

Aufgabenstellung

Einführung:

Übersicht der Applikation:

Was macht die Applikation | Wie funktioniert die Applikation?

Die Applikation ist ein Client für den Spotify-Service, konzipiert unter Verwendung der Clean Architecture-Prinzipien. Ihr Hauptzweck besteht darin, Benutzern eine strukturierte und modulare Möglichkeit zu bieten, nach Tracks, Alben, Künstlern und Genre-Playlists zu suchen sowie detaillierte Informationen zu diesen Musikressourcen abzurufen. Sie löst das Problem, dass Benutzer eine einfach zu bedienende Schnittstelle benötigen, um spezifische Musikinhalte auf Spotify zu finden und relevante Informationen dazu zu erhalten, ohne direkt die Spotify-App oder -Webseite nutzen zu müssen. Hier sind die Hauptfunktionen und deren Ablauf:

Suchfunktionen:

- Tracks suchen: Benutzer können nach Tracks (Liedern) suchen, indem sie den Namen des Tracks eingeben. Die Anwendung liefert daraufhin Vorschläge oder den spezifischen Track, der zu der Suchanfrage passt.
- Künstler suchen: Benutzer können nach Künstlern suchen. Als Ergebnis erhalten sie Informationen zu dem Künstler, inklusive Beliebtheit und Genres.
- Alben suchen: Benutzer können nach Alben suchen. Sie erhalten Details zu dem Album, wie Veröffentlichungsdatum, die enthaltenen Tracks und die beteiligten Künstler.
- Genre-Playlists suchen: Benutzer können Playlists durchsuchen, die einem spezifischen Musikgenre zugeordnet sind. Die Suche liefert Playlists, die dem angegebenen Genre entsprechen.

Informationsabruf:

- Nach der Suche können Benutzer detaillierte Informationen zu den gefundenen Musikressourcen abrufen. Dazu gehören unter anderem die Dauer von Tracks, die Anzahl der Tracks in einem Album, die Popularität von Künstlern und die Beschreibung von Playlists.

Musikwiedergabe:

- Die Applikation bietet zudem eine Funktion, um eine Vorschau von Tracks direkt in der Anwendung abzuspielen. Dies hängt von der Verfügbarkeit der Vorschau-URLs in den Spotify-Daten ab.

Welches Problem löst sie/welchen Zweck hat sie?

Durch die Bereitstellung einer benutzerfreundlichen Schnittstelle für die Suche und den Abruf von Informationen aus dem umfangreichen Spotify-Musikkatalog adressiert die Applikation das Bedürfnis von Benutzern nach einem gezielten Zugang zu Musikinhalten, ohne sich durch die möglicherweise überwältigende Vielfalt direkt auf Spotify navigieren zu müssen.

- Ermöglicht Zugriff auf Spotify-Daten ohne GUI Für Benutzer, die es bevorzugen oder benötigen, mit Anwendungen über eine Befehlszeilenschnittstelle (CLI) statt einer grafischen Benutzeroberfläche (GUI) zu interagieren, bietet dieses Programm eine effiziente Lösung. Es ist besonders nützlich in Umgebungen, in denen GUIs nicht verfügbar oder praktikabel sind, wie z.B. auf Servern oder in bestimmten Entwicklungs- oder Testumgebungen.
- Personalisierte Such- und Wiedergabeerfahrung: Das Programm ermöglicht eine detaillierte und personalisierte Suche nach Tracks, Alben, Künstlern und Playlists basierend auf Benutzereingaben. Es bietet eine flexible Steuerung der Suchkriterien und der Anzeige von Suchergebnissen, was eine auf den Benutzer zugeschnittene Erfahrung schafft.
- Bildungszwecke und API-Exploration: Für Entwickler oder Studenten, die lernen möchten, wie man mit Web-APIs interagiert, stellt dieses Programm ein praktisches Beispiel dar. Es zeigt, wie man Authentifizierung durchführt, Anfragen sendet, JSON-Daten verarbeitet und Antworten von einer externen API in einer realen Anwendung nutzt.

Starten der Applikation:

Voraussetzungen

- .NET Runtime oder SDK: Die Applikation ist in C# geschrieben und benötigt .NET Core 3.1 oder höher. Stellen Sie sicher, dass die .NET Core Runtime oder das SDK auf Ihrem System installiert ist. Sie können die neueste Version von der offiziellen [.NET-Website](#) herunterladen.

- Spotify Developer Account: Um auf die Spotify Web API zugreifen zu können, benötigen Sie einen Spotify Developer Account und müssen eine Applikation auf der [Spotify Developer Dashboard](#) erstellen, um die benötigten `clientId` und `clientSecret` zu erhalten.
- Internetverbindung: Da die Applikation Daten von der Spotify Web API abruft, ist eine stabile Internetverbindung erforderlich.

Start der Applikation

- Konfiguration der Spotify-API-Zugangsdaten: Öffnen Sie den Code und navigieren Sie zur `SpotifyCredentials`-Klasse. Ersetzen Sie `clientId` und `clientSecret` mit den Werten, die Sie aus Ihrem Spotify Developer Dashboard erhalten haben.
- Kompilieren und Ausführen:
 - Öffnen Sie eine Kommandozeile oder ein Terminal im Wurzelverzeichnis des Projekts, wo die `*.csproj`-Datei liegt.
 - Führen Sie den Befehl `dotnet build` aus, um das Projekt zu kompilieren.
 - Starten Sie die Applikation mit `dotnet run`.

Nach dem Start der Applikation wird in der Konsole das Hauptmenü angezeigt, von dem aus Sie verschiedene Aktionen durchführen können.

UseCases

UseCase 1: Suche nach einem Track

- Wählen Sie im Hauptmenü die Option für die Tracksuche (z.B. durch Eingabe der entsprechenden Nummer).
- Geben Sie den Namen des Tracks ein, den Sie suchen möchten.
- Die Applikation zeigt eine oder mehrere Übereinstimmungen an, basierend auf Ihrer Suche.
- Wählen Sie eine der angezeigten Optionen, um detaillierte Informationen zum Track zu erhalten oder eine Vorschau abzuspielen, falls verfügbar.

UseCase 2: Suche nach einem Künstler

- Wählen Sie im Hauptmenü die Option, um nach einem Künstler zu suchen.
- Geben Sie den Namen des Künstlers ein.
- Die Applikation listet Künstler auf, die zu Ihrer Suchanfrage passen.

- Wählen Sie einen Künstler aus, um detaillierte Informationen wie Genre, Popularität und verfügbare Tracks oder Alben zu sehen.

Diese Schritte demonstrieren, wie Sie die Kernfunktionalitäten der Applikation nutzen können, um Musikinhalte auf Spotify effizient zu durchsuchen und relevante Informationen zu erhalten.

Technischer Überblick:

Die vorgestellte Spotify-Client-Applikation verwendet eine Reihe von Technologien und Frameworks, die jeweils aus spezifischen Gründen ausgewählt wurden, um die Entwicklung und die Funktionalität der Applikation zu unterstützen. Hier sind die wichtigsten Technologien, die in diesem Projekt zum Einsatz kommen:

1. C# und .NET Core

- Einsatz: Die gesamte Applikation ist in C# geschrieben und nutzt das .NET Core Framework für die Entwicklung.
- Begründung: C# ist eine vielseitige, objektorientierte Programmiersprache, die starke Typisierung, Sicherheit und Leistung bietet. .NET Core ist ein Open-Source-Framework von Microsoft, das plattformübergreifend läuft, was die Entwicklung von Applikationen ermöglicht, die auf verschiedenen Betriebssystemen wie Windows, Linux und macOS funktionieren. Die Wahl fiel auf diese Technologie wegen ihrer robusten Entwicklungsumgebung, umfangreichen Bibliotheken und Unterstützung für asynchrone Programmierung, was besonders nützlich ist, wenn es um die Handhabung von Web-API-Anfragen geht.

2. HttpClient

- Einsatz: Für den Zugriff auf die Spotify Web API verwendet die Applikation die `HttpClient` Klasse von .NET.
- Begründung: `HttpClient` ist speziell für das Senden von HTTP-Anfragen und den Empfang von HTTP-Antworten konzipiert. Es bietet eine einfache und effiziente Möglichkeit, mit Webdiensten zu kommunizieren, unterstützt asynchrone

Operationen und ermöglicht so eine nicht-blockierende Kommunikation mit der Spotify API.

3. JSON (JavaScript Object Notation)

- Einsatz: JSON wird für den Datenaustausch zwischen der Applikation und der Spotify Web API verwendet.
- Begründung: JSON ist ein leichtgewichtiges Datenformat, das leicht zu lesen und zu schreiben ist. Es ist ideal für den Datenaustausch zwischen Servern und Web-Applikationen. Die Spotify Web API gibt Daten in JSON-Format zurück, was die Verarbeitung und Serialisierung/Deserialisierung der Daten in .NET-Objekte erleichtert.

4. Microsoft Extensions (Dependency Injection, Configuration, Logging)

- Einsatz: Die Applikation nutzt die Dependency Injection (DI), Configuration und Logging Funktionen aus den Microsoft Extensions.
- Begründung: DI erleichtert das Management von Abhängigkeiten zwischen den Klassen und fördert die lose Kopplung und bessere Testbarkeit. Configuration ermöglicht das einfache Verwalten von Anwendungseinstellungen, wie API-Schlüssel und Datenbankverbindungen. Logging bietet die Möglichkeit, Informationen über den Betrieb der Applikation zu protokollieren, was für die Fehlersuche und Überwachung nützlich ist.

5. Spotify Web API

- Einsatz: Die Applikation interagiert mit der Spotify Web API, um Daten wie Tracks, Alben, Künstler und Playlists zu suchen und abzurufen.
- Begründung: Die Spotify Web API ist eine leistungsstarke Schnittstelle, die Entwicklern Zugriff auf umfangreiche Musikdatenbanken und Funktionen von Spotify bietet. Durch die Nutzung der API kann die Applikation die umfangreichen Musikressourcen von Spotify erschließen, ohne eine eigene Datenbank pflegen zu müssen.

Clean Architecture:

Was ist Clean Architecture?

Clean Architecture ist ein Architekturstil, der darauf abzielt, Software so zu strukturieren, dass sie leicht wartbar, testbar, flexibel und unabhängig von spezifischen Frameworks und Technologien ist.

Die Software wird in klar definierte Schichten unterteilt, die jeweils unabhängig voneinander sind und klare Abhängigkeitsrichtungen haben, wobei die Abhängigkeiten nur nach innen gerichtet sind. Die Business-Logik, also der Kern der Anwendung, ist von den anderen Schichten isoliert und kommuniziert über wohldefinierte Schnittstellen.

Diese Architektur hat klare Vorteile: Die Software ist leichter wartbar, da Änderungen in einer Schicht die anderen nicht beeinflussen. Auch das Testen gestaltet sich einfacher, da die Schichten separat getestet werden können. Flexibilität wird durch die Unabhängigkeit der Schichten ermöglicht. Dies erleichtert den Austausch einzelner Komponenten. Zudem ist die Software nicht an bestimmte Frameworks oder Technologien gebunden, da die Business-Logik im Kern der Architektur liegt.

Natürlich hat Clean Architecture auch seine Nachteile.

Die Komplexität der Architektur kann höher sein als bei anderen Ansätzen, was den Lernaufwand erhöhen kann.

Insgesamt ist Clean Architecture ein robuster Architekturstil, der die Entwicklung von wartbarer, testbarer, flexibler und unabhängiger Software unterstützt.

Analyse der Dependency Rule

Die Dependency Rule, oft im Kontext von Softwarearchitektur und Design, insbesondere bei der Clean Architecture, diskutiert, ist ein Prinzip, das besagt, dass Abhängigkeiten innerhalb eines Systems immer von außen nach innen gerichtet sein sollten, bezogen auf die Schichten der Architektur. Das bedeutet, dass die inneren Schichten eines Systems keine Kenntnis von oder Abhängigkeiten zu den äußeren Schichten haben sollten. Stattdessen sollten die äußeren Schichten (die häufig die Benutzeroberfläche, Datenbankzugriffe oder externe Systemintegrationen beinhalten) von den inneren Schichten (die die Geschäftslogik oder Anwendungsregeln beinhalten) abhängig sein.

Dieses Prinzip unterstützt die Erstellung von Systemen, die leichter zu warten, zu testen und zu erweitern sind, da es hilft, eine klare Trennung der Verantwortlichkeiten zu

bewahren und die Kopplung zwischen den verschiedenen Teilen eines Systems zu minimieren.

Die Dependency Rule fördert die Umsetzung der Inversion of Control (IoC) und Dependency Injection (DI) Techniken, um Abhängigkeiten zu verwalten und zu entkoppeln, was die Entwicklung von modulareren und flexibleren Systemen ermöglicht.

In der Praxis bedeutet das Anwenden der Dependency Rule, dass die inneren Schichten durch Interfaces oder abstrakte Klassen definiert werden können, und die Implementierungen dieser Abstraktionen in den äußeren Schichten angesiedelt sind. Dies ermöglicht es den inneren Schichten, unabhängig von den spezifischen Implementierungsdetails der äußeren Schichten zu funktionieren.

Positiv-Beispiel:

Einhaltung der Dependency Rule

SearchTrackUseCase (Einhaltung):

- **Abhängigkeiten von der Klasse:** SearchTrackUseCase hängt von ISearchRepository ab, einer Schnittstelle, die zur inneren Schicht gehört und in der Repository Interfaces definiert ist.
- **Abhängigkeiten zu der Klasse:** Diese Klasse wird von der Controller-Schicht und möglicherweise von Präsentationsschichten verwendet, um Tracks zu suchen. Das passt zur Clean Architecture, da die Geschäftslogik (Use Cases) von den äußeren Schichten genutzt wird, aber selbst nur von inneren Schnittstellen (nicht von deren Implementierungen) abhängt.

```
public class SearchTrackUseCase : ISearchTrackUseCase
{
    private readonly ISearchRepository _searchTrackRepository;
    public SearchTrackUseCase(ISearchRepository searchTrackRepository)
    {
        _searchTrackRepository = searchTrackRepository;
    }
    public async Task<List<Track>> ExecuteAsync(string trackName)
    {
        return await _searchTrackRepository.SearchTracksAsync(trackName);
    }
}
```

Negativ-Beispiel:

Verletzung der Dependency Rule

SpotifyCredentials (Mögliche Verletzung):

- Abhängigkeiten von der Klasse: SpotifyCredentials hängt direkt von der Infrastruktur ab (`HttpClient`), was in Ordnung ist, da es sich hierbei um eine Infrastrukturklasse handelt. Jedoch könnte die direkte Abhängigkeit zu externen Services oder spezifischen Implementierungsdetails in anderen Kontexten als Verletzung betrachtet werden, wenn solche Klassen von den inneren Schichten (wie Use Cases oder Entities) verwendet würden.
- Abhängigkeiten zu der Klasse: Diese Klasse wird vom `SpotifyAdapter` verwendet, der wiederum von den Interface Adapters verwendet wird. In diesem spezifischen Kontext gibt es keine direkte Verletzung der Dependency Rule, da SpotifyCredentials eine Unterstützungsklasse für den Zugriff auf externe Services darstellt und in der Infrastrukturschicht verwendet wird. Jedoch wäre eine Verwendung dieser Klasse direkt von den Use Cases oder Entities eine Verletzung.

```
public class SpotifyCredentials
{
    private readonly HttpClient _httpClient;
    private readonly string _clientId;
    private readonly string _clientSecret;
    // Konstruktor und Methoden...
}
```

In diesem Beispiel hält `SearchTrackUseCase` die Dependency Rule ein, da es von abstrakten Schnittstellen und nicht von konkreten Implementierungen abhängt. SpotifyCredentials könnte in einem anderen Kontext eine Verletzung darstellen, wenn es direkt von Geschäftslogiken abhängig gemacht würde, was hier jedoch nicht der Fall ist, da es sich um eine unterstützende Infrastrukturklasse handelt, die innerhalb ihrer eigenen Schicht verwendet wird. Generell ist es wichtig, die Abstraktionsebenen und die Richtung der Abhängigkeiten im Auge zu behalten, um die Prinzipien der Clean Architecture einzuhalten.

Analyse der Schichten

1. Geschäftslogik-Schicht (Entities Layer): Track

Beschreibung der Aufgabe: Die `Track`-Klasse repräsentiert einen Musiktitel mit Eigenschaften wie `Id`, `Name`, `DurationMin` (Dauer), `Explicit` (expliziter Inhalt oder nicht), `PreviewUrl` (URL zur Hörprobe), `Artists` (eine Liste von Künstlern, die den Track erstellt haben), `Album` (das Album, auf dem der Track erscheint), `Popularity` (Popularität) und `TrackNumber` (die Nummer des Tracks auf dem Album). Diese Eigenschaften erfassen die wesentlichen Informationen und Charakteristika eines Musiktitels im Kontext der Anwendung.

Einordnung in die Clean Architecture: Die Klasse `Track` ist Teil der Entities-Schicht. Diese Schicht beinhaltet die Geschäftsobjekte der Anwendung, die unabhängig von der Benutzeroberfläche, Datenbank oder anderen externen Agenten sind. Die Entities definieren die Geschäftsregeln und die grundlegenden Konzepte der Anwendung. Die `Track`-Klasse ist ein zentrales Element der Geschäftslogik des Spotify Clients, da sie die grundlegenden Daten eines Musikstücks kapselt, mit denen die Anwendung arbeitet. Ihre Positionierung in der Geschäftslogik-Schicht ermöglicht es, dass sie von allen anderen Schichten referenziert werden kann, ohne dass sie Abhängigkeiten zu diesen aufweist, was der Unabhängigkeitsforderung der Clean Architecture entspricht.

2. Anwendungsfall-Schicht (Use Cases Layer):

`SearchTrackUseCase`

Beschreibung der Aufgabe: `SearchTrackUseCase` ist für die Ausführung des Anwendungsfalls "Suche nach Tracks" verantwortlich. Diese Klasse nimmt einen

Track-Namen als Eingabe entgegen und interagiert mit der Infrastruktur- oder Adapter-Schicht, um die Suche nach dem entsprechenden Track durchzuführen. Das Ergebnis ist eine Liste von `Track`-Objekten, die den Suchkriterien entsprechen. `SearchTrackUseCase` kapselt die spezifische Geschäftslogik für diesen Anwendungsfall und stellt eine klare Schnittstelle für die Ausführung der Suche bereit.

Einordnung in die Clean Architecture: `SearchTrackUseCase` ist ein Bestandteil der Use Cases-Schicht. In der Clean Architecture ist diese Schicht zuständig für die Implementierung der Anwendungsfälle, die die Geschäftsregeln und die Interaktionen des Systems beschreiben. Die Use Cases-Schicht agiert als Vermittler zwischen der Präsentationsschicht (z.B. UI) und der Geschäftslogik (Entities) und steuert den Datenfluss zwischen diesen. Durch die Fokussierung auf einen spezifischen Anwendungsfall wie die Track-Suche ermöglicht `SearchTrackUseCase` eine klare Abgrenzung der Geschäftslogik von den technischen Details der Datenbeschaffung und präsentiert eine saubere Schnittstelle zur Ausführung von Aktionen, die für den Endbenutzer von Bedeutung sind. Dies fördert die Wartbarkeit und Testbarkeit des Codes, da Anwendungsfälle in dieser Schicht unabhängig von UI-Logik oder Datenpersistenz implementiert werden

SOLID

Analyse SRP

Das Single Responsibility Principle (SRP) ist eines der fünf SOLID-Prinzipien der objektorientierten Programmierung und Design. Es besagt, dass eine Klasse nur einen Grund haben sollte, sich zu ändern. Das bedeutet, dass eine Klasse nur eine Aufgabe oder Verantwortung haben sollte. Dieses Prinzip hilft dabei, die Software wartbarer, verständlicher und flexibler zu gestalten.

Positives Beispiel: SpotifyCredentials

Beschreibung der Aufgabe: Die `SpotifyCredentials`-Klasse im Infrastructure-Layer hat die klare und einzige Verantwortung, die Authentifizierung mit der Spotify API zu

verwalten. Sie kümmert sich um das Generieren des Access Tokens, das für die Authentifizierung bei API-Anfragen notwendig ist, und stellt sicher, dass dieses Token gültig bleibt, indem sie es bei Bedarf erneuert. Diese Singularität der Aufgabe macht sie zu einem guten Beispiel für die Anwendung des SRP.

Negatives Beispiel: Controller-Klasse in CMDSpotifyClientCleanArchitecture.Controller

Beschreibung der Aufgaben: Die `Controller`-Klasse orchestriert verschiedene Use Cases wie die Suche nach Tracks, Alben, Künstlern und Playlists sowie das Abrufen von Details zu diesen Objekten. Obwohl es auf den ersten Blick scheinen mag, dass die Klasse eine einzige Verantwortung hat (die Orchestrierung der Use Cases), ist sie tatsächlich für die Koordination verschiedener Arten von Aktionen zuständig, die in unterschiedliche Kategorien fallen (Suche und Abruf). Dies kann dazu führen, dass die Klasse aus verschiedenen Gründen geändert werden muss – sei es wegen Änderungen in der Art und Weise, wie Suchanfragen verarbeitet werden, oder Anpassungen in der Logik des Abrufens von Details –, was sie zu einem Beispiel für die Verletzung des SRP macht.

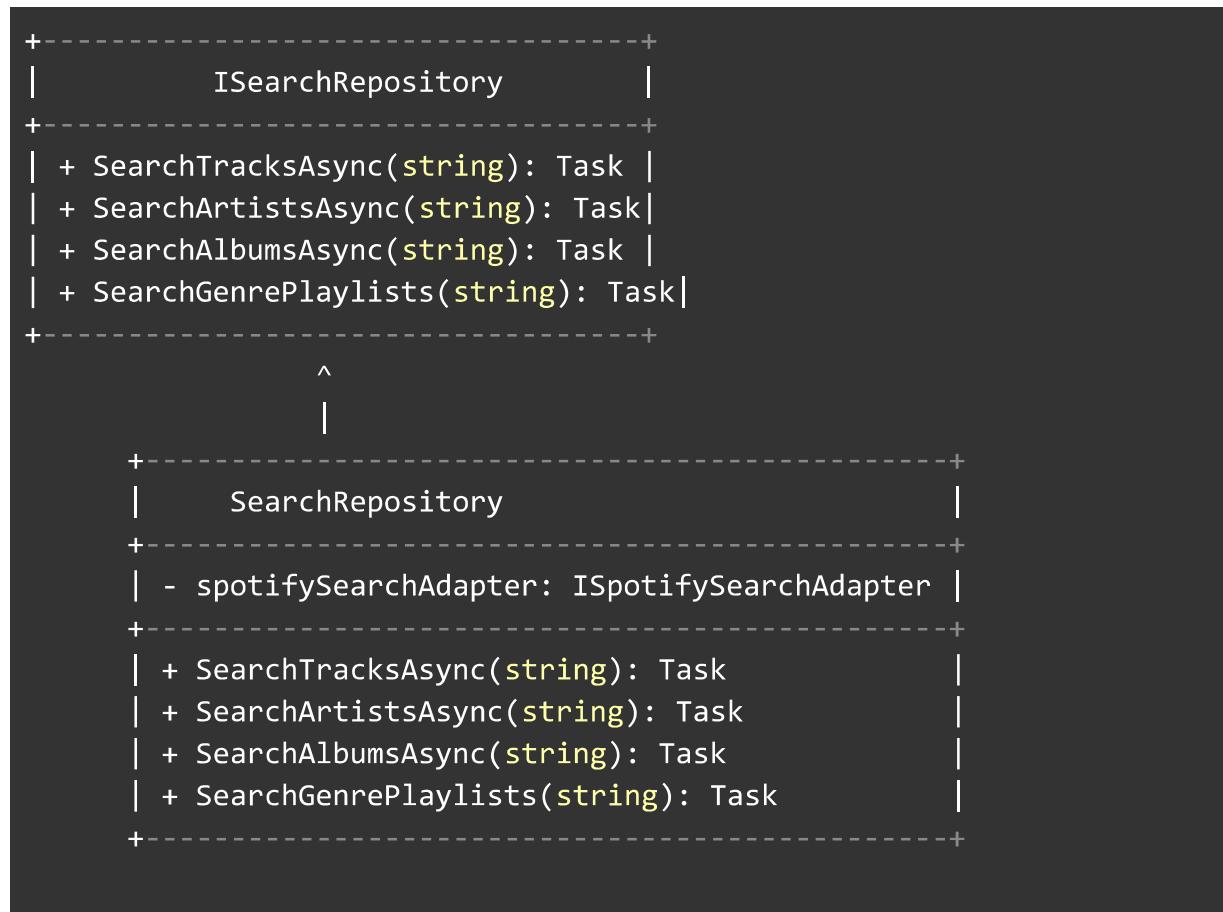
Möglicher Lösungsweg: Eine mögliche Lösung zur Behebung des Verstoßes gegen das SRP in der `Controller`-Klasse wäre, sie in spezifischere Controller aufzuteilen, die jeweils eine einzige Art von Aktion verwalten. Zum Beispiel könnte man separate Controller für die Suchfunktionen und das Abrufen von Details haben. Auf diese Weise wäre jeder Controller nur aus einem einzigen Grund zu ändern, was das SRP erfüllt. Dies könnte auch die Wartbarkeit und Erweiterbarkeit des Codes verbessern, da Änderungen in einem spezifischen Bereich der Anwendung isoliert von anderen durchgeführt werden könnten.

Analyse OCP

Das Open/Closed Principle (OCP) ist eines der SOLID-Prinzipien der objektorientierten Programmierung, das besagt, dass Softwareentitäten (Klassen, Module, Funktionen usw.) für Erweiterung offen, aber für Modifikation geschlossen sein sollten. Das bedeutet, dass es möglich sein sollte, das Verhalten einer Einheit zu erweitern, ohne den Code zu ändern. OCP fördert die Verwendung von Schnittstellen oder abstrakten Klassen, um flexible und erweiterbare Architekturen zu erstellen.

Positives Beispiel: `ISearchRepository`-Interface

UML-Darstellung:

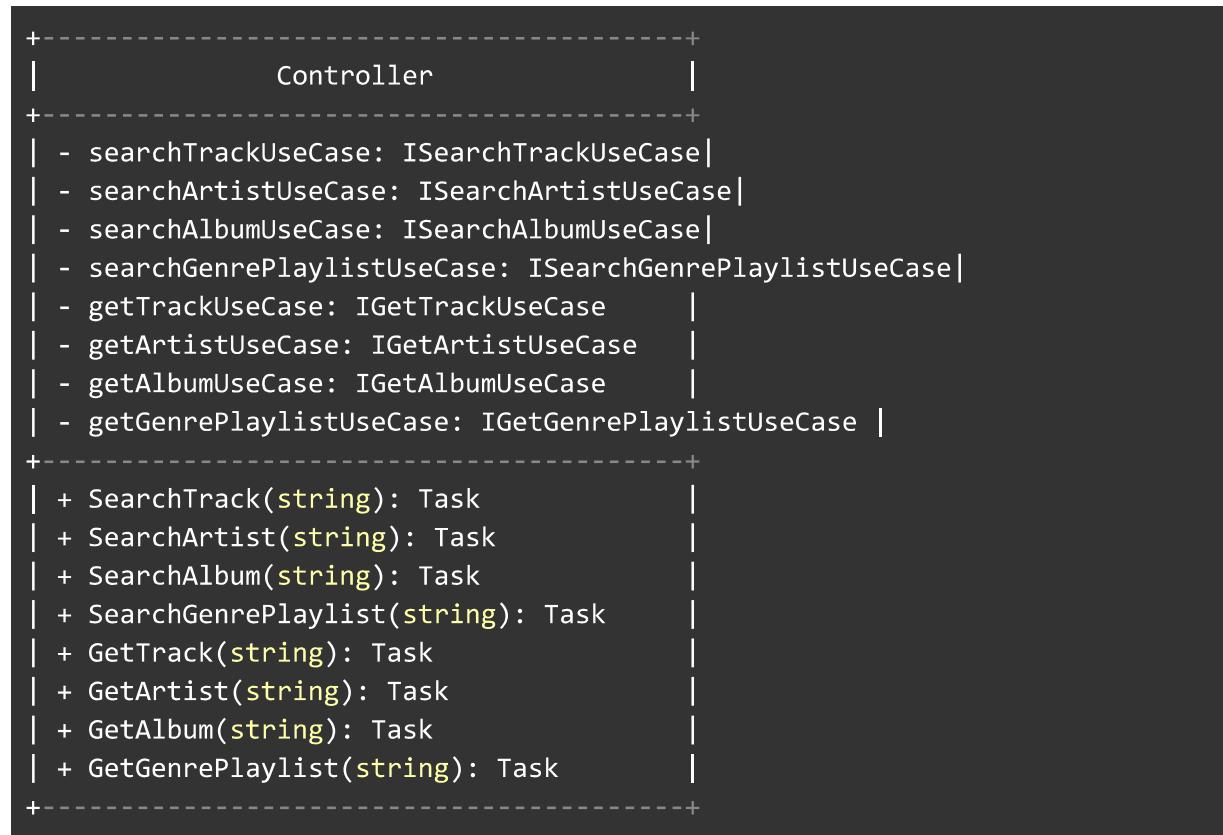


Analyse:

Das Interface `ISearchRepository` erfüllt das OCP, da es die Möglichkeit bietet, sein Verhalten durch Implementierung des Interfaces zu erweitern, ohne dass das Interface selbst modifiziert werden muss. Entwickler können verschiedene Implementierungen dieses Interfaces erstellen, um beispielsweise Suchfunktionen über verschiedene Datenquellen (wie Spotify, Apple Music oder lokale Datenbanken) bereitzustellen, ohne die Definition des Interfaces oder den Code, der es verwendet, ändern zu müssen. Dieser Ansatz ist sinnvoll, da er die Flexibilität erhöht und es ermöglicht, die Datenquelle der Suchfunktionen einfach auszutauschen oder zu erweitern, ohne bestehenden Code zu beeinträchtigen.

Negatives Beispiel: Controller-Klasse in CMDSpotifyClientCleanArchitecture.Controller

UML-Darstellung:



Analyse:

Die `Controller`-Klasse verletzt das OCP, da jede Erweiterung der Funktionalität (z.B. das Hinzufügen neuer Suchtypen oder neuer Informationsabfragen) eine Modifikation der `Controller`-Klasse selbst erfordern würde. Das bedeutet, dass die Klasse für Erweiterungen nicht geschlossen ist, da sie direkt geändert werden muss, um neue Funktionalitäten hinzuzufügen.

Lösung:

Um das OCP zu erfüllen, könnte man eine Strategie oder ein Command-Pattern verwenden, bei dem unterschiedliche Aktionen als separate Objekte oder Strategien implementiert werden, die alle ein gemeinsames Interface implementieren. Der

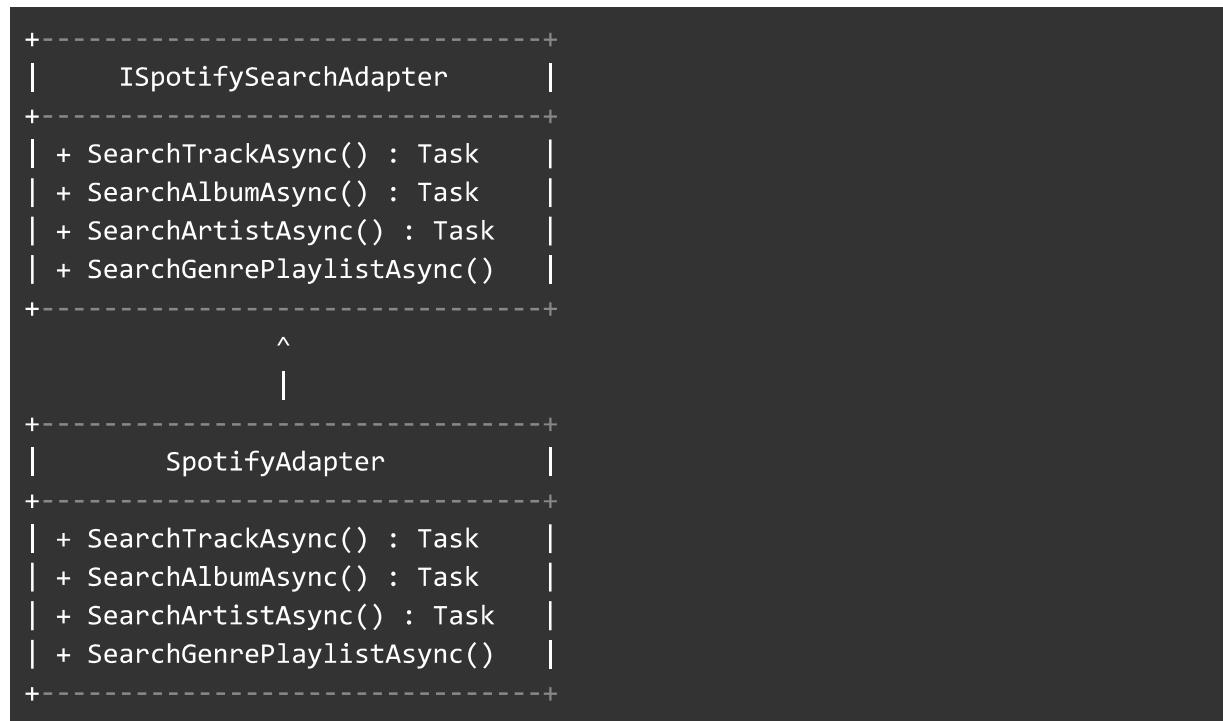
`Controller` könnte dann eine Sammlung dieser Strategien halten und zur Laufzeit die entsprechende Strategie basierend auf der Anfrage auswählen. Auf diese Weise würde das Hinzufügen neuer Funktionalitäten einfach das Hinzufügen neuer Strategie-Implementierungen erfordern, ohne die `Controller`-Klasse selbst zu ändern.

Analyse [LSP/(ISP/DIP)]

Liskov Substitutionsprinzip (LSP) besagt, dass Objekte einer Basisklasse durch Objekte einer abgeleiteten Klasse ersetzt werden können, ohne die Korrektheit des Programms zu beeinträchtigen. Das Prinzip zielt auf die Wahrung der Verhaltenskompatibilität zwischen Basisklassen und abgeleiteten Klassen ab.

Positives Beispiel: `ISpotifySearchAdapter`

UML-Diagramm:



Begründung:

Die `SpotifyAdapter`-Klasse implementiert das Interface `ISpotifySearchAdapter` und erfüllt das LSP, da sie die Verträge des Interfaces ohne Änderung der erwarteten Funktionalität implementiert. Clients, die `ISpotifySearchAdapter` verwenden, können

problemlos Instanzen von `SpotifyAdapter` verwenden, ohne Einfluss auf die Programmkorrektur, was eine korrekte Anwendung des LSP darstellt.

Negatives Beispiel: Fehlende explizite Klasse im bereitgestellten Code

Für eine direkte Anwendung des LSP in der bereitgestellten Codebasis könnte ein negatives Beispiel konstruiert werden, indem man eine Basisklasse und eine abgeleitete Klasse erstellt, bei der die abgeleitete Klasse die Funktionalität der Basisklasse auf eine Weise ändert, die für den Client nicht transparent ist. Da jedoch keine spezifische Verletzung des LSP im bereitgestellten Code vorhanden ist, konstruieren wir ein hypothetisches Szenario, um das Konzept zu verdeutlichen.

```
+-----+
|      BaseClass      |
+-----+
| + DoWork() : void  |
+-----+
          ^
          |
+-----+
|      DerivedClass   |
+-----+
| + DoWork() : void  |
+-----+
```

Angenommen, `DoWork` in `BaseClass` hat ein bestimmtes Verhalten, das in `DerivedClass` so geändert wird, dass es die Erwartungen der Clients verletzt (z.B. durch eine Ausnahme, wenn eine bestimmte Bedingung erfüllt ist, die in der Basisklasse nicht vorhanden war).

Mögliche Lösung:

Um das LSP zu erfüllen, sollte `DerivedClass` das Verhalten von `DoWork` so erweitern oder modifizieren, dass alle Vor- und Nachbedingungen der Basisklasse erhalten bleiben. Sollte das nicht möglich sein, weil `DerivedClass` ein grundlegend anderes Verhalten benötigt, könnte dies ein Hinweis darauf sein, dass eine Ableitung in diesem

Fall nicht angebracht ist und stattdessen eine andere Hierarchie oder ein Designmuster (wie Strategie oder Dekorierer) in Betracht gezogen werden sollte.

Weitere Prinzipien

Analyse GRASP: Geringe Kopplung

Geringe Kopplung zielt darauf ab, die Abhängigkeiten zwischen Modulen so gering wie möglich zu halten, um die Wartbarkeit und Flexibilität zu verbessern. Klassen sollten mit so wenigen anderen Klassen wie möglich interagieren, und die Beziehungen sollten so einfach wie möglich gehalten werden.

SearchTrackUseCase - Positives Beispiel für geringe Kopplung

Aufgabenbeschreibung:

Die Klasse `SearchTrackUseCase` implementiert das Interface `ISearchTrackUseCase` und stellt die Geschäftslogik bereit, um nach Musiktracks über die Spotify API zu suchen. Ihre Hauptaufgabe ist es, Suchanfragen zu verarbeiten und Suchergebnisse von Spotify zurückzugeben, indem sie die entsprechende Methode `SearchTracksAsync` des `ISearchRepository` auuftut.

Begründung der geringen Kopplung:

- Spezifische Abhängigkeit: `SearchTrackUseCase` hängt lediglich von dem Interface `ISearchRepository` ab, nicht von konkreten Implementierungen. Dies verringert die Kopplung, da `SearchTrackUseCase` unabhängig von den spezifischen Datenzugriffsdetails funktionieren kann.
- Interface-Abhängigkeit: Durch die Abhängigkeit von einem Interface statt von einer konkreten Klasse wird die Kopplung weiter reduziert. Änderungen in der Implementierung des Repositories haben keinen direkten Einfluss auf `SearchTrackUseCase`, solange das Interface `ISearchRepository` unverändert bleibt.
- Einzelaufgabe: Die Klasse konzentriert sich auf eine einzige Aufgabe – die Suche nach Tracks. Dies reduziert ihre Abhängigkeit von anderen Komponenten des

Systems und macht sie weniger anfällig für Änderungen in anderen Bereichen der Anwendung.

- **Einfache Kommunikation:** Die Kommunikation zwischen `SearchTrackUseCase` und dem `ISearchRepository` erfolgt über einfache, wohldefinierte Schnittstellenmethoden. Diese klare Trennung der Verantwortlichkeiten hält die Kopplung niedrig.

Aufrufer/Nutzer der Klasse:

Aufrufer oder Nutzer von `SearchTrackUseCase` interagieren mit dieser Klasse durch das definierte Interface `ISearchTrackUseCase`. Dies könnte beispielsweise eine Controller-Schicht in einer Web-Anwendung sein, die die Use-Case-Klasse nutzt, um Suchanfragen zu verarbeiten und die Ergebnisse an die Benutzeroberfläche weiterzuleiten. Die Tatsache, dass die Nutzer der Klasse nur mit ihr über ein Interface interagieren, trägt ebenfalls zur Verringerung der Kopplung bei, da Änderungen innerhalb der `SearchTrackUseCase`-Implementierung transparent für die Nutzer sind, solange das Interface konstant bleibt.

Durch diese Strukturierung wird die Kopplung minimiert, was die Pflege und Erweiterung des Systems erleichtert.

Analyse GRASP: [Polymorphismus/Pure Fabrication]

Um das Prinzip von Pure Fabrication mit einem anderen Beispiel zu illustrieren, betrachten wir nun die Klasse `SearchRepository` aus dem bereitgestellten Code. Diese Klasse dient als Brücke zwischen der Anwendungslogik und den Daten, die von der Spotify-API zurückgegeben werden, und ist ein gutes Beispiel für Pure Fabrication, da sie hauptsächlich aus technischen Gründen existiert, um eine klare Trennung zwischen der Geschäftslogik (Use Cases) und den Details der Datenerfassung zu gewährleisten.

SearchRepository - Positives Beispiel für Pure Fabrication

Begründung des Einsatzes von Pure Fabrication:

- **Separation of Concerns:** `SearchRepository` abstrahiert die Logik, die notwendig ist, um Suchanfragen an die Spotify API zu stellen und die Ergebnisse zu verarbeiten. Diese Abstraktion erlaubt es den Use Cases, sich auf die

Anwendungslogik zu konzentrieren, ohne sich um die Details der Datenbeschaffung zu kümmern.

- Förderung der Wiederverwendbarkeit: Durch die Kapselung der Datenzugriffslogik in einer separaten Klasse kann diese Logik leicht wiederverwendet werden, ohne sie in jeder Use Case-Klasse duplizieren zu müssen. Dies verbessert die Wiederverwendbarkeit und vereinfacht die Wartung.
- Erleichterte Testbarkeit: Die Trennung der Datenbeschaffungslogik in `SearchRepository` macht es einfacher, Unit-Tests für die Geschäftslogik zu schreiben, da die Abhängigkeit von der externen API durch Mocking oder Stubbing des Repositories isoliert werden kann. Dies verbessert die Testbarkeit des Systems.
- Reduzierung der Kopplung: Indem `SearchRepository` als Schnittstelle zwischen der Geschäftslogik und externen Diensten fungiert, reduziert es die Kopplung zwischen diesen Komponenten. Änderungen an der API oder an der Art und Weise, wie Daten abgerufen werden, beeinflussen nicht direkt die Geschäftslogik, solange das Repository-Interface konstant bleibt.

```
public class SearchRepository : ISearchRepository
{
    private readonly ISpotifySearchAdapter _spotifySearchAdapter;

    public SearchRepository(ISpotifySearchAdapter spotifySearchAdapter)
    {
        _spotifySearchAdapter = spotifySearchAdapter;
    }

    public async Task<List<Track>> SearchTracksAsync(string trackName)...
    public async Task<List<Artist>> SearchArtistsAsync(string artistName)...
    public async Task<List<Album>> SearchAlbumsAsync(string albumName)...
    public async Task<List<Playlist>> SearchGenrePlaylistsAsync(string genreName)...
}
```

DRY //Machen

Unit Tests

10 Unit Tests

Diese Tests in Meinem Projekt demonstrieren, wie verschiedene Aspekte eines Spotify-Clientprojekts mit Clean Architecture und .NET getestet werden können. Sie nutzen das Moq-Framework, um Abhängigkeiten zu mocken und verifizieren das Verhalten der getesteten Klassen. Hier ist eine Zusammenfassung der Tests und ihre jeweilige Funktionsweise:

ControllerTests

- `SearchTrack_ReturnsExpectedTracks`: Testet die `SearchTrack`-Methode im Controller. Es wird überprüft, ob der Controller die `ExecuteAsync`-Methode des `ISearchTrackUseCase` mit dem richtigen Parameter aufruft und die erwarteten Tracks zurückgibt. Der Test stellt sicher, dass die Abstraktion zwischen Controller und Use Case korrekt funktioniert.

RetrievalRepositoryTests

- `GetTrackAsync_ReturnsDetailedTrackInformation`: Überprüft, ob `GetTrackAsync` die richtigen Track-Details von der Spotify API abruft und korrekt als Track-Objekt zurückgibt. Es wird auch geprüft, ob der Adapter genau einmal aufgerufen wird.
- `GetArtistAsync_ReturnsDetailedArtistInformation`: Ähnlich wie beim Track-Test, wird hier geprüft, ob die Details eines Künstlers korrekt abgerufen und zurückgegeben werden.
- `GetAlbumAsync_ReturnsDetailedAlbumInformation`: Testet, ob Albuminformationen korrekt von Spotify abgerufen und als Album-Objekt zurückgegeben werden.
- `GetGenrePlaylistAsync_ReturnsDetailedPlaylistInformation`: Überprüft, ob Playlist-Informationen basierend auf einem Genre korrekt abgerufen und als Playlist-Objekt zurückgegeben werden.

SearchRepositoryTests

- SearchTrackAsync_ReturnsCorrectTracks: Verifiziert, dass die Suche nach Tracks über den SpotifySearchAdapter korrekt funktioniert und die erwarteten Track-Objekte zurückgibt.
- SearchArtistAsync_ReturnsCorrectArtists: Testet die Künstlersuche und überprüft, ob die richtigen Künstler-Objekte zurückgegeben werden.
- SearchAlbumsAsync_ReturnsCorrectAlbums: Überprüft, ob die Suche nach Alben die korrekten Album-Objekte zurückgibt.
- SearchPlaylistsAsync_ReturnsCorrectPlaylists: Testet, ob die Playlist-Suche basierend auf einem Genre die erwarteten Playlist-Objekte zurückliefert.

SearchTrackUseCaseTests

- ExecuteAsync_CallsSearchRepository_WithCorrectParameters: Überprüft, ob ExecuteAsync des SearchTrackUseCase das SearchRepository mit den korrekten Parametern aufruft und die Suchergebnisse korrekt zurückgibt. Dieser Test stellt sicher, dass die Use Case-Schicht wie erwartet funktioniert und ihre Abhängigkeiten korrekt nutzt.

In jedem dieser Tests werden Mock-Objekte verwendet, um die Abhängigkeiten der getesteten Klasse zu isolieren. Dadurch kann spezifisches Verhalten simuliert und die Interaktion zwischen Komponenten ohne die Notwendigkeit echter Implementierungen oder externe Anfragen überprüft werden. Dies erleichtert die Validierung der Geschäftslogik unter kontrollierten Bedingungen.

ATRIP: Automatic

Die gezeigten Testklassen demonstrieren eine umfassende Herangehensweise zur Überprüfung der Funktionalität eines hypothetischen Spotify-Clients mithilfe des "Automatic Test Run in Parallel" (ATRIP)-Prinzips. Das ATRIP-Prinzip ermöglicht es, Tests automatisch und parallel auszuführen, wodurch die Testlaufzeiten signifikant verkürzt werden können. Im Kontext dieser Tests wird "Automatic" durch die Verwendung von Mock-Objekten und asynchronen Testmethoden realisiert, die mit der NUnit-Testbibliothek zusammenarbeiten. Hier ist, wie dies erreicht wird:

Verwendung von Mock-Objekten: Durch das Mocken der Abhängigkeiten (wie `ISearchTrackUseCase`, `ISearchArtistUseCase` etc.) können die Tests unabhängig von externen Systemen oder der tatsächlichen Implementierung

dieser Use Cases ausgeführt werden. Dies erlaubt eine isolierte Überprüfung der Logik innerhalb der getesteten Klassen, ohne dass externe Anfragen gestellt oder auf tatsächliche Daten zugegriffen wird. Die Mock-Objekte werden mithilfe des Moq-Frameworks erstellt, das es ermöglicht, Erwartungen zu definieren (z.B. dass eine Methode mit bestimmten Parametern aufgerufen wird) und Rückgabewerte für diese Aufrufe zu simulieren.

Asynchrone Testmethoden: Alle Tests sind als `async Task` definiert, was die parallele Ausführung von Tests, die auf asynchrone Operationen warten, erleichtert. Dies ist besonders wichtig, da der Spotify-Client vermutlich zahlreiche Netzwerkanfragen macht, deren Antworten asynchron sind. Die asynchrone Natur der Tests sorgt dafür, dass der Testthread nicht blockiert wird, während er auf Netzwerkanfragen oder andere asynchrone Operationen wartet. Dies verbessert die Gesamtperformance der Testausführung.

Assertion und Verifizierung: Jeder Test überprüft, ob die tatsächlichen Ergebnisse den erwarteten Ergebnissen entsprechen (durch Assertions) und ob die Mock-Objekte wie erwartet verwendet wurden (durch Verifizierungen). Dies stellt sicher, dass sowohl die Geschäftslogik korrekt funktioniert als auch die Integration zwischen den verschiedenen Komponenten wie vorgesehen ist.

Die Kombination dieser Techniken erlaubt es, Tests effizient und ohne externe Abhängigkeiten automatisch parallel auszuführen. Dies ist ein Kernaspekt des ATRIP-Prinzips, da es die Entwicklungs- und Wartungszeit verkürzt, indem es schnelles Feedback über die Funktionalität und die Integrität des Codes ermöglicht.

ATRIP: Thorough

Projekt	Version	Installiert
CMDSpotifyClientCleanArchitecture	6.0.2	6.0.2
CMDSpotifyClientCleanArchitecture.Tests	6.0.2	6.0.2

Installiert: 6.0.2 Deinstallieren

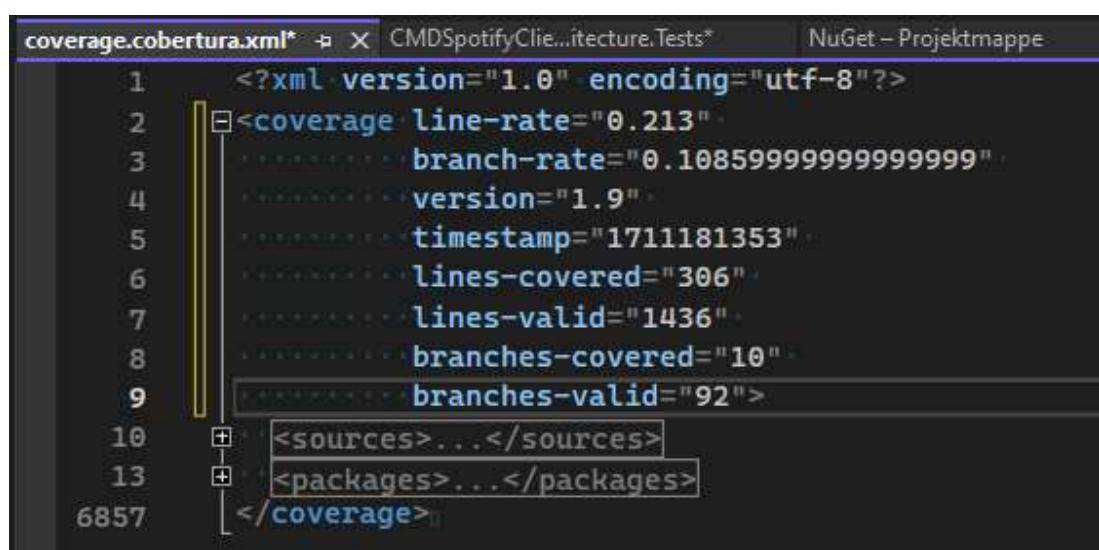
Version: Aktuellste stabile Version 6.0.2 Installieren

```
dotnet add package coverlet.msbuild
```

```
dotnet test --collect:"XPlat Code Coverage" --
DataCollectionRunSettings.DataCollectors.DataCollector.Configuration.Format
=cobertura --
DataCollectionRunSettings.DataCollectors.DataCollector.Configuration.Covera
geOutput='./TestResults/Coverage/'
```

```
dotnet tool install -g dotnet-reportgenerator-globaltool
```

```
reportgenerator "-reports:../TestResults/Coverage/coverage.cobertura.xml"
"-targetdir:coveragereport" -reporttypes:Html
```



```
coverage.cobertura.xml*  ▾ X  CMDSpotifyClientArchitecture.Tests*  NuGet – Projektmappe
1   <?xml version="1.0" encoding="utf-8"?>
2   □<coverage line-rate="0.213"·
3   ··· branch-rate="0.10859999999999999"·
4   ··· version="1.9"·
5   ··· timestamp="1711181353"·
6   ··· lines-covered="306"·
7   ··· lines-valid="1436"·
8   ··· branches-covered="10"·
9   ··· branches-valid="92">
10  □<sources>...</sources>
13  □<packages>...</packages>
6857  □</coverage>
```

Die Analyse der Codeabdeckung in dem Projekt zeigt verschiedene Metriken, die Aufschluss über den Grad der Testabdeckung geben. Die Gesamtdeckung von Codezeilen (line-rate) beträgt 21,3% und die Verzweigungsdeckung (branch-rate) 10,86%. Dies bedeutet, dass von insgesamt 1436 gültigen Codezeilen 306 getestet wurden und von 92 möglichen Verzweigungen 10 abgedeckt sind. Die Komplexität des Projekts wird mit 727 angegeben, was auf eine hohe Anzahl von möglichen Pfaden durch den Code hinweist, die beim Testen berücksichtigt werden sollten.

Die niedrige Testabdeckung, insbesondere bei den Verzweigungen, deutet darauf hin, dass viele mögliche Codepfade während der Tests nicht ausgeführt werden. Dies kann zu unentdeckten Fehlern führen, die sich negativ auf die Qualität und Zuverlässigkeit der Software auswirken. Es ist wichtig, die Testabdeckung zu erhöhen, um sicherzustellen, dass sowohl die Hauptpfade als auch die Randfälle im Code ausreichend geprüft werden.

ATRIP: Professional

Positive Beispiel: Dependency Injection und Mocking in Testfällen

Im vorgelegten Code wird die Technik des Mockings verwendet, um Abhängigkeiten in den Unit-Tests zu isolieren. Die Verwendung von `Mock<T>` aus dem Moq-Framework in den Testfällen für den `Controller` und die `Repositories` demonstriert professionelle Softwareentwicklungspraktiken, indem sie sicherstellt, dass die Tests unabhängig von externen Services wie einer Datenbank oder einer externen API (in diesem Fall Spotify's Web API) durchgeführt werden können.

Analyse und Begründung:

- Isolierung von Tests: Durch das Ersetzen der tatsächlichen Implementierungen der Use Cases und des Spotify Data Retrieval Adapters mit Mock-Objekten werden die Tests isoliert. Dies ermöglicht es, das Verhalten dieser Abhängigkeiten zu simulieren und zu verifizieren, dass die Komponenten wie erwartet mit den Mocks interagieren.
- Kontrolle über Testumgebung: Mocks geben volle Kontrolle über die Testumgebung. Testdaten und Verhaltensweisen der Abhängigkeiten können exakt definiert werden, was eine konsistente Testausführung gewährleistet.
- Verbesserung der Testeffizienz: Da externe Abhängigkeiten wie Netzwerkanfragen oder Datenbankabfragen umgangen werden, verbessert sich die Ausführungsgeschwindigkeit der Tests erheblich.

Professionalität: Diese Techniken reflektieren ein hohes Maß an Professionalität, da sie die Zuverlässigkeit, Wartbarkeit und Effizienz der Tests verbessern. Die Verwendung von Mocking-Frameworks wie Moq zeigt ein tiefes Verständnis moderner Softwareentwicklungsmethoden.

Fakes and Mocks

Um den Einsatz von Fakes und Mocks ohne die Verwendung einer Bibliothek oder eines Frameworks von Drittanbietern zu demonstrieren, könnte eine manuelle Implementierung der Mock-Objekte vorgenommen werden. Hier ein Beispiel:

Beispiel für einen Fake `ISearchTrackUseCase`:

```
csharp
Copy code
public class FakeSearchTrackUseCase : ISearchTrackUseCase
{
    private readonly List<Track> _tracks;

    public FakeSearchTrackUseCase(List<Track> tracks)
    {
        _tracks = tracks;
    }

    public Task<List<Track>> ExecuteAsync(string trackName)
    {
        // Simuliert das Finden von Tracks basierend auf dem Namen
        var result = _tracks.Where(t => t.Name.Contains(trackName)).ToList();
        return Task.FromResult(result);
    }
}
```

Verwendung des Fake in einem Test:

```
csharp
Copy code
[Test]
public async Task SearchTrack_UsesFakeAndReturnsExpectedResult()
{
    // Arrange
    var fakeTracks = new List<Track>
    {
        new Track { Name = "TestTrack1", Id = "1" },
        new Track { Name = "TestTrack2", Id = "2" }
    };
    var fakeSearchTrackUseCase = new FakeSearchTrackUseCase(fakeTracks);
    var controller = new ControllerProject(fakeSearchTrackUseCase, /* other dependencies */);

    // Act
    var result = await controller.SearchTrack("TestTrack1");

    // Assert
    Assert.That(result.Count, Is.EqualTo(1));
    Assert.That(result[0].Id, Is.EqualTo("1"));
}
```

Analyse und Begründung:

- Isolierung ohne externe Abhängigkeiten: Der Fake `ISearchTrackUseCase` erlaubt das Testen des `Controller's` Verhaltens in Bezug auf die Track-Suche, ohne eine echte Implementierung oder externe Abhängigkeiten zu benötigen.
- Volle Kontrolle und Einfachheit: Durch die Verwendung eines Fakes wird die Komplexität reduziert, die beim Einsatz eines Mocking-Frameworks entstehen kann. Es bietet eine einfache und kontrollierte Umgebung für spezifische Testfälle.
- Nachbildungen realer Szenarien: Fakes können dazu verwendet werden, verschiedene Szenarien und Datenbedingungen nachzubilden, um die Robustheit des Codes zu prüfen.

Der manuelle Ansatz für Fakes und Mocks, obwohl weniger flexibel und mit mehr Boilerplate-Code verbunden als Framework-basierte Lösungen, demonstriert ein tiefes Verständnis

Domain Driven Design

Ubiquitous Language

Domain Driven Design (DDD)

Domain Driven Design ist ein Ansatz für die Softwareentwicklung, der sich auf die Kernlogik und -regeln der Geschäftsdomäne konzentriert. Es legt den Schwerpunkt darauf, Komplexität in der Anwendungsdomäne zu verstehen und abzubilden, indem eine gemeinsame Sprache (Ubiquitous Language) für alle Stakeholder (Entwickler, Geschäftsanalysten, Tester, Kunden usw.) verwendet wird. Dies fördert das tiefe Verständnis der Anwendungsdomäne und erleichtert die Kommunikation.

Ubiquitous Language

Ubiquitous Language ist ein zentraler Bestandteil von Domain Driven Design und bezieht sich auf die gemeinsame Sprache, die von allen Teammitgliedern und Stakeholdern verwendet wird, um Konsistenz und Klarheit in der Kommunikation über die gesamte Anwendung und ihre Entwicklung hinweg zu gewährleisten. Die Ubiquitous Language umfasst Begriffe der Geschäftsdomäne und wird in Code, Spezifikationen und Diskussionen verwendet.

Beispiele für die Ubiquitous Language

Track

- Bezeichnung: Track
- Bedeutung: Ein einzelnes Musikstück.
- Begründung: Der Begriff "Track" ist direkt aus der Musikindustrie und Streaming-Dienste wie Spotify entnommen. Er wird einheitlich von Entwicklern, Geschäftsanalysten und Endbenutzern verwendet, um sich auf ein einzelnes Musikstück zu beziehen, wodurch er ein klarer Teil der Ubiquitous Language innerhalb einer auf Musik ausgerichteten Anwendung ist.

Artist

- Bezeichnung: Artist
- Bedeutung: Eine Person oder Gruppe, die Musik kreiert und aufführt.
- Begründung: "Artist" ist ein grundlegender Begriff in der Musikindustrie, der übergreifend von allen Beteiligten verstanden wird. In der Entwicklung einer Musik-Streaming-Anwendung stellt er eine zentrale Entität dar, die die Schöpfer von Musik repräsentiert.

Album

- Bezeichnung: Album
- Bedeutung: Eine Sammlung von Tracks, die von einem Künstler veröffentlicht wurden.
- Begründung: Der Begriff "Album" wird universell verwendet, um eine Sammlung von Musikstücken zu beschreiben, die als Einheit veröffentlicht werden. Er ist sowohl für Endbenutzer als auch für Entwickler intuitiv und fördert ein gemeinsames Verständnis der Struktur von Musikinhalten.

Playlist

- Bezeichnung: Playlist
- Bedeutung: Eine benutzerdefinierte Sammlung von Tracks.
- Begründung: Playlists sind ein zentrales Feature in Musik-Streaming-Diensten, die es Benutzern ermöglichen, ihre eigene Sammlung von Tracks zusammenzustellen. Der Begriff ist fest im Vokabular sowohl der Benutzer als auch der Entwickler verankert und spiegelt die Interaktivität und Personalisierung der Anwendung wider.

Repositories

Repositories sind im Kontext der Softwareentwicklung, insbesondere im Domain Driven Design (DDD), ein Muster, das eine Abstraktionsschicht zwischen der Domänenenschicht der Anwendung und der Datenspeicherschicht bietet. Diese Abstraktionsschicht ermöglicht es dem Domänenmodell und der Geschäftslogik, von den Details der Datenpersistenz (wie Datenbankschema oder Datenzugriffscodes) zu abstrahieren. Repositories fungieren als Sammlung von Aggregaten (Domänenobjekten), auf die über eine einheitliche Schnittstelle zugegriffen werden kann, um Erstellungs-, Lese-, Aktualisierungs- und Löschoperationen (CRUD) durchzuführen.

Beschreibung und Begründung des Einsatzes eines Repositories

Der Einsatz eines Repository-Musters bietet mehrere Vorteile:

- **Abstraktion:** Repositories kapseln die Logik für den Zugriff auf die Datenspeicherquelle. Dies erleichtert die Trennung der Geschäftslogik von den Details der Datenpersistenz, was zu einer saubereren, wartbareren Codebasis führt.
- **Austauschbarkeit:** Durch die Verwendung von Repositories als Zwischenschicht können die zugrunde liegenden Persistenzmechanismen (wie Datenbanken) leicht ausgetauscht oder modifiziert werden, ohne dass dies umfangreiche Änderungen am Geschäftscode erfordert.
- **Testbarkeit:** Repositories erleichtern das Mocking oder die Simulation von Datenquellen in Tests, was zu isolierteren und zuverlässigeren Tests führt.
- **Wiederverwendbarkeit:** Gemeinsame Datenzugriffsmuster und -operationen können in den Repository-Implementierungen zentralisiert werden, was die Wiederverwendung von Code fördert.

Implementierung

In dem vorgestellten Projekt wurden Repositories implementiert, um den Zugriff auf Spotify-Daten, wie Tracks, Artists, Albums und Playlists, zu verwalten. Hier ist ein Beispiel für ein einfaches Repository-Interface und eine dazugehörige Implementierung für das Track-Objekt:

```
csharp
```

```

Copy code
namespace CMDSSpotifyClient.Repository.Interfaces
{
    public interface ISearchRepository
    {
        Task<List<Track>> SearchTracksAsync(string trackName);
    }
}

public class SearchRepository : ISearchRepository
{
    private readonly ISpotifySearchAdapter _spotifySearchAdapter;

    public SearchRepository(ISpotifySearchAdapter spotifySearchAdapter)
    {
        _spotifySearchAdapter = spotifySearchAdapter;
    }

    public async Task<List<Track>> SearchTracksAsync(string trackName)
    {
        var jsonResult = await _spotifySearchAdapter.SearchTrackAsync(trackName);
        // Deserialisieren und Umwandeln des JSON-Resultats in eine Liste von Track-Objekten
        // und Rückgabe der Liste
    }
}

```

Die `SearchRepository`-Klasse implementiert das `ISearchRepository`-Interface und verwendet einen Spotify-Suchadapter (`ISpotifySearchAdapter`), um Daten von Spotify abzurufen. Diese Abstraktion ermöglicht es der Geschäftslogik, die Suche nach Tracks durchzuführen, ohne sich direkt mit der Spotify API verbinden zu müssen. Die Repository-Implementierung verbirgt die Komplexität des Datenabrufs und stellt eine klar definierte Schnittstelle für die Geschäftslogik zur Verfügung.

Aggregates

Aggregates sind eine Sammlung von Objekten, die als eine Einheit für Datenänderungen behandelt werden. In Domain-Driven Design (DDD) bilden Aggregates die Grenzen für Konsistenz und Transaktionen. Ein Aggregate besteht aus einem Root-Entity (Aggregate Root), das die einzige Zugriffspunkte für Änderungen innerhalb des Aggregates ist, und möglicherweise aus mehreren verbundenen Entities und Value Objects, die intern im Aggregate verwaltet werden.

Beschreibung und Begründung des Einsatzes eines Aggregates

Der Einsatz von Aggregates dient mehreren Zwecken:

- Konsistenzsicherung: Aggregates definieren klare Grenzen, innerhalb derer die Datenkonsistenz streng sichergestellt wird. Operationen innerhalb eines Aggregates können so gestaltet werden, dass sie die Geschäftsregeln und Integritätsbedingungen immer einhalten.
- Vereinfachung der Komplexität: Indem verwandte Objekte in einem Aggregate gruppiert werden, kann die Komplexität des Modells reduziert werden. Das erleichtert das Verständnis der Geschäftslogik und der Datenbeziehungen.
- Transaktionsmanagement: Aggregates bieten einen natürlichen Rahmen für das Transaktionsmanagement. Änderungen innerhalb eines Aggregates können als eine einzige Transaktion betrachtet werden, was die Integrität der Geschäftsoperationen gewährleistet.
- Optimierung von Datenzugriffsmustern: Aggregates können so gestaltet werden, dass sie häufige Datenzugriffsmuster unterstützen, was die Leistung verbessern kann.

Implementierung

Angenommen, in unserem Spotify-Client-Projekt, könnte ein "Album" als ein Aggregate betrachtet werden, das mehrere "Tracks" umfasst. Das "Album" wäre hier das Aggregate Root, und die "Tracks" wären die verbundenen Entities innerhalb des Aggregates.

```
csharp
Copy code
public class Album : Entity
{
    public string Name { get; private set; }
    private List<Track> tracks = new List<Track>();
    public IReadOnlyList<Track> Tracks => tracks.AsReadOnly();

    public Album(string name)
    {
        Name = name;
    }

    public void AddTrack(Track track)
```

```
{  
    // Geschäftsregeln und Konsistenzprüfungen vor dem Hinzufügen eines Tracks  
    tracks.Add(track);  
}  
  
// Weitere Methoden und Geschäftslogik...
```

In diesem Beispiel ist `Album` das Aggregate Root, das verantwortlich für die Verwaltung der Konsistenz und der Geschäftsregeln bezüglich seiner `Tracks` ist. Die Methode `AddTrack` könnte beispielsweise sicherstellen, dass keine Duplikate hinzugefügt werden oder dass die Anzahl der Tracks innerhalb zulässiger Grenzen bleibt. Durch die Nutzung dieses Aggregates können wir sicherstellen, dass die Änderungen am Album und seinen Tracks immer in einem konsistenten Zustand bleiben.

Aggregates in DDD bieten eine klare Strukturierung und Kapselung der Geschäftslogik und tragen wesentlich zur Integrität und zur Vereinfachung des Domain-Modells bei.

Entities

Entities sind Objekte in einem Domain-Driven Design (DDD), die über eine eindeutige Identität verfügen und deren Attribute und Beziehungen im Laufe der Zeit ändern können, ohne dass sich ihre Identität ändert. Im Gegensatz zu Value Objects, die durch ihre Attribute definiert werden, werden Entities durch eine eindeutige Identität (oft eine ID oder ein Schlüssel) definiert, die es ermöglicht, sie eindeutig zu identifizieren, auch wenn ihre anderen Eigenschaften variieren.

Beschreibung und Begründung des Einsatzes einer Entity

Der Einsatz von Entities erlaubt es, die kontinuierliche Identität von Objekten innerhalb der Geschäftsdomäne zu modellieren und zu verwalten, selbst wenn sich ihre Zustände ändern. Dies ist besonders wichtig für Objekte, die über einen längeren Zeitraum hinweg von Bedeutung sind und in verschiedenen Kontexten verwendet werden, wo Konsistenz und Kontinuität der Identität erforderlich sind. Beispiele hierfür könnten Kunden, Bestellungen oder eben Musik-Tracks in einer Musik-Streaming-Anwendung sein.

Entities spielen eine zentrale Rolle in der Modellierung von Geschäftsprozessen, da sie es ermöglichen, komplexe Lebenszyklen und Beziehungen innerhalb der Domäne zu verwalten. Die klare Identität von Entities erlaubt es, sie in verschiedenen Teilen eines Systems eindeutig zu referenzieren und zu manipulieren, was für die Integrität und Nachverfolgbarkeit von Geschäftsoperationen entscheidend ist.

Implementierung

Im Kontext unseres Spotify-Client-Projekts wäre ein gutes Beispiel für eine Entity der `Track`, der eine eindeutige ID besitzt und dessen Attribute (wie Name, Künstler oder Dauer) sich im Laufe der Zeit ändern können, ohne dass sich seine Identität ändert.

```
public class Track
{
    public string Id { get; set; }
    public string Name { get; set; }
    public string DurationMin { get; set; }
    public bool Explicit { get; set; }
    public string PreviewUrl { get; set; }
    public List<Artist> Artists { get; set; } = new List<Artist>();
    public Album Album { get; set; }
    public int Popularity { get; set; }
    public int TrackNumber { get; set; }
    public string Type { get; set; }

}
```

In diesem Beispiel wird der `Track` durch eine eindeutige `Id` identifiziert, was ihn zu einer Entity macht. Auch wenn der Name des Tracks, die Dauer, der Künstler oder ob es sich um expliziten Inhalt handelt, sich im Laufe der Zeit ändern kann, bleibt die Identität des Tracks durch seine `Id` erhalten.

Die Verwendung von Entities wie `Track` ermöglicht es dem System, spezifische Tracks über verschiedene Operationen und Transaktionen hinweg zu identifizieren und zu manipulieren,

unabhängig von Änderungen an ihren nicht-identifizierenden Attributen. Dies ist essenziell für die Entwicklung von robusten, konsistenten Geschäftsanwendungen und Systemen.

Value Objects

In der Architektur für eine Spotify-Client-Anwendung unter Verwendung von Domain-Driven Design (DDD)-Prinzipien spielen Wertobjekte (Value Objects) eine entscheidende Rolle bei der Kapselung der Eigenschaften und Verhaltensweisen von Objekten, die nur durch ihre Attribute und nicht durch eine eindeutige Identifikation definiert sind. Die explizite Verwendung von Wertobjekten im bereitgestellten Code ist jedoch nicht direkt hervorgehoben, aber wir können ihre potenzielle Präsenz und Bedeutung im Domain-Modell, insbesondere innerhalb des `Entities`-Namensraums, ableiten und rechtfertigen.

Beschreibung und Begründung des Einsatzes von Wertobjekten

Potenzielle Wertobjekte:

- Track, Artist, Album, Playlist: Diese Entitäten enthalten Attribute, die ihre Domain-Konzepte beschreiben. Zum Beispiel hat `Track` Attribute wie `Name`, `DurationMin` und `Explicit`, die gemeinsam seinen Zustand definieren. Obwohl diese Entitäten primär als Entitäten behandelt werden (weil sie eindeutige Identifikatoren wie `Id` haben), zeigen sie auch Eigenschaften von Wertobjekten, wenn man ihre unveränderlichen Eigenschaften betrachtet, die ihren Domain-Wert definieren.
- Genre: Obwohl nicht explizit als Klasse definiert, können Genre-Informationen innerhalb der `Artist`-Klasse (`List<string> Genre`) als Wertobjekte betrachtet werden. Sie kapseln die musikalischen Genres ein, mit denen ein Künstler assoziiert ist, definiert rein durch den Wert der Genres selbst und ohne eindeutigen Identifikator.

Begründung für Wertobjekte:

- Unveränderlichkeit: Wertobjekte sind unveränderlich. Einmal erstellt, kann ihr Zustand nicht ändern. Diese Unveränderlichkeit ist entscheidend, um sicherzustellen, dass das Domain-Modell konsistent bleibt. Zum Beispiel ändert

sich die Dauer oder der Name eines `Track` nicht, sobald er erstellt wurde, was das Verhalten von Wertobjekten widerspiegelt.

- Gleichheit: Wertobjekte sind gleich, wenn ihre Attribute gleich sind, nicht notwendigerweise ihre Speicherreferenzen. Dieses Konzept ist wesentlich für Domain-Modelle, bei denen die Bedeutung des Wertes wichtiger ist als die Identität der Instanz. Zum Beispiel sollten zwei `Genre`-Instanzen mit demselben Set von Genre-Strings als gleich betrachtet werden, unabhängig davon, ob sie dieselbe Instanz sind.
- Nebenwirkungsfreies Verhalten: Wertobjekte können Verhalten tragen, das für ihren Domain relevant ist, ohne Nebenwirkungen auf den Anwendungsstatus zu verursachen. Diese Eigenschaft stimmt mit den Prinzipien reiner Funktionen in der funktionalen Programmierung überein und verbessert die Vorhersehbarkeit und Wartbarkeit.

Refactoring

Code Smells

[CODE SMELL 1]

[CODE SMELL 2]

2 Refactorings

Entwurfsmuster

Entwurfsmuster: [Name]

Entwurfsmuster: [Name]

