Лабораторная работа №1

Тема занятия: «Использование языка программирования Kotlin»

Цель: Выполнить разработку приложения с использованием языка программирования Kotlin

Время выполнения: 8 часа.

Введение в программирование на *Kotlin*

1.1 Пакеты и импорты

Файл с исходным кодом может начинаться с объявления пакета.

```
package org.example

fun printMessage() { /*...*/ }
class Message { /*...*/ }

// ...
```

Всё содержимое файла с исходниками (например, классы и функции) располагается в объявленном пакете. Таким образом, в приведённом выше примере полное имя функции `printMessage()` будет `org.example.printMessage`, а полное имя класса `Message` - `org.example.Message`.

Если файл не содержит явного объявления пакета, то его содержимое находится в безымянном пакете *по умолчанию*.

1.1.1 Импорт по умолчанию

По умолчанию в каждый файл Kotlin импортируется несколько пакетов:

- kotlin.*
- kotlin.annotation.*
- kotlin.collections.*
- kotlin.comparisons.*
- kotlin.io.*
- kotlin.ranges.*
- kotlin.sequences.*
- kotlin.text.*

1.1.2 Импорт

Помимо импорта по умолчанию каждый файл может содержать свои собственные объявления импорта.

Вы можете импортировать одно имя:

```
import org.example.Message // теперь Message можно использовать б
ез указания пакета
```

Можете импортировать всё доступное содержимое пространства имён (пакет, класс, объект и т.д.):

```
import org.example.* // всё в 'org.example' становится доступно б
ез указания пакета
```

При совпадении имён мы можем разрешить коллизию используя ключевое слово `as` для локального переименования совпадающей сущности.

```
import org.example.Message // Message доступен
import org.test.Message as testMessage // testMessage заменяет им
я 'org.test.Message'
```

Ключевое слово `import` можно использовать не только с классами, но и с другими объявлениями:

- функции и свойства верхнего уровня;
- функции и свойства, объявленные в объявлениях объектов;
- перечислениях.

1.2 Модификаторы доступа

Классы, объекты, интерфейсы, конструкторы, функции, свойства и их сеттеры могут иметь *модификаторы доступа*. Геттеры всегда имеют тут же видимость, что и свойства, к которым они относятся.

В *Kotlin* предусмотрено четыре модификатора доступа: `private`, `protected`, `internal` и `public`. Если явно не использовать никакого модификатора, то по умолчанию применяется `public`.

Функции, свойства, классы, объекты и интерфейсы могут быть объявлены на самом «высоком уровне» прямо внутри пакета.

```
// имя файла: example.kt
package foo

fun baz() { /*...*/ }
class Bar { /*...*/ }
```

- если модификатор доступа не указан, будет использован `public`. Это значит, что весь код данного объявления будет виден из космоса;
- если вы пометите объявление словом `private`, оно будет иметь видимость только внутри файла, где было объявлено;
- если вы используете `internal`, видимость будет распространяться на весь модуль;
- `protected` запрещено использовать в объявлениях «высокого уровня».

Чтобы использовать видимое объявление верхнего уровня из другого пакета, вы должны импортировать его.

Примеры:

```
// имя файла: example.kt
package foo

private fun foo() { /*...*/ } // имеет видимость внутри example.k
t

public var bar: Int = 5       // свойство видно со дна Марианской
впадины
    private set               // сеттер видно только внутри examp
le.kt

internal val baz = 6          // имеет видимость внутри модуля
```

Для членов, объявленных в классе:

- `private` означает видимость только внутри этого класса (включая его члены);
- `protected` – то же самое, что и `private` + видимость в субклассах;
- `internal` – любой клиент *внутри модуля*, который видит объявленный класс, видит и его `internal` члены;
- `public` – любой клиент, который видит объявленный класс, видит его `public` члены.

В *Kotlin* внешний класс не видит `private` члены своих вложенных классов.

Если вы переопределите `protected` или `internal` член и явно не укажете его видимость, переопределённый элемент будет иметь тот же модификатор, что и исходный.

Примеры:

```
open class Outer {
    private val a = 1
    protected open val b = 2
    internal open val c = 3
    val d = 4 // public по умолчанию

    protected class Nested {
        public val e: Int = 5
    }
}

class Subclass : Outer() {
    // a не видно
    // b, c и d видно
    // класс Nested и e видно

    override val b = 5   // b - protected
    override val c = 7   // c - internal
}

class Unrelated(o: Outer) {
```

```
    // o.a и o.b не видно
    // o.c и o.d видно (тот же модуль)
    // Outer.Nested не видно, и Nested::e также не видно
}
```

Для указания видимости основного конструктора класса используется следующий синтаксис:

Вам необходимо добавить ключевое слово `constructor`.

```
class C private constructor(a: Int) { ... }
```

В этом примере конструктор помечен `private`. По умолчанию все конструкторы имеют модификатор доступа `public`, то есть видны везде, где виден сам класс (а вот конструктор `internal` класса видно только в том же модуле).

Локальные переменные, функции и классы не могут иметь модификаторов доступа.

1.3 Функции

В Kotlin функции объявляются с помощью ключевого слова `fun`.

```
fun double(x: Int): Int {
    return 2 * x
}
```

При вызове функции используется традиционный подход:

```
val result = double(2)
```

Для вызова вложенной функции используется знак точки.

```
Stream().read() //создаёт экземпляр класса Stream и вызывает read
()
```

Параметры функций

Параметры функции записываются аналогично системе обозначений в языке `Pascal` - *имя*: *тип*. Параметры разделены запятыми. Каждый параметр должен быть явно указан.

```
fun powerOf(number: Int, exponent: Int): Int { /*...*/ }
```

Вы можете использовать завершающую запятую при объявлении параметров функции.

```
fun powerOf(
    number: Int,
```

```
    exponent: Int, // завершающая запятая
) { /*...*/ }
```

Аргументы по умолчанию

Параметры функции могут иметь значения по умолчанию, которые используются в случае, если аргумент функции не указан при её вызове. Это позволяет снизить уровень перегруженности кода.

```
fun read(
    b: ByteArray,
    off: Int = 0,
    len: Int = b.size,
) { /*...*/ }
```

Значения по умолчанию указываются после типа знаком =.

Переопределённые методы всегда используют те же самые значения по умолчанию, что и их базовые методы. При переопределении методов со значениями по умолчанию в сигнатуре эти параметры должны быть опущены.

```
open class A {
    open fun foo(i: Int = 10) { /*...*/ }
}

class B : A() {
    override fun foo(i: Int) { /*...*/ } // значение по умолчанию
указать нельзя
}
```

Если параметр по умолчанию предшествует параметру без значения по умолчанию, значение по умолчанию можно использовать только при вызове функции с именованными аргументами.

```
fun foo(
    bar: Int = 0,
    baz: Int,
) { /*...*/ }

foo(baz = 1) // Используется значение по умолчанию bar = 0
```

Но если последний аргумент после параметров по умолчанию - лямбда, вы можете передать её либо как именованный аргумент, либо за скобками.

```
fun foo(
    bar: Int = 0,
    baz: Int = 1,
    qux: () -> Unit,
) { /*...*/ }
```

```
foo(1) { println("hello") }     // Используется значение по умолч
анию baz = 1
foo(qux = { println("hello") }) // Используется оба значения по у
молчанию: bar = 0 и baz = 1
foo { println("hello") }        // Используется оба значения по у
молчанию: bar = 0 и baz = 1
```

Именованные аргументы

При вызове функции вы можете явно указать имена одного или нескольких аргументов. Это может быть полезно, когда у функции большой список аргументов, и сложно связать значение с аргументом, особенно если это логическое или `null` значение.

При явном указывании имен аргументов в вызове функции, вы можете свободно изменять порядок их перечисления, и, если вы хотите использовать их значения по умолчанию, вы можете просто пропустить эти аргументы.

Рассмотрим следующую функцию `reformat()`, которая имеет 4 аргумента со значениями по умолчанию:

```
fun reformat(
    str: String,
    normalizeCase: Boolean = true,
    upperCaseFirstLetter: Boolean = true,
    divideByCamelHumps: Boolean = false,
    wordSeparator: Char = ' ',
) { /*...*/ }
```

При её вызове, вам не нужно явно указывать все имена аргументов.

```
reformat(
    "String!",
    false,
    upperCaseFirstLetter = false,
    divideByCamelHumps = true,
    '_'
)
```

Вы можете пропустить все аргументы со значением по умолчанию.

```
reformat("This is a long String!")
```

Вы также можете пропустить не только *все* аргументы со значениями по умолчанию, но и лишь некоторые из них. Однако после первого пропущенного аргумента вы должны указывать имена всех последующих аргументов.

```
reformat("This is a short String!", upperCaseFirstLetter = false,
wordSeparator = '_')
```

Вы можете передать переменное количество аргументов (*vararg*) с именами, используя оператор *spread*.

```
fun foo(vararg strings: String) { /*...*/ }

foo(strings = *arrayOf("a", "b", "c"))
```

В *JVM*: синтаксис именованных аргументов не может быть использован при вызове *Java* функций, потому как байт-код *Java* не всегда сохраняет имена параметров функции.

Функции с возвращаемым типом Unit

Если функция не возвращает никакого полезного значения, её возвращаемый тип – *Unit*. *Unit* – тип только с одним значением – *Unit*. Это значение не нуждается в явном указании возвращения функции.

```
fun printHello(name: String?): Unit {
    if (name != null)
        println("Hello $name")
    else
        println("Hi there!")
    // `return Unit` или `return` необязательны
}
```

Указание типа *Unit* в качестве возвращаемого значения тоже не является обязательным. Код, написанный выше, и следующий код совершенно идентичны:

```
fun printHello(name: String?) { /*...*/ }
```

Функции с одним выражением

Когда функция возвращает одно единственное выражение, фигурные скобки *{ }* могут быть опущены, и тело функции может быть описано после знака =.

```
fun double(x: Int): Int = x * 2
```

Явное объявление возвращаемого типа является необязательным, когда он может быть определен компилятором.

```
fun double(x: Int) = x * 2
```

Явные типы возвращаемых значений

Функции с блочным телом всегда должны иметь явно указанный возвращаемый ими тип данных, если только они не предназначены для возврата `Unit`, тогда указание типа возвращаемого значения необязательно.
Kotlin самостоятельно не вычисляет тип возвращаемого значения для функций с блочным телом, потому что подобные функции могут иметь сложную структуру, и возвращаемый тип будет неочевидным для читающего этот код человека (иногда даже для компилятора).

Нефиксированное число аргументов (varargs)

Параметр функции (обычно для этого используется последний) может быть помечен модификатором `vararg`.

```
fun <T> asList(vararg ts: T): List<T> {
    val result = ArrayList<T>()
    for (t in ts) // ts - это массив (Array)
        result.add(t)
    return result
}
```

Это позволит указать несколько значений в качестве аргументов функции.

```
val list = asList(1, 2, 3)
```

Внутри функции параметр с меткой `vararg` и типом `T` виден как массив элементов `T`, таким образом переменная `ts` в вышеуказанном примере имеет тип `Array<out T>`.

Только один параметр может быть помечен как `vararg`. Если параметр с именем `vararg` не стоит на последнем месте в списке аргументов, значения для последующих параметров могут быть переданы только с использованием синтаксиса именованных аргументов. В случае, если параметр является функцией, для этих целей можно вынести лямбду за фигурные скобки.

При вызове `vararg`-функции вы можете передать аргументы один за другим, например `asList(1, 2, 3)`, или, если у нас уже есть необходимый массив элементов и вы хотите передать его содержимое в функцию, используйте оператор `spread` (необходимо пометить массив знаком `*`).

```
val a = arrayOf(1, 2, 3)
val list = asList(-1, 0, *a, 4)
```

Если вы хотите передать массив примитивного типа в `vararg`, вам необходимо преобразовать его в обычный (типизированный) массив с помощью функции *`toTypedArray()`*.

```
val a = intArrayOf(1, 2, 3) // IntArray - массив примитивного тип
a
val list = asList(-1, 0, *a.toTypedArray(), 4)
```

1.3.1 Область видимости функций

В *Kotlin* функции могут быть объявлены в самом начале файла, что значит, что вам необязательно создавать класс, чтобы воспользоваться его функцией (как в *Java*, *C#* или *Scala*). В дополнение к этому, функции в *Kotlin* могут быть объявлены локально, как функции-члены и функции-расширения.

Локальные функции

Kotlin поддерживает локальные функции, т.е. функции, вложенные в другие функции.

```
fun dfs(graph: Graph) {
    fun dfs(current: Vertex, visited: MutableSet<Vertex>) {
        if (!visited.add(current)) return
        for (v in current.neighbors)
            dfs(v, visited)
    }

    dfs(graph.vertices[0], HashSet())
}
```

Локальная функция может иметь доступ к локальным переменным внешних по отношению к ним функций (типа *closure*). Таким образом, в примере, приведённом выше, *`visited`* может быть локальной переменной.

```
fun dfs(graph: Graph) {
    val visited = HashSet<Vertex>()
    fun dfs(current: Vertex) {
        if (!visited.add(current)) return
        for (v in current.neighbors)
            dfs(v)
    }

    dfs(graph.vertices[0])
}
```

Функции-члены

Функции-члены – это функции, объявленные внутри классов или объектов.

```
class Sample {
    fun foo() { print("Foo") }
}
```

Функции-члены вызываются с использованием точки.

```
Sample().foo() // создаёт инстанс класса Sample и вызывает его фу
нкцию foo
```

1.4 Высокоуровневые функции и лямбды

В *Kotlin* функции являются *функциями первого класса*. Это значит, что они могут храниться в переменных и структурах данных, передаваться в качестве аргументов и возвращаться из других *функций высшего порядка*. Вы можете работать с функциями любым способом, который возможен для других нефункциональных значений.

Чтобы это облегчить, *Kotlin*, как статически типизированный язык программирования, использует семейство функциональных типов для представления функций и предоставляет набор специализированных языковых конструкций, таких как лямбда-выражения.

Функции высшего порядка

Функция высшего порядка – это функция, которая принимает функции как параметры, или возвращает функцию в качестве результата.

Хорошим примером такой функции является идиома функционального программирования `fold` для коллекций, которая принимает начальное значение – `accumulator` вместе с комбинирующей функцией и строит возвращаемое значение, последовательно комбинируя текущее значение `accumulator` с каждым элементом коллекции, заменяя значение `accumulator`.

```
fun <T, R> Collection<T>.fold(
    initial: R,
    combine: (acc: R, nextElement: T) -> R
): R {
    var accumulator: R = initial
    for (element: T in this) {
        accumulator = combine(accumulator, element)
    }
    return accumulator
}
```

В приведённом выше коде параметр `combine` имеет функциональный тип `(R, T) -> R`, поэтому он принимает функцию, которая принимает два

аргумента типа `R` и `T` и возвращает значение типа `R`. Он вызывается внутри цикла `for` и присваивает `accumulator` возвращаемое значение.

Чтобы вызвать `fold`, вы должны передать ему экземпляр функционального типа в качестве аргумента и лямбда-выражение (описание ниже). Лямбда-выражения часто используются в качестве параметра функции высшего порядка.

```
fun main() {
    val items = listOf(1, 2, 3, 4, 5)

    // Лямбда - это блок кода, заключенный в фигурные скобки.
    items.fold(0, {
        // Если у лямбды есть параметры, то они указываются перед
знаком '->'
        acc: Int, i: Int ->
        print("acc = $acc, i = $i, ")
        val result = acc + i
        println("result = $result")
        // Последнее выражение в лямбде считается возвращаемым зн
ачением:
        result
    })

    // Типы параметров в лямбде необязательны, если они могут быт
ь выведены:
    val joinedToString = items.fold("Elements:", { acc, i -> acc
+ " " + i })

    // Ссылки на функции также могут использоваться для вызовов ф
ункций высшего порядка:
    val product = items.fold(1, Int::times)

    println("joinedToString = $joinedToString")
    println("product = $product")
}
```

1.4.1 Функциональные типы

Kotlin использует семейство функциональных типов, таких как `(Int) -> String`, для объявлений, которые являются частью функций: `val onClick: () -> Unit = ....`

Эти типы имеют специальные обозначения, которые соответствуют сигнатурам функций, то есть их параметрам и возвращаемым значениям:

1 У всех функциональных типов есть список с типами параметров, заключенный в скобки, и возвращаемый тип: `(A, B) -> C` обозначает тип, который предоставляет функции два принятых аргумента типа `A` и `B`, а также возвращает значение типа `C`. Список с типами параметров может быть пустым, как, например, в `() -> A`. Возвращаемый тип `Unit` не может быть опущен;

2 У функциональных типов может быть дополнительный тип - *получатель* (ориг.: *receiver*), который указывается в объявлении перед точкой: тип `A.(B) -> C` описывает функции, которые могут быть вызваны для объекта-получателя `A` с параметром `B` и возвращаемым значением `C`. Литералы функций с объектом-приёмником часто используются вместе с этими типами;

3 Останавливаемые функции (ориг.: *suspending functions*) принадлежат к особому виду функциональных типов, у которых в объявлении присутствует модификатор `suspend`, например, `suspend () -> Unit` или `suspend A.(B) -> C`.

Объявление функционального типа также может включать именованные параметры: `(x: Int, y: Int) -> Point`. Именованные параметры могут быть использованы для описания смысла каждого из параметров.

Чтобы указать, что функциональный тип может быть `nullable`, используйте круглые скобки: `((Int, Int) -> Int)?`.

При помощи круглых скобок функциональные типы можно объединять: `(Int) -> ((Int) -> Unit)`.

Стрелка в объявлении является правоассоциативной (ориг.: *right-associative*), т.е. объявление `(Int) -> (Int) -> Unit` эквивалентно объявлению из предыдущего примера, а не `((Int) -> (Int)) -> Unit`.

Вы также можете присвоить функциональному типу альтернативное имя, используя псевдонимы типов.

```
typealias ClickHandler = (Button, ClickEvent) -> Unit
```

Создание функционального типа

Существует несколько способов получить экземпляр функционального типа:

1 Используя блок с кодом внутри функционального литерала в одной из форм:

– лямбда-выражение: `{ a, b -> a + b }`,

– анонимная функция: `fun(s: String): Int { return s.toIntOrNull() ?: 0 }`

– Литералы функций с объектом-приёмником могут использоваться как значения функциональных типов с получателем.

2 Используя вызываемую ссылку на существующее объявление:

– функции верхнего уровня, локальной функции, функции-члена или функции-расширения: `::isOdd, String::toInt`,

– свойства верхнего уровня, члена или свойства-расширения: `List<Int>::size`,

– конструктора: `::Regex`

– К ним относятся привязанные вызываемые ссылки, которые указывают на член конкретного экземпляра: `foo::toString`.

3 Используя экземпляр пользовательского класса, который реализует функциональный тип в качестве интерфейса:

```
class IntTransformer: (Int) -> Int {
    override operator fun invoke(x: Int): Int = TODO()
}

val intFunction: (Int) -> Int = IntTransformer()
```

При достаточной информации компилятор может самостоятельно вывести функциональный тип для переменной.

```
val a = { i: Int -> i + 1 } // Выведенный тип - (Int) -> Int
```

Небуквальные (ориг.: *non-literal*) значения функциональных типов с и без получателя являются взаимозаменяемыми, поэтому получатель может заменить первый параметр, и наоборот. Например, значение типа `(A, B) -> C` может быть передано или назначено там, где ожидается `A.(B) -> C`, и наоборот.

```
fun main() {
    val repeatFun: String.(Int) -> String = { times -> this.repea
t(times) }
    val twoParameters: (String, Int) -> String = repeatFun // OK

    fun runTransformation(f: (String, Int) -> String): String {
        return f("hello", 3)
    }
    val result = runTransformation(repeatFun) // OK

    println("result = $result")
}
```

Обратите внимание, что функциональный тип без получателя выводится по умолчанию, даже если переменная инициализируется со ссылкой на функцию-расширение. Чтобы это изменить, укажите тип переменной явно.

Вызов экземпляра функционального типа

Значение функционального типа может быть вызвано с помощью оператора `invoke(...)`: `f.invoke(x)` или просто `f(x)`.

Если значение имеет тип получателя, то объект-приёмник должен быть передан в качестве первого аргумента. Другой способ вызвать значение

функционального типа с получателем - это добавить его к объекту-приёмнику, как если бы это была функция-расширение: `1.foo(2)`.

Пример:

```
fun main() {
    val stringPlus: (String, String) -> String = String::plus
    val intPlus: Int.(Int) -> Int = Int::plus

    println(stringPlus.invoke("<-", "->"))
    println(stringPlus("Hello, ", "world!"))

    println(intPlus.invoke(1, 1))
    println(intPlus(1, 2))
    println(2.intPlus(3)) // вызывается как функция-расширение
}
```

1.4.2 Лямбда-выражения и анонимные функции

Лямбда-выражения и анонимные функции – это «функциональный литерал», то есть необъявленная функция, которая немедленно используется в качестве выражения. Рассмотрим следующий пример:

```
max(strings, { a, b -> a.length < b.length })
```

Функция `max` является функцией высшего порядка, потому что она принимает функцию в качестве второго аргумента. Этот второй аргумент является выражением, которое в свою очередь есть функция, то есть *функциональный литерал*. Как функция он эквивалентен объявлению:

```
fun compare(a: String, b: String): Boolean = a.length < b.length
```

Синтаксис лямбда-выражений

Полная синтаксическая форма лямбда-выражений может быть представлена следующим образом:

```
val sum: (Int, Int) -> Int = { x: Int, y: Int -> x + y }
```

- лямбда-выражение всегда заключено в скобки `{...}`;
- объявление параметров при таком синтаксисе происходит внутри этих скобок и может включать в себя аннотации типов;
- тело функции начинается после знака `->`;
- если тип возвращаемого значения не `Unit`, то в качестве возвращаемого типа принимается последнее (а возможно и единственное) выражение внутри тела лямбды.

Если вы вынесите все необязательные объявления, то, что останется, будет выглядеть следующим образом:

```
val sum = { x: Int, y: Int -> x + y }
```

Передача лямбды в качестве последнего параметра

В *Kotlin* существует соглашение: если последний параметр функции является функцией, то лямбда-выражение, переданное в качестве соответствующего аргумента, может быть вынесено за круглые скобки.

```
val product = items.fold(1) { acc, e -> acc * e }
```

Такой синтаксис также известен как *trailing lambda*.

Когда лямбда-выражение является единственным аргументом функции, круглые скобки могут быть опущены.

```
run { println(...") }
```

it: неявное имя единственного параметра

Очень часто лямбда-выражение имеет только один параметр.

Если компилятор способен самостоятельно определить сигнатуру, то объявление параметра можно опустить вместе с `->`. Параметр будет неявно объявлен под именем `it`.

```
ints.filter { it > 0 } // этот литерал имеет тип '(it: Int) -> Bo
olean'
```

Возвращение значения из лямбда-выражения

Вы можете вернуть значение из лямбды явно, используя оператор return. Либо неявно будет возвращено значение последнего выражения.
Таким образом, два следующих фрагмента равнозначны:

```
ints.filter {
    val shouldFilter = it > 0
    shouldFilter
}

ints.filter {
    val shouldFilter = it > 0
    return@filter shouldFilter
}
```

Это соглашение, вместе с передачей лямбда-выражения вне скобок, позволяет писать код в стиле LINQ.

```
strings.filter { it.length == 5 }.sortedBy { it }.map { it.upperc
ase() }
```

Символ подчеркивания для неиспользуемых переменных

Если параметр лямбды не используется, то разрешено его имя заменить на символ подчёркивания.

```
map.forEach { _, value -> println("$value!") }
```

Анонимные функции

Единственной особенностью синтаксиса лямбда-выражений, о которой ещё не было сказано, является способность определять и назначать возвращаемый функцией тип. В большинстве случаев в этом нет особой необходимости, потому что он может быть вычислен автоматически. Однако, если у вас есть потребность в определении возвращаемого типа, вы можете воспользоваться альтернативным синтаксисом: *анонимной функцией*.

```
fun(x: Int, y: Int): Int = x + y
```

Объявление анонимной функции выглядит очень похоже на обычное объявление функции, за исключением того, что её имя опущено. Тело такой функции может быть описано и выражением (как показано выше), и блоком.

```
fun(x: Int, y: Int): Int {
    return x + y
}
```

Параметры функции и возвращаемый тип обозначаются таким же образом, как в обычных функциях, за исключением того, что тип параметра может быть опущен, если его значение следует из контекста.

```
ints.filter(fun(item) = item > 0)
```

Аналогично и с типом возвращаемого значения: он вычисляется автоматически для функций-выражений или же должен быть явно определён (если не является типом `Unit`) для анонимных функций с блоком в качестве тела.

Обратите внимание, что параметры анонимных функций всегда заключены в круглые скобки `(...)`. Приём, позволяющий оставлять параметры вне скобок, работает только с лямбда-выражениями.

Одним из отличий лямбда-выражений от анонимных функций является поведение оператора `return` (non-local returns). Слово `return`, не имеющее метки (`@`), всегда возвращается из функции, объявленной ключевым словом

`fun`. Это означает, что `return` внутри лямбда-выражения возвратит выполнение к функции, включающей в себя это лямбда-выражение. Внутри анонимных функций оператор `return`, в свою очередь, выйдет, собственно, из анонимной функции.

Замыкания

Лямбда-выражение или анонимная функция (так же, как и локальная функция или анонимные объекты) имеет доступ к своему *замыканию*, то есть к переменным, объявленным вне этого выражения или функции. Переменные, захваченные в замыкании, могут быть изменены в лямбде.

```
var sum = 0
ints.filter { it > 0 }.forEach {
    sum += it
}
print(sum)
```

1.5 Классы

Классы в *Kotlin* объявляются с помощью использования ключевого слова `class`.

```
class Person { /*...*/ }
```

Объявление класса состоит из имени класса, заголовка (указания типов его параметров, основного конструктора и т.п) и тела класса, заключённого в фигурные скобки. И заголовок, и тело класса являются необязательными составляющими. Если у класса нет тела, фигурные скобки могут быть опущены.

```
class Empty
```

1.5.1 Конструкторы

Класс в *Kotlin* может иметь *основной конструктор* (*primary constructor*) и один или более *дополнительных конструкторов* (*secondary constructors*). Основной конструктор является частью заголовка класса, его объявление идёт сразу после имени класса (и необязательных параметров).

```
class Person constructor(firstName: String) { /*...*/ }
```

Если у основного конструктора нет аннотаций и модификаторов видимости, ключевое слово `constructor` может быть опущено.

```
class Person(firstName: String) { /*...*/ }
```

Основной конструктор не может содержать в себе исполняемого кода. Инициализирующий код может быть помещён в соответствующие блоки (*initializers blocks*), которые помечаются словом `init`.

При создании экземпляра класса блоки инициализации выполняются в том порядке, в котором они идут в теле класса, чередуясь с инициализацией свойств.

```
class InitOrderDemo(name: String) {
    val firstProperty = "Первое свойство: $name".also(::println)

    init {
        println("Первый блок инициализации: ${name}")
    }

    val secondProperty = "Второе свойство: ${name.length}".also(:
:println)

    init {
        println("Второй блок инициализации: ${name.length}")
    }
}
```

Обратите внимание, что параметры основного конструктора могут быть использованы в инициализирующем блоке. Они также могут быть использованы при инициализации свойств в теле класса.

```
class Customer(name: String) {
    val customerKey = name.uppercase()
}
```

Для объявления и инициализации свойств основного конструктора в *Kotlin* есть лаконичное синтаксическое решение:

```
class Person(val firstName: String, val lastName: String, var age
: Int)
```

Такие объявления также могут включать в себя значения свойств класса по умолчанию.

```
class Person(val firstName: String, val lastName: String, var isE
mployed: Boolean = true)
```

Вы можете использовать завершающую запятую при объявлении свойств класса.

```
class Person(
    val firstName: String,
```

```
    val lastName: String,
    var age: Int, // завершающая запятая
) { /*...*/ }
```

Свойства, объявленные в основном конструкторе, могут быть изменяемые (*var*) и неизменяемые (*val*).

Если у конструктора есть аннотации или модификаторы видимости, ключевое слово *constructor* обязательно, и модификаторы используются перед ним.

```
class Customer public @Inject constructor(name: String) { /*...*/
}
```

Дополнительные конструкторы

В классах также могут быть объявлены *дополнительные конструкторы* (*secondary constructors*), перед которыми используется ключевое слово *constructor*.

```
class Person(val pets: MutableList<Pet> = mutableListOf())
class Pet {
    constructor(owner: Person) {
        owner.pets.add(this) // добавляет этого питомца в список
домашних животных своего владельца
    }
}
```

Если у класса есть основной конструктор, каждый дополнительный конструктор должен прямо или косвенно ссылаться (через другой(*ие*) конструктор(*ы*)) на основной. Осуществляется это при помощи ключевого слова *this*.

```
class Person(val name: String) {
    val children: MutableList<Person> = mutableListOf()
    constructor(name: String, parent: Person) : this(name) {
        parent.children.add(this)
    }
}
```

Обратите внимание, что код в блоках инициализации фактически становится частью основного конструктора. Дополнительный конструктор ссылается на основной при помощи своего первого оператора, поэтому код во всех блоках инициализации, а также инициализация свойств выполняется перед выполнением кода в теле дополнительного конструктора.

Даже если у класса нет основного конструктора на него все равно происходит неявная ссылка и блоки инициализации выполняются также.

```
class Constructors {
    init {
        println("Блок инициализации")
```

```
    }
    constructor(i: Int) {
        println("Constructor $i")
    }
}
```

Если в абстрактном классе не объявлено никаких конструкторов (основного или дополнительных), у этого класса автоматически сгенерируется пустой конструктор без параметров. Видимость этого конструктора будет *public*.

Если вы не желаете иметь класс с открытым *public* конструктором, вам необходимо объявить пустой конструктор с соответствующим модификатором видимости.

```
class DontCreateMe private constructor () { /*...*/ }
```

В *JVM* компилятор генерирует дополнительный конструктор без параметров в случае, если все параметры основного конструктора имеют значения по умолчанию. Это делает использование таких библиотек, как *Jackson* и *JPA*, более простым с *Kotlin*, так как они используют пустые конструкторы при создании экземпляров классов.

```
class Customer(val customerName: String = "")
```

1.5.2 Создание экземпляров классов

Для создания экземпляра класса конструктор вызывается так, как если бы он был обычной функцией.

```
val invoice = Invoice()

val customer = Customer("Joe Smith")
```

В *Kotlin* нет ключевого слова `new`.

1.5.3 Члены класса

Классы могут содержать в себе:

- конструкторы и инициализирующие блоки
- функции
- свойства
- вложенные классы
- объявления объектов

1.5.4 Абстрактные классы

Класс может быть объявлен как `abstract` со всеми или некоторыми его членами. Абстрактный член не имеет реализации в своём классе. Обратите внимание, что нам не надо аннотировать абстрактный класс или функцию словом `open` – это и так подразумевается.

```
abstract class Polygon {
    abstract fun draw()
}
class Rectangle : Polygon() {
    override fun draw() {
        // рисование прямоугольника
    }
}
```

Можно переопределить неабстрактный `open` член абстрактным.

```
open class Polygon {
    open fun draw() {
        // некоторый метод рисования полигонов по умолчанию
    }
}
abstract class WildShape : Polygon() {
    // Классы, которые наследуют WildShape, должны предоставлять
свой собственный
    // метод рисования вместо использования по умолчанию для поли
гона
    abstract override fun draw()
}
```

1.5.5 Вспомогательные объекты

Если вам нужно написать функцию, которая может быть использована без создания экземпляра класса, имеющую доступ к данным внутри этого класса (к примеру, фабричный метод), вы можете написать её как член объявления объекта внутри этого класса.

В частности, если вы объявляете вспомогательный объект в своём классе, у вас появляется возможность обращаться к членам класса, используя только название класса в качестве классификатора.

1.6 Наследование

Для всех классов в *Kotlin* родительским суперклассом является класс `Any`. Он также является родительским классом для любого класса, в котором не указан какой-либо другой родительский класс.

```
class Example // Неявно наследуется от Any
```

У *Any* есть три метода: *equals()*, *hashCode()* и *toString()*. Эти методы определены для всех классов в Kotlin.

По умолчанию все классы в *Kotlin* имеют статус *final*, который блокирует возможность наследования. Чтобы сделать класс наследуемым, его нужно пометить ключевым словом *open*.

```
open class Base // Класс открыт для наследования
```

Для явного объявления суперкласса мы помещаем его имя за знаком двоеточия в оглавлении класса:

```
open class Base(p: Int)

class Derived(p: Int) : Base(p)
```

Если у класса есть основной конструктор, базовый тип может (и должен) быть проинициализирован там же, с использованием параметров основного конструктора.

Если у класса нет основного конструктора, тогда каждый последующий дополнительный конструктор должен включать в себя инициализацию базового типа с помощью ключевого слова *super* или давать отсылку на другой конструктор, который это делает. Примечательно, что любые дополнительные конструкторы могут ссылаться на разные конструкторы базового типа.

```
class MyView : View {
    constructor(ctx: Context) : super(ctx)
    constructor(ctx: Context, attrs: AttributeSet) : super(ctx, a
ttrs)
}
```

1.6.1 Переопределение методов класса

Kotlin требует явно указывать модификаторы и для членов, которые могут быть переопределены, и для самого переопределения.

```
open class Shape {
    open fun draw() { /*...*/ }
    fun fill() { /*...*/ }
}

class Circle() : Shape() {
    override fun draw() { /*...*/ }
}
```

Для *Circle.draw()* необходим модификатор *override*. В случае её отсутствия компилятор выдаст ошибку. Если у функции типа *Shape.fill()* нет модификатора *open*, объявление метода с такой же сигнатурой в

производном классе невозможно, с `override` или без. Модификатор `open` не действует при добавлении к членам *final* класса (т.е. класса без модификатора `open`).

Член класса, помеченный `override`, является сам по себе open, т.е. он может быть переопределён в производных классах. Если вы хотите запретить возможность переопределения такого члена, используйте `final`.

```
open class Rectangle() : Shape() {
    final override fun draw() { /*...*/ }
}
```

1.6.2 Переопределение свойств класса

Переопределение свойств работает также, как и переопределение методов; все свойства, унаследованные от суперкласса, должны быть помечены ключевым словом `override`, а также должны иметь совместимый тип. Каждое объявленное свойство может быть переопределено свойством с инициализацией или свойством с `get`-методом.

```
open class Shape {
    open val vertexCount: Int = 0
}

class Rectangle : Shape() {
    override val vertexCount = 4
}
```

Вы также можете переопределить свойство `val` свойством `var`, но не наоборот. Это разрешено, поскольку свойство `val` объявляет `get`-метод, а при переопределении его как `var` дополнительно объявляется `set`-метод в производном классе.

Обратите внимание, что ключевое слово `override` может быть использовано в основном конструкторе класса как часть объявления свойства.

```
interface Shape {
    val vertexCount: Int
}

class Rectangle(override val vertexCount: Int = 4) : Shape // Все
гда имеет 4 вершины

class Polygon : Shape {
    override var vertexCount: Int = 0  // Может быть установлено
любое количество
}
```

1.6.3 Порядок инициализации производного класса

При создании нового экземпляра класса в первую очередь выполняется инициализация базового класса (этому шагу предшествует только оценка

аргументов, передаваемых в конструктор базового класса) и, таким образом, происходит до запуска логики инициализации производного класса.

```
open class Base(val name: String) {

    init { println("Инициализация класса Base") }

    open val size: Int =
        name.length.also { println("Инициализация свойства size в
класса Base: $it") }
}

class Derived(
    name: String,
    val lastName: String,
) : Base(name.replaceFirstChar { it.uppercase() }.also { println(
"Аргументы, переданные в конструктор класса Base: $it") }) {

    init { println("Инициализация класса Derived") }

    override val size: Int =
        (super.size + lastName.length).also { println("Инициализа
ция свойства size в классе Derived: $it") }
}

fun main() {
    println("Построение класса Derived(\"hello\", \"world\")")
    Derived("hello", "world")
}
```

Это означает, что свойства, объявленные или переопределенные в производном классе, не инициализированы к моменту вызова конструктора базового класса. Если какое-либо из этих свойств используется в логике инициализации базового класса (прямо или косвенно через другую переопределенную `open` реализацию члена класса), это может привести к некорректному поведению или сбою во время выполнения. Поэтому при разработке базового класса следует избегать использования членов с ключевым словом `open` в конструкторах, инициализации свойств и блоков инициализации (`init`).

Вызов функций и свойств суперкласса

Производный класс может вызывать реализацию функций и свойств своего суперкласса, используя ключевое слово `super`.

```
open class Rectangle {
    open fun draw() { println("Рисование прямоугольника") }
    val borderColor: String get() = "black"
}
```

```
class FilledRectangle : Rectangle() {
    override fun draw() {
        super.draw()
        println("Заполнение прямоугольника")
    }

    val fillColor: String get() = super.borderColor
}
```

Во внутреннем классе доступ к суперклассу внешнего класса осуществляется при помощи ключевого слова `super`, за которым следует имя внешнего класса: `super@Outer`.

```
class FilledRectangle: Rectangle() {
    override fun draw() {
        val filler = Filler()
        filler.drawAndFill()
    }

    inner class Filler {
        fun fill() { println("Filling") }
        fun drawAndFill() {
            super@FilledRectangle.draw() // Вызывает реализацию ф
ункции draw() класса Rectangle
            fill()
            println("Нарисованный прямоугольник заполнен ${super@
FilledRectangle.borderColor} цветом") // Используется реализация
get()-метода свойства borderColor в классе
        }
    }
}
```

1.6.4 Правила переопределения

В *Kotlin* правила наследования реализации определены следующим образом: если класс наследует многочисленные реализации одного и того члена от ближайших родительских классов, он должен переопределить этот член и обеспечить свою собственную реализацию (возможно, используя одну из унаследованных).

Для того чтобы отметить конкретный супертип (родительский класс), от которого мы наследуем данную реализацию, используйте ключевое слово `super`. Для задания имени родительского супертипа используются треугольные скобки, например `super<Base>`.

```
open class Rectangle {
    open fun draw() { /* ... */ }
}

interface Polygon {
```

```
    fun draw() { /* ... */ } // члены интерфейса открыты ('open')
по умолчанию
}

class Square() : Rectangle(), Polygon {
    // Компилятор требует, чтобы функция draw() была переопределе
на:
    override fun draw() {
        super<Rectangle>.draw() // вызов Rectangle.draw()
        super<Polygon>.draw() // вызов Polygon.draw()
    }
}
```

Это нормально, наследоваться одновременно от `Rectangle` и `Polygon`, но так как у каждого из них есть своя реализация функции `draw()`, мы должны переопределить `draw()` в `Square` и обеспечить нашу собственную реализацию этого метода для устранения получившейся неоднозначности.

1.7 Свойства

Свойства в классах *Kotlin* могут быть объявлены либо как изменяемые (*mutable*) и неизменяемые (*read-only*) – `var` и `val` соответственно.

```
class Address {
    var name: String = "Holmes, Sherlock"
    var street: String = "Baker"
    var city: String = "London"
    var state: String? = null
    var zip: String = "123456"
}
```

Для того чтобы воспользоваться свойством, просто обратитесь к нему по имени.

```
fun copyAddress(address: Address): Address {
    val result = Address() // в Kotlin нет никакого слова `new`
    result.name = address.name // вызов методов доступа
    result.street = address.street
    // ...
    return result
}
```

1.7.1 Геттеры и сеттеры

Полный синтаксис объявления свойства выглядит так:

```
var <propertyName>[: <PropertyType>] [= <property_initializer>]
    [<getter>]
    [<setter>]
```

Инициализатор *property_initializer*, геттер и сеттер можно не указывать. Также необязательно указывать тип свойства, если он может быть выведен из инициализатора или из возвращаемого типа геттера.

```
var initialized = 1 // имеет тип Int, стандартный геттер и сеттер
// var allByDefault // ошибка: необходима явная инициализация,
                    // предусмотрены стандартные геттер и сеттер
```

Синтаксис объявления констант имеет два отличия от синтаксиса объявления изменяемых переменных: во-первых, объявление константы начинается с ключевого слова *val* вместо *var*, а во-вторых, объявление сеттера запрещено.

```
val simple: Int? // имеет тип Int, стандартный геттер,
                 // должен быть инициализирован в конструкторе
val inferredType = 1 // имеет тип Int и стандартный геттер
```

Вы можете самостоятельно определить методы доступа для свойства. Если вы определяете пользовательский геттер, он будет вызываться каждый раз, когда вы обращаетесь к свойству (таким образом, вы можете реализовать вычисляемое свойство). Вот пример пользовательского геттера:

```
class Rectangle(val width: Int, val height: Int) {
    val area: Int
        get() = this.width * this.height // тип свойства необязат
елен, поскольку он может быть выведен из возвращаемого типа гетте
ра
}
```

Вы можете опустить тип свойства, если его можно определить с помощью геттера.

```
val area get() = this.width * this.height
```

Если вы определяете пользовательский сеттер, он будет вызываться каждый раз, когда вы присваиваете значение свойству, за исключением его инициализации. Пользовательский сеттер выглядит так:

```
var stringRepresentation: String
    get() = this.toString()
    set(value) {
        setDataFromString(value) // парсит строку и устанавливает
                                 // значения для других свойств
    }
```

По договорённости, имя параметра сеттера – *value*, но вы можете использовать любое другое.

Если вам нужно изменить область видимости метода доступа или пометить его аннотацией, при этом не внося изменения в реализацию по умолчанию, вы можете объявить метод доступа без объявления его тела.

```
var setterVisibility: String = "abc"
    private set // сеттер имеет private доступ и стандартную реал
изацию

var setterWithAnnotation: Any? = null
    @Inject set // аннотирование сеттера с помощью Inject
```

Теневые поля

В *Kotlin* поле используется только как часть свойства для хранения его значения в памяти. Поля не могут быть объявлены напрямую. Однако, когда свойству требуется теневое поле (*backing field*), *Kotlin* предоставляет его автоматически. На это теневое поле можно обратиться в методах доступа, используя идентификатор `field`:

```
var counter = 0 // инициализатор назначает резервное поле напряму
ю
    set(value) {
        if (value >= 0)
            field = value // значение при инициализации записывае
тся
                          // прямиком в backing field

            // counter = value // ERROR StackOverflow: Использова
ние 'counter' сделало бы сеттер рекурсивным
    }
```

Идентификатор `field` может быть использован только в методах доступа к свойству.

Теневое поле будет сгенерировано для свойства, если оно использует стандартную реализацию как минимум одного из методов доступа, либо если пользовательский метод доступа ссылается на него через идентификатор `field`.

Например, в примере ниже не будет никакого теневого поля:

```
val isEmpty: Boolean
    get() = this.size == 0
```

Теневые свойства

Если вы хотите предпринять что-то такое, что выходит за рамки вышеуказанной схемы *неявного теневого поля*, вы всегда можете использовать *теневое свойство* (`backing property`).

```
private var _table: Map<String, Int>? = null
```

```
public val table: Map<String, Int>
    get() {
        if (_table == null) {
            _table = HashMap() // параметры типа вычисляются авто
матически
                               // (ориг.: "Type parameters are in
ferred")
        }
        return _table ?: throw AssertionError("Set to null by ano
ther thread")
    }
```

В *JVM*: доступ к приватным свойствам со стандартными геттерами и сеттерами оптимизируется таким образом, что вызов функции не происходит.

1.7.2 Константы времени компиляции

Если значение константного (*read-only*) свойства известно во время компиляции, пометьте его как *константы времени компиляции*, используя модификатор `const`. Такие свойства должны соответствовать следующим требованиям:

- Находиться на самом высоком уровне или быть членами `объявления object` или вспомогательного объекта;
- Быть проинициализированными значением типа `String` или значением примитивного типа;
- Не иметь переопределённого геттера.

Такие свойства могут быть использованы в аннотациях.

```
const val SUBSYSTEM_DEPRECATED: String = "This subsystem is depre
cated"

@Deprecated(SUBSYSTEM_DEPRECATED) fun foo() { ... }
```

1.7.3 Свойства и переменные с поздней инициализацией

Обычно, свойства, объявленные *non-null* типом, должны быть проинициализированы в конструкторе. Однако часто бывает так, что делать это неудобно. К примеру, свойства могут быть инициализированы через внедрение зависимостей или в установочном методе (ориг.: *setup method*) юнит-теста. В таком случае вы не можете обеспечить *non-null* инициализацию в конструкторе, но всё равно хотите избежать проверок на *null* при обращении внутри тела класса к такому свойству.

Для того чтобы справиться с такой задачей, вы можете пометить свойство модификатором `lateinit`.

```
public class MyTest {
    lateinit var subject: TestSubject

    @SetUp fun setup() {
```

```
        subject = TestSubject()
    }

    @Test fun test() {
        subject.method() // объект инициализирован, проверять на
null не нужно
    }
}
```

Такой модификатор может быть использован только с `var` свойствами, объявленными внутри тела класса (не в основном конструкторе, и только тогда, когда свойство не имеет пользовательских геттеров и сеттеров), со свойствами верхнего уровня и локальными переменными. Тип такого свойства должен быть *non-null* и не должен быть примитивным.

Доступ к `lateinit` свойству до того, как оно проинициализировано, выбрасывает специальное исключение, которое чётко обозначает свойство, к которому осуществляется доступ, и тот факт, что оно не было инициализировано.

Проверка инициализации lateinit var

Чтобы проверить, было ли проинициализировано `lateinit var` свойство, используйте `.isInitialized` метод ссылки на это свойство.

```
if (foo::bar.isInitialized) {
    println(foo.bar)
}
```

Эта проверка возможна только для лексически доступных свойств, то есть объявленных в том же типе, или в одном из внешних типов, или глобальных свойств, объявленных в том же файле.

1.8 Ключевое слово this

Чтобы сослаться на объект, с которым вы работаете, используется ключевое слово `this`:

- внутри класса ключевое слово `this` ссылается на объект этого класса;
- в функциях-расширениях или в литерале функции с объектом-приёмником `this` обозначает *объект-приёмник*, который передаётся слева от точки.

Если ключевое слово `this` не имеет определителей, то оно ссылается на *область самого глубокого замыкания*. Чтобы сослаться на `this` в одной из внешних областей, используются *метки-определители*.

1.8.1 this с определителем

Чтобы получить доступ к `this` из внешней области (класса, функции-расширения или именованных литералов функций с объектом-приёмником), используйте `this@label`, где `@label` - это метка области, из которой нужно получить `this`.

```
class A { // неявная метка @A
    inner class B { // неявная метка @B
        fun Int.foo() { // неявная метка @foo
            val a = this@A // this из A
            val b = this@B // this из B

            val c = this // объект-приёмник функции foo(), типа I
nt
            val c1 = this@foo // объект-приёмник функции foo(), т
ипа Int

            val funLit = lambda@ fun String.() {
                val d = this // объект-приёмник литерала funLit
            }


            val funLit2 = { s: String ->
                // объект-приёмник функции foo(), т.к. замыкание
лямбды не имеет объекта-приёмника
                val d1 = this
            }
        }
    }
}
```

1.8.2 Подразумеваемое this

Когда вы вызываете функцию-член для `this`, вы можете пропустить `this` часть. Если у вас есть функция, не являющаяся членом, с тем же именем, используйте ее с осторожностью, потому что в некоторых случаях может быть вызвана она.

```
fun printLine() { println("Функция верхнего уровня") }

class A {
    fun printLine() { println("Функция-член") }

    fun invokePrintLine(omitThis: Boolean = false)  {
        if (omitThis) printLine()
        else this.printLine()
    }
}

A().invokePrintLine() // Функция-член
```

```
A().invokePrintLine(omitThis = true) // Функция верхнего уровня
```

1.9 Коллекции. Общий обзор

Стандартная библиотека *Kotlin* предоставляет большой набор инструментов для работы с *коллекциями* – группами с переменным количеством элементов (или нулём элементов), которые используются для решения какой-либо задачи.

Коллекции – это общая концепция для большинства языков программирования.

Обычно в коллекции находится несколько объектов одного типа (но также коллекция может быть пустой). Эти объекты называются *элементами* или *items*. Например, все студенты одного факультета образуют коллекцию, которую можно использовать для расчёта их среднего возраста.

Типы коллекций в *Kotlin*:

1 *List* (список) – упорядоченная коллекция, в которой к элементам можно обращаться по индексам – целым числам, отражающим положение элементов в коллекции. Идентичные элементы (дубликаты) могут встречаться в списке более одного раза. Примером списка является предложение: это группа слов, их порядок важен, и они могут повторяться.

2 *Set* (множество) – коллекция уникальных элементов. Отражает математическую абстракцию множества: группа объектов без повторов. Как правило, порядок расположения элементов здесь не имеет значения. Примером множества является алфавит.

3 *Map* (словарь, ассоциативный список) – набор из пар «ключ-значение». Ключи уникальны и каждый из них соответствует ровно одному значению. Значения могут иметь дубликаты. Ассоциативные списки полезны для хранения логических связей между объектами, например, *ID* сотрудников и их должностей.

Kotlin позволяет управлять коллекциями независимо от того, какой именно тип объектов в них хранится: будь то `String`, `Int` или какой-то собственный класс, общие принципы работы с коллекцией всегда неизменны. Стандартная библиотека *Kotlin* предоставляет общие интерфейсы, классы и функции для создания, заполнения и управления коллекциями любого типа.

Интерфейсы коллекций и связанные с ними функции находятся в пакете `kotlin.collections`.

1.9.1 Типы коллекций

Стандартная библиотека *Kotlin* предоставляет реализации для основных типов коллекций: `Set`, `List`, `Map`. Есть два вида интерфейсов, предоставляющих каждый из этих типов:

- *неизменяемый* (*read-only*) – предоставляет операции, которые дают доступ к элементам коллекции.

– *изменяемый* (*mutable*) – расширяет предыдущий интерфейс и дополнительно даёт доступ к операциям добавления, удаления и обновления элементов коллекции.

Обратите внимание, что изменяемую коллекцию не требуется объявлять с помощью ключевого слова `var`. Связано это с тем, что изменения вносятся в изначальные объекты коллекции без изменения ссылки на саму коллекцию. Но если вы объявите коллекцию с помощью `val` и попытаетесь ее перезаписать, то получите ошибку компиляции.

```
fun main() {
    val numbers = mutableListOf("one", "two", "three", "four")
    numbers.add("five") // this is OK
    //numbers = mutableListOf("six", "seven") // compilation erro
r
}
```

Ниже на рисунке 1 представлена иерархия интерфейсов коллекций *Kotlin*:

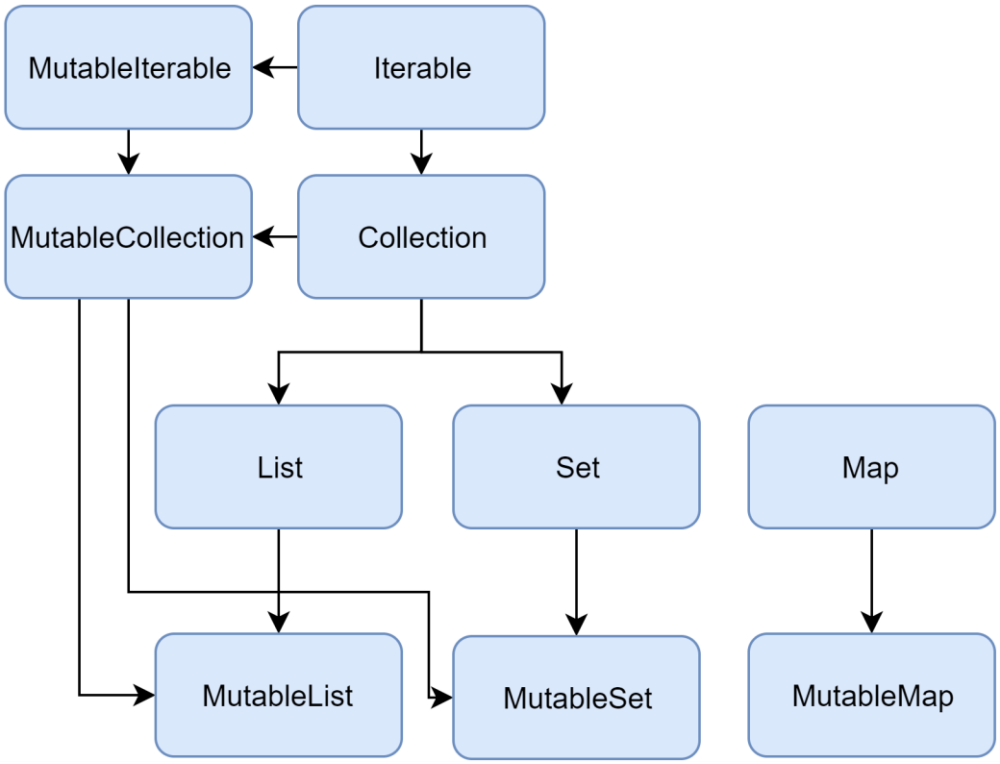

Рисунок 1 – Иерархия коллекций

Неизменяемые типы коллекций ковариантны. Это означает, что если класс `Rectangle` наследуется от `Shape`, вы можете использовать `List<Rectangle>` там, где требуется `List<Shape>`. Другими словами, типы коллекций имеют такое же отношение подтипов, что и типы элементов. `Map`-ы ковариантны по типу значения, но не по типу ключа.

В свою очередь, изменяемые коллекции не являются ковариантными; в противном случае это привело бы к сбоям во время выполнения. Если `MutableList<Rectangle>` был подтипом `MutableList<Shape>`, вы могли добавить в него других наследников `Shape` (например, `Circle`), таким образом нарушая изначальный тип коллекции - `Rectangle`.

Collection

`Collection<T>` является корнем в иерархии коллекций. Этот интерфейс представляет собой обычное поведение неизменяемой коллекции: операции типа `size`, `get` и т. д. `Collection` наследуется от интерфейса `Iterable<T>`, который определяет операции для итерации элементов. Вы можете использовать `Collection` как параметр функции, которая может работать с разными типами коллекций. Для более конкретных случаев следует использовать наследников `Collection`: `List` и `Set`.

```
fun printAll(strings: Collection<String>) {
        for(s in strings) print("$s ")
        println()
    }

fun main() {
    val stringList = listOf("one", "two", "one")
    printAll(stringList) // one two one

    val stringSet = setOf("one", "two", "three")
    printAll(stringSet) // one two three
}
MutableCollection<T> - это Collection с операциями записи, такими
как add и remove.
fun List<String>.getShortWordsTo(shortWords: MutableList<String>,
maxLength: Int) {
    this.filterTo(shortWords) { it.length <= maxLength }
    // throwing away the articles
    val articles = setOf("a", "A", "an", "An", "the", "The")
    shortWords -= articles
}

fun main() {
    val words = "A long time ago in a galaxy far far away".split(
" ")
    val shortWords = mutableListOf<String>()
    words.getShortWordsTo(shortWords, 3)
    println(shortWords) // [ago, in, far, far]
}
```

List

`List<T>` хранит элементы в определённом порядке и обеспечивает к ним доступ по индексу. Индексы начинаются с нуля (0 – индекс первого элемента) и идут до `lastIndex`, который равен `(list.size - 1)`.

```
fun main() {
    val numbers = listOf("one", "two", "three", "four")
    println("Number of elements: ${numbers.size}") // 4
    println("Third element: ${numbers.get(2)}") // three
    println("Fourth element: ${numbers[3]}") // four
    println("Index of element \"two\" ${numbers.indexOf("two")}")
// 1
}
```

Элементы списка (в том числе `null`) могут дублироваться: список может содержать любое количество одинаковых объектов. Два списка считаются равными, если они имеют одинаковый размер и их элементы в одних и тех позициях структурно равны.

```
data class Person(var name: String, var age: Int)

fun main() {
    val bob = Person("Bob", 31)
    val people = listOf(Person("Adam", 20), bob, bob)
    val people2 = listOf(Person("Adam", 20), Person("Bob", 31), b
ob)
    println(people == people2) // true
    bob.age = 32
    println(people == people2) // false
}
MutableList<T> - это List с операциями записи, специфичными для
списка, например, для добавления или удаления элемента в
определённой позиции.
fun main() {
    val numbers = mutableListOf(1, 2, 3, 4)
    numbers.add(5)
    numbers.removeAt(1)
    numbers[0] = 0
    numbers.shuffle()
    println(numbers) // [4, 0, 3, 5]
}
```

Как видите, в некоторых аспектах списки очень похожи на массивы. Однако есть одно важное отличие: размер массива определяется при инициализации и никогда не изменяется; в свою очередь список не имеет предопределённого размера; размер списка может быть изменён в результате операций записи: добавления, обновления или удаления элементов.

По умолчанию в *Kotlin* реализацией `List` является `ArrayList`, который можно рассматривать как массив с изменяемым размером.

Set

`Set<T>` хранит уникальные элементы; их порядок обычно не определён. `null` также является уникальным элементом: `Set` может содержать только один `null`. Два множества равны, если они имеют одинаковый размер и для каждого элемента множества есть равный элемент в другом множестве.

```
fun main() {
    val numbers = setOf(1, 2, 3, 4)
    println("Number of elements: ${numbers.size}") // Number of e
lements: 4
    if (numbers.contains(1)) println("1 is in the set")

    val numbersBackwards = setOf(4, 3, 2, 1)
    println("The sets are equal: ${numbers == numbersBackwards}")
// true
}
```

`MutableSet` – это `Set` с операциями записи из `MutableCollection`.

По умолчанию реализацией `Set` является `LinkedHashSet`, который сохраняет порядок вставки элементов. Следовательно, функции, которые зависят от порядка элементов, такие как `first()` или `last()`, возвращают предсказуемые результаты для таких множеств.

```
fun main() {
    val numbers = setOf(1, 2, 3, 4)  // по умолчанию LinkedHashSe
t
    val numbersBackwards = setOf(4, 3, 2, 1)

    println(numbers.first() == numbersBackwards.first()) // false
    println(numbers.first() == numbersBackwards.last()) // true
}
```

Альтернативная реализация – `HashSet` – не сохраняет порядок элементов, поэтому при вызове функций `first()` или `last()` вернётся непредсказуемый результат. Однако `HashSet` требует меньше памяти для хранения того же количества элементов.

Map

`Map<K, V>` не является наследником интерфейса `Collection`; однако это один из типов коллекций в *Kotlin*. `Map` хранит пары «ключ-значение» (или *entries*); ключи уникальны, но разные ключи могут иметь одинаковые значения. Интерфейс `Map` предоставляет такие функции, как доступ к значению по ключу, поиск ключей и значений и т. д.

```
fun main() {
    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3,
"key4" to 1)

    println("All keys: ${numbersMap.keys}") // [key1, key2, key3,
key4]
    println("All values: ${numbersMap.values}") // [1, 2, 3, 1]
    if ("key2" in numbersMap) println("Value by key \"key2\": ${n
umbersMap["key2"]}")
    if (1 in numbersMap.values) println("The value 1 is in the ma
p")
    if (numbersMap.containsValue(1)) println("The value 1 is in t
he map") // аналогичен предыдущему условию
}
```

Две `Map`-ы, содержащие равные пары, равны независимо от порядка пар.

```
fun main() {
    val numbersMap = mapOf("key1" to 1, "key2" to 2, "key3" to 3,
"key4" to 1)
    val anotherMap = mapOf("key2" to 2, "key1" to 1, "key4" to 1,
"key3" to 3)

    println("The maps are equal: ${numbersMap == anotherMap}") //
The maps are equal: true
}
```

MutableMap – это Map с операциями записи, например, можно добавить новую пару «ключ-значение» или обновить значение, связанное с указанным ключом.

```
fun main() {
    val numbersMap = mutableMapOf("one" to 1, "two" to 2)
    numbersMap.put("three", 3)
    numbersMap["one"] = 11

    println(numbersMap) // {one=11, two=2, three=3}
}
```

По умолчанию реализацией `Map` является `LinkedHashMap` – сохраняет порядок элементов. Альтернативная реализация – `HashMap` – не сохраняет порядок элементов.

Вопросы к лабораторной работе

1Что такое *JVM*?

2Пакеты в *Kotlin*?

3Классы в *Kotlin*.

4Как объявить переменную в *Kotlin*? Различие между *var* и *val*.

5Сколько конструкторов доступно в *Kotlin*?

6Для чего служат абстрактные классы?

7Какой порядок вызова конструкторов при наследовании?

8Как переопределить метод родительского класса?

9Что такое оператор «*Elvis*» в *Kotlin* и как он используется?

10 Как объявить и использовать лямбда-выражения в *Kotlin*?

11 Что такое модификаторы доступа (*access modifiers*) в *Kotlin*?

12 Как работает оператор *when* в *Kotlin* и какие его особенности?

13 Как создать и использовать классы и объекты в *Kotlin*?

14 Что такое *inline*-функции и зачем их применять?

15 Как обрабатывать исключения с помощью блока *try-catch* в *Kotlin*?

16 Как работает оператор in для проверки наличия элемента в коллекции?

17 Как объявить и использовать расширения для стандартных классов в *Kotlin*?

18 Что такое функциональные типы данных (*function types*) и как их применять?

Задание на лабораторную работу № 1

Задание к лабораторной работе:

– согласно варианта задания написать программу (Вариант = номер в списке группы % 30 + 1);

– объекты и их взаимоотношения, имеющиеся в варианте задания, должны быть реализованы;

– функциональная часть приложения, представленная диаграммой последовательности, должна быть реализована;

– разбить функционал приложения на несколько пакетов придерживаясь логики.

Дополнительное задание:

– написать *Unit*-тесты при помощи *JUnit* 5: *https://kotlinlang.org/docs/jvm-test-using-junit.html*.

– сделать валидацию всех вводимых значений.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Изучить теоретическую часть лабораторной работы.

2. Реализовать индивидуальное задание по вариантам (Вариант берётся согласно списку в группе), сделать скриншоты работающих программ. Написать комментарии.

3. Написать отчет, содержащий:

1. Титульный лист, на котором указывается:

а) полное наименование министерства образование и название учебного заведения;

б) название дисциплины;

в) номер практического занятия;

г) фамилия преподавателя, ведущего занятие;

д) фамилия, имя и номер группы студента;

е) год выполнения лабораторной работы.

2. Индивидуальное задание с кодом, комментариями и скриншотами работающих программ.

3. Вывод по проделанной работе.

Варианты заданий

Вариант 1

Добавить класс журнал и организовать взаимодействие с ним. Чтобы при вызове метода Отметить присутствующих класса Староста в классе Строка журнала обозналось присутствие студента на лекции. А при вызове метода Отметить присутствующих класса Преподаватель эти данные считывались и сравнивались с данными объекта Преподаватель.

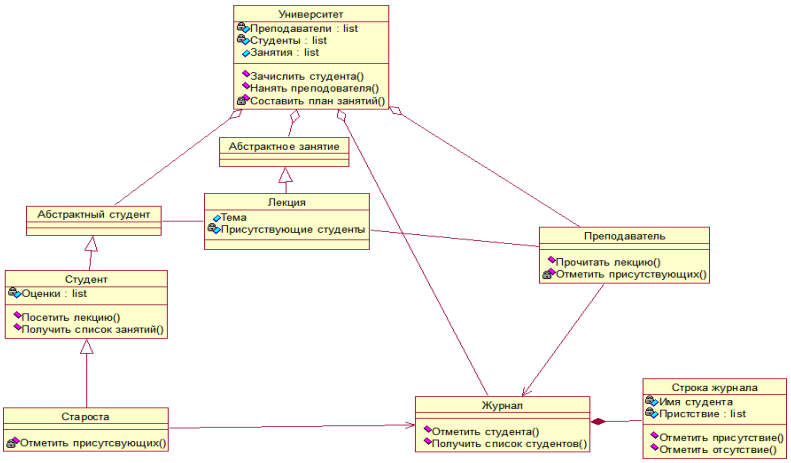

Рисунок 1 – Диаграмма классов

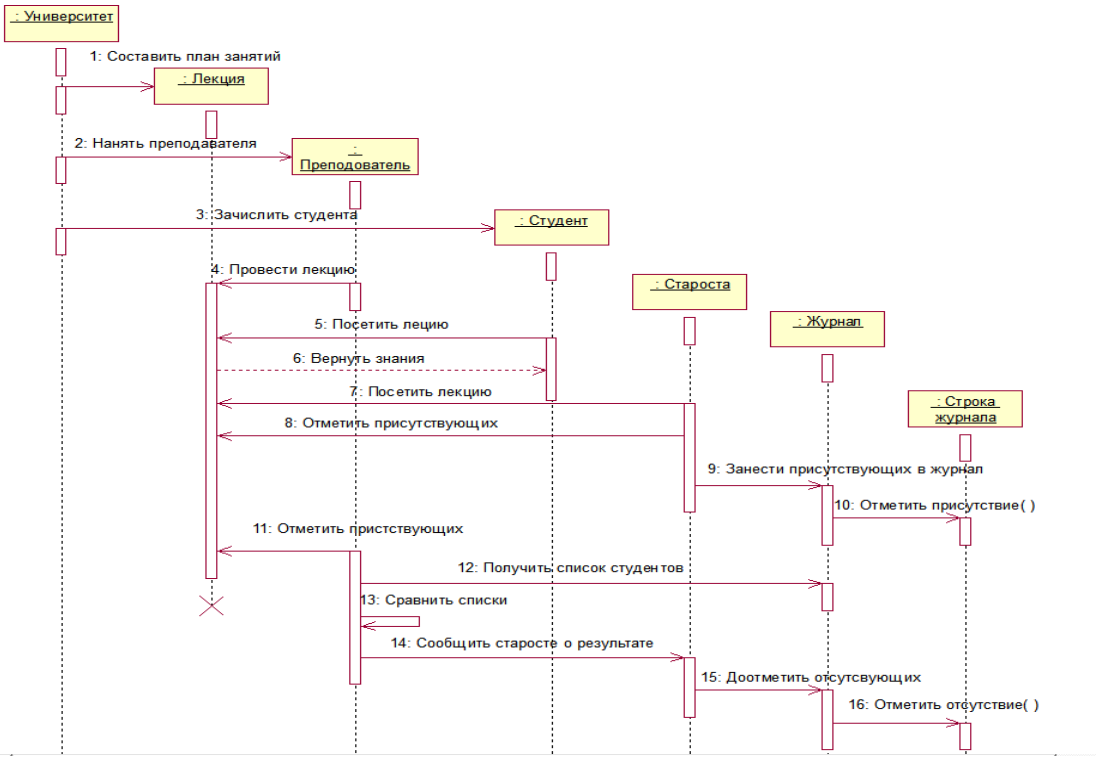

Рисунок 2 – Диаграмма последовательности

Вариант 2

Добавить класс Комплексное число в показательном виде, организовать взаимодействие с классом Комплексное число в алгебраическом виде. Реализовать действие умножения и деления. Реализовать класс Человек, управляющий классом Выражение.

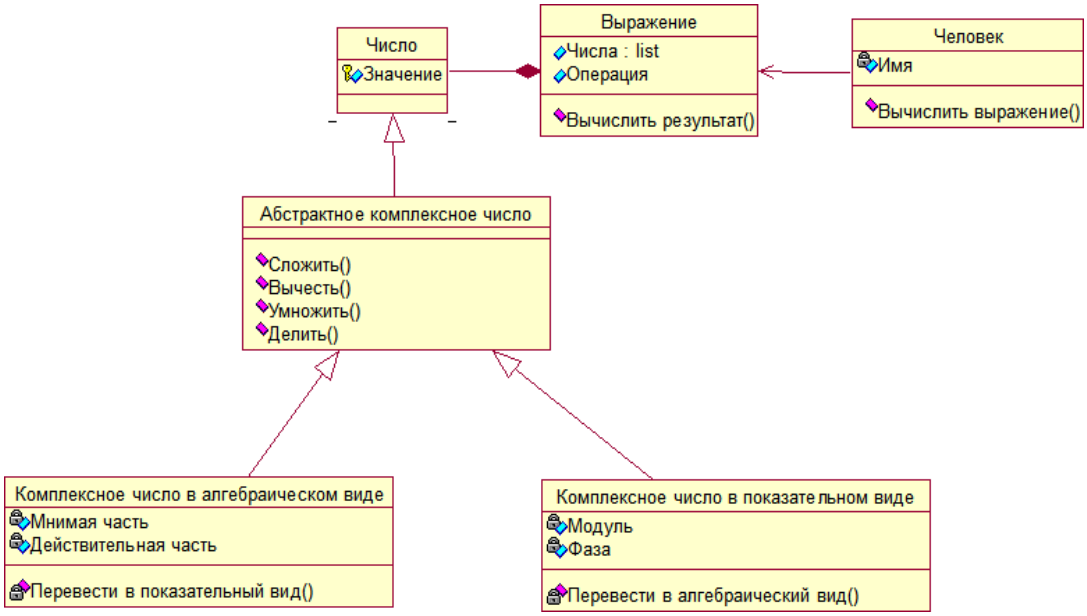

Рисунок 3 – Диаграмма классов

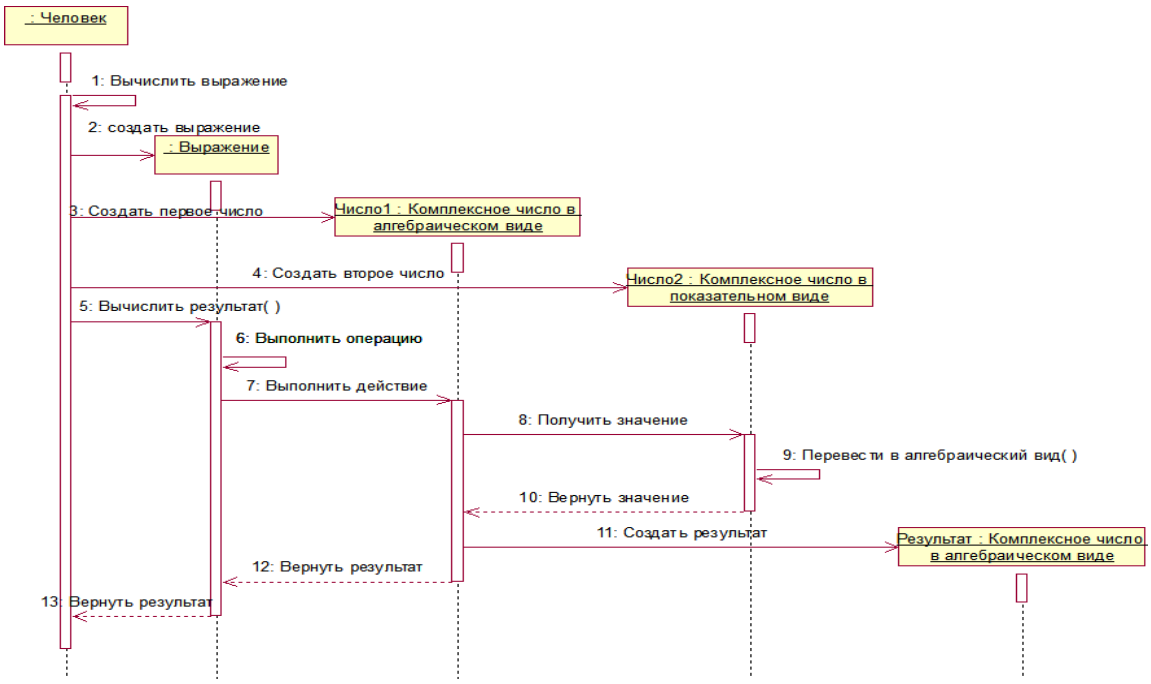

Рисунок 4 – Диаграмма последовательности

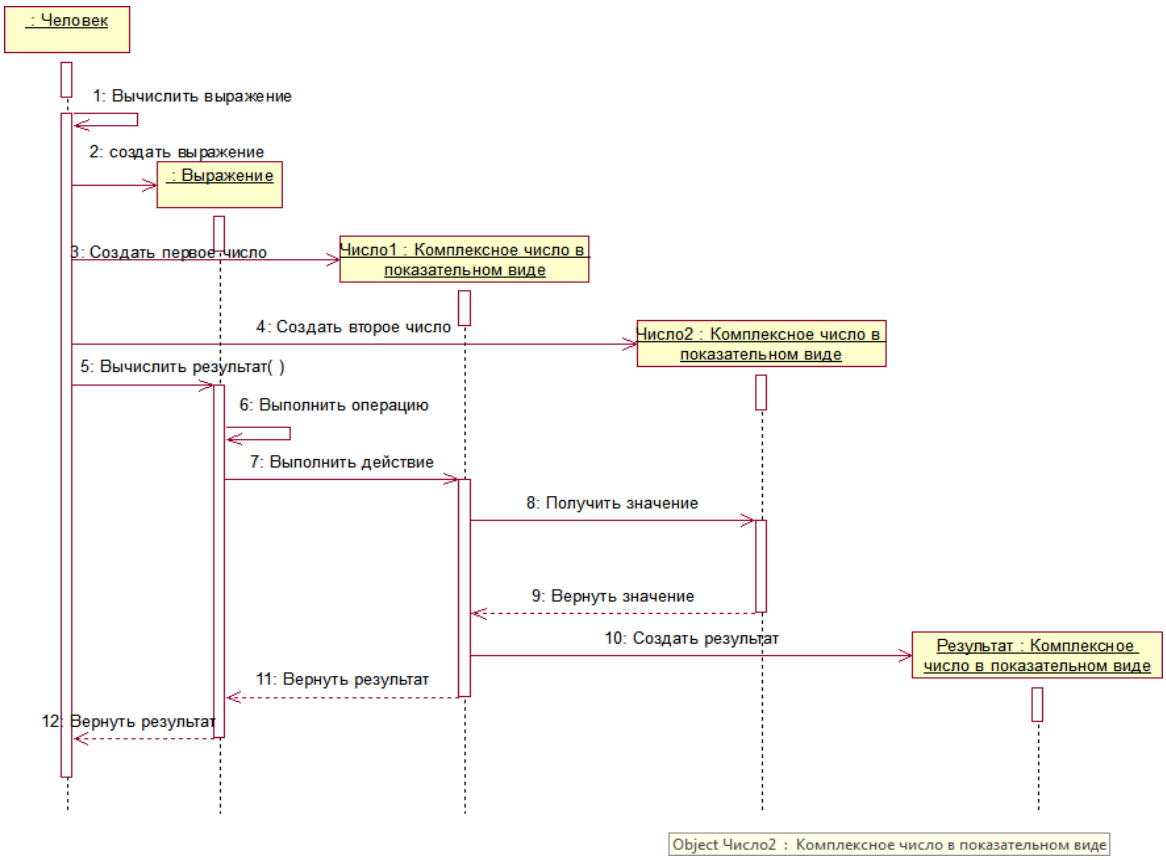

Рисунок 5 – Диаграмма последовательности

Вариант 3

Добавить класс Директор, управляющий классом Завод. Добавить класс Магазин, содержащий ссылку на класс товар, который производит класс Завод. Добавить новые классы, наследующие от класса Продукт.

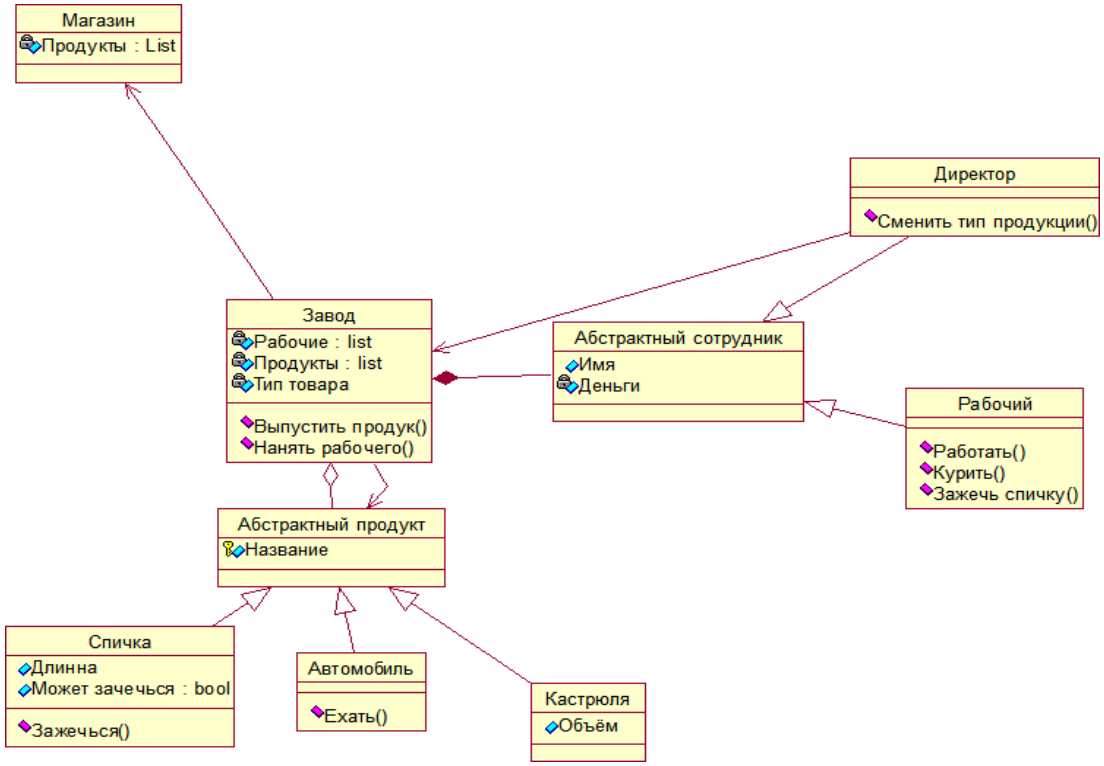

Рисунок 6 – Диаграмма классов

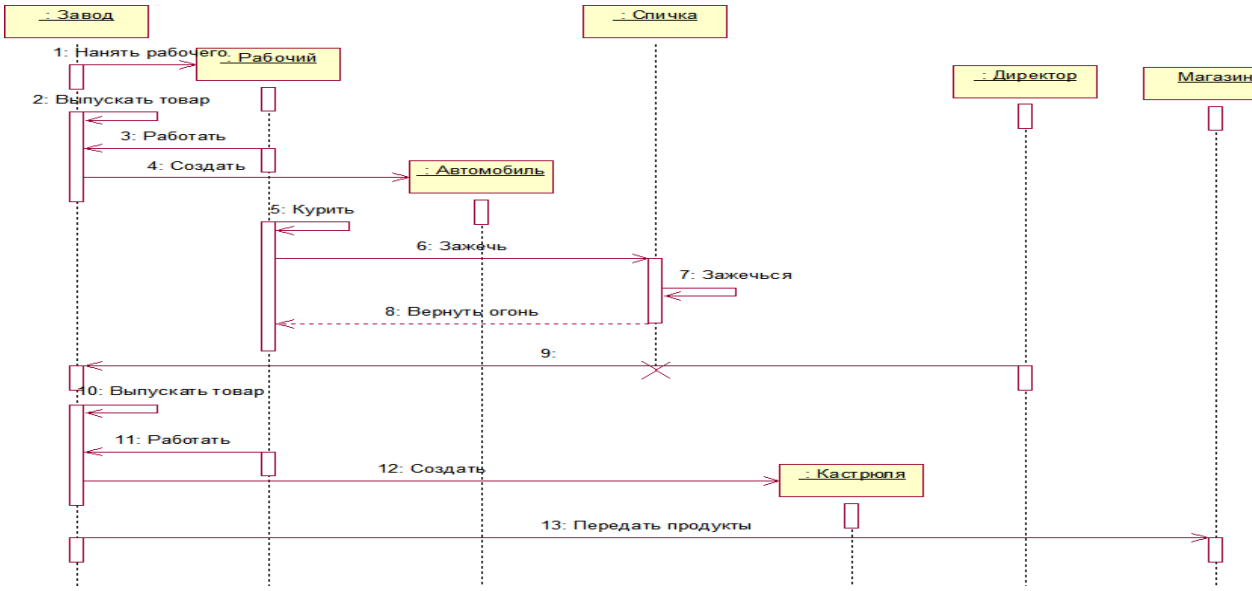

Рисунок 7 – Диаграмма последовательности

Вариант 4

Добавить классы Орган тела и Абстрактный орган. Добавить класс ухо, наследующий от класса Орган головы, класс Желудок, наследующий от класса Орган тела. Организовать взаимодействие класса Рот с классом Желудок и класса Ухо с классом Мозг.

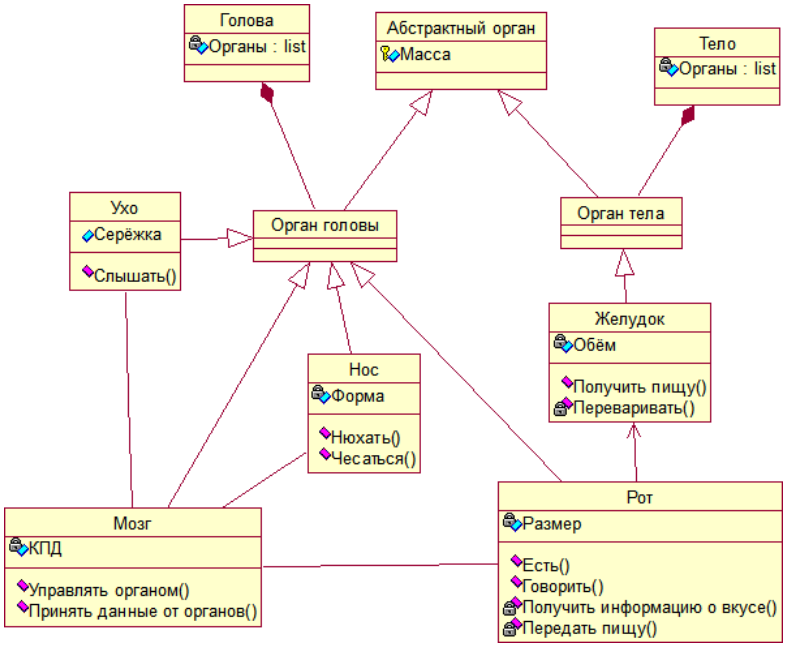

Рисунок 8 – Диаграмма классов

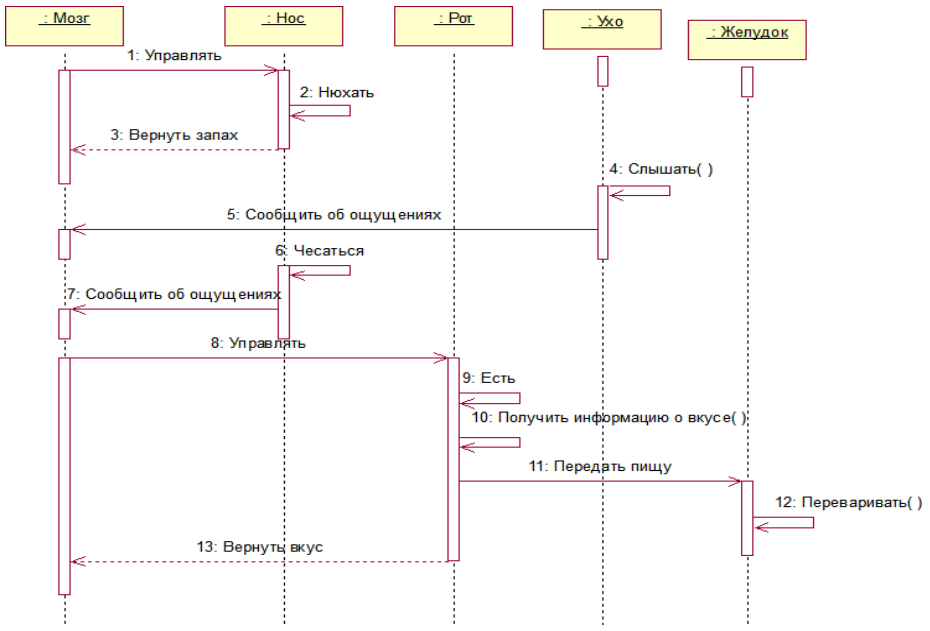

Рисунок 9 – Диаграмма последовательности

Вариант 5

Добавить класс Абстрактный файл, от которого наследует класс Программа. Добавить класс Видеофайл, и организовать его взаимодействие с классом Плеер и классом Браузер.

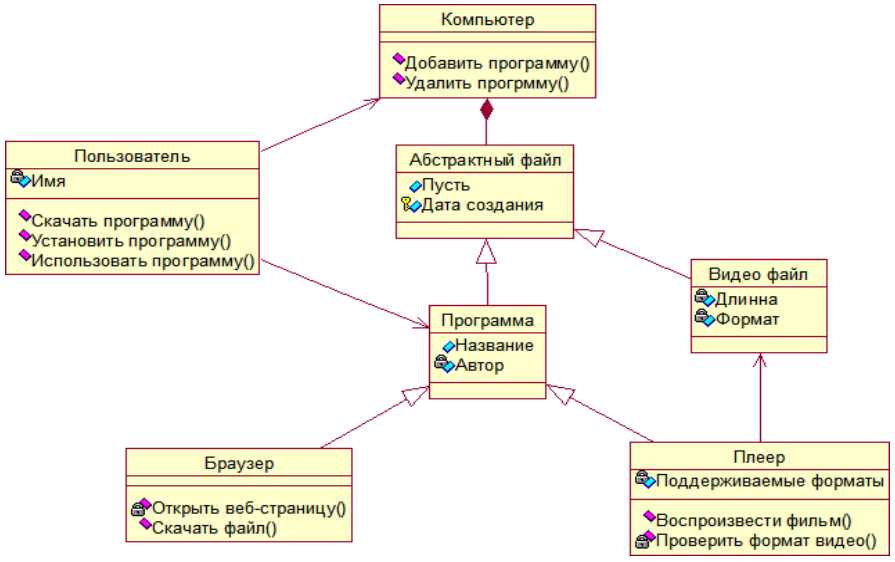

Рисунок 10 – Диаграмма классов

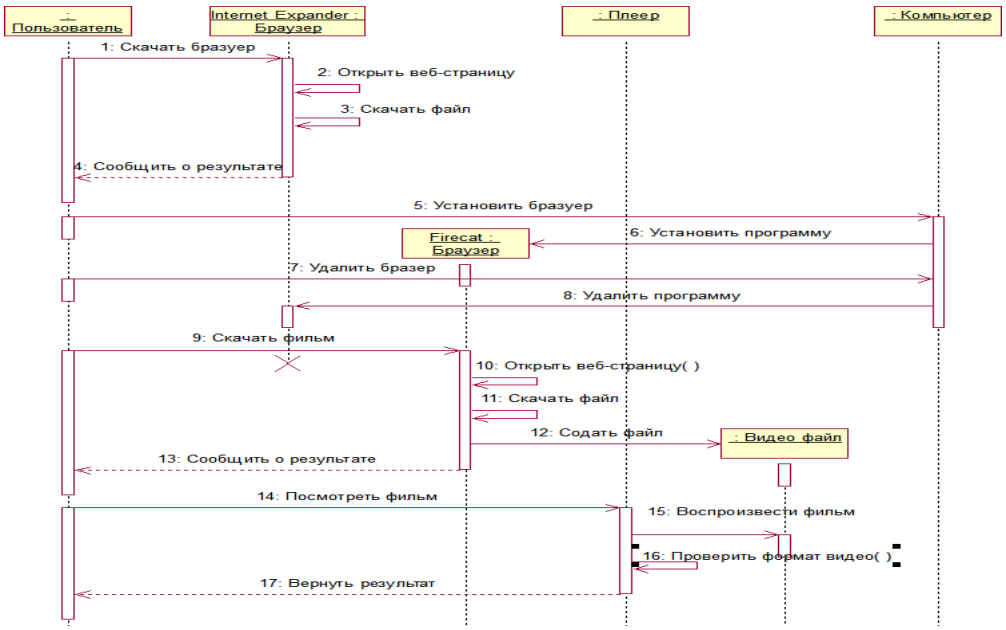

Рисунок 11 – Диаграмма последовательности

Вариант 6

Добавить класс Фермер, наследующий от класса Абстрактный человек и производящийэкземпляры класса Овощ. Эти экземпляры потом использует Повар для производства объектов Еда и разрушает их после производства.

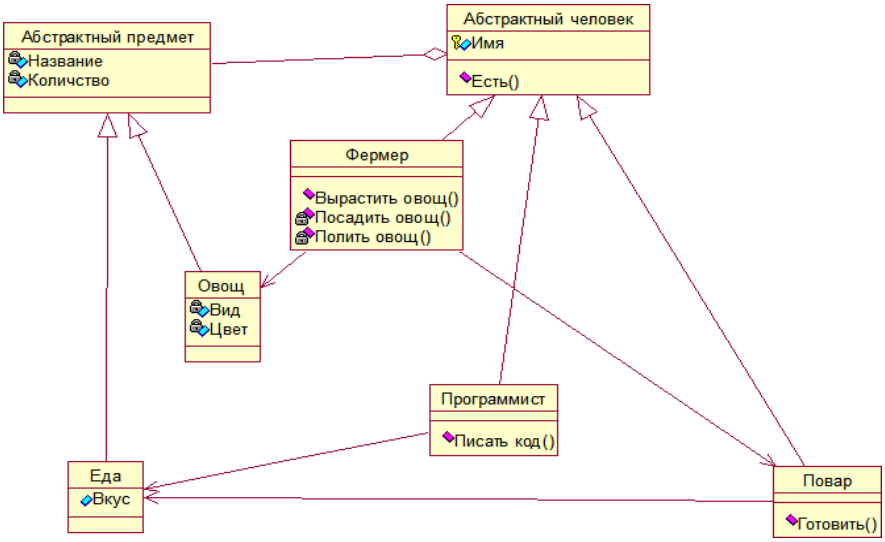

Рисунок 12 – Диаграмма классов

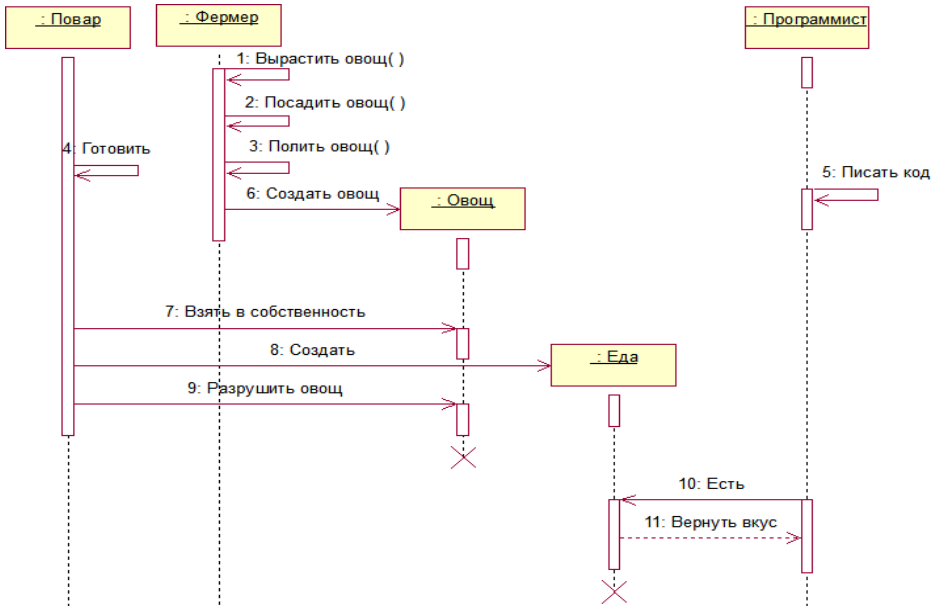

Рисунок 13 – Диаграмма последовательности

Вариант 7

Добавить класс Трактор, наследующий от класса Автомобиль, обладающий методом убирать снег(). Добавить класс Водитель, управляющий классом Автомобиль.

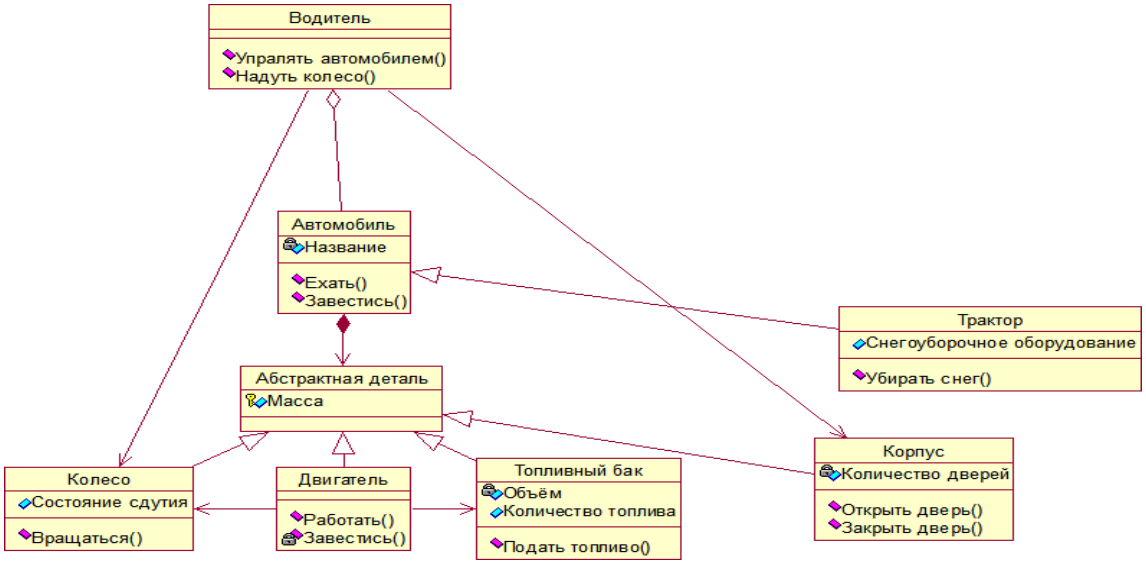

Рисунок 14 – Диаграмма классов

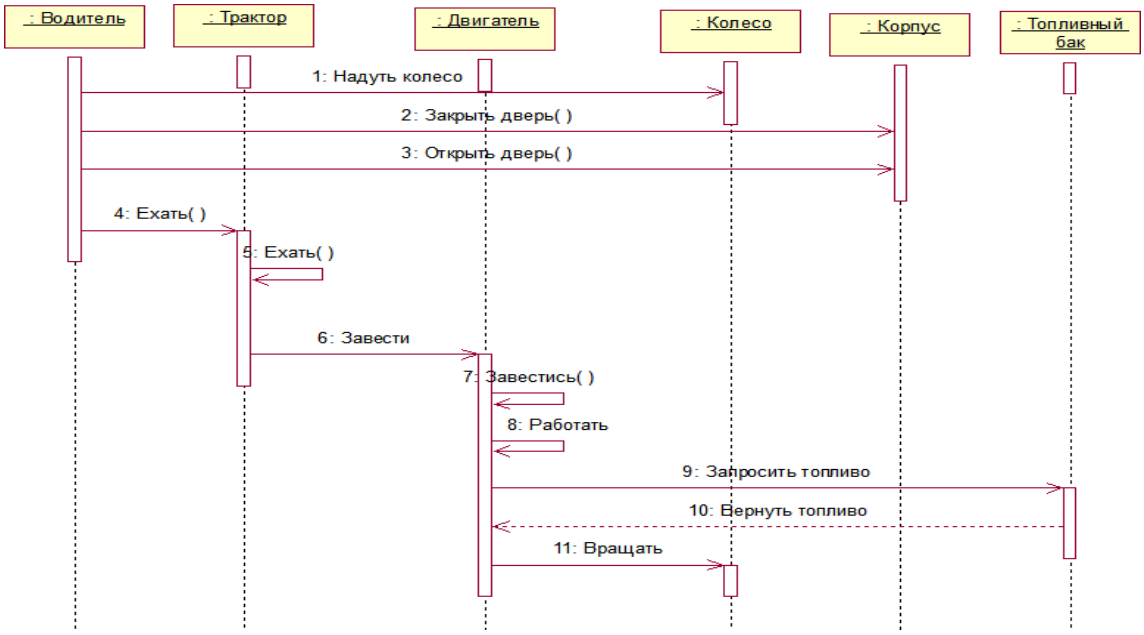

Рисунок 15 – Диаграмма последовательности

Вариант 8

Добавить класс Руководитель Компании, управляющий классом Компания. Добавить класс Вип-клиент, наследующий от класса клиент и класс Срочный заказ, наследующий от класса Заказ.

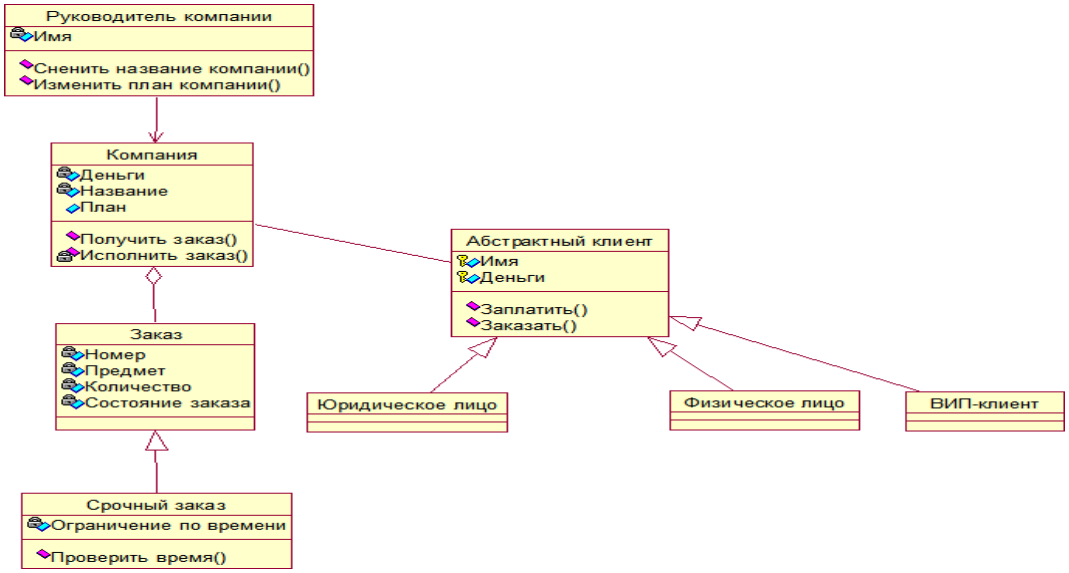

Рисунок 16 – Диаграмма классов

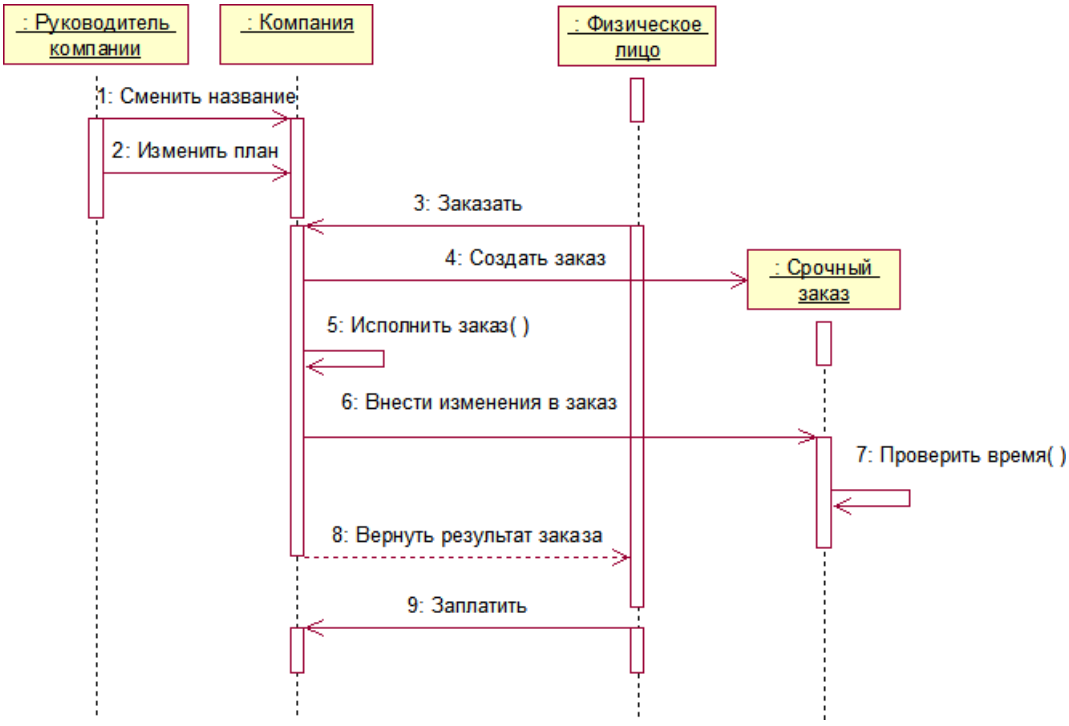

Рисунок 17 – Диаграмма последовательности

Вариант 9

Расширить взаимодействие классов. Организовать получение Работниками дать им возможность попросить увеличить зарплату. Результат просьбы увеличить зарплату должен зависеть от количества выполненных заказов.

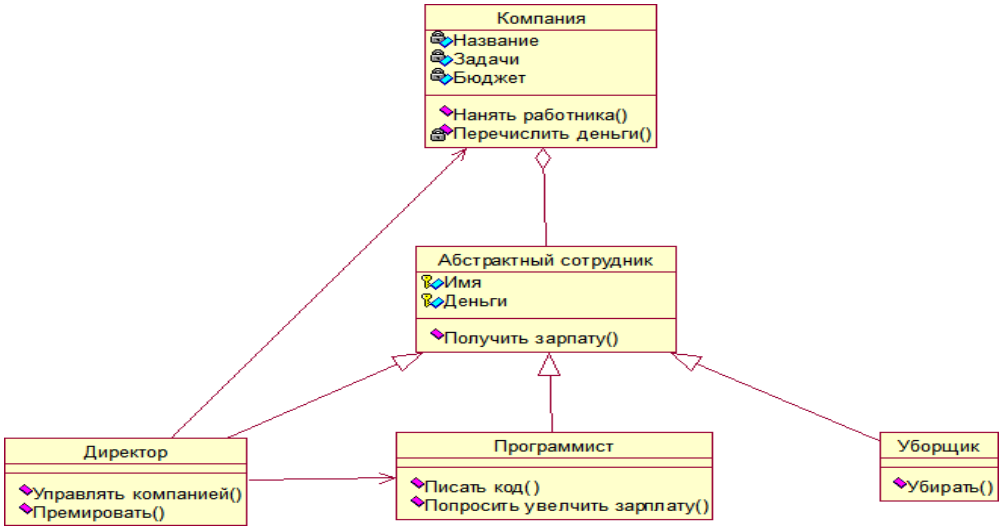

Рисунок 18 – Диаграмма классов

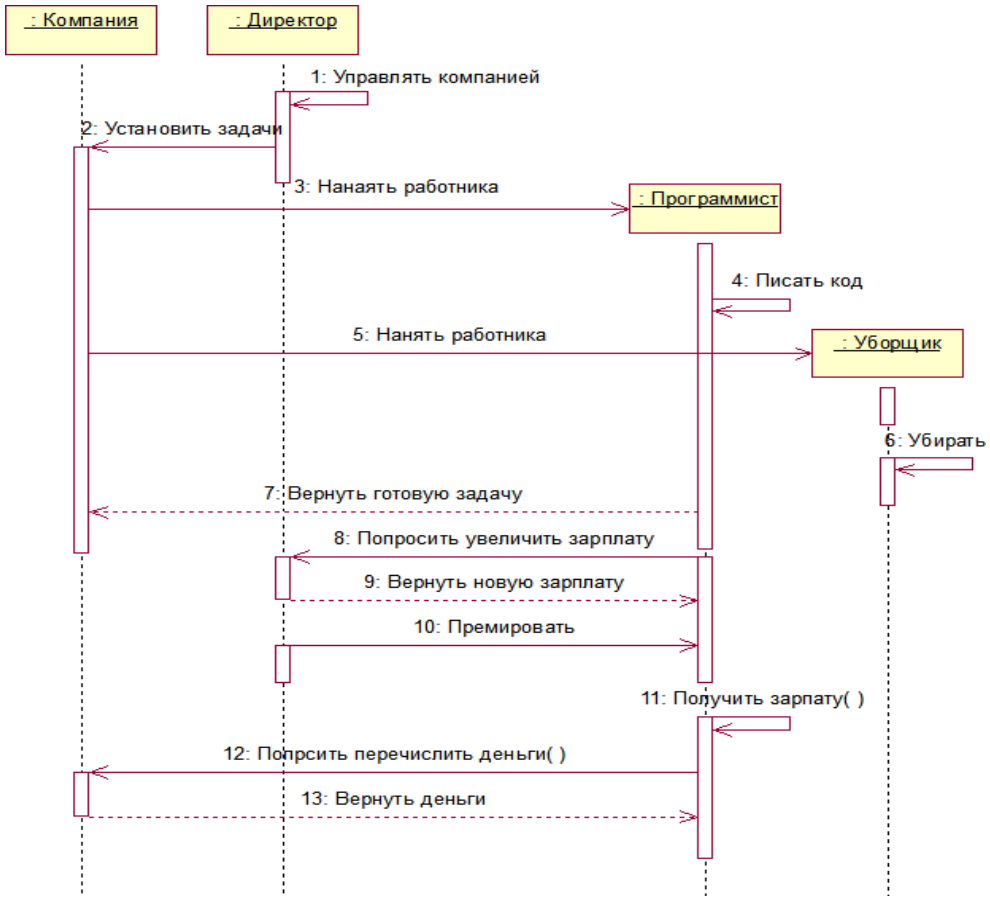

Рисунок 19 – Диаграмма последовательности

Вариант 10

Добавить класс Заведующий кафедрой, который управляет классом университет и отвечает за создание объектов класса экзамен. У класса экзамен есть два состояния сдан и не сдан.

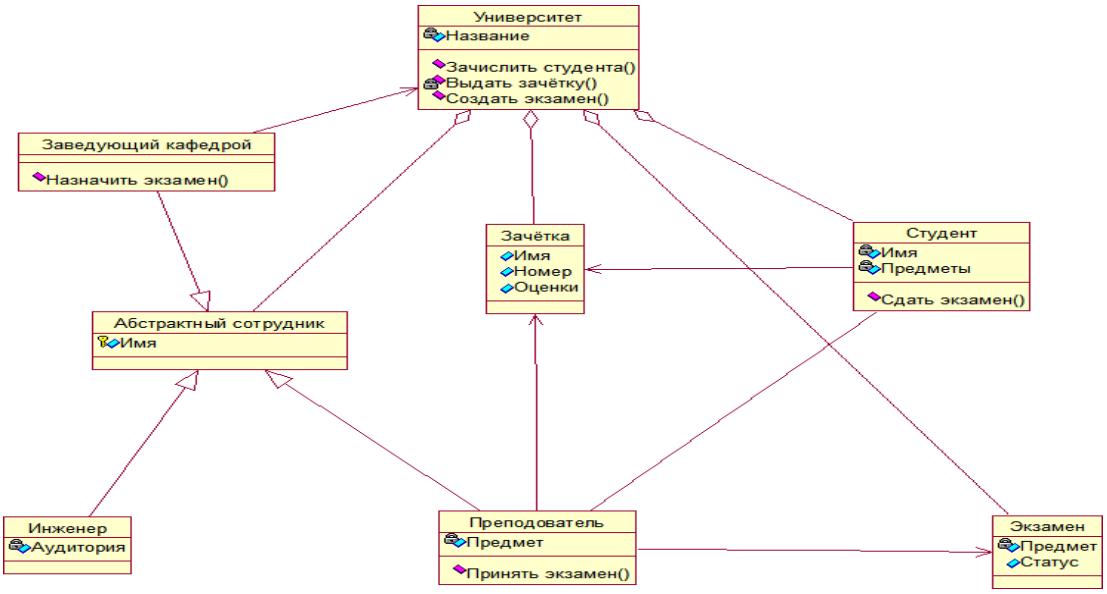

Рисунок 20 – Диаграмма классов

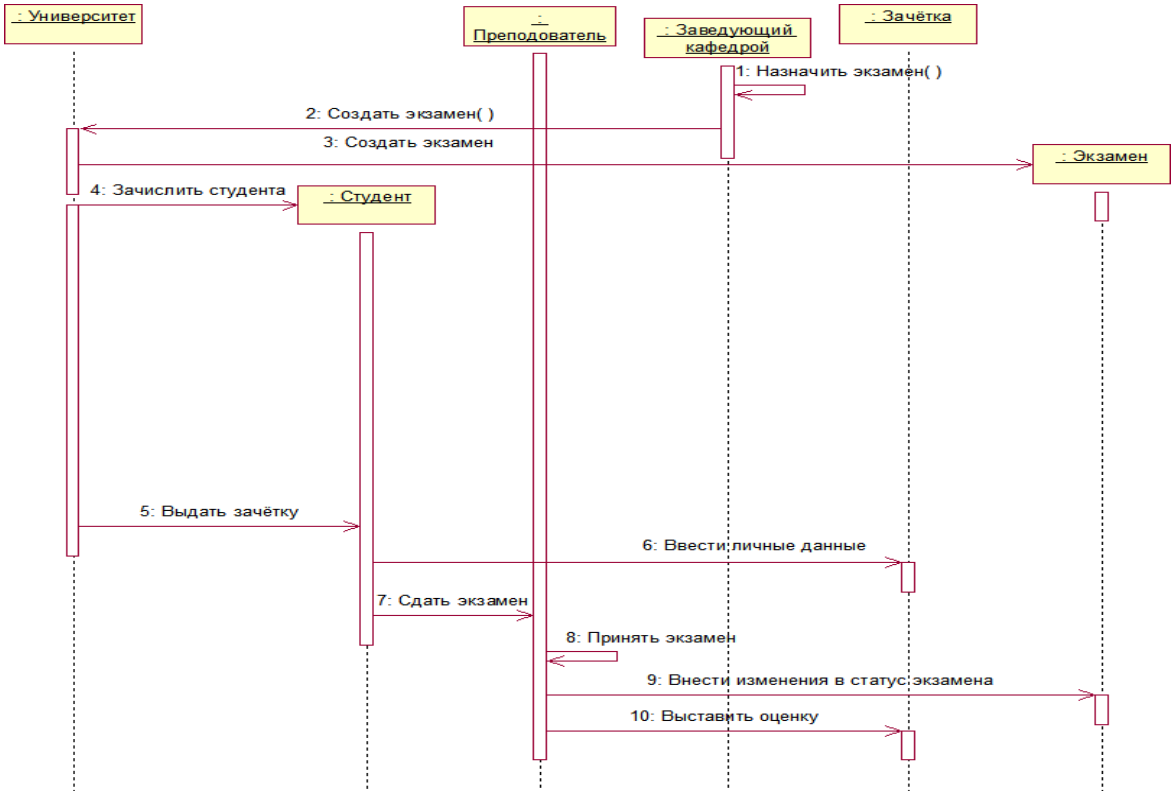

Рисунок 21 – Диаграмма последовательности

Вариант 11

Добавить класс заготовка и обеспечить его взаимодействие с классом отвёртка. Объект класса отвёртка при вызове метода закрутить изменяет статус готовности объекта класса Изделие.

Рисунок 22 – Диаграмма классов

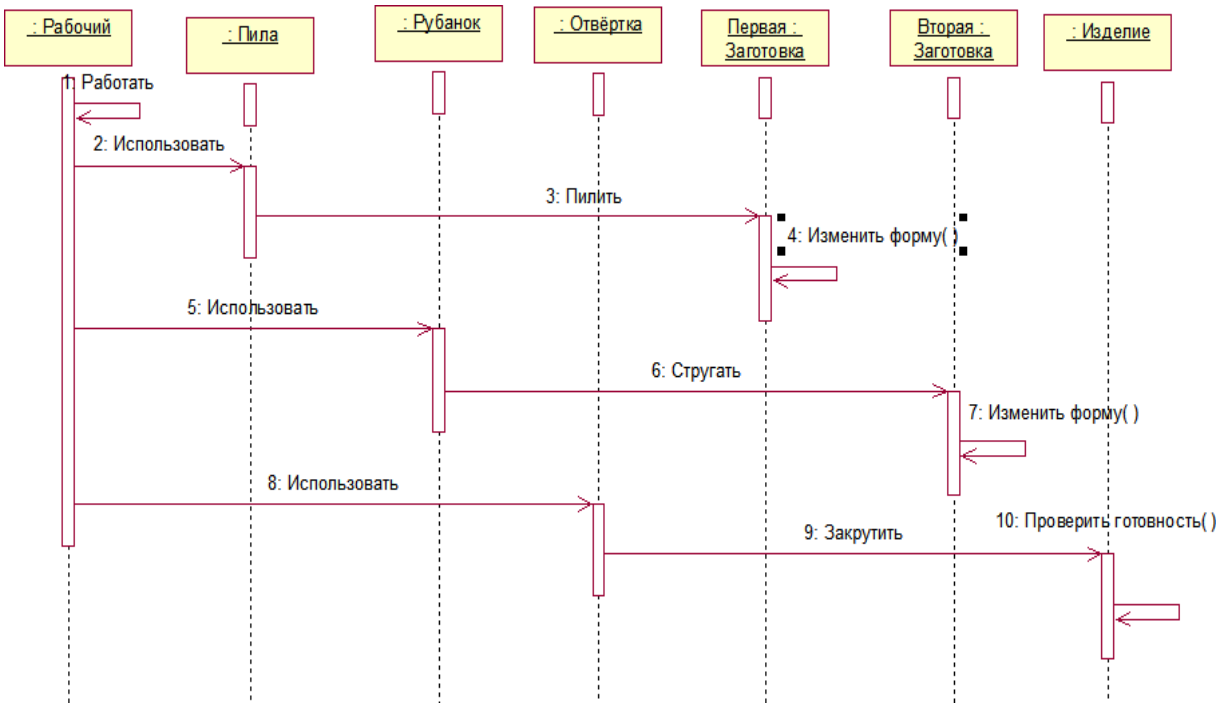

Рисунок 23 – Диаграмма последовательности

Вариант 12

Для классов кофе и сахар сделать родительский класс сыпучий предмет. Добавить наследующий от него класс Чай. Добавить класс молоко. Обеспечить класс Чайник методом Нагреть. Классу студент добавить метод Сделать кофе с молоком() и Сделать чай().

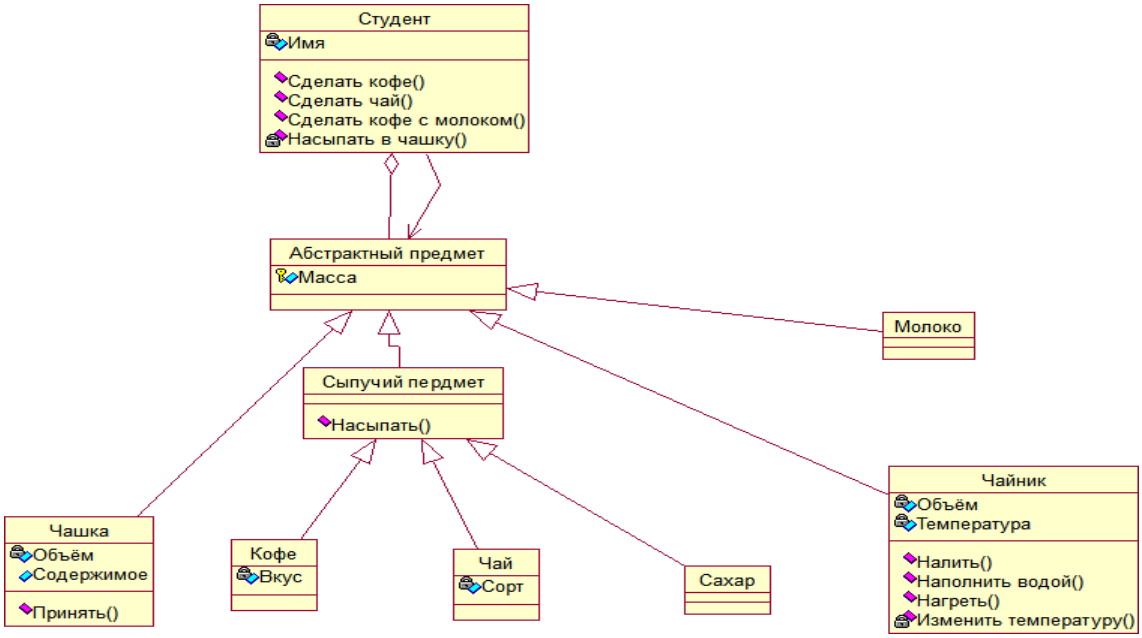

Рисунок 24 – Диаграмма классов

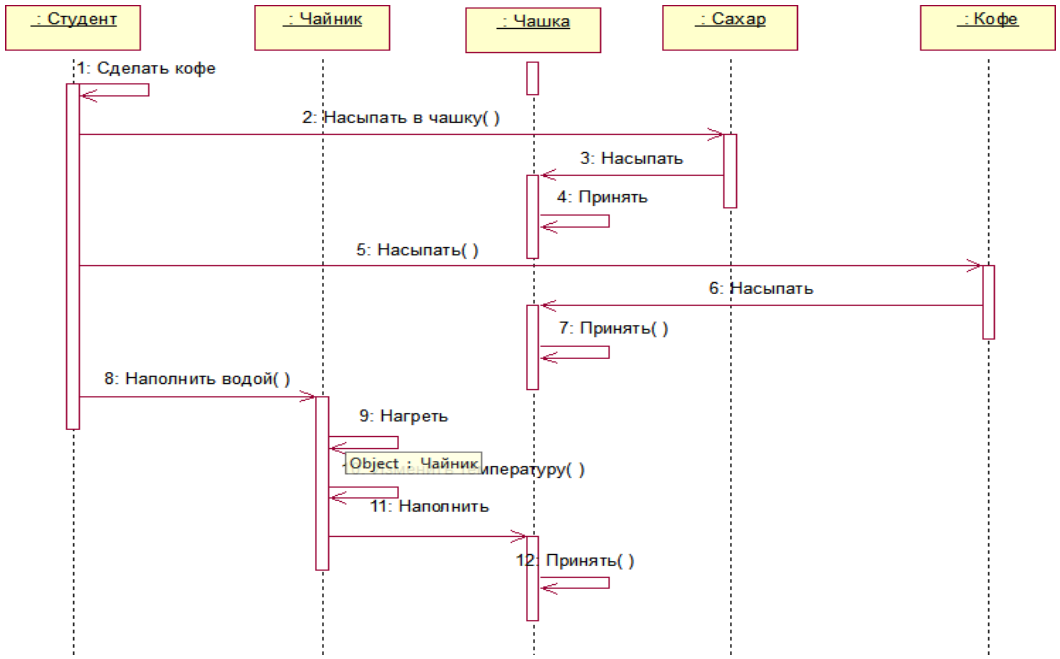

Рисунок 25 – Диаграмма последовательности

Вариант 13

Добавить класс Абстрактный питомец, родительский класс для класса Абстрактная собака и Кошка. Классу хозяин добавить методы Завести собаку() и Завести кошку() создающие объекты классов, наследующих от класса Абстрактная собака и объекты класса Кошка. Классу Хозяин добавить метод Выгулять собаку().

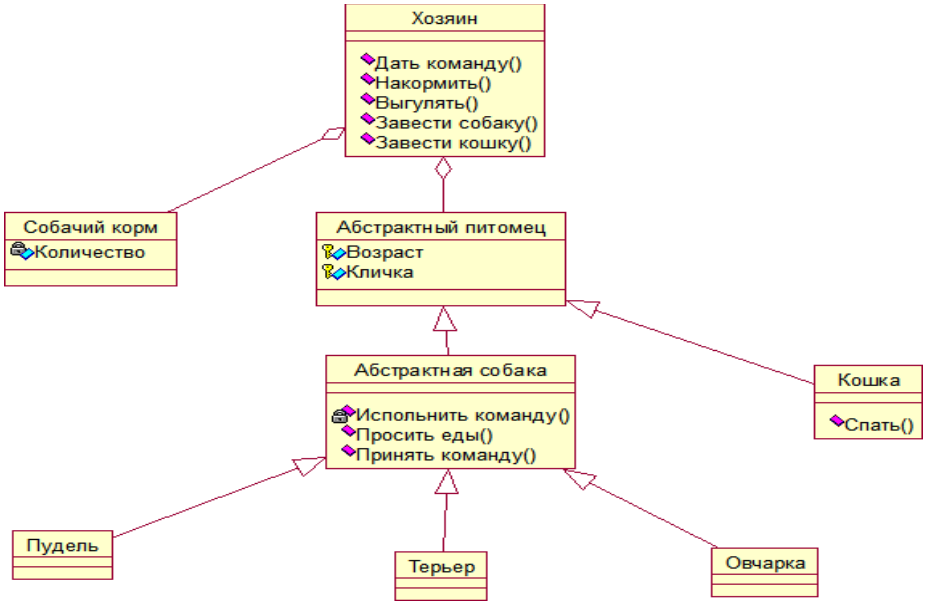

Рисунок 26 – Диаграмма классов

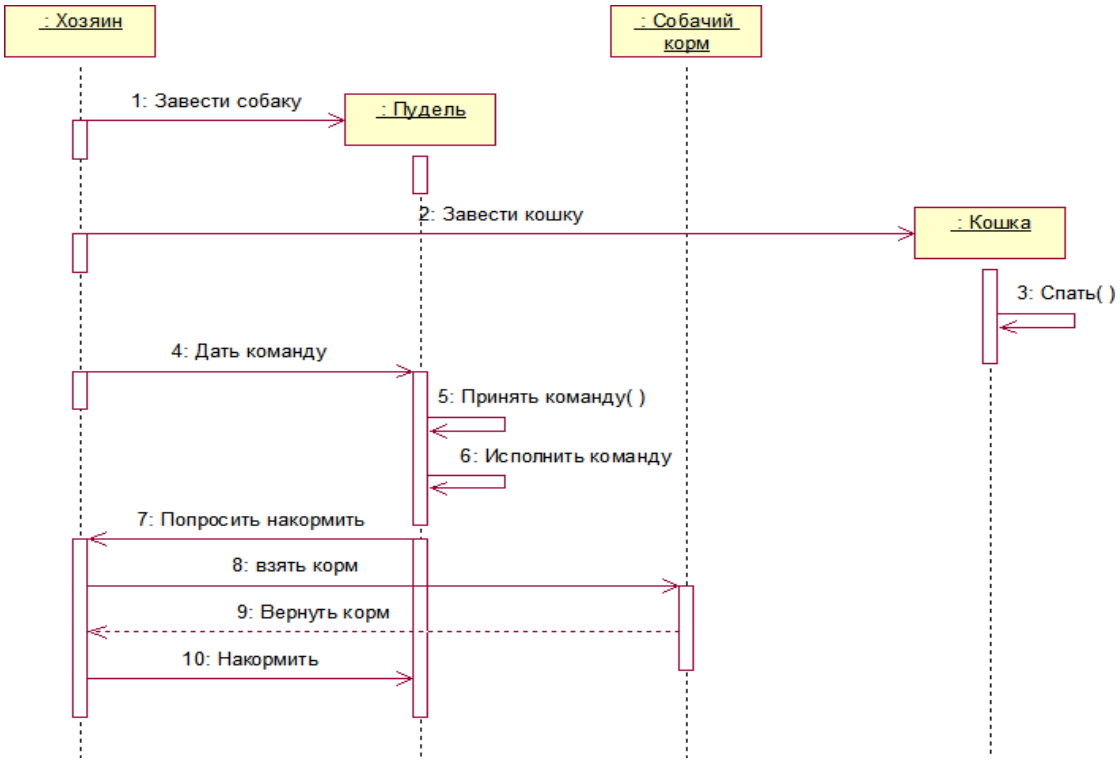

Рисунок 27 – Диаграмма последовательности

Вариант 14

Добавить классы Ангина и Грипп, наследующие от класса Абстрактная болезнь. Пациенту добавить горло и лёгкие. Также добавить метод принять лекарство () разрушающий Болезнь. Добавить класс Лекарство.

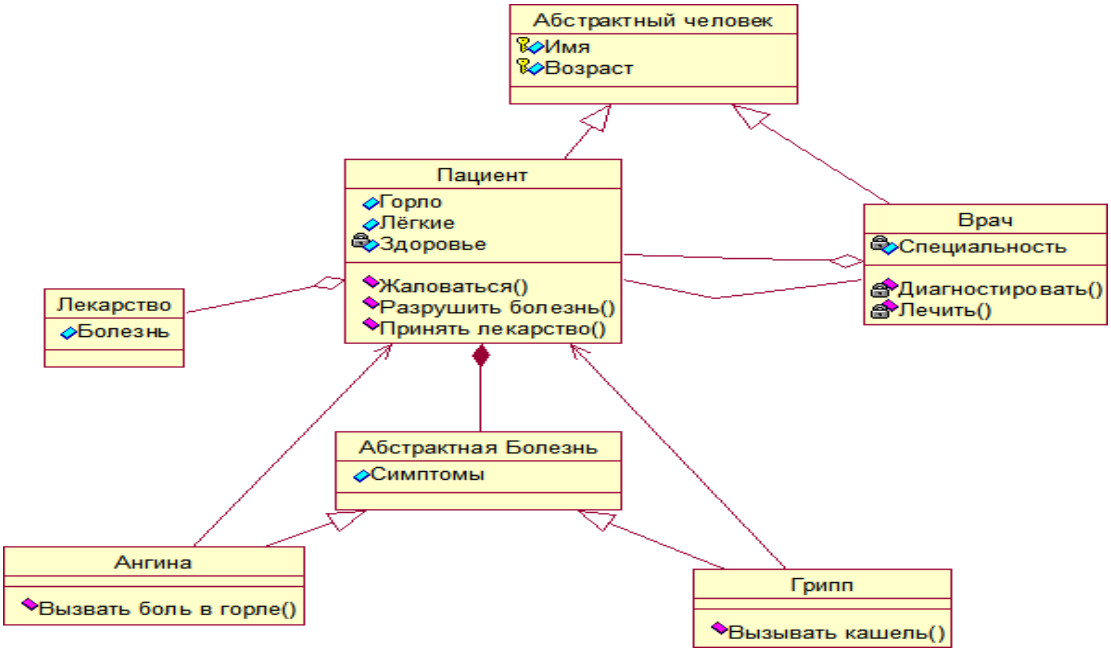

Рисунок 28 – Диаграмма классов

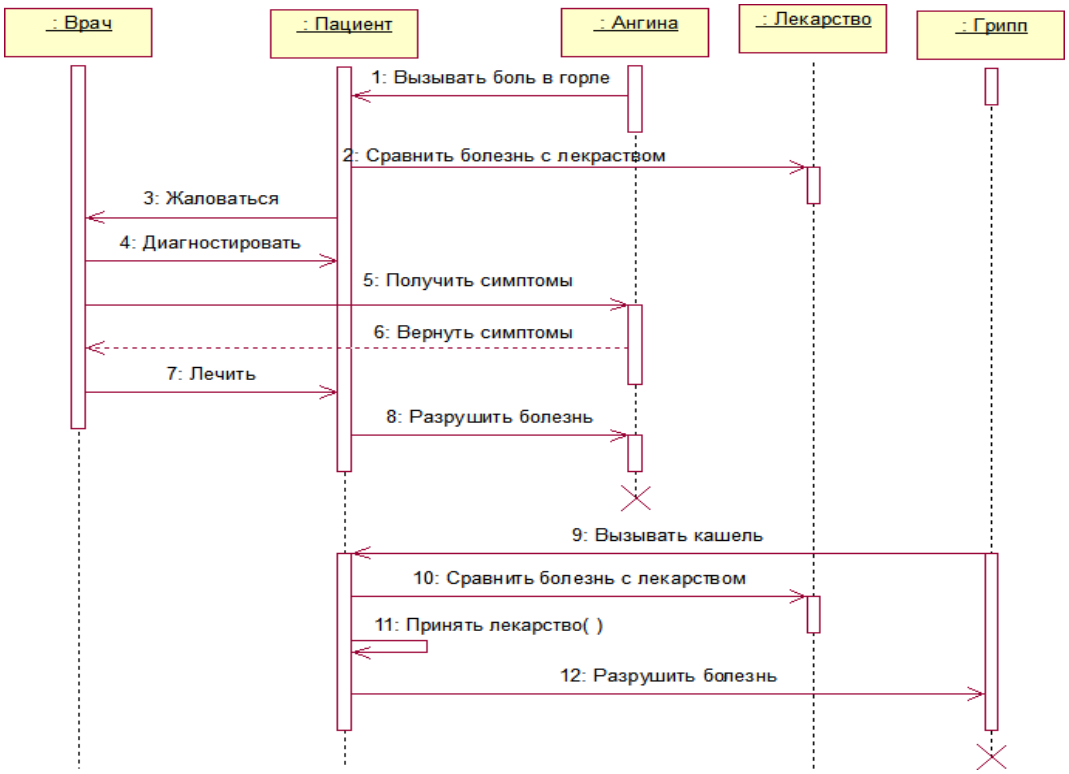

Рисунок 29 – Диаграмма последовательности

Вариант 15

Добавить класс Гладиолус. Добавить класс Лепесток, принадлежащий классу Бутон и класс Лист, принадлежащий классу Цветок. Обеспечить создание Объектов Лепесток одновременно с объектами Бутон. Реализовать метод Засохнуть() у класса Цветок.

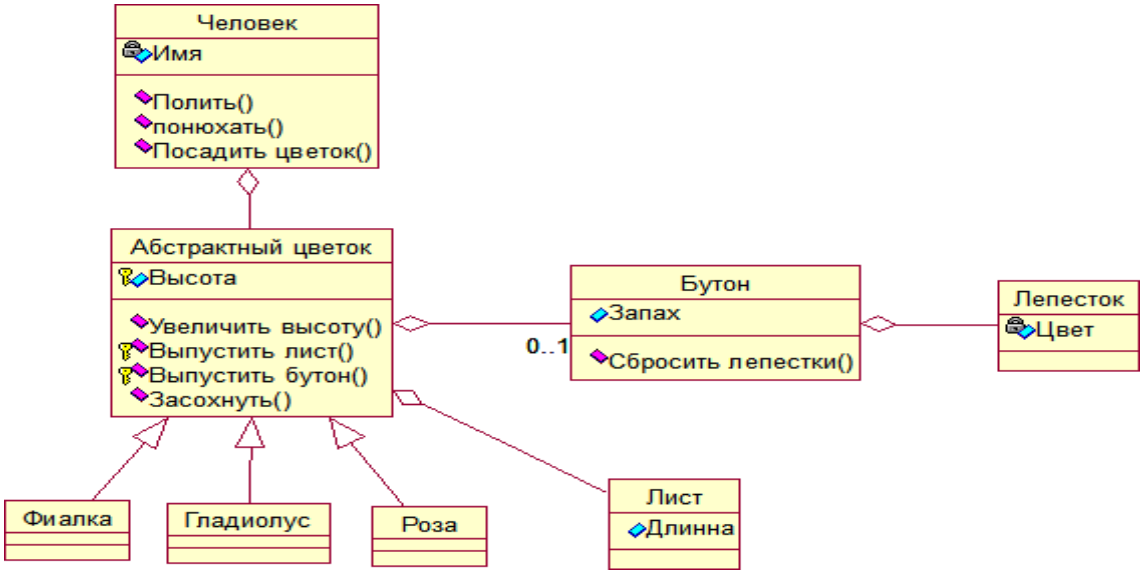

Рисунок 30 – Диаграмма классов

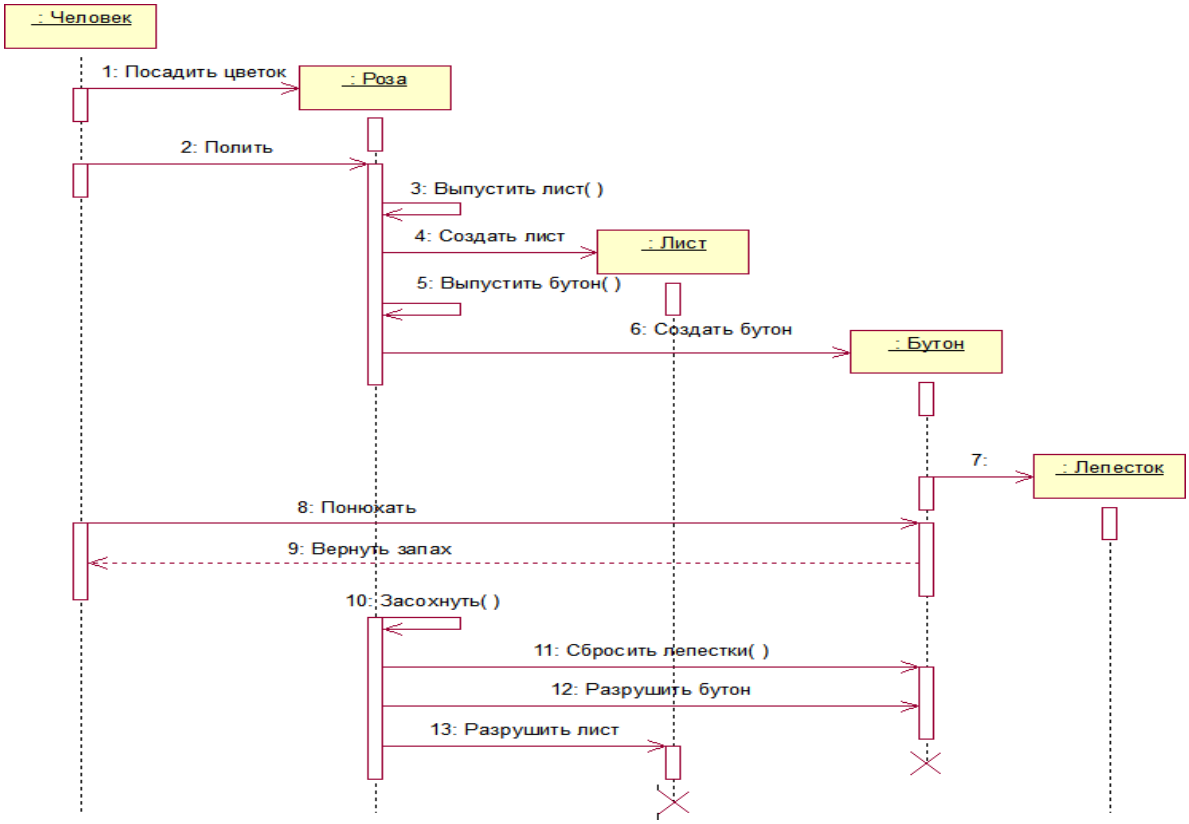

Рисунок 31 – Диаграмма последовательности

Вариант 16

Добавить класс Критик с Методом написать рецензию() и класс Рецензия, который он создаёт. Также создать классы Роман и Рассказ, наследующие от класса Абстрактная книга.

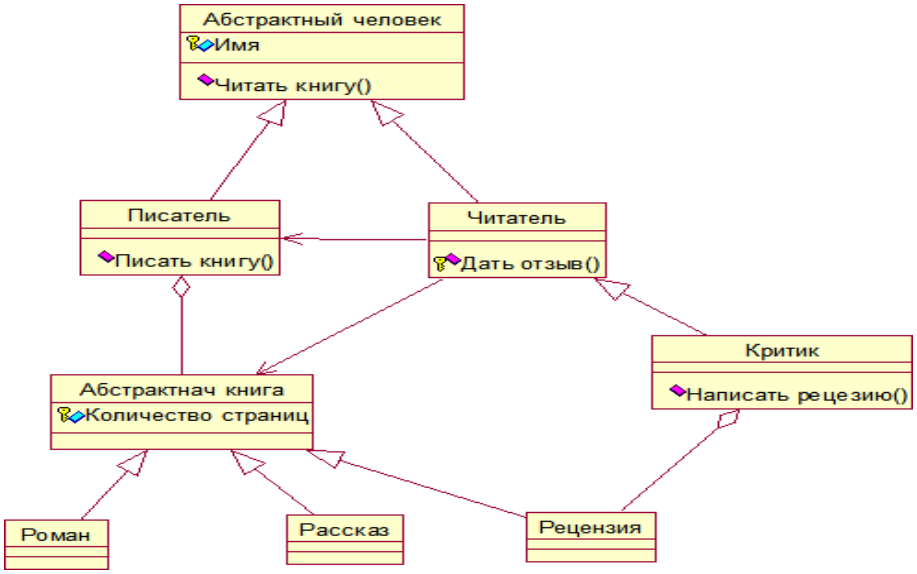

Рисунок 32 – Диаграмма классов

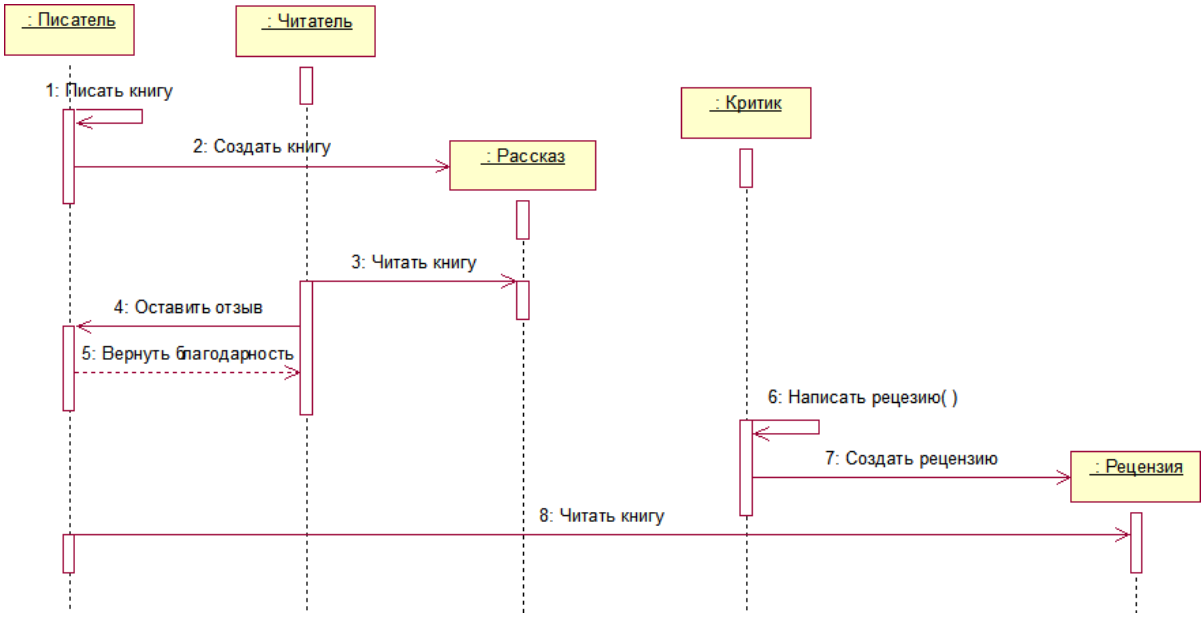

Рисунок 33 – Диаграмма последовательности

Вариант 17

Создать классы Пассажир эконом класса и Пассажир первого класса, наследующие от класса Пассажир. Добавить взаимодействие класса Пассажир первого класса с Классом Стюардесса.

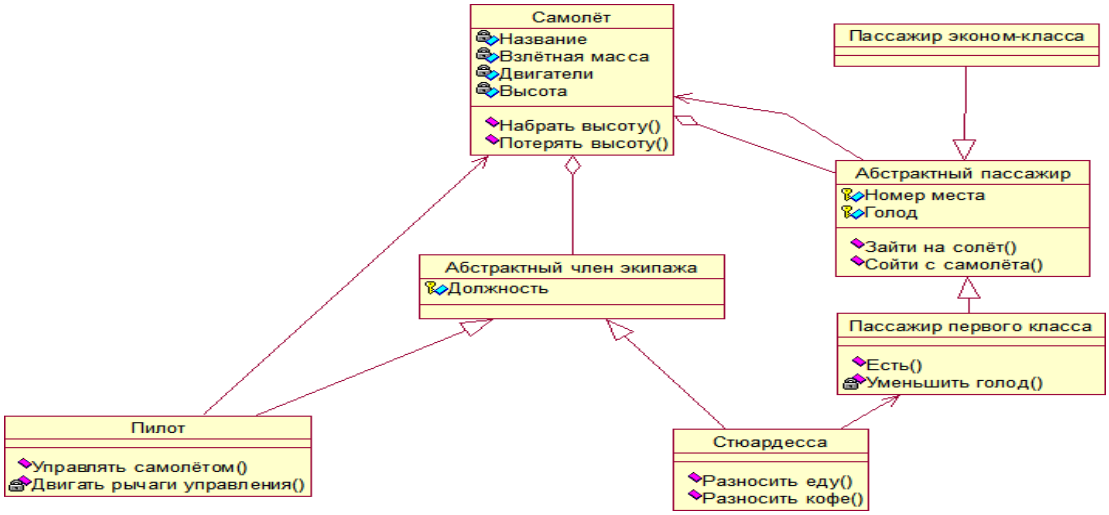

Рисунок 34 – Диаграмма классов

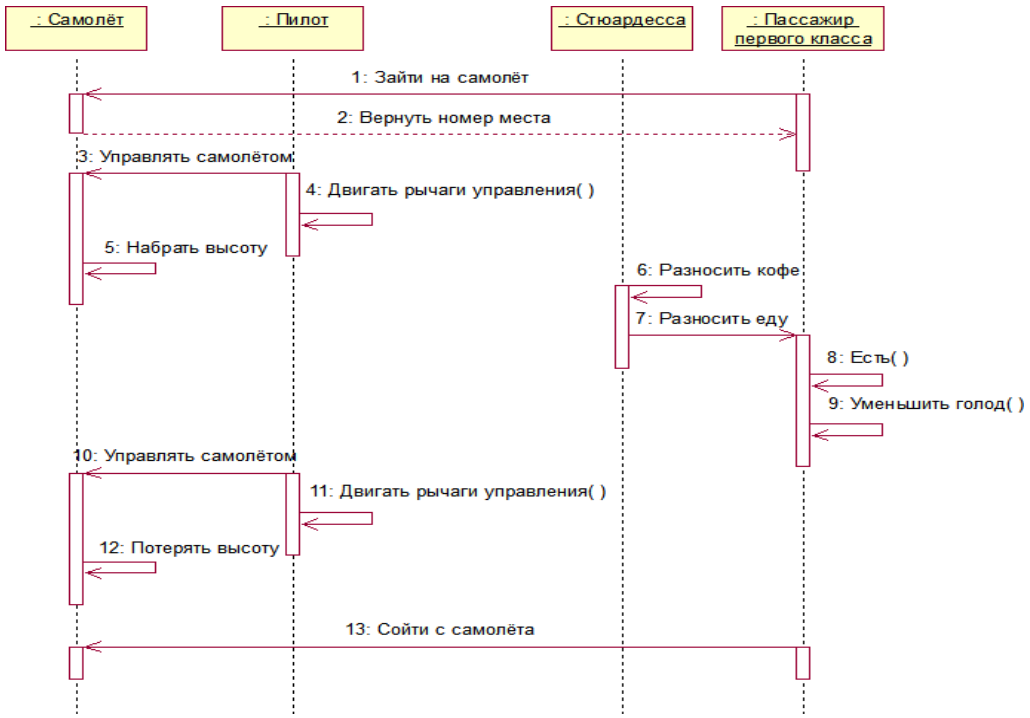

Рисунок 35 – Диаграмма последовательности

Вариант 18

Добавить класс Банк и организовать его взаимодействие с классом Банкомат. В частности реализовать запрос денег Банкоматом из Банка.

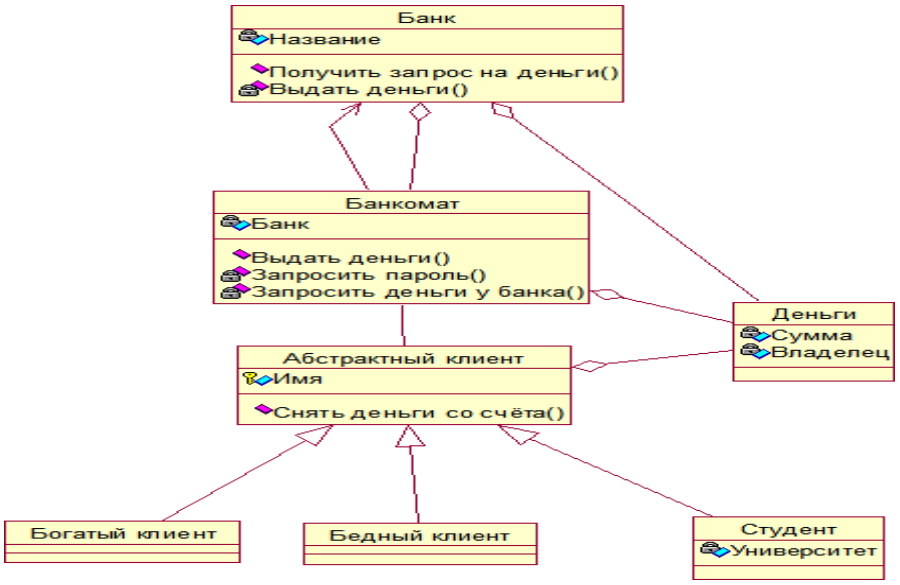

Рисунок 36 – Диаграмма классов

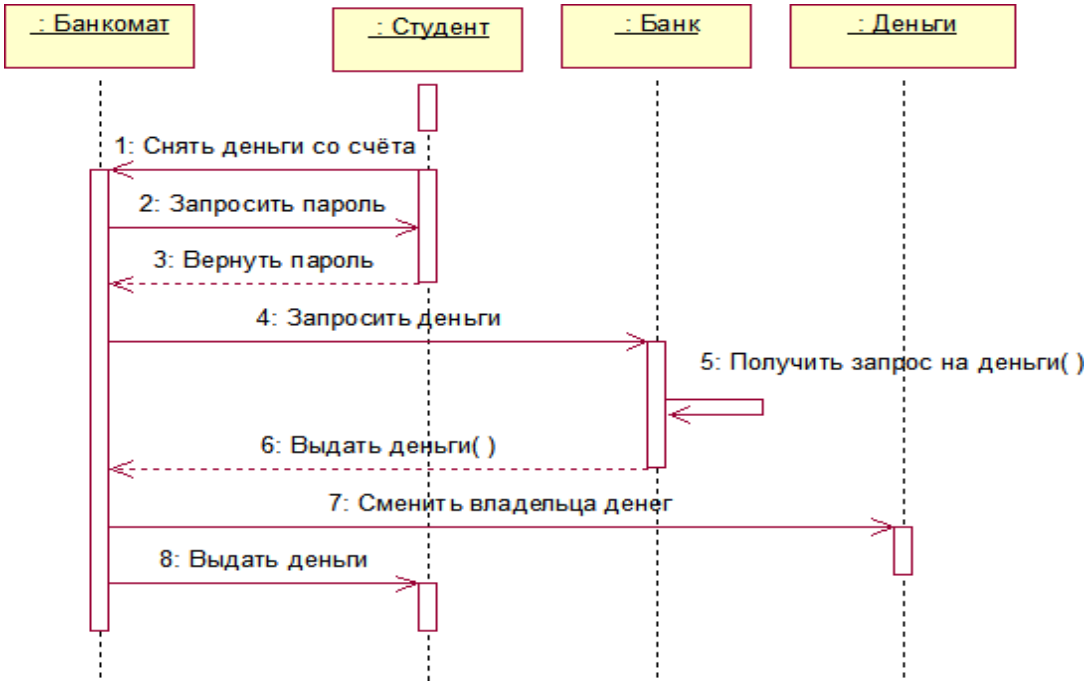

Рисунок 37 – Диаграмма последовательности

Вариант 19

Классу Корабль добавить метод Вести пушечный огонь() и реализовать классы Зенитная пушка и Пушка главного калибра.

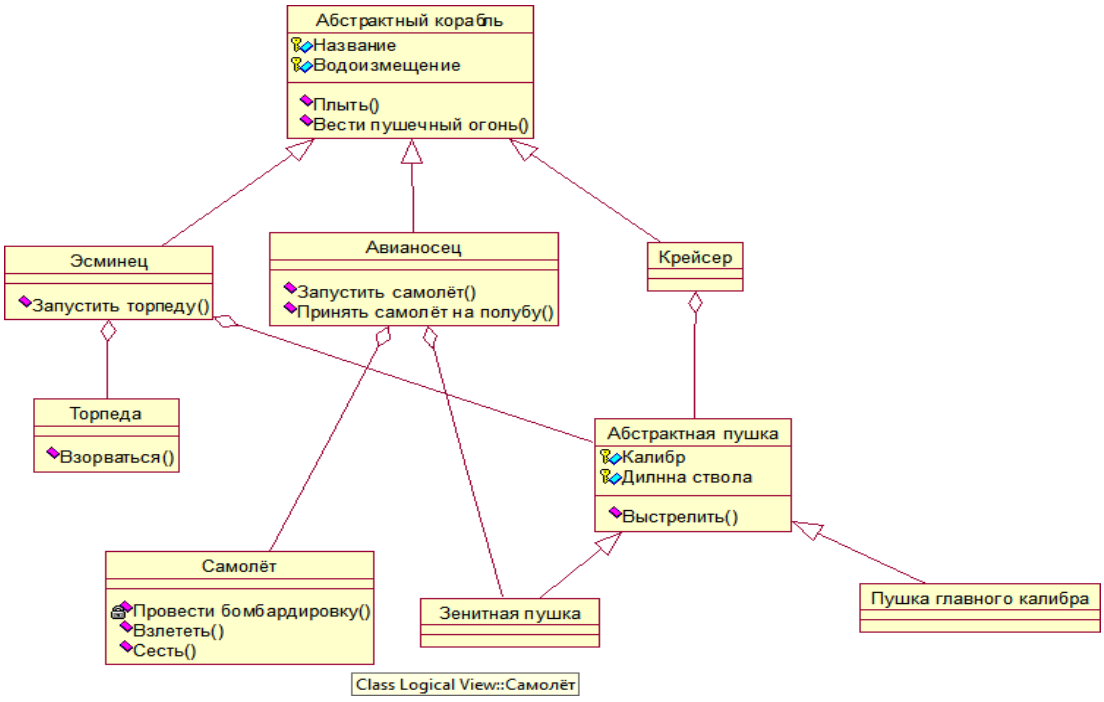

Рисунок 38 – Диаграмма классов

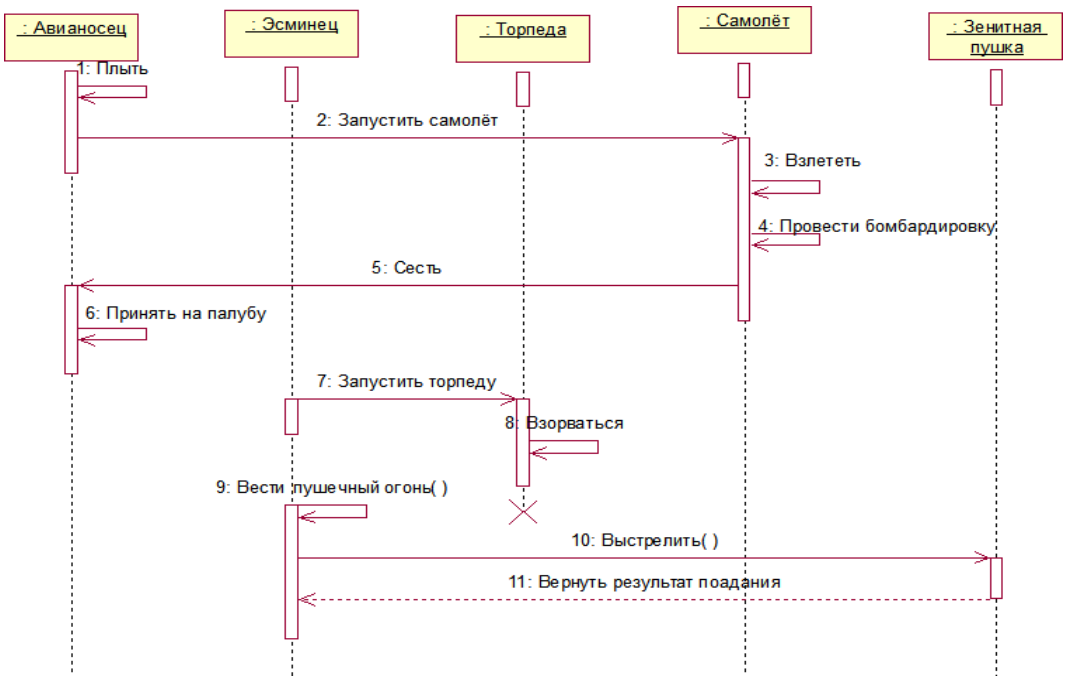

Рисунок 39 – Диаграмма последовательности

Вариант 20

Преобразовать класс Бумага. Добавить в него поле содержимое, которое будет хранить написанное на бумаге. Добавить класс Ксерокс, который будет создавать копии с бумаги.

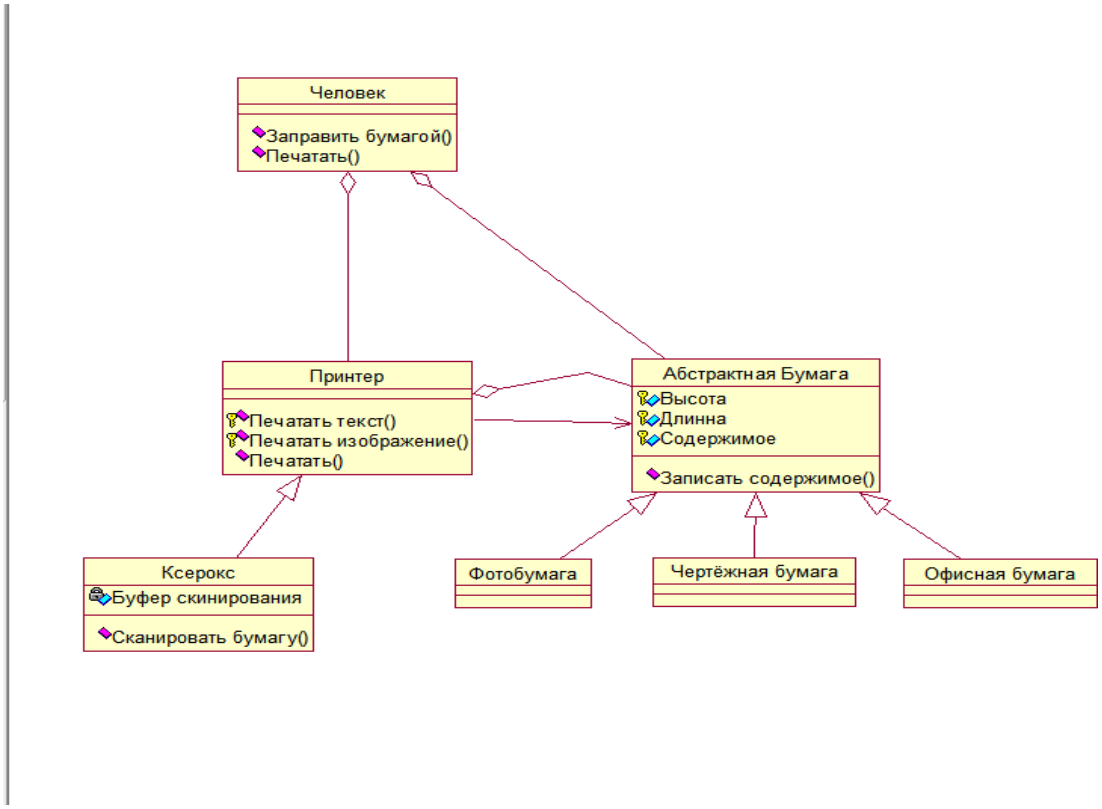

Рисунок 40 – Диаграмма классов

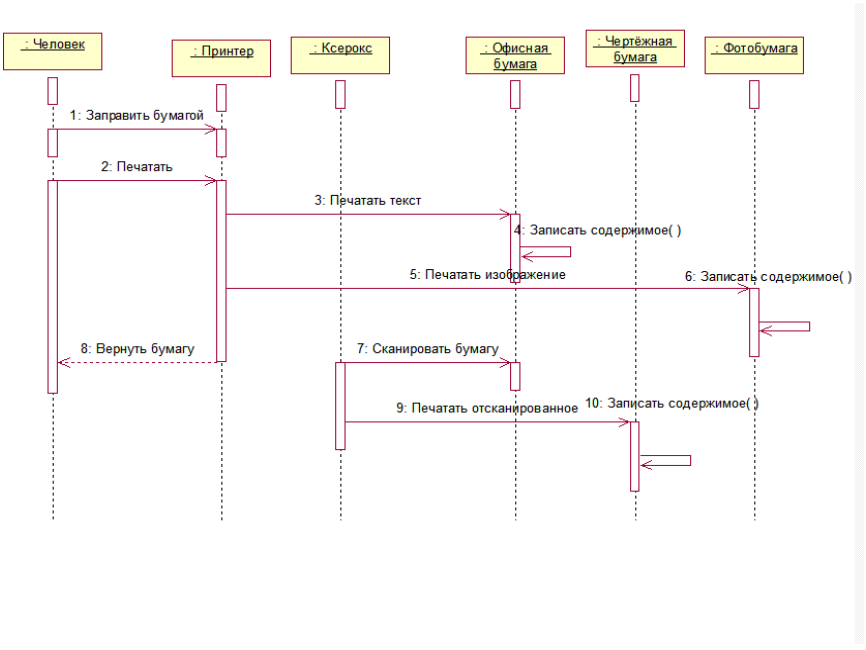

Рисунок 41 – Диаграмма последовательности

Вариант 21

Добавить класс Квартира, хранящий в себе класс Адрес и организовать взаимодействие с ним: вселение жильцов в комнату и выселение.

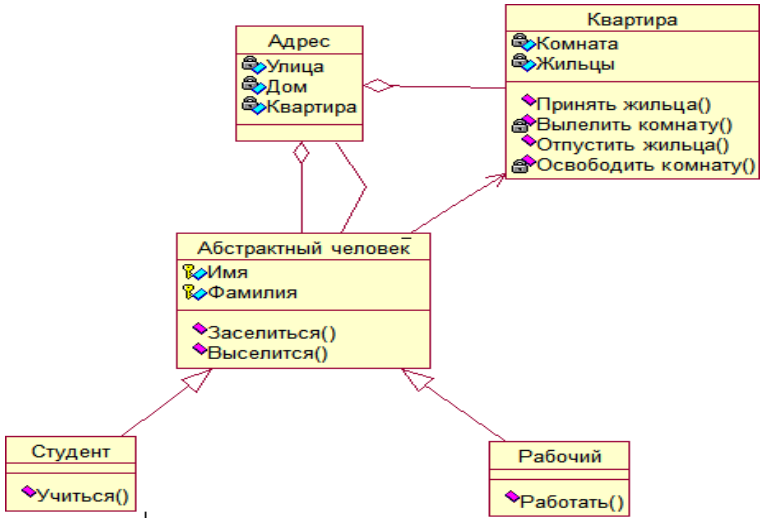

Рисунок 42 – Диаграмма классов

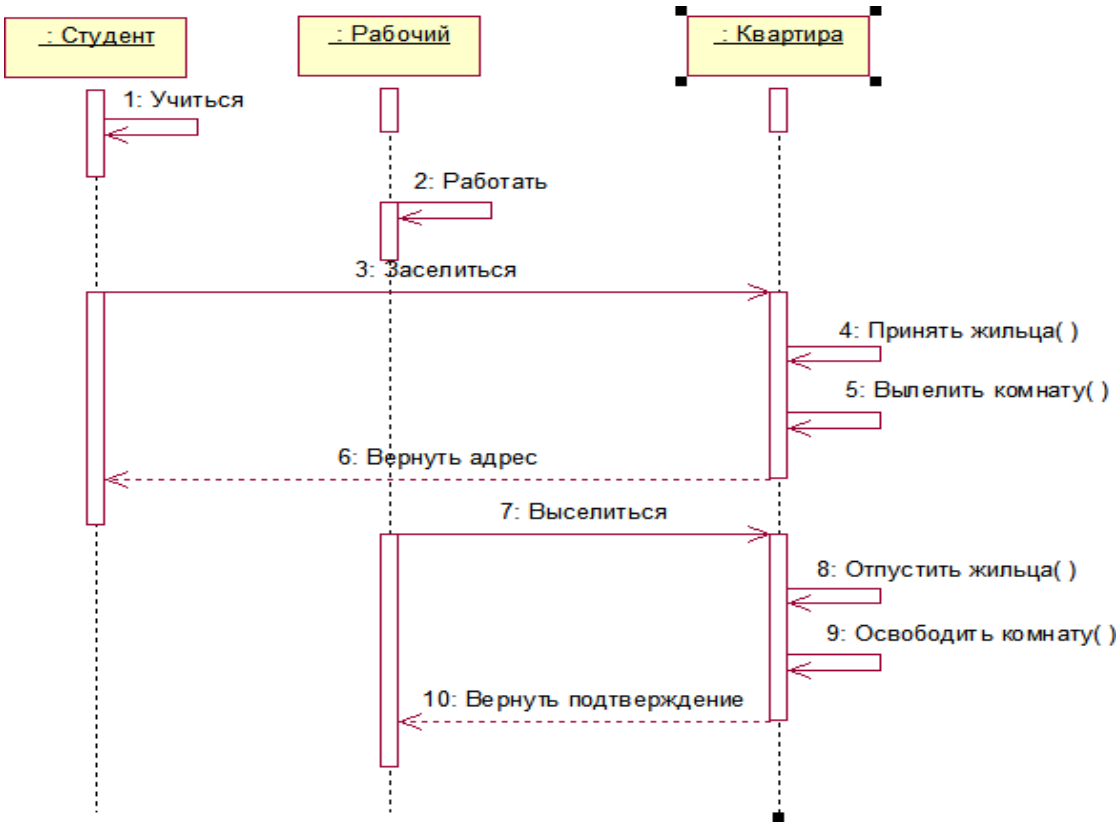

Рисунок 43 – Диаграмма последовательности

Вариант 22

Добавить классы Тигр и Фламинго по аналогии с классами Слон и Страус. Такжедобавить класс Зритель. Его метод посмотреть на животное() должен получать копию класса Животное и отображать на экран информацию о классе.

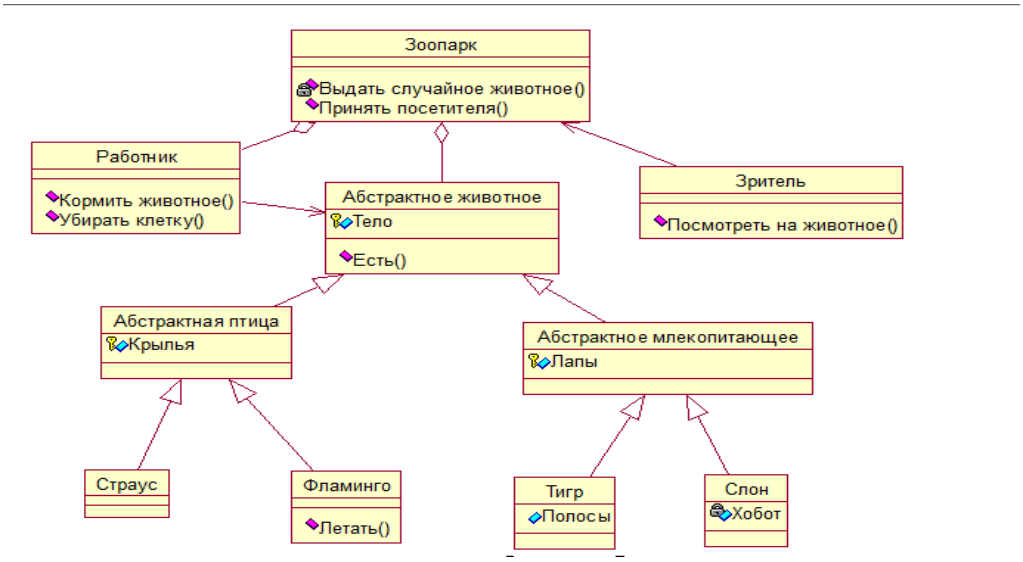

Рисунок 44 – Диаграмма классов

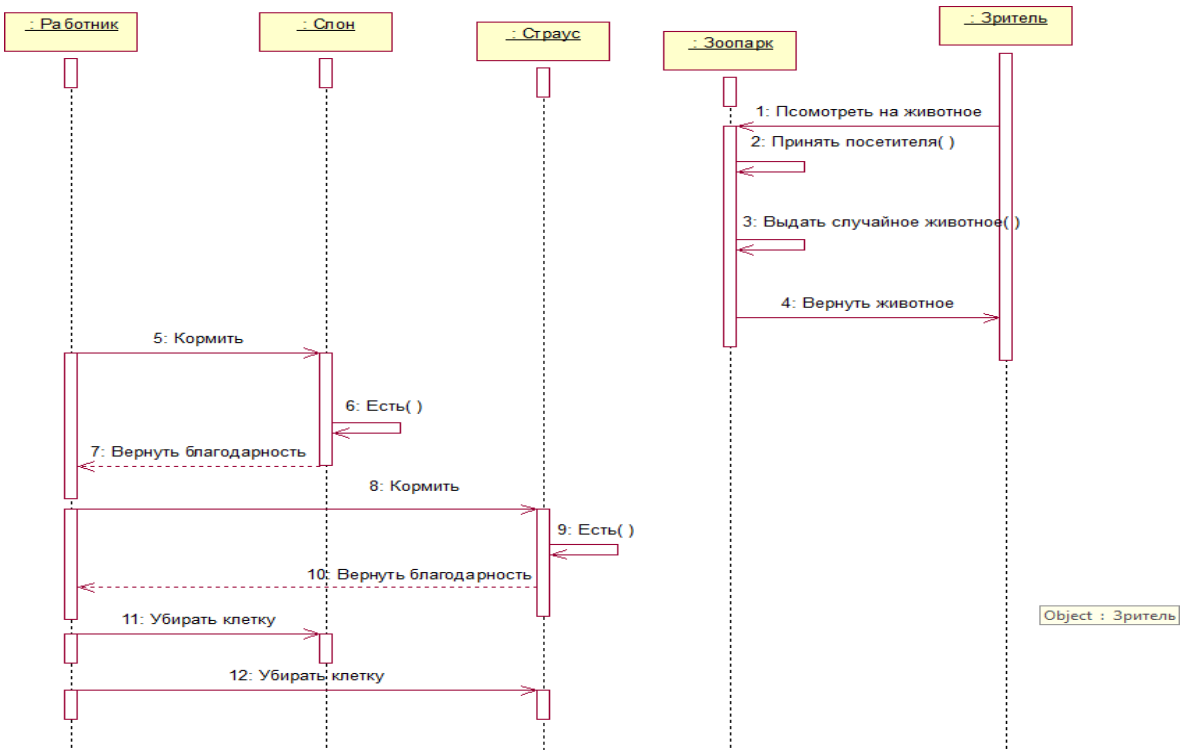

Рисунок 45 – Диаграмма последовательности

Вариант 23

Классу Университет добавить методы Провести лекцию(), Провести семинар(), Провести лабораторный практикум(). Эти методы должны менять поля знания и оценки класса Студент. Добавить класс Детский сад.

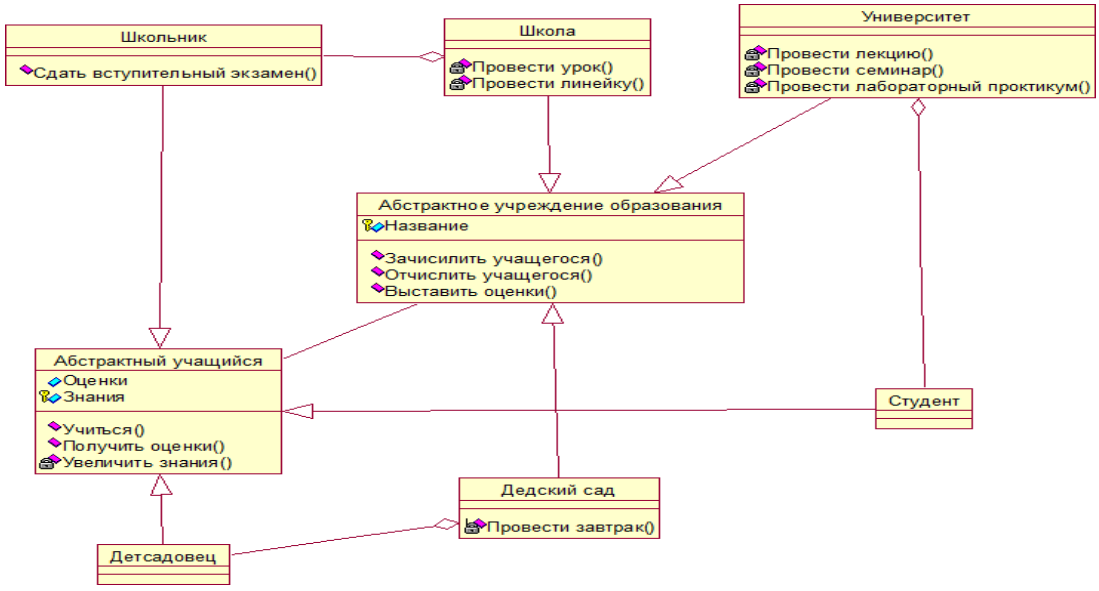

Рисунок 46 – Диаграмма классов

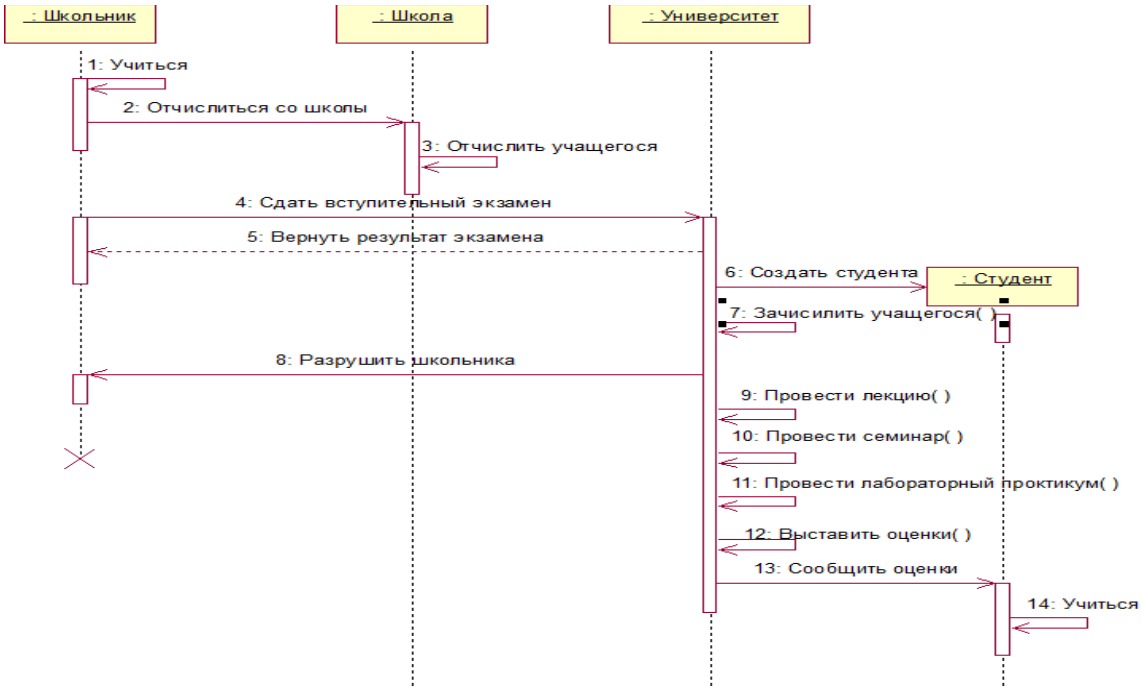

Рисунок 47 – Диаграмма последовательности

Вариант 24

Добавить класс Магазин и организовать взаимодействие класса Человек с ним с помощью метода купить жидкость(). Также добавить классы Чайник, Чай, Вода. И добавить классу Сосуд метод Перелить жидкость().

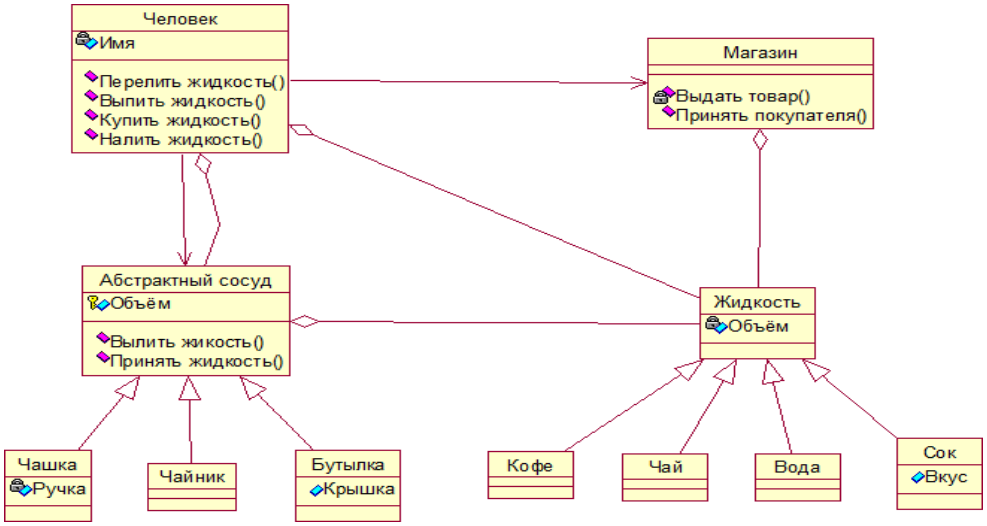

Рисунок 48 – Диаграмма классов

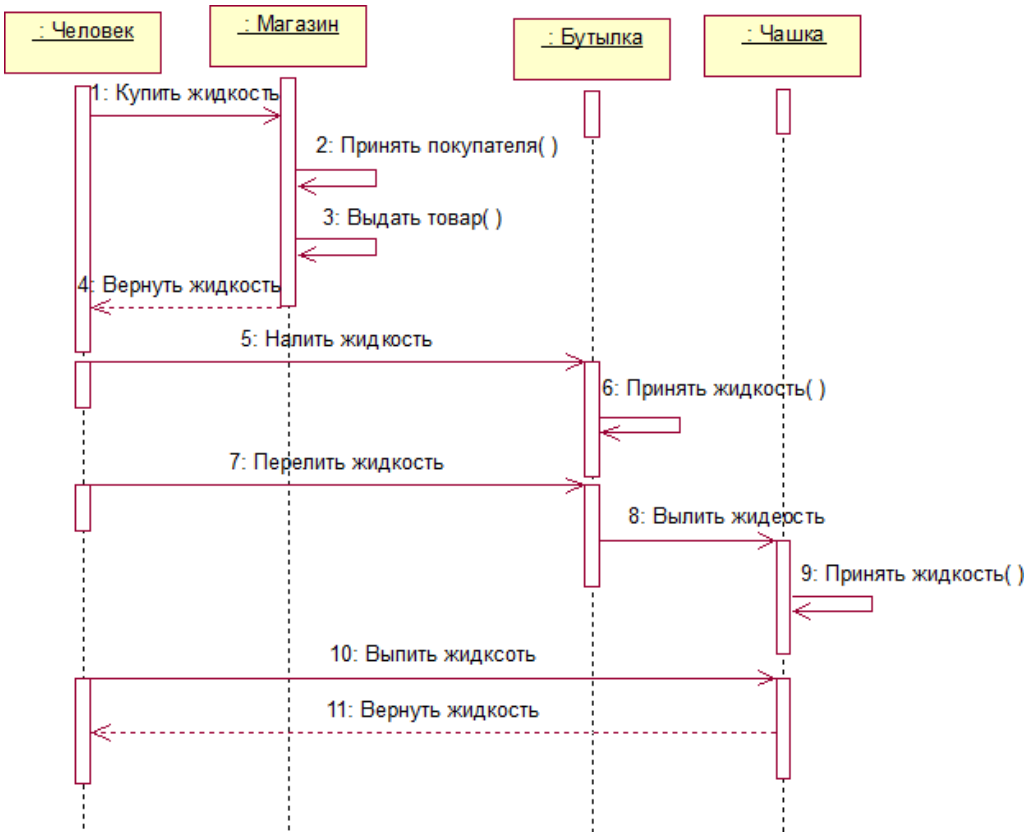

Рисунок 49 – Диаграмма последовательности

Вариант 25

Добавить класс Пакет и организовать возможность помещать туда товар. Добавить классу Тележка метод Выдать товар(). Добавить класс Молоко.

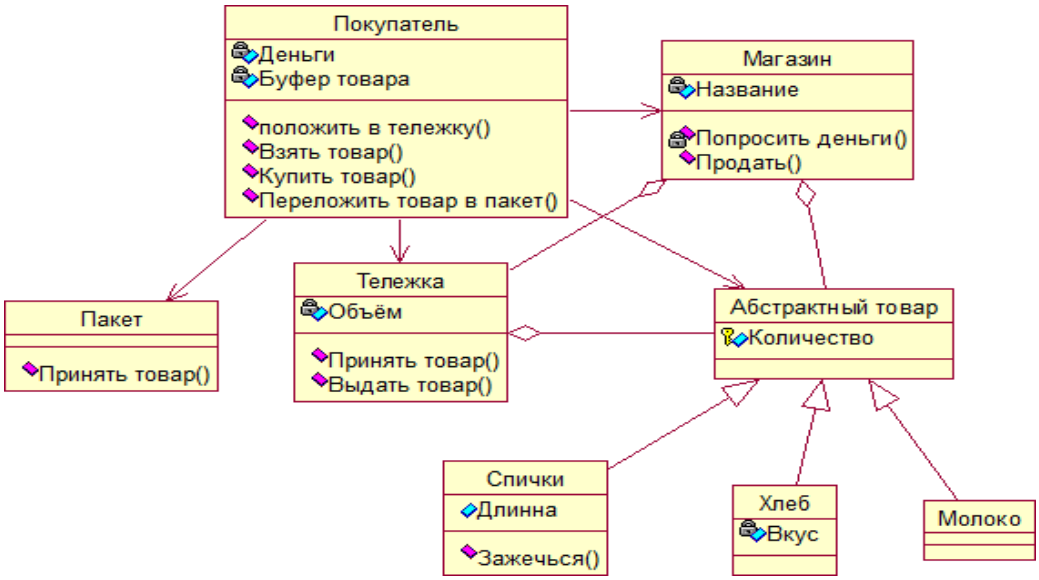

Рисунок 50 – Диаграмма классов

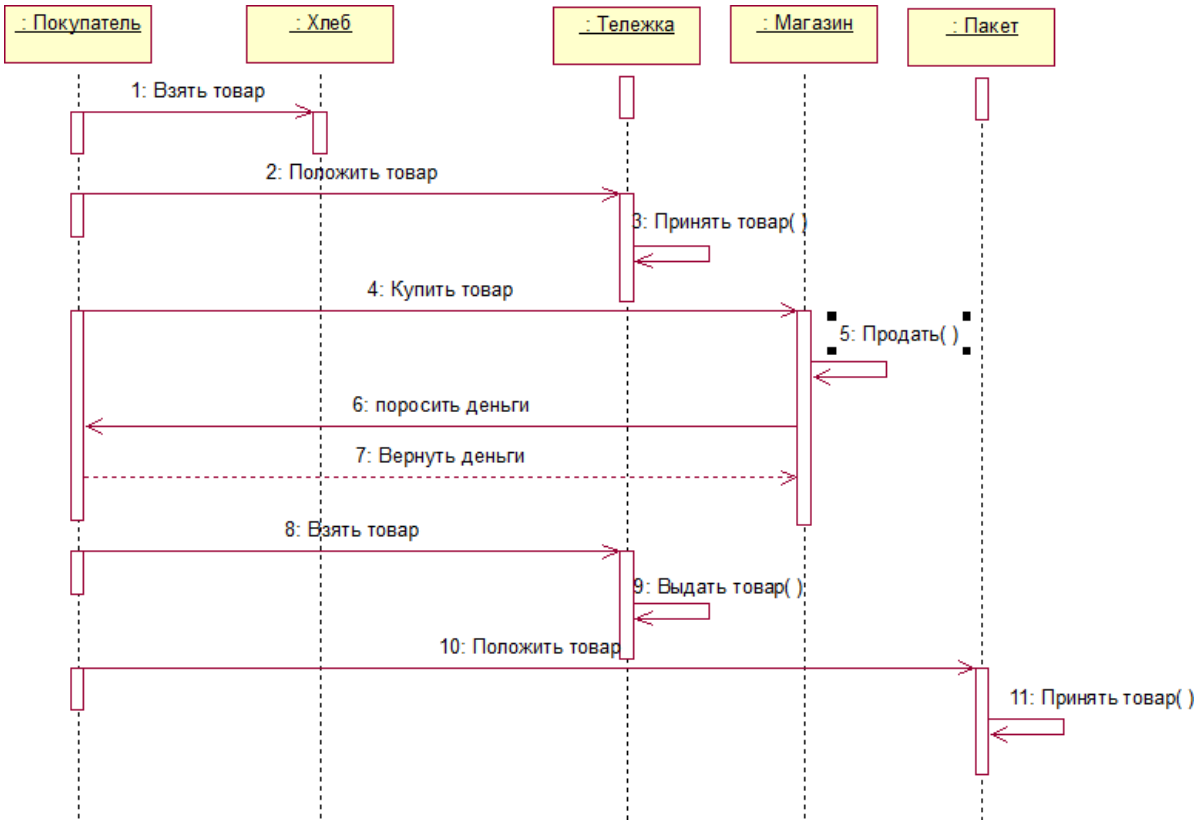

Рисунок 51 – Диаграмма последовательности

Вариант 26

Добавить классы, наследующие от класса одежда, а именно Штаны и Рубашка. Добавить классу Портной фабричные методы, создающие эти классы. Добавить классу Ткань поле количество и организовать его работу.

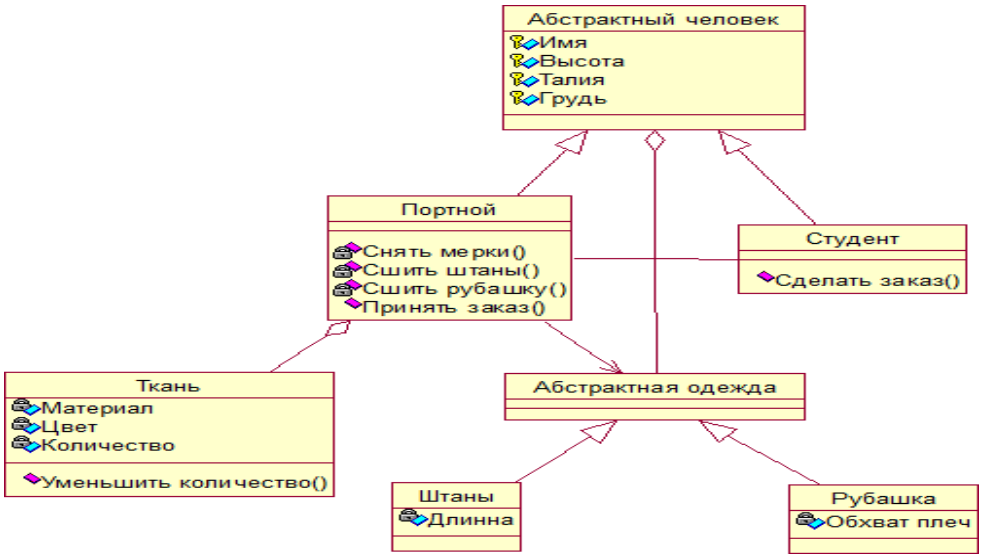

Рисунок 52 – Диаграмма классов

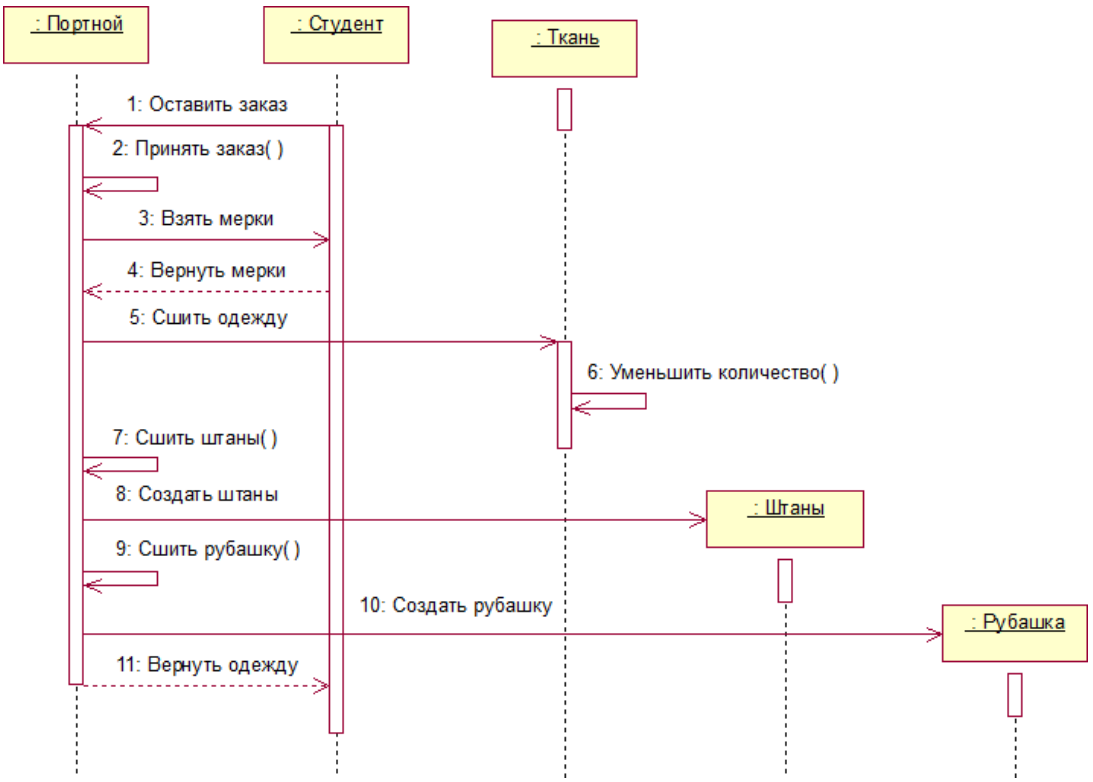

Рисунок 53 – Диаграмма последовательности

Вариант 27

Добавить монстров Кабан и Оборотень. Организовать метод убийства монстра. Переделать работу классов так, чтобы монстры сообщали квесту о своём убийстве.

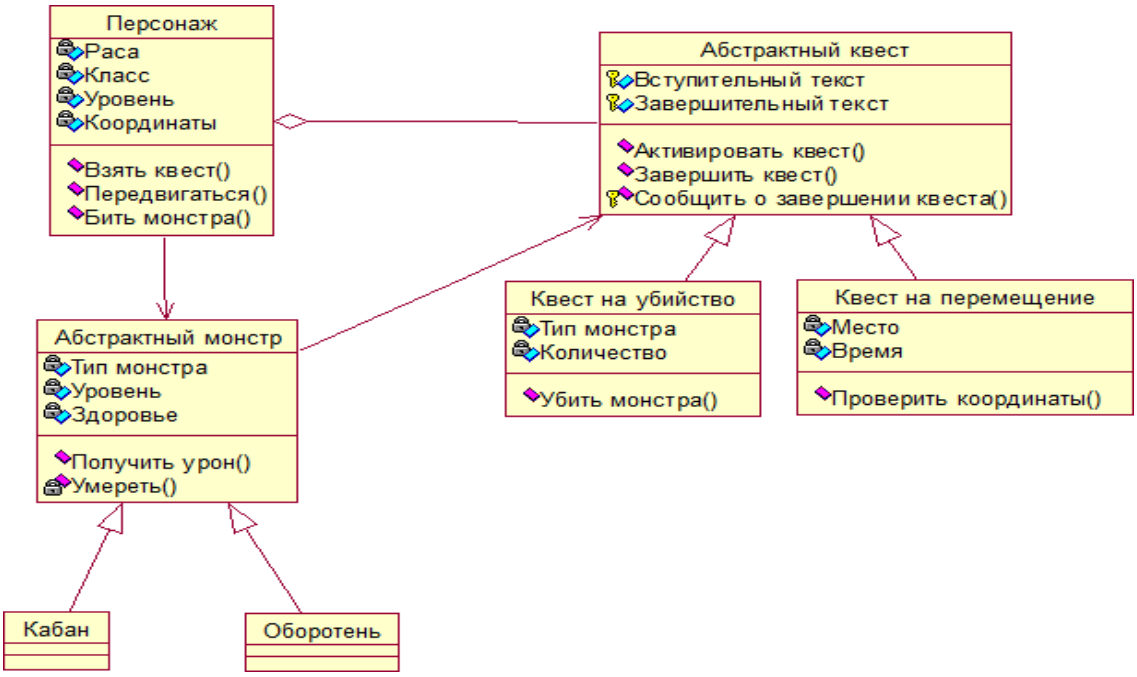

Рисунок 54 – Диаграмма классов

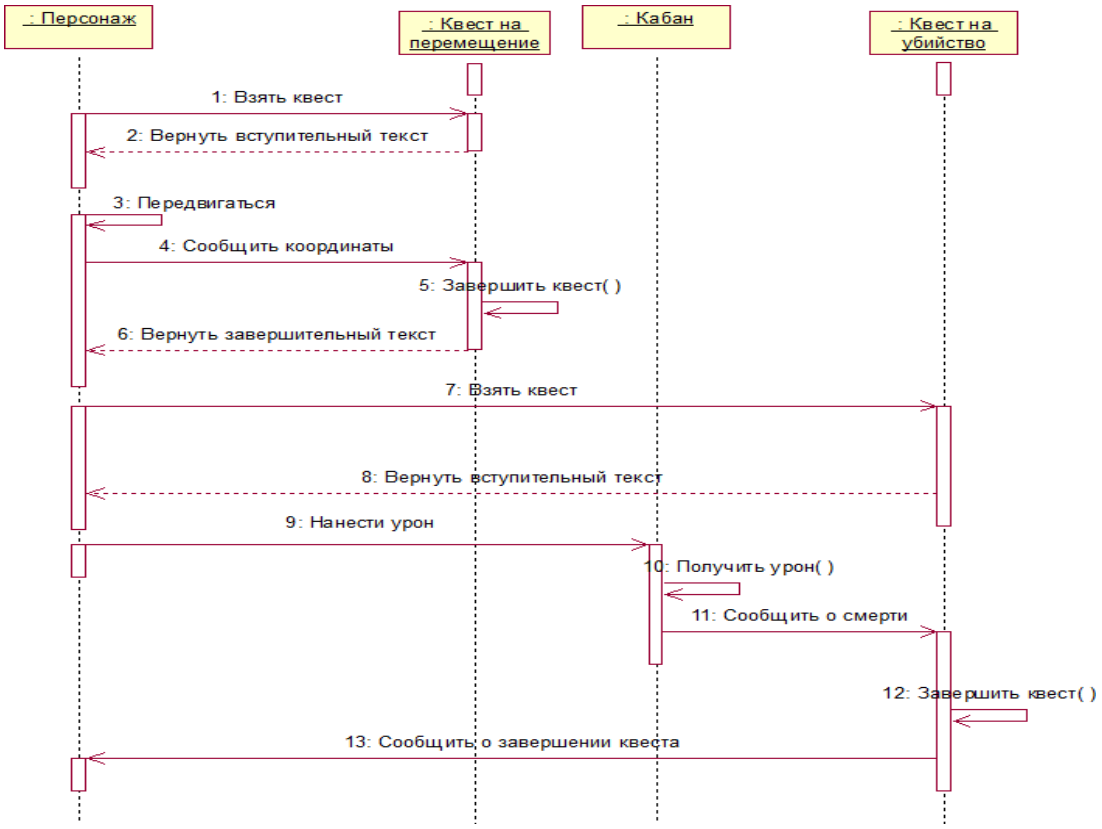

Рисунок 55 – Диаграмма последовательности

Вариант 28

Добавить класс База пользователей, хранящий данные о Зарегистрированных пользователях. Обеспечить управление этим классом с помощью класса Администратор.

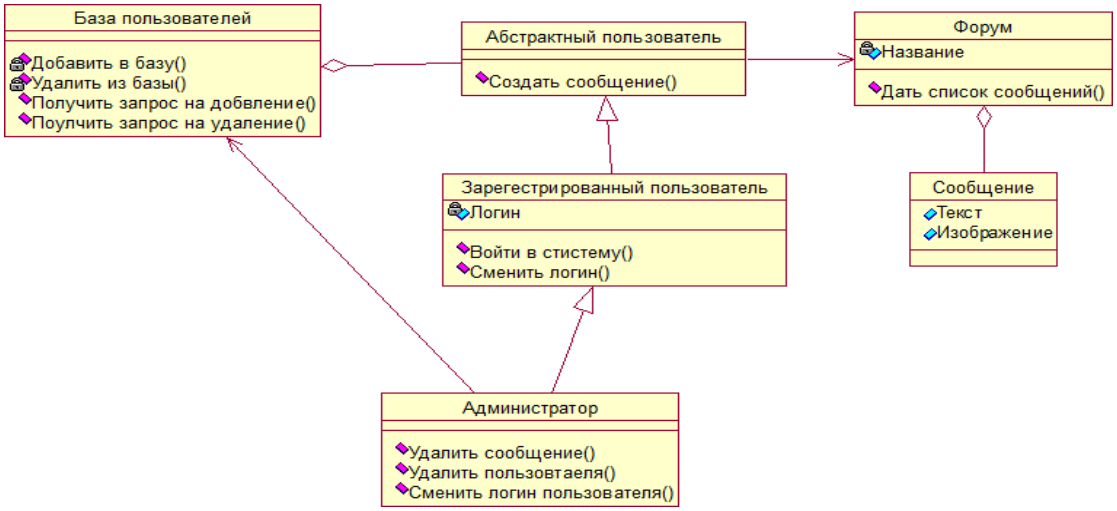

Рисунок 56 – Диаграмма классов

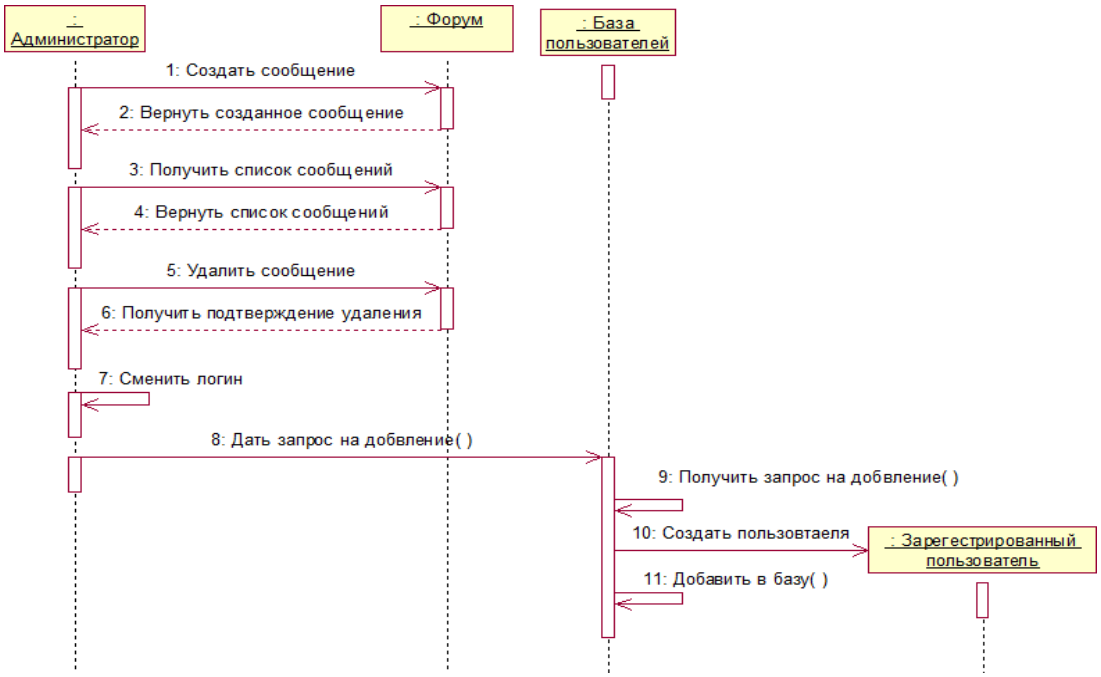

Рисунок 57 – Диаграмма последовательности

Вариант 29

Добавить возможность редактировать информацию о пользователе. В класс Пользователь добавить поля Почта, Ник. Обеспечить работу с ними.

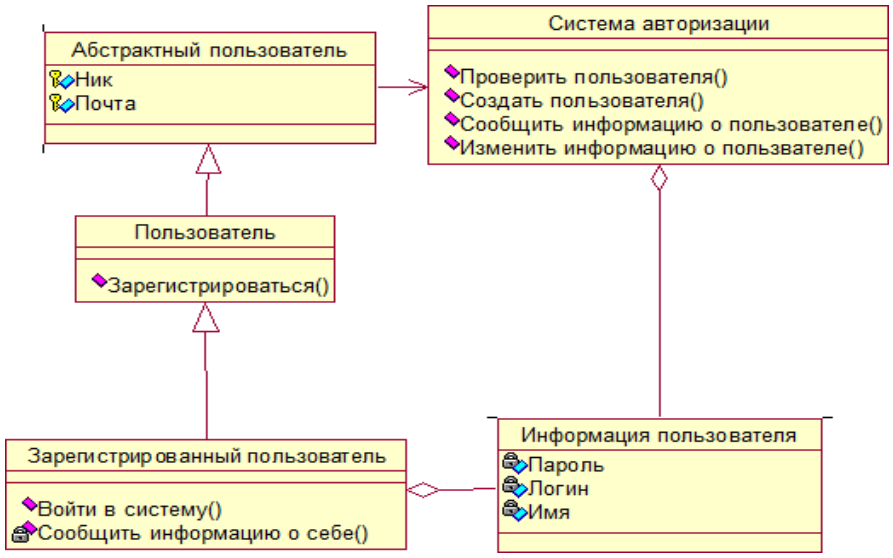

Рисунок 58 – Диаграмма классов

Рисунок 59 – Диаграмма последовательности

Вариант 30

Добавить класс Плейлист, хранящий список видео. Класс должен выдавать следующее видео из списка при вызове метода Воспроизвести плейлист(). Добавить класс Музыкальное видео.

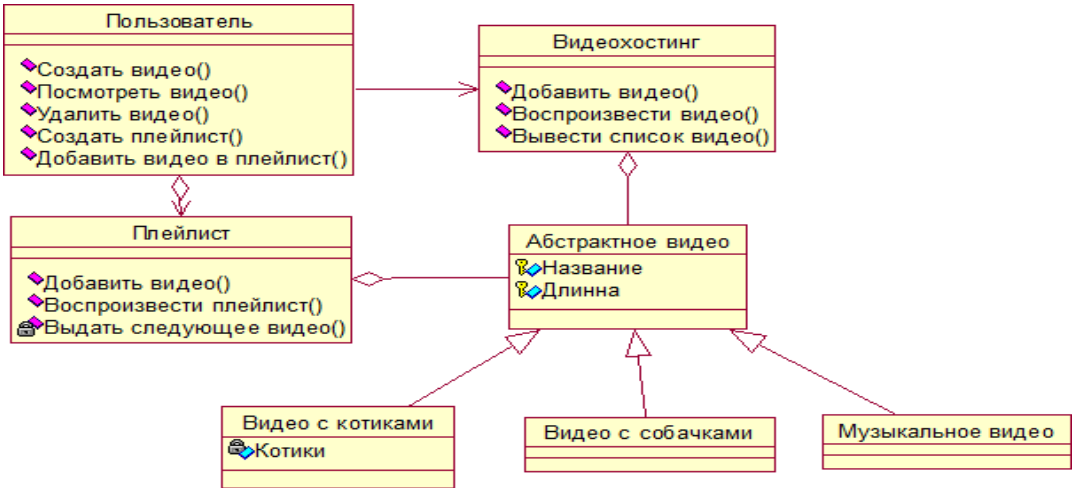

Рисунок 60 – Диаграмма классов

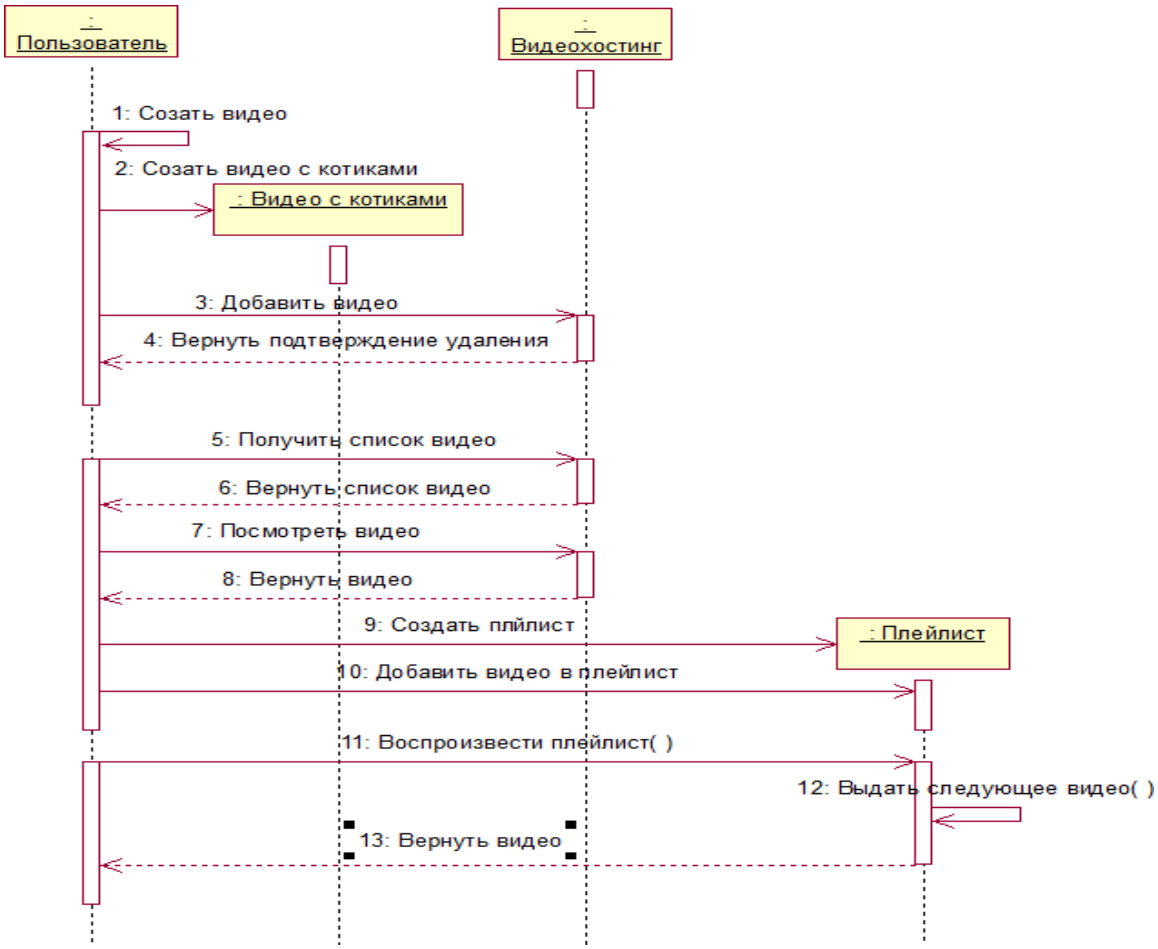

Рисунок 61 – Диаграмма последовательности