

# Programmmentwurf

## MatLearn

Name: Meier, Julien

Matrikelnummer: 8236502

Abgabedatum: 29.05.2022

### Allgemeine Anmerkungen:

- *es darf nicht auf andere Kapitel als 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)*
- *sollten mündliche Aussagen den schriftlichen Aufgaben widersprechen, gelten die schriftlichen Aufgaben (ggf. an Anpassung der schriftlichen Aufgaben erinnern!)*
- *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*
  - *Beispiele*
    - *"Nennen Sie jeweils eine Klasse, die das SRP einhält bzw. verletzt."*
      - *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: volle Punktzahl ODER falls im Code mind. eine Klasse SRP verletzt: halbe Punktzahl*
- *verlangte Positiv-Beispiele müssen gebracht werden*
- *Code-Beispiel = Code in das Dokument kopieren*

# Kapitel 1: Einführung

## Übersicht über die Applikation

MatLearn ist eine CLI-Applikation, die dabei helfen soll das Mathematiklernen besser zu strukturieren, einfach durchsuchbar zu machen und dazu Abfragefunktionalität fürs Wiederholen zu bieten. Mathematische Objekte, darunter zum Beispiel Axiome oder Theoreme, müssen unterschiedliche Anforderungen auch in Hinblick auf ihre „Stellung“ in Bezug auf andere mathematische Objekte erfüllen, um valide zu sein. MatLearn unterscheidet dabei zwischen schwachen Validitätskriterien, die temporär relaxiert werden dürfen, damit das Netzwerk gut aufgebaut werden kann, und strengen Validitätskriterien, die keineswegs fehlerhaft sein dürfen – etwa, dass ein Axiom niemals von etwas anderem logische abhängen darf. Wenn ein Nutzer nun das Netzwerk bearbeiten möchte, so wird stets geprüft, ob alle schwachen Validitätskriterien eingehalten werden.

## Wie startet man die Applikation?

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

Anforderungen:

- Java 11
- Maven
- JAR mit ausreichenden Berechtigungen fürs Schreiben in dem Ordner

Clone das Repository von <https://github.com/keksklauer4/MatLearn.git>, dann bauen durch „mvn package“ und ausführen durch:

„java -jar 00-plugins-1.0-SNAPSHOT-jar-with-dependencies.jar“ im Ordner „00-plugins/target“

Sobald die Applikation läuft, werden Kommandos ausgegeben, deren Nummer oder auch Text man eingeben kann, um das jeweilige Kommando auszuführen. Dabei wird dann noch nach den Parametern gefragt, die ebenfalls über Nummern ausgewählt werden müssen. Die Eingaben werden automatisch validiert. Sobald alle Parameter übergeben wurden und diese valide sind, so wird das jeweilige Kommando ausgeführt und ein Resultat zurückgegeben.

## Wie testet man die Applikation?

*[Wie testet man die Applikation? Welche Voraussetzungen werden benötigt? Schritt-für-Schritt-Anleitung]*

Es wurden Unit Tests geschrieben, die über Maven ausgeführt werden können:

```
mvn test
```

im Root-Ordner reicht, um die Tests auszuführen.

## **Kapitel 2: Clean Architecture**

### **Was ist Clean Architecture?**

Wie die Bezeichnung „Clean Architecture“ bereits impliziert, geht es hierbei um eine Herangehensweise, wie ein Softwareprojekt architektonisch „sauber“ gestaltet werden kann. Clean Architecture soll nun anhand von klassischen Problemen motiviert werden. In einem größeren Projekt werden häufig einige Technologien benötigt, darunter u. a. Datenbankenhandler, Object-Relational Mapper, UI-Libraries, Serialisierungslibraries oder auch unterschiedliche Optimierer. Die Schwierigkeit dabei ist, dass ein Entwickler sich u. U. dazu verführen lässt, sich zu sehr auf eine Library oder ein Framework zu verlassen. Der Technologiestack kann sich allerdings nach einigen Jahren durchaus ändern – was ist wenn Spring mal nicht mehr gewartet wird oder es breaking changes gibt? Daraus lässt sich also die Anforderung ableiten, dass die Architektur eines Softwareprojektes möglichst eine Abhängigkeit von bestimmten Libraries/Frameworks verhindern soll – also die Technologie so weit zu abstrahieren, dass Technologieänderungen vergleichsweise einfach durchzuführen sind. Eine weitere Schwierigkeit ist die Kurzlebigkeit von Libraries/Frameworks, während die Businesslogik typischerweise eher statisch ist. Wenn eine Library ausgetauscht wird, besteht die Gefahr, dass die Businesslogik sich ungewollt verändert. Ein Beispiel: Wenn Businesslogik in einem Callback eines Buttons in einem UI-Framework, ausgeführt wird (also direkt im Callback die Logik definiert wird), so wird es äußerst anstrengend das UI auszutauschen – und ggf. entstehen dabei sogar noch Bugs. Folglich ist es sinnvoll Businesslogik von Bibliothekslogik zu separieren. Und „mal schnell“ eine neue Bibliothek in den Code hineinzubringen führt typischerweise dazu, dass genau diese Separation missachtet wird.

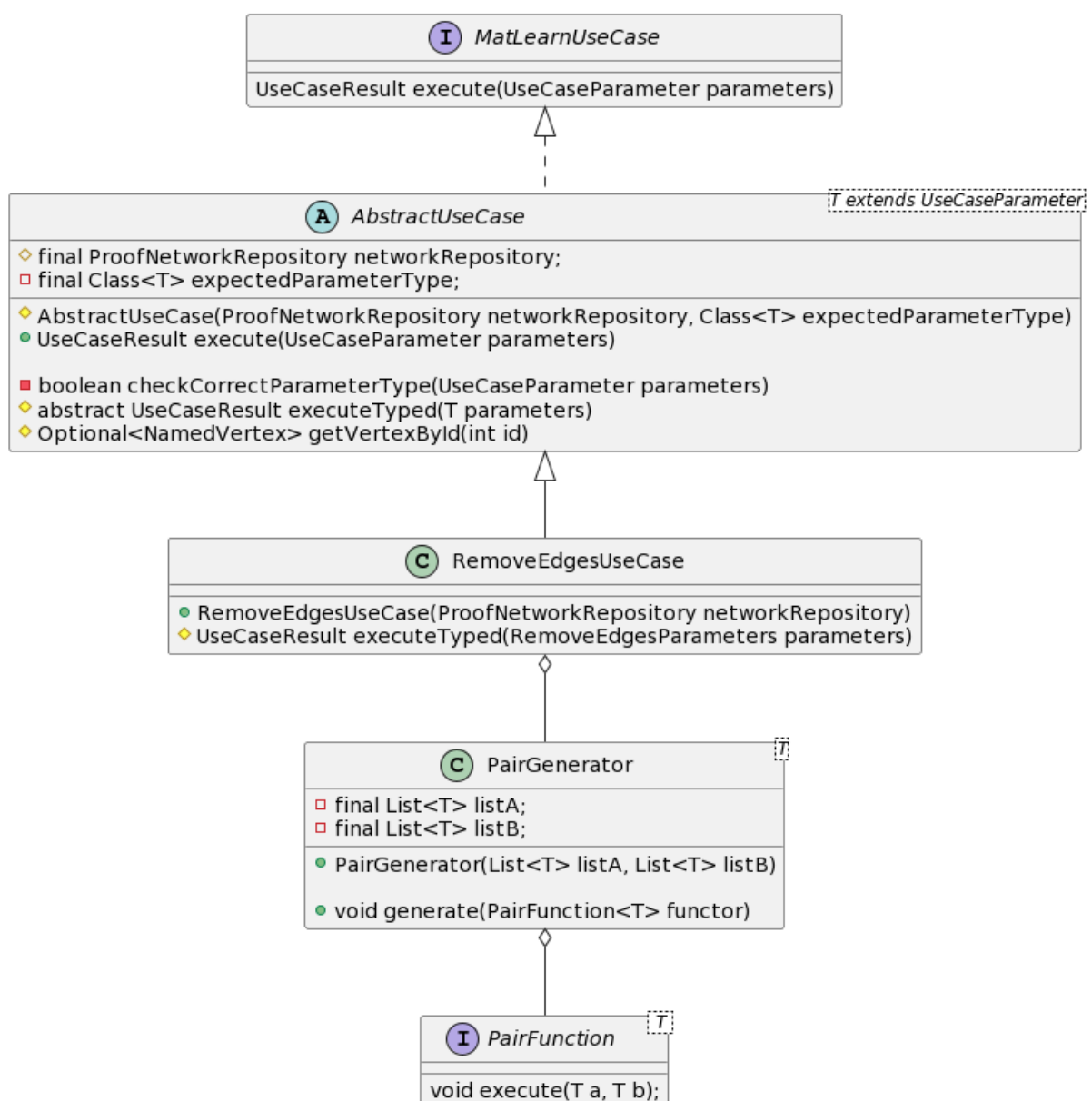
Aus den oben genannten Schwierigkeiten folgen Anforderungen Separation (in Schichten), Abstraktion (von Bibliotheken/Frameworks)

### **Analyse der Dependency Rule**

*[(1 Klasse, die die Dependency Rule einhält und eine Klasse, die die Dependency Rule verletzt); jeweils UML der Klasse 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]*

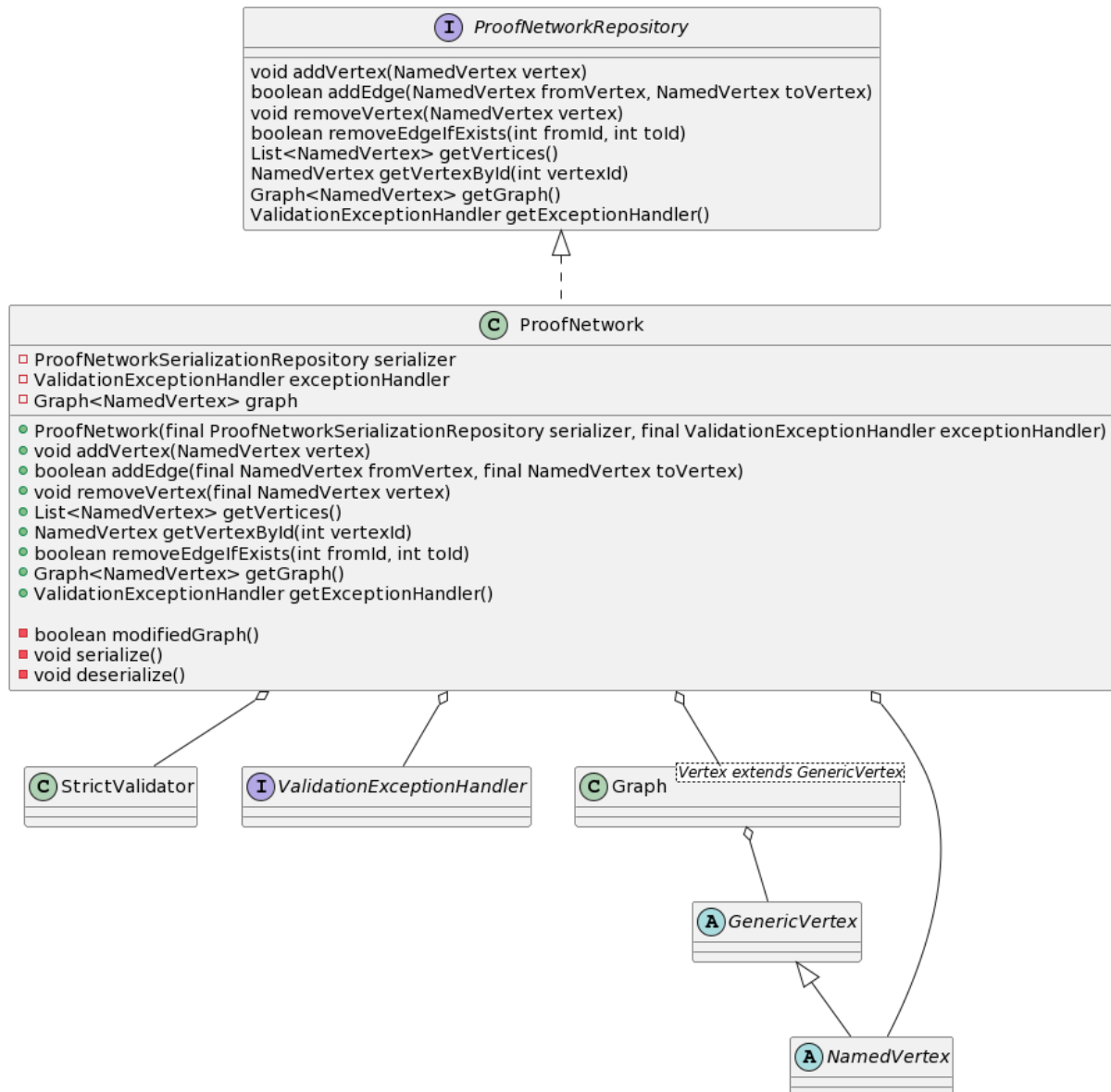
## Positiv-Beispiel: Dependency Rule

Ich habe mein Projekt so aufgesetzt, dass von Anfang an die Dependency Rule immer eingehalten wird, solange die Klassen in der richtigen Schicht eingeordnet wurden. Das liegt daran, dass das Projekt in Modulen aufgebaut ist, die sich gegenseitig als Abhängigkeiten aufführen. Dabei hat die jeweils äußere Schicht alle inneren Schichten als Abhängigkeiten, aber keine innere Schicht hängt von einer äußeren ab. Das ist gar nicht möglich, da sonst „circular dependency“-Fehler auftreten würden. Deshalb wähle ich hier die Klasse PairGenerator, die von einer Use-Case-Klasse in derselben Schicht genutzt wurde und selbst ein Interface PairFunction aus derselben Schicht nutzt. Dementsprechend ist das ein positives Beispiel für die Dependency Rule.



## Negativ-Beispiel: Dependency Rule

Gleichwohl durch die Zyklen die Dependency Rule formal stets eingehalten wird, habe ich durchaus einen Fehler gehabt, da die Implementierung des ProofNetworkRepositories in der falschen Schicht stattfand. Die Implementierung war zunächst im Domain-Code, sollte allerdings in der Plugin-Schicht zu finden sein und an die Klassen durchgereicht werden. Dies habe ich dann behoben. Im UML-Diagramm wurden die Use-Case-Klassen aktiv ausgelassen, damit es einigermaßen übersichtlich bleibt. Alle Use-Case-Klassen erhalten das ProofNetworkRepository als Objekt.



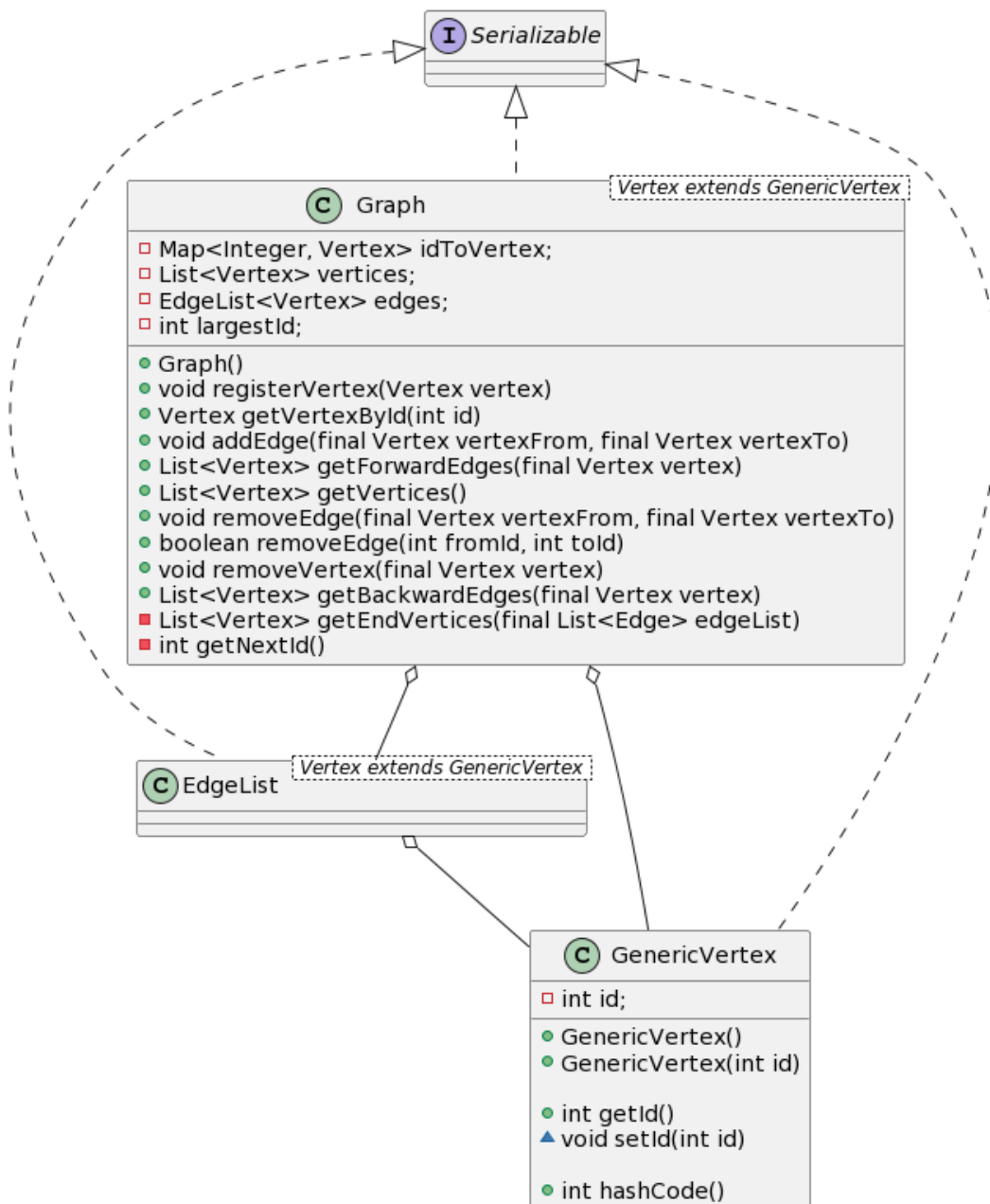
## Analyse der Schichten

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

### ***Schicht: Abstraction-Code***

In der innersten Schicht, dem Abstraction-Code, entschied ich mich einen Graphen generisch zu implementieren, der Kanten speichern, (de-)serialisiert werden und durchsucht werden kann. Dabei heißt die Klasse Graph und kriegt einen Generic „Vertex“ übergeben, der von GenericVertex erbt. GenericVertex bietet eine Id, die abgefragt werden und zudem gesetzt werden kann. Das Setzen der id ist package-private und kann somit nur im Abstraction-Code geschehen – also in diesem Fall durch die Klasse Graph. Es ist sinnvoll die Graph-Klasse und eine generische Tiefensuche (Klasse: GenericDFS) im Abstraction-Code zu implementieren, da ein Graph ein Konzept ist, dass meine Domäne weit übersteigt – selbiges gilt für Tiefensuchen, die in den meisten Graph-Algorithmen vorkommen. Abgesehen davon kann man stark davon ausgehen, dass die Graph-Klasse bis auf ein paar Kleinigkeiten stets gleich bleiben wird, weil es das Konzept eines Graphen etabliert ist.

Der Graph hat einen Member EdgeList, der sich um das Verwalten der Kanten kümmert. Aufrufe werden dann an die EdgeList delegiert. Der Graph selbst kümmert sich um die Vertices, da diese registriert werden müssen, sowie gelöscht und abgefragt werden können. Durch die Nutzung von Generics ermöglicht man, dass allgemeinere Objekte gespeichert werden. In meinem Fall erben die Entitäten aus der Domänen-Schicht von GenericVertex und werden über ein Repository im Graphen gespeichert.



### Schicht: Application-Code

Der Application-Code besteht bei mir größtenteils aus den Use-Case-Klassen, deren Parameter und die Resultate der Use-Cases. Dabei sind die Use-Cases effektiv Domain-Services und somit stateless (bis auf das ProofNetworkRepository als Member). Hier möchte ich den Domain-Service „ListLeavesUseCase“ vorstellen, der die Id eines mathematischen Objects gewrapped in einem Objekt der Klasse SingleObjectParameters erhält und die „Blätter“ des Subgraphen unter diesem Objekt ausgibt. Der Subgraph bezieht sich hierbei auf den gerichteten Graphen, der aus der Menge der forward edges (siehe Ubiquitous language) und der üblichen Knotenmenge entsteht. Ein Blatt ist dann ein mathematisches Objekt, von dem kein anderes Objekt logisch abhängt (also keine einkommenden forward edges hat). Das könnten etwa

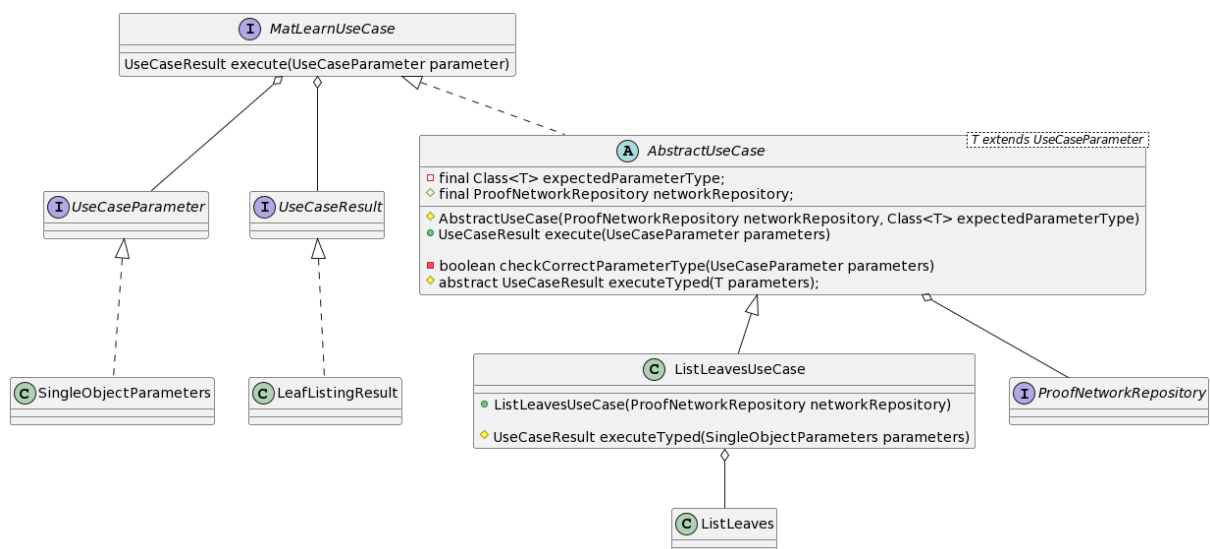


Axiome oder Definitionen sein. Dafür ist dieser Use-Case auch da: Um die Axiome und Definitionen eines bewiesenen Resultats zu erfahren.

Weil es sich hier um Use-Cases handelt passen diese gut in die Application-Code-Schicht, denn es geht hier um anwendungsspezifischen Code. Es gibt immerhin viele weitere Möglichkeiten, was man mit dem Datenbestand machen könnte. Diese Applikation beschränkt sich allerdings aufs Erstellen, Löschen, Validieren, Durchsuchen und Abfragen der mathematischen Objekte.

Im UML-Diagramm wurden aktiv Details ausgelassen, um den Fokus auf die wichtigen Details zu setzen. MatLearnUseCase ist ein Interface, das das Verhalten eines Use-Case beschreibt. Ein Use-Case hat immer eine execute-Methode, die ein UseCaseParameter-Objekt annimmt und ein UseCaseResult-Objekt zurückgibt. Beide Interfaces sind sogenannte Marker-Interfaces, die dazu dienen verbesserte Typsicherheit in Dispatchern zu liefern. Sowohl die UseCaseParameter- als auch die UseCaseResult-Klassen sind sehr unterschiedlich und haben deshalb in den jeweiligen Interfaces keine gemeinsame Funktionalität.

Der Use-Case ListLeavesUseCase erbt von AbstractUseCase<SingleObjectParameter>. AbstractUseCase implementiert das Interface MatLearnUseCase, hat somit die execute Methode und versucht das Objekt vom Typ UseCaseParameter sicher in ein SingleObjectParameter zu casten, indem Typinformationen geprüft werden. Im Allgemeinen ist ohnehin sichergestellt, dass dieser Cast valide ist, da die Parameter über einen Dispatcher gebaut und übergeben werden. Wenn das funktioniert, wird die abstrakte Methode executeTyped aufgerufen, die vom ListLeavesUseCase implementiert wird. Dort wird nun auf das ProofNetworkRepository zugegriffen und die eigentliche Tiefensuche an ListLeaves delegiert. Das Resultat wird als Objekt der Klasse ListLeavesResult zurückgegeben.



## Kapitel 3: SOLID

### Analyse Single-Responsibility-Principle (SRP)

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

#### **Positiv-Beispiel**

Ein gutes Beispiel hier ist der LeafValidator. In der Domain-Logik steht fest, dass ein bewiesenes Resultat (also Lemma, Korollar oder Theorem) als Blätter im Subgraph unter sich nur Definitionen oder Axiome haben darf. Offensichtlich wäre es schlecht, wenn ein Resultat von einem anderen Resultat abhinge, das nicht wirklich auf Grundlage von Axiomen/Definitionen bewiesen wurde. Der LeafValidator macht also eine Tiefensuche durch den Subgraphen unter dem jeweiligen Resultat, merkt sich alle Blattknoten und prüft dann, ob die Blattknoten Definitionen oder Axiome sind. Wenn das gilt, so ist der jeweilige Knoten valide.

Diese Validitätsprüfung wird ausschließlich auf Kommando ausgeführt, da es beim Aufbau des Netzwerkes durchaus dazu kommen darf, dass ein Theorem noch gar nicht bewiesen ist und von nichts abhängt. Wenn allerdings eine vollständige Validierung durchgeführt wird, so wird der LeafValidator aufgerufen, prüft das und gibt ggf. einen Fehler zurück.

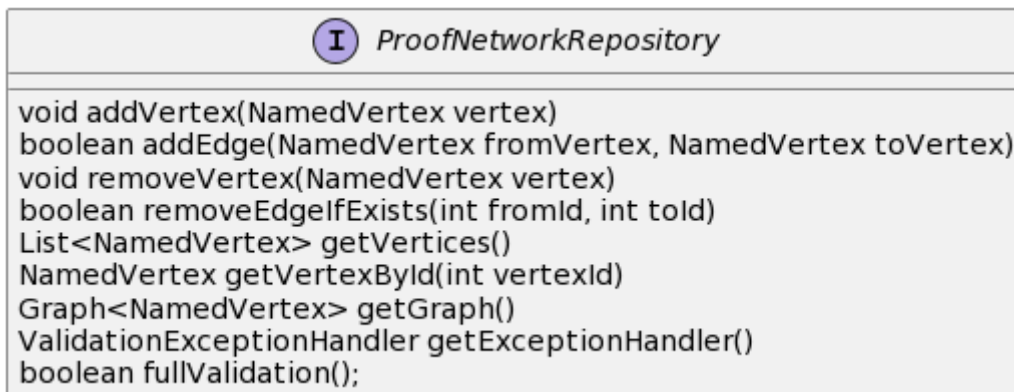
Der LeafValidator prüft nur das, validiert also nur die Blätter im Subgraphen auf ihren Typ und gibt den Fehler zurück.



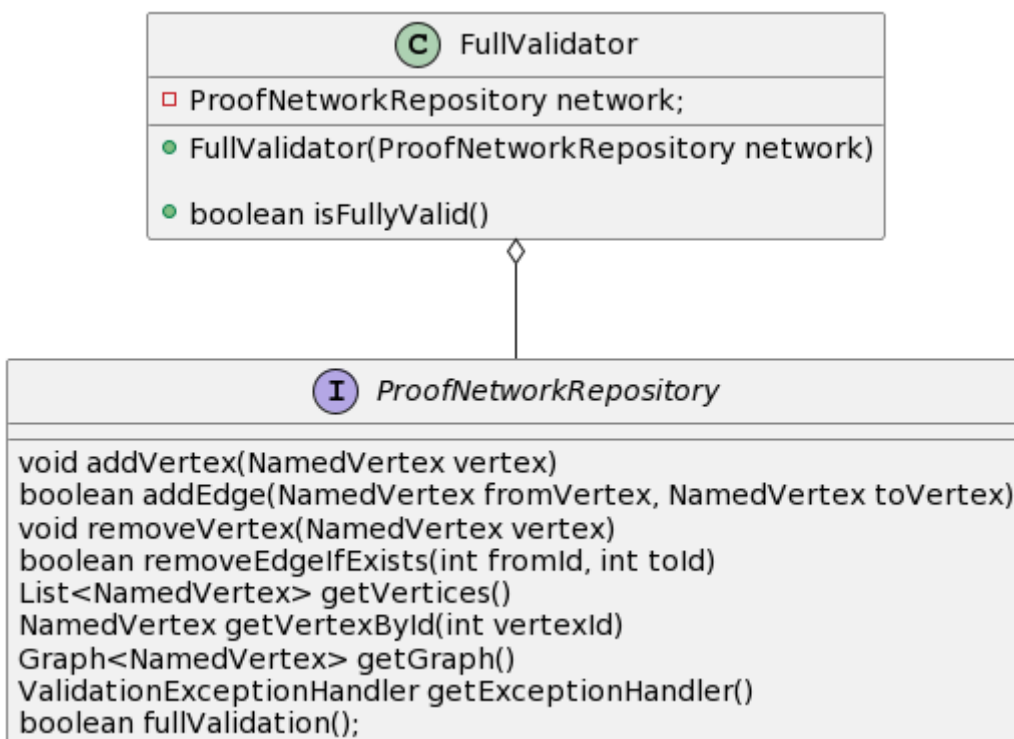
### Negativ-Beispiel

Mein ProofNetworkRepository soll eigentlich klassische Repository-Aufrufe handeln – also generiere (hier registriere) ein neues mathematisches Objekt, füge logische Abhängigkeiten hinzu, entferne Objekte bzw. Abhängigkeiten und Abfragen zum Datenbestand. Stattdessen gab es auch die Methode fullValidation, die, wie der Name es nahelegt, eine vollständige Validierung des Datenbestandes durchführen soll. Das passt meiner Meinung nach nicht zu der Verantwortung des Repositories und sollte deshalb in eine andere Klasse FullValidator ausgelagert werden. Das habe ich tatsächlich auch getan.

Vorher:



Und nach der Verbesserung:

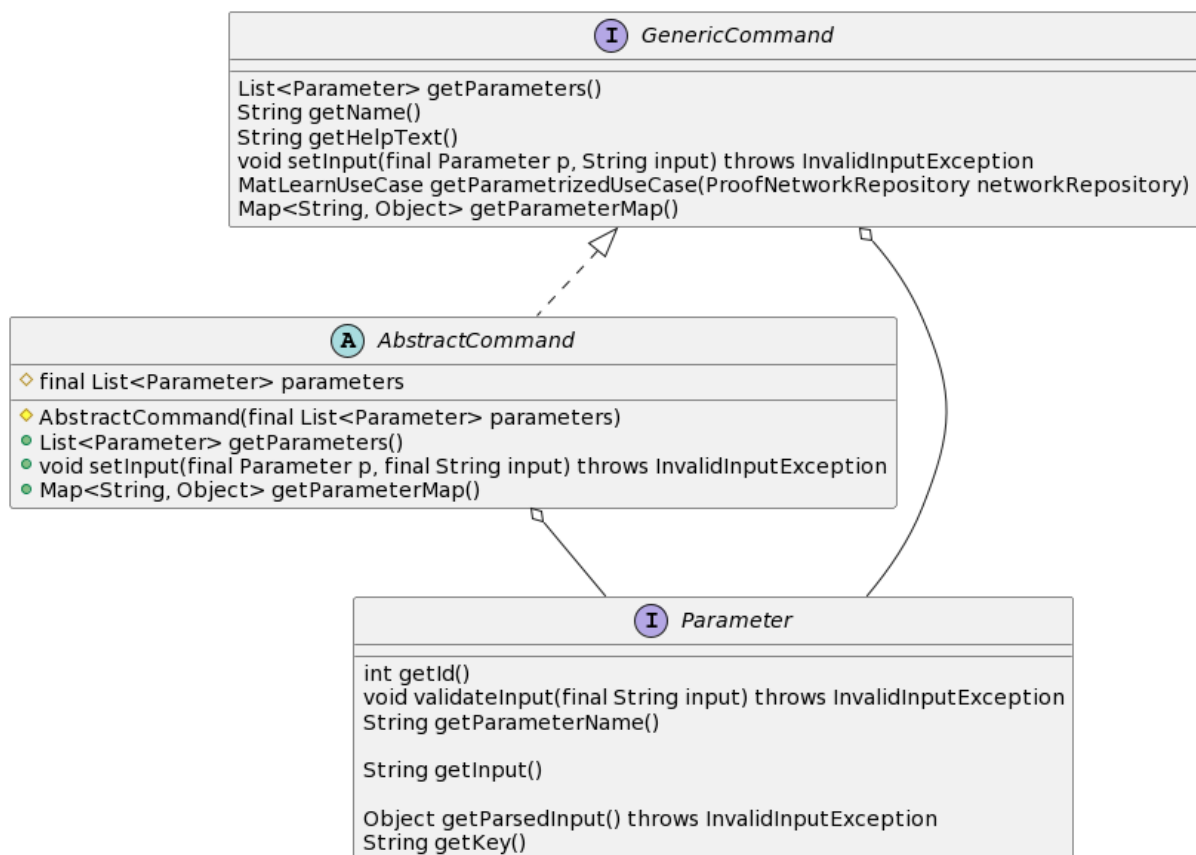


## Analyse Open-Closed-Principle (OCP)

*[jeweils eine Klasse als positives und negatives Beispiel für OCP; jeweils UML der Klasse 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: AbstractCommand + Interface Parameter

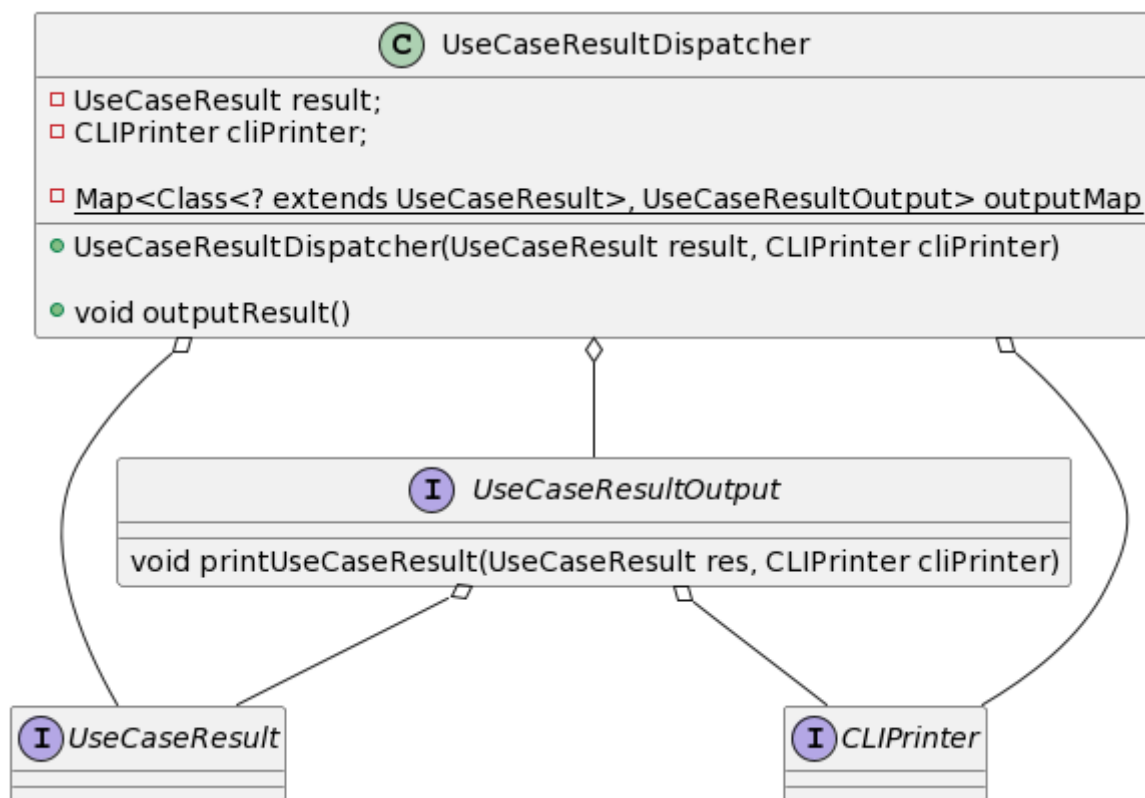
Die verschiedenen Commands, die jeweils einen Command mit verschiedenen Parametern fürs UI darstellen sollen, passen gut, da die Parameter-Typen problemlos erweitert werden können und das CLI immer noch damit klar kommt. Konkret können unterschiedliche Parameter implementiert werden, indem sie das Interface Parameter implementieren. Dabei werden unbedingt notwendige Funktionalitäten implementiert. Die ganze Funktionalität des CLI und der Commands nutzt nur noch allgemein das Interface Parameter zur Interaktion mit den konkreten Aufrufen – außer beim Konstruieren der Commands, da hier natürlich der korrekte Konstruktor des Parameters aufgerufen werden soll.



Negativ-Beispiel: UseCaseResultDispatcher

Der `UseCaseResultDispatcher` nimmt ein `UseCaseResult` an und soll es an den korrekten Output handler für das Objekt leiten. Dabei wird die Klasse des `UseCaseResults` genutzt, um in der Map das korrekte Output-Objekt zu finden. Alle Output-Objekte implementieren jeweils

das Interface UseCaseResultOutput und haben die Methode printUseCaseResult. Das Problem hierbei ist, dass für die Map eine statische Variable genutzt wird. Dadurch kann man die Klasse nicht durch erweitern, sondern muss bei einer neuen Resultat-Klasse die UseCaseResultDispatcher-Klasse ändern, also die Map erweitern. Ich halte es für unwahrscheinlich, dass die Klasse durch Vererbung erweitert werden würde. Stattdessen ist der Sinn des Dispatchers, dass er problemlos erweitert werden kann, indem nur ein Eintrag in der Map verändert werden muss. Dementsprechend würde ich da nichts dran ändern bzw. habe ich es nicht.



## Analyse Liskov-Substitution- (LSP), Interface-Segregation- (ISP), Dependency-Inversion-Principle (DIP)

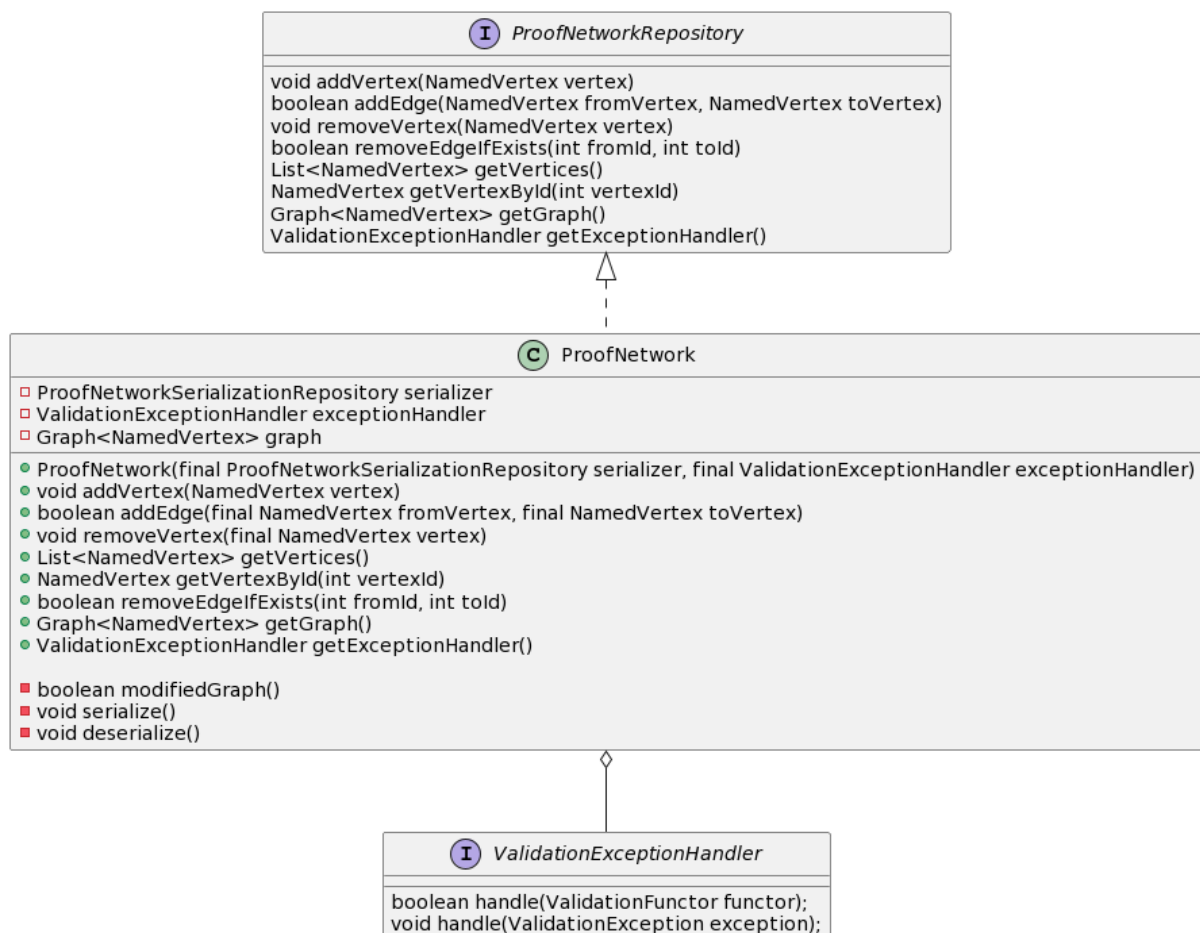
*[jeweils eine Klasse als positives und negatives Beispiel für entweder LSP oder ISP oder DIP); jeweils UML der Klasse und Begründung, warum man hier das Prinzip erfüllt/nicht erfüllt wird]*

*[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: DIP

Mein ProofNetworkRepository enthält einen ValidationExceptionHandler, der im Falle einer geworfenen ValidationException während der Validierung die Exception handeln kann. Dabei

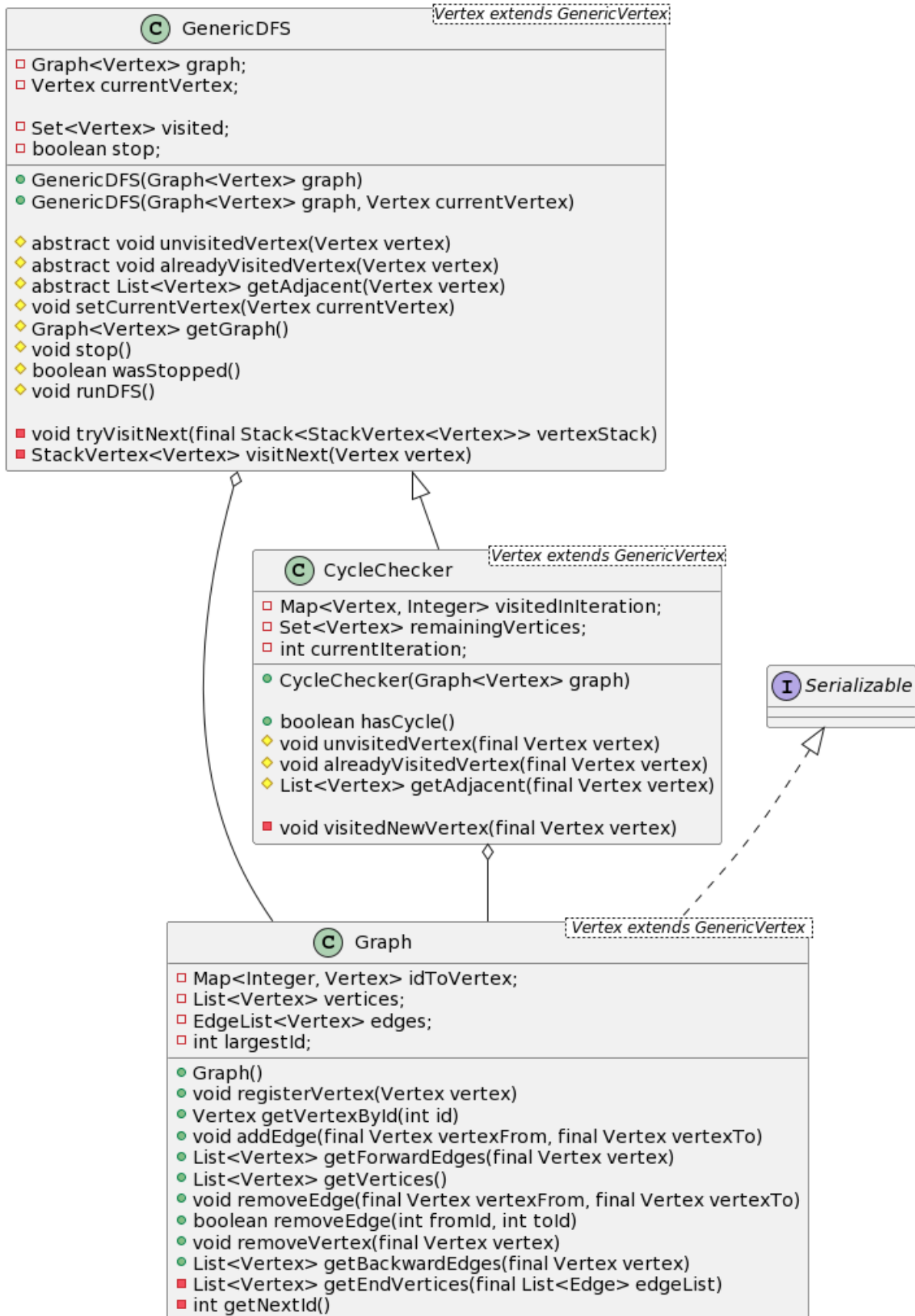
ist das Interface in der Domain-Schicht zu finden und die Implementierung des Handlers in der Plugin-Schicht. Der Exception-Handler wird in der Domain-Schicht dafür genutzt Validitätsprobleme im ProofNetworkRepository an die äußerste Schicht zu leiten. Wenn die Implementierung des ProofNetworkRepository, das ProofNetwork, eine Veränderung an dem Graphen vornimmt, so wird stets geprüft, ob der Graph noch valide ist. Jeder invalide Aspekt (maximal 1 pro Entität) führt zu einer ValidationException, die dann vom Handler genutzt wird, um sie im UI anzuzeigen. Dadurch hängt die innere Schicht nicht von der äußeren Schicht ab und die äußere Schicht auch nicht von der inneren Schicht ab. Der ValidationExceptionHandler wird dem ProofNetwork im Konstruktor dementsprechend als Objekt vom Typen des Interfaces übergeben.



### Negativ-Beispiel: DIP

Der CycleChecker, ein Algorithmus, der im Abstraction-Code anzutreffen ist, hängt direkt von der Klasse Graph ab, sodass eine Änderung von Graph auch eine Änderung der Klasse CycleChecker und alle anderen von GenericDFS erbenden Klassen inkl. GenericDFS selbst mit sich brächte. Es wäre hier also sinnvoll die Klasse Graph durch ein Interface auszutauschen und die eigentliche Implementierung des Interface als GraphImpl zu implementieren. Dadurch kann es auch leichter werden die Klassen zu mocken, die den Graph nutzen. Man beachte, dass

der CycleChecker im Abstraction-Code liegt, weil der Algorithmus an sich nicht domänenspezifisch ist.





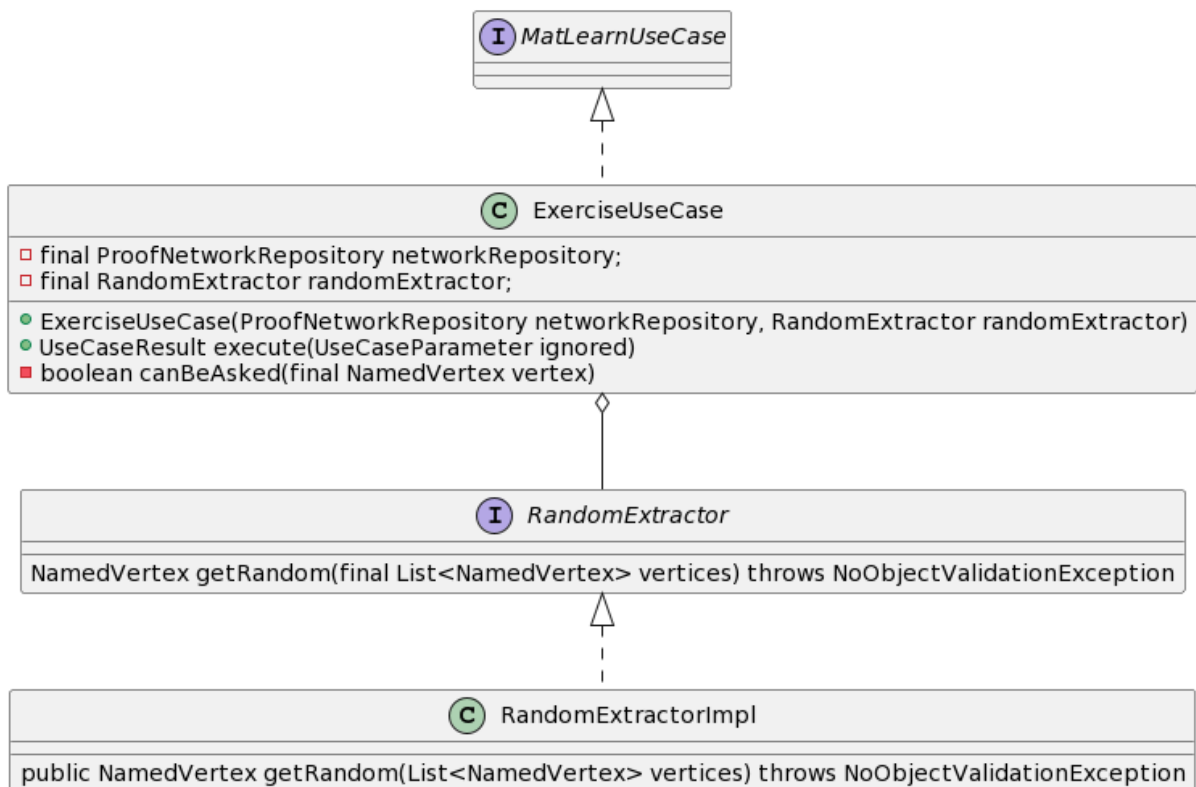
## Kapitel 4: Weitere Prinzipien

### Analyse GRASP: Geringe Kopplung

*[jeweils eine bis jetzt noch nicht behandelte Klasse als positives und negatives Beispiel geringer Kopplung; jeweils UML Diagramm mit zusammenspielenden Klassen, Aufgabenbeschreibung und Begründung für die Umsetzung der geringen Kopplung bzw. Beschreibung, wie die Kopplung aufgelöst werden kann]*

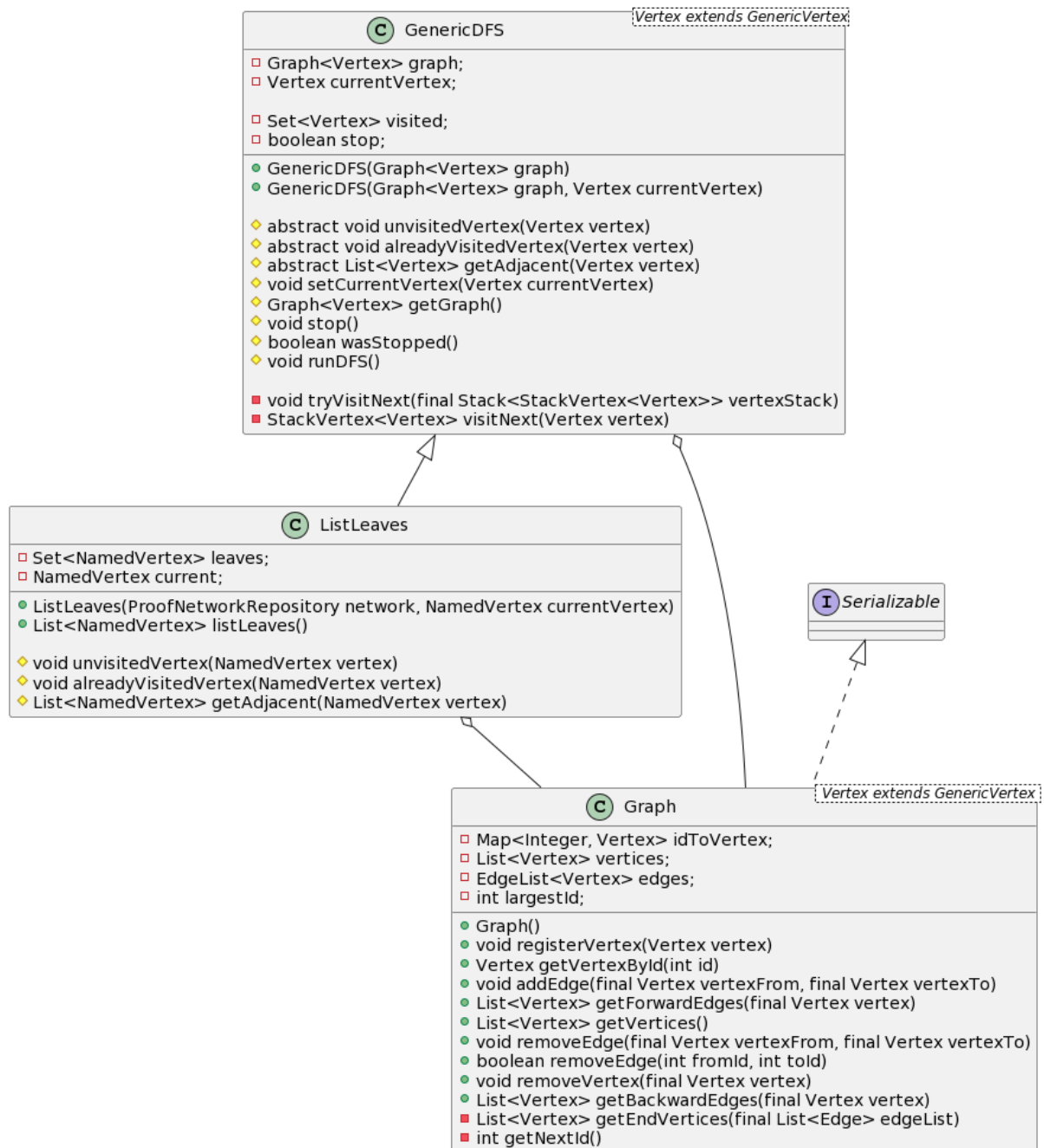
#### **Positiv-Beispiel: ExerciseUseCase + RandomExtractor**

Eine Klasse mit geringer Kopplung ist die Klasse ExerciseUseCase, die alle Vertices vom ProofNetworkRepository erfragt und sie danach filtert, ob sie abgefragt werden können. Bei dem Use-Case geht es darum, dass zufällig ein mathematisches Objekt abgefragt wird. Offensichtlich ergibt es wenig Sinn Objekte vom Typ Topic oder Source abzufragen, weshalb das Filtern mit dem Prädikat canBeAsked sinnvoll ist. Dann soll ein zufälliges Objekt aus der gefilterten Liste ausgewählt werden. Statt festzulegen, dass hier unbedingt ein bestimmter Zufallsgenerator genutzt werden soll, habe ich mich für ein Interface RandomExtractor entschieden, da dadurch der ExerciseUseCase deutlich weniger von der eigentlichen Implementierung des Extractors abhängt. Diese niedrige Kopplung hilft auch dabei, dass man hier sehr simpel ein Mock-Objekt im Konstruktor übergeben kann, um die Klasse zu testen. Das habe ich auch getan.



### ***Negativ-Beispiel: ListLeaves***

Für den ListLeavesUseCase wird eine Klasse benötigt, die die Blätter tatsächlich auch durch Tiefensuche auflistet, sodass der Use-Case diese Aufgabe an die Klasse delegieren kann. Dafür ist die Klasse ListLeaves, die von GenericDFS ableitet. Problematisch hier ist, dass selbst trotz der Tatsache, dass der Graph im Abstraction-Code liegt und somit sich praktisch niemals fundamental ändern sollte, es sinnvoller gewesen wäre ein Interface für den Graphen zu nutzen und den eigentlichen Graphen das implementieren zu lassen. Wenn nun der Graph sich ändert, würde es mit einem Interface nicht mehr zu so vielen anderen Anpassungen kommen. ListLeaves hängt direkt von vom Graphen ab und nutzt auch direkt die Funktionalität des Graphen. Deshalb ist in ListLeaves die Kopplung vergleichsweise groß und es wäre sinnvoll die Abhängigkeit von Graph durch ein sinnvoll definiertes Interface zu ersetzen.



## Analyse GRASP: Hohe Kohäsion

*[eine Klasse als positives Beispiel hoher Kohäsion; UML Diagramm und Begründung, warum die Kohäsion hoch ist]*

Ein Beispiel für eine Klasse mit hoher Kohäsion ist der LeafValidator, der exakt eine Anwendung hat und somit sind folglich alle Methoden darauf ausgerichtet genau diesen Zweck zu erfüllen. Der LeafValidator prüft, ob für ein NamedVertex alle Blätter im Subgraphen Axiome oder Definitionen sind. Wenn das nicht der Fall sein sollte, so wird eine passende Exception geworfen. Im UML-Diagramm sieht man, dass, wie erwartet, tatsächlich alle Member und Methoden funktional zum Zweck der Klasse beisteuern. Die Member sind eine Menge an Blättern, die in der Tiefensuche gefunden wurden und ein boolean, der sicherstellen soll, dass die Tiefensuche maximal einmal ausgeführt wird.

Auch die Methoden zielen alle auf den Zweck der Prüfung der Blattknoten durch eine Tiefensuche ab.



## Don't Repeat Yourself (DRY)

*[ein Commit angeben, bei dem duplizierter Code/duplizierte Logik aufgelöst wurde; Code-Beispiele (vorher/nachher); begründen und Auswirkung beschreiben]*

Klassen, die das Interface GenericCommand implementieren, sind Commands, die in der Plugin-Schicht als „Rezepte“ für die verschiedenen Use-Cases gesehen werden können. Dabei gibt ein Command eine Liste an Parametern zurück, die von vom UI ausgefüllt werden müssen. Die Parameter validieren den Input und geben ggf. eine Exception vom Typ InvalidInputException zurück. Ein Command nimmt also die verschiedenen Inputs für die Parameter an, speichert sie in den Parametern und soll, sobald alle Parameter mit validen

Eingaben befüllt wurden, die Parameter-Inhalte geparsed in eine Map schreiben, hier als ParameterMap bezeichnet. Hierbei wird das Parsen zum Objekt durch die Parameter selbst getan. Sowohl das Eintragen der Inhalte in die Parameter als auch das Auslesen in die ParameterMap wird in allen Commands durchgeführt und wurde am Anfang, weil alle Commands ihre Parameter intern selbst als Liste speicherten (und nicht in einer Oberklasse), immer gleich in verschiedenen Klassen implementiert. Dabei gab es effektiv keine Unterschiede, sodass Code dupliziert wurde. Ich habe das durch eine gemeinsame Oberklasse gelöst, die im Konstruktor eine Parameter-Liste erhält und das Eintragen sowie das Erstellen der ParameterMap vollständig übernimmt.

Commit mit Auflösung der Duplikation: 0b9d15c9cc11efae6fb5b28932667b19bffc8842

Vorher

Vorher wurden die Parameter noch in der Klasse selbst gespeichert und somit wurden auch die Methode setInput und das Erstellen der ParameterMap noch selbst durchgeführt. Die Duplikation war genauso auch im AddTheoremCommand und käme dann in allen anderen Commands ebenso vor. Stattdessen habe ich mich entschieden das in die Oberklasse zu schieben und eine Methode fürs Erstellen der ParameterMap zu schreiben.

```

public class AddDefinedObjectCommand implements GenericCommand {
    private final List<Parameter> parameters;

    public AddDefinedObjectCommand(){
        this.parameters = Arrays.asList(
            new OptionParameter(0, "Object type", "type",
                new String[]{"Axiom", "Definition"},
                new MatType[]{MatType.AXIOM, MatType.DEFINITION}),
            new TextInputParameter(1, "name", "name"),
            new TextInputParameter(2, "description", "desc"),
            new IdListParameter(3, "sources", "sources")
        );
    }

    ...

    @Override
    public void setInput(Parameter p, String input) throws InvalidInputException {
        for (Parameter parameter : parameters){
            if (parameter.equals(p)){
                parameter.validateInput(input);
                return;
            }
        }
        throw new RuntimeException("Invalid parameter supplied!");
    }

    @Override
    public MatLearnUseCase getParametrizedUseCase() {
        HashMap<String, Object> map = new HashMap<>();
        try {
            for (final Parameter parameter : this.parameters) {
                map.put(parameter.getKey(), parameter.getParsedInput());
            }
        } catch (InvalidInputException e) {
            throw new RuntimeException(e.getMessage());
        }
        return new AddMathematicalObjectTask(new AddMatObjectParameters(map));
    }
}

```

Nachher:

<https://github.com/keksklauer4/MatLearn/commit/0b9d15c9cc11efae6fb5b28932667b19bffc8842>

```

public class AddDefinedObjectCommand extends AbstractCommand {

    public AddDefinedObjectCommand(){
        super(Arrays.asList(
            new OptionParameter(1, "Object type", ParameterMapKeys.TYPE_KEY,
                new String[]{"Axiom", "Definition"},
                new MatType[]{MatType.AXIOM, MatType.DEFINITION}),
            new TextInputParameter(2, "name", ParameterMapKeys.NAME_KEY),
            new TextInputParameter(3, "description",
ParameterMapKeys.DESCRPTION_KEY),
            new IdListParameter(4, "sources", ParameterMapKeys.SOURCES_KEY)
        ));
    }

    ...

}

```

Wie man sieht, wird hier von der Klasse AbstractCommand geerbt, die eingeführt wurde, um Code-Duplikate zu entfernen. Im Konstruktor wird eine Liste von Parametern in die Oberklasse übergeben, die dann in einer Member-Variable gespeichert wird. Dabei sind Getter und Konstruktor von AbstractCommand trivial, weshalb die hier aktiv weggelassen wurden. Methoden, die hier nicht relevant sind, wurden mit „...“ angedeutet.

```

public abstract class AbstractCommand implements GenericCommand {
    protected final List<Parameter> parameters;

    ...

    @Override
    public void setInput(final Parameter p, final String input) throws
InvalidInputException {
        for (Parameter parameter : parameters){
            if (parameter.equals(p)){
                p.validateInput(input);
            }
        }
    }

    @Override
    public Map<String, Object> getParameterMap() {
        HashMap<String, Object> parameterMap = new HashMap<>();
        try {
            for (final Parameter parameter : parameters) {
                parameterMap.put(parameter.getKey(), parameter.getParsedInput());
            }
        } catch (InvalidInputException e) {
            throw new RuntimeException(e.getMessage());
        }
        return parameterMap;
    }
}

```

Von AbstractCommand erben nun alle Commands und bei allen Commands werden beide Methoden genutzt. Dementsprechend wurde viel Code-Duplikation aufgelöst.

## Kapitel 5: Unit Tests

### 10 Unit Tests

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

Bevor das kritisiert wird: Ich habe mich aktiv für die Methodenbenennung der Tests entschieden. Ich finde es gut, wenn das Format immer test\_<aufgabe> ist, weshalb ich mich auch dafür entschieden habe.

Unit Test	Beschreibung
CycleCheckerTest#test_CycleGraph	Der Test prüft, ob ein einfacher Zyklus in einem zusammenhängenden Graphen gefunden wird.
CycleCheckerTest#test_DisconnectedPathGraphs	Der Test prüft, ob der CycleChecker auch bei nicht verbundenen Graphen noch korrekt erkennt, dass er azyklisch ist.
CycleCheckerTest#test_DisconnectedOneCycleGraph	Der Test prüft, ob der CycleChecker auch bei nicht verbundenen Graphen noch korrekt Zyklen erkennt.
GraphTest#test_AddEdgeToNonexistingVertex	Der Test prüft, ob fehlerhafte Kanten vorm Erstellen bereits abgefangen werden.
GraphTest#test_RemoveVertex	Der Test prüft, ob das Entfernen korrekt sowohl den Knoten entfernt als auch alle Kanten, die von dem Knoten ausgehen oder zu ihm hinzeigen.
NamedVertexFactoryTest#test_CreateAxiom	Der Test prüft, ob die Factory für Axiome ein valides Objekt zurückliefert.
ParameterMapParserTest#test_ValidEmptyIntegerList	Der Test prüft, ob der ParameterMapParser leere Integer-Listen korrekt behandelt, da es für leere Listen einen Spezialfall gibt.
ParameterMapParserTest#test_InvalidString	Der Test prüft, ob eine Exception bei einem invaliden String geworfen wird.
ExerciseUseCaseTest#test_ValidExerciseChosen	Der Test prüft, das Zusammenspiel zwischen dem ExerciseUseCase und dem RandomExtractor korrekt stattfindet.
RemoveEdgesUseCaseTest#test_AllValidIds	Der Test prüft, ob jedes Pärchen an ids korrekt an das Repository zum Löschen übergeben wird.



## ATRIP: Automatic

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

Testen braucht keine Voraussetzungen, muss also nicht irgendwie Vorbereitet werden (z. B. Starten einer DB etc.), benötigt keine Benutzereingaben und schlägt entweder fehl oder nicht. Es reicht `mvn test` auszuführen und die Tests laufen auch automatisch in Github CICD.

## ATRIP: Thorough

*[jeweils 1 positives und negatives Beispiel zu 'Thorough'; jeweils Code-Beispiel, Analyse und Begründung, was professionell/nicht professionell ist]*

Positiv-Beispiel:

`GraphTest#test_RemoveVertex` ist ein gutes Beispiel, da hier nicht nur geprüft wird, ob der Knoten entfernt wurde, sondern auch, ob alle Kanten bei den Nachbarn entfernt wurden – es wird also ein ausführlicher Konsistenzcheck durchgeführt. Es wird alles wichtige geprüft, was auch „professionell“ ist.

```
@Test
void test_RemoveVertex() {
    GraphBuilder builder = new GraphBuilder();
    builder.addVertices(3);
    builder.addEdges(Map.of(
        0, 1,
        1, 2
    ));
    List<GenericVertex> vertices = builder.getVertices();
    Graph<GenericVertex> graph = builder.getGraph();
    graph.removeVertex(vertices.get(1));
    Assertions.assertEquals(2, graph.getVertices().size());
    Assertions.assertTrue(graph.getVertices().containsAll(
        Arrays.asList(vertices.get(0), vertices.get(2))
    ));
    assertNoEdges(graph, vertices.get(0));
    assertNoEdges(graph, vertices.get(2));
}
```

Negativ-Beispiel:

In der Klasse `RemoveEdgesUseCase` wird zwar geprüft, ob alle Pärchen an Ids an das Repository übergeben wurden, aber durch die Nutzung eines Sets, wird nicht überprüft, wie häufig die Pärchen übergeben werden. Das ist nicht ganz vollständig und sollte eigentlich getan werden.

```

@Test
void test_AllValidIds(){
    setRemainingPairs(Arrays.asList(new Pair<>(1, 3), new Pair<>(1, 4),
        new Pair<>(2, 3), new Pair<>(2, 4)));
    UseCaseResult res = runUseCase(new RemoveEdgesParameters(Arrays.asList(1,2),
Arrays.asList(3,4)));
}

private void setRemainingPairs(List<Pair<Integer, Integer>> pairs){
    remainingPairs = new HashSet<>(pairs);
}

private UseCaseResult runUseCase(UseCaseParameter parameter){
    Mockito.when(proofNetworkRepository.removeEdgeIfExists(Mockito.anyInt(),
Mockito.anyInt()))
        .thenAnswer(invocation -> {
            Integer from = (Integer) invocation.getArguments()[0];
            Integer to = (Integer) invocation.getArguments()[1];
            remainingPairs.remove(new Pair<>(from, to));
            return true;
        });
    UseCaseResult res = useCase.execute(parameter);
    Assertions.assertTrue(remainingPairs.isEmpty());
    return res;
}

```

## ATRIP: Professional

*[jeweils 1 positives und negatives Beispiel zu 'Professional'; jeweils Code-Beispiel, Analyse und Begründung, was professionell/nicht professionell ist]*

Positiv-Beispiel:

Die UseCase-Tests haben eine gemeinsame, generische und abstrakte Oberklasse `UseCaseTest<R extends UseCaseResult>`, die einem erlaubt sowohl das Ausführen des Use-Cases mit Prüfung auf Korrektheit des Resultats vom Typ `UseCaseResult` als auch das Ausführen mit Erwarten einer Exception zu verkürzen, indem jeweils eine Methode dafür zur Verfügung gestellt wird. Das ist in der Klasse `FindVerticesUseCaseTest` von Vorteil, da hier nun nur noch gemocked werden muss.

Negativ-Beispiel:

Die Tests in der `NamedVertexFactoryTest`-Klasse sind sehr ähnlich und testen kaum unterschiedliche Dinge. Man hätte stattdessen auch einen Test schreiben können, der über alle `MatTypes` iteriert und prüft, ob alle Entitäten korrekt aufgebaut werden.

```

public class NamedVertexFactoryTest {
    ...

    @Test
    void test_CreateAxiom(){
        testCreate(MatType.AXIOM);
    }

    @Test
    void test_CreateCorollary(){
        testCreate(MatType.COROLLARY);
    }
    ...
}

```

## Code Coverage

*[Code Coverage im Projekt analysieren und begründen]*

Die Code-Coverage im Projekt ist eher gering, da Unit-Tests zu schreiben ziemlich stumpfe Arbeit ist, ich in meinem Leben bereits sehr viele geschrieben habe und dadurch da nicht allzu viel von gewinnen kann. Ich werde nun also beschreiben, was man noch sinnvoll testen sollte und was eher nachrangig ist.

Schicht	Coverage	Begründung/Anmerkung
04-abstraction	Ca. 89%	Die Abstraction-Schicht zu testen ist ziemlich sinnvoll, da praktisch alles auf dem Graphen und Tiefensuchen aufbaut. Wenn hier Fehler vorkommen, sind sie häufig sehr gravierend und problematisch. Die Coverage sollte hier also am höchsten sein und das ist sie tatsächlich auch.
03-domain	Ca. 34%	Die Domain-Schicht ist ähnlich wichtig. Die Code Coverage ist hier eher niedrig, da die Validator nicht getestet wurden. Es wäre sinnvoll, das zu tun, um wichtige Domäne-spezifischen Anforderungen zu testen. Abgesehen davon ist der Preis eines Bugs größer je tiefer er in der Architektur ist. Bugs in dieser Schicht können wie auch in der Abstraction-Schicht große Auswirkungen haben.
02-application	Ca. 28%	Eine niedrige Coverage in der Application-Schicht ist nicht besonders gut, da Fehler in den Use-Case-Implementierungen problematisch sein können. Dafür sind sie allerdings eher leicht auszubügeln im Gegensatz zu Fehlern in den tieferen Schichten.
01-adapters	Ca. 12%	Die niedrige Code-Coverage in dieser Schicht ist ebenfalls nicht gut, aber bei weitem nicht so problematisch wie in den

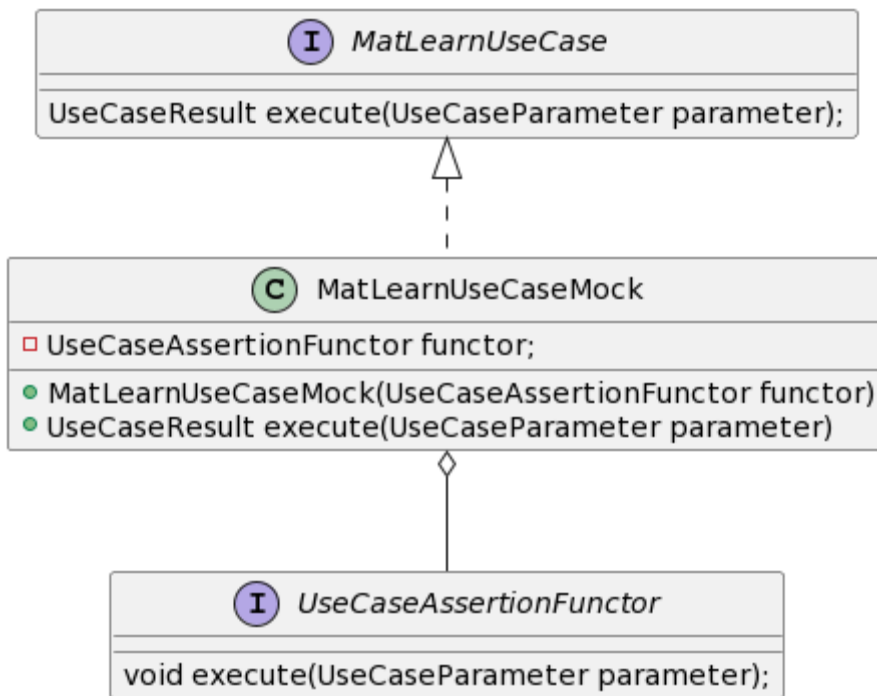
		tieferen Schichten. Es wäre sinnvoll hier besonders die Parameter zu testen, da sie Validierungslogik beinhalten. Die Adapter wurden teilweise getestet, aber sind relativ trivial, weshalb hier kaum davon auszugehen ist, dass es überhaupt Fehler in den Adaptern gibt. Trotzdem wäre auch hier eine höhere Coverage sinnvoll.
00-plugin	0%	Hier sollte man die Plugins testen. Besonders das ProofNetwork und der ProofNetworkSerializer sind sehr wichtig.

## Fakes und Mocks

*[Analyse und Begründung des Einsatzes von 2 Fake/Mock-Objekten; zusätzlich jeweils UML Diagramm der Klasse]*

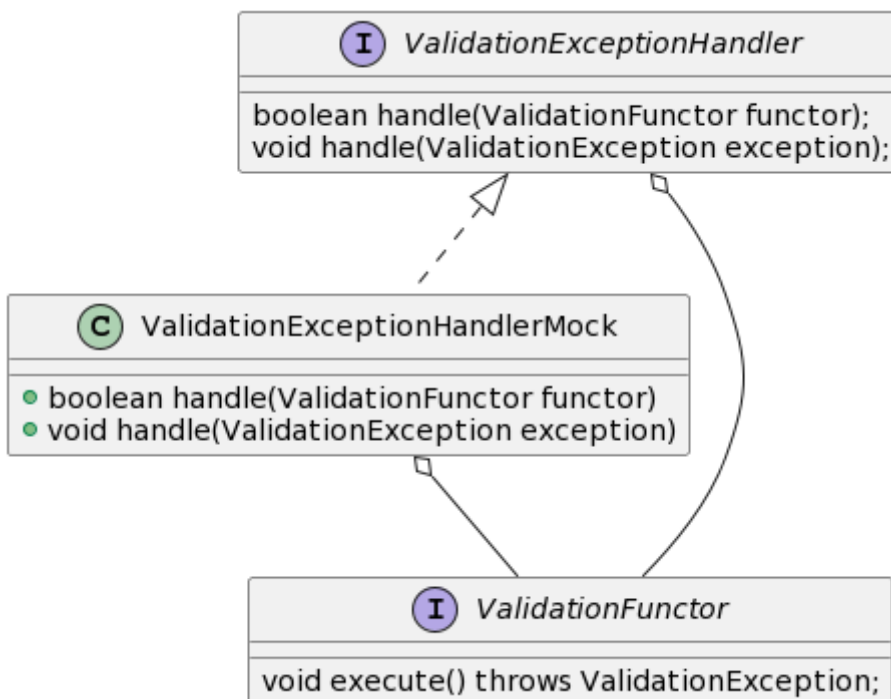
### 1. Mock-Objekt: MatLearnUseCaseMock

Das Mock-Objekt wird in den AdapterTests genutzt, da die Adapter bestimmte Use-Case-spezifische Objekte auf Grundlage der ParameterMap baut und die dann einem Objekt, das das Interface MatLearnUseCase implementiert, übergibt. Bei den Adapter-Tests ist das Ziel zu prüfen, ob das korrekte Objekt mit den korrekten Inhalten konstruiert wird. Deshalb wird ein MatLearnUseCaseMock-Objekt erstellt, das eine Funktion als Konstruktorparameter erhält und diese in der execute-Methode mit dem erhaltenen Parameter-Objekt aufruft. Dadurch kann man die Korrektheit der generierten Parameter prüfen, da im Funktor, der das Interface UseCaseAssertionFunctor implementiert, Assertions durchgeführt können.



## 2. Mock-Objekt: ValidationExceptionHandlerMock

Im `ExerciseUseCaseTest` brauche ich dieses Mock-Objekt, weil das `ProofNetworkRepository` einen Getter dafür hat und dieser in dem Use-Case `ExerciseUseCase` genutzt wird. Das Mock-Objekt führt die Funktion wie gewollt aus, aber wirft im Falle einer Exception eine `RuntimeException`, die dann wiederum von den Tests erwartet werden kann. Im normalen Betrieb würde eine solche Exception an das CLI weitergeleitet werden, aber hier ist das Ziel, dass Exceptions vom Test erwartet werden können.



## Kapitel 6: Domain Driven Design

### Ubiquitous Language

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

Bezeichnung	Bedeutung	Begründung
Source	Literaturquelle für ein mathematisches Objekt	Source hat im Allgemeinen für verschiedene Gruppen stark unterschiedliche Bedeutungen. Source kann von einem Mathematiker als „Source“ aus Flussalgorithmen verstanden werden (also z. B. bei Max-Flow), von einem Entwickler als Quellcode – hier soll es nun als Literaturquelle definiert werden.
Forward Edge/ Backward Edge	Logische Abhängigkeiten und ihre Richtung.  Forward Edge $(u, v)$ bedeutet, dass Objekt $v$ von Objekt $u$ logisch abhängt, während die Darstellung als backward edge dann $(v, u)$ ist.	In der Beschreibung der Validitätskriterien muss regelmäßig das Netzwerk geprüft werden und dabei wird zwischen Forward und Backward edges differenziert.
Lemma	Ein Hilfssatz, von dem ein Theorem logisch abhängt	Lemma ist sehr ähnlich zu einem Theorem – nur dass es semantisch einen kleinen Unterschied gibt: Ein Lemma ist ein Hilfssatz, der bewiesen wird, um ein Theorem zu beweisen. In einer vollständigen Validierung muss also geprüft werden, ob es tatsächlich ein Theorem gibt, das zumindest indirekt auf dem Lemma aufbaut. Für dieses Kriterium muss allen bewusst sein, wo der Unterschied ist.
Axiom	Eine als wahr angenommene Aussage	Ein Axiom hat in der Domäne eine besondere Anforderung, die allen klar sein muss. Konkret darf ein Axiom niemals von anderen mathematischen Objekten logisch abhängen – es gibt also keine backward edges vom Axiom ausgehend. Das heißt es muss regelmäßig validiert werden. Insbesondere muss der

		Unterschied zur mathematischen Definition verstanden werden: Während ein Axiom nicht von anderen Axiomen abhängen darf, darf eine Definition, wie etwa ein metrischer Raum, durchaus genau das.
--	--	---

## Entities

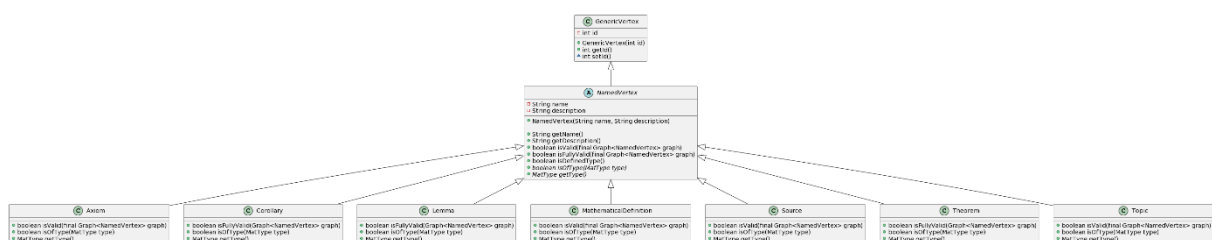
*[UML, Beschreibung und Begründung des Einsatzes einer Entity; falls keine Entity vorhanden: ausführliche Begründung, warum es keines geben kann/hier nicht sinnvoll ist]*

Die Entities in meinem Modell sind mathematische Objekte in einem Graph, die jeweils voneinander abhängen durch forward und backward edges. Die Entities erben alle von GenericVertex, was intern eine id besitzt um sinnvoll Abhängigkeiten zwischen Entities durch Kanten zu ermöglichen. GenericVertex ist in der Abstraction-Schicht definiert und wird für den Graphen und die EdgeList als Oberklasse der eigentlich im Graphen zu speichernden Entitäten genutzt.

Die Entitäten sind alle vom Typ NamedVertex bzw. haben diesen als Oberklasse. NamedVertex ist eine einfache Abstraktion für alle Entitäten, da alle mathematischen Objekte einen Namen und eine Beschreibung haben sollen. Außerdem gibt es mehrere Anforderungen an Funktionalität, die von den ererbenden Klassen überschrieben werden soll. Darunter sind Validitätsprüfungen und Typabfragen, die vor allem Validators brauchen, um Konsistenz und Korrektheit zu gewähren.

In NamedVertex wird zwischen isFullyValid und isValid differenziert. Letzteres ist eine stark relaxierte Validitätsprüfung, bei der nicht alle Domain-Regeln geprüft werden, da beim Aufbau des Netzwerkes zwischenzeitig invalide Zustände erlaubt werden müssen. IsFullyValid ist dann eine Prüfung aller Domain-Regeln – Entity-spezifisch.

Die eigentlichen Entitäten erben alle von NamedVertex und sind Axiom, Corollary, Lemma, MathematicalDefinition, Source, Theorem und Topic. Sie unterscheiden sich in den Validitätskriterien und in Hinblick auf ihren semantischen Typen.

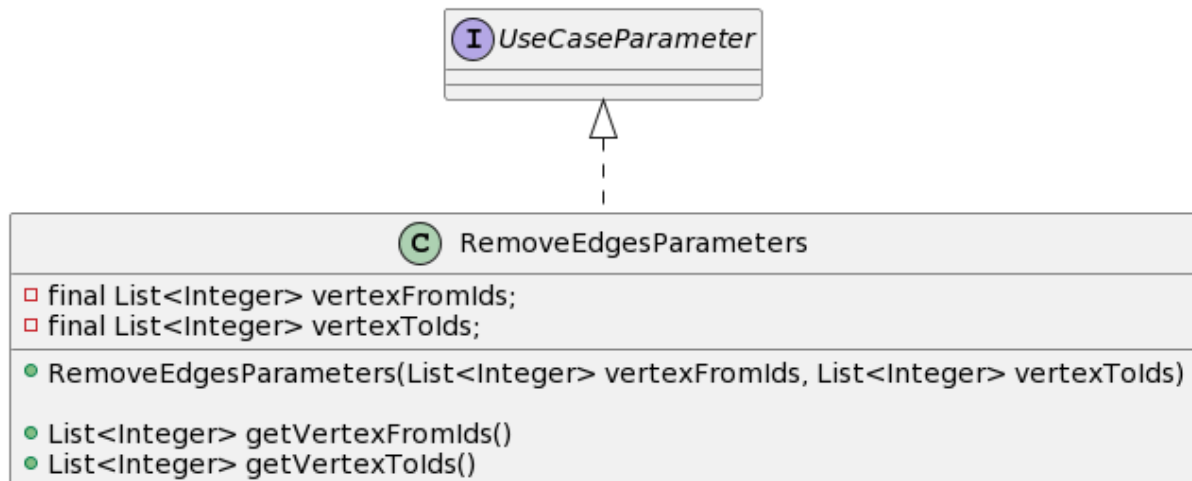


## Value Objects

*[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]*

Die Objekte, die das Interface UseCaseParameter implementieren (Application-Schicht), sind ValueObjects, da sie weder eine Identität haben – es handelt sich um reine data classes – und nicht verändert werden. Ein Beispiel, welches auch im UML-Diagramm visualisiert wird, ist die Klasse „RemoveEdgesParameters“. Im Konstruktor werden zwei id-Listen übergeben, die als

Member gespeichert werden und dann über Getter abgefragt werden können. Offensichtlich gibt es hier keine Identität und Setter gibt ebenfalls nicht. Ich muss zugeben, dass diese Eigenschaften zwar zutreffen, aber ein Value Object in meinem Verständnis mehr als das ist. Die typischen Beispiele waren ja Preise, Gewichtsangaben etc. – also Objekte, die Werte mit unterschiedlichen Einheiten kapseln und besser vergleichbar machen. Trotzdem denke ich, dass diese Eigenschaften reichen, sodass man hier von einem Value Object sprechen kann.

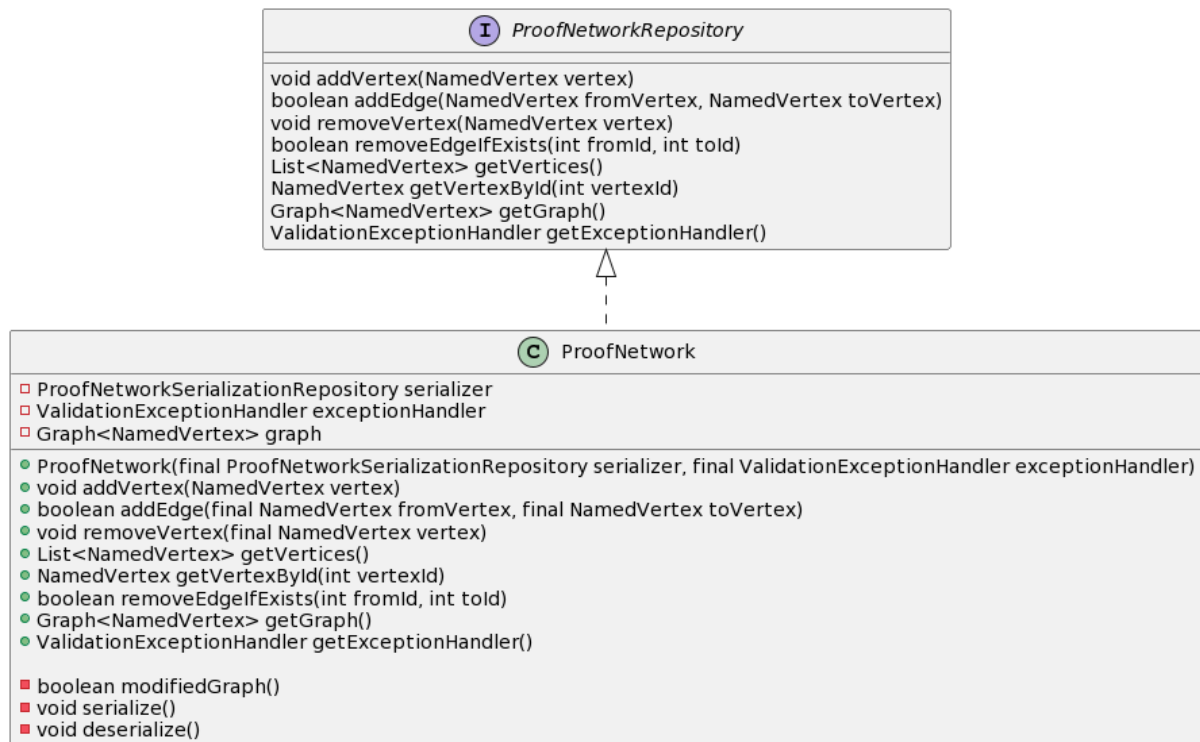


## Repositories

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

Das **ProofNetworkRepository** abstrahiert ein **ProofNetwork** – also ein Graph, in dem man mathematische Objekte speichern und logische Abhängigkeiten zwischen ihnen aufbauen kann. Außerdem können Objekte abgefragt werden, entfernt werden und der Graph kann ausgegeben werden. Das **ProofNetworkRepository** erfüllt also die klassischen Aufgaben eines Repositories – gleichwohl es keine Aggregate speichert/verwaltet. Dementsprechend ergab es Sinn das **ProofNetworkRepository** auch als solches zu bezeichnen.

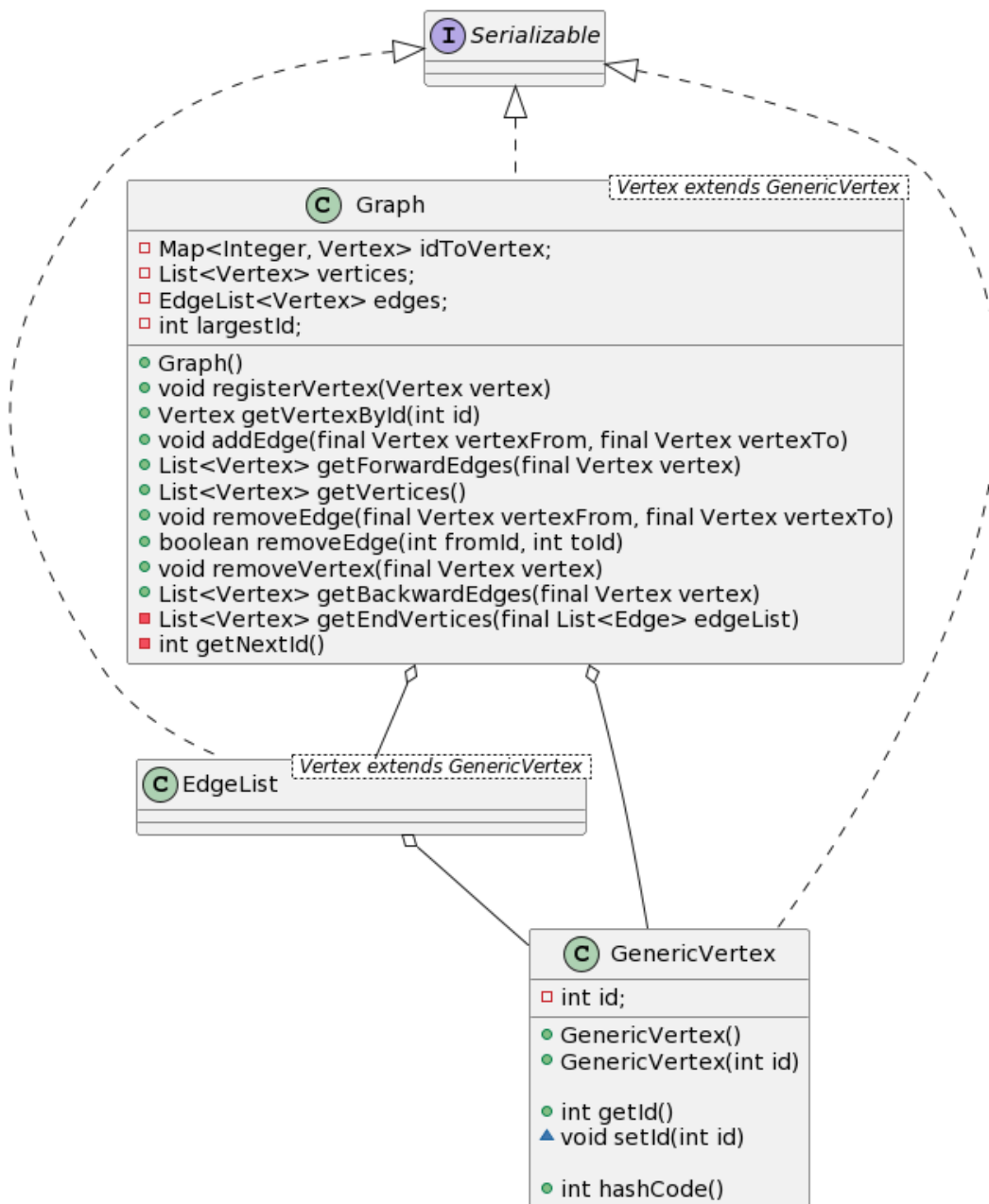




## Aggregates

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

Lars und ich haben über die Frage gesprochen, ob der Graph selbst ein Aggregate ist. Wir sind zum Schluss gekommen, dass der Graph tatsächlich das wäre, was einem Aggregate am nächsten käme. Das liegt daran, dass hier alle Entitäten drin aggregiert sind und es ein Repository, das ProofNetworkRepository, gibt, das den Umgang mit dem Graphen verwaltet. Außerdem wird der Graph immer als ganzes geladen und gespeichert. Besonders die Selbstbezüglichkeit macht es schwierig hier mit Überzeugung von einem Aggregate zu sprechen, doch kann ein Aggregate auch nur aus einer Entität bestehen – hier also die Oberklasse NamedVertex als von GenericVertex abgeleitete Klasse, von der alle anderen Entitäten abhängen. Offensichtlich wäre NamedVertex dann auch die Aggregate Root Entity.



## Kapitel 7: Refactoring

### Code Smells

*[jeweils 1 Code-Beispiel zu 2 Code Smells aus der Vorlesung; jeweils Code-Beispiel und einen möglichen Lösungsweg bzw. den genommen Lösungsweg beschreiben (inkl. (Pseudo-)Code)]*

#### 1. Code Smell: Long Method

```
@Override
public UseCaseResult execute() {
    List<Integer> fromIds = parameters.getVertexFromIds();
    List<Integer> toIds = parameters.getVertexToIds();
    if (fromIds == null || toIds == null){
        return null;
    }
    for (Integer fromId : fromIds){
        if (fromId == null) {
            continue;
        }
        for (Integer toId : toIds){
            if (toId == null){
                continue;
            }
            networkRepository.removeEdgeIfExists(fromId, toId);
        }
    }
    return new ValidCommandResult();
}
```

Die Methode ist zu lang und relativ unschön, weil es so viele returns gibt und mehrere verschachtelte Schleifen. Das kann man generischer machen, wodurch das deutlich verständlicher wird. Das Code-Beispiel ist zwar nun executeTyped (ich wollte die Services stateless machen und die Parameter besser überreichen/generieren), das macht hier allerdings keinen Unterschied. Unabhängig davon wird nun ein generischer PairGenerator mit zwei Listen erstellt und jedes valide generierte Pärchen wird an die Methode an die Methode removeEdgeIfExists des networkRepositories gegeben (die dazugehörigen Kanten werden also gelöscht).

```
@Override
protected UseCaseResult executeTyped(RemoveEdgesParameters parameters) {
    PairGenerator<Integer> pairGenerator = new PairGenerator<>(
        parameters.getVertexFromIds(),
        parameters.getVertexToIds());
    pairGenerator.generate(networkRepository::removeEdgeIfExists);

    return new ValidCommandResult();
}
```

Das generate des PairGenerator sieht wie folgt aus und iteriert über beide Listen und übergibt das Pärchen an den Funktor, falls es valide ist. Der Konstruktor nimmt einfach die beiden Listen und speichert sie, weshalb ich den hier nicht eingefügt habe.

```
public void generate(PairFunction<T> functor){
    if (listA == null || listB == null) return;
    for (final T a : listA){
        for (final T b : listB){
            if (a != null && b != null){
                functor.execute(a, b);
            }
        }
    }
}
```

Dabei ist PairFunction<T> ein generisches Interface mit der Methode  
void execute(T valueA, T valueB).

## 2. Code Smell: Switch-Statement

Vorher:

Ein großes Switch-Statement hat einem die Laune getrübt und hat den Code schlechter gemacht, indem das Statement theoretisch immer weiter wachsen könnte (wenn man neue Entitäten hinzufügen würde). Das Switch-Statement macht den Code auch weniger wartbar und fördert Fehler. Abgesehen davon ist es in diesem Fall auch Code-Duplikation.

```

private void assertCorrectType(NamedVertex vertex, MatType type) {
    Assertions.assertEquals(type, vertex.getType());
    switch (type){
        case AXIOM:
            Assertions.assertEquals(Axiom.class, vertex.getClass());
            break;
        case COROLLARY:
            Assertions.assertEquals(Corollary.class, vertex.getClass());
            break;
        case DEFINITION:
            Assertions.assertEquals(MathematicalDefinition.class,
vertex.getClass());
            break;
        case LEMMA:
            Assertions.assertEquals(Lemma.class, vertex.getClass());
            break;
        case SOURCE:
            Assertions.assertEquals(Source.class, vertex.getClass());
            break;
        case THEOREM:
            Assertions.assertEquals(Theorem.class, vertex.getClass());
            break;
        case TOPIC:
            Assertions.assertEquals(Topic.class, vertex.getClass());
            break;
        default:
            throw new RuntimeException("Invalid type.");
    }
}

```

Nachher:

Das Switch-Statement wurde durch eine Map getauscht. Es wird nun geprüft, ob der Typ in der Map ist und dann ob er mit dem Datentyp des „vertex“ übereinstimmt.

```

public class NamedVertexFactoryTest {

    private static final Map<MatType, Class<? extends NamedVertex>> vertexTypeMap
= Stream.of(
        new AbstractMap.SimpleEntry<>(MatType.AXIOM, Axiom.class),
        new AbstractMap.SimpleEntry<>(MatType.COROLLARY, Corollary.class),
        new AbstractMap.SimpleEntry<>(MatType.DEFINITION,
MathematicalDefinition.class),
        new AbstractMap.SimpleEntry<>(MatType.LEMMA, Lemma.class),
        new AbstractMap.SimpleEntry<>(MatType.SOURCE, Source.class),
        new AbstractMap.SimpleEntry<>(MatType.THEOREM, Theorem.class),
        new AbstractMap.SimpleEntry<>(MatType.TOPIC, Topic.class)
    ).collect(Collectors.toMap(Map.Entry::getKey, Map.Entry::getValue));

    ...

    private void assertCorrectType(NamedVertex vertex, MatType type) {
        Assertions.assertEquals(type, vertex.getType());
        Assertions.assertTrue(vertexTypeMap.containsKey(vertex.getType()));
        Assertions.assertEquals(vertexTypeMap.get(type), vertex.getClass());
    }

    ...

}

```

## 2 Refactorings

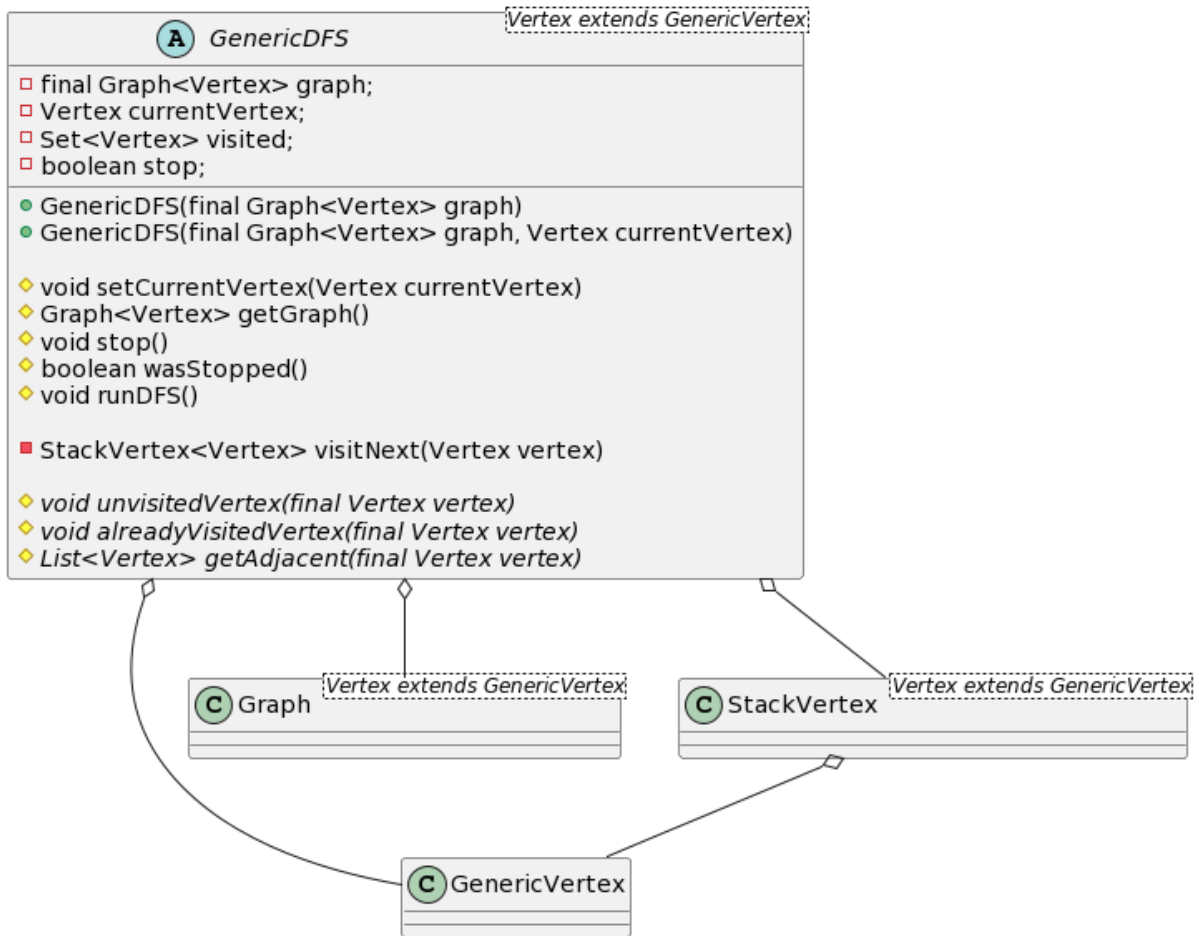
*[2 unterschiedliche Refactorings aus der Vorlesung anwenden, begründen, sowie UML vorher/nachher liefern; jeweils auf die Commits verweisen]*

### 1. Refactoring: Extract Method

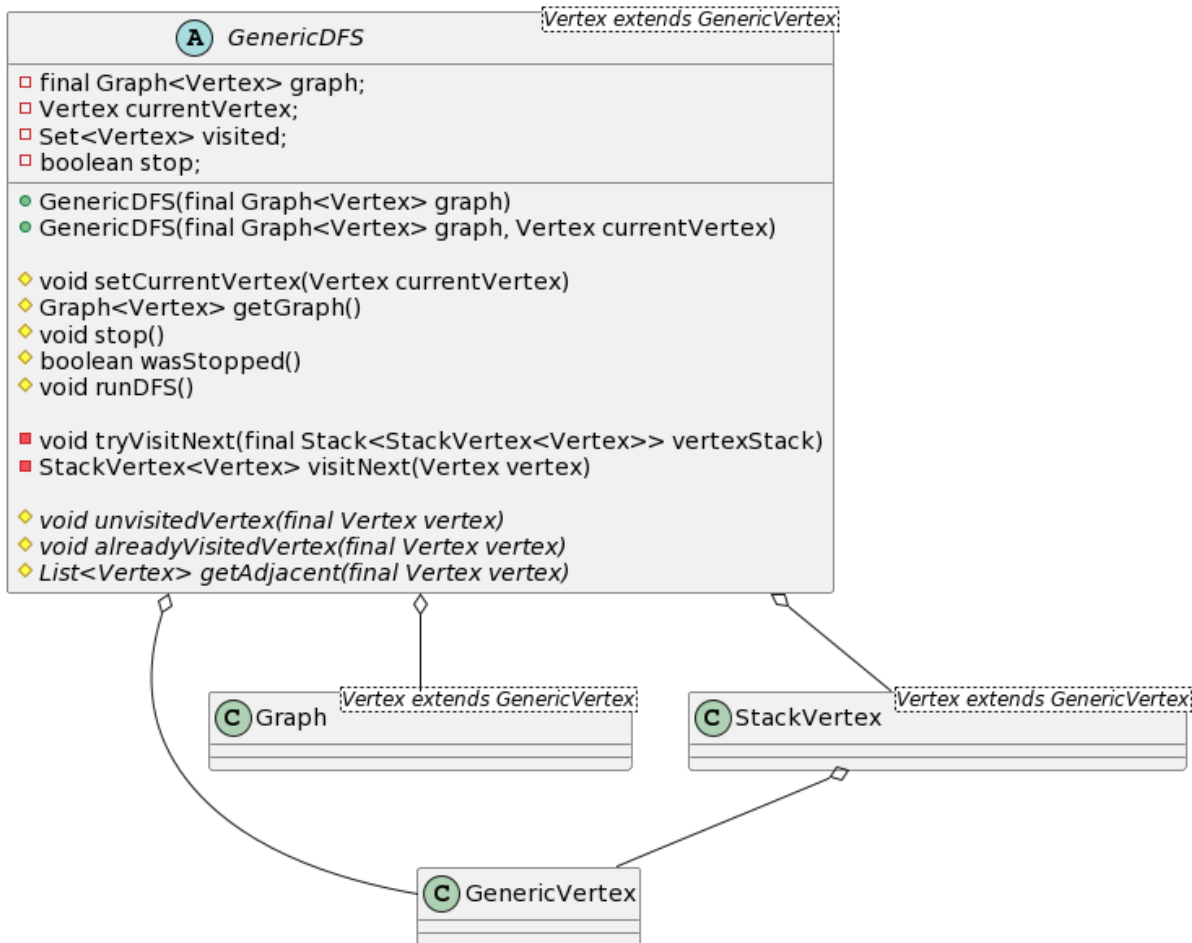
In GenericDFS ist der Kern der Tiefensuche relativ groß und tief verschachtelt gewesen, weshalb es sich hier, um Bugs und schlechte Wartbarkeit zu verhindern, gelohnt hat eine extra Methode für das Prüfen und ggf. Besuchen des nächsten Knoten zu schreiben. Es kam nun also die Methode tryVisitNext hinzu, die einfach einen Teil der eigentlichen Methode runDFS auslagert.

Das Refactoring fand in commit 872530798986db2228f5fc4cb2e1b714ef108422 statt.

Vorher:



Nach dem Refactoring:

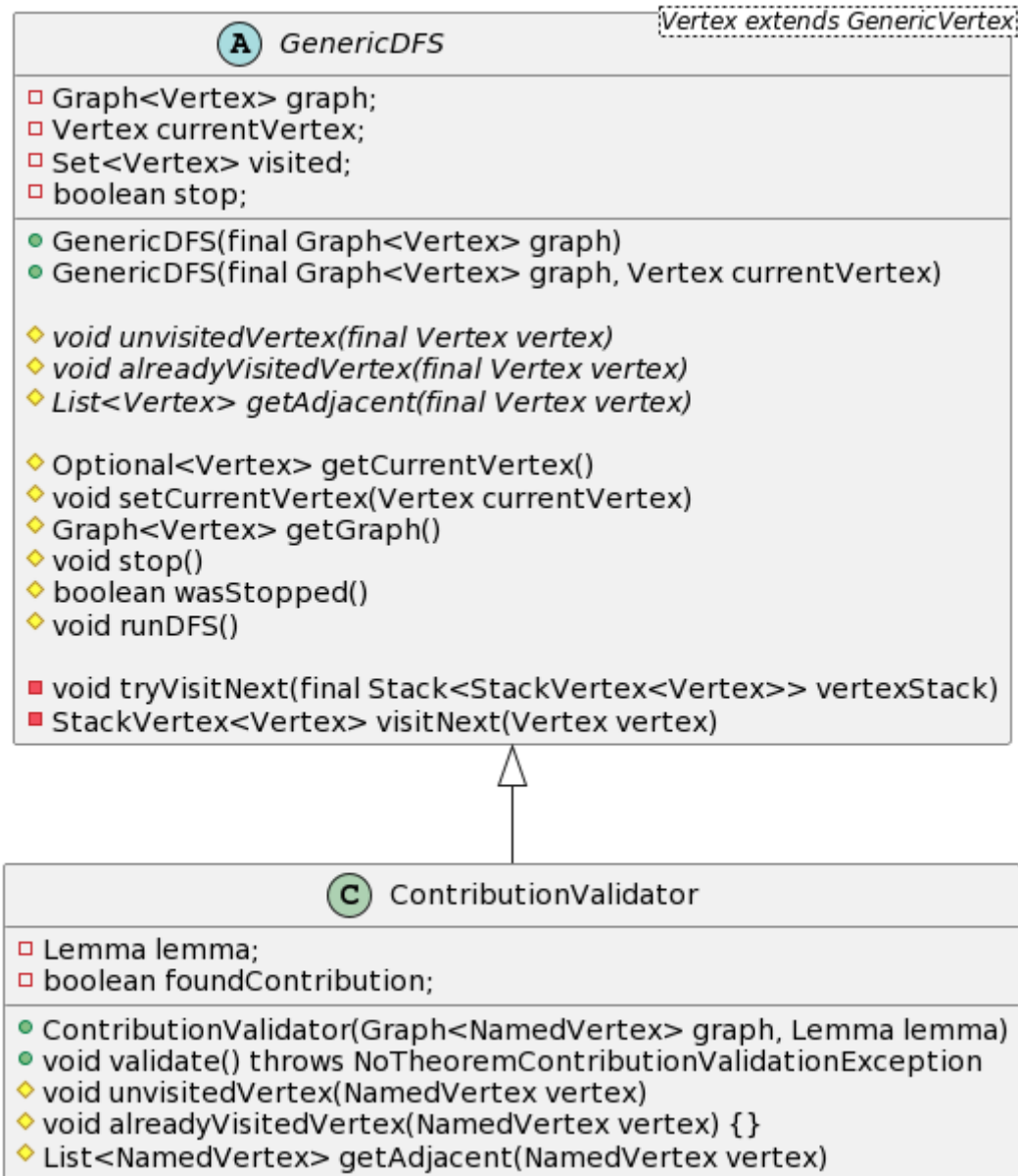


## 2. Refactoring: Rename Method

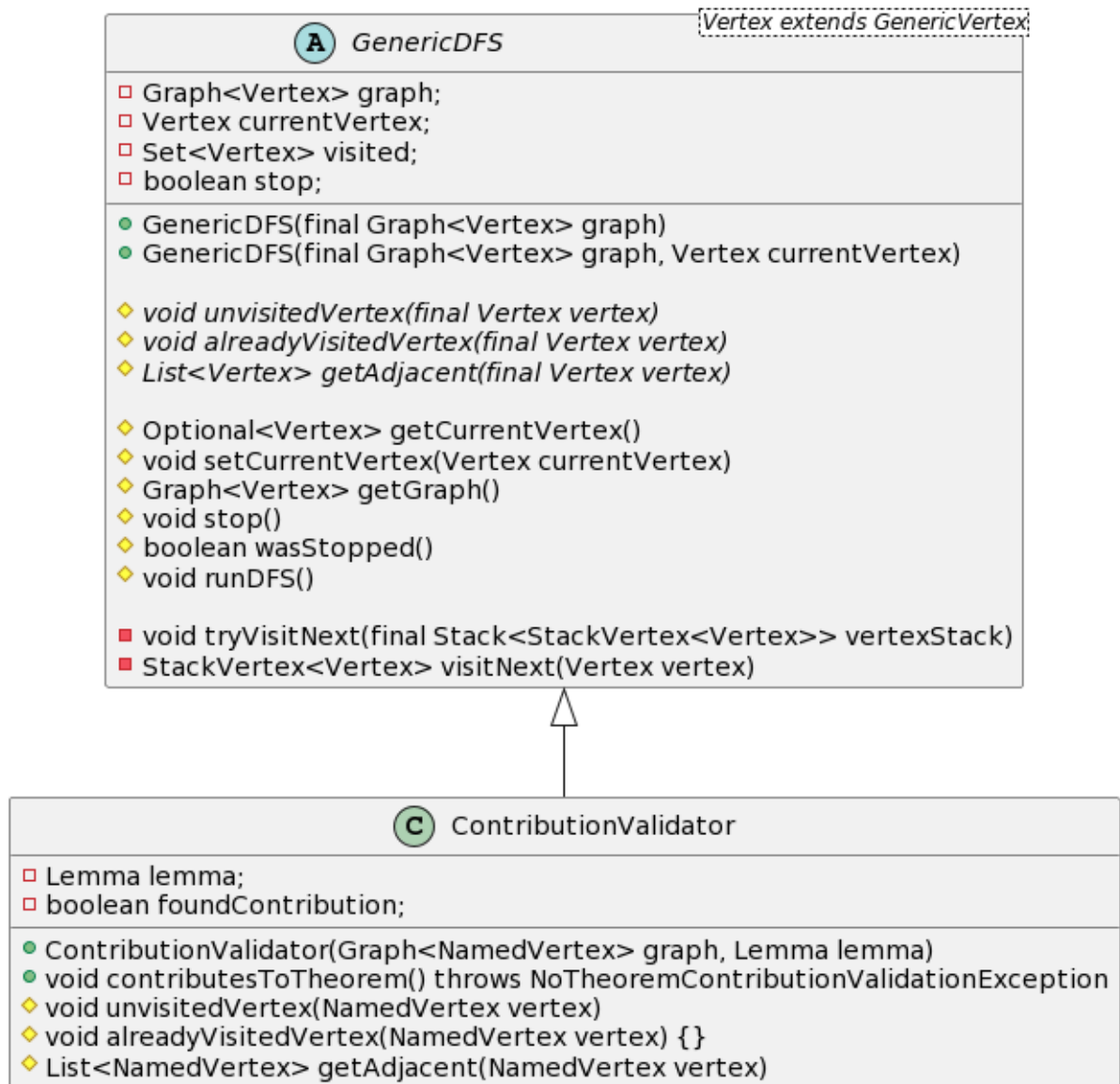
Ich habe, wie bereits erläutert, für verschiedene Entitäten verschiedene Validitätskriterien – darunter auch für Lemmata. Ein Kriterium, das nur bei einer vollständigen Validierung geprüft wird, ist das Folgende: Ein Lemma ist ein Hilfssatz, der eigentlich dafür gedacht ist, ein Beweis eines Theorems zu unterstützen. Wie auch in der Ubiquitous Language definiert, muss ein Lemma also indirekt zu mindestens einem Theorem beitragen. Um das zu Überprüfen wird eine Tiefensuche im ContributionValidator durchgeführt. Damit der Code besser lesbar wird, habe ich die Methode validate() zu contributesToTheorem() umbenannt. Selbst ich hatte zwischendurch vergessen, was der ContributionValidator tut, als ich ihn im Code vom Lemma gesehen habe. Mit dem neuen Methodennamen sollte es deutlich verständlicher sein (, wenn man die Ubiquitous Language kennt).

Vorher:





Nachher:



Siehe Commit 54b9661aefecbcd35e495d4b0dccbbee80329b07

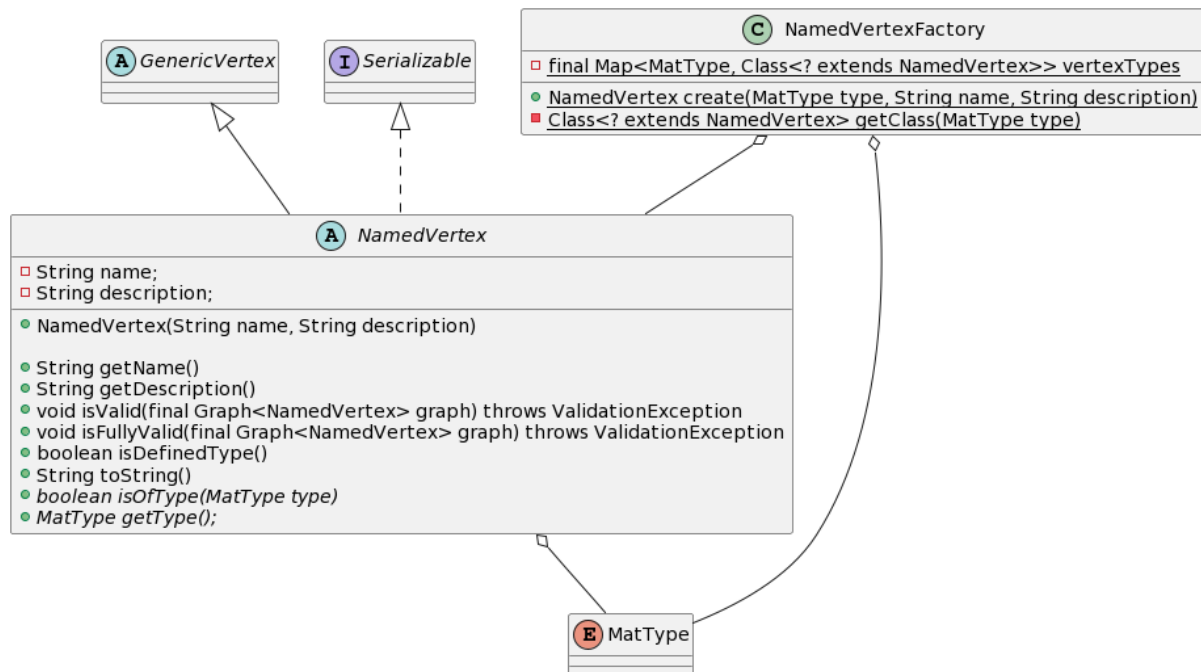
## Kapitel 8: Entwurfsmuster

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

### Entwurfsmuster: Factory

Fürs Konstruieren eines mathematischen Objektes hat man drei Parameter: Der Name des Objektes als String, eine Beschreibung als String und der Typ als ein Enum-Wert vom Datentyp MatType. In Abhängigkeit des Typs muss nun ein von NamedVertex erbenendes Objekt konstruiert werden. Alle diese Entitäten haben einen Konstruktor mit Name und Beschreibung als Parameter, sodass man nun die richtige Klasse finden und die beiden Parameter an den Konstruktor übergeben muss. Das kann man mit einem Switch-Case-Statement machen oder aber über Reflections in mit einer Map. Letzteres habe ich gemacht.

Ich halte es für sinnvoll hier eine Factory zu nutzen, da das Auswählen der korrekten Klasse unnötige Komplexität an anderer Stelle einführen und dementsprechend das Verständnis für andere Entwickler erschweren würde. Durch die Factory versteckt man die Komplexität der Reflections.



## Entwurfsmuster: Adapter

Ich habe das Adapter-Entwurfsmuster für das Übersetzen der ParameterMap aus Schicht 1 in Parameter-Objekte, die das Interface `UseCaseParameter`, genutzt. Es werden also die Sicht der Adapter-Schicht in die Sicht der Applikationsschicht übersetzt. Ein Beispiel hier ist der `AddConnectionAdapter`, der das Interface `UseCaseParameterAdapter` implementiert. Das Interface sieht vor, dass ein das Interface implementierendes Objekt einen Use-Case vom Typ `MatLearnUseCase` und einen Parser für die ParameterMap erhält und dann mit dem Parser ein passendes Objekt von der jeweilig für den Use-Case passenden Parameter-Klasse erstellt. Hier wird direkt das Objekt an die Execute-Methode gegeben und der Use-Case ausgeführt. Das Resultat vom Typ `UseCaseResult` wird zurückgegeben.

Wichtig ist, dass für jede Parameter-Klasse ein eigener Adapter implementiert wird, sodass diese dann sich um das Umwandeln der Inhalte der ParameterMap kümmert. Das Auswählen des korrekten Adapters wird in `UseCaseParameterDispatcher` durchgeführt. Das ermöglicht ein problemloses Erweitern der Use-Cases.

