ФЕДЕРАЛЬНОЕ АГЕНСТВО ПО ОБРАЗОВАНИЮ

ГОСУДАРСТВЕННОЕ ОБРАЗОВАТЕЛЬНОЕ УЧЕРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

НИЖЕГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ИМ. Р.Е. АЛЕКСЕЕВА

ИНСТИТУТ РАДИОЭЛЕКТРОНИКИ И ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Кафедра «Вычислительные системы и технологии»

Функциональное программирование

Пояснительная записка

к курсовой работе

**Тема:  
"Система учёта успеваемости студентов"**

Выполнил студент группы 22-ПО

Соменков Ф.В.

Провел ст. преподаватель кафедры ВСТ

Тарасов А.В.

г. Нижний Новгород

2025

**СОДЕРЖАНИЕ**

|  |  |
| --- | --- |
| **Введение** …………………………………………………………….………  **Техническое задание** …………………………………………………...…  **Постановка задачи** ………………………………………………….…….  **Анализ задачи** …………………………………………………………….  **Теоретическая часть** .................................................................................  **Практическая часть** ....................................................................................  **Описание алгоритма работы** …………………………….………………  **Использованные библиотеки и модули программы** …………………  **Применённые функциональные подходы и техники**…………………  **Примеры**……………………………………………………………………. | 2  3  4  5  7  9  11  13  15  17 |
| **Приложение**………………………………………………………………… | 21 |

**1. Введение**

В условиях цифровой трансформации образования и повсеместного внедрения информационных систем актуальной задачей становится эффективный учет и анализ успеваемости студентов. Традиционные бумажные журналы и локальные базы данных уже не справляются с возрастающими объемами информации, требованиями к скорости обработки данных и необходимостью комплексного анализа учебных результатов.

Современные образовательные учреждения сталкиваются с рядом вызовов:

* **Рост количества данных** (студенты, дисциплины, оценки);
* **Необходимость оперативного доступа** к актуальной информации;
* **Требования к аналитике** (средние баллы, динамика успеваемости);
* **Минимизация рутинных операций** при внесении и корректировке данных.

В этих условиях разработка специализированного программного обеспечения для автоматизированного учета успеваемости становится критически важной. Такие системы позволяют:

* Централизованно хранить и обрабатывать учебные данные;
* Оперативно формировать отчеты и статистические показатели;
* Снижать вероятность ошибок при ручном вводе информации;
* Обеспечивать прозрачность учебного процесса.

Функциональное программирование на языке Scala представляет собой мощный инструмент для создания подобных систем. Благодаря таким особенностям, как:

* **Типобезопасность** - предотвращение ошибок на уровне компиляции;
* **Неизменяемые структуры данных** - гарантия целостности информации;
* **Эффективное управление побочными эффектами** (через Cats Effect);
* **Параллельные вычисления** для обработки больших объемов данных,

Scala идеально подходит для разработки надежных и масштабируемых решений в образовательной сфере. Использование ScalaFX для создания пользовательского интерфейса дополнительно обеспечивает:

* Кроссплатформенность (работа на Windows, macOS, Linux);
* Современный и интуитивно понятный интерфейс;
* Реактивность - мгновенное отображение изменений данных.

Разработанная система студенческого учета демонстрирует, как принципы функционального программирования могут быть успешно применены для решения практических задач в сфере образования, сочетая в себе надежность, производительность и удобство использования.

**2. Техническое задание**

**2.1 Наименование разработки**

Система учёта успеваемости студентов

**2.2 Основание для разработки**

Курсовая работа по дисциплине "Функциональное программирование".

**2.3 Назначение программы**

Автоматизация учета успеваемости студентов с возможностью управления данными (CRUD) и анализа статистики.

2.4 Требования к системе

* **Управление студентами**:
  + Добавление, просмотр, удаление записей.
  + Хранение данных: ID, ФИО, курс, группа.
* **Управление предметами**:
  + Добавление и проверка уникальности.
* **Управление оценками**:
  + Ввод оценок (1–5) с привязкой к студенту и предмету.
  + Валидация корректности данных.
* **Статистика**:
  + Расчет среднего балла по студентам, группам, курсам.
* **Отчетность**:
  + Просмотр структурированных данных (по курсам и группам).

**2.5 Требования к техническим средствам**

* **ОС**: Windows, macOS, Linux (кросс-платформенность за счет JVM).
* **Минимальные требования**:
  + JRE 8+ (для работы ScalaFX).
  + Поддержка UTF-8 (для корректного отображения кириллицы).
* **Рекомендуется**:
  + 4 ГБ ОЗУ, 200 МБ свободного места.

**3. Постановка задачи**

Для достижения цели курсовой работы ставятся следующие задачи:

* Разработать десктоп-приложение на языке Scala с использованием ScalaFX для графического интерфейса и Cats Effect для управления побочными эффектами.
* Реализовать CRUD-функционал для работы с сущностями:
  + Студенты (добавление, просмотр, удаление)
  + Предметы (добавление с проверкой уникальности)
  + Оценки (ввод с валидацией значений 1–5)
* Обеспечить расчёт статистики по успеваемости:
  + Средний балл студента
  + Средний балл группы
  + Средний балл курса
* Реализовать структурированный просмотр всех данных с группировкой по курсам и группам.
* Организовать валидацию ввода (проверка пустых полей, числовых значений, существования записей).
* Обеспечить устойчивую работу приложения с обработкой ошибок и выводом пользовательских сообщений.
* Предусмотреть удобный интерфейс с вкладками для разных операций и областью вывода результатов.
* Использовать in-memory хранилище данных (без персистентности между запусками).
* Поддержать русскоязычный интерфейс для удобства пользователя.

**4. Анализ задачи**

Scala, как функциональный язык, работающий на JVM, предоставляет мощные инструменты для разработки приложений с чёткой структурой и безопасной обработкой данных. Для реализации графического интерфейса используется **ScalaFX** – обёртка над JavaFX, позволяющая создавать кросс-платформенные desktop-приложения. Управление побочными эффектами и асинхронными операциями осуществляется с помощью **Cats Effect**, что делает код более предсказуемым и удобным для тестирования.

**Преимущества выбранных технологий**

* **ScalaFX** обеспечивает:
  + Чёткое разделение логики и интерфейса.
  + Поддержку реактивного программирования (наблюдаемые свойства, привязки данных).
  + Кросс-платформенность (Windows, macOS, Linux).
* **Cats Effect** позволяет:
  + Избегать "callback hell" при асинхронных операциях.
  + Чётко разделять чистые и эффектные вычисления.
  + Обрабатывать ошибки в функциональном стиле.
* **In-memory хранилище** (без СУБД) упрощает разработку, но требует ручного управления состоянием.

**Потенциальные сложности**

1. **Масштабируемость данных**
   * Хранение всех данных в памяти может стать проблемой при большом количестве студентов и оценок.
   * Решение:
     + Ограничить максимальное количество записей (если допустимо по ТЗ).
     + В будущем можно заменить на персистентное хранилище (SQLite, PostgreSQL).
2. **Производительность при расчёте статистики**
   * При большом объёме данных вычисление средних баллов может замедлить интерфейс.
   * Решение:
     + Кэшировать результаты статистики.
     + Выносить тяжёлые вычисления в отдельный поток (через IO).
3. **Обработка ошибок ввода**
   * Пользователь может ввести некорректные данные (буквы вместо цифр, несуществующий ID).
   * Решение:
     + Валидация на уровне GUI (подсветка ошибок).
     + Чёткие сообщения об ошибках через ErrorHandler.
4. **Тестирование GUI**
   * Автоматическое тестирование интерфейса сложнее, чем тестирование бизнес-логики.
   * Решение:
     + Вынести максимум логики в отдельные модули (StudentService, Repository).
     + Тестировать их изолированно (scalatest + cats-effect-testing).

**Оптимизации**

* **Реактивный интерфейс** – обновление данных только при изменениях (через наблюдаемые свойства ScalaFX).
* **Буферизация операций** – если пользователь быстро вводит данные, можно накапливать изменения и применять их пачкой.
* **Ленивые вычисления** – статистика пересчитывается только при явном запросе, а не при каждом изменении оценок.

Выбранный стек технологий (Scala + ScalaFX + Cats Effect) хорошо подходит для учебного проекта: он позволяет создать структурированное, поддерживаемое приложение с чётким разделением ответственности между слоями. Основные риски связаны с масштабированием in-memory хранилища и производительностью при больших данных, но в рамках курсовой работы эти ограничения допустимы.

**5. Теоретическая часть**

В данной работе я использую ScalaFX (обёртку над JavaFX) для создания графического интерфейса и Cats Effect (библиотеку для управления побочными эффектами) для организации асинхронных операций и обработки ошибок.

**1. Архитектура приложения**

Приложение построено на трёх основных уровнях:

* **GUI (ScalaFX)** – отвечает за отображение данных и взаимодействие с пользователем.
* **Service-слой** – содержит бизнес-логику (добавление, удаление, расчёт статистики).
* **Repository (In-memory хранилище)** – временное хранение данных в оперативной памяти.

**2. Работа с данными**

Для управления сущностями используются case-классы Scala:

* Student (ID, имя, курс, группа)
* Subject (название предмета)
* Grade (ID студента, предмет, оценка)

Данные хранятся в изменяемых списках (var List), что позволяет:

* Быстро добавлять и удалять записи.
* Обеспечивать простую валидацию (проверка уникальности, корректности оценок).

**3. Обработка событий и реактивность**

ScalaFX предоставляет наблюдаемые свойства, которые автоматически обновляют интерфейс при изменении данных. Например:

val nameField = new TextField() {

text.onChange { (\_, \_, newValue) =>

errorHandler.validateField(newValue, "Имя")

}

}

**4. Управление состоянием с Cats Effect**

Для безопасной работы с эффектами (ввод-вывод, ошибки) используется IO-монада:

def addStudent(id: Int, name: String): IO[Unit] =

repo.getStudent(id).flatMap {

case Some(\_) => IO.raiseError(new Exception("Студент уже существует"))

case None => repo.addStudent(Student(id, name, course, group))

}

Это позволяет:

* Избегать блокировки GUI при долгих операциях.
* Чётко разделять успешные и ошибочные сценарии.

**5. Валидация данных**

Реализованы проверки:

* Пустые поля (например, имя студента не должно быть пустым).
* Оценки только от 1 до 5.
* Уникальность ID студентов и названий предметов.

Ошибки обрабатываются через ErrorHandler, который выводит сообщения в GUI:

errorHandler.validateFields(

(idField.text.value, "ID студента"),

(nameField.text.value, "Имя")

)

**6. Оптимизации**

* **Статистика** вычисляется "лениво" (только при запросе).
* **In-memory хранилище** упрощает разработку, но требует перехода на БД в реальных проектах.
* **ScalaFX-компоненты** переиспользуются (например, createFormField для генерации полей ввода).

**Выбранный подход (ScalaFX + Cats Effect) обеспечивает:**Чистую архитектуру.  
Безопасную работу с состоянием.  
Удобный интерфейс с валидацией.  
При этом система остаётся гибкой для дальнейшего расширения.

**6. Практическая часть**

Разработка системы учета студентов включала несколько ключевых этапов, каждый из которых требовал внимательной проработки архитектуры и реализации:

**1. Проектирование структуры данных и интерфейса**

* Созданы case-классы для хранения данных:

case class Student(id: Int, name: String, course: Int, groupName: String)

case class Subject(name: String)

case class Grade(studentId: Int, subjectName: String, value: Int)

* Разработан графический интерфейс на ScalaFX с вкладками для каждой операции (добавление, просмотр, статистика).

**2. Реализация хранилища (InMemoryRepository)**

* Данные хранятся в изменяемых списках (var List), что обеспечивает:
  + Быстрое добавление/удаление записей.
  + Простую валидацию (проверка дубликатов, корректности оценок).
* Пример метода добавления студента:

def addStudent(student: Student): IO[Unit] = IO {

studentState = student :: studentState

}

**3. Настройка бизнес-логики (StudentService)**

* Реализованы **CRUD-операции** с проверками:

def createStudent(id: Int, name: String): IO[Unit] =

repo.getStudent(id).flatMap {

case Some(\_) => IO.raiseError(new Exception("Студент уже существует"))

case None => repo.addStudent(Student(id, name, course, group))

}

* Расчет статистики (средние баллы) выполняется по запросу для оптимизации производительности.

**4. Разработка GUI (ScalaFX)**

* **Формы ввода**:
  + Поля с валидацией (например, оценка — число от 1 до 5).
  + Кнопки с привязкой к IO-операциям.
* **Отображение данных**:
  + Динамическое обновление текстовых областей (например, список студентов).
  + Группировка по курсам и группам в разделе «Все данные».

**5. Обработка ошибок (ErrorHandler)**

* Централизованный вывод сообщений через Label:

def showError(message: String): Unit = runOnFxThread {

errorLabel.text = message

errorLabel.style = "-fx-text-fill: red;"

}

* Валидация полей:

def validateFields(fields: (String, String)\*): IO[Unit] = IO {

fields.foreach { case (value, name) =>

if (value.isEmpty) throw new IllegalArgumentException(s"Поле '$name' пустое")

}

}

**6. Оптимизации**

* **Реактивный интерфейс**:
  + Данные обновляются только при изменениях (через ObjectProperty).
* **Ленивые вычисления**:
  + Статистика пересчитывается при явном запросе, а не при каждом изменении оценок.
* **Минимизация блокировок**:
  + Все операции с IO выполняются асинхронно, не блокируя GUI.

**7. Описание алгоритма работы**

1. **Инициализация приложения**
   * При запуске программы загружаются тестовые данные (студенты, предметы, оценки) в InMemoryRepository.
   * Создается графический интерфейс (ScalaFX) с вкладками для операций CRUD и статистики.
2. **Настройка хранилища и сервисов**
   * Инициализируется Repository с пустыми или предзаполненными списками:

private var studentState: List[Student] = List()

private var subjectState: List[Subject] = List()

private var gradeState: List[Grade] = List()

* + Создается экземпляр StudentService, связанный с Repository.

1. **Обработка действий пользователя**
   * **Добавление студента**:
     1. Пользователь заполняет поля (ID, имя, курс, группа).
     2. При нажатии кнопки:
        + Валидация полей (проверка на пустоту, уникальность ID).
        + Вызов studentService.createStudent().
        + Обновление интерфейса (очистка полей, вывод сообщения об успехе).
   * **Просмотр статистики**:
     1. Запрос к repo.calculateStats().
     2. Расчет средних баллов по студентам/группам/курсам.
     3. Форматирование данных в текстовом виде для отображения.
2. **Работа с данными**
   * **In-memory операции**:
     1. Добавление: studentState = student :: studentState.
     2. Удаление: studentState = studentState.filterNot(\_.id == id).
   * **Валидация оценок**:

if (value < 1 || value > 5) throw new IllegalArgumentException("Оценка должна быть от 1 до 5")

1. **Обработка ошибок**
   * Ошибки ввода (некорректные ID, пустые поля) перехватываются ErrorHandler:

def handleIOError[A](io: IO[A]): IO[Unit] =

io.attempt.flatMap {

case Left(ex) => IO(showError(s"Ошибка: ${ex.getMessage}"))

case Right(\_) => IO.unit

}

1. **Формирование отчетов**
   * **Статистика**:
     1. Группировка оценок по студентам/группам.
     2. Расчет среднего балла:

grades.map(\_.value).sum.toDouble / grades.length

* + **Просмотр всех данных**:
    1. Данные выводятся в TextArea с группировкой по курсам и группам.

1. **Завершение работы**
   * При закрытии приложения все данные сбрасываются.
   * Для сохранения данных между сеансами требуется интеграция с БД (дополнительный этап).

**8. Использованные библиотеки и модули программы**

**1. Основные библиотеки**

* **ScalaFX** – для создания графического интерфейса (GUI) на основе JavaFX.
  + scalafx.scene, scalafx.scene.control, scalafx.scene.layout – компоненты интерфейса (кнопки, поля ввода, вкладки).
  + scalafx.application – управление жизненным циклом приложения.
* **Cats Effect** – для работы с эффектами и асинхронными операциями.
  + cats.effect.IO – монадическая обработка побочных эффектов.
  + cats.implicits – синтаксические расширения для работы с монадами.
* **Scala Standard Library** – базовые коллекции (List, Map) и case-классы.

**2. Модули программы**

**1. domain (Модели данных)**

Определяет структуры данных:

* Student – студент (ID, имя, курс, группа).
* Subject – предмет (название).
* Grade – оценка (ID студента, предмет, значение).
* Stats – статистика (средние баллы по студентам/группам/курсам).

**2. repository (Хранилище данных)**

* InMemoryRepository – хранит данные в оперативной памяти:
  + studentState: List[Student] – список студентов.
  + subjectState: List[Subject] – список предметов.
  + gradeState: List[Grade] – список оценок.
* **Методы**:
  + CRUD-операции (addStudent, deleteSubject и т.д.).
  + Расчет статистики (calculateStats).
  + Получение всех данных (getAllData).

**3. service (Бизнес-логика)**

* StudentService – обрабатывает основные сценарии:
  + Создание/удаление студентов.
  + Получение детальной информации о студенте (включая оценки).
  + Валидация данных перед сохранением.

**4. gui (Графический интерфейс)**

* GUI – главное окно приложения (ScalaFX):
  + Вкладки:
    1. Добавление студента.
    2. Просмотр студента.
    3. Удаление студента.
    4. Добавление предмета.
    5. Добавление оценки.
    6. Статистика.
    7. Просмотр всех данных.
  + Динамические элементы: TextArea, TextField, Button.
* ErrorHandler – вывод сообщений об ошибках/успехе.

**5. errorhandler (Обработка ошибок)**

* **Методы**:
  + showError / showSuccess – цветные сообщения в GUI.
  + validateFields – проверка пустых полей.
  + validateIntField – проверка числовых значений.
  + handleIOError – обработка ошибок в IO-операциях.

**6. main (Точка входа)**

* Инициализация тестовых данных.
* Запуск GUI:

new GUI(studentService, repo).main(Array.empty)

**9. Использованные функциональные подходы и техники**

В данной работе я активно применял принципы функционального программирования на Scala. Вот ключевые аспекты:

**1. Чистые функции и иммутабельность**

* **Case-классы (**Student**,**Subject**,**Grade**)** – неизменяемые (immutable) структуры данных.
* **Функции без побочных эффектов** – например, расчет статистики:

def calculateStats: Stats = {

val studentAvg = grades.groupBy(\_.studentId).mapValues(avg)

*// ... (чистые вычисления)*

}

**2. Работа с эффектами через Cats Effect**

* IO-монада – для изоляции побочных эффектов (ввод-вывод, изменение состояния):

def addStudent(student: Student): IO[Unit] = IO {

studentState = student :: studentState

}

* **Композиция операций** через flatMap и map:

repo.getStudent(id).flatMap {

case Some(\_) => IO.raiseError(...)

case None => repo.addStudent(student)

}

**3. Обработка ошибок в функциональном стиле**

* Either-подобный подход через IO.attempt:

ioOperation.attempt.flatMap {

case Left(ex) => IO(showError(ex.getMessage))

case Right(\_) => IO.unit

}

* **Валидация** с использованием IO.raiseError:

def validateGrade(value: Int): IO[Unit] =

if (value < 1 || value > 5) IO.raiseError(...) else IO.unit

**4. Функциональные абстракции**

* **Работа с коллекциями** (методы map, filter, groupBy):

students.groupBy(\_.course)

* Monad для IO – последовательное выполнение операций:

for {

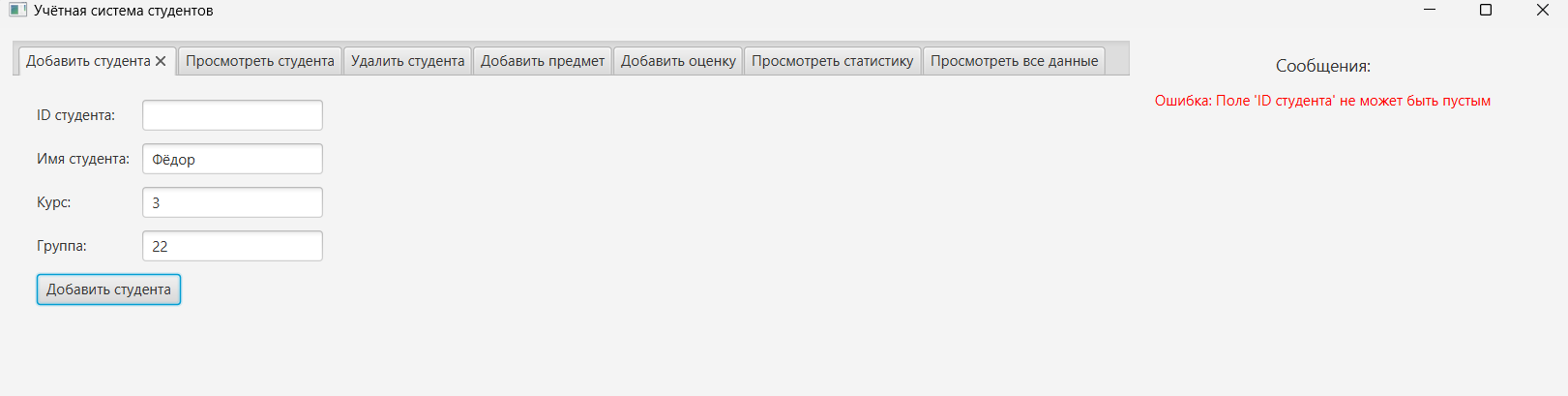
\_ <- validateFields(...)

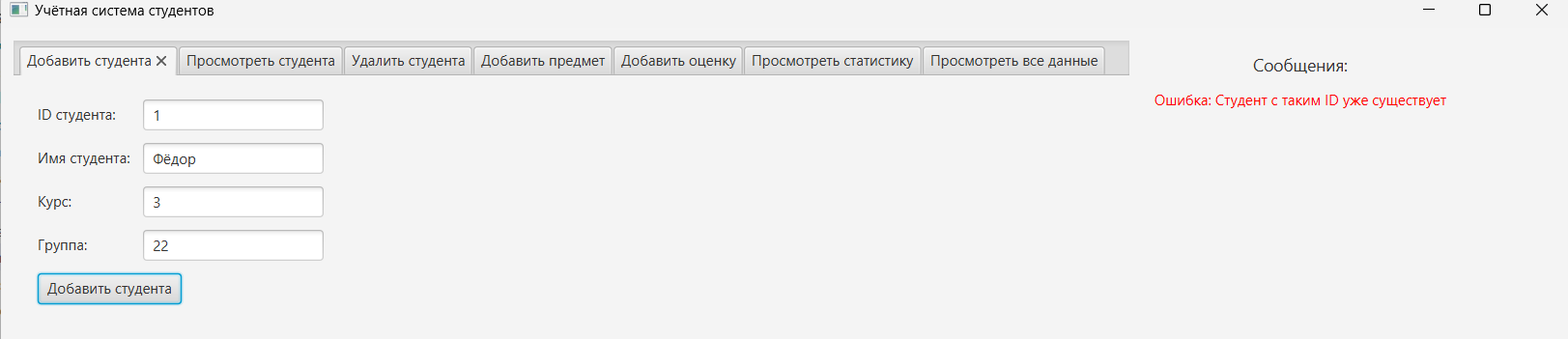
id <- validateIntField(...)

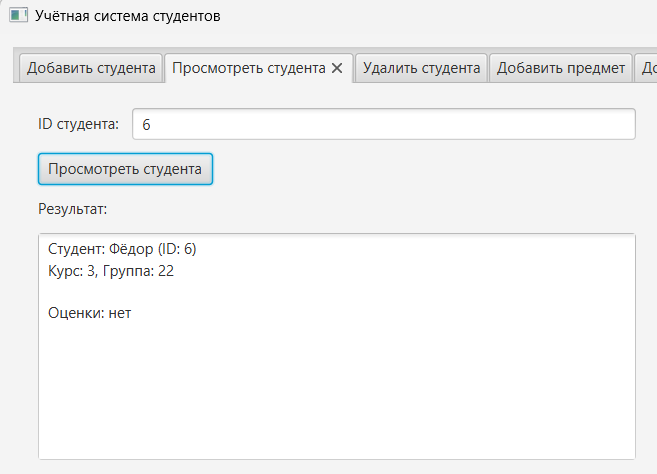
\_ <- repo.addStudent(...)

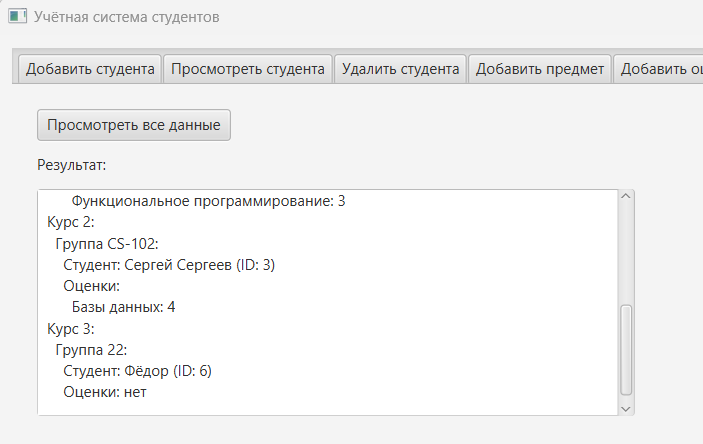
} yield ()

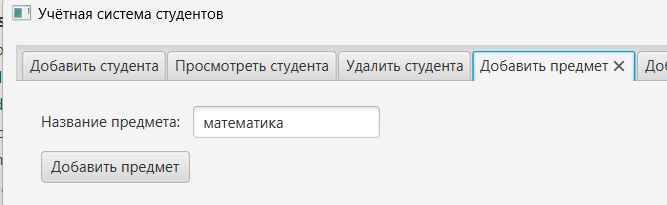
**10. Примеры**

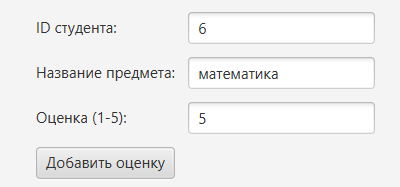
****

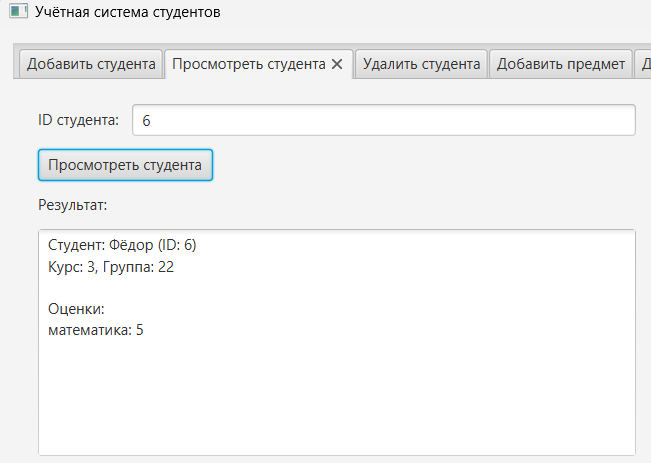
****

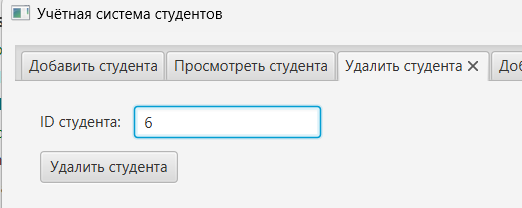
****

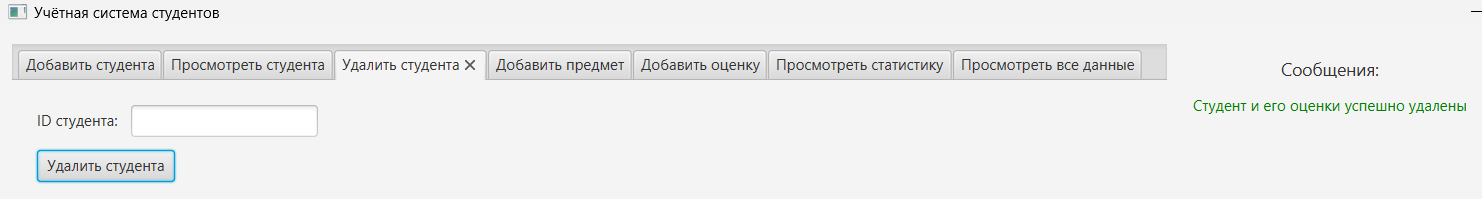
****

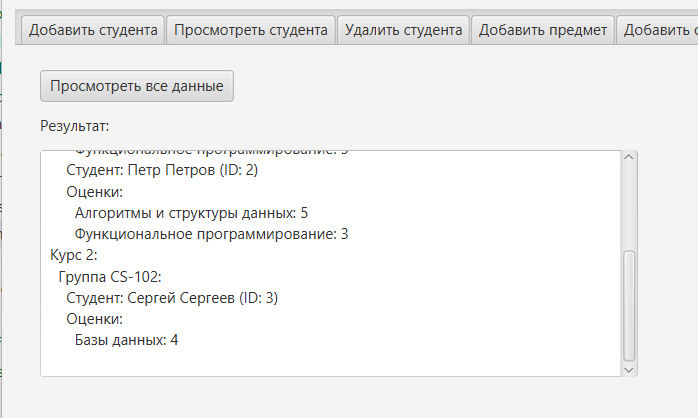
****

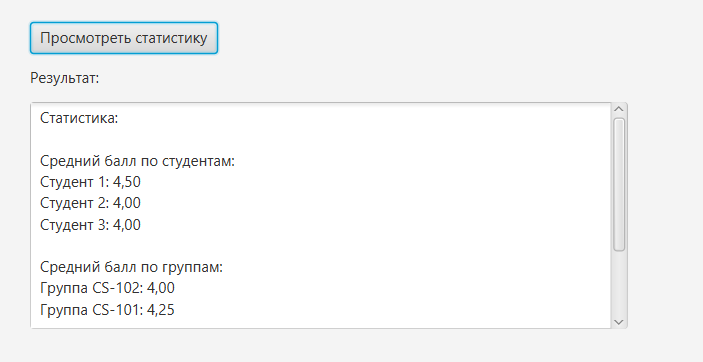
****

****

****

****

****

****

**11. Приложение**

Код:

domain.scala:

package studentaccounting

trait Entity

case class Student(id: Int, name: String, course: Int, groupName: String)

extends Entity

case class Subject(name: String) extends Entity

case class Grade(studentId: Int, subjectName: String, value: Int) extends Entity

case class Stats(

studentAvg: Map[Int, Double],

groupAvg: Map[String, Double],

courseAvg: Map[Int, Double]

)

ErrorHandler.scala:

package studentaccounting

import cats.effect.IO

import scalafx.application.Platform

import scalafx.scene.control.Label

class ErrorHandler(private val errorLabel: Label) {

def runOnFxThread(action: => Unit): Unit = {

Platform.runLater(action)

}

def clearError(): Unit = runOnFxThread {

errorLabel.text = ""

errorLabel.style = "-fx-text-fill: red;"

}

def showError(message: String): Unit = runOnFxThread {

errorLabel.text = message

errorLabel.style = "-fx-text-fill: red;"

}

def showSuccess(message: String): Unit = runOnFxThread {

errorLabel.text = message

errorLabel.style = "-fx-text-fill: green;"

}

def handleIOError[A](io: IO[A]): IO[Unit] = {

io.attempt.flatMap {

case Left(ex) => IO(showError(s"Ошибка: ${ex.getMessage}"))

case Right(\_) => IO.unit

}

}

def validateFields(fields: (String, String)\*): IO[Unit] = IO {

fields.foreach { case (value, name) =>

if (value.isEmpty) {

throw new IllegalArgumentException(s"Поле '$name' не может быть пустым")

}

}

}

def validateIntField(value: String, fieldName: String): IO[Int] =

IO(value.toInt).handleErrorWith(\_ =>

IO.raiseError(

new IllegalArgumentException(s"$fieldName должен быть числом")

)

)

def validateGradeValue(value: Int): IO[Unit] =

if (value < 1 || value > 5) {

IO.raiseError(new IllegalArgumentException("Оценка должна быть от 1 до 5"))

} else IO.unit

}

GUI.scala:

package studentaccounting

import cats.effect.IO

import cats.effect.unsafe.implicits.global

import scalafx.Includes.\_

import scalafx.application.JFXApp3

import scalafx.application.JFXApp3.PrimaryStage

import scalafx.geometry.{Insets, Pos}

import scalafx.scene.Scene

import scalafx.scene.control.\_

import scalafx.scene.layout.{GridPane, HBox, VBox}

import scalafx.scene.text.Font

class GUI(studentService: StudentService[IO], repo: Repository[IO])

extends JFXApp3 {

private val errorLabel = new Label {

style = "-fx-text-fill: red;"

wrapText = true

maxWidth = 300

}

private val errorHandler = new ErrorHandler(errorLabel)

private def createFormField(

labelText: String,

field: TextField,

grid: GridPane,

row: Int

): Unit = {

grid.add(new Label(labelText), 0, row)

grid.add(field, 1, row)

}

private def createButton(buttonText: String, action: => Unit): Button = {

new Button(buttonText) {

onAction = \_ => action

}

}

private def createResultArea(): TextArea = {

new TextArea {

editable = false

wrapText = true

}

}

private def addStudentTab(): Tab = {

val idField = new TextField()

val nameField = new TextField()

val courseField = new TextField()

val groupField = new TextField()

val addStudent = () => {

errorHandler.clearError()

val program = for {

\_ <- errorHandler.validateFields(

(idField.text.value, "ID студента"),

(nameField.text.value, "Имя студента"),

(courseField.text.value, "Курс"),

(groupField.text.value, "Группа")

)

id <- errorHandler.validateIntField(idField.text.value, "ID студента")

course <- errorHandler.validateIntField(courseField.text.value, "Курс")

name = nameField.text.value.trim

group = groupField.text.value.trim

\_ <- repo.getStudent(id).flatMap {

case Some(\_) =>

IO.raiseError(

new IllegalArgumentException("Студент с таким ID уже существует")

)

case None => studentService.createStudent(id, name, course, group)

}

\_ <- IO {

idField.text = ""

nameField.text = ""

courseField.text = ""

groupField.text = ""

errorHandler.showSuccess("Студент успешно добавлен")

}

} yield ()

errorHandler.handleIOError(program).unsafeRunAsync(\_ => ())

}

val grid = new GridPane {

hgap = 10

vgap = 10

padding = Insets(20)

}

createFormField("ID студента:", idField, grid, 0)

createFormField("Имя студента:", nameField, grid, 1)

createFormField("Курс:", courseField, grid, 2)

createFormField("Группа:", groupField, grid, 3)

grid.add(createButton("Добавить студента", addStudent()), 0, 4, 2, 1)

new Tab {

text = "Добавить студента"

content = grid

}

}

private def viewStudentTab(): Tab = {

val idField = new TextField()

val resultArea = createResultArea()

val viewStudent = () => {

errorHandler.clearError()

val program = for {

\_ <- errorHandler.validateFields((idField.text.value, "ID студента"))

id <- errorHandler.validateIntField(idField.text.value, "ID студента")

studentOpt <- studentService.getStudentDetails(id)

\_ <- studentOpt match {

case Some((student, grades)) =>

IO {

val sb = new StringBuilder

sb.append(s"Студент: ${student.name} (ID: ${student.id})\n")

sb.append(

s"Курс: ${student.course}, Группа: ${student.groupName}\n\n"

)

if (grades.nonEmpty) {

sb.append("Оценки:\n")

grades.groupBy(\_.\_1).foreach { case (subject, subjectGrades) =>

sb.append(

s"$subject: ${subjectGrades.map(\_.\_2).mkString(" ")}\n"

)

}

} else {

sb.append("Оценки: нет\n")

}

resultArea.text = sb.toString()

}

case None =>

IO {

resultArea.text = ""

errorHandler.showError("Студент не найден")

}

}

} yield ()

errorHandler.handleIOError(program).unsafeRunAsync(\_ => ())

}

val grid = new GridPane {

hgap = 10

vgap = 10

padding = Insets(20)

}

createFormField("ID студента:", idField, grid, 0)

grid.add(createButton("Просмотреть студента", viewStudent()), 0, 1, 2, 1)

grid.add(new Label("Результат:"), 0, 2)

grid.add(resultArea, 0, 3, 2, 1)

new Tab {

text = "Просмотреть студента"

content = grid

}

}

private def deleteStudentTab(): Tab = {

val idField = new TextField()

val deleteStudent = () => {

errorHandler.clearError()

val program = for {

\_ <- errorHandler.validateFields((idField.text.value, "ID студента"))

id <- errorHandler.validateIntField(idField.text.value, "ID студента")

exists <- repo.getStudent(id).map(\_.isDefined)

\_ <-

if (!exists) {

IO.raiseError(new NoSuchElementException("Студент не найден"))

} else {

studentService.deleteStudentWithGrades(id)

}

\_ <- IO {

idField.text = ""

errorHandler.showSuccess("Студент и его оценки успешно удалены")

}

} yield ()

errorHandler.handleIOError(program).unsafeRunAsync(\_ => ())

}

val grid = new GridPane {

hgap = 10

vgap = 10

padding = Insets(20)

}

createFormField("ID студента:", idField, grid, 0)

grid.add(createButton("Удалить студента", deleteStudent()), 0, 1, 2, 1)

new Tab {

text = "Удалить студента"

content = grid

}

}

private def addSubjectTab(): Tab = {

val nameField = new TextField()

val addSubject = () => {

errorHandler.clearError()

val program = for {

\_ <- errorHandler.validateFields(

(nameField.text.value, "Название предмета")

)

name = nameField.text.value.trim

exists <- repo.getSubject(name).map(\_.isDefined)

\_ <-

if (exists) {

IO.raiseError(

new IllegalArgumentException(

"Предмет с таким названием уже существует"

)

)

} else {

repo.addSubject(Subject(name))

}

\_ <- IO {

nameField.text = ""

errorHandler.showSuccess("Предмет успешно добавлен")

}

} yield ()

errorHandler.handleIOError(program).unsafeRunAsync(\_ => ())

}

val grid = new GridPane {

hgap = 10

vgap = 10

padding = Insets(20)

}

createFormField("Название предмета:", nameField, grid, 0)

grid.add(createButton("Добавить предмет", addSubject()), 0, 1, 2, 1)

new Tab {

text = "Добавить предмет"

content = grid

}

}

private def addGradeTab(): Tab = {

val studentIdField = new TextField()

val subjectField = new TextField()

val gradeField = new TextField()

val addGrade = () => {

errorHandler.clearError()

val program = for {

\_ <- errorHandler.validateFields(

(studentIdField.text.value, "ID студента"),

(subjectField.text.value, "Название предмета"),

(gradeField.text.value, "Оценка")

)

studentId <- errorHandler.validateIntField(

studentIdField.text.value,

"ID студента"

)

gradeValue <- errorHandler.validateIntField(

gradeField.text.value,

"Оценка"

)

\_ <- errorHandler.validateGradeValue(gradeValue)

subjectName = subjectField.text.value.trim

studentExists <- repo.getStudent(studentId).map(\_.isDefined)

\_ <-

if (!studentExists) {

IO.raiseError(new NoSuchElementException("Студент не найден"))

} else IO.unit

subjectExists <- repo.getSubject(subjectName).map(\_.isDefined)

\_ <-

if (!subjectExists) {

IO.raiseError(new NoSuchElementException("Предмет не найден"))

} else IO.unit

\_ <- repo.addGrade(Grade(studentId, subjectName, gradeValue))

\_ <- IO {

studentIdField.text = ""

subjectField.text = ""

gradeField.text = ""

errorHandler.showSuccess("Оценка успешно добавлена")

}

} yield ()

errorHandler.handleIOError(program).unsafeRunAsync(\_ => ())

}

val grid = new GridPane {

hgap = 10

vgap = 10

padding = Insets(20)

}

createFormField("ID студента:", studentIdField, grid, 0)

createFormField("Название предмета:", subjectField, grid, 1)

createFormField("Оценка (1-5):", gradeField, grid, 2)

grid.add(createButton("Добавить оценку", addGrade()), 0, 3, 2, 1)

new Tab {

text = "Добавить оценку"

content = grid

}

}

private def viewStatsTab(): Tab = {

val resultArea = createResultArea()

val viewStats = () => {

errorHandler.clearError()

val program = for {

stats <- repo.calculateStats

\_ <- IO {

val sb = new StringBuilder

sb.append("Статистика:\n\n")

if (stats.studentAvg.nonEmpty) {

sb.append("Средний балл по студентам:\n")

stats.studentAvg.foreach { case (id, avg) =>

sb.append(s"Студент $id: ${"%.2f".format(avg)}\n")

}

} else {

sb.append("Нет данных о студентах\n")

}

if (stats.groupAvg.nonEmpty) {

sb.append("\nСредний балл по группам:\n")

stats.groupAvg.foreach { case (group, avg) =>

sb.append(s"Группа $group: ${"%.2f".format(avg)}\n")

}

} else {

sb.append("\nНет данных по группам\n")

}

if (stats.courseAvg.nonEmpty) {

sb.append("\nСредний балл по курсам:\n")

stats.courseAvg.foreach { case (course, avg) =>

sb.append(s"Курс $course: ${"%.2f".format(avg)}\n")

}

} else {

sb.append("\nНет данных по курсам\n")

}

resultArea.text = sb.toString()

}

} yield ()

errorHandler.handleIOError(program).unsafeRunAsync(\_ => ())

}

val grid = new GridPane {

hgap = 10

vgap = 10

padding = Insets(20)

}

grid.add(createButton("Просмотреть статистику", viewStats()), 0, 0)

grid.add(new Label("Результат:"), 0, 1)

grid.add(resultArea, 0, 2)

new Tab {

text = "Просмотреть статистику"

content = grid

}

}

private def viewAllDataTab(): Tab = {

val resultArea = createResultArea()

val viewAllData = () => {

errorHandler.clearError()

val program = for {

allData <- repo.getAllData

\_ <- IO {

if (allData.isEmpty) {

resultArea.text = "Нет данных о студентах"

} else {

val sb = new StringBuilder

sb.append("Все данные:\n\n")

val groupedByCourse = allData

.groupBy { case (student, \_) => student.course }

.toList

.sortBy(\_.\_1)

groupedByCourse.foreach { case (course, courseStudents) =>

sb.append(s"Курс $course:\n")

val groupedByGroup = courseStudents

.groupBy { case (student, \_) => student.groupName }

.toList

.sortBy(\_.\_1)

groupedByGroup.foreach { case (groupName, groupStudents) =>

sb.append(s" Группа $groupName:\n")

groupStudents.foreach { case (student, grades) =>

sb.append(

s" Студент: ${student.name} (ID: ${student.id})\n"

)

if (grades.nonEmpty) {

sb.append(" Оценки:\n")

grades.groupBy(\_.subjectName).foreach {

case (subject, subjectGrades) =>

sb.append(

s" $subject: ${subjectGrades.map(\_.value).mkString(" ")}\n"

)

}

} else {

sb.append(" Оценки: нет\n")

}

}

}

}

resultArea.text = sb.toString()

}

}

} yield ()

errorHandler.handleIOError(program).unsafeRunAsync(\_ => ())

}

val grid = new GridPane {

hgap = 10

vgap = 10

padding = Insets(20)

}

grid.add(createButton("Просмотреть все данные", viewAllData()), 0, 0)

grid.add(new Label("Результат:"), 0, 1)

grid.add(resultArea, 0, 2)

new Tab {

text = "Просмотреть все данные"

content = grid

}

}

override def start(): Unit = {

stage = new PrimaryStage {

title = "Учётная система студентов"

scene = new Scene(1300, 600) {

val tabPane = new TabPane {

tabs = Seq(

addStudentTab(),

viewStudentTab(),

deleteStudentTab(),

addSubjectTab(),

addGradeTab(),

viewStatsTab(),

viewAllDataTab()

)

}

val mainLayout = new HBox {

spacing = 10

padding = Insets(10)

children = Seq(

tabPane,

new VBox {

spacing = 10

padding = Insets(10)

alignment = Pos.TopCenter

children = Seq(

new Label("Сообщения:") {

font = Font.font(14)

},

errorLabel

)

}

)

}

root = mainLayout

}

}

}

}

main.scala:

package studentaccounting

import cats.effect.{IO, IOApp}

import java.nio.charset.Charset

object Main extends IOApp.Simple {

System.setProperty("file.encoding", "UTF-8")

java.nio.charset.Charset.forName("UTF-8")

val initialData: IO[(List[Student], List[Subject], List[Grade])] = IO {

val students = List(

Student(1, "Иван Иванов", 1, "CS-101"),

Student(2, "Петр Петров", 1, "CS-101"),

Student(3, "Сергей Сергеев", 2, "CS-102")

)

val subjects = List(

Subject("Функциональное программирование"),

Subject("Алгоритмы и структуры данных"),

Subject("Базы данных")

)

val grades = List(

Grade(1, "Функциональное программирование", 5),

Grade(1, "Алгоритмы и структуры данных", 4),

Grade(2, "Функциональное программирование", 3),

Grade(2, "Алгоритмы и структуры данных", 5),

Grade(3, "Базы данных", 4)

)

(students, subjects, grades)

}

override def run: IO[Unit] =

for {

(students, subjects, grades) <- initialData

repo = new InMemoryRepository(students, subjects, grades)

studentService = new StudentService[IO](repo)

\_ <- IO {

scalafx.application.Platform.startup(() => {})

new GUI(studentService, repo).main(Array.empty)

}

} yield ()

}

repository.scala:

package studentaccounting

import cats.effect.IO

import cats.implicits.\_

trait Repository[F[\_]] {

def getAllStudents: F[List[Student]]

def getStudent(id: Int): F[Option[Student]]

def addStudent(student: Student): F[Unit]

def deleteStudent(id: Int): F[Unit]

def getAllSubjects: F[List[Subject]]

def getSubject(name: String): F[Option[Subject]]

def addSubject(subject: Subject): F[Unit]

def deleteSubject(name: String): F[Unit]

def getAllGrades: F[List[Grade]]

def getGradesForStudent(studentId: Int): F[List[Grade]]

def addGrade(grade: Grade): F[Unit]

def deleteGradesForStudent(studentId: Int): F[Unit]

def calculateStats: F[Stats]

def getAllData: F[List[(Student, List[Grade])]]

}

class InMemoryRepository(

students: List[Student],

subjects: List[Subject],

grades: List[Grade]

) extends Repository[IO] {

private var studentState = students

private var subjectState = subjects

private var gradeState = grades

override def getAllStudents: IO[List[Student]] = IO(studentState)

override def getStudent(id: Int): IO[Option[Student]] = IO(

studentState.find(\_.id == id)

)

override def addStudent(student: Student): IO[Unit] = IO {

studentState = student :: studentState

}

override def deleteStudent(id: Int): IO[Unit] = IO {

studentState = studentState.filterNot(\_.id == id)

}

override def getAllSubjects: IO[List[Subject]] = IO(subjectState)

override def getSubject(name: String): IO[Option[Subject]] = IO(

subjectState.find(\_.name.equalsIgnoreCase(name))

)

override def addSubject(subject: Subject): IO[Unit] = IO {

subjectState = subject :: subjectState

}

override def deleteSubject(name: String): IO[Unit] = IO {

subjectState = subjectState.filterNot(\_.name.equalsIgnoreCase(name))

}

override def getAllGrades: IO[List[Grade]] = IO(gradeState)

override def getGradesForStudent(studentId: Int): IO[List[Grade]] = IO(

gradeState.filter(\_.studentId == studentId)

)

override def addGrade(grade: Grade): IO[Unit] = IO {

gradeState = grade :: gradeState

}

override def deleteGradesForStudent(studentId: Int): IO[Unit] = IO {

gradeState = gradeState.filterNot(\_.studentId == studentId)

}

override def calculateStats: IO[Stats] = IO {

val studentGrades = gradeState.groupBy(\_.studentId)

val courseStudents = studentState.groupBy(\_.course)

val groupStudents = studentState.groupBy(\_.groupName)

val studentAvg = studentGrades.view.mapValues { grades =>

grades.map(\_.value).sum.toDouble / grades.length

}.toMap

val groupAvg = groupStudents.view.mapValues { students =>

val studentIds = students.map(\_.id).toSet

val relevantGrades =

gradeState.filter(g => studentIds.contains(g.studentId))

if (relevantGrades.isEmpty) 0.0

else relevantGrades.map(\_.value).sum.toDouble / relevantGrades.length

}.toMap

val courseAvg = courseStudents.view.mapValues { students =>

val studentIds = students.map(\_.id).toSet

val relevantGrades =

gradeState.filter(g => studentIds.contains(g.studentId))

if (relevantGrades.isEmpty) 0.0

else relevantGrades.map(\_.value).sum.toDouble / relevantGrades.length

}.toMap

Stats(studentAvg, groupAvg, courseAvg)

}

override def getAllData: IO[List[(Student, List[Grade])]] = IO {

studentState.map { student =>

val studentGrades = gradeState.filter(\_.studentId == student.id)

(student, studentGrades)

}

}

}

service.scala:

package studentaccounting

import cats.Monad

import cats.implicits.\_

class StudentService[F[\_]](repo: Repository[F])(implicit F: cats.Monad[F]) {

def createStudent(

id: Int,

name: String,

course: Int,

groupName: String

): F[Unit] =

repo.addStudent(Student(id, name, course, groupName))

def getStudentDetails(id: Int): F[Option[(Student, List[(String, Int)])]] =

for {

studentOpt <- repo.getStudent(id)

result <- studentOpt match {

case Some(student) =>

for {

grades <- repo.getGradesForStudent(student.id)

gradesWithSubjects = grades.map(g => (g.subjectName, g.value))

} yield Some((student, gradesWithSubjects))

case None => F.pure(None)

}

} yield result

def deleteStudentWithGrades(id: Int): F[Unit] =

for {

\_ <- repo.deleteStudent(id)

\_ <- repo.deleteGradesForStudent(id)

} yield ()

}