Научно-практический электронный журнал
Оригинальные исследования (ОРИС), *ISSN 2222-0402*
www.ores.su
original-research.ru

УДК 004.415.5

МЕТОДИКА ПРИМЕНЕНИЯ SOLID-ПРИНЦИПОВ В РАЗРАБОТКЕ АВТОТЕСТОВ

Васильев Борис Яковлевич,
магистр, инженер по тестированию программного обеспечения, ЮЭсТи, Бойнтон-Бич, Флорида, США
boris.vasilev.qa@gmail.com

Аннотация

В статье представлена методика применения принципов SOLID при разработке автоматизированных тестов. SOLID — это хорошо известный набор принципов проектирования (единственная ответственность, открытость/закрытость, подстановка Лисков, разделение интерфейсов, инверсия зависимостей), улучшающих модульность, расширяемость и обслуживаемость объектно-ориентированных систем. В работе рассматривается, как каждый из этих принципов может быть использован в автоматизации тестирования для создания гибкого и устойчивого тестового кода. Приводятся примеры на языках Python, JavaScript и C#, иллюстрирующие корректное применение принципов в тестовых сценариях. Следование SOLID при построении архитектуры автотестов позволяет формировать более чистые тестовые наборы, которые легче расширять новыми проверками и которые меньше подвержены поломкам при изменении боевого кода. Особое внимание уделено преимуществам SOLID-ориентированного подхода к проектированию тестов — таким как улучшение читаемости, повторного использования компонентов и стабильности тестов. Статья адресована опытным специалистам в области тестирования программного обеспечения и разработчикам, заинтересованным в продвинутых практиках автоматизации.

Ключевые слова: SOLID-принципы, автоматизированное тестирование, архитектура автотестов, объектно-ориентированное проектирование, обслуживаемость тестов, тестовый код.

METHODOLOGY FOR APPLYING SOLID PRINCIPLES IN AUTOMATED TEST DEVELOPMENT

Boris Ya. Vasilev,
master, software quality assurance engineer, UST,
Boynton Beach, USA

ABSTRACT

This article presents an approach for applying SOLID principles in the development of automated tests. SOLID is a well-known set of design principles (Single Responsibility, Open-

Closed, Liskov Substitution, Interface Segregation, Dependency Inversion) that improve the modularity, extensibility, and maintainability of object-oriented systems. The paper discusses how each of these principles can be utilized in test automation to create flexible and robust test code. We provide examples in Python, JavaScript, and C# to illustrate correct applications of each principle in test scenarios. Adhering to SOLID in test architecture leads to cleaner test suites that are easier to extend with new tests and less prone to breakage when production code changes. The advantages of a SOLID-compliant test design—such as improved readability, reusability of test components, and more stable test suites—are highlighted. The article is intended for experienced software testers and developers interested in advanced test automation practices.

Keywords: SOLID principles, automated testing, test architecture, object-oriented design, test maintainability, test code.

Введение

В современном процессе разработки программного обеспечения автоматизированное тестирование играет ключевую роль в обеспечении качества продукта. Автотесты (написанные сценарии, проверяющие корректность работы кода) – это программные модули, требующие грамотной архитектуры и поддержки. Хотя тестовый код не входит непосредственно в поставляемый продукт, его качество влияет на скорость разработки и сопровождения системы. Неудачная структура автотестов приводит к ложным срабатываниям, сложностям при добавлении новых проверок и росту затрат на поддержку тестового комплекта.

Принципы SOLID, предложенные Р. Мартином в начале 2000-х годов [1], предназначены для улучшения архитектуры объектно-ориентированных систем. Аббревиатура расшифровывается как Single Responsibility (единственная ответственность), Open-Closed (открытость/закрытость), Liskov Substitution (подстановка Лисков), Interface Segregation (разделение интерфейсов) и Dependency Inversion (инверсия зависимостей). Соблюдение SOLID повышает модульность системы и упрощает внесение изменений без нарушения уже работающего кода [1].

Цель статьи – показать, как SOLID-принципы могут применяться при разработке автоматизированных тестов. Тестовый код обладает особыми требованиями: он часто изменяется вслед за развитием продукта, должен быть наглядным и изолированным от внешних зависимостей. В статье рассмотрено применение каждого SOLID-принципа применительно к автотестам, приведены примеры кода на Python, JavaScript и C#, демонстрирующие корректное следование принципам, а также обсуждены преимущества такого подхода. Отдельное внимание уделено типичным ошибкам (анти-паттернам) в тестовом коде при несоблюдении SOLID и способам их избежать.

Краткий обзор принципов SOLID

SOLID – акроним пяти базовых принципов объектно-ориентированного проектирования [1]. Следование этим принципам повышает сцепление (cohesion) внутри модулей и ослабляет связанность (coupling) между ними, делая систему понятной и гибкой [1]. Перечислим принципы SOLID с краткими определениями:

- S – Single Responsibility Principle (SRP), принцип единственной ответственности: каждый класс или модуль должен решать только одну задачу и иметь единственную причину для изменения [1]. Это упрощает поддержку: изменение требования затрагивает минимальную часть системы. При

нарушении SRP модуль выполняет несколько функций, и изменение одной приводит к непредвиденным последствиям для других.

- – Open/Closed Principle (OCP), принцип открытости/закрытости: программные сущности открыты для расширения, но закрыты для изменения [1]. Новый функционал вводится через добавление нового кода, а не изменение существующего. Достигается применением абстракций и полиморфизма. Преимущество – устойчивость отлаженного кода к доработкам, снижение риска регрессий.
- L – Liskov Substitution Principle (LSP), принцип подстановки Лисков: объекты подклассов должны быть полностью заменяемы на объекты базового класса без нарушения логики [2]. Наследники обязаны соблюдать контракт поведения базового класса. Соблюдение LSP гарантирует корректную работу полиморфизма; нарушение проявляется, когда подкласс не выполняет ожидания базового интерфейса, из-за чего использование его вместо базового приводит к ошибкам.
- I – Interface Segregation Principle (ISP), принцип разделения интерфейсов: не следует заставлять клиента зависеть от методов, которыми он не пользуется [1]. Вместо одного «толстого» интерфейса лучше множество небольших. Это означает, что классам и тестам предоставляются только необходимые им интерфейсы, и изменение в одном интерфейсе не затрагивает клиентов, которые от него не зависят.
- D – Dependency Inversion Principle (DIP), принцип инверсии зависимостей: высокоуровневые модули не должны зависеть от низкоуровневых, оба должны зависеть от абстракций; абстракции не зависят от деталей, детали зависят от абстракций [1]. Проще говоря, компоненты должны общаться через интерфейсы. DIP снижает связанность системы и облегчает замену реализаций без изменения кода, который на них полагается. Для тестирования DIP критичен, т.к. позволяет вместо реальных компонентов подставлять имитации (mock/stub) [3].

В сочетании эти принципы дают модульную и расширяемую структуру. Далее рассмотрим, какие особенности имеют автотесты и как SOLID помогает их эффективно разрабатывать.

Особенности автотестов как объекта разработки

Автоматизированные тесты – это код, проверяющий другой код. Их назначение специфично, но разрабатываться они должны с применением инженерных практик, подобных промышленному коду. Отметим важные особенности автотестов:

Читаемость и ясность. Автотесты служат живой спецификацией системы, поэтому должны быть предельно понятны. Каждый тестовый сценарий должен явно отражать проверяемое требование, избегая сложной логики. Понятные тесты проще поддерживать и исправлять при падениях.

Частое изменение. Тесты эволюционируют вместе с системой: новые функции требуют новых тестов, изменение требований ведет к изменению тестов. Значит, тестовая архитектура должна легко адаптироваться к изменениям, позволяя добавлять новые сценарии без ломки существующего кода.

Повторяемость и изоляция. Хорошие тесты изолированы от внешних влияний и друг от друга. Они должны давать воспроизводимый результат независимо от порядка выполнения. Для этого тесты запускаются на контролируемых данных, а внешние зависимости (базы, сервисы) заменяются заглушками. Изоляция увеличивает скорость и надежность прогона тестов [4].

Повторное использование кода. Повторяющиеся шаги тестов выносятся во вспомогательные методы и классы (например, паттерн Page Object [4]), что устраняет дублирование и упрощает модификации.

Иные приоритеты качества. Для тестового кода важнее простота и надежность, чем производительность. Допустимо небольшое дублирование ради лучшей читаемости (принцип DAMP вместо строгого DRY). Однако избыточное копирование усложняет поддержку: при изменении функциональности требуется поправить множество мест. SOLID-принципы позволяют найти баланс между чистотой и простотой тестового кода.

Применение принципов SOLID в автотестах

Рассмотрим каждый SOLID-принцип применительно к автоматизированным тестам, приведем примеры его использования и типичные ошибки нарушения.

S: Принцип единственной ответственности (SRP)

Определение: каждый тест или вспомогательный модуль тестового кода должен отвечать за что-то одно. Один тест – один сценарий.

Применение SRP в тестах: Каждый тестовый метод должен проверять один аспект поведения. Если тест выполняет несколько несвязанных действий, его следует разбить на несколько. Например, вместо одного теста, который регистрирует пользователя, потом логинится им и затем делает заказ, лучше написать три отдельных теста. Тогда при падении сразу понятно, какой функциональный блок неисправен.

Также SRP касается организации кода: тесты группируются по классам/модулям в соответствии с функциональностью (например, TestLogin, TestOrders и т.д.). Вспомогательные действия (настройка данных, очистка окружения) выносятся во внешние функции или фикстуры, чтобы сам тест занимался только проверкой.

Пример (Python/PyTest): до и после применения SRP:

python

Copy

```
# Хороший пример: отдельные тесты для регистрации и входа
def test_user_registration():
    user = register_new_user("bob", "qwerty")
    assert user.is_active

def test_user_login():
    session = login ("bob", "qwerty») # пользователь "bob" уже существует
    assert session.token is not None
```

В исправленном варианте два разных теста выполняют отдельные задачи: один проверяет регистрацию пользователя (что новый пользователь активируется), другой – вход (что выдается токен сессии). Изменение логики регистрации затронет только первый тест, а изменения процесса логина – только второй. Тесты стали короче, понятнее и надежнее.

SRP реализуется и на уровне тестового фреймворка. Например, паттерн Page Object следует SRP: каждый класс страницы веб-приложения отвечает только за операции на этой странице и ни за что больше [4]. Изменение интерфейса страницы логина потребует исправить лишь класс LoginPage, не затрагивая тесты профиля, поиска и т.д. Это упрощает поддержку всего набора тестов.

Нарушения SRP в тестах: длинные тесты-комбайны; один класс тестов покрывает разнородные функции; тесты содержат лишнюю логику подготовки. Решение – рефакторить: дробить тесты, выносить общие части, разделять ответственность.

O: Принцип открытости/закрытости (OCP)

Определение: тестовый код должен быть открыт для добавления новых сценариев, но закрыт для изменения существующих сценариев.

Применение OCP в тестах: При появлении нового функционала мы добавляем новые тесты, а не переписываем старые. Старые тесты фиксируют реализованные требования и должны оставаться неизменными, пока поведение системы соответствует прежним ожиданиям. Это гарантирует, что мы не пропустим регрессию: если новое изменение сломало старую функцию, упадет старый тест.

На практике следует изначально проектировать тесты с расчетом на расширение. Например, если добавилась новая опция или кейс, лучше написать отдельный тест-кейс (или новый параметр для параметризованного теста), а не вносить дополнительные условия в существующий тест.

Пример (JavaScript, Cypress): сначала был тест поисковой функции, затем добавилась фильтрация результатов:

```
javascript
Copy
// Новый тест: проверка фильтрации поиска
it ('should return filtered results when category filter is applied', () => {
  cy.visit('/search');
  cy.get('input[name="query"]'). type ('SOLID {enter}');
  cy.get('#filter-category'). select('Books');
  cy.get('. results'). should ('contain', 'SOLID Principles Book');
  cy.get('. results'). should ('not.contain', 'SOLID Principles Blog');
});
```

Предполагается, что базовый тест поиска уже существует и остался без изменений (проверяет, что поиск по слову "SOLID" выдает результаты). Для новой возможности – фильтра по категории – добавлен отдельный тест. Благодаря этому базовый сценарий по-прежнему покрыт старым тестом, а новая функциональность – новым. Если фильтр сломает базовый поиск, сигнал даст старый тест; если не работает только фильтрация, упадет новый тест. В обоих случаях мы точно знаем, что именно нарушилось.

Следование OCP тесно связано с поддержкой регрессионного набора: тесты не переписываются под новые требования, а отражают их добавлением новых случаев [5]. Если изменение требований все же меняет поведение, закрепленное тестом (например, старая функция теперь официально работает иначе), то тесты обновляются сознательно, но это не должно происходить тихо. В целом же, при расширении системы лучше добавлять, а не заменять тесты.

L: Принцип подстановки Лисков (LSP)

Определение: если модуль работает с объектом базового класса, то он должен успешно работать и с любым его подклассом. Подклассы не должны предъявлять дополнительных требований к коду, использующему базовый тип.

Применение LSP в тестах: В тестовом коде этот принцип актуален в двух аспектах. Во-первых, если тестовый фреймворк использует наследование (например, базовый класс BaseTest с общим setUp(), от которого наследуются конкретные тесты), то наследники не должны нарушать предусловия базового класса. Они либо используют базовую инициализацию как есть, либо расширяют ее, но не отменяют. Иначе, запуская базовые тесты в контексте наследника, мы можем получить неверное поведение. Правило:

наследование в тестах не должно менять базовых ожиданий. Если подклассу нужно совсем другое окружение, лучше не наследоваться, а построить тест иначе.

Во-вторых, принцип LSP важен при использовании тестовых двойников – mock- и stub-объектов. Имитации должны соответствовать контракту реальных компонентов. Если метод реального объекта в определенной ситуации бросает исключение, то и заглушка должна это делать; если реальный метод возвращает определенную структуру, mock должен возвращать эквивалентную структуру. Несоблюдение этого приводит к тому, что тест «зелёный» на имитации, а в реальном окружении код падает. Придерживаясь LSP, мы в тестах полагаемся на интерфейсы и проверяем поведение на уровне интерфейса, не делая предположений, зависящих от конкретной реализации.

Как практически проверить LSP? Порой создают набор тестов для интерфейса, который прогоняется на разных реализациях. Например, тесты для интерфейса хранилища данных могут запускаться на реальной базе, на in-memory реализации и на файковой – и должны проходить во всех случаях. Такой подход (описанный, например, у Б. Лисков [2]) гарантирует заменяемость реализаций: если новый подкласс не проходит старые тесты, это сигнал несоблюдения LSP.

Нарушения LSP в тестах: сложные схемы наследования, где подклассы тестов переопределяют базовые методы и меняют логику (например, не вызывают setUp родителя), использование mock-объектов, поведение которых не соответствует реальным. Исправляется соблюдением контрактов: делать наследование простым, а моки – максимально приближенными к оригиналу.

I: Принцип разделения интерфейсов (ISP)

Определение: тесты (как клиенты) должны зависеть только от тех методов, которые им нужны. Нежелательно предоставлять тестам чрезмерно общие интерфейсы с лишней функциональностью.

Применение ISP в тестовой архитектуре: при организации вспомогательных утилит для тестов полезно разбивать их по зонам ответственности. Вместо одного класса TestHelper с десятком методов (инициализация БД, логин, отправка API-запроса и т.д.) лучше сделать отдельные компоненты: DbHelper, AuthHelper, ApiHelper и т.п. Тогда тесты авторизации используют только AuthHelper и не зависят от методов для базы, а тесты базы не знают ничего про UI или API. Изменения в модуле помощи по базе не потребуют пересмотра тестов, которые базу не используют.

Аналогично, в самих тестах следует избегать сценариев, покрывающих сразу несколько подсистем. Лучше написать два теста (как в примере ниже), чем один, который сначала проверяет вход, а затем – профиль пользователя. Так каждый тест “подключается” только к нужному интерфейсу (в одном случае – интерфейс страницы логина, в другом – интерфейс профиля).

Пример (JavaScript, Cypress): разделение сценариев логина и редактирования профиля:

javascript
Copy

```
// Редактирование профиля (после логина)
it ('should update the profile successfully', () => {
  cy.login ('user', 'pass'); // кастомная команда для авторизации
  cy.get('#profile-link'). click ();
  cy.get('input[name="name"]'). clear (). type ('New Name');
  cy.get('button[type="save"]'). click ();
  cy.get(‘. success-message'). should ('contain', 'Profile updated');
});
```

Вместо одного теста, который сначала логинится, а потом меняет профиль, логин вынесен в отдельную команду cy.login (реализованную, к примеру, через Cypress Commands). Таким образом, тест профиля сфокусирован только на операции профиля и не содержит лишних шагов. Если изменится процедура входа, исправляем команду cy.login и тесты авторизации, но тесты профиля трогать не придется. Если изменится страница профиля, затронуты будут только тесты профиля. Это и есть цель ISP – узкие, конкретные интерфейсы для каждой задачи.

Наконец, ISP улучшает повторное использование: например, команда cy.login может применяться в десятках разных тестов. Она представляет собой небольшой интерфейс (метод с двумя параметрами), который удобно поддерживать централизованно.

Нарушения ISP: монолитные классы утилит; тесты, охватывающие сразу несколько разнородных действий; чрезмерно длинные сценарии. Лечение – разделять интерфейсы: декомпозировать утилиты на более мелкие, разносить проверки по разным тестам. Тогда тесты получатся короче, понятнее и независимее.

D: Принцип инверсии зависимостей (DIP)

Определение: и тестируемый код, и сами тесты должны зависеть от абстракций, а не от конкретных классов. Реализации подставляются через эти абстракции.

Применение DIP в тестировании: Это, пожалуй, самый важный принцип для обеспечения тестопригодности. DIP проявляется на двух уровнях:

В коде приложения (SUT): если система спроектирована с учетом DIP, её компоненты легко изолировать при тестировании. Класс бизнес-логики не создаёт сам экземпляры сервисов, а получает интерфейс. Тогда в тесте мы можем передать вместо реального сервиса заглушку, реализующую тот же интерфейс [3]. Без DIP тест был бы вынужден использовать настоящий сервис, что усложняет настройку и может приводить к нестабильности.

В коде тестов: сами тесты и вспомогательные модули тоже желательно писать через интерфейсы. Например, тесты взаимодействуют не прямо с реальной базой, а через интерфейс IDatabase. На время теста можно подставить in-memory реализацию, а в интеграционном окружении – настоящий драйвер базы. Тестовый код при этом не меняется. Аналогично для внешних API: вместо вызова HTTP в тесте можно внедрить интерфейс клиента API, и подать ему mock-реализацию.

Смысл DIP – убрать жесткую связность. Ни один высокоуровневый тест не должен напрямую создавать низкоуровневый ресурс. Всегда есть прослойка: параметр, интерфейс, фабрика, фикстура – позволяющая подменить реализацию.

Пример (C#): класс PaymentService зависит от платежного шлюза. Правильная реализация с DIP:

```
csharp
Copy
public interface IPaymentGateway {
   bool Process (Order order);
}
class PaymentService {
   private IPaymentGateway gateway;
   public PaymentService(IPaymentGateway gateway) {
      this.gateway = gateway;
   }
   public bool MakePayment(Order order) {
      return gateway.Process(order);
   }
}
```

В приложении PaymentService получит реальный PayPalGateway : IPaymentGateway, а в тесте – тестовый двойник:

```
csharp
Copy
class FakeGateway : IPaymentGateway {
   public bool Process (Order order) => true; // всегда успешно
}
[Test]
public void TestPaymentSuccess() {
   var service = new PaymentService(new FakeGateway());
   bool result = service.MakePayment(new Order(...));
   Assert.IsTrue(result);
}
```

Здесь тест полностью контролирует окружение: вместо реального платежного сервиса используется FakeGateway, который, например, всегда возвращает успех. Тест проверяет логику PaymentService (что он корректно обрабатывает результат), не полагаясь на внешний мир. Если нужно проверить негативный сценарий, FakeGateway можно изменить на возврат false. Обратите внимание: для этого не пришлось модифицировать код PaymentService – он изначально спроектирован гибко благодаря DIP.

DIP лежит в основе практики мокирования: инструменты типа Mockito или Moq позволяют подменять реализации интерфейсов на лету. Но они бесполезны, если код жестко привязан к конкретным классам. Поэтому DIP часто рассматривается как необходимое условие эффективного unit-тестирования [3][6].

Нарушения DIP: тесты напрямую вызывают глобальные синглтоны, работают с конкретными классами, требуют реальных ресурсов. Такой код труднее изолировать и сделать детерминированным. Решение – рефакторинг с введением уровня абстракции. Хотя переписывание существующего кода под DIP может быть трудоемким, практика показывает, что это окупается за счет резкого улучшения тестируемости системы [6].

Преимущества применения SOLID в тестовой архитектуре

Применение SOLID-принципов при разработке автотестов дает ряд ощутимых плюсов [1][4][5]:

Модульность. SOLID снижает связанность тестов: они не зависят друг от друга и легко изолируются от деталей.

Читаемость. Структура тестов понятна, сценарии короткие и специализированные – их проще понимать и править.

Расширяемость. Новые тесты можно добавлять, не изменяя имеющиеся (соответствие OCP), что упрощает эволюцию тестового покрытия.

Надежность. Изменения в коде реже ломают тесты; моки и стабы (DIP) делают прогоны детерминированными.

Отладка. Если тест падает, узкий фокус помогает быстро локализовать проблему; рефакторинг тестов локален.

Масштабируемость. SOLID-архитектура позволяет эффективно организовать сотни и тысячи тестов без хаоса в коде.

Суммарно, SOLID-принципы делают тестовый код чистым, гибким и живучим [7]. Это снижает стоимость поддержки автотестов и напрямую способствует качеству продукта: качественные тесты быстрее обнаруживают дефекты, их проще поддерживать в актуальном состоянии и они не замедляют процесс разработки.

Типичные ошибки при нарушении SOLID в тестах

Нарушение принципов SOLID проявляется в характерных «запахах» тестового кода [4][6]. Некоторые распространенные проблемы:

«Божественный» тест (God Test): слишком длинный сценарий, проверяющий многое за один проход. Его сложно поддерживать, он нарушает SRP/ISP и часто падает по разным причинам.

Дублирование шагов: одни и те же фрагменты кода (настройка, проверки) копируются в нескольких тестах (нарушение SRP/DIP). Это увеличивает трудоемкость изменений и риск ошибок. Лучше вынести повторяющийся код во вспомогательные методы или фикстуры.

Тесная связь с окружением: тест зависит от конкретной БД, файловой системы или внешних API (нарушение DIP). В другом окружении он не работает или дает иной результат. Следует использовать конфигурируемые параметры, моки и заглушки, чтобы отвязать тест от окружения.

Чрезмерное наследование тестов: сложная иерархия классов, где подклассы переопределяют базовое поведение (риск нарушения LSP). Это запутывает исполнение и усложняет поддержку. Предпочтительнее композиция или неглубокое наследование без изменения контрактов.

Зависимые тесты: порядок выполнения влияет на результат (например, один тест создает данные, другой их использует – нарушение SRP). Все тесты должны быть автономными; если требуется общий контекст, он готовится явно (в setUp или общей фикстуре), а не в другом тесте.

Выявив такие проблемы, нужно рефакторить тесты, руководствуясь принципами SOLID и шаблонами рефакторинга кода [6]. Опыт показывает, что улучшение архитектуры тестов существенно экономит время: снижается частота ложных срабатываний, уменьшается объем правок при изменении требований, а значит, повышается эффективность всего процесса разработки ПО.

Заключение

Принципы SOLID изначально разрабатывались для промышленного кода, однако они в равной степени полезны и в архитектуре автоматизированных тестов. Применяя SOLID, инженеры по качеству получают тесты, которые легко читать, удобно расширять и надежно выполнять на разных этапах жизненного цикла продукта. Рассмотрев применение каждого принципа – от SRP до DIP – можно резюмировать:

SOLID-принципы помогают сделать тесты более модульными: каждый тест автономен функционально (SRP, ISP) и по зависимостям (DIP), что упрощает отладку и поддержку.

Соблюдение SOLID ведет к устойчивости тестового набора к изменениям: новые тесты добавляются без изменения старых (OCP), а изменение кода приложения требует минимальных правок в тестах благодаря хорошей абстракции.

Применение SOLID снижает количество типичных ошибок в тестировании, таких как дублирование сценариев, зависимые или хрупкие тесты, чрезмерно сложный в сопровождении код тестовой логики.

Инвестиции в чистую архитектуру тестов окупаются уменьшением времени на поддержку: разработчики тратят меньше усилий на исправление самих тестов и могут сосредоточиться на улучшении продукта.

Методика разработки автотестов на основе SOLID-принципов повышает эффективность процесса тестирования и надежность программного обеспечения. Внедрение этих принципов рекомендуется как часть общей культуры разработки качественного кода в проекте.

Список литературы:

1. Мартин Р. Чистая архитектура: искусство разработки программного обеспечения – Питер, 2019. ISBN: 978-5-4461-0772-8
2. Liskov B. Data Abstraction and Hierarchy. – SIGPLAN Notices, 23(5), 1987, https://dl.acm.org/doi/10.1145/62139.62141, Стр. 17–34.
3. Martin R. C. The Dependency Inversion Principle. – C++ Report, Vol. 8, No. 6, 1996, Стр. 61–72.
4. Месарош Д. Шаблоны тестирования xUnit. Рефакторинг кода тестов – Вильямс 2009. ISBN: 978-5-8459-1448-4
5. Хориков В. Принципы юнит-тестирования – Manning, 2020. ISBN: 978-5-4461-1683-6
6. Feathers M. Working Effectively with Legacy Code. – Prentice Hall, 2005. ISBN: 978-0-1311-7705-5
7. van Deursen A., Moonen L., van den Bergh A., Kok G. Refactoring Test Code. – Proc. 2nd Int. Conf. Extreme Programming, 2001, Стр. 92–95.

References:

1. Martin R. Clean Architecture: The Art of Software Engineering – Peter, 2019. ISBN: 978-5-4461-0772-8
2. Liskov B. Data Abstraction and Hierarchy. – SIGPLAN Notices, 23(5), 1987, https://dl.acm.org/doi/10.1145/62139.62141, pp. 17–34.
3. Martin R. C. The Dependency Inversion Principle. – C++ Report, Vol. 8, No. 6, 1996, pp. 61–72.
4. Meszaros D. xUnit Testing Patterns. Refactoring Test Code – Williams 2009. ISBN: 978-5-8459-1448-4
5. Khorikov V. Unit Testing Principles – Manning, 2020. ISBN:978-5-4461-1683-6
6. Feathers M. Working Effectively with Legacy Code. – Prentice Hall, 2005. ISBN: 978-0-1311-7705-5
7. van Deursen A., Moonen L., van den Bergh A., Kok G. Refactoring Test Code. – Proc. 2nd Int. Conf. Extreme Programming, 2001, pp. 92–95.