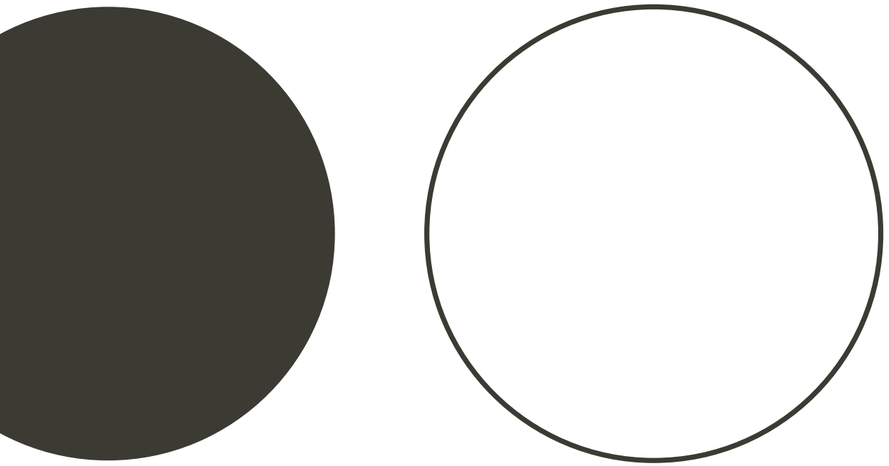
Паттерны и практики написания кода

SOLID-принципы. Часть 2

принцип открытости/закрытости *Open-Closed Principle (OCP)*

КЛЮЧЕВАЯ ИДЕЯ ПРИНЦИПА

изменения в проекте должны происходить через написание нового кода, а не модификацию старого



В долгоживущих проектах происходят постоянные изменения → желание разработчиков меньше заниматься регрессионным тестированием и перепроектированием сущностей → стремление к переиспользованию существующих реализаций и написание исключительно нового кода.

Сформулирован *Бертраном Майером* в 1988 году, затем подведён Робертом Мартином к концепции SOLID.

Программные сущности должны быть открыты для расширения и закрыты для изменения.

свойства абстракций, увеличивающие гибкость системы

- **помогают бороться со сложностью реальности.** За счёт того, что нам достаточно обобщить всё многообразие рассматриваемого объекта

- **выделяют общие части системы, игнорируя некоторые детали.** Так они скрывают свойства и действия, несущественные для реализации задачи, и оставляют минимально необходимый набор методов для манипуляции классами

+ благодаря этим свойствам абстракций усиливается гибкость системы. При взаимодействии сущностей будет проявляться слабая связанность – low coupling

плюсы принципа

- **разработчик всегда пишет только новый код.** Избавляемся от переписывания старых реализаций, занимаемся только написанием нового кода. Минимум рефакторинга: не надо продумывать, как не сломать смежные сценарии и сохранить гибкость системы

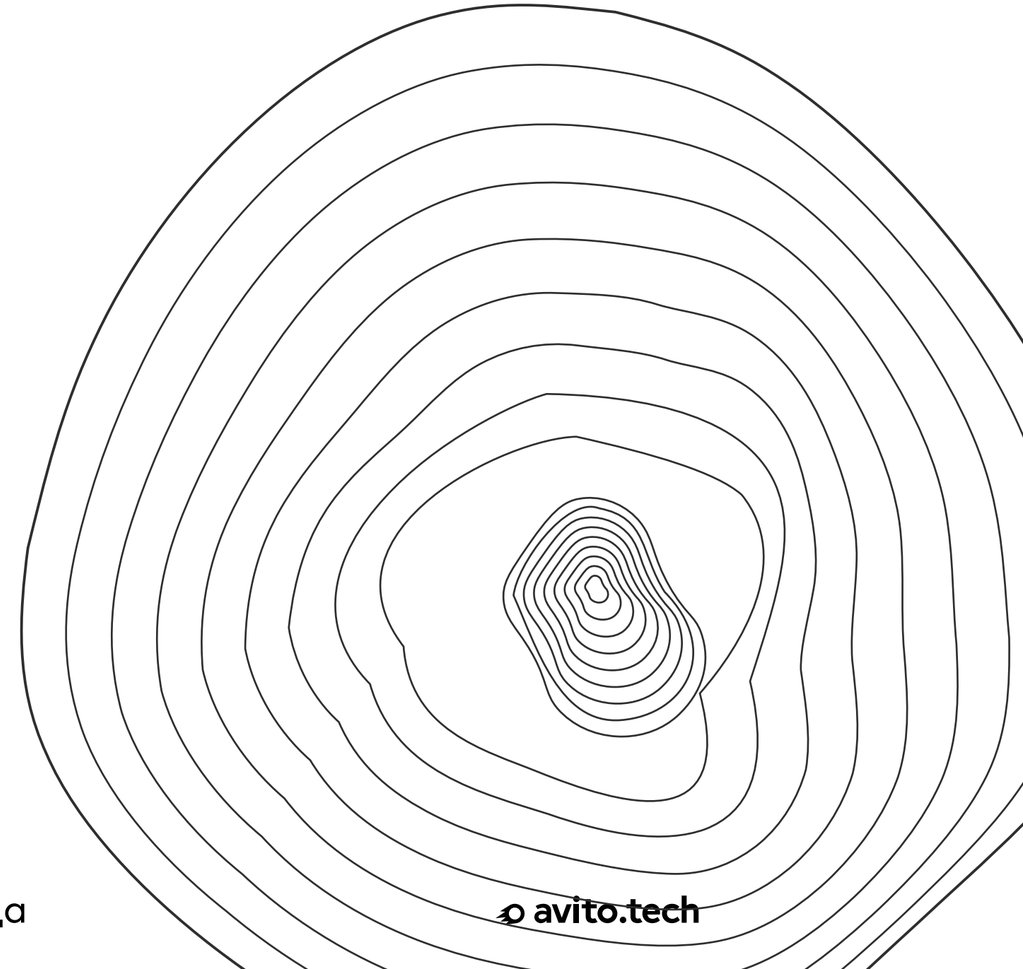
- **тестирование нужно только один раз.** Не тратим время на регресс. Теперь достаточно протестировать только новый код, поскольку старый не будет изменён, а значит в нём не появятся баги

- **не нужно тратить время на частый рефакторинг** за счёт экономии время

ещё несколько советов

- **пишите абстракции, которые подходят под текущие задачи.** Создавайте реализации, которые охватывают небольшое число аспектов

- **обилие избыточных и ненужных абстракций сильно закрепощает и усложняет проект.** Чем меньше задач у сущности, тем проще управлять реализацией



как жить с оператором new, который создает объекты

Оператор new закрепощает код: желая подменить тело метода, приходится что-то делать с инстанцированием. Обычно, при работе с кодом мы делаем что-то одно. Либо мы подменяем реализацию метода, либо инстанцируем другой класс, который подходит под исходный интерфейс. Делать и то, и другое одновременно в одном действии – плохое решение.

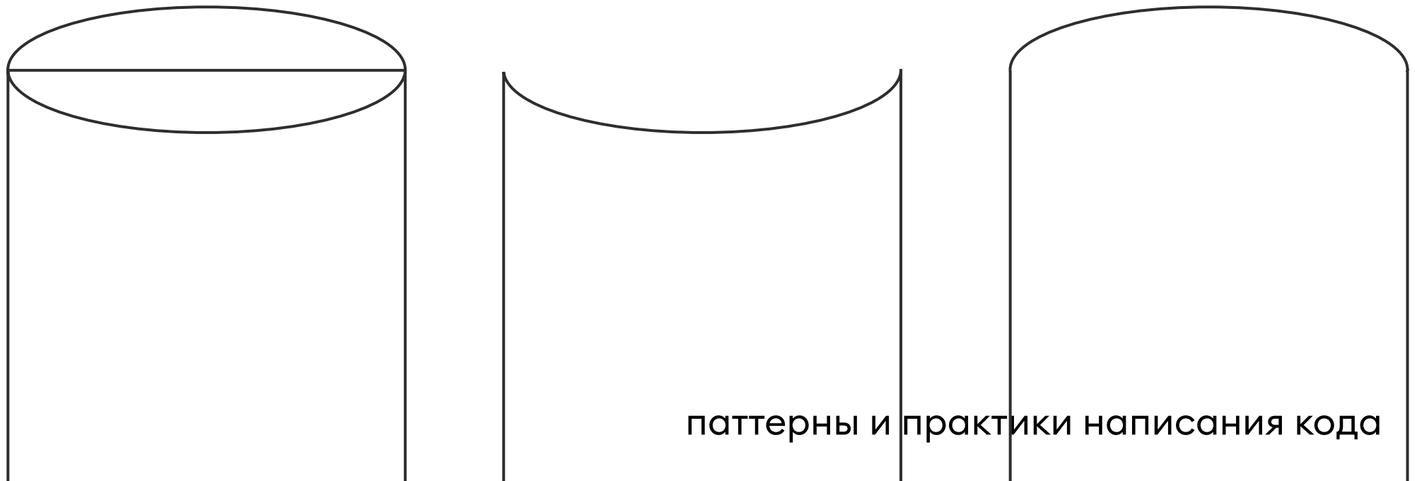
ЧТО ДЕЛАЕМ

Делим большой метод на несколько маленьких. Сам процесс инстанцирования можно вынести в обособленный метод. В этом случае подменить реализацию станет проще.

В идеале же нам хочется, чтобы конструкция new появлялась в коде минимально или отсутствовала совсем. Такие способы тоже есть. Давайте немного забежим вперёд в те темы, которые нам только предстоит изучить.

варианты

- **изолировать инстанцирование в специальные классы (фабрики).** Мы вернемся к этому в следующем сезоне
- **воспользоваться внедрением зависимости (DI-контейнерами).** Подход сложен во внедрении в старый проект, но зато даёт массу преимуществ в повышении гибкости
- **построить зависимости от интерфейсов и абстракций, а не от конкретных классов (DIP).** Об инверсии и для чего она нужна вы узнаете буквально через несколько видео



трудности при использовании принципа открытости/закрытости

→ **на продумывание абстракций может уходить много времени.** Основной проблемой становится придумывание правильных классов, которые можно будет легко расширять. Наградой за такую трату времени будет то, что: очередной рефакторинг будет сильно оттягиваться по времени, хотя к нему все равно придется прибегать

→ **невозможно придумать такую абстракцию, которая удовлетворит всему спектру изменений.** Их цель – бороться со сложностью путём скрытия несущественных деталей. В итоге, происходит заточка на конкретную абстракцию. При необходимости перейти на другую, придётся делать рефакторинг или вставлять костыли

→ **повсеместное применение принципа увеличивает количество классов.** Поскольку принцип требует от нас ничего не переписывать и не рефакторить, то лавинообразный рост классов будет обеспечен. Подходим к этому с головой и всё-таки периодически делаем рефакторинг


avito.tech