Transitiver Schutz in Java durch Sicherheitsmetaobjekte

Thomas Riechmann,

Franz J. Hauck,

Jürgen Kleinöder

http://www4.informatik.uni-erlangen.de/~riechmann/

riechmann@informatik.uni-erlangen.de

Überblick

- Sicherheit in Java
- Sicherheitsmetaobjekte
- Programmierung von Sicherheitsmetaobjekten
- Transitivität
- Zusammenfassung und Ausblick

Sicherheit in Java

- Sicherheit in Java nötig für:
  - ☞ Lokale Anwendungen: genau konfigurierbare Umgebung
  - ☞ Verteilte Anwendungen: Schutz zwischen Objekten
- Sicherheit in Java durch Objektorientierung
  - ☞ Objektreferenz ist „Capability“
  - ☞ Auch bei Verteilung (z.B. mit RMI)
- Sicherheit in Java durch Zugriffslisten („nur Systemklassen dürfen...“)

Sicherheit in Java

- Schutz muß in Applikationsklassen eingebaut werden
  - ☞ Mischung von Applikationscode und Sicherheitscode
  - ☞ Komplexitätsanomalie
    ⇒ Weniger Programmcode durch Objektorientierung
    ⇒ Mehr und komplexerer Sicherheitscode
  - ☞ Implementation von Applikationen mit Sicherheitsanforderungen oft „prozedural“
    - ➢ Mehr Programmcode (kaum Wiederverwendung)
    - ➢ Weniger Sicherheitscode: überschaubarer

Sicherheitsmetaobjekte (SMOs)

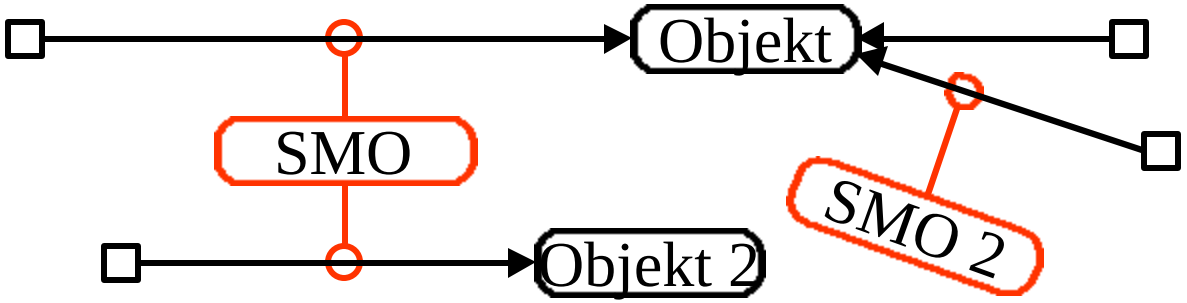
- Sicherheitsmetaobjekte:
  - ➢ Kontrollieren den Anwendungsablauf
  - ➢ Implementieren nur Sicherheitsstrategie
  - ➢ Oft anwendungsunabhängig realisierbar

☞ Anwendung kann ohne Berücksichtigung von Sicherheit implementiert werden

☞ Verwendung geeigneter Sicherheitsmetaobjekte für Ablaufumgebung

Sicherheitsmetaobjekte (SMOs)

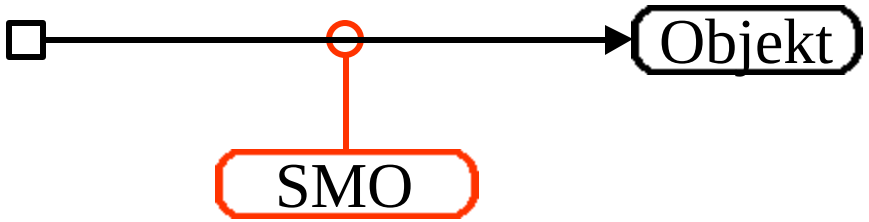
- Ein Sicherheitsmetaobjekt (SMO) realisiert Schutz einer oder mehrerer Objektreferenzen
  - ☞ SMO kann an eine Objektreferenz geheftet werden
    - ➢ Durch Applikation oder durch SMOs
  - ☞ SMO wird automatisch involviert bei
    - ➢ Methodenaufruf
    - ➢ Parameterübergabe / Ergebnisrückgabe



Programmierung von SMOs

- Implementation eines SMO für zeitlich begrenzte Gültigkeit

```
class ExpireSMO extends SMO {
        void incomingCall( ... ) {
                ...
                if ( currentDate >= expireDate )
                        throw new SecException( );
}       }
```

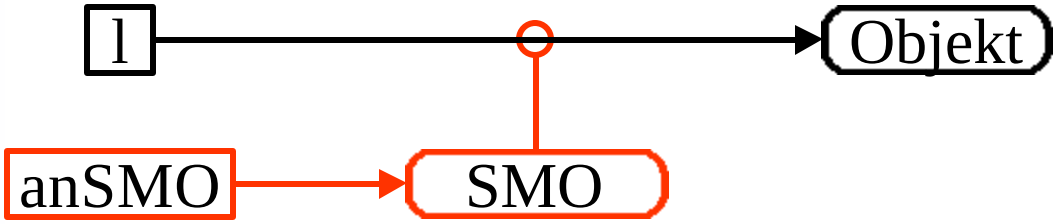


Anwendung von SMOs

- Anheften von SMOs:

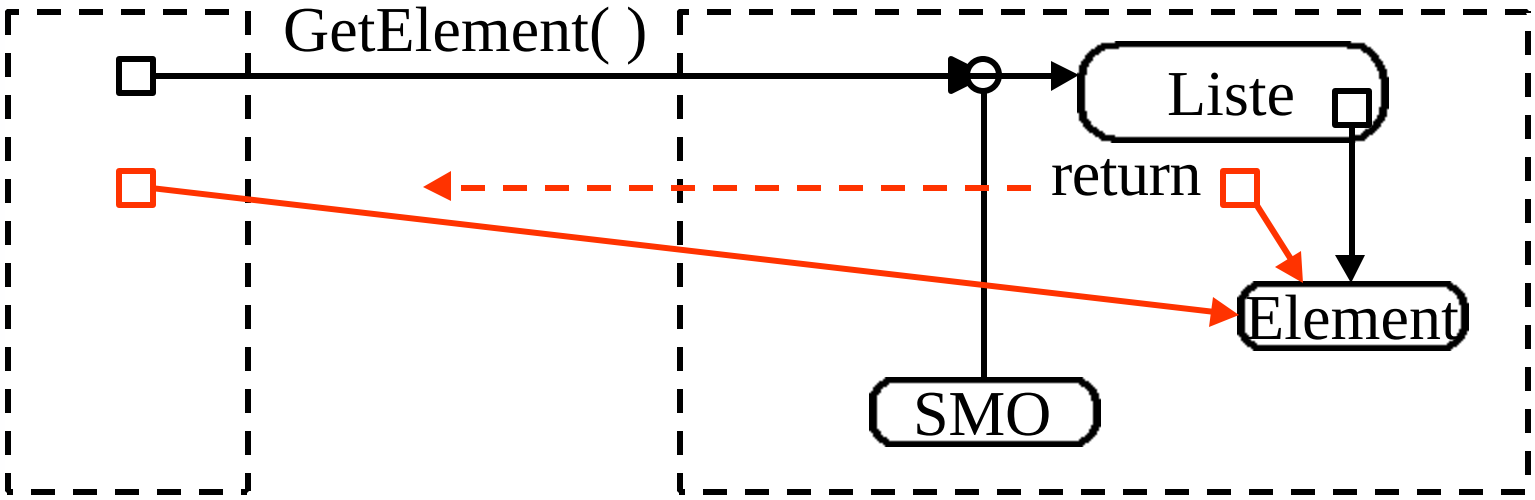
```
List  l = ...
SMO anSMO = new ExpireSMO(...);

l  =  (List) anSMO . dstAttachTo ( l );
....
l . GetElement (  );   // Aufruf wird geprüft
```



Transitivität

- Problem: Austausch von Objektreferenzen
- Beispiel: Referenz auf eine Liste
  - ☞ „GetElement“ Methode liefert Eintragsreferenz zurück
  - ☞ Eintragsobjekt ist nicht geschützt



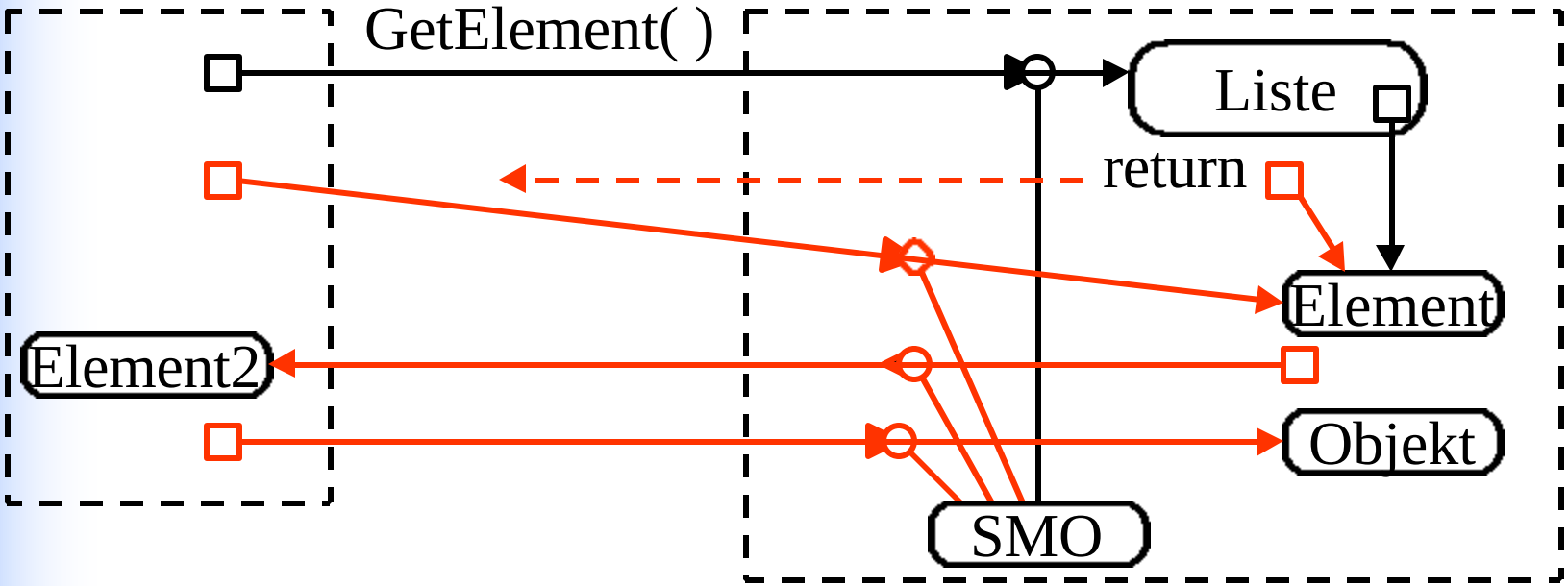
Transitivität

- Lösung: Transitivität von SMOs
- SMO erhält Referenzen, bevor sie übergeben werden
  - ☞ Das SMO kann die Übergabe verhindern
  - ☞ Das SMO kann sich oder ein anderes SMO an Referenz heften
  - ☞ Implementation im SMO:
    - ➢ incomingRef / outgoingRef Methode:

```
Object outgoingRef (Object outRef) {
        return this.dstAttachTo(outRef);
}
```

Transitivität

- Volle Transitivität der Sicherheitsstrategie
- Initiale Zuweisung von SMOs durch Applikation
- Danach: Transitivität über alle ausgetauschten Referenzen



Prototyp

- Prototyp in Java
  - ☞ Ohne Modifikation der Java-Maschine (Proxy-basiert)
  - ☞ Kleinere Einschränkungen ähnlich Java-RMI
- Lokales Modell:
  - ➢ SMOs (Capabilities, Transitivität, Zugriffslisten)
- Verteiltes Modell:
  - ➢ „Sichere“ Referenzen, „sichere“ Aufrufe
  - ➢ Identitäten, Zugriffslisten, Fernaufrufe mit Schlüsseln

Zusammenfassung und Ausblick

- SMOs implementieren konfigurierbare Capabilities:
  - ☞ Trennung von Applikationssemantik und Sicherheitsstrategie
  - ☞ Rückruf, Zugriffsbeschränkung, zeitliche Begrenzung der Gültigkeit
  - ☞ Zusätzlich: Transitivität
- Konfigurierbare Zugriffslisten
  - ☞ Konfigurierbare Identitäten: Rollenbasierte Identitäten
- Formales Modell
  - ☞ Beweise von Sicherheitseigenschaften