SOLID Principles

أتراب أحمد الدبعي

➢ SOLID principles

هي مجموعة من المبادئ التوجيهية في تطوير البرمجيات التي تهدف لجعل الكود أسهل في الكتابة والصيانة والتوسع وإعادة الاستخدام، وتساعد على عمل في مشاريع كبيرة بدون بذل جهد كبير في عملية إصلاح الأخطاء، وهذه المبادئ مفيدة بصورة خاصّة في التصميم كائنيّ التوجّه-object oriented design، حيث تقدم هذه القواعد الأساسيات التي يجب إتباعها عند إنشاء الأصناف وربطها معًا.

هذه المبادئ ليست ضرورية لتصميم البرمجيات ولكنها تساعد مطوّري البرمجيّات على تحقيق تصاميم برمجيّة عالية الجودة.

تتكون SOLID من خمسة مبادئ رئيسية:

1- مبدأ المسؤولية الفردية (Single Responsibility Principle - SSP):

هو أحد مبادئ تصميم البرمجيات والذي ينص على أن كل فئة أو وحدة برمجية يجب أن تكون لها مسؤولية واحدة فقط.

وبمعنى آخر، يجب أن تتعامل كل فئة برمجية أو وحدة برمجية مع مسؤولية واحدة فقط، ولا يجب أن تكون متعددة الوظائف، هذا المبدأ يهدف إلى تحقيق التفصيل والتخصص في البرامج، حيث يتيح للفئات البرمجية أن تكون متسقة وسهلة الفهم والصيانة.

عند اتباع مبدأ المسؤولية الفردية، يتعين على المطورين تقسيم البرامج إلى عدة فئات صغيرة، حيث كل فئة لديها دور محدد ووظائف محدودة.

على سبيل المثال: إذا كان لدينا نظام لإدارة المستخدمين، فيجب أن يكون لدينا فئة لإضافة المستخدمين، وفئة لتعديل المستخدمين، وفئة لحذف المستخدمين، وهكذا. بالتالي، يمكن أن تتعامل كل فئة برمجية مع مسؤولية واحدة فقط.

تطبيق مبدأ المسؤولية الفردية على البرامج يوفر العديد من الفوائد، ومنها:

1. **تحسين جودة البرامج:** عندما يتحمل المطورون والمستخدمون مسؤولية تصميم وتطوير البرامج بشكل فردي، فإنهم يعملون بجهود كبيرة لضمان جودة البرامج، هذا يؤدي إلى إصدار برامج أكثر استقرارًا وأقل أخطاء.

2. **زيادة سرعة التطوير:** عندما يكون لكل فرد مسؤولية تطوير جزء من البرنامج، يتسنى للفريق تقسيم المهام والعمل بشكل مستقل، هذا يزيد من سرعة التطوير ويسهل إضافة المزيد من الميزات والتحسينات.

3. **تحسين مستوى الأمان:** عندما يكون لكل فرد مسؤولية تطوير جزء من البرنامج، فإنه يزيد من اهتمامه بالأمان وحماية البيانات، يتعاون الجميع للكشف عن ثغرات الأمان وإصلاحها بشكل سريع.

4. **توفير التخصص:** عندما يتحمل كل فرد مسؤولية جزء محدد من البرامج، فإنه يسمح للأفراد بالتخصص في هذا المجال، هذا يؤدي إلى زيادة خبرات الفريق وتطور قدراته في مجالات مختلفة.

2- مبدأ الفتح/الإغلاق (Open/Closed Principle - OCP):

هو أحد مبادئ تصميم البرمجيات التي تهدف إلى جعل الكود قابلًا للتوسع والتعديل دون الحاجة إلى تعديله. يقول هذا المبدأ: "الكود يجب أن يكون مفتوحًا للامتداد ومغلقًا للتعديل".
بشكل عام، يشير المبدأ إلى أنه عندما نرغب في إضافة ميزات جديدة أو تعديلات على البرمجيات الموجودة، يجب أن تكون قادراً على القيام بذلك من خلال إضافة كود جديد دون الحاجة لتعديل الكود الموجود بالفعل.
هذا التصور يساعد على تقليل خطر حدوث أخطاء غير متوقعة أو تأثيرات جانبية ويساهم في جعل البرامج أكثر استقرارًا وسهولة في الصيانة، حيث أن التغييرات والإضافات الجديدة لن تؤثر على كود البرامج المستقر.

- هناك طرق عديدة لتحقيق مبدأ الفتح/الإغلاق، ومنها:

1. **استخدام الاستدعاءات الافتراضية (Virtual Calls):** يمكننا تعريف واجهة أو فئة أساسية تحتوي على السلوك المشترك بين المكونات المختلفة، ثم يمكن للفئات المشتقة تعديل هذا السلوك حسب احتياجاتها الخاصة.

2. **استخدام التركيب (Composition):** يمكننا استخدام التركيب بدلاً من التوريث لإضافة سلوك جديد إلى كائن معين. على سبيل المثال، يمكن إضافة كائن "Logger" كعضو في كائن "Service" لتسجيل الأحداث دون تغيير كود "Service" نفسه.

3. **استخدام التصميم بالأشجار (Tree-like Design):** يمكن تصميم البرامج بطريقة تشبه الأشجار، حيث يتم تجزئة المشروع إلى موديلات وعارضات ومحولات وغيرها من المكوّنات. هذا يسمح بإضافة مكونات جديدة دون التأثير على المكونات

3- مبدأ استبدال ليسكوف (Liskov Substitution Principle - LSP):

هو أحد مبادئ تصميم البرمجة الكائنية وهو يعتبر جزءًا من مجموعة المبادئ المعروفة بـ SOLID، وينص على أن يجب أن يكون بإمكان استبدال أي كائن من نوع معين بأي كائن من نوع فرعي له دون أن يتعطل سير البرنامج.

بمعنى آخر، إذا كان لدينا تصنيف A (Class) ولديه تصنيف B (Class) فإن الكائنات التابعة لـ B يجب أن تكون قادرة على استبدال الكائنات التابعة لـ A دون أن يحدث أي تأثير سلبي على سير البرنامج.

هذا المبدأ يساعد على بناء هرمية قوية للتصنيفات ويضمن تعددية الاستخدام (Polymorphism) في البرمجة الكائنية. وبفضل هذا المبدأ، يمكن استخدام كائنات فرعية لتعوض أو توسع وظائف الكائنات الأساسية دون التأثير على كود البرنامج القائم.

لتحقيق مبدأ استبدال ليسكوف، يجب مراعاة النقاط التالية:

1. يجب أن تحتوي الفئات الفرعية على نفس السلوك والوظائف الموجودة في الكلاس الأساسي. يعني هذا أنه يجب أن تكون هناك استجابة واحدة لنفس الطلبات (methods) في الكلاس الأساسي والكلاس الفرعي.

2. يمكن للكائنات الفرعية أن توسع وتضيف وظائف إضافية للكائنات الأساسية، ولكن لا يجب أن تقلل من سلوكها أو تغيره.
3. يجب أن تحترم الكائنات الفرعية جميع الشروط والقيود المفروضة على الكائنات الأساسية. وهذا يعني أنه يجب على الكائنات الفرعية أن تستخدم نفس المدخلات والمخرجات المتوقعة والتعامل بنفس الطريقة مع الحالات الاستثنائية.

مثال:
لدينا فئتيْ "Shape" (الشكل) و "Rectangle" (المستطيل). الفئة "Shape" تحتوي على طرق لحساب مساحة الشكل وطول المحيط. بينما الفئة "Rectangle" تمتد من الفئة "Shape" وتضيف خاصية إضافية وهي طول العرض.
إذا قمنا بتطبيق مبدأ استبدال ليسكوف، يجب أن يكون بإمكاننا استخدام كائن من الفئة "Rectangle" في أي مكان نستخدم فيه كائن من الفئة "Shape". على سبيل المثال، إذا كان لدينا دالة تقوم بحساب مجموع مساحات مجموعة من الأشكال، يجب أن يعمل هذا التطبيق على حساب مساحات المستطيلات أيضًا.

باختصار، مبدأ استبدال ليسكوف يهدف إلى تعزيز التفاعلية بين الكائنات المختلفة وتمكين استخدام الكائنات الفرعية بديلاً للكائنات الأساسية في أي سياق برمجي. يساعد هذا المبدأ على تحقيق مرونة البرمجة وسهولة إضافة وتعديل المزيد من الوظائف في المستقبل دون التأثير على الكود القائم.

4- مبدأ فصل الواجهات (Interface Segregation Principle - ISP):

هو أحد مبادئ تصميم البرمجيات التي تهدف إلى تعزيز فصل واجهات Classes في النظام. يقترح هذا المبدأ أنه يجب على Classes أن تعرض واجهات مختلفة للعملاء بدلاً من واجهة واحدة ضخمة.

الفكرة الأساسية وراء مبدأ تجزئة الواجهة هي أن Classes لا يجب أن تضطر إلى تنفيذ وظائف غير مستخدمة من قبلها. بالتالي، ينصح بتقسيم واجهات Classes إلى عدة واجهات صغيرة وخاصة بكل عميل.

عند اتباع مبدأ تجزئة الواجهة، يتم تحقيق عدة فوائد:

1. **فصل المسؤوليات:** يسمح للكلاس بتركيز جهود التطوير على المسؤولية التي يقوم بها فقط، دون التورط في مسؤوليات أخرى غير ضرورية.
2. **تقليل التبعية:** يقلل من التبعية بين Classes، حيث يمكن لكل كلاس أن يعتمد فقط على الواجهات التي تحتاجها، بدلاً من الاعتماد على واجهة واحدة ضخمة.
3. **تعزيز إعادة الاستخدام:** يسهل إعادة استخدام الواجهات المستقلة في سياقات مختلفة، حيث يمكن استخدام كل واجهة بشكل منفصل دون الحاجة إلى تغيير كود الكلاس.
4. **تسهيل اختبار الوحدات:** يسمح بإنشاء اختبارات وحدية لكل واجهة على حدة، مما يسهل عملية اختبار وتصحيح الأخطاء.
5. **تعزيز المرونة:** يجعل من السهل إضافة وإزالة الميزات في المستقبل، حيث لن يؤثر ذلك على باقي أجزاء النظام.

5- مبدأ عكس التابعيّة (Dependency Inversion Principle - DIP):

هو أحد مبادئ تصميم البرمجة الكائنية وهو يعتبر جزءًا من مبادئ SOLID. يهدف هذا المبدأ إلى تقليل ارتباط الكود بين الطبقات والمكونات المختلفة في التطبيق، وذلك من خلال تحويل اعتمادية الطبقات على بعضها البعض.

يتم تحقيق هذا المبدأ من خلال ثلاث أساسيات:

1- High-level modules should not depend on low-level modules :

يعني أن الطبقات ذات المستوى العالي لا يجب أن تعتمد على الطبقات ذات المستوى المنخفض. بدلاً من ذلك، يجب أن يكون هناك اعتمادية على واجهة مشتركة بينهما.

2- Abstractions should not depend on details:

يشير إلى أن التفاصيل التقنية للطبقة لا يجب أن تؤثر على واجهة البرمجة (API) لهذه الطبقة. بدلاً من ذلك، يجب استخدام واجهة عامة ومستقلة عن التفاصيل التقنية.

3- Depend on abstractions, not on concretions:

يعني أن الكود يجب أن يعتمد على الواجهات (interfaces) بدلاً من الكائنات المحددة (concrete objects). هذا يسمح بتبديل التطبيقات المختلفة للطبقات دون تعديل الكود.

مثال:

لنفترض أن لدينا تطبيقًا يحتوي على طبقتين، طبقة Service وطبقة Repository. في حالة عدم استخدام مبدأ Dependency Inversion، ستعتمد طبقة Service مباشرةً على طبقة Repository وستستخدم كائنًا محددًا منها. هذا يؤدي إلى ارتباط قوي بين الطبقتين وصعوبة استبدال طبقة Repository بأخرى جديدة.

ومع تطبيق مبدأ Dependency Inversion، ستكون هناك واجهة (interface) مشتركة بين الطبقتين، حيث ستعرف طبقة Service وظائف الواجهة وتعتمد عليها بدلاً من الكائن المحدد.