УДК 004.415.2

Реактивное программирование как эволюция архитектурных компромиссов в управлении изменениями

Цуканова Дарья Александровна — студент 4-го курса направления 09.03.01 «Информатика и вычислительная техника». Научные интересы: реактивное программирование, разработка высоконагруженных информационных систем, парадигмы программирования. E-mail: GinD.Nova@yandex.ru

Забродин Андрей Владимирович — кандидат ист. наук, доцент кафедры «Информационные и вычислительные системы». Научные интересы: информационные системы, аналитика данных, парадигмы программирования, облачные технологии. E-mail: zabrodin@pgups.ru

Петербургский государственный университет путей сообщения Императора Александра I, Россия, 190031, Санкт-Петербург, Московский пр., 9

Для цитирования: Цуканова Д. А., Забродин А. В. Реактивное программирование как эволюция архитектурных компромиссов в управлении изменениями // Интеллектуальные технологии на транспорте. 2026. № 2 (46). С. 54–63. DOI: 10.20295/2413-2527-2026-246-54-63

Аннотация. *Представлено исследование эволюции моделей реактивного программирования от теоретических основ функционального реактивного программирования до современных практических реализаций в библиотеках реактивных потоков и UI-фреймворках.* ***Цель:*** *систематизировать подходы к управлению зависимостями и распространению изменений в программных системах.* ***Методы:*** *сравнительный анализ архитектурных решений в FRP, библиотеках реактивных потоков и UI-фреймворках.* ***Результаты:*** *показано, что реактивные модели не устраняют сложность управления изменениями во времени, а перераспределяют ее между кодом разработчика, механизмами выполнения и инструментами разработки. Выявлены ключевые различия между push-моделью, основанной на явном распространении событий и управлении подписками, и pull-моделью, использующей автоматическое отслеживание зависимостей.* ***Практическая значимость:*** *заключается в уточнении архитектурных компромиссов различных моделей реактивности при разработке пользовательских интерфейсов и серверных систем обработки данных.*

Ключевые слова: *реактивное программирование, функциональное реактивное программирование, push-модель, pull-модель, управление зависимостями, архитектурные паттерны*

2.3.5 — *математическое и программное обеспечение вычислительных систем, комплексов и компьютерных сетей (технические науки)*

Введение

Реактивность в программировании — это способ автоматически обновлять состояние системы в ответ на изменение входных данных. Существует множество подходов к реализации этого принципа, каждый из которых накладывает свои архитектурные ограничения и определяет синтаксис кода. Однако работы по сравнению реактивных подходов остаются фрагментарными, сосредоточенными на конкретных инструментах. Они не раскрывают общих принципов. Кроме того, отсутствует единая классификация, охватывающая реализации от теории до практики.

Цель статьи — систематизация моделей реактивности через анализ того, как они распределяют

сложность управления изменениями между кодом разработчика, механизмом выполнения и инструментами.

Основная гипотеза: реактивные модели не устраняют сложность управления изменениями, а перемещают ее с одного уровня на другой. Выбор модели определяется архитектурными особенностями системы и квалификацией разработчиков, поскольку каждый подход требует компенсировать сложность в разных местах.

Задачи:

• проследить эволюцию реактивных моделей от теоретических основ до современных реализаций;

• выявить фундаментальные компромиссы между различными моделями реактивности;

• проанализировать, как эти компромиссы проявляются в архитектуре конкретных инструментов и фреймворков.

Научная новизна исследования заключается в систематизации моделей реактивности с точки зрения распределения сложности между кодом разработчика, механизмами выполнения и инструментами разработки. Показано, что push-, pull- и гибридные модели различаются не только способом распространения изменений, но и тем, на каком уровне локализуются архитектурные риски, а также кто несет ответственность за согласованность данных.

Императивное управление состоянием: ограничения парадигмы

До реактивного программирования доминировал императивный подход. Разработчик явно описывал последовательность действий при каждом событии: получить событие, обновить состояние, найти зависимые части интерфейса, обновить их вручную. Это работало для простых программ, но с ростом масштаба возникали проблемы [1].

Основная сложность — ручное управление графом зависимостей. Как показано на рисунке, при изменении переменной *A* разработчик обязан найти

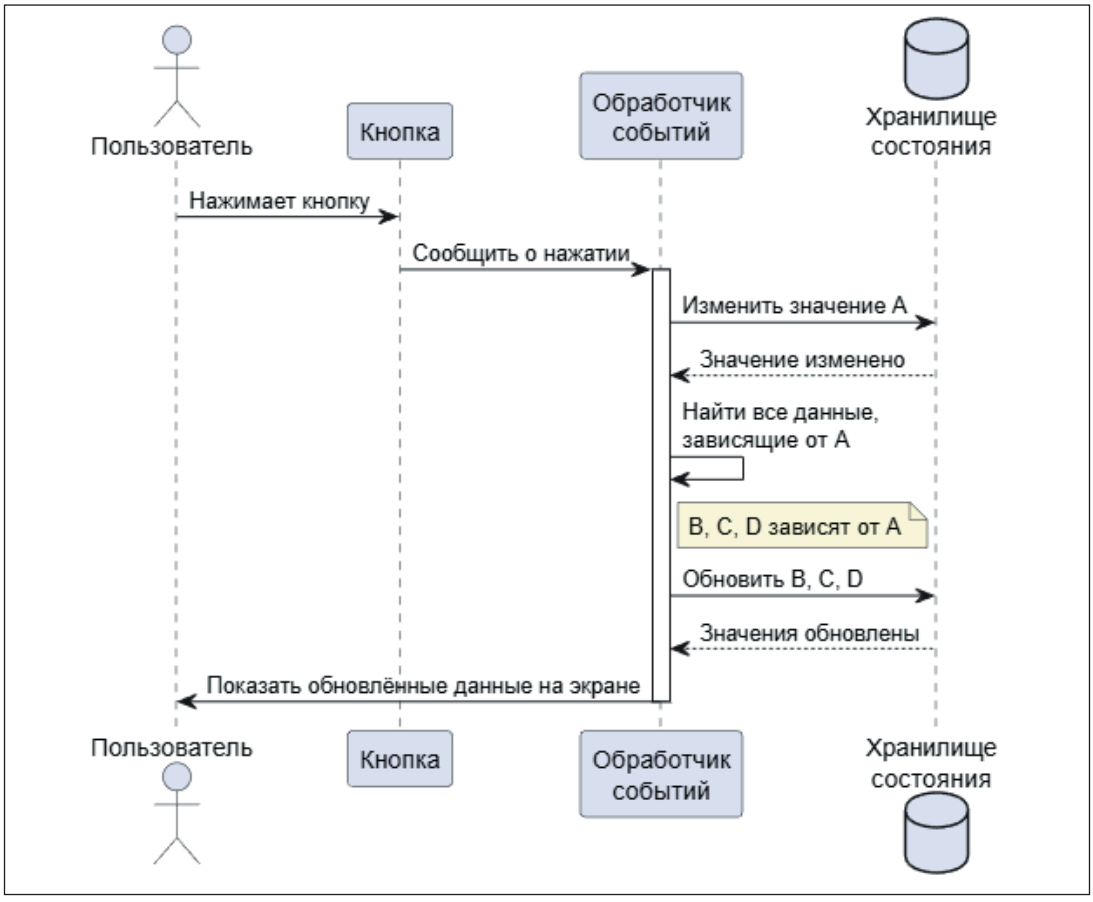

Рисунок. Императивное обновление состояния: ручное управление зависимостями

все зависящие от нее элементы (*B*, *C*, *D*), обновить их в правильном порядке и лишь затем перерисовать экран. При десятках и сотнях взаимозависимых переменных легко пропустить элемент или нарушить порядок обновлений, что приводит к рассогласованности данных и трудноуловимым ошибкам [2, 3].

Классический пример императивного подхода — паттерн Observer: наблюдатель подписывается на изменения субъекта. Это требует явного управления подписками (риск утечек памяти), не гарантирует порядка обновлений при сложных зависимостях и перекладывает ответственность за согласованность данных на разработчика. С ростом графа зависимостей задача становится неуправляемой.

Эти ограничения императивного управления состоянием задали вектор поиска альтернативных моделей: идеи dataflow и функционального программирования, оформившиеся к 1960–1970-м годам, впоследствии легли в основу функционального реактивного программирования (Functional reactive programming, FRP).

Functional reactive programming: формальная модель времени

Функциональное реактивное программирование было формализовано Коналом Эллиоттом и Полом Худаком в [4]. Вместо последовательности действий при событиях они предложили моделировать интерактивные системы как математические функции над временем.

Центральная идея: изменяющиеся величины представляются не переменными, которые обновляются, а непрерывными функциями времени с четкой денотационной семантикой (математическим значением, независимым от реализации).

Эллиотт ввел два базовых понятия, на которых строится вся система:

- events (события) — дискретные значения, возникающие в отдельные моменты времени (например, клики мышью или нажатия клавиш);
- behaviors (поведения) — значения, заданные для каждого момента времени и меняющиеся непрерывно (например, текущие координаты курсора или текущее время).

Различие между behavior и event принципиально [4]. Behavior отвечает на вопрос «каково значение сейчас?», event представляет «что произошло?». Первое непрерывно, второе дискретно.

Ключевое преимущество FRP — композируемость, обеспечиваемая функциональными абстракциями (map, аппликативные функторы, функции высшего порядка). Это позволяет строить сложные поведения из простых компонентов с корректностью, проверяемой статической типизацией. Денотационная семантика дает возможность математически доказывать свойства программ в отличие от операционной семантики, описывающей лишь механизм выполнения.

На практике идеи FRP столкнулись с тремя проблемами.

Первая — расхождение модели с реальностью: непрерывная функция времени против дискретных событий, пакетной обработки и ограниченных ресурсов [5]. Реализации вынуждены вводить эвристики (батчинг, дискретизацию, кэширование) [6], что отдаляет систему от идеальной семантики.

Вторая проблема заключается в сложности взаимодействия с внешним миром (сеть, файловая система, асинхронность) [7], которая требует дополнительных механизмов (подписки, очистка ресурсов, буферизация), иначе возникают утечки памяти и состояния гонки.

Третья проблема — трудности отладки: автоматическое распространение изменений делает порядок вычислений неявным; разработчик не может проследить последовательность обновлений, что требует специальных инструментов профилирования.

Прагматичные адаптации: от математической строгости к практической применимости

Внедрение классического FRP в реальные системы выявило существенные препятствия, которые побудили разработчиков искать более прагматичные решения. Главное отступление от первоначальной концепции коснулось временной модели: вместо непрерывного времени индустрия перешла на обработку последовательности

отдельных событий. Reactive Extensions (Rx), разработанные компанией Microsoft в начале 2000-х годов, переформатировали FRP под новую парадигму — работу с дискретными событиями [8–10]. В этой модели observable функционирует как последовательность значений, снабженная набором функциональных трансформаций (select, where, combine) [8], где поставщик данных уведомляет заинтересованные компоненты о происходящих изменениях. Главным строительным блоком выступает event; система теперь отвечает на вопрос «какие события зафиксированы?» вместо «какое значение существует в момент времени *t*?», и исходное понятие непрерывного поведения (behavior) в практическом применении Rx почти полностью вытеснено событийной моделью [5].

Reactive Extensions развивают концепцию наблюдателя, адаптируя ее для функциональных инструментов. Это позволяет выражать обработку потоков данных в декларативной форме и автоматизировать управление подписками. Такой подход лучше соответствует характеру современных вычислений и упрощает интеграцию асинхронных операций. Однако это требует переосмысления исходной семантики и увеличивает разрыв между практической реализацией и теоретическим идеалом FRP.

Классический Rx столкнулся с проблемой обратной реакции (backpressure) при работе с высокоскоростными потоками данных. Это привело к разработке более совершенных реализаций, таких как Project Reactor [11], которые интегрируют управление обратной реакцией в свою архитектуру.

Современные реактивные системы разработали набор проверенных шаблонов для управления сложностью асинхронных потоков данных [12]. Это сделало event-driven-архитектуру более предсказуемой для разработчиков.

Модели распространения изменений и их различия

После появления Reactive Extensions реактивные системы разделились на два фундаментально разных подхода к управлению зависимостями и распространению изменений: push-модель и pull-модель [13]. Сравнение моделей приведено в таблице.

В push-модели источник данных активно управляет процессом распространения изменений. Он знает о своих подписчиках (наблюдателях) и при каждом изменении самостоятельно инициирует цепочку уведомлений. Данные «проталкиваются» от источника к потребителям.

Таблица

Сравнение push- и pull-моделей

Аспект	Push-модель	Pull-модель
Инициатор изменений	Источник данных активно уведомляет наблюдателей	Реактивное вычисление неявно «вытягивает» значение при выполнении
Регистрация зависимостей	Явная подписка: observable.subscribe()	Неявная: возникает как побочный эффект чтения значения
Граф зависимостей	Статический, определяется явно разработчиком	Динамический, строится во время выполнения
Управление наблюдателями	Источник хранит список наблюдателей и уведомляет каждого	Источник заранее не знает о потребителях
Поток управления	От источника к наблюдателям	От наблюдателей к источникам
Момент уведомления	Определяет источник (уведомление отправляется немедленно)	Определяет наблюдатель (запрос данных по мере необходимости)
Примеры реализации	Reactive Extensions (RxJS, RxJava, Rx.NET), Project Reactor, event bus-системы и pub/sub-паттерны	SolidJS, MobX, Knockout, Svelte, Vue 3 (реактивные ref/effect), Angular Signals

В pull-модели зависимости фиксируются автоматически во время вычисления. Разработчик не подписывается вручную, а просто использует реактивное значение. Система, в свою очередь, самостоятельно отслеживает зависимости и повторно запускает вычисления при изменении источника.

Компромиссы между моделями реактивного программирования

Push-модель естественна для асинхронных сценариев, например непредсказуемых событий, которые могут идти от сети или пользователей. Она обеспечивает явный поток данных: видно, откуда и куда они идут.

Обратная сторона — нужно строго следить за подписками: каждая должна вовремя завершаться, иначе появляются утечки памяти. При множественных одновременных источниках возможны каскадные уведомления и лишние вычисления. Поэтому современные библиотеки предлагают батчинг и координацию, но использовать их — забота разработчика.

Pull-модель решает проблему подписок более радикально: зависимости возникают при чтении, обновляются при изменении источников и автоматически удаляются. Система сама оптимизирует порядок обновлений через граф зависимостей, гарантируя единственное вычисление за цикл с согласованными данными. Однако эта автоматизация работает только синхронно. При асинхронности (await, callback) контекст отслеживания теряется, становится невозможно восстановить связь между чтением и источником. Требуются специальные обертки или явное управление эффектами, что возвращает часть сложности. Поэтому pull доминирует в синхронных сценариях: вычисления состояний UI и производных значений, локальная мемоизация.

Гибридные системы, поддерживающие обе модели, сталкиваются с новыми вопросами: как observable интегрируется с сигналами, как гарантировать согласованность при смешивании синхронных и асинхронных обновлений. Каждый ответ добавляет концептуальную нагрузку.

Это не недостаток конкретных реализаций — это фундаментальные компромиссы, вытекающие из самой природы моделей. Сложность управления изменениями во времени не исчезает, а перераспределяется. Push перекладывает ее на разработчика, pull — на реализацию системы, а гибридные подходы требуют понимания взаимодействия обеих моделей одновременно.

Современные реализации: от теории к практике

Эволюция моделей реактивности от теоретических основ FRP привела к появлению разнообразных практических реализаций в различных областях программирования. Наиболее заметное развитие произошло в двух направлениях: UI-фреймворки, где реактивность решает задачу автоматического отражения изменений данных в интерфейсе, и серверные системы, где реактивность помогает управлять асинхронными потоками данных.

Реактивность в UI-фреймворках

В контексте пользовательских интерфейсов реактивность демонстрирует несколько принципиально разных стратегий локализации сложности.

1. Минимальная реактивность с явным управлением.

Один из подходов — отказаться от автоматического отслеживания зависимостей на уровне данных, локализовав реактивность на уровне более крупных единиц, таких как компоненты. Данный подход используется в React. Его механизмы — Virtual DOM и reconciliation — предполагают, что компонент описывается как функция входных параметров и локального состояния, при изменении которого React перестраивает виртуальное дерево и сравнивает его с предыдущим, находя минимальный набор изменений для применения к DOM-браузера [14, 15].

Преимущество подхода — в простоте ментальной модели и отсутствии неявной магии. Недостатком является необходимость ручной оптимизации, то есть разработчик должен явно мемоизировать вычисления и компоненты,

перечислять зависимости в массивах хуков, из-за чего сложность переносится в код разработчика.

2. Автоматическая реактивность через глобальное отслеживание.

Противоположным подходом можно считать автоматический перехват всех потенциальных источников изменений. Наиболее известным примером реализации данной стратегии можно считать Angular (до версии 17), который использовал Zone.js — библиотеку, перехватывающую все асинхронные операции и автоматически инициирующую процесс проверки изменений по их завершении [16].

Такой механизм помогает избавить разработчика от явного управления состояниями, но при этом проверяется все дерево компонентов на каждое изменение. Это приводит к автоматизму без явных подписок, но также может вызвать избыточную работу и непредсказуемость момента обновления. Данный подход переносит сложность в саму систему, что в некоторых ситуациях влияет на производительность.

3. Pull-модель с автоматическими зависимостями.

Третья стратегия — автоматическое отслеживание зависимостей на уровне отдельных данных, а не компонентов. SolidJS и Angular Signals (с версии 17) реализуют эту модель: компоненты инициализируются один раз, после чего обновляются только конкретные DOM-узлы, зависящие от изменившихся сигналов. Зависимости фиксируются автоматически при чтении [17–19].

Преимущество стратегии — минимальный объем обновлений без ручной оптимизации. Недостаток — необходимость понимать модель выполнения (setup и render) и ограничения при работе с асинхронностью. Сложность локализована в понимании неявной модели.

4. Компиляция реактивности.

Четвертая стратегия — анализ зависимостей статически на этапе сборки. Svelte анализирует код, выявляет реактивные зависимости и генерирует оптимальный императивный код обновлений без дополнительных библиотек во время выполнения. Разработчик пишет декларативно, компилятор превращает это в эффективные обновления DOM-браузера [20].

Преимущество — отсутствие накладных расходов при выполнении, оптимальный код. Недостаток — ограниченность возможностями статического анализа, меньшая динамичность. Сложность локализована в компиляторе и инструментах сборки.

Реактивность в серверных системах и потоках данных

За пределами UI реактивное программирование нашло применение в обработке асинхронных потоков данных, где доминирует push-модель.

1. Reactive Streams и backpressure.

Спецификация Reactive Streams (реализованная в Project Reactor, RxJava, Akka Streams) формализует асинхронную обработку потоков с явным управлением обратным давлением [8, 11]. Когда производитель генерирует данные быстрее, чем потребитель их обрабатывает, система должна координировать скорость через механизм запросов (request/demand). Это push-модель с элементами pull: потребитель явно запрашивает определенное количество элементов, которые источник затем «проталкивает».

Сложность локализована в протоколе взаимодействия между производителем и потребителем. Преимущество — контроль над потреблением ресурсов в распределенных системах. Недостаток — необходимость явно проектировать стратегии backpressure.

2. Реактивные базы данных.

Firebase, RethinkDB и подобные системы предоставляют реактивные запросы — подписки на изменения данных в реальном времени [21, 22]. Клиент подписывается на запрос один раз и автоматически получает обновления при изменении результата на сервере. Это чистая push-модель, где база данных активно уведомляет подписчиков.

Сложность перенесена на сервер: необходимость отслеживать активные подписки; эффективно вычислять, какие подписки затронуты изменением; управлять сетевыми соединениями. Преимущество заключается в простоте

клиентского кода. Недостаток — в масштабируемости сервера при большом количестве подписок.

3. Потоковая обработка данных.

Apache Kafka Streams, Apache Flink представляют модель непрерывной обработки потоков событий. Данные моделируются как бесконечные последовательности, к которым применяются трансформации (map, filter, aggregate). Это push-модель на уровне распределенной системы [23, 24].

Сложность локализована в координации распределенного состояния: партиционирование данных, гарантии доставки, управление временными окнами для агрегаций. Реактивность здесь сталкивается с теми же фундаментальными вопросами, что и в UI-фреймворках: как распространять изменения, как гарантировать согласованность, как управлять ресурсами.

Общая закономерность

Независимо от области применения (UI, серверные системы, базы данных) реактивные системы сталкиваются с одними и теми же фундаментальными компромиссами. Push-модель наиболее уместна там, где важна асинхронность и непредсказуемость событий, pull-модель — для детализации обновлений и синхронности вычислений. Гибридные подходы пытаются совместить преимущества ценой дополнительной концептуальной нагрузки.

Заключение

Таким образом, реактивное программирование эволюционировало от формальной математической модели Эллиотта до множества практических реализаций, каждая из которых решает одну и ту же задачу — автоматическое распространение изменений, но делает разные компромиссы.

Push-модель (Reactive Extensions, Reactive Streams) дает контроль и асинхронность ценой явного управления подписками и координации обновлений. Pull-модель (Signals) дает автоматизм и эффективность ценой ограничений при работе с асинхронностью. Компиляция (Svelte) переносит сложность в инструменты сборки ценой ограничений статического анализа.

Фундаментальная сложность управления изменениями во времени не устраняется — она перераспределяется между кодом разработчика, механизмами выполнения системы и инструментами разработки. Понимание этого распределения важно для практического применения реактивных моделей: разработчик должен учитывать не только возможности выбранного инструмента, но и связанную с ним зону ответственности, а также возможные классы ошибок.

Выбор модели реактивности определяет архитектурные ограничения системы и способ мышления разработчика. Тезис, сформулированный во введении, получает конкретное подтверждение в анализе современных реализаций.

СПИСОК ИСТОЧНИКОВ

1. Moseley B., Marks P. Out of the Tar Pit // Proceedings of the BCS Software Practice Advancement Conference (SPA ‘2006) (Bedfordshire, England, 26–29 March 2006). 66 p. URL: http://curtclifton.net/papers/MoseleyMarks06a.pdf (дата обращения: 26.04.2026).

2. Java Concurrency на практике = Java Concurrency in Practice / Б. Гетц [и др.]; пер. с англ. А. Логунова. СПб.: Питер, 2026. 464 с.

3. Meijer E. The Curse of the Excluded Middle // Communications of the ACM. 2014. Vol. 57, iss. 6. Pp. 50–55. DOI: 10.1145/2605176

4. Elliott C., Hudak P. Functional Reactive Animation // ACM SIGPLAN Notices. 1997. Vol. 32, no. 8. Pp. 263–273. DOI: 10.1145/258948.258973

5. Salvaneschi G., Mezini M. Reactive Behavior in Object-Oriented Applications: An Analysis and a Research Roadmap // Proceedings of the 12th Annual International Conference on Aspect-Oriented Software Development (AOSD ‘13) (Fukuoka, Japan, 24–29 March 2013) / H. Masuhara [et al.] (eds). New York: Association for Computing Machinery, 2013. Pp. 37–48. DOI: 10.1145/2451436.2451442

6. Wan Z., Taha W., Hudak P. Real-Time FRP // ACM SIGPLAN Notices. 2001. Vol. 36, no. 10. Pp. 146–156. DOI: 10.1145/507635.507654

7. Czaplicki E., Chong S. Asynchronous Functional Reactive Programming for GUIs // ACM SIGPLAN Notices. 2013. Vol. 48, no. 9. Pp. 411–422. DOI: 10.1145/2462156.2462161

8. Нуркевич Т., Кристенсен Б. Реактивное программирование с применением RxJava. Разработка асинхронных событийно-ориентированных приложений = Reactive Programming with RxJava: Creating Asynchronous, Event-based Application / пер. с англ. А. А. Слинкина. М.: ДМК Пресс, 2017. 358 с.

9. ReactiveX Observable Documentation. URL: http://reactivex.io/documentation/observable.html (дата обращения: 25.04.2026).

10. RxJS API Docs. URL: http://rxjs.dev/guide/overview (дата обращения: 25.04.2026).

11. Project Reactor Documentation. URL: http://projectreactor.io/docs (дата обращения: 25.04.2026).

12. Кун Р., Ханафи Б., Аллен Дж. Реактивные шаблоны проектирования = Reactive Design Patterns / пер. с англ. С. Черникова. СПб.: Питер, 2018. 416 с.

13. Elliott C. M. Push-Pull Functional Reactive Programming // Haskell 2009: Proceedings of the Second ACM SIGPLAN Symposium on Haskell (Edinburgh, Scotland, 03 September 2009) / S. Weirich (ed.). New York: Association for Computing Machinery, 2009. Pp. 25–36. DOI: 10.1145/1596638.1596643

14. Кумар Т. React. К вершинам мастерства: создание быстрых, производительных и интуитивно понятных веб-приложений = Fluent React: Build Fast, Performant, and Intuitive Web Applications / пер. с англ. Астана: АЛИСТ, 2025. 368 с.

15. React DOM: React Reference Overview. URL: http://react.dev/reference/react (дата обращения: 26.04.2026).

16. What is Angular? URL: http://angular.dev/overview (дата обращения: 26.04.2026).

17. Angular Signals: In-depth Guides. URL: http://angular.dev/guide/signals (дата обращения: 26.04.2026).

18. Solid Overview. URL: http://docs.solidjs.com (дата обращения: 26.04.2026).

19. Signals: Preact Guide. URL: http://preactjs.com/guide/v10/signals (дата обращения: 26.04.2026).

20. Svelte Overview: Svelte Docs. URL: http://svelte.dev/docs/svelte/overview (дата обращения: 26.04.2026).

21. Wingerath W., Ritter N., Gessert F. Real-Time and Stream Data Management: Push-Based Data in Research and Practice. Cham: Springer, 2019. 86 p. DOI: 10.1007/978-3-030-10555-6

22. Changefeeds in RethinkDB. URL: http://rethinkdb.com/docs/changefeeds/python (дата обращения: 26.04.2026).

23. Kafka Streams: Apache Kafka Docs. URL: http://kafka.apache.org/43/streams (дата обращения: 26.04.2026).

24. Apache Flink Documentation. URL: http://nightlies.apache.org/flink/flink-docs-stable (дата обращения: 26.04.2026).

Дата поступления: 18.05.2026
Решение о публикации: 22.05.2026

Reactive Programming as an Evolution of Architectural Trade-offs in Managing Changes

Darya A. Tsukanova — 4th year Bachelor's Degree Student in 09.03.01 Informatics and Computer Technology. Research interests: reactive programming, development of high-load information systems, programming paradigms. E-mail: GinD.Nova@yandex.ru

Andrey V. Zabrodin — PhD in History, Associate Professor of the "Information and Computing Systems" Department. Research interests: information systems, data analytics, programming paradigms, cloud technologies. E-mail: zabrodin@pgups.ru

Emperor Alexander I St. Petersburg State Transport University, 9 Moskovsky ave., Saint Petersburg, 190031, Russia

For citation: Tsukanova D.A., Zabrodin A.V. Reactive Programming as an Evolution of Architectural Trade-offs in Managing Changes, *Intellectual Technologies on Transport*, 2026, No. 2 (46), Pp. 54–63. DOI: 10.20295/2413-2527-2026-246-54-63 (In Russian)

Abstract. *The paper presents a study of the evolution of reactive programming models from the theoretical foundations of functional reactive programming to modern practical implementations in reactive thread libraries and UI frameworks.* ***Purpose:*** *to systematize approaches to dependency management and dissemination of changes in software systems.* ***Methods:*** *comparative analysis of architectural solutions in FRP, reactive flow libraries and UI frameworks.* ***Results:*** *it is shown that reactive models do not eliminate the complexity of managing changes over time, but redistribute it between the developer's code, execution mechanisms, and development tools. The key differences between the push model based on explicit event propagation and subscription management and the pull model using automatic dependency tracking have been identified.* ***Practical significance:*** *it consists in clarifying the architectural trade-offs of various reactivity models in the development of user interfaces and server-side data processing systems.*

Keywords: *reactive programming, functional reactive programming, push model, pull model, dependency management, architectural patterns*

REFERENCES

1. Moseley B., Marks P. Out of the Tar Pit, *Proceedings of the BCS Software Practice Advancement Conference (SPA '2006),* Bedfordshire, England, March 26–29, 2006. 66 p. Available at: http://curtclifton.net/papers/MoseleyMarks06a.pdf (accessed: April 26, 2026).

2. Goetz B., et al. Java Concurrency na praktike [Java Concurrency in Practice]. Saint Petersburg, Piter Publishing House, 2026, 464 p. (In Russian)

3. Meijer E. The Curse of the Excluded Middle, *Communications of the ACM*, 2014, vol. 57, iss. 6, pp. 50–55. DOI: 10.1145/2605176

4. Elliott C., Hudak P. Functional Reactive Animation, *ACM SIGPLAN Notices*, 1997, vol. 32, no. 8, pp. 263–273. DOI: 10.1145/258948.258973

5. Salvaneschi G., Mezini M. Reactive Behavior in Object-Oriented Applications: An Analysis and a Research Roadmap. In: *Masuhara H., et al. (eds) Proceedings of the 12th Annual International Conference on Aspect-Oriented Software Development (AOSD '13),* Fukuoka, Japan, March 24–29, 2013. New York, Association for Computing Machinery, 2013, pp. 37–48. DOI: 10.1145/2451436.2451442

6. Wan Z., Taha W., Hudak P. Real-Time FRP, *ACM SIGPLAN Notices*, 2001, vol. 36, no. 10, pp. 146–156. DOI: 10.1145/507635.507654

7. Czaplicki E., Chong S. Asynchronous Functional Reactive Programming for GUIs, *ACM SIGPLAN Notices*, 2013, vol. 48, no. 9, pp. 411–422. DOI: 10.1145/2462156.2462161

8. Nurkiewicz T., Christensen B. Reaktivnoe programmirovanie s primeneniem RxJava. Razrabotka asinkhronnykh sobytiyno-orientirovannykh prilozheniy [Reactive Programming with RxJava: Creating Asynchronous, Event-based Application]. Moscow, DMK Press Publishing House, 2017, 358 p. (In Russian)

9. ReactiveX Observable Documentation. Available at: http://reactivex.io/documentation/observable.html (accessed: April 25, 2026).

10. RxJS API Docs. Available at: http://rxjs.dev/guide/overview (accessed: April 25, 2026).

11. Project Reactor Documentation. Available at: http://projectreactor.io/docs (accessed: April 25, 2026).

12. Kuhn R., Hanafee B., Allen J. Reaktivnye shablony proektirovaniya [Reactive Design Patterns]. Saint Petersburg, Piter Publishing House, 2018, 416 p. (In Russian)

13. Elliott C. M. Push-Pull Functional Reactive Programming. In: *Weirich S. (ed.) Haskell 2009: Proceedings of the Second ACM SIGPLAN Symposium on Haskell,* Edinburgh, Scotland, September 03, 2009. New York, Association for Computing Machinery, 2009, pp. 25–36. DOI: 10.1145/1596638.1596643

14. Kumar T. React. K vershinam masterstva: sozdanie bystrykh, proizvoditelnykh i intuitivno ponyatnykh veb-prilozheniy [Fluent React: Build Fast, Performant, and Intuitive Web Applications]. Astana, ALIST Publishing House, 2025, 368 p. (In Russian)

15. React DOM: React Reference Overview. Available at: http://react.dev/reference/react (accessed: April 26, 2026).

16. What is Angular? Available at: http://angular.dev/overview (accessed: April 26, 2026).

17. Angular Signals: In-depth Guides. Available at: http://angular.dev/guide/signals (accessed: April 26, 2026).

18. Solid Overview. Available at: http://docs.solidjs.com (accessed: April 26, 2026).

19. Signals: Preact Guide. Available at: http://preactjs.com/guide/v10/signals (accessed: April 26, 2026).

20. Svelte Overview: Svelte Docs. Available at: http://svelte.dev/docs/svelte/overview (accessed: April 26, 2026).

21. Wingerath W., Ritter N., Gessert F. Real-Time and Stream Data Management: Push-Based Data in Research and Practice. Cham, Springer, 2019, 86 p. DOI: 10.1007/978-3-030-10555-6

22. Changefeeds in RethinkDB. Available at: http://rethinkdb.com/docs/changefeeds/python (accessed: April 26, 2026).

23. Kafka Streams: Apache Kafka Docs. Available at: http://kafka.apache.org/43/streams (accessed: April 26, 2026).

24. Apache Flink Documentation. Available at: http://nightlies.apache.org/flink/flink-docs-stable (accessed: April 26, 2026).

Received: May 18, 2026
Accepted: May 22, 2026