Programmieren / Algorithmen & Datenstrukturen 2

Testgetriebene Programmierung

Prof. Dr. Skroch

Universitatea
BABEŞ-BOLYAI

Testgetriebene Programmierung

Inhalt.

- Templates
- Abgeleitete Klassen
- Testgetriebene Programmierung
- Container, Iteratoren und Algorithmen der StdLib
- Fortgeschrittenes Suchen
- Fortgeschrittenes Sortieren
- Grafische Benutzeroberflächen

Komplettes Beispiel

Fallstudie *FizzBuzz*, die Situation...

- Der Cornflakes-Hersteller *C.E. Rosstäuscher AG* plant zur Absatzförderung seines neuen Produkts "BioCereal Loops", einige der 1500 Gramm Großpackungen ("LoopOverdose") auf der Innenseite mit Gewinncodes zu versehen.
- Damit es einerseits möglichst viele Gewinner gibt, andererseits aber auch nicht jede Packung gewinnt, hat sich das Rosstäuscher-Marketing folgendes Schema zur Verteilung der Gewinncodes überlegt:
  - 1/3 der Packungen erhält einen sog. "Fizz"-Code, mit dem man übers Internet den BioCereal Loops Jingle aus dem TV-Werbespot gratis als polyphonen Klingelton herunterladen kann (nach Registrierung der E-Mail Adresse wird ein Download-Link zugemailt).
  - 1/5 der Packungen erhält einen sog. "Buzz"-Code, mit dem man übers Internet eine Packung LoopOverdose gratis bekommen kann (nach Registrierung von Name und Adresse kann ein Gutschein gedruckt werden, den der Einzelhandel annimmt).

FizzBuzz Reference Machine

Wir wollen eine Funktion `fbrm()` programmieren (fizzbuzz reference machine), die die Gewinn-Packungen fehlerfrei identifiziert.

- Die Funktion soll mit einer Produktionscharge von 1500 Gramm Packungen arbeiten, die fortlaufend, eine Packung nach der anderen, an der Verpackanlage ankommt.
- Die Funktion soll an die Verpackanlage folgende Werte zurückgeben:
  - Handelt es sich um eine fortlaufend dritte Packung, soll `Fizz` gemeldet werden, woraufhin eine Codedruck-Maschine einen Fizz-Gewinncode erzeugt und auf einen Zettel druckt, den die Packanlage dann in die Schachtel fallen lässt.
  - Handelt es sich um eine fortlaufend fünfte Packung, soll `Buzz` gemeldet werden, woraufhin eine Codedruck-Maschine einen Buzz-Gewinncode erzeugt und auf einen Zettel druckt, den die Packanlage dann in die Schachtel fallen lässt.
  - Für alle anderen Packungen soll einfach die laufende Nummer der Packung zurückgegeben werden, die Codedruck-Maschine macht dann nichts weiter.
- An diesem einfachen Beispiel soll die prinzipielle Vorgehensweise der testgetriebenen Programmierung demonstriert werden.

FizzBuzz Reference Machine

Wir wollen eine Funktion `fbrm()` programmieren (fizzbuzz reference machine), die die "Gewinn-Packungen" fehlerfrei identifiziert.

- Wir wollen `fbrm()` testgetrieben programmieren...
- Allgemeiner Ablauf bei der testgetriebenen Programmierung:
  - Zuerst wird ein Test für einen kleinen Teil der zu entwickelnden Funktion, hier `fbrm()`, geschrieben.
    - Danach wird ein Programm geschrieben, das diesen Test besteht.
  - Dann wird ein weiterer Test für einen weiteren kleinen Teil der zu entwickelnden Funktion geschrieben.
    - Danach wird das Programm erweitert und ggf. umgeschrieben, damit es beide Tests besteht.
  - Usw. bis die Funktion komplett ist.

Hilfsfunktionen für die Tests

Wir definieren uns zuerst einige allgemein einsetzbare Funktionen, die wir grundsätzlich für unsere Tests brauchen werden.

- Die wohl wichtigste Erkenntnis über Tests: es werden jeweils *erwartete* mit *tatsächlichen* Ergebnissen vergleichen.
- Wir definieren uns dafür einige allgemeine Hilfsfunktionen:

```
bool checkEquals( int exp, int act,
                  string description ) {
    // vergleiche expected und actual vom Typ int
}

bool checkEquals( string exp, string act,
                  string description ) {
    // vergleiche expected und actual vom Typ string
}

bool checkEquals( double exp, double act, double precision,
                  string description ) {
    // vergleiche expected und actual vom Typ double
}

bool checkTrue( bool condition,
                string description ) {
    // pruefe die Boolesche Bedingung auf true
}
```

Hilfsfunktionen für die Tests

Wir definieren uns zuerst einige allgemein einsetzbare Funktionen, die wir grundsätzlich für unsere Tests brauchen werden.

- Eine Funktion, die ein Testergebnis direkt ausgibt:

```
void printTestResult( bool successful,
                      string testType,
                      string description,
                      string expected="",
                      string actual="" ) {
    if( !successful ) cout << "FAILED test ";
    else cout << "  succeeded test ";
    cout << testType << '(' << description << ')';
    cout << ", expected: " << expected;
    cout << ", actual: " << actual << endl;
}
```

Hilfsfunktion für `int` Werte

Wir definieren uns zuerst einige allgemein einsetzbare Funktionen, die wir grundsätzlich für unsere Tests brauchen werden.

- Wir wollen die Ergebnisse direkt aus den Prüffunktionen ausgeben können, hier beim Vergleich von Werten vom `int` Typ.

```
bool checkEquals( int exp, int act,
                  string description,
                  bool silent=false ) {
  bool result{ exp == act };
  if( !silent )
     printTestResult( result,
                      "checkEquals",
                      description,
                      std::to_string( exp ),
                      std::to_string( act ) );
  return result;
}
```

```
// std::to_string
// kann aehnlich auch
// selbst definiert
// werden:
template<class T>
string toString( T t ) {
    stringstream stst{};
    stst << t;
    return stst.str();
}
```

Hilfsfunktion für `string` Werte

Wir definieren uns zuerst einige allgemein einsetzbare Funktionen, die wir grundsätzlich für unsere Tests brauchen werden.

- Wir wollen die Ergebnisse direkt aus den Prüffunktionen ausgeben können, hier beim Vergleich von Werten vom `string` Typ.

```
bool checkEquals( string exp, string act,
                  string description,
                  bool silent=false    ) {
  bool result{ exp == act };
  if( !silent )
     printTestResult( result,
                      "checkEquals",
                      description,
                      exp,
                      act );
  return result;
}
```

Hilfsfunktion für `double` Werte

Wir definieren uns zuerst einige allgemein einsetzbare Funktionen, die wir grundsätzlich für unsere Tests brauchen werden.

- Wir wollen die Ergebnisse direkt aus den Prüffunktionen ausgeben können, hier beim Vergleich von Werten vom `double` Typ.

```
bool checkEquals( double exp, double act, double precision,
                  string description
                  bool silent=false ) {
  double difference { ( exp>=act ) ? ( exp-act ) : ( act-exp ) };
  bool result{ difference <= precision };
  if( !silent )
     printTestResult( result,
                      "checkEquals",
                      description,
                      std::to_string( exp ),
                      std::to_string( act ) );
  return result;
}
```

Hilfsfunktion für `bool` Werte

Wir definieren uns zuerst einige allgemein einsetzbare Funktionen, die wir grundsätzlich für unsere Tests brauchen werden.

- Wir wollen die Ergebnisse direkt aus den Prüffunktionen ausgeben können, hier bei Bedingungen (`bool` Typ).

```
bool checkTrue( bool condition,
                string description,
                bool silent=false ) {
  if( !silent )
     printTestResult( condition,
                      "checkTrue",
                      description );
  return condition;
}
```

LEERE SEITE

FizzBuzz testgetrieben programmiert

Der erste Testfall.

- Mit unseren Testfunktionen können wir nun einen elementaren Test nach dem anderen schreiben.
- Für die Funktionalität von `fbrm()` fangen wir mit dem nahe liegenden, ersten Testfall an: die erste Packung, für die `"1"` zurückgegeben wird.
  - *Zuerst* schreibt man den entsprechenden Test für `fbrm()`

```
checkEquals( "1", fbrm( 1 ), "1.Packung" );
```

  - *Danach* schreibt man eine Version von `fbrm()`, die den Test bestehen soll (diese wird oft von anderen Programmierern geschrieben)

```
string fbrm( int k ) {
    if( k == 1 ) return string{ "1" };
}
```

- Ergebnis: Compilerwarnung

```
main.cpp: In function 'std::string fbrm(int)':
main.cpp:56:1: warning: control reaches end of non-void function
```

  - D.h. dieses `fbrm()` wird noch nicht einmal einwandfrei übersetzt...
    - Bem.: Compileroption `-Wall` für "alle Warnungen" einschalten (immer) !

FizzBuzz testgetrieben programmiert

Der zweite und dritte Testfall.

- Unser bester Freund beim Programmieren, der Compiler, erinnert uns daran, dass die Funktion für `k!=1` keinen definierten Rückgabewert hat.
- Wir folgen der testgetriebenen Programmierung.
  - *Zuerst* zwei entsprechende Tests für `fbrm()`

```
checkEquals( "invalid", fbrm( 0 ), "0.Packung" );
checkEquals( "invalid", fbrm( -1 ),
             "-1.Packung, e.c. negative Zahlen" );
```

  - *Danach* eine Version von `fbrm()`, die alle drei Tests bestehen soll, wobei wir dank der Compilerwarnung nun auch berücksichtigen, dass unsere Funktion *immer* einen Rückgabewert haben muss

```
string fbrm( int k ) {
   if( k == 1 ) return string{ "1" };
   return string{ "invalid" };
}
```

- Ergebnis:

e.c. "equivalence class"

```
succeeded test checkEquals(1.Packung), expected: 1, actual: 1
succeeded test checkEquals(0.Packung), expected: invalid, actual: invalid
succeeded test checkEquals(-1.Packung, e.c. negative Zahlen), expected: invalid, actual: invalid
```

FizzBuzz testgetrieben programmiert

Der vierte Testfall.

- Als nächstes die zweite Packung, für die `"2"` zurückgegeben wird.
  - *Zuerst* der entsprechende Test für `fbrm()`

```
checkEquals( "2", fbrm( 2 ), "2.Packung" );
```

  - *Danach* eine Version von `fbrm()`, die die vier bisherigen Tests bestehen soll

```
string fbrm( int k ) {
    if( k == 1 ) return string{ "1" };
    if( k == 2 ) return string{ "2" };
    return string{ "invalid" };
}
```

- Ergebnis:

```
succeeded test checkEquals(1.Packung), expected: 1, actual: 1
succeeded test checkEquals(0.Packung), expected: invalid, actual: invalid
succeeded test checkEquals(-1.Packung, e.c. negative Zahlen), expected: invalid, actual: invalid
succeeded test checkEquals(2.Packung), expected: 2, actual: 2
```

FizzBuzz testgetrieben programmiert

Der fünfte Testfall.

- Als nächstes die dritte Packung, für die `"Fizz"` zurückgegeben wird.
  - *Zuerst* der entsprechende Test für `fbrm()`

```
checkEquals( "Fizz", fbrm( 3 ), "Fizz in 3.Packung" );
```

  - *Danach* eine Version von `fbrm()`, die die fünf bisherigen Tests bestehen soll

```
string fbrm( int k ) {
    if( k == 1 ) return string{ "1" };
    if( k == 2 ) return string{ "2" };
    if( k == 3 ) return string{ "Fizz" };
    return string{ "invalid" };
}
```

- Ergebnis:

```
succeeded test checkEquals(1.Packung), expected: 1, actual: 1
succeeded test checkEquals(0.Packung), expected: invalid, actual: invalid
succeeded test checkEquals(-1.Packung, e.c. negative Zahlen), expected: invalid, actual: invalid
succeeded test checkEquals(2.Packung), expected: 2, actual: 2
succeeded test checkEquals(Fizz in 3.Packung), expected: Fizz, actual: Fizz
```

FizzBuzz testgetrieben programmiert

Der erweiterte fünfte Testfall.

- Es muss für *jede* dritte Packung `"Fizz"` zurückgegeben werden.
  - *Zuerst* der erweiterte Test für `fbrm()`

```
const int maxProdSize = 60; // hier zur Demonstration
```

```
for( int i {3}; i <= maxProdSize; i += 3 )
   checkEquals( "Fizz", fbrm( i ),
                "Fizz in " + to_string( i ) + ".Packung");
```

  - *Danach* eine Version von `fbrm()`, die alle bisherigen Tests bestehen soll

```
string fbrm( int k ) {
   if( k%3 == 0 ) return string{ "Fizz" };
   if( k%3 != 0 ) return to_string( k );
   return string{ "invalid" };
}
```

- Ergebnis:

```
  succeeded test checkEquals(1.Packung), expected: 1, actual: 1
FAILED test checkEquals(0.Packung), expected: invalid, actual: Fizz
FAILED test checkEquals(-1.Packung, e.c. negative Zahlen), expected: invalid, actual: -1
  succeeded test checkEquals(2.Packung), expected: 2, actual: 2
  succeeded test checkEquals(Fizz in 3.Packung), expected: Fizz, actual: Fizz
  succeeded test checkEquals(Fizz in 6.Packung), expected: Fizz, actual: Fizz
  succeeded test checkEquals(Fizz in 9.Packung), expected: Fizz, actual: Fizz
  succeeded test checkEquals(Fizz in 12.Packung), expected: Fizz, actual: Fizz
  succeeded test checkEquals(Fizz in 15.Packung), expected: Fizz, actual: Fizz
```

  - D.h. wir müssen jetzt die `fbrm()` Ablauflogik etwas umstellen…

FizzBuzz testgetrieben programmiert

Neuentwurf von `fbrm()` nach dem verallgemeinerten fünften Testfall.

- Es muss für *jede* dritte Packung `"Fizz"` zurückgegeben werden.
  - Nochmal der erweiterte Test für `fbrm()`

```
for( int i {3}; i <= maxProdSize; i += 3 )
   checkEquals( "Fizz", fbrm( i ),
                "Fizz in " + to_string( i ) + ".Packung");
```

  - Und die geänderte Version von `fbrm()`, die alle bisherigen Tests bestehen soll

```
string fbrm( int k ) {
   if( k < 1 ) return string{ "invalid" };
   if( k%3 == 0 ) return string{ "Fizz" };
   return to_string( k );
}
```

- Ergebnis:

```
succeeded test checkEquals(1.Packung), expected: 1, actual: 1
succeeded test checkEquals(0.Packung), expected: invalid, actual: invalid
succeeded test checkEquals(-1.Packung, e.c. negative Zahlen), expected: invalid, actual: invalid
succeeded test checkEquals(2.Packung), expected: 2, actual: 2
succeeded test checkEquals(Fizz in 3.Packung), expected: Fizz, actual: Fizz
succeeded test checkEquals(Fizz in 6.Packung), expected: Fizz, actual: Fizz
succeeded test checkEquals(Fizz in 9.Packung), expected: Fizz, actual: Fizz
succeeded test checkEquals(Fizz in 12.Packung), expected: Fizz, actual: Fizz
succeeded test checkEquals(Fizz in 15.Packung), expected: Fizz, actual: Fizz
```

FizzBuzz testgetrieben programmiert

Der sechste Testfall.

- Jetzt kommt die vierte Packung, für die `"4"` zurückgegeben wird.
  - *Zuerst* der entsprechende Tests für `fbrm()`

    ```
    checkEquals( "4", fbrm( 4 ), "4.Packung" );
    ```

  - Unsere unveränderte `fbrm()` sollte auch diesen Test bereits bestehen

    ```
    string fbrm( int k ) {
        if( k < 1 ) return string{ "invalid" };
        if( k%3 == 0 ) return string{ "Fizz" };
        return to_string( k );
    }
    ```

- Ergebnis:

```
succeeded test checkEquals(1.Packung), expected: 1, actual: 1
succeeded test checkEquals(0.Packung), expected: invalid, actual: invalid
succeeded test checkEquals(-1.Packung, e.c. negative Zahlen), expected: invalid, actual: invalid
succeeded test checkEquals(2.Packung), expected: 2, actual: 2
succeeded test checkEquals(Fizz in 3.Packung), expected: Fizz, actual: Fizz
succeeded test checkEquals(Fizz in 6.Packung), expected: Fizz, actual: Fizz
...
succeeded test checkEquals(Fizz in 57.Packung), expected: Fizz, actual: Fizz
succeeded test checkEquals(Fizz in 60.Packung), expected: Fizz, actual: Fizz
succeeded test checkEquals(4.Packung), expected: 4, actual: 4
```

FizzBuzz testgetrieben programmiert

Der siebte Testfall.

- Als nächstes die fünfte Packung, für die `"Buzz"` zurückgegeben wird.
  - *Zuerst* der entsprechende Test für `fbrm()`

```
checkEquals( "Buzz", fbrm( 5 ), "Buzz in 5.Packung" );
```

  - *Danach* eine Version von `fbrm()`, die alle bisherigen Tests bestehen soll

```
string fbrm( int k ) {
   if( k < 1 ) return string{ "invalid" };
   if( k%3 == 0 ) return string{ "Fizz" };
   if( k == 5 ) return string{ "Buzz" };
   return to_string( k );
}
```

- Ergebnis:

```
succeeded test checkEquals(Fizz in 54.Packung), expected: Fizz, actual: Fizz
succeeded test checkEquals(Fizz in 57.Packung), expected: Fizz, actual: Fizz
succeeded test checkEquals(Fizz in 60.Packung), expected: Fizz, actual: Fizz
succeeded test checkEquals(4.Packung), expected: 4, actual: 4
succeeded test checkEquals(Buzz in 5.Packung), expected: Buzz, actual: Buzz
```

FizzBuzz testgetrieben programmiert

Der erweiterte siebte Testfall, und ein Problem taucht auf.

- Es muss für *jede* fünfte Packung `"Buzz"` zurückgegeben werden.
  - *Zuerst* der erweiterte Test für `fbrm()`

```
for( int i {5}; i <= maxProdSize; i += 5 )
    checkEquals( "Buzz", fbrm( i ),
                 "Buzz in " + to_string( i ) + ".Packung");
```

  - *Danach* eine Version von `fbrm()`, die alle bisherigen Tests bestehen soll

```
string fbrm( int k ) {
    if( k < 1 ) return string{ "invalid" };
    if( k%3 == 0 ) return string{ "Fizz" };
    if( k%5 == 0 ) return string{ "Buzz" };
    return to_string( k );
}
```

- Ergebnis:

15, 30, 45, 60, ... sind durch drei *und* durch fünf teilbar

```
  succeeded test checkEquals(Fizz in 60.Packung), expected: Fizz, actual: Fizz
  succeeded test checkEquals(4.Packung), expected: 4, actual: 4
  succeeded test checkEquals(Buzz in 5.Packung), expected: Buzz, actual: Buzz
  succeeded test checkEquals(Buzz in 10.Packung), expected: Buzz, actual: Buzz
FAILED test checkEquals(Buzz in 15.Packung), expected: Buzz, actual: Fizz
  succeeded test checkEquals(Buzz in 20.Packung), expected: Buzz, actual: Buzz
  succeeded test checkEquals(Buzz in 25.Packung), expected: Buzz, actual: Buzz
FAILED test checkEquals(Buzz in 30.Packung), expected: Buzz, actual: Fizz
  succeeded test checkEquals(Buzz in 35.Packung), expected: Buzz, actual: Buzz
  succeeded test checkEquals(Buzz in 40.Packung), expected: Buzz, actual: Buzz
FAILED test checkEquals(Buzz in 45.Packung), expected: Buzz, actual: Fizz
  succeeded test checkEquals(Buzz in 50.Packung), expected: Buzz, actual: Buzz
  succeeded test checkEquals(Buzz in 55.Packung), expected: Buzz, actual: Buzz
FAILED test checkEquals(Buzz in 60.Packung), expected: Buzz, actual: Fizz
```

FizzBuzz testgetrieben programmiert

Der achte Testfall für das Problem, das im verallgemeinerten siebten Testfall aufgetaucht ist.

- Die entsprechende Rückfrage löst beim Marketing eine mehrtägige Diskussion aus, man kommt schließlich zu der Entscheidung, dass in diesen Packungen *beide* Gewinncodes sein sollen.
- Die Funktion soll dann an die Verpackanlage den Wert `"FizzBuzz"` zurückgeben.
  - *Zuerst* der entsprechende Test für `fbrm()`

```
for( int i {15}; i <= maxProdSize; i += 15 )
   checkEquals( "FizzBuzz", fbrm( i ),
                "FizzBuzz in " + to_string( i ) + ".Packung");
```

  - *Danach* eine Version von `fbrm()`, die alle bisherigen Tests bestehen soll

```
string fbrm( int k ) {
   if( k < 1 ) return string{ "invalid" };
   if( k%15 == 0 ) return string{ "FizzBuzz" };
   if( k%3 == 0 ) return string{ "Fizz" };
   if( k%5 == 0 ) return string{ "Buzz" };
   return to_string( k );
}
```

- Ergebnis: siehe nächste Seite.

FizzBuzz testgetrieben programmiert

Überraschung: es tauchen Fehler auf, die nicht am Programm liegen, weil nun die Erwartung bestimmter Testfälle falsch ist (sog. *Testorakel-Problem*).

```
  succeeded test checkEquals(1.Packung), expected: 1, actual: 1
  succeeded test checkEquals(0.Packung), expected: invalid, actual: invalid
  succeeded test checkEquals(-1.Packung, e.c. negative Zahlen), expected: invalid, actual: invalid
  succeeded test checkEquals(2.Packung), expected: 2, actual: 2
  succeeded test checkEquals(Fizz in 3.Packung), expected: Fizz, actual: Fizz
  succeeded test checkEquals(Fizz in 6.Packung), expected: Fizz, actual: Fizz
  succeeded test checkEquals(Fizz in 9.Packung), expected: Fizz, actual: Fizz
  succeeded test checkEquals(Fizz in 12.Packung), expected: Fizz, actual: Fizz
FAILED test checkEquals(Fizz in 15.Packung), expected: Fizz, actual: FizzBuzz
  succeeded test checkEquals(Fizz in 18.Packung), expected: Fizz, actual: Fizz
  succeeded test checkEquals(Fizz in 21.Packung), expected: Fizz, actual: Fizz
  succeeded test checkEquals(Fizz in 24.Packung), expected: Fizz, actual: Fizz
  succeeded test checkEquals(Fizz in 27.Packung), expected: Fizz, actual: Fizz
FAILED test checkEquals(Fizz in 30.Packung), expected: Fizz, actual: FizzBuzz
  succeeded test checkEquals(Fizz in 33.Packung), expected: Fizz, actual: Fizz
  succeeded test checkEquals(Fizz in 36.Packung), expected: Fizz, actual: Fizz
  succeeded test checkEquals(Fizz in 39.Packung), expected: Fizz, actual: Fizz
  succeeded test checkEquals(Fizz in 42.Packung), expected: Fizz, actual: Fizz
FAILED test checkEquals(Fizz in 45.Packung), expected: Fizz, actual: FizzBuzz
  succeeded test checkEquals(Fizz in 48.Packung), expected: Fizz, actual: Fizz
  succeeded test checkEquals(Fizz in 51.Packung), expected: Fizz, actual: Fizz
  succeeded test checkEquals(Fizz in 54.Packung), expected: Fizz, actual: Fizz
  succeeded test checkEquals(Fizz in 57.Packung), expected: Fizz, actual: Fizz
FAILED test checkEquals(Fizz in 60.Packung), expected: Fizz, actual: FizzBuzz
  succeeded test checkEquals(4.Packung), expected: 4, actual: 4
  succeeded test checkEquals(Buzz in 5.Packung), expected: Buzz, actual: Buzz
  succeeded test checkEquals(Buzz in 10.Packung), expected: Buzz, actual: Buzz
FAILED test checkEquals(Buzz in 15.Packung), expected: Buzz, actual: FizzBuzz
  succeeded test checkEquals(Buzz in 20.Packung), expected: Buzz, actual: Buzz
  succeeded test checkEquals(Buzz in 25.Packung), expected: Buzz, actual: Buzz
FAILED test checkEquals(Buzz in 30.Packung), expected: Buzz, actual: FizzBuzz
  succeeded test checkEquals(Buzz in 35.Packung), expected: Buzz, actual: Buzz
  succeeded test checkEquals(Buzz in 40.Packung), expected: Buzz, actual: Buzz
FAILED test checkEquals(Buzz in 45.Packung), expected: Buzz, actual: FizzBuzz
  succeeded test checkEquals(Buzz in 50.Packung), expected: Buzz, actual: Buzz
  succeeded test checkEquals(Buzz in 55.Packung), expected: Buzz, actual: Buzz
FAILED test checkEquals(Buzz in 60.Packung), expected: Buzz, actual: FizzBuzz
  succeeded test checkEquals(FizzBuzz in 15.Packung), expected: FizzBuzz, actual: FizzBuzz
  succeeded test checkEquals(FizzBuzz in 30.Packung), expected: FizzBuzz, actual: FizzBuzz
  succeeded test checkEquals(FizzBuzz in 45.Packung), expected: FizzBuzz, actual: FizzBuzz
  succeeded test checkEquals(FizzBuzz in 60.Packung), expected: FizzBuzz, actual: FizzBuzz
```

Die manuelle Überprüfung der Fehlermeldungen zeigt: die tatsächlichen Ergebnisse sind jeweils korrekt

Die erwarteten Ergebnisse, und damit auch das Testorakel "FAILED", müssen also falsch sein.

Folge: die *Testfälle* müssen korrigiert werden.

FizzBuzz testgetrieben programmiert

Berichtigung der Testorakel für den erweiterten fünften und den erweiterten siebten Testfall.

- *Zuerst* Korrektur der beiden Testfälle.
  - ```
    for( int i {3}; i <= maxProdSize; i += 3 ) {
        if( i%5 == 0 ) continue;
        checkEquals( "Fizz", fbrm( i ),
                     "Fizz in " + to_string( i ) + ".Packung");
    }
    ```
  - ```
    for( int i {5}; i <= maxProdSize; i += 5 ) {
        if( i%3 == 0 ) continue;
        checkEquals( "Buzz", fbrm( i ),
                     "Buzz in " + to_string( i ) + ".Packung");
    }
    ```
- *Danach* zur `fbrm()` (die unverändert bleibt).
- Ergebnis: siehe nächste Seite.

FizzBuzz testgetrieben programmiert

Die Testfälle erzeugen jetzt keine falschen FAILED-Ergebnisse mehr.

- Die Test Suite erzeugt die folgenden Ausgaben:

```
succeeded test checkEquals(1.Packung), expected: 1, actual: 1
succeeded test checkEquals(0.Packung), expected: invalid, actual: invalid
succeeded test checkEquals(-1.Packung, e.c. negative Zahlen), expected: invalid, actual: invalid
succeeded test checkEquals(2.Packung), expected: 2, actual: 2
succeeded test checkEquals(Fizz in 3.Packung), expected: Fizz, actual: Fizz
succeeded test checkEquals(Fizz in 6.Packung), expected: Fizz, actual: Fizz
succeeded test checkEquals(Fizz in 9.Packung), expected: Fizz, actual: Fizz
succeeded test checkEquals(Fizz in 12.Packung), expected: Fizz, actual: Fizz
succeeded test checkEquals(Fizz in 18.Packung), expected: Fizz, actual: Fizz
succeeded test checkEquals(Fizz in 21.Packung), expected: Fizz, actual: Fizz
succeeded test checkEquals(Fizz in 24.Packung), expected: Fizz, actual: Fizz
succeeded test checkEquals(Fizz in 27.Packung), expected: Fizz, actual: Fizz
succeeded test checkEquals(Fizz in 33.Packung), expected: Fizz, actual: Fizz
succeeded test checkEquals(Fizz in 36.Packung), expected: Fizz, actual: Fizz
succeeded test checkEquals(Fizz in 39.Packung), expected: Fizz, actual: Fizz
succeeded test checkEquals(Fizz in 42.Packung), expected: Fizz, actual: Fizz
succeeded test checkEquals(Fizz in 48.Packung), expected: Fizz, actual: Fizz
succeeded test checkEquals(Fizz in 51.Packung), expected: Fizz, actual: Fizz
succeeded test checkEquals(Fizz in 54.Packung), expected: Fizz, actual: Fizz
succeeded test checkEquals(Fizz in 57.Packung), expected: Fizz, actual: Fizz
succeeded test checkEquals(4.Packung), expected: 4, actual: 4
succeeded test checkEquals(Buzz in 5.Packung), expected: Buzz, actual: Buzz
succeeded test checkEquals(Buzz in 10.Packung), expected: Buzz, actual: Buzz
succeeded test checkEquals(Buzz in 20.Packung), expected: Buzz, actual: Buzz
succeeded test checkEquals(Buzz in 25.Packung), expected: Buzz, actual: Buzz
succeeded test checkEquals(Buzz in 35.Packung), expected: Buzz, actual: Buzz
succeeded test checkEquals(Buzz in 40.Packung), expected: Buzz, actual: Buzz
succeeded test checkEquals(Buzz in 50.Packung), expected: Buzz, actual: Buzz
succeeded test checkEquals(Buzz in 55.Packung), expected: Buzz, actual: Buzz
succeeded test checkEquals(FizzBuzz in 15.Packung), expected: FizzBuzz, actual: FizzBuzz
succeeded test checkEquals(FizzBuzz in 30.Packung), expected: FizzBuzz, actual: FizzBuzz
succeeded test checkEquals(FizzBuzz in 45.Packung), expected: FizzBuzz, actual: FizzBuzz
succeeded test checkEquals(FizzBuzz in 60.Packung), expected: FizzBuzz, actual: FizzBuzz
```

- Haben wir damit gezeigt, dass `fbrm()` wirklich immer fehlerfrei arbeitet?

FizzBuzz

Vollständiges Testen von `fbrm()` im Betriebsbereich.

```
void fbrmFullTest( int testRangeStart, int testRangeEnd ) {
  for( int i {testRangeStart}; i <= testRangeEnd; ++i ) {
     if( fbrm(i) == toString(i) ) {
         if( i%3 == 0 ) error( "fbrm() 3-fault" );
         if( i%5 == 0 ) error( "fbrm() 5-fault" );
         if( i%15 == 0 ) error( "fbrm() 15-fault" );
     }
     if( fbrm(i) == "Fizz" ) {
         if( i%3 != 0 ) error( "fbrm() Fizz-fault" );
     }
     if( fbrm(i) == "Buzz" ) {
         if( i%5 != 0 ) error( "fbrm() Buzz-fault" );
     }
     if( fbrm(i) == "FizzBuzz" ) {
         if( i%15 != 0 ) error( "fbrm() FizzBuzz-fault" );
     }
  }
  cout << "fbrmFullTest() ok from " << testRangeStart
       << " to " << testRangeEnd << endl;
}
// Bem.: der Header <limits> bietet
// numeric_limits<int>::max() und numeric_limits<int>::min()
```

Einige Beispielfragen

Testgetriebene Programmierung.

- Machen Sie sich die Vorgehensweise der testgetriebenen Programmierung anhand eines *eigenen* kleinen Beispiels klar, welches Sie selbst testgetrieben programmieren.
- Welche Vorteile sehen Sie, wenn die Tests nicht von den Personen geschrieben werden, die das Programm schreiben? Welche Nachteile sehen Sie?
- Wie erklären Sie sich das Problem, das im erweiterten siebten Testfall aufgetaucht ist?
- Was verstehen Sie unter einem Testorakel? Woher erhalten Sie die Testorakel für eine bestimmte Programmieraufgabe?

Nächste Einheit:
Container, Iteratoren und Algorithmen der StdLib