Паттерны и практики написания кода

SOLID-принципы. Часть 4

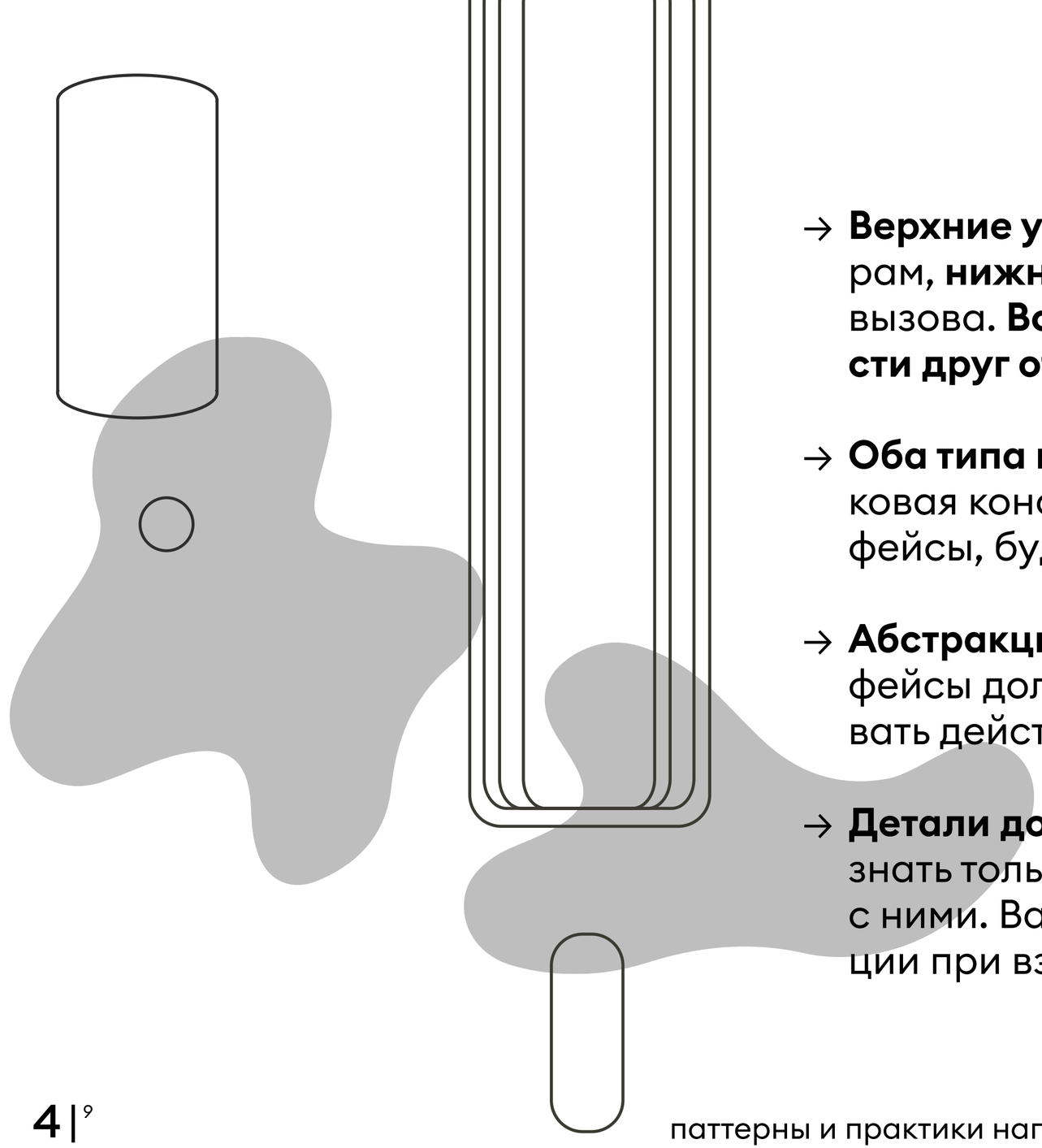
принцип инверсии зависимости

Dependency Inversion Principle (DIP)

Все наши зависимости должны строиться относительно абстракций, а не деталей.

ОПРЕДЕЛЕНИЕ

Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций. Абстракции не должны зависеть от деталей, но детали должны зависеть от абстракций.



→ **Верхние уровни** – те, что находятся ближе к контроллерам, **нижние** – ближе к базе данных или глубже по стеку вызова. **Все модули не должны иметь прямой зависимости друг от друга.**

→ **Оба типа модулей должны зависеть от абстракций.** Языковая конструкция, описывающая абстракции – интерфейсы, будем использовать их.

→ **Абстракции не должны зависеть от деталей.** Наши интерфейсы должны быть полностью независимыми и описывать действительно абстрактную единицу.

→ **Детали должны зависеть от абстракций.** Классы должны знать только об интерфейсах и взаимодействовать только с ними. Важно отсутствие знаний о конкретной реализации при взаимодействии.

минусы прямой зависимости

- **невозможно подменить реализацию классов** в такой схеме – все классы и их создание прописаны внутри метода, замена классов усложняется – приходится переписывать реализации, которые уже были.
- **юнит-тесты становится сложно писать,** всё вызывается по цепочке.
- **бизнес-логика растекается по классам,** становится непонятно, какой класс и какая абстракция должна реализовать ответственность.
- в итоге **увеличивается зависимость классов друг от друга.** Реализовать в такой схеме слабую связанность становится невозможно.

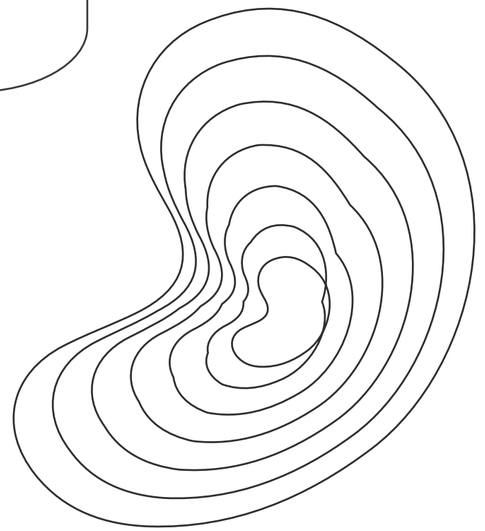
Виновата конструкция new, нужно изолировать ее в отдельных методах или специальных классах. При должном уровне абстракций и реализации такого подхода, мы расщепим классы и применим Low Coupling. Именно эту задачу решает принцип Dependency Inversion.

плюсы инверсии

- **теперь нижний уровень внедряется в верхний.**
  - **улучшается сцепление методов в классе и уменьшается связность** (High Cohesion и Low Coupling).
    - **методы в классе концентрируются на выполнение задач своей сущности, реализуя принцип персональной ответственности.**
      - **инвертирование работы сущностей делает наши классы менее зависимыми друг от друга.**
        - **чтобы работать с абстракциями, мы стремимся использовать интерфейсы.** Они принадлежат верхнему уровню, от которого реализован нижний. Это улучшает слоистость приложения и независимость классов друг от друга.

технические особенности подхода

- **При изменении интерфейсов меняется реализация нижнего уровня.** Чем точнее выбирается абстракция, тем реже нужно будет проводить рефакторинг.
- **Стараемся сделать интерфейс как можно более общим.** Он не должен ни от кого зависеть.
- **Изменения реализации нижних уровней не должны изменять классы верхнего уровня.** Иначе становится понятно, что были сделаны сильные просчёты при начальном проектировании. Эта проблема нейтрализует все преимущества инверсии и превращает код в прямую зависимость.



все пять принципов SOLID

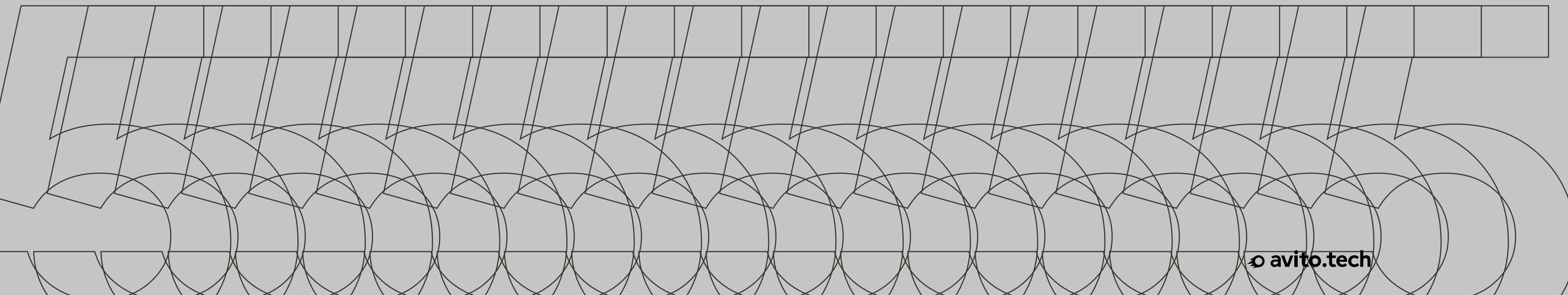
Принцип персональной ответственности. Разделяет код на уровни, фокусирует цели классов и удовлетворяет High Cohesion.

Принцип открытости/закрытости. Борется с изменениями в проекте.

Принцип подстановки Лисков. Задаёт правило наследования классов.

Принцип разделения интерфейсов. Подбирает правильные абстракции и работает совместно с принципом персональной ответственности.

Принцип инверсии зависимости. Удовлетворяет Low Coupling и инвертирует связь между классами.




avito.tech