Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ ИУ «Информатика и системы управления»

КАФЕДРА ИУ-7 «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

«Разработка базы данных для проведения соревнований Российской Федерации Подводного Рыболовства»

Студент группы **ИУ7-64Б** _____ **П. А. Егорова**

(Подпись, дата) (И.О. Фамилия)

Руководитель _____ **Т. А. Никульшина**

(Подпись, дата) (И.О. Фамилия)

***2023** г.*

РЕФЕРАТ

Курсовая работа 52 с., 12 рис., 2 табл., 21 ист.

СУБД, БД, Realm, iOS, Swift, XCode.

Цель работы: спроектировать и разработать базу данных для проведения соревнований Российской Федерации Подводного Рыболовства.

Результат работы:

— формализована задача и определен необходимый функционал;
— описана структура объектов базы данных, а также сделан выбор СУБД для ее хранения и взаимодействия;
— спроектирован и разработан триггер, осуществляющий перерасчёт очков этапов, участников и команд при добавлении улова;
— спроектирован и разработан интерфейс взаимодействия с базой данных;
— проведено исследование времени работы операции добавления улова, а также функции получения списка спортсменов соревнования в зависимости от количества участников.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Автоматизация — процесс, освобождающий человека от множества механических задач и помогающий ему повысить производительность труда. Автоматизация затронула большое количество процессов в современном мире: от работы кассира в магазине и подсчета стоимости покупок до вычисления значений физических величин в труде ученых. Также автоматически производится подсчет очков на соревнованиях различных дисциплин производится автоматически. Однако существуют организации, судейство соствязаний которых происходит вручную: к примеру, Российская Федерация Подводного Рыболовства (РФПР), которая занимается проведением соревнования по подводной охоте [1]. Деятельность федерации, связанная с регистрацией участников и подсчетом очков, не автоматизирована.

Целью курсовой работы является проектирование и разработка базы данных для проведения соревнований Российской Федерации Подводного Рыболовства.

Для достижения поставленной цели, необходимо решить следующие задачи:

— определить необходимый функционал приложения, предоставляющего доступ к базе данных;
— выделить роли пользователей, а также формализовать данные;
— проанализировать системы управления базами данных и выбрать подходящую систему для хранения данных;
— спроектировать базу данных, описать ее сущности и связи, спроектировать триггер;
— реализовать интерфейс для доступа к базе данных.

Итогом работы станет приложение, предоставляющее доступ к базе данных.

1 Аналитический раздел

Аналитический раздел курсовой работы является важным этапом проектирования приложения, поскольку он определяет основные требования к системе и выбирает наиболее подходящую модель базы данных. В данном разделе будут рассмотрены различные типы баз данных, их преимущества и недостатки, а также проведен анализ существующих систем управления базами данных. На основе полученных результатов будет выбрана оптимальная модель базы данных для реализации приложения. Кроме того, в данном разделе будут определены пользователи системы, формализованы данные и представлены сущности, что позволит более точно определить требования к приложению и разработать его структуру.

1.1 Требования к приложению

Чтобы спортсмены имели возможность сразу же после выхода из воды взвесить улов, подсчет очков и судейство происходит прямо на берегу водоемов, зачастую в полевых условиях, где возможность оборудовать рабочее место с ноутбуком предоставляется редко. Разрабатываемое приложение было бы удобно использовать на более компактных устройствах, таких как телефон или планшет. В связи с чем выбор пал на платформу iOS [2] — создаваемое программное обеспечение будет разработано для смартфонов и электронных планшетов компании Apple [3].

Приложение должно поддерживать определенный функционал:

— возможность входа без авторизации для спортсменов;
— персонализация для судей и администраторов;
— формирование соревнований и их этапов;
— создание команд, деление участников по командам;
— добавление информации об улове участников;
— подсчет очков участников, команд;
— формирование личного и командного рейтингов;

— поиск определенного участника или конкретной команды.

1.2 Пользователи системы

В приложении будут представлены следующие роли:

1) Участник — пользователь, обладающий возможностью просматривать командный и личный рейтинги по любым этапам соревнований, устанавливать параметры поиска, а также искать профили конкретных участников и команд. Роль не требует авторизации.
2) Судья — пользователь, обладающий возможностями участника, а также возможностью создавать и удалять соревнования, создавать, редактировать и удалять участников, команды, добавлять участников в команды и удалять из команд, добавлять, редактировать и удалять уловы участников в этапы соревнований. Роль требует авторизации.
3) Администратор — пользователь, обладающий возможностями участника и судьи, а также возможностью редактировать и удалять профили судей. Роль требует авторизации.

На рисунке 1 представлена диаграмма использования приложения.

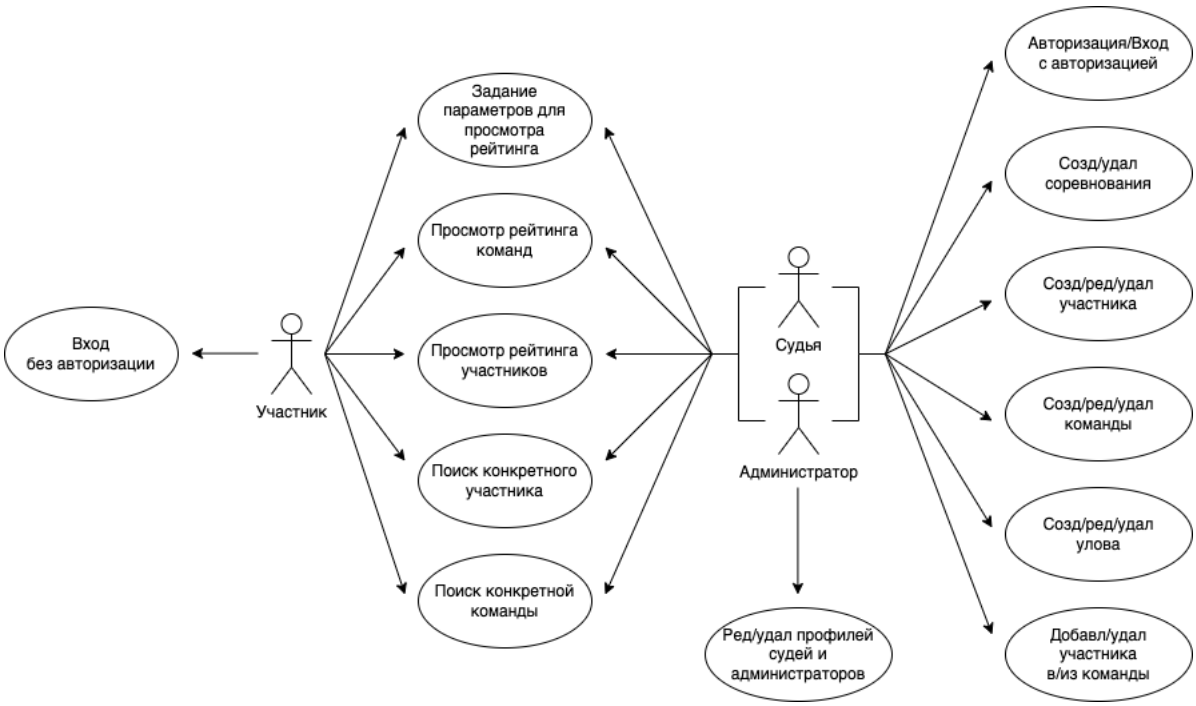

Рисунок 1 – Диаграмма использования приложения

1.3 Формализация данных

База данных должна хранить информацию о следующих сущностях:

— соревнование;

— команда;

— участник;

— этап;

— улов;

— пользователь;

— авторизация.

В таблице 1 представлены сущности и сведения о них.

Таблица 1 – Сущности и сведения о них

Сущность	**Сведения**
Участник	Фамилия, имя, отчество, команда, город, дата рождения, очки
Команда	Название, соревнования, очки
Соревнования	Название, команды
Этап	Название этапа, участник, соревнование, очки
Улов	Рыба, этап, вес, очки
Пользователь	Роль, авторизация
Авторизация	Логин, пароль

1.4 Анализ моделей баз данных

Одним из ключевых этапов разработки приложения является выбор модели базы данных, которая будет использоваться для хранения и управления данными. Существует множество различных типов баз данных совместимых с iOS, каждый из которых имеет свои преимущества и недостатки. Далее будут рассмотрены основные типы баз данных, проведен анализ существующих систем управления базами данных и выбрана наиболее подходящую модель для

реализации приложения. Кроме того, будет определен пользователи системы, формализованы данные и представлены сущности, чтобы более точно определить требования к приложению и разработать его структуру.

1.4.1 Реляционная модель

Реляционная модель базы данных — это способ организации данных, в котором информация хранится в виде таблиц, состоящих из строк и столбцов. Каждая таблица представляет отдельную сущность или объект, а каждая строка в таблице — это конкретный экземпляр этой сущности, содержащий информацию о ней. Основным принципом реляционной модели является использование ключей для связывания таблиц между собой. Ключ — это уникальный идентификатор, который позволяет однозначно идентифицировать каждую строку в таблице. Используя ключи, можно создавать связи между таблицами, чтобы получать более сложные данные.

Реляционная модель базы данных обладает рядом преимуществ. Во–первых, она позволяет хранить и организовывать большие объемы данных. Во–вторых, она обеспечивает гибкость и возможность быстрого изменения структуры данных. В–третьих, реляционная модель обеспечивает высокую степень надежности и целостности данных.

Однако, реляционная модель базы данных также имеет и некоторые недостатки. Например, она не всегда эффективна при работе с большими объемами данных, так как может требоваться много времени на выполнение сложных запросов. Также реляционная модель не всегда подходит для хранения и обработки неструктурированных данных, таких как тексты и мультимедийные файлы.

Наиболее распространенным примером для реляционных баз данных iOS является SQLite [4], использующий Core Data [5] в качестве среды сохранения.

1.4.2 Модель ключ–значение

Модель базы данных ключ-значение (key–value) — это способ организации данных, в котором информация хранится в виде пар ”ключ–значение”. Ключ — это уникальный идентификатор, который используется для доступа

к соответствующему значению. Значение может быть любым типом данных, включая строки, числа, объекты и так далее. Данные хранятся в памяти или на диске в формате, который позволяет быстро и легко извлекать нужную информацию. Эта модель используется в различных задачах, таких как кэширование данных, хранение сессий пользователей, управление настройками приложения и так далее.

Однако, модель ключ–значение имеет и некоторые ограничения. Например, она не подходит для сложных запросов и аналитики данных, которые требуют более сложной структуры хранения информации. Также, в случае большого объема данных, может возникнуть проблема с производительностью при обработке и поиске нужной информации.

Чаще всего модель используется в iOS для хранения небольших фрагментов информации, доступных при запуске приложения.

1.4.3 Объектно–ориентированная модель

Объектно–ориентированная модель базы данных — это способ организации данных, в котором информация представляется в виде объектов, имеющих свойства и методы. Каждый объект представляет отдельную сущность или объект, а свойства объекта содержат информацию о нем.

Модель обладает рядом преимуществ. Во–первых, она позволяет более естественно описывать данные и их взаимосвязь. Во–вторых, обеспечивает более гибкую и быструю обработку данных. В-третьих, позволяет управлять данными на более высоком уровне абстракции.

Однако, объектно–ориентированная модель также имеет и некоторые недостатки. Например, она может быть менее надежной и целостной, если не будет правильно спроектирована.

Примером объектно–ориентированная базы данных является Realm [6].

1.5 Обзор существующих баз данных

При создании приложения, независимо от его назначения и функциональности, одним из самых важных этапов является выбор модели базы данных. От

правильного выбора зависит не только эффективность работы приложения, но и его масштабируемость, безопасность и надежность. Существует множество различных типов баз данных для iOS, каждый из которых имеет свои преимущества и недостатки. Рассмотрим основные типы баз данных, проведем анализ существующих систем управления базами данных и выберем наиболее подходящую модель для реализации приложения. Рассмотрим основные варианты хранилищ для iOS приложений.

1.5.1 UserDefaults

UserDefaults [7]— это интерфейс для доступа к базе данных ключ–значение для хранения значений пользователя по умолчанию при запуске приложения. База может сохранять такие значения, как язык по умолчанию или скорость воспроизведения мультимедиа, сохраняя их неизменными для каждого пользователя до тех пор, пока установлено приложение. Однако UserDefaults имеет ограниченный вариант использования — количество поддерживаемых типов данных мало. Если хранится слишком много данных, что приводит к увеличению свободного места на устройстве, кэш в памяти может замедлить работу приложения. Она не предназначена для больших объемов данных.

1.5.2 Firebase и Firestore

Firebase [8] — это платформа для создания приложений, которая обладает рядом технологий, которые можно комбинировать и подбирать в соответствии с требованиями приложения. Одна из них — Firestore [9]. Это облачная NoSQL [10] база данных, которая предлагает высокую производительность и масштабируемость. Firestore разработана для использования в приложениях, которые работают в режиме реального времени и используют синхронизацию данных между устройствами.

База имеет простой и удобный API, который позволяет легко взаимодействовать с базой данных из iOS приложений. Она поддерживает работу с данными в режиме оффлайн, что позволяет приложению продолжать работу и сохранять изменения, даже если нет подключения к интернету.

К минусам базы можно отнести то, что размер приложения увеличится из–за добавления сторонней библиотеки, а также Firebase имеет более низкую производительность из–за ограничений запросов. Сложные запросы приходится разгружать для обработки облачными функциями, что создает дополнительную нагрузку на разработчиков по управлению сервисом.

1.5.3 SQLite

SQLite [4] — это встроенная в процесс библиотека, которая реализует автономный, бессерверный транзакционный компонент SQL [11]. Код для SQLite находится в общественном достоянии и, таким образом, свободен для использования в любых целях. SQLite не требует установки дополнительного сервера баз данных, что упрощает развертывание приложения на устройствах пользователей. Однако база не предоставляет никаких возможностей для размещения в облаке, а также SQLite не хватает масштабируемости из–за ограничений реляционных баз данных. Часто для этого также требуется ORM (например, Core Data [5]).

1.5.4 Realm

Realm [6] — это реактивная, объектно–ориентированная, кроссплатформенная, NoSQL [10] мобильная база данных.

Благодаря объектно–ориентированной модели данных она работает с объектами идиоматически, поэтому нет необходимости в ORM. Realm может легко использоваться для сложных вариантов запросов. Данные сохраняются на устройстве, не снижая продуктивности разработчиков и не замедляя производительность приложений [12], но также имеется поддержка синхронизации с MongoDB Atlas — мультиоблачной службой баз данных [13] — и, соответственно, синхронизации устройств.

Как и в случае в Firebase, размер приложения увеличивается из–за необходимости добавления сторонней библиотеки.

1.6 Выбор базы данных и СУБД

Проектируемое мобильное приложение должно предоставлять возможность хранения большого объема данных, а также потенциальную возможность синхронизации данных на разных устройствах. Также стоит отметить, что приложения для iOS подразумевает использование Swift [14] — мультипарадигмального языка программирования, в числе поддерживаемых моделей которого — объектно–ориентированное программирование (ООП).

Учитывая все вышеперечисленные факторы, а также плюсы и минусы рассматриваемых баз данных, выбор пал на Realm — объектно–ориентированную базу, предоставляющую механизм синхронизации с мультиоблачной службой. Хранилище предоставляет Realm SDK — программный инструментарий для работы с базой данных в мобильных приложениях. Realm SDK не является самостоятельной СУБД, но использует встроенную базу данных Realm, которая предоставляет функциональность СУБД [15].

1.7 Вывод из раздела

В данном разделе были выделены ролевые модели системы, конкретизированы данные и их связь между собой, построены соответствующие диаграммы.Также были рассмотрены преимущества и недостатки различных типов баз данных и систем управления базами данных, что позволило выбрать наиболее подходящие для реализации приложения. В результате проведенного анализа была выбрана база данных Realm. Проведенный анализ помог определить требования к приложению и разработать его структуру, что является важным этапом в создании качественного и надежного приложения.

2 Конструкторский раздел

Важным этапом создания качественного и надежного приложения является проектирование базы данных. В данном разделе курсовой работы будет рассмотрено проектирование базы данных для разрабатываемого приложения. Будут выделены основные этапы проектирования, рассмотрены конкретные действия ролевой модели, спроектирован триггер и описаны основы проектирования приложения. Целью данного раздела является создание эффективной и масштабируемой базы данных, которая обеспечит стабильную работу приложения и удовлетворит потребности пользователей.

2.1 Проектирование базы данных

На основе выделенных ранее сущностей спроектированы следующие объекты базы данных.

1) Participant — содержит информацию об участнике и имеет следующие поля:
   - participantId — уникальный идентификатор участника;
   - participantLastname — фамилия участника;
   - participantFirstname — имя участника;
   - participantPatronymic — отчество участника;
   - participantTeam — команда участника;
   - participantCity — город проживания участника;
   - participantBirthday — дата рождения участника;
   - participantScore — очки участника.
2) Team — содержит информацию о команде и имеет следующие поля:
   - teamId — уникальный идентификатор команды;
   - teamName — название команды;
   - teamCompetitions — соревнования, в которых принимает участие команда;
   - teamScore — очки команды.

3) Competition — содержит информацию о соревновании и имеет следующие поля:
   - competitionId — уникальный идентификатор соревнования;
   - competitionName — название соревнования;
   - competitionTeams — команды, участвующие в соревнованиях.
4) Step — содержит информацию об этапе соревнования для участника:
   - stepId — уникальный идентификатор этапа;
   - stepName — название этапа;
   - stepParticipant — участник, которому принадлежит этап;
   - stepCompetition — соревнование, которому принадлежит этап;
   - lootScore — очки за зачет.
5) Loot — содержит информацию об улове участника:
   - lootId — уникальный идентификатор улова;
   - lootFish — название рыбы;
   - lootWeight — вес рыб;
   - lootStep — зачет, к которому относится улов;
   - lootScore — очки за рыбу.
6) User — содержит информацию о пользователе и имеет следующие поля:
   - userId — уникальный идентификатор пользователя;
   - userAuthorization — авторизация пользователя;
   - userRole — роль пользователя.
7) Authorization — содержит информацию об авторизации и имеет следующие поля:
   - authorizationId — уникальный идентификатор авторизации;
   - authorizationLogin — логин;
   - authorizationPassword — пароль.

Соответствующая диаграмма по описанным выше данным представлена на рисунке 2 .

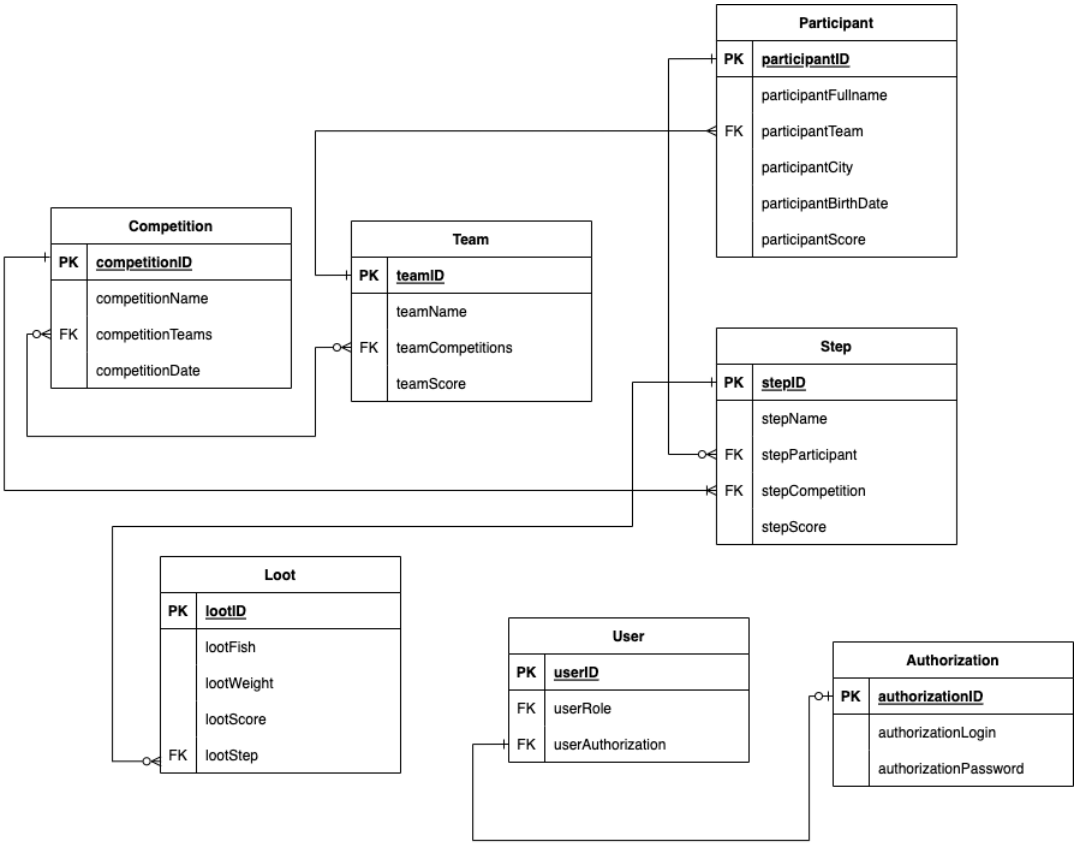

Рисунок 2 – ER–диаграмма

2.2 Ролевая модель

Ролевая модель предполагает наличие трех ролей: участника, судьи и администратора. Стоит отметить, что судья обладает всеми правами участника, а администратор — всеми правами судьи.

Для успешной реализации поставленной задачи программа должна предоставлять следующие возможности:

— просмотр профилей участников;

— просмотр профилей команд;

— просмотр личного и командного рейтингов;

— просмотр личного и командного рейтингов по этапам;

— сортировка личного и командного рейтингов по убыванию/по возрастанию;

— сортировка личного и командного рейтингов по этапам по убыванию/по возрастанию.

Дополнительные возможности судьи:

— регистрация;
— авторизация;
— создание/удаление соревнования;
— создание/редактирование/удаление команды;
— создание/редактирование/удаление участника;
— создание/редактирование/удаление улова;
— добавление улова в этап;
— добавление участника в команду;
— добавление команды в соревнование.

Дополнительные возможности администратора:

— редактирование/удаление профилей судей;
— создание/редактирование/удаление профилей администраторов.

2.3 Триггер

Триггер — это хранимая процедура особого типа, которую пользователь не вызывает непосредственно, а исполнение которой обусловлено действием по изменению данных: добавлением, модификацией, удалением строки в заданной таблице.

В каждой из сущностей, таких как улов, этап, участник и команда, представлено поле «очки». Когда рыба взвешена и очки за нее необходимо добавить в этап участника, присвоить участнику и его команде, нужно в каждой сущности, связанной с уловом, обновить значение поля.

Связи соответствующих сущностей представлены на рисунке 3.

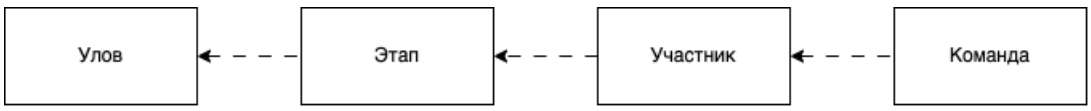

Рисунок 3 – Диаграмма связей сущностей при обновлении очков

Для процедуры перерасчета очков было бы удобно создать триггер, который автоматически обновляет значение соответствующего поля в описанных

четырех сущностях, при добавлении, редактировании и удалении улова.

Схема алгоритма триггера представлена на рисунке 3.

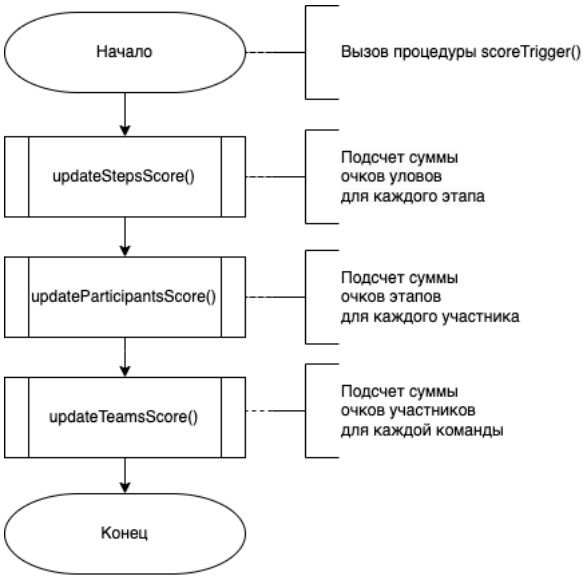

Рисунок 4 – Триггер для обновления очков

2.4 Проектирование приложения

При разработке приложение для iOS может быть разделено на 3 этапа в соответствии с принципами «чистой архитектуры» [16]:

1) слой доступа к данным — подключение к базе данных, отправка запросов, получение информации из базы данных;
2) слой пользовательского интерфейса (UI) — взаимодействие пользователя с программой;
3) слой бизнес логики — взаимодействия между слоем доступа к данным и приложением.

2.5 Вывод из раздела

В результате проектирования базы данных для разрабатываемого приложения были выявлены и описаны основные этапы проектирования, которые позволили создать базу данных, удовлетворяющую требованиям. Были спроектированы сущности базы данных и их связи, ролевая модель и триггер, что обеспечило стабильную работу приложения и удовлетворило потребности пользователей. Были также формализованы требования к программе и описаны эта-

пы разработки приложения, что позволит эффективно продолжить работу над проектом и достичь поставленных целей.

3 Технологический раздел

В рамках конструкторского раздела курсовой работы будут выбраны средства реализации, будет представлено создание базы данных, ролей и выделение им прав, создание триггера, а также разработан пользовательский интерфейс. Важным этапом является тестирование функционала базы данных и программы, которое позволит оценить эффективность и соответствие требованиям. В данном разделе будет также описан метод тестирования функционала базы данных.

3.1 Средства реализации

В качестве языка программирования выбран Swift [14] — мультипарадигмальный язык программирования, разработанный компанией Apple [3] для создания приложений под iOS, macOS, watchOS и tvOS.

В качестве среды разработки используется XCode [17] — интегрированная среда разработки для создания приложений для операционных систем iOS, macOS, watchOS и tvOS. Xcode предоставляет инструменты для написания кода на языках Swift и Objective-C, отладки приложений, создания пользовательских интерфейсов, тестирования и сборки приложений. Xcode также содержит симуляторы устройств для тестирования приложений без фактического устройства.

3.2 Создание базы данных

В процессе проектирования базы данных были учтены все требования к хранению и управлению информацией в приложении. В конструкторском разделе были выбраны подходящие средства реализации для создания базы данных, а затем была спроектирована сама база данных. В соответствии с выбранной СУБД и спроектированной базой данных было осуществлено создание ее сущностей. Программный код представлен в приложении А, листинг 2.

3.3 Создание ролей и выделение им прав

Создание ролевой модели позволило определить права доступа к данным для различных пользователей системы. В конструкторском разделе была выде-

лена ролевая модель, которая определяет права доступа к данным для различных пользователей системы.

В модели были выделены следующие роли:

— участник;

— судья;

— администратор.

Соответствующий этой ролевой модели код выделения прав представлен в листингах 3 — 9.

3.4 Создание триггера

Для обеспечения более гибкой работы с базой данных был создан триггер, который автоматически выполняет определенные действия при изменении определенных данных в базе. Это позволяет ускорить обработку данных и снизить нагрузку на систему. Для создания триггера, обновляющего значения поля score в связанных с уловом сущностях, была написана процедура scoreTrigger, а также вспомогательные функции updateStepScore, updateParticipantScore, updateTeam
В листингах 10 — 13 приложения А представлен соответствующий программный код.

3.5 Методы тестирования

Для тестирования модулей приложения, в том числе уровня доступа к базе данных, использовался XCTest [18] — фреймворк для модульного тестирования приложений, написанных на языках Swift и Objective-C, в Xcode. Он предоставляет разработчикам возможность создавать и запускать автоматические тесты, чтобы убедиться в корректности работы функций отдельных частей приложения. XCTest включает в себя множество методов для создания и выполнения проверок, а также для анализа результатов испытаний.

Так, например, в листинге 1 представлен код unit–теста для проверки корректности работы функции добавления сущности в базу данных.

Листинг 1 – Код unit–теста

```
1 func testCreateLoot() throws {
2     let loot = Loot(id: "6442852b2b74d595cb4f4764", fish: "Щука", weight:
    500, score: 1000)
3     var createdLoot: Loot?
4
5     XCTAssertNoThrow(createdLoot = try lootRepository.createLoot(loot: loot))
6     XCTAssertEqual(createdLoot, loot)
7 }
```

3.6 Пользовательский интерфейс

Важным элементом разработки приложения является пользовательский интерфейс. Был разработан удобный и интуитивно понятный интерфейс, который обеспечивает удобный доступ к данным и удовлетворяет потребности пользователей. Пользователи могут легко находить необходимую информацию и выполнять нужные действия. Для взаимодействия с базой данных был разработан интерфейс в виде мобильного приложения для iOS, представленный на рисунках 6 — 12 приложения Б.

3.7 Вывод из раздела

Одним из ключевых элементов данного раздела является создание базы данных. Для этого ранее были выбраны подходящие средства реализации, в рассмотренном же разделе была создана сама базы данных, которая учитывает все требования к хранению и управлению информацией в приложении. Кроме того, была выделена ролевая модель, которая определяет права доступа к данным для различных пользователей системы. Для обеспечения более гибкой работы с базой данных был создан триггер, который автоматически выполняет определенные действия при изменении определенных данных в базе. Это позволяет ускорить обработку данных и снизить нагрузку на систему. Также был представлен пользовательский интерфейс.

4 Исследовательский раздел

В данном разделе будет произведена постановка исследования. В соответствии с формулировкой исследования будут проведены замеры времени на тестовой базе данных, а также произведен анализ полученных данных. В данном разделе будут представлены результаты исследования. Полученные результаты позволят сделать выводы о соответствии базы данных требованиям к приложению и ее эффективности в работе.

4.1 Цель исследования

Целью исследования является оценка изменения времени в зависимости от количества участников соревнований выполнения функции добавления улова спортсмену, а также функции получения списка спортсменов — участников определенных соревнований. Для достижения данной цели были проведены эксперименты на тестовой базе данных, которая была спроектирована с учетом требований к приложению. В данном разделе будут представлены результаты исследования, а также анализ полученных данных, который позволит сделать выводы о эффективности разработанной базы данных и ее соответствии требованиям к приложению.

4.2 Описание исследования

Для добавления улова используется функция createLoot, представленная в листинге 14 приложения А. Для получения списка участников соревнований используется функция getParticipantByCompetition, представленная в листинге 15.

В таблице 2 представлены временные замеры при наличии 10, 25, 50, 100, 250, 500, 750 и 1000 участников, каждое значение получено усреднением результатов по 10 замерам. По данной таблице построен график (рисунок 5), отображающий исследуемые зависимости.

Из результатов измерений можно сделать вывод, что при увеличении количества участников скорость работы функции получения списка участников

Таблица 2 – Замеры времени (в секундах)

Количество участников	**Время для функции получения списка спортсменов**	**Время для функции добавления улова**
10	0.131317	0.010210
25	0.175450	0.058736
50	0.411645	0.047920
100	0.877204	0.107326
250	2.205371	0.174752
500	6.079674	0.181720
750	12.083022	0.2716641
1000	18.278156	0.430814

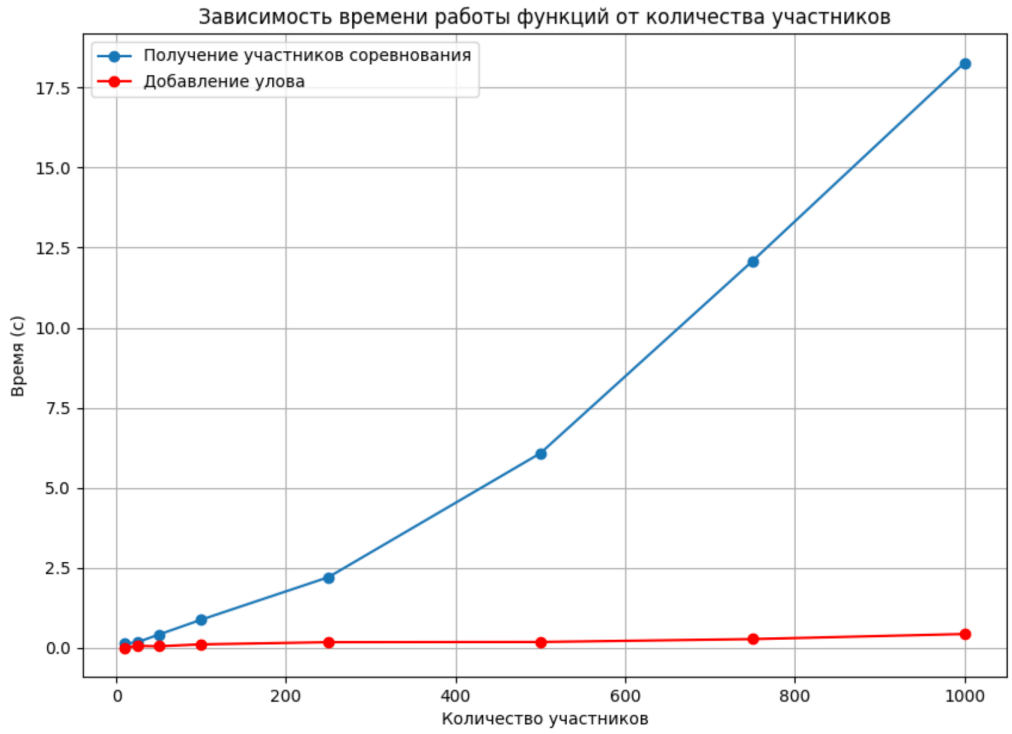

Рисунок 5 – Зависимость времени работы функций от количества участников

соревнования изменяется сильнее, чем скорость работы функции добавления улова. Так, при изменении количества участников в 100 раз (от 10 до 1000), время, затрачиваемое на работу функции получения участников, увеличивается в 139 раз, а время, затрачиваемое на работу функции создания улова, увеличивается лишь в 42 раза.

4.3 Вывод из раздела

Исследование показало, что время, затрачиваемое на работу функции получения всех участников соревнования, увеличивается заметно быстрее, чем время, затрачиваемое на работу функции создания улова. Очевидно, что выборка из всей базы — формирование списка — будет требовать большее количество времени, чем фиксирование улова, что подтвердило исследование. Это показывает, что при одинаковом количестве спортсменов разные функции могут требовать неодинакового (иногда сильно разнящегося) времени для работы.

ЗАКЛЮЧЕНИЕ

В ходе выполнения данной работы были выполнены следующие задачи:

— формализована задача и определен необходимый функционал;
— описана структура объектов базы данных, а также сделан выбор СУБД для ее хранения и взаимодействия;
— спроектирован и разработан триггер, осуществляющий перерасчёт очков этапов, участников и команд при добавлении улова;
— спроектирован и разработан интерфейс взаимодействия с базой данных;
— проведено исследование времени работы операции добавления улова, а также функции получения списка спортсменов соревнования в зависимости от количества участников.

Была достигнута поставленная цель: спроектирована и разработана база данных для проведения соревнований Российской Федерации Подводного Рыболовства, интерфейсом для которой стало приложения для iOS. Необходимый функционал был реализован. Также в ходе исследования было выявлено, что при изменении количества участников в 100 раз (от 10 до 1000), время, затрачиваемое на работу функции получения списка участников, увеличивается в 139 раз, а время, затрачиваемое на работу функции создания улова, увеличивается лишь в 42 раза.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. РФПР | Российская Федерация Подводного Рыболовства [Электронный ресурс]. — Режим доступа: http://rfpr.su/, свободный (дата обращения: 16.04.2023)

2. iOS [Электронный ресурс]. — Режим доступа: https://www.apple.com/ios/ios-16/, свободный (дата обращения: 18.04.2023)

3. Apple [Электронный ресурс]. — Режим доступа: https://www.apple.com/, свободный (дата обращения: 18.04.2023)

4. About SQLite [Электронный ресурс]. — Режим доступа: https://sqlite.org/about.html, свободный (дата обращения: 16.04.2023)

5. Core Data | Apple Developer Documentation [Электронный ресурс]. — Режим доступа: https://developer.apple.com/documentation/coredata, свободный (дата обращения: 16.04.2023)

6. Realm Database [Электронный ресурс]. — Режим доступа: https://www.mongodb.com/docs/realm/sdk/swift/realm-database/, свободный (дата обращения: 30.04.2023)

7. UserDefaults | Apple Developer Documentation [Электронный ресурс]. — Режим доступа: https://developer.apple.com/documentation/foundation/userdefaults, свободный (дата обращения: 30.04.2023)

8. Firebase [Электронный ресурс]. — Режим доступа: https://firebase.google.com/index.html, свободный (дата обращения: 30.04.2023)

9. Cloud Firestore [Электронный ресурс]. — Режим доступа: https://firebase.google.com/products/firestore, свободный (дата обращения: 30.04.2023)

10. Что такое NoSQL? | Amazon AWS [Электронный ресурс]. — Режим доступа: https://aws.amazon.com/ru/nosql, свободный (дата обращения: 16.04.2023)

11. SQL — Structured Query Language [Электронный ресурс]. — Режим доступа: https://docs.microsoft.com/en-us/sql/odbc/reference/structured-query-language-sql?view=sql-server-ver15, свободный (дата обращения: 16.04.2023)

12. iOS Databases | Choosing the Best iOS Database [Электронный ресурс]. — Режим доступа: https://realm.io/best-ios-database/, свободный (дата обращения: 01.05.2023)

13. What is MongoDB Atlas? — MongoDB Atlas [Электронный ресурс]. — Режим доступа: https://www.mongodb.com/docs/atlas/, свободный (дата обращения: 16.04.2023)

14. Swift — Apple Developer [Электронный ресурс]. — Режим доступа: https://developer.apple.com/swift/, свободный (дата обращения: 01.05.2023)

15. Что такое СУБД — RU—CENTER [Электронный ресурс]. — Режим доступа: https://www.nic.ru/help/chto-takoe-subd_8580.html, свободный (дата обращения: 16.04.2023)

16. Мартин Р. Чистая архитектура. Искусство разработки программного обеспечения. – СПб.: Питер, 2018. – 352 с.

17. XCode | Apple Developer Documentation [Электронный ресурс]. — Режим доступа: https://developer.apple.com/documentation/xcode?changes=la, свободный (дата обращения: 02.05.2023)

18. XCTest | Apple Developer Documentation [Электронный ресурс]. —

Режим доступа: https://developer.apple.com/documentation/xctest, свободный (дата обращения: 03.05.2023)

ПРИЛОЖЕНИЕ А

Листинг 2 – Создание спроектированной базы данных

```
class LootRealm: Object {
    @Persisted(primaryKey: true) var _id: ObjectId
    @Persisted var fish: String
    @Persisted var weight: Int
    @Persisted var step: StepRealm?
    @Persisted var score: Int

    convenience init(id: String?, fish: String, weight: Int, step: StepRealm
    ?, score: Int) throws {
        self.init()
        self._id = ObjectId.generate()
        if let id = id { self._id = try! ObjectId.init(string: id) }
        self.fish = fish
        self.weight = weight
        self.step = step
        self.score = score
    }
}

class StepRealm: Object {
    @Persisted(primaryKey: true) var _id: ObjectId
    @Persisted var name: String
    @Persisted var participant: ParticipantRealm?
    @Persisted var competition: CompetitionRealm?
    @Persisted var score: Int

    convenience init(id: String?, name: String, participant: ParticipantRealm
    ?, competition: CompetitionRealm?, score: Int) {
                self.init()
        self._id = ObjectId.generate()
        if let id = id { self._id = try! ObjectId.init(string: id) }
        self.name = name
        self.participant = participant
        self.competition = competition
        self.score = score
    }
}
```

```
class CompetitionRealm: Object {.
    @Persisted(primaryKey: true) var _id: ObjectId.
    @Persisted var name: String.
    @Persisted var teams: List<TeamRealm>.

    convenience init(id: String?, name: String, teams: List<TeamRealm>) {.
        self.init().

        self._id = ObjectId.generate().
        if let id = id { self._id = try! ObjectId.init(string: id) }.
        self.name = name.
        self.teams = teams.
    }.
}.

class TeamRealm: Object {.
    @Persisted(primaryKey: true) var _id: ObjectId.
    @Persisted var name: String.
    @Persisted var competitions: List<CompetitionRealm>.
    @Persisted var score: Int.

    convenience init(id: String?, name: String, competitions: List<
    CompetitionRealm>, score: Int) {.
        self.init().

        self._id = ObjectId.generate().
        if let id = id { self._id = try! ObjectId.init(string: id) }.
        self.name = name.
        self.competitions = competitions.
        self.score = score.
    }.
}.

class ParticipantRealm: Object {.
    @Persisted(primaryKey: true) var _id: ObjectId.
    @Persisted var lastName: String.
    @Persisted var firstName: String.
    @Persisted var patronymic: String?.
    @Persisted var team: TeamRealm?.
    @Persisted var city: String.
    @Persisted var birthday: Date.
```

```
76      @Persisted var score: Int
77
78      convenience init(id: String?, lastName: String, firstName: String,
        patronymic: String?, team: TeamRealm?, city: String, birthday: Date, score
        : Int) {
79          self.init()
80
81          self._id = ObjectId.generate()
82          if let id = id { self._id = try! ObjectId.init(string: id) }
83          self.lastName = lastName
84          self.firstName = firstName
85          self.team = team
86          self.patronymic = patronymic
87          self.city = city
88          self.birthday = birthday
89          self.score = score
90      }
91  }
92
93  class AuthorizationRealm: Object {
94      @Persisted(primaryKey: true) var _id: ObjectId
95      @Persisted var login: String
96      @Persisted var password: String
97
98      convenience init(id: String?, login: String, password: String) throws {
99          self.init()
100
101         self._id = ObjectId.generate()
102         if let id = id { self._id = try! ObjectId.init(string: id) }
103         self.login = login
104         self.password = password
105     }
106 }
107
108 class UserRealm: Object {
109     @Persisted(primaryKey: true) var _id: ObjectId
110     @Persisted var role: String
111     @Persisted var authorization: AuthorizationRealm?
112
113     convenience init(id: String?, role: String, authorization:
        AuthorizationRealm?) throws {
```

```
114         self.init()
115
116         self._id = ObjectId.generate()
117         if let id = id { self._id = try! ObjectId.init(string: id) }
118         self.role = role
119         self.authorization = authorization
120         }
121 }
```

Листинг 3 – Выделение прав в соответствии с ролевой моделью

```
1 extension AuthorizationRepository {
2     func getRight(_ user: User?, _ action: Action) -> Bool {
3         guard let user = user else { return false }
4
5         switch user.role {
6
7         case .participant:
8             switch action {
9             case .create:
10                return false
11            case .read:
12                return false
13            case .update:
14                return false
15            case .delete:
16                return false
17            }
18
19        case .referee:
20            switch action {
21            case .create:
22                return false
23            case .read:
24                return true
25            case .update:
26                return false
27            case .delete:
28                return false
29            }
30
```

```
31          case .admin:
32              switch action {
33              case .create:
34                  return true
35              case .read:
36                  return true
37              case .update:
38                  return true
39              case .delete:
40                  return true
41              }
42          }
43      }
44  }
```

Листинг 4 – Выделение прав в соответствии с ролевой моделью

```
1  extension UserRepository {
2      func getRight(_ user: User?, _ action: Action) -> Bool {
3          guard let user = user else { return false }
4
5          switch user.role {
6
7          case .participant:
8              switch action {
9              case .create:
10                 return false
11             case .read:
12                 return false
13             case .update:
14                 return false
15             case .delete:
16                 return false
17             }
18
19         case .referee:
20             switch action {
21             case .create:
22                 return false
23             case .read:
24                 return true
```

```
25              case .update:
26                  return false
27              case .delete:
28                  return false
29              }
30
31          case .admin:
32              switch action {
33              case .create:
34                  return true
35              case .read:
36                  return true
37              case .update:
38                  return true
39              case .delete:
40                  return true
41              }
42          }
43      }
44  }
```

Листинг 5 – Выделение прав в соответствии с ролевой моделью

```
1  extension LootRepository {
2      func getRight(_ user: User?, _ action: Action) -> Bool {
3          guard let user = user else { return false }
4
5          switch user.role {
6
7          case .participant:
8              switch action {
9              case .create:
10                 return false
11             case .read:
12                 return true
13             case .update:
14                 return false
15             case .delete:
16                 return false
17             }
18
```

```
        case .referee:
            switch action {
            case .create:
                return true
            case .read:
                return true
            case .update:
                return true
            case .delete:
                return true
            }

        case .admin:
            switch action {
            case .create:
                return true
            case .read:
                return true
            case .update:
                return true
            case .delete:
                return true
            }
        }
    }
}
```

Листинг 6 – Выделение прав в соответствии с ролевой моделью

```
extension CompetitionRepository {
    func getRight(_ user: User?, _ action: Action) -> Bool {
        guard let user = user else { return false }

        switch user.role {

        case .participant:
            switch action {
            case .create:
                return false
            case .read:
                return true
```

```
13              case .update:
14                  return false
15              case .delete:
16                  return false
17              }
18
19          case .referee:
20              switch action {
21              case .create:
22                  return true
23              case .read:
24                  return true
25              case .update:
26                  return true
27              case .delete:
28                  return true
29              }
30
31          case .admin:
32
33              switch action {
34              case .create:
35                  return true
36              case .read:
37                  return true
38              case .update:
39                  return true
40              case .delete:
41                  return true
42              }
43          }
44      }
45  }
```

Листинг 7 – Выделение прав в соответствии с ролевой моделью

```
1  extension TeamRepository {
2      func getRight(_ user: User?, _ action: Action) -> Bool {
3          guard let user = user else { return false }
4
5          switch user.role {
```

```
6
7           case .participant:
8               switch action {
9               case .create:
10                  return false
11              case .read:
12                  return true
13              case .update:
14                  return false
15              case .delete:
16                  return false
17              }
18
19          case .referee:
20              switch action {
21              case .create:
22                  return true
23              case .read:
24                  return true
25              case .update:
26                  return true
27              case .delete:
28                  return true
29              }
30
31          case .admin:
32              switch action {
33              case .create:
34                  return true
35              case .read:
36                  return true
37              case .update:
38                  return true
39              case .delete:
40                  return true
41              }
42          }
43      }
44  }
```

Листинг 8 – Выделение прав в соответствии с ролевой моделью

```
extension ParticipantRepository {
    func getRight(_ user: User?, _ action: Action) -> Bool {
        guard let user = user else { return false }

        switch user.role {

        case .participant:
            switch action {
            case .create:
                return false
            case .read:
                return true
            case .update:
                return false
            case .delete:
                return false
            }

        case .referee:
            switch action {
            case .create:
                return true
            case .read:
                return true
            case .update:
                return true
            case .delete:
                return true
            }

        case .admin:
            switch action {
            case .create:
                return true
            case .read:
                return true
            case .update:
                return true
            case .delete:
```

```
                return true
            }
        }
    }
}
```

Листинг 9 – Выделение прав в соответствии с ролевой моделью

```
extension StepRepository {
    func getRight(_ user: User?, _ action: Action) -> Bool {
        guard let user = user else { return false }

        switch user.role {

        case .participant:
            switch action {
            case .create:
                return false
            case .read:
                return true
            case .update:
                return false
            case .delete:
                return false
            }

        case .referee:
            switch action {
            case .create:
                return true
            case .read:
                return true
            case .update:
                return true
            case .delete:
                return true
            }
    case .admin:
        switch action {
        case .create:
            return true
```

```
34          case .read:
35              return true
36          case .update:
37              return true
38          case .delete:
39              return true
40          }
41      }
42  }
```

Листинг 10 – Код функции триггера — scoreTrigger

```
1  func scoreTrigger(_ loot: LootRealm?) throws {
2      let stepsRealm = realm.objects(StepRealm.self)
3
4      for step in stepsRealm {
5          do {
6               try updateStepScore(step)
7          } catch {
8              throw DatabaseError.updateError
9          }
10     }
11 }
```

Листинг 11 – Код вспомогательной функции updateStepScore

```
1  func updateStepScore(_ step: StepRealm) throws {
2      let lootsRealm = realm.objects(LootRealm.self)
3      var newScore = 0
4      for loot in lootsRealm {
5          if loot.step == step {
6              newScore += loot.score
7          }
8      }
9      let id = "\(step._id)"
10     let newStep = StepRealm(id: id, name: step.name, participant: step.
       participant, competition: step.competition, score: newScore)
11     do {
12         try realm.write {
13             realm.add(newStep, update: .modified)
14         }
```

```
15      } catch {.
16          throw DatabaseError.updateError.
17      }.
18
19      let participant = participantRepository.getParticipantByStep(newStep).
20      try? updateParticipantScore(participant).
21  }.
```

Листинг 12 – Код вспомогательной функции updateParticipantScore

```
1  func updateParticipantScore(_ participant: ParticipantRealm) throws {.
2      let stepsRealm = realm.objects(StepRealm.self).
3
4      var newScore = 0.
5      for step in stepsRealm {.
6          if step.participant == participant {.
7              newScore += step.score.
8          }.
9      }.
10
11     let id = "\(participant._id)".
12     let newParticipant = ParticipantRealm(id: id, lastName: participant.
       lastName, firstName: participant.firstName, patronymic: participant.
       patronymic, team: participant.team, city: participant.city, birthday:
       participant.birthday, score: newScore).
13
14     do {.
15         try realm.write {.
16             realm.add(newParticipant, update: .modified).
17         }.
18     } catch {.
19         throw DatabaseError.updateError.
20     }.
21
22     let team = teamRepository.getTeamByParticipant(newParticipant).
23     try? updateTeamScore(team).
24 }.
```

Листинг 13 – Код вспомогательной функции updateTeamScore

```
1  func updateTeamScore(_ team: TeamRealm) throws {.
```

```
    let participantsRealm = realm.objects(ParticipantRealm.self).

    var newScore = 0.
    for participant in participantsRealm {.
        if participant.team == team {.
            newScore += participant.score.
        }.
    }.

    let id = "\(team._id)".
    let newTeam = TeamRealm(id: id, name: team.name, competitions: team.
    competitions, score: newScore).

    do {.
        try realm.write {.
            realm.add(newTeam, update: .modified).
        }.
    } catch {.
        throw DatabaseError.updateError.
    }.
}.
```

Листинг 14 – Код функции createLoot

```
func createLoot(loot: Loot) throws -> Loot? {.
    let realmLoot: LootRealm.
    do {.
        realmLoot = try loot.convertLootToRealm(realm).
    } catch {.
        throw DatabaseError.addError.
    }.
            do {.
        try realm.write {.
            realm.add(realmLoot).
        }.
    } catch {.
        throw DatabaseError.addError.
    }.

    let createdLoot = try getLoot(id: "\(realmLoot._id)").
    do {.
```

```
18          let loot = try createdLoot?.convertLootToRealm(realm).
19          try triggerLootToStep(loot).
20      } catch {.
21          throw DatabaseError.triggerError.
22      }.
23      return createdLoot.
24  }.
```

Листинг 15 – Код функции getParticipantByCompetition

```
25  func getParticipantByCompetition(participant: Participant, competition:
        Competition, stepName: StepsName?) throws -> Participant? {.
26      guard let participant = try getParticipant(id: participant.id!) else {.
27          throw ParameterError.funcParameterError.
28      }.
29
30      let participantRealm = try participant.convertParticipantToRealm(realm).
31      let competitionRealm = try competition.convertCompetitionToRealm(realm).
32      let stepsRealm = realm.objects(StepRealm.self).
33
34      var newScore = 0.
35      for step in stepsRealm {.
36          if step.participant == participantRealm {.
37              if let stepName = stepName {.
38                  if step.name == stepName.rawValue && step.competition ==
        competitionRealm { newScore += step.score }.
39              } else {.
40                  newScore += step.score.
41              }.
42          }.
43      }.
44      do {.
45          try realm.write {.
46              participantRealm.score = newScore.
47              realm.add(participantRealm, update: .modified).
48          }.
49      } catch {.
50          throw DatabaseError.updateError.
51      }.
52      let resultParticipant = participantRealm.convertParticipantFromRealm().
53
```

```
54      return resultParticipant.
55  }.
```

ПРИЛОЖЕНИЕ Б

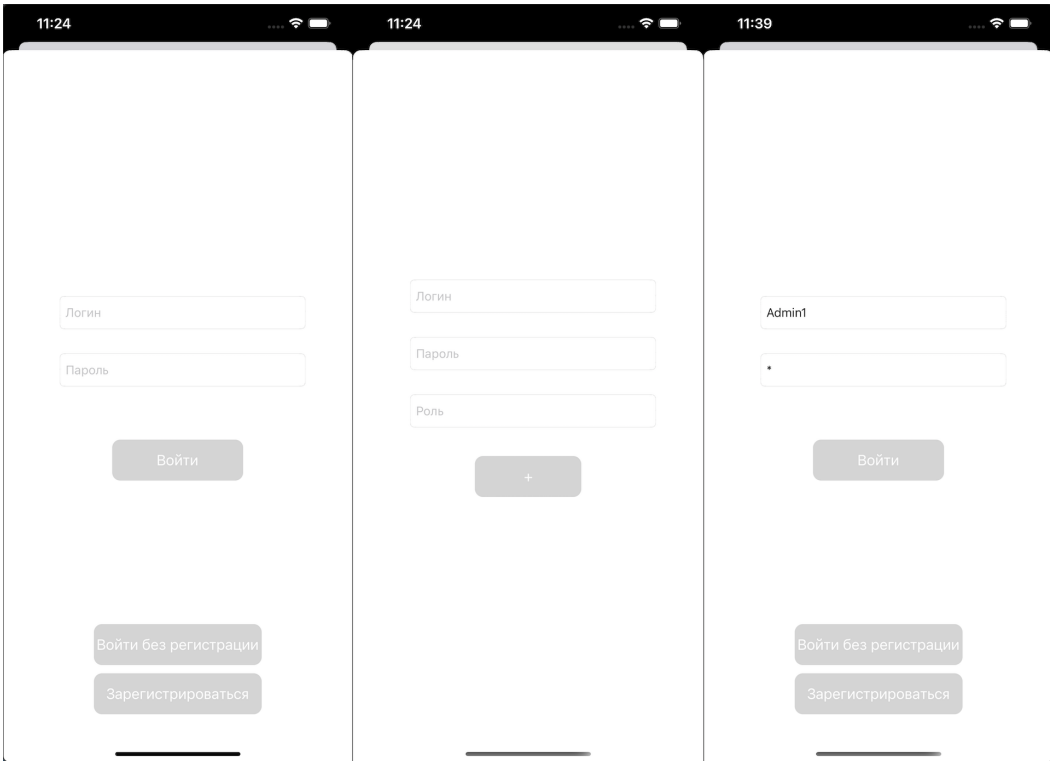

Рисунок 6 – Экраны персонализации

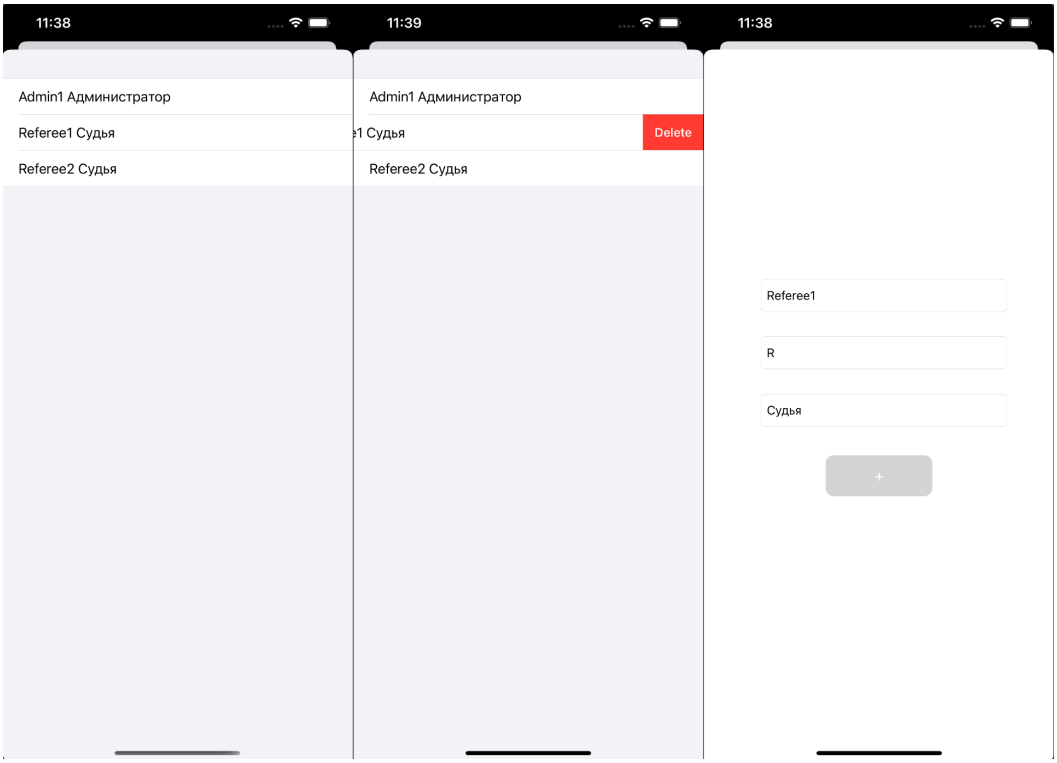

Рисунок 7 – Экраны доступа к данным об авторизованных пользователях системы (для администратора)

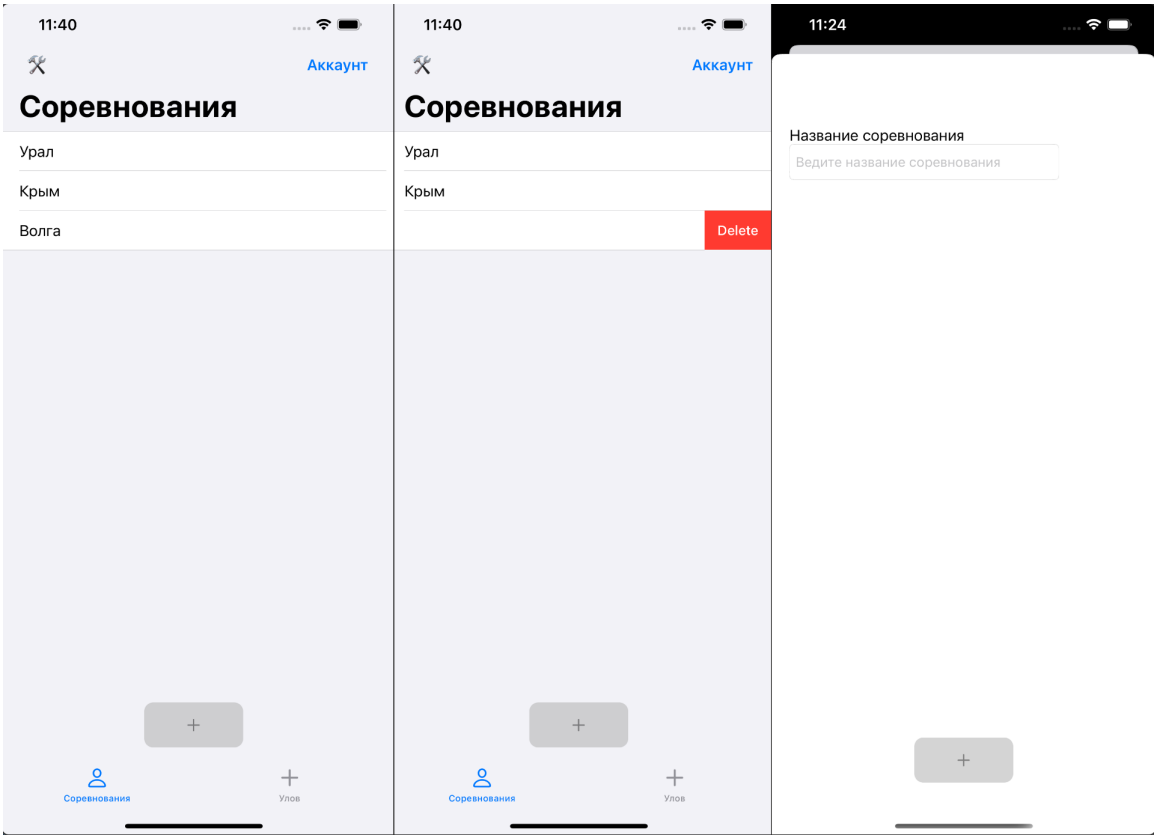

Рисунок 8 – Экраны формирования соревнований

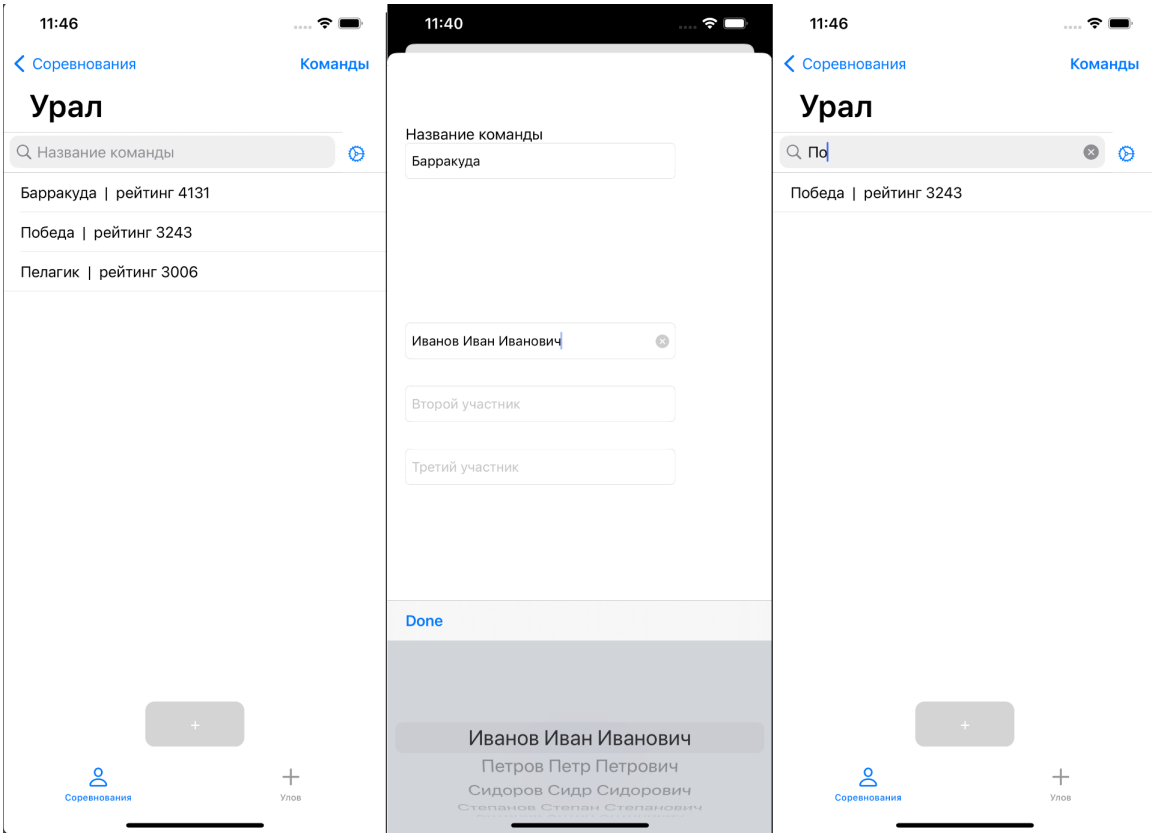

Рисунок 9 – Экраны формирования команд

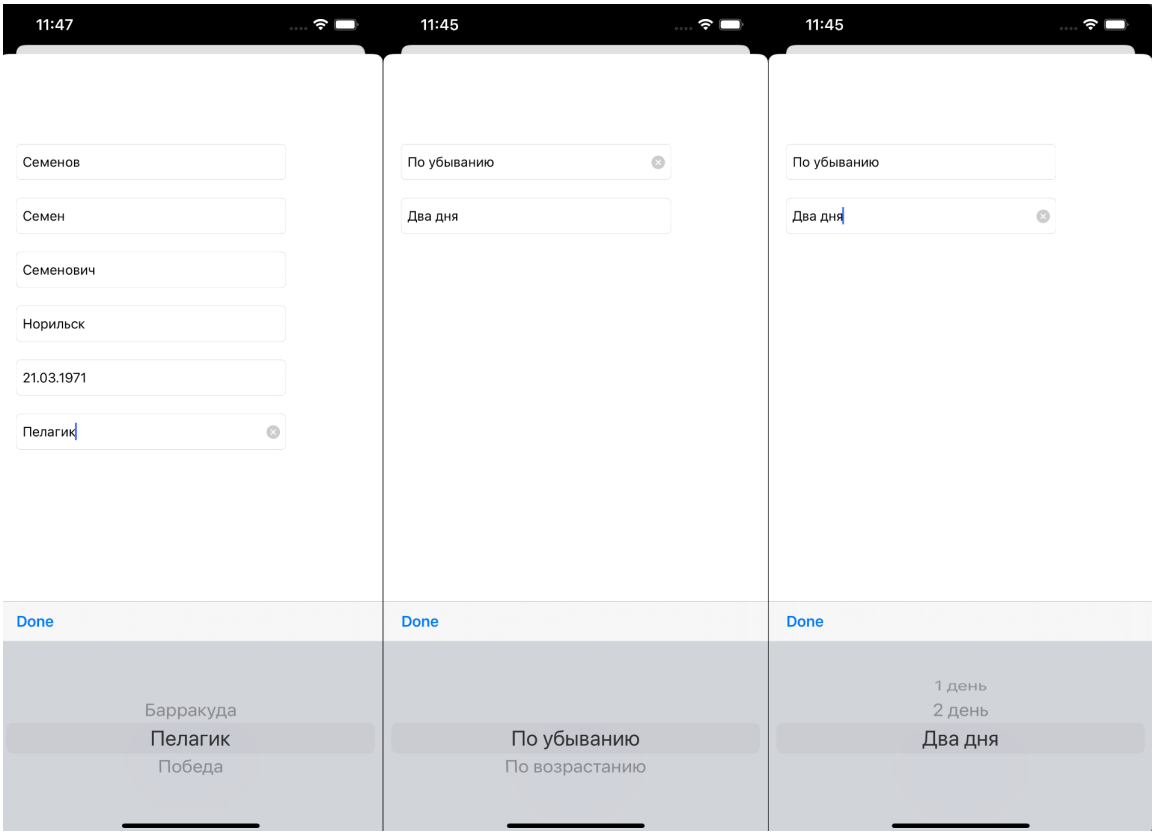

Рисунок 10 – Экраны создания участника, настройки параметров сортировки

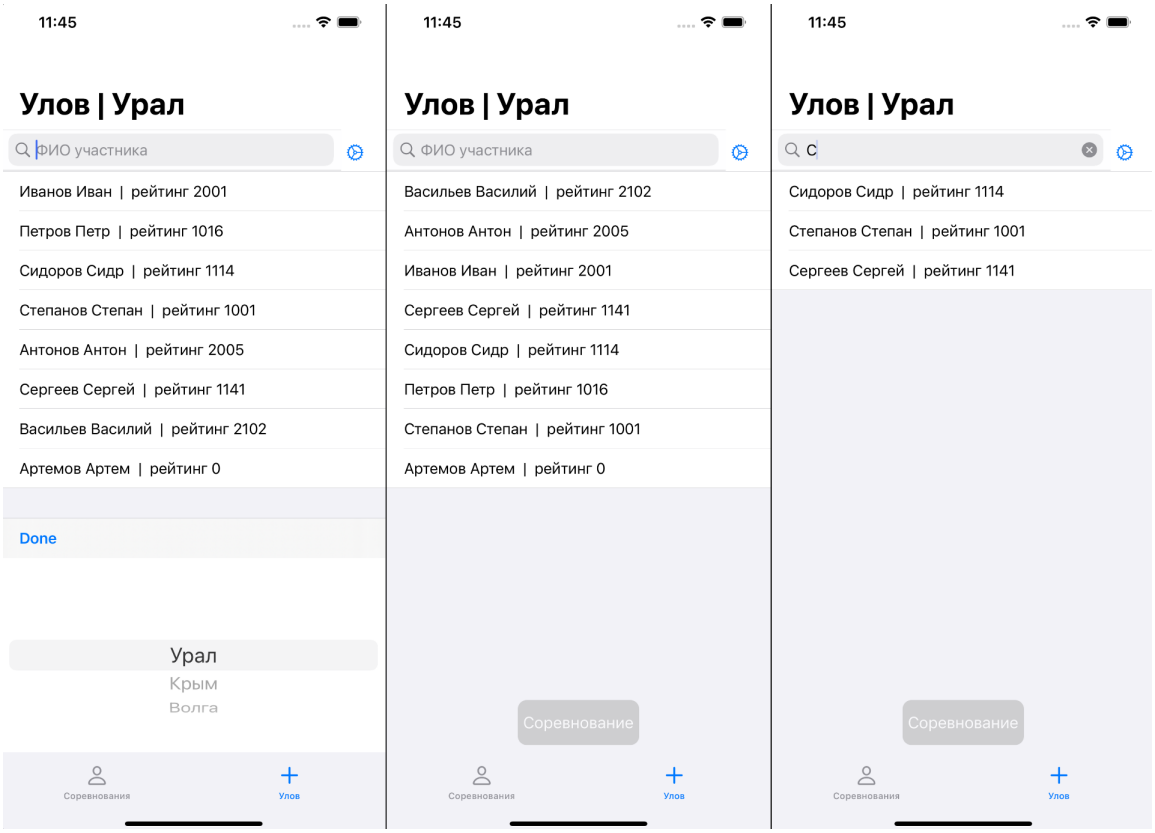

Рисунок 11 – Экраны формирования улова

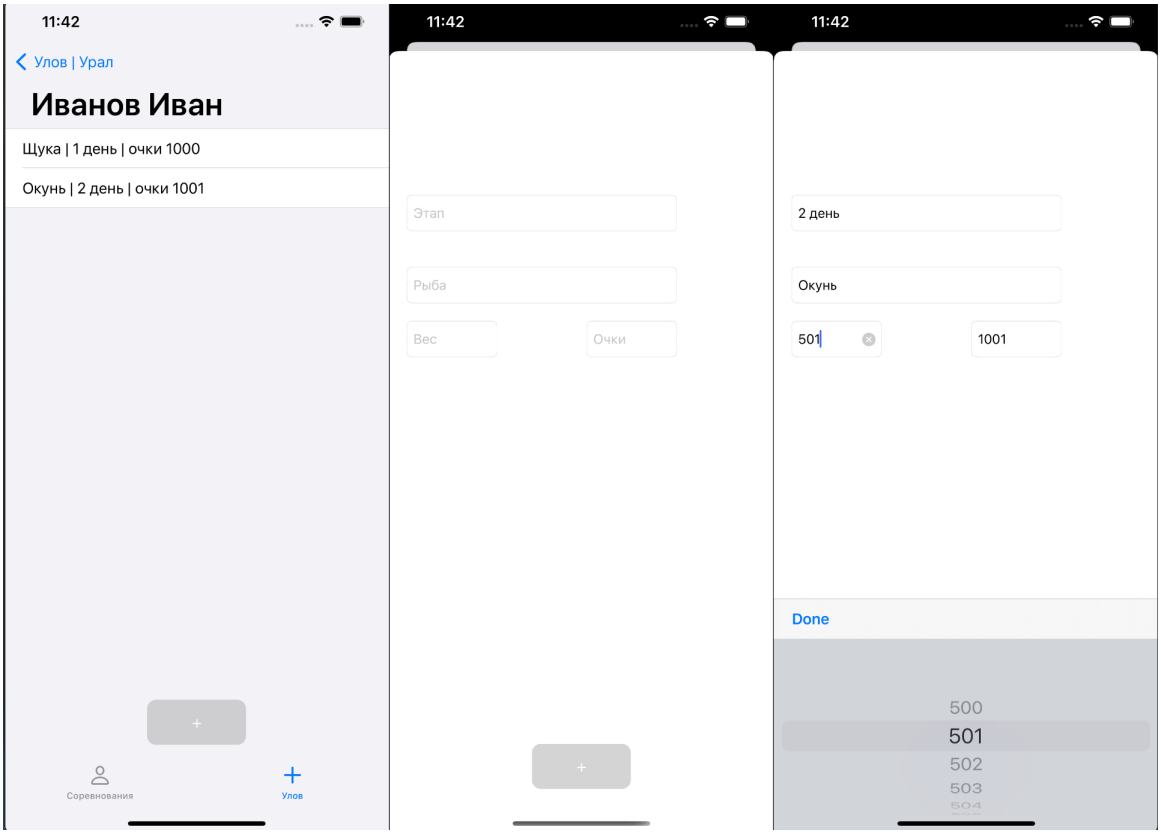

Рисунок 12 – Экраны формирования улова