Санкт–Петербургский государственный университет

Мовсин Марат Петрович

Выпускная квалификационная работа

Разработка бекэнда для мобильного приложения для создания игрового персонажа

Уровень образования: бакалавриат
Направление 02.03.01 «Математика и компьютерные науки»
Основная образовательная программа СВ.5152.2019 «Математика, алгоритмы и анализ данных»

Научный руководитель:
доцент, Факультет математики и компьютерных наук СПбГУ, к. ф.-м. н. Авдюшенко Александр Юрьевич

Рецензент:
Гардер Антон Владимирович,
старший инженер, Общество с ограниченной ответственностью «МПГ АйТи Солюшнз»

Санкт-Петербург
2023 г.

Содержание

Введение . . . 3

Постановка задачи . . . 5

1. Реализация системы способностей для создания персонажа . . . 7

1.1. Способности в DnD . . . 7
1.2. Основные принципы системы способностей . . . 8
1.3. Устройство дерева способностей . . . 12
1.4. CurrentState . . . 13
1.5. Действия . . . 14
1.6. Заклинания . . . 16
1.7. Поле requirements . . . 18
1.8. Приоритеты . . . 19
1.9. actionForChoice . . . 20
1.10. Инвентарь . . . 21

2. Создание базы данных . . . 23

2.1. Сохранение персонажа . . . 23
2.2. Сохранение и загрузка простых объектов . . . 24
2.3. Сохранение и загрузка дерева способностей . . . 27
2.4. Удаление данных и создание точной копии персонажа . . . 29

3. Разметка данных . . . 31

3.1. Прописывание AbilityNode . . . 31
3.2. Заклинания и магические предметы . . . 32

Заключение . . . 35

Список литературы . . . 36

Введение

Настольные ролевые игры (НРИ) — это игры, в которых игроки создают своих персонажей и вместе с другими игроками воплощают их в жизнь вместе с их историями и приключениями. Обычно игроки используют кубики, чтобы определить успех или неудачу в игровых ситуациях, а игра ведется под руководством управляющего игры или «мастера игры». НРИ могут иметь различные темы и жанры — от фэнтези до научной фантастики и мистики. НРИ являются активными и социальными играми, где игроки создают и развивают истории во время игры. Полезность ролевых игр не один раз подчеркивали люди из самых разных сфер жизни. Кроме отличного способа отвлечься от повседневной рутины, они так же помогают социализироваться, развивать воображение и даже обучать чему-то новому: например, они помогают в изучении иностранных языков [1]. За последние 50 лет образовалось много различных ролевых игр, и одна из них — DnD.

DnD (Dungeons and Dragons) — первая и одна из самых популярных настольных ролевых игр. Первая редакция DnD была разработана в 1974 году Гэри Гайгэксом и Дэйвом Арнесоном. С тех пор DnD прошла несколько редакций. Последняя редакция (пятая) вышла в 2014 году. В связи с пандемией covid-19 популярность DnD возросла особо сильно. Так в 2020 году продажи книг по DnD выросли на 30% [2]. По состоянию на 2023 год в DnD играет более 50 миллионов человек [3].

Одним из важнейших элементов DnD является лист персонажа — совокупность информации о способностях персонажа, его числовых характеристиках, инвентаре и так далее. По сути он полностью описывает персонажа. Это довольно сложная система информации и традиционно она записывается на специальном бумажном шаблоне. У листа персонажа записанного на бумаге есть множество недостатков — информацию на них нужно периодически обновлять и от частого использования ластика бумага истирается, их легко забыть дома или потерять и они занимают много места на столе. Бумажным листам персонажа существует альтернатива — электронные листы персонажа. Возможность создавать и использовать такие листы предоставляет сайт dndbeyond.com. Однако при отсутствии платной подписки этот сайт очень

сильно ограничивает выбор, из-за чего не получается полноценно создать лист персонажа. Цель нашего проекта — создание приложения под Android, которое будет бесплатным аналогом dndbeyond.com, тем самым позволив игрокам DnD извлекать для себя как можно больше пользы.

Постановка задачи

Создание приложения для создания DnD персонажа — довольно сложная задача. Базовая книга правил DnD содержит примерно 300 страниц из которых более 100 посвящено непосредственно созданию персонажа, а около 80 страниц занимает список заклинаний, которые тоже необходимо учитывать. При составлении персонажа доступно более 20 рас и 13 классов, каждый из которых состоит из 20 уровней, дающих новые способности и улучшающие старые. Всего в DnD сотни способностей, десятки из них обладают уникальными механиками, препятствующими унифицированию способностей. Приложение должно не только содержать в себе всю эту информацию, но и надлежащим образом структурировать её, чтобы не запутать игроков еще сильнее.

Для создания приложения необходимо реализовать три основных модуля:

- Front-end часть, отвечающую за взаимодействие с пользователем, а в точности изображение информации и интеракция пользователя с ней. Реализацию данной части можно разбить на следующие задачи:
  - Создание и дизайн экранов добавления и изменения персонажа. Суммарно 4 экрана: первый позволяет настроить характеристики, второй - выбрать предысторию и расу, третий - выбрать класс, четвертый - выбрать заклинания полученные от расы/класса.
  - Создание и дизайн экранов отображения информации о персонаже. Суммарно 8 навигационных веток состоящих из 1-2 экранов. Все они должны отображаться в основном экране, предоставляющим такие функциональности как выбор аватара или бросок кости.
- Back-end часть, отвечающая за хранение, обработку и передачу данных. Реализацию данной части можно разбить на данные задачи:
  - Создание и реализация архитектуры общения между базой данных и front-end модулем.
  - Реализация системы способностей для создания персонажа.

    - Реализация базы данных для хранения информации о всех персонажах.
    - Разметка данных (заклинания, инвентарь, информация для создания персонажа).

- **ML:**
  Генерация изображения и предыстории персонажа.
    - Исследование существующих решений и моделей, выбор наиболее оптимальных в контексте D&D.
    - Сбор, подготовка данных и дообучение выбранных моделей.
    - Оценка и настройка гиперпараметров.
    - Интеграция лучших моделей в мобильное приложение.

Проект был выполнен командой из трёх человек. Конкретно я выполнял следующие задачи:

- Реализация системы способностей для создания персонажа.
- Реализация базы данных для хранения информации о всех персонажах.
- Разметка данных (заклинания, инвентарь, информация для создания персонажа).

1. Реализация системы способностей для создания персонажа

1.1. Способности в DnD

Каждый персонаж, созданный в пятой редакции DnD, имеет множество характеристик и свойств, которые нужно уметь обрабатывать. При классическом хранении информации о персонаже - на бумаге - для краткого описания всех свойств персонажа используется оба оборота листа А4 (рисунок 1).

Рис. 1: Лист персонажа.

Каждое поле данного листа - это описание части персонажа, его свойства и характеристики, которые нужно заполнить, следуя правилам DnD. И если некоторые из них заполняется в вольном режиме (такие как внешность или привычки), то остальные появляются в следствие выбора способностей рас и классов.

Способность – это некое свойство, которое может применить к себе пресонаж. В игре сотни различных способностей: начиная от простых повышений

характеристик (например, прибавьте +2 к значению ловкости), заканчивая свойствами, имеющие сложнейшие зависимости, опирающиеся на события, происходящие вокруг персонажа и броски костей, например, способность «Волна дикой магии» чародея. Чаще всего игрок вручную рассчитывает, какие способности можно брать и переписывает их на лист персонажа, что часто приводит к ошибкам - или в вычислениях на стадии генерации персонажа, или при переписывании. Чтобы избежать этих проблем наше приложение предоставляет систему, автоматизирующую данный процесс. Чтобы реализовать данную автоматизацию, наша система должна уметь обрабатывать все возможные способности, предоставленные в пятой редакции DnD. О том, как реализовано решение данной проблемы, будет рассказано в данной секции.

Кроме того, как было упомянуто ранее, данный лист персонажа содержит лишь часть описаний способностей – так как в некоторых случаях не хватило бы и десяти листов A4 чтобы отобразить всю информацию о персонаже. Так как электронное приложение не так сильно ограничено в хранении текстовой информации, наша система должна уметь обрабатывать полные описания способностей, чтобы пользователю не надо было искать их в книге правил или интернете.

1.2. Основные принципы системы способностей

Вся информация о способностях и характеристиках хранится в дата классе [4] characterInfo. Она содержит численные поля, описывающие различные характеристики персонажа, строковые поля для названия класса и расы персонажа и различные контейнеры, содержащие возможные действия персонажа, его пассивные способности, владения инструментами и т.д.

```
data class CharacterInfo(
    var level: Int = 0,
    var spellCasterLevel: Float = 0f,
    var spellsLevel: Int = 0,
    //...
    var conditionImmunities: MutableSet<Conditions> =
        mutableSetOf(),
```

```
    var actionsMap: MutableMap<String, Action> =
        mutableMapOf(),
    var additionalAbilities: MutableMap<String, String> =
        mutableMapOf(),
    var inventory: MutableMap<String, InventoryItemInfo> =
        mutableMapOf(),
    var spellsInfo: MutableMap<String, CharacterSpells> =
        mutableMapOf(),
    var currentState: CurrentState = CurrentState()
)
```

Эта структура не отражает построение персонажа, которое выполняется по определённым правилам. Рассмотрим персонажа с точки зрения генерации. CharacterInfo складывается из блоков способностей, каждый из которых каким-либо образом влияет на характеристики и способности персонажа. Например, блок способностей «Увеличение ловкости» увеличивает характеристику «Ловкость» на 1. Блоки способностей можно представить как функции, которые по CharacterInfo выдают обновлённый CharacterInfo. Однако разбиения на блоки способностей недостаточно, чтобы контролировать корректное создание персонажа. Игрок не может произвольным образом выбирать блоки способностей персонажа. Изначально игроку доступно только несколько блоков способностей. Когда игрок берёт блок, он открывает ему доступ к некоторым другим блокам. По сути блоки способностей представляют из себя дерево. Способности конкретного персонажа образуют некоторое поддерево этого дерева. Для хранения структуры этих деревьев мы создали классы AbilityNode и CharacterAbilityNode соответственно.

AbilityNode хранит в себе информацию о вершине этого дерева, общую для всех персонажей.

```
open class AbilityNode(
    val name: String,
    val changesInCharacterInfo: (
        abilities: CharacterInfo
    ) -> CharacterInfo,
```

```
    val getAlternatives: MutableMap<
        String,
        (abilities: CharacterInfo?) -> List<String>
    >,
    val requirements: (abilities: CharacterInfo) -> Boolean,
    var description: String,
    val isNeedsToBeShown: Boolean,
    val priority: Priority,
    val actionForChoice: Map<
        String,
        (choice: String, abilities: CharacterInfo) -> CharacterInfo
    >,
) {
    //...
}
```

name — название AbilityNode, которое также является его id

changesInCharacterInfo — функция [5], показывающая, как блок способностей влияет на персонажа. Так, например для блока характерисик «Увеличение ловкости», эта функция выглядит так:

```
changesInCharacterInfo = { abilities: CharacterInfo ->
    abilities.dexterity += 1
    abilities
},
```

Поле getAlternatives показывает, к каким другим блокам способностей открывает доступ данный блок способностей или какие другие опции позволяет выбрать данный блок способностей. Каждый элемент getAlternatives — функция, по текущему characterInfo возвращающая список доступных опций, из которых можно выбрать один. Как правило это названия AbilityNode, доступ к которым даёт данный AbilityNode. Однако, иногда это поле может хранить другие выборы, о чём будет сказано позже.

Ниже приведён пример getAlternatives.

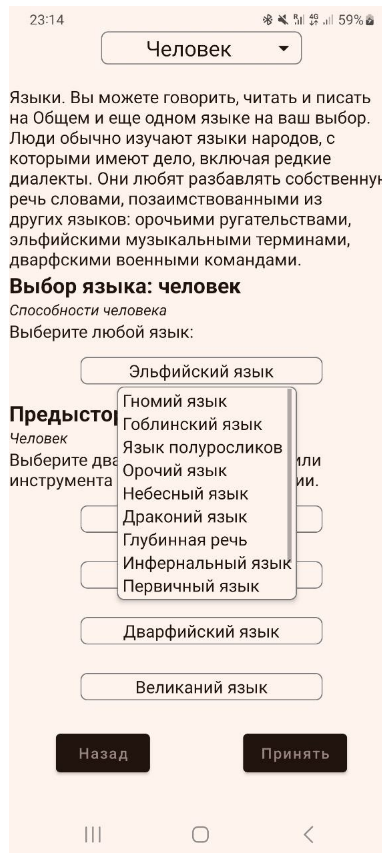

Рис. 2: Выбор языка.

```
getAlternatives = mutableMapOf(
    Pair("first", { listOf(abilityScoreImprovement.name) }),
    Pair("second", { listOf(slowFall.name) }),
),
```

В этом примере списки выборов состоят из одного элемента. В таких списках опции выбираются автоматически, не требуя выбора со стороны пользователя. На рисунке 2 представлен выбор из множества вариантов, показ которых реализован в front-end модуле.

Об остальных полях класса AbilityNode мы более подробно расскажем позже.

CharacterAbilityNode хранит то, как данный блок способностей реализован для конкретного персонажа.

```
open class CharacterAbilityNode(
    open val data: AbilityNode,
```

```
    var chosenAlternatives: MutableMap<String, CharacterAbilityNode>,
    var character: Character? = null,
    var chosenAlternativesForActions: MutableMap<String, String>
        = mutableMapOf()
) {
    //...
}
```

Этот класс хранит информацию о том, что данный персонаж обладает данным AbilityNode и выбрал определённые опции в нём.

Таким образом, набор CharacterAbilityNode хранит дерево, структура которого задана набором AbilityNode. При использовании персонажа мы вычисляем characterInfo, применяя changesInCharacterInfo всех вершин этого дерева к нулевому characterInfo (то есть такому, у которого все численные поля нулевые, а строковые поля и поля-контейнеры пустые). При создании и редактировании персонажа (то есть расширении этого дерева) мы сначала вычисляем characterInfo, а затем смотрим на списки альтернтив для уже взятых вершин.

Данная реализация позволяет структурировать не похожие друг на друга способности. Независимо от типа способности каждая из них как-то описывается полями объекта класса CharacterInfo, что означает что функция changesInCharacterInfo успешно может применить любую способность, представленную в пятой редакции DnD. В свою очередь, архитектура показа только доступных альтернатив структурирует приложение и освобождает пользователя от прочтения бесполезной информации, помогая сфокусироваться только на доступных возможностях.

Далее рассмотрим некоторые конкретные особенности создания и хранения персонажа.

1.3. Устройство дерева способностей

Основным источником способностей персонажа является его класс. Класс состоит из 20 уровней, каждый из которых даёт какие-то новые способности или просто повышает количество хитов. Несмотря на то, что увеличе-

ние уровня имеет всего одну способность в alternatives (следующий уровень), данный выбор не должен выпоняться автоматически, как прочие добавления способностей, так как уровень отдельно выбирается пользователем. Поэтому добавление следующего уровня происходит не через поле getAlternatives, а через специальное поле next_level, содержащее имя abilityNode, отвечающего за следующий уровень. Это поле содержится не у всех abilityNode, а только у тех, котоорые отвечают за повышение уровня. Они имеют тип AbilityNodeLevel, отнаследованный от AbilityNode.

Помимо класса у персонажа есть какие-то базовые, общие для всех способности, а также способности, которые ему даёт раса. На диаграмме 3 показано, как выглядит основная часть дерева способностей для человека-барда 2 уровня.

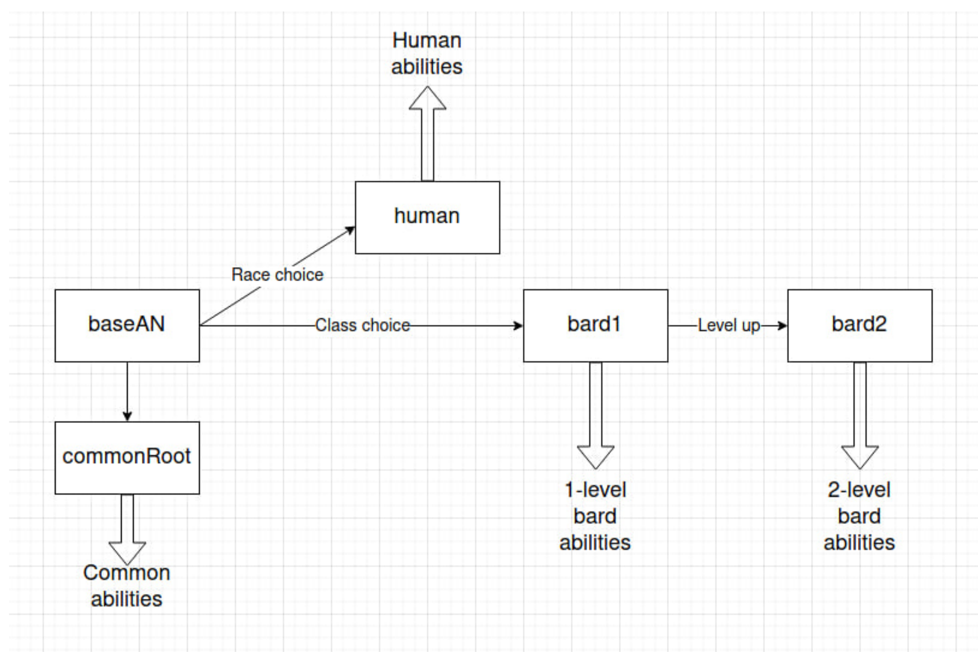

Рис. 3: Дерево способностей барда.

1.4. CurrentState

Большая часть характеристик персонажа складывается из его AbilityNode. Однако есть некоторые параметры, которые нельзя вычислить только по набору способностей. Так например, текущие хиты персонажа постоянно меняются, персонаж может надевать и снимать оружие и броню. Такие пара-

метры хранятся в специальном классе CurrentState. Оно не пересобираются при каждой сборке персонажа. CurrentState хранится отдельно. Часть из них пересобирается при изменении уровня персонажа (например, текущие хиты), часть же меняется только вручную (например, надетая броня).

1.5. Действия

Одна из важнейших характеристик персонажа — набор действий, которые он может совершать. Каждое действие выражается специальной структуре Action

```
data class Action(
    var name: String,
    var description: String,
    var type: ActionType,
    var relatedCharges: String = ""
)
```

Action хранятся в словаре actionsMap, который по названию выдаёт Action.

Действия в DnD классифицируются по длительности выполнения. Эту характеристику действия описывает поле type. Оно представляет из себя enum, принимающий одно из следующих значений: Action, Bonus, Reaction, PartOfAction, Long, Additional. При показе действия разделяются как раз по этому признаку.

Некоторые действия могут выполяться ограниченное число раз в день или требуют использования каких либо ресурсов. Для реализации этого используется поле relatedCharges и класс ChargesCounter:

```
data class ChargesCounter(
    var current: Int,
    var maximum: Int
)
```

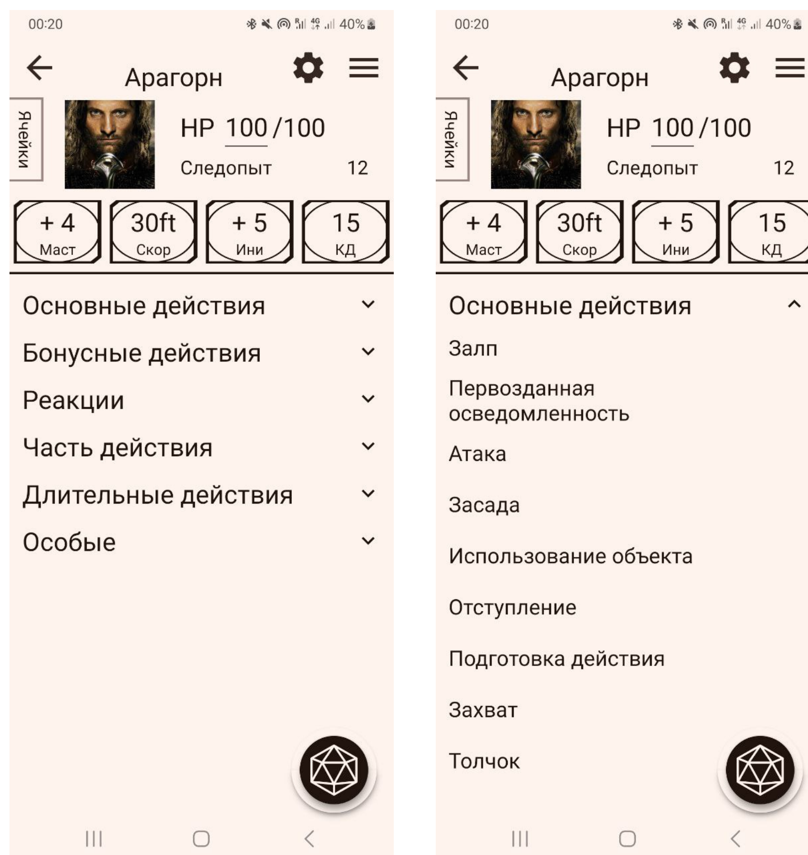

Рис. 4: Отображение действий.

ChargesCounter содержит информацию о текущем и максимальном количестве зарядов данного типа. ChargesCounter хранятся в словаре, который хранится в CurrentState (так как количество зарядов должно уметь меняться без пересборки персонажа). Этот словарь по имени заряда выдаёт ChargesCounter, отвечающий за этот заряд. В то время, как поле relatedCharges говорит, какой заряд используется для данного действия (если действие можно выполнять неограничено, без траты зарядов, то это поле — пустая строка). Приведём пример. Класс «монах» позволяет совершать действие «шквал ударов», которое требует расходования очков «ци». Выглядит это так:

```
abilities.actionsMap["Шквал ударов"] =
            Action(
                name = "Шквал ударов",
                description = "Сразу же после того, как ...\n",
                type = ActionType.Bonus,
```

```
relatedCharges = "Ци"
)
```

Рис. 5: Пример отображения зарядов у действий.

На рисунке 5 представлен пример отображения действия с зарядами.

Если действие не отсылается к конкретному заряду со своим собственным названием, а просто может быть выполнено ограниченное число раз в день, то для него создаётся персональный заряд, имя которого совпадает с именем действия.

1.6. Заклинания

Многие персонажи обладают способностью накладывать заклинания. В этой способности есть два основных аспекта: списки заклинаний и ячейки заклинаний. Списки заклинаний это составленные по особым правилам наборы заклинаний, которые способен накладывать персонаж. Ячейки заклинаний

это магическая энергия, ресурсы, необходимые для накладывания заклинаний. Поподробнее поговорим о каждом из этих аспектов и о их реализации в нашем приложении.

Начнём со списков заклинаний. Есть общий список заклинаний, содержащий все заклинания. У каждого заклинательского класса есть свой подсписок заклинаний, которые в принципе доступны данному классу. Для части классов на этапе создания персонажа необходимо выбрать фиксированное количество заклинаний из этого списка, которые будут считаться изученными, и персонаж будет способен их накладывать. Для части классов все заклинания считаются известными, но необходимо периодически выбирать часть из них во время игры: эти заклинания будут считаться подготовленными и персонаж будет способен накладывать только их. А у класса «волшебник» присутствует и та, и другая механика. Волшебник изучает какое-то количество заклинаний, а потом подготавливает часть из них. Помимо этого у каждого заклинания есть уровень и персонаж не может использовать заклинания со слишком высоким уровнем. Для хранения информации о списках заклинаний используется класс CharacterSpells:

```
data class CharacterSpells(
    var className: String = "",
    var spellLists: SpellLists = SpellLists(),
    var maxKnownSpellsCount: Int = -1,
    var maxKnownCantripsCount: Int = -1,
    var maxPreparedSpellsCount: Int = -1,
    var maxPreparedCantripsCount: Int = -1
```

className говорит, из списка заклинаний какого класса необходимо формировать все списки.

spellLists содержит информацию о дополнительных известных или подготовленных спеллах, выбранных автоматически.

maxKnownSpellsCount говорит, сколько заклинаний можно выбрать в качестве известных (равен -1, если все заклинания автоматически известны), maxPreparedSpellsCount говорит, сколько заклинаний можно выбрать в

качестве подготовленных (равен -1, если все заклинания автоматически подготовлены). maxKnownCantripsCount и maxPreparedCantripsCount делают то же самое для кантрипов (заклинаний 0 уровня, которые выбираются отдельно от остальных заклинаний).

Максимальный доступный уровень заклинания хранится в специальном поле characterInfo spellsLevel.

Теперь разберёмся с ячейками заклинаний. Ячейки заклинаний различаются по уровню. Раз в день каждую ячейку можно потратить, чтобы сотворить заклинание с уровнем не выше уровня ячейки. Набор ячеек зависит от уровня заклинателя. Уровень заклинателя совпадает с уровнем перснажа для большинства заклинателских классов, однако для некоторых классов он равен половине или трети от уровня персонажа. Поэтому мы храним его в отдельном поле characterInfo spellCasterLevel. Сами же ячейки реализованы с помощью ChargesCounter.

1.7. Поле requirements

Не все AbilityNode доступны игроку, даже если персонаж обладает предшествующей AbilityNode. Так например, нельзя взять способность «Оборонительный дуэлянт», если ловкость персонажа меньше 13. Для этого создано поле requirements. Это лямбда функция, которая по текущему characterInfo определяет, можно ли взять способность. Если нельзя, то AbilityNode не отобразится в списке возможных вариантов.

Вот так эта функция выглядит для одной из способностей барда:

```
var additionalMagicalSecrets = AbilityNode(
    //...
    requirements = { abilities: CharacterInfo ->
        abilities.level >= 6
    },
    //...
)
```

Существует так же реализация без этого поля, которая состоит в том, чтобы проверять возможность получения способности у предка данной спо-

собности на стадии получения alternatives. Но данная реализация имеет множество недостатков: если одна и та же способность появляется у многих предков, то у каждого надо будет дублировать код проверки, что является нарушением парадигмы программирования DRY [6]. Кроме того, функция getAlternatives может стать излишне сложной, что может привести к плохой читаемости кода.

1.8. Приоритеты

Функции changesInCharacterInfo в общем случае не коммутируют. Так например, есть способность, увеличивающая значение класса брони в зависимости от значения мудрости, и есть способность, увеличивающая мудрость. Эта проблема решается введением поля priority. Это enum, который может принимать 5 возможных значений: DoFirst, DoAsSoonAsPossible, Basic, DoAfterBasic и DoLast. Сначала выполняются changesInCharacterInfo из AbilityNode с priority DoFirst, затем из AbilityNode с priority DoAsSoonAsPossible и так далее. Большая часть AbilityNode имеет priority Basic. AbilityNode, меняющие такие часто используемые параметры, как «Ловкость», «Сила» или «Уровень» имеют priority DoFirst. Много от чего зависящие AbilityNode имеют priority DoLast. AbilityNode, которые нужно сделать после DoFirst, но до Basic или после Basic, но до DoLast, имеют priority DoAsSoonAsPossible и DoAfterBasic соответственно. В остальном же, функции применяются в порядке DFS, что позволяет вершинам-потомкам менять способности, полученные от вершин-предков с таким же приоритетом.

Сборка персонажа с учётом приоритетов происходит с помощью метода merge класса CharacterAbilityNode. Он рекурсивно применяет все способности с данным приоритетом.

```
fun merge(
    abilities: CharacterInfo,
    priority: Priority
): CharacterInfo {
    var result: CharacterInfo = abilities
    if (data.priority == priority) {
```

```
        result = data.merge(result)
        for ((option, choice) in chosenAlternativesForActions) {
            data.actionForChoice[option]?.let {
                result = it(choice, result)
            }
        }
    }

    for ((_, value) in chosenAlternatives.entries) {
        result = value.merge(result, priority)
    }
    return result
}
```

1.9. actionForChoice

Как уже было сказано выше, альтернативы из getAlternatives как правило позволяют выбрать дочернюю abilityNode, однако иногда выбор происходит между другими опциями. Происходит это когда необходимо сделать выбор из большого количества варианта (нередко нескольких сотен), причём эти варианты влияют на персонажа схожим образом. Создавать abilityNode на каждый вариант такого выбора было бы неоптимально. Приведём пример такого выбора. У класса «бард», есть способность, позволяющая изучить любое заклинание. Хочется, чтобы заклинание можно было бы выбрать аналогично тому, как выбираются дочерние abilityNode. Реализовано это следующим образом. У каждого выбора есть своё имя (как и в случае обычных выборов). getAlternatives для выбора с данным именем позволяет выбрать одну из строк. actionForChoice для выбора с данным именем возвращает функцию, которая в зависимости от выбранной строки меняет characterInfo.

Вот так выглядит использование actionForChoice:

```
var magicalSecrets_10 = AbilityNode(
    //...
    getAlternatives = mutableMapOf(
```

```
        Pair("first", {
            spellist.filter { it.level <= 5 }.map { it.name }
        }),
        Pair("second", {
            //...
        })
    ),
    //...
    actionForChoice = mutableMapOf(
        Pair("first") { choice: String, abilities: CharacterInfo ->
            abilities.spellsInfo["..."] =
                CharacterSpells(
                    className = ...,
                    maxKnownSpellsCount = 0,
                    maxKnownCantripsCount = 0,
                    spellLists = SpellLists(
                        knownSpells = mutableSetOf(choice)
                    )
                )
            abilities
        },
        Pair("second") { choice: String, abilities: CharacterInfo ->
            //...
        }
    )
)
```

Если actionForChoice не содержит функции для данного имени выбора, то используется обычный сценарий с выбором дочерней abilityNode.

1.10. Инвентарь

Информация о каждом предмете из инвентаря хранится в специальном классе InventoryItemInfo. В нём содержится название предмета, количество

экземпляров у персонажа, а также заряды предмета, в том случае если у предмета есть ограничения по количеству использований.

```
data class InventoryItemInfo(
    val itemName: String,
    var count: Int = 0,
    var notes: String = "",
    var maximumCharges: Int = 0,
    var currentCharges: Int? = null
)
```

InventoryItemInfo лежат в словаре, который по названию предмета выдаёт информацию о нём. Он не находится в CurrentState, однако информация в нём также не пересобирается каждый раз.

Информация о том, какие предметы надеты хранится в виде множества строк в CurrentState.

2. Создание базы данных

2.1. Сохранение персонажа

Одна из самых важных функций, без которой приложение не имело бы смысла - сохраненение персонажа. Для создания успешной архитектуры нужно не только научиться обрабатывать все правила игры, но и сохранять полученную структуру в базу данных. Данная реализация должна учитывать устройство способностей, описанные в секции 1, быть эффективной и надежной.

Для сохранения данных использовалась технология, разработанная другим участником команды, которая позволяет сохранять пару ключ-значение, загружать её и удалять значения по ключу, соответствующая требованиям эффективности и надежности. В данной секции будет разобрана реализация сохранения перснажа с использованием этой технологии.

В нашей архитектуре персонаж представлен объектом data class Character:

```
data class Character(
    var id: Int,
    var image: Bitmap?,
    var name: String,
    var characterInfo: CharacterInfo,
    var customAbilities: CharacterInfo,
    var baseCAN: CharacterAbilityNode,
    var notes: MutableList<Note>
)
```

Где:

- id - уникальный индетификатор, необходим для базы данных.
- image - изображение аватара персонажа, равно null в случае если пользователь его не выбрал.
- name - имя персонажа.

- characterInfo - информация о персонаже, генерируемая деревом способностей.
- customAbilities - информация о персонаже, дополнительно введенная пользователем.
- baseCAN - корень дерева способностей.
- notes - дополнительные записи о персонаже, созданные пользователем.

Не все поля данного класса надо сохранять. Например, почти все поле characterInfo генерируется корнем способностей (за исключением spellsInfo, inventoryInfo и currentState), а поле data во всех объектах класса CharacterAbilityNode хранится в ресурсах приложения. Поэтому при процессе сохранения для эффективной работы надо выделить лишь необходимую информацию и сохранить её. Соответсвенно процесс сохранения реализован следующим образом:

- Используя уникальный таг, сохранить лист всех идентификаторов персонажей.
- Сохранить каждого персонажа:
  - Сохранить полностью поля image, name, notes.
  - Из поля characterInfo сохранить spellsInfo, inventoryInfo и currentState.
  - Сохранить дерево способностей.

Разберем подробнее как это реализовано.

2.2. Сохранение и загрузка простых объектов

Все описанные шаги сохранения и загрузки персонажа (кроме работы с деревом способностей) можно реализовать используя библиотеку Gson [7]. Данная библиотека позволяет конвертировать в формат Json примитивы, коллекции примитивов, data classes и комбинации вышеперчисленного. Так как все наши поля попадают под данную классификацию (за исключением дерева

способностей), мы можем без особого труда конвертировать объект в строку и обратно.

Ниже представлен алгоритм загрузки персонажей:

```
private suspend fun loadCharacters() {
    // Для конвертации строки в объект надо знать тип объекта
    val listIdsType: Type = object : TypeToken<List<Int>>() {}.type
    // Получение из базы данных JSON
    val charactersListJson = db.getString(DB_CHARACTER_IDS)
    // Конвертация
    val charactersList: List<Int> =
        Gson().fromJson(charactersListJson, listIdsType) ?:
        emptyList()

    // Для каждого id надо загрузить персонажа
    for (id in charactersList) {
        val character = loadCharacter(id)

        character?.let {
            characters[id] = character
        }
    }
}

private fun loadCharacter(id: Int): Character? {
    // Получить имя и аватар персонажа
    val name = db.getString(DB_CHARACTER_NAME + id.toString())
    // Изображение сохраняется/загружается как файл
    val bitmap = loadCharacterBitmap(id)
    characterImages[id] = bitmap

    // Инициализирвать пустого персонажа
    val character = Character(
        id = id,
```

```
        name = name,
        bitmap = bitmap,
        characterInfo = CharacterInfo(),
    )

    // Получить введеную пользователем дополнительную информацию
    val characterCharacterInfoJson =
        db.getString(id.toString() + DB_CHARACTER_CUSTOM)
    val characterCharacterInfo =
        if (characterCharacterInfoJson != null)
            Gson().fromJson(
                characterCharacterInfoJson,
                CharacterInfo::class.java
            )
        else CharacterInfo()
    // Аналогично получить остальные поля
    //...

    // Получить дерево способностей
    val baseCharacterAbilityNode =
        loadCharacterNode("base_an", id, ".", character)

    // Инициализировать загруженные поля
    character.customAbilities = characterCharacterInfo
    character.baseCAN = baseCharacterAbilityNode
    //...

    //Применить способности к пресонажу
    mergeAllAbilities(character)

    return character
}
```

Алгоритм сохранения аналогичен, за исключением того что теперь Gson используется для конвертации из объекта в JSON:

```
val customInfoJson = Gson().toJson(
    characters[id]!!.customAbilities
)
db.putStringsAsync(
    listOf(
        Pair(
            id.toString() + DB_CHARACTER_CUSTOM,
            customInfoJson
        )
    )
)
```

2.3. Сохранение и загрузка дерева способностей

Напомним, что класс CharacterAbiilityNode реализован следующим образом:

```
open class CharacterAbilityNode(
    open val data: AbilityNode,
    var chosenAlternatives: MutableMap<
        String,
        CharacterAbilityNode
    >,
    var character: Character,
    var chosenAlternativesForActions: MutableMap<String, String>
)
```

Как было упомянуто ранее, для сохранения дерева способностей мы не должны сохранять каждый CharacterAbilityNode полностью, так как поле data есть в ресурсах приложения. Сохранять персонажа тоже не нужно, так как при загрузке дерева способностей каждому узлу назначится соответствующий персонаж. Тогда для успешного восстановления дерева способностей

}

достаточно сохранить chosenAlternatives и chosenAlternativesForActions. Рассмотрим следующий алгоритм сохранения:

```
private fun saveCharacterNode(
        characterAbilityNode: CharacterAbilityNode,
        characterId: Int,
    ) {
    // Рекурсивно повторяем алгоритм для всех узлов - потомков
    val children = characterAbilityNode.chosenAlternatives.values
    for (characterNode in children) {
        saveCharacterNode(
            characterNode,
            characterId,
        )
    }

    // Инициализируем ассоциативный массив option -> option_name
    val chosenAlternativesNames: MutableMap<String, String> = ...
    // Сохраняем данный массив
    saveToDataBase(
        key = characterId + characterAbilityNode.data.name + tag1,
        value = chosenAlternativesNames
    )

    // Аналогично сохраняем chosenAlternativesForActions
    val chosenAlternativesForActions =
        characterAbilityNode.chosenAlternativesForActions
    saveToDataBase(
        key = characterId + characterAbilityNode.data.name + tag2,
        value = chosenAlternativesForActions
    )
}
```

Как же по сохраненным данным восстановить дерево способностей?

Заметим, что мы знаем имя корневого узла и идентификатор персонажа - значит можем для корневого дерева получить списки выбранных узлов. В данных списках содержатся имена потомков, которые нужно загрузить - значит для них мы можем рекурсивно повторить операцию. В результате данного алгоритма мы загружаем все дерево способностей.

В вышеуказанном алгоритме есть одна проблема. Предположим, какую-то способность персонаж взял несколько раз - например, способность "Повышение характеристик"можно в некоторых случаях взять до семи раз. В вышеописанном алгоритме каждый узел, отвечающий за "Повышение характеристик имеет одинаковые идентификаторы способности и одинаковые идентификаторы персонажа - следовательно, произойдет коллизия, в результате которой из всех способностей сохранится последняя - и данные о всех предыдущих будут утеряны.

Заметим, что если способности с разными выборами могут иметь одинаковые индетификаторы способности, то если мы добавим к нему путь (сконкатенированные имена способностей от корня до данной), то мы получим для каждой способности уникальный идентификатор. Другими словами, чтобы сохранить списки выборов нужно использовать данный ключ:

```
key = characterId + path + characterAbilityNode.data.name + tag1
```

Заметим, что при загрузке узлов мы всегда знаем путь до них (так как для загрузки каждого следующего узла надо загрузить всех предков), следовательно алгоритм загрузки все еще корректен.

2.4. Удаление данных и создание точной копии персонажа

Как было упомянуто ранее, в базе данных существует функция, которая по списку ключей удаляет из базы данных значения, лежащие по данным ключам. Следовательно для удаления персонажа реализован алгоритм, который аналогично описанному выше алгоритму сохранения/загрузки персонажа собирает необходимые ключи в список и удаляет данные значения.

Для создания копии персонажа тоже нужно реализовать алгоритм, так как в случае копирования объекта класса Character для всех не примитивных

полей (в нашем случае всех полей) не создатится копия объекта класса, а скопируется ссылка на данный объект. Это означает что при изменении каких либо полей в копии персонажа данные изменения будут отражены и в оригинале, что неприемлимо.

Так как Kotlin не поддерживает рекурсивное клонирование и не имеет методов клонирования объектов класса, был релизован алгоритм, который как и при сохранении/загрузке персонажа выделяет необходимую информацию, вручную создает клон объекта и присваивает его копии персонажа. Благодаря данному алгоритму пользователь может редактировать персонажа и экспериментировать со сборками, никак не изменяя оригинал.

3. Разметка данных

3.1. Прописывание AbilityNode

Основой генерации персонажа являются AbilityNode. Все AbilityNode необходимо было прописать в коде. Реализовано это было следующим образом. Есть некоторое количество словарей, которые по имени AbilityNode выдают сам AbilityNode. Каждый такой словарь определяется в отдельном файле и складывается из AbilityNode, прописанных непосредственно в этом файле, и других словарей, описывающих какие-либо подмножества AbilityNode.

Приведём пример.

```
var mapOfAn: MutableMap<String, AbilityNode> = (
        mutableMapOf(
            Pair(baseAN.name, baseAN),
            Pair(customBackstory.name, customBackstory)
        )
                + mapOfCommonAN
                + mapOfRaces
                + mapOfClasses
                + mapOfFightingStyles
                + mapOfAbilityScoreImprovement
                + mapOfSkills
                + mapOfExpertise
                + mapOfSkillAndExpertise
                + mapOfLanguages
                + mapOfTools
                + mapOfToolsExpertise
        ).toMutableMap()
```

Словарь mapOfAn складывается из AbilityNode baseAN и customBackstory, описанных в том же файле, а также других словарей, описанных в других файлах. Те, в свою очередь имеют подобную структуру. Например, mapOfClasses определяется так:

```
val mapOfClasses = (
        mapOfMonkAbilities
                + mapOfBarbarianAbilities
                + mapOfFighterAbilities
                + mapOfSorcererAbilities
                + mapOfClericAbilities
                + mapOfWizardAbilities
                + mapOfBardAbilities
                + mapOfRogueAbilities
                + mapOfPaladinAbilities
                + mapOfDruidAbilities
                + mapOfRangerAbilities
                + mapOfWarlockAbilities
                + mapOf(Pair(extraAttack.name, extraAttack))
                + mapOf(Pair(evasion.name, evasion))
        ).toMutableMap()
```

Словарь mapOfAn непосредственно используется в приложении, остальные же словри нужны для удобства и распределения AbilityNode по файлам.

3.2. Заклинания и магические предметы

В DnD есть несколько сотен заклинаний и магических предметов. Каждое заклинание и каждый магический предмет обладают определёнными свойствами, которые необходимо учитывать. Информация о заклинаниях и о магических предметах была взята с сайта dungeon.su, а затем при помощи программ на питоне, написанных нами, эта информация была структурирована и переведена в json-формат. Так, например в исходном формате заклинание «Адское возмездие» выглядело так:

Адское возмездие [Hellish rebuke]

1 уровень, воплощение

Время накладывания: 1 реакция, совершаемая вами, когда ...

Дистанция: 60 футов

Компоненты: В, С

Длительность: Мгновенная

Классы: колдун

Архетипы: клятвопреступник (паладин)

Источник: «Player’s handbook»

Вы указываете пальцем, и существо, причинившее вам урон, ...

На больших уровнях. Если вы накладываете это заклинание, используя ячейку 2-го уровня или выше, урон увеличивается на 1к10 за каждый уровень ячейки выше первого.

После обработки оно стало выглядеть так:


```
{
"name": "Адское возмездие",

"engName": "Hellish rebuke",

"level": "1",

"text": "Вы указываете пальцем, и существо, причинившее вам урон, ..."

"school": "воплощение",

"castingTime": " 1 реакция, совершаемая вами, когда ...",

"range": "60 футов",

"materials": "",

"components": "В, С",

"duration": "Мгновенная",

"concentration": "нет",
```

"classes": *"колдун"*,

"sources": *"«Player's handbook»"*,
},

Заключение

В результате выполнения дипломной работы было создано рабочее приложение для игры в D&D пятой редакции.

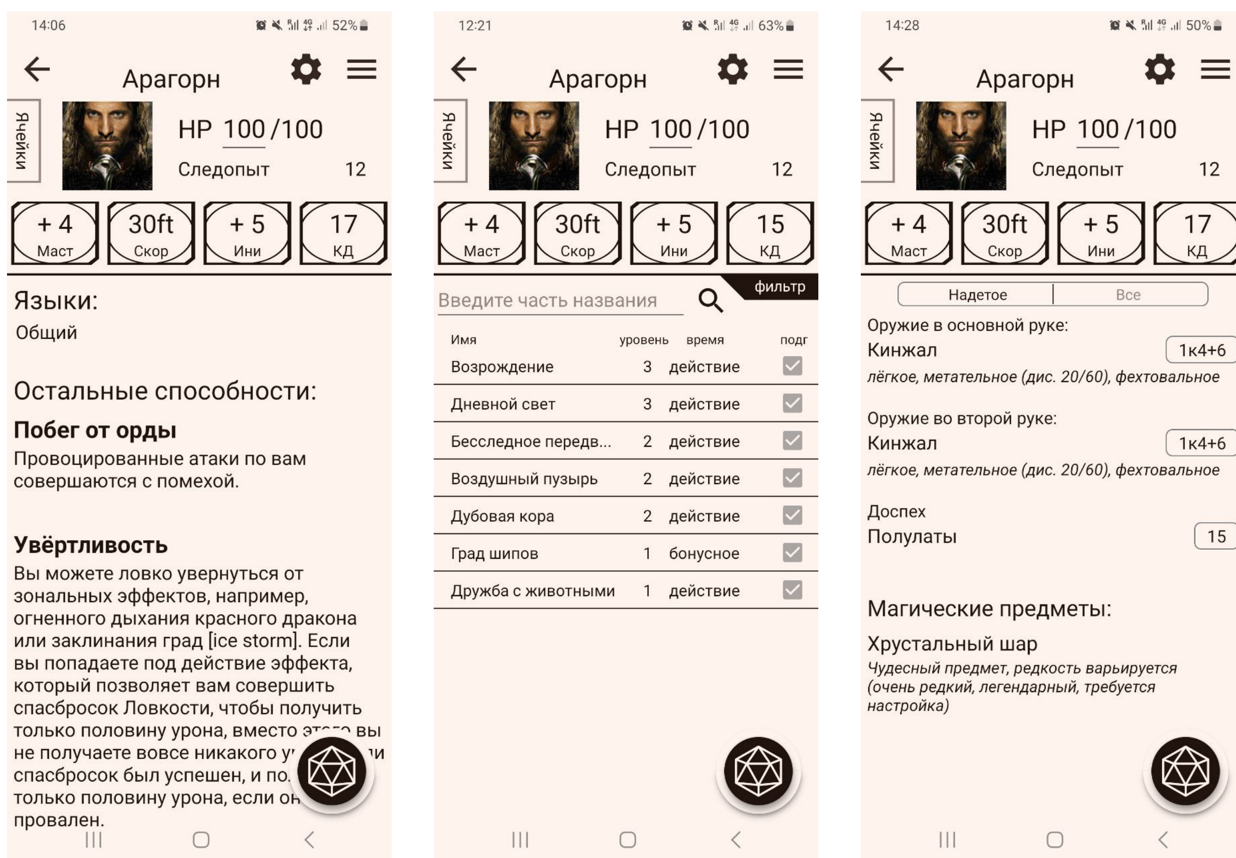

Рис. 6: Пример работающего приложения.

Несмотря на некоторые недостатки, наше приложение имеет ряд преимуществ перед аналогами: оно не требует интернета, мало весит и обладает поддержкой русского языка.

На данный момент выпущена тестовая версия приложения. Тестовую версию уже использует около 80 человек. Из-за авторских прав мы не можем сделать наше приложение платным, однако мы уже получили несколько добровольных пожертвований.

В будущем мы планируем добавить больше вариантов кастомизации (довольно часто в D&D играют, отступая от основных правил) и улучшить дизайн. Кроме того, если приложение будет пользоваться успехом, мы планируем сделать его аналоги для других систем настольных ролевых игр.

Список литературы

[1] Cyberleninka. URL: https://cyberleninka.ru/article/n/rolevye-igry-v-obuchenii-inostrannym-yazykam-teoriya-i-praktika (дата обр. 28.05.2023).

[2] Dice Cove. URL: https://dicecove.com/resources/statistics/ (дата обр. 28.05.2023).

[3] Fiction Horizon. URL: https://fictionhorizon.com/how-many-people-play-dd/ (дата обр. 28.05.2023).

[4] Kotlin, Data Class. URL: https://kotlinlang.org/docs/data-classes.html (дата обр. 28.05.23).

[5] Lambdas. URL: https://kotlinlang.org/docs/lambdas.html (дата обр. 28.05.23).

[6] DRY. URL: https://dzen.ru/a/Yuto6yBwtgLSi1fC (дата обр. 28.05.23).

[7] Github, Google. Gson library. URL: https://github.com/google/gson (дата обр. 28.05.23).