Hochschule Worms
University of Applied Sciences
Fachbereich Informatik
Prof. Dr.-Ing. H. Thielen

Aufgaben der Modulprüfungen im Sommersemester 2020

Angewandte Informatik

Modul: **Testen**

HIS-POS: **124 (Angewandte Informatik B.Sc. PO 2012)**
131 (Angewandte Informatik [– dual] B.Sc. FPO 2018)

Zeit: **120 Minuten**

Hilfsmittel: **Laborrechner mit Linux**

Bitte Studentenausweis bereitlegen!

Nachname (Familienname):			
Vorname:			
Matrikelnummer:		Rechnernummer:	

Git-Hashwert (erste 11 Stellen) des auf den Bitbucket-Server gepushten Lösungs-Commits für Aufgabe 4:

Ich versichere, dass mein abgegebener Lösungs-Commit den obigen Git-Hashwert besitzt, und dass ich diesen Commit per Git-Push fehlerfrei auf den Bitbucket-Server übertragen habe.

Unterschrift: _____

ERREICHTE PUNKTZAHL: _____ *von max. 100 Punkten*

NOTE: _____ *Handzeichen:* _____

Zweitprüfer: _____

Bewertung: ab 50 Punkten Note 4,0; ab 65 P. Note 3,0; ab 80 P. Note 2,0; ab 95 P. Note 1,0

1. Grundlagen (25 Punkte)

Die Fragen zum ersten Abschnitt finden Sie im gewohnten Moodle-Kurs zum Modul Software-Qualität: https://moodle.hs-worms.de/moodle/course/view.php?id=269. Sie können Ihre Zeit frei einteilen und beliebig zwischen den verschiedenen Aufgaben wechseln. ACHTUNG: Sie müssen in Moodle vor Ende der Bearbeitungszeit der Prüfung den „Abgeben“-Knopf drücken!

2. Statische Tests (15 Punkte)

2.1. Beschreiben Sie, wie die verschiedenen Aktivitäten bei der Erstellung und Bearbeitung eines Pull-Requests in Bitbucket den Phasen eines formalen Reviews zugeordnet werden können. Erläutern Sie ggf. auch Unterschiede!
(4P.)

2.2. Gegeben ist folgender Beispielcode in C++. Beschreiben Sie, welche Datenflussanomalien hier auftreten (jeweils Anomalie, Zeile/n und Variable angeben):
(4P.)

```
1 int max (int a, int b ) {
2     int ret;
3     if( a>ret )
4           ret = a;
5     else if( b>a )
6           a = b;
7     return ret;
8 }
```

2.3. Zeichnen Sie den Kontrollflussgraphen zu folgendem Code-Stück! Bezeichnen Sie dabei die Elemente im Graphen so, dass die Zuordnung zu den Zeilennummern des Codes nachvollziehbar ist.
(7P.)

```
1  void SpielAblauf::spiele() {
2      Schachfeld * erstesFeld = generator.erzeugeSchachfeld("e6");
3      string const ziel = "f3";
4
5      Schachfeld * aktuellesFeld = erstesFeld;
6      while( aktuellesFeld->toString() != ziel && in ) {
7          out << "Aktuelle Position: " << aktuellesFeld->toString()
8              << ", Ziel: " << ziel << endl;
9
10         aktuellesFeld = springerZugEingabe();
11         if( !aktuellesFeld->valid() )
12             break;
13
14         if( aktuellesFeld->toString() != ziel ) {
15             out << "Ziel " << ziel << " noch nicht erreicht.\n";
16         }
17         erstesFeld->push_back(aktuellesFeld);
18     }
19
20     delete erstesFeld;
21 }
```

3. Dynamische Tests (35 Punkte)

3.1. Nennen Sie vier verschiedene Whitebox-Verfahren und beschreiben Sie kurz deren wesentliche Eigenschaften!
(4P.)

3.2. Erläutern Sie stichpunktartig das Vorgehen bei der Erstellung von Testfällen mittels der Grenzwertanalyse:
(insgesamt 15P.)

a) Wie ist dabei das vollständige Vorgehen zum Finden der Testfälle? (10P.)

b) Was muss in der Definition jedes einzelnen Testfalls festgelegt sein? (4P.)

c) Nennen Sie zwei für Grenzwertanalyse spezifische Testendekriterien! (1P.)

3.3. Gegeben ist die Klasse `Duration` zur Bearbeitung des Datums von TODO-Listen (vgl. Listing auf der übernächsten Seite).

Mit der Klasse **`Duration`** können **Zeitdauer**-Objekte erzeugt und genutzt werden. Dazu speichert die Klasse als Attribute eine Anzahl von **Tagen**, **Wochen**, **Monaten** und **Jahren**.

Damit `Duration`-Objekte besser vergleichbar sind, sind die Objekte immer teilweise normiert: Z.B. sind ja 7 Tage exakt gleich 1 Woche, und 8 Tage sind exakt gleich 1 Woche + 1 Tag.

Die folgenden beiden Objekte sind also identisch:

```
Duration twentyDays = Duration(20,0,0,0);
Duration twoWeeksAndSixDays(6,2,0,0);
```

Genauso verhält es sich zwischen Monaten und Jahren: 12 Monate sind exakt gleich 1 Jahr, die beiden folgenden Objekte also auch identisch:

```
Duration twentyFiveMonths=Duration(0,0,25,0);
Duration twoYearsAndOneMonth(0,0,1,2);
```

Vielfache von 7 Tagen werden also immer in eine entsprechende Anzahl von Wochen umgerechnet, und Vielfache von 12 Monaten in eine entsprechende Anzahl von Jahren. Die Rückgabewerte der Methode `Duration::days()` liegen also immer im Bereich von 0-6, und die Rückgabewerte der Methode `Duration::months()` liegen immer im Bereich 0-11.

Es existiert *keine* solche Beziehung für die Angabe von Wochen, da ein Monat in der Regel etwas mehr als 4 Wochen enthält, und auch ein Jahr etwas mehr als 52 Wochen enthält.

Generell werden momentan keine negativen Werte für Tage, Wochen, Monate und Jahre unterstützt. Deshalb wird im *Konstruktor* eine Exception vom Typ `std::invalid_argument` ausgelöst, wenn ein negativer Wert an den Konstruktor übergeben wird.

Zur Berechnung von Bankzinsen werden mit der Methode **`Duration::asBankDays()`** Zeitdauern in Zinstage umgerechnet. Dabei wird für diese Aufgabe angenommen, dass die Banken vereinfacht umrechnen (was sie auch tun - aber vielleicht nicht genau so):

- eine Woche hat 7 Tage (wie immer),

- ein Monat hat immer 30 Zins-Tage (Tage mit 28/29/31 Tagen werden also nicht genau umgerechnet),
- ein Jahr hat immer 360 Zins-Tage (= 12 Monate * 30 Zinstage).

Zur Sicherheit gegen falsche Zins-Berechnungen wird (für diese Aufgabe) festgelegt, dass keine Zeitdauer von mehr als 100 Jahren in Zins-Tage umgerechnet werden darf. Anderenfalls wird eine Exception vom Typ `std::out_of_range` ausgelöst.

Das Verhalten der Methode Methode **`Duration::asBankDays()`** (**Zeile 45 im Listing** auf der nächsten Seite) soll mit Testfällen geprüft werden, die mittels Grenzwertanalyse erstellt werden sollen.
(insgesamt 16P.)

a) Bestimmen Sie die nötigen Parameter und Grenzwerte für die Tests von **`Duration::asBankDays()`**. (4P.)

b) Wie viele „Gutfall“-Tests und wie viele „Fehlerfall“-Tests leiten Sie daraus ab (mit Begründung)? (2P.)

c) Gutfall-Tests: Definieren Sie 8 Gutfall-Tests für die Methode **`Duration::asBankDays()`**.(8P.)

d) Fehlerfall-Tests: Definieren Sie 2 Fehlerfall-Tests für die Methode **`Duration::asBankDays()`**. (2P.)

```cpp
/// A duration, defined as simple constant object:
/// It cannot be changed after creation.
///
/// Duration is used e.g. for CalendarDate.add(). It contains
/// publicly accessible methods to get the values of days, weeks, months
/// and years.
///
/// Only the number of days and the number of months is normalized during
/// creation:
/// - every multiple of 7 days adds one week
/// - every multiple of 12 months adds one year
///
/// Example:
/// Duration{8,1,27,2} is automatically converted to Duration{1,2,3,4}:
/// 8 days => 1 week + 1 day
/// 27 months => 2 years + 3 months
///
/// That means that the values are converted into the following ranges
/// during creation:
/// days(): 0-6 (Values >= 7 increase the number of weeks accordingly)
/// months(): 0-11 (Values >= 12 increase the number of years accordingly)
class Duration {
    public:
        /// Default initialization is with zero values.
        Duration();
        /// Days and Months are normalized, see comments above.
        /// NOTE: Negative numbers are not yet supported and lead to an
        /// exception of type std::invalid_argument.
        Duration( int days, int weeks, int months, int years );

        int days() const;
        int weeks() const;
        int months() const;
        int years() const;

        /// Return the number of "bank days" of this Duration object.
        /// We assume that the bank does a simplified computation
        /// of Duration days for the calculation of interest ("Zinsen"):
        /// - a week has 7 days (as usual)
        /// - a month has exactly 30 days (which is not accurate)
        /// - a year has exactly 360 days (which is not accurate)
        /// For security reasons, a Duration longer than 100 years may not
        /// be allowed for calculation of interest, so an exception of type
        /// std::out_of_range shall be omitted then.
        int asBankDays() const;

        /// Operators
        bool operator ==(const Duration & other) const;
        bool operator !=(const Duration & other) const;
        Duration operator +(const Duration & rhs) const;
        /// ...
};
```

4. Praktische Aufgabe (25 Punkte)

Die in Aufgabe 3.4 c) und d) definierten Tests sollen implementiert werden.

Alle Quellen sind verfügbar im Bitbucket-Projekt "**`Testen Pruefung`**", Repository "**`testen-prf-20s`**", Branch "**`master`**":

https://atlas.ai.it.hs-worms.de/bitbucket/projects/TSTPRF/repos/testen-prf-20s

Als Hilfsmittel liegen ein Beispiel-Test sowie die Beschreibung der Aufgabe in der Datei **`src/unittest/aufgabe.cpp`** vor.

Folgende Aufgaben sind durchzuführen:

1. Erzeugen Sie einen **Fork** des oben genannten Repositorys in ihr persönliches Bitbucket-Projekt
2. **Clonen** Sie das Repository auf den Rechner im Labor
3. Erstellen Sie die Tests entsprechend Aufgabe 3.4 c) und d)
4. **Ändern** Sie möglichst den **Git-Commiter auf Ihren Namen und Ihre E-Mail** (es soll also möglichst nicht „labor“ als Commiter auftauchen).
5. **Committen** Sie die Änderungen im lokalen Repository (Laborrechner)
6. **Pushen** Sie die Änderungen zu ihrem Bitbucket-Repository.
7. Schreiben Sie die ersten 11 Zeichen der **git-Commit-Nummer** Ihrer Lösung auf das **Deckblatt** (ablesen in Ihrem Bitbucket-Repository, Menü-Punkt "Navigation→Commits")

ACHTUNG: Ohne Push kann nicht bewertet werden!

Falls durch Ihre Tests Fehler gefunden werden, müssen Sie den Quellcode der Applikation nicht korrigieren; Sie sollten aber möglichst sicher sein, dass nicht Ihr Test den Fehlerzustand enthält.

Minimalanforderungen für die Punktevergabe:

- Sie haben eine Änderung durchgeführt.
- Das Unit-Test-Programm ist mit Ihren Tests übersetzbar, linkbar und ausführbar.
- Sie haben Ihre Lösung in Ihr Git-Repository des Bitbucket-Servers gepusht.
- Sie haben die Commit-Nummer Ihrer Lösung auf das Deckblatt geschrieben.

Sie können beliebig oft in Ihr Git-Repository des Bitbucket-Servers pushen; es gilt der letzte Push vor Ablauf der Bearbeitungszeit, dessen Commit-Nummer Sie auf dem Deckblatt eingetragen haben.

Bewertungskriterien:

- Vollständigkeit des Testumfangs entsprechend Anforderung oben
- Korrektheit der Testfälle
- Struktur der Testfälle (wartbar kodiert, datengetrieben falls sinnvoll, etc.)