Министерство образования Республики Беларусь
Учреждение образования
«Белорусский государственный университет
информатики и радиоэлектроники»

Факультет компьютерных систем и сетей

Кафедра программного обеспечения информационных технологий

А. И. Парамонов, Н. В. Лапицкая, С. Н. Нестеренков

ОСНОВЫ ПРОГРАММНОЙ ИНЖЕНЕРИИ

*Рекомендовано УМО по образованию в области информатики
и радиоэлектроники в качестве учебно-методического пособия
для специальности 6-05-0612-01 «Программная инженерия»*

Минск БГУИР 2023

УДК 004.42(076)
ББК 32.972я73
П18

Р е ц е н з е н т ы:

кафедра программной инженерии
учреждения образования «Белорусский государственный технологический университет» (протокол №2 от 10.10.2022);

специалист 2 категории ИООО «ЭПАМ СИСТЕМЗ»
кандидат технических наук, доцент В. Д. Левчук

Парамонов, А. И.
П18 Основы программной инженерии : учеб.-метод. пособие / А. И. Парамонов, Н. В. Лапицкая, С. Н. Нестеренков. – Минск : БГУИР, 2023. – 122 с. : ил.
ISBN 978-985-543-699-8.

Предназначено для студентов первого курса специальности «Программная инженерия». Содержит теоретический материал, практические работы, а также примеры выполнения практических задач.

Можно использовать для самостоятельной работы студентов указанной специальности очной, заочной и дистанционной форм обучения. Будет полезно студентам специальностей профиля образования 06 – информационно-коммуникационные технологии.

УДК 004.42(076)
ББК 32.972я73

ISBN 978-985-543-699-8

© Парамонов А. И., Лапицкая Н. В., Нестеренков С. Н., 2023
© УО «Белорусский государственный университет информатики и радиоэлектроники», 2023

СОДЕРЖАНИЕ

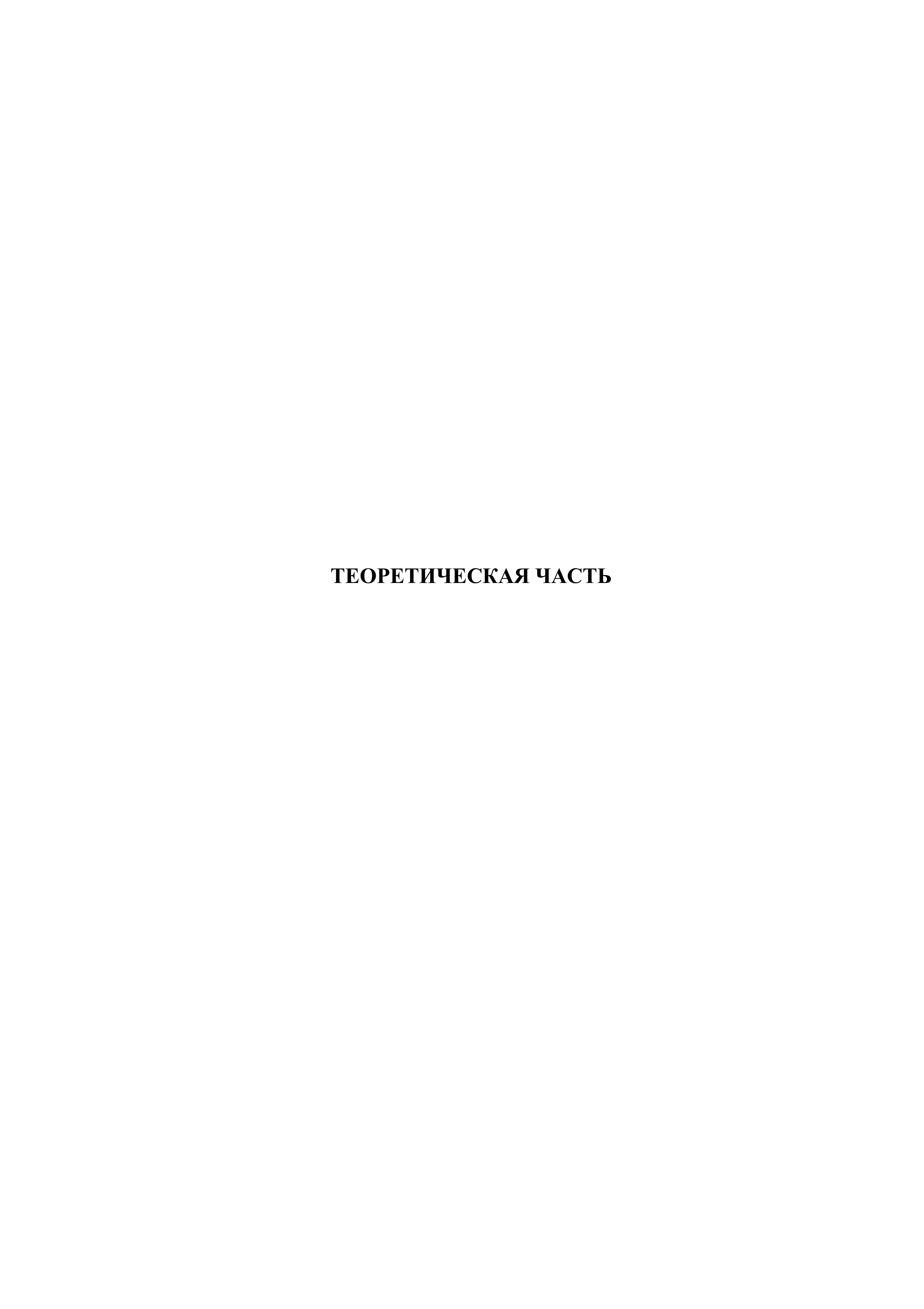
ВВЕДЕНИЕ ... 4
ТЕОРЕТИЧЕСКАЯ ЧАСТЬ
1. ОБЩИЕ СВЕДЕНИЯ О ПРОГРАММНОЙ ИНЖЕНЕРИИ ... 6
2. ОСНОВЫ КОМПЬЮТЕРНОЙ ТЕХНИКИ ... 8
2.1. Традиционная архитектура вычислительных систем ... 8
2.2. Системы счисления ... 9
2.3. Общие принципы кодирования чисел ... 13
2.4. Операции манипуляции данными в компьютере ... 17
2.5. Задания для самоконтроля ... 22
3. ОСНОВЫ КОНСТРУИРОВАНИЯ ПРОГРАММНЫХ ПРОДУКТОВ ... 24
3.1. Программное обеспечение как продукт ... 24
3.2. Классификация программных продуктов ... 26
3.3. Этапы создания программного продукта ... 28
3.4. Разработка требований к программному продукту ... 33
3.5. Системы контроля версий ... 37
3.6. Задания для самоконтроля ... 44
ПРАКТИЧЕСКАЯ ЧАСТЬ
Практическая работа №1. Программное обеспечение как продукт ... 46
Практическая работа №2. Выявление требований к программному продукту ... 50
Практическая работа №3. Разработка требований к программному продукту ... 52
Практическая работа №4. Основы моделирования программного продукта ... 53
Практическая работа №5. Методологии разработки программного продукта ... 54
Практическая работа №6. Организация и планирование проекта ... 56
Практическая работа №7. Средства конструирования программного продукта ... 63
Практическая работа №8. Системы контроля версий исходного кода ... 64
Практическая работа №9. Разработка эксплуатационной документации ... 95
ПРИЛОЖЕНИЕ 1. Правила перевода из одной системы счисления в другую ... 100
ПРИЛОЖЕНИЕ 2. Расчетные таблицы арифметических операций в различных системах счисления, используемых в компьютере ... 103
ПРИЛОЖЕНИЕ 3. Типовая структура содержания технического задания на разработку программного продукта ... 104
ПРИЛОЖЕНИЕ 4. Шаблон учебного технического задания на разработку программного продукта ... 106
ПРИЛОЖЕНИЕ 5. Основные сведения о схемах алгоритмов по стандарту ГОСТ 19.701–90 ... 112
Список использованных источников ... 119

ВВЕДЕНИЕ

Учебно-методическое пособие направлено на формирование у студентов систематизированного представления о современных подходах к конструированию программного обеспечения, методах программной инженерии и ее стандартах, процессах создания и эволюции сложных, многоверсионных, тиражируемых программных продуктов высокого качества, необходимого для практического использования на последующих этапах обучения и в профессиональной сфере деятельности будущего специалиста.

С учетом того, что освоение специальности «Программная инженерия» направлено на выработку компетенций и навыков по умелому использованию принципов информатики и компьютерных наук для их сочетания с инженерными подходами, разработанными для материального производства сложных корпоративных компьютерных систем, учебно-методическое пособие для первой специальной дисциплины в системе подготовки специалистов содержит обширный материал, знакомящий как с самой специальностью, так и с базовыми основами компьютерной техники и технологиями конструирования программного обеспечения.

Наряду с теоретическим материалом учебно-методическое пособие включает практические работы, отражающие все этапы жизненного цикла программного обеспечения.



ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

1. ОБЩИЕ СВЕДЕНИЯ О ПРОГРАММНОЙ ИНЖЕНЕРИИ

Программная инженерия подразумевает использование принципов информатики и компьютерных наук с инженерными подходами, разработанными для материального производства сложных корпоративных компьютерных систем.

Впервые термин «Software Engineering» был предложен в 1968 году. В 1972 году IEEE[1] выпустил первый номер Transactions on Software Engineering – труды по программной инженерии. Первый целостный взгляд на эту область профессиональной деятельности появился 1979 году, когда компьютерное общество IEEE подготовило стандарт IEEE Std 730 по качеству программного обеспечения. В 1986 году IEEE выпустило IEEE Std 1002 Taxonomy of Software Engineering Standards. Наконец, в 1990 году началось планирование международных стандартов, в основу которых легли концепции и взгляды стандарта IEEE Std 1074 и результатов работы образованной в 1987 году совместной комиссии ISO/IEC JTC 1. В 1995 году группа этой комиссии ISO[2] SC7 Software Engineering выпустила первую версию международного стандарта ISO/IEC 12207 Software Life Cycle Processes. Этот стандарт стал первым опытом создания единого общего взгляда на программную инженерию. Издан соответствующий стандарт и в России – ГОСТ Р ИСО/МЭК 12207–99 (ГОСТ 12207, 1999). В свою очередь, IEEE и ACM[3], начав совместные работы еще в 1993 году с кодекса этики и профессиональной практики в данной области (ACM/IEEE-CS Software Engineering Code of Ethics and Professional Practice), к 2004 году сформулировали два ключевых документа по основам программной инженерии – Software Engineering:

– ***Guide to the Software Engineering Body of Knowledge*** (SWEBOK), IEEE 2004 Version – Руководство к своду знаний по программной инженерии;

[1] IEEE – Computer Society of the Institute for Electrical and Electronic Engineers, IEEE Computer Society – IEEE-CS (Компьютерное общество) или просто IEEE (http://www.ieee.org).
[2] ISO – International Organization for Standardization (https://www.iso.org). IEC – International Electrotechnical Commission; JTC 1 – Joint Technical Committee 1, Information technology (https://www.iso.org/committee/45020.html).
[3] ACM – Association of Computer Machinery (https://www.acm.org).

– Software Engineering 2004, ***Curriculum Guidelines for Undergraduate Degree Programs in Software Engineering*** – учебный план для преподавания программной инженерии в высших учебных заведениях.

Программная инженерия – это наука о систематизированных, регламентированных и квантифицируемых методах решения задач разработки, эксплуатации, сопровождения и утилизации программного обеспечения. При этом как сами бизнес-процессы, так и сопровождающее их программное обеспечение должны отвечать четко заданным техническим экономическим и социальным требованиям. Очевидно, что создание высококачественного программного продукта – очень трудоемкий процесс. Здесь должны быть задействованы необходимые для разработки процессы, инструментарии, технологии и человеческие ресурсы. В связи с этим возникла острая необходимость в специалистах, владеющих необходимыми компетенциями в новых технологиях и методах управления сложными проектами разработки больших программных комплексов.

2. ОСНОВЫ КОМПЬЮТЕРНОЙ ТЕХНИКИ

2.1. Традиционная архитектура вычислительных систем

По назначению компьютер – это универсальная машина для работы с информацией. В основе этой работы лежат три базовых информационных процесса: хранение, обработка и передача (прием – отправка) информации.

Базовая структура вычислительной машины представлена на рис. 2.1.

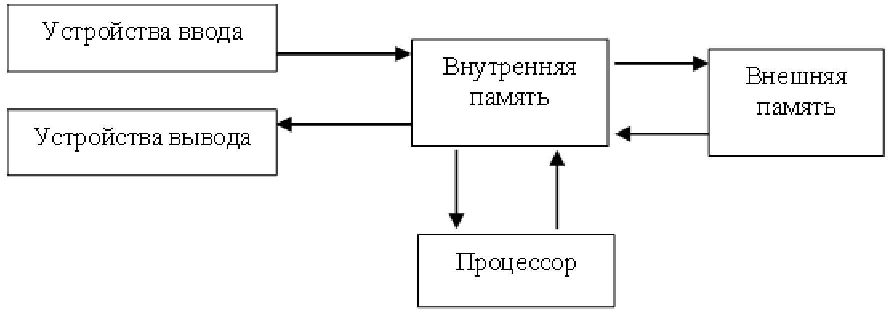


Рис. 2.1. Состав и структура вычислительной машины

В 1946 году Дж. фон Нейман, Г. Голдстайн и А. Беркс в своей общей статье изложили новые принципы построения и функционирования электронных вычислительных машин. На основе этих принципов разрабатывались первые поколения компьютеров. В более поздних поколениях произошли конфигурации, но эти принципы актуальны и сейчас.

Принципы фон Неймана:

1) использование двоичной системы счисления;

2) программное управление вычислительной машиной;

3) память компьютера используется не только для хранения данных, но и приложений;

4) ячейки памяти имеют последовательно пронумерованные адреса;

5) возможность условного перехода в процессе выполнения программы.

Машина фон Неймана состоит из устройства памяти, арифметико-логического устройства (АЛУ), устройств ввода и вывода, устройства управления (управляет всеми частями компьютера).

Структура современных компьютеров несколько отличается от классической архитектуры и отвечает принципам открытой архитектуры.

Особенности открытой архитектуры:

– арифметико-логическое устройство и устройство управления соединены в одно устройство – микропроцессор (МП, или центральный процессор);

– применяются специализированные устройства – контроллеры;

– вместо отдельных линий связи между устройствами используется системная магистраль с соответствующими устройствами сопряжения.

Такую архитектуру первой предложила компания IBM, поэтому компьютеры, имеющие такую структуру, называют IBM-совместимые. Общая схема открытой архитектуры компьютера представлена на рис. 2.2.

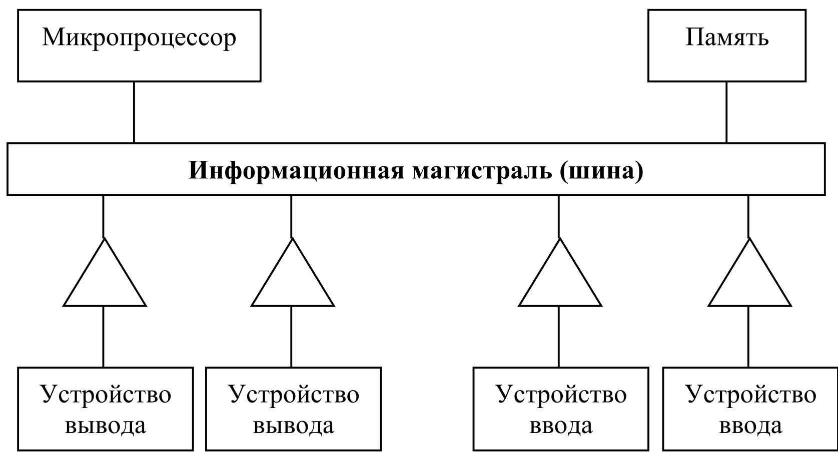


Рис. 2.2. Архитектура персональных компьютеров

Сегодня чаще всего шиной компьютера выступает материнская плата, на которую устанавливаются все остальные устройства (через соответствующие слоты). Важнейшим устройством следует считать микропроцессор (ЦПУ, от англ. CPU – Central Processor Unit). Разрядность процессора – это размер «порции» информации, которую процессор может обработать за операцию (одной командой).

2.2. Системы счисления

Система счисления (ССЧ) – совокупность способов представления и записи чисел; совокупность приемов обозначения чисел с помощью алфавита (множества символов) и синтаксиса (множества правил формулировки однозначности записи и понимания чисел).

Запись числа в некой ССЧ называют кодом числа.

ССЧ делят на две группы:

1. Непозиционная система счисления – ССЧ, в которой значение каждого символа в произвольном месте последовательности, означающее запись числа, не изменяется (унарная/единичная, вавилонская, славянская). В непозиционной системе каждый знак в записи числа независимо от местоположения имеет одно и то же значение. Позиция символов в такой записи не имеет значения.

2. Позиционная система счисления – ССЧ, в которой значение каждого символа строго зависит от места в последовательности записи числа. Алфавит позиционных ССЧ ограничен. Для позиционных систем счисления характерны наглядность изображения чисел и относительная простота выполнения операций. Отдельную позицию в изображении числа называют разрядом числа. Соответственно количество символов в записи числа называется разрядностью. Основой системы может быть произвольное натуральное число, большее единицы, которое задает количество символов, используемых для представления чисел в этой ССЧ.

Десятичная система счисления имеет основу 10, т. е. в ее алфавит входят десять цифр от 0 до 9.

Двоичная система счисления имеет основание 2, в ее алфавит входят только цифры 0 и 1.

Шестнадцатеричная система счисления имеет основу 16, в ее алфавит входят десять цифр от 0 до 9 и шесть букв латинского алфавита от A до F.

Эти системы счисления наиболее распространены при использовании в компьютере (табл. 2.1).

Таблица 2.1

Представление чисел в различных системах счисления

2 ССЧ	**8 ССЧ**	**10 ССЧ**	**16 ССЧ**
1	**2**	**3**	**4**
0	**0**	**0**	**0**
1	**1**	**1**	**1**
0010	**2**	**2**	**2**
0011	**3**	**3**	**3**
0100	**4**	**4**	**4**
0101	**5**	**5**	**5**
0110	**6**	**6**	**6**
0111	**7**	**7**	**7**

Окончание табл. 2.1

1	**2**	**3**	**4**
1000	10	**8**	**8**
1001	11	**9**	**9**
1010	12	10	**A**
1011	13	11	**B**
1100	14	12	**C**
1101	15	13	**D**
1110	16	14	**E**
1111	17	15	**F**
10000	20	16	10

Обычно числа, записанные в одной ССЧ, могут быть переведены и записаны в другие ССЧ. Для этого есть определенные методики.

Правила и примеры перевода чисел из одной системы счисления в другую приведены в прил. 1.

Теоретически доказано, что наиболее экономичными и быстродействующими были бы компьютеры, в которых была бы использована система счисления с основой 2,718281828, что является основой натуральных логарифмов. Но технически они были бы очень сложны. Реализация близкой к ней троичной системы тоже не упрощает конструкцию. Следовательно, оптимальной с экономической точки зрения остается двоичная система, которая и применяется в современных компьютерах.

Однако человеку удобно вводить информацию и получать результаты вычислений в виде десятичной системы счисления. Для этого используются так называемые двоично-десятичные коды. В них один десятичный разряд представляется четырьмя двоичными разрядами (тетрадой). С помощью четырех бит можно закодировать шестнадцать разных символов (цифр). Существует много различных систем кодирования, но наиболее широко используется код прямого замещения – код 8-4-2-1 (табл. 2.2).

Таблица 2.2

Двоично-десятичные коды прямого замещения

Двоично-десятичный код				**Десятичная цифра**
1	**2**	**3**	**4**	**5**
8	**4**	**2**	**1**	–
0	0	0	0	0
0	0	0	1	1

Окончание табл. 2.2

1	**2**	**3**	**4**	**5**
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9

Десятичные цифры от 0 до 9 представляются соответствующими двоичными числами от 0000 до 1001. Для сохранения в памяти компьютера каждого такого числа требуется 1 полубайт, равный 4 битам.

Код в двоично-десятичной системе счисления может быть представлен в памяти компьютера в двух форматах:

1. *Упакованный формат.* В этом формате каждый байт содержит две десятичные цифры (табл. 2.3). Десятичная цифра представляет собой двоичное значение в диапазоне от 0 до 9 в размере четырех битов. При этом код старшей цифры числа занимает старшие четыре бита. Знак числа кодируется в крайнем правом полубайте числа.

Таблица 2.3

Структура упакованного числа

Байт		Байт		…	Байт	
Цифра	Цифра	Цифра	Цифра	…	Цифра	Знак

Упакованный формат обычно используется в компьютере при выполнении операций сложения и вычитания двоично-десятичных чисел.

2. *Неупакованный формат*. В этом формате каждый байт содержит одну десятичную цифру в четырех младших байтах (табл. 2.4). Старшие четыре бита являются символом числа. Для кодирования знака можно использовать шесть двоичных комбинаций, которые не используются для представления десятичных цифр, – это коды 1010–1111 (A–F в шестнадцатеричной системе). Обычно для кодирования знака плюс применяется код 1101, а для знака минус – 1101.

Таблица 2.4

Структура неупакованного числа

Байт		Байт		Байт		Байт	
Знак	Цифра	Знак	Цифра	Знак	Цифра	Знак	Цифра

Неупакованный формат используется при вводе-выводе информации в компьютер, а также при выполнении операций умножения и деления двоично-десятичных чисел. Двоично-десятичная система используется там, где есть необходимость использования процедуры десятичного ввода-вывода (например, электронные часы, калькуляторы и т. п.). В подобных устройствах не всегда целесообразно применять код перевода двоичных чисел в десятичные и наоборот из-за малого объема памяти. В некоторых типах компьютеров в арифметико-логических устройствах имеются специальные блоки десятичной арифметики, выполняющие операции над числами, представленными в двоично-десятичном коде. Это позволяет существенно повысить производительность компьютера.

2.3. Общие принципы кодирования чисел

Для представления двоичных чисел в компьютерной арифметике используются прямые, обратные и дополнительные коды. Они предназначены для записи положительных и отрицательных чисел и проведения над ними арифметических операций.

Прямой код – один из способов представления двоичных чисел с фиксированной запятой. Прямой код используется в двух вариантах:

1) для записи отрицательных чисел;

2) для записи как отрицательных, так и положительных чисел.

В первом варианте (для восьмибитного двоичного числа) можно записать максимальное число 255 (всего чисел 256 – от 0 до 255) (табл. 2.5).

Таблица 2.5

Представление чисел в прямом коде

Десятичное число	**Двоичное восьмибитное число в прямом коде**
1	**2**
0	00000000

Окончание табл. 2.5

1	2
10	00001010
100	01100100
255	11111111

Во втором варианте есть возможность записать отрицательное число.

В этом случае старший бит объявляется знаковым разрядом (знаковым битом) (табл. 2.6).

Таблица 2.6

Представление знакового целого числа

Разряды восьмибитного двоичного числа							
7	6	5	4	3	2	1	0
Знаковый разряд	Значение числа						

Если знаковый разряд равен 0, то число положительное; если знаковый разряд равен 1, то число отрицательное (табл. 2.7).

Таблица 2.7

Представление чисел со знаком в прямом коде

Десятичное число	**Двоичное восьмибитное число со знаком в прямом коде**
127	**0**1111111
100	**0**1100100
0	**0**0000000
–0	**1**0000000
–100	**1**1100100
–127	**1**1111111

В таком случае диапазон десятичных чисел, который можно записать в прямом коде, находится от –127 до +127.

При использовании чисел со знаком у прямого кода есть два недостатка:

1. В прямом коде имеется два варианта записи числа 0 (например, 00000000 и 10000000 в восьмиразрядном представлении). Второе представление называется «отрицательный нуль».

2. Использование прямого кода для представления отрицательных значений в памяти компьютера предусматривает либо выполнение арифметических

операций, либо перевод числа в другое представление (например, в дополнительный код) перед выполнением операций и перевод результатов обратно в прямой код (являющийся неэффективным).

Прямой код используется для хранения чисел в памяти компьютера, а также при выполнении операций умножения и деления, но он неудобен для выполнения вычислений, поскольку сложение и вычитание положительных и отрицательных чисел происходит по-разному, а поэтому нужно анализировать знаковые разряды операндов. По этой причине прямой код почти не используется при реализации арифметико-логического устройства арифметических операций над целыми числами. Вместо этого формата широкое распространение получили форматы представления чисел в обратном и дополнительном кодах.

Обратный код положительного числа выглядит так же, как и прямой код положительного числа с использованием знакового разряда. Обратный код отрицательного числа образуется путем инвертирования всех битов (1 заменяется на 0, а 0 заменяется на 1) положительного числа в прямом коде. Для преобразования отрицательного числа, записанного в обратном коде, в положительное достаточно также его инвертировать.

Диапазон десятичных чисел, который можно записать в обратном коде, составляет от –127 до +127 (табл. 2.8).

Таблица 2.8

Представление чисел в обратном коде

Положительное десятичное число	**Положительное двоичное число в обратном коде**	**Отрицательное десятичное число**	**Отрицательное двоичное число в обратном коде**
0	00000000	–0	11111111
10	00001010	–10	11110101
100	01100100	–100	10011011
127	01111111	–127	10000000

Обратный код ранее использовался в механических калькуляторах и компьютерах. Его перестали использовать из-за его недостатков, а именно из-за того, что арифметические операции проводятся в два этапа и, как и в прямом коде, есть два представления нуля – положительное и отрицательное.

На сегодняшний день в современных компьютерах применяется дополнительный код. В дополняемом коде (как в прямом и обратном) старший разряд отводится для представления знака числа (знаковый бит). Запись положительных чисел происходит так же, как и в прямом и обратном кодах.

Дополнительный код отрицательного числа можно получить двумя способами:

1) инвертировать значение отрицательного числа, записанного в прямом коде, и добавить к нему 1;

2) вычесть модуль числа из нуля.

Рассмотрим пример получения дополнительного кода числа.

Дано десятичное число (–10).

Первый способ.

Получим вид десятичного числа в двоичном прямом коде:

$10_{(10)} = 00001010_{(2)} \Rightarrow (-10)_{(10)} = 100010110_{(2)}$.

Произведем инвертированное значение: 10001010 => 11110101.

К инвертированному значению прибавим 1: 11110101 + 1 = 11110110.

В результате получим десятичное число (–10) в дополнительном коде 11110110.

Второй способ.

Получим модуль числа (–10) в двоичной системе: 10 = 00001010.

Произведем вычитание из нуля: 0 – 00001010 = 11110110.

В результате получим десятичное число (–10) в дополнительном коде 11110110.

Диапазон десятичных чисел, которые можно записать в дополнительном коде, составляет от –128 до +127. Дополнительный код используется для сохранения чисел в памяти компьютера и упрощения выполнения арифметических операций над ними. В прямом и обратном кодах существует два вида представления нуля, а именно 00000000 (0) и 10000000 (−0) в прямом коде, 00000000 (0) и 11111111 (−0) в обратном. Отрицательный нуль можно получить в результате

умножения или деления отрицательного числа на 0. Его наличие усложняет операцию сравнения. В дополнительном коде отрицательного нуля не существует, а при переводе его из прямого или обратного кодов он будет равен 00000000.

При переводе числа 10000000 (–128) из дополнительного кода в прямой число будет иметь вид 110000000 в двоичной системе счисления и занимать девять битов. Но поскольку в АЛУ для отрицательных чисел используется дополнительный код, то при представлении в ней этого числа в прямом коде в восьмибитном переменном старший разряд будет потерян из-за переполнения. В результате будет получено число 10000000, что в прямом коде равно отрицательному нулю (–0) (табл. 2.9).

Таблица 2.9

Сравнительная таблица представления восьмибитных чисел в разных кодах

Число	Прямой код	Обратный код	Дополнительный код
127	01111111	01111111	01111111
1	00000001	00000001	00000001
0	00000000	00000000	00000000
–0	10000000	11111111	–
–1	10000001	11111110	11111111
–2	10000010	11111101	11111110
–3	10000011	11111100	11111101
–4	10000100	11111011	11111100
–5	10000101	11111010	11111011
–6	10000110	11111001	11111010
–7	10000111	11111000	11111001
–8	10001000	11110111	11111000
–9	10001001	11110110	11110111
–10	10001010	11110101	11110110
–11	10001011	11110100	11110101
–127	11111111	10000000	10000001
–128	–	–	10000000

2.4. Операции манипуляции данными в компьютере

К операциям для манипуляции данными (числами) в компьютере относятся битовые операции. Побитовые операции – операции, выполняемые над цепочками битов. Все побитовые операторы производят операции непосредственно на битах числа, поэтому будем рассматривать операции в двоичной системе счисления.

Выделяют два типа битовых операций: логические и сдвиги.

Логические битовые операторы И (обозначается как AND, &), ИЛИ (обозначается как OR, |), НЕТ (обозначается как NOT, ~) и исключительная дизъюнкция (обозначается как XOR, ⊕) используют те же таблицы истинности, что и их логические эквиваленты. Они могут быть описаны с помощью таблиц соответствия (рис. 2.3).

В побитовых операциях значение бита, равное 1, рассматривается как логическая истина, а 0 – как ложь.

X	Y	X AND Y	X OR Y	X XOR Y	NOT X
0	0	0	0	0	1
0	1	0	1	1	1
1	0	0	1	1	0
1	1	1	1	0	0

Рис. 2.3. Таблица соответствия логических операторов

Исключительная дизъюнкция – это логическая и битовая операция, принимающая значение «истины» тогда и только тогда, когда только один из аргументов истинен. В математических нотациях операция исключительной дизъюнкции обозначается символом ⊕. На языках программирования (например, C, C++, C#, JavaScript, Python, Java, Ruby, PHP) операция XOR обозначается символом ^.

С помощью исключительной дизъюнкции можно обменять значение двух переменных (одинаковых типов данных) без использования дополнительных переменных (соответственно без дополнительной памяти).

Еще с помощью операции ^ можно шифровать текст. На практике исключительная дизъюнкция применяется для хеширования или шифрования данных, используя свойство (a XOR k) XOR k = a, где k – это ключ.

Также с помощью исключительной дизъюнкции создают надежные хранилища данных RAID 5. Кроме того, ее применяют в микроэлектронике для создания управляющих инверторов, смесителей сигналов и для формирования коротких импульсов.

В табл. 2.10 и 2.11 показана работа исключительной дизъюнкции для бинарного и тернарного сложения.

Таблица 2.10

Исключительная дизъюнкция для бинарного сложения

α	β	$\alpha \oplus \beta$
0	0	0
0	1	1
1	0	1
1	1	0

Таблица 2.11

Исключительная дизъюнкция для тернарного сложения

X	Y	Z	$\oplus(X, Y, Z)$
0	0	0	0
1	0	0	1
0	1	0	1
1	1	0	0
0	0	1	1
1	0	1	0
0	1	1	0
1	1	1	1

Рассмотрим примеры использования операции исключительной дизъюнкции (XOR).

Пример использования операции XOR на языке C++ для обмена значений двух переменных без использования третьей:

```
int x = 5, y = 7;
x = x^y;                                        // x == 2
y = x^y;                                        // y == 5
x = x^y;                                        // x == 7
std::cout << "x = " << x << ", y = " << y;      // x = 7, y = 5
```

Пример шифрования текста с использованием исключительной дизъюнкции на языке JavaScript:

```
function crypt(str, key) {
      var newstr = '';
      for(let i=0; i < str.length; i++) {
            let char = str.charCodeAt(i) ^ key;
            newstr += String.fromCharCode(char);
      }  return newstr;
}
var text = 'Текст для шифрования',
key = 376000,
hash = crypt(text, key);
console.log(hash);
// текст в зашифрованном виде: 룢룵룺뢁뢂볠룴룻뢏볠뢈룸뢄뢀룾룲룰룽룸뢏
text = crypt(hash, key);
console.log(text);
// дешифрованный текст «Текст для шифрования»
```

Второй тип побитовых операций, которые ускоряют обработку данных, – это две операции сдвига чисел:

1) битовый сдвиг влево (оператор <<);
2) битовый сдвиг вправо (оператор >>).

Побитовый сдвиг влево перемещает биты числа на N количество битов налево, начиная с младшего бита. Пустые места после сдвига заполняются нулями. Эта операция равносильна умножению числа на 2^N. Происходит это следующим образом (табл. 2.12).

Таблица 2.12

Побитовый сдвиг налево

A	1	0	1	1	0	1	0	0
A << 2	1	1	0	1	0	0	0	0

Битовый сдвиг вправо сдвигает биты числа направо. Если число положительное, то пустые места заполняются нулями. Если же оно отрицательное, то все пустые места слева заполняются единицами. Это делается для сохранения знака в соответствии с дополняемым кодом. Смещение числа справа на N позиций по своей сути делит это число на 2^N.

Следует отметить, что использование сдвигов вместо команд возведения до степени N обеспечивает неплохой прирост производительности кода.

Пример применения операций сдвига на языке **C++**:

```
int a = 5;           // в двоичной системе: 0000000000000101
int b = a << 3;      // в двоичной системе: 0000000000101000,
                     // или 40 в десятичной
int c = b >> 3;      // в двоичной системе: 0000000000000101,
                     // или 5 в десятичной
// Следует учесть, что возможны переполнения на границе типа
int a = INT_MAX;     // 1111111111111111111 = 2147483647
int b = a << 1;      // 1111111111111111110 = -2
```

Пример применения операций сдвига на языке **C#**:

```
uint shiftLeft = 15;                 // 00001111
uint doubled = shiftLeft << 1;       // результат = 00011110 = 30
uint shiftFour = shiftLeft << 4;     // результат = 11110000 = 240

uint shiftRight = 240;                // 11110000
uint halved = shiftRight >> 1;        // результат = 01111000 = 120
uint shiftFour = shiftRight >> 4;     // результат = 00001111 = 15

// Возможны переполнения на границе типа
int maxValue = int.MaxValue;
// переменная maxValue имеет значение 1111111111111111111111111111111,
// что равно 2147483647 в десятичной
Console.WriteLine(maxValue << 1);
// команда выведет 1111111111111111111111111111110,
//  что равно (-2) в десятичной
```

Пример применения операций сдвига на языке **Python**:

```
a = 43          # 0b101011
b = a >> 1      # 0b10101 == 21
c = a >> 2      # 0b1010 == 10
d = a >> 3      # 0b101 == 5
e = a >> 5      # 0b1 == 1
```

```
#
a = 5           # 0b101
b = a << 1      # 0b1010 == 10
c = a << 2      # 0b10100 == 20
d = 2 << 3      # 0b101000 == 40
```

В Python нет фиксированного размера типа. Поэтому следует отметить, что при переполнении переменной типа int она автоматически преобразуется в тип long. Со своей стороны тип long не имеет четкого определенного ограничения. Объем доступного адресного пространства составляет практический предел.

Пример применения операций сдвига на языке **JavaScript**:

```
let result = 9 << 2;
console.log(result);              // выводит значение 36
let result1 = 9 >> 2;
let result2 = -9 >> 2;
console.log (result1);            // выводит значение 2
console.log (result2);            // выводит значение –3
        // Возможны переполнения на границе типа
let overflow = Number.MAX_VALUE;   // 1.7976931348623157e+308
console.log(overflow << 1)        // выводит значение 0
```

2.5. Задания для самоконтроля

Задания для отработки навыков работы с системами счисления.

1. Выполните перевод чисел $\mathbf{10001011_{(2)}}$ и $\mathbf{213_{(8)}}$ к десятичной ССЧ.

2. Выполните перевод чисел $\mathbf{95_{(10)}}$ в несколько систем счисления: в двоичную ССЧ, в восьмеричную ССЧ и в ССЧ с основанием 16.

3. Выполните перевод правильной десятичной дроби $\mathbf{0.36_{(10)}}$ к шестнадцатеричной ССЧ.

Задания для отработки навыков работы с битовыми операциями.

4. Выполните сложение, умножение, вычитание и деление чисел А1 = 1101010001 и А2 = 1010 в двоичной ССЧ: А1 + А2, А1 * А2, А1 А2, А1/А2.

5. Выполните сложение и умножение чисел А3 = 345.2 и А4 = 7.3 в восьмеричной ССЧ. Для выполнения задания воспользуйтесь таблицами сложения и умножения чисел (рис. П.2.1).

6. Выполните сложение и умножение чисел А5 = 41А и А6 = ABCDE в ССЧ с основанием 16. Для выполнения задания воспользуйтесь таблицами сложения и умножения чисел (рис. П.2.2 и П.2.3).

☻ **Ответы к заданиям для самоконтроля:**

1. $10001011_{(2)} = 139_{(10)}$; $213_{(8)} = 139_{(10)}$.

2. $95_{(10)} = 1000000_{(2)} = 137_{(8)} = 5F_{(16)}$. Шаги выполнения переводов приведены на рис. П.1.4.

3. $0.36_{(10)} = 0.5C28_{(16)}$. Шаги выполнения переводов приведены на рис. П.1.5.

4. $A1 + A2 = 0000001101011011_{(2)}$; $A1 * A2 = 0010000100101010_{(2)}$; $A1 – A2 = 0000001101000111_{(2)}$; $A1 / A2 = 1010100.111_{(2)}$.

5. $A3 + A4 = 354.5_{(8)}$; $A3 * A4 = 3232.56_{(8)}$.

6. $A5 * A6 = 2C0AA68C_{(16)}$.

3. ОСНОВЫ КОНСТРУИРОВАНИЯ ПРОГРАММНЫХ ПРОДУКТОВ

3.1. Программное обеспечение как продукт

Возможности компьютера как технической основы системы обработки данных связаны с используемым *программным обеспечением*.

Программное обеспечение (software) – совокупность *программ* обработки данных и необходимых для их эксплуатации документов.

Программа (program) – упорядоченная последовательность команд (инструкций) компьютера для решения *задачи*.

Задача (problem, task) – проблема, подлежащая решению.

Все программы по характеру использования и категориям пользователей можно разделить на два класса: *утилитарные* программы и *программные продукты* (изделия, продукция).

Утилитарные программы («программы для себя») предназначены для удовлетворения нужд их разработчиков. Чаще всего утилитарные программы выполняют роль сервиса в технологии обработки данных либо являются программами решения функциональных задач, не предназначенных для широкого распространения.

Программный продукт – комплекс программного обеспечения для решения определенной проблемы (задачи) массового спроса, подготовленный к реализации как любой вид промышленной продукции.

Программный продукт должен быть соответствующим образом подготовлен к эксплуатации, иметь необходимую техническую документацию, предоставлять сервис и гарантию надежной работы программы, иметь товарный знак изготовителя, а также желательно наличие кода государственной регистрации. Только при таких условиях созданный программный комплекс может быть назван *программным продуктом.*

В свою очередь, программные продукты могут создаваться либо в виде индивидуальной разработки под заказ (уникальный продукт), либо как разработка для массового распространения среди пользователей (продукция).

При индивидуальной разработке создается оригинальный программный продукт, учитывающий специфику обработки данных исключительно для конкретного заказчика (пользователя).

При разработке для массового распространения разработчик, с одной стороны, должен обеспечить универсальность выполняемых функций обработки данных, а с другой – гибкость и адаптивность программного продукта к условиям конкретного применения. Отличительной особенностью программных продуктов считается их системность, которая понимается как сочетание функциональной полноты и законченность реализуемых функций обработки данных.

Программный продукт разрабатывается на основе промышленной технологии выполнения проектных работ с применением современных инструментальных средств программирования.

На создание программных продуктов затрачиваются значительные ресурсы (временные, материальные, финансовые) и требуется высокая квалификация разработчиков.

В условиях существования рынка важными характеристиками программных продуктов являются:

– стоимость;
– количество продаж;
– время нахождения на рынке (длительность продаж);
– известность фирмы-разработчика и программы;
– наличие программных продуктов аналогичного назначения.

Программные продукты массового распространения продаются по ценам, которые учитывают спрос и конъюнктуру рынка (наличие и цены программ-конкурентов). Соответственно, важным аспектом в продвижении программной продукции на рынке является наличие конкурентов и репутационный фонд (бренд разработчика и бренд продукта).

Спецификой программных продуктов (в отличие от большинства промышленных изделий) является также и то, что их эксплуатация выполняется на

весьма широкой правовой основе – существуют различные лицензионные соглашения между разработчиком и пользователями с соблюдением авторских прав разработчиков программных продуктов.

В настоящее время существуют несколько вариантов легального распространения программных продуктов: бесплатные программы (freeware), некоммерческие условно-бесплатные программы (shareware), частично или временно бесплатные программы (trial), встроенные или сопутствующие программы (лицензия Original Equipment Manufacturer, OEM) и др.

3.2. Классификация программных продуктов

Программные продукты можно классифицировать по различным признакам. Рассмотрим классификацию с точки зрения сферы использования (область применения) программных продуктов:

- аппаратная часть компьютерных систем и сетей;
- функциональные задачи различных предметных областей;
- технологии конструирования программ.

Для поддержки информационной технологии в каждой из этих областей определены соответствующие типы программных продуктов.

Выделяют три типа программных продуктов:

- системное программное обеспечение;
- прикладное программное обеспечение;
- инструментальное программное обеспечение.

Системное программное обеспечение направлено на решение следующих задач:

- создание операционной среды функционирования других программ;
- обеспечение надежной и эффективной работы самого компьютера и вычислительной сети;
- проведение диагностики и профилактики аппаратуры компьютера и вычислительных сетей;

– выполнение вспомогательных технологических процессов (копирование, архивирование, восстановление файлов и т. д.).

Данный класс программных продуктов тесно связан с типом компьютера и является его неотъемлемой частью. Такие продукты в основном ориентированы на квалифицированных пользователей – профессионалов в компьютерной области (например, системного программиста, администратора сети, прикладного программиста и пр.). Однако знание базовой технологии работы с этим классом программных продуктов требуется и конечным пользователям персонального компьютера, которые самостоятельно не только работают со своими программами, но и выполняют обслуживание компьютера, программ и данных. Программные продукты данного класса носят общий характер применения, независимо от специфики предметной области. К ним предъявляются высокие требования по надежности и технологичности работы, удобству и эффективности использования.

Системное программное обеспечение в свою очередь имеет подклассы (подвиды):

– базовое программное обеспечение (Base Software) – минимальный набор программных средств, обеспечивающих работу компьютера. К таким продуктам относятся операционные системы или операционные оболочки;

– сервисное программное обеспечение (утилиты);

– программные средства, которые расширяют возможности базового программного обеспечения и организуют более удобную среду для работы пользователя.

Среди утилит можно выделить диагностические комплексы, антивирусы, архиваторы, комплексы обслуживания оборудования (дисков, памяти, сети и пр.).

Прикладное программное обеспечение, или пакет прикладных программ (Application Program Package), – комплекс взаимосвязанных программных средств, которые решают задачи определенного класса в конкретной предметной области.

Пакеты прикладных программ служат программным инструментарием для решения функциональных задач и являются самым многочисленным классом программных продуктов. В данный класс входят программные продукты, выполняющие обработку информации различных предметных областей. Данный класс программных продуктов может быть весьма специфичным для отдельных предметных областей.

Инструментальное программное обеспечение, или инструментарий технологии программирования, – совокупность программных средств, обеспечивающих процессы конструирования программных продуктов и их внедрения. Инструментарий технологии программирования обеспечивает процесс разработки программ и включает специализированные программные продукты, которые являются инструментальными средствами разработчика. Программные продукты данного класса поддерживают все технологические этапы процесса проектирования, программирования (кодирования), отладки и тестирования создаваемых программ. Пользователями технологии программирования являются системные и прикладные программисты.

3.3. Этапы создания программного продукта

Современная индустриальная технология создания программ включает в себя комплекс мероприятий, руководящих документов и автоматизированных средств, предназначенных для системного анализа, разработки, отладки, документирования, управления работой специалистов.

В соответствии с обычным значением слова «технология» под технологией программирования (Programming Technology) будем понимать совокупность производственных процессов, приводящую к созданию требуемого программного средства (ПС), а также описание этой совокупности процессов. Другими словами, технологию программирования мы будем понимать в широком смысле как технологию разработки программных средств, включая в нее все процессы, начиная с момента зарождения идеи этого средства, и, в частности, связанные с созданием необходимой программной документации.

Совокупность процессов по конструированию и сопровождению программных продуктов принято называть его жизненным циклом.

В словаре программной инженерии *IEEE Std 610.12-90 «IEEE Standard Glossary of Software Engineering Terminology»* ***жизненный цикл программного обеспечения*** определяется как период времени, который начинается с момента принятия решения о необходимости создания ПО и заканчивается в момент его полного изъятия из эксплуатации. Этот период принято разделять на **шесть этапов** (пять этапов создания ПО и один этап его сопровождения):

1) анализ требований (планирование);
2) проектирование;
3) разработка (реализация, кодирование);
4) тестирование;
5) внедрение (ввод в эксплуатацию);
6) эксплуатация (сопровождение).

Анализ требований (планирование). Жизненный цикл разработки ПО начинается со стадии анализа, в ходе которого участники процесса обсуждают требования, предъявляемые к конечному продукту. Цель этой стадии – определение детальных требований к системе. Кроме этого, необходимо убедиться в том, что все участники правильно поняли поставленные задачи и то, как именно каждое требование будет реализовано на практике.

Чаще всего в обсуждении принимают участие и специалисты по тестированию, которые уже на стадии разработки требований могут вносить собственные пожелания и при необходимости корректировать процесс.

В зависимости от выбранной модели разработки могут отличаться подходы к определению момента перехода с одной стадии на другую. Например, в каскадной или V-модели стадия анализа требований закрепляется в документе – спецификации требований к программному обеспечению (Software Requirement Specification, SRS), оформление которого должно быть закончено до перехода на следующую стадию.

Таким образом, данный этап предусматривает сбор требований по разрабатываемому программному обеспечению, их систематизацию, документирование, анализ, а также выявление и разрешение противоречий.

Проектирование. На стадии проектирования (стадия дизайна и архитектуры) программисты и системные архитекторы, руководствуясь требованиями, разрабатывают высокоуровневый дизайн системы. Различные технические вопросы, возникающие в процессе проектирования, обсуждаются со всеми заинтересованными сторонами, включая заказчика. Определяются технологии, которые будут использоваться в проекте, загрузки команды, ограничения, временные рамки и бюджет. Согласно уточненным требованиям выбираются наиболее приемлемые проектные решения. Утвержденный дизайн системы определяет перечень разрабатываемых программных компонентов, взаимодействие с третьими сторонами, функциональные характеристики программы, используемые базы данных и многое другое. Дизайн, как правило, закрепляется отдельным документом – дизайн-спецификацией (Design Specification Document, DSD).

На этом этапе для упрощения визуализации процесса проектирования используются так называемые нотации – схематическое выражение характеристик разрабатываемой системы. Основные используемые нотации:

- блок-схемы;
- ER-диаграммы;
- UML-диаграммы;
- макеты, например, прототип сайта, нарисованный в Photoshop.

Разработка и программирование (реализация, кодирование). После того как требования и дизайн продукта утверждены, происходит переход к следующей стадии жизненного цикла – непосредственно разработке. Здесь начинается написание программистами кода программы в соответствии с ранее определенными требованиями. Системные администраторы настраивают программное окружение, frontend-программисты разрабатывают пользовательский интерфейс программы и логику ее взаимодействия с сервером. Кроме того, программисты

пишут unit-тесты для проверки правильности работы кода каждого компонента системы, производят ревью написанного кода, создают сборки продуктов и развертывают готовое ПО в программной среде. Этот цикл повторяется, пока все требования не будут реализованы. Обобщенно программирование предусматривает четыре основные фазы:

1) разработка алгоритмов – фактическое создание логики работы программы;

2) написание исходного кода;

3) компиляция – превращение в машинный код;

4) тестирование и отладка (например, unit-тестирование).

Тестирование. Тестировщики занимаются поиском дефектов в программном обеспечении и сравнивают описанное в требованиях поведение системы с реальной. В ходе тестирования выявляются пропущенные при разработке баги. При обнаружении дефекта тестировщик составляет отчет об ошибке, передаваемый разработчикам. Последние его исправляют, после чего тестирование повторяется, но на этот раз для того, чтобы убедиться, что проблема была исправлена, и само исправление не послужило причиной появления новых дефектов в продукте. Тестирование повторяется до тех пор, пока не будут достигнуты критерии его окончания.

Внедрение (ввод в эксплуатацию) и эксплуатация (сопровождение). Когда программа протестирована и в ней больше не осталось серьезных дефектов, наступает время релиза и передачи ее конечным пользователям. После выпуска новой версии программы в работу присоединяется отдел технической поддержки. Его сотрудники обеспечивают обратную связь с пользователями, их консультирование и поддержку. В случае обнаружения пользователями тех или иных пострелизных багов (ошибок после выпуска) информация о них передается в виде отчетов об ошибках команде разработки, которая в зависимости от серьезности проблемы либо немедленно выпускает исправление (hot-fix), либо от-

кладывает его до следующей версии программы. Кроме того, команда технической поддержки помогает собирать и систематизировать разные метрики – показатели работы программы в реальных условиях.

Отдельно можно выделить этап **документации**, которая сопутствует всем этапам жизненного цикла ПО. Этот этап выделяется достаточно условно, поскольку, как мы видели, те или иные документы создаются на всех стадиях жизненного цикла программы. Однако кроме проектной документации и сопровождающих разработку записей существуют и другие текстовые документы, описывающие, например, функции программы и способы ее использования.

Выделяют четыре уровня документации:

– архитектурная (проектная) – например, дизайн-спецификация. Это документы, описывающие модели, методологии, инструменты и средства разработки, выбранные для данного проекта;

– техническая – вся документация, сопровождающая разработку. Сюда входят разные документы, объясняющие работу системы на уровне отдельных модулей. Предпочтительно пишется в виде комментариев к исходному коду, которые впоследствии структурируются посредством HTML-документов;

– пользовательская – включает справочные и объяснительные материалы, необходимые конечному пользователю для работы с системой. Это, например, Readme и User Guide, раздел справки по программе;

– маркетинговая – включает рекламные материалы, сопровождающие выпуск продукта. Ее цель – в яркой форме представить функциональность и конкурентные преимущества продукта.

Дополнительные сведения про виды и назначение документации по проекту приводятся в практической работе №9 «Разработка эксплуатационной документации».

3.4. Разработка требований к программному продукту

> Самая трудная отдельная задача в разработке программной системы – это точно решить, что разрабатывать. Ни одна другая задача работы над концепциями не является столь трудной, как разработка подробных технических требований, включая все интерфейсы пользователей, машинные интерфейсы и интерфейсы к другим программным системам. Ни одна другая часть работы не наносит такого ущерба готовой системе, если сделана неправильно. Ни одна другая часть не исправляется позднее с большим трудом.
>
> *Фредерик Брукс. Мифический человеко-месяц, или Как создаются программные системы*

Проблемы, которые приходится решать специалистам в процессе создания программного обеспечения, очень сложны. Природа этих проблем не всегда ясна, особенно если разрабатываемый программный продукт относится к инновационной продукции. В частности, трудно четко описать те действия, которые должно выполнять программное обеспечение. Описание функциональных возможностей и ограничений, накладываемых на программный продукт, называется требованиями к этому продукту, а сам процесс формирования, анализа, документирования и проверки этих функциональных возможностей и ограничений – разработкой требований.

Требования подразделяются на пользовательские и системные.

Пользовательские требования – это описание на естественном языке (плюс поясняющие диаграммы) функций, выполняемых продуктом, и ограничений, накладываемых на его применение.

Системные требования – это описание особенностей программного обеспечения (архитектура, требования к параметрам оборудования и т. д.), необходимых для эффективной реализации требований пользователя.

Разработка требований – это процесс, включающий мероприятия, необходимые для создания и утверждения документа, содержащего спецификацию системных требований. Различают четыре основных этапа процесса разработки требований (рис. 3.1):

– анализ технической осуществимости создания продукта;

– формирование и анализ требований;

– специфицирование требований и создание соответствующей документации;

– аттестация этих требований.

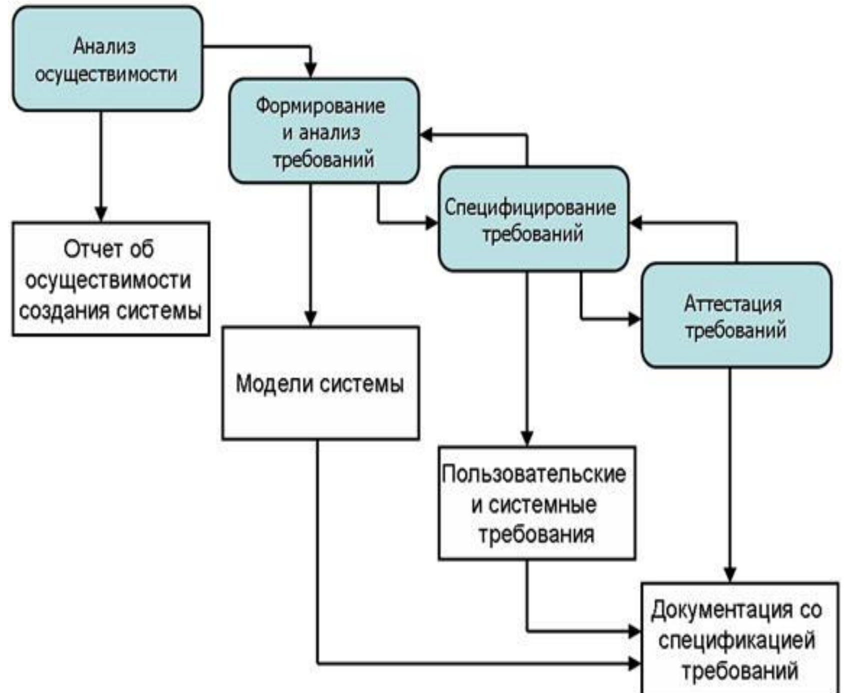


Рис. 3.1. Процесс разработки требований

Обобщенная модель процесса формирования и анализа требований показана на рис. 3.2. Каждая команда использует собственный вариант этой модели, зависящий от особенностей проекта: состава коллектива разработчиков, типа разрабатываемого программного обеспечения, используемых стандартов и пр.

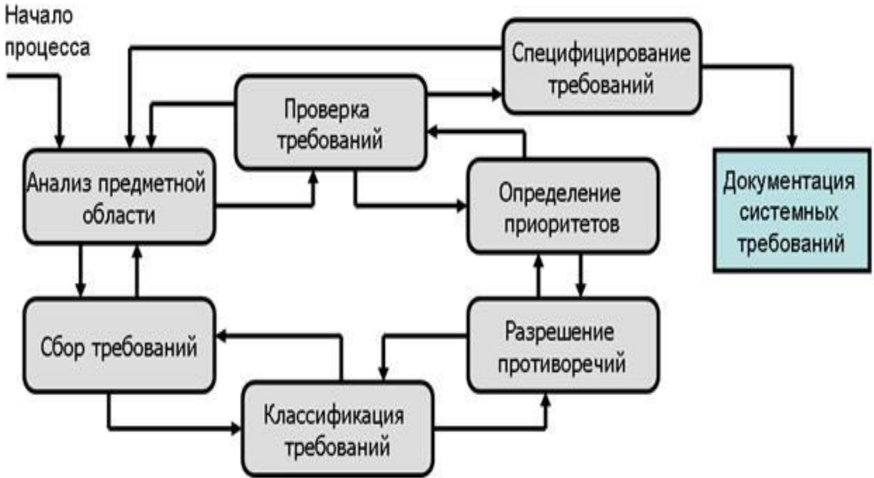


Рис. 3.2. Процесс формирования и анализа требований

Как видно из рис. 3.2, процесс формирования и анализа требований проходит через ряд этапов:

– анализ предметной области (изучение предметной области, в которой будет эксплуатироваться продукт);

– сбор требований (взаимодействие команды разработчиков с лицами, формирующими требования, в ходе которого уточняются детали предметной области);

– классификация требований (преобразование бесформенного «мешка требований» к логически связанной структуре требований);

– разрешение противоречий (выявление и разрешение противоречий различного рода);

– назначение приоритетов (совместно с лицами, формирующими требования, определяются наиболее важные требования и их ранжирование);

– проверка требований (определение полноты, последовательности и непротиворечивости собранных требований).

Проверка требований важна, т. к. ошибки в спецификации требований могут привести к переделке всего продукта и большим затратам, если будут обнаружены во время процесса разработки или после внедрения продукта в эксплуатацию. Стоимость внесения изменений, необходимых для устранения ошибок в

требованиях, намного выше, чем исправление ошибок проектирования или кодирования. Причина в том, что изменение требований обычно влечет за собой значительные изменения во всем продукте, после внесения которых он должен пройти повторное тестирование.

Во время процесса аттестации должны быть выполнены различные типы проверок требований. Существует ряд методов аттестации требований, которые можно использовать совместно или каждый в отдельности.

На основании полученных сведений о предметной области строятся пользовательские требования – описание на естественном языке функций программного продукта и ограничений, накладываемых на его применимость. Пользовательские требования должны описывать внешнее поведение, основные функции и сервисы, предоставляемые программным средством, а также его нефункциональные свойства. Пользовательские требования можно оформить как простым перечислением, так и используя схемы (диаграммы).

Системные требования включают в себя:

– требования к архитектуре системы. Например, число и размещение хранилищ и серверов приложений;

– требования к параметрам оборудования. Например, частота процессоров серверов и клиентов, объем хранилищ, размер оперативной и видеопамяти, пропускная способность канала и т. д.;

– требования к параметрам системы. Например, время отклика на действие пользователя, максимальный размер передаваемого файла, максимальная скорость передачи данных, максимальное число одновременно работающих пользователей и т. д.;

– требования к программному интерфейсу;

– требования к структуре системы, например, масштабируемость, распределенность, модульность, открытость;

– требования по взаимодействию и интеграции с другими системами, например, использование общей базы данных, возможность получения данных из баз данных определенных систем и т. д.

3.5. Системы контроля версий

Система контроля версий (СКВ) – система, которая регистрирует изменения в одном или нескольких файлах с тем, чтобы в дальнейшем была возможность вернуться к предыдущим его версиям этих файлов.

СКВ дает возможность возвращать отдельные файлы к прежнему виду, возвращать в прежнее состояние весь проект, просматривать происходящие со временем изменения, определять, кто последним вносил изменения во внезапно переставший работать модуль, кто и когда внес в код какую-то ошибку, и др.

Основные принципы СКВ:

– хранение нескольких версий одного документа (история версий);

– хранение истории разработки;

– при необходимости возврат к предыдущим версиям документа (отмена изменений);

– определение, кто и когда произвел смену (поиск «виновного»);

– сочетание изменений, произведенных разными разработчиками (синхронизация работы команды);

– реализация альтернативных/экспериментальных вариантов проекта.

Типы систем контроля версий:

– локальные системы контроля версий (одной из наиболее популярных СКВ такого типа является RCS);

– централизованные системы контроля версий (например: CVS, Subversion и Perforce);

– распределенные системы контроля версий (например: Git, Mercurial, Bazaar, Darcs).

Всем известен способ контролировать версии своих документов и проектов путем простого копирования файлов в разные директории (как правило, добавляя текущую дату или номер в название каталога). Такой подход очень распространен, потому что прост, но он и чаще других дает сбои. Очень легко забыть в каком каталоге находишься, перепутать текущий каталог, случайно изменить не тот файл (не той версии), скопировать файлы не туда, куда хотел, и удалить нужные файлы.

Вариант систематизации хранения копий проектов, чтобы решить обозначенную проблему – **локальные системы контроля версий (ЛСКВ)**. Уже давно разработаны ЛСКВ с простой базой данных, в которой хранятся все изменения нужных файлов (рис. 3.3).

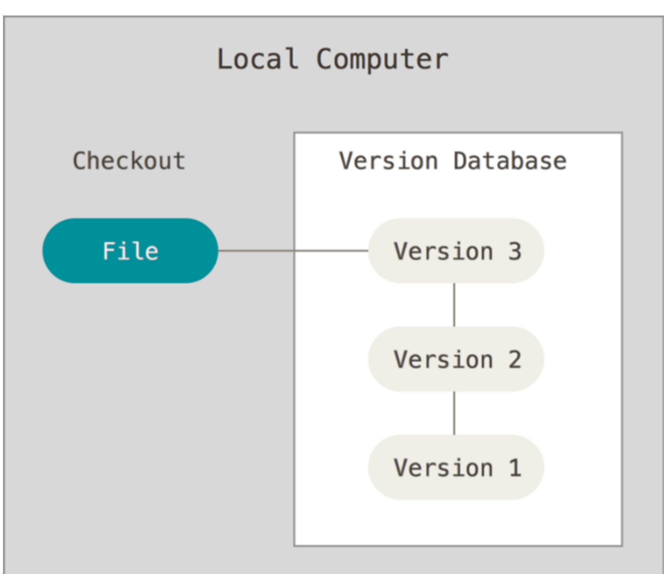


Рис. 3.3. Локальный контроль версий

Следующей критичной проблемой стала необходимость сотрудничать с разработчиками на других компьютерах. Для решения этой проблемы были созданы **централизованные системы контроля версий** (**ЦСКВ**). В таких системах имеется центральный сервер, на котором хранятся все файлы под версионным контролем, и ряд клиентов, получающих копии файлов из него. Многие годы это было стандартом для систем контроля версий (рис. 3.4).

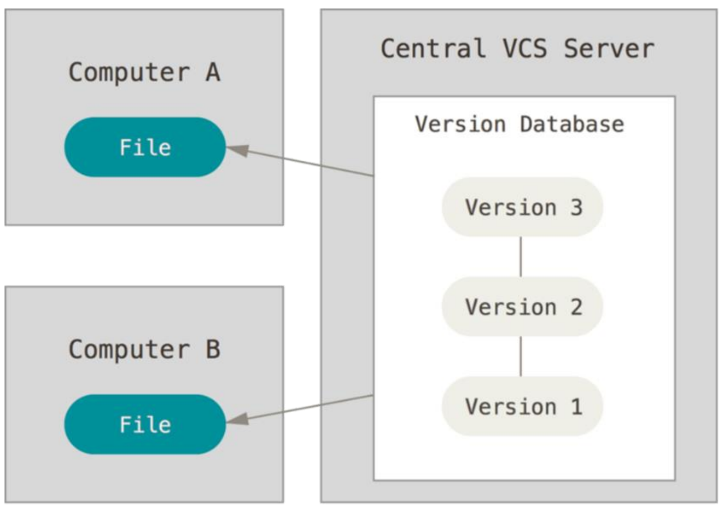


Рис. 3.4. Централизованный контроль версий

Такой подход имеет множество преимуществ, особенно над локальными СКВ. К примеру, все знают, кто и чем занимается в проекте. Администраторы имеют четкий контроль над тем, кто и что может делать, и, конечно, администрировать ЦСКВ гораздо легче, чем локальные базы на каждом клиенте. Однако в таком подходе есть несколько серьезных недостатков. Наиболее очевидный – централизованный сервер является уязвимым местом всей системы. Если сервер выключается на час, то в течение часа разработчики не могут взаимодействовать, и никто не может сохранить новую версию своей работы. Если же поврежден диск с центральной базой данных и нет резервной копии, вы теряете абсолютно все – всю историю проекта, разве что за исключением нескольких рабочих версий, сохраненных на рабочих машинах пользователей. Локальные системы контроля версий подвержены той же проблеме: если вся история проекта сохраняется в одном месте, то есть риск потерять все.

Распределенные системы контроля версий – современные возможности по работе в команде над проектами. В таких системах клиенты не просто скачивают последние версии файлов, а полностью копируют весь репозиторий. Итак, в случае, когда «умирает» сервер, через который шла работа, любой репозиторий клиента может быть скопирован обратно на сервер, чтобы восстановить базу данных. Каждый раз, когда клиент забирает свежую версию файлов, он создает полную копию всех данных (рис. 3.5).

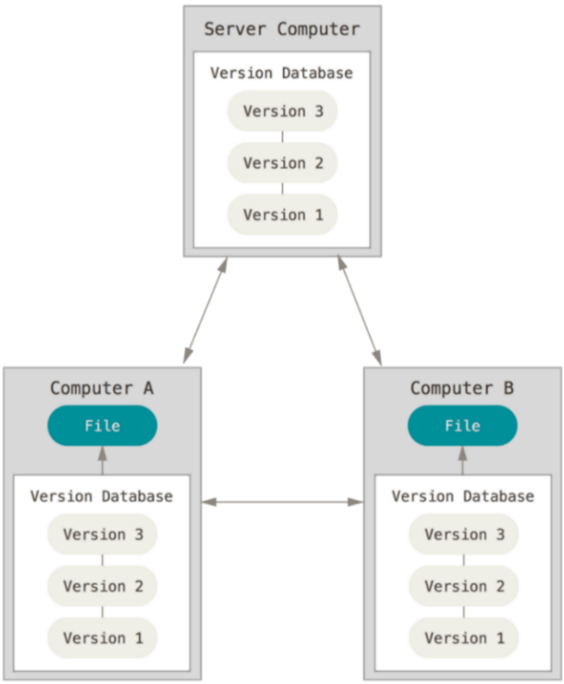


Рис. 3.5. Распределенный контроль версий

Кроме того, в большей части этих систем можно работать с несколькими удаленными репозиториями, а значит, можно одновременно работать по-разному с разными группами людей в рамках одного проекта. В частности, в одном проекте можно одновременно вести несколько типов рабочих процессов, что невозможно в централизованных системах.

Основные преимущества распределенных систем – их гибкость и значительно бóльшая (по сравнению с централизованными системами) автономия отдельного рабочего места.

Каждая система управления версиями имеет свои специфические особенности в наборе команд, порядке работы пользователей и администрировании. Тем не менее общий порядок работы для большинства СКВ стереотипен. Предполагается, что проект существует и размещен в хранилище, к которому разработчик получает доступ для совместной работы.

При некоторых вариациях, определяемых особенностями системы и деталями принятого технологического процесса, обычный цикл работы разработчика в течение рабочего дня выглядит следующим образом:

– обновление рабочей копии;

– модификация проекта;

– фиксация изменений.

С точки зрения пользователя распределенная система отличается необходимостью создавать локальный репозиторий и наличием в командном языке двух дополнительных команд:

– команды получения репозитория от удаленного компьютера (pull);

– команды передачи своего репозитория на удаленный компьютер (push).

Первая команда выполняет слияние изменений удаленного и локального репозиториев с помещением результата в локальный репозиторий, а вторая выполняет слияние изменений двух репозиториев с помещением результата в удаленный репозиторий. Как правило, команды слияния в распределенных системах позволяют выбрать наборы изменений, которые будут передаваться в другой репозиторий или извлекаться из него, исправлять конфликты слияния непосредственно в ходе операции или после ее неудачного завершения, повторять или возобновлять неоконченное слияние. Обычно передача своих изменений в чужой репозиторий (push) завершается удачно только при условии отсутствия конфликтов. Если конфликты возникают, пользователь должен сначала выполнить слияние версий в своем репозитории (pull), и лишь затем передавать их другим.

Одной из самых популярных СКВ является распределенная система **Git**. Git поддерживает быстрое разделение и слияние версий, включает инструменты для визуализации и навигации по нелинейной истории разработки. Система Git спроектирована как набор программ, специально разработанных с учетом их использования в команде. Это позволяет на ее базе удобно создавать другие специализированные СКВ или разные пользовательские интерфейсы.

GitHub – самый большой веб-сервис для хостинга проектов и их совместной разработки, часто позиционируется как социальная сеть для разработчиков. Многие крупные проекты размещают свои официальные репозитории на сервисе. Среди них: Facebook, Twitter, Yahoo, Ruby on Rails, PHP, JUnit, jQuery, Microsoft IronRuby, osCommerce и пр.

Ядро Git представляет собой набор утилит командной строки с параметрами. Все настройки сохраняются в текстовых конфигурационных файлах. Такая реализация позволяет перемещать Git на любую платформу и интегрировать в другие системы (в частности, создавать графические Git-клиенты с любым интерфейсом). Репозиторий Git является каталогом файловой системы, в котором находятся файлы конфигурации хранилища, файлы журналов, хранящих операции, выполняемые над репозиторием, индекс, описывающий расположение файлов, и хранилище, содержащее собственно файлы. Структура хранилища файлов не отображает реальную структуру, хранящуюся в репозитории файлового дерева, она ориентирована на повышение скорости выполнения операций с репозиторием. Когда ядро обрабатывает команду изменения (неважно, во время локальных изменений либо при получении патча от другого узла), оно создает в хранилище новые файлы, соответствующие новым состояниям измененных файлов.

Концептуально данные в Git хранятся в виде модели, которая представлена на рис. 3.6.

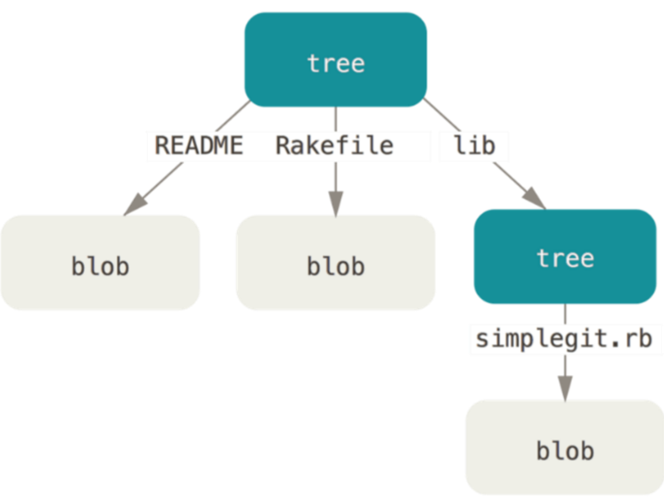


Рис. 3.6. Упрощенная модель данных Git

В Git файлы могут находиться в одном из трех состояний:

– зафиксированном;

– измененном;

– подготовленном.

Зафиксированный файл означает, что он уже сохранен в локальной базе. К измененным относятся файлы, которые изменились, но не были зафиксированы. Подготовленные файлы – это измененные файлы, отмеченные для включения в следующий коммит. Таким образом, проекты в Git имеют три части (рис. 3.7):

- каталог Git (git directory);
- рабочий каталог (working directory);
- область подготовленных файлов (staging area).

Каталог Git – это место, где СКВ хранит метаданные и базу данных объектов вашего проекта. Это наиболее важная часть СКВ, и именно она копируется при клонировании репозитория с другого компьютера.

Рабочий каталог – это извлеченная из базы копия определенной версии проекта. Эти файлы извлекаются из сжатой базы данных в каталоге Git и помещаются на диск для того, чтобы их можно было просмотреть и отредактировать.

Область подготовленных файлов – это обычный файл, обычно хранящийся в каталоге Git, содержащий информацию о том, что должно войти в следующий коммит. Иногда его называют индексом (index), но в последнее время становится стандартом называть его областью подготовленных файлов (staging area).

Рис. 3.7. Операции с каталогами в Git

Распределенная природа Git позволяет более гибко взаимодействовать разработчикам в рамках проекта. Каждый разработчик является и узлом, и хабом, т. е. каждый разработчик может как отправлять код в другие репозитории, так и поддерживать публичный репозиторий, в который другие разработчики смогут отправлять свой код, взяв его за основу. Это предоставляет огромное количество вариаций для организации рабочего процесса над проектом и/или для команды.

3.6. Задания для самоконтроля

1. Ответьте на следующие вопросы:

– Для решения каких задач используют СКВ?

– Что обозначают термины «хранилище», «история», «рабочая копия»?

– Что такое ветви (branches) и для чего они нужны?

– Для чего используются операции «разница» (diff), «слияние» (merge) и «отправка» (commit)?

– В чем разница между командами push и pull в системе Git?

2. Используя хранилища на локальном компьютере, реализуйте следующие операции над проектом:

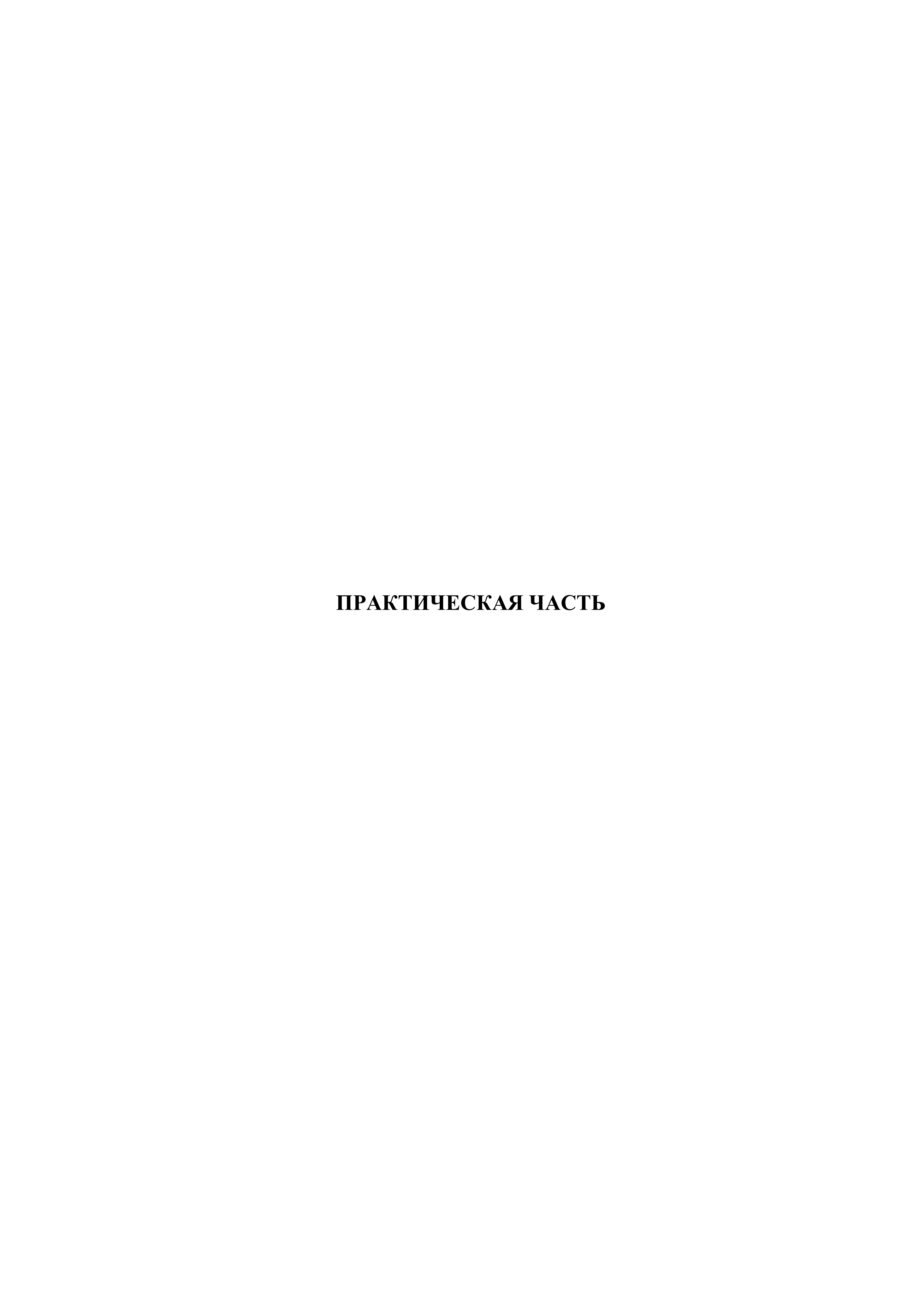
– создание хранилища и выполнение настройки системы (init, config);

– загрузка проекта с сервера и выгрузка кода на локальный компьютер (clone, remote, add, commit);

– передача истории хранилища по сети (push, pull, fetch);

– действия, связанные с ветвлением проекта (branch, checkout);

– действия, связанные с разрешением конфликтов при слиянии документов и проектов (diff, merge, mergetool).



ПРАКТИЧЕСКАЯ ЧАСТЬ

Практическая работа №1

Программное обеспечение как продукт

Цель: ознакомиться с классификацией программных продуктов по различным характеристикам; провести анализ различных программных решений для их описания на основе выделенных типовых характеристик.

Задание:

1. Согласовать вариант задания с преподавателем и в соответствии с ним выбрать список программных продуктов (табл. 1.1).
2. Выполнить сравнительный анализ программных средств по заданным характеристикам (табл. 1.2).
3. Представить результаты анализа преподавателю и защитить работу.
4. Дополнить таблицу списком функциональных возможностей для выбранных программных средств (от 3 до 10 показателей), которые выделят их схожие и уникальные функции.

Таблица 1.1

Варианты заданий для практической работы №1

Вариант	Список программных продуктов
1	Total Commander, Punto Switcher, Winamp
2	Paint.NET, SolidWorks, WhatsApp
3	Microsoft Visual Studio Code, Virtualbox, CCleaner
4	NetBeans, Eclipse, The Bat!
5	Google Таблицы, Vegas, Notepad++
6	Google Документы, GIMP, CLion
7	TortoiseCVS, Wrike, Viber
8	DaVinci Resolve, CorelDRAW, WinZip
9	7-Zip, Canva, AutoCAD
10	Jira, WeChat, Microsoft Visual Studio
11	Антивирус Касперского, Telegram, ZBrush
12	OpenOffice Writer, Microsoft Project, Android Studio
13	IntelliJ IDEA, OpenOffice Calc, Microsoft Excel
14	3Ds Max, Tinkercad, Blender
15	WinRar, PHPStorm, ESET Nod32

Таблица 1.2

Сравнительная таблица программных средств

№ п/п	**Характеристика**	**Название ПС 1**	**Название ПС 2**	**Название ПС 3**
1	Разработчики			
2	Уровень известности фирмы-разработчика (популярность)			
3	Проблемная область			
4	Целевая аудитория			
5	Наличие аналогов – степень уникальности (конкуренция сильная/умеренная/слабая/ нет аналогов)			
6	Политика использования (лицензия)			
7	Уровень сопровождения (активное/пассивное/отсутствует)			
8	Уровень интеграции с другими ПС (независимое/частично зависимое/ полная зависимость)			
9	Текущая версия (степень готовности и зрелости)			
10	Год выпуска (время нахождения на рынке – длительность продаж)			
11	Платформа (ОС)			
12	Вид ПС (системное/инструментальное/ прикладное)			
13	Класс ПС (подвид)			
14	Средства и технологии разработки			
15	Вид пользовательского интерфейса			
16	Поддержка API			
17	Требования к системе			
18	Функциональные возможности (базовые/средние/широкие)			

Примеры выполнения практической работы приведены в табл. 1.3 и 1.4.

Таблица 1.3

Пример анализа программных средств для варианта №14

№ п/п	**Характеристика**	**3Ds Max**	**Tinkercad**	**Blender**
1	**2**	**3**	**4**	**5**
1	Разработчики	*Autodesk*	*Autodesk*	*Blender Foundation*
2	Уровень известности фирмы-разработчика (популярность)	*Высокий*	*Высокий*	*Высокий*
3	Проблемная область	*Трехмерная графика*	*Трехмерная графика*	*Трехмерная графика*
4	Целевая аудитория	*Пользователи ПК*	*Пользователи ПК*	*Пользователи ПК*

Окончание табл. 1.3

1	2	3	4	5
5	Наличие аналогов – степень уникальности	*Сильная конкуренция*	*Сильная конкуренция*	*Сильная конкуренция*
6	Политика использования (лицензия)	*Trialware*	*Freeware*	*Freeware*
7	Уровень сопровождения	*Активное*	*Активное*	*Активное*
8	Уровень интеграции с другими ПС	*Независимое*	*Независимое*	*Независимое*
9	Текущая версия (степень готовности и зрелости)	*Autodesk 3ds Max 2023 (Vesta)*	*1.4*	*3.3*
10	Год выпуска (время нахождения на рынке – длительность продаж)	*1996*	*2011*	*1998*
11	Платформа (ОС)	*macOS, Windows, Linux*	*Веб*	*macOS, Windows, Linux*
12	Вид ПС	*Прикладное*	*Прикладное*	*Прикладное*
13	Класс ПС (подвид)	*Редактор трехмерной графики*	*Редактор трехмерной графики*	*Редактор трехмерной графики*
14	Средства и технологии разработки	*C++*	*JS*	*C/C++, Python*
15	Вид пользовательского интерфейса	*Графический*	*Графический*	*Графический*
16	Поддержка API	*Да*	*Да*	*Да*
17	Требования к системе	*• 64-разрядный процессор Intel® или многоядерный AMD® с поддержкой набора инструкций SSE 4.2 • ОЗУ – 4 Гбайт и более • ПЗУ – 9 Гбайт (для установки)*	*Минимальные*	*•32-битный многоядерный процессор с SSE 2 •ОЗУ – 4 Гбайт и более •ПЗУ – 500 Мбайт и более*
18	Функциональные возможности	*Широкие*	*Базовые*	*Средние*

Таблица 1.4

Пример анализа программных средств для варианта №15

№ п/п	Характеристика	WinRAR	PHPStorm	ESET NOD32
1	**2**	**3**	**4**	**5**
1	Разработчики	*Win.rar GmbH*	*JetBrains s.r.o.*	*ESET*
2	Уровень известности фирмы-разработчика (популярность)	*Высокий*	*Высокий*	*Высокий*

Продолжение табл. 1.4

1	2	3	4	5
3	Проблемная область	*Архиваторы*	*Среда разработки*	*Антивирусы*
4	Целевая аудитория	*Пользователи ПК и мобильных устройств*	*PHP разработчики*	*Пользователи ПК и мобильных устройств*
5	Наличие аналогов – степень уникальности	*Сильная конкуренция, но есть уникальные возможности (работа с форматом rar)*	*Сильная конкуренция*	*Сильная конкуренция*
6	Политика использования (лицензия)	*Shareware*	*Shareware*	*Shareware*
7	Уровень сопровождения	*Активное*	*Активное*	*Активное*
8	Уровень интеграции с другими ПС	*Независимое*	*Независимое, возможна интеграция с другими ПС*	*Независимое*
9	Текущая версия (степень готовности и зрелости)	*6.11 (04.03.2022)*	*2022.2.1 (18.08.2022)*	*15.0.23.0 (15.01.2022)*
10	Год выпуска (время нахождения на рынке – длительность продаж)	*1995*	*2009*	*1987*
11	Платформа (ОС)	*Windows, macOS, Linux, FreeBSD, Android*	*Windows, macOS, Linux*	*Кросс-платформенное ПО*
12	Вид ПС	*Системное*	*Инструментальное*	*Системное*
13	Класс ПС (подвид)	*Архиваторы*	*Среда разработки*	*Антивирусы*
14	Средства и технологии разработки	*С++*	*Java*	*Ассемблер*
15	Вид пользовательского интерфейса	*Интерфейс командной строки, графический*	*Графический*	*Графический интерфейс*
16	Поддержка API	*Нет*	*Нет*	*Нет*

Окончание табл. 1.4

1	2	3	4	5
17	Требования к системе	*Минимальные*	• *ОС: Windows 8 или новее / macOS 10.13 или новее / Linux, с поддержкой Gnome, KDE или Unity DE.* • *Современный процессор* • *ОЗУ – 2 Гбайт и более* • *ПЗУ – 3,5 Гбайт и более*	*Минимальные*
18	Функциональные возможности	*Средние*	*Широкие*	*Средние*

Практическая работа №2

Выявление требований к программному продукту

Цель: исследовать функционал предметной области; провести анализ аналогов (программных решений из одного класса) в заданной предметной области для выявления требований к разрабатываемому программному продукту.

Задание:

1. Согласовать вариант задания с преподавателем и в соответствии с ним выбрать проект программного продукта (табл. 2.1). Кроме предложенного списка проектов можно самостоятельно выбрать проект на основе тематик продуктов, рассмотренных в практической работе №1.

2. Составить перечень заинтересованных лиц и словарь терминов предметной области (глоссарий).

3. Составить подробное описание функциональных требований к разрабатываемому программному продукту.

4. На основании описания функционала провести анализ осуществимости реализации проекта. В ходе анализа ответить на следующие вопросы:

– Что даст введение продукта в эксплуатацию?

– Какие текущие проблемы существуют в предметной области и как новый продукт поможет их решить?

– Требуются ли для разработки продукта технологии, которые до этого не использовались (в команде и/или для решения задач)?

Результатом анализа требований должно явиться заключение о возможности и необходимости реализации проекта.

5. Представить результаты преподавателю и защитить работу.

Таблица 2.1

Варианты заданий для практической работы №2

Вариант	Наименование проекта
1	**2**
1	Приложение по учету личных финансов
2	Приложение для финансового контроля
3	Игра «Звездный транспортер»
4	Приложение для прогноза погоды
5	Приложение для просмотра и изменения расписания
6	Приложение с флеш-картами для запоминания материала
7	Приложение для конвертации валют
8	Приложение для перевода чисел в разные системы счисления
9	Приложение для учета списка дел
10	Игра «Змейка»
11	Приложение «Калькулятор»
12	Игра «Крестики нолики»
13	Приложение для изучения таблицы Менделеева
14	Игра «Тетрис»
15	Игра «2048» – сложение двоек и их степеней
16	Приложение для математического контроля
17	Игра «Шашки»
18	Приложение для составления рецептов
19	Приложение для планирования дня
20	Игра «Лабиринт»
21	Игра «Гонки» (объезд машиной препятствий)
22	Приложение для изучения слов на английском языке
23	Игра «Сапёр»
24	Приложение для просмотра календарных событий
25	Игра «Герой перелетает препятствия»
26	Приложение для подсчета калорий и контроля веса
27	Приложение для перевода слов в выбранный язык
28	Игра «Блиц-опрос» на выбранную тему
29	Игра «Найди предмет»
30	Приложение «Классный журнал» для школы
31	Приложение «Текстовый редактор»

Окончание табл. 2.1

1	2
32	Приложение «Органайзер»
33	Программа тестирования знаний

Практическая работа №3

Разработка требований к программному продукту

Цель: изучить процесс разработки требований к программному продукту и выполнить их анализ; оформить техническое задание на разработку программного обеспечения.

Задание:

1. Изучить теоретический материал по теме, а также справочную информацию о типовой структуре и содержании технической документации, представленную далее.

2. Для варианта проекта из практической работы №2 составить информационную модель будущего продукта, включающую в себя описание основных объектов системы и взаимодействия между ними.

3. На основании информационной модели сформировать системные требования для разработки и эксплуатации продукта.

4. На основании выявленных требований пользователей из практической работы №2, составленной информационной модели, пользовательских и системных требований составить техническое задание (ТЗ) на создание программного обеспечения.

5. ТЗ для учебного проекта должно содержать основные разделы, описанные в шаблоне (прил. 4).

6. Представить результаты преподавателю и защитить работу.

7. Дополнительно расширить структуру уже разработанного ТЗ разделами, которые представлены в шаблоне типовой формы (прил. 3).

Справочная информация

В документе «Техническое задание» (ТЗ) содержится следующая информация: назначение и область применения программы, технические, технико-экономические и специальные требования, предъявляемые к программе, необходимые стадии и сроки разработки, виды испытаний.

ТЗ должно содержать основные разделы, описанные в ГОСТ 34.602–89. Данный ГОСТ приводит рекомендуемый порядок разработки, согласования и утверждения ТЗ на создание автоматизированных систем. Несмотря на то что он был введен еще в 1990 году, его положения вполне приемлемы и в наше время.

ТЗ должно содержать следующие разделы:

- общие сведения о проекте (наименование и область применения);
- назначение и цели создания продукта;
- общие требования к продукту;
- технические требования к программному продукту;
- технико-экономические показатели;
- стадии и этапы разработки;
- порядок контроля и приемки;
- требования к документированию;
- приложения.

В зависимости от особенностей программы или программного изделия допускается уточнять содержание разделов, вводить новые разделы или объединять отдельные из них.

Примерная структура содержания ТЗ приведена в прил. 3.

Практическая работа №4

Основы моделирования программного продукта

Цель: изучить стандарты проектирования; выполнить моделирование программных продуктов с применением методологии структурного проектирования и построить блок-схемы программных средств.

Задание:

1. Изучить теоретический материал, в частности ГОСТ 19.701–90 «Схемы алгоритмов, данных, программ и систем. Условные обозначения и правила выполнения» [26]. Основные нотации, используемые для построения блок-схем, приведены в прил. 5.

2. Для варианта проекта из практической работы №2 составить модель разрабатываемого программного продукта – построить блок-схему с помощью нотаций ГОСТ 19.701–90.

3. Расширить структуру разработанного технического задания на создание программного обеспечения из практической работы №3, добавив в него описание разработанных алгоритмов (блок-схемы).

4. Представить результаты преподавателю и защитить работу.

Практическая работа №5

Методологии разработки программного продукта

Цель: исследовать методологии разработки программного обеспечения; провести анализ программного продукта для выбора подходящей методологии.

Задание:

1. Изучить информацию о современных методологиях разработки программного обеспечения. Выделить и представить критерии их классификации. По каждому критерию привести примеры (соответствующие методологии).

2. Согласовать вариант задания с преподавателем (табл. 5.1) и в соответствии с ним выполнить исследование методологии по общим характеристикам. Результаты анализа оформить в сводную таблицу по шаблону (табл. 5.2).

3. Согласовать с преподавателем тематику будущего программного продукта в соответствии с вариантом проекта из практической работы №2. Определить степень применимости методологии разработки, рассмотренной на предыдущем шаге задания, для вашего проекта, обращая внимание:

– на механизм оказания положительного влияния выбранной методологии при создании проекта;

– риски для проекта, которые несет в себе выбранная методология;
– предполагаемые роли в проекте.

4. Представить результаты преподавателю и защитить работу.

Таблица 5.1

Варианты заданий для практической работы №5

Вариант	Методология разработки ПО
1	Agile Unified Process (AUP)
2	Behavior Driven Development (BDD)
3	Constructionist Design Methodology (CDM)
4	Design Driven Testing (DDT)
5	Design-Driven Development (D3)
6	Domain-Driven Design (DDD)
7	Dynamic Systems Development Method (DSDM)
8	Extreme Programming (XP)
9	Feature Driven Development
10	Lean software development
11	Microsoft Solutions Framework (MSF)
12	Model Driven Development (MDD)
13	Open Unified Process (OpenUP)
14	Rapid application development (RAD)
15	Rational Unified Process (RUP)
16	Test-Driven Development (TDD)

Таблица 5.2

Сводная таблица описания методологии

Характеристика	Описание
Полное название (в оригинале)	
Авторы	
Год появления	
Основные принципы и критерии для выбора	
Стратегия разработки ПО	
Состав команды	
Форма и формулировка требований	
Специализированное ПО (при наличии)	
Достоинства методологии	
Предполагаемые недостатки методологии	
Существующие модификации	
Примеры успешных кейсов (реализованных проектов)	

Дополнительно предлагается обсудить в группе (в команде одногруппников) преимущества и недостатки различных методологий и выбрать наиболее подходящую для конкретного продукта по его требованиям и ТЗ. Результаты оформить в виде эссе, раскрыв в нем вопросы из задания 3.

Практическая работа №6

Организация и планирование проекта

Цель: проанализировать содержание основных фаз программного проекта, рассмотреть возможности программных средств и онлайн-ресурсов управления ИТ-проектом.

Задание:

1. Изучить теоретический материал по теме, а также справочную информацию, представленную далее.

2. Согласовать вариант задания с преподавателем и в соответствии с ним выбрать учебный проект для организации планирования (табл. 6.1).

3. Расширить список работ в предложенном проекте до 15 (или более) путем добавления новых задач или декомпозиции исходных.

4. Выполнить описание проекта, задействовав в нем не менее пяти ролей (участников проекта).

5. Выполнить анализ основных представлений проекта, структуры и содержания работ.

6. Определить следующие характеристики проекта:

- основные представления;
- команда проекта;
- связи между задачами.

7. Выбрать программное средство для визуализации плана проекта (табл. 6.2) (возможно использование и другого подходящего инструментария при согласовании с преподавателем). Выполнить описание программного средства, с помощью которого выполнялось построение диаграммы. Описание должно включать следующую информацию:

- название;
- версию;
- сведения о разработчике;
- адрес загрузки;

– режим использования;

– платформы;

– функциональные возможности.

8. Построить диаграмму Ганта в выбранном программном средстве.

9. Представить план проекта преподавателю и защитить работу.

Таблица 6.1

Варианты заданий для практической работы №6

Вариант	Проект	Базовые работы
1	Постройка дома	Проект дома, закупка материалов, строительство, подключение коммуникаций, ввод в эксплуатацию
2	Создание видеоролика	Обсуждение концепции, написание сценария, подготовка материалов, монтаж, внесение правок
3	Открытие производства	Покупка и ремонт помещения, закупка и монтаж оборудования, заключение договоров, выпуск опытной партии, анализ
4	Организация мероприятия	Создание концепции, найм организатора, выбор дизайна, организация места, проведение мероприятия
5	Разработка программного средства	Сбор требований, проектирование, разработка модулей, тестирование, внедрение
6	Ремонт объекта	Создание проекта, оценка количества материалов, закупка материалов, выполнение работ, размещение мебели
7	Сельскохозяйственные работы	Культивация, внесение удобрений, посев, внесение удобрений, уборка урожая
8	Организация поездки	Сборы, организация транспорта, проведение мероприятий
9	Исследование научно-практической проблемы	Определение цели и задач, обзор литературы, выбор метода, проведение исследования, анализ
10	Маркетинг	Описание продукта, анализ рынка, конкурентный анализ, определение стратегии, оценка
11	Торговля	Установление контактов, выяснение потребностей, презентация товара, работа с возражениями, сделка
12	Тестирование	Анализ требований, планирование, подготовка тест-кейсов, выполнение тест-кейсов, анализ
13	Анализ данных	Извлечение, исследование, создание модели, проверка модели, развертывание
14	Создание компании	Выбор сферы деятельности, изучение рынка, разработка бизнес-плана, выбор формы собственности, регистрация и оценка
15	Подготовка сайта	Согласование материалов, оформление контента, разработка сайта, тестирование, приемка сайта

Таблица 6.2

Рекомендуемые программные средства для построения диаграмм Ганта

№ п/п	Программные средства
1	GanttProject (http://ganttproject.biz)
2	OpenProj
3	Microsoft Office (Project, Excel, Visio)
4	ProjectLibre
5	draw.io

Справочная информация

Эффективное управление программным проектом напрямую зависит от правильного планирования работ, необходимых для его выполнения. План помогает менеджеру предвидеть проблемы, которые могут возникнуть на каких-либо этапах создания ПО, и разработать меры для их предупреждения или решения. План, разработанный на начальном этапе проекта, рассматривается всеми его участниками как руководящий документ, выполнение которого должно привести к успешному завершению проекта. Этот первоначальный план должен максимально подробно описывать все этапы реализации проекта.

Составление графика – одна из самых ответственных работ, выполняемых менеджером проекта. Здесь менеджер оценивает длительность проекта, определяет ресурсы, необходимые для реализации отдельных этапов работ, и представляет этапы в виде согласованной последовательности. Если данный проект подобен ранее реализованному, то график работ последнего проекта можно взять за основу для данного проекта. Но следует учесть, что на отдельных этапах нового проекта могут использоваться методы и подходы, отличные от использованных ранее.

Если проект является инновационным, первоначальные оценки длительности и требуемых ресурсов наверняка будут слишком оптимистичными, даже если менеджер попытается предусмотреть все возможные неожиданности. С этой точки зрения проекты создания ПО не отличаются от больших инновационных технических проектов. Новые аэропорты, мосты и даже новые автомобили, как правило, появляются позже первоначально объявленных сроков их сдачи или

поступления на рынок, причиной чему являются неожиданно возникшие проблемы и трудности. Именно поэтому графики работ необходимо постоянно обновлять по мере поступления новой информации о ходе выполнения проекта.

При расчете длительности этапов менеджер должен учитывать, что выполнение любого этапа не обойдется без больших или маленьких проблем и задержек. Разработчики могут допускать ошибки или задерживать свою работу, техника может выйти из строя либо аппаратные или программные средства поддержки процесса разработки могут поступить с опозданием. Если проект инновационный и технически сложный, это становится дополнительным фактором появления непредвиденных проблем и увеличения длительности реализации проекта по сравнению с первоначальными оценками. В процессе составления графика весь массив работ, необходимых для реализации проекта, разбивается на отдельные этапы и оценивается время, требующееся для выполнения каждого этапа. Обычно многие этапы выполняются параллельно.

Диаграмма Ганта (график Ганта, англ. Gantt chart) – тип столбчатых диаграмм (ленточная диаграмма). Используется для наглядного представления плана проекта или графика работ. Очень популярна в проектном менеджменте.

Идея планирования состоит в том, что его главным ресурсом является время. В связи с этим основой для принятия управленческих решений является сравнение запланированного состояния работ и фактического. На диаграммах по горизонтали указываются интервалы времени, по вертикали – операции, работы или оборудование (рис. 6.1).

	АВГУСТ							СЕНТЯБРЬ											
	1	7	14	21	28	31		1	2	3	4	5	6	7	8	9	10	11	12
Проект дома																			
Материалы																			
Постройка дома																			
Разбивка сада																			
Посадка сада																			
Новоселье																			

Рис 6.1. Пример диаграммы Ганта

Горизонтальные отрезки иллюстрируют длительность выполнения работ. Выбрав по горизонтали текущий момент времени и получив оперативную информацию о ходе проекта, можно сопоставить состояние дел по факту и состояние дел по плану.

Существует четыре возможных типа связей между работами на диаграмме Ганта (рис. 6.2):

1. Финиш – Старт. Данная связь означает, что операция B не может начаться до завершения операции A, или дата окончания операции A определяет дату начала операции B. Например, сначала надо выполнить лабораторную работу, а потом ее можно защищать.

2. Финиш – Финиш – операция B должна окончиться не раньше операции A, или дата окончания операции A определяет дату окончания операции B. Например, если вы пишите высокопроизводительное приложение (операция A) и для его отладки брали в аренду вычислительный сервер (операция B), то процесс отладки должен завершиться к сроку окончания аренды сервера.

3. Старт – Старт – операция B начинается не раньше операции A, или дата начала операции A определяет дату начала операции B. Например, распечатка отчета тесно связана с покупкой бумаги, поэтому данные задачи должны решаться практически одновременно.

4. Старт – Финиш – операция B не может закончиться, пока не начнется операция A, или дата начала операции A определяет дату окончания операции B. Например, время, на которое запланирована защита лабораторной, определяет, когда должны быть выполнены все задания по ней.

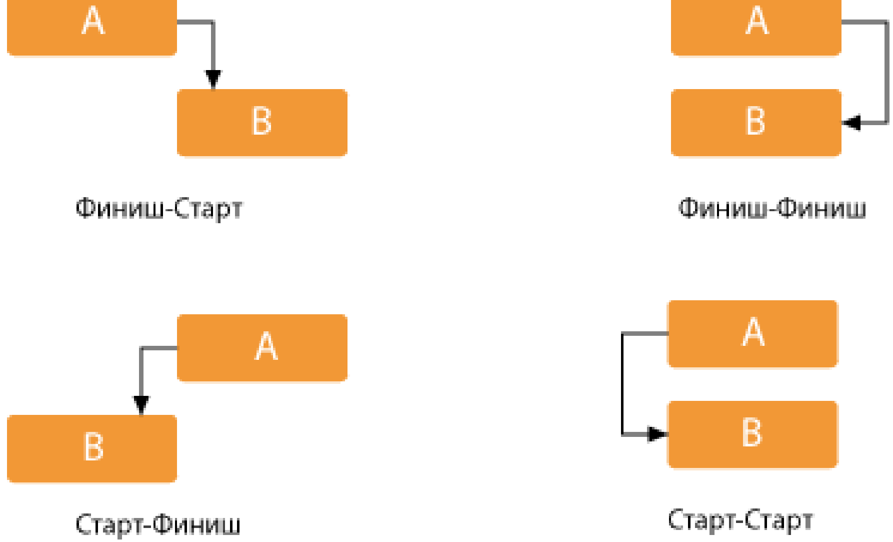


Рис. 6.2. Типы связей на диаграмме Ганта

В программном виде диаграммы Ганта могут быть реализованы различными средствами, в том числе и онлайн-сервисами (см. табл. 6.2).

Примеры диаграмм Ганта, построенных с помощью разных программных средств, представлены на рис. 6.3–6.6.

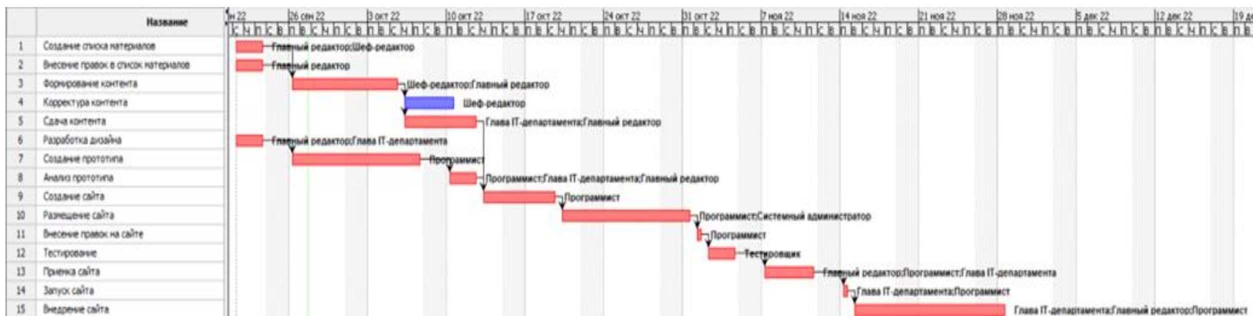


Рис. 6.3. Пример диаграммы Ганта в ProjectLibre

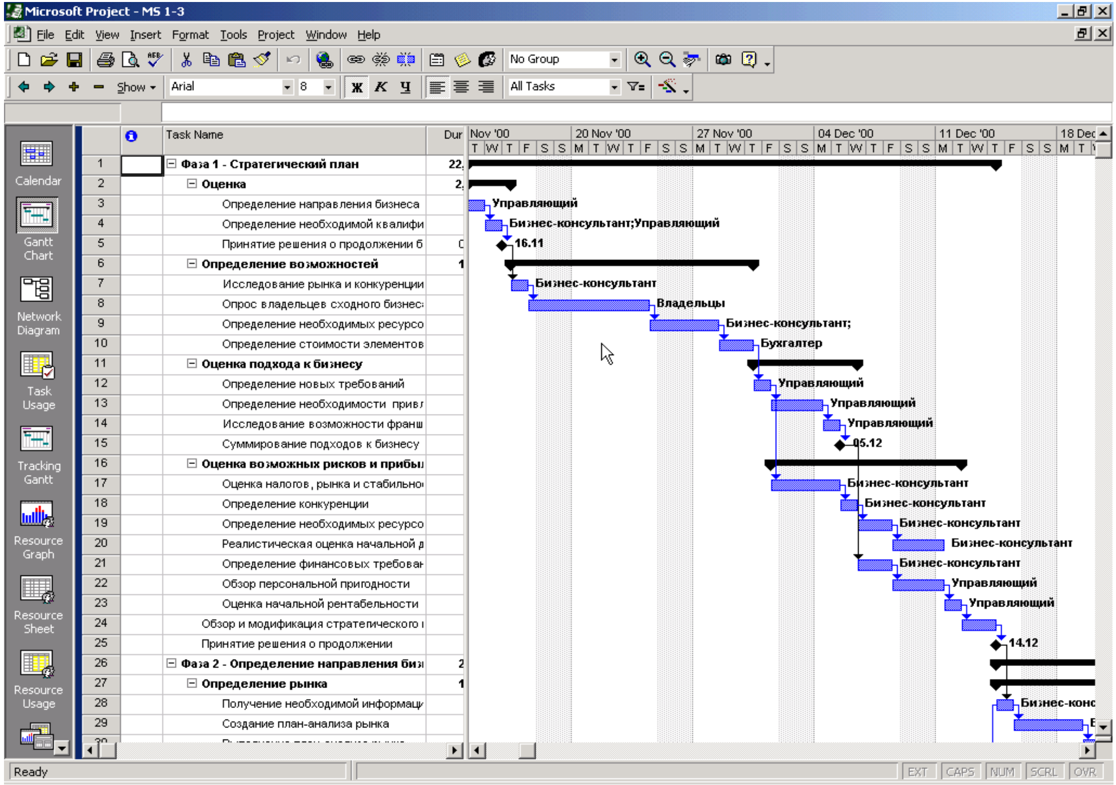


Рис. 6.4. Пример диаграммы Ганта в Microsoft Project

Этап проекта	Начало	Длительность	Задержка	Конец
Организационное собрание	24.04.2011	1	0	24.04.2011
Разработка документации	25.04.2011	11	0	05.05.2011
Общая схема	09.05.2011		3	17.05.2011
Разработка модуля 1	18.05.2011	15	0	01.06.2011
Разработка модуля 2	28.05.2011	25	-5	21.06.2011
Разработка модуля 3	26.06.2011	12	4	07.07.2011
Ввод данных	05.07.2011	12	-3	16.07.2011
Анализ данных	17.07.2011	5	0	21.07.2011
Отчет по разработке	22.07.2011	4	0	25.07.2011
Внедрение	24.07.2011	10	-2	02.08.2011
Итоговый отчет	03.08.2011	5	0	07.08.2011
Итоговое собрание	08.08.2011	1	0	08.08.2011

Рис. 6.5. Подготовка таблицы работ в MS Excel для дальнейшей автоматизации построения диаграммы Ганта

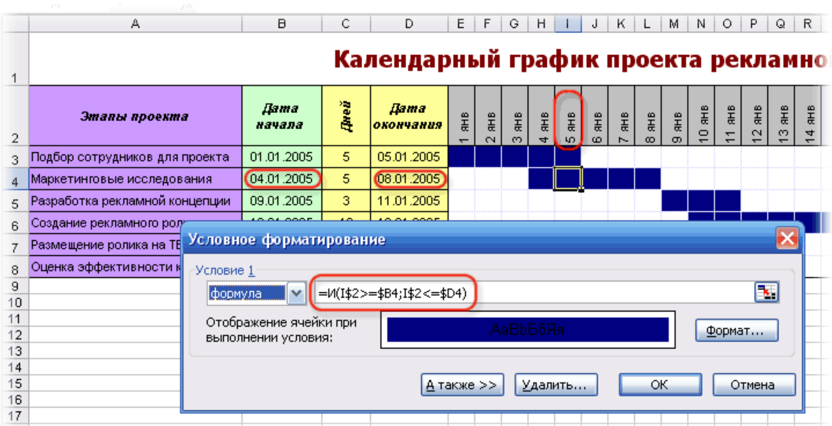


Рис. 6.6. Использование условного форматирования для построения диаграммы Ганта в MS Excel

Практическая работа №7

Средства конструирования программного продукта

Цель: ознакомиться с особенностями и возможностями инструментального программного обеспечения (интегрированной средой разработки).

Задание:

1. Изучить общие теоретические сведения об интегрированной среде разработки (Integrated Development Environment, IDE).

2. Выбрать одну из современных IDE и согласовать свой выбор с преподавателем. Провести исследование выбранной среды разработки по заданным общим характеристикам. Результаты анализа оформить в сводную таблицу по шаблону (табл. 7.1).

3. Представить результаты преподавателю и защитить работу.

Таблица 7.1

Сводная таблица описания интегрированной среды разработки

Характеристика	Описание
Полное название (в оригинале)	
Авторство (компания-разработчик)	
Год появления и дата последнего обновления (версия)	
Поддержка языков программирования	
Политика использования (стоимость)	
Поддержка платформ (ОС)	
Аппаратные требования	
Встроенные инструменты (отладка, тестирование, сборка, инспекция кода и др.)	
Возможности расширения (интегрирования)	
Популярность среди разработчиков	

Дополнительно предлагается обсудить в группе (в команде одногруппников) преимущества и недостатки различных IDE и совместно определить основные критерии при выборе средств разработки для конкретного ИТ-проекта. Результаты оформить в виде эссе.

Практическая работа №8

Системы контроля версий исходного кода

Цель: изучить на практике понятия и компоненты систем контроля версий (СКВ), приемы работы с ними; получить первоначальные навыки использования СКВ, организации коллективной разработки программного продукта; освоить специализированное ПО и распространенный сервис для работы с РСКВ Git.

Задание:

1. Изучить общие теоретические сведения об использовании СКВ исходного кода программного продукта, а также справочную информацию о базовых задачах использования РСКВ Git, представленную далее.

2. Самостоятельно отработать навыки использования хранилища на локальной машине с использованием СКВ:

– создать локальный репозиторий в директории «MyProject_username» (предварительно создать эту директорию);

– выполнить операции по настройке хранилища;

– добавить в репозиторий файл «Hello world» на любом языке программирования (проиндексировать его);

– сделать коммит (фиксацию) этого файла в репозиторий (с комментарием).

3. Изучить действия, связанные с ветвлениями и разрешением конфликтов.

4. Согласовать вариант задания с преподавателем и выбрать список задач для демонстрации своих навыков работы с СКВ. Составить последовательность команд Git, которые выполняют требуемые задачи (табл. 8.1).

5. Представить результаты выполненных операций преподавателю и защитить работу.

6. Для отработки навыков взаимодействия в команде над проектом:

– объединиться в команду по 2–4 человека с распределением ролей;

– лидеру команды создать репозиторий на удаленном сервере (можно использовать веб-сервис GitHub);

– лидеру необходимо добавить остальных участников команды и отправить главную ветку из локального репозитория своей лабораторной работы в удаленный репозиторий;

– членам команды клонировать себе проект;

– поочередно каждому участнику выполнить изменение в файлах проекта и сделать их коммит на сервер в рабочую ветку, а остальным членам команды получить обновленные файлы с сервера, не изменяя его;

– одновременно всем участникам выполнить изменение в файлах проекта и сделать их коммит на сервер в рабочую ветку, выполнить синхронизацию с сервером командой pull (разрешить конфликты различными способами в пользу изменений с сервера);

– поочередно каждому члену команды локально создать новую ветку <branch_username>, создать в ней новый файл и отправить его на сервер, а остальным членам команды скачать эту ветку. На ее основе создать свои локальные ветки <branch_username>. Внести изменения в файл путем добавления строчки «new line from @username» и последовательно слить на сервере ветки в предыдущую. Каждый участник должен получить файл со строками от всех членов команды;

– выполнить предыдущие шаги в различных последовательностях, чтобы все члены команды отработали различные ситуации в разных статусах.

Таблица 8.1

Варианты заданий для практической работы №8

Вариант	**Действия с репозиторием проекта**
1	**2**
1	– внести изменения в существующий файл, сделать коммит; – отменить коммит **способом 1**; – создать ветку **develop**, сделать в ней коммит другого файла; – создать метку «student_name tag» на ветке **develop**; – из ветки **develop** создать две ветки feature_merge_conflict**{1, 2}**; – создать коммиты в ветках (добавить файл с одинаковым полным именем, но абсолютно разным содержанием); – слить (смержить ветки) в **develop**. Разрешить конфликт в пользу ветки feature_merge_conflict**1 способом 1**

Окончание табл. 8.1

1	2
2	– создать ветку **develop**, сделать в ней коммит другого файла; – отменить коммит **способом 2**; – создать метку «student_name tag» на ветке **master**; – из ветки **master** создать две ветки feature_merge_conflict{1, 2}, создать коммиты в ветках (добавить файл с одинаковым полным именем, но абсолютно разным содержанием); – слить (смержить ветки) в **master**. Разрешить конфликт в пользу ветки feature_merge_conflict**2 способом 2**
3	– создать ветку **develop**, сделать в ней коммит другого файла; – создать метку «student_name tag» на ветке **develop**; – отменить коммит **способом 3**; – из ветки **develop** создать две ветки feature_merge_conflict{1, 2}, создать коммиты в ветках (добавить файл с одинаковым полным именем, но абсолютно разным содержанием); – слить (смержить ветки) в **develop**. Разрешить конфликт в пользу ветки feature_merge_conflict**1 способом 3**
4	– создать метку «student_name tag» на ветке **master**; – в ветке **master** сделать коммит другого файла; – отменить коммит **способом 1**; – из ветки **master** создать две ветки feature_merge_conflict{1, 2}, создать коммиты в ветках (добавить файл с одинаковым полным именем, но абсолютно разным содержанием); – слить (смержить ветки) в **master**. Разрешить конфликт в пользу ветки feature_merge_conflict**2 способом 2**
5	– внести изменения в существующий файл, сделать коммит; – создать метку «student_name tag» на ветке **master**; – отменить коммит **способом 2**; – создать ветку **develop**, сделать в ней коммит другого файла; – из ветки **develop** создать две ветки feature_merge_conflict**{1, 2}**; – создать коммиты в ветках (добавить файл с одинаковым полным именем, но абсолютно разным содержанием); – слить (смержить ветки) в **develop**. Разрешить конфликт в пользу ветки feature_merge_conflict**1 способом 3**
6	– создать ветку **develop**, сделать в ней коммит другого файла; – создать метку «student_name tag» на ветке **develop**; – отменить коммит **способом 3**; – из ветки **develop** создать две ветки feature_merge_conflict**{1, 2}**; – создать коммиты в ветках (добавить файл с одинаковым полным именем, но абсолютно разным содержанием); – слить (смержить ветки) в **develop**. Разрешить конфликт в пользу ветки feature_merge_conflict**1 способом 1**

Справочная информация

Рассмотрим типичные задачи, которые решаются с применением СКВ, и используемые для этого команды СКВ Git.

Задача 1. Создание локальной копии проекта

Описание: скачать проект с удаленного сервера https://github.com/pcottle/learnGitBranching.git на локальный компьютер.

Решение

Клонируем репозиторий:

```
$ git clone https://github.com/ pcottle/learnGitBranching.git
Cloning into ’learnGitBranching ’...
remote : Enumerating objects : 20052 , done.
remote : Counting objects : 100% (1908/1908) , done.
remote : Compressing objects : 100% (839/839) , done.
remote : Total 20052 ( delta 1212), reused 1671 (delta 1056), pack – reused 18144
Receiving objects : 100% (20052/20052), 30.16 MiB | 5.58 MiB/s, done.
Resolving deltas : 100% (11857/11857), done.
```

Используемые команды Git

Формат:

```
$ git clone [-l | --local] [-q] [--bare] [-o <name> | --origin <name>]
        [-b <name> | --branch <name>] <repository> [<directory>]
```

Параметр	Описание
-l, --local	Когда репозиторий для клонирования находится на локальном компьютере, этот флаг обходит обычный транспортный механизм с поддержкой Git и клонирует репозиторий, создавая копию указателя HEAD и всего, что находится в каталогах объектов и ссылок
-q, --quiet	«Работать тихо», т. е. ход выполнения не передается в поток ошибок
--bare	Сделайте пустой репозиторий Git, т. е. вместо создания <каталога> и помещения административных файлов в <каталог>/.git, сам <каталог> становится $GIT_DIR. Очевидно, это подразумевает --no-checkout, потому что негде проверить рабочее дерево. При использовании этой опции не создаются ни ветви удаленного отслеживания, ни связанные с ними переменные конфигурации

-o <name>, --origin <name>	Вместо использования источника удаленного имени для отслеживания вышестоящего репозитория используйте <name>. ВНИМАНИЕ! Этот параметр переопределяет clone.defaultRemoteName из конфигурации
-b <name>, --branch <name>	Вместо того чтобы установить указатель вновь созданного HEAD на ветку, на которую указывает HEAD клонированного репозитория, указатель ставится на ветку <name>. В заполненном репозитории это будет ветка, которая извлечена

Примеры:

Клонирование репозитория в определенную папку:

```
$ git clone git://myrepo.com/project.git mydir
```

Клонирование репозитория и переключение на определенную ветку:

```
$ git clone -b branch git://myrepo.com/project.git
```

Задача 2. Создание репозитория

Описание: создать локальный репозиторий для проекта MyFirstGitProject.

Решение

Создадим папку для проекта и перейдем в нее:

```
$ mkdir MyFirstGitProject
$ cd MyFirstGitProject/
```

Создадим репозиторий:

```
$ git init
Initialized empty Git repository in
    /pathToProjects MyFirstGitProject/.git/
```

Используемые команды Git

Формат:

$ git init [-q |--quiet] [--bare] [-b <branch-name> |--initial -branch =<branch-name>]

Параметр	Описание
-q, --quiet	Выводить только сообщения об ошибках и предупреждениях, все иные выходные данные будут скрыты
--bare	Создать пустой репозиторий. Если окружение GIT_DIR не задано, то оно устанавливается на текущий рабочий каталог

-b <branch-name>, --initial-branch =<branch-name>	Указанное имя будет использоваться для начальной ветки во вновь созданном репозитории. Если <branch-name> не будет указано, то откатится к имени ветки по умолчанию (сейчас это «master», но в будущем это имя может быть изменено). ВНИМАНИЕ! Имя можно настроить с помощью переменной конфигурации init.defaultBranch

Задача 3. Сохранение изменений

Описание: добавить файл main.cpp в будущий коммит и сделать его фиксацию (коммит).

Решение

Файлы, находящиеся в директории проекта:

main.cpp makefile runme

Добавим файлы в индекс:

```
$ git add main.cpp runme
```

Посмотрим файлы, которые войдут в отправку на фиксацию (коммит):

```
$ git status
On branch master
No commits yet
```

Уберем лишний файл runme:

```
$ git reset runme
```

Посмотрим файлы, которые войдут в коммит:

```
$ git status
On branch master
No commits yet
Changes to be committed:
(use "git rm --cached <file >..." to unstage )
new file: main.cpp
Untracked files:
(use "git add <file >..." to include in what will be committed)
makefile
runme
```

Сделаем первый коммит:

```
$ git commit -m "added main.cpp"
[ master (root - commit) f4d88df] added main.cpp
1 file changed, 7 insertions (+)
create mode 100644 main.cpp
```

Используемые команды Git

Формат:

$ git add [--verbose | -v] [--dry-run | -n] [--force | -f] [<pathspec>...]

Параметр	Описание
<pathspecs> …	Файлы для добавления содержимого из .Fileglobs (например, *.c) могут быть заданы для добавления всех соответствующих файлов. Здесь также можно указать начальное имя каталога для обновления индекса в соответствии с текущим состоянием каталога в целом. Например, указание dir приведет к записи не только файла dir/file1, измененного в рабочем дереве, файла dir/file2, добавленного в рабочее дерево, но также и файла dir/file3, удаленного из рабочего дерева. ВНИМАНИЕ! Старые версии Git игнорируют удаленные файлы; используйте параметр --no-all, если необходимо добавить измененные или новые файлы, но игнорировать удаленные
-n, --dry-run	На самом деле не добавляет файлы, а просто отражает, существуют ли они вообще и, возможно, будут ли они проигнорированы
-f, --force	Позволяет разрешить добавление игнорируемых файлов

Формат:

$ git status [<options>...] [< pathspec>...]

Параметр	Описание
-s, --short	Выводить всю информацию в сокращенном формате
-b, --branch	Показать ветку и информацию об отслеживании в сокращенном формате
--show-stash	Показать количество записей, которые в настоящее время спрятаны
--long	Выводить в длинном формате. Это значение ***по умолчанию***

-v, --verbose	В дополнение к именам файлов, которые были изменены, также показывать текстовые изменения, которые подготовлены для коммита. Если -v указан дважды, то это показывает изменения в рабочем дереве, которые еще не были проиндексированы

Формат:

$ git reset [-q] <pathspec>...

Параметр	Описание
-q, --quiet	Режим молчания, но при этом будут выводиться сообщения об ошибках

Формат:

$ git commit [-a | --all] [--amend] [--dry-run] [-F <file> | -m <msg>]

[--author=<author>] [--date =<date>] [--[no-]status] [-q] [<pathspec>...]

Параметр	Описание
-a, --all	Задать коммиту, чтобы он автоматически помещал файлы, которые были изменены и удалены, но при этом новые файлы, о которых вы еще не сообщили Git, не затрагиваются
-F <file>, --file=<file>	Сообщение для коммита берется из указанного файла <file>. Можно использовать для чтения сообщений со стандартного ввода
--author =<author>	Позволяет переопределить автора коммита. Чтобы указать явного автора, используется стандартный формат Author <author@example.com>. В противном случае указанное имя <author> считается шаблоном для поиска существующего коммита этого автора. Далее автор коммита копируется из первого найденного такого коммита
--date =<date>	Позволяет переопределить дату для автора, используемую в коммите
-m <msg>, --message =<msg>	Используйте аргумент <msg> в качестве сообщения для коммита. Если указано несколько опций -m, то их значения объединяются в отдельные абзацы. ВНИМАНИЕ! Параметры -m и -F являются взаимоисключающими

--amend	Заменить конец текущей ветки, создав новый коммит. Записанное дерево подготавливается как обычно, а в качестве отправной точки используется сообщение из исходного коммита, а не пустое сообщение, если не указано другое сообщение в командной строке с помощью таких параметров (-m, -F, -c и др.). Новый коммит имеет тех же родителей и автора, что и текущий (однако, параметр --reset-author может это отменить)
-q, --quiet	Не выводить итоговое сообщение коммита
--dry-run	Фиксация файлов (коммит) при этом параметре не выполняется, а только отображаются списки путей, которые должны быть зафиксированы, путей с локальными изменениями, которые не будут зафиксированы, а также путей, которые не отслеживаются
--status	Включает значение git-status в шаблон сообщения коммита при использовании редактора для подготовки сообщения. Данное значение включено ***по умолчанию***. ВНИМАНИЕ! Этот параметр может использоваться для переопределения переменной конфигурации commit.status
--no-status	Не включает значение git-status в шаблон сообщения коммита при использовании редактора для подготовки сообщения. Данное значение может быть включено ***по умолчанию***

Пример. Добавить все измененные или удаленные файлы во время предыдущей фиксации (в предыдущий коммит):

```
$ git commit -a --ammend
```

Задача 4. Просмотр истории изменений

Описание: просмотреть историю коммитов и изменений.

Решение

Просмотрим историю коммитов для текущей ветки:

```
$ git log
commit beddb757ffedd28e12a629afc555f4206d14d0a1 (HEAD -> master)
Author : Some Dev <user@mail.box>
Date : Mon Jun 27 00:53:12 2022 +0000
added makefile
commit f4d88dff62dc6d214f0a87fd6ce83a0caeed6fa1
Author : Some Dev <user@mail.box>
Date : Mon Jun 27 00:15:22 2022 +0000
```

```
added main.cpp
```

Просмотрим изменения между версиями:

```
$ git diff f4d8 bedd
diff --git a/makefile b/makefile
new file mode 100644
index 0000000.. e6641c4
--- /dev/null
+++ b/makefile
@@ -0,0 +1,3 @@
+all:
+ g++ -O3 main.cpp -o runme
```

Используемые команды Git

Формат:

$ git log [<options>] [<revision -range>] [<path>...]

Параметр	Описание
--follow	Продолжить перечисление истории файла за пределами переименования (работает только для одного файла)
--pretty [=<format>], --format =<format>	Вывести содержимое журналов коммитов в заданном формате, где <format> может иметь одно из следующих значений: oneline, short, medium, full, fuller, reference, email, raw и т. д.
--abbrev-commit	Вместо того чтобы отображать полное 40-байтовое шестнадцатеричное имя объекта коммита, показывает только префикс, который однозначно определяет объект. При этом параметр --abbrev=<n> может использоваться для указания минимальной длины префикса. Это поможет сделать формат --pretty=oneline намного более читаемым, например, для тех, кто будет использовать вывод в консоль с 80-колоночным представлением
--no-abbrev-commit	Показать полное 40-байтовое шестнадцатеричное имя объекта коммита. Это отменяет --abbrev-commit, явный или подразумеваемый другими опциями, такими как --oneline. ВНИМАНИЕ! Параметр переопределяет переменную log.abbrevCommit

--oneline	Это сокращение от --pretty=oneline--abbrev-commit, используемое вместе
--full-diff	Без этого флага git log -p <path> показывает коммиты, которые касаются указанных путей, и различает примерно те же указанные пути. При этом полная разница отображается для коммитов, которые касаются указанных путей. Это означает, что <path> ограничивает только коммиты и не ограничивает diff для этих коммитов
<revision-range>	Показать только коммиты в указанном диапазоне ревизий. Если параметр не указан, то ***по умолчанию*** используется HEAD, т. е. отображается вся история изменений, ведущая к текущему коммиту
<path>	Показывать только те коммиты, которых достаточно для понимания того, как появились файлы, соответствующие указанным путям

Формат:

$ git diff [<options>] <commit> <commit> [<path>...]

Параметр	Описание
--diff-algorithm = { minimal | patience | histogram | myers }	Выбирается алгоритм для сравнения. Базовым алгоритмом является «жадное» сравнение (myers), если иной не указан явно. Значение myers включено ***по умолчанию***. Кроме того, оно может быть заменено: – на minimal – предполагает трату дополнительного времени, чтобы убедиться, что возникает минимально возможная разница (рекомендуется при сравнении большого числа изменений в больших файлах); – patience – использует алгоритм «терпеливых различий» при создании патчей. Ключевое отличие в том, что по-разному сопоставляются одинаковые строки между двумя файлами (рекомендуется при переупорядочивании кода/контента, либо когда одни и те же строки добавляются и удаляются в одном и том же файле); – histogram – алгоритм «гистограмма» расширяет алгоритм «терпеливых» различий, чтобы поддерживать общие элементы с низкой частотой появления (рекомендуется в тех случаях, когда у алгоритма «терпеливых» различий возникают проблемы с представлением некоторых сделанных изменений). ВНИМАНИЕ! Чтобы после переопределения вновь использовать алгоритм по умолчанию, необходимо указать параметр --diff-algorithm=default

<path>…	Используется для ограничения различий между указанными путями (можно задать имена каталогов и получить различия по всем файлам, находящимся в них)

Примеры:

Посмотреть краткую историю коммитов:

```
$ git log --oneline
c38fa30 (HEAD -> master) added namespace std
8 c37a87 refactor output
dec5b86 (feature2) added sum function
beddb75 (tag: v0.1) added makefile
f4d88df added main.cpp.
```

Посмотреть краткую историю коммитов по файлу makefile:

```
$ git log --oneline makefile
beddb75 (tag: v0.1) added makefile
```

Задача 5. Отмена изменений

Описание: отменить изменения последнего коммита.

Решение

Способ 1

История изменений:

```
$ git log --oneline
beddb75 (HEAD -> master) added makefile
f4d88df added main.cpp
```

Переходим на первый коммит:

```
$ git checkout f4d8
Note : switching to ’f4d8 ’.
```

You are in ’detached HEAD’ state. You can look around, make experimental changes, and commit them, and you can discard any commits you make in this state without impacting any branches by switching back to a branch.

HEAD is now at f4d88df added main.cpp

Нам сообщают про состояние detached HEAD.

Посмотрим состояние проекта:

```
$ git log --oneline
```

f4d88df (HEAD) added main.cpp

$ ls

main.cpp runme

Отменились действия последнего коммита (файла makefile нет).

Для дальнейшей комфортной работы создадим новую ветку, чтобы выйти из состояния detached HEAD:

$ git checkout -b new_branch

Switched to a new branch ’new_branch’

Проверяем:

$ git log --oneline

f4d88df (HEAD -> new_branch) added main.cpp

Способ 2

История изменений:

$ git log --oneline

beddb75 (HEAD -> master) added makefile

f4d88df added main.cpp

Воспользуемся командой git revert и отменим действия последнего коммита:

$ git revert --no-edit HEAD~0

[master 737182 f] Revert "added makefile"

Date : Tue Jun 28 01:09:22 2022+0000

1 file changed, 3 deletions (-)

delete mode 100644 makefile

Состояние проекта:

$ git log --oneline

737182 f (HEAD -> master) Revert "added makefile"

beddb75 added makefile

f4d88df added main.cpp

$ ls

main.cpp runme

git revert создал коммит, который отменяет действия предыдущего коммита.

Способ 3

История изменений:

```
$ git log --oneline
beddb75 (HEAD -> master) added makefile
f4d88df added main.cpp
```

Отменим действия предыдущего коммита и перейдем на первый:

```
$ git reset --hard f4d8
HEAD is now at f4d88df added main.cpp
```

Состояние проекта:

```
$ git log --oneline
f4d88df (HEAD -> master) added main.cpp
$ ls
main.cpp runme
```

Используемые команды Git

Формат:

$ git revert [--[no-]edit] [-n] [-m parent-number] [-s] [-S[<keyid>]] <commit>...

Параметр	Описание
<commit>...	Указываются коммиты для возврата (откат фиксаций). Используется для получения более полного списка способов написания имен коммитов. Наборы коммитов также могут быть заданы, но без выполнения их обхода
-e, --edit	С этой опцией git revert позволит отредактировать сообщение для коммита перед возвратом коммита. Данное значение задано ***по умолчанию***, если вы запускаете команду из терминала
-m parent-number, --mainline parent-number	Обычно невозможно отменить слияние, потому что неизвестно, какая из сторон слияния была основной линией. Эта опция указывает номер родителя основной ветки (начиная с 1) и позволяет отменить изменение относительно указанного родителя. Отмена коммита слияния задает, что никогда не возникнет изменений в дереве коммитов потомков этого слияния. В результате более поздние слияния принесут в дерево только изменения, внесенные коммитами, которые не являются предками ранее отмененного слияния
--no-edit	С этой опцией git revert не будет запускать редактор сообщений коммита

Формат:

$ git checkout [-q] [-f] [-m] [<branch>] [-b <new -branch>]

Параметр	Описание
-q, --quiet	«Тихий» режим, т. е. подавление сообщений обратной связи
-f, --force	При переключении веток продолжить работу, даже если индекс или рабочее дерево отличаются от HEAD и на пути встречаются неотслеживаемые файлы. Используется для удаления локальных изменений и любых неотслеживаемых файлов или каталогов, которые находятся на пути
-b <new-branch>	Создать новую ветку с именем <new-branch> и начать ее с точки <start-point>
-m, --merge	Если при переключении ветвей у вас есть локальные модификации одного или нескольких файлов, которые отличаются между текущей ветвью и ветвью, на которую вы переключаетесь, то команда отказывается переключать ветки, чтобы сохранить ваши изменения в контексте. ВНИМАНИЕ! С этой опцией выполняется трехстороннее слияние между текущей ветвью, содержимым рабочего дерева и новой ветвью. После этого текущей позицией станет новая ветка. При возникновении конфликта слияния записи индекса для конфликтующих путей остаются не слитыми. Необходимо разрешить конфликты и пометить разрешенные пути с помощью команды git add (или git rm, если слияние должно привести к удалению пути). При извлечении путей из индекса этот параметр позволяет воссоздать конфликтующее слияние для указанных путей. ВНИМАНИЕ! При переключении веток с помощью --merge поэтапные изменения могут быть потеряны

Задача 6. Работа с удаленным (дистанционным) сервером

Описание: добавить новый удаленный сервер, посмотреть список удаленных серверов, отправить изменения на сервер, выполнить синхронизацию с сервером.

Решение

История изменений:

```
$ git log --oneline
beddb75 (HEAD -> master) added makefile
```

f4d88df added main.cpp

Добавление нового удаленного сервера:

$ git remote add origin

git@github.com: SUPERSTUDENT/forEducationalPurposes.git

Просмотр списка удаленных серверов:

$ git remote -v show

origin git@github.com: SUPERSTUDENT/forEducationalPurposes.git (fetch)

origin git@github.com: SUPERSTUDENT/forEducationalPurposes.git (push)

Отправка изменений на сервер:

$ git push origin master

Enumerating objects: 6, done.

Counting objects: 100\% (6/6), done.

Delta compression using up to 8 threads

Compressing objects: 100\% (4/4), done.

Writing objects: 100\% (6/6), 584 bytes | 584.00 KiB/s, done.

Total 6 (delta 0), reused 0 (delta 0), pack -reused 0

To github.com: SUPERSTUDENT/forEducationalPurposes.git

* [new branch] master -> master

Обновление удаленных веток:

$ git fetch origin master

From github.com: SUPERSTUDENT/forEducationalPurposes

* branch master -> FETCH_HEAD

Синхронизация текущей ветки с сервером:

$ git pull origin master

From github.com: SUPERSTUDENT/forEducationalPurposes

* branch master -> FETCH_HEAD

Already up to date .

Предыдущая команда аналогична выполнению:

$ git fetch origin master

$ git merge master

Используемые команды Git

Формат:

$ git remote [-v | --verbose]

$ **git remote add** <name> <URL>

$ **git remote rename** <old> <new>

$ **git remote remove** <name>

$ **git remote** [-v | --verbose] show <name>...

$ **git remote** [-v | --verbose] update [(<group> | <remote>) ...]

Параметр	Описание
-v, --verbose	Устанавливает более подробный режим – показывать дистанционный URL-адрес после имени. Для отложенных дистанционных соединений показывает, какой фильтр установлен. ВНИМАНИЕ! Параметр обязательно следует указать между самой командой **remote** и ее подкомандой

Формат:

$ git fetch [<options>] [<repository> [<refspec>...]]

Параметр	Описание
--all	Получить все изменения из дистанционного репозитория
-q, --quiet	Преобразуется в git-fetch-pack и отключает все другие внутренние команды git. Ход выполнения не протоколируется в стандартном потоке ошибок
-v, --verbose	Включить полное информирование
--progress	Статус выполнения протоколируется в стандартном потоке ошибок при подключении к терминалу (если не указан параметр -q). Этот флаг принудительно устанавливает статус выполнения, даже если стандартный поток ошибок не направлен на терминал

Формат:

$ git push [--atomic] [-n | --dry-run] [-f |--force] [-v |--verbose]
[<repository> [<refspec>...]]

Параметр	Описание
-f, --force	Команда отказывается обновлять дистанционную ссылку, которая не является предком для локальной используемой для ее перезаписи ссылки. При использовании параметра --force-with-lease команда отказывается обновлять дистанционную ссылку, текущее значение которой не соответствует ожидаемому. ВНИМАНИЕ! Использование этого флага может привести к потере коммитов на дистанционном репозитории. Используйте его с осторожностью

-q, --quiet	Подавлять весь вывод, включая список обновленных ссылок, пока не произойдет ошибка. Ход выполнения не протоколируется в стандартный поток ошибок
-v, --verbose	Работать с полным оповещением
-n, --dry-run	Делать все, кроме фактической отправки обновлений

Формат:

$ git pull [<options>] [<repository> [<refspec>...]]

Параметр	Описание
-q, --quiet	Преобразуется в две основные команды: git fetch для подавления отчетов во время передачи и git merge для подавления выходных данных при слиянии
-v, --verbose	Аналогично параметру --verbose, который используется в командах git fetch и git merge

Задача 7. Работа с ветками

Описание: посмотреть все ветки, создать новую ветку, переименовать ветку, удалить ветку, удалить удаленную ветку, перейти к ветке.

Решение

Посмотреть все ветки:

```
$ git branch
* master
```

Создать новую ветку develop:

```
$ git branch develop
```

Список веток:

```
$ git branch
develop
* master
```

Перейти на ветку develop:

```
$ git checkout develop
Switched to branch ’develop’
```

Создать ветку fix-1:

```
$ git branch fix-1
```

Список веток:

```
$ git branch
```

* develop

fix-1

master

Переименовать ветку fix-1 в ветку fix-0:

$ git branch -m fix-1 fix-0

Список веток:

$ git branch

* develop

fix-0

master

Удалить ветку fix-0:

$ git branch -d fix-0

Deleted branch fix-0 (was beddb75).

Список веток:

$ git branch

* develop

master

Используемые команды Git

Формат:

$ git branch [--show-current] [(-r | --remotes) | (-a | --all)] [--list] [<pattern>...]

$ **git branch** (-m | -M) [<oldbranch>] <newbranch>

$ **git branch** (-c | -C) [<oldbranch>] <newbranch>

$ **git branch** (-d | -D) [-r] <branchname>...

Параметр	Описание
-f, --force	Сбросить ветку в <начальную точку>, даже если ветка уже существует. Без -f команда отказывается изменять существующую ветку
-**d, --delete**	Удалить ветку. Ветвь должна быть полностью объединена со своей восходящей ветвью или с HEAD, если восходящая ветвь не была установлена с помощью --track или --set-upstream-to
-D	Преобразуется в два параметра: --delete --force. Разрешает удаление ветки независимо от ее статуса слияния или даже от того, указывает ли она на действительный коммит
-m, --move	Переместить или переименовать ветку вместе с ее конфигурацией и журналом ссылок

-M	Преобразуется в два параметра: --move --force. Разрешает переименование ветки, даже если новое имя уже существует
-c, --copy	Скопировать ветку вместе с ее конфигурацией и журналом ссылок
-C	Преобразуется в два параметра: --copy --force. Разрешает переименование ветки, даже если новое имя уже существует
-r, --remotes	Перечислить или удалить (если используется совместно с -d) ветки дистанционного отслеживания
-a, --all	Перечислить ветки дистанционного отслеживания и локальные ветки
-l, --list	Перечислить ветки. Используется с параметром <pattern>... Можно перечислить только те ветки, которые соответствуют шаблону
--show-current	Вывести название текущей ветки. В отдельном состоянии HEAD ничего не печатается
-q, --quiet	Включает «режим тишины» при создании или удалении ветки, подавляя сообщения об ошибках

Задача 8. Слияние веток

Описание: слить ветки feature1 и feature2 в ветку master.

Решение

Способ 1

История изменений:

```
$ git log --oneline
dec5b86 (HEAD -> feature2) added sum function
beddb75 (master, develop) added makefile
f4d88df added main.cpp
```

Перейдем к ветке, для которой будем применять изменения из другой:

```
$ git checkout develop
Switched to branch ’develop’
```

Выполним слияние изменений из ветки feature2:

```
$ git merge feature2
Updating beddb75 .. dec5b86
Fast - forward
main.cpp | 8 +++++++ -
```

1 file changed , 7 insertions (+) , 1 deletion (-)

История изменений:

$ git log -- oneline

dec5b86 (HEAD -> develop, feature2) added sum function

beddb75 (master) added makefile

f4d88df added main.cpp

Выполним слияние изменения из ветки feature1:

$ git merge feature1

Auto - merging main.cpp

CONFLICT (content): Merge conflict in main.cpp

Automatic merge failed; fix conflicts and then commit the result.

Произошел конфликт при слиянии.

Состояние:

$ git status

On branch develop

You have unmerged paths.

(fix conflicts and run "git commit")

(use "git merge --abort" to abort the merge)

Unmerged paths :

(use "git add <file >..." to mark resolution)

both modified : main.cpp

Untracked files :

(use "git add <file >..." to include in what will be committed)

runme

no changes added to commit (use "git add" and/or "git commit -a"

Файл main.cpp, который был изменен в обоих ветках (рис. 8.1).

```
1  #include <iostream>
2
3  using namespace std;
4
5  void print_fn() {
6      cout << "Hello world from function" << endl;
7  }
8
9  using namespace std;
10
11 void print_sum() {
12     cout << "2 + 2 = " << 2 + 2 << endl;
13 }
14
15 int main() {
16 <<<<<<< HEAD
17     print_sum();
18 =======
19     print_fn();
20 >>>>>>> feature1
21     return 0;
22 }
23
24
```

Рис. 8.1. Ситуация конфликта

Для того чтобы разрешить конфликт, надо изменить файл (выбрать нужную версию или выполнить слияние вручную), добавить его в индекс и сделать коммит:

$ git add main.cpp

$ git commit -m “merged feature1 to develop”

[develop 694 ea16] merged feature1 to develop

Итоговая история изменений:

$ git log --oneline

694 ea16 (HEAD -> develop) merged feature1 to develop

dec5b86 (feature2) added sum function

b45dc56 (feature1) added namespace std

b56daa0 refactor output

beddb75 (master) added makefile

f4d88df added main.cpp

Способ 2

История изменений:

$ git log --oneline

dec5b86 (HEAD -> feature2) added sum function

beddb75 (master, develop) added makefile

f4d88df added main.cpp

Сделаем перенос изменений из ветки feature2 относительно ветки master на ветку feature1:

$ git rebase --onto feature1 master feature2

Auto-merging main.cpp

CONFLICT (content): Merge conflict in main.cpp

error : could not apply dec5b86 ... added sum function

hint : Resolve all conflicts manually , mark them as resolved with hint : "git add/rm <conflicted_files>", then run "git rebase --continue ".

hint : You can instead skip this commit : run "git rebase --skip ".

hint : To abort and get back to the state before "git rebase", run "git rebase --abort".

Could not apply dec5b86 ... added sum function

Произошел конфликт.

Редактируем файл, добавляем его в индекс и выполняем rebase:

$ git add main.cpp

$ git rebase -- continue

[detached HEAD db95e62] added sum function

1 file changed, 5 insertions (+)

Successfully rebased and updated refs/heads/feature2.

Текущая история изменений:

$ git log --oneline

db95e62 (HEAD -> feature2) added sum function

b45dc56 (feature1) added namespace std

b56daa0 refactor output

beddb75 (master, develop) added makefile

f4d88df added main.cpp

Чтобы все изменения оказались в ветке develop, сделаем fast-forward с ветки develop до ветки feature2:

$ git checkout develop

Switched to branch ’develop’

$ git merge feature2

Updating beddb75 .. db95e62

```
Fast-forward
main.cpp | 15 +++++++++++++ - -
1 file changed, 13 insertions (+), 2 deletions (-)
```

Итоговая история изменений:

```
$ git log --oneline
db95e62 (HEAD -> develop, feature2) added sum function
b45dc56 (feature1) added namespace std
b56daa0 refactor output
beddb75 (master) added makefile
f4d88df added main.cpp
```

Способ 3

История изменений:

```
$ git log --oneline
dec5b86 (HEAD -> feature2) added sum function
beddb75 (master, develop) added makefile
f4d88df added main.cpp
```

Выполним слияние ветки feature2 с веткой develop (fast-forward):

```
$ git checkout develop
Switched to branch 'develop'
$ git merge feature2
Updating beddb75 .. dec5b86
Fast-forward
main.cpp | 8 +++++++ -
1 file changed, 7 insertions (+), 1 deletion (-)
```

История изменений ветки feature1:

```
$ git log feature1 --oneline
b45dc56 (feature1) added namespace std
b56daa0 refactor output
beddb75 (master) added makefile
f4d88df added main.cpp
```

Добавим два коммита из ветки feature1 к нашей текущей ветке develop:

```
$ git cherry-pick b56d b45d
Auto-merging main.cpp
```

```
CONFLICT (content): Merge conflict in main.cpp
error : could not apply b56daa0 ... refactor output
hint : After resolving the conflicts, mark them with
hint : "git add/rm <pathspec>", then run
hint : "git cherry-pick -- continue".
hint : You can instead skip this commit with "git cherry-pick --skip".
hint : To abort and get back to the state before "git cherry-pick",
hint : run "git cherry-pick --abort".
```

Произошел конфликт.

Редактируем файл, добавляем индекс, продолжаем cherry-pick:

```
$ git add main.cpp
$ git cherry-pick -- continue
[ develop 8 c37a87 ] refactor output
Date : Wed Jun 29 00:51:45 2022 +0000
1 file changed, 5 insertions (+)
Auto-merging main.cpp
[develop 0266 ad1] added namespace std
Date : Wed Jun 29 00:53:03 2022+0000
1 file changed, 4 insertions (+), 2 deletions (-)
```

Итоговая история изменений:

```
$ git log --oneline
0266 ad1 (HEAD -> develop) added namespace std
8 c37a87 refactor output
dec5b86 (feature2) added sum function
beddb75 (master) added makefile
f4d88df added main.cpp
```

Используемые команды Git

Формат:

$ git merge [-n] [--stat] [--[no-]commit] [--[no-]edit] [-m <msg>] [-F <file>]
[--into-name <branch>] [<commit>...]

Параметр	Описание
--[no-]commit	Выполнить слияние и зафиксировать результат. Эту опцию можно использовать для переопределения --no-commit. С помощью --no-commit можно выполнить слияние и остановить непосредственно перед созданием коммита слияния, чтобы пользователю дать возможность проверить и дополнительно настроить результат слияния перед коммитом. ВНИМАНИЕ! Быстрые обновления не создают фиксацию слияния, и поэтому невозможно остановить эти слияния с помощью --no-commit. Если есть необходимость, чтобы ветка не была изменена или обновлена командой слияния, тогда используйте параметры --no-ff с --no-commit
-e, --edit, --no-edit	Перед совершением успешного механического слияния вызвать редактор для дальнейшего изменения автоматически сгенерированного сообщения слияния. Рекомендуется использовать, чтобы объяснить и обосновать слияние. Опцию --no-edit можно использовать для принятия автоматически сгенерированного сообщения (*не рекомендуется*)
--ff, --no-ff, --ff-only	Указывает, каким образом обрабатывается слияние, если общая история уже является потомком текущей истории. --ff используется ***по умолчанию***. Если происходит слияние аннотированного (подписанного) тега, который не хранится на своем естественном месте в ссылках/тегах/иерархии, то предполагается --no-ff
--stat, -n, --no-stat	Показать статистику различий в конце слияния. Если использовать совместно с параметром -n или --no-stat, то в конце слияния статистика различий не показывается
-q, --quiet	«Режим тишины» – действовать без вывода прогресса в терминал состояния. Подразумевает --no-progress
-v, --verbose	Выводить в терминал все сообщения
--autostash, --no-autostash	Автоматически создавать запись во временном тайнике перед началом операции, записывать ее в специальную ссылку MERGE_AUTOSTASH и применять после завершения операции. Это позволяет запустить операцию на «грязном» рабочем дереве. ВНИМАНИЕ! Используйте с осторожностью: финальное применение такого «тайника» после успешного слияния может привести к нетривиальным конфликтам
-m <msg>	Установка сообщения, которое будет использоваться для коммита слияния (в случае его создания)

--into-name <branch>	Подготовка сообщения слияния, которое будет использоваться по умолчанию. Имя ветки <branch> используется как шаблон вместо имени реальной ветки, с которой производится слияние
-F <file>, --file =<file>	Прочитать сообщение, которое будет использоваться для коммита слияния (в случае его создания)
<commit>...	Коммиты, которые сливаются в текущую ветку (обычно это головы других веток). Указание более одного коммита создаст слияние с более чем двумя родителями (в литературе называется «слиянием осьминога»)

Формат:

$ git rebase [--onto <newbase> | --keep-base] [<upstream> [<branch>]]

Параметр	Описание
<upstream>	Восходящая ветвь для сравнения. Может быть не только имя существующей ветки, но и любой допустимый коммит. ***По умолчанию*** настроен восходящий поток для текущей ветки
<branch>	Рабочая ветка. ***По умолчанию*** используется указатель HEAD
--onto <newbase>	Отправная точка для создания новых коммитов. Может быть не только имя существующей ветки, но и любой допустимый коммит. Если параметр --onto не указан, то отправной точкой является <upstream>
--keep-base	Установить начальную точку для создания новых коммитов в базе слияния <upstream> и <branch>. Запуск git rebase --keep-base <upstream> <branch> эквивалентен запуску git rebase --onto <upstream>... <branch> <upstream> <branch>. Эта опция полезна, когда разрабатывается функция поверх восходящей ветки. Пока эта функция находится в разработке, восходящая ветвь может продвигаться вперед, и может быть не лучшей идеей продолжать перебазирование поверх восходящей ветки, но сохранять базовую фиксацию «как есть»

Формат:

$ git cherry-pick [--edit] [-n] [-m <parent-number>] [-s] [-x] [--ff] <commit>...

Параметр	Описание
<commit>...	Указывает, что Git обязуется выбирать коммит. Можно передавать наборы коммитов, но по умолчанию обход не выполняется, как если бы была указана опция --no-walk. ВНИМАНИЕ! При указании диапазона все аргументы <commit>... передаются в один обход ревизии
-e, --edit	С этой опцией git cherry-pick позволяет редактировать сообщение коммита перед фиксацией
-x	Позволяет при записи коммита добавить строку-описание к исходному сообщению коммита, чтобы указать, из какого коммита было выбрано это изменение. Это делается только для выбора коммита без конфликтов. Нет смысла использовать эту опцию при выборе коммитов из частной ветки, потому что информация будет бесполезна для получателя. Если выбираются коммиты между двумя общедоступными ветками, то добавление такой информации может быть полезным
-m <parent-number>, --mainline <parent-number>	Опция указывает родительский номер (начиная с 1) основной ветки и позволяет воспроизвести изменение относительно указанного родителя. Обычно выбор слияния недопустим, поскольку неизвестно, какая сторона слияния является основной линией
-n, --no-commit	Применяет изменения, необходимые для выбора каждого именованного коммита в рабочем дереве, и индексирует без каких-либо коммитов
--ff	Если текущий указатель HEAD совпадает с родителем выбранного коммита, то будет выполнена быстрая перемотка вперед к этому коммиту

Задача 9. Работа с метками

Описание: добавить метку версии v0.1 на ветку master, выполнить слияние изменения из ветки develop в ветку master, добавить аннотированную метку v0.2.

Решение

Перейдем на ветку master:

```
$ git checkout master
Switched to branch ’master’
```

Добавим метку v0.1:

```
$ git tag v0.1
```

Просмотрим метки:

```
$ git tag
v0.1
```

Выполним слияние изменения из ветки develop:

```
$ git merge develop
Updating beddb75..0266 ad1
Fast-forward
main.cpp | 17 +++++++++++++++ - -
1 file changed , 15 insertions (+), 2 deletions (-)
```

Добавим аннотированную метку v0.2:

```
$ git tag -a v0.2 -m "version 0.2"
```

Просмотрим метки:

```
$ git tag
v0.1
v0.2
```

Просмотрим информацию о метке v0.1:

```
$ git show v0.1
commit beddb757ffedd28e12a629afc555f4206d14d0a1 (tag: v0.1)
Author : Some Dev <user@mail.box>
Date : Mon Jun 27 00:53:12 2022 +0000
added makefile
```

Просмотрим информацию о метке v0.2:

```
$ git show v0.2
tag v0.2
Tagger : Some Dev user@mail.box
Date : Thu Jun 30 01:44:37 2022 +0000
version 0.2
commit 0266 ad1eec89be4af0d8f2d9e7bd89d6cb17838a (HEAD -> master, tag: v0.2, develop)
Author : Some Dev <user@mail.box>
Date : Wed Jun 29 00:53:03 2022 +0000
added namespace std
```

Используемые команды Git

Формат:

$ git tag [-a | -s | -u <keyid>] [-f] [-m <msg>| -F <file>] [-e] <tagname>
[<commit>|<object>]

$ **git tag** [-n[<num>]] -l

$ **git tag** -d <tagname>...

$ **git tag** -v <tagname>...

Параметр	Описание
-a, --annotate	Сделать неподписанный аннотированный объект тега
-s, --sign	Создать тег с подписью GPG (или GnuPG), используя ключ адреса электронной почты по умолчанию. Поведение по умолчанию управляется переменной конфигурации tag.gpgSign, если она существует
--no-sign	Переопределить переменную конфигурации tag.gpgSign, которая настроена на принудительную подпись каждого тега
-u <keyid >, --local-user =<keyid>	Создать тег с подписью GPG, используя указанный в опции ключ
-f, --force	Заменить существующий тег заданным именем (вместо ошибки)
-d, --delete	Удалить существующие теги с указанными именами
-v, -- verify	Проверить подписи GPG для указанных имен тегов
-n <num>	Команда указывает, сколько строк из аннотации, если таковые имеются, печатаются при использовании -l. Подразумевает --list. ***По умолчанию*** строки аннотаций не печатаются. Если номер -n не указан, печатается только первая строка. Если тег не аннотирован, вместо него отображается сообщение коммита
-l, --list	Перечислить теги. Используется с параметром <pattern>... Запуск команды git tag без аргументов также выводит список всех тегов
-m <msg>, --message =<msg>	Использовать сообщение с данным тегом (вместо подсказки). Если задано несколько параметров -m, их значения объединяются в отдельные абзацы
-F <file>, --file =<file>	Взять сообщение тега из данного файла
-e, --edit	Сообщение, взятое из файла с ключом -F либо из командной строки с ключом -m, обычно используется в качестве сообщения тега без изменений. Опция позволяет дополнительно изменять сообщения из этих источников

Формат:

$ git show [< options >] [<object >...]

Параметр	Описание
<object>...	Имена отображаемых объектов. ***По умолчанию*** это указатель HEAD
-- pretty[=<format>] **-- format=<format>**	Печатать содержимое журналов коммитов в заданном формате. Формат может быть одним из значений: oneline, short, medium, full, fuller, reference, email, raw, format:<string> и tformat:<string>
--oneline	Это сокращение от --Pretty=oneline--abbrev-commit для совместного использования

Задача 10. Организация рабочего процесса (работа с Workflow)

Workflow – это модель ветвлений, которая упрощает работу над проектом (рис. 8.2) и помогает организовать работу в команде и(или) над проектом с множеством версий продукта.

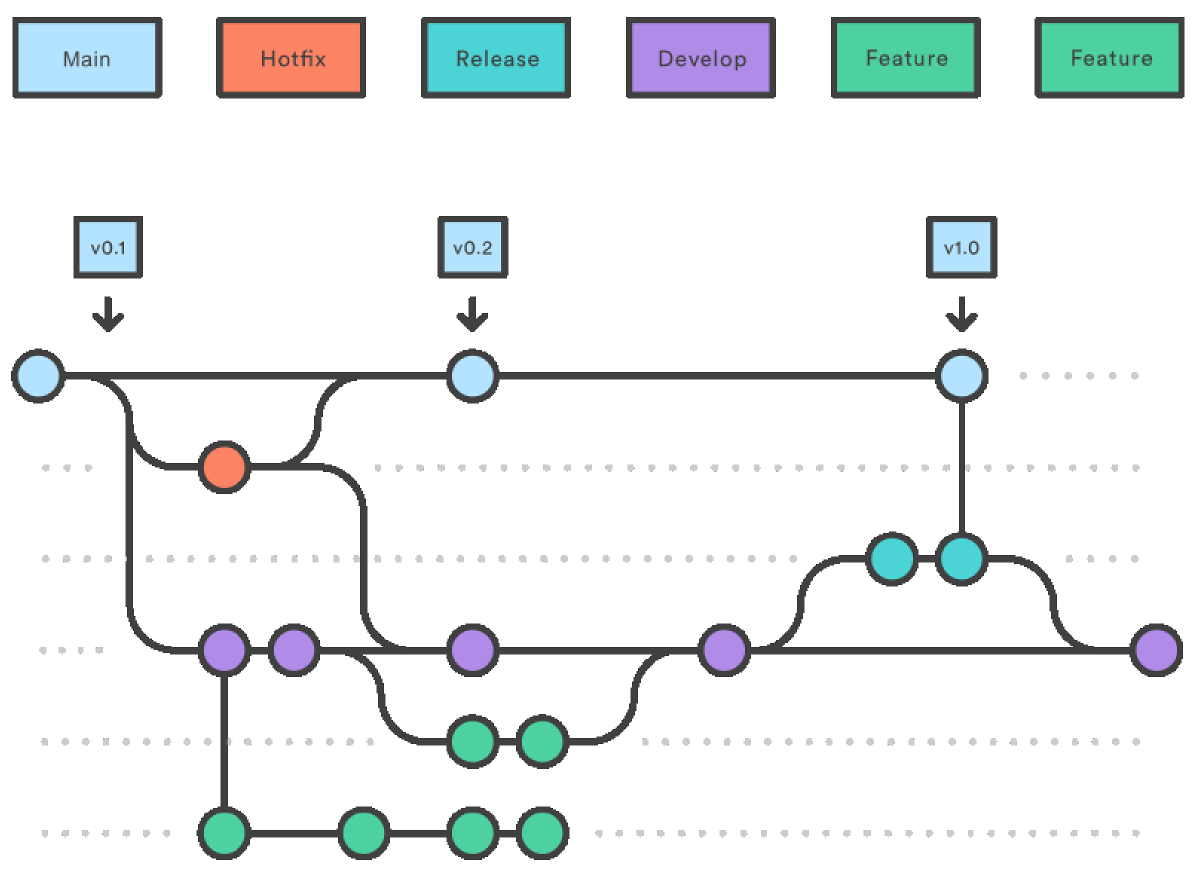


Рис. 8.2. Git Workflow

Основная задача состоит в том, чтобы организовать добавление и публикацию изменений в исходном коде проекта без сбоев в работе.

Предназначение веток (рис. 8.2):

– **Main** – основная ветка, в которой находится «готовый» продукт, т. е. стабильная версия, поставляемая конечному пользователю. На нее обычно добавляются теги с версиями;

– **Hotfix** – множество веток (название может быть Hotfix-bug12345 и т. п.), которые используются для быстрых исправлений (фиксов), каким-то образом попавших в ветку Main;

– **Develop** – текущая ветка для процесса разработки, в которой находится последний актуальный код всего проекта;

– **Feature** – множество веток (название может быть Feature-12345 и т. п.), которые создаются разработчиками для написания нового функционала и его проверки до внедрения в проект. По окончании написания ветка либо закрывается, либо выполняется слияние с веткой Develop;

– **Release** – ветка для подготовки новой версии продукта, в которой делаются поправки в базовых настройках и устраняются мелкие ошибки. Зачастую она ответвляется от Develop, а затем при успешном окончании сливается с веткой Main, также может обратно сливаться с веткой Develop.

Практическая работа №9

Разработка эксплуатационной документации

Цель: ознакомиться с видами эксплуатационной документации программного продукта; изучить нормативно-правовую документацию, регламентирующую разработку пользовательских документов; приобрести навыки разработки эксплуатационных документов.

Задание:

1. Изучить теоретические сведения о разработке эксплуатационной документации программного продукта, а также справочную информацию, представленную далее.

2. Для варианта проекта из практической работы №2 составить руководство пользователя (оператора) по следующему плану:

– для какого продукта предназначено руководство пользователя (наименование, модель);

– основные технические характеристики продукта;

– основные разделы руководства пользователя (название раздела и краткое его описание).

3. Представить результаты преподавателю и защитить работу.

4. Дополнительно составить для проекта из практической работы №2 инструкции по установке и настройке программного продукта – руководство администратора.

Справочная информация

Пользовательская документация (или эксплуатационные документы) объясняет пользователям, как они должны действовать, чтобы применить разрабатываемый программный продукт. Она необходима, если продукт предполагает какое-либо взаимодействие с пользователями.

К такой документации относятся документы, которыми должен руководствоваться потенциальный пользователь при инсталляции (установке) программного продукта, его применении для решения своих задач либо при управлении им.

К эксплуатационным документам относят:

– описание применения (сведения о назначении программы, области применения, применяемых методах, классе решаемых задач, ограничениях для применения, минимальной конфигурации технических средств);

– руководство системного программиста или руководство администратора (сведения обеспечения функционирования программы);

– руководство программиста (сведения для эксплуатации программы);

– руководство оператора (сведения для обеспечения общения оператора с вычислительной системой в процессе выполнения программы);

– описание языка (описание синтаксиса и семантики языка);

– руководство по техническому обслуживанию (сведения для применения тестовых и диагностических программ при обслуживании технических средств).

Допускается объединять отдельные виды документов.

Документ «Описание применения» состоит из следующих разделов:

– назначение программы (возможности, основные характеристики, ограничения области применения);

– условия применения (требования к техническим и программным средствам, общие характеристики входной и выходной информации, а также требования и условия организационного, технического и технологического характера);

– описание задачи (указываются определения задачи и методы ее решения);

– входные и выходные данные.

Документ «Руководство администратора» состоит из следующих разделов:

– общие сведения о программе, структура программы (составные части, связи, объем памяти и т. п.);

– сведения по установке и переносу системы на новую платформу;

– настройка программы (конкретное применение составляющих технических средств);

– необходимые эксплуатационные требования по обеспечению надежной работы продукта при заданных условиях конкретного применения;

– дополнительные возможности программы;

– сообщения (тексты сообщений, выдаваемых администратору в ходе установки или настройки программы, и описание действий, которые необходимо предпринять по этим сообщениям).

Документ «Руководство программиста» включается в программную документацию, если разработанный программный продукт требует обслуживания программистом. Документ состоит из следующих разделов:

– назначение и условия применения программы (назначение и функции программы, сведения о технических и программных средствах, обеспечивающих выполнение данной программы);

– характеристики программы (временные характеристики, режимы работы, средства контроля правильности выполнения и т. п.);

– обращение к программе (способы передачи управления и параметров данных);

– входные и выходные данные (формат и кодирование);

– сообщения (тексты сообщений, выдаваемых программисту или оператору в ходе выполнения программы, и описание действий, которые необходимо предпринять по этим сообщениям).

Документ «Руководство оператора» состоит из следующих разделов:

– назначение программы (информация, достаточная для понимания функций программы и ее эксплуатации);

– условия выполнения программы (минимальный и/или максимальный набор технических и программных средств и т. п.);

– выполнение программы (последовательность действий оператора, обеспечивающих загрузку, запуск, выполнение и завершение программы; описываются функции, форматы и возможные варианты команд, с помощью которых оператор осуществляет загрузку и управляет выполнением программы, а также ответы программы на эти команды);

– сообщения оператору (тексты сообщений, выдаваемых оператору в ходе выполнения программы, и описание действий, которые необходимо предпринять по этим сообщениям).

Эксплуатационный документ «Руководство по техническому обслуживанию» включается в программную документацию, если разработанный программный продукт требует использования тестовых (диагностических) ПС.

Разработка пользовательской документации начинается сразу после создания внешнего описания. Качество этой документации может существенно определять успех программного продукта. Она должна быть достаточно проста и удобна для пользователя (в противном случае этот продукт вообще не стоило создавать). Поэтому, хотя черновые варианты (наброски) пользовательских документов создаются основными разработчиками программного продукта, к созданию их окончательных вариантов часто привлекаются профессиональные технические писатели.

Структура и содержание документов «Руководство оператора», «Руководство программиста», «Руководство системного программиста» регламентированы ГОСТ 19.505−79, ГОСТ 19.504−79 и ГОСТ 19.503−79 соответственно. Единая система конструкторской документации (ЕСКД) определяет документ «Руководство по эксплуатации» и другие документы: ГОСТ 2.601−2013 «Эксплуатационные документы» и ГОСТ 2.610−2006 «Правила выполнения эксплуатационных документов».

ПРИЛОЖЕНИЕ 1

Правила перевода из одной системы счисления в другую

ОБЩИЙ ПРИНЦИП

Для того чтобы перевести число в систему счисления с основанием M (т. е. представлено цифрами 0, … M–1), или, как говорится, в M-ичную ССЧ, нужно представить это число в развернутой форме:

$$A = a_n \cdot M^n + a_{n-1} \cdot M^{n-1} + ... + a_1 \cdot M^1 + a_0 \cdot M^0,$$

где a_i – цифры числа из соответствующего диапазона;

a_n – первая цифра;

a_0 – последняя цифра.

Для того чтобы получить десятичное число, необходимо выполнить сложение всех составляющих развернутой формы числа в другой ССЧ.

Например, если мы имеем число $2023_{(4)}$ в четверичной ССЧ (т. е. основа числа 4), то мы имеем следующую его развернутую форму: $2 \cdot 4^3 + 0 \cdot 4^2 + 2 \cdot 4^1 + 3 \cdot 4^0$.

После поднесения к степени основания и добавления всех составляющих получим $128 + 8 + 3 = 139_{(10)}$.

ПЕРЕВОД ИЗ ДЕСЯТИЧНОЙ ССЧ В ДРУГУЮ

Для перевода из десятичной ССЧ в другую нужно выполнить следующие шаги:

1) разделить число нацело на M (основу ССЧ, в которую переводим число), остаток равен цифре a_0 – это значение младшего разряда;

2) взять долю и выполнить шаг 1, остаток будет равен a_1 и т. д.

Деление выполняют до тех пор, пока доля не станет меньше делителя, т. е. меньше основы ССЧ, в которую переводим.

Значение последней части будет старшим разрядом.

Число, которое мы ищем, будет записано в новой ССЧ полученными цифрами от частного до первого остатка.

Пример перевода чисел 26 и 11 из десятичной системы счисления в двоичную приведен на рис. П.1.1. Примеры переводов числа 95 из десятичной ССЧ к двоичной, восьмеричной и шестнадцатеричной приведены на рис. П.1.2.

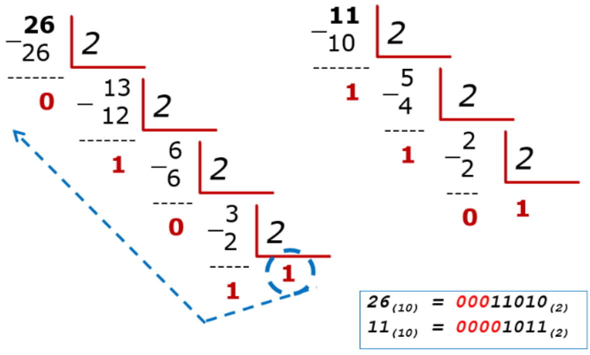


Рис. П.1.1. Шаги перевода из десятичной в двоичную ССЧ:

$\mathbf{26}_{(10)} \rightarrow X_{(2)}$, $11_{(10)} \rightarrow Y_{(2)}$

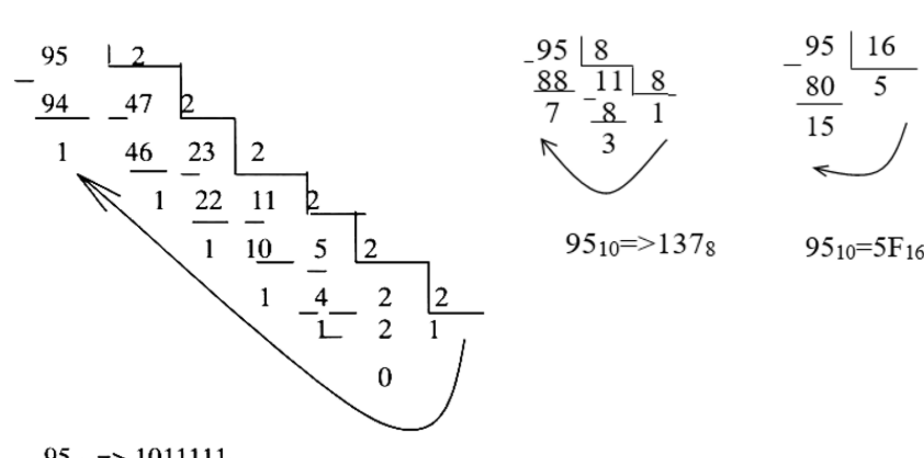


Рис. П.1.2. Шаги перевода числа 95 из десятичной в ССЧ

с основаниями 2, 8 и 16: $\mathbf{95}_{(10)} \rightarrow X_{(2)} \rightarrow Y_{(8)} \rightarrow Z_{(16)}$

Перевод правильной дроби из десятичной ССЧ в другую (М-ичную) выполняется последовательным умножением дроби на основание новой ССЧ. Умножение выполняется до тех пор, пока:

– или дробная часть станет равна нулю;

– или будет достигнута определенная точность;

– или будет обнаружен период.

Результат в новой ССЧ записывается в виде последовательности целых, полученных на каждом шагу умножения, начиная с первого и до последнего.

Пример перевода правильной дроби $0.36_{(10)}$ в другие ССЧ приводится на рис. П.1.3.

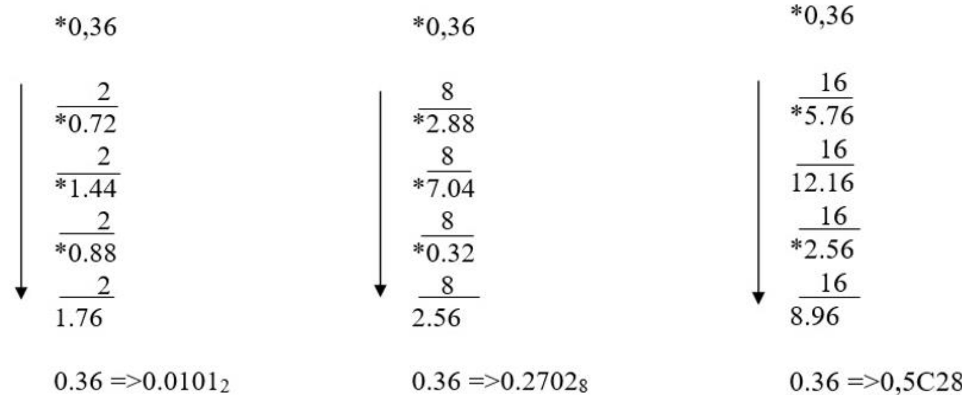


Рис. П.1.3. Шаги перевода десятичной дроби из десятичной в ССЧ с основаниями 2, 8 и 16: $\mathbf{0.36}_{(10)} \rightarrow X_{(2)} \rightarrow Y_{(8)} \rightarrow Z_{(16)}$

Перевод дроби обратно из другой (М-ичной) ССЧ к десятичной дроби происходит так же, как и перевод целой части – через развернутую форму числа. Только в случае с дробями степени основания будут отрицательными.

Пример перевода двоичного числа **1001101.1101**:

$$1001101.1101_2 = 2^6 + 2^3 + 2^2 + 2^0 + 2^{-1} + 2^{-2} + 2^{-4} = 77.8125_{10}.$$

Перевод чисел из восьмеричной ССЧ в шестнадцатеричную ССЧ, и наоборот, выполняется через двоичную ССЧ с помощью триад и тетрад. Пример перевода из шестнадцатеричной в восьмеричную ССЧ приведен на рис. П.1.4.

$F4F,88_{(16)} = 1111\ 0100\ 1111,\ 1000\ 1000_{(2)} =$

$111\ 101\ 001\ 111,\ 100\ 010_{(2)} = 7517,42_{(8)}$

Рис. П.1.4. Шаги перевода числа из шестнадцатеричного в восьмеричное

ПРИЛОЖЕНИЕ 2

Расчетные таблицы арифметических операций в различных системах счисления, используемых в компьютере

+	1	2	3	4	5	6	7
1	2	3	4	5	6	7	10
2	3	4	5	6	7	10	11
3	4	5	6	7	10	11	12
4	5	6	7	10	11	12	13
5	6	7	10	11	12	13	14
6	7	10	11	12	13	14	15
7	10	11	12	13	14	15	16

*	1	2	3	4	5	6	7
1	1	2	3	4	5	6	7
2	2	4	6	10	12	14	16
3	3	6	11	14	17	22	25
4	4	10	14	20	24	30	34
5	5	12	17	24	31	36	43
6	6	14	22	30	36	44	52
7	7	16	25	34	43	52	61

Рис. П.2.1. Таблицы для выполнения арифметических действий (сложение и умножение) в восьмеричной системе счисления

+	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10
2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11
3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12
4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13
5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14
6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15
7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16
8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17
9	A	B	C	D	E	F	10	11	12	13	14	15'	16	17	18
A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19
B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A
C	D	E	F	10	11	12	13	14	15	16	17	18	19.	1A	1B
D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C
E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D
F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E

Рис. П.2.2. Таблица для выполнения арифметических действий сложения в шестнадцатеричной системе счисления

*	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
1	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
2	2	4	6	8	A	C	E	10	12	14	16	18	1A	1C	1E
3	3	6	9	C	F	12	15	18	1B	1E	21	24	27	2A	2D
4	4	8	C	10	14	18	1C	20	24	28	2C	30	34	38	3C
5	5	A	F	14	19	1E	23	28	2D	32	37	3C	41	46	4B
6	6	C	12	18	1E	24	2A	30	36	3C	42	48	4E	54	5A
7	7	E	15	1C	23	2A	31	38	3F	46	4D	54	5B	62	69
8	8	10	18	20	28	30	38	40	48	50	58	60	68	70	78
9	9	12	1B	24	2D	36	3F	48	51	5A	63	6C	75	7E	87
A	A	14	1E	28	32	зс	46	50	5A	64	6E	78	82	8C	96
B	B	16	21	2C	37	42	4D	58	63	6E	79	84	8F	9A	A5
C	C	18	24	30	3C	48	54	60	6C	78	84	90	9C	A8	B4
D	D	1A	27	34	41	4E	5B	68	75	82	8F	9C	A9	B6	C3
E	E	1C	2A	38	46	54	62	70	7E	8C	9A	A8	B6	C4	D2
F	F	1E	2D	зс	4B	5A	69	78	87	96	A5	B4	сз	D2	E1

Рис. П.2.3. Таблицы для выполнения арифметических действий умножения в шестнадцатеричной системе счисления

ПРИЛОЖЕНИЕ 3

Типовая структура содержания технического задания на разработку программного продукта

СОДЕРЖАНИЕ

1. ОБЩИЕ СВЕДЕНИЯ
   1.1. Наименование и реквизиты заказчика
   1.2. Наименование и реквизиты исполнителя
   1.3. Основание для разработки
   1.4. Плановые сроки начала и окончания работ
   1.5. Сведения об источнике и порядке финансирования работ
   1.6. Термины и сокращения
2. НАЗНАЧЕНИЕ И ЦЕЛИ СОЗДАНИЯ СИСТЕМЫ
   2.1. Назначение системы
   2.2. Цели создания системы
3. ХАРАКТЕРИСТИКА ОБЪЕКТА АВТОМАТИЗАЦИИ
   3.1. Краткие сведения об объектах автоматизации
   3.2. Сведения об условиях эксплуатации объекта автоматизации и характеристиках окружающей среды
4. ТРЕБОВАНИЯ К СИСТЕМЕ
   4.1. Требования к системе в целом
      4.1.1. Требования к структуре и функционированию
      4.1.2. Показатели назначения системы
      4.1.3. Требования к надежности
      4.1.4. Требования по обеспечению безопасности при эксплуатации технических средств
      4.1.5. Требования к безопасности и защите информации
      4.1.6. Требования к численности и квалификации персонала
      4.1.7. Требования к эксплуатации, техническому обслуживанию, ремонту и хранению компонентов
      4.1.8. Требования к эргономике и технической эстетике
      4.1.9. Требования к патентной чистоте

4.1.10. Требования по стандартизации и унификации
4.1.11. Требования к масштабируемости и открытости
4.1.12. Номенклатура показателей качества
4.2. Функциональные требования
4.3. Обработка ошибок
4.3.1. Ошибки аутентификации
4.3.2. Ошибки загрузки данных из внешних источников
4.3.3. Внутренние ошибки
4.4. Интерфейс
4.4.1. Основные требования
4.4.2. Дизайн и юзабилити
4.4.3. Навигация
4.5. Требования к видам обеспечения
4.5.1. Требования к информационному обеспечению
4.5.2. Требования к аппаратному обеспечению
4.5.3. Требования к программному обеспечению
4.5.4. Требования к лингвистическому обеспечению
4.5.5. Требования к техническому обеспечению
4.5.6. Требования к объекту внедрения
5. СОСТАВ И СОДЕРЖАНИЕ РАБОТ ПО СОЗДАНИЮ СИСТЕМЫ
5.1. Перечень фаз по созданию системы
5.2. Перечень организаций – исполнителей работ
5.3. Гарантийное сопровождение системы
5.4. Техническая поддержка системы
6. ПОРЯДОК КОНТРОЛЯ И ПРИЕМКИ СИСТЕМЫ
7. ТРЕБОВАНИЯ К ДОКУМЕНТИРОВАНИЮ

ПРИЛОЖЕНИЕ 4

Шаблон учебного технического задания на разработку программного продукта

ТЕХНИЧЕСКОЕ ЗАДАНИЕ

на разработку программного продукта

«_______»

РАЗРАБОТЧИК

____

ФИО

«____»_______________ 20__ г.

СОГЛАСОВАНО

Наименование организации, предприятия	*Должность исполнителя*	*Фамилия, имя, отчество*	*Дата изменения*

Город – год

СОДЕРЖАНИЕ

1. ОБЩИЕ СВЕДЕНИЯ ....

1.1. Наименование программного продукта ....

1.2. Назначение и область применения ....

1.3. Целевая аудитория продукта – пользователи ....

2. ОБЩИЕ ТРЕБОВАНИЯ К ПРОГРАММНОМУ ПРОДУКТУ ....

2.1. Требования к функциональным характеристикам ....

2.2. Взаимосвязь программы с другими информационными системами ....

2.3. Требования к лингвистическому обеспечению ....

3. СИСТЕМНЫЕ ТРЕБОВАНИЯ (УСЛОВИЯ ЭКСПЛУАТАЦИИ) ....

3.1. Требования к составу и параметрам технических средств ....

3.2. Требования к информационной и программной совместимости ....

3.2.1. Требования к информационным структурам и методам решения ..

3.2.2. Требования к исходным кодам и языкам программирования ....

3.2.3. Требования к программным средствам, используемым программой ....

3.2.4. Требования к защите информации и программ ....

4. ТРЕБОВАНИЯ К ИНТЕРФЕЙСУ ....

4.1. Требования к внешнему интерфейсу ....

4.2. Графический интерфейс пользователя ....

1. ОБЩИЕ СВЕДЕНИЯ

1.1. Наименование программного продукта

Например:

Наименование программы: «Первая программа»

1.2. Назначение и область применения

Здесь должно быть указано функциональное и эксплуатационное назначение программы или программного изделия. Здесь можно ограничиться одной-двумя фразами. Главное – четко определить, для чего нужна программа.

Программа предназначена для…

Здесь указывают краткую характеристику области применения программы или программного изделия и объекта, в котором используют программу или программное изделие.

Программа может быть использована для…

1.3. Целевая аудитория продукта – пользователи

Здесь определяют различные классы пользователей, которые, как предполагается, будут работать с вашим продуктом, и приводят их соответствующие характеристики. Некоторые требования могут относиться только к определенным классам пользователей.

Например:

Программа предназначена для использования студентами вузов.

или

Программа предполагает начальный уровень компьютерной подготовки пользователя.

2. ОБЩИЕ ТРЕБОВАНИЯ К ПРОГРАММНОМУ ПРОДУКТУ

2.1. Требования к функциональным характеристикам

Здесь указывают требования к составу выполняемых функций.

Например:

Программа должна позволять…, вычислять…, строить…, создавать….

или

Программа должна обеспечивать возможность выполнения перечисленных ниже функций:

– … (перечислить функциональные возможности).

Также здесь указывают требования к организации входных и выходных данных, временные характеристики и т. п.

Например:

Исходные данные: текстовый файл с заданной…

Выходные данные: графическая и текстовая информация – результаты анализа системы <название>; текстовые файлы – отчеты о…, диагностика состояния системы и сообщения о всех возникших ошибках.

2.2. Взаимосвязь программы с другими информационными системами

Здесь указывают, с какими программами/системами работает/интегрируется ПС и каким образом.

Например:

Программа предполагает взаимодействие с… путем….

или

Связи со сторонними программами не предполагается.

2.3. Требования к лингвистическому обеспечению.

Здесь указывают требования к языку интерфейса ПС, справочной системы и т. п.

Например:

Программа должна поддерживать работу на китайском языке.

Весь интерфейс программы предполагает работу на нескольких языках, что обеспечивается за счет…

3. СИСТЕМНЫЕ ТРЕБОВАНИЯ (УСЛОВИЯ ЭКСПЛУАТАЦИИ)

3.1. Требования к составу и параметрам технических средств

Здесь указывают необходимый состав технических средств и их характеристики.

Например:

В состав технических средств должен входить персональный компьютер со следующими характеристиками:

процессор, не менее…;

оперативную память, объемом не менее … Гбайт;

HDD, дисковое пространство не менее … Гбайт;

графический адаптер…;

манипулятор типа «мышь»;

манипулятор типа «джойстик» и т. д.

3.2. Требования к информационной и программной совместимости

3.2.1. Требования к информационным структурам и методам решения

Здесь указывают требования к информационным структурам на входе и выходе программы, а также методам решения.

Например:

Для стабильной работы программы необходимо обеспечить структуру входных данных в виде…

или

Программа предполагает использование заданного формата обмена данными, например XML.

или

Программа должна работать с заданной расширенной матрицей инциденций исследуемого графа в соответствии с алгоритмом функционирования. Описание матрицы…

3.2.2. Требования к исходным кодам и языкам программирования

Здесь приводят определенные технологии, средства и языки программирования, которые следует использовать или избегать. Необходимо указать правила оформления исходного кода – от организации до комментариев.

Например:

Базовый язык программирования – …

Предполагается использование компонентной технологии и возможностей динамически подключаемых библиотек.

3.2.3. Требования к программным средствам, используемым программой

Здесь указывают рабочую среду программного обеспечения, включая операционные системы и их версии. Перечисляют все остальные компоненты программного обеспечения или приложений, с которыми система должна быть совместима.

Например:

Программа должна работать автономно под управлением ОС… версии не ниже…

Системные программные средства, используемые программой, должны быть представлены…

Система должна функционировать на базе платформы…

Программа должна быть построена по архитектуре…

3.2.4. Требования к защите информации и программ

Здесь необходимо указать требования к сохранности данных, а также реакцию программы на возможные некорректные действия оператора/пользователя.

Например:

Программа должна выдавать сообщения об ошибках при неверно заданных исходных данных, поддерживать диалоговый режим в рамках предоставляемых пользователю возможностей.

4. ТРЕБОВАНИЯ К ИНТЕРФЕЙСУ

4.1. Требования к внешнему интерфейсу

Здесь приводят требования, которые определяют оборудование, программное обеспечение или элементы баз данных, с которыми программа или ее компонент должны взаимодействовать. Информация этого раздела позволяет вам быть уверенным, что программа будет должным образом взаимодействовать с внешними компонентами. Если у разных частей продукта разные внешние интерфейсы, вставьте подобный подраздел в детализированные требования для каждой такой части.

4.2. Графический интерфейс пользователя[1]

Здесь указывают логические характеристики каждого пользовательского интерфейса, который необходим программе, каждой экранной формы.

Здесь можно указать:

– *ссылки на стандарты графического интерфейса пользователей или стилевые рекомендации для семейства продукта, которые необходимо соблюдать;*

– *стандарты шрифтов, значков, названий кнопок, изображений, цветовых схем, последовательностей полей вкладок, часто используемых элементов управления и т. п.;*

– *конфигурацию экрана или ограничения разрешения;*

– *стандартные кнопки, функции или ссылки перемещения, одинаковые для всех экранов, например, кнопка справки;*

– *быстрые клавиши;*

– *стандарты отображения сообщений;*

– *стандарты конфигурации для упрощения локализации программного обеспечения;*

– *специальные возможности для пользователей с проблемами со зрением.*

[1] Следует подробно описать все ограничения интерфейса и обязательно вставить в эскизы экранных форм (или прикрепить их как дополнительное приложение к ТЗ).

ПРИЛОЖЕНИЕ 5

Основные сведения о схемах алгоритмов по стандарту ГОСТ 19.701–90

1. Общие требования

1.1. Схемы алгоритмов, программ, данных и систем (далее – схемы) состоят из имеющих заданное значение символов, краткого пояснительного текста и соединяющих линий.

1.2. Схемы могут использоваться на различных уровнях детализации, причем число уровней зависит от размеров и сложности задачи обработки данных. Уровень детализации должен быть таким, чтобы различные части и взаимосвязь между ними были понятны в целом.

1.3. В настоящем стандарте определены символы, предназначенные для использования в документации по обработке данных, и приведено руководство по условным обозначениям для применения их в следующих схемах:

1) данных;

2) программ;

3) работы системы;

4) взаимодействия программ;

5) ресурсов системы.

1.4. В стандарте используются следующие понятия:

1) основной символ – символ, используемый в тех случаях, когда точный тип (вид) процесса или носителя данных неизвестен или отсутствует необходимость в описании фактического носителя данных;

2) специфический символ – символ, используемый в тех случаях, когда известен точный тип (вид) процесса или носителя данных или когда необходимо описать фактический носитель данных;

3) схема – графическое представление определения, анализа или метода решения задачи, в котором используются символы для отображения операций, данных, потока, оборудования и т. д.

2. Описание схем

2.1. Схема данных

2.1.1. Схемы данных отображают путь данных при решении задач и определяют этапы обработки, а также различные применяемые носители данных.

2.1.2. Схема данных состоит:

1) из символов данных (символы данных могут также указывать вид носителя данных);

2) символов процесса, который следует выполнить над данными (символы процесса могут также указывать функции, выполняемые вычислительной машиной);

3) символов линий, указывающих потоки данных между процессами и (или) носителями данных;

4) специальных символов, используемых для облегчения написания и чтения схемы.

2.1.3. Символы данных предшествуют и следуют за символами процесса. Схема данных начинается и заканчивается символами данных.

2.2. Схема программы

2.2.1. Схемы программ отображают последовательность операций в программе.

2.2.2. Схема программы состоит:

1) из символов процесса, указывающих фактические операции обработки данных (включая символы, определяющие путь, которого следует придерживаться с учетом логических условий);

2) линейных символов, указывающих поток управления;

3) специальных символов, используемых для облегчения написания и чтения схемы.

2.3. Схема работы системы

2.3.1. Схемы работы системы отображают управление операциями и поток данных в системе.

2.3.2. Схема работы системы состоит:

1) из символов данных, указывающих на наличие данных (символы данных могут также указывать вид носителя данных);

2) символов процесса, указывающих операции, которые следует выполнить над данными, а также определяющих логический путь, которого следует придерживаться;

3) линейных символов, указывающих потоки данных между процессами и (или) носителями данных, а также поток управления между процессами;

4) специальных символов, используемых для облегчения написания и чтения блок-схемы.

2.4. Схема взаимодействия программ

2.4.1. Схемы взаимодействия программ отображают путь активаций программ и взаимодействий с соответствующими данными. Каждая программа в схеме взаимодействия программ показывается только один раз (в схеме работы системы программа может изображаться более чем в одном потоке управления).

2.4.2. Схема взаимодействия программ состоит:

1) из символов данных, указывающих на наличие данных;

2) символов процесса, указывающих на операции, которые следует выполнить над данными;

3) линейных символов, отображающих поток между процессами и данными, а также инициации процессов;

4) специальных символов, используемых для облегчения написания и чтения схемы.

2.5. Схема ресурсов системы

2.5.1. Схемы ресурсов системы отображают конфигурацию блоков данных и обрабатывающих блоков, которая требуется для решения задачи или набора задач.

2.5.2. Схема ресурсов системы состоит:

1) из символов данных, отображающих входные, выходные и запоминающие устройства вычислительной машины;

2) символов процесса, отображающих процессоры (центральные процессоры, каналы и т. д.);

3) линейных символов, отображающих передачу данных между устройствами ввода-вывода и процессорами, а также передачу управления между процессорами;

4) специальных символов, используемых для облегчения написания и чтения схемы.

3. Описание символов (табл. П.5.1)

Таблица П.5.1

Описание символов

Наименование	Характеристика	Графическое обозначение
1	**2**	**3**
Символы данных		
Основные символы данных		
Данные	Символ отображает данные, носитель данных не определен. Используется одинаково как для входных, так и для выходных данных (ввод / вывод информации)	
Запоминаемые данные	Символ отображает хранимые данные в виде, пригодном для обработки, носитель данных не определен	
Специфические символы данных		
Оперативное запоминающее устройство	Символ отображает данные, хранящиеся в оперативном запоминающем устройстве	

Продолжение табл. П.5.1

1	2	3
Запоминающее устройство с последовательным доступом	Символ отображает данные, хранящиеся в запоминающем устройстве с последовательным доступом (магнитная лента, кассета с магнитной лентой, магнитофонная кассета)	
Запоминающее устройство с прямым доступом	Символ отображает данные, хранящиеся в запоминающем устройстве с прямым доступом (магнитный диск, магнитный барабан, гибкий магнитный диск)	
Документ	Символ отображает данные, представленные на носителе в удобочитаемой форме (документ для оптического или магнитного считывания, бланки ввода данных и т. п.)	
Ручной ввод	Символ отображает данные, вводимые вручную во время обработки с устройств любого типа (клавиатура, переключатели, кнопки, световое перо, полоски со штриховым кодом)	
Карта	Символ отображает данные, представленные на носителе в виде карты (карты со считываемыми метками, карты с отрывным ярлыком, карты со сканируемыми метками)	
Бумажная лента	Символ отображает данные, представленные на носителе в виде бумажной ленты	
Дисплей	Символ отображает данные, представленные в удобной для человека читаемой форме на носителе в виде отображающего устройства (экран для визуального наблюдения, индикаторы ввода информации)	

Продолжение табл. П.5.1

1	2	3
Символы процесса		
Основной символ процесса		
Процесс	Символ отображает функцию обработки данных любого вида (выполнение определенной операции или группы операций, приводящее к изменению значения, формы или размещения информации или к определению, по которому из нескольких направлений потока следует двигаться)	
Специфические символы процесса		
Предопределенный процесс	Символ отображает предопределенный процесс, состоящий из одной или нескольких операций или шагов программы, которые определены в другом месте (в подпрограмме, модуле)	
Ручная операция	Символ отображает любой процесс, выполняемый человеком	
Подготовка	Символ отображает модификацию команды или группы команд с целью воздействия на некоторую последующую функцию (установка переключателя, модификация индексного регистра или инициализация программы). Обычно используется для отображения цикла с параметром	

Окончание табл. П.5.1

1	2	3
Решение	Символ отображает решение или функцию переключательного типа, имеющую один вход и ряд альтернативных выходов, один и только один из которых может быть активизирован после вычисления условий, определенных внутри этого символа. Соответствующие результаты вычисления могут быть записаны по соседству с линиями, отображающими эти пути	
Параллельные действия	Символ отображает синхронизацию двух или более параллельных операций	

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Орлов, С. А. Программная инженерия : учебник / С. А. Орлов. – 5-е изд., обновл. и доп. – СПб. : Питер, 2017. – 640 с.

2. Вигерс, К. Разработка требований к программному обеспечению / К. Вигерс, Д. Битти. – 3-е изд., доп. – СПб. : BHV, 2019. – 736 с.

3. Халл, Э. Инженерия требований / Э. Халл, К. Джексон, Д. Дик. – М. : ДМК Пресс, 2017. – 218 с.

4. Лаврищева, Е. М. Программная инженерия и технологии программирования сложных систем : учебник для вузов / Е. М. Лаврищева. – 2-е изд., испр. и доп. – М. : Юрайт, 2019. – 432 с.

5. Батоврин, В. К. Толковый словарь по системной и программной инженерии / В. К. Батоврин. – М. : ДМК Пресс, 2012. – 280 с.

6. Гецци, К. Основы инженерии программного обеспечения / К. Гецци, М. Джазайери, Д. Мандртоли. – 2-е изд. – СПб. : БХВ-Петербург, 2005. – 832 с.

7. Sommerville, Ian. Software Engineering / Ian Sommerville. – 9th Edition, Addison-Wesley, 2011. − 773 p.

8. Брауде, Э. Технология разработки программного обеспечения / Э. Брауде. – СПб. : Питер, 2004. – 655 с.

9. Гагарина, Л. Г. Технология разработки программного обеспечения : учеб. пособие / Л. Г. Гагарина, Е. В. Кокорева, Б. Д. Виснадул. – М. : Форум, ИНФРА-М, 2008. – 400 с.

10. Фатрелл, Р. Т. Управление программными проектами. Достижение оптимального качества при минимуме затрат / Р. Т. Фатрелл, Д. Ф. Шафер, Л. И. Шафер. – М. : Вильямс, 2004. – 1136 с.

11. Кагарлицкий, Ю. В. Разработка документации пользователя программного продукта. Методика и стиль изложения / Ю. В. Кагарлицкий. – 2-е изд. – М. : Философт, 2012. – 232 с.

12. Леффингуэлл, Д. Принципы работы с требованиями к программному обеспечению. Унифицированный подход / Д. Леффингуэлл, Д. Уидриг ; пер. с англ. – М. : Вильямс, 2002. – 448 с.

13. Орам, Э. Идеальная разработка ПО. Рецепты лучших программистов / Э. Орам, Г. Уилсон ; пер. с англ. – СПб. : Питер, 2012. – 592 с.

14. Интеграция управления программой и системной инженерии / под ред. Эрика С. Ребентиша ; пер. с англ. В. К. Батоврина, Е. В. Батовриной, А. А. Ефремова ; под ред. В. К. Батоврина. – М. : ДМК Пресс, 2020. – 584 с.

15. Системная инженерия. Принципы и практика / А. Косяков [и др.]. – 2-е изд. – М. : ДМК Пресс, 2017. – 624 с.

16. Липаев, В. В. Процессы и стандарты жизненного цикла сложных программных средств: справочник / В. В. Липаев. – М. : Синтег, 2006. – 276 с.

17. Липаев, В. В. Документирование сложных программных средств / В. В. Липаев. – М. : Синтег, 2005. – 124.с.

18. Липаев, В. В. Программная инженерия. Методологические основы / В. В. Липаев. – М. : ТЕИС, 2006. – 608 с.

19. Мацяшек, Л. А. Практическая программная инженерия на основе учебного примера / Л. А. Мацяшек. – М. : БИНОМ, 2009. – 956 с.

20. Брукс, Ф. Мифический человеко-месяц, или Как создаются программные системы / Ф. Брукс. – СПб. : Питер, 2021. – 368 с.

21. Pro Git book / Scott Chacon, Ben Straub [Электронный ресурс]. – Режим доступа : https://git-scm.com/book/en/v2. – Дата доступа : 10.10.2022.

22. About TortoiseSVN [Электронный ресурс]. – Режим доступа : https://tortoisesvn.net/about.html. – Дата доступа : 10.10.2022.

23. CVS – Concurrent Versions System [Электронный ресурс]. – Режим доступа : https://cvs.nongnu.org. – Дата доступа : 10.10.2022.

24. Redmine guide [Электронный ресурс]. – Режим доступа : https://www.redmine.org/guide. – Дата доступа : 10.10.2022.

25. About Bug–Tracking System Bugzilla [Электронный ресурс]. – Режим доступа: https://www.bugzilla.org/about. – Дата доступа : 10.10.2022.

26. ГОСТ 19.106–78. Единая система программной документации. Требования к программным документам, выполненным печатным способом. – М. : Стандартинформ, 2010.

27. ГОСТ 34.602–89. Информационная технология. Комплекс стандартов на автоматизированные системы. Техническое задание на создание автоматизированной системы. – М. : Стандартинформ, 2009.

28. ГОСТ 19.701–90. Схемы алгоритмов, данных, программ и систем. Условные обозначения и правила выполнения. – М. : Изд. стандартов, 1992.

29. ГОСТ 34.201–89. Виды, комплектность и обозначение документов при создании автоматизированных систем. – М. : Изд. стандартов, 1989.

30. IEEE Std 830–1998. Recommended Practice for Software Requirements Specification. – USA: The Institute of Electrical and Electronics Engineers, 1998.

31. ISO/IEC 15288:2008. Standard for systems and software engineering life cycle processes [Электронный ресурс]. – Режим доступа : http://www.15288.com. – Дата доступа : 10.10.2022.

32. Computer Society of the Institute for Electrical and Electronic Engineers (IEEE) [Электронный ресурс]. – Режим доступа : http://www.ieee.org. – Дата доступа : 10.10.2022.

33. International Organization for Standardization (ISO). International Electrotechnical Commission (IEC). Joint Technical Committee 1, Information technology (JTC 1) [Электронный ресурс]. – Режим доступа : https://www.iso.org/committee/45020.html. – Дата доступа : 10.10.2022.

34. Association of Computer Machinery (ACM) [Электронный ресурс]. – Режим доступа : https://www.acm.org. – Дата доступа : 10.10.2022.

Св. план 2023, поз. 37

Учебное издание

Парамонов Антон Иванович
Лапицкая Наталья Владимировна
Нестеренков Сергей Николаевич

ОСНОВЫ ПРОГРАММНОЙ ИНЖЕНЕРИИ

УЧЕБНО-МЕТОДИЧЕСКОЕ ПОСОБИЕ

Редактор *Е. С. Юрец*
Корректор *Е. Н. Батурчик*
Компьютерная правка, оригинал-макет *А. А. Лущикова*

Подписано в печать 30.08.2023. Формат 60×84 1/16. Бумага офсетная. Гарнитура «Таймс».
Отпечатано на ризографе. Усл. печ. л. 7,32. Уч.-изд. л. 8,0. Тираж 100 экз. Заказ 220.

Издатель и полиграфическое исполнение: учреждение образования
«Белорусский государственный университет информатики и радиоэлектроники».
Свидетельство о государственной регистрации издателя, изготовителя,
распространителя печатных изданий №1/238 от 24.03.2014,
№2/113 от 07.04.2014, №3/615 от 07.04.2014.
Ул. П. Бровки, 6, 220013, г. Минск