Компонентная разработка vs «Durex-код»

□□ Памятка инженера Metabot

1□ В чём разница подходов

Скриптовый подход	Компонентный подход
Код вставляется прямо в сценарий	Сценарий использует готовый компонент
Telegram-логика торчит в webhook-скрипте	Telegram спрятан внутри плагина
Сценарист не может этим пользоваться	Сценарист работает декларативно
Нет чёткого контракта	Есть входы, выходы, параметры
Разовое решение	Переиспользуемый модуль
Зависит от конкретного проекта	Доставляется в любой проект

2□ Что такое «Durex-код»

Durex-код — это:

- код, написанный «под задачу»
- код, который нельзя переиспользовать
- код, который живёт только в одном скрипте
- код, который нарушает контракты
- код, который смешивает логику канала, бизнес-логику и UX

Это не плохо. Иногда это допустимо.

Но это **не архитектурный стиль Metabot**.

---

3 Что такое компонентный стиль Metabot

Компонент — это:

- атомарная команда
- с чёткими входами
- с чёткими выходами
- с понятным поведением
- с изолированной зоной ответственности
- со спрятанной инженерной сложностью

Пример:

```
const VoiceInput = require('Common.Voice.VoiceInput')

return VoiceInput.expect({
  code: "orion_profiling_q1_voice",

  lead,

  successScript: "orion_profiling_q2",
  cancelScript: "orion_profiling_cancelled",

  targetAttr: "orion_profiling_q1_text",
  sourceAttr: "orion_profiling_q1_voice_url",

  extraAttrs: {
    active_agent: "orion",
    voice_context: "orion_profiling_q1",
    input_mode: "profiling"
  },
```

```
  processorScript: "System_VoiceInput_Processor",

  stt: {
    provider: "openai",
    options: { model: "whisper-1", language: "ru" },
    asyncResponse: true,
    tokenKey: "OPENAI_API_KEY"
  }
})
```

Сценарист видит:

- targetAttr
- sourceAttr
- successScript
- cancelScript
- stt-параметры

Он не видит:

- Telegram API
- file_id
- webhook структуру
- async механику
- OpenAI SDK

Это декларативный стиль.

---

4 Что такое декларативный подход

Декларативный подход — это:

> Мы описываем ЧТО должно произойти, а не КАК это происходит.

Сценарист пишет:

```
Ожидай голос.
Перейди в success.
Сохрани результат.
```

А не:

```
Проверь payload.message.voice
Достань file_id
Сходи в Telegram API
Отправь в STT
Подожди callback
Распарси JSON
```

---

5 Принципы компонентной разработки Metabot

1. Чёткий контракт

Каждый компонент обязан иметь:

- входные параметры
- выходные данные
- предсказуемое поведение
- задокументированные переходы

---

2. Нет неявной передачи данных

 Плохо:

```
memory.foo = 123
```

Потом в другом месте:

```
memory.foo
```

Это разрыв контракта.

✔ Правильно:

- вход через параметры
- управляемый выход, например, через targetAttr
- всё прозрачно

---

3. Одна зона ответственности

Пример:

VoiceInput:

- отвечает за голосовой ввод
- не отвечает за каналы
- не отвечает за Telegram

ArtifactResolver:

- отвечает за извлечение артефактов из всех каналов
- не отвечает за STT
- не отвечает за сценарий

---

4. Никаких «всё в один файл»

Если логика растёт — создаём слой.

Пример:

- Был Telegram-код внутри VoiceInput
- Появился Max
- Вместо if/else-ада — создаём Channel.ArtifactResolver

Это инженерный подход.

---

5. Повторное использование — обязательное требование

Если модуль нельзя переиспользовать — это не компонент.

Компонент должен быть:

- переносимым
- подключаемым
- поставляемым как плагин

---

6 Почему это важно

1. Сценаристам проще
2. Интеграторам проще
3. Код чище
4. Проекты масштабируются
5. Плагины можно доставлять в разные компании
6. Мы получаем архитектурную степень свободы

И самое интересное:

 По времени это занимает примерно столько же.

Разница — в мышлении.

6.1 Декларативный стиль = расширяемая палитра Metabot

Декларативный подход — это не только «красиво».

Это позволяет нам:

- расширять палитру компонентов
- развивать low-code платформу
- строить визуальные конструкторы
- делать архитектуру масштабируемой

Сейчас в Metabot есть ~21 атомарная команда.

Примеры:

- SendMessage
- SendImage
- SendFile
- Input
- VoiceInput
- TransferToOperator
- RunScript
- и т.д.

Каждый компонент:

- имеет входы
- имеет выходы
- имеет чёткие параметры
- ведёт себя предсказуемо

Это позволяет:

1 Легко расширять платформу Добавили новый компонент → он сразу становится частью палитры.

2 Делать визуальные конструкторы Если у компонента есть вход/выход — можно рисовать схемы.

3 Строить workflow-системы Компоненты становятся узлами графа.

4 Делать экспортируемые решения Компонент можно доставить в другой проект.

6.2 Атомарность = возможность визуализации

Если компонент описан как:

- чёрный ящик
- с чёткими входами
- с чёткими выходами

его можно:

- визуализировать
- соединять стрелками
- использовать в low-code
- использовать в no-code
- переносить между проектами

Если код:

- размазан по скриптам
- читает memory
- меняет глобальные объекты
- зависит от webhook структуры

его нельзя:

- визуализировать
- стандартизировать
- масштабировать

---

7 Архитектор vs Скриптовик

Скриптовик думает:

> Мне нужно сделать, чтобы работало.

Архитектор думает:

>

> Как сделать, чтобы это работало в 10 проектах.

8️⃣ Как использовать это как AI-чеклист

Можно кидать в AI вместе с кодом и спрашивать перед коммитом:

- Есть ли здесь чёткий контракт?
- Понятен ли этот код сценаристу?
- Есть ли неявная передача данных?
- Нарушена ли зона ответственности?
- Можно ли переиспользовать этот код?
- Можно ли этот код вынести в компонент?
- Можно ли визуализировать это поведение как узел workflow?
- Является ли это Durex-кодом или системным модулем?

Если на 3+ вопроса ответ «нет» — ты пишешь Durex.

9️⃣ Когда допустим Durex-код

Иногда допустимо:

- разовая миграция
- временный костыль
- hotfix

Но:

- он не должен становиться системой
- он не должен множиться
- он не должен ломать архитектуру

---

1 0 Когда допустимо писать код в скрипте

Код в скрипте допускается.

Но только если соблюдены условия.

✔ Допустимые случаи

1 Абсолютно разовая логика

- временная миграция
- одноразовый расчёт
- локальный pipeline
- подсчёт баллов
- простая условная логика

Пример:

```
if (lead.getAttr("score") > 10) {
  return bot.run({ script_code: "vip_branch" })
}
```

Это допустимо.

---

2 Логика понятна сценаристу

Если сценарист:

- может прочитать
- может понять
- может отредактировать

— код допустим.

Если сценарист не может понять код — код не должен находиться в сценарии.

Исключение — системные вещи без которых абсолютно нельзя обойтись, например, вызов плагинов.

3️⃣ Логика не нарушает архитектуру

Допустимый код:

- не читает неявные memory
- не зависит от глобальных переменных
- не создаёт скрытых зависимостей
- не нарушает контракт компонентов

🚫 Недопустимый Durex-код

Код нельзя писать в сценарии, если:

- он сложный
- он интеграционный
- он работает с API
- он обрабатывает webhook
- он содержит асинхронную логику
- он требует инженерного уровня знаний

Такой код должен быть:

- вынесен в плагин
- спрятан под компонент
- задокументирован
- переиспользуем

1️⃣1️⃣ Финальная мысль

Metabot — это не набор скриптов.

Это:

- палитра компонентов
- декларативный конструктор
- платформа

Если ты пишешь код, который нельзя вынести в палитру — ты... выпускаешь Durex.

Если ты пишешь код, который расширяет палитру — ты работаешь как инженер платформы.

P.S. Почему «Durex»?

Компонент — это **капитал**: инвестиция и актив, который ставится на баланс. Его можно переиспользовать, масштабировать и доставлять в десятках проектов — он создаёт системную ценность.

А разовый скрипт — это **расход периода**: проект сделали, прибыль получили — но актива не появилось. Работу сделали один раз, в следующем проекте — делаем то же самое заново. Все при деле, задача выполняется, но капитал не создаётся.

Версия #7
Artem Garashko создал 2026-02-17 13:18:28 UTC
Artem Garashko обновил 2026-02-17 15:59:00 UTC