SOLID হচ্ছে Object Oriented Programming এর গুরুত্বপূর্ণ ৫টি individual design pattern এর কম্বিনেশন, যা Robert Cecil Martin (যিনি uncle bob নামে পরিচিত) ২০০০ সালে তার Design Principles and Design Patterns নামক পেপারে উপস্থাপন করেন।

এর মূল উদ্দেশ্য হলো understandable, reusable, easier to extend এবং well-maintainable সফটওয়্যার ডিজাইন করা। আমরা প্রতিনিয়ত এক্সিস্টিং প্রজেক্টে কাজ করি। আর তা করতে গিয়ে প্রচুর কোড পড়তে হয়, পড়ে বুঝতে হয় এবং নতুন রিকোয়ারমেন্ট অনুযায়ী এক্সটেন্ড করতে হয়। যদি SOLID এর মতো একটা কমন প্যাটার্ন ফলো করে কোড করা হয় তাহলে পরবর্তীতে এসব কোড সহজে বুঝা যাবে এবং দ্রুত এক্সটেন্ড করা যাবে।

SOLID এর ৫টি principle হচ্ছেঃ

- Single Responsibility Principle (SRP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

এই আর্টিকেলে আমি SOLID এর প্রতিটি principle নিয়ে আলোচনা করবো। কিভাবে আমরা প্রতিটি principle কে violate করি এবং কিভাবে তা সমাধান করা যায় সেটা দেখানোর চেষ্টা করবো। সবগুলো উদাহরণ PHP তে দেয়া আছে, তবে যেকোনো OOP ল্যাংগুয়েজ এ ব্যবহার করা যাবে। এই আর্টিকেল এ SOLID এর প্রথম দুইটি principle নিয়ে আলোচনা করবো। বাকি ৩ টি নিয়ে পরবর্তী পর্বে লিখবো ইনশাআল্লাহ। চলুন শুরু করা যাকঃ

Single Responsibility Principle (SRP)

> A class should have one, only one reason to change.

SOLID এর প্রথম principle হলো Single Responsibility Principle (SRP). এটা খুবই সহজ কনসেপ্ট। SRP এর শর্ত হলো একটা Class এর শুধুমাত্র একটাই কাজ/উদ্দেশ্য থাকবে। অর্থাৎ প্রতিটি ভিন্ন-ভিন্ন কাজের জন্য ভিন্ন-ভিন্ন Class থাকবে।

চলুন দেখা যাক কিভাবে SRP violate হচ্ছেঃ

class userInfo

```
{
  public function getUserName()
  {
    return 'name';
  }

  // Here SRP violates
  public function sendMailToUser()
  {
    return 'success';
  }
}
```

উপরের UserInfo ক্লাসে দুটি মেথড আছে। একটা getUserName যেটা ইউজার এর নাম রিটার্ন করে, অন্যটা sendMailToUser যেটা দিয়ে ইউজারকে ইমেইল পাঠানো হয়। তারমানে এই ক্লাস এর জব দুইটা, ১) ইউজার এর নাম বা তথ্য দেয়া এবং ২) ইমেইল পাঠানো।

আর একের অধিক কাজ/উদ্দেশ্যে থাকলেই SRP violate হচ্ছে। কারণ SRP বলে যে, একটা ক্লাস এর একটার বেশি দায়িত্ব থাকবেনা, ফলে পরবর্তীতে ওই ক্লাসে চেঞ্জ আসবে শুধুমাত্র ওই একটা জব/উদ্দেশ্যের কারণেই। এক্ষেত্রে আমরা ইমেইল পাঠানোর জন্য আলাদা ক্লাস ডিজাইন করবো।

এখানে একটা বিষয় খেয়াল রাখতে হবে যে, রিয়েল-লাইফে কাজ করতে গিয়ে যদি এভাবে প্রতিটা স্পেসিফিক কাজের জন্য আলাদা আলাদা ক্লাস ডিজাইন করি তাহলে পরে এতো ক্লাস মেইনটেইন করা কষ্টকর হবে এবং এটা মোটেও ভালো প্রাকটিস না। SRP বলেনা যে একটা ক্লাস এর শুধুমাত্র একটাই মেথড থাকবে। একটা ক্লাসে আমরা যত ইচ্ছা মেথড ও প্রপার্টি রাখিনা কেন, সবকিছুই যাতে একই উদ্দেশ্য নিয়ে কাজ করে।

চলুন আমরা UserInfo ক্লাসে SRP refactor করিঃ

```
class userInfo
{
  public function getUserName()
  {
    return 'name';
  }
```

```
  public function getUserAddress()
  {
    return 'address';
  }

  public function getUserRole()
  {
    return 'role';
  }
}
```

এখানে UserInfo ক্লাস এর ৩টি মেথড আছে। ৩টি মেথড-ই ইউজার এর তথ্য সরবরাহের কাজ করছে। অর্থ্যাৎ এই ক্লাস এর একটাই কাজ- ইউজার এর তথ্য সরবরাহ করা।

Open-Closed Principle (OCP)

> You should be able to extend a classes behavior, without modifying it.

Robert C. Martin এই principle কে maintainable এবং reusable কোড এর ফাউন্ডেশন বলে উল্লেখ করেছেন।

OCP বলে যে, আমাদের সফটওয়্যার এর এনটিটি গুলো (class, function, module) এমনভাবে ডিজাইন করা উচিত যাতে ভবিষ্যতে নতুন রিকোয়ারমেন্ট এলেও এক্সিস্টিং কোড মোডিফাই না করে বরং এক্সিস্টিং কোডকে এক্সটেন্ড করেই এর behavior কে পরিবর্তন করা যায়। এখানে উল্লেখ্য, behavior বলতে সোর্স কোডের আচরণগত পরিবর্তনকেই বুঝায়। সোজা কোথায়, আমাদের সফ্টওয়ার এনটিটি গুলো হবে open for extension but closed for modification.

আমরা যদি যথাযথভাবে OCP প্রয়োগ করতে পারি তবে existing সোর্স কোড মোডিফাই না করেই নতুন ফিচার এড করা যাবে। OCP কনফার্ম করে যে, আমাদের ডিজাইন করা এনটিটি সমূহ ভবিষ্যতের রিকোয়ারমেন্ট এর চাহিদা পূরণে নতুন ও ভিন্ন উপায়ে আচরণ করতে পারবে। এবং সেটা এক্সিস্টিং সোর্স কোডের বড় কোনো পরিবর্তন ছাড়াই।

চলুন নিচের উদাহরণটি দেখিঃ

```
class PaymentService
{
    public function payWithPaypal()
    {
        // pay with paypal
    }

    public function payWithCreditCard()
    {
        // pay with credit card
    }

    public function payWithWireTransfer()
    {
        // pay with wire transfer
    }
}

class PaymentController
{
    public function pay(Request $request, PaymentService $paymentService)
    {
        $payment_type = $request->input('payment_type');

        // Here OCP violates !!!
        switch ($payment_type) {
            case 'Paypal':
                $response = $paymentService->payWithPaypal();
                break;
            case 'Credit Card':
                $response = $paymentService->payWithCreditCard();
                break;
```

```
        default:
            $response = $paymentService->payWithWireTransfer();
        }
        return $response;
    }
}
```

ধরা যাক, আমাদের সিস্টেমে পেমেন্টের জন্য একটি paymentController ক্লাস আছে, এখানে pay মেথড এর মাধ্যমে পেমেন্ট নেয়া হয়। আর paymentService নামক ক্লাস আছে এখানে বিভিন্ন ধরণের পেমেন্ট সার্ভিস এর জন্য আলাদা আলাদা function রয়েছে। pay মেথড এর পেমেন্ট টাইপ এর উপর ভিত্তি করে পেমেন্ট সার্ভিস এর ধরণ সিলেক্ট করা হয়।

এখন এখানে যদি নতুন কোনো পেমেন্ট গেটওয়ে যোগ করতে চাই তাহলে বিদ্যমান দুইটা ক্লাসই মোডিফাই করতে হবে। এখানেই OCP violation হয়েছে। যেহেতু OCP বলেছে বিদ্যমান সোর্স কোড হবে closed for modification, সেখানে আমাদের কোড মোডিফাই করতে হচ্ছে।

চলুন দেখি কিভাবে OCP violation দূর করতে পারিঃ

```
interface paymentInterface
{
    public function pay();
}

class payWithPaypal implements paymentInterface
{
    public function pay()
    {
        // TODO: Implement pay() method.
    }
}

class payWithCreditCard implements paymentInterface
{
    public function pay()
```

```
  {
    // TODO: Implement pay() method.
  }
}

class payWithWireTransfer implements paymentInterface
{
  public function pay()
  {
    // TODO: Implement pay() method.
  }
}

class PaymentService
{
  public function initialize($payment_type)
  {
    switch ($payment_type) {
      case 'Paypal':
        return new payWithPaypal();
        break;
      case 'Credit Card':
        return new payWithCreditCard();
        break;
      default:
        return new payWithWireTransfer();
    }
  }
}

class PaymentController
{
```

```
    public function pay(Request $request, PaymentService $paymentService)
    {
        $payment_type = $request->input('payment_type');
        // OCP refactor
        $payment = $paymentService->initialize('Paypal');
        $payment->pay();
    }
}
```

এখানে আমরা একটা interface করেছি paymentInterface নামে, যেখানে শুধুমাত্র pay মেথড আছে। যেসকল class এই interface কে ইমপ্লিমেন্ট করবে তাদেরকে অবশ্যই pay মেথডটি ডিফাইন করতে হবে।

এরপর আমরা বিভিন্ন ধরণের পেমেন্ট গেটওয়ের জন্য আলাদা-আলাদা class করেছি যারা সবাই paymentInterface কে ইমপ্লিমেন্ট করেছে। আর paymentService ক্লাসে initialize মেথড এর মাধ্যমে পেমেন্ট গেটওয়ে নির্বাচন করছি এবং paymentController ক্লাস এর pay মেথডও পরিবর্তন করেছি, এখানে পেমেন্ট টাইপ দিয়ে paymentService থেকে নির্দিষ্ট পেমেন্ট গেটওয়ে সিলেক্ট করা হয়।

যেহেতু সকল পেমেন্ট গেটওয়ের ক্লাসই paymentInterface কে ইমপ্লিমেন্ট করছে সেহেতু তারা অবশ্যই pay মেথড ও ডিফাইন করেছে। তারপর ওই মেথডে কল করে পেমেন্ট সম্পন্ন হবে।

এখন যদি আমাদেরকে নতুন পেমেন্ট গেটওয়ে এড করতে বলা হয় তাহলে খুব সহজেই সেটা করা যাবে। যা করতে হবেঃ

নতুন পেমেন্ট গেটওয়ের জন্য আলাদা ক্লাস হবে যেটা paymentInterface কে ইমপ্লিমেন্ট করবে।

paymentService ক্লাস এর initalize মেথডে নতুন পেমেন্ট গেটওয়ের জন্য কন্ডিশন এড হবে এবং নতুন ক্লাস এর অবজেক্ট রিটার্ন হবে।

অর্থাৎ, existing কোডে তেমন পরিবর্তন ছাড়াই আমরা নতুন পেমেন্ট গেটওয়ে এড করতে পেরেছি।

কিভাবে বুঝবো কোথায় OCP apply করতে হবেঃ

OCP'র মূল উদ্দেশ্য হলো কোনো class এর behavior কে বড় ধরনের মোডিফিকেশন ছাড়াই এক্সটেন্ড করতে পারা। তারমানে এই নয় যে আমরা সব ক্লাসে OCP এপ্লাই করবো। সিস্টেম এর যেসব behavior প্রতিনিয়ত পরিবর্তনের আশংকা আছে সেসব ক্ষেত্রেই OCP এপ্লাই করে ভালো ফল পাওয়া যায়। সিস্টেম এর ডোমেইন সম্পর্কে জানা থাকলে বুঝতে সহজ হয় যে কোথায় OCP ইমপ্লিমেন্ট করা প্রয়োজন।

Liskov Substitution Principle (LSP)

Objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program.

Subclass/derived class should be substitutable for their base class.

বারবারা লিসকভের (Barbara Liskov) নামানুসারে এর নামকরণ করা হয়েছে। তিনি 1987 সালে এটি উপস্থাপন করেছিলেন। এখানে Substitution বলতে বুঝায়- প্রতিস্থাপন /কোনো কিছুর সাথে রিপ্লেস করা।

Liskov Substitution Principle বলে যে, কোনো object এর ইনস্ট্যান্স কে ওই object এর base/parent class এর ইনস্ট্যান্স দিয়ে রিপ্লেস করা যাবে এবং এজন্য existing code এর correctness বা বিশুদ্ধতা পরিবর্তন করতে হবেনা।

নিচের উদাহরণটি দেখুনঃ

```
abstract class PaymentStatusService
{
  public abstract function getStatus($payment_id);
}

class CreditCardPaymentStatus extends PaymentStatusService
{
  public function getStatus($payment_id)
  {
    return 'success';
  }
}

class SonaliPaymentStatus extends PaymentStatusService
{
  public function getStatus($payment_id)
```

```
  {
    return ['status' => 'success'];
  }
}

$payment_status = new CreditCardPaymentStatus();
$payment_status->getStatus(1);

$payment_status = new SonaliPaymentStatus();
$payment_status->getStatus(1);
// Here the LSP violates !!!
```

এই কোডে দেখা যাচ্ছে, পেমেন্ট স্টেটাস জানার জন্য একটা abstract class আছে paymentStatusService নামে এবং এই ক্লাসে getStatus নাম একটা abstract মেথড আছে। বিভিন্ন পেমেন্ট গেটওয়ের জন্য আলাদা আলাদা class করা হয়েছে যারা paymentStatusService কে extend করে। যেহেতু abstract ক্লাসকে এক্সটেন্ড করলে তার abstract মেথডকে ইমপ্লিমেন্ট করতে হবে, সেহেতু সকল ক্লাস getStatus মেথড ডিফাইন করেছে। সবশেষে আমরা CreditCardPaymentStatus class এর instance তৈরী করেছি এবং getStatus মেথড কল করেছি credit card পেমেন্ট এর স্টেটাস জানার জন্য।

আর LSP বলতে চায়, আমরা CreditCardPaymentStatus class এর instance তৈরী করেছি, সেটা যেনো SonaliPaymentStatus এর instance দিয়ে রিপ্লেস করা যায়। কারণ যেহেতু CreditCardPaymentStatus এবং SonaliPaymentStatus দুইটি ক্লাসই একই base/parent class এর child.

কিন্তু আমাদের কোডে দেখা যাচ্ছে, আমরা এই রিপ্লেসমেন্ট করতে পারবোনা। কারণ, CreditCardPaymentStatus এর getStatus মেথড এর রিটার্ন টাইপ আর SonaliPaymentStatus এর getStatus মেথড এর রিটার্ন টাইপ এক নয়। যেহেতু এই রিটার্ন টাইপ নিয়ে পরবর্তীতে কোনো প্রসেসিং থাকতে পারে, সেহেতু এখানে শুধুমাত্র object এর instance রিপ্লেস করলেই হবেনা। এটাই LSP violation.

উল্লেখ্য, এখানে আমরা base ক্লাস দিয়ে উদাহরণ দিয়েছি, তবে interface এর implementation এর ক্ষেত্রেও একই বিষয় প্রযোজ্য। একটা interface কে ইমপ্লিমেন্ট করা সকল ক্লাস ও পস্পরকে প্রতিস্থাপনযোগ্য হতে হবে।

নিম্নোক্ত শর্তগুলো অনুসরণ করলে আমরা LSP violation এড়িয়ে চলতে পারবোঃ

sub-class গুলোতে base-class এর মেথড ব্যবহার করলে সেক্ষেত্রে মেথড এর input parameter type এবং সংখ্যা অবশ্যই base-class এর মেথড এর সমান হতে হবে।

মেথড এর রিটার্ন টাইপ একই ধরণের হতে হবে।

Exception টাইপ ও একই ধরণের হতে হবে।

Interface-Segregation Principle (ISP)

A client should not be forced to implement an interface that it doesn't use.

এটা খুবই সহজ একটা principle. এর মানে হচ্ছে যে, কোনো class কে এমন কোনো interface ইমপ্লিমেন্ট করতে বাধ্য না করা, যে interface সে ব্যবহার করেনা অথবা যে interface এর মেথডগুলো সে ব্যবহার করেনা।

ISP বলতে চায় যে, আমাদের তৈরিকৃত interface হবে একদম client-specific এবং খুব বেশি generalized না। অর্থাৎ, interface হবে ছোটো এবং client-specific.

একটা উদাহরণ দেখা যাকঃ

```
interface PrinterInterface
{
   public function print();

   public function photocopy();

   public function scan();
}

class DigitalPrinter implements PrinterInterface
{
   public function print()
   {
      return 'Print';
   }
```

```
    public function photocopy()
    {
        return 'Photocopy';
    }

    public function scan()
    {
        return 'Scan';
    }
}

class ModernPrinter implements PrinterInterface
{
    public function print()
    {
        return 'Print';
    }

    public function photocopy()
    {
        return 'Photocopy';
    }

    // ISP Violates here
    public function scan()
    {
        return 'Not supported';
    }
}

class OldPrinter implements PrinterInterface
{
```

```
    public function print()
    {
        return 'print';
    }

    // ISP Violates here
    public function photocopy()
    {
        return 'Not supported';
    }

    // ISP Violates here
    public function scan()
    {
        return 'Not supported';
    }
}
```

এখানে একটা PrinterInterface তৈরী করেছি যেটায় ৩টি মেথড আছে, print, photocopy, এবং scan. তারপর ৩টি printer এর জন্য ৩টি class ডিজাইন করা হয়েছে যারা সবাই PrinterInterface কে ইমপ্লিমেন্ট করেছে। যেহেতু interface ইমপ্লিমেন্ট করলে তার সবগুলো মেথড ইমপ্লিমেন্ট করতে হয়, সেহেতু printer ক্লাসগুলোও সবগুলো মেথড ইমপ্লিমেন্ট করেছে।

এখানে কোথায় ISP violation হয়েছে? দেখুন, আমাদের একটা printer আছে OldPrinter যেটাতে scan এবং photocopy এর ফিচার নেই কিন্তু তবুও OldPrinter class এই মেথডগুলো ইমপ্লিমেন্ট করতে বাধ্য। তেমনিভাবে, ModernPrinter ও বাধ্য হয়েছে scan মেথড ইমপ্লিমেন্ট করতে। যদিও এইসব মেথড ওদের প্রয়োজন নেই, কিন্তু interface এর rules এর কারণে তারা বাধ্য হয়েছে। এটাই Interface-Segregation Principle কে violate করেছে।

কিভাবে এর সমাধান করা যায়ঃ

```
interface PrinterInterface
{
    public function print();
}
```

```
interface PhotocopyInterface
{
   public function photocopy();
}

interface ScannerInterface
{
   public function scan();
}

// ISP refactors here
class DigitalPrinter implements PrinterInterface, PhotocopyInterface, ScannerInterface
{
   public function print()
   {
      return 'Print';
   }

   public function photocopy()
   {
      return 'Photocopy';
   }

   public function scan()
   {
      return 'Scan';
   }
}

// ISP refactors here
class ModernPrinter implements PrinterInterface, PhotocopyInterface
```

```
{
    public function print()
    {
        return 'Print';
    }

    public function photocopy()
    {
        return 'Photocopy';
    }
}

// ISP refactors here
class OldPrinter implements PrinterInterface
{
    public function print()
    {
        return 'print';
    }
}
```

এখানে আমরা আগের interface কে segregate করে small & client-specific interface হিসেবে ৩ ভাগে ভাগ করেছি। এর ফলে যে client এর যে interface প্রয়োজন সে তাকেই ইমপ্লিমেন্ট করবে। এবং কোনো client কেই অপ্রয়োজনীয় মেথড ইউজ করতে বাধ্য করা হবে না।

Dependency Inversion Principle (DIP)

> High level modules should not depend on low-level modules, both should depend on abstractions.
>
> Abstract should not depend on details. Details should depend on abstractions.

সহজে বলা যায় যে, সিস্টেম এর high-level code, low-level code এর উপর ডিপেন্ড করবে না। বরং দুইটাই ডিপেন্ড করবে abstraction এর উপর।

এখানে কিছু টার্মস বুঝার চেষ্টা করিঃ

Dependency: Class A যদি class B এর উপর বা কোনো ভ্যালুর উপর নির্ভর করে তবে class B অথবা ওই ভ্যালুই হলো class A এর ডিপেন্ডেন্সি।

High-level code, Low-level code: High-level code হচ্ছে সেসব যেগুলো আমরা সরাসরি কল করি এবং যেগুলো core business logic ধারণ করে। আর এসব High-level code কে যেসব code সাহায্য করে সেগুলো Low-level code।

Abstraction: Abstraction হলো কোনো concrete class অথবা method এর indirect representation. Abstraction বলতে সাধারণত Abstraction class এবং interface কে বুঝায়।

একটা উদাহরণ দেখা যাকঃ

```
class GoogleAuthenticationService
{
  public function authenticate($email)
  {
    return 'true';
  }
}

class userLogin
{
  public function login($email)
  {
    // DIP violates here
    $google_authentication = new GoogleAuthenticationService();
    $auth_result = $google_authentication->authenticate($email);

    if ($auth_result) {
```

```
        return true;
      }
    }
}

$login = new userLogin();
$login->login('samadocpl@gmail.com');
```

এখানে আমরা login এর জন্য userLogin ক্লাস ডিজাইন করেছি এবং সেখানে login মেথড এর মাধ্যমে GoogleAuthenticationService দিয়ে ইউজারকে authenticate করছি। এক্ষেত্রে আমাদের high-level code হলো userLogin ক্লাস এবং low-level code হলো GoogleAuthenticationService এবং high-level code, low-level code এর উপর ডিপেন্ডেন্ট হয়ে আছে। এটাই DIP violation.

কিছুদিন পর যদি আমাদের অন্যকোনো athentication সার্ভিস প্রয়োজন হয় তাহলে কি করতে হবে? তখন আমরা আলাদা athentication সার্ভিস তৈরী করে userLogin ক্লাস এর login মেথডে সেই সার্ভিস ইউজ করবো, আর এটা কিন্তু SOLID এর OCP কেও violate করছে।

চলুন দেখি কিভাবে এ সমস্যার সমাধান করতে পারিঃ

```
interface AuthenticationService
{
  public function authenticate($email);
}

class GoogleAuthenticationService implements AuthenticationService
{
  public function authenticate($email)
  {
    return 'true';
  }
}

class GithubAuthenticationService implements AuthenticationService
```

```
{
   public function authenticate($email)
   {
      return 'true';
   }
}

class userLogin
{
   public function login($email, AuthenticationService $authenticationService)
   {
      // DIP
      // Dependency inverted on abstraction
      $auth_result = $authenticationService->authenticate($email);
      if ($auth_result) {
         return true;
      }
   }
}

$login = new userLogin();
$login->login('samadocpl@gmail.com', new GithubAuthenticationService());
```

যেহেতু DIP এর শর্ত হলো, high-level code এবং low-level code দুইটাই Abstraction এর উপর নির্ভর করবে, সেহেতু এখানে আমাদেরকে Abstraction (class, interface) ইমপ্লিমেন্ট করতে হবে। আমরা interface ব্যবহার করেছি।AuthenticationService নামে একটি ইন্টারফেস তৈরী করেছি যেখানে authenticate মেথড আছে। প্রতিটা login সার্ভিস এই ইন্টারফেস কে ইমপ্লিমেন্ট করবে এবং authenticate মেথডে তাদের functionality ডিফাইন করবে। তারমানে, আমাদের low-level code এখন Abstraction এর উপর ডিপেন্ডেন্ট হলো।

তারপর, আমাদের userLogin ক্লাস এর login মেথডে কিছু পরিবর্তন করেছি। এখানে AuthenticationService ইন্টারফেস কে ইঞ্জেক্ট করেছি। অর্থাৎ এই মেথডে আসার সময়েই প্যারামিটার হিসেবে AuthenticationService interface কে ইমপ্লিমেন্ট করেছে এমন কোনো class এর instance আসবে আর সেই instance এ authenticate মেথড ডিফাইন করা থাকবে।

তারমানে দেখা যাচ্ছে, আমাদের high-level code অর্থাৎ userLogin ক্লাস এখন কোনো concrete ক্লাস এর উপর নয় বরং একটা Abstraction (AuthenticationService) এর উপর ডিপেন্ডেন্ট।

নোটঃ

SOLID নিয়ে কাজ করার সময় আমাদের মাথায় রাখতে হবে যে, সলিড শুধুমাত্র কিছু practice, এগুলো কোনো আবশ্যক নিয়ম নয়। সবসময় বা সবজায়গায় সলিড apply করতে আমরা বাধ্য না। অনেক সময় দুই-একটা principle apply করলে আরো দুই-একটা automatic apply হয়ে যায়। SOLID এর Principle গুলো পরস্পরের সাথে সম্পর্কযুক্ত।

SOLID এর মূল উদ্দেশ্য হলো reusable এবং maintainableসফটওয়্যার ডিজাইন সেহেতু যেসব জায়গায় SOLID implement করলে আমাদের উদ্দেশ্য সফল হবে সেসব জায়গায় SOLID implement করা উচিত।