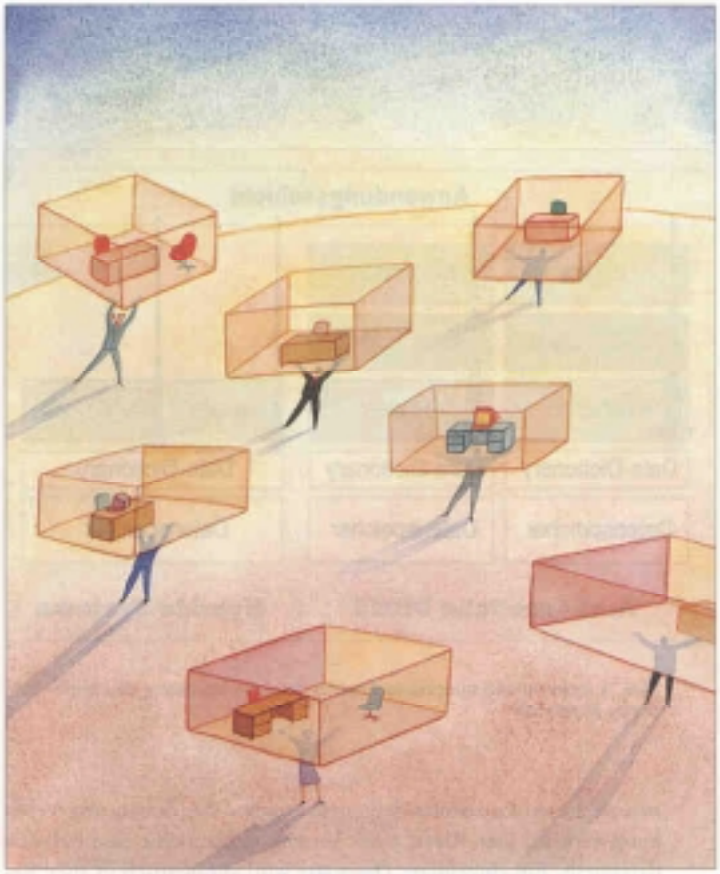

Aufräumaktion

Effiziente Speicherverwaltung

RONALD DAUSTER

Effiziente und zuverlässige Speicherverwaltung ist für die Performance und Stabilität insbesondere von Server-Applikationen von großer Bedeutung. Java-Programmierer vertrauen hier gerne auf den Garbage Collector. Durch automatische Speicherbereinigung verschwinden die Herausforderungen der Speicherverwaltung aber nicht, sondern stellen sich nur in anderer Form dar. Der Artikel fasst für die Praxis relevantes Wissen über Garbage Collection zusammen und zeigt auf, wie diese Kenntnisse schon im Design vorteilhaft eingesetzt werden können.

Mit Java hat erstmals eine Programmiersprache mit automatischer Speicherverwaltung („Garbage Collection") weite Verbreitung erreicht. Auch in anderen objektorientierten Programmiersprachen hielt Garbage Collection bereits Einzug. Die bekanntesten dieser Sprachen sind Smalltalk und Eiffel, aber sie haben im Vergleich mit Java eine wesentlich geringere Bedeutung am Markt. Gleichzeitig sind die meisten Veröffentlichungen zum Thema Garbage Collection theoretischer Natur und nur für Spezialisten interessant.

Dadurch erklärt sich, weshalb viele erfahrene Softwareentwickler keine oder nur oberflächliche Kenntnisse auf dem Gebiet der Garbage Collection haben. Dieser Artikel vermittelt das für die Praxis relevante Wissen über Garbage Collection und zeigt, wie diese Kenntnisse im Design vorteilhaft eingesetzt werden können.

Lebende und tote Objekte

Die zentrale Aufgabe eines Garbage Collectors besteht darin, Objekte zu erkennen, die den weiteren Programmablauf nicht mehr beeinflussen können, um den von diesen Objekten belegten Speicher dem Programm zur erneuten Verwendung zugänglich zu machen. Solche Objekte werden als tot bezeichnet, alle übrigen als lebend.

Wie lassen sich nun lebende von toten Objekten unterscheiden? Eine zu jedem Zeitpunkt korrekte Klassifikation aller Objekte eines Programms ist im Allgemeinen nicht möglich. Daher verwenden alle bekannten Garbage Collection-Algorithmen eine konservative Annäherung und klassifizieren ein Objekt anhand seiner Erreichbarkeit: Nur wenn es an keiner Stelle des Programmcodes mehr eine Referenz auf ein Objekt gibt, wird es als tot angesehen.

Es ist leicht einzusehen, dass ein Objekt, das gar nicht erreichbar ist, sicherlich keinen Einfluss mehr auf den weiteren Programmablauf hat. Andererseits ist die Bedingung zu allgemein: Man muss nur einem beliebigen Programm ein völlig unnützes Singleton [Ga95] hinzufügen, das nirgends verwendet wird. Dieses Singleton ist von jeder Stelle des Programms erreichbar, da es aber nirgends verwendet wird, muss es eigentlich als tot klassifiziert werden.

Zugegeben, das Beispiel wirkt gekünstelt und scheint keine Bedeutung für die Praxis zu haben. Es zeigt auf drastische Weise den zentralen Unterschied zwischen der „Denkweise" des Garbage Collectors und der eines Programmierers. Es ist genau dieser Unterschied zwischen den Klassifikationen „ist noch erreichbar" und „wird noch gebraucht", der für viele Speicherprobleme verantwortlich ist und den es zu verinnerlichen gilt.

Zuvor muss aber der Begriff der Erreichbarkeit präzisiert werden. Für alle praktischen Belange kann die Erreichbarkeit durch zwei einfache Regeln charakterisiert werden:

- Regel 1: Ein Objekt, das im Programm namentlich benannt werden kann, ist erreichbar.
- Regel 2: Ein Objekt ist erreichbar, wenn ein anderes erreichbares Objekt eine Referenz auf dieses Objekt besitzt.

Aus der ersten Regel folgt, dass alle Klassen (Instanzen von java.lang.Class) und die in statischen Feldern gespeicherten Objekte immer erreichbar sind. Außerdem sind alle Objekte erreichbar, die von lokalen Variablen einer Methode referenziert werden, und zwar solange

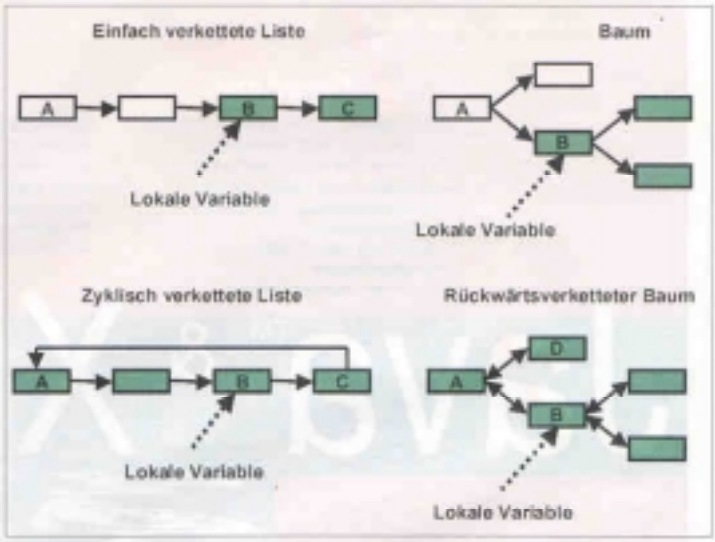

Abb. 1: Elementare Objektstrukturen

die Methode ausgeführt wird. Insbesondere bezeichnet this für nichtstatische Methoden ein erreichbares Objekt.

Der Besitz einer Referenz, der in der zweiten Regel erwähnt wird, bedeutet nichts anderes, als dass eine Referenz in einem Feld des zweiten Objektes gespeichert wird. Diese Regel ist natürlich rekursiv anzuwenden, sodass z. B. eine ganze verkettete Liste erreichbar ist, wenn ihr Kopf erreichbar ist.

Ihre eigentliche Bedeutung erhalten diese beiden Regeln erst durch die folgende dritte:

- Regel 3: Es gibt keine anderen erreichbaren Objekte.

Auf diesen drei Regeln beruht die Wirkungsweise aller Garbage Collectoren für Java. Die erste Regel bestimmt die Menge der a priori als erreichbar erkannten Objekte, im Jargon als Wurzeln („roots", „root set") bezeichnet. Die zweite Regel zeigt, wie ausgehend von diesen Wurzeln alle weiteren erreichbaren Objekte bestimmt werden und in der Tat findet sich in allen Garbage Collection-Algorithmen diese rekursive oder iterative Suche. Die dritte Regel besagt nun, dass alles Übrige als freier Speicher angesehen werden kann.

Strukturen

Die drei oben angegebenen Regeln stellen zwar ein Kriterium dar, anhand dessen zur Laufzeit des Programms die Erreichbarkeit jedes einzelnen Objektes entschieden werden kann, aber für die praktische Arbeit ist es notwendig, die Lebensdauer von Objekten schon zur Entwurfszeit abschätzen zu können. Dazu ist es nötig, aus lokalen Eigenschaften die Lebensdauer eines Objektes erkennen zu können.

Der erste Schritt ist die Klassifikation der möglichen Objektstrukturen. Bei den in Abbildung 1 dargestellten Strukturen handelt es sich aus Sicht des Softwareentwicklers um zwei Listen und zwei Bäume. Aus der Perspektive des Garbage Collectors ist die Klassifikation orthogonal dazu: Die gefüllten Kästchen stellen die lebenden Objekte dar, wenn die lokale Variable die einzige externe Referenz auf diese Strukturen enthält.

In der einfach verketteten Liste und im Baum lebt nur ein Teil der Objekte. Es fällt allerdings auf, dass das Objekt C von B am Leben erhalten wird; das referenzierte Objekt lebt also mindestens so lange, wie der Besitzer der Referenz. Daraus folgt, dass Referenzen von kurzlebigen Objekten zu langlebigen unproblematisch sind, während umgekehrte Referenzen wieder gelöst werden müssen, da sonst das (eigentlich) kurzlebige Objekt unnötig erhalten bleibt.

Andererseits lebt oder stirbt die zyklisch verkettete Liste und der rückwärts verkettete Baum stets als Ganzes. Sie werden dadurch gekennzeichnet, dass jedes Objekt von jedem anderen Objekt direkt oder indirekt erreichbar ist. In der Graphentheorie werden solche Strukturen als „stark zusammenhängend" bezeichnet.

Nun wird in einer Applikation kaum jedes Objekt von jedem anderen erreichbar sein, aber fast immer gibt es stark zusammenhängende Komponenten, also Gruppen von Objekten, die diese Eigenschaft besitzen. Sie gilt es zu identifizieren. In der Praxis häufig vorkommende Beispiele stark zusammenhängender Komponenten sind DOM-Bäume oder Hierarchien von GUI-Elementen.

Um einem verbreiteten Missverständnis vorzubeugen: Kein Garbage Collector hat Schwierigkeiten, solche stark zusammenhängenden Komponenten oder überhaupt zyklische Strukturen zu recyclen. Es ist nur so, dass alle Objekte in einer solchen Komponente die gleiche Lebensdauer haben. Durch diese Eigenschaft werden sie im Gegenteil zu positiven Designelementen, denn bei der Betrachtung der Beziehungen zwischen Objekten kann man die stark zusammenhängenden Komponenten wie ein Objekt behandeln. Die Referenzen innerhalb der Komponente sind unbedeutend und neue Referenzen können bedenkenlos hinzugefügt werden.

Analyse von Beziehungen

Die nachträgliche Analyse der möglichen Objektstrukturen ist schwierig. Hauptsächlich ist dies durch die mangelnde Werkzeugunterstützung bedingt. Eine Besserung ist auch nicht zu erwarten, da die zur Laufzeit möglichen Objektstrukturen weder aus dem Quellcode noch aus einem Designmodell automatisch abgeleitet werden können. Gerade statische Modelle, z. B. UML-Klassendiagramme, täuschen oft über die tatsächlichen Referenzen zwischen Objekten hinweg. Ein Beispiel mag dies veranschaulichen (s. Abb. 2).

Eine Applikation (Paket `app`) verwendet eine Java-Bean (Paket `bean`). Die Bean wurde unabhängig von der Applikation entwickelt und verwendet nur das Paket `java.beans`. Es scheint keinerlei Zyklus zu geben und nichts deutet auf die Existenz einer paketübergreifenden stark zusammenhängenden Komponente hin. Zur Laufzeit erstellt ein Frame der Applikation eine Instanz der Bean und registriert einen Listener. Bei der Betrachtung des Objektdiagramms fällt nur der Zyklus unmittelbar auf: Der Frame referenziert die Bean, die Bean den Listener und der Listener als innere Klasse hat eine implizite Referenz auf den Frame.

Das Beispiel ist durchaus typisch. Die geschilderte Situation tritt in der Praxis häufig in Verbindung mit dem Beobachtermuster [Ga95] auf, in Java meist in Form von Event-Listenern. In dieser Gestalt ist es besonders heimtückisch, da die implizite Referenz des meist als innere Klasse implementierten Listeners nur zu leicht übersehen werden kann.

Die Schwierigkeit der Analyse legt also den Versuch nahe, bereits beim Design den Objektstrukturen gründliche Beachtung zu schenken. In Ermangelung spezieller Verfahren führt der erfolgversprechendste Weg hier, wie so oft, über ein altes Prinzip: „Teile und herrsche."

Lokale und globale Beziehungen

Grundlegendes Prinzip jedes guten Softwaredesigns ist die Modularisierung: Die Klassen innerhalb eines Moduls (Paket, Bean, EJB usw.) stehen zueinander in engerer Beziehung als zu Klassen aus anderen Modulen. Die wichtigste Form der Modularisierung in Java ist zweifellos die Aufteilung der Klassen auf die Paketstruktur, schon alleine, weil sie immer vorgenommen werden muss. Mit diesen engen Beziehungen der Klassen innerhalb eines Paketes geht auch häufig ein enges Geflecht von Referenzen einher. Das ist regelmäßig dann der Fall, wenn die Klassen eines Paketes kollaborieren, um eine Funktionalität zu bieten, die mehr ist als die Summe der Teile. Beispiele hierfür sind die Pakete `java.sql`, `javax.swing` und `org.w3c.dom`.

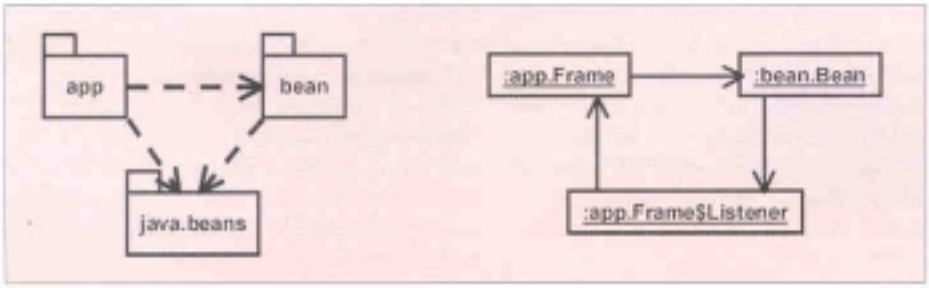

Abb. 2: Paketabhängigkeiten und Objektreferenzen

Kleine Taxonomie der Garbage Collection

Verfahren zur automatischen Speicherverwaltung haben bereits eine lange Tradition und finden seit Jahrzehnten in eher funktionalen Sprachen Anwendung. Lisp ist hier als ältester und bekanntester Vertreter zu nennen. Die Anzahl der verschiedenen Algorithmen ist nur für den Spezialisten vollständig zu überschauen. Für den Einstieg in diese durchaus faszinierende Thematik sei der ausgezeichnete Übersichtsartikel von Wilson [Wi96] empfohlen.

Dennoch lassen sich alle bekannten Verfahren nach verschiedenen Kriterien klassifizieren. Diese Kriterien werden im Folgenden kurz erläutert.

Mark-Sweep, Mark-Compact und Two-Space Copying Collection

Die erste Klassifizierung kann anhand des Verfahrens erfolgen. Die meisten aktuellen Algorithmen lassen sich in eine der drei Kategorien „Mark-Sweep", „Mark-Compact" und „Copying" einteilen. (Es gibt allerdings eine verwirrende Anzahl von Mischformen, z. B. Varianten des Copying Collectors, die gar nicht kopieren.)

Mark-Sweep ist der wahrscheinlich bekannteste aller Garbage Collection-Algorithmen. Von den Wurzeln ausgehend werden alle direkt erreichbaren Objekte markiert, z. B. durch Setzen eines Markbits in den Objekten. Ausgehend von diesen markierten Objekten werden rekursiv alle weiteren erreichbaren Objekte markiert. Danach wird der gesamte Speicher (genaugenommen: der „Heap") durchsucht und alle nicht markierten Objekte werden freigegeben. Der wichtigste Vorteil des Verfahrens ist, dass die lebenden Objekte niemals verschoben werden müssen, die Speicheradresse eines Objektes also während seiner gesamten Lebensdauer konstant bleiben kann. Der größte Nachteil besteht darin, dass die Laufzeit einer Garbage Collection unabhängig von der Anzahl der lebenden Objekte immer proportional zur Heap-Größe ist.

Eine Variante dieses Verfahrens sind *Mark-Compact*-Algorithmen. Die erste Phase, das Markieren, verläuft wie beim Mark-Sweep-Collector. Danach aber werden die lebenden Objekte einfach an ein Ende des Speichers verschoben. Sie bilden damit einen zusammenhängenden Speicherbereich, alles Übrige ist freier Speicher. Dieses Verfahren hat zwei Schwierigkeiten. Die erste ist allen Algorithmen gemein, die Objekte im Speicher verschieben: Wenn ein Objekt verschoben wird, müssen natürlich alle Referenzen auf das verschobene Objekt aktualisiert werden. Das zweite Problem ist spezifisch für Mark-Compact-Algorithmen: Die Verschiebung kann nicht in beliebiger Reihenfolge geschehen, da sonst der Platz, an den ein Objekt verschoben werden soll, schon von einem noch nicht verschobenen Objekt belegt sein kann. Daher ist mindestens ein weiterer Durchlauf aller lebenden Objekte erforderlich, um eine geeignete Verschiebungsreihenfolge zu ermitteln. Auf der Haben-Seite hat der Mark-Compact-Algorithmus eine Laufzeit proportional zur Anzahl der lebende Objekte zu verbuchen. Außerdem vermeidet er die Fragmentierung des freien Speichers.

In ihrer reinsten Form findet man kopierende Verfahren bei den *Two-Space Copying*-Algorithmen. Sie vermeiden die Probleme der Verschiebungsreihenfolge, die im Mark-Compact-Verfahren auftreten, dadurch, dass der Speicher in zwei gleich große Halbräume aufgeteilt wird, von denen stets nur einer verwendet wird. Alle Speicheranforderungen werden aus diesem einen Halbraum (dem „From-Space") befriedigt. Ist dieser voll, so wird bereits während der Markierungsphase jedes noch lebende Objekt gleich in den anderen Halbraum (den „To-Space") kopiert. Ein eigentliches Markieren ist also gar nicht nötig und das Kopieren selbst ist unproblematisch, da ja ein zusammenhängender Speicherbereich von garantiert ausreichender Größe zur Verfügung steht. Nach der Kollektion wird die Rolle der Halbräume vertauscht. Der einzige, allerdings gravierende Nachteil des Verfahrens gegenüber den Mark-Compact-Algorithmen besteht darin, dass nur die Hälfte des verwendeten Speichers der Applikation zur Verfügung steht.

Generational Collection und parallele Verfahren

Die zuvor beschriebenen Verfahren sind am leichtesten zu implementieren, wenn die Garbage Collection synchron erfolgt, d. h. während der Garbage Collection „steht" die Applikation. Da dies zu Pausen führt, die durchaus mehrere Sekunden dauern können, ist eine solche Implementierung für interaktive Applikationen oder Echtzeitprogramme mit kurzen Reaktionszeiten unbrauchbar. Andererseits hat jedes andere Verfahren die für den Garbage Collector unangenehme Eigenschaft, dass die Applikation den Graphen der erreichbaren Objekte ändern kann, während der Collector versucht, ihn zu durchlaufen. Daher muss der Garbage Collector über alle relevanten Veränderungen informiert werden. In der Praxis geschieht das meistens über eine Schreibsperre („write barrier"). Das heißt, schreibende Operationen, die eine neue Referenz in ein Objekt eintragen wollen, werden abgefangen und geeignet behandelt. Wie diese Behandlung im Einzelfall aussieht, hängt stark vom verwendeten Algorithmus ab.

Generationskollektoren versuchen, die Pausenzeiten zu reduzieren, in dem sie zumeist nur einen Teil des Heaps einsammeln. Dies beruht auf der Beobachtung, dass der größte Teil der erzeugten Objekte nur eine sehr kurze Zeit überlebt, während diejenigen Objekte, die einige wenige Läufe des Garbage Collectors überlebt haben, meist sehr langlebig sind. Daher wird der Heap in zwei oder mehr Bereiche aufgeteilt, in denen Objekte verschiedenen Alters residieren. Die meisten Kollektionen umfassen dann nur denjenigen Bereich des Heaps, in dem die seit der letzten Kollektion neu erzeugten Objekte angelegt werden. Überlebt ein Objekt mehrere solcher Kollektionen, wird es in den Bereich der langlebigen Objekte verlegt (auch „mature-space" genannt). Allerdings muss bei der Ermittlung der Erreichbarkeit nicht nur der Bereich der jungen Objekte, sondern es müssen auch die Referenzen aus dem Mature-Space in die jüngere Generation berücksichtigt werden. Um zu diesem Zweck nicht den gesamten Mature-Space nach solchen Referenzen durchsuchen zu müssen, merkt sich die Speicherverwaltung diese Referenzen in einer separaten Liste. Dazu wird die bereits erläuterte Schreibsperre verwendet: Jeder Versuch, eine Referenz in ein Objekt der älteren Generation einzutragen, wird abgefangen. Wenn die Referenz in eine jüngere Generation zeigt, wird die entsprechende Liste aktualisiert.

Gelegentlich müssen aber auch die älteren Generationen in die Garbage Collection einbezogen werden. Die einfachste Methode besteht darin, in diesen Fällen einen vollständigen Garbage Collection-Zyklus durchzuführen – aber damit wird das Problem der Pausenzeiten wieder akut. Es gibt verschiedene Ansätze, den vollständigen Zyklus zu vermeiden. Einer der vielversprechendsten ist der „Train-Algorithmus" [Se95], der als Alternative zu den herkömmlichen Verfahren seit JDK 1.3 in SUNs Virtual Machine implementiert ist.

Der zweite Ansatz, die Pausenzeiten des Kollektors zu verringern, besteht darin, die *Kollektion selbst als nebenläufige Routine* durchzuführen. Dies kann entweder als Co-Routine oder als eigener Thread geschehen. Auf diese Weise wird die Laufzeit des Garbage Collectors gleichmäßig auf die Applikation verteilt. Hierbei besteht die Schwierigkeit, dass der Garbage Collector sicherstellen muss, dass die Applikation die bisher vom Kollektor geleistete Arbeit nicht zwischenzeitlich wieder zunichte machen kann. Auch dafür wird häufig eine Schreibsperre eingesetzt, die schreibende Zugriffe auf den bereits durchsuchten Teil des Speichers abfängt (Ein solcher nebenläufiger Kollektor wird voraussichtlich als Neuerung in JDK 1.4 zur Verfügung stehen).

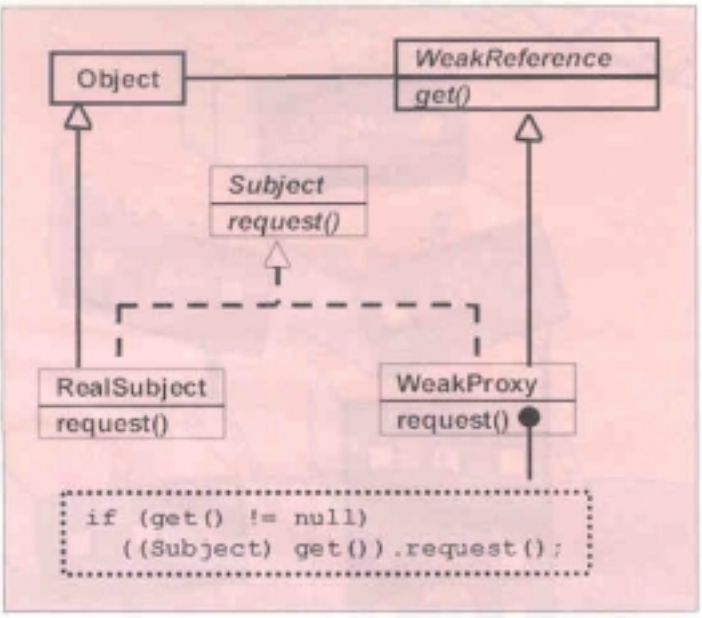

Abb. 3: WeakProxy

Wenn die Schnittstelle des Paketes die Navigation über die einzelnen Objekte ermöglicht (wie das bei Swing und DOM der Fall ist), bietet es sich an alle Objekte in einer einzigen stark zusammenhängenden Komponente zusammenzufassen (Im Beispiel also alle in einem Frame enthaltenen GUI-Elemente oder alle Nodes eines Documents). Neben der Möglichkeit zur Navigation über die Bestandteile der Komponente ist auch die Verwendung des Kompositionsmusters [Ga95] ein Indiz für diese Situation. Innerhalb der Komponente können Referenzen dann ohne Rücksicht auf die Speicherverwaltung, rein nach logischen Designkriterien, verwendet werden.

Das Paket java.sql bietet ein anderes Bild. Es ist durch eine Hierarchie von Objekten abnehmender Lebensdauer gekennzeichnet (Connection, Statement, ResultSet). Daher müssen Referenzen „von oben nach unten" (also z. B. von der Connection zum Statement) vermieden oder explizit beseitigt werden.

Eine andere Art von Paket, das häufig zu finden ist, dient nur der Gruppierung verschiedener alternativer oder kombinierbarer Implementierungen einer Schnittstelle. Die Pakete java.io und java.util fallen in diese Kategorie. Hier sind die innerhalb des Paketes entstehenden Objektstrukturen in der Regel sehr einfach und ihre Analyse ohne Mühe möglich.

Pakete stehen nicht alleine. Die Objekte verschiedener Pakete müssen zusammenarbeiten, um den Zweck der Applikation zu erreichen. Letzten Endes müssen dazu Objekte aus einem Paket Methoden von Objekten anderer Pakete aufrufen und dies erfordert eine Referenz auf das Objekt. Paketübergreifende Referenzen sind also unvermeidbar. Paketübergreifende stark zusammenhängende Komponenten sollten aber möglichst nicht vorkommen. Ganz besonders bei Paketen, die unabhängig voneinander entwickelt werden, kann es sonst nur zu leicht geschehen, dass übersehene Referenzen große Objektstrukturen länger als beabsichtigt am Leben erhalten.

Bereits allgemeine Designprinzipien legen es nahe, zyklische Abhängigkeiten zwischen Paketen zu vermeiden. Das dies nicht vor zyklischen Objektreferenzen schützt, hat das oben angeführte Beispiel gezeigt. Daher müssen paketübergreifende Referenzen stets mit Argwohn betrachtet werden. Gerade die paketinternen zusammenhängenden Komponenten bilden das Werkzeug, das diese Überprüfung praktikabel macht. So verrät jede Verwendung des Paketes org.w3c.dom sofort: „Hier wird ein ganzer DOM-Baum verwendet", ohne dass dazu die verwendete Klasse bekannt sein müsste.

Auflösung von Strukturen

Was nun, wenn eine aus Sicht der Speicherverwaltung unerwünschte Referenz zur Implementierung eine notwendigen Funktionalität unerlässlich ist? In manchen Fällen kann eine bestimmte Methode (close, clear, remove o. ä.) dazu verwendet werden, diese Referenz explizit auf null zu setzen. Dieses Verfahren bietet sich aber nur an, wenn eine solche Methode schon aus anderen Gründen erforderlich ist. Die allgemeingültige Lösung besteht in der Verwendung schwacher Referenzen (java.lang.ref.WeakReference).

Schwache Referenzen werden bei der Ermittlung der Erreichbarkeit eines Objektes im Sinne der zweiten Regel nicht berücksichtigt und erhalten ein Objekt nicht am Leben. Sie haben daher auch keinen Einfluss auf den Zusammenhang der Komponenten.

Tritt die Referenz innerhalb der Implementierung einer Klasse oder zwischen zwei eng verwandten Klassen auf, kann eine schwache Referenz oft direkt verwendet werden. Gerade bei globalen Beziehungen ist das oft nicht möglich, da der Erzeuger und der Besitzer der Referenz hierzu kooperieren müssen, diese sich aber in verschiedenen Paketen befinden. Hier hilft eine Variante des Proxy-Musters [Ga95], das Weak-Proxy. Abbildung 3 stellt eine mögliche Implementierung dieses Entwurfsmusters dar. Alternativ kann (und muss, wenn Subject keine Schnittstelle, sondern eine Klasse ist) statt der Ableitung von WeakReference eine solche schwache Referenz in der Implementierung des WeakProxy verwendet werden.

Zusammenfassung

Die Lebensdauer von Objekten ist durch ihre Erreichbarkeit bestimmt. Referenzen von langlebigen auf kurzlebige Objekte sollen vermieden werden. Stark zusammenhängende Komponenten bilden die Bausteine der Objektstrukturen. Objekte innerhalb einer solchen Komponente haben die gleiche Lebensdauer. Es ist darauf zu achten, dass dies mit der beabsichtigten Lebensdauer übereinstimmt. Referenzen zwischen großen, stark zusammenhängenden Komponenten sind besonders kritisch zu betrachten. Unerwünschte Beziehungen können mittels schwacher Referenzen aufgelöst werden.

Literatur

[Ga95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of reusable object-oriented Software, Addison-Wesley 1995

[Se95] J. Seligmann, St. Grarup, Incremental Mature Garbage Collection Using the Train Algorithm, in: ECOOP 1995, http://www.daimi.aau.dk/~beta/Papers/Train/train.html

[Wi96] P. Wilson, Uniprocessor Garbage Collection Techniques, University of Texas, Computer Science Department, 1996, ftp://ftp.cs.utexas.edu:/pub/garbage/bigsurv.ps

Dipl. Math. Ronald Dauster ist als Projektleiter und IT-Strategieberater bei der Object International Software GmbH tätig. Seit 11 Jahren liegt der Schwerpunkt seiner Tätigkeit auf objektorientierten Technologien und Client-Server-Architekturen. Als Berater unterstützt er mittlere und große Unternehmen bei der erfolgreichen Umsetzung moderner IT-Strategien.
E-Mail: dauster@oisoft.com