شرح عربي مبسط + أمثلة من مشاريع حقيقية

SOLID

اكتب كود سهل يتطوّر مش بس كود شغّال

هل كل تعديل صغير في مشروعك بيخلّيك تغيّر في ملفات كتير وتخاف تبوّظ حاجة تانية؟

المشكلة أحيانًا مش في لغة البرمجة ولا الـFramework، لكن في طريقة تقسيم المسؤوليات وربط أجزاء السيستم ببعض. وهنا بتساعدك مبادئ SOLID.



إيه هي SOLID؟

خمس مبادئ في **Object-Oriented Design** بتوجّه تصميم الـClasses والعلاقات بينها، علشان التغييرات تفضل محدودة ومفهومة، والـFeatures الجديدة تتضاف بأقل مخاطرة ممكنة.

Extensible
إضافة Features جديدة بمرونة.

Maintainable
صيانة وتعديل أسهل مع الوقت.

Low Coupling
تقليل الاعتماد المباشر بين الأجزاء.

Testable
اختبار كل جزء بشكل مستقل.

قبل ما نبدأ: SOLID مش لغة، ولا Framework، ولا بديل عن OOP؛ هي مبادئ تصميم بتستخدمها لما محتاج تخلي السيستم أوضح وأسهل في التطوير.

الفكرة + المشكلة + التطبيق

ابدأ بتقسيم المسؤوليات وبعدها فكّر في التوسّع

هنستخدم مثال واحد طول الدليل: مشروع **E-commerce** فيه Orders وPayments وNotifications.

S مسؤولية واحدة وسبب واضح للتغيير

Single Responsibility Principle

المبدأ مش معناه إن كل Class يكون فيها Method واحدة؛ معناه إن كل Class يكون ليها سبب أساسي واحد للتغيير ومسؤولية مترابطة.

مثال عملي من الـE-commerce

لو **OrderService** بتنشئ الأوردر، وتحسب الضريبة، وتبعت إيميل، وتتكلم مع شركة الشحن: أي تغيير في الإيميلات أو الشحن ممكن يجبرك تعدّل في نفس الـClass. الأفضل تفصل مسؤوليات زي OrderService وNotificationService وShippingService.

النتيجة: كل جزء بيتغيّر لسبب مفهوم، واختباره وصيانته أسهل.

خد بالك: مش مطلوب تقسّم كل وظيفة صغيرة لـClass مستقل. المهم فصل الأسباب المختلفة للتغيير.

O ضيف سلوك جديد من غير ما تكسر الموجود

Open / Closed Principle

خلّي الأجزاء المستقرة مفتوحة للتوسّع، لكن مش محتاجة تعديلات متكررة كل ما يظهر نوع جديد من نفس السلوك.

مثال عملي من الـE-commerce

عندك دفع بالكارت، والعميل طلب Paymob ثم PayPal. بدل ما كل إضافة تجبرك تزود if / else في كود الـCheckout، اعمل عقد موحّد للدفع وتنفيذ مستقل لكل بوابة. ساعتها بتضيف طريقة دفع جديدة من غير ما تعيد كتابة منطق الطلبات.

النتيجة: توسّع أسهل ومخاطرة أقل عند إضافة مزوّد جديد.

مش معناها «ممنوع تعديل الكود القديم»؛ المعنى إنك تصمم نقاط التوسّع المتوقعة بشكل ذكي.

اسأل نفسك: هل كل Feature جديدة بتخلّيني أرجع أعدّل نفس الملف الضخم؟ لو آه، راجع مسؤولياته ونقاط التوسّع فيه.

التعامل بين العقود والتنفيذات

مش كفاية نفس الاسم...
لازم نفس السلوك المتوقع

الاتنين دول مهمين جدًا لما يكون عندك Interfaces أو وراثة أو أكتر من Implementation.

L البديل لازم يحترم العقد المتوقع
Liskov Substitution Principle

لو Class بتوعد بسلوك معيّن، أي Implementation أو Subclass بديلة لازم تقدر تستخدمها في نفس المكان من غير ما تفاجئ باقي النظام بسلوك مخالف.

مثال عملي من الدفع

الـCheckout بيتعامل مع PaymentGateway على أساس إن عملية الدفع إما تنجح أو ترجع فشل معروف. لو تنفيذ جديد بيقول «نجاح» من غير تحصيل، أو بيرمي Exception غير متوقّع في حالة عادية، فهو كسر العقد حتى لو أسماء الـMethods متطابقة.

النتيجة: تقدر تبدّل التنفيذات من غير مفاجآت للمستهلك.

الالتزام بالتوقيع وحده مش كفاية؛ لازم شروط الاستخدام والنتائج المتوقعة تكون متسقة.

I ما تجبرش Class على حاجات مش بتخصّها
Interface Segregation Principle

بدل Interface ضخمة فيها وظائف كتير، صمّم Interfaces أصغر ومحددة، بحيث كل Class تعتمد فقط على الوظائف اللي هي محتاجاها.

مثال عملي من الدفع

تخيل Interface واحدة فيها ()charge و()refund و()saveCard. بعض طرق الدفع ممكن تدعم التحصيل فقط؛ مش منطقي تجبرها تنفّذ حفظ الكارت أو الاسترجاع وترمي «غير مدعوم». الأفضل فصل عقود التحصيل والاسترجاع وحفظ الكروت حسب الإمكانيات الفعلية.

النتيجة: عقود أصغر، وتنفيذات أبسط، ومن غير وظائف وهمية.

اختصارها بالعامية: كل جزء يتعامل مع الحاجات اللي بتخصّه فقط، مش مع واجهة متضخّمة.

الفرق السريع: L بتسأل «هل البديل بيتصرف صح؟» أما I بتسأل «هل الـInterface فيها حاجات مش محتاجها؟»

المبدأ الخامس

اعتمد على عقد ثابت...
مش على شركة أو Class بعينها

D

افصل منطق البيزنس عن التفاصيل الخارجية
Dependency Inversion Principle

الـBusiness Logic العالية المستوى والـImplementations التفصيلية تعتمد على Abstraction مشتركة، بدل ما الخدمة الأساسية تكون مربوطة مباشرة بتفاصيل Stripe أو Paymob.

مثال من Checkout

لو CheckoutService بنفسها بتنشئ new StripeGateway()، تبديل مزوّد الدفع أو كتابة Test Fake هيكون أصعب. الأفضل تستقبل عقد PaymentGateway من برّه.

DESIGN FLOW

CheckoutService (Business Logic)

depends on ↓

PaymentGateway (Interface)

implemented by ↑

PaymobGateway | StripeGateway

إيه اللي كسبته؟ الـCheckout ما بقتش تعرف تفاصيل Stripe أو Paymob. تقدر تبدّل مزوّد الدفع أو تستخدم Test Fake من غير ما تغيّر منطق الطلبات، طالما التنفيذ ملتزم بالعقد.

بعد تطبيق المبدأ

كل الخدمات تعتمد على PaymentGateway، والـService Container يحقن التنفيذ المناسب.

قبل التصميم الصح

كل خدمة بتعمل new StripeGateway() بنفسها؛ فتغيير مزوّد الدفع بيلمس أماكن كتير.

ملحوظة دقيقة: Dependency Injection أداة بتسهّل تطبيق Dependency Inversion، لكنها مش هي نفس المبدأ.

{ } مثال Laravel عملي + مراجعة

من المبدأ للتطبيق

Laravel: افصل منطق الدفع عن مزوّد الخدمة

في المثال ده CheckoutService مش بتنشئ PaymobGateway بنفسها؛ هي بتطلب عقد واحد والخدمة اللي هتنفّذه بتتحدد من Laravel.

مثال مبسط وقابل للتطوير:

```
interface PaymentGateway {
  public function charge(int $amountInCents): void;
}

class CheckoutService {
  public function __construct(
    private PaymentGateway $gateway
  ) {}

  public function pay(int $amountInCents): void {
    $this->gateway->charge($amountInCents);
  }
}

// In a Laravel service provider:
$this->app->bind(
  PaymentGateway::class,
  PaymobGateway::class
);
```

مراجعة سريعة قبل ما تبدأ أي Refactor

إمتى SOLID هتفيدك فعلًا؟

1 **كل تعديل بيؤثّر على أجزاء ملهاش علاقة؟**
ابدأ بمراجعة تقسيم المسؤوليات (S) والاعتمادات بين الخدمات (D).

2 **كل Feature جديدة بتزوّد شروط في نفس الملف؟**
فكّر في نقاط توسّع نظيفة بدل التعديل المتكرر (O).

3 **في Implementation بتكسر توقعات الكود اللي بيستخدمها؟**
راجع العقد الحقيقي والسلوك، مش مجرد أسماء الـMethods (L).

4 **Classes بتنّفذ Functions مش محتاجاها؟**
قسّم الـInterfaces لأدوار أصغر وأنسب (I).

اتعلم وراجع بنفسك

أفضل طريقة تثبّت بيها SOLID: شوف مثال... وطبّقه في مشروعك

الروابط دي قابلة للضغط جوّه ملف الـPDF، وكمان كل رابط معاه QR Code للموبايل.

English · YouTube

SOLID Principles and Best Practices — Geekific

شرح باللغة الإنجليزية، مناسب للمراجعة وتثبيت المصطلحات.

https://www.youtube.com/watch?v=HoA6aZPR5K0

عربي · 6 دروس · Playlist

SOLID Principles بالعربي — Omar Ahmed

شرح أوسع للمبادئ، مع درس مستقل لكل Principle تقريبًا.

https://www.youtube.com/playlist?list=PLwWuxCLIF_uevri_OpofVLXkRRFnZ7TSV

الرابط المرسل سابقًا

فيديو Torti Code القصير

احتفظنا بالرابط الأصلي كما طلبت، لكن لغته الإنجليزية غير مؤكدة؛ لا تعتمد عليه باعتباره شرحًا إنجليزيًا.

https://www.youtube.com/watch?v=vHDgTolPMGM

ملاحظة بخصوص رابط Torti Code: احتفظنا بالرابط اللي اتبعت في المحادثة، لكن مش مؤكّد إنه بالإنجليزي؛ علشان كده ضفنا فيديو Geekific الإنجليزي كمرجع واضح.

تبدأ منين لو أول مرة تسمع عن المبادئ؟

01 · قبل SOLID

راجع OOP: Classes وInheritance وPolymorphism وEncapsulation وInterfaces.

02 · بعد الفيديوهات

اختار Service في مشروعك، حدّد أسباب تغيّرها، وافصل الاعتمادات اللي فعلًا معطّلة التطوير.

الخلاصة: مبادئ SOLID مش Checklist لازم تنفّذها في كل ملف؛ هي طريقة تفكير بتساعدك تكتب كود أوضح، أسهل اختبارًا، وأقل تكلفة في التغيير.