ООП

Семестр 2

SOLID

Кафедра ИВТ и ПМ
ЗабГУ

2019

План

- Прошлые темы
  - MVC
- SOLID
  - Принцип единственной ответственности
  - Принцип открытости/закрытости
  - Принцип подстановки Барбары Лисков
  - Принцип разделения интерфейса
  - Принцип инверсии зависимостей

Outline

Outline

Model-View-Controller (MVC)

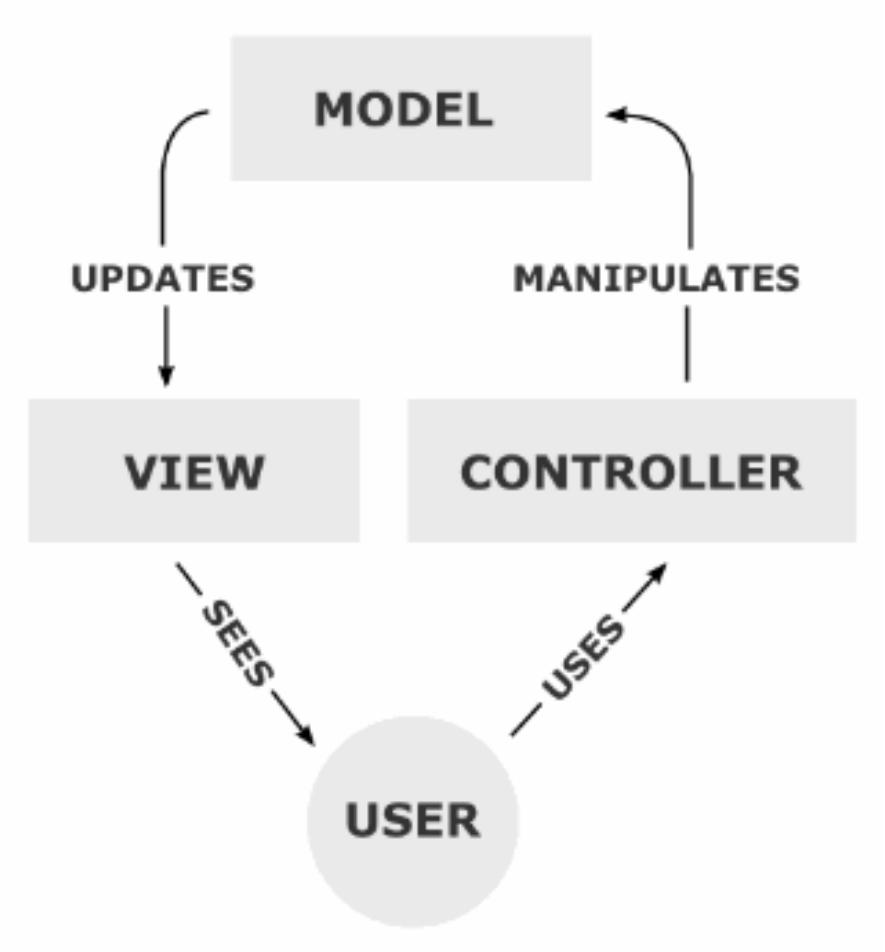
Что представляет собой шаблон проектирования Model-View-Controller?

Model-View-Controller (MVC)

Что представляет собой шаблон проектирования Model-View-Controller?

Model-View-Controller (MVC, Модель-Представление-Контроллер, Модель-Вид-Контроллер) — схема разделения данных приложения, пользовательского интерфейса и управляющей логики на три отдельных компонента: модель, представление и контроллер — таким образом, что модификация каждого компонента может осуществляться независимо.

Model-View-Controller (MVC)



Model-View-Controller (MVC)

Модель (Model) предоставляет данные и реагирует на команды контроллера, изменяя свое состояние.

Представление (View) отвечает за отображение данных модели пользователю, реагируя на изменения модели.

Контроллер (Controller) интерпретирует действия пользователя, оповещая модель о необходимости изменений.

Outline

SOLID

SOLID - принципы объектно-ориентированного проектирования - single responsibility, open-closed, Liskov substitution, interface segregation и dependency inversion)

SOLID

- **S.** Принцип единственной ответственности (The Single Responsibility Principle, SRP)
- **O.** Принцип открытости/закрытости (The Open Closed Principle, OCP)
- **L.** Принцип подстановки Барбары Лисков (The Liskov Substitution Principle, LSP)
- **I.** Принцип разделения интерфейса (The Interface Segregation Principle, ISP)
- **D.** Принцип инверсии зависимостей (The Dependency Inversion Principle, DIP)

Outline

Принцип единственной ответственности

- **S.** Принцип единственной ответственности (The Single Responsibility Principle, SRP)

  Каждый объект должен иметь одну ответственность и эта ответственность должна быть полностью инкапсулирована в класс. Все его поведения должны быть направлены исключительно на обеспечение этой ответственности.

SOLID. Принцип единственной ответственности

Применяется практически для любого масштаба: метод, класс, модуль.

Например, согласно этому принципу не стоит помещать бизнес-логику в класс окна приложения.

SOLID. Принцип единственной ответственности

- Несоблюдение принципа приводи к созданию божественных объектов.

  Объект-бог (God object) — антипаттерн объектно-ориентированного программирования, описывающий объект, который хранит в себе «слишком много» или делает «слишком много».

SOLID. Принцип единственной ответственности

Если не соблюдать принцип единственной ответственности...

SOLID. Принцип единственной ответственности

- Буквальное и неразумное следование приводит - к увеличению числа классов и усложнению приложения.

Пример[1]

Проблема

```
class Person {
public name : string;
public surname : string;
public email : string;
constructor(name : string, surname : string, email : string){
    this.surname = surname;
    this.name = name;
    if(this.validateEmail(email)) {
      this.email = email;
    }  else {
        throw new Error("Invalid email!"); }
}

validateEmail(email : string) {
    var re = /^([\w-]+(?:\.[\w-]+)*)@((?:[\w-]+\.)*\w[\w-]{0,66})\.
    return re.test(email);
}
greet() {
    alert("Hi!");
}
}
```

Пример[2]

Проблема

Класс Person отвечает ещё и за проверку корректности адреса электронной почты. То есть выполняет несвойственную для себя задачу.
Для электронной почты стоило бы создать отдельный класс.

[2] пример на TypeScript

Пример

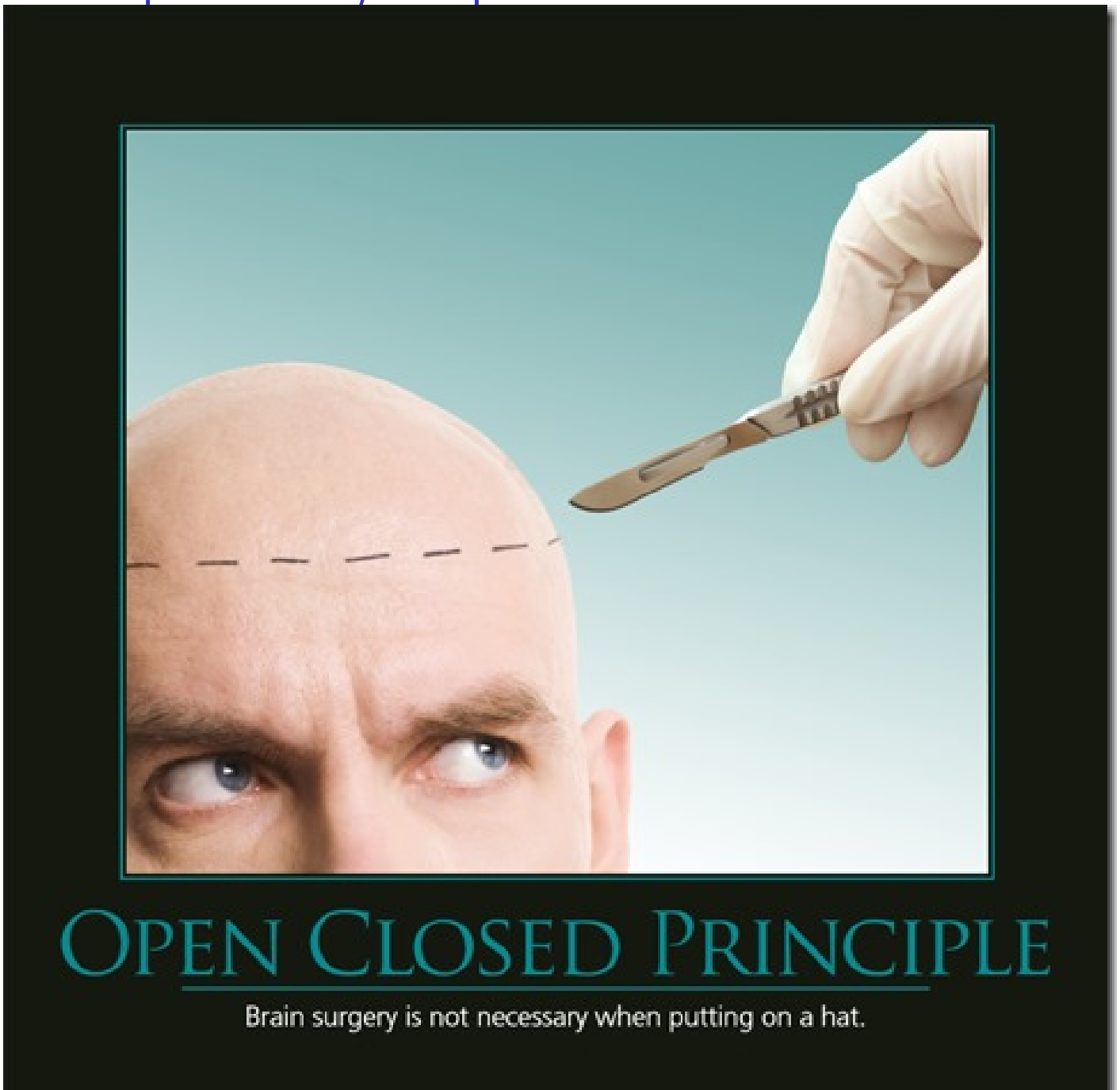
Решение

```
class Email {
public email : string;
constructor(email : string){
    if(this.validateEmail(email)) { //...
    }
    else { throw new Error("Invalid email!"); }
}
validateEmail(email : string) {
    var re = /^([\w-]+(?:\.[\w-]+)*)@((?:[\w-]+\.)*\w[\w-]{0,66})\.
    return re.test(email);
}}

class Person {
    public name : string;
    public surname : string;
    public email : Email;
    // ...
    greet() {
        alert("Hi!");
    }}
```

Outline

Принцип открытости/закрытости



Принцип открытости/закрытости

Как можно разработать проект, устойчивый к изменениям, срок жизни которых превышает срок существования первой версии проекта?

Принцип открытости/закрытости

Как можно разработать проект, устойчивый к изменениям, срок жизни которых превышает срок существования первой версии проекта?

Программные сущности (классы, модули, функции и т. п.) должны быть открыты для расширения, но закрыты для изменения.

Принцип открытости/закрытости

- открыты для расширения: поведение сущности может быть расширено путём создания новых типов сущностей.
- закрыты для изменения: в результате расширения поведения сущности, не должны вноситься изменения в код, который эти сущности использует.

Принцип открытости/закрытости

С помощью какого механизма в ООП можно добиться соблюдения принципа?

Принцип открытости/закрытости

С помощью какого механизма в ООП можно добиться соблюдения принципа?

Наследование.

Принцип открытости/закрытости

С помощью какого механизма в ООП можно добиться соблюдения принципа?

Наследование.

Наследование и полиморфизм

Спецификации интерфейсов могут быть переиспользованы через наследование, но реализации изменяться не должны. Существующий интерфейс должен быть закрыт для модификаций, а новые реализации должны, по меньшей мере, реализовывать этот интерфейс.

Принцип открытости/закрытости

Примеры?

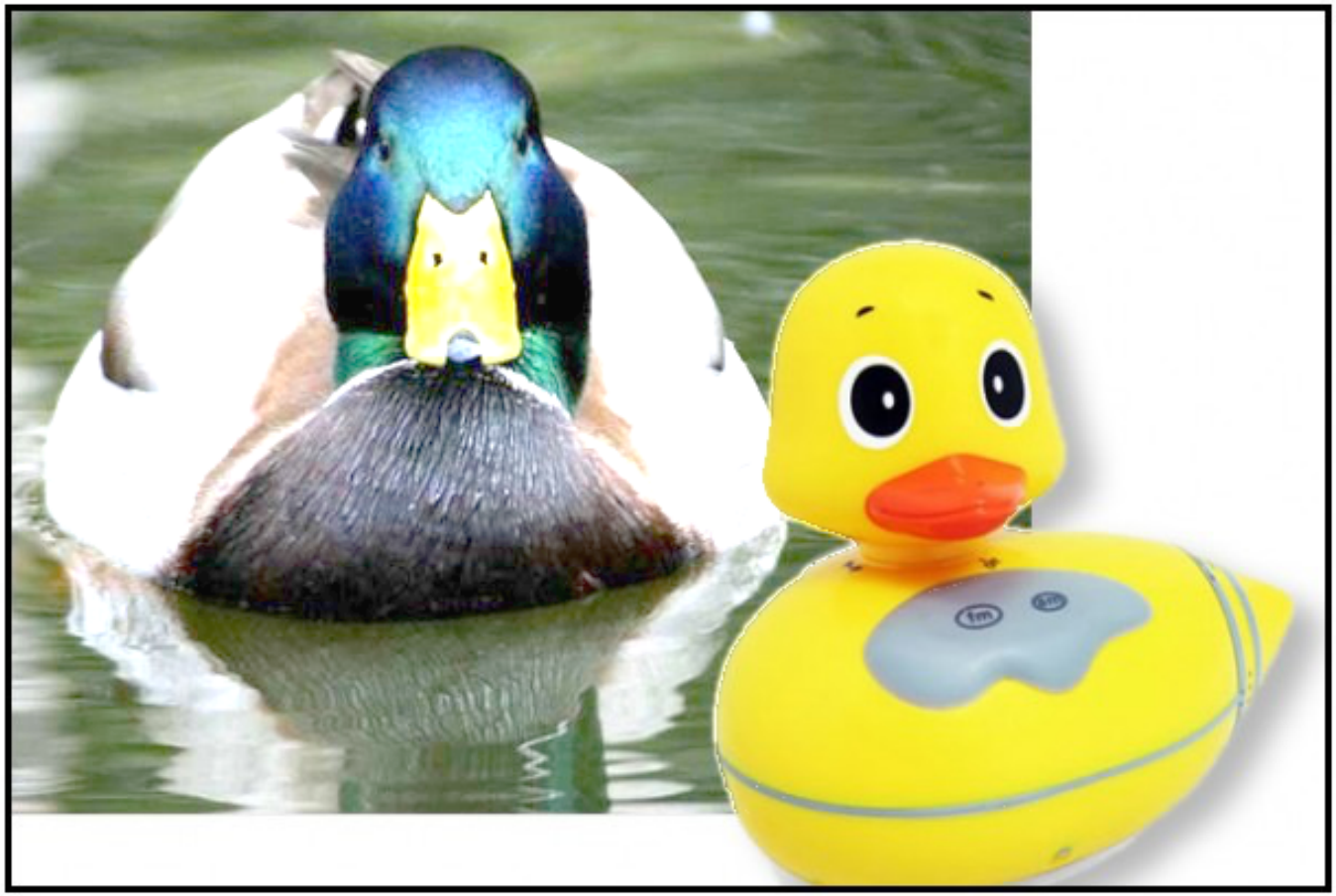
Принцип открытости/закрытости

Примеры?

Создание классов в GUI фреймворках на основе существующих.

Outline

Принцип подстановки Барбары Лисков



LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Принцип подстановки Барбары Лисков

Пусть $q(x)$ является свойством, верным относительно объектов x некоторого типа T.
Тогда $q(y)$ также должно быть верным для объектов y типа S, где S является подтипом типа T.

Принцип подстановки Барбары Лисков

Другими словами...

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

Принцип подстановки Барбары Лисков

```
class Vehicle {
  function startEngine() { /*...*/ }

  function accelerate() { /*...*/ }
}

class Driver {
    function go(Vehicle $v) {
        $v->startEngine();
        $v->accelerate();
    }
}
```

Принцип подстановки Барбары Лисков

Если класс Driver использует Vehicle, то и производные от Vehicle классы должны подходить для Driver.

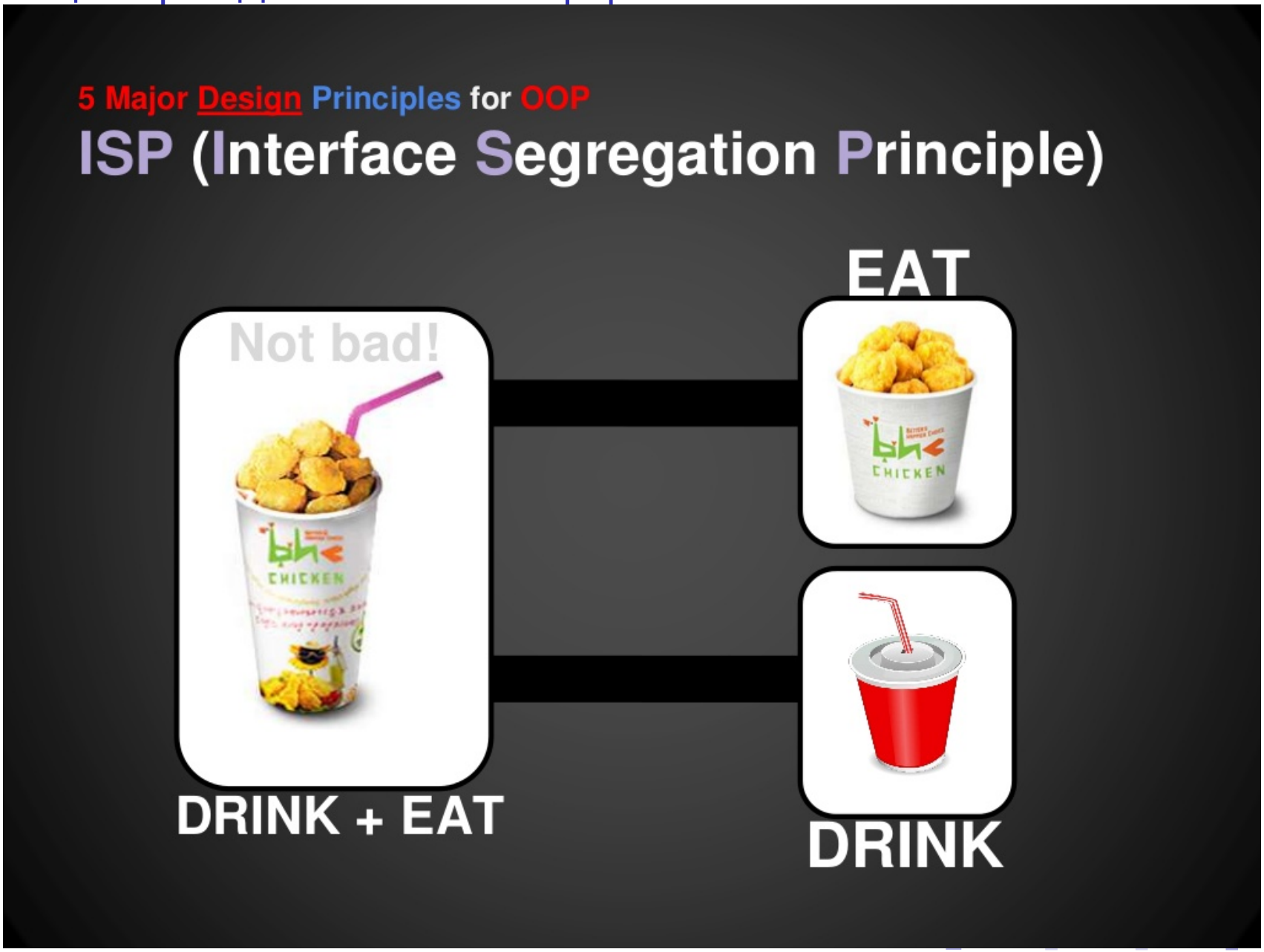
```
class Car extends Vehicle {
    function startEngine() {
        $this->engageIgnition();
        parent::startEngine();}

    private function engageIgnition() {
        // Ignition procedure
    }   }

class ElectricBus extends Vehicle {
    function accelerate() {
        $this->increaseVoltage();
        $this->connectIndividualEngines();}
    private function increaseVoltage() {
        // Electric logic
}
    private function connectIndividualEngines() {
        // Connection logic
    } }
```

Outline

Принцип разделения интерфейса



Принцип разделения интерфейса

Клиенты не должны зависеть от методов, которые они не используют.

Принцип разделения интерфейса

Слишком «толстые» интерфейсы необходимо разделять на более маленькие и специфические, чтобы клиенты маленьких интерфейсов знали только о методах, которые необходимы им в работе.
В итоге, при изменении метода интерфейса не должны меняться клиенты, которые этот метод не используют.

Принцип разделения интерфейса

Плохой пример

```
interface ISmartDevice
{
    void Print();
    void Fax();
    void Scan();
}

class AllInOnePrinter : ISmartDevice
{
    public void Print(){
        // Printing code.
    }

    public void Fax(){
        // Beep booop biiiiiip.
    }

    public void Scan(){
        // Scanning code.
    }
}
```

Принцип разделения интерфейса

Плохой пример (продолжение)

```
class EconomicPrinter : ISmartDevice
{
    public void Print() {
        //Yes I can print.
}

    public void Fax(){
        throw new NotSupportedException();
    }

    public void Scan(){
        throw new NotSupportedException();
    }
}
```

Производный класс EconomicPrinter будет содержать несвойственные для него методы.

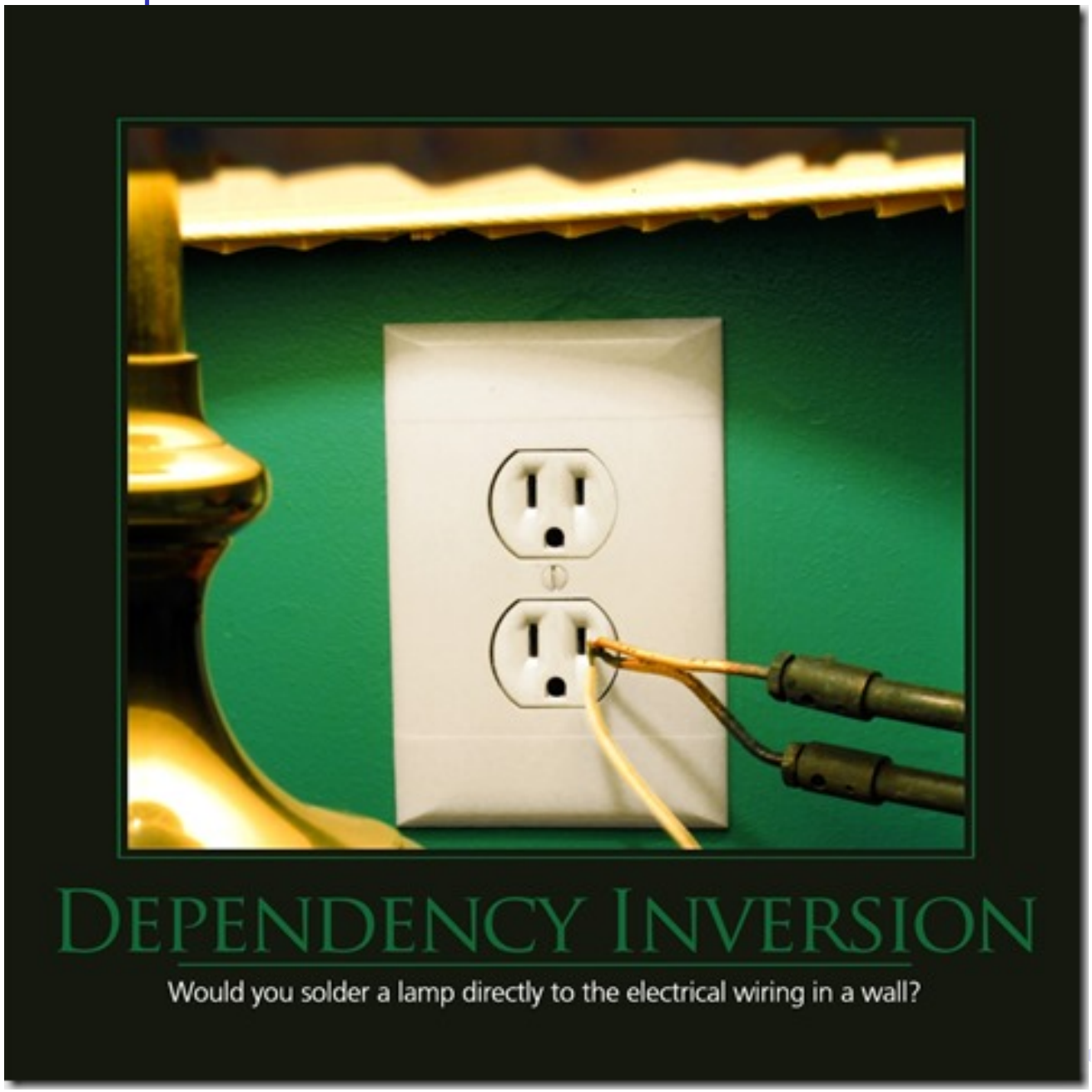
Принцип разделения интерфейса

Решение

```
interface IPrinter{
    void Print();
}
interface IFax{
    void Fax();
}
interface IScanner{
    void Scan();
}
```

Outline

Принцип инверсии зависимостей



Принцип инверсии зависимостей

- Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций.
- Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Принцип инверсии зависимостей

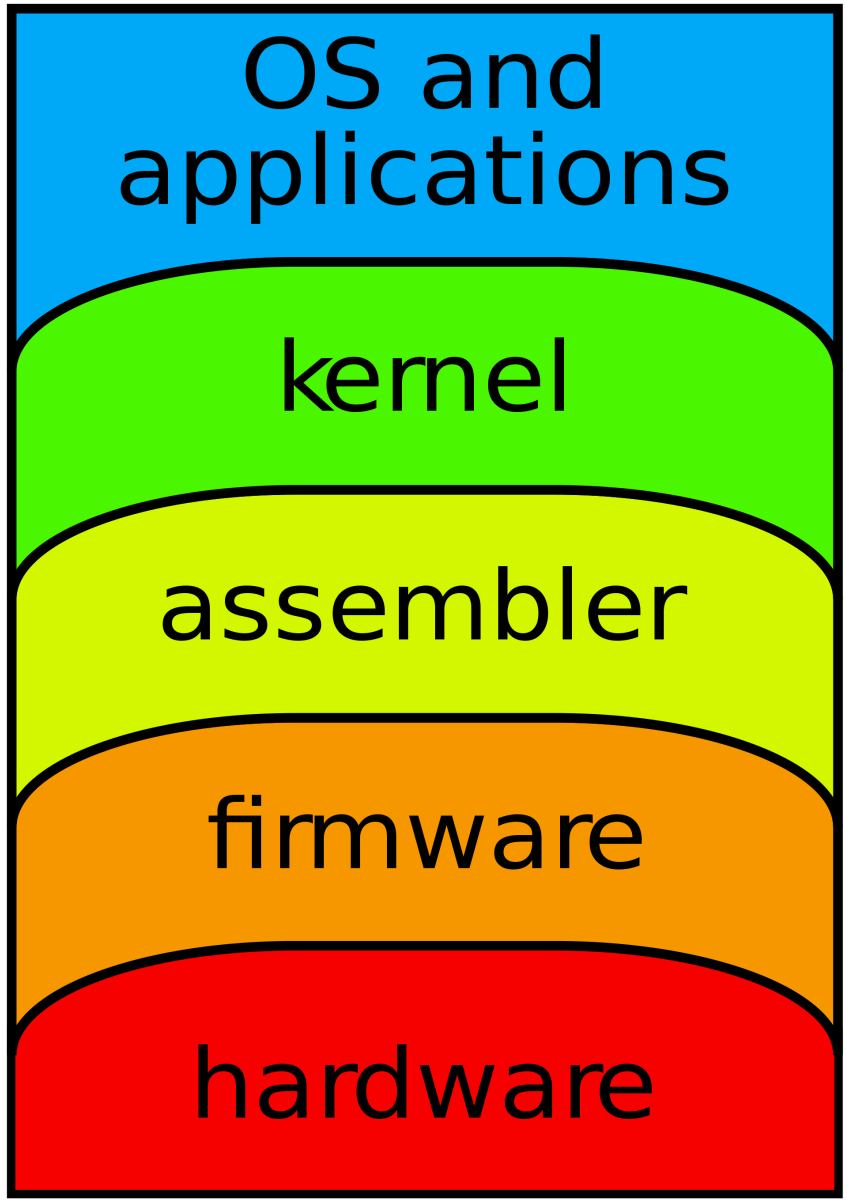
- Инверсия зависимости используется в фреймворках
- Фреймворк управляет кодом программиста, а не программист управляет фреймворком
- Фреимворк здесь - модуль верхнего уровня, код программиста - модуль нижнего уровня
- Изменяя код нижнего уровня не приходится вносить изменения в фреймворк

Уровни абстракции

Абстракция — это модель некоего объекта или явления реального мира, откидывающая незначительные детали, не играющие существенной роли в данном приближении **Уровень**

абстракции — это ступень приближения.

Пример уровней абстракции



Dependency injection

Плохой пример

```
class Worker {
        public void work() {
                // ....working
        }   }
class Manager {
        Worker worker;
        public void setWorker(Worker w) {
                worker = w;         }

        public void manage() {
                worker.work();}
}

// Добавим класс SuperWorker
class SuperWorker {
        public void work() {
                //.... working much more
}   }
```

Проблема?

Dependency injection

Плохой пример

```
class Worker {
        public void work() {
                // ....working
        }   }
class Manager {
        Worker worker;
        public void setWorker(Worker w) {
                worker = w;         }

        public void manage() {
                worker.work();}
}

// Добавим класс SuperWorker
class SuperWorker {
        public void work() {
                //.... working much more
}   }
```

Проблема?

С классом SuperWorker класс Manager не работает...

Dependency injection

Хороший пример

```
interface IWorker {
        public void work();}

class Worker implements IWorker{
        public void work() {
                // ....working
        }}

class SuperWorker  implements IWorker{
        public void work() {
                //.... working much more
        }}

class Manager {
        IWorker worker;
        public void setWorker(IWorker w) {
                worker = w;}
        public void manage() {
                worker.work();
        }}
```

Dependency injection

Выводы

- Класс Manager сам не должен создавать экземпляров класса Worker. Ведь Worker может поменяются
- Иначе получается, что модуль верхнего уровня зависит от модуля нижнего уровня
- Метод класса Manager должны принимать существующий экземпляр класса
- Однако и в этом случае изменение в Worker могут нарушить работу класса Manager. Например поменяется сигнатура методов, или вообще класс целиком.
- Поэтому класс Manager должен указывать в методе *интерфейс*, а не конкретный класс. Работать Manger будет с методами интерфейса.
- Тогда он сможет работать с любым классом, который реализует заданный интерфейс.

Dependency injection

Выводы

- Такой подход называется **Dependency injection (DI)**
- Dependency injection - процесс предоставления внешней зависимости программному компоненту.
- Внешняя зависимость в примере - класс Worker
- Компонент, добавляют дают внешнюю зависимость - класс Manager
- Уровень абстракции здесь представлен интерфейсом IWorker

Ссылки и литература

1. From STUPID to SOLID Code!

Материалы курса

Слайды, вопросы к экзамену, задания, примеры

github.com/VetrovSV/OOP