

Kreativer Zufallsgenerator

Programmentwurf

für die Prüfung zum

Bachelor of Science

des Studienganges Informatik

an der Dualen Hochschule Baden-Württemberg Karlsruhe

von

Sarah Glatt u. Sinja Ohle

Abgabedatum

29.05.2022

Matrikelnummer

1994767 u. 3890129

Kurs

TINF19B2

Dozent

Lars Briem

Eidesstattliche Erklärung

gemäß § 5 (3) der „Studien- und Prüfungsordnung DHBW Technik“ vom 27.07.2020.

Wir versichern hiermit, dass wir den Programmentwurf mit dem Titel

„Kreativer Zufallsgenerator“

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben.

Ort, Datum

Sarah Glatt

Ort, Datum

Sinja Ohle

Inhaltsverzeichnis

Abbildungsverzeichnis	V
Tabellenverzeichnis	VII
Formelverzeichnis	VIII
Listingverzeichnis	IX
Abkürzungsverzeichnis	X
1 Einführung	1
1.1 Übersicht über die Applikation	1
1.2 Wie startet man die Applikation?	1
1.3 Wie testet man die Applikation?	2
2 Clean Architecture	3
2.1 Was ist Clean Architecture?	3
2.2 Analyse der Dependency Rule	3
2.3 Analyse der Schichten	7
3 SOLID	11
3.1 Analyse des S ingle- R esponsibility- P inciple	11
3.2 Analyse des O pen- C losed- P inciple	14
3.3 Analyse des LSP, ISP, D ependency- I nversion- P inciple	19
4 Weitere Prinzipien	24
4.1 Analyse der GRASP: Geringe Kopplung	24
4.2 Analyse der GRASP: Hohe Kohäsion	27
4.3 D on't R epeat Y ourself	28

5	Unit Tests	33
5.1	20 Unit Tests	33
5.2	ATRIP: Automatic	38
5.3	ATRIP: Thorough	38
5.4	ATRIP: Professional	40
5.5	Code Coverage	45
5.6	Fakes und Mocks	45
6	Domain Driven Design	48
6.1	Ubiquitous Language	48
6.2	Entities	49
6.3	Value Objects	49
6.4	Repositories	50
6.5	Aggregates	51
7	Refactoring	52
7.1	Code Smells	52
7.2	4 Refactorings	56
8	Entwurfsmuster	60
8.1	Entwurfsmuster: [Erzeugungsmuster]	60
8.2	Entwurfsmuster: [Strukturmuster]	61
8.3	Entwurfsmuster: [Verhaltensmuster]	61
8.4	Entwurfsmuster: [Verhaltensmuster]	63

Abbildungsverzeichnis

1.1	Maven clean insatll in IntelliJ	2
2.1	Dependency Rule - Controll/ CheckInput	4
2.2	Dependency Rule - Controll/ SearchElements/ Idea	5
2.3	Dependency Rule - Controll/ ManageElement - Aktueller Stand	5
2.4	Dependency Rule - Controll/ ManageElement - Verbessert	6
2.5	Dependency Rule - Jobs/ TxtHandling - Aktueller Stand	6
2.6	Dependency Rule - Jobs/ TxtHandling - Verbessert	7
2.7	Domain Code - Entity/	8
2.8	Domain Code - Controller/ ManageElementInterface & Entity/ CategoryInterface	8
2.9	Application Code - Use Case Controller/ SearchElements/ Idea	9
2.10	Application Code - Use Case Controller/ Element/	10
3.1	SRP -Controller/ Steuerung	11
3.2	SRP -Controller/ GUI	12
3.3	SRP -Controller/ ManageElement - Aktueller Stand	12
3.4	SRP -Controller/ ManageElement - Verbessert	13
3.5	SRP -Entity/ Tag - Aktueller Stand	13
3.6	SRP -Entity/ Tag - Verbessert	14
3.7	OCP -Entity/ Category	15
3.8	OCP -Controller/ SearchElements/ Filter	16
3.9	OCP -Controller/ Element/ AddElement - Aktueller Stand	17
3.10	OCP -Controller/ Element/ AddElement - Verbessert	17
3.11	OCP -Controller/ Element/ DeleteElement - Aktueller Stand	18
3.12	OCP -Controller/ Element/ DeleteElement - Verbessert	19
3.13	DIP -Controller/ SearchElements/ Filter	20
3.14	DIP -Entity/ Category	21
3.15	DIP -Controller/ ManageElement	22
3.16	DIP -Controller/ CheckInput	23

4.1	Geringe Kopplung -Jobs/ TxtHandling & Entity/ Category	25
4.2	Geringe Kopplung -Controller/ ManageElement & Entity/ Category .	25
4.3	Geringe Kopplung -Entity/ Tag	26
4.4	Hohe Kohäsion -Controller/ Element/	28
4.5	Hohe Kohäsion -Entity/ CategoryStatus	28
5.1	Professional - Testverzeichnis	42
5.2	Code Coverage vom kreativen Zufallsgenerator	45
5.3	Fakes & Mocks -test/ Controller/ Element/ AddElementTest	46
5.4	Fakes & Mocks -test/ Jobs/ EntityBuilderTest	46
5.5	Fakes & Mocks -test/ Controller/ CheckInputTest	47
5.6	Fakes & Mocks -test/ Controller/ ManageElementTest	47
6.1	Entities - Entity/ Tag	49
6.2	Value Objects - Entity/ Category	50
6.3	Aggregates - Entity/ Objekt	51
7.1	Code Smells - Controller/ Steuerung - Alter Stand	55
7.2	Code Smells - Controller/ Steuerung - Aktueller Stand	55
7.3	Refactoring - Jobs/ TxtReader - Alter Stand	57
7.4	Refactoring - Jobs/ Refactoring - Verbesselter Stand	57
7.5	Refactoring - Controller/ Element/ AddElement - Alter Stand	57
7.6	Refactoring - Controller/ Element/ AddElement - Verbesselter Stand .	58
7.7	Refactoring - Controller/ GUI - Alter Stand	58
7.8	Refactoring - Controller/ GUI - Verbesselter Stand	58
7.9	Refactoring - Entity/ Entity - Alter Stand	59
7.10	Refactoring - Entity/ Entity - Verbesselter Stand	59
8.1	Erzeugungsmuster - Jobs/ EntityBuilder	60
8.2	Erzeugungsmuster - Controller/ SearchElements/ Idea	61
8.3	Erzeugungsmuster - Controller/ SearchElements/ Filter	62
8.4	Erzeugungsmuster - Controller/ Element/ UpdateElement	63

Tabellenverzeichnis

6.1 Ubiquitous Language	48
-----------------------------------	----

Listings

4.1	Starke Kopplung - <code>getEntity</code>	27
4.2	DRY - Controller/ Element/ <code>UpdateElement</code> - Alter Stand	29
4.3	DRY - Controller/ Element/ <code>UpdateElement</code> - Aktueller Stand	30
4.4	DRY - Controller/ GUI - Alter Stand	31
4.5	DRY - Controller/ GUI - Aktueller Stand	32
5.1	Thourough - test/ Jobs/ <code>TxtHandlingTest</code>	39
5.2	Professional - Controller/ <code>Steuerung</code>	41
5.3	Professional - Jobs/ <code>EntityBuilder</code>	43
5.4	Professional - test/ Jobs/ <code>EntityBuilder</code>	44
7.1	Code Smells - Controller/ Element/ <code>Add-, Delete-, UpdateElement</code>	52
7.2	Code Smells - Jobs/ <code>TxtReader</code> - Alter Stand	53
7.3	Code Smells - Jobs/ <code>EntityBuilder</code> - Aktueller Stand	54
7.4	Code Smells - Controller/ Element/ - Alter Stand	56
7.5	Code Smells - Entity/ <code>CategoryStatus</code> - Aktueller Stand	56

Abkürzungsverzeichnis

ATRIIP	A utomatic T horough R epeatable I ndependent P rofessional
CRUD	C reate R ead U pdate D elete
DIP	D ependency- I nversion- P rinciple
DRY	D on't R epeat Y ourself
GRASP	G eneral R esponsibility A ssignment S oftware P atterns
ISP	I nterface- S egregation- P rinciple
LSP	L iskov- S ubstitution- P rinciple
OCP	O pen- C losed- P rinciple
SOLID	S ingle-Responsibility, O pen-Closed, L iskov- Substitution, I nterface-Segregation, D ependency- Inversion
SRP	S ingle- R esponsibility- P rinciple

1 Einführung

1.1 Übersicht über die Applikation

Der kreative Zufallsgenerator dient der kreativen Ideenfindung. Wenn der User gerne kreativ werden möchte, jedoch keinen Einfall hat, was dieser machen soll, hilft der Zufallsgenerator.

Der Zufallsgenerator kann Ideen liefern zu verschiedenen Arten, Stimmungen und Objekten. Der User kann frei eine Idee generieren lassen oder eigene Einschränkungen festlegen. Für noch bessere Suchbestimmungen, sind die Objekte mit zusätzlichen Tags versehen, nach welchen differenziert werden kann.

Die möglichen Tags sind vom Zufallsgenerator selbst vorgegeben. Alle anderen Eingabemöglichkeiten (Arten, Stimmungen und Objekte) können variabel vom User neu angelegt, bearbeitet oder gelöscht werden. Die Objekte sind zusätzlich durch den User mit Tags zu versehen.

Der Benutzer kann durch Eingaben in der Konsole in festgelegter Reihenfolge die Applikation nutzen und Eingaben zur Steuerung des Programmablaufs tätigen.

1.2 Wie startet man die Applikation?

Voraussetzungen

1. SDK: openjdk-17, java version „17.0.2“
2. Language Level: Level 11 - Local variable syntax for lambda parameters
3. Maven Version: Apache Maven 3.8.4
4. installierte Entwicklungsumgebung (entwickelt mit IntelliJ)

Starten der Applikation

1. GitHub Projekt lokal clonen (`git clone <path>`)
2. Programm in Entwicklungsumgebung öffnen
3. Maven clean install (`mvn clean install`)

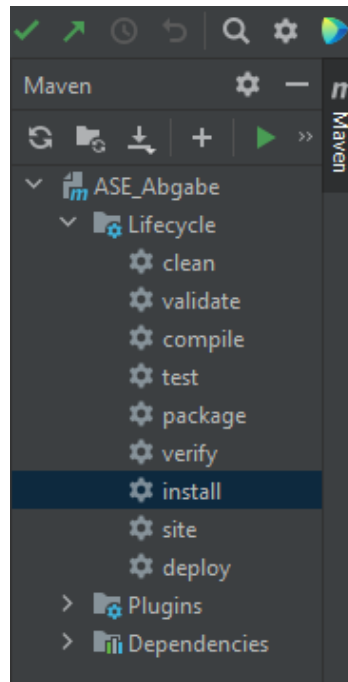


Abbildung 1.1: Maven clean insatll in IntelliJ

4. Controller/ Steuerung öffnen und Main-Methode ausführen
5. Console öffnet sich automatisch
6. Viel Spaß mit dem kreativen Zufallsgenerator

1.3 Wie testet man die Applikation?

1. Maven Version: Apache Maven 3.8.4 ggf. installieren
2. Alle Tests: den Befehl `mvn test` in der Komandozeile ausführen
3. Einzelner Test: Test öffnen und über `Run` durchlaufen lassen

2 Clean Architecture

2.1 Was ist Clean Architecture?

Clean Architecture beschreibt den Aufbau des Sourcecodes, dass langlebiger Code im Zentrum steht und kurzlebiger Code einfach und unkompliziert ausgetauscht werden kann. Die fachliche Anwendung soll unabhängig von der restlichen Infrastruktur getestet und weiterentwickelt werden können.

Hierfür wird die Metapher der Onion Architecture genutzt. Dabei wird der Code in verschiedene Schichten eingeteilt. Es gilt, je weiter innen, desto langlebiger der Code. Die Abhängigkeiten zeigen von außen nach innen.

2.2 Analyse der Dependency Rule

Abhängigkeiten zwischen Klassen werden mittels Pfeile dargestellt. Hierbei wird zwischen Abhängigkeitspfeilen und Aufrufpfeilen differenziert. Abhängigkeitspfeile zeigen von außen nach innen zeigen, dass Klasse A Klasse B benötigt, um compilieren zu können. Aufrufpfeile können in beide Richtungen zeigen. Hierbei erhält Klasse A die Referenz zu B erst während der Laufzeit.

2.2.1 Positiv-Beispiel 1: Dependency Rule

`Controller/ CheckInput` ist ein positives Beispiel für die Dependency Rule. Die Klasse `Controller/ GUI` ist abhängig von `Controller/ CheckInput` und liegt in der Onion Architecture weiter außen, da diese die Schnittstelle zum Benutzer beinhaltet, welche kurzlebiger ist. `Controller/ CheckInput` ist von `Controller/ ManageElement` abhängig. Diese beinhaltet die zentrale Logik der Sammlung von `Categories`, wodurch sich diese

weiter innen befindet.

Controller/ SearchElements/ SearchElements, Controller/ SearchElements/ Idea und Controller/ Element/ HandlingElement rufen Controller/ CheckInput auf. (UML 2.1)

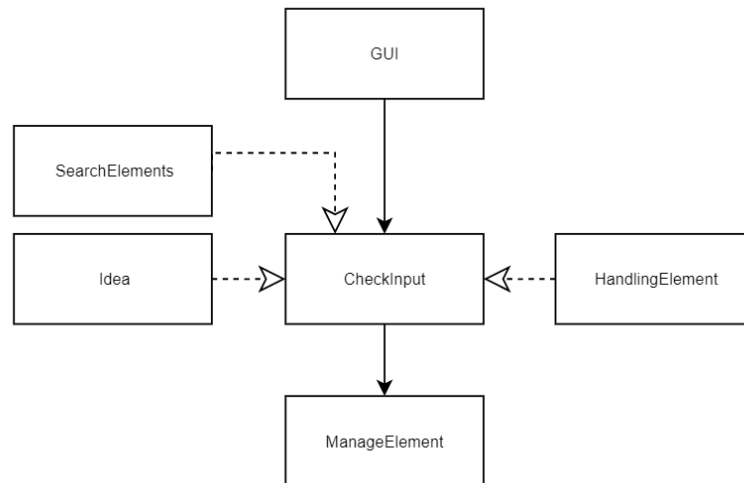


Abbildung 2.1: Dependency Rule - Controller/ CheckInput

2.2.2 Positiv-Beispiel 2: Dependency Rule

Controller/ SearchElements/ Idea ist ein weiteres positives Beispiel für die Dependency Rule. Keine Klasse ist abhängig von Controller/ SearchElements/ Idea. Diese selbst liegt Mittig in der Architecture. Controller/ SearchElements/ Idea ist von Controller/ ManageElement abhängig. Diese beinhaltet die zentrale Logik der Sammlung von Categories, wodurch sich diese weiter innen befindet.

Controller/ SearchElements/ Idea ruft Controller/ CheckInput und Controller/ SearchElements/ SearchElements auf. Letztere ruft Controller/ GUI auf und Koordiniert den Datenstrom zwischen der Controller/ SearchElements/ Idea und Controller/ GUI. (UML 2.2)

2.2.3 Negativ-Beispiel 1: Dependency Rule

Controller/ ManageElement kann die Dependency Rule nicht erfüllen. Die Abhängigkeiten von Controller/ CheckInput, Controller/ SearchElements/ SearchElements, Controller/

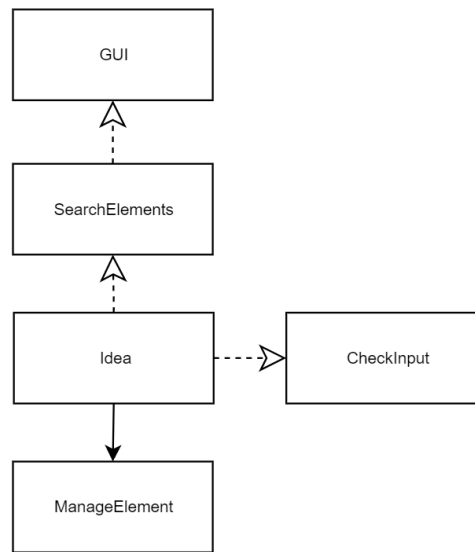


Abbildung 2.2: Dependency Rule - Controll/ SearchElements/ Idea

SearchElements/ Idea und Controller/ Element/ HandlingElement, welche auf Controller/ ManageElement verweisen passen zur Rule. Jedoch ist Controller/ ManageElement von Jobs/ EntityBuilder abhängig. Diese Klasse liegt jedoch weiter außen in der Onion Architecture, da sie die Elemente der **Categories** einließt und mit der Speicherung eng verknüpft ist. Die Speicherung ist kurzlebig, weswegen Jobs/ EntityBuilder nach außen gehört. Da die Abhängigkeitspfeile von außen nach innen zeigen, ist die Rule nicht erfüllt. Des Weiteren ruft Controller/ ManageElement Jobs/ TxtHandling auf. (UMLs 2.3 & 2.4)

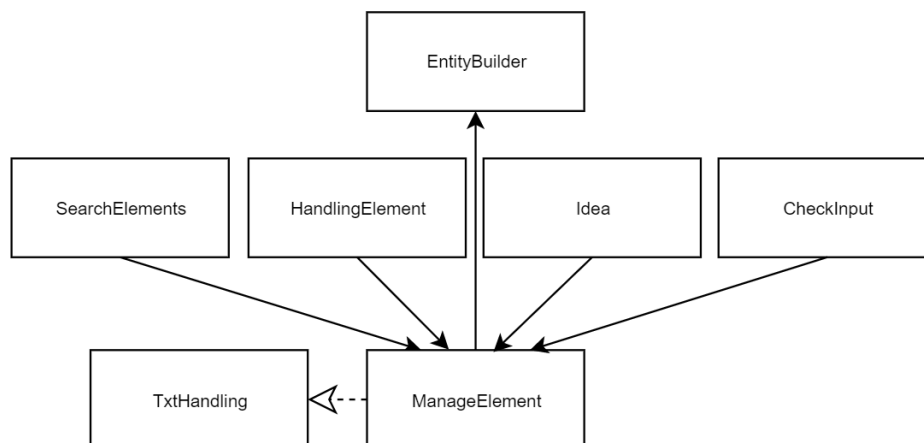


Abbildung 2.3: Dependency Rule - Controll/ ManageElement - Aktueller Stand

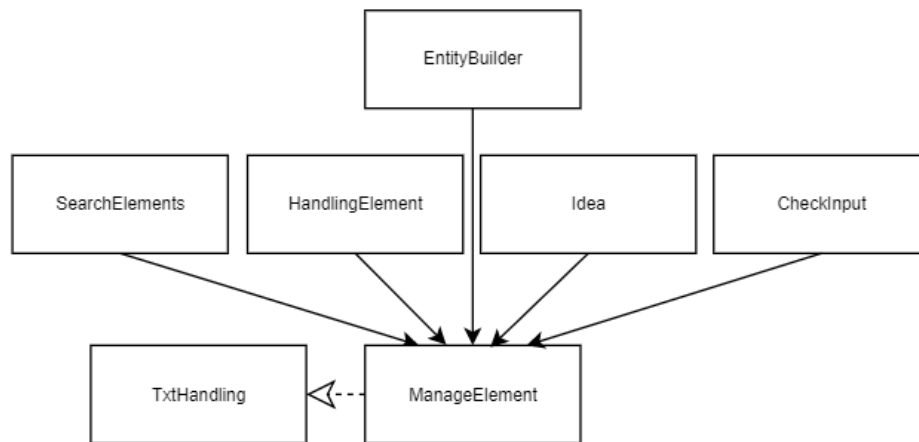


Abbildung 2.4: Dependency Rule - Controll/ ManageElement - Verbessert

2.2.4 Negativ-Beispiel 2: Dependency Rule

Jobs/ TxtHandling ist von keiner anderen Klasse abhängig und es ist ebenfalls keine Klasse von ihr abhängig. Jobs/ TxtHandling arbeitet direkt mit einer Text-Datei in der die Daten gespeichert werden. Zum Aufbereiten der Daten beim Einlesen wird Jobs/ EntityBuilder benötigt. Deswegen sollte Jobs/ TxtHandling auf Jobs/ EntityBuilder zeigen. Da dies nicht der Fall ist, ist die Dependency Rule nicht erfüllt.

Die Verbindung wird über einen Aufruf von Jobs/ EntityBuilder zu Jobs/ TxtHandling realisiert. Ebenfalls ruft Controller/ ManageElement die Klasse auf. (UMLs 2.5 & 2.6)

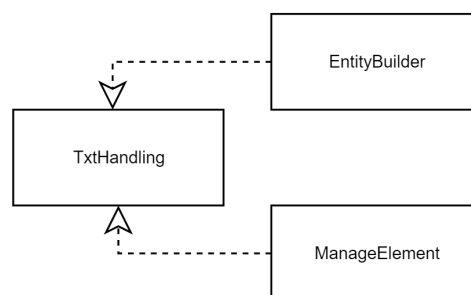


Abbildung 2.5: Dependency Rule - Jobs/ TxtHandling - Aktueller Stand

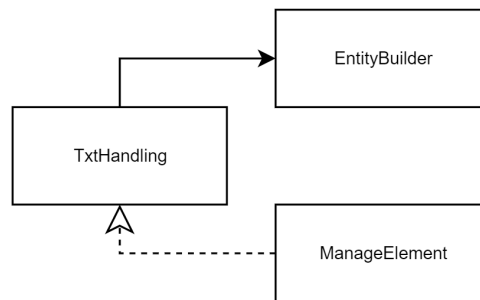


Abbildung 2.6: Dependency Rule - Jobs/ TxtHandling - Verbessert

2.3 Analyse der Schichten

2.3.1 Schicht: Domain Code

Der Domain Code sollte sich am seltensten ändern und Immun sein gegenüber Änderungen an der Anzeige, Speicherung u.ä. Zum Domain Code gehören unter anderem Entities und organisationsweit gültige Geschäftslogik.

Beispiel 1

Die Klassen und Enums des Ordners **Entity** gehören zu der Schicht des Domain Codes. Wie der Name besagt, liegen hier die **Entities** des Projektes hinterlegt, welche die Bedingungen für die weitere Verarbeitungen festlegen. (UML 2.7)

Beispiel 2

Die Interface **Entity/CategoryInterface** und **Controller/ManageElementInterface** liegen ebenfalls im Domain Code. Da diese die grundlegendsten Strukturen vorgeben und projektweite Gültigkeit haben. (UML 2.8)

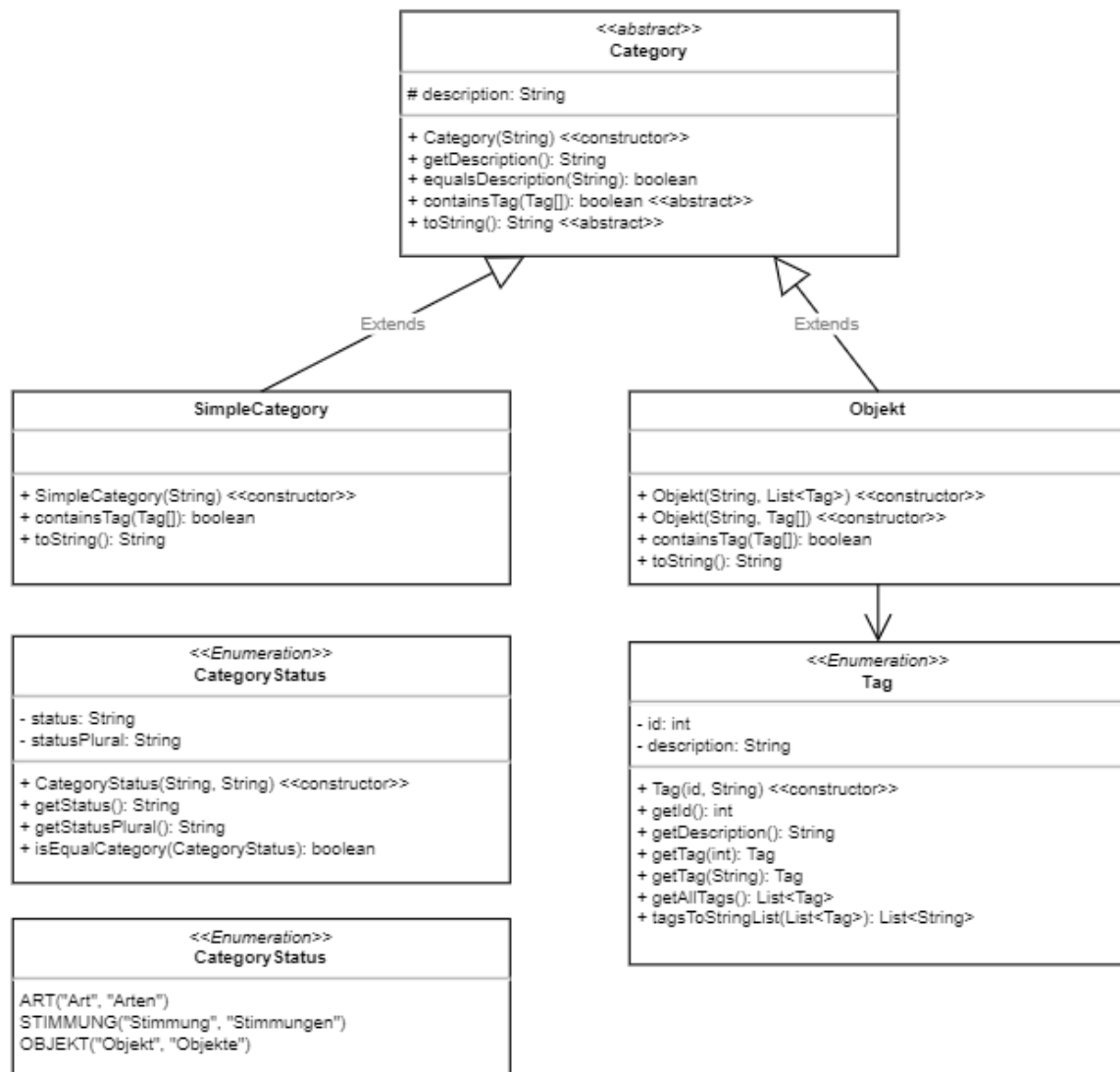


Abbildung 2.7: Domain Code - Entity/

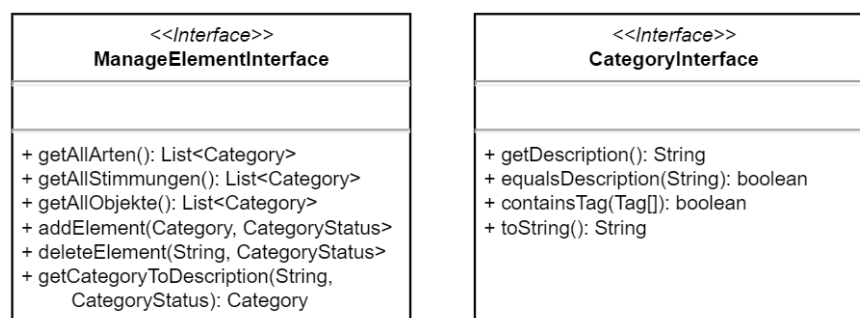


Abbildung 2.8: Domain Code - Controller/ ManageElementInterface & Entity/ CategoryInterface

2.3.2 Schicht: Application Code

Der Application Code beinhaltet Use Cases und implementiert die anwendungsspezifische Geschäftslogik. Zusätzlich steuert die Schicht den Fluss der Daten von und zu den Entities.

Beispiel 1

Die Klassen `Controller/ SearchElements/ FilterIdea` und `Controller/ SearchElements/ FilterObjektIdea` gehören in die Schicht Application Code. Sie beinhalten den Use Case die Bestandteile einer Idee (`Controller/ SearchElements/ Idea`) nach den gegebenen Bedingungen zu filtern. Nach der Prämisse, dass Use Cases zu dem Application Code gehören, liegen die Klassen in dieser Schicht. (UML 2.9)

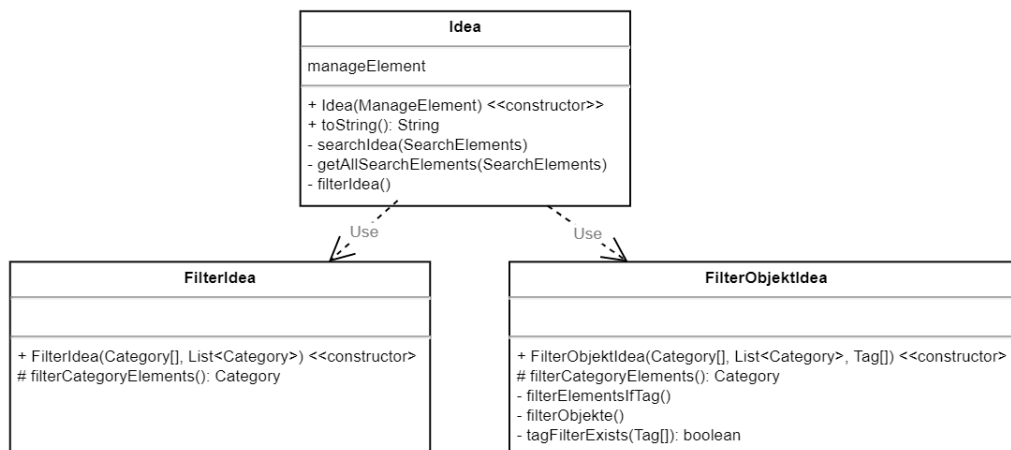


Abbildung 2.9: Application Code - Use Case `Controller/ SearchElements/ Idea`

Beispiel 2

Die Klassen `Controller/ Element/ AddElement`, `Controller/ Element/ DeleteElement` und `Controller/ Element/ UpdateElement` gehören in die Schicht Application Code. Sie beinhalten das **Create Read Update Delete (CRUD)**-Prinzip neue Elemente hinzufügen, bearbeiten und löschen zu können. CRUD ist ein Use Case, wodurch diese Klassen zu dem Application Code gehören. (UML 2.10)

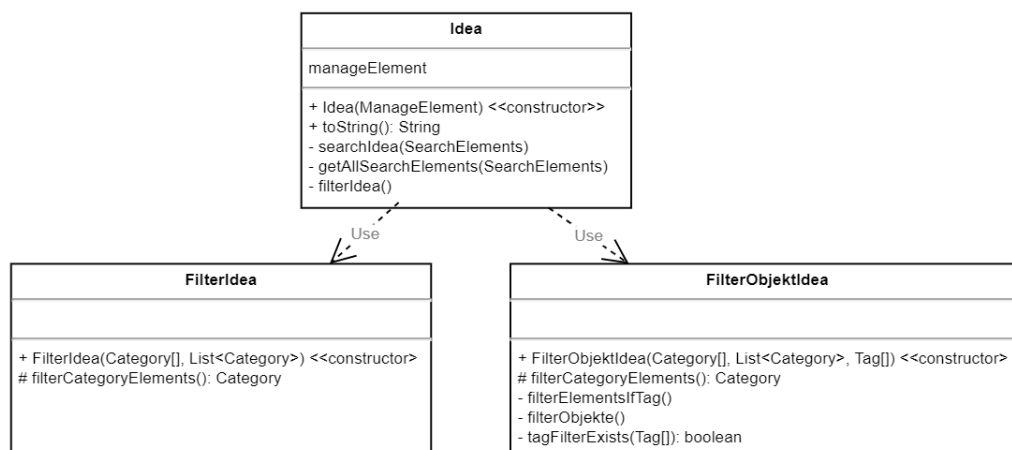


Abbildung 2.10: Application Code - Use Case Controller/ Element/

3 SOLID

3.1 Analyse des Single-Responsibility-Principle

Jede Klasse sollte genau eine Aufgabe erfüllen.

3.1.1 Positiv-Beispiel 1: SRP

Die Klasse **Controller/ Steuerung** besitzt die Aufgabe den Programmablauf zu regeln. Dazu gehört das bereitstellen der **Main** Methode und das entsprechende Aufrufen anderer Klassen, um weiterführende Aufgaben zu verteilen. (UML 3.1)

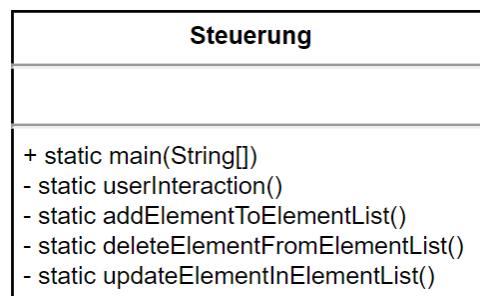


Abbildung 3.1: SRP -Controller/ Steuerung

3.1.2 Positiv-Beispiel 2: SRP

Die Klasse **Controller/ GUI** besitzt die Aufgabe die Kommunikation mit dem Benutzer zu koordinieren. Sie bildet die Schnittstelle zum Benutzer. Dazu zählt das Ausgeben Botschaften und Einlesen von Benutzerinformationen. Das Kontrollieren von Eingaben oder Verarbeiten dieser wird in anderen Klassen erledigt. (UML 3.2)

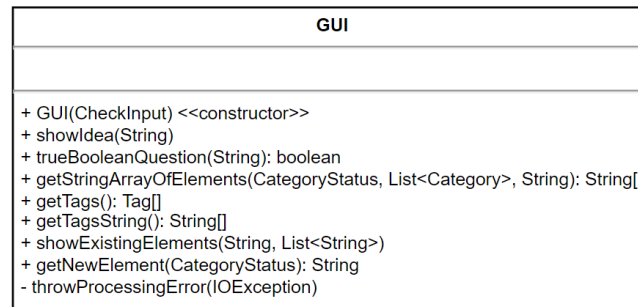


Abbildung 3.2: SRP -Controller/ GUI

3.1.3 Negativ-Beispiel 1: SRP

Die Klasse `Controller/ ManageElement` verwaltet die Gesamtheit aller `Categories`. Dazu gehört das Speichern der `Arten`, `Stimmungen` und `Objekte`. Die Elemente der `Categories` können hinzugefügt und gelöscht werden (bearbeiten ist eine Kombination aus löschen und hinzufügen). Diese werden in der Klasse selbst hinterlegt und mit einer Txt-Datei synchronisiert. Die Verwaltung der Txt-Datei regelt eine andere Klasse. Zusätzlich kann `Controller/ ManageElement` die `Category`-Listen in Arrays und String-Listen umwandeln.

Insgesamt erfüllt die Klasse dadurch mehr als eine Aufgabe. Um dies zu Lösen, kann eine zusätzliche `Helper` Klasse hinzugefügt werden. Diese könnte die Methoden `toArray(elementsList: List<Category>): Category[]` und `toStringList(elementsList: List<Category>): List<String>` beinhalten. (UMLs 3.3 & 3.4)

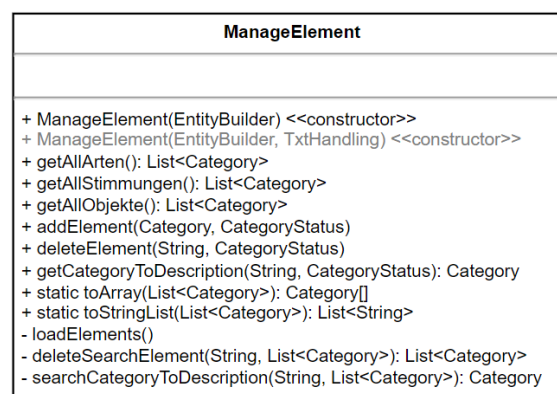
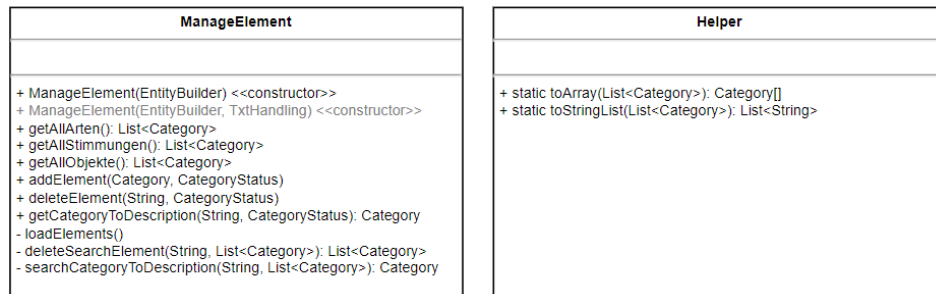


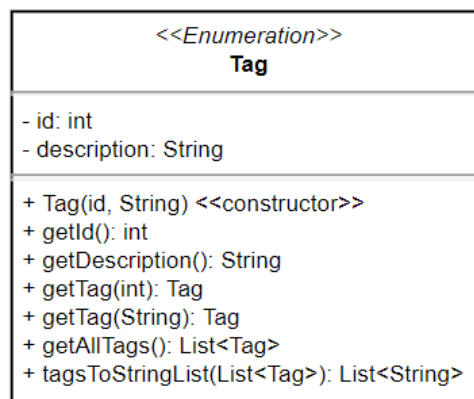
Abbildung 3.3: SRP -Controller/ ManageElement - Aktueller Stand

Abbildung 3.4: SRP -Controller/ **ManageElement** - Verbessert

3.1.4 Negativ-Beispiel 2: SRP

Das Enum **Entity/ Tag** verwaltet fixe Variablen von Tags. Diese können über die Attribute **description** oder **id** gefunden und zurückgegeben werden. Ebenfalls verfügt **Entity/ Tag** über eine Methode, um alle existierenden **Tags** zu erhalten.

Das Enum kann ebenfalls eine **Tag**-Liste in eine String-Liste umwandeln. Äquivalent zum ersten Negativ-Beispiel, gehört diese Methode nicht direkt zu der Aufgabe von **Entity/ Tag** und könnte in eine **Helper** Klasse ausgelagert werden. Gegebenenfalls auch die gleiche Klasse wie in vorherigem Beispiel. (UMLs 3.5 & 3.6)

Abbildung 3.5: SRP -Entity/ **Tag** - Aktueller Stand

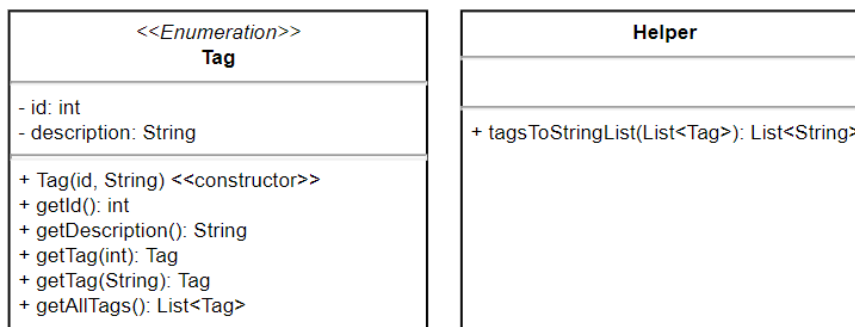


Abbildung 3.6: SRP -Entity/ Tag - Verbessert

3.2 Analyse des Open-Closed-Principle

Offen für Erweiterungen, geschlossen für Änderungen.

3.2.1 Positiv-Beispiel 1: OCP

Die Klasse **Entity/ Category** ist ein Beispiel für das OCP. Es können leicht Erweiterungen durchgeführt werden, in dem neu Klassen von **Entity/ Category** erben und diese um neue Methoden erweitern oder vorhandene Methoden überschreiben. So existieren bereits die Klassen **Entity/ SimpleCategory** und **Entity/ Objekt**. Wird eine neue **Category** benötigt, kann diese äquivalent angelegt werden.

Eine Änderung der Klasse ist dagegen umständlich, da das Interface **Entity/ CategoryInterface** verändert werden muss.

Der Einsatz des OCP bei **Entity/ Category** ist sinnvoll, da die **Categories** von vielen Klassen aufgerufen werden. Dadurch ist eine konstante und zuverlässige Arbeitsweise relevant. Ebenfalls sind die **Categories** ein zentraler Bestandteil des Zufallsgenerators. Nach diesen können neue Elemente angelegt, gesucht und gefiltert werden. Eine mögliche Erweiterung ist nicht ausgeschlossen und kann so leicht realisiert werden. (UML 3.7)

3.2.2 Positiv-Beispiel 2: OCP

Die Klasse **Controller/ SearchElements/ Filter** ist ein weiteres Beispiel für das OCP. Es können leicht Erweiterungen des **Filters** durchgeführt werden, in dem neu Klassen von **Controller/ SearchElements/ Filter** erben und diese um neue Methoden

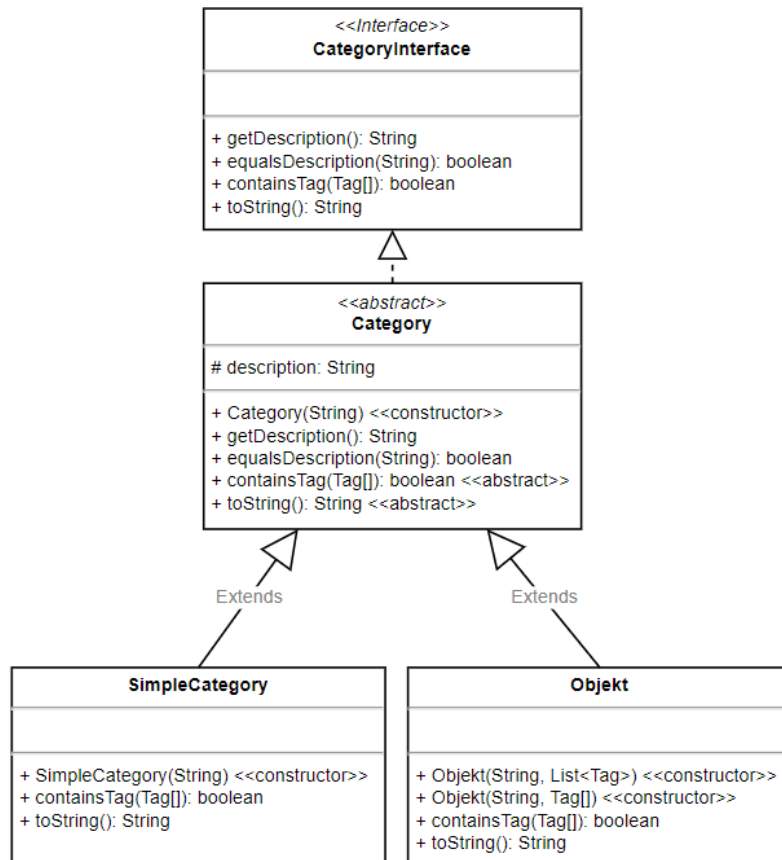


Abbildung 3.7: OCP -Entity/ Category

erweitern oder vorhandene Methoden überschreiben. So existieren bereits die Klassen **Controller/ SearchElements/ FilterIdea** und **Controller/ SearchElements/ FilterObjektIdea**. Wird ein neuer **Filter** benötigt, kann dieser äquivalent angelegt werden.

Eine Änderung der Klasse ist dagegen umständlich, da das Interface **Controller/ SearchElements/ FilterInterface** verändert werden muss.

Der Einsatz des OCP bei **Controller/ SearchElements/ Filter** ist sinnvoll, da dieser mit den **Categories** zusammenarbeitet und abhängig von diesen eine andere Verarbeitungsweise benötigt. Durch das OCP kann der **Filter** parallel zu den **Categories** erweitert werden. Dennoch ändert sich nichts für die **Controller/ SearchElements/ Idea** Klasse, welche mit den **Filtern** arbeitet. (UML 3.8)

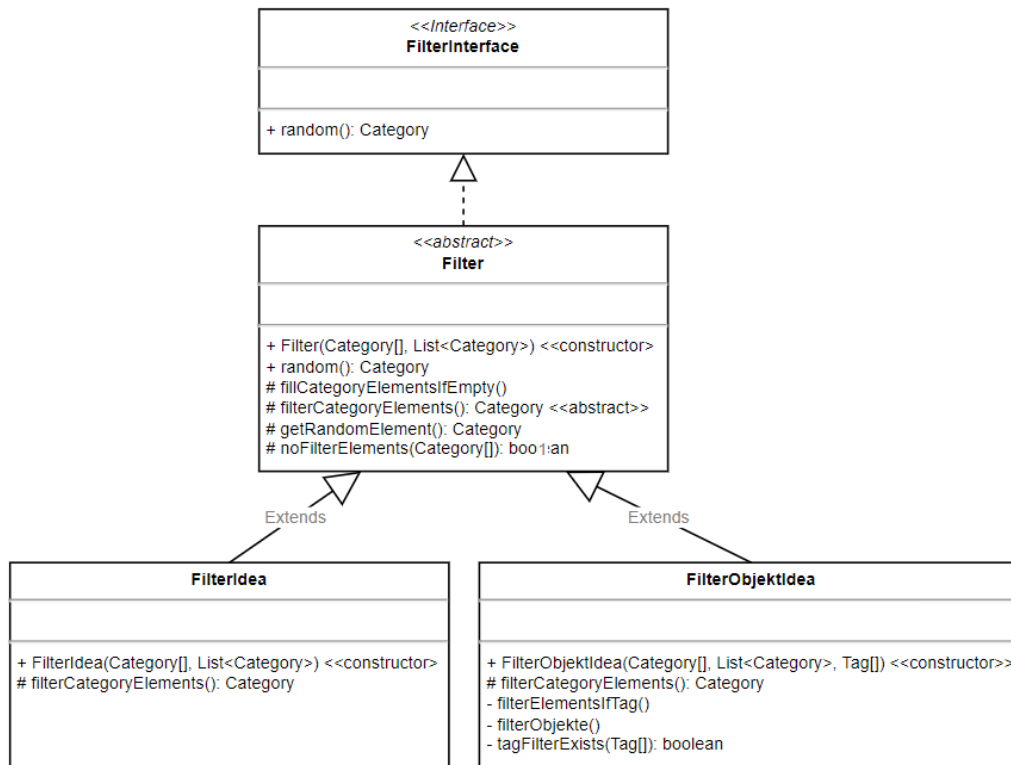


Abbildung 3.8: OCP -Controller/ SearchElements/ Filter

3.2.3 Negativ-Beispiel 1: OCP

Die Klasse **Controller/ Element/ AddElement** erfüllt nicht das OCP. Die Klasse kann nicht wirklich leicht erweitert werden. So existiert kein Interface, welches notwendige Methoden für Erweiterungen vorgibt. Auch die abstrakte Superklasse **Controller/ Element/ HandlingElement** enthält lediglich eine protected Methode, welche bei einem Aufruf von außen keine Rolle spielt.

Des Weiteren ist durch die nicht definierten Strukturen eine Änderung relativ einfach möglich. Auch wenn diese einen hohen Aufwand bei Anpassungen von bekannten Klassen bedeutet.

Um das OCP an dieser Stelle einzuführen, kann für die **Controller/ Element/ AddElement** Klasse ein Interface (**Controller/ Element/ AddElementInterface**) eingeführt werden. Dieses ermöglicht einen festen Rahmen, auf welchem andere Klassen arbeiten können. (UMLs 3.9 & 3.10)

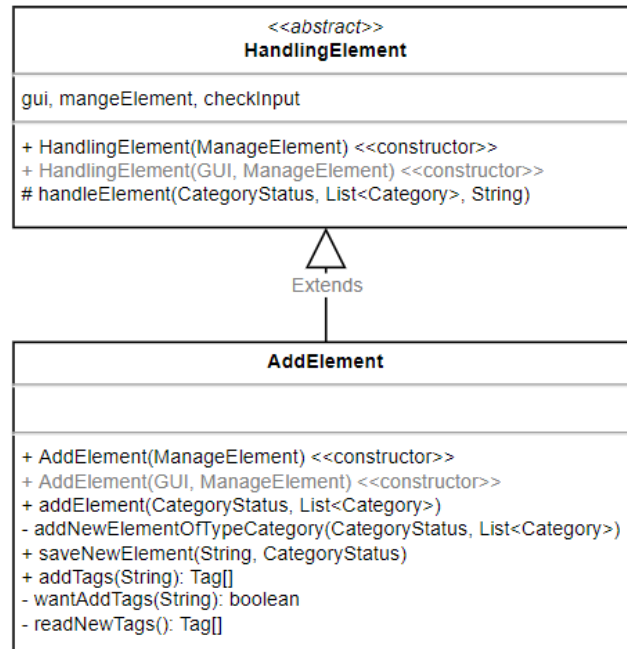


Abbildung 3.9: OCP -Controller/ Element/ AddElement - Aktueller Stand

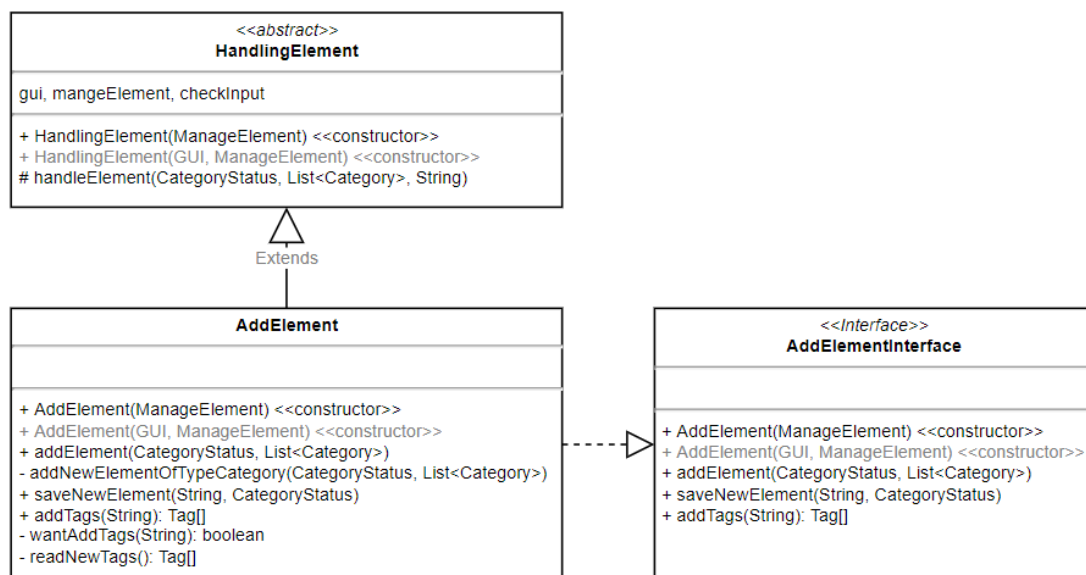


Abbildung 3.10: OCP -Controller/ Element/ AddElement - Verbessert

3.2.4 Negativ-Beispiel 2: OCP

Die Klasse **Controller/ Element/ DeleteElement** erfüllt ebenfalls nicht das OCP. Die Klasse kann nicht wirklich leicht erweitert werden. So existiert kein Interface, welches notwendige Methoden für Erweiterungen vorgibt. Auch die abstrakte Superklasse **Controller/ Element/ HandlingElement** enthält lediglich eine protected Methode, welche bei einem Aufruf von außen keine Rolle spielt.

Des Weiteren ist durch die nicht definierten Strukturen eine Änderung relativ einfach möglich. Auch wenn diese einen hohen Aufwand bei Anpassungen von bekannten Klassen bedeutet.

Um das OCP an dieser Stelle einzuführen, kann für die **Controller/ Element/ DeleteElement** Klasse ein Interface (**Controller/ Element/ DeleteElementInterface**) eingeführt werden. Dieses ermöglicht einen festen Rahmen, auf welchem andere Klassen arbeiten können. (UMLs 3.11 & 3.12)

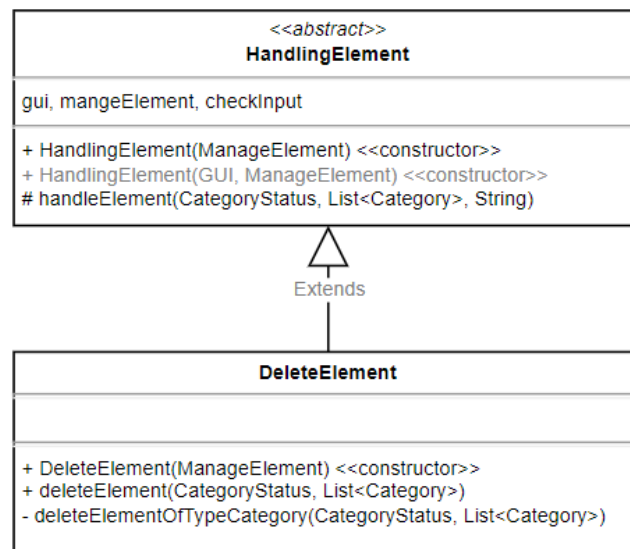


Abbildung 3.11: OCP -Controller/ Element/ DeleteElement - Aktueller Stand

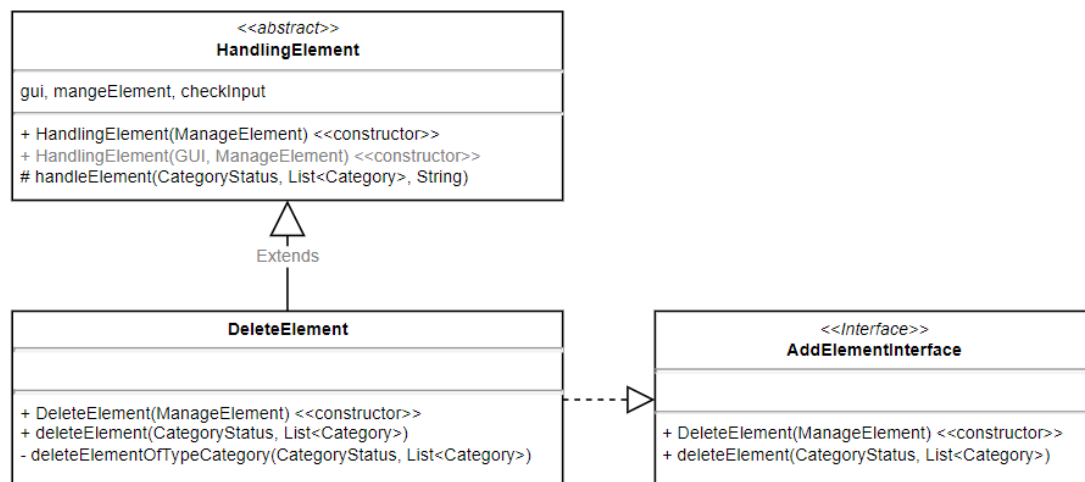


Abbildung 3.12: OCP -Controller/ Element/ DeleteElement - Verbessert

3.3 Analyse des LSP, ISP, **Dependency-Inversion-Principle**

1. LSP: Eine abgeleitete Klasse soll an jeder Stelle ihre Basisklasse ersetzen können, ohne, dass es zu unerwünschten Nebeneffekten kommt.
2. ISP: Viele Client-spezifische Interfaces sind besser als ein Allgemeines.
3. DIP: Klassen sollen von abstrakten Klassen abhängen und nicht von allgemeineren Klassen.

3.3.1 Positiv-Beispiel 1: DIP

Die Klassen `Controller/ SearchElements/ Filter`, `Controller/ SearchElements/ FilterIdea` und `Controller/ SearchElements/ FilterOnjektIdea` erfüllen das DIP. So erben letztere von der `Controller/ SearchElements/ Filter` Klasse, welche selbst abstrakt ist. (UML 3.13)

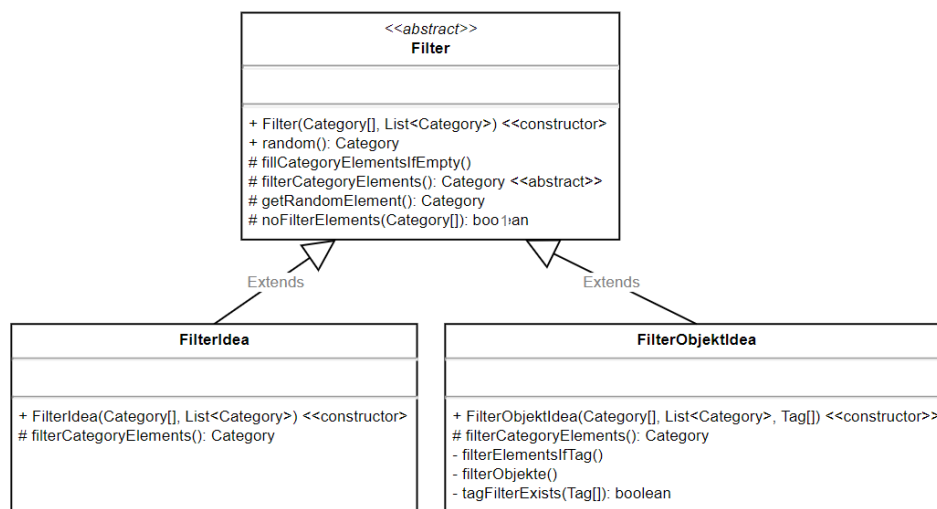


Abbildung 3.13: DIP -Controller/ SearchElements/ Filter

3.3.2 Positiv-Beispiel 2: DIP

Die Klassen **Entity/ Category**, **Entity/ SimpleCategory** und **Entity/ Objekt** erfüllen ebenfalls das DIP. So erben letztere von der **Entity/ Category** Klasse, welche selbst abstrakt ist. (UML 3.14)

3.3.3 Positiv-Beispiel 3: DIP

Die Klasse **Controller/ ManageElement** ist von dem Interface **Controller/ ManageElementInterface** abhängig, welches selbst abstrakt ist. (UML 3.15)

3.3.4 Positiv-Beispiel 4: DIP

Die Klasse **Controller/ CheckInput** ist von dem Interface **Controller/ CheckInputInterface** abhängig, welches selbst abstrakt ist. (UML 3.16)

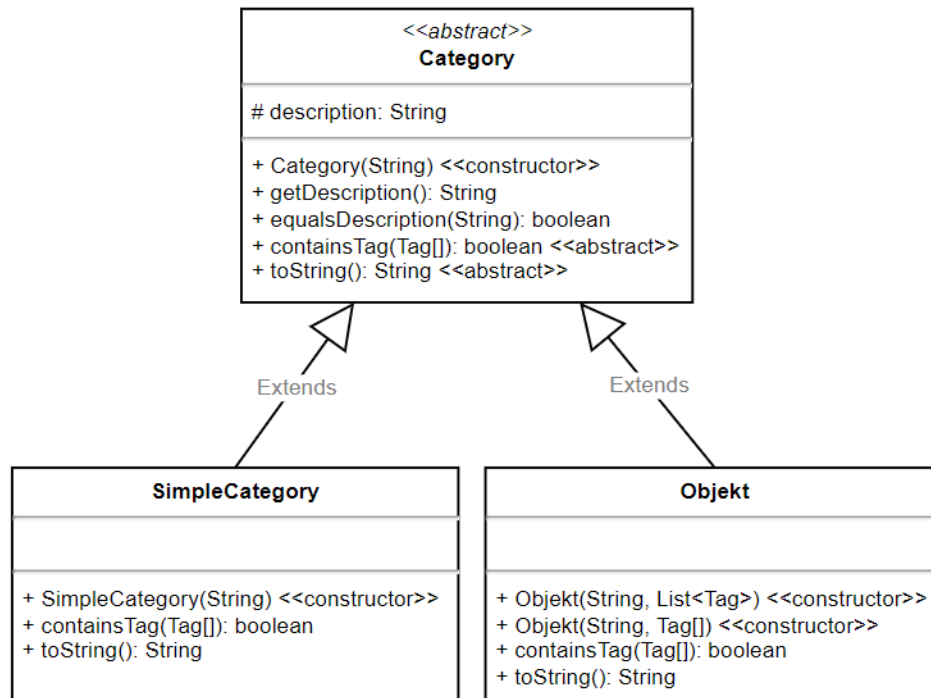


Abbildung 3.14: DIP -Entity/ Category

3.3.5 Negativ-Beispiele: DIP

Alle Klassen in dem Projekt sind entweder von abstrakten Klassen und oder Interfaces abhängig. Eine Vererbung zwischen zwei erzeugbaren Klassen liegt nicht vor. Die Klassen ohne weiter Abhängigkeiten sind die Enums **Entity/ CategoryStatus** und **Entity/ Tag**. Auch die Klasse **Controller/ Element/ HandlingElement** ist von keiner weiteren Klasse oder Interface abhängig. Dafür ist diese selbst abstrakt.

Deswegen verfügt das Projekt über keine Negativ-Beispiele zu dem DIP.

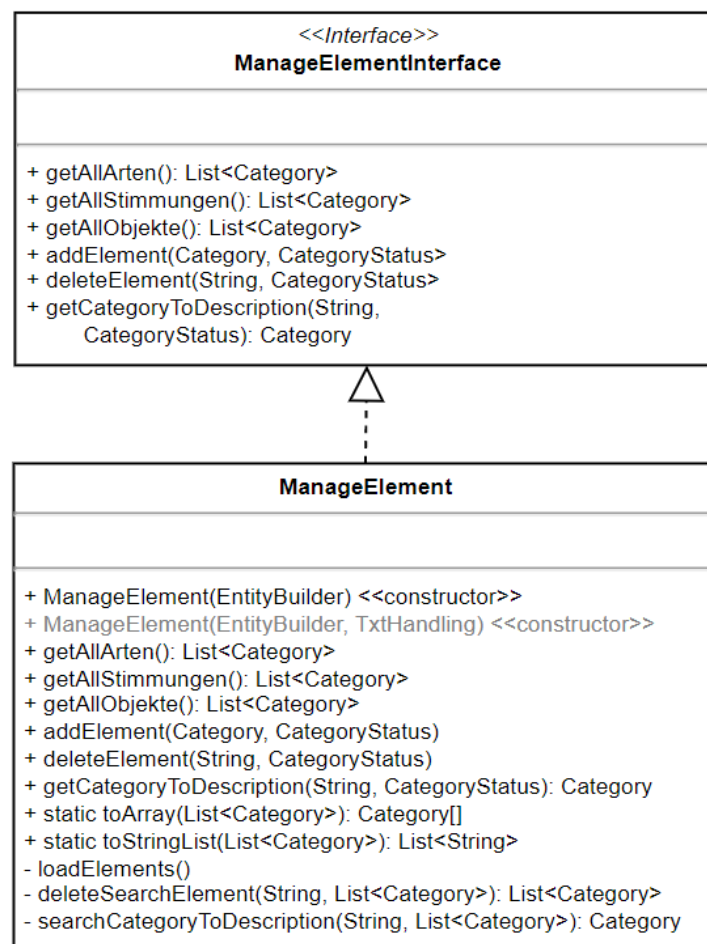


Abbildung 3.15: DIP -Controller/ ManageElement

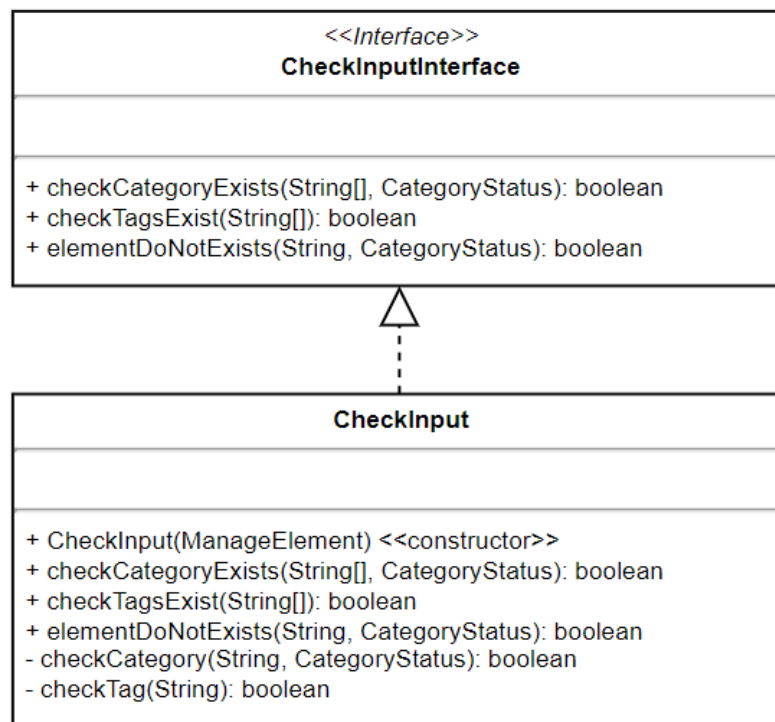


Abbildung 3.16: DIP -Controller/ CheckInput

4 Weitere Prinzipien

4.1 Analyse der GRASP: Geringe Kopplung

Der Begriff Kopplung bezeichnet die Verknüpfung von Anwendungen, Systemen oder Modulen, welche eine resultierende Abhängigkeit beschreibt.

4.1.1 Positiv-Beispiel 1

Die Methode `toTxtString(List<Category>): String` von `Jobs/ TxtHandling` wandelt die einzelnen Elemente in speicherbare Strings um. Dadurch können diese korrekt gespeichert werden, um diese nachfolgend richtig interpretieren zu können. Die benötigte Syntax zur Speicherung ist bei den Elementen selbst hinterlegt.

Folglich liegt eine schwache Kopplung zwischen `Jobs/ TxtHandling` und `Entity/ Category` vor. Durch diese Struktur kann die Speichersyntax geändert oder erweitert werden. Ebenfalls kann die Syntax variabel strukturiert sein für die `Entity/ SimpleCategory` und `Entity/ Objekt` (UML 4.1)

4.1.2 Positiv-Beispiel 2

Die Methode `toStringList` von `Controller/ ManageElement` wandelt eine Liste von `Category` in eine Liste von Strings um. Hierbei wird die Polymorphie genutzt, um den benötigten String des jeweiligen Elements individuell auszulesen. Dadurch ist eine schwache Kopplung der Klassen `Controller/ ManageElement` und `Entity/ Category` gegeben. (UML 4.2)

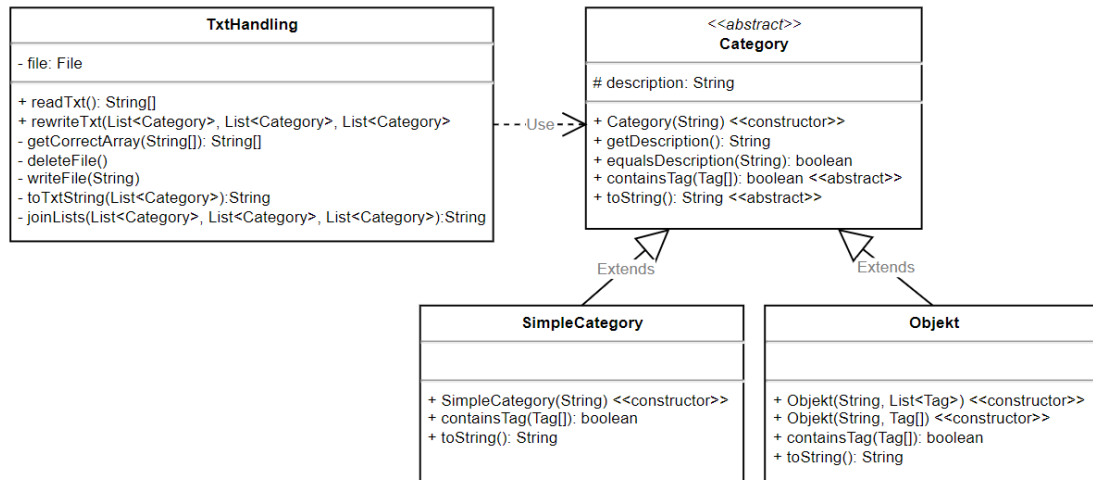


Abbildung 4.1: Geringe Kopplung -Jobs/ TxtHandling & Entity/ Category

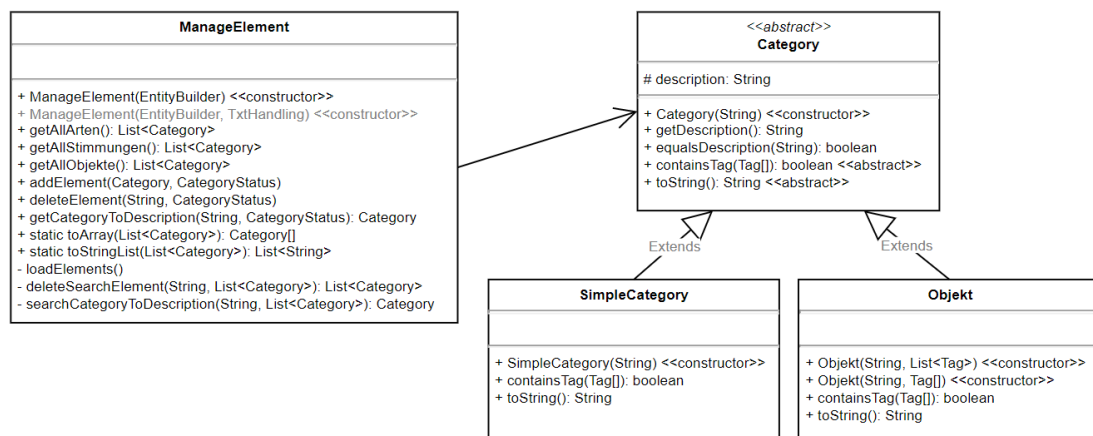


Abbildung 4.2: Geringe Kopplung -Controller/ ManageElement & Entity/ Category

4.1.3 Negativ-Beispiel 1

Das Enum **Entity/ Tag** besitzt statische Methoden. Dadurch erfolgen alle Aufrufe ebenfalls statisch. Folglich liegt zwischen den Klassen **Jobs/ EntityBuilder**, **Controller/ GUI** und **Controller/ CheckInput** zu **Entity/ Tag** eine hohe Kopplung vor.

Um den statischen Aufruf des Enums **Entity/ Tag** zu vermeiden müssten die aufrufenden Klassen mit einer Liste der **Tags** arbeiten und nicht mit dem Enum selbst. (UML 4.3)

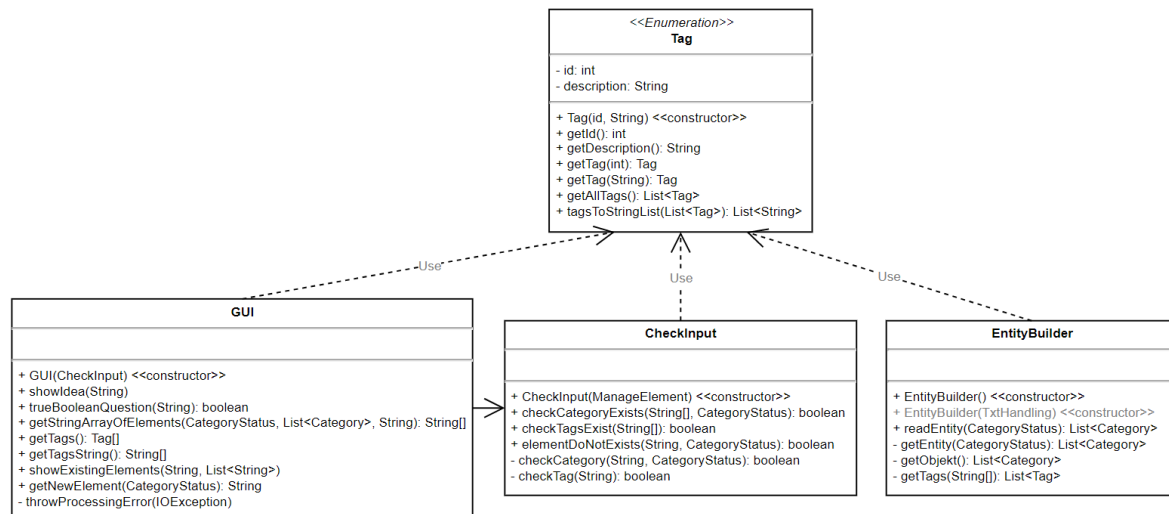


Abbildung 4.3: Geringe Kopplung -Entity/ Tag

4.1.4 Negativ-Beispiel 2

Die Methode `getEntity(CategoryStatus): List<Category>` der Klasse **Jobs/ EntityBuilder** weist eine starke Kopplung auf. Hier werden die gespeicherten **Category** Inhalte als Text eingelesen und anschließend als entsprechende Objekte erzeugt.

Die starke Kopplung kann durch eine Teilung der Methode in zwei verschiedene Methoden gelöst werden. (Listing 4.1.4)

```
1  List<Category> getEntity(CategoryStatus entityStatus) {
2      List<Category> entityList = new ArrayList<>();
3      String[] text = this.handlerTxt.readTxt();
4      String entityText;
5      if (text != null) {
6          if (entityStatus.equals(CategoryStatus.ART)) {
7              entityText = text[0];
8          } else {
9              entityText = text[1];
10         }
11         text = entityText.split(",");
12
13         for (String s : text) {
14             entityList.add(new SimpleCategory(s));
15         }
16     }
17     return entityList;
18 }
19
```

Listing 4.1: Starke Kopplung - `getEntity`

4.2 Analyse der GRASP: Hohe Kohäsion

Eine hohe Kohäsion beschreibt einen starken inhaltlichen Zusammenhang der Elemente innerhalb eines Bausteins.

4.2.1 Beispiel 1

Die Klasse `Controller/ Element/ UpdateElement` weist eine hohe Kohäsion auf. So ist die Klasse nur für das Updaten der Elemente zuständig. Da das Update laut CRUD-Prinzip aus den Funktionen des Neuanlegens (Create) und Löschen (Delete) besteht ist dieser inhaltliche Zusammenhang auch in `Controller/ Element/ UpdateElement` gegeben. Die Klasse ruft nur die jeweils zuständigen Klassen `Controller/ Element/ AddElement` und `Controller/ Element/ DeleteElement` auf, um die entsprechenden Funktionen auszuführen. Dadurch entsteht keine inhaltliche Überschneidung der Klassen und jede Klasse führt nur die eigenen notwendigen Funktionen aus. (UML 4.4)

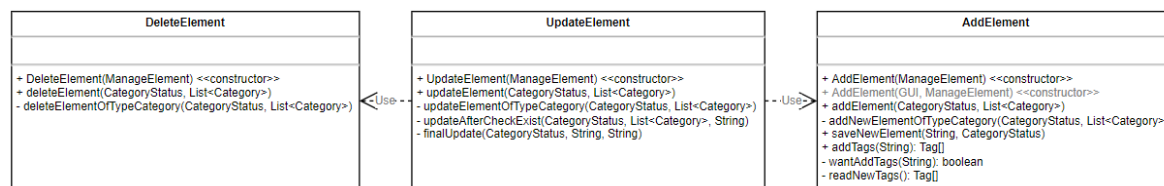


Abbildung 4.4: Hohe Kohäsion -Controller/ Element/

4.2.2 Beispiel 2

Das Enum `Entity/ CategoryStatus` weist eine hohe Kohäsion auf. So ist das Enum für das Managen der möglichen Status zuständig. Dazu gehört das hinterlegen der Statusbezeichnungen mit zugehörigem Plural und die Methode zum Auslesen der Informationen. Zusätzlich kann ein Enum vergleichen, ob der übergeben `CategoryStatus` identisch zum eigenen Status ist.

Folglich liegen alle Methoden und Funktionen zum Auslesen und Arbeiten mit dem `CategoryStatus` in der Klasse `Entity/ CategoryStatus`. Dadurch ist die Klasse inhaltlich zusammenhängend und auf das Wesentliche beschränkt. (UML 4.5)

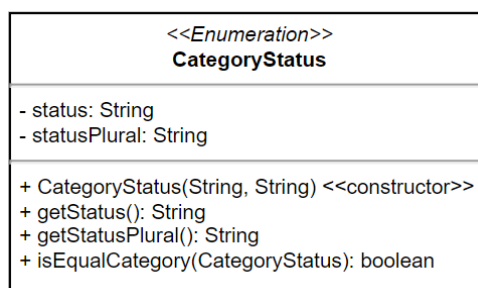


Abbildung 4.5: Hohe Kohäsion -Entity/ CategoryStatus

4.3 Don't Repeat Yourself

4.3.1 Beispiel 1

Im Commit `2e38bff96607757452fefb8ccb9a1a49587ebc82` ist das Management der Elemente bereits in die drei Klassen `Controller/ Element/ AddElement`, `DeleteElement` und `UpdateElement` aufgeteilt. Jedoch ist die Logik von `Update` noch nicht konform zu dem CRUD-Prinzip. So stellt `Controller/ Element/ UpdateElement` die gleichen logischen

Fragen nach den potentiellen **Tags** und Inhalten wie beim Neuanlegen. Im aktuellen Commit dagegen ruft die Klasse die Methoden von **Controller/ Element/ AddElement** auf. Dadurch kann Codedopplung vermieden werden und der Code wird verständlicher. (Listings 4.3.1 & 4.3.1)

```
1  public void updateElement(String kriterium , List<String> tagsOptions)
   throws IOException {
2      String newElement;
3      BufferedReader tastatur = new BufferedReader(new InputStreamReader(
   System.in));
4      System.out.println("Welches Element wollen Sie bearbeiten? ");
5      TxtHandling.deleteElement(tastatur.readLine(), Objects.requireNonNull(
   getEntityStatus(kriterium)));
6      System.out.println("Geben sie das bearbeitete Element ein: ");
7      newElement = tastatur.readLine();
8      TxtHandling.addElement(addNewTagsFrage(newElement), Objects.
   requireNonNull(getEntityStatus(kriterium)));
9  }
10
11 public String addNewTagsFrage(String objekt) throws IOException {
12     String question = "Wollen Sie Tags zum Objekt " + objekt + "
   hinzufuegen?";
13     if (gui.trueBooleanQuestion(question)) {
14         return AddElemente.addingTags(objekt, ElementeController.
   getAllStringTags());
15     } else return objekt;
16 }
17
18 private void showExistingTagsForObjekt(String element, List<String> tags)
   {
19     System.out.println("Diese tags fuer " + element + " existieren bereits:
   ");
20     for (String option : tags) {
21         System.out.print(option + " ");
22     }
23     System.out.println();
24 }
25
```

Listing 4.2: DRY - Controller/ Element/ UpdateElement - Alter Stand

```
1  private void finalUpdate(CategoryStatus categoryStatus, String
newElement, String element){
2      manageElement.deleteElement(element, categoryStatus);
3      new AddElement(manageElement).saveNewElement(newElement,
categoryStatus);
4  }
5
```

Listing 4.3: DRY - Controller/ Element/ UpdateElement - Aktueller Stand

4.3.2 Beispiel 2

Dem User werden zur Simulation der grafischen Oberfläche verschieden „Ja Nein“ Fragen gestellt. Im Commit 0fc0694db9b645b8ecdd73d278684a0b0001a4d7 wird jede Frage separat gestellt und behandelt. Dadurch entstehen viele Codedopplungen. Beim aktuellen Stand wird mit nur einer logischen Abfrage gearbeitet, welche unterschiedliche Fragetexte an den User ausgeben kann und die Antwort des Users zurück gibt. Des Weiteren ist im aktuellen Stand die Fehlerbehandlung bereits ergänzt. (Listings 4.3.2 & 4.3.2)

```

1  final String JA = "j";
2  final String NEIN = "n";
3  private boolean wantSearch(String kriterium) throws IOException {
4      BufferedReader tastatur = new BufferedReader(new InputStreamReader(
5          System.in));
6      System.out.print("Wollen Sie nach " + kriterium + " suchen? (" + this.
7          JA + "/" + this.NEIN + "): ");
8      return tastatur.readLine().equals(this.JA);
9  }
10 private boolean wantAdd() throws IOException {
11     System.out.println("Wollen Sie ein Element hinzufuegen? (" + this.JA +
12         "/" + this.NEIN + "): ");
13     BufferedReader tastatur = new BufferedReader(new InputStreamReader(
14         System.in));
15     return tastatur.readLine().equals(this.JA);
16 }
17 private void addArtElement() throws IOException {
18     System.out.println("Wollen Sie ein Element zum Typ Art hinzufuegen? ("
19         + this.JA + "/" + this.NEIN + "): ");
20     BufferedReader tastatur = new BufferedReader(new InputStreamReader(
21         System.in));
22     if (tastatur.readLine().equals(this.JA)) {
23         [ . . . ]
24     }
25 }
26 private void addStimmungElement() throws IOException {
27     System.out.println("Wollen Sie ein Element zum Typ Stimmung hinzufuegen?
28         (" + this.JA + "/" + this.NEIN + "): ");
29     BufferedReader tastatur = new BufferedReader(new InputStreamReader(
30         System.in));
31     if (tastatur.readLine().equals(this.JA)) {
32         [ . . . ]
33     }
34 }
35 private void addObjektElement() throws IOException {
36     System.out.println("Wollen Sie ein Element zum Typ Objekt hinzufuegen?
37         (" + this.JA + "/" + this.NEIN + "): ");
38     BufferedReader tastatur = new BufferedReader(new InputStreamReader(
39         System.in));
40     if (tastatur.readLine().equals(this.JA)) {
41         [ . . . ]
42     }
43     System.out.println("Wollen sie tags zu diesem Wort hinzufuegen? (" +
44         this.JA + "/" + this.NEIN + "): ");
45     if (tastatur.readLine().equals(this.JA)) {
46         [ . . . ]
47     }
48 }

```



```
1  public boolean trueBooleanQuestion(String question) {
2      try {
3          String JA = "j";
4          String NEIN = "n";
5          BufferedReader tastatur = new BufferedReader(new InputStreamReader(
        System.in));
6
7          System.out.print(question + " (" + JA + "/" + NEIN + "): ");
8          String eingabe = tastatur.readLine();
9          if (eingabe.equals(JA)) {
10             return true;
11          } else if (eingabe.equals(NEIN)) {
12             return false;
13          } else {
14             throw new FalseInputException();
15          }
16      } catch (FalseInputException ex) {
17          System.out.println(ex.getMessage());
18          return trueBooleanQuestion(question);
19      } catch (IOException e) {
20          throwProcessingError(e);
21          return trueBooleanQuestion(question);
22      }
23  }
```

Listing 4.5: DRY - Controller/ GUI - Aktueller Stand

5 Unit Tests

5.1 20 Unit Tests

Unit Test	Beschreibung
<code>Controller.Element.AddElementTest</code>	
<code>addTags_True()</code>	wenn <code>wantAddTags(String)</code> einen True-Wert zurück gibt, wird das zurückgegebene gefüllte Tag-Array überprüft
<code>addTags_False()</code>	wenn <code>wantAddTags(String)</code> einen False-Wert zurück gibt, wird die Länge des zurückgegeben Tag-Array überprüft, ob dieses eine Länge von 0 hat
<code>wanntAddTags_True()</code>	prüft, ob die Abfrage in <code>gui.trueBooleanQuestion(String)</code> einen True-Wert zurück gibt
<code>wanntAddTags_False()</code>	prüft, ob die Abfrage in <code>gui.trueBooleanQuestion(String)</code> einen False-Wert zurück gibt

Unit Test	Beschreibung
<code>Controller.SearchElement.FilterIdeaTest</code>	
<code>randomOneAllElements NullCategory()</code>	prüft, ob <code>FilterIdea.random()</code> einen Wert zurückgibt, wenn nichts vorgegeben wird
<code>randomTwoAllElements OneCategory()</code>	prüft, ob <code>FilterIdea.random()</code> einen Wert zurückgibt, wenn eine Vorgabe zur Auswahl existiert
<code>randomTwoAllElements TwoCategory()</code>	prüft, ob <code>FilterIdea.random()</code> einen Wert zurückgibt, wenn zwei Vorgaben zur Auswahl existieren

Unit Test	Beschreibung
Controller.SearchElement.FilterObjektIdeaTest	
randomOneAllElements NullCategory()	prüft, ob <code>FilterObjektIdea.random()</code> einen Wert zurückgibt, wenn nichts vorgegeben wird
randomTwoAllElements OneCategory()	prüft, ob <code>FilterObjektIdea.random()</code> einen Wert zurückgibt, wenn eine Vorgabe zur Auswahl existiert
randomTwoAllElements TwoCategory()	prüft, ob <code>FilterObjektIdea.random()</code> einen Wert zurückgibt, wenn zwei Vorgaben zur Auswahl existieren
filterCategoryElements Success()	prüft, ob <code>FilterObjektIdea.filterCategoryElements()</code> ein Objekt zurückgibt, das den Tag besitzt, nachdem gefiltert wurde
filterCategoryElements Failure()	prüft, ob <code>FilterObjektIdea.filterCategoryElements()</code> kein Objekt zurückgibt, wenn kein Objekt den Tag besitzt, nachdem gefiltert wurde
filterObjekteSuccess()	prüft, ob <code>FilterObjektIdea.filterObjekte()</code> ein Objekt zurückgibt, das den Tag besitzt, nachdem gefiltert wurde
filterObjekteFailure()	prüft, ob <code>FilterObjektIdea.filterObjekte()</code> kein Objekt zurückgibt, wenn kein Objekt den Tag besitzt, nachdem gefiltert wurde
tagFilterExistsSuccess()	prüft, ob <code>FilterObjektIdea.tagFilterExists(Tag[])</code> true zurückgibt, wenn Tags existieren nach denen gefiltert werden soll
tagFilterExistsFailure()	prüft, ob <code>FilterObjektIdea.tagFilterExists(Tag[])</code> false zurückgibt, wenn keine Tags existieren nach denen gefiltert werden soll

Unit Test	Beschreibung
Controller.SearchElement.FilterTest	
fillCategoryElementsIfEmpty()	prüft, ob <code>FilterIdea.fillCategoryElementsIfEmpty()</code> eine <code>FilterIdea</code> befüllt, wenn keine Filter existieren
noFilterElementsSuccess()	prüft, ob <code>FilterIdea.noFilterElements(Category[])</code> true zurückgibt, wenn keine Filter existieren
noFilterElementsFailure()	prüft, ob <code>FilterIdea.noFilterElements(Category[])</code> false zurückgibt, wenn Filter existieren

Unit Test	Beschreibung
<code>Controller.SearchElement.SearchElementsTest</code>	
<code>getSearchElementsTrue()</code>	prüft, ob <code>SearchElements.getSearchElements(CategoryStatus, List<Category>)</code> ein Category-Array zurückgibt
<code>getSearchElementsFalse()</code>	prüft, ob <code>SearchElements.getSearchElements(CategoryStatus, List<Category>)</code> ein Category-Array der Länge 0 zurückgibt
<code>getSearchTagsTrue()</code>	prüft, ob <code>SearchElements.getSearchTags()</code> ein Tag-Array zurückgibt
<code>getSearchTagsFalse()</code>	prüft, ob <code>SearchElements.getSearchTags()</code> ein Tag-Array der Länge 0 zurückgibt
<code>getFilters()</code>	prüft, ob <code>SearchElements.getFilters(CategoryStatus, List<Category>)</code> ein Category-Array zurückgibt

Unit Test	Beschreibung
<code>Controller.CheckInputTest</code>	
<code>checkCategoriesExistSuccess()</code>	prüft, ob <code>CheckInput.checkCategoriesExist(String[], CategoryStatus)</code> true zurückgibt, wenn alle Elemente eines Arrays einer Kategorie bereits existiert
<code>checkCategoriesExistFailure()</code>	prüft, ob <code>CheckInput.checkCategoriesExist(String[], CategoryStatus)</code> false zurückgibt, wenn ein Element des Arrays einer Kategorie noch nicht existiert
<code>checkCategorySuccess()</code>	prüft, ob <code>CheckInput.checkCategory(String, CategoryStatus)</code> true zurückgibt, wenn ein Element einer Kategorie bereits existiert
<code>checkCategoryFailure()</code>	prüft, ob <code>CheckInput.checkCategory(String, CategoryStatus)</code> false zurückgibt, wenn ein Element einer Kategorie noch nicht existiert
<code>checkTagsExistSuccess()</code>	prüft, ob <code>CheckInput.checkTagsExist(String[])</code> true zurückgibt, wenn alle Tags eines Arrays bereits existiert
<code>checkTagsExistFailure()</code>	prüft, ob <code>CheckInput.checkTagsExist(String[])</code> true zurückgibt, wenn alle Tags eines Arrays bereits existiert
<code>checkTagSuccess()</code>	prüft, ob <code>CheckInput.checkTag(String)</code> true zurückgibt, wenn der Tag bereits existiert
<code>checkTagFailure()</code>	prüft, ob <code>CheckInput.checkTag(String)</code> true zurückgibt, wenn der Tag noch nicht existiert
<code>elementDoNotExistsSuccess()</code>	prüft, ob <code>CheckInput.elementDoNotExists(String, CategoryStatus)</code> true zurückgibt, wenn das Element einer Kategorie noch nicht existiert
<code>elementDoNotExistsFailure()</code>	prüft, ob <code>CheckInput.elementDoNotExists(String, CategoryStatus)</code> false zurückgibt, wenn das Element einer Kategorie bereits existiert

Unit Test	Beschreibung
Controller.ManageElementTest	
<code>addElementEmptyList()</code>	prüft, ob <code>ManageElement.addElement(Category, CategoryStatus)</code> ein Element in eine Leere Liste hinzufügen kann
<code>deleteSearchElementEmptyList()</code>	prüft, ob <code>ManageElement.deleteSearchElement(String, List<Category>)</code> beim Entfernen eines Elements aus einer leeren Liste eine Liste der Länge 0 zurückgibt
<code>deleteSearchElementEmptyString()</code>	prüft, ob <code>ManageElement.deleteSearchElement(String, List<Category>)</code> beim Entfernen eines Elements mit dem Wert Null keine Veränderung der Liste vorgenommen wird
<code>deleteSearchElementSuccess()</code>	prüft, ob <code>ManageElement.deleteSearchElement(String, List<Category>)</code> nach Entfernen eines Elements aus einer Liste, die Liste ohne das zuentfernende Element zurückgibt
<code>deleteSearchElementFailure()</code>	prüft, ob <code>ManageElement.deleteSearchElement(String, List<Category>)</code> eine unveränderte Liste zurückgibt, wenn ein Element entfernt werden soll, das in dieser Liste jedoch nicht existiert
<code>searchCategoryToDescriptionEmptyList()</code>	prüft, ob <code>ManageElement.searchCategoryToDescription(String, List<Category>)</code> null zurückgibt, wenn mit der Beschreibung eines Elements ein Element einer Kategorie in einer leeren Liste gesucht wird
<code>searchCategoryToDescriptionEmptyString()</code>	prüft, ob <code>ManageElement.searchCategoryToDescription(String, List<Category>)</code> null zurückgibt, wenn ein Element einer Kategorie in einer Liste mit einer leeren Beschreibung gesucht wird
<code>searchCategoryToDescriptionSuccess()</code>	prüft, ob <code>ManageElement.searchCategoryToDescription(String, List<Category>)</code> ein Element einer Kategorie mithilfe einer Beschreibung finde
<code>searchCategoryToDescriptionFailure()</code>	prüft, ob <code>ManageElement.searchCategoryToDescription(String, List<Category>)</code> null zurückgibt, wenn eine Beschreibung zur Suche genutzt wird und die Liste jedoch nicht dieses Element beinhaltet
<code>toArrayEmpty()</code>	prüft, ob <code>ManageElement.toArray(List<Category>)</code> ein Array der Länge 0 zurückgibt, wenn eine leere Liste zu einem Array umgewandelt werden soll
<code>toArrayFill()</code>	prüft, ob <code>ManageElement.toArray(List<Category>)</code> eine Liste in ein Array umwandeln kann
<code>toStringListEmpty()</code>	prüft, ob <code>ManageElement.toStringList(List<Category>)</code> eine String-Liste der Größe 0 zurückgibt, wenn eine leere Category-Liste in eine String-Liste umgewandelt werden soll
<code>toStringListFill()</code>	prüft, ob <code>ManageElement.toStringList(List<Category>)</code> eine String-Liste zurückgibt, wenn eine Category-Liste in eine String-Liste umgewandelt werden soll

Unit Test	Beschreibung
Entity.ObjektTest	
containsTag_True()	prüft, ob <code>Objekt.containsTag(Objekt)</code> true zurückgibt, wenn das Objekt den gesuchten Tag beinhaltet
containsTag_False()	prüft, ob <code>Objekt.containsTag(Objekt)</code> false zurückgibt, wenn das Objekt nicht den gesuchten Tag beinhaltet
toStringTest()	prüft, ob <code>Objekt.toString()</code> ein Objekt in einen String umwandeln kann

Unit Test	Beschreibung
Entity.TagTest	
getTag_Id_True()	prüft, ob <code>Tag.getTag(int)</code> ein Tag zurückgibt, wenn dieser mit Hilfe der Id bestimmt werden konnte
getTag_Id_False()	prüft, ob <code>Tag.getTag(int)</code> null zurückgibt, da kein Tag mit Hilfe der Id bestimmt werden konnte
getTag_Description_True()	prüft, ob <code>Tag.getTag(String)</code> ein Tag zurückgibt, wenn dieser mit Hilfe der Beschreibung bestimmt werden konnte
getTag_Description_False()	prüft, ob <code>Tag.getTag(String)</code> null zurückgibt, da kein Tag mit Hilfe der Beschreibung bestimmt werden konnte
tagsToStringList()	prüft, ob <code>Tag.tagsToStringList(List<Tag>)</code> eine String-Liste zurückgibt, wenn eine Tag-Liste in eine String-Liste umgewandelt werden soll

Unit Test	Beschreibung
jobs.EntityBuilderTest	
readEntity_Objekt()	prüft, ob <code>EntityBuilder.readEntity(CategoryStatus)</code> eine Category-Liste mit Elementen des Typs Objekt zurückgibt
readEntity_SimpleCategory()	prüft, ob <code>EntityBuilder.readEntity(CategoryStatus)</code> eine Category-Liste mit Elementen des Typs SimpleCategory zurückgibt
getEntity()	prüft, ob <code>EntityBuilder.getEntity(CategoryStatus)</code> eine Category-Liste mit Elementen des Typs SimpleCategory zurückgibt
getObjekt()	prüft, ob <code>EntityBuilder.getObjekt()</code> eine Category-Liste mit Elementen des Typs Objekt zurückgibt
getTags()	prüft, ob <code>EntityBuilder.getTags()</code> eine Tag-Liste zurückgibt

Unit Test	Beschreibung
<code>Jobs.TxtHandlingTest</code>	
<code>getCorrectArrayCorrect()</code>	prüft, ob <code>TxtHandling.getCorrectArray(String[])</code> eine Array mit drei eingelesenen Elementen zurückgibt
<code>getCorrectArrayFalse()</code>	prüft, ob <code>TxtHandling.getCorrectArray(String[])</code> ein gefülltes Array der Länge 3 zurückgibt, wenn nur zwei Elemente eingelesen wurden
<code>toTxtString()</code>	prüft, ob <code>TxtHandling.toTxtString(List<Category>)</code> eine Kategorie-Liste in einen String umwandeln kann
<code>joinList()</code>	prüft, ob <code>TxtHandling.joinLists(List<Category>, List<Category>, List<Category>)</code> die 3 Category-Listen zu einem String mit vorgegebener Syntax zusammen joinen kann

5.2 ATRIP: Automatic

Um die Tests automatisch mit Maven zu testen, war ein Plugin vonnöten, welches die Test-Klassen erkennt und diese ausführt.

Mit dem Befehl `mvn test` können explizit alle Tests durchlaufen werden. Dabei sieht man, welche Tests erfolgreich waren und welche nicht. Möchte man jedoch die Tests manuell starten, ist nur ein Knopfdruck auf dem Run-Button der jeweiligen Test Klasse nötig. Es wird keine manuelle Eingabe von Daten benötigt um die Tests laufen zu lassen. Jeder Test liefert nur die Ergebnisse bestanden oder fehlgeschlagen, dies wird mit `assertX()`-Methoden in den einzelnen Tests realisiert.

5.3 ATRIP: Thorough

Tests gelten als Vollständig, wenn ein Test alles Notwendige überprüft. Ein weiterer Punkt der Vollständigkeit ist die iterative Vorgehensweise zur Erstellung der Tests.

Jeder Test des Projekts deckt jeweils ein UseCase ab. Dabei wurde darauf geachtet, dass alle kritischen Stellen des Systems getestet wurden, bei denen vom User verursachten Fehler entstehen könnten. Diese kritischen Stellen befinden sich hauptsächlich in den Gebieten, in welchen Daten verarbeitet werden müssen.

Positiv Beispiele sind demnach unsere `getCorrectArrayCorrect()` Methode sowie die `getCorrectArrayFalse()` aus der Klasse `test/ Jobs/ TxtHandlingTest`. Diese Methoden sind erst mit dem auftreten eines Fehlers geschrieben worden. (Listing 5.3)

```
1  @Test
2  void getCorrectArrayCorrect() {
3      // Given
4      String [] text = {"Art1,Art2", "Stimmung1,Stimmung2", "Objekt1,Objekt2"};
5
6      // When
7      String [] correctTextArray = handlerTxt.getCorrectArray(text);
8
9      // Then
10     assertEquals(text.length, correctTextArray.length);
11     assertEquals(text[0], correctTextArray[0]);
12     assertEquals(text[1], correctTextArray[1]);
13     assertEquals(text[2], correctTextArray[2]);
14 }
15
16 @Test
17 void getCorrectArrayFalse() {
18     // Given
19     String [] text = {"Art1,Art2", "Stimmung1,Stimmung2"};
20
21     // When
22     String [] correctTextArray = handlerTxt.getCorrectArray(text);
23
24     // Then
25     assertEquals(3, correctTextArray.length);
26     assertEquals(text[0], correctTextArray[0]);
27     assertEquals(text[1], correctTextArray[1]);
28     assertEquals("", correctTextArray[2]);
29 }
30
```

Listing 5.1: Thourough - test/ Jobs/ TxtHandlingTest

Negativ Beispiele liegen nicht direkt vor, da keine Tests existieren die unwichtige Sachen testen. Folglich wurde für die **Controller/ Steuerung** und die **Controller/ GUI** Klassen keine Test geschrieben, da diese keine eigene Logik besitzen, sondern nur Methoden aufrufen welche selbst schon getestet worden sind. Des Weiteren wurden `toString()`-Methoden nur dann getestet wenn die Original Methode überschrieben worden ist.

5.4 ATRIP: Professional

Test-Handling wird als Professionell eingestuft, wenn diese Tests den gleichen Qualitätsstandards wie der „Produktivcode“ haben, wenn keine unnötigen Tests geschrieben worden und die Tests Teil der Dokumentation sind.

Positiv Beispiele sind demnach, dass für die Klasse **Controller/ Steuerung** keine Tests geschrieben worden sind. Die **Controller/ Steuerung** Klasse besitzt keine eigene Logik. Sie ruft nur bereits getestete Methoden aus anderen Klassen auf. Daher wäre es unnötig Tests für diese Methoden zu schreiben, in denen nur bereits getestete Methoden aufgerufen werden. (Listing 5.4)

```

1  public static void main(String [] args) {
2      [ . . . ]
3  }
4  private static void userInteraction() {
5      if (gui.trueBooleanQuestion("Wollen Sie nach einer kreativen Idee
6      suchen?")) {
7          gui.showIdea((new Idea(manageElement)).toString());
8      }
9      if (gui.trueBooleanQuestion("Wollen Sie neue Elemente hinzufuegen?")) {
10         addElementToElementList();
11     }
12     if (gui.trueBooleanQuestion("Wollen Sie Elemente bearbeiten?")) {
13         updateElementInElementList();
14     }
15     if (gui.trueBooleanQuestion("Wollen Sie Elemente loeschen?")) {
16         deleteElementFromElementList();
17     }
18 }
19 private static void addElementToElementList() {
20     addElemente.addElement(CategoryStatus.ART, manageElement.getAllArten());
21     ;
22     addElemente.addElement(CategoryStatus.STIMMUNG, manageElement.
23     getAllStimmungen());
24     addElemente.addElement(CategoryStatus.OBJEKT, manageElement.
25     getAllObjekte());
26 }
27 private static void deleteElementFromElementList() {
28     deleteElemente.deleteElement(CategoryStatus.ART, manageElement.
29     getAllArten());
30     deleteElemente.deleteElement(CategoryStatus.STIMMUNG, manageElement.
31     getAllStimmungen());
32     deleteElemente.deleteElement(CategoryStatus.OBJEKT, manageElement.
33     getAllObjekte());
34 }
35 private static void updateElementInElementList() {
36     updateElemente.updateElement(CategoryStatus.ART, manageElement.
37     getAllArten());
38     updateElemente.updateElement(CategoryStatus.STIMMUNG, manageElement.
39     getAllStimmungen());
40     updateElemente.updateElement(CategoryStatus.OBJEKT, manageElement.
41     getAllObjekte());
42 }

```

Listing 5.2: Professional - Controller/ Steuerung

Des Weiteren sind alle Test-Funktionen in eigene Test-Klassen und Ordnern unterteilt, um eine bessere und lesbarere Struktur zu ermöglichen.

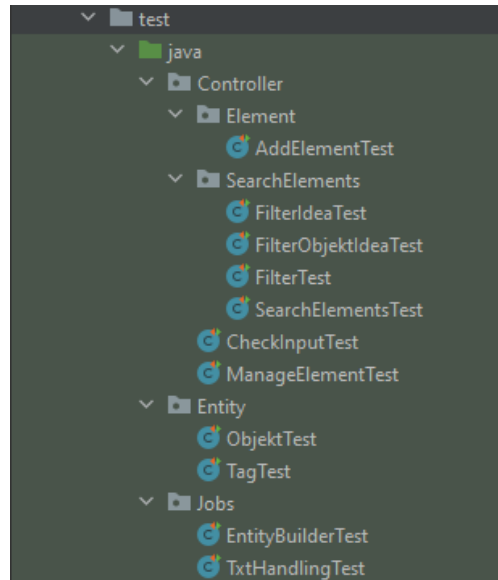


Abbildung 5.1: Professional - Testverzeichnis

Negativ Beispiele für Professionalität sind in diesem Fall bezogen auf die Aussage, dass keine unnötigen Tests geschrieben werden sollen. In den folgenden zwei Test-Methoden wird die Methode `EntityBuilder.readEntity(CategoryStatus)` getestet, welche nur Methoden aufruft die selbst getestet worden sind. In dem Test `readEntity_Objekt()` wird der Fall abgedeckt, wenn `EntityBuilder.readEntity(CategoryStatus)` mit `CategoryStatus.OBJEKT` aufgerufen wird. Der Test `readEntity_SimpleCategory()` deckt den Fall ab, wenn `EntityBuilder.readEntity(CategoryStatus)` mit `CategoryStatus.ART` oder `CategoryStatus.STIMMUNG` aufgerufen wird. Abhängig von dem Fall der Eintritt wird entweder die Methode `EntityBuilder.getObjekt()` oder `EntityBuilder.getEntity(CategoryStatus)` aufgerufen, welche eigene Test-Methoden besitzen. (Listings 5.4 & 5.4)

```
1  public List<Category> readEntity(CategoryStatus categoryStatus) {  
2      if (CategoryStatus.OBJEKT.isEqualCategory(categoryStatus)) {  
3          return getObject();  
4      } else {  
5          return getEntity(categoryStatus);  
6      }  
7  }  
8
```

Listing 5.3: Professional - Jobs/ EntityBuilder

```

1  @Test
2  void readEntity_Objekt() {
3      CategoryStatus categoryStatus = CategoryStatus.OBJEKT;
4      String [] testText = {"art1,art2,art3", "stimmung1,stimmung2,stimmung3", "
      objekt1;1;2,objekt2;3;4,objekt3;3;1"};
5      List<Category> testList = new ArrayList<>();
6      testList.add(new Objekt("objekt1", new Tag[] { Tag.LANDSCHAFT, Tag.
      GEGENSTAND}));
7      testList.add(new Objekt("objekt2", new Tag[] { Tag.FANTASIE, Tag.TIER}));
8      testList.add(new Objekt("objekt3", new Tag[] { Tag.FANTASIE, Tag.LANDSCHAFT
      }));
9      when(handlerTxtMock.readTxt()).thenReturn(testText);
10     List<Category> returnCategoryElements = entityBuilder.readEntity(
        categoryStatus);
11     assertEquals(testList.size(), returnCategoryElements.size());
12     assertEquals(testList.get(0).toString(), returnCategoryElements.get(0).
        toString());
13     assertEquals(testList.get(1).toString(), returnCategoryElements.get(1).
        toString());
14     assertEquals(testList.get(2).toString(), returnCategoryElements.get(2).
        toString());
15 }
16 @Test
17 void readEntity_SimpleCategory() {
18     CategoryStatus categoryStatus = CategoryStatus.ART;
19     String [] testText = {"art1,art2,art3", "stimmung1,stimmung2,stimmung3", "
        objekt1;1;2,objekt2;3;4,objekt3;3;1"};
20     List<Category> testList = new ArrayList<>();
21     testList.add(new SimpleCategory("art1"));
22     testList.add(new SimpleCategory("art2"));
23     testList.add(new SimpleCategory("art3"));
24     when(handlerTxtMock.readTxt()).thenReturn(testText);
25     List<Category> returnCategoryElements = entityBuilder.readEntity(
        categoryStatus);
26     assertEquals(testList.size(), returnCategoryElements.size());
27     assertEquals(testList.get(0).toString(), returnCategoryElements.get(0).
        toString());
28     assertEquals(testList.get(1).toString(), returnCategoryElements.get(1).
        toString());
29     assertEquals(testList.get(2).toString(), returnCategoryElements.get(2).
        toString());
30 }
31

```

Listing 5.4: Professional - test/ Jobs/ EntityBuilder

5.5 Code Coverage

Die Code Coverage wird von dem Jacoco-PlugIn überprüft und erstellt dafür eine `index.html` Datei, nachdem Maven mit dem `mvn -verify` Befehl die Tests durchlaufen hat. In dieser Datei wird die Code Coverage der einzelnen Packages und Klassen gezeigt (siehe Abbildung 5.2).

Bei der **Controller/GUI** Klasse wurden keine Tests geschrieben, da für diese ein Buffere-

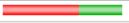









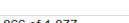
Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
Controller		36 %		43 %	38	63	110	171	20	34	2	4
Controller.Element		15 %		11 %	22	28	52	65	14	19	2	4
Controller.SearchElements		61 %		83 %	9	34	27	77	6	22	1	5
jobs		74 %		83 %	8	23	25	72	5	14	0	2
Entity		95 %		95 %	4	33	4	66	3	23	0	5
Error		100 %		n/a	0	1	0	2	0	1	0	1
Total	866 of 1.877	53 %	57 of 138	58 %	81	182	218	453	48	113	5	21

Abbildung 5.2: Code Coverage vom kreativen Zufallsgenerator

dReader vonnöten ist und dieser nur mit höherem Aufwand simuliert werden kann.

Alle weiteren Klassen die nicht getestet sind, rufen nur bereits getestete Methoden aus anderen Klassen auf oder beinhalten **Getter()**- und **Setter()**-Methoden die als erfolgreich angenommen werden.

5.6 Fakes und Mocks

Mock-Objekte dienen als Stellvertreter für die eigentlich benötigten Objekte und simulieren das Verhalten einer Klasse. Des weiteren reduzieren sie die Abhängigkeiten zu weiteren Komponenten und Klassen. Aus diesen Gründen wurden Mock-Objekte genutzt. Dafür wurde die Dependency von Mockito in das Projekt eingebunden.

5.6.1 Beispiel 1

(UML 5.3)

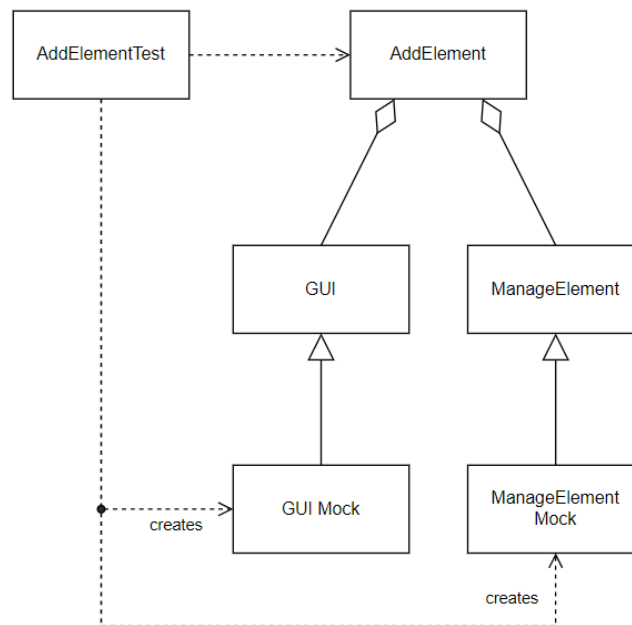


Abbildung 5.3: Fakes & Mocks -test/ Controller/ Element/ AddElementTest

5.6.2 Beispiel 2

(UML 5.4)

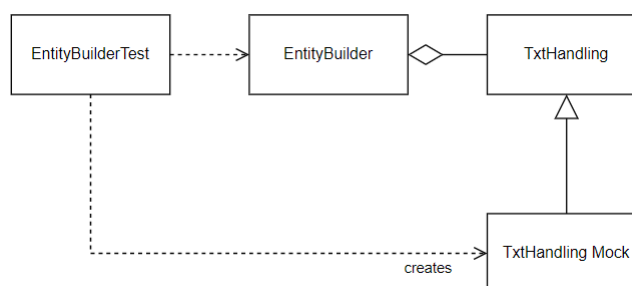


Abbildung 5.4: Fakes & Mocks -test/ Jobs/ EntityBuilderTest

5.6.3 Beispiel 3

(UML 5.5)

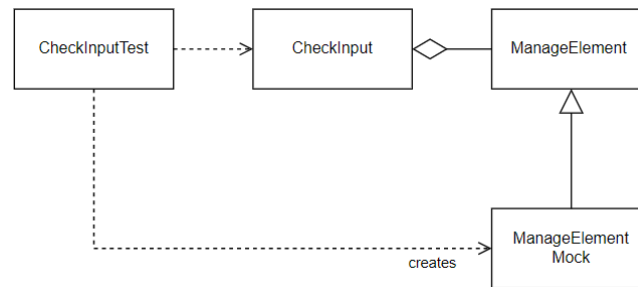


Abbildung 5.5: Fakes & Mocks -test/ Controller/ CheckInputTest

5.6.4 Beispiel 4

(UML 5.6)

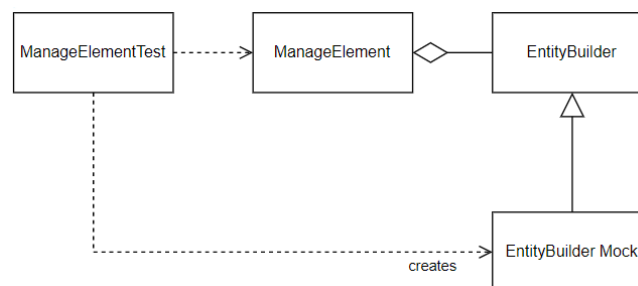


Abbildung 5.6: Fakes & Mocks -test/ Controller/ ManageElementTest

6 Domain Driven Design

6.1 Ubiquitous Language

Bezeichnung	Bedeutung	Begründung
Category	Ist abstrakt und bezeichnet Elemente der Typen Art, Stimmung und Objekt.	Category dient als Oberbegriff für SimpleCategory und Objekt.
SimpleCategory	Elemente vom Typ Art oder Stimmung, welche eine Bezeichnung beinhalten.	Der Aufbau der Elemente der Typen Art und Stimmung ist identisch und kann damit zu SimpleCategory zusammengefasst und ausgedrückt werden. Sie unterscheiden sich letztendlich nur bei dem CategoryStatus.
Filter	Filter sind Tags nach denen die Objekte differenziert werden können.	Dient der Aussortierung der Objekte nach dem eingegebenen Vorgaben des User.
Tag	Ist ein Attribut das im Zusammenhang mit einem Objekt gebracht wird.	Ein Tag dient der Identifizierung eines Objekts wenn in der Filterung nach einem Tag gefiltert wird.
Objekt	Kategorie des CategoryStatus Element, welche eine Bezeichnung und zugehörige Tags besitzen.	Objekt ist eine der Kategorien die für eine Idee genutzt wird.
GUI	Schnittstelle zum Benutzer als Konsole.	Eine GUI ist bekannt als eine Schnittstelle zum Benutzer mit graphischer Oberfläche. In diesem Projekt wird die grafische Oberfläche durch die Konsole ersetzt.
CategoryStatus	Definiert die Zugehörigkeit einer Kategorie zu den Typen Art, Stimmung oder Objekt.	Für die Verarbeitung der kreativen Ideen muss nach den Typen Art, Stimmung und Objekt differenziert werden. Dies wird von CategoryStatus ermöglicht.
Element	Beschreibt ein Wort einer Kategorie.	Elemente sind als Bestandteile größerer Verbindungen bekannt (Bauelemente, Elemente des Periodensystems). Deswegen eignen sie sich in diesem Kontext, um die einzelnen Wörter der Kategorien zu beschreiben.

Tabelle 6.1: Ubiquitous Language

6.2 Entities

Die Klasse `Entity/ Tag` repräsentiert in der Domäne eine Entität. Sie werden eindeutig über ihre ID bestimmt, da es nicht möglich ist, dass zwei Tags dieselbe Bezeichnung und ID haben können. Diese Eindeutigkeit wird bei der Filterung der Objekte benötigt und kann daher ohne höheren Aufwand mit Hilfe der Entität gewährleistet werden. (UML 6.1)

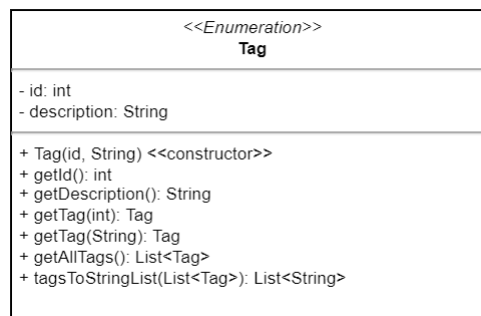


Abbildung 6.1: Entities - `Entity/ Tag`

6.3 Value Objects

In dieser Anwendung sind die Klassen `Entity/ Category`, `Entity/ CategoryStatus`, `Entity/ SimpleCategory` und `Entity/ Objekt` Value-Objects. Diese sind einfache Objekte ohne eigene Identität und Kapseln meistens nur primitive Werte wie die Bezeichnung. Um diese Objekte auf Gleichheit zu prüfen, werden die Werte verglichen und nur wenn diese Werte dieselben sind, gelten die Objekte als gleich. Die Klassen sind Value Objects und keine Entitäten, weil nicht das Objekt selbst wichtig ist, sondern der gekapselte Wert dahinter. Dadurch ergeben sich die Vorteile wie Unveränderbarkeit und dass diese Objekte selbst validierend und leicht testbar sind. (UML 6.2)

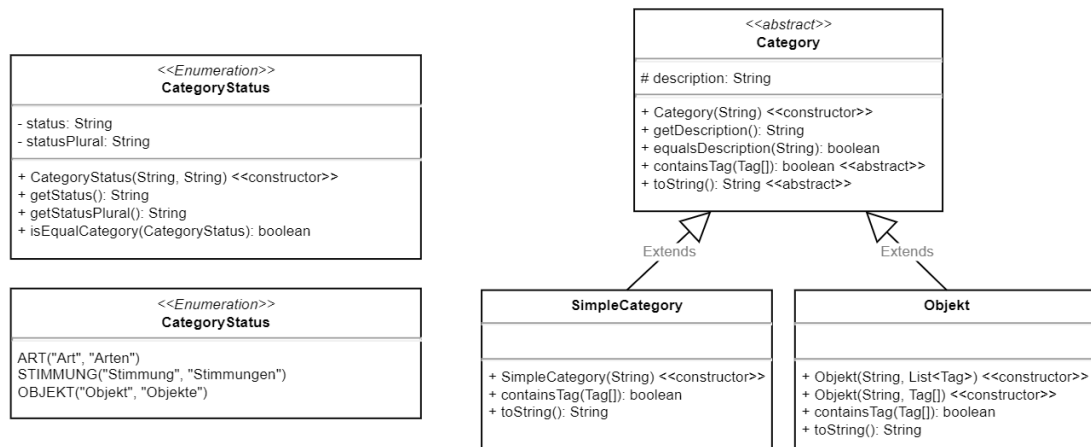


Abbildung 6.2: Value Objects - Entity/ Category

6.4 Repositories

Ein Repository ist ein Verzeichnis zur Speicherung und Beschreibung von Objekten. In dieser Anwendung wurden keine Repositories eingefügt und oder umgesetzt. In der Anwendung wird nur einmal im Ordner **Controller/Element/** das CRUD-Prinzip benötigt und umgesetzt. Dort werden die Methoden des CRUD-Prinzips direkt aufgerufen. So ist die Funktionsweise von **Controller/ Element/ UpdateElement** erste das Löschen des alten Elements und dann das Hinzufügen des neuen Elements.

Da das CRUD-Prinzip nur an einer einzigen Stelle benötigt wird, ist das Einfügen eines Repositories nicht zwingend erforderlich.

6.5 Aggregates

Objekte der Klasse `Entity/ Objekt` und `Entity/ Tags` bilden gemeinsam Aggregate. Dabei können ein, kein oder mehrere `Tags` einem Objekt der Klasse `Entity/ Objekt` zugewiesen werden. Damit wird die Komplexität der Beziehungen zwischen den Objekten reduziert, da ein Aggregat immer als eine Einheit betrachtet und verwaltet werden. (UML 6.3)

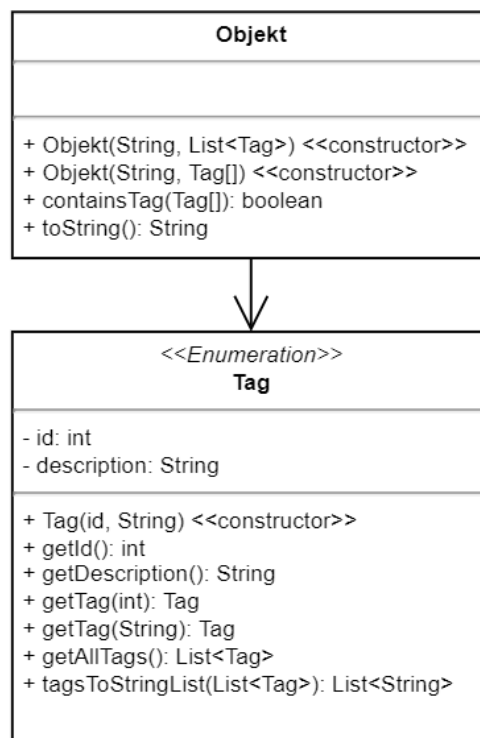


Abbildung 6.3: Aggregates - `Entity/ Objekt`

7 Refactoring

7.1 Code Smells

7.1.1 Beispiel 1: Duplicated Code

Die Methode `showExistingElements()` wurden in den Klassen `Controller/ Element/ AddElement`, `Controller/ Element/ DeleteElemente` und `Controller/ Element/ UpdateElement` identisch implementiert und mit dem Commit `2e38bff96607757452fefb8ccb9a1a49587ebc82` hinzugefügt. (Listing 7.1.1)

```
1  private void showExistingElements(String kriterium , List<String>
   allElements) {
2      System.out.println("Diese Elemente des Typs " + kriterium + "
   existieren bereits:");
3      for (String option : allElements) {
4          System.out.print(option + " ");
5      }
6      System.out.println();
7  }
```

Listing 7.1: Code Smells - Controller/ Element/ Add-, Delete-, UpdateElement

Die Methode `showExistingElements()` wird in die `Controller/ GUI` Klasse ausgelagert. Diese kann über die `Controller/ Element/ HandlingElement` Klasse aufgerufen werden. Die Klassen `Controller/ Element/ AddElement`, `Controller/ Element/ DeleteElemente` und `Controller/ Element/ UpdateElement` rufen an entsprechender Stelle `Controller/ Element/ HandlingElement` und somit die benötigte Methode der `Controller/ GUI` auf.

7.1.2 Beispiel 2: Long Method

Die Methode `getObjekt()` der Klasse `Jobs/ TxtReader` wurde mit dem Commit `e76a2f429cd99d051785a882ebfad4a4d8bbd3d3` hinzugefügt. Dadurch, dass diese Methode viele in sich verschachtelte Schleifen besaß, wurden diese im Laufe des Refactoring in eigene Methoden ausgelagert. (Listings 7.1.2 & 7.1.2)

```
1  public static List<Entity> getObjekt(EntityStatus entityStatus) {
2      List<Entity> objektList = new ArrayList<>();
3      String [] text = readYml();
4
5      String [] objektWithTags;
6      if (text != null) {
7          text = text[2].split(";");
8          for (String s : text) {
9              List<String> tagList = new ArrayList<>();
10             String [] objekt = s.split(",");
11             for (String value : objekt) {
12                 tagList.add(value);
13             }
14             String bezeichnung = tagList.get(0);
15             tagList.remove(0);
16             objektList.add(new Objekt(bezeichnung, tagList));
17         }
18     }
19     return objektList;
20 }
21
```

Listing 7.2: Code Smells - Jobs/ TxtReader - Alter Stand

```
1  List<Category> getObject() {
2      List<Category> objektList = new ArrayList<>();
3      String [] text = this.handlerTxt.readTxt();
4
5      if (text != null) {
6          text = text[2].split(",");
7          for (String s : text) {
8              String [] objekt = s.split(";");
9              String bezeichnung = objekt[0];
10             objektList.add(new Objekt(bezeichnung, getTags(objekt)));
11         }
12     }
13     return objektList;
14 }
15
16 List<Tag> getTags(String [] objekt) {
17     List<Tag> tagList = new ArrayList<>();
18     for(int objektAttribute = 1; objektAttribute < objekt.length;
19     objektAttribute++) {
20         tagList.add(Tag.getTag(Integer.parseInt(objekt[objektAttribute])));
21     }
22     return tagList;
23 }
```

Listing 7.3: Code Smells - Jobs/ EntityBuilder - Aktueller Stand

7.1.3 Beispiel 3: Large Class

Die Klasse Controller/ Steuerung im Commit 04cc1fa86e0b46e3b92f54f2eea24b0129fc0abc beinhaltet mehr Logik und erledigt mehr Aufgaben als sie eigentlich sollte. Aus diesem Grund wurde diese Klasse in mehrere einzelne aufgeteilt. Die Methoden der Klasse Controller/ Steuerung sind aufgeteilt auf die Klassen Controller/ SearchElements/ Filter, Controller/ SearchElements/ FilterObjektIdea und Controller/ SearchElements/ Idea. (UMLs 7.1 & 7.2)

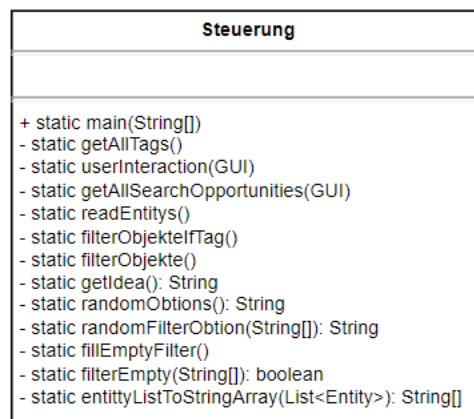


Abbildung 7.1: Code Smells - Controller/ Steuerung - Alter Stand

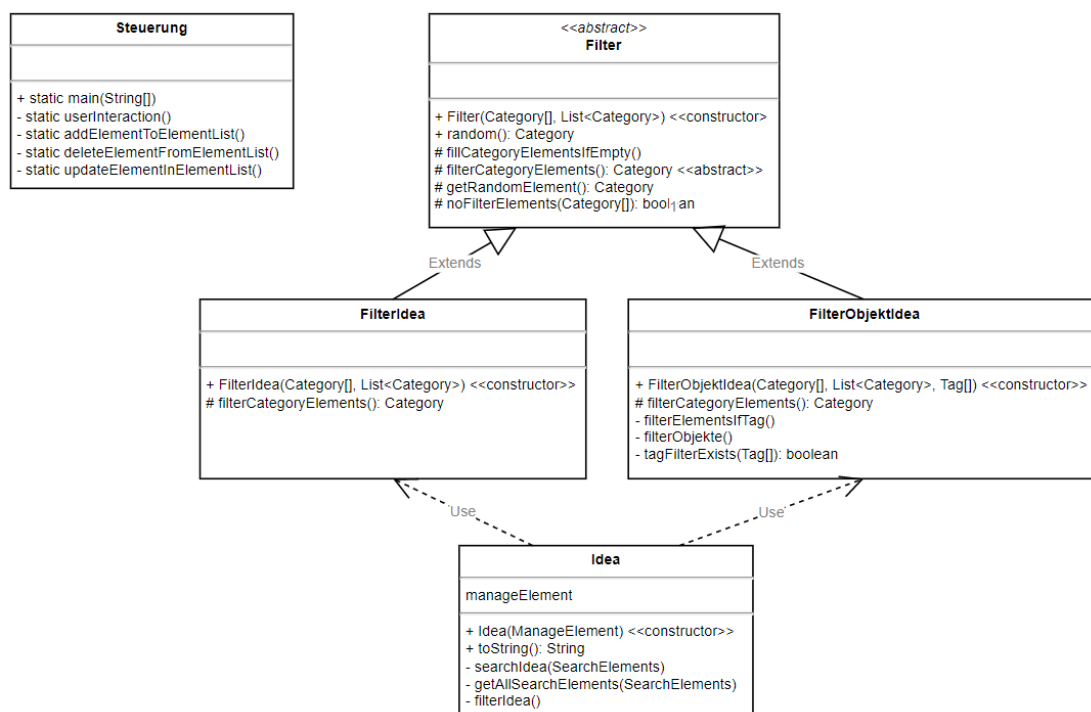


Abbildung 7.2: Code Smells - Controller/ Steuerung - Aktueller Stand

7.1.4 Beispiel 4: Duplicated Code

Mit dem Commit 2e38bff96607757452febf8ccb9a1a49587ebc82 wurden die Klassen `Controller/ Element/ AddElement`, `Controller/ Element/ DeleteElemente` und `Controller/ Element/ UpdateElement` hinzugefügt. Dabei wurde die Methode `getEntityStatus()` in jede dieser Klassen angelegt. (Listing 7.1.4)

```
1  private EntityState getEntityStatus(String kriterium) {
2      for (EntityStatus status : EntityState.values()) {
3          if (status.toString().equals(kriterium.toLowerCase(Locale.ROOT))) {
4              return status;
5          }
6      }
7      return null;
8  }
```

Listing 7.4: Code Smells - `Controller/ Element/` - Alter Stand

Diese Methode wurde als Getter Methode in die Klasse `Entity/ CategoryStatus` ausgelagert. (Listing 7.1.4)

```
1  public boolean isEqualCategory(CategoryStatus category) {
2      return category.equals(this);
3  }
```

Listing 7.5: Code Smells - `Entity/ CategoryStatus` - Aktueller Stand

7.2 4 Refactorings

7.2.1 Beispiel 1: Extract Method

Die Methode `getTag()` wurde aus der Methode `getObjekt()` der Klasse `Jobs/ TxtReader` mit dem Commit 2915e457463d6d46af71d336d2cad89aa40b2183 ausgelagert. Dies ermöglicht eine bessere Wiederverwendbarkeit der Methoden sowie eine bessere Lesbarkeit des Codes. (UMLs 7.3 & 7.4)

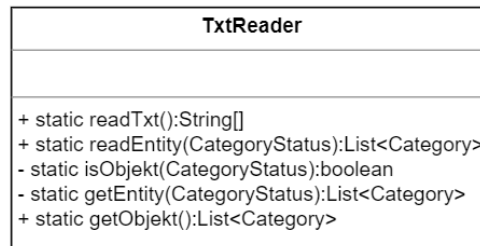


Abbildung 7.3: Refactoring - Jobs/ TxtReader - Alter Stand

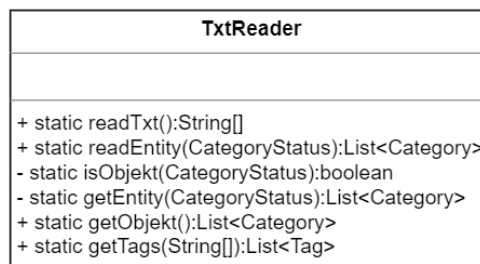


Abbildung 7.4: Refactoring - Jobs/ Refactoring - Verbesselter Stand

7.2.2 Beispiel 2: Rename Method

Der Commit cbe955b62febb968b2aa03be2ce23bf8770846f2 zeigt die Umbenennung der Methode `addNewElement()` der Klasse `Controller/ Element/ AddElement` zu `addNewElementOfTypeCategory()`. Mit der Umbenennung ist es für den Menschen lesbarer und besser zu verstehen, was genau in der Methode passiert. (UMLs 7.5 & 7.6)

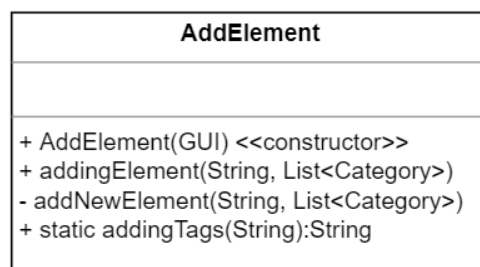


Abbildung 7.5: Refactoring - Controller/ Element/ AddElement - Alter Stand

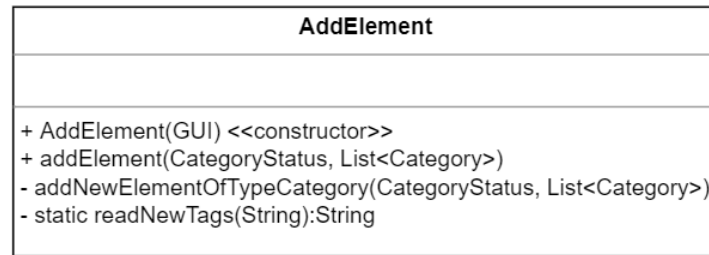


Abbildung 7.6: Refactoring - Controller/ Element/ AddElement - Verbesserter Stand

7.2.3 Beispiel 3: Replace ErrorCode with Exception

In der Methode `trueBooleanQuestion()` der `Controller/ GUI` Klasse wurde der Fall einer falschen Eingabe des Users nicht abgefangen. Mit dem Commit `0198311940ea4a9f442946d889ea2ae87e2069a8` wurde dieser Fall abgefangen, mit Hilfe einer eigens kreierte `FalseInputException()`, die dem User auf den Fehler einer falschen Eingabe hinweist. Damit konnte der Fehler klar definiert werden und der Code ist verständlicher und lesbarer geworden. (UMLs 7.7 & 7.8)

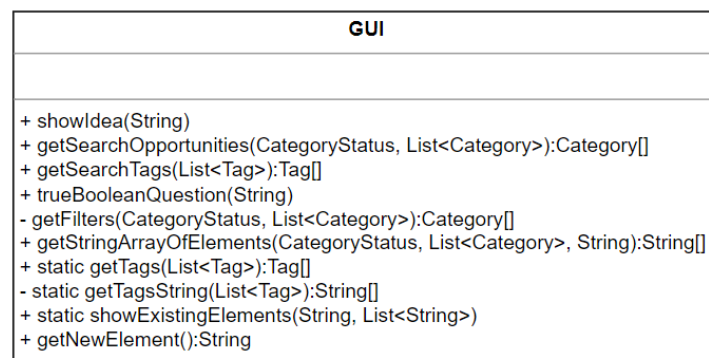


Abbildung 7.7: Refactoring - Controller/ GUI - Alter Stand

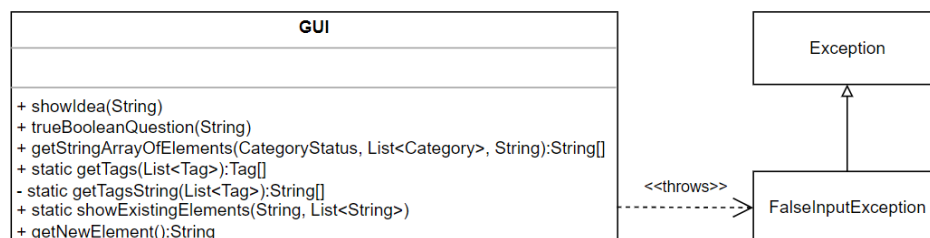


Abbildung 7.8: Refactoring - Controller/ GUI - Verbesserter Stand

7.2.4 Beispiel 4: Replace Conditional with Polymorphism

Mit dem Commit 98152f70088414aac9ef95273e30300a0448bf24 wurden die Entitäten Polymorph gestaltet. Die Entitäten **Art** und **Stimmung** wurden zur Entität **Entity/Entity** (später **Entity/ SimpleCategory**) zusammengeführt. Diese wurden mit dem Commit d10b862066f1c3136854fe5a881064c50d82d8d1 als abstrakt definiert. Dies hat den Vorteil, dass weitere **Categories** dynamisch hinzugefügt werden können. Die neue Klasse **Entity/ Objekt** erbt von **Entity/ Entity**. Mit diesem Verhalten ist es möglich die Software besser zu kapseln. (UMLs 7.9 & 7.10)

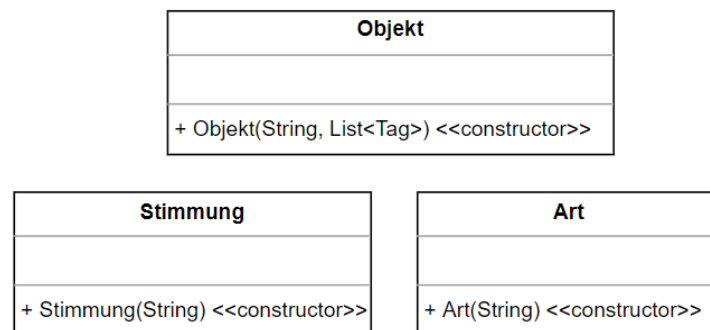


Abbildung 7.9: Refactoring - Entity/ Entity - Alter Stand

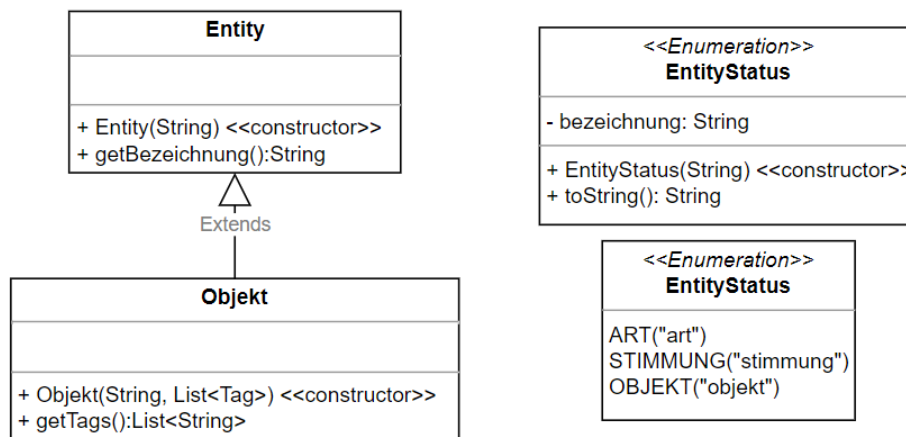


Abbildung 7.10: Refactoring - Entity/ Entity - Verbesselter Stand

8 Entwurfsmuster

8.1 Entwurfsmuster: [Erzeugungsmuster]

In der Klasse Jobs/ EntityBuilder wird ein Erzeugungsmuster eingesetzt. Die Erstellung von Entity/ SimpleCategory und Entity/ Objekt wird in dieser Klasse durchgeführt (getEntity(CategoryStatus) und getObjekt()). Somit wird die Erstellung der Objekte von deren Verwendung getrennt und das System ist unabhängiger von der Implementierung der Objekte. (UML 8.1)

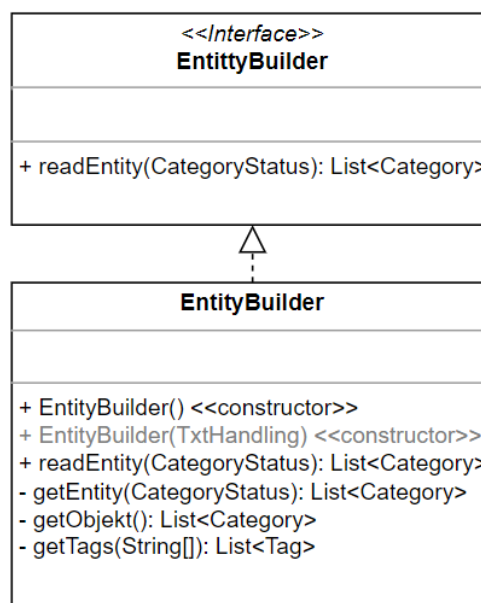


Abbildung 8.1: Erzeugungsmuster - Jobs/ EntityBuilder

8.2 Entwurfsmuster: [Strukturmuster]

Ein Strukturmuster wird als Entwurfsmuster in der Klasse `Controller/ SearchElements/ Idea` verwendet. In der Klasse wird eine größere Struktur geschaffen, um eine Idee ausgeben zu können. Dafür müssen 3 Objekte der Klasse `Entity/ Category` zusammen ausgegeben werden. (UML 8.2)

`Controller/SearchElements/Idea = Entity/Category + Entity/Category + Entity/Category`

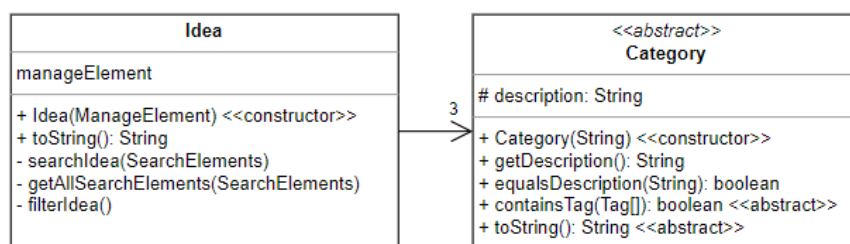


Abbildung 8.2: Erzeugungsmuster - `Controller/ SearchElements/ Idea`

8.3 Entwurfsmuster: [Verhaltensmuster]

Ein Verhaltensmuster wurde als ein weiteres Entwurfsmuster in der Klasse `Controller/ SearchElements/ Idea` angewendet. Die Methode `filterIdea()` überträgt die Verantwortung zur Filterung der Ideeelemente an die Klassen `Controller/ SearchElements/ FilterIdea` und `Controller/ SearchElements/ FilterObjektIdea`. Diese filtern alle Elemente und geben ein zufälliges Element zurück.

Dadurch kann `filterIdea()` sich das Verhalten von `Controller/ SearchElements/ FilterIdea` und `Controller/ SearchElements/ FilterObjektIdea` zu nutzen machen. (UML 8.3)

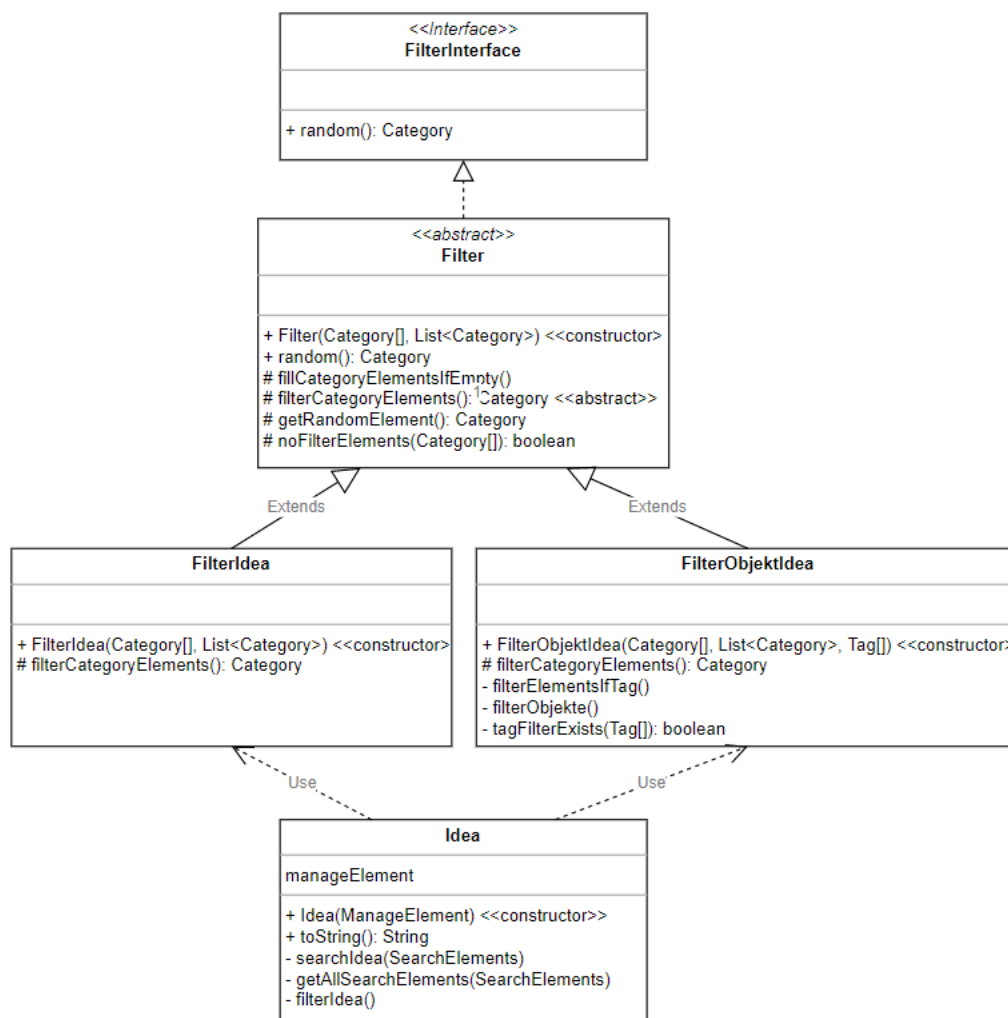


Abbildung 8.3: Erzeugungsmuster - Controller/ SearchElements/ Filter

8.4 Entwurfsmuster: [Verhaltensmuster]

Auch in der Klasse `Controller/ Element/ UpdateElement` wird ein Verhaltensmuster als Entwurfsmuster angewendet. Hier erkennt man in der `finalUpdate()` Methode, dass, wenn ein Element bearbeitet werden soll, dieses zuerst komplett gelöscht und danach neu hinzugefügt werden muss. Damit werden Algorithmen der Klasse `Controller/ ManageElement` und das Verhalten von `Controller/ Elements/ AddElement` genutzt, um eine Bearbeitung von Elementen möglich zu machen. (UML 8.4)

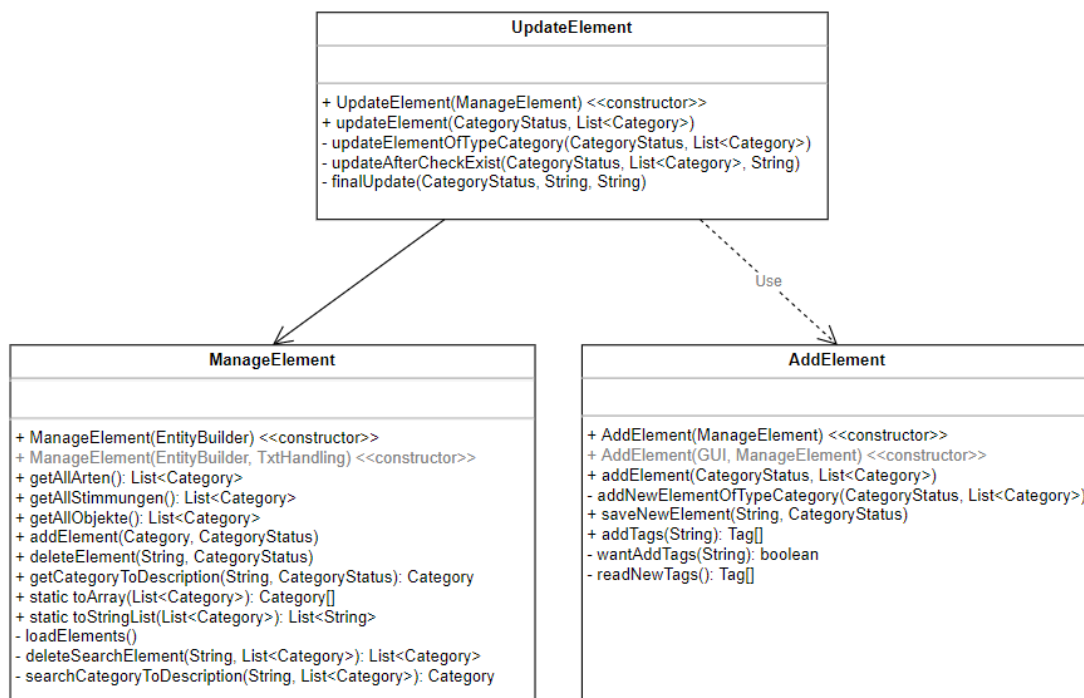


Abbildung 8.4: Erzeugungsmuster - Controller/ Element/ UpdateElement