Anwendung von Design Patterns

Matthias Haselmaier

Zusammenfassung
Diese Arbeit beschäftigt sich mit Design Patterns. Es wird beschrieben was Design Patterns sind und weshalb es sie gibt. Es werden kurz die drei Hauptklassen beschrieben, in welche sie sich einteilen lassen. Als Anwendungsbeispiel wird Schrittweise eine Anwendung zur Auswertung arithmetischer Ausdrücke erarbeitet. Hierbei werden 7 Design Patterns implementiert und besprochen welche Vor- und Nachteile sie mit sich bringen. Anschließend wird gezeigt, wie diese Anwendung um weitere Funktionalitäten erweitert werden kann und besprochen, in wie weit diese Erweiterungen durch die implementierten Design Patterns bei ihrer Integration beeinflusst wurden.

Keywords
Design Patterns, Java, Composite, Bridge, Iterator, Strategy, Factory, Interpreter, Visitor

Hochschule Kaiserslautern
Corresponding author: maha0039@stud.hs-kl.de

Inhaltsverzeichnis

Einleitung

In der Softwareentwicklung stellt sich oft die Frage, wie sich ein bestimmtes Problem am besten lösen lässt. Hierbei ist die Wiederverwendbarkeit der Lösung ein großes Kriterium zur Auswahl, da sich in folgenden Projekten meist ähnliche Problemstellungen ergeben. Für viele solcher häufig auftretenden Probleme gibt es so genannte Design Patterns, welche als allgemein verwendbare Lösung in einem gewissen Kontext eingesetzt werden können. Ursprünglich kommt die Idee der Patterns aus der Architektur, als Christopher Alexander zwischen 1977 und 1979 versuchte, mittels Entwurfsmuster die Bewohner der zu bauenden Gebäude in den Entwicklungsprozess zu integrieren. Erst 1987 wurde diese Idee in der Softwareentwicklung von Kent Beck und Ward Cunningham aufgegriffen, als sie Muster zur Erstellung für grafische Benutzeroberflächen entwickelten. Allerdings haben Design Patterns erst nach der Veröffentlichung des Buches *Design Patterns: Elements of Reusable Object-Oriented Software* [1] von den vier Autoren Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides, welches sich als Standardwerk etablierte, eine gewisse Popularität erreichen können. Über die Jahre haben sich auch einige negative Meinungen über Design Patterns gesammelt. Diese beruhen sehr oft aber auf einer falschen Anwendung. Wie bereits erwähnt sind diese Patterns immer für nur gewissen Kontexte in Erwägung zu ziehen, was ein Anwenden ohne genaues Abwiegen nicht ermöglicht. Es sollte immer ausgiebig über die jeweilige Situation nachgedacht werden und hierbei auch über die Nachteile, die das jeweilige Pattern mit sich bringt.

In den nächsten Abschnitten wird zunächst der Begriff Design Patterns genauer definiert und es wird auf die Klassifizierungen von Patterns eingegangen. Anschließend werden anhand einer Anwendung, welche dazu dient arithmetische Ausdrücke auszuwerten, verschiedene Patterns beispielhaft implementiert und erklärt. Des Weiteren wird diese Anwendung um weitere Funktionalitäten erweitert. Danach folgt eine Evaluierung der zuvor betrachten Anwendung und der in ihr verwendeten Design Patterns.

1. Design Patterns

Grundsätzlich zielen Design Patterns darauf ab, eine abstrakte und allgemeine Lösung zu einer bestimmten Klasse von Problemstellungen zu definieren, welche nicht spezifisch für eine bestimmte Programmiersprache, sondern universell anwendbar sind. Durch Anwendung von Design Patterns kann die Entwicklungsphase beschleunigt werden, da sie als getestetes und erwiesenes Entwicklungsparadigma dienen können. Sie helfen kleinere Fehler in der Architektur der Software schon

früh vorzubeugen, welche häufig zu sehr großen Problemen führen können, wenn sie erst in späterer Entwicklung entdeckt werden und die Refaktorisierung von großen Teilen des Programmcodes erfordern. Des Weiteren wird die Lesbarkeit des Codes gesteigert, wenn das angewendete Design Pattern bekannt ist. Auch ermöglicht die Abstraktion der Problemlösung ein vereinfachtes Diskutieren über die Vor- und Nachteile. Es ist aber auch zu beachten, dass viele Patterns weitere Stufen der Indirektion in das Programm einbringen können und dadurch das entstehende Design komplizieren und die Performanz beeinträchtigen könnten. Hierdurch wird aber auch ein gewisser Grad an Flexibilität gewonnen. Weiterhin ist zu bemerken, dass kein Design Pattern als universelle Lösung zu verwenden ist. Auch zu viele Patterns, oder auch für die aktuelle Problematik ungeeignete Patterns sind zu vermeiden. Wird dies nicht eingehalten, führt es zu einem so genannten Antipattern. Dies kann besonders unerfahrenen Entwickler passieren, welche versuchen möglichst viele ihrer bekannten Design Patterns einzubringen. Ein Großteil der verbreiteten Design Patterns können in Komponenten implementiert werden, so dass sie nicht jedes mal komplett neu geschrieben werden müssen. Allgemein werden Design Patterns in die folgenden drei Klassen unterteilt:

- **Creational Patterns.** Diese Klasse von Patterns haben die Separierung des Systems von der Art und Weise, wie die Objekte erzeugt, zusammengesetzt und repräsentiert werden, als Hauptziel. Hierdurch wird die Flexibilität des Systems in Bezug auf die Fragen wann, wie und welches Objekt durch wen erzeugt wird gesteigert. Diese Art von Patterns können weiterhin in zwei Unterklassen unterteilt werden. Object-Creational Patterns kümmern sich um die eigentliche Erzeugung der Objekte und Class-Creational Patterns sind für die Instanziierung der jeweiligen Klasse zuständig.

- **Structural Patterns.** Structual Patterns werden dazu eingesetzt, um auf einfache Weise die Beziehungen zwischen verschiedenen Entitäten innerhalb des Systems zu identifizieren und realisieren.

- **Behavioral Patterns.** Diese Art von Patterns identifizieren und realisieren verbreitete Verhaltensmuster zwischen den einzelnen Objekten eines Systems. Hiermit wird die Flexibilität bei der Kommunikation der Objekte erhöht.

Über die Zeit wurden einige Design Patterns sogar in bestimmte Programmiersprachen integriert. So ist beispielsweise das Iterator Pattern wesentlicher Bestandteil der Sprachen C++ und Java, wobei die foreach-Schleifen auf Iteratoren zurückgreifen, welche für die jeweiligen Collection-Objekte der Sprachen implementiert wurden. Auch das Singleton Pattern, welches wohl eines der bekanntesten ist, wurde in Scala als fester Bestandteil der Sprache eingebaut. Um maximal nur ein Objekt einer Klasse erzeugen zu lassen wird hierzu in Scala das Schlüsselwort *class* durch *object* ersetzt.

Im folgenden Abschnitt wird nun die Implementierung einiger Design Patterns am Beispiel einer Anwendung zur Auswertung arithmetischer Ausdrücke gezeigt.

2. Auswertung arithmetischer Ausdrücke

In diesem Abschnitt wird der Aufbau einer Anwendung zur Auswertung arithmetischer Ausdrücke besprochen. Hierbei soll ein einfacher String als Eingabe dienen, welcher den Ausdruck repräsentiert. Ein solcher Ausdruck soll zu Demonstrationszwecken lediglich aus natürlichen Zahlen, Operatoren und Klammern zusammengesetzt sein. Zulässige Operationen sind Addition(+), Subtraktion(-), Multiplikation(*) und Division(/). Innerhalb des Ausdrucks haben die Klammern die höchste Priorität, danach gilt, wie allgemein üblich, Punkt vor Strich. Folgen zwei Operationen mit der selben Priorität aufeinander, so wird der Ausdruck von links nach rechts ausgewertet. Weiterhin wird davon ausgegangen, dass der eingegebene Ausdruck frei von Fehlern ist, um das verarbeiten zu vereinfachen.

Da der eingegebene Ausdruck als String nur schwer auszuwerten ist, sollte er zuerst verarbeitet und aufbereitet werden. Anschließend kann man das Ergebnis dieses Schrittes zur eigentlichen Auswertung des Ausdrucks verwenden. Eine solche System-Architektur ist allgemein als Pipes-and-Filters bekannt. Hierbei werden Daten durch so genannte Filter modifiziert, beziehungsweise transformiert. Die Filter sind durch Pipes miteinander verbunden. Pipes wird aus dem englischen Begriff Rohrleitung hergeleitet, und ist im Grunde ein simples Weiterreichen der Daten, welches beispielsweise durch das Übergeben als Parameter an eine Methode realisiert werden kann. Als interne Datenstruktur bietet sich ein Binärbaum an. Hierdurch wird ein hoher Grad an Flexibilität bei der Verarbeitung der Daten erzielt. So kann zum Beispiel nur ein Teil des Ausdrucks ausgewertet werden, oder es könnten verschiedene Eingabe- und Traversierungsmöglichkeiten unterstützt werden. Der in Abbildung 1 gezeigte Ausdruck könnte demnach durch folgende Darstellungen repräsentiert werden:

- **In-Order Darstellung:** 3 + 4 * 2 - 4 / 2
- **Pre-Order Darstellung:** - + 3 * 4 2 / 4 2
- **Post-Order Darstellung:** 3 4 2 * + 4 2 / -
- **Level-Order Darstellung:** - + / 3 * 4 2 4 2

In den nächsten Abschnitten wird nun auf die Implementierung dieser Anwendung mittels Design Patterns eingegangen.

2.1 Composite-Pattern

Das Composite-Pattern fällt in die Klasse der Structural Patterns. Dieses Pattern hat die Absicht, Objekte in einer Teil-Ganzes-Hierarchie darzustellen, indem sie in einer Baumstruktur zusammengesetzt werden. Damit das Pattern als Lösung in

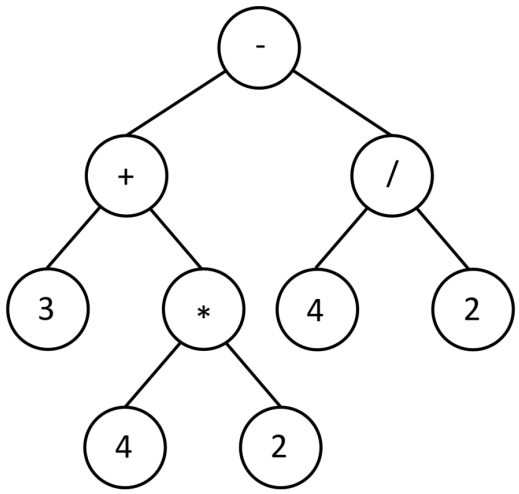

Abbildung 1. Der Ausdruck 3 + 4 * 2 - 4 / 2 als Binärbaum.

Betracht kommt, müssen die Objekte rekursiv zusammengesetzt werden können. Es sollte keine Unterscheidung zwischen einem individuellen und einem zusammengesetzten Objekt geben und beide sollen gleich behandelt werden. Mit diesem Pattern werden sehr häufig Baumstrukturen, wie beispielsweise Dateisysteme, aufgebaut, da so mit den Blättern und den inneren Knoten auf gleiche Art und Weise interagiert werden kann. Dies gestaltet den Code weniger komplex und bietet eine höhere Fehlersicherheit. Abbildung 2 zeigt die grundlegende Struktur des Composite-Patterns.

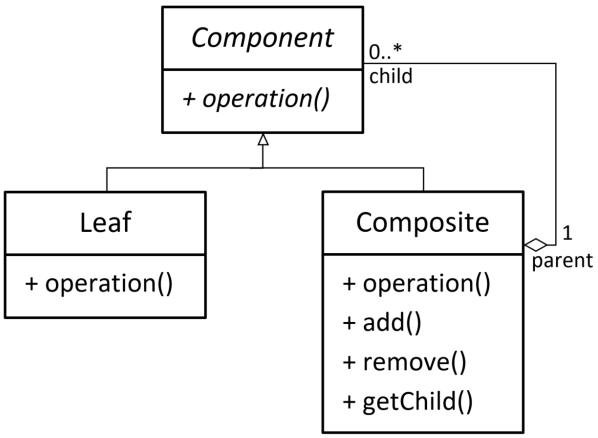

Abbildung 2. Composite-Pattern.

Bei der Implementierung besteht die Frage, ob der Component Typ als Interface oder als abstrakte Klasse realisiert werden sollte. Fällt die Entscheidung auf eine abstrakte Klasse, so könnte es bei späteren Änderungen zum so genannten Fragile-Base-Class Problem kommen. Hierbei wirkt sich eine Änderung in der Basisklasse, welche nicht über die genauen Implementierungsdetails der abgeleiteten Klassen verfügt, eventuell negativ auf die Implementierung der spezialisierten Typen aus. Bei einem Interface fällt die Implementierung aus dem Component Typ weg. Diese Möglichkeit wurde bei der Anwendung auch so implementiert. Zusätzlich werden der Blattknoten und der innere Knoten noch von einer abstrakten Klasse *ANode* abgeleitet, welche den Knoteninhalt verwaltet. Das Klassendiagramm des Composite-Patterns wird in Abbildung 3 dargestellt.

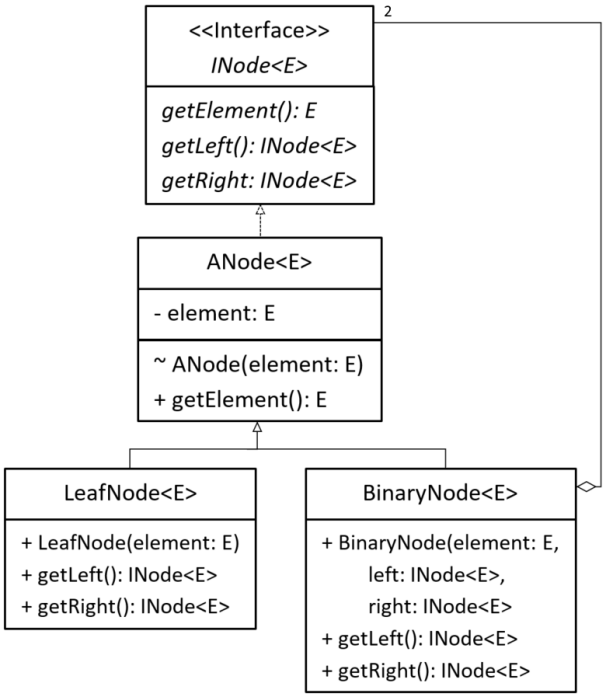

Abbildung 3. Composite-Pattern in der Anwendung.

2.2 Bridge-Pattern

Wird das Bridge-Pattern eingesetzt, so wird die Abstraktion von ihrer jeweiligen Implementierung entkoppelt, damit beide unabhängig voneinander veränderbar sind. Dieses Pattern ist besonders dann nützlich, wenn sowohl die Klasse selbst, als auch das was sie tut häufig geändert wird. Hierbei kann die eigentliche Klasse als Abstraktion gesehen werden und das was sie tut als die Implementierung. Ist nur eine feste Imple-

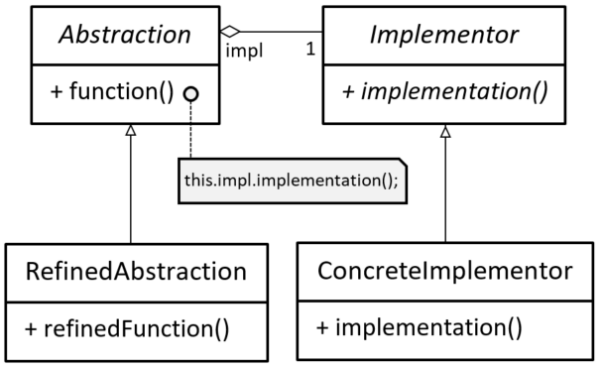

Abbildung 4. Bridge-Pattern.

mentierung vorhanden, entspricht dieses Pattern dem Pimpl Idiom von C++. Durch die Entkoppelung wird also erreicht, dass die Abstraktion und die Implementierung erweiterbar bleiben und die Implementierung ohne Auswirkung auf den Klienten veränderbar bleibt, oder sogar ganz verborgen wird. Der Aufbau dieses Patterns wird in Abbildung 4 gezeigt. Für die Implementierung innerhalb der Anwendung ist es hier nur interessant, die Realisierung der Baumstruktur vor dem Klienten zu verbergen. Deshalb wurde in diesem Beispiel die Implementierung der Abstraktion direkt in die Basisklasse integriert. Diese Klasse *Tree* gibt durch ihre Schnittstellen nach außen nur sich selbst und den generischen Typen *E* preis und verheimlicht die Existenz des *INode* Typen. Die Ergänzung zum Klassendiagramm der Anwendung sind in Abbildung 5 einsehbar.

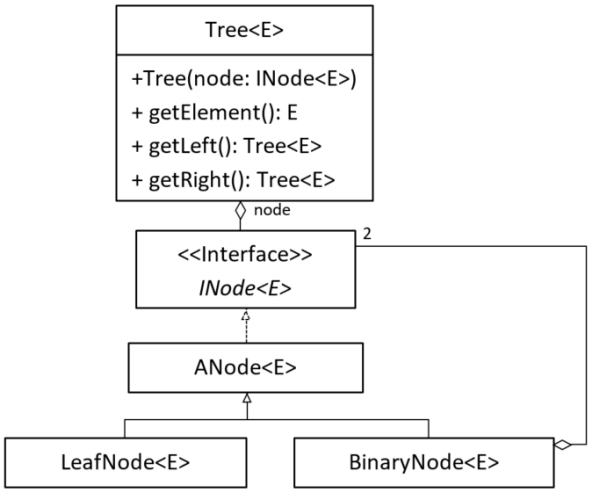

Abbildung 5. Bridge-Pattern in der Anwendung.

2.3 Iterator-Pattern

Das Iterator-Pattern gehört zu der Klasse der Behavioral Patterns. Wurde es realisiert, kann auf die Elemente einer Sammlung zugegriffen werden, ohne dass die zugrunde liegende Struktur der Sammlung offen gelegt werden muss. So werden Algorithmen von den unterschiedlich realisierten Sammlungen entkoppelt und können universell eingesetzt werden. Die C++ Standardbibliothek ist ein gutes Beispiel hierfür. Die bereitgestellten Funktionen um beispielsweise ein Element zu finden oder eine Sammlung zu kopieren sind alle mithilfe eines Iterators implementiert. Bei Verwendung dieses Patterns ist zu bedenken, dass je nach Implementierung des Iterators die Laufzeit und der Speicherverbrauch erhöht werden kann. Auch kann nicht immer sichergestellt werden, dass der Iterator die Änderungen in der Sammlung berücksichtigt. Hier unterscheidet man allgemein zwischen stabilen und nicht-stabilen Iteratoren. Stabile Iteratoren legen bei ihrer Erzeugung die Datenhierarchie der Sammlung in eine interne Repräsentation ab und arbeiten dann auf dieser Kopie, wohingegen nicht-stabile Iteratoren möglichst immer auf der eigentlichen Sammlung agieren. Abbildung 6 zeigt den Aufbau des Patterns.

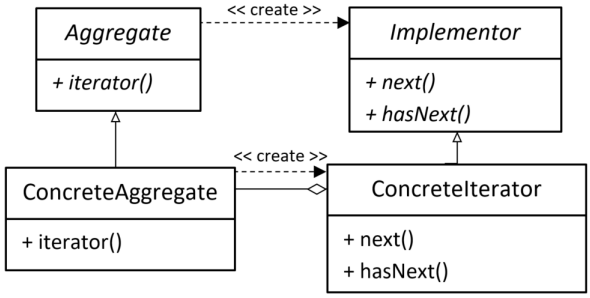

Abbildung 6. Iterator-Pattern.

In der Anwendung wurde das Aggregat in die *Tree* Klasse integriert, so dass über sie ein Iterator angefordert werden kann. Es wurden Iteratoren implementiert, welche den Baum in den vier Darstellungsarten In-Order, Pre-Order, Post-Order und Level-Order traversieren können und jeweils als stabile und nicht-stabile Version verfügbar sind. Das Klassendiagramm sieht mit der Erweiterung durch Iteratoren nun wie in Abbildung 7 gezeigt aus.

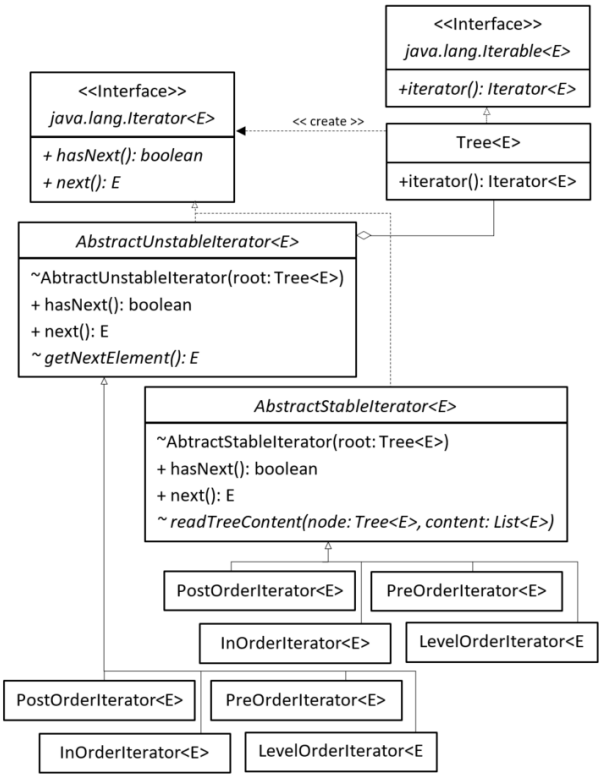

Abbildung 7. Iterator-Pattern in der Anwendung.

2.4 Strategy- und Factory-Pattern

Das Strategy-Pattern wurde schon fast komplett im vorherigen Abschnitt zum Iterator-Pattern in die Anwendung integriert. Es gehört zur Klasse der Behavioral Patterns und

wird eingesetzt, um einen Algorithmus (Strategie) auch zur Laufzeit austauschbar zu machen. So kann aus einer Familie von Algorithmen mit der selben Schnittstelle ein beliebiger ausgewählt werden, was die Flexibilität und die Wiederverwendbarkeit des Programmcodes erhöht. Durch Anwendung dieses Patterns können Mehrfachverzweigungen vermieden werden, was die Übersichtlichkeit des Codes fördert, und es stellt eine Alternative zur Unterklassenbildung der Kontexte dar. Allerdings erhöht das Strategy-Pattern die Anzahl der Objekte und bringt ein neues Problem mit sich, da der Klient die unterschiedlichen Strategien kennen muss, um zwischen ihnen zu wählen. Abbildung 8 zeigt den Aufbau des Patterns.

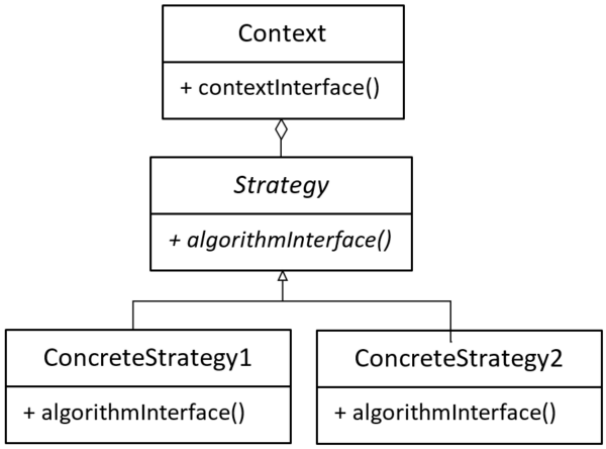

Abbildung 8. Strategy-Pattern.

In der Anwendung wären die Verschiedenen Strategien die Iteratoren des Baumes. Jedoch soll der Klient diese Iteratoren nicht selbst erzeugen müssen. Um dies zu erreichen wird das Factory-Pattern angewandt. Es gehört zur Klasse der Creational Patterns und wird verwendet um die Erzeugen von Objekten, welche gegebenenfalls sehr komplex sein kann, oder Schritte benötigt, welche vor dem Klienten verborgen werden sollen, zu übernehmen. Dies wird in der Anwendung erreicht, indem die *Tree* Klasse eine Methode bekommt, welche einen Enum-Typen erwartet, und den jeweiligen Iterator von der statischen Facotory-Methode in der *TreeIteration* Klasse erzeugen lässt und eine interne Referenz auf ihn speichert. Die *iterator()* Methode gibt nun eine Referenz auf eben diesen erzeugten Iterator zurück. Der *Strategy*-Enum-Typ besitzt für jeden Iterator einen Wert. Das neue Klassendiagramm der Anwendung wird in Abbildung 9 gezeigt.

2.5 Interpreter-Pattern

Das Interpreter-Pattern wird angewandt, um einen Ausdruck, welcher einer simplen Grammatik entspricht, zu interpretieren. Es wird oft zur Auswertung von regulären Ausdrücken und, wie hier zur Berechnung von logischen oder mathematischen Formeln angewandt. Hierbei wird das zuvor besprochene Composite-Pattern verwendet und zwischen terminalen und nicht-terminalen Ausdrücken unterschieden. Nicht-terminale Ausdrücke sind demnach aus weiteren Unterausdrücken zu-

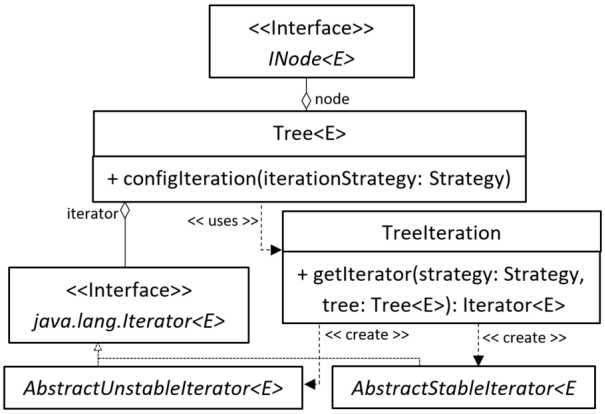

Abbildung 9. Strategy- und Factory-Pattern in der Anwendung.

sammengesetzt. Durch dieses Pattern bleib die Grammatik leicht Änderbar und Erweiterbar und lässt einen Satz durch ändern des Kontextes auf eine andere Art und Weise interpretieren. Werden die Grammatiken allerdings zu komplex, so ist von diesem Pattern abzuraten, da die Klassenhierarchie zu groß werden kann und die Effizienz in Mitleidenschaft gezogen wird. In solchen Fällen eignen sich Beispielweise Zustandsautomaten um die Interpretation zu implementieren. Das Klassendiagramm dieses Patterns ist in Abbildung 10 gezeigt. In der Anwendung wurden für die einzelnen Aus-

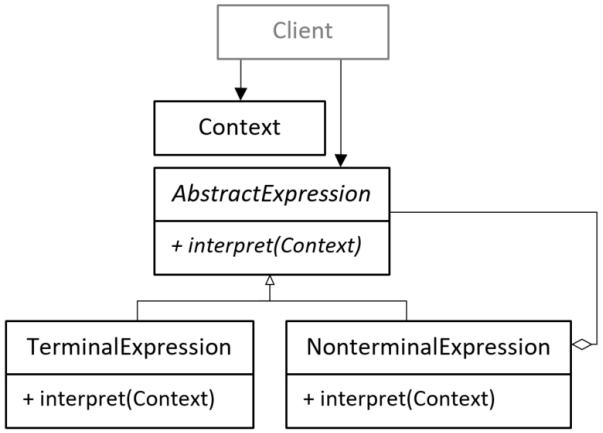

Abbildung 10. Interpreter-Pattern.

drücke keine neuen Klassen angelegt. Es wurde eine Klasse *ExpressionPrecedenceParser* implementiert, welche aus einem String-Array, welches die einzelnen Tokens des auszuwertenden Ausdrucks beinhaltet, die benötigte Baumstruktur aus den Knoten-Klassen, welche im Abschnitt Bridge-Pattern eingeführt wurden, erzeugt. Diese Erzeugung geschieht durch den rekursiven Aufruf der *readExpression(int)* Methode. Das Klassendiagramm dieser Klasse wird in Abbildung 11 dargestellt. Die angewandte Grammatik, welche in dieser Anwendung unterstützt wird, lässt sich folgendermaßen darstellen:

- expr := term | term op expr
- term := zahl | (expr)
- zahl := [0 - 9]*
- op := + | - | * | /

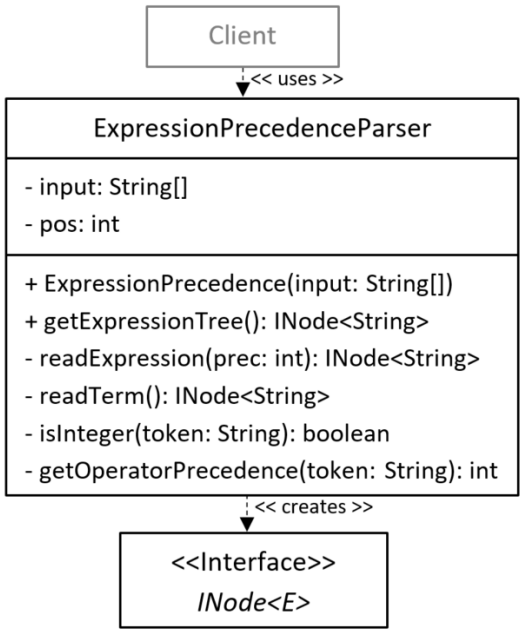

Abbildung 11. Interpreter-Pattern in der Anwendung.

2.6 Visitor-Pattern

Durch das Visitor-Pattern ist es möglich, Operationen auf die Elemente einer Objektstruktur zu implementieren, ohne dass die Klassen dieser Struktur verändert oder angepasst werden müssen. So kann zu einer unveränderlichen Hierarchie, welche für lange Zeit benötigt wird, über die Zeit neue Funktionalität hinzugefügt werden. Jede Operation, welche auf der Struktur ausgeführt werden soll, muss eine Methode für jedes Element überschreiben. In den einzelnen Elementen wird nur eine einzige Methode benötigt, welche die Basisklasse aller Besucher als Parameter erwartet. Wird diese Methode aufgerufen, löst sie einen so genannten Double-Dispatch aus. Hiermit ist gemeint, dass sie die Methode des Besuchers für jenes Element aufruft, dessen Klasse das Objekt der Methode angehört, und sich selbst als Parameter übergibt. In der Methode des Besuchers wird dann wieder auf die Methoden des Elements zugegriffen, um die eigentliche Operation zu implementieren. In Abbildung 12 wird das Klassendiagramm dieses Patterns gezeigt. Bei der Verwendung des Visitor-Patterns ist zu bedenken, dass es eine zyklische Abhängigkeit zwischen dem Besucher und den einzelnen Elementen in den Klassenentwurf einführt, was ein Zeichen eines schlechten Designs sein kann. Sollte trotz der eigentlich festen Klassenhierarchie dennoch mal ein neues Element hinzukommen, ist die Anpassung des bisherig vorhandenen Codes sehr aufwendig, da sämtliche Besucher angepasst werden müssen. Des Weitere kann dieses

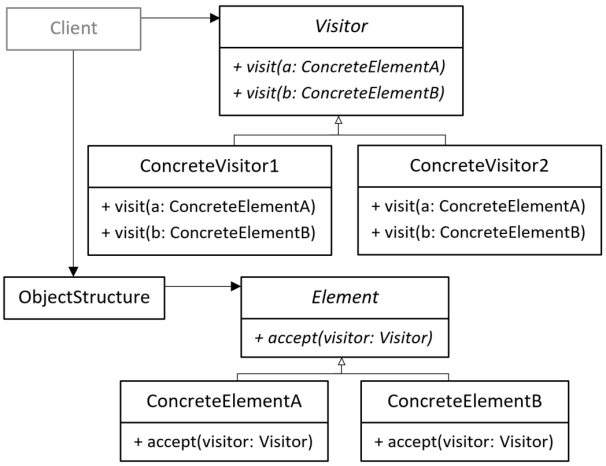

Abbildung 12. Visitor-Pattern.

Pattern einen Verstoß gegen das Geheimnis- bzw. Kapselungsprinzip mit sich führen, da es von außen auf die Elemente zugreift und in bestimmten Situationen so die Preisgebung gewisser Daten dieser Elemente erfordert, welche in anderen Fällen verborgen geblieben wären.
In der Anwendung wurden zwei konkrete Besucher implementiert. Beide implementieren das Interface *INodeVisitor*, welches jeweils eine Methode zur Behandlung der beiden konkreten Knotentypen vorschreibt. Es wurde ein Besucher zum Zählen der Operatoren und ein Besucher zum evaluieren des Ausdrucks entwickelt. Das Klassendiagramm dieses Patterns in der Anwendung wird in Abbildung 13 gezeigt.

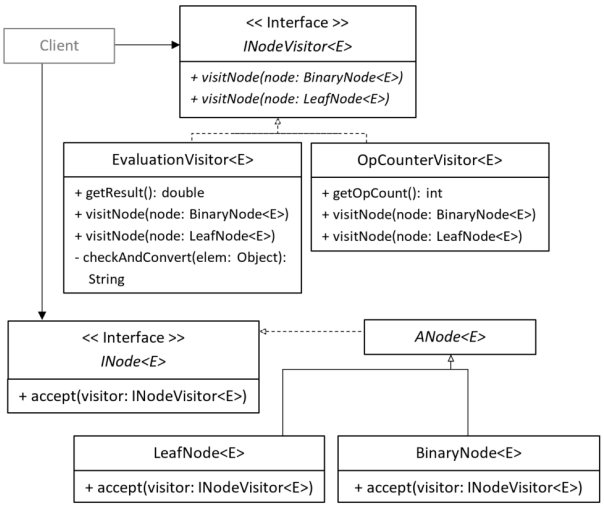

Abbildung 13. Visitor-Pattern in der Anwendung.

3. Erweiterung des Systems

Dieser Abschnitt befasst sich mit drei Erweiterungen, welche die Anwendung jeweils um eine neue Funktionalität erweitern. Hierbei wird auf die Änderungen im Code eingegangen.

3.1 Ergänzung der Potenzrechnung

Bisher wurden nur die vier Operationen Addition, Subtraktion, Multiplikation und Division von der Anwendung unterstützt. Die Potenzrechnung wird allerdings auch häufig verwendet und wäre im momentanen Zustand zwar auch realisierbar, jedoch wäre dies umständlich. Für die Potenzoperator wurde das Zeichen '^' gewählt. Da es sich hierbei um die Einführung eines neuen Operators handelt, muss der *Tokenizer* angepasst werden, welcher den Eingabe-String in ein Array formatiert und hierfür das neue Zeichen richtig erkennen muss. Hierzu wird das Zeichen zu der Abfrage auf die bisherigen Operatoren hinzugefügt, um ihn mit einem Leerzeichen von den Zahlen zu trennen. Dies ist in Listing 1 gezeigt. Ferner muss die *getOperatorPrecedence(String)* Methode des *ExpressionPrecedenceParsers* angepasst werden. Die Potenzrechnung bindet allgemein stärke als normale Multiplikation und Division und bekommt deshalb eine höhere Priorität.

```
public static String[] tokenize(String str)
{ ...
  char c = str.charAt(i);
  if ((c == '^') || (c == '+') || ...)
  {
    strBuilder.append(' ');
    strBuilder.append(c);
    strBuilder.append(' ');
  }
... }
```

Listing 1. Ergänzung im Tokenizer

Letztlich muss noch der *EvaluationVisitor* angepasst werden. In seiner Methode zur Behandlung von *BinaryNode* Objekten muss der neue Operator verarbeitet werden. Dies wird in Listing 2 abgebildet.

```
public void visitNode(BinaryNode<E> node)
{ ...
  double a = this.stack.pop().doubleValue();
  double b = this.stack.pop().doubleValue();

  if( operator.equals("^") == true )
  {
    this.stack.push(new Double(Math.pow(b,
        a)));
  }
... }
```

Listing 2. Ergänzung im EvaluationVisitor

Hiermit ist die Integration der Potenzrechnung in die Anwendung abgeschlossen.

3.2 Zulassen von Gleitkommazahlen

Durch das Zulassen von Gleitkommazahlen wird die Anwendung für viele neue Anwendungsgebiete interessant. Da bisher zur genauen Berechnung des Ergebnisses in der internen Struktur bereits nur mit dem Double-Typen gerechnet wurde, fällt die Implementierung dieser Erweiterung relativ leicht. Die verwendete Grammatik beschreibt einen Term als Zahl, oder als einen Ausdruck in Klammern. Deshalb hatte die *readTerm()* Methode des *ExpressionPrecedenceParsers* zuerst darauf geprüft, ob es sich um eine natürliche Zahl handelt, und anschließend, ob es sich um einen weiteren Ausdruck handelt. Das Prüfen auf eine natürliche Zahl wurde mit der Methode *isInteger(String)* getestet. In ihr wurde versucht den übergebenen String in einen Integer-Typen zu parsen. Käme es bei diesem Versuch zu einer Exception, wird diese abgefangen und *false* zurückgegeben. Andernfalls wird *true* zurückgereicht. Wird diese Methode durch die in Listing 3 gezeigte ersetzt, wurde die neue Funktionalität bereits implementiert.

```
private boolean isDouble(String token)
{
  try
  {
  Double.parseDouble(token);
  return true;
  }
  catch (NumberFormatException nfe)
  {
  }
  return false;
}
```

Listing 3. Ergänzung im EvaluationVisitor

3.3 Erweiterung auf ganze Zahlen

Um bisher einen Ausdruck mit negative Zahlen, wie zum Beispiel '$-4*(3-1)$', zu repräsentieren, musste dies über einen Umweg realisiert werden, da nur natürliche Zahlen zugelassen wurden. Der obige Ausdruck hätte in der folgenden Weise dargestellt werden müssen: '$(0-4)*(3-1)$'. Bei dieser Implementierung werden beide Vorzeichen (+ und -) berücksichtigt. Da der *Tokenizer* das jeweilige Vorzeichen als einzelnes Token in dem String-Array ablegt, muss dies im *ExpressionPrecedenceParser* berücksichtigt werden. Hierzu wird nach der Abfrage auf eine Gleitkommazahl, welche im Abschnitt 'Zulassen von Gleitkommazahlen' eingeführt wurde, und vor der Abfrage auf einen weiteren Ausdruck in Klammern eine neue Abfrage auf ein algebraisches Vorzeichen in der *readTerm(String)* Methode gemacht (Listing 4). Diese Abfrage gibt lediglich den Wert *true* zurück, wenn das übergebene Token ein '+' oder ein '-' ist und andernfalls *false*.

```
private INode<String> readTerm()
{
  INode<String> res = null;
  this.pos++; // next token
  ...

  if (this.isAlgebraicSign(input[this.pos]))
  {
    String value = input[this.pos]
          + input[this.pos + 1];
    res = new LeafNode<String>(value);
    this.pos++;
```

```
    }
... }
```

Listing 4. Ergänzung der Vorzeichenabfrage

Es ist zu beachten, dass innerhalb der IF-Anweisung *this.pos* erneut um eins erhöht wird, da bei diesem Schritt zwei Tokens verarbeitet werden.

4. Evaluierung

Die drei Erweiterungen des vorherigen Abschnittes haben gezeigt, dass die Anwendung ohne großen Aufwand um eine Reihe neuer Funktionen und Operationen erweitert werden kann. Dies wurde bei den hier gezeigten Beispielen hauptsächlich von dem implementierten Interpreter-Pattern im *ExpressionPrecedenceParser* ermöglicht. Lediglich bei dem Hinzufügen eines neuen Operators mussten andere Stellen der Anwendung angepasst werden, welche jedoch im *Tokenizer*, wie auch im *BinaryNode*, sehr klein ausgefallen sind und die anderen Bereiche der Anwendung nicht weiter beeinflusst haben. Dank dem Iterator-Pattern zusammen mit dem Strategy- und Factory-Pattern ist das Iterieren in verschiedenen Art und Weisen sehr einfach und weitere Iterationsmöglichkeiten können ergänzt werden, ohne dass der Rest der Anwendung berücksichtigt werden muss. Durch den Einsatz des Bridge-Patterns wird dem Klienten die zugrundeliegende interne Datenstruktur nicht bekannt und kann nach belieben ausgetauscht und verändert werden, ohne dass die Schnittstellen zur Anwendung angepasst werden müssen. Die verwendete Baumstruktur wurde aufgrund des Composite-Patterns einfach aufgebaut, allerdings ist zu bemerken, dass der durch dieses Pattern gewonnene Vorteil der leichten Ergänzung neuer Elemente durch das Visitor-Pattern verloren ging. Sollte ein neuer Knotentyp hinzukommen, müssten sämtliche konkreten Visitor Klassen abgepasst werden. Da diese Anwendung jedoch nur arithmetische Ausdrücke evaluiert, ist es nicht sehr wahrscheinlich, dass weitere Elemente hinzukommen werden. Demnach sollte dies hier nicht zu einem Problem werden.

Eine alternative zu der hier vorgestellten Anwendung, könnte ein Rekursiver Interpreter sein, welcher die einzelnen Tokens nicht in eine Datenstruktur schreibt, sondern sie direkt evaluiert. In diesem Fall gäbe es keine große Klassenhierarchie. Soll der Ausdruck jedoch nicht nur ausgewertet werden, sondern anschließend auch in verschiedenen Arten und Weisen ausgegeben werden, müsste er jedes mal neu interpretiert werden. Würde später eine Erweiterung, welche das Verwenden von Variablen erlaubt, implementiert werden, käme der Vorteil der Anwendung wie sie hier beschrieben wurde besonders zur Geltung. So könnte hier nach dem einmaligen interpretieren im *EvaluationVisitor* bei den Blattknoten geprüft werden, ob es sich um eine Zahl oder eine Variable handelt. Im Falle einer Variable, würde um Eingabe des Wertes gebeten, welcher anschließend zur Evaluation des Ausdrucks verwendet würde. Bei mehrmaligen Evaluieren, könnten so ohne weiteres interpretieren verschiedene Werte übergeben werden.

5. Zusammenfassung

Zusammenfassend bleibt zu sagen, dass das richtige Design Pattern dabei hilft, eine bestimmte Problemstellung in einem gewissen Kontext entsprechend eines Musters zu lösen. Es gibt verschiedene Klassen von Patterns, welche unterschiedliche Ziele haben. In dieser Arbeit wurden die Klassen der Creational Patterns, Structural Patterns und Behavioral Patterns kurz vorgestellt. Es wurden einige Patterns aus verschiedenen Klassen beschrieben und beispielhaft bei der Entwicklung einer Anwendung zur Auswertung arithmetischer Ausdrücke implementiert. Bei dieser Implementierung war zu sehen, dass nicht jedes verwendete Pattern nur Vorteile bringt und manchmal sogar die Vorteile eines anderen Ansatzes nichtig macht. Zudem wurde gezeigt, wie die Anwendung durch weiter Funktionalitäten erweitert werden kann, welches aufgrund der angewandten Design Patterns leicht und nur mit kleineren Änderungen im Code integriert werden konnten. In weiteren Arbeiten könnte die Verwendung des Extension-Objects-Patterns [2] untersucht werden. Es wird eingesetzt, wenn davon ausgegangen werden kann, dass die Schnittstelle eines Objekts in der Zukunft erweitert wird. Diese neuen Schnittstellen werden hierbei als sogenannt Extension-Objects realisiert.

Literatur

[1] Erich Gamma, Richard Helm, and Ralph Johnson ans John Vlissides. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[2] Erich Gamma. Extension object. In Robert C. Martin, Dirk Riehle, and Frank Buschmann, editors, *Pattern Languages of Program Design 3*, volume 3 of *Conference on Pattern Languages of Program Design (PLoP)w*, pages 79–88. Addison-Wesley Longman Publishing Co., Inc., 1997.

Erklärung zur Ausarbeitung

Hiermit erkläre ich, *Matthias Haselmaier (869995)*, dass ich die vorliegende Ausarbeitung selbstständig und ohne fremde Hilfe angefertigt habe und keine anderen als in der Abhandlung angegebenen Hilfen benutzt habe; dass ich die Übernahme wörtlicher Zitate aus der Literatur sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe. Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben kann.

_____

Unterschrift