2.1 Stack

a) Erstelle eine alleinstehende Funktion, die sich selbst aufruft (Rekursion), mit den folgenden Anforderungen:

- Die Funktion soll eine 64-Bit Ganzzahl (z.B. `uint64_t`) mitführen, die für jeden Funktionsaufruf um eins erhöht wird.
- Bei jedem Aufruf der Funktion soll eine Ausgabe erfolgen, die beinhaltet, das wievielte Mal die Funktion aufgerufen wurde und an welcher Stack-Adresse sich der aktuelle Aufruf *grob* befindet, in der Art: `Aufruf Nr. 5, an Stack−Adresse: 0x00000057BD1CFC70`.

Hinweis: Als Näherung der Adresse des Stack-Frames kannst du die Adresse des Funktionsparameters (= lokale Variable) verwenden.

b) Führe die Funktion mit einem Startwert von 0 aus.

i. Wie oft wird die Funktion auf deinem System aufgerufen, bevor ein Fehler auftritt?
ii. Wie heißt der aufgetretene Fehler und was steckt dahinter?
iii. Wieviel Stack-Speicher wurde für die Funktionsaufrufe verbraucht?
iv. Fällt dir etwas am Verlauf der Speicheradressen auf?

c) Definiere nun eine lokale Variable mit 1 024 Bytes Größe in der Funktion (z.B. ein Character-Array).

i. Wie oft kann die Funktion jetzt aufgerufen werden, bevor ein Fehler auftritt?
ii. Welchen Schluss kannst du daraus in Bezug auf lokale Funktionsvariablen ziehen?

2.2 Heap-Allokator

In dieser Aufgabe wollen wir die Speicherverwaltung unserer `Vector`-Klasse für `double` Typen aus dem letzten Übungsblatt erweitern. Dazu sollen die speicherverwaltenden Methoden über eine einheitliche Schnittstelle abstrahiert und beliebig ausgetauscht werden können. Dabei können die unterschiedlichen Methoden der manuellen Speicherverwaltung näher betrachtet werden.
Zur Bearbeitung der Aufgabe kannst du deine Lösung der Übung 1.1 oder den Lösungsvorschlag dazu verwenden.

a) Erstelle eine neue Schnittstellenklasse `VectorMemory` mit den folgenden Kriterien:

- Eine Methode `allocate(elementCount)`, mit der eine bestimmte Anzahl von Elementen des Vektors allokiert wird, wenn zuvor noch keine Elemente vorhanden waren (also der Vector leer war).
- Eine Methode `resize(elementCount)`, mit der die Elementanzahl im Vector verändert wird. Dabei kann die Anzahl (und somit die Allokation) sowohl vergrößert als auch verkleinert werden.
- Eine Getter-Methode, mit der die aktuelle Anzahl der allokierten Elemente erhalten werden kann.
- Eine Getter-Methode, mit der eine veränderbare Referenz eines Elements an einem spezifischen Index erhalten werden kann.
- Der Destruktor muss von ableitenden Klassen überschrieben werden können.

Hinweis: Bei der Deklaration eines abstrakten Destruktors kann ein bekannter Linkerfehler auftreten. Im Internet findest du Lösungsmöglichkeiten dazu.

b) Inkludiere die Schnittstellenklasse `VectorMemory` in der `Vector`-Klasse (Header). Überarbeite dann die `Vector`-Klasse wie folgt:

- Ändere den Konstruktor, sodass eine Instanz von `VectorMemory` bei der Erstellung eines Vectors angegeben werden kann.
- Führe alle Speicherallokationen durch Aufruf der Methoden `allocate(elementCount)` und `resize(elementCount)` der `VectorMemory`-Instanz durch.
- Entnehme die Vector-Elemente nur noch aus der `VectorMemory`-Instanz.
- Entnehme die Information über die Anzahl der Elemente nur noch aus der `VectorMemory`-Instanz.

Hinweis: Es bietet sich an, die Instanz der `VectorMemory`-Realisierung über eine Membervariable in `Vector` mitzuführen (Pointer oder Referenz).

c) Implementiere eine neue Klasse, die die Schnittstelle `VectorMemory` realisiert. In dieser sollen die Elemente vom Typ `double` mit dem `new[]`-Operator allokiert und mit dem `delete[]`-Operator deallokiert werden. Beachte dabei:

- Wird Speicher neu allokiert, so sollen die alten Vector-Elemente so weit wie möglich erhalten bleiben.
- Neu-allokierte Elemente sollen auf `0` initialisiert werden.

d) Implementiere eine neue Klasse, die die Schnittstelle `VectorMemory` realisiert. In dieser sollen die Elemente vom Typ `double` mit einer Funktion aus der `malloc()`-Reihe allokiert und mit der Funktion `free()` deallokiert werden. Beachte dabei:

- Wird Speicher neu allokiert, so sollen die alten Vector-Elemente so weit wie möglich erhalten bleiben.
- Neu-allokierte Elemente sollen auf `0` initialisiert werden.

e) Erstelle jeweils eine Instanz von `Vector` für die beiden Realisierungen von `VectorMemory` und überprüfe, ob sich die Operationen aus dem Übungsblatt 1 noch fehlerfrei ausführen lassen.

f) Erstelle jeweils eine Instanz von `Vector` für die beiden Realisierungen von `VectorMemory` und vergleiche die Performanz der beiden Methoden zur Speicherverwaltung. Binde hierzu den mitgelieferten Header `PerfTestAllocator.hpp` in dein Projekt ein und nutze die Funktion `perfTestAllocator(Vector& v, int elementCount, int iterations)` um den Test für die beiden Vectoren durchzuführen. Beachte dabei:

- Der Test fügt dem Vector mehrere Elemente hinzu und löscht diese danach wieder einzeln.
- `perfTestAllocator(...)` geht von einer Methode `push_back(double)` zum Einfügen von Elementen und `setSize(unsigned)` zum Ändern der Vector-Größe aus. Sollten deine Methoden anders heißen, musst du sie im Test entsprechend anpassen.
- Führe den Test mit geeigneten Werten für die Elementanzahl im Vector (`elementCount`) und der Testdurchläufe (`iterations`) aus, sodass dieser zwischen 5 und 60 Sekunden im Worst-Case andauert. Ein guter Mittelwert sind 10 000 Elemente bei 100 Testdurchläufen.
- Nutze die Projektkonfiguration „Release“ um ein aussagekräftigeres Testergebenis zu erhalten.

i. Gibt es einen Unterschied zwischen der Performanz von new-delete und malloc-free? Wenn ja, wie unterscheiden sich die beiden Methoden?

ii. Wie erklärst du dir den Performanzunterschied, wenn einer existiert?

g) Können die beiden Allokatoren (new-delete & malloc-free) auch für Klasseninstanzen statt Elementen vom Typ `double` verwendet werden? Begründe für jeden Allokator warum bzw. warum das nicht nicht möglich ist und überlege dir ob es bei Klasseninstanzen sonst noch etwas zu beachten gibt.

h) Hast du bei den beiden Allokatorenklassen und der `Vector`-Klasse die RAII (resource acquisition is initialization) Programmiertechnik verwendet? Wenn ja: wo und wie genau? Wenn nein: warum?

i) Was würde passieren, wenn du versehentlich über die Grenze der Allokation hinaus Array-Elemente beschreibst?

j) *Bonusaufgabe:* Informiere dich, wie Probleme am Heap, die durch Programmierfehler hervorgerufen werden, gefunden und analysiert werden können.

2.3 Wert-Ausdrücke (value categories)

Betrachte die folgenden Codeausschnitte und beantworte die jeweiligen Fragen. Zur Überprüfung kannst du die Ausschnitte in einem eigenen Programm ausführen. Dazu liegen dem Übungsblatt die Dateien `ValCat_A.cpp`, `ValCat_B.cpp`, `ValCat_C.cpp`, `ValCat_D.cpp` für die jeweiligen Teilaufgaben bei. Überlege dir jedoch zuerst die Antworten ohne Zuhilfenahme des Computers.

a)

```
1  #include <iostream>
2
3  int main()
4  {
5      int a = 42;
6      int& b = 42;
7      const int& c = 42;
8      int&& d = 42;
9
10     a = 1;
11     b = 2;
12     c = 3;
13     d = 4;
14
15     std::cout << a << ", "
16         << b << ", "
17         << c << ", "
18         << d;
19 }
```

i. Bennene die Wert-Ausdrücke der Variablen:

- **a** (Zeile 5):
- **b** (Zeile 6):
- **c** (Zeile 7):
- **d** (Zeile 8):

ii. Gibt es Zeilen, die nicht kompilieren? Benenne diese und begründe den Compilerfehler mit Begriffen der Wert-Ausdrücke!

iii. Entstehen in dem Codeausschnitt X-Values? Wenn ja wo und warum? Beachte nur Zeilen, die sich auch erfolgreich kompilieren lassen.

iv. Wie lautet die Ausgabe ab Zeile 15 unter der Annahme, dass nicht-kompilierbare Zeilen herausgenommen wurden? Begründe jede Zahlenausgabe.

b) Die Initialisierungs-Ausdrücke wurden nun verändert (Zeile 6 - 8):

```
1  #include <iostream>
2
3  int main()
4  {
5      int a = 42;
6      int& b = a;
7      const int& c = a;
8      int&& d = a;
9
10     a = 1;
11     b = 2;
12     c = 3;
13     d = 4;
14
15     std::cout << a << ", "
16         << b << ", "
17         << c << ", "
18         << d;
19 }
```

i. Gibt es Zeilen, die nicht kompilieren? Benenne diese und begründe den Compilerfehler mit Begriffen der Wert-Ausdrücke!

ii. Entstehen in dem Codeausschnitt X-Values? Wenn ja wo und warum? Beachte nur Zeilen, die sich auch erfolgreich kompilieren lassen.

iii. Wie lautet die Ausgabe ab Zeile 15 unter der Annahme, dass nicht-kompilierbare Zeilen herausgenommen wurden? Begründe jede Zahlenausgabe.

c) Der Datentyp der Referenzen wurde nun ausgetauscht (`int&` → `double&`, Zeile 6 - 8):

```
1  #include <iostream>
2
3  int main()
4  {
5      int a = 42;
6      double& b = a;
7      const double& c = a;
8      double&& d = a;
9
10     a = 1;
11     b = 2;
12     c = 3;
13     d = 4;
14
15     std::cout << a << ", "
16         << b << ", "
17         << c << ", "
18         << d;
19 }
```

i. Gibt es Zeilen, die nicht kompilieren? Benenne diese und begründe den Compilerfehler mit Begriffen der Wert-Ausdrücke!

ii. Entstehen in dem Codeausschnitt X-Values? Wenn ja wo und warum? Beachte nur Zeilen, die sich auch erfolgreich kompilieren lassen.

iii. Wie lautet die Ausgabe ab Zeile 15 unter der Annahme, dass nicht-kompilierbare Zeilen herausgenommen wurden? Begründe jede Zahlenausgabe.

iv. Vergleiche das Verhalten des Codes mit dem aus Teilaufgabe a). Welche Unterschiede kannst du feststellen?

d) Nun wird ein Objekt vom Typ `std::string` betrachtet. Es sind drei Funktionen für die drei Referenztypen *überladen*:

```
1  #include <iostream>
2  #include <string>
3
4  void func(std::string& s) // modifiable L-Value-Reference
5  {
6      std::cout << "mlvRef" << std::endl;
7  }
8
9  void func(const std::string& s) // non-modifiable L-Value-Reference
10 {
11     std::cout << "nmlvRef" << std::endl;
12 }
13
14 void func(std::string&& s) // R-Value-Reference
15 {
16     std::cout << "rvRef" << std::endl;
17 }
18
19 int main()
20 {
21     std::string s1 = "Hallo";
22     const std::string s2 = "Kempten";
23     const char* s3 = "Amogus";
24
25     func(s1);
26     func(s2);
27     func(s1 + s2);
28     func(s3);
29     func(std::string("Bla"));
30     func("Blub");
31 }
```

i. Betrachte die Funktionsaufrufe (ab Zeile 25). Wie lautet die Ausgabe? Begründe jede Ausgabezeile so präzise wie möglich.

ii. Wie sieht die Ausgabe aus, wenn die Überladung `void func(std::string& s)` gelöscht wird? Ist das Programm dann noch ausführbar? Begründe deine Antwort.

iii. Wie sieht die Ausgabe aus, wenn die Überladungen `void func(std::string& s)` und `void func(std::string&& s)` gelöscht werden? Begründe deine Antwort.

iv. Wie müsste eine Überladung definiert werden, die nur PR-Values annimmt?

2.4 Kopien und Verschiebungen

In dieser Aufgabe soll unsere `Vector`-Klasse für Kopier- und Verschiebeoperationen ertüchtigt werden. Nutze zur Bearbeitung der Aufgabe deine Lösung der Übung 1.1 oder den Lösungsvorschlag dazu. Verwende **nicht** deine Lösung zur Aufgabe 2.2 dieses Übungsblatts (Heap-Allokator)!

a) Eine `Vector`-Instanz soll auf eine andere `Vector`-Instanz kopiert werden können. Erweitere die Klasse hierzu um einen Kopier-Konstruktor und einen Kopier-Zuweisungsoperator. Kopiere dabei die zugrundeliegenden Vector-Elemente, sodass eine vollwertige Kopie der Instanz entsteht (deep copy).

b) Überprüfe deine Kopieroperationen mit folgendem Code, der auch als `TestCopy.hpp` der Übung beiliegt:

```
1  #include <iostream>
2  #include "Vector.h"
3
4  void testCopy()
5  {
6      Vector v1;
7      v1.push_back(1);
8      v1.push_back(2);
9      v1.push_back(3);
10     v1.push_back(5);
11
12     Vector v2 = v1;
13
14     Vector v3;
15     v3.push_back(8);
16     v3.push_back(13);
17     v3.push_back(21);
18     v1 = v3;
19     v3.push_back(34);
20
21     std::cout << "V1: " << v1 << std::endl
22         << "V2: " << v2 << std::endl
23         << "V3: " << v3 << std::endl;
24 }
```

i. Wie lautet die Ausgabe?

ii. In welchen Zeilen wurde ein Konstruktor aufgerufen? Welche Art des Konstruktors wurde aufgerufen?

iii. In welchen Zeilen wurde ein Zuweisungsoperator aufgerufen? Welche Art des Zuweisungsoperators wurde aufgerufen?

iv. Wurde in dem Codeausschnitt eine `Vector`-Instanz unzulässig verwendet?

v. Was würde passieren wenn der Code ohne explizit definierte Kopieroperationen (Konstruktoren & Zuweisungsoperatoren) des Vectors ausgeführt wird?

c) Eine `Vector`-Instanz soll in eine andere `Vector`-Instanz verschoben werden können. Erweitere die Klasse hierzu um einen Verschiebungs-Konstruktor und Verschiebungs-Zuweisungsoperator. Übernehme dabei die zugrundeliegenden Vector-Elemente, sodass die verschobene Instanz nach der Operation leer ist und keine zusätzliche Allokation bei der Zuweisung entsteht.

d) Überprüfe deine Verschiebungsoperationen mit folgendem Code, der auch als `TestMove.hpp` der Übung beiliegt:

```
1  #include <iostream>
2  #include "Vector.h"
3
4  void testMove()
5  {
6      Vector v1;
7      v1.push_back(1);
8      v1.push_back(2);
9      v1.push_back(3);
10     v1.push_back(5);
11
12     Vector v2 = std::move(v1);
13
14     Vector v3;
15     v3.push_back(8);
16     v3.push_back(13);
17     v3.push_back(21);
18     v1 = std::move(v3);
19     v3.push_back(34);
20
21     std::cout << "V1: " << v1 << std::endl
22         << "V2: " << v2 << std::endl
23         << "V3: " << v3 << std::endl;
24 }
```

i. Wurde in dem Codeausschnitt eine `Vector`-Instanz unzulässig verwendet?

ii. Betrachte Zeile 19 genauer in Bezug auf die Implementierung deiner `push_back()`-Methode und des Verschiebungs-Zuweisungsoperators. Wie verhält sich das Programm hier?

iii. Wie lautet die Ausgabe unter der Annahme, dass die Operation in Zeile 19 geglückt ist?

e) Betrachte folgende Verwendung einer `Vector`-Instanz. Der Code liegt auch als `TestAssign.hpp` der Übung bei:

```
1  #include <iostream>
2  #include "Vector.h"
3
4  void testAssign()
5  {
6      Vector v1;
7      v1.push_back(1);
8      v1.push_back(2);
9      v1.push_back(3);
10     std::cout << "V1: " << v1 << std::endl;
11
12     v1 = v1;
13     std::cout << "V1: " << v1 << std::endl;
14 }
```

i. Ist die Zuweisung in Zeile 12 zulässig?

ii. Oberprüfe ob obiger Code mit deiner Implementierung problemlos ausführbar ist. Musst du die Art der Zuweisung in Zeile 12 speziell berücksichtigen?

f) Wie würden sich Zuweisungsoperationen unter den `Vector`-Instanzen verhalten, wenn nur ein Verschiebungs-Konstruktor und ein Verschiebungs-Zuweisungsoperator angegeben wurde?

g) Kannst du alternativ zur Definition des Kopier- und Verschiebungsverhaltens von `Vector` auch definieren, dass generell keine Instanzen von `Vector` kopiert oder verschoben werden dürfen? Wenn ja wie?

h) Ergibt es für die `Vector`-Klasse Sinn, den parameterlosen Konstruktor vom Compiler erzeugen zu lassen (`= default;`), statt diesen selbst anzugeben? Begründe deine Antwort.

2.5 E-Mails mit Smart-Pointern

In dieser Aufgabe soll die Architektur eines einfachen E-Mail-Programms mit Hilfe von Smart-Pointern implementiert werden. Das UML-Klassendiagramm in Abbildung 1 gibt dir einen Überblick über die Programmteile und Prototypen der geforderten Funktionalität. **Die Definition der Smart-Pointer-Typen ist wichtiger als die komplexe Implementierung der Programmlogik!**
Verwende für mehrere Verweise auf eine Klasse („beliebige Mengen“ im UML-Diagramm) ein Array fester Länge (z.B. Maximalanzahl 5 für alle Mengenobjekte). Daten sollen so effizient wie möglich verwaltet werden – vermeide unnötige Kopiervorgänge.

Hinweis: UML-Diagramme stellen Strukturen und Funktionalität unabhängig von der verwendeten Programmiersprache dar. Pointer und Referenzen werden generell nicht dargestellt. Bei dieser Aufgabe musst du dir speziell überlegen, welchen Smart-Pointer-Typen du für welche Objekte und deren Verbindungen einsetzen willst.
Implementiere public Member als private Membervariablen mit einer Getter-Funktion und setze diese über Konstruktorparameter. Setter-Funktionen sind für diese Aufgabe nicht nötig.

a) Erstelle eine Klasse `Person` für eine Person und `AddressBook` für ein Adressbuch, wie sie im UML-Klassendiagramm dargestellt sind. Erfülle dabei folgende Kriterien:

- Personen werden über die `addPerson()`-Methode erstellt und dem Adressbuch hinzugefügt. Die erstellte Person soll dabei zurückgegeben werden.
- Personen, die bereits erstellt wurden, können über die `getPerson()`-Methode über ihren Namen erhalten werden. Dazu definiert der `Person`-Klasse eine `equalsName()`-Methode.
- Personen werden über die `removePerson()`-Methode entfernt. Dabei werden sie ebenfalls über ihren Namen identifiziert.
- Mit `printStatus()` werden alle Personen eines Adressbuchs in die Konsole ausgegeben. Zusätzlich soll pro Person ausgegeben werden, wie viele Referenzen auf die Person gerade existieren.
- Personen können andere Personen als Freunde (`friends`) mit der `addFriend()`-Methode zugewiesen bekommen. Die aktuellen Freunde einer Person werden mit der `printFriends()`-Methode in die Konsole ausgegeben.
- Personen werden vollständig von der `AddressBook`-Klasse verwaltet.
- Das Hinzufügen von Freunden hat keinen Einfluss auf die Lebensdauer von Personenobjekten.
- Jede Instanz der Objekte gibt in die Konsole aus, wenn sie erstellt und wenn sie gelöscht wird. Bei Personen wird dabei zusätzlich ihr Name ausgegeben, damit die Instanzen besser identifiziert werden können.

b) Überprüfe deine Implementierung, indem du ...

- ein Adressbuch anlegst,

- mindestens drei Personen darin erstellst,
- mindestens zwei Personen miteinander befreundest,
- danach eine Person entfernst, die mit einer anderen Person befreundet war.

Hierzu kannst du die Testmethode in der Datei `TestAddressBook.hpp` nutzen. Überprüfe deine Implementierung anhand der Ausgabe der `printStatus()`-Methode. Insbesondere, ob die Personen-Instanzen korrekt gelöscht werden.

c) Eine E-Mail soll später entweder als Klartext oder als HTML-Markup dargestellt werden können. Implementiere dazu die Klassen `MailBody`, `PlainText` und `HtmlMarkup`, wie sie im UML-Klassendiagramm dargestellt sind. Beachte:

- `MailBody` ist eine abstrakte Klasse.
- Für alle Darstellungen kann der E-Mail-Inhalt über die `getContent()`-Methode als Text abgefragt werden. Die jeweiligen Darstellungsformen (Klartext & HTML) geben dabei ihre Darstellung als Text zurück.
- Implementiere **keine** Funktionalität für die Darstellungsformen (Klartext & HTML)! Erstelle lediglich eine Membervariable für den Text bzw. das Markup und gib diese in `getContent()` jeweils zurück.

d) Erstelle eine Klasse `Mail` für eine E-Mail und `Mailbox` für ein Postfach, wie sie im UML-Klassendiagramm dargestellt sind. Erfülle dabei folgende Kriterien:

- E-Mails werden über die `addMail()`-Methode erstellt und dem Postfach hinzugefügt. Die erstellte E-Mail soll dabei zurückgegeben werden.
- Mit `printStatus()` werden alle E-Mails eines Postfachs in die Konsole ausgegeben. Jede Mail gibt dabei ihren Sender, Empfänger und den Inhalt der Mails aus. Zusätzlich soll pro E-Mail ausgegeben werden, wie viele Referenzen auf die E-Mail gerade existieren.
- E-Mails werden vollständig von der `Mailbox`-Klasse verwaltet.
- Personen können nicht gelöscht werden, wenn von ihnen verschickte oder empfangene E-Mails vorhanden sind. Sie können jedoch aus dem Adressbuch entfernt werden.
- Als E-Mail-Körper (`body`) kann eine beliebige Darstellungsform (Klartext oder HTML) genutzt werden. Es kann jedoch nur eine einzige Instanz eines E-Mail-Körpers pro E-Mail existieren.
- Jede Instanz der Objekte gibt in die Konsole aus, wenn sie erstellt und wenn sie gelöscht wird. Bei E-Mails wird dabei zusätzlich ihr Sender, Empfänger und der Inhalt ausgegeben, damit die Instanzen besser identifiziert werden können.

e) Überprüfe deine Implementierung, indem du ...

- ein Adressbuch mit Personen anlegst wie in Teilaufgabe b),
- ein Postfach anlegst,
- mindestens zwei E-Mails erstellst, die von Personen aus dem Adressbuch verschickt umd empfangen wurden,
- danach eine Person aus dem Adressbuch entfernst, die eine E-Mail verschickt oder erhalten hat.

Hierzu kannst du die Testmethode in der Datei `TestMailbox.hpp` nutzen.

Hinweis: Der in der Testmethode verwendete `MakeMyPointer` muss mit dem Pointer deiner Wahl für die E-Mail-Körper ersetzt werden.

Überprüfe deine Implementierung anhand der Ausgabe der `printStatus()`-Methoden. Insbesondere, wie sich die E-Mail-Instanzen verhalten, nachdem ein Sender bzw. Empfänger aus dem Adressbuch entfernt wurde.

f) Enthält die Architektur einen zyklischen Verweis? Wenn ja wo?

g) Was musst du bei zyklischen Verweisen in Bezug auf Smart-Pointer beachten?

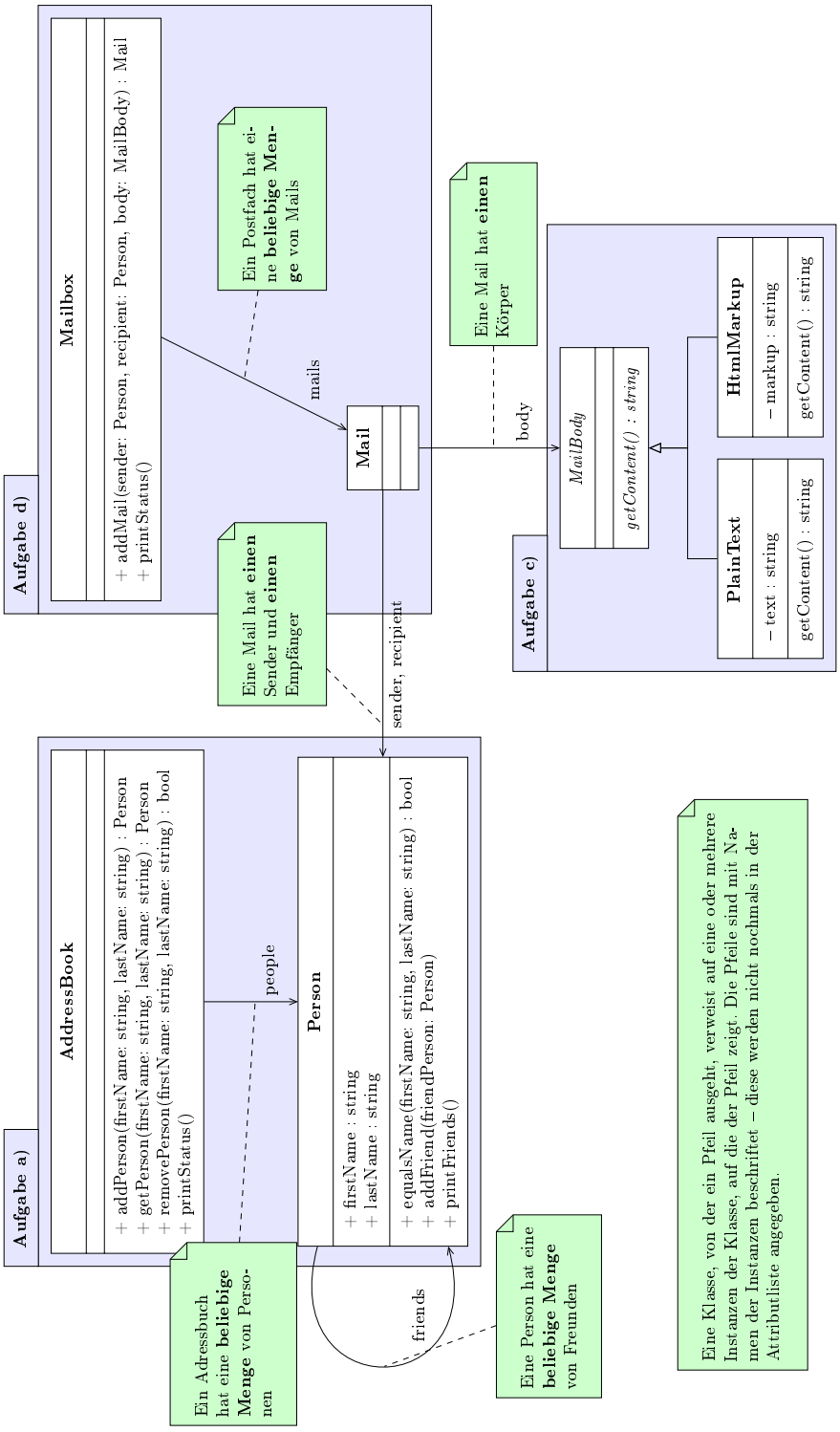

Abbildung 1: E-Mail-Software