Содержание:

Введение

Актуальность темы исследования заключается в том, что мы живем во время стремительного прогресса и с каждым днем накапливается все больший объем информации и нам нужно его каталогизировать и выдавать пользователю в удобном для него виде. Так же из-за появления большого объема информации, программистам нужно чаще обновлять свой программный код, для добавления новых функций или поддержания старых.
Для этих целей и используется архитектура, когда модули приложения не зависят напрямую друг от друга и находятся в определенных местах, что оптимизирует добавление кода, его тестируемость и сохраняет деньги бизнеса и время программистов в будущем. Так же используется различные типы программирования и диаграммы. Так же программирование невозможно без тестирование конечного функционала.

Объект исследования - анализ методов современного программирования.

Предмет исследования - методы современного программирования и ее типы.

Цель работы - изучить особенности методов современного программирования.

Задачи исследования:

1. Изучить и проанализировать литературу по теме.
2. Структурировать знания по данной тематике, выделив основные методы
3. Изучить особенности различных методов
4. Изучить, как методы программирования влияют на оптимизацию программного кода
5. Обобщить полученные в ходе работы знания

Глава 1. Методики программирования

Методология разработки программного обеспечения - совокупность методов, применяемых на различных стадиях жизненного цикла программного обеспечения

и имеющих общий философский подход.

Методики программирования:

1. Методы систематического программирования

- структурный
- объектно-ориентированный
- UML-метод
- компонентный
- аспектно-ориентированный
- генерирующий
- агентны

1. Методы теоретического программирования

- алгебраическое
- экспликативное
- алгоритмическое программирование

1.1 Методы систематического программирования

1.1.1 Структурный подход

Сущность структурного подхода к разработке ПС заключается в разбиении целостной системы на автоматизируемые функции, которые делятся на подфункции, на задачи и так далее. Процесс декомпозиции продолжается до определения конкретных процедур. При этом автоматизируемая система сохраняет целостное представление, в котором все составляющие компоненты взаимосвязаны.

Общие принципы:

- разделение общей задачи на подзадачи для облегченного понимания
- организация составных частей задачи в древовидные структуры с добавлением новых деталей на каждом уровне.

Основные принципы:

- абстрагирование, выделение важных элементов системы от дополнительных
- формализация, общее свойство для решения задачи
- непротиворечивость, согласованность всех элементов системы
- иерархическая структуризация данных.

1.1.2 Объектно-ориентированный подход

Объектно-ориентированный подход - методология разработки, в рамках которой объекты реального мира представляются *классами*, у каждого из которых есть своя функция. Класс служит шаблоном для описания всех атрибутов и поведений, связанных с объектами данного класса.

Приложения содержит взаимодействующие друг с другом объекты, у которых есть свои локальные состояния и методы для взаимодействия с другими объектами. Обычно объекты имеют модификаторы доступа, которые помогают объекту не допустить изменений из вне, в целях безопасности.

Основные принципы:

- абстракция - выделение общих абстрактных свойств без их конкретной реализации
- инкапсуляция - сокрытие реализации. Применяется, чтобы внешние объекты не могли менять значения полей или поведение объекта, без его ведома.
- наследование - позволяет описать новый класс на основе уже существующего. Класс, от которого наследуются называется родителем, новый - потомком. Позволяет не писать заново новую функциональность, а часть кода использовать из ранее написанных классов.
- полиморфизм - многообразие форм. Позволяет использовать объекты с одинаковым интерфейсом без знания конкретного типа.

1.1.3 UML - программирование

UML (United Modeling Language) – язык графического описания для объектов реального мира в виде различных диаграмм.

Модель UML - это совокупность диаграмм, которые визуализируют основные элементы структуры системы.

UML – необходима, чтобы посмотреть на задачу с разных точек зрения и чтобы программистам было легче понять способ реализации задачи.

Что обеспечивает UML:

- иерархическое описание сложной системы путем выделения пакетов;
- формализацию функциональных требований к системе с помощью аппарата вариантов использования;
- детализацию требований к системе путем построения диаграмм деятельностей и сценариев;
- выделение классов данных и построение концептуальной модели данных в виде диаграмм классов;
- выделение классов, описывающих пользовательский интерфейс, и создание схемы навигации экранов;
- описание процессов взаимодействия объектов при выполнении системных функций;
- описание поведения объектов в виде диаграмм деятельностей и состояний;
- описание программных компонент и их взаимодействия через интерфейсы;
- описание физической архитектуры системы.

Основные типы UML:

- Диаграммы состояний (State diagrams);
- Диаграммы деятельностей (Activity diagrams);
- Диаграммы объектов (Object diagrams);
- Диаграммы последовательностей (Sequence diagrams);
- Диаграммы взаимодействия (Collaboration diagrams);
- Одна из диаграмм, которая довольно часто используется на практике - это **диаграмма классов**, которая описываю структуру классов, их атрибуты и зависимости между классами.

1.1.4 Компонентное проектирование

Компонентное программирование является наиболее производительным, так как использует готовые компоненты при создании сложных систем.

Компонентный подход дополняет и расширяет существующие подходы в программировании, например, ООП. Объекты рассматриваются на логическом

уровне проектирования ПС, а компоненты - это физическая реализация объектов.

Типы компонентных структур. Компонент расширяет паттерн - абстракция, который содержит описание взаимодействия и роли для каждого объекта.

Композиция компонентов может быть следующих типов:

- композиция компонент-компонент обеспечивает взаимодействие компонентов через интерфейс на уровне приложения
- композиция каркас-компонент обеспечивает взаимодействие каркаса с компонентами, при котором каркас управляет ресурсами компонентов и их интерфейсами на системном уровне
- композиция компонент-каркас обеспечивает взаимодействие компонента с каркасом по типу "черного ящика", в видимой части которого находится описание файла для развертывания и выполнения определенной функции на сервисном уровне
- композиция каркас-каркас обеспечивает взаимодействие каркасов, каждый из которых может разворачиваться в гетерогенной среде и разрешать компонентам, входящим в каркас, взаимодействовать через их интерфейсы на сетевом уровне

1.1.5 Аспектно - ориентированное программирование

Аспектно - ориентированное программирование - это методология программирования, при которой создаются гибкие к изменениям ПС, благодаря добавлению новых функций, которые обеспечивают безопасность и взаимодействие компонентов с другими системами, так же для синхронизации одновременного доступа к данным.

Основные этапы:

- разделение задач с условием многоразового применения модулей и выделенных свойств, такие как, параллельность, синхронность, безопасность.
- определение точек соприкосновения одних элементов с другими
- разработка фильтров
- создание объектной или компонентной модели, и ее дополнение.
- отладка модулей по-отдельности, так и приложения в целом

1.1.6 Генерирующее программирование

Генерирующее программирование – методология программирования, при которой создаются семейства приложений из отдельных элементов, фабрик и тому подобное. По сути, это ООП, который дополнен многоразовыми элементами, изменяемыми свойствами, взаимодействиями между элементами.

Основные этапы:

- анализ и выявление объектов и отношений между ними
- определение области действий объектов
- определение общих характеристик и построение модели
- создание основы для производства программных членов семейства независимо от их реализации
- подбор и подготовка компонентов для многоразового применения

1.1.7 Агентное программирование

Агентное программирование - методология программирования, при которой программа способна сама управлять своими действиями в информационной среде и получать результаты выполнения задачи, изменение самой среды.

Основные свойства агента:

- автономность - это способность действовать без внешнего воздействия
- реактивность - это способность реагировать на изменения данных и среды и воспринимать их
- активность - это способность самостоятельно ставить задачи и выполнять заданные действия для достижения
- способность к взаимодействию с другими агентами

Основные задачи:

- самостоятельная работа и контроль своих действий
- взаимодействие с другими агентами
- изменение поведения в зависимости от состояния внешней среды
- получение результата

2.1. Теоретическое программирование

2.1.1 Алгебраическое программирование

Алгебраическое программирование - это конструирование программ с использованием алгебраических преобразований и функций интеллектуальных агентов. В основе математического аппарата используется алгебра языка действий и понятие транзитивной системы в качестве механизма определения поведения систем и механизмов ее эквивалентности. В качестве понятий в общем случае могут быть компоненты, программы и их спецификации, объекты, взаимодействующие друг с другом и со средой их существования.

Теория АП обеспечивает создание математической информационной среды с универсальными математическими конструкциями, вычислительными механизмами, учитывающими особенности разработки ПС и функционирования.

Алгебраическое программирование концентрирует внимание на задачах интеллектуализации и аспектах поведения агентов в их среде.

2.1.2 Экспликативное программирование

Экспликативное программирование - ориентировано на разработку теории дескриптивных и декларативных программных формализмов, адекватных моделям структур данных, программ и средств конструирования из них программ. Теоретическую основу ЭП составляют логика, конструктивная математика, информатика, композиционное программирование и классическая теории алгоритмов. Для изображения алгоритмов программ используются алгоритмические языки и методы программирования: функциональное, логическое, структурное, денотационное

Основные принципы ЭП:

- принцип развития понятия программы в абстрактном представлении и постепенной ее конкретизации с помощью экспликаций
- принцип прагматичности - решение задач пользователя

- принцип адекватности - абстрактное построение программ и реализацию задачи с учетом информационности данных и апликативности. Программа рассматривается как функция, вырабатывающая выходные данные на основе входных данных.Функция - это объект, которому сопоставляется денотат имени функции с помощью отношения именования
- принцип дескриптивности позволяет трактовать программу как сложные дескрипции, построенные из более простых и композиций отображения входных данных в результаты на основе принципа вычислимости

2.1.3 Алгоритмическое программирование

Алгоритмическое программирование - структурная схемотология построения последовательных и параллельных программ с использованием алгебраических преобразований и стандартных форм описания логических и операторных выражений.

Основные понятия:

- операции над множествами, булевы операции, предикаты, функции и операторы
- бинарные и n-арные отношения, эквивалентность, частично и полностью упорядоченные множества
- граф-схемы и операции над графовыми структурами
- операции сигнатуры САА, аксиомы и правила вывода свойств программ на основе сверточной, разверточной и комбинированной стратегий
- символьная обработка и методы синтаксического анализа программ

Глава 2. Оптимизации программного кода

Оптимизация программного кода - различные методы преобразования кода ради улучшения его характеристик и повышения эффективности.

Основные принципы оптимизации кода:

- Естественность – код должен быть легко читабельным
- Производительность – увеличение быстродействия приложения
- Затраченное время – отладка должна занимать небольшой период времени

2.1. Основные принципы оптимизации кода

2.1.1. Выявление узких мест программы

Сначала выявляются узкие места программы. В первую очередь нужно обращать внимание на куски кода, которые выполняются регулярно в процессе работы.

Есть смысл оптимизировать код, если производительности или качества кода действительно не хватает для выполнения поставленной задачи, так как если все работает прекрасно, то оптимизация - лишняя трата ресурсов.

2.1.2. Методы оптимизации программ

- свертывание - оптимизация в компиляторах
- распространение констант – поиск постоянных значений и замена на эти значения. Например, было вместо вычисления в переменно, сразу будет результат.
- распространение копий – избавление от промежуточных значений. Например, от ненужных переменных.
- устранение – удаление неиспользуемого кода программы

Хороший оптимизирующий компилятор может повысить быстродействие кода на 40 и более процентов, тогда как многие из методик, используемых программистом вручную, только на 15-30%.

Необходимо проверить код на наличие устаревших или неиспользуемых фрагментов (устранение). Они могут использоваться и замедлять работу системы.

Необходимо изучить, где приложение работает медленнее всего или из-за какого участка кода может вылетать на определенных устройствах. Необходимо изучить код на ошибки и устранить их.

Иногда необходимо проверить настройки системы, возможно они настроены по стандарту или неверно и нужны некоторые корректировки.

Если используется язык Java, необходимо изучить и настроить работу виртуальной машины, это может увеличить производительность не только приложения, но и

всей системы.

Для увеличения скорости работы приложения можно использовать подход сокращения ненужного кода (распространение копий). Вам нужно проанализировать, какие реализованные функции вашего приложения почти не используются пользователями и возможно стоит от них отказаться в пользу производительности.

2.1.3. Параллельное программирование

Процесс - это совокупность кода и данных, разделяющих общее виртуальное адресное пространство. Одна программа состоит из одного процесса. Процессы изолированы друг от друга и не имеют общий доступ к памяти друг друга.

Поток - это одна единица исполнения кода. Каждый поток последовательно выполняет инструкции процесса, независимо от других (при однопоточном программировании).

Параллельное программирование - адаптация алгоритмов, под программы для компьютерных систем с параллельной архитектурой. Обычно применяется при использовании многопроцессорных или многопоточных систем.

Каждый процесс имеет хотя бы один *главный* выполняющийся поток, с которого начинается выполнение программы.

При параллельном программировании увеличивается общая производительность работы программы, так как некоторые действия производятся параллельно, не мешая друг другу. Если нужно синхронизировать данные потоков, то используются специальные команды. Так же бывают случаи, когда потоки одновременно запрашивают данные у метода, тогда нужно выставить приоритет, кто будет иметь доступ к нему раньше, чтобы не было проблем с доступом. Так же бывают случаи, когда может случиться дедлок.

Дедлок - взаимная блокировка, которая происходит из-за цикличной зависимости потоков друг от друга.

Тема многопоточности довольно распространена на практике, так человечество накопило огромный пласт информации, которую нужно постоянно хранить и обрабатывать, и однопоточный приложений уже не справляются с этой задачей.

Ленивые или отложенные вычисления - метод заключается в том, что все расчеты откладываются до тех пор, пока не будет затребован их результат.

Это позволит снизить общий объем вычислений, так как ненужные операции не будут выполняться, когда не нужно пользователю.

Приближение - метод замены точного алгоритма на его приближенные значения, из-за чего теряется часть точности, зато мы получаем рост в производительности. Например, алгоритмы для вычисления координат GPS.

Нужно с умом подходить к выбору алгоритмов и конкретной метрике, однако сам метод может помочь с увеличением производительности.

Программа может медленно работать из-за того, что много времени занимает проверка типов, которая занимает дополнительное время. Чтобы избежать этого эффекта, можно применять фрагменты кода или модули, написанные на других языках. Однако это может создать уязвимости в безопасности, поэтому нужно тщательно протестировать код после.

2.2 Архитектура программных систем.

Архитектура - это организация системы программного обеспечения. Это набор компонентов системы, которые взаимодействую между собой через интерфейсы. Внутренние слои не должны знать о внешних.

Основные принципы построения архитектуры:

• Архитектура должна быть независима от различных фреймворков

• Система должна быть протестирована

• Независимость архитектуры от всего

Для построения архитектуры ПС, была изобретена четырехуровневая схема, где каждая окружность является определенным компонентом системы.

Схема состоит из следующих сущностей (начиная с внутреннего слоя):

• Бизнес-объекты (классы моделей с методами)

• Сценарии взаимодействия (основные методы для работы с системой)

• Слой представления (преобразование данных бизнес-объектов или сценариев взаимодействия в формат для работы системы)

• Фреймворки (конкретные инструменты для решения задач)

Так же для построения грамотной архитектуры используются паттерны проектирования и архитектурные паттерны.

Паттерн проектирования - это часто встречающееся решение определённой проблемы при проектировании архитектуры программ, что позволяет не «изобретать велосипед», а использовать готовый шаблон, проверенный многими специалистами.

Паттерн представляет общую концепцию решения проблемы. Для решения конкретной проблемы нужно использовать паттерн на конкретном языке программирования.

Существует 3 типа паттернов:

• Порождающие - отвечают за создание новых объектов или семейств объектов.

• Структурные - отвечают за построение удобных в поддержке иерархий классов.

• Поведенческие - решают задачи эффективного и безопасного взаимодействия между объектами программы.

Архитектурные паттерны - повторяемая архитектурная конструкция, представляющая собой решение проблемы проектирования, которая используют часть паттернов проектирования для создания архитектуры под определенную область деятельности (Рабочие компьютеры, Веб-сайты, Мобильные приложения), либо в рамках одной области.

Основные архитектурные паттерны:

• MVC (Model - View - Controller)

• MVP (Model – View - Presenter)

• MVVM (Model - View - ViewModel)

• MVI (Model - View - Intent)

• VIPER (View - Interactor - Presenter - Entity - Routing)

2.2.1 MVC (Model - View - Controller)

Model - предоставляет данные и реагирует на команды контроллера, изменяя своё состояние.

View - отвечает за отображение данных на экране пользователя.

Controller - интерпретирует действия пользователя, оповещая Model о необходимости изменений.

2.2.2 MVP (Model-View-Presenter)

Model - предоставляет данные и реагирует на команды Presentor’a.

View - отвечает за отображение данных Model пользователю, обращается к Presenter за обновлениями.

Presenter – является посредником между Model и View

Если пользователь нажал какой-либо элемент на экране устройства, то View сообщает об этом Presentor'у. View не просит Presenter загружать данные, она лишь уведомляет Presenter, что пользователь нажал элемент на экране. Дальше Presenter вызывает нужный метод, который должен сработать при нажатии на конкретный элемент. Если нужно, он так же запрашивает данные у Model и передает их во View, для отображения на экране, например, обновление счетчика при нажатии на кнопку на экране.

Если пользователь нажал какой-либо элемент на экране устройства, то View сообщает об этом Presentor'у.

View не просит Presenter загружать данные, она лишь уведомляет Presenter, что пользователь нажал элемент на экране. Дальше Presenter вызывает нужный метод, который должен сработать при нажатии на конкретный элемент. Если нужно, он так же запрашивает данные у Model и передает их уже View, для отображения на экране, например, обновление счетчика при нажатии на кнопку на экране.

Если View отображает данные из базы данных, то Model - это база данных. Presenter может подписаться на уведомления Model об обновлении. (паттерн: Наблюдатель).

Когда происходит обновление данных в БД, Model оповещает об этих изменениях Presenter. Presenter получает эти изменения и передает во View.

Presenter – это логика приложения, вынесенная из View в отдельный класс. View – отображает данные и взаимодействует с пользователем.

Если понадобиться создать еще несколько новых View, то можно будет воспользоваться готовым Presenter’ом.

Если нужно изменить саму логику приложения, то не нужно будет менять данные во View, вы меняете код в Presenter, либо можно создать новый Presenter и использовать его.

2.2.3 MVVM (Model - View - ViewModel)

Model - предоставляет данные и реагирует на команды ViewModel.

View - Представляет собой интерфейс, с которым взаимодействует пользователь на экране своего устройства.

ViewModel - объект, в котором описывается логика поведения View в зависимости от результата работы Model.

Компоненты MVVM не знаю друг о друге напрямую. Эти компоненты взаимодействуют между собой за счет механизма связывания данных (Bindings), который реализуется средствами той или иной системы.

При этом изменение данных во ViewModel автоматически меняет данные, отображаемые во View. Любое событие или изменение данных во View изменяет данные во ViewModel. Это позволяет держать эти компоненты очень слабо связными, что удобно при тестировании.

Data Binding - это фреймворк от Google, который позволяет выполнить связывание Java-кода и xml-файлов с помощью Binding-объекта. Можно полностью избавиться от работы со View в Java-коде.

С помощью Data Binding View в xml-разметке можно задать любые свойства, что скрывает некоторые детали реализации, однако это переполняет xml файл и не очень удобно при большом количестве Java-классов и переменных в xml.

2.2.4 MVI (Model - View - Intent)

Intent - метод, который принимает входные данные от пользователя (например, события пользовательского интерфейса) и переводит в то, что будет передано как параметр метода Model. Это может быть строка для установки значения Model или, объект.

Model - метод, который использует выходные данные из метода Intent в качестве входных данных для работы с Model. Результат работы - новая Model (с измененным состоянием). Все данные должны были неизменяемыми. Метод Model - часть кода, который создает новый объект модели.

View - Представляет собой интерфейс, с которым взаимодействует пользователь на экране своего устройства.

2.2.5 VIPER (View - Interactor - Presenter - Entity - Router)

View - Представляет собой интерфейс, с которым взаимодействует пользователь на экране своего устройства. Обращается к Presenter за обновлениями.

View ждет действия Presenter'а, чтобы передать содержания для выведения на экран.

Interactor - Включает в себя бизнес-логику для управления объектами данных (Entity). Задача выполняется в Interactor'е, независимо от View

Presenter - содержит логику управления. Определяет запрос, поступающий со View, и решает, куда отправить его дальше в Router для изменения окна View или передает метод в Interactor.

Entity - это объекты данных, которыми управляет Interactor.

Router - осуществляет переключения между экранами приложения.

Data Store - отвечает за доставку Entity в Interactor. Entity не знают о Data Store.

Заключение

Мы рассмотрели основные методы современного программирования, которые применяются на практике, так же были рассмотрены методологии программирования и методы оптимизации кода. Из чего можно сделать вывод, что построение архитектуры ПО позволяет программистам экономить большое количество часов в будущем, при добавлении нового функционала, а бизнесу сохранить свои деньги, не переписывая постоянно основной костяк приложения, каким бы оно ни было.

Необходимо заранее продумывать тип архитектуры, которая будет применена в ПО, так как от типа ПО так же зависит и тип архитектуры, например, не все типы подходят для создания веб-сайта, или для мобильного приложения.

Так же выбранная архитектура зависит от сроков разработки, если компания хочет создать ПО, которое планирует долго поддерживать, то нужно заранее обдумывать последствия выбранной архитектуры и есть ли смысл применять ее в данном случае, так как иногда рынок выпускает MVP (минимально жизнеспособный продукт), где сложная архитектура, наподобие VIPER может усложнить все этапы разработки продукта, или не применять архитектуру вовсе, например на хакатонах про программированию, где обычно дается 20 часов времени на разработку своего ПО.

Помимо различных типов архитектур, разработчики должны знать и применять на практике методологии разработки ПО, что позволит в будущем принимать более взвешенные решения и тратить меньше времени на написание кода и больше на то, как лучше решить поставленную задачу и какой метод или инструмент выбрать.

Из рассмотренных методологий, мы вынесли, что помимо написание самого кода, так же существует и математическая сторона решения задач, что может довольно сильно оптимизировать конечный продукт.

Так же нужно помнить и о том, что нужно иметь четкую и конкретную цель, для чего и для кого создается программный продукт, так как от этого зависит выбор инструментов для ее решения. Так как не имеет значение, сколько работников будет работать внутри компании, какой квалификации и какая архитектура будет выбрана, если конечная цель неверна.

Так же существует несколько типов методологии разработки, таких как Agile и подобных ему, от которых зависит выбор инструментов.

Так же нельзя забывать и про тестирование кода, что поможет в будущем уменьшить количество ошибок в ПО и тратить меньше времени на переписывание кода и отлавливание проблем, вместо добавления нового функционала или написания нового ПО.

Список литературы

1. Перемитина Т. О. - Управление качеством программных систем: учебное пособие - Томск: Эль Контент, 2011. - 228 с.
2. Мартин Р. - Чистая архитектура. Искусство разработки программного обеспечения. - СПб.: Питер, 2018. — 352 с
3. Лекция 4 по архитектуре андроид приложения. Clean Architecture [Электронный ресурс] / URL: https://www.fandroid.info/lektsiya-4-po-arhitekture-android-prilozheniya-clean-arcitecture/
4. Что такое архитектура программного обеспечения? [Электронный ресурс] / URL: https://www.ibm.com/developerworks/ru/library/eeles/index.html
5. Методология программирования [Электронный ресурс] / URL: https://ru.wikipedia.org/wiki/Методология_программирования
6. Оптимизация программного кода [Электронный ресурс] / URL: https://techrocks.ru/2019/01/25/code-optimization-tips/
7. Современные методы программирования [Электронный ресурс] / URL: http://www.ispras.ru/lavrishcheva/textbooks/sovremennye_metody_programmirovanija.pdf
8. Лекция 6: Прикладные и теоретические методы программирования [Электронный ресурс] / URL: https://www.intuit.ru/studies/courses/2190/237/lecture/6126?page=1#sect2
9. Android и архитектура [Электронный ресурс] / URL: https://habr.com/ru/post/328962/
10. Архитектура Android приложений [Электронный ресурс] / URL: https://habr.com/ru/post/278815/
11. VIPER Swift [Электронный ресурс] / URL: http://bytepace.com/ru/blog/viper-swift
12. Реализовываем MVVM в Android [Электронный ресурс] / URL: https://stfalcon.com/ru/blog/post/android-mvvm