**Лабораторная работа №16**

**Выбор и обоснование средств реализации проекта для разрабатываемой программной системы.**

**Цель работы:**

1. Изучить критерии выбора языка программирования. Научиться обосновывать выбор среды разработки в соответствии с критериями выбора языка программирования.

2. Изучить основные парадигмы программирования.

3. Изучить правила формирования листинга программы.

Порядок выполнения работы:

**1. Выполните задания 1-3.**

Задание 1. Критерии выбора языка программирования.

1. Изучить критерии выбора языка программирования. (если информация отсутствует в конспекте, то необходимо ее законспектировать).

2. Обосновать выбор языка программирования для решения поставленной задачи в соответствии с индивидуальным заданием.

Задание 2. Изучение парадигм программирования

1. Изучить различные парадигмы программирования. (если информация отсутствует в конспекте, то необходимо ее законспектировать).

2. Провести сравнительную характеристику различных парадигм программирования.

Результаты оформить в виде таблицы следующего вида:

Дайте характеристику следующим парадигмам программирования:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| № | Название парадигмы | Сущность  парадигмы,  основные  идеи,  принципы,  объекты. | Языки  поддержив  ающие  данную  парадигму.  (2-3 языка) | Достоинства | Недостатки | Примеры  программ |
| 1 | Императивное  программирование | Императи́вное программи́рование — это [парадигма программирования](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%B4%D0%B8%D0%B3%D0%BC%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F) (стиль написания [исходного кода](https://ru.wikipedia.org/wiki/%D0%98%D1%81%D1%85%D0%BE%D0%B4%D0%BD%D1%8B%D0%B9_%D0%BA%D0%BE%D0%B4) [компьютерной программы](https://ru.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D1%8C%D1%8E%D1%82%D0%B5%D1%80%D0%BD%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0)), для которой характерно следующее:  в исходном коде программы записываются [инструкции](https://ru.wikipedia.org/wiki/%D0%9E%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%BE%D1%80_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) (команды);  инструкции должны выполняться последовательно;  данные, получаемые при выполнении предыдущих инструкций, могут читаться из памяти последующими инструкциями;  данные, полученные при выполнении инструкции, могут записываться в память.  Императивная программа похожа на приказы ([англ.](https://ru.wikipedia.org/wiki/%D0%90%D0%BD%D0%B3%D0%BB%D0%B8%D0%B9%D1%81%D0%BA%D0%B8%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA) imperative — приказ, [повелительное наклонение](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B2%D0%B5%D0%BB%D0%B8%D1%82%D0%B5%D0%BB%D1%8C%D0%BD%D0%BE%D0%B5_%D0%BD%D0%B0%D0%BA%D0%BB%D0%BE%D0%BD%D0%B5%D0%BD%D0%B8%D0%B5)), выражаемые повелительным наклонением в [естественных языках](https://ru.wikipedia.org/wiki/%D0%95%D1%81%D1%82%D0%B5%D1%81%D1%82%D0%B2%D0%B5%D0%BD%D0%BD%D1%8B%D0%B9_%D1%8F%D0%B7%D1%8B%D0%BA), то есть представляют собой последовательность команд, которые должен выполнить [процессор](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%BE%D1%86%D0%B5%D1%81%D1%81%D0%BE%D1%80).  При императивном подходе к написанию кода (в отличие от [функционального подхода](https://ru.wikipedia.org/wiki/%D0%AF%D0%B7%D1%8B%D0%BA_%D1%84%D1%83%D0%BD%D0%BA%D1%86%D0%B8%D0%BE%D0%BD%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), относящегося к [декларативной парадигме](https://ru.wikipedia.org/wiki/%D0%94%D0%B5%D0%BA%D0%BB%D0%B0%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) широко используется [присваивание](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D1%81%D0%B2%D0%B0%D0%B8%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5). Наличие операторов присваивания увеличивает сложность [модели вычислений](https://ru.wikipedia.org/wiki/%D0%9C%D0%BE%D0%B4%D0%B5%D0%BB%D1%8C_%D0%B2%D1%8B%D1%87%D0%B8%D1%81%D0%BB%D0%B5%D0%BD%D0%B8%D0%B9) и делает императивные программы подверженными специфическим ошибкам, не встречающимся при функциональном подходе[[1]](https://ru.wikipedia.org/wiki/%D0%98%D0%BC%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5#cite_note-1).  Основные черты императивных языков:  использование именованных [переменных](https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5));  использование оператора присваивания;  использование составных выражений;  использование [подпрограмм](https://ru.wikipedia.org/wiki/%D0%9F%D0%BE%D0%B4%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B0);  и др. | С, С++,  Java | Выходные данные зависят только от входных данных (как функции в математике), поэтому их легче понять. Чистые функции принимают аргументы один раз и выдают неизменяемый результат для каждого из них. Поэтому они не дают скрытых результатов. Они используют неизменяемые значения, что упрощает отладку и тестирование. В дополнение к созданию неизменяемых результатов функциональное программирование поддерживает концепцию ленивых вычислений, когда значение оценивается и сохраняется только тогда, когда это необходимо. Поскольку чистые функции тратят только входные значения, не только проще выполнять функции параллельно, но и легче создавать рекурсивные функции. Такие выражения, как map, reduce или filter, просты и лаконичны. | В функциональном программировании функции не имеют состояния, поэтому они всегда создают новые объекты для выполнения действий вместо изменения существующих объектов. По этой причине приложения FP занимают много памяти. | function double (arr) {  let results = [];  for (let i = 0; i < arr.length; i++){  results.push(arr[i] \* 2);  }  return results;  } |
| 2 | Декларативное  программирование | Декларати́вное программи́рование — [парадигма программирования](https://ru.wikipedia.org/wiki/%D0%9F%D0%B0%D1%80%D0%B0%D0%B4%D0%B8%D0%B3%D0%BC%D0%B0_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D1%8F), в которой задаётся [спецификация](https://ru.wikipedia.org/wiki/%D0%A1%D0%BF%D0%B5%D1%86%D0%B8%D1%84%D0%B8%D0%BA%D0%B0%D1%86%D0%B8%D1%8F) решения задачи, то есть описывается ожидаемый результат, а не способ его получения. Противоположностью декларативного является [императивное программирование](https://ru.wikipedia.org/wiki/%D0%98%D0%BC%D0%BF%D0%B5%D1%80%D0%B0%D1%82%D0%B8%D0%B2%D0%BD%D0%BE%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), при котором на том или ином уровне детализации требуется описание последовательности шагов для решения задачи. В качестве примеров декларативных языков обычно приводят [HTML](https://ru.wikipedia.org/wiki/HTML) и [SQL](https://ru.wikipedia.org/wiki/SQL).  Декларативные программы не используют понятия состояния, в частности, не содержат [переменных](https://ru.wikipedia.org/wiki/%D0%9F%D0%B5%D1%80%D0%B5%D0%BC%D0%B5%D0%BD%D0%BD%D0%B0%D1%8F_(%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5)) и операторов [присваивания](https://ru.wikipedia.org/wiki/%D0%9F%D1%80%D0%B8%D1%81%D0%B2%D0%B0%D0%B8%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5), обеспечивается [ссылочная прозрачность](https://ru.wikipedia.org/wiki/%D0%A1%D1%81%D1%8B%D0%BB%D0%BE%D1%87%D0%BD%D0%B0%D1%8F_%D0%BF%D1%80%D0%BE%D0%B7%D1%80%D0%B0%D1%87%D0%BD%D0%BE%D1%81%D1%82%D1%8C). Наиболее близким к «чисто декларативному» программированию является написание исполнимых спецификаций. | HTML, CSS,SQl | – Визуальную привлекательность и удобство использования: Декоративный подход может помочь создать интерфейсы, которые выглядят привлекательно и легко воспринимаются пользователями, что может повысить их удовлетворенность и лояльность к продукту. – Простоту и понятность: Декоративные интерфейсы могут быть более интуитивно понятными, что упрощает использование продукта и сокращает время обучения пользователей. – Быстрое время разработки: Декоративный подход позволяет создавать интерфейсы быстрее, поскольку не требуется разрабатывать сложную функциональность. | – Ограниченную гибкость и возможности: Декоративные интерфейсы не позволяют реализовывать все функции и возможности продукта, что может ограничить его потенциал. – Сложность в поддержке и обновлении: Декоративные интерфейсы требуют более тщательной работы по обновлению и поддержке, поскольку изменения в дизайне могут потребовать значительных изменений в коде. – Неэффективность в сложных задачах: Декоративный подход не всегда является лучшим выбором для решения сложных задач или создания высокопроизводительных приложений.  Таким образом, декоративное программирование может иметь свои преимущества, но также важно учитывать его недостатки и выбирать подходящий подход в зависимости от конкретной задачи и требований проекта. |  |
| 3 | Структурное  Программирование | Структурное программирование - это парадигма программирования, которая основывается на концепции структурирования программы на модули и подпрограммы. Основные идеи структурного программирования:  – Разделение программы на модули, которые выполняют определенную функцию. – Использование подпрограмм для повторного использования кода. – Обработка ошибок и исключений. – Четкое разделение интерфейса и реализации. – Применение принципов объектно-ориентированного программирования.  Объекты - это абстракции, которые представляют собой совокупность данных и методов для работы с ними. Объекты могут быть созданы и уничтожены, а также иметь свои свойства и методы. В структурном программировании объекты используются для структурирования кода и упрощения его понимания. | C, Ada, Pascal | – Более понятный и структурированный код; – Легкость тестирования и отладки; – Возможность повторного использования кода; – Улучшенная модульность и масштабируемость; – Повышение производительности за счет оптимизации кода. | – Необходимость более детального описания алгоритмов; – Увеличение времени разработки из-за необходимости более четкого разделения кода на модули; – Риск появления ошибок при изменении структуры программы; – Ограниченная гибкость при работе с динамическими структурами данных. |  |
| 4 | Процедурное  Программирование | Процедурное программирование - одна из парадигм, которая основывается на последовательном выполнении операций, каждая из которых имеет свой результат.  Основные идеи процедурного программирования:  – Программирование должно быть структурированным и последовательным. – Программа должна быть написана на понятном языке программирования. – Код должен быть легко читаемым и поддерживаемым.  Принципы процедурного программирования включают:  – Использование функций для определения подпрограмм. – Использование циклов для выполнения повторяющихся операций. – Применение условных операторов для принятия решений.  Объекты процедурного программирования могут быть различными, например, числа, строки, массивы, структуры данных и т.д. Они используются для хранения и обработки информации, а также для выполнения определенных операций. | C, Fortran, COBOL | Плюсы процедурного программирования:  – Простота и понятность кода; – Возможность использования готовых функций и библиотек; – Легкость отладки и сопровождения кода; – Универсальность, применимость к различным языкам программирования. | – Ограниченность в использовании объектно-ориентированных принципов; – Сложность в работе с динамическими данными; – Большая зависимость от контекста выполнения программы. | f (b \* b >= 4 \* a \* c)  {  double d = Sqrt(b \* b - 4 \* a \* c);  double x1 = (-b + d) / (2 \* a);  double x2 = (-b - d) / (2 \* a);  }  else //Решения нет! |
| 5 | Модульное  Программирование | Модульность: программа должна быть разбита на модули, которые можно легко модифицировать и заменять без изменения других частей программы.  Инкапсуляция: каждый модуль должен содержать только те данные и функции, которые необходимы для его работы.  Наследование: модули могут наследовать свойства и методы от других модулей, что позволяет создавать иерархию модулей.  Полиморфизм: модули могут предоставлять общий интерфейс для своих функций, что позволяет использовать их в различных контекстах.  Принципы модульного программирования включают в себя:  – модульность: разделение программы на независимые, легко модифицируемые и заменяемые модули; – инкапсуляцию: концентрация данных и функций в отдельных модулях; – наследование: возможность создания иерархии модулей; – полиморфизм: возможность использования общих интерфейсов для функций и методов в разных модулях.  Объекты модульного программирования - это объекты, которые могут быть использованы в программе. Они могут быть созданы с помощью различных методов и функций, а также могут быть изменены или удалены в процессе работы программы. | Python, Java, C# | – Модульность: позволяет легко изменять и заменять отдельные модули без изменения остальной части программы. – Инкапсуляция: скрывает внутренние детали модулей от пользователя, что упрощает их использование и поддержку. – Наследование: позволяет создавать иерархии модулей, что облегчает создание сложных систем. – Полиморфизм: позволяет использовать один интерфейс для нескольких реализаций, что упрощает разработку и сопровождение кода. | – Более сложный код: модули часто содержат более сложный код, чем функции без модульности, что может затруднить понимание и отладку. – Большее время на разработку: модульность требует больше времени на разработку, так как каждый модуль должен быть разработан и протестирован отдельно. – Сложность поддержки: модули могут быть трудно поддерживать из-за сложности внутренней структуры и взаимодействия между ними. | //main.c  #include <stdlib.h>  #include "hello.h"  int main()  {  hello\_world();  return EXIT\_SUCCESS;  } |
| 6 | Объектно  ориентированное  программирование | Объектно-ориентированное программирование (ООП) - это парадигма программирования, основанная на представлении программ в виде объектов, взаимодействующих друг с другом через сообщения. Основные идеи и принципы ООП включают в себя следующие:  Классы и объекты: классы представляют абстрактные типы данных (атомы), а объекты являются экземплярами этих классов (молекулы). Классы определяют структуру объектов, а объекты содержат их состояние и поведение. Абстракция: ООП использует абстракцию для представления сложной структуры данных в виде более простых и понятных элементов. Абстракция позволяет скрывать детали реализации от пользователей, предоставляя им только необходимый интерфейс. Инкапсуляция и наследование: ООП обеспечивает инкапсуляцию данных и методов внутри объектов, а также наследование свойств и методов от родительских классов. Это позволяет создавать более сложные объекты путем комбинирования свойств и методов из разных источников. Полиморфизм и сокрытие информации: ООП поддерживает полиморфизм, то есть возможность использования одного имени метода для разных реализаций. Это позволяет скрыть информацию о реализации методов от пользователей и упростить их использование. Иерархия классов и наследование объектов: ООП позволяет создавать иерархические структуры классов, где объекты могут наследовать методы и свойства от родительских классов и переопределять их по необходимости. Это обеспечивает гибкость и модульность в проектировании систем. Интерфейсы и абстрактные классы: ООП также поддерживает интерфейсы, которые представляют собой абстрактные классы с набором методов, но без реализации. Объекты могут реализовывать несколько интерфейсов, что обеспечивает гибкость в выборе реализации. Наследование и полиморфизм позволяют создавать сложные системы из простых элементов, а инкапсуляция защищает от изменений в реализации, которые могут повлиять на работу других объектов. Это делает ООП мощным инструментом для разработки сложных программных систем. | Java, Python, C++ | Плюсы ООП:  Модульность. ООП позволяет разбить программу на множество мелких модулей, каждый из которых выполняет свою функцию. Это упрощает поддержку и развитие программы, а также делает ее более гибкой.  Наследование. Наследование позволяет создавать новые классы, которые наследуют свойства и методы от существующих классов. Это упрощает создание сложных объектов и уменьшает количество дублирующегося кода.  Полиморфизм. В ООП полиморфизм означает, что один и тот же метод может быть реализован по-разному в зависимости от типа объекта, на который он вызван. Это позволяет избежать ошибок при вызове методов с неправильным типом объекта.  Абстракция. Абстракция позволяет отделить реализацию от интерфейса объекта, что делает код более понятным и упрощает его сопровождение. | Сложность. ООП требует более высокого уровня абстракции и более сложного синтаксиса, чем другие парадигмы программирования. Это может привести к увеличению времени разработки и усложнению сопровождения кода.  Неэффективность. ООП может быть неэффективным при работе с большими объемами данных, так как объекты требуют больше памяти и времени для обработки.  Ограничения. ООП имеет ряд ограничений, например, невозможность работы с динамическими типами данных или невозможность использования процедур и функций.  Сложности в тестировании. ООП предполагает использование наследования и полиморфизма, что может усложнить тестирование кода. | class Calculator {  var lastOperation; // атрибут класса  // процедура из прошлого примера превратилась в метод  sum (a,b) {  this.lastOperation = ‘sum’;  return a + b;  }  // процедура из прошлого примера превратилась в метод  multiply (a,b) {  this.lastOperation = ‘multiply’;  return a \* b;  }  }  // Основная программа  number1 = 2;  number2 = 4;  calc = new Calculator (); // создаем объект класса Calculator  // вызываем методы для объекта calc  result = calc.sum (number1, number2);  result = calc.multiply (number1, number2); |
| 7 | Функциональное  Программирование | В области функционального программирования парадигма определяет подход к написанию программ на языке, который использует функциональное программирование.  Основные идеи парадигмы функционального программирования:  Функциональный подход. Программы на языке функционального программирования определяются как функции, которые принимают аргументы и возвращают результаты. Функции не имеют побочных эффектов и не изменяют состояние программы.  Абстракция. Программы на функциональном языке абстрагируются от деталей реализации, что позволяет сосредоточиться на логике решения задачи.  Рекурсия. Рекурсивные функции могут быть написаны на функциональном языке, что делает его более мощным инструментом для решения сложных задач.  Оптимизация. Функциональные программы могут быть оптимизированы автоматически, что повышает производительность и эффективность работы программы.  Тип-безопасность. Тип-безопасность является одним из ключевых преимуществ функционального языка. Она гарантирует правильность типов данных и предотвращает ошибки, связанные с неправильным использованием типов.  Принципы функционального программирования включают:  – Императивное программирование: Это означает, что программа должна быть написана в виде последовательности инструкций, которые выполняются одна за другой. – Объектно-ориентированное программирование: Программа должна быть организована в виде объектов, которые имеют свойства и методы. – Функциональное программирование: Программа должна быть написана как функция, которая принимает аргументы и возвращает результат. – Абстрактное программирование: Программист должен абстрагироваться от деталей реализации программы и сосредоточиться на логике ее работы. – Рекурсивное программирование: Рекурсия используется для решения сложных задач путем повторного вызова функции.  Объекты функционального программирования - это функции, которые используются для решения задач в данной области. Они могут быть использованы для создания новых функций, обработки данных, управления состоянием программы и многого другого. | Haskell, Lisp, Scala | – Больше возможностей для оптимизации кода и повышения производительности. – Более безопасное и надежное программирование, так как ошибки типов данных можно обнаружить на этапе компиляции, а не во время выполнения программы. – Возможность использования функционального подхода для решения различных задач, таких как обработка данных, управление состоянием программы, создание алгоритмов и т.д. – Простота и понятность кода, что облегчает его поддержку и модификацию. | – Необходимость изучения нового языка программирования и его особенностей. – Ограниченность применения функционального программирования для некоторых типов задач, например, для разработки игр или приложений с графическим интерфейсом. – Сложность реализации некоторых алгоритмов, требующих использования рекурсии или итерации. – Требование более высокой квалификации программистов, так как функциональное программирование требует более глубокого понимания принципов работы языка и его основных конструкций.  Ответь иначе  Хороший ответ  Плохой ответ  Справка |  |
| 8 | Логическое  программирование | Логическое программирование — это парадигма программирования, основанная на использовании логических выражений и правил вывода для решения задач.  Основными идеями логического программирования являются:  Использование логических выражений для представления информации и выполнения вычислений.  Применение правил вывода, которые позволяют выводить новые утверждения из имеющихся утверждений и правил.  Разделение программы на модули, каждый из которых выполняет определенную задачу.  Использование метаданных для описания структуры программы и ее свойств.  Принципы логического программирования включают следующие:  Императивное программирование (imperative programming) — выполнение программы пошагово, следуя инструкциям. Объектно-ориентированное программирование (object-oriented programming) — разделение программы на объекты, которые взаимодействуют между собой через методы и свойства. Функциональное программирование (functional programming) — использование функций для обработки данных и решения задач. Абстрактное программирование (abstract programming) — абстрагирование от деталей реализации программы и сосредоточение на ее логике. Рекурсивное программирование — повторное использование функций при решении сложных задач | Phorolog, Mercury, Datalog | – Логическое программирование позволяет создавать программы, которые автоматически выводят новые утверждения на основе имеющихся правил и данных. – Оно может быть использовано для решения задач, связанных с обработкой естественного языка, логикой и другими областями, где требуется обработка сложных логических выражений. – Логическое программирование может быть легко интегрировано с другими языками программирования, что упрощает разработку сложных приложений. – Оно обеспечивает высокую степень модульности и гибкости, что позволяет легко изменять и расширять программы. | – Для создания программ на логическом языке необходимо знать логические правила и уметь работать с ними. – Программы на логическом языке могут быть сложными для понимания и отладки, особенно если они содержат много правил и условий. – Некоторые задачи, такие как разработка игр или графических приложений, могут быть лучше решены с помощью других парадигм программирования. | птица (голубь).  есть\_крылья (ворона).  умеет\_летать (ворона).  есть\_крылья (пингвин).  умеет\_плавать (пингвин).  птица (Объект):- умеет\_летать (Объект), есть\_крылья (Объект).. |

**3. Сделать выводы о том, какие из парадигм программирования используются наиболее часто.**

Наиболее часто используемые парадигмы программирования включают объектно-ориентированное, функциональное и логическое программирование. Каждая из этих парадигм имеет свои преимущества и недостатки, и выбор конкретной парадигмы зависит от конкретной задачи и требований проекта.

**4. Для своего индивидуального задания привести обоснование того какие парадигмы программирования вы будете использовать.**

Для разработки своего индивидуального задания я выберу язык программирования, который наилучшим образом соответствует вашим потребностям и возможностям. Для такой задачи лучше всего подойдёт язык программирования Python, как для серверной, так и для клиентской части.

Выбор библиотек Flask и Kivy обоснован легкостью обработки запросов на серверной стороне и быстротой разработки на фреймворке Kivy за счёт большого количества готовых и легких в использовании виджетов.

Flask – это фреймворк для создания веб-приложений на языке программирования Python, использующий набор инструментов Werkzeug, а также шаблонизатор Jinja2.

Kivy — это бесплатная платформа Python с открытым исходным кодом для разработки мобильных приложений и другого мультисенсорного прикладного программного обеспечения с естественным пользовательским интерфейсом.

Преимущества Python:

Простой и читаемый синтаксис: Python разработан так, чтобы код был легко читаемым и понятным. Это делает его отличным выбором для начинающих программистов и способствует повышению производительности разработчиков.

Кросс-платформенность: Python поддерживается на множестве операционных систем, что позволяет запускать код на различных платформах без изменений.

Большое сообщество: Python имеет активное сообщество разработчиков, что означает, что вы можете легко найти поддержку, библиотеки и ресурсы для решения задач.

Множество библиотек и фреймворков: Python предоставляет огромное количество библиотек и фреймворков, которые упрощают разработку различных приложений, включая веб-сайты, научные вычисления, искусственный интеллект и многое другое.

Широкое использование: Python применяется в различных областях, включая веб-разработку, анализ данных, машинное обучение, научные исследования, автоматизацию задач, игры и многое другое.

Открытый исходный код: Python - это свободное программное обеспечение, что означает, что вы можете использовать его бесплатно и иметь доступ к исходному коду для внесения изменений.

Хорошая документация: Python имеет хорошо разработанную и поддерживаемую документацию, что облегчает изучение языка и решение проблем.

Интеграция с другими языками: Python легко интегрируется с языками программирования, такими как C, C++, и Java, что позволяет использовать существующий код и библиотеки.

Широкая поддержка сторонних разработчиков: Существует множество сторонних инструментов и ресурсов, которые расширяют возможности Python.

Продвинутые возможности: Python поддерживает асинхронное программирование, многопоточность, многозадачность и другие продвинутые функции для разработки сложных приложений.

**Задание 3. Изучение правил формирования листинга программ.**

1. Изучить краткие теоретические сведения по теме (если информация отсутствует в конспекте, то необходимо ее законспектировать).

2. Оформить листинги программ в соответствии с приведенными рекомендациями по оформлению программ.

Необходимо привести примеры оформленного листинга для следующих парадигм программирования:

⎯ Структурное программирование (в программе должны использоваться процедуры, функции).

⎯ Модульное программирование (привести пример оформленного модуля).

⎯ Объектно-ориентированное программирование (привести пример оформленного класса).

⎯ Структурное программирование (в программе должны использоваться процедуры, функции).

#include <stdio.h>  
// Прототипы функций  
int add(int a, int b);  
int subtract(int a, int b);  
int main() {  
int x = 10;  
int y = 5;  
int result;  
// Вызов функций  
result = add(x, y);  
printf("Сумма: %d\n", result);  
result = subtract(x, y);  
printf("Разность: %d\n", result);  
return 0;  
}  
  
// Определение функций  
int add(int a, int b) {  
return a + b;  
}  
int subtract(int a, int b) {  
return a - b;  
}

⎯ Модульное программирование (привести пример оформленного модуля).

# Модуль calculations.py  
def add(a, b):  
return a + b  
def subtract(a, b):  
return a - b  
def multiply(a, b):  
return a \* b  
# Модуль main.py  
import calculations  
x = 10  
y = 5  
result\_add = calculations.add(x, y)  
result\_subtract = calculations.subtract(x, y)  
result\_multiply = calculations.multiply(x, y)  
print(f"Сумма: {result\_add}")  
print(f"Разность: {result\_subtract}")  
print(f"Произведение: {result\_multiply}")

– Объектно-ориентированное программирование (привести пример оформленного класса).

class Rectangle:  
def *init*(self, width, height):  
self.width = width  
self.height = height  
def area(self):  
return self.width \* self.height  
def perimeter(self):  
return 2 \* (self.width + self.height)  
# Использование класса  
rect = Rectangle(5, 3)  
print(f"Площадь прямоугольника: {rect.area()}")  
print(f"Периметр прямоугольника: {rect.perimeter()}")

2. Оформить отчет о проделанной работе.

3. Защитить работу и сдать ее преподавателю.

**Контрольные вопросы:**

1. Язык программирования - это формальная система символов и правил, используемая для написания компьютерных программ. Алфавит - набор символов, синтаксис - правила, определяющие структуру программы, семантика - смысл и действие программы, стандарт - набор соглашений и спецификаций языка. Примеры классификации: высокоуровневые (Python), низкоуровневые (C), функциональные (Haskell), объектно-ориентированные (Java).

2. Критерии выбора языка программирования включают производительность, доступность библиотек, удобство разработки, сообщество разработчиков и применимость к конкретной задаче.

3. Парадигма программирования - это стиль или методология разработки программного кода. Известные стили: императивное, функциональное, объектно-ориентированное.

4. Структурное программирование - организует код в логические блоки с использованием последовательных структур управления (последовательность, ветвление, циклы). Процедурное программирование - базируется на использовании процедур и функций для организации кода в более мелкие, переиспользуемые части. Модульное программирование - разделяет программу на модули (независимые компоненты) для упрощения разработки и обслуживания. Объектно-ориентированное программирование - организует код в объекты, объединяющие данные и методы, что способствует абстракции и инкапсуляции. Функциональное программирование - основывается на функциях как на основных строительных блоках, поддерживает неизменяемость данных и отсутствие побочных эффектов. Логическое программирование - использует логические утверждения и инференцию для решения задач, часто ассоциируется с языком Prolog

5. Стандарт оформления программного кода - это набор правил, которые помогают улучшить читаемость и поддерживаемость кода. Необходимость качественного формирования кода включает улучшение совместной разработки и облегчение отладки. Правила формирования включают отступы, выравнивание, комментарии и др.

6. Правила оформления кода: Венгерская нотация - добавление префиксов к идентификаторам, указывающих тип данных (например, "strName" для строки). Верблюжья нотация - начало каждого нового слова с большой буквы, без пробелов (например, "camelCase"). Грамотный выбор идентификаторов - понятные имена переменных, констант, структур; соглашения о длине (например, "counter" вместо "c"). Отступы - горизонтальные и вертикальные отступы для лучшей читаемости кода. Использование комментариев - добавление пояснений к коду для понимания его назначения. Разбиение задачи на подзадачи - разделение больших задач на более мелкие, что делает код более структурированным. Правила оформления процедур, функций, модулей, классов - четкое описание и разделение функциональности, использование адекватных имен и документации.