# Жизненный цикл программного обеспечения

## Основные понятия

**Алгоритм** – точное предписание, определяющее вычислительный процесс, идущий от изменяемых начальных данных к конечному результату.

Человек свою деятельность осуществляет в соответствие с каким-то ранее составленным алгоритмом: покупки в магазине (исходные данные: список покупок, деньги, результат: купленные товары, алгоритм: последовательность шагов, необходимая для того, чтобы купить эти товары).

**Язык программирования** (ЯП) - совокупность средств и правил для представления алгоритма в виде пригодном для выполнения вычислительной машиной.

**Программа** – алгоритм, записанный на ЯП.

**Программирование** – процесс создания компьютерных программ.

Сначала всегда разрабатывается алгоритм, а потом он записывается на одном из языков программирования. ЯП – это искусственные языки, которые имеют очень жесткие правила записи команд. Совокупность этих правил образует **синтаксис** ЯП, смысл конструкций – его **семантику**.

**Программное обеспечение** – набор компьютерных программ, процедур и связанной с ними документации и данных (ISO/IEC 12207).

## Свойства алгоритма

1. Массовость – алгоритм должен применяться не к одной задаче, а к целому классу подобных задач (квадратное уравнение).
2. Результативность – алгоритм должен приводить к результату за конечное число шагов (1:3=0,333(3)).
3. Понятность – каждое действие алгоритма должно быть понятно его исполнителю (входить в систему его команд).
4. Определенность (или точность) алгоритма - каждая команда алгоритма должна однозначно определять действие исполнителя, т.е. на каждом шаге исполнитель знает, что ему делать.
5. Дискретность – процесс должен быть описан с помощью отдельных неделимых шагов (операций).

Примеры алгоритмов: правила выполнения арифметических операций в позиционных СС, описание перемещения человека а городе из одной точки в другую, алгоритм деления отрезка пополам, рецепт приготовления блюда.

## Способы представления алгоритма

1. Словесное описание.
2. Графическое описание (блок-схема).
3. Язык программирования высокого уровня.

Блок-схема – это последовательность блоков, предписывающих выполнение определенных операций, и связей между этими блоками. Внутри блоков указывается информация об операциях, подлежащих выполнению. Конфигурация и размеры блоков, а также порядок графического оформления блок-схем регламентированы ГОСТ 19002-80 и ГОСТ 19003-80 "Схемы алгоритмов и программ". В январе 1992 года введен новый ГОСТ 19–701–90.

## Основные алгоритмические структуры

Любой алгоритм может быть построен из трех базовых структур:

* следование (последовательность),
* ветвление,
* цикл.

**Линейным** называется алгоритм, в котором отдельные предписания выполняются последовательно в порядке записи независимо от значений исходных данных и промежуточных результатов.

**В разветвляющихся** алгоритмах вычислительный процесс проходит по одной из возможных ветвей.

**Циклическими** называются алгоритмы, у которых выполнение некоторых операторов осуществляется многократно с одними и теми же или модифицированными (изменяющимися) данными. Циклы бывают:

* + итерационные,
  + арифметические.

## Языки программирования

* **Ранние языки программирования**

Время появления: 1940-е г.г.

Краткая характеристика: линейная последовательность элементарных инструкций «низкого уровня»

Преимущества:

* + высокая вычислительная эффективность

Недостатки:

* + существенная зависимость от среды вычислений

Примеры:

• машинные коды, ассемблеры.

* **Императивные (процедурные) языки программирования**

Время появления: 1950-е г.г.

Краткая характеристика: программа – последовательность инструкций-операторов, включающих блоки типичных действий – процедуры или функции.

Преимущества:

* + более высокий уровень абстракции;
  + меньшая машинная зависимость;
  + содержательная значимость текстов программ;
  + унификация программного кода;
  + повышение производительности труда программистов.

Недостатки:

* + меньшая эффективность программного кода.

Примеры:

* Fortran, ALGOL, PL/1, APL, BPL, COBOL, Pascal, C, Basic.
* **Декларативные языки программирования**

Время появления: 1960-е г.г.

Краткая характеристика: программа – описание действий, которые необходимо осуществить

* **Функциональные языки программирования**

Время появления: 1960-е г.г.

Краткая характеристика: программа – функция, аргументы которой, возможно, также являются функциями

Преимущества:

* полностью автоматическое управление памятью компьютера («сборка мусора»);
* простота повторного использования фрагментов кода;
* расширенная поддержка функций с параметрическими аргументами (параметрический полиморфизм);
* абстрагирование от машинного представления данных;
* прозрачность реализации рекурсивных функций;
* удобство символьной обработки данных (списки, деревья)

Недостатки:

* нелинейная структура программы;
* относительно низкая эффективность

Примеры:

LISP, SML, CaML, Haskell, Miranda, Hope

* **Логические языки программирования**

Время появления: 1970-е г.г.

Краткая характеристика: программа – совокупность правил или логических высказываний с причиной и следствием

Преимущества:

* + высокий уровень абстракции;
  + удобство программирования логики поведения;
  + удобство применения для экспертных систем;
  + механизм откатов (backtrack)

Недостатки:

* + ограниченный круг задач;
  + нелинейная структура программы;
  + недостаточно эффективная реализация

Примеры:

* Prolog, Mercury.
* Преимущества **декларативных языков программирования** 
  + простота верификации и тестирования программ;
  + строгость математической формализации;
  + высокая степень абстракции.

Недостатки:

* + сложность эффективной реализации;
  + необходимость фундаментальных математических знаний.
* **Объектно-ориентированные языки программирования**

Время появления: 1970-е г.г.

Краткая характеристика:

программа – описание объектов, их совокупностей,

отношений между ними и способов их взаимодействия

Преимущества:

* + интуитивная близость к произвольной предметной области;
  + моделирование сколь угодно сложных предметных областей;
  + событийная ориентированность;
  + высокий уровень абстракции;
  + повторное использование описаний;
  + параметризация методов обработки объектов

Недостатки:

* + сложность тестирования и верификации программ

Примеры:

C++, Visual Basic, Java, C#, Eiffel, Oberon

## Жизненный цикл (ЖЦ) программного обеспечения

Основным понятием программной инженерии является понятие *жизненного цикла.* Основной нормативный документ, регламентирующий ЖЦ ПО – стандарт ISO/IEC 12207: 1995 “Information Technology – Software Life Cycle Processes” (ГОСТ Р ИСО/МЭК 12207-99). Под жизненным циклом ПО понимается совокупность процессов, связанных с последовательным изменением состояния ПО от формирования исходных требований к нему до окончания его эксплуатации.

В ходе жизненного цикла ПО проходит через

* анализ предметной области,
* проектирование,
* кодирование,
* тестирование,
* сопровождение.

Каждый вид представляет собой набор действий, выполняемых для решения одной задачи или группы тесно связанных задач в рамках разработки и поддержки эксплуатации ПО.

## Модели жизненного цикла

Жизненный цикл ПО определяет «что», но не «как» выполнять в процессе программной инженерии.

Подходы к жизненному циклу ПО могут быть грубо разделены на следующие категории:

* Каскадная (водопадная) или последовательная.
* Итеративная и инкрементальная – эволюционная (гибридная, смешанная).

## Жизненный цикл «водопад с обратной связью»

Каскадная модель жизненного цикла («модель водопада», англ. waterfall model) была предложена в 1970 г. [Уинстоном Ройсом](http://ru.wikipedia.org/w/index.php?title=%D0%A3%D0%B8%D0%BD%D1%81%D1%82%D0%BE%D0%BD_%D0%A0%D0%BE%D0%B9%D1%81&action=edit&redlink=1). Она предусматривает последовательное выполнение всех этапов проекта в строго фиксированном порядке. Переход на следующий этап означает полное завершение работ на предыдущем этапе. Требования, определенные на стадии формирования требований, строго документируются в виде технического задания и фиксируются на все время разработки проекта. Каждая стадия завершается выпуском полного комплекта документации, достаточной для того, чтобы разработка могла быть продолжена другой командой разработчиков.

Этапы проекта в соответствии с каскадной моделью:

* Формирование требований (анализ);
* Проектирование;
* Реализация;
* Тестирование;
* Внедрение;
* Эксплуатация и сопровождение.

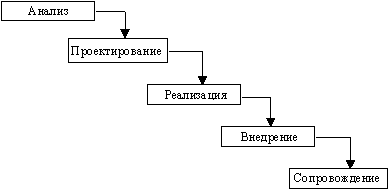


Рисунок 1 Каскадная модель жизненного цикла

Каскадный подход хорошо зарекомендовал себя при построении ИС, для которых в самом начале разработки можно достаточно точно и полно сформулировать все требования, с тем чтобы предоставить разработчикам свободу реализовать их как можно лучше с технической точки зрения. В эту категорию попадают сложные расчетные системы, системы реального времени и другие подобные задачи. Однако, в процессе использования этого подхода обнаружился ряд его недостатков, вызванных прежде всего тем, что реальный процесс создания ПО никогда полностью не укладывался в такую жесткую схему. В процессе создания ПО постоянно возникала потребность в возврате к предыдущим этапам и уточнении или пересмотре ранее принятых решений. В результате реальный процесс создания ПО принимал следующий вид (рис. 2):

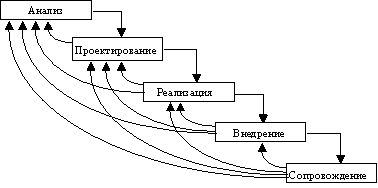


Рисунок 2. Реальный процесс разработки ПО по каскадной схеме

Основным недостатком каскадного подхода является существенное запаздывание с получением результатов. Согласование результатов с пользователями производится только в точках, планируемых после завершения каждого этапа работ, требования к ИС "заморожены" в виде технического задания на все время ее создания. Таким образом, пользователи могут внести свои замечания только после того, как работа над системой будет полностью завершена. В случае неточного изложения требований или их изменения в течение длительного периода создания ПО, пользователи получают систему, не удовлетворяющую их потребностям. Модели (как функциональные, так и информационные) автоматизируемого объекта могут устареть одновременно с их утверждением.

## Итеративный ЖЦ

Итеративный жизненный цикл предполагает шаги — улучшенные или расширенные версии изделия в конце каждой итерации. По этой причине итеративная модель жизненного цикла иногда называется эволюционной или пошаговой.

Итеративный жизненный цикл также предполагает наличие конструкций — исполняемого кода, полученного в итерации. Конструкция — вертикальный срез системы. Это не подсистема. Область действия конструкции — вся система, но с некоторыми уменьшенными функциональными возможностями, с упрощенными пользовательскими интерфейсами, с ограниченной многопользовательской поддержкой, меньшим объемом работы и другими подобными ограничениями.

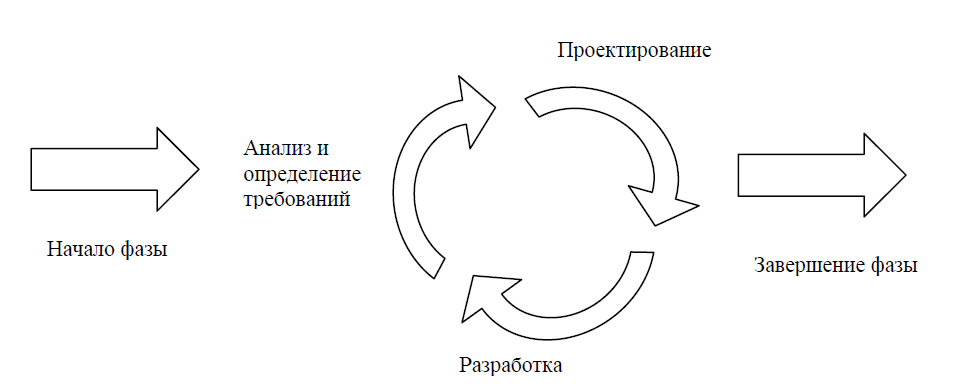
**

Рисунок 3. Итеративный ЖЦ

## Основные участники процесса создания ПО

1. Заказчик – определяет требования к разрабатываемой программе. Различают функциональные и нефункциональные требования.

Функциональные требования представляют собой перечень возможностей, предоставляемых программой пользователю при решении задачи (функциональность программы). К нефункциональным требованиям относится: состав входных и выходных данных, способ получения входных данных от пользователя и способ выдачи выходных данных пользователю, ограничения на время выполнения программы, диапазон значений данных, точность представления данных и т.п. Зафиксированный установленным способом перечень требований будем называть спецификацией программы.

2. Проектировщик – определяет структуру разрабатываемой программы, распределение функциональности между частями программы и их взаимодействие по управлению и обмену данными.

3. Разработчик – определяет способ реализации требуемой функциональности в каждой из частей программы и разрабатывает код программы на языке, доступном исполнителю.

4. Пользователь – применяет разработанную программу по назначению для получения результатов обработки данных.

При разработке спецификации выделяется часть реального мира – предметная область (Рис.4). Предметная область включает только те предметы и понятия, которые имеют отношение к решаемой задаче.



Рисунок 4

Для выделенной предметной области составляется ее формализованное описание – модель предметной области. Модель предметной области и определение спецификации программы выполняется, как правило, системным аналитиком и согласовывается с заказчиком и проектировщиком. Модель предметной области, отражающую взгляд на нее заказчика, называют логической моделью, а представление о программе с точки зрения заказчика называют прагматикой программы.

Очевидно, что для заказчика естественным является желание использовать для построения логической модели понятийный аппарат и средства формализации, принятые в предметной области.

С другой стороны, разработчик должен определить семантику программы. Семантика программы - это представление о программе с точки зрения ЭВМ, которая будет ее выполнять. Другими словами, разработчик должен сформировать физическую модель предметной области, выраженную с использованием понятий и средств формализации, доступных для восприятия ЭВМ.

Для аппаратных средств ЭВМ непосредственно доступна программа, выраженная в терминах машинных команд. Разработчик для определения семантики программы может использовать так называемые языки высокого уровня (ЯВУ). Понятийный аппарат ЯВУ может существенно отличаться от понятийного аппарата машинных команд. При использовании ЯВУ ЭВМ должна быть снабжена комплексом программных средств, которые обеспечивают выполнение на ЭВМ программ, написанных на ЯВУ. Аппаратные средства ЭВМ с программными средствами выполнения программ образуют среду исполнения программ.

В дальнейшем при рассмотрении семантики программы и физической модели предметной области будем исходить из того, что они определяются разработчиком на языке высокого уровня.

## Этапы ЖЦ ПО

## Анализ требований

**Требования пользователя** — это утверждения на естественном языке плюс диаграммы, содержащие сведения, какие услуги ожидаются от системы, и ограничения, при которых система должна работать.

**Анализ требований** включает действия по их определению и составлению их списка. В современной практике анализу требований помогает хорошая степень технической строгости, и поэтому эти требования иногда отождествляют с **техническими требованиями**.

**Определение требований**, оказывается, одна из самых больших проблем любого жизненного цикла разработки ПО. Пользователям часто неясно, что они требуют от системы. Часто они не знают реальные требования, преувеличивают их, предъявляют требования, которые противоречат требованиям коллег и т. д. Имеется также риск, как и в любом общении между людьми, что истинное значение требования будет неправильно истолковано.

Анализ требований завершается созданием **технического задания**.

Большинство организаций использует некоторые шаблоны для технического задания. Шаблон определяет структуру документа и дает руководящие принципы того, как его писать. Основная часть технического задания содержит модели и описания сервисов и ограничений системы.

**Сервисы системы**(то, что система должна делать) часто делятся на функциональные требования и требования к данным.

Ограничения системы(то, чем система ограничена) включают соображения, связанные с пользовательским интерфейсом, работой, безопасностью, эксплуатационными условиями, политическими и юридическими ограничениями и т. д.

## Проектирование системы

Проектирование ПО — это описание структуры ПО, которое будет реализовано, данных, которые являются частью системы, интерфейсов между компонентами системы и, иногда, используемых алгоритмов. Это определение совместимо с определением системы ПО как объединения структур данных и алгоритмов.

## Реализация

**Реализация** в большей мере связана с программированием. Но программирование подразумевает не только группу людей, сидящих в общем помещении и кодирующих на некотором языке программирования в соответствии со спецификацией проекта. Программирование предполагает намного более интеллектуальные требования, чем это. Как указано в предыдущих разделах, проекты будут «недоопределены» в некоторых областях, когда они попадают к программистам, особенно в области проектирования алгоритмов. Завершение спецификаций требует дополнительного проектирования прежде, чем можно будет начать кодирование. В этом смысле программист — тоже проектировщик. Программист — *инженер, имеющий дело с компонентами*. Сегодняшнее программирование редко выполняется на пустом месте. Большая часть программирования основана на многократном использовании уже созданных компонентов. Это означает, что программист должен иметь знание о компонентах ПО и должен знать, как найти это ПО, чтобы добавить к нему новые закодированные компоненты приложения. Это трудный вопрос.

**Отладка** — это процесс удаления из ПО ошибок в программах. Ошибки в синтаксисе программы и некоторые логические ошибки могут быть определены и исправлены средствами отладки. Другие ошибки и дефекты должны быть обнаружены во время тестирования программы. **Тестирование** может иметь форму просмотра кода (сквозной контроль и инспекции) или может основываться на выполнении программы (наблюдение за поведением программы во время ее выполнения). Имеются два вида *тестирования, основанного на выполнении программы*: **тестирование на основе технических требований** (тестирование черного ящика) и **тестирование на основе кода** (тестирование белого ящика). Оба вида используют ту же самую стратегию задания программе входных данных и наблюдения, тот ли выходной результат получается, который ожидался.

## Интеграция и внедрение

**Интеграция** собирает приложение из набора компонентов, предварительно созданных и проверенных.

**Внедрение** — передача системы клиентам для использования в производстве.

Интеграция ПО означает переход от «программирования в малом» к «программированию в большом». Информационные системы предприятий — все достаточно большие и сложные системы и для них интеграция— существенная стадия в жизненном цикле. Это отдельная стадия, даже если ее иногда трудно отделить от реализации, как, например, в случае **непрерывной интеграции**в быстрой разработке.

Перед внедрением ПО **системно тестируется**разработчиками в реальных условиях. Оно иногда называется **альфа***-***тестированием**. За альфа-тестированием следуют **приемочные испытания** специалистами пользователя. Это иногда называется **бета-тестированием***.*

## Процесс функционирования и сопровождения

**Процесс функционирования** представляет стадию жизненного цикла, когда программный продукт ежедневно используется, а работа предыдущей системы (ручной или автоматизированной) постепенно сокращается. **Постепенное сокращение**— обычно поэтапный процесс. В ситуациях, где это возможно и выполнимо, организация обычно использует новую и старую системы параллельно в течение некоторого времени. Это обеспечивает отход назад, если новое изделие не удовлетворяет бизнес-требованиям.

В литературе сопровождение обычно разделяется на:

* **корректирующее**(действия по обслуживанию) — выявление дефектов и ошибок, обнаруженных при работе;
* **адаптивное**— изменение ПО в ответ на изменения в вычислениях или в бизнес-среде;
* **совершенствующее**— развитие изделия путем добавления новых особенностей или улучшения его качества.

## Парадигма программирования

Понятийный аппарат, используемый для разработки моделей предметной области, называют парадигмой программирования.

Наиболее распространенными парадигмами программирования являются процедурно-ориентированное программирование и объектно-ориентированное программирование. Существуют и другие парадигмы программирования, в частности логическое программирование и функциональное программирование. Рассмотрение этих парадигм выходит за рамки данного курса.

## Процедурно-ориентированное программирование

Рассмотрим основные понятия, лежащие в основе процедурно-ориентированного программирования.

В основе парадигмы лежит понятийный аппарат, отражающий принципы логической организации ЭВМ классической архитектуры. Предметная область рассматривается как процесс воздействия на входные данные с целью их преобразования в выходные данные.

Для представления логической модели используются графические диаграммы, дополненные текстовым описанием. В логической модели определяются:

* Входные данные
* Источники входных данных
* Выходные данные
* Потребители выходных данных
* Данные, подлежащие долговременному хранению (накопители данных)
* Процессы преобразования входных данных в выходные данные

Структурно программа представляет собой набор процедур – подпрограмм. Взаимодействие подпрограмм организовано по иерархическому принципу. Выполнение программы начинается с главной подпрограммы.

Выполняющаяся подпрограмма может вызвать подчиненную подпрограмму. Выполнение подпрограммы приостанавливается до завершения вызванной подпрограммы и возврата из нее. Программа считается выполненной после завершения главной подпрограммы.

## Объектно-ориентированное программирование

В основе парадигмы лежит представление предметной области в виде множества **объектов**, взаимодействующих между собой.

**Объект** *-* мыслимая или реальная сущность, обладающая характерным поведением и отличительными характеристиками и являющаяся важной для данной предметной области.

Характеристики объекта называют ***атрибутами.*** Атрибуты определяют состояние объекта***.***

Объект может иметь определенный набор действий (**операций**), которые можно произвести над атрибутами объекта. Набор операций определяет поведение объекта.

Множество объектов, которые имеют одинаковый набор атрибутов и операций, образуют **класс объектов.**

## Тестирование

Тестирование – выполнение программы с целью обнаружения в ней ошибок.

Отладка – определение места возникновения ошибки и ее исправление.

Ошибки в программе бывают трех типов:

* Синтаксические.
* Ошибки выполнения программы.
* Семантические.

Тест – набор исходных данных, для которых заранее известен результат.

Если результаты работы теста не совпадают известными значениями, следовательно, в программе имеются ошибки.

Успешный тест – тест, который выявил ошибку.

Отладка заканчивается, когда достаточное количество тестов закончилось неуспешно.

## Тестирование по стратегии «черного ящика»

**Чёрный ящик** - тестирование функционального поведения программы с точки зрения внешнего мира (текст программы не используется).

1. **Тестирование функций**: проверка всех функций, выполняемых программой.
2. **Тестирование классов входных данных**: делим данные на классы и считаем, что если программа работает правильно на одном наборе входных данных из этого класса, то она будет работать правильно и на других наборах данных из этого же класса.
3. **Тестирование классов выходных данных.**
4. **Тестирование области допустимых значений (границ класса):**
   * + - Нормальные условия (середина класса).
       - Граничные условия.
       - Исключительные условия (выход за границу класса).
5. **Тестирование длины набора данных:** 
   * + - Пустой набор.
       - Единичный набор.
       - Слишком короткий набор.
       - Набор минимально короткой длины.
       - Нормальный набор.
       - Набор из нескольких частей.
       - Набор максимально возможной длины.
       - Слишком длинный набор.
6. **Тестирование упорядоченности** (для сортировок и поиска экстремумов):
   * + - Данные неупорядочены.
       - Данные упорядочены в прямом порядке.
       - Данные упорядочены в обратном порядке.
       - В наборе имеются повторяющиеся значения.
       - Экстремальное значение находится в середине набора.
       - Экстремальное значение находится в начале набора.
       - Экстремальное значение находится в конце набора.
       - В наборе имеются совпадающие экстремальные значения.

## Тестирование по стратегии «белого ящика»

Тестовые данные получаются путем анализа логики работы программы.

Критерии:

1. **Покрытие операторов**: каждый оператор должен быть выполнен хотя бы один раз.

a=0;

if(x>5) a=10;

b=x/a;//деление на 0 при x==0 – не обнаружим при 1-ом критерии

1. **Покрытие ветвей:** каждая ветвь в программе выполняется хотя бы один раз.

a=5;

while(a>x) a=a-1;

b=1/a;//деление на 0 при x==0 – не обнаружим при 2-ом критерии

1. **Покрытие путей:** каждый путь должен быть пройден хотя бы один раз. Сложно выполнить, т.к. путей в циклических алгоритмах может быть бесконечное множество.

Критерии проверки сложных условий:

1. **Критерий покрытия условий**: каждое простое условие получает значение истина.

if(a<b)||(c==0) d=1; else d=1/c;//ошибка не найдена при a=2, b=1, c=0

1. **Критерий покрытия решений/условий** объединяет вместе критерий покрытия ветвей и критерий покрытия условий.

if(a==0)||(b==0)||(c==0) d=1; else d=1/(a+b);

1. **Критерий комбинаторного покрытия условий:** хотя бы один раз должна выполняться любая комбинация простых условий.

## Минимальное грубое тестирование

МГТ – критерий покрытия решений/условий + дополнительные требования по проверке циклов:

1. Каждая ветвь в программе выполняется хотя бы один раз.
2. Каждое простое условие получает значение истина.
3. Каждый цикл проверяется при нулькратном, однократном и многократном повторении.

## Решение задачи

Найти корни квадратного уравнения Ах2+Вх+С=0

**Этап 1. Анализ**

|  |  |  |
| --- | --- | --- |
| Входные данные | Классы входных данных | Выходные данные |
| А, В, С | Не являются вещественными числами | Сообщение об ошибке |
| А, В, С | А=0, В=0, С=0 | х – любое число |
| А ,В, С | А=0, В=0, С!=0 | Уравнение не имеет корней |
| А, В, С | А=0, В!=0, С!=0 | Один корень х |
| А, В, С | А!=0, В!=0, С!=0,  D<0 | Комплексные корни |
| А, В, С | А!=0, В!=0, С!=0,  D=0 | Один корень х |
| А, В, С | А!=0, В!=0, С!=0,  D>0 | Два корня |

**Этап 2. Проектирование (блок-схема)**

**Этап 3. Кодирование.**

**Этап 4. Тестирование**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  |  | Ожидаемый результат | Полученный результат | Примечание |
| T1 | A= «а» | Повторный ввод | Повторный ввод | Ошибка в первом коэффициенте |
| T2 | В=«а» | Повторный ввод | Повторный ввод | Ошибка во втором коэффициенте |
| T3 | С=«а» | Повторный ввод | Повторный ввод | Ошибка во третьем коэффициенте |
| T4 | a=0,b=0, c=0 | х – любое число | х – любое число |  |
| T5 | a=0,b=0,c=1 | Уравнение не имеет корней | Уравнение не имеет корней |  |
| T6 | a=0,b=2,c=1 | -0.5 | -0.5 | Один корень |
| T7 | a=1, b=2, c=3 | Уравнение имеет комплексные корни | Уравнение имеет комплексные корни 2 | D<0 |
| T8 | a=1,b=5,c=1 | -4,79; 0,2 | -4,79; 0,2 | D>0 |
| T9 | a=1,b=2,c=1 | -1 | -1 | D=0 |

Проверка достаточности тестов по критериям черного ящика

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | Т1 | Т2 | Т3 | Т4 | Т5 | Т6 | Т7 | Т8 | Т9 |
| Одно из чисел с ошибкой | + | + | + |  |  |  |  |  |  |
| A=0 |  |  |  | + | + | + |  |  |  |
| B=0 |  |  |  | + | + |  |  |  |  |
| C=0 |  |  |  | + |  |  |  |  |  |
| D=0 |  |  |  |  |  |  |  |  | + |
| D<0 |  |  |  |  |  |  | + |  |  |
| D>0 |  |  |  |  |  |  |  | + |  |

Проверка достаточности тестов по критериям МГТ (белый ящик)

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | Т1 | Т2 | Т3 | Т4 | Т5 | Т6 | Т7 | Т8 | Т9 |
| do  {    } while (!ok);//A | 1 |  |  |  | + |  |  |  |  |  |
| n | + |  |  |  |  |  |  |  |  |
| do  {    } while (!ok);//B | 1 |  |  |  | + |  |  |  |  |  |
| n |  | + |  |  |  |  |  |  |  |
| do  {    } while (!ok);//C | 1 |  |  |  | + |  |  |  |  |  |
| n |  |  | + |  |  |  |  |  |  |
| if (A == 0) | + |  |  |  | + | + | + |  |  |  |
| - |  |  |  |  |  |  | + | + | + |
| if (B == 0) | + |  |  |  | + | + |  |  |  |  |
| - |  |  |  |  |  | + | + | + | + |
| if (C == 0) | + |  |  |  | + |  |  |  |  |  |
| - |  |  |  |  | + | + | + | + | + |
| if (D < 0) | + |  |  |  |  |  |  | + |  |  |
|  | - |  |  |  |  |  |  |  | + | + |
|  | + |  |  |  |  |  |  |  | + |  |
| if (D == 0) | - |  |  |  |  |  |  |  |  | + |