AmiumScripter – Architektur- und Konzeptbeschreibung

# 1. Zielsetzung

AmiumScripter ist ein modulares, dynamisches UI- und Signal-Framework für technische und wissenschaftliche Anwendungen. Im Fokus stehen maximale Flexibilität, Laufzeit-Dynamik und saubere Trennung von Logik, Visualisierung und Steuerung.

# 2. Komponentenüberblick

* a) Projekte & Pages

- Ein Projekt besteht aus mehreren „Pages“ – funktionale Einheiten mit eigener Logik und Oberfläche.  
- Jede Page lebt in einem eigenen Verzeichnis (Pages/PageName/) und besteht aus drei Dateien:

• Page.cs: Ablaufsteuerung, Business-Logik, Signaldefinition (ohne UI)  
 • controls.cs: Visuelle Controls, reines Layout (generiert/änderbar im Editor)  
 • view.cs: UI-Logik (Verhalten, Events, Visual-Logik)

* b) Zentrale Steuerung

- Project.cs: Verwaltet alle Pages und Views, Instanziierung, Initialisierung, Start/Stop.  
- UIEditor: Visual-Editor für Controls, Tab-Verwaltung, Drag & Drop.  
- SignalManager/DataStorage: Zentrale Verwaltung und Pooling aller Signale.

* c) Dynamisches Build-System

- Mit Roslyn werden Pages und Controls zur Laufzeit kompiliert und geladen.  
- Dynamische Assemblies erlauben neue/editiere Pages sofort im laufenden System.

* d) Thread- und Task-Management

- AThread und ATask bieten zentrale Verwaltung und Lifecycle-Management für Hintergrundarbeiten.  
- Ein TokenManager sorgt für kontrollierte, globale Task-Abbrüche bei Rebuild/Unload.  
- Der Build-Checker warnt vor direkter Nutzung von Thread/Task/Sleep/while(true).

# 3. Grundprinzipien & Vorteile

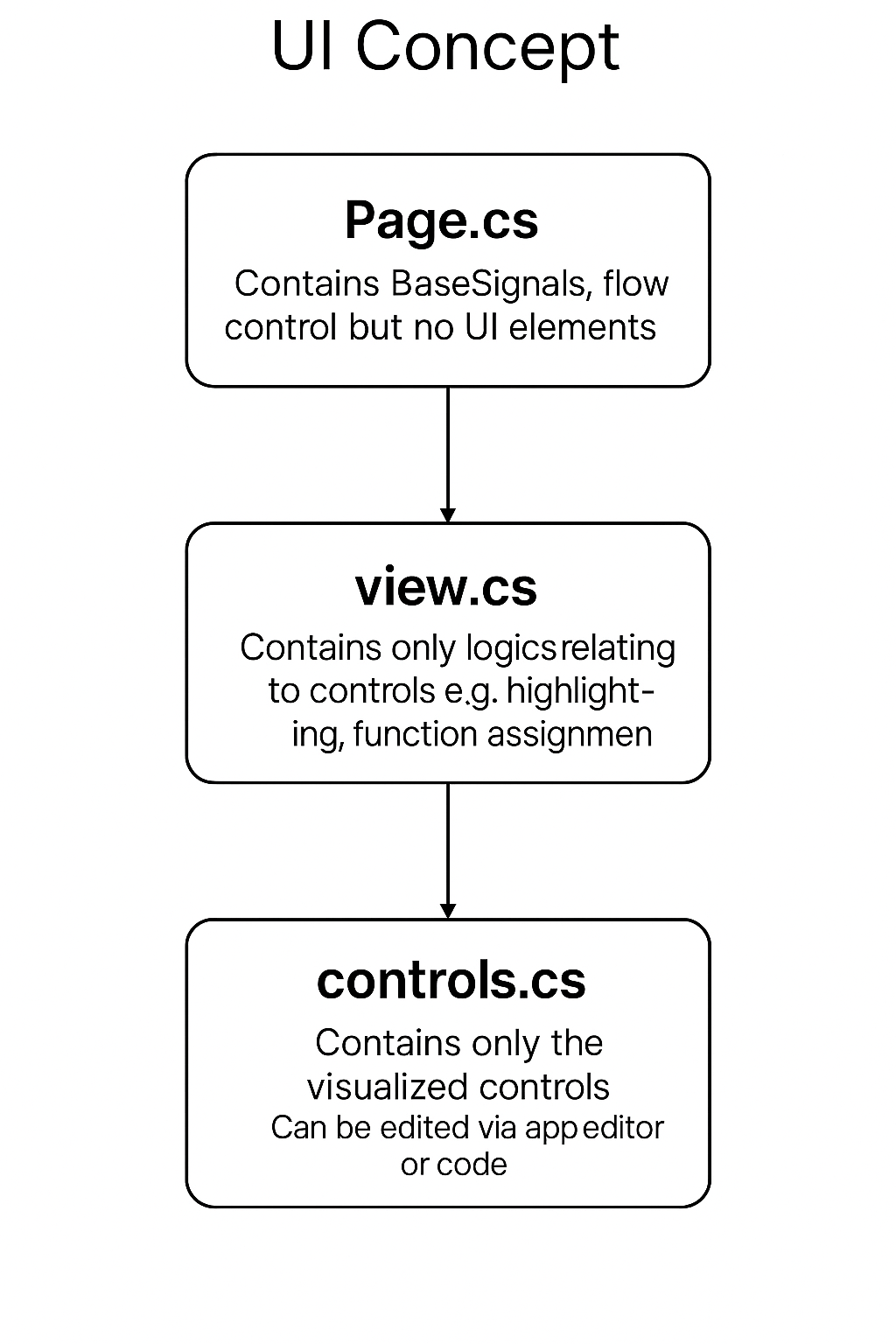
- Trennung von Logik, Layout, UI-Verhalten: Erhöht Wartbarkeit, Testbarkeit, Wiederverwendbarkeit.  
- Signalbasiertes Datenmodell: Alle Datenflüsse laufen über zentral verwaltete Signale.  
- Runtime-Dynamik: Änderungen an Controls, Pages, Layouts ohne App-Neustart möglich.  
- Sauberes Ressourcenmanagement: Zentrale Verwaltung aller Tasks, Threads und Tokens.  
- Automatisiertes Refactoring Enforcement: Build-Checker verhindert kritische Coding-Patterns.

# 4. Technisches Zusammenspiel

Ablauf (Ladeprozess):  
1. Build/Compile: Alle Pages/Views/Controls werden dynamisch kompiliert.  
2. Project.cs: Erstellt Instanzen, fügt sie Dictionaries hinzu, ruft Initialize()/Run() auf.  
3. UIEditor: Bindet die Views ins UI (Tabs).  
4. SignalManager: Stellt Datenpools bereit.  
5. Background-Threads werden (über AThread/ATask) gestartet und sind jederzeit zentral stoppbar.

# 5. Typisches Daten- und Signalfluss-Diagramm

Das folgende Diagramm zeigt das Zusammenwirken von Page, View, Controls und dem zentralen SignalPool:



# 6. Erweiterbarkeit

- Neue Pages lassen sich im Editor oder per Code hinzufügen.  
- Controls können per Drag&Drop, Code oder Editor ergänzt werden.  
- Signal-Konnektivität kann über eigene SignalClients/AClients integriert werden.  
- Fehlerresistenz durch Hot-Rebuilds, zentralen Unload und Build-Checker.

# 7. Best Practices

- Alle Logik, die Threads/Tasks/Signale verwendet, sollte die Helper (AThread, ATask, TokenManager) nutzen.  
- Dumme Controls, smarte Views, intelligente Pages.  
- Keine Business-Logik im UI, keine UI-Logik in der Page.  
- Bei „while(true)“ immer IsRunning oder CancellationToken verwenden.

# Fazit

AmiumScripter bietet eine hochflexible, robuste und teamtaugliche Architektur für modulare, dynamische Anwendungen mit klarer Trennung aller Verantwortlichkeiten.