

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

**Aufgabe 8-1:**

1. Komponenten sind konkrete, ausführbare Programmteile, Entwurfsmuster und Architekturstile nicht. Entwurfsmuster sind im Vergleich zu Architekturstilen „kleiner“. Architekturstile betreffen vor allem die Zusammenarbeit von Komponenten (insbesondere im Hinblick auf nicht-funktionale Anforderungen), Entwurfsmuster die von Klassen und kleinen Modulen (insbesondere zur Erreichung von Funktionalität). Der Einsatz von Architekturstilen (Grobentwurf) liegt im Entwurfsprozess vor den Entwurfsmustern (Feinentwurf). Man vergleiche z.B. den Interpretierer, den es sowohl als Architekturstil gibt als auch als Entwurfsmuster gibt.

2. Beispiele: Druckerschlange, Protokolldatei, Systemzeit, Systemdesktop als Wurzel für alle Fenster, Standard-out-Stream.

Es ist ein (Objekt-) Erzeugungsmuster.

Welchen Sinn macht ein Einzelstück? a) Sicherstellen, dass es nur ein Objekt von der Klasse gibt, b) Möglichst spätes Erzeugen dieses Objektes (lazy instantiation), c) Erzeugung auch von nebenläufigen Threads möglich, d.h. es muss nicht festgelegt werden, welcher Thread das Objekt zuerst erzeugt (wobei man natürlich aufpassen muss, da das normale Singleton-Muster nicht Thread-Safe ist).

Nachteile: a.) Singleton erzeugt üblicherweise eine globale Abhängigkeit auf die Singleton-Methode. b.) Vielmals wird von übereifrigen Programmieren der Konstruktor der Klasse in der Sichtbarkeit eingeschränkt. Wenn dann doch zwei Exemplare in einer Virtual Machine benötigt werden, ist es häufig nicht möglich.

3. Alle vier sind Strukturmuster und dazu noch recht ähnlich in ihrer UML Form: Stets wird eine Schicht zwischen Aufrufer und tatsächlichem Zielobjekt gelegt.

Stellvertreter: macht einen Zugriff auf eine Implementation über die gleiche Abstraktion mittelbar über ein anderes Objekt (z.B. wegen Remote, Cache, Logging, Locking, Zugriffsrecht, etc.)

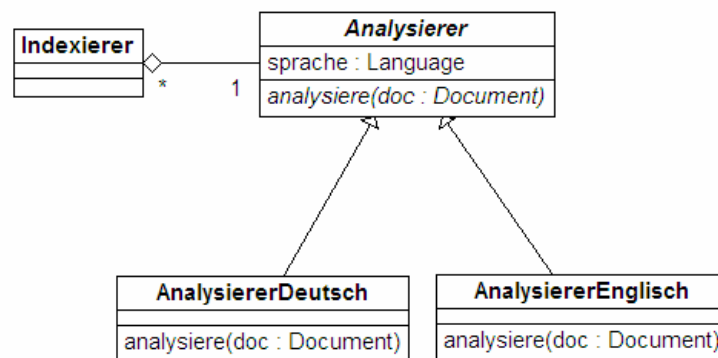
Adapter: passt eine vorgegebene Implementation an eine nicht direkt kompatible Abstraktion an.

Fassade: fasst mehrere Schnittstellen zu einer Schnittstelle zusammen.

Brücke: trennt Implementation von Abstraktion (=Interface), um beide variieren zu können.

**Aufgabe 8-2:**

1. Wir verwenden das Entwurfsmuster Strategie: Zu jeder Sprache existiert eine Analytikerklasse, welche alle Funktionen zum sprachabhängigen Analysieren kapselt.



Achtung: Die Frage, ob die Klasse **Analytiker** abstrakt ist, ist eine unabhängige Design-Entscheidung und hat erstmal nichts mit dem Entwurfsmuster **Strategie** zu tun. Es könnte z.B. sein, dass wir sie abstrakt machen, weil auf jeden Fall eine abgeleitete Klasse festlegen muss, welcher Zeichensatz verwendet wird. Andererseits könnte man auch argumentieren, dass die Klasse **Analytiker** eine sprachunabhängige Analyse durchführt, welche mit den Begriffen Satzzeichen und Stoppwörtern nichts anfangen kann und z.B. einen Index erstellt, der dann auch Punkte und Kommas als Worte enthält. Die sprachabhängigen Unterklassen können dann diese sprachunabhängigen Teile benutzen.

Ich den Aufgabentext bei 8-2-3 so interpretiert, dass ein **Indexierer** letztlich immer nur eine Sprache beherrschen kann und diese am Anfang festgelegt wird (deshalb auch im UML die 1 bei der Multiplizität des **Analytiker**).

Noch ein kleiner Test, eures Objekt-Orientierten Verständnisses. Welche der beiden Aussagen stimmt:

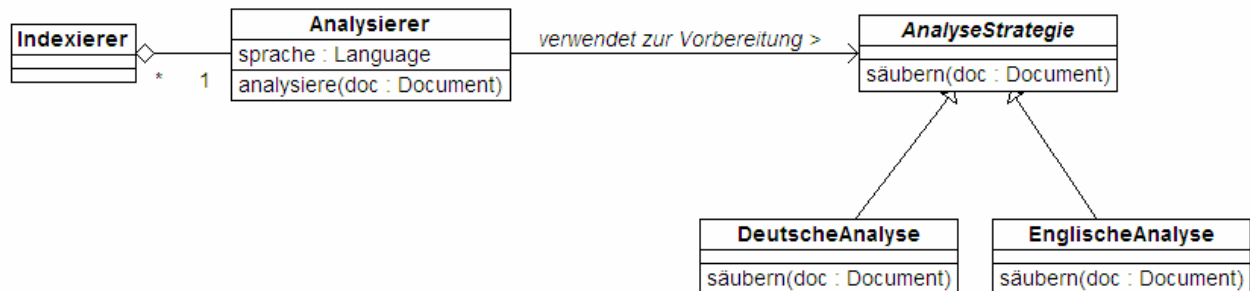
1. Während der Ausführung des Programms gibt es immer ein **Indexierer**-Objekt, das ein **Analytiker**-Objekt hat, welches wieder zwei Unteranalysierer-Objekte (ein **AnalytikerDeutsch**- und ein **AnalytikerEnglisch**-Objekt) hat an die er Aufgaben delegieren kann.
2. Wenn es während der Ausführung des Programms einen **Indexierer** gibt, dann hat dieser entweder einen **AnalytikerDeutsch** oder einen **AnalytikerEnglisch**.

Die zweite Antwort ist richtig nach obigen Klassendiagramm und die erste hat eine ganze Menge Fehler:

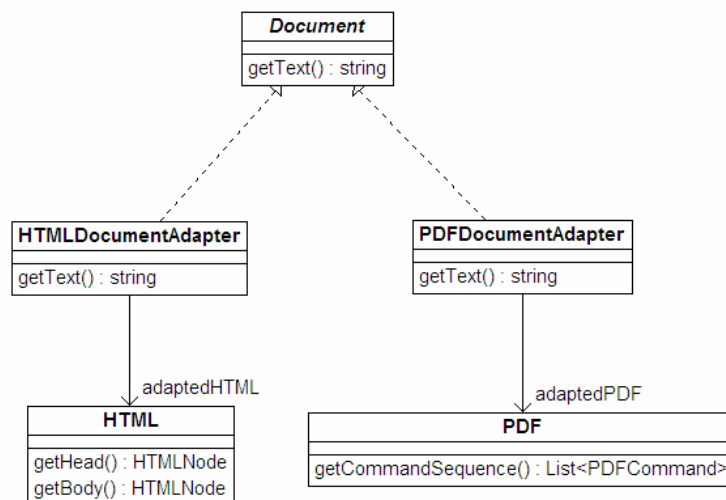
- Das Klassendiagramm sagt NICHT, WIEVIELE Indexierer-Objekte es geben kann, sondern nur, was für JEDES Indexierer-Objekt gilt.
- Es kann keine reinen Analysierer-Objekte geben, denn die Klasse Analysierer ist abstrakt.
- Auch gibt es keine Beziehungen zwischen Objekte der Oberklasse und der Unterklasse. Delegieren kann ich deshalb weder an meine Ober- noch Unterklasse, ich kann nur meine von meinen Oberklassen geerbten Methoden verwenden.

Achtung 2: Ein typischer Fehler von euch war auch, dass ihr zwar einen abstrakten Analysierer mit einer Methode analysiere habt, aber die Methoden in den Unterklassen nicht mehr ersetzt. Wie soll das funktionieren?

Ganz zum Schluß noch eine Lösung von einem eurer Kommilitonen, die mir gut gefallen hat, und die besser klar macht, dass der Analysierer vielleicht aus Teilen besteht, die allgemein sind und solchen die sprachabhängig sind:



2. Wir verwenden das Entwurfsmuster **Adapter**: Zu jeder Klasse, die wir an die Document-Schnittstelle anpassen wollen, schreiben wir eine Adapterklasse, welche diese Schnittstelle implementiert und Aufrufe so umwandeln kann, dass sie zur Ausgangsklasse passen.



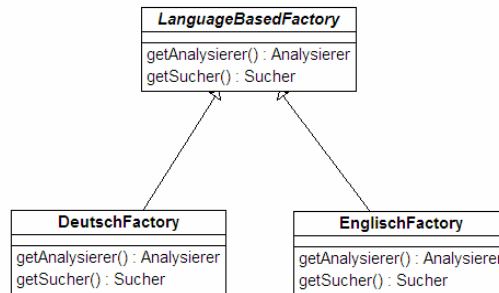
Wird zum Beispiel dem Analysierer ein **HTMLDocumentAdapter** als zu analysierendes Dokument übergeben und ruft der Analysierer `getText()` auf, dann wird der Adapter vielleicht den Body in reinen Text umwandeln und diesen als Text ohne HTML-Tags zurückgeben.

Ohne das Entwurfsmuster wäre man vielleicht auf die Idee gekommen, dem Analysierer beizubringen auch Objekte der Klasse **HTML** und **PDF** analysieren zu können. Dies macht den Analysierer aber zu einer Klasse, die unnötig groß, unübersichtlich und damit letztlich schlechter wartbar und erweiterbar ist.

Achtung: Es entspricht nicht dem Entwurfsmuster Adapter, wenn ein Objekt (z.B. ein **HTML**-Objekt) komplett in ein passendes Objekt (z.B. eins der Klasse **Dokument**) umgewandelt und dann dieses übergeben wird. Der Adapter biegt Aufrufe um, um damit eine andere Schnittstelle vorzugaukeln, er ist kein Konverter.

Achtung2: Es ist nicht Aufgabe des Adapters das adaptierte Objekt zu erstellen. In unserem Fall hat der **PDFDocumentAdapter** keine Ahnung wie man ein **PDF**-Objekte erzeugt.

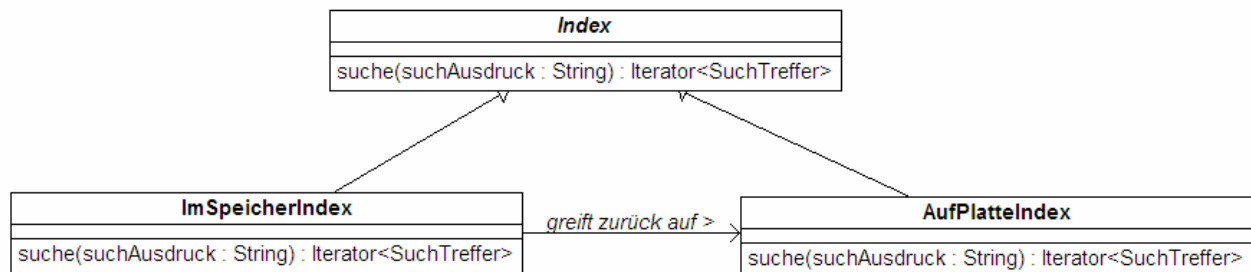
3. Wir verwenden das Entwurfsmuster **Fabrik** und verlagert in eine solche Fabrik alle Logik zum Erstellen der Sucher und Analysierer. Die Fabrik ist dann zuständig dafür zu sorgen, dass Sucher und Analysierer zueinander passen, z.B. könnte immer ein Sucher zurückgegeben werden, der den gleichen Analysierer verwendet, den man auch über `getAnalysierer` erhält.



Achtung: Es geht bei der **Fabrik** nicht unbedingt darum, dass der Aufrufer nicht weiß, was für Objekte er bekommt (dies ist das Ziel bei der **Abstrakten Fabrik**), sondern nur, dass die Objekte aus der Fabrik zueinander passen.

Achtung: Builder passt NICHT. Denn ein Builder baut immer nur ein unabhängiges aber kompliziertes/zusammengesetztes Objekt durch mehrere Aufrufe und keine Objekte einer Familie von Objekten.

4. Wir verwenden das Entwurfsmuster **Stellvertreter (Proxy)**. Wir führen einen Stellvertreter für den Index auf der Festplatte ein, welcher Daten im Hauptspeicher hält und nur dann auf den Index auf der Platte zu greift, wenn er die angeforderten Ergebnisse/Teile des Index nicht im Speicher hat.



Es gibt ein paar ähnliche Muster, die aber nicht passen:

- Ein **Adapter** passt Schnittstellen an, aber AufPlatteIndex ist ja schon selbst ein Index.
- Eine **Facade** verbirgt komplexe Schnittstellen hinter einer einfacheren Klasse, aber ImSpeicherIndex hat keine einfachere Schnittstelle.
- Ein **Dekorierer** fügt den zurückgegebenen Daten einer Schnittstelle etwas hinzu (oder filtert diese), der ImSpeicherIndex sollte aber die gleichen Ergebnisse wie der AufPlatteIndex liefern und nichts an diesen ändern.

Achtung: "Der Proxy verbirgt den Zugriff auf den Speicher" - Nein! Die Schnittstelle Index verhindert, dass der Aufrufer weiß, wo die Daten herkommen. Der Proxy hat damit nichts zu tun.

5. Ablage ist der Index selbst, der bei neu hinzukommenden Dokumenten inkrementell erweitert wird und nicht komplett neu erstellt wird. Ein Datenflussnetz würde in seiner reinen Form immer alle Dokumente neu einlesen und den Index komplett neu erzeugen. Es ist also eine ablagebasierte Architektur angesagt. Aber für Teilkomponenten, z.B. den Analysierer, kann man Datenflussnetz-artig mehrere hintereinander geschaltete Tokenizer/Filter einsetzen, die ein neues Dokument analysieren.

6. Eine schon fertige Komponente suchen und „kaufen“, die die Anforderung erfüllt. Bloß nicht selbst schreiben! In Java ist dies z.B. Lucene (<http://jakarta.apache.org/lucene/>), von dem auch der Entwurf geklaut wurde. ☺

**Aufgabe 8-3:** Eine ähnliche Aufgabe war mal Klausuraufgabe.

- Kompositum (composite): JMenu ist hier der Behälter/Kompositum (composite), zu erkennen z.B. an der Beschreibung der add-Methode für JMenuItem, der Oberklasse von Menu.
- Beobachter (observer): JMenu ist das (beobachtete) Subjekt (subject), erkennbar an der addMenuListener-Methode
- Strategie (strategy): JMenu hat die Strategie für das Austauschen des Look&Feel und ruft sie auf, erkennbar am setUi() in JMenuItem und dem ComponentUI-Argument.
- „Halbes“ Kommando (commando): Menu hat ein add(Action), wobei ein Action das Kommando (Command) ist. JMenu hat hier die Rolle des Invoker, allerdings wird das Aufrufen des Kommandos an die ActionListener (addActionListener in AbstractButton) übergeben. Dies ist etwas schwer zu erkennen