



بخش ۱۵: تکامل یک طراحی از کلاس انتزاعی به رابط

تهیه شده توسط: سید سجاد پیراهاش

شما دیگر یک کدنویس نیستید. شما یک طراح هستید.

در این بخش، ما از نوشتن کد فراتر می‌رویم و به هنر تصمیم‌گیری استراتژیک در طراحی نرم‌افزار وارد می‌شویم.

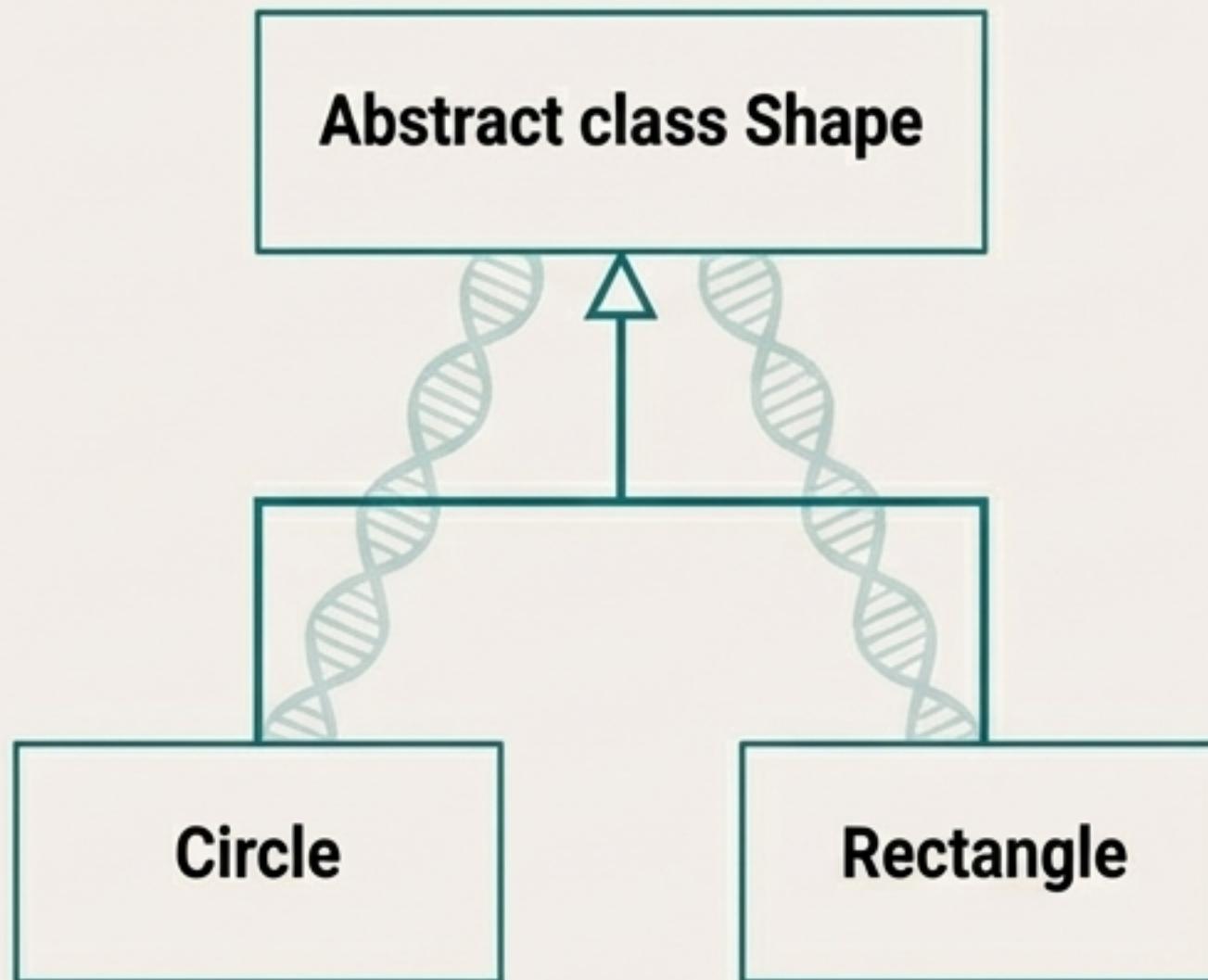
سفر ما:

- 1 **یک شروع خوب:** ساخت یک برنامه نقاشی با استفاده از کلاس انتزاعی (Abstract Class).
- 2 **رویارویی با محدودیت:** کشف دیواری که طراحی اولیه ما را متوقف می‌کند.
- 3 **یک انقلاب فکری:** معرفی رابط (Interface) برای رسیدن به انعطاف‌پذیری نهایی.
- 4 **کسب خرد:** یادگیری اینکه چه زمانی و چرا یکی را بر دیگری انتخاب کنیم.

آماده شوید تا دیدگاه خود را برای همیشه تغییر دهید.



نسخه ۱: بنیاد IS-A (هست یک)



سناریو: برنامه نقاشی (نسخه ۱)

فرض اولیه: هر چیزی که قابل رسم است، یک شکل هندسی است.
این یک رابطه "IS-A" (هست یک) کلاسیک است.

- Circle IS-A Shape

- Rectangle IS-A Shape

چرا یک انتخاب عالی برای شروع است؟

- مشترک: همه اشکال ویژگی‌های مشترک (فیلدها) مانند color و

- origin را به ارث می‌برند.

- کد مشترک: می‌توانیم متدهای پایه را در Shape پیاده‌سازی کنیم.

- قرارداد رفتاری: Shape فرزندانش را مجبور می‌کند متدهای انتزاعی draw() را

- پیاده‌سازی کنند.

این طراحی تمیز، قابل فهم و کاملاً شی‌گرا است.

پیاده‌سازی کد با Abstract Class

Code Sandbox

کد ساختاری (The Foundation)

```
// The shared "DNA" and contract
abstract class Shape {
    protected Color color;
    protected Point origin;
    // ... constructor ...
    public abstract void draw();
}

public class Circle extends Shape {
    // ... fields & constructor ...
    @Override
    public void draw() {
        // Drawing logic for a circle...
    }
}
```

کد اجرایی (The Application)

```
// The drawing area works with any Shape
public class PaintingArea {
    private List<Shape> shapes;

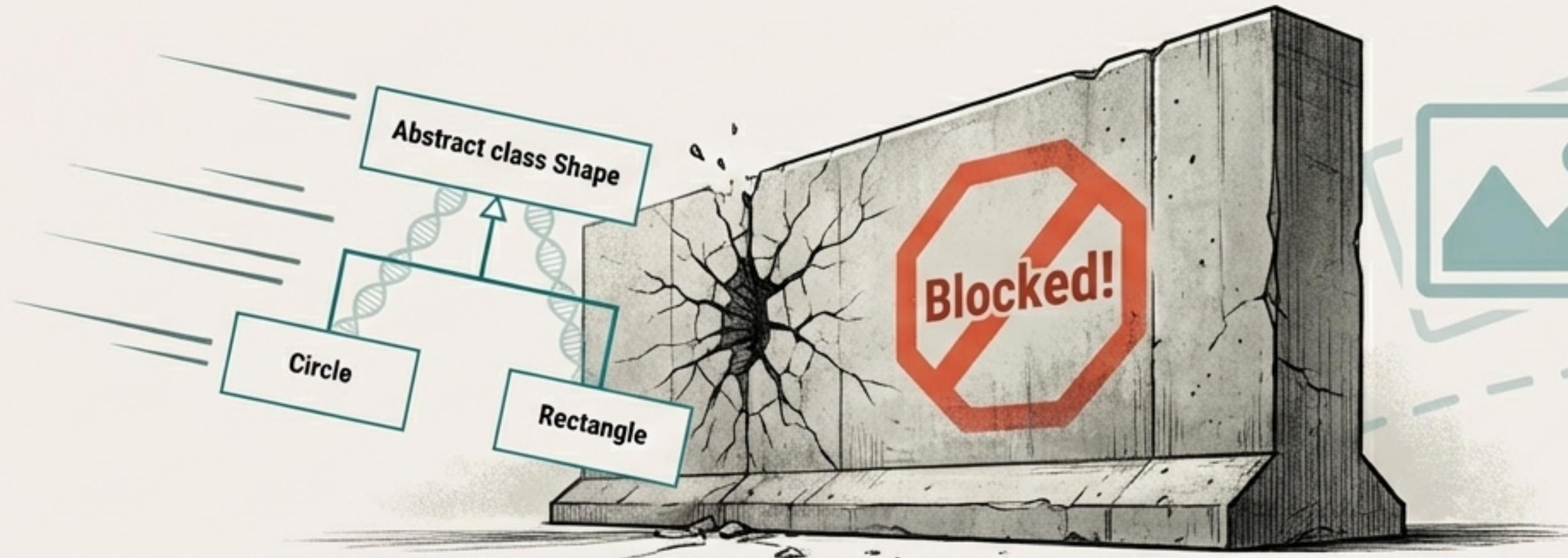
    public PaintingArea() {
        shapes = new ArrayList<>();
    }

    public void addShape(Shape s) {
        shapes.add(s);
    }

    public void drawAll() {
        for (Shape s : shapes) {
            s.draw(); // Polymorphism in action!
        }
    }
}
```

تحلیل: یک سیستم یکپارچه که به زیبایی با استفاده از چندریختی (Polymorphism) کار می‌کند. تا اینجا همه چیز عالی به نظر می‌رسد.

متن



دیوار! محدودیت بزرگ طراحی

"ما می‌خواهیم تصاویر (Images) و برچسب‌های متنی (TextLabels) را هم به صفحه نقاشی اضافه کنیم."

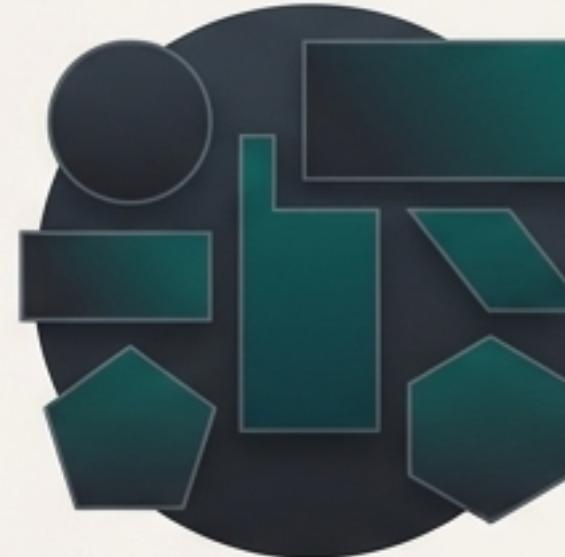
اینجا طراحی ما فرو می‌پاشد. چرا؟

- یک Image یک Shape نیست.
- یک TextLabel یک Shape نیست.

آن‌ها نمی‌توانند از Shape ارث بری کنند، به خصوص اگر از کلاس دیگری ارث برده باشند (جاوا وراثت چندگانه ندارد).

مشکل اصلی: ما "چیستی" یک شیء (اینکه یک شکل است) را با "توانایی" آن (اینکه قابل رسم است) گره زدیم.

طراحی ما به بنبست رسیده است.



انقلاب CAN-DO (می‌تواند...)

راه حل: جدا کردن "چیستی" از "توانایی"

"آیا این شیء می‌تواند رسم شود؟"

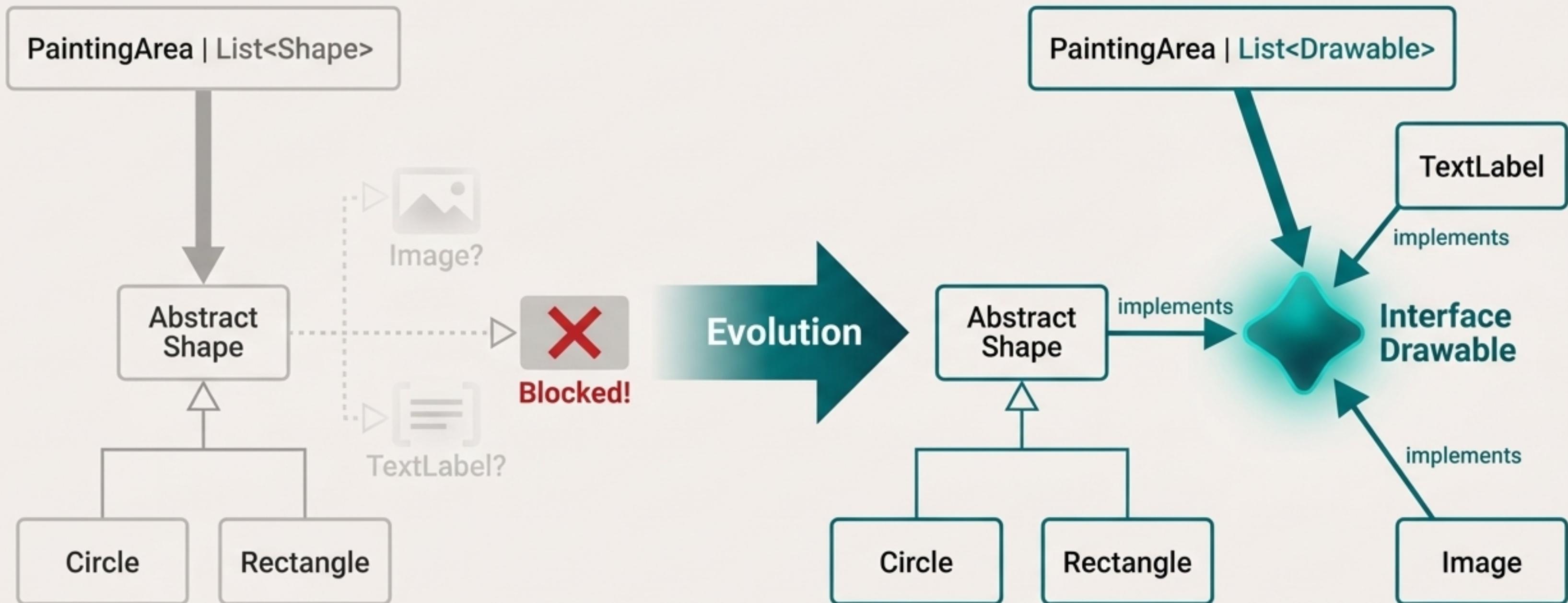
این یک رابطه "CAN-DO" (می‌تواند...) است. ما این توانایی را با یک **interface** تعریف می‌کنیم.

```
// A pure contract for a "capability"
public interface Drawable {
    void draw();
}
```

- قرارداد خالص: رابط Drawable فقط یک قول است: "هر کلاسی که من را پیاده‌سازی کند، باید متدهای draw() را داشته باشد."
- انعاف‌پذیری بی‌نهایت: هر کلاسی، از هر سلسله مراتبی، می‌تواند این قول را بدهد و این "توانایی" را به دست آورد.

این یک تغییر پارادایم است. ما در حال تعریف "مهارت" هستیم، نه "اصل و نسب".

نقشه تکامل طراحی: از یک سلسله مراتب سخت به یک شبکه توانایی‌ها



انقلاب طراحی: انعطاف‌پذیری با رابطه‌ها

کد ساختاری (The New Foundation)

```
public interface Drawable {  
    void draw();  
}  
  
// Shape now GAINS a capability  
abstract class Shape implements Drawable {  
    // ... color, origin fields remain ...  
}  
  
// Circle and Rectangle remain unchanged!  
  
public classTextLabel implements Drawable {  
    private String text;  
    @Override  
    public void draw() {  
        // Drawing logic for text...  
    }  
}
```

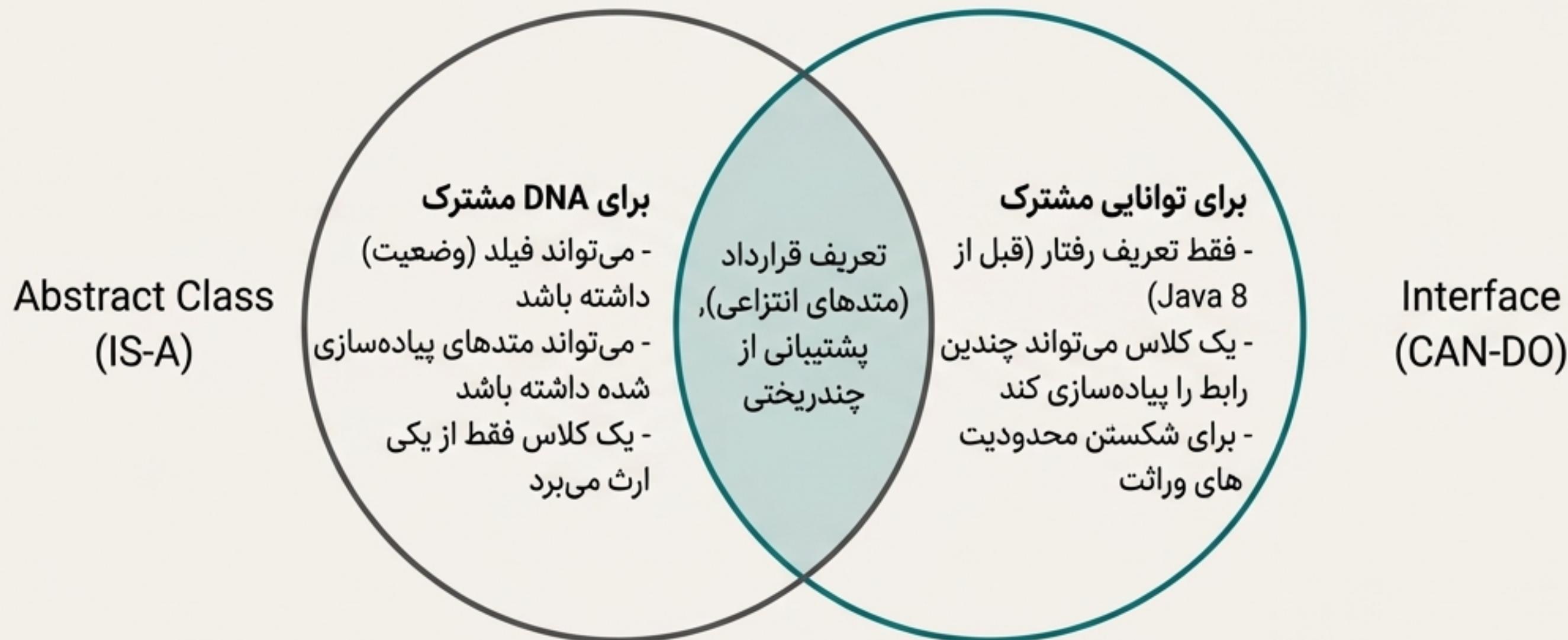
کد اجرایی (Ultimate Flexibility)

```
// The drawing area now depends on the INTERFACE!  
public class PaintingArea {  
    private List<Drawable> drawables;  
  
    public PaintingArea() {  
        drawables = new ArrayList<>();  
    }  
  
    public void addDrawable(Drawable d) {  
        drawables.add(d);  
    }  
  
    public void drawAll() {  
        for (Drawable d : drawables) {  
            d.draw(); // Still polymorphism!  
        }  
    }  
}
```

نتیجه: `PaintingArea` می‌تواند هر چیزی را که **Drawable** باشد در خود جای دهد. طراحی ما دیگر شکننده نیست، بلکه توسعه‌پذیر است.

نبرد نهایی: کلاس انتزاعی در مقابل رابط

چه زمانی از کدام یک استفاده کنیم؟



خلاصه:

- برای خانواده‌ای از کلاس‌های نزدیک به هم که **کد و وضعیت** مشترک دارند.
- برای تعریف یک **قابلیت** که می‌تواند توسط کلاس‌های **کاملاً نامرتب** به اشتراک گذاشته شود.

Program to an interface, not an implementation.

(برنامه‌نویسی برای رابط، نه برای پیاده‌سازی.)

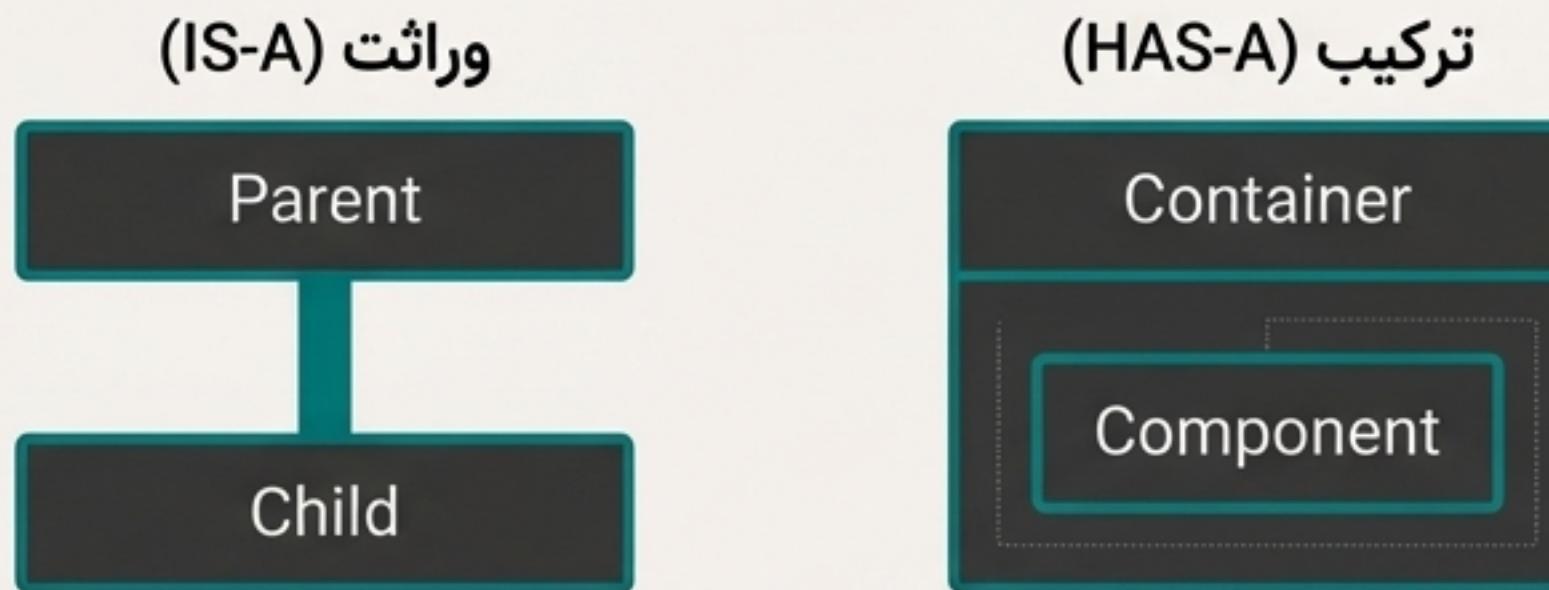
این به چه معناست؟

کد خود را به **قرارداد (Drawable)** وابسته کنید، نه به یک نوع انضمامی (Circle, Shape).

- کد شما (PaintingArea) نمی‌داند و **اهمیتی نمی‌دهد** که شیء واقعی چیست.
- تنها چیزی که اهمیت دارد این است که آن شیء به قرارداد **draw()** پاییند است.

این اصل، کلید ساخت سیستم‌های انعطاف‌پذیر، قابل نگهداری و جدا از هم (Decoupled) است.

نکته پیشرفته: ترکیب بر وراثت ارجح است



چرا طراحان مدرن جاوا اغلب رابطها را ترجیح می‌دهند؟

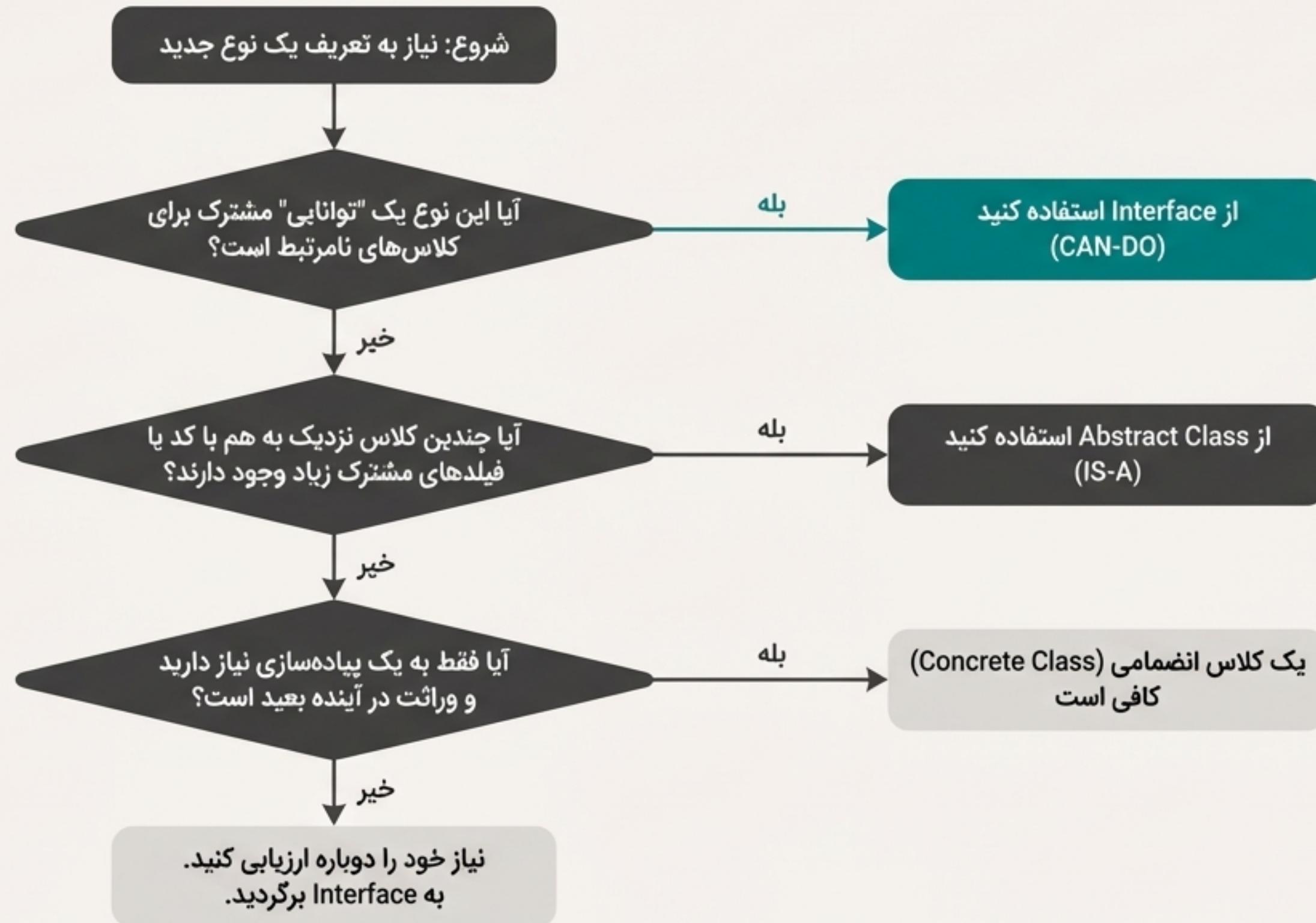
زیرا رابطها اصل قدرتمند دیگری را ترویج می‌کنند: "**Composition over Inheritance**".

- وراثت (IS-A): یک رابطه سفت و سخت که در زمان کامپایل تعیین می‌شود. انعطاف‌پذیری کمی دارد.
- ترکیب (HAS-A): یک رابطه پویا. شما می‌توانید رفتار را در زمان اجرا با استفاده از وابستگی به رابطها تغییر دهید.

مثال: به جای اینکه یک `FlyingCar` از `Car` و `Plane` ارث ببرد (که غیرممکن است)، می‌تواند رابطهای `Flyable` و `Drivable` را پیاده‌سازی کند و رفتار رانندگی و پرواز را از اشیاء دیگر "ترکیب" کند.

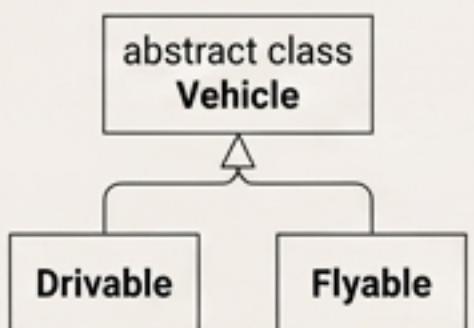
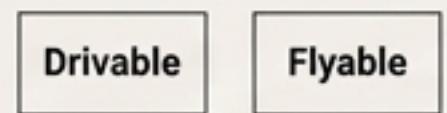
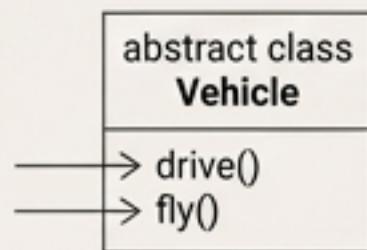
رابطها دروازه‌ای به سوی طراحی‌های بسیار پویاتر هستند.

جعبه ابزار معمار: فلوچارت تصمیم‌گیری

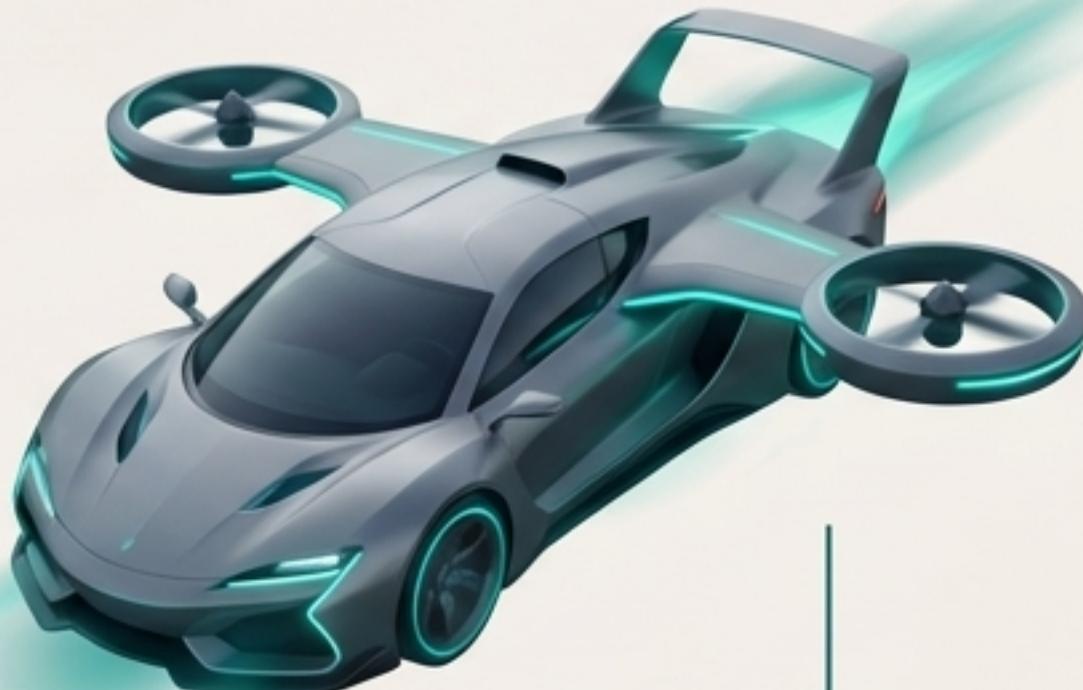


آزمون نهایی: چالش طراحی خودرو

شما طراح ارشد هستید. سه گزینه برای سیستم وسایل نقلیه (Vehicle) به شما ارائه شده است:



- طراحی A (فقط)
`abstract class Vehicle`
با متدهای `drive()` و `fly()`
- طراحی B (فقط)
رابطهای `Drivable` و `Flyable`.
- طراحی C (ترکیبی)
`abstract class Vehicle` که `Drivable` را پیاده‌سازی می‌کند.
یک رابط جدأگانه `Flyable`.

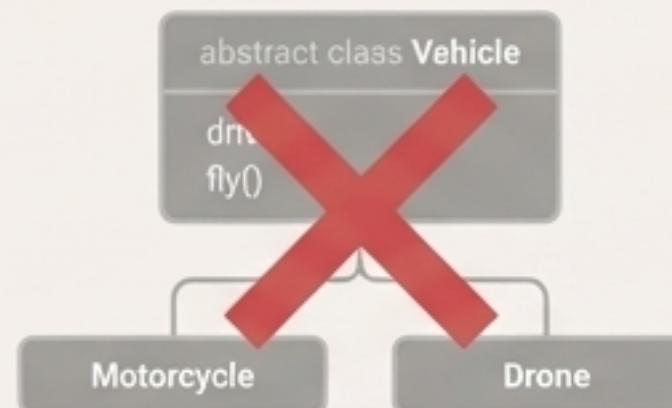


سوال استراتژیک:

برای ساخت یک `FlyingCar`
هم رانندگی می‌کند
و هم پرواز می‌کند،
کدام طراحی بهترین
انعطاف‌پذیری
را برای آینده فراهم
می‌کند؟ چرا؟

پاسخ چالش: طراحی C (ترکیبی) بهترین انتخاب است.

: (Abstract Class طراحی A (فقط

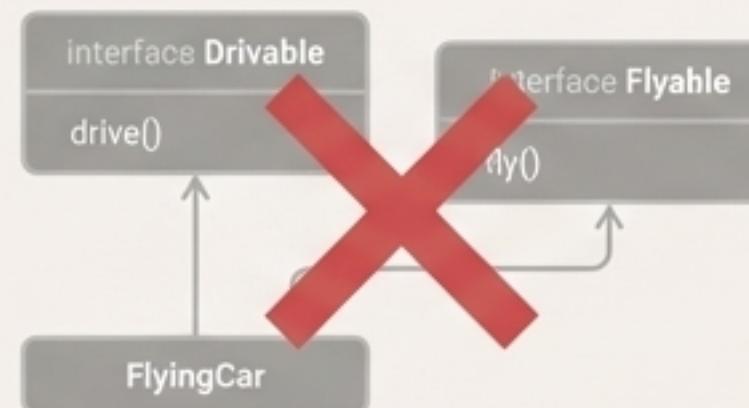


یک 'Motorcycle' مجبور می‌شود یک متد 'fly()' خالی داشته باشد و یک 'Drone' مجبور به داشتن 'drive()' خالی می‌شود. این طراحی غیرمنعطف است.

```
class FlyingCar extends Vehicle implements Flyable { ... }
```

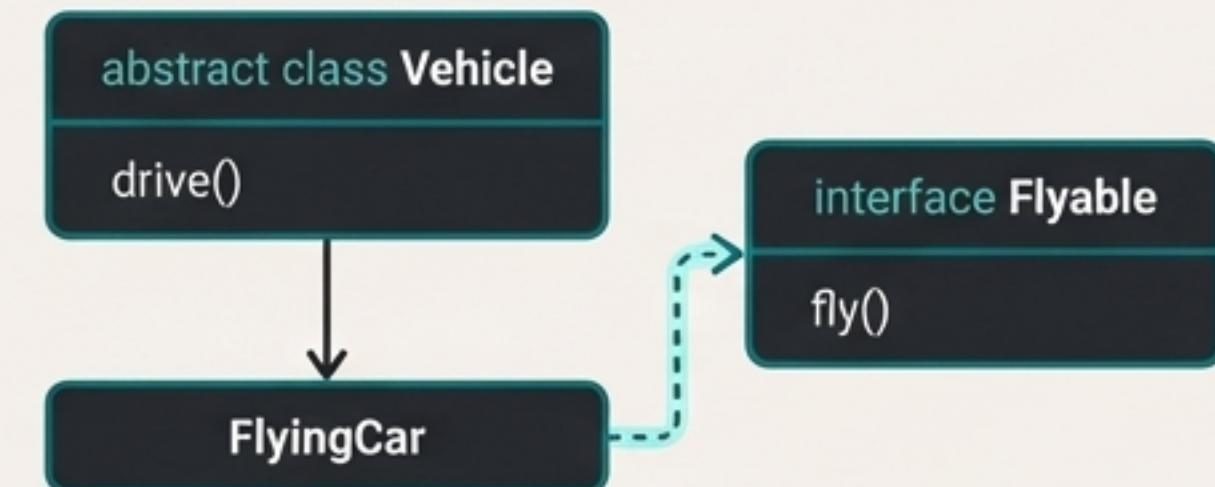
این نمونه‌ای عالی از یک طراحی بالغ و آینده‌نگر است.

: (Interface طراحی B (فقط



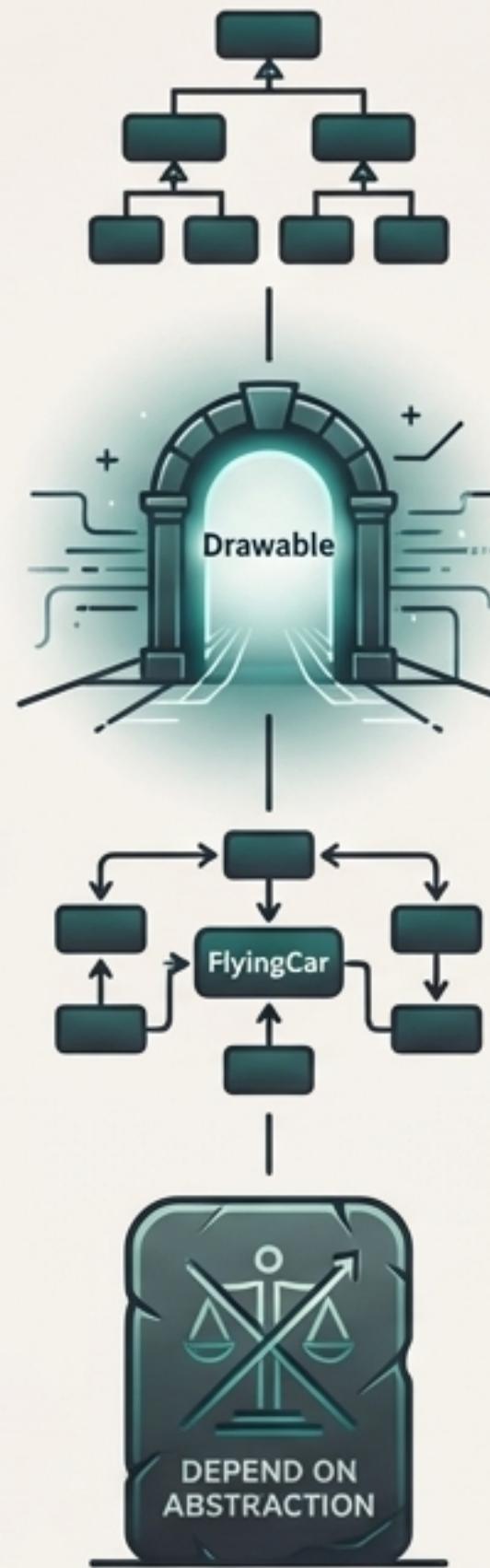
بسیار انعطاف‌پذیر است، اما اگر همه وسائل نقلیه زمینی (Car, Motorcycle) کد مشترکی داشته باشند، منجر به تکرار کد می‌شود.

: (ترکیبی) قدرت طراحی C (ترکیبی):



- بهترین هر دو جهان: کد مشترک رانندگی را در class 'Vehicle' از گذاریم (کاهش تکرار).
- انعطاف‌پذیری برای آینده: "توانایی" پرواز (Flyable) می‌تواند به هر کلاسی، چه 'Vehicle' باشد (مثل 'FlyingCar') و چه نباشد (مثل 'Drone') اضافه شود.
- به سادگی از 'Vehicle' ارث می‌برد (برای رفتار رانندگی) و رابط 'Flyable' را پیاده‌سازی می‌کند.

خلاصه سفر شما



- ✓ شروع با (IS-A) Abstract Class: عالی برای خانواده‌های نزدیک به هم با کد و وضعیت مشترک.
- ✓ تکامل با (Interface (CAN-DO)): کلید شکستن محدودیت‌های وراثت و تعریف "توانایی" برای کلاس‌های نامرتبه.
- ✓ اصل طلایی: همیشه کد خود را به رابط وابسته کنید، نه به پیاده‌سازی. این راز انعطاف‌پذیری است.
- ✓ طراحی قدرتمند: بهترین سیستم‌ها اغلب از ترکیب هوشمندانه هر دو ابزار استفاده می‌کنند تا هم کد را به اشتراک بگذارند و هم انعطاف‌پذیر باقی بمانند.
- شما به خرد طراحی دست یافتید. اکنون بروید و سیستم‌هایی بسازید که برای آینده طراحی شده‌اند.