Entwicklung eines automatischen Verfahrens zur Auflösung statischer zyklischer Abhängigkeiten in Softwaresystemen*

Leo Savernik
Institut für Systemsoftware, Johannes Kepler Universität Linz
leo.savernik@jku.at

Abstract: Viele Softwaresysteme enthalten statische zyklische Abhängigkeiten, die das Verständnis, die Wartbarkeit und die Testbarkeit erschweren. Gesucht sind daher Methoden zur schonenden automatisierten Auflösung dieser Zyklen, um händische Eingriffe zu verringern.

1 Einführung

Als statische Abhängigkeiten zwischen zwei Artefakten betrachten wir die Aufruf-, Zugriffs-, Typ- und Vererbungsabhängigkeit. Bedingen diese Abhängigkeiten einen Pfad von einem Artefakt A über andere Artefakte zu A selbst, sprechen wir von einer statischen zyklischen Abhängigkeit.

Die Problematik statischer zyklischer Abhängigkeiten beschrieb Parnas [Par78] bereits in den 1970er Jahren als Phänomen, welches die Fertigstellung aller Komponenten eines Softwaresystems bedingte, bevor sich das Gesamtsystem oder Teile davon überhaupt übersetzen ließen. Selbst in heutigen modernen, objektorientierten Softwaresystemen sind zyklische Abhängigkeiten typischerweise mehr oder minder Zahl vertreten – zyklenlose Systeme stellen seltene Ausnahmen dar [MT06a].

Damit ergeben sich Nachteile beim Verständnis, bei der Wartung und bei der Testbarkeit dieser Systeme, da in zyklischer Abhängigkeit stehende Artefakte nicht mehr isoliert betrachtet werden können und somit kostenintensive Zusatzaufwände nach sich ziehen.

Was läge daher näher als die Bestrebung, zyklische Abhängigkeiten eines Softwaresystems soweit wie möglich aufzulösen? Eine rein manuelle Auflösung benötigt Zeit des Entwicklers, die diesem nicht gegeben ist. Daher verfolgen wir im Rahmen einer Dissertation die Erforschung einer weitgehend automatischen Auflösung von statischen zyklischen Abhängigkeiten.

Zu erörternde Fragen lauten wie folgt: Inwieweit lassen sich zyklische Abhängigkeiten eines Softwaresystems automatisiert auflösen? Wie brauchbar sind die Ergebnisse einer automatisierten Auflösung auf große Softwaresysteme angewandt?

*Diese Arbeit wurde von Comneon electronic technology GmbH & Co OHG, 4040 Linz, Österreich unterstützt.

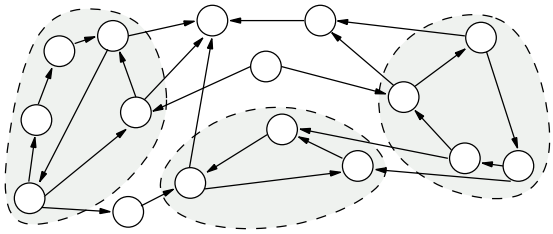

Abbildung 1: Graph mit hervorgehobenen Zyklengruppen

Eine automatisierte Auflösung zyklischer Abhängigkeiten in einem Softwaresystem zeigt nicht nur die vorhandenen Zyklen auf, sondern versucht selbständig eine annehmbare Auflösung dieser Abhängigkeiten vorzunehmen. Dies erleichtert nicht nur das Verständnis, die Wartung und die Testbarkeit eines Systems, sondern reduziert die auch Zeit, die ein Entwickler der händischen Zyklenauflösung widmen muss.

Kapitel 2 beschreibt den Stand der Technik zur Auffindung und Entfernung von Zyklen sowie zum Quelltextumbau. Kapitel 3 stellt die Lösungsansätze zur automatischen Zyklenauflösung dar. Zuletzt resümiert Kapitel 4 über den Inhalt dieses Artikels.

2 Stand der Technik

Bevor wir uns der Auflösung von zyklischen Abhängigkeiten widmen können, müssen wir sie zuerst finden. Hierzu liefert die Graphentheorie den Grundbegriff der *Zyklengruppe* (auch als *starke Komponente* [CH94, S. 256] bekannt). Eine Zyklengruppe eines Graphen ist ein Teilgraph, in dem jeder Knoten von jedem anderen Knoten dieses Teilgraphs erreichbar ist (siehe Abb. 1).

Zur Entdeckung von Zyklen existieren bereits eine Reihe von Werkzeugen wie ByeCycle[1], Classycle[2], JDepend[3], Jepends [MT06b] und JooJ [MT06c]. Ihre Unterstützung ist auf Java-Softwaresysteme beschränkt und sie zeigen dem Benutzer Zyklen auf, bieten jedoch keine weitergehende Hilfestellung zur deren Auflösung.

Sind die Zyklen erkannt, steht deren Auflösung zur Debatte, zum Beispiel mit Hilfe der *minimalen Kantenrückkopplungsmenge* [Ski97]. Die minimale Kantenrückkopplungsmenge eines Graphen bezeichnet jenen Untergraphen, dem gerade so viele Kanten entfernt wurden, dass er azyklisch wird. Der Algorithmus zur Ermittlung der Rückkopplungsmenge ist jedoch NP-schwierig und durch Heuristiken anzunähern.

In Softwaresystemen dürfen allerdings nicht beliebige Kanten entfernt werden, da sich dadurch die Semantik des Systems verändert. Abhilfe bietet hier das speziell auf die Softwareentwicklung zugeschnittene *Abhängigkeitsumkehrprinzip* [Mar96]. Gemäß diesem Prin-

[1] http://byecycle.sourceforge.net/ (Aug. 2006)
[2] http://classycle.sourceforge.net/ (Aug. 2006)
[3] http://www.clarkware.com/software/JDepend.html (Aug. 2006)

(a) Direkte Abhängigkeit (b) Umgekehrte Abhängigkeit

Abbildung 2: Beispiel für Zyklenauflösung durch Abhängigkeitsumkehr

zip soll ein abstrakteres Artefakt A nicht direkt auf ein konkreteres Artefakt B zugreifen, sondern auf eine Schnittstelle, die von B implementiert wird. Die Abhängigkeit ist damit umgekehrt und lässt sich zur Auflösung von Zyklen heranziehen (s. Abb. 2). Unter konsequenter Anwendung der Abhängigkeitsumkehr ließen sich so grundsätzlich alle Zyklen beseitigen.

3 Forschungsbeitrag

Während existierende Verfahren und Werkzeuge die *Erkennung* von Zyklen (ByeCycle, Classycle, JDepends, Jepends [MT06b]) als auch die *Vermeidung* von Zyklen (JooJ [MT06c]) ermöglichen, zielt diese Forschungsarbeit auf die weitgehend automatisierte, schonende *Auflösung* von zyklischen Abhängigkeiten ab. Das zu entwickelnde Verfahren erhebt Anspruch auf

- **weitgehende Auflösung**, die einen möglichst großen Teil der Zyklen automatisch auflöst, sodass die Notwendigkeit von Benutzereingriffen minimal, zumindest aber merklich unter denen einer manuellen Zyklenauflösung bleibt, sowie auf
- **schonende Auflösung**, die erstens das Verhalten des Softwaresystems nicht ändert und zweitens die öffentlichen Schnittstellen sowohl hinsichtlich der Binär- als auch Quellkompatibilität bewahrt.

Die Forschung soll zu einem Prototypen führen, der das zu entwickelnde Verfahren der Zyklenauflösung auf reale Softwaresysteme anwendet. Ein Durchlauf verläuft etwa wie folgt: Zunächst füttert der Benutzer den Prototypen mit notwendigen Umgebungsparametern, die das Verfahren nicht selbständig aus dem Softwaresystem herleiten kann (wie Pfade von Bibliotheken usw.). Danach analysiert der Prototyp sämtliche Artefakte des Systems und findet Zyklengruppen. Heuristiken bestimmen dann die jeweils geeigneten Auflösungsstrategien – immer unter Beachtung der Binär- und Quellkompatibilität.

Jeder Schritt wird nachvollziehbar protokolliert, sodass sich dem Entwickler nicht nur Art und Weise der Transformationen erschließt, sondern vom ihm auch als Analyse der im Ursprungssystem enthaltenen zyklischen Abhängigkeiten verwendet werden kann.

Das Verfahren strebt keine Allgemeingültigkeit in dem Sinne an, dass es alle Zyklen aufzulösen vermag (wie durch öffentliche Schnittstellen induzierte Zyklen), noch dass es die effizienteste Art der Zyklenauflösung für ein gegebenes Softwaresystem findet. Das Verfahren stellt bereits dann einen Fortschritt dar, wenn es Änderungen so vornimmt, wie sie auch ein erfahrener Softwareentwickler durchgeführt hätte.

Bislang wurde die Extraktion von Informationen aus C++- und Java-Softwaresystemen und deren Überführung in ein geeignetes Modell für die Weiterverarbeitung implementiert.

4 Zusammenfassung

In diesem Artikel beschrieben wir einen Ansatz zur automatischen Auflösung von zyklischen Abhängigkeiten in Softwaresystemen. Als Endergebnis der Forschung ist ein Verfahren angedacht, das ein Softwaresystem so umzubauen vermag, dass eine große Anzahl seiner Zyklen aufgelöst wurde, ohne das Verhalten des Systems oder dessen Binär- und Quellkompatibilität zu beeinträchtigen.

Zur Person

Leo Savernik ist Dissertant am Institut für Systemsoftware der Johannes Kepler Universität Linz und betreibt die Auflösung zyklischer Abhängigkeiten im Rahmen eines Forschungsprojektes der Firma Comneon zur Verbesserung der Softwarequalität.

Literatur

[CH94] J. Clark und D. A. Holton. *Graphentheorie*. Spektrum Akademischer Verlag, 1994.

[Mar96] R. C. Martin. The Dependency Inversion Principle. *C++ Report*, 1996.

[MT06a] H. Melton und E. Tempero. Empirical Study of Cycles among Classes in Java. Bericht, Department of Computer Science, Universität Auckland, 2006.

[MT06b] H. Melton und E. Tempero. Identifying Refactoring Opportunities by Identifying Dependency Cycles. In *Proc. 29th Australasian Computer Science Conference*, 2006.

[MT06c] H. Melton und E. Tempero. JooJ: Real-time Support for Avoiding Cyclic Dependencies. Bericht, Department of Computer Science, Universität Auckland, 2006.

[Par78] D. L. Parnas. Designing software for ease of extension and contraction. In *Proceedings of the 3rd international conference on Software engineering*, 1978.

[Ski97] S. S. Skiena. *The Algorithm Design Manual*. Springer-Verlag, 1997.