Programmentwurf
DevTool

Name: Christ, Luca
Martrikelnummer: 3546285
Name: Volk, Michael
Martrikelnummer: 9849811

Abgabedatum: 31.05.2022

Kapitel 0: Erklärung der Commits

Hallo Herr Müller, wir hatten Probleme das Projekt über zwei PC’s zu versionieren, weil die Applikation auf dem zweiten PC nicht lief. Da wir derzeit in einer WG leben haben wir uns dazu entschieden, die Implementierung an einem Laptop vorzunehmen. Folgend erhalten Sie einen Überblick, welcher Commit von wem ist.

	Luca	Michael
Commits	d8fa39171a3c106e8d3afcf24e4c034b9631a48a aca8f8b557477313ec1ea4edd916f001e42f4139 a37ca3ca35c46b904c7b74fc8a57527ea70e37ee 0d5556d8e0aeb4d835cc5bbc308a683ff06ecb56 92b34d50c7cd013b1eee0903c58bf186d3d5cb32 d1fc1c2a21cbf85353d8a001af6336159a1c5326 786d4e4e12e01edee743840916ddd117e5e971a8 d311a78a08cb6473530eda5e416a867a88b8402b 7fc5e2758aba3e804238c8411013282cfa97846d 27c78e676fadfccf22c9b6b73f29e0382cfda63d 5a4052b65112b0ca7b53d136b42c6f54771b1d8c 23c2001f1a82cf7c3b8ab62b9a593a1b8cfbae78 f34273cd4f1fcc14b342b72e006bc27df0d93b70 f74fe03b57042ceb46f99a04597abe17e566ff5d eadb79ecf00593bcf380a9345dae81c49993a80e fcb2d961d99280daeadf7dc86e019a78b2e083cd c5b975a3ef4b5c49608ac8c18c17ce9e13518955 9b828f544bbaf109c4d1a331b12b455c53f0eac8	0ff5c30157dfbb3b34828cb0e37bd586a826d251 6a6248569372a1ec9cca6c65112449a396c5ed34 23048417f7cfadaddbfcc1aab5f63058c158a393 555138f19244a8d3a136016583da595198e3c609 c9c2eed62353b3ee407a77c300f035efbd60cdbb d9e6837934277c3ae49f8fe047b2dbfee31e0faf 62491f9e4d942b6d5c248b0a6b4356538c65044c d622d09b83d1d37e3a93cc3635e47835f28a5926 8cb4811028e6cbca40d3112f576a20d469561b35 737af6040ebca4441fe1991609bc931cc2153b0e 9fe6190abc865d7c2212e03acfcda5ef36e829e6 625d65fa4d1e34d5bb3da12149aeeb05731f1d6b 13f7762a8fe201f85812e2463778d5e0dfc38a7a fad1b71aa4b20b7e4db172404789a0b342f327f0 33cadbfd62f0078714f2951122795e349e41b482 4726ea102dd695bc481628e1a7f30d661e86d377 78984ab9584de812741d5e1f840df51f0edb75a2 047f295fa25bcb11bf22386114217a690699df3e

Kapitel 1: Einführung (4P)

Übersicht über die Applikation (1P)

Die Applikation DevTool besitzt drei unterschiedliche Werkzeuge. Einen *Calculator*, welcher verschieden rechenoperationen wie z.B. Addition und Subtraktion ausführt oder auch trigonometrische und Logarithmische Berechnungen. Weiter verfügt die Applikation über einen *Descriptor*, welcher Definitionen der verschiedenen Kalkulationen enthält. Außerdem über einen *Numberconverter*, der Ganzzahlen im Dezimalsystem entgegennimmt und in ein anderes Zahlensystem umwandelt. Außerdem enthält die Applikation eine Pseudo-Implementierung einer Datenbank und eines *Drawers*. Die Datenbank soll künftig dem Nutzer das Speichern von Rechenergebnissen ermöglichen und der *Drawer* das Zeichnen von geometrischen Formen.

Beim Ausführen des DevTools wird der Nutzer gefragt, ob er was berechnen, sich eine Rechnung erklären lassen oder eine Zahl konvertieren möchte. Je nach Auswahl wird der nutzer anschließend gefragt was er berechnen/erklärt haben möchte oder in welches Zahlensystem er eine Zahl konvertieren möchte.

Somit ermöglicht Die Applikation DevTool dem Nutzer verschiedenste rechenoperationen auszuführen, eine Erklärung zu den rechenoperationen zu erhalten, sowie Zahlen vom Dezimalsystem in andere Zahlensysteme zu konvertieren.

Wie startet man die Applikation? (1P)

Für das Starten der Applikation muss Docker auf dem PC installiert sein. Danach kann das Docker-Image mit dem Befehl: docker pull mvolk336/devtool heruntergeladen und mit dem Befehl: docker run mvolk336/devtool die Applikation gestartet werden.

Schritt-für-Schritt-Anleitung

1. Docker installieren
2. ***docker pull mvolk336/devtool*** ausführen
3. ***docker run mvolk336/devtool*** ausführen

Technischer Überblick (2P)

C# als Programmiersprache

C# ist eine Objektorientierte Programmiersprache(OOP). OOP fasst Daten in Objekten zusammen, was es einfacher macht, die Anwendung in kleinere Teile zu zerlegen, die schneller zu erstellen, zu verwalten und zu kombinieren sind. Des weiteren gilt C# als Hochsprache, da ihre Syntax der menschlichen Sprache ähnelt. Hochsprachen sind für Entwickler vorteilhaft, weil sie im Gegensatz zu Low-Level-Sprachen wie C eine einfachere Syntax haben.

Docker zur Auslieferung

Mit der Hilfe von Docker lassen sich alle Abhängigkeiten einer Applikation in einem Docker-Image speichern. Aus diesem Image kann eine Instanz erzeugt werden, ein sogenannter Docker-Container. Ein Container ermöglicht Prozesse abzusondern, sodass diese unabhängig voneinander ausgeführt werden können. Auf diese Weise wird die Infrastruktur besser genutzt, während gleichzeitig die Sicherheit durch das Arbeiten mit getrennten Systemen steigt. Soll eine weitere Person zugriff auf die Anwendung haben, kann dieser das dazugehörige Docker-Image bereitgestellt werden. Dieses Image kann dann lokal ausgeführt werden.

xUnit zum Testen

xUnit ist ein Unit-Testing-Tool, welches Communityfokussiert und Open-Source agiert. Des Weiteren gilt xUnit als die neueste Technologie für Unit-Tests in C#.. Außerdem ist es Teil der .Net Foundation und arbeitet unter dem diesbezüglichen Verhaltenskodex.

moq zum erzeugen von Mock-Objekten

Moq bietet einen klar verständlichen Syntax und verfügt über ein Standardverhalten, welches praxisorientiert ist.

Kapitel 2: Clean Architecture (8P)

Was ist Clean Architecture? (1P)

Clean Architecture

Clean Architecture ist ein Architekturstil mit der zentralen Eigenschaft, dass die Geschäftslogik nicht von anderen Schichten oder Infrastrukturen beeinflusst werden kann.

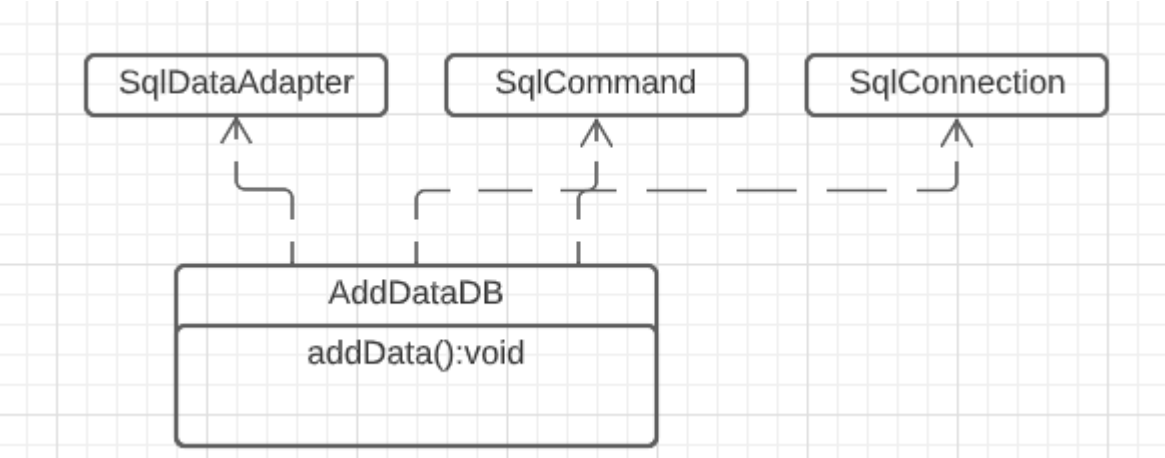
Der Vorteil dieser Architektur ist, dass Komponenten der Infrastruktur zu jederzeit verändert werden können, da die Geschäftslogik unabhängig von jeglichen Frameworks, Nutzerinterfaces, Datenbanken oder anderen äußeren Einflüssen ist.

Durch diese Unabhängigkeit ist auch das Testen und Weiterentwickeln deutlich unkomplizierter und schneller.

Analyse der Dependency Rule (2P)

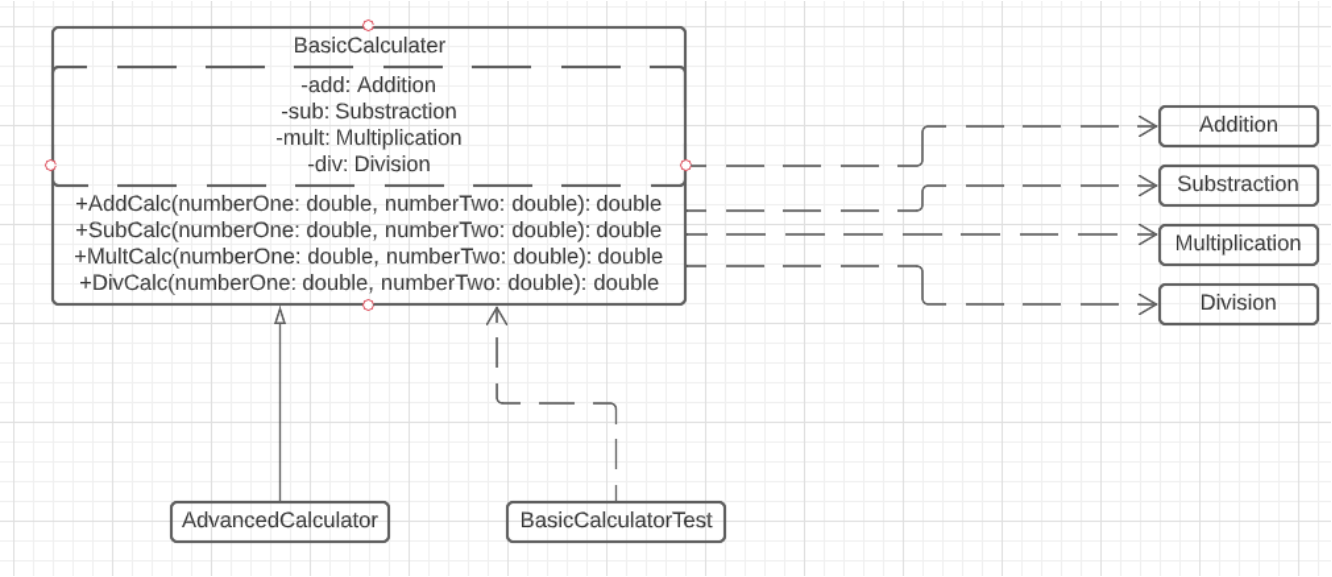
Positiv-Beispiel: Dependency Rule

Ein Positiv-Beispiel ist die Klasse AddDataDB.cs. Sie ist abhängig von drei Klassen aus einer Libary: SqlCommand, SqlConnection, SqlDataAdapter. Keine Klasse ist abhängig von ihr.



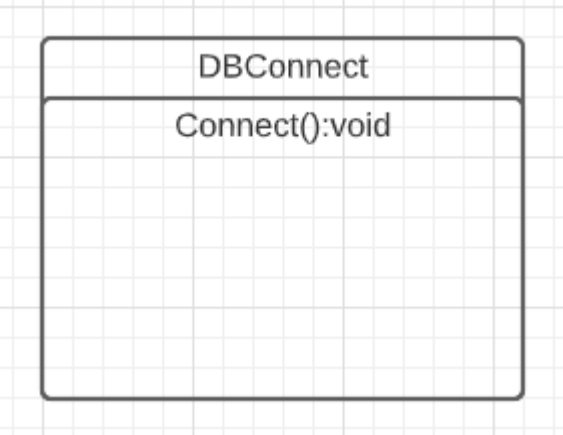
Negativ-Beispiel: Dependency Rule

Die Klasse BasicCalculator.cs ist ein Negativ-Beispiel, da von ihr die Klasse BasicCalculatorTest.cs abhängt und die Klasse AdvancedCalculator.cs erbt von ihr und ist damit auch abhängig von der BasicCalculator.cs Klasse.
Von den Klassen Addition.cs, Substraction.cs, Multiplication.cs und Division.cs hängt wiederum die BasicCalculator.cs Klasse ab.



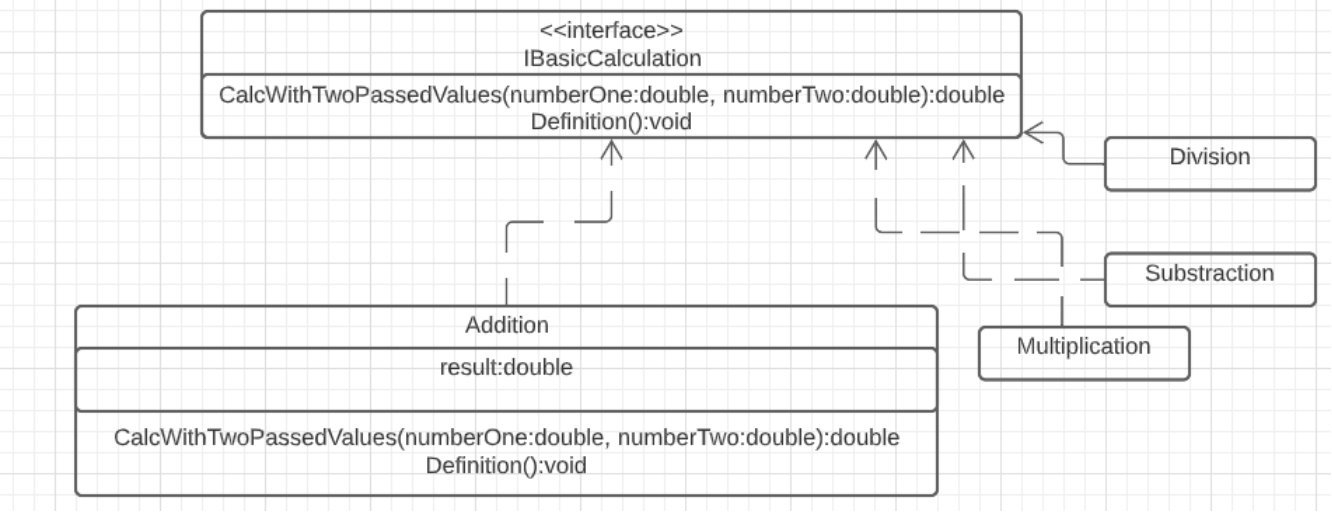
Analyse der Schichten (5P)

Schicht: [Frameworks&Treiber]



Die Klasse DBConnect.cs besitzt eine Funktion Connect(). Diese stellt eine Verbindung zu einem SQL Datenbankserver her. Datenbanken gehören zur äußersten Schicht der Clean-Architecture.

Schicht: [Interface Adapters]



Die Schnittstelle IBasicCalculation stellt zwei Funktionen zur Verfügung: CalcWithTwoPassedValues(double numberOne, double numberTwo) und Definition().

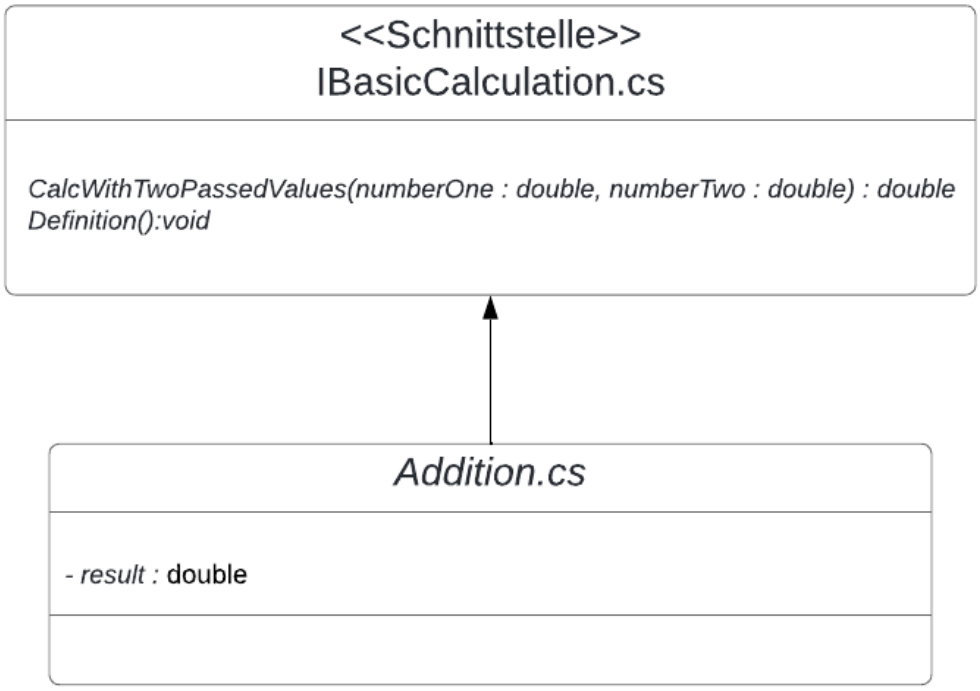
Die Funktionen werden in dem Interface nur deklariert und haben noch keine Funktion. Die zwei Funktionen werden in den Klassen, die das Interface implementieren, initialisiert. Das Interface gehört zu der Interface Adapters Schicht.

Kapitel 3: SOLID (8P)

Analyse SRP (3P)

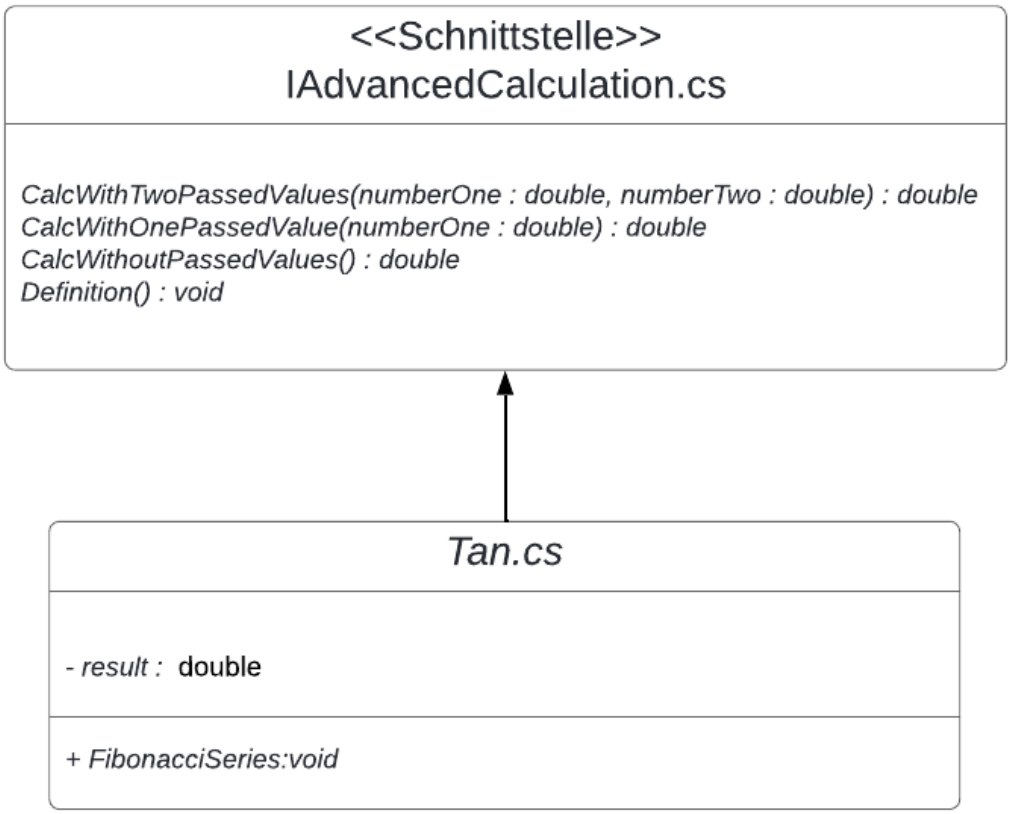
Positiv-Beispiel

Die Klasse *Addition.cs* ist ein positives Beispiel für SRP, da die Klasse rein für das Durchführen und beschreiben einer Addition Verantwortlich ist. Wobei die Methode *CalcWithTwoPassedValues()* eine Addition mit zwei übergebenen Werten durchführt und das Ergebnis zurückliefert. Und die *Definition()*-Methode eine Definition von Addition in der Konsole ausgibt.

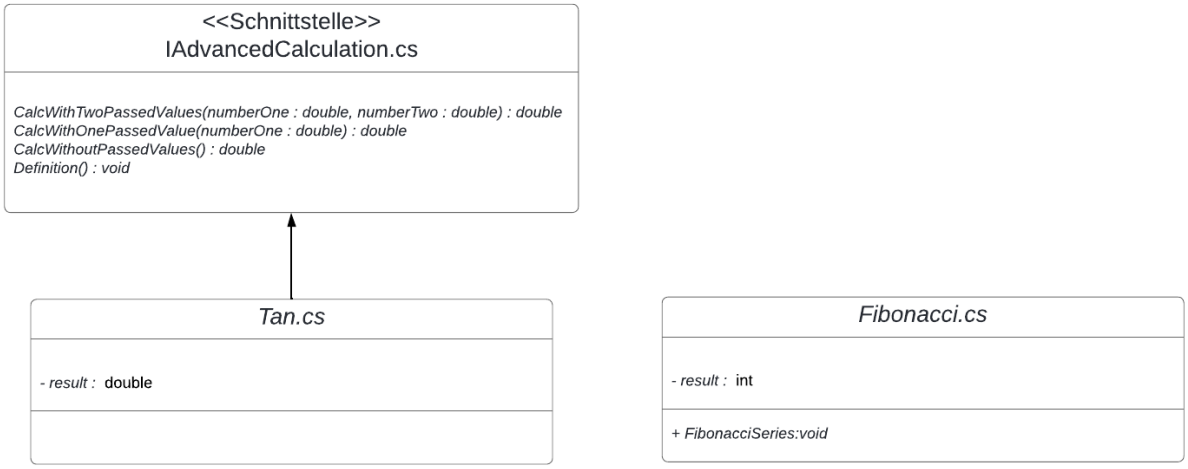


Negativ-Beispiel

Die Klasse *Tan.cs* ist ein negatives Beispiel für SRP, da die Klasse neben dem Durchführen und Beschreiben einer Tangens-Funktion, auch für die Ausgabe einer Fibonacci-Reihe zuständig ist.



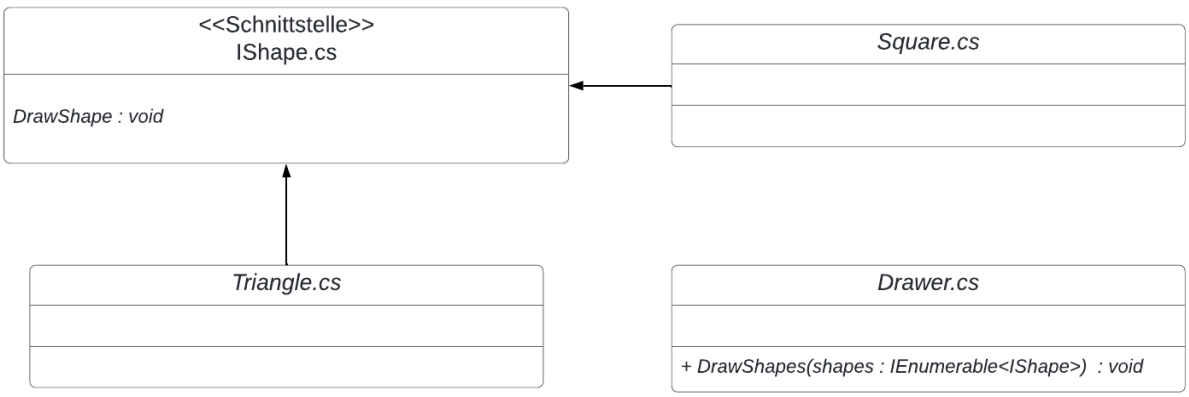
Dies könnte behoben werden indem die FibonacciSeries-Methode einer neuen Klasse *Fibonacci.cs* zugeordnet und aus *Tan.cs* entfernt wird.



Analyse OCP (3P)

Positiv-Beispiel

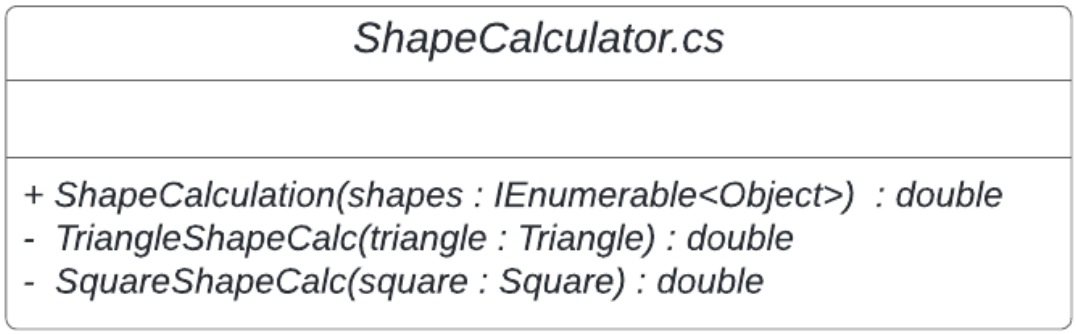
Die Klasse *Drawer.cs* ist ein positives Beispiel für OCP, da falls ein neue Form hinzugefügt wird, wird die bestehende *DrawShapes()*-Methode nicht verändert. Die Methode führt eine foreach-Iteration über shapes aus und ruft für jede Form in shapes die *DrawShape()*-Methode der Form auf. Falls neue Formen hinzugefügt werden sollen, werden diese als eigenständige Klasse definiert und implementieren die *IShape.cs*-Schnittstelle. Die *DrawShape()*-Methode wird dann in der jeweiligen Klasse ausgearbeitet, sodass *Drawer.cs* unverändert bleibt.



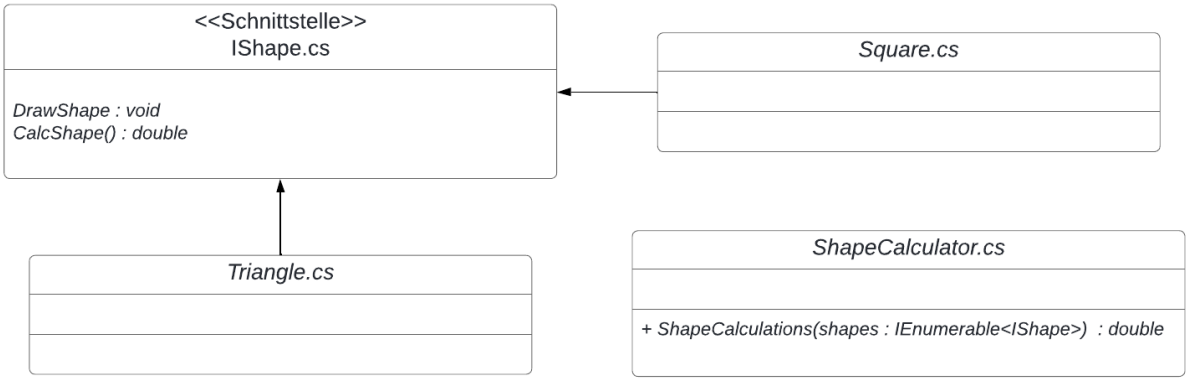
Negativ-Beispiel

Die Klasse *ShapeCalculator.cs* ist ein negatives Beispiel für OCP, da die Kalkulationsmethoden für die Formen in der Klasse definiert werden, anstatt in der

Form-Klasse selbst. Das If-Else-Statement der Methode *ShapeCalculation()* müsste erweitert werden, falls neue Formen hinzu kommen und es müssten neue Methoden für die Formberechnung selbst hinzugefügt werden.



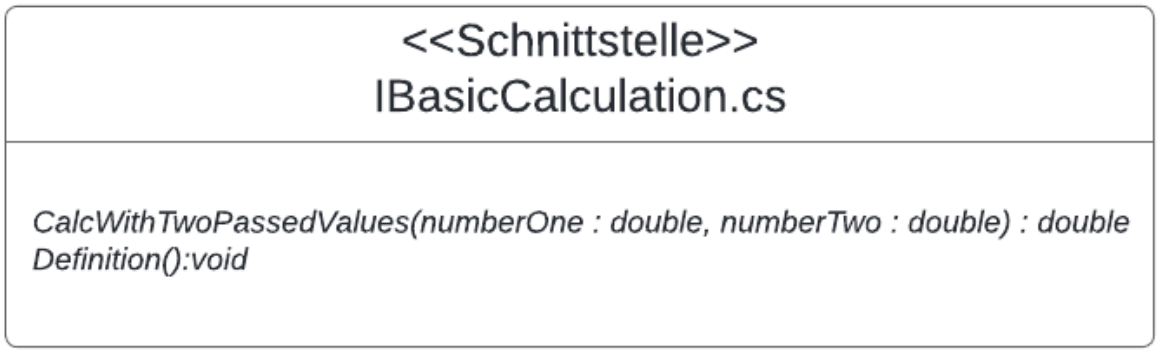
eine mögliche Lösung wäre der IShape.cs-Schnittstelle eine weitere Methode *CalcShape()* hinzuzufügen. Diese würde dann wieder in der jeweiligen Form-Klasse definiert und in der ShapeCalculations()-Methode der *ShapeCalculator.cs*-Klasse aufgerufen. Dies könnte wieder mit einer foreach-Schleife realisiert werden, welche für jede Form in shapes die zugehörige *CalcShape()*-Methode ausführt.



Analyse ISP (2P)

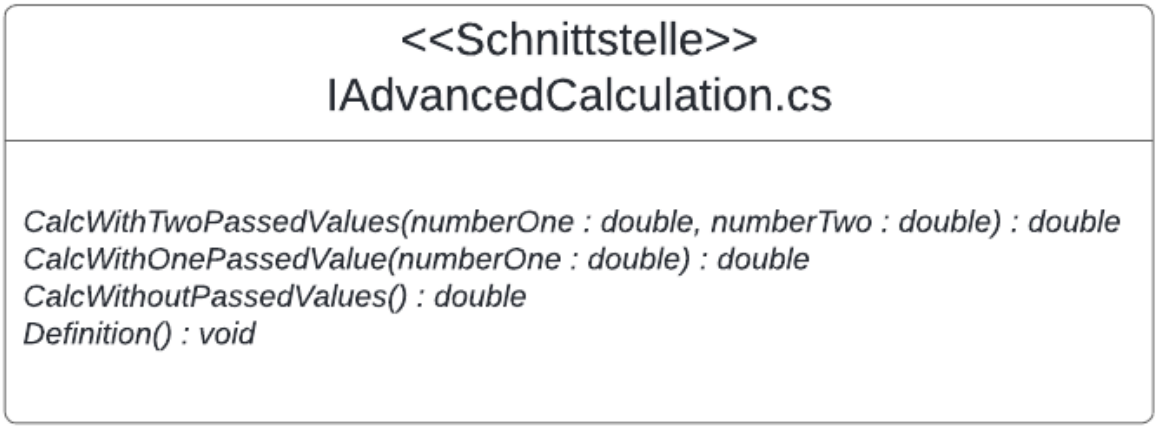
Positiv-Beispiel

Die Schnittstelle *IBasicCalculation.cs* ist ein positives Beispiel für ISP, da die Klassen welche die Schnittstelle implementieren, genau die zwei Methoden *CalcWithTwoPassedValues()* und *Definition()* benötigen.



Negativ-Beispiel

Die Schnittstelle *IAdvancedCalculation.cs* ist ein negatives Beispiel für ISP, da die Klassen welche die Schnittstelle implementieren, zumeist nur die *Definition()*- und eine der Kalkulationsmethoden benötigen. Somit werden mit der Implementation der Schnittstelle zwei für die jeweilige Klasse nicht zu gebrauchende Methoden implementiert.

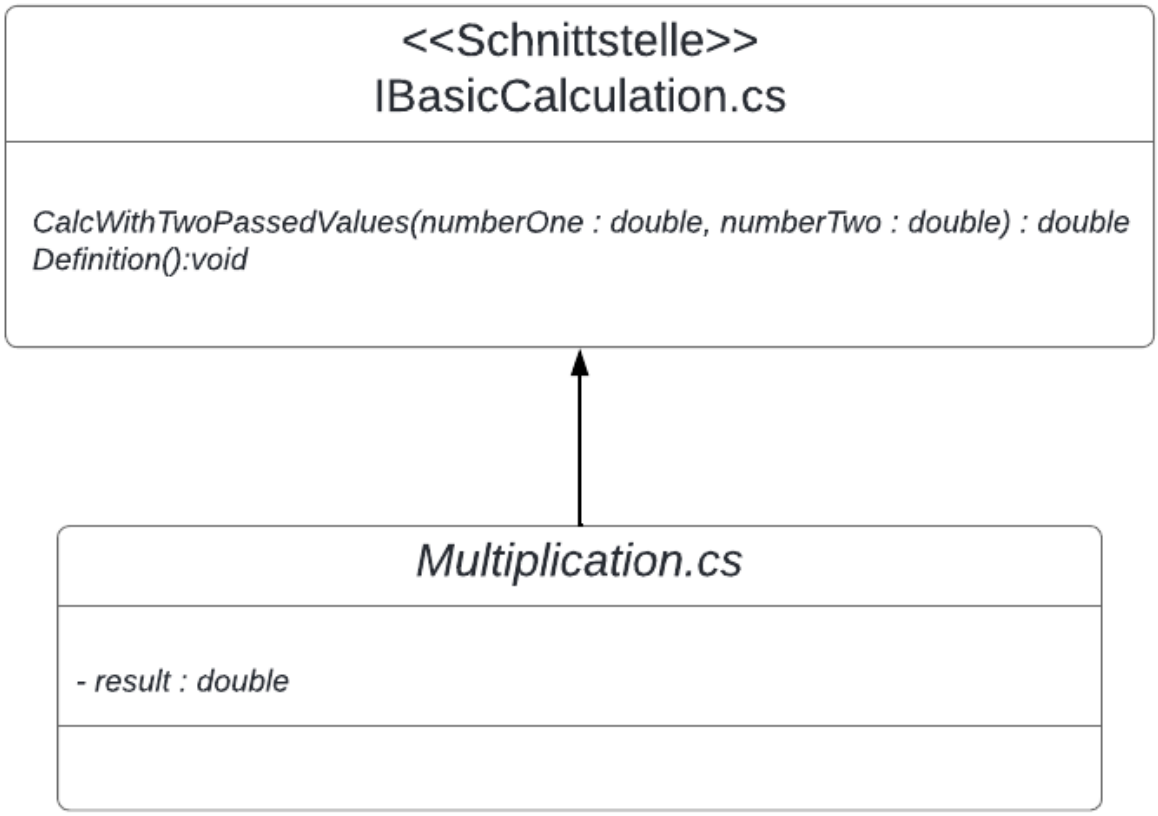


Kapitel 4: Weitere Prinzipien (8P)

Analyse GRASP: Geringe Kopplung (4P)

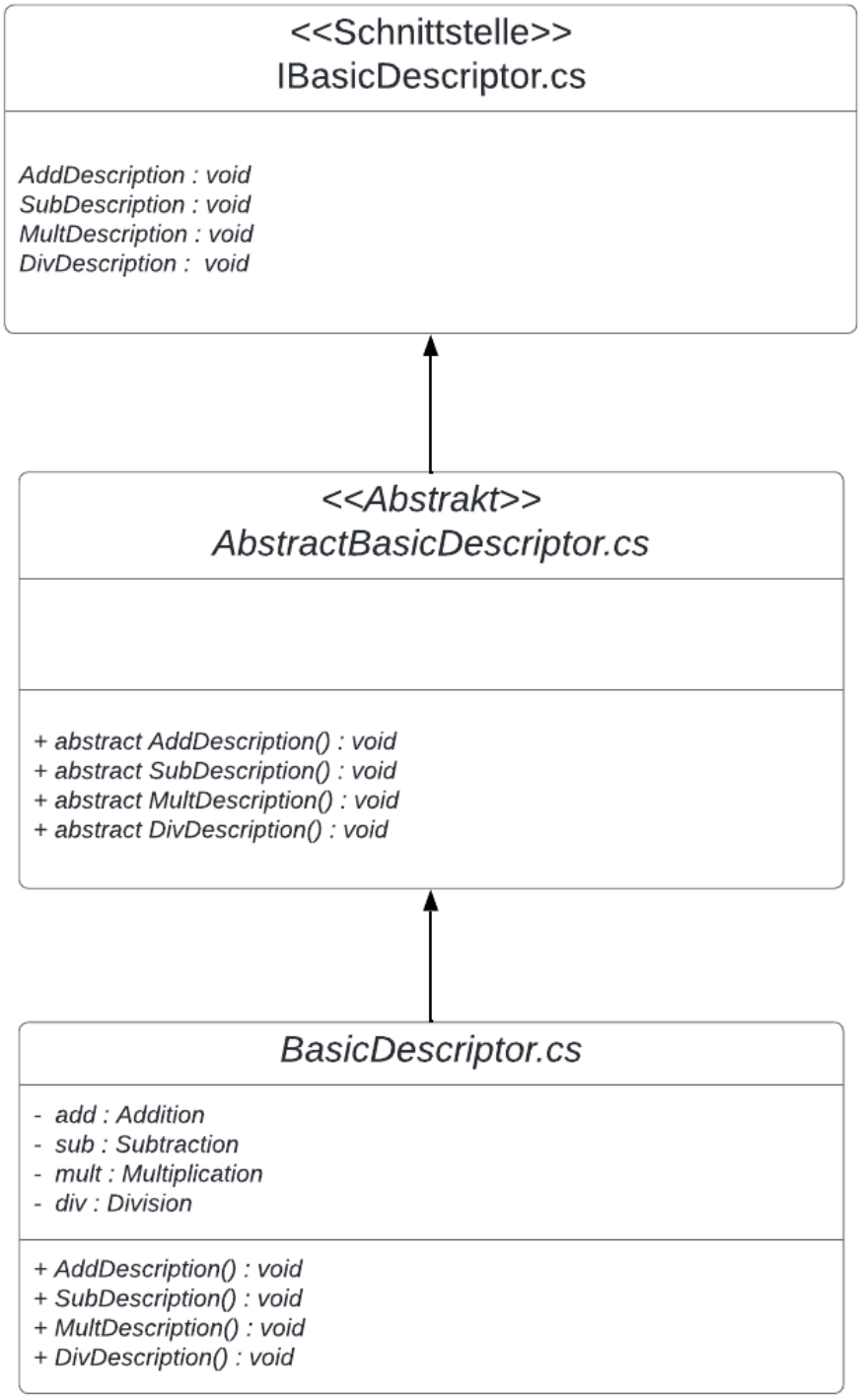
Positiv-Beispiel

Die Klasse *Multiplication.cs* ist ein positives Beispiel für Geringe Kopplung, da die Klasse lediglich eine Schnittstelle *IBasicCalculation.cs* implementiert. Somit betreffen Änderungen anderer Komponenten nur dann *Multiplication.cs*, wenn Änderungen die Schnittstelle betreffen.



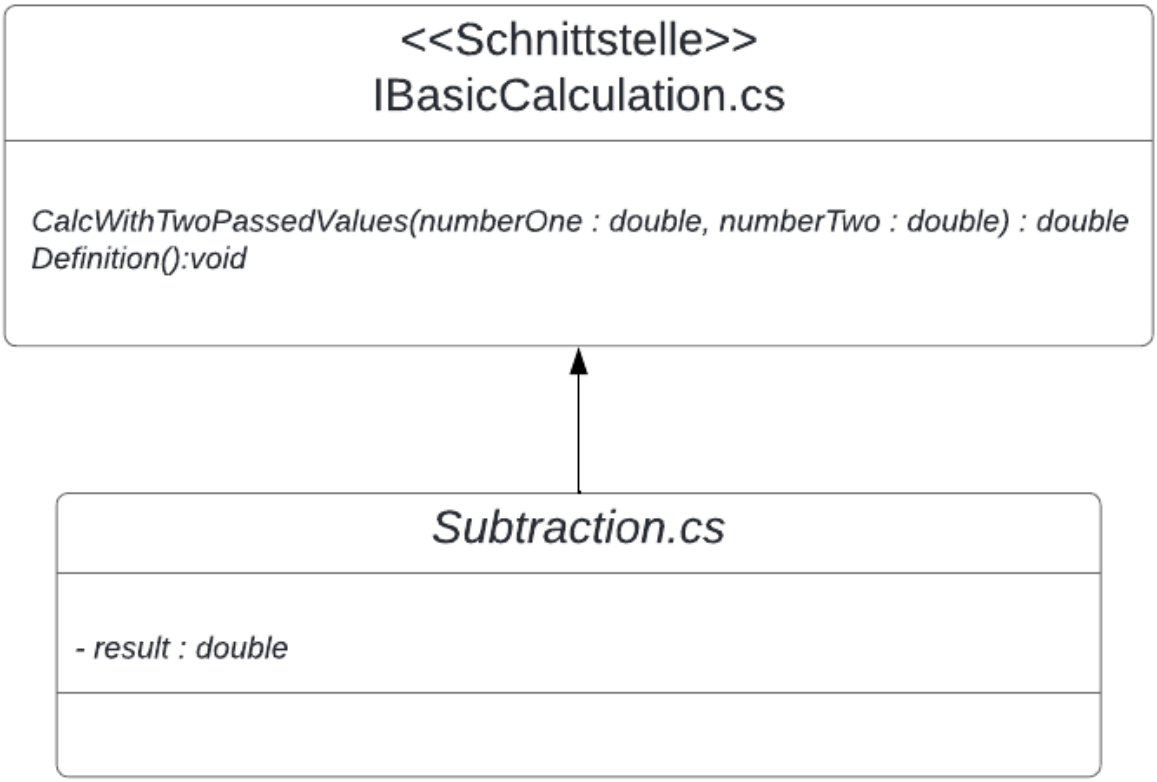
Negativ-Beispiel

Die Klasse *BasicDescriptor.cs* ist ein negatives Beispiel für Geringe Kopplung, da die Klasse zum einen von der abstrakten Klasse *AbstractBasicDescriptor.cs* erbt und dessen abstrakte Methoden überschreiben muss. Zum anderen implementiert *AbstractBasicDescriptor.cs* die Schnittstelle *IBasicDescriptor.cs*, sodass Änderungen der Schnittstelle als auch Änderungen der abstrakten Klasse, Änderungen der Klasse *BasicDescriptor.cs* nach sich ziehen. Die kopplung könnte aufgelöst werden indem auf die Schnittstelle und die abstrakte Klasse verzichtet wird, da beide nur für die Klasse *BasicDescriptor.cs* implementiert wurden und keine andere Klasse weder *IBasicDescriptor.cs* implementiert noch von *AbstractBasicDescriptor.cs* erbt.


<<Schnittstelle>>
IBasicDescriptor.cs
AddDescription : void
SubDescription : void
MultDescription : void
DivDescription : void
<<Abstrakt>>
AbstractBasicDescriptor.cs
+ abstract AddDescription() : void
+ abstract SubDescription() : void
+ abstract MultDescription() : void
+ abstract DivDescription() : void
BasicDescriptor.cs
- add : Addition
- sub : Subtraction
- mult : Multiplication
- div : Division
+ AddDescription() : void
+ SubDescription() : void
+ MultDescription() : void
+ DivDescription() : void

Analyse GRASP: Hohe Kohäsion (2P)

Die Klasse *Subtraction.cs* ist ein positives Beispiel für hohe Kohäsion, da die Klasse rein für das Durchführen und beschreiben einer Subtraktion Verantwortlich ist. Wobei die Methode *CalcWithTwoPassedValues()* eine Subtraktion mit zwei übergebenen Werten durchführt und das Ergebnis zurückliefert. Und die *Definition()*-Methode eine Definition von Subtraktion in der Konsole ausgibt.



DRY (2P)

Vor dem Commit: 786d4e4e12e01edee743840916ddd117e5e971a8

beinhalteten *BasicCalculator.cs* und *AdvancedCalculator.cs* beide Methoden zur Addition, Subtraktion, Multiplikation und Division von Zahlenwerten.

```
1  public class BasicCalculator
2  {
3      private Addition add = new();
4      private Substraction sub = new();
5      private Multiplication mult = new();
6      private Division div = new();
7
8      public double addCalc(double numberOne, double numberTwo)
9      {
10       return add.CalcWithTwoPassedValues( numberOne, numberTwo);
11     }
12     public double subCalc(double numberOne, double numberTwo)
13     {
14       return  sub.CalcWithTwoPassedValues(numberOne, numberTwo);
15     }
16     public double multCalc(double numberOne, double numberTwo)
17     {
18       return mult.CalcWithTwoPassedValues(numberOne, numberTwo);
19     }
20     public double divCalc(double numberOne, double numberTwo)
21     {
22       return div.CalcWithTwoPassedValues(numberOne, numberTwo);
23     }
24 }
```

```
1  public class AdvancedCalculator
2  {
3      private Addition add = new();
4      private Substraction sub = new();
5      private Multiplication mult = new();
6      private Division div = new();
7
8  // weitere Instanzen die fuer dieses Beispiel irrelevant sind...
9
10     public double addCalc(double numberOne, double numberTwo)
11     {
12       return add.CalcWithTwoPassedValues( numberOne, numberTwo);
13     }
14     public double subCalc(double numberOne, double numberTwo)
15     {
16       return  sub.CalcWithTwoPassedValues(numberOne, numberTwo);
17     }
18     public double multCalc(double numberOne, double numberTwo)
19     {
20       return mult.CalcWithTwoPassedValues(numberOne, numberTwo);
21     }
22     public double divCalc(double numberOne, double numberTwo)
23     {
24       return div.CalcWithTwoPassedValues(numberOne, numberTwo);
25     }
26
27 // weitere Methoden die fuer dieses Beispiel irrelevant sind...
28 }
```

Diese Redundanz wurde aufgelöst, indem eine Vererbungsstruktur ausgehend von *AdvancedCalculator.cs* nach *BasicCalculator.cs* implementiert wurde.

```
1  public class AdvancedCalculator : BasicCalculator
2  {
3  // weitere Instanzen die fuer dieses Beispiel irrelevant sind...
4
5  // weitere Methoden die fuer dieses Beispiel irrelevant sind...
6  }
```

Kapitel 5: Unit Tests (8P)

10 Unit Tests (2P)

Unit Test	Beschreibung
BasicCalculatorTest.cs # Addition_Theory()	Diese Theory testet die *AddCalc()*-Methode der Klasse *BasicCalculator.cs* mehrfach mit verschiedenen Zahlenwerten darauf, ob der Kalkulationsvorgang richtige Ergebnisse erzeugt.
BasicCalculatorTest.cs # Subtraction_Theory()	Diese Theory testet die *SubCalc()*-Methode der Klasse *BasicCalculator.cs* mehrfach mit verschiedenen Zahlenwerten darauf, ob der Kalkulationsvorgang richtige Ergebnisse erzeugt
BasicCalculatorTest.cs # Multiplication_Theory()	Diese Theory testet die *MultCalc()*-Methode der Klasse *BasicCalculator.cs* mehrfach mit verschiedenen Zahlenwerten darauf, ob der Kalkulationsvorgang richtige Ergebnisse erzeugt
BasicCalculatorTest.cs # Division_Theory()	Diese Theory testet die *DivCalc()*-Methode der Klasse *BasicCalculator.cs* mehrfach mit verschiedenen Zahlenwerten darauf, ob der Kalkulationsvorgang richtige Ergebnisse erzeugt
AdvancedCalculatorTest.cs # Sin_Test()	Dieser Test, testet die *SinCalc()*-Methode der Klasse *AdvancedCalculator.cs* einmal darauf, ob der Kalkulationsvorgang ein richtiges Ergebnis erzeugt.
AdvancedCalculatorTest.cs # Cos_Test()	Dieser Test, testet die *CosCalc()*-Methode der Klasse *AdvancedCalculator.cs* einmal darauf, ob der Kalkulationsvorgang ein richtiges Ergebnis erzeugt.
AdvancedCalculatorTest.cs # Tan_Test()	Dieser Test, testet die *TanCalc()*-Methode der Klasse *AdvancedCalculator.cs* einmal darauf, ob der Kalkulationsvorgang ein richtiges Ergebnis erzeugt.
AdvancedCalculatorTest.cs # Log2_Test()	Dieser Test, testet die *Log2Calc()*-Methode der Klasse *AdvancedCalculator.cs* einmal darauf, ob der Kalkulationsvorgang ein richtiges Ergebnis erzeugt.
AdvancedCalculatorTest.cs # Log10_Test()	Dieser Test, testet die *Log10Calc()*-Methode der Klasse *AdvancedCalculator.cs* einmal darauf, ob der Kalkulationsvorgang ein richtiges Ergebnis erzeugt.
AdvancedCalculatorTest.cs # Addition_Test()	Dieser Test, testet die *AddCalc()*-Methode der Klasse *AdvancedCalculator.cs* einmal darauf, ob der Kalkulationsvorgang ein richtiges Ergebnis erzeugt.

Hierfür werden *Addition_Theory()* drei Zahlenwerte übergeben. Zahl 1 und 2 werden der

AddCalc()-Methode übergeben und das Ergebnis in einer Variablen *actual* gespeichert. Die 3te Zahl ist das erwartete Ergebnis. Diese wird in einer Variablen *expected* gespeichert. Zu guter Letzt werden *actual* und *expected* mit der *Assert.Equal()*- Methode von xUnit verglichen. Dieser Vorgang wird mehrfach mit verschiedenen Zahlenwerten wiederholt.

ATRIP: Automatic (1P)

Für die Realisierung von Automatic, liefert die verwendete Entwicklungsumgebung Visual Studio bereits ein Kommando “Alle Tests ausführen”. Dies kann entweder über das Menü Test->Alle Tests ausführen ausgewählt werden oder über den Shortcut “Strg + R,A”. Hierbei laufen die Tests dann automatisch ab und überprüfen sich selbst. Es wird nur ein Ergebnis zurückgeliefert, welches besagt ob der Test Erfolgreich oder Fehlerhaft war.

ATRIP: Thorough (1P)

Die Methoden des Calculators der DevTool-Anwendung wurden ausgiebig getestet. Hierfür wurden zwei Testklassen erstellt die die grundlegenden und erweiterten Rechenoperationen des Calculators testen. Der Fokus wurde auf diesen Bereich des DevTools gelegt, da zum einen die Mehrheit des Codes den Calculator betrifft und zum anderen die Ergebnisse der Kalkulationsvorgänge stimmen sollten, da ansonsten der Calculator kein Nutzen für den Anwender hätte. Die anderen Bestandteile des DevTools wurden Stand Heute nicht getestet, könnten allerdings in künftigen Versionen abgedeckt werden.

ATRIP: Professional (1P)

Positiv-Beispiel

Die Methode *Addition_Theory()* der Klasse *BasicCalculatorTest.cs* ist ein positives Beispiel für Professional, da der Test nur eine Aufgabe hat die Methode *AddCalc()* der Klasse *BasicCalculator.cs* zu testen. Des weiteren handelt es sich hierbei um eine Theory d.h. die Methode wird mehrfach mit unterschiedlichen Zahlenwerten getestet. Des Weiteren ist der Code so aufgebaut, dass er mit geringem Aufwand wiederverwendet werden kann. Falls z.B. die *SubCalc()*- Methode getestet werden soll, müssen lediglich die inlineData-Parameter, der Name der Test-Methode und die Methode welche mit der *BasicCalculator*-Instanz aufgerufen wird, geändert werden.

```
1  [Theory]
2  [InlineData(2,2,4)]
3  [InlineData(3, 3, 6)]
4  [InlineData(4, 4, 8)]
5  public void Addition_Theory(double numberOne, double numberTwo,
6                              double numberThree)
7  {
8      //Arrange
9      double expected = numberThree;
10
11     //Act
12     double actual = basicCalculator.AddCalc(numberOne, numberTwo);
13
14     //Assert
15     Assert.Equal(expected, actual);
16
17 }
```

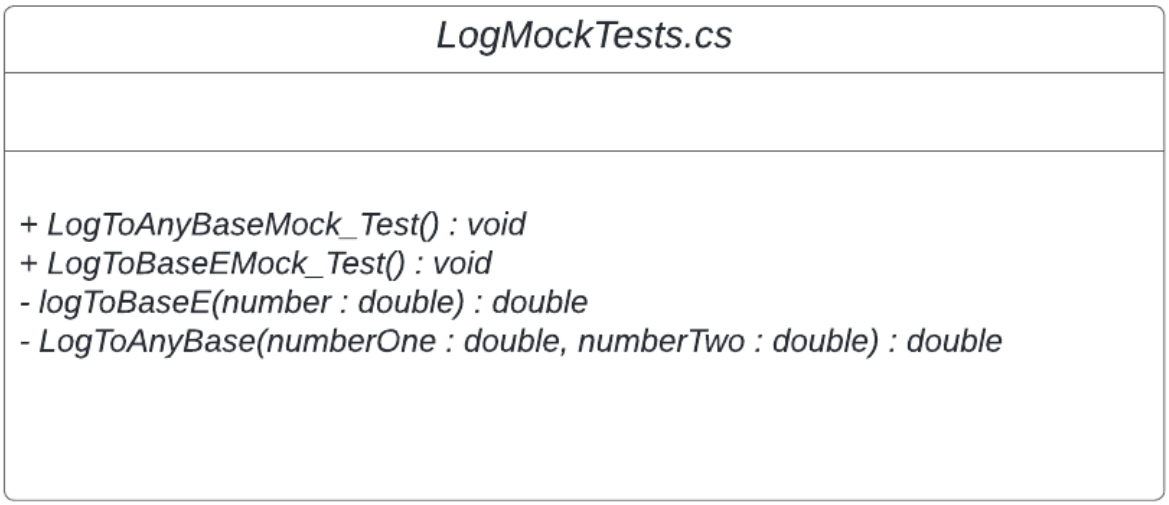
Negativ-Beispiel

Die Methode *Addition_Test()* der Klasse *AdvancedCalculatorTest.cs* ist ein negatives Beispiel für Professional, da *AddCalc()* bereits in der *Addition_Theory()*-Methode der Klasse *BasiCalculatortest.cs* abgedeckt wird und somit Redundant ist. Des weiteren wird die Methode nur einmal mit den immer gleichen Zahlenwerten getestet. Sollte die Methode *AddCalc()* bspw. keine Kalkulation durchführen, sondern immer den Wert 4 zurückgeben, würde der Test als Bestanden gelten obwohl die Methode, die Aufgabe hat zwei Zahlenwerte zu addieren.

```
1  [Fact]
2  public void Addition_Test()
3  {
4      //Arrange
5      double expected = 4;
6
7      //Act
8      double actual = advancedCalculator.AddCalc(2, 2);
9
10     //Assert
11     Assert.Equal(expected, actual);
12
13 }
```

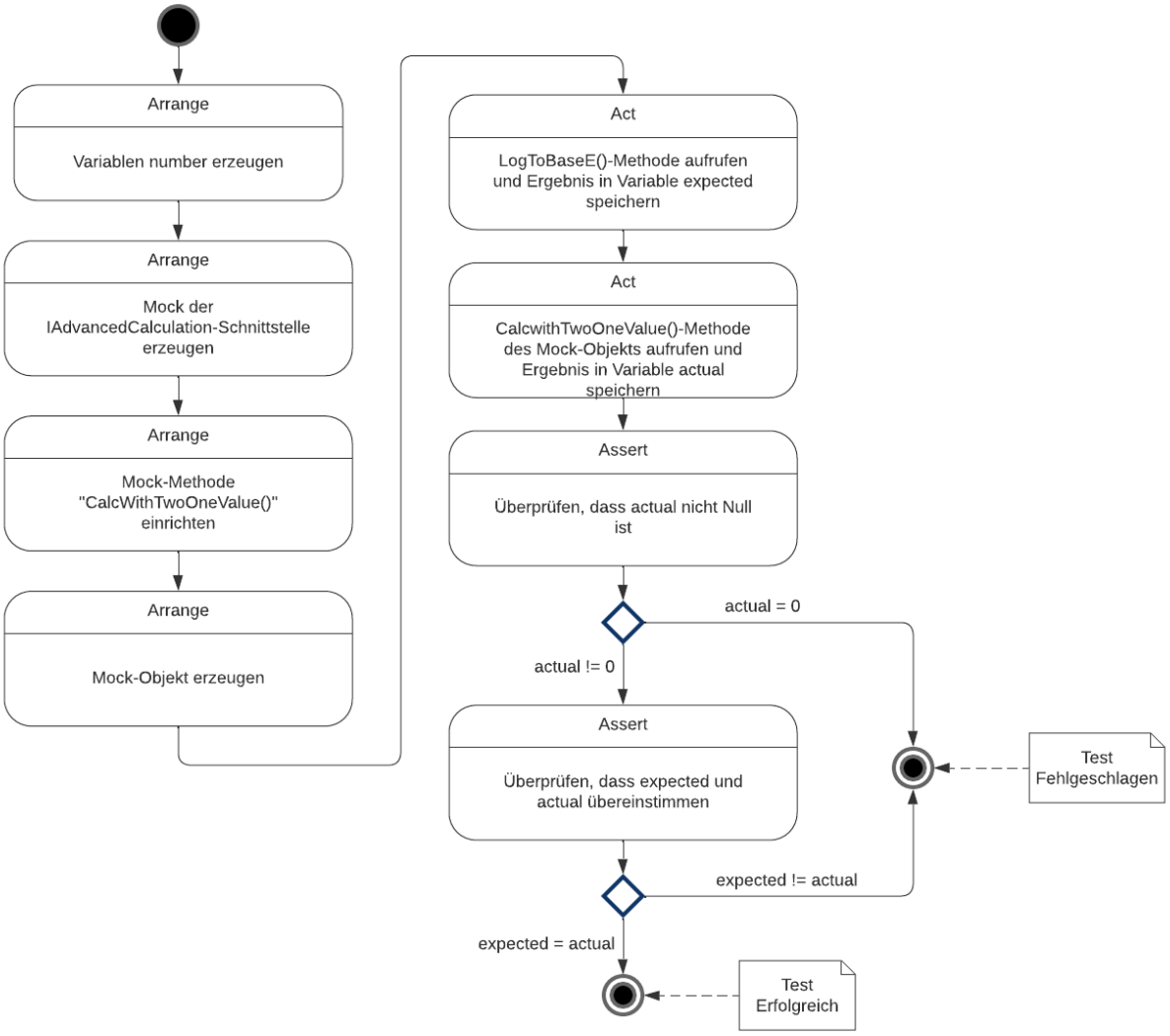
Fakes und Mocks (3P)

Für die Klasse *Log.cs* wurden zwei Tests mit Mock-Objekten implementiert. Dies erfolgte in der Klasse *LogMocktests.cs.* Die Mock-Objekte wurden implementiert, da das reale Objekt nicht durch einen Test beschädigt werden soll.



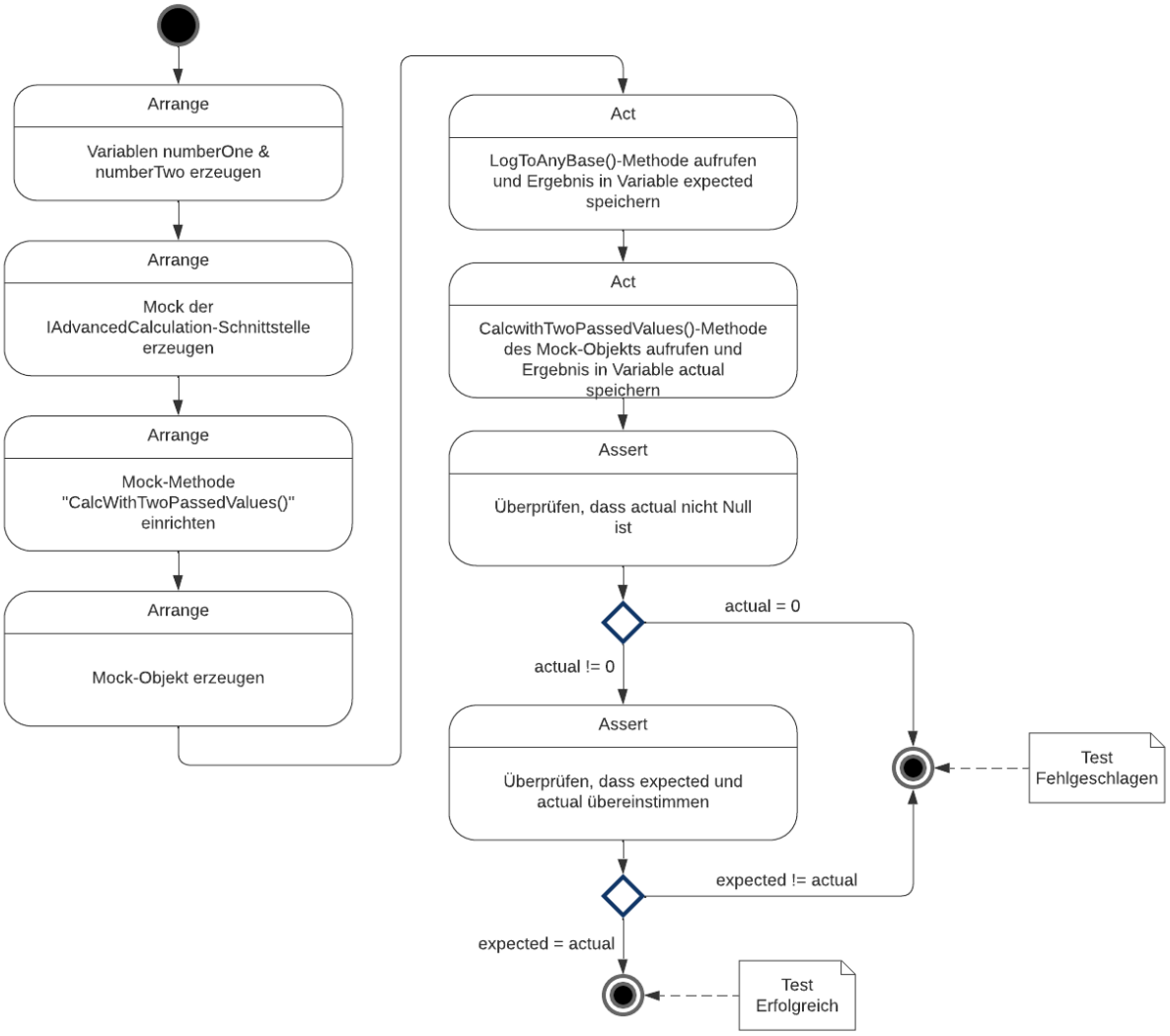
Die Methode *LogToBaseEMock_Test()* ist nach Act-Arrange-Assert aufgebaut. Zuerst werden die benötigten Bausteine für den test eingerichtet. Hierfür wurde eine Variable *number* erzeugt, sowie ein Mock der Schnittstelle *IAdvancedCalculation.cs*. Danach erfolgte ein Setup des Mocks wobei die Methode *CalcWithOnePassedValue()* der Schnittstelle, der return-Wert der Methode *logToBaseE()* zugewiesen wurde. Diese wurde in der Test-Klasse erstellt und

führt den Logarithmus zur Basis e aus. Danach wurde das Mock-Objekt erzeugt. Als Act wurden zwei Variablen angelegt *actual* und *expected*. *actual* ruft die Methode *CalcWithOnePassedValue()* des Mock-Objektes auf und nimmt dessen Rückgabewert entgegen. Während *expected* den Rückgabewert der Methode *logToBaseE()* entgegennimmt. Danach erfolgen zwei Überprüfungen. Zum einen wird überprüft, ob actual nicht null ist und zum anderen werden *expected* und *actual* verglichen. Falls beide Überprüfungen bestanden werden gilt der Test als erfolgreich.



Die Methode *LogToAnyBaseMock_Test()* folgt dem gleichen Muster. Allerdings werden zuerst zwei variablen angelegt *numberOne* und *numberTwo*. Im Setup-Prozess des Mocks wird der Methode *CalcWithTwoPassedValues()* als return Wert *LogToAnyBase()* zugewiesen, welche ebenfalls in der Testklasse implementiert wurde und den Logarithmus zu einer beliebigen

Basis ausführt. Danach wurde das Mock-Objekt erzeugt.Als Act wurden zwei Variablen angelegt *actual* und *expected*. *actual* ruft die Methode *CalcWithTwoPassedValues()* des Mock-Objektes auf und nimmt dessen Rückgabewert entgegen. Während *expected* den Rückgabewert der Methode *LogToAnyBase()* entgegennimmt. Danach erfolgen zwei Überprüfungen. Zum einen wird überprüft, ob *actual* nicht null ist und zum anderen werden *expected* und *actual* verglichen. Falls beide Überprüfungen bestanden werden gilt der test als erfolgreich

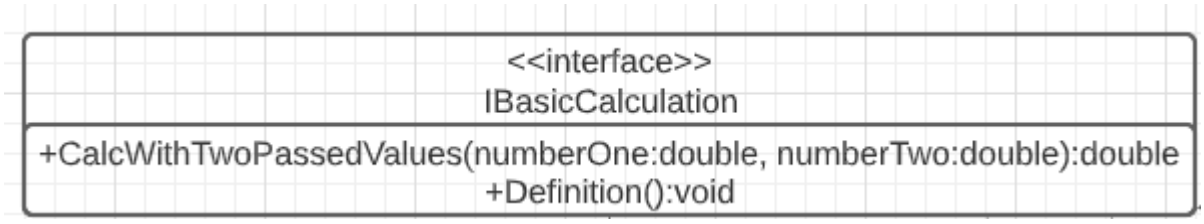


Kapitel 6: Domain Driven Design (8P)

Ubiquitous Language (2P)

Bezeichung	Bedeutung	Begründung
Calculate	Ein Ergebnis wird aus zwei Operanden und einem Operator berechnet	Rechnungen können aus beliebig vielen Operanden und Operatoren bestehen. Bei der Entwicklung stellt das Team aber nur Funktionen zur Verfügung bei denen eine Rechnung aus zwei Operanden und einem Operator besteht. Große Rechnungen können aufgeteilt werden in viele Abschnitte dieser Größe. Die Bezeichnung musste aufgrund dessen definiert werden.
Converter	Ein Zahlenwert wird aus einem Zahlensystem in ein anderes umgewandelt.	Converter gibt es in vielen Bereichen, ob Stromconverter oder Videoconverter. Daher ist eine Definition essenziell.
Descriptor	Erklärung von Rechenarten oder Konvertierung	Descriptor bedeutet auf deutsch Beschreibung. Somit gibt es sehr viele mögliche Bedeutungen von Descriptor.
Drawer	Zeichnet geometrische Formen	Es muss definiert werden was gezeichnet wird vom Drawer.

Repositories (1,5P)



Beschreibung:

Das Interface IBasicCalculation.cs bildet im Code ein Repository. Es deklariert alle Methoden für die Grundrechenarten. Darunter die Methode CalcWithTwoPassedValues(numberOne:double, numberTwo:double) mit der alle Grundrechenarten berechnet werden können. Die Methode soll zwei double Werte vom Nutzer übergeben bekommen.

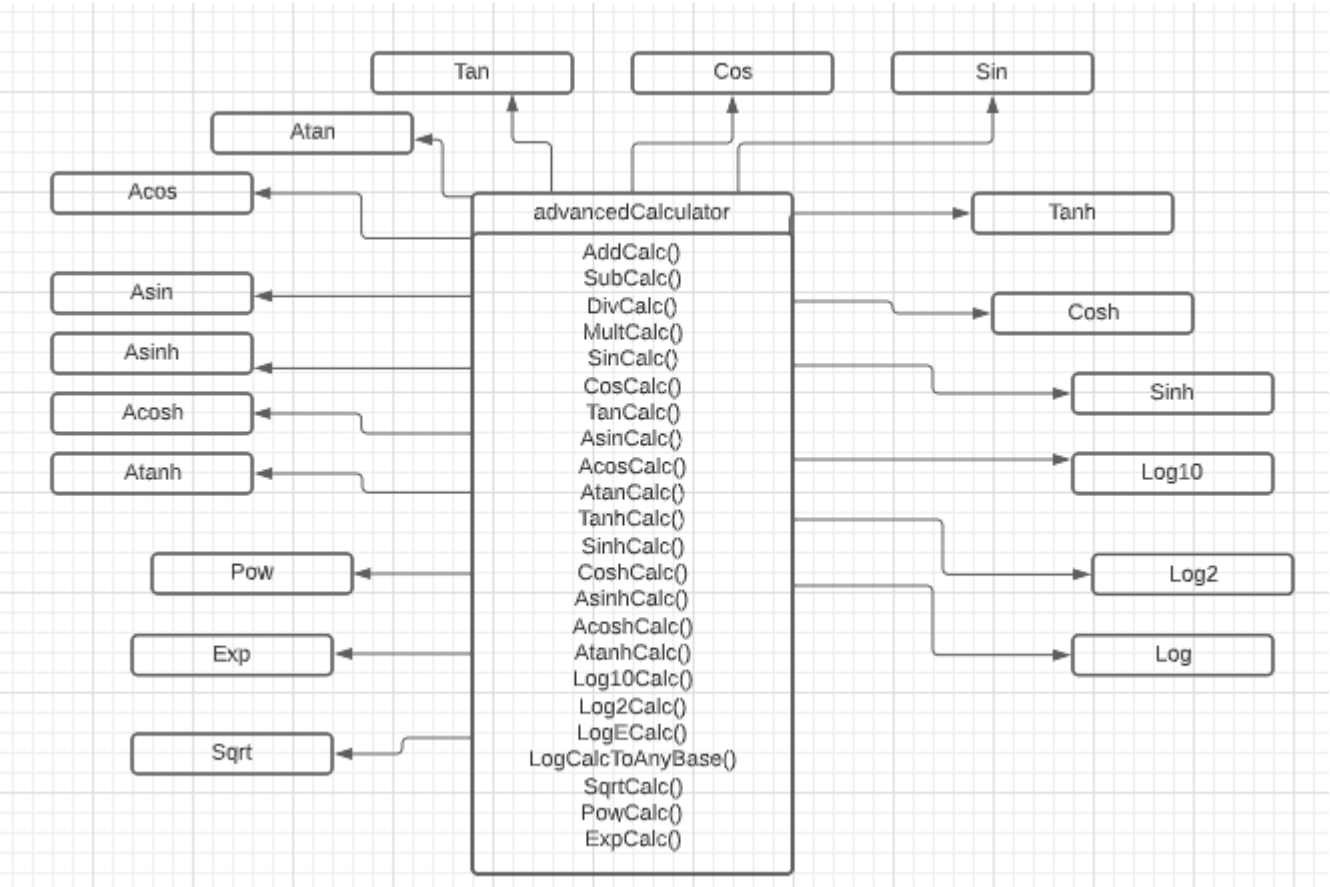
Die andere Methode Definition() soll jede Grundrechenart erklären für den Nutzer.

Aggregates (1,5P)

Es existiert kein Aggregate. Es gibt nicht mehrere Entities, die Bidirektionale Beziehungen zueinander haben, daher sind Aggregate nicht notwendig.

Entities (1,5P)

Es existiert aktuell kein Entity, da noch kein Umgang mit Fehleingaben programmiert wurde und somit der Nutzer die Möglichkeit hat eine Entity nach Konstruktion in einen ungültigen Zustand zu versetzen. Es wäre allerdings durchaus sinnvoll und wird noch umgesetzt, so dass das Programm eine Entity einsetzt. Der aktuelle Stand der Entity wird hier dargestellt.



Value Objects (1,5P)

Es existiert kein Value Object in dem Programm. Jedes Objekt ist durch Methoden veränderbar. Es könnte ein Value Object geben. Objekte haben in dem Projekt keine lange Lebensdauer, da sie direkt nach Erstellung zu Berechnung benutzt werden und danach nicht mehr gebraucht werden. Der Benutzer hat keine Möglichkeit die Werte zu ändern nach Erstellung damit sind somit eigentlich praktisch doch die Objekte unveränderbar. Allerdings haben alle eine Identität und sind somit kein Value Object.

Kapitel 7: Refactoring (8P)

Code Smells (2P)

Große Klasse

Die Klasse *AdvancedDescriptor.cs* enthält zu viele Instanzen, insgesamt 17. Man könnte eine Klasse *TrigonometricDescriptor.cs* und eine Klasse *LogDescriptor.cs* erstellen und die Trigonometrischen und Logarithmischen Definitionen auf diese Weise auslagern. Danach könnte man in *AdvancedDescriptor.cs* jeweils eine Instanz der beiden Klassen aufrufen und somit die insgesamten Instanzen von *AdvancedDescriptor.cs* auf Fünf reduzieren.

```
1   public class AdvancedDescriptor : BasicDescriptor
2   {
3       private TrigonometricDescriptor trgDescriptor = new();
4       private LogDescriptor logDescriptor = new();
5       private Sqrt sqrt = new();
6       private Exp exp = new();
7       private Pow pow = new();
8
9       // Methoden..
10  }
```

Duplizierter Code

Log.cs, *Log2.cs* und *Log10.cs* beinhalteten vor dem Commit: 27c78e676fadfccf22c9b6b73f29e0382cfda63d alle dieselbe Definition. Eine allgemeine für den Begriff Logarithmus. Mit dem Commit beinhalten *Log2.cs* und *Log10.cs* eine Definition, welche die Berechnung zurückgibt und einen Verweis, für eine genauere Erklärung des

Begriffs Logarithmus, die Definition von *Log.cs* auszuführen.

```
1  public class Log10 : IAdvancedCalculation
2  {
3    private double result;
4
5    // weitere Methoden
6
7    public  void Definition()
8    {
9     //Definition Log10
10    // Verweis auf Log fuer genauere Definition von Logarithmus
11   }
12 }
```

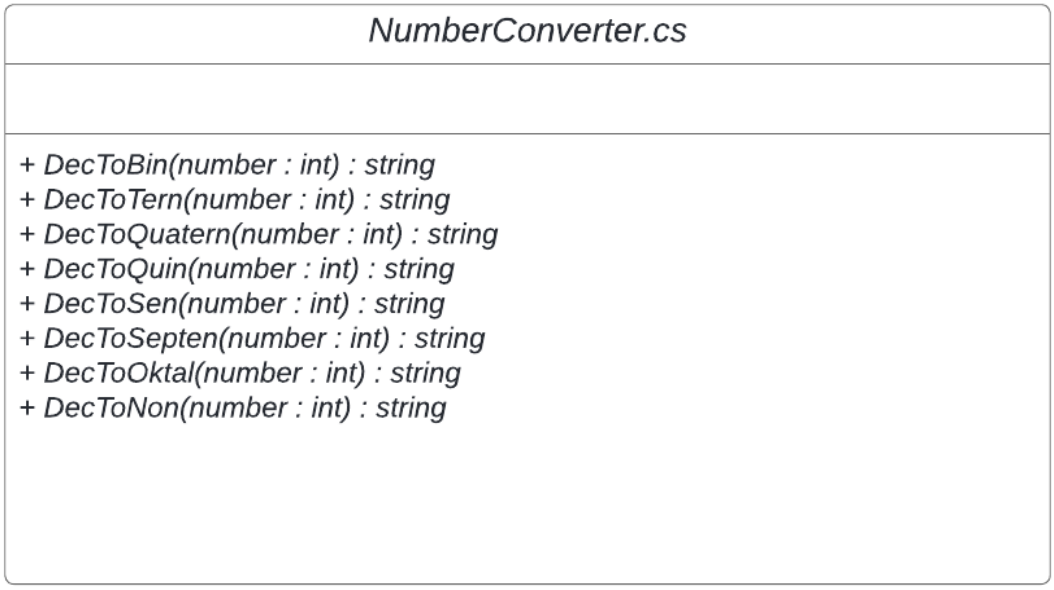
```
1  public class Log2 : IAdvancedCalculation
2  {
3    private double result;
4
5    // weitere Methoden
6
7    public  void Definition()
8    {
9     //Definition Log2
10    // Verweis auf Log fuer genauere Definition von Logarithmus
11   }
12 }
```

2 Refactorings (6P)

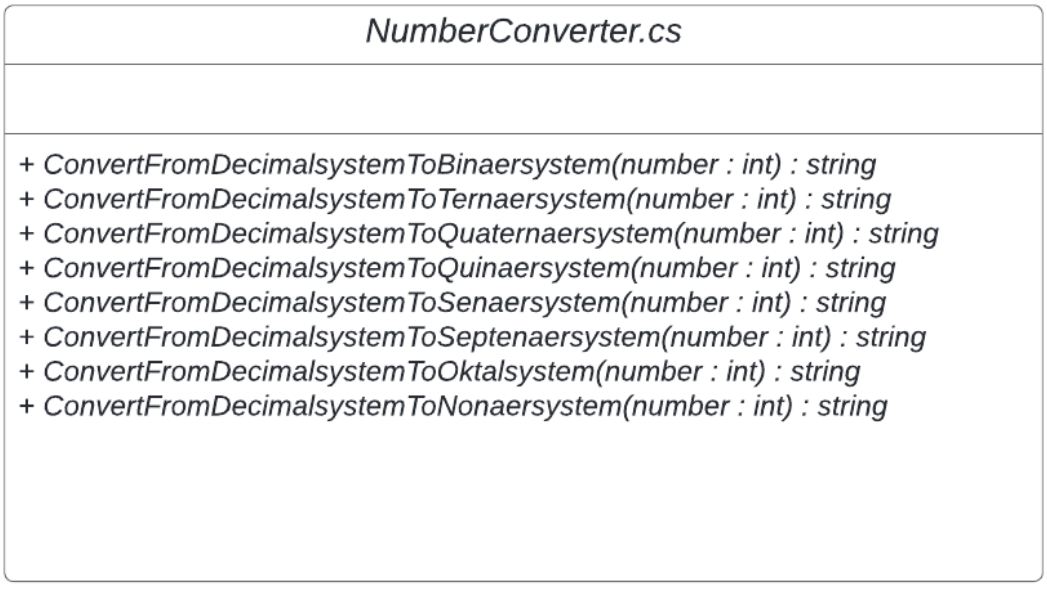
Rename Method:

Vor dem Commit: eadb79ecf00593bcf380a9345dae81c49993a80e, waren die Methodennamen der Klasse NumberConverter.cs kryptisch. Es war nicht auf den ersten Blick ersichtlich für was die methoden zuständig sind. Mit dem Commit wurden die methodennamen dahingehend geändert, dass sofort lesbar ist was die jeweilige Methode ausführt. So wurde bspw. aus DecToBin() - ConvertFromDecimalsystemToBinaersystem(). Hierdurch wurde die Lesbarkeit, Verständlichkeit und Übersichtlichkeit der Klasse sowie Methoden verbessert.

Vorher:



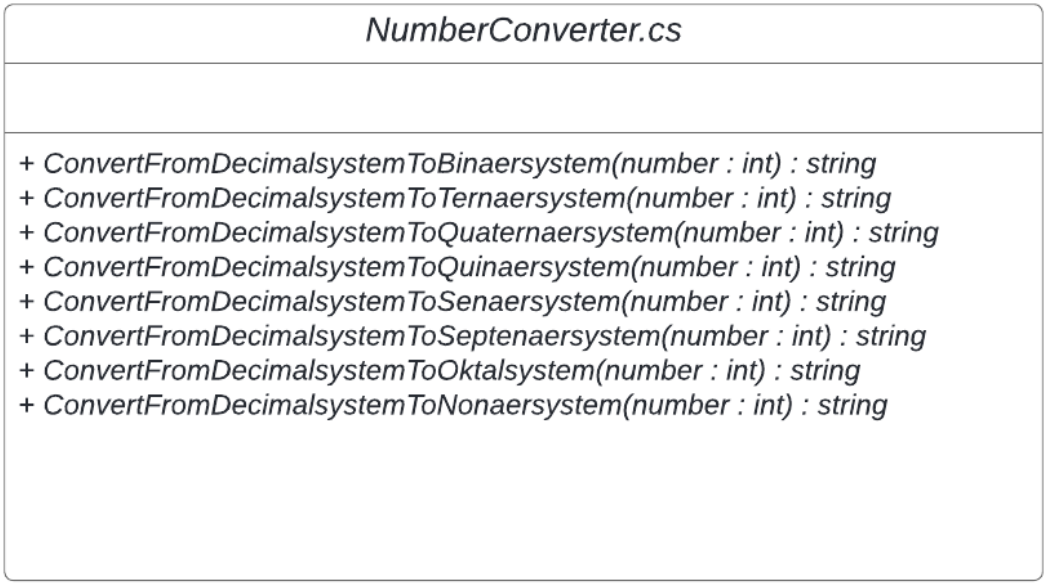
Nachher:



Extract Method:

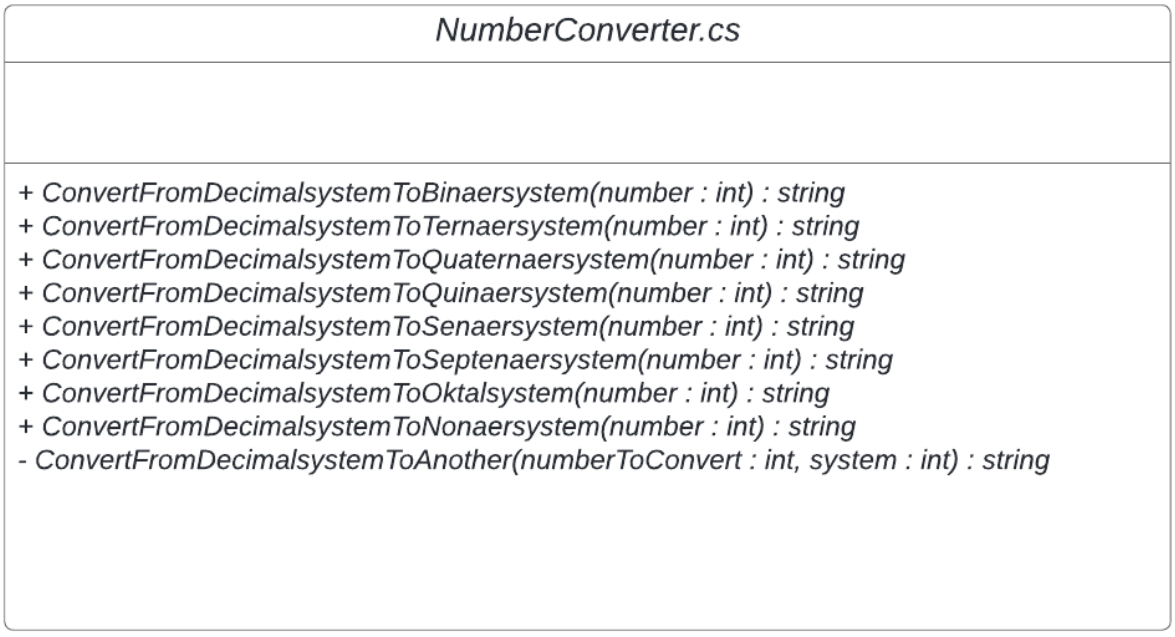
Vor dem Commit: fcb2d961d99280daeadf7dc86e019a78b2e083cd, wurde in jeder Methode der Klasse NumberConverter.cs derselbe Rechenvorgang ausgeführt für die Konvertierung. Mit dem Commit wurde dieser Vorgang in eine extra Methode ConvertFromDecimalsystemToAnother() ausgelagert. Hierdurch wurde redundanz vermieden, sowie die Übersichtlichkeit und die Erweiterbarkeit erhöht.

Vorher:



```
1  public string ConvertFromDecimalsystemToOktalsystem(int number)
2  {
3    string convertedNumber = string.Empty;
4    int remainder;
5    while (number > 0)
6    {
7      remainder = number % 8;
8      number /= 8;
9      convertedNumber = remainder.ToString() + convertedNumber;
10   }
11   return convertedNumber;
12 }
```

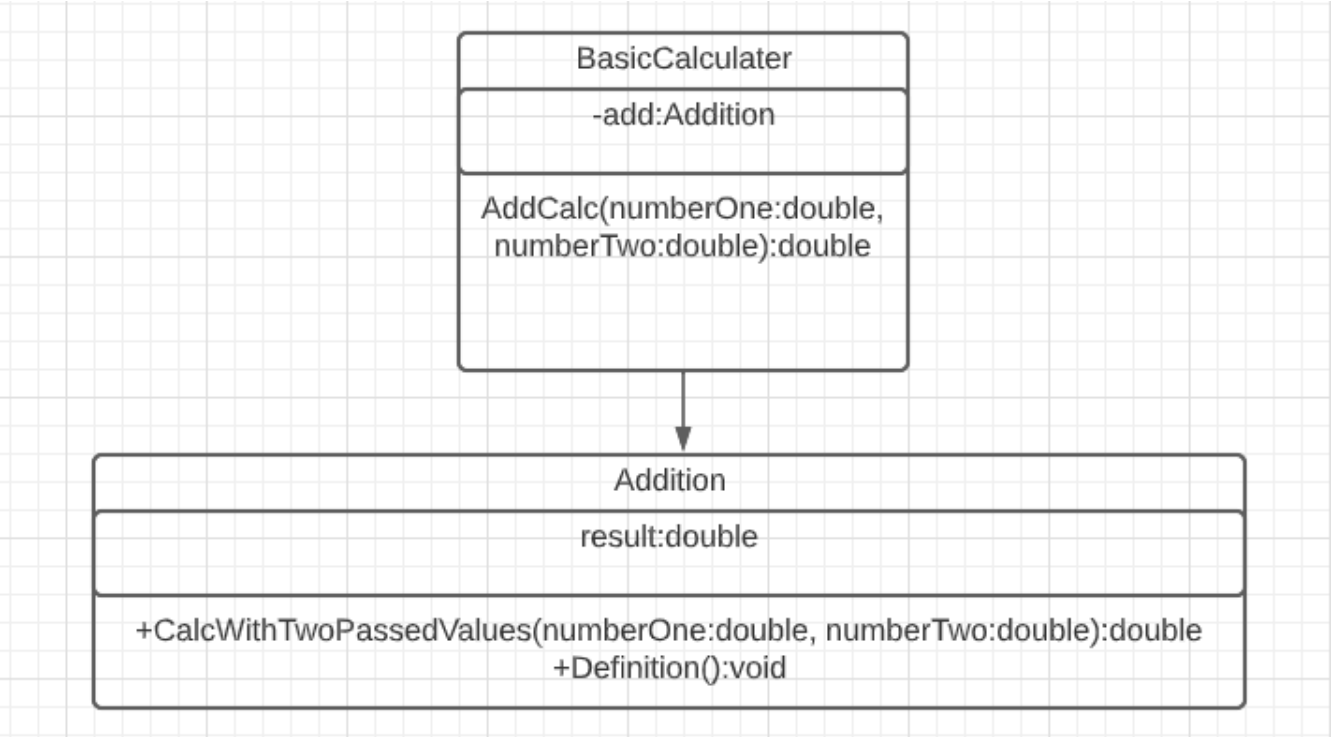
Nachher:



```
1  public string ConvertFromDecimalsystemToOktalsystem(int number)
2  {
3    return ConvertFromDecimalsystemtoAnother(number, 8);
4
5  }
6
7  public string ConvertFromDecimalsystemToNonaersystem(int number)
8  {
9
10   return ConvertFromDecimalsystemtoAnother(number, 9);
11 }
12
13 private string ConvertFromDecimalsystemtoAnother
14                 (int numberToConvert, int system)
15 {
16   string convertedNumber = string.Empty;
17   int remainder;
18
19   while (numberToConvert > 0)
20   {
21     remainder = numberToConvert % system;
22     numberToConvert /= system;
23     convertedNumber = remainder.ToString() + convertedNumber;
24   }
25   return convertedNumber;
26 }
```

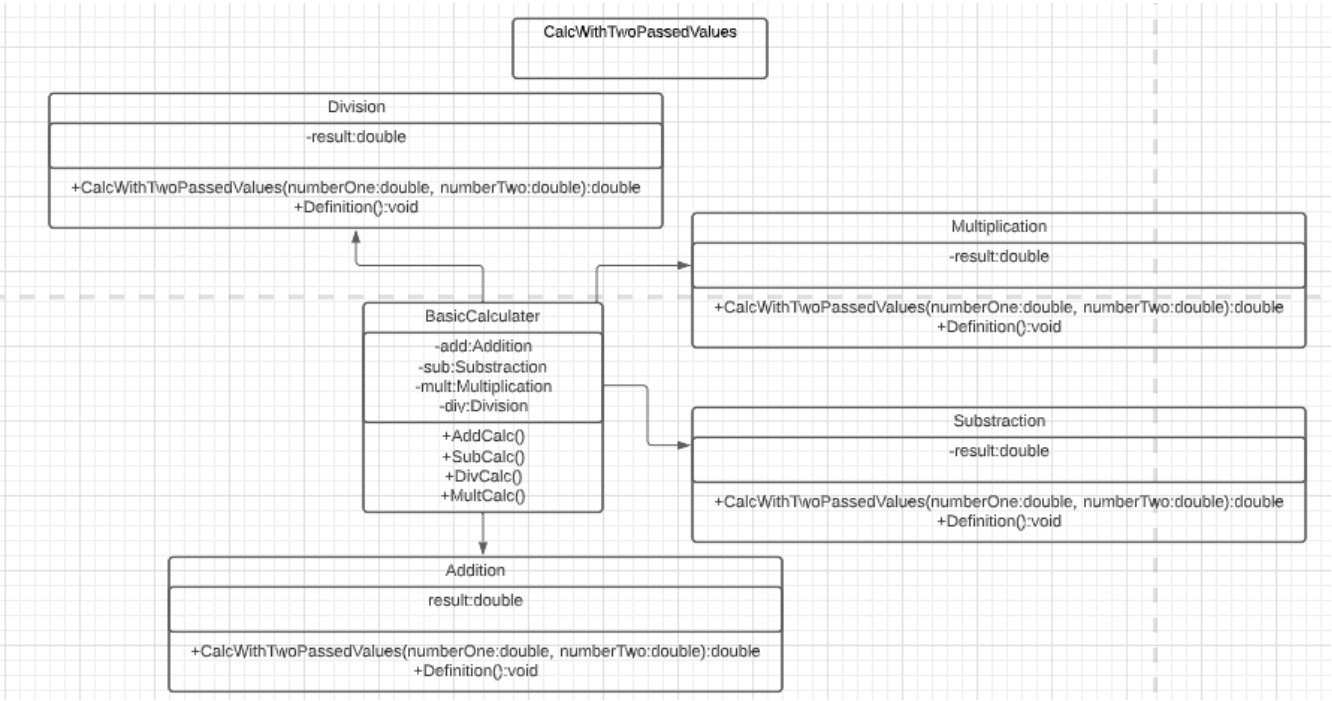
Kapitel 8: Entwurfsmuster (8P)

Entwurfsmuster: Singleton (4P)



Begründung:
Die Klasse Addition.cs wurde als Singleton entworfen, da es immer nur eine Instanz dieses Objekts geben darf. Gibt es mehr als eine Instanz dieses Objekts, so kann es beispielsweise zu fehlerhaften Anzeigen innerhalb der Anwendung kommen.

Entwurfsmuster: abstract Factory (4P)



Begründung:
Die Methode CalcWithTwoPassedValues entscheidet abhängig von der vom Nutzer gewählten Rechenart, wie die zwei Werte miteinander verrechnet werden. Hierzu wird, je nach Rechenart, eines der Objekte Addition, Subtraction, Multiplication, Division, Log oder Pow aufgerufen und mit dem entsprechenden Operator ein Ergebnis wiedergegeben. Ein Factory-Pattern macht hier Sinn, da so die CalcWithTwoPassedValues beliebig erweitert werden kann, sollten weitere Rechenarten hinzukommen.