Programmentwurf Inventuranwendung

Name: Caesar, Mathias
Martrikelnummer: 4943537

Abgabedatum: 31.05.2023

Allgemeine Anmerkungen:

- *Gesamt-Punktzahl: 60P (zum Bestehen mit 4,0 werden 30P benötigt)*
- *die Aufgabenbeschreibung (der blaue Text) und die mögliche Punktanzahl muss im Dokument erhalten bleiben*
- *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)*
- *das Dokument muss als PDF abgegeben werden*
- *es gibt keine mündlichen Nebenabreden / Ausnahmen – alles muss so bearbeitet werden, wie es schriftlich gefordert ist*
- *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 vorne herein 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 sondern 0,5P Abzug für das fehlende Negativ-Beispiel***
  - *Beispiel*
    - *"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: 0,5P*
- *verlangte Positiv-Beispiele müssen gebracht werden – im Zweifel müssen sie extra für die Lösung der Aufgabe implementiert werden*
- *Code-Beispiel = Code in das Dokument kopieren (inkl. Syntax-Highlighting)*
- *falls Bezug auf den Code genommen wird: entsprechende Code-Teile in das Dokument kopieren (inkl Syntax-Highlighting)*

- *bei UML-Diagrammen immer die öffentlichen Methoden und Felder angegeben – private Methoden/Felder nur angeben, wenn sie zur Klärung beitragen*
- *bei UML-Diagrammen immer unaufgefordert die zusammenspielenden Klassen ergänzen, falls diese Teil der Aufgabe sind*
- *Klassennamen/Variablennamen/etc im Dokument so benennen, wie sie im Code benannt sind (z.B. im Dokument nicht anfangen, englische Klassennamen zu übersetzen)*
- *die Aufgaben sind von vorne herein bekannt und müssen wie gefordert gelöst werden – z.B. ist es keine Lösung zu schreiben, dass es das nicht im Code gibt*
    - *Beispiel 1*
        - *Aufgabe: Analyse und Begründung des Einsatzes von 2 Fake/Mock-Objekten*
        - *Antwort: Es wurden keine Fake/Mock-Objekte gebraucht.*
        - *Punkte: 0P*
    - *Beispiel 2*
        - *Aufgabe: UML, Beschreibung und Begründung des Einsatzes eines Repositories*
        - *Antwort: Die Applikation enthält kein Repository*
        - *Punkte*
            - *falls (was quasi nie vorkommt) die Fachlichkeit tatsächlich kein Repository hergibt: volle Punktzahl*
            - *falls die Fachlichkeit in irgendeiner Form ein Repository hergibt (auch wenn es nicht implementiert wurde): 0P*
    - *Beispiel 3*
        - *Aufgabe: UML von 2 implementierte unterschiedliche Entwurfsmuster aus der Vorlesung*
        - *Antwort: es wurden keine Entwurfsmuster gebraucht/implementiert*
        - *Punkt: 0P*

Kapitel 1: Einführung (4P)

Übersicht über die Applikation (1P)

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

Die Inventory Management Applikation ist ein Programm, das verwendet wird, um ein Inventar zu verwalten. Das Programm ermöglicht es dem Benutzer, Artikel hinzuzufügen, zu entfernen und zu aktualisieren, sowie das Inventar insgesamt anzuzeigen. Es gibt auch eine Benutzerverwaltungsfunktion. Die Applikation löst das Problem der manuellen Verwaltung von Inventaren und macht es einfacher, Inventare zu organisieren und zu aktualisieren.

Starten der Applikation (1P)

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

Es wir das .NET 7.0 Framework benötigt. Die Applikation kann mit dem Befehl dotnet InventoryManagement.dll gestartet werden. Alternativ kann das Repository geklont und die .sln Datei mit Beispielsweise Jetbrains Rider gestartet werden.

Technischer Überblick (2P)

[Nennung und Erläuterung der Technologien (z.B. Java, MySQL, ...), jeweils Begründung für den Einsatz der Technologien]

Die vorliegende Applikation wurde in C# entwickelt und nutzt .NET Framework für die Ausführung. Die Wahl von C# als Programmiersprache und .NET Framework als Laufzeitumgebung bietet eine hohe Flexibilität und eine gute Performance. Die Verwendung von C# ermöglicht eine objektorientierte Programmierung, was insbesondere für komplexe Anwendungen von Vorteil ist.

Für die Speicherung von Daten wird eine einfache Textdatei genutzt. Diese einfache Lösung wurde gewählt, um die Komplexität der Applikation gering zu halten und den Fokus auf andere Aspekte wie die Benutzeroberfläche und die Geschäftslogik zu legen.

Für die Lokalisierung der Anwendung wird ein einfaches selbstgeschriebenes System genutzt, welches es ermöglicht, Texte in verschiedenen Sprachen zu hinterlegen und je nach gewählter Sprache anzuzeigen.

Insgesamt wurde bei der Wahl der Technologien darauf geachtet, eine einfache und übersichtliche Architektur zu schaffen, die eine schnelle und effektive Entwicklung ermöglicht.

Kapitel 2: Clean Architecture (8P)

Was ist Clean Architecture? (1P)

[allgemeine Beschreibung der Clean Architecture in eigenen Worten]

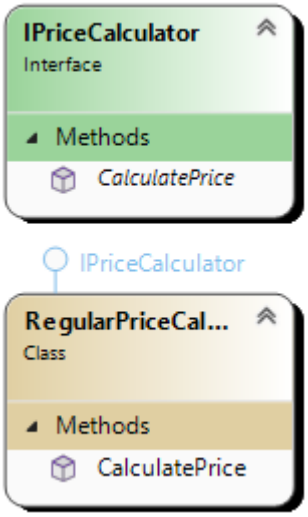
Die Clean Architecture ist ein Ansatz, mit dem versucht wird, eine Anwendung in verschiedene Teile zu unterteilen, die alle ihre Eigenen Aufgaben haben. Die Architektur besteht aus verschiedenen Schiten, die alle ihre eigenen Verantwortlichkeiten haben. Die Schichten sind hierarchisch aufgebaut. Die Abhängigkeiten sind strikt “von oben nach unten”. Die oberen Schichten sind diejenigen, die sich um die Business-Logik kümmern, während die unteren Schichten die technische Implementierung beinhalten. Ziel der Architektur ist eine klare Trennung der verschiedenen Schichten.

Analyse der Dependency Rule (3P)

[1 Klasse, die die Dependency Rule einhält und 1 Klasse, die die Dependency Rule verletzt; jeweils UML (mind. die betreffende Klasse inkl. der Klassen, die von ihr abhängen bzw. von der sie abhängt) 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]

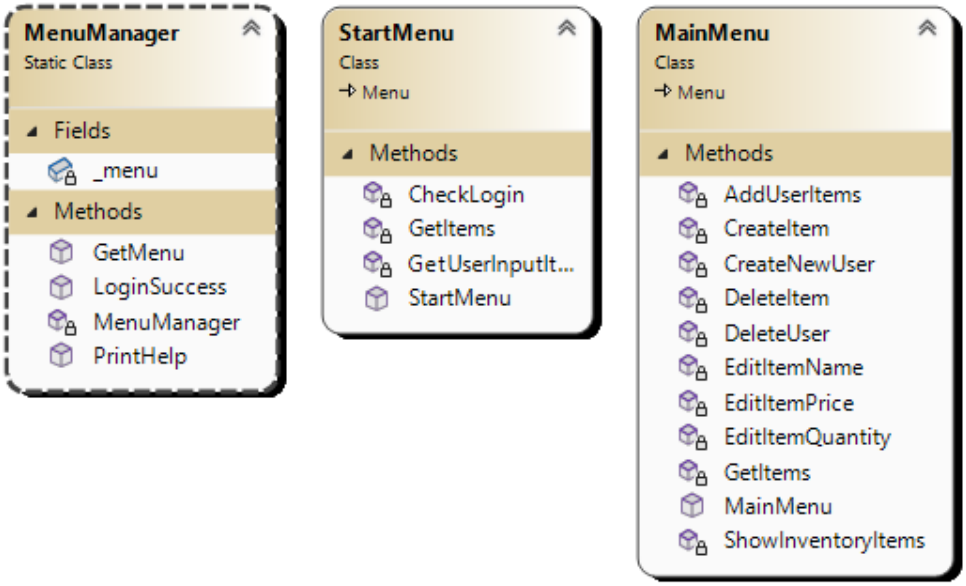
Positiv-Beispiel: Dependency Rule

`RegularPriceCalculator` hält die Dependency Rule ein. Sie implementiert das Interface `IPriceCalculator`, das in der Klasse `Item` als Abhängigkeit verwendet wird. Die Klasse `RegularPriceCalculator` enthält jedoch keine Abhängigkeiten zu anderen Klassen oder Modulen und erfüllt somit die Dependency Rule.



Negativ-Beispiel: Dependency Rule

Als Beispiel für eine Klasse, die die Dependency Rule verletzt, kann die Klasse `MenuManager` dienen. Diese Klasse verwendet die konkrete Implementierung `StartMenu` und `MainMenu` der abstrakten Klasse `Menu` und instanziiert diese direkt. Dadurch ist `MenuManager` von den konkreten Implementierungen abhängig, anstatt von der abstrakten Klasse. Dies verstößt gegen das Prinzip der Dependency Rule, da eine Klasse nicht von konkreten Implementierungen abhängig sein sollte, sondern nur von Abstraktionen.

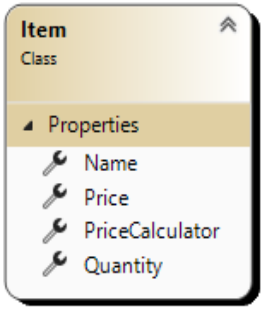


Analyse der Schichten (4P)

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

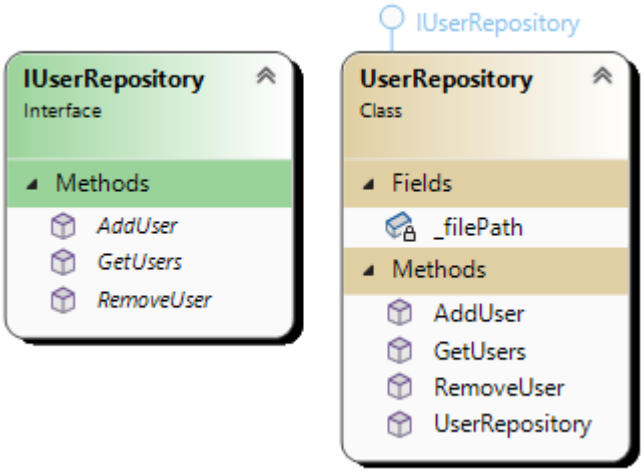
Schicht: Domain Code

Eine Klasse aus der Domain-Schicht ist beispielsweise die Klasse `Item`. Diese Klasse ist für die Modellierung von Inventar-Artikeln verantwortlich und enthält Informationen wie den Namen, die Menge und den Preis des Artikels. Die Klasse enthält keine spezifischen Implementierungsdetails, sondern definiert nur das Domänenmodell und dessen Verhalten.



Schicht: Adapters

Ein Beispiel für die Adapters-Schicht ist die Klasse `UserRepository`, die den Zugriff auf die Datenquelle für Benutzerobjekte verwaltet. Die Klasse implementiert das `IUserRepository` Interface aus der Application-Schicht, wodurch die Application-Schicht keine Details über die konkrete Implementierung der Datenquelle kennen muss. Die Klasse ist in der Adapters-Schicht, da sie die Verbindung zwischen der Anwendungsschicht und der Datenquelle herstellt.



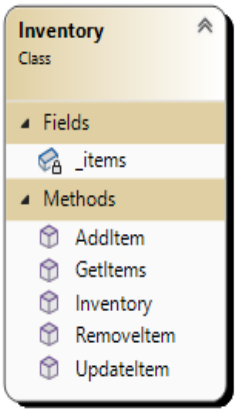
Kapitel 3: SOLID (8P)

Analyse SRP (3P)

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

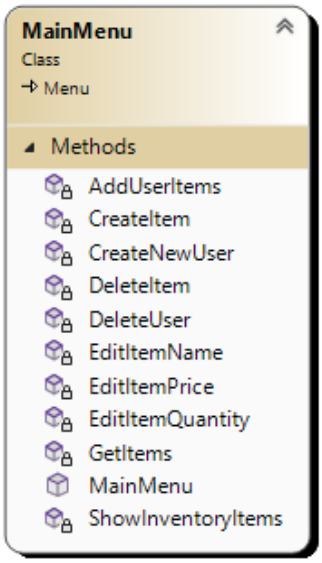
Positiv-Beispiel

Die Klasse `Inventory` ist verantwortlich für die Verwaltung und Speicherung von Gegenständen im Inventar und hat somit nur eine klare Verantwortlichkeit.



Negativ-Beispiel

Ein negatives Beispiel für das Single Responsibility Principle ist die Klasse `MainMenu`. Diese Klasse hat die Verantwortlichkeit, die Benutzeroberfläche für die Hauptmenü-Optionen anzuzeigen und zu verwalten, aber sie hat auch die Verantwortlichkeit, auf Benutzereingaben zu reagieren und Logik auszuführen, um auf diese Eingaben zu reagieren. Eine mögliche Lösung wäre, die Aufgaben der Klasse aufzuteilen, indem eine separate Klasse erstellt wird, um auf Benutzereingaben zu reagieren und die Logik zu verwalten. Dadurch würde die Verantwortlichkeit der `MainMenu`-Klasse auf die Verwaltung der Benutzeroberfläche beschränkt bleiben.

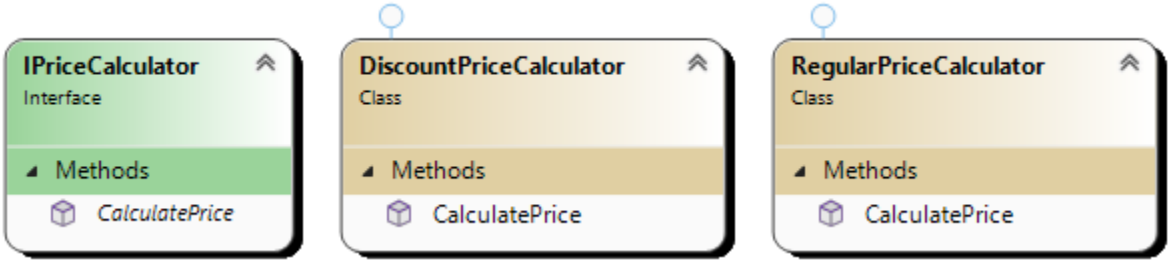


Analyse OCP (3P)

[jeweils eine Klasse als positives und negatives Beispiel für OCP; jeweils UML 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)?]

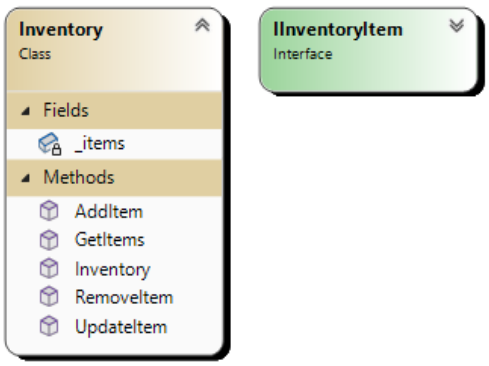
Positiv-Beispiel

Ein positives Beispiel für das Open/Closed Principle ist die `IPriceCalculator`-Schnittstelle und ihre Implementierungen `RegularPriceCalculator` und `DiscountPriceCalculator`. Diese Klassen erfüllen das OCP, weil sie eine offene Erweiterungsmöglichkeit bieten, um zukünftige Preisberechnungsalgorithmen hinzuzufügen, ohne die existierende Funktionalität zu verändern. Man kann einfach eine neue Klasse erstellen, die das `IPriceCalculator`-Interface implementiert, und sie als Abhängigkeit in der `Item`-Klasse oder in anderen Klassen verwenden, die eine Preisberechnung benötigen.



Negativ-Beispiel

Ein negatives Beispiel für das OCP ist die `Inventory`-Klasse. Die Klasse verletzt das OCP, weil sie nicht für Erweiterungen offen ist. Wenn zum Beispiel ein neues Attribut hinzugefügt werden soll, z.B. das Gewicht oder die Farbe eines Artikels, müsste die `Inventory`-Klasse verändert werden. Eine bessere Lösung wäre, eine abstrakte Klasse oder ein Interface wie `IInventoryItem` zu definieren, das alle gemeinsamen Eigenschaften von Artikeln definiert, die in das Inventar aufgenommen werden können. Die `Inventory`-Klasse sollte dann eine Liste von `IInventoryItem`-Objekten speichern und mit ihnen arbeiten. Auf diese Weise können neue Arten von Artikeln einfach durch die Erstellung von neuen Klassen, die `IInventoryItem` implementieren, hinzugefügt werden, ohne dass Änderungen an der `Inventory`-Klasse vorgenommen werden müssen.



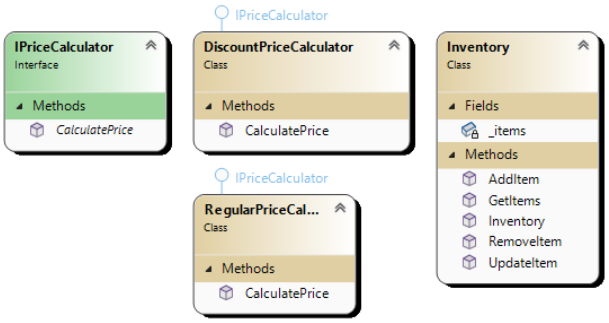
Analyse [LSP/ISP/DIP] (2P)

[jeweils eine Klasse als positives und negatives Beispiel für entweder LSP oder ISP oder DIP; jeweils UML und Begründung, warum hier das Prinzip erfüllt/nicht erfüllt wird; beim Negativ-Beispiel UML einer möglichen Lösung hinzufügen]

[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]

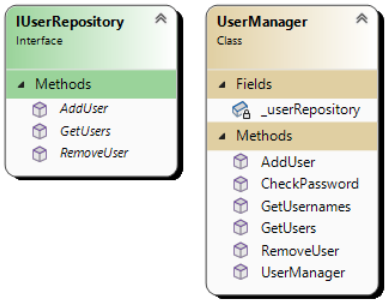
Positiv-Beispiel DIP

Die Klasse `DiscountPriceCalculator` in der `Inventory` Schicht ist ein positives Beispiel für das DIP, da sie von der Schnittstelle `IPriceCalculator` abhängt und nicht von einer konkreten Implementierung. Dadurch ist sie unabhängig von konkreten Klassen wie beispielsweise `RegularPriceCalculator`. Durch die Verwendung von Interfaces als Abhängigkeiten wird das Prinzip der Abstraktion erfüllt und es wird eine höhere Flexibilität und Austauschbarkeit von Komponenten erreicht.



Negativ-Beispiel DIP

Die Klasse `UserManager` in der `Users` Schicht ist ein negatives Beispiel für das DIP, da sie von einer konkreten Implementierung der Schnittstelle `IUserRepository` abhängt und nicht von der Schnittstelle selbst. Das bedeutet, dass Änderungen an der Implementierung zu Änderungen an der Klasse `UserManager` führen können, was zu einer erhöhten Kopplung und einer geringeren Flexibilität führt. Eine Lösung wäre es, eine Abhängigkeit zur Schnittstelle `IUserRepository` anstatt zur konkreten Implementierung zu verwenden. Dadurch würde das Prinzip der Abstraktion eingehalten werden und es würde eine höhere Flexibilität und Austauschbarkeit von Komponenten erreicht werden.

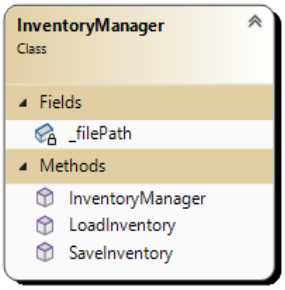


Kapitel 4: Weitere Prinzipien (8P)

Analyse GRASP: Geringe Kopplung (3P)

*[eine **bis jetzt noch nicht behandelte** Klasse als positives Beispiel geringer Kopplung; UML mit zusammenspielenden Klassen, Aufgabenbeschreibung der Klasse und Begründung, warum hier eine geringe Kopplung vorliegt]*

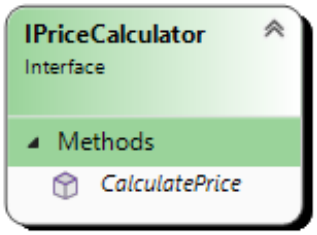
Ein positives Beispiel für geringe Kopplung im GRASP-Prinzip ist die Klasse `InventoryManager`. Die Aufgabe dieser Klasse ist es, das Inventar zu verwalten, indem es in eine Datei gespeichert und geladen wird. Die Klasse hat eine sehr klare Verantwortlichkeit und hat nur eine Abhängigkeit von der `Inventor` Klasse. Es hat keine Abhängigkeiten von anderen Klassen im System, und es ist leicht austauschbar, wenn sich die Implementierung von `Inventor` ändert. Daher ist die Kopplung dieser Klasse sehr gering, was dazu beiträgt, dass das System leichter zu warten und zu erweitern ist.



Analyse GRASP: [Polymorphismus/Pure Fabrication] (3P)

[eine Klasse als positives Beispiel entweder von Polymorphismus oder von Pure Fabrication; UML Diagramm und Begründung, warum es hier zum Einsatz kommt]

Die `IPriceCalculator`-Schnittstelle ist ein Beispiel für Polymorphismus von GRASP in Ihrem Code. Die `IPriceCalculator`-Schnittstelle ermöglicht, dass verschiedene Preisberechnungsalgorithmen implementiert werden können, um den Preis eines Artikels auf unterschiedliche Weise zu berechnen, während die Schnittstelle selbst unverändert bleibt. Durch die Verwendung von Polymorphismus wird die Flexibilität des Systems erhöht, da verschiedene Implementierungen von `IPriceCalculator` hinzugefügt oder ausgetauscht werden können, ohne dass Änderungen an anderen Teilen des Systems vorgenommen werden müssen. So kann beispielsweise ein Rabattrechner hinzugefügt werden, der einen Rabatt auf den Artikel gewährt, oder ein Preiskalkulator für den Verkauf von Artikeln in einem bestimmten Land, der die Währungsumrechnung durchführt, kann hinzugefügt werden, ohne dass der restliche Code des Systems geändert werden muss.



DRY (2P)

[ein Commit angeben, bei dem duplizierter Code/duplizierte Logik aufgelöst wurde; Code-Beispiele (vorher/nachher) einfügen; begründen und Auswirkung beschreiben – ggf. UML zum Verständnis ergänzen]

https://github.com/Xantorhd/InventoryManagement/commit/4ccb2ad87fd3c1bb33a5579d01d309c2eedc2422

Der duplizierte Code wurde aufgelöst, da er unnötigerweise doppelt vorlag. Dies war durch eine vorher eingeführte Featureerweiterung aus Versehen entstanden. Durch die Auflösung des duplizierten Codes, wird der Code besser lesbar und übersichtlicher.

Kapitel 5: Unit Tests (8P)

10 Unit Tests (2P)

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

Unit Test	Beschreibung
LocalizationManagerTests# TestAddLocalization	Dieser Test prüft, ob eine neue Lokalisierung für einen bestimmten Schlüssel hinzugefügt werden kann und ob die hinzugefügte Lokalisierung korrekt abgerufen wird.
LocalizationManagerTests# TestGetLanguageByCode	Dieser Test prüft, ob eine Sprache anhand ihres Codes korrekt abgerufen wird.
LocalizationManagerTests# TestGetLanguageByName	Dieser Test prüft, ob eine Sprache anhand ihres Namens korrekt abgerufen wird.
LocalizationManagerTests# TestSetCurrentLanguage	Dieser Test prüft, ob die aktuelle Sprache korrekt auf eine neue Sprache gesetzt wird.
LocalizationManagerTests# TestGetText	Dieser Test prüft, ob eine Lokalisierung für einen bestimmten Schlüssel korrekt abgerufen wird.
LocalizationManagerTests# TestGetText_DefaultLanguage	Dieser Test prüft, ob die Standardlokalisierung korrekt zurückgegeben wird, wenn keine Sprache ausgewählt ist.
LocalizationManagerTests# TestGetAvailableLanguages	Dieser Test prüft, ob eine Liste aller verfügbaren Sprachen korrekt abgerufen wird.
LocalizationManagerTests# TestGetCurrentLanguage	Dieser Test prüft, ob die aktuelle Sprache korrekt abgerufen wird.
LocalizationManagerTests# TestGetText_InvalidKey	Dieser Test prüft, ob die Standardlokalisierung korrekt zurückgegeben wird, wenn ein ungültiger Schlüssel angegeben wird.
LocalizationManagerTests# TestGetText_InvalidLanguage	Dieser Test prüft, ob die Standardlokalisierung korrekt zurückgegeben wird, wenn eine nicht unterstützte Sprache ausgewählt wird.

ATRIP: Automatic (1P)

[Begründung/Erläuterung, wie ‘Automatic’ realisiert wurde]

Automatic wurde über Github Workflows umgesetzt. Bei jedem Push auf den Master Branch wird automatisch das Projekt gebaut und getestet. Dieser Weg wurde gewählt, damit man 1. nicht selbst daran denken muss und man sich 2. unabhängig vom Endgerät darauf verlassen kann, dass die Tests durchgeführt werden. Bei Fehlschlag wird automatisch eine Mail verstandt.

ATRIP: Thorough (1P)

[Code Coverage im Projekt analysieren und begründen]

Die Codecoverage ist deckt hauptsächlich die Localization Klassen ab. Dies ist der Tatsache geschuldet, dass lediglich 10 Unit Tests gefordert waren.

ATRIP: Professional (1P)

[1 positves Beispiel zu ‘Professional’; Code-Beispiel, Analyse und Begründung, was professionell ist]

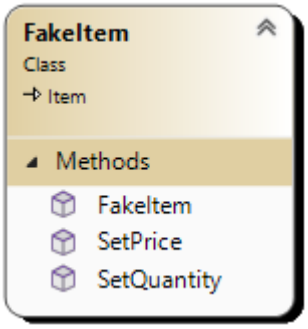
Das automatisierte Testen des Projekts durch Github Workflows erfüllt das Professional Kriterium von ATRIP, da es die Qualität und die Zuverlässigkeit des Codes sicherstellt. Es ermöglicht es, Probleme frühzeitig zu erkennen und zu beheben, bevor der Code in Produktion geht. Das automatisierte Testen durch Github Workflows bietet außerdem den Vorteil der Wiederholbarkeit und Konsistenz der Tests, da sie bei jeder Änderung des Codes automatisch ausgeführt werden. Dies gewährleistet, dass der Code jederzeit den erwarteten Spezifikationen entspricht. Darüber hinaus ermöglicht die Integration von Tests in den Entwicklungsprozess eine schnellere Fehlerbehebung und somit eine höhere Effizienz und Produktivität bei der Entwicklung. Insgesamt trägt das automatisierte Testen durch Github Workflows somit zu einer professionellen und zuverlässigen Softwareentwicklung bei und erfüllt damit das Professional Kriterium von ATRIP.

Fakes und Mocks (3P)

[Analyse und Begründung des Einsatzes von 2 Fake/Mock-Objekten (die Fake/Mocks sind ohne Dritthersteller-Bibliothek/Framework zu implementieren); zusätzlich jeweils UML Diagramm mit Beziehungen zwischen Mock, zu mockender Klasse und Aufrufer des Mocks]

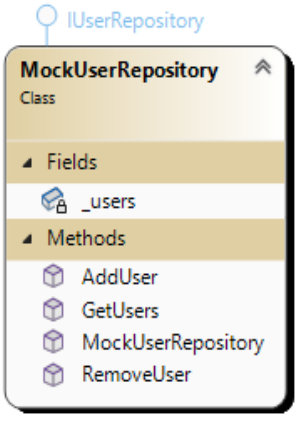
FakeItem

Das `FakeItem` wurde erstellt, um eine einfache Möglichkeit zu schaffen, die Klasse `Item` für Unit Tests zu verwenden. Die Klasse ermöglicht es, gezielt Testdaten zu erstellen und diese für den Test zu verwenden, ohne dass tatsächliche Daten verändert werden müssen. Der Vorteil dabei ist, dass der Test unabhängig von anderen Teilen der Anwendung ausgeführt werden kann, was zu einer besseren Testbarkeit und Robustheit des Codes führt. In diesem speziellen Test wird das FakeItem verwendet, um sicherzustellen, dass das hinzugefügte Item korrekt in das Inventory gespeichert wird.



Mock Object

Das Mock-Objekt wird eingesetzt, um das Verhalten des `UserRepository`-Objekts im Test zu simulieren. In diesem speziellen Beispiel wird eine Testimplementierung von `IuserRepository` erstellt, die in einer Liste von Benutzern im Arbeitsspeicher speichert, anstatt eine Datei zu verwenden. Dadurch kann der UserManager-Test unabhängig von der Implementierung des tatsächlichen `UserRepository`-Objekts ausgeführt werden. Das Mock-Objekt gibt immer die erwarteten Ergebnisse zurück und stellt somit eine stabile und zuverlässige Testumgebung bereit. Durch den Einsatz von Mock-Objekten können Tests schneller ausgeführt werden, da keine Leseoperation erforderlich ist.



Kapitel 6: Domain Driven Design (8P)

Ubiquitous Language (2P)

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

Bezeichung	Bedeutung	Begründung
Item: Ein Objekt, das ein Produkt im Inventar darstellt.	Eine Entität, die Name, Menge und Preis des Produkts enthält.	"Item" ist ein wichtiges Konzept in der Domäne des Inventarmanagements und wird daher als gemeinsame Sprache von allen beteiligten Personen verwendet.
IPriceCalculator: Ein Interface, das die Berechnung des Preises für ein Item ermöglicht.	Ein Preisberechnungsmechanismus, der von verschiedenen Implementierungen abhängen kann.	"IPriceCalculator" ist ein wichtiger Begriff in der Domäne des Inventarmanagements, da es verschiedene Möglichkeiten gibt, den Preis eines Artikels zu berechnen.
User: Ein Objekt, das einen Benutzer im System darstellt.	Eine Entität, die den Benutzernamen und das Passwort enthält, um sich im System anzumelden.	"User" ist ein wichtiger Begriff in der Domäne des Benutzermanagements und wird daher als gemeinsame Sprache von allen beteiligten Personen verwendet.
Localization: Eine Klasse, die die Lokalisierung von Texten in verschiedenen Sprachen ermöglicht.	Eine Möglichkeit, Texte in verschiedenen Sprachen zur Verfügung zu stellen.	"Localization" ist ein wichtiger Begriff in der Domäne der internationalen Softwareentwicklung und wird daher als gemeinsame Sprache von Entwicklern, Testern und anderen beteiligten Personen verwendet.

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 – NICHT, warum es nicht implementiert wurde]

Das `UserRepository` entspricht dem Domain Driven Design, da es eine wichtige Rolle bei der Datenpersistenz und Datenzugriff für die `User`-Entität spielt. Das `UserRepository` dient als Gateway zur Datenbank und sorgt für die Speicherung und Wiederherstellung von `User`-Entitäten. Es isoliert auch das Domain-Modell von der Datenzugriffsinfrastruktur und ermöglicht es dem Domain-Modell, unabhängig von der verwendeten Datenbank oder der Art des Datenzugriffs zu bleiben. Durch die Verwendung eines Repositories kann die Domain-Logik einfach getestet und gewartet werden, da es eine klare Trennung zwischen dem Domain-Code und dem Datenzugriffs-Code gibt.

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– NICHT, warum es nicht implementiert wurde]

Ein mögliches Beispiel für ein Aggregate im Sinne von DDD aus dem gegebenen Code ist die Klasse `Item`. Ein Aggregate ist eine Gruppe von Objekten, die gemeinsam eine konsistente Einheit bilden und als Transaktionsgrenze dienen. In diesem Fall könnten mehrere `Item`-Objekte zusammen als Aggregate fungieren, da sie gemeinsam die Bestandsinformationen eines bestimmten Geschäftsmodells bilden und Änderungen an ihnen in einer einzigen Transaktion durchgeführt werden müssen, um die Konsistenz der Daten sicherzustellen.
Ein Aggregate ist eine wichtige Konstruktion, da es die Semantik und Bedeutung einer Gruppe von Objekten innerhalb des Kontexts der Domäne hervorhebt und es ermöglicht, Änderungen und Validierungen innerhalb dieses Bereichs durchzuführen, ohne den Rest der Anwendung zu beeinflussen.

Entities (1,5P)

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

Die Klasse `User` repräsentiert ein Objekt mit Identität und Zustand, das im Kontext des Inventarmanagements relevant ist. Es hat eine eindeutige Identität, die durch den `Username` bestimmt wird. Da es im System wichtig ist, dass Benutzer verlässlich identifiziert werden können, ist es sinnvoll, die `User`-Klasse als Entity zu modellieren. Zusätzlich wird die `User`-Klasse von anderen Klassen im System referenziert und verwendet, wie beispielsweise vom `UserManager`, der Funktionen zum Hinzufügen und Entfernen von Benutzern bereitstellt. Diese Referenzierung und Verwendung unterstützt den Status als Entity.

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– NICHT, warum es nicht implementiert wurde]

Ein Beispiel für ein Value Object ist die `Language` Klasse in der `Localization`-Schicht. Ein Sprachobjekt wird hier als Wertobjekt modelliert, da es keine Identitätskomponente besitzt, sondern nur durch seine Eigenschaften (Code und Name) definiert wird. Das Sprachobjekt wird oft als Parameter an Methoden innerhalb der `Localization`-Schicht weitergegeben, beispielsweise um eine lokale Übersetzung zu finden. Durch die Verwendung von Value Objects wird sichergestellt, dass sich keine Verwechslungen oder unbeabsichtigte Effekte ergeben, wenn Sprachobjekte verwendet werden.

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)]

[Duplicated Code]

https://github.com/Xantorhd/InventoryManagement/commit/4ccb2ad87fd3c1bb33a5579d01d309c2eedc2422

Der duplizierte Code wurde entfernt. Dazu wurde einer der Codes entfernt und durch einen Methodenaufruf ersetzt.

[Switch Statement]

https://github.com/Xantorhd/InventoryManagement/blob/master/InventoryManagement/ConsoleMenu/Menu.cs#L131

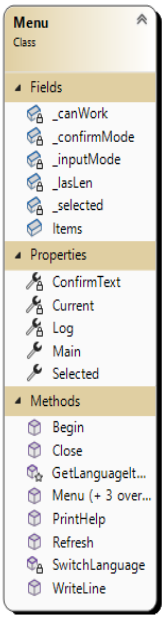
Ein möglicher Lösungsweg wäre das verwenden diverser If-Anweisungen. Dazu müsste der gedrückte Key in eine Variable gespeichert werden und abgeglichen werden. Die jeweilige Anweisung innerhalb des If-Blocks entspräche etwa derer, wie sie jetzt schon in den Switch Statements zu finden sind.

2 Refactorings (6P)

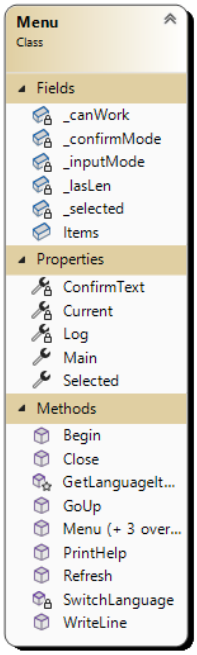
[2 unterschiedliche Refactorings aus der Vorlesung anwenden, begründen, sowie UML vorher/nachher liefern; jeweils auf die Commits verweisen]

[Extract Method]

Die Methode wurde ausgelagert, da der Code sonst doppelt vorhanden gewesen wäre. Durch die Änderung verbessert sich die Lesbarkeit und Wiederverwendbarkeit.

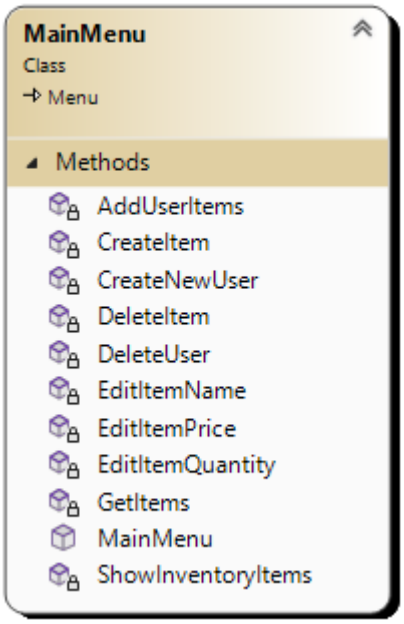


https://github.com/Xantorhd/InventoryManagement/commit/4ccb2ad87fd3c1bb33a5579d01d309c2eedc2422

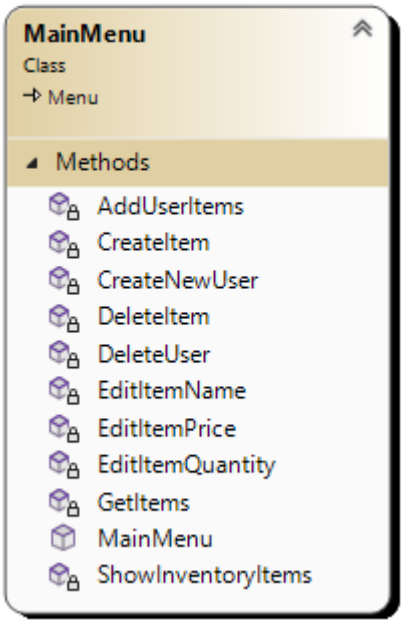


[Rename Method]

In der Methode werden viel eher die Nutzeritems hinzugefügt, als dass welche gezeichnet werden. Die Umbenennung erleichtert damit die Lesbarkeit und das Verständnis des Codes.



https://github.com/Xantorhd/InventoryManagement/commit/023961521e19c6a683c5fe1e75c9541259a631a1

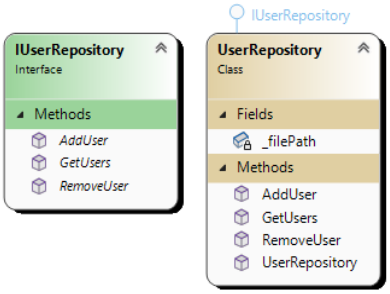


Kapitel 8: Entwurfsmuster (8P)

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

Entwurfsmuster: Adapter (4P)

Ein Beispiel ist die Klasse `UserRepository`, die als Adapter zwischen der Anwendungslogik und der Datenquelle dient, um eine einheitliche Schnittstelle bereitzustellen.
Das Adapter-Entwurfsmuster wird eingesetzt, um eine Schnittstelle eines Objekts an eine andere Schnittstelle anzupassen, ohne dass die zugrunde liegende Implementierung des Adapters geändert werden muss. Dies ermöglicht es, bestehende Klassen in neuen Kontexten zu verwenden, ohne dass Änderungen an der ursprünglichen Implementierung erforderlich sind.
Im Fall des `UserRepository` Adapters ermöglicht es, dass die Anwendungslogik unabhängig von der konkreten Implementierung der Datenquelle bleibt. Wenn später eine andere Datenquelle verwenden werden soll, kann einfach einen neuer Adapter geschrieben werden, der das `IUserRepository`-Interface implementiert, anstatt die Anwendungslogik ändern zu müssen.



Entwurfsmuster: Builder (4P)

Das Builder Entwurfsmuster wird verwendet, um komplexe Objekte schrittweise zu erstellen, indem verschiedene Teilschritte nacheinander ausgeführt werden. Der Builder kapselt den Erstellungsprozess und bietet eine Schnittstelle, über die der Nutzer die einzelnen Schritte steuern und beeinflussen kann. Dadurch wird der Erstellungsprozess flexibler und es wird vermieden, dass das Erstellen eines Objekts zu komplex wird und Code-Duplikation entsteht. Im vorliegenden Beispiel wird der Builder genutzt, um ein neues Item schrittweise über die Konsoleneingabe zu erstellen. Der Nutzer wird aufgefordert, nacheinander den Namen, die Menge und den Preis einzugeben.