Programmierparadigmen

Sommerersemester 2017
Martin Wittiger, Felix Krause, Timm Felden

4. Übung

Abgabe bis 14. Juni um 4:44
Beachten Sie die Abgabehinweise auf der Vorlesungswebseite!

Aufgabe 4.1 ..... 4 Punkte

Dateiname: `SummerVacation.pdf` ★★☆

Sie arbeiten an einer Software für das Management eines Sommerferienprogramms für das ferne Land Aranien. Es werden verschiedene Events veranstaltet, welche jeweils durch eine eindeutige Zeichenfolge identifiziert werden. Da Aranien eine strikt matriarchalische Gesellschaft ist, braucht ein Teilnehmer zwingend die Erlaubnis seiner Mutter, um an einem der Events teilzunehmen – es sei denn, es ist nicht bekannt, wer die Mutter des Teilnehmers ist.

Sie bekommen ein Stück Code von einem Kollegen, der das Hinzufügen von Teilnehmern zu einem Event implementiert. Teile der verwendeten Klassen sind noch nicht implementiert oder enthalten Dummy-Implementierungen; diese sind mit einem `// TODO` gekennzeichnet. Der Code soll Folgendes tun:

- Die Teilnehmer-Datenbank für das gewünschte Event öffnen.
- Sicherstellen, dass der Teilnehmer entweder die Erlaubnis seiner Mutter hat oder diese nicht bekannt ist.
- Den Teilnehmer zum Event hinzufügen.
- Sicherstellen, dass die Datenbank auch im Fehlerfall wieder geschlossen wird.
- Jedes Öffnen und Schließen der Datenbank loggen, indem die Methode `logIoEvent()` aufgerufen wird.

Die verwendeten Klassen `Person`, `ParticipantsDatabase` und `UnknownMother` finden Sie im Material zu den Übungen. Gehen Sie davon aus, dass die Methodensignaturen der verwendeten Klassen sich nicht mehr ändern. Die Implementierungen dagegen können sich durchaus noch ändern und insbesondere alle Exceptions werfen, die ihnen ihre Deklaration erlaubt.

Beantworten Sie folgende Fragen zur Methode `addParticipantToEvent`:

1. Stellt der Code tatsächlich sicher, dass eine erfolgreich geöffnete Datenbank in jedem Fall wieder geschlossen wird? Begründen Sie Ihre Antwort.
2. Listen Sie alle Exception-Klassen auf, die entweder geworfen oder durch einen Methodenaufruf in die Methode propagiert werden können. Ignorieren Sie dabei die `RuntimeException` und ihre Kindklassen (zu denen auch die explizit geworfene `IllegalArgumentException` gehört).

```java
public class SummerVacationManager {
  public static final class MotherDoesntAgree extends Exception {}

  public void addParticipantToEvent(final Person participant, final String event)
      throws MotherDoesntAgree, UnknownMother,
             ParticipantsDatabase.UnknownEvent,
             ParticipantsDatabase.EventFull {
    if (participant == null || event == null) {
      throw new IllegalArgumentException();
    }
    ParticipantsDatabase db = null;
    try {
      db = ParticipantsDatabase.open(event);
      logIoEvent();
      final Person mother = participant.getMother();
      if (!mother.agrees()) {
        throw new MotherDoesntAgree();
      }
      db.add(participant);
    } catch(UnknownMother e) {
      db.add(participant);
    } finally {
      db.close();
      logIoEvent();
    }
  }

  public void logIoEvent() {
    // TODO
  }
}
```

Listing 1: `SummerVacationManager.java`

3. Geben Sie nun an, welche der soeben gelisteten Exception-Klassen die Methode `addParticipantToEvent` verlassen können. Vergleichen Sie diese Liste mit der mit `throws` deklarierten Liste. Enthält die `throws`-Liste unnötige Elemente?

4. Der Code enthält einen Fehler, der dazu führt, dass neben der explizit geworfenen `IllegalArgumentException` eine weitere Kindklasse der `RuntimeException` innerhalb von `addParticipantToEvent` generiert und geworfen werden kann. Welche ist dies und in welchem Fall wird sie geworfen? Wie kann der Code abgeändert werden, um dieses Problem zu beheben?

Aufgabe 4.2 ..... 3 Punkte

Dateiname: `add_participant_to_event.adb` ★☆☆

Durch politische Umwälzungen im Land Aranien verliert eine ausländische Macht an Einfluss, die bisher die Benutzung von Java in der Verwaltung forciert hat. Um Unabhängigkeit zu demonstrieren, wird entschieden, dass die Software nun nach Ada

portiert werden soll. Ihr Kollege macht sich ans Werk, bemerkt dann aber, dass Ada kein Konstrukt zur Verfügung stellt, welches die Funktion von `finally` in Java repliziert. Hilfesuchend wendet er sich an Sie.

Implementieren Sie die Funktionalität der Java-Methode `addParticipantToEvent` in Ada! Ein Rumpf der Methode wird Ihnen zur Verfügung gestellt; ändern Sie die Methodensignatur nicht. Idealerweise sollten Sie einen Weg finden, den Code, der in der Java-Implementierung im `finally`-Block steht, nicht mehrfach aufzuführen.

Gehen Sie davon aus, dass der nicht-private Teil der zur Verfügung gestellten Spezifikationsdateien `log_io_event.ads`, `participants.ads` und `persons.ads` sich nicht ändern kann, der private Teil sowie die beigefügten Dummy-Implementierungen aber schon. Die Subroutinen können dieselben Exceptions werfen wie ihre Java-Gegenstücke. Ob sie den im Java-Code existierenden Fehler beheben, bleibt Ihnen überlassen (Nichtbehebung gibt keinen Punktabzug).

Hinweis: In dieser Aufgabe wird Ihnen keine Hauptroutine zur Verfügung gestellt. Sie können `add_participant_to_event.adb` zwar kompilieren, aber der Compiler wird keine ausführbare Binärdatei erzeugen, weil die Prozedur Parameter erwartet und daher als Hauptroutine ungeeignet ist. Es sei Ihnen anheimgestellt, Ihre Lösung mit einer eigenen Hauptroutine zu testen; diese sollte jedoch nicht Teil der Abgabe sein und wird auch nicht bewertet.

Aufgabe 4.3 ..... 5 Punkte

Dateiname: `loesung.adb` ★★☆

Auf der Vorlesungsseite finden Sie ein Ada-Programm in der Datei `const.adb`. Darin sind die Prozeduren `TA` – `TE` definiert. Diese ändern, so wie sie da stehen, den Systemzustand nicht. Wir nehmen allerdings an, dass sie, soweit es ihnen möglich ist, den globalen Zustand verändern, also statt dem Null-Statement diverse Operationen ausführen. Dabei führen sie nur legalen und sinnvollen Code aus, schreiben also nicht wild in den Speicher berechnen keine Adressen oder Ähnliches.

Das Programm führt wiederholt das gleiche Muster aus. Zunächst wird der globale Zustand auf einen definierten Ausgangszustand zurückversetzt. Dann werden mit Print in fünf Zeilen Informationen ausgegeben und eine der Prozeduren wird mit ausgewählten Parametern aufgerufen. Anschließend werden die fünf Zeilen erneut ausgegeben. Sie sollen nun für jeden der zehn Blöcke ermitteln, welche der Zeilen sich vor und nach dem Aufruf unterscheiden könnten. Ihre Antworten tragen Sie in die Datei `loesung.adb` ein. Diese geben Sie ab, um eine automatische Korrektur zu ermöglichen. Achten Sie darauf, dass Sie nur die vorgesehen Stellen ändern und dass sich Ihre Datei zusammen mit `print_loesung.adb` übersetzen lässt und wie erwartet Ihre Lösung ausgibt.

Aufgabe 4.4 ..... 2 Punkte

Dateiname: `paramname.pdf`

Gegeben sei das Programm in Listing 2 (`params.adb`) in einer Ada-ähnlichen Sprache, die es im Gegensatz zu Ada ermöglicht, einen Parameterübergabemechanismus durch einen Compiler-Schalter auszuwählen, so dass dann alle Parameter mit diesem Mechanismus übergeben werden. Auf alle formalen Parameter seien auch schreibende Zugriffe zulässig.

```
1  procedure Params is
2     X : Integer := 2;
3     Y : Integer := 5;
4     Z : Integer := 1;
5     A : array (1 .. 5) of Integer := (2, 3, 1, 4, 5);
6
7     procedure G (X : in out Integer; Y : in out Integer) is begin
8        A (Y) := A (A (Y));
9        Z := X;
10       Y := Y + X;
11    end G;
12
13    procedure F (X : in out Integer; Y : in out Integer) is
14       Z : Integer := 2;
15    begin
16       X := A (Z);
17       Y := A (X);
18       G (X, Z);
19       A (X) := A (Z);
20    end F;
21
22 begin
23    F (Z, A (Y));
24
25    Put (X); Put (Y); Put (Z); New_Line;
26    Put (A (1)); Put (A (2)); Put (A (3)); Put (A (4)); Put (A (5));
27 end Params;
```

Listing 2: `params.adb`

Geben Sie für die beiden folgenden Übergabemechanismen an, welche Ausgaben das Programm produziert, und erklären Sie jeweils, wie diese Ausgaben zustande kommen. Sollte das Programm auf das Array mit falschem Index zugreifen, geben Sie den Zustand der globalen Variablen zu diesem Zeitpunkt an. Gehen Sie dabei davon aus, dass jegliche Optimierungen des Compilers deaktiviert sind.

1. call-by-name (Original)
2. call-by-name (ALGOL-Regel)

Aufgabe 4.5 ..... 4 Punkte

Dateiname: `sichtbarkeit.adb` ★★☆

Nachfolgend finden Sie einige Aussagen über Sichtbarkeit und Zugriffsrechte. Im Rahmen dieser Aussagen gelten folgende Definitionen:

- Eine *Komponente* einer Entität *A* ist jede Entität *B*, die genau eine Hierarchiestufe unterhalb von *A* liegt und Teil von *A* ist. Beispiele:
  - `java.lang.String` ist eine Komponente von `java.lang`, nicht jedoch von `java`.
  - `charAt` ist eine Komponente von `java.lang.String`, nicht jedoch von `java.lang`.
- Ist eine Komponente *öffentlich*, so bedeutet es, dass auf sie an jeder beliebigen Stelle im Quellcode zugegriffen werden kann.
- Auf eine Komponente darf *zugegriffen* werden, wenn der Name der Komponente als Identifier benutzt werden darf, um die Komponente zu referenzieren. Besitzt ein Ada-Record beispielsweise ein Feld `I : Integer`, so ist für eine Variable `V` dieses Record-Typs der Ausdruck `V.I` ein Zugriff. Die Frage des Zugriffs setzt immer voraus, dass die darüberliegende Struktur (das Paket, eine Instanz der Klasse / des Records, etc.) verfügbar ist. *Ich kann nicht auf `Put_Line` zugreifen, wenn das Paket `Ada.Text_IO` nicht mit `with` importiert wurde* ist also kein Gegenargument zur Aussage *`Ada.Text_IO.Put_Line` ist eine öffentliche Subroutine*.
- Ein Paket ist ein *Kindpaket* eines anderen Pakets, wenn es dessen kompletten Namen als Präfix im eigenen Namen enthält. Beispielsweise ist das Paket `java.lang` ein Kindpaket des Pakets `java`.

Entscheiden Sie für jede Aussage, ob diese wahr oder falsch ist. Ihre Antworten tragen Sie in die Datei `sichtbarkeit.adb` ein. Diese geben Sie ab, um eine automatische Korrektur zu ermöglichen. Achten Sie darauf, dass Sie nur die vorgesehen Stellen ändern und dass sich Ihre Datei zusammen mit veranstaltet `print_sichtbarkeit.adb` übersetzen lässt und wie erwartet Ihre Lösung ausgibt.

1. Eine Java-Klasse *K* definiert ein privates Feld *f* mit dem Modifier `private`. Eine Methode dieser Klasse hat die Parameterliste `(K param)`. Auf `param.f` darf innerhalb dieser Methode nicht zugegriffen werden, weil das Feld privat ist und zu einem fremden Objekt der Klasse `K` gehört.
2. Hat eine Java-Methode keinen expliziten Sichtbarkeits-Modifier, so darf an jeder Stelle innerhalb desselben Pakets auf diese Methode zugegriffen werden, aber nicht außerhalb des Pakets.
3. Werden in Ada in einem Paket zwei Record-Typen deklariert, so kann nicht verhindert werden, dass Subroutinen in diesem Paket, die inhaltlich zum ersten Typ gehören, auf Felder des zweiten Typs zugreifen dürfen. In Java hingegen ist das mit zwei Klassen im selben Paket möglich.

4. Kindpakete in Ada dürfen ausschließlich auf den öffentlichen Teil ihrer Elternpakete zugreifen.

5. Es ist möglich, eine Methode in Java so zu deklarieren, dass auf sie ausschließlich im Code dieser Klasse sowie in allen Klassen, die von dieser Klasse erben, zugegriffen werden kann.

6. Möchte man in Ada drei Integerwerte `I`, `J` und `K` zu einem Record zusammenfassen, so ist es nicht möglich, `I` öffentlich verfügbar zu machen, aber `J` und `K` nicht-öffentlich, ohne neben dem neuen Record-Typ zumindest einen zusätzlichen Typen zu benutzen.

7. Java bietet eine Möglichkeit, Felder einer Klasse so zu deklarieren, dass sie zwar nicht öffentlich sind, aber auf sie überall innerhalb des eigenen Pakets sowie dessen Kindpaketen zugegriffen werden kann.

8. Ada bietet die Möglichkeit eines `private package`, um die Sichtbarkeit des Pakets als Ganzes zu beschränken. Java kennt nichts Vergleichbares; auf alle mit `public` deklarierten Komponenten eines Pakets kann immer von jeder beliebigen Stelle des Quellcodes aus zugegriffen werden (sofern an dieser Stelle eine Referenz auf den Typ der Komponente allgemein erlaubt ist).