Prüfung Informatik 2

Hochschule Reutlingen - Reutlingen University
Fakultät Informatik
Studiengang Medien- und Kommunikationsinformatik
akkreditiert von der ASIIN

mki-Bachelor 2
Sommersemester 2009

☞

Musterlösung

Aufgaben sind in Times-, *Lösungen in blauer Monotype-Corsiva-Schrift gehalten.*

Prüfer: Prof. Dr. Karlheinz Hug

Prüfungstermin und Ort: Freitag, 10. Juli 2009, 10^{30} bis 12^{30} Uhr, Aula

Prüfungsdauer: 120 Minuten

Zugelassene Hilfsmittel: Alle (Literatur, Skripten, Übungsmaterial,...)

Anzahl der Aufgabenseiten: 13

Aufgabe	Punkte		Aufgabe	Punkte	
	möglich	erreicht		möglich	erreicht
1	26		5	16	
2	38		6	34	
3	15		7	10	
4	25		**Summe erreicht:**		
Summe möglich:		**164**	**Note:**		

- Zum Bestehen der Prüfung sind **60**, für die Note „1.0“ **120** Punkte erforderlich.
- Lassen Sie die erhaltenen Blätter zusammengeheftet und geben Sie alle Blätter geheftet wieder ab, sonst droht Abzug von Punkten!
- Bearbeiten Sie die Aufgaben auf dazu freigelassenen Stellen!
- Der Platz reicht normalerweise; vermeiden Sie Extrablätter!
- Teilaufgaben sind oft unabhängig voneinander lösbar.

✌ **Viel Erfolg!**

Aufgabe 1 Keller mit Warteschlange realisieren

(Punkte: 26 |.....)

Eine **Kellerklasse** (Stapelklasse) mit der bekannten Schnittstelle

```
StackDesc = RECORD
   (IN s : StackDesc) Count () : INTEGER, NEW;
   (VAR s : StackDesc) Init, NEW;
   (IN s : StackDesc) IsEmpty () : BOOLEAN, NEW;
   (IN s : StackDesc) Item () : Element, NEW;
   (VAR s : StackDesc) Put (x : Element), NEW;
   (VAR s : StackDesc) Remove, NEW;
END;
```

und **Last-In-First-Out**-Semantik ist zu realisieren. Als Objektdaten in StackDesc ist *nur ein* Objekt einer **Warteschlangenklasse** mit der bekannten Schnittstelle

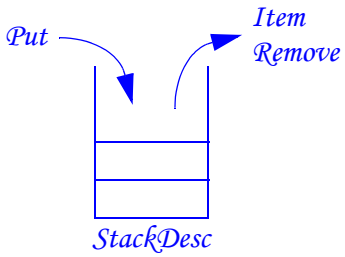
```
QueueDesc = RECORD
   (IN q : QueueDesc) Count () : INTEGER, NEW;
   (VAR q : QueueDesc) Init, NEW;
   (IN q : QueueDesc) IsEmpty () : BOOLEAN, NEW;
   (IN q : QueueDesc) Item () : Element, NEW;
   (VAR q : QueueDesc) Put (x : Element), NEW;
   (VAR q : QueueDesc) Remove, NEW;
END;
```

und **First-In-First-Out**-Semantik zugelassen. Element ist ein beliebiger Elementtyp.

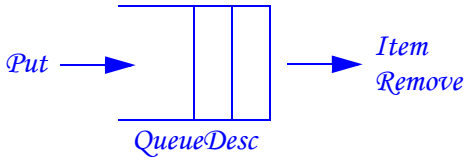
Visualisieren und verbalisieren Sie die **Entwurfsideen** und skizzieren Sie die **Algorithmen** der Schnittstellenoperationen der Kellerklasse!

Entwurfsideen

Ziel ist, die Schnittstelle eines Kellers zu realisieren:



Verlangt ist die Implementation mit einem Warteschlangenobjekt:



Bei Init, Count, IsEmpty, Item, Remove muss das Kellerobjekt den Auftrag jeweils nur an sein Warteschlangenobjekt weiterleiten.

Schwierig ist Put: In die Warteschlange kommt das Element hinten rein, im Keller muss es aber vorne stehen. Deshalb sind nach dem Put auf die Warteschlange die Count - 1 vorderen Elemente vorne wegzunehmen und hinten wieder einzufügen. So wird das letzte zum ersten Element.

Beispiele:

leer → Put(1) → 1.

1 → Put(2) → 2 1 → 1 2.

1 2 → Put(3) → 3 1 2 → 2 3 1 → 1 2 3.

Daten

```
TYPE  StackDesc* =  RECORD
                       q : QueueDesc
                    END;
```

Algorithmen

Stack.Init:

```
q.Init
```

Stack.Count:

```
RETURN q.Count
```

Stack.IsEmpty:

```
RETURN q.IsEmpty
```

Stack.Item:

```
PRE ~IsEmpty
RETURN q.Item
```

Stack.Remove:

```
PRE ~IsEmpty
q.Remove
```

Stack.Put (x):

```
q.Put (x)
q.Rotate
```

Private Prozedur zum Verschieben der vorderen Count - 1 Elemente nach hinten:

Stack.Rotate:

```
VAR i : INTEGER
FOR i := 2 TO q.Count DO
    q.Put (q.Item)
    q. Remove
END
```

Metainformation des Prüfers

Häufiger Fehler ist ein falscher Zählbereich der Art „um 1 daneben“; oft wird das eingefügte Element fälschlich mitrotiert.

Aufgabe 2 Parser mit rekursivem Abstieg entwerfen

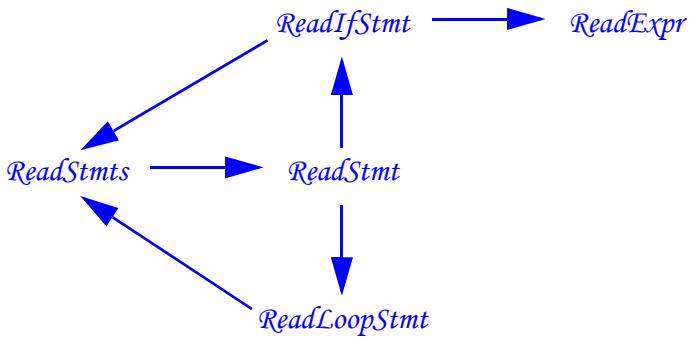
(Punkte: 38 |.....)

Zur Sprache, deren Syntax teilweise durch die EBNF-Regeln

```
Stmts     =  Stmt { ";" Stmt } .
Stmt      =  IfStmt | LoopStmt | EXIT .
IfStmt    =  IF Expr THEN Stmts [ ELSE Stmts ] END .
LoopStmt  =  LOOP Stmts END .
```

gegeben ist, ist ein Zerteiler mit rekursivem Abstieg zu entwerfen. Stellen Sie die statischen **Aufrufbeziehungen** der für die oben vorkommenden fünf Nichtterminale benötigten Prozeduren in einem Diagramm dar und skizzieren Sie die **Algorithmen** der Prozeduren zu den obigen vier Regeln mit den unten angegebenen Vereinbarungen!

Aufrufbeziehungen



Lieferanten: Scanner und Fehlerbehandlung

symbol	Abfrage: zeigt zuletzt gelesenes Symbol an mit möglichen Werten semicolon, exit, if, then, else, end, loop, endOfInput
next	Aktion: liest nächstes Symbol und weist symbol den passenden Wert zu
error	Aktion: meldet Syntaxfehler

Algorithmen

```
ReadStmts:
    ReadStmt
    WHILE symbol = semicolon DO
        next
        ReadStmt
    END

ReadStmt:
    IF symbol = if THEN
        ReadIfStmt
    ELSIF symbol = loop THEN
        ReadLoopStmt
    ELSIF symbol = exit THEN
        next
    ELSE
        error
    END
```

ReadIfStmt:

IF symbol = if THEN next ELSE error END
ReadExpr
IF symbol = then THEN next ELSE error END
ReadStmts
IF symbol = else THEN
next
ReadStmts
END
IF symbol = end THEN next ELSE error END

ReadLoopStmt:

IF symbol = loop THEN next ELSE error END
ReadStmts
IF symbol = end THEN next ELSE error END

Metainformation des Prüfers

Typische Fehler:

- *next an der falschen Stelle: zu viel oder zu wenig*
- *error an der falschen Stelle: zu viel oder zu wenig*
- *falsche Schachtelung, zu tief geschachtelt*

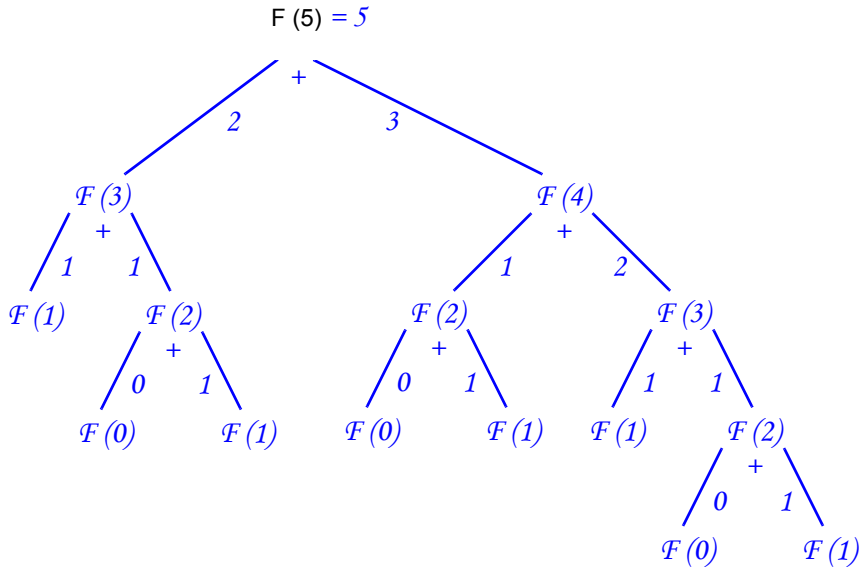
Aufgabe 3 Ablauf einer rekursiven Funktion verfolgen

(Punkte: 15 |.....)

Verfolgen Sie die Ausführung der rekursiven Funktion F beim unten gegebenen Aufruf mit aktuellem Parameterwert 5; zeichnen Sie dazu den **Aufrufbaum** und schreiben Sie die Ergebniswerte an die Aufrufkanten! Füllen Sie dann die **Wertetabelle**!

```
PROCEDURE F (n : INTEGER) : INTEGER;
BEGIN
   ASSERT ((0 <= n) & (n <= 42), BEC.precondPar1InRange);
   IF n < 2 THEN
      RETURN n
   ELSE
      RETURN F (n - 2) + F (n - 1)
   END
END F
```

Aufrufbaum



Wertetabelle

n	0	1	2	3	4	5	6	7	8
F (n)	*0*	*1*	*1*	*2*	*3*	*5*	*8*	*13*	*21*

Was ist **problematisch** an der rekursiven Implementation von F?

Argumente

- *Mehrere Funktionsaufrufe mit gleichem Parameterwert,*
- *z.B. 3mal F (0), 5mal F (1), 3mal F (2), 2mal F (3).*
- *Also: Derselbe Wert wird mehrfach berechnet.*
- *Also: Ineffizient implementiert.*

Aufgabe 4 Objektorientierte Begriffe erläutern

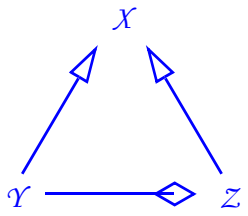
(Punkte: 25 |.....)

Gegeben sind diese Vereinbarungen:

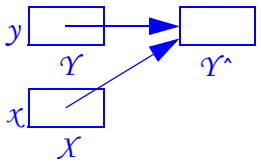
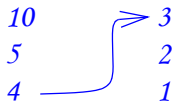
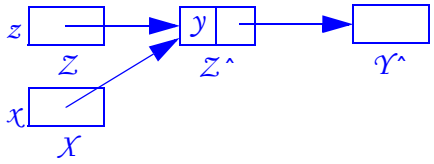
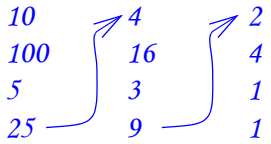
```
TYPE   X = POINTER TO ABSTRACT RECORD END;
       Y = POINTER TO RECORD (X) END;
       Z = POINTER TO RECORD (X) y : Y END;

PROCEDURE (this : X) Do (n : INTEGER), NEW, ABSTRACT;

PROCEDURE (this : X) Repeat (n : INTEGER), NEW;
BEGIN
   WHILE n > 0 DO this.Do (n); DEC (n) END
END Repeat;

PROCEDURE (this : Y) Do (n : INTEGER);
BEGIN
   Out.Int (n, 0); Out.Ln
END Do;

PROCEDURE (this : Z) Do (n : INTEGER);
BEGIN
   IF this.y = NIL THEN NEW (this.y) END;
   this.y.Do (n);
   Out.Int (n * n, 0); Out.Ln
END Do;
```

Klassendiagramm

Zeichnen Sie ein **Klassendiagramm** mit Vererbungs- und Bestandteilbeziehungen!



Die Vereinbarungen x : X; y : Y; z : Z; vorausgesetzt, geben Sie zu den Anweisungsfolgen an, was sie im Speicher und im Logfenster bewirken!

Anweisungsfolge	Zeiger und Objekte im Speicher	Ausgabe im Log
NEW (y); y.Do (10); x := y; x.Repeat (4);		
NEW (z); z.Do (10); x := z; x.Repeat (4);		

Programmelemente

Ordnen Sie den folgenden Begriffen passende, in dieser Aufgabe vorkommende **Programmelemente** zu! (Beispiel: Formalparameter: n : INTEGER.)

abstrakte Klasse: *X*

abstrakte Prozedur: *X.Do*

Empfängerparameter: *this : X, Y, Z*

Schnittstellenklasse: *X*

Implementationsklasse: *Y, Z*

Schablonenmethode: *X.Repeat*

Zeigervariable: *x, y, z, Z^.y*

dynamische Variable: *x^, y^, z^, Z^.y^*

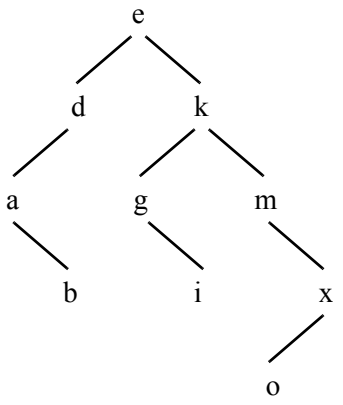
Objekterzeugung: *NEW (y), NEW (z), NEW (this.y)*

polymorphe Größe: *x, this : X*

polymorpher Aufruf: *x.Do (5), x.Repeat (4), this.y.Do (n)*

Aufgabe 5 Dynamische Datenstruktur Binärbaum bearbeiten

(Punkte: 16 |.....)



Geben Sie zu dem **Binärbaum** links die Reihenfolgen an, in der die Traversierungen die Elemente besuchen!

Präorder:...*e d a b k g i m x o*.....

Inorder:...*a b d e g i k m o x*.....

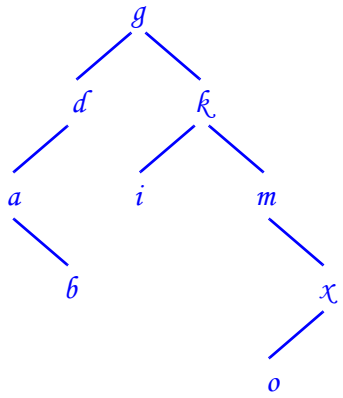
Postorder:...*b a d i g o x m k e*.....

Aus dem geordneten Binärbaum links wird das Element „e“ **entfernt** mit diesem Algorithmus:

```
Falls der Baum nicht leer ist:
    falls item kleiner als das Element der Wurzel ist
        entferne item aus dem linken Teilbaum,
    sonst falls item größer als das Element der Wurzel ist
        entferne item aus dem rechten Teilbaum,
    sonst ist item an der Wurzel, also falls der linke Teilbaum leer ist
        ersetze die Wurzel durch den rechten Teilbaum,
    sonst falls der rechte Teilbaum leer ist
        ersetze die Wurzel durch den linken Teilbaum,
    sonst sind beide Teilbäume nicht leer, also
        lasse den linken Teilbaum an seiner Stelle,
        entferne das kleinste Element aus dem rechten Teilbaum und
        setze es an die Stelle des zu entfernenden item an der Wurzel.
```

Markieren Sie den zutreffenden Zweig im Algorithmus und geben Sie den daraus entstehenden Baum an!

Reduzierter Baum

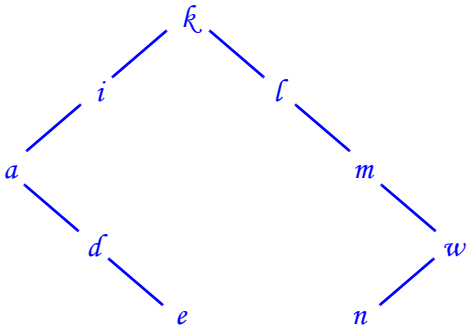


Aus einem leeren Baum entsteht durch **Einfügen** der Elemente

k l i m a w a n d e l

dieser **geordnete Binärbaum**:

Neuer Baum



Im Folgenden gelten die Definitionen:

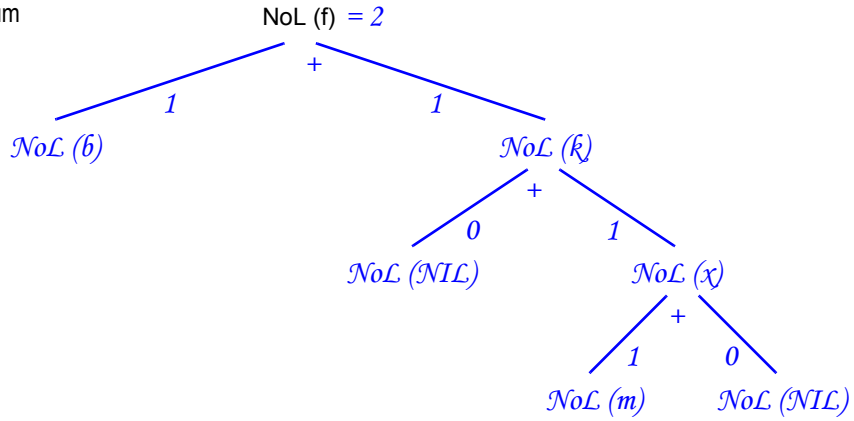
```
TYPE   Element   =  CHAR;
       Node      =  POINTER TO RECORD
                      item         : Element;
                      left, right  : Node
                    END;
```

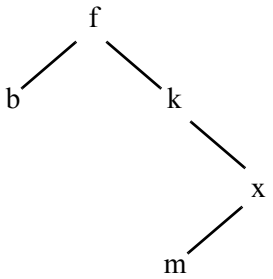
Die Funktion

```
PROCEDURE NoL (tree : Node) : INTEGER;
BEGIN
   IF tree = NIL THEN
      RETURN 0
   ELSIF (tree.left = NIL) & (tree.right = NIL) THEN
      RETURN 1
   ELSE
      RETURN NoL (tree.left) + NoL (tree.right)
   END
END NoL;
```

wird mit dem Baum rechts als aktuellem Parameter aufgerufen. Geben Sie den **Aufrufbaum** an, wobei Sie als Parameter tree.item statt tree hinschreiben, falls tree # NIL ist!

Aufrufbaum





Metainformation des Prüfers

Häufiger Fehler sind zu viele Aufrufe der Art NoL (NIL), wenn beide Kindknoten NIL sind.

Implementieren Sie die folgende Funktion, die die Anzahl der **inneren Knoten** (solche mit wenigstens einem Kind) des übergebenen Baums angibt, mit einem rekursiven Algorithmus!

```
PROCEDURE NumberOfInnerNodes (tree : Node) : INTEGER;
BEGIN
   IF (tree = NIL) OR ((tree.left = NIL) & (tree.right = NIL)) THEN
      RETURN 0
   ELSE
      RETURN
         NumberOfInnerNodes (tree.left) + 1 + NumberOfInnerNodes (tree.right)
   END
END NumberOfInnerNodes;
```

Drei Anweisungen

Aufgabe 6 — Menge als Binärbaum zeichnen, fabrizieren und transformieren

(Punkte: 34 |.....)

Bekannt sind die Mengenklasse ContainersSetsOfComparable.Set und die von BasisGenerals.Comparable erweiterte Klasse ContainersStrings.String:

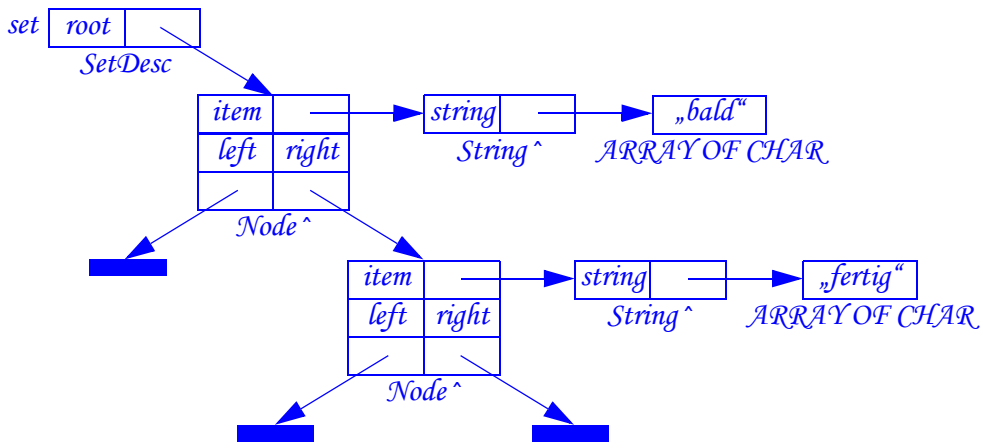
```
TYPE  Element* = BG.Comparable; (* = POINTER TO ... *)
      Node     = POINTER TO RECORD
                   item        : Element;
                   left, right : Node
                 END;
      Set*     = POINTER TO SetDesc;
      SetDesc* = RECORD
                   root        : Node
                 END;
      String*  = POINTER TO EXTENSIBLE RECORD (BG.Comparable)
                   string-     : POINTER TO ARRAY OF CHAR
                 END;
```

Zeichnen Sie das **Objektdiagramm** des Mengenobjekts set : SetDesc, das die in String-Objekte verpackten Werte "bald", "fertig" nacheinander durch Aufrufe von

```
PROCEDURE (VAR set : SetDesc) Put* (x : Element), NEW;
  ASSERT (x # NIL, BEC.precondPar1NotNil);
```

aufgenommen hat!

Objektdiagramm



Implementieren Sie die **Fabrikfunktion** NewSet, die zu einer Reihung von Elementen ein neues Mengenobjekt mit diesen Elementen fabriziert und dazu obiges Put benutzt!

```
PROCEDURE NewSet* (IN a : ARRAY OF Element) : Set;
  VAR
    result  : Set;
    i       : INTEGER;
BEGIN
  NEW (result);
  FOR i:= 0 TO LEN (a) - 1 DO
    IF a [i] # NIL THEN
      result.Put (a [i])
    END
  END;
  RETURN result
END NewSet;
```

Zwei Vereinbarungen

Fünf Anweisungen

Die an die Mengenklasse gebundene **Behältertransformation** AsSortedArray liefert

- bei leerer Empfängermenge NIL, sonst
- den Inhalt der Empfängermenge sortiert in einer dynamisch erzeugten Reihung.

Implementieren Sie die Baumfunktion SortedArray, die die Baumfunktion

```
PROCEDURE Count (tree : Node) : INTEGER;
```

für die Anzahl der Knoten im Baum benutzen kann.

Entwurfsideen

- *Reihung mit Länge = Anzahl der Knoten dynamisch erzeugen*
- *Baum mit Inorder traversieren, Wurzel in Reihung kopieren*

Drei Vereinbarungen

Zehn Anweisungen

```
PROCEDURE SortedArray (tree : Node) : POINTER TO ARRAY OF Element;
  VAR
    result   : POINTER TO ARRAY OF Element;
    i        : INTEGER;

  PROCEDURE Inorder (node : Node);
  BEGIN
    IF node # NIL THEN
      Inorder (node.left);
      result [i] := node.item;
      INC (i);
      Inorder (node.right)
    END
  END Inorder;

BEGIN
  IF tree # NIL THEN
    NEW (result, Count (tree));
    i := 0;
    Inorder (tree)
  END;
  RETURN result
END SortedArray;
```

```
PROCEDURE (IN set : SetDesc) AsSortedArray* () : POINTER TO ARRAY OF Element, NEW;
BEGIN
  RETURN SortedArray (set.root)
END AsSortedArray;
```

Aufgabe 7 Grundwissen über das Testen erläutern

(Punkte: 10 |.....)

Ein mki-B6-Student stellt im SOP-Journal vom Januar 2009 die folgenden unbegründeten, widersprüchlichen und falschen Behauptungen auf. Entgegnen Sie ihm mit gut begründeten, konsistenten und korrekten Aussagen!

Er: „*Testen dient der Verifikation eines entwickelten Systems. Komponententests verifizieren das nach außen sichtbare Verhalten einer Komponente.*“

Ich: *1. Sie benutzen den Begriff „Verifikation“ im ersten Satz falsch: Verifikation ist der Nachweis, dass ein Sachverhalt wahr ist. Ein System ist kein Sachverhalt, sondern ein Ding. Von der Wahrheit von Dingen zu reden ist begrifflich abwegig. Tatsächlich geht es Ihnen um Nachweise, dass ein entwickeltes System dem geplanten System bzw. das sichtbare Verhalten dem gewünschten Verhalten entspricht. Entspricht ein entwickeltes System dem geplanten System bzw. ein sichtbares Verhalten dem gewünschten Verhalten, so spricht man von der Korrektheit des Systems bzw. Verhaltens. Verifikation ist dann der Nachweis der Korrektheit des Systems bzw. Verhaltens.*

2. Sie haben das Wesen von Tests nicht begriffen: Der Zweck von Tests ist es, Fehler zu entdecken, d.h. Abweichungen zwischen dem gewünschten und dem tatsächlichen System bzw. Verhalten. “Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.” sagt schon Edsger W. Dijkstra in The Humble Programmer, ACM Turing Award Lecture 1972, Commun. ACM 15(10): 859-866(1972). Mit Testen kann man also zeigen, dass ein System oder Verhalten fehlerhaft ist, man kann damit aber nicht seine Korrektheit verifizieren (außer in trivialen Fällen).

Er: „*Die Komponententests mussten komplettiert werden, um die Korrektheit der Komponenten zu zeigen.*“

Ich: *1. Wie oben ausgeführt kann man mit Tests nicht die Korrektheit von Komponenten zeigen, sondern nur, dass sie Fehler enthalten.*

2. Der Grund dafür ist, dass die Anzahl möglicher Testfälle zu groß ist, um alle zu prüfen. Tests nichttrivialer Systeme sind nie vollständig, immer unvollständig, d.h. sie decken nur bestimmte, aber nicht alle Testfälle ab. Indem Sie von „komplettierten Tests“ sprechen, behaupten Sie, die Tests seien nach der Komplettierung vollständig, d.h. sie würden alle Testfälle abdecken. Diese Behauptung bezweifle ich, da sie meiner Erfahrung widerspricht.

3. Was Sie vielleicht meinen ist, dass die Tests um einige Testfälle ergänzt wurden, die einige weitere Fehler aufdeckten.

Er: „*Schlägt der Test einer zuvor erfolgreich getesteten Komponente nach Änderungen fehl, so ist die Wahrscheinlichkeit hoch, den Fehler in dieser Komponente zu finden.*“

Ich: *1. Sie benutzen die Eigenschaft „erfolgreich“ falsch: Tests sollen Fehler finden. Ein Test ist genau dann erfolgreich, wenn er einen Fehler findet, sonst ist er erfolglos. Sonst wäre es lächerlich leicht, erfolgreiche Tests zu schreiben - Tests, die keine Fehler finden!*

2. Bei Tests ist es wie in der Medizin: Ein HIV-Test ist positiv, wenn das Virus nachgewiesen werden konnte. Wenn Ihr HIV-Test positiv ist, haben Sie sicher Aids, sind aber nicht glücklich darüber. Ist der HIV-Test negativ, so konnte das Virus nicht nachgewiesen werden. Dann atmen Sie zwar auf, könnten aber trotzdem infiziert sein.

3. Was Sie meinen ist: „Wurde eine Komponente mit einem Test erfolglos getestet (d.h. ohne dass ein Fehler gefunden wurde), dann geändert und dann die geänderte Komponente mit demselben Test wieder getestet, diesmal aber erfolgreich, d.h. es wurde ein Fehler gefunden, so ist die Wahrscheinlichkeit hoch, dass die Fehlerursache in dieser Komponente liegt, und zwar an geänderten Stellen.“ Dieser umformulierten Aussage stimme ich zu.

Er: „*Komponententests verbessern die Qualität und Wartbarkeit des Codes.*“

Ich: *1. Reiner Unsinn! Sie können Qualität nicht in den Code hineintesten, sondern nur hineinkonstruieren. Tests lassen jeden Code unverändert. Ihre Behauptung trägt religiöse Züge.*

2. Was Sie meinen ist, dass sich durch die an Tests anschließende Behebung der Fehler, die durch die Tests gefunden wurden, die Qualität des Codes verbessert. Das mag insofern zutreffen, als sich die Korrektheit des Codes durch Beseitigen von Fehlern verbessert. Ob sich dadurch die Qualität generell oder speziell die Wartbarkeit des Codes verbessert, bezweifle ich. Gängige Praxis sind "quick and dirty bug fixes", die strukturellen Qualitätsmerkmalen wie Wartbarkeit eher abträglich sind. Strukturelle Qualität erzielt man durch systematisches Konstruieren mit guten Methoden, nicht durch Herumverbessern an schlecht zusammengebasteltem, fehlerhaftem Code.

Er: „*Der Einsatz eines Testframeworks verhindert inkorrekte Tests. So wird garantiert, dass die Tests selbst fehlerfrei sind.*“

Ich: *1. Was verstehen Sie unter einem „korrekten“ oder „fehlerfreien“ Test? Ein Test soll einen Fehler finden, das ist seine Aufgabe, die zu spezifizieren ist. Der Test selbst ist dann korrekt, wenn er seine Aufgabe erfüllt, also den spezifizierten Fehler findet, wenn dieser im Prüfling vorhanden ist.*

2. Ihre Behauptung ist damit, dass es beim Einsatz eines Testframeworks nur Tests gibt, die ihre spezifizierten Fehler finden. Dazu muss das Testframework die Spezifikationen der Tests kennen. Diese Spezifikationen müssen formalisiert sein, weil ein Werkzeug informale Spezifikationen nicht interpretieren kann. Formale Spezifikationen von Tests gibt es zwar, sie sind aber weit davon entfernt, gängige Industriepraxis zu sein.

3. Die formale Information, die einem Testwerkzeug in heutiger Praxis zur Verfügung steht, ist die Syntax der Schnittstellen von Prüflingen. Formale Spezifikationen der Semantik von Schnittstellen gibt es zwar auch, aber auch weit von gängiger Industriepraxis entfernt. Aus der syntaktischen Schnittstelleninformation kann ein Testwerkzeug syntaktisch korrekte Zugriffe auf diese Schnittstelle generieren - das ist kein Problem. Aber was haben wir davon? Syntaktisch korrekte Tests - mehr nicht. Von Semantik ist nicht die Rede. Offenbar verstehen Sie unter einem „korrekten“ Test nur einen „syntaktisch korrekten“ Test. Mir ist das zu wenig. Merke: Ein Werkzeug kann nicht mächtiger sein als die Methode, auf der es basiert.