

Dokumentation S.N.A.C.K

Dokumentation zum Praxisprojekt im Kurs Advanced Software
Engineering

an der

Dualen Hochschule Baden-Württemberg Karlsruhe

von

Burak Özkan
(Matrikel-Nr.:9015631)

Marius Engelmeier
(Matrikel-Nr.: 9944072)

Abgabedatum:

31.05.2023

Inhaltsverzeichnis

Inhaltsverzeichnis	2
1. Github Referenz	3
2. Projektbeschreibung	3
3. Clean Architecture.....	4
4. Programming Principles	6
4.1 SOLID-Prinzipien	6
4.1.1 Single Responsibility Principle (SRP).....	6
4.1.2 Open-Closed Principle (OCP)	6
4.1.3 Liskov Substitution Principle (LSP)	7
4.1.4 Interface Segregation Principle (ISP).....	7
4.1.5 Dependency Inversion Principle (DIP)	8
4.2 GRASP-Prinzipien	9
4.2.1 Low Coupling	9
4.3 High Cohesion	10
4.4 DRY-Prinzip	11
5. Entwurfsmuster	12
5.1 Singleton.....	12
5.2 Beobachter	13
6. Unit Tests	15
6.1 ATRIP	15
6.2 Code Coverage.....	16
6.3 Mocks	17
7. Refactoring.....	18
7.1 Einkapseln der Datenbankverbindung in einer Methode	18
7.2 Verwenden des Factory-Musters für die Erstellung von Controllern:	19
7.3 Refactoring von Magic Strings.....	20
7.4 Code Smell: Long Method:	21
7.5 Code Smell :Large Class:	22
7.6 Code Smell Shotgun Surgery	23

1. Github Referenz

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

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

2. Projektbeschreibung

S.N.A.C.K steht für das **st**ilvolle **ne**ue **attr**aktive **co**ole **K**rabbersystem. 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.

Nachdem das Programm gestartet wurde, wird die Standard-Seite angezeigt. Über diese können Münzen eingeworfen und Produkte gekauft werden. Ein Label gibt verschiedene Statusmeldungen über den Kauf aus. Über einen Info-Button können erweiterte Informationen eingesehen sowie Feedback übermittelt werden. Über einen Button namens „Konfigurationsmenü“ gelangt man nach der Eingabe des korrekten Pins in das Konfigurationsmenü, in welchem das Sortiment angepasst, Transaktionen eingesehen, Statistiken angezeigt sowie Konfigurationen am Automaten durchgeführt werden 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. Für die grafische Oberfläche wurde das Framework „Qt“ verwendet. Für die Speicherung von Daten wird eine SQLite-Datenbank verwendet.

3. 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**

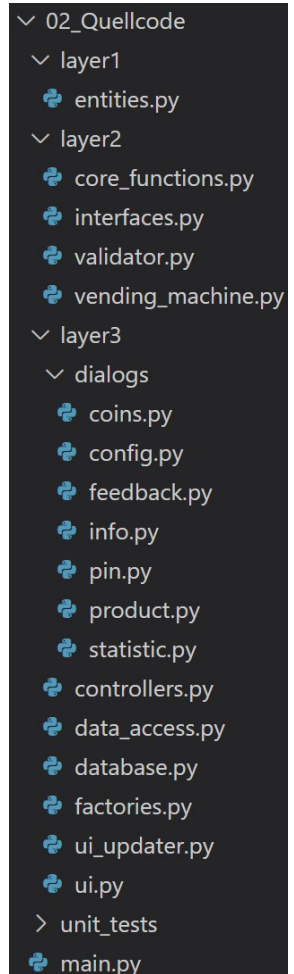
- Core-Logik und grundlegende Implementierung

- 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.



```
02_Quellcode
├── layer1
│   └── entities.py
├── layer2
│   ├── core_functions.py
│   ├── interfaces.py
│   ├── validator.py
│   └── vending_machine.py
├── layer3
│   ├── dialogs
│   │   ├── coins.py
│   │   ├── config.py
│   │   ├── feedback.py
│   │   ├── info.py
│   │   ├── pin.py
│   │   ├── product.py
│   │   ├── statistic.py
│   │   └── controllers.py
│   ├── data_access.py
│   ├── database.py
│   ├── factories.py
│   ├── ui_updater.py
│   ├── ui.py
│   ├── unit_tests
│   └── main.py
```

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.

4. Programming Principles

4.1 SOLID-Prinzipien

4.1.1 Single Responsibility Principle (SRP)

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

- **DefaultProductValidator (validator.py)**

Diese Klasse ist ausschließlich für die Validierung von Eingaben zuständig. Durch die Konzentration auf eine einzige Verantwortung – die Validierung von Eingaben – bleibt der Code in dieser Klasse fokussiert und leicht verständlich. Wenn es notwendig wird, die Validierungslogik zu ändern oder zu erweitern, betrifft dies nur diese Klasse, ohne Auswirkungen auf andere Teile des Programms.

- **UIUpdater (ui_updater.py)**

Diese Klasse 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.

4.1.2 Open-Closed Principle (OCP)

Das Open-Closed Principle besagt, dass Software-Entitäten 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.

4.1.3 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. Die Unterklassen *AddProductDialog* und *EditProductDialog* implementieren diese Methoden entsprechend ihrer jeweiligen Verwendungszwecke. Da die Unterklassen die abstrakten Methoden der Basisklasse implementieren und das Verhalten des Programms nicht negativ beeinflussen, folgen sie dem Liskov Substitution Principle.

4.1.4 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 beispielsweise 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.

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

4.1.5 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. Dieses Prinzip fördert die Entkopplung und Flexibilität in der Softwarearchitektur. Im Programmcode wird das Dependency Inversion Principle zum Beispiel in der Klasse `TransactionLog` angewendet. Die Klasse erhält im Konstruktor ein Objekt, das 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).

```
class TransactionLog(ITransactionLog):
    def __init__(self, data_access: ITransactionDataAccess):
        self.data_access = data_access
        self.transactions = self.data_access.get_transactions()
```


4.2 GRASP-Prinzipien

4.2.1 Low Coupling

Das Low Coupling Prinzip (geringen Kopplung) 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 die Anwendung dieses Prinzips:

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.

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.

4.2.2 High Cohesion

Das High Cohesion Prinzip (hohen Kohäsion) 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

4.3 DRY-Prinzip

Das DRY-Prinzip (Don't Repeat Yourself) zielt darauf ab, Wiederholungen im Code zu vermeiden und Redundanzen zu reduzieren. Durch die Einhaltung des DRY-Prinzips wird der Code wartbarer, verständlicher und wiederverwendbarer. Im nachfolgenden ist ein Beispiel, an dem das Prinzip im Programmcode angewandt wurde, indem redundanter Code in eine Funktion ausgelagert wurde:

Die Klasse *ConfigDialog* verwendete in mehreren Methoden (*setup_ui*, *add_product*, *edit_product*) dieselben Zeilen Code:

```
for i, product in enumerate(product_list):
    name_item = QTableWidgetItem(product.name)
    price_item = QTableWidgetItem(str(product.price))
    stock_item = QTableWidgetItem(str(product.stock))
    image_path_item = QTableWidgetItem(product.image_path)
    name_item.setFlags(name_item.flags() & ~Qt.ItemIsEditable)
    price_item.setFlags(price_item.flags() & ~Qt.ItemIsEditable)
    stock_item.setFlags(stock_item.flags() & ~Qt.ItemIsEditable)
    image_path_item.setFlags(image_path_item.flags() & ~Qt.ItemIsEditable)
```

Die Befehle wurden in eine separate Methode ausgelagert:

```
def create_non_editable_table_item(self, text):
    item = QTableWidgetItem(text)
    item.setFlags(item.flags() & ~Qt.ItemIsEditable)
    return item
```

Dadurch wird nun jeweils lediglich die Methode aufgerufen:

```
for i, product in enumerate(product_list):
    name_item = self.create_non_editable_table_item(product.name)
    price_item = self.create_non_editable_table_item(str(product.price))
    stock_item = self.create_non_editable_table_item(str(product.stock))
    image_path_item = self.create_non_editable_table_item(product.image_path)
```

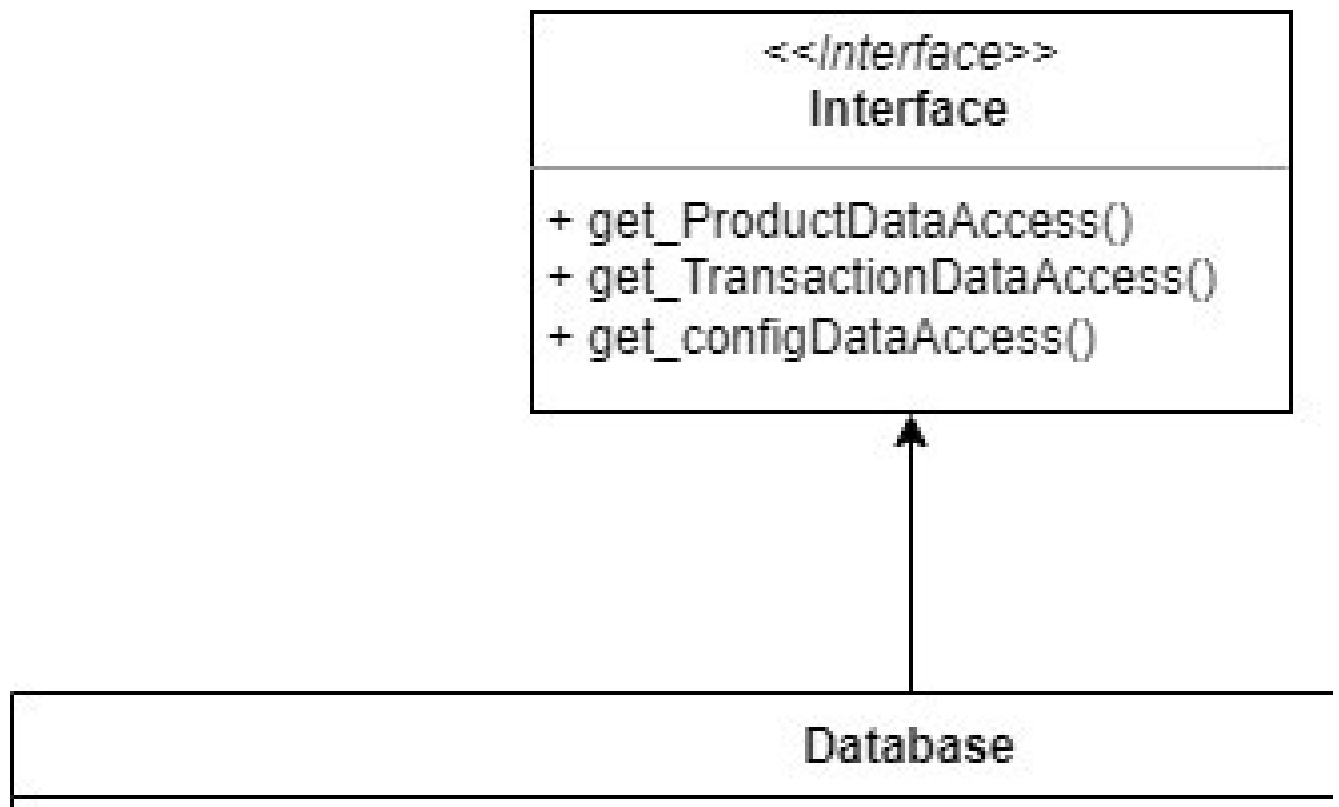
5. Entwurfsmuster

5.1 Singleton

Das Singleton-Muster wird für die Datenbank verwendet, um sicherzustellen, dass nur eine Instanz der Datenbankklasse vorhanden ist. Dadurch wird sichergestellt, dass nur eine Verbindung zur Datenbank besteht und verschiedene Teile des Programms auf dieselbe Datenbankverbindung zugreifen.

Dies hat mehrere Vorteile. Erstens werden Ressourcen effizient genutzt, da nur eine Verbindung geöffnet wird und die Ressourcennutzung optimiert wird. Zweitens werden alle Abfragen an die Datenbank über dieselbe Instanz durchgeführt, wodurch Konsistenz und Datenkonsistenz sichergestellt werden. Drittens vereinfacht es die Kommunikation zwischen verschiedenen Teilen des Programms, nicht mehrere Verbindungsobjekte verwaltet werden oder Informationen zwischen verschiedenen Instanzen synchronisiert werden müssen.

Das Singleton-Muster für Datenbank wird durch die Verwendung der Klassenvariablen `_instance` und der Methode `__new__` erreicht. Die Methode `__new__` prüft, ob bereits eine Instanz von Database vorhanden ist. Andernfalls wird eine neue Instanz erstellt und in `cls._instance` gespeichert. Nachfolgende Aufrufe der Methode `__new__` geben die vorhandene Instanz zurück.



5.2 Beobachter

In der Anwendung wurde auch das Beobachter-Entwurfsmuster verwendet, um die Interaktion zwischen der GUI (Beobachter) und der Anwendungslogik (beobachtete Objekte) zu organisieren. Das Observer-Muster ist ein Software-Design-Konstrukt, bei dem ein oder mehrere Observer-Objekte den Zustand eines Subjekts überwachen und automatisch benachrichtigt werden, wenn sich dieser Zustand ändert.

In der Anwendung verfügt die Klasse „VendingMachineController“ über eine Liste von Beobachtern. Benachrichtigungen werden an diese Beobachter gesendet, wenn bestimmte Aktionen ausgeführt werden, z. B. die Auswahl eines Produkts („select_product“) oder der Kauf eines Produkts („buy_product“).

„UIUpdater“ ist ein Beobachter in der Anwendung. Benachrichtigungen vom „VendingMachineController“ werden erhalten und die Benutzeroberfläche wird dementsprechend aktualisiert. Wenn beispielsweise „VendingMachineController“ eine Benachrichtigung vom Typ „status_label“ sendet, ruft „UIUpdater“ die Methode „update_status_label“ auf, um das entsprechende Label in der Benutzeroberfläche zu aktualisieren.

Dieses Muster trägt dazu bei, eine saubere Trennung zwischen Anwendungslogik und Benutzeroberfläche zu schaffen. Dies erleichtert das Testen und Warten des Codes, da die Anwendungslogik unabhängig von einer bestimmten Benutzeroberfläche getestet werden kann und umgekehrt.

Es trägt auch dazu bei, den Code flexibler und erweiterbarer zu machen. Wenn beispielsweise eine neue Art von Benachrichtigung hinzugefügt werden soll, wird einfach eine neue Methode zu „Observer“ hinzugefügt und die entsprechende Methode „notify_observers“ wird aufgerufen.

Darüber hinaus können ist es möglich mit dem Beobachtermuster auch mehrere Benutzeroberflächen oder andere Beobachter gleichzeitig zu unterstützen, da das Subjekt eine Liste von Beobachtern verwaltet und alle Beobachter gleichzeitig benachrichtigt, wenn sich ihr Zustand ändert.



6. Unit Tests

In der Anwendung wurden die Funktionalitäten der Businesslogik bzw. alle Klassen von Layer 2 durch Unit Test sichergestellt. Insgesamt wurden hierfür 20 Tests erstellt.

6.1 ATRIP

Die Unit Tests befolgen die ATRIP-Regeln (Automatic, Thorough, Repeatable, Independent, Professional), die einen wichtigen Rahmen für effektive Testpraktiken bilden.

Automatic

Die Tests sind so konzipiert, dass sie automatisch ausgeführt und überprüft werden können. Die unittest-Bibliothek in Python erlaubt das Schreiben von Tests, die automatisch durchgeführt und validiert werden. Die assert-Methoden garantieren, dass das erwartete Ergebnis mit dem tatsächlichen Ergebnis übereinstimmt.

Thorough

Die Tests sind gründlich und decken viele verschiedene Fälle und Randbedingungen ab. Die verschiedenen Methoden in den Klassen werden getestet, um sicherzustellen, dass sie unter verschiedenen Bedingungen korrekt funktionieren. Beispielsweise wird in der TestDefaultProductValidator-Klasse die is_valid_name-Methode mit verschiedenen Eingabeparametern getestet, um sicherzustellen, dass sie in allen Situationen korrekt arbeitet.

Repeatable

Alle Tests können jederzeit wiederholt werden und liefern konsistente Ergebnisse. Das Einrichten und Aufräumen von Tests wird durch die setUp- und tearDown-Methoden erreicht, die sicherstellen, dass jeder Test unter den gleichen Bedingungen läuft. Außerdem verwenden die Tests "Mocks", um Abhängigkeiten zu isolieren, was dazu beiträgt, dass sie unter den gleichen Bedingungen wiederholbar sind.

Independent

Die Tests sind unabhängig voneinander gestaltet. Jeder Test kann unabhängig von den anderen ausgeführt werden, und die Reihenfolge der Testausführung hat keinen Einfluss auf das Ergebnis. Dies wird durch die Verwendung von "Mocks" erreicht, die es ermöglichen, Abhängigkeiten zu isolieren und zu steuern. Darüber hinaus sorgen die setUp- und tearDown-Methoden dafür, dass der Zustand vor und nach jedem Test zurückgesetzt wird, um die Unabhängigkeit zu gewährleisten.

Professional

Die Tests sind professionell geschrieben. Sie sind klar und leicht zu verstehen. Jeder Test ist auf eine bestimmte Funktion oder Methode ausgerichtet, was dazu beiträgt, dass der Code übersichtlich und gut organisiert ist. Darüber hinaus sind die Tests an einigen Stellen kommentiert, was dazu beiträgt, dass sie leicht verständlich sind und andere Entwickler sie leichter verstehen und nutzen können.

6.2 Code Coverage

Mit dem Tool *coverage* für Python wurde die Testabdeckung der Anwendung überprüft. Das Resultat ist in nachfolgender Grafik zu sehen.

<i>Module</i>	<i>statements</i>	<i>missing</i>	<i>excluded</i>	<i>coverage</i>
Documents\GitHub\S.N.A.C.K\02_Quellcode\layer1\entities.py	29	16	0	45%
Documents\GitHub\S.N.A.C.K\02_Quellcode\layer2\core_functions.py	87	52	0	40%
Documents\GitHub\S.N.A.C.K\02_Quellcode\layer2\interfaces.py	82	24	0	71%
Documents\GitHub\S.N.A.C.K\02_Quellcode\layer2\validator.py	14	8	0	43%
Documents\GitHub\S.N.A.C.K\02_Quellcode\layer2\vending_machine.py	25	18	0	28%
Documents\GitHub\S.N.A.C.K\02_Quellcode\layer3\controllers.py	91	54	0	41%
Documents\GitHub\S.N.A.C.K\02_Quellcode\layer3\data_access.py	101	78	0	23%
Documents\GitHub\S.N.A.C.K\02_Quellcode\layer3\database.py	21	12	0	43%
Documents\GitHub\S.N.A.C.K\02_Quellcode\layer3\dialogs\coins.py	39	31	0	21%
Documents\GitHub\S.N.A.C.K\02_Quellcode\layer3\dialogs\config.py	139	123	0	12%
Documents\GitHub\S.N.A.C.K\02_Quellcode\layer3\dialogs\feedback.py	28	23	0	18%
Documents\GitHub\S.N.A.C.K\02_Quellcode\layer3\dialogs\info.py	29	22	0	24%
Documents\GitHub\S.N.A.C.K\02_Quellcode\layer3\dialogs\pin.py	42	32	0	24%
Documents\GitHub\S.N.A.C.K\02_Quellcode\layer3\dialogs\product.py	123	91	0	26%
Documents\GitHub\S.N.A.C.K\02_Quellcode\layer3\dialogs\statistic.py	16	12	0	25%
Documents\GitHub\S.N.A.C.K\02_Quellcode\layer3\factories.py	42	13	0	69%
Documents\GitHub\S.N.A.C.K\02_Quellcode\layer3\ui.py	91	72	0	21%
Documents\GitHub\S.N.A.C.K\02_Quellcode\layer3\ui_updater.py	25	19	0	24%
Documents\GitHub\S.N.A.C.K\02_Quellcode\main.py	22	11	0	50%
Total	1046	711	0	32%

6.3 Mocks

Mock-Objekte werden verwendet, um bestimmte Aspekte einer Klasse oder eines Objekts zu simulieren, ohne die konkrete Klasse zu instanziiieren. In den Unit-Tests werden an mehreren Stellen Mock-Objekte eingesetzt. Zum Beispiel in der Klasse *TestSalesCalculator*:

Im Testfall *test_get_total_sales* werden Mock-Objekte verwendet, um Transaktionen zu simulieren. Den Mock-Objekten werden jeweils das Attribut *amount_paid* hinzugefügt. Die Zeile *transactions = [Mock(amount_paid=5), Mock(amount_paid=10)]* erstellt eine Liste von zwei simulierten Transaktionen: eine, bei der 5 Einheiten bezahlt wurden, und eine, bei der 10 Einheiten bezahlt wurden. Dann wird getestet, ob die Methode *get_total_sales* korrekt funktioniert, indem überprüft wird, ob sie den korrekten Gesamtbetrag der Transaktionen zurückgibt.

```
def test_get_total_sales(self):  
    transactions = [Mock(amount_paid=5), Mock(amount_paid=10)]  
    self.assertEqual(self.calculator.get_total_sales(transactions), 15)
```

7. Refactoring

7.1 Einkapseln der Datenbankverbindung in einer Methode

Vor Refactoring wird die Verbindung zur Datenbank im globalen Bereich von main.py erstellt. Um den Code sauberer und wiederverwendbarer zu machen, kann die Datenbankverbindung in einer separaten Methode gekapselt werden.

```
db_path = "03_SQL//database//vendingMachine.db"

if __name__ == "__main__":
    conn = sqlite3.connect(db_path)
    db = Database(ProductDataAccess(conn), TransactionDataAccess(conn), ConfigDataAccess(conn))
```

Der Vorteil dieses Ansatzes besteht darin, dass die Details der Datenbankverbindungserstellung abstrahiert werden, was es einfacher macht, Verbindungsdetails zu einer Datenbank zu ändern oder in Zukunft zu einer anderen Datenbank zu wechseln.

```
def create_database_connection(db_path):
    conn = sqlite3.connect(db_path)
    return Database(ProductDataAccess(conn), TransactionDataAccess(conn), ConfigDataAccess(conn))

class ControllerFactory: ...

if __name__ == "__main__":
    db = create_database_connection("03_SQL//database//vendingMachine.db")
```

7.2 Verwenden des Factory-Musters für die Erstellung von Controllern:

Viele Controller-Objekte werden in main.py erstellt.

```
productcontroller = ProductController(db.get_ProductDataAccess())
transactioncontroller = TransactionController(db.get_TransactionDataAccess())
statcontroller = StatController(transactioncontroller)
coincontroller = CoinController()
configcontroller = ConfigController(db.get_ConfigDataAccess())
vmcontroller = VendingMachineController(db, coincontroller, transactioncontroller)

app = QApplication(sys.argv)
gui = VendingMachineGUI(
    vmcontroller,
    configcontroller,
    statcontroller,
    coincontroller,
    productcontroller,
    transactioncontroller,
```

Dies kann in eine Factory-Funktion oder -Klasse umgestaltet werden, die diese Objekte erstellt. Dadurch wird main.py bereinigt und die Erstellung dieser Objekte wird einfacher.

```
class ControllerFactory:
    def __init__(self, db):
        self.db = db

    def create_controllers(self):
        productcontroller = ProductController(self.db.get_ProductDataAccess())
        transactioncontroller = TransactionController(self.db.get_TransactionDataAccess())
        statcontroller = StatController(transactioncontroller)
        coincontroller = CoinController()
        configcontroller = ConfigController(self.db.get_ConfigDataAccess())
        vmcontroller = VendingMachineController(self.db, coincontroller, transactioncontroller)
        return vmcontroller, configcontroller, statcontroller, coincontroller, productcontroller, transactioncontroller

if __name__ == "__main__":
    db = create_database_connection("03_SQL//database//vendingMachine.db")
    factory = ControllerFactory(db)
    vmcontroller, configcontroller, statcontroller, coincontroller, productcontroller, transactioncontroller = factory.create_controllers()
```

Dieses Refactoring trägt dazu bei, das Hauptskript sauber zu halten und verbessert die Modularität und Lesbarkeit des Codes.

7.3 Refactoring von Magic Strings

Es gibt einige Magic Strings (hartcodierte Zeichenfolgenwerte), die mehrmals vorkommen, beispielsweise die Zeichenfolgen „product_buttons“, „status_label“, „coin_label“.

```
def update(self, notification_type, *args):
    if notification_type == "product_buttons":
        self.update_product_buttons(*args)
    elif notification_type == "status_label":
        self.update_status_label(*args)
    elif notification_type == "coin_label":
        self.update_coin_label(*args)
```

Anstatt die Zeichenfolge direkt zu verwenden, können die Zeichenfolgen als Konstanten deklariert und auf diese Konstante verwiesen werden. Dies hilft, Tippfehler zu vermeiden und zukünftige Zeichenfolgenänderungen einfacher zu machen.

```
PRODUCT_BUTTONS="product_buttons"
STATUS_LABEL= "status_label"
COIN_LABEL= "coin_label"

✓ class UIUpdater(Observer):
✓     def __init__(self, products_groupbox, product_buttons, status_label, coin_label, product_button_factory):
        self.products_groupbox = products_groupbox
        self.product_buttons = product_buttons
        self.status_label = status_label
        self.coin_label = coin_label
        self.product_button_factory = product_button_factory

✓     def update(self, notification_type, *args):
✓         if notification_type == PRODUCT_BUTTONS:
✓             self.update_product_buttons(*args)
✓         elif notification_type == STATUS_LABEL:
✓             self.update_status_label(*args)
✓         elif notification_type == COIN_LABEL:
            self.update_coin_label(*args)
```

Dadurch wird der Code wartbarer. Wenn man die Zeichenfolge ändern möchte, muss dies nur an einer Stelle geändert werden und es werden auch Fehler durch Tippfehler in der Zeichenfolge vermieden.

7.4 Code Smell: Long Method

Ein Beispiel für eine lange Methode ist „setup_ui“ in der Klasse „VendingMachineGUI“. Diese Methode enthält einige Codezeilen und führt mehrere Funktionen aus.

```
def setup_ui(self):
    self.setWindowTitle("S.N.A.C.K Verkaufsautomat")
    self.setWindowIcon(QIcon("04_Images/vm_icon.png"))
    self.product_buttons = []
    layout = QGridLayout()

    self.status_label = QLabel("Bitte wählen Sie ein Produkt aus.")
    layout.addWidget(self.status_label, 0, 0, 1, 3)

    self.products_groupbox = QGroupBox("Produkte")
    products_layout = QGridLayout()
    self.products_groupbox.setLayout(products_layout)
    layout.addWidget(self.products_groupbox, 1, 0, 1, 3)

    self.coin_button = QPushButton("Münzen einwerfen")
    self.coin_button.clicked.connect(self.show_coin_dialog)
    layout.addWidget(self.coin_button, 2, 3)

    self.config_button = QPushButton("Konfigurationsmenü")
    self.config_button.clicked.connect(self.show_config_dialog)
    layout.addWidget(self.config_button, 3, 3)

    self.buy_button = QPushButton("Kaufen")
    self.buy_button.clicked.connect(self.buy_product)
    layout.addWidget(self.buy_button, 4, 3)

    self.coin_label = QLabel("0.0 €")
    layout.addWidget(self.coin_label, 5, 0)

    self.info_button = QPushButton("Info")
    self.info_button.clicked.connect(self.show_info_dialog)
    layout.addWidget(self.info_button, 6, 3)

    self.ui_updater = UIUpdater(self.products_groupbox, self.product_buttons, self.status_label, self.coin_label, self.product_button_factory)
    self.ui_updater.update_product_buttons(self.vmcontroller, self.select_product)

    self.setLayout(layout)
    self.vmcontroller.add_observer(self.ui_updater)
```

Lange Methoden sind schwer zu lesen, zu verstehen und zu warten. Oft wird zu viel Verantwortung auf eine einzelne Methode gelegt. Um dieses Problem zu lösen, sollten lange Methoden in kleinere, spezifischere Methoden aufgeteilt werden, um die Lesbarkeit und Wartbarkeit des Codes zu verbessern.

7.5 Code Smell: Large Class

Ein Beispiel für eine große Klasse ist „VendingMachineController“. Diese Klasse enthält eine Reihe von Methoden und hat mehrere Verantwortlichkeiten, z. B. den Zugriff auf Datenbankobjekte und die Verwaltung der Verkaufslogik.

```
class VendingMachineController:
    def __init__(self, db: IDatabase, coinmanager, transactionmanager):
        self.product_data_access = db.get_ProductDataAccess()
        self.transaction_data_access = db.get_TransactionDataAccess()
        self.config_data_access = db.get_ConfigDataAccess()
        self.product_list = ProductList(self.product_data_access)
        self.coin_manager = coinmanager
        self.transactionmanager = transactionmanager
        self.observers = []
        self.vending_machine = VendingMachine(self.product_list, self.coin_manager, self.transactionmanager)

    def add_observer(self, observer):
        self.observers.append(observer)

    def notify_observers(self, notification_type, *args):
        for observer in self.observers:
            observer.update(notification_type, *args)

    def get_products(self):
        return self.vending_machine.get_products()

    def select_product(self, product):
        self.vending_machine.select_product(product)

    def get_productList(self):
        return self.product_list

    def buy_product(self):
        return self.vending_machine.buy_product()

    def get_total_amount(self):
        return self.coin_manager.get_total_amount()
```

Große Klassen sind schwer zu verstehen, zu testen und zu warten. Sie verstoßen häufig gegen das Single-Responsibility-Prinzip (SRP) und können aufgrund der Vielzahl an Methoden und Attributen verwirrend sein. Um dieses Problem zu lösen, müssen die Funktionalität einer großen Klasse in kleinere, spezifischere Klassen aufgeteilt werden. Das Single-Responsibility-Prinzip sollte verwendet werden, um eine Klasse auf eine einzige Verantwortung zu reduzieren.

7.6 Code Smell: Shotgun Surgery

Ein Beispiel für Shotgun Surgery ist die Klasse „VendingMachineController“. In dieser Klasse erfolgt die Verwaltung von Beobachtern und die Benachrichtigung von Beobachtern mit verschiedenen Methoden wie „add_observer“, „notify_observers“, „get_products“, „select_product“, und „buy product“. Viele Methoden in verschiedenen Klassen müssen angepasst werden, wenn sich die Logik zur Benachrichtigung von Beobachtern ändert. Dies führt zu einer hohen Kopplung und einer geringeren Kohäsion. Um dieses Problem zu lösen, sollte die Verantwortung für die Verwaltung und Benachrichtigung von Beobachtern in eine separate Klasse verlagert werden, um Abhängigkeiten und Streuschussoperationen zu reduzieren. Es sollte das Prinzip der Einzelverantwortung angewendet werden.