```
    public static void main(String[] args) {
        try {
            kill();
        } catch (RuntimeException r) {
            System.out.println("RunTimeException "+ r +"\n");
        } catch (Killed b) {
            System.out.println("Killed It!");
            System.out.println(b);
            System.out.println(b.getMessage());
        }
    } // end of main
}     // end of class Kill
```

- Ein selbstdefinierter Fehler sollte als Unterklasse von `Exception` deklariert werden !
- Die Klasse `Exception` verfügt über die Konstruktoren `public Exception(); public Exception(String str);` (`str` ist die evt. auszugebende Fehlermeldung).
- `throw` exc löst den Fehler exc aus – sofern sich der Ausdruck exc zu einem Objekt einer Unterklasse von `Throwable` auswertet.
- Weil `Killed` keine Unterklasse von `RuntimeException` ist, wird die geworfene Exception erst von der zweiten `catch`-Regel gefangen :-)
- Ausgabe:

```
Killed It!
Killed
Null
```

Fazit:

- Fehler in Java sind Objekte und können vom Programm selbst behandelt werden.
- `try ... catch ... finally` gestattet, die Fehlerbehandlung deutlich von der normalen Programmausführung zu trennen.
- Die vordefinierten Fehlerarten reichen oft aus.
- Werden spezielle neue Fehler/Ausnahmen benötigt, können diese in einer Vererbungshierarchie organisiert werden.

Warnung:

- Der Fehler-Mechanismus von Java sollte auch nur zur Fehler-Behandlung eingesetzt werden:
  - Installieren eines Handlers ist billig; fangen einer `Exception` dagegen teuer.
  - Ein normaler Programm-Ablauf kann durch eingesetzte `Exceptions` bis zur Undurchsichtigkeit verschleiert werden.
  - Was passiert, wenn `catch`- und `finally`-Regeln selbst wieder Fehler werfen?
- Fehler sollten dort behandelt werden, wo sie auftreten :-)
- Es ist besser spezifischere Fehler zu fangen als allgemeine – z.B. mit `catch (Exception e) { }`

19 Programmierfehler und ihre Behebung

(kleiner lebenspraktischer Ratgeber)

Grundsätze:

- Jeder Mensch macht Fehler :-)
- ... insbesondere beim Programmieren.
- Läuft ein Programm, sitzt der Fehler tiefer.
- Programmierfehler sind Denkfehler.
- Um eigene Programmierfehler zu entdecken, muss nicht ein Knoten im Programm, sondern ein Knoten im Hirn gelöst werden.

19.1 Häufige Fehler und ihre Ursachen

- Das Programm terminiert nicht.

  Mögliche Gründe:

  - In einer Schleife wird die Schleifen-Variable nicht modifiziert.

```
...
String t = file.readLine();
while(t! = null)
   System.out.println(t);
...
```

- In einer Rekursion fehlt die Abbruch-Bedingung.

```
public static int find0(int[] a, int x, int l, inr r) {
   int t = (l+r)/2;
   if (x<=a[t]) return find0(a,x,l,t);
   return find0(a,x,t+1,r);
   }
```

- Das Programm wirft eine `NullPointerException`.

  Möglicher Grund:

  Eine Objekt-Variable wird benutzt, ohne initialisiert zu sein:

  - ... weil sie in einem Feld liegt:

```
Stack[] h = new Stack[4];
...
for(int i=0; i<4; ++i)
   h[i].push(i);
....
```

- ... oder einem Objekt ohne passenden Konstruktor:

```
import java.io.*;
class A {
   public A a;
   }
class AA {
   public static void main(String[] args) {
      A aa = (new A()).a;
      System.out.println(aa);
      System.out.println(aa.a);
   }
}
```

- Eine Instanz-Variable verändert auf geheimnisvolle Weise ihren Wert.

  Möglicher Grund:

  Es gibt weitere Verweise auf das Objekt mit (unerwünschten?) Seiteneffekten ...

```
...
List l1 = new List(3);
List l2 = l1;
l2.info = 7;
...
```

- Ein Funktionsaufruf hat überhaupt keinen Effekt ...

```
public static void reverse (String [] a) {
    int n = a.length();
    String [] b = new String [n];
    for (int i=0; i<n; ++i) b[i] = a[n-i-1];
    a = b;
}
```

- Ein Funktionsaufruf hat überhaupt keinen Effekt ...

```
public static void reverse (String [] a) {
    int n = a.length();
    String [] b = new String [n];
    for (int i=0; i<n; ++i) b[i] = a[n-i-1];
    a = b;
}
```

- Eine bedingte Verzweigung liefert merkwürdige Ergebnisse.

  Mögliche Gründe:

  - `equals()` mit == verwechselt?
  - Die `else`-Teile falsch organisiert?

19.2 Generelles Vorgehen zum Testen von Software

(1) Feststellen fehlerhaften Verhaltens.

Problem: Auswahl einer geeigneter Test-Scenarios

Black-Box Testing: Klassifiziere Benutzungen!

Finde Repräsentanten für jede (wichtige) Klasse!

White-Box Testing: Klassifiziere Berechnungen – z.B. nach

- besuchten Programm-Punkten,
- benutzten Datenstrukturen oder Klassen
- benutzten Methoden, geworfenen Fehler-Objekten ...

Finde repräsentantive Eingabe für jede (wichtige) Klasse!

Beispiel: `int find(int[] a, int x);`

Black-Box Test: Klassifizierung denkbarer Argumente:

1. `a == null;`
2. `a != null :`

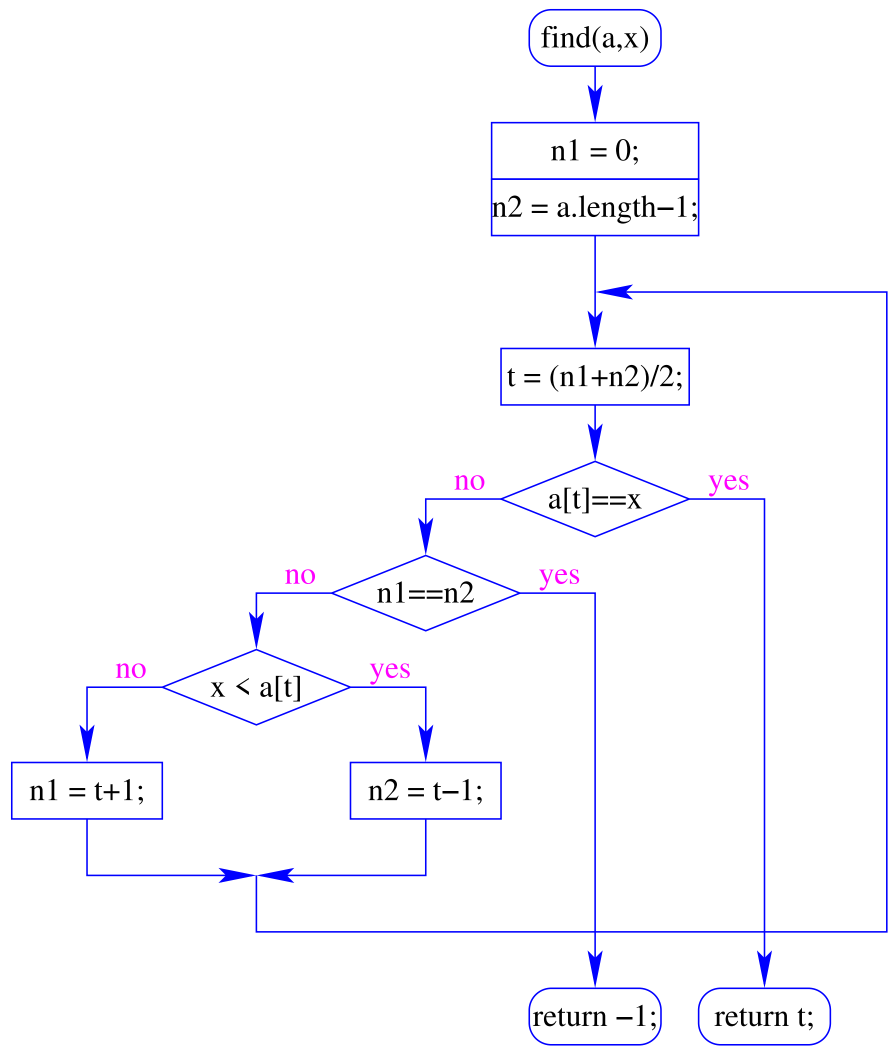
2.1. `x` kommt in `a` vor $\Longrightarrow$ `a == [42], x == 42`

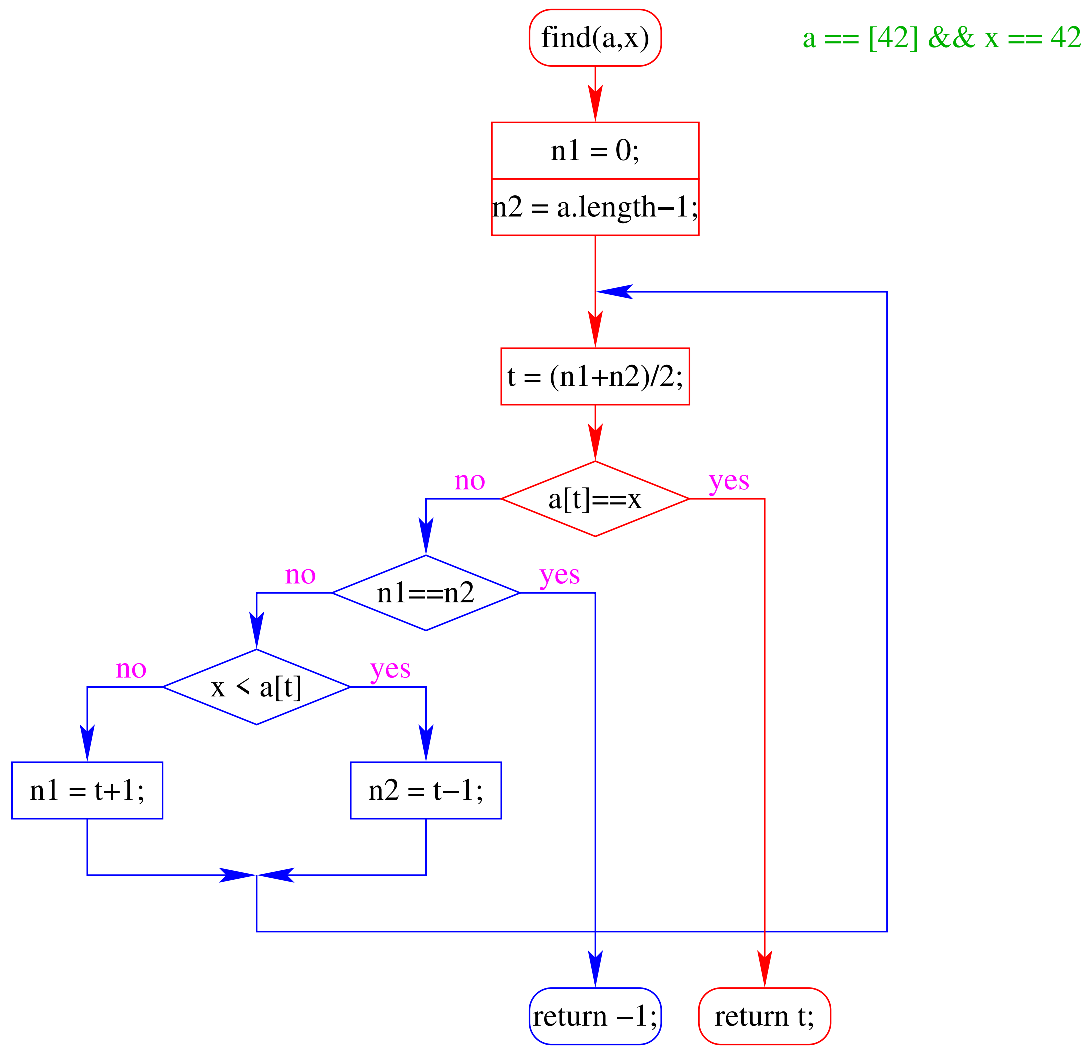
2.2. `x` kommt nicht in `a` vor $\Longrightarrow$ `a == [42], x == 7`

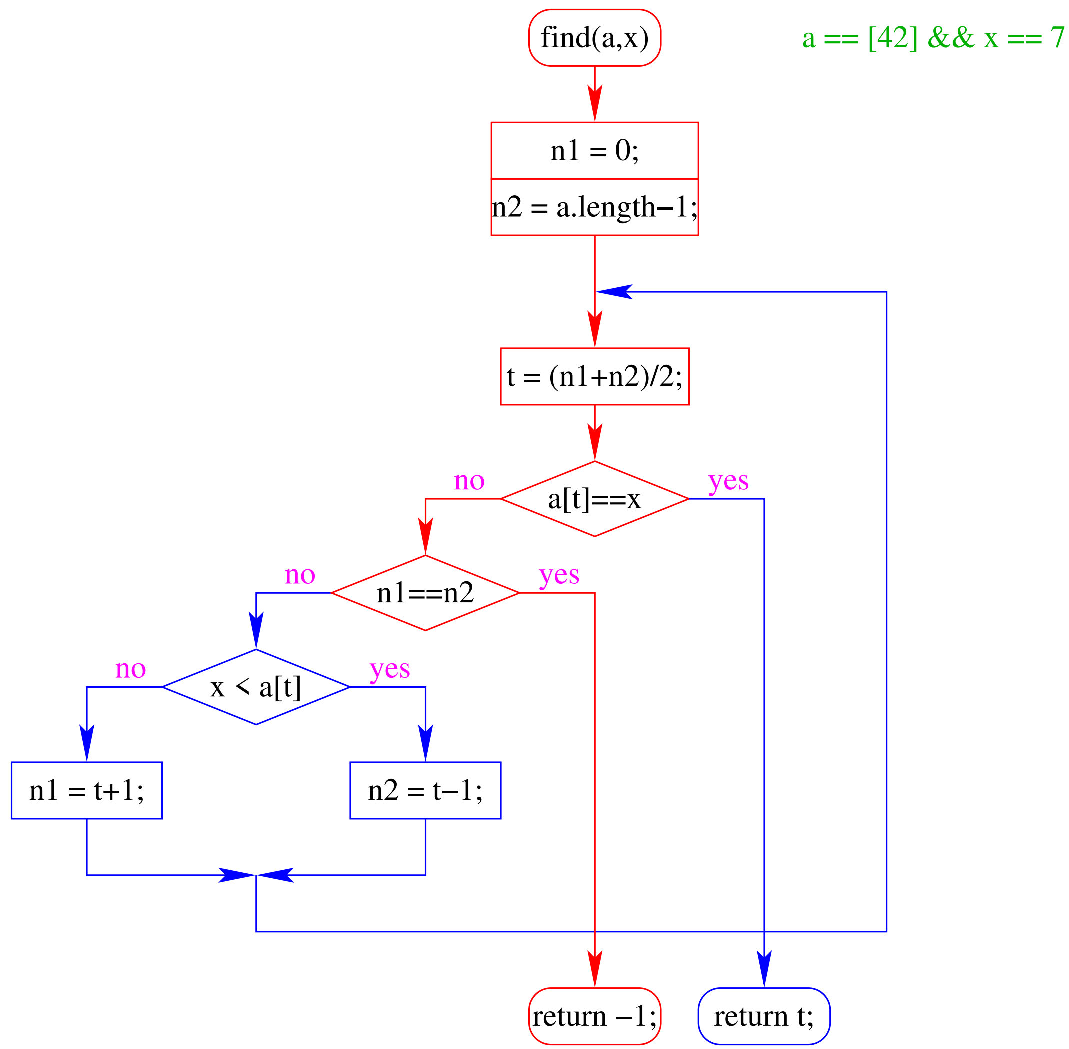
Achtung:

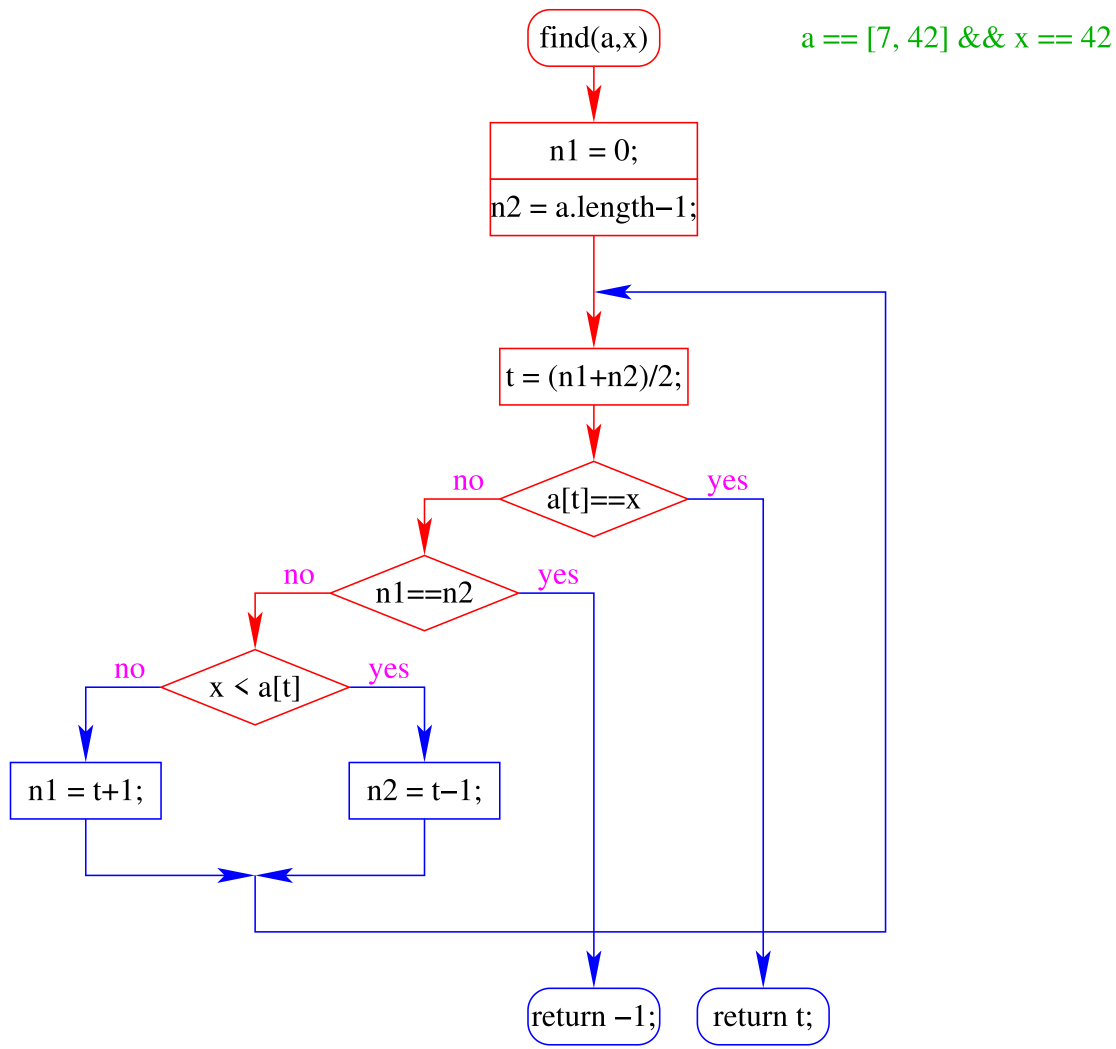
Nicht in allen Klassen liefert `find()` sinnvolle Ergebnisse ...
$\Longrightarrow$ Überprüfe, ob alle Benutzungen in sinnvolle Klassen fallen :-)

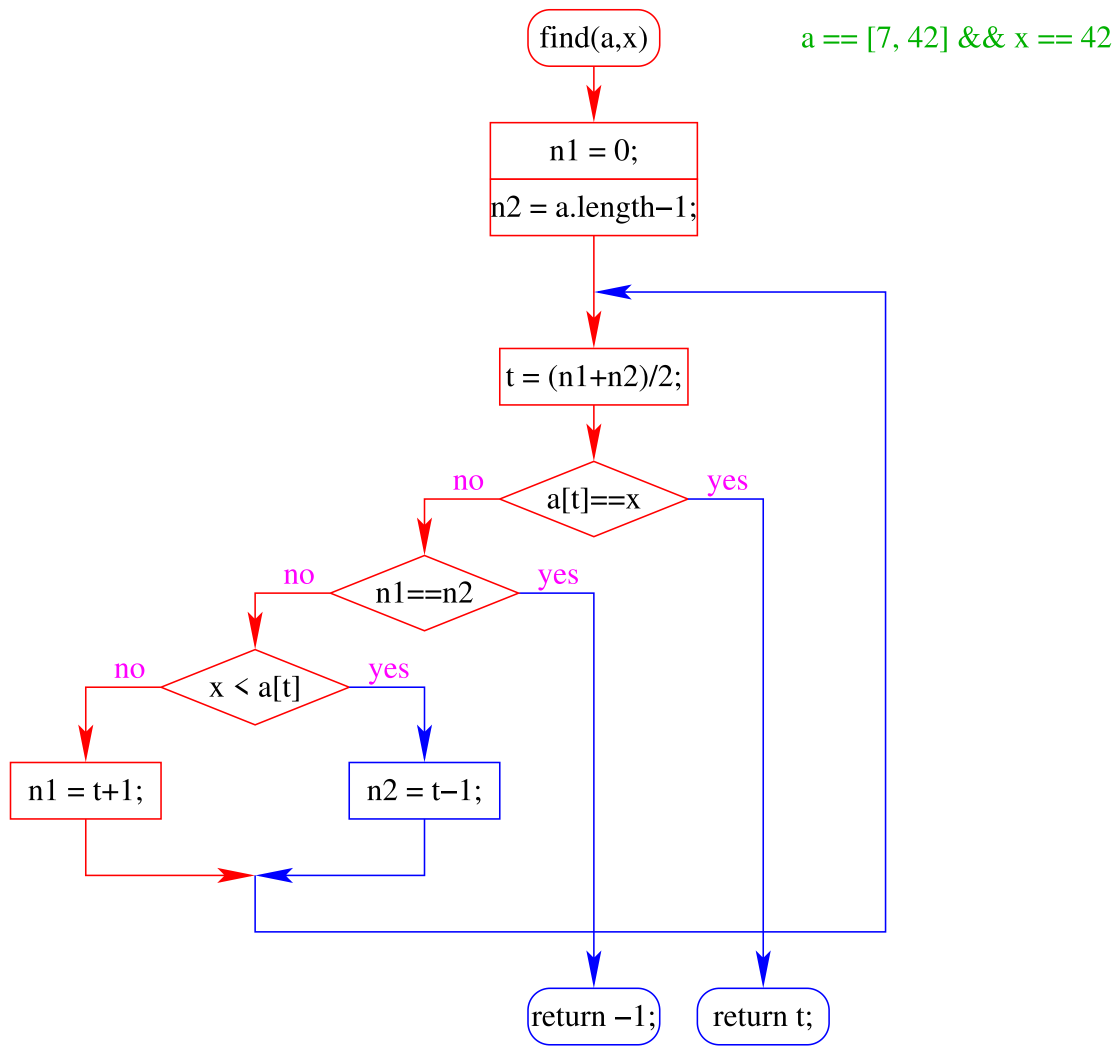
White-Box Test: Klassifizierung von Berechnungen:

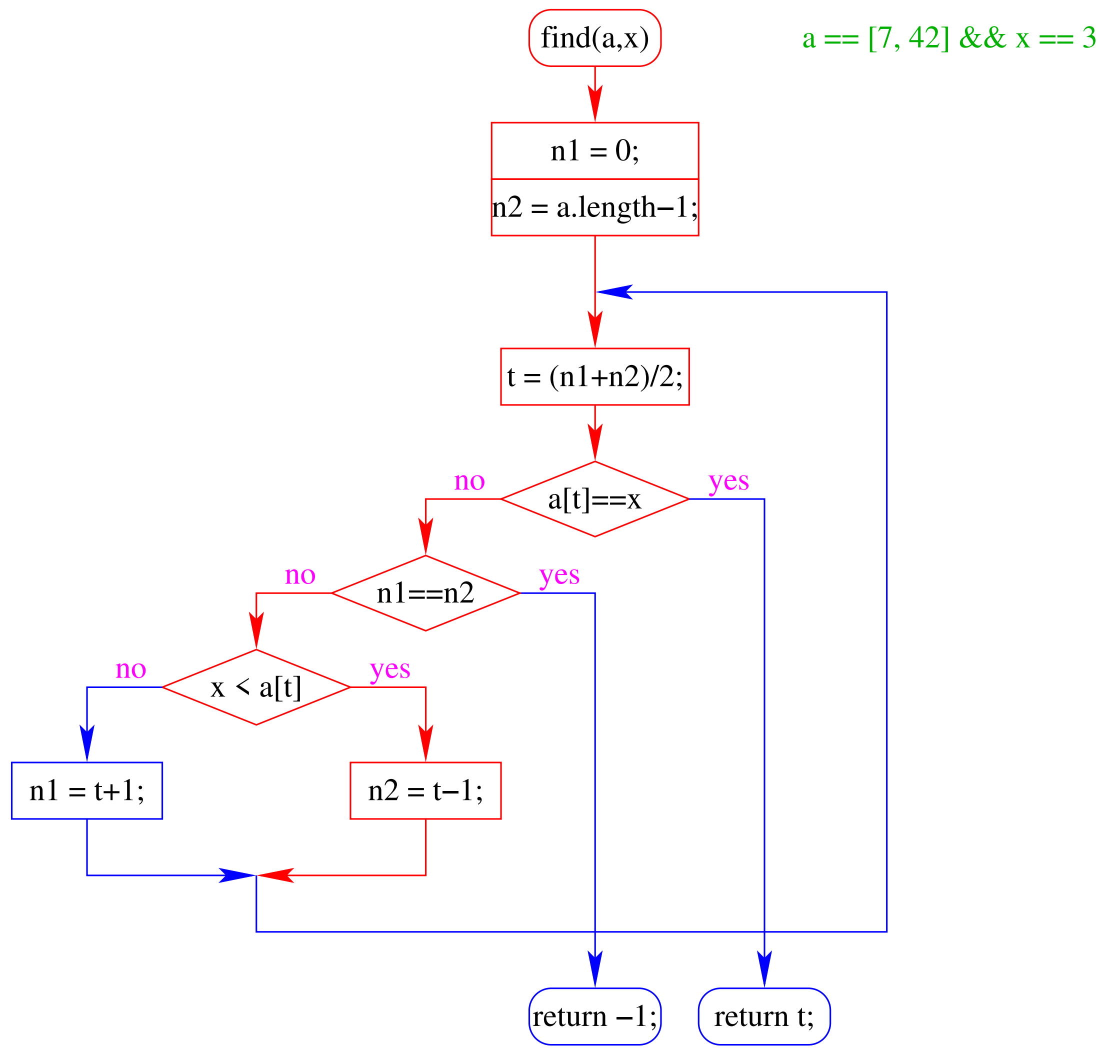

find(a,x)
n1 = 0;
n2 = a.length−1;
t = (n1+n2)/2;
a[t]==x
no
yes
n1==n2
no
yes
x < a[t]
no
yes
n1 = t+1;
n2 = t−1;
return −1;
return t;


find(a,x)
a == [42] && x == 42
n1 = 0;
n2 = a.length−1;
t = (n1+n2)/2;
a[t]==x
no
yes
n1==n2
no
yes
x < a[t]
no
yes
n1 = t+1;
n2 = t−1;
return −1;
return t;


find(a,x)
a == [42] && x == 7
n1 = 0;
n2 = a.length−1;
t = (n1+n2)/2;
no
a[t]==x
yes
no
n1==n2
yes
no
x < a[t]
yes
n1 = t+1;
n2 = t−1;
return −1;
return t;


find(a,x)
a == [7, 42] && x == 42
n1 = 0;
n2 = a.length−1;
t = (n1+n2)/2;
no
a[t]==x
yes
no
n1==n2
yes
no
x < a[t]
yes
n1 = t+1;
n2 = t−1;
return −1;
return t;


find(a,x)
a == [7, 42] && x == 42
n1 = 0;
n2 = a.length−1;
t = (n1+n2)/2;
no
a[t]==x
yes
no
n1==n2
yes
no
x < a[t]
yes
n1 = t+1;
n2 = t−1;
return −1;
return t;


find(a,x)
a == [7, 42] && x == 3
n1 = 0;
n2 = a.length−1;
t = (n1+n2)/2;
no
a[t]==x
yes
no
n1==n2
yes
no
x < a[t]
yes
n1 = t+1;
n2 = t−1;
return −1;
return t;

- Eine Menge von Test-Eingaben überdeckt ein Programm, sofern bei ihrer Ausführung sämtliche interessanten Stellen (hier: Programm-Punkte) mindestens einmal besucht werden.
- Die Funktion `find()` wird überdeckt von:

```
a == [42]; x == 42;
a == [42]; x == 7;
a == [7, 42]; x == 42;
a == [7, 42]; x == 3;
```

- Eine Menge von Test-Eingaben überdeckt ein Programm, sofern bei ihrer Ausführung sämtliche interessanten Stellen (hier: Programm-Punkte) mindestens einmal besucht werden.
- Die Funktion `find()` wird überdeckt von:

```
a == [42]; x == 42;
a == [42]; x == 7;
a == [7, 42]; x == 42;
a == [7, 42]; x == 3;
```

Achtung:

- Konstruktion einer überdeckenden Test-Menge ist schwer ...
- Ein Test für jeden Programm-Punkt ist i.a. nicht genug :-(
- Auch intensives Testen findet i.a. nicht sämtliche Fehler :-((

(2) Eingrenzen des Fehlers im Programm.

- Leicht, falls der Fehler eine nicht abgefangene `exception` auslöste :-)
- Schwer, falls das Programm stumm in eine Endlos-Schleife gerät ... $\Longrightarrow$ Einfügen von Test-Ausgaben, Breakpoints.

(2) Eingrenzen des Fehlers im Programm.

- Leicht, falls der Fehler eine nicht abgefangene `exception` auslöste :-)
- Schwer, falls das Programm stumm in eine Endlos-Schleife gerät ... $\Longrightarrow$ Einfügen von Test-Ausgaben, Breakpoints.

(3) Lokalisieren des Fehlers.

- Leicht, falls der Fehler innerhalb einer Programm-Einheit auftritt.
- Schwer, wenn er aus Missverständnissen zwischen kommunizierenden Teilen (die jede für sich korrekt sind) besteht ... $\Longrightarrow$ Aufstellen von Anforderungen, Abprüfen der Erfüllung der Anforderungen

(4) Verstehen des Fehlers.

Problem: Lösen des Knotens im eigenen Hirn. Oft hilft:

- Das Problem einer anderen Person schildern ...
- Eine Nacht darüber schlafen ...

(4) Verstehen des Fehlers.

Problem: Lösen des Knotens im eigenen Hirn. Oft hilft:

- Das Problem einer anderen Person schildern ...
- Eine Nacht darüber schlafen ...

(5) Beheben des Fehlers.

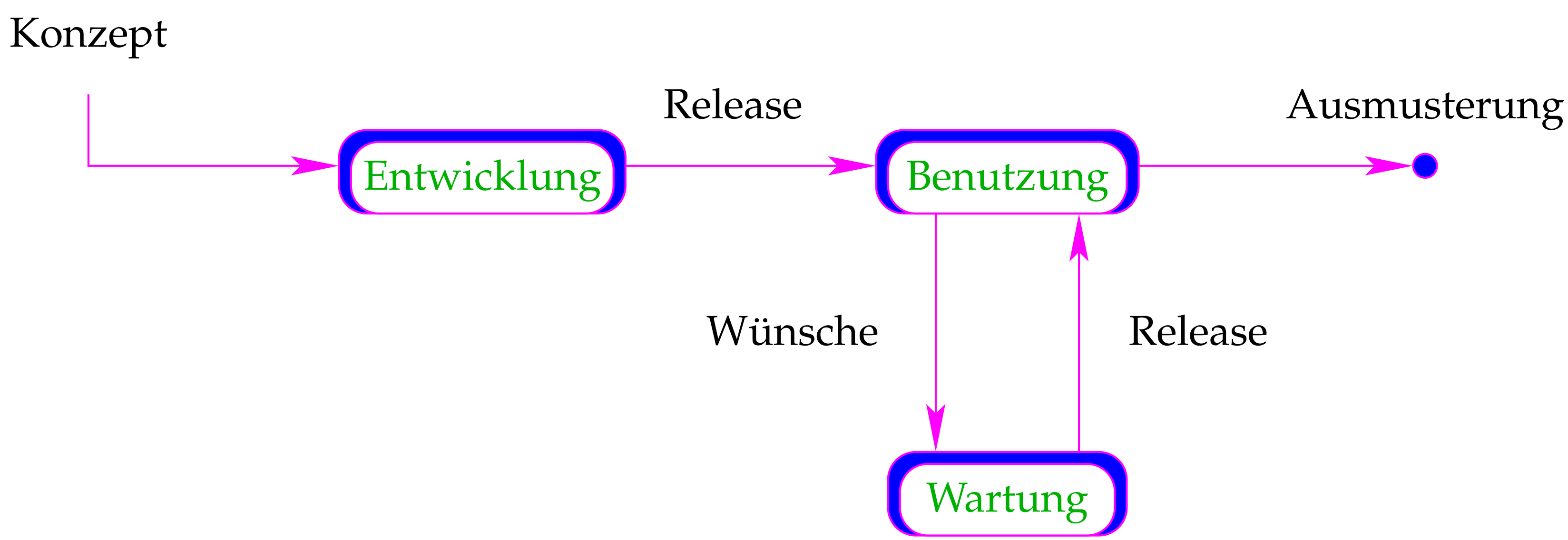
... das geringste Problem :-)

20 Programmieren im Großen

Neu:

- Das Programm ist groß.
- Das Programm ist unübersichtlich.
- Das Programm ist teuer.
- Das Programm wird lange benutzt.

Software-Zyklus:

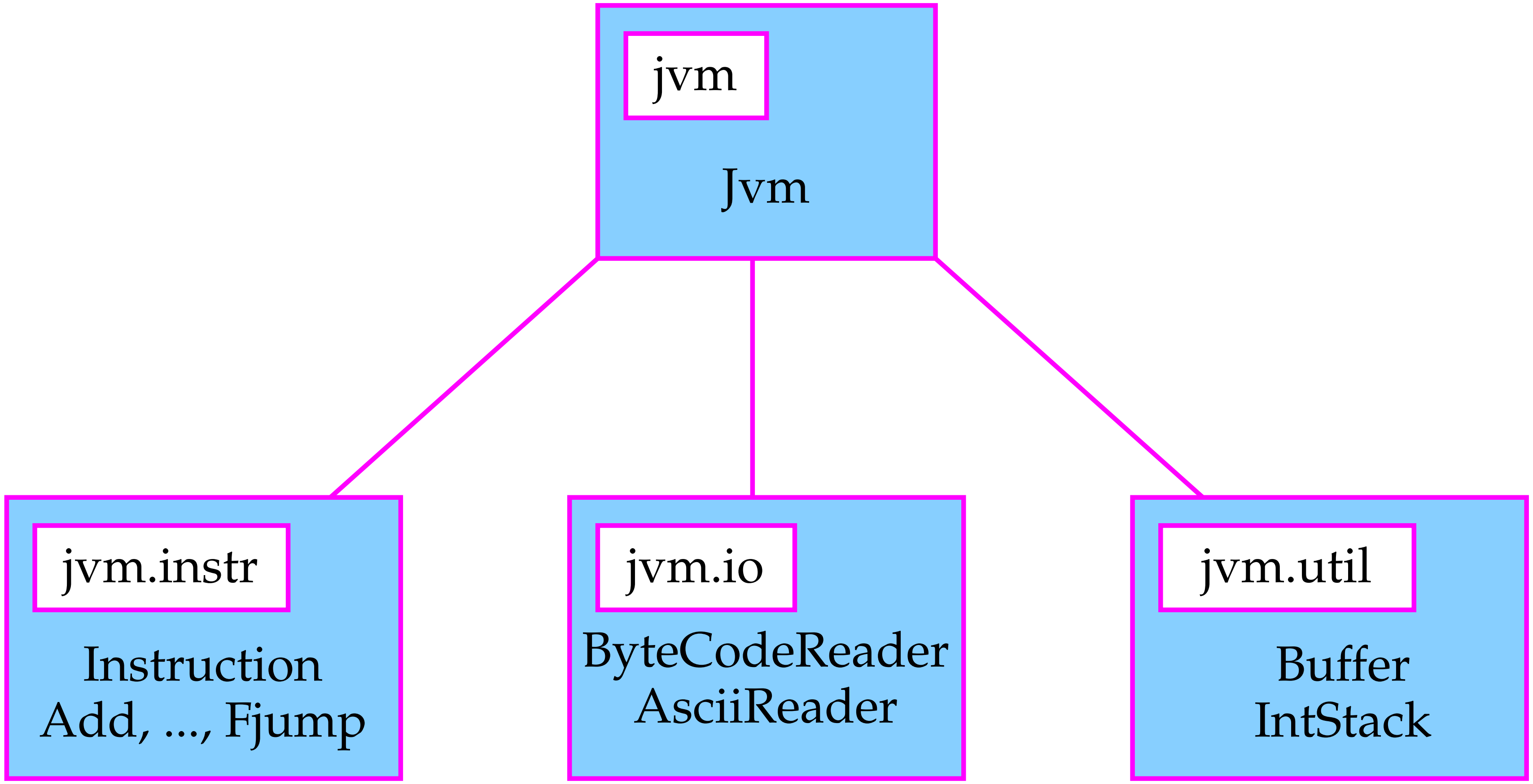


- Wünsche können sein:
  - Beseitigen von Fehlern;
  - Erweiterung der Funktionalität;
  - Portierung auf eine andere Plattform;
  - ...
- Die Leute, die die Wartung vornehmen, sind i.a. verschieden von denen, die das System implementierten.
- Gute Wartbarkeit ergibt sich aus
  - einem klaren Design;
  - einer übersichtlichen Strukturierung $\Longrightarrow$ packages;
  - einer sinnvollen, verständlichen Dokumentation.

Programm-Pakete in Java

- Ein großes System sollte hierarchisch in Teilsysteme zerlegt werden.
- Jedes Teilsystem bildet ein Paket oder `package` ...
- und liegt in einem eigenen Verzeichnis.

Beispiel: Unsere JVM


jvm
Jvm
jvm.instr
Instruction
Add, ..., Fjump
jvm.io
ByteCodeReader
AsciiReader
jvm.util
Buffer
IntStack

- Für jede Klasse muss man angeben:
  1. zu welchem Paket sie gehört;
  2. welche Pakete bzw. welche Klassen aus welchen Paketen sie verwendet.

Im Verzeichnis `a` liege die Datei `A.java` mit dem Inhalt:

```
package a;
import a.d.*;
import a.b.c.C;
class A {
   public static void main(String[] args) {
      C c = new C();
      D d = new D();
      System.out.println(c+ "   "+d);
   }
}
```

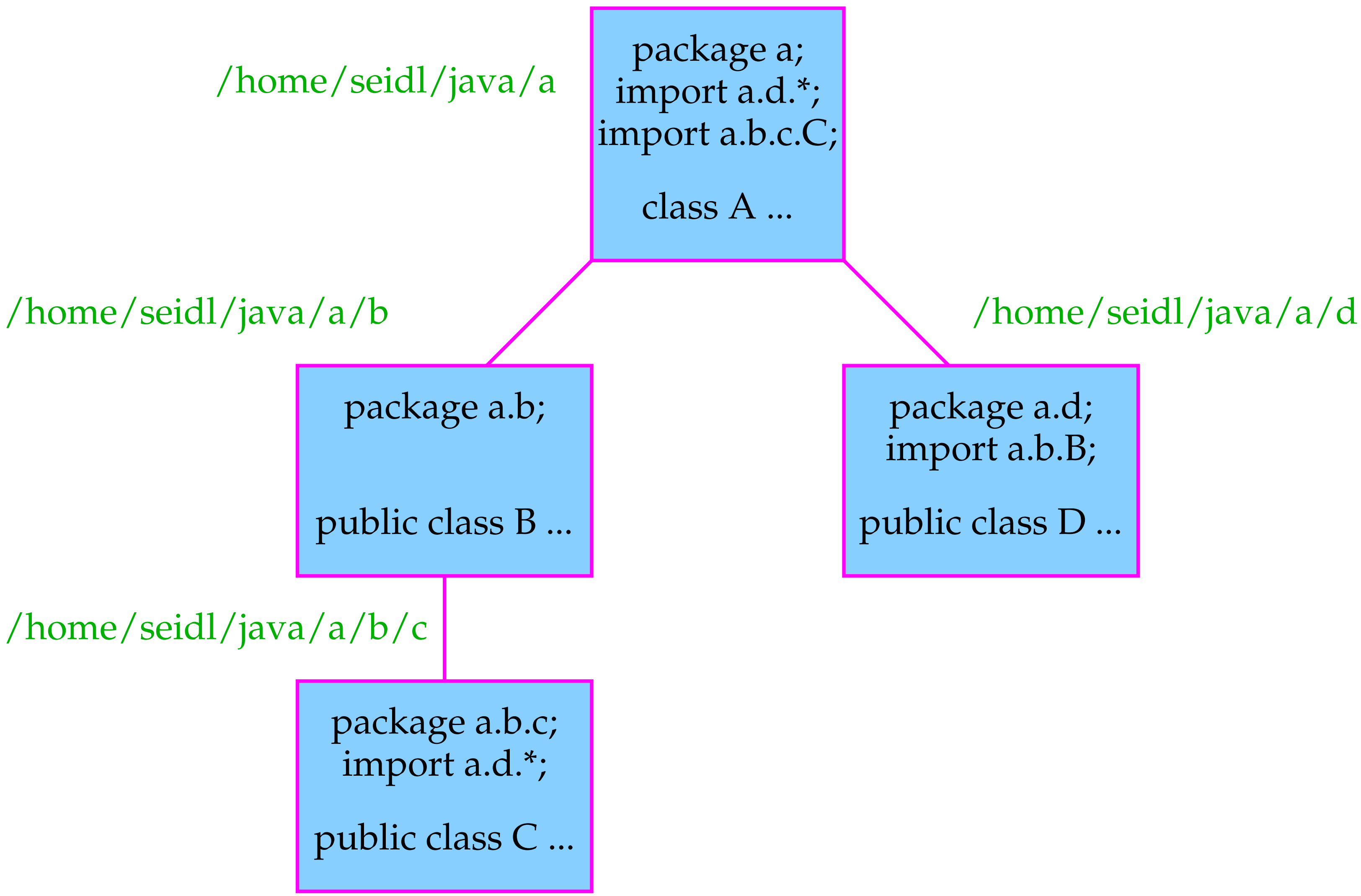
- Jede Datei mit Klassen des Pakets pckg muss am Anfang gekennzeichnet sein mit der Zeile `package pckg;`
- Die Direktive `import pckg.*;` stellt sämtliche öffentlichen Klassen des Pakets pckg den Klassen in der aktuellen Datei zur Verfügung – nicht dagegen die Unterverzeichnisse :-| .
- Die Direktive `import pckg.Cls;` stellt dagegen nur die Klasse Cls des Pakets pckg (d.h. genauer die Klasse pckg.Cls) zur Verfügung.

In den Unterverzeichnissen `b`, `b/c` und `d` von `a` liegen Dateien mit den Inhalten:

```java
package a.b;
public class B { }
class Ghost { }

package a.b.c;
import a.d.*;
public class C { }
```

```java
package a.d;
import a.b.B;
public class D {
   private B b = null;
}
```


/home/seidl/java/a
package a;
import a.d.*;
import a.b.c.C;
class A ...
/home/seidl/java/a/b
package a.b;
public class B ...
/home/seidl/java/a/d
package a.d;
import a.b.B;
public class D ...
/home/seidl/java/a/b/c
package a.b.c;
import a.d.*;
public class C ...