Здесь будет титульник, листай ниже

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Настоящая курсовая работа выполнена в соответствии с требованиями ГОСТ Единой системы программной документации (ЕСПД) [1]. Все этапы решения задач курсовой работы фиксированы, соответствуют требованиям, приведенным в методическом пособии для выполнения практических заданий, контрольных и курсовых работ по дисциплине «Объектно-ориентированное программирование» [2-3] и методике разработки объектно-ориентированных программ [4-6].

Изучение объектно-ориентированного программирования (ООП) имеет несколько важных причин и предлагает ряд преимуществ:

1. Широкое применение: ООП является одним из наиболее распространенных подходов к разработке программного обеспечения. Множество языков программирования, таких как Java, C++, Python и C#, используют ООП-парадигму. Изучение ООП позволяет разработчику быть готовым к работе с этими языками и использовать их возможности на практике;
2. Модульность и повторное использование кода: ООП способствует созданию модульного кода, который легко понять, поддерживать и расширять. Изучение ООП помогает разработчикам понять принципы инкапсуляции, наследования и полиморфизма, которые позволяют повторно использовать код, улучшать его читаемость и облегчать сопровождение;
3. Абстракция и моделирование предметной области: ООП позволяет создавать абстракции, которые моделируют реальные объекты и концепции. Это помогает разработчикам лучше понять предметную область и перевести ее в программный код. Изучение ООП развивает способность абстрагироваться от деталей реализации и фокусироваться на важных

аспектах системы;

4. Расширяемость и гибкость: ООП позволяет разрабатывать гибкие и расширяемые системы. Изучение ООП помогает разработчикам понять, как создавать классы и иерархии классов, которые могут быть легко расширены новыми функциями или изменены без необходимости переписывания существующего кода;

5. Командная разработка: ООП облегчает командную разработку, поскольку код может быть разделен на модули и классы, которые разные разработчики могут разрабатывать независимо. Изучение ООП помогает разработчикам понять, как организовать свой код, чтобы он был более доступным и легко поддерживаемым для других членов команды;

6. Поддержка современных технологий и фреймворков: Множество современных технологий и фреймворков, таких как разработка веб-приложений, мобильная разработка и анализ данных, основаны на ООП-языках и парадигме. Изучение ООП помогает разработчикам в освоении этих современных инструментов и создании высококачественного программного обеспечения.

Изучение объектно-ориентированного программирования не только позволяет разработчикам овладеть важными навыками и принципами, но и открывает двери для понимания и использования широкого спектра современных технологий и платформ.

1 ПОСТАНОВКА ЗАДАЧИ

Реализовать механизм взаимодействия объектов с использованием сигналов и обработчиков, с передачей вместе сигналом текстового сообщения (строковой переменной).

Для организации взаимосвязи по механизму сигналов и обработчиков в базовый класс добавить три метода:

- установления связи между сигналом текущего объекта и обработчиком целевого объекта;
- удаления (разрыва) связи между сигналом текущего объекта и обработчиком целевого объекта;
- выдачи сигнала от текущего объекта с передачей строковой переменной.

Включенный объект может выдать или обработать сигнал.

Методу установки связи передать указатель на метод сигнала текущего объекта, указатель на целевой объект и указатель на метод обработчика целевого объекта.

Методу удаления (разрыва) связи передать указатель на метод сигнала текущего объекта, указатель на целевой объект и указатель на метод обработчика целевого объекта.

Методу выдачи сигнала передать указатель на метод сигнала и строковую переменную. В данном методе реализовать алгоритм:

1. Если текущий объект отключен, то выход, иначе к пункту 2.
2. Вызов метода сигнала с передачей строковой переменной по ссылке.
3. Цикл по всем связям сигнал-обработчик текущего объекта:

    3.1. Если в очередной связи сигнал-обработчик участвует метод сигнала, переданный по параметру, то проверить готовность целевого объекта. Если целевой объект готов, то вызвать метод обработчика

целевого объекта указанной в связи и передать в качестве аргумента строковую переменную по значению.

4. Конец цикла.

Для приведения указателя на метод сигнала и на метод обработчика использовать параметризированное макроопределение препроцессора.

В базовый класс добавить метод определения абсолютной пути до текущего объекта. Этот метод возвращает абсолютный путь текущего объекта.

Состав и иерархия объектов строится посредством ввода исходных данных. Ввод организован как в версии № 3 курсовой работы. Если при построении дерева иерархии возникает ситуация дубляжа имен среди починенных у текущего головного объекта, то новый объект не создается.

Система содержит объекты шести классов с номерами: 1, 2, 3, 4, 5, 6. Классу корневого объекта соответствует номер 1. В каждом производном классе реализовать один метод сигнала и один метод обработчика.

Каждый метод сигнала с новой строки выводит:

```
Signal from «абсолютная координата объекта»
```

Каждый метод сигнала добавляет переданной по параметру строке текста номер класса принадлежности текущего объекта по форме:

```
«пробел»(class: «номер класса»)
```

Каждый метод обработчика с новой строки выводит:

```
Signal to «абсолютная координата объекта»   Text: «переданная строка»
```

Моделировать работу системы, которая выполняет следующие команды с параметрами:

- EMIT «координата объекта» «текст» – выдает сигнал от заданного по координате объекта;
- SET_CONNECT «координата объекта выдающего сигнал» «координата

целевого объекта» – устанавливает связь;

- DELETE_CONNECT «координата объекта выдающего сигнал» «координата целевого объекта» – удаляет связь;
- SET_CONDITION «координата объекта» «значение состояния» – устанавливает состояние объекта.
- END – завершает функционирование системы (выполнение программы).

Реализовать алгоритм работы системы:

- в методе построения системы:
  - построение дерева иерархии объектов согласно вводу;
  - ввод и построение множества связей сигнал-обработчик для заданных пар объектов.
- в методе отработки системы:
  - привести все объекты в состоянии готовности;
  - цикл до признака завершения ввода:
    - ввод наименования объекта и текста сообщения;
    - вызов сигнала заданного объекта и передача в качестве аргумента строковой переменной, содержащей текст сообщения.
  - конец цикла.

Допускаем, что все входные данные вводятся синтаксически корректно. Контроль корректности входных данных можно реализовать для самоконтроля работы программы. Не оговоренные, но необходимые функции и элементы классов добавляются разработчиком.

1.1 Описание входных данных

В методе построения системы.

Множество объектов, их характеристики и расположение на дереве

иерархии. Структура данных для ввода согласно изложенному в версии № 3 курсовой работы.

После ввода состава дерева иерархии построчно вводится:

```
«координата объекта выдающего сигнал» «координата целевого объекта»
```

Ввод информации для построения связей завершается строкой, которая содержит:

```
«end_of_connections»
```

В методе запуска (отработки) системы построчно вводятся множество команд в производном порядке:

- EMIT «координата объекта» «текст» – выдать сигнал от заданного по координате объекта;
- SET_CONNECT «координата объекта выдающего сигнал» «координата целевого объекта» – установка связи;
- DELETE_CONNECT «координата объекта выдающего сигнал» «координата целевого объекта» – удаление связи;
- SET_CONDITION «координата объекта» «значение состояния» – установка состояния объекта.
- END – завершить функционирование системы (выполнение программы).

Команда END присутствует обязательно.

Если координата объекта задана некорректно, то соответствующая операция не выполняется и с новой строки выдается сообщение об ошибке.

Если не найден объект по координате:

```
Object «координата объекта» not found
```

Если не найден целевой объект по координате:

```
Handler object «координата целевого объекта» not found
```

Пример ввода:

```
appls_root
/ object_s1 3
/ object_s2 2
/object_s2 object_s4 4
/ object_s13 5
/object_s2 object_s6 6
/object_s1 object_s7 2
endtree
/object_s2/object_s4 /object_s2/object_s6
/object_s2 /object_s1/object_s7
/ /object_s2/object_s4
/object_s2/object_s4 /
end_of_connections
EMIT /object_s2/object_s4 Send message 1
EMIT /object_s2/object_s4 Send message 2
EMIT /object_s2/object_s4 Send message 3
EMIT /object_s1 Send message 4
END
```

1.2 Описание выходных данных

Первая строка:

```
Object tree
```

Со второй строки вывести иерархию построенного дерева.

Далее, построчно, если отработал метод сигнала:

```
Signal from «абсолютная координата объекта»
```

Если отработал метод обработчика:

```
Signal to «абсолютная координата объекта»   Text: «переданная строка»
```

Пример вывода:

```
Object tree
appls_root
    object_s1
        object_s7
    object_s2
        object_s4
        object_s6
    object_s13
Signal from /object_s2/object_s4
Signal to /object_s2/object_s6 Text:  Send message 1 (class: 4)
Signal to / Text:  Send message 1 (class: 4)
Signal from /object_s2/object_s4
```

```
Signal to /object_s2/object_s6 Text:  Send message 2 (class: 4)
Signal to / Text:  Send message 2 (class: 4)
Signal from /object_s2/object_s4
Signal to /object_s2/object_s6 Text:  Send message 3 (class: 4)
Signal to / Text:  Send message 3 (class: 4)
Signal from /object_s1
```

2 МЕТОД РЕШЕНИЯ

1. Класс cl_base:

- Поля
    - Целочисленное поле number
        - Тип данных - целый тип
        - Название - number
        - Модификатор доступа - public
        - Вектор connects структур oSh
            - Тип данных - вектор структур oSh
            - Название - connects
            - Модификатор доступа - public
    - Методы
        - Метод setConnect
            - Функционал - устанавливает связь между двумя объектами
        - Метод deleteConnect
            - Функционал - удаляет связь между двумя объектами
        - Метод emitSignal
            - Функционал - отправляет сообщение от одного объекта к другому
        - Метод setFullReadiness
            - Функционал - устанавливает готовность на все деревья в дереве

2. Класс cl_application:

- Методы
    - Метод setConnections

        - Функционал - обработка установления связи между объектами
    - Метод handleCommands
        - Функционал - обработка ввода команд

3. Классы cl_1 по класс cl_6:

- Методы
    - Конструктор cl_1 по cl_6
        - Функционал - создание объекта и присвоение полю number номера текущего класса
    - Метод signal
        - Функционал - отправление сообщения
    - Метод handler
        - Функционал - принятие сообщения

4. Структура oSh:

- Поля
    - Поле pSignal
        - Тип данных - пользовательский тип TYPE_SIGNAL
        - Модификатор - public
        - Название - pSignal
    - Поле pClObject
        - Тип данных - указатель на объект класса cl_base
        - Модификатор - public
        - Название - pClObject
    - Поле pHandler
        - Тип данных - пользовательский тип TYPE_HANDLER
        - Модификатор - public
        - Название - pHandler

3 ОПИСАНИЕ АЛГОРИТМОВ

Согласно этапам разработки, после определения необходимого инструментария в разделе «Метод», составляются подробные описания алгоритмов для методов классов и функций.

3.1 Алгоритм метода setConnect класса cl_base

Функционал: устанавливает связь между двумя объектами.

Параметры: пользовательский тип TYPE_SIGNAL, указатель на объект класса cl_base, пользовательский тип TYPE_HANDLER.

Возвращаемое значение: отсутствует.

Алгоритм метода представлен в таблице 1.

Таблица 1 – Алгоритм метода setConnect класса cl_base

№	Предикат	Действия	№ перехода
1		инициализация pValue указателя на структуру oSh	2
2	i меньше длины вектора connects		3
			4
3	все поля текущей связи совпадают с переданными параметрами		∅
			2
4		присвоение pValue значения указателя на структуру oSh	5
5		присвоение полю p_signal значения параметра p_signal	6
6		присвоение полю p_handler значения параметра p_handler	7

№	Предикат	Действия	№ перехода
7		присвоение полю p_target значения параметра p_target	8
8		добавление в вектор connects переменной p_value	∅

3.2 Алгоритм метода deleteConnect класса cl_base

Функционал: удаляет связь между двумя объектами.

Параметры: пользовательский тип TYPE_SIGNAL, указатель на объект класса cl_base, пользовательский тип TYPE_HANDLER.

Возвращаемое значение: отсутствует.

Алгоритм метода представлен в таблице 2.

Таблица 2 – Алгоритм метода deleteConnect класса cl_base

№	Предикат	Действия	№ перехода
1		создание итератора p_it	2
2	i меньше длины вектора connects		3
			∅
3	все поля текущей связи совпадают с переданными параметрами	удаление из списка connects текущей связи	2
			2

3.3 Алгоритм метода emitSignal класса cl_base

Функционал: отправка сообщения от одного объекта к другому.

Параметры: пользовательский тип TYPE_SIGNAL, адрес строкового типа.

Возвращаемое значение: отсутствует.

Алгоритм метода представлен в таблице 3.

Таблица 3 – Алгоритм метода emitSignal класса cl_base

№	Предикат	Действия	№ перехода
1		вызываем указанный сигнал, передавая ему сообщение s_message	1
2	начинаем перебор всех соединений, которые установлены для данного объекта		2
			3
3	проверяем, соответствует ли текущее соединение вызванному сигналу		4
			∅
4		сохраняем указатель на объект-получатель, который был связан с данным соединением	5
5		сохраняем указатель на метод-обработчик, который был связан с данным соединением	6
6		вызываем метод-обработчик на объекте-получателе, передавая ему сообщение s_message	∅

3.4 Алгоритм метода setConnections класса cl_application

Функционал: управляет установкой связью между объектов.

Параметры: отсутствуют.

Возвращаемое значение: отсутствует.

Алгоритм метода представлен в таблице 4.

Таблица 4 – Алгоритм метода setConnections класса cl_application

№	Предикат	Действия	№ перехода
1		инициализация строковой переменной sender_coordinatesenderCoord	2
2		инициализация строковой переменной receiverCoord	3
3		инициализация указателя на объект класса cl_base p_sender	4
4		инициализация указателя на объект класса cl_base p_reciever	5
5		создание списка методов сигнала всех классов	6
6		создание списка обработчиков сигнала всех классов	7
7		ввод переменной senderCoord	8
8	senderCoord равно "end_of_connections"		∅
			9
9		ввод переменной receiverCoord	10
10		присвоение p_sender значения вызова метода find_obj_by_coord	11
11		присвоение p_reciever значения вызова метода find_obj_by_coord	12
12		присвоение переменной signal значения элемента списка SIGNALS_LIST под индексом текущего класса минус 1	13
13		присвоение переменной handler значения элемента списка HANDLERS_LIST под индексом текущего класса минус 1	14
14		вызов метода set_connection от p_sender	7

3.5 Алгоритм конструктора класса cl_1

Функционал: параметризированный конструктор объекта.

Параметры: указатель на объект класса cl_base, строка.

Алгоритм конструктора представлен в таблице 5.

Таблица 5 – Алгоритм конструктора класса cl_1

№	Предикат	Действия	№ перехода
1		присвоение полю cl_number номера текущего класса	∅

3.6 Алгоритм конструктора класса cl_2

Функционал: параметризированный конструктор объекта.

Параметры: указатель на объект класса cl_base, строка.

Алгоритм конструктора представлен в таблице 6.

Таблица 6 – Алгоритм конструктора класса cl_2

№	Предикат	Действия	№ перехода
1		присвоение полю cl_number номера текущего класса	∅

3.7 Алгоритм конструктора класса cl_3

Функционал: параметризированный конструктор объекта.

Параметры: указатель на объект класса cl_base, строка.

Алгоритм конструктора представлен в таблице 7.

Таблица 7 – Алгоритм конструктора класса cl_3

№	Предикат	Действия	№ перехода
1		присвоение полю cl_number номера текущего класса	∅

3.8 Алгоритм конструктора класса cl_4

Функционал: параметризированный конструктор объекта.

Параметры: указатель на объект класса cl_base, строка.

Алгоритм конструктора представлен в таблице 8.

Таблица 8 – Алгоритм конструктора класса cl_4

№	Предикат	Действия	№ перехода
1		присвоение полю cl_number номера текущего класса	∅

3.9 Алгоритм конструктора класса cl_5

Функционал: параметризированный конструктор объекта.

Параметры: указатель на объект класса cl_base, строка.

Алгоритм конструктора представлен в таблице 9.

Таблица 9 – Алгоритм конструктора класса cl_5

№	Предикат	Действия	№ перехода
1		присвоение полю cl_number номера текущего класса	∅

3.10 Алгоритм конструктора класса cl_6

Функционал: параметризированный конструктор объекта.

Параметры: указатель на объект класса cl_base, строка.

Алгоритм конструктора представлен в таблице 10.

Таблица 10 – Алгоритм конструктора класса cl_6

№	Предикат	Действия	№ перехода
1		присвоение полю cl_number номера текущего класса	∅

3.11 Алгоритм метода signal класса cl_1

Функционал: метод сигнала.

Параметры: адрес строковой переменной.

Возвращаемое значение: отсутствует.

Алгоритм метода представлен в таблице 11.

Таблица 11 – Алгоритм метода signal класса cl_1

№	Предикат	Действия	№ перехода
1		вывод на экран "Signal from" координата текущего объекта	2
2		добавление к строке текста "class: " номер класса текущего объекта	∅

3.12 Алгоритм метода signal класса cl_2

Функционал: метод сигнала.

Параметры: адрес строковой переменной.

Возвращаемое значение: отсутствует.

Алгоритм метода представлен в таблице 12.

Таблица 12 – Алгоритм метода signal класса cl_2

№	Предикат	Действия	№ перехода
1		вывод на экран "Signal from" координата текущего объекта	2
2		добавление к строке текста "class: " номер класса текущего объекта	∅

3.13 Алгоритм метода signal класса cl_3

Функционал: метод сигнала.

Параметры: адрес строковой переменной.

Возвращаемое значение: отсутствует.

Алгоритм метода представлен в таблице 13.

Таблица 13 – Алгоритм метода signal класса cl_3

№	Предикат	Действия	№ перехода
1		вывод на экран "Signal from" координата текущего объекта	2
2		добавление к строке текста "class: " номер класса текущего объекта	∅

3.14 Алгоритм метода signal класса cl_4

Функционал: метод сигнала.

Параметры: адрес строковой переменной.

Возвращаемое значение: отсутствует.

Алгоритм метода представлен в таблице 14.

Таблица 14 – Алгоритм метода signal класса cl_4

№	Предикат	Действия	№ перехода
1		вывод на экран "Signal from" координата текущего объекта	2
2		добавление к строке текста "class: " номер класса текущего объекта	∅

3.15 Алгоритм метода signal класса cl_5

Функционал: метод сигнала.

Параметры: адрес строковой переменной.

Возвращаемое значение: отсутствует.

Алгоритм метода представлен в таблице 15.

Таблица 15 – Алгоритм метода signal класса cl_5

№	Предикат	Действия	№ перехода
1		вывод на экран "Signal from" координата текущего объекта	2
2		добавление к строке текста "class: " номер класса текущего объекта	∅

3.16 Алгоритм метода signal класса cl_6

Функционал: метод сигнала.

Параметры: адрес строковой переменной.

Возвращаемое значение: отсутствует.

Алгоритм метода представлен в таблице 16.

Таблица 16 – Алгоритм метода signal класса cl_6

№	Предикат	Действия	№ перехода
1		вывод на экран "Signal from" координата текущего объекта	2
2		добавление к строке текста "class: " номер класса текущего объекта	∅

3.17 Алгоритм метода handler класса cl_1

Функционал: обработчик сигнала.

Параметры: строка.

Возвращаемое значение: отсутствует.

Алгоритм метода представлен в таблице 17.

Таблица 17 – Алгоритм метода handler класса cl_1

№	Предикат	Действия	№ перехода
1		вывод на экран "Signal to " координата текущего объекта "Text: " значение параметра	∅

3.18 Алгоритм метода handler класса cl_2

Функционал: обработчик сигнала.

Параметры: строка.

Возвращаемое значение: отсутствует.

Алгоритм метода представлен в таблице 18.

Таблица 18 – Алгоритм метода handler класса cl_2

№	Предикат	Действия	№ перехода
1		вывод на экран "Signal to " координата текущего объекта "Text: " значение параметра	∅

3.19 Алгоритм метода handler класса cl_3

Функционал: обработчик сигнала.

Параметры: строка.

Возвращаемое значение: отсутствует.

Алгоритм метода представлен в таблице 19.

Таблица 19 – Алгоритм метода handler класса cl_3

№	Предикат	Действия	№ перехода
1		вывод на экран "Signal to " координата текущего объекта "Text: " значение параметра	∅

3.20 Алгоритм метода handler класса cl_4

Функционал: обработчик сигнала.

Параметры: строка.

Возвращаемое значение: отсутствует.

Алгоритм метода представлен в таблице 20.

Таблица 20 – Алгоритм метода handler класса cl_4

№	Предикат	Действия	№ перехода
1		вывод на экран "Signal to " координата текущего объекта "Text: " значение параметра	∅

3.21 Алгоритм метода handler класса cl_5

Функционал: обработчик сигнала.

Параметры: строка.

Возвращаемое значение: отсутствует.

Алгоритм метода представлен в таблице 21.

Таблица 21 – Алгоритм метода handler класса cl_5

№	Предикат	Действия	№ перехода
1		вывод на экран "Signal to " координата текущего объекта "Text: " значение параметра	∅

3.22 Алгоритм метода handler класса cl_6

Функционал: обработчик сигнала.

Параметры: строка.

Возвращаемое значение: отсутствует.

Алгоритм метода представлен в таблице 22.

Таблица 22 – Алгоритм метода handler класса cl_6

№	Предикат	Действия	№ перехода
1		вывод на экран "Signal to " координата текущего объекта "Text: " значение параметра	∅

4 БЛОК-СХЕМЫ АЛГОРИТМОВ

Представим описание алгоритмов в графическом виде на рисунках 1-6.

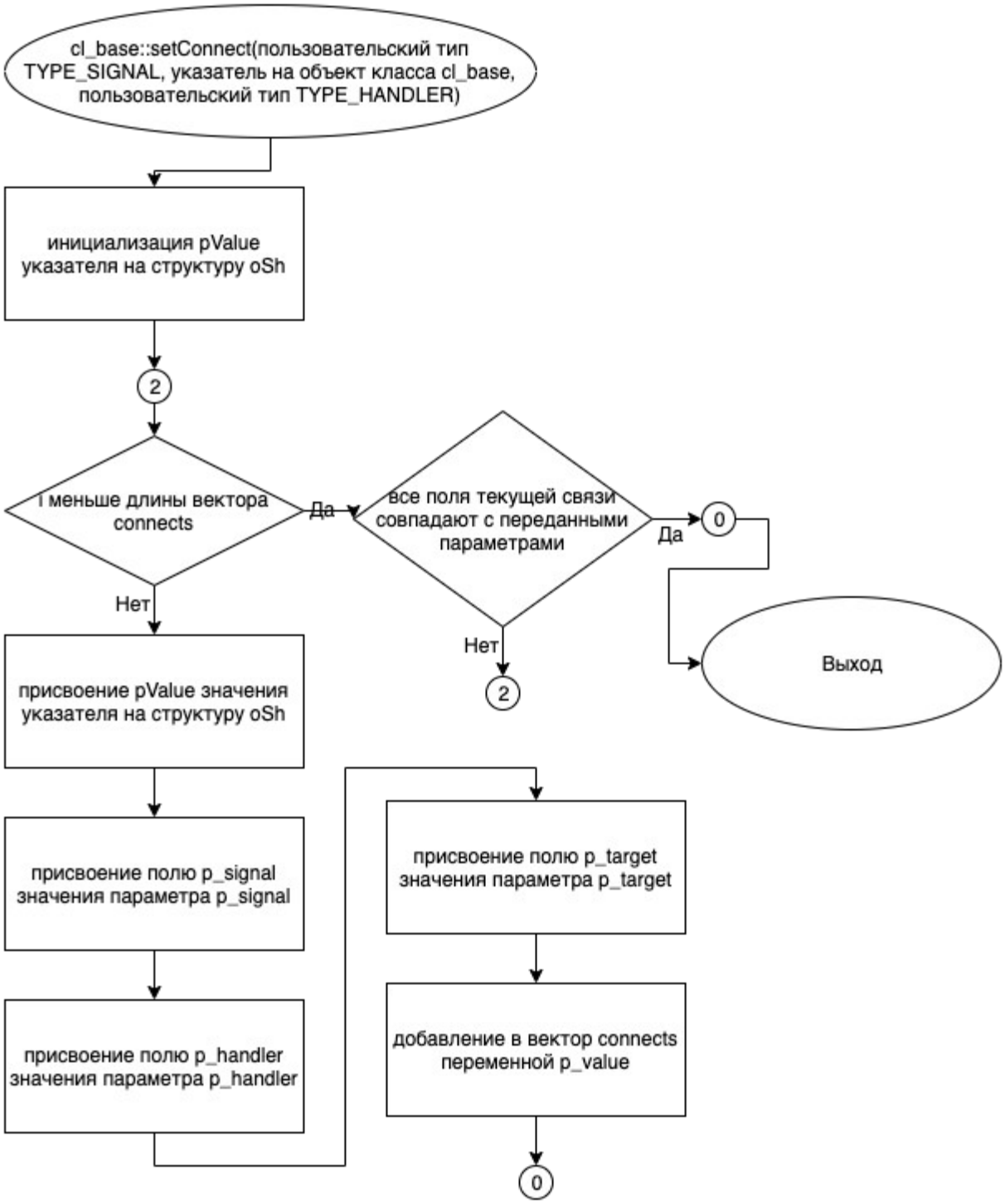

Рисунок 1 – Блок-схема алгоритма

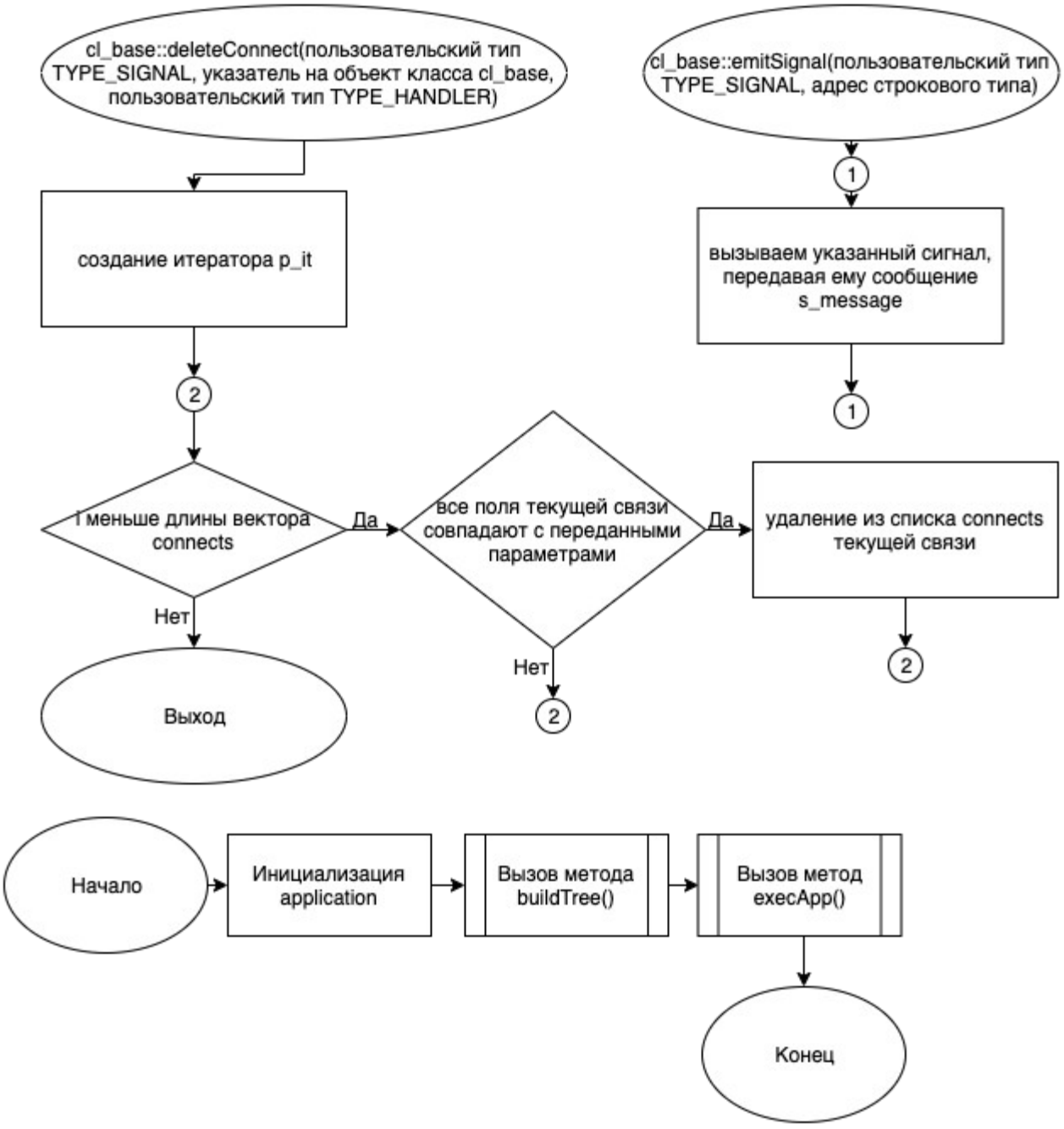

Рисунок 2 – Блок-схема алгоритма

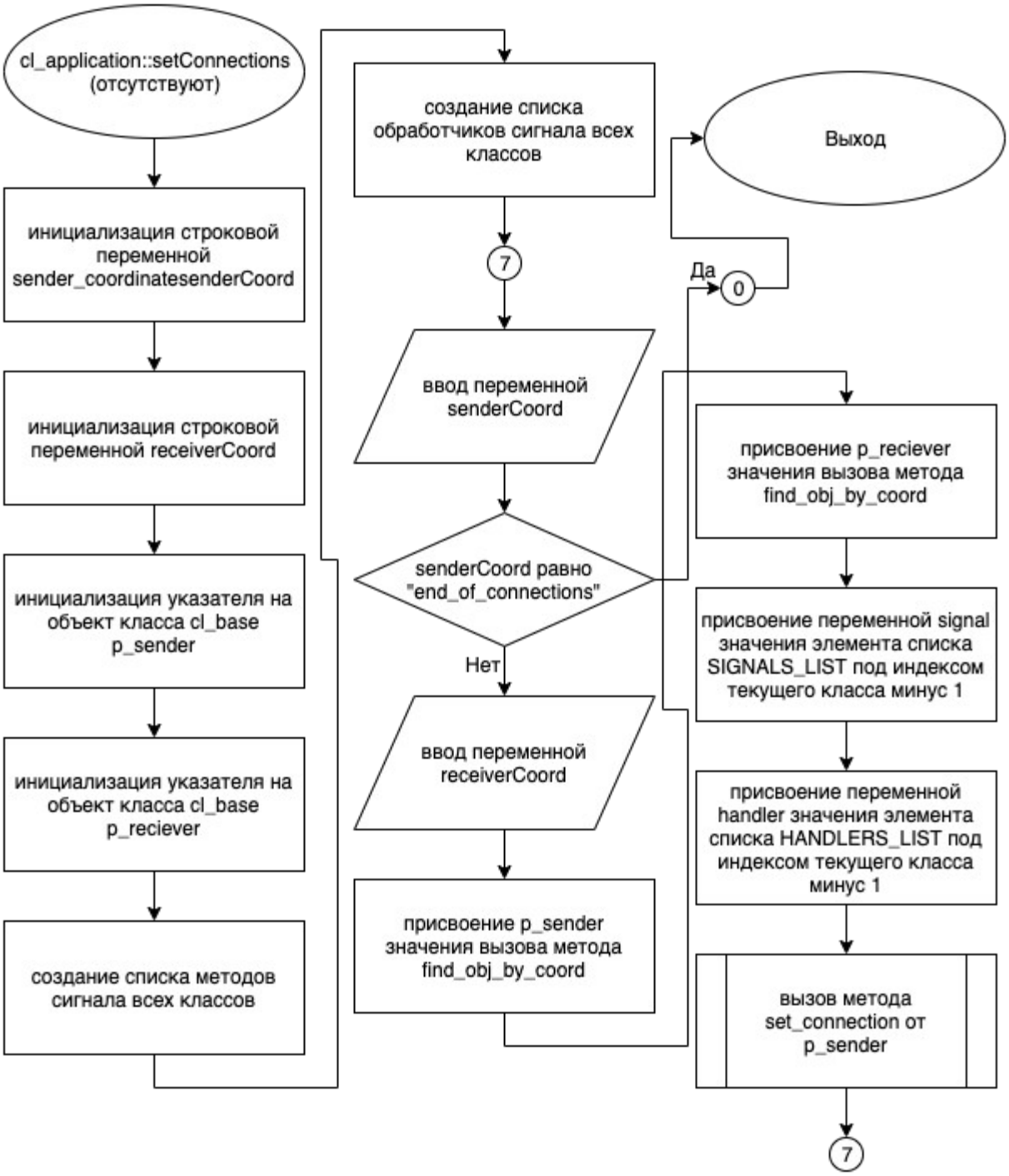

Рисунок 3 – Блок-схема алгоритма

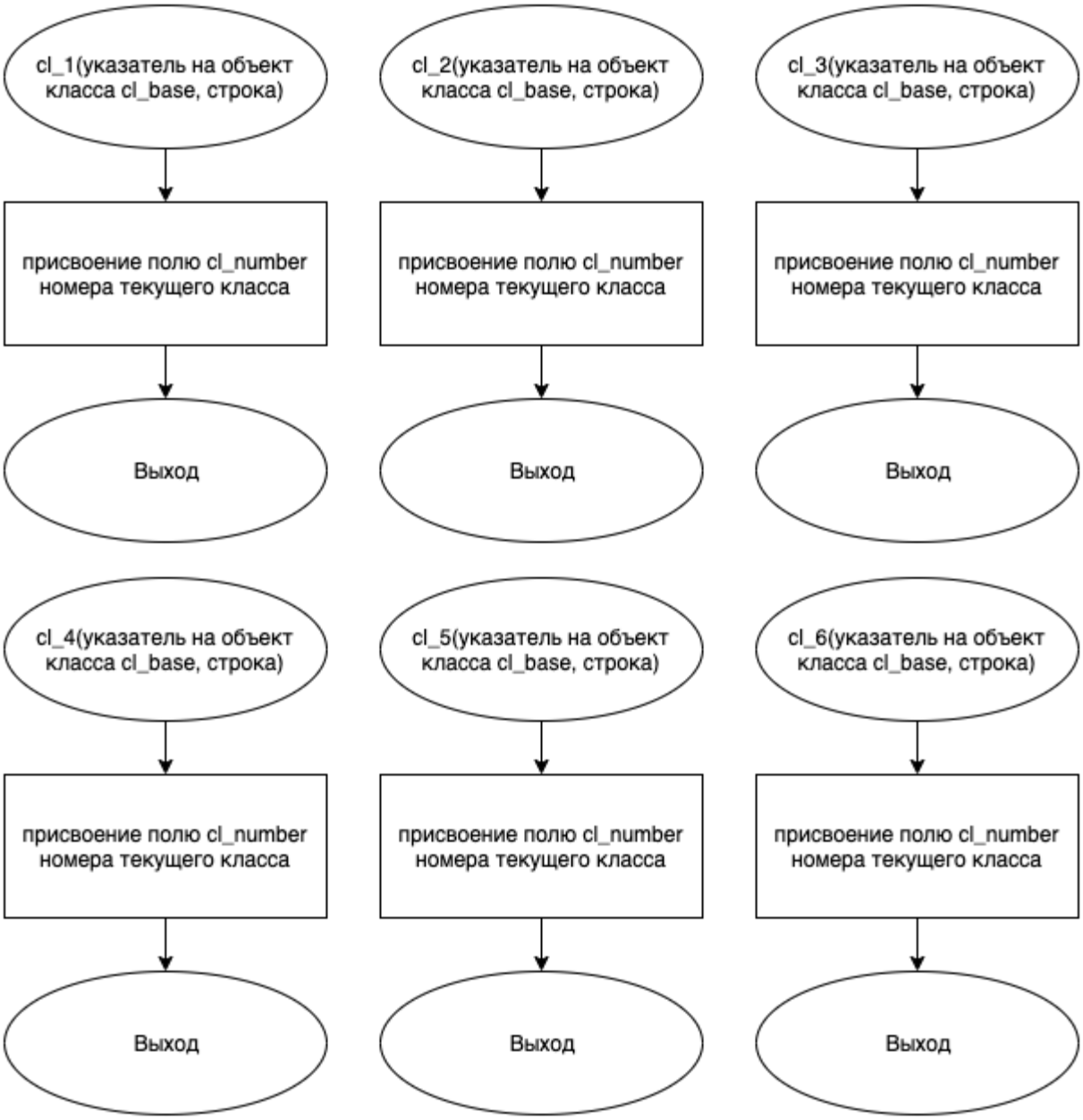

Рисунок 4 – Блок-схема алгоритма

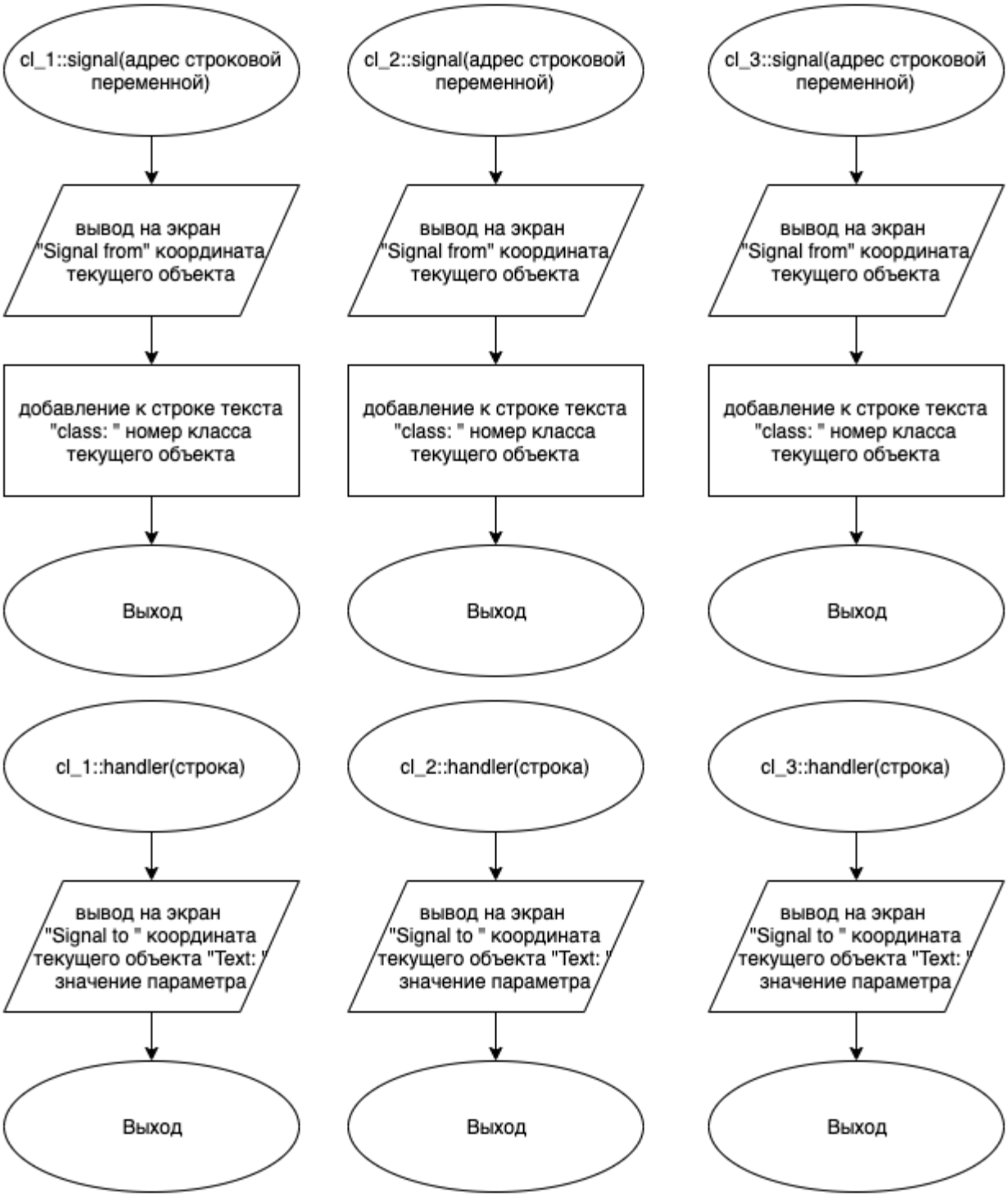

Рисунок 5 – Блок-схема алгоритма

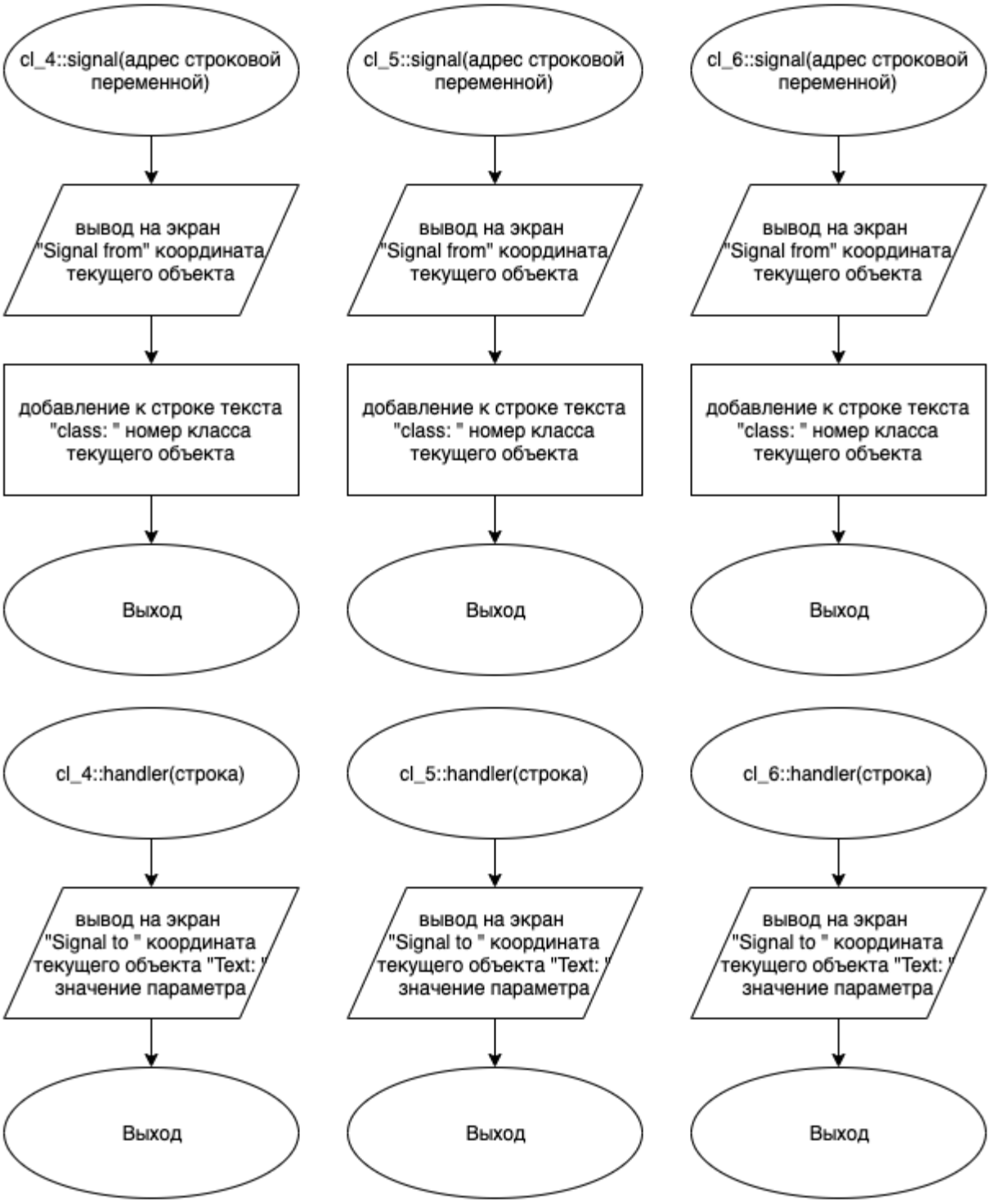

Рисунок 6 – Блок-схема алгоритма

5 КОД ПРОГРАММЫ

Программная реализация алгоритмов для решения задачи представлена ниже.

5.1 Файл cl_1.cpp

Листинг 1 – cl_1.cpp

```
#include "cl_1.h"
cl_1::cl_1(cl_base*  p_head_object,  string  s_name)  :cl_base(p_head_object,
s_name)
{
   this->number = 1;
}
void cl_1::signal_f(string& msg)
{
   cout << endl << "Signal from " << this->get_path();
   msg += " (class: 1)";
}

void cl_1::handler_f(string msg)
{
   cout << endl << "Signal to " << get_path() << " Text:  " << msg;
}
```

5.2 Файл cl_1.h

Листинг 2 – cl_1.h

```
#ifndef __CL_1_H__
#define __CL_1_H__
#include "cl_base.h"

class cl_1 : public cl_base
{
public:
   cl_1(cl_base* p_head_object, string s_name);
   void signal_f(string& msg);
   void handler_f(string msg);
};
#endif
```

5.3 Файл cl_2.cpp

Листинг 3 – cl_2.cpp

```
#include "cl_2.h"
cl_2::cl_2(cl_base*  p_head_object,  string  s_name)  :cl_base(p_head_object,
s_name)
{
   this->number = 2;
}
void cl_2::signal_f(string& msg)
{
   cout << endl << "Signal from " << this->get_path();
   msg += " (class: 2)";
}

void cl_2::handler_f(string msg)
{
   cout << endl << "Signal to " << get_path() << " Text:  " << msg;
}
```

5.4 Файл cl_2.h

Листинг 4 – cl_2.h

```
#ifndef __CL_2_H__
#define __CL_2_H__
#include "cl_base.h"

class cl_2 : public cl_base
{
public:
   cl_2(cl_base* p_head_object, string s_name);
   void signal_f(string& msg);
   void handler_f(string msg);
};
#endif
```

5.5 Файл cl_3.cpp

Листинг 5 – cl_3.cpp

```
#include "cl_3.h"
cl_3::cl_3(cl_base*  p_head_object,  string  s_name)  :cl_base(p_head_object,
```

```
s_name)
{
   this->number = 3;
}
void cl_3::signal_f(string& msg)
{
   cout << endl << "Signal from " << this->get_path();
   msg += " (class: 3)";
}

void cl_3::handler_f(string msg)
{
   cout << endl << "Signal to " << get_path() << " Text:  " << msg;
}
```

5.6 Файл cl_3.h

Листинг 6 – cl_3.h

```
#ifndef __CL_3_H__
#define __CL_3_H__
#include "cl_base.h"

class cl_3 : public cl_base
{
public:
   cl_3(cl_base* p_head_object, string s_name);
   void signal_f(string& msg);
   void handler_f(string msg);
};
#endif
```

5.7 Файл cl_4.cpp

Листинг 7 – cl_4.cpp

```
#include "cl_4.h"
cl_4::cl_4(cl_base*  p_head_object,  string  s_name)  :cl_base(p_head_object,
s_name)
{
   this->number = 4;
}
void cl_4::signal_f(string& msg)
{
   cout << endl << "Signal from " << this->get_path();
```

```
   msg += " (class: 4)";
}

void cl_4::handler_f(string msg)
{
   cout << endl << "Signal to " << get_path() << " Text:  " << msg;
}
```

5.8 Файл cl_4.h

Листинг 8 – cl_4.h

```
#ifndef __CL_4_H__
#define __CL_4_H__
#include "cl_base.h"

class cl_4 : public cl_base
{
public:
   cl_4(cl_base* p_head_object, string s_name);
   void signal_f(string& msg);
   void handler_f(string msg);
};
#endif
```

5.9 Файл cl_5.cpp

Листинг 9 – cl_5.cpp

```
#include "cl_5.h"
cl_5::cl_5(cl_base*  p_head_object,  string  s_name)  :cl_base(p_head_object,
s_name)
{
   this->number = 5;
}
void cl_5::signal_f(string& msg)
{
   cout << endl << "Signal from " << this->get_path();
   msg += " (class: 5)";
}

void cl_5::handler_f(string msg)
{
   cout << endl << "Signal to " << get_path() << " Text:  " << msg;
}
```

5.10 Файл cl_5.h

Листинг 10 – cl_5.h

```
#ifndef __CL_5_H__
#define __CL_5_H__
#include "cl_base.h"

class cl_5 : public cl_base
{
public:
   cl_5(cl_base* p_head_object, string s_name);
   void signal_f(string& msg);
   void handler_f(string msg);
};
#endif
```

5.11 Файл cl_6.cpp

Листинг 11 – cl_6.cpp

```
#include "cl_6.h"
cl_6::cl_6(cl_base*  p_head_object,  string  s_name)  :cl_base(p_head_object,
s_name)
{
   this->number = 6;
}
void cl_6::signal_f(string& msg)
{
   cout << endl << "Signal from " << this->get_path();
   msg += " (class: 6)";
}

void cl_6::handler_f(string msg)
{
   cout << endl << "Signal to " << get_path() << " Text:  " << msg;
}
```

5.12 Файл cl_6.h

Листинг 12 – cl_6.h

```
#ifndef __CL_6_H__
#define __CL_6_H__
#include "cl_base.h"

class cl_6 : public cl_base
{
public:
   cl_6(cl_base* p_head_object, string s_name);
   void signal_f(string& msg);
   void handler_f(string msg);
};
#endif
```

5.13 Файл cl_application.cpp

Листинг 13 – cl_application.cpp

```
#include "cl_application.h"

void setConnections();

// Конструктор класса cl_application, который принимает указатель на
головной объект и передает его в конструктор базового класса
cl_application::cl_application(cl_base*
p_head_object) :cl_base(p_head_object) {}

void cl_application::build_tree_objects()
{
   string s_head, s_sub;
   int s_num_obj;
   // Объявление переменных для хранения названия объектов, количества
подобъектов и готовности к работе
   cl_base* p_head = this, * p_sub = nullptr;
   // Объявление указателей на текущий объект и создаваемый подобъект.
Изначально текущий - это само приложение.
   cin >> s_head;
   set_name(s_head);
   // Считывание названия корневого элемента дерева (головного объект),
установка этого имени как имя приложения
   while (true)
   {
      cin >> s_head;
      if (s_head == "endtree")
      {
```

```
            break;
            // Если считанное слово - endtree (конец описания деревьев), то
выходим из цикла
         }
         cin >> s_sub >> s_num_obj;
         if (p_head != nullptr)
         {
            p_head = find_obj_by_coord(s_head);
            switch (s_num_obj)
            {
            case 1:
               p_sub = new cl_1(p_head, s_sub);
               break;
            case 2:
               p_sub = new cl_2(p_head, s_sub);
               break;
            case 3:
               p_sub = new cl_3(p_head, s_sub);
               break;
            case 4:
               p_sub = new cl_4(p_head, s_sub);
               break;
            case 5:
               p_sub = new cl_5(p_head, s_sub);
               break;
            case 6:
               p_sub = new cl_6(p_head, s_sub);
               break;
            // Создаем подобъект в зависимости от значения переменной количества
потомков
            // Конструкторы классов принимают указатель на родительский объект и
имя создаваемого объекта
            //  Присваиваем указателю на созданный подобъект значение этого
нового объекта
            }
         }
         else
         {
            cout << "Object tree";
            print_from_current();
            cout << endl << "The head object " << s_head << " is not found";
            exit(1);
         }
      }
   }

   void cl_application::build_commands()
   {
      //  Объявляются переменные line, command, coord, text, которые будут
   использоваться для обработки командной строки и ее разделения на отдельные
   части.
      string line, command, coord, text;
      //  Создаются векторы SIGNALS_LIST и HANDLERS_LIST, содержащие список
   сигналов и обработчиков соответственно.
      vector<TYPE_SIGNAL> SIGNALS_LIST =
```

```
   {
      SIGNAL_D(cl_1::signal_f),
      SIGNAL_D(cl_2::signal_f),
      SIGNAL_D(cl_3::signal_f),
      SIGNAL_D(cl_4::signal_f),
      SIGNAL_D(cl_5::signal_f),
      SIGNAL_D(cl_6::signal_f)
   };
   vector<TYPE_HANDLER> HANDLERS_LIST =
   {
      HANDLER_D(cl_1::handler_f),
      HANDLER_D(cl_2::handler_f),
      HANDLER_D(cl_3::handler_f),
      HANDLER_D(cl_4::handler_f),
      HANDLER_D(cl_5::handler_f),
      HANDLER_D(cl_6::handler_f)
   };
   // Бесконечный цикл while (true), который будет обрабатывать команды до
тех пор, пока не будет введена команда "END".
   while (true)
   {
      getline(cin, line); // Считывание строки из входного потока
      command = line.substr(0, line.find(' ')); // Извлечение команды из
строки
      line = line.substr(line.find(' ') + 1, line.size() - 1); // Обновление
строки, исключая команду

      coord = line.substr(0, line.find(' ')); // Извлечение координаты
объекта из строки
      text = line.substr(line.find(' ') + 1); // Извлечение текста из строки

      // END – завершает функционирование системы (выполнение программы).
      if (command == "END")
      {
         break; // Выход из цикла
      }

      if (line == "")
      {
         continue; // Пропуск пустой строки
      }

      cl_base* pSender = this->find_obj_by_coord(coord); // Поиск объекта по
координате
      if (pSender == nullptr)
      {
         cout << endl << "Object " << coord << " not found"; // Вывод
сообщения об отсутствии объекта
         continue; // Продолжение цикла
      }

      // EMIT «координата объекта» «текст» – выдает сигнал от заданного по
координате объекта;
      if (command == "EMIT")
      {
         TYPE_SIGNAL signal = SIGNALS_LIST[pSender->number - 1]; // Получение
```

```
сигнала по номеру объекта
        pSender->emitSignal(signal, text); // Вызов метода emitSignal для
объекта pSender
     }

     // SET_CONNECT «координата объекта выдающего сигнал» «координата
целевого объекта» – устанавливает связь;
     if (command == "SET_CONNECT")
     {
        cl_base* pReceiver = this->find_obj_by_coord(text); // Поиск
целевого объекта по координате
        if (pReceiver == nullptr)
        {
           cout << endl << "Handler object " << text << " not found"; //
Вывод сообщения об отсутствии целевого объекта
        }
        TYPE_SIGNAL signal = SIGNALS_LIST[pSender->number - 1]; // Получение
сигнала по номеру объекта-отправителя
        TYPE_HANDLER handler = HANDLERS_LIST[pReceiver->number - 1]; //
Получение обработчика по номеру целевого объекта
        pSender->setConnect(signal, pReceiver, handler); // Вызов метода
setConnect для объекта pSender
     }

     // DELETE_CONNECT «координата объекта выдающего сигнал» «координата
целевого объекта» – удаляет связь;
     if (command == "DELETE_CONNECT")
     {
        cl_base* pReceiver = this->find_obj_by_coord(text); // Поиск
целевого объекта по координате
        if (pReceiver == nullptr)
        {
           cout << endl << "Handler object " << text << " not found"; //
Вывод сообщения об отсутствии целевого объекта
        }
        else
        {
           TYPE_SIGNAL signal = SIGNALS_LIST[pSender->number - 1]; //
Получение сигнала по номеру объекта-отправителя
           TYPE_HANDLER handler = HANDLERS_LIST[pReceiver->number - 1]; //
Получение обработчика по номеру целевого объекта
           pSender->deleteConnect(signal, pReceiver, handler); // Вызов
метода deleteConnect для объекта pSender
        }
     }

     // SET_CONDITION «координата объекта» «значение состояния» –
устанавливает состояние объекта.
     if (command == "SET_CONDITION")
     {
        int state = stoi(text); // Преобразование текста в целочисленное
значение состояния
        pSender->setState(state); // Вызов метода setState для объекта
pSender
     }
```

```
   }
}


int cl_application::exec_app()
{
   cout << "Object tree";
   print_from_current();
   this->setConnections();
   build_commands();
   return 0;
}

// Метод setConnections является частью класса cl_application и отвечает за
установку связей между объектами системы на основе ввода с консоли.
void cl_application::setConnections()
{
   string senderCoord; // Переменная для хранения координаты отправителя
   string receiverCoord; // Переменная для хранения координаты получателя
   cl_base* pSender; // Указатель на объект-отправитель
   cl_base* pReceiver; // Указатель на объект-получатель
   vector<TYPE_SIGNAL> SIGNALS_LIST = // Вектор сигналов
   {
      SIGNAL_D(cl_1::signal_f),
      SIGNAL_D(cl_2::signal_f),
      SIGNAL_D(cl_3::signal_f),
      SIGNAL_D(cl_4::signal_f),
      SIGNAL_D(cl_5::signal_f),
      SIGNAL_D(cl_6::signal_f)
   };
   vector<TYPE_HANDLER> HANDLERS_LIST = // Вектор обработчиков
   {
      HANDLER_D(cl_1::handler_f),
      HANDLER_D(cl_2::handler_f),
      HANDLER_D(cl_3::handler_f),
      HANDLER_D(cl_4::handler_f),
      HANDLER_D(cl_5::handler_f),
      HANDLER_D(cl_6::handler_f)
   };
   while (true)
   {
      cin >> senderCoord; // Ввод координаты отправителя
      if (senderCoord == "end_of_connections") break; // Если введена команда
"end_of_connections", то выход из цикла
      cin >> receiverCoord; // Ввод координаты получателя
      pSender  =  this->find_obj_by_coord(senderCoord);  //  Поиск  объекта-
отправителя по координате
      pReceiver = this->find_obj_by_coord(receiverCoord); // Поиск объекта-
получателя по координате
      TYPE_SIGNAL signal = SIGNALS_LIST[pSender->number - 1]; // Получение
сигнала по номеру объекта-отправителя
      TYPE_HANDLER  handler  =  HANDLERS_LIST[pReceiver->number  -  1];  //
Получение обработчика по номеру объекта-получателя
      pSender->setConnect(signal,  pReceiver,  handler);  //  Вызов  метода
setConnect для объекта-отправителя
```

```
    }
}
```

5.14 Файл cl_application.h

Листинг 14 – cl_application.h

```
#ifndef __CL_APPLICATION_H__
#define __CL_APPLICATION_H__
#include "cl_base.h"
#include "cl_1.h"
#include "cl_2.h"
#include "cl_3.h"
#include "cl_4.h"
#include "cl_5.h"
#include "cl_6.h"
class cl_application : public cl_base
{
public:
   cl_application(cl_base* p_head_object);
   void build_tree_objects();
   int exec_app();
   void build_commands();
   void setConnections();
   void handleCommands();
};
#endif
```

5.15 Файл cl_base.cpp

Листинг 15 – cl_base.cpp

```
#include "cl_base.h" // Подключение заголовочного файла cl_base.h

// Конструктор класса cl_base с параметрами p_head_object и s_name
cl_base::cl_base(cl_base* p_head_object, string s_name)
{
   this->s_name = s_name; // Установка имени объекта
   this->p_head_object = p_head_object; // Установка родительского объекта
   // Если родительский объект не равен nullptr (т.е. он существует)
   if (p_head_object != nullptr)
   {
      p_head_object->p_sub_objects.push_back(this);  //  Добавление  текущего
объекта в список подчиненных объектов родителя
   }
```

```
}

// Деструктор класса cl_base
cl_base::~cl_base()
{
   if(get_head() != nullptr)
   {
      get_head()->delete_subordinate_obj(get_name());
   }
   get_root()->delete_links(this);
   for (int i = 0; i < p_sub_objects.size(); i++) // Цикл по всем подчиненным
объектам
   {
      delete p_sub_objects[i]; // Удаление подчиненного объекта
   }
}

// Функция установки нового имени объекта
bool cl_base::set_name(string s_new_name)
{
   // Если родительский объект существует
   if (get_head() != nullptr)
   {
      // Проверка на совпадение имени с уже существующими подчиненными
объектами
      for (int i = 0; i < get_head()->p_sub_objects.size(); i++)
      {
         if (get_head()->p_sub_objects[i]->get_name() == s_new_name)
         {
            return false;
         }
      }
   }
   s_name = s_new_name; // Установка нового имени
   return true; // Возвращаем true, если имя успешно изменено
}

// Функция вывода дерева объектов
void cl_base::print_tree(string delay)
{
   cout << endl << delay << get_name(); // Вывод текущего объекта с заданным
отступом
   for (auto p_sub : p_sub_objects) // Цикл по всем подчиненным объектам
   {
      p_sub->print_tree(delay + "    "); // Рекурсивный вызов функции вывода
дерева объектов для подчиненных объектов с увеличенным отступом
   }
}

// Функция вывода состояния готовности объектов
void cl_base::print_ready(string delay)
{
   cout << endl << delay; // Вывод отступа
   get_ready(get_name()); // Вывод состояния готовности текущего объекта
   for (auto p_sub : p_sub_objects) // Цикл по всем подчиненным объектам
```

```
   {
      p_sub->print_ready(delay + "    "); // Рекурсивный вызов функции вывода
состояния готовности для подчиненных объектов с увеличенным отступом
   }
}

// Функция получения имени объекта
string cl_base::get_name()
{
   return s_name;
}

// Функция получения родительского объекта
cl_base* cl_base::get_head()
{
   return p_head_object;
}

// Функция поиска подчиненного объекта по имени
cl_base* cl_base::get_sub_obj(string s_name)
{
   for (int i = 0; i < p_sub_objects.size(); i++) // Цикл по всем подчиненным
объектам
   {
      if  (p_sub_objects[i]->s_name  ==  s_name)  //  Если  имя  подчиненного
объекта совпадает с искомым
      {
         return p_sub_objects[i]; // Возвращаем найденный объект
      }
   }
   return nullptr; // Возвращаем nullptr, если объект не найден
}

// Функция подсчета количества объектов с заданным именем
int cl_base::count(string name)
{
   int count = 0; // Обнуляем счетчик
   if (get_name() == name) // Если имя текущего объекта совпадает с искомым
   {
      count++; // Увеличиваем счетчик на 1
   }
   for (int i = 0; i < p_sub_objects.size(); i++) // Цикл по всем подчиненным
объектам
   {
      count += p_sub_objects[i]->count(name); // Рекурсивно вызываем функцию
подсчета для подчиненных объектов и добавляем результат к текущему счетчику
   }
   return  count;  //  Возвращаем  итоговое  количество  найденных  объектов  с
заданным именем
}

// Функция search_by_name ищет объект с заданным именем в иерархии
cl_base* cl_base::search_by_name(string name)
{
   // Если имя текущего объекта совпадает с заданным именем
```

```
    if (s_name == name)
    {
        return this; // Возвращаем указатель на текущий объект
    }
    cl_base* p_result = nullptr;
    // Перебираем все дочерние объекты текущего объекта
    for (int i = 0; i < p_sub_objects.size(); i++)
    {
        // Ищем объект с заданным именем в поддереве i-го дочернего объекта
        p_result = p_sub_objects[i]->search_by_name(name);
        // Если нашли объект с заданным именем
        if (p_result != nullptr)
        {
            return p_result; // Возвращаем найденный объект
        }
    }
    // Если объект с заданным именем не найден, возвращаем nullptr
    return nullptr;
}

// Функция search_cur ищет объект с заданным именем в текущем объекте
cl_base* cl_base::search_cur(string name)
{
    // Если заданное имя встречается больше одного раза, возвращаем nullptr
    if (count(name) != 1)
    {
        return nullptr;
    }
    // Иначе ищем объект с заданным именем и возвращаем его
    return search_by_name(name);
}

// Функция search_from_root ищет объект с заданным именем, начиная с
корневого объекта
cl_base* cl_base::search_from_root(string name)
{
    // Если есть ссылка на корневой объект
    if (p_head_object != nullptr)
    {
        // Ищем объект с заданным именем, начиная с корневого объекта
        return p_head_object->search_from_root(name);
    }
    else // Иначе ищем объект с заданным именем в текущем объекте
    {
        return search_cur(name);
    }
}

// Функция set_ready устанавливает состояние готовности объекта и его
дочерних объектов
void cl_base::set_ready(int s_new_ready)
{
    // Если новое значение готовности не равно нулю
    if (s_new_ready != 0)
    {
        // Если нет ссылки на корневой объект или ссылка есть, и его состояние
```

```
готовности не равно нулю
      if (p_head_object == nullptr || p_head_object != nullptr &&
p_head_object->p_ready != 0)
      {
         p_ready = s_new_ready; // Устанавливаем состояние готовности
текущего объекта
      }
   }
   else // Если новое значение готовности равно нулю
   {
      p_ready = s_new_ready; // Устанавливаем состояние готовности текущего
объекта
      // Устанавливаем состояние готовности для всех дочерних объектов
      for (int i = 0; i < p_sub_objects.size(); i++)
      {
         p_sub_objects[i]->set_ready(s_new_ready);
      }
   }
}

// Функция get_ready выводит информацию о состоянии готовности объекта с
заданным именем
void cl_base::get_ready(string name)
{
   // Если имя текущего объекта совпадает с заданным именем
   if (get_name() == name)
   {
      // Если состояние готовности текущего объекта не равно нулю
      if (p_ready != 0)
      {
         cout << get_name() << " is ready"; // Выводим сообщение, что объект
готов
      }
      else // Если состояние готовности текущего объекта равно нулю
      {
         cout << get_name() << " is not ready"; // Выводим сообщение, что
объект не готов
      }
   }
   else // Если имя текущего объекта не совпадает с заданным именем
   {
      // Перебираем все дочерние объекты и вызываем функцию get_ready для
каждого из них
      for (int i = 0; i < p_sub_objects.size(); i++)
      {
         return p_sub_objects[i]->get_ready(name);
      }
   }
}


// Функция предназначена для изменения родительского объекта текущего
объекта
bool cl_base::change_head_obj(cl_base* new_head_obj)
{
```

```
   if (new_head_obj != nullptr)
   {
      // Создание временного указателя temp = new_head_obj
      cl_base* temp = new_head_obj;
      // Цикл идёт пока temp не станет нулевым указателем
      while (temp != nullptr)
      {
         temp = temp->p_head_object;
         if (temp == this)
         {
            return false;
         }
      }
      // Проверяется, что новый главный объект не имеет дочернего объекта с
именем текущего объекта и что у текущего объекта есть родительский объект
      if (new_head_obj->get_sub_obj(get_name()) == nullptr && p_head_object !
= nullptr)
      {
         //   Текущий   объект   удаляется   из   списка   дочерних   объектов
родительского объекта
         p_head_object->p_sub_objects.erase(find(p_head_object-
>p_sub_objects.begin(), p_head_object->p_sub_objects.end(), this));
         //  Текущий объект добавляется в список дочерних объектов нового
главного объекта
         new_head_obj->p_sub_objects.push_back(this);
         //   Родительский   объект   текущего   объекта   устанавливается   на
new_head_obj
         p_head_object = new_head_obj;
         return true;
      }
   }
   return false;
}

// Метод, который удаляет подобъект, хранящийся в списке p_sub_objects
void cl_base::delete_subordinate_obj(string name)
{
   for (auto p_sub = p_sub_objects.begin(); p_sub != p_sub_objects.end();
p_sub++)
   {
      if ((*p_sub)->get_name() == name)
      {
         p_sub_objects.erase(p_sub);
         break;
      }
   }
}

cl_base* cl_base::get_root()
{
   if (p_head_object != nullptr)
   {
      p_head_object->get_root();
   }
   return this;
```

```
}

cl_base* cl_base::find_obj_by_coord(string s_object_path)
{
   //------------------------------
   //метод осуществляет поиск объекта по координате
   //------------------------------

   // Если пустая строка
   if (s_object_path == "")
      return nullptr;

   cl_base* head_obj = this;
   string s_path_item;

   if (s_object_path == "/")
      return this->get_root();

   if (s_object_path == ".")
      return head_obj;

   if (s_object_path[0] == '/' && s_object_path[1] == '/')
   {
      // Удаление символов и вызов функции search_from_root для поиска
объекта, начиная с корневого объекта
      s_object_path.erase(s_object_path.begin());
      s_object_path.erase(s_object_path.begin());
      return this->search_from_root(s_object_path);
   }

   if (s_object_path[0] == '.')
   {
      // Удаление символа и вызов функции search_by_name для поиска объекта
по имени
      s_object_path.erase(s_object_path.begin());
      return search_by_name(s_object_path);
   }

   if (s_object_path[0] == '/')
   {
      // Удаление символа и перемещение head_obj по указателям родительских
объектов, пока не будет достигнут корневой объект
      s_object_path.erase(s_object_path.begin());
      while (head_obj->p_head_object != nullptr)
      {
         head_obj = head_obj->p_head_object;
      }
   }
   //   Создание   объекта   stringstream   с   именем   ss_path,   который
инициализируется значением s_object_path
   stringstream ss_path(s_object_path);
   // Считываются элементы из ss_path разделенные символом /
   while (getline(ss_path, s_path_item, '/'))
   {
      // Вызывается метод get_sub_obj у текущего объектаhead_obj с аргументом
```

```
s_path_item, чтобы получить дочерний объект с заданным именем
      head_obj = head_obj->get_sub_obj(s_path_item);
      if (head_obj == nullptr)
      {
         return nullptr;
      }
   }
   return head_obj;
}

// Метод print_from_current класса cl_base. Он служит для вывода информации
об объекте и его подчиненных объектах, с созданием отступа для каждого
уровня подчиненности.
void cl_base::print_from_current(int n)
{
   cout << endl;
   for (int i = 0; i < n; i++)
   {
      cout << "    ";
   }
   cout << s_name;
   for (auto p_subordinate_object : p_sub_objects)
   {
      p_subordinate_object->print_from_current(n + 1);
   }
}

// Метод setConnect класса cl_base. Он служит для установки связи между
сигналом (p_signal), целью (p_target) и обработчиком (p_handler).
void    cl_base::setConnect(TYPE_SIGNAL    p_signal,    cl_base*    p_target,
TYPE_HANDLER p_handler)
{
   o_sh* p_value; // Объявление указателя на объект типа o_sh
   for (int i = 0; i < connects.size(); i++) // Цикл, выполняющийся для
каждого элемента вектора connects
   {
      if (connects[i]->p_signal == p_signal && // Проверка условия: сигнал и
обработчик уже присутствуют в векторе connects
         connects[i]->p_handler == p_handler &&
         connects[i]->p_target == p_target)
      {
         return;  //  Если  условие  выполняется,  функция  завершается  и
возвращается результат
      }
   }
   p_value = new o_sh(); // Выделение памяти и создание нового объекта типа
o_sh
   p_value->p_signal  =  p_signal;  //  Присваивание  значений  полям  объекта
p_value
   p_value->p_handler = p_handler;
   p_value->p_target = p_target;
   connects.push_back(p_value); // Добавление указателя на созданный объект в
вектор connects
}

// Метод deleteConnect класса cl_base. Он служит для удаления связи между
```

```
сигналом (p_signal), целью (p_target) и обработчиком (p_handler).
void cl_base::deleteConnect(TYPE_SIGNAL p_signal, cl_base* p_target,
TYPE_HANDLER p_handler)
{
  vector<o_sh*>::iterator p_it; // Объявление итератора для обхода вектора
connects
  for (p_it = connects.begin(); p_it != connects.end(); p_it++) // Цикл,
выполняющийся для каждого элемента вектора connects
  {
    if ((*p_it)->p_signal == p_signal && // Проверка условия: сигнал, цель
и обработчик соответствуют заданным значениям
      (*p_it)->p_target == p_target &&
      (*p_it)->p_handler == p_handler)
    {
      delete* p_it; // Освобождение памяти, занимаемой объектом, на
который указывает итератор
      p_it = connects.erase(p_it); // Удаление элемента из вектора
connects и получение итератора на следующий элемент
      p_it--; // Декрементирование итератора, чтобы следующая итерация
цикла работала с правильным элементом
    }
  }
}

// Метод emitSignal класса cl_base. Он служит для генерации сигнала
(p_signal) с сообщением (s_message) и передачи его обработчикам,
установленным через метод setConnect.
void cl_base::emitSignal(TYPE_SIGNAL p_signal, string s_message)
{
  if (p_ready != 0) // Проверка условия: значение переменной p_ready не
равно нулю
  {
    TYPE_HANDLER pHandler; // Объявление переменной типа TYPE_HANDLER
    cl_base* pObject; // Объявление переменной типа cl_base*
    (this->*p_signal)(s_message); // Вызов метода, соответствующего
переданному сигналу, на текущем объекте
    for (int i = 0; i < connects.size(); i++) // Цикл, выполняющийся для
каждого элемента вектора connects
    {
      if (connects[i]->p_signal == p_signal) // Проверка условия: сигнал
элемента соответствует переданному сигналу
      {
        pHandler = connects[i]->p_handler; // Присваивание значения
обработчика элемента переменной pHandler
        pObject = connects[i]->p_target; // Присваивание значения цели
элемента переменной pObject
        if (pObject->p_ready != 0) // Проверка условия: значение
переменной p_ready целевого объекта не равно нулю
        {
          (pObject->*pHandler)(s_message); // Вызов метода,
соответствующего обработчику, на целевом объекте
        }
      }
    }
  }
```

```
}

// Метод get_path класса cl_base. Он служит для получения пути текущего
объекта в иерархии объектов.
string cl_base::get_path()
{
   cl_base* p_head_object = this->get_head();
   if (p_head_object != nullptr)
   {
      if (p_head_object->get_head() == nullptr)
      {
         return p_head_object->get_path() + s_name;
      } else
      {
         return p_head_object->get_path() + "/" + s_name;
      }
   }
   return "/";
}

// Метод setState класса cl_base. Он служит для установки состояния объекта
и его подчиненных объектов.
void cl_base::setState(int state)
{
   if (state == 0)
   {
      this->p_ready = 0;
      for (int i = 0; i < p_sub_objects.size(); i++) {
         p_sub_objects[i]->setState(0);
      }
      return; }
   if (this->p_head_object == nullptr || this->p_head_object->p_ready != 0) {
      this->p_ready = state;
   }
}

// Метод delete_links класса cl_base. Он служит для удаления связей (линков)
с указанным целевым объектом targ.
void cl_base::delete_links(cl_base* targ)
{
   for (auto p_it = connects.begin(); p_it != connects.end(); p_it++)
   {
      if ((*p_it)->p_target == targ)
      {
         delete (*p_it);
         connects.erase(p_it);
         p_it--;
      }
   }
   for (auto p_sub : p_sub_objects)
   {
      p_sub->delete_links(targ);
   }
}
```

5.16 Файл cl_base.h

Листинг 16 – cl_base.h

```
#ifndef __CL_BASE_H__
#define __CL_BASE_H__
#include <iostream>
#include <string>
#include <vector>
#include <sstream>
#include <algorithm>
#define SIGNAL_D(signal_f) (TYPE_SIGNAL)(&signal_f)
#define HANDLER_D(handler_f) (TYPE_HANDLER)(&handler_f)
using namespace std;
class cl_base;

typedef void (cl_base::* TYPE_SIGNAL) (string& msg);
typedef void (cl_base::* TYPE_HANDLER) (string msg);

struct o_sh
{
   TYPE_SIGNAL p_signal;
   TYPE_HANDLER p_handler;
   cl_base* p_target;
};

class cl_base
{
private:
   string s_name;
   cl_base* p_head_object;
   vector <cl_base*> p_sub_objects;
   int p_ready = 1;
   vector<o_sh*> connects;
public:
   cl_base(cl_base* p_head_object, string s_name = "Base Object");
   bool set_name(string s_new_name);
   string get_name();
   cl_base* get_head();
   void print_tree(string delay = "");
   cl_base* get_sub_obj(string s_name);
   ~cl_base();
   int count(string name);
   cl_base* search_by_name(string name);
   cl_base* search_cur(string name);
   cl_base* search_from_root(string name);
   void set_ready(int s_new_ready);
   void get_ready(string name);
   void print_ready(string delay = "");
   bool change_head_obj(cl_base* new_head_obj);
   void delete_subordinate_obj(string name);
   cl_base* find_obj_by_coord(string s_object_path);
   void print_from_current(int n = 0);
   void  setConnect(TYPE_SIGNAL  p_signal,  cl_base*  p_target,  TYPE_HANDLER
```

```
p_handler);
   void deleteConnect(TYPE_SIGNAL p_signal, cl_base* p_target, TYPE_HANDLER
p_handler);
   void emitSignal(TYPE_SIGNAL p_signal, string message);
   string get_path();
   int number = 1;
   typedef void (cl_base::* TYPE_HANDLER)(string);
   void setState(int state);
   void delete_links(cl_base* targ);
   cl_base* get_root();
};
#endif
```

5.17 Файл main.cpp

Листинг 17 – main.cpp

```
#include "cl_application.h"
int main()
{
   cl_application  ob_cl_application ( nullptr );
   ob_cl_application.build_tree_objects ( );
   return ob_cl_application.exec_app ( );
}
```

6 ТЕСТИРОВАНИЕ

Результат тестирования программы представлен в таблице 23.

Таблица 23 – Результат тестирования программы

Входные данные	Ожидаемые выходные данные	Фактические выходные данные
appls_root / object_s1 3 / object_s2 2 /object_s2 object_s4 4 / object_s13 5 /object_s2 object_s6 6 /object_s1 object_s7 2 endtree /object_s2/object_s4 /object_s2/object_s6 /object_s2 /object_s1/object_s7 / /object_s2/object_s4 /object_s2/object_s4 / end_of_connections EMIT /object_s2/object_s4 Send message 1 EMIT /object_s2/object_s4 Send message 2 EMIT /object_s2/object_s4 Send message 3 EMIT /object_s1 Send message 4 END	Object tree appls_root object_s1 object_s7 object_s2 object_s4 object_s6 object_s13 Signal from /object_s2/object_s4 Signal to /object_s2/object_s6 Text: Send message 1 (class: 4) Signal to / Text: Send message 1 (class: 4) Signal from /object_s2/object_s4 Signal to /object_s2/object_s6 Text: Send message 2 (class: 4) Signal to / Text: Send message 2 (class: 4) Signal from /object_s2/object_s4 Signal to /object_s2/object_s6 Text: Send message 3 (class: 4) Signal to / Text: Send message 3 (class: 4) Signal from /object_s1	Object tree appls_root object_s1 object_s7 object_s2 object_s4 object_s6 object_s13 Signal from /object_s2/object_s4 Signal to /object_s2/object_s6 Text: Send message 1 (class: 4) Signal to / Text: Send message 1 (class: 4) Signal from /object_s2/object_s4 Signal to /object_s2/object_s6 Text: Send message 2 (class: 4) Signal to / Text: Send message 2 (class: 4) Signal from /object_s2/object_s4 Signal to /object_s2/object_s6 Text: Send message 3 (class: 4) Signal to / Text: Send message 3 (class: 4) Signal from /object_s1

ЗАКЛЮЧЕНИЕ

Изучение курса ООП позволило ознакомиться с методологией объектно-ориентированного программирования. Была освоена концепция классов и объектов, их использование для описания систем. Также было получено понимание таких основных компонентов парадигмы ООП, как наследование, полиморфизм и инкапсуляция. Обучение происходило на языке C++. Это дало возможность изучить такие понятия, как указатели, ссылки, дружественные функции, дружественные классы, виртуальные методы, статические методы, абстрактные классы и так далее.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. ГОСТ 19 Единая система программной документации.

2. Методическое пособие студента для выполнения практических заданий, контрольных и курсовых работ по дисциплине «Объектно-ориентированное программирование» [Электронный ресурс] – URL: https://mirea.aco-avrora.ru/student/files/methodichescoe_posobie_dlya_laboratornyh_rabot_3.pdf (дата обращения 05.05.2021).

3. Приложение к методическому пособию студента по выполнению заданий в рамках курса «Объектно-ориентированное программирование» [Электронный ресурс]. URL: https://mirea.aco-avrora.ru/student/files/Prilozheniye_k_methodichke.pdf (дата обращения 05.05.2021).

4. Шилдт Г. C++: базовый курс. 3-е изд. Пер. с англ.. — М.: Вильямс, 2019. — 624 с.

5. Видео лекции по курсу «Объектно-ориентированное программирование» [Электронный ресурс]. АСО «Аврора».

6. Антик М.И. Дискретная математика [Электронный ресурс]: Учебное пособие /Антик М.И., Казанцева Л.В. — М.: МИРЭА — Российский технологический университет, 2018 — 1 электрон. опт. диск (CD-ROM).