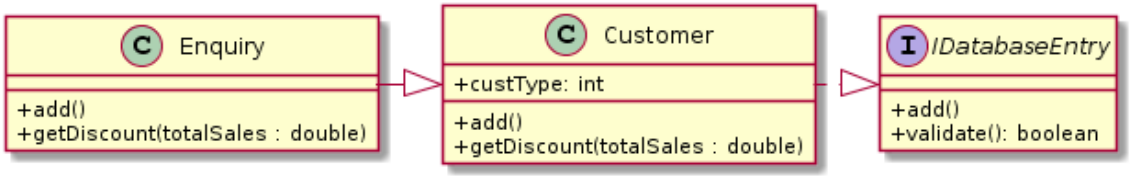
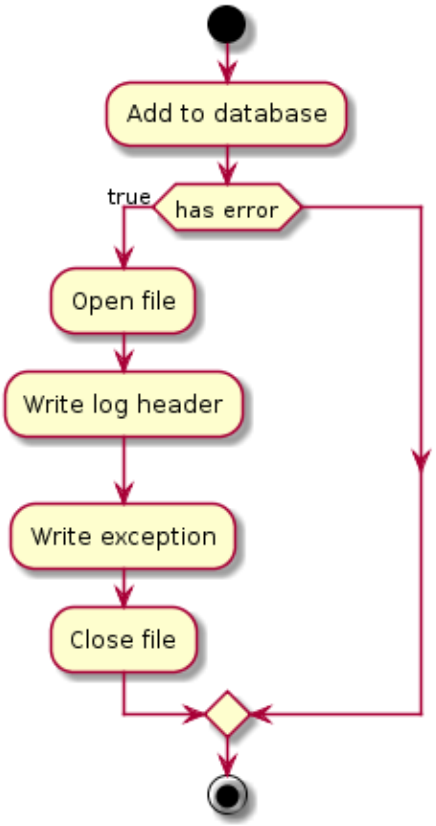
# Übungsblatt 6

Für alle Aufgaben soll mit folgendem UML Klassendiagramm gearbeitet werden.




**Aufgabe 16** Das Single Responsibility Prinzip

Die Methode *add* der *Customer* Klasse verstößt gegen das Single Responsibility Principle. Bis jetzt sieht der Algorithmus für die Methode so aus, wie in folgendem UML Diagramm dargestellt.

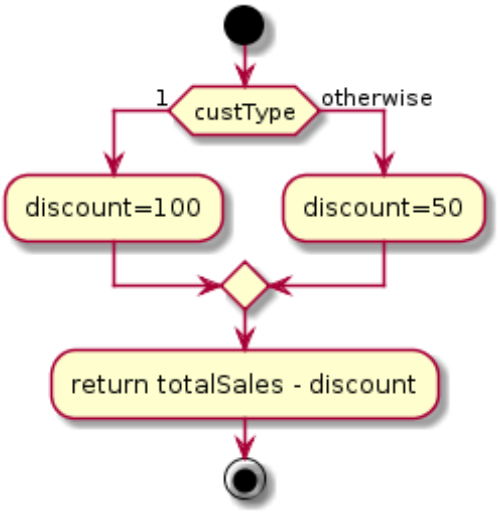



Ändern Sie die Methode (und möglicherweise den Klassenbaum) so ab, dass das SRP eingehalten wird. Eine Möglichkeit ist die Extrahierung des Logging-Funktionalität

in eine Klasse *FileLogger*.

### **Aufgabe 17** Das Open Close Prinzip

Auch die Methode *getDiscount* verstößt gegen ein wichtiges SOLID Prinzip. Um den Code flexibler zu halten soll der Code so geändert werden, dass das Open Closed Principle eingehalten wird. Der bisherige Algorithmus ist im nächsten Diagramm aufgezeichnet.




Ändern Sie die Methode (und möglicherweise den Klassenbaum) so ab, dass das OCP eingehalten wird. Legen Sie hierzu zwei neue Klassen *GoldCustomer* und *SilverCustomer* an, welche verschiedene Typen von Kunden darstellen. Jeder Typ erbt dabei von *Customer*.

### **Aufgabe 18** Das Liskov Substitution Prinzip

Die Hierarchie in dem Klassendiagramm ist nicht mit dem Liskov'schen Substitutionsprinzip vereinbar. So ist Tatsache, dass *Enquiry* von *Customer* erbt in mehrfacher Hinsicht problematisch.
Der folgende Java Code zeigt ein Problem, dass mit dieser Klassenstruktur verbunden ist.

```
class Enquiry extends Customer {
  @Override
    public double getDiscount(double totalSales) {
        return super.getDiscount(totalSales) - 5;
    }
  @Override
    public void add() {
        throw new Exception("Not allowed");
    }
}
```

Sorgen Sie dafür, dass ein *Enquiry* kein *Customer* mehr ist, jedoch beide immer noch

über mindestens ein gemeinsames Interface verfügen (z.B. *IDiscount*).

### **Aufgabe 19** Das Interface Segregation Prinzip

Das Interface *IDatabaseEntry* definiert nicht nur eine Operation *add*, sondern auch eine Operation *validate*. Die beiden Operationen haben nicht sehr viel gemein, und können unabhängig von einer Datenbank eingesetzt werden. Eine Trennung wäre auf jeden Fall angemessen und würde dem Interface Segration Principle folgen.
Führen Sie ein weiteres Interface *IValidate* ein, welches den Validierungsteil des *IDatabaseEntry* Interfaces enthält.

### **Aufgabe 20** Das Dependency Inversion Prinzip

Der in Aufgabe 16 eingeführte *FileLogger* ist zu unflexibel. Die Kopplung zwischen den Klassen *Customer* und *FileLogger* ist eindeutig zu stark. Durch eine Inversion of Control soll es nun möglich sein, auch andere Klassen für das Loggen von Informationen zu verwenden.
Erstellen Sie ein Interface *ILogger*. Dieses Interface soll von allen Loggern, wie z.B. *FileLogger* implementiert werden. Die Klasse *Customer* setzt Objekte, die dieses Interface implementieren zum Loggen ein. Fügen Sie weitere Logger, z.B. *EventViewerLogger* oder *EmailLogger* ein.
Zeichnen Sie anschließend ein Klassendiagramm, welchen den Klassenbaum nach dem Anwenden der SOLID Prinzipien zeigt.

#### **!** Wichtig

Alle Diagramme können über Anwendungen (z.B. PlantUML, yUML, Visual Studio, ...) oder auch per Hand gezeichnet werden.