📚 Résumé des Design Patterns en C#

🎨 CREATIONAL PATTERNS (Patterns de Création)

1. Abstract Factory - Fabrique de Véhicules

Problème résolu : Comment créer des familles d'objets liés sans spécifier leurs classes concrètes ?

Utilité :

- Permet de créer des véhicules électriques ou essence de manière cohérente
- Garantit que tous les véhicules d'une même famille (électrique ou essence) sont compatibles
- Facilite l'ajout de nouvelles familles de produits (ex: hybride) sans modifier le code client

Exemple concret : Une fabrique électrique crée uniquement des automobiles et scooters électriques, garantissant la cohérence.

---

2. Singleton - Instance Unique du Vendeur

Problème résolu : Comment garantir qu'une classe n'a qu'une seule instance dans toute l'application ?

Utilité :

- Évite la création de multiples instances d'un vendeur
- Partage les données du vendeur dans toute l'application
- Économise la mémoire et garantit la cohérence des données

Exemple concret : Un seul vendeur gère toutes les ventes, ses informations sont accessibles partout.

---

3. Builder - Construction de Liasses

Problème résolu : Comment construire des objets complexes étape par étape ?

Utilité :

- Sépare la construction d'une liasse de sa représentation (HTML ou PDF)
- Permet de créer différents formats (HTML/PDF) avec le même processus de construction
- Facilite l'ajout de nouveaux formats sans modifier le code existant

Exemple concret : Construire une liasse de documents (bon de commande + demande d'immatriculation) en format HTML ou PDF.

---

4. Factory Method - Gestion des Commandes

Problème résolu : Comment déléguer l'instanciation d'objets aux sous-classes ?

Utilité :

- Chaque type de client (comptant/crédit) crée son propre type de commande
- Les règles de validation sont spécifiques à chaque type de commande
- Facilite l'ajout de nouveaux types de clients et de commandes

Exemple concret : Un client crédit crée des commandes avec validation de montant (<5000€), un client comptant accepte tous les montants.

---

5. Prototype - Clonage de Documents

Problème résolu : Comment créer de nouveaux objets en copiant des instances existantes ?

Utilité :

- Évite de recréer des documents vierges à chaque fois
- Clone rapidement une liasse de documents pour chaque nouveau client
- Économise du temps et des ressources lors de la création de documents similaires

Exemple concret : Une liasse vierge (bon de commande, certificat, demande) est clonée et personnalisée pour chaque client.

🏗️ STRUCTURAL PATTERNS (Patterns de Structure)

6. Adapter - Documents HTML/PDF

Problème résolu : Comment faire fonctionner ensemble des interfaces incompatibles ?

Utilité :

- Adapte l'interface complexe d'OutilPdf pour correspondre à l'interface Document
- Permet d'utiliser DocumentPdf de la même manière que DocumentHtml
- Intègre du code legacy ou externe sans le modifier

Exemple concret : L'OutilPdf a des méthodes spécifiques (pdfFixeContenu, pdfPrepareAffichage), l'adaptateur les rend compatibles avec l'interface Document standard.

---

7. Bridge - Formulaires d'Immatriculation

Problème résolu : Comment séparer l'abstraction de son implémentation ?

Utilité :

- Sépare la logique métier (validation France/Luxembourg) de l'implémentation technique (HTML/Applet)
- Permet de combiner n'importe quel pays avec n'importe quelle technologie
- Évite une explosion combinatoire de classes (4 classes au lieu de 4 = France-HTML, France-Applet, Luxembourg-HTML, Luxembourg-Applet)

Exemple concret : Les règles de validation des plaques françaises peuvent s'afficher en HTML ou Applet sans dupliquer le code.

---

8. Composite - Composants Graphiques

Problème résolu : Comment traiter uniformément des objets individuels et des compositions d'objets ?

Utilité :

- Permet de créer des structures arborescentes (groupes contenant des groupes)
- Traite les éléments simples (Ellipse, Rectangle) et les groupes de la même manière
- Facilite la gestion de hiérarchies complexes

Exemple concret : Une voiture est un groupe contenant des roues et un châssis, la scène est un groupe contenant la voiture, la route et le soleil. Tout s'affiche et se dessine de manière uniforme.

---

9. Decorator - Décoration de Véhicules

Problème résolu : Comment ajouter dynamiquement des fonctionnalités à un objet ?

Utilité :

- Ajoute des informations (modèle, marque) à une vue de véhicule sans modifier sa classe
- Permet de combiner plusieurs décorateurs (modèle + marque)
- Plus flexible que l'héritage pour ajouter des fonctionnalités

Exemple concret : Une vue simple de véhicule peut être décorée avec le modèle, puis avec la marque, créant ainsi une vue catalogue complète.

🎯 Pourquoi utiliser ces patterns ?

Avantages généraux :

1. **Réutilisabilité** : Code réutilisable et maintenable
2. **Flexibilité** : Facile d'ajouter de nouvelles fonctionnalités

3. **Maintenabilité** : Code mieux organisé et plus facile à comprendre
4. **Évolutivité** : Facilite l'extension sans modifier le code existant
5. **Communication** : Vocabulaire commun entre développeurs

Quand les utiliser ?

- **Creational** : Quand la création d'objets devient complexe
- **Structural** : Quand vous devez gérer des relations entre objets
- **Behavioral** : Quand vous devez gérer la communication entre objets

Principe SOLID respecté :

- **S**ingle Responsibility : Chaque classe a une responsabilité unique
- **O**pen/Closed : Ouvert à l'extension, fermé à la modification
- **L**iskov Substitution : Les sous-classes peuvent remplacer leurs classes parentes
- **I**nterface Segregation : Interfaces spécifiques plutôt que génériques
- **D**ependency Inversion : Dépendre des abstractions, pas des implémentations

---

📊 Tableau récapitulatif

Pattern	Catégorie	Utilité principale	Exemple du projet
Abstract Factory	Creational	Créer des familles d'objets	Véhicules électriques/essence
Singleton	Creational	Instance unique	Vendeur unique
Builder	Creational	Construction complexe	Liasses HTML/PDF
Factory Method	Creational	Déléguer la création	Commandes comptant/crédit
Prototype	Creational	Cloner des objets	Documents vierges
Adapter	Structural	Adapter des interfaces	OutilPdf → Document
Bridge	Structural	Séparer abstraction/implémentation	Pays × Technologie
Composite	Structural	Structure arborescente	Groupes graphiques
Decorator	Structural	Ajouter des fonctionnalités	Vue véhicule enrichie

---

💡 Conclusion

Ces patterns sont des **solutions éprouvées** à des problèmes récurrents en programmation orientée objet. Ils rendent votre code plus **propre**, **flexible** et **maintenable**. Dans le contexte automobile de ce projet, ils permettent de gérer efficacement la création de véhicules, de documents, de commandes et d'interfaces utilisateur de manière professionnelle et évolutive.