OOAD
Aufgabenblatt 01

Prof. Dr.-Ing. Michael Uelschen
Hochschule Osnabrück
Laborbereich Technische Informatik
m.uelschen@hs-osnabrueck.de

Wiederholung und Einarbeitung

Die erste Aufgabe dient zur Wiederholung des Konzeptes *smarter Zeiger* mit C++14 und der Einarbeitung in das Werkzeug *Visual Paradigm* zur Software-Modellierung in UML.

Aufgabe 1: „Smarte Zeiger“ mit C++14

In den bisherigen ersten Semestern haben Sie die objektorientierte Programmierung mit C++ kennengelernt. Das Funktionsprinzip eines Zeigers (Verweis auf eine Variable oder auf ein Objekt) aus der Sprache C führt zusammen mit einer dynamischen Speicherverwaltung (`new`) häufig zu Speicherlöchern, wenn die Speicherfreigabe (`delete`) fehlerhaft implementiert ist. In C++ sind diese „rohen“ Zeiger um intelligente oder „smarte“ Zeiger erweitert worden. Diese überwachen die Verweise auf eine Variable oder auf ein Objekt und löschen diese automatisch, sobald diese nicht mehr verwendet werden. In größeren Anwendungen existiert eine Vielzahl von Objekten, die untereinander Informationen austauschen. Hierzu werden Verweise benötigt, um die entsprechenden Objekte ansprechen zu können.

Um in späteren Aufgaben ein Software-Modell in die Programmiersprache C++ umsetzen zu können, wiederholen und vertiefen Sie Ihre C++-Kenntnisse.

Aufgabe 1a)

Wiederholen Sie die Funktionsweise der smarten Zeiger in C++14. Was sind die Unterschiede von `std::unique_ptr`, `std::shared_ptr` und `std::weak_ptr`? Was ist mit dem smarten Zeiger `std::auto_ptr`? Wozu wird die Funktion `std::move` im Zusammenhang mit `std::unique_ptr` benötigt?

Im Zusammenspiel mit den smarten Zeigern wird die explizite Verwendung durch den Entwickler/in von new und `delete` vermieden. Stattdessen sind die Funktionen `std::make_unique` und `std::make_shared` zu verwenden. Erklären Sie deren Funktionsweise. Wieso gibt es keine Funktion `std::make_weak`? Wie können Sie solch dennoch einen „schwachen“ Zeiger erzeugen?

Sie weisen Ihr Wissen über smarte Zeiger nach, wenn Sie an einfachen, selbst entwickelten Programmen die Funktionsweise darstellen können. Hinweis: Trotz der smarten Zeiger sind weiterhin Referenzen oder einfache, „rohe“ Zeiger in der Programmierung notwendig (siehe auch R.30 der CPP Core Guidelines).

Hilfestellungen im Internetz, z.B. https://www.grimm-jaud.de/index.php/blog/std-unique-ptr und nachfolgende Seiten. Machen Sie sich auch mit den CPP Core Guidelines https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines vertraut (insbesondere mit den Regeln über smarte Zeiger (Regel R.20 – R.37).

Aufgabe 1b)

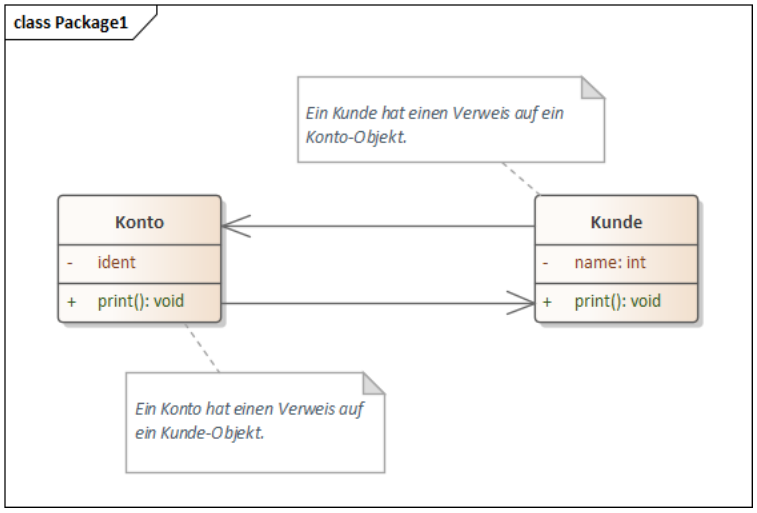

Abbildung 1: Zwei Klassen, die aufeinander verweisen.

Betrachten Sie das einfache Beispiel in der Abbildung 1 zweier vereinfachter Klassen `Konto` und `Kunde`, die aufeinander verweisen. Ein Konto hat das Attribute `ident` (Datentyp `std::uint32_t`), welches eine eindeutige Kennung darstellt. Ein Kunde hat das Attribut `name` (Datentyp `std::string`). Es wird angenommen, dass ein Kunde genauso lange existiert, wie das entsprechende Konto, d.h. wenn ein Konto-Objekt zerstört wird, so wird auch das zugehörige Kunde-Objekt zerstört.

Implementieren Sie die Funktion `createOwnerAccount` entsprechend der Signatur

```
std::unique_ptr<Konto> createOwnerAccount(const std::string& name, std::uint32_t account);
```

die einen Namen (Klasse Kunde) und eine Kennung (Klasse Konto) erhält und ein Konto-Objekt zurückgibt. Realisieren Sie dazu die beiden Klassen und setzen Sie die Verweise entsprechend der Abbildung. Der Verweis der Klasse Konto auf die Klasse Kunde soll ein smarter `std::unique_ptr` Zeiger sein. Welche Art von Zeiger müssen Sie für den Verweis in die umgekehrte Richtung, also von der Kunden- auf die Konto-Klasse, verwenden? Zeigen Sie, dass das zugehörige Kunden-Objekt automatisch zerstört wird, wenn das Konto-Objekt nicht mehr benötigt wird. Implementieren Sie für beide Klassen eine Methode `print`, die den Verweis von-nach Objekt auf die Konsole ausgibt.

Zur besseren Visualisierung und zur Fehlersuche können Sie auf die Basisklasse `Countable` zurückgreifen, die im Konstruktor- und im Destruktoraufruf Informationen auf `std::cout` ausgibt. Hierzu erhält jedes Objekt eine eindeutige Seriennummer.

```
class Countable {
public:

    Countable():ident(ncount) {
        std::cout << "0x" << std::hex << reinterpret_cast<std::uintptr_t>(this)
                  << std::dec << " Object " << ident << "/" << ncount++ << " created.\n";
    }

    ~Countable() {
        std::cout << "0x" << std::hex << reinterpret_cast<std::uintptr_t>(this)
                  << std::dec << " Object " << ident << " destroyed.\n";
    }

    std::uint64_t identifier() const { return ident; }

private:

    std::uint64_t ident;
    static std::uint64_t ncount;

};
```

Um die Klasse `Countable` in Ihrer eigenen Klasse (Konto und Kunde) zu verwenden, leiten Sie von dieser ab. Ein möglicher Ablauf kann die folgende Ausgabe auf dem Bildschirm ergeben:

```
Go to Bank and create account for Ernie.
0x7f97b7402a58 Object 0/0 created.
0x7f97b7402aa8 Object 1/1 created.
Konto (1234, Object 1) --> Kunde (Ernie, Object 0).
Kunde (Ernie, Object 0) --> Konto (1234, Object 1).
Leave the bank and delete complete account.
0x7f97b7402a58 Object 0 destroyed.
0x7f97b7402aa8 Object 1 destroyed.
```

Die blau markierten Texte werden durch die `Countable`-Klasse erzeugt. Die rot markierten Texte werden durch die print-Methode von Konto resp. Kunde erzeugt und geben den Verweis zwischen den beiden Klassen aus.

Aufgabe 1c)

Implementieren Sie die Aufgabe 1b) in dem Sie smarte Zeiger `std::shared_ptr` statt `std::unique_ptr` verwenden. Welchen Zeiger müssen Sie für den Verweis von der Kunde- auf die Konto-Klasse verwenden? Wieso darf es kein `std::shared_ptr` sein?

Die Signatur zum Erzeugen eines Kontos mit einem Kunden ergibt sich dementsprechend zu:

```
std::shared_ptr<Konto> createOwnerAccount(const std::string& name, std::uint32_t account);
```

Hinweis für Aufgabe 1: Vermeiden Sie den expliziten Aufruf von new und delete! Achten Sie darauf, dass Sie Ihren Compiler mit der Option für C++14, C++17 oder neuer aufrufen.

Aufgabe 2: Einarbeitung Visual Paradigm

In den folgenden Praktika soll ein bekanntes Brettspiel als Computeranwendung mit dem UML-Werkzeug Visual Paradigm modelliert werden. Die Aufgaben sind für alle Gruppen identisch, jedoch inhaltlich verschieden in Abhängigkeit des gewählten Brettspiels. Addieren Sie hierzu die Matrikelnummer aller Gruppenteilnehmer/innen und berechnen Sie das Modulo zu 2, wählen Sie bei 0 das Brettspiel „Malefiz“ und bei 1 das Brettspiel „Fang den Hut“. Sie können alternativ auch ein anderes Spiel wählen, welches eine vergleichbare Komplexität hat.

Aufgabe 2a)

Konfigurieren Sie das Programm Visual Paradigm in einem der beiden Poolräume SI0024 oder SI0025 entsprechend des beiliegenden Merkblatts und der Lizenzinformationen. Sie können das Programm zusätzlich (!!) auch auf Ihrem privaten Rechner installieren.

Aufgabe 2b)

Sie finden das Online-Benutzerhandbuch unter https://circle.visual-paradigm.com/docs/. Im ersten Schritt machen Sie sich mit dem allgemeinen Bedienkonzept vertraut (Chapter 3: Getting Started) und legen Sie dazu ein „Analysis Model“ für das entsprechende Spiel an. Ergänzen Sie das Model um ein Glossar (Chapter 20: Project Glossary).

Arbeiten Sie sich in das Regelwerk des ausgewählten Brettspiels ein. Ergänzen Sie das erstellte Glossar um weitere (mindestens 5) Einträge, die zur Erklärung des Brettspiels hilfreich sind. Das können auch spezielle Aktivitäten sein (z.B. „Spielfigur ziehen“).