Rapport Approfondi sur la Programmation Orientée Objet (POO) en MQL5

Chapitre 1 : Document d'Information

1.0 Introduction et Résumé Exécutif

L'adoption de la Programmation Orientée Objet (POO) représente une évolution stratégique majeure pour les développeurs de systèmes de trading automatisés en MQL5. Face à la complexité croissante des stratégies, la POO offre un cadre structuré qui dépasse les limites de la programmation procédurale. En modélisant les concepts de trading comme des objets autonomes, la POO facilite la création de code modulaire, où des blocs logiques peuvent être développés et débogués indépendamment. Cette approche favorise une réutilisation massive du code, améliore la fiabilité et simplifie radicalement la maintenance, accélérant ainsi le développement d'Expert Advisors (EA) complexes et robustes, capables de s'adapter avec agilité aux évolutions du marché.

Résumé Exécutif

Ce rapport explore en détail l'application de la POO en MQL5. Voici les points essentiels à retenir :

- **Modélisation et Réutilisation :** La POO permet de modéliser des problèmes complexes en créant des "types de données abstraits" (objets) qui encapsulent des données et des comportements. Cette approche facilite grandement la réutilisation du code, améliorant ainsi la fiabilité, la flexibilité et la maintenabilité des applications de trading.
- **Les Trois Piliers Fondamentaux :** Le paradigme repose sur trois concepts clés : l'**encapsulation** (masquer les détails d'implémentation), l'**héritage** (créer de nouvelles classes à partir de classes existantes) et le **polymorphisme** (permettre à des objets différents de répondre de manière unique à un même appel de fonction). La maîtrise de ces piliers est essentielle pour construire des systèmes flexibles.
- **La Classe comme Blueprint :** Une classe est un "plan" ou un "blueprint" qui définit la structure et le comportement d'un objet. À partir d'une seule classe, de multiples objets (instances) peuvent être créés, chacun avec son propre état interne mais partageant la même logique, à l'image d'un plan de voiture utilisé pour fabriquer de nombreuses voitures identiques.
- **Les Design Patterns pour la Maturité :** Au-delà des bases, les "design patterns" (patrons de conception) offrent des solutions éprouvées et réutilisables à des problèmes de conception logicielle courants. Leur utilisation en MQL5 témoigne d'une approche mature du développement, permettant de créer des architectures de trading complexes, robustes et élégantes.

Cette analyse détaillée des concepts fondamentaux et de leur mise en œuvre pratique vous fournira les clés pour exploiter pleinement la puissance de la POO dans vos projets MQL5.

1.1 Les Fondements de la POO en MQL5

Avant d'aborder les aspects pratiques de la programmation d'Expert Advisors, il est impératif de solidifier sa compréhension des principes fondamentaux qui sous-tendent la Programmation Orientée Objet. Cette section décompose les concepts théoriques qui constituent le socle de la

POO et explique pourquoi ce paradigme est si bien adapté au développement de systèmes de trading sophistiqués en MQL5.

1.1.1 Qu'est-ce que la Programmation Orientée Objet ?

La Programmation Orientée Objet (POO) est un paradigme qui se concentre principalement sur les données, où les données et le comportement sont liés de manière inséparable. Elle s'appuie sur la modélisation de concepts du monde réel sous forme d'abstractions informatiques appelées **objets**. Chaque objet peut être considéré comme un "mini-programme" responsable de son propre état et de son propre comportement.

L'analogie la plus simple pour comprendre ce concept est celle de la construction d'une voiture.

- Le **plan** de la voiture est la **classe**. C'est un document de conception qui définit les composants (nombre de roues, type de moteur), les états possibles (vitesse, vitesse engagée) et les actions (accélérer, changer de vitesse, tourner).
- La **voiture** fabriquée à partir de ce plan est l'**objet**. C'est une instance concrète de la classe. On peut fabriquer de nombreuses voitures (objets) à partir du même plan (classe), et chaque voiture aura son propre état (sa propre vitesse, sa propre couleur) tout en partageant les mêmes fonctionnalités de base.

En MQL5, une classe est donc le "blueprint" qui définit les variables et les fonctions d'un objet que l'on souhaite créer, rendant le code plus organisé et intuitif.

1.1.2 Avantages de la POO par rapport à la Programmation Procédurale

L'approche procédurale classique, bien que fonctionnelle pour des tâches simples, présente des inconvénients significatifs lors du développement de systèmes complexes. La POO a été conçue pour surmonter ces obstacles.

Critère	Programmation Procédurale	Programmation Orientée Objet (POO)
Modélisation des problèmes	Divise un problème principal en sous-problèmes plus simples, résolus par des fonctions. Rend la modélisation difficile.	Conçoit les applications à un niveau conceptuel, en modélisant des entités du monde réel sous forme d'objets, ce qui simplifie la conception.
Réutilisation du code	Rend la réutilisation du code difficile. Chaque nouvelle application nécessite souvent de réécrire une logique similaire.	Facilite grandement la réutilisation du code grâce à l'héritage et à la composition, considérée comme la solution à de nombreux problèmes de développement.
Coût, fiabilité, flexibilité et maintenance	La faible réutilisabilité du code a un impact négatif sur les coûts, la fiabilité, la flexibilité et la maintenance.	Bénéficie de la réutilisation du code, d'une meilleure fiabilité, d'une plus grande flexibilité et d'une maintenance simplifiée.

L'exemple des chandeliers IsHammer() et IsSpinningTop() illustre parfaitement ce tableau. L'approche procédurale utilise des fonctions indépendantes qui requièrent de passer les quatre

prix (OHLC) à chaque appel, tandis que l'approche POO crée un objet Candlestick qui encapsule ces données, rendant le code plus propre et la logique plus intuitive.

1.1.3 Les Piliers de la POO

La puissance de la POO repose sur trois piliers conceptuels qui, ensemble, permettent de construire des applications robustes et flexibles.

- **Encapsulation**
  - **Définition :** L'encapsulation est le mécanisme qui combine les données (variables) et les fonctions (méthodes) qui opèrent sur ces données au sein d'une même entité (un objet), tout en cachant les détails de l'implémentation interne à l'utilisateur externe. En MQL5, cela est géré par trois niveaux d'accès :
    - public : Les membres sont accessibles depuis n'importe quelle partie du programme.
    - protected : Les membres ne sont accessibles qu'au sein de la classe elle-même et de ses classes dérivées (héritiers).
    - private : Les membres ne sont accessibles qu'au sein de la classe elle-même. C'est le niveau d'accès par défaut.
  - **Exemple :** Dans la classe TradingSettings, les variables lots, stopLoss et takeProfit sont déclarées comme private. Elles ne peuvent pas être modifiées directement de l'extérieur. Pour interagir avec elles, on doit utiliser les méthodes public comme SetLots() et GetLots(). Cela protège les données d'une modification accidentelle et garantit que les changements sont effectués de manière contrôlée.
- **Héritage**
  - **Définition :** L'héritage est la capacité de créer une nouvelle classe (classe dérivée) à partir d'une classe existante (classe de base). La classe dérivée "hérite" des membres publics et protégés de la classe de base, ce qui favorise une réutilisation maximale du code.
  - **Exemple :** La documentation MQL5 illustre cela avec des formes géométriques. Une classe de base CShape est créée avec des propriétés communes à toutes les formes (type, coordonnées). Ensuite, les classes CCircle et CSquare sont créées en héritant de CShape. Elles héritent des propriétés de CShape, y ajoutent leurs propres membres spécifiques (m_radius pour CCircle, m_square_side pour CSquare) et initialisent le type de la forme via leur constructeur (par exemple, m_type=1 pour CCircle), une information cruciale qui sera utilisée plus tard dans l'exemple du polymorphisme. Ainsi, un cercle *est une* forme, mais avec des attributs et une identification supplémentaires.
- **Polymorphisme**
  - **Définition :** Le polymorphisme (qui signifie "plusieurs formes") est la capacité pour des objets de classes différentes, mais liées par l'héritage, de répondre différemment à l'appel d'une même fonction. Cela permet de créer des mécanismes universels capables de traiter des objets variés de manière uniforme.

- **Exemple :** Dans l'exemple des formes, la classe de base CShape définit une fonction virtuelle GetArea(). Chaque classe dérivée, CCircle et CSquare, redéfinit cette fonction pour implémenter son propre calcul de surface. CCircle utilise la formule 3.14 * r² tandis que CSquare utilise côté * côté. On peut alors parcourir un tableau de pointeurs CShape et appeler GetArea() sur chaque objet. Le système saura automatiquement quelle version de la fonction exécuter en fonction du type réel de l'objet (cercle ou carré), démontrant ainsi le polymorphisme en action.

Ces trois piliers constituent la fondation sur laquelle des applications MQL5 complexes, robustes et élégantes sont construites.

1.2 Implémentation Pratique en MQL5

Après avoir exploré la théorie, il est temps de voir comment ces concepts prennent vie dans le code MQL5. Cette section se concentre sur la syntaxe et les mécanismes concrets pour définir des classes, créer des objets et les utiliser pour construire la logique d'un Expert Advisor.

1.2.1 Définir une Classe

En MQL5, une classe est définie à l'aide du mot-clé class, suivi du nom de la classe et d'accolades contenant ses membres. La structure de base inclut des sections pour les niveaux d'accès private, protected et public.

MetaQuotes suggère des conventions de nommage pour la clarté du code :

- Les noms de classe commencent par un C majuscule (ex. CPerson).
- Les membres protégés sont préfixés par m_ (ex. m_first_name).

Voici un exemple "Hello World" de classe en MQL5, la classe CPerson, qui illustre cette structure de base :

```
//+------------------------------------------------------------------+
//| CPerson Class
//| Un exemple simple de la structure d'une classe en MQL5.
//+------------------------------------------------------------------+
class CPerson
{
protected:
   // Variables membres (données) protégées.
   // Accessibles uniquement par cette classe et ses dérivées.
   string      m_first_name;
   string      m_surname;
   datetime    m_birth;
```


```
public:
    // Méthodes (fonctions) publiques.
    // Accessibles depuis n'importe où dans le code.

    //--- Constructeur et destructeur
    CPerson(void);
    ~CPerson(void);

    //--- Méthodes "Getter" pour récupérer les données
    string      GetFirstName(void);
    string      GetSurname(void);
    datetime    GetBirth(void);

    //--- Méthodes "Setter" pour modifier les données
    void        SetFirstName(string first_name);
    void        SetSurname(string surname);
    void        SetBirth(datetime birth);
};
```

1.2.2 Le Constructeur et le Destructeur

Deux fonctions membres sont spéciales dans une classe :

- **Le Constructeur :** C'est une fonction qui porte exactement le même nom que la classe. Elle est appelée automatiquement lors de la création (instanciation) d'un objet. Son rôle principal est d'initialiser les variables membres de l'objet à des valeurs par défaut ou spécifiques.
- **Le Destructeur :** C'est une fonction dont le nom est celui de la classe précédé d'un tilde (~). Elle est appelée automatiquement lorsque l'objet est détruit (par exemple, à la fin de la portée d'une fonction ou via l'opérateur delete). Elle est utilisée pour libérer les ressources allouées par l'objet, comme la mémoire.

```
// Constructeur de la classe CPerson
CPerson::CPerson(void)
{
    Alert("Hello world! Je suis exécuté lorsqu'un objet CPerson est créé !");
```


```
}


// Destructeur de la classe CPerson
CPerson::~CPerson(void)
{
   Alert("Goodbye world! Je suis exécuté lorsque l'objet est détruit !");
}


// Constructeur de la classe CSimpleRandom
// Il initialise les objets membres (m_brain, m_evolution, etc.)
CSimpleRandom::CSimpleRandom(int stop_loss, int take_profit, double lot_size, ENUM_LIFE_EA time_life)
{
   // ... logique d'initialisation ...
   m_brain = new CBrain(...);
   m_evolution = new CEvolution(DO_NOTHING);
   // ...
}
```

1.2.3 Création et Utilisation d'Objets

Créer un objet, c'est créer une **instance** d'une classe. Une fois qu'un objet est créé, vous pouvez interagir avec ses membres public (variables et méthodes) en utilisant l'opérateur point (.).

L'exemple suivant, inspiré du tutoriel vidéo "Toolkit for Traders", montre la création de deux objets distincts (shinyNewCar et reallyFastCar) à partir d'une même classe CCar. Chaque objet est indépendant et peut être manipulé séparément.

```
// Supposons qu'une classe CCar a été définie précédemment.

// 1. Déclaration et création de deux objets de type CCar.
// Le constructeur de CCar est appelé automatiquement pour chacun.
CCar shinyNewCar;
CCar reallyFastCar;

// 2. Utilisation des méthodes publiques des objets.
```

```
// On accède aux méthodes avec l'opérateur point (.).
shinyNewCar.speedUp();
reallyFastCar.changeGear(2); // Passe à la deuxième vitesse.
```

1.2.4 Exemple Concret : Un Expert Advisor Orienté Objet

L'article "Another MQL5 OOP Class" propose une approche de conception pour l'EA ExpertSimpleRandom.mq5 en le pensant comme une "créature vivante". Cette analogie décompose le système en composants logiques, chacun représenté par une classe :

- CBrain : Le cerveau, contenant les données de configuration en lecture seule (StopLoss, TakeProfit, etc.).
- CEvolution : L'évolution, contenant des informations temporelles comme l'état actuel du robot (BUY, SELL, DO_NOTHING) et l'historique des opérations.
- CGraphic : Le graphique, représentant les informations du marché sur lequel le robot opère.

L'idée de trading sous-jacente est que "les courtes tendances des marchés volatils sont quasi-aléatoires".

La classe principale, CSimpleRandom, agit comme un chef d'orchestre. Elle **compose** ces différents objets pour créer le système complet. Cette architecture illustre un principe fondamental de la POO avancée : la **composition plutôt que l'héritage**. La classe CSimpleRandom n'hérite pas d'un cerveau ou d'une évolution ; elle *est composée* de ces objets. Cette approche favorise la flexibilité et la modularité, permettant de construire des systèmes complexes à partir de composants plus simples et indépendants.

```
// Extrait de la classe CSimpleRandom montrant la composition d'objets
class CSimpleRandom
{
protected:
   // Pointeurs vers les objets qui composent l'EA
   CBrain*         m_brain;
   CEvolution*     m_evolution;
   CGraphic*       m_graphic;
   CTrade*         m_trade;
   CPositionInfo*  m_positionInfo;

public:
   // Le constructeur crée les objets composants
   CSimpleRandom(int stop_loss, int take_profit, double lot_size, ENUM_LIFE_EA time_life);
```

```
    // La méthode Go contient la logique de trading principale
    bool Go(double ask, double bid);
};

// Extrait de la méthode Go, montrant l'interaction entre objets
bool CSimpleRandom::Go(double ask, double bid)
{
    // ...
    // Demande au 'cerveau' de fournir un nombre aléatoire pour la décision.
    // Notez l'utilisation de '->' car m_brain est un pointeur.
    int coin = m_brain->GetRandomNumber(0, 1);

    // m_positionInfo est aussi un pointeur vers un objet qui gère les positions ouvertes.
    if (!m_positionInfo->Select(_Symbol))
    {
        // Aucune position ouverte, nous mettons à jour l'état de l'objet 'évolution'.
        if (coin == 0) { GetEvolution()->SetStatus(BUY); }
        else { GetEvolution()->SetStatus(SELL); }
    }
    // ...
}
```

La maîtrise de ces implémentations pratiques est la clé pour transformer les concepts théoriques de la POO en systèmes de trading MQL5 fonctionnels et bien structurés.

1.3 Concepts Avancés et Design Patterns

Une fois les fondements de la POO maîtrisés, l'étape suivante consiste à explorer des solutions architecturales de plus haut niveau. Les "design patterns" (ou patrons de conception) sont des solutions réutilisables, éprouvées et documentées pour des problèmes de conception logicielle qui surviennent fréquemment. Ils ne sont pas des algorithmes spécifiques mais plutôt des schémas directeurs pour structurer le code afin de le rendre plus flexible, robuste et maintenable.

1.3.1 Introduction aux Design Patterns

Un design pattern est un modèle de conception qui décrit une solution à un problème de conception récurrent dans un contexte donné. L'article "Design Patterns in software development and MQL5" classe ces patterns en trois catégories principales :

- **Création :** Concernent les mécanismes de création d'objets, en essayant de créer des objets de manière adaptée à la situation.
- **Structure :** Expliquent comment assembler des objets et des classes pour former des structures plus grandes, tout en gardant ces structures flexibles et efficaces.
- **Comportement :** Se concentrent sur les algorithmes et l'attribution des responsabilités entre les objets. Ce rapport se focalise sur ce type de patterns.

1.3.2 Analyse des Patterns Comportementaux

Voici une analyse de cinq patterns comportementaux et de leur application potentielle.

- **Chain of Responsibility (Chaîne de Responsabilité)**
    - **Objectif :** Découple l'émetteur d'une requête de son récepteur en donnant à plus d'un objet la possibilité de traiter la requête.
    - **Problème Résolu :** Évite de lier directement l'émetteur d'une requête à son destinataire. Utile lorsque plusieurs objets peuvent traiter une requête et que le gestionnaire approprié n'est pas connu à l'avance, ou lorsqu'un ensemble de gestionnaires peut être spécifié dynamiquement.
    - **Structure :**
        - Client : Initie la requête.
        - Handler : Définit une interface pour traiter les requêtes et peut implémenter un lien vers le successeur dans la chaîne.
        - ConcreteHandler : Traite la requête dont il est responsable. S'il ne peut pas la traiter, il la transmet à son successeur.
    - ***Exemple d'Application en MQL5 :*** Dans un EA, une chaîne pourrait traiter un nouvel événement de tick. Le premier Handler vérifie une condition de breakout, le second une condition de croisement de moyennes mobiles, et le troisième une condition de surachat/survente. La requête (le tick) passe le long de la chaîne jusqu'à ce qu'un Handler la traite.
- **Command (Commande)**
    - **Objectif :** Encapsule une requête en tant qu'objet, permettant ainsi de paramétrer les clients avec différentes requêtes, de mettre en file d'attente ou de journaliser les requêtes, et de prendre en charge les opérations annulables (undo).
    - **Problème Résolu :** Découple l'objet qui invoque une opération de celui qui sait comment l'exécuter. Utile pour des actions comme "exécuter à des moments différents", "mettre en file d'attente", ou pour implémenter un système de journalisation des changements ou une fonctionnalité "undo".
    - **Structure :**

        - Client : Crée un objet ConcreteCommand et définit son Receiver.
        - Command : Déclare une interface pour exécuter une opération.
        - ConcreteCommand : Lie un Receiver à une action et implémente l'exécution.
        - Invoker : Reçoit la commande pour exécuter la requête.
        - Receiver : Sait comment effectuer les opérations associées à l'exécution de la requête.
    - ***Exemple d'Application en MQL5 :*** Dans un panneau de trading, chaque bouton (Acheter, Vendre, Clôturer Tout) peut être un objet Command. L'objet Invoker (le panneau) exécute la commande sans connaître les détails de l'action, qui sont encapsulés dans l'objet Receiver (un module de trading).

- **Interpreter (Interpréteur)**
    - **Objectif :** Étant donné un langage, définit une représentation de sa grammaire ainsi qu'un interpréteur qui utilise cette représentation pour interpréter les phrases du langage.
    - **Problème Résolu :** Utile lorsqu'il existe un langage à interpréter et que l'on peut représenter les phrases de ce langage sous forme d'arbres de syntaxe abstraits. Idéal pour des grammaires simples, car des grammaires complexes peuvent rendre la hiérarchie de classes difficile à gérer.
    - **Structure :**
        - Client : Construit l'arbre de syntaxe abstrait et invoque l'opération d'interprétation.
        - Context : Contient des informations globales pour l'interpréteur.
        - AbstractExpression : Déclare l'opération d'interprétation abstraite.
        - TerminalExpression / NonterminalExpression : Implémentent l'opération d'interprétation pour les symboles terminaux et non terminaux de la grammaire.
    - ***Exemple d'Application en MQL5 :*** Ce pattern pourrait être utilisé pour créer un mini-langage de règles de trading. Un utilisateur pourrait écrire une règle simple comme "MA(10) > MA(50) ET RSI(14) < 30", et l'interpréteur analyserait cette chaîne pour exécuter la logique de trading correspondante.

- **Iterator (Itérateur)**
    - **Objectif :** Fournit un moyen d'accéder séquentiellement aux éléments d'un objet agrégé (comme une liste ou un tableau) sans exposer sa représentation sous-jacente.
    - **Problème Résolu :** Permet de parcourir une collection d'objets sans avoir à connaître sa structure interne (tableau, liste, etc.). Supporte plusieurs parcours

simultanés sur la même collection et fournit une interface uniforme pour parcourir différentes structures d'agrégats.

- **Structure :**
  - Iterator : Définit une interface pour accéder et parcourir les éléments.
  - ConcreteIterator : Implémente l'interface Iterator et garde la trace de la position actuelle dans le parcours.
  - Aggregate : Définit une interface pour créer un objet Iterator.
  - ConcreteAggregate : Implémente l'interface de création de l'itérateur pour retourner une instance du ConcreteIterator approprié.
- ***Exemple d'Application en MQL5 :*** Un Iterator pourrait être utilisé pour parcourir une collection d'ordres ouverts ou d'indicateurs personnalisés sans exposer la structure de stockage sous-jacente (un tableau ou une liste). Cela permet d'écrire une logique de parcours générique qui fonctionne sur différentes collections.

- **Mediator (Médiateur)**
  - **Objectif :** Définit un objet encapsulé qui dicte comment un ensemble d'objets interagissent. Il favorise un couplage faible en empêchant les objets de se référer les uns aux autres explicitement.
  - **Problème Résolu :** Réduit la complexité lorsque de nombreux objets communiquent entre eux de manière complexe et mal structurée ("spaghetti code"). Centralise le comportement de communication, ce qui le rend plus facile à comprendre et à maintenir.
  - **Structure :**
    - Mediator : Définit une interface pour communiquer avec les objets Colleague.
    - ConcreteMediator : Implémente le comportement de coopération en coordonnant les objets Colleague.
    - Colleague classes : Chaque classe "collègue" connaît son objet Mediator et communique avec lui plutôt qu'avec les autres collègues directement.
  - ***Exemple d'Application en MQL5 :*** Un Mediator pourrait gérer les interactions complexes entre un module de gestion des risques, un module d'analyse de marché et un module d'exécution d'ordres. Au lieu qu'ils communiquent tous entre eux, ils informent le Mediator qui orchestre ensuite les actions appropriées.

L'application de ces concepts et patrons de conception avancés permet aux développeurs MQL5 de transcender la simple fonctionnalité pour créer des systèmes de trading qui sont non seulement performants, mais aussi véritablement élégants, robustes et évolutifs.

Chapitre 2 : Guide d'Étude

2.0 Introduction

Ce chapitre est conçu comme un outil essentiel pour vous aider à consolider votre compréhension et à tester les connaissances acquises sur la Programmation Orientée Objet en MQL5. En vous engageant activement avec les questions, les sujets de réflexion et le glossaire, vous renforcerez votre maîtrise des concepts et serez mieux préparé à les appliquer dans vos propres projets de trading automatisé.

2.1 Quiz de Connaissances (Réponses Courtes)

Questions

1. Quelle est la différence fondamentale entre une classe et un objet en POO, selon l'analogie de la voiture ?
2. Nommez les trois niveaux d'accès (encapsulation) en MQL5 et décrivez brièvement qui peut accéder aux membres pour chaque niveau.
3. Quel est le rôle d'une fonction constructeur dans une classe MQL5 ?
4. Expliquez le concept d'héritage en utilisant l'exemple des classes CShape et CCircle.
5. Quelle est l'utilité du pattern "Iterator" (Itérateur) selon la description fournie ?
6. Quel est le principal inconvénient de la programmation procédurale en matière de réutilisation du code ?
7. Quel est l'avantage principal du polymorphisme pour le développement de systèmes de trading ?
8. Quel est le rôle d'une fonction destructeur et quand est-elle appelée ?
9. Quel problème le pattern "Command" (Commande) résout-il en encapsulant une requête en tant qu'objet ?
10. Quelle est la convention de nommage recommandée par MetaQuotes pour le nom d'une classe en MQL5 ?

Corrigé

1. Selon l'analogie de la voiture, la **classe** est le "plan" (blueprint) qui définit comment une voiture doit être construite et ce qu'elle peut faire. L'**objet** est la voiture réelle fabriquée à partir de ce plan ; c'est une instance concrète de la classe.
2. Les trois niveaux d'accès sont :
    - public : Accessible depuis n'importe où dans le programme.
    - protected : Accessible uniquement depuis la classe elle-même et ses classes dérivées (héritiers).
    - private : Accessible uniquement depuis la classe elle-même.

3. Le rôle d'un constructeur est d'initialiser les membres d'un objet au moment de sa création. C'est une fonction spéciale qui porte le même nom que la classe et qui est appelée automatiquement.
4. L'héritage permet de créer une classe dérivée (CCircle) à partir d'une classe de base (CShape). CCircle hérite des propriétés communes de CShape (comme les coordonnées) et y ajoute ses propres attributs spécifiques (comme le rayon), favorisant ainsi la réutilisation du code.
5. Le pattern "Iterator" permet d'accéder séquentiellement aux éléments d'un objet agrégé (comme une liste) sans avoir besoin d'exposer sa structure interne. Il fournit une interface uniforme pour parcourir différentes collections.
6. La programmation procédurale rend la réutilisation du code difficile, ce qui a un impact négatif sur les coûts de développement, la fiabilité, la flexibilité et la maintenance du logiciel.
7. Le polymorphisme permet de créer des mécanismes universels. Des objets de différentes classes (par exemple, différentes stratégies de trading héritant d'une classe de base) peuvent répondre de manière unique à un même appel de fonction (par exemple, ExecuteTrade()), ce qui rend le code plus flexible et extensible.
8. Un destructeur est une fonction appelée automatiquement lorsqu'un objet est détruit. Son rôle est de libérer les ressources allouées par l'objet pendant sa durée de vie, comme la mémoire dynamique.
9. Le pattern "Command" permet de découpler l'objet qui invoque une opération de celui qui l'exécute. Cela permet de mettre des requêtes en file d'attente, de les journaliser ou de supporter des opérations annulables (undo).
10. La convention de nommage recommandée par MetaQuotes pour une classe est de la faire commencer par une lettre C majuscule, par exemple CPerson ou CSimpleRandom.

2.2 Sujets de Réflexion (Format Essai)

1. Analysez les arguments pour et contre l'idée que la POO augmente la vitesse d'exécution des programmes en MQL5, en vous basant sur la discussion du forum "PLO - Use of OOP".
2. Évaluez l'approche de conception de l'Expert Advisor ExpertSimpleRandom.mq5. Discutez de la pertinence de la décomposition en "cerveau", "évolution" et "graphique" pour modéliser un système de trading.
3. Comparez et opposez les patterns "Mediator" et "Chain of Responsibility". Dans quels scénarios de trading distincts chacun pourrait-il être le plus bénéfique ?
4. Discutez de l'importance du polymorphisme pour créer des systèmes de trading flexibles. Comment l'utilisation de fonctions virtuelles, comme démontré avec GetArea(), pourrait-elle être appliquée à différentes stratégies de trading ou types d'ordres ?
5. Critiquez l'affirmation selon laquelle la POO est "hautement recommandée" mais pas obligatoire pour le développement en MQL5. Quels sont les facteurs qu'un développeur

devrait considérer pour décider d'utiliser une approche orientée objet ou une approche procédurale pour un nouveau projet d'EA ?

2.3 Glossaire des Termes Clés

Ce glossaire fournit des définitions concises des termes techniques essentiels abordés dans ce rapport, afin de faciliter votre compréhension et votre apprentissage.

Classe (Class) : Un "plan" (blueprint) pour créer des objets. Elle encapsule des données et des méthodes qui opèrent sur ces données.

Constructeur (Constructor) : Une méthode spéciale d'une classe, portant le même nom que la classe, qui est appelée automatiquement lors de la création d'un objet pour initialiser ses membres.

Design Pattern (Patron de Conception) : Une solution générale et réutilisable à un problème couramment rencontré dans la conception de logiciels.

Encapsulation : Le mécanisme consistant à regrouper les données et les méthodes qui les manipulent au sein d'un objet, et à cacher les détails de l'implémentation interne.

Héritage (Inheritance) : Un mécanisme de la POO qui permet à une nouvelle classe (dérivée) d'acquérir les propriétés et les comportements d'une classe existante (de base).

Méthode (Method) : Une fonction qui est membre d'une classe.

MQL5 : Un langage de programmation de haut niveau, similaire à C++, conçu pour développer des applications de trading et supportant la programmation orientée objet.

Objet (Object) : Une instance d'une classe. C'est une variable d'un type défini par l'utilisateur, qui combine données et algorithmes.

Polymorphisme (Polymorphism) : La capacité pour des objets de différentes classes, liées par l'héritage, de répondre de diverses manières à l'appel de la même fonction.

private : Spécificateur d'accès qui rend un membre de classe accessible uniquement au sein de la classe elle-même.

protected : Spécificateur d'accès qui rend un membre de classe accessible uniquement au sein de la classe elle-même et de ses classes dérivées.

public : Spécificateur d'accès qui rend un membre de classe accessible depuis n'importe quelle partie du programme.

Type de Données Abstrait (ADT - Abstract Data Type) : Une abstraction du concept traditionnel de type de données, utilisée pour définir confortablement le domaine de données des applications.

UML (Unified Modeling Language) : Un langage graphique pour la conception de systèmes orientés objet.

Nous vous encourageons à utiliser ce guide pour approfondir et renforcer votre maîtrise de la Programmation Orientée Objet en MQL5.

Chapitre 3 : Foire Aux Questions (FAQ)

3.0 Introduction

Cette section est une ressource rapide conçue pour répondre aux questions les plus courantes et pratiques que les développeurs se posent lorsqu'ils abordent la Programmation Orientée Objet en MQL5. Elle vise à clarifier les doutes et à fournir des réponses directes basées sur les informations contextuelles.

3.1 Questions Fréquemment Posées

1. **Dois-je obligatoirement utiliser la POO pour programmer en MQL5 ?** Non, vous n'êtes pas forcé d'utiliser la POO. Cependant, elle est "hautement recommandée" pour aller plus loin dans la connaissance de la programmation de systèmes de trading automatisés et pour bénéficier de ses avantages en termes de réutilisation du code, de fiabilité et de maintenance.

2. **L'utilisation de la POO rend-elle mon Expert Advisor plus rapide ?** C'est un sujet débattu. Des tests menés par des membres de la communauté en 2010 ont montré que le code orienté objet pouvait être légèrement plus lent à l'exécution. Cependant, un représentant de MetaQuotes a précisé que ces différences étaient principalement dues à l'architecture système de MT5 par rapport à MT4 et que l'optimiseur de code de MQL5 n'était pas encore pleinement activé à l'époque. La POO vise principalement à améliorer la vitesse de développement, la clarté et la maintenance, pas nécessairement la vitesse d'exécution brute.

3. **Quelle est la différence entre l'héritage public, protected et private ?** Le type d'héritage détermine comment les membres de la classe de base sont accessibles dans la classe dérivée :

    - **Héritage public :** Les membres public et protected de la base conservent leur niveau d'accès dans la classe dérivée.
    - **Héritage protected :** Les membres public et protected de la base deviennent protected dans la classe dérivée.
    - **Héritage private :** Les membres public et protected de la base deviennent private dans la classe dérivée, et ne sont donc plus accessibles par d'éventuelles classes héritant de cette dernière.

4. **Qu'est-ce qu'un "pointeur invalide" (invalid pointer access) et comment l'éviter ?** Une erreur "invalid pointer access" se produit lorsque vous essayez d'utiliser un pointeur qui n'a pas été initialisé, c'est-à-dire qu'il ne pointe pas vers un objet valide en mémoire. Dans le cas de l'EA ExpertSimpleRandom.mqh, cette erreur est apparue car l'objet m_positionInfo de type CPositionInfo a été déclaré mais jamais créé. Pour l'éviter, il faut s'assurer que chaque objet pointeur est correctement instancié à l'aide de l'opérateur new (par exemple, m_positionInfo = new CPositionInfo();) avant d'être utilisé, généralement dans le constructeur de la classe.

5. **Qu'est-ce que l'UML et est-il nécessaire pour développer en MQL5 ?** L'UML (Unified Modeling Language) est un langage graphique pour concevoir des systèmes orientés objet. Bien qu'il soit un bon outil d'analyse, il n'est pas nécessaire pour développer en

MQL5. Son utilisation dépend des circonstances du projet, comme le temps disponible ou les compétences de l'équipe.

6. **Comment les classes sont-elles utilisées pour gérer les indicateurs en MQL5 ?** Les classes sont un moyen efficace de représenter et d'utiliser des indicateurs. On peut créer une classe (par exemple, CIndicator) qui encapsule la logique de l'indicateur. Cette classe peut contenir des membres protégés pour stocker le "handle" (identifiant unique) de l'indicateur et ses tampons de données, et des méthodes publiques pour accéder à ces données de manière contrôlée (par exemple, main(int shift) pour obtenir la valeur de l'indicateur sur une bougie spécifique).

7. **Puis-je créer plusieurs objets à partir de la même classe ? Quel est l'intérêt ?** Oui, absolument. C'est l'un des principaux avantages de la POO. L'intérêt est de pouvoir gérer plusieurs instances indépendantes qui partagent la même logique mais ont des états différents. Par exemple, vous pouvez créer deux objets à partir d'une classe d'indicateur de moyenne mobile : un pour une moyenne mobile rapide et un pour une moyenne mobile lente, chacun avec ses propres paramètres et valeurs, mais en utilisant le même code de base.

8. **Qu'est-ce qu'un fichier d'inclusion (.mqh) et pourquoi l'utiliser pour mes classes ?** Un fichier d'inclusion (.mqh) est un fichier contenant du code (comme des définitions de classes, de fonctions ou de variables) qui peut être inséré dans un autre fichier à l'aide de la directive #include. Utiliser des fichiers .mqh pour définir vos classes permet de garder le code de votre Expert Advisor principal propre, organisé et plus facile à lire, en séparant la logique métier de la définition des objets.

9. **Les constructeurs et les destructeurs sont-ils hérités par les classes dérivées ?** Non. Selon la documentation MQL5, les constructeurs et les destructeurs de la classe de base ne peuvent pas être hérités par une classe dérivée. Cependant, lors de la création d'un objet de classe dérivée, le constructeur de la classe de base est appelé en premier, suivi de celui de la classe dérivée.

10. **Quelle est la convention de nommage recommandée par MetaQuotes pour les classes en MQL5 ?** La convention recommandée par MetaQuotes est de faire commencer les noms de classe par la lettre C majuscule. Par exemple : CPerson, CTrade, CSimpleRandom.

Ces réponses devraient aider à clarifier les aspects pratiques de la mise en œuvre de la Programmation Orientée Objet dans vos projets MQL5.

Chapitre 4 : Chronologie de la POO dans l'Écosystème MQL5

4.0 Introduction

Cette chronologie retrace l'évolution de la discussion, de l'enseignement et de l'application de la Programmation Orientée Objet au sein de la communauté MQL5, telle que représentée par les sources fournies pour ce rapport. Elle met en lumière la progression des concepts, depuis les débats initiaux sur ses mérites jusqu'à son intégration en tant que pratique avancée pour la construction de systèmes de trading complexes.

4.1 Événements Clés

- **Juillet 2010 :** Publication du fil de discussion "PLO - Use of OOP".
    - ***Importance :*** Montre les premiers débats au sein de la communauté sur les avantages de la POO, en particulier concernant l'impact sur la vitesse d'exécution par rapport à MQL4. Les discussions révèlent des opinions partagées et un besoin de clarification sur les performances.
- **22 Juillet 2013 :** Publication de l'article "Another MQL5 OOP Class".
    - ***Importance :*** Fournit un guide pratique et détaillé pour construire un EA complet en utilisant les principes de la POO. Cet article établit une base pédagogique solide pour les développeurs, en illustrant la conception et l'implémentation d'un système de trading de A à Z avec une approche orientée objet.
- **5 Décembre 2023 :** Publication de l'article "Design Patterns in software development and MQL5 (Part 3)".
    - ***Importance :*** Indique une maturité avancée de l'écosystème MQL5. La discussion passe des concepts de base de la POO à des patrons de conception logicielle (design patterns) avancés, démontrant que la communauté s'attaque désormais à la construction de systèmes complexes et robustes avec des architectures logicielles éprouvées.
- **8 Août 2024 :** Publication du blog "Learn use of Class in mql5".
    - ***Importance :*** Démontre un besoin continu de contenu éducatif de base. En présentant des exemples simples et en comparant le code avec et sans classes, cet article s'adresse aux nouveaux venus et souligne que les fondamentaux de la POO restent un sujet pertinent et essentiel pour les programmeurs MQL5.
- **Non daté :** Documentation de Référence MQL5 et AlgoBook.
    - ***Importance :*** Servent de documentation fondamentale et pérenne. Ces ressources définissent officiellement la syntaxe, les concepts et les règles de la POO au sein du langage MQL5, agissant comme la source de vérité pour tous les développeurs.
- **Non daté :** Tutoriel Vidéo "MQL5 Programming Tutorial 1.06 - Classes and Objects".
    - ***Importance :*** Représente la diffusion des connaissances sur la POO via des formats multimédias. En utilisant des analogies visuelles et verbales (comme celle de la voiture), ce type de contenu rend les concepts abstraits de la POO plus accessibles à un public plus large et diversifié.

Cette chronologie illustre une progression naturelle de l'adoption de la POO au sein de la communauté MQL5 : depuis les débats initiaux et les questions de performance, en passant par l'établissement de bases éducatives solides, jusqu'à son intégration comme une pratique standard et avancée pour le développement de logiciels de trading sophistiqués.

Chapitre 5 : Liste des Sources

5.0 Introduction

Ce rapport a été compilé en s'appuyant exclusivement sur les informations contenues dans la bibliographie suivante.

5.1 Sources

1. Abdelmaaboud, Mohamed. "Design Patterns in software development and MQL5 (Part 3): Behavioral Patterns 1". *MQL5.com Articles*, 5 Décembre 2023.
2. [Auteur Supprimé]. "Another MQL5 OOP Class". *MQL5.com Articles*, 22 Juillet 2013.
3. MetaQuotes Ltd. "Inheritance - Object-Oriented Programming - Language Basics". *MQL5 Reference*, (s.d.).
4. MetaQuotes Ltd. "Object-Oriented Programming - Language Basics". *MQL5 Reference*, (s.d.).
5. MetaQuotes Ltd. "OOP fundamentals: Encapsulation". *MQL5 AlgoBook*, (s.d.).
6. MetaQuotes Ltd. "Object Oriented Programming in MQL5". *MQL5 AlgoBook*, (s.d.).
7. MetaQuotes Ltd. "Polymorphism - Object-Oriented Programming - Language Basics". *MQL5 Reference*, (s.d.).
8. Nait0391, Rajesh Kumar. "Learn use of Class in mql5". *MQL5.com Traders' Blogs*, 8 Août 2024.
9. Toolkit for Traders. "MQL5 Programming Tutorial 1.06 - Classes and Objects". *YouTube*, (s.d.).
10. Divers contributeurs (Fedoseev, D., Chalyshev, S., Fatkhullin, R.). "PLO - Use of OOP to improve the speed of program". *MQL5.com Forum*, Juillet 2010.

Ce document peut contenir des inexactitudes ; veuillez vérifier attentivement son contenu. Pour plus d'informations, visitez le site PowerBroadcasts.com