Aufgabe 01a: Erstellen Sie ein Projekt **01a_SimUDuck**:

- Implementieren Sie die SimUDuck Applikation unter Anwendung des Strategy-Patterns (für das Flug-Verhalten und das Quack-Verhalten).
  - Sehen Sie jeweils mindestens zwei Strategien vor (z.B. Quack und Mute)
  - Sehen Sie eine Möglichkeit vor, die beiden Verhalten einer Duck während der Laufzeit zu ändern.
  - Hinweis: Da es sich um sehr einfache Klassen handelt, können Sie mehrere Klassen in einer Datei zusammenfassen (z.B. QuackBehavior.cs mit den Klassen Squeak, Quack und MuteQuack;) – Achten Sie jedoch auf eine sinnvolle Aufteilung der Klassen/Dateien.

Aufgabe 01b: Erweitern Sie das Projekt **01b_BankAccount** (Bank-Applikation zum Verwalten von Konten; Download-Link siehe wiki-Seite) um folgende Funktionalität:

Der Bankangestellte soll beim Anlegen des Accounts (wir machen das in der main-Methode) entscheiden können, ob der Inhaber des Accounts jemals einen negativen Kontostand haben darf. Wenn nicht, soll sich der Account entsprechend verhalten. → Withdraw darf nicht durchgeführt werden, falls nicht genug Geld am Konto vorhanden ist. Stellen Sie sicher, dass der Inhaber des Kontos bei einem fehlgeschlagenen Abhebeversuch entsprechend benachrichtigt wird.

Erstellen Sie hierfür eine Klasse BoundedAccount, welche von Account erbt und das entsprechend geänderte Verhalten implementiert. In der main-Methode entscheiden Sie, ob die Klasse Account oder BoundedAccount verwendet werden soll.

Die Methode Manager.DumpAccounts() soll in die Klasse Program verschoben werden. Erweitern Sie die Klasse Manager um neue Funktionalität, die es erlaubt, dass die Klasse Programm (oder andere Klassen) Zugriff auf die Accounts bekommt, um zB. das DumpAccounts durchzuführen, oder die Anzahl der Accounts abzufragen.

Untersuchen Sie das Original-Projekt und ihr geändertes Projekt auf „Unschönheiten“, und Verbesserungsmöglichkeiten → Verändern Sie ihr Projekt dementsprechend.

Sehen Sie Schwierigkeiten bei dem von uns gewählten Ansatz (Account vs. BoundedAccount) in der Praxis? Diesen Punkt besprechen wir in der Übung.

Aufgabe 01c: Erstellen Sie ein Projekt **01c_RPNStrategy** und implementieren die folgende Funktionalität:

Implement an interface for a stack (LIFO). Write two different stack implementations. One that is based on an array with constant size; the other shall be based on a linked list. Both classes shall implement the interface above. Use Generics. The stack classes must not rely on C# stack classes, implement them yourself. Go for a simple solution, but include some basic error handling (e.g. pop-operation on an empty stack).

Use your stacks to implement an RPN (reverse-polish-notation) calculator software component. See `RPNCalc.cs` for the main functionality, but (i) go for a better design, and (ii) include error handling.

The calculator shall work with both types of stacks (and any other implementation of the stack interface) without any modification (see "dependency injection"). The program takes as input an expression in RPN and outputs the calculated result. Example: input: 1.5 2 + 3 4 - * output: -3.5

Achten Sie auf sauberen Code. Beachten Sie Namenskonventionen von C#. Arbeiten Sie als 2er-Team, jedoch muss jeder einzeln im git-Repo einchecken (Verzeichnisname = Projektname). Die beiden Autoren müssen in jedem File im Header angeführt werden! Seien Sie vorbereitet auf Fragen zu „Strategy“ sowie OO-Konzepte und die besprochenen Designprinzipien. ***Deadline: Sonntag, 02.10. 16:00***