

Programmmentwurf – 3er Gruppe

Sudoku

Name: Fien, Alisa

Matrikelnummer: (siehe Mail)

Name: Fuchs, Marco

Matrikelnummer: (siehe Mail)

Name: Sisic, Marcel

Matrikelnummer: (siehe Mail)

Abgabedatum: 31.05.2023

Allgemeine Anmerkungen:

- *Gesamt-Punktzahl: 60P (zum Bestehen mit 4,0 werden 30P benötigt)*
- *die Aufgabenbeschreibung (der blaue Text) und die mögliche Punktzahl muss im Dokument erhalten bleiben*
- *es darf nicht auf andere Kapitel als alleiniger Leistungsnachweis verwiesen werden (z.B. in der Form “XY wurde schon in Kapitel 2 behandelt, daher hier keine Ausführung”)*
- *alles muss in UTF-8 codiert sein (Text und Code)*
- *das Dokument muss als PDF abgegeben werden*
- *es gibt keine mündlichen Nebenabreden / Ausnahmen – alles muss so bearbeitet werden, wie es schriftlich gefordert ist*
- *alles muss ins Repository (Code, Ausarbeitung und alles was damit zusammenhängt)*
- *die Beispiele sollten, wenn möglich vom aktuellen Stand genommen werden*
 - *finden sich dort keine entsprechenden Beispiele, dürfen auch ältere Commits unter Verweis auf den Commit verwendet werden*
 - *Ausnahme: beim Kapitel “Refactoring” darf von vorne herein aus allen Ständen frei gewählt werden (mit Verweis auf den entsprechenden Commit)*
- *falls verlangte Negativ-Beispiele nicht vorhanden sind, müssen entsprechend mehr Positiv-Beispiele gebracht werden*
 - ***Achtung: werden im Code entsprechende Negativ-Beispiele gefunden, gibt es keine Punkte für die zusätzlichen Positiv-Beispiele sondern 0,5P Abzug für das fehlende Negativ-Beispiel***
 - *Beispiel*
 - *“Nennen Sie jeweils eine Klasse, die das SRP einhält bzw. verletzt.” (2P)*
 - *Antwort: Es gibt keine Klasse, die SRP verletzt, daher hier 2 Klassen, die SRP einhalten: [Klasse 1], [Klasse 2]*
 - *Bewertung: falls im Code tatsächlich keine Klasse das SRP verletzt: 2P ODER falls im Code mind. eine Klasse SRP verletzt: 1P*
- *verlangte Positiv-Beispiele müssen gebracht werden – im Zweifel müssen sie extra für die Lösung der Aufgabe implementiert werden*
- *Code-Beispiel = Code in das Dokument kopieren (inkl. Syntax-Highlighting)*
- *falls Bezug auf den Code genommen wird: entsprechende Code-Teile in das Dokument kopieren (inkl. Syntax-Highlighting)*

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

Kapitel 1: Einführung (4P)

Übersicht über die Applikation (1P)

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

Die Anwendung CLI-Sudoku ist ein Kommandozeilenbasiertes Sudokuspiel. Nach dem Start der Anwendung kann der Nutzer / die Nutzerin sich Registrieren beziehungsweise Anmelden. Hierzu sind ein Nutzernamen und Passwort notwendig. Nach erfolgreicher Anmeldung können die Spieler im Auswahlmenü zwischen verschiedenen Optionen wählen. Dabei können sie beispielsweise ein neues Spiel spielen, gespeicherte Sudokus weiterspielen, das Leaderboard einsehen oder ihre Einstellungen anpassen. In den Einstellungen können Hinweise aktiviert oder deaktiviert werden. Wird ein neues Spiel gestartet, so kann der Spieler:in zwischen drei Schwierigkeitsstufen auswählen. Anschließend wird das Sudoku auf der Konsole ausgegeben und es kann gespielt werden. Es gibt die Optionen ein spezifisches Feld zu füllen, zu leeren, das Sudoku zum momentanen Stand zu speichern oder das Spiel zu beenden. Die dafür nötigen Befehle werden immer mit ausgegeben. Bei Falscheingabe bekommen die Anwender:in passende Rückmeldung. Sind die Hinweise in den Einstellungen aktiviert, können spezifische Felder gelöst oder das momentan gelöste Sudoku auf Richtigkeit geprüft werden.

Durch die Anwendung betreten die Sudokus, welche häufig in analoger Form häufig gespielt werden die digitale Welt. Des Weiteren werden immer neue Sudokus generiert, welches ein abwechslungsreiches Spielerlebnis bietet.

Starten der Applikation (1P)

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

Die Applikation ist in der Programmiersprache Java geschrieben. Für die Kompilierung und Ausführung muss die JDK Version 17 installiert sein. Die Anwendung kann dann mit einer gängigen IDE wie beispielsweise IntelliJ gestartet werden. Hierzu die main()-Methode in der Klasse Main.java ausführen.

Soll die Anwendung außerhalb einer IDE gestartet werden ist eine Dockerfile mitgeliefert, welche die den Bau und Ausführung der Applikation vereinfacht. Hierfür muss Docker auf dem System installiert sein. Um die Applikation in einem Dockercontainer zu starten müssen folgende Befehle ausgeführt werden:

1. `“docker build -t sudoku-image .”` (Befehl im Projekordner ausführen – dort wo die Dockerfile liegt)
2. `“docker run -v /home/user/sudokuSaver:/app/src/main/resources/fileStore/ -it sudoku-image”`

Der Pfad `/home/user/sudokuSaver` muss mit einem entsprechenden Pfad auf dem Host-System ersetzt werden. In diesem Verzeichnis werden die für die Applikation benötigten Dateien gespeichert. Somit muss dieser Pfad existieren und Schreib sowie Leserechte müssen vorhanden sein.

Technischer Überblick (2P)

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

Es wurde sich für die Programmiersprache Java entschieden, da in dieser Sprache bereits alle an dem Projekt teilnehmenden Studierenden Erfahrungen gesammelt haben. Als Testframework wurde sich für Junit5 entschieden. Dies ist ein weit verbreitetes Testframework in der Javawelt und bietet gute Einbindungsmöglichkeiten in das Projekt. Zum Persistieren von Daten wurde sich für lokale Textdateien entschieden. Zuerst stand eine Datenbank wie beispielsweise MariaDB im Raum. In Absprache mit dem Dozenten wurde dies aus Komplexitätsgründen wieder verworfen. Daher werden bei der Ausführung des Programms mehrere Dateien angelegt.

Kapitel 2: Clean Architecture (8P)

Was ist Clean Architecture? (1P)

[allgemeine Beschreibung der Clean Architecture in eigenen Worten]

Clean Architecture ist ein Architekturmuster / Vorgehensweise für die Softwareentwicklung, das darauf abzielt, den Code sauber, wartbar und unabhängig von externen Einflüssen zu gestalten und aufzubauen. Dabei basiert es im Wesentlichen auf den Prinzipien der SOLID-Prinzipien.

Die Clean Architecture fördert eine klare Trennung von Verantwortlichkeiten und Schichten in einer Anwendung. Sie definiert verschiedene Schichten, die jeweils spezifische Aufgaben erfüllen. Hierbei gibt es verschiedene Architekturmodelle wie beispielsweise Schichtenarchitektur oder hexagonale Architektur.

Die Clean Architecture fördert die Abhängigkeitsumkehrung (Dependency Inversion Principle) und Inversion of Control (IoC), indem sie sicherstellt, dass die Abhängigkeiten von den äußeren Schichten zu den inneren Schichten fließen. Dadurch wird der Code flexibler, testbarer und leichter wartbar, da die inneren Schichten unabhängig von den äußeren Schichten sind.

Das Hauptziel der Clean Architecture besteht darin, eine klare Trennung zwischen den verschiedenen Schichten zu erreichen, wodurch die Austauschbarkeit, Testbarkeit und Wartbarkeit der Software verbessert wird.

Analyse der Dependency Rule (3P)

[1 Klasse, die die Dependency Rule einhält und 1 Klasse, die die Dependency Rule verletzt; jeweils UML (mind. die betreffende Klasse inkl. der Klassen, die von ihr abhängen bzw. von der sie abhängt) und Analyse der Abhängigkeiten in beide Richtungen (d.h., von wem hängt die Klasse ab und wer hängt von der Klasse ab) in Bezug auf die Dependency Rule]

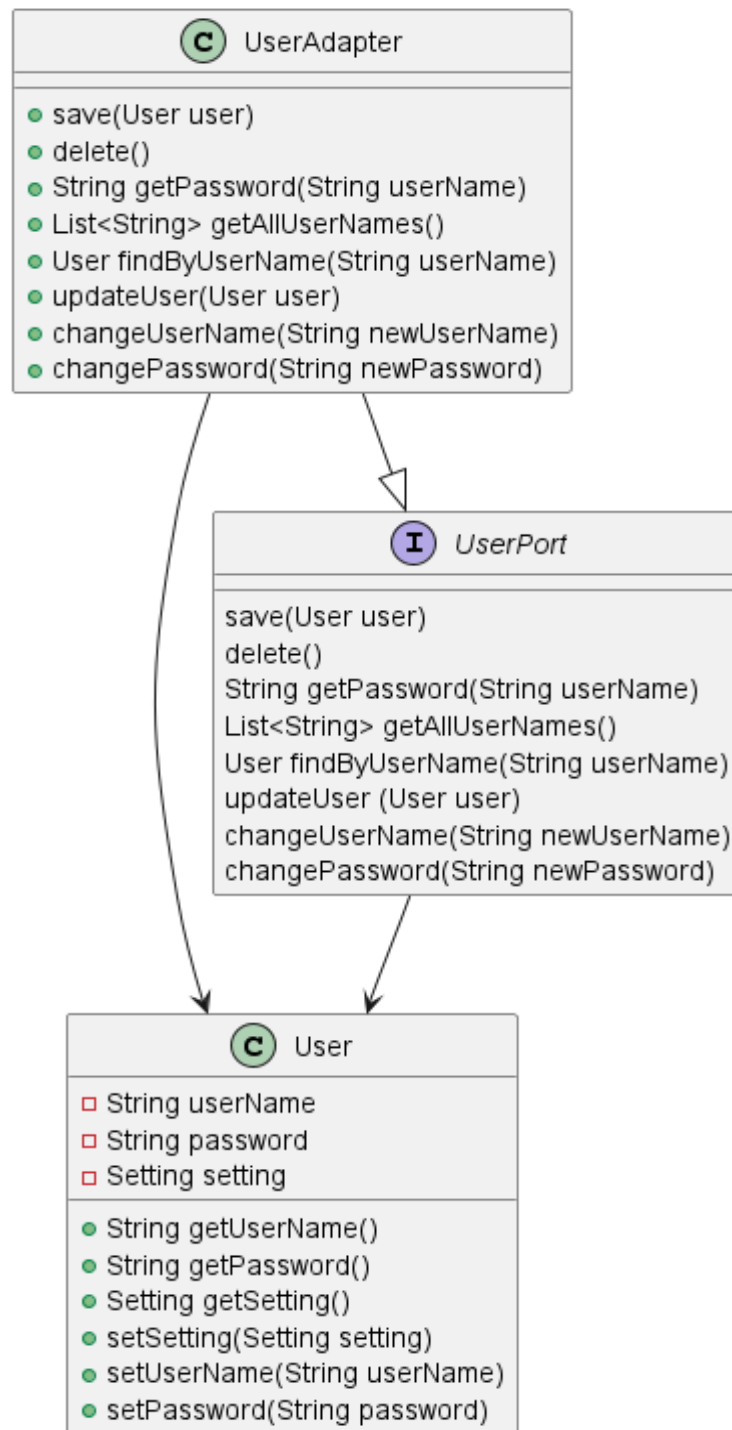
Erläuterung: Unser Projekt hat die Schichten: Domain, Application Infrastructure und Presentation mit jeweiligen Subpackages. Die Schichten sind bei uns im Projekt über Packages aufgebaut. Klassen im Adapter Package enden dabei immer mit dem Wort Adapter. Klassen im Servicepackage in der

Applicationschicht enden mit dem Namen Service. Wir haben bei uns im Projekt Use-Cases in sogenannte Serviceklassen zusammengepackt, da einige Use-Cases ähnliche oder gemeinsame Logik teilen. Dadurch wird der Code vereinfacht und Redundanzen vermieden. Darüber hinaus können benötigte externe Integrationen in einer Serviceklasse gekapselt und somit vereinfacht werden.

Positiv-Beispiel 1: Dependency Rule

In diesem Beispiel greift der UserAdapter auf das Usermodell zu. Der UserAdapter kümmert sich um das Laden und Speichern von Nutzern.

Im dargestellten Beispiel greift der UserAdapter, der zur Adapterschicht gehört, auf das User-Model aus dem Domainkern zu. Dieser Zugriff ist erlaubt, da die Abhängigkeit in die innere Richtung zeigt. Der UserAdapter implementiert das UserPort-Interface, welches als Schnittstelle zwischen dem Domainkern und der Adapterschicht dient. Durch die Verwendung des Ports wird die Adapterschicht abstrahiert und unabhängig vom konkreten Domainkern.

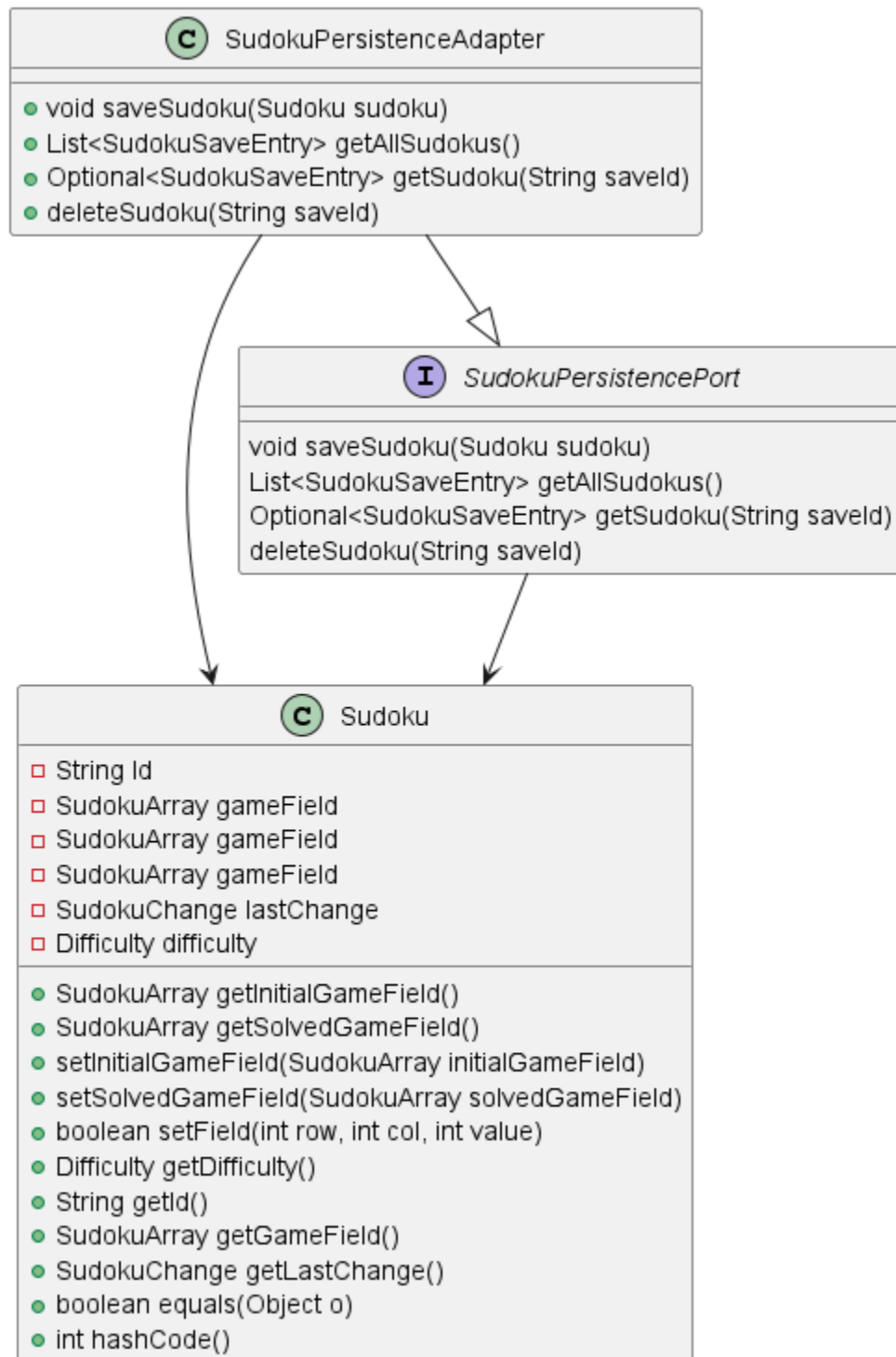


Die Dependency Rule wird hier eingehalten, da der Domainkern nicht von der Adapterschicht abhängt. Dadurch bleibt der Domainkern unberührt, selbst wenn Änderungen in der Adapterschicht vorgenommen werden müssen. Zum Beispiel, wenn Nutzer zukünftig in einer Datenbank gespeichert werden sollen, müsste nur die Adapterklasse geändert werden, während der Rest des Systems unverändert bleibt. Dies zeigt, wie die Abstraktion durch den Port eine klare Trennung zwischen den Schichten ermöglicht und die Änderungen auf diejenigen begrenzt, die direkt von ihnen abhängen.

Positiv-Beispiel 2: Dependency Rule

In diesem Beispiel greift der SudokuPersistenceAdapter, der zur Adapterschicht gehört, auf das Sudoku-Model aus dem Domainkern zu. Dieser Zugriff ist erlaubt, da die Abhängigkeit in die Richtung nach innen zeigt, also von der äußeren Adapterschicht zum inneren Domainkern. Der SudokuPersistenceAdapter implementiert das SudokuPersistencePort-Interface, das als Schnittstelle zwischen dem Domainkern und der Adapterschicht fungiert. Durch die Verwendung des Ports wird die Adapterschicht von den konkreten Details des Domainkerns abstrahiert.

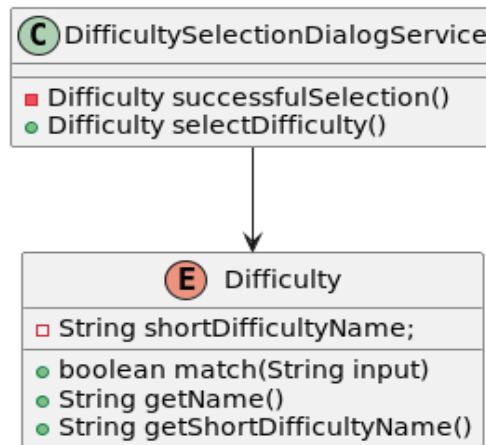
Durch die Einhaltung der Dependency Rule bleibt der Domainkern unberührt, wenn Änderungen in der Adapterschicht vorgenommen werden müssen. Wenn beispielsweise Sudokus in Zukunft in einer Datenbank gespeichert werden sollen, müsste nur die Adapterklasse (SudokuPersistenceAdapter) geändert werden. Der Rest des Systems, einschließlich des Domainkerns, bleibt von dieser Änderung unberührt. Dies zeigt, wie die Dependency Rule dazu beiträgt, die Auswirkungen von Änderungen auf diejenigen Schichten zu begrenzen, die direkt von ihnen abhängen.



Positiv-Beispiel 3: Dependency Rule

In dem gegebenen Beispiel greift der DifficultySelectionDialogService aus der Applicationschicht auf den Domainkern zu, um den Schwierigkeitsgrad des zu erstellenden Sudokus zu ermitteln. Der DifficultySelectionDialogService ist Teil der äußeren Schicht (Applicationschicht), während das Model Difficulty und der Domainkern zur inneren Schicht (Domain) gehören.

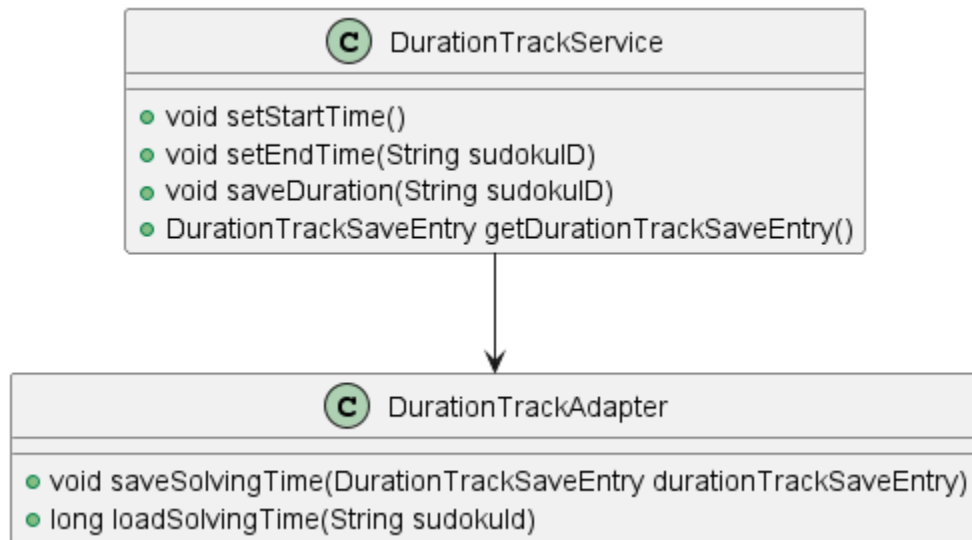
Da der Zugriff von außen (Applicationschicht) auf innen (Domainkern) erfolgt, wird die Dependency Rule eingehalten. Die äußere Schicht hängt von der inneren Schicht ab, aber nicht umgekehrt. Dadurch bleibt der Domainkern unabhängig von den Details der äußeren Schicht und kann sich auf seine Aufgabe, die Businesslogik, konzentrieren.



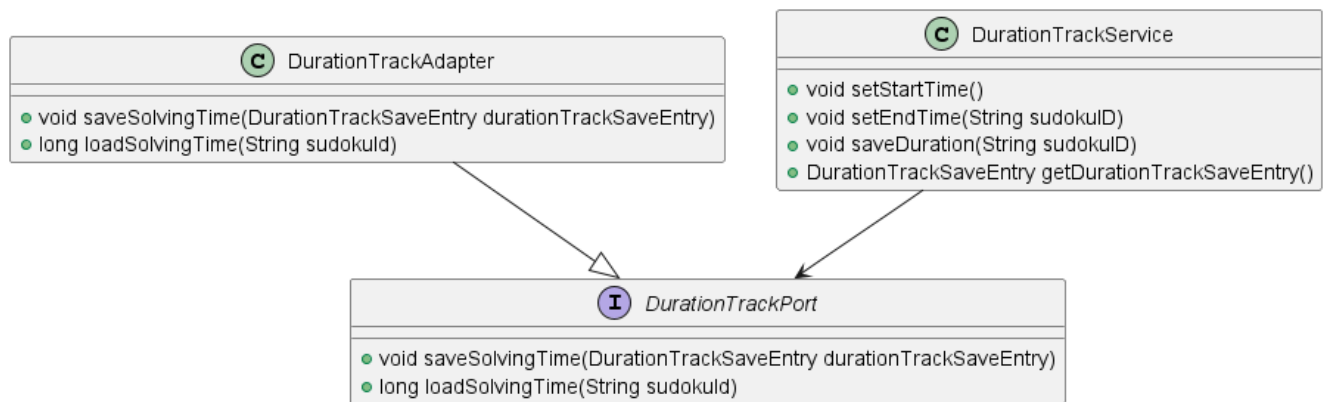
Negativ-Beispiel: Dependency Rule

Die Klasse DurationTrackService welche sich um das Messen der Spieldauer kümmert, nutzt direkt den Adapter für die Persistierung der Zeit. Dadurch greift eine innen liegende Schicht auf eine äußere zu, welches die Dependency Rule verletzt.

Vorher:



Lösung:



Im beschriebenen Beispiel greift die Klasse **DurationTrackService**, die zur inneren Schicht gehört, direkt auf den Adapter für die Persistierung der Zeit zu. Der **DurationTrackService** ist für das Messen der Spieldauer zuständig und nutzt den Adapter, um die Zeit zu persistieren.

Da der **DurationTrackService**, eine innere Schicht, direkt auf den Adapter, der zur äußeren Schicht gehört, zugreift, verletzt dies die Dependency Rule. Die innere Schicht sollte nicht direkt von der äußeren Schicht abhängen, da dies die Trennung der Schichten aufhebt.

Idealerweise sollte der **DurationTrackService** nicht direkt auf den Adapter zugreifen. Stattdessen sollte er über einen Port oder Schnittstelle mit der äußeren Schicht kommunizieren und die konkrete Implementierung des Adapters sollte in der äußeren Schicht erfolgen. Dadurch wird die Dependency Rule eingehalten und die Abhängigkeit wird von innen nach außen gelenkt.

Um die Dependency Rule einzuhalten, könnte beispielsweise ein **DurationTrackPort** definiert werden, über den der **DurationTrackService** mit der äußeren Schicht kommuniziert. Die konkrete Implementierung des Adapters erfolgt dann in der äußeren Schicht, während der **DurationTrackService** nur die abstrakte Schnittstelle des Ports verwendet.

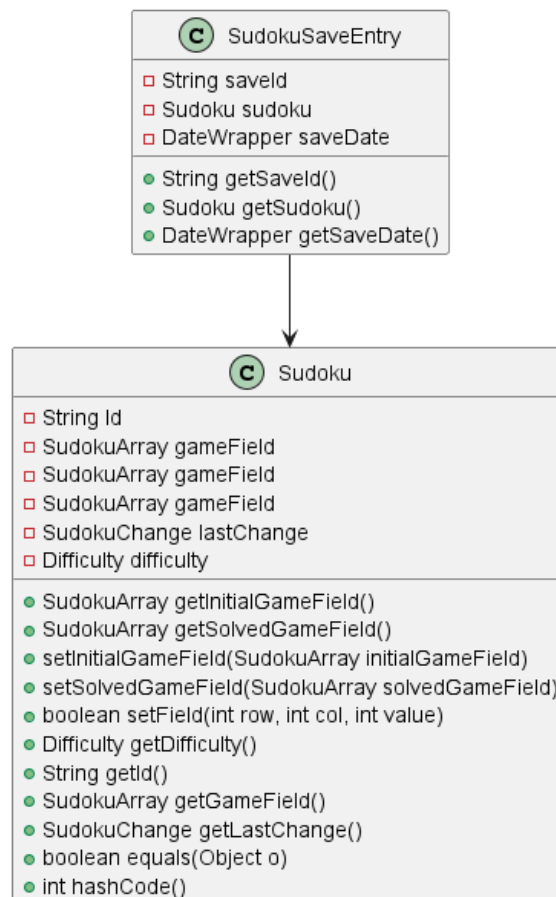
Analyse der Schichten (4P)

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

Erläuterung: Unser Projekt hat die Schichten: Domain, Application Infrastructure und Presentation mit jeweiligen Subpackages. Die Schichten sind bei uns im Projekt über Packages aufgebaut. Klassen im Adapter Package enden dabei immer mit dem Wort Adapter. Klassen im Servicepackage in der Applicationschicht enden mit dem Namen Service. Wir haben bei uns im Projekt Use-Cases in sogenannte Serviceklassen zusammengepackt, da einige Use-Cases ähnliche oder gemeinsame Logik teilen. Dadurch wird der Code vereinfacht und Redundanzen vermieden. Darüber hinaus können benötigte externe Integrationen in einer Serviceklasse gekapselt und somit vereinfacht werden.

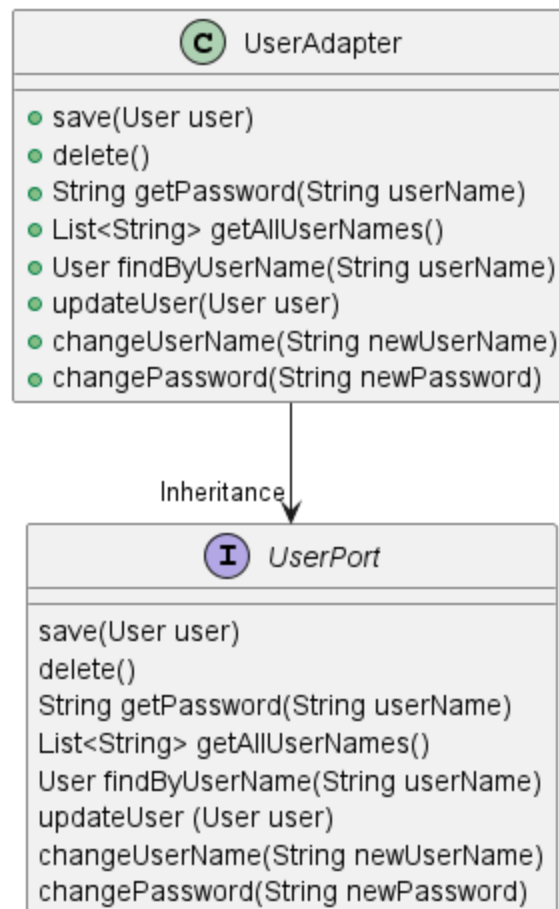
Schicht: Domainschicht

Die Domain-Schicht enthält das Kernmodell Ihrer Anwendung und definiert die Geschäftslogik. In unserem konkreten Anwendungsfall gibt es beispielsweise die Modelle Sudoku, SaveEntry, Difficulty und User. Diese Klassen sind unter anderem die Kernmodelle der Anwendung. Die Modelle kennen sich zum Teil auch gegenseitig wie in der nachfolgenden Abbildung dargestellt. Ein Speichereintrag muss das zu speichernde Sudoku kennen. Da die Modelle in der gleichen Schicht liegen ist das zulässig.



Schicht: Infrastructure / Adapterschicht

Die Adapter-Schicht enthält Implementierungen, die die Details der Infrastruktur und der externen Schnittstellen behandeln, wie z.B. das Laden und Speichern von Nutzern. Die Verwendung von Ports und Interfaces, wie dem UserPort, in der Adapterschicht ermöglicht es, die Abhängigkeiten abstrakt zu halten und die Adapterschicht austauschbar zu machen, ohne die Domäne zu beeinflussen. Dadurch können Änderungen in der Art und Weise, wie Nutzer gespeichert werden, in der Adapterklasse vorgenommen werden, ohne die anderen Schichten zu beeinträchtigen.



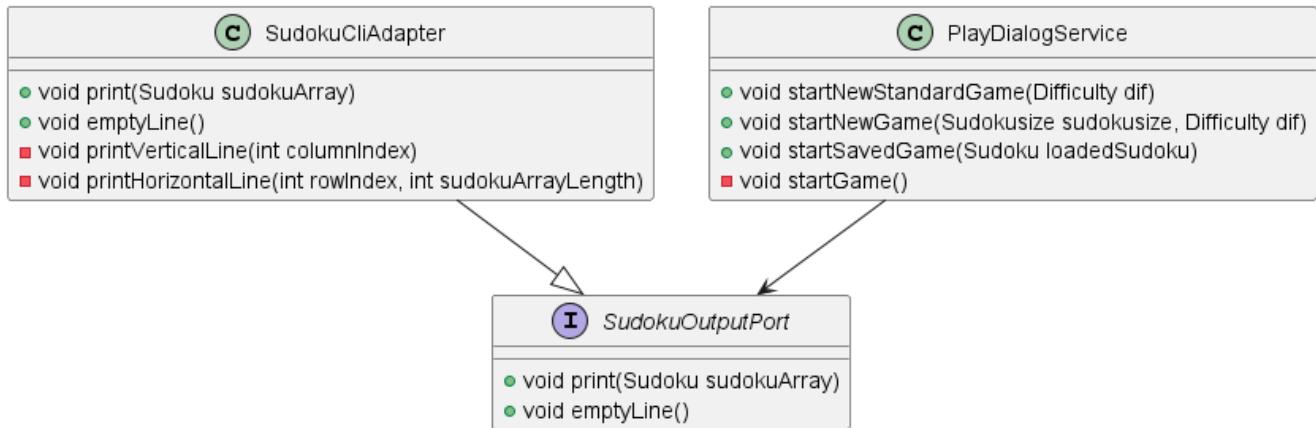
Kapitel 3: SOLID (8P)

Analyse SRP (3P)

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

Positiv-Beispiel 1

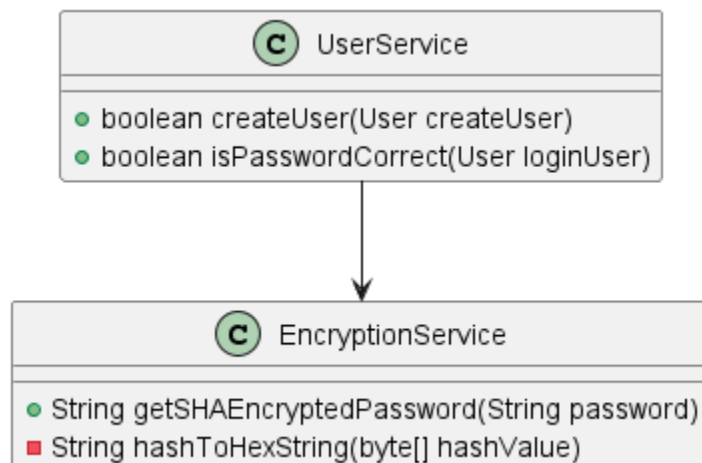
Die Klasse SudokuCliAdapter ist ein positives Beispiel für das Single Responsibility Prinzip. Die Klasse kümmert sich um die Ausgabe eines Sudokus auf der Kommandozeile. Sie beinhaltet die in dem UML angegebenen Felder und Methoden. Neben der Ausgabe erfüllt sie keinen anderen Zweck.



Im Fall der Klasse SudokuCliAdapter ist ihre Verantwortlichkeit klar definiert: die Ausgabe eines Sudokus auf der Kommandozeile. Die Klasse enthält die notwendigen Felder und Methoden, um diese Aufgabe zu erfüllen, darüber hinaus erfüllt sie allerdings keinen anderen Zweck. Das bedeutet, dass Änderungen an der Art und Weise, wie das Sudoku auf der Kommandozeile ausgegeben wird, nur diese Klasse betreffen und keine anderen Teile des Systems beeinflussen.

Positiv-Beispiel 2

Die Klasse EncryptionService das SRP, da sie sich ausschließlich um die Verschlüsselung des Nutzerpassworts kümmert. Die Klasse enthält die entsprechenden Felder und Methoden für die Verschlüsselung, aber darüber hinaus hat sie keine anderen Verantwortlichkeiten. Dies ermöglicht es, die Verschlüsselung unabhängig von anderen Funktionen des Systems zu entwickeln, zu testen und anzupassen.



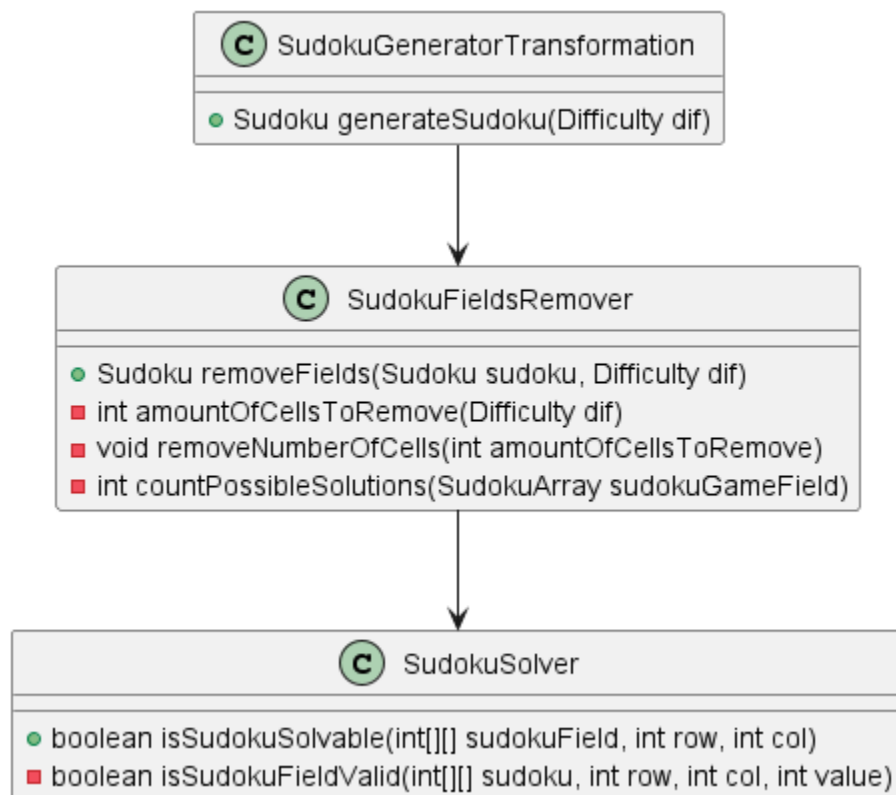
Negativ-Beispiel

Im Fall des SudokuFieldsRemover erfüllt die Klasse nicht das SRP, da sie neben dem Entfernen von Feldern aus einem Sudoku Feld auch noch die Funktion der Überprüfung, ob ein Sudoku Feld lösbar ist oder nicht, bereitstellt. Das sind zwei unterschiedliche Verantwortlichkeiten, die in separaten Klassen oder Modulen implementiert werden sollten.

Durch die Kombination dieser beiden Funktionen in einer Klasse wird die Abhängigkeit zwischen den verschiedenen Funktionen erhöht. Das erschwert die Wartung und Erweiterbarkeit des Codes, da Änderungen in einer Funktion unerwartete Auswirkungen auf eine andere Funktion haben könnten.

Um das SRP einzuhalten, sollte die Klasse in zwei separate Klassen aufgeteilt werden: eine Klasse, die sich ausschließlich auf das Entfernen von Feldern aus dem Sudoku Feld konzentriert, und eine andere Klasse, die die Überprüfung der Lösbarkeit des Sudoku Felds durchführt. Dadurch wird die Verantwortlichkeit klarer abgegrenzt und die Kohäsion und Wartbarkeit verbessert.

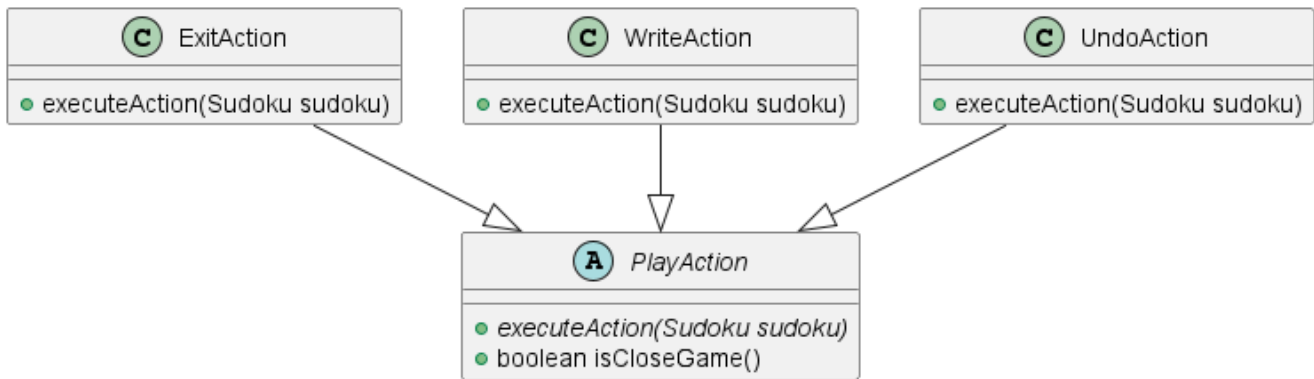
Verbesserung UML:



Analyse OCP (3P)

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

Positiv-Beispiel 1

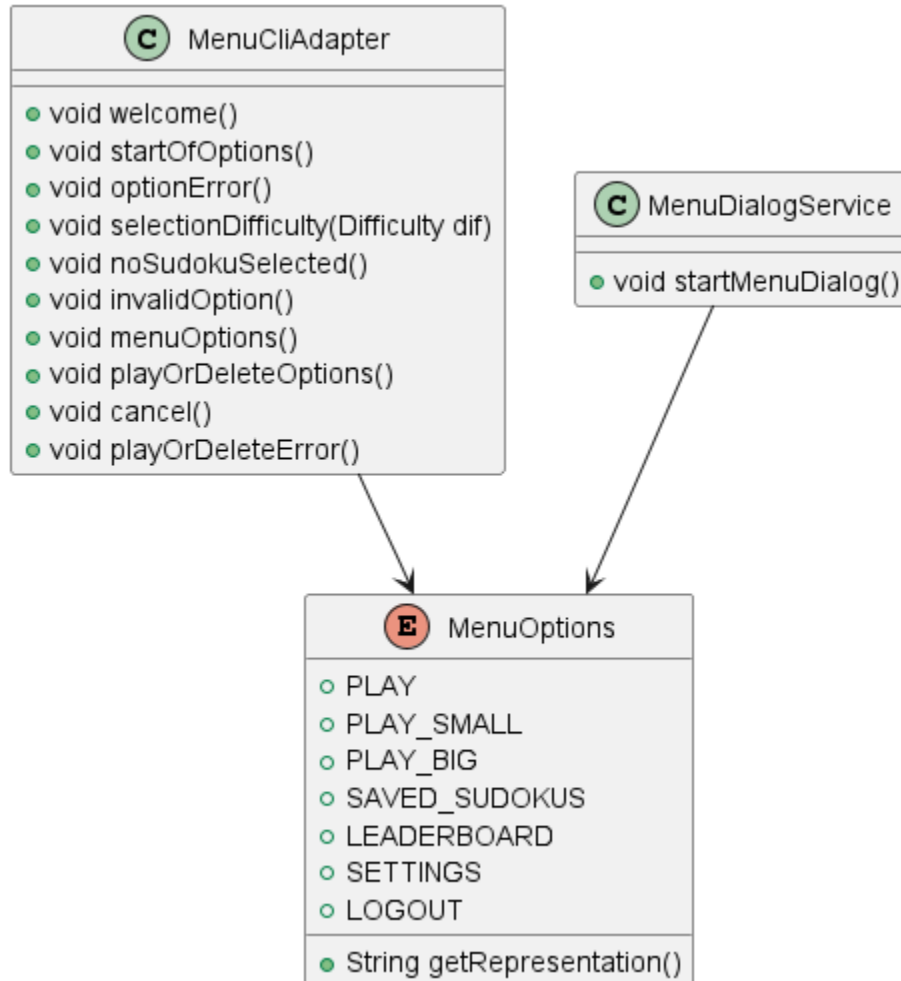


Im vorliegenden Beispiel wurde das Open-Closed-Prinzip angewendet, indem der Input vom Spieldialog mittels einer abstrakten Klasse PlayAction abstrahiert wurde. Durch den Einsatz von Vererbung und Polymorphismus können neue Eingabemöglichkeiten hinzugefügt werden, ohne bereits implementierte Mechanismen zu verändern.

Die abstrakte Klasse PlayAction definiert eine abstrakte Methode executeAction, die von allen konkreten Aktionsklassen implementiert werden muss. Dadurch kann die gewünschte Aktion über eine einheitliche Schnittstelle, die abstrakte Klasse PlayAction, aufgerufen werden. Bei der Auswahl einer Aktion wird der Datentyp PlayAction verwendet, und die Methode executeAction wird aufgerufen. Neue Aktionen können durch Erstellung einer neuen Klasse, die von PlayAction erbt, implementiert und eingebunden werden, ohne dass dies Auswirkungen auf die bereits vorhandenen Aktionen hat.

Dadurch erfüllt das System das Open-Closed-Prinzip, da es für Erweiterungen offen ist. Neue Aktionen können einfach hinzugefügt werden, indem neue Klassen erstellt werden, ohne dass dies Änderungen an bestehendem Code erfordert. Der bestehende Code bleibt unverändert und unbeeinflusst von den neuen Aktionen.

Positiv-Beispiel 2



Im vorliegenden Beispiel wird das Open-Closed-Prinzip ebenfalls angewendet, indem das Enum **MenuOptions** für die Optionen im Hauptmenü verantwortlich ist. Wenn ein neuer Eintrag im Enum hinzugefügt wird, erweitert dies automatisch das Menü. Dadurch kann das Hauptmenü einfach um neue Optionen erweitert werden, ohne dass der bestehende Code des Menüs geändert werden muss.

Ebenso erfüllt das Enum **MenuOptions** das Open-Closed-Prinzip, indem ein Menüpunkt einfach aus dem Enum entfernt werden kann, wenn er nicht mehr benötigt wird. Dadurch kann das Menü sauber und ohne Auswirkungen auf den restlichen Code angepasst beziehungsweise verändert werden.

Diese Vorgehensweise ermöglicht es, das Menü flexibel und erweiterbar zu gestalten, ohne den bestehenden Code zu modifizieren. Dadurch bleibt der Code stabil und es besteht keine Gefahr von ungewollten Nebeneffekten.

Negativ-Beispiel

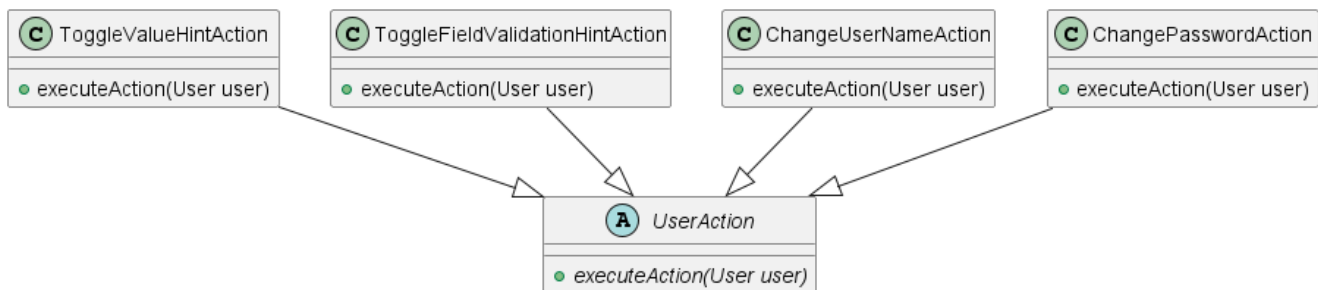
Negativ-Beispiel wurde Refactored, da es über die Zeit sehr unübersichtlich wurde.

Commitlink:

<https://github.com/mohjohfox/cli-sudoku/commit/b471c8596367d025cfe6745a2ea6e905892462f0>

<https://github.com/mohjohfox/cli-sudoku/commit/31c2345060f04dbce07956b5e583cd805a997090>

Bei der Auswahl im Settingsmenu basiert vieles in unstrukturiertem Programmcode. Dort wird die Eingabe durch verschiedene if-Überprüfungen gefiltert und geschaut um welche Auswahl es sich handelt. Durch ein Switch-Case wurde dann der passende Programmcode zur Eingabe ausgeführt. Dies führte jedoch zu sehr komplexem und unübersichtlichen Code. Des Weiteren war er nur noch sehr schlecht wartbar. Die Lösung war eine ähnliche Umsetzung wie beim Spieldialog. Es wurde eine neue abstrakte Klasse namens `UserAction` implementiert. Diese hatte eine Methode `executeAction`. Da im Settingsmenu die Hinweise ausgestellt sowie Username und Passwort geändert werden können, wurden für all diese Aktionen separate Klassen erstellt, welche von `UserAction` erben. Über Polymorphy wurde dann die Ausführung der jeweiligen Aktion umgesetzt.

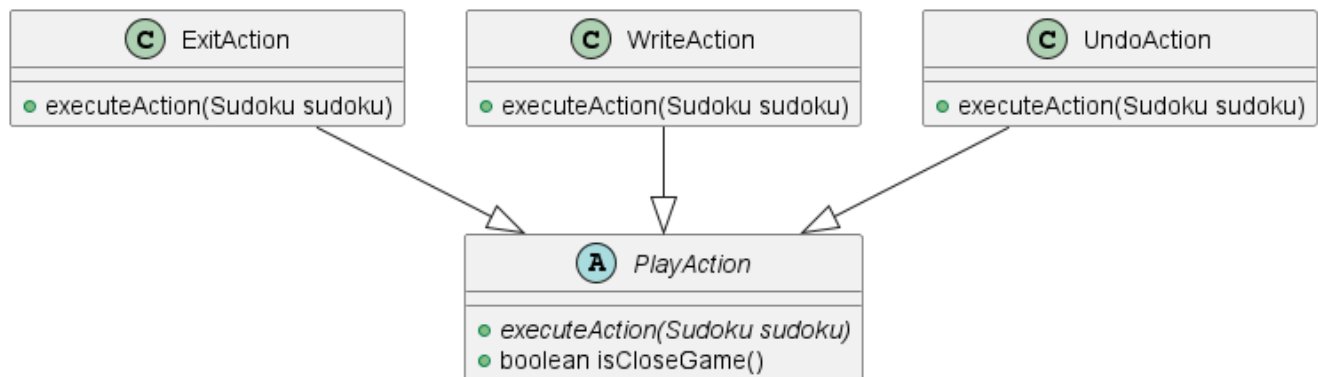


Analyse [LSP/ISP] (1P)

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

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

Positiv-Beispiel

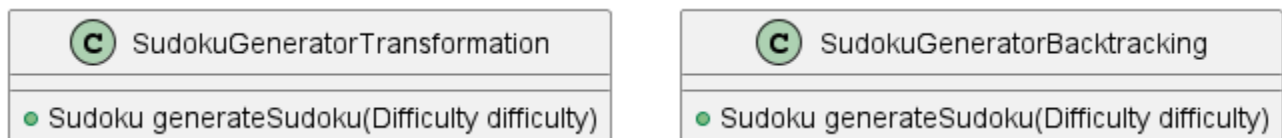


Im vorliegenden Beispiel wird das Liskovsche Substitutionsprinzip eingehalten, indem verschiedene ausführbare Aktionen eines Spielers von der abstrakten Klasse `PlayAction` erben. Die Klasse `PlayAction` dient als Abstraktion für die Zuweisung aller Implementierungen dieser Aktionen.

Durch die Verwendung der abstrakten Klasse PlayAction als Datentyp für die Zuweisung wird das Liskovsche Substitutionsprinzip umgesetzt. Objekte der abgeleiteten Klassen können anstelle von Objekten der Basisklasse verwendet werden, ohne dass dies die Funktionalität des Programms beeinträchtigt. Dies ermöglicht es, die verschiedenen ausführbaren Aktionen einheitlich zu behandeln und sie austauschbar zu verwenden, was die Flexibilität und Erweiterbarkeit des Codes verbessert.

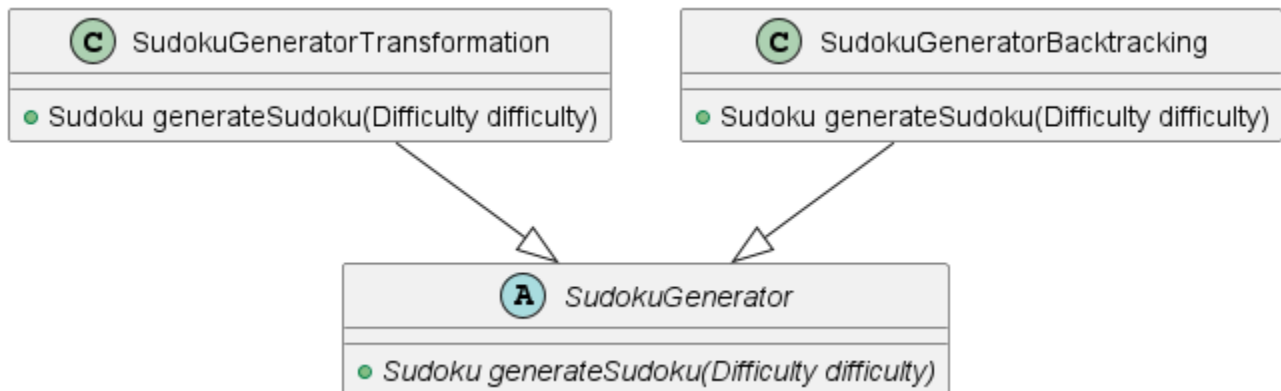
Negativ-Beispiel

Für die Generierung von Sudokus gibt es unterschiedliche Möglichkeiten. Wir haben im Projekt zwei umgesetzt. Für beide Optionen gibt es jeweils eine einzelne Klasse, welche sich um die Generierung auf die jeweilige Art kümmert. Bei der Generierung zur Laufzeit wird dann ein Objekt dieser konkreten Klasse instanziiert und verwendet. Falls weitere Möglichkeiten für die Generierung hinzukommen sollten, kann dies unübersichtlich werden und zu Implementierungsproblemen führen.



Ein möglicher Lösungsansatz wäre ein Interface oder abstrakte Klasse, welche beispielsweise SudokuGenerator heißt. Diese Klasse oder Interface beinhaltet dann die Methode generateSudoku. Für die Generierung kann dann die Klasse SudokuGenerator genutzt werden. Die konkrete Art der Generierung ist dann in der jeweiligen Unterklasse verborgen.

Lösungs-UML:



Um das Liskovsche Substitutionsprinzip einzuhalten und das Beispiel zu verbessern, könnten beispielsweise folgende Schritte umgesetzt werden:

1. Erstellung eines gemeinsamen Interfaces oder einer abstrakten Klasse: Um sicherzustellen, dass die abgeleiteten Klassen die gleichen Verhaltensweisen wie die Basisklasse aufweisen, sollte ein gemeinsames Interface oder eine abstrakte Klasse erstellt werden. In diesem Fall könnte ein Interface namens SudokuGenerator mit der Methode generateSudoku erstellt werden.

2. Implementierung des Interfaces oder der abstrakten Klasse: Die konkreten Generierungsklassen sollten das Interface SudokuGenerator implementieren oder von der abstrakten Klasse "Sudoku-Generator" erben. Dadurch wird sichergestellt, dass alle Generierungsoptionen die gleiche Schnittstelle haben und die Methode generateSudoku implementieren müssen.
3. Verwendung des gemeinsamen Interfaces oder der abstrakten Klasse: Anstatt direkt eine konkrete Klasse für die Generierung zur Laufzeit zu instanziiieren, sollte das gemeinsame Interface oder die abstrakte Klasse verwendet werden. Dadurch kann das Liskovsche Substitutionsprinzip eingehalten werden, da alle konkreten Generierungsoptionen austauschbar sind und die gleiche Schnittstelle bieten.

Durch dieses Vorgehen wird das Hinzufügen weiterer Generierungsoptionen vereinfacht. Wenn weitere Möglichkeiten für die Generierung hinzukommen sollten, können neue Klassen erstellt werden, die das gemeinsame Interface implementieren oder von der abstrakten Klasse erben. Dadurch bleiben die bestehenden Implementierungen unberührt, und es können problemlos neue Generierungsoptionen hinzugefügt werden.

Analyse [DIP] (1P)

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

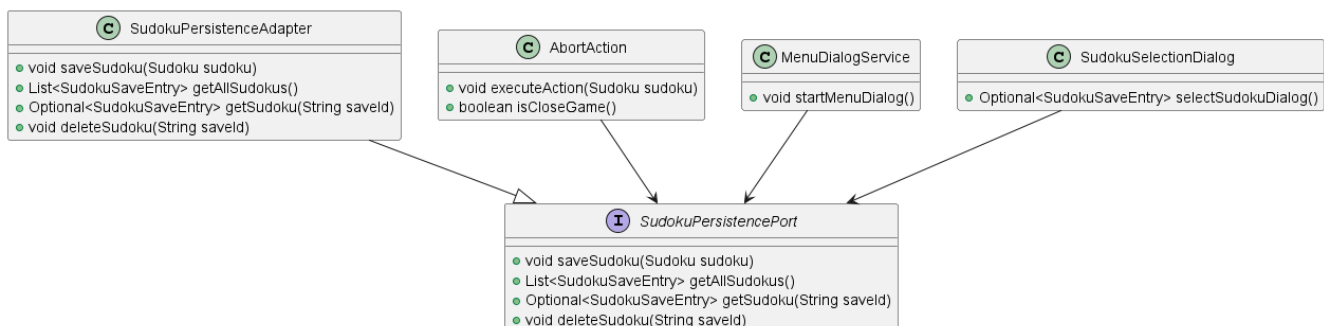
Positiv-Beispiel

Durch die hexagonale Architektur ist das DIP durch die Nutzung von Ports und Adaptern gewährleistet.

In dem Beispiel wird die Businesslogik der Anwendung von externen Systemen wie dem Schreiben oder Lesen in eine Datei entkoppelt. Dabei fungieren die Ports als abstrakte Schnittstellen, die die Funktionalitäten definieren, die von der Businesslogik benötigt werden. Die konkrete Implementierung dieser Schnittstellen erfolgt in den Adaptern, die die Interaktion mit den externen Systemen übernehmen.

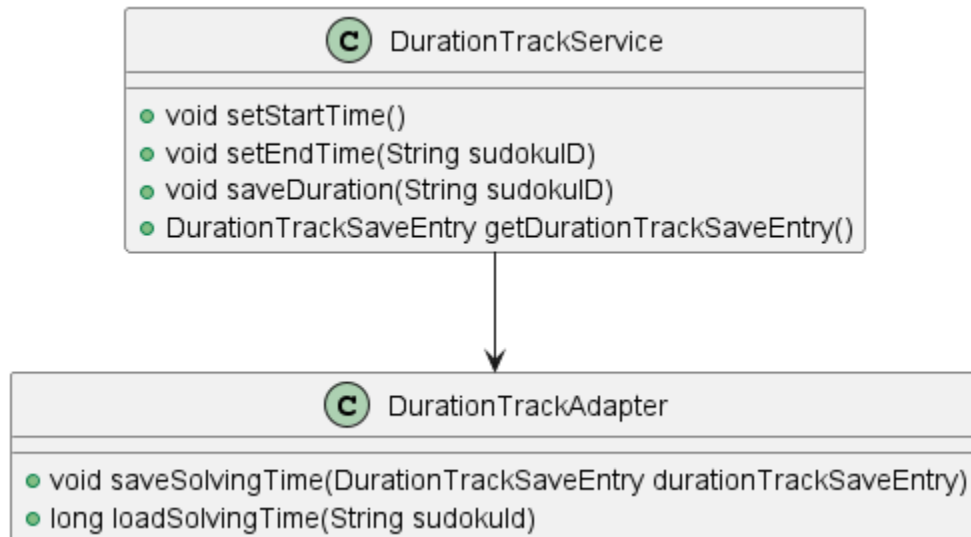
Die Verwendung von Ports und Adaptern ermöglicht eine hohe Flexibilität und Erweiterbarkeit des Systems. Wenn beispielsweise die Art des externen Systems geändert werden soll, muss nur der entsprechende Adapter angepasst werden, während die Businesslogik unberührt bleibt. Dies erfüllt das Dependency-Inversion-Prinzip, da die Businesslogik nicht von konkreten Implementierungen abhängig ist, sondern von den abstrakten Schnittstellen (Ports), die eine Entkopplung und Austauschbarkeit ermöglichen.

In dem nachfolgenden UML Diagramm sind Klassen aufgezeigt, welchen den SudokuPersistencePort nutzen und somit abstrahiert von der Implementierung sind aufgelistet.

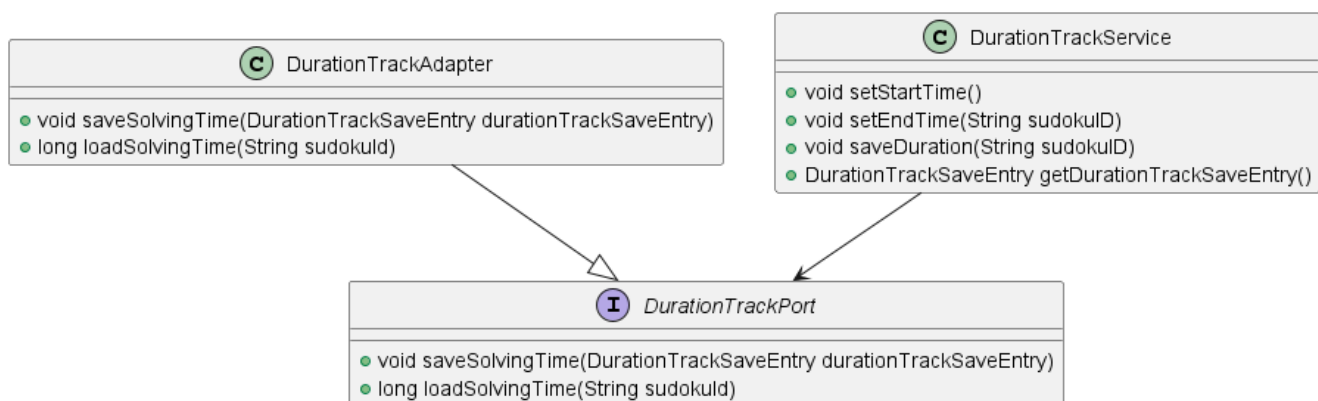


Negativ-Beispiel

Das Dependency-Inversion-Prinzip besagt, dass Abhängigkeiten von konkreten Implementierungen zu abstrakten Schnittstellen umgekehrt werden sollten. In diesem Fall verwendet die Klasse `DurationTrackService` direkt die Adapterklasse, was bedeutet, dass sie von der konkreten Implementierung abhängig ist und nicht von einer abstrakten Schnittstelle oder einem Port.



Um das Dependency-Inversion-Prinzip einzuhalten, wäre es empfehlenswert, einen Port oder eine abstrakte Schnittstelle zu definieren, die die benötigte Funktionalität abstrahiert. Die Klasse `DurationTrackService` sollte dann von diesem Port oder der abstrakten Schnittstelle (`DurationTrackPort`) abhängen, anstatt direkt von der konkreten Adapterklasse.



Indem der Port oder die abstrakte Schnittstelle verwendet wird, wird die Klasse `DurationTrackService` entkoppelt und nicht mehr direkt vom externen System abhängig. Dadurch kann die Implementierung des Adapters ausgetauscht werden, ohne die Klasse `DurationTrackService` zu beeinflussen.

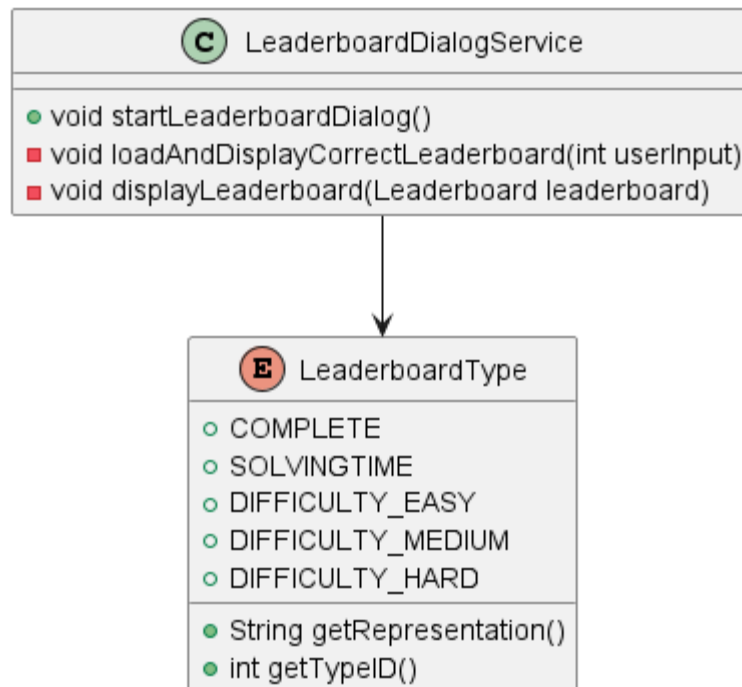
Die Nutzung eines Ports oder einer abstrakten Schnittstelle zur Abstraktion und Entkopplung der Klasse `DurationTrackService` würde dem Dependency-Inversion-Prinzip entsprechen und eine bessere Flexibilität, Testbarkeit und Erweiterbarkeit ermöglichen.

Kapitel 4: Weitere Prinzipien (8P)

Analyse GRASP: Geringe Kopplung (3P)

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

Ein Beispiel für geringe Kopplung liegt zwischen der Klasse LeaderboardDialogService und dem Enum LeaderboardType vor, wie das folgende UML verdeutlicht.



Das Enum **LeaderboardType** dient zur Darstellung der Art des Leaderboards. Es gibt die verschiedenen Leaderboardtypen Complete, Solvingtime, Difficulty_Easy, Difficulty_Medium, Difficulty_Hard. Für jeden Typ wird die jeweilige Repräsentation zur Ausgabe sowie die ID mit abgespeichert. Dadurch kann der LeaderboardTyp zur Ausgabe sowie zum Abgleich innerhalb der Anwendung verwendet werden.

Die Klasse **LeaderboardDialogService** dient zur Steuerung der Interaktion mit dem Anwender oder der Anwenderin des Sudokus. Dabei werden unter anderem Funktionalitäten zum Laden eines zuvor ausgewählten Leaderboards bereitgestellt. Zum Beispiel wird auf das Enum **LeaderboardType** zugegriffen.

Auf das Enum **LeaderboardType** wird aus der Klasse **LeaderboardDialogService** zugegriffen. Dadurch dass es sich jedoch in einer separaten Klasse befindet, lässt sich hier von einer geringen Kopplung sprechen.

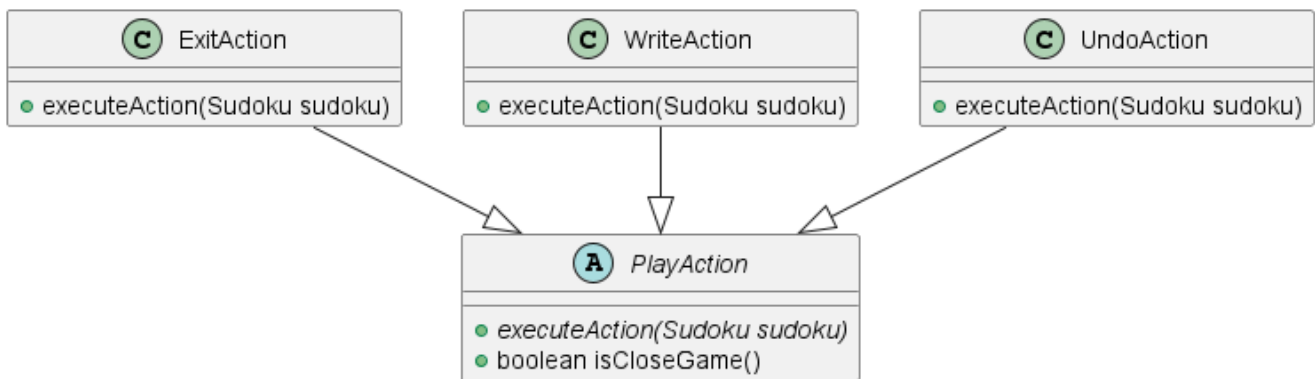
Dass es sich hier um low coupling handelt, lässt sich auch durch die dadurch auftretenden Vorteile aufzeigen. Aufgrund der geringeren Abhängigkeit weißt das Enum **LeaderboardType** eine viel höhere

Anpassbarkeit auf. Außerdem, lässt sich das Enum, auf Grund der separaten Klasse, auch in anderen, nicht hier dargestellten Klassen verwenden. Des Weiteren liegt eine bessere Lesbarkeit sowie Testbarkeit vor.

Analyse GRASP: Polymorphismus (1,5P)

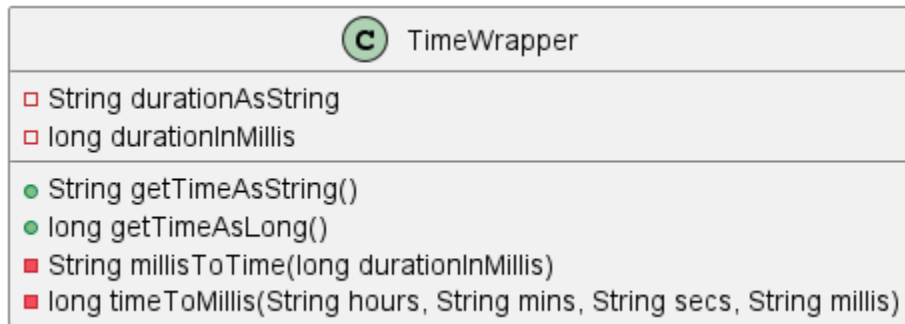
[eine Klasse als positives Beispiel entweder von Polymorphismus oder von Pure Fabrication; UML Diagramm und Begründung, warum es hier zum Einsatz kommt]

Ein Beispiel für Polymorphismus ist die abstrakte Klasse `PlayAction` und die Klassen `AbortAction`, `ExitAction`, `FixMistakesAction`, `RemoveAction`, `UndoAction`, `ValidationHintAction`, `ValueHintAction` und `WriteAction`, welche von der `PlayAction` Klasse erben. Die `PlayAction` Klasse definiert die abstrakte Methode `executeAction()`, welche somit von allen erbenden Klassen implementiert wird. In der Klasse `PlayDialogService` wird Methode `executeAction()` des vorliegenden `PlayAction` Objekts aufgerufen. Dabei ist egal, welche erbende Klasse genau vorliegt. Die Logik zum Behandeln der richtigen Operation wird nicht wie zuvor im `PlayDialogService` durchgeführt, sondern im entsprechenden Objekt. Soll eine weitere Aktion hinzugefügt werden, muss also eine neue Klasse erstellt werden, die von `PlayAction` erbt und zwangsweise die `executeAction()` Methode implementieren muss. Damit kann nicht vergessen werden die entsprechende Logik im `PlayDialogService` hinzuzufügen. Die Klasse `PlayDialogService` muss bei einer neuen Aktion mit diesem Konstrukt gar nicht angepasst werden.



Analyse GRASP: Pure Fabrication (1,5P)

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



Die Klasse `TimeWrapper` ist ein Beispiel für Pure Fabrication. Die Klasse beinhaltet ausschließlich Logik zur Konvertierung von Zeitpunkten.

Diese Funktionalität passt in keine Klasse der Problem-Domäne und wird somit ausgelagert in eine Eigenständige Klasse. Es wird Technologiewissen von Domainwissen getrennt.

Technologiewissen stellt hierbei die Konvertierung eines Zeitpunktes dar und Domainwissen die Verwendung des Zeitpunktes.

DRY (2P)

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

Commit:

<https://github.com/mohjohfox/cli-sudoku/pull/59/commits/d4b3d181180d133cf32fb8a240c2eb1c73f6102c>

Vorher:

```
9 @ public int calculateCompleteLeaderboardScore(String[] unsolvedOrWrongFields, boolean isCorrect, long timeInMillis, String difficultyAsString) {
10     int score = 0;
11     int difficultyAsInt = 0;
12     int scoreForSolvedFields;
13
14     if (difficultyAsString.equals(Difficulty.EASY.getName())) {
15         difficultyAsInt = 1;
16     } else if (difficultyAsString.equals(Difficulty.MEDIUM.getName())) {
17         difficultyAsInt = 2;
18     } else {
19         difficultyAsInt = 3;
20     }
21
22     if (isCorrect) {
23         score += 250;
24     } else {
25         scoreForSolvedFields = this.calculateScoreForUnsolvedOrWrongFields(unsolvedOrWrongFields, isCorrect);
26         score += scoreForSolvedFields;
27     }
28
29     score += difficultyAsInt * 0.75 * 66;
30     score += (System.currentTimeMillis() - timeInMillis) * 0.000001;
31
32     return score;
33 }
34
35 1 usage  ± Marcel
36 public int calculateTimeLeaderboardScore(String[] unsolvedOrWrongFields, boolean isCorrect, long timeInMillis) {
37     int score = 0;
38     int scoreForSolvedFields = 0;
39
40     score += (System.currentTimeMillis() - timeInMillis) * 0.000001;
41
42     scoreForSolvedFields = this.calculateScoreForUnsolvedOrWrongFields(unsolvedOrWrongFields, isCorrect);
43     score += scoreForSolvedFields;
44
45     return score;
46 }
```

Nachher:

```
11 public int calculateCompleteLeaderboardScore(String[] wrongFields, boolean isCorrect, long timeInMillis,
12     Difficulty difficulty) {
13     int score = 0;
14     int difficultyAsInt = getDifficultyAsInt(difficulty);
15
16     int scoreForSolvedFields;
17
18     if (isCorrect) {
19         score += 250;
20     } else {
21         scoreForSolvedFields = this.calculateScoreForWrongFields(wrongFields, isCorrect: false);
22         score += scoreForSolvedFields;
23     }
24
25     score += difficultyAsInt * 0.75 * 6;
26     score += calculateTime(timeInMillis);
27
28     return score;
29 }
30
31 1 usage  ▲ Marcel +1
32 public int calculateTimeLeaderboardScore(String[] wrongFields, boolean isCorrect, long timeInMillis) {
33     int score = 0;
34     int scoreForSolvedFields = 0;
35
36     score += calculateTime(timeInMillis);
37
38     scoreForSolvedFields = this.calculateScoreForWrongFields(wrongFields, isCorrect);
39     score += scoreForSolvedFields;
40
41     return score;
42 }
```

Durch das Entfernen der doppelten Logik zum Berechnen des Scores anhand der Lösungsdauer wurden die redundant Vorhandenen Informationen entfernt (siehe Vorherbild Zeile 30 und 39). Muss dieser Code Abschnitt nun überarbeitet, gepflegt oder eine Änderung vorgenommen werden, so muss die Logik nur an einer und nicht wie davor an mehreren Stellen angepasst werden. Es kann nun nicht mehr vorkommen, dass sich der Verrechnungswert mit der Zeitdifferenz an einer Stelle von der anderen unterscheidet. Die Berechnung ist übersichtlicher und kann auch noch an anderen Stellen einfacher wiederverwendet werden.

Ein Weiteres Beispiel ist das Auslagern der Logik für die Berechnung des Types des Schwierigkeitsgrades (siehe Vorherbild Zeile 14 bis 20).

Das Sudoku als Applikation wird so insgesamt sicherer, weniger Fehleranfällig und die Wartung vereinfacht sich.

Kapitel 5: Unit Tests (8P)

10 Unit Tests (2P)

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

Unit Test	Beschreibung
SudokuAdapterTest.saveSudokuTest()	Es wird die saveSudoku() Methode des SudokuPersistenceAdapters getestet, indem ein neu erzeugtes Sudoku gespeichert und dann anhand seiner Id mit dem Adapter wieder geladen wird. Danach wird geprüft, ob ein Sudoku ausgelesen werden konnte und, ob dieses mit dem erzeugten Sudoku übereinstimmt.
SudokuAdapterTest.deleteSudokuTest()	Es wird die deleteSudoku() Methode des SudokuPersistenceAdapters getestet. Dafür wird ein neues Sudoku generiert und mit der saveSudoku() Methode des SudokuPersistenceAdapters gespeichert. Daraufhin wird versucht das Sudoku anhand seiner Id mit der deleteSudoku() Methode zu löschen. Geprüft wird daraufhin, ob die Speicherdatei leer ist. Da, nach jedem Test die Speicherdatei gelöscht wird, wird bei Start des Testes zunächst die Datei ohne Speichereinträge angelegt.
UserAdapterTest.changeUserNameTest()	Es wird die changeUserName() Methode des UserPorts getestet. Dafür werden vor jedem Test zwei User angelegt. Anhand des Usernames wird der User user1 ermittelt. Daraufhin wird über die changeUserName() Methode der Name zu „MyNewUserName“ verändert. Danach wird anhand des neuen Users das User Objekt aus dem persistenten Speicher mit der getUser() Methode geladen. Außerdem wird versucht den User anhand des alten Namens zu laden. Nun wird geprüft, ob der neue geladene User die gleichen Einstellungen und Passwort hat nur mit dem neuen Namen. Außerdem wird geprüft, ob der User, der anhand der alten Namen geladen wurde null ist.
UserAdapterTest.changePasswordTest()	Es wird die changePassword() Methode der UserAdapter Klasse getestet. Das Vorgehen ist ähnlich zum changeUserNameTest(). Von einem der beiden testUser wird das Passwort mit der changePassword() Methode verändert. Danach wird der User neu geladen und überprüft, ob der Name, das neue Passwort und die Settings stimmen.
UserAdapterTest.deleteTest()	Es wird die delete() Methode der UserAdapter Klasse getestet. Dafür wird einer der zuvorgenerierten User anhand seines UserName mit der delete() Methode gelöscht. Danach wird anhand der getAllUserNames() Methode des UserAdapter Klasse überprüft, ob der Name in der Liste vorkommt.
SudokuValidatorTest.isSudokuValidTest()	Es wird die isSudokuValid() Methode der Klasse SudokuValidatorService getestet. Dafür wird ein MockValidFilledSudoku erstellt, welches ein valides gefülltes

	Sudoku mit der <code>getGameField()</code> Methode zurückliefert. Geprüft wird, ob die <code>isSudokuValid()</code> Methode <code>true</code> für das <code>gameField</code> des <code>MockValidFilledSudoku</code> Objektes zurückliefert.
<code>SudokuValidatorTest.isSudokuNotFullyFilledtest()</code>	Es wird die <code>isSudokuNotFullyFilled()</code> Methode der Klasse <code>SudokuValisatorService</code> getestet. Dafür wird ein Objekt der gleichen <code>MockValidFilledSudoku</code> Klasse erstellt. Die <code>getGameField()</code> methode liefert ein gefülltes Feld. Deswegen wird zuerst getestet, ob <code>sudokuValidatorService.isSudokuNotFullyFilled()</code> mit dem Feld als Parameter <code>false</code> ergibt. Daraufhin wird ein Wert des Feldes auf 0 gesetzt und erneut geprüft, ob die Methode nun <code>true</code> zurückliefert.
<code>SudokuValidatorTest.crossCheckTest()</code>	Es wird die Methode <code>crossCheck()</code> in der Klasse <code>SudokuValidatorService</code> getestet. Um ein valides gefülltes Feld zu erhalten wird ein <code>MockValidFilledSudoku</code> Objekt erstellt und die Methode <code>getGameField()</code> aufgerufen. Geprüft wird zunächst, ob <code>crossCheck</code> eine leere Liste zurückliefert, wenn das volle valide Sudoku geprüft wird. Daraufhin wird ein Feld auf 0 gesetzt und überprüft, ob die Methode weiterhin eine leere Liste liefert. Im letzten Schritt wird ein Feld auf einen falschen Wert gesetzt und überprüft, dass die Liste nun ein Element enthält.
<code>LeaderboardScoreClaculatorTest.calculateDifficultyLeaderboardScoreTest()</code>	Es wird die Methode <code>calculateDifficultyLeaderboardScore</code> in <code>LeaderboardScoreCalculator</code> getestet. Es wird der Score geprüft, der bei drei Fehlern bei jedem Difficulty entstehen und der maximale Score für jeden Difficulty, also, wenn es keiner Fehler gibt geprüft.
<code>UserServiceTest.createUserTest()</code>	Es wird die Methode <code>createUser()</code> der Klasse <code>UserService</code> getestet. Genau wird geprüft, dass eine Speicherung eines Users mit Sonderzeichen nicht erfolgreich ist.

ATRIP: Automatic (1P)

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

Die A-TRIP Eigenschaft Automatic sieht vor, dass die Tests einfach ausführbar und automatisch ablaufen. Des Weiteren müssen sie sich selbst überprüfen, das bedeutet es gibt nur die beiden Ergebnisse bestanden oder fehlgeschlagen.

Bei unserem Projekt werden die Tests automatisiert durch eine GitHub Pipeline bei jedem Build des Branches ausgeführt. Die Daten, welche für die Tests benötigt werden vor dem Durchlauf der Tests automatisiert erstellt, sodass jeder Testdurchlauf isoliert und unabhängig von anderen Testergebnissen läuft. Somit ist keine manuelle Benutzereingabe nötig. Des Weiteren wird dadurch sichergestellt, dass das Ergebnis entweder bestanden oder fehlgeschlagen ist.

Die Umsetzung über die GitHub Actions ist durch das Zusammenspiel von GitHub, Java und Maven möglich.

ATRIP: Thorough (1P)

[Code Coverage im Projekt analysieren und begründen]

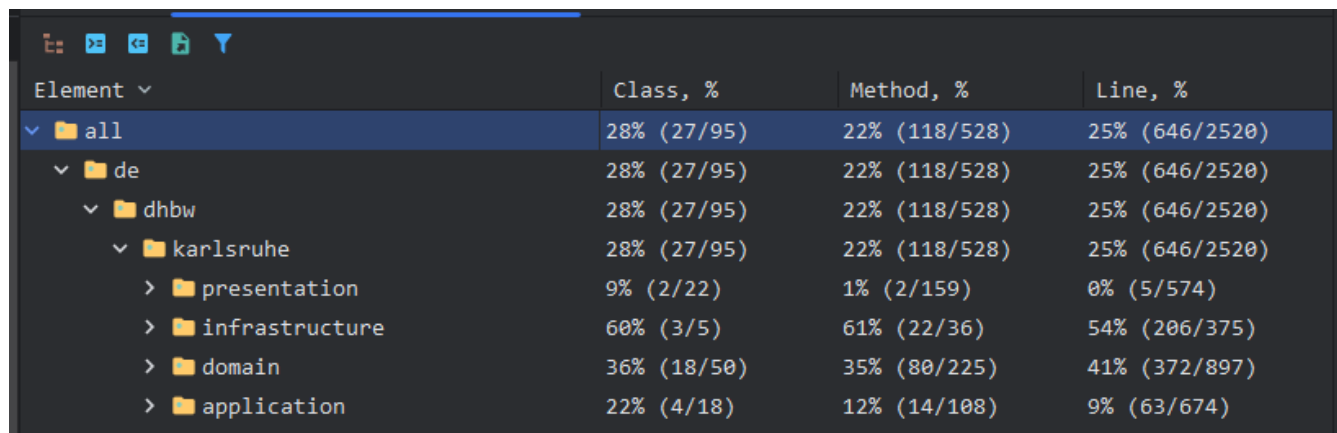
Die A-TRIP Eigenschaft Thorough sieht vor, dass die vorhandenen Tests alles Notwendige überprüfen. Dabei liegt das notwendige im Ermessen des Entwicklers.

Uns war es wichtig, dass vor allem die Benutzerfunktionalitäten sowie das Validieren der Sudokus validiert werden. Dies wird auch durch die Auflistung der Unittests in Kapitel 5 deutlich.

Die Entscheidung basiert auf der Notwendigkeit der Funktionalitäten für das Funktionieren der Applikation. Denn ein Sudoku lässt sich schlecht ohne Benutzer oder Validierung spielen.

Durch das Testen von Funktionalität in bestimmten Bereichen, wird verhindert, dass Fehler übersehen werden. Fehler verteilen sich nicht gleichmäßig über den Code, sondern befinden sich meistens an zusammenhängenden Stellen.

Auch das mehrmalige Ausführen der Tests für unseren Code liefern immer das selbe Ergebnis. Das Ergebnis ist also unabhängig von der Umgebung. Darüber hinaus kann die Code-Coverage über IntelliJ gemessen werden. Folgender Bildausschnitt zeigt die momentane Testcoverage der einzelnen Schicht an:



Element	Class, %	Method, %	Line, %
all	28% (27/95)	22% (118/528)	25% (646/2520)
de	28% (27/95)	22% (118/528)	25% (646/2520)
dhbw	28% (27/95)	22% (118/528)	25% (646/2520)
karlsruhe	28% (27/95)	22% (118/528)	25% (646/2520)
presentation	9% (2/22)	1% (2/159)	0% (5/574)
infrastructure	60% (3/5)	61% (22/36)	54% (206/375)
domain	36% (18/50)	35% (80/225)	41% (372/897)
application	22% (4/18)	12% (14/108)	9% (63/674)

Ein weiterer Schritt wäre das Einbauen von der Test-Coverage in die CI-CD Pipeline. Dort könnte die Pipeline dann prüfen, ob eine gewisse Testabdeckung vorhanden ist.

ATRIP: Professional (1P)

[1 positives Beispiel zu 'Professional'; Code-Beispiel, Analyse und Begründung, was professionell ist]

Als Beispiel dient der UserAdapterTest. In dieser Testklasse sind zwei Hilfsfunktionen eingesetzt, welche vor und nach jedem Test laufen um eine isolierte Testumgebung herzustellen. Dadurch sind Hilfsfunktionen implementiert und werden bei jedem Test genutzt. Darüber hinaus testet die Klasse alle Funktionen des UserAdapters. Die Methode findByUsername wird implizit in den Tests verwendet. Daher ist für diese Funktion kein explizierter Test vorhanden.

👤 mohjohfox +1

@BeforeEach

```
void createFile() {  
    UserPort userPort = new UserAdapter(Location.TEST);  
    for (int i = 0; i < 2; i++) {  
        User user = createUser(i);  
        userPort.save(user);  
    }  
}
```

👤 mohjohfox

@AfterEach

```
void deleteFile() {  
    try {  
        Files.deleteIfExists(Path.of(first: Location.TEST.getLocation() + "userStoreFile"));  
    } catch (IOException e) {  
        e.printStackTrace();  
    }  
}
```

👤 mohjohfox +1

@Test Mohjohfox, 28/05/2023 10:17 • added tests for updating username

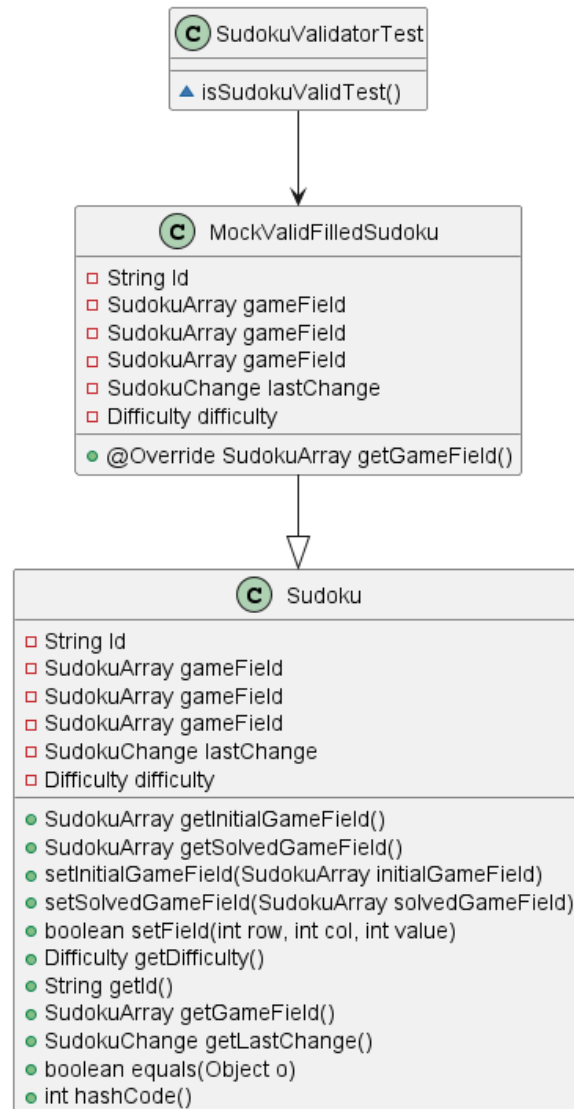
```
void changeUserNameTest() {  
    UserPort userPort = new UserAdapter(Location.TEST);  
    User user = userPort.findByUserName("user1");  
    GameInformation.username = user.getUserName();  
    String newUserName = "MyNewUserName";  
  
    userPort.changeUserName(newUserName);  
  
    User updatedUser = userPort.findByUserName(newUserName);  
    User oldUser = userPort.findByUserName(user.getUserName());  
  
    assertEquals(updatedUser.getUserName(), newUserName);  
    assertEquals(updatedUser.getSetting(), user.getSetting());  
    assertEquals(updatedUser.getPassword(), user.getPassword());  
    assertNull(oldUser);  
}
```

Fakes und Mocks (3P)

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

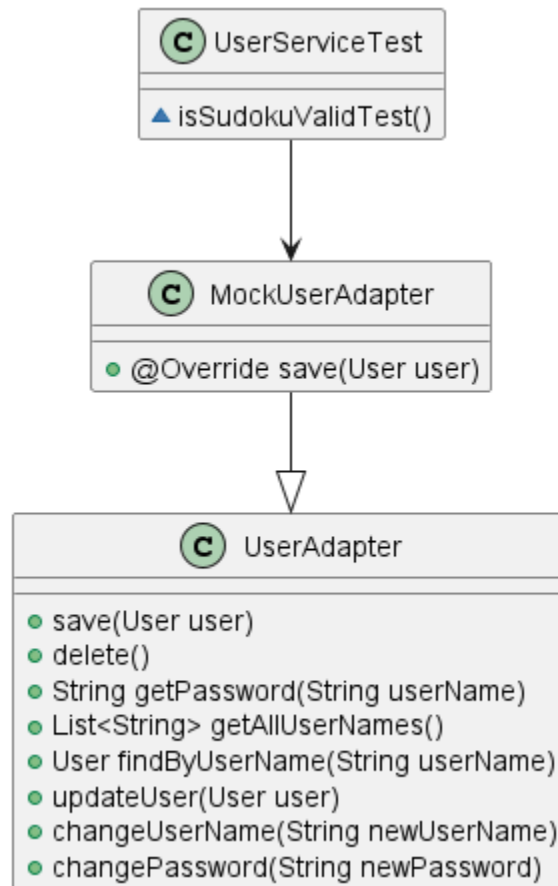
Erstes Beispiel:

Wie oben bereits beschrieben, wird ein Mock der Sudoku Klasse verwendet. Diese Klasse heißt MockValidFilledSudoku und erbt von der Klasse Sudoku. Es ist sinnvoll ein Mock des Sudokus zu verwenden, da für den SudokuValidatorTest.isSudokuValidTest() Test nur ein Attribut der Sudokuklasse benötigt wird. Dieses Attribut ist das gameField, welches außerdem die Bedingung, dass es schon voll ausgefüllt und richtig ist, erfüllen muss. Mit dem SudokuBuilder lässt sich so ein Sudoku nicht erstellen, daher ist der Einsatz des Mocks, welches die Werte im Konstruktor ignoriert und in der Override Methode getGameField() das nötige Feld zurückliefert sinnvoll.



Zweites Beispiel:

Für den letzten, oben beschriebenen Test wird ein Mock des UserAdapter verwendet. Dieser heißt `MockUserAdapter()` und sorgt, dafür, dass die `UserService` Klasse auf einen `UserAdapter` zugreifen kann aber dieser nicht die Schreiboperationen ausführen muss, da diese nicht Teil des Testes sind.



Kapitel 6: Domain Driven Design (8P)

Ubiquitous Language (2P)

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

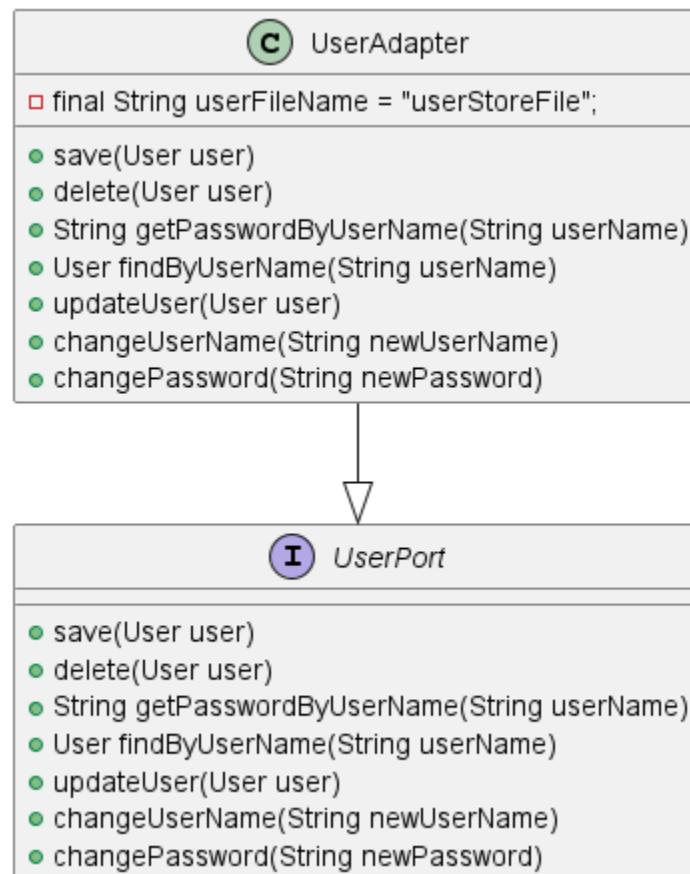
Bezeichnung	Bedeutung	Begründung
Gamefield	Zweidimensionales Array mit Werten. Beide die Größe 9*9. In der vorliegenden Definition wird Dimensionen sind gleich die Größe jedoch nicht beschränkt, da auch 4*4 und groß.	Ein (Sudoku) Game field hat im klassischen Sinne 16*16 Sudokus im Programm vorkommen.
Output	Textuelle Ausgabe von Nutzeranweisungen oder de, Spielfeld in die Konsole.	Output kann in jeder Form vorliegen. Als Output könnten beispielsweise auch die Dateien zur Persistierung der Daten bezeichnet werden. Jedoch ist dies unter der Definition nicht gemeint.
Setting	Informationen, mit welchen Hilfsaktionen ein Sudoku gelöst werden kann.	Setting kann in verschiedenen Spielkontexten unterschiedliche Dinge bedeuten. In unserem Fall geht es nur um aktive Tipps.
Menu	Dialog, der es den Anwender:innen ermöglicht zwischen	Menu ist ein doppeldeutiger Begriff der zum einen sich auf Essen oder auf den hier vorliegenden Fall beziehen kann. Daher ist die Klärung der Definition

verschiedenen Optionen sinnvoll.
auszuwählen.

Repositories (1,5P)

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

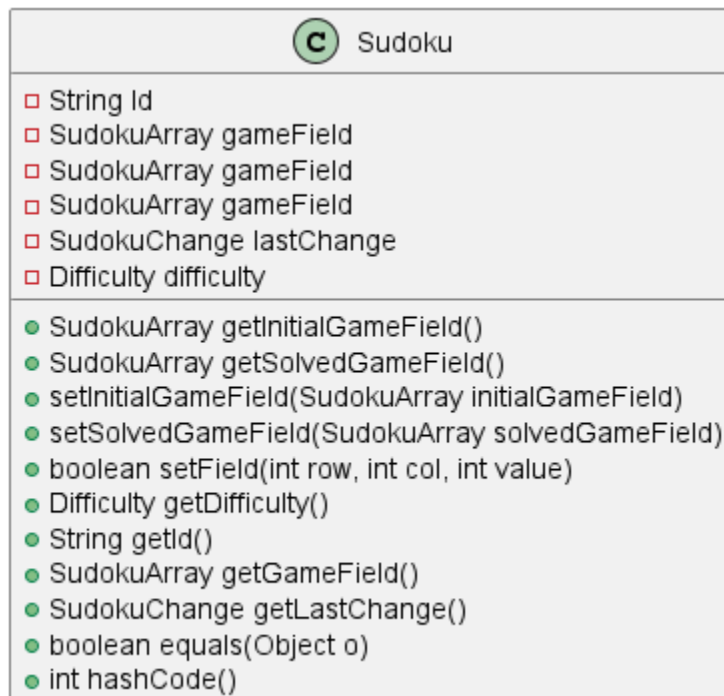
Das Interface UserPort stellt ein erweitertes Repository dar. Es ist die Schnittstelle zum Speichern, Löschen, Verändern und Finden von Nutzern. Das Interface wird implementiert von der Klasse Sudoku Adapter. Während dem Programmablauf ist es nötig, einen User aus dem persistenten Speicher anhand des userNames zu laden, außerdem soll es möglich sein den UserName und das Password eines Users zu ändern. Dafür bietet sich die Nutzung eines User-Repositories in Form des UserPorts an. Insgesamt entspricht diese Beschreibung und das UML-Diagramm einer erweiterten UserRepository Implementierung mit zusätzlichen Funktionalitäten, die über die grundlegenden CRUD-Operationen hinausgehen.



Aggregates (1,5P)

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

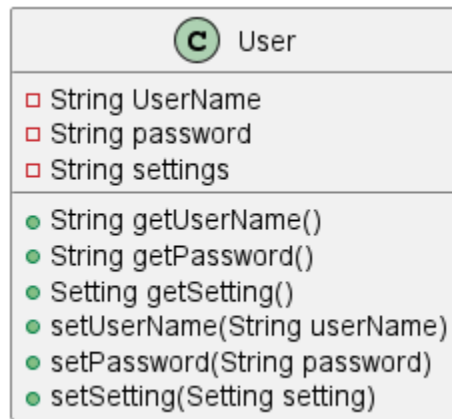
Die Sudoku Klasse stellt zusammen mit seinen Instanzvariablen ein Aggregat dar. Bei den Instanzvariablen handelt es sich um mehrere Objekte der Klasse SudokuArray, einem SudokuChange, einem Id String und einem Difficulty Object. Die letzteren zwei Objekte werden beim Erstellen des Sudsokuobjekts erzeugt und danach nicht mehr verändert. Auch die zwei Objekte initialGameId und solvedGameField der Klasse SudokuArray, werden nach der Erstellung des Sudokus nicht mehr verändert. Jedoch kann das gameField, ebenfalls ein Objekt der Klasse SudokuArray, während des Spielverlaufs verändert werden. Von außen kann jedoch nicht auf dieses Objekt direkt zugegriffen werden, sondern wird es nur modifiziert über die setField() Methode der Sudoku Klasse. Damit garantiert die Sudokuklasse einen konsistenten Zustand des gameField im Zusammenhang zu den anderen Objekten und damit des gesamten Aggregates. Das SudokuChange Objekt ist bei Erstellung zunächst null und wird beim Setzen eines Feldes verändert, es kann ebenfalls nicht von außen manipuliert werden und wird von der setField() Methode des Sudokus immer für den aktuellen Zug neu instanziiert. Die setField() Methode liefert dabei dem Aufrufer zurück, ob die Aktion erfolgreich durchgeführt wurde.



Entities (1,5P)

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

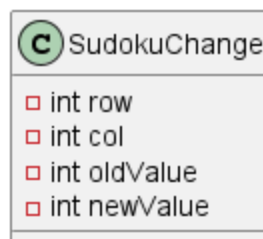
Die Userklasse ist eine zentrale Entity der Anwendung. Sie kennzeichnet sich durch einen eindeutigen UserName, der nicht doppelt vergeben werden darf/kann aus. Das ist sinnvoll, da bei der Anmeldung verschiedene Benutzer über eine eindeutige Eigenschaft unterschieden werden müssen. Im Leaderboard wäre es auch nicht sinnvoll zwei unterschiedliche Nutzer mit dem gleichen UserName anzuzeigen, da diese so nicht unterschieden werden könnten.



Value Objects (1,5P)

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

Das Record SudokuChange stellt ein Value Object dar, da die Werte nach dem setzen im Konstruktor nicht mehr verändert werden können. Das Record wird nur in der Sudokuklasse erstellt und für jede nötige Veränderung wird ein neues SudokuChange Record erstellt. Das SudokuChange record gibt die Änderung eines Feldes von einem alten zu einem neuen Wert an. Ein Value Object ist dafür sinnvoll, da in einem Zug des Spiels sich auch nur ein Feld ändert. Diese Änderung lässt sich perfekt mit dieser Klasse abbilden. Ein bestimmter Zug verändert sich auch nicht, nachdem er durchgeführt wurde. Damit ist die finale Eigenschaft eines Value Objects passend.



Kapitel 7: Refactoring (8P)

Code Smells (2P)

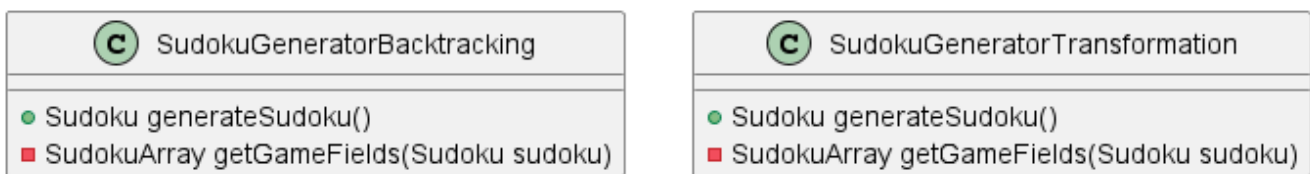
[jeweils 1 Code-Beispiel zu 3 unterschiedlichen Code Smells aus der Vorlesung; jeweils Code-Beispiel und einen möglichen Lösungsweg bzw. den genommen Lösungsweg beschreiben (inkl. (Pseudo-)Code)]

[CODE SMELL 1] – Duplicated Code

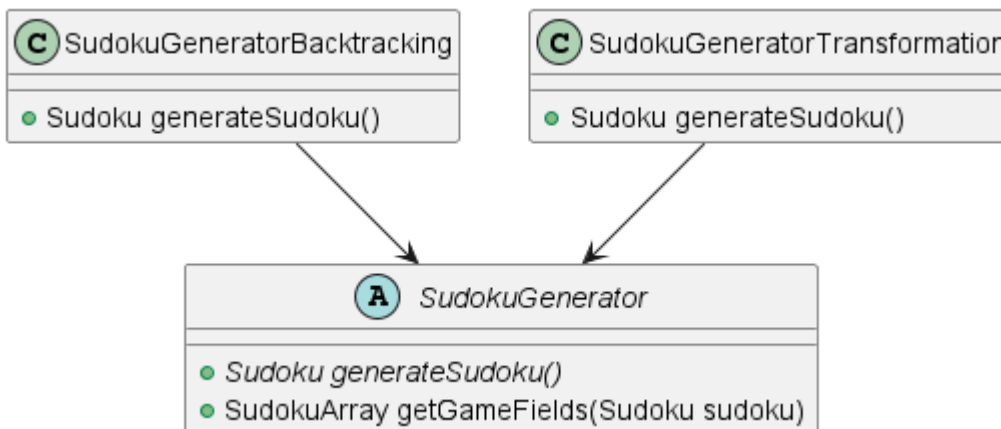
Commit:

<https://github.com/mohjohfox/cli-sudoku/commit/b4335f11c039cb3b043e0bd7cfb7018a9ef0f53a>

Vorher:



Nachher:



Für eine gute Varianz bei der Generierung von Sudokus sind zwei unterschiedliche Generierungsmöglichkeiten implementiert. Beide Generierungsklassen stellen hierbei die Funktion `generateSudoku()` bereit. Hierfür nutzen sie die Methode `getGameFields(Sudoku sudoku)`. Die Generierungsmethoden sind durch die Generierungseigenschaften unterschiedlich. Die `getGameFields` Methode ist jedoch exakt gleich und war anfangs doppelt implementiert. Dadurch muss die Methode bei einer Anpassung oder Veränderung auch angepasst werden und das in beiden Klassen.

```

public class SudokuGeneratorTransformation {

    public Sudoku generateSudoku(Difficulty dif) {
        SudokuTransformation sudokuTransformation = DependencyFactory
            .getInstance()
            .getDependency(SudokuTransformation.class);

        this.sudoku = sudokuTransformation.transform(this.sudoku);

        sudoku.setSolvedGameField(getGameFields(sudoku));
        SudokuFieldsRemover sudokuFieldsRemover = DependencyFactory
            .getInstance()
            .getDependency(SudokuFieldsRemover.class);

        this.sudoku = sudokuFieldsRemover.removeFields(this.sudoku,dif);

        this.sudoku.setInitialGameField(this.sudoku.getGameField());
    }

    private SudokuArray getGameFields(Sudoku sudoku) {

        SudokuArray tmpGameField = new SudokuArray(
            sudoku.getGameField().getCopyOfSudokuArra());

        return tmpGameField;
    }

}

```

```

public class SudokuGeneratorBacktracking {

    public Sudoku generateSudoku(Difficulty difficulty) {
        Sudoku sudoku = new Sudoku(difficulty);
        for (int i = 0; i < 9; i++) {
            for (int k = 0; k < 9; k++) {
                sudoku.getGameField().sudokuArray()[i][k] = 0;
            }
        }
        fillSudokuField(sudoku.getGameField().sudokuArray(), 0, 0);
        sudoku.setSolvedGameField(getGameFields(sudoku));
        int amountOfCellsToRemove = switch (difficulty) {
            case EASY:
                yield 40;
            case MEDIUM:
                yield 50;
            case HARD:
                yield 60;
        };
        removeCells(sudoku.getGameField(), amountOfCellsToRemove);
        SudokuArray tmpGameField = getGameFields(sudoku);
        sudoku.setInitialGameField(tmpGameField);
        return sudoku;
    }

    private SudokuArray getGameFields(Sudoku sudoku) {

        SudokuArray tmpGameField = new SudokuArray(
            sudoku.getGameField().getCopyOfSudokuArra());

        return tmpGameField;

    }

    ...

}

```

Um den duplizierten Code zu vermeiden ist eine abstrakte Klasse namens SudokuGenerator implementiert, welche die getGameField Methode beinhaltet. Darüber hinaus abstrahiert sie auch die Generierungsmethode. Die ursprünglichen Generierungsklassen erben nun von der abstrakten Klasse und haben dadurch die get-Methode zur Verfügung. Darüber hinaus ist sichergestellt, dass eine Generierungsklasse auch immer eine Generierungsmethode zur Verfügung stellt.

Pseudocode für Lösung:

```
public SudokuGeneratorBacktracking extends SudokuGenerator {  
    Sudoku generateSudoku() {  
        ...  
        GameField field = getGameFields(sudoku)  
        return generatedSudoku;  
    }  
}
```

```
public SudokuGeneratorTransformation extends SudokuGenerator {  
    Sudoku generateSudoku() {  
        ...  
        GameField field = getGameFields(sudoku)  
        return generatedSudoku;  
    }  
}
```

```
abstract class SudokuGenerator {  
    abstract Sudoku generateSudoku();  
  
    public SudokuArray getGameFields(Sudoku sudoku) {  
        ...  
        return gameField  
    }  
}
```

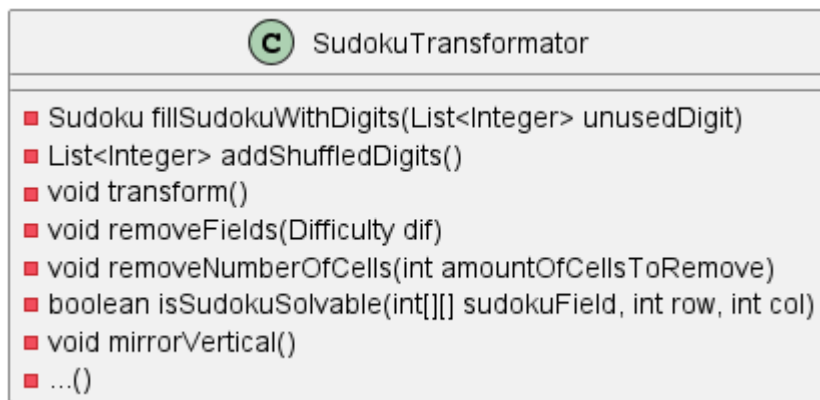
[CODE SMELL 2] – Large Class

Commit:

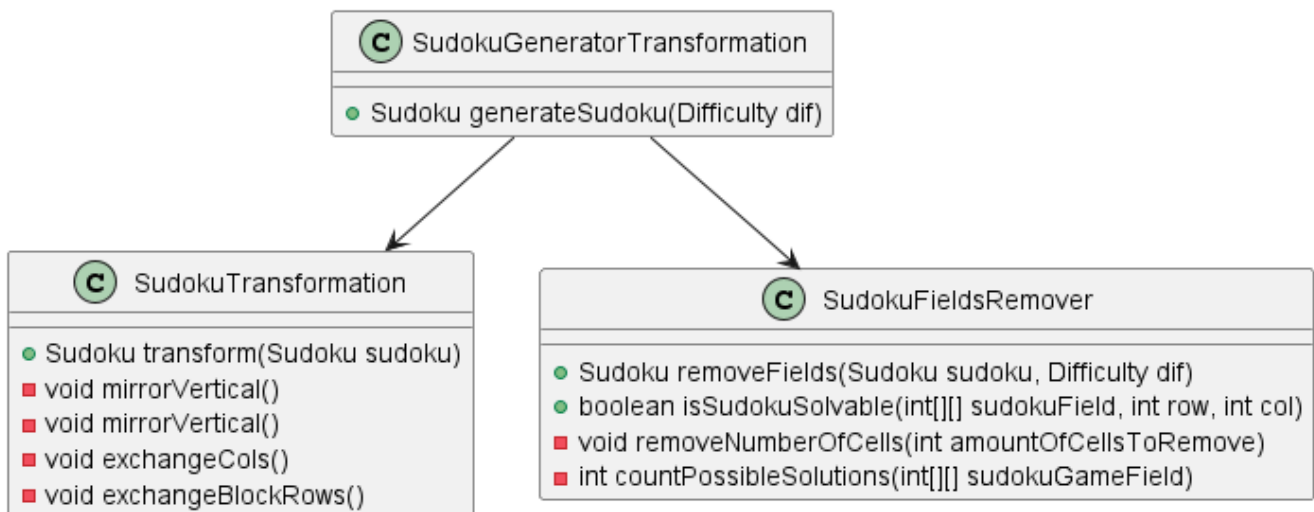
<https://github.com/mohjohfox/cli-sudoku/commit/b6a01380718b74ef77e0975aea87ba9f8c4ec053>

<https://github.com/mohjohfox/cli-sudoku/commit/d43576a649786b4174b1f7f9ce859b84ee7943c4>

Vorher:



Nachher:



Für die Generierung eines Sudokus durch Transformation gab es die Klasse **SudokuTransformator**. Diese hat einige Funktionen für die Transformation beinhaltet. Dadurch wurde die Klasse der groß.

```

public class SudokuGeneratorTransformation {

    public Sudoku generateSudoku(Difficulty difficulty) {
        SudokuTransformator sudokuTransformator = new SudokuTransformator();
        this.sudoku = sudokuTransformator.transform(this.sudoku);
        SudokuFieldsRemover sudokuFieldsRemover = new SudokuFieldsRemover();
        this.sudoku = sudokuFieldsRemover.removeFields(this.sudoku,dif);
        return sudoku;
    }

    private void removeFields(Difficulty dif){
        int amountOfCellsToRemove = amountOfCellsToRemove(dif);
        removeNumberOfCells(amountOfCellsToRemove);
    }

    private int amountOfCellsToRemove(Difficulty dif) {
        int amountOfCellsToRemove = switch (dif) {
            case EASY:
                yield 40;
            case MEDIUM:
                yield 50;
            case HARD:
                yield 60;
        };
        return amountOfCellsToRemove;
    }

    private void removeNumberOfCells(int amountOfCellsToRemove) {
        Random random = rand;
        for (int i = 0; i < amountOfCellsToRemove; i++) {
            int row = random.nextInt(9);
            int col = random.nextInt(9);

            if (this.sudoku.getGameField()[row][col] == 0) {
                i--;
                continue;
            }

            int temp = this.sudoku.getGameField()[row][col];
            this.sudoku.setField(row,col,0);
            int numSolutions = countPossibleSolutions(this.sudoku.getGameField());
            if (numSolutions != 1) {
                this.sudoku.setField(row,col,temp);
                i--;
            }
        }
    }

    ...
}

```

Durch ein Refactoring dieser Klasse wurden die Funktionen in einzelne Klassen weiter aufgeteilt. Zuerst wurde eine SudokuGeneratorTransformation Klasse eingeführt. Diese nutzte dann die ursprüngliche Transformator Klasse. Die ursprünglichen Funktionen wurden zwischen beiden Klassen aufgeteilt. Die neu eingeführte Klasse beinhaltete allerdings immer noch einige Funktionen und war sehr groß. Aus diesem Grund wurden nochmals eine weitere Klasse namens FieldsRemover eingeführt, welche sich um das strukturierte Entfernen von Felder im Spielfeld kümmert. Dadurch war sichergestellt, dass jede Klasse eine gewisse Funktion erfüllt und keine Klasse überfüllt ist.

Pseudocode für Lösung:

```
public class SudokuGeneratorTransformation {  
    public Sudoku generateSudoku(Difficulty dif) {  
        ....  
    }  
}  
  
public class FieldsRemover {  
    public removeFields(Sudoku sudoku, Difficulty dif)  
    removeNumberOfCells()  
}  
  
public class SudokuTransformator {  
    transform(Sudoku sudoku)  
    mirrorVertiacI()  
    exchangeCols()  
    exchangeBlockRows()  
}
```

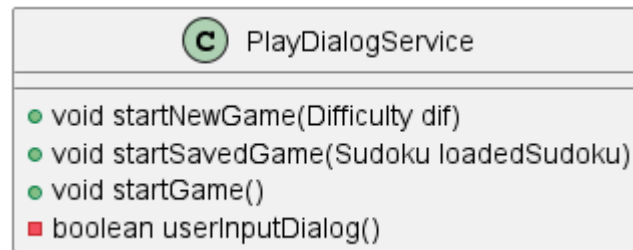
[CODE SMELL 3] – Long Method

Refactoring commits:

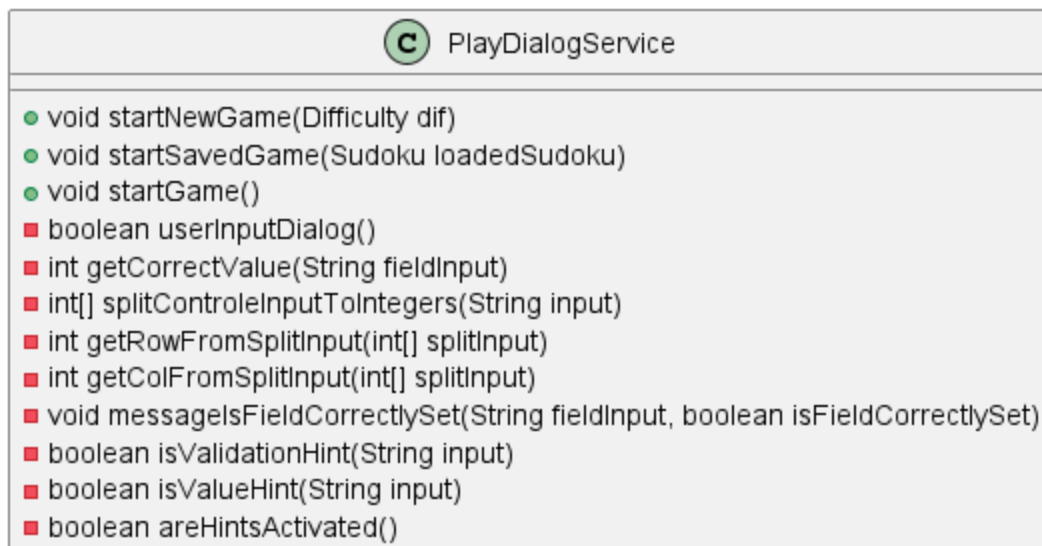
<https://github.com/mohjohfox/cli-sudoku/commit/e359a100d94f71ccd615d0bbdcba66936b10a89f>

<https://github.com/mohjohfox/cli-sudoku/commit/4738147c9426e0db1e6ef323c63690bb3d34c8d8>

Vorher:



Nachher:



```

private boolean userInputDialog() {
    String input = inputPort.getInput();
    while (!inputCorrect(input)) {
        outputPort.inputError();
        outputPort.possibleHints(settingService.getSetting());
        input = inputPort.getInput();
    }
    if (isAbortAction(input)) {
        sudokuPersistencePort.saveSudoku(sudoku);
        outputPort.gameSaved();
        return false;
    }
    if (isExitAction(input)) {
        return false;
    }
    if (isHintAction(input) && (settingService.getSetting().getValueHint() || settingService.getSetting().getFieldValidation())) {
        if (input.equalsIgnoreCase("H") && settingService.getSetting().getValueHint()) {
            outputPort.inputForSolvingField();
            String fieldInput = inputPort.getInput();
            while (!checkInputForSolvingField(fieldInput)) {
                outputPort.inputForSolvingField();
                fieldInput = inputPort.getInput();
            }
            String[] getAction = fieldInput.split(":");
            int[] splitInput = Arrays.stream(getAction[1].split(",")).mapToInt(Integer::parseInt).toArray();
            int row = splitInput[0] - 1;
            int col = splitInput[1] - 1;
            int correctValue = sudoku.getSolvedGameField().sudokuArray()[row][col];
            boolean isFieldCorrectlySet = sudoku.setField(row, col, correctValue);
            if (isFieldCorrectlySet) {
                outputPort.setCorrectField(row, col);
            } else {
                outputPort.defaultFieldError(getAction[1]);
            }
            return true;
        } else if (input.equalsIgnoreCase("V") && settingService.getSetting().getFieldValidation()) {
            List<String> notCorrectFields = sudokuValidator.crossCheck(
                sudoku.getGameField(),
                sudoku.getInitialGameField(),
                sudoku.getSolvedGameField());
            outputPort.notCorrectFields(notCorrectFields);
            return true;
        }
    }
    return true;
}

try {
    String[] getAction = input.split(":");
    int[] splitInput = Arrays.stream(getAction[1].split(",")).mapToInt(Integer::parseInt).toArray();
    boolean actionSuccessful = false;
    if (isWriteAction(getAction[0])) {
        actionSuccessful = sudoku.setField(splitInput[0] - 1, splitInput[1] - 1, splitInput[2]);
    }
    if (isRemoveAction(getAction[0])) {
        actionSuccessful = sudoku.setField(splitInput[0] - 1, splitInput[1] - 1, 0);
    }
    if (!actionSuccessful) {
        outputPort.defaultFieldError(getAction[1]);
    }
    return true;
} catch (IndexOutOfBoundsException e) {
    outputPort.inputError();
    return true;
}
}

```

Die Methode `userInputDialog()` kümmert sich um den Spieldialog während eines Spiels. Anfangs war sämtliche Logik in dieser Methode involviert.

Die einzelnen Überprüfungen wurden dann in einzelne Methoden ausgelagert. Dadurch ist eine einfachere Anpassung und vor allem übersichtlichere Darstellung vorhanden.

Pseudocode für Lösung:

```
private Boolean userInputDialog(){
    getCorrectValue()
    getRowFromSplitInput()
    getColFromSplitInput()
}

private int getCorrectValue(String fieldInput)
private int[] splitControleInputInteger(String input)
private int getRowFromSplitInput(int[] splitInput)
private int getColFromSplitInput(int[] splitInput)
private Boolean areHintsActivated()
private Boolean isValueHint(String input)
....
```

3 Refactorings (6P)

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

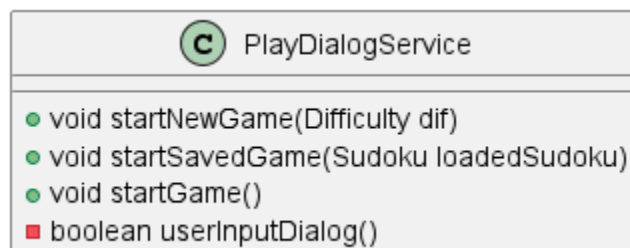
1. Extract Method

Commits:

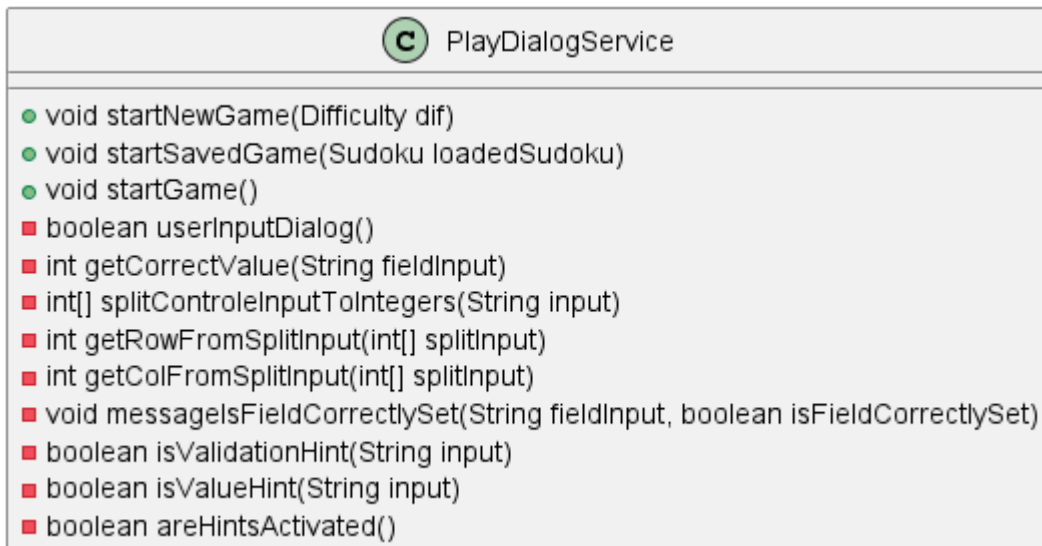
<https://github.com/mohjohfox/cli-sudoku/commit/e359a100d94f71ccd615d0bbdcba66936b10a89f>

<https://github.com/mohjohfox/cli-sudoku/commit/4738147c9426e0db1e6ef323c63690bb3d34c8d8>

Vorher:



Nachher:



Die Methode userInputDialog hat einige Überprüfungen drin, welche beispielsweise prüfen ob Hinweise aktiviert sind. Die einzelnen Überprüfungen und folgende Aktionen sind in jeweils einzelne Methode ausgelagert worden. Dadurch ist der Programmcode deutlich übersichtlicher und jeder Methodename beschreibt was genau passiert.

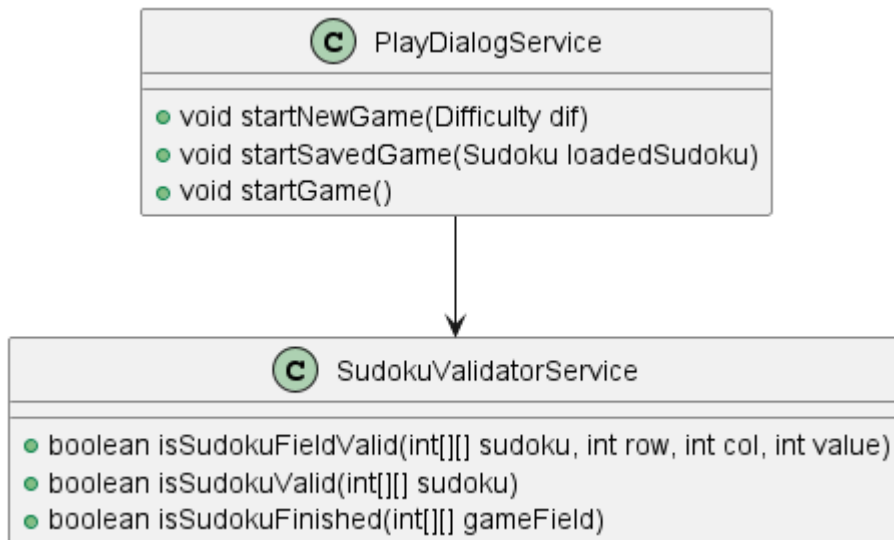
2. Rename Method

Commit:

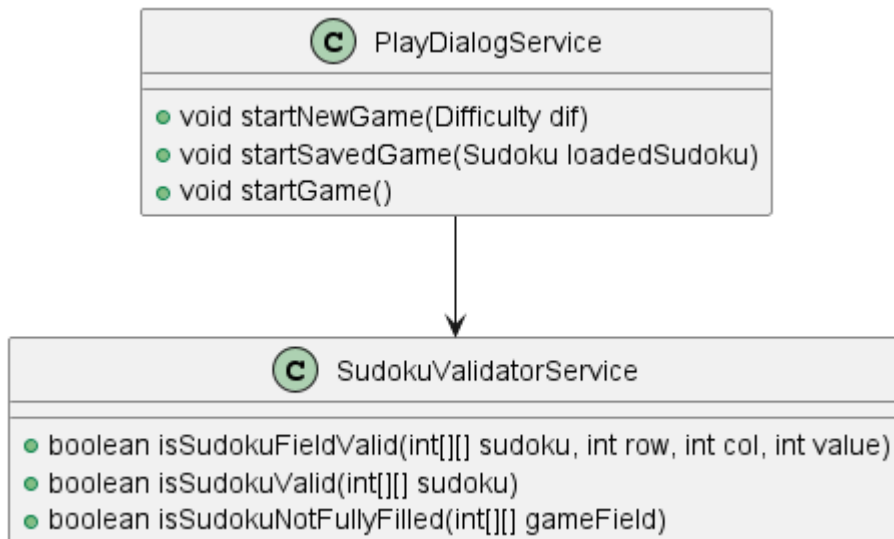
<https://github.com/mohjohfox/cli-sudoku/commit/c1a1c020983c141d73346a8bbd2f2940f61e2819>

<https://github.com/mohjohfox/cli-sudoku/commit/0ec45de57d696fe5f9fb5c3b675a266ce1dbd246#diff-95b9f300de8b48cc2e9e45e0ce43ec26859908aadfec88b05d236991daf9b63a>

Vorher:



Nachher:



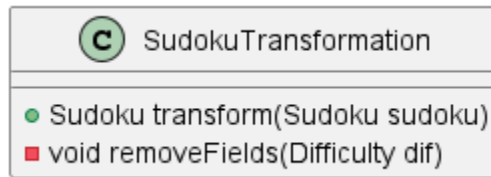
In diesem Beispiel wurde eine Methode namens `isSudokuFinished` implementiert. Diese prüft, ob ein Sudoku vollständig ausgefüllt ist. Sie wird verwendet, um zu prüfen, ob ein Spiel abgeschlossen ist oder nicht. Während der Entwicklung warf diese Methode Fragen auf, da fälschlicherweise gedacht wurde, dass die Methode prüft, ob ein Sudoku vollständig korrekt gelöst wurde. Um diese Unstimmigkeit zu beseitigen, wurde die Methode umbenannt zu `isSudokuNotFullyFilled`. Dadurch herrscht Klarheit und die Methode macht auch exakt das, was ihr Namen vermuten lässt.

3. Replace Temp with Query

Commit:

<https://github.com/mohjohfox/cli-sudoku/commit/7c5748a5c3a075f9ba7ae67c101a72c45ca0b34>

<https://github.com/mohjohfox/cli-sudoku/commit/85b236f90bbbc50a47aa926c810e7be4a170c043>



In diesem Beispiel wurde die Anzahl der zu entfernenden Felder im Spielfeld Inline in einem Switch-Statement ermittelt. Die lokale Variable wurde anschließend verwendet, um die Anzahl an Felder zu entfernen. Da die Berechnung in eine lokale Variable zugewiesen wurde, konnte diese theoretisch noch verändert werden. Um dies zu vermeiden, wurde der Teil überarbeitet. Nun findet die Berechnung der zu entfernenden Felder in einer Methode statt. Diese Methode wird dann direkt an der Stelle, an welcher die Felder entfernt werden, genutzt. Somit ist keine Manipulation mehr möglich.

Kapitel 8: Entwurfsmuster (8P)

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

Entwurfsmuster: [Builder] (4P)

Für das Erstellen eines Sudokuobjekts wird das Builder-Pattern genutzt. Nachfolgend sind die Beweggründe aufgezeigt:

1. Er vereinfacht die Erstellung von Sudoku-Objekten mit optionalen Parametern: Die Sudoku-Klasse hat mehrere Konstruktoren mit unterschiedlichen Parametern, wodurch die Erstellung von Sudoku-Objekten etwas umständlich werden kann, insbesondere wenn einige Parameter optional sind. Mit einem Builder können Sie die gewünschten Parameter einfach und flexibel festlegen, ohne den Aufruf eines spezifischen Konstruktors zu verwenden.
2. Er verbessert die Lesbarkeit des Codes: Die Verwendung eines Builders ermöglicht eine klare und aussagekräftige Syntax beim Erstellen von Sudoku-Objekten. Der Code wird lesbarer und leichter zu verstehen, da die einzelnen Methodenaufrufe des Builders die beabsichtigte Konfiguration der Sudoku-Instanz deutlich machen.
3. Er ermöglicht eine konsistente Erstellung von Objekten: Der Builder kann interne Überprüfungen und Validierungen durchführen, um sicherzustellen, dass die erstellten Sudoku-Objekte gültig und konsistent sind. Zum Beispiel könnte der Builder sicherstellen, dass das `initialSolvedGameField` nur einmal gesetzt werden kann und nicht überschrieben wird.
4. Er erleichtert zukünftige Änderungen und Erweiterungen: Wenn sich die Anforderungen an die Sudoku-Klasse ändern oder neue Optionen hinzugefügt werden sollen, ist es einfacher, den Builder anzupassen, anstatt die vorhandenen Konstruktoren zu ändern. Der Builder isoliert die Erstellungskomplexität und erleichtert die Wartung und Weiterentwicklung des Codes.



Entwurfsmuster: [Singleton] (4P)

Die Klasse `DependencyFactory` ist als Singleton implementiert, da sie eine Art `Dependency Injection-Framework` implementiert.

Das Singleton-Entwurfsmuster gewährleistet, dass nur eine einzige Instanz der Klasse `DependencyFactory` existiert und bietet einen globalen Zugriffspunkt auf diese Instanz über die statische Methode `getInstance()`.

Durch die Verwendung des Singleton-Entwurfsmusters kann sichergestellt werden, dass es nur eine einzige Instanz der `DependencyFactory` gibt, um eine konsistente Verwaltung der Abhängigkeiten zu gewährleisten. Jeder Aufruf von `getInstance()` gibt die gleiche Instanz zurück.

Die `DependencyFactory` ermöglicht das Registrieren von Abhängigkeiten über die Methode `registerDependency()` und das Abrufen von Abhängigkeiten über die Methode `getDependency()`. Damit kann man Objekte verschiedener Klassen registrieren und abrufen, indem man den gewünschten Typ angibt.

