5. Принципы построения кода в ООП

Как правильно использовать ООП

Хорошее ООП

- снижает сложность
- привносит модульность
- повышает совместимость
- локализует изменения
- абстрагирует от реализации
- упрощает код
- легко тестируется

Плохое ООП

- привносит сложность
- сохраняет процедурный подход
- причиняет неудобство

Принцип DRY

DRY (Don't Repeat Yourself)

Выявляйте одинаковые программные сущности и объединяйте их.

Каждая часть знаний или логики должна быть представлена в системе только один раз.
Это:

- помогает избежать дублирования кода
- упрощает поддержку
- снижает вероятность ошибок.

Механизмы реализации DRY в ООП

- **Наследование**. Создание новых классов на основе существующих.
- **Трейты (примеси)**. Добавление повторно используемой функциональности в классы без наследования.
- **Композиция**. Использование объектов других классов внутри текущего класса для делегирования задач.
- **Полиморфизм**. Использование единого интерфейса для работы с объектами разных классов.
- **Служебные классы**. Классы со статическими методами для общих операций.

DRY и наследование

```
class User {
    public function getFullName($firstName, $lastName) {
        return $firstName . ' ' . $lastName;
    }
}

class Admin {
    public function getFullName($firstName, $lastName) {
        return $firstName . ' ' . $lastName; // Повторяем
    }
}
```

Здесь метод `getFullName` дублируется в обоих классах. Чтобы избежать этого, можно вынести общий функционал в базовый класс:

```
class Person {
    public function getFullName($firstName, $lastName) {
        return $firstName . ' ' . $lastName;
    }
}

class User extends Person {}

class Admin extends Person {}
```

DRY и трейты

```php
trait Logger {
    public function log($message) {
        echo "Logging: $message\n";
    }
}

class UserService {
    use Logger;

    public function createUser($name) {
        $this->log("Creating user: $name");
        // Логика создания пользователя
    }
}

class OrderService {
    use Logger;

    public function createOrder($product) {
        $this->log("Creating order for product: $product");
        // Логика создания заказа
    }
}
```

DRY и конфигурации

```php
function connectToDatabase() {
    $host = 'localhost';
    $username = 'root';
    $password = 'password';
    $database = 'my_database';

    // Логика подключения к базе данных
}

function logError($message) {
    file_put_contents('logs/error.log', $message, FILE_APPEND);
}
```

Здесь строки `'localhost'`, `'root'`, `'password'`, `'my_database'` и `'logs/error.log'` могут быть использованы в разных частях программы. Чтобы избежать дублирования, их можно вынести в конфигурационный файл:

DRY и конфигурации

```
// config.php
return [
    'db' => [
        'host' => 'localhost',
        'username' => 'root',
        'password' => 'password',
        'database' => 'my_database',
    ],
    'log_file' => 'logs/error.log',
];

// index.php
$config = require 'config.php';

function connectToDatabase($config) {
    $host = $config['db']['host'];
    $username = $config['db']['username'];
    $password = $config['db']['password'];
    $database = $config['db']['database'];

    // Логика подключения к базе данных
}

function logError($message, $config) {
    file_put_contents($config['log_file'], $message, FILE_APPEND);
}
```

DRY и конфигурации

1. Создание статического класса `StaticConfig`

```php
class StaticConfig {
    // Статические свойства для хранения конфигурации
    public static $dbHost = 'localhost';
    public static $dbUsername = 'root';
    public static $dbPassword = 'password';
    public static $dbDatabase = 'my_database';
    public static $logFile = 'logs/error.log';
}
```

DRY и конфигурации

2. Использование статического класса в функциях

php Копироват

```php
function connectToDatabase() {
    $host = StaticConfig::$dbHost;
    $username = StaticConfig::$dbUsername;
    $password = StaticConfig::$dbPassword;
    $database = StaticConfig::$dbDatabase;

    // Логика подключения к базе данных
    echo "Connecting to database: $host, $username, $password, $database\n";
}

function logError($message) {
    $logFile = StaticConfig::$logFile;

    file_put_contents($logFile, $message, FILE_APPEND);
}
```

DRY и конфигурации

Когда использовать массивы?

- Для простых конфигураций.
- Если конфигурация динамически загружается из внешних источников (JSON, YAML, БД).
- В небольших проектах или скриптах.

Когда использовать статические классы?

- Для фиксированных конфигураций с четкой структурой.
- В крупных проектах, где важны типобезопасность и автозаполнение.
- Если нужно защитить данные от случайного изменения.

Принципы SOLID

Принципы SOLID — это пять основных принципов ООП, сформулированных Робертом Мартином, для создания гибких, поддерживаемых и масштабируемых систем.

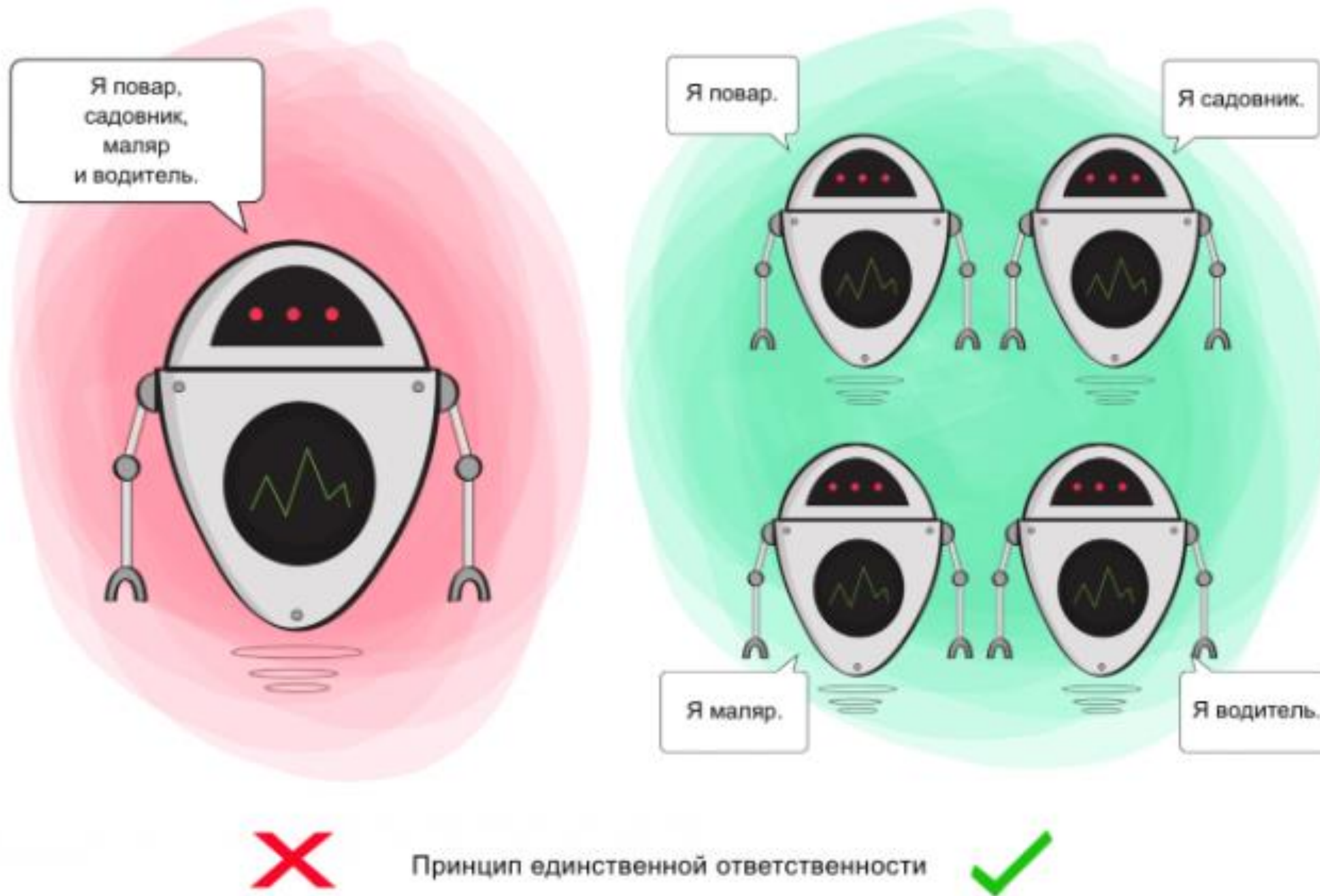
Single Responsibility Principle (SRP)

Single Responsibility Principle

У программной сущности должна быть только одна причина для изменений.

Каждый класс должен решать только одну задачу. Если класс выполняет несколько функций, его сложно поддерживать и тестировать.

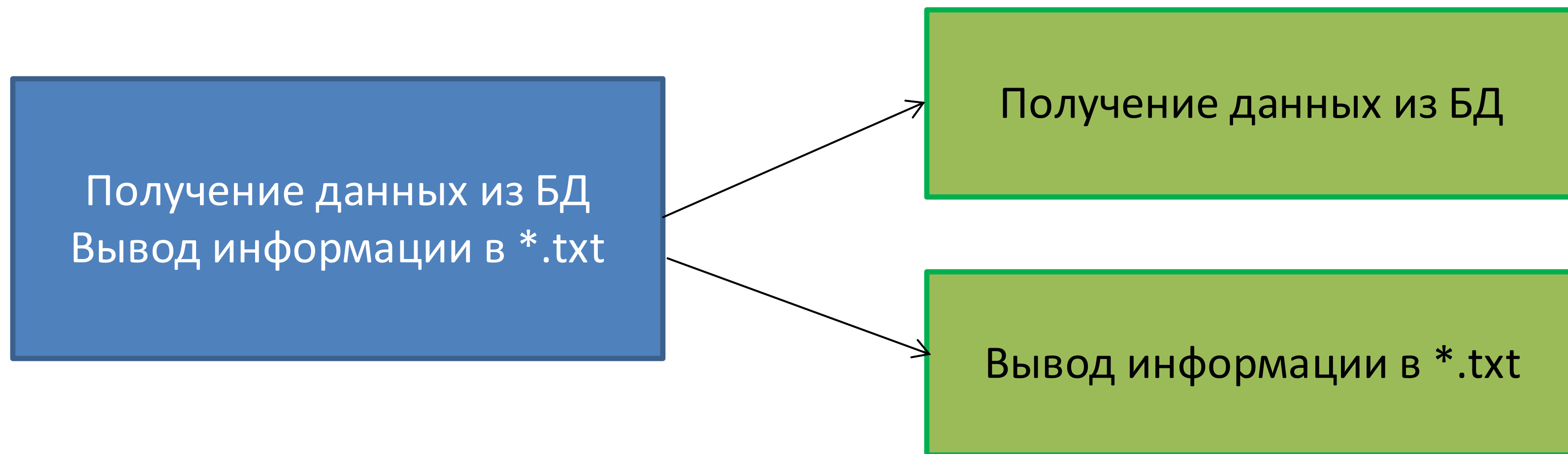
Single Responsibility Principle



Если класс отвечает за несколько операций сразу, вероятность возникновения багов возрастает – внося изменения, касающиеся одной из операций вы, сами того не подозревая, можете затронуть и другие.

Single Responsibility Principle

У программной сущности должна быть только одна причина для изменений.



Single Responsibility Principle

У программной сущности должна быть только одна причина для изменений.

```
// Плохо: Класс выполняет две задачи — работу с БД и логгирование.
class UserManager {
    public function createUser($data) {
        // Логика создания пользователя
    }

    public function logActivity($message) {
        // Логика записи в лог
    }
}
```

Single Responsibility Principle

У программной сущности должна быть только одна причина для изменений.

```
// Хорошо: Разделяем ответственности
class UserRepository {
    public function createUser($data) {
        // Работа с БД
    }
}

class Logger {
    public function log($message) {
        // Логирование
    }
}
```

SRP и отделение чистого кода

Чистая функция — это функция, которая:

1. **Детерминирована** : Для одних и тех же входных данных возвращает одинаковый результат.
2. **Не имеет побочных эффектов** : Не изменяет состояние системы, не модифицирует глобальные переменные, не выполняет ввод-вывод (например, работу с БД, API, файлами).

Пример 1: Чистая функция

php Копировать

```php
function square(int $x): int {
    return $x * $x;
}
```

- **Почему чистая** : Всегда возвращает одинаковый результат для одного и того же `$x`, не зависит от внешних данных и не изменяет их.

SRP и отделение чистого кода

Пример 2: Нечистая функция (побочный эффект)

php Копировать

```php
$counter = 0;

function incrementCounter(): int {
    global $counter;
    return ++$counter; // Изменяет глобальную переменную
}
```

- **Почему нечистая** : Зависит от и изменяет внешнее состояние (`$counter`).

Пример 3: Нечистая функция (недетерминированность)

php Копировать

```php
function getCurrentTime(): string {
    return date('H:i:s'); // Зависит от текущего времени
}
```

- **Почему нечистая** : Результат зависит от внешнего фактора (времени), даже если нет явных побочных эффектов.

SRP и отделение чистого кода

Чистая функция — это детерминированная функция без побочных эффектов. Она тесно связана с принципом SRP, так как фокусируется на выполнении одной задачи, что упрощает разделение ответственностей в коде.

Упрощают разделение логики и действий :

- Чистые функции занимаются **вычислениями** .
- Побочные эффекты (логгирование, работа с БД) выносятся в отдельные функции/классы.

Преимущества чистых функций

1. **Прогнозируемость** : Легко тестировать (нет зависимости от внешних факторов).
2. **Параллелизм** : Безопасны для многопоточного выполнения (нет изменяемого состояния).
3. **Поддерживаемость** : Изменения в одной чистой функции не влияют на другие части системы.

SRP и отделение чистого кода

Неправильно (нарушение SRP и нечистая функция) :

php Копировать

```php
function calculateAndSaveDiscount(int $price, int $discountPercent): void {
    $discountedPrice = $price * (1 - $discountPercent / 100);
    // Побочный эффект: работа с БД
    Database::save('discounts', ['price' => $discountedPrice]);
}
```

Правильно :

php Копировать

```php
// Чистая функция (SRP: только расчет)
function calculateDiscount(int $price, int $discountPercent): int {
    return $price * (1 - $discountPercent / 100);
}

// Отдельная функция для работы с БД (SRP: сохранение данных)
function saveDiscount(int $discountedPrice): void {
    Database::save('discounts', ['price' => $discountedPrice]);
}
```

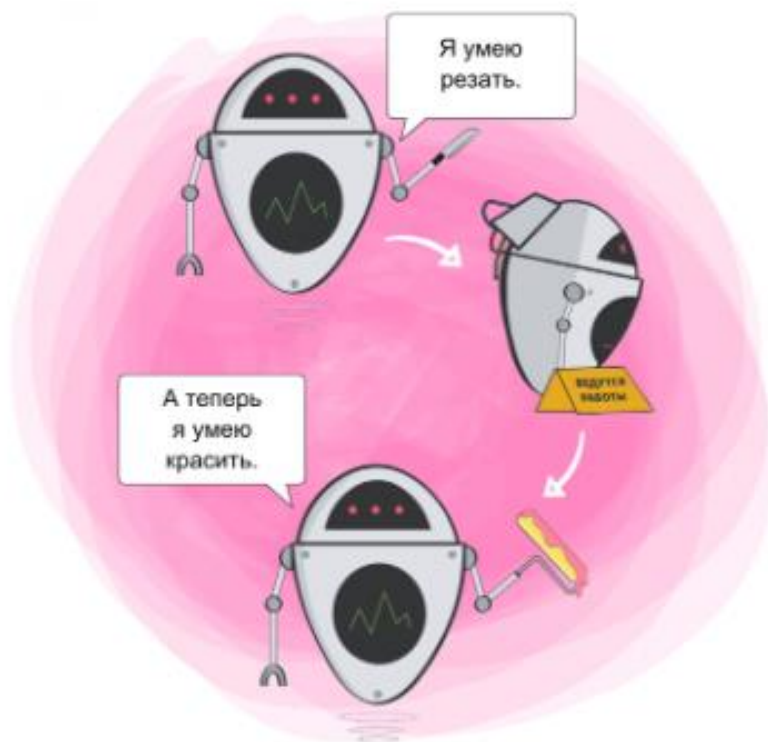
The Open-Closed Principle (OCP)

The Open-Closed Principle

Программные сущности (классы, модули, функции) должны быть открыты для расширения, но закрыты для модификации.

Изменение функционала должно происходить через добавление нового кода, а не изменение существующего.

The Open-Closed Principle



Принцип открытости/закрытости

Когда вы меняете текущее поведение класса, эти изменения сказываются на всех системах, работающих с данным классом. Если хотите, чтобы класс выполнял больше операций, то идеальный вариант – не заменять старые на новые, а добавлять новые к уже существующим.

The Open-Closed Principle

Программные сущности должны быть:

- открыты для расширения (*наследование, полиморфизм, композиция*)
- закрыты для модификации (*инкапсуляция*).

The Open-Closed Principle

```php
// Плохо: При добавлении новой фигуры нужно менять класс AreaCalculator
class AreaCalculator {
    public function calculate($shapes) {
        foreach ($shapes as $shape) {
            if ($shape instanceof Square) {
                $area += $shape->side * $shape->side;
            } elseif ($shape instanceof Circle) {
                $area += M_PI * $shape->radius ** 2;
            }
        }
        return $area;
    }
}
```

The Open-Closed Principle

```php
// Хорошо: Используем полиморфизм
interface Shape {
    public function area();
}

class Square implements Shape {
    public $side;
    public function area() { return $this->side ** 2; }
}

class Circle implements Shape {
    public $radius;
    public function area() { return M_PI * $this->radius ** 2; }
}

class AreaCalculator {
    public function calculate($shapes) {
        $area = 0;
        foreach ($shapes as $shape) {
            $area += $shape->area();
        }
        return $area;
    }
}
```

Liskov Substitution Principle (LSP)

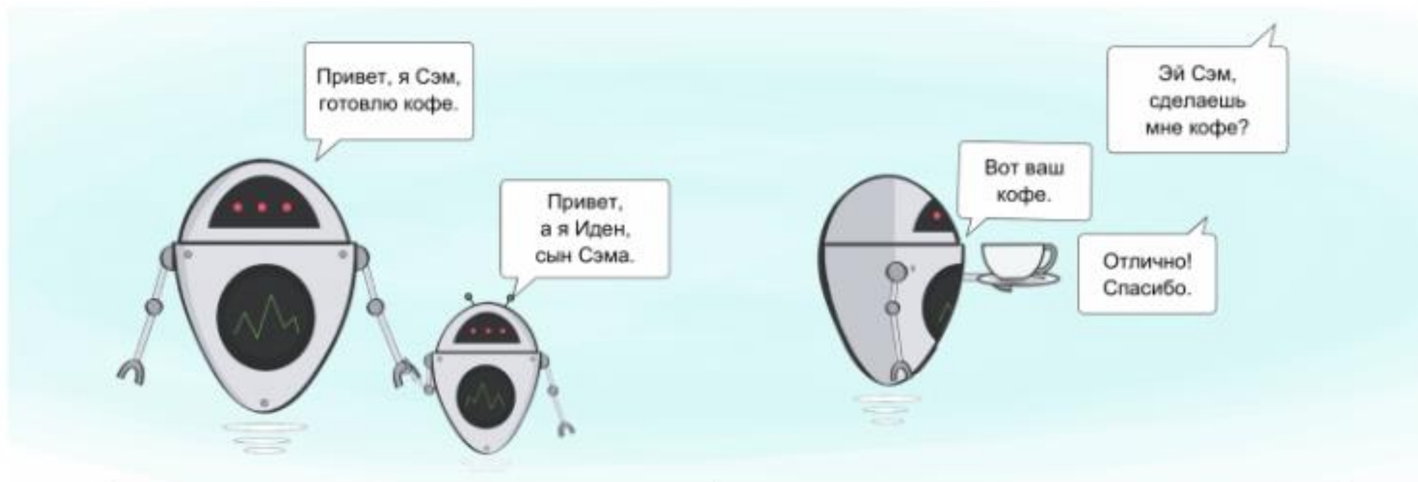
Liskov Substitution Principle

Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом.

Наследующий класс должен дополнять, а не изменять родительский.

Барбара Лисков

Дата рождения	7 ноября 1939 (85 лет)
Место рождения	Лос-Анджелес, Калифорния, США
Страна	США
Род деятельности	специалист в области информатики, преподаватель университета, инженер
Научная сфера	информатика
Место работы	Массачусетский технологический институт
Альма-матер	Стэнфордский университет (1968)[1] Калифорнийский университет в Беркли



Принцип подстановки Барбары Лисков

В случаях, когда класс-потомок не способен выполнять те же действия, что и класс-родитель, возникает риск появления ошибок.

Liskov Substitution Principle

```
// Плохо: Класс Penguin нарушает LSP, так как не может летать
class Bird {
    public function fly() { /* ... */ }
}

class Penguin extends Bird {
    public function fly() {
        throw new Exception("Penguins can't fly!");
    }
}

// Хорошо: Разделяем интерфейсы
interface FlyingBird {
    public function fly();
}

class Sparrow implements FlyingBird {
    public function fly() { /* ... */ }
}

class Penguin {
    // Нет метода fly()
}
```

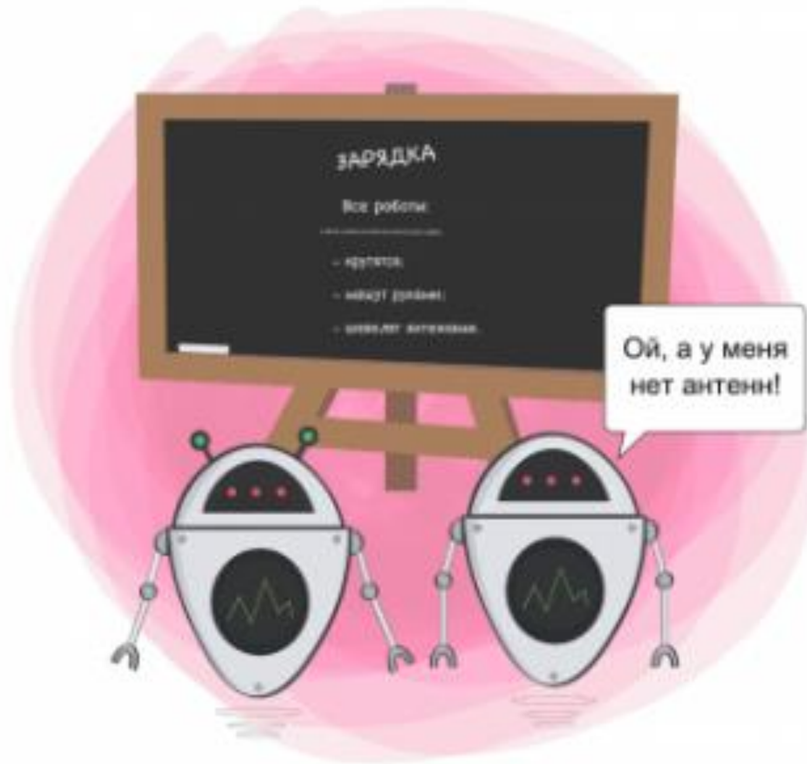
Interface Segregation Principle (ISP)

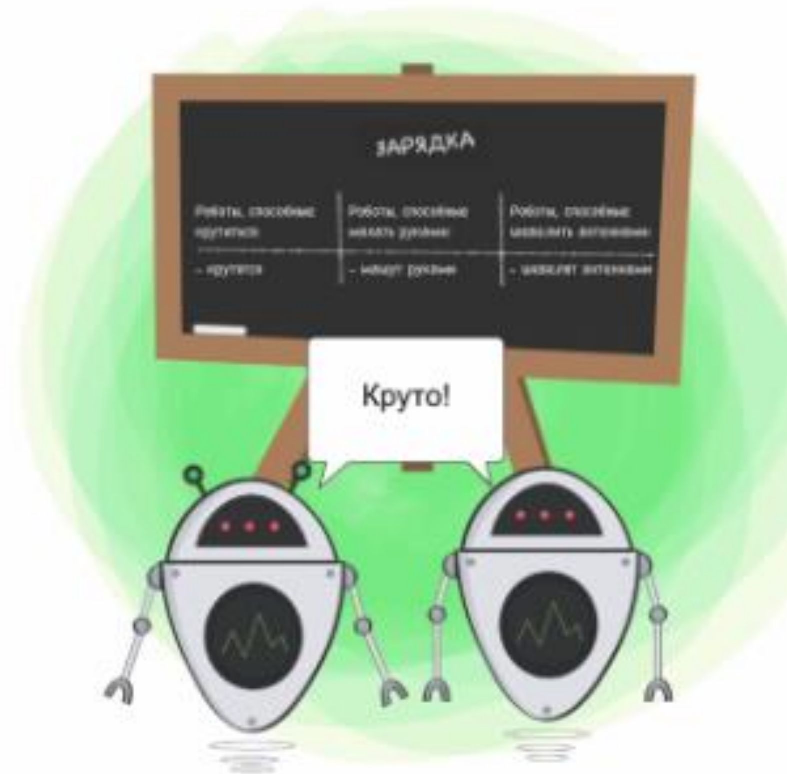
Interface Segregation Principle

Программные сущности не должны зависеть от частей интерфейса, которые они не используют (и знать о них тоже не должны).

Создавайте узкоспециализированные интерфейсы вместо «толстых» универсальных.

Interface Segregation Principle





Принцип разделения интерфейсов

Interface Segregation Principle

```
// Плохо: Интерфейс Worker требует реализации лишних методов
interface Worker {
    public function work();
    public function eat();
}

class HumanWorker implements Worker {
    public function work() { /* ... */ }
    public function eat() { /* ... */ }
}

class RobotWorker implements Worker {
    public function work() { /* ... */ }
    public function eat() {
        throw new Exception("Robots don't eat!");
    }
}
```

Interface Segregation Principle

```
// Хорошо: Разделяем интерфейсы
interface Workable {
    public function work();
}

interface Eatable {
    public function eat();
}

class HumanWorker implements Workable, Eatable {
    // Реализация обоих методов
}

class RobotWorker implements Workable {
    // Реализация только work()
}
```

Dependency Inversion Principle (DIP)

The Dependency Inversion Principle

Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.

Принцип инверсии зависимостей

The Dependency Inversion Principle

1. *Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.*
2. *Абстракции не должны зависеть от деталей реализации. Детали должны зависеть от абстракций.*

Уровни приложения

1. **Представление (UI)** отвечает за взаимодействие с пользователем.
2. **Бизнес-логика (Domain)** содержит правила приложения.
3. **Инфраструктура** обеспечивает работу с БД, API, файловой системой и т.д.

"не должны зависеть" = "не должны знать о деталях"

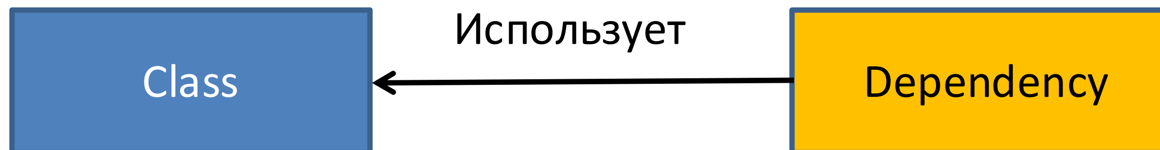
Зависимости между уровнями

Жесткая связь: изменение в инфраструктуре (например, смена БД) ломает бизнес-логику.

- **Бизнес-логика** определяет абстракции (интерфейсы), которые нужны для её работы.
- **Инфраструктура** реализует эти интерфейсы.
- **UI** взаимодействует с бизнес-логикой через абстракции.

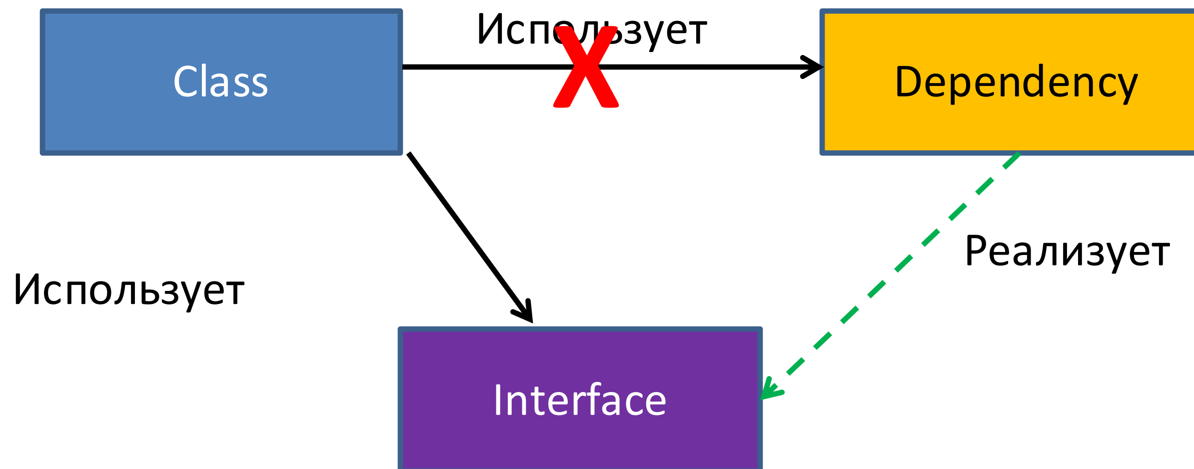
The Dependency Inversion Principle

Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.



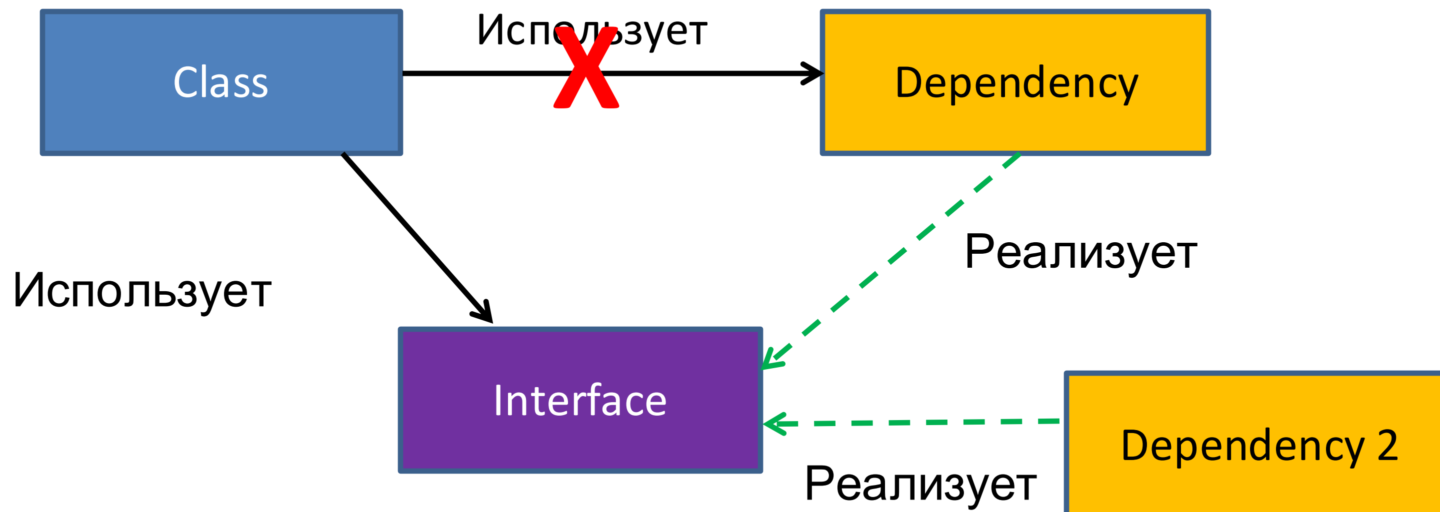
The Dependency Inversion Principle

Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.



The Dependency Inversion Principle

Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.



Без DIP: жесткая зависимость

```
// infrastructure/MySQLRepository.php
class MySQLRepository {
    public function save(array $data): void {
        // Логика сохранения в MySQL
    }
}

// domain/UserService.php
class UserService {
    private MySQLRepository $repository;

    public function __construct() {
        // Жёсткая зависимость от инфраструктурного слоя
        $this->repository = new MySQLRepository();
    }

    public function registerUser(array $userData): void {
        // Бизнес-логика
        $this->repository->save($userData);
    }
}
```

Без DIP: жесткая зависимость

Проблемы :

- Если заменить MySQL на MongoDB, потребуется изменить `UserService`.
- Тестировать `UserService` сложно — нужно поднимать реальную БД.

С DIP: зависимость от абстракций

1. **Domain Layer** определяет интерфейс:

php

```php
// domain/RepositoryInterface.php
interface RepositoryInterface {
    public function save(array $data): void;
}
```

2. **Infrastructure Layer** реализует интерфейс:

php

```php
// infrastructure/MySQLRepository.php
class MySQLRepository implements RepositoryInterface {
    public function save(array $data): void {
        // Логика сохранения в MySQL
    }
}

// infrastructure/MongoDBRepository.php
class MongoDBRepository implements RepositoryInterface {
    public function save(array $data): void {
        // Логика сохранения в MongoDB
    }
}
```

C DIP: зависимость от абстракций

3. **Domain Layer** зависит только от интерфейса:

```php
// domain/UserService.php
class UserService {
    private RepositoryInterface $repository;

    // Зависимость внедряется "снаружи" (Dependency Injection)
    public function __construct(RepositoryInterface $repository) {
        $this->repository = $repository;
    }

    public function registerUser(array $userData): void {
        // Бизнес-логика
        $this->repository->save($userData);
    }
}
```

С DIP: зависимость от абстракций

- **Инверсия зависимостей**:
  - Инфраструктурный слой (`MySQLRepository`) зависит от интерфейса, объявленного в доменном слое.
  - Бизнес-логика (`UserService`) не знает о конкретной реализации хранилища.
- **Внедрение зависимостей (DI)**:

php Копировать

```php
// Входная точка приложения (например, index.php)
$repository = new MySQLRepository(); // или MongoDBRepository
$userService = new UserService($repository);
```

Преимущества DI и DIP

1. **Гибкость** :

```php
// Легко меняем реализацию
$userService = new UserService(new MongoDBRepository());
```

2. **Тестирование** :

```php
// Тест с использованием мока (например, PHPUnit)
$mockRepository = $this->createMock(RepositoryInterface::class);
$mockRepository->expects($this->once())
               ->method('save')
               ->with(['name' => 'John']);

$userService = new UserService($mockRepository);
$userService->registerUser(['name' => 'John']);
```

3. **Слоистая архитектура** :

- **Domain Layer** содержит бизнес-правила и абстракции.
- **Infrastructure Layer** реализует детали (БД, API).
- **Presentation Layer** работает с доменом через интерфейсы.