Certamente, Marco! Ho aggiornato il documento per riflettere l'implementazione di AnomalyDetectedEvent.

### Documento Programmatico: Stato di Implementazione delle Classi del Sistema MIU

**Data:** 11 Luglio 2025 (Aggiornato)

Questo documento elenca le classi chiave per l'architettura event-driven e il motore di tassonomia, specificando il loro ruolo e lo stato di implementazione attuale.

**Legenda Stato:**

* ✅ **Implementato:** La classe è stata creata e il codice fornito.
* ⏳ **Da Implementare:** La classe è pianificata per essere creata.
* ➡️ **Esistente:** La classe esiste già nel tuo progetto e verrà riutilizzata.

#### I. Classi per l'Infrastruttura Event-Driven (Progetto: EvolutiveSystem.Common)

Queste classi rappresentano i "fatti fisici" (eventi) che accadono nel sistema e il meccanismo per la loro trasmissione.

1. **EventBus**
   * **Ruolo:** Il "mezzo di trasmissione" centrale per gli eventi in-memory, permettendo ai componenti di pubblicare e sottoscrivere in modo disaccoppiato.
   * **Stato:** ✅ **Implementato**
2. **RuleAppliedEventArgs**
   * **Ruolo:** La classe che rappresenta l'evento di applicazione di una regola MIU. Contiene i dettagli specifici di tale applicazione.
   * **Stato:** ➡️ **Esistente** (Classe già presente nel tuo progetto, verrà riutilizzata così com'è).
3. **SearchCompletedEvent**
   * **Ruolo:** La classe che rappresenta l'evento di completamento di un'intera ricerca MIU (con il suo esito di successo o i vari tipi di fallimento).
   * **Stato:** ✅ **Implementato**
4. **AnomalyDetectedEvent**
   * **Ruolo:** La classe che rappresenta l'evento di rilevamento di un'anomalia da parte dell'AnomalyDetectionManager.
   * **Stato:** ✅ **Implementato**

#### II. Classi per la Tassonomia (Progetto: EvolutiveSystem.Taxonomy)

Queste classi definiscono la struttura della tassonomia e la logica per la sua generazione.

1. **RuleTaxonomy**
   * **Ruolo:** La struttura dati principale che rappresenta l'intera tassonomia delle regole MIU, contenente nodi radice e la loro gerarchia.
   * **Stato:** ✅ **Implementato**
2. **RuleTaxonomyNode**
   * **Ruolo:** La struttura dati che rappresenta un singolo nodo all'interno della RuleTaxonomy, categorizzando le regole e potendo avere nodi figli.
   * **Stato:** ✅ **Implementato**
3. **RuleTaxonomyGenerator**
   * **Ruolo:** Il "motore" che sa *come* analizzare le statistiche delle regole (e altri dati) per costruire e aggiornare la RuleTaxonomy.
   * **Stato:** ✅ **Implementato**

#### III. Classi per l'Orchestrazione dell'Apprendimento (Progetto: EvolutiveSystem.Orchestration)

Questa classe sarà il "cervello" che interpreta il flusso di eventi per decidere quando innescare l'evoluzione della tassonomia.

1. **TaxonomyOrchestrator**
   * **Ruolo:** Il componente centrale che incapsula la logica della rete di Petri. Sottoscriverà agli eventi pubblicati dal sistema, genererà "token" e, in base a soglie e combinazioni di questi token, deciderà *quando* far scattare la rigenerazione della tassonomia.
   * **Stato:** ⏳ **Da Implementare**

#### IV. Classi di Dominio Esistenti (Progetto: EvolutiveSystem.Common)

Queste classi esistono già e sono fondamentali per il funzionamento del sistema, ma non sono classi evento in sé.

1. **ExplorationAnomaly**
   * **Ruolo:** Rappresenta una singola anomalia rilevata durante l'esplorazione del sistema MIU. Viene persistita nel database e i suoi dati possono essere usati per generare AnomalyDetectedEvent.
   * **Stato:** ➡️ **Esistente** (Classe già presente nel tuo progetto).

Ora che abbiamo tutte le classi evento definite e l'EventBus pronto, il prossimo passo è implementare il **TaxonomyOrchestrator**.

### 1. Creazione del Progetto EvolutiveSystem.Orchestration

Prima di tutto, dobbiamo creare il nuovo progetto che ospiterà il TaxonomyOrchestrator.

**Cosa fare:**

1. Nel tuo Visual Studio, fai clic destro sulla tua soluzione.
2. Seleziona **"Add" (Aggiungi)** -> **"New Project..." (Nuovo Progetto...)**.
3. Scegli il modello **"Class Library" (Libreria di classi)** per C#.
4. Dai al progetto il nome **EvolutiveSystem.Orchestration**.
5. Fai clic su **"Create" (Crea)**.

### 2. Impostazione dei Riferimenti del Progetto EvolutiveSystem.Orchestration

Perché il TaxonomyOrchestrator possa funzionare, avrà bisogno di accedere a classi definite in altri progetti.

**Cosa fare:**

1. Nel tuo **Esplora Soluzioni**, fai clic destro sul nuovo progetto **EvolutiveSystem.Orchestration**.
2. Seleziona **"Add Project Reference..." (Aggiungi riferimento al progetto...)**.
3. Nella finestra che si apre, vai alla scheda **"Projects" (Progetti)**.
4. Seleziona i seguenti progetti:
   * **EvolutiveSystem.Common** (per EventBus, RuleAppliedEventArgs, SearchCompletedEvent, AnomalyDetectedEvent, AnomalyType, IMIUDataManager, RuleStatistics, RegolaMIU)
   * **EvolutiveSystem.Taxonomy** (per RuleTaxonomyGenerator, RuleTaxonomy, RuleTaxonomyNode)
   * **MasterLog** (per il Logger)
5. Fai clic su **OK**.

### 3. Creazione della Classe TaxonomyOrchestrator

Ora possiamo creare la classe TaxonomyOrchestrator all'interno del nuovo progetto.

**Cosa fare:**

1. Nel progetto EvolutiveSystem.Orchestration, nella root (senza sottocartelle), crea un nuovo file C# chiamato TaxonomyOrchestrator.cs.
2. Copia e incolla il codice seguente in questo nuovo file.

// File: EvolutiveSystem.Orchestration/TaxonomyOrchestrator.cs  
using System;  
using System.Collections.Generic;  
using System.Threading.Tasks;  
using EvolutiveSystem.Common; // Per EventBus, AnomalyType, IMIUDataManager  
using EvolutiveSystem.Common.Events; // Per RuleAppliedEventArgs, SearchCompletedEvent, AnomalyDetectedEvent  
using EvolutiveSystem.Taxonomy; // Per RuleTaxonomyGenerator  
using MasterLog; // Per Logger  
  
namespace EvolutiveSystem.Orchestration  
{  
 /// <summary>  
 /// Il TaxonomyOrchestrator è il "cervello" basato su rete di Petri  
 /// che sa quando far scattare la generazione della tassonomia.  
 /// Sottoscrive agli eventi del sistema, genera "token" interni,  
 /// e in base a soglie e combinazioni di questi token, decide quando  
 /// invocare il RuleTaxonomyGenerator.  
 /// </summary>  
 public class TaxonomyOrchestrator  
 {  
 private readonly RuleTaxonomyGenerator \_taxonomyGenerator;  
 private readonly EventBus \_eventBus;  
 private readonly Logger \_logger;  
  
 // "Luoghi" della nostra rete di Petri semplificata (contatori e flag)  
 private int \_newRuleApplicationsCount = 0;  
 private int \_successfulSearchesCount = 0;  
 private int \_failedSearchesCount = 0; // Contatore per tutti i tipi di fallimento  
 private int \_anomalyDetectedCount = 0; // Contatore per anomalie rilevate  
 private DateTime \_lastTaxonomyGenerationTime = DateTime.MinValue;  
  
 // Soglie configurabili per innescare la rigenerazione (i nostri "trigger" del delta di Dirac)  
 private const int RULE\_APP\_THRESHOLD = 500; // Rigenera ogni 500 applicazioni di regole  
 private const int SUCCESS\_SEARCH\_THRESHOLD = 20; // Rigenera ogni 20 ricerche di successo  
 private const int FAILED\_SEARCH\_THRESHOLD = 50; // Rigenera ogni 50 ricerche fallite  
 private const int ANOMALY\_THRESHOLD = 5; // Rigenera ogni 5 anomalie rilevate  
 private const double TIME\_THRESHOLD\_HOURS = 24.0; // Rigenera almeno ogni 24 ore (fallback)  
  
 /// <summary>  
 /// Costruttore di TaxonomyOrchestrator.  
 /// Riceve le dipendenze necessarie tramite Dependency Injection.  
 /// </summary>  
 /// <param name="taxonomyGenerator">L'istanza del generatore di tassonomie.</param>  
 /// <param name="eventBus">L'istanza dell'Event Bus per sottoscrivere agli eventi.</param>  
 /// <param name="logger">L'istanza del logger.</param>  
 public TaxonomyOrchestrator(RuleTaxonomyGenerator taxonomyGenerator, EventBus eventBus, Logger logger)  
 {  
 \_taxonomyGenerator = taxonomyGenerator ?? throw new ArgumentNullException(nameof(taxonomyGenerator));  
 \_eventBus = eventBus ?? throw new ArgumentNullException(nameof(eventBus));  
 \_logger = logger ?? throw new ArgumentNullException(nameof(logger));  
  
 \_logger.Log(LogLevel.DEBUG, "TaxonomyOrchestrator istanziato. Sottoscrizione agli eventi in corso.");  
  
 // Sottoscrizione agli eventi rilevanti  
 \_eventBus.Subscribe<RuleAppliedEventArgs>(HandleRuleAppliedEvent);  
 \_eventBus.Subscribe<SearchCompletedEvent>(HandleSearchCompletedEvent);  
 \_eventBus.Subscribe<AnomalyDetectedEvent>(HandleAnomalyDetectedEvent);  
  
 // Inizializza il tempo dell'ultima generazione all'avvio dell'orchestratore  
 \_lastTaxonomyGenerationTime = DateTime.UtcNow;  
 }  
  
 /// <summary>  
 /// Handler per l'evento RuleAppliedEventArgs. Incrementa il contatore delle applicazioni di regole.  
 /// </summary>  
 /// <param name="e">I dati dell'evento RuleAppliedEventArgs.</param>  
 private Task HandleRuleAppliedEvent(RuleAppliedEventArgs e)  
 {  
 \_newRuleApplicationsCount++;  
 \_logger.Log(LogLevel.DEBUG, $"RuleAppliedEvent ricevuto. Contatore applicazioni: {\_newRuleApplicationsCount}");  
 CheckAndGenerateTaxonomy(); // Controlla se è il momento di generare la tassonomia  
 return Task.CompletedTask;  
 }  
  
 /// <summary>  
 /// Handler per l'evento SearchCompletedEvent. Incrementa i contatori delle ricerche completate.  
 /// </summary>  
 /// <param name="e">I dati dell'evento SearchCompletedEvent.</param>  
 private Task HandleSearchCompletedEvent(SearchCompletedEvent e)  
 {  
 if (e.Outcome == "Success")  
 {  
 \_successfulSearchesCount++;  
 \_logger.Log(LogLevel.DEBUG, $"SearchCompletedEvent (Successo) ricevuto. Contatore successi: {\_successfulSearchesCount}");  
 }  
 else  
 {  
 \_failedSearchesCount++;  
 \_logger.Log(LogLevel.DEBUG, $"SearchCompletedEvent (Fallimento: {e.Outcome}) ricevuto. Contatore fallimenti: {\_failedSearchesCount}");  
 }  
 CheckAndGenerateTaxonomy(); // Controlla se è il momento di generare la tassonomia  
 return Task.CompletedTask;  
 }  
  
 /// <summary>  
 /// Handler per l'evento AnomalyDetectedEvent. Incrementa il contatore delle anomalie.  
 /// </summary>  
 /// <param name="e">I dati dell'evento AnomalyDetectedEvent.</param>  
 private Task HandleAnomalyDetectedEvent(AnomalyDetectedEvent e)  
 {  
 \_anomalyDetectedCount++;  
 \_logger.Log(LogLevel.WARNING, $"AnomalyDetectedEvent ({e.Type}) ricevuto. Contatore anomalie: {\_anomalyDetectedCount}");  
 CheckAndGenerateTaxonomy(); // Controlla se è il momento di generare la tassonomia  
 return Task.CompletedTask;  
 }  
  
 /// <summary>  
 /// Controlla se le condizioni per la rigenerazione della tassonomia sono soddisfatte.  
 /// Questa è la logica della "rete di Petri" semplificata che decide il "collasso della funzione d'onda".  
 /// </summary>  
 public void CheckAndGenerateTaxonomy()  
 {  
 bool shouldGenerate = false;  
 string triggerReason = "Nessuna";  
  
 // Condizione 1: Abbastanza nuove applicazioni di regole  
 if (\_newRuleApplicationsCount >= RULE\_APP\_THRESHOLD)  
 {  
 shouldGenerate = true;  
 triggerReason = $"Soglia applicazioni ({RULE\_APP\_THRESHOLD}) raggiunta.";  
 }  
  
 // Condizione 2: Abbastanza nuove ricerche di successo  
 if (\_successfulSearchesCount >= SUCCESS\_SEARCH\_THRESHOLD)  
 {  
 shouldGenerate = true;  
 triggerReason = $"Soglia ricerche di successo ({SUCCESS\_SEARCH\_THRESHOLD}) raggiunta.";  
 }  
  
 // Condizione 3: Abbastanza nuove ricerche fallite (il fallimento è informazione!)  
 if (\_failedSearchesCount >= FAILED\_SEARCH\_THRESHOLD)  
 {  
 shouldGenerate = true;  
 triggerReason = $"Soglia ricerche fallite ({FAILED\_SEARCH\_THRESHOLD}) raggiunta.";  
 }  
  
 // Condizione 4: Abbastanza anomalie rilevate (segnalano problemi che la tassonomia potrebbe aiutare a risolvere)  
 if (\_anomalyDetectedCount >= ANOMALY\_THRESHOLD)  
 {  
 shouldGenerate = true;  
 triggerReason = $"Soglia anomalie ({ANOMALY\_THRESHOLD}) raggiunta.";  
 }  
  
 // Condizione 5: Tempo trascorso (fallback per evitare stasi del sistema)  
 if ((DateTime.UtcNow - \_lastTaxonomyGenerationTime).TotalHours >= TIME\_THRESHOLD\_HOURS)  
 {  
 shouldGenerate = true;  
 triggerReason = $"Soglia temporale ({TIME\_THRESHOLD\_HOURS} ore) raggiunta.";  
 }  
  
 if (shouldGenerate)  
 {  
 \_logger.Log(LogLevel.INFO, $"Condizioni per la rigenerazione della tassonomia soddisfatte. Motivo: {triggerReason}. Avvio generazione...");  
 try  
 {  
 \_taxonomyGenerator.GenerateRuleTaxonomy(); // Invocazione del generatore di tassonomie  
 \_logger.Log(LogLevel.INFO, "Rigenerazione tassonomia completata.");  
 }  
 catch (Exception ex)  
 {  
 \_logger.Log(LogLevel.ERROR, $"Errore durante la rigenerazione della tassonomia: {ex.Message}{Environment.NewLine}{ex.StackTrace}");  
 }  
 finally  
 {  
 // Resetta i contatori e il timestamp dopo ogni tentativo di generazione  
 \_lastTaxonomyGenerationTime = DateTime.UtcNow;  
 \_newRuleApplicationsCount = 0;  
 \_successfulSearchesCount = 0;  
 \_failedSearchesCount = 0;  
 \_anomalyDetectedCount = 0;  
 \_logger.Log(LogLevel.INFO, "Contatori di trigger per la tassonomia resettati.");  
 }  
 }  
 }  
 }  
}