Содержание

Введение

Проектирование объектно-ориентированных программ – непростая задача, а если их нужно использовать повторно, то все становится еще сложнее. Необходимо подобрать подходящие объекты, отнести их к различным классам, соблюдая разумную степень детализации, определить интерфейсы классов и иерархию наследования и установить существенные отношения между классами. Дизайн должен, с одной стороны, соответствовать решаемой задаче, с другой – быть общим, чтобы удалось учесть все требования, которые могут возникнуть в будущем. Хотелось бы также избежать вовсе или, по крайней мере, свести к минимуму необходимость перепроектирования. Поднаторевшие в объектно-ориентированном проектировании разработчики скажут, что обеспечить «правильный», то есть в достаточной мере гибкий и пригодный для повторного использования дизайн, с первого раза очень трудно, если вообще возможно. Прежде чем считать цель достигнутой, они обычно пытаются опробовать найденное решение на нескольких задачах, и каждый раз модифицируют его.

И все же опытным проектировщикам удается создать хороший дизайн системы. В то же время новички испытывают шок от количества возможных вариантов и нередко возвращаются к привычным не объектно-ориентированным методикам. Проходит немало времени перед тем, как становится понятно, что же такое удачный объектно-ориентированный дизайн. Опытные проектировщики, очевидно, знают какие-то тонкости, ускользающие от новичков. Так что же это?

Прежде всего, опытному разработчику понятно, что не нужно решать каждую новую задачу с нуля. Вместо этого он старается повторно воспользоваться теми решениями, которые оказались удачными в прошлом. Отыскав хорошее решение один раз, он будет прибегать к нему снова и снова. Именно благодаря накопленному опыту проектировщик и становится экспертом в своей области. Во многих объектно-ориентированных системах вы встретите повторяющиеся паттерны, состоящие из классов и взаимодействующих объектов. С их помощью решаются конкретные задачи проектирования, в результате чего объектно-ориентированный дизайн становится более гибким, элегантным, и им можно воспользоваться повторно. Проектировщик, знакомый с паттернами, может сразу же применять их к решению новой задачи.

Паттерны проектирования упрощают повторное использование удачных проектных и архитектурных решений. Представление прошедших проверку временем методик в виде паттернов проектирования облегчает доступ к ним со стороны разработчиков новых систем. С помощью паттернов можно улучшить качество сопровождения существующих систем, позволяя явно описать взаимодействия классов и объектов, а также причины, по

которым система была построена так, а не иначе. Проще говоря, паттерны проектирования дают разработчику возможность быстрее найти «правильный» путь.

Целью данной работы является описание наиболее простых и часто используемых шаблонов проектирования, а также выполнение технического задания с использованием некоторых из них.

1 Назначение шаблонов проектирования

Концепцию шаблонов впервые описал Кристофер Александер в книге «Язык шаблонов. Города. Здания. Строительство». Идея показалась привлекательной авторам Эриху Гамму, Ричарду Хелму, Ральфу Джонсону и Джону Влиссидесу, их принято называть «бандой четырёх» (Gang of Four). В 1995 году они написали книгу «Design Patterns: Elements of Reusable Object-Oriented Software», в которой применили концепцию типовых шаблонов в программировании. В книгу вошли 23 шаблона, решающие различные проблемы объектно-ориентированного дизайна.

Шаблоны проектирования можно разделить на три группы: порождающие шаблоны проектирования (Creational Patterns), структурные шаблоны проектирования классов/объектов (Structural Patterns) и шаблоны проектирования поведения классов/объектов (Behavioral Patterns).

Ниже, в этой главе будут рассмотрены некоторые представители всех трех групп.

1.1 Поведенческие шаблоны проектирования

Шаблоны поведения связаны с алгоритмами и распределением обязанностей между объектами. Речь в них идет не только о самих объектах и классах, но и о типичных способах взаимодействия. Шаблоны поведения характеризуют сложный поток управления, который трудно проследить во время выполнения программы. Внимание акцентировано не на потоке управления как таковом, а на связях между объектами.

В шаблонах поведения уровня класса используется наследование – чтобы распределить поведение между разными классами. В этой главе описано два таких шаблона. Из них более простым и широко распространенным является шаблонный метод, который представляет собой абстрактное определение алгоритма. Алгоритм здесь определяется пошагово. На каждом шаге вызывается либо примитивная, либо абстрактная операция. Алгоритм «обрастает мясом» за счет подклассов, где определены абстрактные операции. Другой шаблон поведения уровня класса – интерпретатор, который представляет грамматику языка в виде иерархии классов и реализует интерпретатор как последовательность операций над экземплярами этих классов.

В шаблонах поведения уровня объектов используется не наследование, а композиция. Некоторые из них описывают, как с помощью кооперации множество равноправных объектов справляется с задачей, которая ни одному из них не под-силу. Важно здесь то, как объекты получают информацию о существовании друг друга. Объекты коллеги могут хранить ссылки друг на друга, но это увеличит степень связанности системы. При максимальной степени связанности каждому объекту пришлось бы иметь информацию обо

всех остальных. Эту проблему решает шаблон посредник. Посредник, находящийся между объектами коллегами, обеспечивает косвенность ссылок, необходимую для разрывания лишних связей.

1.1.1 Шаблон «Стратегия» (Strategy)

Назначение шаблона: определяет семейство алгоритмов, инкапсулирует каждый из них и делает их взаимозаменяемыми. Стратегия позволяет изменять алгоритмы независимо от клиентов, которые ими пользуются. Другими словами, стратегия инкапсулирует определенное поведение с возможностью его подмены.

Шаблон «Стратегия» является настолько распространенным и общепринятым, что многие его используют постоянно, даже не задумываясь о том, что это хитроумный шаблон проектирования, расписанный когда-то «бандой четырех».

Каждый второй раз, когда мы пользуемся наследованием, мы используем стратегию; каждый раз, когда абстрагируемся от некоторого процесса, поведения или алгоритма за базовым классом или интерфейсом, мы используем стратегию. Сортировка, анализ данных, валидация, разбор данных, сериализация, кодирование/декодирование, получение конфигурации — все эти концепции могут и должны быть выражены в виде стратегий или политик (policy).

Стратегия является фундаментальным шаблоном, поскольку она проявляется в большинстве других классических шаблонов проектирования, которые поддерживают специализацию за счет наследования. Абстрактная фабрика — это стратегия создания семейства объектов; фабричный метод — стратегия создания одного объекта; строитель — стратегия построения объекта; итератор — стратегия перебора элементов и т. д

Мотивация использования шаблона «Стратегия»: выделение поведения или алгоритма с возможностью его замены во время исполнения.

Участники:

Strategy — определяет интерфейс алгоритма;

Context — является клиентом стратегии;

ConcreteStrategyA, ConcreteStrategyB — являются конкретными реализациями стратегии.

Шаблон «Стратегия» не определяет, как стратегия получит данные, необходимые для выполнения своей работы. Они могут передаваться в аргументах метода AlgorithmInterface, или стратегия может получать ссылку на сам контекст и получать требуемые данные самостоятельно.

Шаблон «Стратегия» не определяет, каким образом контекст получает экземпляр стратегии. Контекст может получать ее в аргументах конструктора, через метод, свойство или у третьей стороны.

Применимость стратегии полностью определяется ее назначением: шаблон «Стратегия» нужно использовать для моделирования семейства

алгоритмов и операций, когда есть необходимость замены одного поведения другим во время исполнения. Не следует использовать стратегию на всякий случай. Наследование добавляет гибкости, однако увеличивает сложность. Любой класс уже отделяет своих клиентов от деталей реализации и позволяет изменять эти детали, не затрагивая клиентов. Наличие полиморфизма усложняет чтение кода.

Гибкость не бывает бесплатной, поэтому выделять стратегии стоит тогда, когда действительно нужна замена поведения во время исполнения.

1.1.2 Шаблон «Шаблонный метод» (Template Method)

Шаблонный метод определяет основу алгоритма и позволяет подклассам переопределять некоторые шаги алгоритма, не изменяя его структуры в целом. Шаблонный метод —это каркас, в который наследники могут подставить реализации недостающих элементов.

На заре становления объектно-ориентированного программирования (ООП) наследование считалось ключевым механизмом для расширения и повторного использования кода. Однако со временем многим разработчикам стало очевидно, что наследование — не такой уж простой инструмент, использование которого создает сильную связанность (tight coupling) между базовым классом и его наследником. Эта связанность приводит к сложности понимания иерархии классов, а отсутствие формализации отношений между базовым классом и наследниками не позволяет четко понять, как именно разделены обязанности между классами Base(Базовый) и Derived(Насдедник), что можно делать наследнику, а что — нет.

Наследование подтипов подразумевает возможность подмены объектов базового класса объектами классов наследников. Такое наследование моделирует отношение «ЯВЛЯЕТСЯ», и поведение наследника должно соответствовать принципу наименьшего удивления: все, что корректно работало с базовыми классами, должно работать и с наследниками.

Более формальные отношения между родителями и потомками описываются с помощью предусловий и постусловий. Другой способ заключается в использовании шаблона «Шаблонный метод», который позволяет более четко определить «контракт» между базовым классом и потомками.

Шаблонный метод — это один из классических шаблонов, который и по сей день используется в каноническом виде во многих приложениях.

Шаблонный метод может применяться классом-наследником повторно при реализации переменного шага алгоритма, объявленного в базовом классе.

Практически всегда, возникает необходимость повторного использования кода с помощью наследования, стоит подумать, как можно выразить отношения между базовым классом и его наследником максимально четко. При этом нужно помнить о своих клиентах — как о

внешних классах, так и о наследниках: насколько просто создать наследника класса, какие методы нужно переопределить, что в них можно делать, а что — нельзя? Когда и от каких классов иерархии наследования нужно наследовать? Насколько легко добавить еще одного наследника, не вдаваясь в детали реализации базового класса?

Формализация отношений между базовым классом и наследником с помощью контрактов и шаблонного метода сделает жизнь разработчиков наследников проще и понятнее. Шаблонный метод задает каркас, который четко говорит пользователю, что он может сделать и в каком контексте.

1.1.3. Шаблон «Посетитель» (Visitor)

Шаблон описывает операцию, выполняемую с каждым объектом из некоторой иерархии классов. Шаблон «Посетитель» позволяет определить новую операцию, не изменяя классов этих объектов.

Объектно-ориентированное программирование предполагает единство данных и операций. Обычно классы представляют некоторые операции, скрывая структуры данных, над которыми эти операции производятся. Но не всегда удобно или возможно смешивать их в одном месте. Структуры данных некоторых предметных областей могут быть довольно сложными, а операции над ними настолько разнообразными, что совмещать эти два мира нет никакого смысла. Например, в мире финансовых инструментов гораздо логичнее отделить данные ценных бумаг от выполняемых над ними операций. Деревья выражений — это еще один классический пример того, где происходит такое разделение. Но даже в простой задаче, такой как экспорт лог-файлов, можно найти места, где такой подход будет более разумным.

Использовать посетитель нужно лишь тогда, когда появляется необходимость разделить иерархию типов и набор выполняемых операций. Шаблон «Посетитель» позволит легко разбирать составную иерархическую структуру и обрабатывать разные типы узлов особым образом.

Использовать шаблон «Посетитель» нужно тогда, когда набор типов иерархии стабилен, а набор операций — нет.

Классический вариант шаблона лучше всего подходит для больших составных иерархий и когда заранее не известно, какие типы будут посещаться чаще других.

Функциональный вариант посетителя всегда можно построить на основе классической реализации, когда станет известно, что многим клиентам нужно посещать лишь небольшое число типов иерархии. Вариант на основе делегатов в самостоятельном виде подходит лишь для небольшой иерархии типов. По сути, он является простой реализацией сопоставления с образцом и подходит для работы с простыми иерархиями типов, которые моделируют размеченные объединения. Также этот вариант может быть добавлен с помощью методов расширения для существующих иерархий.

1.2. Порождающие шаблоны проектирования

В большинстве объектно-ориентированных языков программирования за конструирование объекта отвечает конструктор. В самом простом случае клиент знает, какого типа объект ему требуется, и создает его путем вызова соответствующего конструктора. Но в некоторых случаях тип объекта может быть неизвестен вызывающему коду или процесс конструирования может быть настолько сложным, что использование конструктора будет неудобным или невозможным. Порождающие шаблоны предназначены для решения типовых проблем создания объектов.

1.2.1. Шаблон «Одиночка» (Singleton)

Данный шаблон гарантирует, что у класса есть только один экземпляр, и предоставляет глобальную точку доступа к нему. Другими словами, синглтон эмулирует глобальные переменные в объектно-ориентированных языках программирования.

Практически в любом приложении возникает необходимость в глобальных переменных или объектах с ограниченным числом экземпляров. Даже в таком простом приложении, как импорт логов, может возникнуть необходимость в логировании. И самый простой способ решить эту задачу — создать глобальный объект, который будет доступен из любой точки приложения.

По своему определению одиночка гарантирует, что у некоего класса есть лишь один экземпляр. В некоторых случаях анализ предметной области строго требует, чтобы класс существовал лишь в одном экземпляре. Однако на практике шаблон «Одиночка» обычно используется для обеспечения доступа к какому-либо ресурсу, который требуется разным частям приложения.

Уже ни для кого не секрет, что количество недостатков у синглтона таково, что можно считать его не столько паттерном, сколько антипаттерном. Бездумное и бесконтрольное его использование однозначно приведет к проблемам сопровождения, но это не значит, что у него нет сферы применения.

Синглтон без видимого состояния. Нет ничего смертельного в использовании синглтона, через который можно получить доступ к стабильной справочной информации или некоторым утилитам.

Настраиваемый контекст. Аналогично нет ничего смертельного в протаскивании инфраструктурных зависимостей в виде Ambient Context, то есть в использовании синглтона, возвращающего абстрактный класс или интерфейс, который можно установить в начале приложения или при инициализации юнит-теста.

Минимальная область использования. Ограничьте использование синглтона минимальным числом классов/модулей. Чем меньше у синглтона прямых пользователей, тем легче будет от него избавиться и перейти на более продуманную модель управления зависимостями. Помните, что чем больше у классов пользователей, тем сложнее его изменить. Если уж вы вынуждены использовать синглтон, возвращающий бизнес-объект, то пусть лишь несколько высокоуровневых классов-медиаторов используют синглтоны напрямую и передают его экземпляр в качестве зависимостей классам более низкого уровня.

Сделайте использование синглтона явным. Если передать зависимость через аргументы конструктора не удается, то сделайте использование синглтона явным. Вместо обращения к синглтону из нескольких методов сделайте статическую переменную и проинициализируйте ее экземпляром синглтона.

1.2.2. Шаблон «Абстрактная фабрика» (Abstract Factory)

Фабрика — это второй по популярности шаблон после синглтона. Существуют две классические разновидности фабрик: «Абстрактная фабрика» и «Фабричный метод», предназначенные для инкапсуляции создания объекта или семейства объектов. На практике очень часто отходят от классических реализаций этих шаблонов и называют фабрикой любой класс, инкапсулирующий в себе создание объектов.

Абстрактная фабрика предоставляет интерфейс для создания семейства взаимосвязанных или родственных объектов (dependent or related objects), не специфицируя их конкретных классов. Другими словами, абстрактная фабрика представляет собой стратегию создания семейства взаимосвязанных или родственных объектов.

Выразительность наследования обеспечивается за счет полиморфизма. Использование интерфейсов или базовых классов позволяет абстрагироваться от конкретной реализации, что делает решение простым и расширяемым. Однако где-то в приложении должна быть точка, в которой создаются объекты и известен их конкретный тип.

В некоторых случаях решение о конкретных типах можно откладывать до последнего, вплоть до корня приложения (Application Root). В этом случае конструирование конкретных типов происходит в методе Main (или аналогичном методе в зависимости от типа приложения), и затем созданные объекты передаются для последующей обработки. Однако в некоторых случаях создание объекта должно происходить раньше — в коде приложения.

Основная особенность абстрактной фабрики заключается в том, что она предназначена для создания семейства объектов, что сильно сужает ее применимость.

Но в некоторых предметных областях или инфраструктурном коде периодически возникают задачи, которые решаются набором классов: сериализаторы/десериализаторы, классы для сжатия/распаковки, шифрования/дешифрования и т. п. Приложение должно использовать согласованные типы объектов, и абстрактная фабрика идеально подходит для решения этой задачи. Интерфейс абстрактной фабрики объявляет набор фабричных методов, а конкретная реализация обеспечивает создание этого семейства объектов.

Абстрактная фабрика представляет собой слой для полиморфного создания семейства объектов. Ее использование подразумевает обязательное наличие двух составляющих: семейства объектов и возможности замены создаваемого семейства объектов во время исполнения.

Иногда необходимость полноценной абстрактной фабрики очевидна с самого начала, но обычно лучше начать с наиболее простого решения и добавить гибкость лишь в случае необходимости.

1.2.3. Шаблон «Фабричный метод» (Factory Method)

Назначение: определяет интерфейс для создания объекта, но оставляет подклассам решение о том, какой класс инстанцировать. Фабричный метод позволяет классу делегировать инстанцирование подклассам.

Как было рассмотрено в п.1.2.2, иерархии классов обеспечивают гибкость за счет полиморфного использования, но привносят дополнительную сложность. Абстрактная фабрика решает задачу полиморфного создания семейства объектов, но очень часто возникает более простая задача — создания одного экземпляра иерархии наследования.

Цель любой фабрики — оградить клиентов от подробностей создания экземпляров класса или иерархии классов.

На практике встречаются три вида шаблона «Фабричный метод»:

- классическая реализация на основе шаблонного метода;
- статический фабричный метод;
- полиморфный фабричный метод.

Классический фабричный метод является частным случаем шаблонного метода. Это значит, что фабричный метод привязан к текущей иерархии типов и не может быть использован повторно в другом контексте.

Полиморфный фабричный метод является стратегией создания экземпляров некоторого семейства типов, что позволяет использовать одну фабрику в разных контекстах. Тип создаваемого объекта определяется типом фабрики и обычно не зависит от аргументов фабричного метода.

Статический фабричный метод является самой простой формой фабричного метода. Статический метод создания позволяет обойти ограничения конструкторов. Например, тип создаваемого объекта может зависеть от аргументов метода, экземпляр может возвращаться из кэша, а не создаваться заново или же фабричный метод может быть асинхронным.

1.3. Структурные шаблоны проектирования

Любая грамотно спроектированная программная система является иерархической. В корне приложения, например, в методе Main, создается набор высокоуровневых компонентов, каждый из которых опирается на компоненты более низкого уровня. Эти компоненты, в свою очередь, также могут быть разбиты на более простые составляющие.

Существует набор типовых решений, которые помогают бороться со сложностью, создавать и развивать современные системы. Так, например, довольно часто возникает потребность в полиморфном использовании классов, которые выполняют схожую задачу, но не обладают единым интерфейсом. Такая ситуация возможна, когда классы разрабатывались в разное время, разными людьми, а иногда и разными организациями. Связать такие классы вместе позволит шаблон «Адаптер». Он дает возможность подстроить разные реализации к одному интерфейсу и использовать их полиморфным образом даже тогда, когда часть существующего кода находится вне вашего контроля.

Еще одним распространенным структурным шаблоном является «Фасад». Он представляет высокоуровневый интерфейс к сторонней библиотеке или модулю, что упрощает код клиентов и делает их менее зависимыми от стороннего кода. Это позволяет существенно упростить логику приложения и делает ее менее зависимой от ошибок и изменений сторонних библиотек. Фасад упрощает миграцию на новую версию библиотеки, поскольку придется не проверять весь код приложения, а лишь запустить тесты фасада и убедиться в том, что поведение осталось неизменным.

Шаблон «Декоратор» позволяет нанизывать дополнительные аспекты поведения один на другой без создания чрезмерного числа производных классов. Декоратор прекрасно справляется с задачами кэширования, логирования или с ограничением числа вызовов. При этом за каждый аспект поведения будет отвечать выделенный класс, что упростит понимание и развитие системы.

1.3.1. Шаблон «Адаптер» (Adapter)

Преобразует интерфейс одного класса в интерфейс другого, который ожидают клиенты. Адаптер делает возможной совместную работу классов с несовместимыми интерфейсами.

Далеко не все системы обладают прекрасным дизайном. Даже если модуль или приложение были хорошо продуманы изначально, внесение изменений разными разработчиками в течение длительного времени может

привести к неприятным последствиям. Одно из таких последствий — рассогласованность реализации однотипных задач.

Адаптер является одним из тех шаблонов проектирования, которые мы используем, почти не задумываясь. Диаграмма классов этого шаблона настолько общая, что практически любую композицию объектов можно считать примером использования адаптеров.

«Бандой четырех» были описаны два вида адаптеров — адаптеры классов и адаптеры объектов. Ранее был рассмотрен пример адаптера объектов. В этом случае создается новый класс, который реализует требуемый интерфейс и делегирует всю работу адаптируемому объекту, хранящемуся в виде закрытого поля.

Классический адаптер классов использует множественное наследование. Адаптер реализует новый интерфейс, но также использует адаптируемый класс в качестве базового класса. Обычно в этом случае используется закрытое наследование, что предотвращает возможность конвертации адаптера к адаптируемому объекту.

Наследование от адаптируемого объекта позволяет получить доступ к защищенному представлению, а также реализовать лишь несколько методов, если новый интерфейс не слишком отличается от интерфейса адаптируемого класса.

Адаптер интерфейсов не всегда можно применить, поскольку это требует изменения адаптируемого класса. Но, если такое изменение возможно, это может быть самой простой реализацией шаблона «Адаптер».

Адаптер позволяет использовать существующие типы в новом контексте.

Повторное использование чужого кода. В некоторых случаях у нас уже есть код, который решает нужную задачу, но его интерфейс не подходит для текущего приложения. Вместо изменения кода библиотеки можно создать слой адаптеров.

Адаптивный рефакторинг. Адаптеры позволяют плавно изменять существующую функциональность путем выделения нового «правильного» интерфейса, но с использованием старой проверенной функциональности.

1.3.2. Шаблон «Фасад» (Facade)

Шаблон Facade предоставляет унифицированный интерфейс вместо набора интерфейсов некоторой подсистемы. Фасад определяет интерфейс более высокого уровня, который упрощает использование подсистемы.

Библиотеки классов обычно предназначены для решения целого спектра задач. Наличие сложных сценариев делает библиотеку полезной, но это же может усложнить решение с ее помощью простых задач. Это приводит к появлению всевозможных оболочек даже над стандартными библиотеками, которые упрощают решение простых задач, или сценариев, специфичных для конкретного приложения.

Такие оболочки являются фасадами, которые скрывают исходную сложность библиотеки или модуля за более простым и, возможно, специфичным для приложения интерфейсом.

В большинстве приложений используется лишь часть функциональности сложной библиотеки, и фасад позволяет сделать более простой интерфейс, максимально подходящий для специфических сценариев.

Очень многие библиотеки или подсистемы приложений содержат встроенные фасады, которые являются высокоуровневыми классами, предназначенными для решения типовых операций. Фасады делают базовые сценарии простыми, а сложные сценарии — возможными. Если клиенту нужна лишь базовая функциональность, достаточно воспользоваться фасадом; если же его функциональности недостаточно, можно использовать более низкоуровневые классы модуля или библиотеки напрямую.

Использование фасадов не только упрощает использование библиотек или сторонних компонентов, но и решает ряд насущных проблем.

Повторное использование кода и лучших практик. Многие библиотеки довольно сложные, поэтому их корректное использование требует определенных навыков. Инкапсуляция работы с ними в одном месте позволяет корректно использовать их всеми разработчиками независимо от их опыта.

Переход на новую версию библиотеки. При выходе новой версии библиотеки достаточно будет протестировать лишь фасад, чтобы принять решение, стоит на нее переходить или нет.

Переход с одной библиотеки на другую. Благодаря фасаду приложение не так сильно завязано на библиотеку, так что переход на другую библиотеку потребует лишь создание еще одного фасада. А использование адаптера сделает этот переход менее болезненным.

Фасад повышает уровень абстракции и упрощает решение задач, специфичных для текущего приложения. Фасад скрывает низкоуровневые детали, но может предоставлять интерфейс, специфичный для конкретного приложения, за счет использования в качестве параметров доменных объектов.

Фасады можно использовать для работы с большинством сторонних библиотек или подсистем. Это уменьшает связанность (coupling) системы с внешними зависимостями, позволяет лучше понять сторонний код, контролировать качество новых версий, а также избавляет код приложения от излишних низкоуровневых деталей.

Фасад не нужно применять, когда в повышении уровня абстракции нет никакого смысла. В большинстве случаев нет смысла в фасаде для библиотеки логирования, достаточно выбрать одно из решений (log4net, NLog, .NETTraces) и использовать его в коде приложения. Фасады бесполезны, когда имеют громоздкий интерфейс, и пользоваться ими сложнее, чем исходной библиотекой.

1.3.3. Шаблон «Декоратор» (Decorator)

Шаблон «Декоратор» динамически добавляет объекту новые обязанности. Является гибкой альтернативой порождению подклассов с целью расширения функциональности.

Хорошо спроектированный класс отвечает за определенную функциональность, не распыляясь на решение второстепенных задач. Но что делать, если второстепенные задачи, такие как логирование, кэширование, замеры времени исполнения, проникают в код класса и непомерно увеличивают сложность реализации? Можно выделить эти аспекты поведения во вспомогательные классы, но все равно останется проблема их координации. Шаблон «Декоратор» элегантно решает задачу нанизывания одних обязанностей на другие.

Декоратор позволяет динамически расширять поведение объектов. Он идеально подходит для расширения поведения всех методов интерфейса, которое не является частью основной функциональности. Если кэшировать нужно лишь результаты одного метода класса, то использование декоратора будет слишком тяжеловесным.

Декораторы применяются для добавления всем методам интерфейса некоторого поведения, которое не является частью основной функциональности. Декораторы отлично подходят для решения следующих задач:

кэширования результатов работы;
замера времени исполнения методов;
логирования аргументов;
управления доступом пользователей;
модификации аргументов или результата работы методов упаковки/распаковки, шифрования и т. п.

Динамическая природа позволяет нанизывать аспекты один на другой, обходя ограничения наследования, использование которого привело бы к комбинаторному взрыву числа наследников.

1 Практические исследования

2.1 Постановка задачи

Цель данного курсового проекта – создать демонстрационное java приложение по теме «Разработка системы конвертации валют, по наиболее низкому курсу» в котором используются шаблоны проектирования.

Приложение будет использоваться пользователями для выбора наилучшего пути обмена валюты из текущей в новую валюту. Проводить операции с данными может любой пользователь

Разработанное приложение реализует следующий функционал:

- Просмотр данных о валюте;
- Добавление данных о валюте;
- Редактирование данных о валюте;
- Удаление данных о валюте;
- Просмотр курса валют;
- Добавление курса валют;
- Редактирование курса валют;
- Удаление курса валют;
- Просмотр записи об обмене валюты;
- Добавление записи об обмене валюты;
- Удаление записи об обмене валюты;

2.2 Описание сущностей

В этом подразделе будут описаны сущности и классы приложения.

Первой сущностью является абстрактная сущность AbstractEntity с единственным полем id типа Long, от которой наследуются все остальные сущности. После это используется для идентификации каждой сущности.

Первая из наследуемых сущностей представляет собой валюту. Название этой сущности – Currency. Она имеет следующее поле name типа String. В этом поле хранится название валюты (будь то доллар, рубль, евро иди др.).

Вторая наследуемая сущность представляет из себя курс валюты по отношению к доллару. Наименование сущности Course. Она включает в себя следующие поля:

- Поле currency типа Currency, которое отвечает за выбранную валюту, которой будет назначен курс;
- Поле courseToOneDollar типа Float, в которое пишется отношение единицы валюты к доллару. Пример, доллар всегда равен одному доллару. Один белорусский рубль равен 0,393 доллара, одно евро равно 1.22 доллара и тд.
- Поле where типа String, которое отвечает за местоположение. В нем пишется наименование банка, в котором указан текущий курс валюты.

Последней наследуемой сущностью является конвертация валют. Конвертировать можно и без сохранения данных в базе, но если для этого будет необходимость и есть данная сущность. Ее название – ConvertSaver. Она имеет следующие поля:

- Поле currencyCurrent типа Currency, которое показывает, какую валюту пользователь хочет обменять.
- Поле currencyNew типа Currency, которое отвечает за показ валюты, которую пользователь хочет получить.
- Поле count типа Float, в которое от пользователя заносится информация о количестве валюты для обмена.
- Поле courseToOneDollar типа String, в которое будет занесена информация о лучшем курсе для обмена валют.
- Поле where типа String, в которое будет занесена информация о банках, в которых лучше всего будет обменять валюту.
- Поле dates типа Date, отвечающее за дату обмена валюты.

– Поле summ типа Float, показывающая итоги обмена, т.е. сколько новой валюты получит пользователь.

Диаграмма этих сущностей показана на рисунке 2.1.

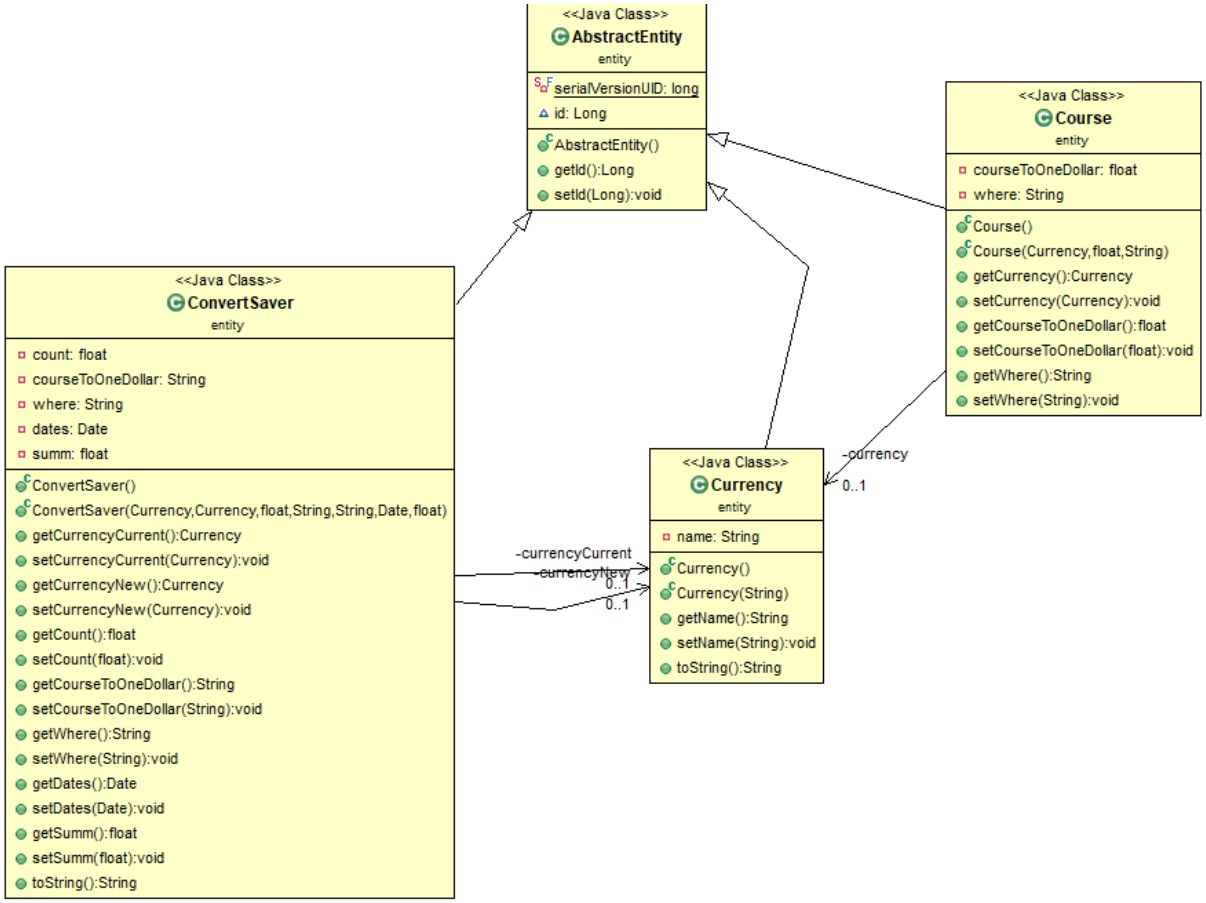

Рисунок 2.1 – Диаграмма сущностей

Для взаимодействия с базой данных были созданы классы DAO, которые соединяются с базой данных и взаимодействуют с информацией внутри нее. Все эти классы реализуют стандартный список методов для взаимодействия с бд (create, read, update, delete, readAll, find).

```
// текущее подключение к бд
	private Connection myConn;
// конструктор по умолчанию
	public CurrencyDAO() throws Exception {
		//Создаем тип свойства и читаем файл с свойствами
		Properties props = new Properties();
		props.load(new FileInputStream("db.properties"));
		// из файла получаем строку подключения к бд
```

```
            String dburl = props.getProperty("dburl");
            //подключаемся по этой строке к бд
            myConn = DriverManager.getConnection(dburl, "", "");
            System.out.println("DB Currency connection success");
      }
// метод получения всех записей из бл
      public List<Currency> readAll() throws Exception {
            // заранее создаем список
            List<Currency> list = new ArrayList<Currency>();
            // и параметры запроса
            Statement myStmt = null;
            ResultSet myRs = null;

            try {
                  // выполняем запрос к бд
                  myStmt = myConn.createStatement();
                  myRs = myStmt.executeQuery("SELECT * FROM currency");
                  while (myRs.next()) {
                        // каждую полученную запись преобразуем в сущность и
записываем в лист
                        Currency tempEntity = convertRowToEntity(myRs);
                        list.add(tempEntity);
                  }
                  // возвращаем лист вызывающему методу
                  return list;
            } finally {
                  close(myStmt, myRs);
            }
      }
      // метод поиска в бд записей по части названия
      public List<Currency> search(String name) throws Exception {
            List<Currency> list = new ArrayList<Currency>();
            // создаем переменный для параметров запроса
            PreparedStatement myStmt = null;
            ResultSet myRs = null;

            try {
                  // выполняем запрос
                  name = "%" + name + "%";
                  myStmt = myConn.prepareStatement("SELECT * FROM currency WHERE
name LIKE ?");
                  myStmt.setString(1, name);
                  myRs = myStmt.executeQuery();
                  while (myRs.next()) {
                        // каждую полученную запись преобразуем в сущность и
записываем в лист
                        Currency tempEntity = convertRowToEntity(myRs);
                        list.add(tempEntity);
                  }
                  // возвращаем лист вызывающему методу
                  return list;
            } finally {
                  close(myStmt, myRs);
            }
      }
      // метод добавления записи в бд, на вход принимает сущность, которую нужно
сохранить
      public void create(Currency entity) throws Exception {
            PreparedStatement myStmt = null;
            try {
                  // выполняем запрос к бд
```

```
                    myStmt = myConn.prepareStatement("insert into currency" + "
(name)" + " values (?)");
                    myStmt.setString(1, entity.getName());
                    myStmt.executeUpdate();
             } finally {
                    close(myStmt);
             }
      }
      // метод чтения по ид записи
      public Currency read(Long id) throws Exception {
             PreparedStatement myStmt = null;
             ResultSet myRs = null;
             try {
                    // выполняем запрос и возвращаем результат, если он есть
                    myStmt = myConn.prepareStatement("SELECT * FROM currency WHERE
id=?");
                    myStmt.setLong(1, id);
                    myRs = myStmt.executeQuery();
                    while (myRs.next()) {
                           return convertRowToEntity(myRs);
                    }

             } finally {
                    close(myStmt, myRs);
             }
             // если его нет, возвращаем нулевой результат
             return null;
      }
      // метод обновления, принимает на вход новое имя валюты и ид записи в бд,
которую обновляем
      public void update(String nameNew, Long id) throws Exception {
             PreparedStatement myStmt = null;
             try {
                    // выполняем запрос к бд, передав все параметры
                    myStmt = myConn.prepareStatement("UPDATE currency SET name=?
WHERE id=?");
                    myStmt.setString(1, nameNew);
                    myStmt.setLong(2, id);
                    myStmt.executeUpdate();
             } finally {
                    close(myStmt);
             }
      }
      // метод удаления записи из бд, на вход принимает ид удаляемой записи
      public void Delete(Long id) throws Exception {
             PreparedStatement myStmt = null;
             try {
                    myStmt = myConn.prepareStatement("DELETE FROM currency WHERE
id=?");
                    myStmt.setLong(1, id);
                    myStmt.executeUpdate();
             } finally {
                    close(myStmt);
             }
             }
```

Диаграмма этих DAO классов отображена на рисунке 2.2

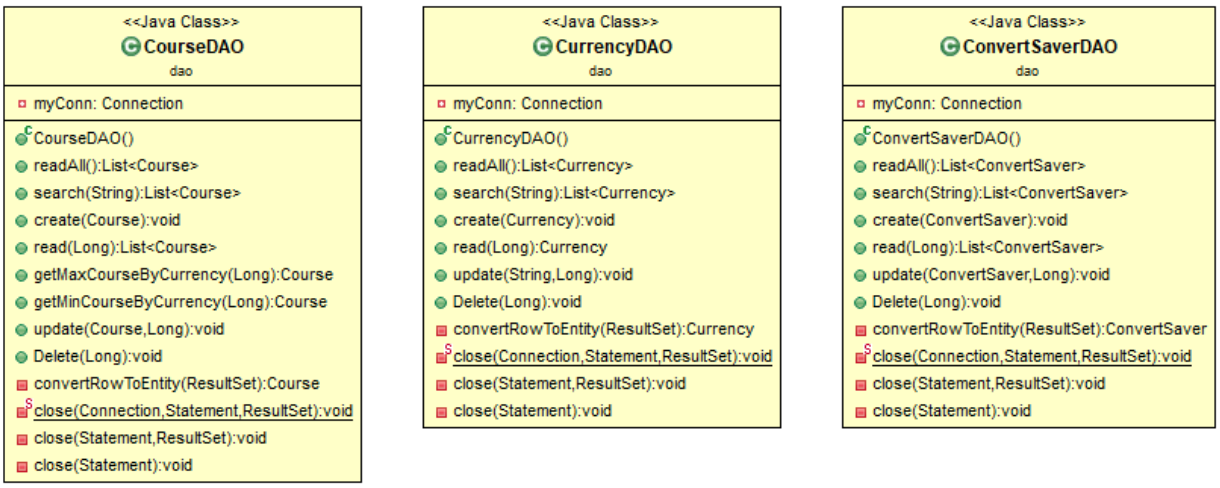

Рисунок 2.2 – Диаграмма DAO классов

Кроме описанных выше классов, приложение также имеет классы-контроллеры, которые используются для контроля взаимодействия пользовательского интерфейса с базой данных и классы пользовательского интерфейса.

Всего есть 5 контроллеров и 4 формы.

Контроллеры отвечают за взаимодействие интерфейса с базой данных.

ApplicationController является главным контроллером приложения, который запускает саму программу и имеет ссылки на все остальные контроллеры.

MainController является контроллером главной формы, на которой расположены кнопки перехода между остальными формами.

CurrencyController является контроллером формы валют, отвечающим на все вызовы формы вызовами методов DAO класса к базе.

CourseController отвечает за форму курсов валют.

ConvertController является последним контроллером и также отвечает за форму отображения, на этот раз за конвертацию валюты и сохранение ее в базе данных.

Общая диаграмма классов показана на рисунке 2.3.

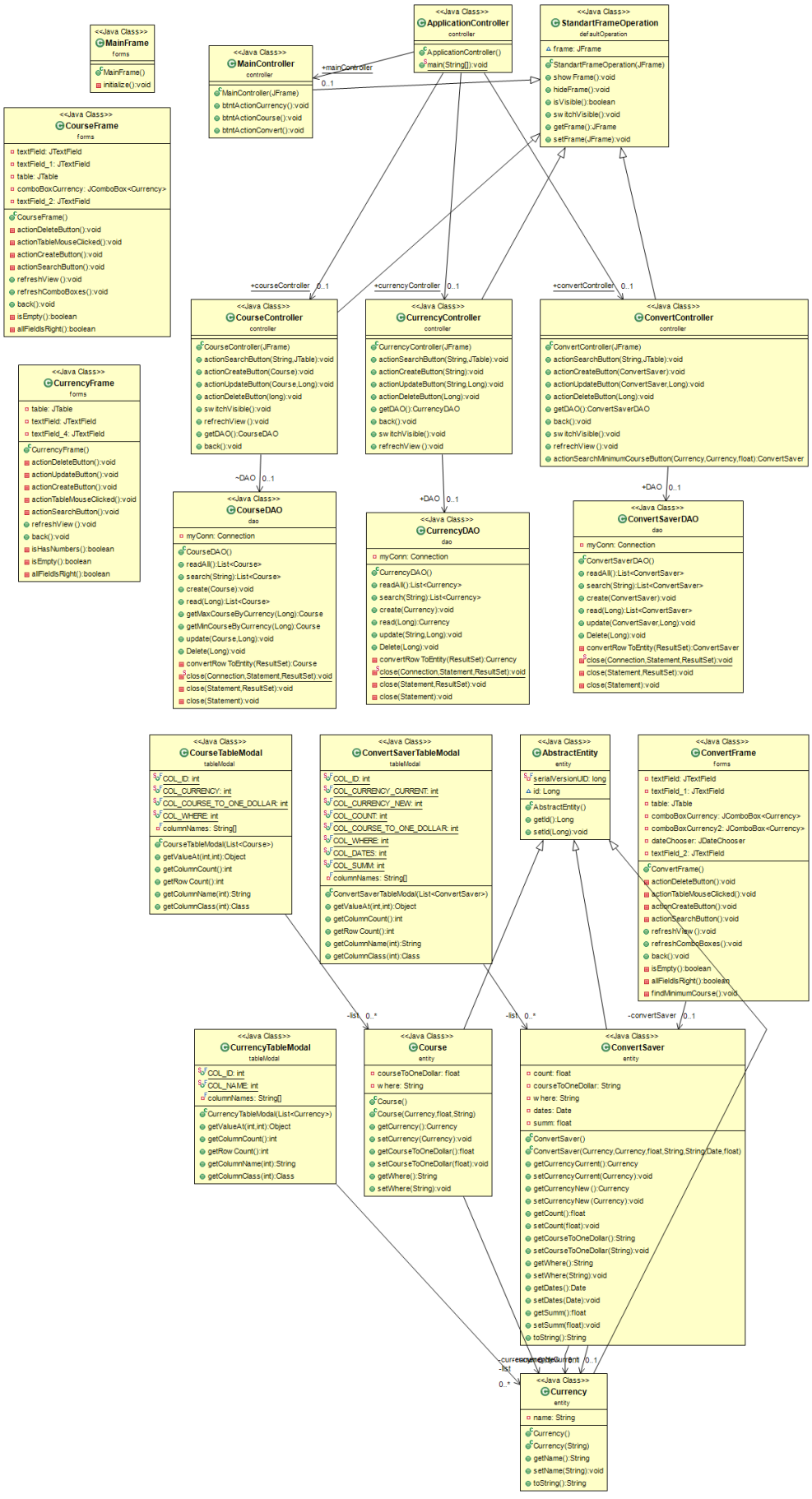

Рисунок 2.3 – Диаграмма классов

2.3 Реализация шаблонов проектирования

В рамках создания приложения были использованы следующие шаблоны проектирования:

- Model View Controller (Модель-Представление-Контроллер)
- Observer

MVC —шаблон проектирования, который позволяет разрабатывать веб-приложения на Java с использованием архитектуры Model — View — Controller.

Архитектура Model — View — Controller состоит из трех компонентов, заложенных в ее названии. Компонент Model подразумевает логику работы с данными, View — логику интерфейса, Controller — логику обработки запросов.

Проще говоря, Controller получает данные из Model и затем эти данные отображает во View (представлении — ред.). Также Controller обрабатывает запросы от пользователя и переадресовывает его на необходимые формы приложения. Форма и есть компонентом View, который видит пользователь.

Моделями в этом шаблоне являются сущности приложения, которые описывали выше – ConvertSaver, Course, Currency.

Отображениями являются формы приложения: ConvertFrame, CourseFrame, CurrencyFrame, MainFrame.

За всю логику отвечают контроллеры, описанные выше.

Observer – является поведенческим шаблоном проектирования. Поведенческий шаблон проектирования — это такой шаблон, который решает какую-то задачу, связанную с поведением. В данном случае Observer как раз позволяет слушать некоторые поведения и реагировать на некоторые события. Данный шаблон часто используется в компонентах Swing (в нашем случае). Например, при добавлении кнопки на форму к этой кнопке

добавляется слушатель событии. Это построено по принципе подписчика и слушателя. Т.е. слушатель наблюдает за нажатием кнопки и как только кнопка нажимается – он отрабатывает какое-то событие. Именно таким образом данный шаблон был реализован и разработанном проекте. Ниже предоставлен код заголовков кнопок в которых использовался шаблон.

```
		// кнопка поиска фалюты по имени
		JButton btnNewButton_1 = new JButton("Искать");
		// добавление кнопке слушателя. При активации слушателя, вызвать внешний
метод
		btnNewButton_1.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e) {
				actionSearchButton();
			}
		});
		panel.add(btnNewButton_1);

		JPanel panel_1 = new JPanel();
		panel_1.setBorder(new EmptyBorder(3, 3, 3, 3));
		getContentPane().add(panel_1, BorderLayout.WEST);

		JLabel lblNewLabel = new JLabel("Название");

		textField = new JTextField();
		textField.setColumns(10);
// кнопка добаления записи в бд
		JButton btnNewButton = new JButton("Добавить");
		// добавление кнопке слушателя. При активации слушателя, вызвать внешний
метод
		btnNewButton.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e) {
				actionCreateButton();
			}
		});
//кнопка редактирования записи бд
		JButton btnNewButton_2 = new JButton("Редактировать");
		// добавление кнопке слушателя. При активации слушателя, вызвать внешний
метод
		btnNewButton_2.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e) {
				actionUpdateButton();
			}
		});
// кнопка удаления записи бд
		JButton btnNewButton_3 = new JButton("Удалить");
		// добавление кнопке слушателя. При активации слушателя, вызвать внешний
метод
		btnNewButton_3.addActionListener(new ActionListener() {
			public void actionPerformed(ActionEvent e) {
				actionDeleteButton();
			}
		});
```

2.4 Тестирование приложения

Запустив приложение, пользователю будет показано главное окно (форма) программы.

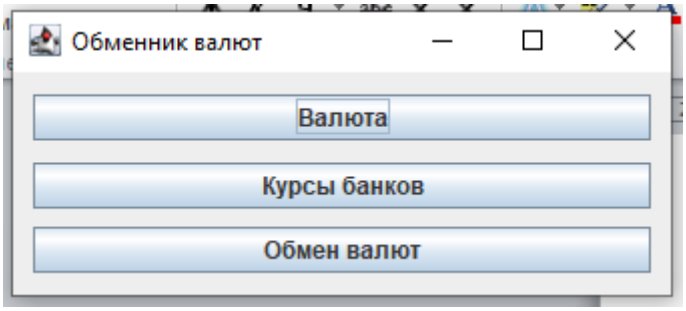

Рисунок 2.4 – Главная форма

На этой странице пользователь может выбрать одно из трех меню: перейти на страницу валют, перейти на страницу курсов различных банков или же открыть форму обмена валют. Пройдем каждый вариант по очереди.

Перейдем на страницу валют, пользователю будет отображена форма как на рисунке 2.5.

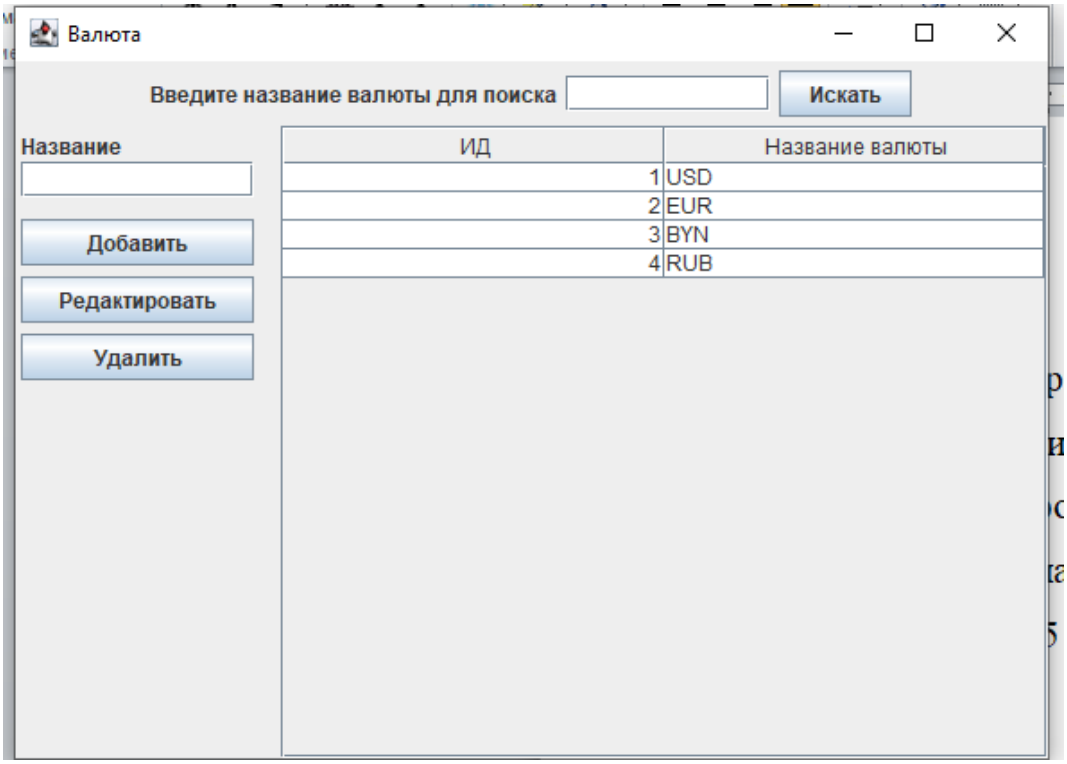

Рисунок 2.5 – Форма валют

На этой форме пользователь может посмотреть список всех валют, которые участвуют в жизни приложения. При необходимости можно добавить или изменить существующую валюту, или же удалить ее вовсе. На всякий случай сделана проверка, чтобы нельзя было добавить в таблицу пустые значения, а также проверка на ввод запрещенных символов (цифр) в поле.

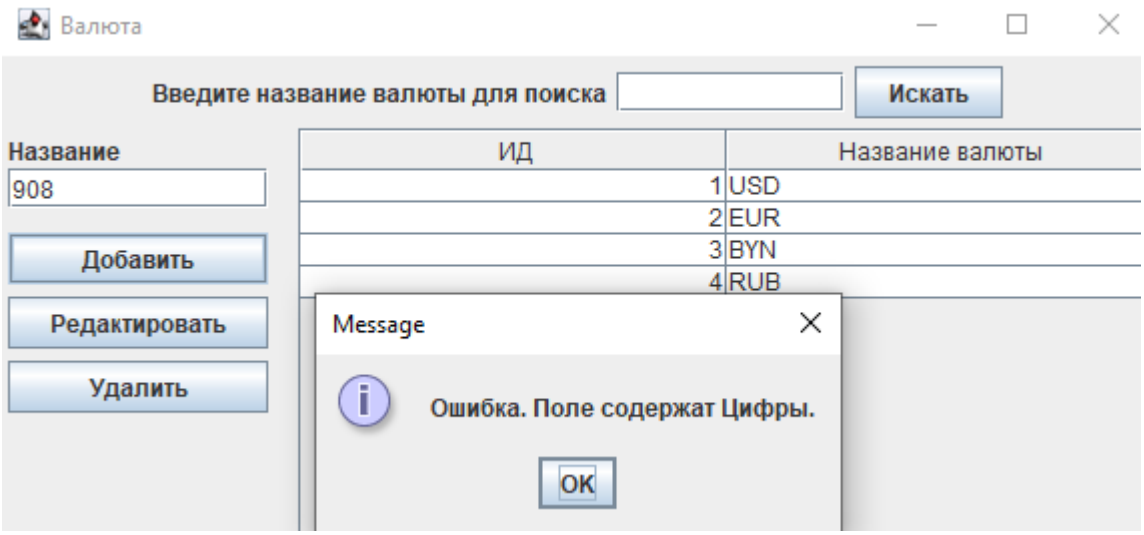

Рисунок 2.6 – Ошибка при неправильном заполнении поля

Для дальнейшей работы добавим собственную валюту

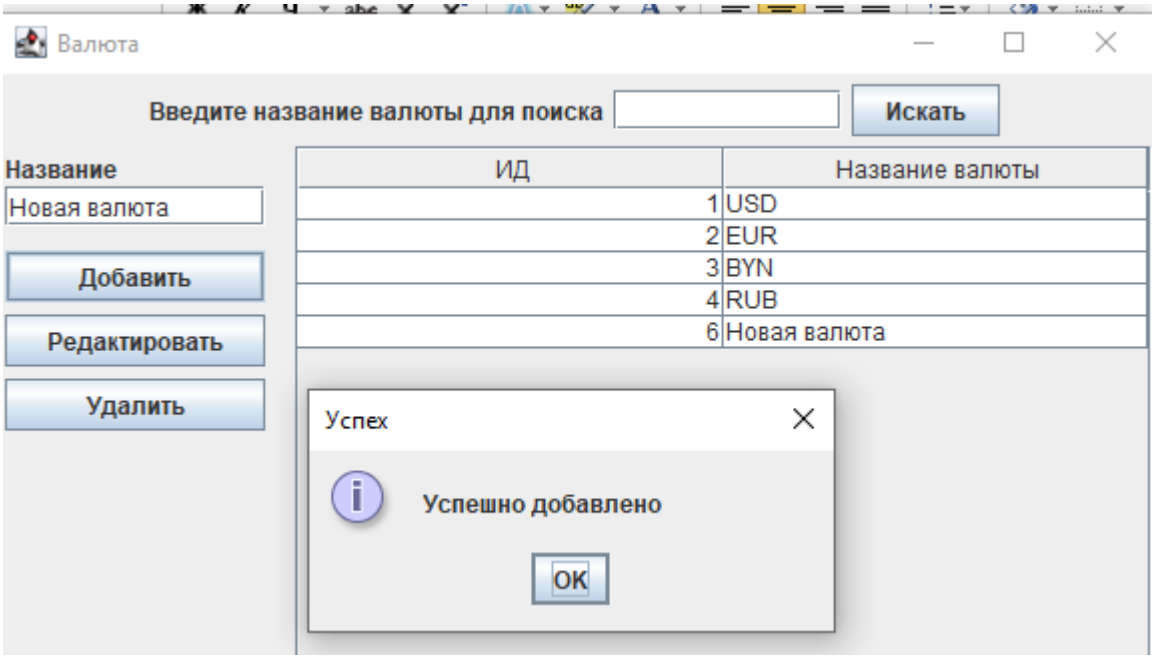

Рисунок 2.7 – Добавление собственной валюты

Закроем форму и перейдем из главной страницы в курс валют. Пользователю будет показана форма, отображенная на рисунке 2.8.

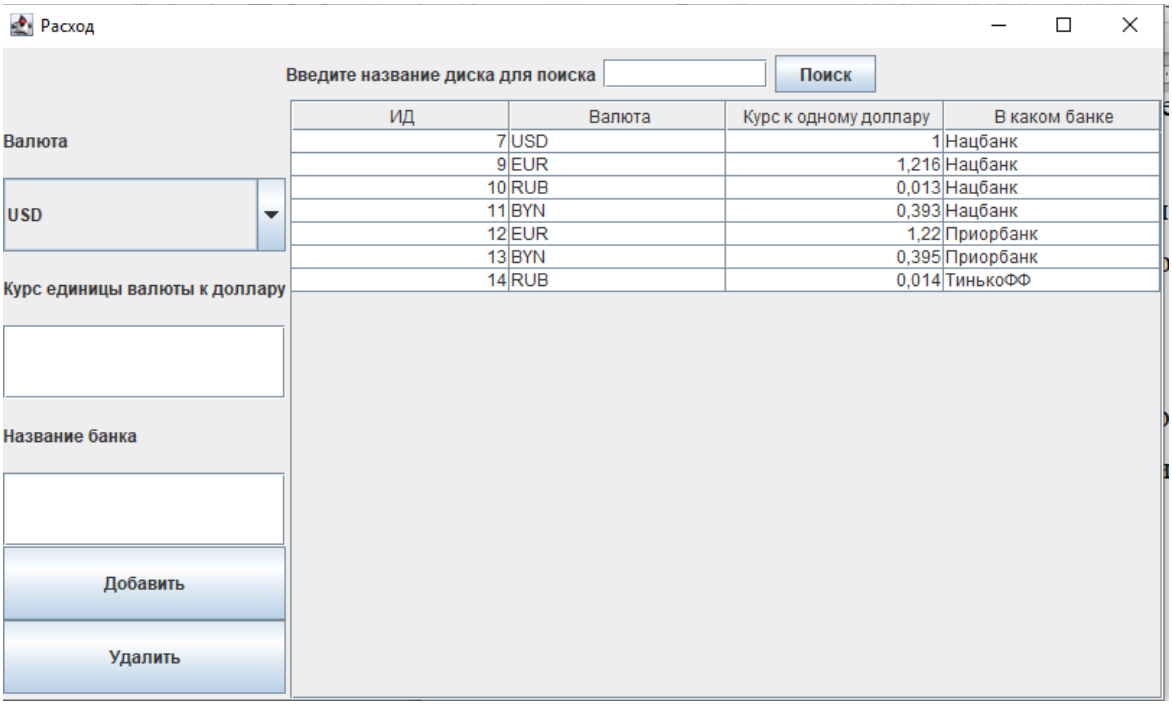

Рисунок 2.8 – Форма курс валют

На данной форме отображено, какая валюта как относится к доллару. Иначе говоря, если за 1 белорусский рубль можно купить 0,393 доллара, то так и должно быть занесено в эту таблицу. Занесем в нее свои значения. К примеру, пусть наша валюта будет с долларом в отношении 2 к 1, т.е. за 1 единицу нашей валюты можно купить 2 доллара.

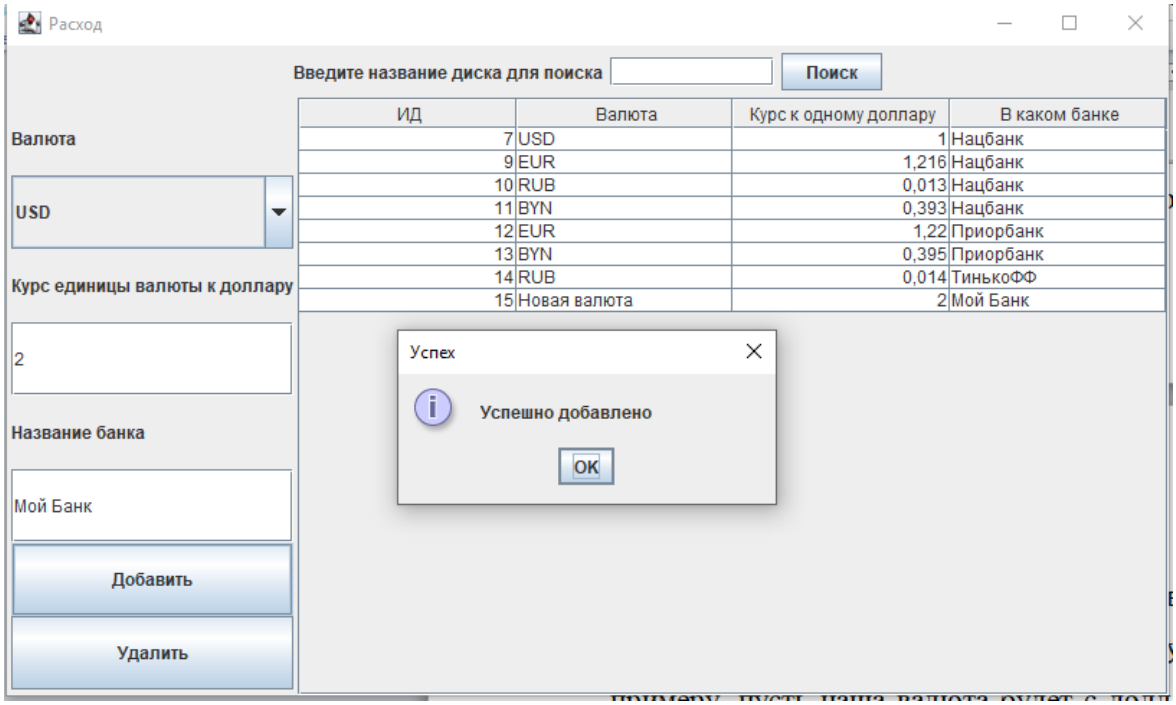

Рисунок 2.9 – Добавление курса своей валюте.

Теперь остается перейти на последнюю форму приложения и проверить работоспособность. Форма обмена показана на рисунке 2.10.

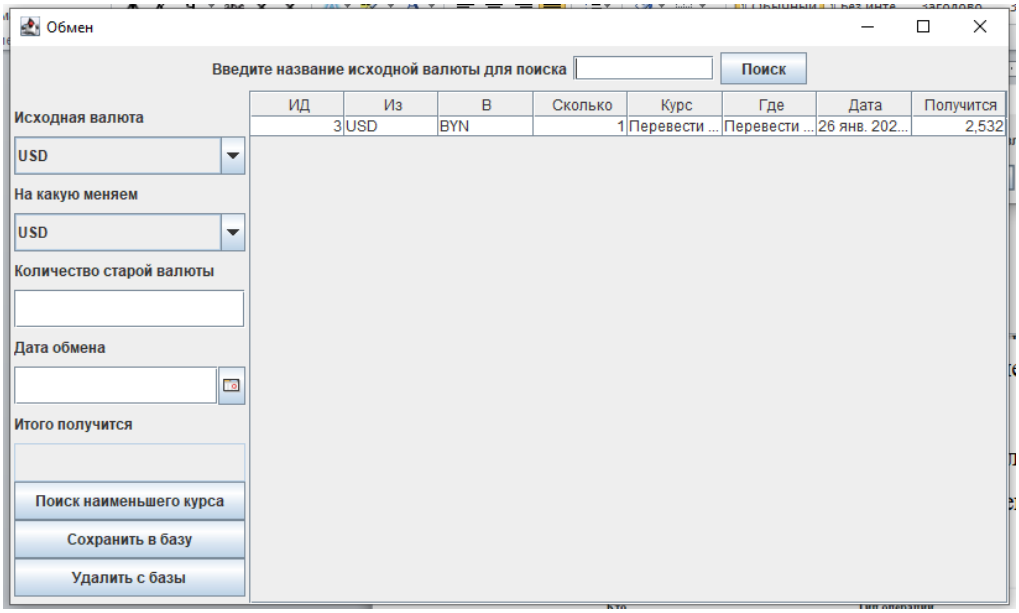

Рисунок 2.10 – Форма обмена

Попробуем перевести одну единицу нашей валюты в доллары и посмотрим что выйдет.

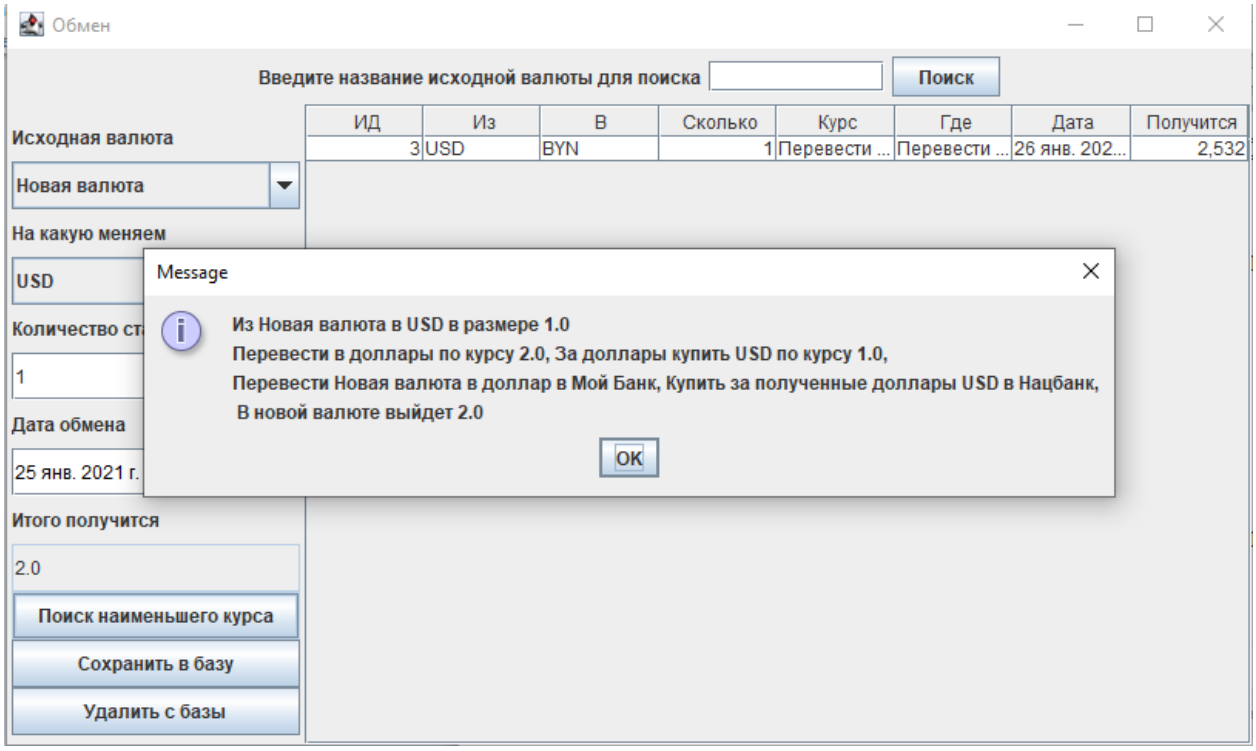

Рисунок 2.11 – Результат перевода собственной валюты в доллары

Результат ожидаемый. Нам отображен ход действий, с помощью которого можно перевести валюты с наилучшей выгодой и по меньшему курсу.

Сперва в сообщении показывает, из какой валюты и в какую мы переводим. Затем отображается лучший курс для перевода текущей валюты в доллары, поскольку весь курс высчитывается по отношению к нему. Название банков для перевода в доллары и в желаемую валюты показаны после этого. Последней строкой показан итоговый результат: сколько новой валюты будет получено.

Для подтверждения эксперимента, в курсах валют поставим нашу валюту по курсу 3 и 0.5 к доллару и посмотрим, выберет ли приложение минимальный курс.

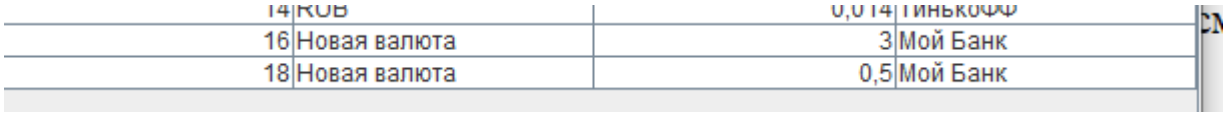

Рисунок 2.12 – Курс собственной валюты

Подтверждение тому, что по лучшему курсу за 1 доллар мы можем купить 2 единицы нашей валюты отображен на рисунке 2.13. Если бы приложение выбрало не наименьший курс, 3 единицы новой валюты стоили бы только 1 доллар, а так 1 единица новой валюты стоят целых 0.5 доллара по наименьшему курсу.

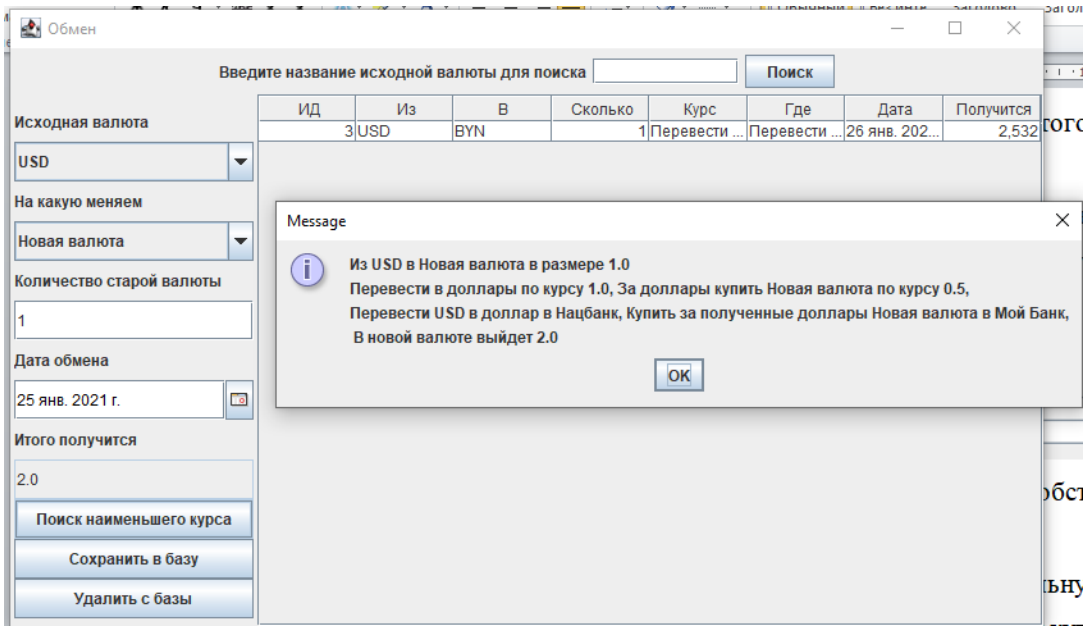

Рисунок 2.13 – Перевод из долларов в нашу валюты

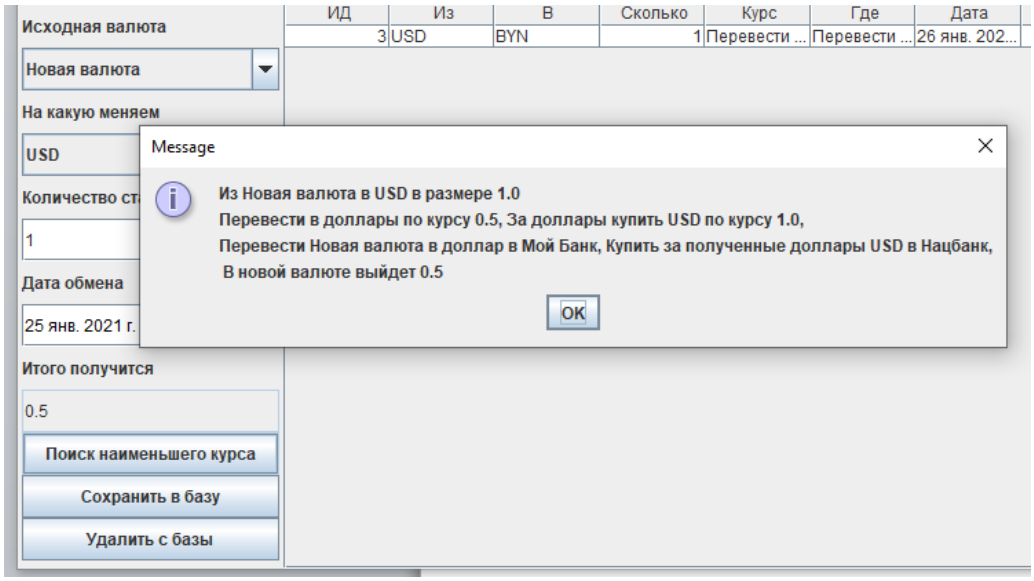

Рисунок 2.14 – Перевод из нашей валюты в доллары

Таким образом, был рассмотрен весь функционал приложения, который выполняет поставленные задачи без ошибок и будет являться отличным помощником при учете доходов и расходов семьи.

Заключение

В результате выполнения курсовой работы было разработано приложение, предназначенное для системы конвертации валют, по наиболее никому курсу. Разработка приложения осуществлялась в среде программирования Eclipse на языке Java на платформе Java EE с использованием Swing.

Одним из главных назначений разработанного приложения является осуществление быстрого и удобного доступа к необходимой информации, что позволит пользователю эффективно использовать свое время. Для реализации этого назначения в приложении предусмотрена такая функциональность, как добавление, редактирование и удаление данных используемых таблиц.

Список источников