|  |
| --- |
| Dokumentation zum Praxisprojekt im Kurs Advanced Software Engineering  an der  Dualen Hochschule Baden-Württemberg Karlsruhe  von  Burak Özkan  Marius Engelmeier  Abgabedatum:  31.05.2023 |

Inhaltsverzeichnis

Inhaltsverzeichnis 2

1. Projektbeschreibung 3

2. Programmstruktur 3

3. Clean Architecture 4

4. Programming Principles 7

4.1 SOLID-Prinzipien 7

4.1.1 Single Responsibility Principle (SRP) 7

4.1.2 Open-Closed Principle (OCP) 8

4.1.3 Liskov Substitution Principle (LSP) 8

4.1.4 Interface Segregation Principle (ISP) 11

4.1.5 Dependency Inversion Principle (DIP) 11

4.2 GRASP-Prinzipien 12

4.2.1 Geringe Kopplung 12

4.3 Hohe Kohäsion 13

4.4 DRY-Prinzip 15

5. Entwurfsmuster 17

5.1 Beobachter 17

6. Unit Tests 17

7. Refactoring 17

# Github Referenz

Das Projekt kann über den nachfolgenden Link in Github geöffnet werden:

<https://github.com/MariusE1234/S.N.A.C.K.git>

# Projektbeschreibung

S.N.A.C.K steht für das **s**tilvolle **n**eue **a**ttraktive **c**oole **K**nabbersystem. Es handelt sich dabei um einen virtuellen Verkaufsautomaten, der eine Vielzahl von Snacks und Getränken zur Verfügung stellt. Der Automat funktioniert wie ein herkömmlicher Verkaufsautomat: Münzen werden eingeworfen und das gewünschte Produkt wird über einen Button ausgewählt. Zudem kann der Automat angepasst werden, um Preise oder das Angebot festzulegen.

Das Ziel dieses Projekts ist es, unerfahrenen Automatenbenutzern durch den virtuellen S.N.A.C.K. bei der Vorbereitung auf die Nutzung von Automaten in der realen Welt zu helfen. Darüber hinaus können Automatenbesitzer lernen, wie sie ihre Geräte einrichten und anpassen können.

Die vollständige Implementierung des Programms erfolgte in Python. Diese Programmiersprache wurde gewählt, weil wir beide bereits Erfahrung mit der Sprache haben und uns damit am sichersten fühlen.

# Programmstruktur

Das Programm ist in drei Schichten unterteilt, die jeweils unterschiedliche Aspekte und Funktionen des Verkaufsautomaten behandeln:

* Layer 1: Domain-Entities
* Layer 2: Core-Logik und grundlegende Implementierung
* Layer 3: GUI und Datenzugriff

**Layer 1: Domain-Entities**

In diesem Layer sind die grundlegenden Entitäten des Systems definiert:

* entities.py: Enthält die Entitäten Produkt, Coin und Transaction, welche sich normalerweise nicht ändern sollten.

**Layer 2: Core-Logik und grundlegende Implementierung**

In diesem Layer sind die grundlegenden Funktionen und Implementierungen enthalten, die für den Betrieb des Verkaufsautomaten erforderlich sind:

* core\_functions.py: Enthält die grundlegenden Funktionen, die der Automat benötigt.
* interfaces.py: Enthält die Interfaces für bestimmte Klassen.
* validator.py: Zuständig für die Validierung von Eingaben.
* vending\_machine.py: Enthält die grundlegende Implementierung des Verkaufsautomaten.

**Layer 3: GUI und Datenzugriff**

In diesem Layer sind die Komponenten für die grafische Benutzeroberfläche und den Datenzugriff auf die Datenbank enthalten:

* dialogs: Enthält verschiedene Dialogfenster für die Benutzeroberfläche, wie zum Beispiel coins.py, config.py, feedback.py, info.py, pin.py, product.py und statistic.py.
* controllers.py: Enthält Controller, die zwischen der GUI und den core\_functions vermitteln.
* data\_access.py: Enthält die Funktionen und SQL-Statements für den Datenzugriff der Datenbank.
* database.py: Enthält die Details zur Verbindung mit der Datenbank.
* factories.py: Enthält factory Klassen.
* ui\_updater.py: Enthält die Funktionen zum Aktualisieren der grafischen Oberfläche.
* ui.py: Enthält die Implementierung der grafischen Oberfläche.

Das Hauptskript main.py erzeugt die Objektinstanzen und startet das Programm.

# Clean Architecture

Das Programm folgt dem Prinzip der Clean Architecture, wie an den folgenden Merkmalen erkennbar ist:

Trennung von Zuständigkeiten

Die Trennung von Zuständigkeiten ist ein zentrales Prinzip der Clean Architecture und trägt dazu bei, die Komplexität des Programms zu reduzieren und die Wartbarkeit zu erhöhen. Im Verkaufsautomaten-Programm wird dies durch die klar definierten Schichten erreicht:

* Layer 1 (Domain-Entities): Beinhaltet die grundlegenden Entitäten des Systems, die von den anderen Schichten verwendet werden, um auf die Geschäftslogik aufzubauen.
* Layer 2 (Core-Logik und grundlegende Implementierung): Beinhaltet die grundlegenden Funktionen und Implementierungen, die für den Betrieb des Verkaufsautomaten erforderlich sind, unabhängig von der Benutzeroberfläche oder dem Datenzugriff.
* Layer 3 (GUI und Datenzugriff): Beinhaltet die Komponenten für die grafische Benutzeroberfläche und den Datenzugriff auf die Datenbank, die von der Core-Logik getrennt sind.

Durch die Trennung von Zuständigkeiten können die Entwickler sich auf die spezifischen Aspekte einer Schicht konzentrieren, ohne sich um die Details der anderen Schichten kümmern zu müssen. Dies erleichtert das Verständnis des Programms und ermöglicht es, Änderungen vorzunehmen, ohne unbeabsichtigte Nebeneffekte zu erzeugen.

Verwendung von Abstraktionen

Das Programm verwendet Interfaces und Abstraktionen, um die Kopplung zwischen den Schichten zu verringern und die Erweiterbarkeit und Testbarkeit des Systems zu erhöhen. Beispielsweise definiert das Programm Interfaces für bestimmte Klassen in interfaces.py. Diese Interfaces ermöglichen es, die Implementierungsdetails zu verbergen und die Abhängigkeiten zwischen den Schichten zu minimieren.

Die Verwendung von Factory-Klassen in factories.py ist ein weiteres Beispiel für Abstraktionen im Programm. Die Factory-Klassen ermöglichen es, Objekte zu erzeugen, ohne die konkrete Implementierung der Klassen zu kennen. Dies macht es einfacher, alternative Implementierungen einzuführen oder das Verhalten des Programms während der Laufzeit zu ändern.

Kontrolle von Abhängigkeiten

Ein weiteres wichtiges Prinzip der Clean Architecture ist die Kontrolle von Abhängigkeiten, bei der die Abhängigkeiten innerhalb des Programms so organisiert sind, dass sie von den äußeren Schichten zu den inneren Schichten fließen. Im Verkaufsautomaten-Programm wird dies erreicht, indem die Core-Logik von den Domain-Entities abhängt und die GUI und der Datenzugriff von der Core-Logik abhängen.

Durch die Kontrolle der Abhängigkeiten wird sichergestellt, dass die inneren Schichten (wie die Core-Logik und die Domain-Entities) unabhängig von den äußeren Schichten (wie der GUI und dem Datenzugriff) sind. Dies ermöglicht es, die inneren Schichten leichter zu testen und die äußeren Schichten auszutauschen oder zu ergänzen, ohne die inneren Schichten zu beeinflussen.

Die controllers.py-Datei dient als Vermittler zwischen der GUI und der Core-Logik, indem sie die Anfragen von der Benutzeroberfläche entgegennimmt und die entsprechenden Funktionen in der Core-Logik aufruft. Auf diese Weise bleiben die Core-Logik und die GUI voneinander entkoppelt und können unabhängig voneinander entwickelt und getestet werden.

Die Datenzugriffsschicht ist ebenfalls von der Core-Logik getrennt und wird durch die data\_access.py- und database.py-Dateien implementiert. Diese Trennung ermöglicht es, verschiedene Datenbankimplementierungen auszuprobieren oder sogar auf eine vollständig andere Datenquelle umzusteigen, ohne dass die Core-Logik oder die GUI davon betroffen sind.

Testbarkeit

Die Clean Architecture fördert die Testbarkeit, indem sie die Trennung von Zuständigkeiten und die Verwendung von Abstraktionen ermöglicht. Im Verkaufsautomaten-Programm wird dies erreicht, indem die verschiedenen Schichten und Komponenten des Programms entkoppelt und modular aufgebaut sind. Dies ermöglicht es, Unit-Tests für die einzelnen Komponenten zu schreiben, ohne dass andere Teile des Programms gestört werden.

Beispielsweise können die Core-Logik-Funktionen in core\_functions.py und die Validierungsfunktionen in validator.py unabhängig von der GUI und der Datenzugriffsschicht getestet werden. Dies erleichtert das Schreiben von Tests und die Sicherstellung der Qualität des Programms.

Erweiterbarkeit

Die Clean Architecture fördert auch die Erweiterbarkeit des Programms, indem sie die Möglichkeit bietet, neue Funktionen und Komponenten hinzuzufügen, ohne die bestehenden Komponenten zu beeinträchtigen. Durch die klare Trennung von Zuständigkeiten und die Verwendung von Abstraktionen können neue Funktionen oder Verbesserungen leichter in das Programm integriert werden.

Zum Beispiel könnten neue Dialogfenster für zusätzliche Funktionen in der GUI hinzugefügt werden, ohne die Core-Logik oder die Datenzugriffsschicht zu beeinflussen. Ebenso könnte die Core-Logik erweitert werden, um neue Funktionen zu unterstützen, ohne die GUI oder die Datenbankanbindung zu ändern.

Insgesamt zeigt das Verkaufsautomaten-Programm eine klare Anwendung der Clean Architecture-Prinzipien durch die Trennung von Zuständigkeiten, die Verwendung von Abstraktionen und die Kontrolle von Abhängigkeiten. Dies führt zu einem besser verständlichen, testbaren und erweiterbaren Programm, das für zukünftige Anforderungen gerüstet ist.

# Programming Principles

## SOLID-Prinzipien

### Single Responsibility Principle (SRP)

Im Programmcode gibt es mehrere Dateien, die das Single Responsibility Prinzip (SRP) gut demonstrieren. Hier sind zwei Beispiele:

* validator.py: Diese Datei ist ausschließlich für die Validierung von Eingaben zuständig. Durch die Konzentration auf eine einzige Verantwortung – die Validierung von Eingaben – bleibt der Code in dieser Datei fokussiert und leicht verständlich. Wenn es notwendig wird, die Validierungslogik zu ändern oder zu erweitern, betrifft dies nur diese Datei, ohne Auswirkungen auf andere Teile des Programms.
* ui\_updater.py: Diese Datei enthält Funktionen zum Aktualisieren der grafischen Oberfläche. Indem sie sich nur auf diese spezifische Aufgabe konzentriert, erfüllt die Datei das Single Responsibility Prinzip. Änderungen oder Erweiterungen an der Aktualisierungslogik der grafischen Oberfläche betreffen nur diese Datei und beeinträchtigen nicht die anderen Komponenten des Programms.

In beiden Fällen hält sich der Code an das SRP, indem er sich auf eine einzige Aufgabe oder Verantwortung konzentriert, was zu einem besser strukturierten, wartbaren und verständlichen Programm führt.

### Open-Closed Principle (OCP)

Das Open-Closed Principle (OCP) besagt, dass Software-Entitäten (Klassen, Module, Funktionen usw.) offen für Erweiterungen, aber geschlossen für Modifikationen sein sollten. Im Verkaufsautomaten-Programm gibt es mehrere Dateien, die das OCP gut demonstrieren. Hier sind zwei Beispiele:

* interfaces.py: Diese Datei enthält die Interfaces für bestimmte Klassen. Interfaces sind ein gutes Beispiel für das Open-Closed Principle, da sie es ermöglichen, die Implementierungsdetails einer Klasse zu ändern oder neue Implementierungen hinzuzufügen, ohne das Interface selbst zu modifizieren. Die bestehenden Klassen, die von diesen Interfaces abhängen, müssen nicht geändert werden, wenn eine neue Implementierung hinzugefügt oder eine bestehende Implementierung geändert wird.
* factories.py: Die Factory-Klassen in dieser Datei erlauben die Erstellung von Objekten, ohne die konkrete Implementierung der Klassen zu kennen. Dies bedeutet, dass die Factory-Klassen offen für Erweiterungen sind, da neue Klassen hinzugefügt oder bestehende Klassen geändert werden können, ohne die Factory-Klassen selbst zu ändern. Die Factory-Klassen sind somit geschlossen für Modifikationen, aber offen für Erweiterungen, und erfüllen so das Open-Closed Principle.

In beiden Fällen ermöglichen die Dateien, dass neue Funktionen oder Implementierungen hinzugefügt werden können, ohne die bestehenden Strukturen und Abhängigkeiten zu verändern, was zu einem flexibleren und besser wartbaren Programm führt.

### Liskov Substitution Principle (LSP)

Im Programmcode wird das Liskov Substitution Principle (LSP) bei der abstrakten Klasse *ProductDialog* und den beiden Unterklassen *AddProductDialog* und *EditProductDialog* angewendet. Hier ist das LSP im Einsatz, da *AddProductDialog* und *EditProductDialog* die Basisklasse *ProductDialog* erweitern und die abstrakten Methoden überschreiben, ohne das korrekte Funktionieren des Programms zu beeinträchtigen.

Die abstrakte Basisklasse *ProductDialog* definiert die Methoden *get\_dialog\_title, get\_dialog\_icon, get\_submit\_button\_text, save\_product* und *validate\_input* als abstrakte Methoden:

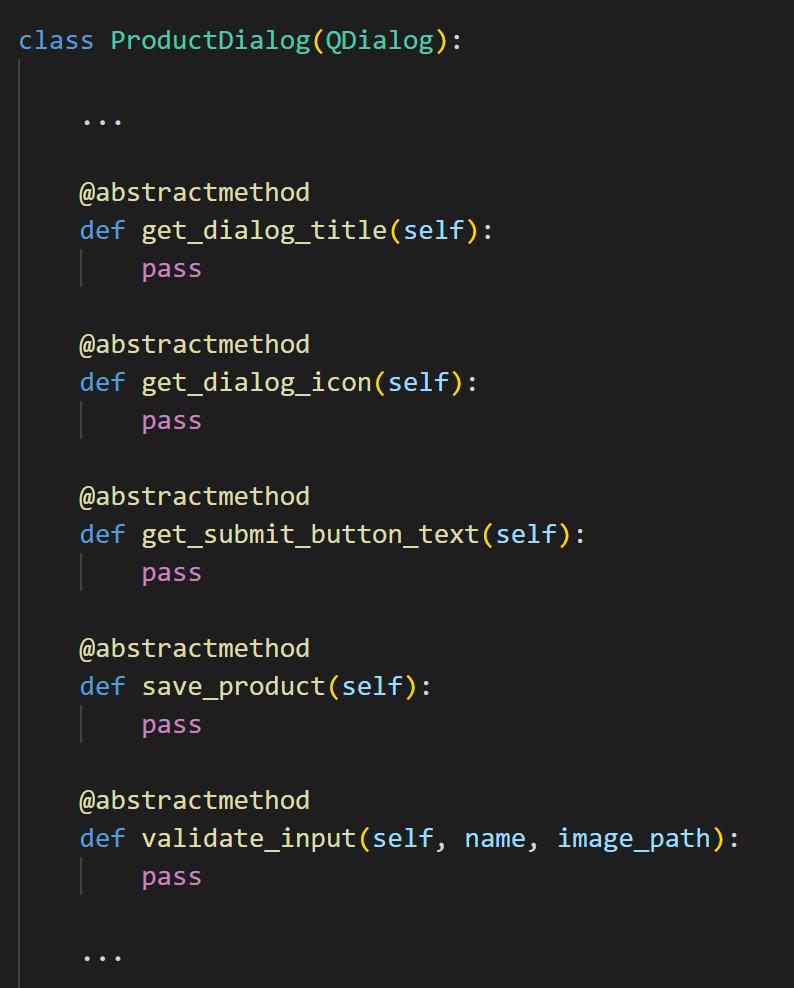


Abbildung : Klasse ProctDialog

Die Unterklassen AddProductDialog und EditProductDialog implementieren diese Methoden entsprechend ihrer jeweiligen Verwendungszwecke:

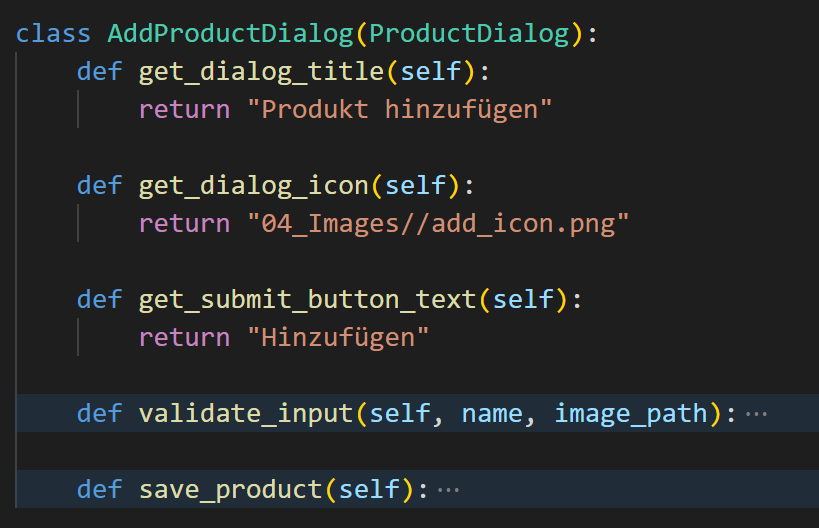


Abbildung : Klasse AddProductDialog

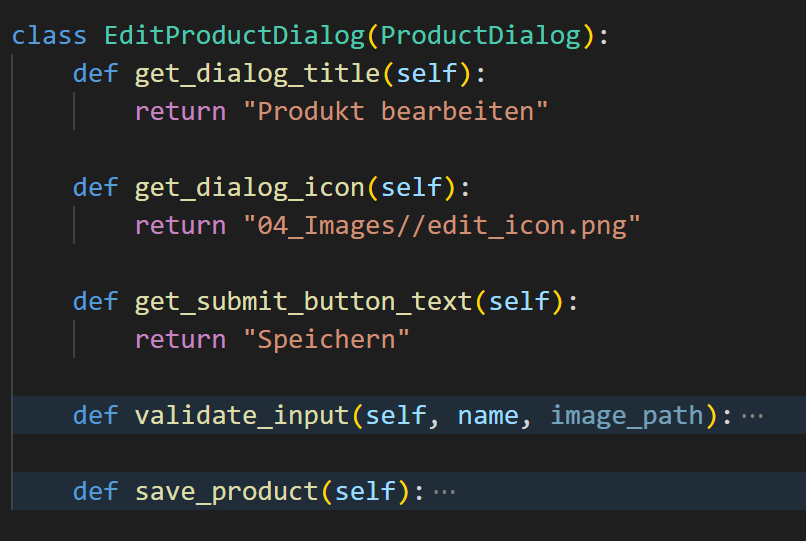


Abbildung : Klasse EditProductDialog

Da die Unterklassen die abstrakten Methoden der Basisklasse implementieren und das Verhalten des Programms nicht negativ beeinflussen, folgen sie dem Liskov Substitution Principle. Das bedeutet, dass Sie *ProductDialog*-Objekte durch *AddProductDialog*- oder *EditProductDialog*-Objekte ersetzen können, und das Programm wird weiterhin korrekt funktionieren.

### Interface Segregation Principle (ISP)

Das ISP besagt, dass Klassen nicht von Interfaces abhängig sein sollten, die sie nicht verwenden. Das Prinzip fördert die Trennung von Verantwortlichkeiten durch die Verwendung mehrerer kleinerer und fokussierter Schnittstellen anstelle einer großen monolithischen Schnittstelle. Im Code werden beispielsweiße folgenden Interfaces verwendet:

* ITransactionDataAccess
* IProductDataAccess
* IProductList
* ITransactionLog

Diese Interfaces definieren jeweils nur die Funktionen, die für die jeweiligen Klassen relevant sind. Zum Beispiel:

* TransactionLog implementiert ITransactionLog und verwendet ITransactionDataAccess. Es definiert nur die für das Transaktionsprotokoll relevanten Methoden.
* ProductList implementiert IProductList und verwendet IProductDataAccess. Es definiert nur die für die Produktliste relevanten Methoden.

Die Verwendung dieser fokussierten Schnittstellen ermöglicht es, die Klassen voneinander unabhängig und leicht verständlich zu gestalten, ohne unnötige Abhängigkeiten einzuführen.

### Dependency Inversion Principle (DIP)

Das Dependency Inversion Principle (DIP) besagt, dass High-Level-Module nicht von Low-Level-Modulen abhängig sein sollten, sondern beide von Abstraktionen abhängig sein sollten. Das bedeutet, dass die Abhängigkeiten auf Abstraktionen und nicht auf konkreten Implementierungen basieren sollten. Dieses Prinzip fördert die Entkopplung und Flexibilität in der Softwarearchitektur.

Im gegebenen Code sind einige Stellen, an denen das Dependency Inversion Principle angewendet wird:

* TransactionLog-Klasse: Die *TransactionLog*-Klasse erhält im Konstruktor ein Objekt, dass das *ITransactionDataAccess*-Interface implementiert, anstatt direkt von einer konkreten Implementierung abzuhängen. Dadurch ist die Klasse von der konkreten Implementierung entkoppelt und abhängig von der Abstraktion (Interface).

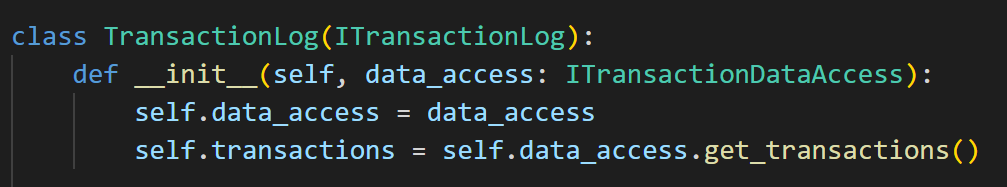


Abbildung : Klasse TransactionLog

* ProductList-Klasse: Ähnlich wie bei *TransactionLog* erhält die *ProductList*-Klasse im Konstruktor ein Objekt, dass das *IProductDataAccess*-Interface implementiert. Dadurch ist die Klasse von der konkreten Implementierung entkoppelt und abhängig von der Abstraktion (Interface).

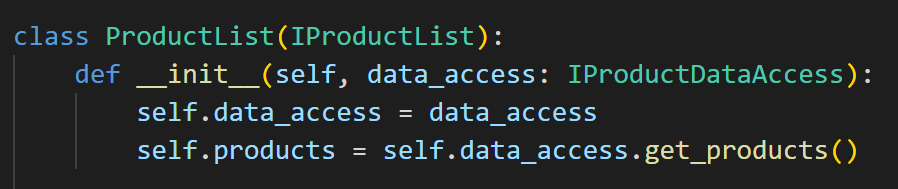


Abbildung : Klasse ProductList

An beiden Stellen wird das Dependency Inversion Principle angewendet, um die Abhängigkeiten auf Abstraktionen zu basieren und somit die Entkopplung und Flexibilität der Softwarearchitektur zu fördern.

## GRASP-Prinzipien

### Geringe Kopplung

Das Prinzip der geringen Kopplung (Low Coupling) ist eines der GRASP-Prinzipien und bezieht sich darauf, wie stark verschiedene Komponenten oder Klassen voneinander abhängig sind. Eine geringe Kopplung bedeutet, dass die Abhängigkeiten zwischen den Komponenten reduziert werden, um die Wartbarkeit und die Flexibilität des Codes zu erhöhen. Im Programmcode gibt es mehrere Beispiele für geringe Kopplung:

Dependency Injection

In der *VendingMachine*-Klasse werden Abhängigkeiten wie *product\_list*, *coinmanager* und *transactionmanager* über den Konstruktor injiziert. Dies ermöglicht es, verschiedene Implementierungen dieser Komponenten auszutauschen, ohne die *VendingMachine*-Klasse selbst zu ändern.

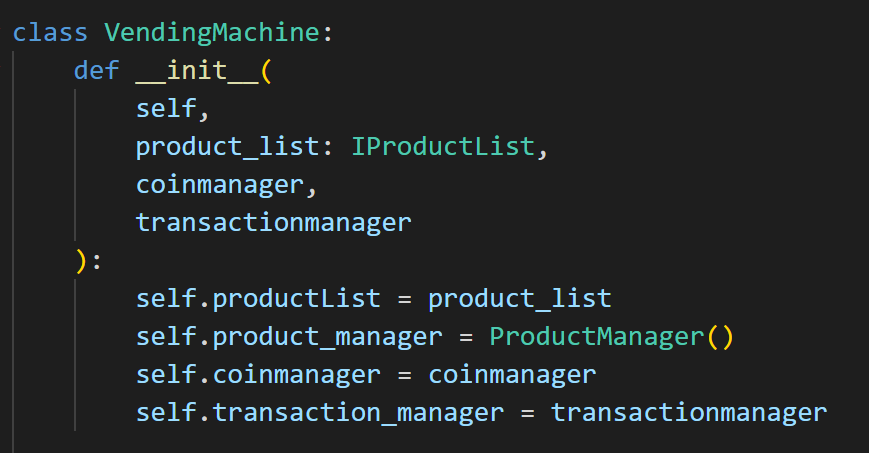


Abbildung : Klasse VendingMachine

Interface-Segregation

In der Datei *interfaces.py* gibt es mehrere Interface-Klassen, die unterschiedliche Verantwortlichkeiten definieren. Dadurch wird die Kopplung zwischen den verschiedenen Komponenten des Systems reduziert. Zum Beispiel werden Datenzugriffsfunktionen für Produkte (*IProductDataAccess*), Transaktionen (*ITransactionDataAccess*) und Konfigurationen (*IConfigDataAccess*) getrennt.

Verwendung von abstrakten Basisklassen (ABC)

Durch die Verwendung von abstrakten Basisklassen (z.B. *IProductList*, *ITransactionLog*, *IProductValidator*) wird die Kopplung zwischen den Klassen reduziert, da sie nur von den abstrakten Schnittstellen abhängig sind und nicht von konkreten Implementierungen.

Durch den Einsatz von Dependency Injection, Interface-Segregation und abstrakten Basisklassen (ABC) setzt der Programmcode das Prinzip der geringen Kopplung um. Diese Techniken ermöglichen es, die Abhängigkeiten zwischen den verschiedenen Komponenten Ihres Systems zu reduzieren und dadurch die Wartbarkeit und Flexibilität des Codes zu erhöhen.

## Hohe Kohäsion

Das Prinzip der hohen Kohäsion (High Cohesion) ist eines der GRASP-Prinzipien und bezieht sich darauf, wie gut die Verantwortlichkeiten innerhalb einer Klasse oder Komponente organisiert sind. Eine hohe Kohäsion bedeutet, dass jede Klasse oder Komponente eine klar definierte Aufgabe oder Verantwortung hat, was die Wartbarkeit, Verständlichkeit und Wiederverwendbarkeit des Codes verbessert.

Im Programmcode gibt es mehrere Beispiele für hohe Kohäsion:

Verantwortlichkeiten der Klassen

Jede Klasse in Ihrem Code hat eine klar definierte Aufgabe und Verantwortung. Zum Beispiel:

* VendingMachine: Verwaltet den Kaufprozess und interagiert mit anderen Komponenten wie *ProductManager*, *CoinManager* und *TransactionManager*
* ProductManager: Verwaltet Produkte und deren Auswahl, Lagerbestand und Erstellung
* CoinManager: Verwaltet Münzen und deren Hinzufügen, Subtrahieren und Zurücksetzen
* TransactionManager: Verwaltet Transaktionen und deren Hinzufügen und Abrufen

**Organisation der Funktionen**

Die Funktionen innerhalb der Klassen sind auf ihre spezifischen Verantwortlichkeiten fokussiert und haben jeweils eine klar definierte Aufgabe. Zum Beispiel in der *ProductManager*-Klasse:

* select\_product: Wählt ein Produkt aus
* get\_products: Gibt die Liste der Produkte zurück
* update\_stock: Aktualisiert den Lagerbestand eines ausgewählten Produkts
* *create\_product*: Erstellt ein neues Produkt

Trennung von Datenzugriff und Geschäftslogik

Durch die Verwendung von Interfaces für den Datenzugriff (z.B. *IProductDataAccess*, *ITransactionDataAccess*) wird die Geschäftslogik von der Datenzugriffsschicht getrennt, was zu einer höheren Kohäsion der einzelnen Komponenten führt.

Verwendung von Interfaces für Validierung und Listenverwaltung

Die Verwendung von Interfaces wie *IProductValidator* und *IProductList* ermöglicht es, die Validierung und Listenverwaltung von den Hauptklassen zu trennen, wodurch die Kohäsion erhöht wird.

Der Code zeigt hohe Kohäsion durch klar definierte Verantwortlichkeiten der Klassen, gut organisierte Funktionen und die Trennung von Datenzugriff, Geschäftslogik, Validierung und Listenverwaltung. Diese Faktoren tragen zur Verbesserung der Wartbarkeit, Verständlichkeit und Wiederverwendbarkeit des Codes bei.

## DRY-Prinzip

Das DRY-Prinzip (Don't Repeat Yourself) ist ein grundlegendes Prinzip in der Softwareentwicklung, das darauf abzielt, Wiederholungen im Code zu vermeiden und Redundanz zu reduzieren. Durch die Einhaltung des DRY-Prinzips wird der Code wartbarer, verständlicher und wiederverwendbarer. Im Folgenden sind einige Beispiele dafür aufgeführt, wie beim Programmcode das DRY-Prinzip umgesetzt wurde:

**Verwendung von Klassen und Funktionen zur Kapselung gemeinsamer Logik**

* CoinManager: Diese Klasse kapselt die Logik zur Verwaltung von Münzen. Sie stellt Funktionen wie add\_coin, get\_total\_amount und sub\_coin bereit, die an verschiedenen Stellen im Code verwendet werden können, ohne dass die Logik dupliziert werden muss.
* ProductManager: Die ProductManager-Klasse kapselt die Logik zur Verwaltung von Produkten und bietet Funktionen wie select\_product, get\_products, update\_stock und create\_product, die den gemeinsamen Code zentralisieren und Wiederholungen vermeiden.

**Auslagern von redundantem Code in Funktionen**

Die Klasse ConfigDialog verwendet in mehreren Methoden (setup\_ui, add\_product, edit\_product) dieselben Code-Zeilen:

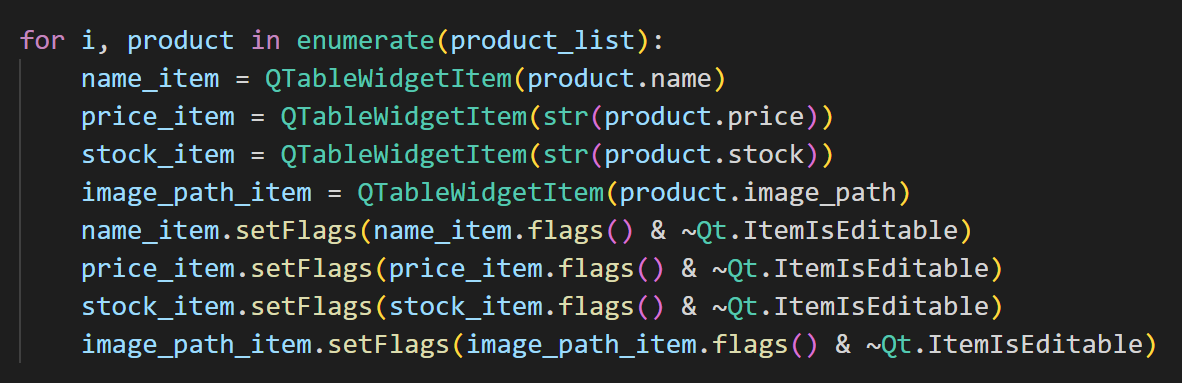


Abbildung . Redundanter Code in der Klasse ConfigDialog

Die Befehle wurden in eine seperate Methode ausgelagert:

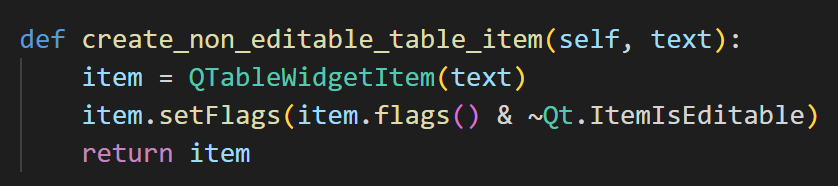


Abbildung : Ausgelagerter redundanter Code

Dadurch wird nun jeweils die Methode aufgerufen:

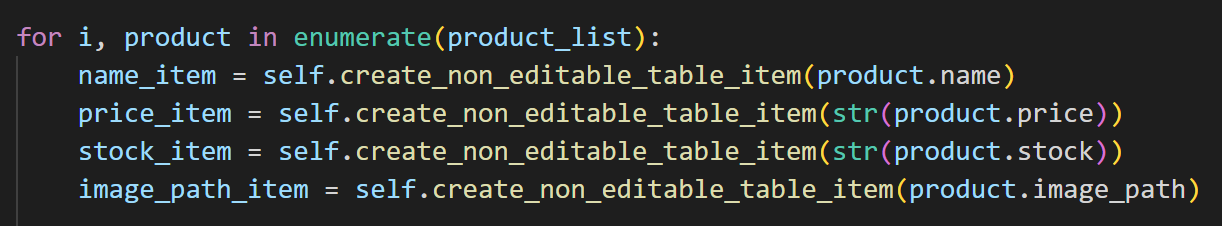


Abbildung : Code ohne duplikation

Dadurch werden Codeduplikate im Sinne des DRY-Prinzips vermieden.

# Entwurfsmuster

# Unit Tests

# Refactoring