Programmentwurf
Kalenderanwendung

Name: Wolff, Nico
Matrikelnummer: 3601588

Name: Fischer, Adrian
Matrikelnummer: 4181324

Abgabedatum: 31.5.2022

Allgemeine Anmerkungen:

- *es darf nicht auf andere Kapitel als alleiniger Leistungsnachweis verwiesen werden (z.B. in der Form "XY wurde schon in Kapitel 2 behandelt, daher hier keine Ausführung")*
- *alles muss in UTF-8 codiert sein (Text und Code)*
- *sollten mündliche Aussagen den schriftlichen Aufgaben widersprechen, gelten die schriftlichen Aufgaben (ggf. an Anpassung der schriftlichen Aufgaben erinnern!)*
- *alles muss ins Repository (Code, Ausarbeitung und alles was damit zusammenhängt)*
- *die Beispiele sollten, wenn möglich vom aktuellen Stand genommen werden*
  - *finden sich dort keine entsprechenden Beispiele, dürfen auch ältere Commits unter Verweis auf den Commit verwendet werden*
  - *Ausnahme: beim Kapitel "Refactoring" darf von vorneherein aus allen Ständen frei gewählt werden (mit Verweis auf den entsprechenden Commit)*
- *falls verlangte Negativ-Beispiele nicht vorhanden sind, müssen entsprechend mehr Positiv-Beispiele gebracht werden*
  - *Achtung: werden im Code entsprechende Negativ-Beispiele gefunden, gibt es keine Punkte für die zusätzlichen Positiv-Beispiele*
  - *Beispiele*
    - *"Nennen Sie jeweils eine Klasse, die das SRP einhält bzw. verletzt." (2P)*
      - *Antwort: Es gibt keine Klasse, die SRP verletzt, daher hier 2 Klassen, die SRP einhalten: [Klasse 1], [Klasse 2]*
      - *Bewertung: falls im Code tatsächlich keine Klasse das SRP verletzt: 2P ODER falls im Code mind. eine Klasse SRP verletzt: 1P*
- *verlangte Positiv-Beispiele müssen gebracht werden*
- *Code-Beispiel = Code in das Dokument kopieren*
- *Gesamt-Punktzahl: 60P*
  - *zum Bestehen (mit 4,0) werden 30P benötigt*

Kapitel 1: Einführung (4P)

Übersicht über die Applikation (1P)

[Was macht die Applikation? Wie funktioniert sie? Welches Problem löst sie/welchen Zweck hat sie?]

Unsere Anwendung bietet einem Nutzer die Möglichkeit, je nach Rolle, sich zu registrieren, sich einzuloggen, eine Kalenderübersicht zu sehen, Termine im Kalender anzulegen, Termine zu editieren, Termine zu löschen und Termine anzulegen.

Sie funktioniert über eine .NET-Konsolenanwendung, welche über Konsolenoutput einen Kalender bzw. einen Vorgang zur Nutzung der Anwendung anzeigt.

Ein Beispielszenario wäre ein Autohaus, welches Kunden und Mitarbeiter des Autohauses zu verwalten hat. Kunden können hierbei Termine anlegen und ihre eigenen Termine einsehen. Mitarbeiter können daraufhin diese Termine einsehen, editieren oder löschen.

Wie startet man die Applikation? (1P)

[Wie startet man die Applikation? Was für Voraussetzungen werden benötigt? Schritt-für-Schritt-Anleitung]

Die Anwendung läuft über das NET 5.0 SDK, das heißt, um die Anwendung starten zu können benötigt man dieses SDK auf dem gewünschten Gerät.

Ebenso wird unter Linux empfohlen, die Anwendung über VSCode zu starten. Dafür befinden sich die nötigen .vscode Dateien bereits im GitHub-Repository oder können in der Readme.md des Projekts nachgelesen werden.

Unter Windows wird empfohlen Visual Studio zu verwenden.Bei beiden IDEs reicht es aus, die Anwendung zu bauen und auszuführen.

Technischer Überblick (2P)

[Nennung + Erläuterung der Technologien, jeweils Begründung für Einsatz der Technologien]

Für unsere Anwendung wurde das .NET Framework als Hauptframework der Anwendung verwendet. Begründung dafür sind die C#-Kenntnisse der Entwickler und die Möglichkeit der Erstellung einer Konsolenanwendung im Framework.

Ebenso wurde das Newtonsoft.Json Package verwendet, da es eine einfache Möglichkeit bietet, Objekte in .NET zu serialisieren bzw. zu deserialisieren.

Die Nutzung des Package beschränkt sich auf die „CustomJsonConverter“-Klasse, sodass diese bei Bedarf einfach mit einer ähnlichen Lösung ausgetauscht werden kann.

Es wurde das ”FakeItEasy” Framework für Mock/Fake-Objekte benutzt.

Kapitel 2: Clean Architecture (8P)

Was ist Clean Architecture? (1P)

[allgemeine Beschreibung der Clean Architecture in eigenen Worten]

Clean Architecture besagt, dass die Codearchitektur in einzelne Schichten aufgeteilt wird. So können einzelnen Schichten ausgetauscht werden, ohne dass die komplette Anwendungsarchitektur in sich zusammenfällt. Die Schichten sind verschachtelt und die inneren Schichten wissen nichts von den äußeren Schichten.

Ebenso wird die Clean Architecture unter vielen Namen, kaum bis wenig verändert, genutzt, unteranderem Domain-Driven-Design oder Onion Architecture.

Die wichtigsten Schichten einer Clean Architecture beinhalten:

- Eine Schicht für das UI
- Eine Datenbankschicht/Datenverarbeitungsschicht
- Eine Domänenschicht
- Eine Anwendungsschicht

Analyse der Dependency Rule (2P)

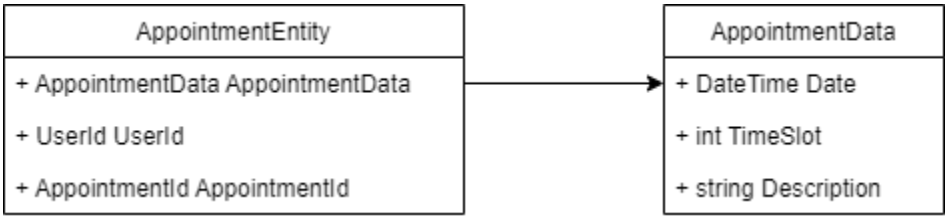
[1 Klasse, die die Dependency Rule einhält und 1 Klasse, die die Dependency Rule verletzt; jeweils UML der Klasse und Analyse der Abhängigkeiten in beide Richtungen (d.h., von wem hängt die Klasse ab und wer hängt von der Klasse ab) in Bezug auf die Dependency Rule]

Positiv-Beispiel: Dependency Rule

Klasse:

AppointmentData

UML:



Analyse der Abhängigkeiten in beide Richtungen:

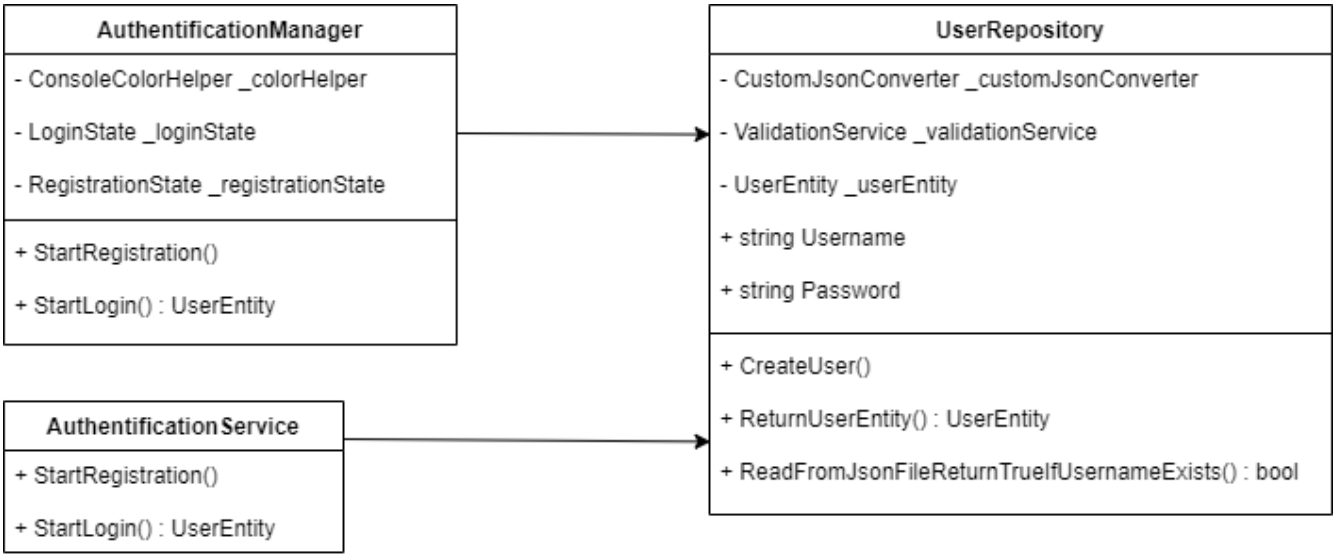
Die AppointmentData-Klasse ist von keiner anderen Klasse abhängig, die AppointmentEntity-Klasse ist jedoch von der AppointmentData-, UserId-, sowie der AppointmentId-Klasse abhängig.

Positiv-Beispiel 2: Dependency Rule

Klasse:

UserRepository

UML:



Analyse der Abhängigkeiten in beide Richtungen:

Die UserRepository-Klasse ist von keinen Klassen außerhalb der Application-Schicht abhängig. Keine ihrer Methoden greift auf die Präsentationsschicht zu, lediglich auf die Domänenschicht der Anwendung.

Die UserRepository-Klasse wird jedoch von Klassen der Präsentationsschicht genutzt.

Analyse der Schichten (5P)

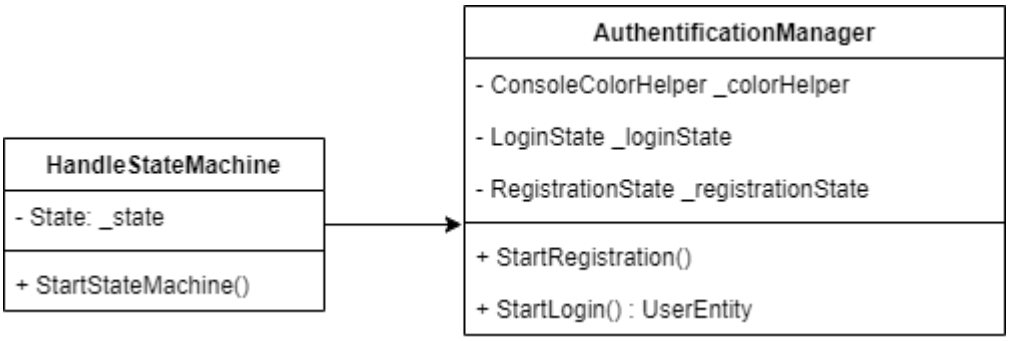
[jeweils 1 Klasse zu 2 unterschiedlichen Schichten der Clean-Architecture: jeweils UML der Klasse (ggf. auch zusammenspielenden Klassen), Beschreibung der Aufgabe, Einordnung mit Begründung in die Clean-Architecture]

Schicht: Präsentationsschicht (ASE_Calendar.ConsoleUI)

Klasse:

AuthentificationManager

UML:



Aufgabe der Klasse:

Die AuthentificationManager-Klasse übernimmt die Abläufe der Registrierung, sowie des Logins in der Konsole.

Die Methode StartRegistration() führt den Nutzer durch eine Reihe von Abfragen wie Nutzername, Passwort und Rolle und ruft dann einen Service auf, welcher die Registration verwaltet.

Die Methode StartLogin() führt den Nutzer ebenfalls durch Abfragen nach Nutzername und Passwort und ruft dann einen Service auf, welcher den Login verwaltet.

Einordnung mit Begründung:

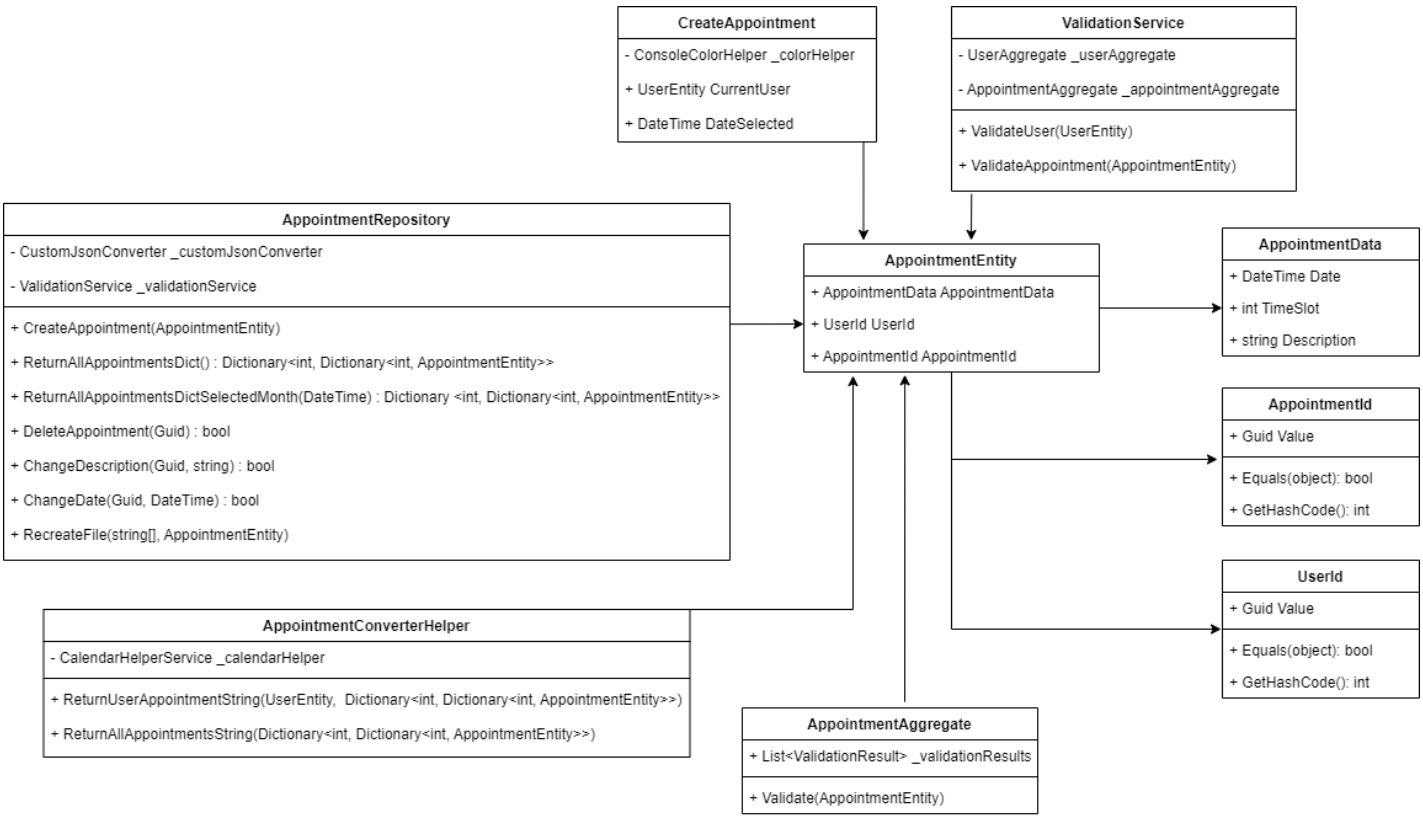
Die Präsentationsschicht übernimmt die komplette Präsentation der Anwendung zu einem Endgerät. In unserem Fall findet die Präsentation in der Konsole statt, möchte man nun jedoch auf eine webbasierte Lösung oder etwas anderes umsteigen, so muss man nur die Präsentationsschicht entsprechend anpassen.

Schicht: Domänenschicht (ASE_Calendar.Domain)

Klasse:

AppointmentEntity

UML:



Aufgabe der Klasse:

Die Appointmententität ist eine der meistgenutzten Klassen der Anwendung.

Diese nutzt die AppointmentId-Klasse, die UserId-Klasse, sowie die AppointmentData-Klasse, um einen Termin im Kalender darzustellen.

Weiter wird die Appointmententität von der Präsentationsschicht zur Erstellung von Terminen oder zur Verarbeitung von Entitäten zu Textfolgen zur Darstellung verwendet.

In einem Appointmentrepository wird die Appointmententität für die CRUD-Operationen benötigt.

Einordnung mit Begründung:

Die Domänenschicht übernimmt im Allgemeinen die Beschreibung von Entitäten, den Domainmodellen, und der Verknüpfung dieser mit der Geschäftslogik und Regeln der Anwendung mithilfe von Aggregaten.

In unserem Fall übernimmt die Domänenschicht die Beschreibung der Termine, sowie der Nutzer der Anwendung, als Entitäten.

Diese Entitäten nutzen Value Objects und ergeben so Objekte mit eindeutiger Id.

Die Geschäftslogik sowie die Regeln der Entitäten werden durch die Aggregate dargestellt, hier wird beispielsweise festgelegt, dass ein Termin keine leere Beschreibung besitzen darf oder ob die Entität einen gültigen Zeitraum besitzt.

Kapitel 3: SOLID (8P)

Analyse SRP (3P)

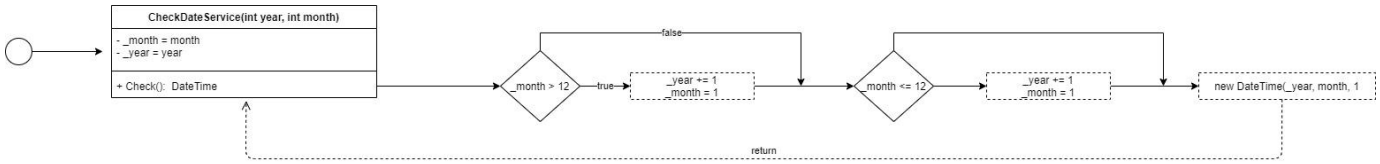
[jeweils eine Klasse als positives und negatives Beispiel für SRP; jeweils UML der Klasse und Beschreibung der Aufgabe bzw. der Aufgaben und möglicher Lösungsweg des Negativ-Beispiels (inkl. UML)]

Positiv-Beispiel

Klasse:

CheckDateService

UML:



Beschreibung der Aufgabe:

Die Klasse erhält ein Jahr und einen Monat.

Wird nun die Methode Check() aufgerufen wird überprüft, ob der Monat kleiner als 12 ist und größer als 0. Falls beides eintrifft, wird an den Werten nichts verändert und diese Werte als neues DateTime-Objekt zurückgegeben.
Falls der Monat größer als 12 ist, wird das gegebene Jahr um eins erhöht und der Monat auf eins gesetzt.

Falls der Monat größer kleiner gleich 0 ist, wird das Jahr um 1 verringert und der Monat auf 12 gesetzt.

Der neue Monat oder das neue Jahr wird ebenfalls als neues DateTime Objekt zurückgegeben.

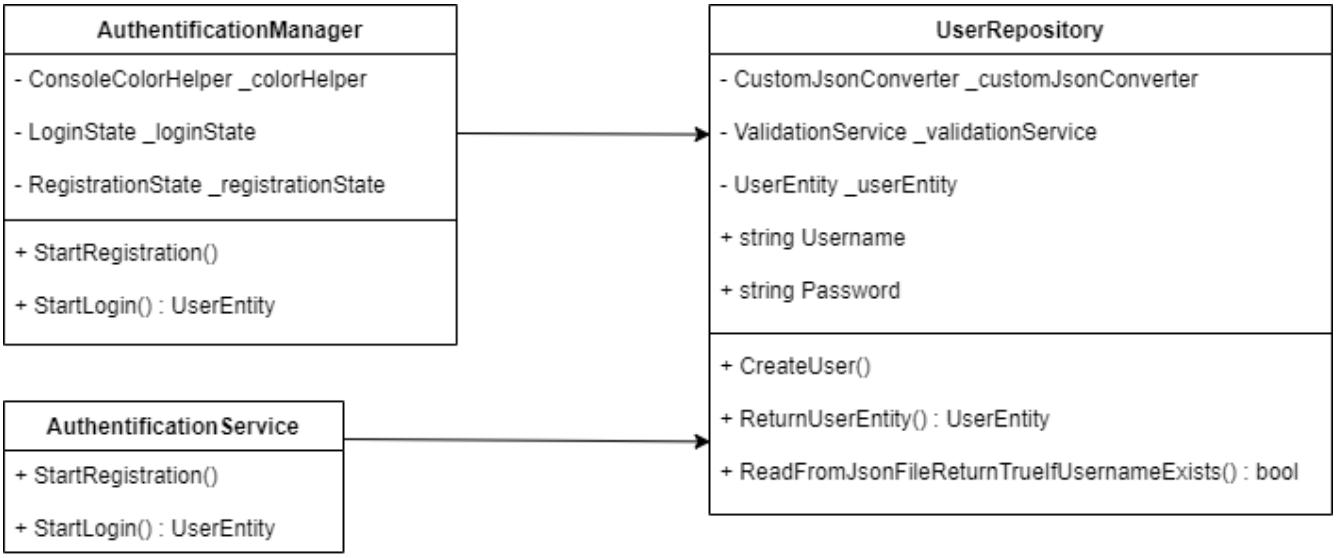
Diese Klasse kümmert sich nur um die Überprüfung, ob die gegebene Monatszahl valide ist. Falls nicht wird das Jahr entsprechend angepasst.

Negativ-Beispiel

Klasse:

UserRepository

UML:



Beschreibung der Aufgabe:

Die Klasse kann über die Methode CreateUser() einen Nutzer erstellen und dem Repository hinzufügen.

Über die Klasse ReturnUserEntity() wird ein Nutzer mit gegebenen Zugangsdaten zurückgeben.

Über die Methode ReadFromJsonReturnTrueIfUsernameExists() wird ein Bool zurückgeben, wenn ein gegebener Nutzername bereits vorhanden oder nicht vorhanden ist.

Diese Klasse kümmert sich einmal um das Erstellen eines Users (Create) und um die Ausgabe der Daten (Read).

Um die Klasse nach dem SRP auszurichten, sollte man sie in zwei Klassen aufteilen.

Eine Klasse, die sich nur um das Erstellen eines Users kümmert und eine Klasse, die nur die Aufgabe hat, das Repository auszulesen.

Analyse OCP (3P)

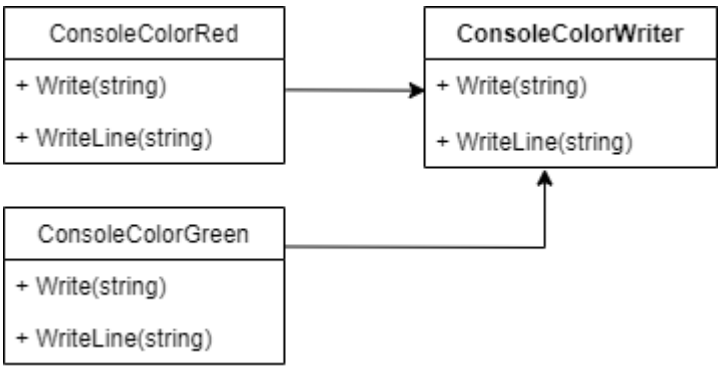
[jeweils eine Klasse als positives und negatives Beispiel für OCP; jeweils UML der Klasse und Analyse mit Begründung, warum das OCP erfüllt/nicht erfüllt wurde – falls erfüllt: warum hier sinnvoll/welches Problem gab es? Falls nicht erfüllt: wie könnte man es lösen (inkl. UML)?]

Positiv-Beispiel

Klasse:

ConsoleColorWriter

UML:



Begründung:

Die abstrakte Klasse ConsoleColorWriter kümmert sich um die Farbe des ausgegebenen Textes in der Konsole. Diese erfüllt das OCP, da diese nicht modifiziert werden muss, um zusätzliche Farben hinzuzufügen. Es ist ausreichend, die Funktionalitäten dieser abstrakten Klasse zu vererben. Somit muss für jede neue Farbe nur eine weitere Klasse mit entsprechender Ausgabe angelegt werden. Somit ist diese Klasse geschlossen für Veränderungen aber offen für Erweiterungen.

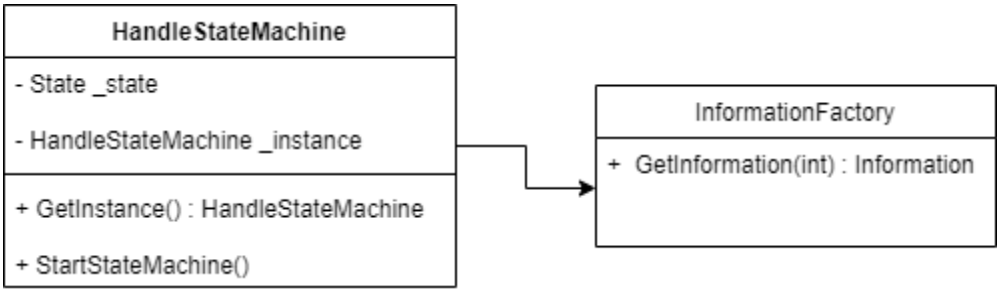
Es ist hier sinnvoll das OCP anzuwenden, da es bei der weiteren Entwicklung dazu kommen kann, dass mehr Farben benötigt werden.

Negativ-Beispiel

Klasse:

InfoHelper

UML:



Begründung:

Diese Klasse muss für jede zusätzliche Info, die angezeigt werden soll, verändert werden. Somit erfüllt diese nicht das OCP Prinzip.

Man könnte dieses Prinzip lösen, indem man eine abstrakte Klasse “InfoWriter”” hinzufügt und die anzuzeigenden Texte in Unterklassen wie z.B. “ShowLoginInfo” auslagert.

Diese Klasse erbt dann die Eigenschaften von “InfoWriter”. Somit muss die Basisklasse nicht verändert, sondern nur durch Unterklassen erweitert werden.

Analyse [LSP/ISP/DIP] (2P)

[jeweils eine Klasse als positives und negatives Beispiel für entweder LSP oder ISP oder DIP); jeweils UML der Klasse und Begründung, warum man hier das Prinzip erfüllt/nicht erfüllt wird]

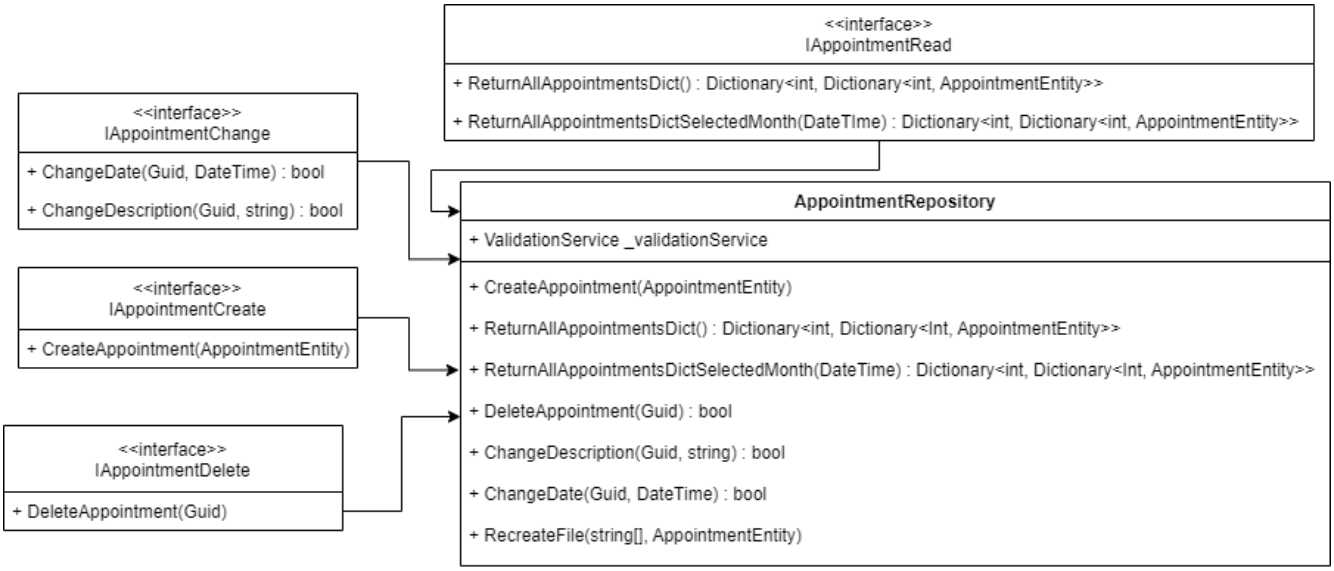
[Anm.: es darf nur ein Prinzip ausgewählt werden; es darf NICHT z.B. ein positives Beispiel für LSP und ein negatives Beispiel für ISP genommen werden]

Positiv-Beispiel ISP

Klasse:

AppointmentRepository

UML:



Begründung:

Die Klasse AppointmentRepository implementiert nur die Interfaces, die diese benötigt. Dazu wurden die CRUD-Operationen in Interfaces aufgeteilt, sodass nur die Funktionen, die benötigt werden, implementiert werden.

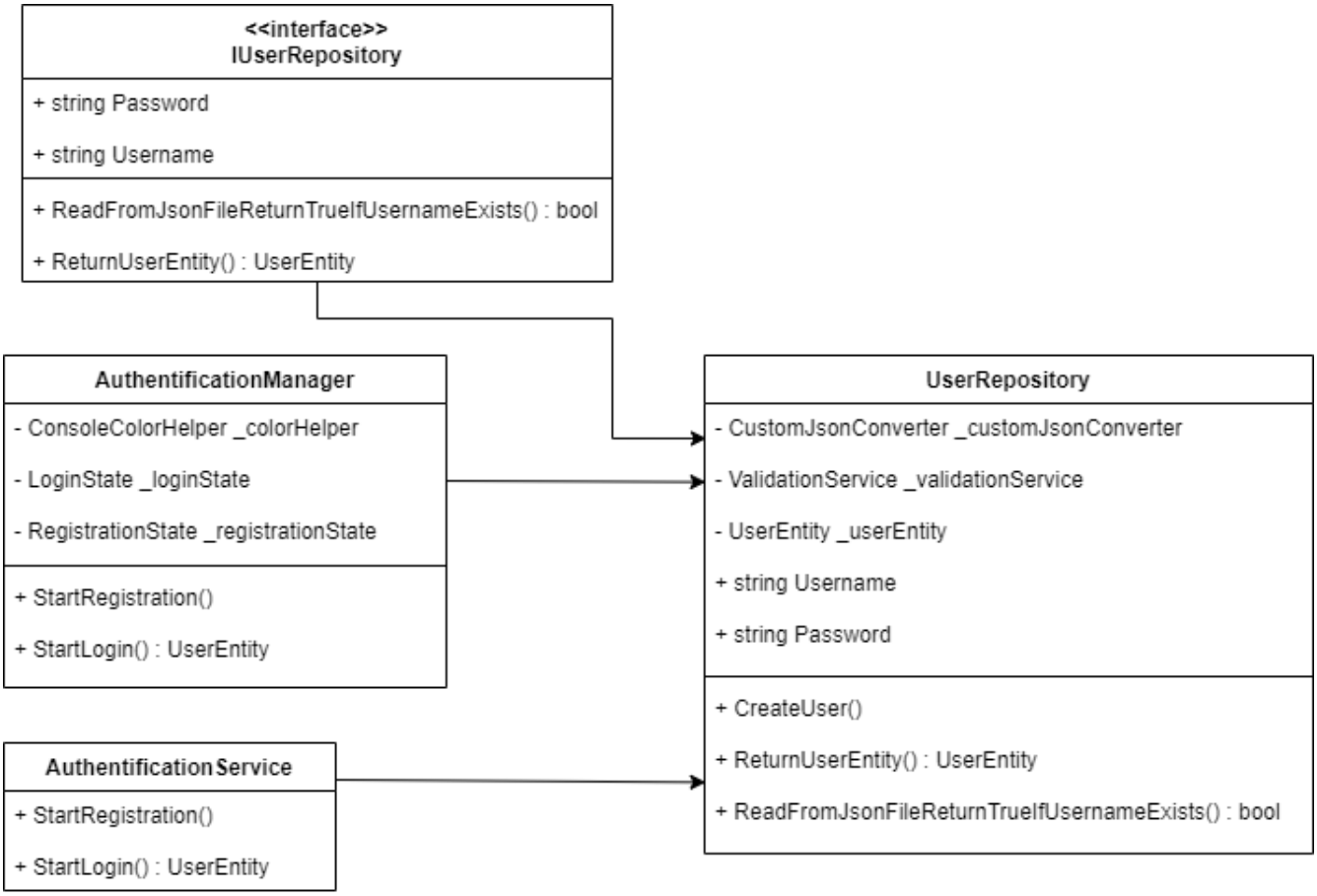
Wenn man beispielweise eine Datenbank für das Speichern der Daten als Backup anlegen möchte, müssen nur die Create-, und Read-Funktionen der entsprechenden Interfaces implementiert werden.

Negativ-Beispiel ISP

Klasse:

IUserRepository

UML:



Begründung:

Das Interface IUserRepository erfüllt nicht das ISP Prinzip, da dieses Interface einmal die Attribute der User für das Erstellen dieser implementiert und gleichzeitig noch die Read-Funktionen zum Auslesen der User.

Falls man nun aber eine Datenbank als Backup implementieren möchte, benötigt man die Funktion “ReadFromJsonFileReturnTrueIfUsernameExists” nicht.

Kapitel 4: Weitere Prinzipien (8P)

Analyse GRASP: Geringe Kopplung (4P)

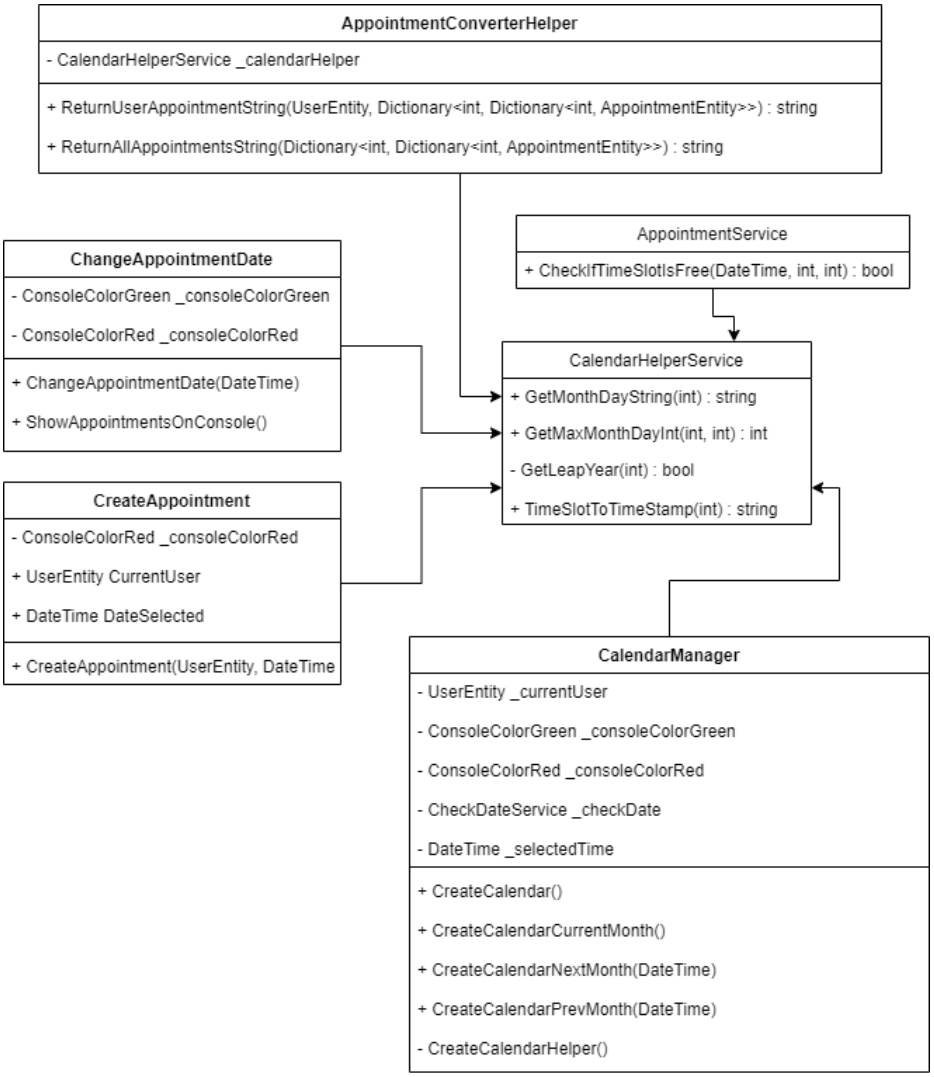
[jeweils eine bis jetzt noch nicht behandelte Klasse als positives und negatives Beispiel geringer Kopplung; jeweils UML Diagramm mit zusammenspielenden Klassen, Aufgabenbeschreibung der Klasse und Begründung, warum hier eine geringe Kopplung vorliegt bzw. Beschreibung, wie die Kopplung aufgelöst werden kann]

Positiv-Beispiel

Klasse:

CalendarHelperService

UML:



Begründung:

Die Klasse CalendarHelperService hat eine geringe Kopplung da die Methoden GetMaxMonthDayInt und GetLeapYear sich auf grundsätzliche Berechnung der maximalen Anzahl der Tage und auf die Berechnung von Schaltjahren konzentrieren.

Diese Formeln der Berechnung ändern sich nicht.

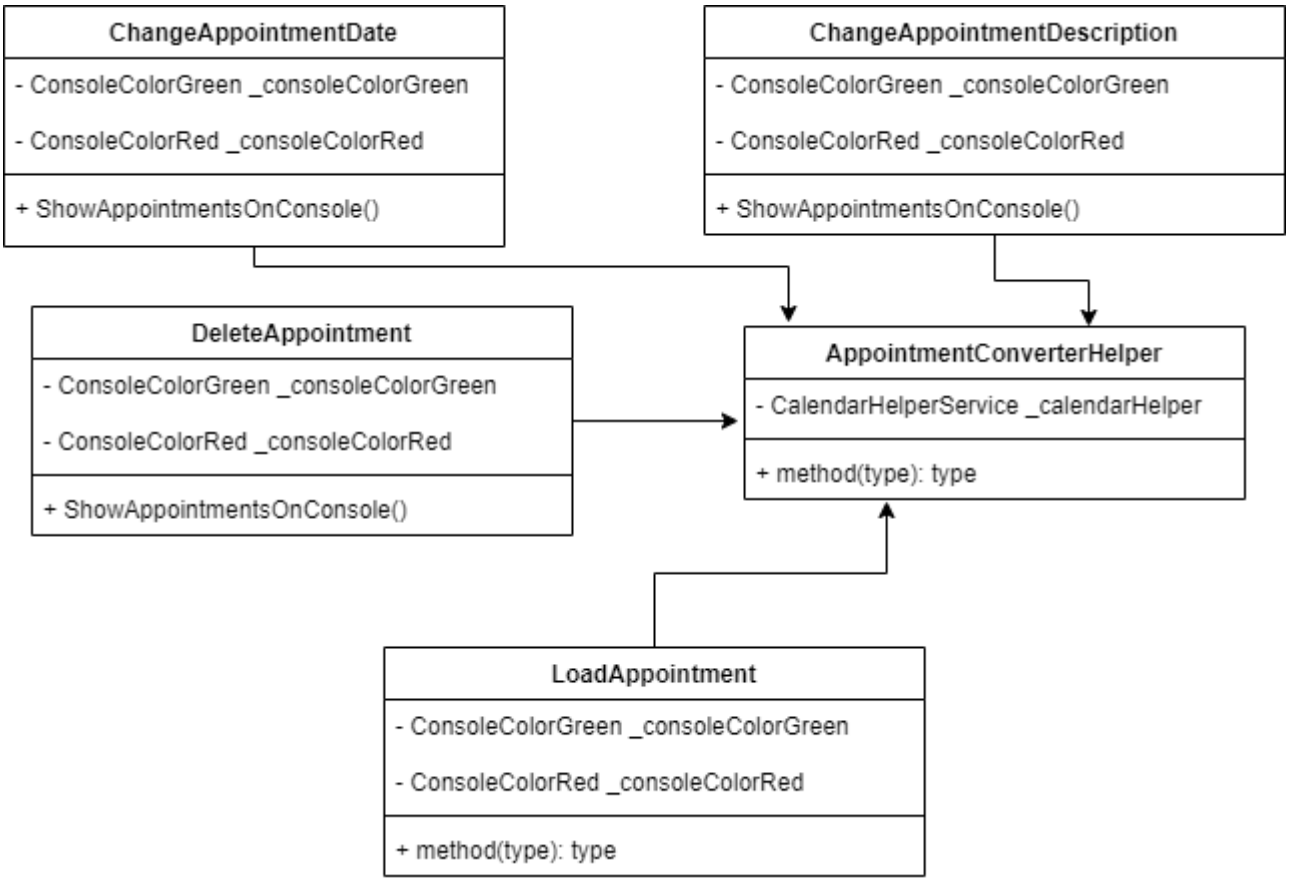
Eine geringe Kopplung besteht bei der Methode TimeSlotToTimeStamp, falls sich die Zeit-Slots ändern, muss es in dieser Methode angepasst werden. Da aber sonst keine Abhängigkeiten bestehen erfüllt die Klasse die Anforderungen an eine geringe Kopplung.

Negativ-Beispiel

Klasse:

AppointmentConverterHelper

UML:



Beschreibung:

Die Klasse AppointmentConverterHelper ist stark abhängig vom Aufbau der JSON-Datei der Appointments. Wenn sich die Struktur oder der Aufbau ändert, muss auch die Funktion der Klasse geändert werden. Somit ist eine geringe Kopplung nicht erfüllt.

Man könnte die Abhängigkeit auflösen, indem man der Klasse einen Übersetzer hinzufügt, der das Dictionary, das aus der JSON-Datei erzeugt wird, übersetzt und immer dieselbe Struktur zurückgibt, egal in welcher Reihenfolge oder Sortierung die Parameter in der JSON-Datei enthalten sind.

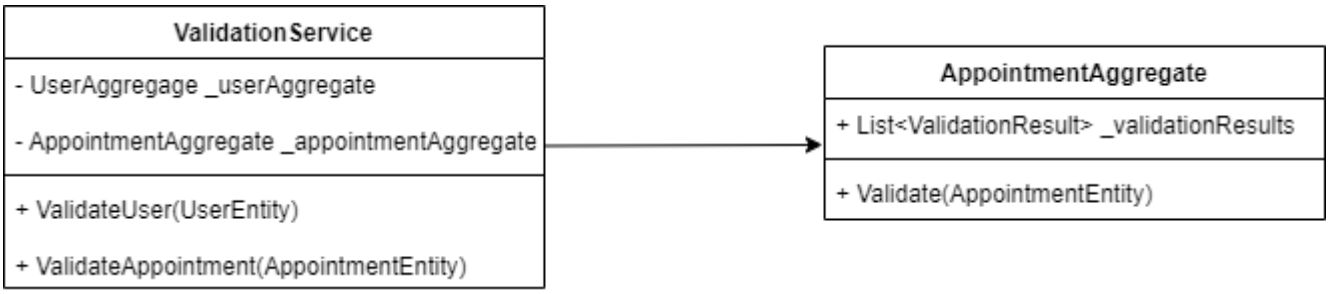
Analyse GRASP: Hohe Kohäsion (2P)

[eine Klasse als positives Beispiel hoher Kohäsion; UML Diagramm und Begründung, warum die Kohäsion hoch ist]

Klasse:

AppointmentAggregate

UML:



Begründung:

Das AppointmentAggregate besitzt ein Feld mit einer Liste von ValidationResults.

Die Methode Validate() besitzt als Parameter ein AppointmentEntity.

Die Validate() Methode überprüft nun die Werte der Entität und sollten gewisse Faktoren nicht erfüllt sein, so wird der Liste ein Eintrag vom Typ ValidationResult hinzugefügt.

Jedes Feld und jeder Parameter haben eine Zugehörigkeit, daher weist diese Klasse eine hohe Kohäsion auf.

DRY (2P)

[ein Commit angeben, bei dem duplizierter Code/duplizierte Logik aufgelöst wurde; Code-Beispiele (vorher/nachher); begründen und Auswirkung beschreiben]

Commit:

https://github.com/Paprikawurst/ASE_Calendar/commit/b677335b68689155744d6229a4b5d52213d13a66

```
172     -
173     -           File.Delete(AppDomain.CurrentDomain.BaseDirectory + "ASECalendarAppointments.json");
174     -
175     -           foreach (var subString in jsonSplit)
176     -           {
177     -               if (subString != "")
178     -               {
179     -                   var customJsonConverter = new CustomJsonConverter<AppointmentEntity>();
180     -                   var appointment = customJsonConverter.DeserializeObject(subString);
181     -                   CreateAppointment(appointment);
182     -               }
183     -           }
184     -
185     -           return "";
    171 +           RecreateFile(jsonSplit);
    172 +           return true;
```

Ein Teil des Codes wurde entfernt und in eine Methode ausgelagert:

RecreateFile(jsonSplit)

```
    257 +       /// <summary>
    258 +       /// Deletes ASECalendarAppointments.json and recreates it with given json-Objects
    259 +       /// </summary>
    260 +       /// <param name="jsonSplit"></param>
    261 +       private void RecreateFile(string[] jsonSplit)
    262 +       {
266 263             File.Delete(AppDomain.CurrentDomain.BaseDirectory + "ASECalendarAppointments.json");
267 264
268 265             foreach (var subString in jsonSplit)
269 266             {
270 267                 if (subString != "")
271 268                 {
272     -                   var customJsonConverter = new CustomJsonConverter<AppointmentEntity>();
273     -                   var appointment = customJsonConverter.DeserializeObject(subString);
274     -                   CreateAppointment(appointment);
    269 +                   var appointmentEntity = _customJsonConverter.DeserializeObject(subString);
    270 +                   CreateAppointment(appointmentEntity);
275 271                 }
276 272             }
```

Diese neue Methode wird an allen Stellen aufgerufen, wo sie gebraucht wird (siehe Commit) und somit wurde die komplette Klasse kleiner und übersichtlicher

Kapitel 5: Unit Tests (8P)

10 Unit Tests (2P)

[Nennung von 10 Unit-Tests und Beschreibung, was getestet wird]

Unit Test	Beschreibung
UnitTestAppointmentRepository# CreateAndReturnAppointmentTest	Es wird getestet, ob ein Appointment erstellt wird und ob dies ausgelesen werden kann. Da die beiden Methoden voneinander abhängig sind werden diese zusammen getestet.
UnitTestAppointmentRepository# ChangeAppointmentDateTest	Es wird getestet, ob die Methode ChangeDate das Datum eines Appointments ändert.
UnitTestAppointmentRepository# ChangeAppointmentDescriptionTest	Es wird getestet, ob die Methode ChangeDescription die Beschreibung eines Appointments ändert.
UnitTestAppointmentRepository# DeleteAppointmentTest	Es wird getestet, ob die Methode DeleteAppointment ein Appointment erfolgreich löscht.
UnitTestUserRepository# CreateAndReturnUserTest	Es wird getestet, ob ein User erstellt wird und ob dieser ausgelesen werden kann. Da die beiden Methoden voneinander abhängig sind werden diese zusammen getestet.
UnitTestUserRepository# ReturnTrueIfUsernameExistsTest	Es wird getestet, ob die Methode ReadFromJsonFileReturnTrueIfUsernameExists True zurückgibt, wenn der übergebene Username existiert.
UnitTestCheckDateService# NotEqualTest	Es wird geprüft, ob die Methode Check das richtige Datum zurückgibt, hier ein unverändertes Datum.
UnitTestCheckDateService# IncreaseTest	Es wird geprüft, ob die Methode Check das Jahr um eins erhöht.
UnitTestCheckDateService# DecreaseTest	Es wird überprüft, ob die Methode Check das Jahr um eins verringert.
UnitTestValidationService# AppointmentAggregateTest	Es wird überprüft, ob ein korrektes Appointment ohne Fehlermeldungen validiert wird.

ATRIP: Automatic (1P)

[Begründung/Erläuterung, wie 'Automatic' realisiert wurde]

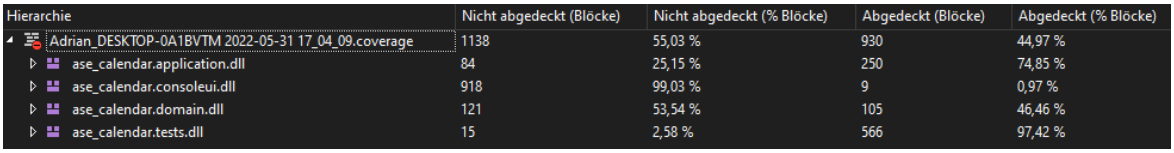
In Visual Studio können Tests automatisch durchgeführt und die Ergebnisse dargestellt werden. Die Ergebnisse werden durch "Assert" Klasse automatisch auf Richtigkeit der Werte geprüft.

Die Erkennung von Unit Tests erfolgt über Notationen des xUnit Test-Tool.

ATRIP: Thorough (1P)

[Code Coverage im Projekt analysieren und begründen]

Wir haben eine Testabdeckung von 44,97%.
Es wurden für die wichtigsten Klassen wie z.B. das Appointment- und User- Repository Tests erstellt. Diese Klassen verwalten das Anlegen, Auslesen, Ändern und Löschen von Appointments und das Anlegen und Auslesen von Usern. Diese Klassen decken also die Grundfunktion unserer Anwendung ab.



Hierarchie	Nicht abgedeckt (Blöcke)	Nicht abgedeckt (% Blöcke)	Abgedeckt (Blöcke)	Abgedeckt (% Blöcke)
Adrian_DESKTOP-0A1BVTM 2022-05-31 17_04_09.coverage	1138	55,03 %	930	44,97 %
ase_calendar.application.dll	84	25,15 %	250	74,85 %
ase_calendar.consoleui.dll	918	99,03 %	9	0,97 %
ase_calendar.domain.dll	121	53,54 %	105	46,46 %
ase_calendar.tests.dll	15	2,58 %	566	97,42 %

Wie man erkennen kann, ist vor allem die Application-Layer und die Domain-Layer von Tests abgedeckt, welche die Appointment-, und User-Repository Klassen enthalten. Die UI-Layer ist kaum von Tests abgedeckt da hier sehr viel User-Input benötigt wird. Der User-Input wird jedoch nach jeder Eingabe auf Richtigkeit geprüft.

ATRIP: Professional (1P)

[jeweils 1 positives und negatives Beispiele zu 'Professional'; jeweils Code-Beispiel, Analyse und Begründung, was professionell/nicht professionell an den Beispielen ist]

Positives Code-Beispiel:

```
[TestMethod]
0 references
public void ReturnTrueIfUsernameExistsTest()
{
    // Arrange

    if (File.Exists(path:AppDomain.CurrentDomain.BaseDirectory + "ASECalendarUsers.json"))
    {
        File.Delete(path:AppDomain.CurrentDomain.BaseDirectory + "ASECalendarUsers.json");
    }

    UserEntity user = new UserEntity(username:"Adrian", password:"12345", roleId:0, Guid.NewGuid());
    UserRepository userRepository = new UserRepository(user);
    userRepository.Username = "Adrian";
    userRepository.Password = "12345";

    // Act
    bool exists = userRepository.ReadFromJsonFileReturnTrueIfUsernameExists();

    //Assert
    Assert.IsTrue(exists);

}
```

Erklärung und Begründung:

Der Arrange-Teil übernimmt die Initialisierung der Instanzen.

Der Act-Teil überprüft die Ausführung der Methode ReadFromJsonFileReturnTrueIfUsernameExists(), diese überprüft jeden Eintrag der Json-Datei auf eine vorher initialisierte Usernamenvariable. Sollte ein Eintrag der Json-Datei mit der Variable übereinstimmen, wird ein „true" als bool-Wert zurückgegeben.

Der Test überprüft hauptsächlich eigens implementierte Logik. Es wird eine ausreichend komplexe Umgebung erstellt und daraufhin ein Überprüfungsdurchlauf gestartet,

```
public bool ReadFromJsonFileReturnTrueIfUsernameExists()
{
    if (File.Exists(path:AppDomain.CurrentDomain.BaseDirectory + "ASECalendarUsers.json"))
    {
        var json:string = File.ReadAllText(path:AppDomain.CurrentDomain.BaseDirectory + "ASECalendarUsers.json");
        var jsonSplit:string[] = json.Split(separator:"\n");

        foreach (var subString in jsonSplit)
        {
            var userEntity = _customJsonConverter.DeserializeObject(subString);

            if (userEntity != null)
            {
                if (userEntity.UserDataRegistered.Username == Username)
                {
                    return true;
                }
            }
        }
    }

    return false;
}
```

Negatives Code-Beispiel:

```
0 references
public void DecreaseTest()
{
    // Arrange
    int year = 2022;
    int month = 0;

    CheckDateService checkDateService = new CheckDateService(year, month);

    // Act
    var checkedDate = checkDateService.Check();

    //Assert
    Assert.AreEqual(expected:2021, actual:checkedDate.Year, message:"Year did not decrease");
    Assert.AreEqual(expected:12, actual:checkedDate.Month, message:"Month did not decrease");
}
```

Erklärung und Begründung:

Der DecreaseTest-Test überprüft, ob ein Jahr bei einem Wert der Variable „month“ richtig aufaddiert bzw. absubtrahiert wird. Ist der Wert der Variable größer als 12, wird das Jahr um 1 addiert und der Monat auf 1 gesetzt. Anderenfalls, wenn der Wert der Variable kleinergleich 0 ist, wird das Jahr um eins subtrahiert und der Monat auf 12 gesetzt.

Bei diesem Test wird implizit getestet, ob die mathematischen Operatoren der Methode richtig funktionieren. Da dieser Test keine komplexeren Strukturen der Anwendung enthält, ist es unwahrscheinlich, dass er in zukünftigen Versionen der Anwendung fehlschlagen wird.

```
public DateTime Check()
{
    if (_month > 12)
    {
        _year += 1;
        _month = 1;
    }

    if (_month <= 0)
    {
        _year -= 1;
        _month = 12;
    }

    return new DateTime(_year, _month, day:1);
}
```

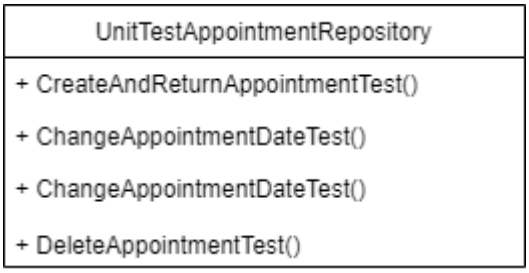
Fakes und Mocks (3P)

[Analyse und Begründung des Einsatzes von 2 Fake/Mock-Objekten; zusätzlich jeweils UML Diagramm der Klasse]

Klasse:

UnitTestAppointmentRepository

UML:



Begründung:

In der Klasse UnitTestAppointmentRepository wurden zwei Fake Objekte eingesetzt.

1. FakeUser
2. FakeAppointment

Diese Fakes wurden erstellt, da diese Objekte normalerweise während der Laufzeit durch eine Eingabe des Benutzers erstellt werden.

FakeUser simuliert einen Namen, ein Passwort und eine UserID. Das UserEntity-Objekt benötigt man für die Erstellung eines Appointments, da hier die UserId im Appointment gespeichert wird.

FakeAppointment simuliert das Datum, die gegebene UserId, eine AppointmentId und die Beschreibung des Appointments.

Kapitel 6: Domain Driven Design (8P)

Ubiquitous Language (2P)

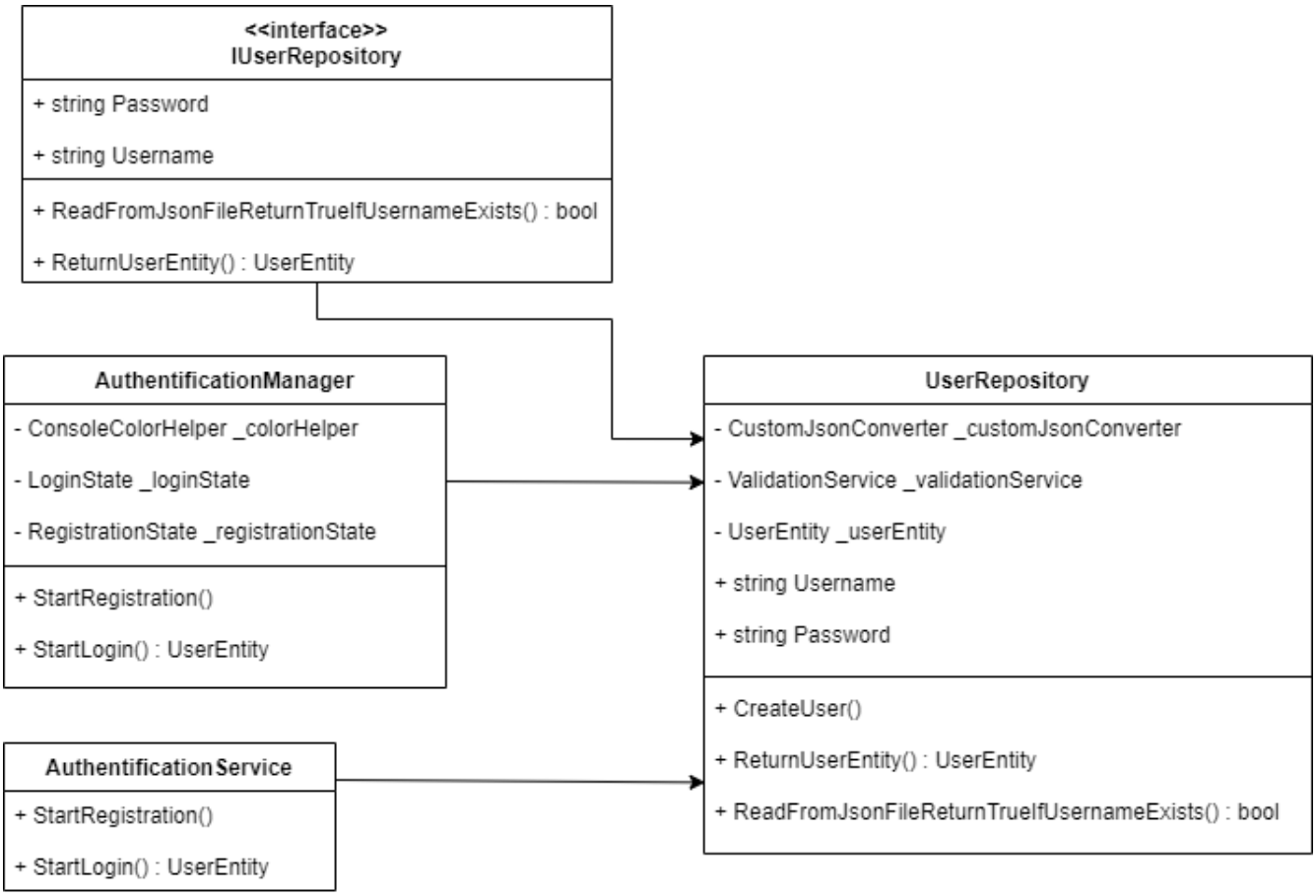
[4 Beispiele für die Ubiquitous Language; jeweils Bezeichnung, Bedeutung und kurze Begründung, warum es zur Ubiquitous Language gehört]

Bezeichung	**Bedeutung**	**Begründung**
Appointment	Termin im Kalender mit Beschreibung, Zeitslot, Nutzer-ID und eigener ID	Ein Sekretär hat andere Auffassungen von einem Termin, Agenda des Termins, Kundenhintergrund etc. während für den Entwickler nur wichtig ist, dass ein Termin z.B. eine Beschreibung und einen verknüpften Nutzer etc. besitzt
User	Anwendungsnutzer in der Kalenderanwendung	Für den Entwickler ist nur die Nutzer-ID wichtig, für einen Mechaniker ist bei einem Kunden wichtig, welches Fahrzeug er angemeldet hat oder welche Wartungen anstehen.
Role	Nutzerrolle in der Anwendung (Admin, Car dealer, Customer)	Eine Rolle eines Nutzer kann vieles bedeuten: CEO eines Unternehmens, Putzkraft, Besucher, Kunde. Hier ist es wichtig festzulegen, dass eine Nutzerrolle in der Anwendung gemeint ist und dass es sich lediglich um die Berechtigungen innerhalb der Anwendungen handelt.
TimeSlot	Zeitraum eines Kalendertermins	Ein Zeitraum eines Kalendertermins muss klar kommuniziert sein. Da ein „TimeSlot“ einen Zeitraum von Tagen, Stunden oder Minuten bezeichnen kann, muss sich jeder im Klaren sein, dass es sich hier um einen der acht Zeiträume bei der Erstellung eines Termins handelt.

Repositories (1,5P)

[UML, Beschreibung und Begründung des Einsatzes eines Repositories; falls kein Repository vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist]

UML:



Beschreibung:
Das UserRepository verwaltet alle CRUD-Operationen für die User-Entitäten. Hier findet sich die Anbindung zur Datenspeicherung, in unserem Fall eine Json-Datei.
Der CustomJsonConverter ist hierbei für die Serialisierung und Deserialisierung der Json-Objekte zuständig.
Der ValidationService ist für die Validierung der User-Entitäten zuständig. Mithilfe dieses Services soll verhindert oder zumindest dokumentiert werden, wenn fehlerhafte Entitäten gespeichert werden.

Die CreateUser()-Methode ist die Create-Operation des Repositoriums. Sie kümmert sich um die Verarbeitung einer User-Entität zum korrekten Json-Format und der Speicherung dieser in die entsprechende Json-Datei.

Die ReturnUserEntity()-Methode ist eine der beiden Read-Operationen des Repositoriums. Sie überprüft eingegebene Logindaten auf Übereinstimmung innerhalb der Json-Datei und gibt eine User-Entität zurück, sollten die Logindaten mit den Daten der User-Entität übereinstimmen.

Die ReadFromJsonFileReturnTrueIfUsernameExists()-Methode ist die zweite der beiden Read-Operationen des Repositoriums. Sie überprüft, ob ein gegebener Nutzername in den gespeicherten Json-Objekten enthalten ist und gibt einen entsprechenden bool-Wert zurück.

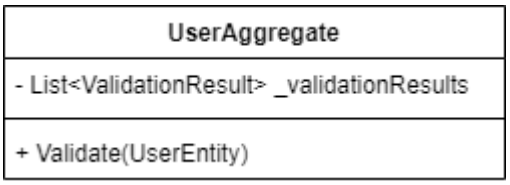
Update-Operationen sowie Delete-Operationen sind für das Repositorium nicht notwendig gewesen, da es in dem momentanen Stand der Anwendung keine Möglichkeit gibt, beispielsweise ein Passwort zu ändern oder einen Nutzereintrag komplett zu löschen.

Begründung:
Ein Repositorium in diesem Umfang ist für unsere Anwendung hilfreich, da es die kompletten Json-Dateiabfragen übernimmt. Es bündelt alle CRUD-Operationen zusammen und lässt uns diese, überall wo diese benötigt werden, einsetzen

Aggregates (1,5P)

[UML, Beschreibung und Begründung des Einsatzes eines Aggregates; falls kein Aggregate vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist]

UML:



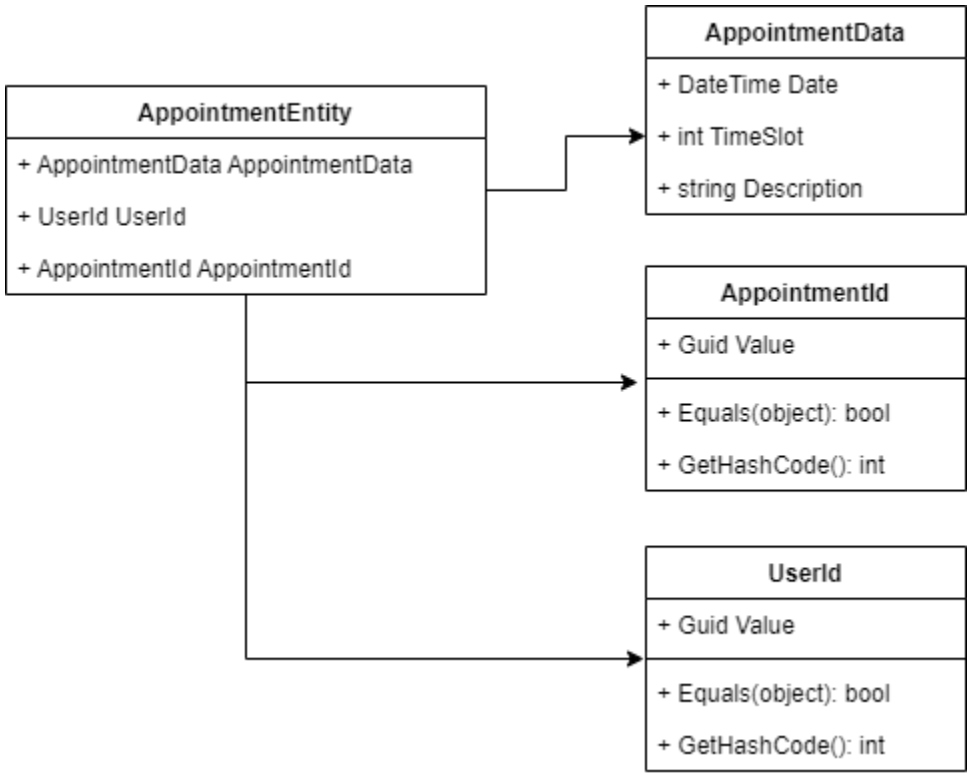
Beschreibung:
Das UserAggregate übernimmt die Definition der Regeln einer Entität.
In unserem Fall enthält das UserAggregate eine Validate(UserEntity)-Methode, welche ein UserEntity als Eingabeparameter nimmt, dieses überprüft, und anschließend eine Liste von ValidationResults erzeugt, welche alle invaliden Werte der UserEntität beinhaltet.
Die Validate-Methode überprüft die UserEntität auf UserId, Usernmae, UserData-Objekt, Passwort, sowie UserRoleId.

Begründung:
Das UserAggregate soll alle Methoden und Regeln einer Entität bündeln. In unserem Falle übernimmt es lediglich die Definition der Regeln einer Entität, sodass der ValidationService auf diese zugreifen kann und die Validate-Methode des Aggregates ausführen kann, überprüfen kann, ob die ValidationResults-Liste gültige Werte liefert und dann mit der Verarbeitung der Entität fortzufahren.

Entities (1,5P)

[UML, Beschreibung und Begründung des Einsatzes einer Entity; falls keine Entity vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist]

UML:



Beschreibung:

Die AppointmentEntität soll einen Termin im Kalender darstellen. Sie enthält eine eindeutige ID, eine ID des Nutzers, welcher den Termin angelegt hat, sowie ein AppointmentData-Objekt, mit Datum, Zeitraum und Beschreibung des Termins.

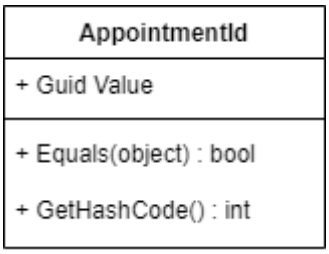
Begründung:

Die Entität soll einen Termin im Kalender so genau wie möglich darstellen. Sie ist ein Bestandteil der Domäne des Projekts und wird dementsprechend oft verwendet.

Value Objects (1,5P)

[UML, Beschreibung und Begründung des Einsatzes eines Value Objects; falls kein Value Object vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist]

UML:



Beschreibung:
Das Value Object „AppointmentId“ beschreibt die eindeutige ID eines Termins im Kalender. Dieses Value Object besitzt keine eindeutige ID, es unterscheidet sich also lediglich im Wert der Variable „Value“ von anderen „AppointmentId“-Value Objects.
Es besitzt zwei Methoden Equals() und GetHashCode(), welche zur Verarbeitung und Unterscheidung des Value Objects benutzt werden.

Diese Value Objects sind unveränderbar, sobald sie einmal erstellt worden sind.

Begründung:
Value Objects werden von Entitäten benutzt, um einzelne Werte zu beschreiben, für welche man eventuelle Helfermethoden benötigt. In unserem Fall haben wir sie eingebaut, um zukünftige Erweiterbarkeit zu gewährleisten. Momentan erfüllen die Value Objects keinen direkten Zweck.

Kapitel 7: Refactoring (8P)

Code Smells (2P)

[jeweils 1 Code-Beispiel zu 2 unterschiedlichen Code Smells aus der Vorlesung; jeweils Code-Beispiel und einen möglichen Lösungsweg bzw. den genommen Lösungsweg beschreiben (inkl. (Pseudo-)Code)]

Code Smell: Large Class

Klasse:

ChangeAppointmentDate

Code-Beispiel:

```
namespace ASE_Calendar.ConsoleUI.ConsoleOptions.Managers.Appointment
{
    /// <summary>
    ///     A class which starts the process on the ui to change the date of an
appointment.
    /// </summary>
    public class ChangeAppointmentDate
    {
        private readonly ConsoleColorGreen _consoleColorGreen = new();
        private readonly ConsoleColorRed _consoleColorRed = new();

        public ChangeAppointmentDate(DateTime selectedTime)
        {
            var changeDateAppointmentSate =
ChangeAppointmentDateState.CheckForAppointments;
            var appointmentGuid = Guid.Empty;
            var newDateTime = new DateTime();
            var inputGuidString = "";
            var appointmentDayString = "";
            var appointmentMonthString = "";
            var appointmentYearString = "";
            var appointmentTimeSlotString = "";
            short appointmentDayInt = 0;
            short appointmentMonthInt = 0;
            short appointmentYearInt = 0;
            var appointmentRepository = new AppointmentRepository();

            switch (changeDateAppointmentSate)
            {
                case ChangeAppointmentDateState.CheckForAppointments:

                    var appointmentDict =
appointmentRepository.ReturnAllAppointmentsDict();
                    AppointmentConverterHelper appointmentConverter = new
AppointmentConverterHelper();
                    string appointmentsString =
appointmentConverter.ReturnAllAppointmentsString(appointmentDict);

                    if (appointmentsString == null)
                    {
                        _consoleColorRed.WriteLine("\nThere are no appointments at the
moment!");
                        _consoleColorRed.WriteLine("Any key to continue.");
                        Console.ReadLine();
                        break;
                    }

                    goto case ChangeAppointmentDateState.UserInputId;

                case ChangeAppointmentDateState.UserInputId:

                    ShowAppointmentsOnConsole();
                    Console.WriteLine("\nEnter the appointment ID you want to change
the date:");
                    inputGuidString = Console.ReadLine();
                    goto case ChangeAppointmentDateState.CheckInputId;

                case ChangeAppointmentDateState.UserInputDay:

                    Console.WriteLine("Enter the new day:");
                    appointmentDayString = Console.ReadLine();

                    goto case ChangeAppointmentDateState.CheckInputDay;
```

```
                case ChangeAppointmentDateState.UserInputMonth:

                    Console.WriteLine("Enter the new month:");
                    appointmentMonthString = Console.ReadLine();

                    goto case ChangeAppointmentDateState.CheckInputMonth;

                case ChangeAppointmentDateState.UserInputYear:

                    Console.WriteLine("Enter the new year:");
                    appointmentYearString = Console.ReadLine();

                    goto case ChangeAppointmentDateState.CheckInputYear;

                case ChangeAppointmentDateState.UserInputTimeSlot:

                    Console.WriteLine(
                        "Please select a timeslot which is free on the selected
day:\n08:00 - 09:00 = 1\n09:00 - 10:00 = 2\n10:00 - 11:00 = 3\n11:00 - 12:00 =
4\n13:00 - 14:00 = 5\n14:00 - 15:00 = 6\n15:00 - 16:00 = 7\n16:00 - 17:00 = 8\n");
                    appointmentTimeSlotString = Console.ReadLine();

                    goto case ChangeAppointmentDateState.CheckInputTimeSlot;

                case ChangeAppointmentDateState.CheckInputId:

                    var isValid = Guid.TryParse(inputGuidString, out appointmentGuid);

                    if (isValid)
                    {
                        goto case ChangeAppointmentDateState.UserInputDay;
                    }

                    _consoleColorRed.WriteLine("Please enter a valid guid!");

                    goto case ChangeAppointmentDateState.UserInputId;

                case ChangeAppointmentDateState.CheckInputDay:

                    var isValidDay = short.TryParse(appointmentDayString, out
appointmentDayInt);

                    if (!isValidDay || appointmentDayInt > 31 || appointmentDayInt <=
0)
                    {
                        _consoleColorRed.WriteLine("Please enter a valid day!");
                        goto case ChangeAppointmentDateState.UserInputDay;
                    }

                    goto case ChangeAppointmentDateState.UserInputMonth;

                case ChangeAppointmentDateState.CheckInputMonth:

                    var isValidMonth = short.TryParse(appointmentMonthString, out
appointmentMonthInt);

                    if (!isValidMonth || appointmentMonthInt > 12 ||
appointmentMonthInt <= 0)
                    {
                        _consoleColorRed.WriteLine("Please enter a valid month!");
                        goto case ChangeAppointmentDateState.UserInputMonth;
                    }

                    goto case ChangeAppointmentDateState.UserInputYear;
```

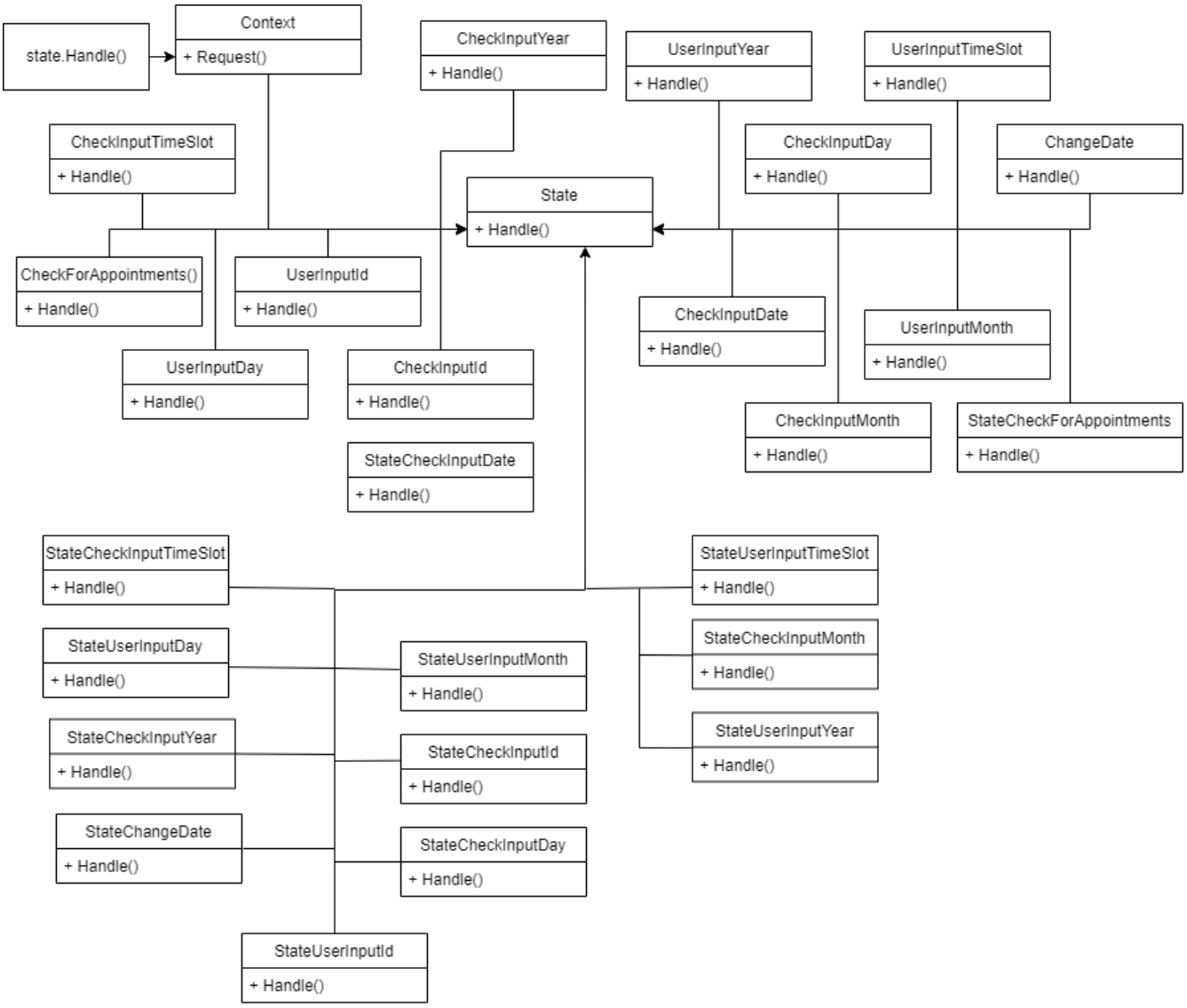
```
                case ChangeAppointmentDateState.CheckInputYear:

                    var isValidYear = short.TryParse(appointmentYearString, out
appointmentYearInt);

                    if (!isValidYear || appointmentYearInt < selectedTime.Year)
                    {
                        _consoleColorRed.WriteLine("Please enter a valid year!");
                        goto case ChangeAppointmentDateState.UserInputYear;
                    }

                    goto case ChangeAppointmentDateState.CheckInputDate;

                case ChangeAppointmentDateState.CheckInputTimeSlot:

                    var isNumber = Regex.IsMatch(appointmentTimeSlotString, @"[1-8]");

                    if (!isNumber || appointmentTimeSlotString == "")
                    {
                        _consoleColorRed.WriteLine("\n" + "Please enter a correct time
slot!" + "\n");
                        goto case ChangeAppointmentDateState.UserInputTimeSlot;
                    }

                    if (!AppointmentService.CheckIfTimeSlotIsFree(newDateTime,
short.Parse(appointmentTimeSlotString),
                            appointmentDayInt))
                    {
                        _consoleColorRed.WriteLine("\n" + "Time slot is occupied!" +
"\n");
                        goto case ChangeAppointmentDateState.UserInputTimeSlot;
                    }

                    goto case ChangeAppointmentDateState.ChangeDate;


                case ChangeAppointmentDateState.CheckInputDate:

                    if (CalendarHelperService.GetMaxMonthDayInt(appointmentMonthInt,
appointmentYearInt) <
                        appointmentDayInt)
                    {
                        _consoleColorRed.WriteLine("Please enter a valid date!");
                        goto case ChangeAppointmentDateState.UserInputDay;
                    }

                    newDateTime = new DateTime(appointmentYearInt,
appointmentMonthInt, appointmentDayInt);

                    goto case ChangeAppointmentDateState.UserInputTimeSlot;

                case ChangeAppointmentDateState.ChangeDate:

                    var successfulChangeDate =
appointmentRepository.ChangeDate(appointmentGuid, newDateTime);

                    if (successfulChangeDate)
                    {
                        _consoleColorGreen.WriteLine("The appointment has been
edited!");
                        _consoleColorGreen.WriteLine("Any key to continue!");
                        Console.ReadLine();
                    }
```

```
                    else
                    {
                        _consoleColorRed.WriteLine("There are no appointments booked
at the moment!");
                        _consoleColorRed.WriteLine("Any key to continue!");
                        Console.ReadLine();
                    }

                    break;
                }
            }

            /// <summary>
            ///     gets a list of appointment entities from the repository and outputs it
to the console
            /// </summary>
            public static void ShowAppointmentsOnConsole()
            {
                var appointmentRepository = new AppointmentRepository();
                var appointmentDict = appointmentRepository.ReturnAllAppointmentsDict();
                AppointmentConverterHelper appointmentConverter = new
AppointmentConverterHelper();
                string appointmentsString =
appointmentConverter.ReturnAllAppointmentsString(appointmentDict);

                Console.WriteLine("\n\n" + appointmentsString + "\n");
            }
        }
    }
```

Möglicher Lösungsweg:

Die Switch-Anweisung nach dem Design-Pattern “State” erstellen. So hat jeder Zustand eine eigene Klasse. Dies erhöht die Lesbarkeit, verkleinert die Klasse und erleichtert das Verständnis.

Pseudocode:


state.Handle()
Context
+ Request()
CheckInputYear
+ Handle()
UserInputYear
+ Handle()
UserInputTimeSlot
+ Handle()
CheckInputTimeSlot
+ Handle()
CheckInputDay
+ Handle()
ChangeDate
+ Handle()
State
+ Handle()
CheckForAppointments()
+ Handle()
UserInputId
+ Handle()
CheckInputDate
+ Handle()
UserInputMonth
+ Handle()
UserInputDay
+ Handle()
CheckInputId
+ Handle()
CheckInputMonth
+ Handle()
StateCheckForAppointments
+ Handle()
StateCheckInputDate
+ Handle()
StateCheckInputTimeSlot
+ Handle()
StateUserInputTimeSlot
+ Handle()
StateUserInputDay
+ Handle()
StateUserInputMonth
+ Handle()
StateCheckInputMonth
+ Handle()
StateCheckInputYear
+ Handle()
StateCheckInputId
+ Handle()
StateUserInputYear
+ Handle()
StateChangeDate
+ Handle()
StateCheckInputDay
+ Handle()
StateUserInputId
+ Handle()

Code Smell: Long Method

Klasse:

AppointmentRepository

Code-Beispiel:

```
public bool ReadFromJsonFileReturnTrueIfUsernameExists()
```

Möglicher Lösungsweg:

Die Methode kann aufgeteilt werden.

Hier kann beispielsweise erst die Json-Datei ausgelesen werden und anschließend in einer separaten Methode überprüft werden, ob ein ausgelesener Inhalt einen übereinstimmenden Nutzernamen besitzt.

Pseudocode:

ReadJsonFile()

ReturnTrueIfUsernameExists()

Hier kann bei der ReturnTrueIfUsernameExists-Methode auch die Variable mit dem Inhalt der gelesenen Json-Datei eingegeben werden, sodass klar wird, dass der Inhalt dieser Variable überprüft wird.

2 Refactorings (6P)

[2 unterschiedliche Refactorings aus der Vorlesung anwenden, begründen, sowie UML vorher/nachher liefern; jeweils auf die Commits verweisen]

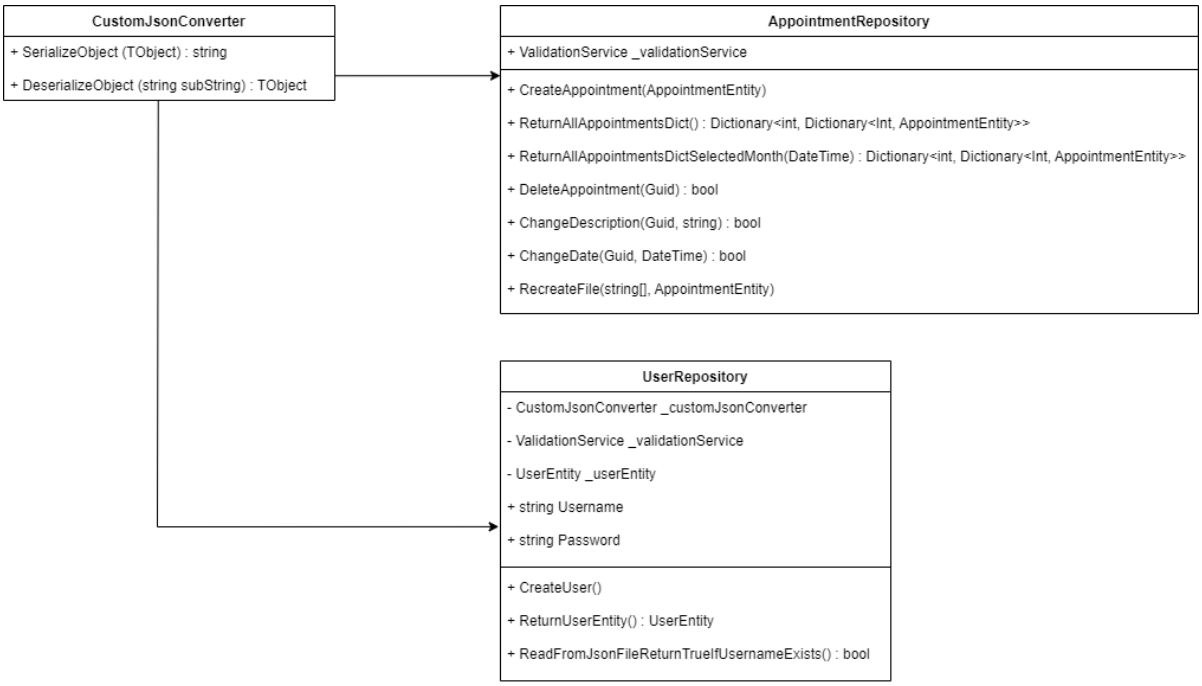
Extract Method:

Commit:

https://github.com/Paprikawurst/ASE_Calendar/commit/92b03f8e394b4ed896fec50abd67c97aec67c9b5

UML Vorher:

UML Nachher:



Beschreibung und Begründung:

Eine Funktion, welche in beiden Repositories verwendet wurde, wurde in eine eigene Klasse gezogen, um so doppelten Code zu vermeiden und die Komplexität zu verringern, sowie das Single-Responsibility Principle zu erfüllen. Ebenso verwendet die extrahierte Methode ein fremdes Framework, welches in einer eigenen Klasse ausgelagert einfach ausgetauscht werden kann.

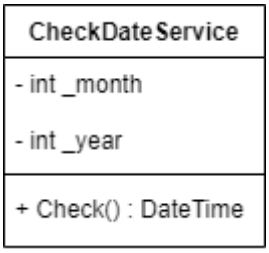
Rename Method:

Commit:

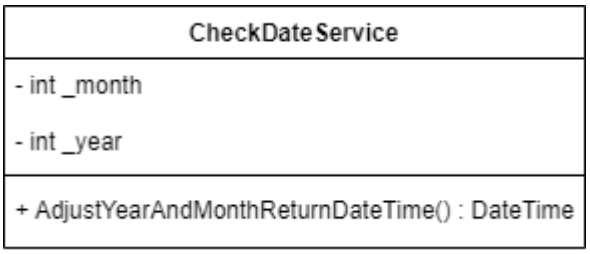
https://github.com/Paprikawurst/ASE_Calendar/commit/7dda52bfca63d6c0e7fe4047b371a263d13aec87

https://github.com/Paprikawurst/ASE_Calendar/commit/bf5af71e5d11fd0a164d62a6ad70a012480adb56

UML Vorher:



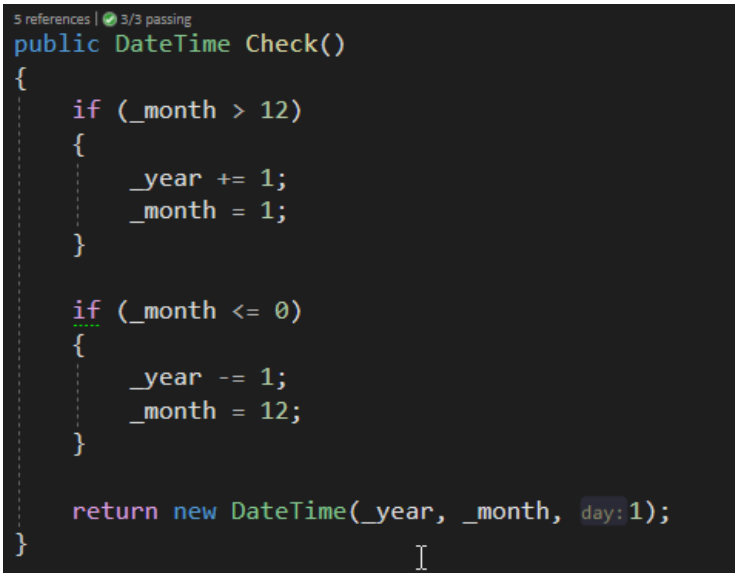
UML Nachher:



Beschreibung und Begründung:

Die ursprüngliche Check()-Methode des CheckDateService war nichtssagend und somit für den Entwickler nicht verständlich. Mit der Angabe des Rückgabewertes der Methode und einer besseren Beschreibung der Funktion der Methode ist diese verständlicher.

Zum Verständnis der Code der Methode:



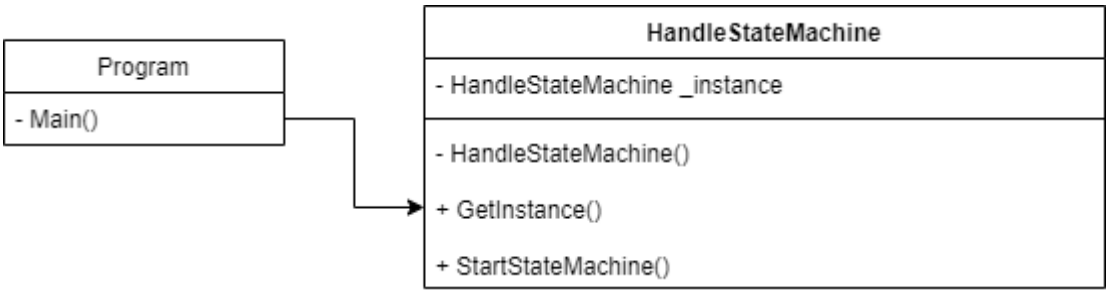
Kapitel 8: Entwurfsmuster (8P)

[2 unterschiedliche Entwurfsmuster aus der Vorlesung (oder nach Absprache auch andere) jeweils sinnvoll einsetzen, begründen und UML-Diagramm]

Entwurfsmuster: Singleton (4P)

Klasse:
HandleStateMachine

UML:



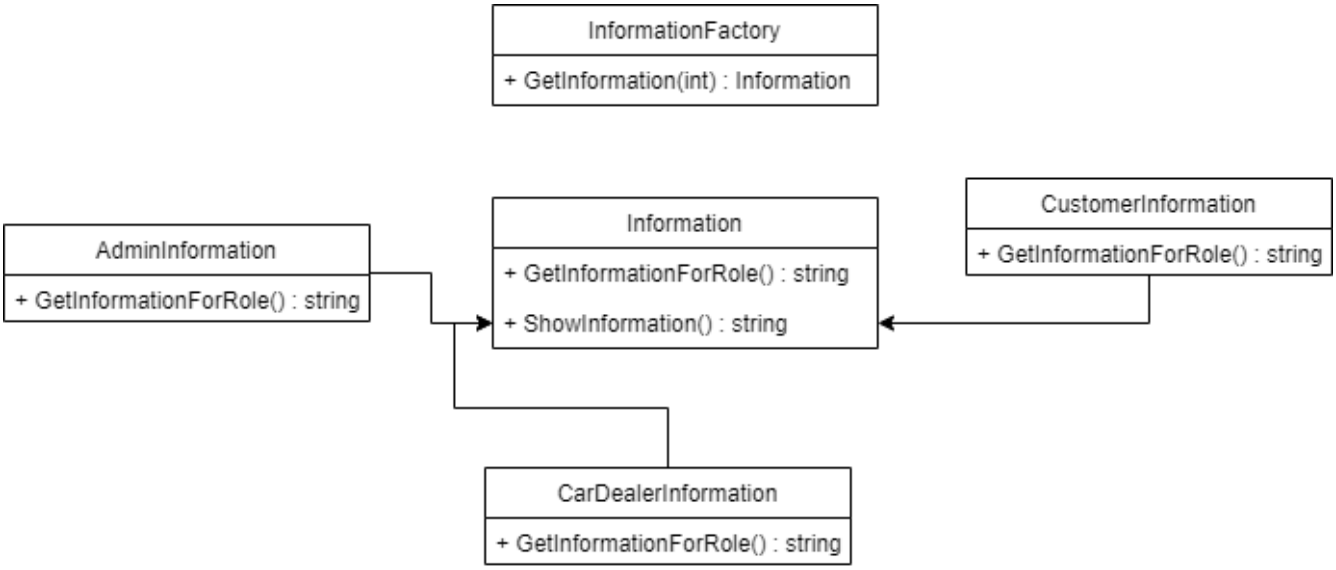
Begründung:
Die Klasse HandleStateMachine wurde als Singleton entworfen, da es immer nur eine Instanz dieses Objekts geben darf. Gibt es mehr als eine Instanz dieses Objekts, so kann es beispielsweise zu fehlerhaften Anzeigen innerhalb der Anwendung kommen.

Entwurfsmuster: abstract Factory (4P)

Klasse:

InformationFactory

UML:



Begründung:

Die Methode InformationFactory entscheidet je nach Rolle des Nutzers welcher Infotext angezeigt werden soll.

Hierzu wird, je nach Rolle, eines der Objekte AdminInformation, CarDealerInformation oder CustomerInformation aufgerufen und der entsprechende Infotext zurückgegeben.

Ein Factory-Pattern macht hier Sinn, da so die InformationFactory beliebig erweitert werden kann, sollten weitere Rollen oder Texte hinzukommen.