

Dies ist keine „Musterlösung“, sondern eine gute von vielen möglichen Lösungen. Kommentare, die nicht Teil der Lösung sind, sind kursiv gesetzt.

Aufgabe 4-1:

1. Schwere Frage. Erstmal noch mal allgemein zum Unterschied zwischen Aggregation und Komposition:

Aggregation und Komposition sind Spezialfälle der allgemeinen Assoziation und sollen beide beschreiben, dass ein Ganzes aus Teilen zusammengesetzt ist. Wenn wir von Ganzen sprechen, werden die Teile dann in der Regel mitberücksichtigt.

Beispiel: Betrachten wir die Klassen Auto und Reifen. Sicherlich gibt es zu einem funktionstüchtigen PKW in der Regel 4 Reifen, welche Teil des Autos sind. Bringen wir ein Auto zur Werkstatt und lassen „das Auto“ überprüfen, erwarten wir, dass auch die Reifen Teil der Überprüfung sind. Gleichzeitig kann das Auto aber noch mit seinen Winterreifen assoziiert sein, welche momentan kein Teil von ihm sind und bei der Überprüfung nicht berücksichtigt werden.

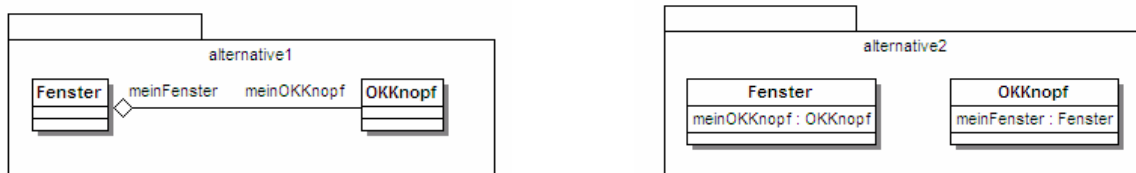
Aggregation und Komposition unterscheiden sich darin, dass bei einer Aggregation das enthaltene Objekt „eigenständig sinnvoll“ ist. Was sinnvoll bedeutet, hängt dann wieder ganz vom Problembereich ab, die UML macht hierzu keine Aussagen (UML Superstructure Specification, v2.1.1, S. 39):

Precise semantics of shared aggregation varies by application area and modeler.

Beispiel: Im Bezug auf echte Gegenstände könnte „eigenständig sinnvoll“ z.B. bedeuten „hat auch ohne ein Ganzes eine Funktion“, die Fahrradklingel als Teil des Fahrrads kann z.B. auch ohne das Fahrrad noch klingeln, was bei dem Bremsbelag nicht geht. Aber Achtung: Ein Fahrradverkäufer könnte dies durchaus anders sehen und sagen, auch einen einzelnen Bremsbelag kann man noch verkaufen.

Die übliche Interpretation von Komposition im Vergleich zu Aggregation in der Welt der Software ist, dass ein komponiertes Teil ohne Ganzes gelöscht wird, während ein aggregiertes Teil auch ohne Ganzes im Speicher/der Datenbank gehalten werden kann.

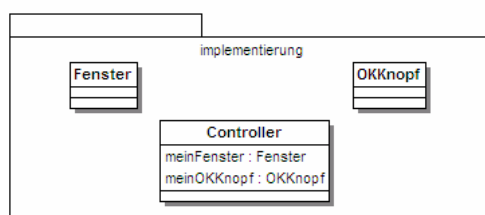
Um erstmal die Frage, was denn nun der Unterschied zwischen Aggregation/Komposition und einem Attribut ist, zu verstehen hilft vielleicht dieses Bild:



Gibt es zwischen alternative1 und alternative2 einen Unterschied? Wer jetzt zu sehr über Quellcode und Implementierung nachdenkt, ist auf der falschen Ebene und würde wahrscheinlich sagen Nein, kein Unterschied, denn Alternative 2 ist genau wie ich es in Java implementieren würde. Aber so wollen wir nicht darüber nachdenken, sondern lieber auf dem Niveau von Analyse bleiben.

Attribute haben auf der Analyseebene nämlich Wertcharakter und keine Objektidentität mehr. Man kann sie dann also nicht mehr wirklich als Objekte betrachten. Angenommen in der Klasse Person gäbe es das Integer-Attribut GrößeInCm. Niemand von euch käme auf die Idee, zu sagen: „Hier, nimm meine 180 aus GrößeInCm, Du bist doch auch so groß“, weil es kein persönliches 180 gibt, sondern nur den Wert. Entsprechend macht Alternative 2 auf Analyseebene keinen Sinn, denn wir könnten dann nicht sagen „Ein Fenster hat diesen Knopf“.

Noch als Abschließender Gedanke für alle die das mit der Analyseebene noch nicht so ganz geschluckt haben. Wenn wir auf die Implementierungsebene runter wollen, gibt es nämlich doch nicht nur die Alternative 2 als Möglichkeit, wie wir das in Quellcode umsetzen können, sondern noch eine ganz Menge mehr. Eine sinnvolle und ziemlich häufig vorkommende wäre z.B.



Nochmal zur **Syntax**: Attribute werden im mittleren Kompartiment einer Klassen angegeben, Aggregationen und Kompositionen mittels einer Verbindungsgerade gezeichnet, mit einer Raute auf Seite des Ganzen. Bei Kompositionen ist diese Raute ausgefüllt, bei Aggregationen leer.

2. Aktionen (action), die in den Zuständen angegeben werden, werden ausgeführt nach Betreten des Zustands (entry), vor Verlassen (exit) oder als Ergebnis einer internen Transition. Aktionen an Übergängen hingegen werden ausgeführt bei genau diesem Übergang, also noch vor einer entry-Aktion des Zielzustands, aber nach einer möglichen exit-Aktion des Quellzustands.

Aktionen sind „unmittelbar“ und ohne zeitliche Ausdehnung. Aktivitäten hingegen haben eine zeitliche Ausdehnung und können nur bei einem do innerhalb eines Zustands ausgeführt werden.

Die genaue Syntax ist aus dem Bild zu entnehmen:



3. Aktivitätsdiagramme verlangen keine Angabe von Objekten/Klassen und bieten nur unzureichende Möglichkeiten, den Einfluss von Akteuren darzustellen. Mit ihnen lassen sich gut nebenläufige Prozesse beschreiben – insbesondere deren Synchronisation (*Petrinetz-Ähnlichkeit, für die die verstehen was gemeint ist...*). Ein weiterer Vorteil ist die übersichtlichere Darstellung von Alternativen/Entscheidungen, die in Sequenzdiagrammen eher mühsam durch opt- und alt-Combined Fragments angegeben werden müssen.

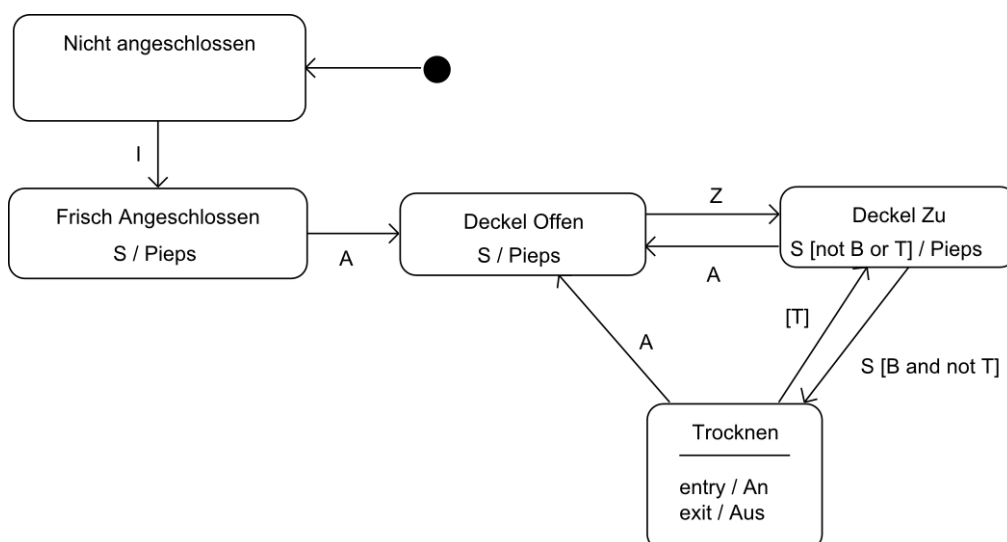
(Folgendes haben wir nicht in der Vorlesung oder im Tutorium besprochen: In Aktivitätsdiagrammen kann auch deutlich mehr Daten-/Objektfluss dargestellt werden, was das Diagramm aber recht kompliziert werden lässt. In Sequenzdiagrammen kann nur Datenfluss von Parametern und Rückgaben dargestellt werden)

Jetzt aber wieder zurück zur Frage (Wann welches Diagramm): Ein Sequenzdiagramm verwende ich dann, wenn ich einen Einblick in den zeitlichen Verlauf einer Interaktion zwischen mehreren Objekten geben will. Ein Aktivitätsdiagramm verwende ich dann, ich losgelöst von Objekten einen Ablauf in all seinen Alternativen/Entscheidungsmöglichkeiten darstellen möchte.

4. Keine, sie gleichen denen der Menschen als Akteure: Der Akteur als System muss außerhalb des zu bauenden Systems liegen, insbesondere haben wir keinen Einfluss auf dieses System. *Dies wird häufig falsch gemacht.*

Häufig falsch abgeschrieben: Akteure haben natürlich Einfluss auf das betrachtete System. Aber WIR als Entwickler haben keinen Einfluss auf die Akteure (Der Kunde macht was er will mit unserer Software)

Aufgabe 4-2: Diese Aufgabe war mal eine Klausuraufgabe.



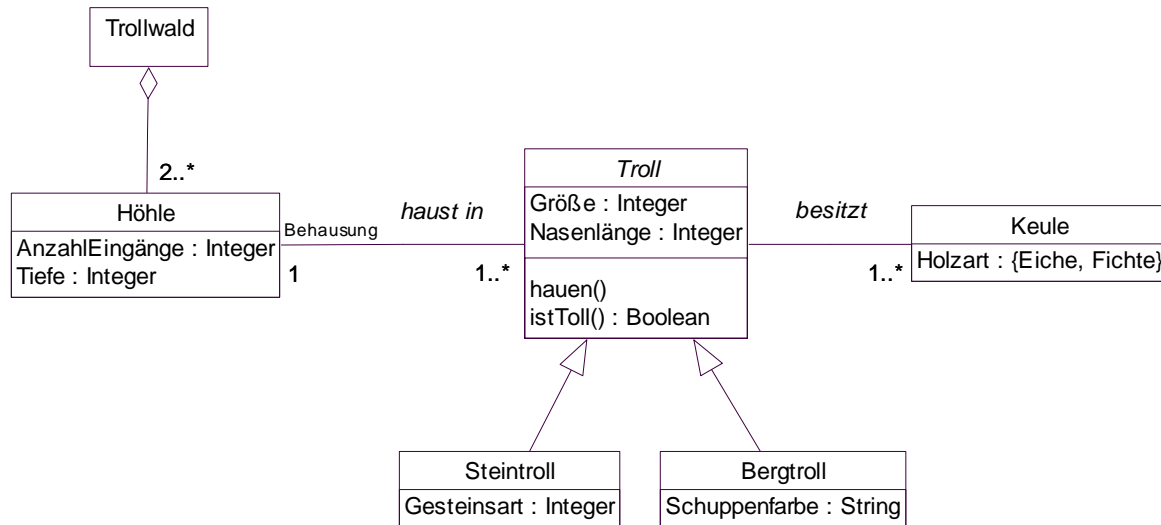
Im Zustand 1 wird davon ausgegangen, dass der Deckel geschlossen ist.

Es ist schlechter Stil (d.h. in der UML eigentlich nicht erlaubt) an den Übergang aus dem Startzustand etwas anders als eine Aktion zu schreiben (d.h. I als Ereignis sollte nicht am ersten Pfeil stehen).

Bei diesem Aufgabentyp ist es nicht nötig sich neue Ereignisse, Bedingungen oder Aktionen auszudenken.

Eine echte Schleuder wird übrigens niemals Ereignissensoren wie „auf machen“ und „zu machen“ haben (wie soll man das technisch machen?), sondern es können höchstens „offen“ und „geschlossen“-Zustände vorliegen.

Aufgabe 4-3:



Häufigste Fehler:

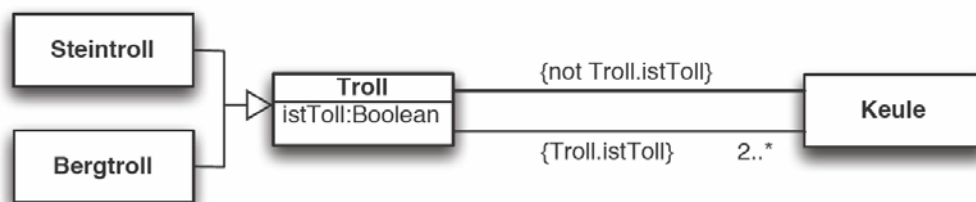
- Alle Sichtbarkeiten auf *private*
- Hauen, toll sein und den Trollwald nicht modelliert
- Troll nicht abstrakt
- Der Trollwald hat **viele** Höhlen (plural, d.h. mindestens 2)
- Hauen als Methode der Keule. Im Text stand aber doch „mit der er hauen kann“, d.h. der Troll haut, nicht die Keule. Wer sich in solchen Situationen unsicher ist, kann ja ruhig mal seine Begründung daneben schreiben. Das hilft dem Korrektor zu verstehen, wie es gemeint ist.

Mit Ausnahme der Holz-Typisierung, müssen die anderen Attribute keinen Typ haben, da er ja nicht klar angegeben wurde. Merke: Im Zweifelsfall unspezifiziert lassen.

Wer zwei Keulenunterklassen Eichenkeule und Fichtenholzkeule anstelle des Attributes *holzart* erstellt hat, soll mal in der Vorlesung fragen was welchen Vorteil hat.

Troll ist abstrakt (der Name ist kursiv), da es nur Stein- oder Bergtrolle gibt. (Auch das ist aber nicht klar gesagt.). Wer seine Diagramme per Hand malt, kann abstrakte Klassen durch Slashes vor und hinter dem Namen erkenntlich machen (informelle Notation von Christopher für die Klausur zugelassen): */Troll/*

Was ist mit „**Toller Troll**“? Man könnte es weglassen (schlecht), man kann es aber auch als Attribut in **Troll** rein nehmen oder wie hier als Methode *istToll() : Boolean*, denn es ist vielleicht berechenbar, z.B. aufgrund der Anzahl der Keulen. Leider fehlt bei allem die Verbindung zwischen Anzahl Keulen und Tollsein. Man könnte es wie folgt machen:



In *{ }* stehen Bedingungen für Beziehungen, sogenannte Diskriminatoren. Leider ist hier die Wirkungsrichtung falsch: Aus den Keulen muss sich die Tollheit ergeben, nicht wie hier andersherum. Richtig gut kann man das erst mit OCL modellieren, aber das kommt erst später in der Vorlesung. Ähnliches gilt für die im Text erwähnte Unterscheidbarkeit der Trolle anhand ihrer Attribute.

Überraschenderweise haben viele von euch die **Sichtbarkeiten** der Attribute falsch gewählt und z.B. die *Nasenlänge* und *Größe* auf *private* gesetzt, so dass man als **Bergtroll** und **Steintroll** gar nicht mehr auf die eigene *Nasenlänge* zugreifen kann. Wieso hinschreiben, wenn man es nicht weiß? Merke: Es gibt zwei Philosophien für die Vergabe von Sichtbarkeiten:

- **Defensiv:** Standardmäßig sind alle Attribute/Methoden *protected*, d.h. sie können nur von allen Klassen im selben Paket und Unterklassen verwendet werden. Alle anderen Sichtbarkeiten müssen begründet werden (Diese Methode ist *public*, weil es benötigt wird, damit die Klasse die Schnittstelle X implementiert; Dieses Attribut ist *private*, da so tief in die Funktionsweise der Klasse verstrickt ist, dass weder Änderungen erlaubt sein dürfen noch ein Benutzer der Klasse in irgendeiner Weise die Werte lesend verwenden darf)
- **Offensiv:** Standardmäßig sind alle Attribute/Methoden *public*, sie sind von allen Klassen sichtbar. Alle anderen Sichtbarkeiten müssen begründet werden (z.B. dieses Attribut ist nur *protected*, da ich zwar nicht weiß, was in Zukunft passieren wird, aber ein Benutzer meiner

Klasse direkt merkt, dass ich diese Methode erstmal nicht für Verwendung durch jeden geschrieben habe)

Aber was ist mit dem Geheimnisprinzip werdet ihr fragen. Das Geheimnisprinzip sollte man lieber durch Schnittstellen realisieren und nicht durch Sichtbarkeiten, damit schießt man sich nur in den Fuß. Also Philosophie wählen und das Begründen nicht vergessen (im Javadoc aufschreiben).

Übrigens: Nach einer übliche Namenskonvention fangen Namen von Attributen und Operationen mit einem Kleinbuchstaben an und diese Musterlösung könnte man also noch etwas besser machen...

Aufgabe 4-4:

Eine mögliche Meinung: Architekturmodelle (Komponentendiagramme oder Paketdiagramme) und Simple Zustandsdiagramme und Sequenzdiagramme werden vom Kunden verstanden, mehr nicht, d.h. der Rest sollte Text sein. Im Allgemeinen sind Klassendiagramme weder leicht verständlich, noch deren Sinn klar. Es sollte sich immer etwas bewegen: dynamische Diagramme ist das, was der Kunde will!