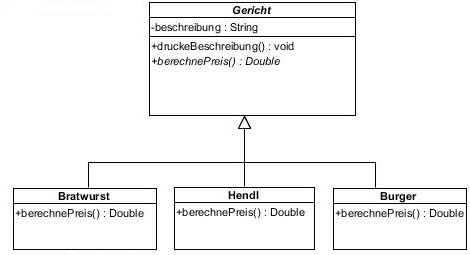
**Objektorientierte Modellierung und Implementierung – Übungsblatt 10**

Fercher, Frick, Qureshi

**Aufgabe 1**

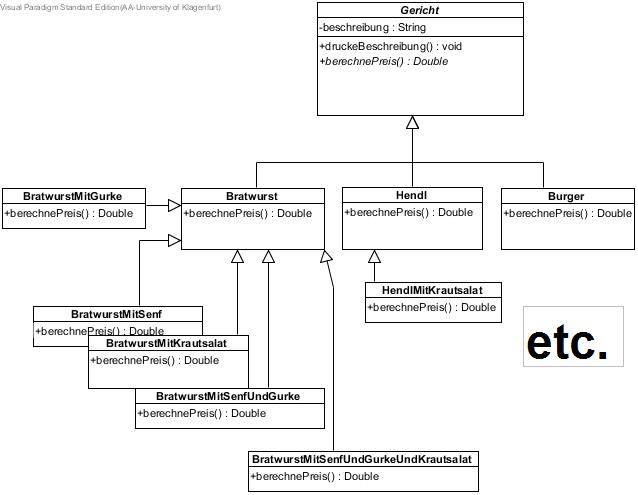
1.)

Eine Imbisskette vertreibt drei verschiedene Gerichte (Bratwurst, Hendl und Burger). Die Klassenarchitektur der verwendeten Abrechnungssoftware ist in folgendem Klassendiagramm dargestellt:



Jedes Gericht besitzt eine Methode *druckeBeschreibung*, die allgemeine Informationen über die jeweilige Speise am Bildschirm ausgibt. Die Methode *berechnePreis* gibt den Preis des jeweiligen Gerichts zurück (Bratwurst: 5 Euro, Hendl 7 Euro, etc.)

Das Unternehmen möchte nun auch für verschiedene Extras (Senf, Gurke, Krautsalat, Specialsauce), die zu den einzelnen Gerichten angeboten werden, Geld verlangen. Die Extras haben jeweils unterschiedliche Preise. Ein erster Versuch diese zusätzlichen Anforderungen im System abzubilden ist in folgendem Klassendiagramm dargestellt:



Für jede Kombination aus Gericht und Extras wird eine eigene in die Vererbungshierarchie eingebundene Klasse erstellt, die jeweils überschriebene Methode *berechnePreis()* liefert die Gesamtkosten des Gerichts.

Zum Beispiel *berechnePreis()* von Klasse B*ratwurstMitSenfUndGurke* (Bratwurst 5 Euro + Senf 0.20 Euro + Gurke 0.50 Euro = 5.70 Euro):

*public double berechnePreis(){*

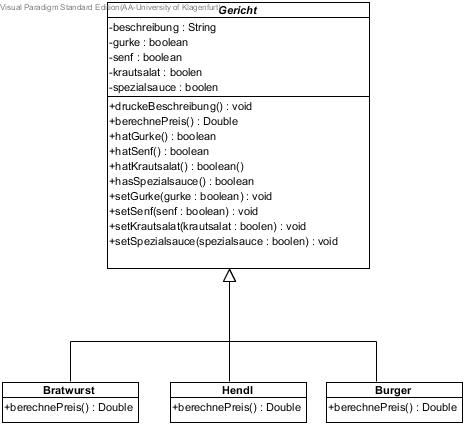
*return 5.70;*

*}*

* Funktioniert diese Lösung grundsätzlich?
* Welche Nachteile könnten bei dieser Lösung insbesondere in Hinblick auf spätere Wartbarkeit und Erweiterbarkeit der Implementierung entstehen?

2.)

Für die Problemstellung aus Teil 1 wird ein weiterer Lösungsvorschlag erstellt:



Die Verwaltung der Kosten für die Extras wird in der nunmehr nicht mehr abstrakten Methode *berechnePreis()* der abstrakten Klasse *Gericht* ausgeführt. Die gewählten Extras können für jedes Gericht über die setter-Methoden konfiguriert werden.

Methode *berechnePreis()* in Klasse *Gericht*:

*public double berechnePreis(){*

*double kosten = 0.00;*

*if(hatSenf()){*

*kosten = kosten + 0.20;*

*}*

*if(hatGurke()){*

*kosten = kosten + 0.50;*

*}*

*if(hatKrautsalat()){*

*kosten = kosten + 1.00;*

*}*

*if(hatSpecialsauce()){*

*kosten = kosten + 0.70;*

*}*

*return kosten;*

*}*

Die Methode *berechnePreis()* der Klasse *Bratwurst* sieht dann z.B. so aus:

*public double berechnePreis(){*

*return 5.00 + super.berechnePreis();*

*}*

Eine Bratwurst könnte wie folgt instanziiert bzw. konfiguriert werden:

*Bratwurst b1 = new Bratwurst();*

*b1. setSenf(true);*

*b1.setGurke(true);*

*b1.setSpecialsauce(false);*

*b1.setKrautsalat(false);*

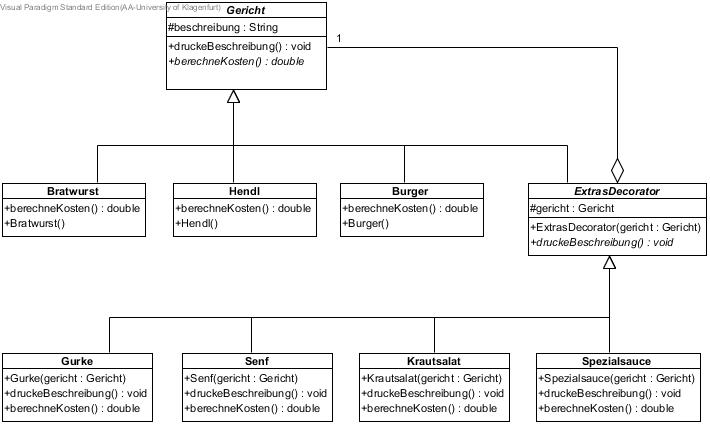
Welche Vorteile bringt diese Lösung gegenüber der ersten Lösung?

Überlegen Sie sich in Hinblick auf die Wartbarkeit und die Erweiterbarkeit des Designs unter anderem folgende Punkte:

* Was passiert, wenn neue Gerichte bzw. Extras hinzukommen (z.B. eine Palatschinke mit dem Extra Schokosauce)?
* Was passiert, wenn ein Kunde zwei Gurken und extra viel Senf zu seiner Bratwurst haben möchte?

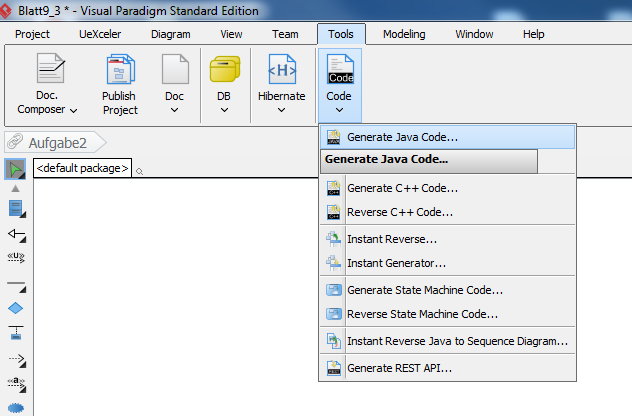
**Aufgabe 2**

Folgendes Klassendiagramm beschreibt eine weitere Alternative zu den Lösungsvorschlägen aus den Aufgaben 2+3, die auf dem Decorator-Pattern basiert:



Verwenden Sie das bereitgestellt Visual Paradigm Projekt um den Java Code für die Klassenhierarchie zu erzeugen.

Öffnen Sie das Projekt und wählen Sie: *Tools->Code->Generate Java Code*



Wählen Sie im nachfolgenden Fenster ein Verzeichnis in dem die Java Klassen erstellt werden sollen und bestätigen Sie mit OK.

Die generierten Klassen können Sie dann in Ihrer Java Entwicklungsumgebung weiter verwenden (Kopieren, Import, etc.). Angezeigte Fehler deuten in der Regel auf Code-Teile hin, die jedenfalls noch von Hand weiter ausimplementiert werden müssen (UnsupportedOperationException).

Finalisieren Sie die Implementierung und testen Sie Ihre Lösung mit der bereitgestellten Testklasse.

Welche Vorteile bietet diese Lösung gegenüber den Designs aus Aufgabe 1?

**Aufgabe 3**

Erklären Sie in Bezug auf UML-Zustandsdiagramme folgende Begriffe bzw. Konzepte:

* Ereignis,
* Bedingung,
* Aktivität, Entry-Aktivität, Exit-Aktivität, Do-Aktivität.
* History-Zustand - Was versteht man in diesem Zusammenhang unter flachem und tiefem History-Zustand?
* Komplexer Zustand
* UND-Verfeinerung, ODER-Verfeinerung

**Aufgabe 4**

Erstellen Sie ein UML-Zustandsdiagramm für eine Spielkonsole, die wie folgt beschrieben ist:

Eine Spielkonsole wird über einen zentralen Schalter ein- bzw. ausgeschalten. Nach dem Einschalten wechselt die Konsole in den Modus „Selbsttest“ wo zunächst alle Hardwarekomponenten und Teilsysteme geprüft werden. Liegt ein Fehler vor so erfolgt ein Wechsel in den Zustand „Fehler“ und die eingebaute LED Blinkt rot. Im Modus „Selbsttest“ wird auch geprüft ob ein Datenträger im Laufwerk eingelegt ist und ob dieser lesbar ist. Ist ein Datenträger eingelegt und ist dieser lesbar so wird das Spiel geladen und die Konsole wechselt in den Modus „Spielbereit“. Die eingebaute LED leuchtet in dem Fall ständig grün. Fehlt der Datenträger oder ist dieser nicht lesbar so erfolgt ein Wechsel in den Modus „kein Datenträger“ und die eingebaute LED blinkt orange. Wird in diesem Zustand ein neuer Datenträger eingelegt so erfolgt wieder ein Selbsttest mit Prüfung des Datenträgers.

Ist ein Spiel geladen und die Konsole spielbereit so kann das jeweilige Spiel mit dem Button „Start“ am Controller gestartet werden. Während des Spiels kann durch neuerliches drücken von „Start“ der Pausenmodus aufgerufen werden. Durch neuerliches Drücken von „Start“ wird der Pausenmodus wieder verlassen und es kann weitergespielt werden. Wird im Pausenmodus „Abbrechen“ gewählt so wird das aktuelle Spiel beendet und es erfolgt ein Wechsel in den Modus „Spielbereit“.

Über den Button „Eject“ an der Konsole kann der Datenträger jederzeit ausgeworfen werden. Das Spiel wird in dem Fall ebenfalls abgebrochen und die Konsole gelangt wieder in den Zustand „kein Datenträger“.