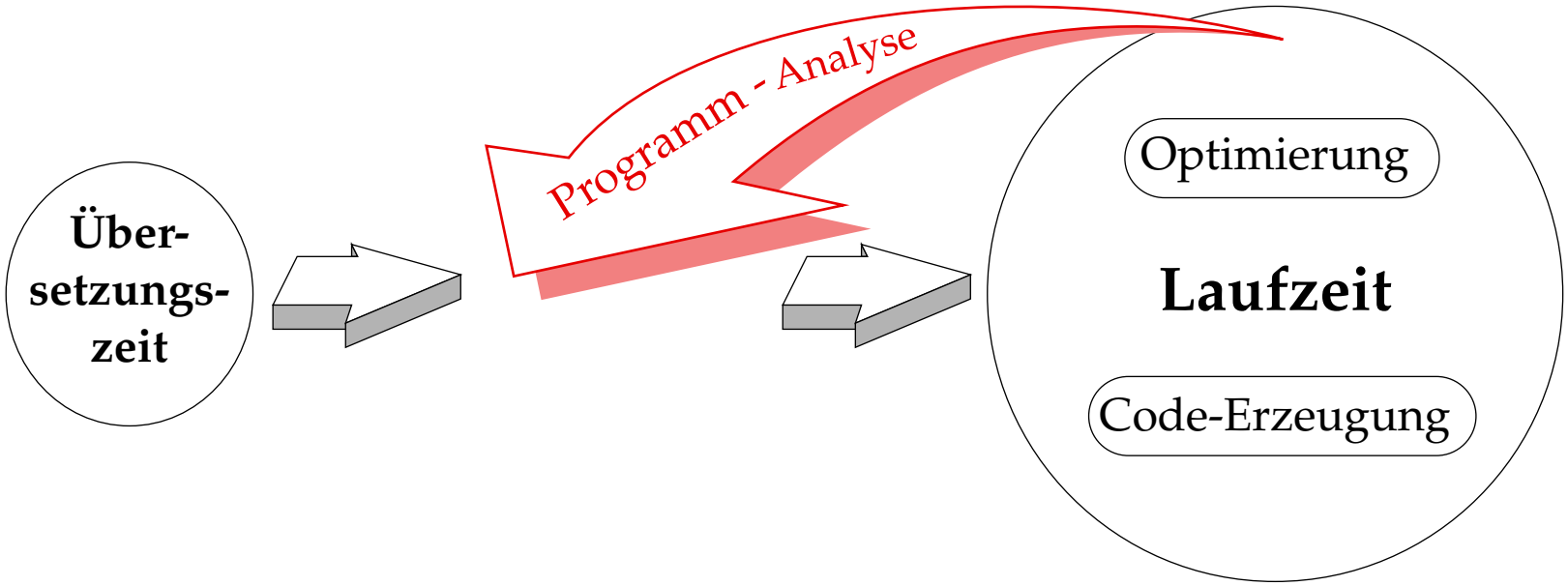
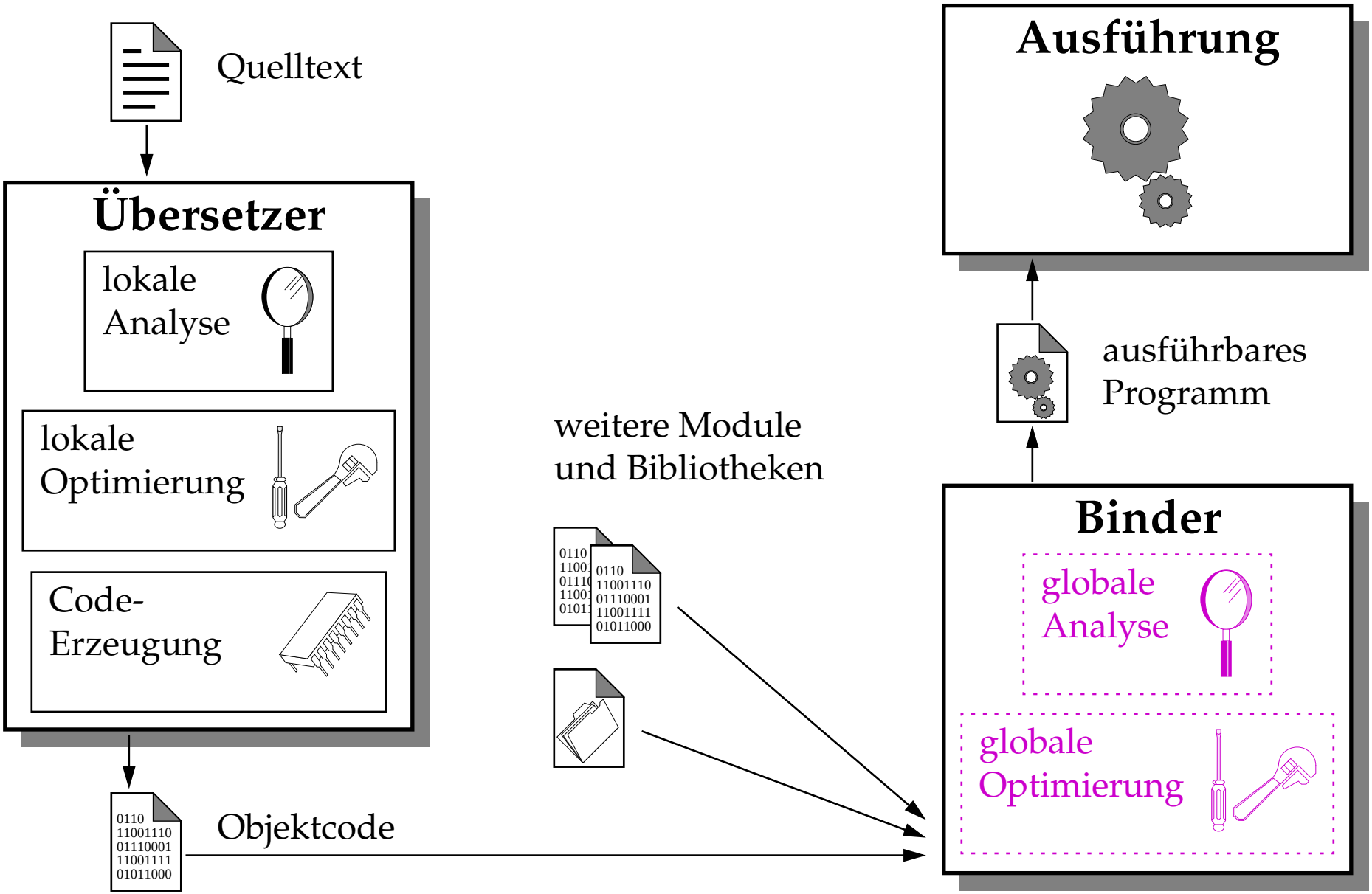
Statische Analyse von Bibliotheken als Grundlage dynamischer Optimierung

Michael Thies und Uwe Kastens, November 1998

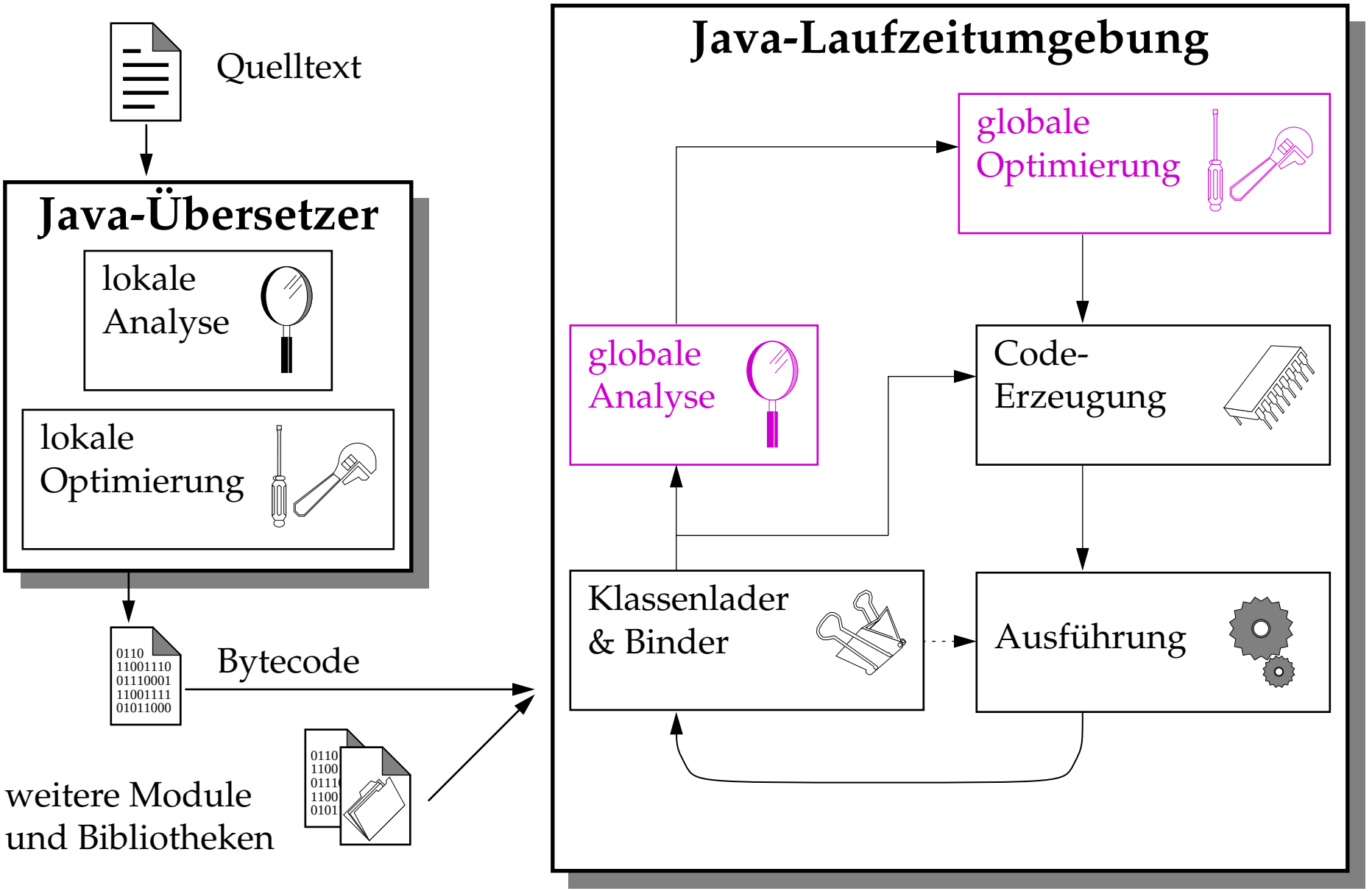
Universität-Gesamthochschule Paderborn
Fachbereich Informatik
Arbeitsgruppe Programmiersprachen und Übersetzer



“Klassische” Programmiersprachen



Java ist anders

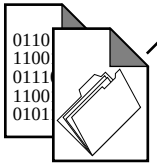


Statische Analyse vor der Laufzeit

Java-Bytecode-erzeugender Übersetzer

0110
11001110
01110001
11001111
01011000

Klassendatei



weitere Module und Bibliotheken, i.d.R. **nicht** das vollständige Programm

Globale Analyse & Optimierung

Optimierung auf Bytecode-Ebene

— schleifeninvarianter Code

— gemeinsame Teilausdrücke

Analyse

— Datenflußanalyse
- MOD/USE-Mengen

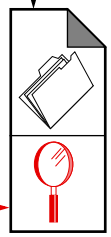
— konkrete Typinferenz

— Schleifenerkennung
- Durchlaufzahl statisch abschätzen

— überflüssige Laufzeittests
- Indexbereiche, Typumwandlungen

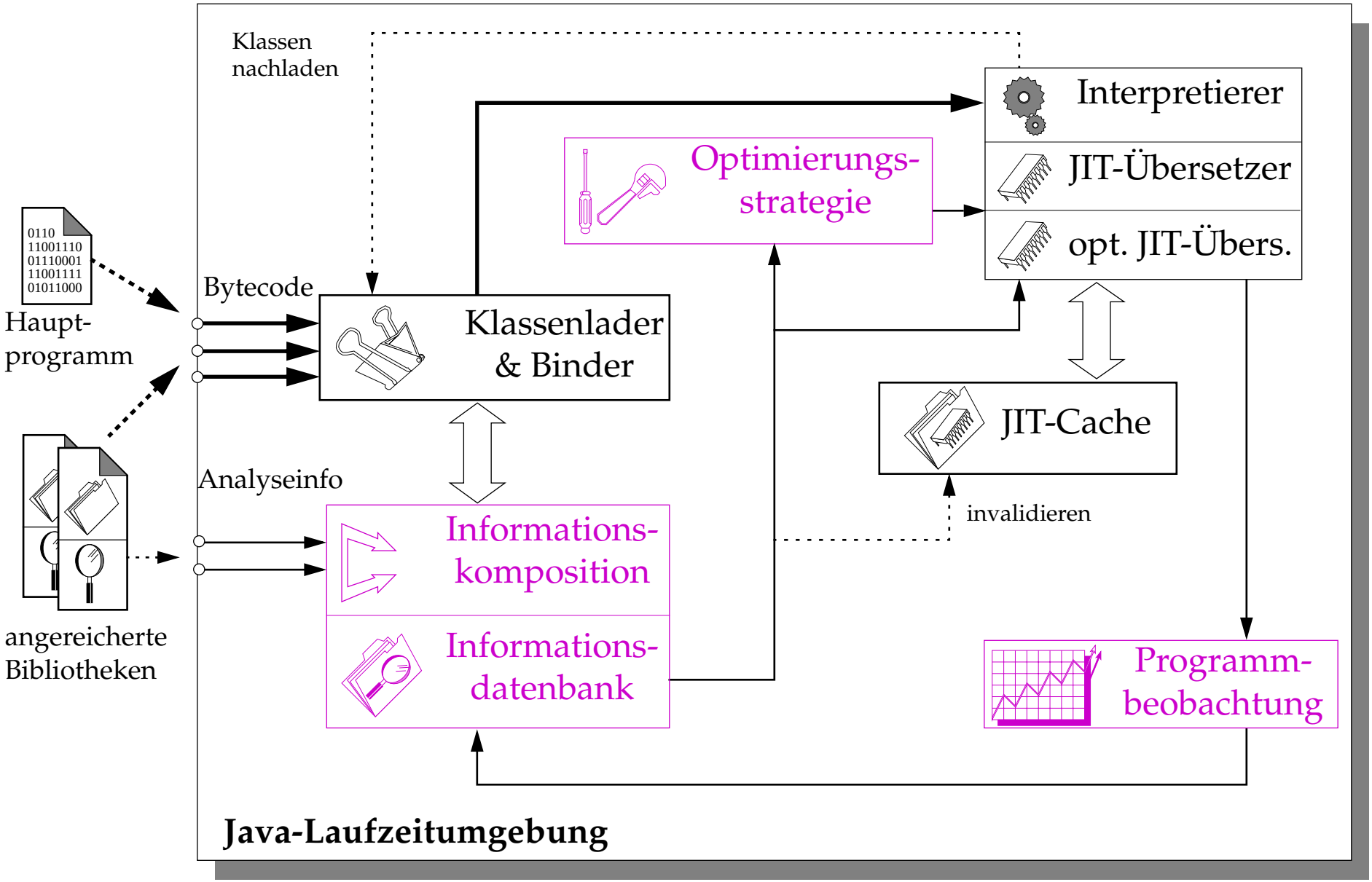
— Aliasanalyse

Filter



mit Analyse-information angereicherte Bibliothek

Modifizierte Laufzeitumgebung



Probleme beim Austausch von Klassen

Übersetzungs-/Analysezeit

Laufzeit

```
class A {
public static void p()
 { int x = B.q();
   int y = B.q();
 }
}
```

A.java

```
class B {
public static int q()
 {
   return 1001;
 }
}
```

B.java

Java-Übersetzer

Statische Analyse

0110
11001110
01110001
11001111
01011000

A.class

B.q() hat keine Seiteneffekte

Binärkompatibilität **nicht** hinreichend für Gültigkeit der Analyseinformation!

Java Virtuelle Maschine

B.q() einmal in A.p() aufrufen

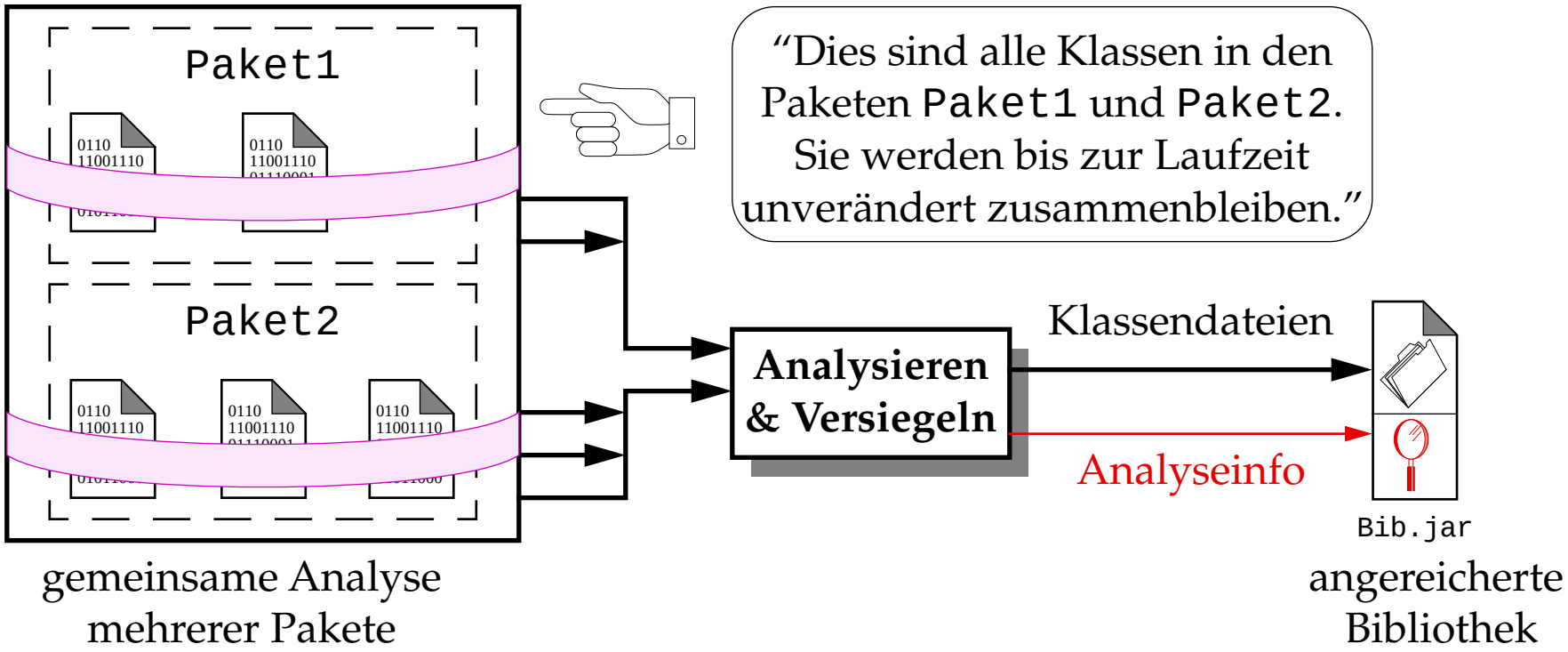
0110
11001110
01110001
11001111
01011000

B.class

Implementierung von B.q() mit Seiteneffekten

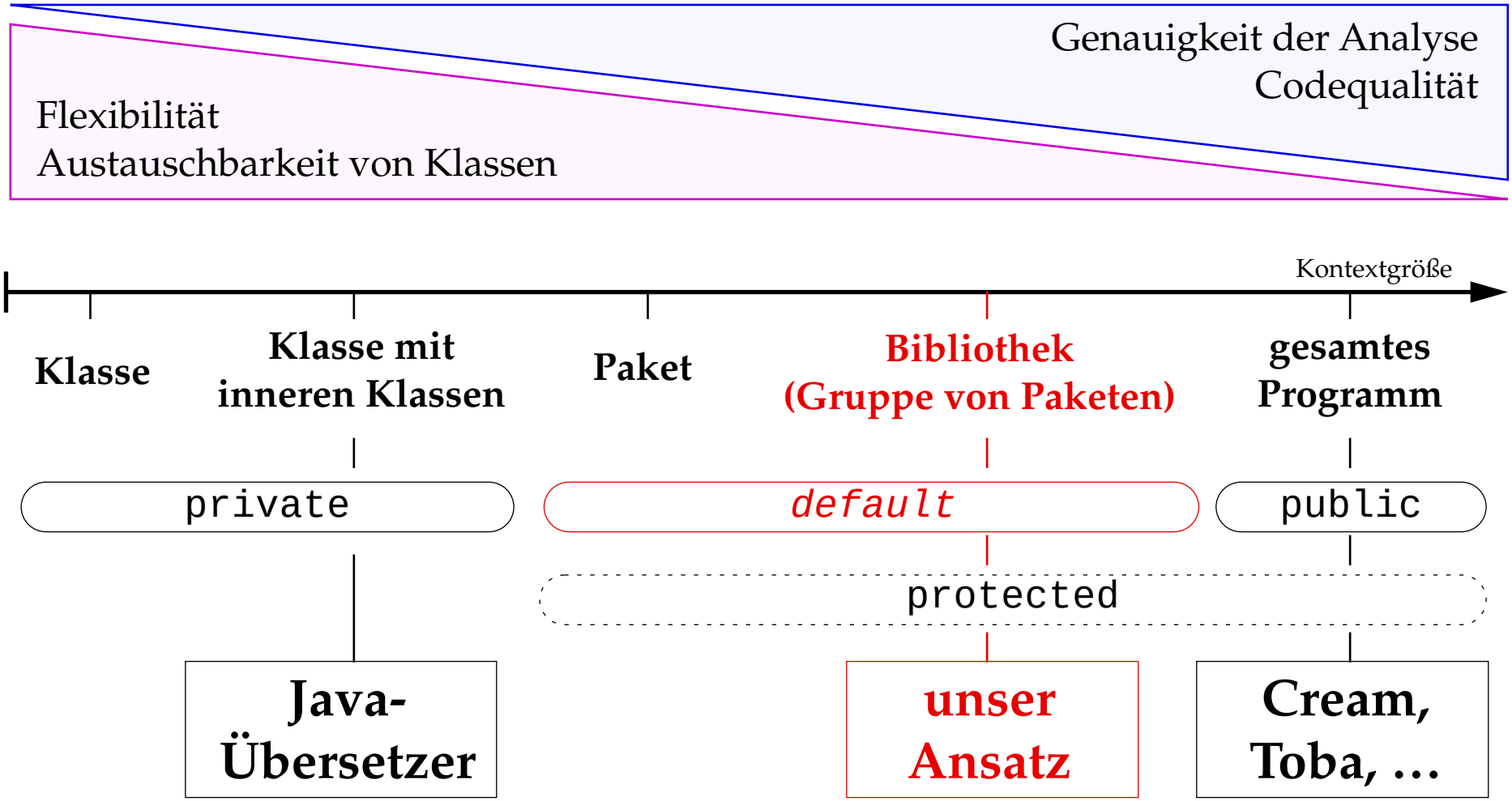
```
class B {
public static int q()
 { System.out.print(".");
   return 1001;
 }
}
```

Versiegeln des Analysekontextes



- **Hinzufügen von Klassen invalidiert evtl. All-Aussagen**
- **Löschen von Klassen erlaubt u.U. schärfere Aussagen**
- **dynamisches Laden von Klassen mittels `Class.forName()` unproblematisch**
  - **alle Kandidatenklassen im Paket werden mitversiegelt**

Größe des Analysekontextes

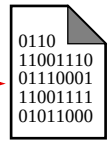
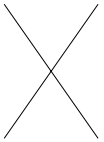


- **Paket bildet in Java natürlichen Namensraum/Kapselung von Softwarekomponenten**
- **Bibliothek bietet genauere Information bei expliziter Benutzung anderer Pakete aus der Bibliothek**

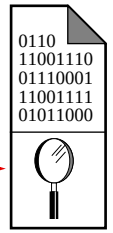
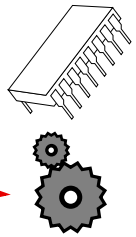
Ziele der Analyse (1)

Statische Analyse

Ergebnis	Aktion zur Laufzeit	Beispiel
1. Bytecode-Optimierung		

 optimierter Bytecode (im größeren Kontext)	 *keine*	`paket.A a;`
		`a = new paket.A();`
		`…`
		`for (int i = 0;`
		`  i < a.getSize();`
		`  i++)`
		`{ … }`
		schleifen-invariant

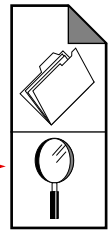
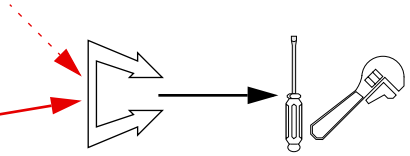
2. Bytecode-Annotation		

 Zusatzinformation zu Bytecodebefehlen	 Steuerung des JIT-Übersetzers (Interpr.)	`int[] a = new int[…];`
		`…`
		`int s = 0;`
		`for (int i = 1;`
		`  i < a.length;`
		`  i++)`
		`{ s += a[i] * a[i-1];`
		`}`
		keine Indexüberprüfung

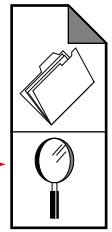
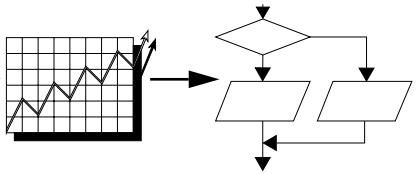
Ziele der Analyse (2)

Statische Analyse

Ergebnis	Aktion zur Laufzeit	Beispiel
3. Vorbereitung dynamischer Optimierung		

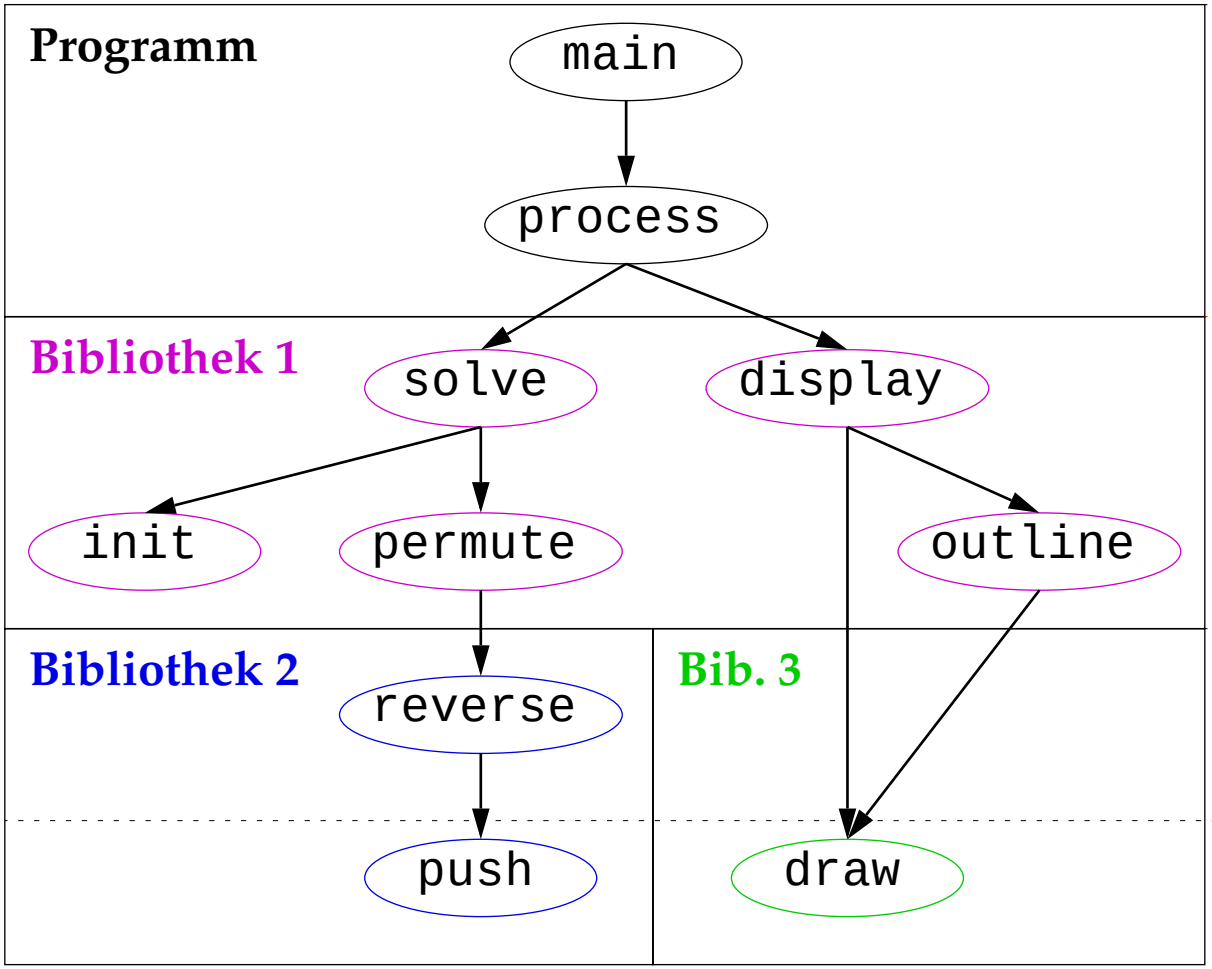
 Analyseinformation auf Bibliotheksebene	 Komposition, Optimierung	`Drawable d;`
		`…`
		`int m, area;`
		`m = d.getWidth() / 2;`
		`area = d.getHeight() *`
		`d.getWidth();`
		gemeinsamer Teilausdruck

4. Vorbereitung optimistischer Optimierung		

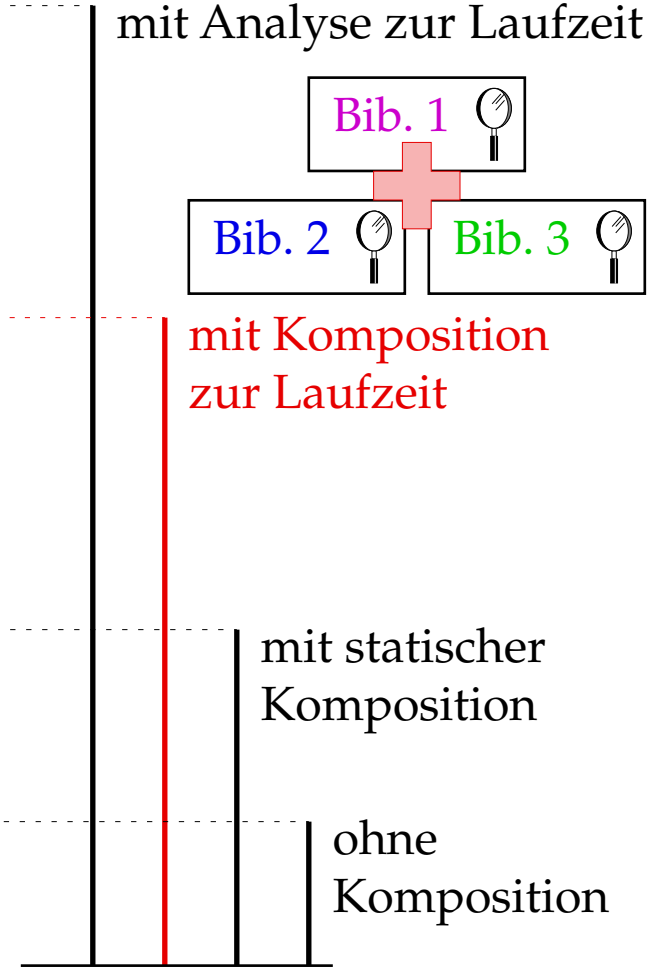
 Schleifenanalyse, Anwendungsstellen	 Programmbeobachtung, Methodenspezialisierung	`int calc (int x,`
		`boolean b)`
		`{ if (b) x = -x;`
		`x = f(x);`
		`if (b) x = -x;`
		`return x;`
		`}`
		b ist in fast allen Aufrufen `false`

Reichweite der Analyseinformation

Aufrufgraph eines Beispielprogramms



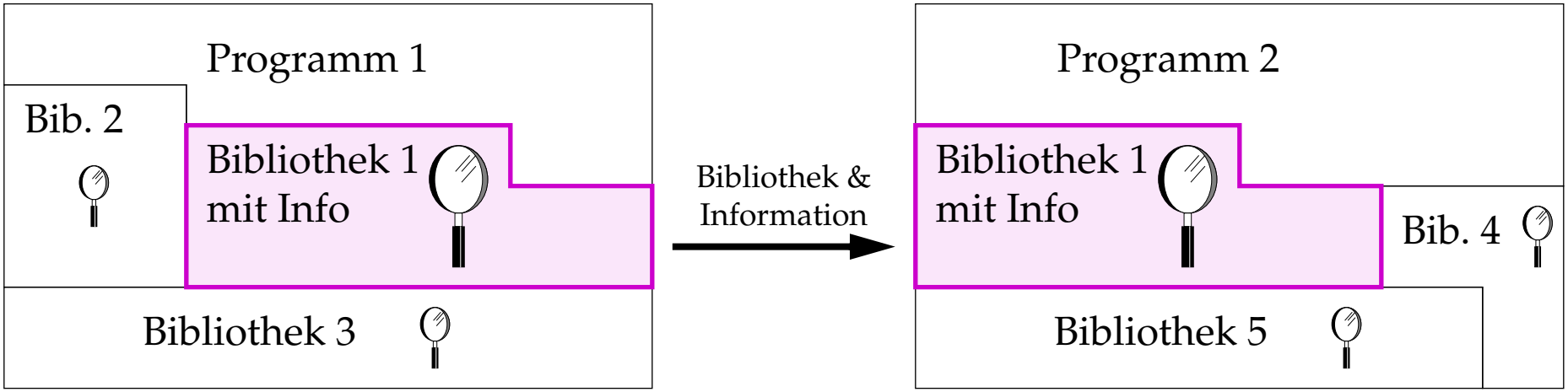
verfügbare Information



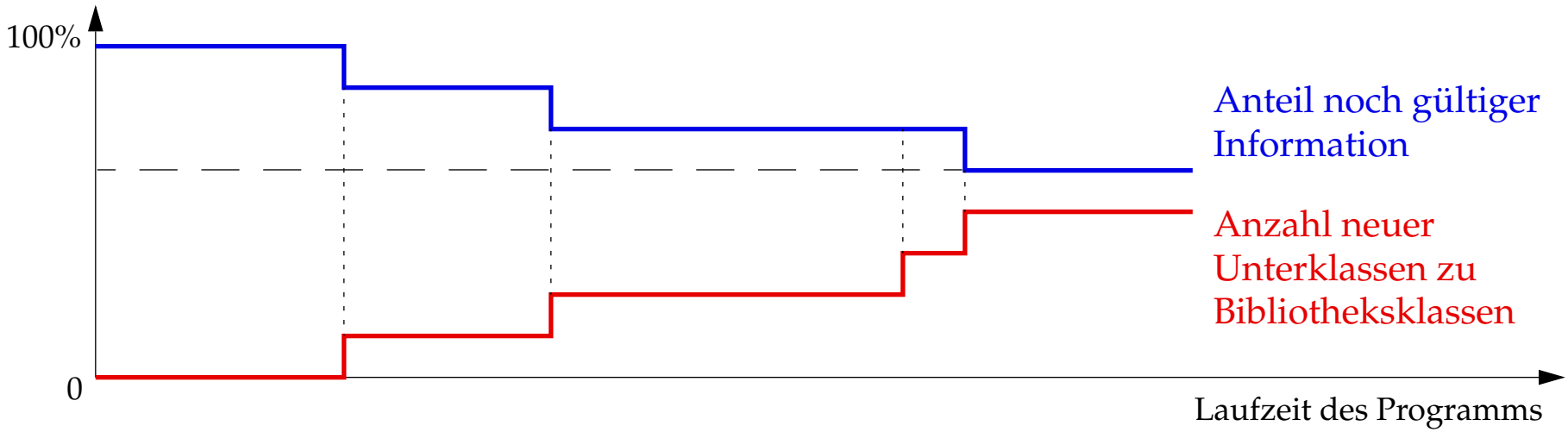
— lokale Analyse zur Laufzeit erlaubt globale Optimierungen in der nächsthöheren Schicht

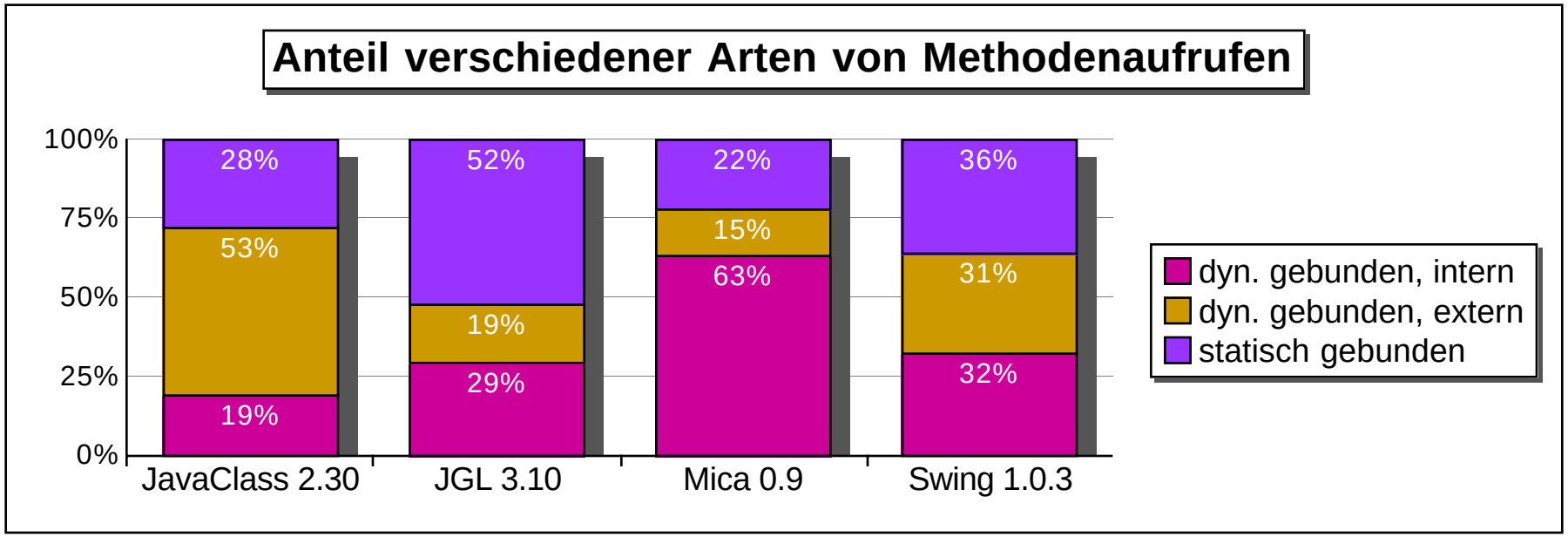
Lebensdauer der Analyseinformation

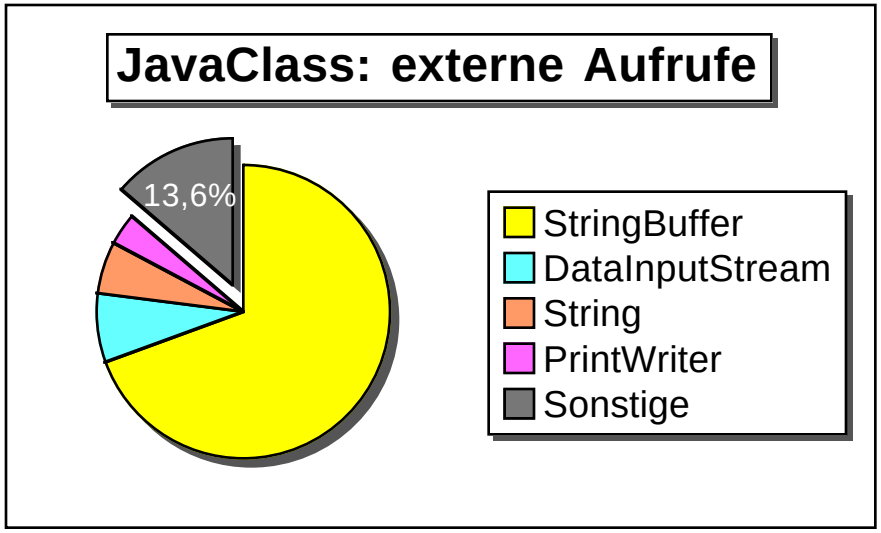
Wiederverwendung der Analyseinformation mit der Bibliothek

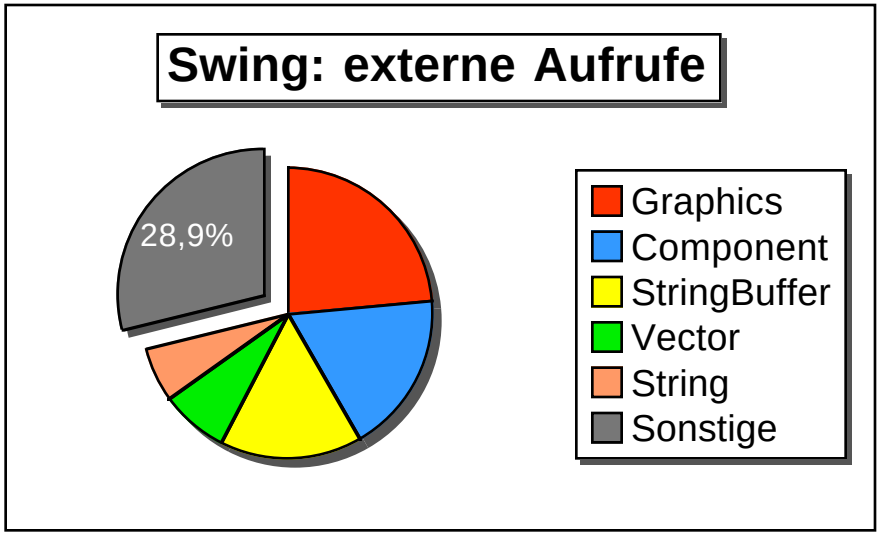


Invalidieren von Analyseinformation während eines Programmlaufs



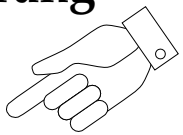

Anteil verschiedener Arten von Methodenaufrufen
100%
75%
50%
25%
0%
28%
53%
19%
52%
19%
29%
22%
15%
63%
36%
31%
32%
JavaClass 2.30
JGL 3.10
Mica 0.9
Swing 1.0.3
dyn. gebunden, intern
dyn. gebunden, extern
statisch gebunden

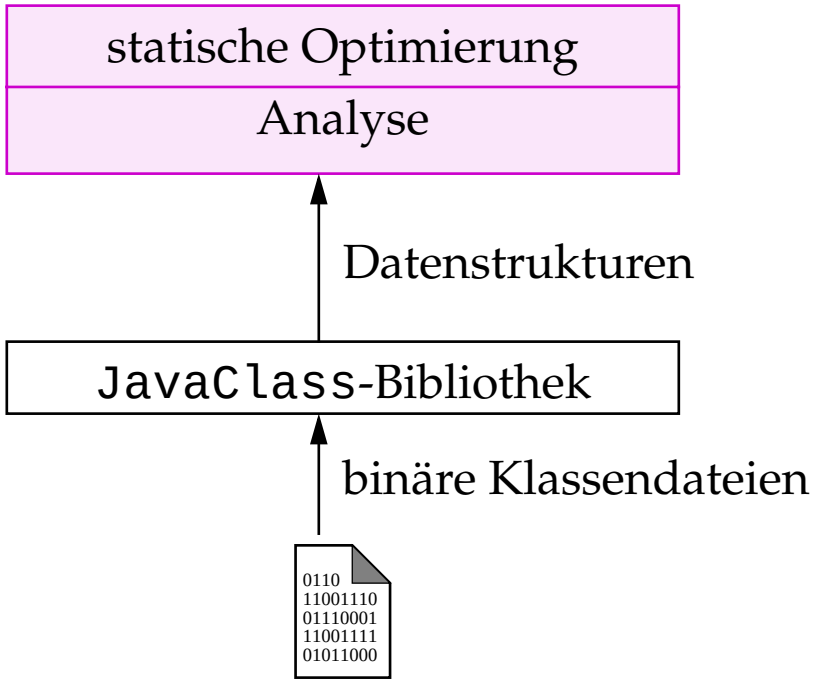

JavaClass: externe Aufrufe
13,6%
StringBuffer
DataInputStream
String
PrintWriter
Sonstige


Swing: externe Aufrufe
28,9%
Graphics
Component
StringBuffer
Vector
String
Sonstige

Realisierung eines Prototyps

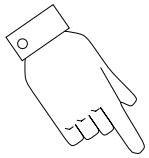
statische Analyse & Optimierung

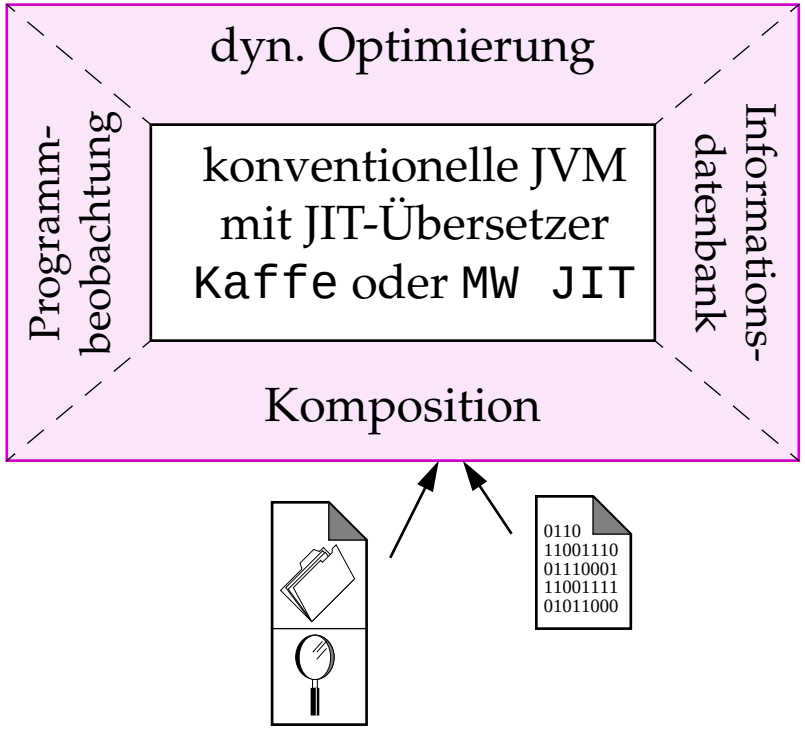




Implementierungssprache: Java
Plattform: viele

veränderte Laufzeitumgebung





Implementierungssprache: C/Assembler
Plattform: typischer RISC-Prozessor, z.B.
PowerPC oder SPARC

Zusammenfassung

— **zeitaufwendige Programmanalyse erfolgt statisch vor der Laufzeit**
  - **fortgeschrittene Analysen zur Laufzeit nicht praktikabel**

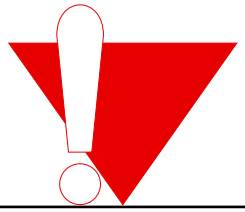
— **Bibliothek (Gruppe von Paketen [*packages*]) als Analysekontext**
  - **Analysekontext wird gegen Änderungen versiegelt**
  - **in der Praxis keine Einschränkung der Flexibilität**

— **Komposition zur Laufzeit liefert Bibliotheks-übergreifende Aussagen**
  - **stellt Austauschbarkeit von Software-Komponenten sicher**

— **Optimierung findet (soweit nötig) zur Laufzeit statt**
  - **zusätzlich statische Optimierung und Bytecode-Annotation**

— **Analyseinformation wird mit Bibliothek wiederverwendet**
  - **gewonnene Ergebnisse sind Plattform-unabhängig**

— **unterstützt Programm-spezifische Optimierungen gemäß Benutzungsmuster**
  - **spezifische Unterklassenbildung zu Bibliotheksklassen**
  - **dynamische Verwendung von Bibliotheksmethoden**



Ziel: Javas Flexibilität nicht mehr durch Geschwindigkeitsnachteile erkaufen