Robustes Programmieren

Strategien zum Umgang mit Fehlern bei der Programmentwicklung

- Identifizierung möglicher Fehlerquellen mit Checklisten
  - Berücksichtigung von Checklisten trägt dazu bei, bestimmte Fehlerquellen zu identifizieren und eventuelle Fehler zu beseitigen.
- Absicherung von Quellcode gegen fehlerhafte Verwendung
  - Besondere Beachtung von Schnittstellen (Code, der fremden Code verwendet oder der von fremdem Code verwendet wird).
  - Strategien: Parameterprüfung bei Methoden, Fehlercodes, Ausnahmebehandlung zur Laufzeit.
- Testen von Programmen
  - Verschiedene Software-Testmethoden.
  - Umfassender Test aller erdenklichen Eingaben i. d. R. nicht möglich.
- Formale Verifikation von Programm(teil)en
  - Formaler Nachweis, dass Programm(stück) das Gewünschte leistet.
- Tolerenz gegen verbleibende Fehler
  - „Graceful Degradation“, sichere Abschaltung bei Ungereimtheiten

Programmentwicklung

- Gemeinsame Programmentwicklung
  - Es hat sich gezeigt, dass bei einer Programmentwicklung wesentlich weniger Fehler entstehen, wenn zwei Personen zusammen ein Programm entwickeln o. einer dem anderen d. entwickelten Code erklärt.
- Gemeinsames „Walkthrough“:
  - Begutachtung eines Programms.
  - Autor stellt den Gutachtern das Programm vor.
  - Meist werden typische Anwendungsfälle (Benutzungssituationen, Szenarien, Probeläufe) im Kreis gleichgestellter Mitarbeiter ablauforientiert durchgespielt.
  - Ziel: Fehler finden, z. B. anhand von Fragen aus Checklisten (s.u.) zur Fehleridentifizierung und -vermeidung.
  - Nacharbeit durch den Autor nach Ende der Sitzung.

Fehlervermeidung bei Methodenargumenten

- Übersetzer
  - Garantiert, dass Typ des aktuellen Parameterwertes beim Aufruf gleich dem Typ des zugehörigen formalen Parameters in der Deklaration ist (oder zumindest automatisch umgewandelt werden kann).
  - Beispiel: verhindert, dass Methode double sqrt(double d), mit String-Parameter aufgerufen wird.
- Dokumentation
  - Autor/-in einer Methode sollte in Kommentaren stets dokumentieren, was diese Methode macht, welche Argumente sie erwartet und was für einen Rückgabewert sie zurückliefert.
  - Bei Benutzereingaben sollte Benutzer/-in (z. B. durch passende Ausgabe) darauf hingewiesen werden, welche Daten einzugeben sind.
- Überprüfung der Parameterwerte am Methodenanfang
  - Naiv: Fehlercodes
  - Besser: Ausnahmen

Grenzen der Rückmeldung von Fehlern über Fehlercodes

- Programmcode und Fehlerbehandlungscode
  - Aufrufendes Programmstück muss Rückgabewert immer direkt testen.
  - Dadurch: Mischung von „normalem“ Programmcode und Fehlerbehandlungscode.
- Fehlermeldung
  - Will Methode Fehler mit Text dokumentieren oder erläutern, muss sie diesen Text selbst ausgeben oder in eine Protokolldatei schreiben.
  - Methode hat i. d. R. keine Möglichkeit, Text an den Aufrufer weiterzureichen, damit dieser entscheidet, was damit zu tun ist.
- Wertebereich
  - Oft sind alle Werte aus dem Wertebereich des Rückgabetyps gültige Ergebniswerte. Dann ist kein Wert für den oder die Fehlercodes übrig.

Ausnahmen (engl. exceptions)

- Java bietet einen leistungsstarken Mechanismus zur Behandlung von Fehlersituationen zur Laufzeit basierend auf sog. Ausnahmen.
- Eine Ausnahme (engl. exception) ist ein Ereignis, durch das die normale Ausführungsreihenfolge unterbrochen wird.
- Programmiertechnisch sind Ausnahmen Objekte bereits in Java enthaltener oder selbstdefinierter Ausnahmeklassen, wobei die Klasse die Ausnahmesituation näher charakterisiert, z. B.
  - IllegalArgumentException: illegaler Parameter
  - FileNotFoundException: benötigte Datei nicht gefunden
- Objekte solcher Ausnahmeklassen werden an der Fehlerstelle im Programm erzeugt und können dann Informationen z. B. über den Fehlerort oder eine Fehlermeldung an die Stelle im Programm „transportieren“, wo die Bearbeitung der Fehlersituation erfolgen soll (z. B. in die aufrufende Methode).
- Man kann nicht jede einzelne Ausnahme vorhersagen.
- Man kann jedoch die Arten von Ausnahmen vorhersagen. Daher werden Ausnahmen klassifiziert: Jeder Ausnahme wird ein Ausnahmetyp (die entsprechende Ausnahmeklasse) zugeordnet.
- Die Behandlung orientiert sich dann am Ausnahmetyp.
- Dies begründet die Trennung von Erkennung und Behandlung.

Aufgaben/Ablauf:

- Erkennen einer Ausnahmesituation
- Erzeugen einer Ausnahmebeschreibung (eines bestimmten Typs)
- Anstoßen der Ausnahmebehandlung („Werfen“)
- Behandeln der Ausnahme gemäß Typ

Beispiel

```
static double division(int zaehler, int nenner) {
        if (nenner == 0) {
                IllegalArgumentException e =
                        new IllegalArgumentException("Fehler: versuchte Division d. 0");
                throw e;
        }
       return zaehler / nenner;
}
```

- Den Konstruktoren der vordefinierten Fehlerklassen kann als Argument ein String mit einer Fehlerbeschreibung übergeben werden.
- Später: Abfrage dieses Strings möglich mit getMessage()-Methode des Ausnahmeobjekts.

```
class ExceptionDemo {
        static double division(int zaehler, int nenner) {
              ...
        }
        public static void main(String[] args) {
                try {
                        double bruch;
                        bruch = division(12,5);
                        System.out.println(bruch);
                        bruch = division(12,0); // loest Ausnahme aus
                        System.out.println(bruch);
                } catch (IllegalArgumentException e) {
                        System.err.println(e.getMessage());
                }
        }
}
```

Hierarchie in der Standardbibliothek definierter Ausnahmen

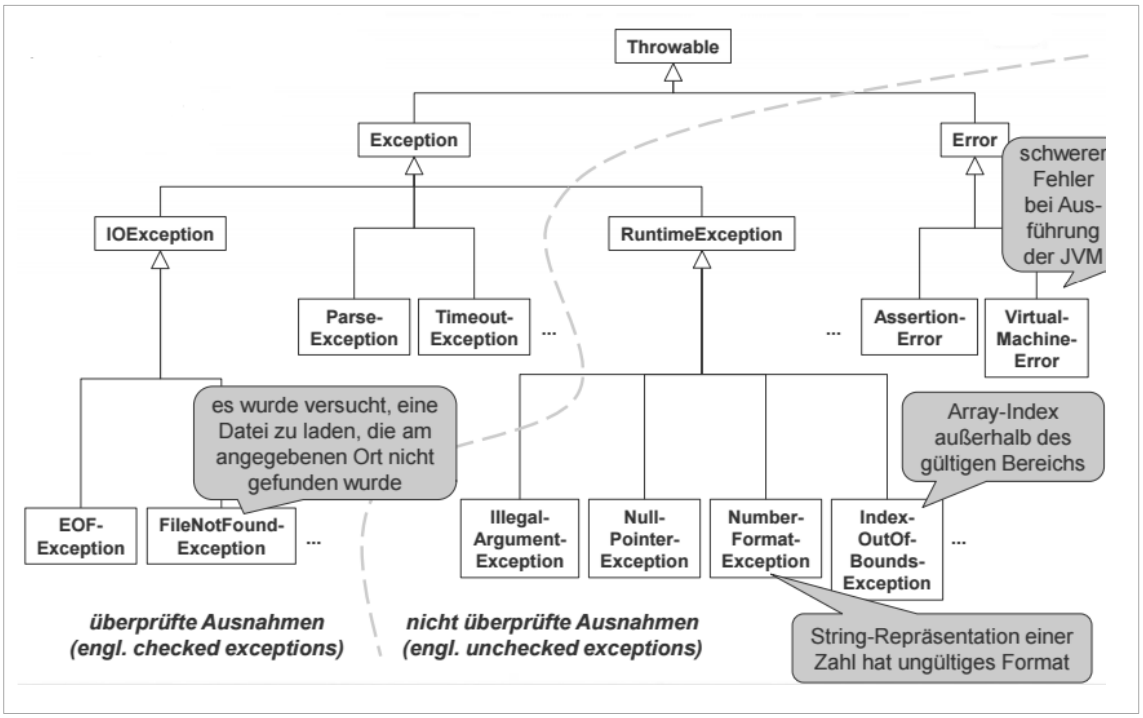
Ausnahmetypen sind in Java gewöhnliche Klassen;
Vorteile:

- Objekt kann Information über den Fehler (z. B. Fehlerbeschreibung, Ort des Auftretens etc.) an die Stelle transportieren, wo der Fehler behandelt wird.
- Über den Typ des catch-Parameters kann gezielt angegeben werden, welche Ausnahmen gefangen werden sollen.

Ausnahmetypen erben alle von der vordefinierten Klasse java.lang.Throwable
java.lang.Throwable hat (genau) zwei direkte Unterklassen:

- java.lang.Error zeigt Probleme bei der Programmausführung an, die das Programm i. d. R. nicht selbst beheben kann/sollte.
- java.lang.Exception ist die Oberklasse,unter der alle Ausnahmetypen zusammengefasst werden, die das Programm selbst behandeln möchte/sollte

Hierarchie in der Standardbibliothek definierter Ausnahmen



Nicht) überprüfte Ausnahmen (engl. (un-)checked exceptions)

- Überprüfte Ausnahmen (engl. checked exceptions)
  - Alle Ausnahmeklassen, die von der Klasse Exception mit Ausnahme von RuntimeException und deren Unterklassen abgeleitet sind.
  - Pflicht, solche Ausnahmen zu behandeln oder weiterzugeben.
  - Sonst Fehler bei der Übersetzung
- Nicht überprüfte Ausnahmen (engl. unchecked exceptions)
  - Alle Ausnahmeklassen, die von der Klasse Error oder RuntimeException sowie deren Unterklassen abgeleitet sind.
  - Deklaration nicht erforderlich, weil jede Methode RuntimeExceptions auslösen kann und Behandlung von Errors u. U. nicht möglich ist.

Ausnahmen behandeln

- try-Block umfasst Code, der Ausnahmen auslösen kann.
- Wird in einem try-Block eine Ausnahme geworfen, dann
  - wird die Ausführung des try-Blocks sofort abgebrochen.
  - Dann wird der Typ des Ausnahmeobjekts mit den Typangaben der nachfolgenden catch-Klauseln verglichen. Von oben nach unten.
  - Der erste catch-Block bei dem der Ausnahemtyp bzgl. Instanceof zur Typmenge passt (Untertyp passt auch!), behandelt die Ausnahme.
  - Der catch-Block gibt dem Ausnahmeobjekt einen Namen (analog formaler Parameter bei Methodendeklarationen).
  - Passt keiner der catch-Blöcke oder löst der catch-Block selbst eine Ausnahme aus, so wird die Ausführung der try-catch-Anweisung abgebrochen und die letzte Ausnahme an den Aufrufer übergeben (ggf. nachdem ein sog. finally-Block bearbeitet wurde, folgt gleich).
- Falls keine Ausnahme ausgelöst wurde oder nach Bearbeitung der passenden catch-Klausel wird (ggf. nach dem finally-Block) mit den nachfolgenden Anweisungen fortgesetzt.

Ausnahmen behandeln am Beispiel

```
try {
        ... // Code, der Ausnahmen ausloesen koennte
}
catch(IllegalArgumentException iae) {
        ... // Behandlungscode
}
catch(EOFException | ZipException fileEx) {
        ... // Behandlungscode
}
catch(Exception e) {
        ... // Behandlungscode
}
finally {
        ... // Abschlussarbeiten
}
System.out.println("Have a nice day!")
```

Der finally-Block

- Manchmal soll im Anschluss an den Code im try-Block ein Stück Code auf jeden Fall ausgeführt wird, egal ob im try-Block (oder in dem diese bearbeitenden catch-Block) eine Ausnahmesituation auftrat oder nicht. Z. B. Datenbanktransaktion rückabwickeln.
- Solcher Code kann am Ende der try-catch-Anweisung nach den catch-Klauseln in einem sog. finally-Block angegeben werden.

```
public static void m() {
        throw new RuntimeException("some exception");
}
public static void main(String[] args) {
         try {
                m();
         } catch (RuntimeException re) {
                // Behandlungscode
         } finally {
                // Dieser Code wird auf jeden Fall ausgefuehrt.
                System.out.println("Print this - no matter what!");
         }
}
```

Besondere try-Anweisung für zu schließende Ressourcen

- Für Objekte, die das Interface java.lang.AutoCloseable implementieren (und deshalb eine close()-Methode anbieten):

```
try (FileOutputStream fos = new FileOutputStream("out.txt");
         FileOutputStream bar = new ...) {
         fos.write(bytes, 0, bytes.length);
         bar.write(bytes, 0, bytes.length);
         fos.flush();
         bar.flush();
}
```

Anstoßen der Ausnahmebehandlung

- Die ausgeführte Methode wird sofort abgebrochen.
- Wenn die Methode besonderen Code für die Ausnahmesituation bereit hält (= Ausnahmebehandlung), dann wird dieser ausgeführt
- andernfalls:
    - Das Ausnahmeobjekt wird als eine Art Methodenresultat an die aufrufende Methode gegeben („die aufgerufene Methode wirft eine Ausnahme“).
    - Entweder wird dort die Ausnahmesituation behandelt oder das Ausnahmeobjekt wird zur nächsten Aufrufermethode weitergegeben usw.
    - Wird main()-Methode erreicht und auch dort kein passender Behandlungsblock gefunden, endet das Programm mit einer Fehlermeldung.

Ausnahmen weitergeben

```
import java.io.*;
class ExceptionDemo {
        // Deklarationen ...
        public static void write() {
               PrintWriter datei = new PrintWriter("daten.txt");
               ...
        }
        public static void main(String[] args) {
               try {
                      write();
               } catch (FileNotFoundException e) {
                      ... // Behandlungscode
               }
        }
}
```

- Möglichkeit 1: Ausnahme in der verwendenden Methode mit try-catch abfangen. (Möglichkeit hier nicht gewählt.)
- Möglichkeit 2: Ausnahme an aufrufende Methode weiterleiten (hier: main()) unter Verwendung einer throws-Klausel, die dem Übersetzer mitteilt, dass Methode Ausnahme des angegebenen Typs wirft.

Ausnahmen weitergeben

- Abhängig vom Typ der Ausnahme ist eine Behandlung oder Weitergabe erforderlich (bei überprüften Ausnahmen).
- Falls Behandlung (mittels catch-Block) nicht in der Methode selbst erfolgt, ist Weitergabe erforderlich.
- Methodenkopf deklariert Liste der Ausnahmtypen <ExceptionClass1>, …, <ExceptionClassn> die ggf. aus der Methode herausgegeben werden.
- Diese Liste ist Teil der öffentlichen Schnittstelle, damit Aufrufer weiß, welche Ausnahmen in der Methode auftreten können.
- Form:

```
<ModifierList> <MethodName>(<ParameterList>)
        throws <ExceptionClass1>, …, <ExceptionClassn>
```

- In der API-Dokumentation des JDK sind die von einer Methode ausgelösten Ausnahmen durch Angabe von throws in der Methodendeklaration erkennbar (und außerdem am Ende der Methodenbeschreibung einzeln und mit genauen Auslösebedingungen aufgeführt).
- Beispiel: Integer.parseInt

```
public static int parseInt(String s)
       throws NumberFormatException
```

- Nicht-überprüfte Ausnahmen (vom Typ java.lang.RuntimeException und java.lang.Error oder deren Unterklassen) können (aber müssen nicht) deklariert werden.

Methoden durch Ausnahmen absichern

```
class Zufallszahl {
        public static int zufallszahl(int min, int max) {
              ...
        }
        public static void main(String[] args) {
              System.out.println("Test der Zufallszahlen-Funktion");
               try {
                      int n = 0;
                      for (int i = 5; i > -5; i--) {
                            n = zufallszahl(0, i);
                            System.out.println(n);
                      }
               } catch(IllegalArgumentException e) {
                      System.err.println(e.getMessage());
                      System.err.println("Programm wird beendet");
               }
        }
}
```

Programmabsturz verhindern

```
class Zufallszahl {
        public static int zufallszahl(int min, int max) {
              ...
        }
        public static void main(String[] args) {
               System.out.println("Test der Zufallszahlen-Funktion");
               int n = 0;
               for (int i = 5; i > -5; i--) {
                      try {
                             n = zufallszahl(0, i);
                            System.out.println(n);
                      } catch(IllegalArgumentException e) {
                            System.err.println(e.getMessage());
                      }
               }
        }
}
```

- Aufruf und Ausgabe gehören logisch zusammen.
- Würde man die Ausgabe unter den catch-Block platzieren, würde in den Schleifendurchgängen, die zu einer Ausnahme führen, der jeweils alte Wert von n ausgegeben
- Schleife wird nicht mehr vorzeitig beendet

Eigene Ausnahmeklassen erstellen

- Die eigene Ausnahmeklasse muss von java.lang.Throwable erben:
  - Die Klasse sollte eine Unterklasse von java.lang.Exception sein.
  - java.lang.Error sollte man nicht nutzen, da man eigene (anwendungsorientierte) Ausnahmen davon unterscheiden können sollte.
- Ein Ausnahmeobjekt sollte den Fehlerzustand aufnehmen können, also muss man entsprechende Attribute vorsehen.
  - Standardmäßig können java.lang.Exception-Objekte optional eine Fehlernachricht (String) und/oder eine andere (verursachende) Exception aufnehmen (die z.B. in einer aufgerufenen Methode geworfen wurde sowie in der aktuellen Methode mit weiteren Informationen angereichert und anschließend nach oben weitergereicht wird).
  - z. B. throw new Exception("my exception message");

Eigene Ausnahmeklassen erstellen

```
public class Account {
        public static final double MAX_AMOUNT = 1000000;
        private double balance = 0;
        private double maxDebt = 0;
        ...
        public double getBalance() {
              return balance;
        }
        public void changeAmount(double amount) throws AccountException {
               // Zu viele Schulden verboten
               if (balance + amount < maxDebt)
                     throw new AccountException(amount);
               // Zu hohe Einzahlung verboten
               if (amount > MAX_AMOUNT)
                     throw new AccountException(amount);
               balance += amount;
        }
}
```

```
public class AccountException extends Exception {
        // positiv bei Einzahlungen
        // negativ bei Abhebungen
        private double invalidAmount;
        public AccountException(double invalidAmount) {
              this.invalidAmount = invalidAmount;
        }
        public double getInvalidAmount() {
              return invalidAmount;
        }
}
```

Ausnahmedeklaration und Überschreiben

- Wenn eine überschriebene Methode den Ausnahmetyp X deklariert, dann kann die überschreibende Methode
  - ganz auf die Deklaration des Ausnahmetyps X verzichten,
  - ebenfalls eine Ausnahme vom Typ X deklarieren oder
  - eine Ausnahme von einem Untertyp von X deklarieren.
- Die überschreibende Methode darf keine neuen Ausnahmetypen oder Obertypen hinzufügen.

```
public abstract class A {
        public abstract void m() throws Exception;
}
abstract class B extends A {
       public abstract void m(); // OK
}
abstract class C extends A {
       public abstract void m() throws NullPointerException; // OK
}
abstract class D extends A {
       public abstract void m() throws Throwable; // nicht OK
}
```

Funktionstest (Black-Box-Test)

- Testen aufgrund externer Betrachtung:
  - Beobachtbares Verhalten (z. B. Ausgaben) des Testobjekts wird bei bestimmten Eingaben betrachtet (innere Abläufe werden ignoriert).
  - Verhalten wird mit dem nach Spezifikation erwarteten verglichen.
  - Grundlage für die Wahl der Testfälle ist einzig die Aufgabenbeschreibg.
- Vorteile:
  - Testfälle können unabhängig von der Implementierung erstellt werden.
- Nachteile:
  - Mögliche kritische Pfade in der Implementierung werden nicht erkannt und werden daher vielleicht nicht getestet.
- Häufig: Überprüfung spezifizierter Funktionen mit typischen Werten (Äquivalenzklassentest) und Randwerten (Grenzwerttest)
  - z. B. sehr kleine oder große double-Werte („gerade noch gültig“), leere Benutzereingaben, …

Strukturtest (White-Box-Test)

- Testen aufgrund interner Betrachtung:
  - Programmablauf wird bei Eingaben betrachtet, die unter Kenntnis des inneren Ablaufs so gewählt sind, dass sie best. Testkriterien erfüllen.
  - Sinnvolles Vorgehen: wähle Testfallmenge, welche z. B. jede Anweisung, jeden Zweig (then/else), jeden Pfad etc. mind. einmal durchläuft.
  - Grundlage für die Wahl der Testfälle ist einzig der Programmcode (die Spezifikation dient höchstens noch der Ergebnisüberprüfung).
- Vorteile:
  - Alle (vorhandenen) Programmteile können getestet werden.
- Nachteile
  - Fehlende Pfade können nicht getestet werden (Fehlen wird im Allg. nur d. externe Betrachtung aufgedeckt).
  - Anzahl aller erforderlichen Testziele (z. B. Pfade durch ein Programm) ist im Allg. zu hoch → kein vollständiges Testen möglich.

Integrationstest, Regressionstest

Integrationstest (oft auch: „Gray-Box-Test“):

- dient zur Überprüfung der Schnittstellen zwischen den Modulen bzw. Komponenten und testet somit die Interaktion zwischen zwei oder mehreren Komponenten (welche einzeln für sich schon im vorangehenden Modultest geprüft worden sind).

Regressionstest:

- bei Änderungen (z. B. Fehlerkorrektur, Erweiterung) eines Programms sollten alle alten, von der Änderung nicht betroffenen Tests weiterhin funktionieren.
  - Idealerweise automatisiert: alle Tests mit deren Ergebnissen werden gespeichert; bei Weiterentwicklung des Programms werden sie wieder durchgeführt und mit den alten Ergebnissen verglichen.
  - Ziel: Test, ob durch die Änderungen neue Fehler eingeführt wurden.

Zusicherungen (engl. assertions)

- Während der Programmentwicklung ist es i.d.R. möglich, Bedingungen zu formulieren, die am Beginn oder am Ende einer Methode oder während der Ausführung einer Schleife gelten müssen, damit die Methode korrekt arbeitet (sog. Vor- bzw. Nachbedingung, Schleifeninvariante).
- Solche Bedingungen können als Behauptungen während der Programmentwicklung (genauer: in der Testphase) automatisiert mittels sog. Zusicherungen (engl. assertions) geprüft werden.
- Zusicherungen
  - boolesche Funktionen für Vor- und Nachbedingungen sowie Invarianten,
  - werden zur Laufzeit ausgewertet und machen sich im Fehlerfall durch eine Ausnahme bemerkbar,
  - stehen meist am Anfang oder Ende von Methoden,
  - sollten unbedingt das eigentliche Programm nicht beeinflussen,
  - decken Fehler vielleicht in der Testphase auf, werden vorher konzipiert.

Syntax und Semantik von Zusicherungen

- AssertStatement
  - assert Expression1
  - assert Expression1 : Expression2
- Expression1 ist dabei ein boolescher Ausdruck und Expression2 ein Fehlermeldungsausdruck (z. B. eine Fehlermeldung (String)).
- Semantik:
  - Falls Expression1 zu true ausgewertet wird, erfolgt keine weitere Aktion.
  - Sonst Auslösen einer Exception vom Typ java.lang.AssertionError; Fehlermeldungsausdruck Expression2 wird Ausnahmeobjekt übergeben und kann mittels getMessage() ausgelesen werden.

Aktivier- und Deaktivierbarkeit von Zusicherungen

- Zusicherungen sind prinzipiell mit if-Anweisungen nachbildbar.
- Aber: benötigter Code ist umfangreicher, nicht auf den ersten Blick erkennbar, dass es sich um eine Bedingungsprüfung handelt, if-Anweisungen sind nicht deaktivierbar.
- Verwendung von Zusicherungen ist aktivier- und deaktivierbar.
  - Ausführung ohne Zusicherungen (schneller, Standardeinstellung):
    - java -disableassertions MyProgram
    - java -da MyProgram
  - Ausführung mit Zusicherungen (langsamer, zum Testen):
    - java -enableassertions MyProgram
    - java –ea MyProgram

Vermeidung von Seiteneffekten

- Zusicherungen dienen während der Programmentwicklung zur Überprüfung, ob die verwendeten Methoden mit den korrekten Datenwerten arbeiten.
- Bei der Ausführung beim Anwender sollten sie i. d. R. nicht eingesetzt und korrekter Programmablauf sollte dort anders sichergestellt werden.
- Deshalb:
  - Zusicherungen sind aktivierbar und deaktivierbar.
  - Zusicherungen sollten niemals Seiteneffekte besitzen, die die normale Programmausführung beeinflussen

Benutzung von Zusicherungen

- Test von Nachbedingungen
  - Vor jeder return-Anweisung, über die eine Methode verlassen werden kann, bzw. vor letzter Anweisung einer Methode erfolgt Überprüfung von Nachbedingungen.
  - Beispiele:
    - Prüfung, ob berechneter Wert innerhalb eines definierten Intervalls liegt.
    - Ist eine Wertfolge wirklich sortiert?
    - Liefert eine Methode leereFeld das gewünschte Ergebnis?

```
private void leereFeld(ArrayList al) {
     // Operationen zum Leeren eines Feldes
     // ...
     assert al.isEmpty(): "Das Feld ist nicht leer";
}
```

- Prüfung des Kontrollflusses
  - Prüfung bei if-Anweisungen, ob bestimmter Zweig ausgeführt wird, der eigentlich nicht zur Ausführung kommen darf.
- bei switch-Anweisungen über default-Zweig testen, ob die verwendeten case-Zweige alle Fälle abdecken.
- Prüfung der Parameter
  - von öffentlichen Methoden
    - Falscher Aufruf ist nie unerwartet: rechne mit dem Schlimmsten!
    - Statt Zusicherungen sollte hier IllegalArgumentException (Unterklasse von RuntimeException) verwendet werden, damit Aufrufer angemessen reagieren können.
  - von geschützten Methoden (private, protected)
    - Falsche Parameter sind Fehler; daher während der Entwicklung Parameterprüfung oder Zustandstests mittels Zusicherungen.
    - Integrierte Überprüfung der übergebenen Parameterwerte und Auslösen von Exceptions im Fehlerfall während des echten Betriebs „verzichtbar“

Formale Verifikation mittels wp-Kalkül Grundidee

- Durch jeweils eine konkrete Belegung der in einem Programm(stück) enthaltenen Variablen ist ein konkreter Zustand dieses Programm(stück)s gegeben.
- Im Allgemeinen interessiert man sich für einen oder mehrere Zielzustände, in denen bestimmte Regeln bzgl. der Variablenwerte gelten (sog. Nachbedingung, engl. post-condition).
- Die Anweisungen des Programmstück(s) werden von der Nachbedingung ausgehend rückwärts analysiert und je nach Typ der Anweisung wird gefolgert, welche Bedingungen bzgl. der Variablenwerte vor der Anweisung mindestens gegolten haben müssen, damit die jeweilige Bedingung nach der Anweisung gerade noch gilt.
- Dieses Vorgehen wird schrittweise bis vor die erste Anweisung des Programm(stück)s wiederholt.
- Folgt die dort geltende Bedingung aus der sog. Vorbedingung (engl. pre-condition), dann arbeitet das Programm korrekt.

Programmzustand

- Ein konkrete Belegung dieser Variablen mit Werten bildet einen sog. Zustand dieses Programm(stück)s.
- Das kartesische Produkt der Wertemengen der Variablen bildet damit den Zustandsraum des Programm(stück)s.

Verifikation, partielle und totale Korrektheit

- Eine Verifikation prüft, ob ein Programm bestimmte sog. Zusicherungen erfüllt, d. h. ob die Nachbedingung Q einer Anweisungsfolge A nach Abarbeitung der Anweisungen aus der Vorbedingung P ableitbar ist.
- Eine Anweisungsfolge A heißt partiell korrekt, wenn die Ausführung von A in einem Zustand, der P genügt, beginnt und falls die Ausführung von A nach endlich vielen Schritten terminiert, dann ein Zustand erreicht wird, der Q erfüllt (Schreibweise: P {A} Q).
- Hinweis: der ganze Ausdruck P {A} Q ist eine logische Formel, die wahr oder falsch sein kann.
- Eine Anweisungsfolge A heißt total korrekt, wenn sie partiell korrekt ist und zusätzlich nachgewiesen ist, dass A immer terminiert (Schreibweise: {P} A {Q}).

Verifikation, partielle und totale Korrektheit

- Es seien I_D die Menge der möglichen Eingabedaten und O_D die Menge der möglichen Ausgabedaten eines Programms.
- Die Vorbedingung ist ein Prädikat P auf I_D
- Die Nachbedingung ist ein Prädikat Q auf $O_D \times I_D$
- Es sei A : $I_D \rightarrow O_D$ die Abbildung, die die Eingabedaten des Programms in die Ausgabedaten überführt. Annahme dabei: Algorithmus A terminiert!
- Notation:
  - Falls gilt: $\forall x \in I_D$: P(x) = true $\Rightarrow$ Q(A(x), x) = true dann schreibt man: {P} A {Q} (für totale Korrektheit v. A)
  - Das bedeutet: Für alle Startzustände x für die P gilt, gilt Q nach Ausführung von A.
- P und Q bilden die sog. prädikatenlogische Spezifikation.

Vor- und Nachbedingung am Beispiel

```
// x ist ungerade
// {P: (x % 2) == 1}
x = x + 1;
// x ist gerade
// {Q: (x % 2) == 0}
```

- Nachbedingung Q (von uns gesetzt): x soll gerade sein.
- Aus Q und der Anweisung folgt, dass (x – 1) % 2 = 0 vor der Anweisung gelten muss, woraus direkt P folgt.
- Resultierende Vorbedingung P: x muss vor der Anweisung ungerade gewesen sein.
- P , Q : Beispiele für sog. Prädikate; Prädikate sind hier Aussagen bzgl. im Programmstück enthaltener Variablen, die erfüllt oder nicht erfüllt sein können.

Vor- und Nachbedingung am Beispiel (II)

- $\{P: i \text{ beliebig}\}$ i=7; $\{Q: i = 7\}$
  - Nach endlich vielen (hier: einem) Schritt terminiert die Anweisung.
  - Wenn vor Ausführung der Zustand P erfüllt war, dann ist nach Ausführung der Zustand Q erfüllt.
- $\{P: i \text{ beliebig}, j \text{ beliebig}\}$ j=2; i=7; $\{Q: i = 7 \wedge j = 2\}$
- $\{P: i \text{ beliebig}, j \text{ beliebig}\}$ i=2; i=7; $\{Q: i = 7 \wedge j \text{ beliebig}\}$
  - j hier in P und Q beliebig: will man etwas über j wissen?
  - falls nein: $\{P: \text{wahr}\}$ i=2; i=7; $\{Q: i = 7\}$
- Also: man muss nicht nur bedenken, was die einzelne Anweisung bewirkt, sondern auch, welche Teile der Vorbedingung in die Nachbedingung übernommen bzw. darin verändert werden.

Formalisierung: Verifikation, wp-Kalkül

- wp-Kalkül (wp: weakest precondition = schwächste Vorbedingung):
- Seien A und Q gegeben. Das schwächste Prädikat P (also das Prädikat, das bei möglichst vielen Eingabedaten x wahr ergibt),für das P { A} Q gilt, heißt $wp\ (A,Q) := P$.
- Das bedeutet:
  - Wähle also die Eingabewerte so, dass sie P erfüllen und wähle P so, dass es die schwächste Vorbedingung zu Q ist, die sich aus der Anweisungsfolge A ergibt.
  - Mit diesem Prädikat kann man also feststellen, für welche Eingabewerte das Programm richtig rechnet
- P heißt schwächer als P genau dann, wenn gilt: $P \Rightarrow P'$
- Idee: Untersuche statt vieler Vorbedingungen nur die schwächste Vorbedingung (weakest precondition), denn ausgehend von dieser wird die Nachbedingung „gerade noch“ erfüllt.

Einschub: einige logische Äquivalenzen

De Morgan‘sche Gesetze

- $\neg(A \wedge B) \Leftrightarrow \neg A \vee \neg B$
- $\neg(A \vee B) \Leftrightarrow \neg A \wedge \neg B$

Distributivgesetze

- $A \wedge (B \vee C) \Leftrightarrow (A \wedge B) \vee (A \wedge C)$
- $A \vee (B \wedge C) \Leftrightarrow (A \vee B) \wedge (A \vee C)$

Äquivalenz

- $(A \Leftrightarrow B) \Leftrightarrow (A \wedge B) \vee (\neg A \wedge \neg B)$

Implikation

- $(A \Rightarrow B) \Leftrightarrow \neg A \vee B$
- $(A \Rightarrow B) \Leftrightarrow (\neg B \Rightarrow \neg A)$

Vorgehen bei der Programmverifikation mit wp-Kalkül

- Das wp-Kalkül ist ein sog. Verifikationskalkül
- Verifikationskalküle
  - verwenden (rein syntaktische) Regelwerke zur Ableitung zu verifizierender Aussagen.
  - Regeln sind rein mechanisch anwendbar.
- Also:
  - Wir benötigen für jeden Anweisungstyp (z. B. Zuweisung, Fallunterscheidung, Schleife) eine Regel zur Herleitung der jeweils schwächsten Vorbedingung aus einer gegebenen Nachbedingung.
  - Dann lässt sich anhand dieser Regeln aus einer Nachbedingung einer Anweisungsfolge deren schwächste Vorbedingung bestimmen.
  - Da imperative Programme aus einer linearen Abfolge von Kontrollstrukturen bestehen, folgt aus der Korrektheit von aufeinander folgend ausgeführten Teilen die Korrektheit der Sequenz dieser Teile.

Zuweisungsregel

$wp(\text{"x := e"}, Q) = Q[x/e]$

- Vorgehensweise:
  - Zur Konstruktion der Vorbedingung wird mit der Nachbedingung Q begonnen, darin werden alle Vorkommen der Variablen x durch den Ausdruck e substituiert.
  - Man simuliert sozusagen den Effekt symbolisch.
- Hinweis: Die Nachbedingung Q kann nur erreicht werden, wenn sowohl x als auch e fehlerfrei ausgewertet werden können und deren Auswertung jeweils seiteneffektfrei bzgl. Q ist.

Beispiele zur Zuweisungsregel

- wp(" a := 7", a = 7) = (**7** = 7) = true
  - im Prädikat aa = 7 wird die Variable aa gemäß der Zuweisung a: = 7 durch 7 ersetzt
- wp(" a := 7" , a = 6) = (**7** = 6) = false
  - wenn das Prädikat a = 6 gelten soll, zuvor aber aa den Wert 7 zugewiesen bekommt, dann kann das Prädikat a = 6 nicht erfüllt werden
- wp(" a := 7" , b = 12) = (b = 12)
  - gilt auch allgemeiner: wp("a := 7", Q) = Q
  - Voraussetzung: das Prädikat Q enthält die Variable a nicht
  - das Prädikat Q wird durch die Zuweisung also nicht verändert
- wp(" a := b" , a > 1) = **b** > 1
  - wenn nach Ausführung von a: = b gelten soll: a > 1, dann muss vorher mindestens b > 1 gegolten haben.

```
public static int ganzzahligeQuadratWurzel(int r) {
        // {P: r >= 1}
        int s = 0, i = 0, x = 0;
        while (s <= r) {
               i++;
               s = s + (2*i - 1);
        }
        // {Q': (i-1)*(i-1) <= r && r < i*i}
        x = i - 1;
        // {Q: x*x <= r && r < (x+1)*(x+1)}
        return x;
}
```

Beispiel zur Verifikation: Zuweisung

- Wenn Vorbedingung P die Terminierung von A garantiert und wenn P { A } Q gilt, dann gilt auch: $P \Rightarrow wp(A,Q)$ und somit: {wp (A,Q) } A { Q }
- Beispiel:
  - zu beweisen: {P: 10 < y < 100} x := y {Q: x > 1}
  - Es gilt: wp("x := y", x > 1) = ($\boldsymbol{y}$ > 1)
    - Wenn nach Ausführung von x := y gelten soll: x > 1, dann muss vorher mindestens y > 1 gegolten haben.
    - Anders ausgedrückt: { wp("x := y", x > 1) } x := y { x > 1 }
  - Wegen (10 < y < 100) ⇒ (y > 1) gilt: $P \Rightarrow wp(A,Q)$

Sequenzenregel

$wp(\text{"}\mathtt{s_1;\ s_2}\text{"}, Q) = wp(\text{"}\mathtt{s_1}\text{"}, wp(\text{"}\mathtt{s_2}\text{"}, Q))$

- Vorgehensweise:
  - s1; s2 bedeutet: führe die Anweisung $s1$ aus und nach deren Terminierung führe die Anweisung $s2$ aus.
  - Q wird als Nachbedingung der Anweisung $s2$ betrachtet und deren schwächste Vorbedingung $wp(s2,Q)$ bestimmt.
  - Das dabei entstehende Prädikat wird dann zur Nachbedingung der Anweisung $s1$ und daraus deren schwächste Vorbedingung $wp(s1, wp(s2,Q))$ bestimmt.
- Hinweis: die Sequenzenregel gilt analog für längere Sequenzen
- Also: zwei (korrekte) Programmstücke s1 und s2 können zu einem (korrekten) Programmstück s1;s2 zusammengesetzt werden, wenn die Vorbedingung von s2 aus der Nachbedingung von s1 folgt

Beispiele zur Sequenzenregel

- Sei s1; s2 gegeben durch " a := a + b; b := a * b" und sei die Nachbedingung irgendein Prädikat $Q(a, b)$

$\square\ wp(s_2, Q(a,b))$
$= wp(\text{"}\mathtt{b := a * b}\text{"}, Q(a,b))$ — Zuweisungsregel
$= Q(a, a \cdot b)$
$\square\ wp(\text{"}\mathtt{s_1; s_2}\text{"}, Q(a,b))$ — Sequenzenregel
$= wp(s_1, wp(s_2, Q(a,b)))$
$= wp(s_1, Q(a, a \cdot b))$
$= wp(\text{"}\mathtt{a := a + b}\text{"}, Q(a, a \cdot b))$ — Zuweisungsregel
$= Q(a+b, (a+b) \cdot b)$

- Das bedeutet: wenn als Nachbedingung $Q(a, b)$ gelten soll, dann muss als Vorbedingung mindestens dieselbe Relation Q zwischen $a + b$ und $a + b \cdot b$ gegolten haben.

Beispiel zur Verifikation: Wertetausch von zwei Variablen

Vertauschen der Werte der Variablen x und y zu verifizieren: {P: $y = X \wedge x = Y$} h := x; x := y; y := h; {Q: $x = X \wedge y = Y$}

$wp(\text{"}\mathtt{h := x;\ x := y;\ y := h;}\text{"},\ x = X \wedge y = Y)$
$= wp(\text{"}\mathtt{h := x}\text{"}, wp(\text{"}\mathtt{x := y}\text{"}, wp(\text{"}\mathtt{y := h}\text{"}, x = X \wedge y = Y)))$
$= wp(\text{"}\mathtt{h := x}\text{"}, wp(\text{"}\mathtt{x := y}\text{"}, x = X \wedge h = Y))$
$= wp(\text{"}\mathtt{h := x}\text{"}, y = X \wedge h = Y)$
$= (y = X \wedge x = Y)$

Fallunterscheidung: if-Regel

- Die if-Regel lautet:
  - bei einer Fallunterscheidung mit then- und else-Teil:
    - wp("if b then s1 else s2",Q) = $b \wedge$ wp($s1$,Q) $\vee\ \neg b \wedge$ wp($s2$,Q)
  - bei einer Fallunterscheidung nur mit then-Teil:
    - wp("if b then s1", Q) = [$b \wedge$ wp($s1$,Q)] $\vee$[$\neg b \wedge$ Q]
- Erläuterung
  - Damit der then-Teil der if-Anweisung ausgeführt wird, muss jeweils vorher b true gewesen sein und damit nach der Anweisung $s1$ im then-Teil Q gilt, muss vorher mindestens $wp(s1,Q)$ gegolten haben.
  - Für den else-Teil muss b false gewesen sein und für die Anweisung $s2$ gilt Analoges wie für $s1$.
  - Bei der Fallunterscheidung nur mit then-Teil ist $s2$ die leere Anweisung.
  - b muss seiteneffektfrei bzgl. Q sein, z. B. nicht seiteneffektfrei wäre wp("if (--a > 0) then ..." , $a > 0$)

Beispiele zur if-Regel

Bestimmung des Maximums zweier Zahlen
Was muss vor der folgenden if-Anweisung mindestens gelten, damit nachher $\mathrm{max} = x$ erfüllt ist?

$wp(\text{"}\mathtt{if\ (x>y)\ then\ max := x\ else\ max := y}\text{"}.\ \mathrm{max} = x)$
$= (x > y \wedge wp(\text{"}\mathtt{max := x}\text{"}, max = x))$
$\quad \vee\ (x \leq y \wedge wp(\text{"}\mathtt{max := y}\text{"}, max = x))$
$= (x > y \wedge x = x) \vee (x \leq y \wedge y = x)$
$= (x > y \vee x = y)$
$= (x \geq y)$

Schleifeninvariante

- Ziel: Prüfung der Korrektheit einer Schleife der Form while b { A } (mit Bedingung b und Schleifenrumpf A)
- Gegeben:
  - Prädikate P,Q zur Spezifikation dessen, was die Schleife leisten soll
- Gesucht:
  - wp("while b { A }",Q) =?
  - Falls dieses Prädikat schwächer als P ist, dann ist das Programm korrekt.
- Definition:
  - Eine Invariante der Schleife: while (b) { A } ist ein Prädikat I, für das gilt: { $I \wedge b$ } A { I }
- Eigenschaften:
  - Wenn b erfüllt ist, dann muss unmittelbar vor dem Schleifenrumpf A auch b gelten; nach A (vor der nächsten Iteration) kann b gelten oder nicht.
  - Wenn vor dem Schleifenrumpf $I \wedge b$ gilt, dann ist I nach der Ausführung von A erfüllt; I ist also vor und nach dem Schleifenrumpf erfüllt, daher auch Bezeichnung als Invariante.
  - In wp-Terminologie muss man zeigen: $I \wedge b \Rightarrow wp(A,I)$ d. h.: $I \wedge b$ muss also stärker sein, als die schwächste Vorbedingung des Schleifenrumpfs.

Vorgehensweise

- Vier Schritte zum Nachweis der Korrektheit:
  - Beweise, dass I vor Eintritt in die Schleife gilt.
  - Beweise, dass { $I \wedge b$ } A { I } gilt, dass also I tatsächlich eine Invariante ist.
  - Beweise, dass ($I \wedge \neg\, b$) $\Rightarrow Q$, so dass bei Terminierung das gewünschte Ergebnis erreicht wird.
  - Beweise, dass die Schleife terminiert (totale Korrektheit) (mittels Terminierungsfunktion (auch: (sic!) Schleifenvariante)).

Hinweis zum Finden von Schleifeninvarianten

- Das Finden geeigneter Schleifeninvarianten ist ein kreativer und erfahrungsbasierter Prozess, für den es kein algorithmisches Verfahren gibt.
- Das Finden geeigneter Schleifeninvarianten ist daher ein Prozess, den man nur durch das Bearbeiten vieler Beispiele wirklich erlernen kann.
- Je genauer man verstanden hat, was eine Schleife wirklich leisten soll, desto leichter fällt es, eine geeignete Schleifeninvariante anzugeben.
- Oft hilfreiche Heuristik:
- Man betrachtet die Variablen, die in der Schleife verändert werden, und setzt diese in Relation zueinander bzw. zu den anderen Variablen (insbes. Eingaben) im Programm.
- Die Invariante muss stark genug sein, um die Nachbedingung daraus ableiten zu können, daher liefert die Nachbedingung gute Hinweise.

Schleifeninvariante finden

```
public static int ganzzahligeQuadratWurzel(int r) {
        int s = 0, i = 0, x = 0;
        while (s <= r) {
              i++;
              s = s + (2*i - 1);
        }
        x = i - 1;
        return x;
}
```

- Offenbar gilt in jeder Schleifeniteration: $i^2 = s$.
- Oben hatten wir bereits gezeigt, dass nach der Schleife gelten muss: $(i - 1)^2 \leq r \wedge r < i^2$
- Dies muss aus $I \wedge \neg b$ folgen: wählt man I zu schwach, gelingt der Nachweis $I \wedge \neg b \Rightarrow Q$ (folgt gleich) nicht.
- Verwende also: $I :\equiv i^2 = s \wedge (i - 1)^2 \leq r$

1. Beweise, dass I vor Eintritt in die Schleife gilt

$wp(\texttt{"s = 0; i = 0; x = 0"}, (i^2 = s) \wedge ((i-1)^2 \leq r))$ — Sequenzenregel

$= wp(\texttt{"s = 0"}, wp(\texttt{"i = 0"}, wp(\texttt{"x = 0"}, (i^2 = s) \wedge ((i-1)^2 \leq r))))$

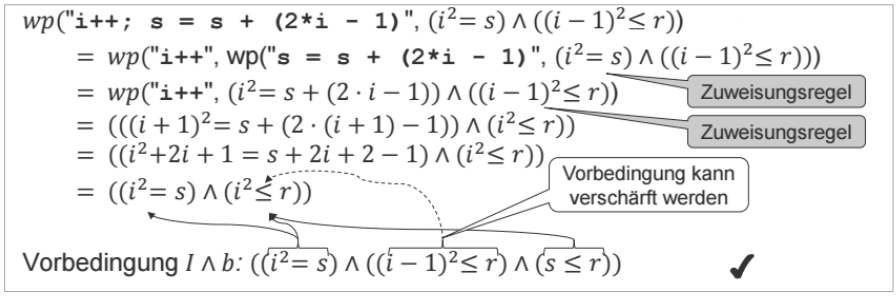
$= wp(\texttt{"s = 0"}, wp(\texttt{"i = 0"}, (i^2 = s) \wedge ((i-1)^2 \leq r)))$ — Zuweisungsregel

$= wp(\texttt{"s = 0"}, (0 = s) \wedge (1 \leq r))$ — Zuweisungsregel

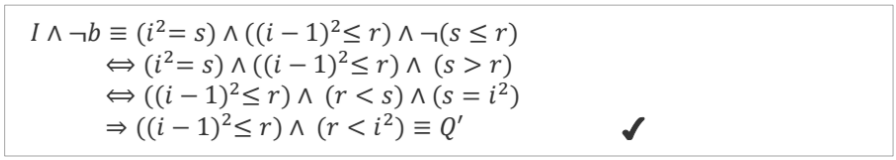
$= ((0 = 0) \wedge (1 \leq r))$ — Zuweisungsregel

$= (r \geq 1)$ ✓

2. Beweise, dass $\{ I \wedge b \}$ A $\{ I \}$ gilt



3.Beweise, dass $(I \wedge \neg b) \Rightarrow Q$



Letzter Schritt nur Implikation (⇒), da Q‘ keine Aussagen über s macht und daher der Rückwärtsschluss (⇐) von Q‘ (ohne s) auf die vorletzte Zeile (mit s) nicht zulässig ist.

Schleifenvariante (auch: Terminierungsfunktion)

- Die relevanten Eigenschaften der Schleifenvariante sind also:
  - V ist eine geeignete Funktion auf den in der Schleife verfügbaren Variablen.
  - V ist ganzzahlig.
  - V ist streng monoton fallend.
  - V ist nach unten beschränkt (meistens durch Konstante c = 0).
- Das Finden einer passenden Schleifenvariante ist ebenso wie das Finden einer Schleifeninvariante ein kreativer und erfahrungsbasierter Prozess, der erst durch viele erarbeitete Beispiele oder „trial-and-fail“ wirklich verstanden wird.

4. Beweis, dass die Schleife terminiert

- Mögliche Schleifenvariante: $V := r - (i - 1)^2$
- Zeige $I \Rightarrow V \geq 0$:
  - es gilt: $V := r - (i - 1)^2 \in \mathbb{Z}$ (VV ist also ganzzahlig), da r und i int-Variablen sind
  - aus I folgt, dass $(i - 1)^2 \leq r$ (für $r \geq 1$) und damit $V := r - (i - 1)^2 \geq 0$
  - also: $I \Rightarrow V \geq 0$:
- Zeige $\{I \wedge b \wedge V = z\}$ A $\{0 \leq V < z\}$ mit $V, z \in \mathbb{N}_0$:
  - Nachweisen, dass V streng monoton fallend ist: wenn V nach dem k-ten Schleifendurchlauf den Wert z hat, folgt aus der geforderten Monotonie, dass V nach dem nächsten $((k + 1)$-ten) Schleifendurchlauf kleiner als z sein muss.
  - Halte den Wert von V_k in z fest: $V_k := r - (i - 1^2) = r - i^2 + 2i - 1 =: z$
  - Für $k \geq 1$ ist $i \geq 1$ und damit auch $2i - 1 > 0$
  - Nach einmaliger Ausführung des Schleifenrumpfes i++; s = s + (2*i - 1);
  - ergibt sich $V_{k+1} := r - ((i + 1) - 1^2) = r - i^2 = z - (2i - 1) < z$
  - Also: V ist streng monoton fallend

Checklisten zur Fehleridentifizierung und -vermeidung

- Variablen- und Konstantendeklarationen
  - Werden beschreibende (sog. mnemonische) Variablen- und Konstantennamen verwendet, die mit den Codier-Regeln übereinstimmen?
  - Gibt es Variablen mit ähnlichen Namen, die verwechselt werden könnten?
  - Ist jede Variable sauber initialisiert?
  - Gibt es Instanzvariablen, die besser als lokale Variablen von Methoden deklariert werden sollten?
  - Gibt es Literale, die lieber als Konstanten deklariert werden sollten?
  - Gibt es Konstanten, die noch nicht als final deklariert sind?
  - Sind Laufvariablen von for-Schleifen im Schleifenkopf deklariert?
  - Haben alle Instanz- und Klassenvariablen eine möglichst restriktive Sichtbarkeit?
  - Gibt es Instanzvariablen (Attribute), die besser Klassenvariablen (static) wären oder umgekehrt?
- Methodendeklaration
  - Entsprechen die Methodennamen der Codier-Regel und sind sie mnemonisch?
  - Werden die Argumente vor ihrer Verwendung auf Plausibilität geprüft?
  - Gibt jede Methode bei jedem Rückkehrpunkt den beabsichtigten Wert zurück?

- Haben alle Methoden eine möglichst restriktive Sichtbarkeit?
- Gibt es Objektmethoden, die besser Klassenmethoden (static) wären oder umgekehrt?

- Klassendeklaration
  - Gibt es Konstruktoren, die alle Klassenvariablen direkt initialisieren?
  - Kann die Vererbungshierarchie vereinfacht werden?
  - Kann man (weitere) Variablen oder Methoden aus Unterklassen in Oberklassen verschieben?
- Datenzugriff
  - Liegen Zugriffe auf Arrays (Reihungen) im gültigen Bereich und werden auch die Randwerte bearbeitet?
  - Ist für jeden Objekt-Zugriff gesichert, dass die Referenz „!= null“ ist?
  - Können Zahlenbereiche über- oder unterlaufen?
  - Stimmen bei Ausdrücken mit mehreren Operatoren die Annahmen über die Präzedenzregeln (Vorrangregeln)?
  - Stimmt die Klammerung?
- Vergleiche, boolean-Operatoren
  - Gibt es unbeabsichtigte Seiteneffekte?
  - Lässt sich der Ausdruck vereinfachen?
- Kontrollfluss
  - Ist die jeweils beste Schleifenanweisung gewählt?
  - Terminiert jede Schleife?
  - Wenn es mehrere Schleifenausgänge gibt, ist jeder korrekt?
  - Hat jede switch-Anweisung einen default-Fall?
  - Sind fehlende break-Anweisungen im switch gewollt und korrekt kommentiert?
  - Bei break und continue mit Label: stimmt das Ziel?
  - Übertriebene Verschachtelungstiefe (Fallunterscheidung, Schleife)?
  - Könnte verschachtelte if-Anweisung durch switch ersetzt werden?
  - Sind leere Rümpfe korrekt und wenn ja, sind sie mit {} markiert?
- Ein-/Ausgabe
  - Werden alle Dateien vor der Verwendung geöffnet?
  - Werden alle Dateien nach der Verwendung wieder geschlossen?
  - Gibt es Rechtschreib- oder Grammatikfehler in angezeigtem oder gedrucktem Text?
  - Werden alle Ausnahmen sinnvoll verarbeitet?
- Schnittstellen
  - Stimmen die Anzahl, Reihenfolge, Typen und Werte von Methodenargumenten mit der Deklaration überein?
  - Stimmen die Einheiten (Meter, Zentimeter, …)?
  - Bei Übergabe von Objekten: gibt es Seiteneffekte? Werden diese beim Aufrufer bedacht?
- Kommentare
  - Hat jede Quelldatei, jede Klasse und jede Methode, jede Konstante einen Kommentar?
  - Wird das Verhalten (gegeben, gesucht, prinzipieller Ansatz) im Kommentar beschrieben?
  - Passen die Kommentare (noch) zum Quelltext?
  - Sind die Kommentare hilfreich für das Verstehen des Quelltextes?
  - Sind die richtigen Kommentare im javadoc-Kommentar für die htmlSeiten und die richtigen Kommentare intern?
- Umfang von Quelldateien und Methoden
  - Sind die Quelldateien kürzer als (z.B.) 2.000 Zeilen?
  - Sind Methoden kürzer als (z.B.) 60 Zeilen?
  - Hinweis: konkrete Zahlenangaben sind projekt-/problemabhängig
- Modularität
  - Hängen die Klassen eines Pakets inhaltlich eng zusammen?
  - Gibt es replizierten Code, den man entfernen könnte?
  - Gibt es mehrere Stellen im Code, an denen dasselbe getan wird, so dass diese Funktionalität zu einer Methode zusammengefasst werden könnte?
  - Wurde aus Unkenntnis der Standard-Bibliothek Funktionalität selbst implementiert?
- Speichereffizienz
  - Sind Arrays (Reihungen) übertrieben groß deklariert?
  - Werden Referenzen auf nicht mehr benötigte Objekte auf null gesetzt, damit der Speicherbereiniger (Garbage Collector) den Platz frei geben kann?
- Leistungsdefekte
  - Können effizientere Datenstrukturen und/oder Algorithmen benutzt werden?
  - Sind logische Tests im Code so angeordnet, dass die oft erfolgreicheren und billigeren Tests vor den teureren und weniger häufigen Tests durchgeführt werden?
  - Können die Kosten für die Wiederberechnung eines Wertes reduziert werden, indem die Berechnung nur einmal ausgeführt und der Wert zwischengespeichert wird?
  - Wird jedes berechnete Ergebnis auch tatsächlich benötigt?