Д. А. Талько

ВЛИЯНИЕ ВЫБОРА UI-ПАРАДИГМЫ (XML/COMPOSE) НА ЭФФЕКТИВНОСТЬ ПРОЕКТИРОВАНИЯ И РАЗРАБОТКИ КЛИЕНТСКИХ ПРИЛОЖЕНИЙ

В работе проведён комплексный сравнительный анализ влияния выбора UI-парадигмы (XML и Jetpack Compose) на эффективность проектирования и разработки клиентских приложений для ОС Android. Рассмотрены исторические предпосылки смены парадигмы с императивной (XML) на декларативную (Compose). Теоретически обоснованы различия в подходах к управлению состоянием, связанности компонентов и когнитивной нагрузке на разработчика. Экспериментально сопоставлены ключевые метрики разработки: объём кодовой базы, время холодного старта, потребление памяти на сложных экранах и время отрисовки кадров. Установлено, что применение Jetpack Compose позволяет сократить объём UI-кода, снизить использование памяти и улучшить предсказуемость потока данных за счёт нативной поддержки Unidirectional Data Flow. Предложены стратегии гибридной миграции с XML на Compose для существующих проектов.

Введение

В современной индустрии разработки программного обеспечения проектирование интерфейсов пользователя (UI) перестало быть чисто эстетической задачей, превратившись в сложный инженерный процесс, определяющий архитектурную устойчивость и производительность систем обработки информации. Выбор между традиционной императивной моделью на основе Extensible Markup Language (XML) и современной декларативной парадигмой Jetpack Compose представляет собой фундаментальную дилемму для проектировщиков информационных систем в 2024–2026 годах. Данная работа представляет собой комплексный сравнительный анализ этих двух подходов, рассматривая их влияние на метрики разработки, производительность среды выполнения и долгосрочную стоимость владения кодовой базой.

I. Сравнительный анализ UI-парадигм

На протяжении более десяти лет стандартом проектирования клиентских приложений для ОС Android являлась система View, основанная на XML-разметке. В этой модели структура интерфейса описывалась в статических файлах, а динамическое поведение и управление состоянием возлагалось на императивный код на языках Java или Kotlin. Такая архитектура следовала принципу разделения ответственности, однако по мере роста сложности приложений она привела к возникновению ряда системных проблем, включая избыточность кода, сложность синхронизации состояний и трудности при отладке иерархий представлений [1].

Однако стало очевидным, что традиционная модель достигла предела эффективности. Проблема заключалась в необходимости вручную манипулировать каждым элементом UI при изменении данных, что требовало использования таких инструментов, как findViewById или ViewBinding [1]. Эти механизмы создавали жесткую связность между разметкой и логикой, увеличивая вероятность ошибок при обновлении интерфейса. В ответ на эти вызовы компания Google представила Jetpack Compose – декларативный инструментарий, который кардинально меняет способ построения UI, переводя его из категории «набора манипулируемых объектов» в категорию «функции от состояния» [2].

Фундаментальное различие между рассматриваемыми технологиями лежит в области теории программирования. Императивный подход, характерный для XML-системы, требует от разработчика пошагового описания процесса изменения UI. Если информационная система получает обновление данных, разработчик должен найти нужный виджет и вызвать метод обновления его свойств, например, textView.setText(data). Это создает ситуацию, в которой состояние UI распределено по множеству компонентов, что затрудняет поддержание целостности данных.

Декларативный подход Jetpack Compose базируется на концепции неизменяемости и реактивности. В этой модели разработчик лишь описывает, как должен выглядеть экран для конкретного состояния данных. Когда состояние изменяется, фреймворк автоматически пересчитывает необходимые части UI и обновляет их. Этот процесс, называемый рекомпозицией, позволяет избежать множества ошибок, связанных с забытыми обновлениями или некорректным состоянием виджетов [2].

С точки зрения теории когнитивной нагрузки, императивный подход заставляет разработчика удерживать в оперативной памяти ментальную модель не только текущего состояния интерфейса, но и всей последовательности действий, необходимых для перехода между состояниями. Это особенно критично в больших командах, где разные разработчики работают над разными частями этой временной цепочки. Декларативная парадигма, напротив, позволяет мыслить категориями фактов и зависимостей, что фундаментально снижает сложность отладки и доказательства корректности кода, перенося фокус с управления побочными эффектами на описание структуры.

Сопоставление ключевых архитектурных аспектов XML и Jetpack Compose позволяет оценить их влияние на структуру информационных систем. Прежде всего, технологии различаются по своей философии: XML-Based View System использует императивный подход, отвечая на вопрос «как делать», тогда как Jetpack Compose основывается на декларативной пара-

дигме, описывая «что делать». Это фундаментальное различие влечёт за собой и другие расхождения. С точки зрения связанности компонентов XML-система характеризуется высокой связностью из-за необходимости маппинга идентификаторов, тогда как Compose обеспечивает низкую связанность благодаря размещению логики и пользовательского интерфейса в одном языке. Управление состоянием в традиционной системе является распределённым и часто требует ручного вмешательства, тогда как в Compose оно централизованное и автоматическое. Жизненный цикл компонентов в XML-системе сложный и включает Activity, Fragment и View, а в Compose он упрощён до области рекомпозиции. Динамичность интерфейсов в XML ограничена и требует дополнительного вмешательства разработчика, в то время как Compose обеспечивает высокую динамичность за счёт естественного поведения функций. Наконец, переиспользование кода в XML достигается через механизмы include и создание custom Views, а в Compose – через композицию функций.

Такое разграничение подчеркивает, что Compose не просто заменяет XML как язык разметки, а заменяет всю систему управления представлениями на более предсказуемую и масштабируемую модель [2].

Эффективность разработки клиентских приложений измеряется не только скоростью написания первой версии, но и простотой последующей поддержки, тестирования и модификации.

Одним из наиболее заметных преимуществ Jetpack Compose является радикальное сокращение объема кодовой базы. Исследования показывают, что при миграции с XML на Compose количество строк кода в UI-слое может уменьшиться значительно. Команда Play Store, например, обнаружила, что написание UI требует иногда до 50% меньше кода, что напрямую повышает производительность и поддерживаемость [3]. Это сокращение достигается за счет отсутствия необходимости в: XML-файлах разметки и стилей; классах-адаптерах для списков (RecyclerView); описании связей через идентификаторы ресурсов.

В результате разработчик может сосредоточиться на бизнес-логике. Проектирование сложных анимаций также становится значительно проще: в Compose они описываются несколькими строками декларативного кода [1].

При выборе стандарта программирования решающее значение имеют показатели производительности приложения у конечного пользователя.

Традиционная система XML страдает от накладных расходов на этапе выполнения, связанных с «инфляцией» – процессом парсинга XML-файлов и создания объектов в памяти. Compose полностью исключает этот этап, так как его макеты компилируются непосредственно в исполняемый байт-код. Согласно некоторым измерениям, холодный старт приложений на базе Compose может быть быстрее по сравнению с XML-аналогами, хотя здесь многое зависит от конкретных условий [1].

С точки зрения использования памяти XML-системы часто оказываются более эффективными для простых интерфейсов, в то время как Compose требует загрузки рантайма, что несколько увеличивает базовое потребление памяти, однако при проектировании сложных интерфейсов всё происходит наоборот.

Сравнение метрик производительности XML-верстки и Jetpack Compose наглядно демонстрирует это различие. По времени холодного старта XML-система показывает результат 800–1200 мс, тогда как Jetpack Compose значительно быстрее – 400–700 мс. Что касается использования памяти на сложных экранах, XML потребляет 80–100 МБ, а Compose – лишь 50–70 МБ. При отрисовке кадров в сложных списках XML тратит 12–16 мс на кадр, тогда как Compose укладывается в 8–10 мс на кадр.

Эти данные показывают, что Compose не только обеспечивает более плавную работу, но и потребляет значительно меньше памяти на сложных экранах, что делает его эффективным выбором для динамических интерфейсов [1].

II. Управление состоянием и архитектурные паттерны

Одной из центральных задач проектирования систем обработки информации является обеспечение предсказуемого потока данных.

Jetpack Compose нативно поддерживает паттерн Unidirectional Data Flow (UDF). Состояние «спускается» вниз от источника истины (ViewModel) к компонентам, а события «поднимаются» вверх [4]. Это разделение ответственности делает систему более тестируемой. Для управления состоянием в Compose используются:

– Snapshot System: механизм отслеживания изменений для вызова рекомпозиции [4];

– State Hoisting: вынос состояния из компонента для повышения его переиспользования [4];

– Derived State: оптимизация для вычисления производных значений только при изменении исходных данных [4].

С точки зрения эмпирической программной инженерии, такой подход значительно упрощает формальную верификацию и тестирование. Состояние хранится в одном месте (например, ViewModel), и его можно тестировать как чистую функцию: на входе «состояние X», на выходе должен быть «UI с такими-то свойствами». В императивной XML-системе состояние «размазано» по десяткам View-объектов, что вынуждает писать сложные и хрупкие интеграционные тесты вместо простых модульных. Compose позволяет перейти к более надежному тестированию логики отображения.

Важным преимуществом Compose является его способность сосуществовать с XML-кодом. Разработчики могут использовать ComposeView внутри XML-макетов или вставлять традиционные View в Compose-функции через AndroidView. Процесс перехода требует изменения ментальной модели. Исследования подтверждают, что основной вызов миграции

заключается не в изучении синтаксиса, а в понимании механизмов рекомпозиции и жизненного цикла декларативного интерфейса [5]. Практические советы включают постепенную замену сложных RecyclerView на LazyColumn, что дает немедленный выигрыш в читаемости кода и стабильности приложения.

III. Заключение

Таким образом, выбор между XML и Jetpack Compose в 2026 году является стратегическим решением, которое должно учитывать не только сиюминутную выгоду, но и долгосрочную эволюцию продукта.

Для новых проектов рекомендуется использовать Jetpack Compose как стандарт де-факто. Для существующих систем на основе XML наиболее рациональной стратегией является гибридный подход с постепенным переписыванием наиболее сложных и динамических частей интерфейса. Выбор должен быть осознанным: XML остается надежным «рабочим инструментом» для простых и статичных интерфейсов, в то время как Compose – это инвестиция в архитектурную гибкость и масштабируемость будущего продукта.

Список литературы

1. Trivedi, N. Why Jetpack Compose, Not XML, is the Future of Android UI – With Real-World Benchmarks / N. Trivedi // Medium. – URL: https://medium.com/@naimish-trivedi/why-jetpack-compose-not-xml-is-the-future-of-android-ui-with-real-world-benchmarks-5afd8193cfeb. – Дата публ.: 06.07.2025.
2. Gaur, A. Difference between XML vs Jetpack Compose? / A. Gaur // Medium. – URL: https://medium.com/@anandgaur2207/difference-between-xml-vs-jetpack-compose-057abbc86b95. – Дата публ.: 01.04.2025.
3. The 2025 Android UI Toolkit Showdown: Compose vs XML vs Flutter // Bootcamp / Medium. – 13.08.2025. – URL: https://medium.com/design-bootcamp/the-2025-android-ui-toolkit-showdown-compose-vs-xml-vs-flutter-55310ecb95b7. – Дата публ.: 13.08.2025.
4. Compose UI Architecture // Android Developers / Google. – URL: https://developer.android.google.cn/develop/ui/compose/architecture?hl=en#udf (дата обращения: 20.01.2026).
5. Jetpack Compose vs XML for Android UI Development // Aubergine Solutions. – URL: https://www.aubergine.co/insights/jetpack-compose-vs-xml-a-comprehensive-comparison-for-android-ui-development (дата обращения: 20.02.2026).

Талько Дарья Алексеевна, студент кафедры управления информационными ресурсами АупПРБ, dt656503@gmail.com.

Научный руководитель: Шабанович Роман Александрович, старший преподаватель кафедры управления информационными ресурсами АупПРБ, shabanovich_ra@pac.by.