Министерство образования и науки Российской Федерации
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ (НИ ТГУ)
Факультет информатики
Кафедра программной инженерии (ПИ)

ДОПУСТИТЬ К ЗАЩИТЕ В ГЭК
Руководитель ООП
_____А.Л.Фукс
«_01_»_06_____2017 г

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА

РАЗРАБОТКА МОДЕЛИ ПАСЬЯНСОВ ТИПА “КОСЫНКА” И ЕЁ РЕАЛИЗАЦИЯ

по основной образовательной программе подготовки бакалавров
направление подготовки 02.03.02 – Фундаментальная информатика и информационные технологии

Арцимович Роман Александрович

Руководитель ВКР
доц. каф. программной инженерии
_____А.М.Бабанов
подпись
«_8_»_06_____2017 г.
Автор работы
студент группы № 1431
_____Р.А.Арцимович
подпись

Томск – 2017

РЕФЕРАТ

Дипломная работа: 61 с., 30 рис., 1 табл., 14 источников, 1 прил.

ПАСЬЯНС, КОСЫНКА, КАРТОЧНАЯ ИГРА, МОДЕЛЬ, UML, XML, XAML, WPF, VISUAL STUDIO, C#.

Объект исследования – объектная модель, реализованная в виде Windows-приложения.

Цель исследования – осуществить построение объектной модели карточной игры пасьянс «Косынка», интегрировать эту модель в программную оболочку, способную реализовать многообразие вариантов игры.

Метод исследования – теоретическое исследование и практическая реализация.

Область применения – приложение для ОС семейства Windows.

Результаты работы – детально изучены и проанализированы процессы и принципы игры в пасьянсах типа «Косынка», произведена систематизация результатов анализа, на основе которой разработана объектная модель игры, охватывающая многообразие вариантов ее реализации. Были изучены основные современные технологии разработки Windows-приложений – с их помощью осуществлена реализация приложения пасьянс «Косынка» с гибкими возможностями по реализации различных вариантов игры. Приведено детальное описание разработанной модели и приложения.

ОГЛАВЛЕНИЕ

РЕФЕРАТ 2
ОГЛАВЛЕНИЕ 3
ВВЕДЕНИЕ 5
1. ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ 6
1.1. Описание игры пасьянс "Косынка" 6
1.2. Модификации игры 7
1.3. Постановка задачи 8
2. ОПИСАНИЕ ТЕХНОЛОГИЙ МОДЕЛИРОВАНИЯ И РЕАЛИЗАЦИИ 9
2.1. Технология разработки WPF 9
2.2. Язык проектирования интерфейса приложения XAML 10
2.3. Язык программирования C# 11
2.4. Хранение данных с помощью XML 11
2.5. Проектирование программного обеспечения 11
2.6. СУБД SQLite 12
3. РАЗРАБОТКА МОДЕЛИ 13
3.1. Взаимодействие пользователя с игрой 13
3.2. Объектное проектирование 14
3.2.1. Основные объекты игры 14
3.2.2. Класс подготовки и управления игровым процессом – Game 26
3.2.3. Представление объектов в интерфейсе игры 29
3.2.4. Представление модификаций игры 30
3.2.5. Диаграмма пакетов 34
3.2.6. Диаграмма классов 35
3.3. Проектирование модели поведения 38
3.3.1. Общий алгоритм подготовки игры и игрового процесса 38
3.3.2. Диаграмма состояний класса Game 39
4. ПРОЕКТИРОВАНИЕ ПРИЛОЖЕНИЯ 40

4.1. Структура приложения 40
4.2. Проектирование компонентного состава приложения 41
4.2.1. Диаграмма компонентов приложения 41
4.2.2. Диаграмма компонентов проекта 42
5. РЕАЛИЗАЦИЯ ПРИЛОЖЕНИЯ 45
5.1. Проект интерфейса основного окна 45
5.2. Проект интерфейса вспомогательных окон 47
5.2.1. Окно настроек игры 47
5.2.2. Окно рекордов 48
5.2.3. Окно выбора игры 49
5.2.4. Окно описания правил игры 50
5.3. Проект интерфейсных компонентов объектов игры 51
5.3.1. Компонент карты 51
5.3.2. Компонент стопки 52
5.4. Реализация алгоритмов 53
5.4.1. Подготовка и составление списка модификаций 53
5.4.2. Раздача пасьянса 53
5.4.3. Инициализация классов стопок 54
5.4.4. Анимация перемещения карт 54
5.4.5. Манипуляции игрока 55
ЗАКЛЮЧЕНИЕ 57
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 58
Приложение А – Связи реализации вариантов использования Solitair20 компонентами (окнами) интерфейса приложения 59

ВВЕДЕНИЕ

Пасьянс – карточная игра для одного человека (от фрацн. Patience – терпение). Целью любого пасьянса ставится достичь некоторого расположения карт на игровом поле. При этом для пасьянса установлен набор правил, по которым можно карты перекладывать между стопками. Цель игры также определяется правилами конкретного пасьянса. Результат игры достигается с помощью интеллектуальных усилий играющего. При формировании начального состояния игры (расклада) присутствует фактор случайности, что вносит разнообразие в процесс игры и делает каждый сеанс уникальным.

Одной из разновидности карточных пасьянсов является пасьянс под названием «Косынка». Данный пасьянс традиционно поставляется в составе развлекательной составляющей операционной системы семейства Windows, начиная еще с версии Windows 3.0. Также существует множество разновидностей данного вида пасьянсов.

1. ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

1.1. Описание игры пасьянс "Косынка"

Пасьянс «Косынка» как компьютерная игра был разработан в 1989 году сотрудником корпорации Microsoft Уэсом Черри.

Классический вариант пасьянса «Косынка» имеет игровую колоду карт, состоящую из 52 карт – набор от двойки до туза в каждой масти. В начале игры 28 карт из стартовой колоды раскладывают в 7 игровых столбцов, при этом начальное количество карт в каждом из них соответствует номеру столбца (в первом - одна, во втором - две и т.д., в последнем, седьмом - семь карт). Карты в столбцах изначально закрыты (перевернуты «рубашкой» вверх), кроме самой верхней – ее кладут открытой.

Оставшиеся карты составляют колоду, которую используют для раскладки пасьянса с помощью переборов (по одной или по три карты). На рисунке 1.1 приведен исходный вид пасьянса сразу после раздачи в операционной системе Windows 98.

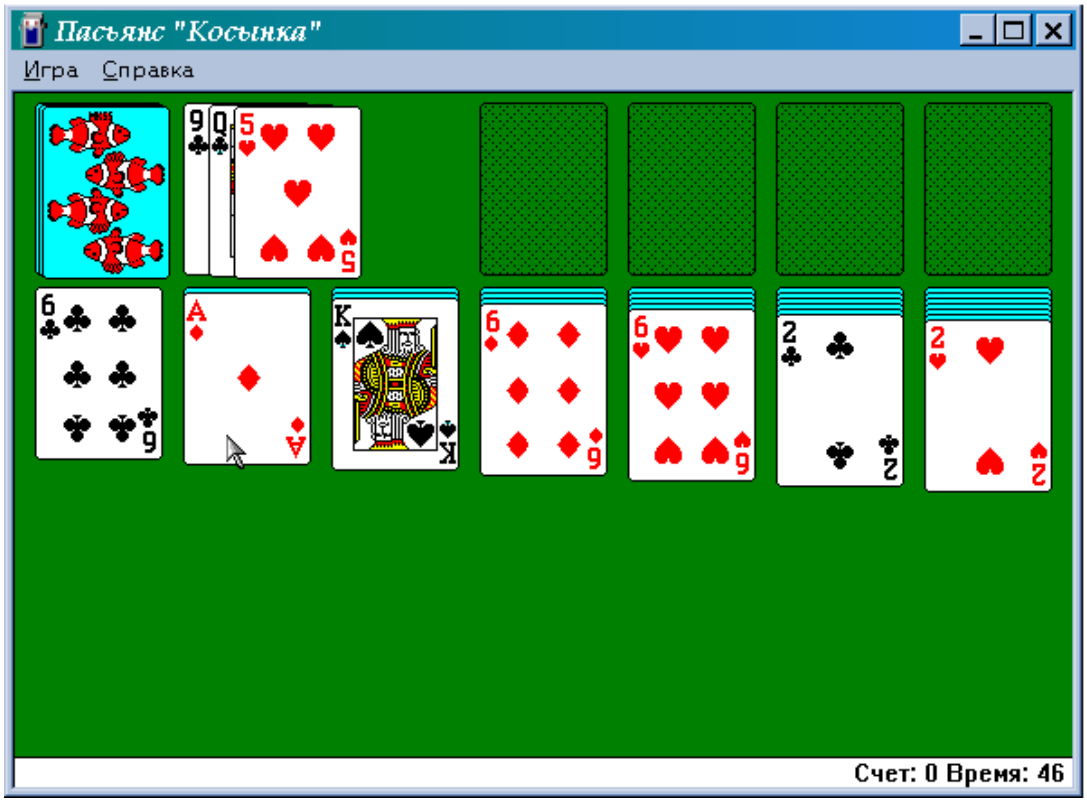

Рисунок 1.1 – Исходный вид пасьянса после раздачи.

В верхней части игрового поля помещаются целевые стопки, которые являются объектами основной цели игры. Игроку требуется разложить все выданные карты (манипулируя перемещениями карт в игровых стопках и стопке перебора и раздачи) в целевые стопки по возрастанию их достоинства - начиная от туза и заканчивая королём. Игрок, собравший все карты в четыре целевые стопки в нужном порядке, является победителем. В классической версии пасьянса по окончанию игры карты «разлетаются» по игровому полю.

В игре ведется простейший подсчет очков, основанный на времени, затраченном на получение победного расклада. Чем меньше затрачено времени на прохождение сеанса игры, тем больше очков получает игрок.

В классической игре игрок имеет дополнительные возможности, связанные с элементарной настройкой игрового процесса и внешнего вида игры: выбор цвета «рубашки» карт, выбор способа подсчёта очков (стандартные очки, виртуальные деньги, без подсчета), выбор способ сдачи карт из колоды раздачи (по одной или по трём), а также и включение или выключение отсчёта времени.

1.2. Модификации игры

Пасьянс «Косынка» в настоящее время имеет множество разновидностей, отличающихся друг от друга в различной степени по одному или нескольким параметрам. К таким параметрам можно отнести правила организации перемещений карт между стопками, количество карт и стопок, способы раздачи и перебора карт, и так далее.

В данной работе за основу для анализа был взят набор пасьянсов от Алексея Аношенко «Коллекция пасьянсов 250+». Данная коллекция включает 37 разновидностей пасьянса «Косынка». Анализ всех 37 разновидностей «Косынки» показал, что основные типы различий, дифференцирующие варианты игры, заключаются в:

- Количестве карт в игре.
- Количестве игровых и целевых стопок.
- Типе перебора карт в основной стопке раздачи.
- Правилах перемещений карт между стопками.
- Правилах размещении карт в игровых стопках.
- Начальных условиях раздачи.
- Наличии дополнительных стопок.
- Других мелких различиях.

1.3. Постановка задачи

В рамках данной работы ставятся следующие задачи:

1. Выявить и систематизировать различия в составе и поведении пасьянсов типа “Косынка”.
2. Обобщить все выявленные различия, выделить их описательные структуры и закономерности для разных вариантов игры.
3. Разработать общую объектную модель игры, охватывающую все отобранные варианты.
4. Разработать игровое приложение – пасьянс «Косынка» - версия Solitair20.

Варианты игры были выбраны из указанного ранее пакета «Коллекция пасьянсов 250+». Отбор вариантов обусловлен критериями, которые максимально широко охватывают степень различия вариантов игры и представляют наибольший интерес для реализации. В отобранные варианты вошли следующие типы игр:

«Косынка сброс по 1», «Косынка сброс по 3», «Косынка 2 сброс по 1», «Косынка 2 сброс по 3», «Тупики», «Гаргантюа», «Номер 10», «Звание и досье», «Красный и черный», «Палец и мешок», «Уайтхед», «Двойная косынка 1», «Двойная косынка 2», «Королева», «Родж эт Наир», «Сумасшедший брод», «Сумасшедший брод 2», «Западный обрыв», «Шаги», «Спидайк», «Австралийский пасьянс», «Аляска».

2. ОПИСАНИЕ ТЕХНОЛОГИЙ МОДЕЛИРОВАНИЯ И РЕАЛИЗАЦИИ

В данной главе описаны технологии, которые применяются при разработке приложения Solitair20. Применяемые технологии включают:

- WPF – основная концепция создания приложения;
- XAML – вспомогательный язык системы WPF, предназначенный для проектирования интерфейса приложения;
- C# - объектно-ориентированный язык программирования, применяемый для создания функциональной логики приложения;
- XML – язык разметки, описывающий в данном применении конфигурации вариантов игры;
- UML – язык унифицированного моделирования, используемый при проектировании архитектуры и поведения программного обеспечения.
- SQLite – портативная (компактная) СУБД, реализующая информационное обеспечение приложения;

2.1. Технология разработки WPF

Технология WPF (Windows Presentation Foundation) предсатавляет собой принципиально новый (по сравнению с GDI, Windows Forms) подход к созданию приложений. WPF – это подход к разработке программного обеспечения на базе принципа «всем миром» [2, с.13]. Технология WPF сумела объединить в себе документы, формы и мультимедийный контент в единый пакет, использующий принципы императивного и декларативного программирования. Декларативное программирование основывается на языке разметки интерфейса XAML, императивное – на базе современных языков программирования – C# или Visual Basic.

В графической основе классических приложений Windows Forms состоит GDI/GDI+ (Graphical Device Interface), являющийся компонентом, который в тесном взаимодействии с ядром API Windows составляет базовый пользовательский интерфейс программ. GDI может быть применен в простых игровых приложениях, не требующих сложной графики и высокой степени динамичности, однако его нельзя применять для качественного выполнения сложной и быстрой анимации, поскольку он не поддерживает синхронизацию с кадровым буфером (участком в памяти, хранящем кадры перед их непосредственным выводом на устройство).

Технология WPF в своей графической основе использует DirectX, что повышает производительность WPF-приложений за счет возможности применения аппаратного ускорения графики. Пользовательский интерфейс приложений, использующих WPF, построен на базе векторной графики, что дает независимость от разрешения конечного устройства. Основные преимущества графической подсистемы WPF:

- Независимость от разрешения экрана вывода и устройства – единица измерения графики в WPF – это аппаратно-независимый пиксель, равный 1/96 части дюйма и не зависящий от разрешения экрана.
- Поддержка эффектов анимации объектов.
- Использование аппаратного ускорения.
- Встроенная проработанная библиотека 2-D и 3-D объектов.

2.2. Язык проектирования интерфейса приложения XAML

XAML (eXtesible Application Markup Language) представляет собой расширяемый язык разметки приложений – декларативный язык проектирования пользовательского интерфейса. Структура языка XAML основывается на XML.

Язык XAML является всего лишь вспомогательным средством проектирования и компоновки интерфейса приложения, его использование вообще не обязательно, однако, как показывает практика, применение XAML в проектах разработки дает свои несомненные преимущества:

- Более наглядное представление структуры интерфейса проектируемого приложения, обеспечиваемое благодаря иерархичности языка разметки.
- Разделение визуальной и функциональной частей приложения, что позволяет разделить разработку на отдельно программистов и отдельно дизайнеров.
- Компактность и высокая степень читаемости и поддержки кода, написанного на XAML – как правило, код, написанный на языке разметки, гораздо меньше по объему эквивалентного кода, написанного на процедурном языке.
- Поддержка применения стилей интерфейса.

XAML в основном используется для компоновки макета форм приложения и инициализации обработчиков событий. Сами же обработчики должны быть запрограммированы на процедурном языке (таком, как, например, C#). Такая концепция позволяет осуществлять проект дизайна приложения с одной стороны, пока с другой стороны программист будет наполнять функциональную логику, также основанную на взаимодействии компонентов интерфейса.

2.3. Язык программирования C#

C# - один из самых мощных, быстро-развивающихся и востребованных языков программирования. C# является объектно-ориентированным, он поддерживает в полной мере полиморфизм, механизмы инкапсуляции и наследования, перегрузку операторов, лямбда-выражения и многие другие современные возможности быстрой и качественно разработки.

Язык C# является основным средством реализации технологии .NET. Основой фреймворка .NET является применяемая в рамках него общеязыковая исполняющая среда – CLR (Common Language Runtime). Код, выполняемый в CLR, является управляемым. CLR умеет автоматически управлять памятью и имеет сборщик мусора, который выполняется совместно с программой и восстанавливает память, которая занята объектами, на которые больше нет ссылок. Таким образом, отпадает необходимость контроля за своевременным освобождением памяти, что необходимо при программировании, например, на C++.

Цель языка C# - обеспечение продуктивности работы программистов. Для этого в языке соблюдается баланс между простотой, выразительностью и производительностью.

2.4. Хранение данных с помощью XML

Язык проектирования XML создавался как универсальный язык разметки, применяемый в различных областя.

В приложении Solitair20 язык XML используется для хранения структуры данных, определяющих разновидности пасьянса, таким образом реализуя процесс сериализации – преобразования объекта (структуры игры) в форму, подготовленную для эффективного использования в логике программы.

2.5. Проектирование программного обеспечения

UML (Unified Modelling Language) – унифицированный язык моделирования, который использует графическое описание проектируемой системы и позволяет построить статическую и динамическую модели программного продукта с точки зрения его: логического представления, представления реализации, представления процесса функционирования и представления размещения и состава компонентов.

На рынке программного обеспечения представлено множество продуктов (CASE-средств), которые предназначены для создания UML-моделей приложений. К наиболее известным можно отнести Rational Rose, Power Designer, StarUML,

Enterprise Architect, и другие. Все эти продукты полностью поддерживают спецификацию UML 2.0 и обладают мощным инструментарием.

В рамках данной работы для создания моделей приложения Solitair20 применено CASE-средство, представленное программой Visual Paradigm for UML2.0 версии 7.2 (VP). Выбор был сделан в пользу данного продукта, так как он обладает наибольшим удобством среди всех рассмотренных. Кроме того, VP поддерживает средства forward engineering, что позволяет сгенерировать файлы шаблонов исходного кода, исходя из построенной модели (например, диаграммы классов).

2.6. СУБД SQLite

Для хранения информации о рекордах сеансов игр разных типов разрабатываемый продукт будет использовать локальное хранилище, исполненное в виде файла базы данных. Для таких решений идеально подходит СУБД SQLite со своими возможностями.

SQLite – компактная встраиваемая реляционная база данных, которая имеет исходные коды в открытом доступе. Встраиваемая – означает, что SQLite не использует устоявшуюся парадигму «клиент-сервер», то есть ядро SQLite не является отдельно работающим процессом, с которым взаимодействует программа, а представляет собой библиотеку, с которой программа компонуется, а само ядро становится частью программы.

Основные преимущества СУБД SQLite, которые определили ее выбор в качестве применяемого хранилища в данном проекте, описаны в [6, с.23] и определяются следующими характеристиками:

- Отсутствие сервера.
- Отсутствие необходимости конфигурирования и установки СУБД.
- Кроссплатформенность.
- Интеграция ядра СУБД в сам файл базы данных.
- Высокая надежность.
- Производительность.

3. РАЗРАБОТКА МОДЕЛИ

В данной главе представлены основные аспекты, которые использованы при проектировании модели будущего приложения Solitair20. Здесь представлены диаграммы UML, характеризующие основные стороны проекта приложения – статическую, объектную и поведенческую.

Диаграммы выполнены в системе VP, а в качестве иллюстраций приведены фрагменты копий экрана этой системы.

3.1. Взаимодействие пользователя с игрой

Взаимодействие программной системы с внешней средой (пользователями) описывается диаграммами вариантов использования (прецедентов). В [4, с. 93] диаграмма вариантов использования определена как модель, представляющая разрабатываемую программу как систему в виде совокупности вариантов ее использования пользователем. На этой диаграмме показывается внешняя сторона системы (программы), не учитывая ее внутреннюю структурную и архитектурную составляющую. Диаграмма должна отображать все предоставляемые программой связи с пользователем.

Диаграмма вариантов использования приложения Solitair20 приведена на рисунке 3.1.

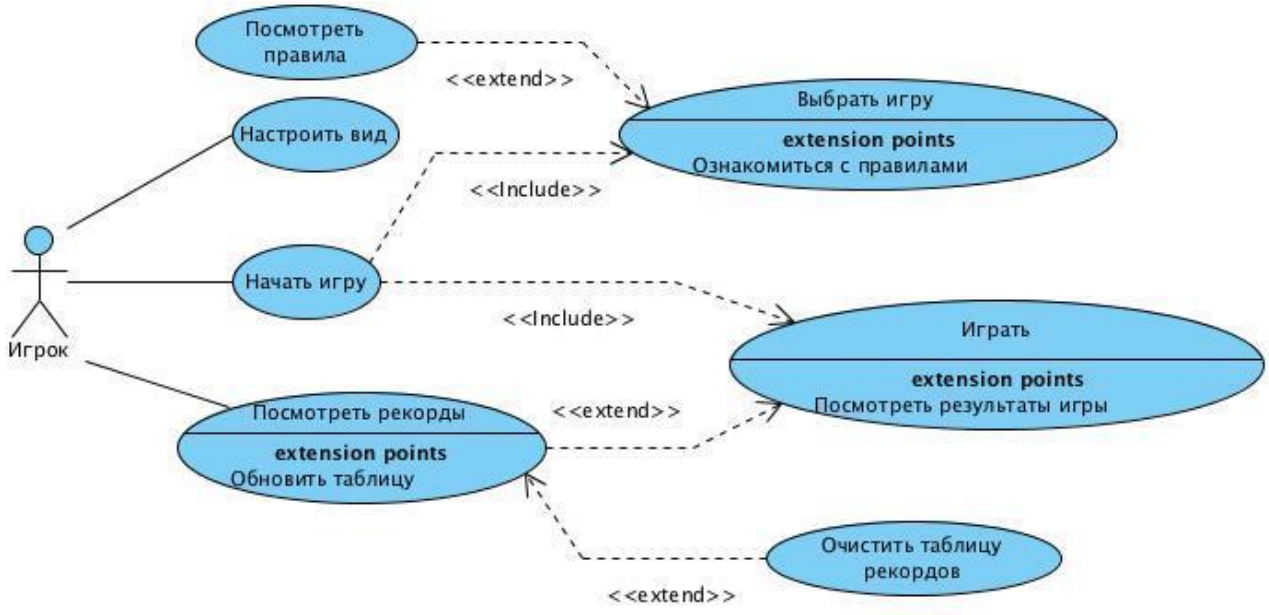

Рисунок 3.1 – Диаграмма вариантов использования.

Из диаграммы 3.1 прослеживаются основные действия, которые игрок может выполнять в программе:

- Настройка внешнего вида отображения в игре.
- Начало игры, которое сопровождается предварительным выбором типа игры и самим игровым процессом (манипуляциями с картами с целью получения выигрышной комбинации).
- Просмотр таблицы рекордов, достигнутых по каждому виду игры.

Из топологии диаграммы прецедентов видно, что:

- Таблица рекордов может быть вызвана в любой момент игры, а также вызывается после каждого завершенного сеанса игры.
- Таблица рекордов может быть очищена.
- В процессе выбора игры пользователь может просмотреть правила типа игры.

Прецеденты реализуются в программе посредством экранных форм пользовательского интерфейса. В приложении А показана связь каждого варианта использования с окном интерфейса, реализующим этот вариант

3.2. Объектное проектирование

3.2.1. Основные объекты игры

Основной этап объектного проектирования приложения заключается в выделении объектов предметной области, установлении их основных свойств и связей.

Пасьянс – карточная игра, поэтому логично, чтобы основными объектами этой игры стали карты. Карты в игре сами по себе не существуют, а объединяются в стопки. Таким образом, карты и стопки становятся основными объектами приложения, связанными отношениями агрегации.

Исследования различных вариантов игры показали, что в игре может существовать до 5 видов стопок, имеющих различную функциональность и назначение. В то же время все типы стопок имеют много общего. На рисунке 3.2 показан исходный расклад варианта игры «Сумасшедший брод» - эта разновидность «Косынки» характеризуется наличием всех типов стопок (в соответствии с обозначениями на рисунке):

1. Целевые стопки (***Task Decks***) – стопки, в которые в конечном итоге собираются все карты с игрового поля. Присутствуют во всех разновидностях игры.

2. Игровые стопки (***Game Decks***) – вертикальные ряды, которые являются основными для совершения промежуточных игровых манипуляций. Игровые стопки определены для всех разновидностей игры.
3. Стопка раздачи (***Distribution Deck***) – основная, первоначальная стопка, изначально содержит все карты игры, определенное количество которых при задании исходного расклада перемещаются в другие стопки в соответствии с правилами разновидности пасьянса. Стопка присутствует во всех разновидностях игры, однако, в некоторых все карты этой стопки раздаются в игровое поле, делая эту стопку пустой во время игры.
4. Стопка перебора (***Bust Deck***) – вспомогательная стопка, в которую карты попадают из стопки раздачи по определенному правилу. Стопка присутствует не во всех разновидностях игры.
5. Дополнительная стопка (***Additional Deck***) – еще одна не обязательная стопка, которая является неким буфером в игре и определяется установленным правилами поведением.

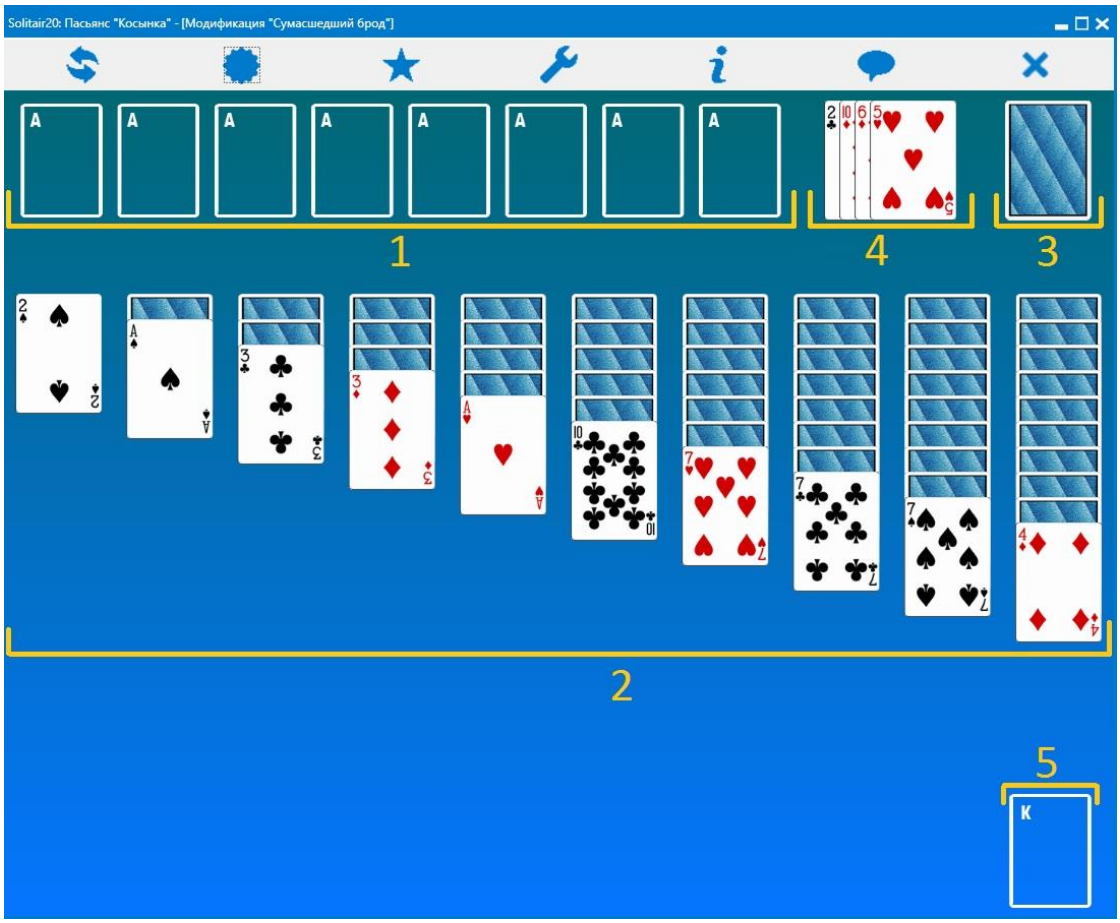

Рисунок 3.2 – Схема стопок в игре.

Объект, представляющий карту в игре, характеризуется двумя основными параметрами – масть (Suit) и достоинство (Advantage). Средствами языка программирования масть карты может быть задана как нумератор, имеющий 4 значения (см. листинг 3.1).

Листинг 3.1 – Определение нумератора масти карты.

```
    public enum Suit
    {
        Clubs,           // крестовая
        Hearts,          // червовая
        Spades,          // пиковая
        Diamonds         // бубновая
    }
```

Аналогично задается нумератор достоинства карты, который имеет 13 значений (см. листинг 3.2).

Листинг 3.2 – Определение нумератора достоинства карты.

```
public enum Advantage
    {
        Any = 0,         // не важна (любая)
        Two = 2,         // 2
        Three,           // 3
        Four,            // 4
        Five,            // 5
        Six,             // 6
        Seven,           // 7
        Eight,           // 8
        Nine,            // 9
        Ten,             // 10
        Jack,            // валет
        Queen,           // дама
        King,            // король
        Ace              // туз
    }
```

Таким образом, указанные нумераторы являются неотъемлемыми атрибутами карты – свойствами соответствующего класса карты (Card). Класс объекта карта, помимо значимых характеристик карты, также должен содержать дополнительные свойства, такие как:

- Тип отображения на игровом поле («рубашкой» вверх или вниз – face);
- Визуальную компоненту - ui;
- Настройку отображения (тип или текстуру «рубашки») – cardBack;
- Ссылку на стопку, в который находится карта – myDeck.

Помимо конструктора, в классе Card необходимо предусмотреть метод, который будет «переворачивать» карту в стопке. Таким методом является метод filpCard().

Предварительный каркас класса карты приведен в листинге 3.3.

Листинг 3.3 – Каркас класса карты.

```
public class Card
{
    // =true - рубашкой вниз
    private bool face;
    // масть
    private Suit suit;
    // достоинство
    private Advantage advantage;
    // визуальная компонента
    private UICard ui;
    // колода, в которой карта находится в данный момент
    private Deck myDeck;
    // тип выбранной рубашки карты
    private int cardBack = 7;
    // конструктор --------------------------------
    public Card(Suit suit, Advantage advantage, bool face)
    // перевернуть карту
    public void filpCard()
}
```

Объекты из множества типов «стопка» обладают более расширенными свойствами, определяющими их различия и виды. Одновременно с этим, во всех объектах этого типа можно выделить много общих признаков и свойств. Поэтому должен создан быть некий генеральный абстрактный класс стопки (Deck), который будет «порождать» все типы стопок в игре на этапе инициализации игры, реализация каждой из которых будет происходить посредством переопределения свойств и методов, наследуемых от генерального класса. Генеральный класс должен предоставлять своим потомкам такие свойства, как:

- Количество карт – count;
- Набор правил поведения – deckRules;
- Ориентация на игровом поле – orientation;
- Визуальная компонент – ui;
- Набор карт – cards, который лучше всего представить структурой строго типизированных объектов Card, коллекцией List<Card>, предоставляемой в .NET.

Ориентация стопок на игровом поле может реализовываться в виде ряда, ориентированного горизонтально или вертикально, либо в виде одной стопки,

отображающей одновременно одну карту (верхнюю). В коде программы ориентация стопки задается соответствующим нумератором (см. листинг 3.4).

Листинг 3.4 – Нумератор ориентации стопки.

```
    // Способ размещения колоды (стэка)
    public enum PlacingMethod
    {
        TopOnlyVisible,  // видна только верхняя
        HorizontalLine,  // в горизонтальную линию
        VerticalLine     // в вертикальную линию
    }
```

Обобщив результаты исследования вариантов реализации «Косынки», был выделен набор правил поведения, которыми может быть охарактеризован любой тип стопки. Данный набор правил можно представить структурой нумераторов. Правила поведения стопок включают одно правило взятия карт из стопки и 4 правила перемещения карт в стопку. Листинг 3.5 иллюстрирует структуру правил поведения стопки.

Листинг 3.5 – Структура правил поведения стопки.

```
    // Структура: основные правила колоды
    public struct DeckRules
    {
        public TakeFromRule takeFromRule; // правило взятия карт
        // правило помещения карт в колоду - порядок
        public TopPlaceRuleOrder topPlaceRuleOrder;
        // правило помещения карт в колоду - последовательность
        public TopPlaceRuleSequence topPlaceRuleSequence;
        // правило помещения карт в колоду - количество
        public TopPlaceRuleMultiplicity topPlaceRuleMultiplicity;
        // правило помещения карты в самый низ (достоинство первой карты)
        public Advantage bottomCard;
    }
```

Правило взятия карт из стопки определяет, какие карты могут быть перемещены из данной стопки в другую. Предпроектные исследования показали, что различные варианты реализации «Косынки» дают игроку право перетаскивать между стопками как одну карту, так и целые стопки (или части стопок), строго регламентируя последовательность расположения карт в перемещаемой стопке – устанавливая ограничения на чередование по цвету масти, достоинству, порядок сортировки, и т.д. В листинге 3.6 приведен нумератор соответствующего правила.

Листинг 3.6 – Структура правил поведения стопки.

```
    // ПРАВИЛО СТОПКИ - взятие карт из стопки
    public enum TakeFromRule
    {
        Forbidden,                // запрещено
        OneCard,                  // только одну сверху
        AnySequence,              // любую последовательность
        ColorAlternation,         // последовательность с чередованием цвета (порядок не
                                     важен)
        SingleColor,              // последовательность одного цвета (порядок не важен)
        SingleSuit,               // последовательность одной масти (порядок не важен)
        ColorAlternationArranged, // последовательность с чередованием цвета (по порядку
                                     достоинства)
        SingleColorArranged,      // последовательность одного цвета (по порядку
                                     достоинства)
        SingleSuitArranged        // последовательность одной масти (по порядку
                                     достоинства)
    }
```

Следующие 4 правила, наоборот, определяют возможность переместить карты, взятые из другой стопки, в данную.

Правило количества. Данное правило определяет количество карт, которое может быть одновременно перемещено в данную стопку. Возможные варианты: одну, несколько, либо запрет на перемещение. Последний вариант, например, применим во всех вариантах игры к стопке раздачи, поскольку в стопку раздачи нельзя перемещать карты из других стопок. Код нумератора по данному правилу приведен в листинге 3.7.

Листинг 3.7 – Нумератор правила по количеству

```
    // ПРАВИЛО СТОПКИ - положить карты на верх стопки - количество
    public enum TopPlaceRuleMultiplicity
    {
        None,    // запрещено
        OnlyOne, // можно только одну
        Deck     // можно стопку
    }
```

Правило порядка сортировки. Данное правило устанавливает порядок сортировки в стопке карт, которая может быть перемещена в данную. Данное правило применимо только к стопкам, в которых установлено TopPlaceRuleMultiplicity = Deck. Вариантов порядка сортировки может быть два – восходящий (от двойки до туза), либо нисходящий (от туза до двойки). Также предусмотрен вариант, позволяющий игнорировать порядок сортировки. В листинге 3.8 приведен исходный код соответствующего нумератора.

Листинг 3.8 – Нумератор правила порядка сортировки.

```
// ПРАВИЛО СТОПКИ - положить карты на верх стопки - порядок
public enum TopPlaceRuleOrder
{
    // нижняя карта накладываемой стопки должна быть
    Any,     // любой
    TwoAce,  // быть по порядку восходящей (с 2 до туза)
    AceTwo   // быть по порядку нисходящей (с туза до 2)
}
```

Правило последовательности. Данное правило накладывает ограничения на формирование последовательности карт в данной стопке с учетом накладывания в нее сверху другой карты (последовательности карт). Такое ограничение позволяет формировать стопку с определенной последовательностью карт (одной масти или одного цвета масти, с чередованием по масти, цвету масти и т.д.). Также предусмотрена возможность игнорирования последовательности. Код соответствующего нумератора приведен в листинге 3-9.

Листинг 3.9 – Нумератор правила последовательности.

```
// ПРАВИЛО СТОПКИ - положить карты на верх стопки - последовательность
public enum TopPlaceRuleSequence
{
    // нижняя карта накладываемой стопки должна быть
    Any,                // не имеет значения
    SameSuit,           // той же масти
    SameColor,          // того же цвета
    SameAdvantage,      // того же достоинства
    AlternativeColor,   // другого цвета
    AlternativeSuit     // другой масти
}
```

Правило нижней карты. Определяет тип карты, которая может быть помещена в начало пустой стопки. Определена, например, для целевых стопок, т.к. эти стопки начинаются со строго определенной (по достоинству) карты. Данное правило представлено нумератором достоинства карты Advantage, предусматривающим также игнорирование этого правила при Advantage = Any.

Поскольку игровой процесс подразумевает постоянное перемещение карт из между стопками разных типов и назначения, то следует базовый класс стопки снабдить методами, контролирующими события проведения пользовательских манипуляций:

- событие попытки захватить стопку (или ее часть) для перемещения – tryCaptureToMove.

- событие «отпускания» стопки, определяющее либо возврат захваченных карт туда, откуда взяли, либо успешное перемещение их в другую стопку - releaseMoving.

Захвата карт стопки для их перемещения определяется правилами стопки deckRules.takeFromRule, но это не всегда возможно. Поэтому, прежде чем разрешать захват карт в стопке, необходимо сопоставить содержимое последовательности захватываемых карт правилам стопки. За это сопоставление отвечает метод CanCardsMove. Этот метод должен быть вызван в теле метода tryCaptureToMove для проверки возможности перемещения карт. Вызовы методов захвата и отпускания карт стопок должны осуществляться на уровне отлавливания событий в компонентах пользовательского интерфейса.

Каркас класса генеральной колоды приведен в листинге 3.10.

Листинг 3.10 – Каркас класса генеральной колоды.

```
public abstract class Deck
{
    private int count;                  // кол-во карт
    private DeckRules deckRules;        // правила
    private PlacingMethod orientation; // ориентация
    private UIDeck ui;                  // визуальная компонента
    private List<Card> cards;           // карты стопки
    // Попытка переместить часть стопки
    public void tryCaptureToMove(Card topCard)
    // Функция определения возможности перемещения карт стопки
    // Определяется на основе правил стопки в DeckRules
    private bool CanCardsMove(List<Card> cards)
    // Завершение перемещения стопки
    public void releaseMoving()
}
```

Колода раздачи является основной в игре. Именно она должна быть определена и наполнена в самом начале подготовки к игре и осуществлении раздачи исходного расклада пасьянса. Поведение этой колоды в основном определяется вышеописанными правилами. В дополнение к ним колода раздачи обладает еще уникальными свойствами и методами.

К числу уникальных свойств относятся:

- Количество разрешенных переборов, или пересдач (через колоду перебора) - bustCount;
- Буфер перемещения карт – специальный буфер, через который будет организована анимация перемещения карт из колоды раздачи в другие колоды - List<SendCardsInfo> bufferToSend;

- Правило поведения колоды по действию пользователя (клике мышью) clickBehavior – данное правило определяет, будут ли по клику игрока карты отправлены в колоду перебора, или же в игровые колоды. Соответствующий нумератор приведен в листинге 3.11.

Листинг 3.11 – Нумератор правила поведения по клику колоды раздачи.

```
// Поведение колоды раздачи при клике
public enum DDeckClickBehavior
{
    AddToBustDeck, // добавить карты в колоду перебора
    AddToGameDeck  // добавить в игровые колоды
}
```

Класс колоды раздачи имеет уникальные методы:

- Инициализации: Initialize – в методе генерируется и раскладывается общее число карт, определенных конкретной разновидностью игры, переопределяются наследуемые правила стопки.
- Перетасовывание карт: Shuffle – метод, привносящий элемент случайности в игру.
- Методы перебора стопки: ReBust, SendCardsToBustDeck – определены для реализации возможности определенного количества переборов стопки раздачи через стопку перебора.
- Методы, отвечающие за анимацию и логику перемещения карт: SendCardsToBustDeck, SendCardsToGameDecks, SendNextCard.

В листинге 3.12 приведен каркас класса DistributionDeck – колоды раздачи.

Листинг 3.12 – Каркас класса колоды раздачи.

```
public class DistributionDeck : Deck
{
    // Список карт, которые переносятся из одной стопки в другую
    public struct SendCardsInfo
    {
        public Deck destination;
        public bool turn;
    }
    public List<SendCardsInfo> bufferToSend = new List<SendCardsInfo>();
    // правила поведения при действии
    private DDeckClickBehavior clickBehavior;
    // кол-во разрешенных пересдач
    private int bustCount;
    // инициализировать
    private void Initialize(int count)
    // перетасовать
    private void Shuffle(byte depth)
    // конструктор --------------------------------
    public DistributionDeck(int volume,DeckRules dr, DDeckClickBehavior ddcb, int bc)
```

```
        // добавление карт в стопку перебора
        public void SendCardsToBustDeck()
        // ----------------------------------------
        // отправка очередной карты в стопку (для анимации)
        public void SendNextCard()
        // перебор стопки
        private void ReBust()
        // добавление карт в игровые стопки
        public void SendCardsToGameDecks()
        // перемещение заданной карты на верх стопки
        public void MoveCardOnTop(int imc)
    }
```

Колода перебора представляет собой своеобразный буфер в игре. В дополнение к вышеописанным правилам, эта колода имеет еще и уникальные свойства, и методы:

- Свойство additionalCount – определяет количество карт, которое будет отправлено из колоды раздачи в колоду (колоды) назначения. В качестве колод назначения может быть колода перебора или игровые стопки – определяется свойством DDeckClickBehavior колоды раздачи.
- Метод инициализации: Initialize – реализует переопределение структуры правил поведения DeckRules, унаследованной от базового класса Deck.

Листинг 3.13 – Каркас класса колоды перебора.

```
    public class BustDeck : Deck
    {
        // количество карт, добавляемых из колоды раздачи за 1 раз
        private int additionalCount;
        // инициализировать
        private void Initialize(int count)
        // конструктор --------------------------------
        public BustDeck(int volume, DeckRules dr, int ac)
    }
```

Целевые колоды являются основными в игре, для которых анализируется состав для определения победного расклада. Для целевых колод жестко определено свойство DeckRules.bottomCard, входящее в структуру правил поведения – это означает, что данная колода должна начинаться с карты, строго соответствующей по достоинству свойству DeckRules.bottomCard. При этом некоторые модификации пасьянса требуют при раздаче расклада помещения указанной карты из колоды раздачи в целевую стопку сразу в начале игры. За это требование отвечает свойство initialCard, определяющее: должна ли в стопке лежать первая карта в начале игры или нет. При этом у разных целевых стопок это значение может отличаться.

Например, модификация «Спидайк» требует наличия «верхней» карты только в одной целевой стопке из четырех, а модификация «Тупики» – 4 из 4.

Целевая стопка также имеет метод инициализации Initialize, который реализует переопределение структуры правил поведения DeckRules, унаследованной от базового класса Deck.

Листинг 3.14 – Каркас класса целевой колоды.

```
    public class TaskDeck : Deck
    {
        // начальная карта должна лежать в колоде сразу при раздаче
        private bool initialCard;
        // инициализировать
        private void Initialize(int count)
        // конструктор --------------------------------
        public TaskDeck(DeckRules dr, bool initialCard)
    }
```

Игровые колоды предназначены для выполнения основных манипуляций перекладки карт в игре с целью их упорядочивания в требуемые последовательности. Количество игровых стопок может варьироваться в зависимости от конкретного варианта игры. Также может варьироваться и количество карт (с указанием их положения – «рубашкой» вверх или вниз) в каждой из игровых стопок.

Игровая стопка также имеет метод инициализации Initialize, который реализует переопределение структуры правил поведения DeckRules, унаследованной от базового класса Deck.

Листинг 3.15 – Каркас класса игровой колоды.

```
    public class GameDeck : Deck
    {
        // инициализировать
        private void Initialize(int count)
        // конструктор --------------------------------
        public GameDeck(DeckRules dr)
    }
```

Дополнительная колода является дополнительным карточным буфером в игре. Эта стопка встречается только в модификациях «Сумасшедший брод» и «Сумасшедший брод 2». Правила данной стопки разрешают помещать в нее только королей. Это значит, что структура, инициализирующая правила данной стопки будет выглядеть, как показано в листинге 3.16. В листинге 3.17 приведен исходный код каркаса класса AdditionalDeck.

Дополнительная стопка также имеет метод инициализации Initialize, который реализует переопределение структуры правил поведения DeckRules, унаследованной от базового класса Deck.

Листинг 3.16 – Исходная структура правил дополнительной стопки.

```
    takeFromRule = TakeFromRule.OneCard;
    topPlaceRuleOrder = TopPlaceRuleOrder.Any;
    topPlaceRuleSequence = TopPlaceRuleSequence.SameAdvantage;
    topPlaceRuleMultiplicity = TopPlaceRuleMultiplicity.OnlyOne;
    bottomCard = Advantage.King;
```

Листинг 3.17 – Каркас класса дополнительной колоды.

```
    public class AdditionalDeck : Deck
    {
        // инициализировать
        private void Initialize(int count)
        // конструктор --------------------------------
        public AdditionalDeck(DeckRules dr)
        // конструктор --------------------------------
    }
```

Игровой процесс сопровождается постоянным перекладыванием карт из одной стопки в другую, при этом перемещать можно как отдельно карты, так и целые стопки (части стопок). Поэтому в структурную схему объектов целесообразно ввести дополнительный объект – ***буферную стопку*** – стопку, которая будет появляться только в моменты проведения игровых манипуляций и будет содержать набор перемещаемых карт. Очевидно, что буферная стопка будет всегда вертикально-ориентированной, то есть свойство orientation = PlacingMethod.VerticalLine.

Буферная стопка, как и все остальные, должна иметь метод инициализации Initialize, который реализует переопределение структуры правил поведения DeckRules, унаследованной от базового класса Deck.

Буферная стопка в любом случае создается как часть некоторой другой стопки, из которой происходит перемещение карт, поэтому для реализации логики перемещений в буферной стопке необходимо хранить ссылку на бывшего владельца находящихся в ней карт – эта информация будет храниться в свойстве source.

Буферная стопка будет свободно перемещаться по игровому полю посредством манипуляций игрока с помощью мыши. При этом, чтобы позиционировать и отображать перемещаемые карты корректно, необходимо учитывать место захвата этой стопки кнопкой мыши еще в тот момент, когда она

не была отделена от основной стопки source. Это место определяется свойством (точкой) offsetPoint. Также необходимо наличие дополнительного метода, который будет управлять положением стопки при ее перемещении по игровому полю. За эту операцию будет отвечать метод UpdateLocalPosition. В листинге 3.18 приведен прототип класса буферной стопки.

Листинг 3.18 – Прототип класса буферной стопки.

```
public class BufferedDeck : Deck
{
    // исходная стопка, из которой выполняется перемещение
    Deck source;
    // Точка смещения - точка захвата карты в стопке при перемещении
    Point offsetPoint;
    // инициализировать
    private void Initialize(int count)
    // конструктор --------------------------------
    public BufferedDeck(DeckRules dr)
    // конструктор --------------------------------
    // Обновление позиции буферного стека на поле
    public void UpdateLocalPosition(Point updatePoint)
}
```

3.2.2. Класс подготовки и управления игровым процессом – Game

Весь приведенный в п. 3.2.1 состав объектов игрового процесса еще не представляет собой законченную модель игры, поскольку приведенные классы стопок и карт не могут реализовать сам процесс игры в динамике. Они позволяют всего лишь задать исходные данные игры: основные правила расположения, состава игровых объектов, правила манипуляции – базу, отличающую возможные разновидности реализации игры.

Очевидна необходимость в наличии некоторого класса, который будет обслуживать весь игровой процесс. Таким классом является класс Game. Основные задачи этого класса:

- ***Располагать полной актуальной информацией об объектах текущей игры***.

  Класс Game должен быть создателем и владельцем ссылок на все колоды, доступные в игре:

  - ✓ список игровых колод List<GameDeck> __GD;
  - ✓ список целевых колод List<TaskDeck> __TD;
  - ✓ колода раздачи __DD;
  - ✓ колода перебора __BD;
  - ✓ дополнительная колода __AD;
  - ✓ буферная колода __MD.

Все эти колоды создаются и параметрируются в данном классе.

- ***Подготовить, создать, проинициализировать и завершить игру***.
  В рамках данной задачи происходит инициализация базовой конфигурации игры, определение номенклатуры объектов, участвующих в игровом процессе. За процесс инициализации и подготовки отвечает метод Prepare. Метод newPlayOut осуществляет процессы подготовки объектов игры, руководствуясь установленными правилами конкретного варианта игры, прописанными в соответствующем XML-файле, в методе происходят установки правил поведения всех колод и такие частные параметры, как:
  - ✓ TaskDeckCount – количество целевых стопок в игре;
  - ✓ GameDeckCount – количество игровых стопок в игре;
  - ✓ IsBustDeckExist – наличие или отсутствие в игре колоды перебора;
  - ✓ IsAdditionalDeckExist – наличие или отсутствие в игре дополнительной колоды;
  - ✓ int[] gdCards, gdfCards – массивы, указывающие соответственно общее количество карт и количество карт, лежащих «рубашкой» вверх при раздаче карт в игровые стопки – будет считаться, что при наполнении игровой стопки в процессе раздачи расклада пасьянса сначала укладываются карты, перевернутые «рубашкой» вверх, а все остальные карт gdCards[i]-gdfCards[i] – «картинкой» вверх;
  - ✓ TDRandomInitialCard – определяет, будут ли целевые стопки начинаться с карты со случайно выбранным достоинством (по умолчанию – с туза);
  - ✓ TDInitialCardCount – определяет количество целевых стопок игры, в которых первая (нижняя) карта должна быть роздана сразу в начале игры;
  - ✓ selectedGameID – идентификатор выбранной разновидности пасьянса, по которому будет происходить поиск конфигурации в структурном массиве установок вариантов игры при создании новой игры;
  - ✓ Title, Rules – метаданные выбранной разновидности пасьянса, определяемые идентификатором selectedGameID – содержат соответственно наименование и правила выбранной модификации игры.
- ***Обеспечить взаимодействие с игроком через пользовательский интерфейс главного окна***.

Взаимодействие игровой модели с пользователем должно осуществляться через главное окно пользовательского интерфейса, являющееся одновременно игровым полем. Для этого классу Game необходимо располагать ссылкой на экземпляр окна игрового поля – MainWindow MW. Все событийные действия игрока (имеются ввиду манипуляции мышью – захват, перемещение, отпускание) отслеживаются механизмами обработки событий соответствующих UI- компонентов. Однако в конечном итоге совершение игроком указанных событий должно передавать управление в класс Game. Для этого предусмотрено поле capturedToMove, которое сообщает игровой модели, что произведен захват некоторой стопки карт для их возможного перемещения, и соответствующие методы:

- ✓ CaptureDeck – выполняет операции подготовки к перемещению;
- ✓ CompleteMoving – выполняет операции, завершающие перемещение захваченных карт.

- ***Обеспечить реализацию визуальных эффектов***.

  Комплекс визуальных эффектов игрового процесса состоит в обеспечении эффекта неуправляемого пользователем перемещения карты в пределах игрового поля из одной колоды в другую. Технология WPF предоставляет широкие возможности простого и мало затратного способа реализации анимационных эффектов посредством управления свойствами зависимостей. Класс Game будет использовать свойство ThicknessAnimation TA для анимации одной карты. Возможности ThicknessAnimation позволяют создавать анимацию перемещения, управляя свойством зависимости MarginProperty любого UI-объекта. Для выполнения анимации необходимо знать саму карту, которая будет перемещаться – MC, а также стопки источника – dFrom и назначения – DTo. Основной метод, реализующий анимацию – MoveCard, в этом методе будут происходить все подготовки и вычисления; по завершении анимации будет выполнен метод – CardAnimationComplete, задача которого поддерживать последовательную цепочку анимаций выбранного количества карт на базе структуры SendCardsInfo.

Листинг 3.19 – Прототип класса управления игровым процессом.

```
public static class Game
{
    public static MainWindow MW;
    public static int selectedGameID = -1; // выбранная игра - идентификатор из меню
```

```
        public static DistributionDeck __DD; //         С Т О П К И
        public static BustDeck __BD;
        public static AdditionalDeck __AD;
        public static List<GameDeck> __GD;
        public static List<TaskDeck> __TD;
        public static BufferedDeck __MD;
        public static bool capturedToMove = false; // флаг начала перемещения
        // Захват стопки мышью для перемещения
        public static void CaptureDeck(Deck src, Card topCard)
        public static void CompleteMoving() // Завершение перемещения
        // Анимация
        private static ThicknessAnimation TA; // Анимация перемещения карты
        public static void MoveCard(Deck dFrom, Deck dTo, bool turn)
        private static Deck DFrom, DTo; // стопка источник и цель
        private static Card MC; // перемещаемая карта
        // Анимция перемещения карты: этап - завершение и обновление
        private static void CardAnimationComplete(object sender, EventArgs e)
        // Настройки пасьянса общие
        public static int TaskDeckCount; // кол-во целевых стопок
        public static int GameDeckCount; // кол-во игровых стопок
        public static bool IsBustDeckExist = true; // наличие колоды перебора
        public static bool IsAdditionalDeckExist = false; // наличие дополн. колоды
        public static string Title = ""; // название пасьянса
        public static string Rules = ""; // правила пасьянса
        // распределение карт по игровым стопкам
        private static int[] gdCards, gdfCards;
        // начальная карта в целевой колоде - случайная
        public static bool TDRandomInitialCard;
        // кол-во целевых колод, у которых первая карта уже должна лежать
        public static int TDInitialCardCount;
        // раздача пасьянса - конструирование игры
        public static void Prepare(MainWindow MWnd)  // подготовка
        public static bool CheckForWin()  // проверка на победу
        public static void newPlayOut()   // инициализация игры
    }
```

3.2.3. Представление объектов в интерфейсе игры

Основные объекты игры представлены в интерфейсе как пользовательские компоненты UserControl. Компонент для карты – UICard, для стопки – UIDeck. Любой тип стопки реализуется на базе единого компонента UIDeck.

UserControl, помимо визуальной части, также реализуется классом, предоставляющим свой исходный код для редактирования.

Класс карты UICard предназначен главным образом для отображения карты, а также для отслеживания некоторых событий, посредством которых возможна реализация логики игры. Основной метод этого класса, отвечающий за отображение карты – drawCard. Данный метод использует поля связанного класса Card – масть и достоинство – для прорисовки внешнего вида карты. Изображения лицевой части и «рубашки» карт «подхватываются» из PNG-ресурсов, подключенных к проекту. В UICard также должна происходить обработка пользовательских событий мыши.

Прототип класса UICard приведен в листинге 3.20.

Листинг 3.20 – Прототип класса визуальной компоненты карты.

```
public partial class UICard : UserControl
{
    Card MyClass; // Ссылка на класс карты
    public UICard(bool face, Card MyParent) // конструктор
    public void drawCard(bool face) // переворачивание карты
}
```

Класс визуализации стопки отвечает главным образом за процесс компоновки карт в рамках стопки, пересчитывая расстояния перекрытия карт в стопке для их оптимального отображения. При этом учитываются размеры карт, их количество в топке, а также ориентация стопки. Простейший случай – ориентация в одну стопку, когда все карты последовательно прорисовываются одна поверх другой. При вертикальной или горизонтальной ориентации должны мыть вычислены размеры перекрытий соседних карт, чтобы все карты стопки были видны игроку. Все эти действия реализуются методом UpdateView. В UIDeck также должна происходить обработка пользовательских событий мыши.

Прототип класса UIDeck приведен в листинге 3.21.

Листинг 3.21 – Прототип класса визуальной компоненты стопки.

```
public partial class UIDeck : UserControl
{
    private Deck myDeck;
    public UIDeck(Deck parentDeck, Advantage firstCardRule)
    // Перерисовка стопки при изменении количества карт
    public void UpdateView()
}
```

3.2.4. Представление модификаций игры

Модификации (разновидности) «Косынки» отличаются между собой предустановленной конфигурацией правил, описывающих наличие и количество объектов, используемых в игре, а также свойства, характеризующие их поведение. Поэтому конфигурация варианта пасьянса должна включать данные по количеству стопок разного назначения, количеству карт в этих стопках и способу их распределения, а также структуру правил DeckRules каждой стопки.

Таким образом, совокупность данных, которые будут описывать конкретный вариант игры, будут представлены (записаны) в виде файла параметризации модели пасьянса.

Файл параметризации модели «Косынки» будет иметь иерархическую структуру. Для описания структур подобного вида широко используются языки разметки, например, XML. Таким образом, каждый файл параметризации пасьянса должен быть представлена в виде XML-файла.

В листинге 3.22 приведен пример описания конфигурации игры для ее разновидности «Красный и черный».

Структура XML файла заключается в описании общих параметров игры и инициализации правил всех стопок, которые должны присутствовать в данной разновидности игры.

Листинг 3.22 – Структура файла параметризации пасьянса в XML-формате.

```
<sol20>
	<Title>Красный и черный</Title>
	<Rules>   </Rules>
	<TaskDecks>
		<Count>8</Count>
		<RandomInitialCard>0</RandomInitialCard>
		<InitialCardsCount>0</InitialCardsCount>
		<DeckRules>
			<TakeFromRule>1</TakeFromRule>
			<TopPlaceRuleOrder>1</TopPlaceRuleOrder>
			<TopPlaceRuleSequence>1</TopPlaceRuleSequence>
			<TopPlaceRuleMultiplicity>1</TopPlaceRuleMultiplicity>
			<Advantage>14</Advantage>
		</DeckRules>
	</TaskDecks>
	<GameDecks>
		<Count>8</Count>
		<Cards>1,1,1,1,1,1,1,1</Cards>
		<FlipCards>0,0,0,0,0,0,0,0</FlipCards>
		<DeckRules>
			<TakeFromRule>6</TakeFromRule>
			<TopPlaceRuleOrder>2</TopPlaceRuleOrder>
			<TopPlaceRuleSequence>4</TopPlaceRuleSequence>
			<TopPlaceRuleMultiplicity>2</TopPlaceRuleMultiplicity>
			<Advantage>0</Advantage>
		</DeckRules>
	</GameDecks>
	<BustDeck>
		<AdditionalCount>1</AdditionalCount>
		<DeckRules>
			<TakeFromRule>1</TakeFromRule>
			<TopPlaceRuleOrder>0</TopPlaceRuleOrder>
			<TopPlaceRuleSequence>0</TopPlaceRuleSequence>
			<TopPlaceRuleMultiplicity>0</TopPlaceRuleMultiplicity>
			<Advantage>0</Advantage>
		</DeckRules>
	</BustDeck>
	<DistributionDeck>
		<Cards>104</Cards>
		<DDeckClickBehavior>0</DDeckClickBehavior>
		<BustCount>0</BustCount>
		<DeckRules>
```

```
                    <TakeFromRule>0</TakeFromRule>
                    <TopPlaceRuleOrder>0</TopPlaceRuleOrder>
                    <TopPlaceRuleSequence>0</TopPlaceRuleSequence>
                    <TopPlaceRuleMultiplicity>0</TopPlaceRuleMultiplicity>
                    <Advantage>0</Advantage>
                </DeckRules>
        </DistributionDeck>
        <AdditionalDeck> <Exists>0</Exists> </AdditionalDeck>
</sol20>
```

Основные разделы файла параметризации:

- Заголовок, или название разновидности – Title.
- Правила игры – Rules, приведенные в текстовом виде, чтобы игрок мог ознакомиться с ними.
- Описание целевых стопок – TaskDecks, включающее:
  - ✓ Количество стопок – Count.
  - ✓ Случайный выбор нижней карты стопки – RandomInitialCard.
  - ✓ Количество стопок, в которых первая карта открыта – InitialCardsCount.
  - ✓ Правила поведения – DeckRules.
- Описание игровых стопок – GameDecks, включающее:
  - ✓ Количество стопок – Count.
  - ✓ Количество карт в каждой из игровых стопок – Cards.
  - ✓ Количество карт, лежащих снизу «рубашкой» вверх, в каждой из игровых стопок – FlipCards.
  - ✓ Правила поведения – DeckRules.
- Описание стопки перебора – BustDeck, включающее:
  - ✓ Количество карт, которое добавляется из стопки раздачи в другие стопки (перебора или игровые) по клику игрока – AdditionalCount.
  - ✓ Правила поведения – DeckRules.
- Описание стопки раздачи – DistributionDeck, включающее:
  - ✓ Общее количество карт в игре – Cards.
  - ✓ Поведение при клике игрока – DdeckClickBehavior.
  - ✓ Количество разрешенных переборов стопки перебора – BustCount.
  - ✓ Правила поведения – DeckRules.
- Описание дополнительной стопки – AdditionalDeck, включающее:
  - ✓ Флаг, определяющий наличие дополнительной стопки – Exists.
  - ✓ Правила поведения – DeckRules, которые описываются только в случае, если Exists = 1.

Модель игры должна каким-то образом получать данные всех имеющихся файлов параметризации, хранить их, а затем при выборе разновидности пасьянса игроком соответствующим образом настроить свое поведение в соответствии с установленными правилами. Последнее реализуется методом newPlayOut класса управления моделью Game.

Для операций поиска и распознавания моделей используется специальный класс XMLStructure. В этом классе определена структура klondike, в которую должно считываться информация из XML-файла. В листингах 3.23 и 3.24 приведено описание структуры хранения файла параметризации пасьянса и класса поиска и распознавания этих файлов.

Листинг 3.23 – Структура хранения.

```
public struct klondike:IDisposable
{
    public bool isNull;  // =false - успешно создана
    public string title; // название
    public string rules; // описание правил
    public bool additionalExists; // существует доп. стопка
    // кол-во целевых и игровых стопок, карт в игре
    public int taskDecks, gameDecks, cards;
    // первая карта в целевых стопках - случайный выбор
    public bool randomInitialCard;
    public int initialCardCount; // кол-во целевых стопок с лежащей первой картой
    // поведение стопки раздачи при клике мыши
    public DDeckClickBehavior ddClickBehaviour;
    // кол-во переборов, значение инкремента стопки перебора
    public int bustCount, bustAdditionalCount;
    // кол-во карт по игровым стопкам (кол-во перевернутых руь. вверх)
    public int[] gameDecksCount, gameDecksFlipCount;
    // конфигурации правил стопок
    public DeckRules taskDeckCfg,gameDeckCfg,bustDeckCfg,addDeckCfg, distrDeckCfg;
}
```

Листинг 3.24 – Класс поиска и распознавания.

```
public static class XMLStructure
{
    public static List<klondike> Games = new List<klondike>();// Список вариантов игры
    //   Составление списка доступных игр (по файлам XML)
    public static void ReadGameList()
    //   Получение инициализирующей структурыиз XML файла
    static klondike ParseXMLToGameStructrure(string fileName)
}
```

Класс XMLStructure реализует метод поиска файлов – ReadGameList. Найденные этим методом файлы должны передаваться в метод обработки XML файла и распознавания конфигурации игры – ParseXMLToGameStructrure. Если конфигурация корректна и распознана, то она заносится в список доступных вариантов пасьянса Games.

Список найденных файлов пасьянса реализован в виде коллекционной структуры List<klondike> Games.

3.2.5. Диаграмма пакетов

На рисунке 3.3 приведена модель пакетов, основанная на вышеописанных концептуальных решениях. В состав пакетов модели входят:

- Пользовательский интерфейс (***USER INTERFACE***), объединяющий в себе классы окон (игровое поле, вспомогательные окна) ***Windows*** и пользовательские компоненты (карты, стопки) – ***uiGameComponents***.
- Основной пакет, содержащий логику модели и ее управляющие структуры – ***GameModel***.
- Пакет, выполняющий поиск и инициализацию возможных конфигураций модели под различные варианты пасьянса – ***XML Parser***.
- Непосредственно набор файлов, описывающих различные конфигурации игры на языке XML – ***Games***.
- Пакет информационного обеспечения модели для записи и хранения пользовательских рекордов игры (***Records***), включающий в себя файл базы данных, библиотеки провайдера данных и ядро СУБД.
- Дополнительные вспомогательные библиотеки из пакета ***.NET Framework 4.0***, необходимые для реализации всех возможностей построенной модели.

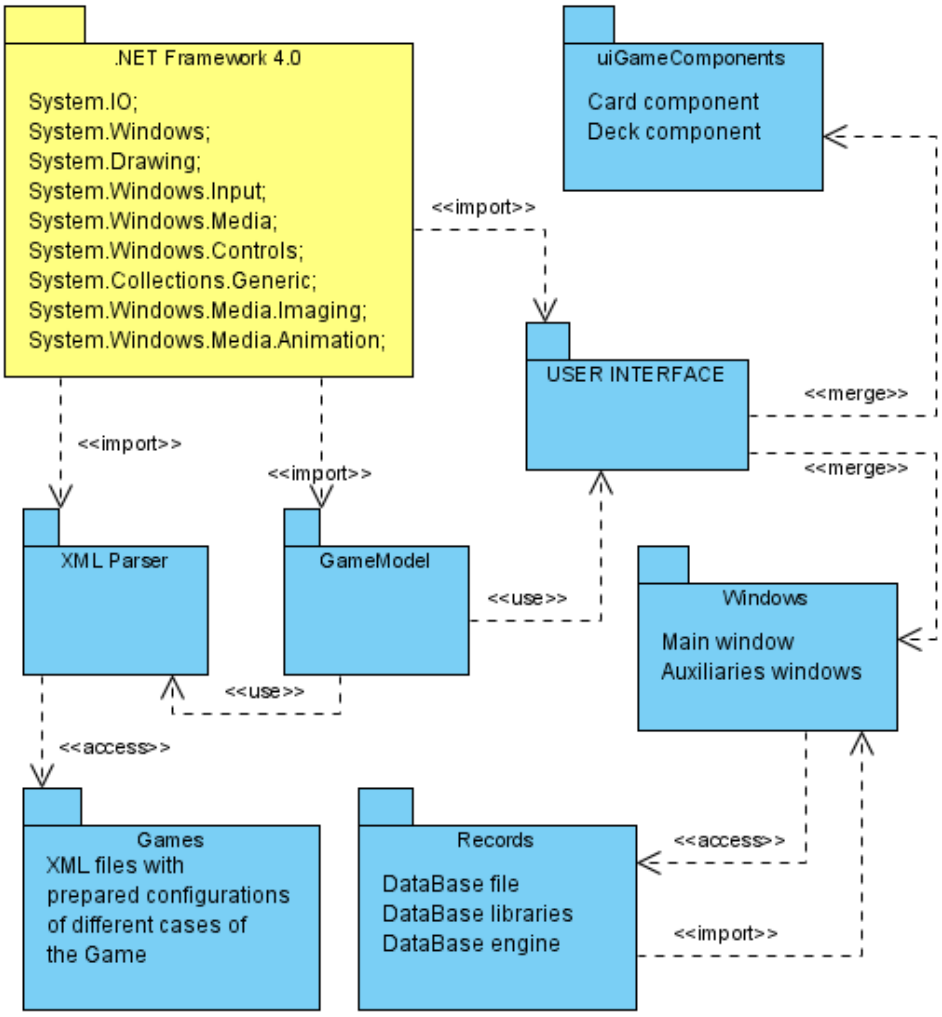

Рисунок 3.3 - Диаграмма пакетов

3.2.6. Диаграмма классов

Классы – это структурные блоки, из которых выстраивается любая объектно-ориентированная система. Классы являются описанием совокупности объектов с общими атрибутами, свойствами, методами и схемой отношений.

Диаграмма классов – это набор статических декларативных объектов, связанных между собой в определенную схему. Применяя при моделировании диаграммы классов с специальные (CASE) средства, классовую модель можно преобразовать в файлы исходного кода программы на соответствующем языке программирования. В данной работе первичные файлы программного кода получены таким путем – сгенерированы средой проектирования Visual Paradigm.

Таким образом, диаграмма классов является конечным результатом проектирования модели и началом ее реализации (процесса непосредственной разработки).

На рисунке 3.4 приведена диаграмма классов основных объектов модели, выявленных в п. 3.2.1 – 3.2.3.

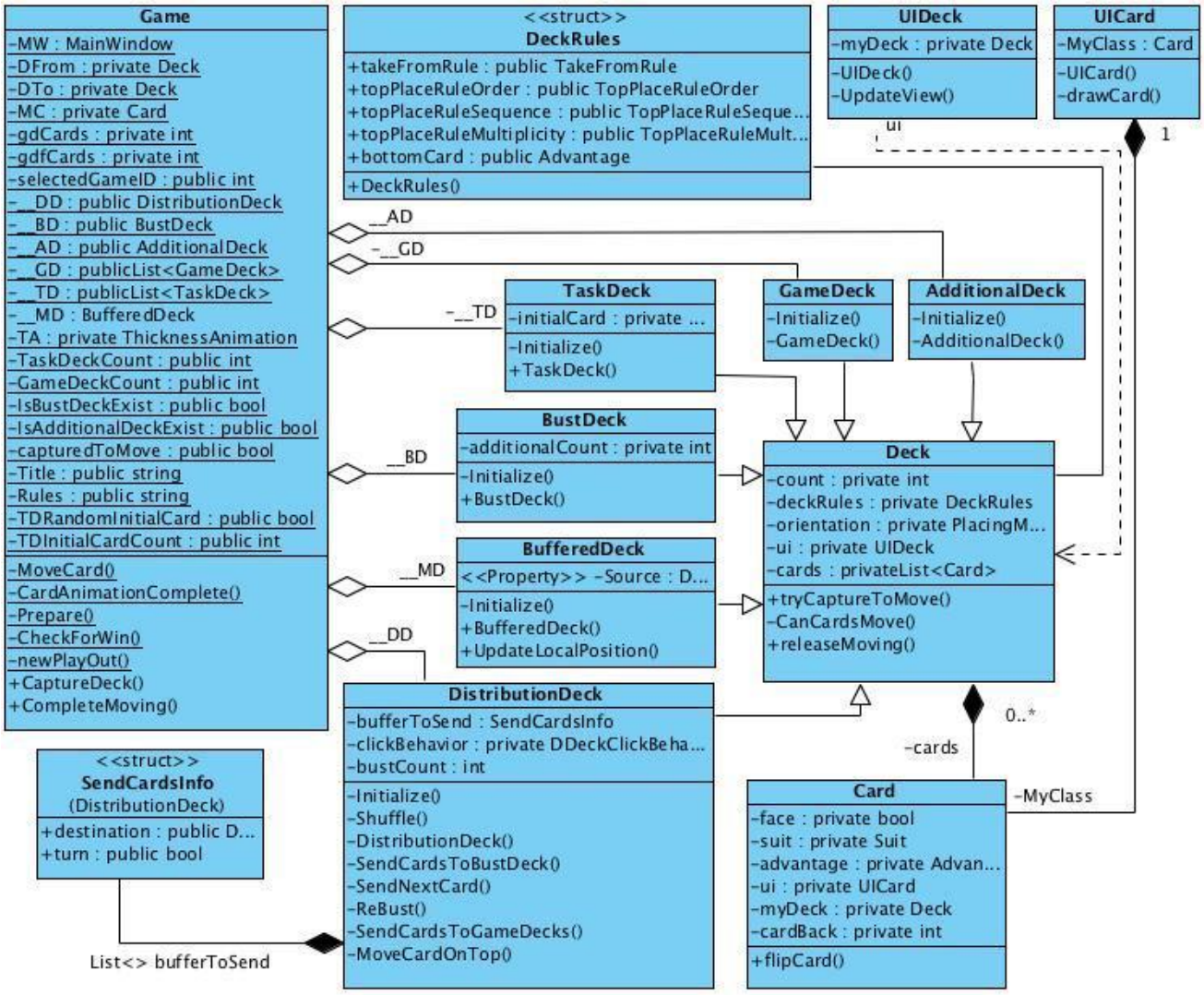

Рисунок 3.4 – Диаграмма классов модели.

На рисунке 3.5 приведена диаграмма классов описания параметров основных объектов модели, выявленных в п. 3.2.1 – 3.2.3, в виде структур данных и значений соответствующих нумераторов.

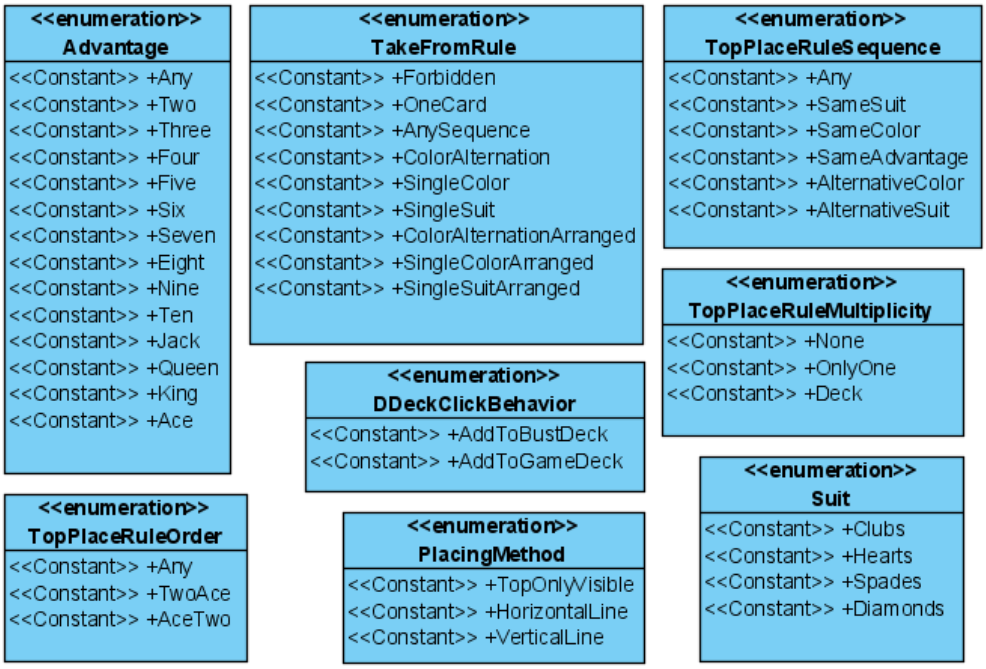

Рисунок 3.5 – Диаграмма классов – структуры и нумераторы.

На рисунке 3.6 приведена диаграмма классов модуля поиска и распознавания файлов параметризации пасьянса, выполненная в соответствии с описанием, приведенном в п. 3.2.4.

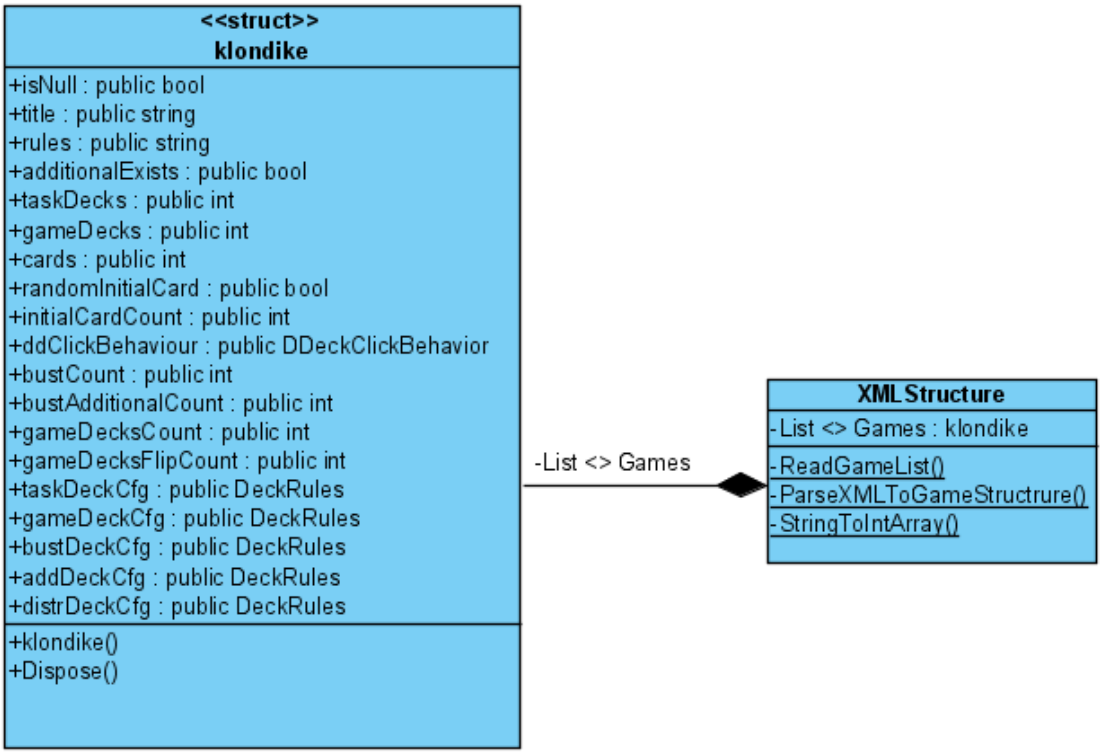

Рисунок 3.6 – Диаграмма классов модуля распознавания файлов параметризации модели.

3.3. Проектирование модели поведения

3.3.1. Общий алгоритм подготовки игры и игрового процесса

На рисунке 3.7 представлена модель алгоритма подготовки, выбора и начала новой игры. Алгоритм начинается с процесса поиска и распознавания файлов параметризации пасьянса – все найденные варианты игры собираются в коллекцию игр. Собранная коллекция пасьянсов будет доступна в окне выбора игры.

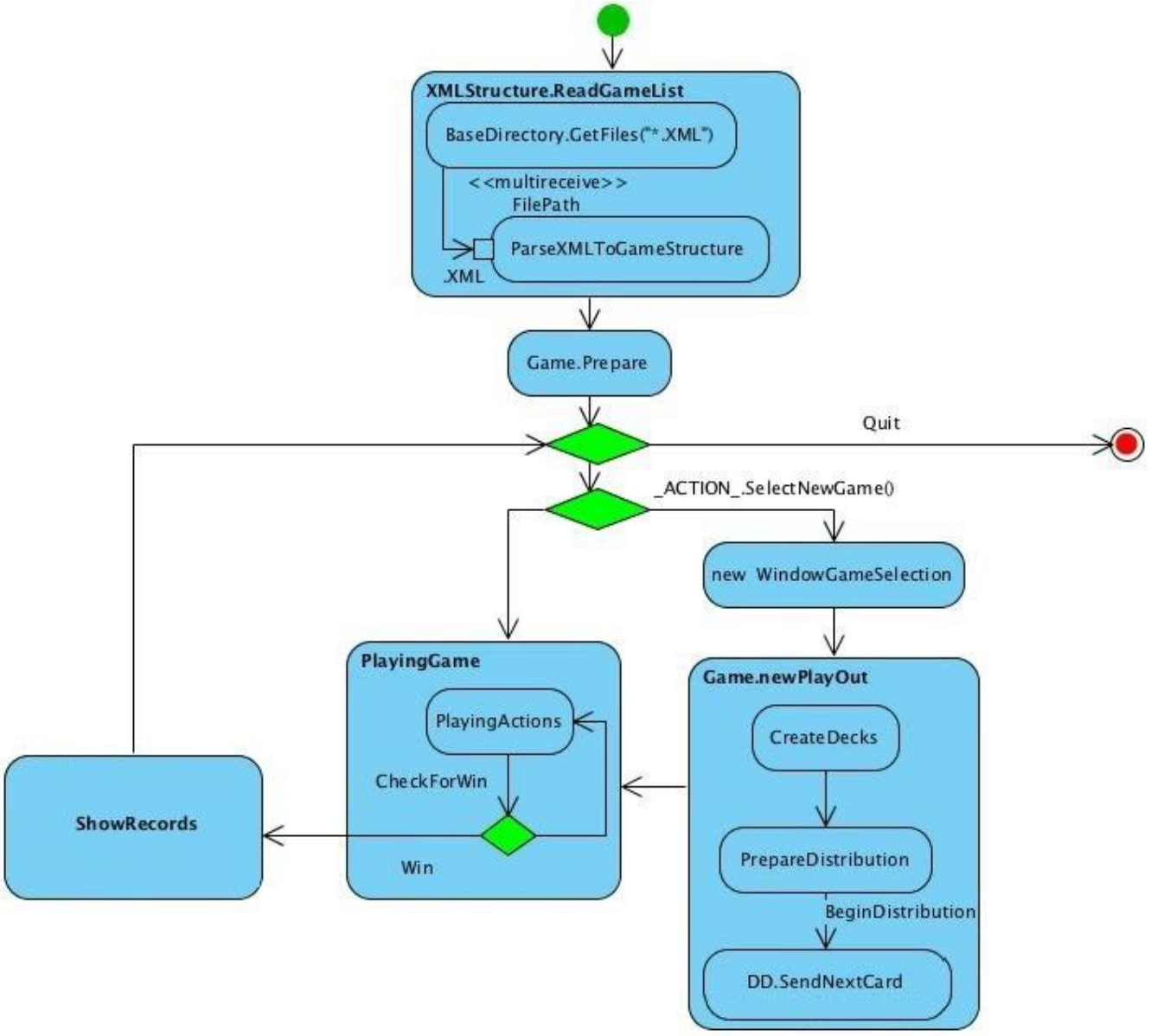

Рисунок 3.7 – Диаграмма активности подготовки игры.

После выбора игрока класс Game создает расклад указанного пасьянса: создает основную колоду раздачи, создает коллекцию карт в ней, затем создает все остальные колоды и устанавливает их правила. В завершении, собирается коллекция карт для пересылки их из стопки раздачи в остальные (игровые и

целевые) – далее анимируется перемещение карт из этой коллекции в соответствующие стопки.

После создания расклада, начинается процесс игры – игрок совершает игровые манипуляции, после каждой из которых класс Game проверяет ситуацию на победный расклад (когда все карты помещены в целевые стопки). По завершении игры подсчитываются очки.

3.3.2. Диаграмма состояний класса Game

На рисунке 3.8 показана диаграмма состояний класса Game, иллюстрирующая описание, приведенное в п. 3.3.1. На диаграмме указаны также состояния классов основных стопок игры, приведенные в рамках состояния формирования расклада пасьянса.

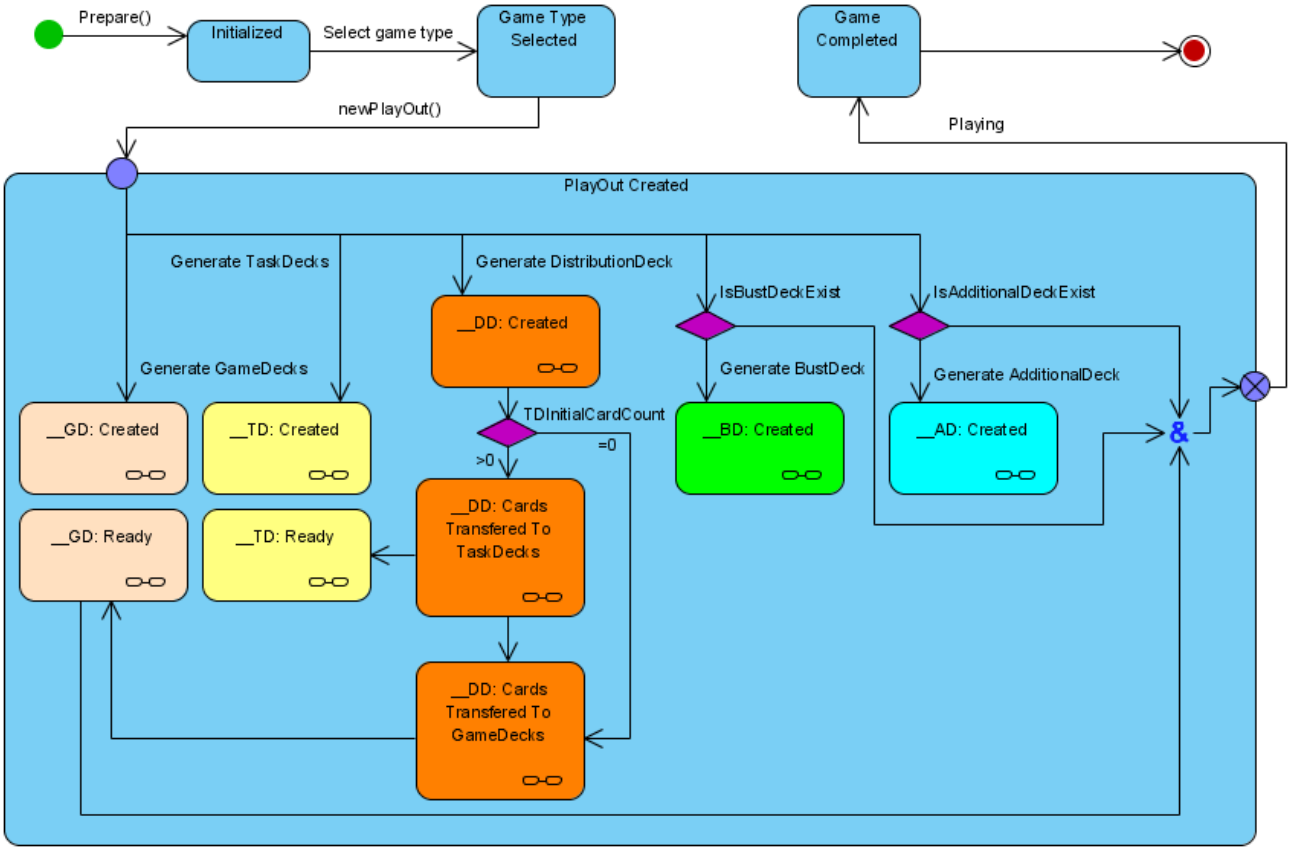

Рисунок 3.8 – Диаграмма состояний при подготовке игры.

4. ПРОЕКТИРОВАНИЕ ПРИЛОЖЕНИЯ

4.1. Структура приложения

Структура приложения пасьянса представлена на рисунке 4.1 и выполнена на основе разработанной модели. Структурная схема показывает место разработанной объектной модели пасьянса в составе проектируемого приложения.

В составе структуры присутствуют 3 основных компонента:

- модель игры, состоящая из двух частей:
  - ✓ статической: представлена совокупностью классов карт и колод различного назначения, реализует схему отношений между ними, инициализирует правила их поведения; в статической части также объявлены и описаны все основные структуры данных и параметров, а также нумераторы, значения которых используются в качестве этих параметров;
  - ✓ динамической: является классом верхнего уровня, предоставляющим механизм реализации базовой логики игры и поддержки основной и дополнительной функциональности (создания и инициализации игры, поддержки эффектов визуализации и анимации объектов, проверки на победу, подсчета количества очков, заработанных игроком, и т.д.), а также взаимодействия с игроком (через интерфейсную часть);
- пользовательский интерфейс, представленный игровым полем и другими вспомогательными (обслуживающими) окнами, в составе которых:
  - ✓ окно настроек внешнего вида игрового поля;
  - ✓ окно просмотра правил выбранной игры;
  - ✓ окно просмотра рекордов игроков;
- набор разновидностей игры, представленный в виде специально выделенной директории XML-файлов, содержащих конфигурации вариантов пасьянса, а также механизм, поддерживающий обнаружение и распознавание этих разновидностей, сбор их в отдельный список (доступный игроку для выбора) и последующего интегрирования их в игровую модель для инициализации разновидности игры, выбранной пользователем.

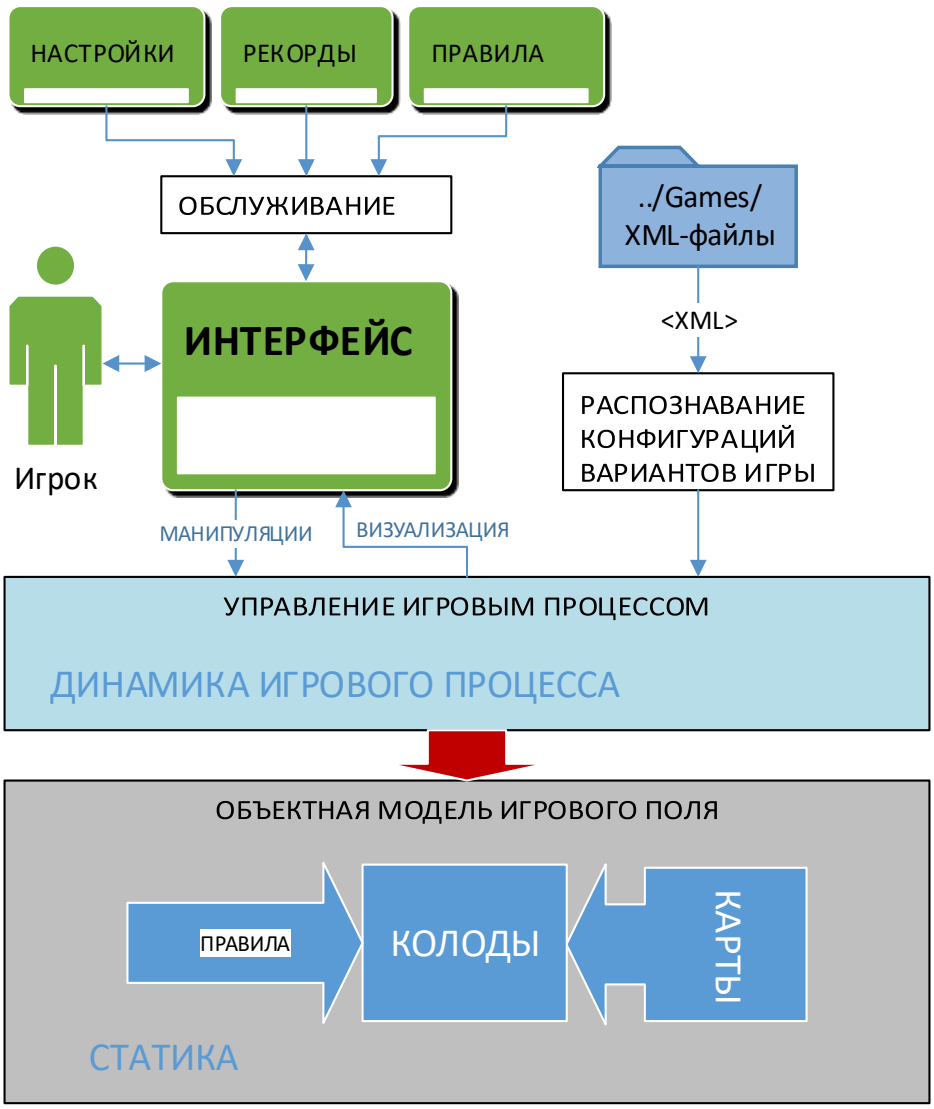

Рисунок 4.1 – Структура приложения на базе модели.

4.2. Проектирование компонентного состава приложения

4.2.1. Диаграмма компонентов приложения

Конечное приложение состоит из следующих основных компонентов (в соответствии с рисунком 4.2:

- Исполняемый файл программы – Solitair20.exe.
- Предустановленный пакет библиотек в виде фреймворка .NET версии 4.0 и выше.
- Пакет различных вариантов игры, выполненный в XML-файлах в директории Games.
- Компоненты информационного обеспечения:
  - SQLite Engine – пакет «движка» СУБД:

- ✓ SQLite.Interop.dll, System.Data.SQLite.Linq.dll – библиотеки БД SQLite;
- ✓ System.Data.SQLite.dll – библиотека провайдера данных SQLite;

▪ records.db – файл базы данных, содержащий рекорды игроков.

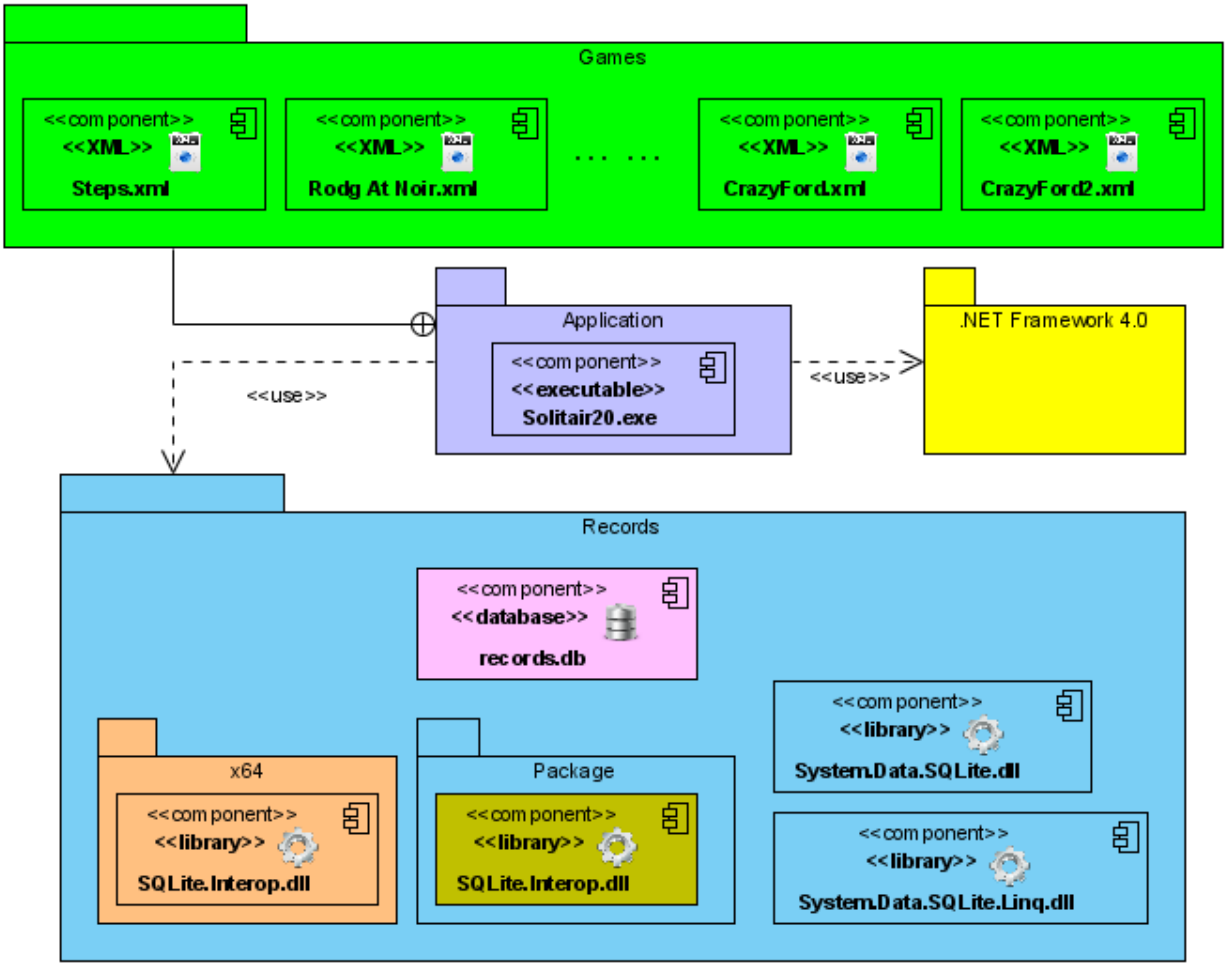

Рисунок 4.2 – Диаграмма компонентов приложения.

4.2.2. Диаграмма компонентов проекта

Детализированная диаграмма компонентов, составляющих проект приложения, приведена, на рисунке 4.3.

Основные пакеты проекта:

▪ ***USER INTERFACE*** – включают в себя компоненты пользовательского интерфейса приложения, среди которых:

- ✓ ***Windows*** – окна приложения:
  - ➢ *WindowSettings* – окно настройки отображения игры.
  - ➢ *WindowRecords* – окно просмотра рекордов игроков.

- *MainWindow* – основное игровой поле.
- *WindowRules* – окно просмотра правил текущей игры.
- *WindowColorDialog* – вспомогательное окно, предлагающее задание цвета фона игрового поля.
- *WindowGameSelection* – окно выбора разновидности пасьянса для начала раздачи.

✓ ***uiGameComponents*** – пользовательские компоненты типа UserControl:

- UICard – компонент визуализации карты.
- UIDeck – компонент визуализации стопки.

▪ ***GameModel*** – основной компонент, реализующий модель игрового процесса, включающий в себя:

✓ Структуры данных модели.

✓ Нумераторы, описывающие параметры структур данных модели.

✓ Класс, реализующий свойства и поведение карты в игровом процессе.

✓ Базовый и производный классы стопок, задействованных в игровом процессе.

✓ Класс управления игровым процессом – подготовка, управление событиями, анимация, логические проверки завершения игры, предустановки отображения.

▪ ***XML Parser*** – компонент, реализующий поиск и преобразование XML файлов в конфигурацию вариантов игры, включающий:

✓ ***klondike*** – описание структуры, посредством которой происходит считывание и распознавание вариантов игры и последующая их подготовка в игровом поле.

✓ ***XMLStructure*** – класс реализации методов поиска, чтения и обработки XML файлов в массив структур *klondike*, который является источником доступных вариантов игры.

▪ ***ImageProcessor*** – статический класс, содержащий методы обработки графической информации, позволяющий преобразовывать графическую информацию из ресурсов проекта в источники, пригодные для применения их в компонентах пользовательского интерфейса.

▪ ***Auxiliaries*** – класс, предоставляющий вспомогательные методы.

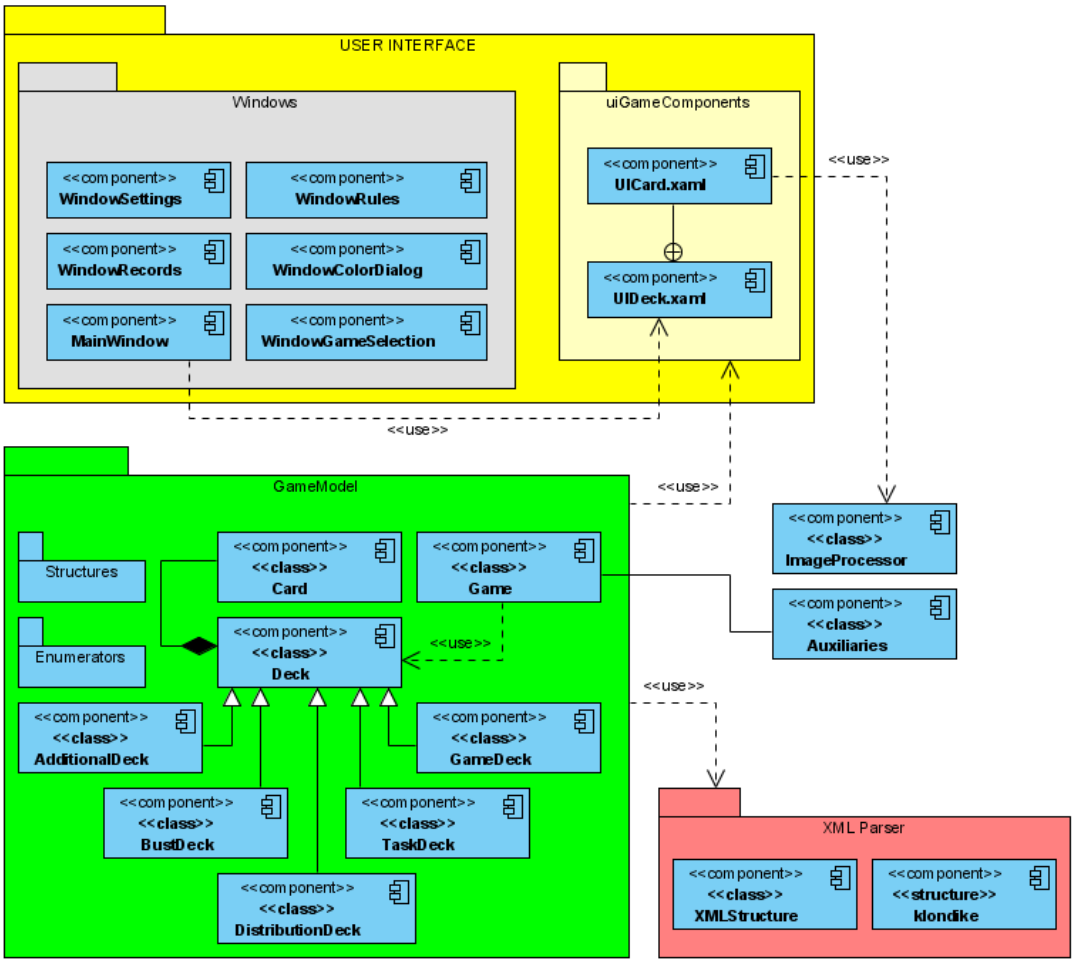

Рисунок 4.3 – Диаграмма компонентов проекта приложения.

5. РЕАЛИЗАЦИЯ ПРИЛОЖЕНИЯ

В данной главе описаны основные концепции реализации приложения Solitair20. Здесь описаны принципы построения пользовательского интерфейса приложения, а также приведены алгоритмы выполнения основной игровой логики.

5.1. Проект интерфейса основного окна

Основное окно приложения спроектировано с помощью XAML-разметки. Данное окно должно предложить пользователю игровую область и некую панель команд управления приложением. Построение главного окна основано на «контейнерном» принципе, при котором используются некоторые объекты-контейнеры, в которые «вкладываются» интерфейсные компоненты. В рамках технологии WPF таким контейнером может быть компонент типа Grid. Его преимущество заключается в том, что данный компонент можно гибко конфигурировать под содержимое в плане размеров, то есть – задавать конкретные размеры (минимальные) с возможностью их автоматического расширения под содержимое – это идеально подходит для реализации отображения стопок, т.к. их размер напрямую зависит от количества карт, находящихся в них (которое постоянно варьируется).

Кроме того, Grid можно разбивать на ячейки, причем свойства размеров и автоподстройки под содержимое для каждой из ячеек можно задавать независимо друг от друга аналогично, как и для всего Grid. На рисунке 5.1 приведен макет главного окна. Цифрами условно показаны основные контейнеры объектов (строка заголовка Grid Name="Header" не учтена):

0. Панель команд Grid Name="grdManiMenu" (меню игры) – назначение команд приведено в таблице 5.1.
1. Контейнер хранения целевых стопок Grid Name="grdTaskDecks".
2. Контейнер хранения стопки перебора Grid Name="grdBustDeck".
3. Контейнер хранения стопки раздачи Grid Name="grdGeneralDeck".
4. Контейнер хранения игровых стопок Grid Name="grdGameDecks".
5. Контейнер хранения дополнительной стопки Grid Name="grdAdditionalDeck".
6. Общий контейнер, который хранит в себе контейнеры 0-5, в нем также хранится и перемещается буферная стопка Grid Name="grdGlobal".

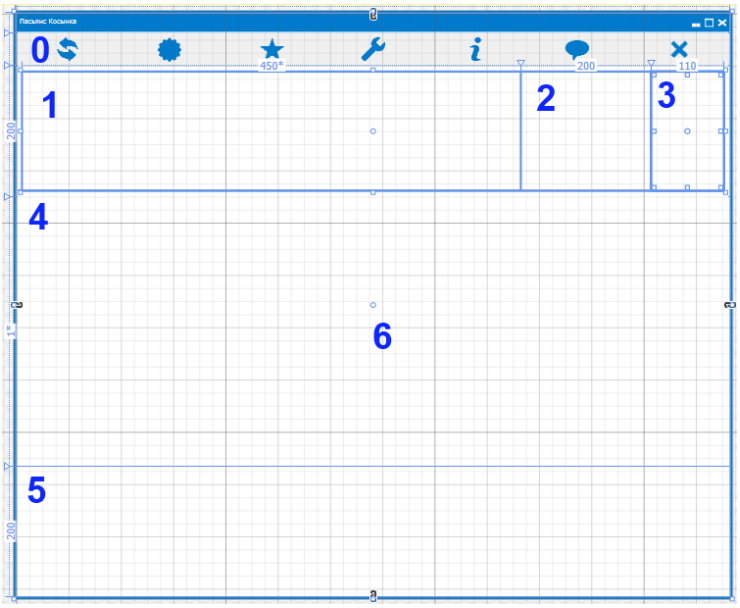

Рисунок 5.1 – Проект главного окна приложения.

Контейнеры 1-3 объединены в один контейнер более высокого уровня (по иерархии) – Grid Name="grdTopDecks", который помещен непосредственно в контейнер 0.

Контейнеры хранения целевых и игровых стопок впоследствии (на этапе подготовки игрового поля под выбранную разновидность пасьянса) будут разделены на горизонтальные ячейки, каждая из которых будет хранить отдельный экземпляр визуальной компоненты соответствующей стопки.

В листинге 5.1 приведен код XAML-разметки окна в соответствии с описанной схемой разбивки на контейнеры.

Листинг 5.1 – XAML-разметка главного окна приложения.

```
    <Grid Name="grdGlobal">
        <Grid Name="grdField">
            <Grid.RowDefinitions>
                <RowDefinition Height="30"/>
                <RowDefinition Height="50"/>
                <RowDefinition Height="200"/>
                <RowDefinition Height="*"/>
                <RowDefinition Height="200"/>
            </Grid.RowDefinitions>
            <!--  ========================  ЗАГОЛОВОК ОКНА  ======================  -->
            <Grid Name="Header" Grid.Row="0" Background="#FF007ACC"> </Grid>
            <!--  ========================  ЗАГОЛОВОК ОКНА  ======================  -->

            <!--  ==========================  ГЛАВНОЕ МЕНЮ  ======================  -->
            <Grid Grid.Row="1" Name="grdManiMenu" Background="#FFF0F0F0"> </Grid>
            <!--  ==========================  ГЛАВНОЕ МЕНЮ  ======================  -->

            <!--  ========================  ВЕРХНИЕ СТОПКИ  ======================  -->
            <Grid Name="grdTopDecks" Grid.Row="2" Margin="10,10,10,10">
                <Grid.ColumnDefinitions>
                    <ColumnDefinition Width="450*"/>
```

```
                    <ColumnDefinition Width="200"/>
                    <ColumnDefinition Width="110"/>
                </Grid.ColumnDefinitions>
                <!-- _________________ ЦЕЛЕВЫЕ СТОПКИ _________________ -->
                <Grid Grid.Column="0" Name="grdTaskDecks"/>
                <!-- _________________ СТОПКА ПЕРЕБОРА _________________ -->
                <Grid Grid.Column="1" Name="grdBustDeck"/>
                <!-- _________________ СТОПКА РАЗДАЧИ _________________ -->
                <Grid Grid.Column="2" Name="grdGeneralDeck"/>
            </Grid>
            <!--  ========================  ВЕРХНИЕ СТОПКИ  ====================  -->

            <!--  ========================  ИГРОВЫЕ СТОПКИ  ====================  -->
            <Grid Grid.Row="3" Name="grdGameDecks" VerticalAlignment="Top"/>
            <!--  ========================  ИГРОВЫЕ СТОПКИ  ====================  -->

            <!--  ==================  ДОПОЛНИТЕЛЬНАЯ СТОПКА  ====================  -->
            <Grid Grid.Row="4" Name="grdAdditionalDeck" HorizontalAlignment="Right"
Margin="0,5,20,5" />
            <!--  ==================  ДОПОЛНИТЕЛЬНАЯ СТОПКА  ====================  -->

        </Grid>
    </Grid>
```

Таблица 5.1 – Назначение игрового меню.

Команда	Назначение
	Раздать новый расклад текущего варианта пасьянса
	Выбрать другой вариант пасьянса и создать его расклад
	Просмотреть таблицу рекордов игроков
	Получить доступ к окну настроек отображения игры
	Получить справочную информацию о приложении
	Просмотреть правила текущего варианта пасьянса
	Завершить работу приложения

5.2. Проект интерфейса вспомогательных окон

5.2.1. Окно настроек игры

Макет окна настройки отображения игрового поля и объектов приведен на рисунке 5.2. В данном окне можно настроить «рубашку» карт, цвет градиентного перехода игрового поля и скорость анимации карт при раздаче. Для задания цвета фона игрового поля добавлено окно, макет которого приведен на рисунке 5.3.

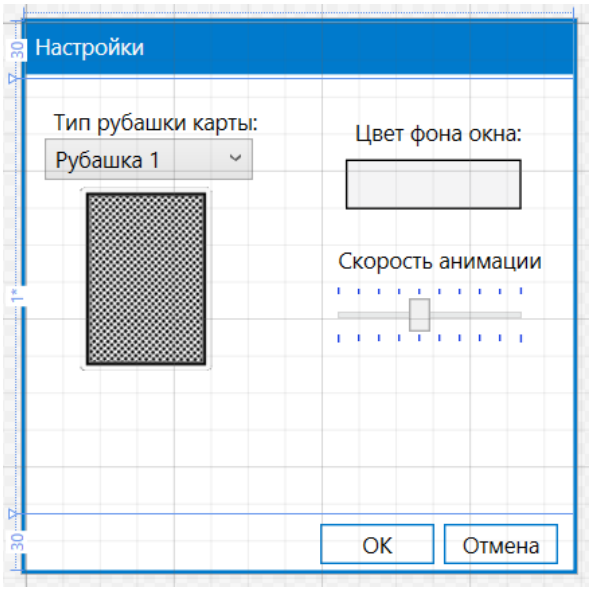

Рисунок 5.2 – Макет окна настройки внешнего вида игры.

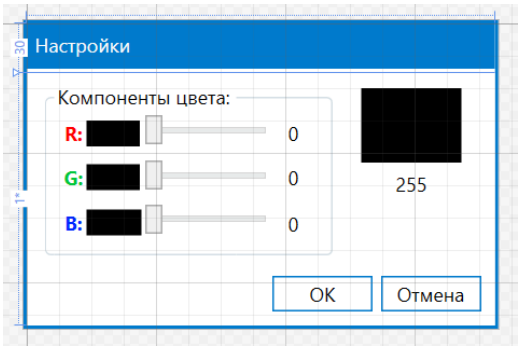

Рисунок 5.3 – Макет окна определения цвета.

5.2.2. Окно рекордов

Макет окна рекордов приведен на рисунке 5.4. Окно приводит результат текущей игры и предлагает пользователю ввести имя и зафиксировать свой результат в таблице рекордов. Результат будет занесен в таблицу рекордов и базу данных в случае, если в данной разновидности он лучший среди уже имеющихся.

Результат игры рассчитывается как количество секунд, потраченных на игру, при этом чем быстрее игрок разложил пасьянс, тем лучше результат. Таким образом, наилучшим считается наименьшее количество набранных игровых очков.

Таблица результатов может быть сброшена. Для этого предусмотрена соответствующая кнопка «Сброс».

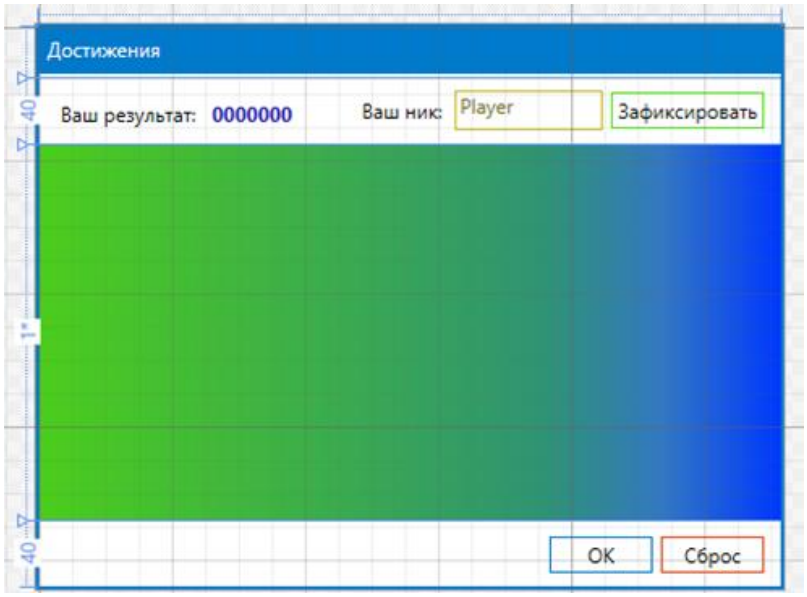

Рисунок 5.4 – Макет окна просмотра рекордов игроков.

На рисунке 5.5 приведена структура записи одного рекорда, которая содержит:

- Наименование разновидности пасьянса;
- Имя игрока, введенное при фиксации результата;
- Дата создания записи;
- Количество набранных очков.

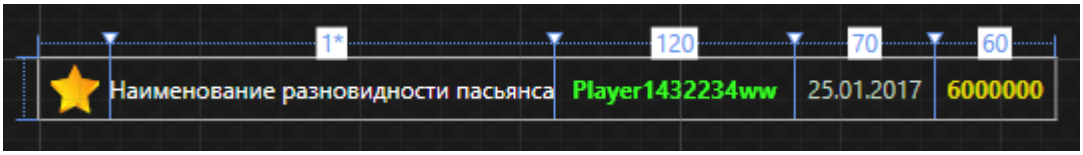

Рисунок 5.5 – Содержимое рекорда.

5.2.3. Окно выбора игры

Окно выбора разновидности пасьянса предоставляет пользователь возможность выбрать наиболее интересную для него разновидность игры. Макет этого окна приведен на рисунке 5.6.

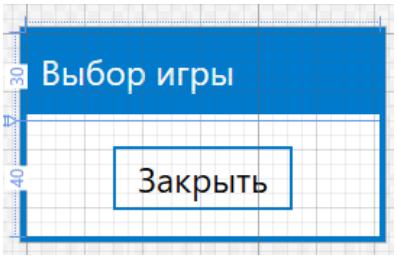

Рисунок 5.6 – Макет окна выбора игры.

По умолчанию макет окна пустой и не содержит никаких доступных вариантов игры. Все возможные варианты пасьянса собираются на этапе инициализации приложении при его запуске посредством механизмов. В результате операций, выполненных этими механизмами, в окне выбора игры формируется список компонентов, инициирующий запуск расклада пасьянса указанной модификации. Макет такого компонента приведен на рисунке 5.7.

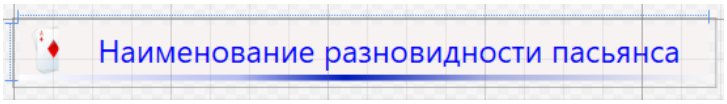

Рисунок 5.7 – Компонент выбора разновидности игры.

5.2.4. Окно описания правил игры

Окно описания правил игры доступно только во время самого игрового процесса, когда какой-либо вариант пасьянса был уже выбран и запущен. Описание правил игры приводится в компоненте типа `WebBrowser` в виде форматированного HTML-кода.

Макет окна доступа к просмотру правил выбранной игры приведен на рисунке 5.8.

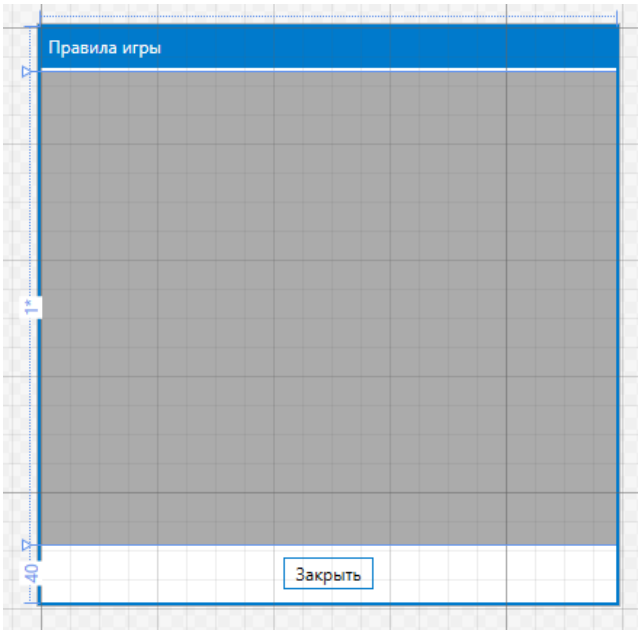

Рисунок 5.8 – Макет окна просмотра правил игры.

5.3. Проект интерфейсных компонентов объектов игры

5.3.1. Компонент карты

Компонент карты реализован на базе элемента Grid с заданными размерами (рисунок 5.10). Вложенный в Grid элемент Image призван отображать изображение карты, подключаемое из ресурсов проекта в двух случаях:

- при инициализации карты в конструкторе;
- в реализации метода filpCard, «переворачивающего» карту.

Поддержка маршрутизируемых событий делает возможным формирование процедур игровой логики, которые связаны с такими событиями, как:

- нажатие левой кнопки мыши – инициируют в управляющем классе Game процедуры захвата колоды (начиная с текущей карты);
- нажатие правой кнопки мыши – обеспечивают перемещение текущей карты на верх стопки, обеспечивая предпросмотр текущей карты, если она «скрыта за другими» (см. рисунок 5.11, а);
- отпускание правой кнопки мыши – возвращает карту на свое место после процедуры предпросмотра (см. рисунок 5.11, б); процедура предпросмотра задается свойством расположения интерфейсного объекта – Panel.SetZIndex).

В листинге 5.2 приведен программный код реализации этих событий.

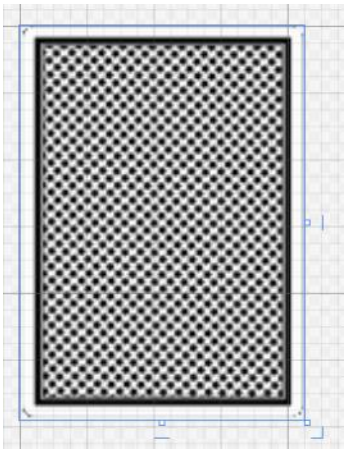

Рисунок 5.9 – Макет компонента карты.

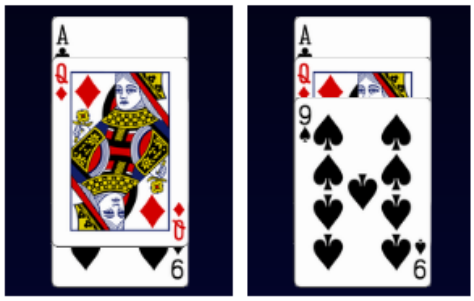

Рисунок 5.10 – Визуализация предпросмотра карты.

Листинг 5.2 – Реализация реакции на события мыши.

```
        // ЛЕВАЯ - НАЖАТЬ: попытка перетаскивания
        private void UserControl_MouseLeftButtonDown(object sender,
System.Windows.Input.MouseButtonEventArgs e)
        {   MyClass.MyDeck.tryCaptureToMove(MyClass);  }
        // ПРАВАЯ - НАЖАТЬ: просмотр карты
        private void UserControl_MouseRightButtonDown(object sender,
System.Windows.Input.MouseButtonEventArgs e)
        { // Если зажата правая кнопка мыши, то показать выбранную карту поверх всех
            Panel.SetZIndex(this, 1);
        }// Обработка отпускания кнопки мыши
         // ПРАВАЯ - ОТПУСТИТЬ: завершение просмотра карты
        private void UserControl_MouseRightButtonUp(object sender,
System.Windows.Input.MouseButtonEventArgs e)
        {// Спрятать карту на своем месте в стопке (если отображена поверх всех)
            if (Panel.GetZIndex(this) != 0)
            { Panel.SetZIndex(this, 0);
              this.MyClass.MyDeck.Ui.UpdateView();
            }
        }
```

5.3.2. Компонент стопки

XAML-конструкция компонента стопки такая же, как и у компонента карты (рисунок 5.12). Это обусловлено тем, что по умолчанию стопка считается пустой и имеет размер для помещения одной карты. Элемент Image стопки служит для задания фона стопки – фон задается только для стопок, для которых жестко определено достоинство карты, которая должна лежать первой в стопке.

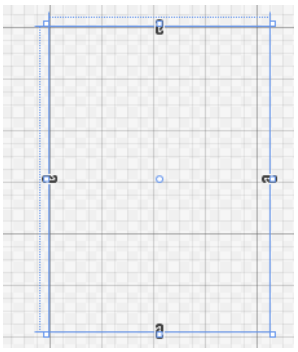

Рисунок 5.11 – Макет компонента стопки.

5.4. Реализация алгоритмов

5.4.1. Подготовка и составление списка модификаций

Подготовка и составление списка модификаций пасьянса реализуется в модуле пакета XML Parser. Алгоритм работы этого модуля запускается в самом начале при старте приложения и состоит в выполнении следующих действий:

- Поиск доступных XML файлов в текущей директории – обеспечивается методом ReadGameList.
- Для каждого из найденных XML файлов производится парсинг его структуры методом ParseXMLToGameStructrure.

Каждая успешно распознанный файл создает новый вариант игры, представленный компонентом выбора. Каждый такой компонент имеет идентификатор файла, который передается классу Game для формирования и раздачи соответствующего варианта пасьянса. Этот идентификатор представлен полем Game.selectedGameID.

5.4.2. Раздача пасьянса

Раздача расклада пасьянса инициируется методом newPlayOut класса Game. В раздаче задействованы методы инициализации основной стопки раздачи. Раздача первоначального расклада игры состоит из следующих шагов:

1. Инициализируется класс стопки раздачи DistributionDeck.
2. В процессе инициализации генерируются все карты игры – карты в стопку раздачи генерируются последовательно по порядку от двойки до туза, чередуя масти.
3. Вызывается метод Shuffle (byte depth) стопки раздачи, который имитирует перетасовку колоды.
4. После перетасовки колоды раздачи метод newPlayOut класса Game продолжает раздачу пасьянса. При этом в колоде раздачи формируется структура DistributionDeck.SendCardsInfo, которая содержит карты, подлежащие анимации перемещения. Эта структура наполняется в соответствии с правилами, указанными в файле параметризации.
5. У колоды раздачи вызывается метод __DD.SendNextCard(), начинающий процесс анимации раздачи пасьянса.
6. Фиксируется время начала игры (для подсчета очков игрока).
7. По завершении анимации последней розданной карты пасьянс считается готовым для игры.

5.4.3. Инициализация классов стопок

Инициализация классов стопок происходит на начальном этапе создания расклада пасьянса. Для всех типов стопок задаются инициализирующие параметры: количество карт, правила поведения, создается структура для хранения коллекции карт.

Для стопки раздачи на этапе инициализации создается полная коллекция карт игры, которая впоследствии перетасовывется.

5.4.4. Анимация перемещения карт

Анимация перемещения карт используется для раздачи карт на игровое поле в начале игры или по запросу пользователя. Направление перемещения карт при этом: из стопки в стопки, которые предписаны правилами: это могут быть целевые стопки, игровые стопки или стопка перебора.

Поскольку анимация карт происходит всегда из стопки раздачи, то она соодержит основные методы для реализации этой анимации.

Эффект анимации карты базируется на специальных средствах, предоставляемых технологией WPF, из классов пространства имен System.Windows.Media.Animation. Все анимации в WPF основаны не на перерисовке элемента, а на изменении его свойств через интервалы времени. Конкретно для задания положения элемента в WPF используется свойство Margin класса Thickness. Это свойство представляет собой структуру, поля которой указывают расстояния элемента со всех четырех его сторон относительно родительского контейнера. Соответственно, синхронно изменяя эти свойства, можно добиваться эффекта перемещения. Для создания анимаций перемещений в WPF выделен специальный класс ThicknessAnimation.

В Solitair20 анимации карт последовательны – то есть следующая карта начинает анимироваться после того, как закончит предыдущая. Для этого необходимо:

- Наличие единого экземпляра класса static ThicknessAnimation TA для осуществления всех анимаций.
- Привязать к TA событие, возникающее при завершении анимации, чтобы подготовить и начать анимацию следующей карты: TA.Completed += CardAnimationComplete. На этод метод возложены функции по совершению логических перестановок, именно: переложить перемещаемую карту из коллекции стопки-источника в коллекуию стопки назначения, обновить

счетчики, «перекинуть» визуальную компоненту карты UICard из UIDeck стопки-источника в UIDeck стопки назначения.

- Иметь очередь карт для организации последовательных анимациий. Такая очередь реализуется коллекцией структур List<SendCardsInfo> bufferToSend, определенной в классе стопки раздачи. Элемент этой структуры содержит информацию о стопке назначения и флаг, который заставит или не заставит эту карту «перевернуться». Например, при попадании карты в стопку перебора из колоды раздачи карта предварительно поворачивается «рубашкой» вниз.
- Иметь методы, которые будут подготавливать и запускать анимацию. Такие методы содержатся в классе DistributionDeck:
  - ✓ SendCardsToBustDeck – метод подготовки анимации карт в стопку перебора, также может вызвать метод ReBust – метод, который делает перебор стопки перебора (процесс возврата карт из стопки перебора в колоду раздачи).
  - ✓ SendCardsToGameDecks – метод подготовки анимации карт в игровую стопку с «переворачиванием» перемещаемой карты.
  - ✓ SendNextCard – метод непосредственного запуска анимации (с первой карты, находящейся в списке bufferToSend).
  - ✓ Анимация перемещения карты в целевую стопку не требует дополнительной логики, поэтому для нее не предусмотрено соответствующего метода в DistributionDeck, для этого в SendCardsInfo просто задается соответствующий элемент с указанием стопки назначения.
- Иметь универсальный метод обработки каждой конкретной анимации. Такой метод – MoveCard реализуется классом Game. Основное назначение данного метода состоит в том, чтобы, зная стопок источника и назначения анимации, подготовить начальное и конечное значения свойства Margin и запустить анимацию на выполнение.

5.4.5. Манипуляции игрока

Манипуляции игрока состоят в перетаскивании карт из одной стопки в другую. Для оргнизации этого действия была выделена особая буферная стопка – BufferedDeck, в которую перемещаются перекладываемые карты для временного хранения. При перемещении карт реализуется основная логика, предписываемая файлом параметризации модели игры:

1. Захват стопки мышью – вызывается метод класса Deck генеральной колоды – tryCaptureToMove, который «ищет» индекс первой «захваченной» карты в коллекции карт колоды.
2. Перед тем, как вызвать метод захватывания карт, проводится проверка возможности захвата выбранных карт – для этого метод CanCardsMove сопоставляет захватываемую стопку карт с правилом TakeFromRule стопки, из которой пытается осуществить захват. В случае, если захват разрешен, осуществляется переход к следующему шагу.
3. Метод tryCaptureToMove вызывает метод CaptureDeck класса Game. Данный метод формирует буферную колоду из захваченных карт и «запоминает», откуда эти карты были захвачены (чтобы при необходимости можно было вернуть их обратно).
4. Игрок перемещает карты по игровому полю – отслеживается событие перемещения мыши MouseMove. Перемещение реализуется визуально изменением все того же свойства Margin, только его изменение теперь происходите не автоматически, как при расчете анимации, а вычисляется постоянно в методе UpdateLocalPosition буферной колоды BufferedDeck каждый раз, когда вызывается реакция на событие MouseMove.
5. Метод releaseMoving класса Deck – вызывается при замеченном событии «отпускания» кнопки мыши, которое сигнализирует о том, что перемещение завершено. При этом возможны ситуации:
    5.1. Во время «отпускания» мыши буферная стопка находилась над пустой областью игрового поля – в этом случае метод CompleteMoving класса Game возвращает карты туда, откуда были взяты.
    5.2. Во время «отпускания» мыши буферная стопка находилась над какой-либо другой стопкой – в этом случае метод CompleteMoving класса Game помещает карты в стопку назначения (над которой находилась мышь). Перед эти перемещаемая стопка и стопка назначения сопоставляются на совместимость по правилам DeckRules.TopPlaceRuleSequence, DeckRules.TopPlaceRuleMultiplicity, DeckRules.TopPlaceRuleOrder, DeckRules.TopPlaceRuleMultiplicity.
6. Метод CompleteMoving после каждой манипуляции вызывает метод CheckForWin, в котором производится проверка на завершение игры – игра считается завершенной, если сумма всех карт в целевых стопках оказывается равной значению, соответствующему количеству карт, которые изначально генерируются в стопке раздачи.

ЗАКЛЮЧЕНИЕ

В данной работе были рассмотрены принципы объектно-ориентированного проектирования приложений на примере приложения пасьянс «Косынка». Был произведен анализ данного пасьянса и его модификаций с последующей структурной декомпозицией на объекты и компоненты. В результате проведенного анализа была построена объектная модель пасьянсов типа «Косынка», охватывающая все его варианты. Построенная модель пасьянса позволила разработать Windows-приложение, которое характеризуется следующими результатами:

- Современный стильный интерфейс;
- Сохранение стандартных возможностей игры (подсчет очков, настройка внешнего вида);
- Использование визуальных анимационных эффектов;
- Единая объектная модель, позволяющая реализовать множество различных вариантов игры без ее изменения и без вмешательства в программный код приложения;
- Автоматическое определение вариантов игры из соответствующей директории;
- Ориентирование на конкретный вариант игры посредством считывания из структуры XML-файлов и настройки свойств и поведения объектов модели.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Албахари Джозеф– C#. Справочник. Полное описание языка. Пер. с англ. - М: ООО «И.Д. Вильямс», 6-е изд., 2016, 1040 с.
2. Андерсон К. – Основы Windows Presentation Foundation. Пер. с англ. А. Силикина – М.: ДМК Пресс, 2008, 432 с.
3. Бадд Т. Объектно-ориентированное программирование в действии/ Т. Бадд: пер. с англ. А. Бердникова СПб.: Питер, 1997, 460 с.
4. Медведев В.И. Особенности объектно-ориентированного программирования на C++/CLI, C# и Java. 2-е изд., испр. и доп. – Казань: РИЦ «Школа», 2010 – 444 с.
5. Петцольд Ч. – Microsoft Windows Presentation Foundation. Базовый курс, пер. с англ. Е. Матвеев: – М.: Издательство «Русская Редакция»; СПб.: Питер, 2008, 944 с.
6. Пискунов А.Г. Неполное Руководство по SQLite для пользователей Windows, 2015, Пер.: А.Г.Пискунов, ua.agpl.SQLite.AllOw 0.70 – 119 с.
7. Andrew Troelsen – Pro C# 5.0 and the .NET 4.5 Framework: Apress, 6th edition: NY, 2012, 1534 p.
8. Jay A. Kreibich. Using SQLite. – O'Reilly Media, Inc., First Edition – 2010, 528 p.
9. Svetlin Nakov & Co. – Fundamentals of computer programming with C#. The Bulgarian C# book: – Sofia, 2013, 1122 p.
10. Visual Paradigm for UML 13. User's guide: Software documentation – user's manual - 1485 p.
11. UML 2.0. Superstructure specification. – Final adopted specification – ptc/03-08-02; 2003, 640 p.
12. Введение в C# - [Электронный ресурс]: https://metanit.com/sharp/ (дата обращения: 29.12.2016)
13. Пасьянс – Википедия. [Электронный ресурс]: https://ru.wikipedia.org/wiki/Пасьянс (дата обращения: 29.12.2016)
14. SQLite – Википедия. [Электронный ресурс]: https://ru.wikipedia.org/wiki/SQLite (дата обращения: 29.12.2016)

Приложение А – Связи реализации вариантов использования Solitair20 компонентами (окнами) интерфейса приложения

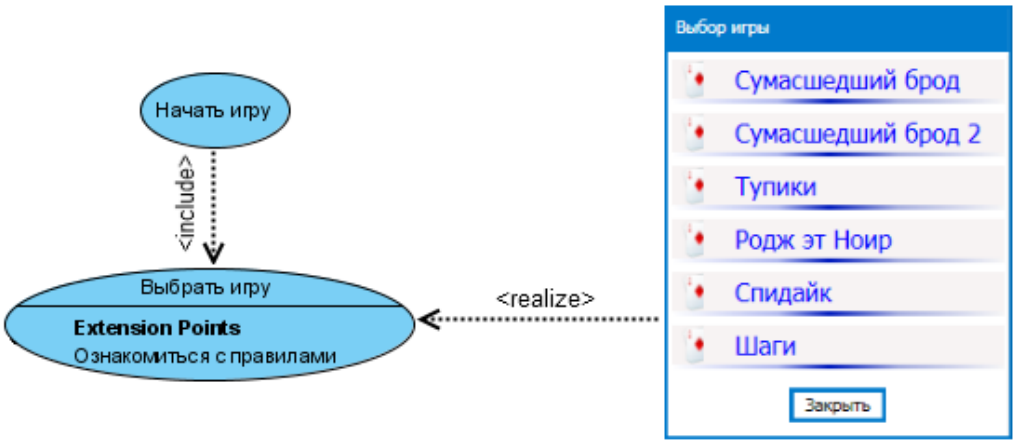

Рисунок А.1 – Реализация прецедента «Выбрать игру».

Сценарий прецедента:

1. Игрок выбирает команду меню «выбрать игру».
2. Формируется список доступных модификаций пасьянса.
3. Игрок выбирает подходящий ему вариант игры.

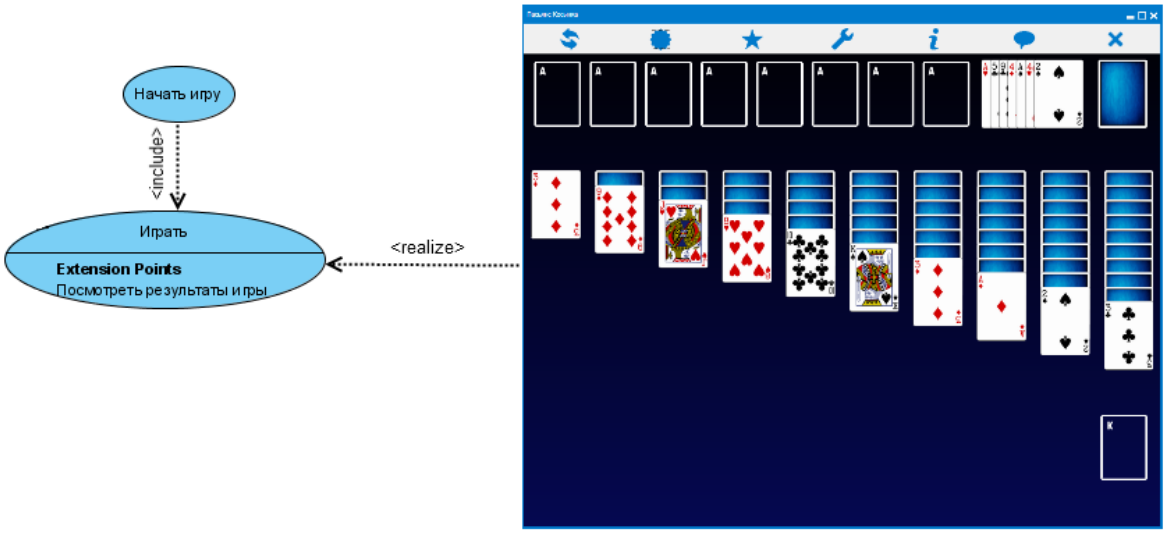

Рисунок А.2 – Реализация прецедента «Играть».

Сценарий прецедента:

1. Игрок выбрал вариант игры (прецедент А.1).
2. Система сформировала и раздала исходный расклад пасьянса.
3. Игрок осуществляет игровые манипуляции.

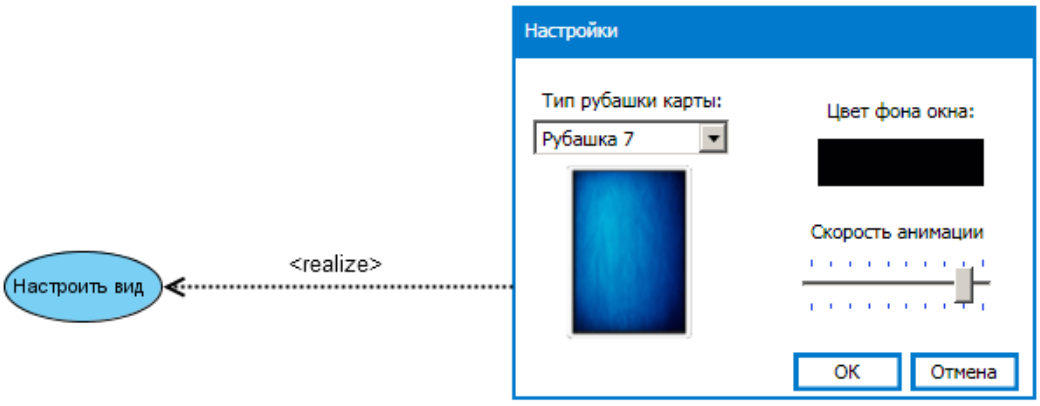

Рисунок А.3 – Реализация прецедента «Настроить вид».

Сценарий прецедента:

1. Игрок выбирает команду меню «настройки отображения».
2. Система отображает окно настроек отображения игрового поля.
3. Игрок выбирает подходящий настройки: вид «рубашки», цветовую схему, скорость анимации.
4. При подтверждении изменений программа применяет выбранные установки.

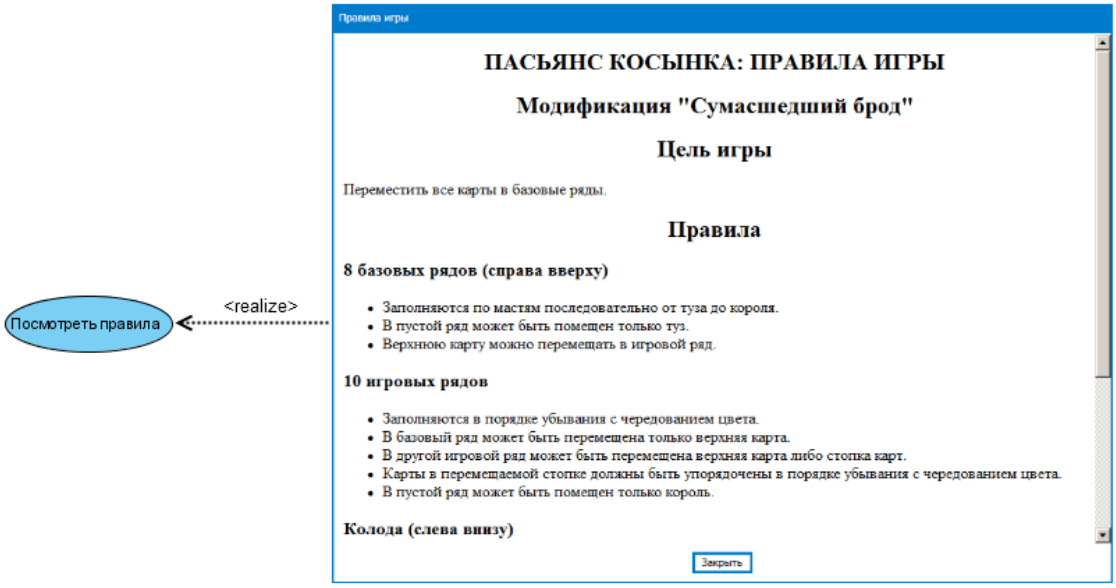

Рисунок А.4 – Реализация прецедента «Просмотреть правила»

Сценарий прецедента:

1. Игрок осуществляет игровые манипуляции (прецедент А.2).
2. Игрок выбирает команду меню «посмотреть правила игры».
3. Система показывает окно с описанием правил текущей игры.

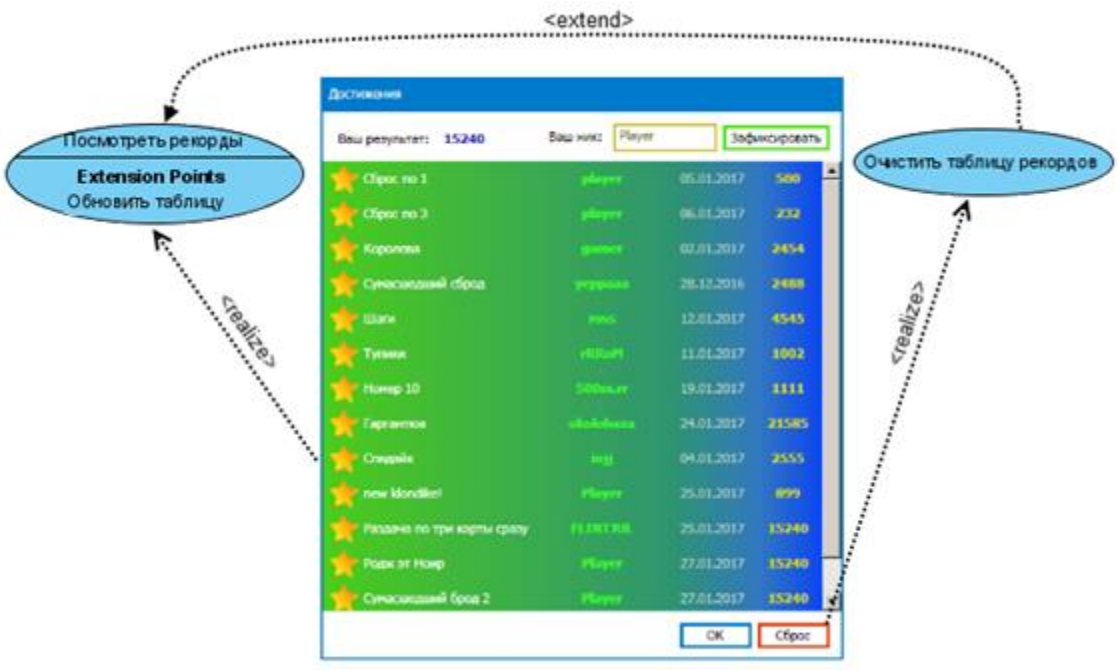

Рисунок А.5 – Реализация прецедентов «Просмотреть рекорды» и «Очистить таблицу рекордов».

Сценарий прецедента – вариант 1:

1. Игрок выбирает команду меню «просмотр таблицы рекордов».
2. Формируется и отображается таблица рекордов из БД.
3. Точка ветвления 1.

Сценарий прецедента – вариант 2:

1. Игрок осуществляет игровые манипуляции (прецедент А.2), которые привели к победному раскладу.
2. Формируется и отображается таблица рекордов из – приводится результат, достигнутый игроком.
3. Игрок вводит свое имя и нажимает кнопку «Зафиксировать».
4. Если достигнутый игроком результат лучше имеющегося в таблице (по данной разновидности пасьянса), то результат перезаписывается с указанием имени игрока и даты.
5. Точка ветвления 1.

   Точка ветвления 1:

   1. Игрок нажимает кнопку «ОК» - окно рекордов закрывается.
   2. Игрок нажимает кнопку «Сброс» - рекорды стираются из БД.

Уважаемый пользователь! Обращаем ваше внимание, что система «Антиплагиат» отвечает на вопрос, является ли тот или иной фрагмент текста заимствованным или нет. Ответ на вопрос, является ли заимствованный фрагмент именно плагиатом, а не законной цитатой, система оставляет на ваше усмотрение.

Отчет о проверке № 1

дата выгрузки: 30.05.2017 16:40:55
пользователь: arcimovich@gmail.com / ID: 4098447 отчет предоставлен сервисом «Антиплагиат» на сайте http://www.antiplagiat.ru

Информация о документе

№ документа: 2
Имя исходного файла: diplom.docx
Размер текста: 1502 кБ
Тип документа: Не указано
Символов в тексте: 84784
Слов в тексте: 9920
Число предложений: 508

Оригинальность: 99.83%
Заимствования: 0.17%
Цитирование: 0%

Информация об отчете

Дата: Отчет от 30.05.2017 16:40:55 - Последний готовый отчет
Комментарии: не указано
Оценка оригинальности: 99.83%
Заимствования: 0.17%
Цитирование: 0%

Источники

Доля в тексте	Источник	Ссылка	Дата	Найдено в
0.17%	[1] Пасьянс «Косынка»	http://ru.wikipedia.org	03.12.2012	Модуль поиска Интернет

Текст отчета

Министерство образования и науки Российской Федерации
НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ (НИ ТГУ)
Факультет информатики
Кафедра программной инженерии (ПИ)
ДОПУСТИТЬ К ЗАЩИТЕ В ГЭК
Руководитель ООП
_____А.Л.Фукс
«_____»_____2017 г.
ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА БАКАЛАВРА
РАЗРАБОТКА МОДЕЛИ ПАСЬЯНСОВ ТИПА “КОСЫНКА”
И ЕЁ РЕАЛИЗАЦИЯ
по основной образовательной программе подготовки бакалавров направление подготовки 02.03.02 – Фундаментальная информатика и информационные технологии
Арцимович Роман Александрович Руководитель
ВКР
доц. каф. программной инженерии
_____А.М.Бабанов подпись
«_____»_____2017 г. Автор
работы
студент группы № 1431 _____Р.А.Арцимович
подпись
Томск – 2017
РЕФЕРАТ
Дипломная работа: 61 с., 30 рис., 1 табл., 14 источников, 1 прил.
ПАСЬЯНС, КОСЫНКА, КАРТОЧНАЯ ИГРА, МОДЕЛЬ, UML, XML, XAML, WPF, VISUAL STUDIO, C#.
Объект исследования – объектная модель, реализованная в виде Windows-приложения.
Цель исследования – осуществить построение объектной модели карточной игры пасьянс «Косынка», интегрировать эту модель в программную оболочку, способную реализовать многообразие вариантов игры.
Метод исследования – теоретическое исследование и практическая реализация.
Область применения – приложение для ОС семейства Windows.
Результаты работы – детально изучены и проанализированы процессы и принципы игры в пасьянсах типа «Косынка», произведена систематизация результатов анализа, на основе которой разработана объектная модель игры, охватывающая многообразие вариантов ее реализации. Были изучены основные современные технологии разработки Windows-приложений – с их помощью осуществлена реализация приложения пасьянс «Косынка» с гибкими возможностями по реализации различных вариантов игры. Приведено детальное описание разработанной модели и приложения.
ОГЛАВЛЕНИЕ
РЕФЕРАТ 2

ОГЛАВЛЕНИЕ 3
ВВЕДЕНИЕ 5
1. ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ 6
1.1. Описание игры пасьянс "Косынка" 6
1.2. Модификации игры 7
1.3. Постановка задачи 8
2. ОПИСАНИЕ ТЕХНОЛОГИЙ МОДЕЛИРОВАНИЯ И РЕАЛИЗАЦИИ 9
2.1. Технология разработки WPF 9
2.2. Язык проектирования интерфейса приложения XAML 10
2.3. Язык программирования C# 10
2.4. Хранение данных с помощью XML 11
2.5. Проектирование программного обеспечения 11
2.6. СУБД SQLite 12
3. РАЗРАБОТКА МОДЕЛИ 13
3.1. Взаимодействие пользователя с игрой 13
3.2. Объектное проектирование 14
3.2.1. Основные объекты игры 14
3.2.2. Класс подготовки и управления игровым процессом – Game 26
3.2.3. Представление объектов в интерфейсе игры 29
3.2.4. Представление модификаций игры 30
3.2.5. Диаграмма пакетов 34
3.2.6. Диаграмма классов 35
3.3. Проектирование модели поведения 38
3.3.1. Общий алгоритм подготовки игры и игрового процесса 38
3.3.2. Диаграмма состояний класса Game 39
4. ПРОЕКТИРОВАНИЕ ПРИЛОЖЕНИЯ 40
4.1. Структура приложения 40
4.2. Проектирование компонентного состава приложения 41
4.2.1. Диаграмма компонентов приложения 41
4.2.2. Диаграмма компонентов проекта 42
5. РЕАЛИЗАЦИЯ ПРИЛОЖЕНИЯ 45
5.1. Проект интерфейса основного окна 45
5.2. Проект интерфейса вспомогательных окон 47
5.2.1. Окно настроек игры 47
5.2.2. Окно рекордов 48
5.2.3. Окно выбора игры 49
5.2.4. Окно описания правил игры 50
5.3. Проект интерфейсных компонентов объектов игры 51
5.3.1. Компонент карты 51
5.3.2. Компонент стопки 52
5.4. Реализация алгоритмов 53
5.4.1. Подготовка и составление списка модификаций 53
5.4.2. Раздача пасьянса 53
5.4.3. Инициализация классов стопок 54
5.4.4. Анимация перемещения карт 54
5.4.5. Манипуляции игрока 55
ЗАКЛЮЧЕНИЕ 57
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ 58
Приложение А – Связи реализации вариантов использования Solitair20 компонентами (окнами) интерфейса приложения 59

ВВЕДЕНИЕ

Пасьянс – карточная игра для одного человека (от франц. Patience – терпение). Целью любого пасьянса ставится достичь некоторого расположения карт на игровом поле. При этом для пасьянса установлен набор правил, по которым можно карты перекладывать между стопками. Цель игры также определяется правилами конкретного пасьянса. Результат игры достигается с помощью интеллектуальных усилий играющего. При формировании начального состояния игры (расклада) присутствует фактор случайности, что вносит разнообразие в процесс игры и делает каждый сеанс уникальным. Одной из разновидности карточных пасьянсов является пасьянс под названием «Косынка». Данный пасьянс традиционно поставляется в составе развлекательной составляющей операционной системы семейства Windows, начиная еще с версии Windows 3.0. Также существует множество разновидностей данного вида пасьянсов.

ОБЗОР ПРЕДМЕТНОЙ ОБЛАСТИ

Описание игры пасьянс "Косынка"

Пасьянс «Косынка» как компьютерная игра был разработан в 1989 году сотрудником корпорацииMicrosoftУэсом Черри.

Классический вариант пасьянса «Косынка» имеет игровую колоду карт, состоящую из 52 карт – набор от двойки до туза в каждой масти. В начале игры 28 карт из стартовой колоды раскладывают в 7 игровых столбцов, при этом начальное количество карт в каждом из них соответствует номеру столбца (в первом - одна, во втором - две и т.д., в последнем, седьмом - семь карт). Карты в столбцах изначально закрыты (перевернуты «рубашкой» вверх), кроме самой верхней – ее кладут открытой.

Оставшиеся карты составляют колоду, которую используют для раскладки пасьянса с помощью переборов (по одной или по три карты). На рисунке 1.1 приведен исходный вид пасьянса сразу после раздачи в операционной системе Windows 98. Рисунок 1.1 – Исходный вид пасьянса после раздачи.

В верхней части игрового поля помещаются целевые стопки, которые являются объектами основной цели игры. Игроку требуется разложить все выданные карты (манипулируя перемещениями карт в игровых стопках и стопке перебора и раздачи) в целевые стопки по возрастанию их достоинства
- начиная от туза и заканчивая королём. Игрок, собравший все карты в четыре целевые стопки в нужном порядке, является победителем. В классической версии пасьянса по окончанию игры карты «разлетаются» по игровому полю.

В игре ведется простейший подсчет очков, основанный на времени, затраченном на получение победного расклада. Чем меньше затрачено времени на прохождение сеанса игры, тем больше очков получает игрок.

В классической игре игрок имеет дополнительные возможности, связанные с элементарной настройкой игрового процесса и внешнего вида игры:

выбор цвета «рубашки» карт, выбор способа подсчёта[1] очков (стандартные очки, виртуальные деньги,[1] без подсчета), выбор способ сдачи карт из колоды[1] раздачи (по одной или по трём), а[1] также и включение или выключение отсчёта времени.[1]

Модификации игры

Пасьянс «Косынка» в настоящее время имеет множество разновидностей, отличающихся друг от друга в различной степени по одному или нескольким параметрам. К таким параметрам можно отнести правила организации перемещений карт между стопками, количество карт и стопок, способы раздачи и перебора карт, и так далее.

В данной работе за основу для анализа был взят набор пасьянсов от Алексея Аношенко «Коллекция пасьянсов 250+». Данная коллекция включает 37 разновидностей пасьянса «Косынка». Анализ всех 37 разновидностей «Косынки» показал, что основные типы различий, дифференцирующие варианты игры, заключаются в:

Количестве карт в игре.

Количестве игровых и целевых стопок.

Типе перебора карт в основной стопке раздачи.

Правилах перемещений карт между стопками.

Правилах размещении карт в игровых стопках.

Начальных условиях раздачи.

Наличии дополнительных стопок.

Других мелких различиях.

Постановка задачи

В рамках данной работы ставятся следующие задачи:

Выявить и систематизировать различия в составе и поведении пасьянсов типа “Косынка”.

Обобщить все выявленные различия, выделить их описательные структуры и закономерности для разных вариантов игры.

Разработать общую объектную модель игры, охватывающую все отобранные варианты.

Разработать игровое приложение – пасьянс «Косынка» - версия Solitair20.

Варианты игры были выбраны из указанного ранее пакета «Коллекция пасьянсов 250+». Отбор вариантов обусловлен критериями, которые максимально широко охватывают степень различия вариантов игры и представляют наибольший интерес для реализации. В отобранные варианты вошли следующие типы игр:

«Косынка сброс по 1», «Косынка сброс по 3», «Косынка 2 сброс по 1», «Косынка 2 сброс по 3», «Тупики», «Гаргантюа», «Номер 10», «Звание и досье», «Красный и черный», «Палец и мешок», «Уайтхед», «Двойная косынка 1», «Двойная косынка 2», «Королева», «Родж эт Наир», «Сумасшедший брод», «Сумасшедший брод 2», «Западный обрыв», «Шаги», «Спидайк», «Австралийский пасьянс», «Аляска».

ОПИСАНИЕ ТЕХНОЛОГИЙ МОДЕЛИРОВАНИЯ И РЕАЛИЗАЦИИ

В данной главе описаны технологии, которые применяются при разработке приложения Solitair20. Применяемые технологии включают:

WPF – основная концепция создания приложения;

XAML – вспомогательный язык системы WPF, предназначенный для проектирования интерфейса приложения;

C# - объектно-ориентированный язык программирования, применяемый для создания функциональной логики приложения;

XML – язык разметки, описывающий в данном применении конфигурации вариантов игры;

UML – язык унифицированного моделирования, используемый при проектировании архитектуры и поведения программного обеспечения.

SQLite – портативная (компактная) СУБД, реализующая информационное обеспечение приложения;

Технология разработки WPF

Технология WPF (Windows Presentation Foundation) предсатавляет собой принципиально новый (по сравнению с GDI, Windows Forms) подход к созданию приложений. WPF – это подход к разработке программного обеспечения на базе принципа «всем миром» [2, с.13]. Технология WPF сумела объединить в себе документы, формы и мультимедийный контент в единый пакет, использующий принципы императивного и декларативного программирования. Декларативное программирование основывается на языке разметки интерфейса XAML, императивное – на базе современных языков программирования – C# или Visual Basic.

В графической основе классических приложений Windows Forms состоит GDI/GDI+ (Graphical Device Interface), являющийся компонентом, который в тесном взаимодействии с ядром API Windows составляет базовый пользовательский интерфейс программ. GDI может быть применен в простых игровых приложениях, не требующих сложной графики и высокой степени динамичности, однако его нельзя применять для качественного выполнения сложной и быстрой анимации, поскольку он не поддерживает синхронизацию с кадровым буфером (участком в памяти, хранящем кадры перед их непосредственным выводом на устройство).

Технология WPF в своей графической основе использует DirectX, что повышает производительность WPF-приложений за счет возможности применения аппаратного ускорения графики. Пользовательский интерфейс приложений, использующих WPF, построен на базе векторной графики, что дает независимость от разрешения конечного устройства. Основные преимущества графической подсистемы WPF:

Независимость от разрешения экрана вывода и устройства – единица измерения графики в WPF – это аппаратно-независимый пиксель, равный 1/96 части дюйма и не зависящий от разрешения экрана.

Поддержка эффектов анимации объектов.

Использование аппаратного ускорения.

Встроенная проработанная библиотека 2-D и 3-D объектов.

Язык проектирования интерфейса приложения XAML

XAML (eXtesible Application Markup Language) представляет собой расширяемый язык разметки приложений – декларативный язык проектирования пользовательского интерфейса. Структура языка XAML основывается на XML.

Язык XAML является всего лишь вспомогательным средством проектирования и компоновки интерфейса приложения, его использование вообще не обязательно, однако, как показывает практика, применение XAML в проектах разработки дает свои несомненные преимущества:

Более наглядное представление структуры интерфейса проектируемого приложения, обеспечиваемое благодаря иерархичности языка разметки. Разделение визуальной и функциональной частей приложения, что позволяет разделить разработку на отдельно программистов и отдельно дизайнеров.

Компактность и высокая степень читаемости и поддержки кода, написанного на XAML – как правило, код, написанный на языке разметки, гораздо меньше по объему эквивалентного кода, написанного на процедурном языке. Поддержка применения стилей интерфейса.

XAML в основном используется для компоновки макета форм приложения и инициализации обработчиков событий. Сами же обработчики должны быть запрограммированы на процедурном языке (таком, как, например, C#). Такая концепция позволяет осуществлять проект дизайна приложения с одной стороны, пока с другой стороны программист будет наполнять функциональную логику, также основанную на взаимодействии компонентов интерфейса.

Язык программирования C#

C# - один из самых мощных, быстро-развивающихся и востребованных языков программирования. C# является объектно-ориентированным, он поддерживает в полной мере полиморфизм, механизмы инкапсуляции и наследования, перегрузку операторов, лямбда-выражения и многие другие современные возможности быстрой и качественно разработки.

Язык C# является основным средством реализации технологии .NET. Основой фреймворка .NET является применяемая в рамках него общеязыковая исполняющая среда – CLR (Common Language Runtime). Код, выполняемый в CLR, является управляемым. CLR умеет автоматически управлять памятью и имеет сборщик мусора, который выполняется совместно с программой и восстанавливает память, которая занята объектами, на которые больше нет ссылок. Таким образом, отпадает необходимость контроля за своевременным освобождением памяти, что необходимо при программировании, например, на C++.

Цель языка C# - обеспечение продуктивности работы программистов. Для этого в языке соблюдается баланс между простотой, выразительностью и производительностью.

Хранение данных с помощью XML

Язык проектирования XML создавался как универсальный язык разметки, применяемый в различных областя.

В приложении Solitair20 язык XML используется для хранения структуры данных, определяющих разновидности пасьянса, таким образом реализуя процесс сериализации – преобразования объекта (структуры игры) в форму, подготовленную для эффективного использования в логике программы.

Проектирование программного обеспечения

UML (Unified Modelling Language) – унифицированный язык моделирования, который использует графическое описание проектируемой системы и позволяет построить статическую и динамическую модели программного продукта с точки зрения его: логического представления, представления реализации, представления процесса функционирования и представления размещения и состава компонентов.

На рынке программного обеспечения представлено множество продуктов (CASE-средств), которые предназначены для создания UML-моделей приложений. К наиболее известным можно отнести Rational Rose, Power Designer, StarUML, Enterprise Architect, и другие. Все эти продукты полностью поддерживают спецификацию UML 2.0 и обладают мощным инструментарием.

В рамках данной работы для создания моделей приложения Solitair20 применено CASE-средство, представленное программой Visual Paradigm for UML2.0 версии 7.2 (VP). Выбор был сделан в пользу данного продукта, так как он обладает наибольшим удобством среди всех рассмотренных. Кроме того, VP поддерживает средства forward engineering, что позволяет сгенерировать файлы шаблонов исходного кода, исходя из построенной модели (например, диаграммы классов).

СУБД SQLite

Для хранения информации о рекордах сеансов игр разных типов разрабатываемый продукт будет использовать локальное хранилище, исполненное в виде файла базы данных. Для таких решений идеально подходит СУБД SQLite со своими возможностями.

SQLite – компактная встраиваемая реляционная база данных, которая имеет исходные коды в открытом доступе. Встраиваемая – означает, что SQLite не использует устоявшуюся парадигму «клиент-сервер», то есть ядро SQLite не является отдельно работающим процессом, с которым взаимодействует программа, а представляет собой библиотеку, с которой программа компонуется, а само ядро становится частью программы.

Основные преимущества СУБД SQLite, которые определили ее выбор в качестве применяемого хранилища в данном проекте, описаны в [6, с.23] и определяются следующими характеристиками:

Отсутствие сервера.

Отсутствие необходимости конфигурирования и установки СУБД.

Кроссплатформенность.

Интеграция ядра СУБД в сам файл базы данных.

Высокая надежность. Производительность.

РАЗРАБОТКА МОДЕЛИ

В данной главе представлены основные аспекты, которые использованы при проектировании модели будущего приложения Solitair20. Здесь представлены диаграммы UML, характеризующие основные стороны проекта приложения – статическую, объектную и поведенческую. Диаграммы выполнены в системе VP, а в качестве иллюстраций приведены фрагменты копий экрана этой системы.

Взаимодействие пользователя с игрой

Взаимодействие программной системы с внешней средой (пользователями) описывается диаграммами вариантов использования (прецедентов). В [4, с. 93] диаграмма вариантов использования определена как модель, представляющая разрабатываемую программу как систему в виде совокупности вариантов ее использования пользователем. На этой диаграмме показывается внешняя сторона системы (программы), не учитывая ее внутреннюю структурную и архитектурную составляющую. Диаграмма должна отображать все предоставляемые программой связи с пользователем.

Диаграмма вариантов использования приложения Solitair20 приведена на рисунке 3.1.

Рисунок 3.1 – Диаграмма вариантов использования.

Из диаграммы 3.1 прослеживаются основные действия, которые игрок может выполнять в программе:

Настройка внешнего вида отображения в игре.

Начало игры, которое сопровождается предварительным выбором типа игры и самим игровым процессом (манипуляциями с картами с целью получения выигрышной комбинации).

Просмотр таблицы рекордов, достигнутых по каждому виду игры.

Из топологии диаграммы прецедентов видно, что:

Таблица рекордов может быть вызвана в любой момент игры, а также вызывается после каждого завершенного сеанса игры.

Таблица рекордов может быть очищена.

В процессе выбора игры пользователь может просмотреть правила типа игры.

Прецеденты реализуются в программе посредством экранных форм пользовательского интерфейса. В приложении А показана связь каждого варианта использования с окном интерфейса, реализующим этот вариант

Объектное проектирование

Основные объекты игры

Основной этап объектного проектирования приложения заключается в выделении объектов предметной области, установлении их основных свойств и связей.

Пасьянс – карточная игра, поэтому логично, чтобы основными объектами этой игры стали карты. Карты в игре сами по себе не существуют, а объединяются в стопки. Таким образом, карты и стопки становятся основными объектами приложения, связанными отношениями агрегации.

Исследования различных вариантов игры показали, что в игре может существовать до 5 видов стопок, имеющих различную функциональность и назначение. В то же время все типы стопок имеют много общего. На рисунке 3.2 показан исходный расклад варианта игры «Сумасшедший брод» - эта разновидность «Косынки» характеризуется наличием всех типов стопок (в соответствии с обозначениями на рисунке):

Целевые стопки (Task Decks) – стопки, в которые в конечном итоге собираются все карты с игрового поля. Присутствуют во всех разновидностях игры. Игровые стопки (Game Decks) – вертикальные ряды, которые являются основными для совершения промежуточных игровых манипуляций. Игровые стопки определены для всех разновидностей игры.

Стопка раздачи (Distribution Deck) – основная, первоначальная стопка, изначально содержит все карты игры, определенное количество которых при задании исходного расклада перемещаются в другие стопки в соответствии с правилами разновидности пасьянса. Стопка присутствует во всех разновидностях игры, однако, в некоторых все карты этой стопки раздаются в игровое поле, делая эту стопку пустой во время игры.

Стопка перебора (Bust Deck) – вспомогательная стопка, в которую карты попадают из стопки раздачи по определенному правилу. Стопка присутствует не во всех разновидностях игры.

Дополнительная стопка (Additional Deck) – еще одна не обязательная стопка, которая является неким буфером в игре и определяется установленным правилами поведением.

Рисунок 3.2 – Схема стопок в игре.

Объект, представляющий карту в игре, характеризуется двумя основными параметрами – масть (Suit) и достоинство (Advantage). Средствами языка программирования масть карты может быть задана как нумератор, имеющий 4 значения (см. листинг 3.1). Листинг 3.1 – Определение нумератора масти карты.

```
public enum Suit
{
Clubs, // крестовая
Hearts, // червовая
Spades, // пиковая
Diamonds // бубновая
}
```

Аналогично задается нумератор достоинства карты, который имеет 13 значений (см. листинг 3.2).

Листинг 3.2 – Определение нумератора достоинства карты. public enum Advantage

```
{
Any = 0, // не важна (любая)
```

Two = 2, // 2
Three, // 3
Four, // 4
Five, // 5
Six, // 6
Seven, // 7
Eight, // 8
Nine, // 9
Ten, // 10
Jack, // валет
Оиееп, // дама
King, // король
Ace // туз
}
Таким образом, указанные нумераторы являются неотъемлемыми атрибутами карты – свойствами соответствующего класса карты (Card). Класс объекта карта, помимо значимых характеристик карты, также должен содержать дополнительные свойства, такие как: Тип отображения на игровом поле («рубашкой» вверх или вниз – face);
Визуальную компоненту - ui;
Настройку отображения (тип или текстуру «рубашки») – cardBack; Ссылку
на стопку, в который находится карта – myDeck.
Помимо конструктора, в классе Card необходимо предусмотреть метод, который будет «переворачивать» карту в стопке. Таким методом является метод filpCard().
Предварительный каркас класса карты приведен в листинге 3.3. Листинг
3.3 – Каркас класса карты.
public class Card
{
// =true - рубашкой вниз
private bool face;
// масть private
Suit suit; //
достоинство
private Advantage advantage; //
визуальная компонента
private UICard ui;
// колода, в которой карта находится в данный момент private
Deck myDeck;
// тип выбранной рубашки карты
private int cardBack = 7;
// конструктор ---------------------------------
public Card(Suit suit, Advantage advantage, bool face)
// перевернуть карту
public void filpCard()
}
Объекты из множества типов «стопка» обладают более расширенными свойствами, определяющими их различия и виды. Одновременно с этим, во всех объектах этого типа можно выделить много общих признаков и свойств. Поэтому должен создан быть некий генеральный абстрактный класс стопки (Deck), который будет «порождать» все типы стопок в игре на этапе инициализации игры, реализация каждой из которых будет происходить посредством переопределения свойств и методов, наследуемых от генерального класса. Генеральный класс должен предоставлять своим потомкам такие свойства, как:
Количество карт – count;
Набор правил поведения – deckRules;
Ориентация на игровом поле – orientation;
Визуальная компонент – ui;
Набор карт – cards, который лучше всего представить структурой строго типизированных объектов Card, коллекцией List<Card>, предоставляемой в .NET.
Ориентация стопок на игровом поле может реализовываться в виде ряда, ориентированного горизонтально или вертикально, либо в виде одной стопки, отображающей одновременно одну карту (верхнюю). В коде программы ориентация стопки задается соответствующим нумератором (см. листинг 3.4).
Листинг 3.4 – Нумератор ориентации стопки.
// Способ размещения колоды (стэка) public
enum PlacingMethod
{
TopOnlyVisible, // видна только верхняя
HorizontalLine, // в горизонтальную линию
VerticalLine // в вертикальную линию
}
Обобщив результаты исследования вариантов реализации «Косынки», был выделен набор правил поведения, которыми может быть охарактеризован любой тип стопки. Данный набор правил можно представить структурой нумераторов. Правила поведения стопок включают одно правило взятия карт из стопки и 4 правила перемещения карт в стопку. Листинг 3.5 иллюстрирует структуру правил поведения стопки. Листинг 3.5 – Структура правил поведения стопки.
// Структура: основные правила колоды
public struct DeckRules
{
public TakeFromRule takeFromRule; // правило взятия карт
// правило помещения карт в колоду - порядок public
TopPlaceRuleOrder topPlaceRuleOrder;
// правило помещения карт в колоду - последовательность
public TopPlaceRuleSequence topPlaceRuleSequence; //
правило помещения карт в колоду - количество

public TopPlaceRuleMultiplicity topPlaceRuleMultiplicity;
// правило помещения карты в самый низ (достоинство первой карты) public
Advantage bottomCard;
}
Правило взятия карт из стопки определяет, какие карты могут быть перемещены из данной стопки в другую. Предпроектные исследования показали, что различные варианты реализации «Косынки» дают игроку право перетаскивать между стопками как одну карту, так и целые стопки (или части стопок), строго регламентируя последовательность расположения карт в перемещаемой стопке – устанавливая ограничения на чередование по цвету масти, достоинству, порядок сортировки, и т.д. В листинге 3.6 приведен нумератор соответствующего правила. Листинг 3.6 – Структура правил поведения стопки.
// ПРАВИЛО СТОПКИ - взятие карт из стопки
public enum TakeFromRule
{
Forbidden, // запрещено
OneCard, // только одну сверху
AnySequence, // любую последовательность
ColorAlternation, // последовательность с чередованием цвета (порядок не важен)
SingleColor, // последовательность одного цвета (порядок не важен)
SingleSuit, // последовательность одной масти (порядок не важен)
ColorAlternationArranged, // последовательность с чередованием цвета (по порядку достоинства)
SingleColorArranged, // последовательность одного цвета (по порядку достоинства)
SingleSuitArranged // последовательность одной масти (по порядку
достоинства) }
Следующие 4 правила, наоборот, определяют возможность переместить карты, взятые из другой стопки, в данную.
Правило количества. Данное правило определяет количество карт, которое может быть одновременно перемещено в данную стопку. Возможные варианты: одну, несколько, либо запрет на перемещение. Последний вариант, например, применим во всех вариантах игры к стопке раздачи, поскольку в стопку раздачи нельзя перемещать карты из других стопок. Код нумератора по данному правилу приведен в листинге 3.7.
Листинг 3.7 – Нумератор правила по количеству
// ПРАВИЛО СТОПКИ - положить карты на верх стопки - количество
public enum TopPlaceRuleMultiplicity
{
None, // запрещено
OnlyOne, // можно только одну
Deck // можно стопку
}
Правило порядка сортировки. Данное правило устанавливает порядок сортировки в стопке карт, которая может быть перемещена в данную. Данное правило применимо только к стопкам, в которых установлено TopPlaceRuleMultiplicity = Deck. Вариантов порядка сортировки может быть два – восходящий (от двойки до туза), либо нисходящий (от туза до двойки). Также предусмотрен вариант, позволяющий игнорировать порядок сортировки. В листинге 3.8 приведен исходный код соответствующего нумератора.
Листинг 3.8 – Нумератор правила порядка сортировки.
// ПРАВИЛО СТОПКИ - положить карты на верх стопки - порядок
public enum TopPlaceRuleOrder
{
// нижняя карта накладываемой стопки должна быть Апу,
// любой
TwoAce, // быть по порядку восходящей (с 2 до туза)
AceTwo // быть по порядку нисходящей (с туза до 2)
}
Правило последовательности. Данное правило накладывает ограничения на формирование последовательности карт в данной стопке с учетом накладывания в нее сверху другой карты (последовательности карт). Такое ограничение позволяет формировать стопку с определенной последовательностью карт (одной масти или одного цвета масти, с чередованием по масти, цвету масти и т.д.). Также предусмотрена возможность игнорирования последовательности. Код соответствующего нумератора приведен в листинге 3-9.
Листинг 3.9 – Нумератор правила последовательности.
// ПРАВИЛО СТОПКИ - положить карты на верх стопки - последовательность public
enum TopPlaceRuleSequence
{
// нижняя карта накладываемой стопки должна быть
Апу, // не имеет значения
SameSuit, // той же масти
SameColor, // того же цвета
SameAdvantage, // того же достоинства
AlternativeColor, // другого цвета
AlternativeSuit // другой масти
}
Правило нижней карты. Определяет тип карты, которая может быть помещена в начало пустой стопки. Определена, например, для целевых стопок, т.к. эти стопки начинаются со строго определенной (по достоинству) карты. Данное правило представлено нумератором достоинства карты Advantage, предусматривающим также игнорирование этого правила при Advantage = Any.
Поскольку игровой процесс подразумевает постоянное перемещение карт из между стопками разных типов и назначения, то следует базовый класс стопки снабдить методами, контролирующими события проведения пользовательских манипуляций: событие попытки захватить стопку (или ее часть) для перемещения – tryCaptureToMove.
событие «отпускания» стопки, определяющее либо возврат захваченных карт туда, откуда взяли, либо успешное перемещение их в другую стопку releaseMoving.
Захвата карт стопки для их перемещения определяется правилами стопки deckRules.takeFromRule, но это не всегда возможно. Поэтому, прежде чем разрешать захват карт в стопке, необходимо сопоставить содержимое последовательности захватываемых карт правилам стопки. За это сопоставление отвечает метод CanCardsMove. Этот метод должен быть вызван в теле метода tryCaptureToMove для проверки возможности перемещения карт. Вызовы методов захвата и отпускания карт стопок должны осуществляться на уровне отлавливания событий в компонентах пользовательского интерфейса.
Каркас класса генеральной колоды приведен в листинге 3.10.
Листинг 3.10 – Каркас класса генеральной колоды.
public abstract class Deck

```
{
private int count; // кол-во карт private
DeckRules deckRules; // правила private
PlacingMethod orientation; // ориентация private
UIDeck ui; // визуальная компонента private
List<Card> cards; // карты стопки // Попытка
переместить часть стопки public void
tryCaptureToMove(Card topCard)
// Функция определения возможности перемещения карт стопки
// Определяется на основе правил стопки в DeckRules
private bool CanCardsMove(List<Card> cards) //
Завершение перемещения стопки
public void releaseMoving()
}
```

Колода раздачи является основной в игре. Именно она должна быть определена и наполнена в самом начале подготовки к игре и осуществлении раздачи исходного расклада пасьянса. Поведение этой колоды в основном определяется вышеописанными правилами. В дополнение к ним колода раздачи обладает еще уникальными свойствами и методами.

К числу уникальных свойств относятся:

Количество разрешенных переборов, или пересдач (через колоду перебора) - bustCount;

Буфер перемещения карт – специальный буфер, через который будет организована анимация перемещения карт из колоды раздачи в другие колоды - List<SendCardsInfo> bufferToSend;

Правило поведения колоды по действию пользователя (клике мышью) clickBehavior – данное правило определяет, будут ли по клику игрока карты отправлены в колоду перебора, или же в игровые колоды. Соответствующий нумератор приведен в листинге 3.11. Листинг 3.11 – Нумератор правила поведения по клику колоды раздачи.

```
// Поведение колоды раздачи при клике
public enum DDeckClickBehavior
{
AddToBustDeck, // добавить карты в колоду перебора
AddToGameDeck // добавить в игровые колоды
}
```

Класс колоды раздачи имеет уникальные методы:

Инициализации: Initialize – в методе генерируется и раскладывается общее число карт, определенных конкретной разновидностью игры, переопределяются наследуемые правила стопки.

Перетасовывание карт: Shuffle – метод, привносящий элемент случайности в игру.

Методы перебора стопки: ReBust, SendCardsToBustDeck – определены для реализации возможности определенного количества переборов стопки раздачи через стопку перебора.

Методы, отвечающие за анимацию и логику перемещения карт: SendCardsToBustDeck, SendCardsToGameDecks, SendNextCard.

В листинге 3.12 приведен каркас класса DistributionDeck – колоды раздачи.

Листинг 3.12 – Каркас класса колоды раздачи.

```
public class DistributionDeck : Deck
{
// Список карт, которые переносятся из одной стопки в другую public
struct SendCardsInfo
{
public Deck destination;
public bool turn;
}
public List<SendCardsInfo> bufferToSend = new List<SendCardsInfo>();
// правила поведения при действии
private DDeckClickBehavior clickBehavior;
// кол-во разрешенных пересдач
private int bustCount; //
инициализировать
private void Initialize(int count)
// перетасовать
private void Shuffle(byte depth)
// конструктор ---------------------------------
public DistributionDeck(int volume,DeckRules dr, DDeckClickBehavior ddcb, int bc)
// добавление карт в стопку перебора public
void SendCardsToBustDeck()
// ---------------------------------------------
// отправка очередной карты в стопку (для анимации)
public void SendNextCard() // перебор стопки private
void ReBust()
// добавление карт в игровые стопки public
void SendCardsToGameDecks()
// перемещение заданной карты на верх стопки public
void MoveCardOnTop(int imc)
}
```

Колода перебора представляет собой своеобразный буфер в игре. В дополнение к вышеописанным правилам, эта колода имеет еще и уникальные свойства, и методы:

Свойство additionalCount – определяет количество карт, которое будет отправлено из колоды раздачи в колоду (колоды) назначения. В качестве колод назначения может быть колода перебора или игровые стопки – определяется свойством DDeckClickBehavior колоды раздачи.

Метод инициализации: Initialize – реализует переопределение структуры правил поведения DeckRules, унаследованной от базового класса Deck. Листинг 3.13 – Каркас класса колоды перебора.

```
public class BustDeck : Deck
```

```
{
// количество карт, добавляемых из колоды раздачи за 1 раз
private int additionalCount; // инициализировать
private void Initialize(int count)
//    конструктор    --------------------------------
public BustDeck(int volume, DeckRules dr, int ac)
}
```

Целевые колоды являются основными в игре, для которых анализируется состав для определения победного расклада. Для целевых колод жестко определено свойство DeckRules.bottomCard, входящее в структуру правил поведения – это означает, что данная колода должна начинаться с карты, строго соответствующей по достоинству свойству DeckRules.bottomCard. При этом некоторые модификации пасьянса требуют при раздаче расклада помещения указанной карты из колоды раздачи в целевую стопку сразу в начале игры. За это требование отвечает свойство initialCard, определяющее: должна ли в стопке лежать первая карта в начале игры или нет. При этом у разных целевых стопок это значение может отличаться. Например, модификация «Спидайк» требует наличия «верхней» карты только в одной целевой стопке из четырех, а модификация «Тупики» – 4 из 4. Целевая стопка также имеет метод инициализации Initialize, который реализует переопределение структуры правил поведения DeckRules, унаследованной от базового класса Deck. Листинг 3.14 – Каркас класса целевой колоды.

```
public class TaskDeck : Deck
{
// начальная карта должна лежать в колоде сразу при раздаче
private bool initialCard; // инициализировать
private void Initialize(int count)
// конструктор --------------------------------
public TaskDeck(DeckRules dr, bool initialCard)
}
```

Игровые колоды предназначены для выполнения основных манипуляций перекладки карт в игре с целью их упорядочивания в требуемые последовательности. Количество игровых стопок может варьироваться в зависимости от конкретного варианта игры. Также может варьироваться и количество карт (с указанием их положения – «рубашкой» вверх или вниз) в каждой из игровых стопок.

Игровая стопка также имеет метод инициализации Initialize, который реализует переопределение структуры правил поведения DeckRules, унаследованной от базового класса Deck. Листинг 3.15 – Каркас класса игровой колоды. public class GameDeck : Deck

```
{
// инициализировать
private void Initialize(int count)
// конструктор --------------------------------
public GameDeck(DeckRules dr)
}
```

Дополнительная колода является дополнительным карточным буфером в игре. Эта стопка встречается только в модификациях «Сумасшедший брод» и «Сумасшедший брод 2». Правила данной стопки разрешают помещать в нее только королей. Это значит, что структура, инициализирующая правила данной стопки будет выглядеть, как показано в листинге 3.16. В листинге 3.17 приведен исходный код каркаса класса AdditionalDeck.

Дополнительная стопка также имеет метод инициализации Initialize, который реализует переопределение структуры правил поведения DeckRules, унаследованной от базового класса Deck.

Листинг 3.16 – Исходная структура правил дополнительной стопки.

```
takeFromRule = TakeFromRule.OneCard; topPlaceRuleOrder =
TopPlaceRuleOrder.Any;
topPlaceRuleSequence = TopPlaceRuleSequence.SameAdvantage;
topPlaceRuleMultiplicity = TopPlaceRuleMultiplicity.OnlyOne;
bottomCard = Advantage.King;
```

Листинг 3.17 – Каркас класса дополнительной колоды.

```
public class AdditionalDeck : Deck
{
// инициализировать
private void Initialize(int count)
// конструктор --------------------------------
public AdditionalDeck(DeckRules dr)
// конструктор --------------------------------
}
```

Игровой процесс сопровождается постоянным перекладыванием карт из одной стопки в другую, при этом перемещать можно как отдельно карты, так и целые стопки (части стопок). Поэтому в структурную схему объектов целесообразно ввести дополнительный объект – буферную стопку – стопку, которая будет появляться только в моменты проведения игровых манипуляций и будет содержать набор перемещаемых карт. Очевидно, что буферная стопка будет всегда вертикально-ориентированной, то есть свойство orientation = PlacingMethod.VerticalLine.

Буферная стопка, как и все остальные, должна иметь метод инициализации Initialize, который реализует переопределение структуры правил поведения DeckRules, унаследованной от базового класса Deck.

Буферная стопка в любом случае создается как часть некоторой другой стопки, из которой происходит перемещение карт, поэтому для реализации логики перемещений в буферной стопке необходимо хранить ссылку на бывшего владельца находящихся в ней карт – эта информация будет храниться в свойстве source.

Буферная стопка будет свободно перемещаться по игровому полю посредством манипуляций игрока с помощью мыши. При этом, чтобы позиционировать и отображать перемещаемые карты корректно, необходимо учитывать место захвата этой стопки кнопкой мыши еще в тот момент, когда она не была отделена от основной стопки source. Это место определяется свойством (точкой) offsetPoint. Также необходимо наличие дополнительного метода, который будет управлять положением стопки при ее перемещении по игровому полю. За эту операцию будет отвечать метод UpdateLocalPosition. В листинге 3.18 приведен прототип класса буферной стопки. Листинг 3.18 – Прототип класса буферной стопки.

```
public class BufferedDeck : Deck
{
// исходная стопка, из которой выполняется перемещение
Deck source;
// Точка смещения - точка захвата карты в стопке при перемещении Point
offsetPoint;
// инициализировать
private void Initialize(int count)
// конструктор --------------------------------
public BufferedDeck(DeckRules dr)
// конструктор --------------------------------
```

```
// Обновление позиции буферного стека на поле public
void UpdateLocalPosition(Point updatePoint)
}
```

Класс подготовки и управления игровым процессом – Game

Весь приведенный в п. 3.2.1 состав объектов игрового процесса еще не представляет собой законченную модель игры, поскольку приведенные классы стопок и карт не могут реализовать сам процесс игры в динамике. Они позволяют всего лишь задать исходные данные игры: основные правила расположения, состава игровых объектов, правила манипуляции – базу, отличающую возможные разновидности реализации игры.

Очевидна необходимость в наличии некоторого класса, который будет обслуживать весь игровой процесс. Таким классом является класс Game. Основные задачи этого класса:

Располагать полной актуальной информацией об объектах текущей игры.

Класс Game должен быть создателем и владельцем ссылок на все колоды, доступные в игре:

список игровых колод List<GameDeck>

__GD; список целевых колод List<TaskDeck>

__TD; колода раздачи __DD; колода

перебора __BD; дополнительная колода

__AD; буферная колода __MD.

Все эти колоды создаются и параметрируются в данном классе.

Подготовить, создать, проинициализировать и завершить игру.

В рамках данной задачи происходит инициализация базовой конфигурации игры, определение номенклатуры объектов, участвующих в игровом процессе. За процесс инициализации и подготовки отвечает метод Prepare. Метод newPlayOut осуществляет процессы подготовки объектов игры, руководствуясь установленными правилами конкретного варианта игры, прописанными в соответствующем XML-файле, в методе происходят установки правил поведения всех колод и такие частные параметры, как:

TaskDeckCount – количество целевых стопок в игре;

GameDeckCount – количество игровых стопок в игре;

IsBustDeckExist – наличие или отсутствие в игре колоды перебора;

IsAdditionalDeckExist – наличие или отсутствие в игре дополнительной колоды;

int[] gdCards, gdfCards – массивы, указывающие соответственно общее количество карт и количество карт, лежащих «рубашкой» вверх при раздаче карт в игровые стопки – будет считаться, что при наполнении игровой стопки в процессе раздачи расклада пасьянса сначала укладываются карты, перевернутые «рубашкой» вверх, а все остальные карт gdCards[i]-gdfCards[i] – «картинкой» вверх;

TDRandomInitialCard – определяет, будут ли целевые стопки начинаться с карты со случайно выбранным достоинством (по умолчанию – с туза);
TDInitialCardCount – определяет количество целевых стопок игры, в которых первая (нижняя) карта должна быть роздана сразу в начале игры;
selectedGameID – идентификатор выбранной разновидности пасьянса, по которому будет происходить поиск конфигурации в структурном массиве установок вариантов игры при создании новой игры;

Title, Rules – метаданные выбранной разновидности пасьянса, определяемые идентификатором selectedGameID – содержат соответственно наименование и правила выбранной модификации игры.

Обеспечить взаимодействие с игроком через пользовательский интерфейс главного окна.

Взаимодействие игровой модели с пользователем должно осуществляться через главное окно пользовательского интерфейса, являющееся одновременно игровым полем. Для этого классу Game необходимо располагать ссылкой на экземпляр окна игрового поля – MainWindow MW. Все событийные действия игрока (имеются ввиду манипуляции мышью – захват, перемещение, отпускание) отслеживаются механизмами обработки событий соответствующих UI-компонентов. Однако в конечном итоге совершение игроком указанных событий должно передавать управление в класс Game. Для этого предусмотрено поле capturedToMove, которое сообщает игровой модели, что произведен захват некоторой стопки карт для их возможного перемещения, и соответствующие методы:

CaptureDeck – выполняет операции подготовки к перемещению;

CompleteMoving – выполняет операции, завершающие перемещение захваченных карт. Обеспечить

реализацию визуальных эффектов.

Комплекс визуальных эффектов игрового процесса состоит в обеспечении эффекта неуправляемого пользователем перемещения карты в пределах игрового поля из одной колоды в другую. Технология WPF предоставляет широкие возможности простого и мало затратного способа реализации анимационных эффектов посредством управления свойствами зависимостей. Класс Game будет использовать свойство ThicknessAnimation TA для анимации одной карты. Возможности ThicknessAnimation позволяют создавать анимацию перемещения, управляя свойством зависимости MarginProperty любого UI-объекта. Для выполнения анимации необходимо знать саму карту, которая будет перемещаться – MC, а также стопки источника – dFrom и назначения – DTo. Основной метод, реализующий анимацию – MoveCard, в этом методе будут происходить все подготовки и вычисления; по завершении анимации будет выполнен метод – CardAnimationComplete, задача которого поддерживать последовательную цепочку анимаций выбранного количества карт на базе структуры SendCardsInfo. Листинг 3.19 – Прототип класса управления игровым процессом.

```
public static class Game
{
public static MainWindow MW;
public static int selectedGameID = -1; // выбранная игра - идентификатор из меню
public static DistributionDeck __DD; // С Т О П К И public static BustDeck __BD;
public static AdditionalDeck __AD; public static List<GameDeck> __GD; public static
List<TaskDeck> __TD; public static BufferedDeck __MD;
public static bool capturedToMove = false; // флаг начала перемещения
// Захват стопки мышью для перемещения public
static void CaptureDeck(Deck src, Card topCard)
public static void CompleteMoving() // Завершение перемещения
// Анимация
private static ThicknessAnimation TA; // Анимация перемещения карты
public static void MoveCard(Deck dFrom, Deck dTo, bool turn) private
static Deck DFrom, DTo; // стопка источник и цель private static Card
MC; // перемещаемая карта
// Анимция перемещения карты: этап - завершение и обновление private
static void CardAnimationComplete(object sender, EventArgs e)
// Настройки пасьянса общие
public static int TaskDeckCount; // кол-во целевых стопок public static int
GameDeckCount; // кол-во игровых стопок public static bool
IsBustDeckExist = true; // наличие колоды перебора public static bool
IsAdditionalDeckExist = false; // наличие дополн. колоды public static
string Title = ""; // название пасьянса public static string Rules = ""; //
правила пасьянса // распределение карт по игровым стопкам private
static int[] gdCards, gdfCards;
```

```
// начальная карта в целевой колоде - случайная
public static bool TDRandomInitialCard;
// кол-во целевых колод, у которых первая карта уже должна лежать public
static int TDInitialCardCount;
// раздача пасьянса - конструирование игры
public static void Prepare(MainWindow MWnd) // подготовка
public static bool CheckForWin() // проверка на победу public
static void newPlayOut() // инициализация игры
}
```

Представление объектов в интерфейсе игры

Основные объекты игры представлены в интерфейсе как пользовательские компоненты UserControl. Компонент для карты – UICard, для стопки – UIDeck. Любой тип стопки реализуется на базе единого компонента UIDeck.

UserControl, помимо визуальной части, также реализуется классом, предоставляющим свой исходный код для редактирования.

Класс карты UICard предназначен главным образом для отображения карты, а также для отслеживания некоторых событий, посредством которых возможна реализация логики игры. Основной метод этого класса, отвечающий за отображение карты – drawCard. Данный метод использует поля связанного класса Card – масть и достоинство – для прорисовки внешнего вида карты. Изображения лицевой части и «рубашки» карт «подхватываются» из PNG-ресурсов, подключенных к проекту. В UICard также должна происходить обработка пользовательских событий мыши. Прототип класса UICard приведен в листинге 3.20.

Листинг 3.20 – Прототип класса визуальной компоненты карты.

```
public partial class UICard : UserControl
{
Card MyClass; // Ссылка на класс карты
public UICard(bool face, Card MyParent) // конструктор public
void drawCard(bool face) // переворачивание карты
}
```

Класс визуализации стопки отвечает главным образом за процесс компоновки карт в рамках стопки, пересчитывая расстояния перекрытия карт в стопке для их оптимального отображения. При этом учитываются размеры карт, их количество в топке, а также ориентация стопки. Простейший случай – ориентация в одну стопку, когда все карты последовательно прорисовываются одна поверх другой. При вертикальной или горизонтальной ориентации должны мыть вычислены размеры перекрытий соседних карт, чтобы все карты стопки были видны игроку. Все эти действия реализуются методом UpdateView. В UIDeck также должна происходить обработка пользовательских событий мыши.

Прототип класса UIDeck приведен в листинге 3.21.

Листинг 3.21 – Прототип класса визуальной компоненты стопки.

```
public partial class UIDeck : UserControl
{
private Deck myDeck;
public UIDeck(Deck parentDeck, Advantage firstCardRule)
// Перерисовка стопки при изменении количества карт
public void UpdateView()
}
```

Представление модификаций игры

Модификации (разновидности) «Косынки» отличаются между собой предустановленной конфигурацией правил, описывающих наличие и количество объектов, используемых в игре, а также свойства, характеризующие их поведение. Поэтому конфигурация варианта пасьянса должна включать данные по количеству стопок разного назначения, количеству карт в этих стопках и способу их распределения, а также структуру правил DeckRules каждой стопки.

Таким образом, совокупность данных, которые будут описывать конкретный вариант игры, будут представлены (записаны) в виде файла параметризации модели пасьянса.

Файл параметризации модели «Косынки» будет иметь иерархическую структуру. Для описания структур подобного вида широко используются языки разметки, например, XML. Таким образом, каждый файл параметризации пасьянса должен быть представлена в виде XML-файла.

В листинге 3.22 приведен пример описания конфигурации игры для ее разновидности «Красный и черный».

Структура XML файла заключается в описании общих параметров игры и инициализации правил всех стопок, которые должны присутствовать в данной разновидности игры.

Листинг 3.22 – Структура файла параметризации пасьянса в XML-формате.

```
<sol20>
<Title>Красный и черный</Title>
<Rules> </Rules>
<TaskDecks>
<Count>8</Count>
<RandomInitialCard>0</RandomInitialCard>
<InitialCardsCount>0</InitialCardsCount>
<DeckRules>
<TakeFromRule>1</TakeFromRule>
<TopPlaceRuleOrder>1</TopPlaceRuleOrder>
<TopPlaceRuleSequence>1</TopPlaceRuleSequence>
<TopPlaceRuleMultiplicity>1</TopPlaceRuleMultiplicity>
<Advantage>14</Advantage>
</DeckRules>
</TaskDecks>
<GameDecks>
<Count>8</Count>
<Cards>1,1,1,1,1,1,1,1</Cards>
<FlipCards>0,0,0,0,0,0,0,0</FlipCards>
<DeckRules>
<TakeFromRule>6</TakeFromRule>
<TopPlaceRuleOrder>2</TopPlaceRuleOrder>
<TopPlaceRuleSequence>4</TopPlaceRuleSequence>
<TopPlaceRuleMultiplicity>2</TopPlaceRuleMultiplicity>
<Advantage>0</Advantage>
</DeckRules>
```

</GameDecks>
<BustDeck>
<AdditionalCount>1</AdditionalCount>
<DeckRules>
<TakeFromRule>1</TakeFromRule>
<TopPlaceRuleOrder>0</TopPlaceRuleOrder>
<TopPlaceRuleSequence>0</TopPlaceRuleSequence>
<TopPlaceRuleMultiplicity>0</TopPlaceRuleMultiplicity>
<Advantage>0</Advantage>
</DeckRules>
</BustDeck>
<DistributionDeck>
<Cards>104</Cards>
<DDeckClickBehavior>0</DDeckClickBehavior>
<BustCount>0</BustCount>
<DeckRules>
<TakeFromRule>0</TakeFromRule>
<TopPlaceRuleOrder>0</TopPlaceRuleOrder>
<TopPlaceRuleSequence>0</TopPlaceRuleSequence>
<TopPlaceRuleMultiplicity>0</TopPlaceRuleMultiplicity>
<Advantage>0</Advantage>
</DeckRules>
</DistributionDeck>
<AdditionalDeck> <Exists>0</Exists> </AdditionalDeck>
</sol20>

Основные разделы файла параметризации:

Заголовок, или название разновидности – Title.

Правила игры – Rules, приведенные в текстовом виде, чтобы игрок мог ознакомиться с ними. Описание целевых стопок – TaskDecks, включающее:

Количество стопок – Count.

Случайный выбор нижней карты стопки – RandomInitialCard.

Количество стопок, в которых первая карта открыта – InitialCardsCount.

Правила поведения – DeckRules.

Описание игровых стопок – GameDecks, включающее:

Количество стопок – Count.

Количество карт в каждой из игровых стопок – Cards.

Количество карт, лежащих снизу «рубашкой» вверх, в каждой из игровых стопок – FlipCards.

Правила поведения – DeckRules.

Описание стопки перебора – BustDeck, включающее:

Количество карт, которое добавляется из стопки раздачи в другие стопки (перебора или игровые) по клику игрока – AdditionalCount. Правила поведения – DeckRules.

Описание стопки раздачи – DistributionDeck, включающее:

Общее количество карт в игре – Cards.

Поведение при клике игрока – DdeckClickBehavior.

Количество разрешенных переборов стопки перебора – BustCount.

Правила поведения – DeckRules.

Описание дополнительной стопки – AdditionalDeck, включающее:

Флаг, определяющий наличие дополнительной стопки – Exists.

Правила поведения – DeckRules, которые описываются только в случае, если Exists = 1.

Модель игры должна каким-то образом получать данные всех имеющихся файлов параметризации, хранить их, а затем при выборе разновидности пасьянса игроком соответствующим образом настроить свое поведение в соответствии с установленными правилами. Последнее реализуется методом newPlayOut класса управления моделью Game.

Для операций поиска и распознавания моделей используется специальный класс XMLStructure. В этом классе определена структура klondike, в которую должно считываться информация из XML-файла. В листингах 3.23 и 3.24 приведено описание структуры хранения файла параметризации пасьянса и класса поиска и распознавания этих файлов.

Листинг 3.23 – Структура хранения.

```
public struct klondike:IDisposable
{
public bool isNull; // =false - успешно создана public
string title; // название public string rules; // описание
правил public bool additionalExists; // существует доп.
стопка // кол-во целевых и игровых стопок, карт в
игре public int taskDecks, gameDecks, cards;
// первая карта в целевых стопках - случайный выбор
public bool randomInitialCard;
public int initialCardCount; // кол-во целевых стопок с лежащей первой картой
// поведение стопки раздачи при клике мыши public
DDeckClickBehavior ddClickBehaviour;
// кол-во переборов, значение инкремента стопки перебора
public int bustCount, bustAdditionalCount;
// кол-во карт по игровым стопкам (кол-во перевернутых руь. вверх)
public int[] gameDecksCount, gameDecksFlipCount;
// конфигурации правил стопок
public DeckRules taskDeckCfg,gameDeckCfg,bustDeckCfg,addDeckCfg, distrDeckCfg;
}
```

Листинг 3.24 – Класс поиска и распознавания.

```
public static class XMLStructure
{
public static List<klondike> Games = new List<klondike>();// Список вариантов игры
// Составление списка доступных игр (по файлам XML)
public static void ReadGameList()
// Получение инициализирующей структурыиз XML файла static
klondike ParseXMLToGameStructrure(string fileName)
}
```

Класс XMLStructure реализует метод поиска файлов – ReadGameList. Найденные этим методом файлы должны передаваться в метод обработки XML файла и распознавания конфигурации игры – ParseXMLToGameStructrure. Если конфигурация корректна и распознана, то она заносится в список доступных вариантов пасьянса Games.

Список найденных файлов пасьянса реализован в виде коллекционной структуры List<klondike> Games.

Диаграмма пакетов

На рисунке 3.3 приведена модель пакетов, основанная на вышеописанных концептуальных решениях. В состав пакетов модели входят:

Пользовательский интерфейс (USER INTERFACE), объединяющий в себе классы окон (игровое поле, вспомогательные окна) Windows и пользовательские компоненты (карты, стопки) – uiGameComponents.

Основной пакет, содержащий логику модели и ее управляющие структуры – GameModel.

Пакет, выполняющий поиск и инициализацию возможных конфигураций модели под различные варианты пасьянса – XML Parser.

Непосредственно набор файлов, описывающих различные конфигурации игры на языке XML – Games.

Пакет информационного обеспечения модели для записи и хранения пользовательских рекордов игры (Records), включающий в себя файл базы данных, библиотеки провайдера данных и ядро СУБД.

Дополнительные вспомогательные библиотеки из пакета .NET Framework 4.0, необходимые для реализации всех возможностей построенной модели.

Рисунок 3.3 - Диаграмма пакетов

Диаграмма классов

Классы – это структурные блоки, из которых выстраивается любая объектно-ориентированная система. Классы являются описанием совокупности объектов с общими атрибутами, свойствами, методами и схемой отношений.

Диаграмма классов – это набор статических декларативных объектов, связанных между собой в определенную схему. Применяя при моделировании диаграммы классов с специальные (CASE) средства, классовую модель можно преобразовать в файлы исходного кода программы на соответствующем языке программирования. В данной работе первичные файлы программного кода получены таким путем – сгенерированы средой проектирования Visual Paradigm.

Таким образом, диаграмма классов является конечным результатом проектирования модели и началом ее реализации (процесса непосредственной разработки).

На рисунке 3.4 приведена диаграмма классов основных объектов модели, выявленных в п. 3.2.1 – 3.2.3.

Рисунок 3.4 – Диаграмма классов модели.

На рисунке 3.5 приведена диаграмма классов описания параметров основных объектов модели, выявленных в п. 3.2.1 – 3.2.3, в виде структур данных и значений соответствующих нумераторов.

Рисунок 3.5 – Диаграмма классов – структуры и нумераторы.

На рисунке 3.6 приведена диаграмма классов модуля поиска и распознавания файлов параметризации пасьянса, выполненная в соответствии с описанием, приведенном в п. 3.2.4.

Рисунок 3.6 – Диаграмма классов модуля распознавания файлов параметризации модели.

Проектирование модели поведения

Общий алгоритм подготовки игры и игрового процесса

На рисунке 3.7 представлена модель алгоритма подготовки, выбора и начала новой игры. Алгоритм начинается с процесса поиска и распознавания файлов параметризации пасьянса – все найденные варианты игры собираются в коллекцию игр. Собранная коллекция пасьянсов будет доступна в окне выбора игры.

Рисунок 3.7 – Диаграмма активности подготовки игры.

После выбора игрока класс Game создает расклад указанного пасьянса: создает основную колоду раздачи, создает коллекцию карт в ней, затем создает все остальные колоды и устанавливает их правила. В завершении, собирается коллекция карт для пересылки их из стопки раздачи в остальные (игровые и целевые) – далее анимируется перемещение карт из этой коллекции в соответствующие стопки.

После создания расклада, начинается процесс игры – игрок совершает игровые манипуляции, после каждой из которых класс Game проверяет ситуацию на победный расклад (когда все карты помещены в целевые стопки). По завершении игры подсчитываются очки.

Диаграмма состояний класса Game

На рисунке 3.8 показана диаграмма состояний класса Game, иллюстрирующая описание, приведенное в п. 3.3.1. На диаграмме указаны также состояния классов основных стопок игры, приведенные в рамках состояния формирования расклада пасьянса. Рисунок 3.8 – Диаграмма состояний при подготовке игры.

ПРОЕКТИРОВАНИЕ ПРИЛОЖЕНИЯ

Структура приложения

Структура приложения пасьянса представлена на рисунке 4.1 и выполнена на основе разработанной модели. Структурная схема показывает место разработанной объектной модели пасьянса в составе проектируемого приложения.

В составе структуры присутствуют 3 основных компонента: модель

игры, состоящая из двух частей:

статической: представлена совокупностью классов карт и колод различного назначения, реализует схему отношений между ними, инициализирует правила их поведения; в статической части также объявлены и описаны все основные структуры данных и параметров, а также нумераторы, значения которых используются в качестве этих параметров;

динамической: является классом верхнего уровня, предоставляющим механизм реализации базовой логики игры и поддержки основной и дополнительной функциональности (создания и инициализации игры, поддержки эффектов визуализации и анимации объектов, проверки на победу, подсчета количества очков, заработанных игроком, и т.д.), а также взаимодействия с игроком (через интерфейсную часть); пользовательский интерфейс, представленный игровым полем и другими вспомогательными (обслуживающими) окнами, в составе которых:

окно настроек внешнего вида игрового поля;

окно просмотра правил выбранной игры; окно

просмотра рекордов игроков;

набор разновидностей игры, представленный в виде специально выделенной директории XML-файлов, содержащих конфигурации вариантов пасьянса, а также механизм, поддерживающий обнаружение и распознавание этих разновидностей, сбор их в отдельный список (доступный игроку для выбора) и последующего интегрирования их в игровую модель для инициализации разновидности игры, выбранной пользователем. Рисунок 4.1 – Структура приложения на базе модели.

Проектирование компонентного состава приложения

Диаграмма компонентов приложения

Конечное приложение состоит из следующих основных компонентов (в соответствии с рисунком 4.2:

Исполняемый файл программы – Solitair20.exe.

Предустановленный пакет библиотек в виде фреймворка .NET версии 4.0 и выше.

Пакет различных вариантов игры, выполненный в XML-файлах в директории Games.

Компоненты информационного обеспечения:

SQLite Engine – пакет «движка» СУБД:

SQLite.Interop.dll, System.Data.SQLite.Linq.dll – библиотеки БД SQLite;

System.Data.SQLite.dll – библиотека провайдера данных SQLite; records.db

– файл базы данных, содержащий рекорды игроков.

Рисунок 4.2 – Диаграмма компонентов приложения.

Диаграмма компонентов проекта

Детализированная диаграмма компонентов, составляющих проект приложения, приведена, на рисунке 4.3.

Основные пакеты проекта:

USER INTERFACE – включают в себя компоненты пользовательского интерфейса приложения, среди которых: Windows

– окна приложения:

WindowSettings – окно настройки отображения игры.

WindowRecords – окно просмотра рекордов игроков.

MainWindow – основное игровой поле.

WindowRules – окно просмотра правил текущей игры.

WindowColorDialog – вспомогательное окно, предлагающее задание цвета фона игрового поля.

WindowGameSelection – окно выбора разновидности пасьянса для начала раздачи. uiGameComponents

– пользовательские компоненты типа UserControl:

UICard – компонент визуализации карты.

UIDeck – компонент визуализации стопки.

GameModel – основной компонент, реализующий модель игрового процесса, включающий в себя:

Структуры данных модели.

Нумераторы, описывающие параметры структур данных модели.

Класс, реализующий свойства и поведение карты в игровом процессе.

Базовый и производный классы стопок, задействованных в игровом процессе.

Класс управления игровым процессом – подготовка, управление событиями, анимация, логические проверки завершения игры, предустановки отображения.

XML Parser – компонент, реализующий поиск и преобразование XML файлов в конфигурацию вариантов игры, включающий:

klondike – описание структуры, посредством которой происходит считывание и распознавание вариантов игры и последующая их подготовка в игровом поле.

XMLStructure – класс реализации методов поиска, чтения и обработки XML файлов в массив структур klondike, который является источником доступных вариантов игры.

ImageProcessor – статический класс, содержащий методы обработки графической информации, позволяющий преобразовывать графическую информацию из ресурсов проекта в источники, пригодные для применения их в компонентах пользовательского интерфейса.

Auxiliaries – класс, предоставляющий вспомогательные методы.

Рисунок 4.3 – Диаграмма компонентов проекта приложения.

РЕАЛИЗАЦИЯ ПРИЛОЖЕНИЯ

В данной главе описаны основные концепции реализации приложения Solitair20. Здесь описаны принципы построения пользовательского интерфейса приложения, а также приведены алгоритмы выполнения основной игровой логики. Проект интерфейса основного окна

Основное окно приложения спроектировано с помощью XAML-разметки. Данное окно должно предложить пользователю игровую область и некую панель команд управления приложением. Построение главного окна основано на «контейнерном» принципе, при котором используются некоторые объекты-контейнеры, в которые «вкладываются» интерфейсные компоненты. В рамках технологии WPF таким контейнером может быть компонент типа Grid. Его преимущество заключается в том, что данный компонент можно гибко конфигурировать под содержимое в плане размеров, то есть – задавать конкретные размеры (минимальные) с возможностью их автоматического расширения под содержимое – это идеально подходит для реализации отображения стопок, т.к. их размер напрямую зависит от количества карт, находящихся в них (которое постоянно варьируется).

Кроме того, Grid можно разбивать на ячейки, причем свойства размеров и автоподстройки под содержимое для каждой из ячеек можно задавать независимо друг от друга аналогично, как и для всего Grid. На рисунке 5.1 приведен макет главного окна. Цифрами условно показаны основные контейнеры объектов (строка заголовка Grid Name="Header" не учтена):

Панель команд Grid Name="grdManiMenu" (меню игры) – назначение команд приведено в таблице 5.1.

Контейнер хранения целевых стопок Grid Name="grdTaskDecks".

Контейнер хранения стопки перебора Grid Name="grdBustDeck".

Контейнер хранения стопки раздачи Grid Name="grdGeneralDeck".

Контейнер хранения игровых стопок Grid Name="grdGameDecks".

Контейнер хранения дополнительной стопки Grid Name="grdAdditionalDeck".

Общий контейнер, который хранит в себе контейнеры 0-5, в нем также хранится и перемещается буферная стопка Grid Name="grdGlobal".

Рисунок 5.1 – Проект главного окна приложения.

Контейнеры 1-3 объединены в один контейнер более высокого уровня (по иерархии) – Grid Name="grdTopDecks", который помещен непосредственно в контейнер 0.

Контейнеры хранения целевых и игровых стопок впоследствии (на этапе подготовки игрового поля под выбранную разновидность пасьянса) будут разделены на горизонтальные ячейки, каждая из которых будет хранить отдельный экземпляр визуальной компоненты соответствующей стопки.

В листинге 5.1 приведен код XAML-разметки окна в соответствии с описанной схемой разбивки на контейнеры. Листинг

5.1 – XAML-разметка главного окна приложения.

```
<Grid Name="grdGlobal">
<Grid Name="grdField">
<Grid.RowDefinitions>
<RowDefinition Height="30"/>
<RowDefinition Height="50"/>
<RowDefinition Height="200"/>
<RowDefinition Height="*"/>
<RowDefinition Height="200"/>
</Grid.RowDefinitions>
<!-- ======================== ЗАГОЛОВОК ОКНА ====================== -->
<Grid Name="Header" Grid.Row="0" Background="#FF007ACC"> </Grid>
<!-- ======================== ЗАГОЛОВОК ОКНА ====================== -->
<!-- ========================== ГЛАВНОЕ МЕНЮ ====================== -->
```

```
<Grid Grid.Row="1" Name="grdManiMenu" Background="#FFF0F0F0"> </Grid>
<!-- ========================== ГЛАВНОЕ МЕНЮ ====================== -->
<!-- ========================== ВЕРХНИЕ СТОПКИ ====================== -->
<Grid Name="grdTopDecks" Grid.Row="2" Margin="10,10,10,10">
<Grid.ColumnDefinitions>
<ColumnDefinition Width="450*"/>
<ColumnDefinition Width="200"/>
<ColumnDefinition Width="110"/>
</Grid.ColumnDefinitions>
<!-- _________________ ЦЕЛЕВЫЕ СТОПКИ _________________ -->
<Grid Grid.Column="0" Name="grdTaskDecks"/>
<!-- _________________ СТОПКА ПЕРЕБОРА _________________ -->
<Grid Grid.Column="1" Name="grdBustDeck"/>
<!-- _________________ СТОПКА РАЗДАЧИ _________________ -->
<Grid Grid.Column="2" Name="grdGeneralDeck"/>
</Grid>
<!-- ========================== ВЕРХНИЕ СТОПКИ ====================== -->
<!-- ========================== ИГРОВЫЕ СТОПКИ ====================== -->
<Grid Grid.Row="3" Name="grdGameDecks" VerticalAlignment="Top"/>
<!-- ========================== ИГРОВЫЕ СТОПКИ ====================== -->
<!-- ==================== ДОПОЛНИТЕЛЬНАЯ СТОПКА ==================== -->
<Grid Grid.Row="4" Name="grdAdditionalDeck" HorizontalAlignment="Right" Margin="0,5,20,5" />
<!-- ==================== ДОПОЛНИТЕЛЬНАЯ СТОПКА ==================== -->
</Grid>
</Grid>
```

Таблица 5.1 – Назначение игрового меню.

Команда

Назначение

Раздать новый расклад текущего варианта пасьянса

Выбрать другой вариант пасьянса и создать его расклад

Просмотреть таблицу рекордов игроков

Получить доступ к окну настроек отображения игры

Получить справочную информацию о приложении

Просмотреть правила текущего варианта пасьянса

Завершить работу приложения

Проект интерфейса вспомогательных окон

Окно настроек игры

Макет окна настройки отображения игрового поля и объектов приведен на рисунке 5.2. В данном окне можно настроить «рубашку» карт, цвет градиентного перехода игрового поля и скорость анимации карт при раздаче. Для задания цвета фона игрового поля добавлено окно, макет которого приведен на рисунке 5.3.

Рисунок 5.2 – Макет окна настройки внешнего вида игры.

Рисунок 5.3 – Макет окна определения цвета.

Окно рекордов

Макет окна рекордов приведен на рисунке 5.4. Окно приводит результат текущей игры и предлагает пользователю ввести имя и зафиксировать свой результат в таблице рекордов. Результат будет занесен в таблицу рекордов и базу данных в случае, если в данной разновидности он лучший среди уже имеющихся.

Результат игры рассчитывается как количество секунд, потраченных на игру, при этом чем быстрее игрок разложил пасьянс, тем лучше результат. Таким образом, наилучшим считается наименьшее количество набранных игровых очков.

Таблица результатов может быть сброшена. Для этого предусмотрена соответствующая кнопка «Сброс».

Рисунок 5.4 – Макет окна просмотра рекордов игроков.

На рисунке 5.5 приведена структура записи одного рекорда, которая содержит: Наименование

разновидности пасьянса;

Имя игрока, введенное при фиксации результата;

Дата создания записи;

Количество набранных очков.

Рисунок 5.5 – Содержимое рекорда.

Окно выбора игры

Окно выбора разновидности пасьянса предоставляет пользователь возможность выбрать наиболее интересную для него разновидность игры. Макет этого окна приведен на рисунке 5.6.

Рисунок 5.6 – Макет окна выбора игры.

По умолчанию макет окна пустой и не содержит никаких доступных вариантов игры. Все возможные варианты пасьянса собираются на этапе инициализации приложении при его запуске посредством механизмов. В результате операций, выполненных этими механизмами, в окне выбора игры формируется список компонентов, инициирующий запуск расклада пасьянса указанной модификации. Макет такого компонента приведен на рисунке 5.7.

Рисунок 5.7 – Компонент выбора разновидности игры.

Окно описания правил игры

Окно описания правил игры доступно только во время самого игрового процесса, когда какой-либо вариант пасьянса был уже выбран и запущен. Описание правил игры приводится в компоненте типа WebBrowser в виде форматированного HTML-кода. Макет окна доступа к просмотру правил выбранной игры приведен на рисунке 5.8. Рисунок 5.8 – Макет окна просмотра правил игры.

Проект интерфейсных компонентов объектов игры

Компонент карты

Компонент карты реализован на базе элемента Grid с заданными размерами (рисунок 5.10). Вложенный в Grid элемент Image призван отображать изображение карты, подключаемое из ресурсов проекта в двух случаях:

при инициализации карты в конструкторе; в реализации

метода filpCard, «переворачивающего» карту.

Поддержка маршрутизируемых событий делает возможным формирование процедур игровой логики, которые связаны с такими событиями, как:

нажатие левой кнопки мыши – инициируют в управляющем классе Game процедуры захвата колоды (начиная с текущей карты); нажатие правой

кнопки мыши – обеспечивают перемещение текущей карты на верх стопки, обеспечивая предпросмотр текущей карты, если она

«скрыта за другими» (см. рисунок 5.11, а);

отпускание правой кнопки мыши – возвращает карту на свое место после процедуры предпросмотра (см. рисунок 5.11, б); процедура предпросмотра задается свойством расположения интерфейсного объекта – Panel.SetZIndex).

В листинге 5.2 приведен программный код реализации этих событий.

Рисунок 5.10 – Макет компонента карты.

Рисунок 5.11 – Визуализация предпросмотра карты.

Листинг 5.2 – Реализация реакции на события мыши. //

ЛЕВАЯ - НАЖАТЬ: попытка перетаскивания

```
private void UserControl_MouseLeftButtonDown(object sender, System.Windows.Input.MouseButtonEventArgs e) {
MyClass.MyDeck.tryCaptureToMove(MyClass); }
// ПРАВАЯ - НАЖАТЬ: просмотр карты
private void UserControl_MouseRightButtonDown(object sender, System.Windows.Input.MouseButtonEventArgs e) {
// Если зажата правая кнопка мыши, то показать выбранную карту поверх всех Panel.SetZIndex(this, 1);
}// Обработка отпускания кнопки мыши
// ПРАВАЯ - ОТПУСТИТЬ: завершение просмотра карты
private void UserControl_MouseRightButtonUp(object sender, System.Windows.Input.MouseButtonEventArgs e)
{// Спрятать карту на своем месте в стопке (если отображена поверх всех) if
(Panel.GetZIndex(this) != 0) { Panel.SetZIndex(this, 0);
this.MyClass.MyDeck.Ui.UpdateView();
}
}
```

Компонент стопки

XAML-конструкция компонента стопки такая же, как и у компонента карты (рисунок 5.12). Это обусловлено тем, что по умолчанию стопка считается пустой и имеет размер для помещения одной карты. Элемент Image стопки служит для задания фона стопки – фон задается только для стопок, для которых жестко определено достоинство карты, которая должна лежать первой в стопке. Рисунок 5.12 – Макет компонента стопки.

Реализация алгоритмов

Подготовка и составление списка модификаций

Подготовка и составление списка модификаций пасьянса реализуется в модуле пакета XML Parser. Алгоритм работы этого модуля запускается в самом начале при старте приложения и состоит в выполнении следующих действий:

Поиск доступных XML файлов в текущей директории – обеспечивается методом ReadGameList.

Для каждого из найденных XML файлов производится парсинг его структуры методом ParseXMLToGameStructrure.

Каждая успешно распознанный файл создает новый вариант игры, представленный компонентом выбора. Каждый такой компонент имеет идентификатор файла, который передается классу Game для формирования и раздачи соответствующего варианта пасьянса. Этот идентификатор представлен полем Game.selectedGameID. Раздача пасьянса

Раздача расклада пасьянса инициируется методом newPlayOut класса Game. В раздаче задействованы методы инициализации основной стопки раздачи. Раздача первоначального расклада игры состоит из следующих шагов:

Инициализируется класс стопки раздачи DistributionDeck.

В процессе инициализации генерируются все карты игры – карты в стопку раздачи генерируются последовательно по порядку от двойки до туза, чередуя масти.

Вызывается метод Shuffle (byte depth) стопки раздачи, который имитирует перетасовку колоды.

После перетасовки колоды раздачи метод newPlayOut класса Game продолжает раздачу пасьянса. При этом в колоде раздачи формируется структура DistributionDeck.SendCardsInfo, которая содержит карты, подлежащие анимации перемещения. Эта структура наполняется в соответствии с правилами, указанными в файле параметризации.

У колоды раздачи вызывается метод __DD.SendNextCard(), начинающий процесс анимации раздачи пасьянса.

Фиксируется время начала игры (для подсчета очков игрока).

По завершении анимации последней розданной карты пасьянс считается готовым для игры.

Инициализация классов стопок

Инициализация классов стопок происходит на начальном этапе создания расклада пасьянса. Для всех типов стопок задаются инициализирующие параметры: количество карт, правила поведения, создается структура для хранения коллекции карт.

Для стопки раздачи на этапе инициализации создается полная коллекция карт игры, которая впоследствии перетасовывется.

Анимация перемещения карт

Анимация перемещения карт используется для раздачи карт на игровое поле в начале игры или по запросу пользователя. Направление перемещения карт при этом: из стопки в стопки, которые предписаны правилами: это могут быть целевые стопки, игровые стопки или стопка перебора.

Поскольку анимация карт происходит всегда из стопки раздачи, то она соодержит основные методы для реализации этой анимации.

Эффект анимации карты базируется на специальных средствах, предоставляемых технологией WPF, из классов пространства имен System.Windows.Media.Animation. Все анимации в WPF основаны не на перерисовке элемента, а на изменении его свойств через интервалы времени. Конкретно для задания положения элемента в WPF используется свойство Margin класса Thickness. Это свойство представляет собой структуру, поля которой указывают расстояния элемента со всех четырех его сторон относительно родительского контейнера. Соответственно, синхронно изменяя эти свойства, можно добиваться эффекта перемещения. Для создания анимаций перемещений в WPF выделен специальный класс ThicknessAnimation. В Solitair20 анимации карт последовательны – то есть следующая карта начинает анимироваться после того, как закончит предыдущая. Для этого необходимо:

Наличие единого экземпляра класса static ThicknessAnimation TA для осуществления всех анимаций.

Привязать к TA событие, возникающее при завершении анимации, чтобы подготовить и начать анимацию следующей карты: TA.Completed += CardAnimationComplete. На этод метод возложены функции по совершению логических перестановок, именно: переложить перемещаемую карту из коллекции стопки-источника в коллекуию стопки назначения, обновить счетчики, «перекинуть» визуальную компоненту карты UICard из UIDeck стопкиисточника в UIDeck стопки назначения.

Иметь очередь карт для организации последовательных анимациий. Такая очередь реализуется коллекцией структур List<SendCardsInfo> bufferToSend, определенной в классе стопки раздачи. Элемент этой структуры содержит информацию о стопке назначения и флаг, который заставит или не заставит эту карту «перевернуться». Например, при попадании карты в стопку перебора из колоды раздачи карта предварительно поворачивается «рубашкой» вниз.

Иметь методы, которые будут подготавливать и запускать анимацию. Такие методы содержатся в классе DistributionDeck:

SendCardsToBustDeck – метод подготовки анимации карт в стопку перебора, также может вызвать метод ReBust – метод, который делает перебор стопки перебора (процесс возврата карт из стопки перебора в колоду раздачи).

SendCardsToGameDecks – метод подготовки анимации карт в игровую стопку с «переворачиванием» перемещаемой карты.

SendNextCard – метод непосредственного запуска анимации (с первой карты, находящейся в списке bufferToSend).

Анимация перемещения карты в целевую стопку не требует дополнительной логики, поэтому для нее не предусмотрено соответствующего метода в DistributionDeck, для этого в SendCardsInfo просто задается соответствующий элемент с указанием стопки назначения.

Иметь универсальный метод обработки каждой конкретной анимации. Такой метод – MoveCard реализуется классом Game. Основное назначение данного метода состоит в том, чтобы, зная стопок источника и назначения анимации, подготовить начальное и конечное значения свойства Margin и запустить анимацию на выполнение.

Манипуляции игрока

Манипуляции игрока состоят в перетаскивании карт из одной стопки в другую. Для оргнизации этого действия была выделена особая буферная стопка – BufferedDeck, в которую перемещаются перекладываемые карты для временного хранения. При перемещении карт реализуется основная логика, предписываемая файлом параметризации модели игры:

Захват стопки мышью – вызывается метод класса Deck генеральной колоды – tryCaptureToMove, который «ищет» индекс первой «захваченной» карты в коллекции карт колоды.

Перед тем, как вызвать метод захватывания карт, проводится проверка возможности захвата выбранных карт – для этого метод CanCardsMove сопоставляет захватываемую стопку карт с правилом TakeFromRule стопки, из которой пытается осуществить захват. В случае, если захват разрешен, осуществляется переход к следующему шагу.

Метод tryCaptureToMove вызывает метод CaptureDeck класса Game. Данный метод формирует буферную колоду из захваченных карт и «запоминает», откуда эти карты были захвачены (чтобы при необходимости можно было вернуть их обратно).

Игрок перемещает карты по игровому полю – отслеживается событие перемещения мыши MouseMove. Перемещение реализуется визуально изменением все того же свойства Margin, только его изменение теперь происходите не автоматически, как при расчете анимации, а вычисляется постоянно в методе UpdateLocalPosition буферной колоды BufferedDeck каждый раз, когда вызывается реакция на событие MouseMove.

Метод releaseMoving класса Deck – вызывается при замеченном событии «отпускания» кнопки мыши, которое сигнализирует о том, что перемещение завершено. При этом возможны ситуации:

Во время «отпускания» мыши буферная стопка находилась над пустой областью игрового поля – в этом случае метод CompleteMoving класса Game возвращает карты туда, откуда были взяты.

Во время «отпускания» мыши буферная стопка находилась над какой-либо другой стопкой – в этом случае метод CompleteMoving класса Game помещает карты в стопку назначения (над которой находилась мышь). Перед эти перемещаемая стопка и стопка назначения сопоставляются на совместимость по правилам DeckRules.TopPlaceRuleSequence, DeckRules.TopPlaceRuleMultiplicity, DeckRules.TopPlaceRuleOrder, DeckRules.TopPlaceRuleMultiplicity.

Метод CompleteMoving после каждой манипуляции вызывает метод CheckForWin, в котором производится проверка на завершение игры – игра считается завершенной, если сумма всех карт в целевых стопках оказывается равной значению, соответствующему количеству карт, которые изначально генерируются в стопке раздачи. ЗАКЛЮЧЕНИЕ

В данной работе были рассмотрены принципы объектно-ориентированного проектирования приложений на примере приложения пасьянс «Косынка». Был произведен анализ данного пасьянса и его модификаций с последующей структурной декомпозицией на объекты и компоненты. В результате проведенного анализа была построена объектная модель пасьянсов типа «Косынка», охватывающая все его варианты. Построенная модель пасьянса позволила разработать Windows-приложение, которое характеризуется следующими результатами:

Современный стильный интерфейс;

Сохранение стандартных возможностей игры (подсчет очков, настройка внешнего вида);

Использование визуальных анимационных эффектов;

Единая объектная модель, позволяющая реализовать множество различных вариантов игры без ее изменения и без вмешательства в программный код приложения;

Автоматическое определение вариантов игры из соответствующей директории;

Ориентирование на конкретный вариант игры посредством считывания из структуры XML-файлов и настройки свойств и поведения объектов модели.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

Албахари Джозеф– C#. Справочник. Полное описание языка. Пер. с англ. - М: ООО «И.Д. Вильямс», 6-е изд., 2016, 1040 с.

Андерсон К. – Основы Windows Presentation Foundation. Пер. с англ. А. Силикина – М.: ДМК Пресс, 2008, 432 с.

Бадд Т. Объектно-ориентированное программирование в действии/ Т. Бадд: пер. с англ. А. Бердникова СПб.: Питер, 1997, 460 с.

Медведев В.И. Особенности объектно-ориентированного программирования на C++/CLI, C# и Java. 2-е изд., испр. и доп. – Казань: РИЦ «Школа», 2010 – 444 с.

Петцольд Ч. – Microsoft Windows Presentation Foundation. Базовый курс, пер. с англ. Е. Матвеев: – М.: Издательство «Русская Редакция»; СПб.: Питер, 2008, 944 с.

Пискунов А.Г. Неполное Руководство по SQLite для пользователей Windows, 2015, Пер.: А.Г.Пискунов, иа.agpl.SQLite.AllOw 0.70 – 119 с.

Andrew Troelsen – Pro C# 5.0 and the .NET 4.5 Framework: Apress, 6th edition: NY, 2012, 1534 p.

Jay A. Kreibich. Using SQLite. – O'Reilly Media, Inc., First Edition – 2010, 528 p.

Svetlin Nakov & Co. – Fundamentals of computer programming with C#. The Bulgarian C# book: – Sofia, 2013, 1122 p.

Visual Paradigm for UML 13. User's guide: Software documentation – user's manual - 1485 p.

UML 2.0. Superstructure specification. – Final adopted specification – ptc/03-08-02; 2003, 640 p.

Введение в C# - [Электронный ресурс]: https://metanit.com/sharp/ (дата обращения: 29.12.2016)

Пасьянс – Википедия. [Электронный ресурс]: https://ru.wikipedia.org/wiki/Пасьянс (дата обращения: 29.12.2016)

SQLite – Википедия. [Электронный ресурс]: https://ru.wikipedia.org/wiki/SQLite (дата обращения: 29.12.2016)

Приложение А – Связи реализации вариантов использования Solitair20 компонентами (окнами) интерфейса приложения Рисунок

А.1 – Реализация прецедента «Выбрать игру».

Сценарий прецедента:

Игрок выбирает команду меню «выбрать игру».

Формируется список доступных модификаций пасьянса.

Игрок выбирает подходящий ему вариант игры.

Рисунок А.2 – Реализация прецедента «Играть».

Сценарий прецедента:

Игрок выбрал вариант игры (прецедент А.1).

Система сформировала и раздала исходный расклад пасьянса.

Игрок осуществляет игровые манипуляции.

Рисунок А.3 – Реализация прецедента «Настроить вид».

Сценарий прецедента:

Игрок выбирает команду меню «настройки отображения».

Система отображает окно настроек отображения игрового поля.

Игрок выбирает подходящий настройки: вид «рубашки», цветовую схему, скорость анимации.

При подтверждении изменений программа применяет выбранные установки.

Рисунок А.4 – Реализация прецедента «Просмотреть правила» Сценарий прецедента:

Игрок осуществляет игровые манипуляции (прецедент А.2). Игрок выбирает команду меню «посмотреть правила игры».

Система показывает окно с описанием правил текущей игры.

Рисунок А.5 – Реализация прецедентов «Просмотреть рекорды» и «Очистить таблицу рекордов».

Сценарий прецедента – вариант 1:

Игрок выбирает команду меню «просмотр таблицы рекордов».

Формируется и отображается таблица рекордов из БД.

Точка ветвления 1.

Сценарий прецедента – вариант 2:

Игрок осуществляет игровые манипуляции (прецедент А.2), которые привели к победному раскладу.

Формируется и отображается таблица рекордов из – приводится результат, достигнутый игроком. Игрок

вводит свое имя и нажимает кнопку «Зафиксировать».

Если достигнутый игроком результат лучше имеющегося в таблице (по данной разновидности пасьянса), то результат перезаписывается с указанием имени игрока и даты.

Точка ветвления 1.

Точка ветвления 1:

Игрок нажимает кнопку «ОК» - окно рекордов закрывается.

Игрок нажимает кнопку «Сброс» - рекорды стираются из БД.