

MQL5 Best Practices Guide

TradeAssistant V2.012

Projekt:	DowHowSignalService TradeAssistant
Version:	V2.012
Analyse-Datum:	04.01.2026
Qualitäts-Score:	76/100 - GUT ✓✓

Executive Summary

Dieser Guide analysiert den TradeAssistant V2.012 anhand etablierter MQL5 Best Practices. Der Code zeigt eine solide Architektur mit Verbesserungspotenzial in spezifischen Bereichen. Jede Kategorie enthält konkrete Code-Beispiele aus dem Projekt sowie Verbesserungsvorschläge.

Kategorie	Bewertung	Priorität
Positive Aspekte	5 gefunden ✓	Beibehalten
Verbesserungen	2 identifiziert	Hoch
Kritische Issues	1 gefunden	Mittel
Performance	0 Issues	OK ✓
Code-Style	3 Issues	Niedrig

1. Positive Aspekte (Was gut ist)

1.1 Modulare Architektur ✓✓✓

Status: Sehr gut implementiert

Der Code ist in 10+ separate Dateien aufgeteilt, was Wartbarkeit und Testbarkeit erhöht.

✓ **Aktuelles Design:**

- `trade_assistant.mq5` (Haupt-EA)
- `gui_elemente.mqh` (UI-Funktionen)
- `trades_panel.mqh` (Panel-Management)
- `db_service.mqh` (Datenbank-Layer)
- `discord_client.mqh` (Discord-Integration)

Vorteile:

- Klare Separation of Concerns
- Einfacheres Debugging (Fehler isoliert in Modulen)
- Wiederverwendbarkeit (Module in anderen Projekten nutzbar)
- Parallele Entwicklung möglich

1.2 Objektorientiertes Design ✓✓

Status: Gut - Mehrere Klassen für Kapselung

Classes: CDBService, CTradeManager, CUIManager, CDiscordClient, CLogger, CConfig

✓ **Gutes Beispiel - CDBService:**

```
class CDBService {
private:
    int m_db_handle;
    string m_db_path;
public:
    bool Init();
    bool SavePosition(...);
    bool LoadPosition(...);
};
```

Vorteile:

- Datenkapselung (private Member)
- Klar definierte Schnittstellen (public Methods)
- Zustandsverwaltung in Objekten

1.3 Error Handling ✓✓

Status: Gut - Systematische Fehlerbehandlung

22 GetLastError() Aufrufe für Fehlerprüfung gefunden

✓ **Gutes Beispiel:**

```
ResetLastError();
if(!ObjectCreate(...)) {
    int err = GetLastError();
    CLogger::Add(LOG_LEVEL_ERROR, "Failed: " + IntegerToString(err));
}
```

```
return false;  
}
```

Best Practice:

- Immer `ResetLastError()` vor kritischen Operationen
- `GetLastError()` sofort nach Fehlschlag prüfen
- Fehler loggen mit Context-Information
- Return false bei Fehlern für Fehler-Propagierung

1.4 Strukturiertes Logging ✓✓✓

Status: Sehr gut - Dedizierte Logger-Klasse

CLogger bietet Log-Levels, File-Logging und strukturierte Ausgabe

✓ **Gutes Beispiel:**

```
CLogger::Add(LOG_LEVEL_INFO, "EA Started");  
CLogger::Add(LOG_LEVEL_WARNING, "Discord init failed");  
CLogger::Add(LOG_LEVEL_ERROR, "DB connection error: " + err);
```

Vorteile:

- Log-Levels für Filterung (DEBUG, INFO, WARNING, ERROR)
- Zentrale Log-Datei für Troubleshooting
- Strukturierte Ausgabe statt Print()
- Production/Debug Modes möglich

1.5 Konstanten statt Magic Numbers ✓✓

Status: Gut - 23 #defines für wichtige Werte

Beispiele: EntryButton, SLButton, SENDTRADEBTN, Entry_Long, SL_Long

✓ **Gutes Beispiel:**

```
#define EntryButton "EntryButton"  
#define SLButton "SLButton"  
#define SENDTRADEBTN "SendOnlyButton"
```

✗ **Vermeide Magic Strings:**

```
// Schlecht:  
if(ObjectFind(0, "EntryButton") >= 0) { ... }  
  
// Gut:  
if(ObjectFind(0, EntryButton) >= 0) { ... }
```

Vorteile:

- Zentrale Änderung möglich (DRY-Prinzip)
- Keine Tippfehler bei String-Literalen
- IntelliSense/Autocomplete funktioniert
- Compiler findet Fehler bei falscher Verwendung

2. Verbesserungspotenzial

2.1 Lange Funktionen [HOCH]

Problem: 2 Funktionen mit >100 Zeilen

Längste: UI_TradesPanel_RebuildRows() mit 216 Zeilen

Warum ist das ein Problem?

- Schwer zu verstehen und zu testen
- Hohe Komplexität (viele Code-Pfade)
- Schwer zu debuggen
- Verletzt Single Responsibility Principle

X Aktuell (vereinfacht):

```
void UI_TradesPanel_RebuildRows() {  
    // 50 Zeilen: Daten laden  
    // 60 Zeilen: UI erstellen  
    // 50 Zeilen: Buttons konfigurieren  
    // 40 Zeilen: Labels positionieren  
    // 16 Zeilen: Cleanup  
    // = 216 Zeilen INSGESAMT  
}
```

✓ Besser - Aufteilen in Subfunktionen:

```
void UI_TradesPanel_RebuildRows() {  
    TradeData data = LoadTradeData();  
    ClearOldRows();  
    CreateRowUI(data);  
    PositionLabels(data);  
    // = 10-15 Zeilen mit klarer Struktur  
}
```

Empfehlung:

- Funktionen auf max. 50 Zeilen begrenzen
- Eine Abstraktionsebene pro Funktion
- Logische Blöcke in private Helper-Funktionen auslagern
- Extract Method Refactoring anwenden

2.2 Input-Validierung [MITTEL]

Problem: Nur 0/11 Input-Parameter werden validiert

Input-Parameter können ungültige Werte haben

X Aktuell - Keine Validierung:

```
input int riskMoney = 250;  
input int DistancefromRight = 300;  
  
int OnInit() {  
    // Keine Prüfung ob riskMoney > 0  
    // Keine Prüfung ob DistancefromRight sinnvoll  
    return INIT_SUCCEEDED;  
}
```

✓ Besser - Mit Validierung:

```
int OnInit() {  
    if(riskMoney <= 0) {  
        Alert("Risk Money muss > 0 sein!");  
        return INIT_PARAMETERS_INCORRECT;  
    }  
    if(DistancefromRight < 100 || DistancefromRight > 1000) {  
        Alert("Distance sollte zwischen 100-1000 sein");  
        return INIT_PARAMETERS_INCORRECT;  
    }  
    return INIT_SUCCEEDED;  
}
```

Best Practice:

- Alle Input-Parameter in OnInit() validieren
- Bei ungültigen Werten: INIT_PARAMETERS_INCORRECT returnen
- Klare Fehlermeldung mit Alert() oder Comment()
- Min/Max-Werte für numerische Inputs prüfen
- Strings auf Leerheit und Format prüfen

3. Kritisches Issue

3.1 Race Conditions [MITTEL]

Problem: Viele globale Variablen + OnTick = potenzielle Race Conditions
OnTick kann gleichzeitig mit Chart-Events (OnChartEvent) laufen

Beispiel-Szenario:

1. OnTick liest globale Variable 'active_long_trade_no'
2. Gleichzeitig: User klickt Cancel-Button
3. OnChartEvent ändert 'active_long_trade_no' auf 0
4. OnTick arbeitet mit altem Wert weiter → Inkonsistenz!

Lösungsansätze:

Option 1: Atomare Operationen

```
// Nutze eingebaute atomare Funktionen:  
long InterlockedExchange(long& var, long value);  
long InterlockedAdd(long& var, long value);
```

Option 2: State-Flags

```
bool is_processing = false;  
  
void OnTick() {  
    if(is_processing) return; // Skip wenn busy  
    is_processing = true;  
    // ... Arbeit ...  
    is_processing = false;  
}
```

Option 3: Kapselung in Klassen

```
// Am besten: Keine globalen Variablen!  
class CTradeState {  
private:  
    int m_active_long;  
    int m_active_short;  
public:  
    int GetActiveLong() { return m_active_long; }  
    void SetActiveLong(int val) { m_active_long = val; }  
};
```

Empfehlung:

- Globale Variablen minimieren
- Zustand in Klassen kapseln
- OnTick so schlank wie möglich halten
- Schwere Operationen in OnChartEvent verschieben

4. Code-Style & Conventions

4.1 Inkonsistente Naming Convention

Problem: Mix aus Hungarian Notation, CamelCase und snake_case

Beispiele: m_db_handle (Hungarian), TradeManager (CamelCase), active_long_trade_no (snake_case)

MQL5 Naming Best Practices:

Element	Convention	Beispiel
Klassen	CUpperCamelCase	CTradeManager, CDBService
Methoden	UpperCamelCase	Init(), SavePosition()
Lokale Variablen	lower_snake_case	trade_no, entry_price
Globale Variablen	g_lower_snake	g_trade_manager, g_db
Member-Variablen	m_lower_snake	m_db_handle, m_trade_no
Konstanten	UPPER_SNAKE	MAX_TRADES, DEFAULT_RISK
#defines	UpperCamelCase	EntryButton, SLButton
Input-Parameter	InpUpperCamel	InpDebug, InpRequireDiscord

4.2 Lange Zeilen

Problem: 515 Zeilen über 120 Zeichen

Erschwert Lesbarkeit auf kleineren Bildschirmen

X Zu lang:

```
if(!ObjectCreate(0, name, OBJ_BUTTON, 0, 0, 0) || !ObjectSetInteger(0, name, OBJPROP_XDISTANCE, x) || !ObjectSetInteger(0, name, OBJPROP_YDISTANCE, y)) { ... }
```

✓ Besser - Umgebrochen:

```
if(!ObjectCreate(0, name, OBJ_BUTTON, 0, 0, 0) ||
!ObjectSetInteger(0, name, OBJPROP_XDISTANCE, x) ||
!ObjectSetInteger(0, name, OBJPROP_YDISTANCE, y)) {
// Error handling
}
```

Best Practice:

- Max. 120 Zeichen pro Zeile
- Bei Funktionsaufrufen: Ein Parameter pro Zeile
- Bei Conditions: Logische Operatoren am Zeilenanfang
- String-Konkatenation über mehrere Zeilen verteilen

4.3 Magic Strings

Problem: ~90 hart-kodierte Strings im Code
Fehleranfällig und schwer zu ändern

✗ Aktuell - Magic Strings:

```
ObjectSetString(0, "EntryButton", OBJPROP_TEXT, "Entry");  
ObjectSetString(0, "SLButton", OBJPROP_TEXT, "SL");  
// Mehrfache Verwendung derselben Strings
```

✓ Besser - Konstanten:

```
#define ENTRY_BTN_NAME "EntryButton"  
#define ENTRY_BTN_TEXT "Entry"  
#define SL_BTN_NAME "SLButton"  
#define SL_BTN_TEXT "SL"  
  
ObjectSetString(0, ENTRY_BTN_NAME, OBJPROP_TEXT, ENTRY_BTN_TEXT);
```

Noch besser - String-Ressourcen-Klasse:

```
class CStrings {  
public:  
    static const string ENTRY_BTN_NAME;  
    static const string ENTRY_BTN_TEXT;  
    static const string SL_BTN_NAME;  
    static const string SL_BTN_TEXT;  
};  
  
// Verwendung:  
ObjectSetString(0, CStrings::ENTRY_BTN_NAME, OBJPROP_TEXT,  
CStrings::ENTRY_BTN_TEXT);
```

5. Zusätzliche MQL5 Best Practices

5.1 OnTick() Optimierung

OnTick() wird bei jedem Tick aufgerufen und sollte daher extrem performant sein.

X Schlecht - Teure Operationen in OnTick:

```
void OnTick() {  
    // SEHR TEUER!  
    DatabaseExecute(...);  
    FileOpen(...);  
    ObjectCreate(...);  
    ChartRedraw();  
    for(int i=0; i<1000; i++) { ... }  
}
```

✓ Gut - Lean OnTick:

```
void OnTick() {  
    static datetime last_check = 0;  
    datetime now = TimeCurrent();  
  
    // Nur jede Sekunde prüfen  
    if(now == last_check) return;  
    last_check = now;  
  
    // Minimale Checks  
    CheckSLHit();  
    UpdateCurrentPrices();  
}
```

OnTick Best Practices:

- KEINE DB/File-Operationen
- KEINE UI-Erstellung (ObjectCreate)
- KEIN ChartRedraw()
- Mit static Variablen throttlen (z.B. nur jede Sekunde)
- Nur schnelle Preis-Checks und SL-Monitoring

5.2 Memory Management

Dynamische Objekte löschen:

```
CTradeManager *mgr = new CTradeManager();  
// ... Verwendung ...  
delete mgr; // WICHTIG!
```

Arrays freigeben:

```
double prices[];  
ArrayResize(prices, 1000);  
// ... Verwendung ...  
ArrayFree(prices); // Speicher freigeben
```

UI-Objekte in OnDeinit():

```
void OnDeinit(const int reason) {  
    ObjectsDeleteAll(0, "MyPrefix_");  
}
```

```
// Oder:  
ObjectDelete(0, "MyButton");  
ChartRedraw();  
}
```

5.3 const Correctness

const für read-only Parameter:

```
// Besser:  
void ProcessTrade(const string symbol, const int trade_no) {  
    // symbol und trade_no können nicht geändert werden  
}  
  
// Statt:  
void ProcessTrade(string symbol, int trade_no) { ... }
```

const für unveränderliche Variablen:

```
const double RISK_PERCENT = 0.02;  
const int MAX_POSITIONS = 10;  
const string DB_FILE = "trades.db";
```

5.4 Error Handling Patterns

Pattern 1: Early Return

```
bool DoSomething() {
    if(!Validate()) return false;
    if(!Initialize()) return false;
    if(!Execute()) return false;
    return true;
}
```

Pattern 2: Guard Clauses

```
void ProcessTrade(CTrade *trade) {
    if(trade == NULL) {
        CLogger::Add(LOG_LEVEL_ERROR, "Null trade");
        return;
    }
    if(!trade.IsValid()) {
        CLogger::Add(LOG_LEVEL_ERROR, "Invalid trade");
        return;
    }
    // Normale Verarbeitung
}
```

Pattern 3: Error Codes

```
enum ETradeError {
    TRADE_SUCCESS = 0,
    TRADE_INVALID_PARAMS = 1,
    TRADE_DB_ERROR = 2,
    TRADE_BROKER_ERROR = 3
};

ETradeError CreateTrade(...) {
    if(!ValidateParams()) return TRADE_INVALID_PARAMS;
    // ...
    return TRADE_SUCCESS;
}
```

5.5 Testbarkeit

Testbarer Code ist:

- Modular (kleine, fokussierte Funktionen)
- Entkoppelt (keine harten Dependencies)
- Vorhersagbar (keine globalen States)
- Dokumentiert (klare Interfaces)

✓ Testbare Funktion:

```
// Pure Function - kein globaler State
double CalculateLotSize(double risk, double entry, double sl, double balance) {
    double distance = MathAbs(entry - sl);
    if(distance == 0) return 0;
    return (balance * risk) / (distance * 100000);
}

// Einfach zu testen mit verschiedenen Inputs
```

X Schwer testbar:

```
// Nutzt globale Variablen
double CalculateLotSize() {
// Liest: global_risk, Entry_Price, SL_Price, AccountBalance()
// Schwer zu testen - viele Dependencies
return ...;
}
```

6. Aktionsplan

Priorisierte Verbesserungen für TradeAssistant V2.012:

Priorität	Aktion	Aufwand	Nutzen
1 - Hoch	Lange Funktionen refactoren	2-3 Tage	Wartbarkeit ↑↑
2 - Hoch	Input-Validierung hinzufügen	2-4 Stunden	Stabilität ↑↑
3 - Mittel	Race Conditions absichern	1-2 Tage	Zuverlässigkeit ↑
4 - Mittel	Naming Conventions vereinheitlichen	1 Tag	Lesbarkeit ↑
5 - Niedrig	Lange Zeilen umbrechen	4-6 Stunden	Lesbarkeit ↑
6 - Niedrig	Magic Strings in Konstanten	2-3 Stunden	Wartbarkeit ↑

Fazit

Der TradeAssistant V2.012 zeigt eine **solide Code-Basis** mit einem Qualitäts-Score von **76/100 (GUT)**. Die Architektur ist modular, Error-Handling ist vorhanden und ein strukturiertes Logging-System ist implementiert.

Hauptstärken:

- Klare Modul-Struktur mit Separation of Concerns
- Objektorientiertes Design mit mehreren Klassen
- Systematisches Error-Handling und Logging
- Verwendung von Konstanten statt Magic Numbers

Verbesserungspotenzial:

- Refactoring langer Funktionen (>100 Zeilen)
- Input-Validierung implementieren
- Race Conditions bei globalen Variablen absichern
- Code-Style vereinheitlichen

Mit den vorgeschlagenen Verbesserungen kann der Score auf **85+ (EXZELLENT)** gesteigert werden. Die Änderungen sind in 5-7 Arbeitstagen umsetzbar und würden Wartbarkeit, Stabilität und Lesbarkeit deutlich verbessern.