الجمهورية العربية السورية
جامعة حمص
الكلية التطبيقية
قسم تقانات الحاسوب
السنة الرابعة

الفصل الاول
٢٠٢٥- ٢٠٢٦

هندسة برمجيات متقدمة – القسم العملي

Solid Design Principle

المحاضرة الخامسة

مدرسو المقرر

م.روز المشرقي

م.جنان الكردي

مبادئ تصميم البرمجيات

هي مجموعة من الارشادات التي تساعدنا على تجنب التصميم السيء الذي يتصف ب:

- الصلابة (Rigidity): التغيير صعب فكل تغيير يؤثر على اجزاء اخرى كثيرة من النظام
- الهشاشة (Fragility): اي تعطل اجزاء غير متوقعة من النظام
- الجمود (immobility): صعوبة اعادة استخدام الكود في تطبيق اخر لانه لايمكن فصله عن التطبيق الحالي

فالهدف من هذه المبادئ :

- جعل الكود قابل للصيانة (maintainable)
- وسهل التوسيع (extensible)
- اقل ترابط (coupling) واكثر تجريد (abstraction)

مما يسهل اختبار الكود وتطويره على المدى الطويل

SOLID

وهي اختصار لخمس مبادئ اساسية في تصميم البرمجيات كائنية التوجه وضعها المبرمج " Robert martin "

1- SRP : Single Responsibility Principle

2- OCP : Open Close Principle

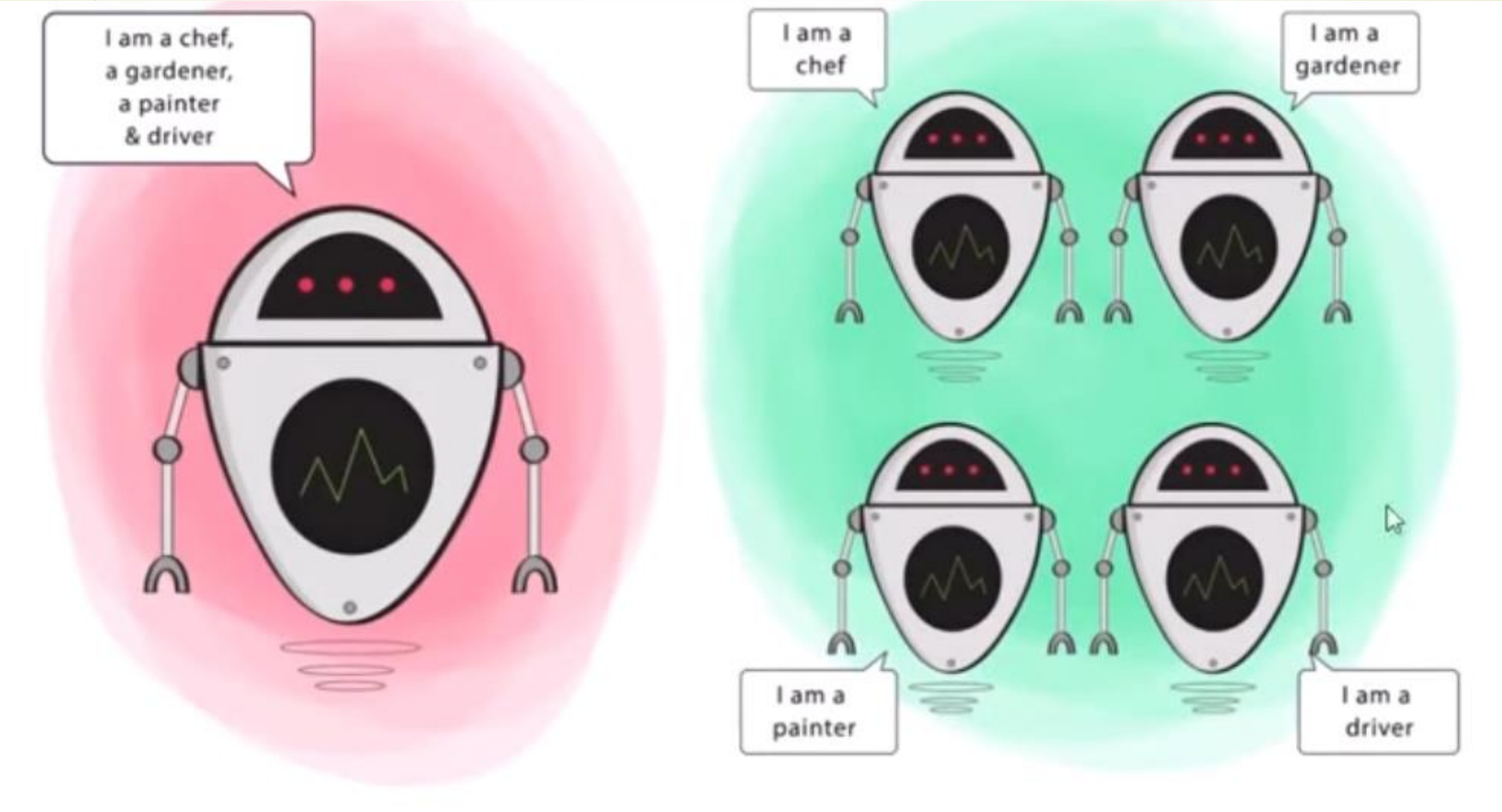
3-LSP : Liskov's Substitution Principle

4-ISP : Interface Segregation Principle

5-DIP : Dependency Inversion Principle

Single Responsibility Principle(SRP)

ال class يجب ان يكون له مسؤولية واحدة (وظيفة واحدة) ويكون هناك سبب واحد لتغييره
وتطبيق هذه المبدأ يفيد في ال testing و الlower coupling و easier to understand و organized
وسهولة الصيانة وسهولة اكتشاف الاخطاء



مثال برمجي:

```
public class Customer {

    private String name;

    // getter and setter methods ...

    // This is a Responsibility
    public void storeCustomer(String customerName) {
        // store customer into a database ...
    }

    // This is another Responsibility
    public void generateCustomerReport(String customerName) {
        // generate a report ...
    }
}
```


public class Customer {
private String name;
// getter and setter methods ...
}


public class CustomerDB {
public void storeCustomer(String customerName) {
// store customer into a database ...
}
}

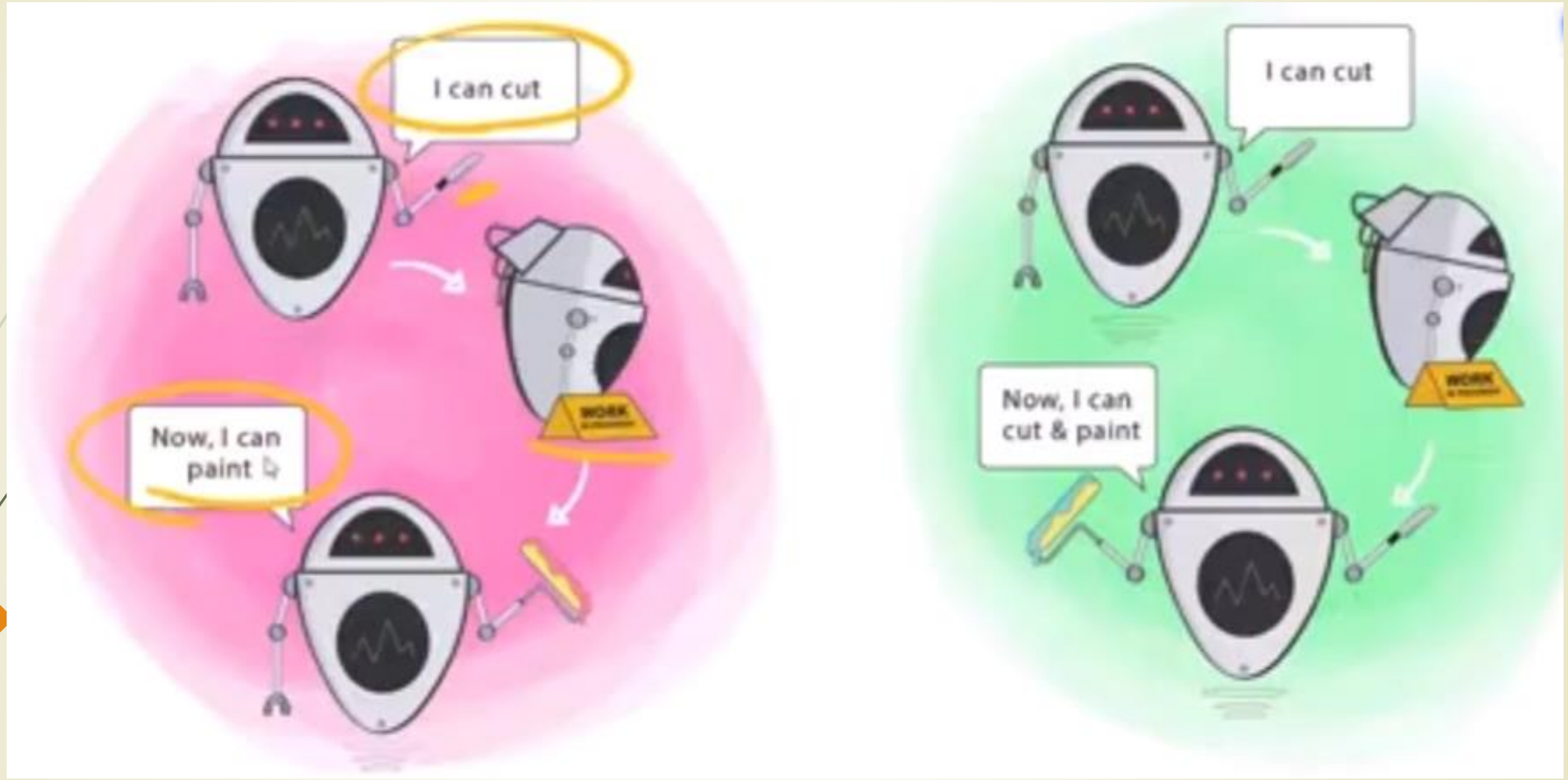

public class CustomerReportGenerator {
public void generateReport(String customerName) {
// generate a report ...
}
}

Open Close Principle(OCP)

الكائنات البرمجية مثل Function , Modules , Classes…..etc يجب ان تكون مفتوحة للتوسع ومغلقة للتعديل

حيث يمكننا اضافة وظائف جديدة (سلوك جديد) للكائنات البرمجية ولاضافة هذا السلوك الجديد لسنا بحاجة الى التعديل على الكود الموجود مسبقا ولتطبيق هذا المبدأ نحتاج لاستخدام Abstraction و polymorphism

مثلا اذا كانت لديك Library تحتوي على مجموعة من ال classes فهناك العديد من الاسباب التي تدفعك الى تفضيل توسيعها دون تغيير الكود الموجود مسبقا


I can cut
Now, I can paint
I can cut
Now, I can cut & paint

مثال برمجي:

```java
public class Square {

    private int side;

    // getter and setter methods...
}
```

```java
public class Rectangle {

    private int width;
    private int height;

    // getter and setter methods...
}
```

```java
public class ShapePrinter {

    public void drawShape(Object shape) {

        if (shape instanceof Rectangle) {
            // Draw Rectangle...
        } else if (shape instanceof Square) {
            // Draw Square...
        }
    }
}
```

```java
public abstract class Shape {

    abstract void draw();

}
```

```java
public class ShapePrinter {

    public void drawShape(Shape shape) {
        shape.draw();
    }

}
```

```
public class Square extends Shape {

    private int side;

    // getter and setter method

    @Override
    public void draw() {
        // Draw the Square
    }

}
```

```
public class Rectangle extends Shape {

    private int width;
    private int height;

    // getter and setter methods...

    @Override
    public void draw() {
        // Draw the Rectangle...
    }

}
```

Liskov's Substitution Principle(LSP)

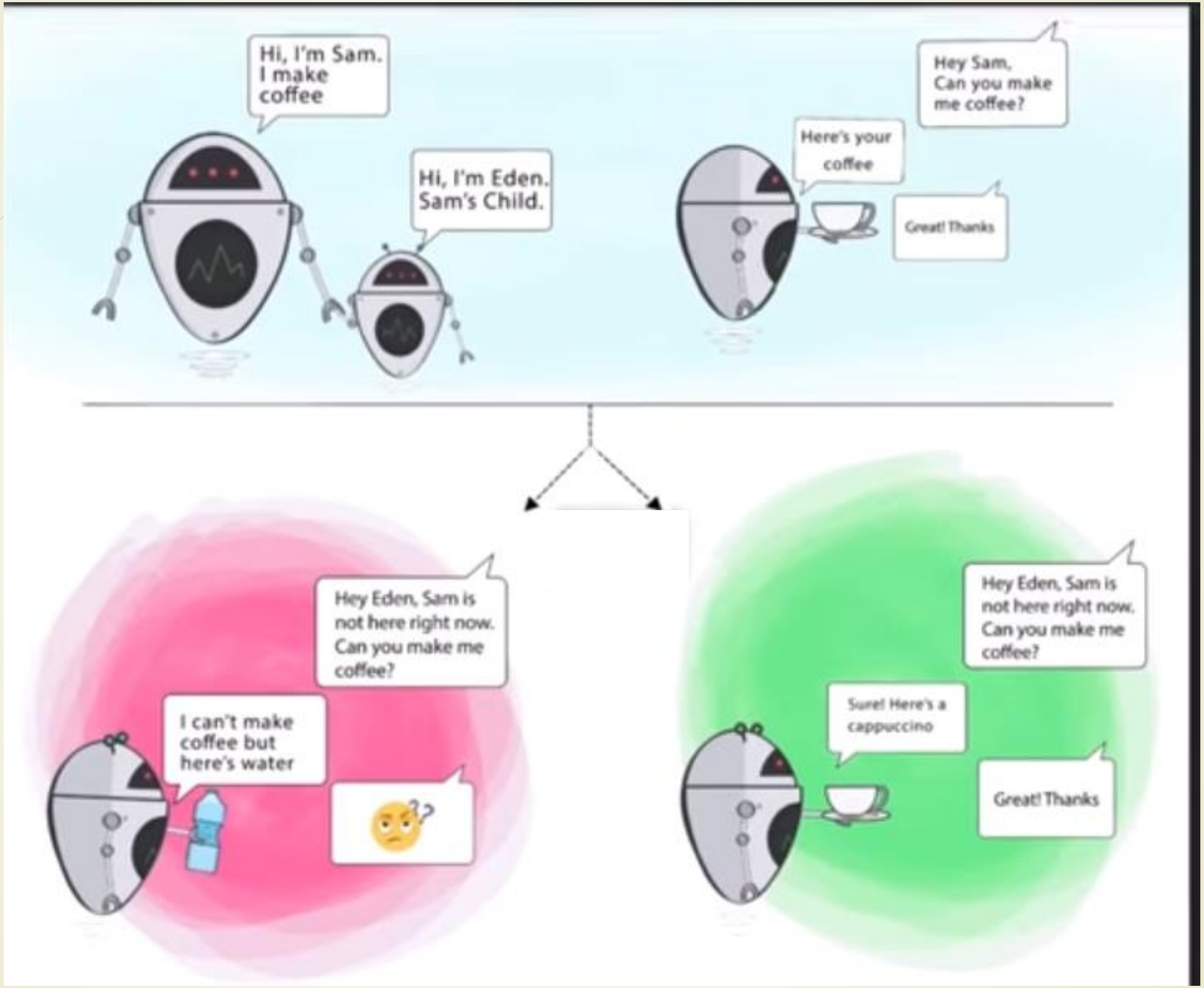
يجب ان تكون الاغراض المشتقة (في الصف الابن) قابلة للاستبدال تماما بالاغراض الاساسية من الصف

الاب

اذا كان لدينا كلاس S هو subclass من الكلاس Tفال objects الخاصة ب T يمكن استبدالها ب

objectsمن النمط S دون مشاكل

```
T t1=new T
T t1=new S
```


Hi, I'm Sam. I make coffee
Hi, I'm Eden. Sam's Child.
Hey Sam, Can you make me coffee?
Here's your coffee
Great! Thanks
Hey Eden, Sam is not here right now. Can you make me coffee?
I can't make coffee but here's water
Hey Eden, Sam is not here right now. Can you make me coffee?
Sure! Here's a cappuccino
Great! Thanks

مثال برمجي:

```java
public class Rectangle {

    private int width;
    private int height;

    public void setWidth(int width) {
        this.width = width;
    }

    public void setHeight(int height) {
        this.height = height;
    }

    public int getArea() {
        return width * height;
    }
}
```

```java
public class Square extends Rectangle {

    @Override
    public void setWidth(int width) {
        super.setWidth(width);
        super.setHeight(width);
    }

    @Override
    public void setHeight(int height) {
        super.setWidth(height);
        super.setHeight(height);
    }
}
```

```java
public class LiskovSubstitutionTest {

    public static void main(String args[]) {
        Rectangle rectangle = new Rectangle();
        rectangle.setWidth(2);
        rectangle.setHeight(5);

        if (rectangle.getArea() == 10) {
            System.out.println(rectangle.getArea());
        }
    }
}
```

```java
public class LiskovSubstitutionTest {

    public static void main(String args[]) {
        Rectangle rectangle = new Square(); // Square
        rectangle.setWidth(2);
        rectangle.setHeight(5);

        if (rectangle.getArea() == 10) {
            System.out.println(rectangle.getArea());
        }
    }
}
```

```java
public interface Shape {
    int area();
}
```

```java
public class Square implements Shape {

    private int size;

    public void setSize(int size) {
        this.size = size;
    }

    @Override
    public int area() {
        return size * size;
    }
}
```

```java
public class Rectangle implements Shape {

    private int width;
    private int height;

    public void setWidth(int width) {
        this.width = width;
    }

    public void setHeight(int height) {
        this.height = height;
    }

    @Override
    public int area() {
        return width * height;
    }
}
```

Interface Segregation Principle(ISP)

يهدف الى تقسيم ال interfaces الكبيرة الى interfaces صغيرة محددة حيث لا ينبغي اجبار العملاء على الاعتماد على واجهات لا يستخدمونها

عند كتابة الواجهات يجب اضافة الدوال التي يجب ان تكون موجودة فقط اذا اضفنا دوال لا يجب ان تكون موجودة فسيتعين على ال classes التي تحقق الواجهة تنفيذ تلك الدوال ايضا

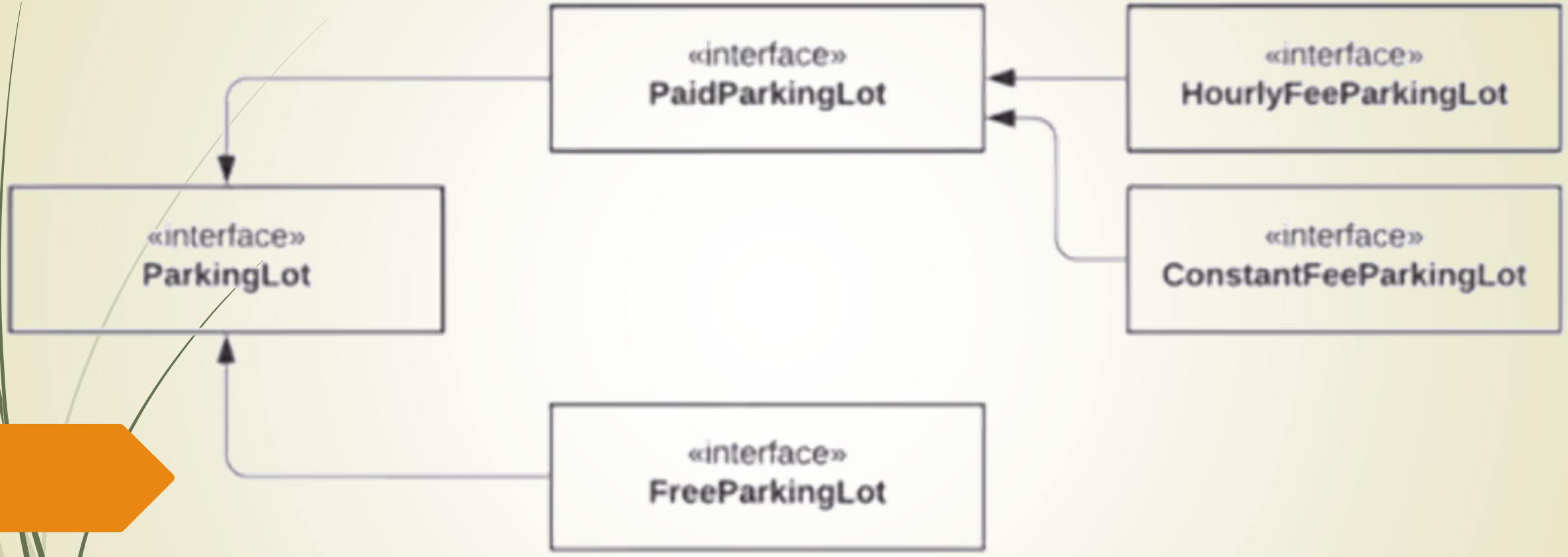
وعند تحقيق هذا المبدأ يستطيع اي مكون برمجي اخر القيام ب override لل methods التي يحتاجونها


EXERCISES
All Robots
- spin around
- rotate arms
- wiggle antennas
Oops! But I don't have antennas
EXERCISES
Robots that can spin around
Robots that can rotate arms
Robots that can wiggle antennas
- spin around
- rotate arms
- wiggle antennas
Awesome!

مثال برمجي:

```java
public interface ParkingLot {

    void parkCar(); // Decrease empty spot count by 1

    void unparkCar(); // Increase empty spots by 1

    void getCapacity(); // Returns car capacity

    double calculateFee(Car car); // Returns the price based on number of hours

    void doPayment(Car car);

}
```

```java
public class FreeParking implements ParkingLot {

    @Override
    public void parkCar() {

    }

    @Override
    public void unparkCar() {

    }

    @Override
    public void getCapacity() {

    }

    @Override
    public double calculateFee(Car car) {
        return 0;
    }

    @Override
    public void doPayment(Car car) {
        throw new Exception("Parking lot is free");
    }
}
```

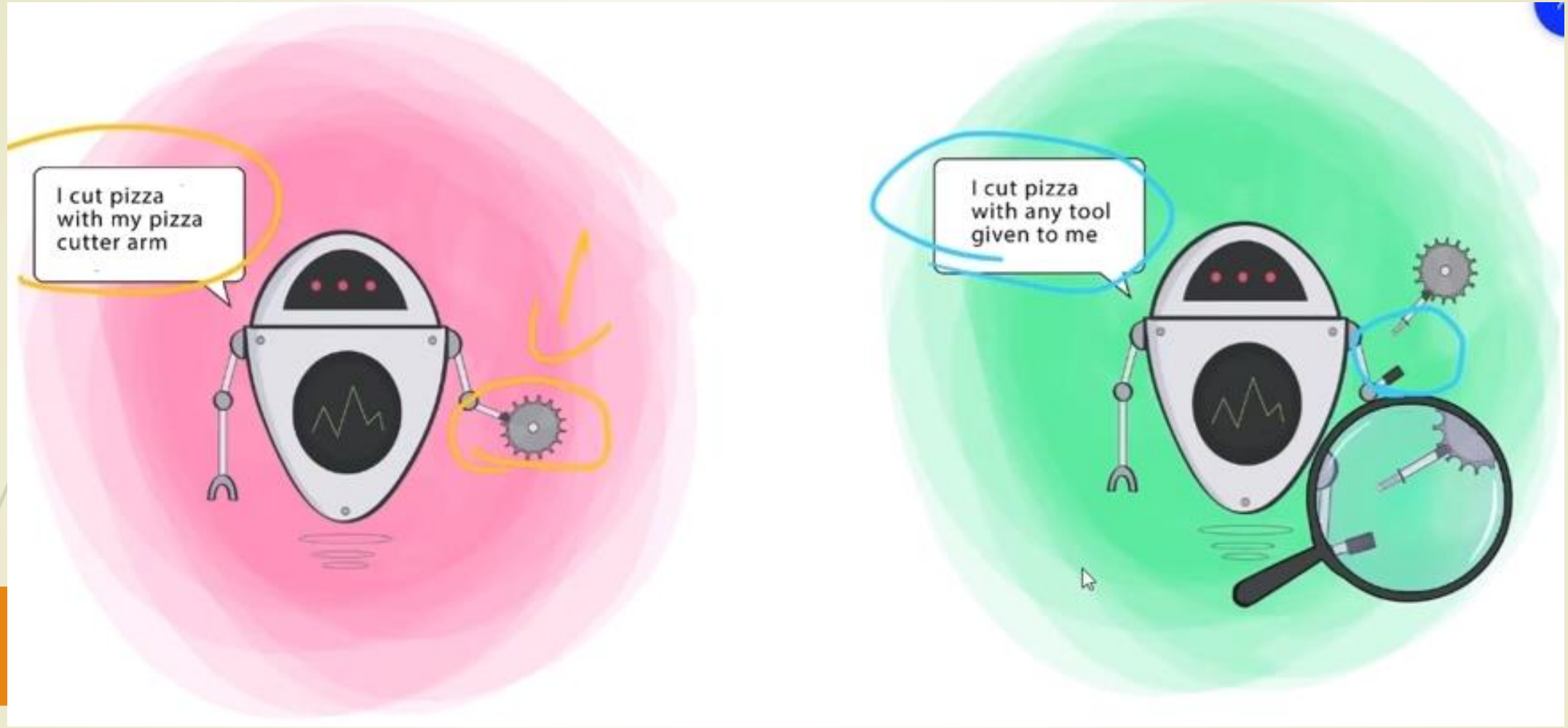

«interface»
PaidParkingLot
«interface»
HourlyFeeParkingLot
«interface»
ParkingLot
«interface»
ConstantFeeParkingLot
«interface»
FreeParkingLot

Dependency Inversion Principle(DIP)

يجب الا تعتمد ال high level module على ال low level module وكلاهما يجب ان يعتمد على ال abstraction

ال abstraction يجب الا يعتمد على التفاصيل (concrete implementation)

وال concrete يجب ان يعتمد على ال abstraction


I cut pizza with my pizza cutter arm
I cut pizza with any tool given to me

مثال برمجي:

```
public class DeliveryDriver {

    public void deliverProduct(Product product){
        // deliver product...
    }
}
```

```
public class DeliveryCompany {

    public void sendProduct(Product product) {

        DeliveryDriver deliveryDriver = new DeliveryDriver();

        deliveryDriver.deliverProduct(product);

    }
}
```

```java
public interface DeliveryService {

    void deliverProduct(Product product);

}
```

```java
public class DeliveryDriver implements DeliveryService {

    @Override
    public void deliverProduct(Product product) {
        // deliver product...
    }
}
```


public class DeliveryCompany {
private DeliveryService deliveryService;
public DeliveryCompany(DeliveryService deliveryService) {
this.deliveryService = deliveryService;
}
public void sendProduct(Product product) {
this.deliveryService.deliverProduct(product);
}
}