

Spielebibliothek

Programmmentwurf

im Studiengang **Informationstechnik**

an der Dualen Hochschule Baden-Württemberg Karlsruhe

von

Jakob Gehrmann

Jannick Gutekunst

Abgabedatum

31. Mai 2023

Kurs.....TINF20B3

Dozent.....Herr Daniel Lindner

Inhaltsverzeichnis

Einleitung.....	3
Idee	3
Erste Umsetzung.....	3
Clean Architecture.....	5
Domain Code	5
Application Code.....	5
Adapter	5
Plugins.....	6
Main.....	6
UML-Diagramm	6
Entwurfsmuster.....	7
Beobachter	7
Besucher	8
Erbauer	10
Programming Principles	11
SOLID.....	11
Single Responsibility Principle	11
Open-Closed Principle.....	12
Liskov Substitution Principle	12
Interface Segregation Principle.....	14
Dependency Inversion Principle	14
GRASP	15
Low Coupling.....	15
High Cohesion	16
DRY.....	16
Refactoring	17
Code Smell 1 – High Cyclomatic complexity.....	17
Optimierung.....	18
Code Smell 2 – Long Method.....	19
Optimierung.....	19
Code Smell 3 – Duplicated Code.....	20
Optimierung.....	20

Code Smell 4 – Shotgun Surgery	20
Optimierung	21
Unit Test	22
Klasse <i>Account</i>	22
Klasse <i>AccountList</i>	22
Klasse <i>PostgresSqlAdapter</i>	23
Klasse <i>Statistics</i>	24
Klasse <i>TicTacToe</i>	25
ATRIP-Regeln	25
Automatic	25
Thorough	26
Repeatable	26
Independent	26
Professional	26
Code-Coverage	26
Einsatz von Mocks	28

Einleitung

Idee

Wir Programmieren eine Spielebibliothek in Python (Entwicklungsumgebung Visual Studio Code). Wir haben uns Python ausgesucht, da es eine Programmiersprache ist, die wenig Vorgaben mit Blick auf Struktur (keine Klassen-Vorgabe oder komplizierte Syntax) hat. Ebenfalls bietet Python diverse Bibliotheken zu verschiedensten Anwendungszwecken, darunter auch für Spiele mit Hilfe von „pygame“.

Die Spielebibliothek ist ein Konstruiertes Beispiel, um die in der Vorlesung vermittelten Konzepte einzubringen. Dabei sollen gleichzeitig verschiedenen Spielen, welche bereits online vorhanden sind, die genauere Ausarbeitung der Spielelogik abnehmen, und die Entwicklung beschleunigen.

Der Grundgedanke ist es eine Plattform zu schaffen, welche vergleichsweise einfach um Spiele erweitert werden kann. Zu diesem Zweck soll das Menü, die Spieler, die Spielerstatistiken und die generelle Spielausführung von den hinzufügbaren Spielen und der Visualisierungsebene getrennt sein. Am Ende ist das Ziel, nur die allgemeine Spiellogik und prinzipielle Darstellung zu implementieren. Das Spiel wird daraufhin als Objekt behandelt und hat selbst keinen Kontakt zu dem Code, mit welchem das Spiel gespielt wird.

Eine Datenbank soll die Speicherung der Spielerdaten und deren Statistiken übernehmen.

Erste Umsetzung

Die einzelnen Spiele sind Unterklassen des allgemeinen Spiels (Game). In diesen soll das genaue Verhalten bei Spieleraktionen, die Spielregeln, sowie der Aufbau des Spielfelds definiert sein.

Wird ein Spiel gewählt, übergibt die Spielebibliothek das Spiel an den Spielplatz (Playground). Der Spielplatz besitzt Unterklassen, welche genauer auf die Vorgehensweise bei einem Spiel eingehen. Es wird dabei zwischen Einzelspieler (Singleplayer) und Mehrspieler (Multiplayer) unterschieden.

Um ein Spiel zu Spielen wird ein Benutzerkonto (Account) benötigt unter welchem der Spieler seinen Namen speichern kann.

Hinter diesem befindet sich die Statistik Klasse, welche Unterklassen zu den einzelnen Spielen besitzt. Über diese lassen sich Auswertungen über die bisherigen Spielergebnisse einsehen, um einen Wettkampf sowohl im Mehr- als auch im Einzelspieler möglich zu machen.

Clean Architecture

Domain Code

Die Entitäten in unserer Anwendung bilden deren Kern und haben nach dem Clean Architecture Konzept keine Abhängigkeiten. Dabei handelt es sich um die Klassen, dessen Objekte einzelnen Nutzern direkt zugewiesen werden können, wie Accounts und GameStatistics.

<pre> """ imports """ class Account(): """ global variables """ def __init__(self, player_id, name, password, age, is_admin): self.player_id = player_id self.name = name self.password = password self.age = age self.is_admin = is_admin </pre>	<pre> """ imports """ class GameStatistic(): """ global variables """ def __init__(self, player_id, game_id, wins, losses): self.player_id = player_id self.game_id = game_id self.wins = wins self.losses = losses </pre>
Link: Account.py	Link: GameStatistic.py

Application Code

Der Application Code umfasst einen Großteil der Klassen unserer Anwendung. Dabei gibt es viele Abhängigkeiten innerhalb dieser Ebene, wie beispielsweise die Menüs oder der Playground. Erstere beschreiben Nutzungsmöglichkeiten der Anwendung und erlauben die Durchgabe der Daten, sowie deren Anpassungen, an die Entitäten der Ebene 3. Der MenuManager dient als Verhaltensbeschreibung wie mit den übergebenen Menüs umzugehen ist. Der Playground beschreibt stattdessen den Use-Case eines gestarteten Spiels, wobei die konkrete Ausführung der Spiele auf derselben Ebene delegiert wird.

Link: MenuManager.py	Link: Playground.py
--------------------------------------	-------------------------------------

Adapter

Die Adapterebene 1 ist in dem Spielebibliothek Projekt ziemlich schmal. Da keine aufwändigeren Umformungen durchgeführt werden müssen, besteht diese Schicht ausschließlich aus Interfaces, welche die Schnittstellen zu den Plugins definieren. Es

handelt sich dabei um die Interfaces `GuiBuilder` und `DatabaseAccess`. Diese erheben Ansprüche an die Plugins welche Funktionen diese umzusetzen haben damit der Applications Code nicht abgeändert werden muss, sollte beispielsweise auf ein neues Datenbanksystem oder eine neue Art der Visualisierung umgestellt werden.

Link: GuiBuilder.py	Link: DatabaseAccess.py
-------------------------------------	---

Plugins

Die Spielebibliothek wird visualisiert mit Hilfe von PyGame, einer Bibliothek für Python zur Erstellung meist monolithischen Spielen. Um unabhängig von PyGame zu sein, invitieren wir diese durch die Implementierung des `GuiBuilder` Interfaces. Neben den Methoden zur Erstellung und Interaktion mit dem GUI, beinhaltet das Plugin `PygameBuilder` eine `run` Methode, in der diese mit über das Messaging-System verbundenen Komponenten kommunizieren kann. Eine mittels Docker Container bereitgestellten PostgreSQL Datenbank werden die Nutzerdaten gespeichert. Hierfür gibt es die `psycopg2` Bibliothek. Entsprechend besteht auch hier eine Abhängigkeit die von uns in das `PostgreSQLAdapter` Plugin isoliert haben.

Link: PygameBuilder.py	Link: PosgreSqlAdapter.py
--	---

Main

Die Main agiert als Ersteller und erstmaliger Koordinator der verschiedenen Komponenten. Hierzu besitzt sie Abhängigkeiten über die ganze Anwendung verteilt. Ein möglicher Kritikpunkt besteht darin dass die Main neben der Zusammenstellung auch den Wechsel zwischen dem Themenkomplex Menüs und der Spieleausführung beschreibt. Dieser nicht unkritischer Use-Case könne aus dem Eintrittspunkt (Ebene -1) in eine Klasse der Application Ebene 2 verschoben werden.

UML-Diagramm

Link: Ganzer Programmwurf

Entwurfsmuster

Beobachter

Zur direkten Kommunikation der GuiBuilder Implementierung mit Komponenten, die auf struktureller Ebene eine größere Distanz aufweisen, zu ermöglichen haben wir uns für ein Messaging-System entschieden. Dieses lehnt sich an das Entwurfsmuster des Beobachters an. Die Problemsituation war die Übermittlung von Grafikinformatoren von den Spielen zum PygameBuilder sowie Benutzereingaben zum Spiel, die dazwischenliegenden Koordinationsobjekte hinweg. Der Beobachter wirkte mit dem Publisher-Subscriber-Konzept für diesen Einsatz sinnvoll.

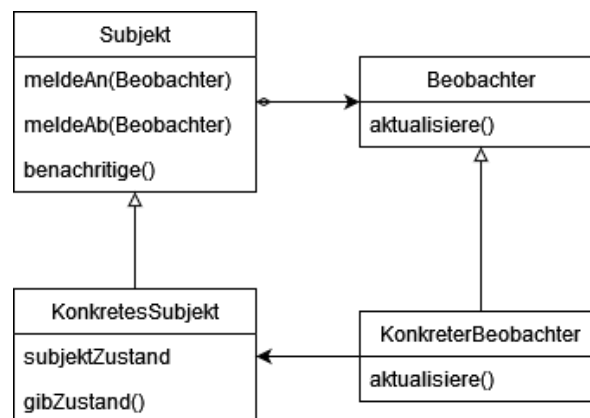


Abbildung 1 - UML-Diagramm Beobachter (Original)

Indem einer unserer Sender einen Receiver von einer Koordinierenden Instanz zugewiesen bekommen hat, kann dieser Nachrichten versenden, welche in einer Liste des Receivers gespeichert werden. Bei den Übertragenen Events handelt es sich um Dictionaries aus welchen beispielsweise die Kategorie ausgelesen werden kann. Anhand dieser Information wird entschieden ob die Nachricht näher behandelt oder verworfen werden soll. Die Nähe zum Beobachter besteht darin, dass die Informationen zwingend an den Receiver weitergegeben werden jedoch dort auf Abruf warten.

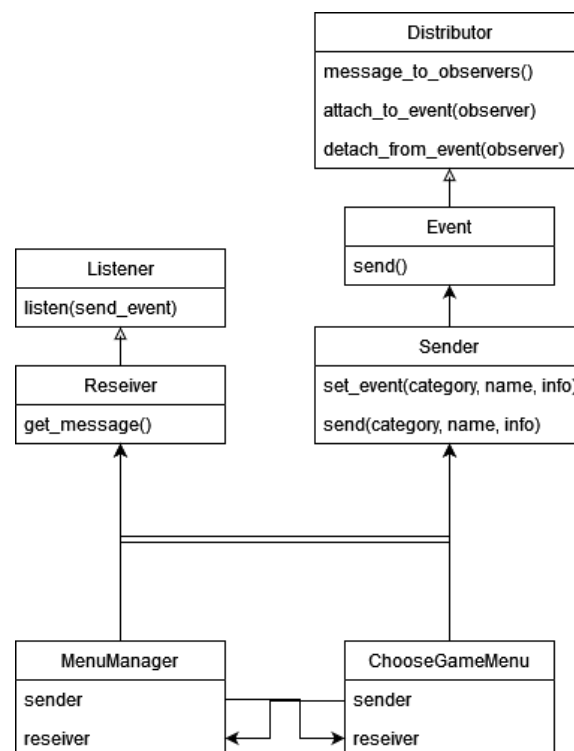


Abbildung 2: UML - Sender-Receiver (Umsetzung)

Link: Reseiver.py	Link: Sender.py	Link: Distributor.py
-----------------------------------	---------------------------------	--------------------------------------

Besucher

Um für das Verhalten unserer Menüs einheitlich zu gestalten verwenden wir ein Verwaltungsmuster. Da Wechsel zwischen den Menüs geschehen, indem diese durch Menü Interaktionen ausgelöst werden, sahen wir erst das Verwalter Muster in unserem Code. Dennoch ist den einzelnen Menüs nicht bekannt, dass sie mit einer koordinierenden Instanz kommunizieren, was mehr auf einen Besucher hinweist.

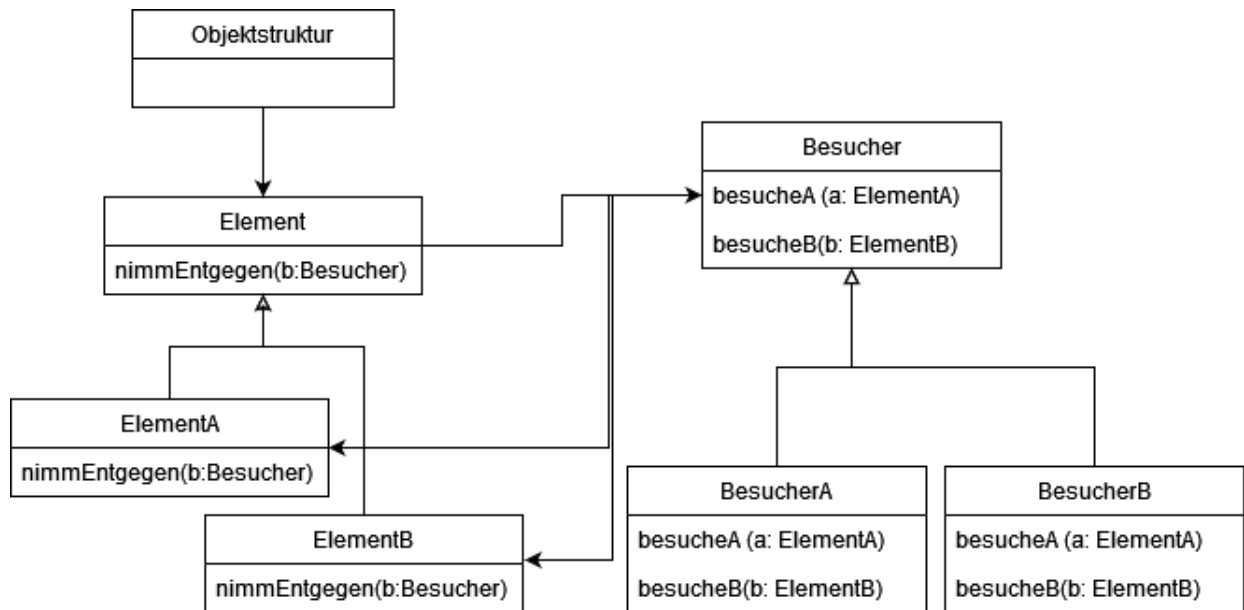


Abbildung 3: UML - Besucher (Original)

Im Falle einer Account Anpassung und dem damit einhergehenden Wechsel vom **ManageAccountMenu** in das **EditAccountMenü**, und zurück, werden jedoch zusätzliche Informationen über den **MenuManager** übertragen. So besteht in Unserem Programmentwurf eine Mischform aus **Besucher** und **Verwalter**, jedoch mit Hang zu ersterem.

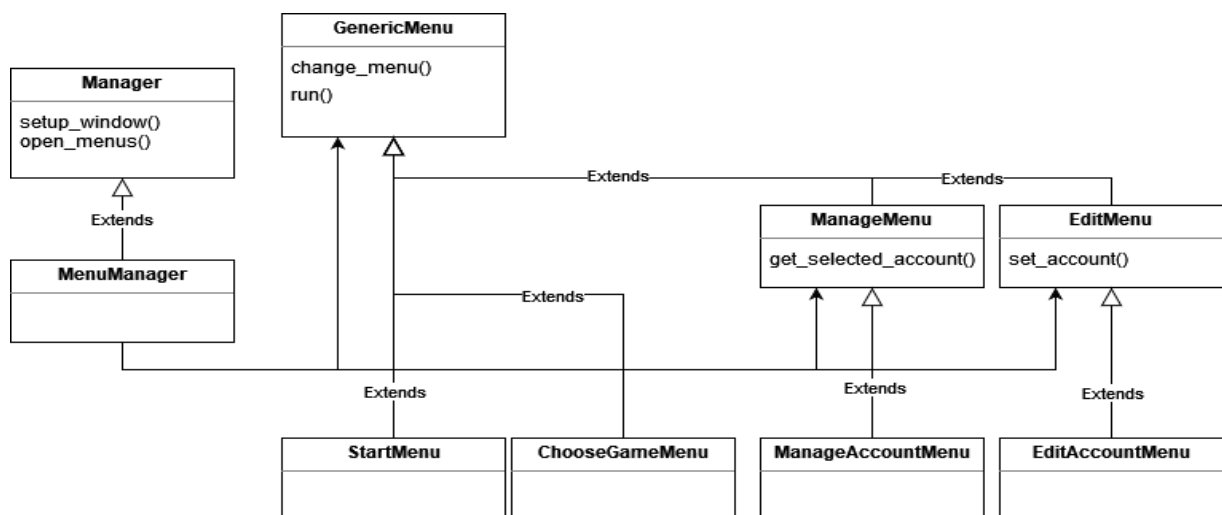


Abbildung 4: UML - Besucher (Umsetzung)

Link: [MenuManager.py](#)Link: [GenericMenu.py](#)

Erbauer

Durch die Isolation der GuiBuilder Implementation wurde indirekt eine Art Erbauer umgesetzt. So erbaut die PygameBuilder Klasse auf Anfrage von außen Objekte zur visuellen Darstellung der Spielebibliothek. Ein Erbauer ist für diesen Einsatzzweck aus dem Grund sinnvoll, da keine Abhängigkeit zu dem schlussendlich erstellten Objekt besteht. So ist Strukturell die Möglichkeit gegeben durch weitere Erbauer die Spielebibliothek zwischen verschiedene Darstellungsverfahren wechseln zu lassen.

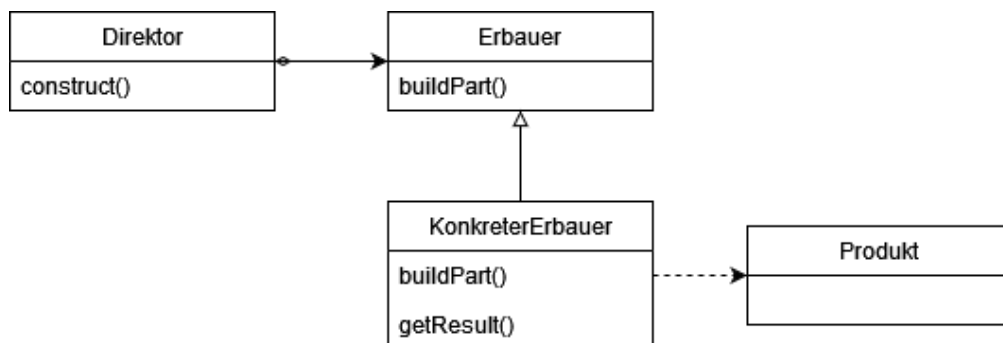


Abbildung 5: UML - Erbauer (Original)

Beispielsweise erstellt der Erbauer das Fenster Objekt durch den Aufruf der `create_window()` Methode, welche in dem GuiBuilder Interface als Schnittstelle festgelegt ist. Dabei ist egal wie der Erbauer das GUI aufbaut und welche Plattform dafür verwendet wird.

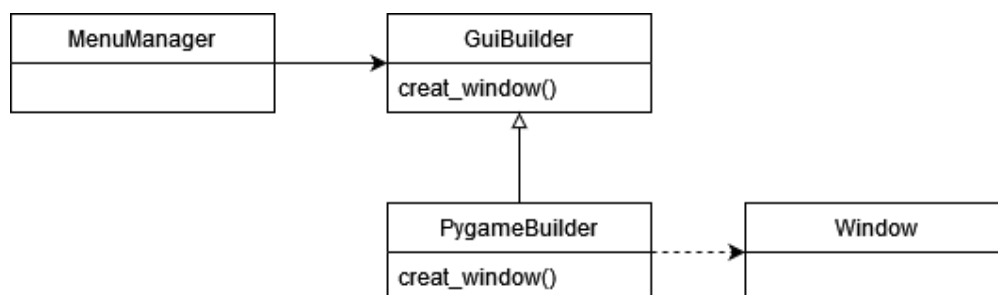


Abbildung 6: UML - Erbauer (Umsetzung)

Link: MenuManager.py	Link: GuiBuilder.py	Link: PygameBuilder.py
--------------------------------------	-------------------------------------	--

Programming Principles

SOLID

Single Responsibility Principle

Zur Demonstration der Einhaltung von Single Responsibility eignen sich die Klassen rund um die Verwaltung der Spielerdaten. So gibt es Klassen die sich ausschließlich um Aufbewahrung und Anpassung der Daten kümmern, wie die Account- oder die GameStatistic Klasse. Die Verwaltung der Objekte dieser Klassen wird jeweils von den dedizierten Verwaltungsklasse, AccountList und Statistics, übernommen, welche deren Änderungen auf die Datenbank widerspiegelt.

Datenklasse	Verwaltungsklasse
<pre> class Account(): """ global variables """ def __init__(self, player_id, name, password, age, is_admin): self.player_id = player_id self.name = name self.password = password self.age = age self.is_admin = is_admin """ functions """ def get_id(self): return self.player_id def set_name(self, new_name): self.name = new_name def get_name(self): return self.name def set_password(self, new_password): self.password = new_password def get_password(self): return self.password def set_age(self, new_age): self.age = new_age def get_age(self): return self.age def set_admin(self, is_admin): self.is_admin = is_admin def get_admin(self): return self.is_admin </pre>	<pre> class AccountList(): """ global variables """ def __init__(self, datamanager: DatabaseAccess): self.datamanager = datamanager self.account = [] """ functions """ # get accounts from database into a local list def get_accounts(self): players = self.datamanager.get_player_table() player_list = [] for player in players: player_list.append(Account(player[0], player[1], player[2], player[3], player[4])) self.account = player_list def add_account(self, username, password, age, is_admin): if len(self.account) <= MAX_NUMBER_OF_ACCOUNTS: self.datamanager.add_account(username, password, age, is_admin) player_id = self.datamanager.last_added_account() self.datamanager.add_account_gamestats(player_id) self.account.append(Account(player_id, username, password, age, is_admin)) else: print("maximum of usable accounts has been reached") def save_account_data(self, account: Account): self.datamanager.update_account(account.player_id, account.name, account.password, account.age, account.is_admin) def delete_account(self, account): self.datamanager.delete_account(account.get_id()) self.get_accounts() </pre>
Link: Account.py	Link: AccountList.py
Link: GameStatistic.py	Link: Statistics.py

Als Antibeispiel lässt sich unser MenuManager heranziehen. Dieser hat zwei Aufgaben, welche getrennt voneinander mehr Stabilität liefern würden. Als Beobachter traversiert in erster Linie die ihm übergebenen Menüs, jedoch fungiert er

auch als Relay zwischen den diesen und dem GUI. Eine Trennung dieser Aufgaben würde es vereinfachen Anpassungen am MenuManager durchzuführen.

Open-Closed Principle

Das Projekt einer Spielebibliothek ist lehnt sich als Konzept an dem Open-Closed-Principle an. Denn dieses bestand darin eine Plattform zu haben, welche statisch ist, jedoch leicht um Spiele mit beliebiger Funktion zu erweitern ist. Beispielhaft ist hierfür der Mehrspieler TicTacToe. Dieser erbt von der Game Oberklasse und überschreibt deren Funktionen. Das Objekt der Multiplayer Klasse verlässt sich auf deren Existenz und koordiniert mit diesen den Grundsätzlichen Ablauf eines Mehrspielers.

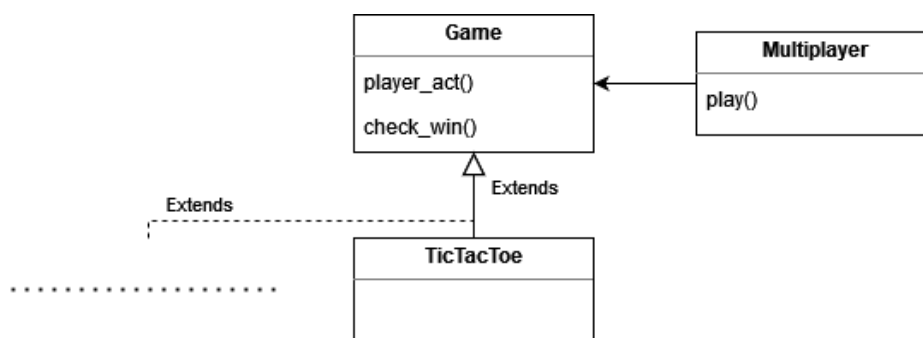
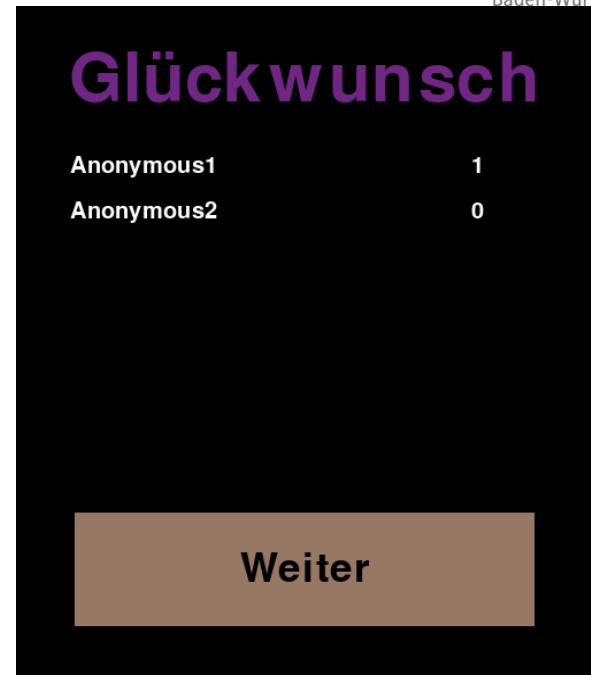
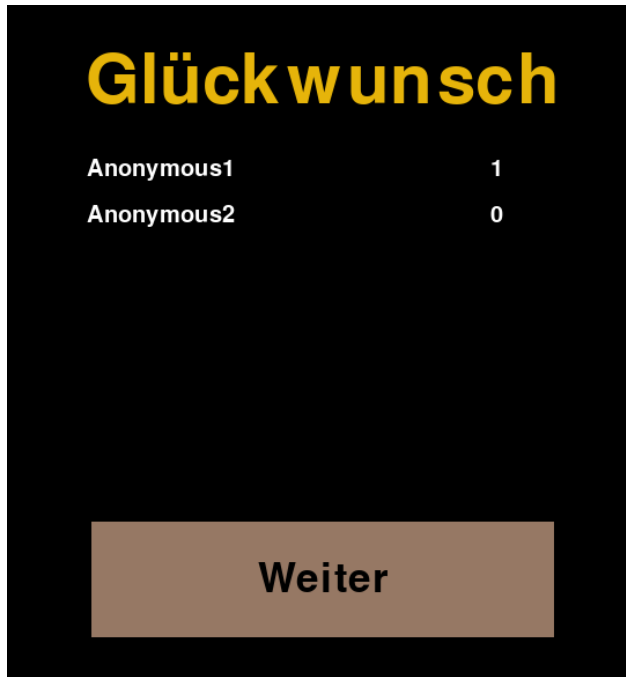


Abbildung 7: UML - Open-Closed-Principle

Link: Multiplayer.py	Link: Game.py	Link: TicTacToe.py
--------------------------------------	-------------------------------	------------------------------------

Liskov Substitution Principle

Die Erfüllung des Liskov Substitutions Principle ist in den WinScreen Klassen zu finden. Der WinScreen hat die Aufgabe „Congratulation“ sowie die Namen und erhaltenen Punkte der Spieler anzuzeigen. Ein Button soll den Nutzer zurück zum Hauptmenü bringen. Der DiscoWinScreen bildet dabei einen Untertypen dieses Menüs. Der DiscoWinScreen erfüllt die selbe Aufgabe. Spezieller lässt er dabei das „Congratulation“ in zufälligen Farben blinken. Nach dem LSP sind die Klassen konvergent und die Funktionalität bleibt uneingeschränkt korrekt.



Link: [WinScreenMenu.py](#)

Link: [DiscoWinScreenMenu.py](#)

Verbesserungswürdig ist die allgemeine Nutzung von Untertypen. Wir verwenden in erster Linie Interfaces, welche die in unserer Anwendung geforderten Funktionen voraussetzt. Die Verwendung einer tieferen Vererbungsstruktur würde zusätzlich einzelnen Code Dopplungen entgegenwirken und hier einen Raum für mehr Beispiele lassen.

Interface Segregation Principle

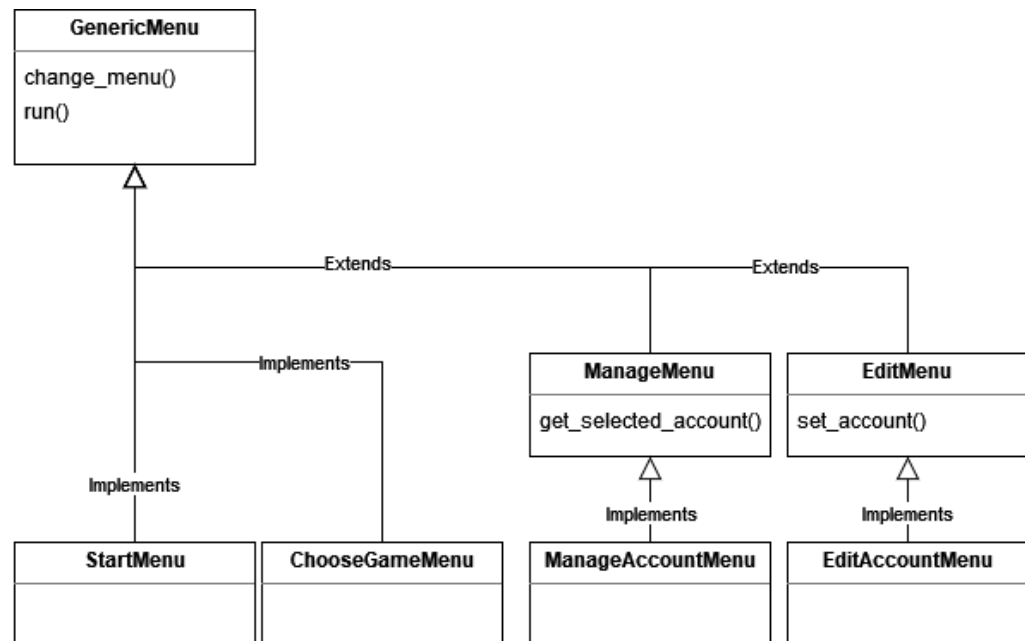


Abbildung 8: UML - Interface Segregation Principle

Der MenuManager erwartet die Übergabe von instanziierten Menüs. Im Durchlaufen einer Schleife werden Funktionen dieser Instanzen aufgerufen, welche durch das GenericMenu Interface vorausgesetzt werden. Das EditAccountMenu und ManageAccountMenu haben dabei eine Sonderrolle, da von diesen zusätzlichen Funktionen verlangt werden. Damit gewährleistet ist, dass die Funktionen ebenso Implementiert werden, erben diese von spezielleren Interfaces.

Python arbeitet nicht wie Java oder C# mit den klassischen Interfaces, sondern erlaubt stattdessen Mehrfachvererbung sowie abstrakte Klassen. Damit ist ein ähnliches Verhalten modellierbar.

Dependency Inversion Principle

Nach dem Dependency Inversion Principle drehen wir die Abhängigkeit von der pygame-Bibliothek um. So wird die Visualisierung unserer Anwendung nicht zu einem Kernelement. Über die Implementierung des GuiBuilder Interfaces schaffen wir eine Schnittstelle zwischen dem PygameBuilder und den inneren Ebenen der Anwendung. Das bringt Stabilität und erlaubt es verschiedene Visualisierungsmethoden anzubinden ohne den Anwendungskern anzupassen.

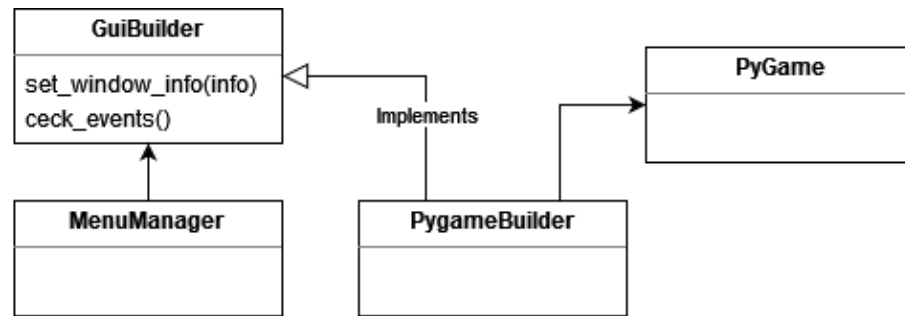


Abbildung 9: UML - Dependency Inversion Principle

Link: MenuManager.py (Kern)	Link: GuiBuilder.py (Adapter)	Link: PygameBuilder.py (Plugin)
---	---	---

GRASP

Low Coupling

Zur Erfüllung des Low Couplings sind einzelne Teile der Anwendung zu in sich abgeschlossenen Paketen gebündelt. Als Beispiel dient hierzu TicTacToe , welches von Game erbt und weitere das GuiBuilder Interface importiert. Zusätzlich importiert es eine Klasse welche nur die in TicTacToe verwendeten Elemente beschreibt, und damit eine Auslagerung der Variablen darstellt. Die Menüs verfahren nach dem gleichen Muster. Beispielhaft implementiert das ChooseGameMenu das GenericMenu-Interface und importiert ausschließlich eine Klasse zur Elementenbeschreibung sowie die MenuActions Klasse, welche Funktionen enthält die alle Menüs nutzen. Die einzelnen Pakete werden daraufhin von der Main koordiniert und zugewiesen.

Link: Main.py	Link: TicTacToe.py	Link: ChooseGameMenu.py
-------------------------------	------------------------------------	---

Die Imports der Klassen Sender und Reseiver wurden in der Beschreibung übergangen, da wir uns für ein Messaging-System entschieden haben. Diese Weise Informationen über weite Strecken in der Anwendung zu transportieren wird damit zu einem zentralen Bestandteil. Nahezu alle Elemente besitzen dazu eine direkte oder indirekte Abhängigkeit, weshalb ein Fehler darin kritische Folgen für das Gesamtsystem hätte.

High Cohesion

Die Account-Klasse hat im Kontext der Spielebibliothek eine hohe Kohäsion. Konkrete Personen haben keine große Relevanz außer diese Spielerkonten zuzuweisen, weshalb diese Daten ebenso gut direkt als Accounts angelegt werden können. Alle verarbeiteten Informationen, wie Nutzernamen, Passwort, Alter und ob der Account über Administrationsrechte verfügt, haben eine direkte kontextuelle Bindung an ein Nutzerkonto. Weiter verfügt die Klasse ausschließlich über die Get- und Set Methoden für die genannten Objektdaten. Ein Negativbeispiel bei dem eine niedrige Kohäsion besteht ist der Klasse Playground. Diese beinhaltet sowohl die Wahl des richtigen Spielmusters für das zu startende Spiel als auch die Aufbauvorbereitung für die Spielerauswahl und die Anzeige der finalen Punkteverteilung. Dies spricht ebenso gegen die Erfüllung des Single Responsibility Principle. Es lässt sich dennoch argumentieren, dass der Playground wiederholbare Spielerfahrung modelliert, worin die Spielerwahl und das Ergebnis beinhaltet sind.

Link: Account.py (hoch)	Link: Playground.py (niedrig)
---	---

DRY

Zur Umsetzung von DRY passen die Beschreibungsklassen der Menüs. Das Wissen um die Struktur und visuelle Vorgaben wird dort unabhängig beschrieben. Klassen, welche die Oberfläche darstellen oder anpassen erhalten nur Verweise auf die Informationsquelle.

Zusätzlich wird im Refactoring auf die Beseitigung des Duplicated Code - Code Smells unter der Berücksichtigung von DRY eingegangen. Ein Beispiel für WET Code findet sich in den Menü-Klassen. Die dort definierten run() Funktionen wiederholen sich in jedem Menü, wobei diese vereinzelt vorausgehende oder nachfolgende Aktionen durchführt. Eine Schwierigkeit, welche das Refactoring z.B. in die MenuActions Klasse verhindert, ist die Verwendung der check_menu_actions() Methode. Diese Methode unterscheidet sich in einem Großteil der Menüs.

Link: Start.py (DRY) Link: StartMenu.py	Link: StartMenu.py (WET)	Link: ManageAccountMenu.py (WET)
--	--	--

Refactoring

Code Smell 1 – High Cyclomatic complexity

Um eine Menüstruktur aufzubauen, mit welcher in und aus Untermenüs navigiert werden kann, haben wir Abfrageschleifen erstellt. Diese Reagieren auf eine Benutzereingabe und öffnen das entsprechende Fenster. In diesem wurde eine weitere Schleife betreten welche der Schleife des überliegenden Menüs gleicht. Da die Auswahlmöglichkeiten eines Menüs diesem bei jeweils vor dem Öffnen mitgegeben wurden, hatte dieses sich selbst sowie seine Untermenüs zu verwalten. Diese Kopplung macht die gesamte Struktur schwerfälliger und kompliziert zu warten, da ein Menü dass viele Untermenüs hatte weit gestreute Abhängigkeiten enthält. Der Code Smell einer hohen zyklomatischen Komplexität ist ebenfalls sehr auffällig, da jede traversierte Schleife eine Komplexität von 6 bis 13 aufweist.

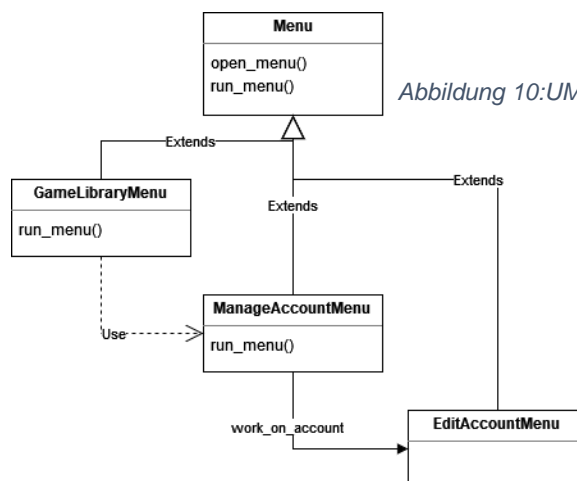


Abbildung 10:UML - Code Smell 1

```

def run_menu(self):
    print("running main menu...")
    main_menu_active = True
    action = "waiting for action"
    while main_menu_active:
        self.gui.update_window()
        action = self.gui.check_events(self.menu_interactables)
        if action != "no action":
            if action == "quit":
                self.gui.terminate_window()
            if action == "choose_game_button":
                self.gui.clear_window()
                self.menu_interactables = []
                self.choose_game.open_menu()
                self.choose_game.run_menu()
                self.open_menu()
                sleep(Menu.blocking_wait_seconds)
            if action == "account_button":

```

Link: [GameLibrary.py](#)

```

def run_menu(self):
    sleep(Menu.blocking_wait_seconds)
    choose_menu_active = True
    action = "waiting for action"
    while choose_menu_active:
        self.get_account_list_on_screen()
        self.gui.set_element_styles()
        self.gui.update_window()
        action = self.gui.check_events(self.menu_interactables)
        if action != "no action":
            if action == "quit":
                self.gui.terminate_window()
            if action == "account_1_name":
                self.edit_account(1+self.anonyme_user_count)
            if action == "account_1_delete":
                self.delete_account(1+self.anonyme_user_count)
            if action == "account_2_name":

```

Link: [ManageAccountMenu.py](#)

Optimierung

Die Optimierung wurde durch ein Umdenken der Menükommunikation bewerkstelligt. Hierbei sind wir zu einem Messaging-System umgestiegen, mit dem MenuManager als Besucher der Menüs. Die Interaktionen mit einem Menü werden als Nachrichten der GUI-Instanz über den MenuManager als Relay direkt an die einzelnen Menüs weitergeleitet. Bei einem Menüwechsel wird diese Aufforderung an den MenuManager weitergegeben welcher diese Verarbeitet, die Messaging-Verbindung mit dem verlassenen Menü trennt und eine Verbindung mit dem neuen Menü aufbaut. So sind die einzelnen Menüs getrennt voneinander und laufen in eigenen Abschnitten ohne Abhängigkeiten zueinander. Der Verlauf des MenuManagers bleibt bei dem Durchlauf der Menüs weitestgehend unverändert. Mit einer Komplexität von 3 ist die der Durchlauf eines Untermenüs immer noch verbesserungsfähig, dennoch ist eine erkennbare Verbesserung festzustellen.

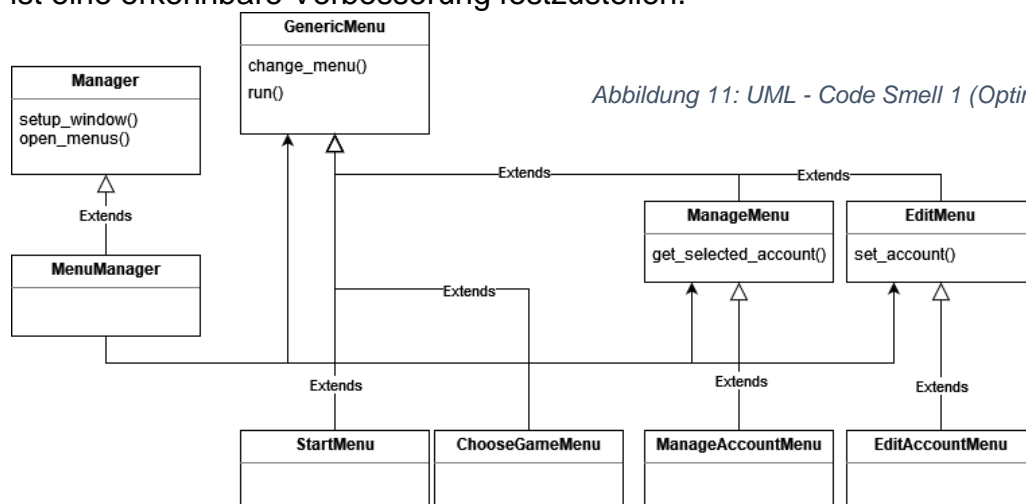


Abbildung 11: UML - Code Smell 1 (Optimierung)

```

def open_menus(self):
    print("open manager")
    self.clear_window()
    active_menu = self.next_menu
    while self.next_menu != "no menu":
        try:
            self.sender.add_listener(active_menu.receiver)
            active_menu.sender.add_listener(self.receiver)
            active_menu.change_menu()
            active_menu.receiver.empty_message_queue()

            if not self.should_come_back_to_menu():
                print("closing manager")
                return

            active_menu.run()
        except:
            "no menu is set on menu start"
        try:
            self.sender.remove_listener(active_menu.receiver)
            active_menu.sender.remove_listener(self.receiver)
        except:
            "no menu is set on menu close"
            active_menu = self.next_menu

    self.set_next_menu("start_menu")
    print("closing manager")
  
```

Link: [MenuManager.py](#)

Link: [StartMenu.py \(GameLibrary\)](#)

Link: [ManageAccountMenu.py](#)

(Quelle: Thomas J. McCabe; Structured Testing; IEEE Computer Society Press, 1983)

Code Smell 2 – Long Method

Im Playground gibt es eine Methode, welche 30 Zeilen umfasst. Die Methode `play()` beschreibt wie das Ausführen eines Spiels in den in den größten Zügen abläuft. Dabei gibt es jedoch einen Unterschied zwischen Einzelspieler- und Mehrspieler-Spielen. Abhängig von der benötigten Spieleranzahl werden diese aufgefordert einen Account auszuwählen.

```
def play(self, game_id):
    game = self.game_list[game_id]
    play_pattern = "no pattern set"
    if game.get_player_count() > 1:
        self.m_player.set_game(game)
        play_pattern = self.m_player
        self.connect_to_gui(play_pattern)
        for user in range(game.player_count):
            player = self.choose_player()
            if player == "not set":
                return
            play_pattern.add_player(player)
    else:
        self.s_player.set_game(game)
        play_pattern = self.s_player
        self.connect_to_gui(play_pattern)
        player = self.choose_player()
        if player == "not set":
            return
        play_pattern.add_player(player)

    play_pattern.receiver.empty_message_queue()
    play_pattern.play()
    self.clean_up(play_pattern)
    self.remove_from_gui(play_pattern)
    if play_pattern.get_interrupt_exit():
        return
    self.save_results(play_pattern)
    self.show_results(play_pattern)
    play_pattern.reset_game()
```

Abbildung 12: UML - Code Smell 2

Optimierung

Um die Funktion zur besseren Lesbarkeit zu verkleinern, haben wir die große Funktion in mehrere kleine zerteilt. Die Vereinzelung hilft dabei sowohl gegen eine gewisse Code Wiederholung als auch zur Abstraktion der `play`-Methode. Diese beinhaltet nun nur sprechende Funktionsname, welche den Spielablauf beschreiben. Die darin verwendeten Methoden übernehmen das wiederholende Verhalten und die Aktivierung von Untermenüs. Die Länge der Methode ist dabei von 30 auf 13 Zeilen zurückgegangen.

```
def play(self, game_id):
    game = self.game_list[game_id]
    play_pattern = self.choose_play_pattern(game)
    if play_pattern == "no pattern set":
        return
    play_pattern.receiver.empty_message_queue()
    play_pattern.play()
    self.clean_up(play_pattern)
    self.remove_from_gui(play_pattern)
    if play_pattern.was_exited_by_interrupt():
        return
    self.save_results(play_pattern)
    self.show_results(play_pattern)
    play_pattern.reset_game()
```

(Quelle: <https://refactoring.guru/smells/long-method>)

```
def choose_play_pattern(self, game):
    play_pattern = "no pattern set"
    if game.get_player_count() > 1:
        self.m_player.set_game(game)
        play_pattern = self.m_player
        self.connect_to_gui(play_pattern)
        for user in range(game.player_count):
            if not self.add_player_to_pattern(play_pattern):
                return "no pattern set"
    else:
        self.s_player.set_game(game)
        play_pattern = self.s_player
        self.connect_to_gui(play_pattern)
        if not self.add_player_to_pattern(play_pattern):
            return "no pattern set"
    return play_pattern
```

Code Smell 3 – Duplicated Code

Es gibt Funktionen, welche in jedem Menü vorkommen, wobei sich diese nur in den verwendeten Variablen unterscheiden. Dazu gehört der Aufbau des Menüs, durch Senden der entsprechenden Elemente über das Relay, sowie die Prüfung der Nutzerinteraktionen. Jedes Menü muss für dessen Elemente testen ob eine Interaktion stattfand und eine Reaktion ausgeführt werden muss.

Link 1: choose_game_menu()	Link 2: manage_account_menu()
--	---

Optimierung

Zu Codeeinsparung haben wir die MenuActions-Klasse angelegt welche diese Funktionen implementiert. Die Menüs übergeben der benötigten Funktion einfach die zu berücksichtigen Parameter. Hierbei muss MenuActions nicht instanziiert werden, da dieses selbst keine Werte speichert.

Link: 1: choose_game_menu()	Link: 2: manage_account_menu()	Link: menu_actions.py
---	--	---------------------------------------

(Quelle: <https://refactoring.guru/smells/duplicate-code>)

Code Smell 4 – Shotgun Surgery

Für eine unabhängige Beschreibung der GUI-Darstellung haben wir die Elemente in eigene Klassen ausgelagert. Jedes darzustellende Oberflächenobjekt besitzt eine eigenes Dictionary die genau beschreibt wie dieses aussehen soll. Da die Menüs jedoch ein geordnetes Erscheinungsbild haben sollen, müssen die Verhältnisse der Objekte zueinander übereinstimmen. Um eine Veränderung in der Positionierung zu erzielen, müssen somit immer alle Objekte angepasst werden, damit dieses Verhältnis gewahrt bleibt.

Link: ChooseGameSpecifics.py	Link: EditAccountSpecifics.py
--	---

Optimierung

Wir haben die Lesbarkeit optimiert, indem wir explizite Zahlen (magic numbers) durch Literale ersetzt haben. Ebenso ist es nun einfacher Änderungen an Button-Reihen vorzunehmen, da mit Hilfe der Literalen das Verhältnis aufrechterhalten wird. Diese dienen nun als zentrale Stellschrauben zur Darstellungsanpassung, während Feinjustierungen in den Dictionaries immer noch möglich sind.

Link: new-ChooseGameSpecifcy.py	Link: new-EditAccountSpecifics.py
---	---

(Quelle: <https://refactoring.guru/smells/shotgun-surgery> , <https://refactoring.guru/replace-magic-number-with-symbolic-constant>)

Unit Test

In unserer Anwendung wurden die Funktionalitäten durch Unit Tests sichergestellt. Insgesamt haben wir 14 Testfälle implementiert.

Klasse Account

Die folgenden Unittests prüfen grundlegende Funktionen der „Account“-Klasse. Es werden folgende Methoden getestet:

1. *test_get_values()*: In diesem Testfall wird die Funktion *get_values()* getestet. Dazu werden einem Account Werte zugewiesen, wovon dann die ID abgefragt und verglichen wird.
2. *test_set_age()*: Hier wird die Funktion *set_age()* getestet. Dabei wird einem Account auch Werte zugewiesen. Anschließend wird das Alter geändert und geprüft, ob die Änderung korrekt war.
3. *test_set_password()*: Bei diesem Test wird, durch gleiches Verfahren wie bei *test_set_age()*, die Funktion *set_password()* geprüft.

Generell prüfen diese Testfälle, ob Werte korrekt ausgegeben und geändert werden können. In Folgender Abbildung ist der Testfall *test_get_values()* zu sehen:

```
def test_get_values(self):  
    #arrange  
    account = Account(2, "name", "password", 15, 0)  
    #act  
    id = account.get_id()  
    #assert  
    self.assertEqual(id, 2)
```

Link: [test_Account.py](#)

Klasse AccountList

Die nachstehenden Testfälle prüfen grundlegende Funktionen der „AccountList“-Klasse. Es werden folgende Methoden getestet:

1. *test_get_accounts()*: In diesem Testfall wird die Funktion *get_accounts()* getestet. Da bei dieser Funktion eine Datenbankabfrage stattfindet, wird eine Instanz der Klasse „DatabaseAccess“ erstellt und als Mock-Objekt definiert. Dem Mock-Objekt wird dann eine Instanz der Klasse „AccountList“ übergeben. Danach wird ein Account mit Parametern definiert, welche anschließend durch die Funktion *get_Accounts()* ausgegeben werden und verglichen werden.

2. `test_add_accounts()`: Die Funktion `add_accounts()` wird mit demselben Verfahren wie oben beschrieben getestet.
3. `test_save_account_data()`: Auch zum Testen der Funktion `save_account_data()` wird ein Mock-Objekt verwendet.

Die Tests prüfen, ob Werte korrekt ausgegeben, aktualisiert und geändert werden können. Dazu wird eine Mock-Instanz der Datenbank erstellt, um die Tests unabhängig von einer tatsächlichen Datenbank durchzuführen. In Folgender Abbildung ist der Testfall `test_get_accounts()` zu sehen:

```
def test_get_accounts(self):  
    #arrange  
    self.mock_datamanager = Mock(spec=DatabaseAccess)  
    self.account_list = AccountList(self.mock_datamanager)  
    self.mock_datamanager.get_player_table.return_value = [  
        (1, 'user1', 'pass1', 25, False),  
        (2, 'user2', 'pass2', 30, True)  
    ]  
    #act  
    self.account_list.get_accounts()  
    #assert  
    self.assertEqual(len(self.account_list.account), 2)  
    self.assertEqual(self.account_list.account[0].player_id, 1)  
    self.assertEqual(self.account_list.account[0].name, 'user1')  
    self.assertEqual(self.account_list.account[1].player_id, 2)  
    self.assertEqual(self.account_list.account[1].name, 'user2')
```

Link: [test_AccountList.py](#)

Klasse `PostgresSqlAdapter`

Diese Unittests testen verschiedene Funktionen der Klasse „PostgreSqlAdapters“.

Folgende Methoden werden dabei berücksichtigt:

1. `test_add_accounts()`: In diesem Testfall wird die Funktion `add_accounts()` getestet. Dazu wird das „`@patch`“-Dekorator verwendet, um die Methode `get_connection` zu mocken. Der Parameter `mock_get_connection` wird dabei automatisch erstellt. Für die Datenbankverbindung wird ein Mock-Objekt erstellt. Dann wird ein Account mit entsprechenden Parametern hinzugefügt. Anschließend wird geprüft, ob der SQL-String für die Datenbank richtig erstellt wird.
2. `test_add_accounts_gamestats()`: Die Funktion `add_accounts_gamestats()` wird nach dem gleichen Prinzip getestet.

3. `test_last_added_account()`: Auch bei der Funktion `last_added_accounts()`, wird geprüft, ob der richtige SQL-String erstellt wird.
4. `test_update_account()`: Für die Funktion `update_accounts()` wird ebenfalls dasselbe Testverfahren angewendet.

Die Unittests decken alle wichtigen Funktionen des Adapters ab. Jeder Test definiert eine klare Vorbedingung, führt eine Aktion durch und überprüft dann das erwartete Ergebnis. Für diese Tests ist eine Mock-Instanz der Datenbank notwendig. In Folgender Abbildung ist der Testfall `test_add_accounts()` zu sehen:

```
@patch('adapter.PostgreSQLAdapter.get_connection')
def test_add_account(self, mock_get_connection):
    # arrange
    mock_conn = MagicMock()
    mock_cur = MagicMock()
    mock_get_connection.return_value = (mock_conn, mock_cur)

    # act
    self.adapter.add_account('test_user', 'test_password', 18, False)

    # assert
    mock_conn.commit.assert_called_once()
    mock_cur.execute.assert_called_once_with(
        "INSERT INTO player(username, password, age, is_admin) VALUES(%s, %s, %s, %s);",
        ('test_user', 'test_password', 18, False)
    )
```

Link: [GameStatistic.py](#)

Klasse Statistics

Die folgenden Unittests sind für die „Statistics“-Klasse, die Spielstatistiken von Spielern verwaltet. In einer `setUp()`-Methode werden einige Testdaten initialisiert, einschließlich einer Mock-Instanz von `DatabaseAccess`, um die Datenbankabfragen zu simulieren. Die Testklasse enthält drei Testmethoden:

1. `test_get_statistics()`: Prüft, ob die Methode `get_statistics()` korrekte Ergebnisse liefert. Dies wird Anhand der Länge des Objektes geprüft.
2. `test_increase_wins()`: Testet, ob die Methode `increase_wins()` bei einem Sieg die Anzahl der Siege korrekt addiert.
3. `test_increase_losses()`: Zum Testen der Funktion `increase_losses()` wird dasselbe Verfahren wie bei der Funktion `test_increase_wins()` angewendet.

Link: [test_Statistic.py](#)

Klasse TicTacToe

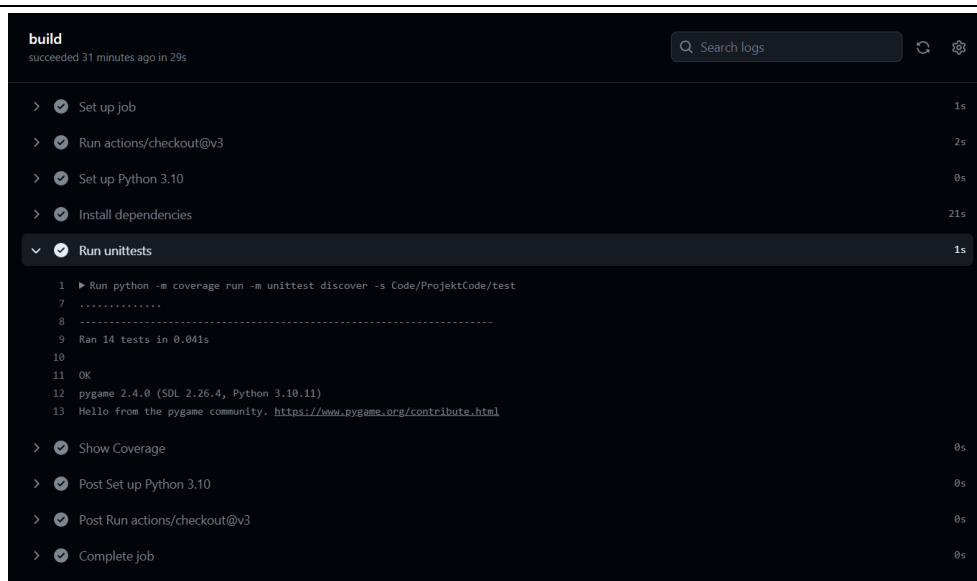
Ebenfalls wurden für die „TicTacToe“-Klasse ein UnitTest implementiert. Die Testklasse enthält folgende Testmethode:

1. `test_change_symbols`: Diese Methode testet, ob die Methode `change_symbols` der TicTacToe-Klasse die Symbole der aktiven Spieler wechselt, indem sie zunächst das Symbol des aktiven Spielers auf "O" setzt und dann erwartet, dass es auf "X" wechselt, und dann erwartet, dass es wieder auf "O" wechselt.

Link: [test_TicTacToe.py](#)

ATRIP-RegelnAutomatic

Unsere Tests laufen eigenständig durch und benötigen keine Interaktion vom Benutzer, wie z. B. eine Eingabe über ein Dialogfenster. Außerdem wird das Ergebnis automatisch geprüft, wobei ausgegeben wird welche Tests erfolgreich waren oder nicht bestanden haben. Zum einen werden alle vorhandenen Tests mit der Eingabe „`python -m coverage run -m unittest`“ in der Kommandozeile durchgeführt. Zum anderen werden alle Tests bei jedem „push“ automatisch in Github durchgeführt. Das haben wir durch den Continuous Integration-Service von Github Actions realisiert.



```
build
succeeded 31 minutes ago in 29s

> Set up job 1s
> Run actions/checkout@v3 2s
> Set up Python 3.10 0s
> Install dependencies 21s
> Run unittests 1s
  1 ▶ Run python -m coverage run -m unittest discover -s Code/ProjektCode/test
  7 .....
  8 .....
  9 Ran 14 tests in 0.041s
 10
 11 OK
 12 pygame 2.4.0 (SDL 2.26.4, Python 3.10.11)
 13 Hello from the pygame community. https://www.pygame.org/contribute.html
> Show Coverage 0s
> Post Set up Python 3.10 0s
> Post Run actions/checkout@v3 0s
> Complete job 0s
```

Link: [automated-tests.yml](#)

Thorough

Die Testfälle wurden gründlich ausgeführt, um sicherzustellen, dass alle getesteten Funktionen fehlerfrei laufen. Bei den Testfällen wurde darauf geachtet, dass möglichst alle relevanten Funktionen getestet werden. Da alle Testfälle reibungslos durchlaufen, mussten wir keine separaten Testfälle, um erneutes Auftreten zu verhindern, implementieren.

Repeatable

Alle Tests können jederzeit, auch automatisch ausgeführt werden. Dabei wird zu jeder Zeit dasselbe Ergebnis geliefert. Die Ausführung ist unabhängig von bestimmten Szenarien, wie bedingte Variablen oder Benutzereingaben. Da wir keine Dateisystemzugriffe verwenden, mussten wir die Plattformabhängigkeit nicht beachten.

Independent

Da in unserem Projekt einiges von der Datenbank abhängig ist, standen wir vor der Herausforderung, die Tests unabhängig von der Datenbank auszuführen. Um dieses Problem zu lösen, haben wir für die betroffenen Klassen Mock-Instanzen eingesetzt, um die Datenbankabfrage zu simulieren. Dabei handelt es sich um die Klassen „AccountList“ und „PostgreSqlAdapter“. Die Klasse „AccountList“ ist zudem abhängig von der Klasse „Accounts“. Auch diese Abhängigkeit haben wir bei den Tests mit dem Einsatz einer Mock-Instanz gelöst. Somit laufen alle Tests unabhängig voneinander. Alle Testfälle können in beliebiger Zusammenstellung und Reihenfolge ausgeführt werden.

Professional

Bei der Entwicklung unserer Testfälle, haben wir darauf geachtet, dass diese so leicht wie möglich zu verstehen sind. Jede Test-Methode testet nur eine Funktion, wodurch die Tests sehr übersichtlich sind. Trotz allem haben wir bei den Testfällen die wichtigsten Funktionen priorisiert, wie zum Beispiel Funktionen, welche Datenbankabfragen enthalten.

Code-Coverage

Für die Code-Coverage-Analyse wurde die Erweiterung namens "Python Test Explorer for Visual Studio Code" in Visual Studio Code angewendet, die es

ermöglicht, Unittests innerhalb von Visual Studio Code auszuführen. Die Erweiterung bietet eine grafische Benutzeroberfläche zum Ausführen von Unittests und zum Anzeigen der Testergebnisse sowie der Code-Abdeckung. Die Ergebnisse können als HTML- oder JSON-Datei exportiert werden.

Um die Code-Abdeckung zu messen wird das Modul „coverage“ angewendet. Dieses Modul wird verwendet, um zu messen, wie viel Prozent des Quellcodes durch die Unittests abgedeckt werden. Wenn die Tests den Code nicht vollständig abdecken, können Lücken in der Testabdeckung identifiziert werden und zusätzliche Tests geschrieben werden, um diese Lücken zu schließen. Folgendes Ergebnis haben wir erhalten:

Link: htmlcov.rar

Modul	Stmt.	Miss.	Exc.	Cov.
adapter__init__.py	8	0	0	100%
adapter\interfaces__init__.py	6	0	0	100%
adapter\interfaces\database_interface.py	43	14	0	67%
adapter\interfaces\edit_menu_interface.py	9	2	0	78%
adapter\interfaces\generic_menu_interface.py	8	2	0	75%
adapter\interfaces\gui_builder_interface.py	34	11	0	68%
adapter\interfaces\manage_menu_interface.py	9	2	0	78%
adapter\interfaces\manager_interface.py	14	4	0	71%
adapter\pygame_builder.py	85	66	0	22%
adapter\to_from_postgresql.py	181	121	0	33%
communication__init__.py	5	0	0	100%
communication\messaging\distributer.py	16	9	0	44%
communication\messaging\event_message.py	28	7	0	75%
communication\messaging\listener_interface.py	5	1	0	80%
communication\reseiver.py	18	7	0	61%
communication\sender.py	16	7	0	56%
core_files__init__.py	9	0	0	100%
core_files\account.py	26	6	0	77%
core_files\account_list.py	26	3	0	88%
core_files\game.py	30	11	0	63%
core_files\game_list.py	13	5	0	62%
core_files\game_statistic.py	21	1	0	95%
core_files\multiplayer.py	91	72	0	21%
core_files\playground.py	95	74	0	22%
core_files\singleplayer.py	62	46	0	26%
core_files\statistics.py	27	2	0	93%

games__init__.py	4	0	0	100%
games\graphics__init__.py	2	0	0	100%
games\graphics\space_invaders\space_invaders_specifications.py	13	1	0	92%
games\graphics\tic_tac_toe\tic_tac_toe_specifications.py	11	1	0	91%
games\space_invaders.py	9	3	0	67%
games\tic_tac_toe.py	117	92	0	21%
gui__init__.py	16	0	0	100%
gui\account_section_menu.py	55	42	0	24%
gui\choose_game_menu.py	35	24	0	31%
gui\edit_account_menu.py	118	91	0	23%
gui\gui_specifications__init__.py	7	0	0	100%
gui\gui_specifications\account_selection.py	18	1	0	94%
gui\gui_specifications\choose_game.py	13	1	0	92%
gui\gui_specifications\edit_account.py	25	1	0	96%
gui\gui_specifications\manage_account.py	30	1	0	97%
gui\gui_specifications\start.py	17	1	0	94%
gui\gui_specifications\win_screen.py	34	1	0	97%
gui\gui_specifications>window.py	5	1	0	80%
gui\manage_account_menu.py	79	58	0	27%
gui\menu_actions.py	15	8	0	47%
gui\menu_manager.py	81	68	0	16%
gui\start_menu.py	31	21	0	32%
gui\win_screen_menu.py	48	36	0	25%
gui\win_screen_menu_disco.py	32	23	0	28%
test_Account.py	19	0	0	100%
test_Accountlist.py	39	0	0	100%
test_PostgreSqlAdapter.py	44	0	0	100%
test_Statistic.py	26	0	0	100%
test_TicTacToe.py	13	0	0	100%
Total	1841	948	0	49%

Das Ergebnis ist nicht sehr Aussagekräftig, da zum Beispiel alle „Funktions-Header“ und Imports als getestet gelten. Aus diesem Grund werden beispielsweise alle „__init__.py“-Dateien als 100% abgedeckt bewertet. Somit ist die Codeabdeckung mit 49% im Verhältnis zu 14 Testfällen sehr hoch. Mit diesem Ergebnis haben wir nicht gerechnet.

Einsatz von Mocks

In den UnitTests der Klassen „AccountList“, „Statistic“ und „PostgreSqlAdapter“ werden Mocks eingesetzt, um die Abhängigkeiten von der Datenbankanbindung zu reduzieren und den Code besser zu testen.

Ein „Mock“ ist ein Dummy-Objekt, das die gleiche Schnittstelle wie ein echtes Objekt hat, aber vordefinierte Werte oder Verhaltensweisen liefert. Im Fall der Klassen „Statistic“ und „AccountsList“ werden die Methoden von „DatabaseAccess“ durch ein Mock-Objekt ersetzt, das die erwarteten Rückgabewerte für die getesteten Methoden zurückgibt. Beim Testen der Klasse „AccountList“ wird zudem die Klasse „Account“ durch ein „Mock“-Objekt ersetzt. Für die Klasse „PostgreSqlAdapter“ wird das Mocking-Framework „MagicMock“ in Verbindung mit der „patch“-Funktion verwendet.

Das Mocking ermöglicht es, das Verhalten der abhängigen Objekte zu kontrollieren und die Interaktionen mit ihnen zu überprüfen, ohne tatsächlich auf die reale Datenbank zugreifen zu müssen. Somit können die Tests unabhängig von der tatsächlichen Datenbankbindung ausgeführt werden, was die Wartbarkeit und Lesbarkeit des Codes erhöht sowie die Testabdeckung verbessert.