ТЕМА 1. КЛАССИФИКАЦИЯ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ.

Программное обеспечение — это код или набор инструкций, которые сообщают компьютеру или оборудованию, как работать. Программное обеспечение обычно является универсальным, но оно также может быть создано на заказ. Универсальное программное обеспечение открыто для рынка, и его технические характеристики разрабатываются программистом. В основном предназначен для широкого потребительского рынка. Индивидуальное программное обеспечение — это программное обеспечение, технические характеристики которого разработаны в соответствии с требованиями конкретной фирмы или организации. Она не открыта для всех. В основном предназначен для конкретных бизнес-целей. Программное обеспечение в основном подразделяется на семь категорий – Системное программное обеспечение, Прикладное программное обеспечение, инженерное/научное программное обеспечение, встроенное программное обеспечение, программное обеспечение линейки продуктов, веб-приложения и программное обеспечение для искусственного интеллекта.

Далее мы подробно обсудим различные классификации программного обеспечения.

1. Системное программное обеспечение

Он напрямую взаимодействует с компьютерным оборудованием. В первую очередь это касалось эффективного управления компьютерной системой. Он используется для разработки новых системных программ, и с помощью начальной загрузки мы можем сделать их переносимыми. Это зависит от машины. Системное программное обеспечение далее классифицируется на три категории – операционная система, которая действует как интерфейс между пользователем и оборудованием и предоставляет различные услуги пользователям. Второе-это программное обеспечение для поддержки системы, которое более эффективно управляет оборудованием. Другой-программное обеспечение для разработки систем, которое поддерживает среду разработки программ для пользователя.

2. Прикладное программное обеспечение

Он предназначен для решения пользовательских проблем в соответствии с требованиями пользователя. Прикладное программное обеспечение может быть универсальным или настраиваемым. Прикладное программное обеспечение далее подразделяется на две категории – одно из них является программным обеспечением общего назначения, которое используется для большого числа задач и предоставляет множество функций. Другое - программное обеспечение специального назначения, предназначенное только для определенной цели. Например, пользовательские программы. Основное внимание уделяется

приложению, а не компьютерной системе. В первую очередь это касается решения некоторых задач с использованием компьютера в качестве инструмента.

3. Инженерное/Научное программное обеспечение

Он имеет дело с требованиями к обработке в конкретном приложении. Это программное обеспечение специально используется для рисования, моделирования, составления чертежей, расчета нагрузок и анализа инженерных и статистических данных для интерпретации и принятия решений. Например, CAD (Автоматизированное проектирование), CAM (Автоматизированное производство) и CAE (Автоматизированное проектирование). Это программное обеспечение используется в области механики, электротехники, черчения, проектирования и анализа. Они работают на мэйнфреймах, рабочих станциях общего назначения и ПК (персональных компьютерах).

4. Встроенное программное обеспечение

Это программное обеспечение встроено в аппаратное обеспечение как часть более крупных систем для управления различными функциями. Этот тип программного обеспечения встроен в ПЗУ (только для чтения) систем. Например, программное обеспечение для управления клавиатурой, ПО встроенное в микроволновую печь или стиральную машину, где необходимо выполнить входной анализ, принять решение и предпринять действия, которые позволят продукту работать требуемым образом. Эти программные продукты также называются интеллектуальным программным обеспечением из-за его производительности.

5. Программное обеспечение Линейки Продуктов

Этот тип программного обеспечения относится к методам, инструментам и методам разработки программного обеспечения для создания набора аналогичных программных систем из общего набора программных ресурсов с использованием общих средств производства. Это набор программных продуктов, которые имеют общие функции, но каждый из них в чем-то отличается. Например, они могут быть разработаны для конкретного заказчика или для встроенного программного обеспечения (документ Word, электронные таблицы, компьютерная графика, персональные и бизнес-приложения).

6. Веб-Приложения

Это приложение, доступ к которому осуществляется через веб-браузеры по сети, такой как Интернет или интрасеть. это также компьютерное программное приложение, которое закодировано на языке поддержки браузера и надежно в обычном веб-браузере для выполнения приложения. Первое поколение веб-приложений позволяет бизнесу публиковать информацию публично. Таким образом, эта информация доступна любому пользователю с веб-браузером и доступом в Интернет. Проблема первого поколения заключается в том, что информация находится в статической форме. Веб-приложения второго поколения позволяют пользователям выполнять интерактивные запросы к базам данных из

веб-приложения. Он характеризуется как облегчающий общение, обмен информацией, ориентированный на пользователей и сотрудничество в Интернете. Приложение третьего поколения более полезно, чем приложение второго поколения. В сочетании с запросами приложения второго поколения и статической информацией первого поколения, третье поколение является мощным бизнес-инструментом для организаций в их усилиях в области электронной торговли.

7. Программное обеспечение для Искусственного Интеллекта

Это программное обеспечение использует нечисловые алгоритмы, которые используют данные, полученные в системе, для решения сложных задач, которые не поддаются процедурам решения задач и требуют конкретного анализа и интерпретации проблемы для ее решения. Например, искусственные нейронные сети, программы для роботов, экспертные системы и компьютерные игры. Все это программное обеспечение может работать как в режиме реального времени, так и в автономном режиме.

Далее давайте рассмотрим модели разработки программного обеспечения.

Модели разработки программного обеспечения – это процедуры, которые определяют процесс и поток, в которых должен выполняться проект, определяют, как должно разрабатываться программное обеспечение на основе требований бизнеса и пользователей. Эти процедуры разработки помогают как разработчикам, так и тестировщикам правильно разработать проект. Существуют различные доступные модели, каждая из которых имеет свои преимущества и недостатки, в зависимости от проекта выбирается подходящая модель для работы.

Модели разработки программного обеспечения, которые используются при разработке программного обеспечения, следующие.

1. Модель водопада (Waterfall)

Модель водопада является пионером процесса жизненного цикла разработки программного обеспечения. Это была первая модель, которая использовалась при разработке программного обеспечения. Водопад разделен на различные фазы, где выход каждой фазы действует как вход для следующей фазы цикла. Прежде чем перейти к следующему этапу, обязательно завершите предыдущий этап, без завершения первого этапа мы не сможем перейти к следующему этапу. Поскольку фазы модели перемещаются с верхнего уровня на нижний, как водопад сверху вниз, она называется моделью «водопада».

Модель водопада работает следующим образом:

- Соберите требования пользователя там, где от мала до велика, все требования указаны правильно, чтобы понять требования пользователя к продукту

- После требований пользователя перечислены системные требования для работы с системой
- Исходя из требований, дизайн был разработан для продукта
- После глобального проектирования разрабатывается подробный дизайн продукта, включая систему, аппаратное обеспечение, программное обеспечение и т. Д.
- После полной проработки проекта фактическая реализация продукта начинается с формирования команды разработчиков проекта
- После внедрения продукта команда тестирования тестирует продукт в целом.

2. V Модель

V модель — это структура, которая используется для описания деятельности жизненного цикла разработки программного обеспечения от спецификаций требований до технического обслуживания. В модели V мероприятия по тестированию интегрированы в каждый этап жизненного цикла разработки программного обеспечения.

Уровни тестирования, используемые для модели V

- Тестирование компонентов: используется для проверки функций всех программных компонентов, используемых в продукте, работают хорошо или нет. Каждый компонент может быть протестирован отдельно.
- Интеграционное тестирование: используется для тестирования интерфейса между компонентами, т. е. взаимодействия с различными частями системы, такими как файловая система, операционная система, аппаратное обеспечение или интерфейсы между системами.
- Системное тестирование: используется для тестирования системы в целом. он проверяет систему или продукт на соответствие указанным требованиям.
- Приемочное тестирование: используется для выполнения валидационного тестирования в соответствии с потребностями пользователей, требованиями и бизнес-процессами, проводимыми для определения того, следует ли принимать систему или нет.

Эти уровни тестирования могут быть объединены и переупорядочены в зависимости от проекта. В модели V валидационное тестирование выполняется на более ранних стадиях жизненного цикла.

3. Итеративная Модель Разработки

Модель итеративной разработки — это жизненный цикл, в котором проект разбит на большое количество итераций, где итерация представляет собой полный цикл разработки, приводящий к выпуску исполняемых продуктов, т. е.

подмножества разрабатываемого конечного продукта, которое растет от итерации к итерации, чтобы стать конечным продуктом. Быстрая разработка приложений, гибкая разработка и Рациональный унифицированный процесс являются примерами итерационных моделей разработки.

4. Модель Быстрой Разработки Приложений (Rapid application development, RAD)

Быстрая разработка приложений формально представляет собой параллельную разработку функций и последующую интеграцию. Модель RAD позволяет пользователям в процессе разработки просматривать продукт.

RAD работает следующим образом

- Понимаются требования пользователей или организации заказчика.
- Разрабатывается дизайн продукта.
- Разрабатываются основные функции и компоненты проекта.
- Предоставляется уменьшенная версия продукта клиенту или пользователю для ежедневного использования.
- Собираются отзывы пользователей.
- Созданный продукт переосмысляется и в него вносятся коррективы, после делается обновление или перезапуск.

На основе обратной связи практически всегда разрабатывается другая версия проекта, и процесс идут по следующему «витку».

RAD позволяет быстро изменять и развивать продукт. Это позволяет на ранней стадии оценить технические риски и быстро реагировать на меняющиеся требования клиентов. В модели RAD пользователи получают раннюю версию продукта (альфа и бета версии) и могут предоставить отзывы о его дизайне, а также могут решить, основываясь на существующей функциональности, продолжать разработку или нет, какие функции необходимо включить в следующую поставку продукта и т.д. Пользователи также могут решить, останавливать проект или нет, если он не дает ожидаемых результатов.

5. Гибкая Модель Разработки (Agile)

Гибкая разработка — это группа методологий разработки программного обеспечения, основанных на итеративной инкрементной разработке, в которой требования и решения развиваются за счет сотрудничества между самоорганизующимися кросс-функциональными командами. существует ряд гибких методологий, доступных для использования при разработке программного обеспечения, но гибкая методология Scrum и экстремальное программирование, т.е. являются основными источниками гибкой разработки.

Преимущества гибкой разработки

- Гибкая разработка фокусируется на хорошем качестве кода и его работе.
- Тестирование выполняется на ранних стадиях жизненного цикла.
- Дизайн делается сразу, но потом может развиваться.
- Заинтересованные стороны бизнеса могут быть привлечены, чтобы помочь команде тестирования, и чтобы тестировщики могли проверить продукт на основе указанных требований от бизнес-сообщества.

Мы рассмотрели модели разработки программного обеспечения с различных сторон.

Как видно из этого анализа, программное обеспечение может быть разным, но отдельно здесь следует поговорить о веб-сервисах, так как в последние годы все больше проектов переносятся в сеть Интернет, программы для ПЭВМ оказываются уже не так нужны, если они не работают с физическими датчиками, платами расширения и другими специфическими элементами вычислительной машины.

И даже если приложение должно оставаться на компьютере, его можно разделить на две части: системный уровень (который и работает с оборудованием) и веб-интерфейс (веб-сервис), который работает в браузере и является веб-приложением, а сервером для него как раз и является первая часть приложения.

Что такое Веб-сервис?

Веб-служба — это согласованная среда, помогающая в эффективной коммуникации между клиентом и серверным приложением по сети. Это программный модуль, специально предназначенный для выполнения ряда задач. Клиент запрашивает у сервера вызов ряда веб-служб, и в ответ сервер размещает веб-службы. Запросы выполняются с помощью удаленных вызовов процедур (RPC). Наиболее важным элементом веб-службы является передача данных между клиентом и сервером в формате .xml. Многие языки программирования четко понимают этот формат. Точнее, приложения (хотя и написанные на разных языках программирования) взаимодействуют друг с другом в формате XML. Таким образом, он обеспечивает общую платформу для координации и работы программистов.

Веб – сервисы обеспечивают множество преимуществ-снижение затрат на связь, предоставление бизнес-функций в сети, наличие стандартизированного протокола, понятного всем, обеспечение взаимодействия между различными приложениями и создание сервиса для конкретной задачи. Веб-служба имеет сервис-ориентированную архитектуру и представляет собой комбинацию как внутренних, так и внешних сервисов для любой организации. Он способен производить и предоставлять необходимые услуги для клиента.

Типы веб-сервисов.

На техническом уровне веб-сервисы могут быть реализованы различными способами. Ниже перечислены два типа веб-сервисов.

SOAP

Первоначально разработанный корпорацией Майкрософт, он также известен как независимый от транспорта протокол обмена сообщениями. Он передает XML - данные в виде сообщений SOAP. Каждое сообщение содержит XML-документ. Документ следует определенному шаблону. Он использует HTML и SMTP для передачи сообщений. Веб-службы SOAP имеют стандарт безопасности и адреса. Они жестко закодированы и генерируются без использования репозитория. SOAP имеет встроенную систему обработки ошибок. Ответы на запросы содержат информацию об ошибках, которая помогает отслеживать и исправлять ошибки.

SOAP работает очень хорошо, когда он использует распределенные корпоративные настройки. Он обладает преимуществами автоматизации при использовании с несколькими языковыми продуктами (когда компоненты системы разработаны с применением разных технологий). Единственным серьезным недостатком является то, что он использует WSDL (Язык описания веб-служб) для обнаружения службы и не имеет другого альтернативного механизма.

Сообщение SOAP представляется в виде XML-файла и содержит следующие основные элементы:

<Envelope> - тэг, содержащий начало и конец сообщения. Это корневой элемент. Он включает в себя «заголовок» и «тело».

Заголовок содержит атрибуты, которые используются для обработки сообщения.

Тело содержит XML-данные, которые необходимо отправить.

Ошибка, которая выдает сообщения об ошибках при обработке данных.

REST

Передача состояния представления (REST) привлекает разработчиков, поскольку имеет более простой интерфейс, чем SOAP. Это архитектурный стиль (каждый URL-адрес представляет отдельный объект). Он обеспечивает связь и подключение между устройствами и Интернетом для задач на основе API. REST позволяет использовать различные форматы данных, такие как HTML, JSON, XML и т.д. По сравнению с SOAP, он потребляет меньше ресурсов и может использовать каналы с меньшей пропускной способностью. Веб-сервисы могут быть написаны на любом языке программирования и использоваться на любой платформе. Он также может использовать веб-службы SOAP, но SOAP не может использовать веб-службы REST.

Веб-службы REST обеспечивают гибкость приложений, созданных с использованием различных языков программирования и платформ, для эффективного взаимодействия. Теперь все последовательно перемещается в облако. Поэтому все облачные архитектуры разрабатываются и работают по принципу веб-сервисов REST.

Ключевыми элементами реализации REST являются:

- Ресурсы: они дает команду веб-серверу предоставлять требуемые данные.
- Глаголы запроса: они описывают все, что вы хотите сделать с конкретным ресурсом (требуемая информация).
- Заголовки запроса: дополнительные инструкции, отправляемые вместе с запросом; они определяют тип требуемого запроса.
- Тело запроса: тело запроса содержит необходимые сведения о конкретном ресурсе, который необходимо добавить на сервер.
- Тело ответа: это основная часть полученного ответа.
- Коды состояния ответа: это общие коды, которые возвращаются вместе с ответами, полученными с веб-сервера.

Каждая структура нуждается в какой-то архитектуре, чтобы гарантировать, что вся структура работает или функционирует по желанию. В архитектуре веб-служб существуют три различные роли:

- Поставщик услуг: поставщик — это тот, кто создает веб-службу и делает ее полностью доступной для клиентского приложения, которое хочет использовать ее для своих конкретных целей.
- Отправитель запроса на обслуживание: это клиентское приложение, необходимое для связи с веб-службой. Клиентским приложением может быть .Net, Java или любое другое приложение на основе языка программирования, которое ищет определенный вид функциональности через веб-службу.
- Service Broker: это приложение, которое предоставляет доступ к UDDI (Универсальное описание, обнаружение и интеграция), и это то, что обеспечивает хранилище, в котором могут размещаться различные файлы WSDL.

Все три элемента архитектуры веб-служб помогают находить, привязывать и публиковать веб-службу.

Современные веб-сервисы в настоящее время полностью изменили цифровой сценарий с развитием системной интеграции и взаимодействия. Они предлагают современный и в то же время менее сложный уровень функциональности. Все это говорит лишь о том, что веб-сервисы и спрос на них растут с ростом цифровизации во всех сферах торговли и бизнеса. Их

актуальность и важность не могут приниматься как должное и должны быть согласованы для понимания функциональности технологии и ее использования.

ТЕМА 2. АППАРАТНОЕ ОБЕСПЕЧЕНИЕ ИС И АС

Современные автоматизированные системы размещаются на самых разных платформах. Сама платформа зависит от цели на сферы применения системы.

АС и ИС общего назначения в основном размещаются на «бытовых» ПЭВМ, размещаемых в корпусах разного вида и мощности, но АС специального назначения могут размещаться совершенно в разных местах. Например:

- АС, применимые в промышленности, нередко «прошиваются» полностью или частично в промышленные контроллеры с повышенной защитой и устойчивостью к воздействиям.

- АС, предназначенные для применения в составе мобильных или носимых комплексов должны размещаться на маломощных и энергоэффективных платформах, имеющих низкое энергопотребление и рассеиваемую тепловую мощность.

Собственно говоря, от платформы размещения зависит и принцип реализации автоматизированной системы или ее отдельных частей.

АС и ИС могут быть составными, то есть часть системы может размещаться на ПЭВМ общего назначения (например, веб-интерфейсы и удаленные терминалы), а часть должна размещаться на разносортных мобильных или маломощных системах.

Если на ПЭВМ общего назначения можно использовать практически любые языки программирования и фреймворки, то большинство специализированных платформ «жестко» привязаны к определенным технологиям.

Конечно, можно на всех платформах использовать язык Си и его производные, так как это уникальный кроссплатформенный язык общего назначения. Для него действует только одно правило движения между платформами: если создан компилятор или кросс-компилятор на целевую платформу, - то можно переносить программу практически «один к одному».

Конечно, нужно помнить, что есть программный код, который не зависит от платформы (от типа и архитектуры микропроцессора), а есть код, который сильно зависит от платформы.

Сейчас язык Си присутствует практически на 99% доступных платформ, которые можно программировать. Здесь имеются в виду программируемые логические интегральные схемы (ПЛИС), для которых «программа» заменяется «конфигурацией», выполняемой либо в визуальном режиме, либо в виде описаний на специальных языках типа AHDL, VHAL, Verilog HDL. Причем первый язык является специальным языком фирмы ALTERA и не «работает» на ПЛИС других фирм, а вот второй и третий языки описания являются «кроссплатформенными»,

Однако сейчас редко можно найти «чистые» АС, разработанные исключительно в рамках одной и только одной технологии. Обычно в состав автоматизированной системы могут входить:

- датчики (активные и пассивные)

- исполнительные механизмы
- микрокомпьютеры
- ЭВМ общего назначения
- специальные платы для обработки сигналов
- специальные платы для поддержки искусственного интеллекта
- сетевые платформы
- серверные системы и т.д.

Для того, чтобы «вывести сигналы датчиков в сеть», применяют сетевые технологии, формируют подключения, создают клиент-серверные конфигурации. И вот тут нам нужен целый комплекс программных и аппаратных технологий с различной базой языковых средств и инструментов.

Представим себе систему типа «умной теплицы». Очевидно, что нам нужна система датчиков температуры, влажности и положения окон, ряд механизмов для управления климатическим оборудованием, локальные передающие устройства, система сбора информации с датчиков, сетевые устройства, сервер и ряд клиентов для приема данных и управления.

На сервере можно использовать любую технологию общего назначения для написания ПО, так как микропроцессор там классической архитектуры (Java, Python, Ruby on Rails, JavaScript, Golang, PHP и т.д.).

На сетевых устройствах скорее всего уже находятся свои собственные «прошивки» на языках типа Си или вообще на базе аппаратных конфигураций.

На устройствах сбора данных с датчиков скорее всего будет Си и Ассемблер, а на клиентском оборудовании также может быть «все, что угодно».

Таким образом, мы понимаем, что в большой, распределенной и сложной автоматизированной и информационной системе может быть использовано сразу несколько технологий и такую конфигурацию сложно разработать одной командой.

Давайте кратко рассмотрим принципы работы различных платформ: от платформ общего назначения до специализированных микрокомпьютеров и дадим им краткую характеристику.

Аппаратные платформы общего назначения

Аппаратные платформы общего назначения в основном имеют одну общую платформу с архитектурой Фоне Неймана и микропроцессоры фирм Intel или AMD.

Сейчас разрабатывается и активно распространяется отечественная платформа Эльбрус, разрабатываются для этих микропроцессоров целые сервера и персональные ЭВМ.

Более подробно механизмы выполнения программ нужно рассматривать в рамках других специальных курсов, но мы можем здесь посмотреть сами принципы запуска приложений на классической ЭВМ.

В любом случае, классическая ЭВМ работает по следующему принципу, диктуемому фоннеймановской архитектурой (в которой в состав ЭВМ входит сам микропроцессор, оперативная память, внешние устройства и системная шина, разделяемая на шину адреса, шину данных и шину управления).

Программы или исполняемые коды пользователя или операционной системы хранятся на жестком диске или нас сетевом хранилище.

После инициации процесса запуска они загружаются полностью или частично в оперативную память и там, в зависимости от типа организации памяти (сегментный или страничный механизм), программам, данным и стеку выделяется область памяти. При необходимости дополнительная информация может быть загружена с диска и переразмещена в оперативной памяти (ОП) или, наоборот, выгружена из ОП на внешнее хранилище.

После загрузки, начальный адрес сегмента кода или кодовой страницы передается в микропроцессор и тот начинает загружать исполняемые коды в свои внутренние буферы. Затем команды читаются ядром микропроцессора и встроенный дешифратор команд распознает коды операций, записанные в командах вместе с адресами и/или данными программы.

Если код операции распознан удачно, то микропроцессор подготавливает данные, загружая их во внутренние регистры. Затем, ядро передает код на Устройство управления, которое вырабатывает управляющие сигналы на исполнительные устройства типа контроллеров, портов, шину или память.

Эти сигналы включают тот или иной режим работы в исполнительных устройствах, и они выполняют определенные действия, заложенные в них разработчиками.

Результаты выполненных операций выводятся обратно из ядра согласно программе.

Таким образом, сама платформа диктует как будет выполняться программа и какие алгоритмы поддерживаются аппаратно.

Кроме фоннеймановской архитектуры существуют еще и другие, например Гарвардская, в которой уже две различные шины для кодов программ и для данных и две соответствующие области памяти, то есть код программы и данные для обработки закачиваются в микропроцессор уже не последовательно друг за другом, а параллельно.

«Другие» микропроцессорные ядра обладают рядом достоинств для решения «других» задач и имеют дополнительные блоки для аппаратного ускорения специфических операций.

Например, микроконтроллеры созданы для контроля и управления устройствами и для проведения операций за 1 такт синхросигнала благодаря, в частности, большому количеству внутренних регистров общего назначения (более 30 против 4-8 в классических микропроцессоров) и большому количеству специальных устройств, перемещенных с системной платы внутрь ядра (контроллер прерываний, таймеры и счетчики, порты ввода вывода и т.п.).

Микроконтроллеры

Микроконтроллеры – это специализированные микропроцессоры, построенные в основном по RISC-принципу. RISC – это reduce instructions set computer, то есть вычислитель с сокращенным набором команд. В микроконтроллерах их около сотни против тысячи в микропроцессорах ЭВМ

общего назначения. Последние строятся по CISC-принципу, то есть с полным набором команд (common instructions set computer).

Микроконтроллер – это однокристальная микроЭВМ, то есть вычислительная машина внутри одной микросхемы. Первый микроконтроллер был разработан фирмой Texas Instruments еще в 1971 г. Дальнейшее развитие было внутри компаний Intel, Motorola и т.д. Например в 1980 г. Intel выпустила первый экземпляр легендарного микроконтроллера i8051. Он получился настолько удачным, что микроконтроллеры на его основе выпускаются до сих пор. Микроконтроллеры разрабатывались и в СССР (например, 16-разрядные микроЭВМ К1801ВЕ1 и т.п. устройства).

Настоящий прорыв в этой области был с 2000-х г., когда на рынок вышли такие компании как Atmel, PIC, STM. Вскоре появилась архитектура ARM, которая сейчас является основной в мобильных устройствах и даже применяется в настольных компьютерах.

При проектировании микроконтроллеров приходится соблюдать компромисс между размерами и стоимостью с одной стороны и гибкостью, и производительностью с другой. Для разных приложений оптимальное соотношение этих и других параметров может различаться очень сильно. Поэтому существует огромное количество типов микроконтроллеров, различающихся архитектурой процессорного модуля, размером и типом встроенной памяти, набором периферийных устройств, типом корпуса и т. д.

В отличие от обычных компьютерных микропроцессоров, в микроконтроллерах часто используется гарвардская архитектура памяти, то есть раздельное хранение данных в ОЗУ, а команд — в ПЗУ.

Кроме ОЗУ, микроконтроллер может иметь встроенную энергонезависимую память для хранения программы и данных. Многие модели контроллеров вообще не имеют шин для подключения внешней памяти.

Наиболее дешёвые типы памяти допускают лишь однократную запись, либо хранимая программа записывается в кристалл на этапе изготовления (конфигурацией набора технологических масок). Такие устройства подходят для массового производства в тех случаях, когда программа контроллера не будет обновляться. Другие модификации контроллеров обладают возможностью многократной перезаписи программы в энергонезависимой памяти.

Неполный список периферийных устройств, которые могут использоваться в микроконтроллерах, включает в себя:

- универсальные цифровые порты, которые можно настраивать как на ввод, так и на вывод, причем можно использовать как весь 8-ми битный порт целиком, так и его отдельные «ножки»;
- различные интерфейсы ввода-вывода, такие, как UART, I^2C, SPI, CAN, USB, IEEE 1394, Ethernet (так называемые аппаратные порты);
- аналого-цифровые и цифро-аналоговые преобразователи;
- компараторы;
- широтно-импульсные модуляторы (ШИМ-контроллер);
- таймеры;

- контроллеры бесколлекторных двигателей, в том числе шаговых;
- контроллеры дисплеев и клавиатур;
- радиочастотные приемники и передатчики;
- массивы встроенной флеш-памяти;
- встроенные тактовый генератор и сторожевой таймер;

Ограничения по цене и энергопотреблению ограничивает тактовую частоту контроллеров. Хотя производители стремятся обеспечить работу своих изделий на высоких частотах, они, в то же время, предоставляют заказчикам выбор, выпуская модификации, рассчитанные на разные частоты и напряжения питания. Во многих моделях микроконтроллеров используется статическая память для ОЗУ и внутренних регистров. Это даёт контроллеру возможность работать на меньших частотах и даже не терять данные при полной остановке тактового генератора. Часто предусмотрены различные режимы энергосбережения, в которых отключается часть периферийных устройств и вычислительный модуль.

Сейчас наиболее известны следующие семейства микроконтроллеров:

- AVR (от компании Atmel)
  - ATMega
  - ATtiny
  - XMega
- PIC
- STM
- RL
- MCS
- ESP
- MSP
- ARM
  - ST Microelectronics STM32 ARM-based MCUs
  - ARM Cortex, ARM7 и ARM9-based MCUs
  - Texas Instruments Stellaris MCUs
  - NXP ARM-based LPC MCUs
  - Toshiba ARM-based MCUs
  - Analog Devices ARM7-based MCUs
  - Cirrus Logic ARM7-based MCUs
  - Freescale Semiconductor ARM9-based MCUs
  - Silicon Labs EFM32 ARM-based MCUs

В общем сейчас микроконтроллеры в том или ином виде применяются практически везде: в датчиках, в портативных приборах, в бытовой электронике, в «умных домах», в Интернете вещей и т.д.

Устройство с низкой стоимостью и весьма малыми размерами можно применять везде, где нужна низкая стоимость, малая рассеиваемая мощность, малые габариты и предсказуемость работы.

Например, микроконтроллер ATtiny10 с 8-битной архитектурой, частотой выполнения инструкций в 12 МГц, системой защиты от «зависаний», АЦП (преобразование из аналогового сигнала в цифровой) и ЦАП (обратно из

цифровых данных в аналоговый сигнал), напряжением питания от 1.8 до 5.5 В, спокойно можно поставить в «умный» датчик – сейчас стоит около 60 руб. за штуку оптом.

А микроконтроллер STM32F042G6U6, являющийся уже 32-битным ARM процессором (на архитектуре ARM Cortex-M0) с напряжением питания от 2 до 3.6 В, с 24 программируемыми ножками, встроенными аппаратными интерфейсами CAN, I2C, SPI, USART, USB, девятью (!!!) встроенными таймерами, 12-битным 10-канальным АЦП, 14 типами поддерживаемых сенсорных каналов для сенсорных или линейных датчиков поворота, встроенным на уровне кристалла отладчиком и частотой работы до 48 МГц – сейчас стоит всего 130 руб. за штуку.

Естественно, что такие микропроцессоры изготавливаются и внедряются миллионами штук, создано огромное количество книг и руководств, а также десятки фреймворков и инструментов быстрого прототипирования.

Здесь имеются в виду отладочные платы от различных фирм.

Самыми известными сейчас является проект Arduino, с которым работают даже школьники.

По сути, Arduino — торговая марка аппаратно-программных средств для построения и прототипирования простых систем, моделей и экспериментов в области электроники, автоматики, автоматизации процессов и робототехники.

Программная часть состоит из бесплатной программной оболочки (IDE) для написания программ, их компиляции и программирования аппаратуры. Аппаратная часть представляет собой набор смонтированных печатных плат, продающихся как официальным производителем, так и сторонними производителями. Полностью открытая архитектура системы позволяет свободно копировать или дополнять линейку продукции Arduino.

Используется как для создания автономных объектов, так и подключения к программному обеспечению через проводные и беспроводные интерфейсы. Подходит для начинающих пользователей с минимальным входным порогом знаний в области разработки электроники и программирования.

Программирование ведется целиком через собственную бесплатную программную оболочку Arduino IDE. В этой оболочке имеется текстовый редактор, менеджер проектов, препроцессор, компилятор и инструменты для загрузки программы в микроконтроллер. Оболочка написана на Java на основе проекта Processing, работает под Windows, Mac OS X и Linux. Используется также и собственный комплект библиотек Arduino.

Язык программирования Arduino называется Arduino C и представляет собой язык C++ с фреймворком Wiring.

Он имеет некоторые отличия по части написания кода, который компилируется и собирается с помощью avr-gcc, с особенностями, облегчающими написание работающей программы — имеется набор библиотек, включающий в себя функции и объекты.

Менеджер проекта Arduino IDE имеет нестандартный механизм добавления библиотек. Библиотеки в виде исходных текстов на стандартном C++ добавляются в специальную папку в рабочем каталоге IDE. При этом название

библиотеки добавляется в список библиотек в меню IDE. Программист отмечает нужные библиотеки, и они вносятся в список компиляции.

Arduino IDE не предлагает никаких настроек компилятора и минимизирует другие настройки, что упрощает начало работы для новичков и уменьшает риск возникновения проблем.

Однако этот факт также порождает критику платформы, так как разработчик «на Arduino» может совершенно не знать, что у микроконтроллера «глубоко внутри» (за него много работы делает сама система).

Закачка программы в микроконтроллер Arduino происходит через предварительно запрограммированный специальный загрузчик (все микроконтроллеры от Ардуино продаются с этим загрузчиком). Загрузчик создан на основе Atmel AVR Application Note AN109. Загрузчик может работать через интерфейсы RS-232, USB или Ethernet в зависимости от состава периферии конкретной процессорной платы.

Пользователь может самостоятельно запрограммировать загрузчик в чистый микроконтроллер. Для этого в IDE интегрирована поддержка программатора на основе проекта AVRDude. Поддерживается несколько типов популярных дешёвых программаторов. И это помогает распространять проект даже на те микроконтроллеры, которые изначально не предназначались для размещения платформы.

Интересный факт: распространенность и простота платформы позволяет разрабатывать собственные решения на ее основе и ярким примером таких систем являются графические языки программирования:

- Minibloq;
- Scratch for Arduino;
- Snap4Arduino;
- Ardublock;
- Ardublock Kode;
- Productivity Blocks — дополнение к Arduino IDE от компании AutomationDirect с графическим языком программирования и набором библиотек промышленной автоматики.
- Modkit;
- FLProg — бесплатный, позволяет создавать программное обеспечение на промышленных логических языках программирования — FBD и LAD.
- XOD — графический язык программирования Ардуино и Raspberry Pi с открытым исходным кодом.

Такие системы позволяют «рисовать» программу в виде блоков, соединяя их линиями связей. Именно этот вариант начинают изучать еще в детстве или в школе начинающие разработчики.

Промышленные контроллеры

Промышленные микроконтроллеры отличаются от обычных по многим характеристикам. С самого начала следует отметить, что они имеют более

высокую надежность, устойчивость к внешним эксплуатационным условиям, имеют специальные встроенные модули, ну и конечно более высокую цену.

Например, промышленные контроллеры DevLink®-C1000.

Это контроллеры российского производства DevLink-C1000 предназначены для создания «легких» и «средних» АСУ ТП, а также могут применяться в составе больших, сложных систем.

Универсальный свободно программируемый промышленный контроллер DevLink-C1000 осуществляет информационный обмен с верхним уровнем по общепринятым протоколам и стандартам (MODBUS, OPC, МЭК 60870-5-104, МЭК 60870-5-101).

В сочетании с модулями ввода-вывода DevLink-A10 контроллер DevLink-C1000 способен опрашивать различные датчики и приборы (термопары, термосопротивления, приборы с унифицированным токовым выходом и т.д.) и формировать управляющие воздействия. ПЛК способен опрашивать множество различных приборов и считывать архивы.

Высокопроизводительный 32-разрядный процессор на базе архитектуры ARM9 (с частотой 400 Мгц) в сочетании с быстрой памятью и специально оптимизированной под данную платформу системой реального времени контроллера (СРВК) позволяют достичь высокого уровня быстродействия.

Этот промышленный контроллер и подобные ему, имеют следующие основные функции:

- Сбор данных с контрольно-измерительных приборов.
- Контроль в режиме реального времени параметров системы (контроль нормативных значений).
- Резервирование контроллеров в системе и одиночный режим работы.
- Анализ в реальном времени значений параметров, полученных с приборов.
- Формирование и инициативная передача сообщений при определении аварийной ситуации на верхний уровень.
- Передача данных на верхний уровень по расписанию.
- Ведение архивов, доступных для передачи на верхний уровень.
- Выдача управляющих воздействий на исполнительные механизмы.
- Автоматическое регулирование параметров.
- Выполнение алгоритмов пользователя.
- Учет тепла, газа и других энергоносителей в связке с другими специальными приборами.
- Выполнение алгоритмов пользователя, разработанных на специальных языках программирования или языках общего назначения.

Если разрабатываемая АС или ИС относится к классу специализированных или имеет особые условия эксплуатации, то приходится использовать именно такой класс приборов.

Однако, если необходимо сочетать мощь вычислительной техники и возможности работы со специальными аппаратными устройствами, то можно использовать «одноплатники» или микрокомпьютеры.

Микрокомпьютеры

«Микрокомпьютер» — это термин, изначально обозначавший компьютер, выполненный на основе микропроцессора. В таком значении употреблялся с конца 1970-х до конца 1980-х (тогда же популярным было название «микроЭВМ»). Термин вышел из употребления в 1990-е годы, когда был вытеснен термином «персональный компьютер», так как размер таких компьютеров стал считаться обычным.

С начала 2010-х употребляется в ином значении, а именно снова вошёл в употребление в связи с появившейся популярностью компьютеров, размером с банковскую карту и сопоставимых по мощности с персональными компьютерами предыдущих поколений. Также микрокомпьютерами иногда называют микроконтроллеры — устройства, которые заключают в одной микросхеме все составляющие компьютера (процессор, оперативная и постоянная память, видеопамять, порты данных и др.).

Современные микрокомпьютеры с 2010 г. разрабатываются как миниатюрные одноплатные компьютеры общего назначения с малым энергопотреблением и открытой ОС, наподобие Raspberry Pi. Как правило, они основаны на архитектуре ARM, несовместимой с IBM PC, по возможностям/производительности они ближе всего к планшетам/смартфонам без экрана, но с HDMI-видеовыходом. Предполагаемое назначение таких компьютеров — учебные ПК, АРМы, медиацентры, домашние серверы, управляющие компьютеры в различных хобби-проектах.

Особенности:

- Миниатюрная конструкция, рассчитанная на установку в малогабаритный корпус или в подходящей нише.
- Низкое энергопотребление, позволяющее держать компьютер постоянно включённым или запитывать от небольшого аккумулятора.
- Как следствие такие вычислительные системы довольствуется пассивным охлаждением, в них нет движущихся частей системы охлаждения.
- Бывают как x86-совместимыми, так и нет.
- Низкая цена всей системы, по сравнению с системным блоком ПК.
- Легковесная ОС (как правило, основанная на открытой ОС с ядром Linux).
- Стандартные разъёмы для компьютерной периферии: USB, Secure Digital, eSATA, Ethernet, HDMI (совместим с DVI, также есть возможность подключения к разъёму VGA через специальные переходники).

Самым известным современным проектом в этом плане является Raspberry Pi.

Минимальная стоимость комплекта Raspberry Pi Pico сейчас составляет около 700 руб.

Основа платы — двухъядерный чип Arm Cortex M0+, с частотой работы ядра в 133 МГц. У платы — 264 КБ ОЗУ и 2 МБ флеш-памяти. Кроме того, есть

разъем USB 1.1 и I/O каналы, из которых пользователю доступны 26. Есть возможность задействовать интерфейсы 2 × UART, 2 × I2C, 2 × SPI (всего до 16 Мбайт QSPI Flash с XIP), а также 16 PWM-каналов. Также в наличии температурный датчик и 4 АЦП-канала.

Такой одноплатник уже годится как устройство сбора данных сдатчиков, а со следующим нашим «примером» можно уже полноценно работать как с персональной ЭВМ.

Самый дорогой Микрокомпьютер Raspberry Pi 4 Model B 8GB имеет цену в 12 тыс. руб, но представляет собой одноплатное компактное решение, которое при этом отличается высокой функциональностью и широкими возможностями использования. Данное устройство может выступать в качестве платформы для разработки программного обеспечения, эмулятора игровой приставки, а также медиацентра и просто рабочей станции.

В основе микрокомпьютера Raspberry Pi 4 Model B 8GB используется 4-ядерный процессор Broadcom BCM2711, работающий на частоте 1500 МГц, что вкупе с 8 ГБ оперативной памяти типа LPDDR4 может обеспечить комфортный уровень производительности для различных базовых задач. Для использования накопителей на плате предусмотрено множество интерфейсов периферии. Также есть полноформатные разъемы USB и HDMI 2.0, позволяющий выводить изображение в разрешении до 4K. Для доступа к сети модель поддерживает Wi-Fi и проводное подключение посредством Ethernet. Также для беспроводного соединения микрокомпьютер оснащен модулем Bluetooth. Устройство поставляется без операционной системы.

Следует также помнить, что сейчас разработаны десятки решений для расширения подобных устройств как Arduino или Raspberry Pi (они часто называются «шилдами»):

- Модули датчиков (температура, влажность, давление, расстояние, звук).
- Модули интерфейсов (проводные, беспроводные).
- Модули управления устройствами и двигателями.
- Силовые агрегаты.
- Экраны (7-сегментные, текстовые или графические).
- Клавиатуры, сенсоры, кнопки и джойстики.
- Отладчики и программаторы.
- Готовые роботизированные платформы (колесный, гусеничные, шагающие или «летающие» базы).

Все это делает возможным найти и применить в АС и ИС практически любое оптимальное по многим критериям решение.

ТЕМА 3. ПРИМЕНЕНИЕ АППАРАТНЫХ И ПРОГРАММНЫХ ПЛАТФОРМ ДЛЯ РЕШЕНИЯ ПРИКЛАДНЫХ И НАУЧНЫХ ЗАДАЧ

Комбинированные программно-аппаратные системы для решения исследовательских задач и задач моделирования.

Сначала мы рассмотрим классические инструменты решения задач при помощи программных и аппаратных инструментов, а затем и их аналоги. Начнем, пожалуй, с MATLAB.

MATLAB

MATLAB (сокращение от англ. «Matrix Laboratory», в русском языке произносится как Матлаб) — пакет прикладных программ для решения задач технических вычислений. Пакет используют более миллиона инженерных и научных работников, он работает на большинстве современных операционных систем, включая Linux, Windows и macOS.

MATLAB как язык программирования был разработан Кливом Моулером (англ. Cleve Moler) в конце 1970 гг. Вскоре новый язык распространился среди других университетов и был с большим интересом встречен учёными, работающими в области прикладной математики.

Язык MATLAB является высокоуровневым интерпретируемым языком программирования, включающим основанные на матрицах структуры данных, широкий спектр функций, интегрированную среду разработки, объектно-ориентированные возможности и интерфейсы к программам, написанным на других языках программирования.

Программы, написанные на MATLAB, бывают двух типов — функции и скрипты. Функции имеют входные и выходные аргументы, а также собственное рабочее пространство для хранения промежуточных результатов вычислений и переменных. Скрипты же используют общее рабочее пространство. Как скрипты, так и функции сохраняются в виде текстовых файлов и компилируются в машинный код динамически. Существует также возможность сохранять так называемые pre-parsed программы — функции и скрипты, обработанные в вид, удобный для машинного исполнения. В общем случае такие программы выполняются быстрее обычных, особенно если функция содержит команды построения графиков.

В MATLAB можно «рисовать» графические интерфейсы в виде панелей со стандартными элементами управления и графиками различных видов.

MATLAB может использоваться для моделирования 3D сцен и построения трехмерных моделей с использованием технологий VRML и OpenGL.

Но основной особенностью было и остается большое количество модулей расширения (toolboxes), - практически «на все случаи жизни».

MATLAB расширяется специальными функциями для решения задач в следующих сферах:

- Работа с матрицами и линейной алгеброй: алгебра матриц, линейные уравнения, собственные значения и векторы, сингулярности, факторизация матриц и другие.
- Работа с многочленами и интерполяцией/экстраполяцией: корни многочленов, операции над многочленами и их дифференцирование, интерполяция и экстраполяция кривых и другие.
- Решение задач математической статистики и анализа данных: статистические функции, статистическая регрессия, цифровая фильтрация, быстрое преобразование Фурье и другие.
- Обработка данных — набор специальных функций, включая построение графиков, оптимизацию, поиск нулей, численное интегрирование (в квадратурах) и другие.
- Решение дифференциальных уравнений: решение дифференциальных и дифференциально-алгебраических уравнений, дифференциальных уравнений с запаздыванием, уравнений с ограничениями, уравнений в частных производных и другие.
- Работа с разреженными матрицами: специальный класс данных пакета MATLAB, использующийся в специализированных приложениях.
- Работа с целочисленной арифметикой: выполнение операций целочисленной арифметики в среде MATLAB.

Также MATLAB имеет множество внешних интерфейсов для связи с другими языками и технологиями:

- COM-объекты и ActiveX.
- .NET.
- DDE-серверы.
- Веб-сервисы.
- Аппаратные интерфейсы компьютера.

Можно привести список инструментов MATLAB (тулбоксов), о которых мы говорили ранее:

- Цифровая обработка сигналов, изображений и данных: Signal Processing Toolbox (появился в 1987 г.), DSP System Toolbox, Image Processing Toolbox (появился в 1993 г.), Wavelet Toolbox, Communications System Toolbox — наборы функций и объектов, позволяющих решать широкий спектр задач обработки сигналов, изображений, проектирования цифровых фильтров и систем связи.
- Системы управления: Control Systems Toolbox, Robust Control Toolbox, System Identification Toolbox, Model Predictive Control Toolbox, Model-Based Calibration Toolbox — наборы функций и объектов, облегчающих анализ и синтез динамических систем, проектирование, моделирование и идентификацию систем управления, включая современные алгоритмы управления, такие как робастное управление, H∞-управление, ЛМН-синтез, μ-синтез и другие.
- Финансовый анализ: Econometrics Toolbox, Financial Instruments Toolbox, Financial Toolbox, Datafeed Toolbox, Trading Toolbox — наборы функций

и объектов, позволяющие быстро и эффективно собирать, обрабатывать и передавать различную финансовую информацию.

- Анализ и синтез географических карт, включая трёхмерные: Mapping Toolbox.
- Сбор и анализ экспериментальных данных: Data Acquisition Toolbox, Image Acquisition Toolbox, Instrument Control Toolbox, OPC Toolbox — наборы функций и объектов, позволяющих сохранять и обрабатывать данные, полученные в ходе экспериментов, в том числе в реальном времени. Поддерживается широкий спектр научного и инженерного измерительного оборудования.
- Визуализация и представление данных: Virtual Reality Toolbox — позволяет создавать интерактивные миры и визуализировать научную информацию с помощью технологий виртуальной реальности и языка VRML.
- Средства разработки: MATLAB Builder for COM, MATLAB Builder for Excel, MATLAB Builder for NET, MATLAB Compiler, HDL Coder — инструменты, позволяющие создавать независимые приложения из среды MATLAB.
- Взаимодействие с внешними программными продуктами: MATLAB Report Generator, Excel Link, Database Toolbox, MATLAB Web Server, Link for ModelSim — наборы функций, позволяющие сохранять данные различных видов таким образом, чтобы другие программы могли с ними работать.
- Базы данных: Database Toolbox — инструменты работы с базами данных.
- Научные и математические пакеты: Bioinformatics Toolbox, Curve Fitting Toolbox, Fixed-Point Toolbox, Optimization Toolbox, Global Optimization Toolbox, Partial Differential Equation Toolbox, Statistics And Machine Learning Toolbox, RF Toolbox — наборы специализированных математических функций и объектов, позволяющие решать широкий спектр научных и инженерных задач, включая разработку генетических алгоритмов, решения задач в частных производных, целочисленные проблемы, оптимизацию систем и другие.
- Нейронные сети: Neural Network Toolbox — инструменты для синтеза и анализа нейронных сетей.
- Нечёткая логика: Fuzzy Logic Toolbox — инструменты для построения и анализа нечётких множеств.
- Символьные вычисления: Symbolic Math Toolbox (появился в 1993 г.) — инструменты для символьных вычислений с возможностью взаимодействия с символьным процессором программы Maple.

Помимо вышеперечисленных, существуют тысячи других наборов инструментов для MATLAB, написанных другими компаниями и энтузиастами.

Все это уже делает MATLAB практически идеальным программным комплексом для научных вычислений и решения практических задач.

Альтернативные решения:

- GNU Octave.
- FreeMat.
- Maxima.
- Scilab.

Другие решения с «близкой» функциональностью:

- Julia
- R, S и SPlus.
- APL и его потомки.
- Python, при использовании пакета программ Python(x,y), а также с такими библиотеками как NumPy, SciPy и matplotlib реализует сходные возможности.
- IDL (англ. Interactive Data Language, интерактивный язык описания данных), - когда-то был коммерческим конкурентом MATLAB.
- Fortress, язык программирования, созданный Sun Microsystems, является наследником Фортрана, но с ним не совместим.

Далее рассмотрим «второй» компонент MATLAB, который все привыкли использовать либо вместе с основным MATLAB, либо вообще отдельно.

Это MATLAB Simulink.

Simulink — это графическая среда программирования на базе MATLAB для моделирования и анализа динамических систем.

Его основной интерфейс представляет собой графический инструмент построения блок-схем и настраиваемый набор библиотек блоков. Он обеспечивает тесную интеграцию с остальной средой MATLAB и может либо управлять ядром функций MATLAB, либо создаваться из MATLAB по сценарию.

Simulink широко используется в автоматическом управлении и цифровой обработке сигналов для моделирования и проектирования на основе моделей.

Здесь для нас особенно интересен такой момент: после разработки и отладки, схему Simulink можно загрузить в ПЛИС, получив код прошивки.

Этот компонент называется HDLCoder и он помогает генерировать коды описания аппаратуры на языках VHDL и Verilog.

HDL Coder предоставляет собой «ассистента по рабочим процессам», который автоматизирует программирование ПЛИС Xilinx, Microsemi и Intel. Мы можем управлять архитектурой HDL и конкретной реализацией каждой схемы, выделять критические пути и генерировать оценки использования аппаратных ресурсов. HDL Coder обеспечивает прослеживаемость между вашей моделью Simulink и сгенерированным кодом Verilog и VHDL, позволяя проверять код для приложений с высокой степенью целостности, соответствующих промышленным стандартам.

Единственным недостатком MATLAB с нашей точки зрения является его высокая стоимость. Конечно, у него есть и студенческие лицензии (около 100$ за базовое ядро плюс по 30 $ за каждый тулбокс), но на сайте mathworks.com вполне конкретно указана цена Стандартной лицензии в 940 $ в год или 2 350 $ за одну

бессрочную лицензию (и это без учета тулбоксов или Simulink). По некоторым подсчетам, одно рабочее место Matlab Simulink может стоить до 9 млн. руб., а лицензия для институтского кампуса – до 1 миллиона долларов США.

Не все могут позволить себе такие расходы, поэтому для решения математических и практических задач часто используют другие решения.

SIMINTECH

SimInTech (Simulation In Technic) — среда динамического моделирования технических систем, предназначенная для расчётной проверки работы систем управления сложными техническими объектами. SimInTech осуществляет моделирование технологических процессов, протекающих в различных объектах, с одновременным моделированием системы управления, и позволяет повысить качество проектирования систем управления за счет проверки принимаемых решений на любой стадии проекта.

Интересно то, что SimInTech разработан в ООО «3В Сервис», так что это наша отечественная разработка, уже давно признанная на мировом уровне.

SimInTech появился в 1994 г. и с тех пор «пишется» на языках Delphi, C, C++.

Он работает на ОС Windows/Linux и предназначен для детального исследования и анализа нестационарных процессов в ядерных и тепловых энергоустановках, в системах автоматического управления, в следящих приводах и роботах, и в любых технических системах, описание динамики которых может быть представлено в виде системы дифференциально-алгебраических уравнений и/или реализовано методами структурного моделирования. Основными направлениями использования SimInTech являются создание моделей, проектирование алгоритмов управления, их отладка на модели объекта, генерация исходного кода на языке Си для программируемых контроллеров.

Для SimInTech созданы и разрабатываются модули расширения позволяющие создавать модели на базе специализированных расчетных кодов и интегрировать их в комплексные модели и проекты.

SimInTech может:

- использоваться для моделирования нестационарных процессов в физике, в электротехнике, в динамике машин и механизмов, в астрономии и т. д., а также для решения нестационарных краевых задач (теплопроводность, гидродинамика и др.);
- функционировать в многокомпьютерных моделирующих комплексах, в том числе и в системах удаленного доступа к технологическим и информационным ресурсам;
- функционировать как САПР при групповой разработке и сопровождении жизненного цикла изделия (проекта) при модельно-ориентированном подходе к проектированию.

SimInTech содержит библиотеки типовых блоков для моделирования:

- теплогидравлики/пневматики;

- электроцепей, в действующих и мгновенных значениях;
- силовых машин гидравлических/пневматических;
- механических взаимодействий;
- точечной кинетики нейтронов;
- баллистики космических аппаратов;
- динамики полета летательных аппаратов в атмосфере;
- электрических приводов;

Для разработки алгоритмов управления в SimInTech есть общетехнические библиотеки блоков автоматики. Среди них библиотеки:

- конечных автоматов;
- релейной автоматики;
- нечеткой логики.

Кроме этого, SimInTech обладает:

- инструментами для создания интерфейсов управления (SCADA);
- библиотекой цифровой обработки сигналов;
- библиотекой статистики;
- функционалом оптимизации/подбора параметров;
- протоколами обмена (OPC, UDP, TCP/IP, MODBUS, RS, FMI и т.д.);
- функционалом распараллеливания расчетов на разных вычислительных узлах;
- модулем для верификации кода ПЛИС;
- модулем анализа надежности, безопасности и живучести системы на принципиальной схеме;
- библиотекой нейронных сетей;
- библиотекой видеообработки.

Так же, как и MATLAB, SimInTech может работать с аппаратурой так как содержит автоматический генератор кода для микроконтроллеров (он поддерживает язык ANSI C и ST.

Что более интересно, так это то, что к компьютеру с SimInTech мы можем подключить стандартную отладочную плату типа Arduino или Raspberry Pi, сгенерировать код для «модели», а потом «заменить» этой платой узел в схеме моделирования, загрузив в нее реальный код.

Эта возможность позволяет отнести SimInTech к классу систем с гибридным или полунатурным моделированием, при этом на SimInTech + реальные аппаратные платы можно смоделировать практически все, что угодно: от маятника до узла атомной электростанции.

SimInTech дешевле «конкурентов». Есть вообще бесплатная версия с ограничением на 250 блоков – для обучения более чем достаточно. А полная платная версия стоит в десятки раз меньше, чем MATLAB.

Ну а когда и эти возможности излишни, разработчики и исследователи используют альтернативные решения типа Python/Julia/R и платы типа Raspberry Pi.

Решение научных задач при помощи математических пакетов в языках программирования.

Мы уже говорили, что Python/Julia/R в некотором смысле являются альтернативой для математического моделирования, когда обсуждали MATLAB.

Все дело в том, что, например, Python чрезвычайно удобен для научных вычислений и много пакетов и библиотек разработаны именно для него.

Сейчас вообще идет «бум» на Python так как самым «прорывным» направлением разработки является Data Science или наука о данных или искусственный интеллект.

Сейчас мы просто перечислим библиотеки Python для научных вычислений:

NumPy (сокращенно от Numerical Python)— библиотека с открытым исходным кодом для языка программирования Python/

SciPy библиотека для языка программирования Python с открытым исходным кодом, предназначенная для выполнения научных и инженерных расчётов.

Statsmodels — это пакет Python, который позволяет пользователям изучать данные, оценивать статистические модели и выполнять статистические тесты. Обширный список описательной статистики, статистических тестов, функций построения графиков и статистики результатов доступен для различных типов данных и каждого оценщика. Он дополняет модуль статистики Scipy.

Scikit-learn (ранее scikits.learn, также известный как sklearn) - бесплатная библиотека машинного обучения для языка программирования Python. Он включает в себя различные алгоритмы классификации, регрессии и кластеризации, включая машины опорных векторов, случайные леса, повышение градиента, k-средние значения и DBSCAN, и предназначен для взаимодействия с числовыми и научными библиотеками Python NumPy и SciPy.

Pandas — программная библиотека на языке Python для обработки и анализа данных. Работа pandas с данными строится поверх библиотеки NumPy, являющейся инструментом более низкого уровня. Предоставляет специальные структуры данных и операции для манипулирования числовыми таблицами и временны́ми рядами. Название библиотеки происходит от эконометрического термина «панельные данные», используемого для описания многомерных структурированных наборов информации.

PyTorch — фреймворк машинного обучения для языка Python с открытым исходным кодом, созданный на базе Torch. Используется для решения различных задач: компьютерное зрение, обработка естественного языка. Разрабатывается преимущественно группой искусственного интеллекта Facebook. Также вокруг этого фреймворка выстроена экосистема, состоящая из различных библиотек, разрабатываемых сторонними командами: PyTorch Lightning и Fast.ai, упрощающие процесс обучения моделей, Pyro, модуль для вероятностного

программирования от компаний Uber и Flair, модуль для обработки естественного языка и Catalyst, а также модули для обучения DL и RL моделей.

PyTorch предоставляет две основные высокоуровневые модели:

- Тензорные вычисления (по аналогии с NumPy) с развитой поддержкой ускорения на GPU.
- Глубокие нейронные сети на базе системы autodiff.

TensorFlow — открытая программная библиотека для машинного обучения, разработанная компанией Google для решения задач построения и тренировки нейронной сети с целью автоматического нахождения и классификации образов, достигая качества человеческого восприятия. Применяется как для исследований, так и для разработки собственных продуктов Google. Основной API для работы с библиотекой реализован для Python, также существуют реализации для R, C Sharp, C++, Haskell, Java, Go и Swift.

Keras — открытая библиотека, написанная на языке Python и обеспечивающая взаимодействие с искусственными нейронными сетями. Она представляет собой надстройку над фреймворком TensorFlow. До версии 2.3 поддерживались разные версии нейросетевых библиотек, такие как TensorFlow, Microsoft Cognitive Toolkit, Deeplearning4j, и Theano. Нацелена на оперативную работу с сетями глубинного обучения, при этом спроектирована так, чтобы быть компактной, модульной и расширяемой.

Как видно из этого обзора, решений на Python просто «море» и можно найти решение практически для любой инженерной задачи. Тем более, что Python в «науке о данных» используют даже такие гиганты как Google и Facebook.

Отдельно нужно сказать про применение «чистой» аппаратуры в научных вычислениях.

Это можно понять если представить, что процессор вместо того, чтобы выполнять набор инструкций, будет перестраиваться под каждую программу и превращать алгоритм непосредственно в «железо».

То есть алгоритм преобразуется в описание конфигурации аппаратуры, а потом в часть «железа».

С помощью FPGA можно в буквальном смысле проектировать цифровые микросхемы, сидя у себя дома с доступной отладочной платой и софтом разработчика за пару тысяч рублей или долларов (если нужна большая мощность).

В результате работы получается физическая цифровая схема, выполняющая определенный алгоритм на аппаратном уровне, а не программа для процессора.

Микросхема FPGA — это та же заказная микросхема, состоящая из таких же транзисторов, из которых собираются триггеры, регистры, мультиплексоры и другие логические элементы для обычных схем (ну или элементов процессора). Изменить порядок соединения этих транзисторов, конечно, нельзя. Но архитектурно микросхема построена таким хитрым образом, что можно

изменять коммутацию сигналов между более крупными блоками: их называют CLB — программируемые логические блоки.

Также можно изменять логическую функцию, которую выполняет CLB. Достигается это за счет того, что вся микросхема пронизана ячейками конфигурационной памяти Static RAM. Каждый бит этой памяти либо управляет каким-то ключом коммутации сигналов, либо является частью таблицы истинности логической функции, которую реализует CLB.

Так как конфигурационная память построена по технологии Static RAM, то, во-первых, при включении питания FPGA микросхему обязательно надо сконфигурировать, а во-вторых, микросхему можно реконфигурировать практически бесконечное количество раз.

Блоки CLB находятся в коммутационной матрице, которая задает соединения входов и выходов блоков CLB.

На каждом пересечении проводников находится шесть переключающих ключей, управляемых своими ячейками конфигурационной памяти. Открывая одни и закрывая другие, можно обеспечить разную коммутацию сигналов между CLB.

CLB очень упрощенно состоит из блока, задающего булеву функцию от нескольких аргументов (она называется таблицей соответствия — Look Up Table, LUT) и триггера (flip-flop, FF). В современных FPGA LUT имеет шесть входов, но на рисунке для простоты показаны три. Выход LUT подается на выход CLB либо асинхронно (напрямую), либо синхронно (через триггер FF, работающий на системной тактовой частоте).

Значение каждой из ячеек подается на свой вход выходного мультиплексора LUT, а входные аргументы булевой функции используются для выбора того или иного значения функции. CLB — важнейший аппаратный ресурс FPGA. Количество CLB в современных кристаллах FPGA может быть разным и зависит от типа и емкости кристалла. У Xilinx есть кристаллы с количеством CLB в пределах примерно от четырех тысяч до трех миллионов.

Помимо CLB, внутри FPGA есть еще ряд важных аппаратных ресурсов. Например, аппаратные блоки умножения с накоплением или блоки DSP. Каждый из них может делать операции умножения и сложения 18-битных чисел каждый такт. В топовых кристаллах количество блоков DSP может превышать 6000.

Другой ресурс — это блоки внутренней памяти (Block RAM, BRAM). Каждый блок может хранить 2 Кбайт. Полная емкость такой памяти в зависимости от кристалла может достигать от 20 Кбайт до 20 Мбайт. Как и CLB, BRAM и DSP-блоки связаны коммутационной матрицей и пронизывают весь кристалл. Связывая блоки CLB, DSP и BRAM, можно получать весьма эффективные схемы обработки данных.

А теперь представьте себе, что всю эту мощь и разнообразие конструктивных элементов сформировать в определенную схему, например, реализующую операции с матрицами.

На реализацию одного вычислительного элемента пойдет всего несколько десятков или сотен вентилей. То есть, имея сотни тысяч или миллионы вентилей, можно разместить в ПЛИС десятки или даже тысячи вычислителей параллельно.

Это называется проблемно-ориентированной системой, которая великолепно справляется с ограниченным кругом задач и не имеет никаких лишних компонент.

Именно так работает MATLAB Simulink, который преобразует наши научный вычисления в аппаратные схемы.

ТЕМА 4. ПРОГРАММНЫЕ ИНТЕРФЕЙСЫ ИНФОРМАЦИОННЫХ И АВТОМАТИЗИРОВАННЫХ СИСТЕМ

В данной лекции мы рассмотрим понятие интерфейса программных систем на различных уровнях и как они применяются на практике.

В действительности можно назвать всего несколько типов программных «интерфейсов», посредством которых можно передавать данные между программами: это, пожалуй, обмен через общую память (кэш или базу данных), прямой вызов функций с передачей управления (например, вызов библиотечной функции) и данных, сигналы и сетевое взаимодействие.

Согласно классической литературе, программный интерфейс также определяется как способ, которыми одна компьютерная программа может взаимодействовать с другой программой.

Обмен через общую память

Обмен через общую память, один из самых «старых» способов построить интерфейс доступа в приложениях. Он базируется на принципе «разделяемой памяти».

Разделяемая память или Shared memory до сих пор является самым быстрым средством обмена данными между процессами или даже между аппаратными схемами.

Разделяемой памяти позволяет осуществлять обмен информацией через общий для процессов сегмент памяти без использования системных вызовов ядра. Сегмент или страница разделяемой памяти подключается в свободную часть виртуального адресного пространства процесса. Таким образом, два разных процесса могут иметь разные адреса одной и той же ячейки подключенной разделяемой памяти.

После создания разделяемого сегмента памяти любой из пользовательских процессов может подсоединить его к своему собственному виртуальному пространству и работать с ним, как с обычным сегментом памяти. Недостатком такого обмена информацией является отсутствие каких бы то ни было средств синхронизации, однако для преодоления этого недостатка можно использовать технику семафоров.

В схеме обмена данными между двумя процессами — (клиентом и сервером), использующими разделяемую память, — должна функционировать группа из двух семафоров. Первый семафор служит для блокирования доступа к разделяемой памяти, его разрешающий сигнал — 1, а запрещающий — 0. Второй семафор служит для сигнализации сервера о том, что клиент начал работу, при этом доступ к разделяемой памяти блокируется, и клиент читает данные из памяти. Теперь при вызове операции сервером его работа будет приостановлена до освобождения памяти клиентом.

Кратко можно описать механизм семафоров следующим образом (используя понятийную систему сетевого взаимодействия «клиент-сервер»):

- Сервер получает доступ к разделяемой памяти, используя семафор.
- Сервер производит запись данных в разделяемую память.
- После завершения записи данных сервер освобождает доступ к разделяемой памяти с помощью семафора.
- Клиент получает доступ к разделяемой памяти, запирая доступ к этой памяти для других процессов с помощью семафора.
- Клиент производит чтение данных из разделяемой памяти, а затем освобождает доступ к памяти с помощью семафора.

В программном обеспечении разделяемой памятью называют:

Метод межпроцессного взаимодействия (IPC), то есть способ обмена данными между программами, работающими одновременно. Один процесс создаёт область в оперативной памяти, которая может быть доступна для других процессов.

Метод экономии памяти, путём прямого обращения к тем исходным данным, которые при обычном подходе являются отдельными копиями исходных данных, вместо отображения виртуальной памяти или описанного метода. Такой подход обычно используется для разделяемых библиотек и для XIP.

Поскольку оба процесса могут получить доступ к общей области памяти как к обычной памяти, это очень быстрый способ связи.

Обмен данными через разделяемую память используется, например, для передачи изображений между приложением и X-сервером на Unix системах.

Стандарт открытых систем POSIX даже предоставляет стандартизированное API для работы с разделяемой памятью — POSIX Shared Memory. Одной из ключевых особенностей операционных систем семейства UNIX является механизм копирования процессов (системный вызов fork()), который позволяет создавать анонимные участки разделяемой памяти перед копированием процесса и наследовать их процессами-потомками. После копирования процесса разделяемая память будет доступна как родительскому, так и дочернему процессу.

В операционной системе Windows для создания разделяемой памяти используется функция CreateSharedMemory из пакета Win32-SDK. С другой стороны, возможно использование функций CreateFileMapping и MapViewOfFile из MSDN.

Следует также отметить, что процедуры разделяемой памяти также поддерживаются на уровне аппаратуры специальным контролером Прямого доступа к памяти ПДП или Direct Memory Access (DMA).

Процесс обмена данными между внешним устройством и памятью происходит без участия процессора и предназначен в основном для устройств, обменивающихся большими блоками данных с оперативной памятью. Инициатором обмена всегда выступает внешнее устройство. Процессор инициализирует контроллер DMA, и далее обмен выполняется под управлением контроллера. Если выбранный режим обмена не занимает всей пропускной способности шины, во время операций DMA процессор может продолжать работу.

Альтернативой всегда является программно-управляемый ввод-вывод.

Программно-управляемый ввод-вывод означает обмен данными с внешними устройствами с использованием команд процессора. Передача данных происходит через регистры процессора и при этом в конечном счете может реализовываться обмен, собственно, с процессором, обмен внешнего устройства с памятью, обмен между внешними устройствами.

Процессоры x86 имеют раздельную адресацию памяти и портов ввода-вывода и соответственно ввод-вывод может быть отображен либо в пространство ввода-вывода, либо вы пространство оперативной памяти (memory-mapped I/O). В последнем случае адрес памяти декодируется во внешнем устройстве и для выполнения ввода-вывода могут быть использованы все команды обращения к памяти.

Каждое адресуемый элемент адресного пространства ввода-вывода именуется портом ввода, портом вывода или портом ввода-вывода. Для обращения к портам предназначены четыре основные команды процессора: In (ввод в порт), Out (вывод из порта), Ins (ввод из порта в элемент строки памяти) и Outs (вывод элемента из строки памяти). Последние две строковые команды ввода-вывода используются для быстрой пересылки блоков данных между портом и памятью в случае последовательно расположенных адресов портов во внешнем устройстве. Обмен данными с портами, при котором используются строковые команды ввода-вывода, получил название PIO (Programmed Input/Output) - программированный ввод-вывод.

Собственно программно-управляемый обмен может инициироваться несколькими причинами:

- Процессором, точнее соответствующей командой в его программе. Эта ситуация подразумевает, что обмен данными является основной задачей процессора.
- Запросом аппаратного прерывания. Аппаратные прерывания вызываются внешними устройствами и теми компонентами компьютера, которые требуют немедленной обработки своей информации и приходят асинхронно по отношению к исполняемой программе. Прерывание можно рассматривать как некоторое особое событие в системе, которое заставляет процессор приостановить выполнение своей программы для реализации некоторой затребованной деятельности. Программные обработчики аппаратных прерываний инициализируют блочный обмен или выполняют одиночную операцию пересылки по системной шине с внешним устройством. Практически это основной способ инициализации обмена. Прерывания существенно увеличивают эффективность вычислительной системы, поскольку они позволяют внешним устройствам "обращать на себя внимание" процессора только по мере надобности.
- Возможно также и комплексное решение - опрос готовности одного или нескольких внешних устройств (polling) по периодическим прерываниям, например, от системного таймера. Готовое устройство

обслуживается, неготовое пропускается до следующего прерывания. Без анализа готовности возможно и периодическое выполнение каких-то действий с внешним устройством.

Вызов функций

Если говорить о программном интерфейсе между приложениями, как о работе с функциями, то мы можем также говорить о вызовах функций или процедур (понятие зависит от языка программирования).

Фу́нкция в программировании, или подпрограмма — фрагмент программного кода, к которому можно обратиться из другого места программы. В большинстве случаев с функцией связывается идентификатор, но есть так же безымянные и анонимные функции (лямбды, lambda).

С именем функции неразрывно связан адрес первой инструкции (оператора), входящей в функцию, которой передаётся управление при обращении к функции. После выполнения функции управление возвращается обратно в адрес возврата — точку программы, где данная функция была вызвана.

Функция может принимать параметры и должна возвращать некоторое значение, возможно пустое. Функции, которые возвращают пустое значение, часто называют процедурами. В некоторых языках программирования объявления функций и процедур имеют различный синтаксис, в частности, могут использоваться различные ключевые слова.

Функция должна быть соответствующим образом объявлена и определена. Объявление функции, кроме имени, содержит список имён и типов передаваемых параметров (или: аргументов), а также, тип возвращаемого функцией значения. Определение функции содержит исполняемый код функции. В одних языках программирования объявление функции непосредственно предваряет определение функции, в то время как в ряде других языков необходимо сначала объявить функцию, а уже потом привести её определение.

В объектно-ориентированном программировании функции, объявления которых являются неотъемлемой частью определения класса, называются методами. Также в языках с ООП возможно объявление абстрактной(виртуальной) функции без объявления тела функции.

Для того, чтобы использовать ранее определённую функцию, необходимо в требуемом месте программного кода указать имя функции и перечислить передаваемые в функцию параметры. Параметры, которые передаются функции, могут передаваться как по значению, так и по ссылке: для переменной, переданной по значению создаётся локальная копия и любые изменения, которые происходят в теле функции с переданной переменной, на самом деле, происходят с локальной копией и никак не сказываются на самой переменной, в то время как изменения, которые происходят в теле функции с переменной, переданной по ссылке, происходят с самой переданной переменной.

В любом случае, функция это не что иное, как просто область памяти (страница или сегмент), которая помечена как «исполняемая».

Соответственно вызов связан с загрузкой кода функции в оперативную память и переходом микропроцессора по адресу начала функции с передачей параметров через разделяемую память или стек.

В этом случае, правила и форматы вызова являются его интерфейсом.

Сигналы или сообщения ОС

Сигналы и слоты — подход, используемый в некоторых языках программирования и библиотеках (например, Qt) который позволяет реализовать шаблон «наблюдатель», минимизируя написание повторяющегося кода.

Концепция заключается в том, что компонент (часто виджет) может посылать сигналы, содержащие информацию о событии (например: был выделен текст «слово», была открыта вторая вкладка). В свою очередь другие компоненты могут принимать эти сигналы посредством специальных функций — слотов. Система сигналов и слотов хорошо подходит для описания графического интерфейса пользователя. Также механизм сигналов/слотов может быть применён для асинхронного ввода-вывода (включая сокеты, pipe, устройства с последовательным интерфейсом, др.) или уведомления о событиях.

Механизм сигналов и слотов типобезопасен. Сигнатура сигнала должна совпадать с сигнатурой слота-получателя. (Фактически слот может иметь более короткую сигнатуру чем сигнал, который он получает, так как он может игнорировать дополнительные аргументы). Так как сигнатуры сравнимы, компилятор может помочь нам обнаружить несовпадение типов. Сигналы и слоты слабо связаны. Класс, который вырабатывает сигнал, не знает и не заботится о том, какие слоты его получат.

Механизм сигналов и слотов Qt, например, гарантирует, что, если мы подключим сигнал к слоту, слот будет вызван с параметрами сигнала в нужное время. Сигналы и слоты могут принимать любое число аргументов любого типа. Они полностью типобезопасны.

Мы можем подключать к одному слоту столько сигналов, сколько захотим, также один сигнал может быть подключен к стольким слотам, сколько необходимо. Так же возможно подключать сигнал к другому сигналу (это вызовет выработку второго сигнала немедленно после появления первого). Сигналы и слоты вместе составляют мощный механизм создания компонентов.

На платформе .NET также есть похожий механизм. Он связан с понятиями Action / Invoke. Вкратце, определенные компоненты (программные коды) при определенных событиях могут отправлять сигналы на внутреннюю шину сообщений приложения. Также есть другие компоненты, которые «слушают» эту шину и при приеме соответствующего сообщения с подписью и параметрами, они выполняют соответствующие действия.

В веб-разработке на уровне фронтэнда тоже есть понятие экшенов (actions). Оно включено, например в концепцию Redux Actions в фреймворке React.

Даже в небольших JavaScript-приложениях, действий десятки, а то и сотни. В итоге появляется множество одинакового кода. Типичная работа программиста в этом случае сводится к тому, чтобы создать функцию, принимающую данные и

возвращающей объект действия. Actions - это константы, описывающие событие. Обычно это просто строка с названием описывающее событие.

Также, например в React есть хранилище Store и есть подписка на изменение хранилища subscribe. Данный метод subscribe принимает функцию, которая будет вызывается каждый раз после обновления store. Он как бы «подписывает» функцию, переданную ему на обновление.

Все эти функции являются «помощниками» программиста и позволяют структурировать код, разделить события и методы их обработки, а также обмениваться данными в строго определенном формате (посредством интерфейса).

Сетевое взаимодействие

Понятие программного интерфейса в этом контексте обычно входит в описание какого-либо интернет-протокола (например, SOAP, gRPC, RFC), программного каркаса (фреймворка) или стандарта вызовов функций операционной системы.

Часто реализуется отдельной программной библиотекой или сервисом операционной системы. Используется программистами при написании всевозможных приложений.

Также часто используется аббревиатура API или программный интерфейс приложения, интерфейс прикладного программирования.

API любого типа упрощает процесс программирования при создании приложений, абстрагируя базовую реализацию и предоставляя только объекты или действия, необходимые разработчику, ну или позволяя «разбить» одно монолитное приложение на массу взаимодействующих компонент.

Если графический интерфейс для почтового клиента может предоставить пользователю кнопку, которая выполнит все шаги для выборки и выделения новых писем, то API для ввода/вывода файлов может дать разработчику функцию, которая копирует файл из одного места в другое, не требуя от разработчика понимания операций файловой системы, происходящих за «кулисами».

API может служить инструментом интеграций приложений от различных разработчиков – именно на этом принципе сейчас базируются любые современные фреймворки – их разработчики предоставляют доступ извне к внутренним функциям.

Если программу (модуль, библиотеку) рассматривать как чёрный ящик, то API — это набор «ручек», которые доступны пользователю данного ящика и при помощи которых он может управлять действиями «ящика».

Программные компоненты взаимодействуют друг с другом посредством API. При этом обычно компоненты образуют иерархию — высокоуровневые компоненты используют API низкоуровневых компонентов, а те, в свою очередь, используют API ещё более низкоуровневых компонентов.

По такому принципу построены протоколы передачи данных по Интернету. Стандартный стек протоколов (сетевая модель OSI) содержит 7 уровней (от физического уровня передачи бит до уровня протоколов приложений, подобных

протоколам HTTP и IMAP). Каждый уровень пользуется функциональностью предыдущего («нижележащего») уровня передачи данных и, в свою очередь, предоставляет нужную функциональность следующему («вышележащему») уровню.

Понятие протокола близко по смыслу к понятию API. И то, и другое является абстракцией функциональности, только в первом случае речь идёт о передаче данных, а во втором — о взаимодействии приложений.

API библиотеки функций и классов включает в себя описание сигнатур и семантики функций.

Для работы с API сервера со стороны клиента, там создаются маршруты, которые имеют уникальные идентификаторы запрашиваемых ресурсов, параметры, передаваемые в адресной строке или в «теле» запроса.

Существует несколько основных концепций для построения такого типа интерфейсов, например REST API.

REST или Representational State Transfer — это «передача репрезентативного состояния» или «передача „самоописываемого“ состояния») — архитектурный стиль взаимодействия компонентов распределённого приложения в сети.

Другими словами, REST — это набор правил того, как программисту организовать написание кода серверного приложения, чтобы все системы легко обменивались данными и приложение можно было масштабировать. REST представляет собой согласованный набор ограничений, учитываемых при проектировании распределённой гипермедиа-системы. В определённых случаях (интернет-магазины, поисковые системы, прочие системы, основанные на данных) это приводит к повышению производительности и упрощению архитектуры.

В широком смысле[уточнить] компоненты в REST взаимодействуют наподобие взаимодействия клиентов и серверов во Всемирной паутине. REST является альтернативой RPC.

В сети Интернет вызов удалённой процедуры может представлять собой обычный HTTP-запрос (обычно GET или POST; такой запрос называют «REST-запрос»), а необходимые данные передаются в качестве параметров запроса.

Для веб-служб, построенных с учётом REST (то есть не нарушающих накладываемых им ограничений), применяют термин «RESTful».

В отличие от веб-сервисов (веб-служб) на основе SOAP, не существует «официального» стандарта для RESTful веб-API. Дело в том, что REST является архитектурным стилем, в то время как SOAP является протоколом. Несмотря на то, что REST не является стандартом сам по себе, большинство RESTful-реализаций используют такие стандарты, как HTTP, URL, JSON и, реже, XML.

Свойства архитектуры, которые зависят от ограничений, наложенных на REST-системы:

- Производительность — взаимодействие компонентов системы может являться доминирующим фактором производительности и эффективности сети с точки зрения пользователя;

- Масштабируемость для обеспечения большого числа компонентов и взаимодействий компонентов.

Рой Филдинг — один из главных авторов спецификации протокола HTTP, описывает влияние архитектуры REST на масштабируемость следующим образом:

- Простота унифицированного интерфейса;
- Открытость компонентов к возможным изменениям для удовлетворения изменяющихся потребностей (даже при работающем приложении).
- Прозрачность связей между компонентами системы для сервисных служб.
- Переносимость компонентов системы путем перемещения программного кода вместе с данными.
- Надёжность, выражающаяся в устойчивости к отказам на уровне системы при наличии отказов отдельных компонентов, соединений или данных.

Есть несколько требований к архитектуре REST интерфейса сервера:

- Модель множество клиентов – один сервер.
- Отсутствие состояния и зависимости между состояниями сервера.
- Должно присутствовать кэширование.
- Все «точки» вызова интерфейса должны иметь одинаковый вид и правила формирования.
- Приложение должно быть поделено на слои.

Сейчас некоторые разработчики называют REST «устаревшим» и предлагают использовать в клиент-серверных приложениях GraphQL.

GraphQL — это язык запросов и обработки данных с открытым исходным кодом для API, а также среда выполнения для выполнения запросов с существующими данными. GraphQL был разработан внутри Facebook в 2012 году, а затем публично выпущен в 2015 году. 7 ноября 2018 года проект GraphQL был перенесен из Facebook в недавно созданный фонд GraphQL, размещенный некоммерческим фондом Linux.

Он обеспечивает подход к разработке веб-API и сопоставим с REST и другими архитектурами веб-сервисов. Это позволяет клиентам определять структуру требуемых данных со стороны клиента таким образом, что с сервера возвращается та структура данных, которая нужна.

Например, если в модели базы данных Пользователь содержится 10 столбцов (или полей), то для формирования ниспадающего списка пользователей нужно только два поля: его идентификатор и имя (ну или email). Это что предотвращает возврат чрезмерно больших объемов данных, так как номенклатура данных как бы «заказывается» серверу со стороны клиента.

Кроме того, для REST характерно большое количество маршрутов (иногда сотни), что не свойственно для GraphQL – здесь используется только одна точка доступа, как в случае с сокетными соединениями, так как все манипуляции с данными происходят «внутри» запросов и ответов.

Однако, гибкость и богатство языка запросов также добавляют сложности реализации, так что для простых API использовать GraphQL нецелесообразно.

GraphQL состоит из системы типов, языка запросов и семантики выполнения, статической проверки и анализа типов. Он поддерживает чтение, запись (изменение) и подписку на изменения данных (обновления в реальном времени – чаще всего реализуемые с помощью Websockets).

Серверы GraphQL доступны для нескольких языков, включая Haskell, JavaScript, Perl, Python, Ruby, Java, C++, C#, Scala, Go, Rust, Elixir, Erlang, PHP, R, D и Clojure.

Таким образом, данной лекции мы рассмотрели все основные виды программных интерфейсов между элементами информационных и автоматизированных систем.

ТЕМА 5. АППАРАТНЫЕ ИНТЕРФЕЙСЫ СВЯЗИ ИНФОРМАЦИОННЫХ И АВТОМАТИЗИРОВАННЫХ СИСТЕМ

Мы уже рассмотрели принципы сопряжения программных компонент и приложений, далее мы проведем анализ способов передачи данных между аппаратными устройствами, например между сенсорами, исполнительными механизмами, цифровыми микросхемами и устройствами.

Необходимо понимать, что многие элементы имеют свои собственные интерфейсы связи, но, как и на программном уровне, эти интерфейсы стандартизированы, иначе бы устройства разных производителей и серий не могли бы обмениваться данными и командами.

Первым рассматриваемым интерфейсом будет аналоговый. До сих пор мы можем увидеть множество аналоговых устройств, которые «вводят в контур общения» цифровых устройств. На самом деле есть только один способ такого обмена: аналого-цифровой преобразователь (АЦП) для приема информации от аналогового устройства и цифро-аналоговый (ЦАП) для передачи информации на такое устройство.

АЦП — это устройство, преобразующее входной аналоговый сигнал в дискретный код (цифровой сигнал).

Как правило, АЦП — электронное устройство, преобразующее напряжение в двоичный цифровой код. Тем не менее, некоторые неэлектронные устройства с цифровым выходом следует также относить к АЦП, например, некоторые типы преобразователей угол-код. Простейшим одноразрядным двоичным АЦП является компаратор.

Одной из основных характеристик АЦП является его разрядность или разрешение.

Разрешение АЦП — минимальное изменение величины аналогового сигнала, которое может быть преобразовано данным АЦП — связано с его разрядностью. В случае единичного измерения без учёта шумов разрешение напрямую определяется разрядностью АЦП.

Разрядность АЦП характеризует количество дискретных значений, которые преобразователь может выдать на выходе. В двоичных АЦП измеряется в битах. Например, двоичный 8-разрядный АЦП способен выдать 256 дискретных значений (0…255), поскольку 2^8=256.

Также распространены 10 и 12 битные АЦП.

Например, диапазон входных значений АЦП: от 0 до 10 вольт, разрядность двоичного АЦП 12 бит: 212 = 4096 уровней квантования, разрешение двоичного АЦП по напряжению: (10–0)/4096 = 0,00244 вольта = 2,44 мВ.

На практике разрешение АЦП ограничено отношением сигнал/шум входного сигнала. При большой интенсивности шумов на входе АЦП различение соседних уровней входного сигнала становится невозможным, то есть ухудшается разрешение.

По способу применяемых алгоритмов АЦП делят на:

- Последовательные прямого преобразования
- Последовательного приближения
- Последовательные с сигма-дельта-модуляцией
- Параллельные одноступенчатые
- Параллельные двух- и более ступенчатые (конвейерные)

Подробное рассмотрение работы АЦП на физическом уровне выходит за рамки нашего курса, но мы должны понимать, что АЦП просто подключается к аналоговому выходу датчика, который, например, измеряет температуру и представляет эту величину в виде выходного напряжения. Выход (последовательный или параллельный) с АЦП подключается к микроконтроллеру или ПЛИС, которые способны проанализировать такие данные. Далее микроконтроллер уже сам разберется, как передавать данные по своим каналам связи.

Антиподом АЦП является ЦАП, — это устройство для преобразования цифрового (обычно двоичного) кода в аналоговый сигнал (ток, напряжение или заряд). Цифро-аналоговые преобразователи являются интерфейсом между дискретным цифровым миром и аналоговыми сигналами. Современные ЦАП создаются по полупроводниковым технологиям в виде интегральной схемы.

ЦАП применяется всегда в телекоммуникационных системах и системах управления. Например:

- в системах воспроизведения аудио;
- в дисплеях;
- для формирование информационного сигнала для смесителей и управляемых генераторов;
- в системах управлением двигателем;
- в системах прямого цифрового синтеза (DDS — Direct digital synthesizer).

Для ЦАП также выделяют следующие основные характеристики:

- Разрядность. Определяет количество уровней аналогового сигнала, которое может воспроизводить ЦАП. Для N разрядного ЦАП число уровней аналогового сигнала равно 2N (включая значение для нулевого кода);
- Напряжение питания.

Также есть специальные характеристики типа статических характеристик преобразования, статической нелинейности, монотонность, смещение нуля, ошибки усиления, а также быстродействие, отношение сигнала к шуму и динамический диапазон.

Существуют последовательные ЦАП, широтно-импульсные модуляторы, циклические ЦАП, конвейерные УАП, а также параллельные ЦАП.

Как и АЦП, изучение ЦАП также выходит за рамки курса, но мы должны понимать, что мы можем подключить цифровой выход микроконтроллера к ЦАП и передавать на него цифровые данные, в которых закодировано нужное нам напряжение «после» ЦАП и он сделает подобное преобразование.

Имея микроконтроллер и пару ЦАП/АЦП, можно построить, например, управление нагревателем в паре с термодатчиком. При этом аналоговый сигнал с термометра через АЦП выбирается на микропроцессор, там производится расчет «сдвига» нагревательного прибора и через ЦАП на последний выводится управляющий сигнал.

Естественно, что реальные системы гораздо сложнее, но принципы, по сути, те же самые.

Далее рассмотрим, как подключаются цифровые датчики к микроконтроллерам и как соединяются сами по себе микроконтроллеры.

Существует несколько стандартных интерфейсов сопряжения, применяющихся в промышленности для коммутации цифровых устройств:

- RS232;
- CAN;
- USB;
- 1WARE;
- I2C;
- SPI и другие

Шина USB широко нам известна, поэтому мы сосредоточим наше внимание на других стандартах сопряжения аппаратных устройств в АС и ИС.

RS232

RS232 – это достаточно «старый» интерфейс для сопряжения устройств. Он называется в более расширенной версии Recommended Standard 232 и предназначен для работы асинхронных устройств USART на физическом уровне.

Примечательно то, что устройство, поддерживающее этот стандарт, широко известно как последовательный порт обычных персональных компьютеров. Сейчас этот порт полностью вытеснен USB, но исторически стандарт имел широкое распространение в телекоммуникационном оборудовании. В настоящее время используется для подключения к компьютерам широкого спектра оборудования, нетребовательного к скорости обмена, особенно при значительном удалении его от компьютера и отклонении условий применения от стандартных. В компьютерах, занятых офисными и развлекательными приложениями, практически вытеснен интерфейсом USB.

RS-232 обеспечивает передачу данных и некоторых специальных сигналов между терминалом или Data Terminal Equipment, DTE и коммуникационным устройством Data Communications Equipment, DCE на расстояние до 15 метров на максимальной скорости 115 200 бод.

Так как этот интерфейс известен не только простотой программирования, но и неприхотливостью, в реальных условиях это расстояние увеличивается во много раз с примерно пропорциональным снижением скорости.

С другой стороны, на коротких расстояниях можно «разогнать» этот интерфейс до скорости около 1 Мбит/с.

Протокол интерфейса предполагает два режима передачи данных — синхронный и асинхронный, а также два метода управления обменом данных:

аппаратный и программный. Каждый режим может работать с любым методом управления. В протоколе также предполагается вариант управления передачей данных по специальным сигналам, устанавливаемым хостом (DSR — сигнал состояния готовности, DTR — сигнал готовности передачи данных), но для работы между двумя микроконтроллерами, дополнительные сигналы не используются.

Для передачи данных по интерфейсу RS-232 используется код NRZ, который не является самосинхронизирующимся, поэтому для синхронизации используются стартовый и стоповый биты, позволяющие выделить битовую последовательность и синхронизировать приёмник с передатчиком.

Рассмотрим принцип работы интерфейса.

Информация передаётся по проводам двоичным сигналом с двумя уровнями напряжения (код NRZ). Логическому «0» соответствует положительное напряжение (от +5 до +15 В для передатчика), а логической «1» — отрицательное (от −5 до −15 В для передатчика).

Такие напряжения касаются только персональных ЭВМ, так как у микроконтроллеров максимальное напряжение около 5 В.

Для электрического согласования линий RS-232 и стандартной цифровой логики UART выпускается большая номенклатура микросхем драйверов, например, MAX232 или современные драйверы типа FT232 (которые кроме всего прочего, преобразуют RS-232 в физический USB в точке подключения к ЭВМ и организуют виртуальный COM порт на самом компьютеры, так что логические процедуры и программные библиотеки не меняются).

Передача начинается с удержания линии в логической единице, далее вводится 1 стартовый бит, «поймав» который, приемная сторона подготавливает приемник.

После стартового бита передаются 7, 8, 9, 10 бит данных, а после них бит паритета. Бит паритета является контрольным битом, и он бывает битом четного или нечетного паритета. В любом случае, паритет считает количество единиц в пакете данных. Если число 1 в пакете четное и само паритет четный, то он будет равен 1. И так далее.

После паритета следует 1, 1.5 или 2 стоповых битов для остановки передачи.

Помимо линий входа и выхода данных, RS-232 регламентировал ряд необязательных вспомогательных линий для аппаратного управления потоком данных, которые можно применять для управления внешними устройствами.

CAN

CAN или Controller Area Network — сеть контроллеров. Это стандарт промышленной сети, ориентированный, прежде всего, на объединение в единую сеть различных исполнительных устройств и датчиков. Режим передачи — последовательный, широковещательный, пакетный.

CAN разработан компанией Robert Bosch GmbH в середине 1980-х и в настоящее время широко распространён в промышленной автоматизации, технологиях домашней автоматизации («умного дома»), автомобильной

промышленности и многих других областях. Сейчас это стандарт для автомобильной автоматики, промышленности и микроконтроллеров.

Непосредственно стандарт CAN компании Bosch определяет передачу в отрыве от физического уровня — он может быть каким угодно, например, радиоканалом или оптоволокном. Но на практике под CAN-сетью обычно подразумевается сеть топологии «шина» с физическим уровнем в виде дифференциальной пары, определённым в стандарте ISO 11898. Передача ведётся кадрами, которые принимаются всеми узлами сети. Для доступа к шине выпускаются специализированные микросхемы — драйверы CAN-шины.

CAN является синхронной шиной с типом доступа Collision Resolving (CR, разрешение коллизии), который, в отличие от Collision Detect (CD, обнаружение коллизии) сетей (Ethernet), детерминировано (приоритетно) обеспечивает доступ на передачу сообщения, что особо ценно для промышленных сетей управления (fieldbus). Передача ведётся кадрами. Полезная информация в кадре состоит из идентификатора длиной 11 бит (стандартный формат) или 29 бит (расширенный формат, надмножество предыдущего) и поля данных длиной от 0 до 8 байт. Идентификатор говорит о содержимом пакета и служит для определения приоритета при попытке одновременной передачи несколькими сетевыми узлами.

По шине CAN могут передаваться следующие типы кадров:

- Кадр данных (data frame) — передаёт данные.
- Кадр удаленного запроса (remote frame) — служит для запроса на передачу кадра данных с тем же идентификатором.
- Кадр перегрузки (overload frame) — обеспечивает промежуток между кадрами данных или запроса.
- Кадр ошибки (error frame) — передаётся узлом, обнаружившим в сети ошибку.

Кадры данных и запроса отделяются от предыдущих кадров межкадровым промежутком.

Каждый кадр имеет свой формат и в общем случае, все они могут содержать:

- Начало кадра.
- Идентификатор.
- Запрос на передачу.
- Служебные и зарезервированные биты.
- Длины данных.
- Контрольные суммы.
- Разграничители.
- Конец передачи.

Так как CAN это шина, то в ней существует арбитраж. При свободной шине любой узел может начинать передачу в любой момент. В случае одновременной передачи кадров двумя и более узлами проходит арбитраж доступа: передавая идентификатор, узел одновременно проверяет состояние шины. Если при передаче рецессивного бита принимается доминантный — считается, что другой

узел передаёт сообщение с большим приоритетом, и передача откладывается до освобождения шины. Таким образом, в отличие, например, от Ethernet, в CAN не происходит непроизводительной потери пропускной способности канала при коллизиях. Цена этого решения — возможность того, что сообщения с низким приоритетом никогда не будут переданы.

CAN имеет несколько механизмов контроля и предотвращения ошибок, процедуры определения скорости передачи данных и длины сети.

Интересно так же то, что в шине CAN существуют протоколы верхнего уровня, расположенные над описанными выше принципами передачи данных на физическом уровне.

1WARE

1-Wire (с англ. — «один провод») — двунаправленная шина связи для устройств с низкоскоростной передачей данных (обычно 15,4 Кбит/с, максимум 125 Кбит/с в режиме overdrive), в которой данные передаются по цепи питания (то есть всего используются два провода — один общий (GND), а второй для питания и данных; в некоторых случаях используют и отдельный провод питания).

Соответственно, топология такой сети — общая шина. Сеть устройств 1-Wire со связанным основным устройством названа «MicroLan».

Обычно используется для того, чтобы связываться с недорогими простыми устройствами, такими, как, например, цифровые термометры и измерители параметров внешней среды.

Шина обладает определенными достоинствами. В первую очередь это то, что для связи с устройством требуется лишь два провода: на данные и заземление. Интегральная схема включает конденсатор ёмкостью 800 пФ для питания от линии данных (так называемое паразитное питание). Вторым достоинством является то, что расстояние передачи считается большим и достигает 300 м.

Устройство 1-Wire может находиться как на печатной плате вместе с устройством управления, так и отдельно. Иногда они предназначены лишь для поддержки устройств 1-Wire, но во многих коммерческих приложениях устройство 1-Wire — просто один из чипов, создающих нужное решение. Иногда они присутствуют, например, в аккумуляторных батареях ноутбуков и сотовых телефонов.

Примечательно также то, что 1WARE является основой популярной технологии электронных ключей iButton. стандарт механической упаковки, в котором компонент 1-Wire размещается внутри небольшой «таблетки» из нержавеющей стали и подключается к системам шины 1-Wire посредством розеток с контактами, которые касаются «крышки» и «дна» таблетки.

Каждая микросхема 1-Wire имеет уникальный номер. Это позволяет использовать устройства iButton в качестве простых идентификаторов личности, например, в системах контроля и управления доступом (СКУД). В этом качестве они успешно конкурируют с бесконтактными карточками, использующими технологию RFID.

Имеются устройства iButton с поддержкой криптографии, что позволяет создавать на их основе защищённые хранилища небольших объёмов данных или средства сильной аутентификации. Такие устройства могут конкурировать со смарт-картами в некоторых применениях.

1WARE широко используется в датчиках. Не требуется отдельного питания, возможно подключить по одному проводу целую гирлянду разнообразных датчиков. Система таких датчиков легко контролируется на предмет аварий. Записи о калибровках могут храниться прямо в датчиках.

Измерение температуры — одно из самых массовых применений 1-Wire устройств. В сельском хозяйстве применяется для многоточечного контроля температуры в теплицах, ульях, элеваторах, инкубаторах, овощехранилищах. Популярны домашние метеостанции, подключаемые по этому интерфейсу.

I2C

I2C или Inter-Integrated Circuit – это последовательная асимметричная шина для связи между интегральными схемами внутри электронных приборов. Использует две двунаправленные линии связи (SDA и SCL), применяется для соединения низкоскоростных периферийных компонентов с процессорами и микроконтроллерами (например, на материнских платах, во встраиваемых системах, в мобильных телефонах).

Шина I2C синхронная, состоит из двух линий: данных (SDA) и тактов (SCL). Есть ведущий (master) и ведомые (slave). Инициатором обмена всегда выступает ведущий, обмен между двумя ведомыми невозможен. Всего на одной двухпроводной шине может быть до 127 устройств.

Такты на линии SCL генерирует master. Линией SDA могут управлять как мастер, так и ведомый в зависимости от направления передачи. Единицей обмена информации является пакет, обрамленный уникальными условиями на шине, именуемыми стартовым и стоповым условиями. Мастер в начале каждого пакета передает один байт, где указывает адрес ведомого и направление передачи последующих данных. Данные передаются 8-битными словами. После каждого слова передается один бит подтверждения приема приемной стороной.

Стандартные напряжения +5 В или +3,3 В, однако допускаются и другие.

Классическая адресация включает 7-битное адресное пространство с 16 зарезервированными адресами. Это означает, что разработчикам доступно до 112 свободных адресов для подключения периферии на одну шину.

Основной режим работы — 100 кбит/с; 10 кбит/с в режиме работы с пониженной скоростью. Также немаловажно, что стандарт допускает приостановку тактирования для работы с медленными устройствами.

Процедура обмена начинается с того, что ведущий формирует состояние СТАРТ: при ВЫСОКОМ уровне на линии SCL он генерирует переход сигнала линии SDA из ВЫСОКОГО состояния в НИЗКОЕ. Этот переход воспринимается всеми устройствами, подключенными к шине, как признак начала процедуры обмена. Генерация синхросигнала — это всегда обязанность ведущего; каждый ведущий генерирует свой собственный сигнал синхронизации при пересылке данных по шине.

При передаче посылок по шине I²C каждый ведущий генерирует свой синхросигнал на линии SCL. После формирования состояния СТАРТ ведущий опускает состояние линии SCL в НИЗКОЕ состояние и выставляет на линию SDA старший бит первого байта сообщения. Количество байт в сообщении не ограничено. Спецификация шины I²C разрешает изменения на линии SDA только при НИЗКОМ уровне сигнала на линии SCL. Данные действительны и должны оставаться стабильными только во время ВЫСОКОГО состояния синхроимпульса. Для подтверждения приёма байта от ведущего-передатчика ведомым-приёмником в спецификации протокола обмена по шине I²C вводится специальный бит подтверждения, выставляемый на шину SDA после приёма 8 бит данных.

Процедура обмена завершается тем, что ведущий формирует состояние СТОП — переход состояния линии SDA из НИЗКОГО состояния в ВЫСОКОЕ при ВЫСОКОМ состоянии линии SCL. Состояния СТАРТ и СТОП всегда вырабатываются ведущим. Считается, что шина занята после фиксации состояния СТАРТ. Шина считается освободившейся через некоторое время после фиксации состояния СТОП.

Таким образом, передача 8 бит данных от передатчика к приёмнику завершаются дополнительным циклом (формированием 9-го тактового импульса линии SCL), при котором приёмник выставляет низкий уровень сигнала на линии SDA, как признак успешного приёма байта.

Подтверждение при передаче данных обязательно, кроме случаев окончания передачи ведомой стороной. Соответствующий импульс синхронизации генерируется ведущим. Передатчик отпускает (переводит в ВЫСОКОЕ состояние) линию SDA на время синхроимпульса подтверждения. Приёмник должен удерживать линию SDA в течение ВЫСОКОГО состояния синхроимпульса подтверждения в стабильном НИЗКОМ состоянии.

В том случае, когда ведомый-приёмник не может подтвердить свой адрес (например, когда он выполняет в данный момент какие-либо функции реального времени), линия данных должна быть оставлена в ВЫСОКОМ состоянии. После этого ведущий может выдать состояние СТОП для прерывания пересылки данных. Если в пересылке участвует ведущий-приёмник, то он должен сообщить об окончании передачи ведомому-передатчику путём неподтверждения последнего байта. Ведомый-передатчик должен освободить линию данных для того, чтобы позволить ведущему выдать состояние СТОП или повторить состояние СТАРТ.

Синхронизация выполняется с использованием подключения к линии SCL по правилу монтажного И. Это означает, что ведущий не имеет монопольного права на управление переходом линии SCL из НИЗКОГО состояния в ВЫСОКОЕ. В том случае, когда ведомому необходимо дополнительное время на обработку принятого бита, он имеет возможность удерживать линию SCL в низком состоянии до момента готовности к приёму следующего бита. Таким образом, линия SCL будет находиться в НИЗКОМ состоянии на протяжении самого длинного НИЗКОГО периода синхросигналов.

Устройства с более коротким НИЗКИМ периодом будут входить в состояние ожидания на время, пока не кончится длинный период. Когда у всех задействованных устройств кончится НИЗКИЙ период синхросигнала, линия SCL перейдет в ВЫСОКОЕ состояние. Все устройства начнут проходить ВЫСОКИЙ период своих синхросигналов. Первое устройство, у которого кончится этот период, снова установит линию SCL в НИЗКОЕ состояние. Таким образом, НИЗКИЙ период синхролинии SCL определяется наидлиннейшим периодом синхронизации из всех задействованных устройств, а ВЫСОКИЙ период определяется самым коротким периодом синхронизации устройств.

Механизм синхронизации может быть использован приёмниками как средство управления пересылкой данных на байтовом и битовом уровнях.

Каждое устройство, подключённое к шине, может быть программно адресовано по уникальному адресу. Для выбора приёмника сообщения ведущий использует уникальную адресную компоненту в формате посылки. При использовании однотипных устройств ИС часто имеют дополнительный селектор адреса, который может быть реализован как в виде дополнительных цифровых входов селектора адреса, так и в виде аналогового входа. При этом адреса таких однотипных устройств оказываются разнесены в адресном пространстве устройств, подключенных к шине.

В обычном режиме используется 7-битная адресация.

Адрес ведомого может состоять из фиксированной и программируемой части. Часто случается, что в системе имеется несколько однотипных устройств (к примеру, ИМС памяти, или драйверов светодиодных индикаторов), поэтому при помощи программируемой части адреса становится возможным подключить к шине максимально возможное количество таких устройств. Количество программируемых битов в адресе зависит от количества свободных выводов микросхемы. Иногда используется один вывод с аналоговой установкой программируемого диапазона адресов.

I²C находит применение в устройствах, предусматривающих простоту разработки и низкую себестоимость изготовления при относительно неплохой скорости работы.

Список возможных применений I2C:

- доступ к модулям памяти NVRAM;
- доступ к низкоскоростным ЦАП/АЦП;
- регулировка контрастности, насыщенности и цветового баланса мониторов;
- регулировка звука в динамиках;
- управление светодиодами, в том числе в мобильных телефонах;
- чтение информации с датчиков мониторинга и диагностики оборудования, например термостат центрального процессора или скорость вращения вентилятора охлаждения;
- чтение информации с часов реального времени (кварцевых генераторов);
- информационный обмен между микроконтроллерами.

ТЕМА 6. ПРИНЦИПЫ КОМПЛЕКСИРОВАНИЯ И СОПРЯЖЕНИЯ АППАРАТНЫХ И ПРОГРАММНЫХ СИСТЕМ

Комплексирование и сопряжение аппаратных и программных систем можно так же назвать «системной интеграцией».

Системная интеграция в большинстве случаев является вынужденной мерой, направленной на повышение эффективности бизнес-процессов, использующих информационную систем.

Что происходит если отсутствует интеграция в АС и ИС: каждый элемент системы будет работать сам по себе и придется собирать данные вручную. Например, если мы строим информационную систему для обеспечения деятельности компании. Пусть у нас есть три отдельные информационные системы: «Система учета запасов» (учет и анализ движения на складе), «CRM-система» (учет и анализ продаж и других отношений с клиентами) и «Система учета» (учет и финансовый анализ). И между ними нет обмена информацией.

Это приводит к тому, что продажи после выставления счетов вашим клиентам приходится распечатывать и относить в бухгалтерию. В бухгалтерском учете они учитываются в системе бухгалтерского учета. Бухгалтерский учет фиксирует поступление денежных средств на счет. Менеджеры по продажам, не имея возможности получать оплату автоматически в CRM-системе, вынуждены ежедневно запрашивать в бухгалтерии поступления денег от клиентов. В такой ситуации возникает большой поток документов между менеджерами, бухгалтерией и складом, а также двойная проверка действия (один раз в системе инвентаризации, второй раз в бухгалтерии).

Если вы подсчитаете затраты оплаченного компанией времени сотрудников на выполнение дублирующих процедур в разных системах (выделено красным на диаграмме), вы можете получить значительную долю от общих затрат фирмы.

Далее рассмотрим разные способы интеграции или комплексирования.

Вертикальная интеграция

В соответствии с этим подходом системы интегрируются по принципу функциональной экспертизы. Например, в данном случае выбираются два вида экспертизы: оперативный учет и бухгалтерский учет. В то время как бухгалтерский учет находится вертикально выше оперативного счета. В нашем примере подсистема оперативного учета доставляет данные в подсистему бухгалтерского учета.

Это значительно сокращает рабочую силу, необходимую для дублирования и бумажных операций, однако есть два отягчающих момента.

Во-первых, такую систему чрезвычайно трудно функционально расширить. Например, компания может захотеть создать подсистему-экспертизу» для «Аналитика», которая будет вертикально расположена над экспертизой «Бухгалтерский учет». Это исследование в значительной степени основано на данных «Оперативного учета». Поэтому в дополнение к развитию

подсистемы «Аналитику» придется модифицировать подсистему «Бухгалтерия», чтобы получить от нее и отделить ее от дополнительной информации «Оперативного учета».

Интеграции типа «звезда» или «спагетти»

В соответствии с этим подходом каждая из компаний, используемых в подсистемах, может при необходимости ссылаться на функциональность любой другой подсистемы, в то время как каждая подсистема может также использоваться любой другой подсистемой. Этот тип отношений между элементами называется «многие ко многим».

В этом случае имеют место практически неограниченные возможности интеграции подсистем друг с другом (конечно, если технологически подсистемы позволяют это сделать).

Но, с другой стороны, затраты на поддержку таких интеграционных схем растут экспоненциально с увеличением числа интегрированных подсистем.

Например, если в нашем случае нам нужно что-то изменить в подсистеме учета (например, изменить ее объектную модель), это может привести к необходимости обработки всех других подсистем, использующих ее, потому что вызовы старой объектной модели завершатся неудачей. Для трех взаимодействующих систем это может быть не так важно, но для нескольких десятков систем становится очень важным.

Горизонтальная интеграция

Этот подход предполагает использование специализированного промежуточного программного обеспечения - так называемой корпоративной служебной шины. Основной целью этого является сохранение функциональности репозитория корпоративных приложений, подключенных к нему, и использование этих функций другими приложениями, также подключенными к этой шине. Связь между приложениями может осуществляться, например, в форме обмена сообщениями или вызова функций, опубликованных в виде веб-служб. Подключение системы к шине осуществляется путем создания специального адаптера для каждой системы. Затем «опубликованные» функции системы становятся доступными для других подключенных систем.

Например, CRM-система при подключении к шине публикует свои функции для работы с клиентской базой данных. В шине предусмотрена возможность их использования учетными и складскими системами. В свою очередь, CRM-система получает возможность работы с данными бухгалтерского учета и инвентаризации.

Преимущество такого подхода заключается в том, что системы могут быть заменены в рамках существующих спецификаций опубликованных функций. Однако никаких изменений в других системах не требуется. Более того, подключение новой системы достаточно стандартизировано и упрощено. Например, можно подключить новую систему «Аналитик», который немедленно получит доступ ко всем остальным системам.

Замена интеграции простым множеством функциональных модулей

Например, все вышеперечисленные подсистемы могут быть реализованы как функциональные модули ERP - системы любого поставщика. В этом случае необходимость интеграции отпадает, так как система изначально однородна, обеспечивая гораздо большую согласованность между функциональными модулями, чем любая из вышеперечисленных интеграций между различными системами.

Давайте более подробно рассмотрим объекты и методы интеграции систем.

Ранее в описании подходов к интеграции систем мы рассматривали каждую информационную систему как» неделимый «объект». Однако информационная система представляет собой комбинацию нескольких компонентов, поэтому, говоря об интеграции информационных систем, правильнее говорить об интеграционных компонентах.

Обычно информационная система содержит следующие компоненты:

- Платформа, на которой работают остальные компоненты системы, включая оборудование (аппаратное обеспечение) и системное программное обеспечение.
- Данные, на которых работает система. Состоят из СУБД и баз данных.
- Приложения, реализующие бизнес-логику для работы с системой данных. Состоит из бизнес-логики, пользовательского интерфейса, вспомогательных компонентов (фреймворков) и сервера приложений, который обеспечивает хранение и доступ к компонентам приложений.
- Бизнес-процессы, представляющие сценарии взаимодействия пользователей с системой.

Поэтому интеграция информационных систем заключается в интеграции одного или нескольких компонентов интегрируемых информационных систем (объектов интеграции).

- Платформы интеграции.
- Интеграция данных.
- Интеграция приложений.
- Интеграция бизнес-процессов.

Целями интеграционных платформ являются:

- Обеспечение взаимодействия между приложениями, работающими на различных аппаратных и программных платформах (например, между приложениями, работающими на серверах Windows, Solaris, Linux и т.д.).
- Для обеспечения того, чтобы приложения разрабатывались для одной аппаратной платформы, других программных и аппаратных платформ (например, приложений под управлением Windows на платформах Linux, Solaris и т.д.).

Существует несколько подходов, направленных на достижение этих целей. В рамках каждого подхода существуют различные технологии.

- Удаленный вызов процедур (RPC, веб-службы, REST и т.д.).
- Промежуточное программное обеспечение (Microsoft.NET, Среда выполнения Java).
- Виртуализация.

Технология удаленного вызова процедур (в широком смысле процедура определяется как некоторая функциональность приложения) позволяет опубликовать процедуру и разрешить вызов (передавать входные параметры и выдавать выходные результаты) для приложений, работающих на других платформах.

Концепция программного обеспечения промежуточного уровня (платформа, среда выполнения, виртуальная машина) заключается в разработке программных приложений, не использующих службы конкретной операционной системы (например, API Windows), а использующих службы промежуточного программного обеспечения. Разработчики промежуточного ПО, созданного для его реализации под различными операционными системами, которые передают вызовы соответствующим функциям, в framework вызывают соответствующую операционную систему. Типичным примером является среда выполнения технологии Java. Приложения, разработанные для этой технологии, работают на всех аппаратных и программных платформах (Windows, Linux и т.д.) без каких-либо модификаций самих приложений. Аналогичные возможности предоставляет Microsoft .Net Framework.

Интересной и инновационной концепцией является виртуализация. Интеграция платформ, она актуальна в той мере, в какой позволяет существенно упростить использование различных платформ и, следовательно, использование систем, которые требуют для своего функционирования наличия определенных платформ. При отсутствии виртуализации возможна одновременная работа N операционных сред на N серверах, использование технологий виртуализации позволяет обеспечить функционирование операционных сред от N до M серверов. Если $N > M$, это позволяет снизить затраты на оборудование за счет более эффективного использования. Виртуализация позволяет развертывать и одновременно использовать на одном физическом сервере несколько операционных систем: Windows, Linux и т.д. На каждом из этих виртуальных серверов могут быть развернуты соответствующие системы, которые будут доступны одновременно. Примеры технологий виртуализации: Microsoft Hyper-V, KVM, OpenVZ, Virtuozzo, VMware, Xen, Docker, Kubernetes и др.

По определению, любая информационная система работает с данными. В подавляющем большинстве случаев система включает базу данных для их хранения. Интеграция на уровне данных предполагает обмен данными между различными системами. Интеграция данных может быть проще, чем интеграция приложений, поскольку промышленные базы данных, в которых обычно хранятся информационные системы данных, развили способность программного

доступа к данным из других приложений. Таким образом, сами приложения могут иметь очень ограниченные возможности программного обеспечения (за пределами их собственного пользовательского интерфейса) использовать свои функциональные возможности для внешних систем.

Подходы к интеграции данных.

- Универсальный доступ к данным.
- Хранилище данных.

Технология универсального доступа к данным позволяет обеспечить единый доступ к данным в различных СУБД. Посредником для работы с конкретной СУБД в данном случае является драйвер для вашей СУБД. Например, один и тот же SQL-запрос для извлечения данных SELECT * FROM TABLE может использоваться для извлечения данных из таблицы TABLE, хранящейся в разных СУБД. Это позволяет абстрагироваться от специфики конкретной СУБД и легко интегрировать данные, хранящиеся в разных СУБД. Наиболее распространенные технологии в этом классе: ODBC, JDBC, ADO.NET. Кроме того, сегодня широко распространены технологии объектно-реляционное отображение (ORM), которое также позволяет абстрагироваться от деталей взаимодействия с конкретными СУБД. Примерами таких технологий являются Entity Framework, Hibernate, NHibernate, Flexberry ORM, Django ORM и т.д.

Концепция хранилища данных заключается в создании корпоративного хранилища данных. Хранилище данных – это база данных, в которой хранятся данные, собранные из баз данных различных информационных систем, с целью их дальнейшего анализа. Например, может существовать единое хранилище данных компании, в котором собрана информация из бухгалтерии, операционных систем, внешних систем наших партнеров. Для хранения данных используется технология (OLAP), отличная от технологий создания оперативной базы данных (OLTP). В основном это делается для повышения производительности сложных аналитических запросов во многих аспектах (многомерные запросы). Подходы к созданию и заполнению хранилищ данных отражены в парадигме ETL (извлечение, преобразование, загрузка = извлечение, преобразование и загрузка). Технологии и инструменты анализа больших массивов данных для выявления закономерностей предметной области объединила концепция интеллектуального анализа данных. Термин для набора технологий хранения данных и инструментов, обеспечивающих перевод транзакционной бизнес – информации в форму, подходящую для бизнес-анализа-Бизнес-аналитика.

Интеграция на прикладном уровне подразумевает использование готовых функций приложений другими приложениями. Например, разработав систему электронного документооборота, можно использовать эту систему в качестве текстового редактора MS Word вместо того, чтобы разрабатывать собственный текстовый редактор. Или, например, ЧЕРЕЗ колл-центр, получая входящий звонок от клиента, который имеет возможность получить доступ к биллинговой системе для проверки баланса (ввод номера телефона вызывающего абонента, вывод текущего баланса) и в зависимости от состояния баланса подключается к оператору или автоматически сообщает о необходимости пополнения вашего счета. Структура базы данных биллинговой системы — это ее внутренняя

информация, публикуемая для конкретной функции, позволяющая другим системам работать с конкретными данными.

Стоит упомянуть следующие подходы к интеграции приложений:

- Интерфейсы прикладного программирования.
- Обмен сообщениями (корпоративная служебная шина).
- Сервис-ориентированная архитектура.
- Интеграция пользовательских интерфейсов.

Программный интерфейс конкретной системы представляет опубликованную» «функциональность системы, которая может быть использована извне. Функциональность может быть опубликована в виде набора функций (пример – Windows API) или объектной модели (объекты со свойствами и методами, пример – объектная модель приложений Microsoft Office).

В большинстве случаев интеграция нескольких систем заключается в передаче информации между ними, например, в форме запрос-ответ. Если системы работают в гетерогенных распределенных средах, крайне важно обеспечить гарантию, безопасность и обработку информации, передаваемой между приложениями. Эти и другие принципы реализованы в корпоративных системах обмена сообщениями. В данном случае речь идет об обмене сообщениями между приложениями, а не людьми, как, например, в случае электронной почты или мессенджеров. Функциональность этих систем достаточно прозрачна – получение сообщения из одного приложения, транспортировка, основанная на правилах, и передача этого сообщения в другое приложение. Это может быть шифрование сообщений (для невозможности считывания данных при передаче), цифровая подпись (для защиты от преднамеренных изменений данных во время сообщения пути), настройка подписок (для отправки одного сообщения нескольким приложениям), определение метаданных для сообщений (для облегчения использования сообщений со сложной структурой содержимого) и другие. Для передачи сообщений может быть использована, например, Сервисная шина предприятия или ESB.

Сервисно-ориентированная архитектура (SOA) является логическим продолжением концепции веб-сервисов, которая заключается в публикации функциональных блоков любого приложения, позволяющего получить доступ к другому приложению через сеть. Веб (HTTP) в данном случае привлекателен благодаря возможности его использования и, следовательно, использованию функциональности опубликованных веб-приложений на любых аппаратных и программных платформах. Веб – службы-небольшая программная надстройка над функциональностью приложения, которая преобразует вызов, полученный через Интернет, во внутреннюю функцию приложения и возвращает результаты обратно.

Стоимость создания новых приложений на основе существующих веб-сервисов будет значительно ниже, чем разработка приложений с нуля или обширная интеграция с другими системами.

Например, у компании (оператора) есть Служба поддержки (служба технической поддержки клиентов) и биллинговая система (биллинговая

система). Перед компанией стоит задача создания новой системы «Личный кабинет абонента», в которой абонент мог бы через Интернет просматривать состояние своего аккаунта и сообщать о неисправности. Для этого компания должна создать «Личный кабинет» со своей собственной базой данных, которая синхронизирована с базой данных биллинговой системы и Службы поддержки, которая использует веб-сервисы «Подписчик» (опубликованная функциональность биллинговой системы) и «Создать запрос в техническую поддержку» (опубликованная функциональность системы службы поддержки). Очевидно, что вся работа над новым «Личным кабинетом» заключается в создании веб-пользовательского интерфейса на сайте компании.

Также распространенным подходом является интеграция пользовательских интерфейсов. Например, для создания приложений «одно окно». Самый простой пример кадров на веб-странице. Внутри каждого фрейма содержится отдельное веб-приложение. Благодаря рамкам все эти приложения появляются на экране одновременно. Пользовательские интерфейсы Веб-приложения легко интегрировать, однако существуют возможности интеграции и классических» «пользовательских интерфейсов и их фрагментов (ActiveX).

Наиболее целостным подходом к интеграции систем является интеграция на уровне бизнес-процессов. В рамках интеграции бизнес-процессов и интеграции приложений, а также интеграции данных и, что не менее важно, людей, участвующих в этом бизнес-процессе. Интеграция на уровне бизнес-процессов является наиболее естественной» «для организаций, поскольку их деятельность состоит в основном из бизнес-процессов, а не приложений, баз данных и платформ.

Идея, лежащая в основе интеграции бизнес-процессов, довольно проста:

- Чтобы составить сценарий бизнес-процесса в организации, опишите его, операции взаимодействия пользователя с различными системами между ними. Таким образом, бизнес-процесс — это элемент, который логически интегрирует различные системы. Сценарий создается с использованием специализированного программного обеспечения, которое продолжит управлять этим сценарием бизнес-процесса.
- Взаимодействие с системами в рамках бизнес-процесса подробно описано с точки зрения обмена информацией: форматы обмена, используемые службами, приложениями, событиями, правилами, политиками и т.д.
- Интегрирующее программное обеспечение, с помощью которого описывается сценарий бизнес-процесса, подключается через адаптеры интегрируемых систем, участвующих в бизнес-процессе. Таким образом, становится возможным автоматизированный обмен информацией между системами.
- Готовый к выполнению бизнес-процесс отображается на «пульте дистанционного управления» менеджера, с помощью которого он может запускать и останавливать бизнес-процессы, отслеживать их состояние, вводить данные и принимать решения по отдельным бизнес-процессам, требующим вмешательства человека. и взаимодействие

между системами, не требующими вмешательства человека, осуществляется автоматически.

ТЕМА 7. ЖИЗНЕННЫЙ ЦИКЛ ПРОГРАММНЫХ КОМПОНЕНТ ИС И АС

Создание приложений для ЭВМ или микроконтроллера может показаться очень простым процессом: вы открываете редактор или среду разработки, выполняете сборку приложения, выполняете отладку или тестирование, проверяете на различных устройствах и затем отправляете приложение в какой-либо Магазин приложений или размещаете на сайте. Все это занимает максимум несколько часов.

Или, наоборот, вы реализуете чрезвычайно сложную задачу с тщательным предварительным проектированием, тестированием на удобство использования, проверкой качества на множестве устройств и обслуживанием полного жизненного цикла бета-версии с последующим развертыванием различными способами.

Оба этих подхода к разработке и развертыванию имеют право на жизнь, но лучше сразу ориентироваться на мировые практики, которые основаны на принципах жизненного цикла приложений.

Рассмотрим далее основные этапы жизненного цикла кроссплатформенного приложения, принципы которого можно описать как:

- **Процесс** — процесс разработки ПО, называемым жизненным циклом разработки программного обеспечения (SDLC). Мы рассмотрим все стадии этого цикла в контексте разработки кроссплатформенных приложений: формирование идеи, проектирование, разработка, стабилизация, развертывание и обслуживание и так далее.
- **Рекомендации** — при создании кроссплатформенных приложений следует учитывать ряд моментов, которые отличают их от классических приложений и традиционных веб-приложений. Мы рассмотрим, как они влияют на разработку конечного продукта.

Исторически, жизненный цикл приложений регулируется международными стандартами IEEE Std 610.12, ISO/IEC 12207:2008 или отечественными стандартами ГОСТ 34.601-90 и ГОСТ Р ИСО/МЭК 12207-2010 Информационная технология. Системная и программная инженерия. Процессы жизненного цикла программных средств.

Классический жизненный цикл программных продуктов имеет множество этапов, но мы рассмотрим их именно через призму современных кроссплатформенных приложений:

- Формирование требований к программному продукту:
  - обследование объекта и обоснование необходимости создания КП;
  - формирование требований пользователя к КП;
  - оформление списка выполняемых работ и заявки на разработку КП.
- Разработка основной концепции КП:
  - изучение объекта разработки;
  - проведение патентных исследований;
  - проведение необходимых исследовательских работ;

- разработка вариантов концепции КП и выбор варианта концепции КП, удовлетворяющего требованиям будущих пользователей.
- оформление отчета о проделанной работе
- Оформление технического задания:
- разработка и утверждение технического задания на создание КП.
- Создание эскизного проекта:
- разработка предварительных проектных решений по системе и её частям;
- разработка документации на КП и её части.
- Создание технического проекта:
- разработка проектных решений по системе и её частям;
- разработка документации на КП и её части;
- разработка и оформление документации на поставку комплектующих изделий, программных библиотек и компонент;
- разработка заданий на проектирование в смежных частях проекта.
- Оформление рабочей документации:
- разработка рабочей документации на КП и её части;
- разработка и адаптация программ.
- Ввод КП в действие:
- подготовка КП;
- подготовка персонала;
- комплектация КП поставляемыми программными и техническими средствами;
- работы по размещению приложений на сервера;
- пусконаладочные работы;
- проведение предварительных испытаний в реальной обстановке;
- проведение опытной эксплуатации в реальной обстановке;
- проведение приёмочных испытаний;
- оформление методики испытаний.
- Тестирование КП.
- Сопровождение КП:
- выполнение работ в соответствии с гарантийными обязательствами (сопровождение);
- послегарантийное обслуживание приложения.

Если говорить более простым языком, все эти этапы можно свести к пятиосновным:

- Зарождение проекта.
- Проектирование, макетирование.
- Разработка.
- Стабилизация.
- Развертывание и распространение.

Многие из этих стадий часто перекрываются. Например, достаточно распространена практика, когда разработка идет уже на стадии окончательной

доводки пользовательского интерфейса и может влиять на этот процесс. Кроме того, приложение может возвращаться на стадию стабилизации при добавлении новых функций в очередную версию.

Это утверждение верно, так как сейчас разработка в основном ведется с применением «гибких методологий», и ее траектория имеет форму спирали с периодическим возвратом к предыдушим этапам.

Рассмотрим далее выделенные этапы более подробно.

Зарождение проекта

Сегодня цифровые устройства получили широчайшее распространение и проникли во все сферы жизни. Благодаря этому практически любой пользователь имеет возможность пользоваться необходимыми программными инструментами на любой выбранной платформе.

Особое внимание в этом ключе уделяется мобильным устройствам. Мобильные устройства открывают совершенно новый способ взаимодействия с вычислительными ресурсами, Интернетом и даже корпоративной инфраструктурой. В этой связи, актуальной является задача переноса алгоритмов и бизнес-логики приложений с персональной ЭВМ на мобильные устройства и покрытие максимального количества доступных платформ.

Сейчас ведущими платформами являются Android и iOS и большинство компаний желают иметь в своем арсенале приложения на обе эти платформы. Именно поэтому, выбор инструментов разработки приложений падает на кроссплатформенные, позволяющие достаточно легко и на одном и том же программном стеке реализовать и веб-приложений и мобильной.

На этапе зарождения формируется и дорабатывается идея самого приложения, а также определяются основные платформы для реализации. Для создания успешного приложения важно получить ответы на некоторые базовые вопросы. Ниже приводятся некоторые моменты, на которые следует обратить внимание перед публикацией приложения в одном из общедоступных магазинов приложений.

- Какие конкуретные преимущества у приложения? Есть ли аналоги? Чем приложение от них отличается? На каких платформах размещаются аналогичные решения и где будет максимально эффективное покрытие?
- В какую существующую инфраструктуру будет интегрироваться приложение и какие дополнительные возможности оно привнесет?
- Какие ценностные преимущества даст пользователям приложение?
- Как приложение будет использоваться? В каких ситуациях?
- Как это приложение будет работать с мобильными устройствами разных форм-факторов? Какие аппаратные возможности необходимы приложениям? Какие аппаратные ресурсы будет приложение использовать?

Чтобы упростить проектирование функций приложения, рекомендуется определить субъекты и варианты использования.

Субъекты использования — это роли в рамках приложения, которые часто соответствуют его пользователям.

Варианты использования — это типовые действия или цели.

Например, для интернет-магазина можно выделить два субъекта: продавец и покупатель. Продавец может создавать магазин, размещать товары, а покупатель искать товары и покупать товары. В этом случае достаточно определить две основные “главные” истории приложения и разрабатывать именно их, без рассеяния внимания на дополнительные возможности, по крайней мере на этапе проектирования и основной разработки.

Благодаря именно этой стадии Зарождения проекта в цикле разработки мы можем сосредоточиться на том, как реализовать приложение, а не на том, что оно должно делать.

Проектирование, макетирование приложения

После определения функций и функциональных возможностей приложения можно приступить к проектированию механизма взаимодействия с пользователем.

Взаимодействие с пользователем обычно реализуется на основе каркасов или макетов с помощью многочисленных наборов инструментов для проектирования. Применение макетов позволяет проектировать взаимодействие с пользователем, не заботясь о его фактической разработке.

Результатом этого этапа могут быть:

- Рабочий поток приложения (application flow).
- Варфреймы (wireframe).
- Прототип (Prototype).
- Мокапы (mockups).
- Дизайн (design).

Вайрфрейм — это низко детализированное представление дизайна. Он чётко должен показывать:

- Основные группы содержимого, распределенные по экранам.
- Информационную структуру приложения и заполнение интерфейса.
- Описание взаимодействия пользователя с интерфейсом и его примерную визуализацию.

Вайрфрейм — не просто набор серых блоков, это именно скелет дизайна, представляющий самые важные части проекта с внешней точки зрения.

Вайрфреймы должны создаваться быстро и большую часть этого времени следует провести за обсуждениями с командой и размышлениями.

Также часто вайрфреймы используются как составная часть документации по проекту.

Прототип иногда путают с варфреймом, хотя это средне или высоко детализированное представление конечного продукта, которое уже может имитировать реакцию на действия пользователя.

Прототипы позволяют оценить общую композицию интерфейса и протестировать основные методы работы с ним.

Мокап — это средне или высоко детализированное статичное представление дизайна, то есть переходной вариант между прототипом и дизайном.

Мокап представляет информационную структуру интерфейса в цвете, достаточным образом визуализирует контент и демонстрирует клиенту и разработчика базовую функциональность КП.

Дизайн — это законченное оформление проекта с учетом всех наработок предыдущих этапов. Подразумевает работу художника-оформителя и, конкретно, дизайнера.

На этом этапе также проводится стилизация и брэндинг (оформление под стиль заказчика).

Результатом работы на этапе дизайна является множество готовых для верстки экранов КП с полным оформлением и дополнительной информацией для верстальщиков. Под дополнительной информацией здесь будем понимать набор:

1. Внешних и внутренних отступов элементов, а также их смещений.
2. Размеры сеток контейнеров элементов.
3. Цветовые гаммы.
4. Наборы шрифтов и глифов.
5. Наборы фотографий, фонов, иконок.
6. Анимации и трансформации элементов интерфейса.

Особое внимание уделяется внешнему виду для различных разрешений экранов и различных устройств. Стандартно, мокапы и дизайны формируются для:

1. Экрана стандартного ПЭВМ (монитора).
2. Экрана переносимых ЭВМ (ноутбук с экранами до 13 дюймов).
3. Планшетов (вертикальная и горизонтальная ориентации).
4. Смартфонов (вертикальная и горизонтальная ориентации).
5. Смарт-часов и экранов встраиваемых устройств (если таковое подразумевается).

При изучении технологий проектирования и макетирования современные разработчики ориентируются на следующие основные источники:

1. Human Interface Guidelines от компании Apple (https://developer.apple.com/design/human-interface-guidelines/ios/overview/themes/).
2. Design for Android от Google (https://developer.android.com/design/index.html).
3. Design basics for Windows apps от компании Microsoft (https://docs.microsoft.com/en-au/windows/uwp/design/basics/).
4. Create intuitive and beautiful products with Material Design так же от компании Google (https://material.io/design).

Другим важным элементом является используемые фреймворки для стилизации приложений.

Дело в том, что интерфейс кроссплатформенных приложений в основном разрабатывается либо на чистой комбинации HTML + CSS3 + JavaScript, либо с использованием какого-либо фреймворка.

Первый вариант позволяет сделать проект “с нуля” и так как это требуется в деталях, но занимает намного больше времени, чем второй вариант.

Второй вариант позволяет быстро сверстать интерфейс с применением готовых блоков и шаблонов, но накладывает определенные ограничения на использование стилей и элементов разметки. Это обусловлено тем, что каскадные таблицы стилей накладываются друг на друга и не всегда можно гарантировать, особенно в условиях недостаточного опыта разработки, как поведет себя тот или иной компонент на экране.

Классический CSS фреймворк – набор базовых стилей для вёрстки веб-страницы, включающий в себя следующие элементы:

- сетка;
- иконки;
- таблицы;
- элементы форм и кнопок;
- типографика;
- интерфейсные паттерны, например карточки и модальные окна;
- вспомогательные классы оформления элементов: отступы, цвета и т. д.

Сейчас популярны следующие фреймворки для быстрой разработки интерфейсов:

1. **Bootstrap** (https://getbootstrap.com/). Известен проработанной адаптивной сеткой, большим количеством готовых решений, полной документацией и низким порогом вхождения.
2. **Foundation** (https://foundation.zurb.com/). Второй по популярности фреймворк. Очень гибок и подходит для крупных проектов (им пользуются Facebook, eBay, Mozilla, Adobe, HP, Cisco и Disney).
3. **Bulma** (https://bulma.io/). Небольшой, гармонично построенный, отзывчивый и удобный, понятный интуитивно фреймворк (см. рис. 4). Написан на чистом CSS.
4. **Semantic UI** (https://semantic-ui.com/). Большой проект, имеет более 3000 настраиваемых переменных и 50 компонентов для создания сайтов.
5. **Materialize CSS** (https://materializecss.com/). Создан компанией Google в 2014 и до сих пор занимает лидирующие места в гонке фреймворков (см. рис. 6). Предлагает набор готовых к использованию компонентов в стиле Material Design.

Разработка

Стадия разработки, как правило, начинается достаточно рано. Фактически, после формирования и созревания идеи на стадии зарождения уже подготавливается рабочий прототип, который позволяет оценить

функциональные возможности, предположения и получить общее понимание предстоящего объема работы.

Разработка может начинаться даже раньше, чем готов макет и прототип. Конечно, разработка здесь ведется только в области каких-то общих моментов.

Например, если мы знаем, что будет использоваться авторизация пользователей на основе JWT (JSON Web Token, https://jwt.io/), то мы можем уже выстраивать логику входа и регистрации клиентов, профили пользователей, алгоритмы валидации регистрации, функции уведомлений п электронной почте и так далее. В этом случае нам вообще не нужен интерфейс - нам нужен только программный код, алгоритмы и средства отладки.

Процесс разработки может занимать от нескольких недель до нескольких лет, все зависит от желаемых функций КП и от возможностей разработчиков.

Стабилизация

Стабилизации — это процесс устранения ошибок в приложении. При этом ошибки могут быть не только функциональными, например возникающими при нажатии определенной кнопки, но и относиться к удобству использования и производительности приложения в целом.

Кроме того, стабилизация позволяет зафиксировать приложение в каком-то определенном состоянии.

Обычно выделяют несколько этапов стабилизации:

1. Прототип — приложение находится на стадии экспериментальной проверки концепции. При этом работают только его отдельные основные функции или части. На этом этапе в приложении могут присутствовать ошибки, что-то еще не реализовано, что-то работает не так, как ожидается. Это состояние нужно для оценки возможностей КП и для демонстрации.
2. Альфа-версия (закрытая или публичная) — обычно готов код основных функциональных возможностей, однако его полное тестирование еще не завершено. На этом этапе по-прежнему присутствуют ошибки и могут отсутствовать некоторые дополнительные функции.
3. Бета-версия (закрытая или публичная) — большая часть функциональных возможностей завершена и прошла хотя бы предварительное тестирование с устранением ошибок. На этом этапе могут по-прежнему присутствовать основные известные проблемы.
4. Версия-кандидат (релиз) — все функциональные возможности завершены и прошли тестирование. За исключением новых ошибок, приложение готово к выпуску.

В современном процессе разработки стабилизация обычно занимает фиксированные периоды времени, например релизы с мажорной версией проводят раз в два месяца. К этому времени уже тестируется несколько релизов с минорными версиями.

Тестирование приложения никогда не может быть начато слишком рано. Например, если на стадии прототипа обнаружена серьезная проблема, вы по-

прежнему сможете изменить механизм взаимодействия с пользователем, чтобы устранить ее.

Как правило, по мере продвижения по этапам жизненного цикла все больше людей получают доступ к нему с возможностью опробовать, протестировать, поделиться своими отзывами и т. д. Например, доступ к прототипам приложений обычно получают только ключевые заинтересованные лица, тогда как версия-кандидат может распространяться среди клиентов, которые зарегистрировались в программе раннего доступа.

На начальных этапах тестирования и развертывания на относительно небольшом количестве устройств обычно достаточно выполнить развертывание непосредственно с компьютера разработчика. Тем не менее по мере расширения аудитории приложения этот процесс заметно усложняется.

Развертывание и распространение

Это сложный процесс, который обычно требует автоматизации. Для развертывания (deploy) и распространения (distribution) зачастую применяются такие инструменты, как:

1. Gitlab CI/CD.
2. Bitbucket Pipelines.
3. Travis CI.
4. Jenkins.
5. Kubernetes.

Они позволяют автоматизировать сборку приложений, проведение тестов, автодокументирование кода, проверка его соответствия стандартам написания, компиляцию, а также формируют специальные образы для дальнейшей загрузки на сервера размещения по зашифрованным каналам связи.

Чем более автоматизирован процесс сборки, развертывания и распространения, тем быстрее будет доставлено приложение до клиента или заказчика.

ТЕМА 8. ЖИЗНЕННЫЙ ЦИКЛ АППАРАТНЫХ ПЛАТФОРМ ИС И АС

Жизненный цикл аппаратных платформ ИС и АС несколько отличается от жизненного цикла программного обеспечения, так как это все-таки устройства, содержащие электронные компоненты, требующие промышленного изготовления и отдельного контроля.

Рынок аппаратных систем всегда был зрелым, постоянно развивающимся и популярным выбором для организаций различного уровня. Что нового сегодня, - так это внезапный спрос на встроенные системы среди компаний, разбирающихся в цифровых технологиях нового века, которые увеличили оценку отрасли изготовления и потребления устройств до более чем 500 млрд долларов США. Электротехнические и машиностроительные компании, специализирующиеся на новейших технологиях разработки оборудования, постоянно увеличивают обороты разработки, производства и продаж новых схем, особенно для мобильных устройств и носимой электроники (элементы Умного дома и Интернета Вещей).

Несмотря на то, что возможности для будущих продуктов огромны, комплексная разработка аппаратных систем сама по себе является сложным проектом. Несмотря на то, что системы Интернета Вещей состоят из аппаратных, встроенных и программных компонентов, 80% затрат и хлопот по разработке приходится на аппаратное и встроенное программное обеспечение встроенной системы.

Поэтому компании по проектированию и разработке продуктов должны следовать комплексному гибкому подходу для безопасной и своевременной реализации своих сборок на основании следующих шагов жизненного цикла.

Шаг 1: Осуществимость

Перво-наперво следует записать идею продукта, определить его назначение и сферы сбыта (будущих клиентов).

В описание включаются такие детали, как функции, конфигурации, целевой клиент, разрыв на рынке, который вы пытаетесь преодолеть, конкурирующие продукты, существующий набор навыков и количество необходимых ресурсов в команде, а также объем капитала, который можно выделить на разработку.

Предполагаем, что команда (технический архитектор, дизайнер решений, исследователь рынка, составитель бюджета) уже есть, следует чтобы сузить основную область деятельности чтобы сосредоточиться на основных моментах разработки и создать технико-экономическое обоснование.

Идея технико-экономического обоснования заключается в формировании критериев достижения Минимального Жизнеспособного продукта (MVP), который будет разработан на последующих этапах и потом превратится в прототип и конечный продукт.

В первую очередь следует четко и подробно изложить приоритеты продукта. Нужно перечислить функциональные возможности продукта, за

которыми следуют расширенные функции. Быстрый способ закрепить технико-экономическое обоснование заключается в разборке известных продуктов в аналогичном сегменте.

Осуществимость проекта имеющимися средствами может быть критическим этапом любого проекта по разработке аппаратного обеспечения. Чем больше вы проводите мозговой штурм и чем больше общаетесь с потенциальными реальными пользователями вместе с MVP, тем меньше вы страдаете на более поздних этапах. Исследуйте и запишите свои варианты использования в таблице данных. Добавление ресурса контроля качества на этом этапе является хорошей идеей для расширения возможностей тестовых случаев.

Шаг 2. Предварительное Проектирование Оборудования

Предварительное проектирование выполняется для устранения пробелов между концепцией дизайна и фактическим дизайном. Начните с блок – схемы системного уровня, чтобы указать все электронные функции и их взаимосвязь с другими функциональными компонентами. Для любого аппаратного продукта микроконтроллер, микропроцессор или «центральное ядро» является основным компонентом, который связан с другими компонентами, такими как дисплеи, датчики, микросхемы памяти и т.д.

Это отправная точка для разработки устройств на Шаге 2 жизненного цикла аппаратного проекта.

Например, на схемах следует указать тип и количество всех портов, необходимых для ядра, а на основе спецификации микроконтроллера следует определить все необходимые сопутствующие компоненты, такие как датчики, дисплеи, разъемы и микросхемы.

За этим должно последовать тщательное исследование компонентов с целью получения **Спецификации материалов (СПЕЦИФИКАЦИИ)** к концу этого этапа.

Далее можно переходить к разработке принципиальной схемы устройства.

Для начала следует взять за основу структурную схему системы и использовать ее в качестве отправной точки для проектирования Принципиальной схемы.

В то время как Структурная схема системы в основном фокусируется на высокоуровневой функциональности продукта, Принципиальная схема отмечает мельчайшие детали микросхем, резисторов, датчиков и других компонентов, соединенных вместе, образуя функциональную схему.

Там же указываются электрические характеристики и некоторые детали конструкции.

Для более эффективной работы используется подход Иерархического проектирования, когда создаются отдельные подсхемы для каждого блока на структурной схеме системы.

После того как создали все отдельные компоненты схемы, следует их соединить согласно правилам, изложенным в документации на компоненты, чтобы сформировать полную принципиальную схему.

Для повышения эффективности рекомендуется использовать программное обеспечение для проектирования схем, такое как DipTrace, Altium и KiCad.

Эти инструменты имеют встроенные библиотеки, которые очень полезны в случае, если вы используете популярные компоненты в своем дизайне. Структурная схема системы должна быть в состоянии указать вам точные характеристики требуемых компонентов.

Далее можно переходить к Проектированию механических и промышленных компонентов схемы.

Механическое проектирование в основном выполняется на инструментах САПР, но оно следует всем общепринятым принципам для достижения полностью совместимого результата. В зависимости от вашего бюджета и типа продукта выберите свой инструмент САПР. Если вы работаете над проектом интернета вещей, то OnShape, Fusion 360 и FreeCAD являются надежными вариантами.

Как и во всех видах проектной деятельности, изложение цели проекта является здесь первым шагом. Цель здесь состоит в том, чтобы определить значение, функции и внешний вид всех компонентов в общей системе.

После определения всех компонентов в общей схеме, следует оценить возможность нормального взаимодействия компонентов друг с другом (например, совместимость по электрическим и тепловым характеристикам, уровень шумов, цепи питания, уровни напряжения и т.д.).

Независимо от того, работают ли они беспрепятственно друг с другом в едином корпусе, следует отдельно проработать возможности подключения и совместного использования. Это особенно актуально для мобильных или переносимых устройств, когда вся работа ведется от аккумулятора и отдельные компоненты могут потреблять больше энергии, чем вся остальная плата (например, цветной экран может иметь потребляемую мощность в несколько раз выше, чем микропроцессор, память и окружающие их элементы).

Ниже приведены ключевые этапы в области механического и промышленного дизайна

- Геометрическое моделирование — это математическое представление объекта с использованием специальных графических средств.
- Инженерный анализ - неважно, насколько мало или надежно устройство, но все его компоненты подвержены той или иной степени нагрузки в ходе эксплуатации. На этом этапе анализируется соответствие всех компонентов общей системе на основе их уровней напряжения и выносливости
- Осуществимость проекта оценивается в соответствии с такими показателями, как стоимость и масштаб.
- Осуществимость проекта проверяется в соответствии с производственными спецификациями.
- Вам также необходимо проверить совместимость электрического оборудования с механическим корпусом и деталями.

Далее можно переходить к проектированию печатной платы (PCB) аппаратного устройства.

Правило здесь соблюдается простое – чем меньше изделие, тем плотнее компоненты и, следовательно, сложнее создать компоновку печатной платы.

Если продукт потребляет больший ток и обеспечивает беспроводную связь, процесс проектирования становится более сложным.

С другой стороны, миниатюризация печатной платы и снижение общего энергопотребления схемы являются не менее сложными задачами.

Перед проектированием схемы проверьте следующие факторы – маршрутизацию питания, тип и точность кварцевого генератора тактовых сигналов, адресные линии или линии передачи данных, другие высокоскоростные сигналы и соединения, — все это увеличивает сложность печатной платы, не говоря уже о радиочастотной секции беспроводных схем.

Проектирование производство устройств с «радио» — это вообще отдельная тема для исследования и изучения.

Для проектирования физической платы всех электронных компонентов используйте инструмент проверки, чтобы сопоставить схематическое представление с процессом создания идеализированной печатной платы.

Несколько популярных имен включают в себя исполнителя печатных плат, печатную плату Solidworks, дизайнера Altium и т.д.

Большинство из этих инструментов поставляются с диагностическими алгоритмами, такими как проверки DRC, и моделированием для обнаружения любых ошибок. И последнее, но не менее важное: физическая площадь печатной платы должна быть совместима с определенными ранее требованиями.

Примером может являться устройство, которое встраивается в другой прибор – там ему нередко выделяется пространство, измеряемое в «литрах» или в кубических сантиметрах, причем сложной формы. Задача разработчиков – выполнить проект так, чтобы уложиться в заявленный размер и форму.

После этого можно переходить к Созданию окончательной спецификации (спецификации).

Спецификация – это специальным образом оформленный список всех номенклатур механических и электронных компонентов, которые необходимо приобрести и смонтировать в устройство.

Независимо от того, насколько дешев или мал компонент, нужно перечислить их все, а затем указать их количество и основные технические характеристики.

Большинство программных средств схематического проектирования автоматически заполняют спецификацию, но этот процесс нужно контролировать. Лучше все равно проверить документ вручную, прежде чем обращаться к поставщику для закупки деталей.

По завершении этих мероприятий нужно обратиться за помощью к надежному партнеру-поставщику деталей. Источники в основном игнорируются, но все же это критическая фаза в любом аппаратном проекте. На самом деле, поиск партнера по оборудованию — это «проект» сам по себе и может занять больше времени, чем планировалось изначально.

Особенно если закупки выполняются при помощи конкурсов.

Тем не менее, нужно изучить цифровой рынок для всех владельцев проектов, чтобы, например, сотрудничать с другими поставщиками, производственными и другими партнерами. Такие платформы не только предоставляет лучшие в своем классе услуги и продукты, но и позволяют владельцам проектов определять свои требования.

Шаг 3: Создание прототипа

Прототипирование — это мост между вашим аппаратным продуктом на бумаге и реальной сборкой, которая когда-нибудь появится на полке рынка. Не торопитесь и сначала сосредоточьтесь на создании простого инженерного прототипа. Цель этого этапа-оценить осуществимость уже разработанных функций. Предпочтительно передать производство печатных плат и сборку компонентов на аутсорсинг надежному производителю печатных плат, который предлагает производство небольших объемов. Помимо экономии времени и усилий, это обеспечит безупречное прототипирование печатных плат.

Мы можем выполнить сборку печатных плат самостоятельно, если у вас есть квалифицированные специалисты, и только в том случае, если конструкция проста.

Можно использовать быстрое прототипирование механических деталей, также известное как 3D-печать. Этот подход быстро набирает популярность, и многие небольшие и средние аппаратные проекты используют его. Например, можно «печатать» корпуса устройств и массо-габаритные модели.

Хотя это все еще дорогостоящее дело, тем не менее, большинство компаний предпочитают его за точную физическую сборку, которую он создает для лучшего анализа. Обратитесь к надежному партнеру по 3D - моделированию, который поможет вам в этом процессе. Обычно это делается путем преобразования 3D-модели (из инструмента САПР) в файл STL, а затем ее разрезания на цифровые слои. С помощью специального программного обеспечения компьютера файл STL передается на принтер. В итоге устанавливается принтер с соответствующими параметрами и загружаются расходные материалы.

В конце этапа нужно оценить полностью интегрированную сборку на предмет внешнего вида, возможности прототипирования, технической осуществимости, операционной точности, готовности к проектированию и восприимчивости к рынку.

Прототипирование будет исчерпывающим и должно выполняться под полным наблюдением промышленных дизайнеров, инженеров-механиков и разработчиков приложений и прикладных программистов для кросс-функциональных программных планов.

Шаг 4: Проектирование для производства и сборки (DFMA)

DFMA является важным шагом перед началом крупносерийного производства. Это упрощает сложность производства и тем самым снижает любые накладные расходы. Это помогает оптимизировать общую стоимость производственных компонентов. DFA сократила время сборки изделия за счет

минимизации количества этапов сборки. В современных производственных условиях обе фазы объединены в DFDA. Ключевыми соображениями для выполнения DFDA являются следующие:

- Консультационные производственные эксперты предоставляют информацию о снижении производственных затрат путем анализа каждого компонента.
- Определение материалов, соответствующих требованиям производства, которые контролируют стоимость, не влияя на качество сборки.
- Следуя всем законодательно установленным производственным процессам
- Использование всех стандартизированных деталей позволяет избежать каких-либо сложностей с инвентаризацией.

Нужно получить отзывы обо всех электрических и механических конструкциях от партнеров-производителей. Необходимо уделять пристальное внимания обратной, так как игнорирование этого процесса в конечном итоге приводит к увеличению затрат, задержкам в производстве и т.д.

Шаг 5: Производство

Прежде чем приступить к массовому производству, первым в списке дел должен быть поиск надежного партнера по производству.

Это важное соображение, которое большинство владельцев проектов упускают из виду, но «раскаиваются» об этом позже.

Возможно, потребуется обучить производителя процессу создания нужного продукта в качестве прототипа, так как это может быть новый тип устройств для него. На этом этапе также можно определить и решить любые оставшиеся проблемы в дизайне для производимой конструкции и внести ряд корректив.

Этот этап называется Проверка дизайна (Design Verification) и это шанс оценить производственные настройки и инструменты, которые будут использоваться на каждом шаге изготовления прототипа. Возможно придется произвести от 50 до 100 единиц (тестовых образцов) для того, чтобы провести все испытания и оценить характеристики конечного продукта.

Проверка процесса (Process Verification).

После завершения Process Verification следует приступить к самой проверке устройства. На этом этапе инженерный проект, производственные тестовые системы и процессы оцениваются на предмет готовности к следующему этапу массового производства. Проверка процесса иногда ассоциируется с проверкой самого производства, на котором будет создаваться первая опытная партия.

В процессе изготовления также желательно выполнить несколько тестов сборки, чтобы оценить, готов ли производственный процесс к массовому производству.

Этот этап используется для выявления и устранения всех проблем в процессе проектирования и производства, которые могут привести к непредвиденному отказу продукта в будущем.

Это комплексный этап, и он может длиться до 6 месяцев.

В это время инженеры, дизайнеры и консультанты по производству оценивают надежность и другие эксплуатационные характеристики изделия в течение определенного периода использования, а также в определенных условиях эксплуатации.

Для проверки используются специальные стенды – испытания на них дают общее представление о качестве продукции, а также служат основой для получения различных сертификатов.

Как только ваши тестировщики официально утвердят качество изготовленного прототипа, нужно подготовить линии и все производство вообще к большим объемам, чтобы уложиться в сроки «выхода на рынок».

В это время логично обратиться в регулирующие органы власти для получения сертификата продукта для конкретных регионов и групп населения.

Также хорошей практикой является получение лицензий, патентов авторских свидетельств.

ТЕМА 9. ПРИНЦИПЫ ОБЕСПЕЧЕНИЯ КАЧЕСТВА ПРОГРАММНОГО И АППАРАТНОГО ОБЕСПЕЧЕНИЯ ИС И АС

Понятие качества относится почти к любому продукту, будь то физический объект или часть программного обеспечения. Веб-сайт, который вы найдете в Интернете, поначалу может показаться прекрасным, но при прокрутке вниз, переходе на другую страницу или попытке отправить запрос на контакт он может начать показывать некоторые недостатки и ошибки дизайна.

Это делает контроль качества настолько важным во всех областях, где создается продукт конечного пользователя. Однако некачественно выполненный сайт не нанесет такого ущерба, как движущийся самостоятельно на автопилоте автомобиль с плохим качеством программного обеспечения. Одна ошибка в системе управления может поставить под угрозу жизнь граждан. Но даже некачественный веб-сайт электронной коммерции, у которого проблемы с производительностью или безопасностью, может стоить владельцу миллионов рублей дохода.

Именно поэтому нормальные компании придают большое значение качеству программного обеспечения, которое создается для клиентов. В этой лекции мы рассмотрим процесс обеспечения качества и тестирования к программному и аппаратному обеспечению и современные предпочтительные стратегии.

Концепция качества программного обеспечения

Чтобы убедиться, что выпущенное программное обеспечение безопасно и функционирует должным образом, была введена концепция качества программного обеспечения. Это часто определяется как «степень соответствия явным или неявным требованиям и ожиданиям». Эти так называемые явные и неявные ожидания соответствуют двум основным уровням качества программного обеспечения.

- Функциональность – соответствие продукта функциональным (явным) требованиям и техническим характеристикам. Этот аспект фокусируется на практическом использовании программного обеспечения, с точки зрения пользователя: его функциях, производительности, простоте использования, отсутствии дефектов.
- Нефункциональный – внутренние характеристики и архитектура системы, т. е. структурные (неявные) требования. Это включает в себя удобство обслуживания кода, понятность, эффективность и безопасность.

Структурным качеством программного обеспечения, как правило, трудно управлять: оно в основном полагается на опыт инженерной команды и может быть обеспечено с помощью анализа, анализа и рефакторинга кода. В то же время функциональный аспект может быть обеспечен с помощью комплекса

специальных мероприятий по управлению качеством, которые включают обеспечение качества, контроль качества и тестирование.

Часто используемые взаимозаменяемо, эти три термина относятся к несколько различным аспектам управления качеством программного обеспечения. Несмотря на общую цель предоставления продукта наилучшего качества, как структурно, так и функционально, они используют разные подходы к этой задаче.

Обеспечение качества (QA)

Обеспечение качества — это широкий термин, который объясняется в блоге Google по тестированию как «непрерывное и последовательное совершенствование и поддержание процесса, обеспечивающего контроль качества». Как следует из определения, контроль качества больше фокусируется на организационных аспектах управления качеством, отслеживая последовательность производственного процесса.

Посредством контроля качества команда проверяет соответствие продукта функциональным требованиям. По определению Investopedia, это “процесс, посредством которого бизнес стремится обеспечить поддержание или улучшение качества продукции, а также сокращение или устранение производственных ошибок”. Это действие применяется к готовому продукту и выполняется до выпуска продукта. С точки зрения обрабатывающей промышленности это похоже на то, как если бы вы вытащили случайный товар с конвейера, чтобы проверить, соответствует ли он техническим характеристикам.

Контроль качества (QC) и тестирование

Тестирование — это основная деятельность, направленная на выявление и решение технических проблем в исходном коде программного обеспечения и оценку общего удобства использования, производительности, безопасности и совместимости продукта. Он имеет очень узкую направленность и выполняется инженерами-тестировщиками параллельно с процессом разработки или на специальной стадии тестирования (в зависимости от методологического подхода к циклу разработки программного обеспечения).

Процесс:	**QA**	**QC**	**Тестирование**
Цель	Создание адекватных процессов, внедрение стандартов качества для предотвращения ошибок и недостатков в продукте	Проверка того, что продукт соответствует требованиям и спецификациям, прежде чем он будет выпущен	Обнаружение и устранение программных ошибок и недостатков

Фокус	На процессе	На продукте вцелом	На исходном коде и дизайне
Что выполняется?	Профилактика критических ситуаций	Верификация выполненных работ	Обнаружение частных ошибок кода
Кто выполняет?	Команда, включая заказчиков	Команда	Инженеры по тестированию, разработчики
Когда?	В течение всего процесса разработки	Перед релизом продукта	На стадии тестирования или прямо во время процесса разработки

Применительно к процессу производства автомобилей наличие надлежащего процесса обеспечения качества означает, что каждый член команды понимает требования и выполняет свою работу в соответствии с общепринятыми руководящими принципами. А именно, он используется для обеспечения того, чтобы каждое отдельное действие выполнялось в правильном порядке, каждая деталь была должным образом реализована, а общие процессы согласованы, чтобы ничто не могло оказать негативное влияние на конечный продукт.

Контроль качества можно сравнить с тем, как старший менеджер заходит в производственный отдел и выбирает случайный автомобиль для осмотра и тест-драйва. Мероприятия по тестированию в этом случае относятся к процессу проверки каждого соединения, каждого механизма в отдельности, а также всего продукта, будь то вручную или автоматически, проведения краш-тестов, тестов производительности и реальных или имитируемых тест-драйвов.

Благодаря такому практическому подходу деятельность по тестированию программного обеспечения остается предметом жарких дискуссий. Именно поэтому в данной статье мы сосредоточимся в первую очередь на этом аспекте управления качеством программного обеспечения. Но прежде, чем мы перейдем к деталям, давайте определим основные принципы тестирования программного обеспечения.

Основные принципы тестирования программного обеспечения

Сформулированные за последние 40 лет семь принципов тестирования программного обеспечения представляют собой основные правила этого процесса.

- Тестирование показывает наличие ошибок. Тестирование направлено на выявление дефектов в программном обеспечении. Но независимо от того, насколько тщательно тестируется продукт, мы никогда не можем быть на 100 процентов уверены в отсутствии дефектов. Мы можем использовать тестирование только для уменьшения числа необоснованных проблем.

- Исчерпывающее тестирование невозможно. Невозможно проверить все комбинации входных данных, сценариев и предварительных условий в приложении. Например, если один экран приложения содержит 10 полей ввода с 3 возможными вариантами значений каждое, это означает, что для покрытия всех возможных комбинаций инженерам по тестированию потребуется создать несколько сотен сценариев тестирования. А что, если приложение содержит более 50 таких экранов? Чтобы не тратить недели на создание миллионов таких менее возможных сценариев, лучше сосредоточиться на потенциально более значимых.
- Раннее тестирование. Как упоминалось выше, стоимость ошибки растет экспоненциально на всех этапах разработки. Поэтому важно как можно скорее начать тестирование программного обеспечения, чтобы обнаруженные проблемы были устранены и не превратились в снежный ком.
- Кластеризация дефектов. Этот принцип часто называют применением принципа Парето к тестированию программного обеспечения. Это означает, что примерно 80 процентов всех ошибок обычно обнаруживаются только в 20 процентах системных модулей. Поэтому, если в определенном модуле программного обеспечения обнаружен дефект, есть вероятность, что могут быть и другие дефекты. Вот почему имеет смысл тщательно протестировать эту область продукта.
- Парадокс «пестицидов». Повторное выполнение одного и того же набора тестов не поможет вам найти больше проблем. Как только обнаруженные ошибки будут исправлены, эти тестовые сценарии станут бесполезными. Поэтому важно регулярно пересматривать и обновлять тесты, чтобы адаптироваться и потенциально находить больше ошибок.
- Тестирование зависит от контекста. В зависимости от их назначения или отрасли различные приложения должны тестироваться по-разному. Хотя безопасность может иметь первостепенное значение для продукта fintech, она менее важна для корпоративного веб-сайта. Последнее, в свою очередь, делает акцент на удобстве использования и скорости.
- Ошибка отсутствия ошибок. Полное отсутствие ошибок в вашем продукте не обязательно означает его успех. Независимо от того, сколько времени вы потратили на полировку кода или улучшение функциональности, если ваш продукт бесполезен или не соответствует ожиданиям пользователей, он не будет принят целевой аудиторией.

Хотя вышеперечисленные принципы являются бесспорными руководящими принципами для каждого специалиста по тестированию программного обеспечения, необходимо учитывать и другие аспекты. Некоторые источники отмечают другие принципы в дополнение к основным:

- Тестирование должно быть независимым процессом, которым должны заниматься беспристрастные профессионалы.
- Проверьте наличие недопустимых и неожиданных входных значений, а также допустимых и ожидаемых.
- Тестирование должно проводиться только на статической части программного обеспечения (в процессе тестирования не следует вносить никаких изменений).
- Используйте исчерпывающую и исчерпывающую документацию для определения ожидаемых результатов испытаний.

Роль тестирования в жизненном цикле разработки программного обеспечения.

Модель Водопада

Представляя собой традиционный жизненный цикл разработки программного обеспечения, модель водопада включает в себя 6 последовательных этапов: планирование, анализ, проектирование, внедрение, тестирование и техническое обслуживание.

На этапе тестирования продукт, уже разработанный и закодированный, проходит тщательное тестирование перед выпуском. Однако практика показывает, что ошибки и дефекты программного обеспечения, обнаруженные на этом этапе, могут быть слишком дорогостоящими для исправления, поскольку стоимость ошибки, как правило, увеличивается на протяжении всего процесса разработки программного обеспечения.

Например, если в спецификациях есть ошибка, ее обнаружение на ранней стадии планирования не приведет к значительным потерям для вашего бизнеса. Однако ущерб растет экспоненциально на всех последующих стадиях процесса. Если такая ошибка будет обнаружена на этапе проектирования, вам нужно будет переработать свои проекты, чтобы исправить ее. Но если вы не сможете обнаружить ошибку до создания продукта, вам, возможно, потребуется внести некоторые серьезные изменения в дизайн, а также в исходный код. Это потребует значительных усилий и инвестиций.

То же самое относится и к ошибкам, возникающим в процессе внедрения. Если в логике какой-либо функции есть изъян, расширение функциональности поверх нее может нанести серьезный ущерб в долгосрочной перспективе. Поэтому лучше протестировать каждую функцию, пока продукт еще находится в стадии разработки. Именно здесь итеративные гибкие методы оказываются полезными.

Гибкое тестирование

Являясь неотъемлемой частью процесса разработки программного обеспечения, Agile разбивает процесс разработки на более мелкие части, итерации и спринты. Это позволяет тестировщикам работать параллельно с остальной частью команды на протяжении всего процесса и устранять недостатки и ошибки сразу после их возникновения.

Основная цель такого процесса заключается в быстрой и качественной поставке новых функций программного обеспечения. Поэтому этот подход менее затратен: исправление ошибок на ранних стадиях процесса разработки, до того, как возникнет больше проблем, значительно дешевле и требует меньше усилий. Кроме того, эффективная коммуникация внутри команды и активное участие заинтересованных сторон ускоряют процесс и позволяют принимать более обоснованные решения. Вы можете узнать больше о ролях и обязанностях в команде тестирования в специальных статьях по данной тематике.

Подход к гибкому тестированию больше связан с созданием практики контроля качества, а не с созданием команды контроля качества. Создавая команду контроля качества, мы часто рискуем отделить тестировщиков от жизненно важных бесед с владельцами продуктов, разработчиками и т.д. В гибких проектах контроль качества должен быть встроен в команды scrum, потому что тестирование и качество не являются второстепенной мыслью. Качество должно быть заложено с самого начала.

Тестирование DevOps

Для тех, у кого есть гибкий опыт, DevOps постепенно становится обычной практикой. Эта новая методология разработки программного обеспечения требует высокого уровня координации между различными функциями цепочки поставок, а именно разработкой, контролем качества и операциями.

DevOps часто называют расширением Agile, которое устраняет разрыв между разработкой, контролем качества и операциями. Однако, в отличие от Agile, DevOps включает концепцию непрерывной разработки, при которой код, написанный и переданный для контроля версий, будет собран, развернут, протестирован и установлен в производственной среде, готовой к использованию конечным пользователем. DevOps уделяет большое внимание автоматизации и инструментам непрерывной интеграции, которые обеспечивают высокоскоростную доставку приложений и услуг.

Тот факт, что тестирование происходит на каждом этапе в модели DevOps, меняет роль тестировщиков и общую идею тестирования. Поэтому, чтобы эффективно проводить тестирование, от тестировщиков теперь ожидается, что они будут обладать техническими навыками и даже разбираться в коде.

Согласно опросу PractiTest, тенденция Agile является бесспорным лидером, в то время как почти 90 процентов респондентов работают, по крайней мере, в некоторых гибких проектах в своих организациях. Тем не менее, треть респондентов все еще применяет модель водопада в некоторых проектах, несмотря на неуклонное сокращение использования этого метода. DevOps продолжает расти, только медленнее, чем раньше.

Организация тестирования на практике

Планирование тестирования: Артефакты и стратегия

Как и любому другому официальному процессу, мероприятиям по тестированию, как правило, предшествуют тщательная подготовка и планирование. Основная цель этого этапа-убедиться, что команда понимает цели клиента, основное назначение продукта, возможные риски, которые им необходимо устранить, и результаты, которых они ожидают достичь. Один из документов, созданных на этом этапе, миссия или задание на тестирование, служит для решения этой задачи. Это помогает согласовать мероприятия по тестированию с общей целью продукта и координирует усилия по тестированию с остальной работой команды.

Роджер С. Прессман, профессиональный инженер-программист, известный автор и консультант, утверждает: «Стратегия тестирования программного обеспечения представляет собой дорожную карту, в которой описываются шаги, которые необходимо выполнить в рамках тестирования, когда эти шаги планируются и затем выполняются, и сколько потребуется усилий, времени и ресурсов».

Стратегия тестирования, также называемая подходом к тестированию или архитектурой, является еще одним артефактом этапа планирования. Джеймс Бах, гуру тестирования, создавший курс быстрого тестирования программного обеспечения, определяет цель стратегии тестирования как «прояснение основных задач и проблем тестового проекта». Хорошая стратегия тестирования, по его мнению, специфична для конкретного продукта, практична и оправдана.

В зависимости от того, когда именно в процессе они используются, стратегии могут быть классифицированы как превентивные или реактивные.

Хотя стратегия тестирования-это документ высокого уровня, план тестирования имеет более практический подход, подробно описывающий, что тестировать, как тестировать, когда тестировать и кто будет проводить тестирование. В отличие от документа статической стратегии, который относится к проекту в целом, план тестирования охватывает каждый этап тестирования отдельно и часто обновляется руководителем проекта на протяжении всего процесса.

В соответствии со стандартом IEEE для документации по тестированию программного обеспечения документ плана тестирования должен содержать следующую информацию:

- Идентификатор плана тестирования.
- Введение.
- Список литературы (список сопутствующих документов).
- Тестовые задания (продукт и его версии).
- Функции, подлежащие тестированию.
- Функции, не подлежащие тестированию.
- Критерии прохождения или отказа элемента.

- Подход к тестированию (уровни, типы, методы тестирования).
- Критерии приостановления.
- Результаты (План тестирования (сам этот документ), Тестовые случаи, Сценарии Тестирования, Журналы Дефектов/Улучшений, Отчеты о тестировании).
- Среда тестирования (оборудование, программное обеспечение, инструменты).
- Оценки.
- Расписание тестирования.
- Потребности в кадрах и профессиональной подготовке.
- Обязанности разработчиков в процессе тестирования.
- Риски.
- Допущения и зависимости.
- Утверждения.

Написание плана, который включает в себя всю перечисленную информацию, является трудоемкой задачей. В гибких методологиях, где основное внимание уделяется продукту, а не документам, такая трата времени кажется недостаточной.

Проектирование и исполнение

В качестве отправной точки для выполнения теста нам нужно определить, что подлежит тестированию. Чтобы ответить на этот вопрос, команды контроля качества разрабатывают тестовые случаи. В двух словах, тестовый случай описывает предварительные условия, желаемые результаты и постусловия определенного тестового сценария, направленные на проверку соответствия функции основным требованиям.

Следующим шагом в выполнении теста является настройка среды тестирования. Основные критерии для этой части заключаются в том, чтобы убедиться, что среда тестирования максимально приближена к реальной среде конечного пользователя (аппаратное и программное обеспечение). Например, типичная среда тестирования для веб-приложения должна включать веб-сервер, базу данных, операционную систему и браузер.

В процессе тестирования программного обеспечения выделяются две широкие категории: статическое тестирование и динамическое тестирование.

Статическое тестирование первоначально проверяет исходный код и проектные документы программного обеспечения, чтобы выявить и предотвратить дефекты на ранних этапах жизненного цикла тестирования программного обеспечения. Также называемое методом неисполнения или проверочным тестированием, статическое тестирование может проводиться в виде проверок, неофициальных и технических обзоров или обзоров во время пошаговых совещаний.

Неофициальный обзор — это дешевый вариант тестирования, который аналитик по качеству может провести в любое время во время проекта.

Инспекция, также называемая официальным обзором, планируется и контролируется модератором. Во время совещания по обзору ошибки, обнаруженные аналитиками контроля качества, обсуждаются и документируются в отчете по обзору.

Как только основные подготовительные работы завершены, команда приступает к динамическому тестированию, при котором программное обеспечение тестируется во время выполнения. В этом техническом документе основное внимание уделяется процессу динамического тестирования как практическому и наиболее часто используемому способу проверки поведения кода. Динамическое тестирование может быть описано методами, уровнями и типами основных мероприятий по обеспечению качества. Давайте подробнее рассмотрим этот сегмент процесса динамического тестирования.

Методы тестирования программного обеспечения-это способы проведения тестов. Они включают тестирование черного ящика, тестирование белого ящика, тестирование серого ящика и специальное тестирование.

Уровни тестирования программного обеспечения описывают этапы разработки программного обеспечения при проведении тестирования. Тем не менее, существует четыре уровня прогрессивного тестирования в зависимости от области, в которой они сосредоточены на процессе разработки программного обеспечения: модульное тестирование, интеграционное тестирование, системное тестирование и тестирование на приемлемость для пользователей (UAT).

Типы тестирования программного обеспечения — это подходы и методы, которые применяются на данном уровне с использованием соответствующего метода для наиболее эффективного удовлетворения требований к тестированию. Их огромное количество служит различным целям.

Подводя итог, можно выполнить тестирование на случай использования во время системного или приемочного тестирования с использованием тестирования черного ящика.

Поскольку идеального программного обеспечения не существует, тестирование никогда не будет завершено на 100 процентов. Это непрерывный процесс. Однако существуют так называемые «критерии выхода», которые определяют, было ли проведено «достаточное тестирование», основанное на оценке рисков проекта.

Есть общие моменты, которые присутствуют в основном в критериях выхода:

- Выполнение тестового задания завершено на 100 процентов.
- Система не имеет дефектов с высоким приоритетом.
- Производительность системы остается стабильной независимо от внедрения новых функций.
- Программное обеспечение поддерживает все необходимые платформы и/или браузеры
- Завершено приемочное тестирование пользователей.

- Как только все эти критерии (или любые пользовательские критерии, установленные вами в вашем проекте) будут выполнены, тестирование завершится.

Журналы тестирования и отчеты о состоянии документируются на протяжении всего процесса выполнения теста. О каждой проблеме, обнаруженной в продукте, следует сообщать и обрабатывать соответствующим образом. Резюме тестирования и отчеты о завершении тестирования подготавливаются и предоставляются заинтересованным сторонам. Команда проводит ретроспективное совещание, чтобы определить и задокументировать проблемы, возникшие в ходе разработки, и улучшить процесс.

Уровни тестирования программного обеспечения

Компонентное/Модульное тестирование

Самую маленькую тестируемую часть программной системы часто называют единицей. Поэтому этот уровень тестирования направлен на изучение каждого отдельного блока программной системы, чтобы убедиться, что она соответствует первоначальным требованиям и функционирует должным образом. Модульное тестирование обычно выполняется на ранних стадиях процесса разработки самими инженерами, а не командой тестирования.

Интеграционное тестирование

Цель следующего уровня тестирования-проверить, хорошо ли объединенные подразделения работают вместе как группа. Интеграционное тестирование направлено на выявление недостатков во взаимодействии между блоками внутри модуля. Существует два основных подхода к этому тестированию: методы «снизу вверх» и «сверху вниз». Интеграционное тестирование «снизу вверх» начинается с модульных тестов, последовательно повышающих сложность тестируемых программных модулей. Метод «сверху вниз» использует противоположный подход, сначала фокусируясь на комбинациях высокого уровня, а затем изучая простые.

Тестирование системы

На этом уровне тестируется полная программная система в целом. Этот этап служит для проверки соответствия продукта функциональным и техническим требованиям и общим стандартам качества. Тестирование системы должно проводиться высокопрофессиональной командой тестирования в среде, максимально приближенной к реальному сценарию использования в бизнесе.

Приемочное Тестирование Пользователей

Это последний этап процесса тестирования, на котором продукт проверяется на соответствие требованиям конечного пользователя и на точность. Этот последний шаг помогает команде решить, готов ли продукт к отправке или нет. Хотя небольшие проблемы должны быть обнаружены и устранены на более ранних этапах процесса, этот уровень тестирования фокусируется на общем

качестве системы, начиная с контента и пользовательского интерфейса и заканчивая проблемами производительности. За этапом принятия может последовать альфа-и бета-тестирование, что позволит небольшому числу реальных пользователей опробовать программное обеспечение до его официального выпуска.

Методы тестирования программного обеспечения

Тестирование Черного Ящика

Этот метод получил свое название, потому что инженер по контролю качества фокусируется на входных данных и ожидаемых результатах, не зная, как приложение работает внутри и как эти входные данные обрабатываются. Цель этого метода-проверить функциональность программного обеспечения, убедившись, что оно работает правильно и соответствует требованиям пользователя. Этот метод может быть применен к любому уровню тестирования, но используется в основном для приемочного тестирования системы и пользователя.

Тестирование Белого Ящика

В отличие от тестирования черного ящика, этот метод требует глубоких знаний кода, поскольку предполагает тестирование некоторой структурной части приложения. Поэтому, как правило, разработчики, непосредственно участвующие в написании кода, несут ответственность за этот тип тестирования. Целью тестирования «белого ящика» является повышение безопасности, потока входов/выходов через приложение, а также улучшение дизайна и удобства использования. Этот метод в основном используется на уровне модульного и интеграционного тестирования.

Тестирование Серой Коробки

Этот метод представляет собой комбинацию двух предыдущих, поскольку он предполагает тестирование как функциональных, так и структурных частей приложения. Используя этот метод, опытный тестировщик частично знаком с внутренней структурой приложения и на основе этих знаний может разрабатывать тестовые примеры, продолжая тестирование с точки зрения черного ящика. Этот метод в основном применим к уровню интеграционного тестирования.

Специальное Тестирование

Это неофициальный метод тестирования, поскольку он выполняется без планирования и документации. Проводя тесты неофициально и случайным образом без каких-либо формальных ожидаемых результатов, тестировщик импровизирует шаги и произвольно выполняет их. Хотя дефекты, обнаруженные с помощью этого метода, сложнее воспроизвести, учитывая отсутствие письменных тестов, этот подход помогает быстро находить важные дефекты, что невозможно сделать с помощью формальных методов.

Функциональное тестирование

Набрав 83 процента голосов респондентов различных опросов, функциональное тестирование является наиболее важным видом тестирования. Этого следовало ожидать, поскольку без функциональности не было бы возможности использовать все другие нефункциональные аспекты системы.

При функциональном тестировании система тестируется на соответствие функциональным требованиям путем подачи на вход и проверки вывода. При этом типе тестирования применяется метод черного ящика. Следовательно, это придает значение не самой обработке, а, скорее, ее результатам. Функциональное тестирование обычно проводится на уровнях системы и приемки.

Как правило, процесс функционального тестирования включает в себя следующий набор действий:

1. Описывает функции, которые должно выполнять программное обеспечение.
2. Составляет входные данные в зависимости от спецификаций функций.
3. Определяет выходные данные в зависимости от спецификаций функций.
4. Выполняет тестовый случай.
5. Сопоставляет полученные и ожидаемые выходные данные.

Тестирование производительности

Тестирование производительности было выбрано 60,7 процента респондентов в качестве наиболее важного нефункционального типа тестирования. Тестирование производительности направлено на изучение быстродействия и стабильности работы системы при определенной нагрузке.

В зависимости от рабочей нагрузки поведение системы оценивается с помощью различных видов тестирования производительности:

- Нагрузочное тестирование — при постоянно увеличивающейся рабочей нагрузке.
- Стресс — тестирование в пределах или за пределами ожидаемой рабочей нагрузки.
- Испытание на выносливость — при постоянной и значительной рабочей нагрузке.
- Спайковое тестирование — при внезапно и существенно возросшей рабочей нагрузке.

Тестирование Вариантов использования

Это наиболее широко используемый метод тестирования, за которым следует исследовательское тестирование. Вариант использования описывает, как система будет реагировать на данный сценарий, созданный пользователем. Он ориентирован на пользователя и фокусируется на действиях и субъекте, не

принимая во внимание ввод и вывод системы. Имея в виду концепции проекта, разработчики пишут сценарии использования, и после их завершения поведение системы соответствующим образом тестируется. Тестировщики, в свою очередь, используют их для создания тестовых случаев.

Тестирование вариантов использования широко применяется при разработке тестов на уровне системы или приемки. Это также помогает выявить дефекты в интеграционном тестировании. Тестирование вариантов использования проверяет, работает ли путь, используемый пользователем, должным образом, и гарантирует, что задачи могут быть успешно выполнены. Применяя тестирование вариантов использования, аналитики могут выявлять недостатки и изменять систему таким образом, чтобы она достигала эффективности и точности.

Исследовательское тестирование

Исследовательская методика тестирования была впервые описано как стиль тестирования программного обеспечения, которое подчеркивает личной свободы и ответственности личности тестер, чтобы постоянно оптимизировать стоимость ее работы путем обработки тестов, связанных с обучением, тест-дизайн, выполнение теста и результат теста интерпретации как взаимодополняющие мероприятия, которые должны выполняться параллельно на протяжении всего проекта.

Используя специальный метод, исследовательское тестирование не полагается на предопределенные и документированные тестовые случаи и этапы тестирования, как это делают большинство типов тестирования. Вместо этого это интерактивный процесс в свободной форме, в котором основное внимание уделяется проверке пользовательского опыта, а не кода. Это имеет много общего со специальным или интуитивным тестированием, но более систематично. Применяя исследовательское тестирование, квалифицированные тестировщики могут предоставить ценные и проверяемые результаты.

Тестирование юзабилити

Выбранное 44,1 процентами респондентов, юзабилити-тестирование проводится с точки зрения конечного пользователя, чтобы убедиться, что система проста в использовании. Этот тип тестирования не следует путать с приемочным тестированием пользователей. Последний проверяет, соответствует ли конечный продукт установленным требованиям; первый гарантирует, что подход к реализации будет работать для пользователя.

ТЕМА 10. ИНСТРУМЕНТАЛЬНЫЕ СРЕДСТВА РАЗРАБОТКИ ПРОГРАММНОГО И АППАРАТНОГО ОБЕСПЕЧЕНИЯ ИС И АС

Инструментальное программное обеспечение (Software tools) — программное обеспечение, используемое в ходе разработки, корректировки или развития других программ:

- Редакторы.
- Компиляторы.
- Отладчики.
- Вспомогательные системные программы.
- Графические пакеты и др.

Сюда также входят языки программирования, интегрированные среды разработки программ, CASE-системы и др.

Все эти инструменты сейчас объединяются понятием Интегрированной средой разработки (IDE), то есть интегрированной средой разработки программного обеспечения называют систему программных средств, используемую программистами для разработки программного обеспечения.

Обычно среда разработки включает в себя текстовый редактор (оснащенный средствами анализа и дополнения кода), компилятор и/или интерпретатор, компоновщик, отладчик и справочную систему. Иногда также содержит систему управления версиями и разнообразные инструменты для упрощения конструирования графического интерфейса пользователя.

Многие современные среды разработки также включают инспектор объектов, браузер классов и диаграмму иерархии классов, которые используются для объектно-ориентированной разработки ПО.

Обычно среда разработки предназначается для одного определенного языка программирования, как, например, Visual Studio или Embarcadero RAD, но существуют среды разработки, предназначенные для нескольких языков, такие как Eclipse или Microsoft Visual Studio Code, IntelliJ Idea и т.д.

В последнее время, с развитием объектно-ориентированного программирования, широкое распространение получили упоминавшиеся ранее среды визуального программирования, в которых наиболее распространенные блоки программного кода представлены в виде графических объектов – это поколение IDE сейчас развивается в направлении low-code или no-code платформ.

Большую популярность в наши дни получила технология .NET Framework, предложенная фирмой Microsoft в качестве платформы для создания как обычных программ, так и веб-приложений. Основным преимуществом .NET является совместимость различных служб, написанных на разных языках.

Например, служба, написанная на C++ для .NET, может обратиться к методу класса из библиотеки, написанной на Delphi; на C# можно написать класс, наследующий от класса, написанного на Visual Basic .NET, а исключение,

выброшенное методом, написанным на C#, может быть перехвачено и обработано в Embarcadero RAD.

Давайте, например рассмотрим IDE Microsoft Visual Studio.

Это, на самом деле, целая линейка продуктов компании Microsoft, включающих интегрированную среду разработки программного обеспечения и ряд других инструментов. Данные продукты позволяют разрабатывать как консольные приложения, так и игры и приложения с графическим интерфейсом, в том числе с поддержкой технологии Windows Forms и WPF, а также веб-сайты, веб-приложения, веб-службы как в родном, так и в управляемом кодах для всех платформ, поддерживаемых Windows, Windows Mobile, Windows CE, .NET Framework, Xbox и т.д.

Visual Studio включает в себя редактор исходного кода с поддержкой технологии IntelliSense и возможностью простейшего рефакторинга кода. Встроенный отладчик может работать как отладчик уровня исходного кода, так и отладчик машинного уровня. Остальные встраиваемые инструменты включают в себя редактор форм для упрощения создания графического интерфейса приложения, веб-редактор, дизайнер классов и дизайнер схемы базы данных. Visual Studio позволяет создавать и подключать сторонние дополнения (плагины) для расширения функциональности практически на каждом уровне, включая добавление поддержки систем контроля версий исходного кода (как, например, Subversion и Visual SourceSafe), добавление новых наборов инструментов (например, для редактирования и визуального проектирования кода на предметно-ориентированных языках программирования) или инструментов для прочих аспектов процесса разработки программного обеспечения (например, клиент Team Explorer для работы с Team Foundation Server).

Visual Studio также позволяет устанавливать дополнения (плагины) из специального магазина. Они могут быть бесплатные, условно-бесплатные или платные.

Также существует целый набор инструментов разработчика, которые могут применяться для разработки широкого спектра проектов на разных языках программирования.

Их сейчас в основном рассматривают через призму веб-разработки, так как они обладают универсальностью и переносимостью на различные платформы и операционные системы.

Давайте дальше будем рассматривать классический вариант full-stack разработки на языке JavaScript.

Среди инструментальных средств разработки решений на языке JavaScript мы можем выделить:

- редакторы кода;
- интегрированные среды разработки;
- средства отладки и диагностики;
- сборщики, компиляторы, транспиллеры.

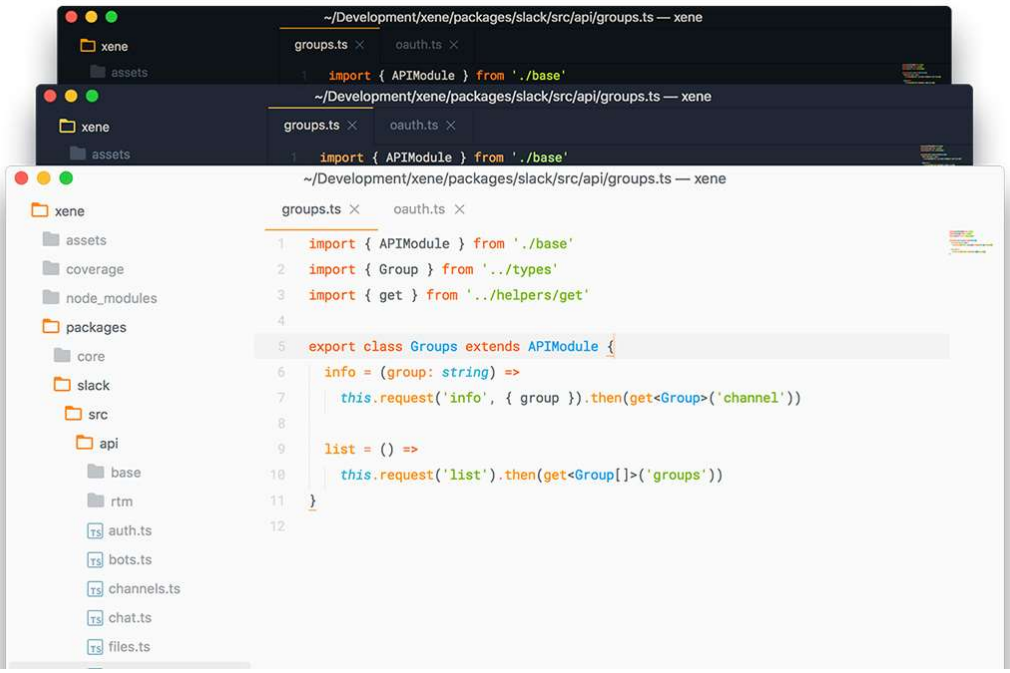
Рассмотрим эти элементы последовательно и начнем с самых популярных редакторов кода.

SublimeText3.

Sublime Text — это, пожалуй, самый простой кроссплатформенный редактор искходного кода. Его интерфейс настраивается, а выполнять некоторые действия можно с помощью горячих клавиш (https://www.sublimetext.com/).

Он бесплатен, хотя и постоянно «просит» купить платную версию для поддержки проекта.

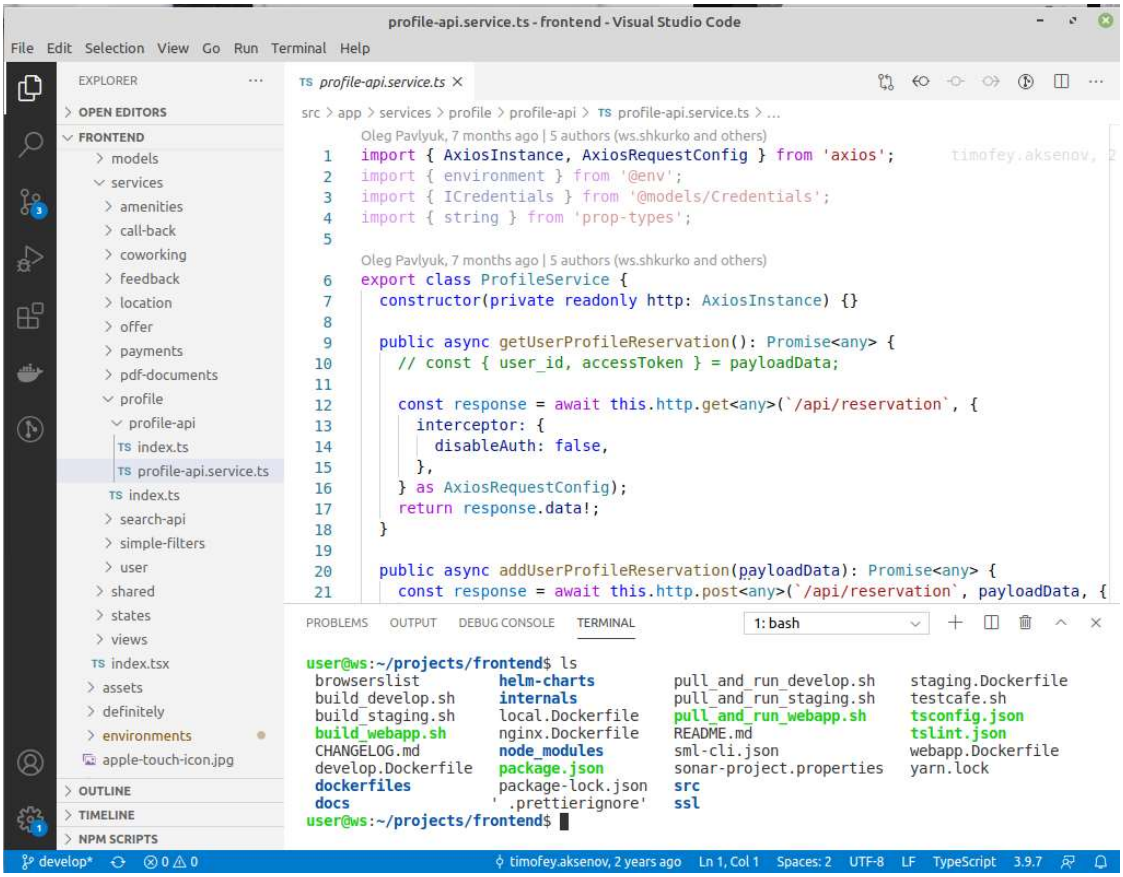
Легковесный, имеет систему пакетов и расширений package control. Имеет темы оформления, поддерживает «нечеткий» поиск по файлам, автодополнение кода, работает практически всеми современными языками программирования и даже с системами сборки.



Следующим лидером опросов является Visual Studio Code от компании Microsoft.

Visual Studio Code.

Также кроссплатформенный, поддерживает огромное количество языков программирования, имеет систему плагинов, современный отладчик, встроенную консоль и систему контроля версий. По своей сути, это облегченная версия Visual Studio, разработанная, кстати говоря, на основе JavaScript + Electron.



Этот редактор используется даже как онлайн редактор, даже без установки на компьютер. Многие современные разработчики кроссплатформенных приложений выбирают его как основной инструмент.

Другим конкурентом Visual Studio Code (VSCode) является Atom от GitHub.

Atom.

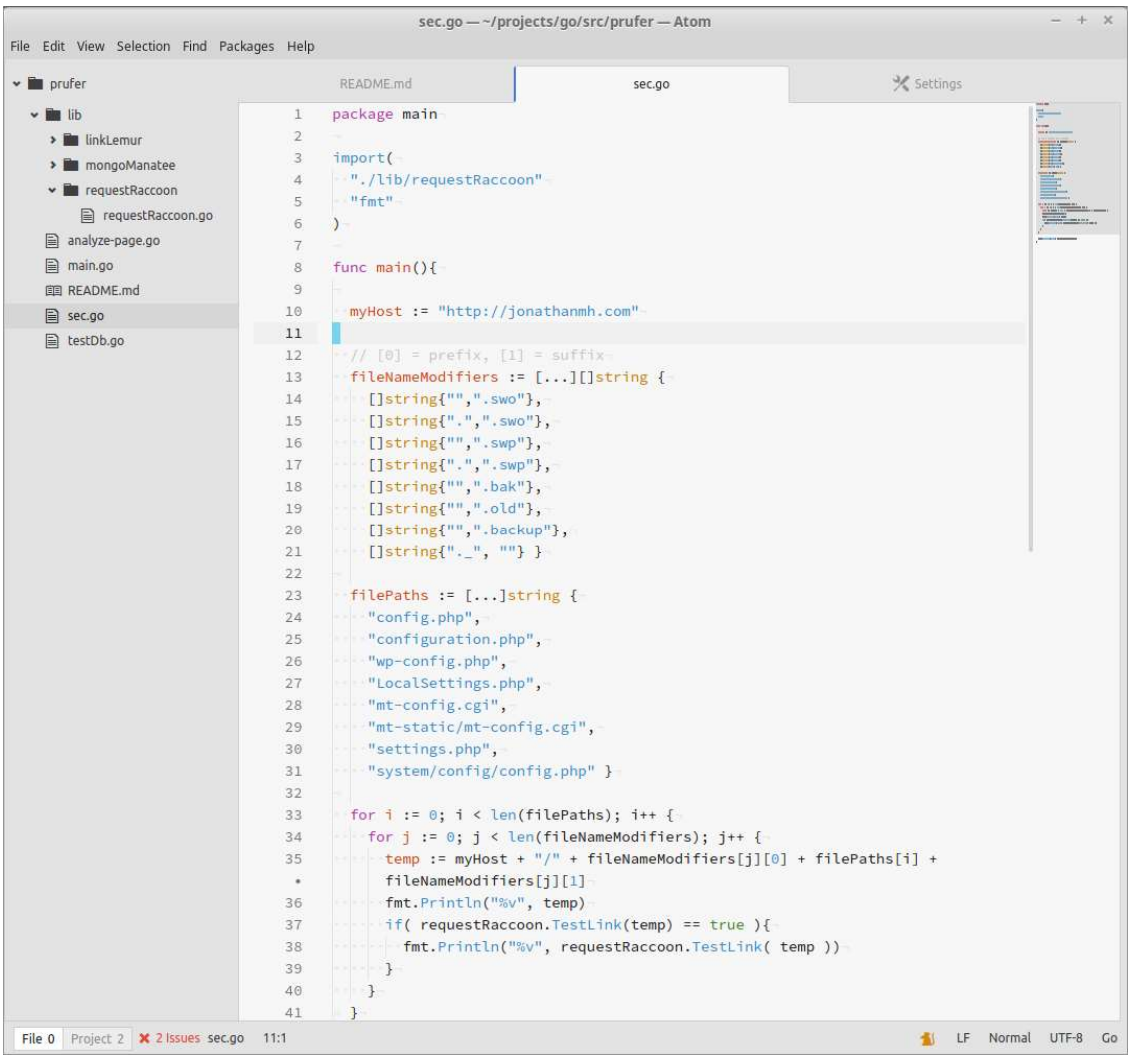
Также разработан на базе браузера Chrome, кроссплатформенный. Имеет возможность расширения при помощи модулей.

Atom, как и VSCode, имеет доступ более чем к трем тысячам официальных расширений. Но в настоящий момент многие модули расширения уже настолько мощные, что позволяют превратить Atom из редактора кода, практически в полноценную среду разработки.

Имеет возможность гибкой настройки, поддерживает навигацию и управление при помощи горячих клавиш.

Так же существует множество других редакторов кода, которые также можно попробовать и найти свой идеальный вариант.

Это Brackets, Eclipse, Notepad++, emacs и даже vim. Все зависит от предпочтений и от мощности рабочей станции. Если рабочая станция совсем слабая, то подойдет Sublime и vim, если есть лишний гигабайт ОЗУ чисто для редактора - можно попробовать VSCode или Atom.



Далее рассмотрим интегрированные среды разработки.

Чем IDE отличается от текстового редактора кода?

IDE представляет собой более сложный инструмент, чем обычный текстовый редактор. Несмотря на то, что в текстовых редакторах есть масса полезных функций вроде подсветки синтаксиса, единственная их задача – обеспечивать работу с кодом. То есть для полноценной разработки вам понадобится еще хотя бы компилятор и отладчик. По сути, термин IDE обозначает то, что у вас под рукой будет все, что необходимо для разработки приложений и программ.

WebStorm от компании JetBrains.

Это интегрированная среда разработки на JavaScript, CSS & HTML от компании JetBrains, разработанная на основе платформы IntelliJ IDEA.

WebStorm обеспечивает автодополнение, анализ кода на лету, навигацию по коду, рефакторинг, отладку, и интеграцию с системами управления версиями. Важным преимуществом интегрированной среды разработки WebStorm является работа с проектами.

Так же IDE работает с множеством расширений, но весь необходимый функционал работает прямо “из коробки”. WebStorm имеет несколько удивительных функций.

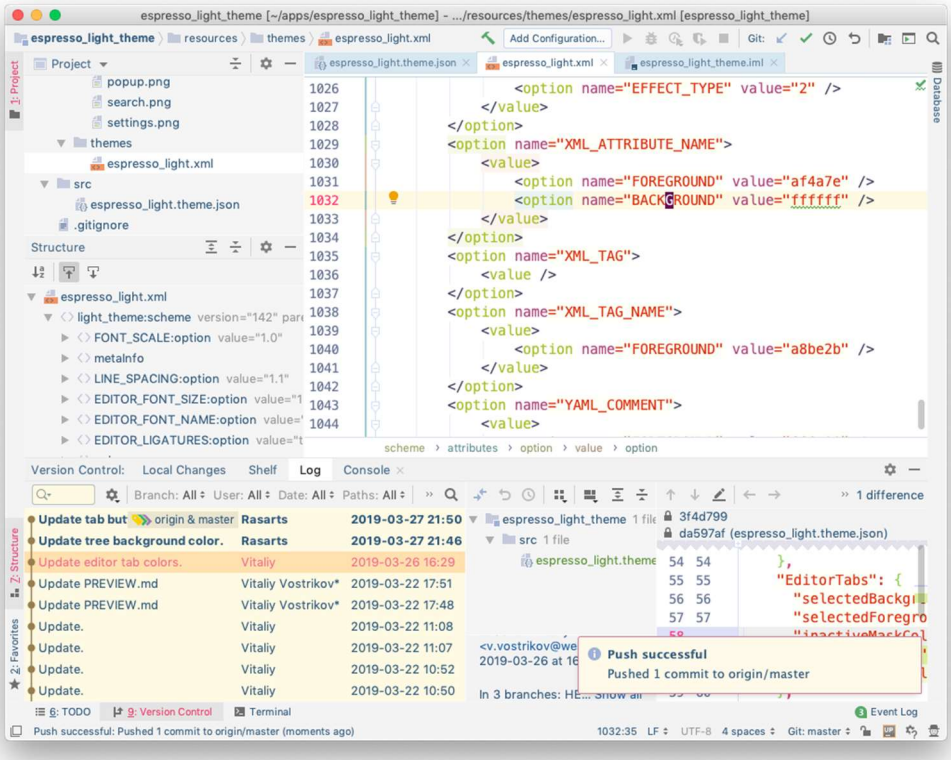
LiveEdit — новая возможность WebStorm, появившаяся в версии 5 и позволяющая одновременно редактировать код html, css или javascript и видеть, как результат отображается в браузере. Для этого требуется поддержка такой возможности со стороны браузера, поэтому WebStorm при установке ставит плагин для Google Chrome.

WebStorm поддерживает отладку приложений в node.js. Также поддерживается полный набор функций редактирования приложений на javascript — как для исполнения на сервере, так и в браузере: автодополнение, навигация по коду, рефакторинг и проверка на ошибки.

Языки стилей LESS, Sass, SCSS и Stylus которые расширяют возможности описаний стилей в CSS, полностью поддерживаются в WebStorm, в частности, поддерживается рефакторинг кода для них.

В WebStorm для CoffeeScript, Dart и TypeScript есть навигация по коду, автодополнение, рефакторинг, подсветка синтаксиса и проверка на ошибки.

Более того, редактор кода в IDE WebStorm настолько совершенный, что содержит средства, подсказывающие и применяющие улучшения программного кода.



Кроме всех перечисленных преимуществ, продукты компании JetBrains поддерживают разные конфигурации запуска, автотесты и возможность использовать виртуализацию Docker прямо из редактора.

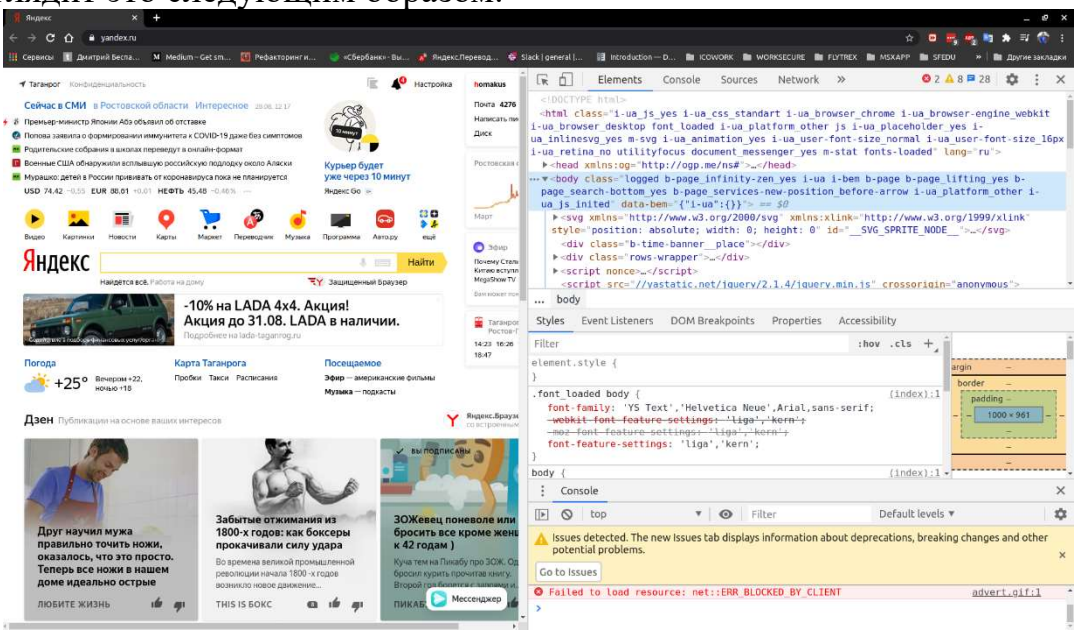
Конечно, редактор является платным и его стоимость сегодня составляет (при оплате сразу за год) 129 $ за первый год использования, 103 $ за второй и 77 $, начиная с третьего года. Ежемесячная оплата составляет 12.90 $ на август 2020.

Еще одним необходимым инструментом разработчика кроссплатформенных приложений на языке JavaScript является сам по себе браузер и его встроенные инструменты.

Мы используем браузер Chrome, однако подобные инструменты доступны и на других платформах: Firefox, Safari, Microsoft Edge, Opera.

Инструменты разработчика открываются по нажатию клавиши F12 или после клика правой кнопки мышки на страницу и выбора пункта меню Просмотреть код.

Выглядит это следующим образом:



Справа вы можете видеть несколько вкладок, разберемся, какие из них за какую функцию отвечают.

Вкладка **Elements** (Элементы). На ней представлен конечный код HTML страницы, который отличается от исходного. Любые элементы HTML, созданные или измененные с помощью JavaScript при загрузке страницы, будут отражены в конечном коде HTML. Исходный код при этом останется без изменений. Эта вкладка является основной для верстальщика, так как позволяет разобраться с тем, какие конкретно элементы разметки прорисовались и где конкретно. Кроме того, здесь можно посмотреть конечное представление стилей элементов страницы и, изменив эти стили, увидеть, как поменялась страница.

Вкладка **Resources** (Ресурсы) позволяет проверить таблицу различных ресурсов, которые были загружены вместе с рассматриваемой страницей. К их числу относятся изображения, документы HTML, файлы JavaScript и многое другое. Эта вкладка удобна для устранения неполадок, связанных с Менеджером кампаний, потому в ней можно выполнять поиск по всем ресурсам на странице, а не только в ней самой.

Вкладка **Network** (Сеть) – это встроенный прокси-анализатор, который позволяет отслеживать HTTP-трафик страницы во время и после ее загрузки. Эта вкладка является основной для фронт-энд разработчиков, которым нужно отслеживать все пакеты, отправляемые на серверный API. Для каждого пакета

доступен просмотр параметров, его структуры, форматированного и неформатированного ответа, а также шкалу времени (сколько выполнялся каждый запрос и какие были задержки).

Вкладка **Scripts** (Скрипты) предназначена для отладки кода JavaScript. Она является незаменимым инструментом для веб-разработчиков, но не используется для проверки или устранения неполадок в Менеджере кампаний.

Вкладка Timeline (Хронология) показывает хронологию трафика HTTP и загрузки памяти. Как и вкладка Network, она помогает выявить причины задержки. Это единственное ее применение, связанное с Менеджером кампаний.

Вкладка **Profiles** (Профили) – это инструмент, который веб-разработчики могут использовать для оптимизации загрузки процессора в веб-приложениях. Эта вкладка не используется в связи с Менеджером кампаний.

Вкладка **Audits** (Проверка) позволяет анализировать страницы в процессе загрузки и предлагать варианты их оптимизации для уменьшения времени загрузки и улучшения кажущейся (и действительной) реакции. Эта вкладка не используется в связи с Менеджером кампаний.

Вкладка **Console** (Консоль) автоматически обнаруживает ошибки в коде страницы. После того как с помощью вкладки Network вы определили, что тег не срабатывает, вкладка Console поможет вам понять, почему это происходит. Кроме того, прямо в этой консоли можно написать функции JavaScript и посмотреть их выполнение именно с элементами загруженной стреницы. Это вы можете видеть на примерах из данного методического пособия.

Вкладка **Производительность** (Perfomance) позволяет записать действия на странице и измерить временные задержки выполнения.

Вкладка **Память** (Memory) позволяет оценить затраты памяти на веб-приложение.

Вкладка **Приложение** (Application) позволяет провести анализ следующих паарметров веб-приложения: Манифест (Manifest), Сервисные потоки (Service Workers), Хранилища (Storage): Локальное (Local Storage), Сессионное (Session Storage), IndexedDB, WebSQL, Cookies, Кэш (Cache): Локальный кэш (Cache Storage) и кэш приложения (Application Cache), Фоновые задачи (Background Services), а также запросы, уведомления, обработчики платежных систем и так далее.

Вкладка **Безопасность** (Security) позволяет получить информацию о сертификатах веб-сервера приложения.

Дополнительная вкладка **Lighthouse** позволяет оценить соответствие веб-приложения современным стандартам и сгенерировать отчет (report) по следующим критериям: Производительность, Прогрессивное веб-приложение, Лучшие практики Доступность, SEO.

Кроме этого, для разработчиков доступны дополнительные вкладки, поставляемые с расширениями. Например, React Developer Tools - Google Chrome или Vue.js devtools - Google Chrome.

В этом плане Chrome с такими расширениями является универсальным средством отладки веб-приложений, так как содержит даже отладчик и полнофункциональный профилировщик программы.

Следует помнить, что IDE – это настоящая фабрика для производства программного кода и каждый разработчик обязан грамотно подбирать среду разработки «под себя» и под те условия, в которых приходится работать, включая операционную систему и язык программирования.

ТЕМА 11. ИНТЕЛЛЕКТУАЛЬНЫЕ ТЕХНОЛОГИИ ОБЕСПЕЧЕНИЯ ИС И АС

Интеллектуальные технологии сейчас используются везде: для анализа данных, принятия решений, интерпретации результатов исследований, управления объектами и т.п.

Их также научились использовать и при разработке программного обеспечения. Это направление называется «использование искусственного интеллекта в разработке программного обеспечения».

Искусственный интеллект (ИИ) сейчас помогает оптимизировать и ускорить процесс проектирования, разработки и внедрения программного обеспечения (ПО).

Суть заключается не в том, чтобы инженеров-разработчиков заменили роботы.

Скорее, инструменты на базе ИИ работают в роли ассистентов руководителей проектов, бизнес-аналитиков, программистов и инженеров по тестированию. За счет этого специалисты по разработке могут создавать и тестировать части кода быстрее и с меньшими затратами. Таким образом, ИИ может стать важным фактором, который приведет к увеличению производительности программистов и повышению качества продуктов.

Сейчас AI-augmented Software Engineering - в списке самых перспективных технологий будущего.

Разработка программного обеспечения с помощью ИИ, AI-augmented Software Engineering – термин, который ввели в Gartner в 2020 году для описания процесса использования технологий искусственного интеллекта (например, машинного обучения, обработки естественного языка и др.) для ускорения циклов разработки приложений и DevOps. Ряд вендоров уже выпустил продукты для такого типа работ. Среди них Codota, Deep Code, Google, Kite, Mendix, Microsoft, OutSystems, Parasoft.

Прогнозируют, что к 2022 – 2023 г. минимум 40% новых проектов по разработке программного обеспечения будут использовать ресурс виртуальных ИИ-разработчиков.

Однако технологии искусственного интеллекта и машинного обучения уже около пяти лет применяются компаниями и отделами по разработке ПО по всему миру. В 2020 году Gartner впервые включил это понятие в список самых перспективных технологий будущего – Hype Cycle for Emerging Technologies (цикл зрелости технологий).

Давайте рассмотрим области применения искусственного интеллекта при разработке ПО.

ИИ используется в разработке программных решений на следующих этапах:

- Сбор технических требований. Цифровые ассистенты анализируют документы с собранными требованиями, указывают на разногласия в тексте, нестыковки в цифрах, единицах измерений, суммах и предлагают возможные решения.

- Быстрое прототипирование. Преобразование бизнес-требований в программный код обычно занимает месяцы или даже годы. Однако машинное обучение значительно сокращает этот процесс, позволяя специалистам с меньшим опытом использовать методы разработки естественного языка или визуального интерфейса для создания прототипа.
- Кодирование. В процессе написания кода, работающая на базе ИИ система автозаполнения предлагает рекомендации для завершения строчек кода. Интеллектуальные помощники сокращают время на создание кода на 50%. Дополнительно они могут рекомендовать обратиться к связанным документам, лучшим практикам и дать примеры кода.
- Анализ и обработка ошибок. Виртуальный ассистент может извлекать уроки из прошлого опыта, чтобы выявлять типичные ошибки и автоматически помечать их на этапе разработки. Машинное обучение можно использовать для анализа системных журналов для быстрого и даже упреждающего выявления ошибок.
- Автоматический рефакторинг кода. Чистый код необходим для совместной работы и долгосрочного обслуживания. По мере развития компании, программные решения могут изменяться, и остро встает вопрос о том, как модифицировать код для лучшей работы приложений. Машинное обучение используется в этом случае с целью анализа кода и автоматической оптимизации кода для легкой интерпретируемости и повышения производительности.
- Тестирование. Автоматизированные системы тестирования используют ИИ не только для того, чтобы запускать процесс тестирования, но и для создания test кейсов.
- Ввод в эксплуатацию. Иногда ошибки в программном коде становятся явными только после того, как программное обеспечение введено в эксплуатацию. Но AI-инструменты предотвращают подобные ситуации, проверяя статистику предыдущих релизов и логи приложений.
- Управление проектами. Разработка программного обеспечения иногда выходит за рамки бюджета и графика. Системы продвинутой аналитики позволяют использовать данные большого количества проектов по разработке ПО для прогнозирования технических задач, необходимых ресурсов и времени на выполнение проекта. Машинное обучение может извлекать данные из прошлых проектов, такие как истории пользователей, определения функций, оценки и фактические условия, для более точного прогнозирования рабочей нагрузки и бюджета.

Как же происходит развитие инструментов на базе ИИ для разработки программного обеспечения.

Согласно исследованию Deloitte 2020 года, на рынке отмечается большой интерес к решениям, которые используют инструменты искусственного интеллекта в области разработки ПО:

- За последние годы на рынке появились десятки продуктов, которые используют ИИ для повышения эффективности разработки программных решений.
- Объем годовых инвестиций, привлеченных стартапами, которые предлагают ИИ-продукты для разработки ПО, составил 704 млн долларов США к сентябрю 2019 года.
- Прогнозируется, что объем мирового рынка заказной разработки ПО вырастет до 61 млрд долларов США к 2023 году.
- Благодаря растущему спросу на программное обеспечение, количество инженеров-разработчиков вырастет на 21% в 2028 году.

Что же говорят эксперты рынка программного обеспечения.

По их мнению, каждые пару лет сложность программного обеспечения возрастает. И во время такого «роста», искусственный интеллект «учится» - модели машинного обучения находят важные функции и закономерности в данных, а области, которые больше всего выигрывают от программного обеспечения, включают компьютерное зрение, распознавание речи, машинный перевод, игры, робототехнику и базы данных.

Кэмбриджский университет заявляет в своих исследованиях, что 50% рабочего времени тратится на компиляцию кода и исправление ошибок.

С помощью искусственного интеллекта можно значительно сократить эту часть работ.

Давайте подумаем, как искусственный интеллект может помогать разработчикам в создании программного обеспечения.

В области мобильной разработки ИИ предоставляет разработчикам новые возможности. Прежде всего, это связано с тем, что использование ИИ помогает привлечь к использованию приложения большее количество пользователей.

Инструменты ИИ автоматически выполняют определенные алгоритмы, гарантирующие, что все больше пользователей начнут использовать это приложение. Например, ИИ может отслеживать закономерности и предпочтения пользователей еще на этапе разработки приложения, и прогнозировать будущие решения и работать соответствующим образом, давая «советы» разработчику. Это дает разработчикам возможность оперативно вносить изменения в новые версии приложений, приближая его к «идеальному» и «желаемому» образцу.

Растущая популярность интеллектуальных устройств подстегнула и рост использования ИИ в разработке операционных систем и пользовательских интерфейсов. А потребность в доступности мобильных приложений, которые создают персонализированный опыт пользователей, растет день ото дня. Искусственный интеллект выступает как персональный виртуальный помощник, приложение может фиксировать большое количество пользовательских действий.

Поэтому ИИ-помощники могут оценить UI/UX с точки зрения корректности и востребованности пользователями и дать «совет».

ИИ может оценить качество кода или предсказать места возможного появления ошибок.

Так что растущее количество инструментов на базе ИИ поддерживает процессы разработки программного обеспечения. Часть из этих решений доступна бесплатно. Ведущие технологические вендоры используют подобные инструменты и предлагают их своим клиентам в виде дополнительных продуктов (plug-in).

Facebook использует рекомендательный сервис для исправления ошибок и улучшения кода. Последние проекты IBM Mono2Micro и Application Modernization Accelerator (AMA) предоставляют архитекторам приложений инструменты для обновления устаревших приложений и повторного их применения. А Microsoft в 2021 году объявила, что интегрирует технологии искусственного интеллекта со своим языком программирования Power Fx, который применяется в разработке приложений на платформе Power Platform. Это позволит клиентам компании создавать программы практически без необходимости написания кода.

В России активно использует ИИ для создания программных продуктов Сбер. В июле 2021 года Sber AI зарегистрировала в Роспатенте программу, позволяющую искусственному интеллекту распознавать и анализировать объекты в виртуальной реальности, следует из материалов ведомства.

Согласно опросу Forrester, 37% респондентов признали, что они используют ИИ для более эффективного процесса тестирования и разработки.

Мы можем найти примеры интеллектуальных помощников и оценить, что они предлагают разработчикам программного обеспечения:

Bayou	Генерация идиом API
Clever-Commit	Интеллектуальный поиск ошибок (багов)
Deep Code	Улучшение кода, автоматическая генерация юнит-тестов
Kite, PyCharm	«Умное» автодополнение кода на Python
Sketch2Code	Преобразование скетчей, нарисованных от реки в HTML страницы
Mabl	Помощь в автоматизации тестирования приложений
GPT 3	Генерация веб-страниц и мобильных приложений
Applitools	Визуальное тестирование веб-приложений и мобильных приложений
Embold	Проверка исходного кода и его улучшение

Однако есть и другая сторона у ИИ-продуктов для разработки ПО. Команды, которые используют инструменты для улучшения кода, могут сначала испытывать падение продуктивности, так как ИИ-продукты требуют навыков и умения с ними обращаться. Лишь после глубокого погружения, они способны выдавать точные рекомендации для оптимизации разработки ПО.

Давайте рассмотрим пример последнего разработанного интеллектуального помощника.

Не так давно Microsoft запустила сервис Copilot, представляющий собой виртуального помощника программиста на базе искусственного интеллекта. Он

изучает код и комментарии к нему и предлагает разработчику функции и целые строки для добавления в этот код. Такой подход ускоряет процесс написания программ и отказаться от поиска решений в интернете. К тому же в процессе работы Copilot обучается и с каждым разом становится все умнее. Инструмент полностью бесплатный.

Copilot развернут на базе GitHub, одного из крупнейших в мире Git-репозиториев. С 2018 г. он принадлежит Microsoft. Open AI тоже имеет определенные связи с корпорацией. Как сообщал CNews, в 2019 г. она вложила в этот стартап $1 млрд.

Виртуальный помощник программиста обучен взаимодействовать с различными фреймворками (средами разработки). Он понимает несколько языков программирования, но на момент публикации материала Copilot был заточен в первую очередь под JavaScript, Go, Python, Ruby и TypeScript.

Со слов главы GitHub Нэта Фридмана (Nat Friedman), Copilot – это именно помощник программиста, а не его заменитель. Это находит свое отражение и в названии проекта.

Copilot, утверждает Фридман, даст разработчикам возможность писать более качественный код. Он даже может позволить ускорить этот процесс за счет быстрого поиска решения возникших в ходе написания кода проблем. Другими словами, Copilot может заменить программистам поиск решений в интернете.

Как работает Copilot.

В основе системы Copilot лежит система искусственного интеллекта Codex за авторством специалистов OpenAI. Для ее обучения были задействованы ресурсы самого GitHub, то есть миллионы строк кода в файлах, хранящихся в открытых репозиториях.

В теории Copilot можно сравнить с коллегой-программистом, который постоянно смотрит в чужой монитор и дает советы. Система изучает код, над которым в данный момент идет работа, и заодно комментарии в нем. Также она следит за положением курсора и знает, над какой строчкой трудится программист.

На основе этих данных Copilot начинает предлагать добавить в код одну или несколько строк или функций в зависимости от выполняемой работы. Разработчик сам решает, последовать совету искусственного интеллекта или отклонить его.

Решения программиста тоже влияют на обучение Copilot. Чем больше он взаимодействует с системой, тем быстрее она обучается программированию на различных языках, и тем более полезными становятся ее советы. Как следствие, процесс программирования при наличии постоянно набирающегося знаний помощника, пусть и виртуального, постепенно ускоряется.

На момент публикации материала Copilot был доступен в виде дополнения (плагина) к бесплатному редактору Microsoft Visual Studio Code. Также он работает и через любой современный в браузер в GitHub Codespaces. Оценить возможности Copilot сможет любой разработчик, без каких-либо ограничений.

Крупные компании делают все возможное, чтобы процесс программирования стал проще, быстрее и понятнее. Их действия также

направлены на снижение порога входа в эту сферу, чтобы начать писать код мог почти каждый.

Немалую роль в этом играет и сама компания Microsoft. За месяц до запуска Copilot она создала инструмент для написания ПО без развитых навыков программирования. Проект получил название Microsoft Power Apps, и он пригодится тем, кто пишет на языке Power Fx.

С его помощью пользователи смогут разрабатывать программы в формате диалога с компьютером. Например, при разработке приложения в сфере электронной коммерции можно будет описать в диалоге желаемую цель на естественном английском языке: «find products where the name starts with 'kids» («найти продукты, название которых начинается со слова "детский"» - англ).

После этого Power Apps задействует алгоритмы искусственного интеллекта и предложит варианты преобразования этого запроса в формулу Microsoft Power Fx. Пользователю же останется только выбрать наиболее подходящий вариант, например «Filter('BC Orders' Left('Product Name', 4)="Kids")».

В октябре 2020 г. Microsoft выложила в открытый доступ приложение Lobe для создания готовых моделей машинного обучения. Lobe дает возможность создавать такие модели даже тем, кто не умеет программировать - от пользователей нужно лишь загрузить в нее данные, а всю работу программа выполнит сама.

Lobe полностью бесплатна. Результат ее работы затем можно использовать в сторонних ПО и устройствах.

Но дальше всех пошла компания Amazon. Пока Microsoft развивает помощников программиста, Amazon продвигает сервис, полностью заменяющий разработчика.

В июне 2020 г. CNews писал, что Amazon запустила сервис Honeycode для создания приложений без необходимости написания программного кода. Проект полностью бесплатный, и использовать его могут как обычные потребители, так и крупные компании. К примеру, Slack, разработчик одноименного мессенджера, уже заявила о готовности к использованию Honeycode в своей работе.

Такие проекты, как Microsoft Power Apps, настроены на написание ПО вообще без навыков программирования или с их минимальным уровнем.

Этот подход сейчас называется «Программирование на естественном языке».

Модель GPT-3 предложит варианты преобразования запроса в формулу Microsoft Power Fx, языка программирования Power Platform. Пользователю же останется только выбрать наиболее подходящий вариант.

GPT-3 (Generative Pre-trained Transformer) – крупнейшая языковая модель в мире, разработанная OpenAI для решения любых задач на английском языке. OpenAI является некоммерческой исследовательской организацией, основателями которой выступают главный исполнительный директор Tesla Илон Маск (Elon Musk) и Сэм Альтман (Sam Altman).

GPT-3 работает в фирменном облаке Microsoft Azure, а для ее дообучения под задачу был использован сервис Azure Machine Learning.

Осенью 2020 г. Microsoft получила эксклюзивную лицензию на использование модели GPT-3. Интеграция с Power Apps является первым применением модели GPT-3 в продукте, доступном широкому кругу пользователей, утверждают в Microsoft.

Несмотря на простоту языка Power Fx, формирование, к примеру, сложных запросов к данным все еще может требовать достаточно глубоких технических знаний – по крайней мере, понимания логики написания формул. Использование естественного языка в процессе создания приложений, по мнению специалистов Microsoft, позволит еще больше снизить порог вхождения в разработку приложений.

Помимо интеграции с GPT-3, Microsoft планирует дать возможность разработчикам бизнес-приложений Power Apps писать код в рамках концепции «программирование на основе примера» (Programming by Example, PBE). В ней искусственный интеллект генерирует программный код для преобразования данных на базе шаблона, который строит, предварительно проанализировав пользовательский пример: исходную информацию и конечный результат.

В качестве иллюстрации принципа работы техники PBE Microsoft вновь приводит ситуацию, которая могла бы возникнуть в процессе создания приложения для электронной коммерции. Допустим, разработчику необходимо поменять формат отображения имен клиентов в некоторой таблице данных – вместо имени и фамилии теперь должны отображаться имя и инициал, заканчивающийся точкой.

Чтобы это реализовать на практике, разработчику достаточно «скормить» системе исходное и желаемое значения, например, “John Snow” и “John S.”, после чего она сгенерирует формулу на языке Power Fx (весьма громоздкую в данном случае) на основе выявленных искусственным интеллектом закономерностей. Эта формула позволит преобразовать все оставшиеся данные по заданному шаблону.

За реализацию принципа PBE отвечает технология PROSE (Program Synthesis Using Examples), разработанная командой исследовательского подразделения Microsoft Research. PROSE уже используется в таких инструментах как Power BI, Excel и Visual Studio.

Power Fx – это язык программирования, предназначенный для настройки процессов в Power Platform. Язык основан на синтаксисе функций табличного редактора Microsoft Excel и относится к категории так называемых low-code-инструментов, то есть не требующих от пользователя серьезных навыков программирования для успешного применения.

Код интерпретатора Power Fx открыт и опубликован на хостинге ИТ-проектов Github.

Microsoft впервые объявила о запуске Power Fx в марте 2021 г. Ожидается, что Power Fx поможет снизить порог вхождения в разработку и позволит бизнес-пользователям создавать приложения самостоятельно. Профессиональные же разработчики смогут с его помощью ускорить процесс разработки.

Одновременно с анонсом Power Fx Microsoft также открыла всем пользователям операционной системы Windows 10 бесплатный доступ к

инструменту Power Automate Desktop, который позволяет автоматизировать рутинные задачи. Причем для этого не нужно уметь программировать – разработка и отладка осуществляются в интуитивно понятной визуальной среде.

Таким образом, искусственный интеллект становится ценным источником обратной связи для разработчиков, которые могут в сжатые сроки реализовывать выявленные потребности пользователей.

ТЕМА 12. CASE-СРЕДСТВА ИС И АС

По своему определению, CASE-средства — это инструменты, которые способны автоматизировать процесс разработки информационных систем различного масштаба. Также CASE-средства применяются при проектировании и для создания программных продуктов.

То есть основная цель – это сокращение времени, затрачиваемого на разработку информационной системы, уменьшение числа ошибок и повышение качества конечного продукта.

Многие современные CASE средства предоставляют возможности для моделирования практически всех предметных областей деятельности организаций. В составе этих средств существуют инструменты для описания моделей бизнес-процессов за счет различных диаграмм, схем, графов и таблиц.

Также CASE-средства применяются в процессах управления, для совершенствования продуктов и улучшения самого рабочего процесса, так как разработка и создание информационных систем управления предприятием связаны с выделением бизнес-процессов, их анализом, определением взаимосвязи элементов процессов, оптимизации их инфраструктуры и т.д.

На сегодня известно достаточно много инструментов, которые мы можем отнести к категории CASE-средств, поэтому целесообразно провести их классификацию.

Выделяют следующие основные группы CASE средств:

- CASE средства верхнего уровня. Эти CASE средства ориентированы на начальные этапы построения информационной системы. Они связаны с анализом и планированием. CASE средства верхнего уровня обеспечивают стратегическое планирование, расстановку целей, задач и приоритетов, а также графическое представление необходимой информации. Все CASE средства верхнего уровня содержат графические инструменты построения диаграмм, таких как диаграммы сущность-связь (ER диаграммы), диаграммы потока данных (DFD), структурные схемы, деревья решений и пр.
- CASE средства нижнего уровня. Эти CASE средства больше сфокусированы на последних этапах разработки информационной системы – проектирование, разработка программного кода, тестирование и внедрение. CASE средства нижнего уровня зависят от данных, которые предоставляют средства верхнего уровня. Они используются разработчиками приложений и помогают создать информационную систему, однако не являются полноценными инструментами разработки программного обеспечения.
- Интегрированные CASE средства (I – CASE). Эти CASE средства охватывают полный жизненный цикл разработки информационной системы. Они позволяют обмениваться данными между инструментами верхнего и нижнего уровня и являются своего рода «мостом» между CASE средствами верхнего и нижнего уровней.

Такое разделение на уровни логично, так как для моделирования и оптимизации бизнес-процессов применяются CASE средства верхнего уровня и интегрированные CASE средства, а в процессе создания программного обеспечения чаще всего непрерывно применяются CASE-средства третьего уровня.

Давайте рассмотрим дальше основные характеристики CASE-средств.

Основными характеристиками CASE средств, важными с точки зрения моделирования и оптимизации бизнес-процессов, являются следующие:

- Наличие графического интерфейса. Для представления моделей процессов CASE средства должны обладать возможностью отображать процессы в виде схем. Схемы много проще в использовании, чем различные текстовые и числовые описания. Это позволяет получать легко управляемые компоненты модели, обладающие простой и ясной структурой.
- Наличие репозитория. Репозиторий это общая база данных, которая содержит описание элементов процессов и отношений между ними. Каждый объект репозитария должен обладать перечнем свойств, характерных только для этого объекта.
- Гибкость применения. Эта характеристика дает возможность представлять бизнес-процессы в различных вариантах, важных с точки зрения анализа. CASE средства должны позволять проводить анализ процессов и создавать модели, сфокусированные на различных аспектах деятельности предприятия.
- Возможность коллективной работы. Анализ и моделирование процессов может требовать совместной работы нескольких человек. Для одновременной работы над моделями процессов CASE средства должны обеспечивать управление изменениями любыми фрагментами моделей и их модификацией при коллективном доступе.
- Построение прототипов. Прототипы процессов необходимы для того, чтобы на ранних стадиях изменения процессов можно было понять, насколько процесс будет соответствовать требованиям.
- Построение отчетов. CASE средства должны обеспечивать построение отчетов по всем моделям процессов с учетом взаимосвязи элементов. Такие отчеты необходимы для анализа моделей и определения возможностей по оптимизации. За счет отчетов обеспечивается контроль полноты и достаточности моделей, уровень декомпозиции процессов, правильность синтаксиса диаграмм и типов применяемых элементов.

Прежде чем CASE-средства применяются в конкретной ситуации при работе над решением конкретных задач, необходимо выполнять выбор этих инструментов. Выбор CASE-средств на самом деле зависит от многих факторов, начиная от технических характеристик создаваемого продукта, и заканчивая финансовыми возможностями заказчика.

Давайте рассмотрим «базовые» факторы, которые нужно принимать во внимание при выборе CASE-средств.

- Цели моделирования и анализа процессов. Исходя из целей моделирования, определяются необходимые методы, которые должны поддерживать CASE средства. Также цели моделирования определяют необходимый уровень детализации моделей и формы представления отчетов.
- Удобство для пользователей. Этот фактор определяет набор критериев для представления результатов моделирования наиболее понятным и приемлемым способом. Выбор CASE средств необходимо проводить с учетом того, чтобы пользователям приходилось затрачивать как можно меньше усилий на работу в среде CASE средств. CASE средства должны быть визуально и интуитивно понятны пользователям.
- Применение стандартных методологий. Этот фактор определяет критерии выбора CASE средств, связанные с применением стандартных методологий анализа и моделирования бизнес-процессов. Как правило, моделирование не заканчивается созданием новых моделей процессов. Модели используются для внедрения информационных систем управления и автоматизации процессов. За счет стандартизации обеспечивается упрощение взаимодействия между CASE средствами и различными информационными системами.
- Удобство эксплуатации. При выборе CASE средств необходимо учитывать такие характеристики как эффективность применения, сопровождаемость, переносимость моделей с одной системы на другую. Этот фактор в значительной степени связан с критериями, относящимися к техническим характеристикам аппаратного обеспечения.
- Трудоемкость. Этот фактор определяет набор критериев, связанных с освоением и изучением работы CASE средств. При выборе следует учесть, сколько времени потребуется на обучение пользователей.
- Субъективность. Данный фактор также не следует исключать из набора критериев по выбору CASE средств. При выборе могут существовать субъективные соображение выбора того или иного CASE средства, не связанные с рациональными критериями выбора.

Давайте дальше рассмотрим на примере применение CASE-средств для проектирования баз данных.

Как мы знаем, в базах данных различных типов хранится вся информация проекта. И к ним выполняется много обращений из различных функций. Поэтому, чем быстрее выполняются запросы, чем больше оптимизированы данные – тем быстрее и качественнее работает вся система вцелом.

К ключевым понятиям проектирования баз данных в таком ключе мы можем отнести:

- CASE-технологии - программная основа CASE-средств, применяемая для разработки и поддержки процессов жизненных циклов ПО, используемых в моделировании данных и генерации схем баз данных. Чаще всего программные коды в CASE-технологиях пишутся на языке SQL;
- концептуальное проектирование - построение обобщенной, не имеющей конкретики, модели базы данных с описанием ее объектов и связей между ними;
- логическое проектирование - создание схемы базы данных с учетом специфики конкретной модели данных (но не конкретной СУБД). Например, для реляционной модели данных логическая схема БД будет содержать определенный набор таблиц и связей между ними;
- физическое проектирование - построение схемы базы данных под конкретную СУБД. При таком проектировании учитываются ограничения на именование объектов базы данных, ограничения на определенные типы данных, физические условия хранения данных в БД (разделение по файлам и устройствам), возможность доступа к БД.

При выполнении процесса управления важным является следующее: обеспечить управляемость и контролируемость процессов разработки и сопровождения ПО. Для этого необходима точная и достоверная информация о состоянии ПО и его компонент в каждый момент времени, а также о всех предполагаемых и выполненных изменениях.

Для решения задач управления применяются методы и средства, обеспечивающие идентификацию состояния компонент, учет номенклатуры всех компонент и модификаций системы в целом, контроль за вносимыми изменениями в компоненты, структуру системы и ее функции, а также координированное управление развитием функций и улучшением характеристик системы.

Стандартные программные системы, относящиеся к классу CASE-средств и применяющиеся для организации процессов управления, предназначены для управления всеми компонентами проекта и ведения планомерной многоверсионной и многоплатформенной разработки силами команды разработчиков в условиях одной или нескольких локальных сетей. Понятие "проект" трактуется как совокупность файлов. В процессе работы над проектом промежуточное состояние файлов периодически сохраняется в архиве проекта, ведутся записи о времени сохранения, соответствии друг другу нескольких вариантов разных файлов проекта. Кроме этого, фиксируются имена разработчиков, ответственных за тот или иной файл, состав файлов промежуточных версий проекта и др. Это позволяет вернуться при необходимости к какому-либо из предыдущих состояний файла (например, при обнаружении ошибки, которую в данный момент трудно исправить).

Часто CASE-средства управления проектами оперируют сущностями и ролями пользователей. Например, могут быть выделены следующие роли в системе (и определен спектр обязанностей, распределенных между ролями)?

1. пользователи (Submitters) - имеют ограниченные права на внесение замечаний и сообщений об ошибках в базу данных системы управления проектами;
2. разработчики (Development Engineers) - имеют право производить основные операции с требованиями и замечаниями в базе данных системы. Если разработчики делятся на подгруппы, то для каждой подгруппы могут быть заданы отдельные списки прав доступа;
3. тестировщики (Quality Engineers) - имеют право производить основные операции с требованиями и замечаниями;
4. сопровождение (Support Engineers) - имеют право вносить любые замечания, требования и рекомендации в базу данных, но не имеют прав по распределению работ и изменению их приоритетности и сроков исполнения;
5. руководители (Managers) - имеют право распределять работы между исполнителями и принимать решения о их надлежащем исполнении. Руководителям разных групп могут заданы различные права доступа к базе данных системы.

В дополнение к этим пяти предопределенным группам, существует группа администратора базы данных и 11 дополнительных групп, которые могут быть настроены в соответствии со специфическими должностными обязанностями сотрудников.

Кажется, что все эти тонкости должны быть связаны только с управлением кадрового состава предприятия, но при создании программных продуктов, именно состав рабочих групп определяет пределы эффективности всей работы.

Именно с персоналом связаны многие функции CASE-инструментов, задействованных в управлении проектом. Например, программные средства такого типа часто содержат в себе функции формирования статистических отчетов: частотные, тренды и диаграммы распределения.

Частотные отчеты CASE-инструментов позволяют провести, например, оценку качества влияют тип методов тестирования, серьезность выявленных ошибок и значение дефектных модулей для функционирования всей системы, оценить число фатальных ошибок, приводящих к полной остановке разработки, и так далее, - все завязано на правила формирования отчетов и статистических срезов.

Тренды содержат информацию об изменениях того или иного показателя во времени и характеризуют стабильность и непрерывность процесса разработки. Они позволяют ответить на вопросы:

1. успевает ли группа разработчиков справляться с поступающими замечаниями;
2. улучшается ли качество программного продукта и какова динамика этого процесса;

3. как повлияло то или иное решение (увеличение числа разработчиков, введение скользящего графика, внедрение нового метода тестирования) на работу группы и т.п.

Диаграммы распределения - наиболее разнообразные и полезные для осуществления оперативного руководства формы отчетов. Они позволяют ответить на вопросы: какой метод тестирования более эффективен, какие модули вызывают наибольшее число нареканий, кто из разработчиков лучше справляется с конкретным типом заданий, нет ли перекоса в распределении работ между исполнителями, нет ли модулей, тестированию которых было уделено недостаточно внимания и т.д.

Хорошо, мы рассмотрели «управленческую» составляющую CASE-средств, применяемых в процессе разработки программного обеспечения.

Давайте сейчас перейдем к технической стороне вопроса создания ПО.

При проектировании баз данных с помощью CASE-средств выделяются и анализируются определенные бизнес-процессы, для которых создается БД, определяются взаимосвязи их элементов, оптимизируется их инфраструктура. CASE-средства позволяют существенно сократить время на разработку БД и уменьшить количество ошибок в них.

Для создания баз данных под наиболее распространенные СУБД чаще всего используются следующие CASE-средства:

- ERwin (Logic Works) - CASE-инструмент «среднего масштаба» для создания концептуальных и логических схем баз данных. Он позволяет редактировать различные наборы данных, представляя их в виде электронных таблиц, разрабатывать структуры баз данных, синхронизировать модели, скрипты и БД, настраивать шаблоны, выводить рабочую информацию в виде отчетов, строить удобные и понятные диаграммы, отображающие различные процессы в системе и взаимосвязи между ними.
- DataBase Designer (ORACLE) – «крупная» интегрированная CASE-среда, которая позволяет анализировать предметную область создания БД, выполнять программирование и проектирование, проводить оценку и тестирование, осуществлять сопровождение, обеспечивать качество, управлять конфигурацией и проектом, разрабатывать и анализировать требования к информационной системе.
- Dbdiagram.io – «простая», свободная среда визуального проектирования структуры базы данных, где можно описывать структуру БД при помощи простого условного языка со служебными символами и формой представления и видеть результат в виде схемы со связями. Схемой можно делиться или работать над ней совместно. Также можно экспортировать результат в SQL-скрипт и запустить уже на своем сервер базы данных.
- Draw.io – «простой редактор схем» для создания практически любых диаграмм, от блок-схем алгоритмов, до UML-диаграмм программных продуктов, баз данных или целых систем.

Что же касается разработки самого программного обеспечения, то CASE-средства также здесь используются повсеместно. Как мы уже говорили выше, в основном применяются CASE-средства третьего уровня.

Собственно говоря, CASE-средства для разработки ПО – это средства автоматизации разработки программ, то есть это инструменты автоматизации процессов проектирования и разработки программного обеспечения для системного аналитика, разработчика ПО и программиста.

Первоначально под CASE-средствами понимались только инструменты для упрощения наиболее трудоёмких процессов анализа и проектирования, но с приходом стандарта ISO/IEC 14102 CASE-средства стали определять, как программные средства для поддержки процессов жизненного цикла ПО.

Здесь основной целью CASE-технологии является разграничение процесса проектирования программных продуктов от процесса кодирования и последующих этапов разработки, максимально автоматизировать процесс разработки. Для выполнения поставленной цели CASE-технологии используют два принципиально разных подхода к проектированию: структурный и объектно-ориентированный.

Структурный подход предполагает декомпозицию (разделение) поставленной задачи на функции, которые необходимо автоматизировать. В свою очередь, функции также разбиваются на подфункции, задачи, процедуры. В результате получается упорядоченная иерархия функций и передаваемой информацией между функциями.

Структурный подход подразумевает использование определённых общепринятых методологий при моделировании различных информационных систем:

- SADT (structured analysis and design technique).
- DFD (data flow diagrams).
- ERD (entity-relationship diagrams).

Существует три основных типа моделей, используемых при структурном подходе: функциональные, информационные и структурные.

Основным инструментом объектно-ориентированного подхода является язык UML — унифицированный язык моделирования, который предназначен для визуализации и документирования объектно-ориентированных систем с ориентацией их на разработку программного обеспечения. Данный язык включает в себя систему различных диаграмм, на основании которых может быть построено представление о проектируемой системе.

Исходя из этих сфер применения мы можем определить основные типы систем, которые могут относиться к классу CASE-средств:

- средства анализа — предназначены для построения и анализа модели предметной области;
- средства проектирования баз данных;
- средства разработки приложений;
- средства реинжиниринга процессов;
- средства планирования и управления проектом;

- средства тестирования;
- средства документирования.

То есть с функциональной точки зрения, CASE-средствами являются любые инструменты, которые относятся к одному из следующих типов:

1. инструменты управления конфигурацией;
2. инструменты моделирования данных;
3. инструменты анализа и проектирования;
4. инструменты преобразования моделей;
5. инструменты редактирования программного кода;
6. инструменты рефакторинга кода;
7. генераторы кода;
8. инструменты для построения UML-диаграмм.

Ну и наконец, мы можем отметить, что сейчас все программные инструменты, используемые в повседневной работе программиста, содержат элементы CASE-средств в том или ином виде.

Мы ведем учет задач согласно гибкой методологии управления проектами прямо в нашей интегрированной среде разработки, в репозитории, в сложном многофункциональном блокноте.

Мы формируем отчеты, ставим и решаем задачи, улучшаем качество кода, не выходя из редактора.

Наши редакторы кода оснащены искусственным интеллектом и нейронными сетями, которые обучены на десятках тысяч репозиториях с исходными кодами и могут подсказать нам пути улучшения кода согласно «мировым практикам» и математическому расчету.

Так что главное – это понимать какой инструмент нам лучше использовать для какой ситуации, исключить дублирование и максимально повысить эффективность работы и ее качество.

ТЕМА 13. СОВРЕМЕННЫЙ СТЕК ТЕХНОЛОГИЙ ДЛЯ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ ИС И АС

Мы можем делить современный стек технологий для разработки программного обеспечения по разным принципам, но чаще всего классификация проводится по сфере применения языка и сопутствующих технологий.

Поэтому мы будем здесь рассматривать языки программирования с точки зрения их «отдаленности» от уровня аппаратуры и с точки зрения возможности использования низкоуровневых программ и аппаратуры компьютера при работе.

Исходя из такой концепции, мы можем выделить следующий стек разработки программного обеспечения информационных систем:

- Системные языки программирования.
- Скриптовые языки программирования.
- Языки высокого уровня.

Давайте рассмотрим их последовательно.

Системные языки программирования

Системные языки программирования – это языки, ориентированные в первую очередь на разработку системных программ, библиотек и приложений. Такие программы требуют иного подхода, нежели обычный прикладной софт. Языки этой группы можно называть «машинно-ориентированными».

Кроме того, языки общего назначения, как правило, сосредоточены на общих функциях, позволяющих программам, написанным на язык, чтобы использовать один и тот же код на разных платформах, в отличие от языков системного уровня, которые разработаны для производительности и простоты доступа к оборудованию, предоставляя разработчику, однако, достаточно высокоуровневые конструкции, приравнивающие системные языки по удобству к некоторым прикладным языкам.

Первым массовым языком, согласно такой концепции, был язык Си.

Си и C++

Он является компилируемым и статически типизированным языком. Изначально разработан для платформ типа Unix, однако потом был перенесен практически на все существующие платформы благодаря технологиям компиляции и кросскомпиляции. Из-за этого язык считается переносимым или кроссплатформенным.

Оказал влияние на множество современных языков типа C++, C#, Java, JavaScript.

Язык программирования Си разрабатывался в период с 1969 по 1973 годы в лабораториях Bell Labs, и к 1973 году на него была переписана большая часть

ядра UNIX. По мере развития язык сначала стандартизировали как ANSI C, а затем этот стандарт был принят комитетом по международной стандартизации ISO как ISO C, ставший также известным под названием C90. В стандарте C99 язык получил новые возможности, такие как массивы переменной длины и встраиваемые функции. А в стандарте C11 в язык добавили реализацию потоков и поддержку атомарных типов. Однако с тех пор язык развивается медленно, и в стандарт C18 попали лишь исправления ошибок стандарта C11.

Язык Си разрабатывался как язык системного программирования, для которого можно создать однопроходный компилятор. Стандартная библиотека также невелика. Как следствие данных факторов — компиляторы разрабатываются сравнительно легко.

Целью языка было облегчение написания больших программ с минимизацией ошибок по сравнению с ассемблером, следуя принципам процедурного программирования, но избегая всего, что может привести к дополнительным накладным расходам, специфичным для языков высокого уровня.

Основные особенности Си:

- простая языковая база, из которой в стандартную библиотеку вынесены многие существенные возможности, вроде математических функций или функций работы с файлами;
- ориентация на процедурное программирование;
- система типов, предохраняющая от бессмысленных операций;
- использование препроцессора для абстрагирования однотипных операций;
- доступ к памяти через использование указателей;
- небольшое число ключевых слов;
- передача параметров в функцию по значению, а не по ссылке (передача по ссылке эмулируется с помощью указателей);
- наличие указателей на функции и статические переменные;
- области видимости имён;
- структуры и объединения — определяемые пользователем собирательные типы данных, которыми можно манипулировать как одним целым.

С другой стороны, в Си отсутствуют:

- вложенные функции;
- прямое возвращение нескольких значений из функций;
- сопрограммы;
- средства автоматического управления памятью;
- встроенные средства объектно-ориентированного программирования;
- средства функционального программирования.

Отчасти в последующей версии языка C++, эти недостатки были исправлены.

C++ это компилируемый, статически типизированный язык программирования общего назначения.

Поддерживает такие парадигмы программирования, как процедурное программирование, объектно-ориентированное программирование, обобщённое программирование. Язык имеет богатую стандартную библиотеку, которая включает в себя распространённые контейнеры и алгоритмы, ввод-вывод, регулярные выражения, поддержку многопоточности и другие возможности. C++ сочетает свойства как высокоуровневых, так и низкоуровневых языков. В сравнении с его предшественником — языком C — наибольшее внимание уделено поддержке объектно-ориентированного и обобщённого программирования.

C++ широко используется для разработки программного обеспечения, являясь одним из самых популярных языков программирования. Область его применения включает создание операционных систем, разнообразных прикладных программ, драйверов устройств, приложений для встраиваемых систем, высокопроизводительных серверов, а также игр. Существует множество реализаций языка C++, как бесплатных, так и коммерческих и для различных платформ.

Синтаксис C++ унаследован от языка C. Изначально одним из принципов разработки было сохранение совместимости с C. Тем не менее C++ не является в строгом смысле надмножеством C; множество программ, которые могут одинаково успешно транслироваться как компиляторами C, так и компиляторами C++, довольно велико, но не включает все возможные программы на C.

У C++ богатая история развития и в ее контексте разработчики придерживались следующих концепций:

- Получить универсальный язык со статическими типами данных, эффективностью и переносимостью языка C.
- Непосредственно и всесторонне поддерживать множество стилей программирования, в том числе процедурное программирование, абстракцию данных, объектно-ориентированное программирование и обобщённое программирование.
- Дать программисту свободу выбора, даже если это даст ему возможность выбирать неправильно.
- Максимально сохранить совместимость с C, тем самым делая возможным лёгкий переход от программирования на C.
- Избежать разночтений между C и C++: любая конструкция, допустимая в обоих языках, должна в каждом из них обозначать одно и то же и приводить к одному и тому же поведению программы.
- Избегать особенностей, которые зависят от платформы или не являются универсальными.
- «Не платить за то, что не используется» — никакое языковое средство не должно приводить к снижению производительности программ, не использующих его.
- Не требовать слишком усложнённой среды программирования.

Golang

Golang, или Go — язык программирования, начало которого было положено в 2007 году сотрудниками компании Google.

Go поддерживает типобезопасность, возможность динамического ввода данных, а также содержит богатую стандартную библиотеку функций и встроенные типы данных вроде массивов с динамическим размером и ассоциативных массивов.

С помощью механизмов многопоточности Go упрощает распределение вычислений и сетевого взаимодействия, а современные типы данных открывают программисту мир гибкого и модульного кода. Программа быстро компилируется, при этом есть сборщик мусора и поддерживается рефлексия.

На данный момент поддержка официального компилятора, разрабатываемого создателями языка, осуществляется для операционных систем FreeBSD, OpenBSD, Linux, macOS, Windows, DragonFly BSD, Plan 9, Solaris, Android, AIX.

Golang предназначен для создания высокоэффективных программ, работающих на современных распределённых системах и многоядерных процессорах. Он может рассматриваться как попытка создать замену языкам Си и C++ с учётом изменившихся компьютерных технологий и накопленного опыта разработки крупных систем.

В качестве основных таких проблем других языков, которые должен решать Golang, называют следующие:

- медленную сборку программ;
- неконтролируемые зависимости;
- использование разными программистами разных подмножеств языка;
- затруднения с пониманием программ, вызванные неудобочитаемостью кода, плохим документированием и так далее;
- дублирование разработок;
- высокую стоимость обновлений;
- несинхронные обновления при дублировании кода;
- сложность разработки инструментария;
- проблемы межъязыкового взаимодействия.

Проанализировав эти проблемы, создатели Golang заложили следующие требования:

- Язык должен предоставлять небольшое число средств, не повторяющих функциональность друг друга.
- Простая и регулярная грамматика. Минимум ключевых слов, простая, легко разбираемая грамматическая структура, легко читаемый код.
- Простая работа с типами. Типизация должна обеспечивать безопасность, но не превращаться в бюрократию, лишь увеличивающую код. Отказ от иерархии типов, но с сохранением объектно-ориентированных возможностей.

- Отсутствие неявных преобразований.
- Сборка мусора.
- Встроенные средства распараллеливания, простые и эффективные.
- Поддержка строк, ассоциативных массивов и коммуникационных каналов.
- Чёткое разделение интерфейса и реализации.
- Эффективная система пакетов с явным указанием зависимостей, обеспечивающая быструю сборку.

Go создавался в расчёте на то, что программы на нём будут транслироваться в объектный код и исполняться непосредственно, не требуя виртуальной машины, поэтому одним из критериев выбора архитектурных решений была возможность обеспечить быструю компиляцию в эффективный объектный код и отсутствие чрезмерных требований к динамической поддержке.

У него есть некоторая альтернатива – это язык Rust.

Rust

Создатели Rust обещают, что их язык затмит собой C и C++. Разработчики Evrone использовали его в ряде действующих проектов, каждый из которых позволил нам набрать внушительный опыт. В статье поделимся нашими выводами о применении языка и его возможном будущем.

В C и C++ есть общепризнанные проблемные моменты, с которыми программистам приходится иметь дело из раза в раз. Это и трудности при компиляции, и высокий шанс утечки памяти, сам процесс управления этой памятью вручную, известные ошибки segfault. Задача Rust — обойти эти недоработки, одновременно увеличив производительность и повысив безопасность.

В чем особенность Rust:

1. Лаконичный синтаксис с ключевыми словами, похож на синтаксис C.
2. Кодовый анализатор, помогающий не допускать утечек памяти и не совершать ошибок при работе с многопоточностью.
3. Самостоятельное управление расположением данных в памяти (используя указатели).
4. Нет garbage collection.
5. Мощная статистическая типизация.

Разработкой языка занялся в свободное время сотрудник Mozilla Грэйдон Хор 14 лет назад, в 2006. Три года он действовал самостоятельно, пока в 2009-м к работе официально не подключилась Mozilla. В 2010 Rust был представлен официально.

Первый альфа-релиз языка состоялся в 2012. На Rust был разработан Servo, движок для веб-браузеров. В 2013 к работе над Servo присоединился Samsung, в результате чего код Servo был портирован на ARM-архитектуру.

Rust 1.0 вышел в 2015. В свой первый год Rust взял бронзу в голосовании «Любимый язык программирования» на портале Stack Overflow. Все последующие годы Rust занимает только первое место.

Так что это язык с интересной историей и хорошими «ожиданиями».

Преимущества и недостатки.

Аргументы «за»:

1. без проблем работает на Unix и Mac;
2. есть абстракции, которые существенно упрощают регулирование памяти вручную;
3. надёжная система взаимодействия с памятью, исключающая ошибки сегментации;
4. автоматическое представление способов исправить ошибки при компиляции;
5. компилятор содержит сборщик и менеджер пакетов, инструменты для тестирования и создания документации;
6. в безопасном коде нет возможности применять указатели (только ссылки на 100% реальные объекты);
7. доступное описание ошибок в шаблонах.

Аргументы «против»:

4. Компилятор слишком строго фиксирует вызовы к памяти;
5. Нет типичных для ООП-стиля наследования и классов.

Несмотря на это, Rust активно развивается и применяется в следующих сферах:

1. программирование клиентских приложений и веб-серверов;
2. blockchain;
3. создание собственных ОС;
4. написание программ и приложений по мониторингу систем и серверов;
5. разработка ПО общего назначения;
6. создание инфраструктуры;
7. написание движков для браузеров и игр.

Скриптовые языки программирования

Скриптовые языки программирования отличаются от компилируемых прежде всего тем, что они не переводятся сразу в бинарную форму, исполняемую на микропроцессоре, а, либо переводятся в промежуточную форму и потом исполняются, либо исполняются сразу специальными интерпретаторами.

В качестве примера рассмотрим Python.

Python

Это высокоуровневый язык программирования общего назначения с динамической строгой типизацией и автоматическим управлением памятью, ориентированный на повышение производительности разработчика, читаемости кода и его качества, а также на обеспечение переносимости написанных на нём программ. Язык является полностью объектно-ориентированным в том плане, что всё является объектами. Необычной особенностью языка является выделение блоков кода пробельными отступами. Синтаксис ядра языка минималистичен, за

счёт чего на практике редко возникает необходимость обращаться к документации. Сам же язык известен как интерпретируемый и используется в том числе для написания скриптов. Недостатками языка являются зачастую более низкая скорость работы и более высокое потребление памяти написанных на нём программ по сравнению с аналогичным кодом, написанным на компилируемых языках, таких как Си или C++.

Python является мультипарадигмальным языком программирования, поддерживающим императивное, процедурное, структурное, объектно-ориентированное программирование, метапрограммирование и функциональное программирование. Задачи обобщённого программирования решаются за счёт динамической типизации. Аспектно-ориентированное программирование частично поддерживается через декораторов, более полноценная поддержка обеспечивается дополнительными фреймворками. Такие методики как контрактное и логическое программирование можно реализовать с помощью библиотек или расширений. Основные архитектурные черты — динамическая типизация, автоматическое управление памятью, полная интроспекция, механизм обработки исключений, поддержка многопоточных вычислений с глобальной блокировкой интерпретатора (GIL), высокоуровневые структуры данных. Поддерживается разбиение программ на модули, которые, в свою очередь, могут объединяться в пакеты.

Эталонной реализацией Python является интерпретатор CPython, поддерживающий большинство активно используемых платформ и являющийся стандартом де-факто языка.

CPython компилирует исходные тексты в высокоуровневый байт-код, который исполняется в стековой виртуальной машине.

Стандартная библиотека включает большой набор полезных переносимых функций, начиная от функционала для работы с текстом и заканчивая средствами для написания сетевых приложений. Дополнительные возможности, такие как математическое моделирование, работа с оборудованием, написание веб-приложений или разработка игр, могут реализовываться посредством обширного количества сторонних библиотек, а также интеграцией библиотек, написанных на Си или C++, при этом и сам интерпретатор Python может интегрироваться в проекты, написанные на этих языках.

Python стал одним из самых популярных языков, он используется в анализе данных, машинном обучении, DevOps и веб-разработке, а также в других сферах, включая разработку игр. За счёт читабельности, простого синтаксиса и отсутствия необходимости в компиляции язык хорошо подходит для обучения программированию, позволяя концентрироваться на изучении алгоритмов, концептов и парадигм.

Применяется язык многими крупными компаниями, такими как Google или Facebook. По состоянию на октябрь 2021 года Python занимает первое место в рейтинге TIOBE популярности языков программирования с показателем 11,27%. «Языком года» по версии TIOBE Python объявлялся в 2007, 2010, 2018 и 2020 годах.

Языки высокого уровня

Здесь мы рассмотрим два современных языка высокого уровня, получившие возможность переноса с платформы на платформу, благодаря наличию виртуальной машины для запуска программ. Собственно говоря, эти языки работают там, где запускается их среда исполнения.

C#

Это объектно-ориентированный язык программирования. Разработан в 1998—2001 годах группой инженеров компании Microsoft.

C# относится к семье языков с C-подобным синтаксисом, из них его синтаксис наиболее близок к C++ и Java. Язык имеет статическую типизацию, поддерживает полиморфизм, перегрузку операторов (в том числе операторов явного и неявного приведения типа), делегаты, атрибуты, события, переменные, свойства, обобщённые типы и методы, итераторы, анонимные функции с поддержкой замыканий, LINQ, исключения, комментарии в формате XML.

C# разрабатывался как язык программирования прикладного уровня для CLR (Common Language Runtime, - общеязыковая среда исполнения для байт-кода, в который компилируются программы, написанные на .NET-совместимых языках программирования) и, как таковой, зависит, прежде всего, от возможностей самой CLR. Это касается, прежде всего, системы типов C#, которая отражает BCL (Base Class Library - стандартная библиотека классов платформы «.NET Framework». Программы, написанные на любом из языков, поддерживающих платформу .NET, могут пользоваться классами и методами BCL — создавать объекты классов, вызывать их методы, наследовать необходимые классы BCL и т. д.). Присутствие или отсутствие тех или иных выразительных особенностей языка диктуется тем, может ли конкретная языковая особенность быть транслирована в соответствующие конструкции CLR. Так, с развитием CLR от версии 1.1 к 2.0 значительно обогатился и сам C#; подобного взаимодействия следует ожидать и в дальнейшем (однако, эта закономерность была нарушена с выходом C# 3.0, представляющего собой расширения языка, не опирающиеся на расширения платформы .NET). CLR предоставляет C#, как и всем другим .NET-ориентированным языкам, многие возможности, которых лишены «классические» языки программирования. Например, сборка мусора не реализована в самом C#, а производится CLR для программ, написанных на C# точно так же, как это делается для программ на VB.NET, J# и др.

C# обладает богатой библиотекой, расширяется плагинами, работает с различными инструментами и библиотеками построения интерфейса приложений (Windows Forms, WPF и т.д.).

Используется для написания прикладных программ персональных ЭВМ, мобильных устройств (Xamarin) и сервером (.NET Core).

Java

Java — строго типизированный объектно-ориентированный язык программирования общего назначения, разработанный компанией Sun Microsystems (в последующем приобретённой компанией Oracle).

Приложения Java обычно транслируются в специальный байт-код, поэтому они могут работать на любой компьютерной архитектуре, для которой существует реализация виртуальной Java-машины. Дата официального выпуска — 23 мая 1995 года. Занимает высокие места в рейтингах популярности языков программирования (2-е место в рейтингах IEEE Spectrum (2020) и TIOBE (2021)/.

Достоинством подобного способа выполнения программ является полная независимость байт-кода от операционной системы и оборудования, что позволяет выполнять Java-приложения на любом устройстве, для которого существует соответствующая виртуальная машина. Другой важной особенностью технологии Java является гибкая система безопасности, в рамках которой исполнение программы полностью контролируется виртуальной машиной. Любые операции, которые превышают установленные полномочия программы (например, попытка несанкционированного доступа к данным или соединения с другим компьютером), вызывают немедленное прерывание.

Часто к недостаткам концепции виртуальной машины относят снижение производительности. Ряд усовершенствований несколько увеличил скорость выполнения программ на Java:

6. применение технологии трансляции байт-кода в машинный код непосредственно во время работы программы (JIT-технология) с возможностью сохранения версий класса в машинном коде,
7. обширное использование платформенно-ориентированного кода (native-код) в стандартных библиотеках,
8. аппаратные средства, обеспечивающие ускоренную обработку байт-кода (например, технология Jazelle, поддерживаемая некоторыми процессорами архитектуры ARM).

По данным сайта shootout.alioth.debian.org, для семи разных задач время выполнения на Java составляет в среднем в полтора-два раза больше, чем для C/C++, в некоторых случаях Java быстрее, а в отдельных случаях в 7 раз медленнее.

С другой стороны, для большинства из них потребление памяти Java-машиной было в 10—30 раз больше, чем программой на C/C++. Также примечательно исследование, проведённое компанией Google, согласно которому отмечается существенно более низкая производительность и боольшее потребление памяти в тестовых примерах на Java в сравнении с аналогичными программами на C++.

ТЕМА 14. СОВРЕМЕННЫЙ АППАРАТНЫЙ БАЗИС ДЛЯ РАЗМЕЩЕНИЯ ИС И АС

Современные информационные и автоматизированные системы могут размещаться практически на любом оборудовании, так как существуют высокоуровневые технологии разработки программного обеспечения и языки программирования, на которых можно создавать переносимые решения.

Здесь мы рассмотрим несколько характерных примеров аппаратных устройств, применимых для размещения ИС и АС в большинстве случаев.

Рассмотрим микроконтроллеры Atmel

Первыми мы рассмотрим семейство 8-битных микроконтроллеров фирмы Atmel, так как они широко распространены в нашей стране и за рубежом. Они используются на самых разных уровнях: от построения любительских конструкций и обучения школьников и студентов, до создания промышленных образцов ИС и АС.

Atmel AVR – это микроконтроллеры, разрабатывающиеся еще с 1996 г. Они имеют RISC-ядро (reduce instruction set computer – ядра с сокращенным набором команд). Первым микроконтроллером был AT90S1200 и его архитектурные принципы используются до сих пор в различных решениях.

Микроконтроллеры AVR имеют гарвардскую архитектуру (программа и данные находятся в разных адресных пространствах) и систему команд, близкую к идеологии RISC. Процессор AVR имеет 32 8-битных регистра общего назначения, объединённых в регистровый файл.

Система команд микроконтроллеров AVR весьма развита и насчитывает в различных моделях от 90 до 135 различных инструкций.

Большинство команд занимает только 1 ячейку памяти (16 бит).

Большинство команд выполняется за 1 такт.

Всё множество команд микроконтроллеров AVR можно разбить на несколько групп:

- команды логических операций;
- команды арифметических операций и команды сдвига;
- команды операции с битами;
- команды пересылки данных;
- команды передачи управления;
- команды управления системой.

Управление периферийными устройствами осуществляется через адресное пространство данных. Для удобства существуют «сокращённые команды» IN/OUT.

Сейчас существует несколько семейств микроконтроллеров с таким ядром:

- tinyAVR с памятью до 16 Кб, с маленьким числом ножек (4 – 18) и ограниченным набором периферийных устройств.

- megaARV – с памятью до 256 Кб, числом выводов до 86, аппаратными ускорителями вычислений и несколько расширенным набором команд.
- XMEGA ARV – с памятью до 384 Кб, дополнительными встроенными контроллерами, системами обработки событий и дополнительными блоками ввода-вывода.

Как правило, цифры после префикса обозначают объём встроенной flash-памяти (в КБ) и модификацию контроллера. А именно — максимальная степень двойки, следующая за префиксом, обозначает объём памяти, а оставшиеся цифры определяют модификацию (напр., ATmega128 — объём памяти 128 КБ; ATmega168 — объём памяти 16 КБ, модификация 8; ATtiny44 и ATtiny45 — память 4 КБ и так далее.

На основе стандартных семейств выпускаются микроконтроллеры, адаптированные под конкретные задачи:

- со встроенными интерфейсами USB, CAN, контроллером LCD;
- со встроенным радиоприёмопередатчиком — серии ATAxxxx, ATAMxxx;
- для управления электродвигателями — серия AT90PWMxxxx;
- для автомобильной электроники;
- для осветительной техники.

Кроме указанных выше семейств, ATMEL выпускает 32-разрядные микроконтроллеры семейства AVR32, которое включает в себя подсемейства AT32UC3 (тактовая частота до 66 МГц) и AT32AP7000 (тактовая частота до 150 МГц).

Огромное преимущество микроконтроллеров по отношению к другим устройствам проявляется в области управления устройствами, сбора данных и при работе в условиях ограниченных ресурсов.

В общем случае микроконтроллеры AVR имеют:

- до 86 многофункциональных двунаправленных «ножек» (линий ввода-вывода, называемых GPIO);
- до 32 внешних источников прерываний, обрабатываемых встроенным контроллером прерываний;
- несколько вариантов источников тактовых сигналов;
- встроенная энергонезависимая флещ-память до 256 Кб и множеством циклов перезаписи;
- встроенный отладчик, который подключается к ПЭВМ через специальную шину и позволяет отлаживать программы прямо в устройства со стороны ЭВМ;
- дополнительные блоки памяти EEPROM, SRAM;
- несколько таймеров с разными возможностями и разрядностями;
- ШИМ-модулятор для управления внешней нагрузкой;
- аналоговые компараторы;
- АЦП;
- ЦАП;

- последовательные интерфейсы связи, типа UART/USART, 1WARE, I2C, USB, ISO 7816, SPI, CAN и так далее;
- интерфейсы доступа к ЖК-экранам;
- интерфейсы доступа к внешней памяти;
- датчики температуры;
- режимы самопрограммирования и программирования по внешнему интерфейсу;
- возможность иметь начальный загрузчик в защищенной области памяти;
- ряд режимов пониженного энергопотребления и так далее.

Практически все эти возможности управляются программно, так что мы можем считать микроконтроллеры в некотором плане динамически управляемыми устройствами.

Следует также помнить, что цены на такие микроконтроллеры разные: от 36 руб. до 1000 руб. на конец 2021 г. в зависимости от возможностей.

При помощи микроконтроллеров можно строить свои собственные устройства, создавая конструкции самостоятельно, а можно покупать и использовать готовые комплекты разработчика.

Официальные наборы разработчика делятся на STKxxx Starter Kit, ISP, Dragon, JTAG, ICE, ONE, Arduino и прочие серии.

С точки зрения разработки программного обеспечения, можно использовать:

- Atmel AVR-studio – среду разработки с программатором и программным эмулятором ядер микроконтроллеров;
- AVR-eclipse – плагин для среды разработки Eclipse, позволяющий разрабатывать программы на C/C++ и ассемблере, программировать и отлаживать контроллеры, используя внешний набор инструментов (Atmel AVR Toolchain, WinAVR).
- Avrdude — средство для прошивки микроконтроллеров
- Code::Blocks — кроссплатформенная среда разработки, также используемая для программирования на языке Си и C++;
- PonyProg — универсальный программатор через LPT-порт, COM-порт (поддерживается и USB-эмулятор COM-порта);
- IAR AVR — коммерческая среда разработки для микроконтроллеров AVR
- Bascom-avr — среда разработки, основанная на Basic-подобном языке программирования.
- CodeVisionAVR — компилятор C и программатор — CVAVR, генератор начального кода.
- Proteus — симулятор электрических цепей, компонентов, включая различные МК и другое периферийное оборудование.

Также архитектура AVR позволяет применять операционные системы при разработке приложений, например, FreeRTOS, uOS, ChibiOS/RT, scmRTOS(C++), TinyOS, Femto OS и др, а также Linux на AVR32.

Таким образом, при помощи микроконтроллеров Atmel мы можем построить самые разные системы.

Например, AVR tiny чаще всего используются для построения следующих устройств:

- электронные игрушки;
- различные датчики в автомобильной промышленности;
- детекторы дыма и пламени, датчики температур, измерители разных величин;
- недорогие зарядные устройства, индикаторы напряжения и тока;
- пульты управления для разнообразной бытовой и промышленной техники;
- другие не дорогие и миниатюрные электронные устройства.

Чипы семейства "mega", "xmega" и 32-bit AVR применяются в более сложных устройствах:

- робототехника;
- спутниковые навигационные системы;
- функциональные разрядно-зарядные устройства с программированием;
- сложные дистанционные системы управления;
- сетевые устройства;
- быстродействующие системы для передачи и обработки данных;
- сложная бытовая техника;
- устройства ввода и отображения информации с тач-скринами (Touch-screen);
- другие многофункциональные устройства.

Другим популярным типом микроконтроллеров сегодня является продукция фирмы STMicroelectronics – STM.

STM

Компания STMicroelectronics одной из первых вывела на рынок семейство микроконтроллеров на ядре ARM Cortex-M3 и на сегодняшний день по праву занимает лидирующее место среди производителей микроконтроллеров на этом ядре. Все началось в 2007 году с двух семейств — Performance Line (STM32F103) и Access Line (STM32F101). Компания постоянно работает как над расширением номенклатуры семейства, так и над улучшением характеристик, не забывая при этом также пополнять программную составляющую продукта. На сегодняшний момент STM32 уже состоит из 10 линеек.

Они подходят для всевозможных применений — микроконтроллеры с высокой производительностью, недорогие микроконтроллеры общего применения, микроконтроллеры с ультранизким энергопотреблением, микроконтроллеры со встроенным радиомодулем для беспроводных решений, и все это — на одном ядре ARM Cortex-M3.

Семейство микроконтроллеров STM32 состоит из 16 серий микроконтроллеров: F0, F1, F2, F3, F4, F7, L0, L1, L4, L4+, L5, G0, G4, H7, WB, WL[1]. Каждая из серий базируется на одном из ядер ARM: Cortex-M33, Cortex-M7F, Cortex-M4F, Cortex-M3, Cortex-M0+, Cortex-M0.

Производитель делит все серии микроконтроллеров STM32 на 4 платформы (группы):

6. Высокопроизводительные: F2, F4, F7, H7.
7. Широкого применения: F2, F4, F7, H7.
8. Сверхнизкого потребления: L0, L1, L4, L4+, L5.
9. Беспроводные: WB, WL.

Сложно обобщить свойства различных семейств STM, но мы попытаемся объединить список возможностей, которые они в своих разных вариантах могут предоставить разработчику:

10. Тактовая частота до 550 МГц.
11. Несколько вычислительных ядер.
12. Аппаратная поддержка вычислений с плавающей точкой, двойной точности.
13. Наличие ядер цифровой обработки сигналов DSP.
14. До нескольких мегабайт встроенной памяти.
15. Богатый выбор интерфейсов: CAN, UART/USART, SPI, ISO 7816, I2C, CDIO, Ethernet (до 100 Мбит/с), цифровой интерфейс камеры DCMI.
16. ЦАП/АЦП.
17. До 140 линий ввода-вывода.
18. Часы реального времени и табличные таймеры.
19. Аппаратный генератор случайных чисел.
20. Криптосопроцессор с методами шифрования DES, TDES, AES, SHA-1, ND5.
21. Контроллеры ЖК экранов.
22. Матрица внутренних шин с распределением ведущих и ведомых устройств в микроконтроллере, арбитражом и разными режимами доступа к памяти.
23. Рабочее напряжение от 1.8 до 3.6 В.

Под ARM-архитектуру существует довольно широкий выбор программных средств разработки. Приведем лишь основные и самые популярные программные пакеты на российском рынке:

- IAR Systems
- Keil
- Raisonancw
- Atollic
- OpenSource решения типа Eclipse.

Наиболее популярными (но и самыми дорогими) среди разработчиков для разработки ПО под ARM архитектуру являются инструментарии от компаний Keil и IAR Systems. Это обусловлено наиболее продвинутыми С-инструментариями с точки зрения оптимизации и компактности кода. Также,

помимо лидирующих позиций в С-инструментариях, данные компании предоставляют широкие наборы дополнительного ПО — операционные системы реального времени, USB-стеки, TCP/IP-стеки и многое другое, но за дополнительную плату. К тому же компания Keil принадлежит ARM, и при пользовании услугами этих двух компаний вы получаете очень хорошую техническую поддержку. Но мы все же остановимся на инструментарии от IAR Systems. Выбор обусловлен универсальностью инструментария, поддерживающего большинство известных нам архитектур микроконтроллеров таких производителей как STMicroelectronics, Texas Instruments, Microchip, Atmel и т.д.

Также следует отметить популярность средств на основе компилятора GCC. Существуют как платные их варианты, так и бесплатные. Помимо всего, GCC является лидером по количеству поддерживаемых процессоров и операционных систем. Как пример варианта платных средств в сводной таблице мы привели инструментарии от компаний Raisonance и Atollic. По сравнению с двумя ранее описанными вариантами вы получаете за гораздо меньшие деньги полноценный С-инструментарий со средой разработки и технической поддержкой. Также существует вариант полностью бесплатного инструментария, например, среда разработки Eclipse и компилятор GCC.

Как и в случае с AVR, на рынке существует огромный выбор оценочных плат для STM32 как от STMicroelectronics, так и от сторонних производителей. Например, недорогие и оригинальные модули «Махаон» и «Барракуда» предлагает компания Терраэлектроника. Но наша основная цель — использовать для ознакомления и изучения микроконтроллеров семейства STM32 доступные и по возможности недорогие модули. Именно для таких целей компания STMicroelectronics разработала линейку оценочных плат «Discovery»: для восьмибитных микроконтроллеров — STM8S-Discovery и STM8L-Discovery, для 32-битных — STM32VLDiscovery. Особенность данных оценочных плат заключается в завершенном решении для старта разработки программного обеспечения на микроконтроллерах — сам микроконтроллер с необходимой обвязкой и внешними компонентами, а также интегрированный программатор-отладчик ST-Link. Это полноценное решение, не требующее дополнительных затрат, а рыночная цена плат «Discovery» составляет 10…15$. Используя эти платы в собственных разработках, можно применять для программирования и отладки собственных приложений встроенный ST-Link через выведенный внешний разъем. С учетом всего вышеописанного, для широкого круга радиолюбителей и разработчиков коммерческих компаний отпадает необходимость в самостоятельном изготовлении отладочных плат и программаторов

Большинство этих систем работают «вокруг» стандартной библиотеки STM. Она написана в соответствии со стандартом ANSI C и может использоваться с любым компилятором. Структура библиотеки не так сложна, как кажется на первый взгляд, и состоит из двух взаимодополняющих составляющих.

Первая составляющая — заголовочные файлы и файлы реализации всей периферии микроконтроллеров STM32 — STM32F10x_StdPeriph_Driver. Вся функциональность периферийных модулей описана в заголовочных файлах и файлах реализации. Например, для портов ввода-вывода это два файла — stm32f10x_gpio.h и stm32f10x_gpio.c.

Вторая составляющая — заголовочные файлы и файлы реализации самого ядра ARM Cortex-M3 от компании ARM — CMSIS (ARM® CortexTM Microcontroller Software Interface Standard). Ядро ARM Cortex-M3 выходит за рамки обычного понятия ядра микроконтроллера и представляет собой мини-микроконтроллер с периферией — встроенные системный таймер, контроллер прерываний и т.д. CMSIS предоставляет собой константы и определения, функции доступа к регистрам и периферийным модулям ядра, независимый интерфейс для операционных систем реального времени (RTOS).

Продукты компании STMicroelectronics сейчас, фактически, захватывает рынок микроконтроллеров, так как можно подобрать ядро практически для любой системы. И цены на STM и комплекты разработчика очень лояльные к клиентам.

Делать с STM можно точно такие же проекты, как и на AVR, а в некоторых случаях, STM предоставляет даже большие возможности за те же самые деньги.

Программируемые логические интегральные схемы (ПЛИС)

Напомним, что ПЛИС — это электронный компонент (интегральная микросхема), используемый для создания конфигурируемых цифровых электронных схем. В отличие от обычных цифровых микросхем, логика работы ПЛИС не определяется при изготовлении, а задаётся посредством программирования (проектирования). Для программирования используются программатор и IDE (отладочная среда), позволяющие задать желаемую структуру цифрового устройства в виде принципиальной электрической схемы или программы на специальных языках описания аппаратуры: Verilog, VHDL, AHDL и др.

Некоторые производители для своих ПЛИС предлагают программные процессоры (софт-процессоры), которые можно модифицировать под конкретную задачу, а затем встроить в ПЛИС. Тем самым:

1. обеспечивается увеличение свободного места на печатной плате (возможность уменьшения размеров платы);
2. упрощается проектирование самой ПЛИС;
3. увеличивается быстродействие ПЛИС.

Сейчас основными производителями ПЛИС (которые наиболее известны в России) являются Altera и Xilinx.

В любом случае, ПЛИС широко используется для построения различных по сложности и по возможностям цифровых устройств, например:

1. устройств с большим количеством портов ввода-вывода (бывают ПЛИС с более чем 1000 выводов («пинов»));
2. устройств, выполняющих цифровую обработку сигнала (ЦОС);
3. цифровой видеоаудиоаппаратуры;

4. устройств, выполняющих передачу данных на высокой скорости;
5. устройств, выполняющих криптографические операции, систем защиты информации;
6. устройств, предназначенных для проектирования и прототипирования интегральных схем специального назначения (ASIC);
7. устройств, выполняющих роль мостов (коммутаторов) между системами с различной логикой и напряжением питания;
8. реализаций нейрочипов;
9. устройств, выполняющих моделирование квантовых вычислений;
10. устройств, выполняющих обработку радиолокационной информации.

Сама идея ПЛИС не нова. Еще в Советском Союзе были распространены БИС и СБИС (большие и сверхбольшие интегральные схемы), которые также позволяли «погружать» целые схемы внутрь одного кристалла.

За рубежом разработка ПЛИС начиналась в 1970 г. в компании TI (Texas Instruments) – тогда были созданы «масочные» интегральные схемы (программируемые при помощи маски). Тогда же появился термин PLA – программируемая логическая матрица.

Далее были долгие годы развития, пока мы не получили ту ПЛИС, что известна на сегодняшний день. Сейчас с ПЛИС ассоциируется термин FPGA (field-programmable gate array).

ПЛИС содержат блоки умножения-суммирования, которые широко применяются при обработке сигналов (DSP, англ. digital signal processing), а также логические элементы (как правило, на базе таблиц перекодировки — таблиц истинности) и их блоки коммутации. FPGA обычно используются для обработки сигналов, имеют больше логических элементов и более гибкую архитектуру, чем CPLD. Программа для FPGA хранится в распределённой памяти, которая может быть выполнена как на основе энергозависимых ячеек статического ОЗУ (подобные микросхемы производят, например, фирмы «Xilinx» и «Altera») — в этом случае программа не сохраняется при исчезновении электропитания микросхемы, так и на основе энергонезависимых ячеек flash-памяти или перемычек antifuse (такие микросхемы производит фирма «Actel» и «Lattice Semiconductor») — в этих случаях программа сохраняется при исчезновении электропитания. Если программа хранится в энергозависимой памяти, то при каждом включении питания микросхемы необходимо заново конфигурировать её при помощи начального загрузчика, который может быть встроен и в саму FPGA. Альтернативой ПЛИС FPGA являются более медленные цифровые процессоры обработки сигналов. FPGA применяются также, как ускорители универсальных процессоров в суперкомпьютерах

Для всех ПЛИС можно описать стандартную процедуру разработки «прошивки», то есть конфигурации:

1. Задание принципиальной электрической схемы или программы на специальных языках описания аппаратуры: Verilog, VHDL, AHDL и др.
2. Логический синтез с помощью программ-синтезаторов (получение списка электрических соединений (в виде текста) из абстрактной модели, записанной на языке описания аппаратуры).

3. Проектирование печатной платы устройства с помощью системы автоматизированного проектирования (САПР) печатных плат (Altium Designer, P-CAD и др.), на которой размещается микросхема ПЛИС и прочие электронные компоненты (резисторы, конденсаторы, генераторы, АЦП, разъёмы и т.д.).
4. Создание файла конфигурации ПЛИС.
5. Загрузка файла в микросхему ПЛИС или отдельную микросхему памяти конфигурации. В результате загрузки конфигурации микросхема ПЛИС обретает заданную функциональность.

ПЛИС программируется, а точнее конфигурируется или при помощи блоков элементов в визуальном средстве проектирования или при помощи исходных кодов на языках VHDL, Verilog.

Средами для работы служат Altera Quartus II или Xilinx IDE.

В этих программах разработчику доступны и редакторы кода и средства отладки, а также инструменты для оптимизации проекта и его обслуживания.

ТЕМА 15. НАТИВНЫЕ ПРОГРАММНЫЕ СРЕДСТВА, ИНСТРУМЕНТЫ И БИБЛИОТЕКИ

Нативные приложения отличаются от кроссплатформенных. Если разработчики в процессе написания приложения пользуются принятым для конкретной платформы языком программирования, будь то Objective-C и Swift для iOS или Java или Kotlin для Android, такое приложение будет называться нативным (от англ. native — родной, естественный).

Преимущества нативных приложений:

- скорость работы и отклика интерфейса. Приложение реагирует на нажатия мгновенно, практически отсутствуют задержки в анимации, скроллировании, получении и выводе данных;
- понятный и простой доступ к функциям и датчикам устройства. Для разработчика не представляет проблемы работа с геолокацией, пуш-уведомлениями, съёмкой фото и видео через камеру, звуком, акселерометром и другими датчиками;
- возможность углублённой работы с функциями смартфона. Такие вещи, как анимации, создание сложных интерфейсов и работа нейросетей прямо на устройствах реализуются не просто, но прогнозируемо;
- родной для платформы интерфейс. Нативные приложения обычно оперируют «платформенными» элементами интерфейса: меню, навигация, формы и все остальные элементы дизайна берутся от операционной системы и потому привычны и понятны пользователю.

Недостаток у такого подхода один — дороговизна разработки и поддержки. Для каждой платформы надо писать свой код, следовательно для каждой платформы должен быть как минимум один свой профессиональный разработчик, а чаще — своя команда.

С ростом рынка мобильных приложений услуги такого разработчика, например, стали не просто дороги, а очень дороги.

Нативная разработка основывается на понятиях компиляции и кросскомпиляции.

По своему определению, компилятор — это программное обеспечение, переводящее текст программы на языке высокого уровня в элементарную программу на языке низкого уровня, которая может быть выполнена микропроцессором ЭВМ. В качестве языка низкого уровня зачастую используется язык ассемблера или похожий на него. Входной информацией для компилятора является сам текст программы и параметры (флаги) компиляции. Результат преобразования текста программы — это фактически эквивалентное выражение заложенного в программу алгоритма на языке, понятной машине.

Так же компилятор выполняет трансляцию и компоновку программы из модулей и библиотек. Компоновка подразумевает линковку или связывание статическим или динамическим методом.

Статическое связывание подразумевает прямое включение программного кода вызываемых функций в общий код для преобразования в машинный код, а

динамическое связывание подразумевает проставление ссылок на подключаемые модули.

Различают следующие виды компиляторов:

- Векторизующий. Компилирует исходный код в машинный код для компьютеров, оснащенных векторным процессором.
- Гибкий. Сконструирован по модульному принципу, управляется таблицами и запрограммирован на языке высокого уровня или реализован с помощью компилятора компиляторов.
- Диалоговый. См.: диалоговый транслятор.
- Инкрементальный. Повторно транслирует/компонует фрагменты программы и дополнения к ней без перекомпиляции всей программы.
- Интерпретирующий (пошаговый). Последовательно выполняет независимую компиляцию каждого отдельного оператора (команды) исходной программы.
- Компилятор компиляторов. Компилятор, воспринимающий формальное описание языка программирования и генерирующий компилятор для этого языка.
- Отладочный. Устраняет отдельные виды синтаксических ошибок.
- Резидентный. Постоянно находится в оперативной памяти и доступен для повторного использования многими задачами.
- Самокомпилируемый. Написан на том же языке, с которого осуществляется компиляция.
- Универсальный. Основан на формальном описании синтаксиса и семантики входного языка. Важными составными частями такого компилятора являются: ядро, синтаксический и семантический загрузчики.

Сами по себе виды компиляции классифицируются следующим образом:

- Пакетная. Компиляция нескольких исходных модулей в одном пункте задания.
- Построчная. То же, что и интерпретация.
- Условная. Компиляция, при которой транслируемый текст зависит от условий, заданных в исходной программе директивами компилятора. Так, в зависимости от значения некоторой константы, можно включать или выключать трансляцию части текста программы.

Как правило, код, получаемый на выходе компилятора, уже готов к выполнению на целевой архитектуре. И это ключевое свойство компилятора, так как компиляция выполняется именно на платформе. Для другой ЭВМ с другой операционной системы или для другого процессора потребуется перекомпиляция, то есть другая сборка программы. Это вызвано тем, что машинный код для разных архитектур может содержать различные инструкции и может использовать различные аппаратные блоки внутри микропроцессора. Такая архитектура и такая платформа для компилятора называется целевой.

Некоторые компиляторы используют промежуточный рабочий вариант между программным кодом и машинным кодом или предварительный вариант. Последний называется прекомпиляцией, а первый вариант аналогичен подходу,

применяемому на платформах Java и .NET. Их промежуточные варианты исполняемого кода называют байт-кодами, и они выполняются виртуальными машинами, которые, в свою очередь, уже вызывают на выполнение блоки машинного кода непосредственно в системе.

Выполнение промежуточного кода перед непосредственным исполнением машинного кода называется “компиляцией на лету” (JIT-компиляцией).

Кросскомпиляция позволяет на одной машине формировать исполняемый код для другой машины с учетом особенностей целевой платформы, то есть микропроцессора и операционной системы.

На этапе компиляции и тем более кросскомпиляции может выполняться дополнительная оптимизация промежуточного и конечного исполняемого кода с учетом архитектурных особенностей целевой платформы. В основном это заключается в расширенном использовании систем команд и дополнительных аппаратных блоков инструкций внутри микропроцессора.

Следует отметить, что существуют и программы, которые выполняют обратные действия - переводят код из низкоуровневого представления в высокоуровневое, то есть, фактически, восстанавливают исходный программный код из кодов на языке ассемблера или из байт-кода.

Любой компилятор состоит из двух основных частей:

- Компоновщика.
- Линковщика.

Часто компоновщик является внешним компонентом компилятора, реализован в виде отдельной программы и вызывается при необходимости.

Компилятор может быть реализован и как своеобразная программа-менеджер, для трансляции программы вызвающая соотвествующий транслятор (трансляторы - если разные части программы написаны на разных языках программирования) и затем - для компоновки программы, - вызывающая компоновщик. Ярким примером такого компилятора является имеющаяся во всех UNIX-системах (и Linux-системах в том числе) утилита make (имеются реализации утилиты make и в других системах, в частности в Windows-системах).

Процесс компиляции состоит из следующих основных фаз:

- Лексический анализ.
- Синтаксический или иначе грамматический анализ.
- Семантический анализ.
- Оптимизация.
- Генерация кода исполняемого.

На фазе лексического анализа последовательность символов исходного файла преобразуется в последовательность лексем.

Далее на этапе синтаксического анализа, последовательность лексем преобразуется в дерево разбора.

Семантический анализ обрабатывает дерево разбора с целью установления его семантики, то есть смысла. Например, здесь привязываются идентификаторы к их определениям, к типам данных, проводится проверка совместимости типов

данных, выполняется определение результирующих типов данных выражений и так далее. Результат обычно называется «промежуточным представлением/кодом», и может быть дополненным деревом разбора, новым древом, абстрактным набором команд или чем-то еще, удобным для дальнейшей обработки.

После семантического анализа выполняется оптимизация. Здесь удаляются избыточные команды и упрощается программный код с сохранением его смысла, то есть реализуемого им алгоритма. Оптимизирующий компонент производит предварительное вычисление выражений, результаты которых далее практически являются константами. Оптимизация, может быть, на разных уровнях и этапах — например, над промежуточным кодом или над конечным машинным кодом.

Генерация исполняемого кода — это заключительный этап работы компилятора, когда из промежуточного представления порождается код на целевом языке. На этом этапе также выполняется компоновка программы.

В конкретных реализациях компиляторов эти фазы могут быть разделены или, наоборот, совмещены в том или ином виде.

Написание компилятора может потребоваться в самых разнообразных условиях – для различных языков, целевых платформ, с различными требованиями к работе компилятора и т.д. Например, одна из типичных проблем написания компилятора связана с тем, что на целевой платформе могут отсутствовать подходящие средства для разработки.

Именно так обычно обстоят дела при создании новой компьютерной архитектуры: на начальном этапе жизни компьютера системные программисты не имеют никаких средств разработки, кроме системы команд самого компьютера. Более того, на этом этапе даже привычный всем ассемблер может отсутствовать).

Бывают и обратные ситуации, когда компилятор создается для еще не существующей целевой платформы, так сказать, на будущее.

Таким образом, цели и условия написания компиляторов могут очень сильно варьироваться от одного проекта к другому. Поэтому существует целый ряд методик разработки компиляторов.

Далее остановимся на наиболее распространенных методиках разработки компиляторов.

- Прямой метод, в котором целевым языком и языком реализации является язык ассемблера.
- Метод раскрутки.
- Использование кросс-трансляторов.
- Использование виртуальных машин.
- Компиляция “на лету”.

Прямой метод разработки компилятора применяется тогда, когда средства трансляции уже существуют и можно переводить программный код непосредственно на язык ассемблера. Тогда для разработчиков процедура создания компилятора проста: начальная и конечная точка, а также инструменты определены и существуют.

Метод раскрутки, фактически появился первым еще в 1950 году и заключался в том, что целевой транслятор пишется на том же языке программирования, для трансляции которого создается. Тогда транслятором выполняется создание исполняемых файлов из исходного кода самого транслятора.

Метод используется и по сей день для переноса трансляторов на новые платформы. Применялся, например, для создания современных трансляторов многих языков программирования, включая языки Basic, C, Pascal, Haskell, Common List, Scheme, Java, Python, Scala. Следует упомянуть, что метод раскрутки позволяет создать транслятор, который генерирует сам себя.

Кросс-транслятор, это инструмент (компилятор), который априори работает на одной платформе и создает код для другой платформы. В рамках теории кросс-трансляторов самыми важными вопросами являются переносимость программного кода и переносимость компиляторов. Под переносимостью здесь будем понимать способность программы, которая без дополнительных инструментов может выполняться как минимум на двух платформах: на исходной и на целевой.

Большинство существующих компиляторов порождают непереносимый программный код, так как в них применяется объектный подход. А одним из способов получения переносимых объектных программ является генерация объектной программы на языке более высокого уровня, чем язык ассемблера. Такие компиляторы иногда называют конвертерами.

Упоминание использования виртуальных машин в контексте термина “компилятор” применимо совместно с понятием компиляции на лету (JIT) и динамической трансляции. Это технология увеличения производительности программных систем, использующих байт-код, путём компиляции байт-кода в машинный код непосредственно во время работы программы. Таким образом достигается высокая скорость выполнения по сравнению с интерпретируемым байт кодом.

Применение байт-кодов как основного исполняемого элемента позволяет достичь высокой производительности программ, сравнимой с производительностью программ, разработанных на языках типа Си.

В языках, компилирующихся в байт-код, таких как Lua, Perl, GNU CLISP или Java, исходный код транслируется в одно из промежуточных представлений, которое не является исполняемым или машинным. Байт-код может переноситься на другие платформы и выполняться на них. Из-за этой особенности упомянутые инструменты называют кроссплатформенными.

Из-за необходимости дополнительной обработки, то есть интерпретации, байт-код практически в любой системе выполняется значительно медленнее машинного кода со сравнимой функциональности, однако он более переносим. Вплоть до того, что код не зависит от операционной системы и модели процессора, а проблема совместимости и использования специфических процессорных ресурсов решается самой платформой.

Для того, чтобы ускорить выполнение байт-кода, используется динамическая компиляция, когда виртуальная машина транслирует псевдокод в

машинный код непосредственно перед его первым исполнением (и при повторных обращениях к коду исполняется уже скомпилированный вариант).

Наиболее популярной разновидностью динамической компиляции является CIL-код, который также компилируется в код целевой машины JIT-компилятором. Примером набора инструментов и компиляторов, реализующих концепцию CIL является фреймворк .Net от компании Microsoft.

Также можно упомянуть существование такого интересного подхода как раздельная компиляция.

Раздельная компиляция — это трансляция частей программы по отдельности с последующим объединением их компоновщиком в единый загрузочный модуль. Вскоре после появления первых крупных программных комплексов возникла необходимость разделять программы на части и выделять библиотеки, которые можно компилировать независимо друг от друга. При трансляции каждой части программы компилятор порождает объектный модуль, содержащий дополнительную информацию, которая потом, при компоновке частей в исполняемый модуль, используется для связывания и разрешения ссылок между частями.

Далее рассмотрим такие инструменты как интерпретаторы и трансляторы.

Интерпретатор — это разновидность транслятора, который выполняет интерпретацию программы, то есть ее пошаговое или блочное выполнение.

По своей сути, это изначально простой построчный анализ исполняемого кода программы или запроса. В отличие от компилятора, который обрабатывает исходный код целиком, собирает модули и переводит в тот или иной вид исполняемого кода, интерпретатор не просматривает код наперед. Поэтому компилятор способен выявить все ошибки программы перед сборкой, а интерпретатор может обнаружить ошибку только во время выполнения. Современные версии интерпретаторов проводят предварительный анализ программного кода, но, зачастую, ограничиваются обнаружением лексических и синтаксических ошибок, “оставляя” другие типы ошибок на момент выполнения.

Первым интерпретируемым языком являлся язык Lisp. Первый его интерпретатор был создан еще в 1958 году Стивом Расселом, когда он обнаружил, что функция eval (вычисление входного выражения в свободной форме) может быть вставлена в программный код. На этом принципе и работают интерпретаторы - программа выполняют другие программы.

Существует несколько типов интерпретаторов:

- Простой интерпретатор.
- Интерпретатор компилирующего типа.

Простые интерпретаторы стараются выполнить программу сразу после получения исходного кода как входного параметра. При этом они выполняют разбиение программного кода на блоки минимального размера и буферизируют. Каждый блок отправляется на выполнение с отслеживанием возможных ошибок. Из этого следуют очевидные достоинства и недостатки подхода. Достоинством является мгновенная реакция интерпретатора на задание, а недостатком - ошибки отслеживаются только в момент их исполнения и в момент возникновения самой ошибки.

Интерпретатор компилирующего типа — это программный комплекс, состоящий из предварительного компилятора и интерпретатора. Предварительный компилятор переводит исходный код программы и исполняемый байт код, а сам по себе внутренний интерпретатор уже его исполняет. Достоинством подобных систем является их быстродействие за счет вынесение анализа исходного кода в отдельный поток и выполнение его отдельным инструментом. Недостаток - высокие требования к вычислительным ресурсам.

Последний тип интерпретаторов используется в современных языках программирования. Это особенно актуально для таких инструментов, как Java, так как ее виртуальная машина фактически играет роль предварительного компилятора.

Часто в предварительный компилятор включают простой анализатор кода для оптимизации последующего исполнения. Причем исходный код для такой обработки не обязательно должен иметь текстовый формат или быть байт-кодом, который понимает только данный интерпретатор, это может быть машинный код какой-то существующей аппаратной платформы.

Некоторые интерпретаторы могут работать в диалоговом режиме, то есть реализуют цикл из следующих трех действий:

1. Чтение программы.
2. Вычисление.
3. Печать результатов.

В таком режиме интерпретатор считывает конструкции языка из потока ввода, выполняет их и печатает результаты в поток вывода или в поток ошибок.

Уникальным примером языка программирования с интерпретатором компилирующего типа является язык Forth. Он способен работать как в режиме интерпретации, так и компиляции входных данных с переключением. Переключение между этими режимами происходит в произвольный момент, как во время трансляции исходного кода, так и во время работы программы.

Режимы интерпретации встречаются не только в программном обеспечении, но и на уровне аппаратуры. Многие аппаратные платформы и микропроцессоры интерпретируют программный код в набор блоков микропрограмм, подгружаемых из памяти процессора и управляющих системой.

Обобщенный алгоритм работы интерпретатора можно описать следующим образом:

1. прочитать инструкцию;
2. проанализировать инструкцию и определить соответствующие действия;
3. выполнить соответствующие действия;
4. если не достигнуто условие завершения программы, прочитать следующую инструкцию и перейти к пункту 2.

Трансляция программы — преобразование программы, представленной на одном из языков программирования, в программу на другом языке. Транслятор обычно выполняет также диагностику ошибок, формирует словари идентификаторов, выдаёт для печати текст программы и т. д.

Язык, на котором представлена входная программа, называется исходным языком, а сама программа — исходным кодом. Выходной язык называется целевым языком, а выходная программа — объектным кодом.

На современном рынке программного обеспечения распространены следующие современные виды трансляторов:

4. Диалоговый транслятор.
5. Синтаксически-ориентированный транслятор.
6. Однопроходной транслятор.
7. Многопроходной транслятор.
8. Оптимизирующий транслятор.
9. Тестовый транслятор.
10. Обратный транслятор.

Диалоговый транслятор — это программа, обеспечивающая использование языка программирования в режиме разделения времени.

Синтаксически-ориентированный, или, иными словами, синтаксически-управляемый транслятор, получающий на вход описание синтаксиса и семантики языка, текст на описанном языке и выполняющий трансляцию в соответствии с заданным описанием.

Однопроходной транслятор — это инструмент, создающий объектный модуль при однократном последовательном чтении исходного кода за один проход.

Многопроходной транслятор делает то же самое, но не за один проход.

Оптимизирующий транслятор выполняет оптимизацию создаваемого кода перед записью в объектный файл.

Тестовый транслятор получает исходный код и выдает на выходе измененный исходный код. Запускается перед основным транслятором для добавления в исходный код отладочных процедур. Например, транслятор с языка ассемблера может выполнять замену макрокоманд на код.

Обратный транслятор выполняет преобразование машинного кода в текст на каком-либо языке программирования.

Фактическая цель транслятора, это преобразование исходного кода с одного языка на другой язык. Зачастую этот процесс выполняется посредством нескольких этапов преобразования, или же транслятор способен выдавать несколько результатов (то есть преобразовывать один код к нескольким представлениям).

В таком случае транслятор уместно называть мульти-транслятором.

Понятия трансляция и интерпретация часто путают. Во время трансляции выполняется преобразование кода программы с одного языка на другой, во время интерпретации программа исполняется.

То есть в чистом виде, транслятор переводит с одного языка на другой, но не выполняет код, а интерпретатор построчно преобразовывает код в исполняемый язык и выполняет последовательно программы.

Так как целью трансляции является, обычно, подготовка к интерпретации, эти процессы рассматриваются вместе. Например, языки программирования часто характеризуются как компилируемые или интерпретируемые. Это зависит

от того, какие функции преобладают при использовании языка: компиляция или интерпретация.

Практически все языки низкого уровня и третьего поколения, вроде ассемблера или Си, являются компилируемыми, а более высокоуровневые языки, вроде Python — интерпретируемыми.

С другой стороны, наблюдается интеграция процессов трансляции и интерпретации. Интерпретаторы могут быть компилирующими, а в трансляторах может требоваться интерпретация для реализации метапрограммирования. Более того, один и тот же язык программирования может и транслироваться, и интерпретироваться. Однако в обоих случаях должны присутствовать общие этапы анализа и распознавания конструкций и директив исходного языка. Это относится и к программным реализациям, и к аппаратным.

Рассмотрим далее современные технологии виртуализации, которые позволяют реализовывать принцип кроссплатформенной разработки программного обеспечения.

Рассмотрим далее инструменты нативной разработки.

Здесь под нативной разработкой будем понимать создание программное обеспечение при помощи определенных средств и инструментов, позволяющих запускать приложения на платформе разработки и получить от этого максимальный эффект. Фактически, это разработка под конкретные операционные системы и аппаратные архитектуры с оптимальным использованием их возможностей и максимальным использованием рекомендованных для платформы инструментов.

Нативная мобильная разработка, например, подразумевает создание приложений для конкретных мобильных версий операционных систем. Такие приложения в основном распространяются через специализированные магазины (например, App Store или Google Play). При их создании используются инструменты, фактически, от владельцев этих магазинов и платформ.

Создавая продукт для iOS, разработчики работают с языками программирования Objective-C или Swift, а разработка под систему Android требует знания языков Java или Kotlin. Такие крупные корпорации, как Apple и Google, предоставляют разработчикам собственные наборы средств разработки, отладки, профилирования и тестирования. Все больше и больше компаний предпочитают инвестировать в этот способ создания мобильных приложений из-за множества его преимуществ, хотя и осознают, что этот способ дороже, чем использование кроссплатформенных средств, ведь приходится для каждой новой платформы, фактически, заново писать приложение на другом языке программирования с другими методами вызова аппаратных устройств, датчиков, а также с другой обработкой событий.

Какими же все-таки плюсами обладает нативная мобильная разработка:

1. У нативных приложений лучше производительность.
2. Нативные приложения более безопасны.
3. Нативные приложения более интерактивны и интуитивно понятны.
4. Нативные приложения позволяют разработчикам получить доступ к полному набору функций устройств.

5. При нативной разработке происходит меньше ошибок.

6. Нативные приложения обычно не содержат “прослоек” программного обеспечения.

По поводу производительности можно сказать, что приложение проектируется, разрабатывается и, самое главное, оптимизируется для конкретной платформы и операционной системы. В результате этого продукт показывает очень высокий уровень эффективности и производительности на серии устройств с целевой операционной системой.

Нативные приложения собираются, компилируются и запускаются с использованием основного языка программирования и библиотеки интерфейса платформы. Приложение хранится на самом устройстве (это означает, что его содержимое и визуальные элементы находятся в локальной памяти), благодаря чему загрузка происходит очень быстро, а пользователям не приходится терять своё время на удаленную загрузку ресурсов. Кроме того, приложение с нативным интерфейсом так же не содержит абстракций и дополнительных вызовов к графической библиотеке устройства, а интерфейс выглядит более естественно для модели устройства.

В вопросах безопасности кроссплатформенному ПО, построенному на базе веб-технологий, например, приходится полагаться на браузеры и технологии типа JavaScript, HTML5 и CSS. С нативными мобильными приложениями пользователи получают гораздо более надежную защиту данных, тем более что алгоритмическая, функциональная и даже аппаратная поддержка алгоритмов защиты данных уже встроена в платформу и в сам процессов устройства.

Нативные такие приложения работают более естественным образом, без временных «разрывов», подтормаживаний, странного для платформы оформления и стилей. Они сделаны по определенным стандартам и рекомендациям пользовательского интерфейса для каждой платформы “от производителей”. Это облегчает пользователю понимание принципов работы продукта по аналогии с ранее уже использованными программами с этой же платформы.

Нативные приложения в полной мере используют возможности программного обеспечения и операционной системы, а также не создают дополнительных возможностей для возникновения ошибок. Это касается и принципов вызовов функций устройств и работы с форматами данных. Приложения, фактически, имеют прямой доступ к камере, микрофону, акселерометру и другим функциям устройства без «посредников».

При разработке нативных приложений не приходится полагаться на кроссплатформенные инструменты, благодаря чему количество ошибок, также как и требования к системе, значительно снижаются.

Гибридные приложения получают доступ к аппаратному обеспечению через своеобразный мост, что часто уменьшает скорость работы программы в разы. Такой мост называется промежуточным программным обеспечением и позволяет осуществлять вызовы и передачу данных между приложениями, разработанными при помощи различных технологий.

Однако, кроме удобства разработки, здесь кроется одна огромная проблема, — разработка ведется различными коллективами и компаниями, и они не всегда договариваются друг с другом и проводят совместное тестирование. Нередко возникает ситуация, когда каждый коллектив выполнил разработку «со своей стороны», а этот «мост» не сошелся на середине.

Разработчики нативных приложений имеют доступ к наборам средств разработки с новейшими функциями, благодаря чему в таких приложениях новые возможности появляются сразу же после обновления операционной системы. Это дает им значительное преимущество перед гибридными приложениями, которые могут неправильно работать сразу после очередного обновления ОС, которое затронуло используемые в приложении функции.

Та же самая ситуация наблюдается не только на мобильных системах, но и на рынке приложений для персональных ЭВМ. Меньше всего “дружат” ОС Windows и Linux/MacOS. И, хотя есть возможность разработки сразу для двух платформ на одном и том же языке программирования, приходится учитывать особенности обоих платформ. Например, под Windows есть библиотека WinForms и WPF, но для Linux/MacOS WPF нет, а есть GTK# и проект Mono.

Сейчас много различных технологий разработки нативных приложений, рассмотрим их последовательно на примере инфраструктуры разработки от компании Microsoft, основу которой составляют:

1. язык программирования Visual Basic,
2. язык программирования C#,
3. язык программирования C++,
4. язык программирования iPython,
5. язык программирования JavaScript,
6. язык программирования F# и так далее.

Все они работают не на «голой» системе, а на платформе исполнения типа .NET.

.NET Framework — программная платформа, выпущенная компанией Microsoft в 2002 году. Основой платформы является общеязыковая среда исполнения CLR - Common Language Runtime, которая подходит для разных языков программирования. Функциональные возможности CLR доступны в любых языках программирования, использующих эту среду.

Основным языком платформы сейчас является C#.

Читается и произносится это название как «Си шарп». Создан он был в компании Microsoft в 1998–2001 гг. как некоторая “замена” языку Java. Впоследствии был стандартизирован как отдельная технология?

Сегодня этот язык представляет целое направление, включающее в себя разработку прикладных программ для ОС Windows, ОС Android/iOS. Также используется для компьютерного зрения, программирования встраиваемых систем и написания игр в Unity.

Основной особенностью языка программирования является использование «среды исполнения» для запуска программ, «компиляция на лету», управление кодом и сборкой мусора. Все в C# считается «объектом» (даже простое целое

число), а значит поддерживает концепцию ООП и имеет несколько стандартных методов, например variableName.toString().

С точки зрения графического интерфейса позволяет рисовать сложные интерфейсы при помощи библиотек Windows Forms, WPF, GTK#, а также инструментов Xamarin.

Новичку достаточно просто освоить этот язык благодаря большому числу учебных пособий, книг, видеоуроков и форумов.

Его синтаксис напоминает C++, так что переход с этого языка на C# будет простым.

C# постоянно развивается. Например: с 2 версии поддерживаются обобщенные типы (дженерики), итераторы и ленивые вычисления, анонимные методы; с 3 версии — интегрирован специальный язык запросов LINQ, лямбда-выражения, деревья выражений, неявная типизация (с ключевым словом var — привет JavaScript); с 4 версии — динамическое связывание, опциональные аргументы, библиотека параллельных задач и вычислений, встроенный кэш (MemoryCach); с 5 версии — новые шаблоны, асинхронные выражения (async/await); в 6 версии — компилятор является сервисом, фильтры исключений, автосвойства, null-условные операции, сжатие функций до выражений; в 7 версии — новые шаблоны вычислений, кортежи, локальные переменные, возврат значений по ссылке и многое другое.

Как видим, язык динамично вписывается в концепцию современных языков программирования, впитывая лучшие подходы и технологии из конкурирующих инструментов.

Отдельно стоит упомянуть технологию Xamarin. Изначально это компания, которая разрабатывает и поддерживает технологию Mono для разработки приложений под платформы Windows, iOS, Mac, Android. Microsoft купила Mono в 2016 году и с тех пор Mono Develop стала Xamarin Studio и теперь входит в комплект инструментов Microsoft Visual Studio. Соответственно, можно писать код один раз, а компилировать его на разные платформы. Это очень популярный сегодня подход, позволяющий сократить накладные расходы на разработку программного обеспечения и “содержать” одного программиста “вместо” трех.

Кроме того, разработчиков серверных решений Microsoft постаралась обеспечить полным набором инструментов. Сейчас это два основных течения: ASP .NET и .NET Core.

Первое решение достаточно старое и, можно сказать, “устоявшееся”, “классическое”. Второе — новая волна. Инструменты .NET Core — это модульная платформа для разработки программного обеспечения под различные платформы. Так же основана на .NET Framework, но отличается от нее легкостью, модульной структурой, кроссплатформенностью, ориентацией на облачные технологии. Все модули управляются пакетным менеджером NuGet (в отличе от .NET Framework, который всегда ставится и обновляется целиком).

Согласно свежему рейтингу языков программирования, C# находится на шестом месте. Он немного потерял в популярности, спустившись с пятого места самых распространенных языков, но по-прежнему его используют в качестве основного языка многие программисты.

Также можно рассмотреть альтернативу для конкурирующей платформы Mac OS X от компании Apple на примере языка Swift (так как Java для ОС Android более уместно будет рассмотреть в следующем разделе, посвященном кроссплатформенной разработке благодаря ее широкому охвату операционных систем и платформ).

Сегодня мы рассмотрим один из самых современных и модных языков программирования от компании Apple. Он называется Swift и является компилируемым языком программирования общего назначения. Он создан в первую очередь для разработки программ для iOS и Mac OS. Авторы задумывали язык совместимым со всей коровой базой Objective-C, который до сих пор остается основным языком для разработки продуктов Apple. Программы на Swift компилируются при помощи LLVM (Low Level Virtual Machine). LLVM — это универсальное решение для обработки и оптимизации программ, может использоваться как преобразователь байткода в машинный код либо как интерпретатор + JIT компилятор. Программы для LLVM в его терминологии называются фронтендом, так что Swift — это фактически интерфейс между программистом и LLVM. Собственно говоря, благодаря этой машине, Swift так легко интегрируется с Objective-C даже в рамках одной и той же программы.

Сейчас многие программисты переходят с Objective-C на Swift благодаря его достоинствам и современным инструментам. Он заимствовал многое из Objective-C, однако многое было добавлено. Очень похож на скриптовые языки. Например, после Python на Swift программировать достаточно приятно— они несколько похожи по подходу к разработке программ. Вообще говоря, Swift испытал влияние многих языков (это в основном Objective-C, Ruby, Haskell, Python, C#). Компания Apple утверждает, что он был заложен еще в начале 90-х годов, но текущий вариант языка ведет свою историю с 2010 г. Он стал еще более активно развиваться с 2014 г. и на 2018 г. имел мажорное обновление вместе с iOS 12.

Тесты подтверждают, что программы на этом языке работают быстро (почти в 3–4 раза быстрее Python и даже в 1.5 раза быстрее Objective-C).

Язык имеет свою инфраструктуру — это компилятор, стандартная библиотека, ядро, менеджер пакетов, расширения. Для разработки используется среда XCode от Apple, так что для работы нужен компьютер с MacOS. Но приложения можно писать и для мобильных устройств, и для компьютера.

Swift прост в изучении и полностью соответствует современным принципам языка программирования. В нем есть определенные “фишки”, которых нет во многих других языках даже в Objective-C. Например, в нем нет неопределенных и неинициализированных переменных, объявления переменных можно выполнять по-разному, но всегда будет значение, нет ошибок с размерностями масссивов, ошибок переполнения, можно явно обрабатывать null-значения, он автоматически управляет памятью, чистит за собой и т.п.

Одним словом, язык вместе со своей инфраструктурой разработки берет много забот на себя, так что программист имеет больше времени на реализацию идеи и на разработку бизнес-логики приложения по сравнению с другими языками.

Написание программ на Swift занимает гораздо меньше времени, чем на других языках, так как он немногословен и выразителен. Его скорость работы удивляет, потому что в нем много мощных и функциональных элементов, таких как дженерики, замыкания, кортежи, итераторы, встроенные шаблоны функционального программирования.

Другой особенностью языка является безопасность. Здесь я имею в виду безопасность приложений, так как допустить ошибку в программе гораздо сложнее, чем, например, в C++. Плюс еще сама среда контролирует доступ к данным, осуществляет управление их изменением, а также минимизирует неопределенное поведение. В компиляторе Swift присутствует механизм обнаружения и управления распространенными ошибками и реализована система подсказок. О синтаксисе языка тоже позаботились разработчики, внедрив интеллектуальный помощник в XCode.

Следует так же сказать, что Apple сделала Swift бесплатным и открытым, а для компаний такого масштаба — это редкость. Swift находится в руках сообщества разработчиков, а это значит, что в него вкладываются такие изменения, которые жизненно важны как для профессионалов, так и для начинающих программистов. Можно участвовать в проекте, улучшать его, вводить новые функции…

Согласно опросам, Swift уже входит в десятку самых востребованных языков программирования и вакансий для разработчиков много. Если верить статистике ресурсов-аналитиков, спрос на сотрудников компаний, которые могут программировать на Swift, вырос на 600% только за 2016 год.

ТЕМА 16. КРОССПЛАТФОРМЕННЫЕ СРЕДСТВА, ИНСТРУМЕНТЫ И БИБЛИОТЕКИ АС И ИС

Развитие компьютерных технологий привело к возникновению потребности в совместимости программных продуктов. Раздробленность форматов и существование различных вариантов операционных систем привели к необходимости появления такой категории софта, как кроссплатформенное программное обеспечение. Понятие это появилось давно, а со временем выработался и ряд критериев, которым оно должно соответствовать.

Кроссплатформенные приложения пишутся сразу для нескольких платформ на одном языке, отличном от нативного. Существует два подхода, которые позволяют работать такому коду на разных устройствах.

Первый подход заключается в том, что на этапе подготовки приложения к публикации он превращается в нативный для определённой платформы с помощью транспилера. Это означает, что, фактически, один кроссплатформенный язык программирования «переводится» на другой.

Второй — в том, что к получившемуся коду добавляется определенная обёртка, которая, работая уже на устройстве, на лету транслирует вызовы из неродного кода к родным функциям системы.

Предполагается, что большая часть такого кода может переноситься между платформами — очевидно, что, например, логика совершения покупок, сохранения товара в корзину, просчета маршрута для такси, написания сообщения в мессенджер не меняется в зависимости о того, Android у клиента или iOS. Нужно лишь доработать UI и UX для платформ, но сейчас, в определенных пределах, даже это можно объединить. Внесение исправления в интерфейс для того, чтобы приложение отвечало духу и букве нужной платформы — вопрос желания, необходимой скорости и качества разработки.

Преимущества кроссплатформенности:

Увеличение количества пользователей

Чем больше платформ выбирает себе приложение, тем больше пользователей смогут установить его на свои устройства. Рынки iOS и Android конкурируют между собой за пользователей, а кроссплатформенность приложения позволяет получить пользователей с обоих рынков сразу.

Маркетинг становится проще

При такой большой аудитории нет необходимости таргетировать маркетинг на узкие сегменты пользователей — можно обращаться ко всем сразу. Проблема выбора средств связи с пользователем становится менее насущной — вы можете выбирать себе любые средства распространения информации для аудитории.

Повышение эффективности ресурсов

Современные средства позволяют разрабатывать одно приложение, которое будет доступно для нескольких платформ сразу. Итого, при тех же трудозатратах вы увеличиваете свою аудиторию и, соответственно, денежные потоки.

Недостатки:

Неродной интерфейс или, как минимум, необходимость работы с интерфейсом каждой платформы отдельно.

У каждой системы свои требования к дизайну элементов и иногда они взаимоисключающи. При разработке это необходимо учитывать.

Проблемы в реализации сложных функций или возможные проблемы работы даже с простыми процедурами в силу ошибок самих фреймворков разработки.

Кроссплатформенная среда лишь транслирует запросы к системным вызовам и интерфейсам в понимаемый ею, системой, формат, и потому на этом этапе возможны как сложности с пониманием, так и возникновение ошибок внутри самого фреймворка;

Скорость работы.

Так как кроссплатформенная среда является «надстройкой» над кодом (не всегда, но в определённых ситуациях), в ней возникают свои задержки и паузы в отработке действий пользователя и выводе на экран результатов. Это было особенно заметно несколько лет назад на смартфонах, более маломощных относительно сегодняшних, однако сейчас, с ростом производительности мобильных устройств, этим уже можно пренебречь.

Эти два метода практически являются зеркальным отражением друг друга — то, что плюсы у нативной разработки, минусы у кроссплатформенной, и наоборот.

Современные подходы к разработке программного обеспечения в этой области можно описать следующим образом:

Единое стилистическое решение. В этом случае программа должна выглядеть одинаково под всеми операционными системами. К положительным сторонам этого подхода относят «жесткое» закрепление элементов управления, а к отрицательным – отличие стиля программы от общего стиля ОС.

Адаптивный интерфейс. Подразумевается, что программа, построенная по такому принципу, должна легко вписаться в интерфейс операционной системы за счет изменения тем оформления.

Предполагается полное или частично автоматическое определение языковых параметров и оптимальных размеров экрана, под которые должно подстроиться программное обеспечение.

Положительные стороны – относительно свободная интеграция под стиль операционной системы и даже возможность изменять графическую схему приложения прямо в процессе работы или после перезагрузки. А единственный существенный недостаток - сложность и, соответственно, высокая стоимость разработки.

Гибридная схема сочетает в себе положительные и отрицательные стороны предыдущих подходов. Относительно легкая интеграция и частичная автоматизация настройки, но при этом различие в стилях оформления и сложности, связанные с «плавающей» компоновкой элементов управления. Даже общее описание подходов дает понять, что кроссплатформенное программное обеспечение — это головная боль для разработчиков софта и неисчерпаемый

источник возмущения для пользователей, которые, не вдаваясь в подробности, просто хотят иметь одинаковые возможности на разных платформах.

Рассмотрим далее такие способы перевода программ в исполняемый код, как компиляция и кросскомпиляция, а также принципы работы интерпретаторов и трансляторов.

Рассмотрим далее инструменты кроссплатформенной разработки.

Для начала проведем анализ одного из самых успешных классических языков разработки программного обеспечения – языка Си и C++.

Язык C++ является одним из старейших инструментов разработки приложений самого различного уровня. Причем, сейчас C++ применяется и для встраиваемых систем, и для персональных ЭВМ и даже для мобильных устройств.

Это один из старейших языков программирования в мире, который не потерял актуальности и до сих пор, он основан на стандарте ANSI C и развивается до сих пор.

Язык был разработан еще в 1969 году сотрудником Bell Labs Деннисом Ричи и первоначально предназначался для операционной системы Unix.

Впоследствии перенесен на все другие платформы и стал основой практически для всех системных программ современности.

Его конструкции очень близки к машинным командам, благодаря чему его называют «низкоуровневым». Это значит, что между микропроцессором, который обрабатывает машинные команды, и самим языком, совсем маленький слой абстракции и он работает быстро. Очень быстро.

До сих пор язык Си используется для ускорения программ на других языках и в качестве эталона для разработки новых языков, поэтому большая часть современных языков программирования в той или иной степени испытала влияние этого языка.

Язык имеет несколько стандартов и реализаций, но со временем язык становился все более декларативным, библиотеки с встраивались в ядро языка.

Си и C++ по-прежнему является системными инструментами, а значит работают с однопроходным компилятором. Любая программа, прежде чем стать исполняемым файлом, подвергается сборке, связыванию и переводу в бинарное представление, понятное компьютеру. А так как компиляторов достаточно много, то язык доступен на всех современных платформах: от персональной ЭВМ до встраиваемых систем, мобильных устройств и «интернета вещей».

Программа, выполненные в соответствии со стандартом языка, может компилироваться для различных архитектур и типов процессоров. Использоваться там как на системном, так и на прикладном уровне.

На Си можно писать как небольшие программы и утилиты, так и большие программные комплексы. В стандартную библиотеку внесено множество нужных функций, включая математические функции и функции для работы с файловой системой.

Отдельного упоминания стоят графические возможности языка, так как существует целый набор библиотек для построения графического интерфейса. Например: GTK+, WxWidgets, Qt, Juce, Cegui.

Стиль программирования Си - вцелом процедурный и, более того, в базовом Си отсутствует объектно-ориентированное программирование.

Язык со статической типизацией, так что тип переменной определяется заранее и не меняется в дальнейшем. Очень много операций с адресами переменных в памяти (указателями) - много контроля, но большая ответственность для программиста.

На функцию тоже можно сделать указание через адрес. Присутствует область видимости имен переменных. На низком уровне используются специальные типы данных типа структур и объединений, жестко связанных со строением памяти, механизмами доступа к памяти, а также особенностями аппаратуры.

В отличие от многих современных языков, например, отсутствуют вложенные функции, возможность вернуть из функции сразу несколько переменных, инструменты автоматического управления памятью, встроенные средства объектно-ориентированного программирования и инструменты функционального программирования.

В современных стандартах встроенными можно уже найти такие новые инструменты как коллекции, шаблоны классов, вектора, строки и т.п.

Несмотря на это писать на языке Си приятно. Чувствуется мощь и ответственность, масса контроля и множество деталей. Он достаточно минималистичный, поэтому синтаксис языка и основные конструкции можно выучить за час - полтора.

Другое дело, что прежде, чем перейти к сложным программам необходимо понять адресную арифметику и принять тот факт, что перед использованием сложных переменных (например, массивов), необходимо вручную выделить память, а при изменении их размера, необходимо эту область памяти также изменить.

Многие специалисты говорят, что знание Си дает понимание того, как работает практически любая программа и любой компьютер.

Разработка на Си требует больших затрат времени программиста, чем, например, разработка на Java или Python, но программы, написанные на Си, работают быстрее, намного быстрее аналогов. А это бывает чрезвычайно важно. Конечно, для стандартного компьютера это вторично, а вот для встраиваемой системы или микроконтроллера, для которого важна каждая миллисекунда и каждый такт операции, это важно.

Именно поэтому на языке Си пишут расширения для других языков программирования, делают библиотеки, отдельные утилиты. В общем применяют там, где должно работать быстро.

Язык Си так же часто изучают в качестве первого языка, несмотря на некоторую сложность и особенности программирования. Считается, что каждый уважающий себя программист с высокой квалификацией должен уметь писать (или хотя бы свободно читать) софт на Си. И именно на Си, а не на C++ или C#.

При этом дополнительный эффект от знания языка и от имеющихся навыков появляется именно тогда, когда программист знаком с самой платформой, с ее особенностями и сложностями разработки.

В принципе, многообразие современных инструментов и новый стандарт языка переводит C++ в ранг кроссплатформенных, но, так как это язык компилируемый, то через призму какой-то одной конкретной платформы C++ понимается как именно специализированный язык нативной разработки.

Для C++ существует огромное количество средств и инструментов (библиотек). В данном случае мы рассмотрим две наиболее мощных библиотеки, имеющих как внутренние надстройки над языком, так и библиотеки для работы с графическим интерфейсом.

Сначала рассмотрим библиотеку Qt.

Qt представляет собой инструментарий разработки ПО, предназначенный для разработки на C++ кроссплатформенных приложений с графическим пользовательским интерфейсом. Есть также “привязки” ко многим другим языкам программирования: Python — PyQt, PySide; Ruby — QtRuby; Java — Qt Jambi; PHP — PHP-Qt и другие.

Графический интерфейс в Qt более всего похож на Java Swing — здесь тоже существуют схемы стилей, например Windows, CDE, Motif, копирующие известные оболочки. Присутствуют также layout’ы, автоматически размещающие элементы управления; кроме того, есть "спейсеры", которые раздвигают соседние компоненты.

Есть также немало инструментов, напоминающих Delphi: подсказок, размеров, масштабирование компонентов и прочее. Как и в Java Swing, все элементы нарисованы от руки, то есть стандартные механизмы рисования элементов управления не применяются — вместо этого используется, например, GDI WinAPI. Автоматически определяется версия ОС и, соответственно, реализуются или игнорируются те или иные свойства, вроде прозрачности. Однако, Qt использует и некоторые стандартные диалоговые окна Windows, в частности диалоги открытия файла и настройки печати.

Для ускорения и упрощения создания пользовательских интерфейсов Qt предоставляет программу Qt Designer, позволяющую делать это в интерактивном режиме.

Вторым самым популярным фреймворком для программирования на C++ с разработкой графического интерфейса, является GTK.

GTK+ — это фреймворк для создания кроссплатформенного графического интерфейса пользователя (GUI). Наряду с Qt он является одной из двух наиболее популярных на сегодняшний день библиотек для X Window System.

Изначально эта библиотека была частью графического редактора GIMP, но позже стала независимой и приобрела популярность. GTK+ — это свободное ПО, распространяемое на условиях GNU LGPL и позволяющее создавать как свободное, так и проприетарное программное обеспечение.

GTK+ написан на языке Си, однако несмотря на это, является объектно-ориентированным. Также можно использовать обёртки для следующих языков: Ada, C, C++, C#, D, Erlang, Fortran, GOB, Genie, Haskell, FreeBASIC, Free Pascal, Java, JavaScript, Lua, OCaml, Perl, PHP, PureBasic, Python, R, Ruby, Smalltalk, Tcl, Vala.

Внутри GTK+ состоит из двух компонентов: GTK, который содержит набор виджетов (кнопка, метка и т.д.) и GDK, который занят выводом результата на экран.

Внешний вид приложений может меняться программистом и/или пользователем. По умолчанию приложения выглядят нативно, т.е. так же, как и другие приложение в этой системе. Кроме того, начиная с версии 3.0, можно менять внешний вид элементов с помощью CSS.

GTK+ имеет большое количество виджетов и интерфейсов:

Windows (обычное окно или диалоговое окно).

Дисплеи (ярлык, изображение, индикатор выполнения, строка состояния).

Кнопки и переключатели (кнопки проверки, переключатели, переключатели и кнопки связи).

Числовые (горизонтальные или вертикальные шкалы и кнопки вращения) и текстовые данные (с завершением или без завершения).

Многострочный текстовый редактор.

Дерево, список и просмотрщик сетки значков (с настраиваемыми визуализаторами и разделением модели/вида).

Поле со списком (с записью или без нее).

Меню (с изображениями, переключателями и контрольными элементами).

Панели инструментов (с переключателями, кнопками переключения и кнопками МЕНЮ).

GtkBuilder (создает пользовательский интерфейс из XML).

Селекторы (выбор цвета, выбор файла, выбор шрифта).

Макеты (табличный виджет, виджет таблицы, виджет расширения, рамки, разделители и многое другое).

Значок состояния (область уведомлений в Linux, значок в трее в Windows).

Печать виджетов.

Недавно использованные документы (меню, диалог и менеджер) и так далее.

Эта библиотека имеет возможность составлять нативные интерфейсы, но может делать и кроссплатформенный интерфейс. Наряду с Qt он является одной из двух наиболее популярных на сегодняшний день библиотек для X Window System.

Изначально эта библиотека была частью графического редактора GIMP, но позже стала независимой и приобрела популярность. GTK+ — это свободное ПО, распространяемое на условиях GNU LGPL и позволяющее создавать как свободное, так и проприетарное программное обеспечение.

Другой популярной платформой для разработки приложений еще с 90-х гг. является язык Pascal.

Pascal и Object Pascal - это одни из самых старых и состоявшихся языков программирования современности. Они используется для обучения программированию в старших классах и на первых курсах вузов, является основой для ряда других языков. Язык был создан Никлаусом Виртом в 1968—1969 годах после его участия в работе комитета разработки стандарта языка Алгол-68. Язык назван в честь французского математика, физика, литератора и

философа Блеза Паскаля. Первая публикация Вирта о языке датирована 1970 годом; представляя язык, автор в качестве цели его создания указывал построение небольшого и эффективного языка, способствующего хорошему стилю программирования, использующему структурное программирование и структурированные данные.

Сейчас существует множество вариаций языка и множество его последователей:

UCSD Pascal, разработан в 1978 году в университете в Калифорнии. В последствии стал одним из основных диалектов языка и основой для проектирования новых языков.

Object Pascal, предложен в 1986 году фирмой Apple при консультировании с самим Виртом.

Turbo Pascal, начал разрабатываться в 1983 году и добавлен в основное направление развития семейства языков в 1989 году.

Современными версиями языка являются вариации Object Pascal, Free Pascal и GNU Pascal.

Начиная с Delphi 2003, создана реализация языка для платформы .Net, хотя разработчики продолжают использовать Delphi более ранних версий.

О коммерческих разработках на Free Pascal, GNU Pascal и TMT Pascal на данный момент известно мало.

В Южном федеральном университете разработан PascalABC.NET — язык программирования Паскаль, включающий большинство возможностей языка Delphi, а также ряд собственных расширений. Он основан на платформе Microsoft.NET и содержит практически все современные языковые средства: классы, перегрузку операций, интерфейсы, обработку исключений, обобщенные классы и подпрограммы, сборку мусора, лямбда-выражения.

PascalABC.NET является мультипарадигменным языком: на нём можно программировать в структурном, объектно-ориентированном и функциональном стилях.

PascalABC.NET — это также простая и мощная интегрированная среда разработки, поддерживающая технологию IntelliSense, содержащая средства автоформатирования, встроенный отладчик и встроенный дизайнер форм. Кроме того, консольный компилятор PascalABC.NET функционирует на Linux и MacOS под Mono.

Проприетарным вариантом развития языка является проект Delphi. Сейчас это наименование настолько популярно, что сам по себе язык Object Pascal почти всегда называют именно Delphi.

Delphi, это императивный, структурированный, объектно-ориентированный, высокоуровневый язык программирования со строгой статической типизацией переменных. Основная область использования — написание прикладного программного обеспечения.

Изначально среда разработки Delphi была предназначена исключительно для разработки приложений Microsoft Windows, затем был реализован вариант для платформ Linux (под торговой маркой Kylix), однако после выпуска в 2002 году Kylix 3 его разработка была прекращена, и вскоре было объявлено о

поддержке Microsoft .NET, которая, в свою очередь, была прекращена с выходом Delphi 2007.

В настоящее время, наряду с поддержкой разработки 32 и 64-разрядных программ для Windows, реализована возможность создавать приложения для Apple Mac OS X. Также есть версии под iOS, Android и Linux Server.

При создании языка (и здесь качественное отличие от языка C) не ставилась задача обеспечить максимальную производительность исполняемого кода или лаконичность исходного кода для экономии оперативной памяти. Изначально, язык ставил во главу угла стройность и высокую читаемость, поскольку был предназначен для обучения дисциплине программирования. Эта изначальная стройность, в дальнейшем, как по мере роста аппаратных мощностей, так и в результате появления новых парадигм, упростила расширение языка новыми конструкциями.

Так, сложность объектного C++, по сравнению с C, выросла весьма существенно и затруднила его изучение в качестве первого языка программирования, чего нельзя сказать об Object Pascal относительно Pascal.

В Delphi идентификаторы типов, переменных, а равно и ключевые слова читаются независимо от регистра: например, идентификатор SomeVar полностью эквивалентен somevar. Зависимые от регистра идентификаторы в начале компьютерной эпохи ускоряли процесс компиляции, и кроме того, позволяли использовать очень короткие имена, порой отличающиеся лишь регистром.

И хотя к настоящему времени обе эти практики − использование нескольких идентификаторов, различающихся лишь регистром, равно как и чрезмерная их лаконичность, осуждены и не рекомендованы к применению, практически все унаследованные от C языки − C++, Java, C# − являются регистро-зависимыми, что, с одной стороны, требует достаточно большой внимательности к объявлению и использованию идентификаторов, а с другой — принуждает писать более строгий код, когда каждая переменная имеет чётко определённое имя (вариации регистра могут вызвать путаницу и ошибки).

В Delphi в исходных файла pas (которые, как правило, и содержат основное тело программы) на уровне языковых средств введено строгое разделение на интерфейсный раздел и раздел реализации. В интерфейсной части содержатся лишь объявления типов и методов, тогда как код реализации в интерфейсной части не допускается на уровне компиляции. Подобное разделение свойственно также языкам C/C++, где в рамках культуры и парадигмы программирования вводится разделение на заголовочные и собственно файлы реализации, но подобное разделение не обеспечивается на уровне языка или компилятора.

В C# и Java такое разделение утрачено вовсе − реализация метода, как правило, следует сразу же после его объявления. Инкапсуляция обеспечивается лишь принадлежностью метода к той или иной области видимости. Для просмотра одной только интерфейсной части модуля исходного кода используются специальные средства.

В Delphi объектное и объектно-ориентированное программирование хоть и поощряется, однако не является единственно возможным и рекомендованным.

Так, допустимо, опять же в отличие от C#, объявление и использование глобальных или статических функций и переменных.

Язык C# принудительно работает с объектами. Глобальные функции без привязки к классу. Value-типы, наподобие структур struct, унаследованы от общего типа C# несмотря на то, что сами по себе они не могут быть унаследованы. То есть, наследование структур в C# запрещено. Вместе с тем, экземпляры классов и Delphi и C# являются неявно-ссылочными типами.

Поскольку системные вызовы в Windows, как, впрочем, и в POSIX-системах наподобие Linux, Mac OS, формально необъектны, взаимодействие кода во многих языках программирования с ними затруднено даже без учёта разной парадигмы управления временем жизни переменных в памяти. А Delphi не имеет подобных ограничений.

Для наиболее гибкой и эффективной реализации объектно-ориентированного подхода в Delphi, введены два механизма полиморфного вызова:

Классический виртуальный.

Динамический.

Если в случае классического виртуального вызова, адреса всех виртуальных функций будут содержаться в таблице виртуальных методов каждого класса, то в случае с динамическим вызовом указатель на метод существует лишь в таблице того класса, в котором он был задан или перекрыт.

Подобная оптимизация имеет своей целью уменьшение размера статической памяти, занимаемой под таблицы методов. Экономия может быть существенна для длинных иерархий классов, с очень большим количеством виртуальных методов. В C-подобных языках динамические полиморфные вызовы не применяются.

Эволюция современного кроссплатформенного Delphi насчитывает следующие этапы:

Borland Delphi.

CodeGear Delphi.

Embarcadero Delphi.

Текущий владелец технологии, компания Embarcadero много вложила в развитие и популяризацию языка.

На сегодняшний день выпущено множество версий языка и множество версий среды разработки, называемой RAD Studio.

Сейчас существует даже свободная для некоммерческого использования версия среды разработки Community Edition.

Embarcadero Delphi Community Edition — это отличный способ начать создание высокопроизводительных приложений для Windows, mac OS, iOS и Android на Delphi. Delphi Community Edition включает в себя упрощенную версию IDE, редактор кода, интегрированный отладчик, двусторонние визуальные дизайнеры для ускорения разработки, сотни визуальных компонентов и ограниченную лицензию для коммерческого использования.

Для студента или преподавателя ВУЗа, а также для обычного разработчика проектов с открытым исходным кодом, важно знать об этой версии следующее:

Delphi Community Edition предоставляет возможность использования встроенных профессиональных инструментов разработки с самого первого дня.

Разработка приложений для Windows, macOS, Android и iOS осуществляется с использованием единой базы кода.

Визуальная разработка с использованием программных каркасов Delphi VCL и FireMonkey.

Встроенные инструменты позволяют осуществлять отладку на любом устройстве.

Создание приложений для баз данных с локальным и встроенным подключением.

Сотни встроенных компонентов позволяют повысить уровень разрабатываемых приложений и сократить количество циклов разработки.

Лицензия на использование продолжает действовать до тех пор, пока прибыль физического лица или компании от приложений Delphi не достигнет 5 000 долларов США, или штат команды разработчиков не превысит 5 человек.

Профессиональная версия продукта стоит очень дорого для обычного разработчика, однако Community Edition вполне применима для индивидуальных проектов или для обучения.

Также существует альтернатива коммерческой RAD Studio в лице проекта с открытым исходным кодом Lazarus.

Lazarus — это открытая среда разработки программного обеспечения на языке Object Pascal для компилятора Free Pascal (часто используется сокращение FPC — Free Pascal Compiler, бесплатно распространяемый компилятор языка программирования Pascal). Интегрированная среда разработки предоставляет возможность кроссплатформенной разработки приложений в Delphi-подобном окружении.

Позволяет достаточно несложно переносить Delphi-программы с графическим интерфейсом в различные операционные системы: Linux, FreeBSD, Mac OS X, Microsoft Windows, Android.

Основные возможности системы, следующие:

Поддерживает преобразование проектов Delphi

Реализован основной набор элементов управления

Редактор форм и инспектор объектов максимально приближены к Delphi

Интерфейс отладки (используется внешний отладчик GDB)

Простой переход для Delphi программистов благодаря близости LCL к VCL

Полностью юникодный (UTF-8) интерфейс и редактор и поэтому отсутствие проблем с портированием кода, содержащего национальные символы

Мощный редактор, включающий систему подсказок, гипертекстовую навигацию по исходным текстам, автозавершение и рефакторинг

Форматирование исходного текста «из коробки», используя механизмы Jedi Code Format

Поддержка двух стилей ассемблера: Intel и AT&T (поддерживаются со стороны компилятора)

Поддержка множества типов синтаксиса Pascal: Object Pascal, Turbo Pascal, Mac Pascal, Delphi (поддерживаются со стороны компилятора)

Имеет собственный формат управления пакетами

Автосборка самого себя (под новую библиотеку виджетов) нажатием одной кнопки

Поддерживаемые для компиляции ОС: Linux, Microsoft Windows (Win32, Win64), Mac OS X, FreeBSD, WinCE, OS/2

На Lazarus написано много современных приложений, используемых широкими массами пользователями. Среди них выделяются:

Total Commander - файловый менеджер.

Double Commander - файловый менеджер.

GLScene - графический 3D движок.

PeaZip - архиватор.

Ubuntu Control Center - центр управления ОС Ubuntu.

Transmission Remote GUI - Подсистема управления торрентов.

DataExpress - система управления бухгалтерским учетом.

В общем, Lazarus является идеальным инструментом разработки на языке Pascal со свободной лицензией для распространения.

И в завершении лекции мы рассмотрим экосистему Java.

Java — это, пожалуй, один из самых востребованных языков программирования. Его история насчитывает уже почти 25 лет, он появился в 1995 г. в компании Sun Microsystems. В настоящее время принадлежит компании OpenSource и имеет свободную модель распространения по лицензии GPL. В развитие Java вносят свой вклад такие мировые гиганты как Google, RedHat, IBM, JetBrains. Последняя, кстати, является Российской компанией и известность им принесли достижения в сфере инструментов рефакторинга и программирования на языке Java.

Java — это один из самых популярных языков программирования. На нем пишут кроссплатформенные приложения, серверное программное обеспечение, коммерческое ПО, небольшие утилиты. На Java написаны крупные пакеты прикладного программного обеспечения, среды и инструменты разработчика, игровые сервера типа Minecraft, Lineage.

На Java разрабатывались игры еще для кнопочных телефонов типа тетриса и змейки, гонок и ролевых игр. Это было мобильное направление разработки, Java ME.

И, конечно, 90% игр и программ под ОС Android написано на Java. Это нативный, то есть естественный язык приложений для самой популярной операционной системы для смартфонов. Кроме того, многие современные технологии кроссплатформенного программирования типа React Native, в той или иной мере используют Java.

Сам по себе язык Java красивый, выразительный, но, по моему мнению, многословный.

Он транслируется в промежуточный байт-код специального вода, поэтому может работать абсолютно на любой архитектуре процессора, которая поддерживает виртуальную машину Java — JVM. Именно виртуальная машина пробрасывает системные вызовы на аппаратную и программную платформы компьютера и готовит байт-код для запуска уже на процессоре.

Такой подход обеспечивает широкое применение языка. Эта тенденция наблюдается вплоть до использования Java для программирования микропроцессорных пластиковых карт (подмножество языка Java Card). Этот факт мало известен, но часть SIM-карт и банковских карт работает именно благодаря Java-апплетам.

У Java высокий уровень безопасности, так как исполнение программы контролируется виртуальной машиной и любые операции, превышающие установленный для машины уровень привилегий, запрещены и вызывают немедленное прерывание процесса выполнения.

Многие называют недостатками Java низкую производительность и высокие затраты памяти. Однако программа Java работает со скоростью работы самой машины, а память сейчас дешева. Зато покрываются практически все существующие сейчас платформы.

Язык Java является строго типизированным, в нем существуют аннотации и метаданные в программе, средства обобщенного программирования, инструментарий шаблонов, методы с неопределенным количеством параметров, поэлементные циклы для коллекций, внедрены сложные типы данных вроде коллекций, система кэшей разного уровня. Впоследствии добавлена возможность подключения расширений на базе динамически-типизированных языков, модулей расширения JVM, инструменты проверки class-файлов, инновационные механизм исключений, способы закрытия ресурсов, методы обновления коллекций и связывания.

Так же Java может реализовывать несколько типов интерфейсов пользователя: AWT — был первым фреймворком, использует нативные элементы управления, существующие именно в той операционной системе, в которой программа запущена, Swing — пришел на замену AWT и является его надстройкой, то есть для своей собственной отрисовки применяет средства AWT, SWT — как и AWT использует нативные элементы управления, создан IBM, является основой такой популярной программы, как eclipse, однако не входит в стандартную поставку вместе с языком или платформой исполнения, JavaFX — спонсируется и разрабатывается Oracle в качестве замены Swing — в отличие от которого применяет аппаратное ускорение и создание разметки при помощи CSS и HTML.

Уже много лет Java держится в десятке самых популярных и востребованных языков программирования. По рейтингу Stack Overflow, запросы, связанные с языком Java задают более 45% пользователей портала. По данным Github, Java стабильно занимает второе или третье место по популярности.

Классическими инструментами разработки проектов на Java являются:

Eclipse IDE.

NetBeans IDE.

IntelliJ IDEA.

JDeveloper.

Android Studio.

Рассмотрим принцип работы виртуальной машины Java.

JVM (Java Virtual Machine) это среда для запуска Java приложений. При запуске Java программы вызывается метод main, который реализован в java коде. JVM это часть среды исполнения JRE (Java Runtime Environment).

JRE это именно среда для запуска Java приложений. Это не JDK. JDK (Java Development Kit — это набор инструментов, направленный на разработку приложения на Java. JDK включает в себя как JRE так компилятор Java кода и библиотеки для разработки Java приложений).

Java приложения называются WORA (Write Once Run Everywhere) приложениями, то есть буквально напиши один раз и запускай везде. Это означает что разработчик может создать Java код на одной операционной системе и запустить его в другой системе, в которой установлена JRE без каких-либо дополнительных настроек. Это все возможно, потому что это JVM.

Когда компилируется .java файл, создается .class файл (java byte code) с таким же названием. Этот файл автоматически генерируется Java компилятором при успешной сборке проекта.

В основном Java Class отвечает за три следующие активности:

Загрузка.

Ссылки.

Инициализация.

Загрузчик классов (Class loader) считывает .class файл, генерирует соответствующие бинарные данные, а также сохраняет полученные данные в области методов для использования.

Для каждого .class файла, JVM хранит следующую информацию в области методов:

Полное имя загружаемого класса и его непосредственные родительские классы.

Определяет является ли .class файл Java классом, интерфейсом или Enum.

Определяет поля, переменные и информацию о методах и т.д.

После загрузки .class файла, JVM создает объекты класса в области Heap памяти. Необходимо помнить, что этот объект является типом Class, который определен в пакете java.lang. Объект этого класса программист может использовать для получения информация уровня класса, например, имя класса, имя родительского класса, методы класса, переменные и так далее. Чтобы получить все эти данные необходимо вызывать метод getClass() у этого объекта. Причем для каждого загруженного класса (.class), создается только один объект.

Ссылки обеспечивают проверку, подготовку и (опционально) разрешение работы с .class файлом.

Проверка: гарантирует корректность .class файлов. Проверяет что конкретный .class был скомпилирован и сгенерирован валидным компилятором. Если проверка не пройдена, то возникает run-time exception java.lang.VerifyError. Подготовка: JVM выделяет память для переменных класса и инициализирует эти переменные в памяти значениями по умолчанию.

Решение: Этот процесс заменяет символические ссылки на класс прямыми. При поиске в области методов обнаруживает зависимые сущности класса. Инициализация: В этой фазе, все статические переменные инициализируются

заданными значениями в коде, так же выполняются статические блоки, если такие есть. Выполнение программы происходит сверху вниз, в классе и так же от родителя к наследнику в иерархии классов.

Существует три основные загрузчика классов (cloass loaders):

Bootstrap class loader: Каждая реализация JVM должна иметь bootstrap class loader, способная загружать проверенные классы. Загружается основное API Java классов, которое находится в директории JAVA_HOME/jre/lib. Этот путь так же называют bootstrap path. Загрузчик реализован на языках C, C++.

Extension class loader: Это дочерний загрзчик bootstrap class loader. Он загружает классы, которые представлены в дирекотрии JAVA_HOME/jre/lib/ext(Extension path) или в любой другой директории, которая описана в системной переменной java.ext.dirs. Это функция реализована с помощью Java sun.misc.Launcher$ExtClassLoader class

System/Application class loader: В свою очередь это дочерний загрузчик extension class loader. Загружает классы из области приложения (application classpath). Так же реализован с помощью Java sun.misc.Launcher$ExtClassLoader class.

JVM соблюдает принцип иерархического делегирования при загрузке классов. Система загрузчика классов делегирует запрос в extension class loader, a extension class loader делегирует запрос в boot-strap classloader.

Если класс найден в boot-strap пути, то класс загружается, в противном случае запрос отправляется обратно в extension class loader, а затем в system class loader. И если system class loader не смог загрузить класс, то возникает run-time exception java.lang.ClassNotFoundException.

Особое внимание следует уделить памяти виртуальной машины Java.

Память содержит следующие области:

Область хранения методов. В области методов находится все информация о классе, например, имя класса, имена родительских классов, методы и переменные, а также прочая информация, включая статические переменные. Существует всего одна область методов во всей JVM и это область общий ресурс для всех программ, работающих в этой JVM.

Область хранения информации об объектах (Heap area). Существует всего одна область Heap в JVM и доступная всем программам, которые работают в этой JVM.

Область стека (Stack area). Для каждого потока JVM создает свою специальную область (run-time stack). Каждый блок этого стека называется активная запись/окно стека (activation record/stack frame). В этой области хранятся вызовы методов и все локальные переменные этих методов. После того как поток завершает свою работу, выделенный стек уничтожается, так как это не общедоступный ресурс.

Область хранения адресов текущих инструкций потока (PC Registers). У каждого потока она своя.

Область нативных методов (Native method stacks). Для каждого потока создается отдельная область Native method stacks. В этой области хранится информация о нативных методах.

Также важным элементом является движок выполнения (Execution Engine).

Execution engine выполняет байткод .class файла. Он считывает байткод последовательно, строчка за строчкой, использует информацию, которая находится в разных областях памяти JVM и выполняет инструкции кода. Можно разделить на три составляющих:

Интерпретатор (Interpreter): интерпретирует байт код в команды и выполняет их. Недостаток в том что каждый раз когда нужно выполнить команду, необходимо интерпретировать байт код в понятную для JVM команду, даже если вызывается один и тот же метод несколько раз.

Just-In-Time Compiler(JIT): Используется для увеличения эффективности интерпретатора. Он компилирует весь байткод и преобразует его в нативный код, всякий раз когда встречает повторяющиеся вызовы методов. JIT поставляет нативный код и повторная интерпретация не нужна, поэтому увеличивается эффективность.

Сборщик мусора (Garbage Collector): уничтожает неиспользуемые объекты в памяти.

Другим составным элементом виртуальной машины является Java Native Interface (JNI).

Это интерфейс для взаимодействия с нативными методами системных библиотек, которые могут быть написаны на C, C++. Для вызова методов из системных библиотек (например, .dll или .so) необходимо указать, где именно это библиотека находится, чтобы JVM знала об этой библиотеке.

Ну и наконец Native Method Libraries.

Это набор нативных библиотек (C, C++), которые необходимы для работы Execution Engine.

Далее целесообразно рассмотреть относительно новый продукт программных технологий от компании JetBrains. Это фактически наше отечественное решение, которое оценено очень высоко мировой общественностью.

Язык Kotlin является статически типизированным языком программирования, как и Java, работает под управлением и поверх виртуальной Java-машины JVM и управляет рядом платформ через инфраструктуру LLVM. Кроме всего прочего, Kotlin компилируется в JavaScript.

При создании языка учитывалось все богатство функций Java, все его положительные черты и все недостатки. Kotlin хорошо поддерживается платформой исполнения и инструментами разработки. Он полностью совместим с Java и даже поддерживает прямое преобразование кода с ним.

Kotlin подходит для разработки сложных приложений как для персонального компьютера, так и для платформы Android. Возможности интеграции с Java позволяют встраивать Kotlin-код непосредственно в уже существующие Java-проекты без каких-либо проблем. То есть если Вы уже знаете Java, переход на Kotlin будет занимать максимум час времени на освоение синтаксиса языка и его спецификации.

Язык относительно молодой, разработка ведется с 2010 года. Официальный релиз произведен летом 2011 г. Язык имеет открытую реализацию, то есть является проектом с открытым исходным кодом.

В мае 2017 г. компания Google сообщила, что инструменты Kotlin будут включаться в качестве стандарта в Android Studio, то есть этот язык стал официальным языком Google/Android, а в 2019 году было объявлено, что Kotlin стал приоритетным языком для разработки под ОС Android.

Синтаксис языка использует лучшее от Pascal, TypeScript, SQL, F#, Go, Scala, Java, C#, Rust, D и C++. Kotlin поддерживает как декларативный и объектно-ориентированный, так и функциональный подход к программированию.

В Kotlin убраны некоторые опасные особенности Java. В первую очередь все вспоминают о Null Safety, то есть об определении возможности хранения в переменной ссылки на null прямо во время объявления и возможности безопасного вызова. Также Kotlin отлично поддерживается анализатором исходного кода в средах разработки, можно определить метод для некоторого типа отдельно от объявления самого типа, есть возможность использовать лямбда-выражения (как сущности первого класса и претензию на функциональность), есть возможность размещения функции прямо по месту ее использования, есть возможность создавать делегаты, дженерики и так далее.

В общем Kotlin получился достаточно удобным и в принципе годится для первого языка программирования. Однако некоторые аспекты программирования, тонкости и понятия «тянутся» из Java, так что их придется осваивать уже по ходу работы, что не всегда удобно.

ТЕМА 17. СЕТЕВЫЕ ИНСТРУМЕНТЫ И ОБЛАЧНЫЕ ТЕХНОЛОГИИ ИС И АС

В данной лекции мы проведем анализ применения компьютерных сетей при построении информационных и автоматизированных систем, изучим основные понятия сетей передачи данных, рассмотрим топологии сетей передачи данных, методы коммутации и доступа к моноканалам, проведем исследование основных сетевых протоколов АС и ИС, рассмотрим протоколы маршрутизации и, отдельно, рассмотрим протоколы верхнего уровня, часто используемые на практике в клиент-серверных архитектурах АС и ИС.

Как известно, компьютеры и другое оборудование, имеющее «выход в сеть», могут иметь различную архитектуру, протоколы обмена и программное обеспечение. Их совместимость, по крайней мере для передачи данных, достигается исключительно за счет использования стандартизированных коммуникационных протоколов. А все протоколы собраны в наборы формальных норм и правил, касающихся использования физических и логических уровней передачи информации по сетям. Такие стандарты, также регламентируют использование аппаратного и программного обеспечения.

Система таких стандартов в рамках одного или нескольких уровней сетевой модели OSI/ISO образуют так называемый **Стек протоколов.**

Далее мы рассмотрим один из самых распространенных сетевых стеков протоколов, который носит название transmission control protocol/internet protocol или TCP/ IP. Этот стандарт уже давно стал одним из основных в индустрии сетевых технологий. На этом стеке к настоящему времени разработаны сотни тысяч надежных сетевых приложений, поддерживающих связь между различными устройствами в глобальных сетях. Технологии TCP/IP настолько прочно вошли в современную жизнь, что их поддерживают абсолютно все современные операционные системы, библиотеки приложений, фреймворки и платформы. TCP/IP фактически служит основой для создания современной сети Интернет.

Протокол TCP может быть охарактеризован как простой протокол управления надежной передачей данных. Составляющие его компоненты эффективно фрагментируют (разбивают) информацию на порции или пакеты определенного размера, нумеруют и отправляют по установленным ранее соединениям. Программное обеспечение при выходе из локальной сети добавляет к каждому пакету служебные данные, регламентированные другой частью протокола – IP. IP – это протокол именно уровня Интернета, а не локальной сети. Служебные данные содержат поля с адресами отправителя и получателя. Наличие этих адресов обеспечивает доставку всех пакетов в глобальной сети благодаря системе маршрутизации.

Особенностью стека TCP/IP также является то, что в нем фактически отсутствует состояние «абонент занят» или «канал связи не доступен». Каждый компьютер, имеющий подключение TCP/IP, может принимать одновременно пакеты от большого числа источников.

В таком случае нет необходимости формировать или находить отдельные каналы связи между всеми абонентами для передачи адресных сообщений. Все сообщения следуют по одному сетевому каналу, разделяя его между собой, хотя, логически, существует множество соединений или «связей по данным» между устройствами и службами.

Примечательно также то, что протоколы и службы Интернета можно применять и в рамках локальной сети, т.е. стек имеет «обратную совместимость» с менее масштабными сетями. В таком случае сеть имеет название Интранет.

Если провести анализ сетевой модели OSI/ISO, то стек протоколов TCP/IP объединяет технологии следующих типов.

- Telnet. Это протокол эмуляции удаленного или локального терминала. Протокол позволяет получить и удерживать доступ к удаленному компьютеру, вызывать на нем программы на выполнение, скачивать или загружать данные, выполнять отдельные команды.
- FTP. Это простой протокол передачи файлов через сети различного масштаба. FTP или File Transmission Protocol – один из самых известных протоколов и в настоящее время применяется практически всеми хостинговыми провайдерами для размещения файловых серверов или серверов приложений, так как кроме поддержки пересылки файлов он предоставляет возможность интерактивной работы с ресурсами, аутентификации пользователей и так далее.
- SMTP. Это протокол передачи почтовых сообщений. Simple Mail Transfer Protocol поддерживается практически всеми серверами, даже теми, кто уже перешел на новый протокол IMAP, и, судя по всему, будет поддерживаться еще достаточно долго.
- DNS. Это протокол сопоставления сетевых адресов и вербальных имен сервисов. Domain Name System реализуется системой специальных серверов и дает пользователям возможность использовать «говорящие» имена ресурсов вместо IP-адреса.
- RIP. Это протокол межсетевого взаимодействия. Routing Internet Protocol предназначен для сбора маршрутной информации в сетях различного уровня, оснащенных специальным оборудованием.
- SNPM. Это простой протокол управления сетью. Так называемый Simple Network Management протокол дает возможность использовать функции управления узлами сети из некоторого единого центра. К таким узлам мы можем отнести сервер, станции, аппаратные или программные маршрутизаторы, сетевые устройства типа мостов и концентраторов. Особенности протокола SNMP позволяют использовать его для удаленной настройки устройств, а также для мониторинга сети, детектирования ошибок и попыток несанкционированного доступа и аудита.
- TCP. Это уже рассмотренный нами ранее протокол управления надежной передачей данных. Transmission Control Protocol обеспечивает надежную сквозную передачу сообщений посредством узлов глобальной сети за счет создания и удержания виртуальных (логических) соединений поверх физических соединений.

- UDP. Это также протокол взаимодействия в глобальных сетях, как и TCP/IP, только со своей спецификой. User Datagram Protocol – это дейтаграммный протокол, который обеспечивает передачу информации особым способом при помощи дейтаграмм пользователя. Он работает как связующее звено между прикладными процессами внутри сети и сетевым протоколом.
- IP. Это, наряду с TC, один из основных протоколов сетевого уровня. Он в основном применяется для маршрутизации пакетов данных. Internet Protocol – это «направляющий передачу данных» протокол сетевого уровня. Изначально он задумывался как частный протокол трансляции пакетов в сетях с неоднородной структурой, т.е. в «больших» сетях, которые формируются из некоторого количества локальных сетей меньшего масштаба, объединенных связями.
- ARP. Это несложный служебный (вспомогательный) сетевой протокол. Он расшифровывается как Address Resolution Protocol и предназначен, в первую очередь, для определения неизвестного аппаратного MAC-адреса узла по его известному логическому IP-адресу.
- ICMP. Это еще один вспомогательный протокол стека TCP/IP. Расшифровывается как Internet Control Message Protocol. Он служит для обмена информацией об ошибках передачи данных по протоколу маршрутизации IP и для обмена управляющей информацией на сетевом уровне.
- GMP. Специализированный протокол для широковещательной передачи данных. Расшифровывается как Internet Group Management Protocol. Он применяется в основном тогда, когда необходимо отправить данные определенной группе получателей, т.е. сделать информационную служебную рассылку.

Сопоставление рассмотренного семейства сетевых протоколов, стека TCP/IP и сетевой модели OSI/ISO.

Как можно заметить из данного графического представления, несколько уровней модели OSI/ISO действительно соответствуют одному слою TCP/IP, и «поверх» этого слоя могут работать несколько прикладных протоколов.

На самом деле, чем выше используемый пользователем протокол по этой иерархии, тем больше он абстрагируется от конкретной реализации на более низких уровнях.

Например, протокол HTTP реализуется «поверх» протокола TCP/IP и является частью его «семейства». Ему «все равно» какая технология используется на низких уровнях, не важно какая среда передачи применяется в той или иной сети – всю работу по адаптации пакетов данных к особенностям сетей и все манипуляции на уровне сигналов выполняет соответствующее оборудование.

Рассмотрим далее другой ключевой вопрос для организации трансляции данных между абонентами в локальных сетях – сетевые топологии.

Любая архитектура сети описывает баланс между реализациями физического и канального уровней сетевой модели OSI/ISO. Она формализует требования к кабельной системе, к методам кодирования сигналов,

регламентирует скорость передачи, надежность и качество обслуживания, определяет формат кадров, топологию и методы доступа к удаленным ресурсам.

Все это в целом накладывает требования на оконечное оборудование данных и на используемое программное обеспечение.

В локальных и глобальных сетях сейчас применяются самые разные сетевые технологии. При выборе таких технологий в основном смотрят на следующие требования:

- максимальная и минимальная пропускная способности сети;
- скорость отклика сети;
- взаимное расположение узлов;
- расстояние между источниками и приемниками данных;
- условия прокладки коммуникаций;
- надежность соединений;
- конфиденциальность связи;
- защита от несанкционированного доступа;
- стоимость аппаратуры и создания коммуникаций;
- стоимость эксплуатации получившейся сети.

Топология сети отражает такие сети, которые не совсем зависят от их размера, но отражают структуру, образуемую узлами и связями между ними. При этом практически не учитываются следующие факторы:

5. производительность сети;
6. принцип работы конечных узлов;
7. длина каналов связи;
8. типы связей.

Классическим способом разделения типов соединений сетей является идентификация физического расположения различных функциональных компонентов сети. К таким компонентам можно отнести кабели, рабочие станции, коммуникационное оборудование и так далее.

Вторым критерием разделения является определение метода доступа к среде передачи.

Здесь также следует сказать, что данное разделение является весьма условным в современной практике «больших» сетей, так как в целом топология любой сети, хотя бы «среднего» масштаба, всегда окажется «смешенной».

Если обратиться к классическим трудам по компьютерным сетям, то в обязательном порядке определяют следующие базовые топологии.

- общая шина;
- звезда;
- кольцо;
- ячеистая или сотовая.

Рассмотрим последовательно особенности каждой топологии.

Сеть с топологией «общая шина». Также носит название моноканальной сети. Логично, что это сеть, основой для которой служит «моноканал».

Однозначно характеризуется подключением группы абонентских систем к одному моноканалу.

Топология с общей **шиной** обладает преимуществами:

- в небольших сетях работает наиболее надежно;
- очень проста в использовании;
- понятна для проектировщика;
- требует немного кабеля для соединения по сравнению с другими топологиями;
- как следует из предыдущего пункта, «общая шина» дешевле других типов построения сетей;
- легко расширить, просто подключив новых абонентов к уже имеющейся шине;
- общая протяженность кабелей меньше, чем у альтернативных решений;
- как следует из п. 7, у таких сетей более высокая надежность;
- как следует из п. 8, выход из строя одного узла не нарушает работоспособности сети в целом.

Недостатки топологии с общей шиной состоят в следующем:

- обрыв основного кабеля может привести к нарушению работы сразу всей сети;
- разрыв моноканала в одном месте как минимум поделит «шину» на две независимые части;
- по единственному каналу прокачивается трафик с высокой нагрузкой;
- как следует из п. 3, производительность такой сети снижается при увеличении «прокачки» данных;
- информация в системе слабо защищена на физическом уровне;
- как следует из п. 5, сообщения могут быть приняты и на любом другом подключенном компьютере или вообще перехвачены или изменены аппаратной «закладкой».

Сеть со «звездообразной» топологией – это разновидность древовидной сети, в которой имеется только один центральный узел. Этот же узел считается промежуточным, так как через него осуществляется передача всех информационных пакетов и служебных сообщений.

В качестве центрального элемента сети обычно выступает мультиплексор, концентратор или маршрутизатор.

Мультиплексор – это устройство, которое преобразует несколько входящих сигналов в один исходящий.

Концентратор – это устройство, позволяющее обслуживать большее количество источников и приемников данных по небольшому числу каналов.

Маршрутизатор – это устройство, управляющее трафиком на основе таблиц маршрутизации и устанавливающее соединения на физическом и логическом уровне между абонентами.

«Звездообразная» сеть, как уже сказано выше, имеет только один узел «в центре сети» и набор расходящихся от него связей со станциями на концах. В

«звездообразной топологии» все станции никак не связаны друг с другом напрямую, но взаимодействуют исключительно через этот центральный узел. Этот же самый узел управляет потоком сообщений в сети, а также контролирует и пропускает через себя весь трафик.

Естественно, что при выходе из строя этого центрального узла, сеть уничтожается полностью, т.е. перестает функционировать.

Расширять звездообразную топологию можно следующим образом: вместо одного из компьютеров подключают еще один концентратор и присоединяют к подсоединенному концентратору дополнительные машины, образуя вторичную сеть, которая логически является просто одной машиной для первичной сети.

Следует сказать, что сейчас такая топология является самой распространенной как для проводных и беспроводных сетей. В бытовом секторе, например, часто ставят маршрутизатор Wi-Fi и подключают к нему все устройства по беспроводному каналу. Этот маршрутизатор и является центральным звеном сети.

Преимущества сети звездообразной топологии состоят в том, что:

- сеть дешева в построении и в эксплуатации;
- сеть допускает добавление активных устройств, не нарушая уже имеющуюся архитектуру;
- центральный узел удобно использовать:
  - для мониторинга;
  - диагностики;
  - фильтрации трафика;
  - защиты данных;
  - проверки лицензий;
  - также других служебных целей;
- отказ одного из подключенных узлов не приводит к остановке сети;
- в сети допускается применение нескольких типов физических соединений – это все зависит только возможностей центрального узла (есть соответствующие порты) и возможностей абонента.

Недостатки подобной сети заключаются в том, что:

- отказ центрального узла «ломает» всю сеть;
- нужно много кабелей для соединения всех элементов, так как канал тянется ко всем участникам сети, что сказывается на конечной стоимости (хотя это не так актуально для беспроводной сети, так как мы там платим только за центральный узел – маршрутизатор).

Следующий тип сети носит название «кольцевая топология». По своей сути – это сеть, в которой каждый элемент соединен с другими узлами последовательно, но последний связан с первым. Таким образом, сообщения всегда будут передаваться с выхода одного узла на вход другого и с выхода последнего на вход первого. Каждая станция выступает в роли ретранслятора пакетов и прямо связана с двумя соседними.

Сеть с кольцевой топологией все же обладает определенными преимуществами:

1. все сетевые карты компьютеров имеют один и тот же уровень доступа и приоритет, но никто из них не может монополизировать сеть и «вытеснить» других участников обмена;
2. система более надежна по сравнению с другими топологиями, так как к каждому компьютеру подведено два канала связи, а в современных реализациях разрыв соединения просто делит сеть на два сегмента;
3. совместное использование передающей среды делает предсказуемой нагрузку и производительность, даже в условиях увеличения числа абонентов.

Недостатки сети с кольцевой топологией:

1. у таких сетей большая протяженность кабелей, так как необходимо покрыть все подключенные машины «с двух сторон»;
2. сетевая карта компьютера в кольцевой топологии сложнее в производстве, так как априори имеет два коннектора;
3. более слабая защищенность информации по сравнению со «звездой» и другими технологиями и возможность подключить «прозрачную закладку» в разрез сети и перехватывать трафик;
4. более низкое быстродействие по сравнению с топологией «звезда», но сравнимое с более простыми технологиями.

Мы рассмотрели простые топологические решения для компьютерных сетей, – они однородны внутри и являются «чистыми».

Однако в реальной ситуации топологии сетей являются комбинацией других более простых типов вследствие их размера и сложности. Рассмотрим далее несколько вариантов топологий, которые все чаще применяются на практике.

Это топологические организации сетей, имеющие сложную структуру: **ячеистую, сотовую или полносвязную.**

Например, полносвязная топология – это сеть, в которой все узлы связаны со всеми другими узлами сразу. Подобная сеть характеризуется наличием избыточных связей между устройствами. Для большого числа сетевых устройств такая схема вообще неприемлема, но отлично проявляет себя в высокопроизводительных вычислительных системах, в коммутационных матрицах устройств и в многопроцессорных системах.

Древовидная топология фактически является объединением нескольких «слоев» звездообразных сетей, где каждый следующий «слой» подключается к лучу предыдущего «слоя».

Следующим вариантом является сеть **гибридной топологии.** Такая сеть применяется чаще всего для подключения нескольких сетей, причем каждая такая подсеть может иметь свою топологию. Например, часто встречается сочетание магистральной линии в виде «кольца» и разветвленной системы древовидных подключений. С другой стороны, гибридная топология используется для создания глобальных вычислительных сетей.

Как следует из реальной практики, топология практически любой реальной сети может повторять одну из приведенных выше или включать их комбинацию.

Мы коснулись способов организации обмена и построения сети локального уровня, то есть, когда источник и приемник данных расположены рядом друг с другом с точки зрения соединения.

Если нужно передавать данные «далеко» за пределы локальной сети, используются методы и инструментальные средства маршрутизации в глобальной сети Интернет. В первую очередь это протоколы TCP/IP и UDP.

Протокол TCP обеспечивает надежность передачи сообщений через сети между абонентами за счет создания и поддержания виртуальных соединений.

Протокол UDP обеспечивает быструю передачу прикладных пакетов дейтаграммным способом. Он, как и IP, выполняет исключительно только те функции связующего звена, которые обеспечивают взаимодействие между сетевым протоколом и многочисленными прикладными процессами.

Верхний уровень стека протоколов I называется прикладным.

За долгие годы совершенствования стек TCP/IP вобрал в себя большое количество протоколов и сервисов прикладного уровня.

К ним относятся такие широко используемые протоколы, как:

1. протокол копирования файлов FTP;
2. протокол эмуляции терминала telnet;
3. почтовый протокол SMTP;
4. гипертекстовые сервисы доступа к удаленной информации, такие как WWW и многие другие.

Рассмотрим далее протокол межсетевого взаимодействия IP, дополняющий протокол TCP.

Основу этого стека протоколов составляет протокол межсетевого взаимодействия – Internet Protocol (IP).

К основным функциям протокола IP относят следующие:

1. инструментальные средства протокола осуществляют перенос различных типов адресной информации в унифицированной форме между различными сетями;

2. пакеты при переходе между сетями собираются и разбираются даже с различным максимальным значением длины пакета.

Пакет IP включает раздел данных и заголовок.

Рассмотрим последовательно основные поля пакета IP.

Поле номера версии Vers содержит версию используемого протокола IP.

Сейчас повсеместно используется версия 4, но все чаще начинает встречаться версия протокола IPv6.

Поле длины заголовка HLen занимает всего 4 бита и показывает число 32-битовых слов, составляющих саму длину поля заголовка.

Стандартный заголовок имеет длину в 20 байт, т.е. определяется как пять 32-битовых слов. При увеличении объема служебных данных длина может быть увеличена за счет резерва IP OPTIONS.

Поле типа сервиса или Type of Service (ToS) занимает всего 1 байт и задает приоритетность пакета. Приоритет пакета определяется видом критерия выбора маршрута.

Первые три бита ToS как раз и образуют подполе приоритета или precedence.

Приоритет может иметь значения от 0 до 7. Приоритет 0 имеет нормальный пакет, а приоритет 7 имеет пакет с управляющей информацией.

Использование приоритета было внедрено для того, чтобы маршрутизаторы могли определять очередь обработки пакетов и «пропускать» вперед более важные пакеты.

Поле Тип сервиса содержит также три бита, определяющие критерии выбора маршрута:

1. установленный бит паузы delay (D) говорит о том, что маршрут должен выбираться для минимизации задержки доставки данного пакета;
2. установленный бит T – для максимизации пропускной способности;
3. установленный бит R – для максимизации надежности доставки.

Поле общей длины или total length имеет размер в 2 байта и определяет общую длину пакета. В это значение входит длина заголовка и длина поля данных.

Поле идентификатора пакета или identification также имеет размер в 2 байта. По нему распознается отдельный небольшой пакет, который получается после фрагментации одного базового или исходного пакета. Все фрагменты этого пакета должны иметь одинаковое значение этого поля, соответственно это поле должно быть уникальным в текущем сеансе связи.

Поле флагов или flags имеет размер всего 3 бита, но этого достаточно для указания на возможность фрагментации пакета.

Поле смещения фрагмента или fragment offset занимает в пакете 13 бит. Поле используется для указания значения смещения этого поля данных в пакете от начала поля данных исходного пакета (который был подвергнут) фрагментации. Т.е. поле показывает смещение конкретно передаваемого блока данных в общем исходном блоке. Поле используется при сборке и при разборке фрагментов исходного пакета и помогает управлять передачей информации при трансляции через несколько сетей с различными величинами максимальной длины пакета.

Поле времени жизни или time to live имеет длину в 1 байт. Это значение указывает на предельное значение временного интервала, в течение которого пакет может транслироваться по сети между начальной и конечной станциями. Оно задается в виде максимально допустимого значения станцией-источником. Исходя из размера поля, максимальное время жизни пакета равно 255 секундам. На ретранслирующих узлах сети из текущего времени жизни вычитается единица либо когда действительно проходит секунда, либо при каждой транзитной передаче (даже если эта секунда еще не прошла). При истечении времени жизни пакет аннулируется, т.е. в сети невозможно бесконечно транслировать пакеты.

Идентификатор протокола верхнего уровня или protocol, занимает всего 1 байт. Он указывает, какому конкретно протоколу верхнего уровня принадлежит пакет. Чаще всего пакет может принадлежать протоколам TCP, UDP или RIP.

Контрольная сумма или header checksum занимает привычные 2 байта. Она рассчитывается по всему заголовку, исключая поле данных.

Поле адреса источника или source IP address и поле адреса назначения destination IP address имеют одинаковую длину – по 32 бита каждое. Также они имеют и одинаковую структуру.

Поле резерва или IP options необязательно используется в пакете. Но если оно существует, то применяется, как правило, только при отладке сети. Такое поле составляют из нескольких подполей. Каждое подполе может быть одного из восьми предопределенных типов.

В этих подполях можно указывать:

1) точный маршрут прохождения маршрутизаторов;
2) регистрировать проходимые пакетом маршрутизаторы;
3) помещать данные системы безопасности;
4) размещать временные отметки.

Так как число подполей может быть произвольным, то в конце поля резерва должно быть добавлено несколько байт для выравнивания заголовка пакета по 32-битной границе. Это стандартный подход для того, чтобы положение служебных полей и их размеры становились более «ожидаемыми» и упрощали анализ информационных пакетов, а также разработку сетевых программных и аппаратных средств.

Максимальная длина поля данных пакета ограничена разрядностью поля, определяющего эту величину (16 бит). Следовательно, оно составляет 65 535 байтов. Но это максимально возможное значение и, естественно, при передаче пакета по сетям с разным предельным значением длины поля данных, фактическая длина будет отличаться от максимальной. Например, если это кадры Ethernet, то выбираются пакеты с максимальной длиной в 1,5 Кб, умещающиеся в поле данных кадра Ethernet, а если это Fast Ethernet или Gigabit Ethernet, то длина будет другой, соответствующей стандарту.

Рассмотрим далее принципы управления фрагментацией пакетов в протоколах глобальных сетей.

Протоколы транспортного уровня TCP или UDP «считают», что изначальный размер поля данных IP-пакета равен 65 535. Поэтому они способны регламентировать логическую передачу таких пакетов через интерсеть. Но как уже говорилось выше, и эти пакеты будут разбиваться на составные для конкретного типа составляющей сети сообщения.

В большинстве локальных и глобальных сетей внедрено такое важное понятие, как максимальный размер поля данных кадра или пакета, в которые должен инкапсулировать свой пакет протокол IP.

Эту величину обычно называют максимальной единицей транспортировки или maximum transfer unit (MTU), например:

1) сети Ethernet имеют значение MTU, равное 1 500 байт;

2) сети FDDI имеют значение MTU, равное 4 096 байт;
3) сети X.25 чаще всего работают с MTU в 128 байт.

Работа инструментальных средств протокола IP по фрагментации пакетов расскажем на примере.

Допустим, что компьютер № 1 связан с сетью, определяющей MTU в 4 096 байтов (например, это оптоволоконная сеть стандарта FDDI). При приеме на IP-уровень компьютера 1-го сообщения с размером в 5 600 байт, протокол IP разделит его на два IP-пакета (проведет фрагментацию), установив признак фрагментации в первом пакете и присвоив пакету идентификатор (например, 486).

В первом пакете поле смещения будет содержать 0, так как это первый пакет серии. Во втором пакете поле смещения будет равным 2 800, так как это уже второй пакет из серии, и он смещен относительно начала именно на это значение байт.

Естественно, что признак фрагментации во втором пакете будет равен нулю. Мало того, что это второй пакет по счету, – это одновременно и последний пакет в серии.

Таким образом, общая величина IP-пакета составляет 2 800 байт + 20 байт (выделяемых под заголовок IP). Общая длина будет 2 820 байт.

Такая длина спокойно умещается в поле данных кадра FDDI и эти два пакета могут транслироваться через указанную оптическую сеть без проблем.

Далее компьютер № 1 транслирует эти пакеты на канальный уровень К1. После канального уровня пакет «опускается» на физический уровень Ф1.

С физического уровня пакеты отправляются маршрутизатору, который обеспечивает связь с сетью. Маршрутизатор смотрит свою внутреннюю таблицу маршрутизации и по сетевому адресу определяет, что эти два пакета нужно передать дальше в сеть № 2. Вторая сеть имеет несколько меньшее значение MTU. Например, оно равно 1 500 байт. Допустим, что это классическая сеть Ethernet.

Маршрутизатор должен извлечь фрагмент транспортного сообщения из каждого пакета FDDI и разделить его пополам. Это делается для того, чтобы каждая часть разделенного пакета поместилась в поле данных кадра Ethernet.

Таким образом он формирует новые пакеты в рамках протокола IP и каждый пакет уже имеет длину 1 400 байт + 20 байт, т.е. 1 420 байт.

Это значение, конечно, меньше 1 500 байтов, поэтому они спокойно размещаются внутри поля данных каждого кадра Ethernet.

В результате таких действий, в компьютер № 2 уже по сети Ethernet поступает уже четыре IP-пакета. Все они имеют один общий идентификатор 486. Идентификатор позволяет протоколу IP корректно собрать исходное сообщение обратно.

Примечательно то, что даже если пакеты по сетям вдруг пришли не в том порядке, в котором они были отправлены с компьютера-источника, то поле смещения укажет на правильный порядок для их объединения. Естественно, что все пакеты буферизуются внутри сетевых устройств и для них составляются

отдельные индексные таблицы и таблицы с указателями и отметками о приеме. Для отсутствующих пакетов сработают отдельные протоколы повторного запроса и восстановления.

Стоит также отметить, что маршрутизаторы IP-сетей не будут собирать фрагменты пакетов в более крупные, даже если дальше в соответствии с картой маршрутов будут встречаться сети, позволяющие передавать пакеты большего размера.

Это связано с тем, что нет никакой гарантии, что маршрут не поменяется и пакеты дальше будут перемещаться к соответствующим сетям с большой длиной пакета. Таким образом IP-сети реализуют как бы односторонний механизм фрагментации и отсекают лишнюю работу по постоянной разборке и сборке пакетов.

В IP-сетях также работает таймер обратного отсчета, который запускается для определения допустимого времени нахождения пакета в пределах одной сети (время жизни пакета). Если таймер истекает раньше момента фиксации факта прибытия последнего фрагмента из серии, то все фрагменты, полученные к этому моменту, просто отбрасываются. В этом случае, на узел, который отправил исходный пакет, отправляется сообщение об ошибке с помощью служебного протокола ICMP.

Рассмотрим далее, как в распределенных автоматизированных и информационных системах выполняется маршрутизация при помощи IP-адресов.

Мы можем показать данный процесс на примере, иллюстрированный рисунком ниже.

Пусть у нас есть составная сеть из 20 маршрутизаторов. Они изображены в виде пронумерованных блоков. Маршрутизаторы объединяют 18 сетей в общую сеть, где N1, N2, ..., N18 – это номера сетей.

На конечных узлах A и B установлены маршрутизаторы, работающие с протоколом IP.

Все маршрутизаторы имеют по несколько портов, к которым присоединяются физические сети. Каждый интерфейс (порт) маршрутизатора можно рассматривать как отдельный узел сети, так как он имеет свой отдельный сетевой адрес. Его сетевой адрес является локальным в той сети, которая подключена к порту.

Например, маршрутизатор № 1 имеет три порта, к которым подключены соответствующие сети N1, N2, N3. Их сетевые адреса обозначены как IP11, IP12 и IP13.

Логично, что порт IP11 является узлом сети N1. Следовательно, в поле номера сети порта IP11 содержится номер N1.

Аналогично работает и интерфейс IP12. Это фактически узел в сети N2, а порт IP13 – это узел в соответствующей локальной сети N3.

Любой маршрутизатор такой составной сети можно описывать как сочетание нескольких узлов, каждый из которых входит в свою собственную сеть.

В составных сетях практически всегда сосуществуют несколько альтернативных маршрутов для передачи пакетов. Информация о таких маршрутах описывает возможные пути передачи данных со своими собственными характеристиками, обрабатывается соответствующими аппаратными и программными алгоритмами, и содержится в таблице маршрутизации.

Пакет, отправленный из узла А в узел В через показанную сеть, может транслироваться через маршрутизаторы № 17, № 12, № 5, и № 1. Или же транслироваться через альтернативный путь при помощи маршрутизаторов № 17, № 13, № 7, № 6 и № 3.

Мы также можем указать еще несколько маршрутов между узлами А и В, но это скорее подтверждает правило и не важно для выбранного примера работы составной IP-сети.

Маршрут для трансляции пакетов определяется на основании имеющейся информации о конфигурации сети и на основании критериев выбора маршрута. В качестве критерия выбора маршрута могут быть выбраны следующие параметры:

1) средняя задержка прохождения маршрута пакетами;
2) средняя пропускная способность маршрута для последовательности пакетов;
3) количество промежуточных ретранслирующих устройств

Рассмотрим далее строение таблицы маршрутизации.

По своей сути это простая таблица из строк и столбцов, в которой находятся начальные, промежуточные и конечные адреса сетевых устройств и дополнительная служебная информация.

На рисунке ниже приведена примерная структура таблицы маршрутизации (представим, что это таблица из маршрутизатора № 4 в предыдущем примере).

Конечно, таблица в нашем примере значительно упрощена в сравнении с реальными таблицами. Здесь, например, отсутствуют маски маршрутов, признаки состояния маршрута, время валидности маршрута и так далее.

Следует помнить, что вместо номера целевой сети (сети назначения) может быть использован сетевой узел.

Первый столбец таблицы из нашего примера содержит адреса назначения пакетов. За адресом назначения указывается сетевой адрес следующего маршрутизатора в рассчитанной цепи следования пакетов. Точнее говоря, здесь указан сетевой адрес интерфейса следующего маршрутизатора.

Указанные в строках таблицы сегменты маршрута в своей совокупности показывают цепочку связанных сегментов сети и, перед тем как передать пакет следующему маршрутизатору, текущий маршрутизатор должен определить, на какой из нескольких собственных портов он должен поместить данный пакет. В нашем примере он выбирает один из двух портов: IP41 или IP42.

Для того чтобы сделать оптимальный выбор, маршрутизатор проводит анализ содержимого третьего столбца таблицы маршрутизации. Этот столбец содержит сетевые адреса выходных интерфейсов.

Следует помнить, что каждая реализация алгоритма маршрутизации может использовать свой собственный формат таблицы, например некоторые алгоритмы допускают наличие в таблице маршрутизации сразу нескольких строк для одного и того же адреса назначения. Это, фактически, альтернативные маршруты.

В случае присутствия альтернатив для выбора маршрута, во внимание принимается с параметром «расстояние до сети назначения». Данное расстояние измеряется в любой удобной и допустимой метрике.

Расстояние в сети может измеряться не только числом проходимых сетей, но и следующими параметрами, а зависимости от используемых алгоритмов:

1) временем нахождения пакета в линиях связи;
2) различными характеристиками надежности линий связи;
3) пропускной способностью сети;
4) величинами, отражающими качество данного маршрута по определенным критериям.

В нашем примере мы будем измерять расстояние между сетями специальным параметром, который называется «хоп». Хопы – это количество скачков или промежуточных сетей между начальной и конечной точками маршрута. В некоторых стандартах счетчик хопов начинается с 0, а в некоторых с 1.

Как только какой-либо пакет поступает на маршрутизатор, модуль IP извлекает номер сети назначения и строчка за строчкой сравнивает его последовательно с номерами сетей из таблицы маршрутизации.

Если в какой-либо строке номер совпадает с номером из пакета, то из таблицы считывается ближайший маршрутизатор, на который пакет направляется дальше.

Для нашего примера, если на какой-либо порт маршрутизатора № 4 поступает пакет, адресованный в сеть N6, то из таблицы маршрутизации будет определено, что адрес следующего по пути маршрутизатора будет равным IP21. Соответственно, следующим этапом движения данного пакета будет порт № 1 у маршрутизатора № 2.

Часто в качестве адреса назначения в таблице маршрутизации упоминается не весь IP-адрес, а только номер сети назначения. Тогда для всех пакетов, предназначенных для одной и той же сети, протокол IP будет формировать один и тот же маршрут. Правда, в этом примере мы не учитываем изменяющееся состояние сети, возможные обрывы соединений и появление новых промежуточных сетей по маршруту следования.

Если такие изменения всех же происходят, то дополнительный или альтернативный маршрут помечается в таблице путем внедрения новой строки, содержащей полный IP-адрес и маршрутную информацию.

Такая запись показана в нашем примере для узла B. Например, мы решили поменять таблицу маршрутизации для маршрутизатора № 4, руководствуясь соображениями безопасности. Т.е. пакеты, следующие в узел B с полным адресом IPв, должны будут транслироваться через маршрутизатор № 2 с портом IP21, а не

через маршрутизатор № 1 с портом IP12. При этом через маршрутизатор № 1 и порт IP12 по-прежнему должны передаваться данные всем остальным узлам сети N3.

В такой ситуации маршрутизатор автоматически будет отдавать предпочтение специфическому маршруту.

Пакет данных может быть отправлен в любую подсеть из составной сети. Поэтому может показаться, что таблица маршрутизации должна иметь записи обо всех этих составных сетях. Это ошибочное утверждение, поскольку в таком случае единая таблица маршрутизации будет иметь очень большой размер и потребует много места для хранения и много места для поиска маршрутов. Поэтому вместо одной большой таблицы создается набор небольших таблиц, содержащих в себе части общей картины. Таким образом глобальная таблица маршрутизации по своей сути является распределенной составной таблицей.

Другим способом уменьшения объема любой таблицы маршрутизации является введение маршрута по умолчанию или default route. Для формирования маршрута по умолчанию используются особенности топологии сети.

Давайте рассмотрим пример. Пусть существуют некоторые маршрутизаторы, находящиеся на периферии составной сети. У них существует всего несколько путей до центральных частей составной сети, которые можно объединить и сгенерировать универсальный маршрут только до той сети, которая непосредственно соединяет периферию с центральной частью. В нашем примере маршрутизатор № 4 указывает специфические маршруты только для пакетов, следующих в сети N1−N6. Для всех остальных пакетов (в сети N7−N18), маршрутизатор формирует путь по умолчанию через один и тот же порт IP51 и маршрутизатор № 5. Этот маршрутизатор также может называться маршрутизатором по умолчанию или default router.

В этой таблице в полях колонки «Адрес сети назначения» находятся адреса всех сетей, к которым подключен данный маршрутизатор.

Такое последовательное указание пути в виде отдельных «шагов» обусловлено тем, что в стеке TCP/IP применяется одношаговый подход к оптимизации маршрута или next-hop routing. Суть алгоритма заключается в том, что каждый маршрутизатор или конечный узел последовательно принимает участие в выборе только следующего одного шага для определения пути передачи пакета.

Именно поэтому в каждой строке таблицы фиксируется только один шаг, а не весь маршрут в виде последовательности IP-адресов. Это еще один фактор распределенной таблицы маршрутизации.

Еще одна особенность состоит в том, что вместе с пакетом данных, следующему элементу маршрута передается и ответственность за дальнейший выбор пути. Т.е. задача выбора маршрута следования пакетов данных также является распределенной, как и сама таблица.

В этом случае ограничение на максимальное количество транзитных маршрутизаторов на пути пакета действует только в плане ограничения времени жизни пакета.

Альтернативой же одношаговому подходу является указание в пакете всей последовательности маршрутизаторов, которые пакет должен пройти на своем пути.

Это второй глобальный способ маршрутизации, и он называется маршрутизацией от источника или source routing.

В этом случае выбор маршрута выполняешься первым маршрутизатором на пути пакета. Все остальные маршрутизаторы на заранее определенном пути только отрабатывают выбранный маршрут, передавая их с одного порта на другой и коммутируя пакеты.

Следует упомянуть, что алгоритм маршрутизации от источника в основном применяется в сетях IP только для отладки или анализа возможностей сети. В этом случае маршрут задается в поле Резерв или IP OPTIONS пакета.

Если в таблице маршрутов находится более одной строки для одного адреса назначения, то для принятия решения о передаче пакета в расчет берется только та строка, в которой указано наименьшее значение «Расстояния до сети назначения», выраженное в любом принятом формате измерения (например, в числе промежуточных сетей трансляции).

Так как в IP-сетях для маршрутизации используются адреса портов или устройств, то нужно каким-то образом определять физические адреса этих элементов. Для нахождения локального адреса по известному IP-адресу необходимо воспользоваться специальным протоколом определения адреса или address resolution protocol (ARP).

Когда для перенаправления пакетов сетевое устройство выбирает следующий узел, то он просматривают специальную кэш-таблицу адресов протокола ARP и находит там соответствие IP-адреса и MAC-адреса искомого устройства.

Если же такое соответствие еще не установлено или потеряно, т.е. запись в таблице отсутствует, то по локальной сети транслируется специальный широковещательный ARP-запрос. Устройства отвечают на этот запрос специальными пакетами, из которых извлекается локальный адрес и заносится в таблицу.

Следует учесть, что такое действие может проводиться на каждом промежуточном устройства при одношаговой маршрутизации. Особенно это актуально при прокладке первого маршрута. Т.е. сначала идет опрос устройств, затем формирование таблицы соответствий логических адресов сетевых устройств их физическим адресам, а затем уже формирование следующего шага маршрутизации.

Кстати говоря, нередко конечный узел большой составной сети работает вообще без таблицы маршрутизации. Он имеет только сведения об IP-адресе маршрутизатора по умолчанию, стоящего «выше его по иерархии».

Если в локальной сети присутствует только один маршрутизатор, то этот вариант – единственно возможный для всех конечных узлов.

Кроме маршрута по умолчанию, в таблице маршрутизации также могут встретиться два типа специальных записей:

1) запись о специфичном для узла маршруте;

2) запись об адресах сетей, непосредственно подключенных к портам маршрутизатора.

Специфичный для узла маршрут вместо номера сети содержит полный IP-адрес. Это потому, что считается, что для конечного узла маршрут может выбираться не так, как для всех остальных узлов сети.

Существуют различные алгоритмы построения таблиц для одношаговой маршрутизации. Их можно разделить на три класса:

1) алгоритмы фиксированной маршрутизации.
2) алгоритмы простой маршрутизации;
3) алгоритмы адаптивной маршрутизации.

Независимо от выбранного администратором сети алгоритма, результат их работы имеет единый формат.

Именно за счет стандартизации и единых форматов таблиц, разные алгоритмы могут строить таблицы маршрутизации, а разные узлы могут обмениваться между собой данными.

Алгоритмы фиксированной маршрутизации заключаются в простом определении таблиц маршрутизации администратором сети до запуска последней. Таблицы формируются «вручную» и автоматически не изменяются в процессе работы сети. Для внесения изменений опять требуется участие администратора сети. Нередко, таблицы маршрутизации при таком подходе «прошиваются» сразу в «железо» и работают уже не на программном уровне, а на уровне аппаратуры. Маршрутные данные быстро загружаются при запуске сети, быстро восстанавливаются после сбоев, но не адаптируются к изменениям сети.

Алгоритмы простой маршрутизации подразделяются на три основных подкласса:

1) случайная маршрутизация;
2) лавинная маршрутизация;
3) маршрутизация по предыдущему опыту.

Если используется алгоритм случайной маршрутизации, то пакеты передаются в любом, случайном направлении, кроме исходного.

При лавинной маршрутизации пакеты передаются во всех направлениях, кроме исходного. Также этот метод применяется в мостах для пакетов с неизвестным адресом доставки.

Маршрутизация по предыдущему опыту работает таким образом, что маршрутизаторы читают адресную информацию из передаваемых пакетов, а затем формируют таблицы маршрутизации. Именно так работают прозрачные мосты, собирая сведения об адресах узлов, входящих в сегменты сети. Такой способ маршрутизации обладает медленной адаптируемостью к изменениям топологии сети.

Алгоритмы адаптивной маршрутизации являются основным видом алгоритмов, применяющихся в современных сетях со сложной топологией.

Адаптивность основана на том, что сетевые устройства обмениваются специальной информацией о топологии и о составе сети. Также учитывается их пропускная способность, состояние, надежность и качество сервиса.

Адаптивные протоколы позволяют собирать и анализировать о связях в сети, оперативно отрабатывая изменения конфигурации связей.

Такие протоколы имеют «распределенный» характер. Он выражается в том, что в сети отсутствуют «выделенные» или «центральные» маршрутизаторы, которые обладают всей собственной информацией: все знания о топологии и качестве сети распределены между всеми маршрутизаторами, как и обязанности сбора и поддержания таких данных в актуальном состоянии.

Современные адаптивные протоколы маршрутизации мы можем поделить на две основные группы:

1) дистанционно-векторные алгоритмы или distance vector algorithms (DVA);
2) алгоритмы состояния связей или link state algorithms (LSA).

Сетевое оборудование для работы алгоритмов дистанционно-векторного типа периодически рассылает широковещательный «вектор» (специальный пакет), содержащий в себе информацию о расстояниях (хопах) от данного узла до всех известных ему сетей. При получении данного вектора от своего соседа, промежуточный маршрутизатор наращивает это расстояние до указанных в векторе сетей на расстояние до своего данного соседа. Кроме того, пакет дополняется информацией обо всех известных сетях для данного маршрутизатора, о которых он знает непосредственно сам или из других подобных векторов. Новое значение вектора отправляется далее. В конце концов, все маршрутизаторы в результате такого «лавинообразного» обмена данными будут знать обо всех соседних сетях.

Логично, что дистанционно-векторные алгоритмы будут хорошо работать только в небольших сетях, так как они создают большой периодический трафик.

В больших сетях они просто замусорят линии связи своими данными. Кроме того, изменения конфигураций сетей отрабатываются этими алгоритмами не слишком хорошо, так как не все маршрутизаторы своевременно получают обновления и не все имеют точные данные о топологических особенностях соседних сетей (в основном потому, что такие данные передаются от соседа к соседу, через посредников, а не централизовано).

Работа маршрутизатора с дистанционно-векторным протоколом в большей степени напоминает работу «моста», так как он транслирует информацию и не имеет точной топологической картины.

Однако такие алгоритмы хорошо зарекомендовали себя в некоторых служебных областях, например в протоколе маршрутной информации или routing information protocol (RIP). Протокол RIP. В свою очередь, распространен сейчас в двух основных версиях:

1) RIP IP, работающий с протоколом IP;
2) RIP IPX, работающий с протоколом IPX.

Следующая группа алгоритмов маршрутизации носит название алгоритмов состояния связей. Она обеспечивает необходимой информацией каждый маршрутизатор и ее достаточно для построения точного графа связей в компьютерной сети.

В основе алгоритмов лежит графовый подход и все маршрутизаторы располагаются в виртуальных вершинах графов, между которыми присутствуют связи. В ходе работы алгоритмов между узлами (маршрутизаторами) связанных сетей формируется подсистема связей. Эти подграфы объединяются для создания полного графа для всех сетей. Этот полный граф будет известен на всех маршрутизаторах. В качестве инструмента сбора данных тут так же используется широковещательная рассылка.

Кстати говоря, вершинами графа могут быть как маршрутизаторы, так и объединяемые ими сети в зависимости от «масштаба» представления информации.

Распространяемая по сети информация состоит из описания связей различных типов:

1) связь маршрутизатор – маршрутизатор;
2) связь маршрутизатор – сеть.

Для того чтобы определить в каком состоянии находятся линии связи, каждый маршрутизатор периодически посылает короткий пакет «HELLO» на соседние маршрутизаторы. Это служебный трафик. И он также засоряет сеть, но не в такой степени как, например, RIP-пакеты. Все-таки пакеты HELLO имеют совсем малый размер.

Рассмотрим далее альтернативную технологию для TCP, применяющуюся для трансляции пользовательских дейтаграмм – протокол UDP.

Протокол user datagram protocol (UDP) – это транспортный протокол для передачи данных в сетях IP, немного похожий на TCP, но без установления соединения.

Он является одним из самых простых и востребованных протоколов обмена транспортного уровня в современной сетевой модели OSI/ICO.

Основным «элементом» передачи данных согласно протоколу UDP является дейтаграмма.

В отличие от TCP, протокол UDP не подтверждает факт доставки данных, не заботится о порядке следования пакетов и не делает повторов при возникновении сбоев и откатов сети.

В связи с этим, аббревиатуру UDP иногда вольно расшифровывают как unreliable datagram protocol, то ест протокол ненадежных датаграмм.

Тем не менее протокол часто востребован для высокоскоростного обмена данными, где подтверждение не является обязательным.

С другой стороны, отсутствие соединения, дополнительного трафика и возможность широковещательных рассылок делают его удобным для применений, где малы потери, в массовых рассылках локальной подсети, в медиапротоколах и так далее.

Значение поля «длина датаграммы» указывает на длину всего UDP-сообщения, т.е. включая и UDP-заголовок. Измеряется в октетах (байтах). Для вычисления максимальной длины данных в UDP-сообщении при передаче в IP-сетях необходимо учесть, что UDP-сообщение в свою очередь является содержимым области данных IP-сообщения.

Максимальный размер IP-сообщения с учетом заголовка, передаваемый при помощи протокола UDP, равен 65 535 байт. Это потому, что максимальная длина UDP-сообщения (за вычетом минимального IP-заголовка) равна 65 535 байт – 20 байт, т.е. 65 515 байт.

Длина заголовка UDP-сообщения равна 8 байт, следовательно, максимальная длина данных в UDP-сообщении равна 65 515 байт − 8 байт, т.е. 65 507 байт, как это показано на рисунке ниже.

На практике нерационально использовать максимальную величину IP-пакета, так как такой размер превышает MTU основных протоколов канального уровня, и, следовательно, требует фрагментации IP-пакета, поэтому обычно используется размер, соотнесенный с MTU используемого канального протокола.

Заполнение IP-сообщения UDP-заголовок не содержит информации об адресе отправителя и получателя, поэтому даже при совпадении порта получателя нельзя с точностью сказать, что сообщение пришло в нужное место. Для проверки того, что UDP-сообщение достигло пункта своего назначения, используется дополнительный псевдозаголовок.

Поле «протокол» содержит в себе значение 17 (00010001 в двоичном виде, 0x11 − в шестнадцатеричном) − идентификатор UDP-протокола. Поле «длина UDP-датаграммы» содержит в себе длину UDP-сообщения (UDP-заголовок + данные; длина псевдозаголовка не учитывается) в байтах, т.е. совпадает с одноименным полем в UDP-заголовке.

Псевдозаголовок не включается в UDP-сообщение. Он используется для расчета контрольной суммы перед отправлением сообщения и при его получении (получатель составляет свой псевдозаголовок, используя адрес хоста, с которого пришло сообщение, и собственный адрес, а затем считает контрольную сумму).

Перед расчетом контрольной суммы UDP-сообщение дополняется в конце нулевыми битами до длины, кратной 16 битам (псевдозаголовок и добавочные нулевые биты не отправляются вместе с сообщением). Поле контрольной суммы в UDP-заголовке во время расчета контрольной суммы отправляемого сообщения принимается нулевым. Для расчета контрольной суммы псевдозаголовок и UDP-сообщение разбивается на слова (1 слово = 2 байта (октета) = 16 бит). Затем рассчитывается поразрядное дополнение до единицы суммы всех слов с поразрядным дополнением. Результат записывается в соответствующее поле в UDP-заголовке. В том случае, если контрольная сумма получилась равной нулю, то поле заполняют единицами. Если контрольную сумму не требуется рассчитывать, то значение поля оставляют нулевым.

При получении сообщения получатель считает контрольную сумму заново (уже учитывая поле контрольной суммы), и, если в результате получится двоичное число из шестнадцати единиц (т.е. код 0xffff), то контрольная сумма считается сошедшейся, и сообщение принимается.

Порт отправителя фактически указывает на порт процесса или устройства, которое посылает датаграмму. Если порт нужно указать пустым, то в него просто пишутся нули. Порт получателя имеет смысл только в контексте конкретного Internet адреса получателя.

К UDP-заголовку добавляется псевдозаголовок, в котором указываются адреса отправителя и получателя, номер протокола и длина UDP-датаграммы. Процедура вычисления контрольной суммы такая же, как и в протоколе TCP. Если расчетная контрольная сумма равна нулю, то соответствующее поле заполняется всеми единицами. Если поле заполнено одними нулями, это означает, что отправитель датаграммы не вычислял контрольной суммы, что может быть сделано при отладке, а также для тех протоколов, которые не требуют точности передачи. Здесь следует внести небольшое пояснение. При дополнении до единицы, или же, при обратном представлении числа, запись со всеми нулями эквивалентна записи со всеми единицами. Обе они обозначают нуль. Поскольку первый бит является знаковым, число, состоящее из всех нулей, называют положительным нулем, а состоящее из всех единиц − отрицательным. Несмотря на их математическую эквивалентность, в поле контрольной суммы UDP, как можно было видеть, они используются по-разному. Положительный ноль указывает на то, что контрольная сумма намеренно не вычислялась, а отрицательный ноль, на то, что вычисленная контрольная сумма оказалась равной нулю. Модуль протокола UDP должен иметь возможность извлекать из Internet заголовка датаграммы Internet адреса отправителя и получателя, а также тип протокола. Один из возможных интерфейсов UDP/IP мог бы возвращать в ответ на команду получения полную Internet датаграмму, включая Internet заголовок целиком. Такой интерфейс мог бы также позволить протоколу UDP передавать протоколу IP для посылки некую готовую Internet датаграмму вместе с заголовком. Протокол IP мог бы лишь проверять определенные поля Internet-заголовка на совместимость, а также вычислять контрольную сумму.

Протокол считается не очень надежным, что выражается вероятностью потери отдельных пакетов или, наоборот, в их дублировании.

Вследствие этого UDP чаще всего используется при передаче потокового видео- или аудио-, в компьютерных играх для связи клиентов и сервера, а также для передачи некоторых других специфических типов данных.

Ненадежность протокола UDP надо понимать в том смысле, что в случаях влияния внешних факторов, приводящих к сбоям, протокол UDP не предусматривает стандартного механизма повторения передачи потерянных пакетов. В этом смысле он настолько же надежен, как и протокол ICMP. Если приложению требуется большая надежность, то используется протокол TCP или SCTP либо реализуется какой-нибудь свой нестандартный алгоритм повторения передач в зависимости от условий.

UDP используется в следующих известных протоколах:

1) служба доменных имен или domain name system (DNS);

2) протокол передачи данных в реальном времени или real-time transport protocol (RTP);

3) протокол управления передачей в реальном времени или real-time transport control protocol (RTCP);

4) защищенный протокол передачи медиа-потока в реальном времени или secure real-time media flow protocol (RTMFP);

5) простой протокол передачи файлов или trivial file transfer protocol (TFTP);

6) протокол синхронизации времени по компьютерной сети или Simple Network Time Protocol (SNTP);

7) протокол синхронизации времени по компьютерной сети или simple network time protocol (NTP);

8) протокол сетевого доступа к файловым системам или network file system (NFS);

9) протокол динамической конфигурации узла или dynamic host configuration protocol (DHCP).

Все эти протоколы применяются для организации серевого обмена в автоматизированных системах на разном уровне, а также для создания устойчивых конфигураций распределенных или облачных систем.

Облачные технологии сейчас это модель обеспечения удобного сетевого доступа по требованию к некоторому общему фонду конфигурируемых вычислительных ресурсов (например, сетям передачи данных, серверам, устройствам хранения данных, приложениям и сервисам — как вместе, так и по отдельности), которые могут быть оперативно предоставлены и освобождены с минимальными эксплуатационными затратами или обращениями к провайдеру.

Потребители облачных вычислений могут значительно уменьшить расходы на инфраструктуру информационных технологий (в краткосрочном и среднесрочном планах) и гибко реагировать на изменения вычислительных потребностей, используя свойства эластичных вычислений.

С точки зрения поставщика, благодаря объединению ресурсов и непостоянному характеру потребления со стороны потребителей, облачные вычисления позволяют экономить на масштабах, используя меньшие аппаратные ресурсы, чем требовались бы при выделенных аппаратных мощностях для каждого потребителя, а за счёт автоматизации процедур модификации выделения ресурсов существенно снижаются затраты на абонентское обслуживание.

С точки зрения потребителя эти характеристики позволяют получить услуги с высоким уровнем доступности (англ. high availability) и низкими рисками неработоспособности, обеспечить быстрое масштабирование вычислительной системы благодаря эластичности без необходимости создания, обслуживания и модернизации собственной аппаратной инфраструктуры.

Сейчас «в облаках» расположены целые вычислительные сети и кластеры, системы документооборота, библиотеки, приложения из сферы услуг и продаж.

Давайте рассмотрим основные классификации облачных услуг по признаку вычислительной сети.

1. Инфраструктура как услуга (IaaS)

Модель, в которой имеется инфраструктура, предоставляемая на аутсорсинг для поддержки операций внутри предприятия, называется IaaS. Оборудование, программное обеспечение, хранилище, центры обработки данных, серверы и сетевое пространство предоставляются в этой услуге. IaaS справедливо называют аппаратным обеспечением как услугой (HaaS). Виртуализация платформы осуществляется в IaaS.

2. Платформа как услуга (PaaS)

Служба, предоставляющая аппаратные и программные средства для разработки приложений для пользователей, называется PaaS. С помощью этой службы различные компоненты интегрируются в базовую инфраструктуру организации. Кроме того, он предоставляет системы управления базами данных и библиотеки языков программирования. В этой службе можно выполнять редактирование, компиляцию, тестирование и управление версиями. Интеграция веб-сервисов является преимуществом PaaS. Эта услуга проста и удобна в использовании.

3. Программное обеспечение как услуга (SaaS)

В рамках этой услуги программное обеспечение, распространяемое централизованно, размещается и лицензируется. SaaS аналогичен поставщику услуг приложений (ASP). Поставщик создает единственную копию приложения, которая предоставляется всем пользователям. Пользователи могут добавлять новые функции или функциональные возможности в зависимости от их использования в программное обеспечение на основании соглашения с поставщиком. API могут быть интегрированы с собственными инструментами компании.

4. Функция как услуга (FaaS)

Пользователи могут создавать, запускать и защищать приложение со всеми его функциями и службами с помощью FaaS. Услуги зависят от событий, и с пользователей взимается плата в зависимости от их использования. Пользователям не нужно беспокоиться о серверах и их работе. Задачи могут быть запланированы, и приложения с большим объемом памяти могут быть легко использованы в этом сервисе.

5. Сеть как услуга (NaaS)

Пользователи, которые не хотят использовать свои собственные сети, получают помощь от поставщиков услуг для размещения сетевой инфраструктуры. Подключение и пропускная способность предоставляются поставщиком услуг на период действия контракта. Он в основном представляет сеть как транспортное соединение. Виртуализация сети выполняется в этой службе.

Теперь рассмотрим классификацию на основе принципа развертывания.

1. Публичное облако

Пользователи через Интернет используют облачные сервисы, в которых инфраструктура основана на компании облачных вычислений. Однако общедоступное облако не является безопасным, и организациям с конфиденциальной информацией не следует использовать это облако. Эта информация доступна для всех, кто использует облако.

2. Частное облако

Облачная инфраструктура расположена внутри организации, и она никому не предоставляется без разрешения организации и называется частным облаком. Мы можем сказать, что частное облако является наиболее безопасным среди всех облачных развертываний. Настройка, масштабирование и управление гибкостью выше в частном облаке. Аппаратное и программное обеспечение создаются только для владельца.

3. Гибридное облако

Сочетание частных и общедоступных облачных развертываний называется гибридным облаком. Организации имеют преимущество публичного облака при выполнении больших рабочих нагрузок, и информация защищена, поскольку частное облако включено в то же самое.

4. Облако сообщества

Модель развертывания, предоставляемая для ограниченного числа отдельных лиц и организаций, так что службы доступны только внутри них, называется облаком сообщества. Это обеспечивается как внутри, так и снаружи и размещается в зависимости от необходимости.

Различные типы облачных сервисов помогают пользователям всеми способами в полной мере использовать сервис. Поставщики услуг используют весь свой потенциал для обслуживания клиентов. Клиенты должны быть бдительны, чтобы их данные были защищены с помощью поставщиков услуг, предлагающих полную безопасность.

ТЕМА 18. ПРОГРАММНЫЕ И АППАРАТНЫЕ СРЕДСТВА ЗАЩИТЫ ИНФОРМАЦИИ В ИС И АС

В данной лекции мы рассмотрим методы защиты информации в автоматизированных и информационных системах.

Мы рассмотрим способы защиты сетевых компонент АС, а также дополнительно будет проведен анализ методов защиты соединения для передачи информации в локальных и глобальных сетях и будут описаны основные криптографические протоколы безопасности.

Следует учесть, что практически все криптографические методы и средства сейчас реализуются как на программном уровне, так и на уровне аппаратуры при помощи микроконтроллеров, программируемых логических интегральных схем (ПЛИС) или специализированных микрочипов, например в смарт-картах.

Для начала рассмотрим метода защиты данных на стороне клиента.

Для начала определимся с задачей защиты данных на стороне клиента. Под такой защитой мы подразумеваем невозможность (ну или хотя бы высокую сложность) для злоумышленника получить конфиденциальные данные клиента.

Как уже говорилось выше, в наши дни всё больше программ переводятся в так называемый «веб-ориентированный» вид, т.е. используется принцип клиент-сервер, что позволяет хранить данные удалённо и получать к ним доступ через тонкий клиент или браузер.

Одновременно с удобством использования остро встаёт вопрос о защищённости этих данных. Конфиденциальная информация может стать доступна другим людям несколькими путями.

Во-первых, к пользователю могут быть применены физические меры.

Во-вторых, при передаче данные могут быть перехвачены различными снифферами.

В-третьих, на сервер могут быть произведены хакерские атаки, что позволит злоумышленникам похитить информацию, либо недобросовестный администратор сервера воспользуется ею в личных целях.

Первое, что приходит на ум для защиты данных клиента, это обеспечение невозможности эти данные перехватить или модифицировать. А это можно сделать, например, при помощи шифрования.

В самом простом случае мы можем ориентироваться на три типа информации, которая переходит между клиентом и сервером:

- Текстовая информация.
- Файлы.
- Изображения и другие файлы.

Для защиты текста достаточно его собрать на стороне клиента, зашифровать и отправить на сервер.

Обратный процесс происходит на стороне сервера: информация получается, расшифровывается, обрабатывается, затем сервер шифрует данные

и отправляет ответ на клиент. Ответ будет дешифроваться уже на стороне клиента.

Работа с файлами и изображениями похожа на работу с текстом, только шифрование и расшифровка должна выполняться не над текстом, а над файловыми потоками. Кроме того, в этот процесс включается еще и этап преобразования содержимого файлов в формат Base64.

Можно самому реализовывать такой процесс при помощи программ (например, для работы в современных браузерах применяются HTML5, JavaScript и XmlHttpRequest Level 2) или при помощи фреймворков.

Например, клиентская библиотека службы хранилища Azure для .NET поддерживает шифрование данных в клиентских приложениях перед их отправкой в службу хранилища Azure и их расшифровку во время скачивания клиентом. Библиотека также поддерживает интеграцию с хранилищем ключей Azure для управления ключами учетной записи хранения.

Этот процесс называется «шифрование конвертным методом» и происходит следующим образом.

- Клиентская библиотека хранилища Azure создает ключ шифрования содержимого (CEK), который является симметричным ключом для однократного использования.
- Данные пользователя шифруются с помощью этого ключа CEK.
- Ключ CEK, в свою очередь, шифруется с помощью ключа шифрования ключа KEK. KEK определяется идентификатором ключа и может быть парой асимметричных ключей или симметричным ключом. Им можно управлять локально, а также хранить его в хранилище ключей Azure.
- Сама клиентская библиотека хранилища не имеет доступа к ключу KEK. Библиотека вызывает алгоритм шифрования ключа, который обеспечивается хранилищем ключей.
- Пользователи могут при необходимости использовать настраиваемые поставщики для шифрования и расшифровки ключа.
- Зашифрованные данные затем передаются в службу хранилища Azure. Зашифрованный ключ вместе с дополнительными метаданными шифрования хранится как метаданные (в большом двоичном объекте) или вставляется в зашифрованные данные (сообщения в очереди и табличные сущности).

Расшифрование серверных сообщений конвертным методом происходит следующим образом.

- Клиентская библиотека предполагает, что пользователь управляет ключом шифрования ключа KEK локально или через хранилище ключей Azure. Пользователь может не знать, какой именно ключ использовался для шифрования. Вместо этого достаточно настроить и использовать сопоставитель ключей, который будет распознавать разные идентификаторы ключей.
- Клиентская библиотека скачивает зашифрованные данные вместе с данными шифрования, которые хранятся в службе.
- Зашифрованный ключ шифрования содержимого CEK расшифровывается с помощью ключа шифрования ключа KEK. Клиентская

библиотека не имеет доступа к ключу KEK. Она просто вызывает пользовательский алгоритм или алгоритм расшифровки поставщика хранилища ключей.

- Затем ключ шифрования содержимого CEK используется для расшифровки зашифрованных пользовательских данных.

Естественно, что этот процесс является примером и конвертный метод может быть реализован не только на базе технологии .NET. Целью данного примера было показать общий подход и продемонстрировать тот факт, что проблема защиты данных на стороне клиента настолько важна, что даже такие мировые лидеры в области разработки программного обеспечения как Microsoft, давно включили эти инструменты в стандартный набор разработки веб-приложений.

Также существует чуть более простой метод защиты данных на стороне клиента – это защита от CSRF.

CSRF – это межсайтовая подделка запроса, т.е. вид атаки на клиента, при которой, если жертва заходит на сайт, созданный злоумышленником, вместо оригинального ресурса, то от лица этой жертвы тайно отправляется запрос на другой сервер (например, на сервер платёжной системы), осуществляющий некую вредоносную операцию (например, перевод денег на счёт злоумышленника).

Для осуществления данной атаки жертва должна быть аутентифицирована на том сервере, на который отправляется запрос, и этот запрос не должен требовать какого-либо подтверждения со стороны пользователя, которое не может быть проигнорировано или подделано атакующим скриптом.

Защита от CSRF выполняется следующим образом: в каждый запрос, которым можно что-то поменять на сервере, вставляется специальный ключ или токен.

Токен должен удовлетворять следующим требованиям:

- для каждой операции должен генерироваться свой уникальный токен;
- токен работает только один раз, потом он становится недействительным;
- токен должен иметь размер, устойчивый к простому подбору за разумное время;
- токен должен быть сгенерирован криптографически стойким генератором псевдослучайных чисел;
- токен должен иметь ограниченное время жизни.

По сути, сейчас существует всего три основных метода использования токенов в защите:

- Synchronizer Tokens (Statefull).
- Double Submit Cookie (Stateless).
- Encrypted Token (Stateless).

Говоря простыми словами, Synchronizer Tokens – это самый простой подход, использующийся повсеместно.

Он требует хранения токена на стороне сервера. Суть метода такова:

- При старте сессии на стороне сервера генерируется токен.
- Токен кладется в хранилище данных сессии (т.е. сохраняется на стороне сервера для последующей проверки).
- В ответ на запрос (который стартовал сессию) клиенту возвращается токен.
- Если рендеринг происходит на сервере, то токен может возвращаться внутри HTML, как, например, одно из полей формы, или внутри <meta> тега.
- В случае, если ответ возвращается для JS приложения, токен можно передавать в header (часто для этого используют X-CSRF-Token).

При последующих запросах клиент обязан передать токен серверу для проверки. А при рендере контента сервером токен принято возвращать внутри POST данных формы.

Современные JavaScript-приложения (например, написанные на React, Angular), обычно присылают XHR запросы с header (X-CSRF-Token), содержащим токен.

При получении запроса небезопасным методом (POST, PUT, DELETE, PATCH) сервер обязан проверить на идентичность токен из данных сессии и токен, который прислал клиент.

Если оба токена совпадают, то запрос не подвергся CSRF-Атаке, в ином случае – фиксируем событие и отклоняем запрос.

Достоинства у этого метода, следующие:

- Защита от CSRF на высоком уровне.
- Токен обновляется только при пересоздании сессии, а это происходит, когда сессия истекает.
- Во время жизни одной сессии все действия будут проверяться по одному токену.
- В браузере поддерживается защита данных между несколькими вкладками, на которых открыт один и тот же сайт, т.е. токен не инвалидируется после выполнения запроса, что позволяет разработчику не заботиться о синхронизации токена в разных табах браузера, так как токен всегда один.

Однако, если произойдет утечка токена, то злоумышленник сможет выполнить CSRF-Атаку на любой запрос и в течение долгого срока. А это плохо.

Подход Double Submit Cookie не требует хранения данных на стороне сервера, а значит, является Stateless.

Используется, если есть необходимость быстро и качественно масштабировать Web-сервис горизонтально, т.е. распределять нагрузку на копии одного и того же сервиса, стоящие за прокси-сервером балансировки нагрузки.

Идея в том, чтобы отдать токен клиенту двумя методами: в куках и в одном из параметров ответа (header или внутри HTML).

Суть метода состоит в следующем:

1. При запросе от клиента на стороне сервера генерируется токен. В ответе токен возвращается в cookie (например, X-CSRF-Token) и в одном из параметров ответа (в header или внутри HTML).

2. В последующих запросах клиент обязан предоставлять оба полученных ранее токена. Один как cookie, другой либо как header, либо внутри POST данных формы.

3. При получении запроса небезопасным методом (POST, PUT, DELETE, PATCH) сервер обязан проверить на идентичность токен из cookie и токен, который явно прислал клиент.

4. Если оба токена совпадают, то запрос не подвергся CSRF-Атаке, в ином случае – логируем событие и отклоняем запрос.

В результате применения данного метода мы имеем хорошую Stateless CSRF защиту.

Однако необходимо учитывать, что поддомены могут читать cookie основного домена, если явно это не запрещать (т.е. если cookie установлена на site.ru, то её могут прочитать как a.site.ru, так и b.site.ru).

Таким образом, если ваш сервис доступен на домене 3-го уровня, а злоумышленник имеет возможность зарегистрировать свой ресурс на вашем домене 2-го уровня, то следует установить cookie на свой домен явно.

Encrypted Token, так же, как и Double Submit, является Stateless подходом. Основная идея метода состоит в том, что если вы зашифруете надежным алгоритмом какие-то данные и передадите их клиенту, то клиент не сможет их подделать, не зная ключа. Этот подход не требует использования cookie. Токен передаётся клиенту только в параметрах ответа.

В данном подходе токеном являются факты, зашифрованные ключом. Минимально необходимые факты – это идентификатор пользователя и timestamp времени генерации токена. Ключ не должен быть известен клиенту.

Суть метода заключается в следующем.

1. При запросе от клиента на стороне сервера генерируется токен.
2. Генерация токена состоит в зашифровке фактов, необходимых для валидации токена в дальнейшем.
3. Минимально необходимые факты – это идентификатор пользователя и timestamp. В ответе токен возвращается в одном из параметров ответа (В header или внутри HTML).
4. В последующих запросах клиент обязан предоставлять полученный ранее токен.
5. При получении запроса небезопасным методом (POST, PUT, DELETE, PATCH) сервер обязан валидировать токен, полученный от клиента.

Валидация токена заключается в его расшифровке и сравнении фактов, полученных после расшифровки, с реальными. (Проверка timestamp необходима для ограничения времени жизни токена).

Если расшифровать не удалось либо факты не совпадают, считается, что запрос подвергся CSRF-Атаке.

Для данного метода также можно указать высокий уровень Stateless CSRF защиты данных на стороне клиента и отсутствие необходимости хранить данные в cookie. Также отсутствуют определенные нюансы с поддоменами, присущие предыдущему рассмотренному методу.

Теперь перейдем к методам защиты данных на стороне сервера.

Мы рассмотрели некоторые способы защиты данных на стороне клиента в клиент-серверной архитектуре, теперь настало время рассмотреть, как можно выполнить защиту данных на стороне сервера. Это непростая задача, но без должной защиты вся инфраструктура окажется под угрозой.

В первую очередь мы должны позаботиться о безопасном доступе к серверу с «внешней» стороны.

Самым популярным методом защиты доступа является SSH-соединение и использование SSH-ключей.

В основе этой технологии лежит пара криптографических ключей, которые используют для проверки подлинности в качестве альтернативы аутентификации с помощью пароля. Система входа использует закрытый и открытый ключи, которые создают до аутентификации. Закрытый ключ хранится в тайне надежным пользователем, в то время как открытый ключ может раздаваться с любого сервера SHH, к которому нужно подключиться.

Между клиентами и сервером устанавливается SSH-соединение с ключом и дальнейший обмен данными будет проходить уже внутри защищенной сессии.

Чтобы настроить аутентификацию через SSH-ключи, мы должны поместить открытый ключ в специальной директории на сервере. Когда пользователь подключается к серверу, SSH увидит запрос соединения и адрес, откуда производится запрос.

Далее он использует открытый ключ, чтобы создать и отправить вызов. Вызов – это зашифрованное сообщение, на которое нужен соответствующий ответ, чтобы получить доступ к серверу. Корректно ответить на сообщение сможет только держатель закрытого ключа. Т.е. только он может принять вызов и создать соответствующий ответ. Открытый же ключ используется для зашифровки сообщения, но это же самое сообщение расшифровать не может.

Вызов и ответ проходят незаметно для пользователя. Пока существует закрытый ключ, который обычно хранится в зашифрованном виде в ~/.ssh/, любой клиент SSH сможет отправить правильный ответ серверу.

Давайте подумаем, как SSH повышает безопасность сервера.

С помощью SSH любой вид аутентификации полностью зашифрован. Однако, если разрешена аутентификация на основе пароля, злоумышленники могут добраться до данных сервера. С помощью современных вычислительных мощностей можно получить доступ к серверу за счёт автоматизации попыток взлома, вводя комбинацию за комбинацией, пока правильный пароль не будет найдет.

Установив аутентификацию по SSH-ключам, мы можем забыть о паролях. Ключи имеют гораздо больше битов данных, чем пароли (например, 1024 бита или даже 4096 бит), что означает значительно большее число комбинаций, которые должны подобрать взломщики. Многие алгоритмы SSH-ключей

считаются «невзламываемыми» современной вычислительной техникой просто потому, что они требуют слишком много времени для подбора совпадений.

Другим способом повысить безопасность сервера является фаервол или брэндмауэр.

Брандмауэр – это часть программного или программно-аппаратного обеспечения, которая фильтрует сетевой трафик и контролирует доступ к сети. Это означает блокирование или ограничение доступа к каждому открытому порту, кроме исключений.

На типичном сервере ряд компонентов фаервола запущен по умолчанию. Их можно разделить на группы:

1. Открытые службы, к которым может подключиться каждый в интернете, часто анонимно. Хороший пример – веб-сервер, который разрешает доступ к вашему сайту.

2. Закрытые службы, которые доступны только из определенных мест или авторизованным пользователям. Пример – панель управления сайтом или базой данных.

3. Внутренние службы, доступные внутри самого сервера, без доступа к внешним источникам. Например, база данных, которая принимает только локальные соединения.

Применение брандмауэра гарантирует, что доступ к программному обеспечению и данным будет ограничен в соответствии с вышеуказанными категориями. Закрытые службы могут настраиваться по множеству параметров, что дает гибкость в построении защиты. Для не использующихся портов можно настроить блокировку в большинстве конфигураций.

Давайте подумаем, как фаервол может помочь в защите сервера от атак.

Для этого можно рассмотреть следующий пример, который часто встречается на практике. В современном мире вредоносного программного обеспечения достаточно просто найти и использовать сканер портов. Его даже можно легко сделать самому.

Работа сканера портов заключается в повторении попыток соединения с известным хостом (например, 15.12.54.243) и переменным портом, например:

1. Соединение с 15.12.54.243:1… неудача.
2. Соединение с 15.12.54.243:1… неудача.
3. …
4. Соединение с 15.12.54.243:5432… удача.
5. … и так далее…

В данном примере кто-то оставил закрытым порт 5432, а мы знаем, что база данных PostgreSQL имеет стандартный порт 5432. Это известный факт и для другого серверного программного обеспечения:

1. Порт 80 или 8080 часто используют PHP сервера.
2. Порт 3000 часто используют NodeJS сервера.
3. Порт 8000 часто используют Django сервера.
4. Порт 6379 применяется на сервере Redis.
5. Порт 3306 часто установлен для баз данных MySql и так далее.

Т.е. если мы видим какой-либо открытый порт, то, скорее всего, есть что-то, что работает через него. Если эта база данных, то мы можем применять атаки на основе SQL-инъекций для создания пользователей на сервере и вызова системных утилит.

Позволив захватить базу данных, мы дадим злоумышленнику возможность украсть или зашифровать базу данных, очистить ее и потребовать выкуп, т.е. наш ущерб ограничен только целями и фантазией злоумышленника.

Из этого следует, что сокрытие неиспользуемых портов или переназначение их номера существенно повышает защищенность сервера. В идеале, внешние системы или клиенты вообще не должны видеть служебные порты, а мы должны иметь возможность при помощи фаервола полностью контролировать зоны доступности для портов, направления приема и передачи данных и так далее.

Еще одним способом защиты сервера можно называть VPN и Private networking.

VPN (виртуальная частная сеть) – способ создать защищенное соединение между удаленными компьютерами и текущим соединением. Дает возможность настроить свою работу с сервером таким образом, словно вы используете защищенную локальную сеть.

Фактически VPN «накладывает» логическую сеть на физическую и «перемещает» ваш компьютер в новую локацию. Это особенно актуально, если вы работаете с защищенным контентом или с программным обеспечением, которое функционально только в некоторой локальной сети. С другой стороны, можно просто объединить несколько серверов в локальную частную сеть и добавлять туда клиента для большей безопасности.

Если выбирать между частной и общей сетью, первый вариант всегда предпочтительнее. При этом стоит помнить, что пользователи дата-центра связаны одной сетью, вы должны максимально избавить себя от рисков, приняв дополнительные меры для безопасной связи между серверами.

Использование VPN, по сути, способ создать частную сеть, которую могут видеть только ваши серверы. Связь будет полностью приватной и безопасной. Кроме того, VPN можно настроить для отдельных приложений и служб, чтобы их трафик проходил через виртуальный интерфейс. Таким образом, можно обезопасить процессы внутри компании, открыв общественный доступ только для клиентской стороны, а внутреннею часть работы сервера скрыть VPN.

Следующим способом защиты на стороне сервера является PKI и SSL/TLS шифрование.

Инфраструктура открытых ключей (PKI) – это совокупность систем, которые предназначены для создания, управления и проверки сертификатов для идентификации лиц и шифрования передаваемых данных. После аутентификации они также могут быть использованы для шифрованной связи.

Создание центра сертификации и управления сертификатами для серверов позволяет каждому в пределах серверной инфраструктуры шифровать свой трафик и использовать проверки идентичности других пользователей. PKI поможет предотвратить атаки посредника (man-in-the-middle), когда

злоумышленник имитирует поведение сервера в вашей инфраструктуре, чтобы перехватить трафик или подменить сообщение.

Каждый сервер можно настроить таким образом, чтобы все участники проходили аутентификацию через удостоверяющий центр, который создает пару ключей: открытый и закрытый. Удостоверяющий центр или УЦ может раздавать открытые ключи всем участникам, у которых низкий уровень доверия друг к другу, но высокий к УЦ. Только последний может подтвердить принадлежность открытого ключа к его владельцу.

Если вы используете приложения и протоколы, которые поддерживают TLS/SSL шифрование, то это способ снизить расходы на VPN (в которых часто используют SSL).

Конечно, на этом методы защиты сервера приложений не заканчиваются, их существует еще достаточно много, однако их подробное изучение выходит за рамки данного курса.

В завершении этой темы можно дать один совет разработчикам для формирования защищенного серверного пространства: используйте аудит безопасности.

Аудит безопасности заключается в заключении договора с компанией или частными лицами, которые по факту являются «белыми» или «этичными» хакерами. Они получают контракт на исследование безопасности ваших продуктов и пытаются всеми силами его «взломать» при помощи различных подходов и инструментов.

Результат своих исследований они оформляют в специальный отчет, где также размещают рекомендации по повышению уровня защищенности сервера.

Получив этот отчет можно выдать задание программистам и системным администраторам на закрытие этих «дыр» безопасности и существенно повысить надежность серверного и даже клиентского программного обеспечения.

Следуя практике, они чаще всего советуют убрать следующие виды уязвимостей:

1) SQL Injection (один из распространённых способов взлома сайтов и программ, работающих с базами данных, основанный на внедрении в запрос произвольного SQL-кода.).

2) Cross-site scripting (тип атаки на веб-системы, заключающийся во внедрении в выдаваемую веб-системой страницу вредоносного кода, который будет выполнен на компьютере пользователя при открытии им этой страницы и взаимодействии этого кода с веб-сервером злоумышленника).

3) Data manipulation (манипуляции с контролируемыми параметрами пользователя, что может привести к мошенническим действиям при покупке товара (фрод), изменение стоимости, подмена данных и так далее).

4) CSRF (это вид атак на посетителей веб-сайтов, использующий недостатки протокола HTTP, заключающийся в следующем – если жертва заходит на сайт, созданный злоумышленником, от её лица тайно отправляется запрос на другой сервер, осуществляющий некую вредоносную операцию.).

5) Cleartext submission of passwords (передача паролей пользователей в открытом виде, что позволяет их перехватывать и использовать в различных сценариях).

6) Sensitive information disclosure (предоставляется конфиденциальная информация субъекту, который явно не уполномочен иметь доступ к этой информации).

7) Weak passwords store (хранение паролей в незащищенном формате).

8) Full path disclosure (позволить злоумышленнику увидеть полный путь к файлу и узнать абсолютные пути файлов проекта на сервере, например: /home/user/docs/file.ext).

9) Cookie without HttpOnly flag set (куки являются хранилищем данных, а HTTPonly не доступны из JavaScript через свойства Document.cookie API, что помогает избежать межсайтового скриптинга).

10) Insecure authentication (механизмы аутентификации могут быть проэксплуатированы посредством автоматических атак с использованием доступных или специально созданных для этого инструментов).

Все перечисленные (и дополнительные уязвимости) снабжаются четкими инструкциями по устранению, ссылками на стандарты (например OWASP, CWE, CPE, CVE) и описания.

Кроме того, аудиторы часто добавляют в отчеты специальные меры по повышению защищенности программного обеспечения, например выполнение:

1. физического контроля доступа в помещения, наблюдение за помещениями;
2. настройки аппаратного обеспечения информационной системы;
3. настройки сетевого обеспечения информационной системы;
4. настройки системного программного обеспечения;
5. настройки интрасетевых устройств и коммутационных устройств;
6. настройки организационного обеспечения информационной системы;
7. настройки нормативного обеспечения;
8. контроля корпоративных данных;
9. модернизации внутренней IT-структуры компании;
10. настройки серверной IT-структуры;
11. стратегических рекомендаций по организации серверной структуры информационной системы;
12. распределение прав доступа к ресурсам и так далее.