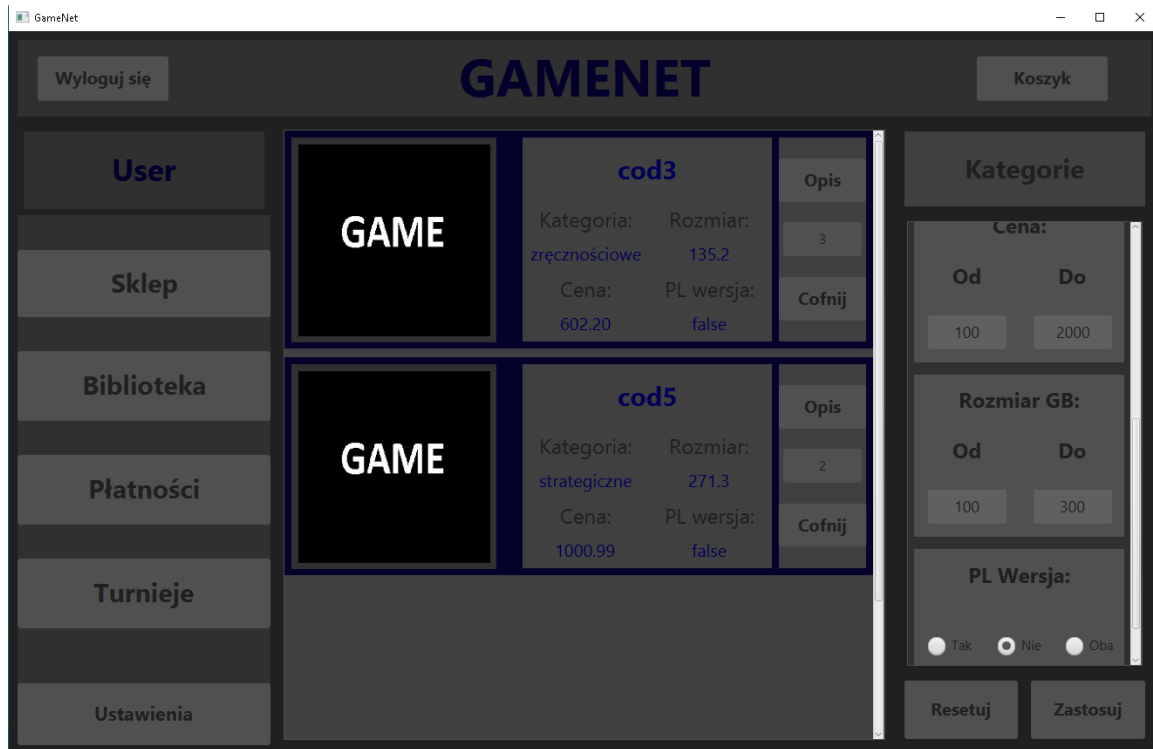
Po zalogowaniu można w oknie Sklep można filtrować i kupić gry (kody do gier).

Filtrowanie:

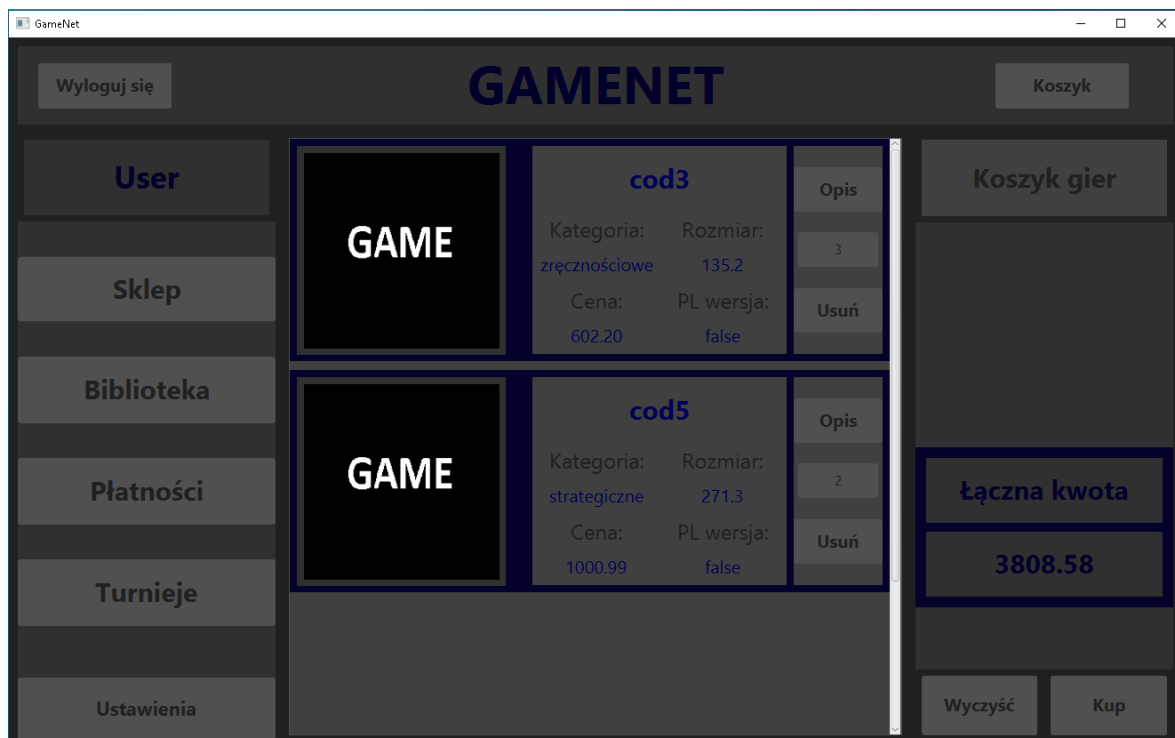


Powyższe gry zostały znalezione po zastosowaniu filtrów.

Następnie wybrano ilość sztuk do zakupu i zakupiono je:



Zakupione kody trafiły do okna Koszyka, widać nim gry oraz łączną cenę za zamówienie:



Po kliknięciu przycisku kup dodały się do bazy danych nowe dane:

Tabela PlatformOrder:

	ORDER_ID	ORDERDATE	USER_ID	VERSION
1	908	2023-06-10 18:25:46.573000000	217	0
2	910	2023-06-10 18:28:28.405000000	217	0
3	929	2023-06-10 20:54:18.944000000	217	0
4	931	2023-06-10 21:35:19.931000000	1	0
5	935	2023-06-10 21:36:09.918000000	425	0
6	1026	2023-06-12 14:41:39.173000000	217	0
7	1029	2023-06-12 15:31:48.547000000	217	0
8	1031	2023-06-12 15:40:40.234000000	217	0
9	1033	2023-06-12 16:07:50.626000000	217	0
10	1143	2023-06-12 20:41:39.263000000	217	0
11	1270	2023-06-12 20:52:59.002000000	217	0
12	1293	2023-06-12 21:12:04.712000000	217	0
13	1298	2023-06-12 21:14:30.214000000	1132	0

OrderID 1298

Tabela OrderDetails:

	ORDERDETAILS_ID	CODEPRICE	GAMECODE_ID	ORDER_ID	VERSION
1	904	210.80	825	908	0
2	905	210.80	834	908	0
3	906	210.80	835	908	0
4	907	602.20	854	908	0
5	909	102.73	863	910	0
6	927	102.73	864	929	0
7	928	102.73	865	929	0
8	930	210.80	836	931	0
9	934	210.80	837	935	0
10	1025	210.80	838	1026	0
11	1028	210.80	839	1029	0
12	1030	102.73	866	1031	0
13	1032	141.99	895	1033	1
14	1142	141.99	896	1143	1
15	1267	141.99	897	1270	1
16	1268	141.99	898	1270	1
17	1269	141.99	899	1270	1
18	1294	210.80	840	1293	0
19	1295	210.80	841	1293	0
20	1296	400.91	846	1293	0
21	1297	400.91	847	1293	0
22	1299	602.20	855	1298	0
23	1300	602.20	856	1298	0
24	1301	602.20	857	1298	0
25	1302	1000.99	878	1298	0
26	1303	1000.99	879	1298	0

OrderDetailsID 1299-1303

Funkcja sprawdza czy użytkownik może kupić gry:

```
public boolean userCanBuy(String user, BigDecimal money) {  
    PlatformUser platformUser = getUsersByName(user);  
    return platformUser.canBuy(money);  
}
```



Funkcja do kupienia kodów do gier z koszyka:

```
public void purchase(String username, Cart cart, BigDecimal totalSum) {
    final EntityManager manager = getManager();
    EntityTransaction etx = manager.getTransaction();
    try {
        etx.begin();

        HashMap<Game, Integer> gamesDetails = cart.getGamesCart();

        PlatformUser platformUser = getUsersByName(username);

        PlatformOrder platformOrder = new PlatformOrder(platformUser);
        manager.persist(platformOrder);

        for (Map.Entry<Game, Integer> entry : gamesDetails.entrySet()) {

            List<GameCode> gameCodes = getFreeCodes(entry.getKey());

            if (gameCodes.size() - entry.getValue() < 0) {
                throw new IllegalArgumentException("Too few game codes!");
            }

            Game game = entry.getKey();
            int quantity = entry.getValue();

            for (GameCode code : gameCodes.subList(0, quantity)) {
                OrderDetails orderDetails = new
OrderDetails(game.getPrice(), platformOrder, code);
                manager.persist(orderDetails);
                manager.merge(code);
            }
        }

        platformUser.updateMoney(totalSum.negate());

        manager.merge(platformUser);

        etx.commit();

        cart.clearCart();

    } catch (Exception exception) {
        if (etx.isActive()) {
            etx.rollback();
        }
    }
    finally {
        manager.close();
        setEntityManager();
    }
}
```

Funkcja pobiera użytkownika, tworzy nowe zamówienie, potem pobiera wolne kody dla danych gier i sprawdza czy jest ich wystarczająca ilość, następnie dodaje szczegóły zamówienia i aktualizuje dodawane kody. Na końcu zmienia stan konta użytkownika, aktualizuje użytkownika i używa commit, w razie wyjątku rollback.

## 4. Płatności użytkownika

W oknie płatności widać wszystkie zamówienia złożone przez użytkownika wraz z odpowiednią ilością kodów do gier oraz informacją, czy kod był użyty:

Zamówienie nr:	1298	Data:	2023-06-12 21:14:30....
Gra:	cod3	Cena:	602.20
Kod:	0LqNxRISZeG56Hv	Użyto:	Nie
Gra:	cod3	Cena:	602.20
Kod:	wEOF3zm2O4etKpJ	Użyto:	Nie
Gra:	cod3	Cena:	602.20
Kod:	eo4LoM6vIEHeDy	Użyto:	Nie
Gra:	cod5	Cena:	1000.99
Kod:	młcsXTGKiJ2HOzN	Użyto:	Nie
Gra:	cod5	Cena:	1000.99
Kod:	QQFT7ySAZISmmM4	Użyto:	Nie

Aby pobrać zamówienia użytkownika stosujemy funkcję:

```
public PlatformUser getUsersByName(String username){
    final EntityManager manager = getManager();

    Query query = manager.createQuery("SELECT u from PlatformUser u
    where u.username= :username");
    query.setParameter("username",username);
    List<PlatformUser> platformUsers = query.getResultList();

    PlatformUser user=platformUsers.size()==0 ? null :
    platformUsers.get(0);

    return user;
}
```

Funkcja pobiera użytkownika, po jego pobraniu za pomocą gettera można pobrać listę jego zamówień.

Okno zawiera także stan konta użytkownika, a także formularz do dodawania wpłat na konto użytkownika.

Po wypełnieniu informacji o wpłacie:

Wyloguj się

GAMENET

Koszyk

User

Sklep

Biblioteka

Płatności

Turnieje

Ustawienia

Zamówienie nr: 1298    Data: 2023-06-12 21:14:30....

Gra: cod3    Cena: 602.20

Kod: 0LqNxRISZeG56Hv    Użyto: Nie

Gra: cod3    Cena: 602.20

Kod: wEOF3zm2O4etKpJ    Użyto: Nie

Gra: cod3    Cena: 602.20

Kod: eoa4LoM6vLEHeDy    Użyto: Nie

Gra: cod5    Cena: 1000.99

Kod: mlcsXTGKIj2HOzN    Użyto: Nie

Gra: cod5    Cena: 1000.99

Kod: QQFT7ySAZISmmM4    Użyto: Nie

Wpłata

500.00

Nowa wpłata

2023-06-10 22:10:11

11-1111-1111-1111

Doładuj

☐ Zamówienia    ☒ Wpłaty

Stan konta

6191.42

Odśwież

I kliknięciu doładuj we wpłatach widać nowe doładowanie:

Wyloguj się

GAMENET

Koszyk

User

Sklep

Biblioteka

Płatności

Turnieje

Ustawienia

Numer wpłaty: 1304    Data: 2023-06-10 22:10:11.0

Tytuł wpłaty: Nowa wpłata

Kwota: 500.00

Zweryfikowano: false

Numer konta: 11-1111-1111-1111

Wpłata

Kwota

Tytuł wpłaty

yyyy-MM-dd hh:mm:ss

Numer konta: NN-NNNN-NNNN-NNNN-NNNN

Doładuj

☐ Zamówienia    ☒ Wpłaty

Stan konta

6191.42

Odśwież

W bazie danych dodano nowy rekord dla użytkownika o UserID 1132:

	PAYMENT_ID	AMOUNT	BANKACCOUNTNUMBER	DATE	ISVERIFIED	TITLE	USER_ID	VERSION
1	926	300.00	11-1111-1111-1111-1111	2023-06-08 22:22:22.000000000	true	Nowa wpłata	217	0
2	1304	500.00	11-1111-1111-1111-1111	2023-06-10 22:10:11.000000000	false	Nowa wpłata	1132	0

Aby wyświetlić wpłaty użyto podobnie jak z zamówieniami funkcji `getUserByName`, a następnie za pomocą gettera pobrano listę jego wpłat.

Aby dodać nową wpłatę użyto funkcji:

```
public void addPayment(String username, Payment payment) {
    final EntityManager manager = getManager();
    EntityTransaction etx = manager.getTransaction();
    try {
        etx.begin();

        PlatformUser platformUser = getUsersByName(username);

        payment.setPlatformUser(platformUser);
        manager.persist(payment);

        platformUser.addPaymentToUser(payment);
        manager.merge(platformUser);

        etx.commit();
    } catch (Exception exception) {
        if (etx.isActive()) {
            etx.rollback();
        }
    }
    finally {
        manager.close();
        setEntityManager();
    }
}
```

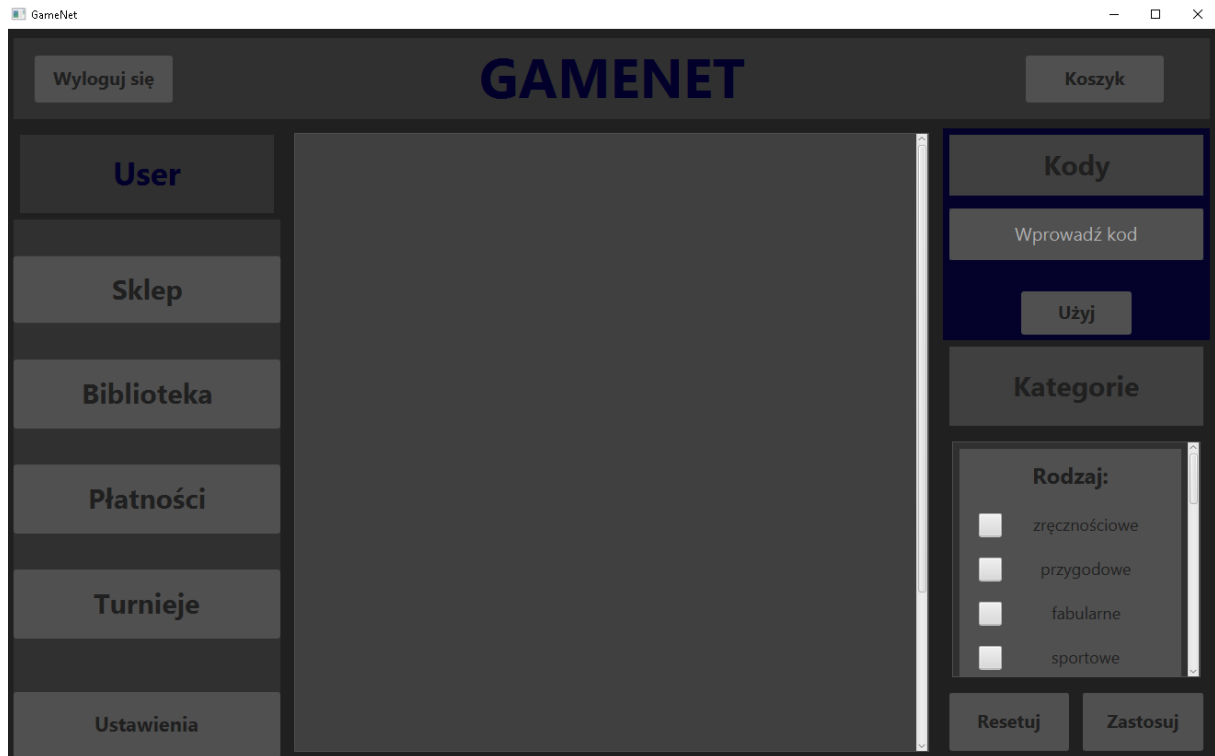
Funkcja pobiera użytkownika, ustawia płatności użytkownika i zapisuje ją, dodaje użytkownikowi płatność i aktualizuje użytkownika, potem commit lub w przypadku wyjątku rollback.

Po dodaniu płatność nie jest dodawana do konta do momentu jej weryfikacji przez pracownika.

## 5. Biblioteka użytkownika

Okno biblioteki daje dostęp do aktywowanych gier użytkownika. Użytkownik może kupić wiele kodów do tej danej gry, jednak dana gra może być aktywowana tylko raz dla danego konta za pomocą jednego kodu.

Widok biblioteki:



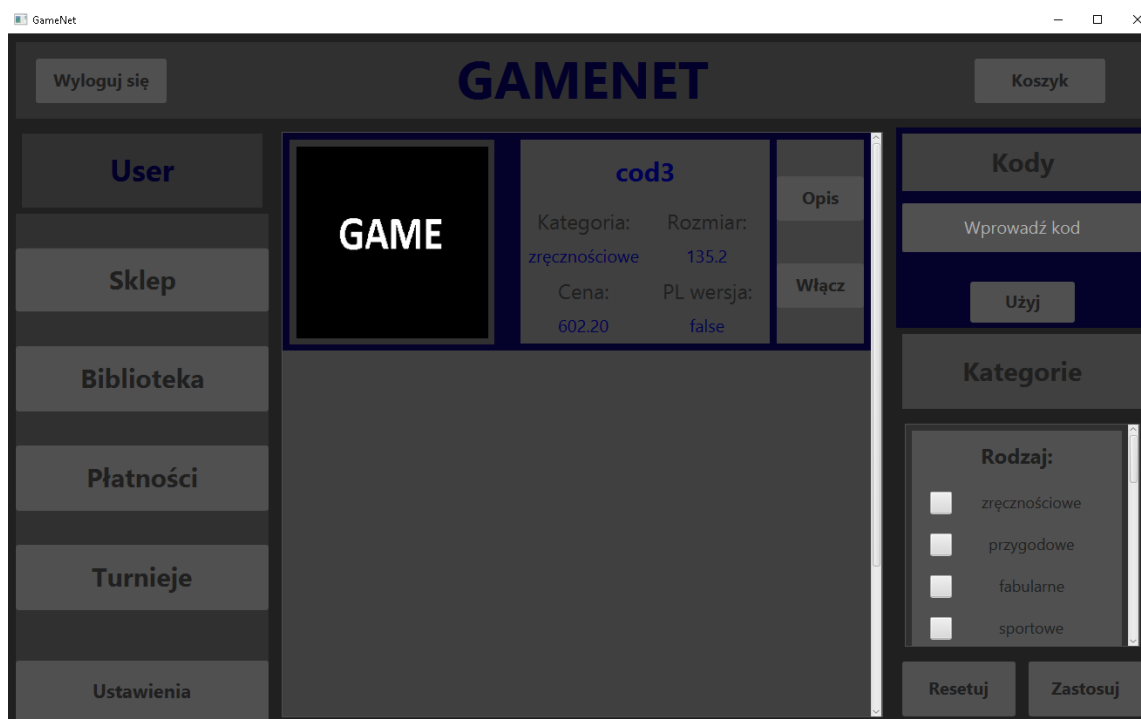
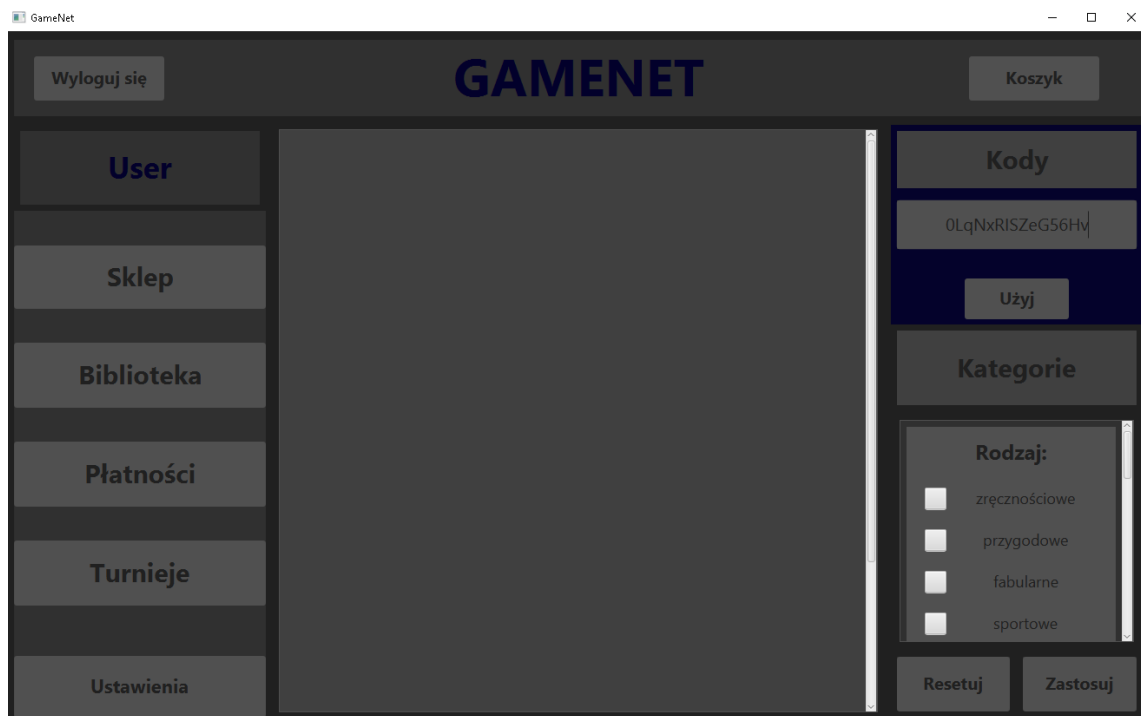
Funkcja do pobierania aktywnych gier dla danego użytkownika:

```
public List<Game> getUserGames(String username) {
    PlatformUser platformUser = getUsersByName(username);

    List<Game> games = new ArrayList<>();
    for(ActiveCode activeCode:platformUser.getActiveCodes()) {
        games.add(activeCode.getGameCode().getGame());
    }
    return games;
}
```

Pobiera użytkownika, bierze jego aktywne kody i na tej podstawie można stwierdzić, jakie gry ma użytkownik.

Użytkownik jest nowym użytkownikiem, więc nie ma aktywnych gier. Jednak kupił już kody. Po wybraniu kodu i wpisaniu go do okna na kody oraz kliknięciu użyj można aktywować kod:



Do bazy danych dodano nowe dane:

Tabela ActiveCode użytkownik o userID 1132:

	ACTIVECODE_ID	GAMECODE_ID	USER_ID	VERSION
1	911	825	217	0
2	913	863	217	0
3	932	836	1	0
4	936	837	425	0
5	1034	895	217	0
6	1305	855	1132	0

Tabela GameCode o gamecodeID 855 zostało zaktualizowane na true, czyli kod użyty:

	GAMECODE_ID	ACCESSCODE	USED	GAME_ID	VERSION
1	825	CKLDs3HDQmt6eQt	• true	725	0
2	834	eIev1rhp9U9H03Q	false	725	0
3	835	GmcHAPiZ49pRs7k	false	725	0
4	836	dkGv5w0N0s1tQEf	• true	725	0
5	837	mYuvzC0H6MsFAUL	• true	725	0
6	838	AgC2AiLzNZdaPc6	false	725	0
7	839	CZzYIGzjnhwe7co	false	725	0
8	840	3lindMU8KQp0F1p	false	725	0
9	841	NVylwyPWmRv1k6H	false	725	0
10	842	5sneOPUpBpZmCcW	false	725	0
11	843	KWZIo654xuVuHep	false	725	0
12	844	DTW10Dn86EZseej	false	725	0
13	845	xFeRBLQnA2FBDUx	false	725	0
14	846	wdCrcykaXmBBEM8	false	827	0
15	847	ZSiihxH99fB9ZnG	false	827	0
16	848	maWqmDqpYRkSxk9	false	827	0
17	849	HjhxZ0VMTXnxUui	false	827	0
18	850	R6oqYudEIrTDx2P	false	827	0
19	851	ycIvIFqaQk4FjQQ	false	827	0
20	852	9kSYgtDs7AEA1U0	false	827	0
21	853	V3Bk1CjtChFF5mD	false	827	0
22	854	2j0uV36xijdXZ09	false	828	0
23	855	0LqNxRISZeG56Hv	• true	828	1

Funkcja sprawdza czy kod może być użyty:

```
public boolean isCodeCanBeUsed(String code,String username){

    final EntityManager manager = getManager();

    Query query = manager.createQuery("SELECT cod from GameCode cod
where cod IN (SELECT od.gameCode FROM OrderDetails od) and
cod.accessCode=:code and cod not in (SELECT ac.gameCode FROM
ActiveCode ac)",GameCode.class);
    query.setParameter("code",code);
    List<GameCode> gameCodes = new
ArrayList<>(query.getResultList());

    if(gameCodes.size()==0){
        return false;
    }
    GameCode gameCode = gameCodes.get(0);
    return !userOwnGame(username,gameCode.getGame());
}
```

Kod może być użyty gdy został kupiony (nie musi być kupiony przez aktywującego go użytkownika) czyli jest w OrderDetails, oraz nie został aktywowany, czyli nie jest w ActiveCode. Sprawdzane jest też czy użytkownik nie posiada już gry, której kod aktywuje.

Funkcja dodaje kod do użytkownika:

```
public void addCodeToUser(String username,String code){
    final EntityManager manager = getManager();
    EntityTransaction etx = manager.getTransaction();
    try {
        etx.begin();

        PlatformUser user = getUsersByName(username);
        GameCode gameCode = getGameCode(code);

        ActiveCode activeCode = new ActiveCode(user,gameCode);

        manager.persist(activeCode);

        manager.merge(gameCode);
        manager.merge(user);

        etx.commit();

    }catch (Exception exception){
        if(etx.isActive()){
            etx.rollback();
        }
    }
    finally {
        manager.close();
        setEntityManager();
    }
}
```

Pobierany jest użytkownik i kod, a także tworzony jest obiekt klasy ActiveCode, służy on powiązaniu użytkownika i kodu oraz wskazuje, że kod został użyty przez użytkownika, następnie nowy obiekt jest zapisywany, a pobrane obiekty są aktualizowane. Na końcu commit lub w razie wyjątku rollback.

Użytkownik ma przypisaną grę do konta (kod do gry został aktywowany). Należy zaznaczyć, że aby aktywować kod użytkownik aktywujący kod nie musiał go kupić, kod może mu zostać podarowany przez innego użytkownika. Jedynymi warunkami aktywacji jest to że kod został kupiony, nie został aktywowany oraz użytkownik aktywujący kod nie ma gry którą kod aktywuje. Można też filtrować posiadane gry.



## 6. Turnieje

Uprawnienia użytkowników zależą od tego czy użytkownik to indywidualny użytkownik czy firma.

Tylko użytkownicy firmowi mogą dodawać turniej w oknie stwórz:

The screenshot shows the 'GAMENET' application interface. The left sidebar contains navigation links: Wyloguj się, Firma, Sklep, Biblioteka, Płatności, Turnieje, and Ustawienia. The main content area is titled 'Tworzenie nowego turnieju'. It contains the following fields and buttons:

- Nazwa:** CUP
- Gra:** cod3
- Od:** 2023-06-15 19:00:00
- Do:** 2023-06-15 22:00:00
- Limit graczy:** 300
- Opis:** Nowy turniej
- Buttons:** Stwórz, [Empty button]

The right sidebar contains buttons: Koszyk, Dostępne, Informacje, Stwórz, Stworzone, and Odśwież.

W oknie stworzone pojawia się turniej, te okno wyświetla turnieje stworzone przez daną firmę:

The screenshot shows the 'GAMENET' application interface. The left sidebar is the same as in the previous image. The main content area is titled 'Stworzone'. It displays the details of a tournament created by the user:

- Turniej nr:** 1311
- Nazwa:** CUP
- Od:** 2023-06-15 19:00:00.0
- Do:** 2023-06-15 22:00:00.0
- Stworzony przez:** Firma
- Gra:** cod3
- Limit graczy:** 300
- Zapisanych:** 0
- Opis:** Nowy turniej
- Buttons:** Lista

The right sidebar contains buttons: Koszyk, Dostępne, Informacje, Stwórz, Stworzone, and Odśwież.

Dane zostały dodane do bazy danych:

Tabela Tournament (TournamentID 1311):

	TOURNAMENT_ID	DESCRIPTION	ENDTOURNAMENT	NAME	STARTTOURNAMENT	USERLIMIT	COMPANY_ID	GAME_ID	VERSION
1	914	Other	2023-09-11 22:00:00.000000000	Cup	2023-09-11 19:00:00.000000000	10	217	725	0
2	1027	Other	2023-08-19 22:00:00.000000000	Nowy	2023-08-19 20:00:00.000000000	100	217	827	0
3	1311	Nowy turniej	2023-06-15 22:00:00.000000000	CUP	2023-06-15 19:00:00.000000000	300	1306	828	0

Funkcja sprawdzająca czy gra w którą turniej ma być rozgrywany istnieje:

```
public Game getGameByName(String name){
    final EntityManager manager = getManager();

    Query query = manager.createQuery("SELECT g from Game g where
g.gameName=:game");
    query.setParameter("game",name);
    List<Game> games = new ArrayList<>(query.getResultList());

    return games.size()>0? games.get(0):null;
}
```

Funkcja dodająca turniej do bazy danych:

```
public void addTournament(Tournament tournament){
    final EntityManager manager = getManager();
    EntityTransaction etx = manager.getTransaction();
    try {
        etx.begin();

        manager.persist(tournament);

        etx.commit();

    }catch (Exception exception){
        if(etx.isActive()){
            etx.rollback();
        }
    }
    finally {
        manager.close();
        setEntityManager();
    }
}
```

Dodawany i commitowany jest turniej. W razie wyjątku rollback.

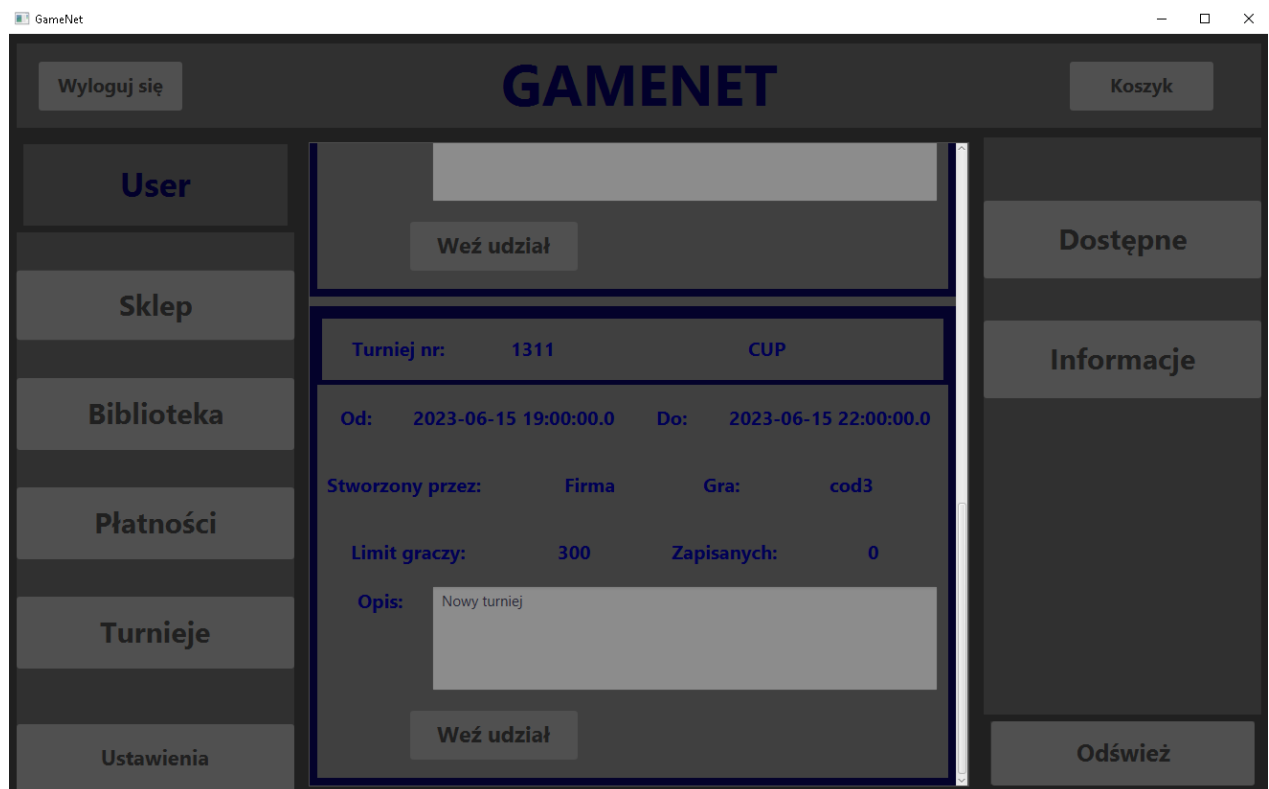
Funkcja wyświetlająca turnieje stworzone przez firmę:

```
public List<Tournament> getTournamentsCreatedByUser(int companyID) {  
    final EntityManager manager = getManager();  
  
    TypedQuery<Tournament> query = manager.createQuery(  
        "SELECT t FROM Tournament t WHERE  
t.companyUser.User_ID=:id", Tournament.class);  
    query.setParameter("id", companyID);  
    List<Tournament> tournaments = query.getResultList();  
  
    return tournaments;  
}
```

Firma nie może brać udziału w turniejach, które stworzyła.

Zarówno firma jak i użytkownik indywidualny ma okna Dostępne i Informacje.

Użytkownik po kliknięciu Weź udział (pod warunkiem że ma aktywowaną grę w którą turniej jest rozgrywany) jest dopisany do turnieju:



Funkcja wyświetlająca turnieje dostępne:

```
public List<Tournament> getAvailableTournaments(int userID) {  
    final EntityManager manager = getManager();  
  
    TypedQuery<Tournament> query = manager.createQuery(  
        "SELECT t FROM Tournament t WHERE  
t.startTournament>current_timestamp and  
t.userTournaments.size<t.userLimit and t NOT IN (SELECT  
ut.tournament FROM UserTournament ut WHERE ut.user.User_ID = :user)  
and not t.companyUser.User_ID=:user", Tournament.class);  
    query.setParameter("user", userID);  
    List<Tournament> tournaments = query.getResultList();  
  
    return tournaments;  
}
```

Wyszukiwane są turnieje których czas rozpoczęcia jest większy niż aktualny czas, maksymalny limit graczy jest większy niż liczba zapisanych graczy, użytkownik nie jest zapisany już na turniej, a także nie stworzył go.

Po kliknięciu przycisku dane są dodawane do bazy danych:

Tabela łącznikowa UserTournament (UserTournamentID 1312):

	USERTOURNAMENT_ID	ACCESSCODE	TOURNAMENT_ID	USER_ID	VERSION
1	933	RWDfwIz2E754V4s	914	1	0
2	937	QNwjWE1W5cPiwIN	914	425	0
3	1312	KrkqCQPfjQmcSSJ	1311	1132	0

Użyto funkcji:

```
public Tournament getTournamentByName(String name){  
    final EntityManager manager = getManager();  
  
    Query query = manager.createQuery("SELECT t from Tournament t  
where t.name= :name");  
    query.setParameter("name", name);  
    List<Tournament> tournaments = query.getResultList();  
  
    Tournament tournament=tournaments.size()==0 ? null :  
tournaments.get(0);  
  
    return tournament;  
}
```

Funkcja dodająca użytkownika do turnieju:

```
public void addUserToTournament(String username, String
tournamentName, String code) {
    final EntityManager manager = getManager();
    EntityTransaction etx = manager.getTransaction();
    try {
        etx.begin();

        PlatformUser platformUser = getUsersByName(username);
        Tournament tournament = getTournamentByName(tournamentName);

        UserTournament userTournament = new
UserTournament(tournament, platformUser, code);
        manager.persist(userTournament);

        platformUser.addUserTournament(userTournament);
        tournament.addUserTournament(userTournament);

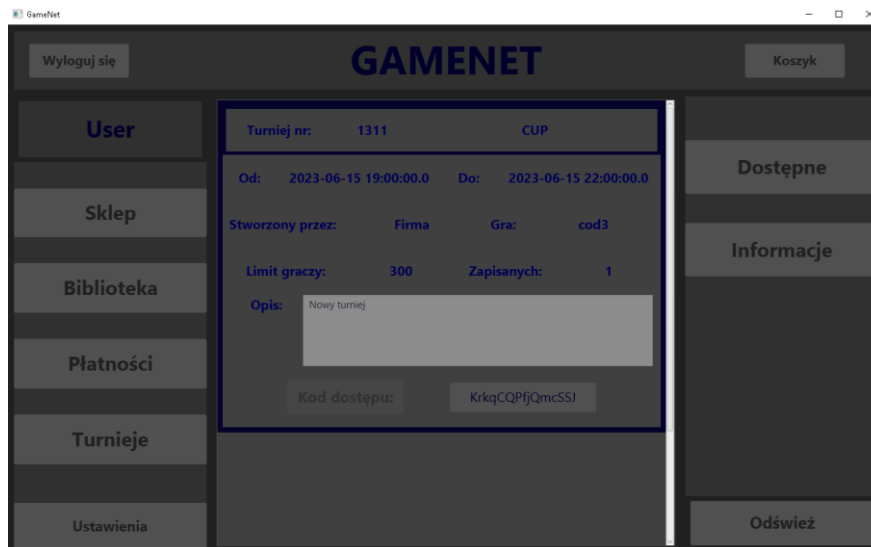
        manager.merge(platformUser);
        manager.merge(tournament);

        etx.commit();

    } catch (Exception exception) {
        if (etx.isActive()) {
            etx.rollback();
        }
    }
    finally {
        manager.close();
        setEntityManager();
    }
}
```

Pobierany jest użytkownik, turniej oraz tworzony i dodawany jest nowy obiekt klasy UserTournament czyli tabeli łącznikowej, aktualizowane są także obiekty użytkownika oraz turnieju. Na końcu commit, w razie wyjątku rollback.

Gdy użytkownik weźmie udział w turnieju, to w oknie Informacje pojawi się turniej. W tym oknie są turnieje w których użytkownik bierze udział:



Używana jest funkcja do wyświetlania turniejów, w których użytkownik bierze udział:

```
public List<Tournament> getUserTournaments(int userID) {

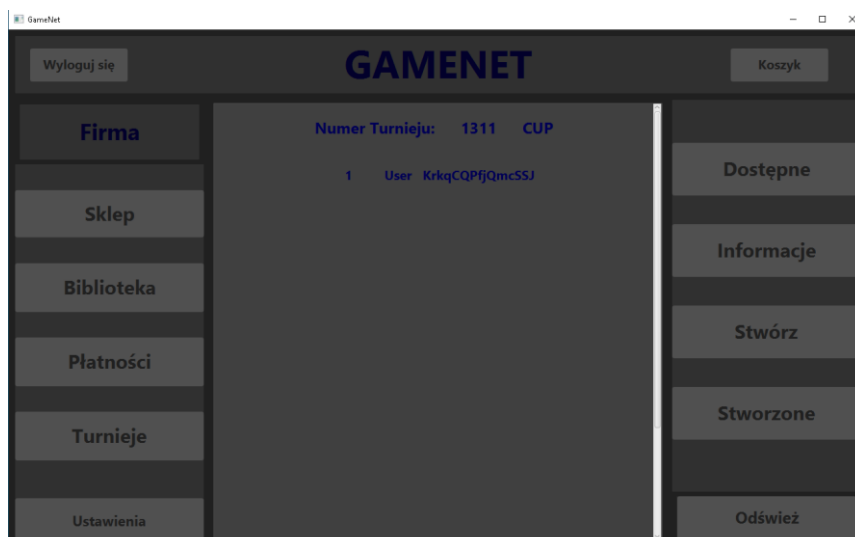
    final EntityManager manager = getManager();

    TypedQuery<Tournament> query = manager.createQuery(
        "SELECT t FROM Tournament t WHERE t IN (SELECT ut.tournament FROM UserTournament ut WHERE ut.user.User_ID = :user)", Tournament.class);
    query.setParameter("user", userID);
    List<Tournament> tournaments = query.getResultList();

    return tournaments;
}
```

Brane są turnieje które są w zbiorze obiektów UserTournament, w których występuje użytkownik.

Firma organizująca turniej po wejściu w okno Stworzone oraz kliknięciu Lista dla danego turnieju widzi listę użytkowników zapisanych na turniej wraz z ich kodem dostępowym:



Aby wyświetlić listę użytkowników korzysta się z funkcji:

```
public List<PlatformUser> getTournamentsAllUsers(int tournamentID){
    final EntityManager manager = getManager();

    TypedQuery<PlatformUser> query = manager.createQuery(
        "SELECT user FROM PlatformUser user where user IN
(SELECT ut.user FROM UserTournament ut WHERE
ut.tournament.Tournament_ID=:id)", PlatformUser.class);
    query.setParameter("id", tournamentID);
    List<PlatformUser> users = query.getResultList();

    return users;
}
```

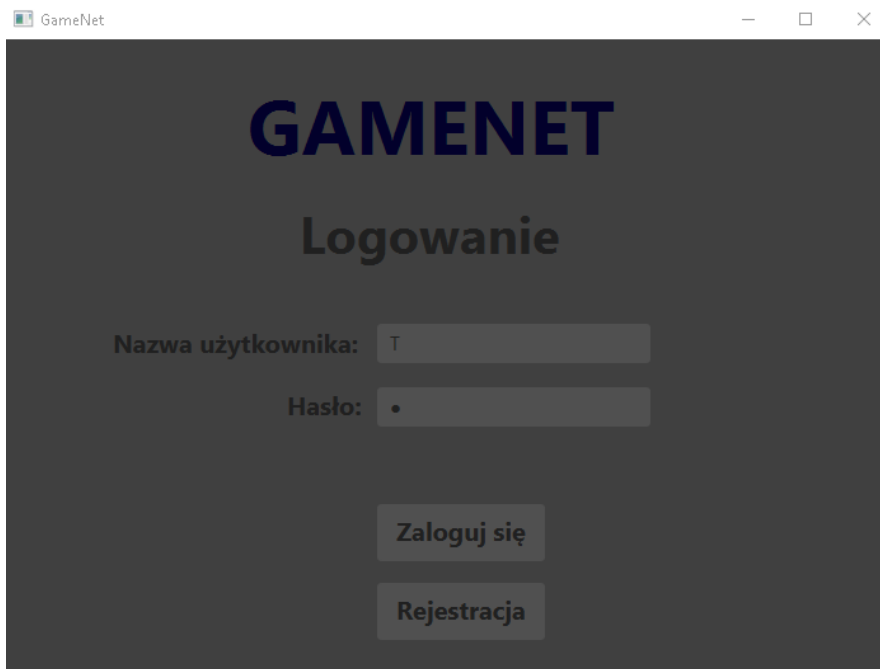
Pobiera wszystkich użytkowników, którzy wstępują w obiektach UserTournament dla danego turnieju.

## 7. Pracownik i jego uprawnienia

W bazie danych jest pracownik:

	EMPLOYEE_ID	EMAIL	FIRSTNAME	LASTNAME	PASSWORD	USERNAME	VERSION
1	20	newemployee@example.com	T	B	a	T	0

Logowanie:



Aby zalogować pracownika stosuje się funkcje:

```
public boolean employeeLoginAuthorization(String username,String
password){
    Employee employee = getEmployee(username);
    if(employee!=null){
        return password.equals(employee.getPassword());
    }
    return false;
}
```

Która używa funkcji pobierającej pracownika z bazy danych:

```
public Employee getEmployee(String username){
    final EntityManager manager = getManager();

    Query query = manager.createQuery("SELECT e from Employee e
where e.username= :username");
    query.setParameter("username",username);
    List<Employee> employees = query.getResultList();

    Employee user=employees.size()==0 ? null : employees.get(0);

    return user;
}
```



Po zalogowaniu może dodawać gry, kody oraz kategorie:

GameNet

Wyloguj się

GAMENET

T

Tworzenie nowej gry

Nazwa:

Zdjęcie:

Polska wersja:

tak lub nie

Cena:

Rozmiar:

GB

Nazwa kategoria:

Stwórz

Dodawanie kodu do gry

Nazwa gry:

Kod:

Stwórz

Dodawanie kategorii:

Nazwa:

Opis:

Stwórz

☐ Dodawanie

☐ Wpłaty

Ustawienia

Odśwież

Dodawanie gry:

GameNet

Wyloguj się

GAMENET

T

Tworzenie nowej gry

Nazwa:

ModernGame

Zdjęcie:

cod.png

Polska wersja:

tak

Cena:

350.99

Rozmiar:

127.80

Nazwa kategoria:

zręcznościowe

Stwórz

Dodawanie kodu do gry

Nazwa gry:

Kod:

Stwórz

☐ Dodawanie

☐ Wpłaty

Ustawienia

Odśwież

Dodano w bazie danych dane w tabeli Game (GameID 1313):

	GAME_ID	GAMENAME	IMAGE	ISPOLISH	PRICE	SIZEGB	CATEGORY_ID	VERSION
1	725	cod1	cod.png	true	210.80	300.7	11	0
2	827	cod2	cod.png	true	400.91	1205.2	826	0
3	828	cod3	cod.png	false	602.20	135.2	8	0
4	829	cod4	cod.png	true	102.73	400	13	0
5	830	cod5	cod.png	false	1000.99	271.3	12	0
6	831	cod10	cod.png	true	2501.98	900.9	11	0
7	832	cod11	cod.png	false	3211.11	240.91	9	0
8	833	cod15	cod.png	true	141.99	88.78	8	0
9	1313	ModernGame	cod.png	true	350.99	127.8	8	0

Użyto funkcji zwracającej kategorie:

```
public Category getCategory(String name) {
    final EntityManager manager = getManager();

    Query query = manager.createQuery("SELECT c from Category c
where c.categoryName= :name");
    query.setParameter("name", name);
    List<Category> categories = query.getResultList();

    Category category = categories.size()==0 ? null :
categories.get(0);

    return category;
}
```

Funkcja dodająca do bazy danych grę:

```
public void addGame(Game game) {
    final EntityManager manager = getManager();
    EntityTransaction etx = manager.getTransaction();
    try {
        etx.begin();

        manager.persist(game);

        etx.commit();

    } catch (Exception exception) {
        if(etx.isActive()) {
            etx.rollback();
        }
    }
    finally {
        manager.close();
        setEntityManager();
    }
}
```

Dodawanie gry, commit lub w razie wyjątku rollback.

Dodawanie kodu do gry:

Wyloguj się

# GAMENET

T

Rozmiar: GB

Nazwa kategoria:

Stwórz

Dodawanie kodu do gry

Nazwa gry: ModernGame

Kod: hAC2AiLjIZdaPc6

Stwórz

Dodawanie kategorii:

Nazwa:

Opis:

☐ Dodawanie ☐ Wpłaty

Odśwież

Ustawienia

Dodano w bazie danych dane w tabeli GameCode (GameCodeID 1314):

	GAMECODE_ID	ACCESSCODE	USED	GAME_ID	VERSION
174	1248	ZR08HY60x69dZ7P	false	831	0
175	1249	vqvX04pyh6i6Mac	false	831	0
176	1250	pCJjwNPiijDJ25s	false	831	0
177	1251	VrPeG9XLJMLpakU	false	831	0
178	1252	XTcYuHLbtT3buSY	false	832	0
179	1253	l9HY3kbuDcz9n4q	false	832	0
180	1254	mVyWhcDk6JUYhK8	false	832	0
181	1255	wGct6r9yEIu0UxL	false	832	0
182	1256	10MVnbv775IxCdD	false	832	0
183	1257	qJLk5Jfi6BxxUGB	false	833	0
184	1258	RIAp0Rf0IhNtd8f	false	833	0
185	1259	sCscMakVQ7HMODs	false	833	0
186	1260	S6RmhqoJPGp0Spw	false	833	0
187	1261	ZgZoTUHGULxsc1a	false	833	0
188	1262	c7Blp0BbTs3ntrr	false	833	0
189	1263	BWCo98Fdocn6ftb	false	833	0
190	1264	xBed68GIjiCd10M	false	833	0
191	1265	SmtvsXCKwqoXvgg	false	833	0
192	1266	yzPZx20WRb7SMML	false	833	0
193	1314	hAC2AiLjLZdaPc6	false	1313	0

Użyto funkcji która zwraca kod jeśli istnieje:

```

public GameCode getGameCode(String code){
    final EntityManager manager = getManager();

    Query query = manager.createQuery("SELECT cod from GameCode cod
where cod.accessCode=:code", GameCode.class);
    query.setParameter("code", code);
    List<GameCode> gameCodes = new
ArrayList<>(query.getResultList());

    return gameCodes.size()>0 ? gameCodes.get(0) : null;
}

```

Funkcja dodająca do bazy danych kod do gry:

```

public void addGameCode(GameCode gameCode){
    final EntityManager manager = getManager();
    EntityTransaction etx = manager.getTransaction();
    try {
        etx.begin();

        manager.persist(gameCode);

        etx.commit();

    }catch (Exception exception){
        if(etx.isActive()){
            etx.rollback();
        }
    }
    finally {
        manager.close();
        setEntityManager();
    }
}

```

Dodawanie kodu, commit lub w razie wyjątku rollback.

Dodawanie kategorii:

Dodano w bazie danych dane w tabeli Category (CategoryID 1316):

	CATEGORY_ID	CATEGORYNAME	DESCRIPTION	VERSION
1	8	zręcznościowe	other	0
2	9	przygodowe	other	0
3	10	fabularne	other	0
4	11	sportowe	other	0
5	12	strategiczne	other	0
6	13	symulacyjne	other	0
7	826	akcji	other	0
8	1316	wyścigowe	other	0

Użyto funkcji zwracającej kategorię jeśli istnieje:

```
public Category getCategory(String name) {
    final EntityManager manager = getManager();

    Query query = manager.createQuery("SELECT c from Category c
where c.categoryName= :name");
    query.setParameter("name", name);
    List<Category> categories = query.getResultList();

    Category category = categories.size()==0 ? null :
categories.get(0);

    return category;
}
```

Użyto funkcji dodającej do bazy danych kategorię:

```
public void addCategory(Category category) {
    final EntityManager manager = getManager();
    EntityTransaction etx = manager.getTransaction();
    try {
        etx.begin();

        manager.persist(category);

        etx.commit();

    } catch (Exception exception) {
        if(etx.isActive()) {
            etx.rollback();
        }
    }
    finally {
        manager.close();
        setEntityManager();
    }
}
```

Dodawanie kategorii, commit lub w razie wyjątku rollback.

W oknie wpłaty widać doładowania do zatwierdzenia:

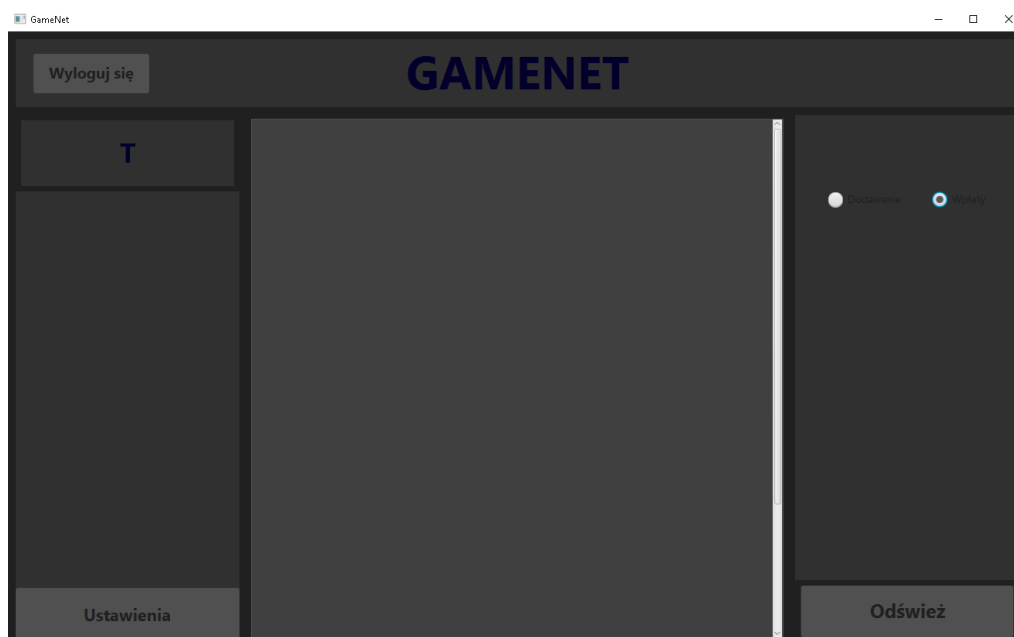


Wyświetla je funkcja:

```
public List<Payment> getAllNotVerifiedPayment() {  
    final EntityManager manager = getManager();  
  
    Query query = manager.createQuery("SELECT p FROM Payment p where  
p.isVerified=false", Payment.class);  
    List<Payment> payments = query.getResultList();  
  
    return payments.size()>0 ? payments : null;  
}
```

Funkcja bierze wszystkie nie zweryfikowane wpłaty.

Po kliknięciu Potwierdź z listy znika wpłata:



W bazie danych zmieniana jest wartość pola isVerified na true w tabeli Payment:

	PAYMENT_ID	AMOUNT	BANKACCOUNTNUMBER	DATE	ISVERIFIED	TITLE	USER_ID	VERSION
1	926	300.00	11-1111-1111-1111	2023-06-08 22:22:22.000000000	• true	Nowa wpłata	217	0
2	1304	500.00	11-1111-1111-1111	2023-06-10 22:10:11.000000000	• true	Nowa wpłata	1132	1

A także dodawane są środki do konta użytkownika o id 1132 (tabela PlatformUser).

Przed wpłatą było:



Po wpłacie:

	USER_ID	EMAIL	MONEY	PASSWORD	USERNAME	VERSION
1	1	a@o2.pl	9071.20	a	Nowy	0
2	217	game1111@o2.pl	5953.11	b	GameDev	5
3	425	a@o2.pl	9789.20	a	AM	0
4	525	new@o2.pl	9699.20	a	Nowy2	0
5	1132	tn@example.com	6691.42	abc	User	4
6	1306	newcompany@example.com	8795.60	abc	Firma	2

Funkcja obsługuje potwierdzenie płatności:

```
public void verifiedPayment(int paymentID) {
    final EntityManager manager = getManager();
    EntityTransaction etx = manager.getTransaction();
    try {
        etx.begin();

        Query query = manager.createQuery("SELECT p FROM Payment p
where p.payment_ID=:id", Payment.class);
        query.setParameter("id", paymentID);
        List<Payment> payments = query.getResultList();

        if(payments.size()==0 ||
payments.get(0).isVerified()==true){
            throw new Exception();
        }

        Payment payment = payments.get(0);

        payment.setVerified(true);

        manager.merge(payment);

        PlatformUser platformUser =
getUsersByName(payment.getPlatformUser().getUsername());

        platformUser.updateMoney(payment.getAmount());

        manager.merge(platformUser);

        etx.commit();

    } catch (Exception exception) {
        if(etx.isActive()){
            etx.rollback();
        }
    }
    finally {
        manager.close();
        setEntityManager();
    }
}
```

Pobiera płatność, aktualizuje pole isVerified oraz płatność, a także pobiera użytkownika, dodaje mu fundusze do konta oraz aktualizuje użytkownika. Potem commit, w razie wyjątku rollback.



# Uruchomienie aplikacji

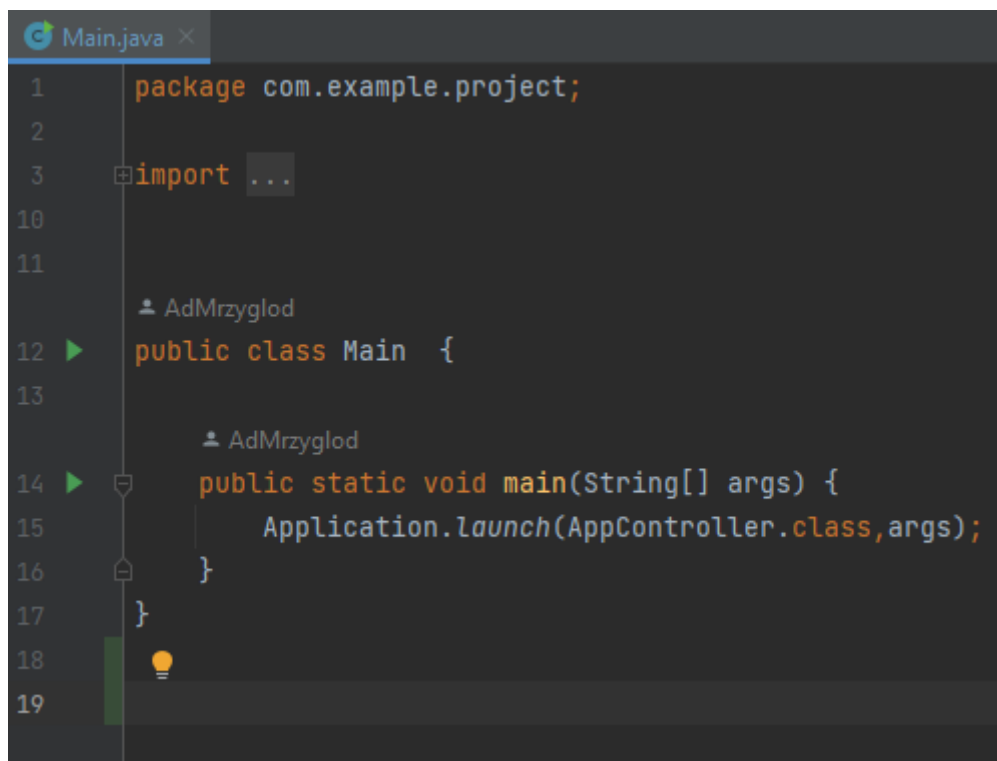
Aby uruchomić aplikację należy:

-Włączyć server apache derby:

```
PS C:\Apache\db-derby-10.14.2.0-bin\bin> ./startNetworkServer
Mon Jun 12 19:22:14 CEST 2023 : Security manager installed using the Basic server security policy.
Mon Jun 12 19:22:14 CEST 2023 : Serwer sieciowy Apache Derby - 10.14.2.0 - (1828579) uruchomiony i gotowy do zaakceptowa
nia po|ł|cze" na porcie 1527 w {3}
```

Server zapewnia połączenie i dostęp do bazy danych.

-Uruchomić funkcję main klasy Main zielonym przyciskiem:



```
1 package com.example.project;
2
3 import ...
4
5
6
7
8
9
10
11
12 public class Main {
13
14     public static void main(String[] args) {
15         Application.launch(AppController.class, args);
16     }
17 }
18
19
```

Uruchamiana jest aplikacja.