

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

Aufgabe 7-1:

1.a.

Achtung: Mit Schnittstelle in SWT normalerweise NICHT ein Java-Interface gemeint.

Die Signatur einer Methode definiert deren formale Parameter, den Rückgabotyp und in Java die Menge an möglichen Checked-Exceptions, die von der Methode ausgelöst werden können. Eine Signatur ist damit etwas, was grobe Regeln für die Eingabe und Ausgabe einer Methode festlegt.

Zur Schnittstelle einer Methode gehört aber auch noch alles andere was ein Aufrufer wissen sollte: z.B. zulässige Werte für die Parameter, Vorbedingungen, Zusicherung, Nebenläufigkeitseigenschaften, Fehlerbehandlung, Nicht-Funktionale Eigenschaften allgemein, etc.

Man könnte sagen: Schnittstelle = Signatur + Vertrag (Contract)

Die Signatur einer Methode enthält also nur so viele Informationen, dass der Compiler überprüfen kann, ob eine Methode richtig verwendet wurde (z.B. passen die Typen der Parameter). Die Schnittstelle einer Methode enthält (hoffentlich) so viele Informationen, dass der Programmierer, der die Methode aufrufen möchte, verstehen kann, was die Methode macht und von ihm erwartet.

Ein Interface in Java entspricht der Methoden-Signatur auf Klassenebene und sollte nicht mit dem Begriff Schnittstelle verwechselt werden: Man definiert, welche Methoden mit welchen Signaturen in einer Klasse existieren müssen, wenn diese Klasse dem Java-Interface entsprechen will. Ein Java-Interface ist also wieder nur ein Konstrukt für den Compiler. Um ein Java-Interface zu einer Schnittstelle werden zu lassen, muss man sie wie die Methoden-Schnittstellen erheblich um Informationen anreichern. Der Begriff Klassensignatur ist unüblich und man würde ihn eher so verstehen, dass er auf die Typverträglichkeit der Klasse hinweist.

Eine Modulschnittstelle/Komponentenschnittstelle geht dann noch eine Ebene höher (siehe auch 1.b. Unterschied Klasse Komponente) und lässt sich z.B. in Java nicht mehr gut durch die Sprache abbilden (Packages helfen nur bedingt). Eine Modulschnittstelle umfasst alle Klassen, Interfaces die für den Nutzer einer Komponente wichtig sind UND die Informationen, wie man diese verwendet.

Der Begriff Benutzerschnittstelle (graphical user interface GUI) hat mit all diesen Begriffen eigentlich nichts zu tun.

1.b. Eine Komponente (auch häufig Modul) ist ein funktionales, wieder verwendbares, nutzbares Teilsystem, während eine Klasse meist alleine keinen Sinn macht. In ganz seltenen Fällen besteht eine Komponente nur aus einer Klasse oder nur aus einer Methode, meistens ist sie umfangreicher.

1.c. Kohäsion beschreibt den Grad des Zusammenhalts der Teile eines Moduls. Der Zusammenhalt definiert sich aus der Anzahl der Benutzungen/Aufrufe der Teile untereinander. Eine hohe Kohäsion spricht für einen hohen Grad an Gemeinsamkeit (das sog. Geheimnis, *concern* oder Entscheidung) der Teile. Die Kopplung ist die Außenbeziehung des Moduls zu anderen Modulen, also die Nutzung der Module untereinander. Eine niedrige Kopplung spricht für einen hohen Wiederverwendungsgrad. Eine Modularisierung ist – vereinfacht gesagt – dann gut, wenn die Kohäsion pro Modul hoch und die Kopplung der Module niedrig ist.

2.

Achtung: Ein Entwurfsmuster ist keine Architektur (es kann aber dabei helfen eine Architektur umzusetzen, z.B. kann das Entwurfsmuster Facade verwendet werden um zwei Komponenten sauber zu trennen).

Achtung 2: In der Vorlesung wurden die Begriffe Architektur (die Struktur eines Systems) und Architekturstile (Prinzip wie man ein System/Teile strukturieren kann) unterschieden.

- *Schichten*: KVV mit den Schichten Datenbank, Serverlogik und Oberfläche. Die Telekommunikationsschichten des ISO/OSI-Modells sind kein so gutes Beispiel, da das Modell ja keine Softwaresystem ist, sondern nur ein Modell. Eine Implementierung dieses Modells in einem TCP/IP-Stack (z.B. für Linux) auf der anderen Seite verwendet vielleicht aus Performancegründen überhaupt keine Schichtenarchitektur.
- *Datenflussnetze*:
 - o Reportgenerator, z.B. zum Ausdrucken von Serien-E-mails: Adressdatenaufbereitung -> Abfragen fehlender Werte -> Briefgenerierung -> Abschicken.
 - o Übersetzer-Werkzeugkette: Parser (besteht aus Unterschritten Lexikalische Analyse und Syntaktische Analyse) -> Compiler -> Linker
- *Objektnetze*: Steckt in jedem objektorientierten Programm...

- *Ereignissteuerung*: In Benutzungsoberflächen, z.B. auf Basis von Swing/AWT. Das Abholen von Email ist **nicht** ereignisgesteuert.
- *Ablagebasierte Struktur*: Alle Datenbank Anwendungen, mit mehr als einem gleichzeitig aktiven Klient. Aber wahrscheinlich auch jeder Editor (z.B. ein Malprogramm bei dem das Bild in der Ablage gespeichert wird und alle Befehle auf diese zugreifen).
- *Interpretierer*: Jede Tabellenkalkulation erlaubt Formeln einzugeben, welche dann interpretiert werden.

Ein Softwaresystem kann viele solcher Stile enthalten. Sie ergänzen sich. Man kann bei jedem gegebenen Beispiel also nachfragen, welche Stile ansonsten noch dort drin stecken.

3.a. Echtzeit => Ereignisbasiert (z.B. Insulinpumpe). *Eigentlich ist hier Unterbrechungsbasiert angesagt, da ein Ereignis meist sofort bearbeitet werden muss, d.h. es wird die aktuelle Ausführung unterbrochen.*

3.b. Portabilität => Schichtenbasiert, weil sich die Schichten austauschen lassen. Beispiele wären z.B. die Verwendung einer anderen SQL-Datenbank (wer das schon mal gemacht hat, weiß dass es nicht ganz so leicht ist) oder z.B. alle Java-Anwendungen: Die Java-Engine ist eine Schicht über dem Betriebssystem, die für jedes Betriebssystem ausgetauscht wird und damit die Schichten darüber ohne Änderung weiterlaufen können.

3.c. Speicher => Objektnetz, weil ich hier den Zugriff auf Daten sehr gut optimiert kann. Ich verliere hierbei natürlich z.B. die Kapslung, die ich einer Schichtenarchitektur hatte, oder bessere Nachvollziehbarkeit des Programmablaufs einer Ereignissteuerung.

Eine Ablagebasierte Architektur kann auch zur Optimierung des Speicherplatzes dienen, weil hier alle auf einer gemeinsamen Datenbasis operieren.

Aufgabe 7-2:

1. Der Text bis „Eine solche Volltextsuche...“ ist Anforderungsbeschreibung, der nächste Satz + Auflistung (die Trennung Indizieren/Suchen und der Schlagwortindex) ist Entwurf und der letzte Satz ist beides: Die Anforderung ist es „in verschiedenen Sprachen getrennt suchen zu können“; Entwurfsentscheidung ist es, deshalb „mehrere Indizes verwalten zu können“. Eine nicht-funktionale Anforderung steckt in „effizient“.

2. Gemeint sind nicht Java-Interfaces, siehe 1.a. Methodensignaturen sind o.k., aber man muss auch dazu schreiben, was die Methoden machen. Es folgt eine simple Lösung. Es ist nicht zwingend nötig, hier mit Vorbedingungen und Zusicherungen zu kommen – wäre natürlich schön!

```
// Erzeugt leeren Index, der mit search() durchsucht werden kann
Index createIndex ()

// Löscht einen Index. i wurde mit createIndex() erzeugt
void deleteIndex (Index i)

// Fügt ein neues Dokument zu einem Index hinzu. Der Index kann // leer sein, aber nicht null.
Zurückgegeben wird eine ID, die
// das indexierte Dokument referenziert. Diese ID wird bei
// search() zurückgeliefert.
DocumentID addDocumentToIndex (Document d, Index i)

// Liefert alle DocumentIDs des Index i zurück, in deren
// zugehörigem Dokument d (siehe addDocumentToIndex) der Suchtext
// s vorkommt, vorausgesetzt, vorher wurde
// addDocumentToIndex(d,i) aufgerufen.
DocumentIDList search (String s, Index i)
```

3. siehe folgendes Komponentendiagramm

