**Software-Technik**

**Prüfungsvorbereitung 2023**

*Für den klassischen Last-Minute-Lückenlerner!*

1. UML
2. Datenbanken (SQL)
3. Entwurfsmuster
4. Testen

Inhaltsverzeichnis

[1 UML 4](#_Toc133324616)

[1.1 Strukturdiagramme 4](#_Toc133324617)

[1.1.1 Klassendiagramm 4](#_Toc133324618)

[1.1.2 Kardinalitäten 5](#_Toc133324619)

[1.2 Verhaltensdiagramme 5](#_Toc133324620)

[1.2.1 Anwendungsfalldiagramm (Use-Case) 5](#_Toc133324621)

[1.2.2 Aktivitätsdiagramm 5](#_Toc133324622)

[1.2.3 Sequenzdiagramm 6](#_Toc133324623)

[1.3 Datenbanken 6](#_Toc133324624)

[1.3.1 Normalformen 6](#_Toc133324625)

[1.3.2 ERM 7](#_Toc133324626)

[1.3.3 RDM 7](#_Toc133324627)

[2 SQL 8](#_Toc133324628)

[2.1 Einfache SQL-Abfragen 8](#_Toc133324629)

[2.1.1 SELECT 8](#_Toc133324630)

[2.1.2 CREATE TABLE 8](#_Toc133324631)

[2.1.3 INSERT 8](#_Toc133324632)

[2.1.4 UPDATE 8](#_Toc133324633)

[2.2 PL/SQL 8](#_Toc133324634)

[2.2.1 Stored Procedures 9](#_Toc133324635)

[2.2.2 Functions 9](#_Toc133324636)

[2.2.3 Weitere Syntax 9](#_Toc133324637)

[3 Design-Pattern (Entwurfsmuster) 11](#_Toc133324638)

[3.1 Singleton 11](#_Toc133324639)

[3.2 Fabrik-Methode 11](#_Toc133324640)

[3.2.1 Simple-Factory 12](#_Toc133324641)

[3.2.2 Abstract-Factory 12](#_Toc133324642)

[3.3 Composite 13](#_Toc133324643)

[3.4 Observer 14](#_Toc133324644)

[4 Testen 15](#_Toc133324645)

[4.1 Grundsätze 15](#_Toc133324646)

[4.2 Voraussetzungen 15](#_Toc133324647)

[4.3 Blackbox-Tests 15](#_Toc133324648)

[4.4 Äquivalenzklassen 15](#_Toc133324649)

[4.5 Whitebox-Tests 16](#_Toc133324650)

[4.5.1 Anweisungsüberdeckung 16](#_Toc133324651)

[4.5.2 Zweigüberdeckung 16](#_Toc133324652)

[4.5.3 Termüberdeckung 16](#_Toc133324653)

[5 Eigene Notizen 17](#_Toc133324654)

# UML

**U**nified **M**odeling **L**anguage: Vereinheitlichte Modelle, um die reale Welt zu repräsentieren, als Kommunikationsbasis für alle Beteiligten bei Analyse, Design, Entwicklung und Wartung.

**Vorteile:**

* Eindeutigkeit
* Verständlichkeit
* Ausdrucksstärke
* Standardisierung
* Plattform- und Sprachunabhängig

## Strukturdiagramme

### Klassendiagramm

Ein normales Klassendiagramm beinhaltet alle im System benötigten Klassen und gibt Auskunft, über Attribute, Vererbungen, Beziehungen zwischen Entitäten und deren Kardinalitäten. Es kann direkt aus dem ERM oder RDM abgeleitet werden, nachdem Beziehungen aufgedröselt wurden.

Ein gutes Vorgehen ist es zuerst skizzenhaft das ERM zu zeichnen und Klassen geschickt zu ordnen, damit Beziehungen später übersichtlich dargestellt werden können. Danach fängt man mit den Klassen, den Attributen und Methoden an und fügt erst zum Schluss die Beziehungen und die Kardinalitäten ein. So kommt man am wenigsten durcheinander.

|  |  |
| --- | --- |
| + attribut: Datentyp | Public Attribut *(gilt auch für Methoden)* |
| - attribut: Datentyp | Private Attribut *(gilt auch für Methoden)* |
| + attribut: Datentyp | Statisches Attribut *(gilt auch für Methoden)* |
| <<abstract>> KlassenName | Abstrakte Klasse |
| <<interface>> InterfaceName | Interface |
|  | Implements-Pfeil (für Interfaces) |
|  | Extends-Pfeil (Für Vererbung) |
|  | Assoziationspfeil mit Kardinalität |

### Kardinalitäten

Um anzugeben wie viele Objekte einer anderen Entität angehören können, muss man eine Kardinalität nach Chen einfügen. Das gilt übrigens auch für das ERM. Wichtig ist dabei die Leserichtung!

|  |  |
| --- | --- |
| 0…\* | Beliebig viele Assoziationen |
| 1…\* | Mindestens eine Assoziation |
| 1…3 | Mindestens eine, höchstens 3 Assoziationen |
| 1 | Genau eine Assoziation |

## Verhaltensdiagramme

Grundsätzlich gilt, dass man bei allen Anwendungsfalldiagrammen einen großen Kasten darumziehen und ein „sinnvollen“ Namen für das System vergeben muss.

### Anwendungsfalldiagramm (Use-Case)

Ist ein Schmarrn! 🡪 Aus Risiko weggelassen!

### Aktivitätsdiagramm

* Alternative Abläufe
* Reihenfolge von Aktivitäten
* Parallele Aktivitäten
* Verschachtelte Aktivitäten
* Verantwortungsbereiche
* Ausnahmen und deren Behandlung

**Komponenten:**

Man kann einzelne kleinere Kasten einführen, um zusammengehörende Funktionalitäten zusammenzufassen. Aus diesen Unterkästen kann man dann mit Notizen escapen, die neue Zweige initiieren.

|  |  |
| --- | --- |
|  | Startknoten |
|  | Endknoten |
|  | Ende eines Zweigs |
|  | Verzweigung - Beide Fälle müssen explizit angegeben werden |
|  | Parallele Aktivitäten, die wieder zu einem Zweig zusammenführen |
|  | **Pins:** Symbolisieren Ein- und Ausgangsparameter (Objekte) |
|  | Mit einer Notiz kann man innerhalb eines Kastens eine Escape-Condition beschreiben und damit einen Zweig initiieren, der zu jeder Zeit getriggert werden kann |

### Sequenzdiagramm

Ist absolut dreckig! 🡪 Aus Risiko weggelassen!

## Datenbanken

Bei Datenbanken gibt es zwei verschiedene Modelle: Das Entity Relation Model (ERM) und das Relationale Datenbankmodell (RDM). Beide können nur erstellt werden, wenn die dritte Normalform gewährleistet ist. Dann folgt das RDM aus dem ERM und andersrum – nachdem wie die Aufgabe gestellt ist.

### Normalformen

|  |  |
| --- | --- |
| NF 1 | Atomisieren |
| NF 2 | Jedes nicht-primäre Attribut ist jeweils von allen Primär-Schlüsseln abhängig |
| NF 3 | Redundanzen werden vollständig entfernt |

### ERM

Ein Entity-Relation-Model gibt Auskunft über Entitäten (Tabellen einer Datenbank), deren Beziehungen zu anderen Entitäten und Kardinalitäten an. Zusätzlich werden Attribute sowie Primärschlüssel angegeben.

|  |  |
| --- | --- |
|  | Entitätsnamen müssen immer sinnvoll und im **Singular!!** benannt werden |
|  | Reguläres Attribut einer Klasse. Normalerweise mit einem private Modifier |
|  | Primärschlüssel einer Entität. In der Regel eine Nummer |
|  | Beziehung zwischen zwei Entitäten. Hat immer ein sinnvoll beschreibendes Wort und wird nur selten bis gar nicht mit Wörtern wie „ist“ oder „hat“ betitelt.  Auch Beziehungen können (nur bei M:N) Attribute haben. |
|  | Vererbung – Dreieck zeigt immer auf die Elternklasse |

### RDM

M:N-Beziehungen werden im ERM-Diagramm in einer Beziehungsraute angezeigt. Im RDM muss man diese allerdings aufdröseln und daraus eine separate Tabelle erstellen. Der Name setzt sich aus den beiden Entitäten zusammen und der Primärschlüssel ist eine Kombination beider Primärschlüssel.

Primärschlüssel werden unterstrichen, Fremdschlüssel gestrichelt unterstrichen **(schwer zu machen in Word, deshalb hier kursiv)**.

EntitaetsName(Primärschlüssel, Attribut1, Attribut2, …, *Fremdschlüssel1, …*)

Bei M:N-Beziehungen werden beide Primärschlüsse zuerst gestrichelt und dann nochmal solide unterstrichen **(hier unterstrichen und kursiv)**.

Ent1\_Ent2(*Prim1, Prim2*, …)

# SQL

## Einfache SQL-Abfragen

### SELECT

**SELECT** col1, col2 **FROM** table\_name;

### CREATE TABLE

**CREATE TABLE** [Name] ([var1] [Datentyp1], [var2] [Datentyp2], …,

**PRIMARY\_KEY** [var1])

### INSERT

**INSERT** **INTO** [Name] **VALUES** ([var1], [var2], …)

### UPDATE

**UPDATE** [Name] **SET** col1 = val1, col2 = val2 **WHERE** condition;

## PL/SQL

PL/SQL teil sich in zwei Hauptkomponenten auf, die sich wie folgt unterscheiden:

|  |  |
| --- | --- |
| **Stored Procedure** | **Function** |
| * Haben keinen Rückgabewert * **Können** Funktionen aufrufen | * Haben einen Rückgabewert * Können **keine** Prozeduren aufrufen |

Um eine Stored Procedure oder eine Funktion aufzurufen, muss man den Befehl EXECUTE [PROCEDURE\_NAME] aufrufen.

Um in PL/SQL eine Ausgabe zu machen, muss man dbms\_output.put\_line(…) aufrufen.

**Achtung**: put\_line(…) gibt eine Ausgabe – nur put(…) nicht! Danach muss nochmal dbms\_output.new\_line aufgerufen werden, um die Ausgabe zu bestätigen.

Strings werden mit || konkateniert!

**Achtung:** Strichpunkte sind wichtig!

Das Initialisieren von Variablen erfolgt mit := (Bsp.: va\_a number := 5;)

### Stored Procedures

CREATE OR REPLACE PROCEDURE [Name] [([param] [Datentyp], …)]

AS

Deklarationsteil. Zb.: variable number;

BEGIN

Prozedurkörper

END [Name];

### Functions

CREATE OR REPLACE FUNCTION [Name] [([param] [Datentyp], …)]

RETURN [Datentyp]

AS

Deklarationsteil. Zb.: variable number;

BEGIN

Funktionskörper

RETURN [Rückgabewert];

END [Name];

### Weitere Syntax

#### Cursor / For-Schleife

FOR record IN cursor\_name

LOOP

Statements;

END LOOP;

Der cursor\_name kann hierbei auch ein Select-Statement sein. Jede Zeile ist dann ein einzelner record in der For-Schleife.

#### Select 🡪 Variable

…

AS

result number;

BEGIN

SELECT preis **INTO** result FROM Produkt WHERE id = 4;

END;

#### IF-Verzweigung

IF (condition)

THEN

…

ELSIF (condition) THEN

….

ELSE

…

END IF;

#### Delete

DELETE FROM [Table\_Name] where condition;

# Design-Pattern (Entwurfsmuster)

Für besseres Verständnis vom Code kann folgendes GitHub-Repository zurate gezogen werden:

<https://github.com/TecuilaCat/SoftwareTechnik>

## Singleton

Der Singleton ist das *trivialste* Entwurfsmuster, da es genau eine Klasse erfordert, die sich selbst als privates Attribut hat. Dieses muss statisch sein und kann direkt zu Beginn (*EAGER*-Loading) oder erst später (*LAZY*-Loading) initialisiert werden.

In der Klasse gibt es eine statische getInstance()-Methode, die das private Objekt zurückgibt. So wird sichergestellt, dass es von dieser Klasse immer nur genau ein Objekt gibt. Das ist vor allem dann nützlich, wenn man eine Ressource von mehreren Klassen aus nutzen möchte und keinen gleichzeitigen Zugriff erlauben will.

## Fabrik-Methode

Bei einer Fabrikmethode wird ein Objekt erzeugt. Grundsätzlich ist das nicht anders als ein Konstruktor, nur dass dieses Entwurfsmuster den Vorteil hat, dass man genau steuern kann, was vor dem Initialisieren passiert. Im ST-Unterricht behandeln wir nur das allgemeine Muster und die Theorie dahinter. In der realen Welt steht da viel mehr dahinter.

Ein Bild, das Diagramm enthält.

Automatisch generierte Beschreibung

### Simple-Factory

Eine Simple Factory besteht, wie der Name sagt, aus genau einer Klasse, die eine Methode beinhaltet, welche verschiedene Datentypen, die aber alle von einer Elternklasse erben, zurückgeben kann.

Hat man beispielsweise eine Reihe mehrerer Dokumente (Gutschrift, Lieferschein, Rechnung) und will sich diese über eine zentrale Methode erzeugen lassen, so müsste man über einen String (oder besser ein Enum) abprüfen, welches Objekt man genau haben will.

public enum Type {

GUTSCHRIFT,

LIEFERSCHEIN,

RECHNUNG

}

public static Dokument erzeugeDokument(Type type) {

**Achtung**: Java 17 Syntax!

return switch(type) {

case GUTSCHRIFT -> new Gutschrift();

case LIEFERSCHEIN -> new Lieferschein();

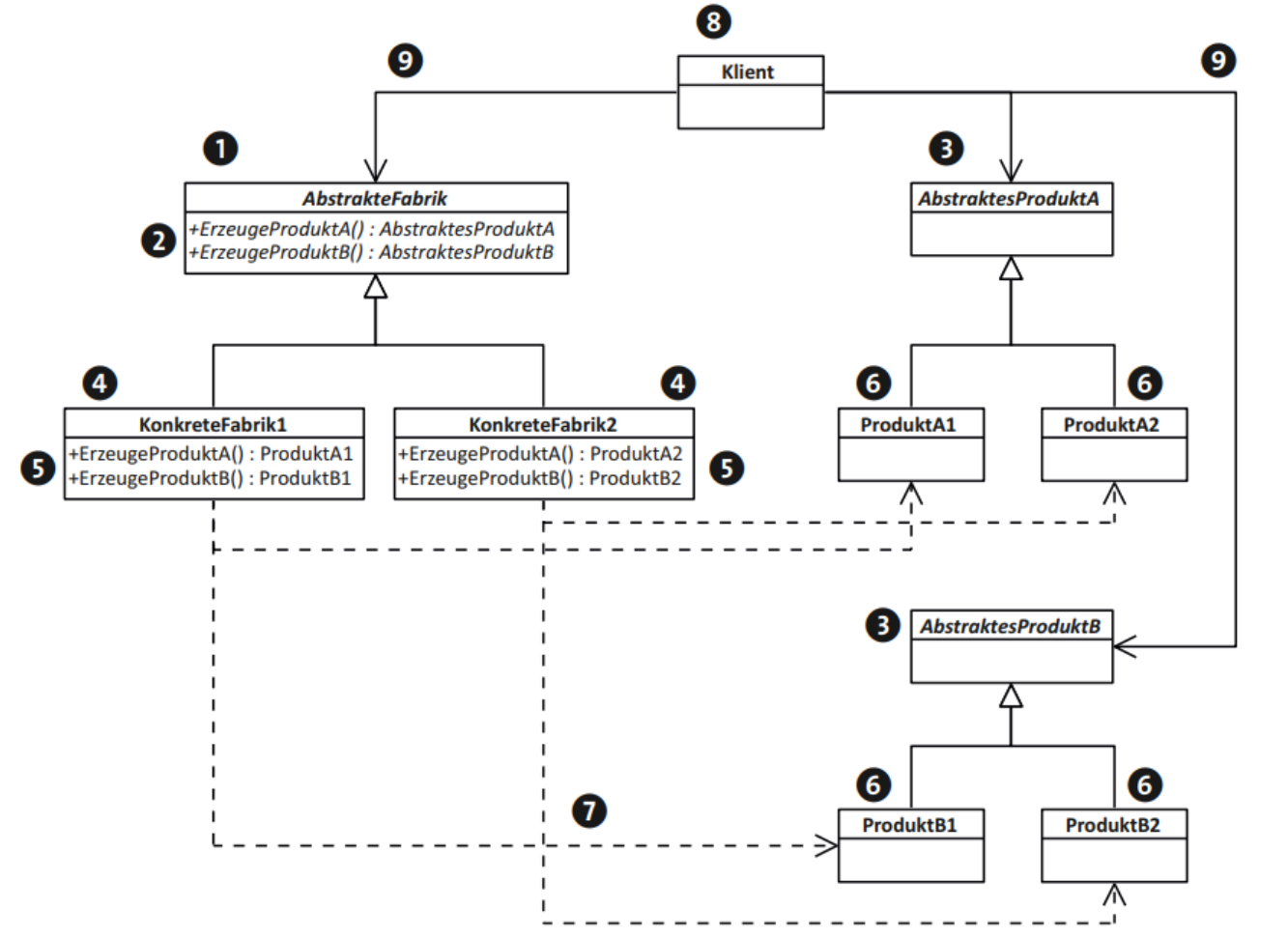
case RECHNUNG -> new Rechnung();

};

}

### Abstract-Factory

Bei der Abstrakten Fabrik geht es darum Produktlinien zu realisieren. Dabei wird kein String und kein Enum übergeben, sondern welches konkrete Produkt erstellt wird, hängt nur von den verwendeten Klassen und Methoden ab.



## Composite

Durch das Kompositum ergibt sich eine generelle Baumstruktur, bei der man komplexe Berechnungen oder Ähnliches rekursiv abarbeiten muss / kann. Ein Kompositum kann demnach beliebig viele Kinder haben, muss aber mindestens ein Blatt haben, sonst wird das Kompositum automatisch zum Blatt.

Wichtig: Immer bei Abfragen an den eigenen Wert denken. Man vergisst schnell, dass wenn ein Gesamtwert abgefragt wird, der Ausgangsknoten ebenfalls einen Wert hat, der verrechnet werden muss.

Ein Bild, das Diagramm enthält.

Automatisch generierte Beschreibung

## Observer

Das Observer Pattern setzt Publish und Subscribe um. Idee dabei ist es eine beliebige Anzahl an Observern an ein Subjekt zu hängen, welche alle benachrichtigt werden, sobald im Konkreten Subjekt etwas Bestimmtes passiert.

In dem unteren Beispiel könnte man zum Beispiel die benachrichtige()-Methode in der Methode setZustand(zustand: int) aufrufen. Dazu wird dann über alle Observer des Subjekts iteriert und auf jedem Objekt aktualisiere aufgerufen. Parameter ist dann immer das Objekt selbst (also this).

Ein Bild, das Diagramm enthält.

Automatisch generierte Beschreibung

**Vorteile:**

* Zustandskonsistenz
* Flexibilität und Modularität
* Wiederverwendbarkeit
* Kompatibel zu Schichtenmodell

**Nachteile:**

* Aktualisierungskaskaden möglich
* Abmeldung von Beobachtern zur Laufzeit

# Testen

Vergleich zwischen IST-Zustand und Soll-Zustand 🡪 Testen = Sicherstellung von Softwarequalität

## Grundsätze

1. Testen zeigt nur Anwesenheit von Fehlern
2. Vollständiges Testen ist nicht möglich
3. Frühes Testen spart Zeit und Geld
4. Fehlerclustering
5. Vorsicht vor Pestizid-Paradoxon
6. Testen ist kontextunabhängig
7. Trugschluss: keine Fehler bedeuten ein brauchbares System

## Voraussetzungen

* Gut und ausreichende Anforderungen
* Kenntnisse

## Blackbox-Tests

Bei Blackbox-Tests kennt der Tester nicht den Code, sondern kann nur überprüfen, ob das Gesamtergebnis mit dem erwarteten Ergebnis übereinstimmt. Ziel ist es die Software mit der Anforderung zu vergleichen.

Blackbox-Tests sind unabhängig vom Programmcode und können daher von Personen durchgeführt werden, die keine Programmierkenntnisse haben. Entwickler können dadurch nicht um ihre eigenen Fehler herumtesten.

## Äquivalenzklassen

Zu jedem Test können Äquivalenzklassen mit zugehörigen validen, invaliden und angrenzenden Daten erstellt werden. Diese gilt es dann übersichtlich in einer Tabelle darzustellen.

Dazu teilt man alles in 4 Spalten auf: Bereich gültiger Daten, Klassen gültiger Eingaben, Klassen ungültiger Eingaben und Repräsentanten dieser Klassen.

Hat man beispielsweise eine Attribut „Geburtsjahr“, welches zwischen 1962 und 2004 sein darf und will dazu die Tabelle aufstellen, sieht diese wie folgt aus:

|  |  |  |  |
| --- | --- | --- | --- |
| Bereich gültiger Daten | Klassen gültiger Eingaben | Klassen ungültiger Eingaben | Repräsentanten der Klasse |
| geburtsJahr (E)  Ganze Zahlen zwischen a=1962 bis n=2004 | a ≤ E ≤ n  Grenzfall  Grenzfall |  | 1990  1962  2004 |
|  | E < a  Grenzfall | 1900  1961 |
| E > n  Grenzfall | 2100  2005 |

## Whitebox-Tests

Bei Whitebox-Tests kennt der Tester den Code und kann entsprechende Abdeckung erreichen. Dazu gehören beispielsweise Unit-Testes. Das Ziel ist es die Korrektheit und Qualität der Software auf Code-Ebene festzustellen und Fehler aufzudecken, die das interne Verhalten beeinträchtigen.

Bei Whitebox-Tests lässt sich ein Programmablaufplan und ein Ablaufplan erstellen, aus dem hervorgeht, wie die Verzweigungen aufgebaut sind und wohin man zurückspringen muss, um jeden Zweig, jede Anweisung und jeden Term zu prüfen.

### Anweisungsüberdeckung

Versuch alle Anweisungen im Test mindestens einmal auszuführen

### Zweigüberdeckung

Versuch jeden Zweig mindestens einmal durchzulaufen

### Termüberdeckung

Versuch jeden Term (beispielsweise in if-Verzweigungen) durchzulaufen. Dabei muss beachtet werden, dass bei zwei veroderten Abfragen die zweite nicht ausgeführt werden, sollte die erste *True* sein. Genauso wird bei verundeten Abfrage die zweite nicht ausgeführt, sollte die erste *False* ergeben.

# Eigene Notizen