МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
КЫРГЫЗСКОЙ РЕСПУБЛИКИ
КЫРГЫЗСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ
УНИВЕРСИТЕТ им. И. РАЗЗАКОВА
БИШКЕКСКИЙ ТЕХНИЧЕСКИЙ КОЛЛЕДЖ

«Согласовано»
«01» сентя 2023 год.

«Утверждаю»
Зам.директора по УР
«__» _____ 2023 год.

Учебно-методической комплекс

На **2023-2028** учебного года

По предмету: Технология разработки программных продуктов

Для специальности: Программное обеспечение

Рассмотрено и одобрена на заседании ПЦК СД: Батырбекова Д.

Протокол №1 от «30» 08 2023 года.

Преподаватель БТК ***Уркунбаевой А.К.***

Бишкек 2023

РЕЦЕНЗИЯ
НА УЧЕБНО-МЕТОДИЧЕСКИЙ КОМПЛЕКС ДИСЦИПЛИНЫ
“Технология разработки программных продуктов”

Учебно-методический комплекс по дисциплине “Технология разработки программных продуктов” разработан для обеспечения выполнений требования государственного образовательного стандарта СПО КР для общеобразовательного цикла типовых учебных планов к минимум содержания и подготовки специалистов по всем специальностям.

Учебно-методический комплекс включает в себя следующие элементы:

- Рабочую программу учебной дисциплины;
- Календарно-тематический план;
- Лекционные материалы;
- Задания для текущего контроля знаний студентов;
- Методические рекомендации по планированию организации и проведению практических занятий;

Рабочая программа составлена логично и носит упорядочнный, модульный подход к изучаемым разделем “Технология разработки программных продуктов”. Последовательность тем, предлагаемых к изучению, направлена на качественное усвоение учебного материала. Календарно-тематический план соответствует по своему содержанию рабочей программе по дисциплине. Методическое рекомендации по выполнению практических заданий позволяют углубить полученные теоритическое знания и привить опыт практического применения.

Методические рекомендации по организации самостоятельной работы направлены на закрепление умения поиска, накопления и обработки научной информации. Слайд-сопровождение теоритического материала нуждается в расширении.

Представленный учебно-методический комплекс дисциплины “Технология разработки программных продуктов” имеет практическую направленность, включает достаточное количество разнообразных элементов, направленных на развитие умственных, творческих способностей студентов, приобретение общенаучных, инструментальных, социально-личностных и общекультурных компетенций.

Рекомендуется ввести лабораторные занятие по дисциплине и разделить по сложности индивидуальные задания, расширить количество тестовых контрольных заданий.

Рецензент
«Кафедрой ИСТ
им.А.Жайнакова»
старший преподаватель Бакиева Ж.З.

Содержание

1. Паспорт рабочей программы учебной дисциплины «Технология разработки программного продукта»

1.1 Область применения программы

Рабочая программа учебной дисциплины «Технология разработки программного продукта» является частью основной профессиональной образовательной программы в соответствии с ГОС СПО по специальности 230109 «Программное обеспечение вычислительной техники и автоматизированных систем», утвержденный приказом Министерства образования и науки КР №567/1 от 15.05.2019 года (регистрационный № 180 от 07.06.2019 года МЮ КР)

Изучение «Технологии разработки программного продукта» направлено на достижение следующих ***целей:***

1.2. Цели учебной дисциплины

Целью данного курса является изучение среды программирования Java, которая является средой разработки, используемой для поддержки и разработки приложений, предназначенных как для рабочих станций, так и для серверов.

1.3. Место дисциплины в структуре основной профессиональной образовательной программы:

Этот курс является составной частью профессионального цикла, определяющих подготовку студентов в области современных информационных технологий.

Задачи учебной дисциплины: для достижения цели студент должен

знать:

- основные этапы технологии проектирования программных продуктов;
- приемы оптимизации программ;
- особенности модульного программирование;
- инструментальные средства разработки программа
- принципы и методы коллективной разработки программных средств;

уметь:

- разрабатывать алгоритм программный реализации поставленной задачи;
- создавать программный продукт по разработанному алгоритму;
- выполнять отладку и тестирование программного продукта;

владеть:

- навыками выбора, проектирование и реализации алгоритмов для решения профессиональных задач;
- современными технологиями и средствами проектирования, тестирование программных приложений.

1.4 Перечень формируемых компетенций

При освоении основной профессиональной образовательной программы студента должен овладеть следующими компетенциями

Общие:

ОК 1. Уметь организовать собственную деятельность, выбирать методы и способы выполнения задач, оценивать их эффективности и качество;

ОК 2. Решать проблемы, принимать решения стандартных и нестандартных ситуациях, проявлять инициативу и ответственность;

ОК 3. Осуществлять поиск, интерпретацию и использование информации, необходимой для эффективного выполнения профессиональных задач, профессионального и личностного развития;

ОК 4. Использовать информационно-коммуникационные технологии в профессиональной деятельности;

ОК 5. Уметь работать в команде, эффективно общаться с коллегами, руководством и клиентами;

ОК 6. Брать ответственность за работу членов команды(подчиненных) и их обучение на рабочем месте, за результат выполнение задания;

ОК 7. Управлять собственным личностным и профессиональным развитием, адаптироваться к изменениям условий труда и технологий в профессиональный деятельности;

ОК 8. Быть готовым к организационно-управленческий работе с малыми коллективами;

ОК 9. Быть способным анализировать и оценивать социально- экономические и культурные последствия новых явлений в науке, технике и технологии, профессиональной сфере;

ОК 10. Быть способным на научной основе оценивать свой труд; оценивать большой степенью самостоятельности результаты своей деятельности;

Профессиональные:

ПК 1. Выполнять разработку спецификацией отдельных компонентов;

ПК 2. Осуществлять разработку кода программного продукта на основе готовых спецификаций на уровне модуля;

ПК 3. Выполнять отладку программных модулей с использованием специализированных программных средств;

ПК 4. Выполнять тестирование программных модулей;

ПК 5. Осуществлять оптимизацию программного кода модуля;

ПК 6. Разрабатывать компоненты проектной и технической документации с использованием графических языков спецификации.

1.5 Рекомендуемое количество часов на основе рабочей программы учебной дисциплины:

Максимальная учебная нагрузка студентов 180 часов в том числе:

- Обязательная аудиторная учебная нагрузка -108 часов;
- Самостоятельная работа студентов – 72 часов;
- Курсовая работа -18 часов;

2. Структура и содержание учебной дисциплины

2.1. Объем учебной дисциплины и виды учебной работы

Вид учебной работы	Объем часов
Максимальная учебная нагрузка (всего)	180
Обязательная аудиторная нагрузка (всего)	108
В том числе:	
Теоретические занятия	30
Практические занятия	60
Курсовая работа	18
Самостоятельная работа студента	72
Итоговая аттестация в форме экзамена	

2.2. Примерный тематический план и содержание учебной дисциплины

№	Наименование разделов и тем	Ауд. 108 ч	СРС 72 ч	В том числе:			Уровень освоения
				Тео. 30	ЛПЗ 60	КП 18	
1	2	3		4	5	6	7
Раздел I. Общие принципы разработки программных продуктов							
1.1	Введение. Программные продукты и их основные характеристики	2		2			2
1.2	Классификация программных продуктов	4		2	2		2
1.3	Особенности создания программного продукта	4	2	2	2		2
1.4	Жизненный цикл программы	4	2	2	2		2
1.5	Проектирование программных продуктов	4	2	2	2		2
1.6	Структура и формат статические и динамические данные	2	2	2			2
	Итого по разделу:	**20**	**8**	**12**	**8**		
Раздел 2. Методология проектирования программных продуктов							
2.1	Методы проектирования ПП	4	2	2	2		2
2.2	Структура ПП	4	2	2	2		2
2.3	Проектирование интерфейса пользователя	4	2	2	2		2
	Итого по разделу:	**12**	**6**	**6**	**6**		
Раздел 3. Разработка программных продуктов							
3.1	Стиль программирования	4	2	2	2		2
3.2	Языки программирования	2	2		2		2

3.3	Модульное программирование	2	2		2		2
3.4	Структурное программирование	4	2	2	2		2
3.5	Объектно-ориентированное программирование	4	2		4		2
3.6	Эффектность и оптимизация программ	4	2	2	2		2
3.7	Обеспечение качества программного продукта	4	2	2	2		2
	Итого по разделу:	**24**	**14**	**8**	**16**		
Раздел 4. Отладка, тестирование и сопровождение программ							
4.1	Ошибки программного обеспечения	4	4		4		2
4.2	Отладка программ	4	4		4		2
4.3	Тестирование программ	4	6		4		2
4.4	Сопровождение программ	4	2		4		2
4.5	Защита программ	4	2		4		2
4.6	Пакеты прикладных программ	6			6		2
	Итого по разделу:	**26**	**18**		**26**		
Раздел 5. Инструментальные средства разработки программ							
5.1	Общая характеристика инструментальных средств разработки программ	2	2		4		2
5.2	Применение CASE-средств	2	2		2		2
	Итого по разделу:	**4**	**4**		**6**		
Раздел 6. Коллективная разработка программных средств							
6.1	Организация работ при коллективной разработке программных продуктов	2		2			2
6.2	Экономические аспекты создания программных средств	2		2			2
	Итого по разделу:	**4**		**4**			
	Раздел 7. Курсовая работа						
7.1	Курсовая работа	18	18			18	18
	Итого по разделу:	**18**	**18**			**18**	
	Итого по предмету	**108**	**72**	**30**	**60**	**18**	

Для характеристики уровня освоения учебного материала используется следующие обозначения:

1. Ознакомительный (узнавание ранее изученных объектов, свойств);
2. Репродуктивный (выполнение деятельности по образцу инструкции или под руководством);
3. Продуктивный (планирование и самостоятельное выполнение деятельности решение проблемных задач).

СОДЕРЖАНИЕ ДИСЦИПЛИНЫ

Раздел I. Введение. Общие принципы разработки программных продуктов

Тема 1.1 Программные продукты и их основные характеристики (2 ч)

Основные понятия программного обеспечения. Программа, программное обеспечение, задачи и приложения. Технологические и функциональные задачи. Процесс создания программ: постановка задачи, алгоритмизация, программирование. Понятие программного продукта. Характеристика программного продукта и его специфика. Показатели качества программного продукта: мобильность, надежность, эффективность, легкость применения, модифицируемость, коммуникативность.

Тема 1.2 Классификация программных продуктов (2 ч)

Классификация программных продуктов по сфере использования: системное программное обеспечения, инструментарий технологий программирования, пакеты прикладных программ. Состав и назначение инструментария технологий программирования. Средства для создания приложений. CASE-технологий. Программные продукты для создания приложений. Пакеты прикладных программ. Характеристика классов пакетов прикладных программ.

Практическая работа №1 (2 ч)

1. Основные понятия программного обеспечения
2. Классификация программных продуктов
3. Системное программное обеспечение
4. Пакеты прикладных программ
5. Инструментарий технологии программирования

1.3. Особенности создания программного продукта (2 ч)

Технология программирования. Программа. Программное обеспечение(software). Приложение (application). Постановка задачи (problem definition). Алгоритм.

Практическая работа №2 (2 ч)

Управление требованиями

- постановка задачи
- выбор критериев эффективности
- проведение предварительных научно-исследовательских работ;
- разработка ТЗ.

Эскизный проект:

- структура входных и выходных данных;
- уточнение методов решения; общий алгоритм;
- разработка документации эскизного проекта.

Технический проект:

- уточнение структуры входных и выходных данных;

• разработка алгоритмов;

• формы данных;

• семантика и синтаксис языка;

• структура программы; конфигурация технических средств;

• план работ.

Рабочий проект:

• программирование и отладка;

• разработка документов;

• подготовка и проведение испытаний;

• корректировка программы и документов по итогам испытаний.

Внедрение:

• передача программы и документов для сопровождения;

• оформление акта;

• передача в Фонд алгоритмов и программ

Тема 1.4 Жизненный цикл программ (2 ч)

Понятие жизненного цикла программы и его этапы. Анализ требований к программе, определения спецификации программы, проектирование, кодирование и тестирование, эксплуатация и сопровождения программы. Характеристики этапов жизненного цикла программного продукта и его специфика. Особенности разработки программного продукта.

Практическая работа №3 (2 ч)

1. Жизненный цикл разработки ПО
2. Сбор и анализ требований (Planning and Requirement Analysis)
3. Документирование требований (Defining Requirements)
4. Дизайн (Design the Product Architecture)
5. Разработка ПО (Building or Developing the Product)
6. Тестирование (Testing the Product)
7. Внедрение и поддержка продукта (Deployment in the Market and Maintenance)

Тема 1.5 Проектирование программных продуктов (2 ч)

Технологический процесс разработки программного обеспечения. Стадии разработки программ и программной документации. Сопровождаемая документация. Основные требования к содержанию документации. Правила написания технологического задания к разрабатываем программным продуктом. Техническое задание и требование к его содержанию. Эскизный и технический проекты. Рабочий проект. Внедрение.

Практическая работа №4 (2 ч)

1. Описание: совместная работа заказчика (говорит о пользе продукта, требованиях к работоспособности и внешнему виду) и EDISON (предлагает технические и алгоритмические решения).

2. Архитектура: утверждается язык программирования, база данных, серверы и фреймворки.
3. Техническое задание: составляется архитектором на основании описания и ответов заказчика на вопросы, согласовывается с менеджером проекта, затем передается клиенту, производятся правки.
4. Макеты (добавляются к техзаданию): интерфейсов, принципиальные схемы устройства, диаграммы структуры базы данных, схемы взаимодействия компонентов.
5. Контроль: архитектор устраняет замечания менеджера проектов.
6. Утверждение: заказчик проверяет и меняет ТЗ самостоятельно или сообщает список правок проект-менеджеру, замечания устраняются, ТЗ утверждается и прилагается к контракту.

Тема 1.6. Структура и формат статические и динамические данные (2 ч)

Понятие о ЕСПД. Виды программ. Виды программных документов. Виды эксплуатационных документов. Общие требование к программному документу. Обозначение программ и программных документов. Требование и правила для оформления структурных схем, алгоритмов. Понятие спецификации. Внешняя и внутренняя спецификации и их особенности. Требования к структуре внешней спецификации.

Раздел 2. Методология проектирование программных продуктов

Тема 2.1 Методы проектирования ПП (2 ч)

Методы проектирования программных продуктов и признаки их классификации. Неавтоматизированное и автоматизированное проектирование алгоритмов и программ.

Структурное проектирование программных продуктов и его методы. Принцип системного проектирования. Нисходящие проектирование. Модульное проектирование. Структурное программирование. Функционально-ориентированные методы и методы структурирования данных.

Информационное моделирование предметной области и его составляющие. Технологии информационного моделирования. Инфологическая и даталогические модели. Логический и физический уровень представления даталогической модели.

Сущность объектно-ориентированного подхода к проектированию программных продуктов. Объектно-ориентированный анализ предметной области и объектно-ориентированное проектирование. Объектно-ориентированная технология и ее преимущества.

Лабораторная работа №1 (2 ч)

Пример «Банкомат».

Диаграмма вариантов использования «Банкомат»
Оформление СЯС-карты
Клиент
Банк
Примеры С11С-карт
Примеры С11С-карт

Отчет по лабораторной работе должен состоять из:
1. Постановки задачи.
2. Описания действующих лиц и прецедентов системы.
3. Диаграммы прецедентов.
4. СЯС-карты.
5. Диаграммы взаимодействия.

Тема 2.2 Структура ПП (2 ч)

Внутренняя организация программного продукта. Цели структуризации программных продуктов. Типовая структура программного продукта. Головной, управляющий модуль, рабочие и сервисные модули. Структура пакетов прикладных программ.

Библиотеки стандартных программ и подпрограмм. Правила работы с библиотеками стандартных программ, встроенные функции. Возможность использования встроенных функций.

Лабораторная работа №2 (2 ч)

Системное ПО
Прикладное ПО
Инструментальные системы

Тема 2.3 Проектирования интерфейса пользователя (2 ч)

Интерфейс пользователя программного продукта. Классификация систем, поддерживающих диалоговые процессы. Системы с жестким сценарием, дескрипторные системы, тезаурусные системы, системы с языком деловой прозы. Характеристика сценария диалогового процесса. Требования, предъявляемые к стандартному графическому интерфейсу пользователя. Инструментарий создания интерфейса пользователя.

Лабораторная работа №3 (2 ч)

«Автоматизация работы организации по трудоустройству населения».
«Автоматизация работы транспортного агентства».
«Автоматизация обработки информации по работе туристической фирмы».
«Автоматизация учета реализации книжной продукции»
«Учет эмиграции населения».
«АРМ администратора ресторана».
«Автоматизация учебного процесса».

Раздел 3 Разработка программных продуктов

Тема 3.1 Стиль программирования (2 ч)

Понятия «стиль» и «стилистика» программирования. Правила хорошего стиля. Требования к стилю написания программы. Типы существующих стилей написания программы (классический, пользовательский, программиста и т.д.).

Лабораторная работа №4 (2 ч)

Краткие теоретические сведения
Стиль программирования связан с удобочитаемостью программы**.**
Правила хорошего стиля программирования – это результат соглашения между опытными программистами.

Правило стандартизации стиля заключается в следующем: если существует более одного способа сделать что-либо и выбор произвольный, остановитесь на одном способе, и всегда его придерживайтесь. Программное средство, представленное в хорошем стиле, имеет комментарии (пояснительные, вводные иногда оглавления), значимые идентификаторы, хорошо воспринимаемый текст ПС.
Пользовательский интерфейс также должен быть разработан в хорошем стиле, придерживаясь следующих рекомендаций:
- пользовательский интерфейс должен базироваться на терминах и понятиях, знакомых пользователю;
- пользовательский интерфейс должен быть единообразным;
- пользовательский интерфейс должен позволять пользователю исправлять собственные ошибки;
- пользовательский интерфейс должен позволять получение пользователем справочной информации: как по его запросу, так и генерируемой ПС.

Тема 3.2 Языки программирования (2 ч)

Практическая работа №5

Языки программирования и их классификация. Выбор и обоснование языка программирования. Языки программирования для решения экономических, научных, инженерных задач. Языки системного программирования. Комбинирование языков программирования в рамках одной задачи.

Современные языки программирования. общие понятия. Язык программирования Java. Общее сведение о языках программирования. Знакомства с интерфейсом программного языка Java. Создание первой программы «Привет мир!!!»

Тема 3.3 Модульное программирование (2 ч)

Лабораторная работа №5

Модульное программирование как метод разработки программ. Программный модуль и его основные характеристики. Типовая структура программного модуля. Порядок разработки программного модуля.

Задание:
А) Оформить в виде основной программы и процедуры (функции) с параметрами программу, выполняющую обработку матрицы в соответствии с заданием лабораторной работы №2.
В) Оформить в виде модуля программу, выполняющую обработку матрицы в соответствии с заданием лабораторной работы №2.

Тема 3.4 Структурное программирование (2 ч)

Теория и методы структурного программирования. Методы восходящей и нисходящей разработки структуры программы. Конструктивный и архитектурный подход к разработке программы.

Основные управляющие конструкции структурного программирования. Метод пошаговой детализации текста модуля. Структурное кодирования. Правила составления структурированных алгоритмов и их структурная композиция.

Основная концепция структурирования программ. Метод структурирования программ.

Практическая работа №6 (2 ч)

Структурное программирование.
Цель структурного программирования.
Цикл.
Бесконечный цикл.
Цикл с предусловием.
Цикл со счетчиком.
Задание 1 – Написать программу, выполненную в структурном стиле. Программа должна рассчитывать площадь фигур (программа должна корректно отрабатывать данные согласно варианту в приложении А). На вход программа запрашивает строку, если в нее введено название фигуры, то программа запрашивает необходимые параметры фигуры, если введено значение отличное от названия фигуры, то программа повторно предлагает ввести название фигуры, если пользователь не справляется с этой задачей более 3 раз подряд, то программа сообщает о некорректности действий пользователя и завершается. В случае введения корректных данных программа должна выдать ответ, а также описание хода решения. Программа должна быть выполнена в виде блок-схемы и на ЯВУ.
Задание 2 – Написать программу вычисляющую площадь неправильного многоугольника. Многоугольник на плоскости задается целочисленными координатами своих N вершин в декартовой системе. Стороны многоугольника не соприкасаются (за исключением соседних - в вершинах) и не пересекаются. Программа в первой строке должна принимать число N – количество вершин многоугольника, в последующих N строках – координаты соответствующих вершин (вершины задаются в последовательности против часовой стрелки). На выход программа должна выдавать площадь фигуры. Программа должна быть выполнена в виде блок-схемы и на ЯВУ.

Тема 3.5 Объектно-ориентированное программирование

Практическая работа №7 (2 ч)

Основные понятия объектно-ориентированного проектирования. Объект, свойства объекта, метод обработки, событие, класс объектов.

Методика объектно-ориентированного проектирования и его основные принципы. Инкапсуляция, наследование, полиморфизм. Основные составляющие объектно-ориентированного анализа. Этапы объектно-ориентированного проектирования. Структура объектно-ориентированных программ.

1. Изучить теоретический материал.
2. Написать программу согласно индивидуальному варианту задания.
3. Ответить на контрольные вопросы.

Лабораторная работа №6 (2 ч)

ЗАДАНИЕ: Разработать объектно-ориентированную программу для вычисления длины любых отрезков прямых линий. Отрезки должны задаваться парами координат своих концов на плоскости через поток стандартного ввода. Результат вычисления длины каждого заданного отрезка должен отображаться через поток стандартного вывода. Программная реализация этих вычислений должна быть основана на разработке контейнерного класса отрезка прямой. Его компонентные данные должны включать подобъекты класса точки для хранения координат концов отрезка, а компонентный метод должен обеспечивать вычисление длины отрезка по ним. Конструкторы классов отрезка и точки должны использовать списки инициализации своих компонентных данных.

Тема 3.6 Эффективность и оптимизация программ (2 ч)

Понятие эффективности программ. Основные критерии эффективности программного продукта. Организация эффективной работы программы при экономичном использовании ресурсов ПЭВМ. Возможности увеличения быстродействия. Оптимизация программ на этапе отладки. Принципы и приемы оптимизации. Работа с оптимизирующими компиляторами.

Практическая работа №8 (2 ч)

Задание 1. Создание исходной таблицы.
Задание 2. Сортировка данных.
Задание 3. Фильтрация. Задание
4. Итоги.

Тема 3.7 Обеспечение качества программного продукта (2 ч)

Принципы обеспечения показателей качества программного продукта. Функциональность и надежность как обязательные критерии качества программного продукта. Корректность программ, ее составляющие, программные эталоны и методы проверки корректности. Обеспечение легкости применения продукта. Обеспечение мобильности, модифицируемости и интеграции программных продуктов.

Лабораторная работа №7 (2 ч)

Цель работы: в лабораторной работе оцениваем качественные показатели ПП.

Раздел. 4 Отладка, тестирование и сопровождение программ

Тема 4.1 Ошибки программного обеспечения

Лабораторная работа №8,9 (4 ч)

Понятие об ошибке программного обеспечения. Источники ошибок программного обеспечения. Классификации ошибок программного обеспечения. Основные пути и методы борьбы с ошибками программного обеспечения. Обнаружение и локализация ошибок ввода и обработки данных.

1. Технические ошибки
2. Программные ошибки
3. Алгоритмические ошибки
4. Системные

Тема 4.2 Отладка программ

Лабораторная работа №10,11 (4 ч)

Понятие отладки программ. Составляющие процесса отладки. Принципы и виды отладок. Автономная и комплексная отладки программ. Методы отладки Средства отладки. Рекомендации по организации отладки Автономная отладка модуля. Использование средств отладки.

Отладка модулей программы и программы в целом.

Отладчик IntelliJ IDEA
1. *Запуск отладчика*

После того как вы настроите конфигурацию запуска вашего проекта, вы можете запускать его в режиме отладки, нажав *Shift + F9.*
В окне отладчика вы можете видеть стек вызовов функций и список потоков, с их состояниями, переменными и окнами просмотра состояния. Когда вы выбираете контекст вызова функции, вы можете просмотреть значения переменных соответствующих выбранному контексту.

2. *Полезные клавиатурные сокращения отладчика*

- Установить/снять точку останова - *Ctrl + F8 (Cmd + F8 для Mac)*
- Возобновить выполнение программы - *F9*
- Перейти к следующей инструкции - *F8*
- Перейти внутрь функции - *F7*
- Приостановить выполнение - *Ctrl + F2 (Cmd + F2)*
- Переключить между просмотром списка точек останова и подробной информацией о выбранной точке - *Shift + Ctrl + F8 (Shift + Ctrl + F8)*
- Запустить отладку кода с точки на которой стоит курсор - *Shift + Ctrl + F9* (если это внутри метода *main()*)

3. *Умный переход внутрь*

Иногда вам надо при пошаговой отладке перейти внутрь определенного метода, но не первого который будет вызван. В таком случае вы можете нажать *Shift + F7 (Cmd + F7 для Mac)* чтобы выбрать из предложенного списка метод который вам нужен. Это может сэкономить вам массу времени.

Тема 4.3 Тестирование программ

Лабораторная работа №12,13 (4 ч)

Сущность и необходимость тестирования программного обеспечения. Различие между тестированием и отладкой программного обеспечения.

Основные принципы организации тестирования. Стадии тестирования. Виды тестовых проверок. Объекты тестирования и категории тестов. Виды тестирования.

Методы структурного тестирования программного обеспечения. Принцип «белого ящика». Пошаговое и монолитное тестирование модулей. Нисходящее и восходящее тестирование программного обеспечения.

Методы функционального тестирования. Принцип «черного ящика». Метод эквивалентного разбиения. Метод анализа граничных условий. Метод функциональных диаграмм. Комбинированные методы тестирования.

Средства тестирования. Ручное и автоматизированное тестирование. Применение методов и инструментальных средств тестирования.

• постановка задачи для теста;
• проектирование теста;
• написание тестов;
• тестирование тестов;
• выполнение тестов;
• изучение результатов тестирования.

Тема 4.4 Сопровождение программ

Лабораторная работа №14,15 (4 ч)

Сопровождение программных продуктов, внесение изменений, обеспечение надежности при эксплуатации. Необходимая документация и предпродажная подготовка программных средств.

Технические требования (спецификации). Руководства специалиста по сопровождению. Руководства пользователя. Руководства по вводу в действие и инсталляции.

Тема 4.5. Защита программ

Практическая работа №9,10 (4 ч)

Основные понятия о защите программных продуктов. Методы защиты программных продуктов от несанкционированного доступа и копирования. Системы разграничения доступа. Криптографические методы защиты программных продуктов, их особенности. Аппаратные средства защиты программного продукта.

Правовые методы защиты программных продуктов. Патентная защита лицензионные соглашения.

Алгоритм реализации:
1. Запомнить текущее время;
2. Выполнить контрольный участок кода;
3. Запомнить текущее время и разность текущего и предыдущего запомненного времени;
4. Выполнить контрольный участок кода повторно;
5. Сравнить разность текущего времени и предыдущего запомненного текущего времени с предыдущей запомненной разностью;
6. Если разности совпадают, продолжить выполнение, иначе – выйти из программы.

Тема 4.6 Пакеты прикладных программ

Практическая работа №11, 12, 13 (6 ч)

Проблемно–ориентированные ППП. Офисные ППП. Коммуникационные ППП

Раздел 5. Инструментальные средства разработки программ

5.1 Общая характеристика инструментальных средств разработки программ

Практическая работа №14 (2 ч)

Общая характеристика инструментальных средств разработки программ. Инструменты разработки программных продуктов. Инструментальные системы технологии программирования и их черты: комплексность, ориентированность на коллективную разработку, технологическая определенность, интегрированность. Основные компоненты инструментальных систем технологии программирования: репозиторий, инструментарий, интерфейсы.

CASE-средства, их назначение и применение. Классификации CASE-средств. Характеристика современных CASE-средств.

Текстовые редакторы. Интегрированные среды разработки. Компиляторы. Интерпретаторы. Линковщики. Парсеры и генераторы парсеров (см. Javacc). Ассемблеры.

Отладчики. Профилировщики. Генераторы документации. Средства анализа покрытия кода. Средства непрерывной интеграции. Средства автоматизированного тестирования. Системы управления версиями и др.

Тема 5.2 Применение CASE-средств

Практическая работа №15 (2 ч)

Построение модулей программных систем с использованием структурного и объектно-ориентированного подхода. Диаграммы потоков данных и диаграммы «сущность связь».

Основные сведения о языке UML. Построение концептуальной модели предметной области. Диаграммы моделирования языка UML. Работа в среде CASE-средства.

Построение диаграмм потоков данных.

Раздел 6. Коллективная разработка программных средств (2 ч)

Тема 6.1 Организации работ при коллективной разработке программных продуктов

Категории специалистов, занятых разработкой и эксплуатацией программ. Принципы и методы коллективной разработки программных продуктов. Организация коллективной работы программистов. Схема взаимодействия специалистов, связанных с созданием и эксплуатацией программ. Типы организации бригад. Бригада главного программиста. Обязанности членов бригады. Распределение обязанностей в бригаде.

Тема 6.2 Экономическая аспекты создания и использования программных средств

Стоимость программных средств. Факторы, влияющие на стоимость программных средств. Методики оценки трудоемкости разработки программного продукта. Особенности продаж программных продуктов. Обновление версий программных средств. Способы прогнозирования рынка программного обеспечение.

Раздел 7. Курсовое проектирование (18 ч)

Курсовое проектирование является завершающим этапом в изучении дисциплины «Технология разработки программных продуктов», в ходе которого осуществляется обучение применению полученных знаний и умений при решении комплексных задач, связанных со сферой профессиональной деятельности будущих специалистов.

Целью курсового проектирование является закрепление и углубление теоретических знаний, и приобретение практических навыков по разработке и проектированию ПО для заданной проблемы. Основными задачами курсового проекта являются:

- анализ возможных подходов и методов решения с обоснованием, выбранного подход; выбор или разработка модели (математической, структурной, информационной), необходимой для достижения цели;
- выбор или эффективных алгоритмов с учетом их точности, устойчивости, сходимости; анализ полученных результатов работы программного обеспечения. систематизации и закрепления полученных теоретических знаний и практических умений по общепрофессиональным и специальным дисциплинам;
- углубления теоретических знаний в соответствии с заданной темой;
- формирования умения применять теоретические знания при решении поставленных профессиональных задач;

- формирования умения использовать справочную, нормативную и правовую документацию;
- развития творческой инициативы, самостоятельности, ответственности и организованности;
- подготовки к итоговой государственной аттестации.

1.1. Введение. Программные продукты и их основные характеристики

Основные понятия программного продукта

Программа (program) — упорядоч. последовательность команд компьютера для решения задачи.

Программное обеспечение (sowtware) — совокупность программ обработки данных и необходимых для их эксплуатации документов.

Программы предназначены для машинной реализации задач. Термины задача и приложение имеют очень широкое употребление в контексте информатики и программного обеспечения. При этом **задача (problem)** — это проблема, подлежащая решению, а **приложение (application)** — программная реализация на компьютере решения задачи. Т. о., задача означает проблему, подлежащую реализации с использованием средств информационных технологий, а приложение — реализованное на компьютере решение этой задачи. Приложение считается более удачным термином и широко исп-ся в информатике. Необходимо заметить, что термин **задача** также употребляется в сфере программирования как единица работы вычислительной системы, требующая выделения вычислительных ресурсов (процессорного времени, основной памяти и т.п.).

Существует разнообразные классификации задач. С позиций специфики разработки и вида ПО— **технологические** и **функциональные**.

Технологические задачи ставятся и решаются при организации технологического процесса обработки информации на компьютере. **Функциональные задачи** требуют решения при реализации функций управления в рамках информационных систем предметных областей. **Предметная (прикладная) область (application domain)** — это совокупность связанных между собой функций, задач управления, с помощью которых достигается выполнение поставленных целей.

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

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

Программные продукты (изделия) предназначены для удовлетворения потребностей пользователей, широкого распространения и продажи.

В настоящее время существуют и другие варианты легального распространения программных продуктов, которые появились с использованием глобальных или региональных телекоммуникаций:

- **freeware** — бесплатные программы, свободно распространяемые, поддерживаются самим пользователем, который правомочен вносить в них необходимые изменения;
- **shareware**— некоммерческие (условно-бесплатные) программы, которые могут использоваться, как правило, бесплатно. При условии регулярного использования подобных продуктов осуществляется взнос определенной суммы.

Ряд производителей использует **OEM-программы (Original Equipment Manufacturer)**, т.е. встроенные программы, устанавливаемые на компьютеры или поставляемые вместе с вычислительной техникой.

Программный продукт — это комплекс взаимосвязанных программ для решения определенной проблемы (задачи) массового спроса, подготовленный к реализации как любой вид промышленной продукции.

Программные продукты могут создаваться как:

- индивидуальная разработка под заказ;
- разработка для массового распространения среди пользователей.

Сопровождение программного продукта — это поддержка работоспособности программного продукта, переход на его новые версии, внесение изменений, исправление обнаруженных ошибок и т.п.

Основными характеристиками программ являются:

1)алгоритмическая сложность (логика алгоритмов обработки информации); **2)**состав и глубина проработки реализованных функций обработки; **3)**полнота и системность функций обработки; **4)**объем файлов программ; **5)**требования к операционной системе и техническим средствам обработки со стороны и др.

Программные продукты имеют многообразие **показателей качества**

Структура характеристик качества программных продуктов можно представить в виде схемы :

Мобилъностъ программных продуктов означает их независимость от технического комплекса системы обработки данных, операционной среды, сетевой технологии обработки данных, специфики предметной области и т.п. **Надежность** работы программного продукта определяется бессбойностью и устойчивостью в работе программ, точностью выполнения предписанных функций обработки, возможностью диагностики возникающих в процессе работы программ ошибок.

Эффективность программного продукта оценивается как с позиций прямого его назначения — требований пользователя, так и с точки зрения расхода вычислительных ресурсов, необходимых для его эксплуатации.

Расход вычислительных ресурсов оценивается через объем внешней памяти для размещения программ и объем оперативной памяти для запуска программ.

Учет человеческого фактора означает обеспечение дружественного интерфейса для работы конечного пользователя, наличие контекстно-зависимой подсказки или обучающей системы в составе программного средства, хорошей документации для освоения и использования заложенных в программном средстве функциональных возможностей, анализ и диагностику возникших ошибок и др.

Модифицируемость программных продуктов означает способность к внесению изменений, например расширение функций обработки, переход на другую техническую базу обработки и т.п.

Коммуникативность программных продуктов основана на максимально возможной их интеграции с другими программами, обеспечении обмена данными в общих форматах представления (экспорт/импорт баз данных, внедрение или связывание объектов обработки и др.).

В условиях существования рынка программных продуктов важными характеристиками являются:1)стоимость; 2)количество продаж; 3)время нахождения на рынке;4)известность фирмы-разработчика и программы; 5)наличие программных продуктов аналогичного назначения.

Спецификой программных продуктов является также и то, что их эксплуатация должна выполняться на правовой основе — лицензионные соглашения между разработчиком и пользователями с соблюдением авторских прав разработчиков программных продуктов.

. Методы защиты программных продуктов

Методы защиты программных продуктов условно можно подразделить на программные и правовые. При использовании программных систем программа выполняется только при опознании некоторого уникального некопируемого **ключевого элемента**.

Таким **ключевым элементом** могут быть:

- дискета, на которой записан не подлежащий копированию ключ;
- определенные характеристики аппаратуры компьютера;
- специальное устройство (электронный ключ), подключаемое к компьютеру и предназначенное для выдачи опознавательного кода.

Программные системы защиты от копирования программных продуктов:

- идентифицируют среду, из которой будет запускаться программа;
- устанавливают соответствие среды, из которой запущена программа, той, для которой разрешен санкционированный запуск;
- вырабатывают реакцию на запуск из несанкционированной среды;
- регистрируют санкционированное копирование;
- противодействуют изучению алгоритмов и программ работы системы.

Для идентификации **запускающих дискет** применяются следующие методы:

- нанесение повреждений на поверхность дискеты (т.н. "лазерная дыра"), которая с трудом может быть воспроизведена в несанкционированной копии дискеты;
- нестандартное форматирование запускающей дискеты.

Идентификация среды компьютера обеспечивается за счет:

- закрепления месторасположения программ на жестком магнитном диске (т.н. **неперемещаемые программы**);
- привязки к номеру **BIOS** (расчет и запоминание с последующей проверкой при запуске контрольной суммы системы);
- привязки к аппаратному ключу, вставляемому в порт ввода-вывода.

Правовые методы защиты программ включают:

1)патентную защиту(устанавливает приоритет в разработке и использовании нового подхода или метода, примененного при разработке программ, удостоверяет их оригинальность), **2)**закон о производственных секретах (статус **производственного секрета** для программы ограничивает круг лиц, знакомых или допущенных к ее эксплуатации, а также определяет меру их ответственности за разглашение секретов); **3)**лицензионные соглашения и контракты (**лицензионные соглашения** распространяются на все аспекты правовой охраны программных продуктов, включая авторское право, патентную защиту, производственные секреты); 4)закон об авторском праве.

1.2.1. Классификация программных продуктов

1.2.2. Классификация программных продуктов

- *Классы программных продуктов*
- *Системное программное обеспечение*
- *Инструментарий технологии программирования*
- *Пакеты прикладных программ*

КЛАССЫ ПРОГРАММНЫХ ПРОДУКТОВ

Программные продукты можно классифицировать по различным признакам. Рассмотрим классификацию, в которой основополагающим признаком является сфера (область) использования программных продуктов:

- аппаратная часть автономных компьютеров и сетей ЭВМ;
- функциональные задачи различных предметных областей;
- технология разработки программ.

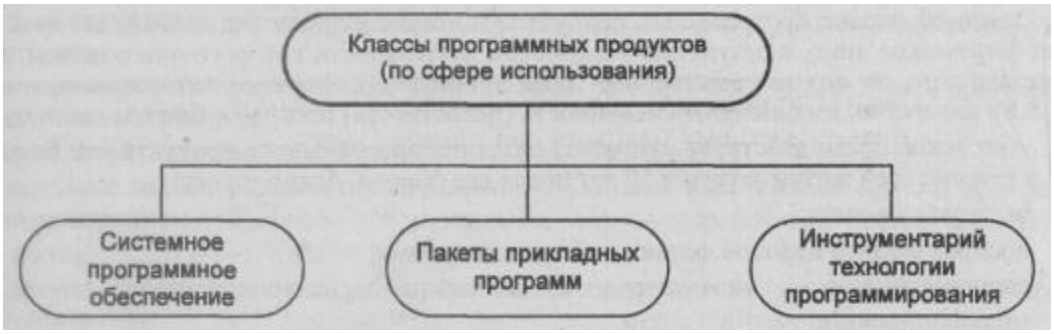


Рис. 8.9. Классы программных продуктов

Для поддержки информационной технологии в этих областях выделим соответственно три класса программных продуктов, представленных на рис. 8.9:

- системное программное обеспечение;
- пакеты прикладных программ;
- инструментарий технологии программирования.

Системное программное обеспечение направлено:

- на создание операционной среды функционирования других программ;
- на обеспечение надежной и эффективной работы самого компьютера и вычислительной сети;
- на проведение диагностики и профилактики аппаратуры компьютера и вычислительных сетей;
- на выполнение вспомогательных технологических процессов (копирование, архивирование, файлов программ и баз данных и т.д.).

Данный класс программных продуктов тесно связан с типом компьютера и является его неотъемлемой частью. Программные продукты в основном ориентированы на квалифицированных пользователей – профессионалов в компьютерной области: системного программиста, администратора сети, прикладного программиста, оператора. 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 (System Software) – совокупность программ и программных комплексов для обеспечения работы компьютера и сетей ЭВМ.

Пакеты прикладных программ (ППП) служат программным инструментарием решения функциональных задач и являются самым многочисленным классом программных продуктов. В данный класс входят программные продукты, выполняющие обработку информации различных предметных областей.

Установка программных продуктов на компьютер выполняется квалифицированными пользователями, а непосредственную их эксплуатацию осуществляют, как правило, конечные пользователи – потребители информации, во многих случаях деятельность которых весьма далека от компьютерной области. Данный класс программных продуктов может быть весьма специфичным для отдельных предметных областей.

Пакет прикладных программ (application program package) – комплекс взаимосвязанных программ для решения задач определенного класса конкретной предметной области.

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.

Инструментарий технологии программирования – совокупность программ и программных комплексов, обеспечивающих технологию разработки, отладки и внедрения создаваемых программных продуктов.

СИСТЕМНОЕ ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ

Структура системного программного обеспечения

На рис. 8.10 представлена структура системного программного обеспечения – *базового программного обеспечения,* которое, как правило, поставляется вместе с компьютером, и *сервисного программного обеспечения,* которое может быть приобретено дополнительно.

Базовое программное обеспечение *(base software)* – минимальный набор программных средств, обеспечивающих работу компьютера. **Сервисное программное обеспечение** – программы и программные комплексы, которые расширяют возможности базового программного обеспечения и организуют более удобную среду работы пользователя.

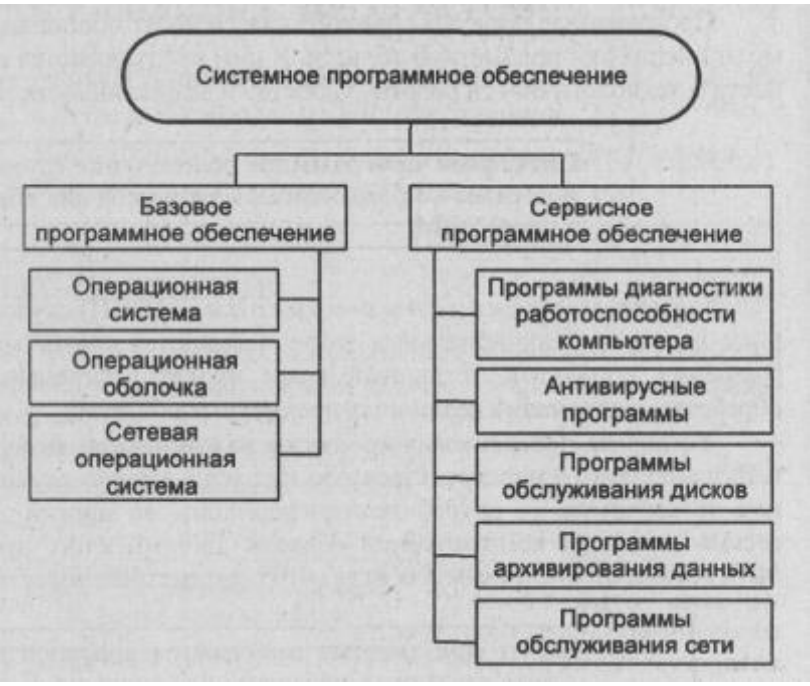


Рис. 8.10. Классификация системного программного обеспечения компьютера

Базовое программное обеспечение

В базовое программное обеспечение входят:

- операционная система;
- операционные оболочки (текстовые и графические);
- сетевая операционная система.

Операционная система предназначена для управления выполнением пользовательских программ, планирования и управления вычислительными ресурсами ЭВМ.

В секторе программного обеспечения и операционных систем ведущее положение занимают фирмы IBM, Microsoft, UNISYS, Novell. Доход от продаж операционных систем в среднем превышает 20 млрд. дол. в год. Рассмотрим наиболее распространенные типы операционных систем.

Операционные системы для персональных компьютеров делятся на:

- одно- и многозадачные (в зависимости от числа параллельно выполняемых прикладных процессов);
- одно- и многопользовательские (в зависимости от числа пользователей, одновременно работающих с операционной системой);
- непереносимые и переносимые на другие типы компьютеров;
- несетевые и сетевые, обеспечивающие работу в локальной вычислительной сети ЭВМ.

Большое значение сегодня имеет применение 32-разрядных операционных систем для персональных компьютеров:

- OS/2 во всех модификациях (IBM);
- Windows NT во всех модификациях (Microsoft);
- Unix во всех модификациях;
- Next Step 3.2 (Next);
- SCO Open Desktop 3.0 (Santa Cruz Operation);
- Solaris 2.1 (SunSoft) – x86;
- UnixWare Personal Edition 1.0 (Novell).

По данным опроса пользователей программных продуктов, проведенного в 1996 г., мнение респондентов относительно операционных систем распределилось так, как указано в табл. 8.1.

Таблица 8.1. Рейтинг операционных систем

Тип ОС	Имеют ОС	Считают лучшей
MS DOS	62.4%	18.8%
Windows 3.x	52.8%	14.0%
Windows 95	45.4%	23.1%
OS/2	14.0%	12.5%
NetWare	10.2%	6.8%
Windows NT	10.2%	13.0%
Unix	7.9%	5.4%

Операционная система MS DOS (фирма Microsoft) появилась в 1981 г. В настоящее время существуют версии 6.22 и 7.0 (в составе Windows 95), а также ее разновидности других фирм-разработчиков (DR DOS, PC DOS). Сегодня эта операционная система установлена на подавляющем большинстве персональных компьютеров. Начиная с 1996 г. MS DOS распространяется в виде Windows 95 – 32-разрядной многозадачной и

многопоточной операционной системы с графическим интерфейсом и расширенными сетевыми возможностями.

Операционная система OS/2 разработана фирмой IBM для персональных компьютеров на основе системной прикладной архитектуры, ранее используемой для больших ЭВМ. Это многозадачная, однопользовательская, высоконадежная операционная система, обеспечивающая как текстовый, так и графический интерфейс пользователя. OS/2 обеспечивает:

- поддержку графического интерфейса пользователя;
- одновременную обработку нескольких приложений;
- многопоточную обработку нескольких задач одного приложения;
- 32-разрядную обработку данных;
- сжатие данных при записи на магнитные диски;
- защиту памяти.

Важной особенностью операционной системы OS/2 является высокопроизводительная файловая система HPFS (High Perfomance File System), имеющая преимущества для серверов баз данных (в отличие от MS DOS поддерживаются длинные имена файлов), поддержка мультипроцессорной обработки – до 16 процессоров типа INTEL и PowerPC. Версия OS/2 Warp работает с мультисредой и имеет встроенный доступ в сеть Internet, систему распознавания речи VoiceType, интегрированную версию Lotus Notes Mail для передачи через Internet почты. В OS/2 могут выполняться прикладные программы Windows 3.1 и Win32s, но не могут выполняться приложения, работающие в среде Windows 95 или Windows NT. Спецификация Open 32 позволяет поставщикам программного обеспечения переносить его на новую платформу.

Перспективной является многопользовательская и многозадачная *операционная система Unix,* созданная корпорацией Bell Laboratory. Данная операционная система реализует принцип открытых систем и широкие возможности по комплексированию в составе одной вычислительной системы разнородных технических и программных средств.

Unix обладает наиболее важными качествами, такими, как:

- переносимость прикладных программ с одного компьютера на другой;
- поддержка распределенной обработки данных в сети ЭВМ;
- сочетаемость с процессорами RISC.

Unix получила распространение для суперкомпьютеров, рабочих станций и профессиональных персональных компьютеров, имеет большое количество версий, разработанных различными фирмами. Согласно прогнозам объем мирового рынка вычислительных систем, базирующихся на ОС Unix, существенно будет возрастать, особенно с переходом к сетевым технологиям.

Наиболее традиционное сравнение ОС осуществляется по следующим характеристикам процесса обработки информации:

- управление памятью (максимальный объем адресуемого пространства, типы памяти, технические показатели использования памяти);
- функциональные возможности вспомогательных программ (утилит) в составе операционной системы;

- наличие компрессии диска;
- возможность архивирования файлов;
- поддержка многозадачного режима работы;
- поддержка сетевого программного обеспечения;
- наличие качественной документации;
- условия и сложность процесса инсталляции.

Сетевые операционные системы - комплекс программ, обеспечивающий обработку, передачу и хранение данных в сети. Сетевая ОС предоставляет пользователям различные виды сетевых служб (управление файлами, электронная почта, процессы управления сетью и др.), поддерживает работу в абонентских системах. Сетевые операционные системы используют архитектуру *клиент-сервер* или одноранговую архитектуру. Вначале сетевые операционные системы поддерживали лишь локальные вычислительные сети (ЛВС), сейчас эти операционные системы распространяются на ассоциации локальных сетей. Наибольшее распространение имеют LAN Server, NetWare, VINES, Windows NT, Windows 95.

Они оцениваются по комплексу критериев: производительность, разнообразие возможностей связи пользователей, возможности администрирования.

Основные характеристики широко распространенной сетевой операционной системы Novell NetWare рассмотрены в гл. 7. Операционная система Windows NT является многозадачной, предназначенной для архитектуры клиент-сервер и использования различных протоколов транспортного уровня сетевой операционной системы, имеет 32-разрядную архитектуру и обеспечивает функции локальной сети:

- возможность каждой абонентской системы в сети быть сервером или клиентом;
- совместную работу группы пользователей;
- адресацию оперативной и внешней памяти большого размера;
- многозадачность и многопоточность обработки данных;
- поддержку мультипроцессорной обработки и др.

В табл. 8.2 приведены характеристики некоторых популярных ОС.

Таблица 8.2. Характеристики операционных систем

Операционная система	Объем ОЗУ, Мбайт	Память на МД, Мбайт	Средства связи с Internet	Сетевые средства связи	Многопроцессорная обработка	Средства взаимодействия прикладных программ
MS Windows NT Workstation 4.0	12 – 16	90 – 120	Internet Explorer 2.0, Peer Web Services, Point to Point, FTP, telnet	Banyan, Digital, IBM, Novell, Microsoft SNA, TCP/IP и др.	Есть	OLE, Open GL, Win 32
MS Windows NT 3.51	12 – 16	75 – 95	То же	То же	Есть	То же
MS Windows 95	4 – 12	25 – 85	То же	То же	Нет	OLE, Win 32
MS Windows 3.11	4 – 8	20 – 35	—	—	Нет	OLE, Win 32s
OS/2 Warp 4.0	8 – 12 (12 – 24)	100 – 300	FTP, Gopher, Newsreader/2, telnet, Web Explorer	ArtiSoft IBM, Novell, Microsoft, TCP/IP	Нет	Java, OpenDoc, OpenGL, PM API, Win32s
OS/2 Warp Connect 3.0	8	80 – 120	То же	То же	Нет	PM API, Win32s

Работа в сети ЭВМ требует использования программных продуктов для администрирования и обслуживания рабочих станций типа, например:

- IBM Adstar Distributed Storage Manager 1.0 – средство резервного копирования в масштабах предприятия, которое позволяет обслуживать ЭВМ различных классов (мэйнфреймы, мини-ЭВМ, настольные системы), используя при этом один общий интерфейс);
- Symantec Norton Administrator for Networks 2.0 – обеспечивает администрирование локальной вычислительной сети и управление приложениями для корпоративных сетей (масштаба предприятия);
- Microsoft NT File and Print Service for NetWare – устраняет барьеры между NetWare и Windows NT, обеспечивает полную эмуляцию для NT возможностей среды NetWare;
- Armon OnSite Manager – сегментирование сетей, фильтрация и поиск неисправностей в сетях масштаба предприятия и др.

Операционные оболочки – специальные программы, предназначенные для облегчения общения пользователя с командами операционной системы. Операционные оболочки имеют текстовый и графический варианты интерфейса конечного пользователя.

Наиболее популярны следующие виды текстовых оболочек операционной системы MS DOS:

- Norton Commander 5.0 – фирма Symantec *(см.* гл. 10);
- XTree Gold 4.0;
- Norton Navigator и др.

Эти программы существенно упрощают задание управляющей информации для выполнения команд операционной системы, уменьшают напряженность и сложность работы конечного пользователя.

Во всем мире имеют огромную популярность такие графические оболочки MS DOS, как Windows 3.1 *(см.* гл. 12), Windows 3.11 for WorkGroup, которые позволяют изменить среду взаимодействия пользователя с компьютером, расширяют набор основных (диспетчер файлов, графический редактор, текстовый редактор, картотека и т.п.) и сервисных функций, обеспечивающих пользователю интегрированную информационную технологию вплоть до создания одноранговых локальных сетей.

1.3.1. Особенности создания программного продукта

1.3.2. Особенности создания программного продукта

Разрабатывая конкретное приложение, необходимо четко разграничивать понятие "прикладная программа" (приложение) и "программный продукт".

Программный продукт -- комплекс взаимосвязанных программ для решения определенной проблемы (задачи) массового спроса, подготовленный к реализации как любой вид промышленной продукции.

В настоящее время существуют различные варианты легального распространения программных продуктов, которые появились с использованием глобальных или региональных телекоммуникаций:

* freeware -- бесплатные программы, свободно распространяемые, поддерживаются самим пользователем, который правомочен вносить в них необходимые изменения;

* software -- некоммерческие (условно-бесплатные) программы, которые могут использоваться, как правило, бесплатно. При условии регулярного использования подобных продуктов осуществляется взнос определенной суммы.

Ряд производителей использует OEM-программы (Original Equipment Manufacturer), т.е. встроенные программы, устанавливаемые на компьютеры или поставляемые вместе с вычислительной техникой.

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

Программные продукты могут создаваться как:

* индивидуальная разработка под заказ;

* разработка для массового распространения среди пользователей.

При индивидуальной разработке фирма-разработчик создает оригинальный программный продукт, учитывающий специфику обработки данных для конкретного заказчика.

При разработке для массового распространения фирма-разработчик, с одной стороны, должна обеспечить универсальность выполняемых функций обработки данных, с другой стороны, гибкость и настраиваемость программного продукта на условия конкретного применения. Отличительной особенностью программных продуктов должна быть их системность -- функциональная полнота и законченность реализуемых функций обработки, которые применяются в совокупности.

Программный продукт разрабатывается на основе промышленной технологии выполнения проектных работ с применением современных инструментальных средств программирования. Специфика заключается в уникальности процесса разработки алгоритмов и программ, зависящего от характера обработки информации и используемых инструментальных средств. На создание программных продуктов затрачиваются значительные ресурсы -- трудовые, материальные, финансовые; требуется высокая квалификация разработчиков.

Как правило, программные продукты требуют сопровождения, которое осуществляется специализированными фирмами -- распространителями программ (дистрибьюторами), реже -- фирмами-разработчиками. Сопровождение программ массового применения сопряжено с большими трудозатратами -- исправление обнаруженных ошибок, создание новых версий программ и т.п.

Сопровождение программного продукта -- поддержка работоспособности программного продукта, переход на его новые версии, внесение изменений, исправление обнаруженных ошибок и т.п.

Поскольку любое разработанное приложение может быть доведено до уровня программного продукта, рассмотрим этапы его разработки:

1. Постановка задачи:

· сбор информации о задаче;

· формулировка условия задачи;

· определение конечных целей решения задачи;

· определение формы выдачи результатов;

· описание данных (их типов, диапазонов величин, структуры и т.п.).

На первом этапе раскрывается организационно-экономическая сущность задачи, т.е. формулируется цель ее решения; определяется взаимосвязь с другими задачами; указывается периодичность ее решения; устанавливаются состав и формы представления входной, промежуточной и результатной информации; характеризуются формы и методы контроля достоверности информации на ключевых этапах решения задачи; специфицируются формы взаимодействия пользователя с ЭВМ в ходе решения задачи и т.п.

2. Анализ и исследование задачи, построение модели:

· анализ существующих аналогов;

· анализ технических и программных средств;

· разработка математической и информационной модели;

· разработка структур данных.

На втором этапе технологического процесса разработки программ выполняется формализованное описание задачи, т.е. устанавливаются и формулируются логико-математические зависимости между исходными и результатными данными. Экономико-математическое описание задачи обеспечивает ее однозначное понимание пользователем и разработчиком программы.

3. Разработка алгоритма:

· выбор метода проектирования алгоритма;

· выбор формы записи алгоритма (блок-схема, псевдокод и др.);

· выбор тестов и метода тестирования;

· проектирование алгоритма.

Третий этап технологического процесса подготовки решения задач на ЭВМ представляет собой алгоритмизацию ее решения, т.е. разработку оригинального или адаптацию (уточнение и корректировку) уже известного алгоритма.

4. Программирование:

· выбор языка программирования;

· уточнение способов организации данных;

· запись алгоритма на выбранном языке программирования.

Четвертый этап технологического процесса подготовки решения задач на ЭВМ представляет собой составление (адаптацию) программ (кодирование). Процесс кодирования заключается в переводе описания алгоритма на один из доступных для ЭВМ языков программирования.

5. Тестирование и отладка:

· синтаксическая отладка;

· отладка семантики и логической структуры;

· тестовые расчеты и анализ результатов тестирования;

· совершенствование программы.

Тестирование и отладка составляют заключительный этап разработки программы решения задач. Тестирование представляет собой совокупность действий, предназначенных для демонстрации правильности работы программы. Процессу тестирования сопутствует понятие "отладка", которое подразумевает совокупность действий, направленных на устранение ошибок в программах, начиная с момента обнаружения фактов ошибочной работы программы и завершая устранением причин их возникновения.

6. Анализ результатов решения задачи и уточнение в случае необходимости математической модели с повторным выполнением этапов 2 - 5.

7. Сопровождение программы:

· доработка программы для решения конкретных задач;

· составление документации к решенной задаче, математической модели, алгоритму, программе по их использованию;

· приемо-сдаточные испытания;

· опытная эксплуатация;

· промышленная эксплуатация.

После завершения процесса тестирования и отладки программные средства вместе с сопроводительной документацией передаются пользователю для эксплуатации. При реализации

достаточно сложных и ответственных программных комплексов по согласованию пользователя (заказчика) с разработчиком этап эксплуатации программных средств может быть разбит на два подэтапа: экспериментальная (опытная) и промышленная эксплуатация.

В зависимости от специфических особенностей конкретной задачи, профессионального уровня подготовки специалистов и ряда других факторов некоторые этапы технологического процесса, представленные в общей схеме, могут быть объединены в более крупные этапы или реализовываться в неявном виде.

1.4.1. Жизненный цикл программы

1.4.2. Жизненный цикл программы

Жизненный цикл ПО – этапы, через которые проходит любое приложение, начиная от зарождения первоначальной идеи до непосредственного релиза. Так называют разработку и развитие программных продуктов.

Моделей жизненного цикла очень много. Если ошибиться с его выбором, можно или так и не выпустить итоговое приложение, или столкнуться с его провалом, а также серьезными затратами на реализацию.

Основной состав

Жизненный цикл (ЖЦ) обычно включает в себя:

- подготовку;
- проектирование;
- поддержку;
- написание (непосредственное создание).

Каждый этап ЖЦ может иметь различные названия, делиться на более мелкие составляющие. Обычно он предусматривает:

1. Приобретение. Так характеризуются действия заказчика, позволяющие сформировать требования и ограничения к желаемой программе. Пример – заключение договора на разработку ПОЮ анализ и аудит выполненных задач. Результатом станет получение готового продукта.
2. Поставку. Представлена мероприятиями, организованными специалистами. Работники анализируют выдвинутые клиентские требования, создают приложение, подводят итоги. После этого осуществляется решение вопросов, связанных с непосредственным программированием. Завершается проверкой (так называют тестирование) и поставкой программы.
3. Разработку. Это – комплекс мероприятий, характеризующийся написанием программного кода и формированием дизайна.
4. Эксплуатацию. Здесь начинается использование готового приложения.

5. Сопровождение. Поддержка пользователей по мере необходимости. Разработчики будут заниматься исправлением ошибок и возникающих неполадок.

Это – основные этапы жизненного цикла любого ПО. Сопровождение и эксплуатация – стадии, которые реализовываются одновременно.

Модель разработки – описание стадий жизненного цикла программного обеспечения. Она отражает то, что происходит на каждом этапе создания итогового продукта.

Методология – набор методов по управлению процессами разработки ПО. Правила, принципы и разнообразные техники, помогающие достигнуть максимальной эффективности/результативности.

Ключевые модели

Существуют различные виды ЖЦ. Они меняются в зависимости от конкретного приложения. Если уточнить особенности каждого варианта, получится выбрать оптимальное решение для создания ПО.

Основные модели жизненного цикла программного обеспечения:

- модель кодирования и устранения ошибок;
- водопадная модель;
- разработка через тестирование;
- инкрементная модель;
- итерационная модель;
- спиральная модель;
- модель «хаоса»;
- разработка через прототипирование.

Далее каждый вариант будет рассмотрен детально. А еще – сравнение для тех или иных целей разработки.

«Водопад»

Модель водопада (или каскадная модель) ЖЦ – это концепция, при которой жизненный цикл выглядит как поток, последовательно проходящий фазы анализа требований, проектирования, реализации, тестирования, интеграции и поддержки. Все здесь осуществляется «шаг за шагом». Следующий этап начнет после того, как завершится предыдущий.

Данная модель предусматривает независимость шагов. На каждом этапе выполняются дополнительные вспомогательные и организационные процессы и работы:

- управление проектом;
- оценка и манипуляция качеством;
- верификация;
- аттестация;
- менеджмент конфигурации;
- разработка документации.

По завершению этапов формируются так называемые промежуточные продукты. Они не подлежат изменению на последующих стадия жизненного цикла программного обеспечения.

Главные фазы этой системы обычно следующие:

- анализ требований;
- проектирование;
- кодирование (программирование);
- тестирование и отладка;
- эксплуатация и сопровождение.

Грамотная организация каскадной системы сделает разработку быстрой, эффективной и понятной. Она существует с 1970-х годов.

Плюсы и минусы

Изучая модели жизненного цикла ПО, нужно учитывать преимущества и недостатки каждого варианта. Они позволят выбрать оптимальное решение для проектов в тех или иных случаях.

К преимуществам каскадной модели относят:

1. Стабильность требований. Они будут едиными на протяжении всего жизненного цикла разработки.
2. Каждая стадия приводит к формированию законченного набора проектной документации. Она будет полной и согласованной.
3. Все фазы модели определены и понятны. Они с легкостью применяются на практике.
4. В процессе работы с этой моделью можно предсказать, когда начнется релиз и эксплуатация программы.

Перед тем как начнется разработка по каскадной модели, удается рассчитать стоимость работ. Она не будет меняться на протяжении всей реализации процесса.

Недостатки здесь тоже есть:

1. Динамические изменения невозможны. А еще здесь трудно сформулировать четкие требования.
2. Низкая гибкость в управлении проектом.
3. Линейная структура процесса разработки при обнаружении ошибок приводит к увеличению затрат и нарушению установленных изначально сроков реализации продукта.
4. Промежуточные приложения нельзя использовать. Они не будут работать.
5. Отсутствие гибкости моделирования уникальных систем.
6. Проблемы, связанные со сборкой, обнаруживаются на завершающих этапах разработки.
7. Отсутствие гарантий качестве главного (итогового) продукта до того момента, как завершится «жизнь» проекта. Пока он не будет «собран», неизвестно, что получится в итоге.
8. Каждая фаза – это предпосылка для выполнения последующих операций. Это повышает риски для систем без аналогов. Связан соответствующий момент с отсутствием гибкого моделирования.

Каскадные модели жизненного цикла имеющегося ПО неплохо подходят для небольших проектов. В больших приложениях их реализовать можно, но сделать это весьма проблематично.

Область применения

Данный вариант сильно ограничен в области применения. Связано это с ее непосредственными недостатками. Эффективное применение cascade model обосновано, если:

1. Речь идет о программах с четко сформированным технических заданием. Все его методики обладают понятной реализацией.
2. Подразумевается разработка ПО, ориентированного на построение приложения, которое схоже с уже реализованным. Пример – новый музыкальный проигрыватель.
3. Необходимо выпустить новую версию продукта или перенести его на другую платформу.

А еще соответствующий вариант подойдет для небольших программ. Обычно они быстро пишутся, а в разработке ПС участвует или небольшая команда, или вовсе один человек.

Инкрементный вариант

Рассматривая модели жизненного цикла разработки ПО, нужно обратить внимание на инкрементный подход. Это понятие включает в себя поэтапное создание ПС с промежуточным контролем.

Ведется она итерациями с циклами обратной связи между этапами. Промежуточные корректировки позволяют учесть реально существующие факторы и взаимовлияние результатов разработки на тех или иных этапах ЖЦ, время жизни каждого шага растягивает на весь период создания программы.

Здесь происходит следующее:

1. Начинается работа над проектом. Определяются ключевые системные требования. Они делятся на более и менее важные.

2. Ведется проектирование системы по принципу приращений. Делается это так, чтобы полученные результаты можно было задействовать в будущем при написании приложения. Каждый инкремент – это определенная функциональность. Выпуск начинается с наиболее значимого элемента.
3. Когда основная часть проекта готова, проводится ее детализация. В это время можно уточнить требования для других фрагментов программы, которые в текущей связи требований заморожены. При необходимости к ним разрешено вернуться позже.
4. Если часть программы готова, ее можно использовать в работе. Это помогает уточнять требования для оставшихся элементов.
5. Ведется «программирование» остальных компонентов проекта.
6. Приложение совершенствуется до того момента, пока ТЗ не будет реализовано во всей своей полноте.

Если провести сравнение с «каскадом», то инкрементный подход используется в сложных и комплексных системах. Это – «программирование версиями».

Особенности

К сильным сторонам incremental process относят:

- минимизирование затрат на итоговый релиз и исправление ошибок;
- получить отзывы от клиентов намного проще – тут хорошо налажена обратная связь;
- клиент сможет быстро получить основной проект, готовый к эксплуатации.

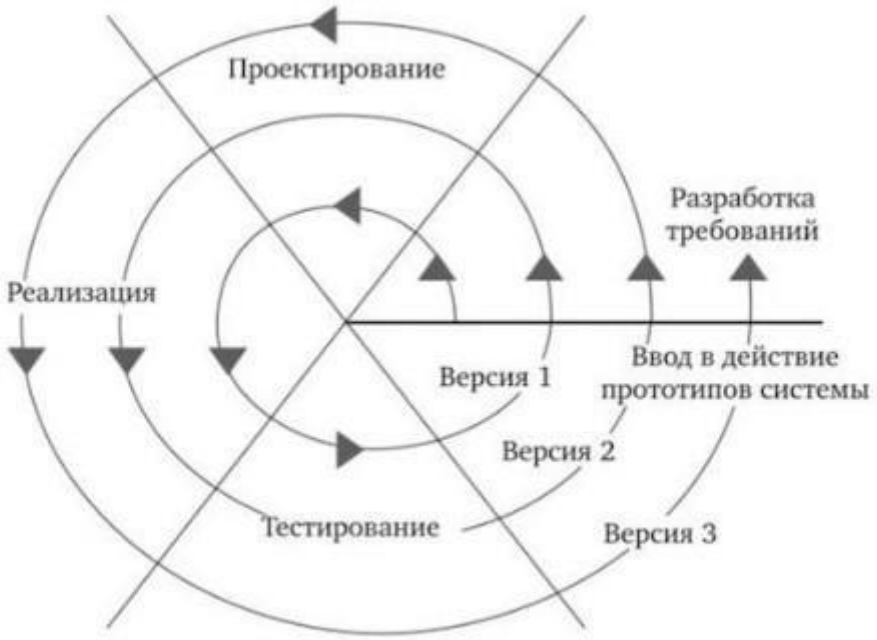
Недостатки:

- по мере внедрения новых элементов в итоговое приложение наблюдается тенденция ухудшению его итоговой работы;
- нужно постоянно измерять прогресс операций;
- если структура ПС не продумана, то исправления в будущем обойдутся дорого.

Также здесь отсутствует возможность оперативного реагирования на изменения и уточнения требований к итоговому ПО.

Спиральный тип

Типы вариантов программирования приложений разнообразны. Среди них есть так называемая спиральная концепция. Это понятие включает в себя ситуацию, при которой на каждом витке выполняется создание новой версии приложения. Далее происходит уточнение требований, определение качества ПС и ведется планирование следующего этапа. Первым шагам (анализу и проектированию) уделяется больше всего времени.



Этот вариант сочетает в себе проектирование и постадийное прототипирование. Особое внимание здесь уделяется оценке возможных рисков. Написание проекта осуществляется итерационно. Каждый этап опирается на предыдущий. Очередной шаг начинается тогда, когда решение относительно дальнейшей судьбы первоначальной идеи уже принято.

Сильные и слабые стороны

Spiral processes имеет такие преимущества:

1. Дает возможность быстрее показать работоспособный продукт.
2. Предусматривает изменение требований при написании ПО.
3. Здесь предусматривается гибкое проектирование. Итерации тоже не запрещены.
4. Итоговый продукт получается более стабильным и надежным. Слабые места команда программистов должна определить на каждой конкретной итерации.
5. Пользователи могут провести сравнение версий и поучаствовать в непосредственной доработке проекта. Обратная связь с целевой аудиторией налажена по максимуму.

Недостатки:

1. Реализация проекта может обойтись дорого. Особенно если он имеет низкую степень риска или небольшие размеры.
2. Жизненный цикл программного обеспечения имеет усложненную структуру. Это приводит к затруднению ее применения заказчиками и менеджерами.
3. Спираль может длиться бесконечно долго.

4. Промежуточные циклы в больших количествах приводят к созданию дополнительной документации.
5. Проблематично определить цели и стадии, указывающие на готовность продукта.

Такой вариант обычно применяется в инновационных технологиях, а также при выпуске новых серий систем и долгосрочных проектов.

Итеративный вариант

Виды (концепции) разработки могут быть итеративными. Этот вариант наиболее прост для заказчика, и сложен для программистов. Тут этапы жизненного цикла желаемого программного обеспечения не определены «от начала до конца». Клиент знает, что он хочет получить в конечном итоге. А вот как – дело разработчиков. Известна первоначальная идея, реализация не имеет детализации. Пример: клиент хочет выпустить собственную социальную сеть, но что там конкретно будет, как она будет выглядеть – нет.

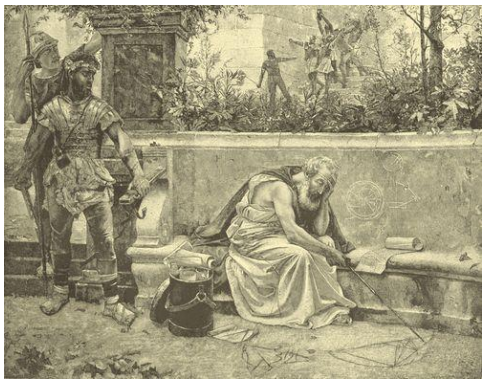
Преимуществом становится быстрый старт с минимальным функционалом. Этот момент позволяет оперативно получать обратную связь от пользователей. А еще продукт постоянно тестируется.

Фиксированного бюджета у такого проекта не будет, как и определенного бюджета. Концепция подойдет для масштабных приложений инновационного характера. А еще могут возникнуть проблемы с этапами жизненного цикла ПО, если не получается найти общий язык с заказчиком.

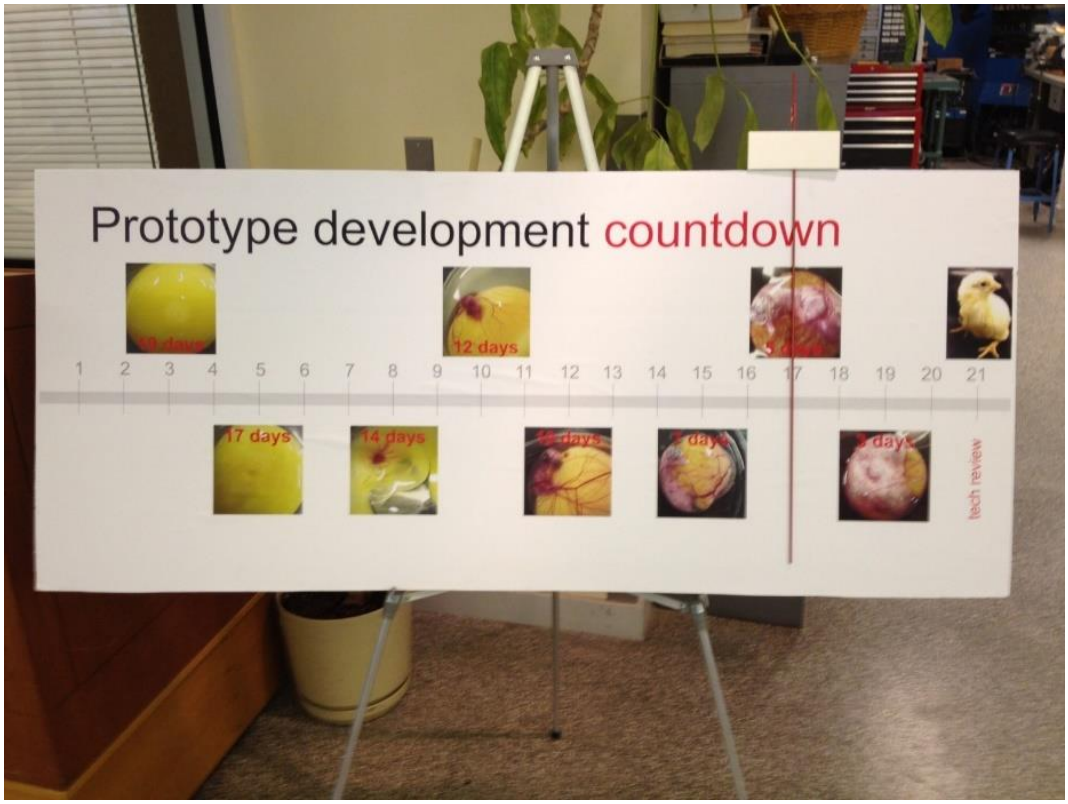
1.5.1. Проектирование программных продуктов

1.5.2. Проектирование программных продуктов

Если бы мы запланировали статью, которая не будет никому интересна, то наверное написали про важность проектирования зданий перед их постройкой. Но, к счастью, любой человек понимает, почему не стоит строить дома на глазок, добавляя фичи прямо в процессе строительства. При разработке же программного обеспечения по-прежнему полезно напоминать о том, что начинать её следует с проектирования — т.е. с полного планирования того, что непосредственно нам придётся разрабатывать, в какие сроки, с какими исходными данными и ожидаемым результатом.



За 13 лет опыта компании «Эдисон» в аутсорс-разработке для средних и крупных компаний из России, США, Европы и Австралии мы выработали собственную схему проектирования ПО, о которой в этом посте и расскажем.



Зачем нужно проектирование программного обеспечения

Определив требования к программному обеспечению, разработчик получает согласованный четкий план действий, график оплат и сроков, сокращает время разработки и повышает её качество, а также позволяет предусмотреть любые другие нюансы разработки, например, юридические (в частности по передаче авторских прав на программное обеспечение).

Проектируя ПО заранее, разработчик получает возможность:

- оценить стоимость и время разработки программного продукта,
- исключить потери времени и денег на ненужные действия, вынужденные доработки, длительное согласование,
- избежать разногласий и неудовлетворённости клиента и исполнителя.

Подготовительный этап

В зависимости от особенностей проекта порядок разработки программного обеспечения может отличаться, но в общем виде он такой:

При подготовке к проектированию решаются организационные вопросы:

- что клиент может предоставить (ТЗ, макеты, дизайн), насколько достаточны исходники и какие этапы закрывают — таким образом определяется состав работ,
- бюджет и сроки: на основе имеющихся материалов утверждается примерная стоимость, срок всего проекта, а также срок и точная стоимость ближайшего этапа.

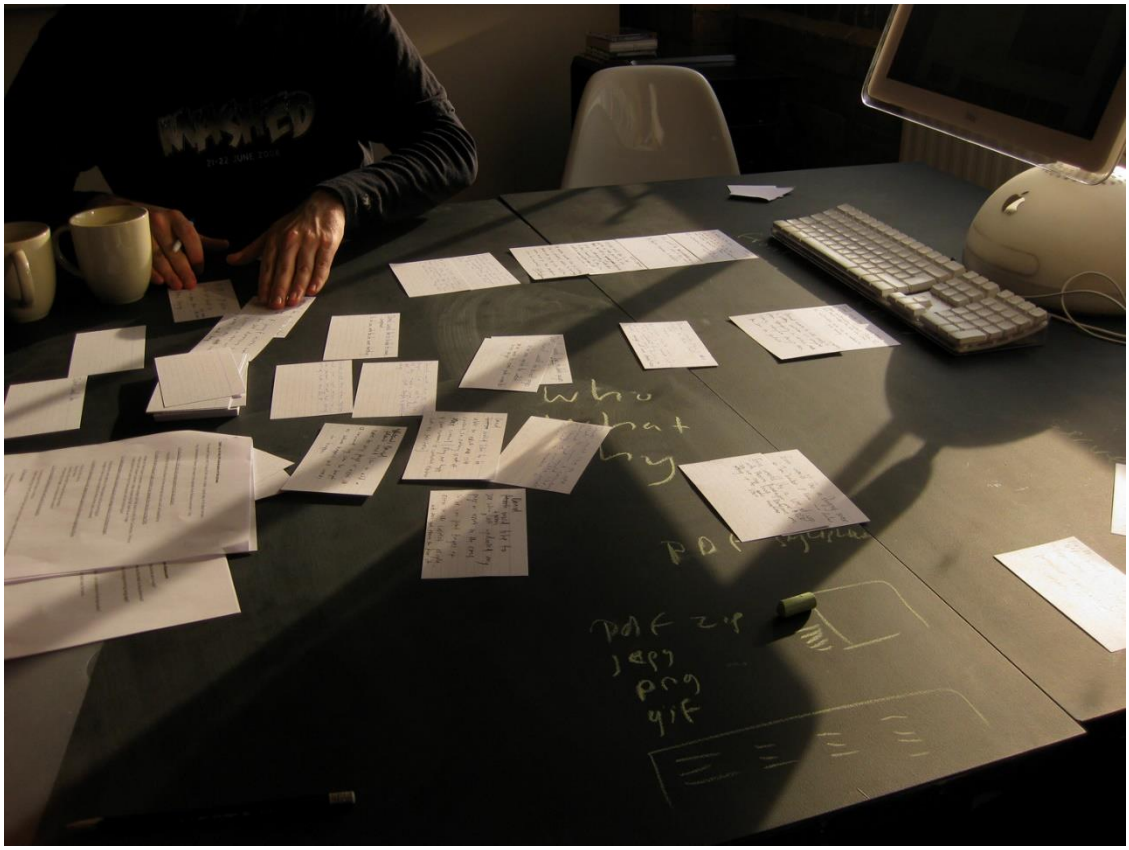
Теперь можно подписывать контракт, получать предоплату и все необходимые для работы материалы.

Этапы и результаты проектирования

1. Описание: совместная работа заказчика (говорит о пользе продукта, требованиях к работоспособности и внешнему виду) и EDISON (предлагает технические и алгоритмические решения).
2. Архитектура: утверждается язык программирования, база данных, серверы и фреймворки.
3. Техническое задание: составляется архитектором на основании описания и ответов заказчика на вопросы, согласовывается с менеджером проекта, затем передается клиенту, производятся правки.
4. Макеты (добавляются к техзаданию): интерфейсов, принципиальные схемы устройства, диаграммы структуры базы данных, схемы взаимодействия компонентов.
5. Контроль: архитектор устраняет замечания менеджера проектов.
6. Утверждение: заказчик проверяет и меняет ТЗ самостоятельно или сообщает список правок проект-менеджеру, замечания устраняются, ТЗ утверждается и прилагается к контракту.

Как результат проектирования, мы получаем техническое задание с понятной и однозначной для заказчика и исполнителя (руководителя проекта, программистов, тестировщиков, дизайнеров и других участников процесса разработки) иллюстрацией ответов на вопросы:

1. Что делаем (описание продукта, функционала, пользователей)?
2. Как делаем (архитектура)?
3. Как проверить, что цель достигнута (тестирование, критерии оценки)?



Примеры техзаданий на разработку ПО

Естественно, чем сложнее проект, тем дольше и дороже подготовка к нему. Проектирование небольших проектов занимает от недели до месяца. Чтобы процесс шёл быстрее и стоил меньше, мы предоставляем заказчикам по запросу инструкцию по составлению ТЗ и примеры готовых технических заданий. Приведем примеры и тут.

ТЗ на программное обеспечение Protector

Объект ТЗ: разработка и интеграция с существующей системой модульного ПО для мониторинга удаленных устройств охраны
Заказчик: ООО «ВТИМБ»

Сценарии использования образовательной системы

Объект ТЗ: создание образовательной системы

ТЗ на разработку ПО SMPP-шлюз

Объект ТЗ: разработка программного обеспечения SMPP-шлюза
Заказчик: IMT

В ходе разработки ТЗ, как в последнем кейсе, мы обязательно визуализируем основные моменты в виде схем, диаграмм, моделируем бизнес-процессы, создаем макеты интерфейсов, по желанию клиента выполняем ТЗ на русском или английском языках.

1.6.1. Структура и формат статические и динамические данные

В этой статье мы узнаем о **статической структуре данных** . Прежде чем приступить к работе со статической структурой данных, давайте кратко рассмотрим, что такое **структура данных** и какая структура данных имеет статический тип.

Структура данных: компьютер может содержать большое количество данных, поэтому **структуры данных** . помогает организовать эти данные и хранить их определенным образом, чтобы мы могли эффективно выполнять с ними множество операций. Проще говоря, структуры данных используются для уменьшения сложности (в основном временной сложности) кода.

По сути, **структура данных** бывает двух типов:

1. Статические структуры данных
2. Динамические структуры данных

Давайте также получим их краткий обзор.

Статическая структура данных. Статическая структура данных — это организация или набор данных в памяти, которые имеют фиксированный размер, то есть в ней может храниться ограниченное количество элементов или данных. **Массив** является примером статической структуры данных.

Динамическая структура данных. Динамическая структура данных — это организация или набор данных в памяти, которые не имеют фиксированного размера, то есть ее размер может быть изменен во время выполняемых над ней операций и может хранить переменное количество элементов или данных в памяти. это. Несколько примеров динамической структуры данных: ArrayList, LinkedList.

В этой статье мы собираемся подробно обсудить статические структуры данных. Итак, давайте теперь начнем с основной темы нашей статьи — статической структуры данных.

Что такое статическая структура данных?

Статическая **структура данных** — это организация или набор данных в памяти, которые имеют фиксированный размер, то есть могут хранить в себе ограниченное количество элементов или данных. Максимальный объем данных, которые будут храниться в статической структуре данных, должен быть известен до объявления структуры данных, а ее размер фиксирован в памяти, то есть мы не можем перераспределить для нее дополнительную память позже, хотя содержимое структуры данных может быть модифицированный.

На изображении выше мы видим, как фиксируется размер **статической структуры данных .** Как только структура данных выделена в памяти фиксированного размера, мы не можем изменить выделенное ей пространство памяти. Хотя содержимое структуры данных можно изменить.

Давайте посмотрим на некоторые структуры данных, которые являются статическими по своей природе и имеют фиксированные размеры, расположенные в памяти.

Пример статической структуры данных

Одним из наиболее ярких примеров статической структуры данных является **Array** . Давайте обсудим это подробно.

Множество

Массив — это объект-контейнер или коллекция, в которой хранится фиксированный объем данных одного типа в смежных ячейках памяти. Под «похожим типом» мы подразумеваем, что элементы массива должны иметь один и тот же тип данных в Java, то есть они однородны по своей природе. Длина массива определяется во время его объявления, и эта длина выделяется в памяти, которая фиксирована и не может быть изменена позже. Мы не можем перераспределить дополнительную память для массива, если он инициализирован фиксированным размером. Следовательно, мы можем сказать, что массив — это **статическая структура данных** .

Массивы в Java не похожи на массивы в C/C++. В Java размер массива фиксирован, тогда как в C++ он не фиксирован. В Java, если мы выполним какую-либо операцию с индексом, который не связан с массивом, будет выдано сообщение об ошибке, указывающее, что **индекс находится за пределами границ** , тогда как в C/C++ такое сообщение об ошибке не будет выдано. В C++ можно создать массив полностью построенных объектов за одну операцию, тогда как в Java для создания массива объектов требуется несколько операций. Как и в C++, для создания массива объектов используется следующий синтаксис имя_класса имя_массива [размер] ; и создается массив объектов, но в Java нам сначала нужно объявить массив объектов с синтаксисом Class_Name[ ] objectArrayReference; , то нам нужно создать экземпляр массива объектов с синтаксисом Class_Name obj[ ]= new Class_Name[Array_Length]; , то нам нужно инициализировать массив объектов, используя либо конструкторы, либо отдельный метод-член, и массив объектов будет создан.

На изображении выше мы можем видеть, как данные распределяются в памяти в смежных ячейках памяти.

Давайте посмотрим на синтаксис структуры данных массива.

В приведенном выше синтаксисе мы инициализируем массив с помощью **нового** ключевого слова, и с помощью **нового** ключевого слова структура данных массива будет инициализирована в куче памяти. Мы видим, как мы заранее задаем фиксированный размер массива для хранения в памяти. Следовательно, мы можем сказать, что массив — это статическая структура данных с фиксированным размером в памяти.

Особенности статических структур данных

Статические структуры данных имеют множество особенностей. Давайте посмотрим на основные особенности статических структур данных:

- Нам не нужно явно хранить структурную информацию вместе с элементом. Он часто хранится в различных физических и логических заголовках.

Статическая структура данных	**Динамическая структура данных**
Статическая структура данных имеет фиксированный размер памяти, и ее размер не может быть случайно обновлен во время выполнения.	Динамическая структура данных не имеет фиксированного размера памяти, и ее размер может произвольно обновляться во время выполнения.
Память выделяется для структуры данных во время компиляции. Исправленный размер.	Память выделяется для структуры данных динамически, то есть во время выполнения программы.
Статическая структура данных обеспечивает более простой доступ к элементам через их индексы.	Динамическая структура данных не обеспечивает более легкий доступ к элементам. поскольку распределение памяти происходит не подряд. Однако структуры данных, такие как **ArrayList** , являются динамическими по своей природе и обеспечивают легкий непрерывный доступ через свои индексы.
В статической структуре данных вместе с данными нам не нужно хранить информацию об адресе памяти следующих или предыдущих элементов. Поскольку данные располагаются в памяти непрерывным образом, мы можем напрямую обращаться к следующим или предыдущим элементам с помощью индексов.	В динамических структурах данных, таких как **LinkedList,** нам необходимо хранить информацию об адресах памяти следующих или предыдущих элементов. Поскольку данные не располагаются в памяти непрерывно, мы не можем напрямую получить доступ к их следующим или предыдущим элементам без информации об адресе.

Распределение памяти фиксировано, поэтому проблем с добавлением и удалением элементов данных не возникнет. До тех пор, пока заданный размер не превысит свободное непрерывное пространство, оставшееся в памяти.	Поскольку распределение памяти в динамической структуре данных не фиксировано, может возникнуть ошибка **переполнения** , если предел превышен и мы пытаемся добавить к нему еще элементы. Также выдаст ошибку **переполнения** , если динамическая структура данных пуста, и мы пытаемся удалить из нее элемент.
Может быть очень неэффективно, поскольку память для структуры данных выделяется независимо от того, нужна она или нет во время выполнения программы.	Динамическая структура данных использует память наиболее эффективно, поскольку она использует ее ровно столько, сколько необходимо.
Статическая структура данных не такая гибкая, как динамическая структура данных.	Динамическая структура данных не такая гибкая, как статическая структура данных.
В случае статической структуры данных программировать проще, поскольку нет необходимости проверять размер структуры данных в любой момент.	В случае динамической структуры данных ее сложно кодировать, поскольку нам необходимо постоянно отслеживать ее размер и расположение данных.

- Обычно элементы любой статической структуры данных хранятся в памяти непрерывно, поскольку они удерживаются в одном разделе памяти.
- В зависимости от определения структуры данных определяется вся описательная информация, кроме физического расположения выделенного элемента в памяти.
- Вся описательная информация, кроме физического местоположения выделенного элемента в памяти, определяется определением его структуры данных.
- Отношения между элементами любой статической структуры данных остаются неизменными на протяжении всего срока службы структуры.

Статическая структура данных против динамической структуры данных

Давайте теперь посмотрим на некоторые различия между **статической структурой данных** и **динамической структурой данных** .

Преимущества и недостатки статических структур данных

Ниже приведены преимущества статических структур данных:

- Ключевое преимущество статических структур данных заключается в том, что, поскольку распределение памяти фиксировано, нам не нужно контролировать или контролировать потенциальные проблемы переполнения или недостаточного заполнения при добавлении новых элементов или удалении существующих. До тех пор, пока заданный размер не превысит свободное непрерывное пространство, оставшееся в памяти.
- Его очень легко программировать, поскольку нет необходимости в любой момент проверять размер структуры данных.
- Компилятор и он выделяет место во время компиляции.
- Статическая структура данных обеспечивает более простой доступ к элементам через их индексы.

- Также легко проверить ситуацию переполнения в случае статической структуры данных.

Ниже приведены недостатки статических структур данных:

- Нам нужно оценить максимальный объем места, необходимый для хранения элемента в памяти.
- Мы не можем перераспределить дополнительное пространство в памяти, как только статическая структура данных инициализируется с фиксированным размером в памяти.
- Много места может оказаться потраченным впустую. Если они остаются неиспользованными в Статической структуре данных (если размер слишком велик).
- Статическая структура данных не такая гибкая, как динамическая структура данных.

Заключение

В этой статье мы узнали о статической структуре данных. Подведем итоги, которые мы обсуждали в статье:

- Структура данных — это набор данных, который помогает организовать эти данные и хранить их определенным образом, чтобы мы могли эффективно выполнять с ними множество операций.
- По сути, **структура данных** бывает двух типов: **статические структуры данных** и **динамические структуры данных** .
- Статическая **структура данных** — это организация или набор данных в памяти, которые имеют фиксированный размер, то есть в ней может храниться ограниченное количество элементов или данных.
- Динамические **структуры данных** — это организация или совокупность данных в памяти, которые не имеют фиксированного размера, то есть их размер может быть изменен во время выполняемых над ними операций и в них может храниться переменное количество элементов или данных.
- **Массивы** являются ярким примером статической **структуры данных** .
- Массив — это объект-контейнер или коллекция, в которой хранится фиксированный объем **данных** одного типа.
- Мы обсудили некоторые особенности **статических структур данных** .
- Мы также видели некоторые различия между **статической структурой данных** и **динамической структурой данных** .
- Ключевое преимущество **статических структур данных** заключается в том, что, поскольку распределение памяти фиксировано, нам не нужно контролировать или контролировать потенциальные проблемы переполнения или недостаточного заполнения при добавлении новых элементов или удалении существующих.
- Недостаток **статических структур данных** заключается в том, что много места может быть потрачено впустую. Если они остаются неиспользованными в статической структуре данных.

2.1.1. Методы проектирования ПП

2.1.2. Методы проектирования ПП

Проектирование алгоритмов и программ - наиболее ответственный этап жизненного цикла программных продуктов, определяющий, насколько создаваемая программа соответствует спецификациям и требованиям со стороны конечных пользователей. Затраты на создание, сопровождение и эксплуатацию программных продуктов, научно-технический уровень разработки, время морального устаревания и многое другое - все это также зависит от проектных решений.

Методы проектирования алгоритмов и программ очень разнообразны, их можно классифицировать по различным признакам, важнейшими из которых являются:

- степень автоматизации проектных работ;

- принятая методология процесса разработки.

По *степени автоматизации* проектирования алгоритмов и программ можно выделить:

- методы традиционного (неавтоматизированного) проектирования;

- методы автоматизированного проектирования (CASE-технология и ее элементы).

Неавтоматизированное проектирование алгоритмов и программ преимущественно используется при разработке небольших по трудоемкости и структурной сложности программных продуктов, не требующих участия большого числа разработчиков. Трудоемкость разрабатываемых программных продуктов, как правило, небольшая, а сами программные продукты имеют преимущественно прикладной характер.

При нарушении этих ограничений заметно снижается производительность труда разработчиков, падает качество разработки, и, как ни парадоксально, увеличиваются трудозатраты и стоимость программного продукта в целом.

Автоматизированное проектирование алгоритмов и программ возникло с необходимостью уменьшить затраты на проектные работы, сократить сроки их выполнения, создать типовые "заготовки" алгоритмов и программ, многократно тиражируемых для различных разработок, координации работ большого коллектива разработчиков, стандартизации алгоритмов и программ.

Автоматизация проектирования может охватывать все или отдельные этапы жизненного цикла программного продукта, при этом работы этапов могут быть изолированы друг от друга либо составлять единый комплекс, выполняемый последовательно во времени. Как правило, автоматизированный подход требует технического и программного "перевооружения" труда самих разработчиков (мощных компьютеров, дорогостоящего программного инструментария, а также повышения квалификации разработчиков и т.п.).

Автоматизированное проектирование алгоритмов и программ под силу лишь крупным фирмам, специализирующимся на разработке определенного класса программных продуктов, занимающих устойчивое положение на рынке программных средств.

Проектирование алгоритмов и программ может основываться *на различных подходах,* среди которых наиболее распространены:

1. Структурное проектирование программных продуктов.

2. Информационное моделирование предметной области и связанных с ней приложении.

3. Объектно-ориентированное проектирование программных продуктов.

1. В основе ***структурного проектирования*** лежит последовательная декомпозиция, целенаправленное структурирование на отдельные составляющие. Начало развития структурного проектирования алгоритмов и программ падает на 60-е гг. Методы структурного проектирования представляют собой комплекс технических и организационных принципов системного проектирования.

Типичными методами структурного проектирования являются:

- нисходящее проектирование, кодирование и тестирование программ;

- модульное программирование;

- структурное проектирование (программирование) и др.

В зависимости от объекта структурирования различают:

- функционально-ориентированные методы - последовательное разложение задачи или целостной проблемы на отдельные, достаточно простые составляющие, обладающие функциональной определенностью;

- методы структурирования данных.

Для функционально-ориентированных методов в первую очередь учитываются заданные функции обработки данных, в соответствии с которыми определяется состав и логика работы (алгоритмы) отдельных компонентов программного продукта. С изменением содержания функций обработки, их состава, соответствующего им информационного входа и выхода требуется перепректирование программного продукта. Основной упор в структурном подходе делается на моделирование процессов обработки данных.

Для методов структурирования данных осуществляется анализ, структурирование и создание моделей данных, применительно к которым устанавливается необходимый состав функций и процедур обработки. Программные продукты тесно связаны со структурой обрабатываемых данных, изменение которой отражается на логике обработки (алгоритмах) и обязательно требует перепроектирования программного продукта.

Структурный подход использует:

- диаграммы потоков данных (информационно-технологические схемы) - показывают процессы и информационные потоки между ними с учетом "событий", инициирующих процессы обработки;

- интегрированную структуру данных предметной области (инфологическая модель, ER-диаграммы);

- диаграммы декомпозиции - структура и декомпозиция целей, функций управления, приложений;

- структурные схемы - архитектура программного продукта в виде иерархии взаимосвязанных программных модулей с идентификацией связей между ними, детальная логика обработки данных программных модулей (блок-схемы).

2.2.1. Структура ПП

2.2.2. Структура ПП

Таким образом, в большей степени программные продукты не являются «монолитом» и имеют *конструкцию* (архитектуру) построения – состав и взаимосвязь программных модулей.

Программный продукт обладает *внутренней организацией,* или *внутренней структурой,* образованной взаимосвязанными программными модулями. Это справедливо для сложных и многофункциональных программных продуктов, которые часто называются *программными системами.*

При создании программных продуктов выделяются многократно используемые модули, проводится их типизация и унификация, за счет чего сокращаются сроки и трудозатраты на разработку программного продукта в целом.

Некоторые программные продукты используют модули из готовых библиотек стандартных подпрограмм, процедур, функций, объектов, методов обработки данных.

На рис. приведена типовая структура программного продукта, состоящего из отдельных программных модулей и библиотек процедур, встроенных функций, объектов и т. п.

Р и с Структура программного продукта

Среди множества модулей различают:

- головной модуль – управляет запуском программного продукта (существует в единственном числе);
- управляющий модуль – обеспечивает вызов других модулей на обработку;
- рабочие модули – выполняют функции обработки;
- сервисные модули и библиотеки, утилиты – осуществляют обслуживающие функции.

В работе программного продукта активизируются необходимые программные модули. Управляющие модули задают последовательность вызова на выполнение очередною модуля. Информационная связь модулей обеспечивается за счет использования общей базы данных либо межмодульной передачи данных через переменные обмена.

Каждый модуль может оформляться как самостоятельно хранимый файл. Для функционирования программного продукта необходимо наличие программных модулей в полном составе.

Программы произвольных размеров и сложности могут быть написаны на основе ограниченного множества базисных структур. Этот принцип положен в основу проектирования схем, где любая логическая структура может быть создана из элементарных структур:

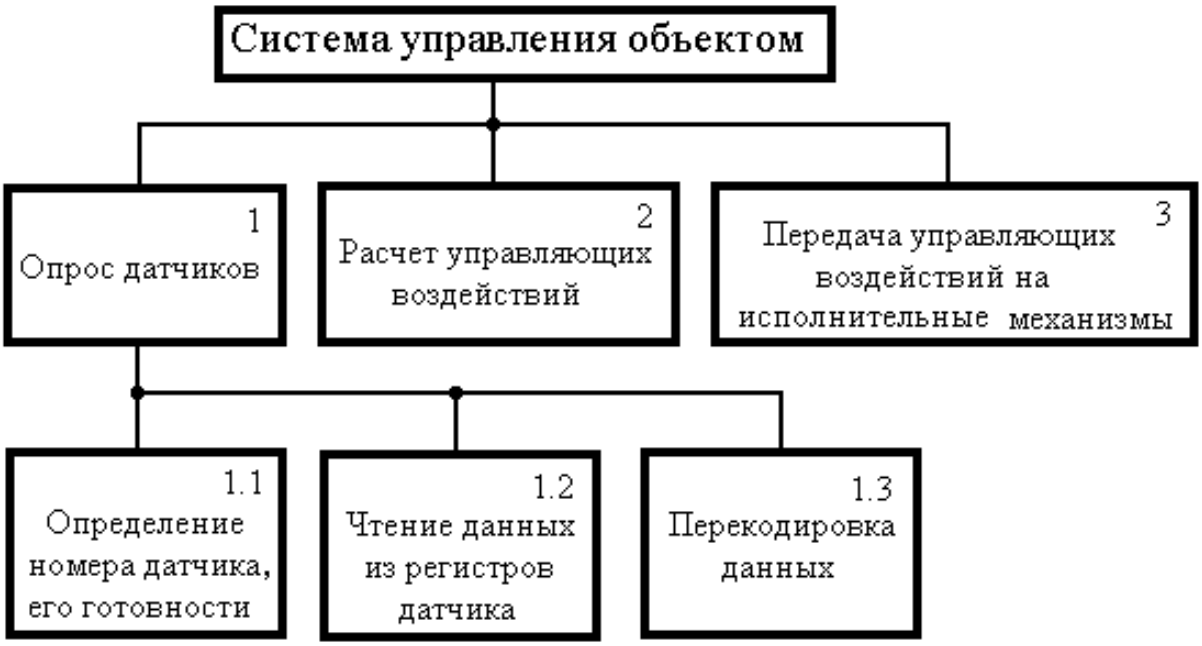
1. *структура последовательности* – это формализация того, что операторы программы выполняются в порядке их появления в программе, пока что-то не изменит их последовательность.
2. *структура выбора* – это выбор одного из двух действий исходя из выполненного некоторого условия.
3. *структура повторения* – используется для повторного выполнения группы команд до тех пор, пока не выполниться некоторое условие.

Получение правильной программы путем замены операторов программы на управляющие логические структуры называется вложением структур.

Простота исходных конструкций структурного программирования предотвращает появление сложных информационных связей и запутанных передач управления.

Структурированными считаются программы, которые не имеют циклов с несколькими выходами, не имеют переходов внутрь циклов или условных операторов и не имеют выходов из внутренней части циклов или условных операторов.

Любое ПО можно представить в виде иерархии модулей. Модули верхнего уровня выполняют более общие функции и вызывают модули нижних уровней, в которых детализируется решение задачи. Например, при работе программы системы управления объекта, показанной на рис. 7, поочередно вызываются модули 1, 2, 3. В свою очередь модуль 1 вызывает модули нижнего уровня (1.1; 1.2; 1.3).



Р и с. 7. Пример иерархии программных модулей

Модули нижних уровней или одного уровня иерархии могут вызываться для исполнения только модулями высших уровней, т.е. модули нижних уровней не могут вызывать модули высших уровней, а модули одного уровня – вызывать друг друга.

Информация зон глобальных переменных доступна для использования любыми модулями, входящих в комплекс программ или группу программ, в соответствии с областью действия зоны глобальных переменных, т. е. глобальные переменные могут быть доступны не для всего комплекса программ, а лишь для указанной в описании группы модулей. Локальные переменные доступны лишь в пределах того модуля, в котором они определены. Для взаимодействия вызываемых и вызывающих модулей создаются зоны обменных переменных, информация из которых доступна лишь модулям непосредственно связанных по управлению.

2.3.1. Проектирование интерфейса пользователя

2.3.2. Проектирование интерфейса пользователя

Пользовательский интерфейс имеет важное значение для любой программной системы и является неотъемлемой ее составляющей, ориентированной, прежде всего, на конечного пользователя. Именно через интерфейс пользователь судит о прикладной программе в целом. Более того, часто решение об использовании прикладной программы пользователь принимает по тому, насколько ему удобен и понятен пользовательский интерфейс. Вместе с тем, трудоемкость проектирования и разработки интерфейса может быть достаточно велика, и достигать более половины общего времени реализации проекта.

Основным предназначением системы является предоставление пользователю необходимой функциональности. Поэтому разработку интерфейса следует реализовать в следующей последовательности:

- определение перечня основных функций системы, которые должны быть отражены в интерфейсе;

- определение перечня окон, их предназначение и общее содержимое;
- схематичное отображение детального содержимого каждого окна.

Кроме того, при разработке интерфейса пользователя следует придерживаться следующих критериев качества:

1) Удобство и интуитивность (привычные названия, возможность самостоятельного изучения и использования функций системы, легкость работы с системой).

2) Единообразие (предпочтителен стандарт, принятый в операционной системе, недопустимо использование одинаковых функционально, но различных внешне элементов).

3) Отсутствие перегруженности (небольшое число объектов на экране – не более 10).

4) Устойчивость (по возможности предотвращение некорректных действий пользователя).

В результате я предлагаю следующий интерфейс АИС «торговая компания»:

На этапе анализа были сформулированы задачи, которые пользователи системы будут выполнять с использованием созданной БД. Перечислим задачи пользователя, для решения которых необходим пользовательский интерфейс разработанной АИС.

1. Создание справочника товаров – форма товар.
2. Создание справочника клиентов – форма клиент.
3. Создание справочника счетов – форма счет.
4. Ведение и изменение товара – форма ЖУРНАЛ ТОВАРОВ.
5. Введение и изменение клиентов – форма ЖУРНАЛ КЛИЕНТОВ
6. Введение и изменение счетов – форма ЖУРНАЛ СЧЕТОВ
7. Выбор отчетов – форма ОТЧЕТ.
8. Главная форма, меню программы – форма МЭЙН.

3. Экспериментальный раздел

3.1. Тестирование программы

Вероятно, одной из самых больших трудностей при разработке качественного ПО является обеспечение целостности и согласованности всех действий и требуемых результатов, в особенности при многочисленной команде разработчиков. Компании-производители коммерческого ПО стремятся повысить качество программных продуктов с помощью тестирования. Существуют специальные драйверы, автоматизирующие процесс тестирования разрабатываемого ПО. Также используется «бета-тестирование», при котором разработчики передают пользователям пробные предварительные версии разрабатываемых систем. При этом даже после распространения финальных версий своих программных продуктов производители коммерческого ПО продолжают искать и исправлять ошибки, выпуская «пакеты обновлений» и «патчи».

Таким образом, тестирование – один из основных инструментов обеспечения безотказной корректной работы ПО, в конечном итоге влияющим на общее качество и коммерческую конкурентоспособность программного продукта.

В практике программирования наиболее часто в роли метрики качества продукта выступает остаточная плотность ошибок, то есть плотность ошибок на тысячу строк кода или на одну функциональную точку.

Тестирование ПО – процесс поиска ошибок, заключающийся в выявлении отличий ожидаемых результатов работ ПО от фактических. Несмотря на разнообразие существующих подходов к тестированию ПО, в том числе с использованием средств автоматизации, следует признать, что тестирование сложных программных систем – это процесс в значительной степени творческий, не сводящийся к следованию строгим и чётким процедурам. При этом очевидно, что тестирование не позволяет полностью избавиться от ошибок в ПО, а лишь может позволить (при правильном планировании и добросовестном выполнении) существенно уменьшить их количество.

В общем виде тестирование предусматривает последовательное выполнение следующих этапов:

- разработку плана тестирования;
- разработку тестовых заданий;
- выполнение тестовых процедур;
- формирование заключения по результатам.

План тестирования должен содержать:

- описание объекта тестирования (система, клиентское приложение, оборудование) и тестовой среды (например, операционная система клиентского приложения);
- критерии начала тестирования (готовность тестовой платформы, законченность разработки требуемого функционала, наличие необходимой документации);
- критерии окончания тестирования (результаты тестирования удовлетворяют критериям качества продукта, выдержка определенного периода без изменения исходного кода приложения, выдержка определенного периода без появления новых ошибок);
- виды тестирования и их применение к тестируемому объекту (например, тестирование основных сценариев, тестирование с некорректными действиями пользователя, нагрузочное тестирование, тестирование аварийных ситуаций и т.п.);
- последовательность тестирования (подготовка, тестирование, анализ результатов);
- спецификацию тестирования (список функций и/или компонент тестируемой системы).

После подготовки плана тестирования разрабатывают тестовые задания (Test Cases)– совокупность шагов, конкретных условий и параметров, необходимых для проверки реализации тестируемой функции или её части. Тестовое задание может иметь структуру вида

<действие> > <ожидаемый результат> > <фактический результат>.

Очевидно, что возможны различные уровни детализации при разработке тестовых заданий. Целесообразно использовать такую детализацию, которая позволяет достичь разумного соотношения времени выполнения тестового задания к «тестовому покрытию».

После выполнения запланированных тестовых процедур следует подготовить заключение о результатах тестирования, позволяющее сделать вывод об устойчивости и корректности

работы при различных условиях (видах тестирования, тестовых заданиях) отдельных модулей и подсистем, а также системы в целом.

Основным требованием к такому заключению является то, что при внешней оценке оно должно позволить сделать вывод либо об успешном завершении этапа тестирования и возможности передачи разработанной системы в опытную эксплуатации, либо о необходимости ее доработки (с указанием – в какой части: подсистема, возможные причины и пути устранения).

В моей программе мы провели тестовые испытания:

- на входные данные – во всех полях ввода в АИС;
- на промежуточный результат – в местах, где используются формулы;
- на конечный результат;

В программном коде в виде комментариев обозначеныввсе проверки.

3.1.1. Стиль программирования

3.1.2. Стиль программирования

Мы следуем стандартным соглашениям по оформлению кода на Java. Мы добавили к ним некоторые правила:

1. Исключения: никогда не перехватывайте и не игнорируйте их без объяснения.
2. Исключения: не используйте обобщенные исключения, кроме кода в библиотеках, в корне стека.
3. Финализаторы: не используйте их.
4. Импорты: полностью уточняйте импорты.

Правила Java библиотек

Существуют соглашения, по поводу использования Java библиотек и инструментов для Android. В некоторых случаях соглашения могут быть изменены, например, в таких как использование старого кода, который, возможно, использует неодобренный паттерн или библиотеку.

Правила Java стиля

Программы гораздо проще поддерживать, когда все файлы имеют согласованный стиль. Мы следуем стандартному стилю программирования на Java, определенному Sun в их Code Conventions for the Java Programming Language, с несколькими исключениями и дополнениями. Данное руководство по стилю является подробным и всесторонним, а также широко используется Java сообществом.

В дополнение, мы обязываем использовать следующие правила для кода:

1. Комментарии/Javadoc: пишите их; используйте стандартный стиль.
2. Короткие методы: не пишите гигантских методов.
3. Поля: должны быть вверху файла, или прямо перед методом, который их использует.
4. Локальные переменные: ограничивайте область видимости.
5. Импорты: android; сторонние (в алфавитном порядке); java(x)
6. Отступы: 4 пробела, без табуляций.
7. Длина строки: 100 символов.
8. Имена полей: не public и не static поля начинаются с «m».
9. Фигурные скобки: открывающие фигурные скобки не находятся в отдельной строке.
10. Аннотации: используйте стандартные аннотации.
11. Сокращения: используйте сокращения как слова в именах, например, XmlHttpRequest, getUrl() и т.п.
12. Стиль TODO: «TODO: пишите описание здесь».
13. Согласованность: смотрите, что находится вокруг вас.

Правила языка Java

Не игнорируйте исключения

Возможно, вам захочется написать код, который игнорирует исключения, например:

```
void setServerPort(String value) {
   try {
      serverPort = Integer.parseInt(value);
   } catch (NumberFormatException e) { }
}
```

Никогда так не делайте. В то время как вы думаете, что ваш код никогда не столкнется с таким условием или, что неважно обрабатывать это условие, игнорирование исключений создает скрытые проблемы. Вы в принципе должны обрабатывать каждое исключение. Специфика в каждом конкретном случае зависит от ситуации.
Приемлимые альтернативы:

- Перебрасывайте исключения к вызывающему методу.

  ```
  void setServerPort(String value) throws NumberFormatException {
     serverPort = Integer.parseInt(value);
  }
  ```

- Выбрасывайте исключения, соответственно вашему уровню абстракции

```
void setServerPort(String value) throws ConfigurationException {
   try {
      serverPort = Integer.parseInt(value);
   } catch (NumberFormatException e) {
      throw new ConfigurationException("Port " + value + " is not valid.");
   }
}
```

- Перехватите ошибку и замените соответствующее значение в блоке catch{}

```
/** Set port. If value is not a valid number, 80 is substituted. */
void setServerPort(String value) {
   try {
      serverPort = Integer.parseInt(value);
   } catch (NumberFormatException e) {
      serverPort = 80;  // default port for server
   }
}
```

- Перехватите ошибку и выбросьте RuntimeException. Это опасно: делайте это только если вам все равно случится ли эта ошибка.

```
/** Set port. If value is not a valid number, die. */
void setServerPort(String value) {
   try {
      serverPort = Integer.parseInt(value);
   } catch (NumberFormatException e) {
      throw new RuntimeException("port " + value " is invalid, ", e);
   }
}
```

Заметьте, что изначальное исключение передается конструктору RuntimeException. Если вы используете компилятор Java 1.3, то опустите исключение.

- Если вы уверены в том, что игнорирование исключения в этом случае имеет место, то хотя бы прокомментируйте, почему вы так решили.

```
/** If value is not a valid number, original port number is used. */
void setServerPort(String value) {
   try {
      serverPort = Integer.parseInt(value);
   } catch (NumberFormatException e) {
      // Method is documented to just ignore invalid user input.
      // serverPort will just be unchanged.
   }
}
```

Не перехватывайте обобщенные исключения

Иногда бывает заманчиво полениться с обработкой исключений и написать что-то вроде этого:

```
try {
   someComplicatedIOFunction();        // may throw IOException
   someComplicatedParsingFunction();   // may throw ParsingException
   someComplicatedSecurityFunction();  // may throw SecurityException
   // phew, made it all the way
} catch (Exception e) {                // I'll just catch all exceptions
   handleError();                      // with one generic handler!
}
```

Вам не следует так делать. Суть в том, что возможно появление исключения, которого вы не ожидали и, в итоге, ошибка будет отлавливаться на уровне приложения. То есть, если кто-то добавит новый тип исключения, то компилятор не сможет вам помочь понять, что это другая ошибка.

Существуют редкие исключения из этого правила: определенный тестовый код, или код верхнего уровня, где вы хотите перехватывать все типы ошибок (для того, чтобы предотвратить их отображение в пользовательском интерфейсе, или чтобы продолжить какую-то пакетную задачу).

Альтернативы обобщенным исключениям:

- Перехватывайте каждое исключение отдельно в блоке catch, после одиночного try. Возможно это неудобно, но всё равно это предпочтительный способ для перехвата всех исключений.
- Измените ваш код для более гранулированной обработки ошибок с несколькими блоками try. Отделите IO от парсинга, обрабатывайте ошибки отдельно в каждом случае.
- Перебросьте исключение. Во многих случаях вам не нужно обрабатывать все исключения на текущем уровне, просто позвольте методу перебросить их.

Помните: исключения — ваши друзья! Не сердитесь, когда компилятор указывает на то, что вы их не отлавливаете.

Финализаторы

Что это: Финализаторы — это способ запускать программный код перед тем как объект собирается сборщиком мусора.
За: могут быть полезны при очистке, в особенности внешних ресурсов.
Против: нет никаких гарантий того, когда будет вызван финализатор, и, вообще, будет ли он вызван.

Решение: Мы не используем финализаторы. В большинстве случаев, всё то, что вам нужно от финализатора, вы сможете сделать при помощи обработки исключений. Если

вам действительно нужен финализатор, то объявите метод close() и задокументируйте, когда он точно будет вызываться.

Импорты

Групповой символ в импортах

Что это: Когда вы хотите использовать класс Bar из пакета foo, то есть два способа сделать это:

1. **import** foo.*;
2. **import** foo.Bar;

За #1: Потенциально уменьшает количество возможных операторов импорта.
За #2: Делает явным то, какой класс на самом деле используется. Делает код более удобочитаемым для тех, кто его поддерживает.

Решение: Используйте стиль #2 для импорта любого Android кода. Явное исключение делается для стандартных библиотек (java.util.*, java.io.*, и т.п) и для кода модульного тестирования (junit.framework.*).

Комментарии/Javadoc

Каждый файл должен иметь объявление об авторских правах в самом начале. Далее идут объявления операторов package и import, причем каждый блок разделяется пустой строкой. За ними следуют объявления класса или интерфейса. Опишите, что делает класс в Javadoc-комментариях.

```
/*
 * Copyright (C) 2010 The Android Open Source Project
 *
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 *      http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

package com.android.internal.foo;

import android.os.Blah;
```

```
import android.view.Yada;

import java.sql.ResultSet;
import java.sql.SQLException;

/**
 * Does X and Y and provides an abstraction for Z.
 */
public class Foo {
   ...
}
```

Каждый класс и нетривиальный public метод должен содержать Javadoc, по крайней мере с одной фразой, описывающей что он делает. Фраза должна начинаться с описательного глагола 3-го лица. Примеры:

```
/** Returns the correctly rounded positive square root of a double value. */
static double sqrt(double a) {
}

/**
 * Constructs a new String by converting the specified array of
 * bytes using the platform's default character encoding.
 */
public String(byte[] bytes) {
}
```

Вам не нужно описывать Javadoc для тривиальных get и set методов, таких как setFoo(), если ваш Javadoc говорит только «sets Foo». Если метод делает что-то более сложное (например, соблюдение неких ограничений, или если его действия имеют важный эффект вне его самого), тогда его обязательно нужно задокументировать. И если это не просто объяснение того, что означает Foo, то вам также следует его задокументировать.

Вообще, любой метод, который вы написали получает пользу от Javadoc, неважно public он или нет. Public методы являются частью API, и поэтому они требуют описания в Javadoc.

Для написания Javadoc'ов вам следует придерживаться Sun Javadoc conventions.

Короткие методы

Методы должны быть небольшими и решающими конкретную задачу настолько, насколько это возможно. Однако, понятно, что иногда большие методы бывают целесообразны, так что нет строгого ограничения на длину метода. Если метод превышает 40 строк, то вам, возможно, стоит подумать о том, можно ли его разбить на части, не нарушив структуры программы.

Локальные переменные

Область видимости локальных переменных должна сводиться к минимуму. Делая это, вы улучшаете читаемость и поддерживаемость кода, а также уменьшаете вероятность ошибок. Каждая переменная должна объявляться в самом глубоком блоке, который окружает все возможные места использования переменной.

Локальные переменные должны объявляться в том месте, где впервые необходимо её использовать. Почти каждая локальная переменная нуждается в инициализаторе. Если вы еще не знаете, как точно инициализировать переменную, то вам следует отложить её объявление, пока вы это не узнаете.

Существует одно исключение, касательно блока try-catch. Если переменная инициализируется при помощи оператора return метода, который выбрасывает проверяемое исключение, то она должна инициализироваться в блоке try. Если же переменная должна использоваться вне блока try, тогда она объявляется перед ним, неважно, знаете ли вы как её точно нужно инициализировать:

```
// Instantiate class cl, which represents some sort of Set
Set s = null;
try {
   s = (Set) cl.newInstance();
} catch(IllegalAccessException e) {
   throw new IllegalArgumentException(cl + " not accessible");
} catch(InstantiationException e) {
   throw new IllegalArgumentException(cl + " not instantiable");
}

// Exercise the set
s.addAll(Arrays.asList(args));
```

Но даже этот случай можно обойти при помощи инкапсуляции блока try-catch в методе.

```
Set createSet(Class cl) {
   // Instantiate class cl, which represents some sort of Set
   try {
      return (Set) cl.newInstance();
   } catch(IllegalAccessException e) {
      throw new IllegalArgumentException(cl + " not accessible");
   } catch(InstantiationException e) {
      throw new IllegalArgumentException(cl + " not instantiable");
   }
}

...

// Exercise the set
Set s = createSet(cl);
s.addAll(Arrays.asList(args));
```

Переменные в циклах должны объявляться внутри самого оператора, если только нет непреодолимой причины этого не делать.

```
for (int i = 0; i <= n; i++) {
   doSomething(i);
}

for (Iterator i = c.iterator(); i.hasNext(); ) {
   doSomethingElse(i.next());
}
```

Импорты

Порядок операторов импорта следующий:

1. Android импорты.
2. Сторонние импорты (com, junit, net, org).
3. java и javax.

Для полного соответствия настройкам IDE, импорты должны иметь следующий вид:

- Отсортированы по алфавиту внутри каждой группы.
- Заглавные буквы должны быть впереди букв нижнего регистра (например, Z перед a).
- Главные группы должны разделяться пустой строкой.

Почему?
Порядок такой, чтобы:

- Импорты, которые люди хотят видеть в первую очередь, находятся вверху (android).
- Импорты, которые люди хотят видеть в последнюю очередь, находятся внизу (java).
- Люди могут с легкостью использовать этот стиль.
- IDE может придерживаться этого стиля.

Отступы

мы используем 4 пробела для блоков. Мы никогда не используем табуляцию. Мы используем 8 пробелов для переноса строк, включая вызовы функций и присваивания, например правильно так:

```
Instrument i =
      someLongExpression(that, wouldNotFit, on, one, line);
```

а так неверно:

```
Instrument i =
   someLongExpression(that, wouldNotFit, on, one, line);
```

Названия полей

- Не static и не public имена начинаются с «m».
- static поля начинаются с «s».
- Другие поля начинаются с буквы нижнего регистра.
- Поля public static final (константы) пишутся полностью в верхнем регистре, с использованием подчеркивания (ALL_CAPS_WITH_UNDERSCORES)

Например:

```
public class MyClass {
   public static final int SOME_CONSTANT = 42;
   public int publicField;
   private static MyClass sSingleton;
   int mPackagePrivate;
   private int mPrivate;
   protected int mProtected;
}
```

Фигурные скобки

Для открывающих фигурные скобок не выделяется отдельная строка, они находятся в той же строке, что и код перед ними:

```
class MyClass {
   int func() {
      if (something) {
         // ...
      } else if (somethingElse) {
         // ...
      } else {
         // ...
      }
   }
}
```

Мы требуем фигурные скобки для оператора условия. Исключением является, когда

оператор условия и его тело помещаются в одну строку. То есть можно писать так:

```
if (condition) {
   body(); // ok
}
if (condition) body(); // ok
```

Но так нельзя:

```
if (condition)
   body(); // bad
```

Длина строки

Каждая строка текста в коде должна быть не длиннее 100 символов.
Исключение: если комментарий содержит пример команд, или URL (удобнее использовать copy/paste).
Исключение: строки импорта могут быть длиннее 100 символов, так как люди редко на них смотрят. Также это упрощает написание инструментов.

Сокращения в именах

Рассматривайте сокращения и аббревиатуры как слова. Имена более удобочитаемы:

Хорошо	Плохо
XmlHttpRequest	XMLHTTPRequest
getCustomerId	getCustomerID

Этот стиль также применяется, когда сокращение и аббревиатура — это полное имя:

Хорошо	Плохо
class Html	class HTML

String url; String URL;

long id; long ID;

Стиль TODO

Используйте комментарии TODO для кода, который является временным, краткосрочным, или хорошим, но не идеальным. Комментарий должен включать в себя «TODO:», например:

// TODO: Remove this code after the UrlTable2 has been checked in.

// TODO: Change this to use a flag instead of a constant.

Если ваш комментарий имеет вид «В будущем сделать что-то», то убедитесь, что он включает в себя конкретную дату (1 января 2011 года), или конкретное событие «Удалить после выхода версии 2.1».

Согласованность

Если вы изменяете код, то потратьте минуту на то, чтобы посмотреть на код вокруг вас и определить его стиль. Если в нем используются пробелы, то и вам следует их использовать. Если комментарии содержат небольшой набор звездочек, то и вам следует их использовать.

Весь смысл рекомендаций к стилю кода в создании общей лексики, чтобы люди концентрировались на том, что они говорят, вместо того как они говорят. Мы представляем глобальные правила стиля, чтобы люди знали эту лексику. Но локальный стиль также важен. Если код, который вы добавляете в файл выглядит резко отличным от того, что был, то это выбросит будущего читателя из его ритма и будет мешать ему понимать структуру. Старайтесь избегать этого.

3.2. Языки программирования

Выбор языка программирования может быть сложной задачей для новичка. В мире программирования существует огромное количество языков, каждый из которых имеет свои уникальные особенности и применение. В этой статье мы рассмотрим топ-10 самых популярных языков программирования по версии TIOBE Index.

1. Java

Java — это мощный, объектно-ориентированный язык программирования, который обладает большой платформой независимости. Java используется везде — от веб-разработки до мобильных приложений. Он также является основой для многих популярных фреймворков, таких как Spring и Hibernate.

2. C

C — это один из самых старых и наиболее широко используемых языков программирования. Он предлагает высокую степень контроля над системой и используется в разработке операционных систем и встроенных систем.

3. Python

Python — это высокоуровневый язык программирования, который стал очень популярным благодаря своей простоте и читаемости. Python широко используется в научных вычислениях, анализе данных, веб-разработке и машинном обучении.

4. C++

C++ — это язык программирования общего назначения, который предоставляет средства для низкоуровневого программирования. Он широко используется в разработке программного обеспечения, в том числе в играх, автоматизации производства и многом другом.

5. C

C# — это современный, объектно-ориентированный язык программирования, разработанный Microsoft. Он является основой .NET Framework и обычно используется для разработки Windows-приложений.

5. JavaScript

JavaScript — это язык программирования, который обычно используется в веб-разработке для создания интерактивных веб-страниц. С появлением Node.js JavaScript также стал популярен в серверной разработке.

7. PHP

PHP — это серверный язык программирования, который широко используется для веб-разработки. Большинство веб-сайтов работают на PHP, включая Facebook и WordPress.

8. Swift

Swift — это язык программирования, разработанный Apple для разработки iOS и macOS приложений. Swift быстро набирает популярность благодаря своей производительности и современному синтаксису.

9. Ruby

Ruby — это высокоуровневый язык программирования, который привлекает разработчиков своей простотой и элегантностью. Ruby является основой Ruby on Rails, популярного фреймворка для веб-разработки.

10. TypeScript

TypeScript — это строго типизированный суперсет JavaScript, который добавляет статическую типизацию и другие функции для улучшения масштабируемости и надежности кода.

Выбор языка программирования зависит от ваших целей и интересов.

3.3. Модульное программирование

С выходом JDK 9 в языке Java появилась новая возможность - модульность. Модульность позволяет разбить код на отдельные структурные единицы - модули. Фактически модуль представляет группу пакетов или ресурсов, объединенных в одно целое и к которым можно обращаться по имени модуля.

До Java 9 было несколько уровней инкапсуляции функционала. Первый уровень представлял класс, в котором мы могли определить переменные и методы с различным уровнем доступа. Следующий уровень представлял пакет, который, в свою очередь, представлял коллекцию классов. Однако со временем этих уровней оказалось недостаточно. И модуль стал следующим уровнем инкапсуляции, который объединял несколько пакетов.

Модуль состоит из группы пакетов. Также модуль включает список все пакетов, которые входят в модуль, и список всех модулей, от которых зависит данный модуль. Дополнительно (но необязательно) он может включать файлы ресурсов или файлы нативных библиотек.

В качестве названия модуля может использоваться произвольный идентификатор из алфавитно-цифровых символов и знаков подчеркивания. Но рекомендуется, чтобы название модуля соответствовало названию, которого начинаются пакеты этого модуля.

Определим и используем простейший модуль. Допустим, файлы с исходными кодами помещаются в папку C:\java (либо какую-нибудь другую папку на жестком диске). Создадим в этой папке каталог, который назовем **demo**. Этот каталог будет представлять модуль.

В каталоге demo определим новый файл **module-info.java** со следующим кодом:

```
module demo {
}
```

Этот файл представляет дескриптор модуля (module descriptor). Этот файл может содержать только определение модуля.

С помощью ключевого слова **module** определяется модуль, который называется demo, то есть так же, как и каталог, в котором данный файл расположен. После имени модуля с помощью фигурных скобок можно определить тело модуля, но в данном случае код модуля не содержит никаких инструкций.

Далее в каталоге demo создадим папку **com**. В папке com создадим папку **metanit**, а в папке com/metanit - папку **hello**.

В папке *com/metanit/hello* определим новый файл **Hello.java**:

```
package com.metanit.hello;


public class Hello{

	public static void main(String[] args){
		System.out.println("Hello Demo Module!");
	}
}
```

Название пакета файла - *com.metanit.hello* отражает структуру папок, в которых расположен файл. Сам файл определяет класс Hello, который в методе main выводит на консоль строку.

В итоге у нас получится следующая стуктура проекта:

Теперь скомпилируем все это. Для этого вначале перейдем в командной строке/терминале к папке, в которой находится модуль demo.

Затем для компиляции модуля выполним следующую команду:

```
javac demo/module-info.java demo/com/metanit/hello/Hello.java
```

После компиляции модуля demo выполним программу с помощью следующей команды:

```
java  --module-path demo --module demo/com.metanit.hello.Hello
```

Параметр *--module-path* указывает на путь к модулю, а *--module* - на главный класс модуля.

При наборе команды вместо параметра *--module-path* можно указать его сокращение *-p*, а вместо параметра *--module* - сокращение *-m*.

И на консоли отобразится сообщение "Hello Demo Module!"

3.4.1. Структурное программирование

3.4.2. Структурное программирование

Структурное программирование — это методологический подход к написанию программного кода, который представляет программу в виде структуры из набора блоков, расположенных в иерархической последовательности.

Структурное программирование возникло еще на ранних этапах развития методологий создания кода. По мере того, как развивались ЭВМ и появлялись более сложные программы, возникла необходимость упрощать сам процесс написания программ.

Тогда были сформулированы три принципа структурного программирования:

1. *Алгоритмическая нисходящая декомпозиция.* Поставленная задача пошагово детализируется, в направлении от самого верхнего уровня, вниз к мелким деталям. Этот метод позволяет создать четкую структуру программы.
2. *Модульная структура программы.* В результате декомпозиции программа разбивается на модули, простые элементы.
3. *Структурное кодирование.* При структурном кодировании используются три управляющие конструкции: последовательное управление, ветвление, цикл. Это базовые структуры.

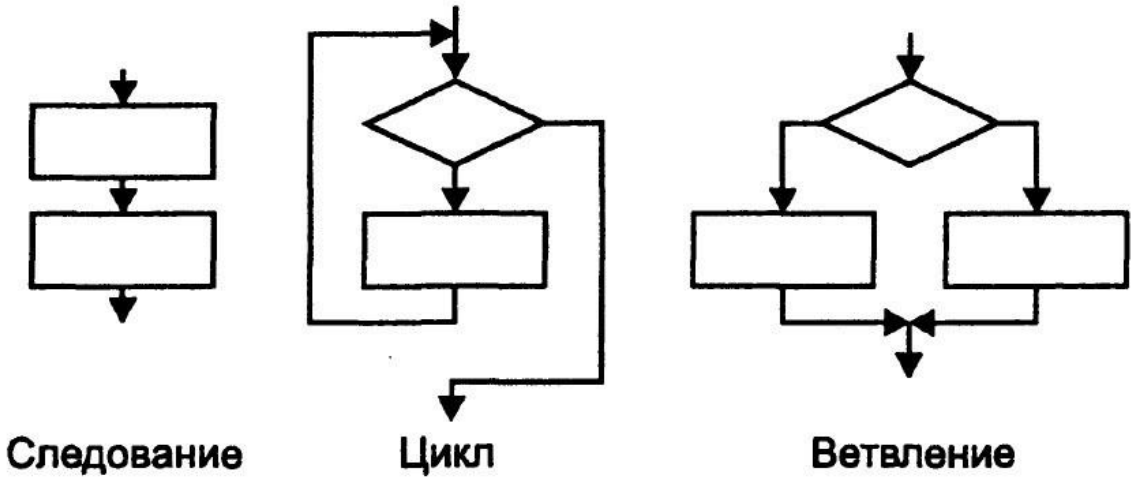
Базовые конструкции структурного программирования

Структурное программирование основывается на теореме, которую упрощенно можно пересказать так:

- Две программы считаются эквивалентными, если при любых одинаковых входных данных они всегда выдают одинаковые выходные данные или одинаково закрываются по ошибке.
- Структурная программа использует только три конструкции:
    - последовательное исполнение – «выполнить действие 1, выполнить действие 2»;

- ветвление – «если условие соблюдено, выполнить действие 1, иначе действие 2»;
- цикл – «пока условие соблюдается, выполнять действие 1».

- Оператор GoTo используется для неструктурных программ.
- Для каждой неструктурной программы существует эквивалентная структурная программа.



Базовые конструкции структурного программирования

Основываясь на этой теореме, иногда структурное программирование называют «программированием без Go-to». Но это не совсем верно. В каждом правиле есть исключения, и в структурном программировании оператор go-to использовать допустимо. Но все же, прежде чем его использовать, стоит пересмотреть, нельзя ли написать код без этого оператора.

Преимущества структурного программирования

Структурное программирование часто рассматривается как идеальный подход к программированию, поскольку оно позволяет создавать структурированный и легко поддерживаемый код. Это особенно полезно в больших и сложных приложениях, где организация кода важна как для программистов, так и для конечных пользователей.

Структурное программирование имеет ряд преимуществ перед неструктурированными методами, в том числе:

- Код лучше читается;
- Код становится проще;
- Код становится надежнее, что означает меньше ошибок или сбоев;
- Улучшено обслуживание кода, то есть его легче изменить или обновить по мере необходимости.

Преимущества структурного программирования особенно ярко проявляются в больших проектах. Благодаря структурности:

- легко прослеживается логика программы, что важно для разработчиков, не участвовавших в проекте с самого начала;
- фрагменты кода, оформленные как процедуры, легко использовать повторно;
- поддерживать проект можно без участия первых авторов годы спустя.

Методы и концепции структурного программирования

Существует несколько методов и концепций, которые используются в структурном программировании для создания хорошо структурированного и эффективного кода.

1. Принцип единой ответственности. Этот принцип гласит, что у каждого модуля или класса должна быть только одна причина для изменения. Каждый модуль имеет четкую ответственность и не отвечает за несколько областей программы.
2. Объектно-ориентированное программирование. Объектно-ориентированное программирование — это тип программирования, в котором используются объекты, включающие в себя как данные, так и функции или методы, работающие с этими данными. Объектно-ориентированное программирование часто используется для создания сложных и модульных программ.
3. Абстракция функции или метода: это процесс создания высокоуровневого интерфейса для фрагмента кода, который может использоваться другими частями программы. Это позволяет другим частям программы взаимодействовать с кодом без необходимости знать конкретные детали.
4. Инкапсуляция: это процесс сокрытия деталей структуры данных или объекта от остальной части программы. Это дает возможность использовать объект или структуру без необходимости вникать в ее внутреннюю работу.
5. Наследование: это создание нового класса или модуля, который наследует свойства и методы другого класса или модуля. Таким образом можно создавать семейство связанных классов или модулей с общими характеристиками.
6. Полиморфизм. Это способность программы обращаться с объектами разных классов так, как если бы они принадлежали к одному и тому же классу. Это позволяет программе использовать один метод для работы с несколькими классами.

Есть много других методов и концепций, которые встречаются в структурном программировании, но эти чаще всего используются для создания надежного и эффективного программного обеспечения.

Хотите изучить язык программирования Java и научиться использовать структурное программирование на практике? Тогда менторинг — это именно то, что вам нужно! Мы предлагаем индивидуальный подход к каждому студенту и возможность общаться с опытными наставниками, которые помогут вам на каждом этапе обучения. В курс Java Spring входят практические задания, которые позволят вам углубленно изучить все аспекты языка Java и структурного программирования. Наши наставники имеют многолетний опыт работы с Java и хорошо знакомы со структурным программированием. Они помогут вам понять все особенности этого подхода и научат, как применять его на практике.

Языки структурного программирования

Структурное программирование использовать разные языки, но самые популярные это C, C++, Java и Python. Эти языки используют конструкции структурного программирования, такие как операторы if-else, циклы и функции, что позволяет программистам создавать с их помощью хорошо структурированный и эффективный код.

Кроме того, многие языки программирования имеют специальные методы и концепции, поддерживающие структурное программирование, например объектно-ориентированное программирование, инкапсуляцию, наследование и полиморфизм. Эти методы и концепции помогают программистам создавать надежное и модульное программное обеспечение, которое легко поддерживать и расширять.

В целом, использование принципов и методов структурного программирования может улучшить качество и удобство сопровождения программного обеспечения на любом языке программирования, что делает его важным инструментом для освоения разработчиками.

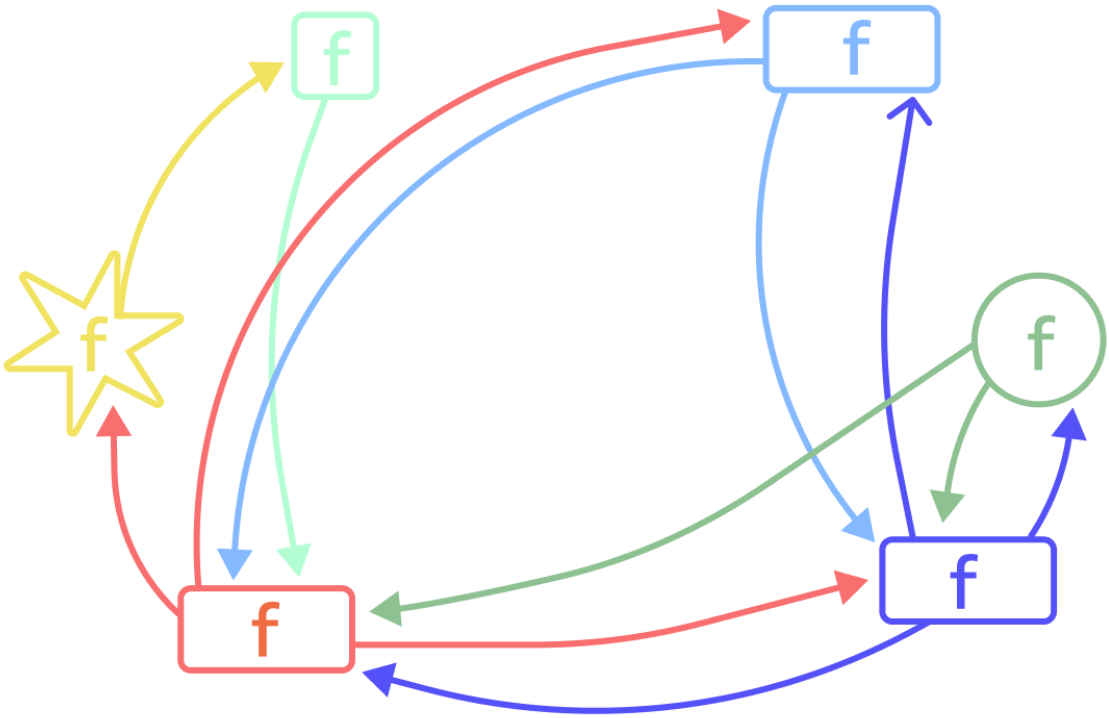
3.5. Объектно-ориентированное программирование

Объектно-ориентированное программирование (ООП) — это подход, при котором программа рассматривается как набор объектов, взаимодействующих друг с другом. У каждого есть свойства и поведение. Если постараться объяснить простыми словами, то ООП ускоряет написание кода и делает его более читаемым.

Идеология объектно-ориентированного программирования (ООП) разрабатывалась, чтобы связать поведение определенного объекта с его классом. Людям проще воспринимать окружающий мир как объекты, которые поддаются определенной классификации (например, разделение на живую и неживую природу).

Зачем нужно ООП

До ООП в разработке использовался другой подход — процедурный. Программа представляется в нем как набор процедур и функций — подпрограмм, которые выполняют определенный блок кода с нужными входящими данными. Процедурное программирование хорошо подходит для легких программ без сложной структуры. Но если блоки кода большие, а функций сотни, придется редактировать каждую из них, продумывать новую логику. В результате может образоваться много плохо читаемого, перемешанного кода — «спагетти-кода» или «лапши».



Пример «спагетти-кода»

В отличие от процедурного, объектно-ориентированное программирование позволяет вносить изменения один раз — в объект. Именно он — ключевой элемент программы. Все операции представляются как взаимодействие между объектами. При этом код более читаемый и понятный, программа проще масштабируется.

Объектно-ориентированное программирование используется, чтобы:

- структурировать информацию и не допускать путаницы;
- точно определять взаимодействие одних элементов с другими;
- повышать управляемость программы;
- быстрее масштабировать код под различные задачи;
- лучше понимать написанное;
- эффективнее поддерживать готовые программы;
- внедрять изменения без необходимости переписывать весь код.

Возможности ООП поддерживает большинство популярных языков программирования, включая JavaScript, PHP, Python и другие.

Читайте такжеСамые востребованные IT-профессии 2024 года

Структура ООП

Объекты и классы

Чтобы сделать код проще, программу разбивают на независимые блоки — объекты. В реальной жизни это может быть стол, чашка, человек, книга, здание и многое другое. В программировании объекты — это структуры данных: пользователь, кнопка, сообщение. У них, как и у реальных предметов, могут быть свойства: цвет, содержание или имя пользователя. А чтобы объединить между собой объекты с похожими свойствами, существуют классы.

Класс — это «шаблон» для объекта, который описывает его свойства. Несколько похожих между собой объектов, например профили разных пользователей, будут иметь одинаковую структуру, а значит, принадлежать к одному классу. Каждый объект — это экземпляр какого-нибудь класса.

Понять, что такое ООП, поможет аналогия.

- Понятие «программист» — это класс.
- Конкретный разработчик по имени Иван — это объект, принадлежащий к классу «программист» (экземпляр класса).
- Зарплата, рабочие обязанности, изученные технологии и должность в компании — это свойства, которые есть у всех объектов класса «программист», в том числе у Ивана. У разных объектов свойства различаются: зарплата и обязанности Ивана будут отличаться от таковых у другого разработчика Миши.

Атрибуты и методы

Объект — это набор переменных и функций, как в традиционном функциональном программировании. Переменные и функции и есть его свойства.

Атрибуты

— это переменные, конкретные характеристики объекта, такие как цвет поля или имя пользователя.

Методы

— это функции, которые описаны внутри объекта или класса. Они относятся к определенному объекту и позволяют взаимодействовать с ними или другими частями кода.

Объект, класс, атрибуты и методы в ООП на примере

Принципы ООП

Объектно-ориентированное программирование определяют через четыре принципа, по которым можно понять основы работы. Иногда количество сокращают до трех — опускают понятие абстракции.

Абстракция

Абстрагирование — это способ выделить набор наиболее важных атрибутов и методов и исключить незначимые. Соответственно, абстракция — это использование всех таких характеристик для описания объекта. Важно представить объект минимальным набором полей и методов без ущерба для решаемой задачи.

Пример: объекту класса «программист» вряд ли понадобятся свойства «умение готовить еду» или «любимый цвет». Они не влияют на его особенности как программиста. А вот «основной язык программирования» и «рабочие навыки» — важные свойства, без которых программиста не опишешь.

Набор атрибутов и методов, доступный извне, работает как интерфейс для доступа к объекту. Через них к нему могут обращаться другие структуры данных, причем им не обязательно знать, как именно объект устроен внутри.

Инкапсуляция

Каждый объект — независимая структура. Все, что ему нужно для работы, уже есть у него внутри. Если он пользуется какой-то переменной, она будет описана в теле объекта, а не снаружи в коде. Это делает объекты более гибкими. Даже если внешний код перепишут, логика работы не изменится.

Инкапсуляция помогает с легкостью управлять кодом. Выше мы сказали, что для обращения к объекту не нужно понимать, как работают его методы. Начальнику разработчика Ивана не обязательно знать, как именно он программирует: главное — чтобы выполнялись поставленные задачи.

Внутреннее устройство одного объекта закрыто от других: извне «видны» только значения атрибутов и результаты выполнения методов.

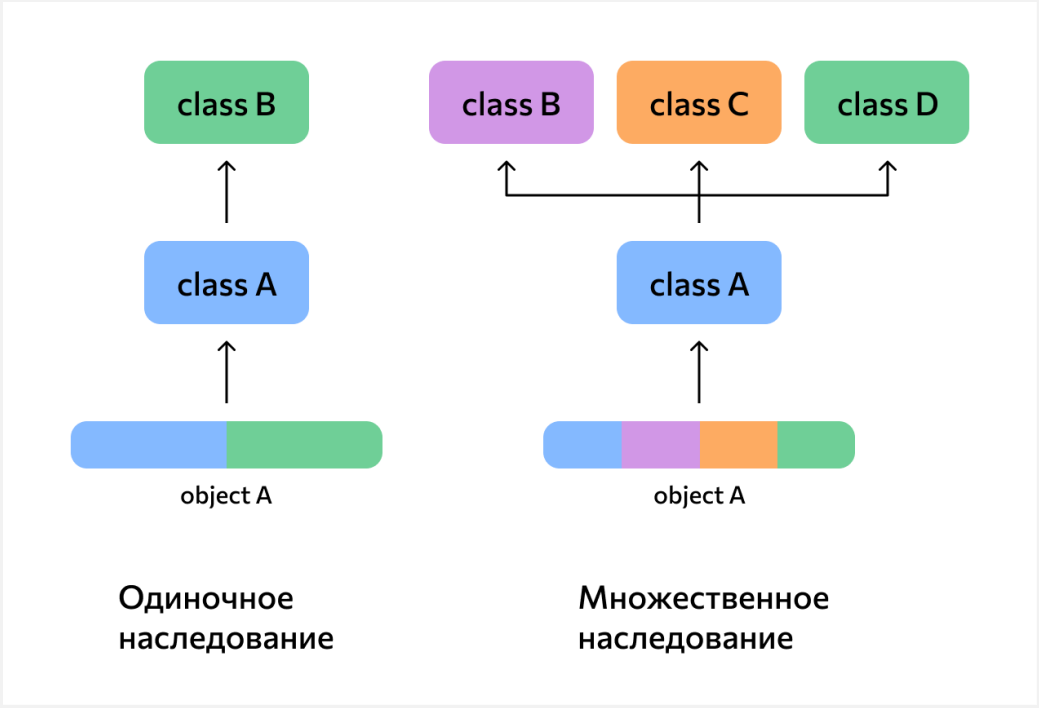
Наследование

Можно создавать классы и объекты, которые похожи друг на друга, но немного отличаются — имеют дополнительные атрибуты и методы. Более общее понятие в таком случае становится «родителем», а более специфичное и подробное — «наследником».

Упомянутый программист Иван — это человек. Но «человек» — более общее определение, которое не описывает свойства, важные именно для программиста. Можно сказать, что класс «программист» унаследован от класса «человек»: программист тоже является человеком, но у него есть дополнительные свойства.

В таком случае разработчик Иван будет и человеком, и программистом одновременно. У него будут наборы свойств от обоих классов.

У одного «родителя» может быть несколько дочерних структур. Например, от «человека» можно наследовать не только «программиста», но и «директора».



Одиночное и множественное наследие

Наследование позволяет реализовывать сложные схемы с четкой иерархией «от общего к частному». Это облегчает понимание и масштабирование кода. Не нужно много раз переписывать в разных объектах одни и те же свойства. Достаточно унаследовать эти объекты от одного «родителя», и «родительские» свойства применятся автоматически.

Полиморфизм

Одинаковые методы разных объектов могут выполнять задачи разными способами. Например, у «человека» есть метод «работать». У «программиста» реализация этого метода будет означать написание кода, а у «директора» — рассмотрение управленческих вопросов. Но глобально и то, и другое будет работой.

Тут важны единый подход и договоренности между специалистами. Если метод называется delete, то он должен что-то удалять. Как именно — зависит от объекта, но заниматься такой метод должен именно удалением. Более того: если оговорено, что «удаляющий» метод называется delete, то не нужно для какого-то объекта называть его remove или иначе. Это вносит путаницу в код.

Преимущества ООП

Модульность

Объектно-ориентированный подход позволяет сделать код более структурированным, в нем легко разобраться стороннему человеку. Благодаря инкапсуляции объектов уменьшается количество ошибок и ускоряется разработка с участием большого количества программистов, потому что каждый может работать независимо друг от друга.

Гибкость

ООП-код легко развивать, дополнять и изменять. Это обеспечивает независимая модульная структура. Взаимодействие с объектами, а не логикой упрощает понимание кода. Для модификации не нужно погружаться в то, как построено ПО. Благодаря

полиморфизму можно быстро адаптировать код под требования задачи, не описывая новые объекты и функции.

Экономия времени

Благодаря абстракции, полиморфизму и наследованию можно не писать один и тот же код много раз. Это ускоряет разработку нового ПО. Интерфейсы и классы в ООП могут легко преобразовываться в подобие библиотек, которые можно использовать заново в новых проектах. Также ООП экономит время при поддержке и доработке приложения.

Безопасность

Программу сложно сломать, так как инкапсулированный код недоступен извне.

Недостатки ООП

Сложный старт

Чтобы пользоваться ООП, нужно сначала изучить теорию и освоить процедурный подход, поэтому порог входа высокий.

Снижение производительности

Объектно-ориентированный подход немного снижает производительность кода в целом. Программы работают несколько медленнее из-за особенностей доступа к данным и большого количества сущностей.

Большой размер программы

Код, написанный с использованием ООП, обычно длиннее и занимает больше места на диске, чем «процедурный». Это происходит, потому что в такой программе хранится больше конструкций, чем в обычном процедурном скрипте.

3.6.1. Эффектность и оптимизация программ

3.6.2. Эффектность и оптимизация программ

Главным критерием эффективности программ является распределение ресурсов вычислительных систем. Неравномерность задач по допустимому времени задержки или допустимой вероятности пропуска решений, а также различия параметров вычислительных систем, позволяют изменить качество решения задач выделением соответствующих ресурсов вычислительных систем. Упорядочивание последовательности решения задач и рациональное использование ресурсов вычислительных систем сокращает запаздывание в решении задач, и в некоторой степени приводит к эквивалентно повышению производительности вычислительных систем. Производительность выч.системы является одним из важнейших критериев эффективности выч.систем в целом и методов распределения ресурсов в частности. Существуют такие понятия, как относительный и абсолютный приоритеты. При распределении буферной памяти для приема и выдачи сообщений применяются буферные накопители, объем которых ограничивает эффективность использования. Ограничение буферных накопителей зависит от их структурного построения и распределения имеющейся памяти на зоны. На эффективность существенно влияют степень заполнения памяти и передача инф-ции на накопители на обработку.

Тема 2.8. Оптимизация программ. Оптимизирующие компиляторы.

Понятие оптимизации программ

Оптимизация программы - это улучшение какой-либо характеристики программы, называемой критерием оптимизации. Оптимизация программ в основном выполняется по двум основным критериям: быстродействие и объему используемых данных.

Производительность приложения определяется самым узким его участком, поэтому в первую очередь нужно определить части программы, на которых будет выполняться оптимизация. Процесс оптимизации следует начать с профилировки программы. Профилировкой называют измерение производительности как всей программы, так и отдельных ее фрагментов, с целью нахождения «горячих точек» - тех участков программы, на выполнение которых расходуется наибольшее количество времени. При этом важно отметить, что ликвидация не самых «горячих» точек программы, практически не увеличивает ее быстродействия.

Основная цель профилировки – это исследование характера поведения приложения во всех его точках. В зависимости от степени детализации в качестве «точки» рассматривается как отдельная машинная команда, так и целая конструкция высокого языка - функция, цикл, процедура. Сложная программа состоит из большого числа функций. Нет смысла оптимизировать их все – трудоемкость такого подхода будет выше выгод, полученных от оптимизации программы целиком. Для начала необходимо локализовать участки кода с максимальной вычислительной трудоемкостью. Участки программы, которые в наибольшей степени влияют на ее производительность, в силу наиболее частого выполнения или своей ресурсоемкости называются критическим кодом. В поиске критического кода программы используют профайлеры (профилировщики) – специальные программы, которые измеряют временные затраты на выполнение участков кода программы. Профилировщики представляют возможности для оптимизации программ. К таким программам относятся Intel VTune, AMD Code Analyst, profile.exe и множество других. Наиболее мощным из них на сегодняшний день является пакет от Intel.

Эта программа позволяет измерить время обработки каждой команды и вывести полную статистику о состоянии процессора при выполнении каждой команды.

Большинство современных профилировщиков поддерживают следующий набор базовых операций:

• определение общего времени исполнения каждой точки программы;

• определение удельного времени исполнения каждой точки программы;

• определение причины и/или источника конфликтов;

• определение количества вызовов той или иной точки программы;

• определение степени покрытия программы.

Основные правила оптимизации:

1. Прежде чем приступать к оптимизации, необходимо иметь надежно работающий неоптимизированный вариант.

2. Основной прирост оптимизации дает не учет особенностей системы, а алгоритмическая оптимизация.

3. Обнаружив профилировщиком узкие места необходимо произвести оптимизацию в рамках языка высокого уровня.

Возможны ситуации, где в неудовлетворительной производительности кода виноваты процессор или подсистема памяти, а не компилятор. Лишь после анализа листинга следует приступать к ассемблерной оптимизации.

Оптимизация начинается с выделения профилировщиком критического кода и анализа его неоптимальности. Причем каждое внесенное изменение необходимо проверять профилировщиком. После завершения оптимизации локального фрагмента программы, необходимо выполнить контрольную профилировку всей программы целиком на предмет обнаружения новых появившихся «горячих точек».

Проводя оптимизацию, не следует забывать о ее цели. Фактически идеал недостижим, поэтому оптимизацию следует завершать когда:

1. Производительность программы признана удовлетворяющей;

2. В программе отсутствуют «горячие точки», то есть количество инструкций равномерно распределено по все программе, и дальнейшая оптимизация потребует переписывания большого количества кода;

3. Сложность алгоритма настолько высока, что не представляется возможным дальнейшая оптимизация без значительных временных затрат;

4. Критическая зависимость от платформы, когда дальнейшая машинно-зависимая оптимизация приведет к потере совместимости с одной из целевых платформ.

Ко всем методам оптимизации алгоритма предъявляются следующие требования:

1. оптимизация должна быть по возможности максимально машинно- независимой и переносимой на другие платформы (операционные системы) без существенных потерь эффективности.

2. оптимизация не должна увеличивать трудоемкость разработки (в том числе тестирования) приложения более чем на 10-15%.

3. оптимизирующий алгоритм должен давать выигрыш не менее чем на 20-25% в скорости выполнения.

4. оптимизация не должна допускать безболезненное внесение изменений.

Алгоритмические приемы оптимизации

Приемы оптимизации программы можно разделить на алгоритмические и машинно-зависимые способы. В случае использования алгоритмических приемов оптимизации используются различные математические и логические методы для улучшения параметров алгоритма. Такой способ оптимизации невозможно автоматизировать, успешность его применения зависит от программиста. Способность программиста к алгоритмической оптимизации программы зависит от его понимания предметной области: владения им базовых концепций применяемых алгоритмов и особенностей предметной области программы.

В первую очередь это замена алгоритмов на более быстродействующие. Часто бывает, что более простой алгоритм показывает низкую производительность по сравнению с более сложными. Тогда, возможна замена эквивалентных алгоритмов, например, замена пузырьковой сортировки массива на быструю сортировку.

В некоторых случаях возможна оптимизация программы за счет снижение точности. В зависимости от особенностей предметной области возможно уменьшить разрядность представления чисел или перейти от выполнения операций с числами с плавающей запятой к целым числам или числам с фиксированной запятой.

На практике используется весьма широкий набор машинно-независимых оптимизирующих преобразований, что связано с большим разнообразием неоптимальностей. К ним относятся:

• разгрузка участков повторяемости - это такой способ оптимизации, который состоит в вынесении вычислений из многократно исполняемых участков программы на участки программы, редко исполняемые. К этому виду преобразования относятся различные чистки зон, тел циклов и тел рекурсивных процедур, когда инвариантные по результату выполнения выражения, исполняемые при каждом прохождении участка повторяемости, выносятся из него. Если размещение осуществляется перед входом в участок повторяемости, то эту ситуацию называют чисткой вверх, если же за выходом из участка повторяемости, то чисткой вниз

• упрощение действий - этот способ оптимизации ориентирован на улучшение программы за счет замены групп (как правило, удаленных друг от друга) вычислений на группу вычислений, дающий тот же результат с точки зрения всей программы, но имеющих меньшую сложность.

• чистка программы - данный способ повышает качество программы за счет удаления из нее ненужных объектов и конструкций. Набор преобразований этого типа включает в себя следующие оптимизации: удаление идентичных операторов, удаление из программы операторов, недостижимых по управлению от начального, удаление несущественных операторов, то есть операторов не влияющих на результат программы, удаление процедур, к которым нет обращений, удаление неиспользуемых переменных и другие.

• экономия памяти и оптимизация работы с памятью - улучшения быстродействия возможно за счет уменьшения объема памяти, отводимой под информационные объекты программы в каждом ее исполнении.

• реализация действий - это способ повышения быстродействия программы за счет выполнения определенных ее вычислений на этапе трансляции.

• сокращение программы и другие методы.

Машинно-зависимые приемы оптимизации

Машинно-зависимые используют особенности устройства и работы конкретной системы. Ярким примером машинно-зависимой оптимизации является векторизация операций, т.е. использование потоковых расширений процессора, таких как MMX (MultiMedia eXtensions), SSE (Streaming SIMD Extensions) и т.п. Машино-зависимую оптимизацию можно выполнять двумя различными способами. Первый способ основан на понимании работы кодогенератора компилятора, его алгоритма и рекомендуется для приложений, в которых компилятор выбирается в начале проекта и в дальнейшем не меняется. При использовании такого способа преобразуется исходный код программы, написанный на языке высокого уровня. Для тех проектов, в которых заранее не известен компилятор (OpenSource проекты, кроссплатформенные приложения) применятся другой способ, основанный на замещении ресурсоемких участков кода ассемблерными вставками. При такой оптимизации ухудшается переносимость кода на другие платформы. Машинно-зависимые способы оптимизации довольно хорошо автоматизируются и большую часть их выполняют оптимизирующие компиляторы. Однако всегда остаются моменты в программе, которые можно оптимизировать вручную.

3.7.1. Обеспечение качества программного продукта

3.7.2. Обеспечение качества программного продукта

Обеспечение качества программного обеспечения (Software Quality Assurance, SQA) — это комплекс мероприятий, который гарантирует, что все процессы и методы разработки ПО контролируются и соответствуют установленным стандартам. К этим стандартам относятся, например, ISO 9000, модель CMMI и ISO 15504.

SQA включает в себя все процессы разработки программного обеспечения, от формирования техзадания до разработки программы (включая написание кода) и вплоть до выпуска готового продукта. Главная цель — обеспечить качество.

- План контроля качества ПО
- Этапы обеспечения качества
- Стандарты обеспечения качества программного обеспечения
- Элементы обеспечения качества программного обеспечения
- Методы SQA

План контроля качества ПО

План контроля качества ПО (по-английски сокращается как SQAP, от Software Quality Assurance Plan) включает в себя методы и средства, позволяющие убедиться, что продукт или услуга отвечают установленным для него требованиям спецификации (в англоязычных источниках SRS, Software Requirement Specification).

В этом плане определяются обязанности команды по обеспечению качества ПО и перечисляются пункты, которые должны быть проверены. Также в нем указывается, к какому результату должен привести весь процесс обеспечения качества.

Документ с планом контроля качества состоит из следующих разделов:

1. Цели работы
2. Аннотация
3. Управление конфигурацией
4. Отчеты об ошибках и внесение изменений
5. Технологии, методы и средства
6. Контроль качества кода
7. Протокол тестирования: сбор, ведение и хранение записей
8. Методология тестирования

Этапы обеспечения качества

1. Создание плана контроля качества

В первую очередь, необходимо составить четкий план того, как именно в вашем проекте будет осуществляться управление качеством.

В зависимости от того, какого подхода вы собираетесь придерживаться и какие технологии планируется задействовать будут проводиться, план также может включать контроль за подбором экспертов в вашей команде.

2. Установка контрольных точек

Команда устанавливает ряд контрольных точек, в соответствии с которыми она оценивает качество проектной деятельности на каждом этапе проекта. Это обеспечивает регулярную проверку качества и работу в соответствии с графиком.

3. Участие в сборе требований для группы разработчиков

Для достижения высокого качества продукта необходимо совмещать тестирование с процессом его разработки. Для сбора необходимой информации разработчик может использовать такие методы, как интервью или метод быстрого анализа решений (Functional Analysis System Technique, FAST).

Позже, основываясь на собранной информации, разработчик может вычислить оценку временных затрат на реализацию проекта, используя такие метрики, как иерархическая структура работ (Work Breakdown Structure,WBS), количество строк коды (Source Lines of Code, SLOC) и метод функциональных точек (Function Points, FP).

4. Проведение технического анализа

Технический анализ (Formal Technical Review, FTR) проводится для оценки качества и дизайна прототипа ПО.

В этом процессе проводится встреча с разработчиками и другими техническими экспертами для обсуждения фактических требований к качеству программного обеспечения и качества его прототипа. Эта работа помогает обнаружить ошибки на ранней стадии жизненного цикла ПО и сократить усилия на его переработку на последующих этапах.

5. Работа над стратегией мультитестирования

Под стратегией мультитестирования подразумевается, что не следует полагаться только на какой-либо единственный подход к тестированию. Вместо этого необходимо проводить несколько разных видов тестирования для достижения лучшего качества.

6. Контроль за соблюдением технологических процессов

Этот вид деятельности обеспечивает соблюдение технологических процессов в ходе разработки ПО. Процесс разработки должен происходить в соответствии с установленными требованиями.

Данный этап можно условно разделить на две составляющие:

(i) Оценка продукта

Подтверждает, что программный продукт соответствует заданным требованиям, и что стандарты соблюдены.

(ii) Мониторинг процессов

Периодический контроль за разного рода метриками позволяет отследить, налажена ли работа по разработке ПО должным образом.

7. Управление изменениями

Позволяет убедиться, что все изменения контролируемы и вносятся с нашего ведома. Контроль изменений осуществляется вручную или с помощью инструментов автоматизации.

Подтверждая запросы на изменения, оценивая их характер и контролируя последствия, мы обеспечиваем поддержание качества программного обеспечения на этапах разработки и поддержки продукта.

8. Отслеживание зависимостей

Команда активно участвует в оценки влияния изменений, внесённых в результате устранения ошибок или существенных нововведений. Этот этап необходим, чтобы избежать нежелательных эффектов.

Для этого используются метрики качества программного обеспечения, которые позволяют руководителям проекта и самим разработчикам наблюдать за ходом разработки и предлагаемыми изменениями в течение всего жизненного цикла ПО и вносить коррективы там, где это необходимо.

9. Аудит системы обеспечения качества

Аудит позволяет проверить все фактические процессы жизненного цикла ПО и сравнить их с установленными требованиями.

Он также позволяет выяснить, было ли на самом деле выполнено то, о чем команда сообщала в своих отчетах. Эта деятельность также выявляет любые неожиданные проблемы.

10. Ведение записей и протоколов тестирования

Очень важно хранить необходимую документацию, связанную с обеспечением качества, и делиться требуемой информацией о процессах с заинтересованными сторонами. Результаты тестирования, аудита, отчеты о проверках, документация по запросам на изменения и тому подобное, должны храниться как для анализа, так и на будущее.

11. Работа с коллективом

Сильная команда разработчиков — та, в которой есть гармония между разными отделами. В самом деле, очень важно поддерживать хорошие отношения между отделом контроля качества и отделом разработки.

Известно, что тестировщики (испытатели) и разработчики ПО часто чувствуют превосходство друг над другом. Подобного рода взаимоотношений в командах следует избегать, так как это также может повлиять на общее качество продукта.

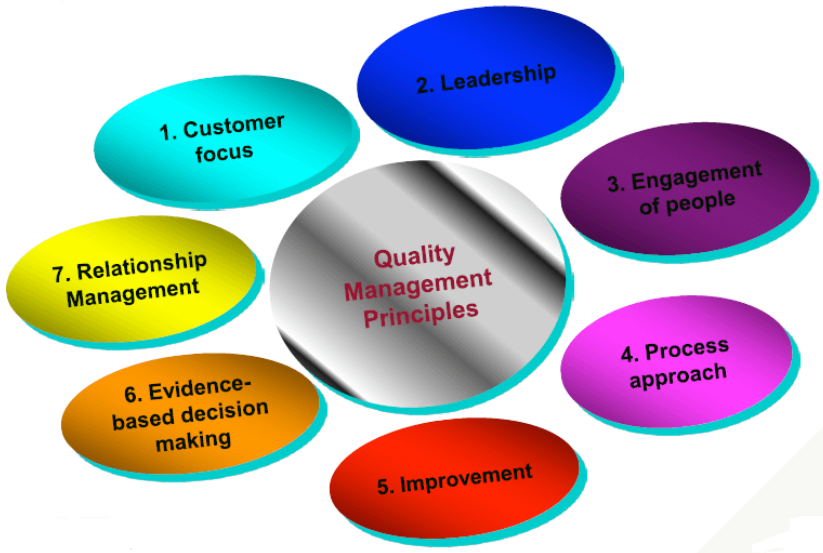
Стандарты обеспечения качества программного обеспечения

Разработка ПО в целом и обеспечение его качества в частности может потребовать соблюдения одного или нескольких стандартов.

Ниже рассматриваются некоторые из наиболее популярных стандартов

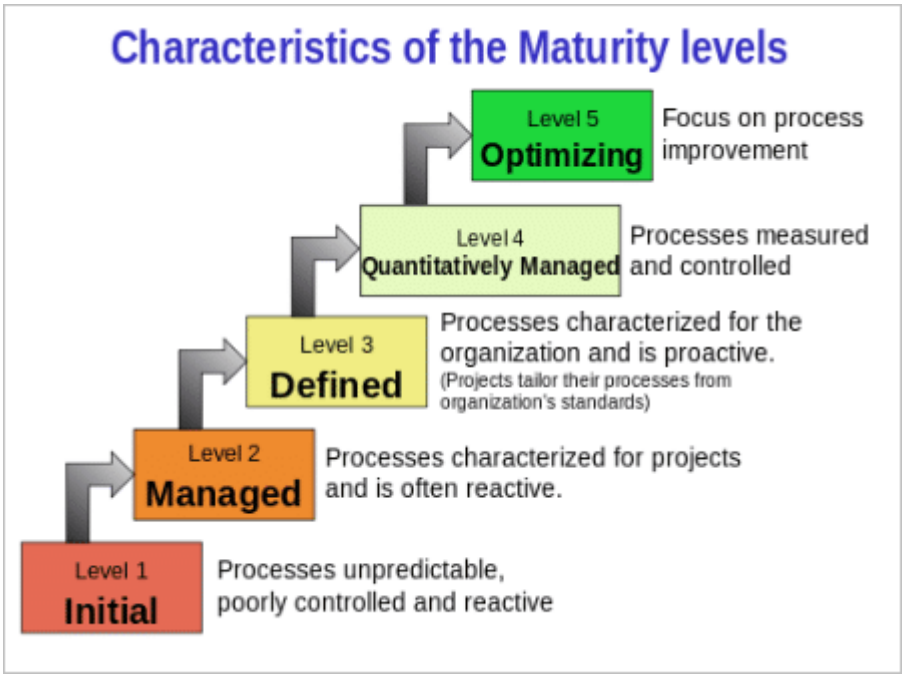
ISO 9000: Этот стандарт основан на семи принципах управления качеством, которые помогают организациям гарантировать, что их продукция или услуги соответствуют потребностям клиентов.

Семь принципов ISO 9000 представлены на рисунке ниже:



Уровень CMMI: CMMI расшифровывается как набор моделей совершенствования процессов (**Capability maturity model Integration)**. Эта модель возникла в программной инженерии. Она может быть использована для совершенствования процессов в рамках проекта, отдела или всей организации.

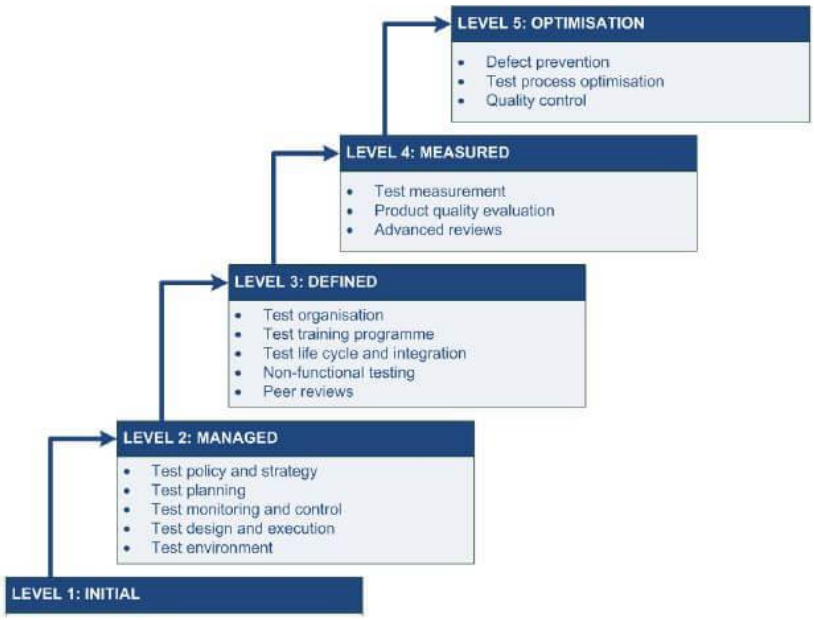
Пять уровней CMMI и их характеристики описаны на рисунке ниже



Организация оценивается и получает рейтинг уровня зрелости (1—5) в зависимости от типа оценки.

Test Maturity Model integration, **TMMi**: Обобщённая модель зрелости процессов тестирования. Основанная на **CMM**i, эта модель фокусируется на уровнях зрелости в управлении качеством программного обеспечения и тестировании.

Пять уровней TMMi показаны на рисунке ниже



По мере перехода на более высокий уровень зрелости, организация достигает более высоких возможностей для производства высококачественной продукции с меньшим количеством дефектов и близким соответствием требованиям бизнеса.

Элементы обеспечения качества программного обеспечения

Существует десять основных элементов:

1. Стандарты разработки ПО
2. Технический анализ и аудит
3. Тестирование программного обеспечения для контроля качества
4. Отслеживание и анализ ошибок
5. Периодическое снятие метрик
6. Отслеживание изменений
7. Контроль за исполнителями
8. Управление безопасностью
9. Управление рисками
10. Обучение

Техники обеспечения качества

Существует несколько техник, из которых наиболее распространен аудит. Но не стоит забывать и про другие.

Различные техники SQA включают в себя:

- **Аудит:** Аудит включает в себя проверку продуктов труда и связанной с ними информации, чтобы определить, соблюдался ли набор стандартных процессов или нет.

- **Рецензирование:** Встреча, на которой программный продукт рассматривается внутренними и внешними заинтересованными сторонами с целью получения их комментариев.
- **Ревизия кода:** Это наиболее формальный вид проверки, при котором проводится статическое тестирование для поиска ошибок и предотвращения роста дефектов на более поздних стадиях. Она проводится обученным экспертом-посредником или незаинтересованным лицом и основывается на правилах и контрольных списках, критериях ввода-выводаа. Рецензент не должен быть автором кода.
- **Комплексная инспекция проекта:** Эта проверка осуществляется с помощью контрольного списка, который проверяет следующие области разработки программного обеспечения:
    - Общие требования и основные возможности
    - Требования к функциям и к интерфейсу
    - Соответствие соглашениям
    - Прослеживаемость выполнения требований
    - Компоненты и интерфейсы
    - Логика
    - Производительность
    - Обработка ошибок и способность к восстановлению
    - Возможность проверки и расширения
    - Связность элементов
- **Симуляция:** Это инструмент, который моделирует реальное состояние исследуемой системы, чтобы изучить ее поведение.
- **Функциональное тестирование:** Это техника QA позволяет проверить, что именно делает система, не вдаваясь в подробности, как именно она это делает. Программа рассматривается как "черный ящик". А главное внимание привлечено к проверке системы на функциональность или соответствие спецификациям.
- **Стандартизация:** Стандартизация играет решающую роль в обеспечении качества. Чем меньше двусмысленностей и чем меньше приходится гадать, тем выше качество.
- **Статический анализ:** Это анализ программного обеспечения, который выполняется автоматизированным инструментом без фактического выполнения программы. Эта техника широко используется для обеспечения качества в медицинском, ядерном и авиационном программном обеспечении. Среди популярных форм такого анализа — сбор метрик и обратная разработка.
- **Разбор продукта:** Это вид экспертной оценки, когда члены команды разработки просматривают продукт по созданной автором кода инструкциям и задают вопросы, предлагают альтернативы и оставляют комментарии относительно возможных дефектов, нарушений стандартов или любых других вопросов.
- **Модульное тестирование:** Это метод тестирования "белого ящика", при котором обеспечивается полное исследование кода — каждая независимая ветвь кода и каждое условие выполняется как минимум по одному разу.
- **Стресс-тестирование:** Этот вид тестирования проводится для проверки надежности системы путем тестирования ее под высокой нагрузкой, т.е. за пределами нормальных условий.
- **Шесть сигм:** Шесть сигм – это подход к обеспечению качества, направленный на достижение практически идеальных показателей продуктов или услуг. Он широко применяется во многих областях, включая программное обеспечение. Основная цель "Шести сигм" – совершенствование процессов таким образом, чтобы производимое программное обеспечение было на 99,76 % бездефектным.

Заключение

SQA — это комплексная деятельность, которая осуществляется на протяжении всего жизненного цикла программного обеспечения.

Обеспечение качества программного обеспечения очень важно для того, чтобы программный продукт или услуга преуспели на рынке и оправдали ожидания клиентов.

Существуют различные виды деятельности, применяемые стандарты и методы, которым необходимо следовать, чтобы убедиться, что поставляемое программное обеспечение имеет высокое качества и соответствует потребностям бизнеса.

4.1. Ошибки программного обеспечения

Ошибки в программировании – дело обычное, хоть и неприятное. В данной статье будет рассказано о том, какими бывают ошибки (баги), а также что собой представляют исключения.

Определение

Ошибка в программировании (или так называемый баг) – это ситуация у разработчиков, при которой определенный код вследствие обработки выдает неверный результат. Причин данному явлению множество: неисправность компилятора, сбои интерфейса, неточности и нарушения в программном коде.

Баги обнаруживаются чаще всего в момент отладки или бета-тестирования. Реже – после итогового релиза готовой программы. Вот несколько вариантов багов:

1. Появляется сообщение об ошибке, но приложение продолжает функционировать.
2. ПО вылетает или зависает. Никаких предупреждений или предпосылок этому не было. Процедура осуществляется неожиданно для пользователя. Возможен вариант, при котором контент перезапускается самостоятельно и непредсказуемо.
3. Одно из событий, описанных ранее, сопровождается отправкой отчетов разработчикам.

Ошибки в программах могут привести соответствующее приложение в негодность, а также к непредсказуемым алгоритмам функционирования. Желательно обнаруживать баги на этапе ранней разработки или тестирования. Лишь в этом случае программист сможет оперативно и относительно недорого внести необходимые изменения в код для отладки ПО.

История происхождения термина

Баг – слово, которое используется разработчиками в качестве сленга. Оно произошло от слова «bug» – «жук». Точно неизвестно, откуда в программировании и IT возник соответствующий термин. Существуют две теории:

1. 9 сентября 1945 года ученые из Гарварда тестировали очередную вычислительную машину. Она называлась Mark II Aiken Relay Calculator. Устройство начало работать с ошибками. Когда его разобрали, то ученые заметили мотылька, застрявшего между реле. Тогда некая Грейс Хоппер назвала произошедший сбой упомянутым термином.

2. Слово «баг» появилось задолго до появления Mark II. Термин использовался Томасом Эдисоном и указывал на мелкие недочеты и трудности. Во время Второй Мировой войны «bugs» называли проблемы с радарной электроникой.

Второй вариант кажется более реалистичным. Это факт, который подтвержден документально. Со временем научились различать различные типы багов в IT. Далее они будут рассмотрены более подробно.

Как классифицируют

Ошибки работы программ разделяются по разным факторам. Классификация у рядовых пользователей и разработчиков различается. То, что для первых – «просто программа вылетела» или «глючит», для вторых – огромная головная боль. Но существует и общепринятая классификация ошибок. Пример – по критичности:

1. Серьезные неполадки. Это нарушения работоспособности приложения, которые могут приводить к непредвиденным крупным изменениям.

2. Незначительные ошибки в программах. Чаще всего не оказывают серьезного воздействия на функциональность ПО.

3. Showstopper. Критические проблемы в приложении или аппаратном обеспечении. Приводят к выходу программы из строя почти всегда. Для примера можно взять любое клиент-серверное приложение, в котором не получается авторизоваться через логин и пароль.

Последний вариант требует особого внимания со стороны программистов. Их стараются обнаружить и устранить в первую очередь. Критические ошибки могут отложить релиз исходной программы на неопределенный срок.

Также существуют различные виды сбоев в плане частоты проявления: постоянные и «разовые». Вторые встречаются редко, чаще – при определенных настройках и действиях со стороны пользователя. Первые появляются независимо от используемой платформы и выполненных клиентом манипуляций.

Иногда может получиться так, что ошибка возникает только на устройстве конкретного пользователя. В данном случае устранение неполадки требует индивидуального подхода. Иногда – полной замены компьютера. Связано это с тем, что никто не будет редактировать исходный код, когда он «глючит» только у одного пользователя.

Виды

Существуют различные типы ошибок в программах в зависимости от типовых условий использования приложений. Пример – сбои, которые возникают при возрастании нагрузки на оперативную память или центральный процессор устройства. Есть баги граничных

условий, сбоя идентификаторов, несовместимости с архитектурой процессора (наиболее распространенная проблема на мобильных устройствах).

Разработчики выделяют следующие типы ошибок по уровню сложности:

1. «Борбаг» – «стабильная» неполадка. Она легко обнаруживается на этапе разработки и компилирования. Иногда – во время тестирования наработкой исходной программы.
2. «Гейзенбаг» – баги с поддержкой изменения свойств, включая зависимость от среды, в которой было запущено приложение. Сюда относят периодические неполадки в программах. Они могут исчезать на некоторое время, но через какой-то промежуток вновь дают о себе знать.
3. «Мандельбаг» – непредвиденные ошибки. Обладают энтропийным поведением. Предсказать, к чему они приведут, практически невозможно.
4. «Шредингбаг» – критические неполадки. Приводят к тому, что злоумышленники могут взломать программу. Данный тип ошибок обнаружить достаточно трудно, потому что они никак себя не проявляют.

Также есть классификация «по критичности». Тут всего два варианта – warning («варнинги») и критические весомые сбои. Первые сопровождаются характерными сообщениями и отчетами для разработчиков. Они не представляют серьезной опасности для работоспособности приложения. При компилировании такие сбои легко исправляются. В отдельных случаях компилятор справляется с этой задачей самостоятельно. А вот критические весомые сбои говорят сами за себя. Они приводят к серьезным нарушениям ПО. Исправляются обычно путем проработки логики и значительных изменений программного кода.

Типы багов

Ошибки в программах бывают:

- логическими;
- синтаксическими;
- взаимодействия;
- компиляционные;
- ресурсные;
- арифметические;
- среды выполнения.

Это – основная классификация сбоев в приложениях и операционных системах. Логические, синтаксические и «среды выполнения» встречаются в разработке чаще остальных. На них будет сделан основной акцент.

Ошибки синтаксиса

Синтаксические баги распространены среди новичков. Они относятся к категории «самых безобидных». С данной категорией ошибок способны справиться компиляторы тех или иных языков. Соответствующие инструменты показывают, где допущена неточность. Остается лишь понять, как исправить ее.

Синтаксические ошибки – ошибки синтаксиса, правил языка. Вот пример в Паскале:

```
WriteLn('Hello, World !!!')
ReadLn;
```

Код написан неверно. Согласно действующим синтаксическим нормам, в Pascal в первой строчке нужно в конце поставить точку с запятой.

Логические

Тут стоит выделить обычные и арифметические типы. Вторые возникают, когда программе при работе необходимо вычислить много переменных, но на каком-то этапе расчетов возникают неполадки или нечто непредвиденное. Пример – получение в результатах «бесконечности».

Логические сбои обычного типа – самые сложные и неприятные. Их тяжелее всего обнаружить и исправить. С точки зрения языка программа может быть написана идеально, но работать неправильно. Подобное явление – следствие логической ошибки. Компиляторы их не обнаруживают.

```
for i := 1 to 10 do
  if i = 15 then WriteLn('i = 15');
```

Выше – пример логической ошибки в программе. Тут:

1. Происходит сравнение значения i с 15.
2. На экран выводится сообщение, если I = 15.
3. В заданном цикле i не будет равно 15. Связано это с диапазоном значений – от 1 до 10.

Может показаться, что ошибка безобидная. В приведенном примере так и есть, но в более крупных программах такое явление приводит к серьезным последствиям.

Время выполнения

Run-time сбои – это ошибка времени выполнения программы. Встречается даже когда исходный код лишен логических и синтаксических ошибок. Связаны такие неполадки с ходом выполнения программного продукта. Пример – в процессе функционирования ПО был удален файл, считываемый программой. Если игнорировать подобные неполадки, можно столкнуться с аварийным завершением работы контента.

```
Write('y = ');
ReadLn(y);
x := 100 / y;
WriteLn('100 / ', y, ' = ', x);
```

Самый распространенный пример в данной категории – это неожиданное деление на ноль. Предложенный фрагмент кода с точки зрения синтаксиса и логики написан грамотно. Но, если клиент наберет 0, произойдет сбой системы.

Компиляционный тип
Встречается при разработке на языках высокого уровня. Во время преобразований в машинный тип «что-то идет не так». Причиной служат синтаксические ошибки или сбои непосредственно в компиляторе.

Наличие подобных неполадок делает бета-тестирование невозможным. Компиляционные ошибки устраняются при разработке-отладке.

Ресурсные
Ресурсный тип ошибок – это сбои вроде «переполнение буфера» или «нехватка памяти». Тесно связаны с «железом» устройства. Могут быть вызваны действиями пользователя. Пример – запуск «свежих» игр на стареньких компьютерах.

Исправить ситуацию помогают основательные работы над исходным кодом. А именно – полное переписывание программы или «проблемного» фрагмента.

Взаимодействие
Подразумевается взаимодействие с аппаратным или программным окружением. Пример – ошибка при использовании веб-протоколов. Это приведет к тому, что облачный сервис не будет нормально функционировать. При постоянном возникновении соответствующей неполадки остается один путь – полностью переписывать «проблемный» участок кода, ответственный за соответствующий баг.

Исключения и как избежать багов

Исключение – событие, при возникновении которых начинается «неправильное» поведение программы. Механизм, необходимый для стабилизации обработки неполадок независимо от типа ПО, платформ и иных условий. Помогают разрабатывать единые концепции ответа на баги со стороны операционной системы или контента.

Исключения бывают:

1. Программными. Они генерируются приложением или ОС.
2. Аппаратными. Создаются процессором. Пример – обращение к невыделенной памяти.

Исключения нужны для охвата критических багов. Избежать неполадок помогут отладчики на этапе разработки. А еще – своевременное поэтапное тестирование программы.

4.2. Отладка программ

Отладка – это процесс поиска и исправления ошибок или неполадок в исходном коде какого-либо программного обеспечения. Когда программное обеспечение не работает, как ожидалось, компьютерные программисты изучают код, чтобы выяснить причину появления ошибок. Они используют инструменты отладки для запуска программного обеспечения в контролируемой среде, пошаговой проверки кода, а также анализа и поиска проблем.

Где возник термин «отладка»?

Термин «отладка» берёт своё начало от адмирала Грейс Хоппер, которая работала в Гарвардском университете в 1940-х годах. Когда один из ее коллег обнаружил моль, мешающую работе одного из компьютеров университета, она сказала, что они отлаживают систему. Компьютерные программисты впервые стали использовать термины «ошибки» и «отладка» в 1950-х годах, а к началу 1960-х годов термин «отладка» стал общепринятым в сообществе программистов.

Почему отладка важна?

Недочеты и ошибки случаются в компьютерном программировании, поскольку это абстрактная и концептуальная работа. Компьютеры обрабатывают данные в виде электронных сигналов. Языки программирования абстрагируют эту информацию, чтобы люди могли более эффективно взаимодействовать с компьютерами. Любой тип программного обеспечения имеет несколько уровней абстракции, на которых различные компоненты взаимодействуют для правильной работы приложения. Когда возникают ошибки, найти и решить проблему может быть непросто. Инструменты и стратегии отладки помогают быстрее устранять проблемы и повышать производительность разработчиков. В результате улучшается как качество программного обеспечения, так и опыт конечного пользователя.

Как работает процесс отладки?

Процесс отладки обычно требует выполнения указанных ниже шагов.

Определение ошибки

Разработчики, тестировщики и конечные пользователи сообщают об ошибках, обнаруженных во время тестировании или использовании программного обеспечения. Разработчики определяют точную строку или модуль кода, вызывающий ошибку. Это может быть утомительно и отнимать много времени.

Анализ ошибки

Программисты анализируют ошибку, записывая все изменения состояния программы и значения данных. Они также определяют приоритет исправления ошибок на основе влияния ошибки на функциональность программного обеспечения. Кроме того, команда разработчиков программного обеспечения определяет график исправления ошибок в зависимости от целей и требований разработки.

Устранение и проверка

Разработчики исправляют ошибку и выполняют тесты, чтобы убедиться, что программное обеспечение продолжает работать в обычном режиме. Они могут написать новые тесты, чтобы проверить, повторяется ли ошибка в будущем.

Сравнение отладки и тестирования

Отладка и тестирование – это взаимодополняющие процессы, которые гарантируют, что программы работают должным образом. После написания полного раздела или части кода программисты выполняют тестирование, чтобы обнаружить недочеты и ошибки. Как только они обнаружены, кодировщики могут начать процесс отладки и работать над устранением ошибок в программном обеспечении.

Какие ошибки кодирования требуют отладки?

Дефекты программного обеспечения возникают из-за сложности, присущей разработке программного обеспечения. Кроме того, незначительные производственные ошибки наблюдаются после запуска программного обеспечения, потому что клиенты используют его неожиданным образом. Ниже мы приводим некоторые распространенные типы ошибок, которые зачастую требуют отладки.

Синтаксические ошибки

Синтаксическая ошибка возникает, когда в компьютерной программе неправильно указано значение. Это эквивалент опечатки или орфографической ошибки в текстовом редакторе. Если есть синтаксические ошибки, программа не будет компилироваться или запускаться. Как правило, программное обеспечение для редактирования кода выделяет эту ошибку.

Семантические ошибки

Семантические ошибки возникают из-за неправильного использования операторов программирования. Например, если вы переводите выражение $x/(2\ \pi)$ into Python, можете написать:

*y = x / 2 * math.pi*

Однако это утверждение неверно, поскольку умножение и деление имеют одинаковый приоритет в Python и вычисляются слева направо. Следовательно, это выражение вычисляется как (xπ)/2, а это приводит к ошибкам.

Логические ошибки

Логические ошибки возникают, когда программисты искажают поэтапный процесс или алгоритм компьютерной программы. Например, код может выйти из цикла слишком рано или может иметь неверный результат «если, то». Вы можете определить логические ошибки, выполнив пошаговую отладку коду для нескольких различных сценариев ввода/вывода.

Ошибки времени выполнения

Ошибки времени выполнения возникают из-за среды вычисления, в которой выполняется программный код. Примеры включают нехватку памяти или переполнение стека. Вы

можете устранить ошибки времени выполнения, заключив операторы в блоки try-catch или зарегистрировав исключение при помощи соответствующего сообщения.

Какие есть общие стратегии отладки?

Программисты используют несколько стратегий, чтобы свести к минимуму количество ошибок и сократить время отладки.

Пошаговая разработка программы

Пошаговая разработка – это разработка программ в управляемых разделах с целью тестирования небольших частей кода. Так программисты могут локализовать какие-либо найденные ошибки, а также могут работать над одной ошибкой за раз, а не над несколькими ошибками после написания больших участков кода.

Возврат

Возврат – это популярный метод отладки, в частности для небольших программ. Разработчики работают в обратном порядке с места, где произошла фатальная ошибка, чтобы определить точную точку ее возникновения в коде. К сожалению, этот процесс становится все более сложным, поскольку увеличивается количество строк кода.

Удаленная отладка

Удаленная отладка – это отладка приложения, работающего в отличной от вашей локальной машины среде. Например, вы можете использовать инструменты отладки, установленные удаленно, чтобы устранять ошибки.

Ведение журналов

Большинство компьютерных программ записывают внутренние данные и другую важную информацию, например время выполнения и состояние операционной системы, в файлы журналов. Разработчики изучают файлы журналов, чтобы находить и устранять ошибки. Кроме того, они используют такие инструменты, как анализаторы журналов, чтобы автоматизировать обработку файлов журналов.

Отладка в облаке

Отладка сложных облачных приложений – это сложная задача, поскольку разработчикам приходится эмулировать облачные архитектуры на локальных компьютерах. Дополнительные различия в конфигурации могут возникнуть между облачной и эмулируемой средой. Это приводит к большему количеству ошибок в производстве и более длительным циклам разработки. Чтобы отладка облака была более эффективной, требуются специальные инструменты.

Как AWS помогает с отладкой?

Есть несколько разных способов, при помощи которых AWS поддерживает кодирование и отладку.

Плагины для популярных IDE

Чтобы писать код, разработчики используют интегрированную среду разработки (IDE). AWS имеет несколько подключаемых модулей, совместимых с IDE, и поддерживает

процесс отладки. Например, Набор инструментов AWS для Eclipse – это подключаемый модуль с открытым кодом для Eclipse Java IDE, позволяющий упростить процессы разработки, развертывания и отладки Java-приложений с помощью Amazon Web Services. Аналогично AWS также предоставляет поддержку отладки для других языков с помощью подключаемых модулей, которые интегрируются с другими популярными IDE, такими как:

- PyCharm для Python.
- IntelliJ IDEA для Java и Python.
- Rider для .Net.
- WebStorm для JavaScript.

Эти модули поддерживают отладку в облаке, чтобы разработчики могли отлаживать приложения облаке, напрямую обращаясь к коду, работающему в облаке.

AWS X-Ray

AWS X-Ray – это инструмент отладки, который разработчики используют для анализа приложений в процессе разработки и производства. Он охватывает все: от простых трехуровневых приложений до сложных приложений микросервисов, состоящих из тысяч сервисов.

С X-Ray вы сможете:

- понимать, как работает ваше приложение и его основные сервисы;
- выявлять и устранять основные причины проблем с производительностью и возникновения ошибок;
- анализировать сквозное представление запросов по мере их прохождения через ваше приложение.

4.3. Тестирование программ

Тестирование программного обеспечения – это:

– процесс исследования ПО с целью получения информации о качестве продукта;

– **процесс проверки соответствия заявленных к продукту требований и реально реализованной функциональности, осуществляемый путем наблюдения за его работой в искусственно созданных ситуациях и на ограниченном наборе тестов, выбранных определенным образом;**

– оценка системы с тем, чтобы найти различия между тем, какой система должна быть и какой она есть.

В широком смысле, тестирование – это одна из техник контроля качества (Quality Control), которая включает планирование, составление тестов, непосредственно выполнение тестирования и анализ полученных результатов.

Важно понимать, что тестирование ПО включает не только собственно проведение тестов, но и многие другие действия, связанные с процессом обеспечения качества:

– анализ и планирование;

– разработку тестовых сценариев;

– оценку критериев окончания тестирования;

– написание отчетов;

– рецензирование документации (в том числе и исходного кода);

– проведение статического анализа.

4.4. Сопровождение программ

Сопровождение программ — это процесс улучшения, оптимизации и устранению дефектов ПО после передачи в эксплуатацию.Сопровождение ПО осуществляется сопроводителем.

Термин «Сопровождение ПО» IEEE (IEEE Standard 1216) - процесс модификации программной системы или ее компонентов, проводимый после поставки системы заказчику с целью устранения отказов, повышения производительности или улучшения других характеристик системы или адаптации к изменившемуся программному окружению.

При сопровождении в программное обеспечение вносятся следующие изменения:

1. Исправление ошибок.

   Подразумевается корректировка программ, ограниченных техническим заданием и документацией. Составляет около 20% общих затрат на сопровождение.

2. Адаптация программного обеспечения.

   Регламентирование документами программного обеспечения.
   Адаптация программного обеспечения к условиям конкретного использования, учитываются характеристики внешней среды и конфигурации аппаратуры.Занимает около 20% общих затрат на сопровождение.

3. Модернизация.

   Улучшение характеристик решения задач в соответствии с новым или дополнительным техническим заданием на программное изделие на программное изделие. До 60% общих затрат на сопровождение.

Сопровождение отнимает значительные ресурсы, как человеческие, так и финансовые (наем аутсорсинговых и консалтинговых компаний), а ведь хочется эти ресурсы использовать значительно эффективнее. Для многих типовая ситуация, когда лучшие специалисты, знающие систему, как свои пять пальцев, тратят массу времени на работу с пользователями, при этом, как правило, одни и те же пользователи и одни и те же проблемы.

Термин определяется как «процесс модификации программной системы или ее компонентов, проводимый после поставки системы заказчику с целью устранения отказов, повышения производительности или улучшения других характеристик системы или адаптации к изменившемуся программному окружению».

Из данного определения можно выделить основные направления:

Корректирующее сопровождение — это исправление или обход ошибок и недочетов, выявленных в ходе эксплуатации программного обеспечения. На этой стадии пользователи ожидают от специалистов максимально оперативного решения проблем; а в идеале выявление ошибок вообще не должно касаться бизнес-пользователей. Количество выявленных проблем автоматически снижает удовлетворенность пользователей и повышает напряженность внутри компании.

Улучшающее сопровождение — это дополнение программного продукта новыми функциями. Требования на расширение функционального охвата системы обычно исходят от пользователей и аналитиков. При этом задачи улучшающего сопровождения, как правило, выделяются в отдельные проекты с самостоятельными бюджетами, сроками и ответственными лицами.

Адаптивное сопровождение — можно определить как внесение изменений в работающее приложение, необходимое для поддержки новых программных и аппаратных средств. При этом функциональность системы не нуждается в расширении. Приложение должно выполнять старые функции в новых условиях.

Профилактическое сопровождение (preventive maintenance) — это модификация программного продукта на этапе эксплуатации для идентификации и предотвращения скрытых дефектов до того, как они приведут к реальным сбоям.Данное определение справедливо для фирм-разработчиков или для ИТ-подразделений, осуществляющих полный цикл разработки приложений с «нуля». Но для бизнеса подобный подход часто является неоправданно дорогим, ведь приобретение тиражного продукта всегда дешевле самостоятельной разработки. В связи с этим при сопровождении бизнес-приложений существует еще одно направление сопровождения, а именно обновления от вендора; в профессиональном мире это называют миграцией версий ПО. Разработчик тиражируемого решения регулярно поставляет своим пользователям новые версии, в которых расширяется автоматизируемый функционал, исправляются обнаруженные ошибки, приложение адаптируется для поддержки изменений внешней среды (это очень характерно для бухгалтерских приложений, существующих в условиях нестабильности законодательства).

Для данного направления характерны достаточно серьезные проблемы, связанные с совместимостью изменений, поступивших от поставщика решения, с изменениями, ранее внесенными собственной службой сопровождения фирмы-пользователя. Анализ данной

ситуации для аналитиков является огромной головной болью и требует от команды сопровождения высокого уровня документирования. Многие компании в связи с высокой трудоемкостью подобной функции вообще отказались от этого направления сопровождения. В такой ситуации предприятие-пользователь теряет возможность использовать ресурс разработчика. Вот живой пример: изменяются правила формирования книги покупок, при этом компания-пользователь, поддерживающая сервис обновления, получает от компании-разработчика соответствующее обновление практически бесплатно, а компания, не поддерживающая данный сервис, вынуждена производить соответствующую разработку самостоятельно.

Существует еще и другая сторона сопровождения, также поглощающая немало человеко-часов, — это управление сопровождением. Независимо от того, к какому типу сопровождения относится требование, и от кого оно поступило, от пользователя или от аналитика, требование это должно пройти определенный процесс согласований, меняя свой внутренний статус в зависимости от проведенных работ. Не секрет, что часто трудоемкость, связанная с согласованием требования, равна, а иногда и превышает трудоемкость его реализации.

Процесс управления требованиями представляет собой последовательность действий по регистрации, отслеживанию, анализу, принятию по нему решений, реализации, проверке и закрытию. Этот процесс требует принятия ряда решений руководителями различных подразделений и обмена информацией о поставленных задачах и произведенных работах между заинтересованными лицами.

Сценарии работ по сопровождению

Как должно быть

Рассмотрим обычный сценарий работы с требованиями к программному продукту. Например, у пользователя возникла проблема. Он звонит по телефону или пишет письмо по электронной почте в службу сопровождения и сообщает о ней. Назовем это письмо «Требованием», оно должно быть зарегистрировано, пользователю сообщается код регистрации. Далее это требование должно быть передано экспертам для анализа. В случае, если есть возможность решить проблему «мирным» путем, то есть данное требование не порождает задач разработки программного кода или дополнительных регламентов, проблема может быть решена путем краткой консультации для пользователя. Данная консультация в письменном виде, факсом или по электронной почте предоставляется пользователю и требование закрывается. В противном случае руководитель службы сопровождения определяет трудоемкость и техническую возможность исполнения данных работ. Координационный совет осуществляет определение бюджета реализации и приоритетов реализации данного требования. Далее к данному требованию формируются задачи, определяются исполнители, устанавливаются планы и т.д.

Как часто бывает

Обычная картина для неформализованного процесса сопровождения — специалист по системе затрачивает значительную часть своего рабочего времени на общение с

пользователями по телефону, на разъяснение вопросов, касающихся эксплуатации ПО. При этом поступающая информация в лучшем случае фиксируется в Excel-файле и по мере внесения исправлений удаляется, а в худшем — вообще записывается на клочках бумаги, периодически куда-то пропадающих.

Налицо непродуктивная трата времени квалифицированного специалиста; вдобавок к этому возникают трудности с планированием работ по устранению замечаний. У руководителя нет полной информации по выявленным дефектам, нет сводной статистики по замечаниям, которые уже исправлены или еще нуждаются в рассмотрении.

Кто в подобных случаях расставляет приоритеты? Сам специалист сопровождения — он принимает требования и решает, что делать в первую очередь, а что отодвинуть на потом. Это должен быть чрезвычайно ответственный человек, который болеет за свое дело и аккуратно складывает распечатки дефектов в папочку, а таких среди программистов, в массе своей — приверженцев разумного хаоса, еще нужно поискать. В течение года такой разработчик забивает бумагами ящики своего рабочего стола, и гарантии, что хотя бы одно замечание пользователя при этом не потеряется, — никакой.

Но на самом деле это полбеды, настоящая беда случится, когда этот специалист придет к своему начальнику и скажет, что хочет от этой жизни чего-то большого и светлого, и в его жизненные планы никоим образом не входило просиживание штанов в попытках объяснить «тупому» пользователю, кто есть кто. И, либо его должны оградить от пользователей, либо он найдет себе новую творческую работу. А при текущем состоянии рынка труда действительно найдет, не сильно утруждая себя, и при этом еще и на более высокую зарплату. А теперь представьте себе, как новый сотрудник остается один на один с системой, с пользователями и бардаком, оставленным предшественником. Сколько у него уйдет времени на разбор ситуации и на какое время ваша корпоративная система останется без сопровождения вообще и без качественного сопровождения, в частности?

Есть еще одна проблема, которая возникает в результате неуправляемого процесса сопровождения, — часто такое сопровождение закачивается коллапсом системы, пусть и не в прямом смысле этого слова. Систему настолько набивают заплатками, что буквально еще одну она перенести уже не в состоянии, а на полный рефакторинг нет ни времени, ни ресурсов, ни документации. Автор видел такие системы неоднократно, причем в подобное состояние они приводились всего за два года сопровождения двумя вполне работоспособными специалистами. Причем специалисты были неплохие и работали они иногда по 12 часов в день, — пользователи-то просят сделать как можно быстрее и как можно проще.

Пока нет вала замечаний, и со всеми проблемами справляется один специалист, а система достаточно проста, с этим еще можно как-то мириться, но когда объемы возрастают, требуется кардинальное решение проблемы. Необходим инструмент, который позволит сделать сопровождение прозрачным, будет поддерживать необходимую организацию работ, запрещая отступать от правил и не требуя больших усилий на поддержание процесса, и, конечно же, содержащий хранилище всех полученных данных о требованиях и истории их обработки в одном месте для быстрого и беспрепятственного доступа команды сопровождения — от руководителя до программиста.

Управление требованиями в программном продукте

Управление внедрением и сопровождением

> Модель управления требованиями, автоматизирующая следующий бизнес-процесс:

Подобный подход к управлению требованиями позволяет гарантированно вести учет требований пользователей, отслеживать процедуры закрытия требований пользователей, регулировать поток требований, проходящих через процедуры согласования и бюджетного управления.

Документирование подлежащих исполнению работ происходит на уровне связки «Задачи» с объектами метаданных.

Данный подход позволяет сформировать следующую цепочку данных:

Такая информация, размещенная в базе данных, не обременяет пользователей — в большинстве случаев пользователю все равно приходится писать письмо на электронную почту — она не обременяет разработчиков, — они все равно делают отчет о своей работе. Но при этом данный инструмент позволяет получить информацию о всех работах по объекту метаданных, в частности, для кого это делалось, зачем и кто это делал, — что является бесценной информацией для анализа произведенных в системе изменений.

Не менее важным фактором является экономическая оценка требований пользователей через сопоставление трудозатраты ИТ-подразделения и эффективности результата. В целом любую функцию автоматизации управления можно всегда проанализировать по следующим показателям:

Количество операций в месяц.

Трудоемкость исполнения без автоматизации.

Стоимость исполнения функции без автоматизации (рассчитывается как трудоемкость × количество × ставка исполнителя).

Трудоемкость исполнения с автоматизацией.

Стоимость исполнения с автоматизацией (рассчитывается как трудоемкость × количество × ставка исполнителя).
Экономия на автоматизации.

Затраты на автоматизацию (трудоемкость реализации × ставку исполнителя).

Значимость человеческого фактора (определяет коэффициент, повышающий значимость автоматизации в связи с возможными ошибками оператора при неавтоматизированной подготовке информации).

Требования к данной информации в других процессах (коэффициент, повышающий важность автоматизации с точки зрения использования данной информации в других процессах).

Таким образом, можно вывести экономическую эффективность любого функционального требования, но, увы, кроме экономики на предприятиях существует еще и политика. В связи с этим ограничивается процедура принятия требования лишь процедурой бюджетного согласования, но наша практика показывает, что даже подобное ограничение снижает поток требований в среднем в два раза, просто благодаря введению процедуры, при которой держатель бюджета сопоставляет содержание требования с затратами на его исполнение и отсекает необоснованные требования.

Автоматизация процесса сопровождения описанным способом может вдвое улучшить качество сопровождения системы, снизить риски, связанные с текучкой кадров в организации и оптимизировать распределение работ внутри службы поддержки.

4.5. Защита программ

Защита программного обеспечения подразумевает под собой создание таких условий, при которых невозможно нелегальное копирование, распространение и использование программного продукта.

Актуальность вопроса обусловлена внушительным процентом пиратской (нелегальной) продукции, широко распространяющейся в сети Интернет.

Из-за несовершенства законодательных систем многих стран привлечь к ответственности или найти распространителя нелицензионного продукта проблематично.

Ситуация защиты программного обеспечения

Защита программного обеспечения в России приобрела широкое распространение только в последние пять лет, когда этот вопрос был впервые поднят на международном уровне.

Тем не менее, предпосылки к редуцированию пиратского ПО можно было наблюдать и значительно раньше.

В графике выше можно заметить, что стремительное падение позиций нелицензионного ПО началось с 2006 года.

Эти показатели обуславливаются несколькими факторами:

1. **Социальный.** Наличие лицензионного программного обеспечения стало признаком определенного статуса, появилась массовая тенденция оплаты труда разработчикам ПО.
2. **Качественный.** Легальное ПО дает больше возможностей и требует меньше времени и усилий для установки.
3. **Улучшенная защита программного обеспечения.** Механизмы защиты совершенствуются с каждым годом. Это позволяет сделать взлом настолько трудоемким и длительным, что затраты по времени не оправдывают себя.

Тем не менее, по данным международной ассоциации Business Software Alliance, среднемировой показатель пиратства в 2014 году составил 42%, а в Российской Федерации – 64%.

Такая ситуация наносит ощутимый ущерб деловым отношениям с производителями программного обеспечения, которые не желают продвигать свою продукцию на рынке с таким высоким показателем нелегального распространения ПО. Важно заметить, что показатели РФ являются далеко не худшими.

Методы и проблемы защиты программного обеспечения

Говоря о защите ПО, следует уточнить, что на данный момент все методы относятся к средствам программной защиты.

Это обусловлено переходом пользователей с физических на цифровые и облачные хранилища информации и файлов.

Наиболее распространенный способ защиты – серийный номер. Для каждой единицы программного продукта существует свой уникальный код активации. Тем не менее, такой метод можно обойти. Как правило, для этого используется способ дизассемблирования и последующего сохранения программы без проверяющего подлинность модуля.

Аналогом является цифровой ключ. Его подделать сложнее, так как он высылается разработчиком непосредственно получателю и имеет вшитые команды и информацию, без которой запуск и регистрация программы невозможна.

Сетевой мониторинг действенен в случае, если часть функционала ПО выполняется на удаленном сервере. Необходимо будет каждый раз высылать серийный номер. В случае наличия в сети двух одинаковых номеров функционирование программы прекращается.

На рынке не так давно появился ряд продуктов, направленных на непосредственную защиту кода ПО.

Такие приложения:

- шифруют модули;
- обеспечивают защиту внутренних функций и переменных;
- не позволяют запускать программу под отладчиком или в виртуальных машинах;
- позволяют делать привязку к оборудованию.

Все вышеперечисленные функции тяжело, а иногда и невозможно обеспечить программно.

Защита программного обеспечения необходима не только с экономической точки зрения. В нелицензионное ПО могут быть вшиты вредоносные приложения, что несет прямую угрозу всем компьютерам, находящимся в одной сети с зараженным.

Нередко через необходимое для работы нелегальное ПО компьютер пользователя становится частью бот-нета.

4.6. Пакеты прикладных программ

Паке́т прикладны́х програ́мм, (ППП), комплекс взаимосвязанных, дополняющих друг друга программных приложений или модулей, обслуживающий потребности какого-либо вида деятельности человека.

Например, пакет Microsoft Office, включающий приложения Word (текстовый процессор), Excel (электронные таблицы), Outlook (электронная почта) и другие, обслуживает деятельность офисного работника. При проектировании программного продукта в виде ППП на этапе декомпозиции (выделения основных составных частей) осуществляется формирование расширяемого набора программных приложений или модулей, покрывающего данную область. Покрытие предметной области означает, что для любой задачи (из данной области) либо найдётся уже готовое обслуживающее её программное приложение или искомое приложение можно будет построить в форме надлежащим образом организованного подмножества модулей пакета. Основное отличие ППП от библиотеки программ состоит в том, что библиотека не покрывает предметную область – она обслуживает только программистов, которые включают в свои программы обращения к библиотечным модулям, а с ППП может работать как программист, так и конечный пользователь, незнакомый с программированием. Достоинством ППП является возможность его расширения, которая достигается за счёт того, что на стадии проектирования определяют наиболее вероятные направления развития пакета, и каждому из этих направлений ставится в соответствие свой пополняемый набор однородных программных модулей или приложений. В результате развитие пакета удаётся свести к подключению новых модулей или приложений без какого-либо редактирования существующих текстов программ, обеспечивая тем самым сохранность и работоспособность отлаженного ранее кода ППП. Преимущества пакетного подхода наглядно проявляются, в частности, при программировании задач вычислительного эксперимента, где основной объём работ приходится не на создание первоначальной версии программы, а на многократные модификации программного кода (отражающие эволюцию математической модели и методов её расчёта). Если на стадии проектирования в структуре программы были предусмотрены места (т. н. гнёзда), предназначенные для подстановки варьируемых однородных модулей, то появление нового варианта модели или метода потребует лишь создания ещё одного однородного программного модуля, подставляемого в соответствующее гнездо, а весь написанный и отлаженный ранее код останется без изменений.

ППП – один из самых многочисленных видов современных программных продуктов. В начале 21 в. наибольшее распространение получили ППП, предназначенные для автоматизации офисной, банковской, финансовой, издательской деятельности, а также пакеты для управления телекоммуникационными системами, базами данных, системами автоматизированного проектирования, графическими редакторами и др.

5.1. Общая характеристика инструментальных средств разработки программ

Трудоемкость разработки ИИС в значительной степени зависит от используемых инструментальных средств. Инструментальные средства для разработки интеллектуальных приложений можно классифицировать по следующим основным параметрам:

- уровень используемого языка;
- парадигмы программирования и механизмы реализации;
- способ представления знаний;
- механизмы вывода и моделирования;

• средства приобретения знаний;

• технологии разработки приложений.

Уровень используемого языка. Мощность и универсальность языка программирования определяет трудоемкость разработки ЭС.

1. Традиционные (в том числе объектно-ориентированные) языки программирования типа С, С++ (как правило, они используются не для создания ЭС, а для создания инструментальных средств).

2. Специальные языки программирования (например, язык LISP, ориентированный на обработку списков; язык логического программирования PROLOG; язык рекурсивных функций РЕФАЛ и т.д.). Их недостатком является слабая приспособленность к объединению с программами, написанными на языках традиционного программирования.

3. Инструментальные средства, содержащие многие, но не все компоненты ЭС (например, система OPS 5, которая поддерживает продукционный подход к представлению знаний; языки KRL и FRL, используемые для разработки ЭС с фреймовым представлением знаний). Такое программное обеспечение предназначено для разработчиков, владеющих технологиями программирования и умеющих интегрировать разнородные компоненты в программный комплекс.

4. Оболочки ЭС общего назначения, содержащие все программные компоненты, но не имеющие знаний о конкретных предметных средах. Средства этого типа и последующего не требуют от разработчика приложения знания программирования. Примерами являются ЭКО, Leonardo, Nexpert Object, Kappa, EXSYS, GURU, ART, KEE и др. В последнее время все реже употребляется термин «оболочка», его заменяют более широким термином «среда разработки». Если хотят подчеркнуть, что средство используется не только на стадии разработки приложения, но и на стадиях использования и сопровождения, то употребляют термин «полная среда» (complete environment). Для поддержания всего цикла создания и сопровождения программ используются интегрированные инструментальные системы, например KEATS, VITAL. Основными компонентами системы KEATS являются: ACQUIST - средства фрагментирования текстовых источников знаний, позволяющие разбивать текст или протокол беседы с экспертом на множество взаимосвязанных, аннотированных фрагментов и создавать понятия (концепты); FLIK — язык представления знаний средствами фреймовой модели; GIS — графический интерфейс, используемый для создания гипертекстов и концептуальных моделей, а также для проектирования фреймовых систем;

ERI — интерпретатор правил, реализующий процедуры прямого и обратного вывода; TRI — инструмент визуализации логического вывода, демонстрирующий последовательность выполнения правил; TaBles — интерфейс манипулирования таблицами, используемыми для хранения знаний в БЗ; CS — язык описания и распространения ограничений; TMS — система поддержания истинности.

При использовании инструментария данного типа могут возникнуть следующие трудности:

а) управляющие стратегии, заложенные в механизм вывода, могут не соответствовать методам решения, которые использует эксперт, взаимодействующий с данной системой, что может привести к неэффективным, а возможно, и неправильным решениям;

б) способ представления знаний, используемый в инструментарии, мало подходит для описания знаний конкретной предметной области.

Большая часть этих трудностей разрешена в проблемно/предметно-ориентированных средствах разработки ИИС.

5. Проблемно/предметно-ориентированные оболочки и среды (не требуют знания программирования):

• проблемно-ориентированные средства — предназначены для решения задач определенного класса (задачи поиска, управления, планирования, прогнозирования и др.) и содержат соответствующие этому классу альтернативные функциональные модули;

• предметно-ориентированные средства — включают знания о типах предметных областей, что сокращает время разработки БЗ.

При использовании оболочек и сред разработчик приложения полностью освобождается от программирования, его основные трудозатраты связаны с формированием базы знаний.

Парадигмы программирования и механизмы реализации. Способы реализации механизма исполняемых утверждений часто называют парадигмами программирования. К основным парадигмам относят следующие:

• процедурное программирование;

• программирование, ориентированное на данные;

• программирование, ориентированное на правила;

• объектно-ориентированное программирование.

Парадигма процедурного программирования является самой распространенной среди существующих языков программирования (например, С и Паскаль). В процедурной парадигме активная роль отводится процедурам, а не данным; причем любая процедура активизируется вызовом. Подобные способы задания поведения удобны для описаний последовательности действий одного процесса или нескольких взаимосвязанных процессов.

При использовании программирования, ориентированного на данные, активная роль принадлежит данным, а не процедурам. Здесь со структурами активных данных связывают некоторые действия (процедуры), которые активизируются тогда, когда осуществляется обращение к этим данным.

В парадигме, ориентированной на правила, поведение определяется множеством правил вида «условие-действие». Условие задает образ данных, при возникновении которого действие правила может быть выполнено. Правила в данной парадигме играют такую же роль, как и операторы в процедурной парадигме. Однако если в процедурной парадигме поведение задается последовательностью операторов, не зависящей от значений обрабатываемых данных, то в парадигме, ориентированной на правила, поведение не задается заранее предписанной последовательностью правил, а формируется на основе значений данных, которые в текущий момент обрабатываются программой. Подход, ориентированный на правила, удобен для описания поведения, гибко и разнообразно реагирующего на большое многообразие состояний данных.

Парадигма объектного программирования в отличие от процедурной парадигмы не разделяет программу на процедуры и данные. Здесь программа организуется вокруг сущностей, называемых объектами, которые включают локальные процедуры (методы) и локальные данные (переменные). Поведение (функционирование) в этой парадигме организуется путем пересылки сообщений между объектами. Объект, получив сообщение, осуществляет его локальную интерпретацию, основываясь на локальных процедурах и данных. Такой подход позволяет описывать сложные системы наиболее естественным образом. Он особенно удобен для интегрированных ЭС.

Способ представления знаний. Наличие многих способов представления знаний вызвано стремлением представить различные типы проблемных сред с наибольшей эффективностью. Обычно способ представления знаний в ЭС характеризуют моделью представления знаний. Типичными моделями представления знаний являются правила

(продукции), фреймы (или объекты), семантические сети, логические формулы. Инструментальные средства, имеющие в своем составе более одной модели представления знаний, называют гибридными. Большинство современных средств, как правило, использует объектно-ориентированную парадигму, объединенную с парадигмой, ориентированной на правила.

Механизмы вывода и моделирования. В статических ЭС единственным активным агентом, изменяющим информацию, является механизм вывода экспертной системы. В динамических ЭС изменение данных происходит не только вследствие функционирования механизма исполняемых утверждений, но также в связи с изменениями окружения задачи, которые моделируются специальной подсистемой или поступают извне. Механизмы вывода в различных средах могут отличаться способами реализации следующих процедур.

1. Структура процесса получения решения:

• построение дерева вывода на основе обучающей выборки (индуктивные методы приобретения знаний) и выбор маршрута на дереве вывода в режиме решения задачи;

• компиляция сети вывода из специфических правил в режиме приобретения знаний и поиск решения на сети вывода в режиме решения задачи;

• генерация сети вывода и поиск решения в режиме решения задачи, при этом генерация сети вывода осуществляется в ходе выполнения операции сопоставления, определяющей пары «правило — совокупность данных», на которых условия этого правила удовлетворяются;

• в режиме решения задач ЭС осуществляет выработку правдоподобных предположений (при отсутствии достаточной информации для решения); выполнение рассуждений по обоснованию (опровержению) предположений; генерацию альтернативных сетей вывода; поиск решения в сетях вывода.

2. Поиск (выбор) решения:

• направление поиска — от данных к цели, от целей к данным, двунаправленный поиск;

• порядок перебора вершин в сети вывода — «поиск в ширину», при котором сначала обрабатываются все вершины, непосредственно связанные с текущей обрабатываемой вершиной G; «поиск в глубину», когда сначала раскрывается одна наиболее значимая вершина – G_1 связанная с текущей G, затем вершина G_1 делается текущей, и для нее раскрывается одна наиболее значимая вершина G_2 и т. д.

3. Процесс генерации предположений и сети вывода:

• режим — генерация в режиме приобретения знаний, генерация в режиме решения задачи;

• полнота генерируемой сети вывода — операция сопоставления применяется ко всем правилам и ко всем типам указанных в правилах сущностей в каждом цикле работы механизма вывода (обеспечивается полнота генерируемой сети); используются различные средства для сокращения количества правил и (или) сущностей, участвующих в операции сопоставления; например, применяется алгоритм сопоставления или используются знания более общего характера (метазнания).

Механизм вывода для динамических проблемных сред дополнительно содержит: планировщик, управляющий деятельностью ЭС в соответствии с приоритетами; средства, гарантирующие получение лучшего решения в условиях ограниченности ресурсов; систему поддержания истинности значений переменных, изменяющихся во времени.

В динамических инструментальных средствах могут быть реализованы следующие варианты подсистемы моделирования:

• система моделирования отсутствует;

• существует система моделирования общего назначения, являющаяся частью инструментальной среды;

• существует специализированная система моделирования, являющаяся внешней по отношению к программному обеспечению, на котором реализуется ЭС.

Средства приобретения знаний. В инструментальных системах они характеризуются следующими признаками:

1. Уровень языка приобретения знаний:

• формальный язык;

• ограниченный естественный язык;

• язык пиктограмм и изображений;

• ЕЯ и язык изображений.

2. Тип приобретаемых знаний:

• данные в виде таблиц, содержащих значения входных и выходных атрибутов, по которым индуктивными методами строится дерево вывода;

• специализированные правила;

• общие и специализированные правила.

3. Тип приобретаемых данных:

• атрибуты и значения;

• объекты;

• классы структурированных объектов и их экземпляры, получающие значения атрибутов путем наследования.

5.2. Применение CASE-средств

Программная инженерия (Software Engineering, SE) – новое направление проектирования и реализации ИС.

По мнению ряда специалистов, *программная инженерия должна содержать* аспекты программной разработки, управления программным обеспечением, организации и использования проектов. Другие включают в её состав вопросы разработки программного обеспечения, проектирования, кодирования и тестирования вместе с использованием наилучших практических решений. Будем придерживаться мнения, что "***программная инженерия"*** включает совокупность современных методов проектирования и реализации ИС.

Разработка информационных систем в последние годы является распространённой и важной задачей. Очевидно, что программные инженеры должны быть способными разрабатывать ПО с помощью наилучших практических решений с долговременной перспективой.

Наиболее адекватной представляется разработка архитектуры сложных прикладных информационных систем на основе эффективного объединения разных видов оборудования и программного обеспечения, применения стандартизованных интерфейсов между компонентами системы и т.п. Такой подход позволяет повторно использовать программные средства на разных вычислительных платформах без перепрограммирования и тем самым экономить значительные финансовые средства, а также поэтапно наращивать вычислительную мощность прикладной системы в соответствии с потребностями пользователя и его финансовыми возможностями.

Проекты средней, высокой сложности и уникальные рекомендуется создавать с помощью CASE-средств, целесообразность применения которых определяется возможностью точного учёта требований конечного пользователя к проектируемой ИС, значительным снижением уровня системных ошибок в проекте до начала программирования и тем самым снижением общей трудоемкости разработки и особенно отладки программ.

Термин **CASE** (Computer Aided Software Engineering) первоначально означал решение вопросов автоматизации разработки программного обеспечения. В настоящее время это понятие охватывает процессы разработки сложных ИС в целом.

Термин **CASE-средства** означает программные средства, поддерживающие процессы создания и сопровождения ИС, включая анализ и формулировку требований, проектирование прикладного ПО (приложений) и БД, генерацию кода, тестирование, документирование, обеспечение качества, конфигурационное управление и управление проектом, а также другие процессы.

CASE-средства вместе с системным ПО и техническими средствами образуют *полную среду разработки ИС*.

Классификация CASE-средств
Современные CASE-средства можно классифицировать по типам и категориям. Классификация по типам отражает функциональную ориентацию CASE-средств на процессы жизненного цикла ИС. Классификация по категориям определяет степень интегрированности по выполняемым функциям и включает отдельные локальные средства, решающие небольшие автономные задачи, набор частично интегрированных средств, охватывающих большинство этапов жизненного цикла ИС (ЖЦ ИС) и полностью интегрированные средства, поддерживающие весь ЖЦ ИС и связанные общим *репозитарием* (англ. “repository” – объектно-ориентированное хранилище).

Современные реализации CASE-средств направлены на создание интегрированной среды комплексной автоматизации процессов проектирования, разработки и сопровождения, реализующих некоторую методологию проектирования ИС. Как правило, они ориентированы на решение задач комплексной автоматизации процесса разработки и сопровождения ИС.

Обычно *результат проектирования с помощью CASE* – проектная документация, а в некоторых случаях и прототип интерфейса с конечным пользователем.

Интегрированные CASE-средства обычно поддерживают стандарт, регламентирующий состав и содержание проектной документации на программное средство или ИС (глава 5). Одновременно этот стандарт фактически регламентирует и модель жизненного цикла ИС (глава 2).

Основой реализации CASE-технологий является репозитарий, доступ к которому имеют все подсистемы. Хранилище содержит сведения о каждом элементе проекта отдельно вне зависимости от способа их получения: из графического редактора или таблиц.

Внедрение CASE-средств
Потребность внедрения CASE-средств определяется достижением понимания нужд организации и технологических процессов. Оно должно привести к выделению тех областей деятельности организации, в которых применение CASE-средств принесет реальную пользу.

Результатом исследования возможностей применения CASE-средств для создания ИС является документ, определяющий стратегию внедрения CASE-средств.

Процесс внедрения CASE-средств состоит из следующих этапов:

1. определение потребностей в CASE-средствах;
2. оценка и выбор CASE-средств;
3. выполнение пилотного проекта;
4. практическое внедрение CASE-средств.

Внедрение CASE-средств не ограничивается только их использованием. Оно охватывает планирование и реализацию множества технических, организационных, структурных процессов, изменений в общей культуре организации, и основано на четком понимании возможностей CASE-средств.

Направления развития CASE-средств

Направления развития CASE-средств определяются потребностями практики. Обычно они ориентированы на:

- расширение применяемых моделей описания автоматизируемых систем;
- охват автоматизацией новых архитектур ИС;
- более глубокий уровень контроля целостности проекта;
- интеграцию с многими СУБД и CASE;
- использование новых платформ, прежде всего рабочих станций;
- развитие графических, гипертекстовых и мультимедийных компонент.

Несмотря на *высокие потенциальные возможности CASE-технологии* (увеличение производительности труда, улучшение качества программных продуктов, поддержка унифицированного и согласованного стиля работы) не все разработчики информационных систем достигают ожидаемых результатов. Основной причиной неудач является недопонимание сути программирования ИС с применением CASE-средств.

6.1. Организация работ при коллективной разработке программных продуктов

Модель группы и иерархическая модель. Обязанности членов группы. Модель проектной группы. Менеджер продукта. Менеджер программы. Разработчик. Тестер. Инструктор. Логистик. Размеры группы и масштаб проекта. Повышение эффективности. коллективной работы.

Модель группы и иерархическая модель

В настоящее время сложность промышленных приложений и систем такова, что процесс их разработки стал практически неуправляемым. Кроме того, их развертывание на сотнях компьютеров, расположенных в разных местах, значительно раздвигает границы процесса разработки.

Один человек не способен создать приложение масштаба предприятия. Ни один разработчик просто не удержит в голове все требования к системе и варианты проекта. Поэтому сегодня разработкой промышленных систем занимаются проектные группы, и все обязанности распределяются среди членов группы.

Существует две основные модели организации коллектива при разработке ПО:

1) иерархическая модель

2) модель группы

Работа в коллективе отличается определенными сложностями. Прежде всего, коллектив хочет знать «кто начальник». Ответ на этот вопрос дает иерархическая модель организации, определяющая начальников и подчиненных. Однако, если в современных производственных средах один менеджер проекта отвечает за все тонкости разработки и принимает все важные решения, возникает множество проблем, ведущих к провалу проекта. **Иерархическая модель грешит множеством недостатков:**

- нехваткой информации;
- невозможностью учесть все особенности проекта;
- отсутствием полноценной связи между всеми участниками проекта, так как вся информация идет в одном направлении — вверх по иерархии, к главному менеджеру;
- трудностью освоения новых технологий, необходимых при создании кроссплатформенных приложений;
- сложностью расстановки приоритетов.

Кроме того, опыта одного человека чаще всего недостаточно для быстрого решения задачи и для интеграции приложения в существующую инфраструктуру.

В организациях, построенных на основе иерархической модели, затруднен обмен информацией — в этой модели он, по определению, осуществляется через посредников. Вся информация иерархических групп «фильтруется» тремя или четырьмя менеджерами, что значительно повышает вероятность утери самого важного. Часто такое отсеивание идей происходит при прохождении сообщения от разработчика, непосредственно занимающегося проектом, к высшему руководству. Естественно, некоторые участники «выпадают» из процесса, что снижает эффективность их труда и повышает вероятность провала проекта.

Дабы сгладить недостатки иерархической модели, в проектной группе предусматривается распределение обязанностей руководителя между членами коллектива. При этом за проект отвечает не один человек, а все члены группы — каждый за свой участок.

Модель группы не определяет структуру коллектива с точки зрения отдела кадров. Ведь в такую разностороннюю группу привлечены ресурсы из разных отделов организации. Задача модели проектной группы — определить цели проекта и распределить обязанности. Руководители каждого направления с помощью выделенных им ресурсов выполняют возложенную на них часть работы. Обязанности ролей определяются работой над проектом, а не деятельностью «штатной единицы». При этом руководители направлений выполняют свои обычные функции: составляют график выплаты премий, распределяют отпуска и контролируют эффективность работы сотрудников. Начальник может оценить степень участия и эффективность работы сотрудников в проектной группе, но это — прерогатива менеджера конкретного сотрудника, а не проектной группы.

И у коллективного подхода есть недостатки:

- разрозненная связь с внешними источниками информации;
- несогласованное представление о разных сторонах проекта;
- несогласованность личных планов членов группы;
- отсутствие опыта, снижающее эффективность коллективной работы.

Обязанности членов группы

MSF — не готовое решение, а каркас, который можно адаптировать для нужд любой организации. Один из элементов этого каркаса - *модель проектной группы.* Она описывает структуру группы и принципы, которым надо следовать для успешного выполнения проекта.

Хотя модель группы разработчиков весьма конкретна, при знаков знакомстве с MSF ее нужно рассматривать в качестве отправной точки. Разные коллективы реализуют этот каркас по-разному, в зависимости от масштаба проекта, размеров группы и уровня подготовки ее членов.

Чтобы проект считался удачным, следует решить определенные задачи: I

- **удовлетворить требования заказчика** — проект должен выполнить требования заказчиков и пользователей, иначе ни о каком успехе не может быть и речи, возможна ситуация, когда бюджет и график соблюдены, но проект провалился, так как не выполнены требования заказчика;
- **соблюсти ограничения** — разработчики проекта должны уложиться в финансовые и временные рамки;
- **выполнить спецификации, основанные на требованиях пользователей** — спецификации — это подробное описание продукта, создаваемое группой для заказчика; они представляют собой соглашение между проектной группой и клиентом и регулируют вопросы, касающиеся приложения, в основе этого требования лежит принцип «сделать все, что обещано»;
- **выпустить продукт только после выявления и устранения всех проблем** — не существует программ без дефектов, однако группа должна найти и устранить их до выпуска продукта в свет, причем устранением ошибки считается не только ее исправление,

но и, например, занесение в документацию способа ее обхода; даже такой способ устранения проблем предпочтительнее, чем выпуск приложения, содержащего невыявленные ошибки, которые в любой момент могут преподнести неприятный сюрприз пользователям и разработчикам;

· **повысить эффективность труда пользователей** — новый продукт должен упрощать работу пользователей и делать ее более эффективной. Поэтому приложение, обладающее массой возможностей применять которые сложно или неудобно, считается провальным;

· **гарантировать простоту развертывания и управления —** эффективность развертывания непосредственно влияет на оценку пользователем качества продукта, например, ошибка в программе установки может создать у пользователей впечатление, что и само приложение небезгрешно, от проектной группы требуется не только подготовить продукт к развертыванию и гладко провести его, но и обеспечить пользователей поддержкой, организовав сопровождение приложения.

Примечание. На проект влияет множество факторов, некоторые из которых создают дополнительные ограничения. Когда проектная группа рассматривает на совещаниях цели проекта и график его создания и выпуска, следует убедиться, что проект окупится, то есть удастся выпустить нужный продукт, не превышая запланированных расходов.

Для достижения этих целей в модели проектной группы выполняемые задачи распределяются по шести ролям: менеджмент продукта, менеджмент программы, разработка, тестирование, обучение пользователей и логистика. Люди, выполняющие конкретную роль, должны рассматривать проект со своей «колокольни» и обладать необходимой для этого квалификацией.

Шесть ролей модели проектной группы, подробно описанные в этой главе, связаны с шестью целями проектной группы, что проиллюстрировано в таблице. Все эти цели важны для успеха проекта в целом, и поэтому все роли равноправны. В этой модели нет руководителя всего проекта — есть группа людей, знающих, что нужно делать и делающих это.

Таблица. Цели и роли

Цель	Роль
Удовлетворение требований заказчика	Менеджер продукта
Соблюдение ограничений проекта	Менеджер программы
Соответствие спецификациям	Разработчик
Выпуск только после выявления и устранения проблем	Тестер
Повышение эффективности труда пользователя	Инструктор
Простота развертывания и постоянное сопровождение	Логистик

Как же начать работу над проектом, не зная, сколько времени на это потребуется, сколько проект будет стоить и каких результатов ожидать? Ответить на эти вопросы поможет модель проектной группы MSF, обсуждаемая в следующих разделах этой главы, и модель процесса разработки, описанная в главе 4. Обе эти модели предлагают составлять расписания и графики «снизу — вверх», в проектировании придерживаться подхода «сверху — вниз» и, кроме этого, распределять ответственность за результаты работы между членами группы. На первый взгляд может показаться, что эти рекомендации не отвечают на поставленный вопрос. Однако на самом деле они позволяют сократить риски на ранних стадиях проекта. Приняв на этих стадиях правильное решение, вы избежите серьезных изменений в дальнейшем, что намного сократит время выполнения проекта и затраты на него.

Модель проектной группы

Работа над проектом включает множество разных видов деятельности и изучение требований с разных точек зрения, поэтому распределение основных задач по нескольким ролям повышает шансы на успех проекта. Как видно из рисунка, в MSF определены шесть таких ролей, которые и составляют модель проектной группы. У каждой из ролей свои обязанности, выполнение которых и обуславливает удачу проекта.

Рис. Роли в модели проектной группы

Обычно каждую роль исполняют несколько человек, один из которых считается руководителем, именно он на совещаниях с другими членами группы представляет свое направление.

Внимание! Коллективная ответственность не должна выливаться в коллективную безответственность. Это особенно важно в модели проектной группы, поскольку выполнение каждой ролью всех своих обязанностей — обязательное условие успеха проекта.

В проектную группу должны входить:

- опытные руководители;
- инициативные сотрудники, способные принимать решения и нести ответственность за свое направление работы.

Их задача:

- сконцентрироваться на выпуске продукта;
- выработать общее представление о проекте.

Для эффективной работы проектной группы требуется соблюдение следующих правил в отношениях между людьми:

- **доверие** — делает действия людей согласованными, при этом все
- следуют принципу «мы делаем то, что обещали сделать»;
- **уважение** — люди признают способности других, следуя правилу
- «каждый из нас необходим нашей группе»;
- **согласие** — все должны знать и поддерживать цели проекта и верить в его успех — «мы завершим проект, и точка»;
- **ответственность** — люди должны ясно понимать цели проекта, свои обязанности и чего от них ожидают — «я сделаю свою работу, вы — свою, и к четвергу мы построим наш дом».

Подытожив эти характеристики, мы получаем главный принцип — обязанность каждого — выпустить нужный продукт в нужное время».

Менеджер продукта

Менеджер продукта должен вовремя реагировать на потребности заказчика. Его главная задача — сформировать общее представление о поставленной задаче и о том, как ее решать. Он должен ответить на вопрос «Зачем мы делаем все это?» и убедиться, что все члены группы знают и понимают ответ на него.

Основная цель этой роли — удовлетворение требований заказчика. Для этого менеджер продукта выступает представителем заказчика в группе разработчиков и представителем группы у заказчика. (На этом этапе важно понимать разницу между заказчиком и пользователем: заказчик платит за создание продукта, а пользователи с ним работают.) Кроме того, менеджер продукта вместе с менеджером программы должны прийти к компромиссному решению относительно функциональных возможностей продукта, сроков его разработки и финансирования проекта.

Как представитель заказчика, менеджер продукта отвечает за выполнение требований заказчика, создание бизнес-сценариев, формирование общего представления о проекте у группы и клиента, а также проверяет, удовлетворяют ли решения группы потребностям заказчика.

Как представитель группы, менеджер продукта отвечает за взаимодействие с заказчиком и управление его ожиданиями. При этом он проводит брифинги с клиентом и главными менеджерами, устраивает встречи с пользователями и демонстрации продукта.

Менеджер продукта сообщает группе предварительную дату выпуска продукта, устанавливаемую заказчиком. Однако в действительно успешных проектах графики составляются «снизу — вверх». При этом группа сама определяет возможную дату выхода приложения, которую менеджер продукта сообщает клиенту.

Примечание Заказчик (лицо, финансирующее проект) всегда ожидает от проекта чего-то большего. Естественно, ему хочется, чтобы все его пожелания воплотились в жизнь. И здесь основная роль отводится менеджеру продукта, следящему за ожиданиями клиента. Таким образом, работа менеджера часто определяет успех или провал проекта.

Менеджер продукта должен постоянно общаться с заказчиком, выясняя: «Мы сделали то, что обещали?» Допустим, в рамках проекта предполагалось реализовать пять операций за две недели и 10 000 долларов. Однако за полторы недели было потрачено 7 000 долларов, реализованы только три функции. Как вы думаете, покажется ли этот проект заказчику успешным? Однако квалифицированный менеджер продукта мог бы ответить: «Был проведен второй этап переговоров, и мы договорились с заказчиком относительно самых важных операций, которые можно реализовать, не превышая финансирование и не выходя из графика. То, что мы обещали, мы выполнили. Сейчас же мы начинаем новый проект, в рамках которого реализуем две оставшиеся функции и еще две дополнительные». Таким образом, менеджер продукта пришел с заказчиком к компромиссу, продемонстрировав процесс принятия решений и проинформировав клиента об изменившихся рисках и проблемах. (Обычно трудности возникают при определении затрат на проект, даты его выполнения и функциональных возможностей, а также при выделении ресурсов.) Если все участники проекта смогут сказать: «Да, мы сделали то, что обещали», между заказчиком и проектной группой воцарится доверие и уважение.

Примечание Основная причина неудачи многих проектов — непонимание ожиданий заказчика. При любых — запланированных или нет — изменениях проекта, его ресурсов или даты выхода продукта ожидания заказчика, пользователей и проектной группы должны быть скорректированы.

Группа менеджмента продукта представляет интересы заказчика и помогает ему определить необходимые функции приложения и их приоритеты. Отложить работу над какой-либо функцией вправе только заказчик, а переговоры по этим вопросам ведет менеджер продукта. За выполнение и составление графиков отвечает менеджер программы, поэтому в его власти изъять определенные функциональные возможности, чтобы текущая

версия появилась в срок. Если менеджер программы решит сократить набор функциональных возможностей, чтобы не выбиться из графика, он обязан сообщить об этом менеджеру продукта и заказчику, которые вправе согласиться или нет. В последнем случае заказчик и менеджер продукта должны согласовать изменение даты выпуска продукта. Кроме того, они могут увеличить финансирование с целью расширения состава группы. Затем менеджер программы определяет, как и на что потратить дополнительные деньги и поможет ли это выполнить план. Окончательное решение о финансировании принимают заказчик и менеджер продукта.

Внимание! *Дополнительные ресурсы не всегда ускоряют разработку, так как возрастают издержки, связанные с обменом информацией и управлением.*

Руководителя группы менеджмента часто называют «борцом за продукт». Как правило, это один из топ-менеджеров организации. Остальные члены группы менеджмента продукта должны хорошо разбираться в структуре организации, ее стратегии и бизнес-целях.

Примечание Если проектная группа состоит из консультантов или приглашенных из других организаций людей, менеджер продукта должен учитывать интересы как заказчика, финансирующего проект, так и заказчика, нанявшего консультантов и оплачивающего их услуги. Как правило, консультационная группа требуется для ознакомления организации с MSF и обеспечения взаимодействия заказчика и проектной группы.

Менеджер программы

Задача менеджера программы — вести процесс разработки с учетом всех ограничений. Руководитель этого направления должен понимать разницу между понятиями «руководитель» и «начальник». В своей книге «Dynamics of Software Development» бывший директор отдела программного менеджмента Microsoft Джим Маккарти пишет:

«Запомните, что ваша цель — повысить авторитет каждого члена группы, а не подавить его своим».

Главная обязанность менеджера программы — выполнить все стадии разработки так, чтобы нужный продукт был выпущен в нужное время. Он координирует деятельность других членов группы. И хотя иногда ему придется подгонять своих сотрудников, он не должен и помышлять о диктаторском стиле управления.

Главный менеджер программы составляет график проекта на основе информации, полученной от остальных членов группы. Он координирует этот график с руководителями всех подгрупп. Буферным временем проекта также управляет менеджер программы. Если отдельные части работы выполняются раньше графика или, наоборот, задерживаются, именно менеджер программы должен выяснить, как это скажется на проекте, и изменить график.

Для выполнения своих обязанностей менеджер программы должен отлично разбираться в деловой стороне проекта и иметь ясное представление о технологиях, необходимых для его выполнения. Естественно, что от руководителя отдела программного менеджмента требуется коммуникабельность и талант организатора.

Программный менеджер компании, где только начинают применять MSF, должен полностью понимать все модели и процессы, повышающие эффективность труда членов группы. И хотя умение руководить нужно всем ролям, особенно это качество важно для менеджера программы. Его авторитет должен стать непререкаемым еще до начала проекта у всех его участников, включая бизнес-отделы и руководство организации. Как правило, группа менеджмента программы управляет ресурсами, используемыми другими ролями, а также составляет и координирует расписания совещаний.

Так как менеджер программы отвечает за набор функциональных возможностей приложения, одна из его обязанностей — определить их набор, необходимый для выполнения требований заказчика. Критичные для проекта требования выявляет менеджер продукта, но набор реализующих их функций определяет менеджер программы. Кроме

того, менеджер программы дает заказчику рекомендации, касающиеся дальнейшего развития продукта. Отдел программного менеджмента отвечает за функциональные возможности, изложенные в спецификациях (здесь описано, что будет создано) и в главном плане проекта (определяет, как это будет сделано). Он также координирует работу над этими документами. Тем не менее каждый участник проекта вносит свой вклад в функциональные спецификации и в план проекта.

Внимание! Важно, чтобы менеджер программы понимал, что каждый член группы разбирается в своих обязанностях намного лучше его.

Менеджер программы должен полностью положиться на опыт остальных сотрудников и только следите за соблюдением всех условий и ограничений.

Менеджер программы отвечает и за бюджет проекта, объединяя требования к ресурсам всех членов группы в единый план расходов. Естественно, его задача — не только разобраться в этих требованиях, но и контролировать реальные затраты, сравнивая их с запланированными. Кроме того, менеджер программы должен регулярно сообщать о состоянии работы всем основным участникам проекта. Ведь одним из симптомов неудачного продукта является отсутствие информации о состоянии проекта, пока деньги на него не кончились. Важно внимать, что решение о дополнительном финансировании может требовать изменения других сторон проекта, а следовательно, его будет принимать менеджер программы совместно с заказчиком.

Разработчик

Разработчики знакомят остальных членов группы с применяемыми технологиями и собственно создают продукт. В качестве консультантов они предоставляют исходные данные для проектирования, проводят оценку технологий, а также разрабатывают прототипы и тестовые системы, необходимые для проверки решений и сокращения рисков на ранних стадиях процесса разработки. Чтобы создать продукт определенного качества, разработчикам не следует замыкаться на создании кода, они должны участвовать и в решении прикладной задачи. Они творят не ради творчества, а для реализации требований заказчика. Часто, чтобы полностью разобраться в проекте, приходится создавать прототипы, а чтобы протестировать новую технологию, — испытательные системы, помогающие принять окончательное решение относительно архитектуры приложения. Этим также занимаются разработчики.

Как программисты разработчики отвечают за низкоуровневое проектирование и оценку затрат на реализацию продукта. В большинстве организаций несколько основных разработчиков занимаются и архитектурой приложения. Как правило, это требуется на ранних стадиях проекта, когда уточняются детали функциональных спецификаций и описывается взаимодействие продукта с внешними системами.

Разработчики сами оценивают сроки своей работы. Такая концепция MSF — создание графиков ответственными за выполнение конкретного участка членами группы — называется *составлением расписания «снизу — вверх».* Она позволяет выпустить нужный продукт в нужное время за счет уточнения графиков и повышения ответственности за выполнение работы в запланированные сроки.

Разработчики отвечают и за техническую реализацию проекта — в основном на фазах создания логической и физической модели, обсуждавшейся в главе 2. На этих стадиях их задача — определить методы реализации функциональных возможностей и заданной архитектуры, а также оценить сроки выполнения этой работы. Заметим, что разработчики не выбирают функции — они только решают, как их реализовать.

Кроме того, на стадии «Планирование» разработчики решают, какое влияние окажет на проект добавление или удаление некоторых функций. Разработчики не участвуют в заключительной стадии проекта — развертывании продукта, однако они должны тесно сотрудничать с логистиками на стадии установки приложения.

Внимание! Руководителю группы разработки не рекомендуется совмещать несколько ролей. Будучи программистом, он должен делать то, что у него лучше всего получается — писать качественный код.

Тестер

Задача тестеров — испытание продукта в реальных условиях, дабы определить, что в продукте работает и что не работает, и нарисовать таким образом точный «портрет» приложения. Естественно, для проведения тестов нужно отлично разбираться и в требованиях пользователей, и в том, как их удовлетворить.

Тестеры разрабатывают стратегию, планы, графики и сценарии тестирования, которые позволяют убедиться, что все ошибки выявлены и исправлены до выпуска приложения. Ошибкой называют любую проблему, из-за которой продукт не выполняет свои функции. Ею может оказаться и ошибка в коде, называемая «жучком», и отклонение от спецификаций, заданных менеджером программы, и недоработки в документации, подготовленной группой обучения пользователей.

Примечание Важно различать тестирование и общую оценку качества, (Total Quality Assurance, TQA). Тестирование касается только проекта — точнее, его технической стороны. Проверку качества организует руководитель, ответственный за качество, — он выясняет соответствие продукта корпоративным, правительственным и другим стандартам.

Нельзя совмещать должности тестера и разработчика. Разделение этих обязанностей:

- гарантирует независимую проверку того, что продукт действительно выполняет все требования;
- повышает качество продукта за счет конкуренции между группами.

Хотя проверяют качества продукта только тестеры, за выпуск хорошего продукта отвечают все члены проектной группы.

Внимание! За качество кода отвечают разработчики. Отделение тестирования от разработки не снимает с них эту обязанность.

Контроль изменений

При работе над проектом необходимо контролировать изменения, им должны заниматься все участники группы, но чаще всего в полном объеме этим приходится заниматься именно группе тестирования. Разработчики уже не один год применяют системы, подобные Microsoft Visual SourceSafe, но они осуществляют только контроль версий. Такие системы пригодны для отслеживания версий любых документов, функциональных спецификаций, исходного и скомпилированного кода, но они только частично регистрируют изменения, в качестве примера можно привести сохранение версий документа в Microsoft Word — эта функция не гарантирует, что вы внесете верные правки и представите в качестве результата правильную версию. Таким образом, для управления изменениями необходимо:

- создать эталонный документ;
- определить изменяемые элементы;
- определить влияние изменений на существующие системы, •процессы или документы;
- определить метод реализации изменений;
- назначить человека, который внесет изменения;
- определить влияние изменения на условия выполнения проекта, •его бюджет, график и политику;
- получить одобрение изменений (скажем, у руководителя проекта);
- внести изменения;
- сообщать новый документ, в котором изменение учтено.

Внимание! *Изменения лучше анализировать и принимать порциями, иначе на это уходит слишком много времени.*

Прочие обязанности

Некоторые важные обязанности тестеров часто упускают из виду. К ним относятся:

· **уведомление об ошибках и их отслеживание** — тестовая группа отвечает не только за управление изменениями, но и за систему вывления ошибок и информирования о них;

· **сборка продукта** — в группе должен быть человек, ответственный за сборку (компиляцию) продукта, и часто такой «главный сборщик» является тестером, он может использовать только код, хранящийся в системе управления версиями; эту рутинную работе
удается автоматизировать с помощью сценариев, однако необходимо проверять правильность сборки;

· **выявление и контроль рисков** — это обязанность всех членов группы, менеджер программы должен разработать метод контроля - например, с помощью электронных таблиц Microsoft Excel; тестеры отвечают за работу программы в реальных условиях, поэтому ил задача — представить группе анализ рисков с этой точки зрения.

Внимание! План контроля рисков, созданный или пересмотренный за пять минут до начала совещания, совершенно бесполезен. Процесс контроля рисков должен быть непрерывным и постоянным — только тогда гарантируется качество продукта и соблюдение сроков выпуска

Инструктор

Цель группы обучения — повысить эффективность труда пользователей. Поэтому инструкторы «принимают сторону» пользователей подобно тому, как менеджеры продукта представляют интересы заказчика. Однако перед пользователями инструкторы выступают в роли представителей проектной группы.

В этом последнем качестве группа обучения отвечает за выпуск удобного, полезного продукта, которому практически не нужна поддержка. Персонал группы тестирует удобство использования продукт та, выявляет проблемы в этой области и проверяет проект пользовательского интерфейса.

Активно участвуя в создании пользовательского интерфейса, инструкторы сокращают затраты на сопровождение продукта и поддержку пользователей. Часто же бывает, что изучению этих расходов практически не уделяется внимания, хотя все расчеты очень просты; чем легче работать с приложением, тем меньше затраты на поддержку.

Внимание! Не попадайтесь в ловушку: не завышайте планируемый прирост эффективности труда пользователей — это приводит к переоценке прибыли от инвестиций в проект.

Довольно часто приложение сдается в эксплуатацию без плана обучения, а проектные группы даже не имеют представления о том как пользователи будут изучать приложение. Задача инструкторов - не допустить этого, заранее спроектировав, составив и протестировав все необходимые материалы, включая краткие памятки, руководства пользователей, системы, онлайновой помощи, Web-страницы и даже — если понадобится — целый учебный курс. Когда материалы подготовлены, группа координирует обучение пользователей.

Управлять ожиданиями пользователей так же важно, как и ожиданиями заказчика. Однако не следует думать, что пользователи будут разбираться в функциональных спецификациях. Они более благосклонно отнесутся к прототипам системы, демонстрация которых значительно повысит шансы на успех проекта. Кроме того, пользователям полезно показать действующие модули программы. Хотя маркетинг входит в обязанности менеджеров продукта, также важно вовремя информировать и пользователей. Для этого

следует периодически рассылать электронные сообщения о новых функциях программы и ее бета-версиях.

Примечание. Роль обучения важна и в случае разработки приложения для другой организации. Ведь при этом вам неизвестно, насколько хорошо ее сотрудники информируют пользователей. Так же, как отдел менеджмента продукта управляет ожиданиями заказчика, отдел обучения должен управлять ожиданиями пользователей. Настоящий успех дает продукт, если пользователи узнают о его возможностях не сразу, а постепенно. При этом важны обучение и подготовка.

Логистик

Логистик представляет интересы служб поддержки и сопровождения, справочных служб и других служб канала доставки. Он занимается развертыванием продукта и его сопровождением и контролирует продукт с этой точки зрения в процессе проектирования. Кроме того, его задача — составление графиков развертывания приложения. Логистики, менеджеры продукта и менеджеры программы совместно определяют порядок передачи продукта пользователям и организации, после чего логистики готовят их к развертыванию приложения.

Логистик, участвующий в крупном проекте, должен обладать опытом развертывания крупномасштабных приложений на нескольких сотнях компьютеров. Именно поэтому от него требуется коммуникабельность и хорошая техническая подготовка. Логистик руководит всеми сотрудниками, устанавливающими и настраивающими пользовательские системы. Кроме того, он должен уметь координировать установку программного обеспечения и оценивать ее результаты.

Логистик обязан разбираться в инфраструктуре продукта и требованиях к его сопровождению. Его задача — проверить, чтобы все серверы развертывания и рабочие станции пользователей удовлетворяли требованиям. Обычно данные вопросы учитываются в планах выпуска, развертывания и сопровождения продукта. Заметим, что развертывание приложений значительно упрощается при использовании таких средств Microsoft Windows 2000, как Active Directory и System Management Server.

Хотя основная задача логистика заключается в плавном развертывании продукта, обязанность руководителя этого направления — составить план сопровождения и эксплуатации и убедиться, что существующие группы способны с этим справиться. Важно обучить не только пользователей, но и персонал справочной службы. Причем последних надо начать знакомить с продуктом еще на ранних стадиях разработки — скажем, на этапе бета-тестирования.

Перед сдачей приложения в эксплуатацию следует составить документацию, определить требования к резервному копированию данных и разработать план восстановления на случай отказа систем. После развертывания логистики в течение некоторого времени консультируют группу сопровождения. Конечно, сложно предсказать все трудности, которые могут возникнуть в процессе эксплуатации приложения, поэтому во всех планах следует предусматривать и порядок действий в экстренном случае.

Помимо перечисленных выше обязанностей, логистик занимается сопровождением промежуточных версий продукта в процессе разработки. Его знания требуются и при изучении поведения приложения в тестовой среде. Кроме того, помощь логистиков необходима при создании тестовых, сертификационных и производственных систем. Они должны также сопровождать приложение в процессе моделирования эксплуатации на этапе бета-тестирования.

Многие организации используют системы мониторинга производительности и стабильности. Поэтому логистики обязательно должны проинформировать разработчиков о наличии таких средств. Например, интеграция средств управления продуктом в среду

административной консоли Microsoft Management Console (MMC) упростит интеграцию продукта с другими системами Microsoft.

Размер группы логистики определяется графиком развертывания, уровнем автоматизации установки, наличием средств автоматического развертывания приложений и другими характеристиками этого процесса.

Размеры группы и масштаб проекта

В проектной группе за каждое направление должен отвечать как минимум один человек. При реализации крупного проекта возникает затруднение, связанное с эффективным обменом информацией. В небольших организациях или при работе над мелкими проектами роли можно совмещать. Однако в этом случае существует другая проблема — как «усидеть на нескольких стульях» одновременно, не упустив из виду ни одной существенной детали проекта с точки зрения каждой роли.

Крупные проекты

В своей книге «Rapid Development: Taming Wild Software Schedules» бывший сотрудник Microsoft Стив Макконнелл пишет: *«Для реализации крупного проекта необходимо, чтобы в организации был наработан опыт формализованного и непрерывного обмена информацией. ...А это возможно при наличии иерархической структуры, то есть небольших групп, в каждой из которых есть сотрудник, отвечающий за взаимодействие с другими группами и менеджерами».*

Таким образом, чтобы справиться с крупным проектом, приходится и делить проектную группу на тематические и функциональные подгруппы.

Тематические группы

Это небольшие подгруппы из одного или нескольких человек, роли которых различны. Каждой из таких групп выделяется некий набор функциональных возможностей приложения, за все стороны проектирования и разработки которого она и отвечает (включая составление проекта и графика реализации). Например, какой-либо группе нужно предоставить решать задачу вывода данных на печать.

По этому поводу Стив Маконнелл замечает: *«Достоинства тематических групп — эффективность, сбалансированность и ответственность. Возможности каждой такой группы велики, ведь в ее состав входят представители... всех заинтересованных сторон. Ее члены принимают решения, учитывая все возможные мнения, а, следовательно, нет никаких оснований их изменять.*

Но этой же причине в такой группе высока ответственность. Ее члены могут обратиться ко всем людям, опыт которых необходим в их работе. Поэтому, если они так и не найдут оптимального решения, в этом они смогут винить только себя. Такая группа сбалансирована. Вряд ли вы захотите, чтобы окончательные спецификации создавал *только отдел разработки, маркетинга или контроля качества. Только решение, принятое группой представителей каждого из этих отделов, будет по-настоящему сбалансировано».*

Функциональные группы

Функциональные группы формируются в рамках одной роли. Они нужны в очень крупных проектных группах или при работе над крупномасштабными проектами, когда отдельные роли нуждаются в дополнительном подразделении. Например, в Microsoft отдел менеджмента продукта обычно состоит из групп планирования и маркетинга. Обе они занимаются менеджментом продукта, но первая отвечает за определение действительно необходимых заказчику функций приложения, а вторая — за информирование потенциальных клиентов о достоинствах продукта.

Приведем еще один пример. Иногда группу разработчиков требуется разделить на три подгруппы, каждая из которых занимается одним уровнем архитектуры (пользовательским, прикладным или уровнем данных). Довольно часто в крупных отделах

разработки группу программистов подразделяют на группу решений и компонентную группу. Первые создают собственно приложение, «склеивая» вместе его компоненты, разработанные вторыми. Кстати, это разделение часто оказывается и разделением по языкам программирования: разработчики решений, как правило, работают с языками, позволяющими быстро собирать приложения из готовых компонентов, тогда как выбор языка для разработки повторно используемых компонентов, пригодных для многих проектов, определяется прежде всего соображениями эффективности.

Небольшие проекты

Хотя в модели группы разработчиков предусмотрено шесть направлений деятельности, необязательно включать в проектную групп шесть человек. Другими словами, некоторые должности можно совмещать. Конечно, основной смысл такого разделения в том, чтобы каждую из шести задач решал один из членов группы Однако не во всех проектах это возможно.

В небольших группах один человек может играть несколько ролей. При этом мы рекомендуем соблюдать следующие принципы разделения должностей.

· **Нельзя совмещать разработку с другими видами деятельности —** ее создателей приложения не стоит отвлекать от основной задачи. Если «повесить» на разработчиков дополнительные обязанности, то скорее всего график работ будет нарушен, а дату выпуска продукта придется отодвинуть.

· **Конфликт интересов** — нельзя совмещать роли, интересы которых противоположны. Пример — менеджер продукта и менеджер программы. Первый хочет выполнить все требования заказчика, второму же надо уложиться в график и бюджет. Если совместить эти роли, возникает опасность упустить просьбу заказчика о внесении изменений в проект либо, напротив, принять их без должного анализа влияния на график работ. Таким образом, назначение на эти роли разных людей позволяет соблюсти интересы всех участников проекта.

На рис. показаны комбинации ролей, оказывающие позитивное и негативное влияние на проект. Роли, отмеченные буквой «З» — Запрещено— нельзя совмещать из-за конфликтов интересов. Вероятность совмещения ролей, отмеченных буквой «Н» — Нежелательно — и из-за сильного различия в необходимой квалификации Например, знания и опыт менеджера продукта и логистика сильно отличаются. Сочетания ролей, помеченные буквой «Д» — Допустимо — возможны, так как их интересы совпадают. Например, как тестеры, так и инструкторы отвечают за выполнение требований пользователей.

Естественно, успех совмещения ролей зависит от конкретных членов группы, их навыков и опыта. В некоторых проектных группах возможно успешное совмещение ролей, комбинация которых согласно нашей таблице опасна. Главное — помнить цели каждого направления и контролировать совмещение должностей в соответствии с ними, предотвращая конфликты. В противном случае некоторые обязанности какой-то роли будут не выполнены, что увеличивает число неконтролируемых рисков.

	Менеджер продукта	Менеджер программы	Разра-ботка	Тестиро-вание	Инструктор	Логис-тика
Менеджер продукта		**З**	**З**	**Д**	**Д**	**Н**
Менеджер программы	**З**		**З**	**Н**	**Н**	**Д**
Разработка	**З**	**З**		**З**	**З**	**З**
Тестирование	**Д**	**Н**	**З**		**Д**	**Д**
Инструктор	**Д**	**Н**	**З**	**Д**		**Н**
Логистика	**Н**	**Д**	**З**	**Д**	**Н**	

Д - Допустимо **Н** - Нежелательно **З** - Запрещено

Рис. Деструктивные и созидательные сочетания ролей

Создание группы

Модель проектной группы описывает структуру группы для работы над проектом создания приложений масштаба предприятия. Однако одной структуры недостаточно — важным фактором успеха является квалификация членов группы. В этой главе мы опишем методы проверки их квалификации.

Поиск руководителей

Главная задача человека, ответственного за создание проектной группы, — подобрать квалифицированных исполнителей. Эта кажущаяся простой (но на самом деле сложная) задача имеет огромное значение для успеха всего проекта.

Найти лидеров — несложная проблема; в любой организации они всем известны. Важно понимать, что мы говорим именно о лидерах, а не начальниках. Конечно, в любой организации есть менеджеры директора и так далее, но положение в иерархической структуре далеко не всегда гарантирует наличие качеств лидера. Лидеров определяют действия и качества, а не должности.

Зачем нужны именно лидеры? Потому что исполнителей и так избытке. Важно понимать, что деление сотрудников на лидеров и исполнителей не умаляет деловых качеств и квалификации последних. Чтобы лидер добился удачи, он должен набрать отличных исполнителей. Однако между лидерами и подчиненными должно быть взаимопонимание. Группа обсуждает, что нужно делать, а затем выполняет принятое решение, поэтому все, и руководители, и подчиненные, одинаково важны для проекта — в отсутствие кого-либо и них успех проекта невозможен.

Руководители должны обладать:

- умением понимать и помогать;
- коммуникабельностью;
- авторитетом внутри организации и за ее пределами;
- чувством ответственности за поставленные цели;
- умением принимать конструктивные решения;
- уверенностью в своих силах;
- достаточной для решения поставленных задач квалификацией;
- способностью помочь другим развить свои таланты и приобрести опыт.

Многие прочтут этот список и скажут: «Я обладаю всеми этим качествами». Еще больше людей подумают: «Я могу обладать этим качествами». Однако настоящего лидера отличают именно эти *способности, а* не *намерения.* Это относится и к качествам руководителя: нужно оценивать оказываемое им влияние, а не его намерения. Ведь давно известно, что «реальность — это то, что видят другие».

Вот какие отношения друг с другом и к работе возникают в такой группе:

- заинтересованность;
- надежда, оптимизм, готовность к работе;
- определение задач и решений;
- проявление взаимопомощи;
- доверительные уважительные отношения;
- единение.

Последнее очень важно: эффективность работы группы при этом выше всего.

Для успеха проекта недостаточно только распределить роли и обязанности. Помимо структуры, следует придерживаться определенных принципов и методов. Ниже обсуждаются «лучшие методы и принципы», которые успешно применялись не только внутри Microsoft, но также партнерами и заказчиками компании.

Общее представление о проекте

Важнейший фактор для успеха проекта — единое понимание целей и задач проекта всеми участниками. Каждый из них изначально имеет **свое** мнение, касающееся приложения. В процессе формирования общего представления о проекте все эти мнения обсуждаются, что позволяет добиться единства целей всех членов проектной группы и заказчика.

На основе выработанного представления о проекте создается документ «Концепция проекта», который:

- описывает не только то, что делает продукт, но и то, чего он не делает;
- конкретизирует продукт (например, позволяет включать и исключать определенные функциональные возможности из данной версии); •побуждает группу достичь сформулированной цели;
- содержит описание путей реализации проекта, благодаря чему проектная группа и заказчик могут начать работу.

Примечание Создание концепции не гарантирует согласия с ней всех участников проекта (разработчиков, заказчика и пользователей). Поэтому необходимо проследить, чтобы все они тщательно изучили ее и полностью в ней разобрались.

Выработка согласованного мнения о проекте позволяет избежать разногласий между участниками проекта, мешающих достичь поставленных целей и разъединяющих группу. Оптимальный способ согласования концепции — обсуждение целей и задач проекта. Такая дискуссия дает возможность добиться взаимопонимания всех членов группы. При этом их обязательства будут не простым согласием: «Да, мы будем работать над проектом», в них появится мотивация: «Давайте приступим к работе и сделаем что-то действительно новое». Дискуссию на ранних стадиях проекта, как правило, организуют ее сотрудники отделов менеджмента продукта и менеджмента программы, имеющие опыт маркетинговых исследований.

Внимание! Не обольщайтесь тем, что действия проектной группы и их цель понятны заказчику. Подробно обсуждайте проект с заказчиком на всех стадиях его выполнения.

Группа равных

В группе равных важна каждая роль. Такой подход, и только он, делает возможным неограниченный обмен информацией между членами группы, повышает ответственность за выполнение работы и усиливает понимание того, что все шесть целей проекта одинаково важны. Естественно, в таких группах надо проводить проверку качества продукта; это делает представитель заказчика, разбирающийся в решаемой задаче.

Подчеркнем, что равенство — это не анархия. Равенство существует лишь в отношении ролей; в рамках каждой роли следует придерживаться обычной иерархической модели. В каждой группе необходима должностная иерархия и наличие руководителя,

ответственного за управление и координирование работы своего направления. Задача остальных членов группы, исполняющих ту же роль, — выполнять поставленные перед ними задачи.

Ориентация на продукт

Ориентация на продукт — это не призыв работать над коммерческим программным обеспечением одним способом, а над приложениями для внутреннего пользования — иначе. Это требование — относиться к результату работы, как к продукту.

Прежде всего нужно выяснить, является проект самостоятельным или частью более крупного проекта. MSF рекомендует идентифицировать проекты — это позволяет людям чувствовать себя членами команды. Скажем, в Microsoft принята практика присвоения проектам кодовых имен (например «Чикаго» для Windows 95 и «Мемфис» для Windows 98). Кодовые имена четко идентифицируют проект и работающую над ним группу, позволяют людям четче ощущать причастность к проекту и ответственность за него. Создать и усилить значимость группы, поднять ее боевой дух можно разными способами — скажем, помещая название проекта на футболки, кружки и прочие сувениры.

Сформированное отношение к продукту позволяет сконцентрироваться на результате проекта, а не на процессе его достижения вовсе не означает, что процесс не важен для группы. Мы просто иначе расставляем приоритеты — процесс должен служить достижению цели, а сам по себе он не имеет ценности. Увлеченность процессом не должна стоять на пути к результату. Необходимо прежде всего, чтобы все, чувствовали ответственность за выпуск продукта.

Бывший менеджер программ Microsoft Крис Питере описал отношение к продукту следующим образом:

«*У нас всех... одна и та же работа — выпуск продукта. Ваша задача — не писать код, не тестировать его и не создавать спецификации. Ваша задача — выпустить продукт. Именно это делает группа разработки продукта.*

Ваша роль разработчика или тестера — вторична. Я не хочу сказать, что она не имеет значения — конечно, она важна, — но она вторична по отношению к вашей основной задаче — выпуску продукта.

Когда вы просыпаетесь утром и приходите на работу, вы спрашиваете себя: «Что мы делаем, пишем код или делаем продукт?» Ответ очевиден — мы делаем продукт. Вы не должны программировать, вы обязаны не программировать, и это не каламбур».

Ориентация на отсутствие дефектов

Ориентация на отсутствие дефектов — это требование соблюдения качества продукта. Оно вовсе не подразумевает, что продукт должен быть выпущен без дефектов; достаточно, чтобы дефектов было не больше нормы, установленной группой на ранних стадиях разработки. Кроме того, это означает, что группе придется постоянно контролировать качество на всех стадиях: если завтра потребуется выпустить готовый продукт, группа должна быть готова выполнить это обязательно. Таким образом, с точки зрения качества продукт всегда должен быть практически готов.

В успешных группах все отвечают за качество продукта — эту обязанность нельзя переложить на других. В этом смысле все члены группы являются представителями заказчика.

Понимание целей бизнеса

Чтобы выпустить успешный продукт, группе недостаточно разбираться только в технической стороне проекта. Создано множество технически совершенных и безупречно написанных продуктов, не способных выполнять задачи, поставленные заказчиком. Чтобы избежать такой ситуации, члены группы должны ясно представлять себе решаемую задачу.

И здесь необходимо активное участие заказчика в процессе разработки. Его обязательно нужно привлекать к созданию концепции проекта, к принятию решений по продукту и его использованию и к анализу промежуточных версий.

Ответственность в равной мере

Крис Питерс однажды не без юмора заметил:

«*Чрезвычайно важно, чтобы ответственность разделяли сотрудники как можно более низких уровней. Ваша цель не в том, чтобы лишиться сна, переживая из-за проекта, которым вы руководите.*

Ваша цель не в том, чтобы лишить сна руководителей направлений.

Не спать по ночам от переживаний за проект должны абсолютно все. Вот когда вы этого добьетесь, считайте, что распределили ответственность должным образом».

Чтобы добиться согласованной работы всех членов группы, вы должны сделать их роли взаимозависимыми и перекрывающимися, они все в одинаковой мере должны разделять ответственность за выпуск нужного продукта в запланированные сроки. Таким образом разрушается узкая специализация членов группы, которая ведет к изолированности, а не к совместной работе.

Совместное проектирование

Джим Маккарти так описал концепцию общего участия в проектировании в своей книге «Dynamics of Software Development»:

«Цель проектирования любого продукта — собрать лучшие идеи.

Поэтому в проектировании должны участвовать все члены группы.»

В создании функциональных спецификаций должны участвовать представители всех направлений, так как у каждого из них свое представление о проекте. Они должны убедиться, что в проекте учтены не только требования всей группы, но и требования каждой роли.

Обучение на опыте других проектов

Чтобы не полагаться на удачу, стоит воспользоваться опытом работы над другими проектами, как успешными, так и провалившимися. Изучение опыта — основа совершенствования. Один из методов структурирования и накопления опыта — подведение итогов каждого этапа, зафиксированное в обзоре. Обзор, основа которого — сопоставление планов с результатами, помогает скорректировать дальнейший ход работы и избежать ошибок в дальнейшем. Кроме того, при этом можно выявить хорошо зарекомендовавшие себя методы, чтобы применять их в будущем.

Примечание. Очень важно делиться информацией о методах работы с другими группами разработчиков своей организации.

Подытоживая рекомендации по подбору штата, предложенные Барри Бёмом в книге «Software Engineering Economics», мы рекомендуем следующие правила:

- используйте меньше людей, но лучших в своей области;
- подбирайте задачи в соответствии с квалификацией и мотивацией людей;
- помогайте людям набирать опыт и знания;
- подбирайте людей, дополняющих друг друга;
- не бойтесь отсекать все, что не подходит.

Обучение группы

Работа группы эффективна, если все ее члены понимают, что они делают. Однако во многих случаях это простое правило реализуется недостаточно полно. Выпустить же нужный продукт в срок удастся только при наличии формального и неформального планов обучения проектной группы.

Изучение методологии

Разработка программного обеспечения — это не только создание кода. Поэтому изучение методологии разработки руководителями групп и основными участниками проекта сократит затраты и время выполнения проекта.

Целью любого проекта должно стать улучшение обмена информацией. Для этого в MSF и других моделях разработки применяется общий язык передачи информации о состоянии продукта. Основные элементы этого процесса существуют не один год, и многие

разработчики применяют их на практике. Используя универсальный язык MSF, опытные разработчики смогут поделиться своим опытом с другими членами группы.

Изучение технологий

Создать хороший продукт очень сложно. Поэтому группам разработчиков, логистиков и, по возможности, менеджеров программы надо изучать существующие технологии. Каждый день появляются новые формы обучения современнейшим технологиям, однако для успешной реализации таких технологий недостаточно только знания их основ, необходимо понимать способы их применения.

Вот какие технологии необходимы для работы над проектом по созданию *системы управления ресурсами* (Resource Management System, RMS), о которой идет речь в практикумах этой книги:

- HTML;
- Dynamic HTML;
- Microsoft Active Server Pages (ASP);
- Microsoft Visual Basic Scripting Edition (VBScript); •Microsoft Internet Information Server (US);
- Microsoft Windows NT 4.0;
- Microsoft Windows 2000;
- Microsoft Systems Management Server 2.x (SMS);
- Microsoft Outlook 2000;
- Microsoft Visual InterDev;
- Microsoft Visual Basic 6.0 (VB);
- Microsoft Visual C++ 6.x (C++), библиотеки ATL и STL;
- Microsoft Transaction Server 2.0 (MTS);
- Microsoft SQL Server 7.x;
- создание COM-объектов с помощью VB и C++;
- ActiveX Data Objects (ADO);
- Collaborative Data Objects (CDO).

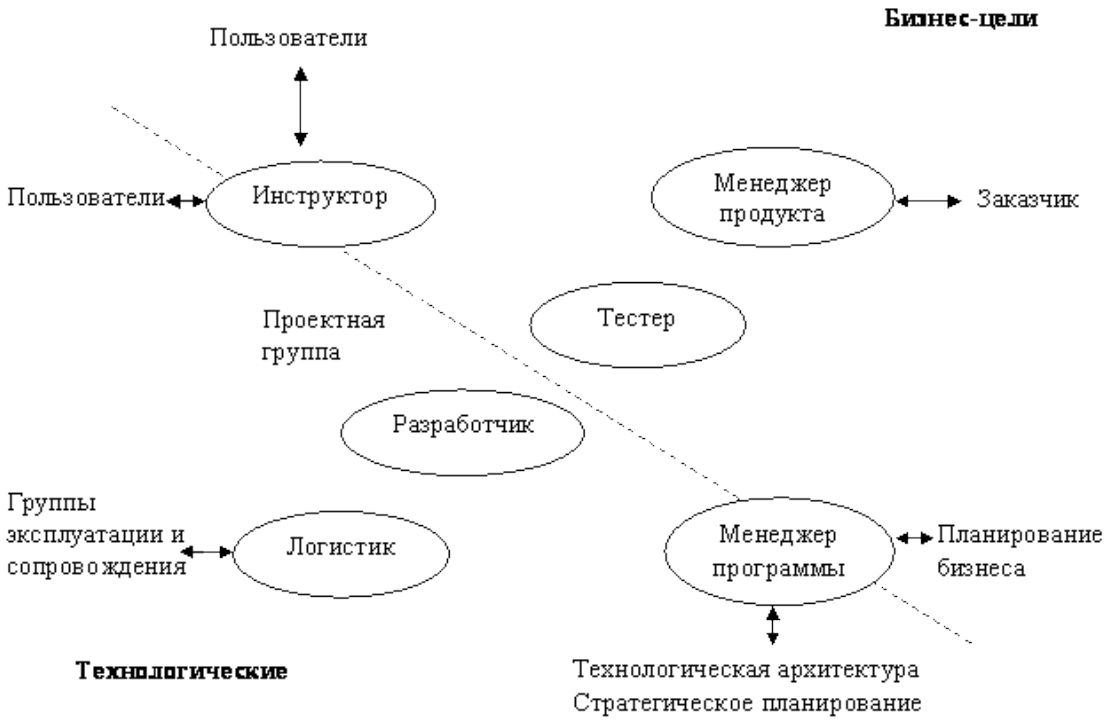
Часто при обучении упускают из вида сопровождение приложения в существующей информационной инфраструктуре. Однако успех проекта зависит не только от качества созданного кода. Проектная группа должна исследовать взаимное влияние инфраструктуры и приложения друг на друга. Как скажется на приложении появление новых операционных систем, новых версий существующих операционных систем или новых базовых приложений, используемых продуктом?

Кроме того, проектной группе придется учесть то обстоятельство, что во время работы над проектом могут появиться новые технологии или новые версии приложений. Нужно уметь оценивать влияние новых функциональных возможностей таких продуктов на жизненный цикл проекта. Поэтому время на обучение следует включать в график проекта еще до определения даты выпуска приложения.

Координация работы с внешними группами

Проектная группа, рассчитывающая на успех, должна взаимодействовать с внешними группами — как с заказчиком и пользователями, так и с другими разработчиками. Этим занимаются менеджер программы, менеджер продукта, инструктор и логистик. В обязанности этих ролей входят как внутренние, так и внешние контакты, в то время как разработчики и тестеры изолированы от общения с внешним миром. (Такая изоляция приводит к повышению эффективности труда этих двух групп.)

Важно, чтобы взаимодействие с внешними группами было явным и понятным. На рис. 3.3 проиллюстрирована координация взаимодействий, связанных как с бизнесом, так и с технологиями. Проектным группам, как правило, приходится взаимодействовать с большим количеством внешних организаций, включая финансовые и юридические подразделения и отделы контроля качества продукта.



Коллективное владение кодом

Коллективное владение кодом позволяет каждому разработчику выдвигать новые идеи в любой части проекта, изменять любую строку программы, добавлять функциональность, фиксировать ошибку и проводить реорганизацию. Один человек просто не в состоянии удержать в голове проект нетривиальной системы. Благодаря коллективному владению кодом снижается риск принятия неверного решения (главным разработчиком) и устраняется нежелательная зависимость проекта от одного человека.

Работа начинается с создания тестов модуля, она должна предшествовать программированию модуля. Тесты необходимо помещать в библиотеку кодов вместе с кодом, который они тестируют. Тесты делают возможным коллективное создание кода и защищают код от неожиданных изменений. В случае обнаружения ошибки также создается тест, чтобы предотвратить её повторное появление.

Кроме тестов модулей, создаются тесты приемки, они основываются на пользовательских историях. Эти тесты испытывают систему как «черный ящик» и ориентированы на требуемое поведение системы.

На основе результатов тестирования разработчики включают в очередную итерацию работу над ошибками.

Все коды в проекте создаются парами программистов, работающими за одним компьютером. Парное программирование приводит к повышению качества без дополнительных затрат времени. А это, в свою очередь, уменьшает расходы на будущее сопровождение программной системы.

Оптимальный вариант для парной работы – одновременно сидеть за компьютером, передавая друг другу клавиатуру и мышь. Пока один человек набирает текст и думает (тактически) о создаваемом методе, второй думает (стратегически) о размещении метода в классе.

Во время очередной итерации всех сотрудников перемещают на новые участки работы. Такие перемещения помогают устранить изоляцию знаний и «узкие места». Особенно полезна смена одного из разработчиков при парном программировании.

Замечено, что программисты очень консервативны. Они продолжают использовать код, который трудно сопровождать, только потому, что он все еще работает. Боязнь модификации кода у них в крови. Это приводит к предельному понижению эффективности систем. В XP считают, что код нужно постоянно обновлять – удалять лишние части, убирать ненужную функциональность. Этот процесс называется реорганизацией кода. Поощряется безжалостная реорганизация, сохраняющая простоту проектных решений. Реорганизация поддерживает прозрачность и целостность кода, обеспечивает легкое понимание, исправление и расширение. На реорганизацию уходит значительно меньше времени, чем на сопровождение устаревшего кода. Увы, нет ничего вечного – когда-то отличный модуль теперь может быть совершенно не нужен.

И еще одна составляющая коллективного владения кодом – непрерывная интеграция.

Без последовательной и частой интеграции результатов в систему разработчики не могут быть уверены в правильности своих действий. Кроме того, трудно вовремя оценить качество выполненных фрагментов проекта и внести необходимые коррективы.

По возможности XP-разработчики должны интегрировать и публично отображать, демонстрировать код каждые несколько часов. Интеграция позволяет объединить усилия отдельных пар и стимулирует повторное использование кода.

6.2. Экономические аспекты создания программных средств

Оценка затрат на разработку ПО является одним из наиболее важных видов деятельности в процессе создания проекта. При отсутствии оценки программного обеспечения невозможно правильно спланировать и управлять проектом. Раньше, когда стоимость программного обеспечения составляла незначительную часть в общей стоимости компьютерной системы, ошибки в ее оценке не оказывали большого влияния на планирование общей затраты на разработку. В данный момент ПО являются самым дорогим элементов системы.

Ошибки в подсчете затрат на проектировании изделия существенно влияют на бюджет проекта. Стоимость проекта, время и ресурсы, требуемые для создания программного обеспечения важные вещи и к их оценке нужно подойти основательно. Сжатые сроки разработки ведут за собой нервную обстановку в команде разработчиков, увеличение количества ошибок. В случаи нарушения графика работы к разработчикам теряется доверие. Если же необоснованно увеличен время на разработку то это добра не принесет, не будет ни какой экономии. Во всем важна золотая середина.

Оценка трудозатрат на разработку программного продукта определяется производительностью труда группы разработчиков, на которую влияет следующая совокупность факторов:

1. Человеческий фактор, связан с опытом и знаниями компании которая занимается разработкой ПО;

2. Ресурсные факторы, характеризующие наличие ресурсов для разработки программных продуктов;

3. Проблемный фактор, определяемый сложностью проблемы которая должна быть решена;

4. Факторы технологий разработки, которые могут быть охарактеризованы используемые методами анализа и проектирования, имеющимися средствами CASE и средствами контроля.

Данные факторы оказывают значительное влияние на производительность труда разработчика. Наибольшее воздействие оказывают факторы программной продукции. При изменении производительности могут достигать 150%, а при изменении за счет ресурсных факторов не превышают 50%. Длительность проекта не зависит линейно от трудозатрат. Разделение задачи между несколькими людьми вызывает дополнительные затраты на обучение и обмен информацией.

Стоимость разработки и ее трудоемкость рассчитывается по данным, которые могут быть либо получены в результате экспертной оценки специалистами, либо на основе аналогичных разработок, выполненных прежде. Очевидно, что во втором случае данные собираются в течение длительного времени по большому числу проектов и должны быть хорошо систематизированы и документированы. В результате обработки этих данных стремятся установить определенные зависимости между параметрами программного изделия и трудоемкостью его разработки. Подобные зависимости могут быть положены в основу эмпирических моделей, позволяющих достаточно просто оценивать трудоемкость разработки программной продукции. Стоимость и трудозатраты разработки программного обеспечения оцениваются, как правило, с использованием декомпозиции ПО либо методом сверху вниз, либо снизу вверх. В первом случае интегральная оценка проекта осуществляется по общим характеристикам программного обеспечения, а затем распределяется по компонентам, а во втором - вначале оцениваются работы по каждому компоненту ПО, а результаты затем суммируются.

Оценка стоимости и требуемых усилий на разработку программного обеспечения не базируется на строгой научной основе, что обусловлено наличием многочисленных факторов, оказывающих влияние на эти оценки. Вместе с тем на практике следуют некоторым методическим рекомендациям, которые позволяют получить оценки с приемлемым риском. Для достижения достаточно надежных оценок стоимости и усилий есть следующие возможности:

■ откладывайте оценку на более поздний срок:

■ используйте для оценки простые методы декомпозиции:

■ разрабатывайте и используйте эмпирические модели для оценки стоимости и усилий:

■ приобретайте и используйте одно или несколько автоматизированных средств для получения оценок.

Первая возможность привлекательна, но практически нереализуема. Действительно, чем дольше мы откладываем момент определения оценок, тем больше мы знаем о разрабатываемом программном продукте и тем менее вероятны грубые ошибки в наших оценках. Но, к сожалению, оценки нужны на начальных этапах проектирования.

Оставшиеся три возможности следует использовать в комбинации друг с другом для взаимного контроля полученных оценок. Методы декомпозиции основаны на разделении

большой проблемы на ряд под проблем и на соответствующие им задачи автоматизации. Оценка стоимости и усилий может быть проведена пошагово применительно к отдельным блокам программного изделия.

Эмпирические модели оценки, которые целесообразно использовать в качестве дополнения к методам декомпозиции, а также самостоятельно, основаны, как правило, на накопленных статистических данных разработки аналогичных программных продуктов. В этих моделях оцениваемая величина (стоимость и трудозатраты) рассматривается как функция некоторых параметров проекта.

Автоматические средства оценки обычно реализуют один или несколько методов декомпозиции или эмпирических моделей, объединенных с интерактивным так называемым человеко-машинным интерфейсом, который позволяет пользователю изменять параметры проекта и анализировать различные варианты оценок для широкого диапазона влияющих факторов. Все рассматриваемые подходы дают падежные оценки, если используются хорошие данные о прежних разработках. Рассмотрим один метод подсчета.

Методы функциональной декомпозиции

Лучший подход к решению любой сложной проблемы разделение ее на более простые и решение каждой в отдельности с последующим объединением полученных решений. Оценка проекта программного изделия - пример сложной проблемы, которую целесообразно разделить на более простые.

На практике широко используются оценки, основанные на размере программного продукта числе строк кода. (LOC lines of code.) Данные о числе строк кода при оценке проекта программного изделия используются:

■ как оценочные переменные для разрабатываемого проекта;

■ как показатели базовой линии, полученные из прошлых проектов и используемые для разработки оценок стоимости и усилий рассматриваемого проекта.

Первый этап процесса получения оценок - является функциональная декомпозиция проектируемого проекта. Для каждой выделенной нами функции определяем число строк кода в виде диапазона значений, указывая оптимистическую, пессимистическую и наиболее вероятную оценки, а затем подсчитывается ожидаемая величина для LOC-оценки рассматриваемой функции. Оценки для каждой функции лучше когда дают группы экспертов.

После получения LOC-оценок по каждому функциональному блоку проекта необходимо обратиться к базовым оценкам производительности труда и удельной стоимости проектирования и разработки одной строки кода полученным для предыдущих разработок.

Можно применить два различных подхода:

■ получить интегральную оценку для всего проекта в целом, используя суммарное число строк кода и усредненные показатели производительности и стоимости работ. В результате будет получена оценка общих затрат на проектирование изделия в рублях и оценка усилий (в человеко-месяцах) на разработку проекта:

■ получить опенку затрат и усилий для каждой отдельной функции, используя для каждой функции свои Показатели по производительности и стоимости работ. Этот подход дает более точные оценки, но требует более детальной информации и 5 прошлых проектов.

Методом, позволяющим определить цену разрабатываемого проекта, является также так называемый метод оценки усилий. В нем как и в предыдущем случае, проблема разбивается на ряд функциональных блоков (подсистем), а затем для каждого блока производится экспертная оценка трудозатрат (человеко-месяцев) на выполнение каждой фазы жизненного цикла программного изделия. Так можно оценить трудозатраты на проектирование каждого функционального блока и на выполнение каждой фазы разработки. В этом случае для каждой инженерной задачи могут быть использованы свои оценки оплаты труда соответствующих специалистов таких как аналитики, программисты и подсчитана общая стоимость разработки изделия. Кроме того, легко может быть получена общая оценка трудозатрат на разработку всего изделия.

Полученные оценки стоимости и усилий целесообразно сравнить с оценками, полученными па основе размерных характеристик продукта (числа строк кода). Оценки в обоих случаях должны быть достаточно близкими. Значительные расхождения в оценках могут быть из-за:

■ неадекватного понимания сферы применения проекта или неправильной интерпретации границ его использования;

■ неправильных или устаревших данных о производительности труда в LOC-методе или их несоответствия прикладной области.

В этом случае необходимо выявить причину расхождения оценок и провести повторные расчеты.

7.1.Курсовая работа

Содержание

О том, как написать курсовую работу задумывается каждый студент. Это задание выполняют все учащиеся ВУЗов и средне-специальных учебных заведений независимо от формы обучения и специальности. Написание курсового проекта является головной болью многих студиозусов, ведь в самом названии этого типа работы отражена его суть — закрепляются и проверяются знания по целому курсу, при этом, как правило, курсовые пишутся по профильным дисциплинам. Этим и обусловлена важность успешного прохождения этого этапа обучения, несвоевременная сдача работы легко может

послужить причиной отчисления из учебного заведения, и, если при сдаче экзамена или зачёта можно понадеяться на счастливый билет, то к вопросу подготовки курсовой работы нужно подходить очень серьезно.

Что такое курсовая работа

Курсовая исследовательская работа или **курсовой проект** — это один из важных этапов обучения, который заключается в самостоятельном научном исследования студентом определённой темы профильной дисциплины.
В магистратуре курсовая может называться научно-исследовательской работой, но сути дела это не меняет.

Цель написания курсовой работы – это закрепление студентом знаний, полученных в ходе учебного процесса по пройденному предмету. Благодаря курсовой работе преподавателю легче проверить качество полученных студентом знаний и способность применять эти знания к решению профессиональных задач.

Темы курсовых работ утверждаются на соответствующей кафедре, доводятся до сведения студентов, где они, как правило, выбирают понравившуюся тему. Если студент отсутствовал в это время, тогда ему тема назначается (из числа оставшихся). Бывает что темы раздают по списку без права выбора.

Пример курсовой работы

Задание может отличаться в зависимости от того, по какой специальности и на каком курсе обучается студент. Иногда требования диктуют необходимость проведения полноценного научного исследования, а иногда достаточно проработки теоретического материала.

К примеру, курсовая работа по психологии студента первого курса может состоять только из теории, а его будущие коллеги со старших курсов должны провести эмпирическое исследование.

Курсовая работа может содержать:

1. титульный лист;
2. содержание;
3. введение;
4. основную часть;
5. практическую часть;
6. заключение;
7. список литературы;
8. приложения.

Этапы написания курсовой работы

Чаще всего выполнение курсового задания предполагает прохождение нескольких этапов.

Подготовительный.

Возможность самостоятельного выбора научного руководителя — большая удача для студента, очень важно быть с ним «на одной волне», это позволит заручиться его поддержкой, что немаловажно. Кроме того, у каждого преподавателя существует несколько профилирующих тем, и лучше, чтобы они были интересны и студенту.

Лайфхак для студента: залог успеха — подойти к научному руководителю для обсуждения деталей курсового проекта сразу, даже если в планах нет приступить к работе в ближайшее время. Таким образом, можно показать свою заинтересованность и серьезный настрой.

Изучение общей информации и составление плана.

Первое что нужно сделать, получив тему курсовой работы, это подобрать и изучить литературу по теме. О том, как найти литературу для любой курсовой и дипломной работы читайте нашу статью.

Лайфхак для студента: по сути курсовая – это та же дипломная, только меньше раза в 3 по объему, и чаще всего, дипломная работа пишется на основании курсовых, поэтому качественная проработка этих этапов позволит сэкономить свое время и нервы в будущем — останется только объединить всё в дипломе.
Далее необходимо составить< содержание курсовой работы и согласовать его с научным руководителем. Читайте — «Как составить содержание курсовой работы»

Подбор источников и составление списка литературы.

Как правило, преподаватель вручает своему подопечному список рекомендуемой литературы. Но так как одной из целей курсовой работы является научить студента проводить самостоятельные научные исследования, то часть источников придётся искать самостоятельно.

Лайфхак для студента: для научной работы не подойдут рефераты из интернета. Придётся искать серьезные научные труды, многие из них отсутствуют в открытом доступе в сети, поэтому следует готовиться к путешествию в научную библиотеку. Однако кое-что можно найти и находясь дома за компьютером.
Важно: во многих ВУЗах требуется использовать источники на иностранном языке. Не стоит бездумно вставлять в список первый попавшийся труд, который случайно попался в примечаниях какой-нибудь монографии — вдруг это редкое издание, не имеющее перевода на русский язык. Лучше использовать книги, которые есть возможность хотя бы пролистать.

Написание курсовой работы.

Итак, после согласования темы работы и источников, на основании которых она будет строиться, можно приступать к самому ответственному этапу курсового проекта. Он, в свою очередь, должен состоять из нескольких шагов.

Как же писать курсовую работу самому? Как правильно написать курсовую работу — читайте далее пошаговые инструкции.

- **Шаг 1.** Написание введения. В этом разделе необходимо описать актуальность работы, степень ее разработки, методы исследования и прочее. Много полезного можно найти в статьях «Как написать введение к курсовой работе» и «Как написать введение к дипломной работе».
- **Шаг 2.** Написание основной части курсовой работы. Основная часть работы должна содержать 2 -3 главы, всесторонне раскрывающих тему курсовой работы. Каждую главу необходимо разделить на несколько параграфов. Каждая глава должна содержать одинаковое количество параграфов. Необходимо выделять основные тезисы, каждый из которых должен содержать доказательную базу. Материал необходимо излагать последовательно и лаконично, чтобы один вопрос логично вытекал из другого. Более подробно о том, как пишется основная часть научной работы можно почитать в нашей статье.

Важно: не стоит забывать, что весь теоретический материал необходимо уникализировать, т.е изложить своими словами. В противном случае работа будет признана плагиатом и не допущена к защите. При этом в разных ВУЗах, а иногда и на разных кафедрах одного учёбного заведения, используются разные системы проверки, поэтому лучше сразу уточнить, какой пользуется научный руководитель.

Важно: при написании курсовой работы лучше сразу применять корректное оформление. Как это сделать можно посмотреть в видео ниже:

- **Шаг 3.** Выполнение практической части. Так, для инженеров ПГС — это чертежи, выполненные при помощи программы AutoCAD, ArchiCAD и т.д., для математиков — при помощи программ Mathcad, Maple и др.; для бухгалтеров – это анализ финансовой деятельности предприятия (используется программа 1С и прочие). Более подробную информацию о том, как правильно писать этот раздел можно найти в статье: Теоретические и практические части курсовых работ.
- **Шаг 4.** Написание заключения (формулировка кратких, но емких выводов по теме). Заключение курсовой работы содержит выводы, итоги решения поставленных задач, проанализированных и решенных в работе, описание того, какой вклад внесла курсовая работа в современную науку. Примеры заключений курсовой работы можно посмотреть здесь.
- **Шаг 5.** Оформление списка литературы. Здесь информация о том Как правильно оформить список литературы в курсовой работе.
- **Шаг 6.** Оформление приложений. В приложение выносятся графические, табличные, иллюстрационные материалы курсовой работы. Приложения в курсовой, как правило, не нумеруются. Более подробно о том, как правильно работать с этим разделом Смотрите в нашей статье.
- **Шаг 7.** Проверка курсовой работы преподавателем, исправление недочетов. О том, что может стать причиной неудовлетворительной оценки можно почитать здесь.

- **Шаг 8.** Подготовка к защите проекта: написание краткого, но объемного конспекта, то есть охватывающего основные вопросы и проблемы по теме. Не стоит недооценивать этот этап, часто от него зависит 50% успеха проекта: главное — правильно преподнести свою работу и держаться уверенно.

Заключение

Итак, написание курсовой работы — процесс весьма трудоёмкий. Однако, по мере накопления опыта, этот процесс будет занимать у вас все меньше и меньше времени. Так что будет вполне реально написать курсовую быстро и даже за одну ночь. Тем не менее, лучше все же не откладывать написание курсовика на последний момент.

Если же студент не уверен, что он располагает временем и силами на самостоятельное выполнение курсовой, лучше обратиться за помощью к профессионалам. Это позволит своевременно сдать курсовую работу, которая будет соответствовать всем необходимым требованиям.

Темы курсовых работ

Технология разработки программного обеспечения

1	Разработки информационной подсистемы по анализу расхода топлива
2	Разработка информационной подсистемы по Анализ финансового состояния предприятия
3	Разработка информационной подсистемы системы по учету персонала
4	Разработки информационной подсистемы по анализу расхода топлива в зависимости от расстояния
5	Разработки информационной подсистемы по анализу расхода топлива зависимости от климата
6	Разработка информационной системы "Оптовая продуктовая база"
7	Разработка информационной подсистемы учета строительно-монтажных работ
8	Разработка информационной подсистемы программы автоматизации процесса подбора запчастей для ремонта автомобилей
9	Разработка дистанционного обучения в рамках МДК 01.02 Информационное обеспечение перевозочного процесса (по видам транспорта)
10	Автоматизация работы ресторана
11	Разработка информационной системы складского учета медицинской аптеки.
12	Разработка информационной системы станции техобслуживания компьютеров.
13	Проектирование информационной подсистемы «Банк – модуль «Кредитный калькулятор»»
14	Разработка информационной системы складского учета ювелирного магазина
15	Разработки информационной подсистемы по созданию и заполнению календарно-тематического планирования в соответствии с учебным планом

16	Разработка информационной подсистемы управления заказами клиентов для мебельной фабрики
17	Проектирование электронного учебника по применению программных продуктов, используемых при подготовке студента специальности 09.02.03 Программирование в компьютерных системах по дисциплине Информатика и ИКТ
18	Разработка подсистемы тестирования специалистов по защите информации
19	Разработка информационного и программного обеспечения электронного учебника при подготовке студента специальности 09.02.03 Программирование в компьютерных системах по дисциплине Основы программирования
20	Разработка информационной подсистемы приема заказов на подключения цифрового телевидения
21	Разработка информационной подсистемы по оперативно-диспетчерскому управлению автобусного парка
22	Разработки информационной подсистемы по созданию и заполнению рабочей программы в соответствии с учебным планом
23	Разработки информационной подсистемы по расчету калькуляции строительства жилого дома
25	Разработки информационной подсистемы по расчету калькуляции себестоимости готовых блюд в ресторанах и предприятиях общепита
26	Программная реализация автоматизированной системы складского учета, для фирмы, торгующей компьютерами и их комплектующими.
27	Разработки информационной подсистемы по расчету заработной платы (сдельной, повременной) на основе ОАО «Хлебокомбинат».

7. Словарь ключевых терминов

A

Автогад — система автоматизированного проектирования и черчения AutoCAD.

Аджайл (от англ. Agile) — общий термин, характеризующий подходы гибкой разработки. Слово стало широко использоваться программистами после Манифеста гибкой разработки ПО в 2001 году.

Адаптив — адаптивный дизайн, то есть дизайн веб-страниц, обеспечивающий правильное отображение сайта на разных устройствах.

Айдишник — идентификатор. Сленговое слово пошло от понятия id (англ. identifier).

Альфа — период работы над программным обеспечением, когда разработчики добавляют новые функции, а тестировщики проверяют корректность их работы.

Апишка — API. Программный интерфейс приложения или интерфейс прикладного программирования.

Апликуха — приложение (англ. application).

Аппрувить — согласовывать что-то (англ. approve).

Апгрейд — качественное улучшение чего-либо, обычно это слово употребляют, описывая обновления начинки компьютера (англ. upgrade).

Апдейт — то же самое, что и апгрейд, только касаемо модернизации программного обеспечения (англ. update).

Аттач — приложение к чему-либо, прикрепляемый файл (англ. attachment). Аттачить — прикреплять.

Артефакт — искажение цветов, форм объектов, несовместное движение частей изображения и т.д.

Аутсорс — аутсорсинг, передача некоторых задач подрядчикам.

Б

Баг — ошибка, всплывающая в программе (англ. bug — клоп, жучок).

Багрепорт — отчёт об ошибке с пояснением причины возникновения (англ. bug report).

Банщик — дизайнер, который верстает рекламные баннеры.

Батник — командный файл с расширением .bat. Используется для работы с командной строкой в Windows.

Батон — кнопка (англ. button).

Бахнуть — за короткое время изменить или дополнить функции программы или приложения.

Бенефит — бонус, премия.

Бенч — режим ожидания. Сленговое слово используется при простое в задачах, когда программист ждёт новые задачи и фактически ничего не делает.

Битый — нерабочий. Например, «битая ссылка».

Бэкап — резервная копия, бэкапить — создавать ее. Бэкапить информацию надо как можно чаще, чтобы вернуть удаленные данные или сохранить работу, если произойдет сбой.

Бэклог — порядок функций в зависимости от приоритета для их использования в следующих версиях продуктов.

В

Валидный — действительный, соответствующий требованиям (англ. valid).

Варез — программное обеспечение, полученное с нарушением авторского права (англ. warez).

Варезник — сайт, на котором размещены пиратские фильмы, музыка, ПО.

Виндузятник — неуважительное название пользователя Windows.

Воркшоп — обучающее мероприятие, включающее в себя практику (англ. workshop).

Ворнинг — предупреждение (англ. warning).

Выкатить — опубликовать обновления программы, приложения, игры и тд.

Выпадашка — выпадающее меню.

Г

Галера — компания, где платят низкие по рынку зарплаты и не ценят программистов.

Гик — фанат своего дела. Слово часто применяется по отношению к программистам.

Гребцы — сотрудники галеры.

Грумить — «причесывать», то есть приводить в порядок и устранять недочеты.

Гит — система контроля версий Git или сервис GitHub.

Гуй — графический интерфейс пользователя.

Грызун — компьютерная мышь.

Д

Дамп — файл с полной или частичной копией содержимого памяти ПК или базы данных.

Движок — системы управления контентом, обычно это слово используют в веб-разработке.

Дебажить код — проверять код на ошибки или «баги» (англ. to debug).

Деплой — публикация рабочей версии приложения в том месте, где оно должно функционировать.

Деплоить — публиковать и переносить. Например, задеплоить сайт = перенести его с тестового на рабочий сервер.

Джун — начинающий специалист с минимальным опытом, младший разработчик.

Директория — папка.

Драй — принцип программирования, предлагающий избегать повторений кода.

Дрова — драйверы.

Дропать — удалять что-то. Иногда просто ненужное, иногда — ошибки.

Дропдаун — «выпадашка», выпадающее меню.

Дыра — недочет, лазейка в коде, которую могут использовать хакеры.

Ж

Жаба — язык программирования Java.

Жабаскрипт — язык программирования JavaScript.

Железо — начинка ПК. Аппаратное обеспечение (комплектующие) устройств.

З

Зазиповать — архивировать файлы в формате zip.

Законнектиться — присоединиться к чему либо, например, к рабочей онлайн-конференции.

Зарарить — архивировать файлы в формате rar.

Залить — загрузить.

Запилить — добавить функции.

Звуковуха — звуковая карта.

Змея — язык программирования Python.

И

ИБ — информационная безопасность.

Ибешник — сотрудник отдела информационной безопасности.

Имба — несбалансированный персонаж, предмет или иной элемент компьютерной игры.

Исходник — файл с первоначальным вариантом кода.

Индусский код — длинный и сложно написанный код, в котором есть лишние строки.

Иксы — стандарт X Window System.

К

Камень — процессор.

Капча — картинка, которая позволяет убедиться, что пользователь не робот.

Кастомный — сделанный на заказ под потребности клиента.

Коммитить — сохранять код. Например, скоммитить код в приложении = сохранить код в репозитории.

Костыль — код, который поможет исправить несовершенства имеющегося кода. Метод исправления ошибок без серьёзного вмешательства, чтобы код «не хромал».

Кодер — программист.

Кодить — программировать.

Код-ревью — проверка кода.

Кирпич — неработающее или сломанное устройство.

Кракозябры — бессмыслица, набор непонятных символов.

Кресты — язык программирования C++.

Кряк — программа или дополнение к ПО для взлома данного ПО.

Крякнуть — взломать.

Л

Лаг — задержка в реакции программы на действия пользователя.

Ламер — неопытный человек, возомнивший себя профи.

Легаси-код — очень старый код, написанный предыдущей командой.

Либа — библиотека.

Линтер — программы, которые анализируют код и предупреждают об ошибках.

Лицуха — лицензия.

Лог-файл — файл, в котором записываются в хронологическом порядке все выполненные действия в программе.

М

Майнинг — добыча чего-либо.

Мамка — материнская плата.

Мануал — инструкция.

Мержить — объединять или выполнять слияние веток кода.

Меншить — ставить упоминания в чатах или соцсетях.

Митап — встреча специалистов для обмена опытом.

Моб — нестатичный объект в игре.

Моник — монитор.

Мост — сетевое устройство второго (канального) уровня сетевой модели OSI.

Мэтчить — сопоставлять наборы данных из разных источников

Н

Навбар — навигационный блок на сайте или в интерфейсе программы.

Накатить — внести изменения, загрузить новую версию ПО.

Ноулайфер — человек, который большую часть жизни проводит за компьютером.

Нуб — новичок, у которого ничего не получается.

О

Откатить — вернуть все как было, отменить обновления.

Ось — операционная система.

Оверлокинг — оптимизация, «разгон» компьютера.

Оверклокер — специалист, модернизирующий ПК.

Овертайм — переработка.

Опенсорс — программное обеспечение с открытым исходным кодом, которое может использовать/дополнять/изменять любой желающий.

Операнды — данные, с которыми работает выражение.

Откат — возвращение на исходную позицию.

Оффтоп — сообщение не по теме.

П

Падаван — неуважительное название стажера или джуна.

Парсить — собирать данные.

Патч — дополнение или обновление для исправления багов (ошибок).

Песочница — безопасное пространство для выполнения программы.

Пингагуть — проверить доступность определенного IP-адреса, сетевую связность.

Пилот — пробная версия продукта.

Питон — язык программирования Python.

Плюшки — бонусы и подарки.

Профит — выгода, польза.

Подвал — самая нижняя часть страницы.

Пушить — публиковать что-то.

Пэхапэ — язык программирования PHP.

Р

Разраб — разработчик.

Рандом — произвольный, случайный.

Ребутнуть — перезагрузить.

Редирект — перенаправление пользователя с одного сайта на другой.

Релиз — выпуск готовой версии продукта.

Репа — репозиторий, хранилище данных.

Ридми — файл Readme, в котором содержится информация о программе.

Рояль — клавиатура.

С

Саббатикал — творческий отпуск, чтобы избежать эмоционального выгорания.

Сабж — тема.

Саппорт — служба поддержки.

Свитчнуть, свичнуть — переключить.

Сетка — модульная сетка для дизайна и вёрстки страниц.

Сейвить — сохранять.

Секьюрный — защищённый, безопасный.

Сервак — сервер.

Сишка — язык программирования C.

Сионист — программист, пишущий на языке программирования C.

Сиквел — язык SQL или сервер SQL.

Снести — удалить.

Стек — список технологий, использующихся компанией или разработчиком.

Софт — программное обеспечение.

Скиллы — навыки.

Т

Таска — задание.

Темплейт — шаблон.

Тимлид — начальник, руководитель IT-команды.

Трейни — стажёр.

Туториал — учебник.

У

Упс — источник (устройство) бесперебойного питания электроэнергией.

Утилита — вспомогательная программа.

Уши — наушники.

Ф

Фаервол — программа до защита сегментов сети или отдельных хостов от несанкционированного доступа.

Фак — часто задаваемые вопросы.

Факап — неудача.

Факапить — делать ошибки и проваливать задачи.

Фича — функция.

Фидбек — обратная связь.

Фиксить — исправлять ошибки.

Фича — уникальная особенность.

Фреймворк — инструмент разработки, набор типовых шаблонных решений, упрощающих работу программиста.

X

Хаб — класс устройств для объединения компьютеров в сетях Ethernet.

Хакатон — мероприятие для разработчиков, на котором они в течение нескольких дней работают над каким-то продуктом.

Хакнуть — взломать.

Хатэмээль, хатээмэль — язык гипертекстовой разметки HTML.

Хацкер — название начинающего специалиста.

8. Тесты

1 Какие программы можно отнести к системному ПО

1) **+драйверы**
2) текстовые редакторы
3) электронные таблицы
4) графические редакторы
5) все ответы верны

2 Специфические особенности ПО как продукта

1) +продажа по ценам ниже себестоимости (лицензирование)

2) низкие материальные затраты при создании программ

3) возможность создание программ небольшие коллективом или даже одним человеком

4) разнообразие решаемых задач с помощью программных средств

5) все ответы верны

3 Какие программы нельзя отнести к системному ПО

1) **+игровые программы**
2) компиляторы языков программирования
3) операционные системы
4) системы управления базами данных
5) все ответы верны

4 Специфические особенности ПО как продукта

1) **+низкие затраты при дублировании**
2) универсальность
3) простота эксплуатации
4) наличие поддержки (сопровождения) со стороны разработчика
5) все ответы верны

5 Какие программы можно отнести к системному ПО

1) **+утилиты**
2) экономические программы
3) статистические программы
4) мультимедийные программы
5) все ответы верны

6 Этап, занимающий наибольшее время, при разработке программы

1) +тестирование
2) сопровождение
3) проектирование
4) программирование
5) формулировка требований

7 Первый этап в жизненном цикле программы

1) +формулирование требований
2) анализ требований
3) проектирование
4) автономное тестирование
5) комплексное тестирование

8 Один из необязательных этапов жизненного цикла программы

1) +оптимизация
2) проектирование
3) тестирование
4) программирование
5) анализ требований

9 Самый большой этап в жизненном цикле программы

1) +эксплуатация
2) изучение предметной области
3) программирование
4) тестирование
5) корректировка ошибок

10 Какой этап выполняется раньше

1) отладка
2) оптимизация
3) +программирование
4) тестирование
5) все ответы верны

11 Что выполняется раньше

1) +компиляция
2) отладка
3) компоновка
4) тестирование
5) нет правильного ответа

12 Что выполняется раньше

1) +проектирование
2) программирование
3) отладка
4) тестирование
5) компоновка

13 В стадии разработки программы не входит

1) **+автоматизация программирования**
2) постановка задачи
3) составление спецификаций
4) эскизный проект
5) тестирование

14 Самый важный критерий качества программы

1) +работоспособность
2) надежность
3) эффективность
4) быстродействие
5) простота эксплуатации

15 Способы оценки качества

1) +сравнение с аналогами
2) наличие документации
3) оптимизация программы
4) структурирование алгоритма
5) хранение и запоминание информации

16 Наиболее важный критерий качества

1) **+надежность**
2) быстродействие
3) удобство в эксплуатации
4) удобный интерфейс
5) эффективность

17 Способы оценки надежности

1) +тестирование
2) сравнение с аналогами
3) трассировка
4) оптимизация
5) удобный интерфейс

18 В каких единицах можно измерить *надежность*

1) +отказов/час
2) км/час
3) Кбайт/сек
4) операций/сек
5) мб/сек

19 В каких единицах можно измерить *быстродействие*

1) отказов/час
2) км/час
3) Кбайт/сек
4) **+операций/сек**
5) мб/сек

20 Что относится к этапу программирования

1) +написание кода программы
2) разработка интерфейса
3) работоспособность
4) анализ требований
5) создание базы данных

21 Последовательность этапов программирования

1) +компилирование, компоновка, отладка
2) В) компоновка, отладка, компилирование
3) отладка, компилирование, компоновка
4) компилирование, отладка, компоновка
5) все ответы верны

22 Инструментальные средства программирования

1) +компиляторы, интерпретаторы
2) СУБД (системы управления базами данных)
3) BIOS (базовая система ввода-вывода)
4) ОС (операционные системы)
5) нет правильного ответа

23 На языке программирования составляется

1) **+исходный код**
2) исполняемый код
3) объектный код
4) **алгоритм**
5) предметный код

24 Правила, которым должна следовать программа это

1) **+алгоритм**
2) структура
3) спецификация
4) состав информации
5) последовательность

25 Доступ, при котором записи файла читаются в физической последовательности, называется

1) **+последовательным**
2) прямым
3) простым
4) основным
5) вторичным

26 Доступ, при котором записи файла обрабатываются в произвольной последовательности, называется

1) **+прямым**
2) последовательным
3) простым
4) основным
5) вторичным

27 Методы программирования (укажите НЕ верный ответ)

1) **+логическое**
2) структурное
3) модульное
4) компиляторное
5) линейное

28 Что выполняется раньше

1) **+разработка алгоритма**
2) выбор языка программирования
3) написание исходного кода
4) компиляция
5) Все ответы верны

29 Найдите НЕ правильное условие для создания имен

1) **+имена могут содержать пробелы**
2) длинное имя можно сократить
3) из имени лучше выбрасывать гласные
4) можно использовать большие буквы
5) нет правильного ответа

30 Какие символы не допускаются в именах переменных

1) **+пробелы**
2) цифры
3) подчеркивание
4) знаки препинания
5) заглавные буквы

31 Как называется способ составления имен переменных, когда в начале имени сообщается тип переменной

1) прямым указанием
2) **+венгерской нотацией**
3) структурным программированием
4) поляризацией
5) Нет правильного ответа

32 На каком этапе производится выбор языка программирования

1) **+проектирование**
2) программирование
3) отладка
4) тестирование
5) разработка

33 Для решения экономических задач характерно применение

1) **+СУБД (систем управления базами данных)**
2) языков высокого уровня
3) языков низкого уровня
4) применение сложных математических расчетов
5) Нет правильного ответа

34 Для решения инженерных задач характерно применение

1) **+САПР (систем автоматизированного проектирования)**
2) СУБД (систем управления базами данных)
3) ОС (операционных систем)
4) (ТРПП) Технология и разработка программного продукта
5) Нет правильного ответа

35 Причины синтаксических ошибок

1) +плохое знание языка программирования
2) ошибки в исходных данных
3) ошибки, допущенные на более ранних этапах
4) неправильное применение процедуры тестирования
5) неправильная установка ПО

36 Когда можно обнаружить синтаксические ошибки

1) +при компиляции
2) при отладке
3) при тестировании
4) на этапе проектирования
5) при эксплуатации

37 Ошибки компоновки заключаются в том, что

1) +указано внешнее имя, но не объявлено
2) неправильно использовано зарезервированное слово
3) составлено неверное выражение
4) указан неверный тип переменной
5) Все ответы верны

38 Защитное программирование это

1) **+встраивание в программу отладочных средств**
2) создание задач защищенных от копирования
3) разделение доступа в программе
4) использование паролей
5) оформление авторских прав на программу

39 Вид ошибки с неправильным написанием служебных слов (операторов)

1) +синтаксическая
2) семантическая
3) логическая
4) символьная
5) алгоритмическая

40 Вид ошибки с неправильным использованием служебных слов (операторов)

1) +семантическая
2) синтаксическая

3) логическая
4) символьная
5) алгоритмическа

41 Ошибки при написании программы бывают

1) +синтаксические
2) орфографические
3) лексические
4) фонетические
5) морфологические

42 Процедура поиска ошибки, когда известно, что она есть это

1) +отладка
2) тестирование
3) компоновка
4) транзакция
5) трансляция
6)

43 Программа для просмотра значений переменных при выполнении программы

1) +отладчик
2) компилятор
3) интерпретатор
4) трассировка
5) тестирование

44 Отладка – это

1) +процедура поиска ошибок, когда известно, что ошибка есть
2) определение списка параметров
3) правило вызова процедур (функций)
4) составление блок-схемы алгоритма
5) нет правильного ответа

45 Когда программист может проследить последовательность выполнения команд программы

1) +при трассировке
2) при тестировании
3) при компиляции
4) при выполнении программы
5) при компоновке

46 На каком этапе создания программы могут появиться синтаксические ошибки

1) **+программирование**
2) проектирование
3) анализ требований
4) тестирование
5) разработка ПО

47 Когда приступают к тестированию программы

1) **+когда программа уже закончена**
2) после постановки задачи
3) на этапе программирования
4) на этапе проектирования
5) после составления спецификаций

48 Тестирование бывает

1) **+автономное**
2) инструментальное
3) визуальное
4) алгоритмическое
5) структурное

49 Тестирование бывает

1) **+комплексное**
2) инструментальное
3) визуальное
4) алгоритмическое
5) структурное

50 При комплексном тестировании проверяются

1) **+согласованность работы отдельных частей программы**
2) правильность работы отдельных частей программы
3) быстродействие программы
4) эффективность программы
5) все ответы верны

51 Чему нужно уделять больше времени, чтобы получить хорошую программу

1) **+тестированию**
2) программированию
3) отладке
4) проектированию
5) разработке

52 Процесс исполнения программы с целью обнаружения ошибок

1) **+тестирование**
2) кодирование
3) сопровождение
4) проектирование
5) разработка

53 Автономное тестирование это

1) **+тестирование отдельных частей программы**
2) инструментальное средство отладки
3) составление блок-схем
4) пошаговая проверка выполнения программы
5) все ответы верны

54 Трассировка это

1) **+проверка пошагового выполнения программы**
2) тестирование исходного кода
3) отладка модуля
4) составление блок-схемы алгоритма
5) нет правильного ответа

55 Локализация ошибки

1) **+определение места возникновения ошибки**
2) определение причин ошибки
3) обнаружение причин ошибки
4) исправление ошибки
5) анализ данных

56 Назначение тестирования

1) **+повышение надежности программы**
2) обнаружение ошибок
3) повышение эффективности программы
4) улучшение эксплуатационных характеристик
5) приведение программы к структурированному виду

57 Назначение отладки

1) **+поиск причин существующих ошибок**
2) поиск возможных ошибок
3) составление спецификаций
4) разработка алгоритма
5) разработка проекта

58 Создание исполняемого кода программы без написания исходного кода называется

1) составлением спецификаций
2) отладкой
3) проектированием
4) +автоматизацией программирования
5) анализ данных

Какие символы не допускаются в именах переменных

58 Один из методов автоматизации программирования

1) структурное программирование
2) модульное программирование
3) +визуальное программирование
4) объектно-ориентированное программирование
5) машинное программирование

59 Автоматизация программирования позволяет

1) повысить надежность программы
2) **+сократить время разработки программы**
3) повысить быстродействие программы
4) ускорить процесс программы
5) все ответы верны

60 Что легко поддается автоматизации

+A) интерфейс

B) работа с файлами

C) сложные логические задачи

D) алгоритмизация

E) разработка ПО

61 Нахождение наилучшего варианта из множества возможных

+A) оптимизация

B) тестирование

C) автоматизация

D) отладка

E) сопровождение

62 Что такое оптимизация программ

+A) улучшение работы существующей программы

B) создание удобного интерфейса пользователя

C) разработка модульной конструкции программы

D) применение методов объектно-ориентированного программирования

E) Все ответы верны

63 Критерии оптимизации

+A) время выполнения или размер требуемой памяти

B) размер программы и ее эффективность

C) независимость модулей

D) качество программы, ее надежность

E) Нет правильного ответа

64 В чем заключается оптимизация условных выражений

A) в изменении порядка следования элементов выражения

B) в использовании простых логических выражений

C) в использовании сложных логических выражений

D) в использовании операций AND, OR и NOT

E) в использовании всех операций выражения

65 Оптимизация циклов заключается в

+A) уменьшении количества повторений тела цикла

B) просмотре задачи с другой стороны

C) упрощение задачи за счет включения логических операций

D) увеличении количества повторений тела цикла

E) упрощение задачи за счет отключения логических операций

66 Оптимизация программы это

+A) модификация

B) отладка

C) повышение сложности программы

D) уменьшение сложности программы

E) быстродействие программы

67 Критерии оптимизации программы

+A) быстродействие или размер программы

B) быстродействие и размер программы

C) надежность или эффективность

D) надежность и эффективность

E) Все ответы верны

68 Результат оптимизации программы

+A) эффективность

B) надежность

C) машино-независимость

D) мобильность

E) Все ответы верны

69 Сущность оптимизации циклов

+A) сокращение количества повторений выполнения тела цикла

B) сокращение тела цикла

C) представление циклов в виде блок-схем

D) трассировка циклов

E) поиск ошибок в циклах

70 Рекомендуемые размеры модулей

+A) небольшие

B) большие

C) равные

D) фиксированной длины

71 В чем заключается независимость модуля

+A) в написании, отладке и тестировании независимо от остальных модулей

B) в разработке и написании независимо от других модулей

C) в независимости от работы основной программы

D) в зависимости от работы вторичной программы

E) в разработка и написании в зависимости от вторичных программ

72 При модульном программировании желательно, чтобы модуль имел

A) большой размер

+B) небольшой размер

C) фиксированный размер

D) любой размер

E) Все ответы верны

73 Достоинство модульного программирования

+A) создание программы по частям в произвольном порядке

B) не требует компоновки

C) всегда дает эффективные программы

D) снижает количество ошибок

E) Все ответы верны

74 Недостаток модульного программирования

A) увеличивает трудоемкость программирования

+B) усложняет процедуру комплексного тестирования

C) снижает быстродействие программы

D) не позволяет выполнять оптимизацию программы

E) Все ответы верны

75 Достоинство модульного программирования

+A) возможность приступить к тестированию до завершения написания всей программы

B) не требует комплексного тестирования

C) уменьшает размер программы

D) повышает надежность программы

E) Все ответы верны

76 Программирование без GO TO применяется при

+A) структурном программировании

B) модульном программировании

C) объектно-ориентированном программировании

D) все ответы верные

E) машинном программировании

77 Достоинство структурного программирования

+A) можно приступить к комплексному тестированию на раннем этапе разработки

B) можно приступить к автономному тестированию на раннем этапе разработки

C) нет необходимости выполнять тестирование

D) можно пренебречь отладкой

E) Все ответы верны

78 Недостаток структурного программирования

+A) увеличивает размер программы

B) снижает эффективность

C) уменьшает количество ошибок

D) не требует отладки

E) Все ответы верны

79 Что такое объект, в объектно-ориентированное программировании

+A) тип данных

B) структура данных

C) событие

D) обработка событий

E) использование стандартных процедур

80. Инкапсуляция это
A) определение новых типов данных
B) определение новых структур данных
+C) объединение переменных, процедур и функций в одно целое
D) разделение переменных, процедур и функций
E) применение стандартных процедур и функций

81. Наследование это
A) передача свойств экземплярам
B) передача свойств предкам
+C) передача свойств потомкам
D) передача событий потомкам
E) Все ответы верны

82. Полиморфизм это
+A) изменение поведения потомков, имеющих общих предков
B) передача свойств по наследству
C) изменение поведения потомков на разные события
D) изменение поведения экземпляров, имеющих общих предков
E) Все ответы верны

83. Три "кита" объектно-ориентированного метода программирования
A) предки, родители, потомки
+B) полиморфизм, инкапсуляция, наследование
C) свойства, события, методы
D) визуальные, не визуальные компоненты и запросы
E) Все ответы верны

84. Какое утверждение верно
A) предки наследуют свойства родителей
B) родители наследуют свойства потомков
C) потомки не могут иметь общих предков
+D) потомки наследуют свойства родителей
E) Все ответы верны

85. Могут ли два визуальных компонента иметь общего предка
+A) да
B) нет
C) если их свойства совпадают
D) если их методы совпадают
E) Все ответы верны

86. Есть ли различие в поведении объекта и экземпляра того же типа
A) да
B) если у них есть общий предок
+C) нет
D) если у них нет общего предков
E) Все ответы неверны

87. Изменение свойств, приводит к изменению поведения экземпляра

A) нет
B) только для визуальных
C) только НЕ для визуальных
+D) да
E) Все ответы неверны

88. Процесс преобразования постановки задачи в план алгоритмического или вычислительного решения это

+A) проектирование

B) анализ требований

C) программирование

D) тестирование

E) Все ответы неверны

89. Составление спецификаций это

+A) формализация задачи

B) эскизный проект

C) поиск алгоритма

D) отладка

E) Все ответы неверны

90. Этап разработки программы, на котором дается характеристика области применения программы

+A) техническое задание

B) эскизный проект

C) технический проект

D) внедрение

E) рабочий проект

91 Укажите правильную последовательность создания программы

+A) формулирование задачи, анализ требований, проектирование, программирование

B) анализ требований, проектирование, программирование, тестирование, отладка

C) анализ требований, программирование, проектирование, тестирование

D) анализ требований, проектирование, программирование, модификация, трассировка

E) формулирование задачи, анализ требований, программирование, проектирование, отладка

92. Метод проектирования

+A) нисходящее

B) алгоритмическое

C) логическое

D) использование языков программирования

E) составление блок-схем

93. Нисходящее проектирование это

+A) последовательное уточнение (детализация)

B) составление блок-схем

C) разделение программы на отдельные участи (блоки)

D) трассировка

E) Все ответы верны

94. Признаки нисходящего программирования

+A) последовательная детализация

B) наличие оптимизации

C) наличие тестирования

D) автоматизация программирования

E) Все ответы верны

95. Модульное программирование применимо при

A) проектировании сверху вниз

B) проектирование снизу-вверх

+C) и в том, и другом случае

D) ни в коем случае

E) Все ответы неверны

96 В каких единицах измеряются затраты на проектирование

+A) в человеко-днях

B) в терабайтах

C) в гигабайтах

D) в килобайтах

E) в мегабайтах

97. Упорядоченная последовательность команд (инструкций) компьютера для решения конкретной задачи.

A. Свойство программы

B. Программное обеспечение

C. Постановка задачи

+D. Программа

E. Язык программирования

98. С позиции специфики разработки и вида программного обеспечения, на какие два класса делятся задачи?

A. Позиционные и функциональные

+B.Технологические и функциональные

C. Позиционные и непозиционные

D. Технологические и параметрические

E. Нет верного ответа

99. Какими последовательными действиями можно представить процесс создания программ?

A. Программирование, постановка задачи, построение алгоритма

B. Построение алгоритма, решение задачи

C. Построение алгоритма, программирование

+D. Программирование, построение алгоритма, постановка задачи

E. Постановка задачи, построение алгоритма решения, программирование

100. Постановка задачи - это …

A. упорядоченная последовательность команд компьютера для решения задач

B. точная формулировка решения задачи на компьютере с описанием входных и выходных данных

+C. совокупность связанных между собой функций, задач управления, с помощью которых достигается выполнение поставленных целей

D. система точно сформулированных правил

E. Все ответы верны

101. Алгоритм - это …

A. разбиение процесса обработки информации на более простые этапы

B. задача, подлежащая реализации с использованием средств информационных технологий

+C. точная формулировка решения задачи на компьютере с описанием входных и выходных данных

D. система точно сформулированных правил, определяющая процесс преобразования допустимых исходных данных в желаемый результат за конечное число шагов

E. нет верного ответа

102. Разбиение процесса обработки информации на более простые этапы (шаги выполнения), выполнение которых компьютером или человеком не вызывает затруднений

A. Массивы

B. Безопасность

C. Программное обеспечение

+D. Алгоритм

E. Все ответы неверны

103. Выполнимость - это …

A. конечность действий алгоритма решения задач, позволяющая получить желаемый результат при допустимых исходных данных за конечное число шагов

+B. разбиение процесса обработки информации на более простые этапы (шаги выполнения), выполнение которых компьютером или человеком не вызывает затруднений

C. действие алгоритма решения задач, позволяющая получить не желаемый результат при допустимых исходных данных за бесконечное число шагов

D. система точно сформулированных правил, определяющая процесс преобразования допустимых исходных данных в желаемый результат за конечное число шагов

E. нет верного ответа

104. Осуществляет разработку и отладку программ для решения функциональных задач

A. Системный программист

B. Программист-аналитик

+C. Прикладной программист

D. Администратор

E. Постановщик задач

105. Занимается разработкой, эксплуатацией и сопровождением системного программного обеспечения, поддерживающего работоспособность компьютера и создающего среду для выполнения программ

+A. Прикладной программист

B Программист-аналитик

C. Системный программист

D. Администратор БД

E. нет верного ответа

106. Анализирует и проектирует комплекс взаимосвязанных программ для реализации функций предметной области

A. Прикладной программист

B. Программист-аналитик

C. Системный программист

D. Постановщик задач

+E. Администратор

107. Участвует в процессе создания программ на начальной стадии работ

A. Администратор БД

+B. Прикладной программист

C. Постановщик задач

D. Системный программист

E. все ответы верны

108. Является основным потребителем программ

A. Прикладной программист

B. Программист-аналитик

C. Системный программист

D. Конечный пользователь

+E. Нет верного ответа

109. Свойство системы сохранять во времени в установленных пределах значения всех характеристик, определяющих способность системы выполнять требуемые функции в условиях заданных режимов эксплуатации

A. Дискретность

B. Экономичность

+C. Готовность

D. Работоспособность

E. Надежность

110. Возможность доступа к услугам АИС с использованием соответствующих технологий всегда, когда в ней возникает необходимость

A. Определенность

B. Работоспособность

C. Надежность

D. Экономичность

+E. Готовность

111. Количество и степень занятости ресурсов, процессов, ОП, внешней и внутренней памяти, каналов ввода/вывода, терминалов и каналов сети

A. Экономичность

B. Готовность

C. Надежность

+D. Определенность

E. Работоспособность

112. Устойчивость - …

A. характеризует способность к безотказному функционированию при наличии сбоев

B. возможность доступа к услугам АИС с использованием соответствующих технологий всегда, когда в ней возникает необходимость

C. Свойство системы сохранять во времени в установленных пределах значения всех характеристик, определяющих способность системы выполнять требуемые функции в условиях заданных режимов эксплуатации

D. количество и степень занятости ресурсов, процессов, ОП, внешней и внутренней памяти, каналов ввода/вывода, терминалов и каналов сети

+E. Нет верного ответа

113. Процесс обеспечивает возобновления нормально функционирования АИС

A. Устойчивость

+B. Перезапуск

C. Готовность

D. Надежность

E. Все ответы верны

С каким этапом жизненного цикла программного продукта связано с алгоритмизацией

114.Процесса обработки данных, детализацией функций обработки, разработкой структуры ПП, выбором методов и средств создания программ?

A. Документирование

B. Программирование

C. Сопровождение

D. Проектирование

+E. нет верного ответа

115. С каким этапом жизненного цикла программного продукта связано с технической реализацией проектных решений и выполнение с помощью выбранного инструментария разработчика (алгоритмические языки и системы программирования и.т.д.)?

A. Документирование

B. Проектирование структуры ПП

+C. Программирование, тестирование и отладка

D. Сопровождение ПП

E. Все ответы верны

116.На каком этапе жизненного цикла программного продукта составляются необходимые сведения по установке и обеспечению надежной работы ПП и т.д.?

A. Проектирование

B. Эксплуатация

C. Документирование

D. Программирование

+E. нет верного объекта

117. Жизненный цикл ПО - …

A. непрерывный процесс, который начинается с момент его полного изъятия из эксплуатации и заканчивается в момент принятия решения о необходимости его создания

+B. процесс, который начинается с момента его полного описания и заканчивается в момент принятия решения о необходимости его создания

C. непрерывный процесс, который начинается с момента принятия решения о необходимости его создания и заканчивается в момент его полного изъятия из эксплуатации

D. прерывающийся процесс, который начинается с момента написания структуры программы и заканчивается в момент его полного изъятия из эксплуатации

E. Нет верного ответа

118. На какие три группы процессов делится структура жизненного цикла ПО по стандарту ISO/IEC 12207?

A. Составные, действующие и вспомогательные процессы

B. Основные, дополнительные и остальные процессы

C. Вспомогательные, основные и дополнительные процессы

+D. Основные, вспомогательные и организационные процессы

E. Нет верного ответа

119. Основные процессы жизненного цикла ПО делятся на …

A. Процесс документирования, процесс обеспечения качества, процесс верификации

B. Процесс поставки, процесс обеспечения качества, процесс верификации

+C. Процесс управления, процесс создания инфраструктуры, процесс обучения

D. Процесс приобретения, процесс поставки, процесс разработки*

E. Процесс управления, процесс разработки, процесс обучения

120. Вспомогательные процессы жизненного цикла ПО делятся на …

A. Процесс документирования, процесс обеспечения качества, процесс верификации*

B. Процесс поставки, процесс обеспечения качества, процесс верификации

+C. Процесс управления, процесс создания инфраструктуры, процесс обучения

D. Процесс приобретения, процесс поставки, процесс разработки

E. Процесс управления, процесс разработки, процесс обучения

9. Методические рекомендации для студентов и преподавателей

В ходе работы со студентами необходимо обеспечить соблюдение законодательства и требований:
1) непрерывность и своевременность образовательного процесса;
2) соблюдение расписания занятий;
3) соблюдение требований рабочих программ дисциплин;
4) обеспечение студентам возможности получать необходимую информацию по дисциплинам;
5) обеспечение студентам возможности использования обратной связи;
6) обеспечение контроля текущей успеваемости и оценивания работы студента;
7) корректировка оценочных средств с целью их ориентирования на новый формат работы.

При организации работы со студентами необходимо соблюдать следующие требования:
1) вся работа должна осуществляться в рамках учебного портала (работа через портал фиксирует результаты освоения программы студентом и работу преподавателя). Работа с помощью электронной почты и других ресурсов возможна, но не заменяет работу на учебном портале.
2) необходимо соблюдать расписание – материалы и задания должны быть выложены не позднее дня занятия по расписанию, так как необходимо планировать деятельность студентов во времени по всем дисциплинам.
3) работа со студентами должна осуществляться на личной странице преподавателя.

Личная страница – это курс, наименование которого – ФИО преподавателя.
4) если преподаватель ведет несколько дисциплин, материалы по дисциплинам должны структурироваться отдельными элементами на личной странице преподавателя.
5) материалы должны быть выложены по всем формам обучения, в том числе дистанционная работа должна вестись со студентами заочной формы обучения независимо от времени проведения сессии.
6) материалы необходимо размещать по всем видам работы со студентами всех уровней обучения, в том числе по практике, научно-исследовательской деятельности.

Взаимодействие со студентами

 на учебном портале должны быть даны рекомендации по самостоятельной работе студента, но важно, что бы была организована контактная работа со студентами, то есть преподаватель организует работу таким образом, чтобы взаимодействие со студентом было регулярным и полноценным;
 задание необходимо оформлять с возможностью предоставления ответа студентом на Учебном портале (например, элемент «Задание»);
 задания, выполненные студентами, должны быть своевременно оценены преподавателем с проставлением баллов в АИС Тандем;
 оценка работы студента должна сопровождаться комментированием со стороны преподавателя с целью указания на ошибки студента (при необходимости);
 на сообщения студентов должны даваться своевременные ответы;
 при установлении ограничений по времени выполнения заданий необходимо учитывать скорость работы портала и возможность сбоев в его работе, установленные сроки должны позволять студентам выполнить задание.

Содержание личной страницы преподавателя

По каждой дисциплине минимальный набор материалов:

 краткая информация для студентов с описанием способов и методов работы на ближайшее время с необходимыми документами (сроками, графиками работы и т.д.).
 лекции (видео-лекция, презентация, презентация с аудио сопровождением, текст). Если по дисциплине не предусмотрены лекции, можно не выкладывать такой формат, но краткую информацию, направляющую студента в изучении дисциплины, необходимо обязательно указать.
 форум для обсуждения проблемных вопросов.

Методические рекомендации для студента по организации

самостоятельной работы

В современный период востребованы высокий уровень знаний, академическая и социальная мобильность, профессионализм специалистов, готовность к самообразованию и самосовершенствованию. В связи с этим должны измениться подходы к планированию, организации учебно-воспитательной работы, в том числе и самостоятельной работы студентов.

Прежде всего, это касается изменения характера и содержания учебного процесса, переноса акцента на самостоятельный вид деятельности, который является не просто

самоцелью, а средством достижения глубоких и прочных знаний, инструментом формирования у студентов активности и самостоятельности.

Виды и формы самостоятельной работы

Самостоятельная работа студентов предполагает многообразные виды индивидуальной и коллективной деятельности студентов, осуществляемые под руководством, но без непосредственного участия преподавателя в специально отведенное для этого аудиторное и внеаудиторное время.

Самостоятельная работа – это особая форма обучения по заданию преподавателя, выполнение которой требует творческого подхода и умения получать знания самостоятельно.

Структурно самостоятельную работу студента можно разделить на две части:

 организуемая преподавателем и четко описываемая в РПД;

 студент организует по своему усмотрению, без непосредственного контроля со стороны преподавателя.

Виды самостоятельной работы:

 познавательная деятельность во время основных аудиторных занятий;

 самостоятельная работа в компьютерных классах под контролем преподавателя в форме плановых консультаций;

 внеаудиторная самостоятельная работа студентов (в том числе с электронными ресурсами);

 самостоятельное овладение студентами материалом конкретных учебных модулей, предложенных для самостоятельного изучения;

 самостоятельная работа студентов по поиску материала, который может быть использован для написания рефератов, курсовых и квалификационных работ;

 учебно-исследовательская работа;

 научно-исследовательская работа;

 самостоятельная работа во время прохождения практик;

 другие виды.

Формы самостоятельной работы студентов:

 конспектирование;

 реферирование литературы;

 аннотирование книг, статей;

 выполнение заданий поисково-исследовательского характера;

 углубленный анализ научно-методической литературы;

 работа с лекционным материалом: проработка конспекта лекций, работа на полях конспекта с терминами, дополнение конспекта материалами

из рекомендованной литературы;

 подготовка сообщений, докладов, заданий;

 подготовка и участие в лабораторно-практических занятиях:

изучение теоретической базы, выполнение задания в соответствии с

инструкциями и методическими указаниями преподавателя, получение и

анализ результатов;

 научно-исследовательская работа, выполнение курсовых и квалификационных работ;

 подготовка ко всем видам контрольных испытаний;

 выполнение письменных контрольных работ;

 выполнение заданий по сбору материала во время практики;

 другие формы.

СРС должна способствовать развитию творческого потенциала студента.

Контроль за выполнением СРС должен быть сугубо индивидуальным, при том, что задания могут быть комплексными, групповыми.

Методические рекомендации для студента по организации

самостоятельной работы

Самостоятельная работа студентов на семестр регламентируется

графиком СРС, предусматривающим выполнение индивидуальных заданий, рефератов, курсовых работ и пр. по всем дисциплинам.

Организация самостоятельной работы студентов по дисциплине (курсу) планируется и организуется преподавателем и описывается в РПД и

Методических рекомендаций по организации самостоятельной работы по дисциплине», в которых подробно описываются предлагаемое содержание СРС, конкретные задания, сроки их выполнения, справочный материал, формы отчетности и способы контроля с критериями оценки.

Студенту следует:

Внимательно ознакомиться с этими материалами. Это позволит четко представить как перечень изучаемых тем, так и глубину их постижения.

Составить подборку литературы, достаточную для изучения предлагаемых тем. В РПД представлены перечни основной и дополнительный литературы. Это означает, что всегда есть литература, которая может не входить в данный список, но является рекомендованной для освоения темы. При этом следует иметь в виду, что необходимо изучать литературу различных видов:

 учебники, учебные и учебно-методические пособия;

 первоисточники, к которым относятся оригинальные работы теоретиков, разрабатывающих соответствующие проблемы. Первоисточники изучаются при чтении как полных текстов, так и хрестоматий, где работы классиков содержатся не полностью, а в виде избранных мест, подобранных тематически;

 монографии, сборники научных статей, публикации в журналах, эмпирический материал;

 справочная литература: энциклопедии, словари, тематические, терминологические справочники, раскрывающие категориально-понятийный аппарат.

Основное содержание той или иной проблемы следует уяснить, изучая учебную литературу. При этом важно понимать, что вопросы в истории любой науки трактуются многообразно. С одной стороны подобное многообразие объясняется различиями в мировоззренческих позициях, на которых стояли авторы; с другой - свидетельствует об их сложности, позволяет выделить наиболее значимый аспект в данный исторический период. Кроме того, работа с учебником требует постоянного уточнения сущности и содержания категорий посредством обращения к энциклопедическим словарям и справочникам.

Абсолютное большинство проблем носит не только теоретический, умозрительный характер, но самым непосредственным образом связано с реальной жизнью, с практикой социального развития, преодоления противоречий и сложностей в обществе. Это предполагает наличие у студентов не только знания категорий и понятий, но и умения использовать их в качестве инструмента для анализа социальных проблем. Иными словами, студент должен совершать собственные интеллектуальные усилия, а не только механически заучивать понятия и положения.

Соотнесение изученных закономерностей с жизнью, умение достигать аналитического знания предполагает у студента наличие мировоззренческой культуры. Формулирование выводов осуществляется прежде всего в процессе творческой дискуссии, протекающей с соблюдением методологических требований к научному познанию.

При работе с литературными источниками требуется соблюдать определённые методические приёмы:

 во-первых, сосредоточенно прочитать текст, уяснив и усвоив прочитанное;

 во-вторых, продумать прочитанное, определив для себя главные и сопутствующие идеи;

 в-третьих, сделать из прочитанного необходимые для памяти выписки;

 в-четвёртых, проанализировать новый материал из проработанного литературного источника.

Одним из важнейших этапов работы с литературными источниками

является ведение записей. Этот вид методической работы необходим как для

уяснения и усвоения знаний, так и для подготовки устного выступления и написания реферата. Письменная фиксация полученной информации способствует не только её лучшему запоминанию, но и более глубокому осмыслению. Самое главное при этом –

формирование собственных суждений. Римский философ Сенека говорил: «Что приобретается при чтении посредством пера – превращается в плоть и кровь».

Содержание литературных источников можно записывать в любой

удобной форме, в зависимости от предназначения записей. Известно несколько апробированных форм ведения записей: план, дословные выписки (цитаты), конспект, тезисы.

План – это перечень основных вопросов содержания книги или другого источника, в том числе и учебных лекций. План удобен для повторения материала, но только в ближайшее время после изучения

источника, так как в нём нет конкретного подробного изложения материала.

В тоже время его можно использовать как памятку и при подготовке устного

выступления на практическом занятии.

Выписки (цитаты) – это дословные записи фрагмента текста

источника, обязательно сопровождающиеся ссылкой на этот источник, оформленной в соответствии с установленными библиографическими требованиями.

Тезисы (греч. – основное положение) – краткая формулировка

основных положений устного или письменного источника без подробного

раскрытия содержания. Обычно тезис кратко отвечает на вопрос плана.

Тезисы можно формулировать своими словами, включая в них основные положения автора книги или лекции.

Фиксацию прочитанного материала можно производить в форме свободной записи. Она включает в себя различные комбинации названных выше письменных форм.

Конспект (лат. – обзор) – самая распространённая форма записи учебной информации, особенно получаемой в процессе лекций или других форм учебных занятий. Это сжатое, связное изложение основных положений изучаемого источника. Конспект помогает студентам зафиксировать и изучить прочитанную или полученную иным путём учебную информацию, позволяет по мере надобности возвращаться к ней для дополнительной

работы или в случае другой потребности. Записи в форме конспекта производятся в той же последовательности, которая имеется в источнике.

Чем лаконичнее запись, тем серьёзнее надо относиться к формулировкам, к

подбору понятий, вербальных и графических схем, рисунков. Большое

значение имеет выделение в конспекте различными графическими средствами (цвет, подчёркивание, символы и т.д.) основных проблем, формулировок, наиболее интересных и важных мыслей и пр.

Для поддержания работоспособности в ходе умственного труда необходимо придерживаться следующих правил:

Для улучшения мозгового кровообращения

Исходное положение (и.п.) - сидя на стуле. 1-2 - отвести голову назад и плавно наклонить назад, 3-4 - голову наклонить вперед, плечи не поднимать. Повторить 4-6 раз. Темп медленный.

И.п. - стоя или сидя, руки на поясе. 1 - поворот головы направо, 2 -

и.п., 3 - поворот головы налево, 4 - и.п. Повторить 6-8 раз. Темп медленный.

И.п. - стоя или сидя, руки на поясе. 1 - махом левую руку занести

через правое плечо, голову повернуть налево. 2 - и.п., 3 - тоже правой рукой.

Повторить 6-8 раз. Темп медленный.

Для снятия утомления с плечевого пояса и рук • И.п. - стоя или сидя, руки на поясе. 1 - правую руку вперед, левую вверх. 2 - переменить

положение рук. Повторить 3-4 раза, затем расслабленно опустить руки вниз и потрясти кистями, голову наклонить вперед. Темп — средний.

И.п. - стоя или сидя, кисти тыльной стороны на поясе. 1-2 – свести локти вперед, голову наклонить вперед. 3-4 - локти назад, прогнуться.

Повторить 6-8 раз, затем руки вниз и потрясти расслабленно. Темп медленный.

И.п. - сидя, руки вверх. 1 - сжать кисти в кулак. 2 - разжать кисти.

Повторить 6-8 раз, затем руки расслабленно опустить вниз и потрясти кистями. Темп средний.

Для снятия утомления с туловища

И.п. - стойка - ноги врозь, руки за голову. 1 - резко повернуть таз направо. 2 - резко повернуть таз налево. Во время поворотов плечевой пояс остается неподвижным. Повторить 6-8 раз. Темп средний.

И.п. - стойка - ноги врозь, руки за голову. 1-5 - круговые движения тазом в одну сторону. 4-6 - то же в другую сторону. 7-8 - руки вниз,

расслабленно потрясти кистями. Повторить 4-6 раз. Темп средний.

И.п. - стойка - ноги врозь. 1-2 - наклон вперед, правая рука скользит вдоль ноги вниз, левая, сгибаясь, вдоль тела вверх. 3-4 - и.п. , 5-8 - то же в

другую сторону. Повторить 6-8 раз. Темп средний.

Гимнастика для глаз

Быстро поморгать, закрыть глаза и посидеть спокойно, медленно считая до 5. Повторить 4-5 раз.

Крепко зажмурить глаза (считать до 3, открыть их и посмотреть вдаль (считать до 5). Повторить 4-5 раз.

Вытянуть правую руку вперед. Следить глазами, не поворачивая головы, за медленными движениями указательного пальца вытянутой руки влево и вправо, вверх и вниз. Повторить 4-5 раз.

Посмотреть на указательный палец вытянутой руки на счет 1- 4, потом перевести взгляд вдаль на счет 1-6. Повторить 4-5 раз.

В среднем темпе проделать 3-4 круговых движения глазами в правую сторону, столько же в левую. Расслабив глазные мышцы, посмотреть вдаль на счет 1-6. Повторить 1-2 раза.

Если трудно запомнить эти движения, то за рабочим столом, особенно при работе за компьютером, следует просто почаще менять позу, контролировать мышечное напряжение. Каждые 10-15 минут проверять, не напряжена ли спина, не подняты ли плечи, не утомлены ли руки от работы на клавиатуре, подвигаться на стуле, сжать и разжать пальцы, встать и походить по комнате. Эти несложные движения помогут поддержать тонус организма, избежать заболеваний, вызванных гиподинамией.

10. Условия реализации учебной дисциплины

3.1. Требования к минимальному материально-техническому обеспечению.

Реализация программы дисциплины требует наличие учебного кабинета «Информационная технология»

Методические указания:

1.Методические указания: по выполнению курсовой работы по дисциплине: «Технология разработки программных продуктов» Разработал: преподаватель информационных технологий:

Н.В.Данилова Челябинск 2017г.

2.методические указания по выполнению практической работы по дисциплине «Технология разработки программных продуктов» Новороссийских 2019г. Автор: Николаенко Т.П. – преподаватель НКРП

3.Системное программное обеспечение (СПО). (методические указания) Рубан В.И. Москва 2015г.

Наглядные пособия:

1. Плакат «Типичный процесс разработки ПО»
2. Плакат «процесс жизненного цикла ПО»
3. Презентация на тему: «Проектирование и разработка интерфейса ПО»
4. Видео по Питону: «Преимущества и недостатки программного языка Питон»

Технические средства обучении

1. Компьютер
2. Проектор
3. Доска интерактивная

3.2. Технические средства обучения:

Реализация учебной дисциплины требует наличия учебного кабинета оборудованного ТСО

Оборудование учебного кабинета:

- посадочные места по количеству студентов;

- рабочее место преподавателя;

- методические материалы по курсы дисциплины (курс лекций, методические рекомендации по подготовке к занятиям, дидактические единицы по дисциплине);

- компьютер;

- лицензионное программе обеспечение: Java, Notepade++, Sublime Text, Браузеры, Internet Explorer;

- интерактивная доска;

- проектор;

- комплекс мультимедиа – презентация по курсу дисциплины.

3. Контроль и оценка результатов освоения учебной дисциплины.

Критерии оценка выполнения студентами отчетных работ

Оценка "5" (отлично ставится в случае, если студент показывает правильное понимание сущности изучаемых ситуаций и закономерностей, методов и принципов; дает точное определение и истолкование основных понятий, принципов и методов; указывает все свойства тех или иных объектов изучения; выполняет работу полностью, без ошибок и недочетов, с указанием всех необходимых свойств, законов, пояснений; схемы, графики, диаграммы выполнены точно; сделаны необходимое выводы.

Оценка "4" (хорошо) ставится, если работа студента от основным требованиям к работе на оценку "5" но в ней допущены одна ошибка или не более двух недочётов допущены ошибки в диаграммах: работа выполнена небрежно расчетных данных сделаны недостаточно полно.

Оценка "3" (удовлетворительно) ставится, если студент правильно понимает сущность изучаемых методов, понятий, теорем, законов, но в знаниях имеются пробелы, не мешающие выполнению основных требований, предусмотренных программой; если студент правильно выполнил не менее 2/3 всей работы или допустил не более одной грубой ошибки, не более трех негрубых ошибок, одной негрубой ошибки и трех недочетов, при наличии четырех-пяти недочетов.

Оценка "2" (неудовлетворительно) ставятся если студент выполнил менее 2/3 работы или допустил больше ошибок и недочетов, чем необходимо для "3"; не усвоил основные закономерности и понятия по курсу учебной дисциплины.

2. Основная литература

1. Иванова Г.С. Технология программирование: Учебник для вузов. М.: Изд-во МГТУ им. Н.Э. Баумана, 2020г.-320с.
2. Информатика: Учебник. – 3-е изд. перераб., /Под ред. Н.В. Макаровой. М.: Финансы и статистика, 2018г.

3. Орлов. В.В. Технологии разработки программных продуктов. СПб.: Питер, 2019-г.-437с.
4. Эрик Дж. Брауде. Технология разработки программного обеспечения. СПб.: Питер, 2020-г.-658с.
5. А.В. Рудаков Технология разработки программных продуктов. 2-е изд.-М.: изд. центр «Академия» 2019.г. 210с.

Дополнительная литература:

1. Стандарт IEEE по языку функционального моделирования - Синтаксис и семантика IDFF0. – Введ. 2016г. Нью-Йорк : IEEE, 2016г.
2. Стандарт IEEE по синтаксису и семантике языка концептуального моделирования IDEFIX(IDEF Object). - Введ. 2015г. Нью-Йорк : IEEE, 2015г.
3. Информационная технология – Процессы жизненного цикла программных средств. – Введ. 2013г. – Женева: ISO/IEC,2013г.
4. Системная и программная инженерия – Процессы жизненного цикла программных средств. – Введ. 2013г. – Нью-Йорк: ISO/IEC- IEEE,2013г.
5. Информационная технология – Оценка программного продукта – Ч.1: Общий обзор.
6. Программная инженерия – Качество продукта Ч.1: Модель качества. – Введ. 2014г. Женева: ISO/IEC,2014г.
7. Информационная технология. Классификация программных средств. – Введ. 2017г. М.: Издательство стандартов, 2016.
8. Информационная технология. Руководство по применению ГОСТ Р ИСО/МЭК 12207 (Процессы жизненного цикла программных средств). -М.: Изд-во стандартов, 2014г.
9. Информационные технологии. Процессы жизненного цикла программных средств. Минск: Госстандарт Респ. Беларусь, 2015.г
10. Информационные технологии. Оценки программной продукции. Характеристики качества и руководство по их – применению. – Минск: Госстандарт Респ. Беларусь, 2016.г
11. Информационные технологии поддержки жизненного цикла продукции. Методология функционального моделирование. Рекомендации по стандартизации. – Введ . 2016г.