Grundlagen der Informatik für Ingenieure I

17. Weitere Themen kurz angerissen

17.1 Garbage Collection

- In den herkömmlichen Programmiersprachen hat man über das statische Anlegen von Datenbereichen hinaus die Möglichkeit, selbst, zur Laufzeit, eine dynamische Speicherverwaltung zu organisieren, um ggf. dem dynamischen Verhalten eines Datenbestandes besser gerecht werden zu können *(malloc, free)*. (Beispiel: verkettete Listen)
- Objektorientierte Systeme sind hochgradig dynamisch. Jedes *"new"* ist mit einer dynamischen Speicheranforderung verbunden.
- Wie gibt man in Java nicht mehr benötigten Speicherbereiche (Objekte) wieder frei?
  - Ein explizites *"deallocate"* gibt es nicht, jedoch kann man der Speicherverwaltung durch Zuweisung eines *null-pointers*; **besser einer** *null-referenz* indirekt mitteilen, dass man dieses Objekt nicht mehr verwenden möchte.
  - Für ebenfalls indirekt mit dem *"new"* angeforderten Systemressourcen, wie z. B. E/A-Kanälen, leistet der Aufruf der Methode *close()* die gleichen Dienste.

17.1 Garbage Collection

- Der Aufruf eines *"deallocate"* beim Unix-Betriebssystem hat zur Folge, dass sich das Betriebssystem **synchron** um die Freigabe des Speichers kümmert.
- Bei Java geht man einen anderen Weg:
  Ein Aktivitätsträger, der auf niedriger Priorität im Hintergrund läuft, überprüft periodisch, ob auf Objekte noch Referenzen existieren. Ist dies nicht der Fall, schlägt er den Speicher wieder seinem Freispeicherpool zu. Bei diesem **asynchronen** Verfahren der Speicherverwaltung spricht man von *Garbage Collection.*
- Die Idee dabei ist, dem Benutzer nicht unnötig mit Aufräumarbeiten zu belästigen (was man in Wirklichkeit dann ja doch tut!) und Prozessorleistung zu nutzen, wenn sie sonst keiner benötigt. Besonders heikel ist dieses Verfahren bei *"Notsituation"*, z. B. wenn der Speicher ausgeht.
  - Gute *Garbage Collectoren* werden mit solchen Situationen fertig, ohne dass es der Benutzer merkt.
  - Kritisch (besser ungeeignet!) sind solche Verfahren für Echtzeitsysteme.

17.2 Java und Security

- Java ist - wenn auch nicht ursprünglich - auch als Sprache für Programmsysteme zum *Downloaden* über das *Internet* entwickelt worden.
- Es ist offensichtlich, dass das nur funktionieren kann, wenn dieser Sprache ein Sicherheitssystem zugrunde liegt, dass dem Benutzer zusichern kann, dass er genau das Programm bekommt, was er zu bekommen glaubt und das kein Programm in der Lage ist, lokal irgendeinen Schaden anzurichten.
- Die Zeit reicht in dieser Vorlesung nicht, ins Detail zu gehen, jedoch sei hier soviel gesagt:
  - Das Java Security-Modell besteht aus 4 Ebenen
    - Die erste Ebene ist die Sprache selbst; sie beinhaltet Regeln, die vom Compiler überprüfbar sind und durchgesetzt werden.
    - Dadurch, dass der (Applet) - Code interpretativ abgearbeitet wird, ist das Java-Laufzeit-System in der Lage den Code *"zu screenen"* - auf Codeschlüssel zu überwachen, deren lokale Ausführung verboten ist.
    - Der *class-loader* überprüft während des Ladens, dass der Namensraum und die Zugriffsrestriktionen eingehalten werden.

17.2 Java und Security

- Für die Überwachung, ob das was ich bekomme auch das ist, was ich glaube zu bekommen, gibt es Verfahren, die das sicherstellen können (*Digital Fingerprints*, Authentisierungsverfahren).
- Mit Kryptosierungsverfahren werden Daten verschlüsselt und so gegen Missbrauch geschützt.

❏ Zum Abschluss: Sicherheitssysteme können noch so gut sein; wenn Systeme schlampig administriert werden, bringen auch ausgefeilte Sicherheitssysteme keine Sicherheit!

❏ Auch *Firewalls* können diese Probleme nicht lösen!

17.3 Native Methods und Libraries

❏ Aus Effizienzgründen oder auch aus systemarchitekturbedingten Gründen kann es unabdingbar sein, Programmteile in Java einzubinden, die nicht in Java, sondern in einer anderen Sprache (z. Zt. nur C!) geschrieben sind. Wenn Sie vor diesem Problem stehen, schauen sie in die einschlägige Literatur unter den o. g. Schlagworten nach.

❏ Mit einem “Applicationbuilder” sind sie in der Lage Javacode auf Maschinensprache “herunterzucompilieren”. Diese “Applicationbuilder” sind natürlich rechnerarchitekturabhängig und werden nicht für jeden Rechner angeboten.

17.3 Java Beans

- ...sind letztendlich besondere Klassen, organisiert in Klassenbibliotheken, mit denen man in der Lage ist, mit Hilfe eines über eine graphische Oberfläche bedienbaren "*builder tool*" aus wiederverwendbaren universellen Softwarekomponenten neue, mächtigere Softwarekomponenten zu komponieren ohne eine Zeile Code zu schreiben.
- Damit das funktioniert, müssen die Klassen nach gewissen Regeln gebaut sein, die, wie in Java üblich, durch *Interfaces* und abstrakte Klassen beschrieben werden *(java.bean-package)*. Für die Interaktion zwischen den *Beans* werden *Events* verwendet.
- http://java.sun.com/beans