Міністерство освіти і науки України
Національний технічний університет України
“Київський політехнічний інститут імені Ігоря Сікорського”
Факультет інформатики та обчислювальної техніки
Кафедра інформаційних систем та технологій

Лабораторна робота №2
Технології розроблення програмного забезпечення
ДІАГРАМА ВАРІАНТІВ ВИКОРИСТАННЯ. СЦЕНАРІЇ ВАРІАНТІВ ВИКОРИСТАННЯ. ДІАГРАМИ UML. ДІАГРАМИ КЛАСІВ. КОНЦЕПТУАЛЬНА МОДЕЛЬ СИСТЕМИ
Особиста Бухгалтерія

Виконала:
Студентка групи ІА-22
Лахоцька С. О.

Перевірив:
Мягкий М. Ю

Київ 2024

Зміст

Тема: Діаграма варіантів використання. Сценарії варіантів використання. Діаграми UML. Діаграми класів. Концептуальна модель системи
Мета: Проаналізувати тему, намалювати схему прецеденту, діаграму класів, розробити основні класи і структуру бази

Теоретичні відомості

У другій лабораторній роботі розглядаються основи створення UML-діаграм та проектування системи, зокрема, її класової структури, сценаріїв використання та зв'язку з базою даних. Основні теоретичні відомості:

UML (Unified Modeling Language)

UML – це універсальна мова візуального моделювання, яка використовується для специфікації, візуалізації та документування різних аспектів програмного забезпечення. UML допомагає створювати моделі, які показують структурні та поведінкові характеристики системи.

Діаграма варіантів використання (Use Case Diagram)

Це графічне представлення того, як користувачі (актори) взаємодіють із системою через різні сценарії або варіанти використання. Діаграма дозволяє визначити межі системи, сформувати загальні вимоги до функціональності та розробити концептуальну модель на початковому етапі проектування.

Елементи діаграми:

- Актори (Actors): суб'єкти, які взаємодіють із системою. Це можуть бути користувачі, інші системи або пристрої.
- Варіанти використання (Use Cases): функції, які система виконує для акторів. Кожен варіант представляє певний сценарій взаємодії.
- Відносини (Relationships): асоціації між акторами та варіантами використання, що вказують на їх взаємодію.

Діаграма варіантів використання є початковою точкою для збору вимог і допомагає зрозуміти функціональні вимоги до системи.

Сценарії використання (Use Case Scenarios)

Сценарії використання уточнюють діаграму варіантів використання, надаючи текстовий опис конкретних процесів. Вони містять:

- Передумови (Preconditions): умови, які мають бути виконані перед початком сценарію.
- Постумови (Postconditions): результати, які досягаються після успішного виконання сценарію.
- Основний потік подій (Main flow): основні дії, які відбуваються в рамках сценарію.
- Виключення (Exceptions): альтернативні потоки або помилки, які можуть виникнути.

Діаграма класів (Class Diagram)

Діаграма класів описує структуру системи з точки зору класів, їх атрибутів і методів. Вона показує статичні аспекти системи, такі як:

- Класи: основні будівельні блоки, що описують об'єкти системи. Кожен клас має назву, атрибути (дані) і методи (функції).
- Атрибути: змінні, які зберігають дані об'єктів.
- Методи: функції, що визначають поведінку об'єктів.
- Відносини між класами: можуть бути різних типів – асоціація, агрегація, композиція, наслідування.

Діаграма класів часто використовується для моделювання структури бази даних та її об'єктів.

Проектування бази даних

Проектування бази даних є важливим аспектом розробки системи, оскільки воно забезпечує організацію та зберігання даних. Існує дві моделі бази даних.

Логічна модель бази даних: описує структуру таблиць, зв'язки між ними та індекси. Вона визначає, як дані зберігаються та організовуються на логічному рівні.

Фізична модель бази даних: описує, як дані зберігаються фізично на диску, включаючи структури файлів і способи доступу до них.

Проектування логічної моделі бази даних включає нормалізацію. Нормалізація – це процес приведення структури бази даних до нормальних форм для мінімізації дублювання даних та забезпечення їхньої цілісності. Основні нормальні форми:

Перша нормальна форма (1НФ): кожен атрибут таблиці містить лише одне значення.

Друга нормальна форма (2НФ): всі неключові атрибути повинні залежати від первинного ключа.

Третя нормальна форма (3НФ): не повинно бути транзитивних залежностей між неключовими атрибутами.

Хід роботи

Схема прецедентів

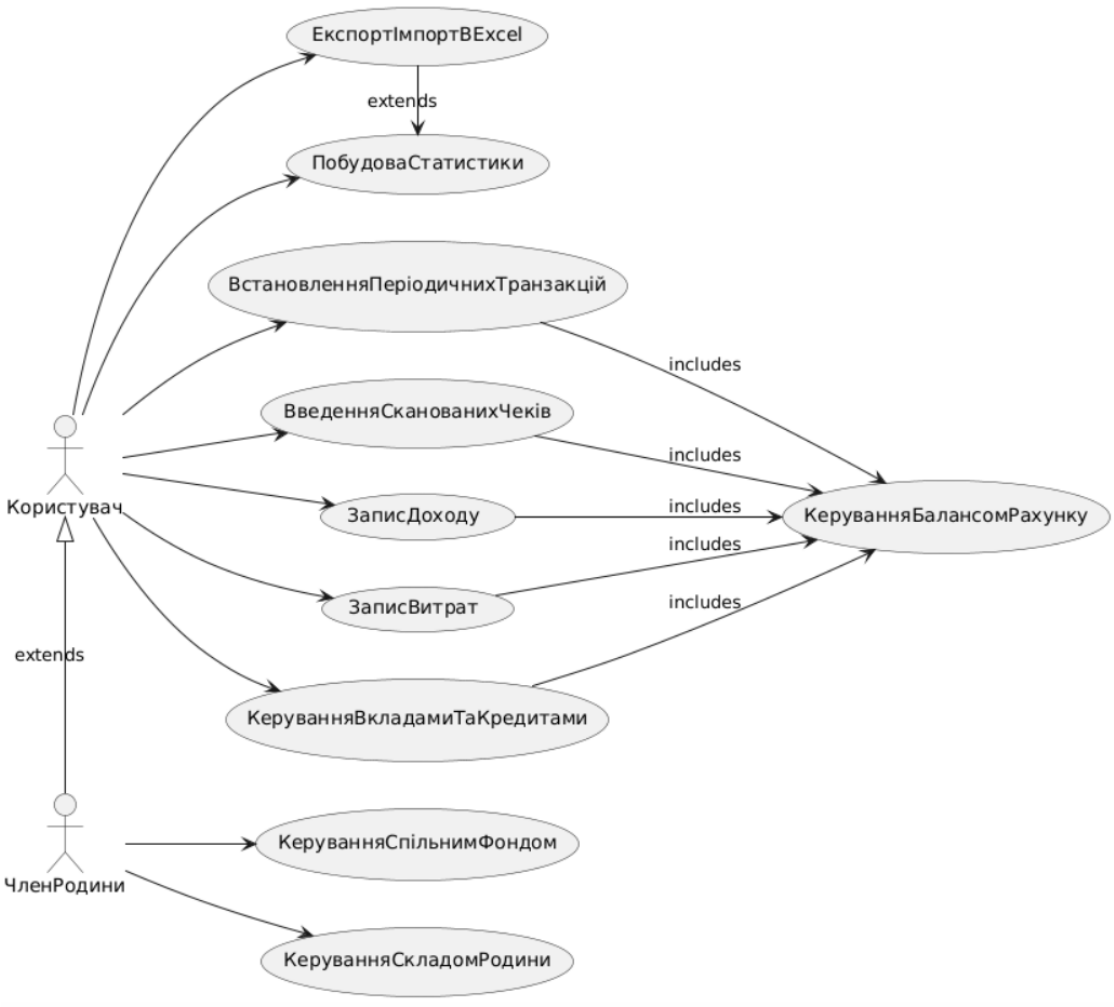
Схема прецедентів, що відповідає обраній темі «Особиста Бухгалтерія».

Ця система являє собою програму для управління фінансами, зокрема особистими рахунками, родинним бюджетом та спільними фінансовими фондами. У її центрі знаходиться користувач, який може виконувати різноманітні фінансові операції, починаючи від введення періодичних

транзакцій і запису доходів та витрат, до управління вкладами та кредитами. Усі ці дії взаємопов'язані з керуванням балансом рахунку, що є основним компонентом фінансової діяльності користувача.

Система також підтримує можливість введення сканованих чеків для зручного відстеження витрат, і ця функція безпосередньо взаємодіє з балансом рахунку. Користувач може здійснювати експорт та імпорт даних між програмою та Excel, що дозволяє йому використовувати зовнішні інструменти для додаткової аналітики. Побудова статистики в програмі допомагає користувачу аналізувати свої фінансові операції та оцінювати стан своїх рахунків.

Для сімейних користувачів система має розширений функціонал, який дозволяє членам родини управляти спільними фондами та складом родини. Учасники родини мають доступ до цих функцій, розширюючи свої можливості у веденні родинного бюджету. Це надає змогу підтримувати прозорість і співпрацю у фінансових питаннях.



На основі діаграми опишемо 3 сценарії використання:

Сценарій 1: Введення сканованих чеків

Передумови: Сканований чек доступний користувачу.

Постумови: Витрати з чеку були успішно додані до системи, відповідно категоризовані.

Взаємодіючі сторони: Користувач, Система.

Короткий опис: Даний сценарій описує процес додавання витрат шляхом завантаження сканованого чеку.

Основний потік подій:

1. Користувач ініціює функцію введення сканованого чеку.
2. Система запитує файл сканованого чеку.
3. Користувач завантажує сканований файл у систему.
4. Система проводить розпізнавання тексту на чеку за допомогою технології OCR.
5. Система розподіляє витрати на відповідні категорії.
6. Система зберігає витрати в базі даних, користувач може переглядати та редагувати їх, якщо необхідно.

Виключення:

Помилка розпізнавання: Якщо текст на сканованому чеку не може бути розпізнаний, система повідомляє користувача та запрошує повторне завантаження або ручне введення.

Примітки: Відсутні.

Сценарій 2: Встановлення періодичних транзакцій

Передумови: Користувач має активний обліковий запис та підключені рахунки.

Постумови: Періодична транзакція була успішно створена, і надалі буде виконуватись автоматично.

Взаємодіючі сторони: Користувач, Система.

Короткий опис: Даний сценарій описує процес налаштування періодичних транзакцій, таких як орендна плата або зарплата.

Основний потік подій:

1. Користувач відкриває розділ періодичних транзакцій.

2. Система запитує дані про нову періодичну транзакцію (сума, тип, періодичність).
3. Користувач вводить дані для транзакції.
4. Система зберігає налаштування, періодична транзакція додається до розкладу платежів.

Виключення:
Неправильний формат даних: Якщо введено некоректні дані (наприклад, недопустима сума), система повідомляє про помилку і просить виправити дані.

Примітки: Відсутні.

Сценарій 3: Експорт/Імпорт даних в/з Excel

Передумови: Користувач має збережені фінансові дані та встановлене програмне забезпечення для роботи з Excel.

Постумови: Дані були успішно експортовані у файл Excel або імпортовані в систему.

Взаємодіючі сторони: Користувач, Система.

Короткий опис: Даний сценарій описує процес експорту фінансових даних до файлу Excel або імпорту даних із файлу Excel.

Основний потік подій:

1. Користувач обирає функцію експорту або імпорту даних.
2. Система запитує, які дані необхідно експортувати/імпортувати та формат файлу.
3. Користувач вибирає файл або вказує дані для експорту.
4. Система здійснює експорт даних до файлу Excel або імпорт з файлу в базу даних.

Виключення:
Неправильний формат файлу: Якщо вибраний файл не відповідає вимогам системи, з'являється повідомлення про помилку.

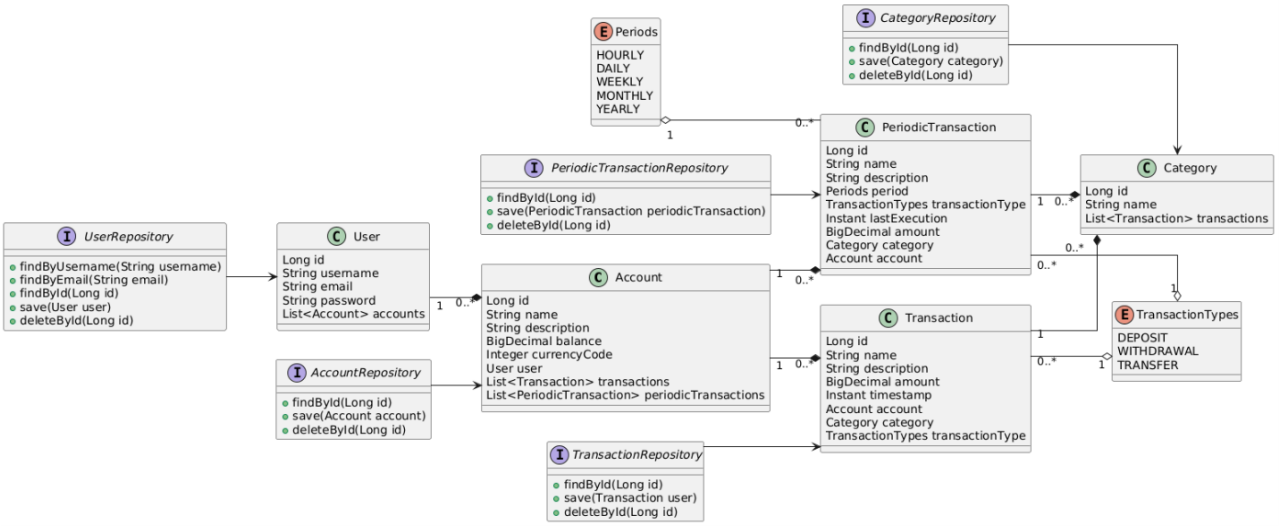
Примітки: Відсутні.

Діаграма класів

Ця діаграма класів моделює структуру фінансової системи, де користувачі можуть керувати рахунками, транзакціями та категоріями витрат. Основний клас User представляє користувача системи, що має унікальний ідентифікатор, ім'я користувача, електронну пошту та пов'язаний список рахунків. Клас Account зберігає інформацію про окремий рахунок користувача, включаючи його баланс, валютний код та зв'язані транзакції.

Транзакції представлені класом Transaction, що зберігає інформацію про суму, час проведення операції, тип транзакції (внесення, зняття чи переказ) та зв'язок з категоріями витрат і рахунками. Існують також періодичні транзакції, що представляються класом PeriodicTransaction, де визначено періоди (годинні, щоденні, щотижневі тощо), а також інформація про останнє виконання транзакції.

Категорії витрат згруповані в класі Category, який має зв'язок з транзакціями, що допомагає організовувати різні типи витрат. Для керування збереженням та обробкою даних у базі існують відповідні репозиторії: UserRepository, AccountRepository, CategoryRepository, TransactionRepository, та PeriodicTransactionRepository. Вони дозволяють шукати, зберігати та видаляти об'єкти системи.



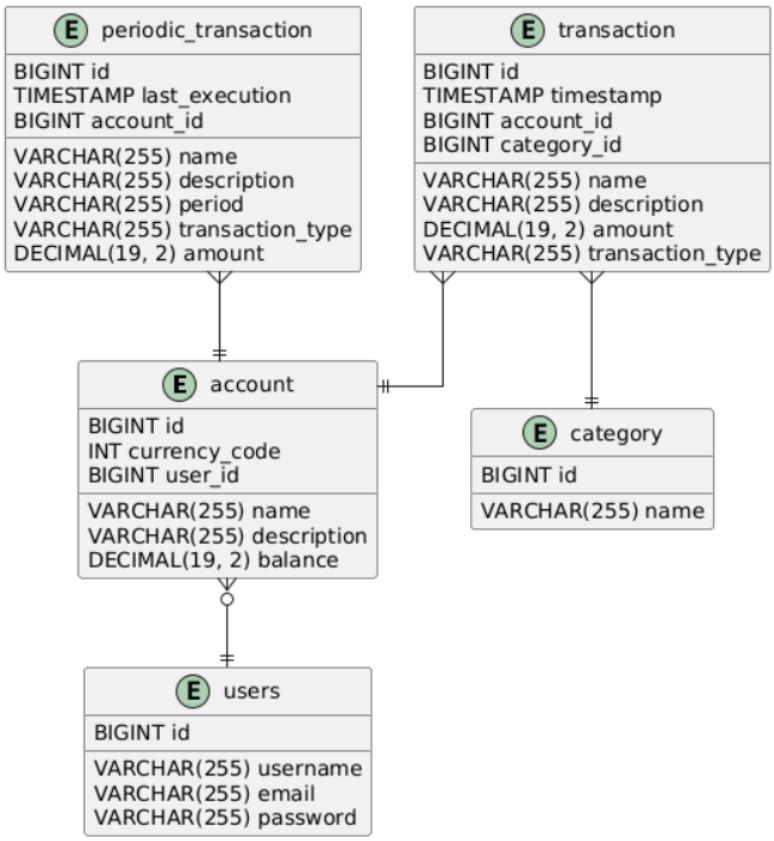
Проектування бази даних

Ця діаграма описує структуру бази даних, яка використовується для управління фінансовими операціями користувачів. У центрі знаходиться сутність account (рахунок), яка містить інформацію про фінансові рахунки користувача, такі як баланс, опис та валютний код. Кожен рахунок належить певному користувачу, що відображено через зв'язок з сутністю **users**, яка зберігає дані про користувачів системи, включаючи ім'я, електронну пошту та пароль.

Сутність transaction (транзакція) представляє фінансові операції, які здійснюються на рахунку. Кожна транзакція містить суму, опис, час проведення операції, тип транзакції, а також пов'язана з конкретним рахунком і категорією витрат. **category** (категорія) визначає тип витрат або доходів, і вона пов'язана з транзакціями, що дозволяє класифікувати фінансові операції за певними категоріями.

Сутність periodic_transaction описує періодичні транзакції, які автоматично виконуються через певний інтервал часу (день, тиждень, місяць тощо). Ця таблиця містить інформацію про періоди, тип транзакції, суму та останню дату виконання. Періодичні транзакції також прив'язані до певних рахунків, що дозволяє автоматизувати регулярні фінансові операції для користувача.

Зв'язки між таблицями відображають залежності між сутностями: рахунки пов'язані з користувачами, транзакції та періодичні транзакції пов'язані з рахунками, а транзакції також класифікуються за категоріями.



Висновки

У цій лабораторній роботі було виконано детальний аналіз теми, розроблено діаграму прецедентів та діаграму класів, а також спроектовано базу даних для управління фінансовими операціями. Отримані результати дозволяють краще зрозуміти структуру системи та взаємозв'язки між її компонентами, що є важливим кроком у проектуванні програмного забезпечення для особистої бухгалтерії.

Вихідний код

```
// file: src/main/java/org/example/accounting/api/model/User.java

package org.example.accounting.api.model;


import jakarta.persistence.*;

import lombok.Getter;

import lombok.Setter;

import org.hibernate.proxy.HibernateProxy;


import java.util.*;


@Getter

@Setter

@Entity

@Table(name = "USERS")

public class User {

    @Id

    @GeneratedValue(strategy = GenerationType.SEQUENCE)

    @Column(nullable = false)

    private Long id;


    @Column(name = "username")

    private String username;
```

```
    @Column(name = "email")
    private String email;

    @Column(name = "password")
    private String password;

    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL, orphanRemoval =
true)
    private List<Account> accounts = new ArrayList<>();

    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL, orphanRemoval =
true)
    private List<Transaction> transactions = new ArrayList<>();

    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL, orphanRemoval =
true)
    private List<PeriodicTransaction> periodicTransactions = new ArrayList<>();

    @ManyToMany(mappedBy = "users")
    private List<Family> families = new ArrayList<>();

    @OneToMany(mappedBy = "creator", orphanRemoval = true)
    private Set<Family> families_created = new LinkedHashSet<>();

    public void addFamilyCreated(Family family) {
        this.families_created.add(family);
        family.setCreator(this);
    }

    public void addFamily(Family family) {
        this.families.add(family);
        family.getUsers().add(this);
    }
```

```
    public void addAccount(Account account) {
        this.accounts.add(account);
        account.setUser(this);
    }


    @Override
    public final boolean equals(Object o) {
        if (this == o) return true;
        if (o == null) return false;
        Class<?> oEffectiveClass = o instanceof HibernateProxy ?
((HibernateProxy) o).getHibernateLazyInitializer().getPersistentClass() :
o.getClass();
        Class<?> thisEffectiveClass = this instanceof HibernateProxy ?
((HibernateProxy) this).getHibernateLazyInitializer().getPersistentClass() :
this.getClass();
        if (thisEffectiveClass != oEffectiveClass) return false;
        User user = (User) o;
        return getId() != null && Objects.equals(getId(), user.getId());
    }


    @Override
    public final int hashCode() {
        return this instanceof HibernateProxy ? ((HibernateProxy)
this).getHibernateLazyInitializer().getPersistentClass().hashCode() :
getClass().hashCode();
    }
}


// file: src/main/java/org/example/accounting/api/model/Family.java
package org.example.accounting.api.model;


import jakarta.persistence.*;
import lombok.Getter;
```

```
import lombok.Setter;

import java.util.ArrayList;
import java.util.LinkedHashSet;
import java.util.List;
import java.util.Set;

@Getter
@Setter
@Entity
@Table(name = "FAMILIES")
public class Family {
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE)
    @Column(nullable = false)
    private Long id;

    private String name;

    @ManyToMany
    @JoinTable(name = "FAMILIES_users",
            joinColumns = @JoinColumn(name = "family_"),
            inverseJoinColumns = @JoinColumn(name = "users_"))
    private List<User> users = new ArrayList<>();

    @ManyToOne
    @JoinColumn(name = "user_id")
    private User creator;

    @OneToMany(mappedBy = "family", orphanRemoval = true)
    private Set<Account> accounts = new LinkedHashSet<>();
```

```
    public void addAccount(Account account) {
        accounts.add(account);
        account.setFamily(this);
    }
}


// file: src/main/java/org/example/accounting/api/model/Account.java
package org.example.accounting.api.model;

import jakarta.persistence.*;
import lombok.Getter;
import lombok.Setter;

import java.math.BigDecimal;
import java.util.ArrayList;
import java.util.List;

@Getter
@Setter
@Entity
@Table(name = "account")
public class Account {
  @Id
  @GeneratedValue(strategy = GenerationType.SEQUENCE)
  @Column(nullable = false)
  private Long id;

  @Column(name = "name")
  private String name;

  @Column(name = "description")
```

```
  private String description;

  @Column(name = "balance", precision = 19, scale = 2)
  private BigDecimal balance;

  @Column(name = "currency_code")
  private Integer currencyCode;

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

  @OrderBy("timestamp ASC")
  @OneToMany(mappedBy = "account", cascade = CascadeType.ALL, orphanRemoval =
true)
  private List<Transaction> transactions = new ArrayList<>();

  @OneToMany(mappedBy = "account", cascade = CascadeType.ALL, orphanRemoval =
true)
  private List<PeriodicTransaction> periodicTransactions = new ArrayList<>();

  @ManyToOne
  @JoinColumn(name = "family_id")
  private Family family;

  public void addTransaction(Transaction transaction) {
    transactions.add(transaction);
    transaction.setAccount(this);
  }

  public void removeTransaction(Transaction transaction) {
    transactions.remove(transaction);
    transaction.setAccount(null);
```

```
  }

  public void addPeriodicTransaction(PeriodicTransaction transaction) {
    periodicTransactions.add(transaction);
    transaction.setAccount(this);
  }

  public void removePeriodicTransaction(PeriodicTransaction transaction) {
    periodicTransactions.remove(transaction);
    transaction.setAccount(null);
  }

}

// file: src/main/java/org/example/accounting/api/model/Periods.java
package org.example.accounting.api.model;

public enum Periods {
    MINUTELY, HOURLY, DAILY, WEEKLY, MONTHLY, YEARLY;
}


// file: src/main/java/org/example/accounting/api/model/Category.java
package org.example.accounting.api.model;

import jakarta.persistence.*;
import lombok.Getter;
import lombok.Setter;

import java.util.ArrayList;
import java.util.List;
```

```
@Getter
@Setter
@Entity
@Table(name = "category")
public class Category {
  @Id
  @GeneratedValue(strategy = GenerationType.SEQUENCE)
  @Column(nullable = false)
  private Long id;

  @Column(name = "name")
  private String name;

  @OneToMany(mappedBy = "category", orphanRemoval = true)
  private List<Transaction> transactions = new ArrayList<>();

  @OneToMany(mappedBy = "category", orphanRemoval = true)
  private List<PeriodicTransaction> periodicTransactions = new ArrayList<>();

}

// file: src/main/java/org/example/accounting/api/model/ExportType.java
package org.example.accounting.api.model;

public enum ExportType {
    CSV, XLS
}


// file: src/main/java/org/example/accounting/api/model/Transaction.java
package org.example.accounting.api.model;
```

```
import jakarta.persistence.*;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

import java.math.BigDecimal;
import java.time.Instant;

@Getter
@Setter
@Entity
@NoArgsConstructor
@Table(name = "transaction")
public class Transaction {
  @Id
  @GeneratedValue(strategy = GenerationType.SEQUENCE)
  @Column(nullable = false)
  private Long id;

  @Column(name = "name")
  private String name;

  @Column(name = "description")
  private String description;

  @Column(name = "amount", precision = 19, scale = 2)
  private BigDecimal amount;

  @Column(name = "timestamp")
  private Instant timestamp = Instant.now();
```

```
@ManyToOne
@JoinColumn(name = "account_id")
private Account account;

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

@Enumerated(EnumType.STRING)
@Column(name = "transaction_type")
private TransactionTypes transactionType;

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

private Transaction(Builder builder) {
  this.name = builder.name;
  this.description = builder.description;
  this.amount = builder.amount;
  this.category = builder.category;
  this.transactionType = builder.transactionType;
}

public static class Builder {
  private String name;
  private String description;
  private BigDecimal amount;
  private Category category;
  private TransactionTypes transactionType;

  public Builder name(String name) {
```

```
      this.name = name;
      return this;
    }

    public Builder description(String description) {
      this.description = description;
      return this;
    }

    public Builder amount(BigDecimal amount) {
      this.amount = amount;
      return this;
    }

    public Builder category(Category category) {
      this.category = category;
      return this;
    }

    public Builder transactionType(TransactionTypes transactionType) {
      this.transactionType = transactionType;
      return this;
    }

    public Transaction build() {
      return new Transaction(this);
    }
  }


}
```

```
// file: src/main/java/org/example/accounting/api/model/TransactionTypes.java
package org.example.accounting.api.model;

public enum TransactionTypes {
    DEPOSIT, WITHDRAWAL
}


// file: src/main/java/org/example/accounting/api/model/PeriodicTransaction.java
package org.example.accounting.api.model;

import jakarta.persistence.*;
import lombok.Getter;
import lombok.Setter;

import java.math.BigDecimal;
import java.time.Instant;

@Getter
@Setter
@Entity
@Table(name = "periodic_transaction")
public class PeriodicTransaction {
  @Id
  @GeneratedValue(strategy = GenerationType.SEQUENCE)
  @Column(nullable = false)
  private Long id;

  @Column(name = "name")
  private String name;

  @Column(name = "description")
```

```
    private String description;

    @Enumerated(EnumType.STRING)
    @Column(name = "period")
    private Periods period;

    @Enumerated(EnumType.STRING)
    @Column(name = "transaction_type")
    private TransactionTypes transactionType;

    @Column(name = "last_execution")
    private Instant lastExecution;

    @Column(name = "timestamp")
    private Instant timestamp = Instant.now();

    @Column(name = "amount", precision = 19, scale = 2)
    private BigDecimal amount;

    @ManyToOne
    @JoinColumn(name = "account_id")
    private Account account;

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

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

}
```

```java
// file: src/main/java/org/example/accounting/api/repository/UserRepository.java
package org.example.accounting.api.repository;

import org.example.accounting.api.model.User;
import org.springframework.data.jpa.repository.JpaRepository;

import java.util.Optional;

public interface UserRepository extends JpaRepository<User, Long> {
    Optional<User> findByEmail(String email);

    Optional<User> findByUsername(String username);

    boolean existsByUsername(String username);
}
```

```java
// file:
src/main/java/org/example/accounting/api/repository/FamilyRepository.java
package org.example.accounting.api.repository;

import org.example.accounting.api.model.Family;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface FamilyRepository extends JpaRepository<Family, Long> {

}
```

```
// file:
src/main/java/org/example/accounting/api/repository/AccountRepository.java

package org.example.accounting.api.repository;


import org.example.accounting.api.model.Account;

import org.springframework.data.jpa.repository.JpaRepository;


import java.util.List;


public interface AccountRepository extends JpaRepository<Account, Long> {


    List<Account> findByUserUsername(String username);


}



// file:
src/main/java/org/example/accounting/api/repository/CategoryRepository.java

package org.example.accounting.api.repository;


import org.example.accounting.api.model.Category;

import org.springframework.data.jpa.repository.JpaRepository;


public interface CategoryRepository extends JpaRepository<Category, Long> {

}



// file:
src/main/java/org/example/accounting/api/repository/TransactionRepository.java

package org.example.accounting.api.repository;


import org.example.accounting.api.model.Transaction;

import org.springframework.data.jpa.repository.JpaRepository;
```

```
public interface TransactionRepository extends JpaRepository<Transaction, Long>
{

}
```

```
// file:
src/main/java/org/example/accounting/api/repository/PeriodicTransactionRepositor
y.java

package org.example.accounting.api.repository;


import org.example.accounting.api.model.PeriodicTransaction;

import org.springframework.data.jpa.repository.JpaRepository;


public interface PeriodicTransactionRepository extends
JpaRepository<PeriodicTransaction, Long> {

}
```