Tutorium 13: Design by Contract

Paul Brinkmeier

28. Januar 2020

Tutorium Programmierparadigmen am KIT

Design by Contract

ProPa-Stoff zu Design by Contract:

- Grundlagen: Pre-/Postconditions, Caller, Callee
  - A.K.A.: Vor-/Nachbedingungen, Aufrufer, Aufgerufener
- JML (Java Modeling Language):
  - `@ requires`
  - `@ ensures` (mit `\old` und `\result`)
  - `@ invariant`
  - `/*@ pure @*/`, `/*@ nullable @*/`, `/*@ spec_public @*/`
  - Quantoren: `\forall`, `\exists`
  - Liskovsches Substitutionsprinzip

Design by Contract

ProPa-Stoff zu Design by Contract:

- Grundlagen: Pre-/Postconditions, Caller, Callee
  - A.K.A.: Vor-/Nachbedingungen, Aufrufer, Aufgerufener
- JML (Java Modeling Language):
  - `@ requires`
  - `@ ensures` (mit `\old` und `\result`)
  - `@ invariant`
  - `/*@ pure @*/`, `/*@ nullable @*/`, `/*@ spec_public @*/`
  - Quantoren: `\forall`, `\exists`
  - Liskovsches Substitutionsprinzip
  - Java-`assert`-Keyword

JML-Klausuraufgabe

JML-Klausuraufgabe

Klausur 19SS, Aufgabe 6d (3P.)

```
class MaxAbsCombinator {
    //@ requires left != Integer.MIN_VALUE;
    //@ requires right != Integer.MIN_VALUE;
    //@ ensures \result <= left || \result <= right;
    //@ ensures \result >= left && \result >= right;
    int combine(int left, int right) {
        return Math.max(Math.abs(left), Math.abs(right));
    }
}
```

(d) Der Vertrag der Methode `combine` wird *vom Aufgerufenen* verletzt. Begründen Sie dies und geben Sie an, wie die verletzte Nachbedingung angepasst werden könnte.

JML-Klausuraufgabe

Klausur 19SS, Aufgabe 6e (2P.)

```
class MaxAbsCombinator {
    //@ requires left != Integer.MIN_VALUE;
    //@ requires right != Integer.MIN_VALUE;
    //@ ensures \result <= left || \result <= right;
    //@ ensures \result >= left && \result >= right;
    int combine(int left, int right) {
        return Math.max(Math.abs(left), Math.abs(right));
    }
}

new MaxAbsCombinator().combine(
    random.nextInt(),
    random.nextInt());
```

(d) Wird der Vertrag hier *vom Aufrufer* erfüllt? Begründen Sie kurz.

JML

@ requires

```
//@ requires b != 0;
int divide(int a, int b) {
    return a / b;
}
```

- `@ requires` definiert eine Vorbedingung für eine Methode.
- Vorbedingungen müssen vom Aufrufer erfüllt werden.

@ ensures

```
//@ ensures \result.length() == s.length();
String[] reverse(String [] s) { ... }

//@ requires amount > 0;
//@ ensures balance > \old(balance);
void deposit(int amount) {
    this.balance += amount;
}
```

- `@ ensures` definiert eine Nachbedingung für eine Methode.
- Nachbedingungen müssen vom Aufgerufenen erfüllt werden.
- Mit `\old` und `\result` werden Beziehungen zwischen Ursprungszustand, Rückgabewert und neuem Zustand eingeführt.

@ invariant

```
class FixedSizeList<A> {
    //@ invariant elementCount <= elements.length;
    A[] elements;
    int elementCount;
}
```

- `@ invariant` definiert Invarianten für eine Klasse.
- Diese können bspw. wiederverwendet werden, um Vorbedingungen für Methoden zu erfüllen.

/*@ pure @*/

```
class ResizingArray<A> {
    private A[] elements;
    private int elementCount;
    /*@ pure @*/ public int getElementCount();

    //@ ensures getElementCount() ==
    //@         \old(getElementCount()) + 1;
    public void add(A element) { ... }
}
```

- Verträge sind implizit `public`.
  $\rightsquigarrow$ `private`-Attribute nicht verwendbar
- Um Getter-Funktionen in Verträgen nutzen zu können, müssen diese frei von Seiteneffekten und mit `/*@ pure *@/` markiert sein.

/*@ spec_public @*/

```
class ResizingArray<A> {
    private A[] elements;
    private /*@ spec_public @*/ int elementCount;

    //@ ensures elementCount ==
    //@         \old(elementCount) + 1;
    public void add(A element) { ... }
}
```

- Alternative: `private`-Attribute als `/*@ spec_public @*/` markieren.
- Immer noch `private`, können vom Checker aber trotzdem gesehen werden.

Quantoren, logische Operatoren

```
/*@ requires \forall int i;
    0 <= i && i < xs.length;
    xs[i] != null;
    ensures xs.length == 0 ==> \result == 0;
  @*/
int totalLength(String[] xs) {
    ...
}
```

- Für das Arbeiten mit Aussagen in Verträgen gibt es ein paar Helferchen:
  - `\forall<decl>; <cond>; <expr>`
  - `\exists<decl>; <cond>; <expr>`
  - `<cond> ==> <expr>`

Übungsaufgabe: RPN-Taschenrechner

Reverse Polish Notation

$$(2 + 4) * (10 - 3)$$

- „Natürliche“ Darstellung: Infix-Notation
  - „Problem“: Man braucht Klammern!

Reverse Polish Notation

$$(2 + 4) * (10 - 3)$$

- „Natürliche“ Darstellung: Infix-Notation
  - „Problem“: Man braucht Klammern!
- Alternative: Polnische (Präfix-)Notation
  - `* + 2 4 - 10 3`
  - `multipy (add 2 4) (subtract 10 3)` ($\rightsquigarrow$ LISP)

Reverse Polish Notation

$$(2 + 4) * (10 - 3)$$

- „Natürliche“ Darstellung: Infix-Notation
  - „Problem“: Man braucht Klammern!
- Alternative: Polnische (Präfix-)Notation
  - `* + 2 4 - 10 3`
  - `multipy (add 2 4) (subtract 10 3)` ($\rightsquigarrow$ LISP)
- Einfach zu implementieren: Umgekehrte polnische Notation
  - `10 3 - 2 4 + *`
  - Links nach rechts durchgehen
  - Zahlen werden auf einen Stack gelegt
  - Operatoren nehmen Operanden vom Stack, legen Ergebnis auf den Stack

Reverse Polish Notation

$$(2 + 4) * (10 - 3)$$

- „Natürliche“ Darstellung: Infix-Notation
  - „Problem“: Man braucht Klammern!
- Alternative: Polnische (Präfix-)Notation
  - `* + 2 4 - 10 3`
  - `multipy (add 2 4) (subtract 10 3)` ($\rightsquigarrow$ LISP)
- Einfach zu implementieren: Umgekehrte polnische Notation
  - `10 3 - 2 4 + *`
  - Links nach rechts durchgehen
  - Zahlen werden auf einen Stack gelegt
  - Operatoren nehmen Operanden vom Stack, legen Ergebnis auf den Stack
- Bspw. alte Taschenrechner, Forth, Shunting-yard-Algorithmus

RPN-Taschenrechner

- `demos/java/rpncalculator/RpnCalculator.java`
- Überprüft manuell den Vertrag von `pop()`.

RPN-Taschenrechner

- `demos/java/rpncalculator/RpnCalculator.java`
- Überprüft manuell den Vertrag von `pop()`.
- Überprüft den Vertrag mit OpenJML:

  ```
  java -jar oj/openjml.jar -exec <solver> -esc <.java> -method
  pop
  ```

RPN-Taschenrechner

- `demos/java/rpncalculator/RpnCalculator.java`
- Überprüft manuell den Vertrag von `pop()`.
- Überprüft den Vertrag mit OpenJML:

  ```
  java -jar oj/openjml.jar -exec <solver> -esc <.java> -method pop
  ```

- Überlegt euch einen entsprechenden Vertrag für `push()`.

RPN-Taschenrechner

- `demos/java/rpncalculator/RpnCalculator.java`
- Überprüft manuell den Vertrag von `pop()`.
- Überprüft den Vertrag mit OpenJML:

  ```
  java -jar oj/openjml.jar -exec <solver> -esc <.java> -method pop
  ```

- Überlegt euch einen entsprechenden Vertrag für `push()`.
- Identifiziert seiteneffektfreie Methoden und markiert sie mit `/*@ pure @*/`.

RPN-Taschenrechner

- `demos/java/rpncalculator/RpnCalculator.java`
- Überprüft manuell den Vertrag von `pop()`.
- Überprüft den Vertrag mit OpenJML:

  ```
  java -jar oj/openjml.jar -exec <solver> -esc <.java> -method pop
  ```

- Überlegt euch einen entsprechenden Vertrag für `push()`.
- Identifiziert seiteneffektfreie Methoden und markiert sie mit `/*@ pure @*/`.
- Überlegt euch Invarianten für `elementCount` und `stack.length`.
- Überpüft dann zusäztlich die Methode `execute(List<Token>)`.

Mehr Java

Übungsaufgabe 1

- `demos/java/bruteforce/Bruteforce.java`
- `new Bruteforce(w).countZeroStartSequential(k)` berechnet die Anzahl der `w`-stelligen kleinbuchstabigen Worte, deren SHA-256-Hash mit `k` 0-Bytes beginnt.
- Ist dieses Problem parallelisierbar?

Übungsaufgabe 1

- `demos/java/bruteforce/Bruteforce.java`
- `new Bruteforce(w).countZeroStartSequential(k)` berechnet die Anzahl der `w`-stelligen kleinbuchstabigen Worte, deren SHA-256-Hash mit `k` 0-Bytes beginnt.
- Ist dieses Problem parallelisierbar?
- $\rightsquigarrow$ Ja; daten-/taskparallel?

Übungsaufgabe 1

- `demos/java/bruteforce/Bruteforce.java`
- `new Bruteforce(w).countZeroStartSequential(k)` berechnet die Anzahl der `w`-stelligen kleinbuchstabigen Worte, deren SHA-256-Hash mit `k` 0-Bytes beginnt.
- Ist dieses Problem parallelisierbar?
- $\rightsquigarrow$ Ja; daten-/taskparallel?
- $\rightsquigarrow$ Datenparallel; Eingabemenge ist auf identische Ausführungsträger (Threads) aufteilbar.

Übungsaufgabe 1

- `demos/java/bruteforce/Bruteforce.java`
- `new Bruteforce(w).countZeroStartSequential(k)`
  berechnet die Anzahl der `w`-stelligen kleinbuchstabigen Worte, deren SHA-256-Hash mit `k` 0-Bytes beginnt.
- Ist dieses Problem parallelisierbar?
- $\rightsquigarrow$ Ja; daten-/taskparallel?
- $\rightsquigarrow$ Datenparallel; Eingabemenge ist auf identische Ausführungsträger (Threads) aufteilbar.
- Implementiert:
  `new Bruteforce(w).countZeroStartParallel(k, n)`,
  wobei `n` die Zahl der verwendeten Threads bezeichnet.
- Vergleicht `Sequential` mit `Parallel`
  für $\texttt{w} \in \{5, 6\}$, $\texttt{k} = 2$, $\texttt{n} \in \{1, 2, 4\}$.

Übungsaufgabe 2

- `demos/java/set/Set.java`
- Behebt alle Compiler- und Laufzeitfehler in der Klasse `Set`.

Übungsaufgabe 2

- `demos/java/set/Set.java`
- Behebt alle Compiler- und Laufzeitfehler in der Klasse `Set`.
- Behebt ggf. auftretende JML-Warnings.
  - Geht auch online:
    https://www.rise4fun.com/OpenJMLESC/

Übungsaufgabe 2

- `demos/java/set/Set.java`
- Behebt alle Compiler- und Laufzeitfehler in der Klasse `Set`.
- Behebt ggf. auftretende JML-Warnings.
  - Geht auch online:
    https://www.rise4fun.com/OpenJMLESC/
- Fügt je mind. eine (sinnvolle) Vor- und Nachbedingung zu folgenden Methoden hinzu:
  - `size()`
  - `isEmpty()`
  - `add()`
  - `contains()`
  - `getElements()`

Gefilterte Summe (18SS)

```
xs: List<Integer>
foreach x of xs:
    threads.add(new Thread () -> {
        if x < 10 then element else 0
    })
foreach t of threads:
    result += t.join()
```

- Wie lässt sich ein „Speedup“ von 0.005 gegenüber einer Lösung ohne Threads erklären, wenn $|\texttt{xs}| \approx 10^7$ (3P.)?
- Ersetzen Sie [die entsprechenden Zeilen] durch eine Implementierung mit möglichst hohem Speedup (7,5P.).
- Geben Sie eine Implementierung mittels parallelen Streams an (2,5P.).
- Zusätzlich: DbC-Fragen für 5P.

Ende

Ende

- Im Campus-System kann man sich bis zum 17.03. für die ProPa-Klausur anmelden
- Rückmelden bis zum 15.02.