Паттерны и практики написания кода

SOLID-принципы. Часть 1

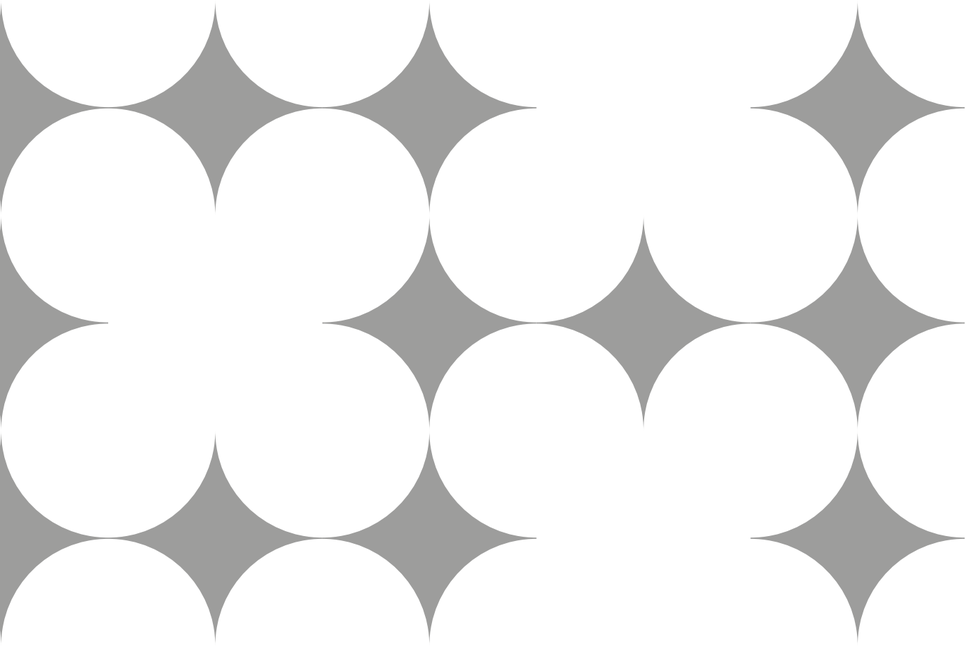
вступление

SOLID-принципы описаны в книге **«ГИБКАЯ РАЗРАБОТКА ПРОГРАММ»** – 2002, *Роберт Мартин aka Дядюшка Боб.*

В книге «Чистая архитектура» –2017 их рассмотрели с точки зрения архитектуры, а также были даны пояснения для их использования.

Прежде всего, SOLID-принципы формируют правила работы в больших проектах. Для маленьких проектов и узкоспециализированных задач SOLID будет бесполезен. В таких задачах его поддержка будет обходиться дорого, а профит минимален.

о каких принципах мы говорим



- **SOLID** (англ. твёрдый, крепкий, прочный), акроним
- Single Responsibility Principle (SRP) – Принцип персональной ответственности
- Open-Closed Principle (OCP) – Принцип открытости/закрытости
- Liskov Substitution Principle (LSP) – Принцип подстановки Лисков
- Interface Segregation Principle (ISP) Принцип разделения интерфейсов
- Dependency Inversion Principle (DIP) – Принцип инверсии зависимости

основные плюсы принципов

SOLID-принципы делают код устойчивым к постоянным изменениям. Это происходит за счёт внедрения правильных проектировочных решений.

SOLID уменьшает необходимость проведения рефакторинга. Это полезный процесс, но он времязатратен. Освободившееся время можно использовать на разработку новой функциональности.

Увеличение гибкости проекта. Речь идет о простоте расширения, масштабируемости и переиспользовании кода.

SOLID удовлетворяет Low Coupling и High Cohesion. Он может стать для них прикладной реализацией.

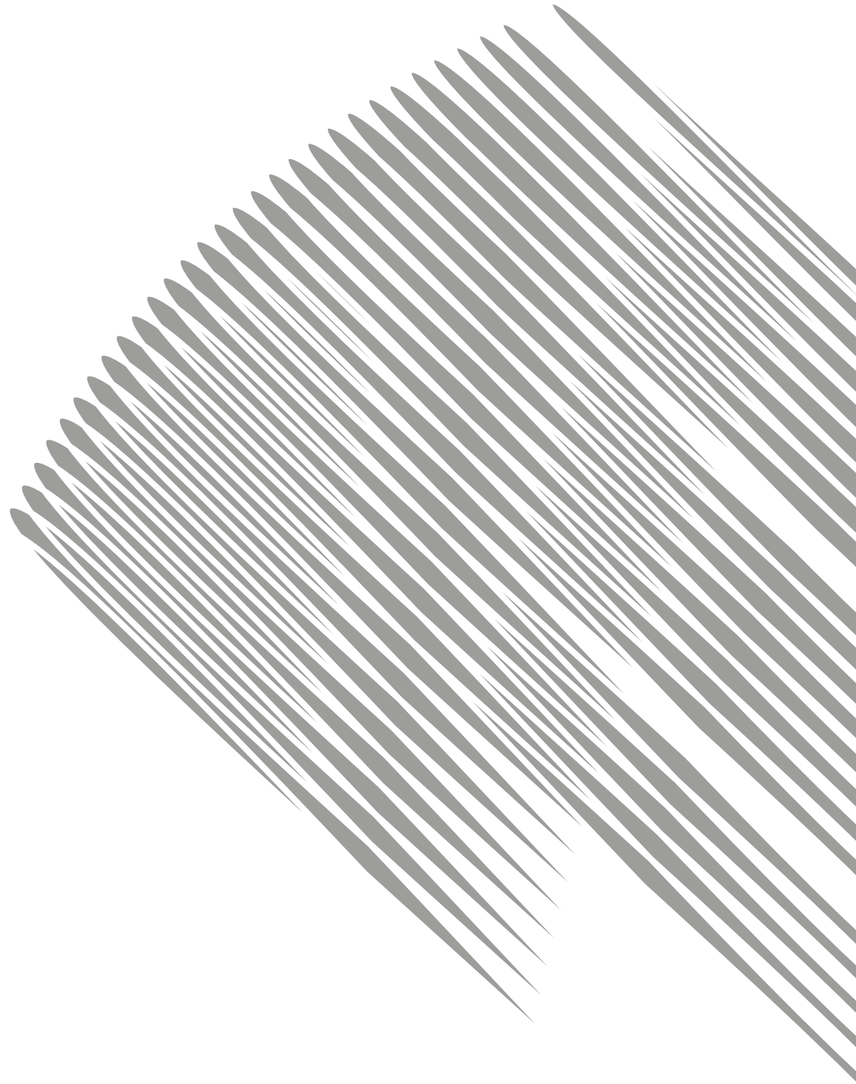
SOLID идеален для написания тестов. Теперь не придётся думать, тестопригодный код или нет. А если в вашем проекте нет практики написания юнитов, то, придерживаясь SOLID, вам будет проще его внедрить.

когда можно отказаться от SOLID

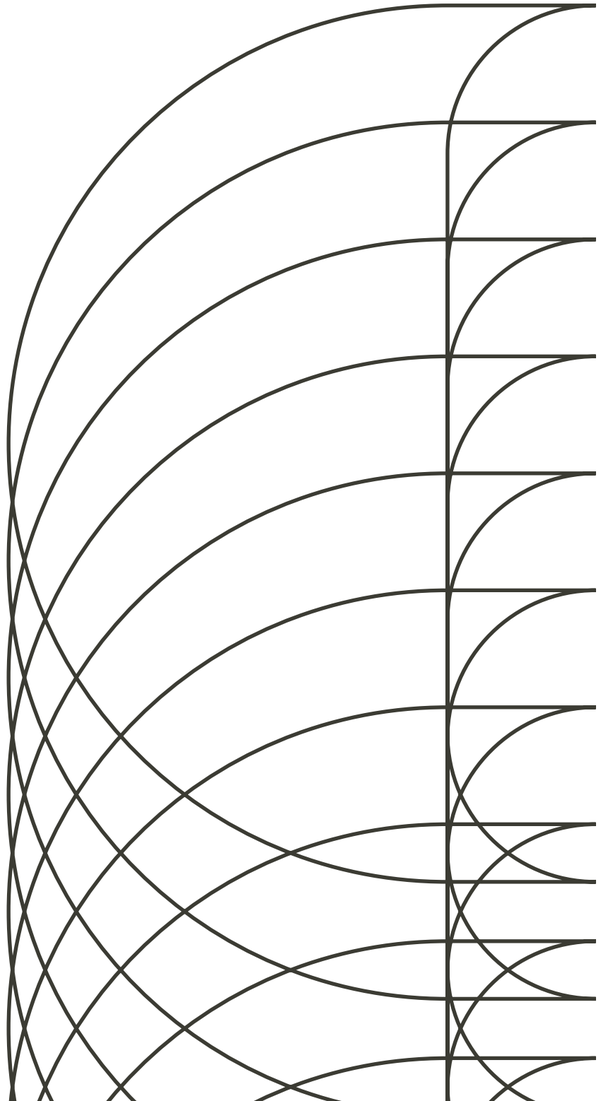
→ **В местах, которые редко меняются.** Например, вы делаете проект, после запуска которого не будет происходить в нём изменений.

→ **Когда не нужно придавать гибкость коду.** Если код не будет меняться или будет удален в будущем, то нет смысла придавать ему гибкость.

→ **В узкоспециальных задачах.** Когда код обслуживает узкоспециализированные задачи, его сложно переиспользовать из-за специфической логики. Значит нет большого смысла делать его гибким.



принцип персональной ответственности



Первый принцип SOLID – Single Responsibility Principle (Принцип персональной ответственности)

Часто возникает проблема, когда класс выполняет слишком много задач. Здесь следует установить границы обязанностей и правила того, что класс делает. Всё, что не входит в эти правила, выносим в соседние или новые сущности. Границы каждого класса зависят от задачи и вероятности внесения правок.

Класс должен иметь одну и только одну причину для изменения. Сформулирован *Томом Де-Марко (1979)* и *Мейлером Пейдж-Джонсаном (1988)*.

минусы нескольких ответственностей у кода

— **Ответственность начинает растекаться по разным классам.** Слишком большое количество логики трудно сохранить в рамках одного класса – границы начинают раздуваться и сопрягаться с другими подобными классами.

Ответственности сильно сплетаются и их становится сложно отделить друг от друга. При разделении приходится проводить большой рефакторинг. Иногда приходится создавать несколько дополнительных классов. Нужно это чтобы переложить особые действия, которые раньше выполняли классы, работающие вместе.

При объединении ответственностей тестирование усложняется за счёт множества вариантов поведения. Повышается риск появления багов, не все тест-кейсы будут покрыты.

плюсы принципа персональной ответственности

+ **Ответственность локализована в одном месте.** Каждый класс атомарен в своих действиях. С ним становится легко работать, переписать или заменить на другую реализацию. Можно добавить или изменить выполняемые в нём действия. При этом у вас не будет проблем с выбором того, в какой класс добавить тот или иной новый метод.

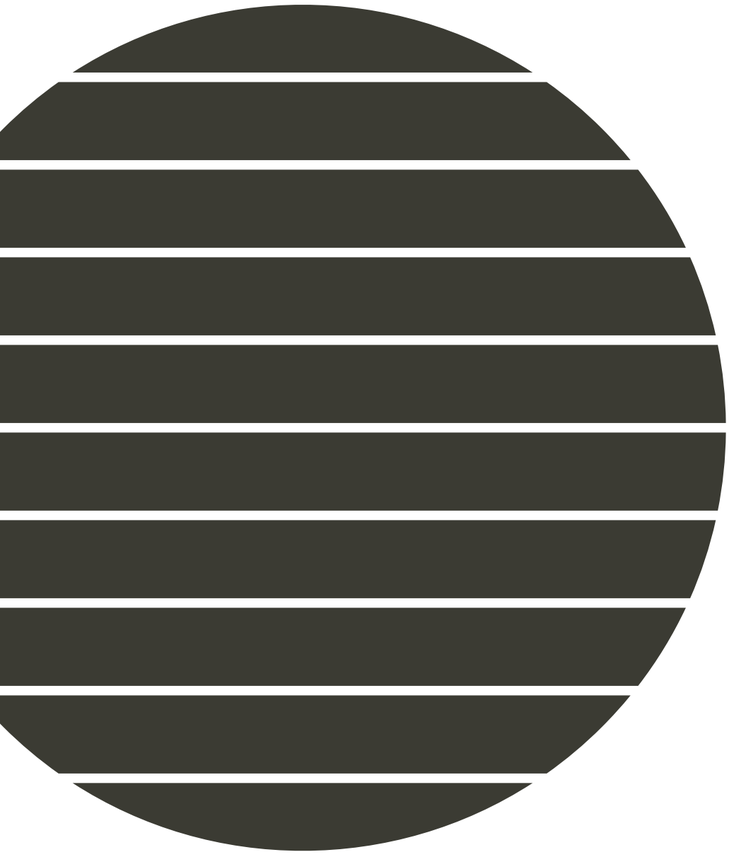
Не нарушается DRY. SRP удовлетворяет DRY и они работают в тандеме друг с другом.

+ **Маленький атомарный класс просто протестировать.** Поскольку такие классы небольшие, их легко прочитать. Сразу понятно за что они отвечают без комментариев и специальных пояснений.

Класс должен иметь одну и только одну причину для изменения, но в особо специфичных сценариях ее бывает сложно выбрать. В таком случае фокусируемся на конкретном пользователе или заинтересованном лице (акторе). Таким образом:

Класс должен отвечать за одного и только за одного актора.

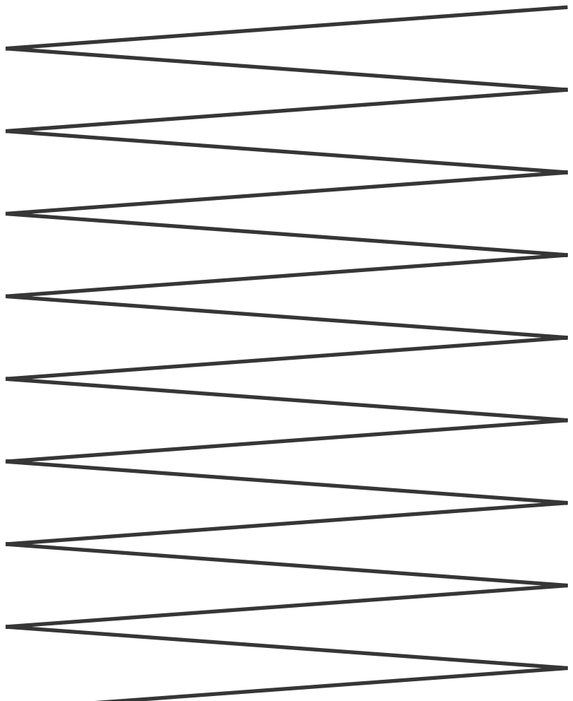
минусы принципа персональной ответственности



— **Появляется множество декомпозированных классов.** В специфических сценариях это может стать бессмысленной работой

= **Происходит усложнение чтения и изучения системы из-за большой кодовой базы.** Когда во всём проекте бессмысленно применяется подход, то тысячи классов с одним-двумя методами начнут усложнять навигацию по нему.

когда наделение задач классам не даёт никаких преимуществ?

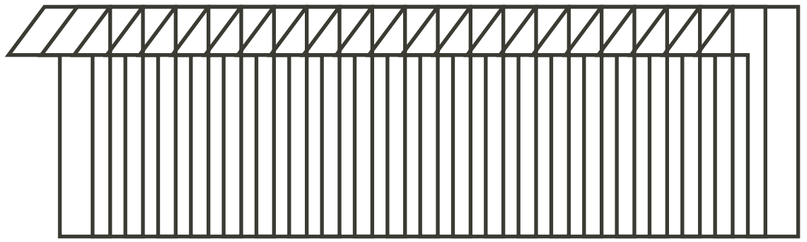


→ **Когда заранее известно о неизменности кода в конкретном месте.** Обычно это реализация узко специфичной задачи. Другим вариантом будет проверка продуктовой гипотезы, в будущем его могут переписать или совсем удалить – это зависит от востребованности функциональности пользователями.

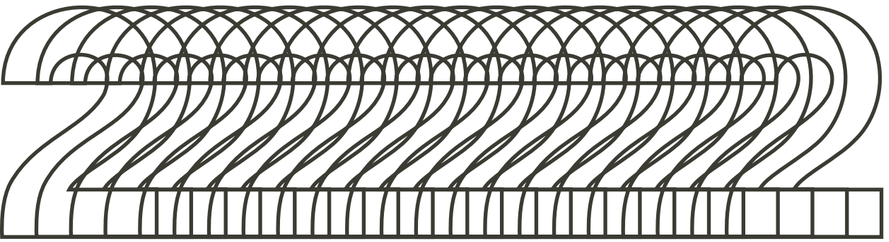
→ **Когда решение сильно усложняет разработку и поддержку.** Большое число классов может создавать сложности в чтении и разборе кода. Для улучшения лаконичности, можно не так сильно дробить классы на сущности.

→ **Архитектурные решения, которые предопределяют реализацию, несмотря на каноны и принципы.** В монолитных приложениях часто код разных команд пересекается, на первых этапах можно пренебречь некоторыми принципами.

еще два важных правила по поддержании кода в чистоте



Структурируйте код. Группируйте элементы исходя из причины их изменений.



Классами с одной обязанностью легко оперировать, их просто модифицировать.


avito.tech