Программирование: как проверить код на соответствие структурной парадигме в 2026 году

26.11.2026 · PDF-версия статьи с сайта cms.dolinais.ru

Зачем в 2026 году вспоминать структурное программирование

Структурное программирование — парадигма, в основе которой лежит представление программы в виде иерархической структуры блоков. Концепция сформировалась в конце 1960-х — начале 1970-х годов благодаря теореме Бёма — Якопини и работе Эдсгера Дейкстры «О вреде оператора goto». Согласно парадигме, любая программа, построенная без goto, состоит из трёх базовых управляющих конструкций: последовательности, ветвления и цикла, а также подпрограмм. Разработка ведётся пошагово, методом «сверху вниз».

Может показаться, что в эпоху ИИ-агентов и вайбкодинга это архаизм. На практике всё наоборот: чем больше кода генерируют машины, тем важнее проверяемый инженерный фундамент. Структурный код проще читать, тестировать и объяснять — и людям, и ИИ-агентам. В этой статье — практические шаги, как проверить проект на соответствие парадигме и что делать с находками.

> Структурность — это не про запрет goto, а про предсказуемость потока управления. Если по коду можно мысленно пройти сверху вниз без телепортов, он структурный.

Шаг 1. Зафиксируйте правила проверки

Прежде чем запускать линтеры, договоритесь, что считаете нарушением. Минимальный набор критериев:

- нет оператора goto и его аналогов (например, меток и переходов в языках, где они есть);
- каждая функция имеет одну точку входа и, желательно, одну точку выхода — либо выходы явно перечислены и легко обозримы;
- глубина вложенности управляющих конструкций не превышает 3–4 уровней;
- циклы не меняют счётчик внутри тела (никаких i = i + 1 в теле for);
- побочные эффекты отделены от вычислений: функция либо возвращает значение, либо меняет состояние, но не то и другое одновременно без явной причины.

Эти правила не догма, а отправная точка. Запишите их в CONTRIBUTING.md или в конфиг линтера — чтобы и люди, и ИИ-агенты видели одни и те же ожидания.

Шаг 2. Подключите автоматические проверки

Ручной ревью не масштабируется. Вот что стоит включить в CI в 2026 году:

1. **Линтеры с правилами сложности.** В ESLint — complexity, max-depth, no-labels; в Pylint — too-many-branches, too-many-nested-blocks; в SonarQube — когнитивная сложность. Порог подбирайте по кодовой базе, но не выше 15–20 для цикломатической сложности функции.
2. **Поиск goto и меток.** Даже если язык допускает goto (C, C++, Go), добавьте правило запрета или ограничьте его применение отдельным списком исключений.

3. **Статический анализ потока управления.** Инструменты вроде Semgrep или CodeQL позволяют описать собственные шаблоны: например, «цикл, внутри которого есть return» или «функция с более чем тремя return».
4. **Проверка на «спагетти» через метрики.** Считайте среднюю глубину вложенности и долю функций с когнитивной сложностью выше порога. Рост этих метрик от релиза к релизу — сигнал, что структурность размывается.

Важно: автоматика ловит форму, но не смысл. Линтер не поймёт, что функция на 40 строк с одним return всё равно нечитаема. Поэтому шаг 3 обязателен.

Шаг 3. Проведите выборочный ручной аудит

Возьмите 5–10 самых изменяемых файлов за последний квартал (это легко сделать через git log) и пройдитесь по ним по чек-листу:

- можно ли прочитать функцию сверху вниз, не возвращаясь назад?
- нет ли флагов-переключателей, которые меняют поведение в середине тела?
- вынесены ли вложенные условия в отдельные функции с говорящими именами?
- нет ли дублирования блоков, которое можно свернуть в подпрограмму?

Типичная находка — «ромбовидный» код: множество вложенных if-else, где каждый уровень добавляет условие. Лечится ранними возвратами (early return) и выделением предикатов: if (!isValid(user)) return; вместо трёх уровней вложенности.

Шаг 4. Исправляйте по приоритету, а не всё подряд

Не превращайте проверку в крестовый поход. Приоритеты такие:

1. Код, который часто меняется и часто падает — сначала там.
2. Код на критическом пути (платежи, авторизация, обработка данных) — вторым.
3. Остальное — по мере касания.

Рефакторинг ради соответствия парадигме без тестов — риск. Поэтому перед переписыванием убедитесь, что поведение зафиксировано тестами. Если тестов нет, начните с характеризационных тестов: они фиксируют текущее поведение, даже если оно странное.

Шаг 5. Встройте проверку в работу с ИИ-агентами

ИИ-агенты генерируют код быстро, но склонны к глубокой вложенности и «умным» однострочникам. Чтобы структурность не размывалась:

- добавьте в промпт явные ограничения: «глубина вложенности не более 3», «одна точка выхода», «без goto»;
- прогоняйте сгенерированный код через те же линтеры до ревью человеком;
- просите агента объяснить поток управления — если объяснение путаное, код, скорее всего, тоже.

Это не бюрократия, а способ сохранить инженерный фундамент. Структурный код дешевле сопровождать: по

данным многолетних наблюдений, основная стоимость ПО — не написание, а чтение и изменение.

Что считать успехом

Хороший результат проверки — не «ноль нарушений», а управляемая динамика. Зафиксируйте базовые метрики (средняя сложность, глубина вложенности, доля функций-«монстров»), а затем следите, чтобы они не росли. Периодически пересматривайте пороги: кодовая база меняется, и то, что было нормой год назад, сегодня может быть техническим долгом.

Структурное программирование — не музейный экспонат, а рабочий фильтр. Он не мешает ни ИИ, ни новым парадигмам, а даёт общий язык для людей и машин. Проверяйте код регулярно, автоматизируйте форму, оставляя смысл людям, — и проект переживёт не один рефакторинг.

Источник: https://cms.dolinais.ru/programmirovanie-kak-proverit-kod-na-sootvetstvie-strukturnoy-paradigme-v-2026-godu