ЛАБОРАТОРНАЯ РАБОТА № 3

«Исследование объектно-ориентированного подхода к программированию с использованием JavaScript»

1. Цель работы

Изучение особенностей объектно-ориентированного подхода разработки ПО, получение практических навыков разработки ПО с использованием JavaScript.

2 Краткие теоретические сведения

Объектно-ориентированный подход (ООП) — это парадигма программирования, которая базируется на концепции объектов и их взаимодействия. Основной идеей ООП является моделирование реального мира через объекты, которые обладают свойствами и методами. Объекты представляют собой абстракции реальных сущностей, которые могут быть использованы для описания различных аспектов программы.

ООП способствует упрощению разработки программного обеспечения путем разделения сложных систем на более мелкие и понятные части. Это позволяет разработчикам лучше структурировать код, уменьшить его сложность и повысить его повторное использование. Кроме того, ООП способствует повышению гибкости программы, позволяя легко вносить изменения и расширять функциональность.

Ключевые идеи:

- Программа — это совокупность объектов, способных взаимодействовать друг с другом посредством сообщений;
- Каждый объект является экземпляром определенного класса;
- Классы образуют иерархию наследования.

Фактически, ОО-программа – это работающая модель:

- Как реализовано поведение элементов этой модели – императивно или декларативно – значения не имеет.
  - Хотя большинство популярных ОО-языков имеют императивные корни (C#, Java, C++), есть и функциональные ОО-языки (OCaml, ООдиалекты Haskell);
- Само по себе понятие «метода» не является обязательным для ОО-парадигмы. Метод – частный случай реакции на сообщение.

Основные принципы объектно-ориентированного подхода включают инкапсуляцию, наследование и полиморфизм.

1. Инкапсуляция - скрытие внутреннего состояния и функций объекта и предоставление доступа только через открытый набор функций.

2. Наследование - возможность создания новых абстракций на основе существующих.

3. Полиморфизм - возможность реализации наследуемых свойств или методов отличающимися способами в рамках множества абстракций.

Для обеспечения высокой степени гибкости, расширяемости и удобства поддержки кода важно также знать принципы SOLID.

Принципы SOLID — это набор основных принципов, разработанных для создания гибких, расширяемых и поддерживаемых программ. Каждая буква в аббревиатуре SOLID представляет собой один из этих принципов. Давайте подробно рассмотрим каждый из них:

1. Принцип единственной ответственности (Single Responsibility Principle - SRP). Принцип SRP утверждает, что класс должен иметь только одну причину для изменения. Это означает, что класс должен быть ответственным только за одну конкретную часть функциональности или задачу. Если класс выполняет

сразу несколько различных задач, его сложнее поддерживать и изменять. Разделение функциональности на отдельные классы с одной ответственностью повышает читаемость, уменьшает зависимости и упрощает тестирование.

2. Принцип открытости/закрытости (Open/Closed Principle - OCP). Принцип OCP утверждает, что классы должны быть открыты для расширения, но закрыты для модификации. Это означает, что поведение класса можно расширить путем добавления нового кода, но не изменяя существующий код. Это достигается путем использования абстракций и интерфейсов, что позволяет создавать новые классы, расширяющие базовый функционал без изменения существующего кода.

3. Принцип подстановки Барбары Лисков (Liskov Substitution Principle - LSP). Принцип LSP утверждает, что объекты базового класса могут быть заменены объектами его производного класса без изменения свойств программы. Это означает, что производные классы должны быть совместимы с базовыми классами и сохранять их поведение. При соблюдении этого принципа код становится более гибким и устойчивым к изменениям.

4. Принцип разделения интерфейсов (Interface Segregation Principle - ISP). Принцип ISP утверждает, что интерфейсы должны быть маленькими, специфичными и разделенными на более мелкие части, чтобы клиенты могли использовать только те методы, которые им действительно нужны. Это помогает избежать создания "толстых" интерфейсов, которые содержат избыточные методы. Разделение интерфейсов уменьшает зависимости между компонентами и повышает гибкость системы.

5. Принцип инверсии зависимостей (Dependency Inversion Principle - DIP). Принцип DIP утверждает, что модули высокого уровня не должны зависеть от модулей низкого уровня, а оба типа модулей должны зависеть от абстракций. Это означает, что классы должны зависеть от абстракций (интерфейсов), а не от конкретных реализаций. Использование абстракций позволяет легко заменять компоненты системы без изменения других частей кода.

Пример использования ООП в JavaScript:

```
// Создаем класс "Человек"
class Person {
  constructor(name, age) {
    this.name = name;
    this.age = age;
  }
  // Метод для вывода информации о человеке
  getInfo() {
    console.log(`Имя: ${this.name}, Возраст: ${this.age}`);
  }
}

// Создаем экземпляр класса "Человек"
const person = new Person("Иван", 20);
// Вызываем метод getInfo для экземпляра person
person.getInfo();
```

В примере создан класс Person с конструктором, который принимает имя и возраст человека, а также метод getInfo для вывода информацию о человеке. Затем был создан экземпляр класса Person с именем person и вызван метод getInfo для этого экземпляра.

4 Порядок выполнения

3.1 Изучить основные средства языка JavaScript для разработки программ с использованием объектно-ориентированного подхода.

3.2 Выполнить две задачи из варианта на языке JavaScript согласно своему варианту.

3.3 Разработать тестовые примеры.

3.4 Выполнить отладку программ.

3.5 Сформулировать выводы, проанализировав разницу разработки программ с использованием декларативных и императивных парадигм.

3.6 Оформить отчет по проделанной работе.

3 Варианты заданий

Для каждого варианта требуется создать одноместный и двухмерный массивы. Для однокамерного массива нужно добавить минимум 10 элементов, для матрицы 16 элементов.

Вариант 1

Найти сумму всех элементов в массиве.

Удалить все строки матрицы, в которых есть отрицательные элементы.

Вариант 2

Найти наибольший элемент в массиве.

Проверить, содержит ли матрица магический квадрат (сумма элементов всех строк, столбцов и диагоналей одинакова).

Вариант 3

Найти индекс первого вхождения определенного элемента в массив.

Найти среднее арифметическое элементов в каждой строке матрицы.

Вариант 4

Посчитать количество четных элементов в массиве.

Поменять местами две заданные строки матрицы.

Вариант 5

Отсортировать массив по возрастанию.

Найти сумму всех элементов в двумерном массиве.

Вариант 6

Изменить порядок элементов массива на обратный.

Отсортировать строки матрицы по возрастанию суммы их элементов.

Вариант 7

Удалить все дубликаты из массива.

Посчитать сумму элементов каждого столбца и сохранить результаты в одномерный массив.

Вариант 8

Найти среднее арифметическое всех элементов массива.

Посчитать количество строк матрицы, в которых есть хотя бы один отрицательный элемент.

Вариант 9

Проверить, является ли массив палиндромом.

Найти наименьший элемент в двумерном массиве.

Вариант 10

Найти сумму элементов на нечетных позициях массива.

Найти сумму элементов главной диагонали матрицы.