

Парадигмы программирования в 2026: от императивного к агентному

16.10.2026 · PDF-версия статьи с сайта cms.dolinais.ru

Программирование — это процесс создания и модификации компьютерных программ. Никлаус Вирт сформулировал это коротко: «Программы = алгоритмы + структуры данных». Но за этой простой формулой скрывается множество подходов к тому, как описывать алгоритмы и организовывать данные. Эти подходы называют **парадигмами программирования** — стилями мышления и наборами правил, которые определяют, как вы пишете код.

В 2026 году выбор парадигмы перестал быть академическим спором. Он напрямую влияет на скорость разработки, читаемость кода и даже на то, как ИИ-агенты смогут работать с вашей кодовой базой. Разберём основные парадигмы, их сильные стороны и практические сценарии.

Императивное программирование: классика, которая никуда не ушла

Императивный подход описывает программу как последовательность команд, изменяющих состояние. Вы говорите компьютеру «сделай это, потом это». Пример на Python:

```
total = 0
for price in prices:
    total += price
print(total)
```

Это самый интуитивный способ мышления для новичков. Языки C, Python, Java, PHP в основе своей императивны. Однако у подхода есть цена: чем сложнее система, тем труднее отслеживать изменения состояния и избегать побочных эффектов.

Когда выбирать

- Скрипты автоматизации, утилиты, простые алгоритмы.
- Задачи, где важна производительность и контроль над памятью (C, Rust).
- Обучение основам: циклы, условия, переменные.

Декларативное и функциональное программирование

Декларативный стиль отвечает на вопрос «что нужно получить», а не «как это сделать». Функциональное программирование (ФП) — его строгая ветвь, где функции рассматриваются как математические объекты без побочных эффектов. Пример на JavaScript:

```
const total = prices.reduce((acc, price) => acc + price, 0);
```

Здесь нет изменяемой переменной `total` — мы описываем преобразование. ФП становится стандартом в современных фронтенд- и бэкенд-проектах: React, Redux, Scala, Elixir, Haskell. Чистые функции легче тестировать и распараллеливать.

Комбинаторное программирование

Отдельный теоретический случай — **бесточечное (комбинаторное) программирование**, где при объявлении функций не используются промежуточные переменные, а строятся цепочки из функций. Оно основано на комбинаторной логике и представляет скорее академический интерес: излишне сложно для повседневной разработки, но полезно для понимания природы функций.

Объектно-ориентированное программирование (ООП)

ООП организует код вокруг объектов — сущностей, объединяющих данные и методы. Четыре столпа: инкапсуляция, наследование, полиморфизм, абстракция. Языки: Java, C#, C++, Python, TypeScript. ООП доминирует в корпоративной разработке благодаря возможности моделировать предметную область.

Однако злоупотребление наследованием приводит к хрупким иерархиям. Современная практика рекомендует **композицию вместо наследования** и применение принципов SOLID.

Агентное программирование: парадигма 2026 года

С развитием ИИ-агентов появился новый стиль — **агентное программирование**. Здесь программа не просто выполняет фиксированный алгоритм, а ставит цели и делегирует их выполнение автономным агентам, которые могут использовать инструменты, память и другие агенты. Это надстройка над существующими парадигмами: агент может быть написан на Python с использованием функциональных цепочек, но его поведение описывается декларативно — через цели и ограничения.

Ключевой вызов — память агента. Как и человек, агент теряет контекст при длительной сессии. Решения вроде внешней памяти (например, проекта MemShelf, обсуждаемого в 2026 году) позволяют агенту сохранять и извлекать знания за пределами текущего диалога. Это меняет архитектуру приложений: вместо жёсткого кода мы всё чаще пишем промпты, конфигурации инструментов и правила маршрутизации.

Практический пример: агент для QA

Представьте задачу создать 50 тестовых сущностей через API. Вместо ручного клика по интерфейсу (20 секунд на сущность) можно написать агента, который через `fetch` в DevTools или через библиотеку выполняет запросы пачкой. Это экономит более 15 минут на итерацию. Агентный подход позволяет описать цель: «создай 50 сущностей с разными именами» — и запустить выполнение.

Как выбрать парадигму в 2026 году

Не существует «лучшей» парадигмы. Выбор зависит от задачи, команды и экосистемы. Вот краткий гид:

1. **Скрипты и автоматизация** — императивный стиль (Python, Bash).
2. **Веб-фронтенд** — функциональный + компонентный подход (React, Svelte).
3. **Корпоративные системы** — ООП с чистой архитектурой (Java, C#).
4. **Высоконагруженные сервисы** — функциональный (Elixir, Scala) или Rust с императивным контролем.
5. **ИИ-агенты и автоматизация решений** — агентная парадигма поверх существующих языков.

На практике большинство проектов используют **мультипарадигменный** подход. Например, Python позволяет писать и императивно, и функционально, и ООП. Главное — осознанно выбирать инструмент под конкретную часть системы, а не следовать моде слепо.

Что это значит для разработчика

В 2026 году востребованы специалисты, которые понимают не один язык, а несколько парадигм. Это позволяет быстрее переключаться между проектами, лучше читать чужой код и эффективнее взаимодействовать с ИИ-ассистентами. Начните с императивного и ООП, затем добавьте функциональные приёмы, а когда столкнётесь с агентами — изучите принципы их памяти и оркестрации.

Парадигма — это линза, через которую вы смотрите на задачу. Чем больше линз в вашем арсенале, тем гибче вы как инженер.

Источник: <https://task.dolinais.ru/paradigmy-programmirovaniya-v-2026-ot-imperativnogo-k-agentnomu>