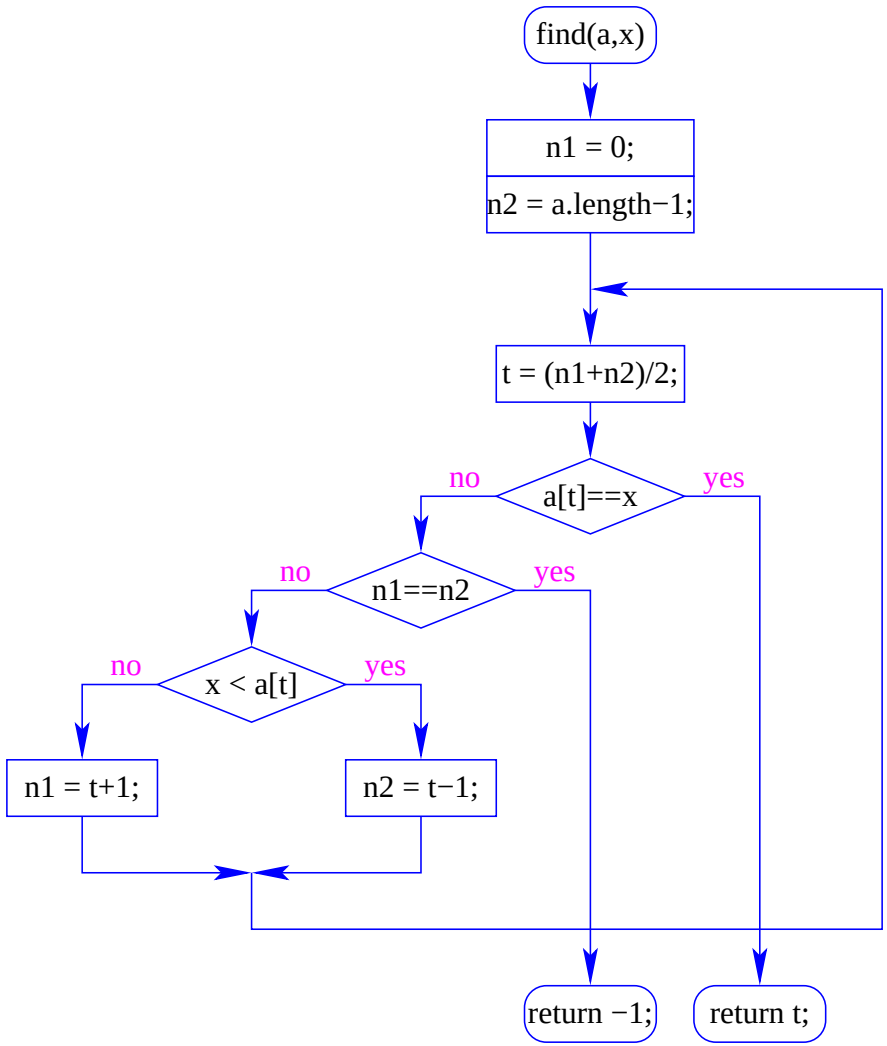
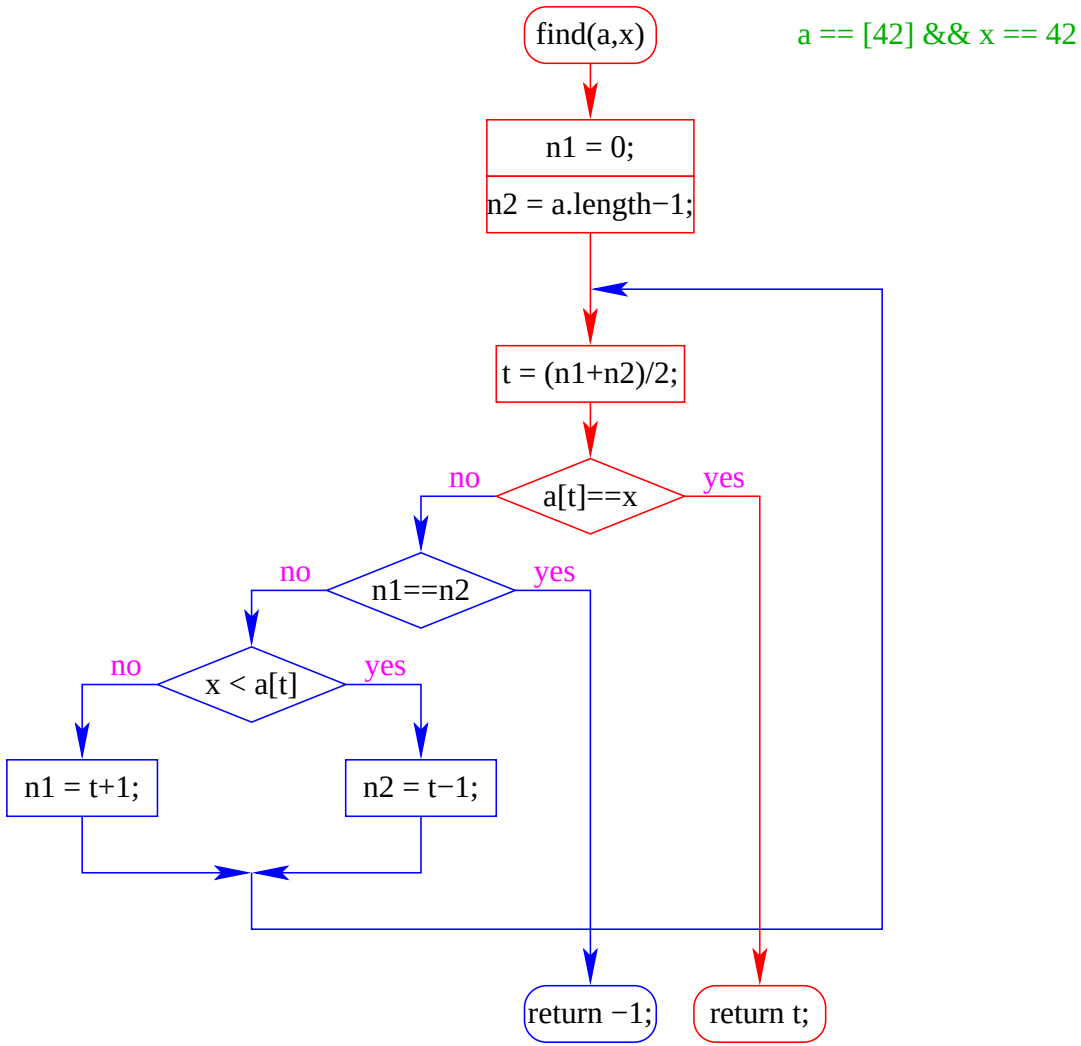
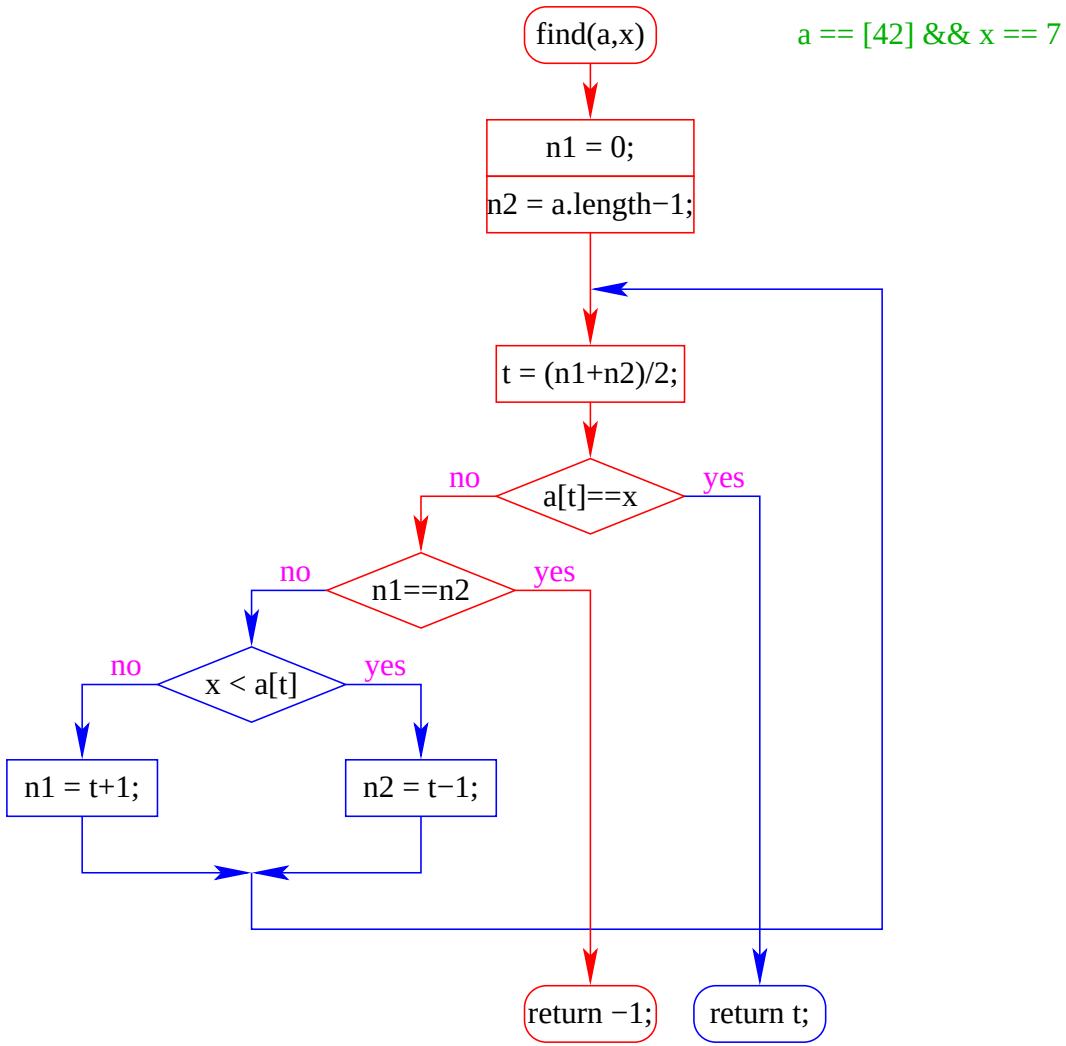
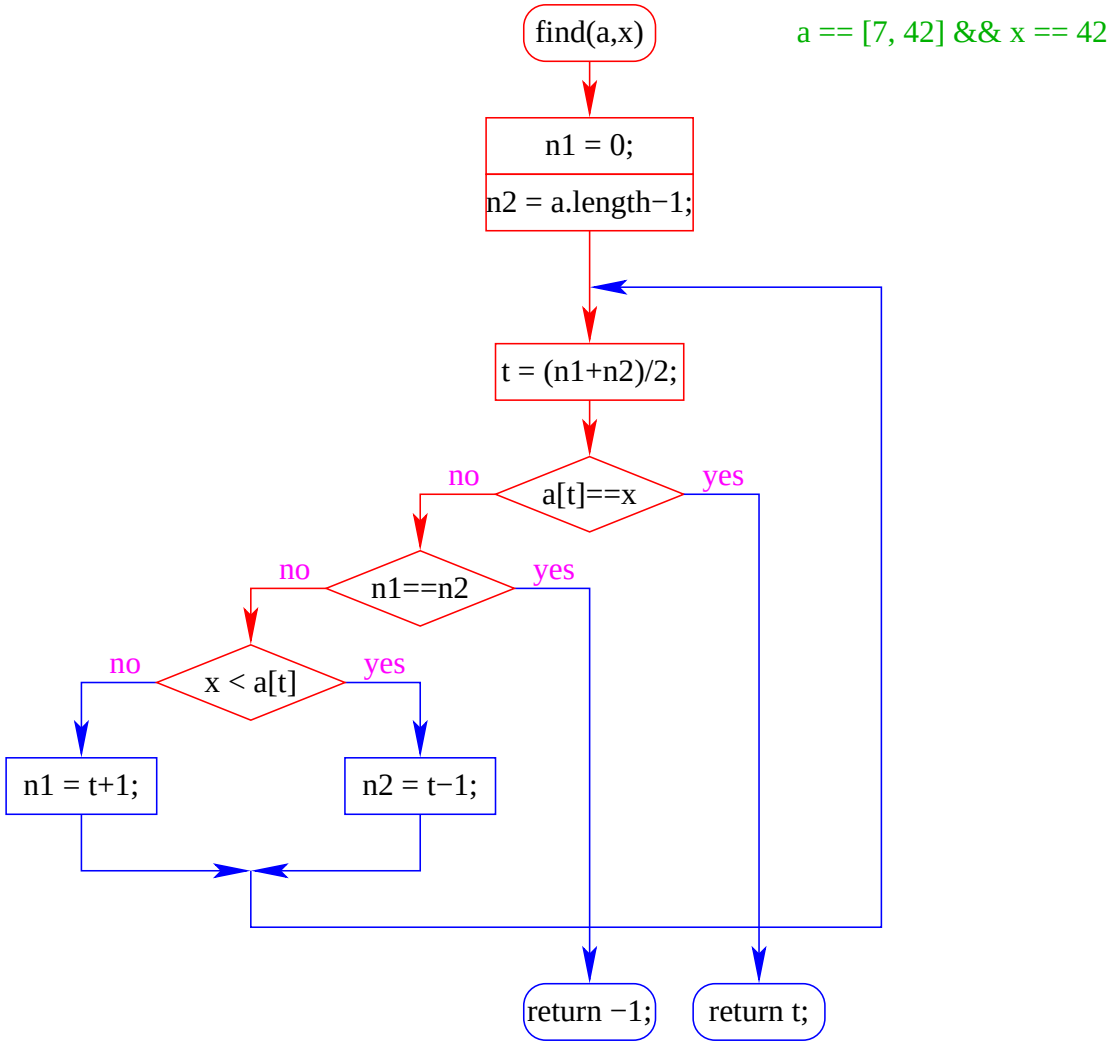
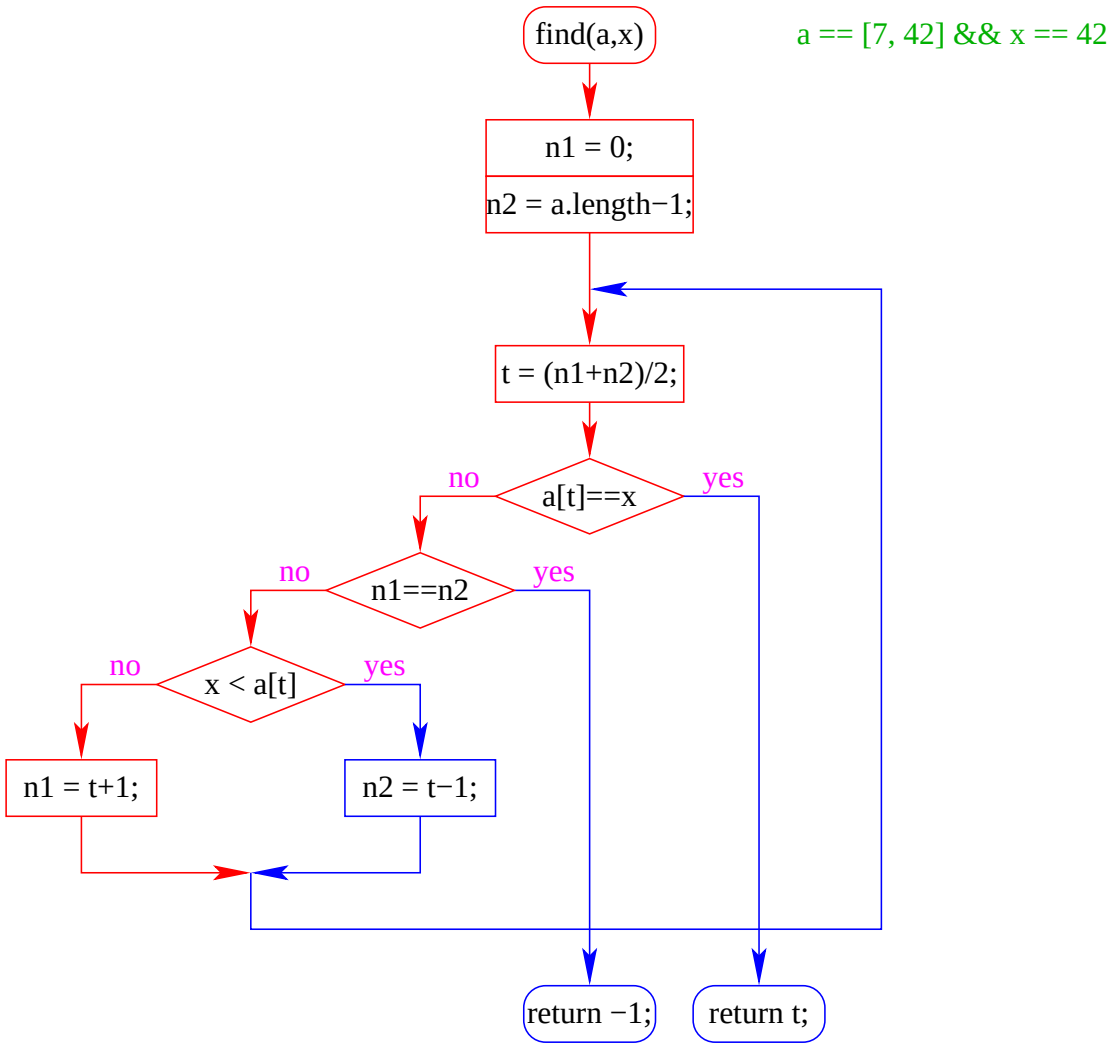
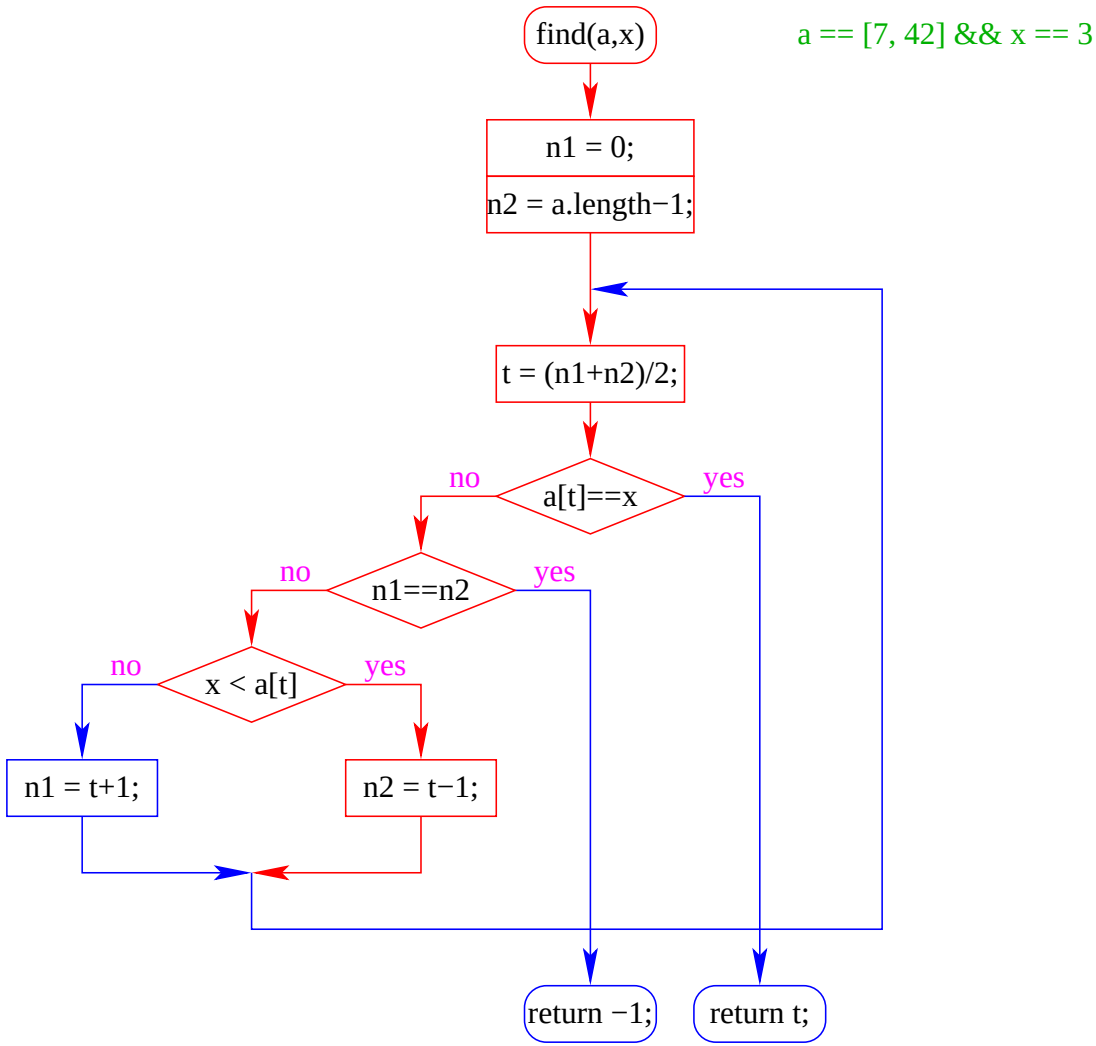

find(a,x)
n1 = 0;
n2 = a.length−1;
t = (n1+n2)/2;
a[t]==x
no
yes
n1==n2
no
yes
x < a[t]
no
yes
n1 = t+1;
n2 = t−1;
return −1;
return t;


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;
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 == [7, 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 == [7, 42] && x == 3
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;

- 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.

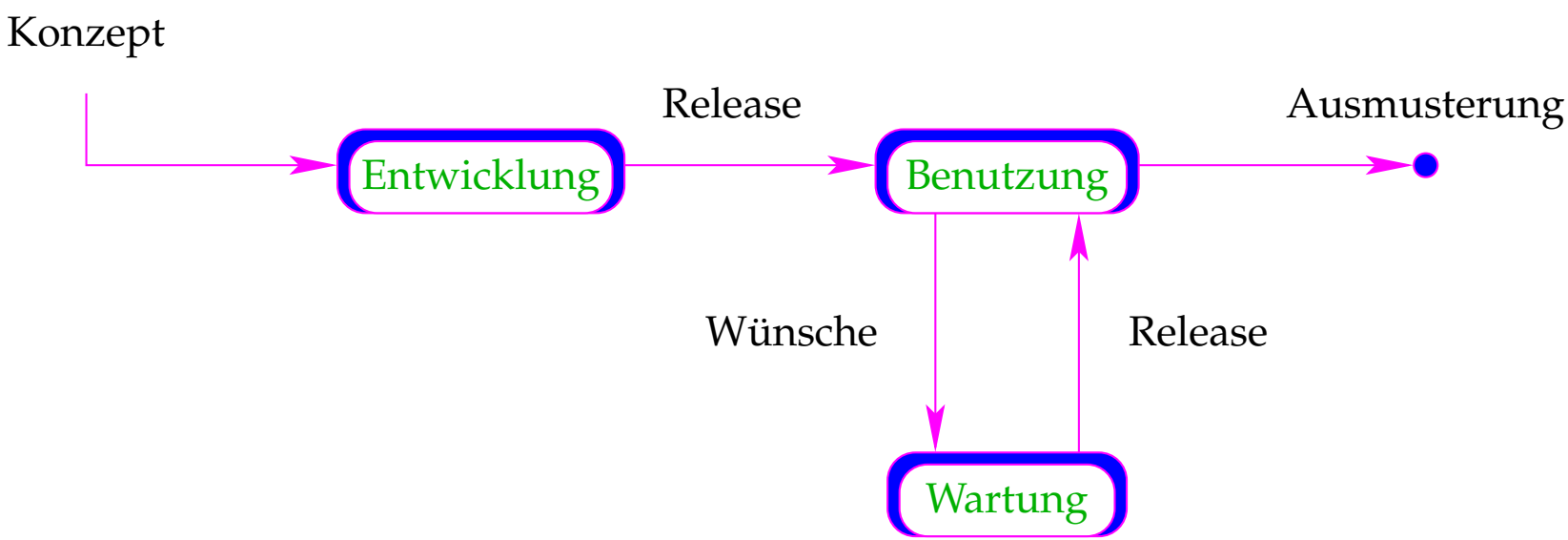
... das geringste Problem :-)

19 Programmieren im Großen

Neu:

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

Software-Zyklus:

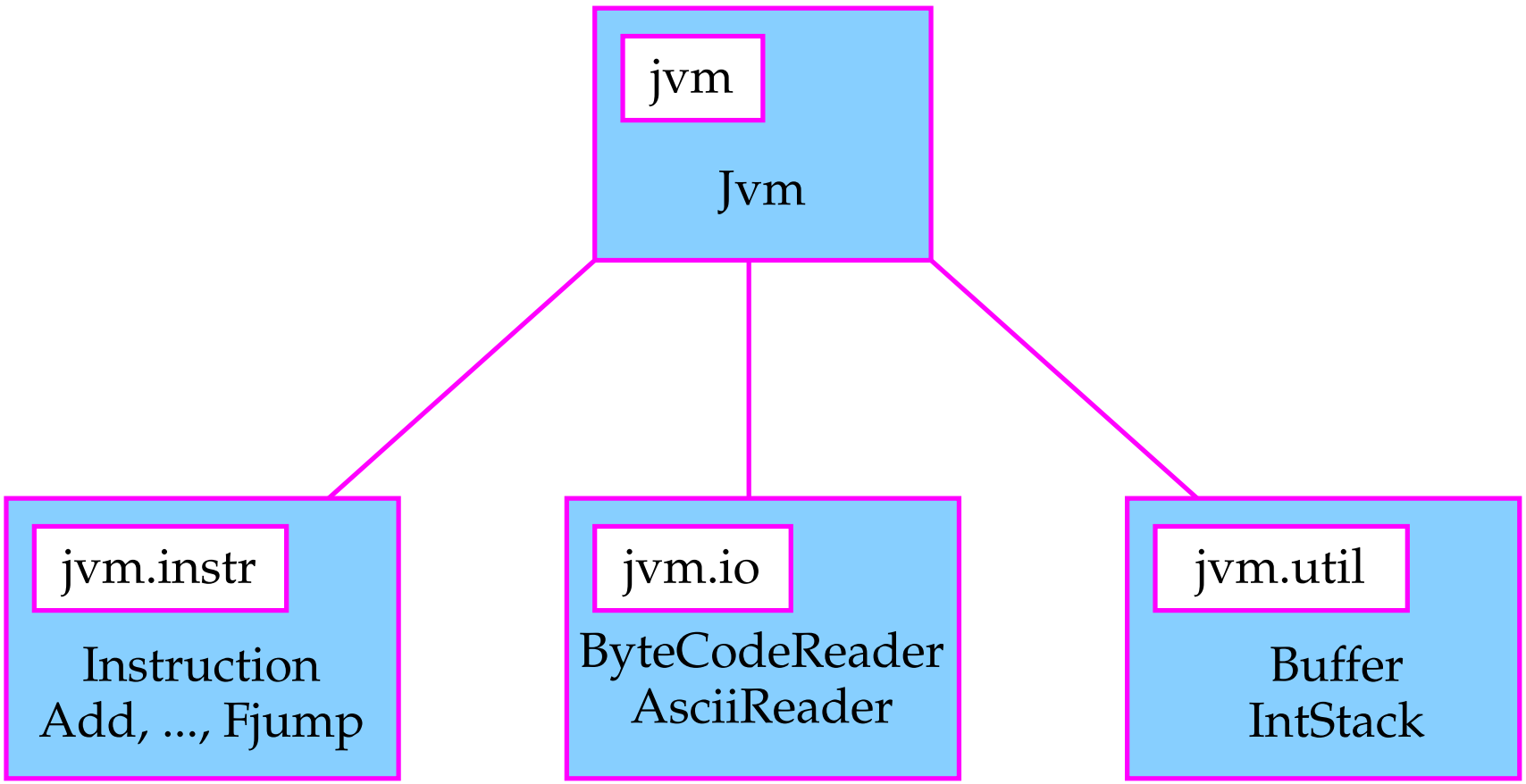


- Wünsche können sein:
  - Beseitigen von Fehlern;
  - Erweiterung der Funktionalität;
  - Portierung auf eine andere Plattform;
  - ...
- Die Leute, die die Wartung vornehmen, sind i.a. verschieden von denen, die das System implementierten.
- Gute Wartbarkeit ergibt sich aus
  - einem klaren Design;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

19.1 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);
   }
}
```

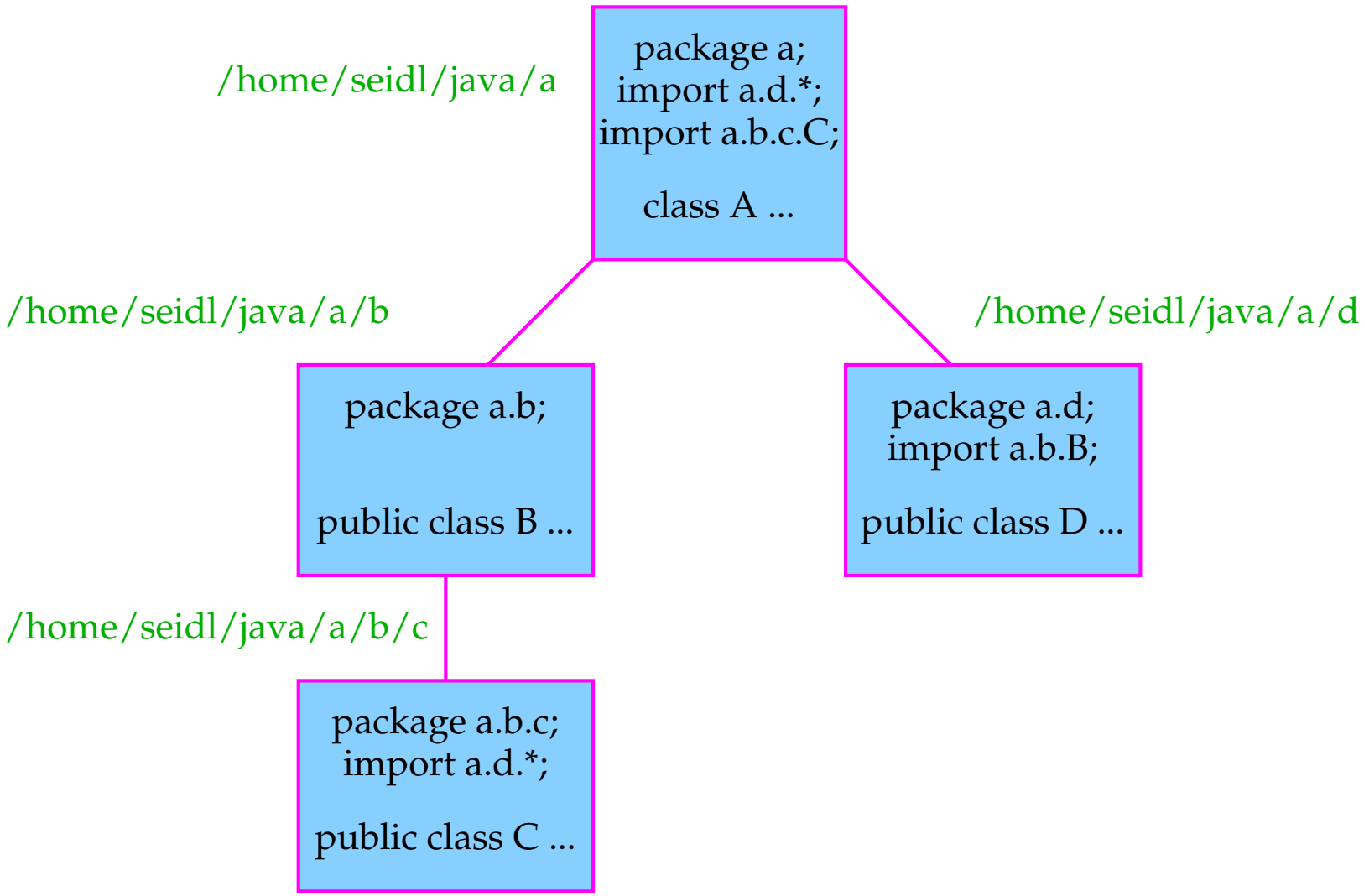
- Jede Datei mit Klassen des Pakets pckg muss am Anfang gekennzeichnet sein mit der Zeile `package` pckg`;`
- Die Direktive `import` pckg`.*;` stellt sämtliche öffentlichen Klassen des Pakets pckg den Klassen in der aktuellen Datei zur Verfügung – nicht dagegen die Unterverzeichnisse :-| .
- Die Direktive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

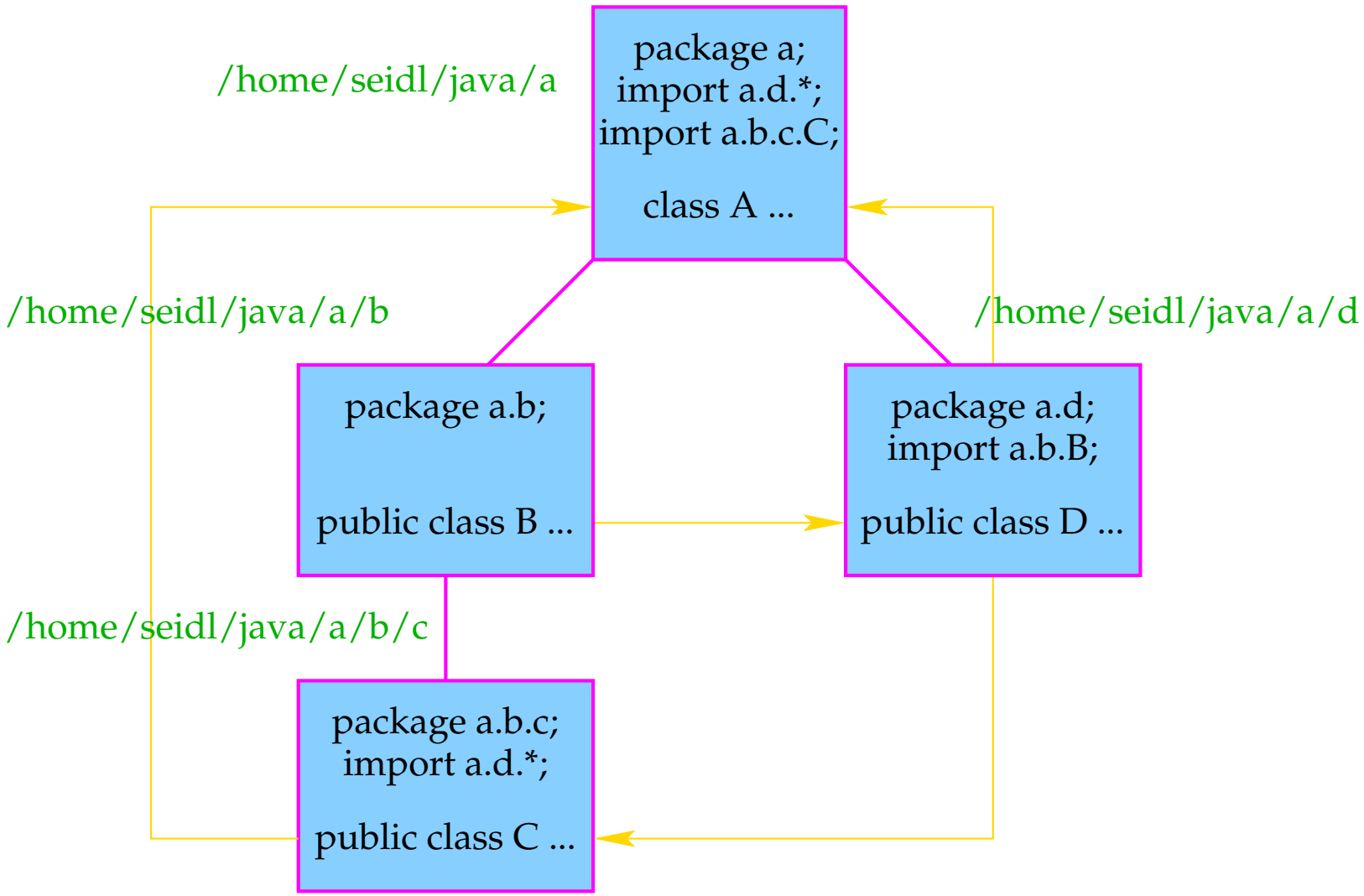
In den Unterverzeichnissen `b`, `b/c` und `d` von `a` liegen Dateien mit den Inhalten:

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

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

```
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
/home/seidl/java/a/d
package a.b;
public class B ...
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 ...


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

Achtung:

- Jede Klasse eines Pakets, die in einer Klasse außerhalb des Pakets benutzt werden soll, muss als `public` gekennzeichnet werden.
- Jede Datei darf zwar mehrere Klassen-Definitionen enthalten, aber nur eine einzige, die `public` ist.
- Der Name der öffentlichen Klasse muss mit demjenigen der Datei übereinstimmen ... :-)
- Der Paket-Name enthält den gesamten absoluten Zugriffs-Pfad von dem Wurzel-Paket.
- Abhängigkeiten zwischen Paketen können zirkulär sein.

Im Verzeichnis a lässt sich das Programm compilieren. Allerdings liefert ...

```
> java A
Exception in thread "main" java.lang.NoClassDefFoundError: a/A (wrong name
        at java.lang.ClassLoader.defineClass0(Native Method)
        at java.lang.ClassLoader.defineClass(Compiled Code)
        at java.security.SecureClassLoader.defineClass(Compiled Code)
        at java.net.URLClassLoader.defineClass(Compiled Code)
        at java.net.URLClassLoader.access$1(Compiled Code)
        at java.net.URLClassLoader$1.run(Compiled Code)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(Compiled Code)
        at java.lang.ClassLoader.loadClass(Compiled Code)
        at sun.misc.Launcher$AppClassLoader.loadClass(Compiled Code)
        at java.lang.ClassLoader.loadClass(Compiled Code)
```

Aufruf von `java a.A` ist schon besser:

```
> java a.A
Exception in thread "main" java.lang.NoClassDefFoundError: a/A
```

Aufruf von `java a.A` ein Verzeichnis oberhalb von `a` liefert dagegen:

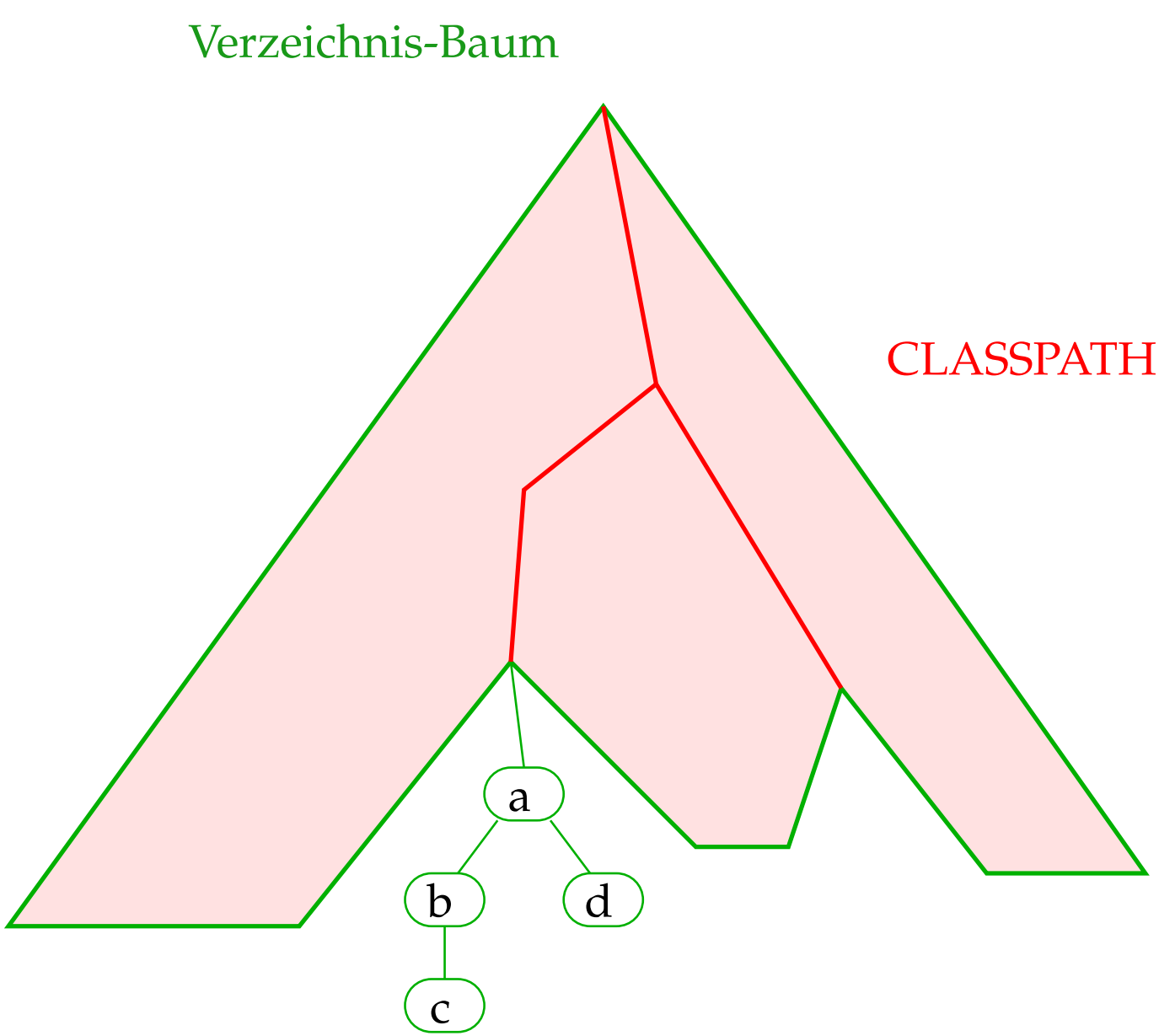
```
> java a.A
a.b.c.C@67bb4c68  a.d.D@69df4c68
```

Der Grund:

- Damit Java Verzeichnisse mit Paketen findet, sollte die Umgebungsvariable `CLASSPATH` gesetzt werden, z.B. hier mithilfe des Kommandos:

  ```
  setenv CLASSPATH ~/java:.
  ```

- Diese Variable enthält die Start-Verzeichnisse, in denen bei einem Aufruf nach Klassen oder Paketen gesucht wird.
- Bei einem Aufruf `> java A` durchsucht das Laufzeit-System sämtliche in `CLASSPATH` angegebenen Verzeichnisse nach einer Datei `A.class` und führt diese aus (– sofern sie vorhanden ist).
- Bei einem Aufruf `> java a.b.c.A` sucht das Laufzeit-System eine Datei `A.class` in Unterverzeichnissen `a/b/c` von Verzeichnissen aus `CLASSPATH`.
- Voreingestellt ist das aktuelle Verzeichnis, d.h.: `CLASSPATH=.`


Verzeichnis-Baum
CLASSPATH
a
b
d
c