

# Programmmentwurf

## BookTracker

Name: Neumann, Leon  
Matrikelnummer: 8226045

Abgabedatum: 28.05.2023

## Kapitel 1: Einführung

### Übersicht über die Applikation

Die Applikation wird verwendet, um eine persönliche Digitale Bibliothek zu verwalten. Die zwei Hauptbestandteile sind:

1. Die Leseliste. Hier werden Titel hinzugefügt und vermerkt die der Benutzer in der Zukunft lesen möchte, unabhängig davon, ob er die Titel bereits besitzt oder sich diese wünscht. Des Weiteren kann die Leseliste benutzt werden, um sich eine Wunschliste zu generieren.
2. Das Lesearchive. Hier wird gespeichert welche Bücher von der Leseliste wann gelesen wurden. Das Archive wird vor allem als Datenquelle für eine Analyse für das Leseverhaltens zu machen.

Des Weiteren können, aus den erfassten Daten, Statistiken generiert werden. Die Applikation beschäftigt sich vor allem aber mit der Verwaltung der Bücher. So ist es zum Beispiel möglich auch Titel per Web-Suche hinzuzufügen (Realisiert über die Google Books API). Die Applikation ist

ausschließlich eine Kommandozeilen Applikation. Zum Persistieren der Daten wurde ein eigenes einfaches, CSV ähnliches Format verwendet.

## **Wie startet man die Applikation?**

Installation einer Java IDE (Während der Entwicklung wurde IntelliJ verwendet). Projekt von git zu klonen. Navigation zu der App Klasse BookTracker\src\main\java\app.java und ausführen der dort zu findenden main Methode. Dann startet die Applikation und man wird aufgefordert eine Bibliotheksdatei zu laden bzw. eine neue anzulegen. Dateipfad angeben (Test Bibliothek bereits verfügbar unter BookTracker\text.txt). Danach kann man die geladene Bibliothek verwalten. Für eine genauere Beschreibung der Befehle „help“ eingeben.

## **Wie testet man die Applikation?**

Um die Applikation zu testen, kann die Testfunktionalität, der IDE genutzt werden. Die Tests befinden sich im unter BookTracker\src\test\java. JUnit5 wird benötigt für die Tests.

## **Kapitel 2: Clean Architecture**

### **Was ist Clean Architecture?**

Clean Architecture ist ein Softwareentwurfskonzept, das von Robert C. Martin entwickelt wurde. Es beschreibt eine Methode zur Strukturierung von Software-Systemen, die darauf abzielt, die Abhängigkeiten zwischen verschiedenen Komponenten zu reduzieren und die Wartbarkeit, Erweiterbarkeit und Testbarkeit der Software zu verbessern.

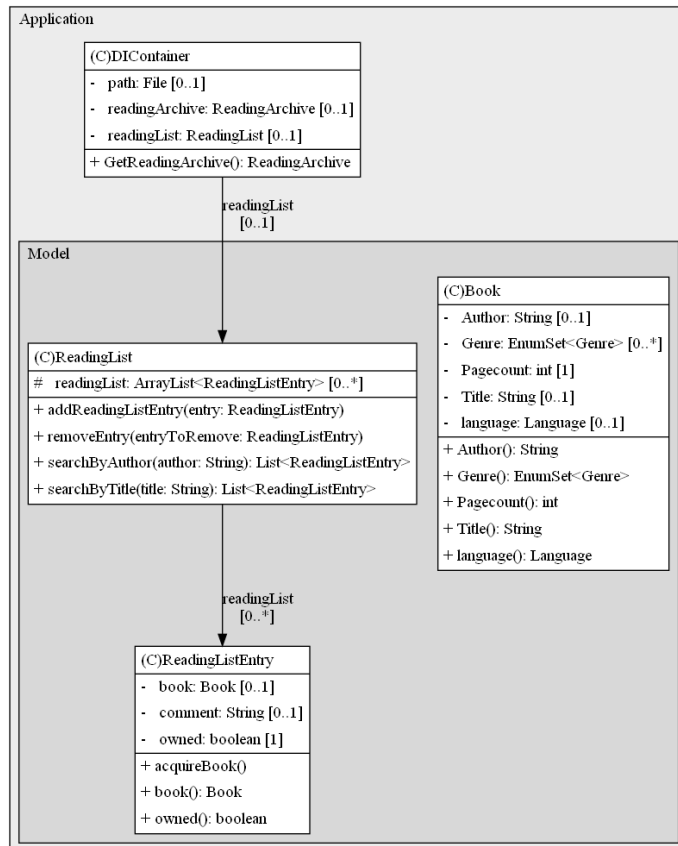
Die Clean Architecture teilt ein Software-System in verschiedene Schichten ein, wobei jede Schicht eine bestimmte Verantwortung hat und nur von den Schichten darunter abhängig ist. Die äußerste Schicht ist die Plugin Schicht, die unter anderem die Interaktion mit dem Benutzer ermöglicht. Des Weiteren ist diese Schicht die Schnittstelle für Datenbanken oder andere Drittsysteme. Darunter liegt die Adapterschicht. Dort liegen Adapter und Controller, diese Schicht wird aber oft erst sinnvoll bei komplexeren und größeren Systemen und kann bei kleineren Anwendungen wegfallen. Die nächste Schicht ist die Anwendungsschicht die Business-Logik enthält. Sie ist für die direkte Funktionalität der Anwendung verantwortlich. Die wichtigste Schicht und die Domänenschicht. Sie beinhaltet das eigentliche Model das direkt aus der Problem Domäne kommt. Diese Schicht sollte die echte Welt so gut wie möglich abbilden. Die unterste Schicht ist die Abstraktionsschicht. In ihr werden, falls zutreffend, mathematische Konzepte oder ähnliches abgebildet. Insgesamt gibt es also zwischen drei und fünf Schichten je nach Anwendung, Größe und Komplexität des Systems.

Die Schichten sind so organisiert, dass sie nur nach innen hin abhängig sind, d.h. eine höhere Schicht kann die darunter liegenden Schichten verwenden, aber umgekehrt darf eine niedrigere Schicht nicht auf eine höhere Schicht zugreifen. Dies wird als "Dependency Rule" bezeichnet, die sicherstellt, dass Änderungen in einer Schicht keine Auswirkungen auf andere Schichten haben.

## Analyse der Dependency Rule

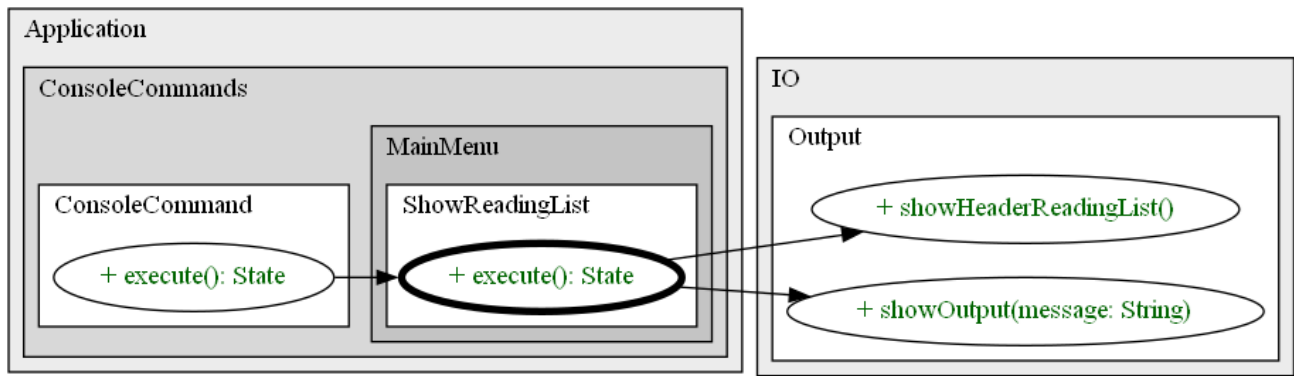
### **Positiv-Beispiel: Dependency Rule**

Die Klasse ReadingList ist in der Domänenschicht. Sie repräsentiert die Leseliste, also Bücher, welche der Benutzer noch lesen möchte. Die Klasse hat keine Abhängigkeiten außer zur Klasse ReadingListEntry und Book, welche auch in der Domänenschicht liegen.



### **Negativ-Beispiel: Dependency Rule**

Die Klasse ShowReadingList liegt in der Anwendungsschicht. Sie ist dafür zuständig den Inhalt der Leseliste zu formatieren und anzuzeigen. Sie hat aber eine direkte Abhängigkeit zu der Klasse Output, welche in der Plugin Schicht liegt und für die Ausgabe zuständig ist. Damit ist sie zu einer höher liegenden Schicht direkt abhängig. Dieses Problem kommt in der Anwendung sehr häufig vor und eine direkte Verletzung der Dependency Rule. Im folgenden Call Diagramm ist zu sehen wie die Klasse direkt von der Statischen Klasse Output abhängt weil es die Methoden aufruft.

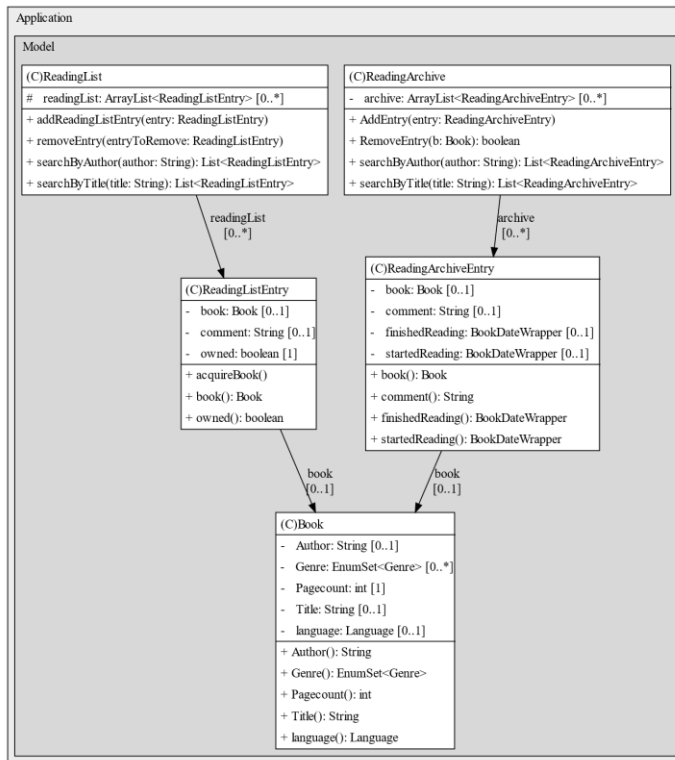


## Analyse der Schichten

### **Schicht: Plugin Schicht**

Die GoogleBooksWebApi Klasse ist dafür verantwortlich im Web nach Büchern zu suchen, die dann in der Anwendung benutzt werden. Die Suche funktioniert mit einer http Anfrage an die Google Books Web API. Die Klasse implementiert das BookFinder interface, welches in der Anwendungsschicht ist. Die Aufgabe von einem BookFinder ist im Grunde sehr simple. In der Anwendung soll ein Buch gesucht werden, der BookFinder nimmt einen Suchstring entgegen und gibt mögliche Ergebnisse zurück. Wie die Suche stattfindet (DB, API, etc.) ist für die Anwendung egal. Die Klasse erweitert die Kernfunktionalitäten der Anwendung und liegt deswegen in der Plugin Schicht.



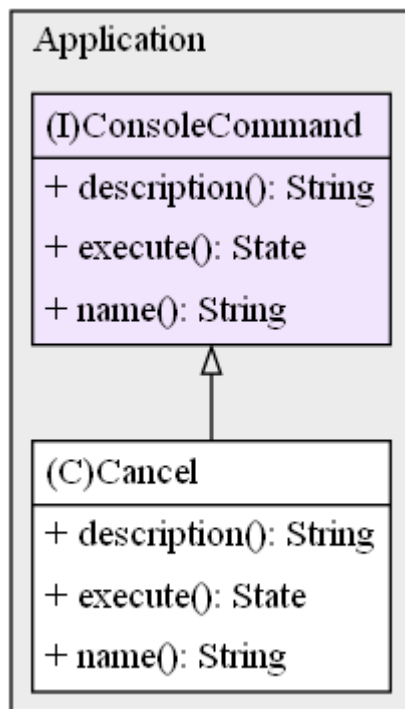


## Kapitel 3: SOLID

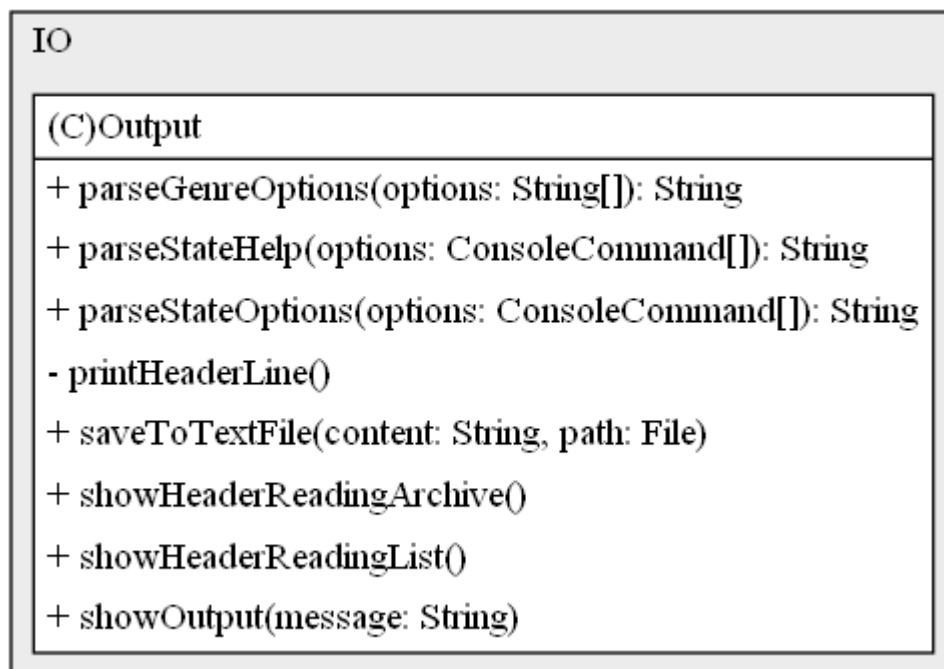
### Analyse Single-Responsibility-Principle (SRP)

#### *Positiv-Beispiel*

Die Cancele Klasse erlaubt es dem User den momentanen Anwendungszustand zu verlassen, damit kehrt die Anwendung immer zurück in das Hauptmenü (Bzw. den dazugehörigen Zustand). Die Klasse ist absolut minimal gehalten und implementiert nur das benötigte Interface und die einzige Aktion, die die Klasse in der Methode „Execute“ ausführt, ist die Rückgabe des Hauptmenü Zustandes.

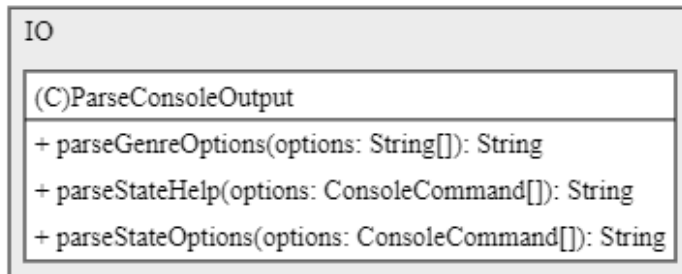
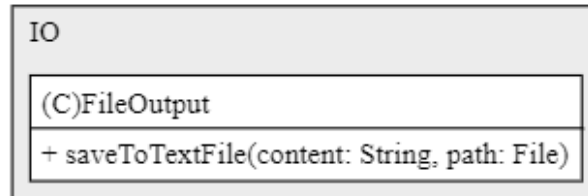
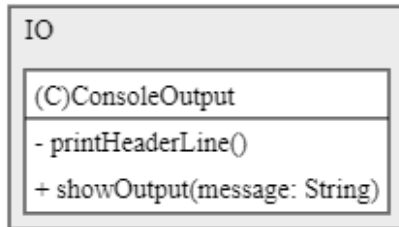


### ***Negativ-Beispiel***



Die Output Klasse soll die Ausgabe von der Anwendung zu dem User in jeglicher Form darstellen. Sie ist jedoch sehr lange und der Usecase ist nicht sehr klar definiert. So übernimmt die Klasse nicht nur strikt die direkte Ausgabe von Text zur Konsole, wie es der Name vermuten lässt, sondern sie

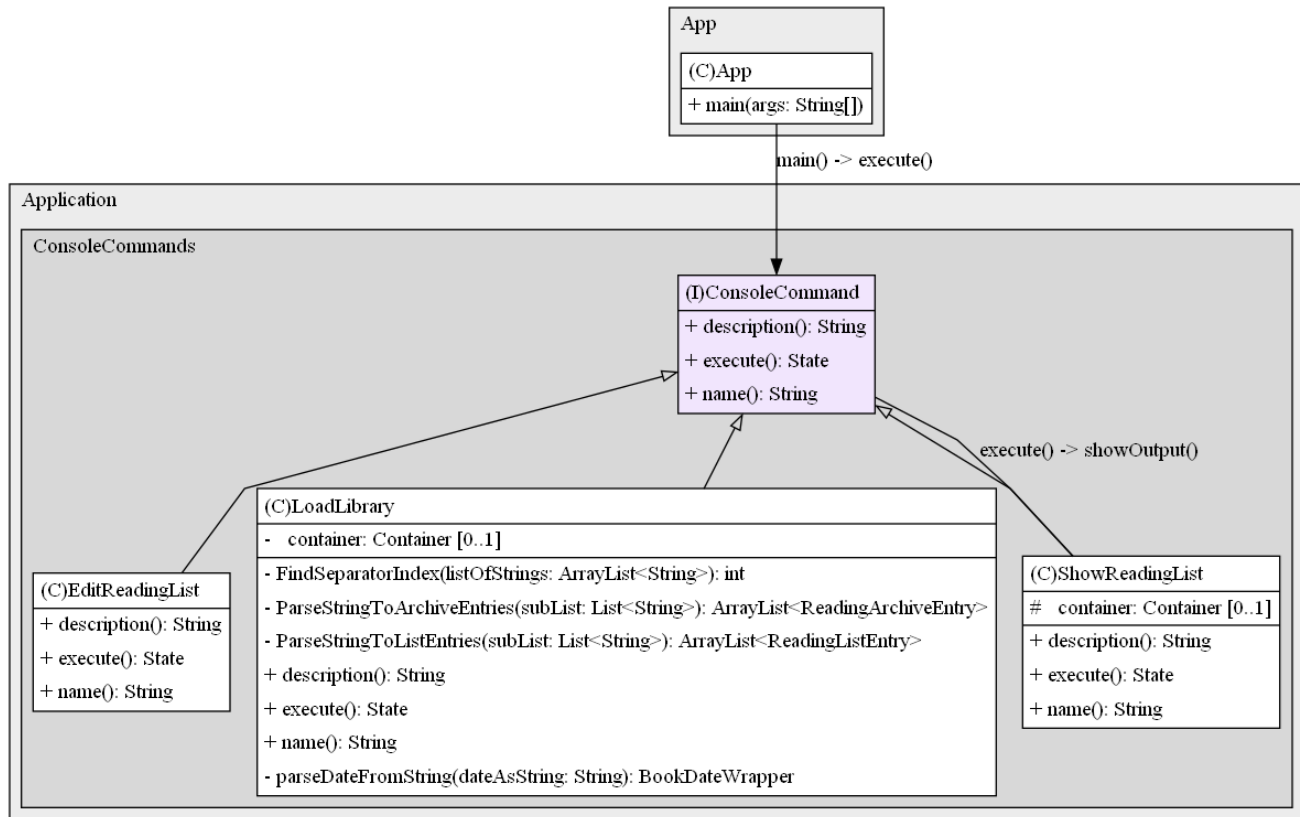
übernimmt auch das Formatieren/parsen von Daten und die Ausgabe von Dateien. Gerade die Ausgabe von Dateien. Eine mögliche Lösung wäre die Klasse aufzuteilen in z.B. ConsoleOutput, FileOutput und ParseOutput. Damit wären die verschiedenen Methoden nach ihren Funktionen aufgeteilt und die einzelnen Klassen würden nur noch eine einziges „Thema“ behandeln.





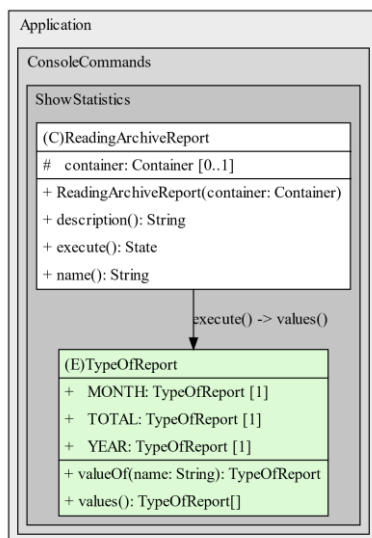
## Analyse Open-Closed-Principle (OCP)

### Positiv-Beispiel



Das Positive Beispiel für das Open-Closed-Principle ist das ConsoleCommand Interface. Alle ausführbaren Befehle implementieren dieses Interface. Die App kennt nur das Interface und führt die Befehle damit aus. (Aus Gründen der Lesbarkeit wurde einige Klassen, die das Interface implementieren weggelassen). Die App Klasse hat mehrere Listen mit ConsoleCommands und führt dann, abhängig von einer Usereingabe und dem Zustand der App, das entsprechende ConsoleCommand aus. Diese Klasse ist ein gutes Beispiel für das OCP weil einfach neue Klassen das Interface implementieren können und dadurch von der App ausgeführt werden können (Open for Extension), ohne die App verändern zu müssen (Closed for modification).

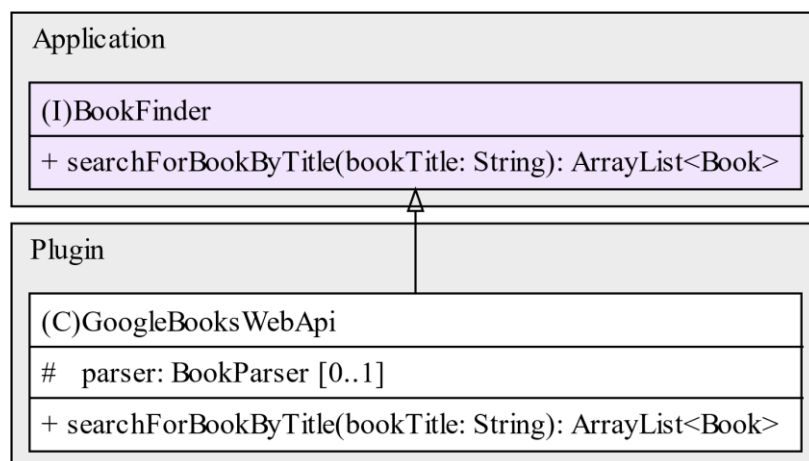
## Negativ-Beispiel



Die Klasse ReadingArchiveReport hängt direkt von dem Enum ab, im Konstruktor wird mit einem switch Case entschieden wie das Objekt initialisiert wird. Wenn das Enum geändert wird, dann muss auch die Klasse angepasst werden. Das widerspricht dem Prinzip, dass die Klasse geschlossen für Veränderungen ist. Eine Lösung wäre anstatt eines Enums wäre mit einem Interface zu arbeiten oder einer Klasse die Überladenen Methoden bereitstellt.

## Analyse Liskov-Substitution- (LSP), Interface-Segregation- (ISP), Dependency-Inversion-Principle (DIP)

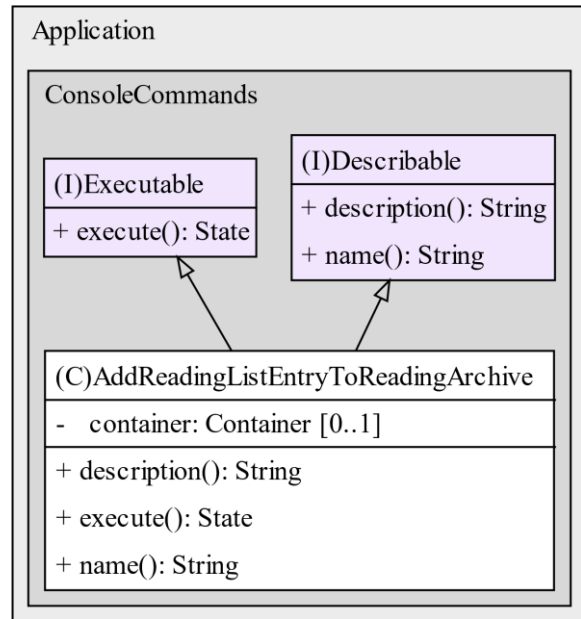
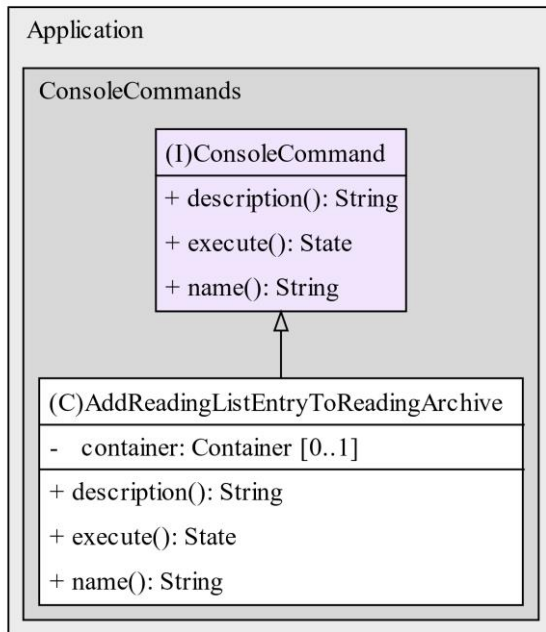
### Positiv-Beispiel



Da dieses Interface nur eine Methode implementiert ist es absolut minimal und erfüllt auch nur die im Namen angesprochene Funktion. Es gibt keine Möglichkeit das Interface weiter aufzuteilen.

## Negativ-Beispiel

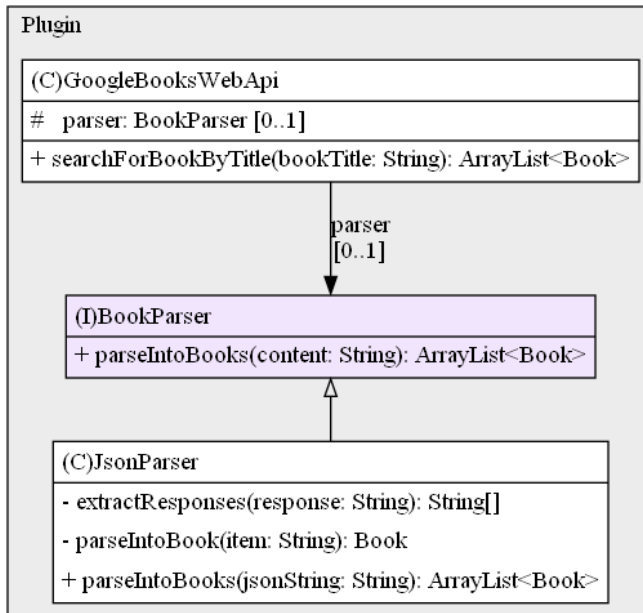
In diesem Beispiel erfüllt das Interface ConsoleCommand mehrerer Funktionen. Der Name beschreibt auch nicht direkt die Funktionalität. Das Interface ist aus praktischen Gründen so entstanden. Nach dem ISP wäre es jedoch sinnvoller das Interface aufzuteilen. Ein Teil des Interfaces macht das Objekt beschreibbar, der andere Teil stellt sicher das das Objekt ausgeführt werden kann. Es wäre also sinnvoll das Interface nach den Funktionalitäten zu unterteilen.



## Kapitel 4: Weitere Prinzipien

### Analyse GRASP: Geringe Kopplung

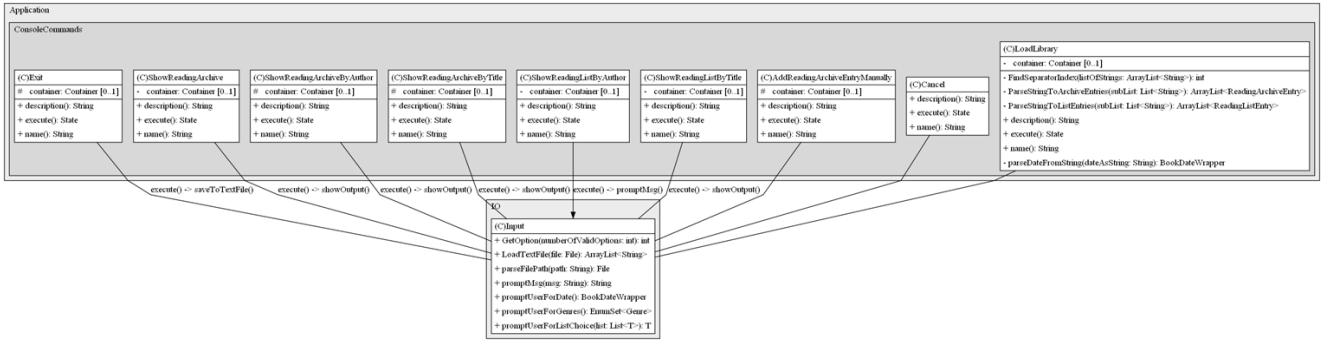
#### *Positiv-Beispiel*



In der Klasse `GoogleBooksWebApi` werden Https requests an die `GoogleBooksApi` gesendet. Die Antworten sind Json Strings. Diese Strings werden mithilfe eines `BookParsers` zu Büchern geparkt, die dann in der Anwendung verwendet werden. Die Implementierung des `JsonParsers` ist momentan eine String Manipulation, die nicht sehr performant ist, aber funktioniert. In der Zukunft soll diese selbstgeschriebene Implementierung mit einer Json Parser Bibliothek ersetzt werden. Durch die geringe Kopplung, die durch das Interface erreicht wurde, ist dies leicht möglich.

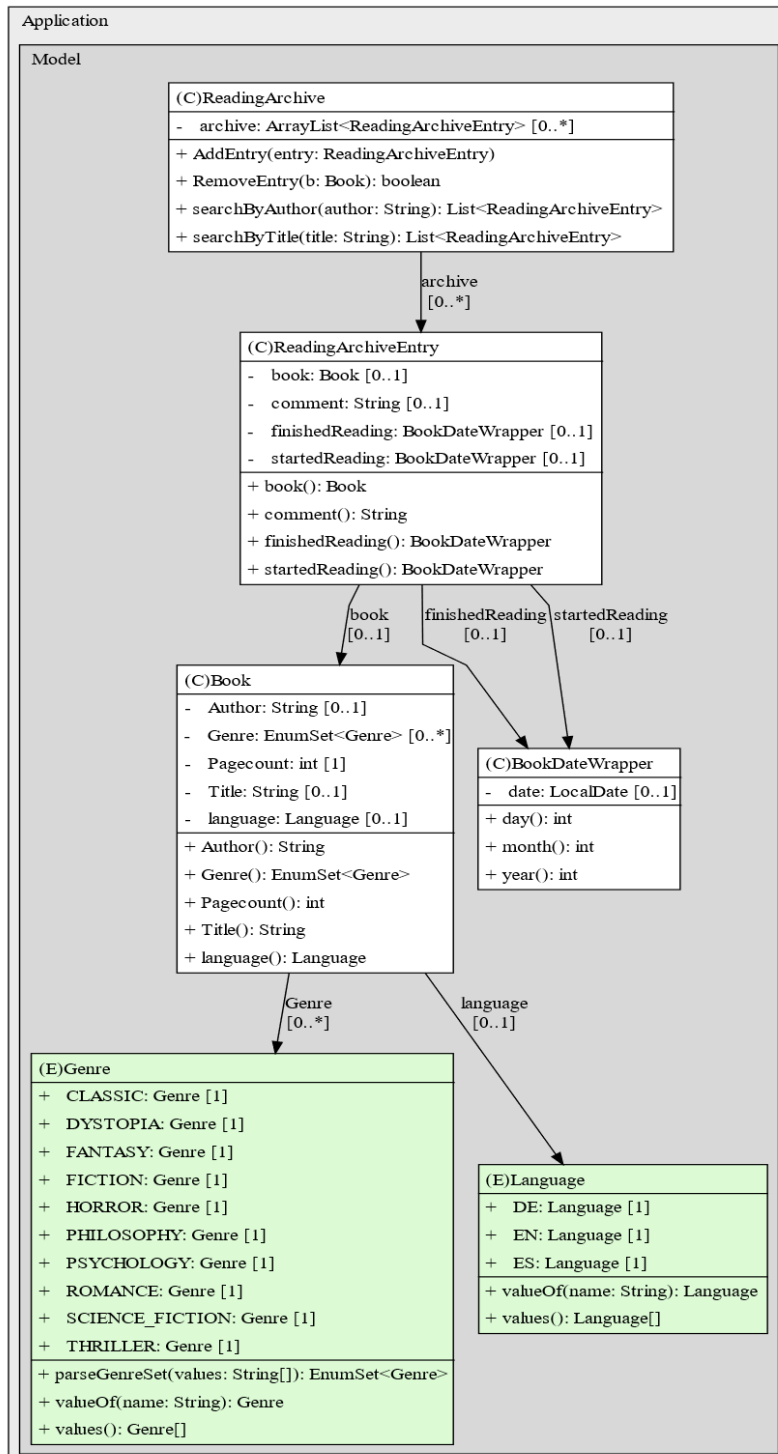
#### ***Negativ-Beispiel***

Die Input Klasse verletzt die Geringe Kopplung Regel sehr stark. Die Klasse ist statisch und fast jede `ConsoleCommand` Implementierung hängt direkt von dieser Statischen Klasse ab. Sollte diese jemals ausgewechselt werden, dann muss jede andere Klasse angepasst werden. Besser wäre es ein Input Interface festzulegen und diesen Input im Konstruktor der Nutzerklasse zu übergeben.



## Analyse GRASP: Hohe Kohäsion

Die Klasse `ReadingArchive` weist eine hohe Kohäsion auf. Sie ist ein Teil der Bibliothek, nämlich der Teil, in dem die bereits gelesenen Bücher sich befinden. In der Domäne würde man die Bücher in dem Archive erwarten, was hier auch der Fall ist. Zwischen die Archive und dem Buch liegt noch eine Art Wrapper Klasse `ReadingArchiveEntry`, welche das Speichern von weiteren Informationen zu den Büchern ermöglicht.



## Don't Repeat Yourself (DRY)

Commit: „Dry Commit“ <https://github.com/Leonneu/BookTracker/commit/e3844f4e9c529db6ca0a9240b925a9fa3bc5c559>

In diesem Commit wurde duplizierter Code in eine Methode ausgelagert. Dies führt zu einer besseren Wartbarkeit und zu einer besseren Lesbarkeit (Methodenname erklärt was passiert).

Vorher:

```
BookDateWrapper dateStart;
if (!values[5].trim().equals("-")) {
    var dateStr = values[5].trim().split("\\.");
    dateStart = new BookDateWrapper(Integer.parseInt(dateStr[0]), Integer.parseInt(dateStr[1]), Integer.parseInt(dateStr[2]));
} else {
    dateStart = null;
}
BookDateWrapper dateEnd;
if (!values[6].trim().equals("-")) {
    var dateStr = values[6].trim().split("\\.");
    dateEnd = new BookDateWrapper(Integer.parseInt(dateStr[0]), Integer.parseInt(dateStr[1]), Integer.parseInt(dateStr[2]));
} else {
    dateEnd = null;
}
```

Nachher:

```
BookDateWrapper dateStart = parseDateFromString(values[5]);
BookDateWrapper dateEnd = parseDateFromString(values[6]);
```

neue Methode:

```
private BookDateWrapper parseDateFromString(String dateAsString) {
    BookDateWrapper dateStart;
    if (!dateAsString.trim().equals("-")) {
        var dateStr = dateAsString.trim().split("\\.");
        dateStart = new BookDateWrapper(Integer.parseInt(dateStr[0]), Integer.parseInt(dateStr[1]), Integer.parseInt(dateStr[2]));
    } else {
        dateStart = null;
    }
    return dateStart;
}
```

## Kapitel 5: Unit Tests

### 10 Unit Tests

Unit Test	Beschreibung
-----------	--------------

TestAddBookByWebSearch#ensureBookGetsAdded	Testet ob Bücher richtig zur ReadingList hinzugefügt werden.
TestOutputBuilder#testOutputBuilderGetsUsedProperly	Testet ob der OutputBuilder richtig genutzt wird von den ConsoleCommands.
TestInput#testGenreSelection	Testet ob die Usereingabe bei der Genre Auswahl richtig geparkt wird.
TestJsonParser#testStringBookParsing	Testet ob der JsonParser richtig den String zu Book Objekten parset.
TestBookDateWrapper#testDurationCalculation	Testet ob die Länge zwischen zwei Daten richtig berechnet wird.
TestReadingArchiveGenerator#testFastestRead	Testet ob der richtige Eintrag gefunden wird.
TestReadingArchiveGenerator#testShortestRead	Testet ob der richtige Eintrag gefunden wird.
TestReadingListGenerator#testReadingListOwned	Testet ob der Anteil an Bücher welche in Besitz sind richtig berechnet wird.
TestReadingListGenerator#testTotal	Testet ob die Anzahl an der Seiten insgesamt richtig berechnet wird.
TestReadingListGenerator#testAvg	Testet ob die Durchschnittliche Seitenanzahl richtig berechnet wird.

## ATRIP: Automatic

Die Automatisierung wurde realisiert, indem die Tests mit minimalem Aufwand aufgerufen werden können. Durch das verwendete Testing Framework JUnit 5 können alle Tests über einen Knopfdruck ausgeführt werden. Die Ergebnisse werden direkt in der IDE angezeigt.

## ATRIP: Thorough

Positiv Beispiel:

```
public static List<Arguments> genreTest() {
    return List.of(
        Arguments.of("0", EnumSet.of(Genre.CLASSIC), null),
        Arguments.of("0123", EnumSet.of(Genre.CLASSIC,
            Genre.THRILLER,
            Genre.ROMANCE,
            Genre.FANTASY), null),
        Arguments.of("asd", null, NoSuchElementException.class),
        Arguments.of("11", EnumSet.of(Genre.THRILLER), null),
        Arguments.of("-1", null, NoSuchElementException.class)
    );
}

@ParameterizedTest
@MethodSource("genreTest")
public void testGenreSelection(String input, EnumSet<Genre> expected, Class<? extends Throwable>
```



```

expectedException) {
    Input.userInput = new Scanner(input + Output.lineBreak);
    if(expectedException != null){
        Assertions.assertThrowsExactly(expectedException, Input::promptUserForGenres);
        return;
    }
    Assertions.assertEquals(expected, Input.promptUserForGenres());
}

```

Hier wird getestet ob die Benutzereingabe richtig geparkt wird. Um alle Möglichkeiten abzudecken, wird ein “ParameterizedTest” verwendet. Die TestMethode wird mit zufälligen erwarteten Werten, wie “0”, aufgerufen, sowie mit Edgecases wie “asd” die keine zulässige Eingabe sind, hier wird erwartet das ein Fehler auftritt.

### Negativ Beispiel:

```

@BeforeEach
public void setup(){
    testArchive = new ReadingArchive(new ArrayList<>());
    for (int i = 0; i < 100; i++) {
        testArchive.AddEntry(generateEntry());
    }
}
@Test
public void testFastestRead() {
    ReadingArchiveEntry expected = new ReadingArchiveEntry(new Book("title","author",100000,Language.EN),
        new BookDateWrapper(10,10,2023),
        new BookDateWrapper(11,10,2023),
        "");
    testArchive.AddEntry(expected);
    ReadingArchiveStatisticGenerator generator = new ReadingArchiveStatisticGenerator(testArchive);
    Assertions.assertEquals(expected, generator.fastestRead());
}

```

Dieser Test überprüft, ob die Statistik für das schnellst gelesene Buch richtig errechnet wird. Da der Test jedoch nur einen einzigen Fall abdeckt ist es nicht thorough. Es wäre sinnvoll an dieser Stelle mit einem ParameterizedTest zu arbeiten, um sicher zu gehen, dass immer das richtige Buch gefunden wird. Vor Allem auch weil mit einem BeforeEach setup gearbeitet wurde, sollte es ein leichtes sein diesen Test gründlicher durchzuführen.

## **ATRIP: Professional**

### Positiv Beispiel:

```

public static List<Arguments> testReadingListOwned() {
    return List.of(
        Arguments.of(new ReadingList(generateArrayList(1, 2)), 0.33, 0.01),
        Arguments.of(new ReadingList(generateArrayList(10, 0)), 1, 0),
        Arguments.of(new ReadingList(generateArrayList(4, 6)), 0.4, 0),
        Arguments.of(new ReadingList(generateArrayList(500, 500)), 0.5, 0),
        Arguments.of(new ReadingList(generateArrayList(0, 0)), Double.NaN, 0),
        Arguments.of(new ReadingList(generateArrayList(0, 20)), 0, 0));
}

@ParameterizedTest
@MethodSource("testReadingListOwned")
public void testPercentOfBooksOwned(ReadingList readingList, double expected, double delta) {
    ReadingListStatisticGenerator generator = new ReadingListStatisticGenerator(readingList);
}

```

```
Assertions.assertEquals(expected, generator.percentOfBooksOwned(), delta);
}
```


Hier wird getestet, ob die prozentuale Anzahl an Bücher im besitzt, richtig errechnet wird. Dieser Test ist Professional, weil er sich nur auf das wesentliche konzentriert. Dadurch ist er auch leicht lesbar und wartbar. Des Weiteren wurden extra Methoden wie “generateArrayList” geschrieben, um Code Duplikate zu vermeiden. Auch wurde mit einem Delta für lange Nachkommastellen gearbeitet (wie bei z.b.  $1/3 \Rightarrow 33.333\dots\%$ ). Dieser Test hat mindestens die Qualität des Produktivcodes, wenn nicht sogar eine höhere.

### Negativ Beispiel:

```
@Test
public void testStringBookParsing() {
    String outputWebSearch = ""
    {
        "kind": "books#volumes",
        "totalItems": 148,
        "items": [
            {...}
        ]
    }"";
    ArrayList<Book> re = new JsonParser().parseIntoBooks(outputWebSearch);
    Book b = r.get(0);
    assertEquals("Die Worte des Lichts",b.title());
    assertEquals("Brandon Sanderson",b.author());
    assertEquals(976,b.pagecount());
    assertEquals(Language.DE,b.language());
}
```

Diese Klasse testet, ob der JSON Parser den gegebenen String richtig zu einem Book Objekt parst. Dieser Test entspricht nicht den Clean Code Standards. Die Methode ist unübersichtlich und viel zu lange durch den (hier für die Lesbarkeit verkürzten) String, der alleine schon über 200 Zeilen lang ist. Außerdem wird das Ergebnis der Websuche nicht aktualisiert, auch wenn sich die WEB-API ändern sollte. Des Weiteren wird mit Magic Ints wie der Anzahl der Buchseiten gearbeitet. Die Variablen Namen sind auch nicht sprechen (siehe “r” und “b”). Die vier unterschiedlichen asserts könnten auch zusammengefasst werden, würde man direkt mit dem erwartendem Buch vergleichen und nicht jedes Attribut einfach. Aufgrund der vier asserts ist außerdem Debugging schwerer, weil nicht direkt ersichtlich ist warum der Test fehlschlägt.

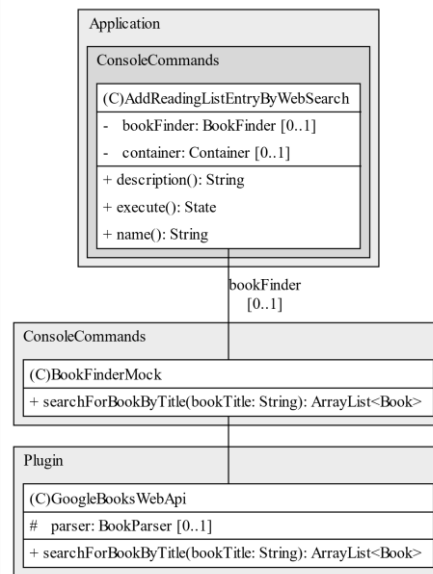
## Code Coverage

Element ^	Class, %	Method, %	Line, %
>  all	48% (24/50)	34% (78/225)	31% (218/703)

Die Code Coverage ist mit 31% Line Coverage gering. Das Problem ist, das die Anwendung größtenteils aus Benutzerinteraktion besteht. Der wichtigste Bestandteil sind die Console Commands, welche oft Benutzerinteraktion durch den Input/Output Klasse benutzen. Dadurch das diese statischen Klassen sind und diese die Dependency Rule verletzen, ist es relativ schwierig diese effizient zu mocken, für das Testen der Console Commands. Außerdem wurde bei der Entwicklung kein Test driven Development Ansatz gewählt, was dazu geführt hat, dass weniger Tests geschrieben wurden.

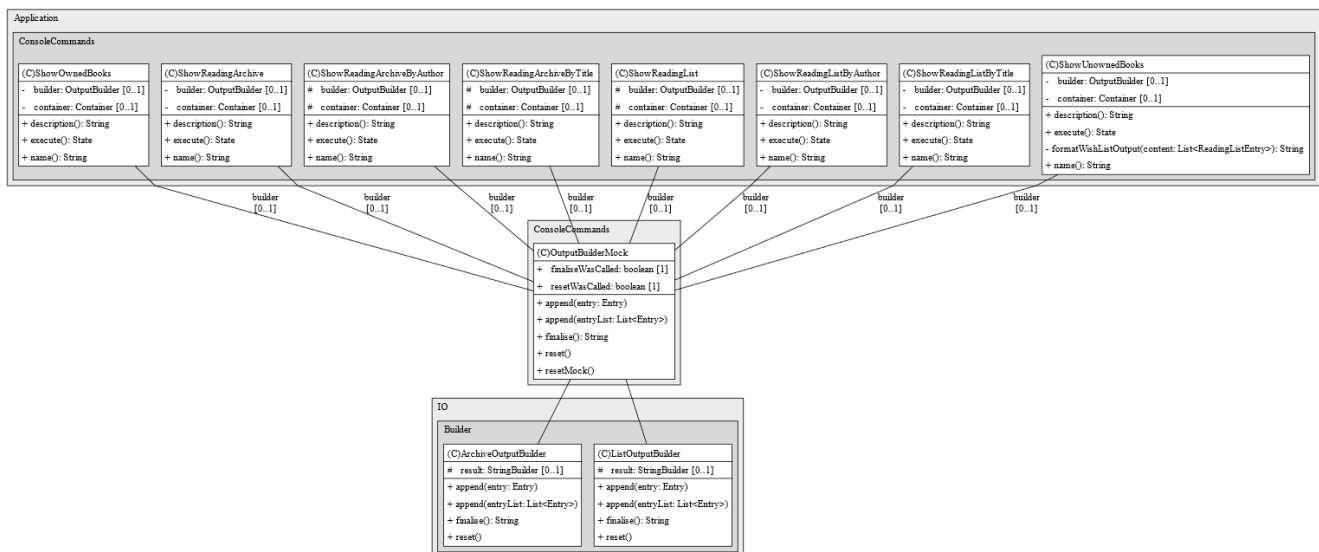
## Fakes und Mocks

### BookFinderMock:



Beim Testen des AddBookByWebsearch Klasse wird eine Methode getestet, die normalerweise eine http Anfrage an eine WEB-API macht. Damit die Tests isoliert und sicher durchgeführt werden können wird an dieser Stelle ein Mock verwendet, um nicht bei jedem Testdurchlauf eine https Request zu machen.

### OutputBuilderMock:



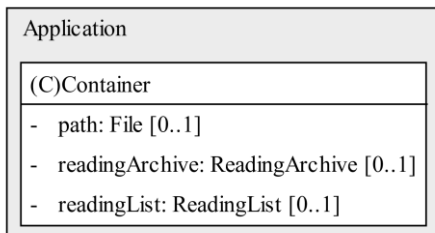
Um sicherzustellen, dass der Outputbuilder richtig genutzt wird von allen ConsoleCommands, wurde ein Outputbuilder Mock erstellt, welcher testet, ob der Outputbuilder richtig initialisiert und finalisiert wird. Das Mock zählt die relevanten Aufrufe und anschließend wird sichergestellt, dass die Anzahl der Aufrufe der erwartenden Anzahl entspricht.

## Kapitel 6: Domain Driven Design

### Ubiquitous Language

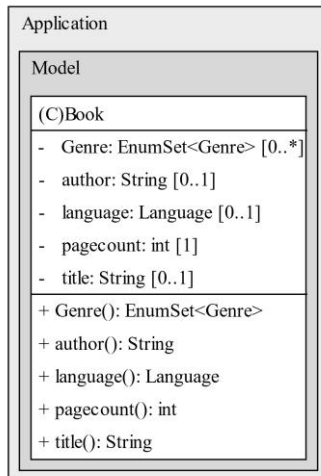
Bezeichnung	Bedeutung	Begründung
Book	Ein Objekt, welches Informationen persistiert (z.B. Titel) und ein Buch über das Objekt, welches Teil der echten Welt repräsentiert.	Book ist Teil der UL weil es zentraler Bestandteil der Anwendung ist, und Entwicklern ermöglicht welcher auch Teil der Problemdomäne der echten Welt ist, leicht zu verstehen.
StateDictionary	Das Lexikon, welches definiert, welche Befehle im momentanen Zustand ausführbar sind.	Teil der UL weil es Entwicklern die Möglichkeit gibt leicht zu verstehen was für Aktionen in dem Momentanen Zustand der Anwendung möglich sind. Hier können Informationen „nachgeschlagen“ werden.
Genre	Ein Enum welches alle möglichen Genre beinhaltet welches ein Buch haben kann.	Ermöglicht es den Entwicklern sich leicht über die möglichen Kategorien von Büchern zu unterhalten.
Container	Ein zentrales Objekt welches den Zugriff auf gespeicherte Objekte ermöglicht.	Der Container entspricht in seinem Namen auch seiner Funktion, und ermöglicht den Entwicklern sich ein sinnvolles Mentales Model zu machen und so die Funktionsweise der Klasse leicht zuzuordnen.

### Entities



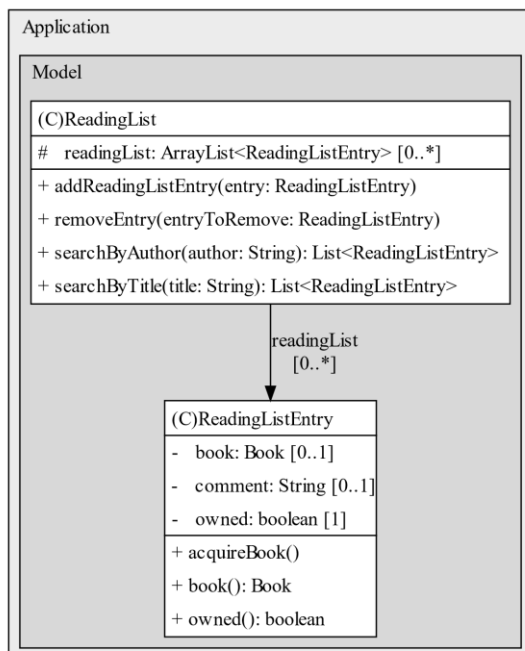
Die Klasse Container ist einmalig und eindeutig identifizierbar. Sie enthält das aktuelle ReadingArchive und die aktuelle ReadingList. Die Aufgabe der Klasse ist es den ConsoleCommands übergeben zu werden und dann eine Zentralen zugriffpunkt zu ermöglichen. Die Klasse selbst hat praktisch kein Verhalten definiert, ist aber einzigartig und wird an viele Stellen verwendet, um auf andere Datenstrukturen zuzugreifen und diese zu speichern, was ein Merkmal von Entities ist.

## Value Objects



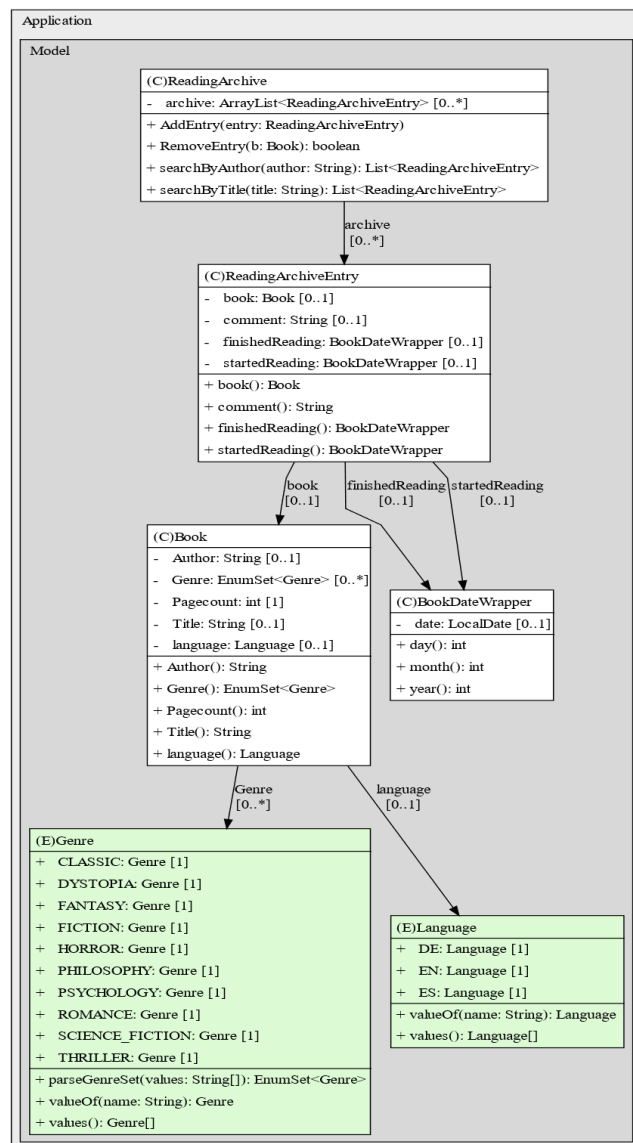
Die Book Klasse wurde als Java Record implementiert. Aus diesem Grund ist auch nicht extra die hashCode und equals Methode aufgeführt. Es wurde sich entschieden das Buch als reinen Record zu implementieren, weil die Felder eigentlich keinen Grund haben sich jemals zu ändern, die Seitenanzahl z.B. ändert sich nie von einem Buch.

## Repositories



Die ReadingList ist ein Repository. Sie verwaltet alle ReadingListEntries. Die Readinglist bietet eine zentrale Möglichkeit, um auf alle ReadingList zuzugreifen und diese auch zu speichern. Sie ermöglicht es außerdem dann vereinheitlicht alle ReadingEntries auf einmal als CSV zu persistieren, weil in ihr alle Entries vereinheitlicht abgespeichert sind.

# Aggregates



Die Klasse `ReadingArchiveEntry` ist ein Aggregat. Sie verwaltet das Buch, die dazugehörigen Datum. Wenn eine Ressource aus dem Entry gebraucht wird, dann wird diese über dieses Aggregat angefordert.

## Kapitel 7: Refactoring

### Code Smells

#### Duplicate Code:

Folgender Duplicate Code wurde gefunden:

```
public static String parseStateOptions(ConsoleCommand[] options) {
    int n = options.length;
    StringBuilder result = new StringBuilder();
    for (int i = 0; i < n; i++) {
```

```

        result.append(i).append(". ").append(options[i].name()).append(
lineBreak);
    }
    return result.toString();
}

public static String parseStateHelp(ConsoleCommand[] options) {
    int n = options.length;
    StringBuilder result = new StringBuilder();
    for (int i = 0; i < n; i++) {
        result.append(i).append(". ").append(options[i].description()).append(
lineBreak);
    }
    return result.toString();
}

```

Die beiden Methoden machen exakt das gleiche, außer das die eine Methode den Namen ausgibt und die andere die Beschreibung. Wenn beim Aufruf der Methode einfach direkt das gewünschte Attribut übergeben wird, dann kann die Methode wiederverwendet werden. Das Übergeben der gewünschten Attribute lässt sich auch sehr leicht mit java streams.map() realisieren. Nach dem Entfernen des Code Smells bleibt nur noch eine Methode übrig:

```

public static String parseOptions(List<String> options){
    int n = options.size();
    StringBuilder result = new StringBuilder();
    for (int i = 0; i < n; i++) {
        result.append(i).append(". ").append(options.get(i)).append(
lineBreak);
    }
    return result.toString();
}

```

Dadurch wurde die Wartbarkeit des Codes erhöht. (Commit <https://github.com/Leonneu/BookTracker/commit/0148390838de84a7eb7619bb06f2ccc468e03f8e>)

### Long Method:

Folgende lange Methode wurde gefunden:

```

public State execute() {
    ReadingList readingList = container.getReadingList();
    var result = readingList.getUnownedBooks();
    OutputBuilder builder = new ListOutputBuilder();
    for (var e:result
    ) {
        builder.append(e);
    }
    Output.showOutput(builder.finalise());
    String ans = Input.promptMsg("Als Wunschliste speichern?
(Y/N)").toLowerCase();
    if(ans.startsWith("j") || ans.startsWith("y")){
        try {

```

```

        String path = Input.promptMsg("Pfad?");
        File f = new File(path+"\\Wunschliste.txt");
        if(f.createNewFile()){
            Output.saveToTextFile(formatWishListOutput(result), f);
        }
    } catch (Exception e) {
        Output.showOutput("Fehler beim Speichern: "+e.getMessage());
    }
    Output.showOutput("Datei erfolgreich gespeichert!");
}

return State.SHOWREADINGLIST;
}

```

Die Methode ShowUnownedBooks realisiert die Funktionalität dem Benutzer die Möglichkeit zu geben das Ergebnis des Befehls als Datei zu speichern. Der Speichervorgang an sich hat aber eigentlich keine Berechtigung in dieser Klasse zu sein. Eigentlich gehört sie in die I/O Klasse. Dieser Codesmell hat aufgezeigt, wie die Klasse das Single-Responsibility-Principle verletzt. Die angesprochene Funktionalität wurde in die entsprechende Klasse Output ausgelagert.

```

public State execute() {
    ReadingList readingList = container.getReadingList();
    var result = readingList.getUnownedBooks();
    OutputBuilder builder = new ListOutputBuilder();
    for (var e : result) {
        builder.append(e);
    }
    String output = builder.finalise();
    Output.showOutput(output);
    if (Input.promptUserIfSave()) {
        String path = Input.promptMsg("Pfad?");
        Output.saveToNewTextFile(path, output);
    }
    return State.SHOWREADINGLIST;
}

```

Commit:

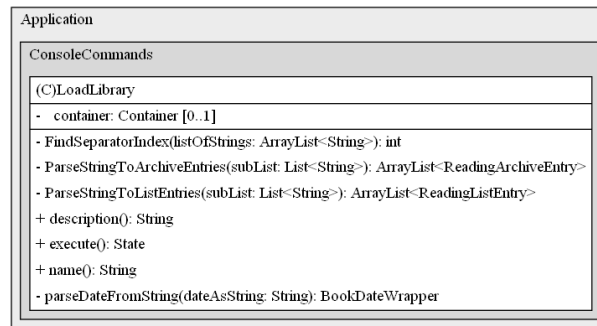
<https://github.com/Leonneu/BookTracker/commit/a2ef2d9c085f0b99cc52c2b99fbc5be3c95f00c4>

## Refactorings

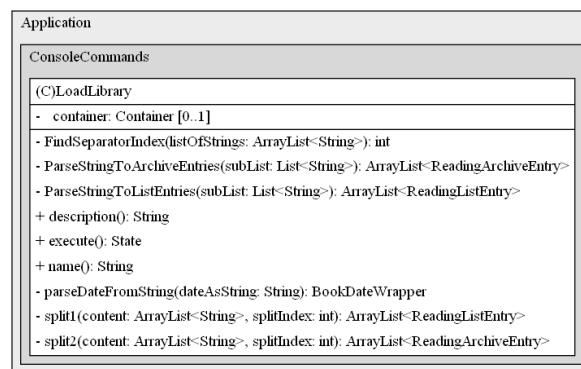
### Extract Method:

In der Klasse LoadLibrary wurden zwei Codeblöcke extrahiert, um die Lesbarkeit und die Länge der execute Methode zu verbessern.





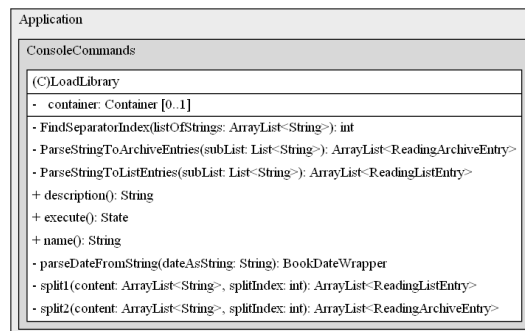
Nach der Änderung gibt es zwei Methoden mehr. Diese Änderung verbessert die Readability und Maintainability der Klasse. Der Commit in dem es passiert ist folgender:  
<https://github.com/Leonneu/BookTracker/commit/2bdafe46358bec693c94a03db492c956b32c5e4a>



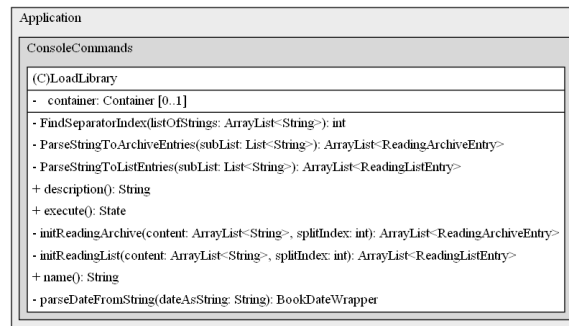
## Rename Method:

Beim Extrahieren der Methode hatte der Autor eine kreative Blockade, weshalb es zu einem unpassenden Methodennamen kam. Die Methoden split1 und split2 sind nicht sehr aussagekräftig und erklären nicht deren Bedeutung, des Weiteren erschweren sie die Maintainability des Codes. Nach intensiverer Auseinandersetzung mit dem Problem wurde ein besserer Methodenname gefunden und der Code refactored.

Davor:



Danach:

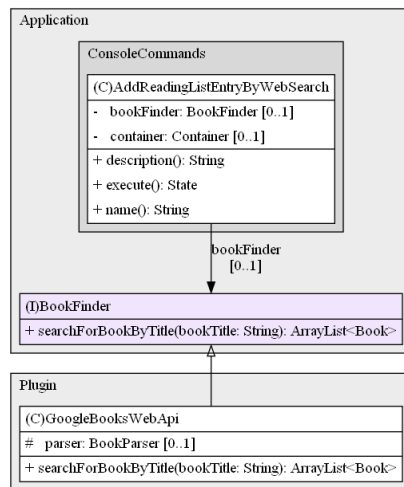


Der Methodenname `initReadingArchive` beschreibt was in der Methode passiert und verbessert auch die Readability der `execute` Methode. (Commit:

<https://github.com/Leonneu/BookTracker/commit/c075b359c02cce8a5dae7d4920744bfb780f5c46>)

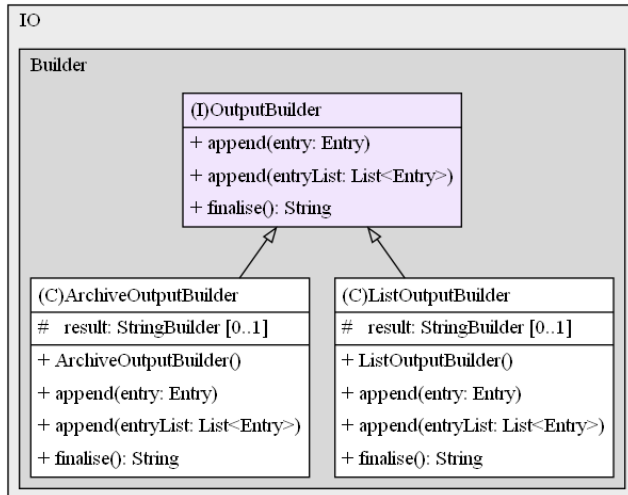
## Kapitel 8: Entwurfsmuster

### Entwurfsmuster: Strategie



Bei der Implementierung wurde sich für das Strategie Entwurfsmuster entschieden. Um Bücher zur Anwendung hinzuzufügen, gibt es momentan nur die Möglichkeit eine Websuche zu machen. Diese gibt dann passende Bücher zurück. In der Zukunft ist es aber auch möglich andere Datenquellen zu benutzen, wie zum Beispiel eine Datenbank oder auch CSV Dateien oder ähnliches. Wo gesucht wird kann der Nutzer dann entscheiden, bzw. es wird in allen anderen Quellen auch gesucht. Die Voraussetzung ist lediglich das es ein Klasse gibt die das `BookFinder` Interface implementiert. Damit wird sichergestellt das man flexibel bei der Datenquelle für die Suche bleibt.

## Entwurfsmuster: Builder



Um den Bücherliste schön formatiert auszugeben wurde ein Builderpattern verwendet. Abhängig davon was für ein Entry vorliegt muss die Implementierung des Outputbuilders gewählt werden. Die Implementierung von dem OutputBuilder verwendet eine StringBuilder um die Bücher formatiert als Tabelle mit Header und Trennzeichen auszugeben. Der Builder bekommt Bücher oder Bücherlisten übergeben. Es können immer weitere angehängt werden und am Ende wird `finalise()` aufgerufen welches die Ausgabe fertig formatiert. Im folgenden Schaubild ist erkenntlich, wie eine Klasse den OutputBuilder verwendet, um eine Ausgabe zu erstellen, diese wird dann direkt mit `Output.show()` angezeigt.

