Объектно-ориентированное программирование

4 семестр
Лекция 1.
Содержание лекции:

- Отношения между классами и объектами
- Принципы разработки ПО
- Примеры соблюдения и нарушения принципов
- Принципы SOLID
- Принципы GRASP

Преподаватель курса: Клюкин Даниил Анатольевич, ст. преподаватель каф. ПМиИТ

Отношения между классами и объектами

Наследование является базовым принципом ООП и позволяет одному классу (наследнику) унаследовать функционал другого класса (родительского). Нередко отношения наследования еще называют генерализацией или обобщением. Наследование определяет отношение IS A, то есть "является". Например:

```
class User
{
    public int Id { get; set; }
    public string Name { get; set; }
}
class Manager : User
{
    public string Company { get; set; }
}
```

Реализация предполагает определение интерфейса и его реализация в классах. Например, имеется интерфейс IMovable с методом Move, который реализуется в классе Car:

```
public interface IMovable
{
    void Move();
}
public class Car : IMovable
{
    public void Move()
    {
        Console.WriteLine("Машина едет");
    }
}
```

Ассоциация - это отношение, при котором объекты одного типа неким образом связаны с объектами другого типа. Например, объект одного типа содержит или использует объект другого типа. Например, игрок играет в определенной команде:

```
class Team
{}
class Player
{
    public Team Team { get; set; }
}
```

Класс Player связан отношением ассоциации с класом Team. Нередко при отношении ассоциации указывается кратность связей. В данном случае одна команда будет соответствовать многим игрокам.

Агрегация и композиция являются частными случаями ассоциации.

Композиция определяет отношение HAS A, то есть отношение "имеет". Например, в класс автомобиля содержит объект класса электрического двигателя:

```
public class ElectricEngine { }
public class Car
{
    ElectricEngine engine;
    public Car()
    {
        engine = new ElectricEngine();
    }
}
```

Класс автомобиля управляет жизненным циклом двигателя. При уничтожении автомобиля вместе с ним будет уничтожен и объект двигателя. Объект автомобиля – главный, а объект двигателя - зависимый.

Агрегация. Она также предполагает отношение HAS A, но реализуется она иначе

```
public abstract class Engine
{ }
public class Car
{
    Engine engine;
     public Car(Engine eng)
     {
         engine = eng;
     }
}
```

При агрегации реализуется слабая связь, то есть в данном случае объекты Car и Engine будут равноправны. В конструктор Car передается ссылка на уже имеющийся объект Engine. И, как правило, определяется ссылка не на конкретный класс, а на абстрактный класс или интерфейс, что увеличивает гибкость программы.

Один из принципов проектирования гласит, что при создании системы классов надо программировать на уровне интерфейсов, а не их конкретных реализаций. Под интерфейсами в данном случае понимаются не только типы C#, определенные с помощью ключевого слова **interface**, а **определение функционала без его конкретной реализации**. То есть под данное определение попадают как собственно интерфейсы, так и абстрактные классы, которые могут иметь абстрактные методы без конкретной реализации.

Когда следует использовать **абстрактные классы**:

- Если надо определить общий функционал для родственных объектов
- Если мы проектируем довольно большую функциональную единицу, которая содержит много базового функционала
- Если нужно, чтобы все производные классы на всех уровнях наследования имели некоторую общую реализацию. При использовании абстрактных классов, если мы захотим изменить базовый функционал во всех наследниках, то достаточно поменять его в абстрактном базовом классе.
- Если же нам вдруг надо будет поменять название или параметры метода интерфейса, то придется вносить изменения и также во всех классы, которые данный интерфейс реализуют.

Когда следует использовать интерфейсы:

- Если нам надо определить функционал для группы разрозненных объектов, которые могут быть никак не связаны между собой.
- Если мы проектируем небольшой функциональный тип

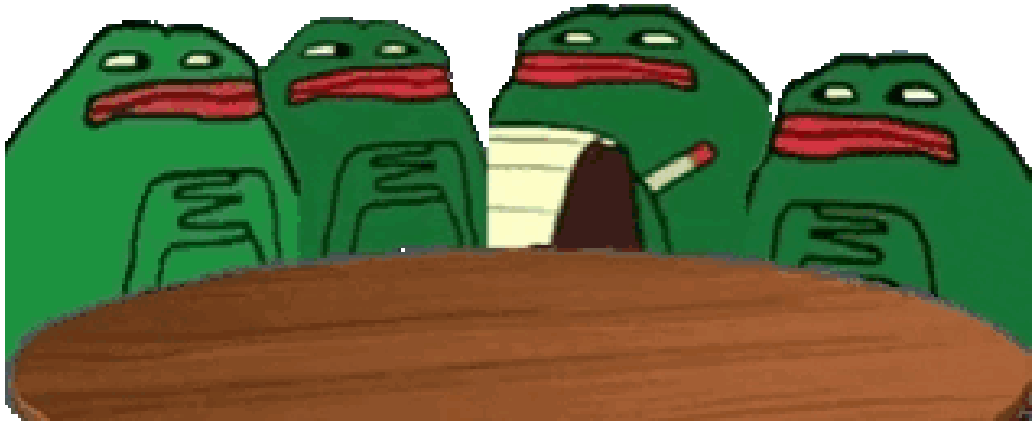
```
var tr = new Tram();

public abstract class Vehicle
{
    public abstract void Move();
}
public class Car : Vehicle
{
    public override void Move() =>Console.WriteLine("Машина едет");
}
public class Bus : Vehicle
{
    public override void Move() =>Console.WriteLine("Автобус едет");
}
public class Tram : Vehicle
{
    public override void Move() =>Console.WriteLine("Трамвай едет");
}
```

```
public interface IMovable{
    void Move();
}
public abstract class Vehicle : IMovable{
    public abstract void Move();
}
public class Car : Vehicle{
    public override void Move() =>Console.WriteLine("Машина едет");
}
public class Bus : Vehicle{
    public override void Move() =>Console.WriteLine("Автобус едет");
}
public class Hourse : IMovable{
    public void Move() => Console.WriteLine("Лошадь скачет");
}
public class Aircraft : IMovable{
    public void Move() => Console.WriteLine("Самолет летит");
}
```

Принципы разработки ПО

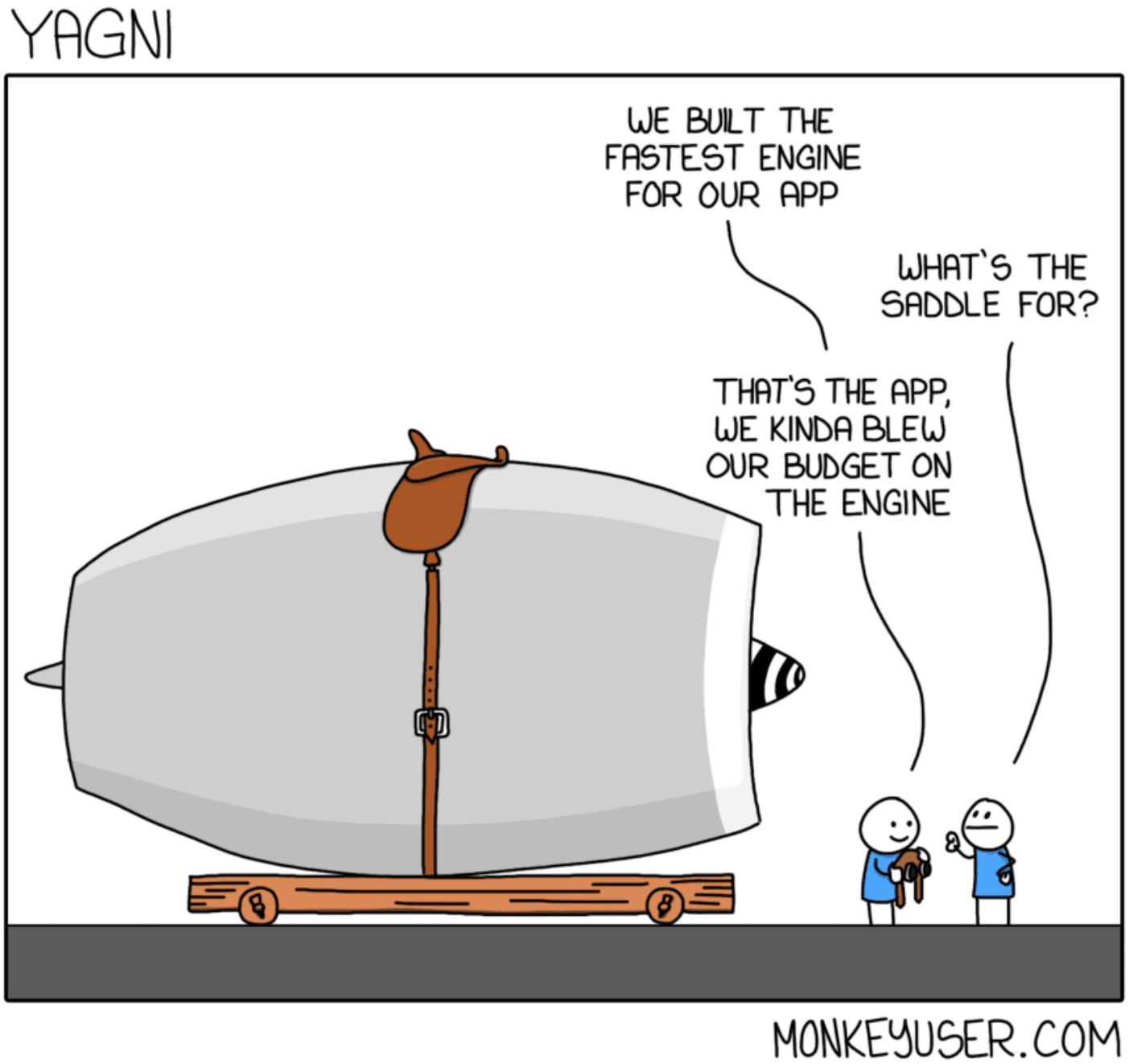
Принципы разработки программного обеспечения необходимо знать каждому инженеру, который хочет писать чистый код. Следование этим принципам позволяет вам и другим разработчикам понять проект.

Кроме того, обслуживание или изменение проекта в будущем станет легким. Таким образом, вы в конечном итоге сэкономите деньги, время и ресурсы. Если вы хотите, чтобы проект развивался более плавно, то рекомендуется жить по этим законам.



Принцип	Пояснение, источник
SOLID	single responsibility – единственной ответственности, open-closed – открытости / закрытости, Liskov substitution – подстановка Лисков, interface segregation – разделения интерфейсов, dependency inversion – инверсии зависимостей. *Источник: Р. Мартин «Чистая Архитектура», Часть III Принципы дизайна*
GRASP	Группа из 9 принципов: Information Expert, Creator, Controller, Low Coupling, High Cohesion, Polymorphism, Pure Fabrication, Indirection, Protected Variations. *Источник: К. Ларман «Применение UML 2.0 и шаблонов проектирования», Главы 17.10-17.14, 25*
KISS	Keep It Simple, Stupid / Делай проще *Источник*
DRY	Don't Repeat Yourself / Не повторяйся *Источник*

Принцип	Пояснение, источник
YAGNI	You Aren't Gonna Need It / Вам это не понадобится Вы не должны создавать функции, в которых на самом деле нет необходимости. *Источник*
BDUF	Big Design Up Front / Глобальное проектирование прежде всего *Источник*
APO	Avoid Premature Optimization / Избегайте преждевременной оптимизации. *Источник: The Art of Computer Programming, Knut*
LoD	Law of Demeter / Закон Деметры Объект A не должен иметь возможность получить непосредственный доступ к объекту C, если у объекта A есть доступ к объекту B и у объекта B есть доступ к объекту C. *Источник: Lieberherr, Karl J. and Ian M. Holland. "Assuring good style for object-oriented programs." IEEE Software 6 (1989): 38-48.*



- Мы сделали самый быстрый движок для нашего приложения
- А зачем нужно седло?
- Это наше приложение, просто мы потратили все средства на движок.

Принцип	Пояснение, источник
LKP	Principle of Least Knowledge / Принцип наименьшего знания, аналог LoD Объект должен иметь как можно меньше представления о структуре и свойствах чего угодно (включая собственные подкомпоненты) *Источник*
Okkama Razor	Бритва Оккама: «Не следует множить сущности без необходимости» *Источник*
Principle of Least Astonishment	Правило наименьшего удивления, principle of least surpris. *Источник*
Принцип Эйнштейна	Сделай настолько просто, насколько возможно, но не проще *Источник*

Принцип	Пояснение, источник
Римский принцип	Разделяй и властвуй – разбиение задачи на подзадачи *Источник: The Yale Book of Quotations, 2006, p. 610.*
Принцип Калашникова	Избыточная сложность – это уязвимость *Источник*
Tell-Don't-Ask	Говори, не спрашивай. Вместо того, чтобы спрашивать данные у объекта, мы должны сказать объекту что с ними делать. *Источник*
CQS	Command-query Separation: каждая функция является ЛИБО командой (command), которая выполняет действие, ЛИБО запросом (query) для получения данных *Источник*
Measure Twice and Cut Once	Семь раз отмерь, один раз отрежь *Источник*

Принцип/ группа	Пояснение, источник
Unix-философия	Группа принципов. Состоит из 17 правил, описанных Эриком Рэймондом в книге «Искусство программирования в Unix». Правило модульности: Пишите простые части, соединенные понятными интерфейсами. Правило ясности: Ясность лучше, чем продуманность. Правило композиции: Разрабатывайте программы так, чтобы они были связаны с другими программами. Правило простоты: Разрабатывайте для простоты; усложняйте только там, где это необходимо. Правило расширяемости: Создавайте дизайн для будущего, потому что оно наступит раньше, чем вы думаете. И т.д. *Источник*

Принципы SOLID

Термин "**SOLID**" представляет собой **акроним** для набора практик проектирования программного кода и построения гибкой и адаптивной программы. Данный термин был введен известным американским специалистом в области программирования Робертом Мартином (Robert Martin), более известным как "дядюшка Боб" или Uncle Bob.

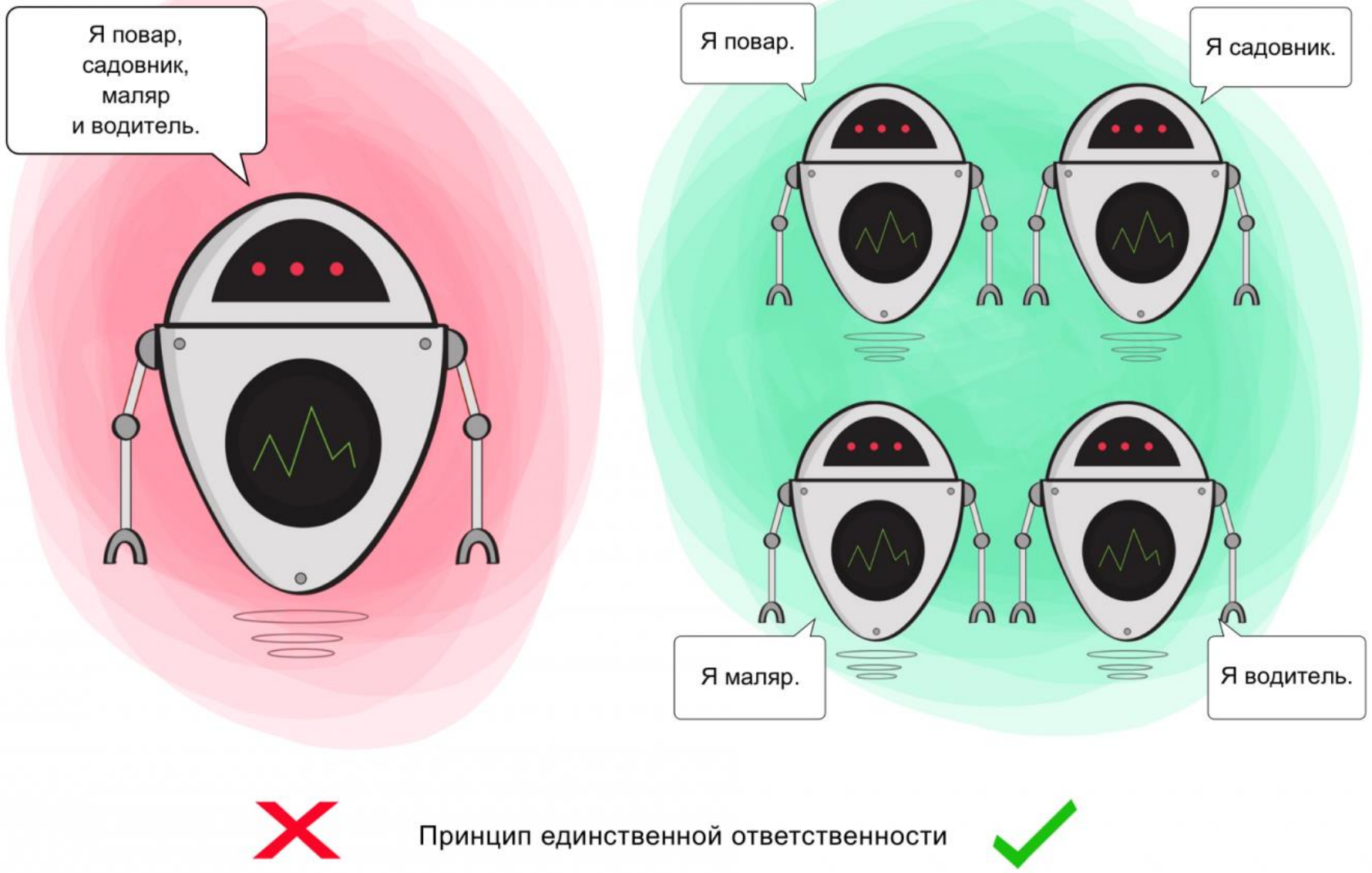
Сам акроним образован по первым буквам названий SOLID-принципов:

- Single Responsibility (Принцип единственной обязанности)
- Open/Closed (Принцип открытости/закрытости)
- Liskov Substitution (Принцип подстановки Лисков)
- Interface Segregation (Принцип разделения интерфейсов)
- Dependency Inversion (Принцип инверсии зависимостей)

Принципы SOLID - это не паттерны, их нельзя назвать какими-то определенными догмами, которые надо обязательно применять при разработке, однако их использование позволит улучшить код программы, упростить возможные его изменения и поддержку.

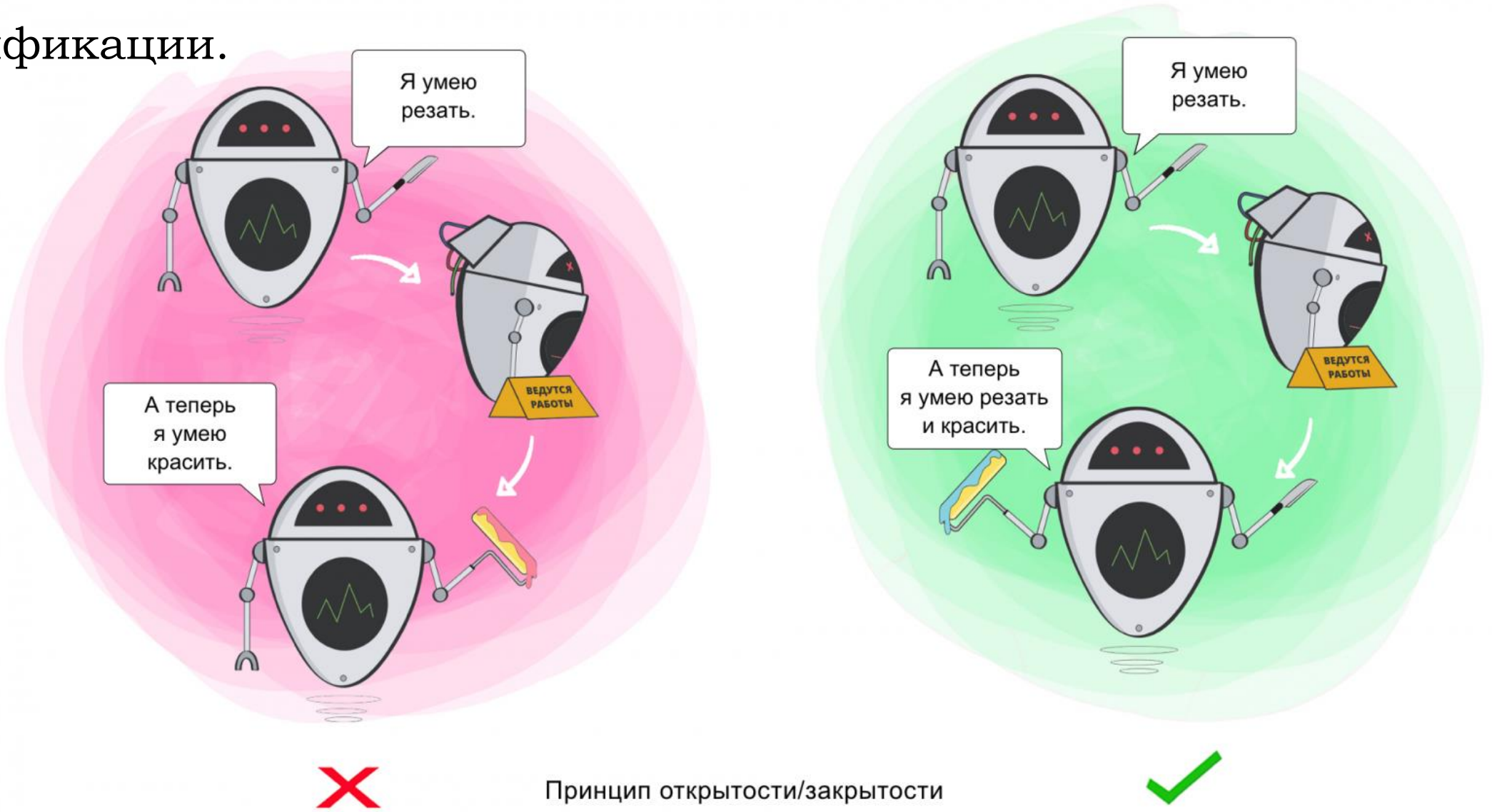
Single Responsibility (Принцип единственной обязанности)

Каждый класс должен отвечать только за одну операцию.



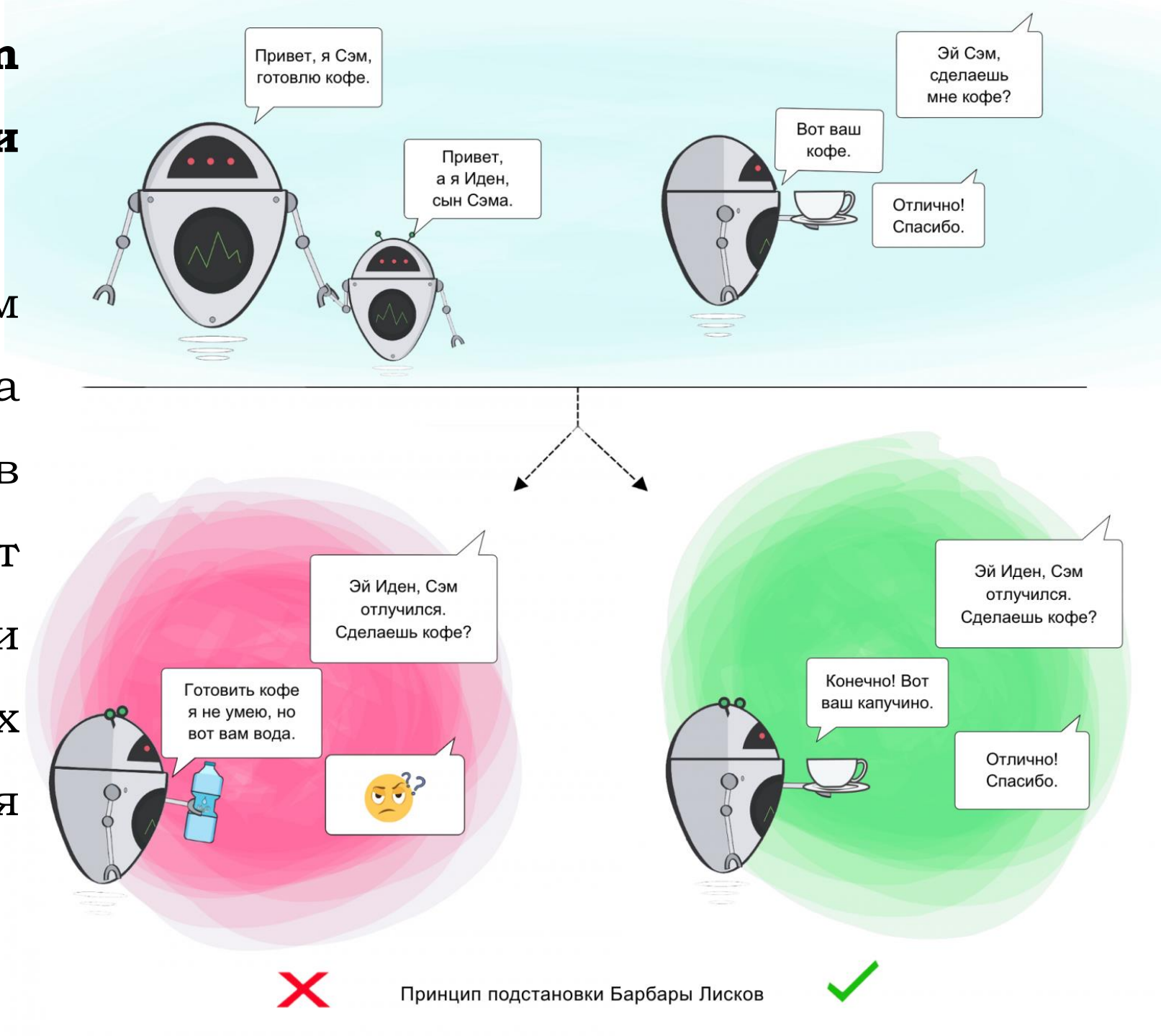
Open/Closed (Принцип открытости/закрытости)

Классы должны быть открыты для расширения, но закрыты для модификации.



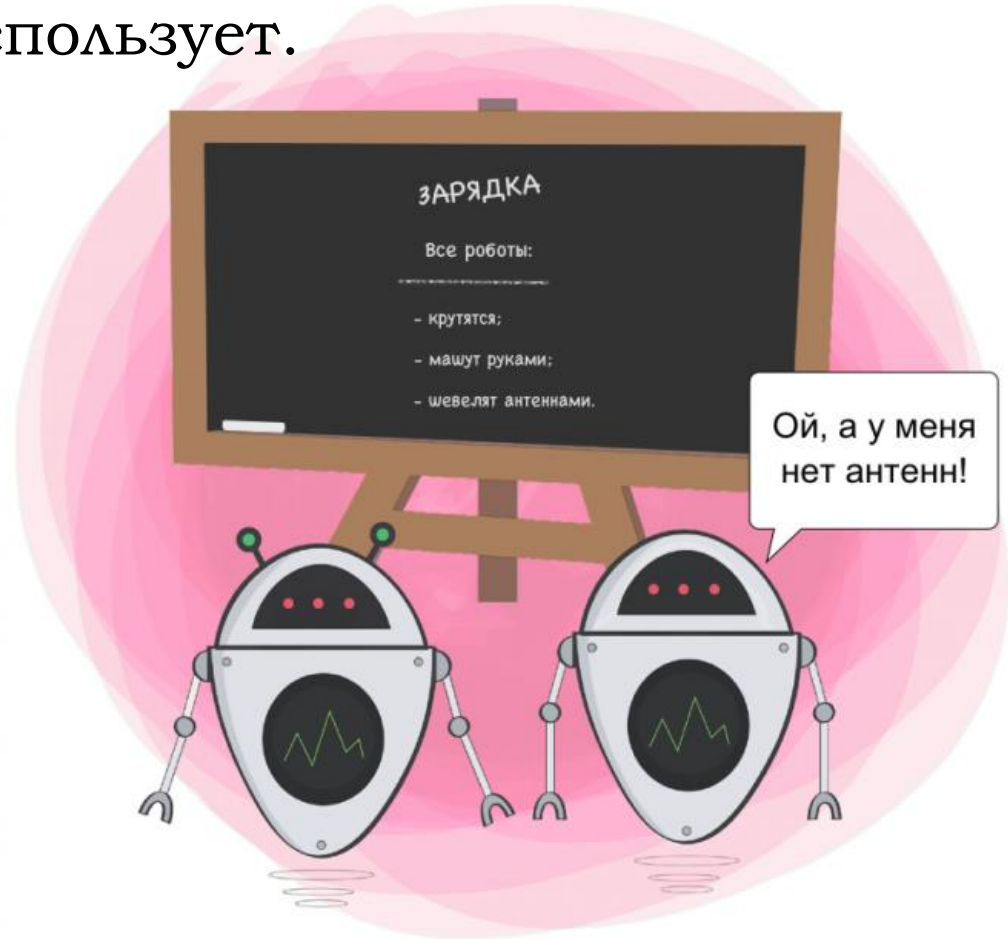
Liskov Substitution (Принцип подстановки Лисков)

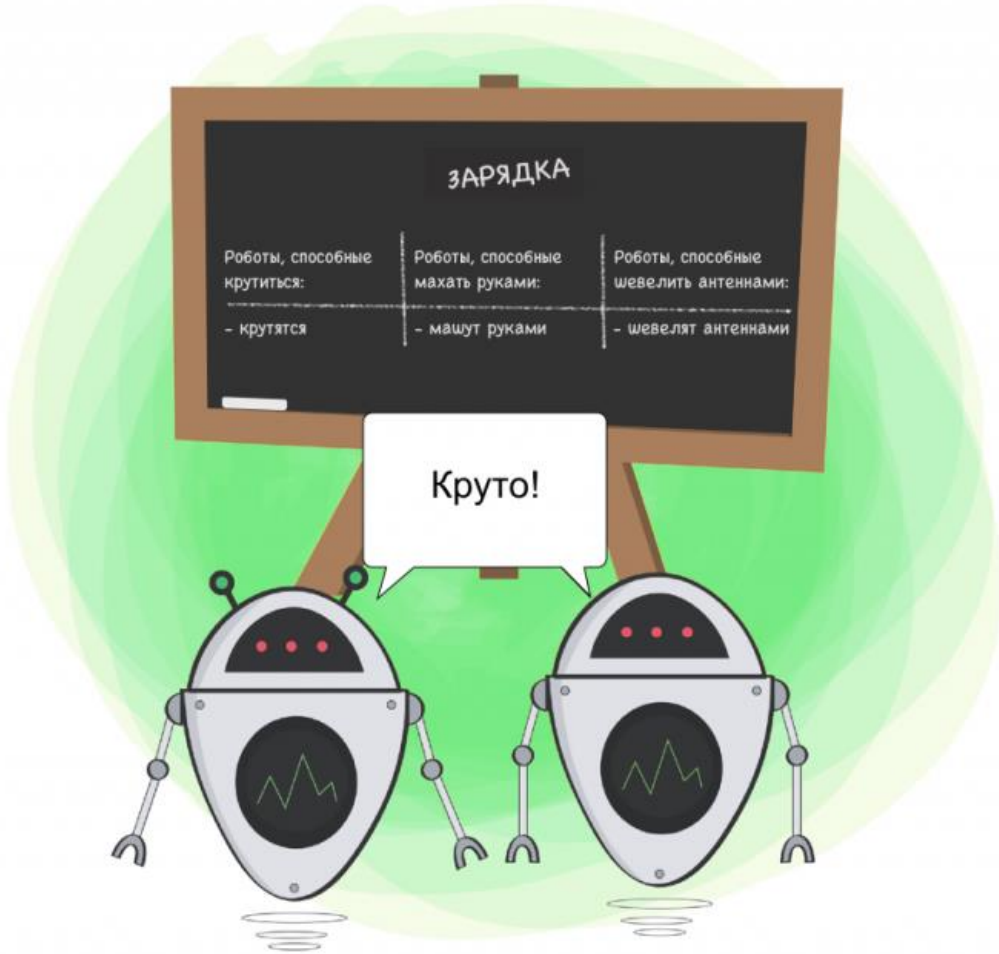
Если **П** является подтипом **Т**, то любые объекты типа **Т**, присутствующие в программе, могут заменяться объектами типа **П** без негативных последствий для функциональности программы.



Interface Segregation (Принцип разделения интерфейсов)

Не следует ставить клиент в зависимость от методов, которые он не использует.

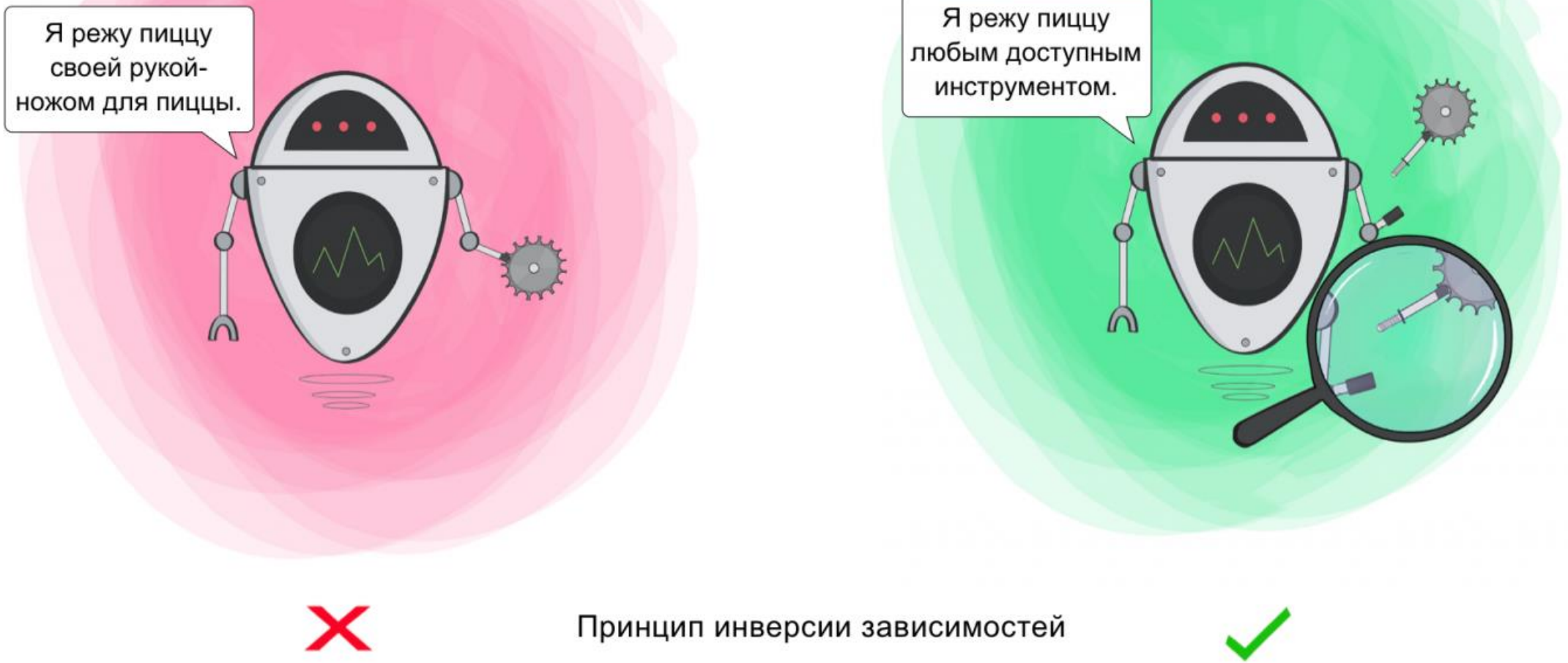




Принцип разделения интерфейсов

Dependency Inversion (Принцип инверсии зависимостей)

Модули верхнего уровня не должны зависеть от модулей нижнего уровня. И те, и другие должны зависеть от абстракций. Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.



Принципы GRASP

GRASP (от англ. General Responsibility Assignment Software Patterns — шаблоны ПО для назначения главных ответственностей; также отсылает к англ. grasp — «способность быстрого восприятия, понимание, схватывание») — шаблоны, используемые в объектно-ориентированном проектировании для решения общих задач по назначению ответственностей классам и объектам.

1. Информационный эксперт (Information Expert)

Обязанности должны быть назначены объекту, который владеет максимумом необходимой информации для выполнения обязанности. Этот шаблон — самый очевидный и важный из девяти. Если его не учесть — получится **спагетти-код**, в котором трудно разобраться.

Локализация же ответственностей, проводимая согласно шаблону:

•Повышает:

- **Инкапсуляцию**;
- Простоту восприятия;
- Готовность компонентов к повторному использованию;

•Снижает:

- степень **зацепления**.

2. Создатель (Creator)

Шаблон «*Creator*» — это интерпретация шаблона «*Information Expert*» (смотрите пункт № 1) в контексте создания объектов.

Проблема: Кто отвечает за создание объекта некоторого класса A?

Решение: Назначить классу B обязанность создавать объекты класса A, если класс B:

•содержит или агрегирует объекты A;

•записывает объекты A;

•активно использует объекты A;

•обладает данными для инициализации объектов A

Большинство порождающих шаблонов проектирования так или иначе выводятся или опираются на шаблон «*Creator*».

3. Контроллер (Controller)

Отвечает за операции, запросы которых приходят от пользователя, и может выполнять сценарии одного или нескольких вариантов использования (например, создание и удаление);

Не выполняет работу самостоятельно, а делегирует компетентным исполнителям;

Может представлять собой:

- Систему в целом;
- Подсистему;
- Корневой объект;
- Устройство.

4. Слабое (низкое) зацепление (Low Coupling)

Основная статья: ***Зацепление (программирование)***

Зацепление — мера того, насколько взаимозависимы разные подпрограммы или модули.

Сильное зацепление рассматривается как серьёзный **недостаток**, поскольку затрудняет понимание логики модулей, их модификацию, автономное тестирование, а также переиспользование по отдельности. **Слабое** зацепление, напротив, является признаком хорошо структурированной и хорошо спроектированной системы.

5. Сильная (высокая) связность (High Cohesion)

Основная статья: ***Связность (программирование)***

Связность — мера силы взаимосвязанности элементов внутри модуля; способ и степень, в которой задачи, выполняемые некоторым программным модулем, связаны друг с другом.

Сильная связность класса / модуля означает, что его элементы тесно связаны и сфокусированы.

Слабая (низкая) связность класса / модуля означает, что он не сфокусирован на одной цели, его элементы предназначены для слишком многих несвязанных обязанностей. Такой модуль трудно понять, использовать и поддерживать.

6. Полиморфизм (Polymorphism)

См. также: Полиморфизм (информатика)

Устройство и поведение системы:

Определяется данными;

Задано полиморфными операциями её интерфейса.

Пример: Адаптация коммерческой системы к *многообразию* систем учёта налогов может быть обеспечена через внешний интерфейс объектов-адаптеров (см. также: Шаблон «Адаптеры»).

7. Чистая выдумка (Pure Fabrication)

Не относится к предметной области, но:

Уменьшает зацепление;

Повышает связность;

Упрощает повторное использование.

«*Pure Fabrication*» отражает концепцию сервисов в модели предметно-ориентированного проектирования.

Пример задачи: Не используя средства класса «А», внести его объекты в базу данных.

Решение: Создать класс «Б» для записи объектов класса «А» (см. также: «***Data Access Object***»).

8. Перенаправление (Indirection)

См. также: Посредник (шаблон проектирования)

Слабое зацепление между элементами системы (и возможность повторного использования) обеспечивается назначением промежуточного объекта их **посредником**.

Пример: В архитектуре Model-View-Controller, *контроллер* ослабляет зацепление данных с их представлением *(англ. view)*.

9. Устойчивость к изменениям (Protected Variations)

Шаблон защищает элементы от изменения другими элементами (объектами или подсистемами) с помощью вынесения взаимодействия в фиксированный интерфейс, через который (и только через который) возможно взаимодействие между элементами. Поведение может варьироваться лишь через создание другой реализации интерфейса.

Картинки взяты с сайта:

https://habr.com/ru/companies/productivity_inside/articles/505430/