

المبادئ السوية

لإنتاج

برامج قوية

خالد السعيداني

دليل مختصر ومتكامل لمبادئ

SOLID Design Principles in C#

﴿ يَا أَيُّهَا الَّذِينَ آمَنُوا اتَّقُوا اللَّهَ وَقُولُوا قَوْلًا سَدِيدًا

(70)

يُصْلِحْ لَكُمْ أَعْمَالَكُمْ وَيَغْفِرْ لَكُمْ ذُنُوبَكُمْ وَمَنْ يُطِيعِ اللَّهَ

وَرَسُولَهُ فَقَدْ فَازَ فَوْزًا عَظِيمًا (71) ﴾

سورة الأحزاب

همسة في أذنك قبل الانطلاق

هذا الكتاب الذي بين يديك الآن هو مجهود بشري قد تتخلله مجموعة من الأخطاء، أو أقول لك بصراحة: حتما ستكون فيه أخطاء نتيجة للسهو، النسيان، ضعف التركيز، أخطاء في الكتابة، أو نوع آخر من الأخطاء نحن لا نعلمه، أنت قد تعلمه أثناء سفرك مع هذا السفر! ولأن هذا السفر الصغير أعد خصيصاً لك، من أجل نفعك، فلا تتردد على صاحبه المغمور بالتماس الأعذار إن صادفت فيه أي نقص، ولو أردت أن تسدي إليه معروفاً فراسله على الإيميل التالي بتفاصيل الخطأ:

Khalid ESSAADANI Email:

Khalid_essaadani@hotmail.fr

وسيكون لك من الشاكرين!
وإن شعرت أن هذا الكتاب قد أضاف إلى رصيدك المعرفي شيئاً، فلا تبخل على صاحب الكتاب ووالديه وسائر المسلمين أجمعين بدعوة بظهر الغيب، رفع الله قدرك في الدارين.

همسة أخيرة قبل فتح باب الكتاب:
أسأل الله عز وجل أن يجعل هذا الكتاب خالصاً لوجهه الكريم، وأن ينفعك به نفعاً يختصر عليك الوقت والجهد.

دام لكم البشر والفرح!

خالد السعداني، في 21/أغسطس/2019

الفهرس

4	المقدمة
5	هل هذا الكتاب يناسبك؟
5	هل هذا الكتاب لا يناسبك؟
5	ماهي مبادئ التصميم Design Principles
6	ماهي نماذج التصميم Design Patterns
7	ماهو الفرق بين مبادئ ونماذج التصميم
7	لماذا خصصنا هذا الكتاب لمبادئ SOLID
8	دقيقة من فضلك، ماهي SOLID أصلا
8	مصطلحات ستعيش معها في مجال التصميم
8	ماهي المسؤولية Responsibility
9	ماهو الارتباط Coupling
12	ماهو التماسك Cohesion
13	ماهو التجريد Abstraction
14	مامعنى التفاصيل Details
15	ماهو الفرق بين التعديل والتوسيع Modification and Extension
15	مامعنى الانتهاك Violation
15	جلسة مع مشروعنا الجديد NumberConverter
17	المبدأ الأول: المسؤولية الواحدة Single Responsibility
17	لماذا نحتاج إلى مبدأ المسؤولية الواحدة SRP
17	متى نحتاج إلى مبدأ المسؤولية الواحدة SRP
18	كيف نطبق مبدأ المسؤولية الواحدة SRP
21	خلاصة مبدأ المسؤولية الواحدة SRP
22	المبدأ الثاني: مبدأ الفتح والإغلاق Open-Closed Principle
23	لماذا نحتاج إلى مبدأ الفتح والإغلاق OCP
23	متى نحتاج إلى مبدأ الفتح والإغلاق OCP
23	كيف نطبق مبدأ الفتح والإغلاق OCP
26	خلاصة مبدأ الفتح والإغلاق OCP

27	المبدأ الثالث: مبدأ الاستبدال Liskov Substitution Principle
27	نموذج Factory Design Pattern
28	مبدأ قلب التحكم Inversion of Control Principle
28	استعمال Reflection لإنشاء الأنواع
30	مشكلة الدائرة والشكل البيضاوي Circle-Ellipse Problem
33	الحل الأول لمشكلة Circle-Ellipse Problem
33	الحل الثاني لمشكلة Circle-Ellipse Problem
34	الحل الثالث لمشكلة Circle-Ellipse Problem
35	لماذا نحتاج إلى مبدأ الاستبدال LSP
36	متى نحتاج إلى مبدأ الاستبدال LSP
36	كيف نطبق مبدأ الاستبدال LSP
37	نموذج Null Object Design Pattern
39	خلاصة مبدأ الاستبدال LSP
40	المبدأ الرابع: مبدأ فصل الواجهات Interface Segregation Principle
40	لماذا نحتاج إلى مبدأ فصل الواجهات ISP
40	متى نحتاج إلى مبدأ فصل الواجهات ISP
40	كيف نطبق مبدأ فصل الواجهات ISP
44	خلاصة مبدأ فصل الواجهات ISP
45	المبدأ الخامس: مبدأ انعكاس التبعية Dependency Inversion Principle
46	لماذا نحتاج إلى مبدأ انعكاس التبعية DIP
46	متى نحتاج إلى مبدأ انعكاس التبعية DIP
47	كيف نطبق مبدأ انعكاس التبعية DIP
54	خلاصة مبدأ انعكاس التبعية DIP
55	خاتمة

المقدمة

تتأثر جودة البرمجيات بمستوى التصميم Design الذي تم اعتماده من قبل الفريق الذي يسهر على تطويرها، حيث يلعب تصميم البرمجيات دورا كبيرا في جعلها قابلة للصيانة والتحديث Maintainable، ومفتوحة على الزيادة والتوسيع Extensible، وقادرة على الخضوع للاختبارات Testable التي من شأنها التحقق من سلامة اشتغال مختلف أجزاء البرنامج وأدائها للمطلوب منها بالشكل المتوقع. لذلك ينبغي أن يولي كل مشغل في مجال صناعة البرمجيات أهمية كبيرة لعملية التصميم نظرا لتأثيرها المباشر على جودة المنتج المقدم.

ولا يمكن للتصميم أن يكون ناجحا إلا إذا كان يمنح المزايا الثلاثة التي تقدم ذكرها وهي كالآتي:

- القابلية للتحديث والصيانة: إذ يلزم أن يكون المشروع قابلا للتحديث في أي وقت دون أن يكون لهذه العملية تأثير سلبي على باقي أجزاء المشروع.
- القابلية للزيادة والتوسيع: كل مشروع لا يسمح بإضافة أجزاء جديدة عليه، أو توسيع أجزاء موجودة مسبقا، دون أن تتأثر باقي المكونات بشكل سلبي، فهو مشروع ناقص ومحدود.

- القابلية للاختبار: الاختبارات مهمة جدا لضمان سلامة اشتغال أجزاء المشروع، ولكي يكون المشروع قابلا للاختبار عليه أن يكون منضبطا لمجموعة من القواعد التابعة لمجال التصميم.

عملية تصميم البرمجيات تركز على مجموعة من المبادئ والنماذج Principles and Patterns التي وضعها أناس متضلعون في هذا المجال بغرض حل إشكاليات تعرضوا لها باستمرار، فقاموا بتقديم حلول جاهزة يمكن لأي مطور أن يستعملها في وضعيات مماثلة. أغلب هذه الحلول - التي تأتي على شكل مبادئ ونماذج - متوافقة بشكل كبير مع نمط البرمجة كائنية التوجه Object Oriented Programming Paradigm، حيث تستعمل هذه المبادئ والنماذج مفاهيم البرمجة الكائنية مثل الوراثة Inheritance وتعدد الأشكال Polymorphism و التجريد Abstraction وغيرها من أجل صياغة حلول متكاملة ومنظمة.

الهدف من وراء هذا الكتاب أن نعرف بمجال تصميم البرمجيات وماله من تأثير بالغ على نجاح البرمجيات، لأنني شخصا عانيت من عواقب تغييب ممارسات هذا المجال، فاضطرت في مرات عديدة إلى إعادة بعض المشاريع من الصفر لأنني لم أحترم فيها مبادئ ونماذج التصميم، الشيء الذي جعل هذه المشاريع منغلقة على ذاتها غير قابلة للتوسيع أو الصيانة، إضافة إلى صعوبة العودة إلى الكود وتفحصه وتتبعه بغرض تجويده وتحسينه أو بغرض إصلاحه وتصويبه فيما يعرف بعملية Refactoring، ناهيك عن إمكانية كتابة اختبارات على هذه الأكواد

وخصوصا الاختبارات الأحادية Unit Tests، فقد كان ذلك ضربا من الوهم لأن الكود الذي كنت أكتبه كان هو الوهم نفسه :). فتعلمت من الأخطاء السالفة إلزامية الإلمام بمجال التصميم في صناعة البرمجيات، وتكييف الوضعيات البرمجية مع مبادئ التصميم Design Principles، فأحببت أن أشارككم هذه المعلومات على هيئة كتاب. نسأل الله عز وجل أن يجعل هذا المجهود خالصا لوجهه الكريم، وألا يجعل فيه للنفس حظا، وأن ينفع به كل من له به حاجة.

هل هذا الكتاب يناسبك؟

إن كنت طالبا في مجال صناعة وتطوير البرمجيات، أو عاملا في هذا القطاع مهما كان مسماك (Senior, Junior, Architect, ...) فهذا الكتاب يناسبك، شريطة أن تكون على اطلاع شبه شامل وفهم شبه كامل لنمط البرمجة كائنية التوجه Object Oriented Programming، وذلك لارتباط مجال تصميم البرمجيات بمفاهيم هذا النمط، وأن تكون ملما بلغة C#.

هل هذا الكتاب لا يناسبك؟

هذا الكتاب قد لا يكون ملائما لك إذا لم يكن لك سابق عهد مع البرمجة كائنية التوجه، لأنك ستتعب ذهنك لسبب بسيط، وهو أن هذا الكتاب يقوم بالأساس على مفاهيم البرمجة كائنية التوجه، وبالتالي غيابها يؤدي بشكل مباشر إلى جعل المفاهيم التي سنتناولها غامضة لديك. لذلك عد إلى كتابنا الذي ألفناه قبل سنوات، والذي كنا قد أسميناه: الإكليل الجزء 2 - المختصر المفيد في البرمجة الكائنية التوجه OOP. بالرغم من أن الكتاب الذي ذكرت يتناول لغة الفيجوال بيزك دوت نيت إلا أنني أراه مناسبا لك، ولا تسألني كيف عرفت ذلك؟ لأنني أعرفك جيدا: أنت مشروع لمبرمج ناجح!

ماهي مبادئ التصميم Design Principles

مبادئ التصميم هي مجموعة من القواعد والتوجيهات التي توضح لنا كيف ينبغي أن يكون شكل وسلوك الكود الذي نقوم بكتابته، حيث تفرض علينا مجموعة من القيود التي بالانضباط لها سنكتب كودا نظيفا Clean Code، قابلا للصيانة Maintainable، قابلا للزيادة والتوسيع Extensible، وقابلا للاختبار Testable كالاختبارات الأحادية Unit Tests، والغرض من هذه المبادئ هو تمكين المطور من التخلص من العادات السلبية التي تدفعه إلى التركيز على

الوصول إلى الحل فقط دون الأخذ بعين الاعتبار جودة هذا الحل، ومدى قدرة هذا الحل على التكيف مع باقي أجزاء المشروع بحيث لا يؤدي تعديله إلى سلسلة من الكوارث، ومدى قابلية هذا الحل إلى التوسيع بحيث لو أردنا في المستقبل أن نضيف إليه أموراً جديدة أمكننا ذلك بكل يسر، والأهم من هذا وذلك مدى موافقة هذا الحل لمواصفات الاختبارات الأحادية ليخضع لها بغرض التأكد من أنه يؤدي المطلوب بالشكل المرغوب.

توجد العديد من مبادئ التصميم مثل مبدأ DRY اختصاراً لـ Don't Repeat Yourself، والذي يقضي بأن التعليمات البرمجية لا ينبغي أن تتكرر في أكثر من مكان، ويجب تفادي ذلك عبر استعمال الوظائف Methods مثلاً، بحيث بدل تكرار كتابة الأوامر نقوم بتجميعها في وظيفة واحدة ونستدعيها كلما احتجنا إليها.

توجد كذلك المبادئ الخمسة الشهيرة باسم SOLID والتي سنراها بالتفصيل في غضون هذا الكتاب إن شاء الله.

هذه باختصار هي مبادئ التصميم Design Principles، ذكرت بعض مزاياها فقط اختصاراً للوقت، وإلا فإن الحديث عن أهميتها سيطول بنا في مدحها، مما قد يوحى إليك - عزيزي القارئ - أنني على علاقة بها أو أن أهلها قد دفعوا لي المال من أجل "التطيل لها" والحقيقة على خلاف ذلك، فتنبهه !)

ويا رب أم ياسر ما تقرأ هذه الفقرة!

ماهي نماذج التصميم Design Patterns

نماذج التصميم هي حلول عملية تم بناؤها لتناسب وضعيات برمجية معينة، والغرض الأساسي منها هو توفير حلول برمجية لمشاكل شائعة، هذه الحلول تم اختبارها مرات ومرات وأثبتت نجاعتها، لذلك حينما تجد نفسك في وضعية برمجية من الممكن استعمال نموذج للتصميم فيها لا داعي لأن تتعب أعصابك في كتابة حل قد سبقك إليه غيرك، فقط قم بتطبيق هذا الحل واستمتع بالنتيجة.

إذن فنماذج التصميم Design Patterns هي حلول برمجية عملية عامة يمكنك استعمالها في وضعيات برمجية معينة، ويوجد العديد من نماذج التصميم مثل Repository الذي يسمح لنا بتجميع العمليات الممكن أن نقوم بها على البيانات في مكان واحد، أو مثل Singleton الذي يسمح لنا باستعمال Object واحد من نوع معين، وكذلك Factory الذي يسمح لنا بإنشاء الكائنات من أنواع أخرى، والعديد من نماذج التصميم وأشهرها GoF اختصاراً لـ Gang of Four وهي مجموعة من نماذج التصميم التي صاغها ثلة من المحترفين في مجال تصميم البرمجيات.

ماهو الفرق بين مبادئ ونماذج التصميم

الفرق بين مبادئ التصميم Design Patterns ونماذج التصميم هو الفرق بين ما يقوله المهندس وبين ما يفعله البناء، الأول يعطيك تعليمات وتوجيهات عامة لتكون المنشأة موافقة لمعايير البناء الجيد، والثاني يقوم باستعمال حلول فعلية لإنجاز عملية البناء، فطريقة وضع اللبنة استعملها عدة بنائين، وطريقة تبليط الحائط كذلك، وهكذا دواليك...،
فحينما نتحدث عن توجيهات عامة فنحن نتحدث عن مبادئ التصميم Design Principles،
وحينما نتحدث عن حلول عملية لوضعيات معينة فنحن نتحدث عن نماذج التصميم Design Patterns.

يمكننا باختصار أن نقول أن الفرق بين مبادئ التصميم ونماذج التصميم هو أن هذه الأخيرة تعنى بالمستوى المنخفض المتعلق بالحلول البرمجية لوضعيات معينة، بينما مبادئ التصميم تعطيك قواعد ومبادئ عامة على مستوى أعلى لكي توظفها في مشاريعك بغرض تجويدها.

لماذا خصصنا هذا الكتاب لمبادئ SOLID

لأننا كمطورين للبرامج نحتاجها باستمرار لنضمن أن مشاريعنا خاضعة لمعايير الجودة البرمجية، علما أن تطبيق مبادئ SOLID لا يعني أن برامجنا جيدة فهناك عدة اعتبارات أخرى تحكم هذا الجانب، لكن على الأقل استعمال هذه المبادئ يساهم في كتابة كود جيد، قابل للتعديل والصيانة والتوسيع والاختبار، كما أن استعمالها يعطي لمشاريعنا طابع الاحترافية لأن ادعاء الاحتراف يحسنه كل أحد، لكن قراءة الكود تفضح كاتبه.
مبادئ SOLID تعطينا تعليمات وتوجيهات نحتاجها بشكل دائم أثناء اشتغالنا في تطوير البرمجيات، من أجل تفادي الوقوع في أخطاء ومشاكل محتملة، ومن أجل تقوية مشاريعنا والتقليل من عمليات الإصلاح التي إن تكررت بشكل مبالغ فيه تتحول من إصلاح إلى ترقيع، ولا تسألني عن ضرر الترقيع وسل عنه من وضع نقوده في كيس به رقع!
باختصار، خصصنا هذا الكتاب لمبادئ SOLID لقربها الكبير من عملنا اليومي، ولإضافاتها القوية التي تجعلنا نشتغل باحتراف وراحة نفسية يكاد أصحاب "السباغيتي كود Spaghetti Code" يقاتلوننا عليها بالسيوف.

دقيقة من فضلك، ماهي SOLID أصلا

SOLID هي كتابة مختصرة لخمس مبادئ أساسية تسمح لنا بإنشاء برمجيات قوية قابلة للتحديث والتوسيع والاختبار، وكل حرف من حروف هذه الكلمة يمثل مبدأ من المبادئ الخمسة وهي كما يلي:

- حرف S: يرمز إلى مبدأ المسؤولية الواحدة Single Responsibility Principle
- حرف O: يرمز إلى مبدأ الفتح والإغلاق Open-Closed Principle
- حرف L: يرمز إلى مبدأ الاستبدال Liskov Substitution Principle
- حرف I: يرمز إلى مبدأ فصل الواجهات Interface Segregation Principle
- حرف D: يرمز إلى مبدأ انعكاس التبعية Dependency Inversion Principle

قبل أن أقول أي شيء، اسمح لي أن أهمس في أذنك..
الطلاسم أعلاه قد تبث في نفسك شيئا من الرهبة، أعلم ذلك لأنني جربت هذا الشعور قبلك، لكن كيف سيكون شعورك حينما أخبرك أننا سنتناول كل واحدة منها في معزل ونقتلها شرحا؟ بل كيف سيكون شعورك إذا اشتغلنا على مشروع فعلي ورأينا كيف نطبق كل مبدأ من هذه المبادئ بشكل تطبيقي؟ بل أعدك بأكثر من ذلك، فقط ابتسم ودع عنك هذا العبوس، لأنه لا يليق بك!

ما قلنا بلاش عبوس يا راجل!

مصطلحات ستعيش معها في مجال التصميم

في قاموس مجال تصميم البرمجيات ستجد عدة مصطلحات منتشرة بكثرة، مثل مصطلح Coupling، ومصطلح Responsibility، ومصطلح Abstraction وغيرها، هذه المصطلحات تلعب دورا جوهريا في فهم آليات التصميم، حيث سنجدها تدخل في تعريفات بعض المبادئ والنماذج، لذلك يعد الإلمام بها مدخلا أساسيا لتذليل صعوبات هذا المجال وفهم تفاصيله. سنحاول في الفقرات القادمة أن نشرح أبرز المفاهيم التي سنراها بشكل متكرر في مجال تصميم البرمجيات.

ماهي المسؤولية Responsibility

نقصد بالمسؤولية، العمل الذي على الوحدات القيام به، وحينما نقول الوحدات Modules فنحن نقصد أي جزء من الكود يقوم بعمل ما، مثل الوظائف Methods و الكلاسات Classes.

مسؤولية كل وحدة هي العمل الذي عليها القيام به، فمثلا لو عندنا كلاس في المشروع دورها هو قراءة البيانات من قاعدة البيانات فيمكننا أن نقول بأن مسؤولية هذا الكلاس هو قراءة البيانات، أو عندنا كلاس يقوم بالتحقق من سلامة البيانات وصحتها، فهذا العمل الذي ذكرنا هو مسؤولية هذه الكلاس، أو مثلا تكون عندنا وظيفة تقوم بحساب الوزن المثالي فنقول بأن هذه هي مسؤوليتها، وهكذا دواليك.

إذن فالمسؤولية Responsibility هي المهمة المنوطة بوحدة معينة والتي عليها أن تقوم بها.

ماهو الارتباط Coupling

نتحدث عن الارتباط Coupling في سياق العلاقات بين الوحدات Modules، فحينما تكون عندنا كلاس تستعمل Object من كلاس أخرى فنقول بأن هاتين الفئتين مترابطتين، ويمكن لهذا الارتباط أن يكون ضعيفا Loose coupling أو قويا Tight coupling حسب درجة الارتباط بين الوحدات.

سأعطيك مثالا بسيطا، لنفترض أن عندنا كلاس اسمها EmployeeService وهي كلاس تحتوي على مجموعة من الوظائف بما فيها الوظيفة Add(Employee employee) كما يلي:

EmployeeService.cs:

```
public class EmployeeService
{
    public void Add(Employee employee)
    {
        // Save to Database for example.
    }
}
```

ثم عندنا كلاس آخر يستعمل الكلاس EmployeeService بغرض إضافة موظف جديد كما يلي:

TestEmployee.cs:

```
public class TestEmployee
{
    readonly EmployeeService service = new EmployeeService();

    public void Save()
    {
        service.Add(new Employee());
    }
}
```

الكلاس TestEmployee يقوم بإنشاء Instance من النوع EmployeeService، أي صار بينهما ارتباط، هذا الارتباط هو ارتباط قوي Tight Coupling لأن الكلاس TestEmployee مرتبطة

بتفاصيل الكلاس EmployeeService وبالتالي أي تغيير يحدث على مستوى هذا الكلاس الأخير سيجرنا إلى التعديل على الكلاس TestEmployee، وهذه إحدى سلبيات الارتباط القوي. وحتى نفهم الأمر أكثر سنعطي مثالا آخر يبين علاقة الارتباط بين الفئات الثلاثة Book و BookRepository و BookController بحيث:

■ الفئة Book عبارة عن موديل يحتوي على الخصائص الخاصة بالكتاب مثل العنوان، عدد الصفحات، تاريخ النشر.

■ الفئة BookRepository: عبارة عن كلاس يقوم بجلب البيانات من مصدر معين وليكن مثلاً قاعدة بيانات أو RESTful service.

■ الفئة BookController: عبارة عن كلاس يقوم باستهلاك BookRepository بغرض الاستفادة من الوظائف التي يقدمها. ولنبدأ في الأول مع الكود الخاص بالفئة Book:

Book.cs:

```
public class Book
{
    public int Id { get; set; }
    public string Title { get; set; }
    public int NumberOfPages { get; set; }
    public DateTime PublishedAt { get; set; }
}
```

وهنا الكود الخاص بالكلاس BookRepository التي تجلب البيانات الخاصة بالكتب من مصدر معين، تم اختصار الكود لدواعي الشرح وتم الاقتصار على ما يعيننا:

BookRepository.cs:

```
public class BookRepository
{
    public IEnumerable<Book> GetAllBooks()
    {
        // GET DATA FROM A DATA SOURCE
    }
}
```

بالنسبة للفئة التي ستستهلك هذه Repository فهي الفئة BookController، والتي سيكون الكود الخاص بها تقريبا كما يلي:

EmployeeService.cs:

```
public class BookController
{
    public BookRepository Repository { get; set; }

    public BookController(BookRepository repository)
    {
        Repository = repository;
    }

    public IEnumerable<Book> GetAllBooks()
    {
        return Repository.GetAllBooks();
    }
}
```

الكلاس أعلاه يقوم بإنشاء Instance من BookRepository ثم يقوم بإعطائها قيمتها داخل المشيد Constructor، وبعد ذلك يقوم باستعمال هذه Instance من أجل إرجاع بيانات الكتب باستعمال الوظيفة GetAllBooks المعرفة على مستوى BookRepository.

هذه الكلاس باختصار لكي تؤدي عملها فهي تحتاج إلى نسخة من الكلاس BookRepository بمعنى أنها مرتبطة بها، وأن بين الفئتين BookController و BookRepository ارتباط قوي، بحيث لو أردنا في المستقبل أن نغير التفاصيل Details الخاصة بالكلاس BookRepository فإننا ملزمون كذلك بتغيير Details الخاص بالكلاس BookController، أي أن هذين الفئتين مرتبطتان ببعض بشكل كبير، وهو ما يعرف بالارتباط القوي Tight Coupling.

الارتباط القوي يؤثر في الغالب بشكل سلبي على جودة المشاريع لأن الكلاسات المرتبطة يكون لها تأثير مباشر على بعضها البعض، ولأن الارتباطات تجعل من عملية الاختبارات أمرا صعبا، خاصة الاختبارات الأحادية Unit Tests، ناهيك عن صعوبة صيانة الكود وقراءته والزيادة عليه في حال وجود ارتباطات كثيرة ومتشعبة.

قد لا يبدو لك الضرر جليا في هذا المثال الذي نستعمله لأنه مجرد مثال صغير، لكن في المشاريع الفعلية الكبيرة التي تحتاج إلى أن يكون الكود مفهوما Understandable، قابلا للاختبار Testable، قابلا للصيانة Maintainable، قابلا للتوسيع والزيادة Extensible ستكون الارتباطات القوية الكثيرة عائقا حقيقيا أمام هذه المتطلبات.

لذلك ينصح دائما أن يكون التصميم المعتمد في المشروع يسعى إلى تقليل الارتباطات أو إلى توهينها وإضعافها فيما يعرف بعملية Loose Coupling كأقل تقدير عبر تطبيق بعض نماذج التصميم Design Patterns مثل Dependency Injection وغيرها، لكن لأن الحال ما يزال مبكرا للحديث عن هذه الأمور فدعني أختصر عليك الكلام عن الارتباطات.

الارتباطات هي العلاقات بين وحدات المشروع، والتي كلما ازدادات وقويت كان المشروع عصيا على الصيانة والاختبار والتوسيع، وكلما قلت وضعفت تحققت هذه الأمور، وسنرى كيف نقوم بتوهمين وإضعاف الارتباطات بين الكلاسات لاحقا إن شاء الله.

ماهو التماسك Cohesion

التماسك هو مفهوم قريب جدا من مفهوم الارتباط غير أنه لا يعكس العلاقة بين الوحدات، وإنما العلاقة بين المكونات الموجودة في نفس الوحدة، مثلا لو عندنا كلاس يحتوي على مجموعة من المكونات (وظائف، خصائص) التي لا يوجد بينها أي ترابط فنحن نقول بأن مستوى التماسك في هذا الكلاس ضعيف جدا وهو أمر غير جيد من ناحية التصميم، وحينما يكون العكس، أي عندنا كلاس يحتوي على مكونات متناغمة فيما بينها فنقول بأن هذا الكلاس على مستوى عال من التماسك.

مثلا الكلاس التالي يعكس تماسكا ضعيفا نظرا للضعف الترابط بين مكوناته:

Employee.cs:

```
public class Employee
{
    public string Name { get; set; }
    public string Company { get; set; }
    public DateTime DateOfBirth { get; set; }
    public bool IsLocked { get; set; }

    public string GetEmployeeName() => this.Name;

    public int Age() => DateTime.Now.Year - DateOfBirth.Year;

    public bool EmployeeLoginState() => this.IsLocked;
}
```

الكلاس أعلاه غير متماسك لأن عناصره غير مرتبطة فيما بينها، فالكلاس Employee لا ينبغي أن تشتمل على مكون خاص باسم الشركة Company، كما أن حالة المستخدم IsLocked يفترض بها أن تكون في كلاس آخر خاص بمعلومات المستخدم User، بالتالي الكلاس ليس كلاساً متماسكاً نظراً لاشتماله على عناصر غير متماسكة بالنظر إلى المسؤولية الأساسية للكلاس، فهو كلاس مسؤول عن الموظف Employee وليس عن حالة تسجيل الدخول أو معلومات الشركة.

يمكننا التعديل على الكلاس أعلاه لنحصل على ثلاث فئات متماسكة كما يلي، علماً أننا اختصرنا الكود الخاص بالفئات للتركيز على ما يهمنا:

Employee.cs:

```
public class Employee
{
    public string Name { get; set; }
    public DateTime DateOfBirth { get; set; }
    public string GetEmployeeName() => this.Name;
    public int Age() => DateTime.Now.Year - DateOfBirth.Year;
}
```

Company.cs:

```
public class Company
{
    public string Name { get; set; }
    public string Address { get; set; }
}
```

User.cs:

```
public class User
{
    public string UserName { get; set; }
    public string Password { get; set; }
    public bool IsLocked { get; set; }
}
```

الآن صار عندنا ثلاث فئات متماسكة، كل واحدة تحتوي على عناصر مترابطة فيما بينها، وتعكس المسؤولية المنوطة بالكلاس، فالكلاس Employee خاصة بالموظف، والكلاس Company خاصة بالشركة، والكلاس User خاصة بالمستخدم، وبهذا نكون قد جعلنا تصميمنا أكثر تماسكا More cohesive، وهي سمة جيدة تعكس جودة المشروع.

ماهو التجريد Abstraction

لنفهم التجريد Abstraction علينا في الأول أن نفهم الواقعية Concretness، نظرا لقيام المفهوم الأول على المفهوم الأخير، وحتى لا أدخل بك في دوامات من الفلسفة سأعطيك مثلا بسيطا لتفهم الفرق بين الأمرين بشكل جلي وواضح.

حينما تفتح صفحتك على الفيسبوك وترى صديقك قد كتب منشورا على الشكل التالي:

لغة البرمجة صعبة جدا وتستلزم منك الصبر نظرا لتعقيداتها وصعوباتها.

من غير شك فإنك مثلي ستسأل: عن أية لغة برمجة يتحدث صديقك؟

لأن لغة البرمجة هكذا: عبارة عن شيء مجرد Abstract، غير واقعي not concrete، لكن حينما يوضح في منشوره مثلا أنه يتحدث عن لغة سي شارب أو جافا أو بايتون، فهو الآن دخل في

نطاق الواقعية Concreteness لأنه تحدث عن شيء بعينه، وهذا هو الفرق بين التجريد والواقعية.

سأزيدك من الشعور بيتا لفهم الأمر بشكل جيد، حينما نقول: حيوان Animal فنحن نتحدث بشكل مجرد أي أن الحيوان ليس شيئاً واقعياً هو مجرد تجريد Abstraction لنوع معين، لكن حينما نقول أسد Lion، أو نمر Tiger، أو كلب Dog فنحن هنا نتحدث عن نوع موجود وملموس Concrete.

نفس الشيء حينما أقول لك: بشر Human ففي الحقيقة هذا اصطلاح تجريدي ولا يوجد شيء محدد ملموس في الواقع يسمى بشر، الموجود فعلياً هم أشخاص بمواصفات معينة ووظائف محددة، فالبشر عبارة عن Abstract type، والنجار مثلاً أو سائق التاكسي عبارة عن Concrete type.

في البرمجة يسري نفس التعريف، فلو عندنا كلاس تحتوي على وظائفها وخصائصها الخاصة ويمكننا إنشاء نسخ Instances منها، فنحن نتحدث عن فئات واقعية Concrete Classes، وحينما يكون عندنا نوع مجرد لا يحتوي على أية تفاصيل، فقط يقوم بعرض بعض المعلومات العامة التي من الممكن أن تشترك فيها العديد من الأنواع فنحن هنا نتحدث عن التجريد Abstraction، وفي البرمجة العناصر التي تسمح لنا بتطبيق مفهوم التجريد هي الفئات المجردة Abstract classes والواجهات Interfaces.

مأمعنى التفاصيل Details

في الحقيقة يستخدم هذا المفهوم في مجال تصميم البرمجيات كنقيض لمفهوم التجريد Abstraction، حيث يتم استعماله للتدليل على الفئات الواقعية Concrete classes، ومنه يقول روبرت مارتن في شرح البند الثاني من المبدأ الخامس لمبادئ SOLID، والحديث هنا عن مبدأ Dependency Inversion Principle، يقول فيه:

Abstraction should not depend on details, details should depend on abstraction.

ويعني به أن الواجهات Interfaces أو الفئات المجردة Abstract classes لا ينبغي أن تكون مرتبطة بالفئات الواقعية Concrete classes بل العكس هو الذي ينبغي أن يكون. وطبعاً هذا الكلام يأتي في سياق آليات إضعاف الارتباط بين الفئات والذي سنراه في وقته بالتفصيل إن شاء الله.

الشاهد هنا، أن العم بوب (وهو نفسه السيد روبرت مارتن) جعل Details على الطرف النقيض لل Abstraction مما يعني أن التعريف الذي أعطيناه في الفقرة السابقة للفئات

الواقعية Concrete classes يبقى ساريا أيضا على مفهوم التفاصيل Details، وبالتالي يمكننا تعريف هذه الأخيرة بأنها كل كلاس واقعي نستدعيه بشكل مباشر في كلاس آخر، حيث يصبح هذا الكلاس الآخر مقيدا بتفاصيل الكلاس الأول.

ماهو الفرق بين التعديل والتوسيع Modification and Extension

التعديل Modification نقصد به التعديل على الكود الموجود في الكلاس، بحيث نعود إلى الكلاس ونقوم بالتعديل عليها، أي نشتغل على نفس الملف ونعدل ما كنا قد كتبناه سابقا، أو كتبه مطور آخر غيرنا.

أما التوسيع Extension فيعني أن نزيد في النوع الواحد مهام إضافية دون أن نعدل عليه، وذلك من خلال تطبيق بعض الآليات مثل الوراثة Inheritance أو استعمال الوظائف التوسيعية Extension Methods كما هو الحال في الدوت نيت.

مامعنى الانتهاك Violation

الانتهاك هو حينما نقوم بفعل معين يخرق مبدأ من مبادئ التصميم البرمجي، كأن نقوم مثلا بجعل كلاس معين يقوم بعدة مهام بدل أن يقوم بمهمة واحدة فنكون بذلك قد خرقنا المبدأ الأول من مبادئ SOLID والذي يقضي بأن كل كلاس له مسؤولية واحدة فقط لاغير، فممارستنا لهذا الخرق هو انتهاك Violation لهذا المبدأ.

جلسة مع مشروعنا الجديد NumberConverter

مقاربة التعلم عبر المشاريع Project-based learning approach من أحب البيداغوجيات إلى قلبي لأنني ألحظ أثرها ومفعولها بشكل بارز وفي وقت قصير، قياسا بالمقاربة التقليدية التي تجعل المتعلم صندوقا لتكديس المعلومات دون أن يعلم سبيل توظيفها، وكل همه أن يستحضرها لحظة الامتحان ثم لا بأس بعد ذلك إن نسيتها، بل غصبا عنه سينساها كما لو أن المعلومات كانت مخزنة في RAM وليس على Hard Disk.

لذلك سأعتمد في هذا الكتاب مقاربة التعلم عبر المشاريع لنرى كل مبدأ من مبادئ SOLID بشكل عملي، حيث سنخصص كل مبدأ لجزئية معينة تناسب استعماله كما لو أننا نجمع قطع صورة لتكتمل، أو نرتب القطع في لعبة Puzzle الشهيرة.

فكرة المشروع الذي سنعمل عليه بسيطة جدا، برنامج يقوم بتحويل الأرقام العشرية إلى أنظمة رقمية أخرى مثل النظام الثنائي Binary System، والنظام الثماني Octal System،

والنظام الست عشري Hexadecimal System، ثم شيئاً فشيئاً سنضيف إليه بعض المزايا حسب حاجتنا.

يمكنك تحميل المشروع من الرابط التالي:

<https://github.com/Essaadani/solid>

المبدأ الأول: المسؤولية الواحدة Single Responsibility

أول مبدأ من مبادئ SOLID هو مبدأ Single Responsibility الذي يقول لنا بأن كل وحدة Module عليها أن تقوم فقط بمسؤولية Responsibility واحدة لاغير. ويقول نص المبدأ:

Classes should have a single responsibility and thus only a single reason to change.

نص المبدأ هو نفس الكلام الذي ذكرناه، باستثناء الفقرة الأخيرة التي تبين أن سبب تغير الكلاس ينبغي أن يكون واحدا فقط وهو نفس فكرة المسؤولية الواحدة، حيث الكلاس الذي يقوم بمسؤولية واحدة فقط سيكون هنالك سبب واحد فقط ليتغير.

لماذا نحتاج إلى مبدأ المسؤولية الواحدة SRP

تقسيم فئات المشروع إلى فئات صغيرة، محددة، تقوم بعمل واضح يجعل المشروع سهل القراءة والفهم Understandable، قابل للصيانة في أي وقت Maintainable، وكذلك قابل للاختبار Testable، لأن كل كلاس عبارة عن كتلة واحدة تحتوي على الوحدات Units الخاصة بها والتي يمكننا إجراء اختبارات أحادية Unit Tests عليها. بالإضافة إلى ما تقدم فإن الفئة التي تقوم بمسؤولية واحدة أكثر موثوقية من الفئة التي تقوم بعدة مهام، ثم لأن فصل المهام Separation of concerns يجعل المشروع قويا والمشاكل قابلة للسيطرة لأنها مرتبطة بوحدات صغيرة يسهل التحكم فيها، ناهيك عن أن تطبيق مبدأ المسؤولية الواحدة يجعل الكلاس أكثر تماسكا More cohesive.

متى نحتاج إلى مبدأ المسؤولية الواحدة SRP

حينما تجد أن كلاس معين يقوم بأكثر من مسؤولية، كأن تجد كلاس يقوم بقراءة البيانات وبحفظها في قاعدة البيانات، وتسجيل الأنشطة Logging، والتحقق من سلامة البيانات Validation، وغيرها من المهام الأخرى، فهذا إنذار لضرورة تقسيم هذا الكلاس إلى كlasses صغيرة مركزة ومتماسكة Focused and Cohesive classes.

كيف نطبق مبدأ المسؤولية الواحدة SRP

في مشروعنا الذي يقوم بالتحويل من النوع العشري إلى قواعد رقمية أخرى، نجد الكلاس الرئيسي التالي:

NumberConverter.cs:

```
namespace SolidSample
{
    public enum BaseType
    {
        Binary = 2,
        Octal = 8,
        None = 0
    }

    public class NumberConverter
    {
        public int DecimalNumber { get; set; }

        public void Convert()
        {
            Console.WriteLine("Program is starting...");

            Console.WriteLine("Enter the number to convert:");
            DecimalNumber = int.Parse(Console.ReadLine());

            Console.WriteLine("Enter the base type (Ex: 2,8):");
            var baseType = (BaseType)int.Parse(Console.ReadLine());

            string result = String.Empty;

            switch (baseType)
            {
                case BaseType.Binary:
                    result = System.Convert.ToString(DecimalNumber, 2);
                    break;

                case BaseType.Octal:
                    result = System.Convert.ToString(DecimalNumber, 8);
                    break;

                default:
                    result = "No base found!";
                    break;
            }

            Console.WriteLine($"The result is: {result} ");

            Console.WriteLine("Program is ending..");
        }
    }
}
```

في الأول أنشأنا enum تحتوي على أنواع الأنظمة التي نريد التحويل إليها، ومبدئياً عندنا النوع الثنائي Binary، والنوع الثماني Octal، والقيمة الثالثة في حال لم يتم المستخدم باختيار أحد النوعين.

بعد ذلك عندنا الكلاس الرئيسي NumberConverter الذي يحتوي على الخصائص التالية:

■ DecimalNumber: وهو القيمة الرقمية الأساسية التي سنعمل على تحويلها

■ Base: وهو نوع النظام الرقمي الذي نريد التحويل إليه

■ Result: وهي النتيجة التي نريد طباعتها بعد القيام بعملية التحويل

ثم انتقلنا إلى الوظيفة الجوهرية Convert التي تقوم بطلب إدخال القيمة الرقمية من المستخدم، وكذلك رمز النظام المراد التحويل إليه إما الثنائي أو الثماني، ثم بعد ذلك تقوم بفرز الحالات حسب نوع النظام الذي اختاره المستخدم وتجري عملية التحويل، ثم تطبع النتيجة في شاشة الكونسول.

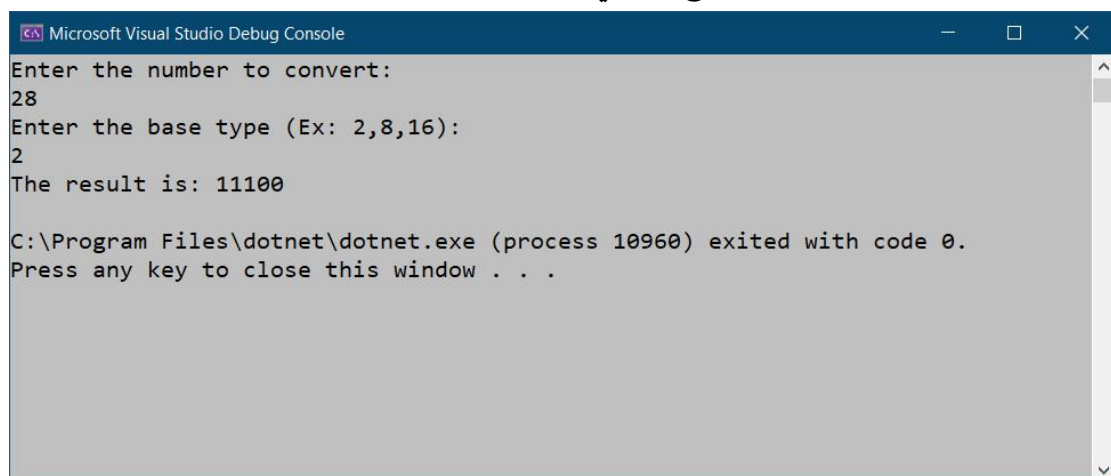
لتجربة الوظيفة Convert، يمكننا أن نأتي إلى الكلاس الرئيسي Program.cs ونكتب ما يلي:

Program.cs:

```
class Program
{
    static void Main(string[] args)
    {
        NumberConverter converter = new NumberConverter();

        converter.Convert();
    }
}
```

عند التنفيذ سيكون تسلسل البرنامج كما يلي:



```
Microsoft Visual Studio Debug Console
Enter the number to convert:
28
Enter the base type (Ex: 2,8,16):
2
The result is: 11100

C:\Program Files\dotnet\dotnet.exe (process 10960) exited with code 0.
Press any key to close this window . . .
```

من الجانب العملي، فالبرنامج يشتغل على الشكل المطلوب، لكن على مستوى التصميم Design فقد تكون لاحظت معي أن الكلاس NumberConverter يقوم بعدة مسؤوليات كالتالي:

■ يقوم بمسؤولية عرض الرسائل في شاشة الكونسول

■ يقوم بمسؤولية قراءة البيانات من شاشة الكونسول

■ يقوم بتسجيل الأنشطة Logging

■ يقوم بعملية التحويل وهي مسؤوليته الأساسية

لذلك علينا أن نكلفه فقط بمسؤولية التحويل وأن نضع المسؤوليات الأخرى في فئات مستقلة لنكون بذلك قد طبقنا مبدأ المسؤولية الواحدة Single Responsibility Principle بنجاح.

سنقوم بإنشاء كلاس نسميها Logger دورها هو طباعة الرسائل في شاشة الكونسول، وقد نتساهل حالياً ونترك لها أيضاً مسؤولية تسجيل الأنشطة Logging لسبب بسيط هو أن المسؤوليتين تشتركان في نفس العمل وهو طباعة الرسائل على شاشة الكونسول وبالتالي وفي إطار تنفيذ مبدأ DRY الذي يختصر Don't Repeat Yourself والذي ينص على أهمية تفادي تكرار الكود الذي يؤدي نفس الغرض، سندمج للكلاس بالقيام بالمسؤوليتين معا. الكلاس التي ستقوم بطباعة النصوص وتسجيل الأنشطة كما يلي:

Logger.cs:

```
public class Logger
{
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}
```

والكلاس الثاني سنكلفه بقراءة البيانات من شاشة الكونسول ومحتواه كما يلي:

Reader.cs:

```
public class Reader
{
    public int ReadInteger()
    {
        return int.Parse(Console.ReadLine());
    }
}
```

الآن سيصبح شكل الكلاس NumberConverter كالآتي:

NumberConverter.cs:

```
public class NumberConverter
{
    public int DecimalNumber { get; set; }

    public Logger Logger { get; set; } = new Logger();
    public Reader Reader { get; set; } = new Reader();

    public void Convert()
    {
        Logger.Log("Program is starting...");

        Logger.Log("Enter the number to convert:");
        DecimalNumber = Reader.ReadInteger();

        Logger.Log("Enter the base type (Ex: 2,8):");
        var baseType = (BaseType)Reader.ReadInteger();

        string result = String.Empty;

        switch (baseType)
        {
            case BaseType.Binary:
                result = System.Convert.ToString(DecimalNumber, 2);
                break;

            case BaseType.Octal:
                result = System.Convert.ToString(DecimalNumber, 8);
                break;

            default:
                result = "No base found!";
                break;
        }

        Logger.Log(result);

        Logger.Log("Program is ending..");
    }
}
```

الآن من الناحية المنهجية صرنا أكثر انتظاماً، والكود الذي قمنا بكتابته سهل القراءة، مقسم حسب المهام، حيث كل كلاس تؤدي مسؤوليتها الخاصة بها، وبالتالي قمنا بتطبيق مبدأ المسؤولية الواحدة بنجاح.

خلاصة مبدأ المسؤولية الواحدة SRP

على الكلاس الواحد أن يقوم بمسؤولية واحدة فقط لا غير.

المبدأ الثاني: مبدأ الفتح والإغلاق

Open-Closed Principle

ينص هذا المبدأ على أن الوحدات Modules مثل الكلاسات ينبغي أن تكون مفتوحة على التوسيع Open for extension ومغلقة في وجه التعديل Closed for modification. هذا الكلام باختصار يعني لو عندك كلاس ستحتاج إلى إضافة بعض الأمور إليها، فإنه من الجيد أن تستطيع القيام بعملية الإضافة عبر توسيع الفئة من خلال آلية الوراثة Inheritance مثلاً، وليس عبر التعديل المباشر في السورس كود. لنفترض في مشروعنا أن العميل طلب منا أن نضيف في البرنامج خاصية التحويل إلى النوع الست عشري Hexadecimal، في الحالة العادية سندخل مباشرة إلى الوظيفة Convert ونضيف case جديدة إلى switch statement وكذلك إضافة قيمة جديدة إلى enum الخاصة بالأنواع كما يلي:

```
public enum BaseType
{
    Binary = 2,
    Octal = 8,
    Hexadecimal = 16,
    None = 0
}
```

وشكل الوظيفة Convert كالآتي (سنكتب الجزء الذي يهمنا فقط):

NumberConverter.cs:

```
switch (baseType)
{
    case BaseType.Binary:
        result = System.Convert.ToString(DecimalNumber, 2);
        break;

    case BaseType.Octal:
        result = System.Convert.ToString(DecimalNumber, 8);
        break;

    case BaseType.Hexadecimal:
        result = DecimalNumber.ToString("X");
        break;

    default:
        result = "No base found!";
        break;
}
```

الآن قد يبدو أننا "جبنا الذيب من ذيلو" كما يقول إخوتنا في مصر، لكن في الحقيقة نحن وضعنا حلاً ترقيعياً على الرغم من أنه يؤدي المطلوب، لسبب بسيط هو أننا خرقنا مبدأ Open-Closed Principle وقمنا بتعديل Behavior الخاص بالوظيفة Convert دون اللجوء إلى عملية التوسيع Extension وإنما من خلال التعديل المباشر.

وقد تسألني في انتشاء: ومالو يا عم؟ ماهو شغال كده زي الفل! وطبعاً سؤالك بريء لأنه يليق بالوضعية البرمجية التي نشتغل عليها والتي تتصف بالبساطة، لكن في المشاريع الكبرى التي ستكون من العاملين فيها إن شاء الله، الأمور لا تمشي بهذا الشكل، لأنك قد تقوم ببناء API وترسلها إلى عميل لا يعرف عنها سوى Contract، أو ترسلها له مجمعة على شكل DLL Assembly، وبالتالي لا يستطيع أن يعدل على الكود بشكل مباشر، فأنت حينها مطالب بشدة أن تجعل فئاتك قابلة للتوسيع لأن التعديل المباشر ليس متاحاً في كل الأوقات، أضف إلى ذلك خطورة هذه الممارسة التي قد تؤثر سلباً على الكود الذي تم اختباره فتؤدي عملية تعديل سلوك الوظيفة إلى إفشال Unit Tests الخاصة بها. هذا مجرد وجه من عدة وجوه لإبراز أهمية استعمال هذا المبدأ في تصميمك البرمجي، لذلك يجب أن تأخذه بعين الاعتبار وبجدية إن كنت تنوي أن ترقى بمسارك المهني وتتعامل مع عملاء من طينة الكبار (حلو التعبير ده).

لماذا نحتاج إلى مبدأ الفتح والإغلاق OCP

نحتاج إلى تطبيق هذا المبدأ من أجل جعل وحداتنا البرمجية More extensible and maintainable، لأن فتحها على التوسيع يسمح بالإضافة إليها دون التأثير على ما هو موجود فيها، على عكس التعديل المباشر الذي قد يضر بأجزاء كانت شغالة، وقد يؤدي إلى تعطيل الاختبارات الأحادية الخاصة بالوظيفة المعدلة.

متى نحتاج إلى مبدأ الفتح والإغلاق OCP

في الحقيقة لست ملزماً بأن تحمل هم هذا المبدأ في كل كلاس تقوم بكتابتها، وإنما استحضره في الحالات التي ترى أن الكلاس الذي ستقوم بكتابتها من الممكن أن يتطلب تعديلات مستقبلية.

كيف نطبق مبدأ الفتح والإغلاق OCP

الآن دعنا نعود إلى مشروعنا، ونحاول أن نطبق Open Closed Principle لنفهمه بوضوح، قبل ذلك حاول أن تفكر معي في حل محتمل يناسب وضعيتنا الحالية.

اقترحي الذي أتقدم به وأرجو أن يعجبك هو أن ننشئ كلاس مجرد abstract class نسميه Converter أو واجهة نسميها IConverter ونقوم بوضع العناصر التي من الممكن أن نعيد تعريفها فيها، ثم نرث منها الكلاسات التي سنحتاجها مثل BinaryConverter و OctalConverter وكذلك النوع الجديد الذي نريد إضافته وهو HexadecimalConverter، فإذا احتجنا أن نعدل على سلوك الوظيفة Convert في المستقبل نأتي بكل بساطة ونرث من النوع Converter ونضع Behavior الذي يناسبنا.

إذن بتطبيق ما ذكرنا، فإن الكلاس المجرد Converter قد يكون كما يلي:

Converter.cs:

```
public abstract class Converter
{
    public int DecimalNumber { get; set; }

    public Converter(int decimalNumber)
    {
        DecimalNumber = decimalNumber;
    }

    public abstract string Convert();
}
```

الكلاس بكل بساطة يقوم باستقبال قيمة DecimalNumber عبر المشيد ويسندها إلى property الخاصة به، ثم يعرف وظيفة مجردة اسمها Convert. الآن شكل الكلاسات التي سترث من هذا الكلاس المجرد ستكون كما يلي:

BinaryConverter.cs:

```
public class BinaryConverter : Converter
{
    public BinaryConverter(int decimalNumber)
        : base(decimalNumber)
    {
    }

    public override string Convert()
    {
        return $"The result is: {System.Convert.ToString(DecimalNumber, 2)}";
    }
}
```

OctalConverter.cs:

```
public class OctalConverter : Converter
{
    public OctalConverter(int decimalNumber)
    : base(decimalNumber)
    {
    }

    public override string Convert()
    {
        return $"The result is: {System.Convert.ToString(DecimalNumber, 8)}";
    }
}
```

والآن نستطيع أن نضيف كلاس جديد للتحويل إلى النوع الست عشري دون الحاجة إلى تغيير سلوك الوظيفة Convert إذ يكفي أن نعمل Extend للكلاس الرئيسي Converter كما يلي:

HexadecimalConverter.cs:

```
public class HexadecimalConverter : Converter
{
    public HexadecimalConverter(int decimalNumber) : base(decimalNumber)
    {
    }

    public override string Convert()
    {
        return $"The result is: {DecimalNumber.ToString("X")}";
    }
}
```

بالنسبة لشكل الوظيفة Convert في الكلاس NumberConverter سيصبح بالشكل التالي:

NumberConverter.cs:

```
public void Convert()
{
    Logger.Log("Program is starting...");

    Logger.Log("Enter the number to convert:");
    DecimalNumber = Reader.ReadInteger();

    Logger.Log("Enter the base type (Ex: 2,8, 16):");
    var baseType = (BaseType)Reader.ReadInteger();

    string result = String.Empty;

    switch (baseType)
    {
        case BaseType.Binary:
            var binaryConverter = new BinaryConverter(DecimalNumber);
            result = binaryConverter.Convert();
            break;
    }
}
```

```

        case BaseType.Octal:
            var octalConverter = new OctalConverter(DecimalNumber);
            result = octalConverter.Convert();
            break;

        case BaseType.Hexadecimal:
            var hexadecimalConverter = new
HexadecimalConverter(DecimalNumber);
            result = hexadecimalConverter.Convert();

            break;

        default:
            result = "No base found!";
            break;
    }

    Logger.Log(result);

    Logger.Log("Program is ending..");
}

```

أعتقد صار جلياً عندك أهمية مبدأ الفتح والإغلاق بعد أن قدمنا لهذا الفصل بشرح نظري مركز وأردفناه بمثال عملي موجز يبين أهمية المبدأ وكيفية استعماله.

خلاصة مبدأ الفتح والإغلاق OCP

يشير إلينا مبدأ الفتح والإغلاق Open Closed Principle بضرورة الابتعاد عن التعديل المباشر لسلوك الوظائف Methods behavior لما لذلك من أثر سلبي تقدم بيانه، وإنما الواجب أن نعطي للكلاسات القدرة على أن تصبح Extensible، بحيث عند الحاجة إلى التعديل نلجأ إلى عملية إنشاء أنواع جديدة وليس إلى تغيير السورس كود الخاص بالنوع.

المبدأ الثالث: مبدأ الاستبدال Liskov

Substitution Principle

قبل أن نقوم بشرح مبدأ الاستبدال Liskov Substitution Principle سنجري بعض التعديلات على برنامجنا لكي يكون شرح هذا المبدأ واضحاً بشكل كبير.

في الأول قد تلاحظ معي أن الوظيفة Convert الموجودة في الكلاس NumberConverter تحتوي على العديد من الأوامر البرمجية التي يمكن إجراء Refactoring عليها، وإلا فإن شكل الوظيفة سيُسوء مع ظهور متطلبات جديدة، ولك أن تتخيل إذا طلب منا العميل إقحام أنواع أخرى جديدة للتحويل إليها وكم من case ستصبح في Switch statement.

نموذج Factory Design Pattern

لذلك سنلجأ إلى حيلة ذكية تتمثل في تطبيق أحد نماذج التصميم Design Patterns والذي يسمى Factory ودوره هو إنشاء الأنواع Creating types، بحيث سنوكل له مهمة إنشاء الأنواع بدل أن نضعها داخل الوظيفة Convert.

لذلك تعالوا لنقم بإنشاء كلاس جديد نسميه ConverterFactory ونكتب فيه المحتوى التالي:

ConverterFactory.cs:

```
public class ConverterFactory
{
    public static Converter Create(BaseType baseType, int decimalNumber)
    {
        if (baseType == BaseType.Binary)
        {
            return new BinaryConverter(decimalNumber);
        }

        else if (baseType == BaseType.Octal)
        {
            return new OctalConverter(decimalNumber);
        }

        else if (baseType == BaseType.Hexadecimal)
        {
            return new HexadecimalConverter(decimalNumber);
        }
        else
        {
            return null;
        }
    }
}
```

لاحظ أن هذا الكلاس يستقبل النوع المراد التحويل إليه ويقوم بإنشاء Object من النوع المناسب، وهي حيلة ذكية لتطبيق أحد أشهر مبادئ التصميم المعروف ب Inversion Of Control حيث قلبنا عملية إنشاء الأنواع من الكلاس NumberConverter إلى الكلاس ConverterFactory.

مبدأ قلب التحكم Inversion of Control Principle

ومبدأ قلب التحكم Inversion of Control ليس من ضمن مبادئ SOLID، لكنه يصب في نفس أهدافها، حيث نستعمله لجعل كل كلاس يقوم فقط بالمسؤولية المنوطة به، أما عملية إنشاء الأنواع Creating instances فهي مسؤولية إضافية يستحسن أن نقلب التحكم فيها من خلال توكيلها إلى نوع جديد عبر تطبيق نموذج التصميم Factory Design Pattern. هذا باختصار هو مبدأ قلب التحكم Inversion of Control، الآن لنواصل عملنا، لكن قبل ذلك قد تلاحظ أن الكود الخاص ب ConverterFactory يحتوي على عدة if statement وهي عادة سيئة يستحسن تفاديها لأن كل if تقوم بإنشاء مسار تنفيذ Code Path خاصة بها مما يعني الحاجة إلى إنشاء Unit test خاص بكل حالة، أضف إلى ذلك صعوبة قراءة الكود، وطوله غير المبرر.

إذن سنلجأ إلى حيلة ذكية أخرى لتقليص الكود ولتفادي كتابة if statements أقصى ما نستطيع، والحل هذه المرة مع Reflection.

استعمال Reflection لإنشاء الأنواع

تسمح لنا Reflection بإنشاء الأنواع types بشكل ديناميكي وربطها مع أنواع موجودة عندنا في البرنامج، ولأن المقال يتضح بالمثل، فلا أرى أفضل من أن نشرح Reflection بالتطبيق. سنغير محتوى الكلاس NumberFactory ليصبح كما يلي:

ConverterFactory.cs:

```
public class ConverterFactory
{
    public static Converter Create(BaseType baseType, int decimalNumber)
    {
        try
        {
            return (Converter)
                Activator.CreateInstance(
                    Type.GetType($"SolidSample.{baseType.ToString()}Converter"),
                    new object[] { decimalNumber });
        }
    }
}
```

```

        catch (Exception)
        {
            return null;
        }
    }
}

```

إذا استثنينا الأمر try catch فإننا اختصرنا كل الكود في الأمر التالي:

```

Activator.CreateInstance(
    Type.GetType($"SolidSample.{baseType.ToString()}Converter"),
    new object[] { decimalNumber }
);

```

الكود أعلاه يقوم بإنشاء instance من نوع محدد حسب BaseType المرسل في argument، ويقوم بتمرير البرامترات التي تنتظرها هاته Instance ولأن الأنواع التي ترث من Converter كلها لديها Property باسم decimalNumber ينتظرها المشيد Constructor من أجل أن يقوم بإنشاء نسخة من النوع، فقد أرسلناه أيضا في عملية Reflection.

الأمر Type.GetType يستخرج نوع الكلاس من النص الممرر له، وفي حالتنا هذه سيكون اسم النوع هو SolidSample والذي هو اسم namespace ثم اسم نوع التحويل وهو القادم في parameter baseType ثم نقوم بدمجه مع الكلمة Converter لنحصل على اسم النوع كاملا والذي نريد إنشاء instance منه.

يعني لو مررنا إلى factory قيمة BaseType.Octal فإن الأمر Type.GetType سينشئ نوع من النص التالي:

SolidSample.OctalConverter

وهكذا دواليك.

الآن أنجزنا الكثير، بقي أن نقوم بتعديل طفيف على الوظيفة Convert لتستفيد من هذا الجهد الذي بذلناه، وبالتالي سيصير محتواها كما يلي:

NumberConverter.cs:

```

public void Convert()
{
    Logger.Log("Program is starting...");

    Logger.Log("Enter the number to convert:");
    DecimalNumber = Reader.ReadInteger();

    Logger.Log("Enter the base type (Ex: 2,8):");
    var baseType = (BaseType)Reader.ReadInteger();
}

```

```

var type = ConverterFactory.Create(Base, DecimalNumber);

string result = type.Convert();

Logger.Log(result);

Logger.Log("Program is ending..");
}

```

الآن لاحظ كيف صار شكلها جميلا ومتناسكا من خلال تطبيق مبدأ Inversion of Control ونموذج Factory Design Pattern.

أرجو أن تكون قد استوعبت هذا الفصل بشكل جيد، وإلا فإنك مطالب بالعودة إليه من أوله وقراءته وتطبيقه، لأن القادم يرتكز على الحالي.

إلى حدود الساعة قمنا فقط ب Refactoring للكود الموجود، ولم نتطرق بعد إلى شرح مبدأ Liskov Substitution Principle.

هذا المبدأ باختصار ينص على أن الوظائف التي تستعمل Object من نوع رئيسي Base type ينبغي أن تؤدي المطلوب منها حتى ولو تم استبدال هذا النوع الرئيسي بنوع فرعي يرث منه. ما معنى هذا الكلام؟

تعالوا بنا نتناول مثالا نشرح من خلاله المسألة، وليكن ذلك أشهر مثال يتم به شرح مبدأ Liskov Substitution Principle وهو مشكلة Ellipse-Circle و Rectangle-Square.

هندسيا فالمربع عبارة عن مستطيل، أو بتعبير أدق حالة خاصة من المستطيل حيث يتساوى الطول والعرض.

برمجيا قد نرى أن المربع ممكن أن يكون Subtype / Derived من النوع الرئيسي: المستطيل.

مشكلة الدائرة والشكل البيضاوي Circle-Ellipse Problem

نفس الكلام ينطبق على الدائرة والشكل البيضاوي، فالشكل البيضاوي هو شكل دائري، بينما الدائرة هي حالة خاصة من الشكل البيضاوي له نفس الشعاع Radius.

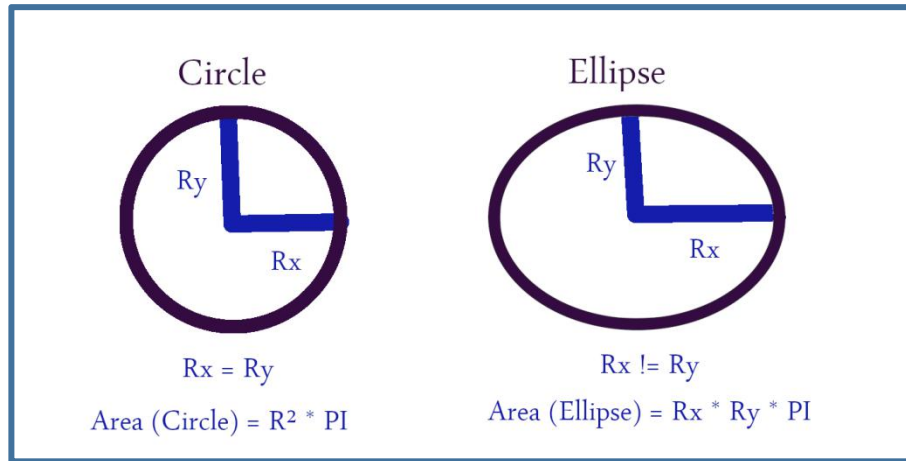
وبالتالي فإن صيغة حساب مساحة الدائرة كما يلي:

$$\text{Area}_{\text{Circle}} = R * R * \text{PI} = R^2 * \text{PI} \quad (\text{لأن الشعاع } R \text{ هو نفسه في الدائرة فبالتالي كتبنا } R^2)$$

بينما مساحة الشكل البيضاوي تكون كما يلي:

$$\text{Area}_{\text{Ellipse}} = R_x * R_y * \text{PI} \quad (\text{حيث } R_x \text{ هو الشعاع الأول، و } R_y \text{ هو الشعاع الثاني) والصورة التالية}$$

توضح بجلاء هذه العلاقات الرياضية:



حتى لا أكثر عليك الكلام، شاهد معي شكل الكلاس Ellipse:

Ellipse.cs:

```
public class Ellipse
{
    public double Rx { get; set; }
    public double Ry { get; set; }

    public virtual void SetRx(double value)
    {
        Rx = value;
    }
    public void SetRy(double value)
    {
        Ry = value;
    }
}
```

لأن الدائرة Circle عبارة عن Ellipse فبرمجيا يمكننا أن نشتق الكلاس Circle من الكلاس Ellipse، كما يلي:

Circle.cs:

```
public class Circle : Ellipse
{
    public override void SetRx(double value)
    {
        base.SetRx(value);
        Ry = value;
    }
}
```

على مستوى الكلاس Circle قمنا بإعادة تعريف الوظيفة SetRx لكي يصبح Behavior الخاص بها مناسباً للدائرة، حيث كل من Rx و Ry سيأخذان نفس القيمة القادمة في argument.

الآن سننشئ كلاساً جديداً لحساب مساحة الأشكال الدائرية كما يلي:

AreaCalculator.cs:

```
public class AreaCalculator
{
    public double Area(Ellipse ellipse)
    {
        return ellipse.Rx * ellipse.Ry * Math.PI;
    }
}
```

العلاقة الحسابية المستعملة في الكلاس AreaCalculator صحيحة رياضياً، وصحيحة نسبياً من المنظور البرمجي لكنها في بعض الحالات ستعطي نتائج غير صحيحة. لو قمنا بتجربة الكلاس فسوف نحصل على النتائج التالية إذا تعاملنا مع الأشكال البيضوية بشكل عام:

Program.cs:

```
Ellipse ellipse = new Ellipse();

ellipse.SetRx(5);
ellipse.SetRy(4);

AreaCalculator calculator = new AreaCalculator();

// result = 5 * 4 * PI ==> 62.83 (True)
double result = calculator.Area(ellipse);
```

أنشأنا شكلاً بيضوياً وأعطينا ل Rx القيمة 5، ول Ry القيمة 4، المفترض أن مساحة هذا الشكل ستكون هي 5 في 4 في الثابتة PI، أي أن النتيجة ستكون على التقريب 62.83، وهي نفس النتيجة التي يعطيها برنامجنا. أي أن العملية الحسابية تمثلي بشكل جيد. لكن المفاجأة ستكون حينما نتعامل مع الشكل الدائري Circle، ركز في المثال التالي:

Program.cs:

```
Circle circle = new Circle();

circle.SetRx(5);
circle.SetRy(4);

//result = 5 * 5 * PI ==> 78.54 But Actual = 62.83(False)
result = calculator.Area(circle);
```

المفترض أن تكون مساحة الدائرة هي 5 في 5 (لأن الشعاع هو نفسه) في الثابتة الرياضية PI وبالتالي نتوقع أن يكون الناتج هو 78.54، فإذا بنا نفاجأ بالنتيجة 62.83 فما الذي حدث يا ترى؟

الذي حدث هو أن الكلاس Circle قامت بإعطاء Rx و Ry نفس القيمة 5 لكن الوظيفة SetRy أعطت ل Ry القيمة 4 وبالتالي تم حساب الدائرة كما لو أنها Ellipse. هذا المشكل هو خرق لمبدأ Liskov Substitution Principle، الذي ينص على أن الفئات المشتقة Derived types ينبغي أن تعمل كالفئة الرئيسية Base type دون أن يؤثر ذلك على Behavior الخاص بالوظيفة التي تستعمله.

الحل الأول لمشكلة Circle-Ellipse Problem

لحل هذه الإشكالية يمكننا بكل بساطة أن نعيد تعريف SetRy على مستوى الكلاس Circle بنفس الشكل الذي أعدنا به تعريف SetRx أي:

Circle.cs:

```
public override void SetRy(double value)
{
    Ry = Rx;
}
```

لاحظ أن كل من Rx و Ry صارت لهما نفس القيمة، لو نفذنا الآن سنحصل على نتيجة صحيحة لكن الكود الذي كتبنا ليس جيداً، لأنه لو تلاحظ معي فالوظيفة SetRy التي أعدنا تعريفها تستقبل برامتر value ولا تستعمله بتاتا.

الحل الثاني لمشكلة Circle-Ellipse Problem

الحل الثاني هو أن نعتبر Circle كلاساً مستقلاً، لا يرث من الكلاس Ellipse وبالتالي في الكلاس AreaCalculator سنضطر إلى كتابة كود إضافي. أي أن شكل الكلاس Circle سيكون كما يلي:

Circle.cs:

```
public class Circle
{
    public double Radius { get; set; }
    public void SetRadius(double radius)
    {
        Radius = radius;
    }
}
```

وشكل الكلاس AreaCalculator سيكون كما يلي:

AreaCalculator.cs:

```
public class AreaCalculator
{
    public double Area(object shape)
    {
        if(shape is Ellipse)
        {
            return ((Ellipse)shape).Rx * ((Ellipse)shape).Ry * Math.PI;
        }

        else if (shape is Circle)
        {
            return ((Circle)shape).Radius * ((Circle)shape).Radius * Math.PI;
        }

        else
        {
            return 0;
        }
    }
}
```

لكن هذا الحل يخرق مبادئ التصميم في عدة وجوه، أبرزها أنه يخرق مبدأ Liskov Substitution نفسه لأن التحقق من الأنواع بالشكل الذي اعتمدناه في الكلاس AreaCalculator يعد خرقاً لمبدأ LSP، إضافة إلى سوء التصميم Bad design من خلال إنشاء فئتين لا علاقة بينهما، ونحن نعلم يقيناً أن Circle هي عبارة عن Ellipse.

الحل الثالث لمشكلة Circle-Ellipse Problem

الحل الثالث أن نكتفي فقط بالكلاس Ellipse ونضيف إليه خاصية منطقية Boolean property نسميها مثلاً IsCircle، وقبل أن نسند القيم في كل من SetRx و SetRy نتحقق من هذه الخاصية، إن كانت true نمرر القيم على أساس أننا نتعامل مع دائرة، وإن كانت false نمرر القيم على أساس أننا مع أي شكل دائري كما يبين المثال التالي:

Ellipse.cs:

```
public class Ellipse
{
    public double Rx { get; set; }
    public double Ry { get; set; }

    public bool IsCircle => Rx == Ry;

    public void SetRx(double value)
    {
        if (IsCircle)
        {
            Ry = value;
        }
    }
}
```



```

    }

    Rx = value;
}

public void SetRy(double value)
{
    if (IsCircle)
    {
        Ry = Rx;
    }

    Ry = value;
}
}

```

لاحظ أن الخاصية IsCircle سهلت علينا العمل بشكل كبير، حيث تقارن Rx ب Ry فإن وجدت لديهما نفس القيمة أدركت أن الشكل عبارة عن دائرة Circle، والعكس بالعكس. ثم داخل كل من SetRx و SetRy نقوم بإسناد القيم حسب قيمة IsCircle، الآن لو كتبنا البرنامج التالي:

Program.cs:

```

Ellipse ellipse = new Ellipse();

ellipse.SetRx(5);
ellipse.SetRy(4);

AreaCalculator calculator = new AreaCalculator();

// Rx = 5; Ry = 4; Rx != Ry => IsCircle = False
double result = calculator.Area(ellipse);

ellipse.SetRx(5);
ellipse.SetRy(5);

// Rx = Ry = 5 => IsCircle = True
result = calculator.Area(ellipse);

```

في المرة الأولى لدينا Rx و Ry مختلفان إذن سيتم حساب الشكل البيضاوي على أساس أنه ليس Circle، وفي المرة الثانية لأنهما متساويان سيتم حساب الشكل على أنه Circle، وهكذا نكون قد قمنا بحل المشكل بطريقة ذكية وبأقل كود حيث قمنا بالاستغناء عن الكلاس Circle.

لماذا نحتاج إلى مبدأ الاستبدال LSP

نحتاج إلى مبدأ Liskov Substitution Principle من أجل أن نضمن أن الفئات المشتقة تستطيع أن تتصرف كما لو أنها فئات رئيسية دون أن يؤثر ذلك على سلوك الوظيفة التي تستعمل هذا

النوع من الفئات، وذلك لضمان أن الكائنات Objects المشتقة لا تؤثر على عمل الوظائف في حال استعمالها مكان الأنواع الرئيسية.

متى نحتاج إلى مبدأ الاستبدال LSP

نحتاج إلى مبدأ LSP حينما نجد بأن الأنواع الفرعية لا تتصرف بنفس الكيفية التي تتصرف بها الأنواع الرئيسية كما رأينا في موضوع Ellipse-Circle Problem. كما نحتاج إلى تفعيل هذا المبدأ حينما نقوم بتطبيق واجهة Interface في كلاس معين، فنلاحظ أن العديد من الوظائف لم تأخذ Implementation وبالتالي ستعطي استثناء من نوع NotImplementedException.

في الأماكن التي نستعمل فيها بعض الروابط التحقق من النوع مثل الرابط is أو رابط التحويل as فهناك احتمالية كبيرة أن يكون هنالك انتهاك لمبدأ LSP. لذلك حينما ترى مؤشرا من هذه المؤشرات فأنت مطالب بجعل الكود الخاص بك موافقا لمبدأ Liskov Substitution Principle لتتفادى المشاكل الناتجة.

كيف نطبق مبدأ الاستبدال LSP

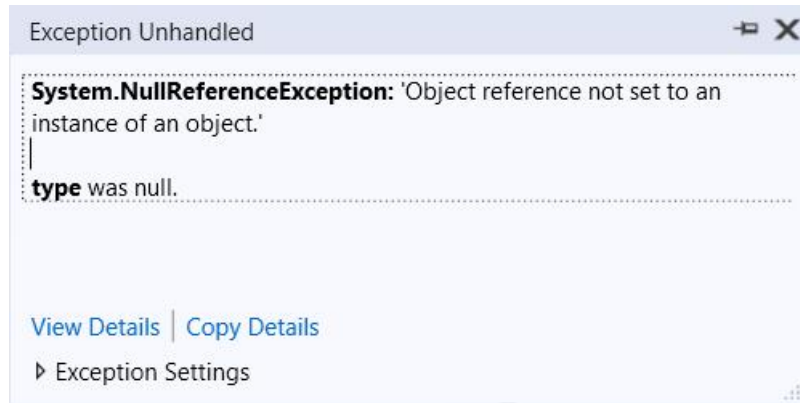
لنعد إلى مثالنا الرئيسي وهو برنامج NumberConverter، وتحديدًا إلى الكلاس ConverterFactory:

ConverterFactory.cs:

```
public class ConverterFactory
{
    public static Converter Create(BaseType baseType, int decimalNumber)
    {
        try
        {
            return (Converter)
                Activator.CreateInstance(
                    Type.GetType($"{SolidSample.{baseType.ToString()}Converter"},
                        new object[] { decimalNumber }));
        }
        catch (Exception)
        {
            return null;
        }
    }
}
```

لاحظ أن الكود أعلاه عند حدوث استثناء في عملية reflection سيقوم بإرجاع null، مما يعني أن الوظيفة Convert لن تشتغل بالشكل المطلوب وستعطي الخطأ الشهير:

System.NullReferenceException: 'Object reference not set to an instance of an object.'



والصورة أعلاه من غير شك قد صادفتها أكثر من مرة :) تمام، الآن الخطأ هنا وارد جداً، وبالتالي إرجاع null ممكن أن يحدث مما قد يوقف برنامجنا، يمكن أن يخطر ببالك حل يتمثل في التحقق من القيمة المرجعة بواسطة ConverterFactory قبل أن تستدعي الوظيفة Convert كما يلي:

```
if (type != null)
{
    result = type.Convert();
}
```

الحل سيعمل من غير شك، لكن تذكر أن التحقق من null فيما يعرف بـ null checks هو انتهاك لمبدأ LSP لأنها مثل التحقق من الأنواع باستعمال الذي رأيناه في الحل الثاني لموضوع Ellipse-Circle Problem.

وبالتالي فإن الحل مرة أخرى هنا، هو أن نفكر في أحد نماذج التصميم Design Patterns التي يمكننا استعمالها لحل المشكل وفي نفس الوقت تفادي انتهاك مبدأ LSP. والحل هذه المرة سيكون مع Null Object Design Pattern.

نموذج Null Object Design Pattern

يسمح لنا هذا النموذج بحل المشاكل المتعلقة بـ Null checks عبر إنشاء نوع جديد يتفاعل في الحالات التي تولد NullReferenceException، كما هو الحال في الكلاس ConverterFactory

التي من الممكن أن تسبب في هذا الاستثناء إذا تم تمرير نوع غير صحيح للأمر GetType، وبالتالي سيتوقف البرنامج جراء هذا الاستثناء. لتفعيل نموذج Null Object يكفي أن ننشئ نوعاً جديداً يرث من الكلاس الرئيسي Converter ثم في حال حصول استثناء نقوم بإرجاع هذا النوع. فيما يلي الكود الخاص بهذا الكلاس والذي سنسميه مثلاً InvalidBaseConverter:

InvalidBaseConverter.cs:

```
public class InvalidBaseConverter : Converter
{
    public InvalidBaseConverter(int decimalNumber):base(decimalNumber)
    {
    }

    public override string Convert()
    {
        return $"This base type is not a valid base.";
    }
}
```

ثم بعد ذلك نأتي إلى الكلاس ConverterFactory وأسفل الجزء catch نقوم بإرجاع instance من هذا النوع الجديد كما يلي:

ConverterFactory.cs:

```
public class ConverterFactory
{
    public static Converter Create(BaseType baseType, int decimalNumber)
    {
        try
        {
            return (Converter)
                Activator.CreateInstance(
                    Type.GetType($"SolidSample.{baseType.ToString()}Converter"),
                    new object[] { decimalNumber });
        }
        catch (Exception)
        {
            return new InvalidBaseConverter(decimalNumber);
        }
    }
}
```

بعد هذا التعديل صار الكود منظماً وقمنا بتطبيق مبدأ LSP بنجاح وتفادينا القيام ب Null checks في برنامجنا.

يمكن لمبدأ LSP أن ينتهك أيضا مع مفهوم الواجهات Interfaces كما ذكرنا في وقت سابق، حينما تكون عندنا واجهة تحتوي على عدة وظائف، بينما الكلاس الذي يقوم بتطبيقها يستعمل فقط بضعة وظائف منها، مما يعني أن استدعاء باقي الوظائف سيولد NotImplementedException. وهذا الأمر يمكن حله بالاعتماد على المبدأ الرابع Interface Segregation Principle والذي سنراه في الفصل المقبل إن شاء الله.

خلاصة مبدأ الاستبدال LSP

مبدأ LSP يسمح لنا بجعل الكود الخاص بنا more cleaner، كما يسمح بجعل الكود الخاص بنا موثوقا فيه لأن تطبيقه يعني أن الأنواع الفرعية subtypes ستصرف كما لو أنها base type، مما يعني أن behavior الخاص بالوظيفة التي تستدعي object من النوع الرئيسي لن يتأثر في حال استبدال هذا object ب object آخر من نوع فرعي.

المبدأ الرابع: مبدأ فصل الواجهات

Interface Segregation Principle

هذا المبدأ من أسهل مبادئ SOLID وهو ينص على أن العميل Client والمقصود به في هذا السياق الكود الذي سيطبق واجهة Interface معينة، ليس ملزماً بأن يقوم بتعريف كافة الوظائف المذكورة فيها، وإنما ينبغي أن يطبق فقط ما يحتاجه، وذلك لتفادي الوقوع في انتهاك لمبدأ Liskov Substitution Principle.

لماذا نحتاج إلى مبدأ فصل الواجهات ISP

كما ذكرنا في الفقرة السابقة فإننا نحتاج إلى مبدأ ISP من أجل أن نتفادي الوقوع في أخطاء منهجية تخرق مبدأ LSP، بحيث نقوم ببناء Interface تحتوي على مجموعة من الوظائف التي من الوارد أن الكود العميل لن يحتاجها كلها، وبالتالي ستبقى عدة وظائف Not Implemented مما سيضر بالبرنامج.

كما أن الواجهات الكبيرة Large Interfaces تضر بجودة التصميم، الذي يشير إلى أن المكونات البرمجية ينبغي أن تكون صغيرة small، مركزة focused، متماسكة Cohesive، وبالتالي الاعتماد على واجهات كبيرة قد لا يحترم هذه المعايير.

متى نحتاج إلى مبدأ فصل الواجهات ISP

حينما تجد أنك جعلت الواجهة كبيرة جداً Large interface فهنا ينبغي أن تفكر في مبدأ Interface Segregation Principle، وحينما تجد بأن بعض الكلاسات تحتوي على وظائف غير مبرمجة Not implemented methods فمن الضروري أن تعتمد هذا المبدأ.

كيف نطبق مبدأ فصل الواجهات ISP

لتطبيق مبدأ فصل الواجهات Interface Segregation Principle، سنقوم بإضافة بعض الخصائص إلى مشروعنا NumberConverter للقيام ببعض العمليات الإضافية كتحويل البيانات الثنائية إلى نصوص، والعكس صحيح، وكذلك تحويل البيانات الست عشرية إلى نصوص والعكس صحيح.

سنأتي إلى مشروعنا NumberConverter ونقوم بإضافة واجهة Interface نسميها IConverter، سنضع فيها كافة الوظائف التي سنحتاج أن نقوم بتطبيقها على مستوى الكلاسات الخاصة بالتحويل.

فيما يلي محتوى الواجهة IConverter:

IConverter.cs:

```
public interface IConverter
{
    string BinaryToText(string binary);
    string TextToBinary(string text);
    string HexadecimalToText(string text);
    string TextToHexadecimal(string text);
}
```

أول ملاحظة لهذه الواجهة أنها غير متماسكة Not cohesive لأنها تحتوي على وظائف مختلفة بعضها يعمل مع النظام الست عشري والبعض الآخر مع النظام الثماني. لكن ما علينا، سنقوم بتطبيق هذه الواجهة على مستوى الفئتين BinaryConverter و HexadecimalConverter، وفيما يلي محتوى الكلاس BinaryConverter:

BinaryConverter.cs:

```
public class BinaryConverter : Converter, IConverter
{
    public BinaryConverter(int decimalNumber)
        : base(decimalNumber)
    {
    }

    public override string Convert()
    {
        return $"The result is: {System.Convert.ToString(DecimalNumber, 2)}";
    }

    public string BinaryToText(string binaryNumber)
    {
        binaryNumber = binaryNumber.Replace(" ", "");

        var list = new List<Byte>();

        for (int i = 0; i < binaryNumber.Length; i += 8)
        {
            String t = binaryNumber.Substring(i, 8);

            list.Add(System.Convert.ToByte(t, 2));
        }

        var result = list.ToArray();

        return Encoding.ASCII.GetString(result);
    }
}
```

```

    }

    public string TextToBinary(string text)
    {
        var bytes = Encoding.ASCII.GetBytes(text);

        return string.Join(" ",
            bytes.Select(byt => System.Convert.ToString(byt, 2).PadLeft(8,
                '0')));
    }

    public string HexadecimalToText(string text)
    {
        throw new NotImplementedException();
    }

    public string TextToHexadecimal(string text)
    {
        throw new NotImplementedException();
    }
}

```

كل الوظائف التي تم تطبيقها من الواجهة IConverter تحتوي على Implementation باستثناء الوظائف HexadecimalToText و TextToHexadecimal، مما يعني أننا خرقنا مبدأ Liskov Substitution، نفس المشكلة سيتكرر مع الكلاس HexadecimalConverter وفيما يلي محتواها:

HexadecimalConverter.cs:

```

public class HexadecimalConverter : Converter, IConverter
{
    public HexadecimalConverter(int decimalNumber) : base(decimalNumber)
    {
    }

    public override string Convert()
    {
        return $"The result is: {DecimalNumber.ToString("X")}";
    }

    public string HexadecimalToText(string text)
    {
        text = text.Replace(" ", "");

        byte[] raw = new byte[text.Length / 2];

        for (int i = 0; i < raw.Length; i++)
        {
            raw[i] = System.Convert.ToByte(text.Substring(i * 2, 2), 16);
        }

        return Encoding.ASCII.GetString(raw); ;
    }

    public string TextToHexadecimal(string text)
    {
        byte[] bytes = Encoding.ASCII.GetBytes(text);
    }
}

```



```

        var hexString = BitConverter.ToString(bytes);

        hexString = hexString.Replace("-", "");

        return hexString;
    }

    public string TextToBinary(string text)
    {
        throw new NotImplementedException();
    }

    public string BinaryToText(string binary)
    {
        throw new NotImplementedException();
    }
}

```

لاحظ أن الوظيفتين TextToBinary و BinaryToText لا تحتويان على أي Implementation مما يعني أننا أيضا على مستوى هذا الكلاس انتهكنا مبدأ LSP. هنا ينبغي أن نفكر في حل مناسب يضمن لنا تفادي الوقوع في انتهاك مبدأ LSP وفي نفس الوقت جعل الواجهة أكثر تماسكا. الحل هنا هو أن نقوم بفصل حالات الواجهة IConverter ليصبح لدينا واجهتين، الأولى خاصة ب BinaryConverter والثانية خاصة ب HexadecimalConverter. وفيما يلي محتوى الواجهة الجديدة التي أسميناها IBinaryConverter:

IBinaryConverter.cs:

```

public interface IBinaryConverter
{
    string BinaryToText(string binary);
    string TextToBinary(string text);
}

```

لاحظ أن الواجهة أصبحت صغيرة، متماسكة ومفهومة بسرعة. الآن سنقوم بتطبيقها بدل استعمال IConverter وبالتالي سنقوم بحذف الكود الزائد الذي يولد NotImplementedException. نفس الأمر سنقوم به مع الواجهة الثانية IHexadecimalConverter والتي محتواها كالاتي:

IHexadecimalConverter.cs:

```

public interface IHexadecimalConverter
{
    string HexadecimalToText(string text);
    string TextToHexadecimal(string text);
}

```

هذه الكيفية نكون قد طبقنا مبدأ ISP بنجاح، وتفادينا أن نفرض على الكود العميل أن يقوم بتعريف وظائف لا يحتاجها.

خلاصة مبدأ فصل الواجهات ISP

هذا المبدأ يخبرنا بأن علينا تقديم واجهات مركزة تخدم حاجة العميل، دون إلزامه بوظائف لا يحتاجها وبالتالي سيتكرها Not implemented مما يؤدي إلى خرق مبدأ LSP وبالتالي حدوث مشاكل في البرنامج من جراء هذا التصميم السيء.

المبدأ الخامس: مبدأ انعكاس التبعية

Dependency Inversion Principle

حتى أختصر عليك فصولاً طويلاً ومحاضرات كثيرة سأقول لك: هذا المبدأ يهدف بالأساس إلى إضعاف ارتباط الكلاسات مع بعضها البعض، بحيث إذا قررنا في المستقبل أن نستبدل كلاس بكلاس أو أن نعدل على كلاس فلن تتأثر الكلاسات الأخرى المرتبطة به. مبدأ DIP مبدأ كثير التردد نظراً لارتباطه بنموذج Dependency Injection Pattern والذي بدوره يعتبر حلاً من الحلول المتاحة لتفعيل مبدأ DIP. لننطلق من الصفر مع المبدأ، سنبدأ بنصه الأصلي الذي يقول:

High-level modules should not depend on low-level modules, both should depend on abstractions.

Abstractions should not depend on details, details should depend on abstractions.

لفهم هذين البندين بالتفصيل سنبدأ بتحرير المصطلحات، ماهو المقصود بالوحدات العليا High-level modules والوحدات الدنيا Low-level modules؟ وماهو المقصود ب abstraction و details؟ علماً أن المفهومين الأخيرين سبق وأن شرحناهما في بداية الكتاب ويمكنك العودة إليهما.

حينما تكون عندنا وحدتان مرتبطتان مع بعضهما البعض، فالوحدة التي تحتاج إلى الأخرى هي الوحدة العليا High-level module، والوحدة الأخرى هي Low-level module. يعني لو عندنا الفئتين التاليتين:

ClassA.cs:

```
public class ClassA
{
    public void MethodA()
    {
        // Do something
    }
}
```

ClassB.cs:

```
public class ClassB
{
    ClassA a;

    public void MethodB()
    {
        a = new ClassA();
        a.MethodA();
    }
}
```

لاحظ أن الكلاس ClassB تحتاج إلى object من الكلاس ClassA أي أن هنالك ترابط بينهما. الكلاس ClassB هي التي تحتاج إلى الارتباط، إذن فهي وحدة عليا High-level module. الكلاس ClassA لا تحتاج إلى أي ارتباط مع الكلاس ClassB، إذن فالكلاس ClassB هي الوحدة الدنيا Low-level module.

النص الأول لمبدأ DIP يقول بأن الوحدات العليا لا ينبغي أن تكون مرتبطة بالوحدات الدنيا، بل كليهما ينبغي أن ترتبطا بالتجريد Abstraction، وذكرنا في أول الكتاب أن التجريد نقصد به الفئات المجردة Abstract classes والواجهات Interfaces. وسنرى كيف نقوم بذلك بالتفصيل إن شاء الله.

النص الثاني لمبدأ DIP يقول بأن التجريد لا ينبغي أن يهتم بالتفاصيل، بل هذه الأخيرة من ينبغي أن تكون مرتبطة بالتجريد، بمعنى أن الواجهات أو الفئات المجردة لا ينبغي أن تكون مرتبطة بكلاسات واقعية Concrete classes، بل العكس هو الذي ينبغي أن يكون.

لماذا نحتاج إلى مبدأ انعكاس التبعية DIP

نحتاج إلى مبدأ DIP من أجل إضعاف الارتباطات بين الكلاسات فيما يعرف ب Loose coupling، لأن الارتباطات القوية بين الكلاسات تؤثر سلباً على مستوى الاختبار خاصة الاختبارات الأحادية Unit Tests، وعلى مستوى الصيانة Maintainability، حيث يصعب صيانة الكود الذي يتوفر على عدة ارتباطات، ناهيك عن مشاكل صعوبة فهم الكود Understandability ومشاكل توسيع الكود وإضافة خصائص جديدة إليه Extensibility، فلكي نتمكن من الاستفادة من كل هذه المزايا علينا أن نجعل كودنا موافقاً لمبدأ Dependency Inversion.

متى نحتاج إلى مبدأ انعكاس التبعية DIP

حينما تجد في مشروعك كلاسات مرتبطة مع بعضها البعض بعيداً عن آليات التجريد abstraction كالواجهات والفئات المجردة.

حينما تجد أن الكلاسات التي تكتبها تأثرت بالارتباطات dependencies لدرجة أنك لم تعد قادرا على كتابة اختبارات أحادية تفي بالغرض.

حينما تجد أن مشروعك مرتبط أكثر بالتفاصيل details بحيث لو فكرت في المستقبل أن تستعمل أنواعا جديدة وجدت الأمر مرهقا جدا.

باختصار، إذا كنت تنوي أن تبني مشروعا قويا متماسكا تشتغل عليه على المدى الطويل أو القصير دون أن تجده يضع أمامك العراقيل، فقم بتصميم مشروعك وفق مبادئ SOLID وعلى رأسها مبدأ Dependency Inversion Principle.

كيف نطبق مبدأ انعكاس التبعية DIP

لنفهم هذا المبدأ بشكل جيد، سنعود إلى مشروعنا، وتحديدًا إلى الكلاس الأساسي NumberConverter وشاهد أعلى الكلاس ستجد أننا صرحنا عن كائنين من النوع Logger و Reader كما يلي:

NumberConverter.cs:

```
public Logger Logger { get; set; } = new Logger();
public Reader Reader { get; set; } = new Reader();
```

طبعا استندسنا هذين النوعين لنتمكن من استعمالهما في الكلاس الحالي، لكن هنالك مشكلة هنا.

الكلاس الحالي NumberConverter أصبح مرتبطا بشكل كبير Tightly coupled بالفئتين Logger و Reader، مما يعني أن عندنا مشاكل في التصميم وأنا انتهكنا مبدأ DIP الذي يقضي بأن الوحدات العليا وفي حالتنا هذه هي الكلاس NumberConverter لا ينبغي أن تكون مرتبطة بالوحدات الدنيا وفي حالتنا هذه هما Logger و Reader، هذا من جهة ومن جهة أخرى ينص نفس المبدأ على أن كل من الطرفين ينبغي أن يكون مرتبطا بالتجريد Abstraction مثل Abstract classes و Interfaces.

لذلك سنقوم بإنشاء الواجهات اللازمة لارتبط بها بدل أن نرتبط بالأنواع الواقعية، والبداية مع الواجهة ILogger وهذا محتواها:

ILogger.cs:

```
public interface ILogger
{
    void Log(string message);
}
```

الآن سنعود إلى الكلاس Logger ونقوم بتطبيق الواجهة أعلاه عليها:

Logger.cs:

```
public class Logger: ILogger
{
    public void Log(string message)
    {
        Console.WriteLine(message);
    }
}
```

نفس الكلام مع الكلاس Reader سننشئ لها واجهة نسميها IReader ونقوم بتطبيقها كما يلي:

IReader.cs:

```
public interface IReader
{
    int ReadInteger();
}
```

Reader.cs:

```
public class Reader:IReader
{
    public int ReadInteger()
    {
        return int.Parse(Console.ReadLine());
    }
}
```

جميل جدا، هكذا نكون قد ربطنا الوحدات الدنيا Low-level modules مع Abstraction، بقي أن نقوم بربط الوحدات العليا High-level module في حالتنا هذه NumberConverter مع Abstraction أيضا.

ثم سنقوم بتمرير الأنواع اللازمة لهذا التجريد عبر مشيد الفئة Constructor ليتم حقنها في الكلاس، وهذا الحقن هو ما يسمى ب Dependency Injection حيث نقوم بحقن الأنواع الواقعية لتعمل مكان الأنواع المجردة في كلاس معين. لندخل الآن إلى الكلاس NumberConverter ونستبدل التصريحين التاليين:

NumberConverter.cs:

```
public Logger Logger { get; set; } = new Logger();
public Reader Reader { get; set; } = new Reader();
```

ونضع مكانهما التصريحين الجديدين اللذان يجعلان الكلاس مرتبطا بالتجريد:

NumberConverter.cs:

```
public ILogger Logger { get; set; }  
public IReader Reader { get; set; }
```

ثم بعد ذلك سنقوم بتمرير قيم هذين الكائنين عبر مشيد الكلاس كما يلي:

NumberConverter.cs:

```
public NumberConverter(ILogger logger, IReader reader)  
{  
    Logger = logger;  
    Reader = reader;  
}
```

بهذه الكيفية نكون قد قطعنا أشواطاً مهمة في تفعيل مبدأ Dependency Inversion Principle عبر تطبيق نموذج Dependency Injection Pattern، وأرجو أن تكون قد أدركت الفرق بينهما. الآن لو أتينا لتجربة الكلاس NumberConverter فنحن مطالبون بتقديم الأنواع الفعلية Concrete types ليتم حقنها على شكل ارتباطات عبر مشيد الكلاس كما يلي:

Program.cs:

```
var converter = new NumberConverter(new Logger(), new Reader());
```

الجميل في العملية، أننا لو أردنا أن نستبدل الكلاس Reader بكلاس آخرى تقرأ البيانات من قاعدة بيانات أو من Web Service أو من جهاز قارئ الباركود أو غيره، فإننا لن نغير الكود الخاص بالكلاس NumberConverter وإنما سنقوم بإنشاء نوع جديد يطبق الواجهة IReader ثم نقوم باستعماله مكان القارئ الأصلي هكذا:

Program.cs:

```
var converter = new NumberConverter(new Logger(), new BarcodeReader());
```

وهذه إحدى مميزات مبدأ DIP.

وتعالوا بنا نستعرضها باستفاضة، وهذه المرة سنقوم بإنشاء كلاس جديد يطبق الواجهة ILogger ودوره هو تخزين الأنشطة التي تحدث على مستوى التطبيق في ملف نصي Text File، لذلك دعنا نسمي الكلاس الجديد TextFileLogger، وهذا هو محتواه:

TextFileLogger.cs:

```
public class TextFileLogger : ILogger
{
    public void Log(string message)
    {
        using (StreamWriter writer = File.AppendText("logFile.txt"))
        {
            writer.WriteLine(message);
        }
    }
}
```

الكلاس باختصار يقوم بتخزين القيمة النصية message في ملف نصي باسم logFile.txt من خلال استعمال StreamWriter الذي يسمح بإنشاء الملفات والكتابة فيها. الآن سنأتي إلى مكان إنشاء كائن من الكلاس NumberConverter ونستبدل Logger ب TextFileLogger كما يلي:

Program.cs:

```
var converter = new NumberConverter(new TextFileLogger(), new Reader());
```

عند تنفيذ البرنامج سيتم إنشاء ملف logFile.txt وسيتم تسجيل بعض الأنشطة فقط وهذا هو محتواه:



لعلك ستلاحظ معي أنه تم تسجيل رسالتين فقط، وذلك بسبب أن Logger نستعمله كمسجل للأنشطة وفي نفس الوقت من أجل عرض المعلومات على شاشة الكونسول مما أحدث خلا برنامجنا.

لذلك سنقوم بإنشاء نسخة جديدة من Logger خاصة بعملية عرض الرسائل على Console، لعمل ذلك تعالوا بنا إلى الكلاس NumberConverter وفي مكان التصريح عن الكائنات سنكتب ما يلي:

NumberConverter.cs:

```
public ILogger Logger { get; set; }
public IReader Reader { get; set; }
public ILogger Writer { get; set; }
```

ثم نقوم بتمرير قيمة الكائن عبر المشيد:

NumberConverter.cs:

```
public NumberConverter(ILogger logger, IReader reader, ILogger writer)
{
    Logger = logger;
    Reader = reader;
    Writer = writer;
}
```

ثم نقوم بتعديل محتوى الوظيفة Convert لتصبح كما يلي:

NumberConverter.cs:

```
public void Convert()
{
    Logger.Log("Program is starting...");

    Writer.Log("Enter the number to convert:");
    DecimalNumber = Reader.ReadInteger();

    Logger.Log($"The number to convert is: {DecimalNumber}");

    Writer.Log("Enter the base type (Ex: 2,8,16):");
    var baseType = (BaseType)Reader.ReadInteger();

    Logger.Log($"The selected base type is: {baseType.ToString()}");

    var type = ConverterFactory.Create(baseType, DecimalNumber);

    Logger.Log($"using the converter: {type.GetType()}");

    string result = type.Convert();

    Writer.Log(result);

    Logger.Log($"The result is: {result} ");

    Logger.Log("Program is ending..");
}
```

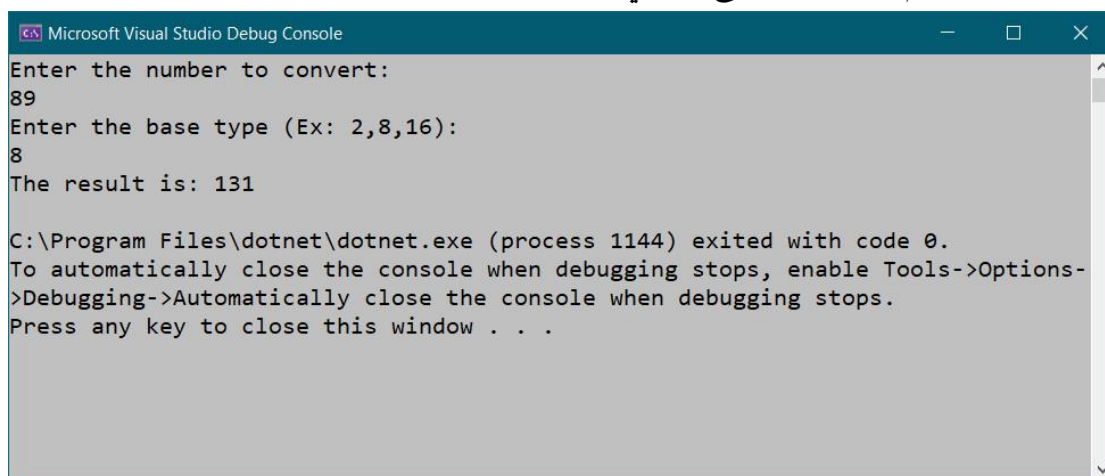
الآن صار عندنا نسختين من نفس النوع Logger، النسخة الأولى اسمها Logger ودورها عمل Logging من أجل تسجيل الأنشطة في ملف نصي، والنسخة الثانية اسمها Writer ودورها طباعة المعلومات على شاشة الكونسول.

الآن كود استدعاء كائن من النوع NumberConverter سيكون كما يلي:

Program.cs:

```
var converter = new NumberConverter(new TextFileLogger(),
                                     new Reader(),
                                     new Logger());
```

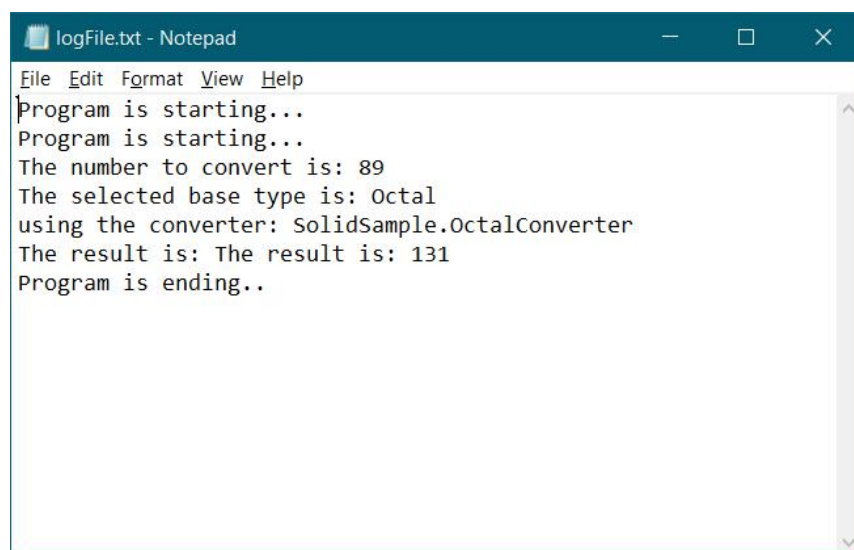
عند التنفيذ سيتم تشغيل البرنامج كما يلي:



```
Microsoft Visual Studio Debug Console
Enter the number to convert:
89
Enter the base type (Ex: 2,8,16):
8
The result is: 131

C:\Program Files\dotnet\dotnet.exe (process 1144) exited with code 0.
To automatically close the console when debugging stops, enable Tools->Options-
>Debugging->Automatically close the console when debugging stops.
Press any key to close this window . . .
```

وبالمقابل سيتم تخزين الأنشطة في ملف نصي باسم logFile مخزن في مجلد bin/Debug/netcoreapp، وهذا هو المحتوى المخزن:



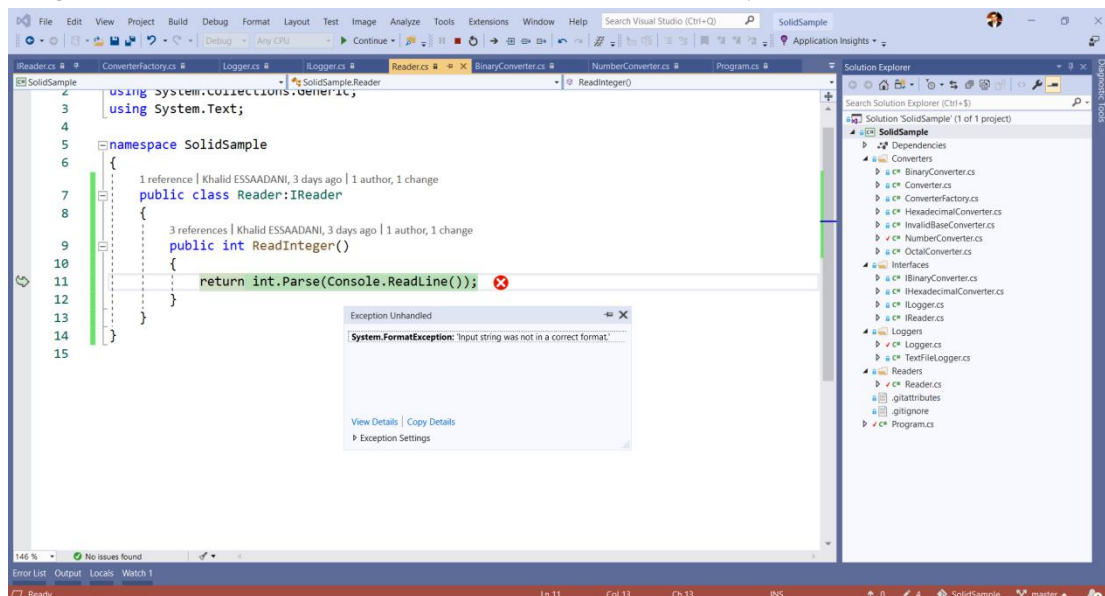
```
logFile.txt - Notepad
File Edit Format View Help
Program is starting...
Program is starting...
The number to convert is: 89
The selected base type is: Octal
using the converter: SolidSample.OctalConverter
The result is: The result is: 131
Program is ending..
```

إلى هنا نكون قد انتهينا من شرح مبادئ SOLID.
بقي تعديل طفيف سنقوم به إن شاء الله قبل ختم هذا الكتاب، وهو الكلاس Reader الذي لو عدنا إليه سنجد محتواه كالآتي:

Reader.cs:

```
public class Reader: IReader
{
    public int ReadInteger()
    {
        return int.Parse(Console.ReadLine());
    }
}
```

الوظيفة ReadInteger وظيفة ساذجة، لأن المسكينة تعتقد أن المستخدم سيدخل حتما قيمة رقمية، لكن ماذا لو خذناها وقام بإدخال قيمة نصية مثلا، الجواب ما ترى أسفله لا ما تسمع:



حصل استثناء من نوع FormatException بسبب أننا أدخلنا قيمة نصية بدل قيمة رقمية فتوقف البرنامج، وهذه نصيحة عابرة: لا تثق أبداً بالمدخلات مهما كان مصدرها، عليك دائما ب Validation.
الآن سنعدل على الكلاس Reader ليمكننا تفادي مثل هذه المشاكل، وهذا هو محتواها الجديد:

Reader.cs:

```
public class Reader : IReader
{
    public Reader(ILogger logger)
    {
        Logger = logger;
    }
}
```

```

public ILogger Logger { get; }

public int ReadInteger()
{
    try
    {
        string value = Console.ReadLine();

        return int.Parse(value);
    }
    catch (Exception)
    {
        Logger.Log("The entered value is invalid.");

        return 0;
    }
}
}

```

الآن صار الكلاس يقتنص الأخطاء وفي حال حصولها سيعرض الرسالة:

The entered value is invalid.

ستلاحظ أن الكلاس Reader أصبح يحتاج إلى النوع ILogger الذي استعملناه من أجل عرض رسالة الخطأ أعلاه في شاشة الكونسول.

الآن لكي يعمل التطبيق بنجاح، سنعود إلى الكلاس Program.cs ونقوم بتغيير صيغة إنشاء object من الكلاس NumberConverter كما يلي:

Program.cs:

```

var consoleLogger = new Logger();

var converter = new NumberConverter(new TextFileLogger(),
                                     new Reader(consoleLogger),
                                     consoleLogger);

```

استعملنا نفس Logger في الكلاس NumberConverter و Reader لأننا نطبع الرسائل في شاشة كونسول واحدة.

خلاصة مبدأ انعكاس التبعية DIP

مبدأ انعكاس التبعية Dependency Inversion Principle يهدف بالأساس إلى تمكيننا من بناء مشاريع قوية قابلة للصيانة Maintainable، قابلة للاختبار Testable، قابلة للتوسيع Extensible، سهلة الفهم عند قراءتها Understandable، وهو مبدأ مهم جداً من مبادئ التصميم والذي يقضي باعتماد آلية التجريد Abstraction بدل التعامل المباشر مع تفاصيل الأنواع Details وذلك بغرض إضعاف الارتباط Loose coupling.

خاتمة

بعد هذه الأشواط المهمة التي قطعناها مع هذا الكتاب المختصر، تمكنا بفضل الله وعونه من التعرف على مبادئ التصميم الخمسة التي يصطلح عليها بـ SOLID Design Principles، وبين الفينة والأخرى كنا ننثر بعض الفوائد حسب المقام، فرأينا بعض نماذج التصميم Design Patterns مثل Factory، Null Object، و Dependency Injection. كل ذلك بشكل عملي مفصل يوضح الفكرة وهدفها وكيفية إنجازها.

بالرغم من كل ذلك، فإن مجهودنا مجهود بشري قد نخطئ فيه وقد نصيب، ومع أننا حاولنا جاهدين أن نخرج الكتاب بأقل الأخطاء الممكنة، فإنه من الوارد جدا أن تكون هنالك أخطاء نتيجة السهو، ضعف التركيز، الاستعجال في الكتابة وغيرها من موجبات حدوث الخطأ. غير أننا نلتمس فيك القارئ الذي يفيد ويستفيد، فلا يمنعك شكرنا على هذا المجهود من نقدنا ومراسلتنا بأخطائنا.

وهذا بريدنا الإلكتروني لاستقبال الرسائل، التعليقات، الاستفسارات:

Khalid_essaadani@hotmail.fr

ثم في الختام، أسأل المولى تبارك وتعالى أن يتقبل منا هذا العمل وأن يجعله خالصا لوجهه الكريم وألا يجعل فيه للنفس ولا للشيطان حظا، وأن يحل عليه بركة من عنده تجعله كتابا نافعا، يهتدي به العاملون في مجال تطوير البرمجيات ويعينهم على بناء أنظمة معلوماتية قوية ومتماسكة.

دام لكم البشر والفرح!

خالد السعداني