Quels principes SOLID le code de votre API REST respecte-t-il et lesquels ne respecte-t-il pas ?

Concernant les principes SOLID dans notre projet :

Single Responsibility Principle (SRP) : nous avons cherché à le respecter en séparant les responsabilités à travers différents controllers (marque, produit, type produit), en utilisant des DTOs adaptés et des tests dédiés.

Open/Closed Principle (OCP) : afin de faciliter l'extension sans modification, nous avons créé un *Generic Manager*, un `IDataRepository` et des managers avec des tâches bien précises. Même si le *Generic Manager* n'est pas encore utilisé, l'idée est que le backend n'ait plus besoin d'être modifié une fois finalisé, sauf en cas de changement de base de données ou de technologie.

Liskov Substitution Principle (LSP) : nous n'avons pas réussi à appliquer ce principe, car nous n'avons pas trouvé de classe générique suffisamment pertinente pour les controllers (trop abstraite dans notre cas), et nous n'avons pas pu généraliser complètement les managers.

Interface Segregation Principle (ISP) : ce principe a bien été respecté, puisque chaque manager et chaque aspect du projet dispose de ses propres interfaces, ce qui rend le code plus clair et plus modulaire.

Dependency Inversion Principle (DIP) : ce principe n'a pas encore été mis en œuvre. Notre objectif était d'utiliser davantage de classes abstraites pour les managers, les controllers et également dans la partie Blazor, mais cela reste à compléter.

Si vous ne les avez pas appliqués dans votre code, quelles améliorations pourriez-vous mettre en place pour améliorer la qualité du code ?

Nous n'avons pas réussi à mettre en place des tests E2E, mais cela constituerait une amélioration importante pour renforcer la robustesse globale du projet. Nous avons cependant intégré FluentValidation afin de valider les données reçues dans les DTOs de manière déclarative, lisible et testable, tout en affichant des messages explicites lorsqu'il manque une information ou qu'une règle métier n'est pas respectée. Nous avons également mis en place des tests avec Mocks pour les entités Marques et TypeProduit, ainsi que du data seeding afin de disposer de données de référence lorsqu'il n'en existe pas pour les tests. Des tests sur les mappers (DTO ↔ entités) ont été réalisés pour garantir la cohérence des transformations. L'ajout de produits, marques et types de produit à partir des DTOs simplifie par ailleurs la saisie et réduit le risque d'erreurs. Enfin, nous avons commencé à implémenter un filtrage via les requêtes MySQL et à améliorer les contraintes en base de données, afin d'assurer une meilleure fiabilité et intégrité des données.

Si vous avez appliqué des améliorations dans votre code, lesquelles sont-elles et que permettent-elles d'améliorer en termes de qualité ?

Dans notre projet, nous avons déjà appliqué plusieurs améliorations qui contribuent directement à la qualité de l'API.

Tout d'abord, l'utilisation de FluentValidation sur les DTOs nous permet de valider les données de manière claire, déclarative et testable. Cela renforce la robustesse de l'application et fournit aux utilisateurs des messages d'erreurs explicites lorsqu'une règle métier ou une donnée obligatoire n'est pas respectée.

Ensuite, nous avons mis en place des tests unitaires avec mocks pour les entités Marques et TypeProduit, ce qui facilite la détection rapide de régressions et garantit que les règles métiers sont correctement respectées sans dépendre de la base de données.

Nous avons également ajouté du data seeding afin de disposer de données par défaut pour nos scénarios de test. Cela rend les tests plus fiables et reproductibles.

De plus, nous avons intégré des tests pour les mappers (DTO ↔ entités) afin d'assurer la cohérence des transformations et d'éviter des erreurs de conversion entre la couche API et la couche métier.
Enfin, la possibilité de créer des produits, marques et types de produit directement à partir des DTO simplifie la saisie et améliore la maintenabilité du code.
Toutes ces améliorations renforcent la robustesse, la cohérence et la maintenabilité de notre application, et facilitent à la fois l'évolution du projet et sa stabilité à long terme.

Si vous ne les avez pas appliqués dans votre code, quelles améliorations pourriez-vous mettre en place pour améliorer la qualité du code ?

Nous n'avons pas réussi à mettre en place des tests E2E, mais cela constituerait une amélioration importante pour renforcer la robustesse globale du projet. Nous avons cependant intégré FluentValidation afin de valider les données reçues dans les DTOs de manière déclarative, lisible et testable, tout en affichant des messages explicites lorsqu'il manque une information ou qu'une règle métier n'est pas respectée. Nous avons également mis en place des tests avec Mocks pour les entités Marques et TypeProduit, ainsi que du data seeding afin de disposer de données de référence lorsqu'il n'en existe pas pour les tests. Des tests sur les mappers (DTO ↔ entités) ont été réalisés pour garantir la cohérence des transformations. L'ajout de produits, marques et types de produit à partir des DTOs simplifie par ailleurs la saisie et réduit le risque d'erreurs. Enfin, nous avons commencé à implémenter un filtrage via les requêtes MySQL et à améliorer les contraintes en base de données, afin d'assurer une meilleure fiabilité et intégrité des données. (edited)

Nous n'avons pas réussi à implémenter l'usage de DTO dans la partie Blazor, à cause de comment MarqueNavigation, TypeProduitNavigation marchait. Afin de pouvoir quand même présenter quelque chose, nous avons changé ProductController du côté API de ne pas utiliser DTO dans la méthode GetAll(). Mais nous avons laisser dans Marque et TypeProduit pour montrer que les tests marchent correctement, et que ce choix a été pûrement pour pouvoir montrer ce qu'on a pu faire. Le teste de Produits a été modifié lors de "Assert.IsInstanceOfType" de DTO à Produit. A cause de ce choix, nous avons pas pu faire fonctionner les pages Marques et Type Produits justement pour ça, et de ne pas revert la méthode GetAll() de celui-ci. La création d'une marque, et type produit marcheraient à théorie, puisquie leur méthodes GetAll() marche de la même manière que celle de Produits, sauf qu'on utiliserais l'API pour voir que la création a bien marché. L'utilisation de l'attribut [JsonIgnore] aurait été à la création.