كما اشارت الاجوبة السابقة فإن كتاب Clean Code الشهير للكاتب Robert Cecil Martin المعروف ايضا بـ Uncle Bob هو مرجع مهم لكتابة كود قابل للقراءة والتطوير, هنا قمت بتلخيصه لأني لاحظت أن الاجوبة السابقة ليست كاملة بالاضافة الى بعض النقاط الاخرى التي لم يتم ذكرها في الكتاب والتي تسهم بنظافة الكود ايضا:

ملخص كتاب Clean Code

قواعد عامة

- اذا كنت تريد ان تكون سريعا وان تجعل كتابة الكود سهلة اجعله سهل القراءة.
- اتبع قاعدة فتى الكشافة: اترك المكان انظف مما وجدته عليه.
- اختر اسماء ذات دلالة:
  - اسماء تكشف الغرض من المسمى (class, function, variable)
  - اجعل اسماء الـ objects, classes جمل اسمية او اسماء.
  - اجعل اسماء الـ Methods افعال او جمل فعلية.
- اتبع نمط تسمية موحد naming convention.
- اتبع تنسيق (format) موحد لـ (indentations, whitespaces, parentheses, new lines, grouping), انا افضل استخدام اداة لذلك مثل Visual Studio Format, Clang Format.
- يفضل استخدام الـ Polymorphisms على if/else, switch/case لانها تقلل من التفرع branching بالكود بشكل كبير.
- تجنب التكرار (duplication) (do not repeat yourself (التكرار موطن الكثير من الشرور)
- يفضل استخدام الـ exceptions على ارجاع قيم تدل على نوع الخطأ الحاصل.
- غلف سلوك try/catch في functions لكي لا تشوش على هيكلية الكود الاساسية.
- اتبع قواعد Kent Beck الاربع للتصميم البسيط:
  - شغل جميع الـ tests
  - تخلص من التكرار.
  - عبر عن المقصود اثناء البرمجة (classes/functions صغيرة, اسماء ذات دلالة, اتبع المعايير, اكتب tests)
  - قلل من عدد الـ classes/methods لا تعارض مع النقطة السابقة فليس كل class يحتاج لـ interface وليس دائما كتابة data class افضل من object class.

Function Writing

- الـ Funstions يجب ان تقوم بمهمة واحدة فقط وفقط واحدة.
- قلل من عدد الـ parameters الخاصة بـ function قدر المستطاع.
- لا تمرر او ترجع null.

Class Design

- قم باخفاء الـ internal states واجعل فقط الـ methods قالبة للوصول الخارجي.
- في الانظمة التي تميل الى اضافة انماط معطيات data types اكثر من اضافة functions يفضل استخدام الـ OOP بحيث يمكن للـ class ان يحتوي على functions تقوم بعمليات على المعطيات الخاصة بها, أما في الانظمة التي تميل الى اضافة functions اكثر من اضافة انماط معطيات data types فيفضل استخدام classes فقط للمعطيات دون اي عمليات عليها ومن ثم عمليات (functions) على تلك data classes .

- Law of Demeter للتقليل من الارتباط بين الوحدات بحيث كل وحدة تتصل بالوحدات التي لها اتصال المباشر بها فقط.
- يجب اتباع مبدأ Single Responsibility في تصميم الـ class.
- من الافضل رفع درجة الـ cohesion في الـ class (زد من عدد متحولات الـ class المستخدمة في الـ methods الخاصة به), الكتاب ايضا يشير الى ان تحقيق اعلى درجة من الـ cohesion لايمكن الوصول لها ولا ينصح بذلك.
- يجب اتباع مبدأ Open-Close Principle.
- يجب اتباع مبدأ Separation of Concerns في التصميم بشكل عام عن طريق استخدام بعض التقنيات مثل استخدام الـ design patterns و dependency injection.

Comments

- لا تستخدم التعليقات لشرح كود معقد, الكود يجب ان يشرح نفسه, التعليقات تكون لـ:
  - شرح بعض المظاهر التقنية.
  - شرح الـ returned value
  - TODO Comments
  - شرح بعض اجزاء الكود الهامة التي لا تبدو كذلك.
- احذف الكود غير المستخدم ولا تعمل له فقط comment out .

Third-Party Packages

- استخدم thrid-party interfaces فقط داخليا.
- تغليف thrid-party libs افضل من السماحية بالوصول المباشر لها للتقليل من الاعتماد عليها.
- قم بكتابة tests لتلك thrid-party libs.

Tests

- الـ tests مهمة بأهمية المشروع نفسه, اهمال الـ tests يسبب فشل المشروع.
- كود الـ test يجب ان يكون بنظافة كود المشروع.
- كل test يجب ان يختبر مفهوم واحد فقط.
- قلل من الـ asserts في الـ test.
- كلما زادت درجة تغطية الـ tests ازداد الشعور بالامان.
- الـ tests يجب ان تكون سريعة, مستقلة, قابلة للتكرار, سهلة التقييم (نجاح/فشل).

Concurrency

- افصل الكود الخاص بالـ concurrency عن بقية الكود.
- استخدم تقنيات اللغة قبل تطوير خاصتك.
- اجعل القطاعات المتزامنة صغيرة.
- فكر بشروط التوقيف مبكرا واجعلها تعمل.
- اختبر الكود الخاص بالـ threads:
  - اعتبر الاخطاء التي لا يمكن ان تحدث في الكود غير الحاوي على threads كأخطاء محتملة في الـ threads.
  - اجعل الكود غير الحاوي على threads يعمل مسبقا.

- اجعل الكود الـ threads pluggable.
- اجعل الكود الـ threads قابل للتعديل.
- شغل threads اكثر من عدد المعالجات المتوفرة.
- شغل threads على اكثر من منصة.
- حاول تعمد انتاج الاخطاء.

ذلك كان تلخيصي لكتاب Clean Code فيما يلي أيضا بعض النقاط المهمة في هذا السياق:

ان اتباع المفاهيم الهندسية التصميمية والتنفيذية الخاصة بتطوير البرمجيات يؤدي بالضرورة الى كتابة كود اكثر نظافة:

- **مفاهيم تنفيذية:**
  - KISS (Keep It Simple Stupid) تجنب اي تعقيد غير ضروري
  - DRY (Don`t Repeat Yourself) تجنب التكرار كما ورد سابقا.
  - YAGNI (You Aren`t Gonna Need It) لا تقوم بكتابة شيء لظنك انك ستستخدمه مستقبلا
- **مفاهيم تصميمية:**
  - SOLID: هي 5 مفاهيم كالتالي:
    - Single Responsibilty: اي function يجب ان يكون له وظيفة واحدة, اي class يجب ان يكون له مفهوم واحد, كذلك الامر بالنسبة للـ modules, libraries يجب ان تقدم خدمة بسياق واحد.
    - Open-Close Principle: الكيانات (classes, modules, libs) يجب ان تكون قابل للتوسعة لكن ليس للتعديل.
    - Liskov Substitution Principle: استبدال object من class ما بـ object من class اخر مشتق (derived) من الـ class الاول لا يجب أن يؤدي الى خطأ في سلوك البرنامج.
    - Interface Segregation Principle: تقديم interfaces متعددة كل منها يحوي على functions لخدمة واحدة ومن ثم تقديم هذه الـ interfaces حسب الحاجة للـ client (من يحتاج البها) افضل من تقديم inteface تحوي على وظائف لا تهم ذلك الـ client.
    - Dependency Injection: يساعد هذا المفهوم على تقليل اعتماد الـ objects على بعضها فبدل ان يقوم object ما ببناء object اخر داخليا من اجل الحصول منه على خدمة, نمكن ذلك الـ object المعتمد على خدمة من غيره باستقبال تلك الـ objects (عن طريق constructor مثلا) بدل من انشاءها داخليا.
  - استخدام الـ Design Patterns الشهيرة عند امكانية تطبيقها, يقدم كتاب Design Patterns: Elements of Reusable Object-Oriented Software اشهر 23 pattern
  - تجنب الـ Anti-Patterns وهي العادات السيئة لحل المشاكل مثلا كتابة class ليكون مسؤول عن ادارة كل شيء.
  - تجنب الـ code smells وهي الاشياء التي تشعر انها مربكة لقراءة الكود مثلا function له الكثير من الـ parameters أو اسم غريب, أو رقم ثابت أو...
- استخدم static code analzyer مثل Cppcheck, Resharper.