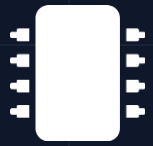


# Meta-Programming & Source Generators

## Der Auto-Discovery Orchestrator

*Wissenschaftliche Frage: Vorteile von Source Generators gegenüber Runtime-Reflection bei der dynamischen Modul-Entdeckung*

Gruppe 5: Cedric · Max · Marco

IU Internationale Hochschule · C# · Q1/2026

# Agenda

01

## Theorie & Wissenschaftliche Grundlagen

Metaprogrammierung, Reflection, Source Generators

02

## Implementierung — IIncrementalGenerator

Roslyn, AST, Projekt-Setup, Generator-Kern

03

## Diagnostics, Pattern Matching & Integration

GAE001/002, Kategorisierung, Architektur, Benchmark

# Teil 1

## Theorie & Wissenschaftliche Grundlagen

*Metaprogrammierung · Reflection · Source Generators*

TEIL 1

# Warum reden wir darüber?

Der Hub soll automatisch neue Spiele erkennen,
ohne dass wir jedes einzelne manuell registrieren.

```
Register(new Tetris());
Register(new Snake());
Register(new Pong());
```

Neues Spiel hinzufügen

Es funktioniert automatisch

Projekt neu bauen

TEIL 1

# Was ist Metaprogrammierung?

## Code, der Code als Daten behandelt

— analysiert, verändert oder generiert anderen Code

*Ingebrigtsen (2023), Kap. 1*

## Zwei Dimensionen

*(Hazzard & Bock, 2013)*

**Zeitlich:** Compile-time vs. Runtime
**Metadaten:** Implizit vs. Explizit

### Reflection

Runtime-Analyse von Typen

### Attributes

Explizite Compile-time Metadaten

### Expression Trees

Code als Datenstruktur (Runtime)

### Source Generators

Compile-time Code-Generierung

TEIL 1

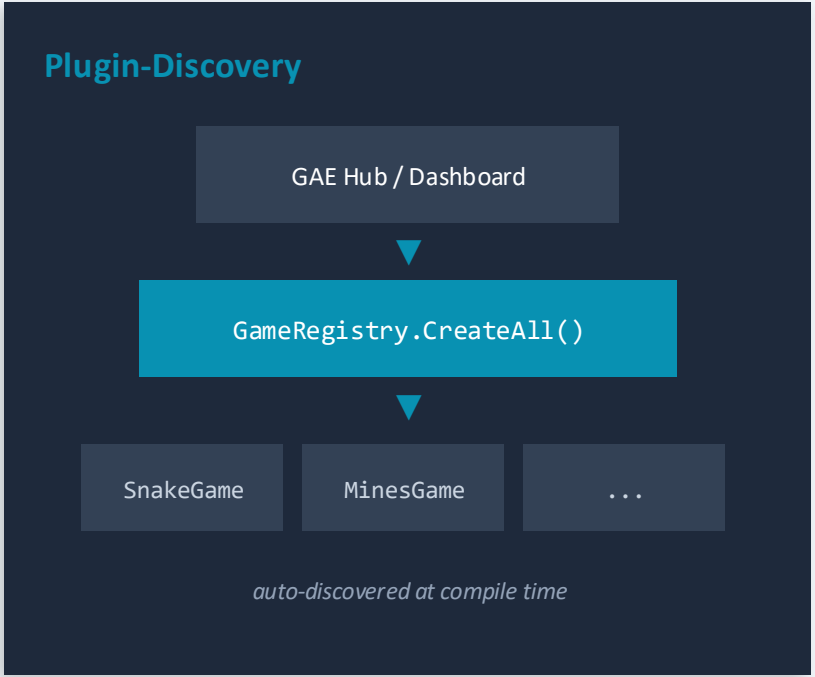
# Einordnung: Das GAE-Projekt

## Grand Arcade Ecosystem

Der GAE-Hub muss Spiele-Module dynamisch entdecken, ohne sie explizit zu kennen.

**Gruppe 5 untersucht zwei Ansätze:**

- Runtime-Reflection
→ langsam, AOT-inkompatibel
- Source Generators
→ zero overhead, typsicher



TEIL 1

# Reflection — Mechanismus & Kosten

```
// Assembly laden → Typen scannen → Interface-Check
var gameTypes = assembly.GetTypes()
   .Where(t => typeof(IArcadeGame)
         .IsAssignableFrom(t) && !t.IsInterface);

foreach (var type in gameTypes)
  if (Activator.CreateInstance(type) is IArcadeGame game)
     GameRegistry.Register(game);
```

## Performance-Problem

Signifikante Overheads bei
performance-kritischen Szenarien

*Coulson et al. (2004), IEEE ICSE*

## AOT-Problem: Trimming

Native AOT entfernt ungenutzten Code.
Reflection versteckt Abhängigkeiten → Typen werden getrimmt.
(Ingebrigtsen 2023, Kap. 16)

## CLR-Architektur: 'Cold Data'

EEClass speichert Reflection-Metadaten als 'cold data'
→ Schnelle Reflection war kein CLR-Design-Ziel.
(Warren 2016)

TEIL 1

# Source Generators – Mechanismus & Vorteile

```
// Compile-Time Analyse → Code-Generierung → direkte Registrierung
// Generated at compile time
public static partial class GeneratedGameRegistry
{
    public static void RegisterAll()
    {
                        GameRegistry.Register(new SnakeGame());
                        GameRegistry.Register(new MinesGame());
    }
}
```

### Zero Runtime Overhead

Keine Reflection zur Laufzeit
Kein Metadata-Scan
Kein Activator.CreateInstance
Startup konstant

*(Ingebrigtsen 2023, Kap. 16)*

### AOT-kompatibel

Code ist explizit vorhanden
Keine versteckten Typ-Abhängigkeiten
Kein Trimming-Problem
Native AOT geeignet

### Typsicherheit (Compile Time)

Fehler beim Build, nicht zur Laufzeit
Vollständige Compilerprüfung
Kein Reflection.Emit
Type-Safe Metaprogramming (Sheard 2001)

TEIL 1

# Source Generators vs. Reflection

| Kriterium | Runtime Reflection | Source Generators |
|---|---|---|
| Zeitpunkt | Laufzeit | **Kompilierzeit** |
| Performance | Overhead (Metadata-Scan) | **Kein Runtime-Overhead** |
| AOT-Kompatibilität | Problematisch (Trimming) | **Vollständig kompatibel** |
| Typsicherheit | Laufzeitfehler möglich | **Kompilierfehler** |
| Flexibilität | Kann unbekannte DLLs laden | Nur compile-time bekannter Code |

*Quellen: Ingebrigtsen (2023), Coulson et al. (2004), Warren (2016)*

TEIL 1

# Wissenschaftliche Einordnung

## Type-Safe Metaprogramming

*Sheard & Taha (2001)*

Metaprogrammierung ist nur verlässlich, wenn der generierte Code nachweislich typsicher ist — und diese Garantie kann man nur bei Compile-Time-Generierung geben.

✓ **Source Generators erfüllen dieses Kriterium**

## Performance-Überlegenheit

*Banaszewski et al. (2019)*

Compile-Time-Ansätze sind sowohl in Performance als auch Korrektheit den Runtime-Ansätzen überlegen.

*Bestätigt durch iapgos-Paper (Informatyka, Automatyka, Pomiary)*

**Fazit: Source Generators sind die wissenschaftlich fundierte Wahl für Plugin-Discovery in geschlossenen Systemen.**

# Zusammenfassung Teil 1

Metaprogrammierung = Code, der Code als Daten behandelt

*Ingebrigtsen 2023; Hazzard & Bock 2013*

Reflection: mächtig, aber langsam und AOT-inkompatibel

*Coulson et al. 2004; Warren 2016*

Source Generators: zero overhead, typsicher, AOT-ready

*Ingebrigtsen 2023, Kap. 16*

Wissenschaftlich fundiert durch Sheard/Taha und Banaszewski et al.

---

*→ Nächster Teil: WIE wir den Source Generator gebaut haben*

# Teil 2

## Implementierung — Der IIncrementalGenerator

*Roslyn · AST · Projekt-Setup · Generator-Kern*

TEIL 2

# Roslyn & Abstract Syntax Trees

## Compiler as a Service

*Hazzard & Bock (2013, Kap. 10)*

Roslyn macht den C#-Compiler zu einer offenen API.
Source Generators haben Zugriff auf:

**Syntax Tree** — Code als Baumstruktur
**Semantic Model** — Typsystem & Bedeutung

### IIncrementalGenerator (empfohlen)

vs. ISourceGenerator (veraltet)

Inkrementelles Caching → nur geänderte
Dateien werden neu verarbeitet.

### AST-Beispiel

```
CompilationUnit
└─ ClassDeclaration: SnakeGame
   ├─ AttributeList
   │  └─ [ArcadeGame]
   ├─ BaseList: IArcadeGame
   ├─ PropertyDecl: Name
   ├─ MethodDecl: Initialize()
   ├─ MethodDecl: Update(dt)
   ├─ MethodDecl: Render()
   └─ MethodDecl: Dispose()
```

TEIL 2

# Projekt-Setup

**GAE.Generators.csproj**

```
<TargetFramework>
   netstandard2.0
</TargetFramework>
<EnforceExtendedAnalyzerRules>
   true
</EnforceExtendedAnalyzerRules>
<IsRoslynComponent>true</IsRoslynComponent>

+ Microsoft.CodeAnalysis.CSharp 4.8.0
```

**Consumer-Projekt (z.B. Demo.csproj)**

```
<ProjectReference
  Include='..\GAE.Generators\..'

  OutputItemType='Analyzer'
  ReferenceOutputAssembly='false'
/>

<!-- Generierte Dateien sichtbar: -->
<EmitCompilerGeneratedFiles>
  true
</EmitCompilerGeneratedFiles>
```

**Wichtig:** netstandard2.0 ist Pflicht (Roslyn-Anforderung) · `OutputItemType='Analyzer'` statt normaler Referenz · `ReferenceOutputAssembly='false'` → Generator-DLL wird nicht in Output kopiert

TEIL 2

# Der Generator-Kern

## Das [ArcadeGame]-Attribut

```
[AttributeUsage(AttributeTargets.Class)]
public sealed class ArcadeGameAttribute : Attribute
{
    public string? DisplayName { get; init; }
    public string? Description { get; init; }
}
```

## Initialize() — Syntax-Filter + Emission

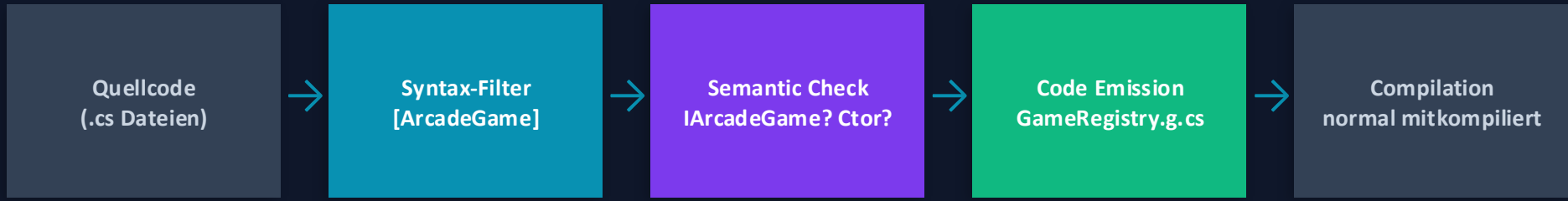
```
// PHASE 1: Syntax-Filter
var classes = context.SyntaxProvider
  .ForAttributeWithMetadataName(
    'GAE.Shared.Core.ArcadeGameAttribute',
    predicate: (n, _) => n is ClassDeclarationSyntax,
    transform: (ctx, _) => GetGameInfo(ctx));
```

## GetGameInfo() — Semantische Prüfung

```
private static GameInfo? GetGameInfo(GeneratorAttributeSyntaxContext ctx) {
    if (ctx.TargetSymbol is not INamedTypeSymbol classSymbol) return null;

    // Implementiert die Klasse IArcadeGame?
    var implementsInterface = classSymbol.AllInterfaces.Any(i => i.Name == "IArcadeGame");
    if (!implementsInterface) return null;    // → GAE001 Diagnostic

    // Öffentlicher parameterloser Konstruktor?
    var hasValidCtor = classSymbol.Constructors
        .Any(c => c.Parameters.Length == 0
               && c.DeclaredAccessibility == Accessibility.Public);

    return new GameInfo(classSymbol.ToDisplayString(), classSymbol.Name, hasValidCtor);
}
```

TEIL 2

# Die Generator-Pipeline




**Generierter Output: GameRegistry.g.cs**

```
namespace GAE.Generated;
public static class GameRegistry
{
    public static IReadOnlyList<IArcadeGame> CreateAll()
        => new IArcadeGame[] {
            new GAE.Module.Snake.SnakeGame(),     // ← auto-discovered
            new GAE.Module.Mines.MinesGame(),     // ← auto-discovered
        };
}
```

TEIL 2

# Vorher vs. Nachher

## ✗ Vorher: Reflection

```
// Langsam, AOT-unsicher, Laufzeitfehler
var games = new DiscoveryService()
    .DiscoverViaReflection();
```

- Assembly-Scan zur Laufzeit
- Activator.CreateInstance()
- TypeLoadException möglich

## ✓ Nachher: Source Generator

```
// Zero overhead, AOT-safe, compile-time
var games = GAE.Generated
    .GameRegistry.CreateAll();
```

- Statischer Methodenaufruf
- Direktes new — kein Reflection
- Fehler zur Kompilierzeit

**Eine einzige Zeile.** Kein Reflection, kein Scanning, kein Activator.CreateInstance().

# Zusammenfassung Teil 2

1. Roslyn = 'Compiler as a Service' — offene API für Code-Generierung
2. Source Generators nutzen AST (Syntax) + Semantic Model (Typsystem)
3. IIncrementalGenerator mit inkrementellem Caching (empfohlene API)
4. Unser Generator: 3 Phasen — Syntax-Filter → Semantic Check → Emission
5. Ergebnis: GameRegistry.CreateAll() ohne Runtime-Overhead

*→ Nächster Teil: Diagnostics, Pattern Matching & GAE-Integration*

# Teil 3

## Diagnostics, Pattern Matching & Integration

*GAE001/002 · Kategorisierung · Architektur · Benchmark*

TEIL 3

# Custom Compiler Diagnostics

## GAE001

**Missing IArcadeGame Interface**

```
Error GAE001: Class 'MyClass' has
[ArcadeGame] attribute but does not
implement IArcadeGame.
```

## GAE002

**Missing Parameterless Constructor**

```
Error GAE002: Class 'SnakeGame'
implements IArcadeGame but has no
public parameterless constructor.
```

### Implementierung im Generator

```
private static readonly DiagnosticDescriptor MissingInterfaceRule = new(
    id: 'GAE001',  title: 'Missing IArcadeGame implementation',
    messageFormat: 'Class '{0}' has [ArcadeGame] but does not implement IArcadeGame',
    category: 'GAE.Design',  defaultSeverity: DiagnosticSeverity.Error, isEnabledByDefault: true);

context.ReportDiagnostic(Diagnostic.Create(MissingInterfaceRule, classSymbol.Locations[0], name));
```

TEIL 3

# Pattern Matching für Spiel-Kategorisierung

```
public static class GameClassifier
{
    public static string Categorize(IArcadeGame game)
        => game switch
    {
        // Property Pattern
        { Name: var n } when n.Contains('Retro')
            => 'Classic',
        { Name: var n } when n.Contains('Puzzle')
            => 'Brain',

        // Default
        _ => 'Arcade'
    };

    public static string FormatForDashboard(
        IArcadeGame game)
        => $'[{Categorize(game)}] {game.Name}';
}
```

**Wie hängt das zusammen?**

**1.** Source Generator entdeckt Spiele

**2.** Pattern Matching kategorisiert sie für das Dashboard-Menü

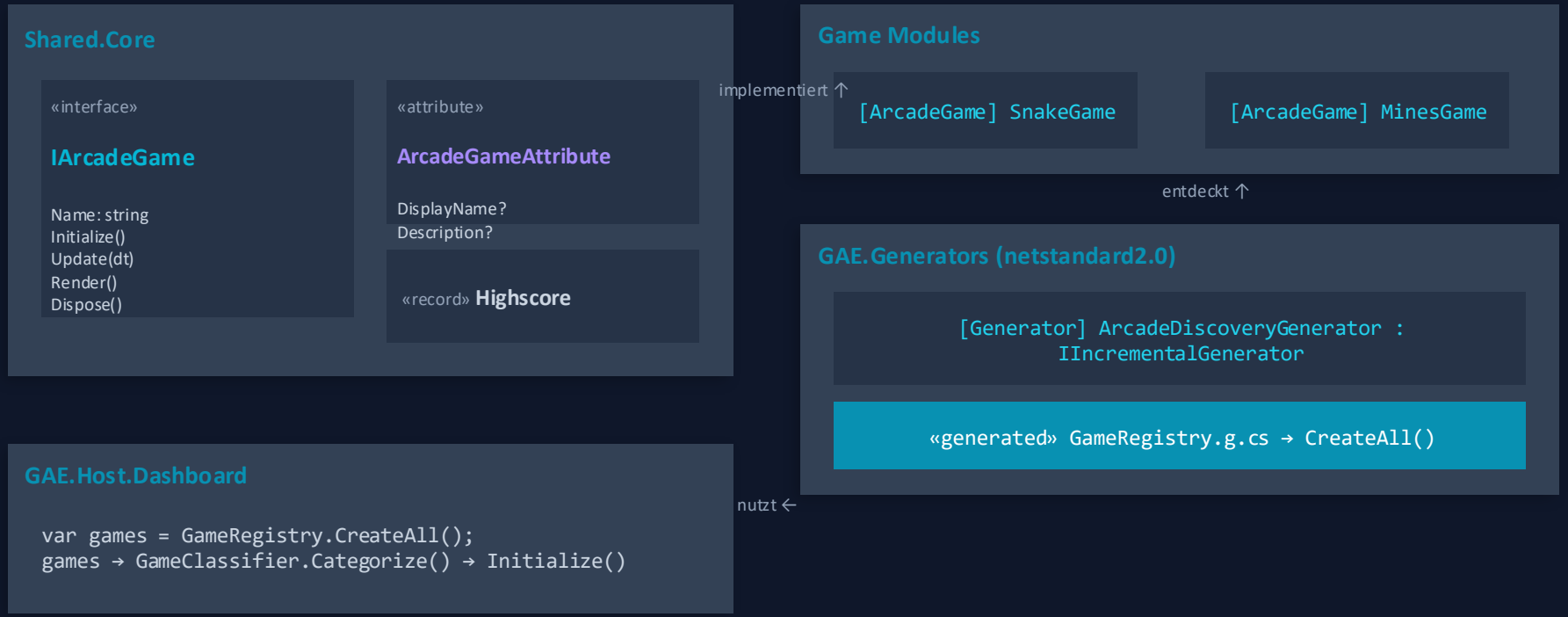
**3.** Typsichere Verzweigungslogik auf dynamisch entdeckte Objekte

*C# 9+ Switch Expressions mit Property Patterns und Guards*

*Ingebrigtsen (2023): Pattern Matching + Source Generators = typsichere Verzweigung auf dynamisch entdeckte Objekte*

TEIL 3

# Architektur-Übersicht



TEIL 3

# Benchmark: Reflection vs. Source Generator

**Reflection**

**~16 ms**

Startup-Latenz

**Source Generator**

**~0.1 ms**

Startup-Latenz

**~130×**

schneller

| Metrik | Reflection | Source Generator |
|---|---|---|
| Startup-Latenz | ~16 ms | **~0.1 ms** |
| Speicherverbrauch | Höher (Reflection-Cache) | **Minimal (direct new)** |
| AOT-kompatibel | Nein | **Ja** |
| Fehler-Zeitpunkt | Laufzeit | **Kompilierzeit** |

*Gemessen mit Stopwatch auf dem Demo-Projekt (2 Spiele, .NET 8, Release-Build)*

# Fazit & Ausblick

Source Generators sind Reflection überlegen für Plugin-Discovery in geschlossenen Systemen

Custom Diagnostics (GAE001/002) erzwingen korrekte Implementierungen beim Tippen

Pattern Matching ergänzt Discovery — typsichere Kategorisierung entdeckter Spiele

Reflection hat Berechtigung für offene Systeme (Post-Compile DLLs)

## Ausblick

Source Generators in .NET: System.Text.Json · Microsoft.Extensions.Logging · ASP.NET Minimal APIs
Andere Sprachen: Rust (Procedural Macros) · Kotlin (KSP) · Swift (Macros)

# Quellen & Fragen


1. Ingebrigtsen, E. (2023). Metaprogramming in C#. Packt Publishing.
2. Hazzard, K. & Bock, J. (2013). Metaprogramming in .NET. Manning.
3. Coulson, G. et al. (2004). On the Performance of Reflective Systems Software. IEEE ICSE.
4. Banaszewski, R. et al. (2019). Metaprogramming Techniques. IAPGOS.
5. Warren, M. (2016). Why is Reflection slow? Performance is a Feature!
6. Sheard, T. (2001). Accomplishments in Meta-Programming. LNCS 2196.
7. Microsoft (2024). .NET Source Generators & Native AOT Documentation.


**Fragen?**