Anatomie zyklischer Abhängigkeiten in Softwaresystemen

Leo Savernik

Technischer Bericht
1 / 2007

Johannes Kepler Universität Linz
Institut für Systemsoftware
Altenbergerstraße 69, 4040 Linz

Juli 2007

Anatomie zyklischer Abhängigkeiten in Softwaresystemen

Leo Savernik

11. April 2007
Zuletzt überarbeitet am 24. Juli 2007

Zusammenfassung

In diesem Artikel untersuchen wir die zyklischen Abhängigkeiten zwischen Artefakten in Java-Applikation verschiedener Größe, teilen die gefundenen Zyklengruppen nach Größe und Struktur ein und beschreiben beobachtete Merkmale. Weiters wird der Einfluss von objektorientierten Entwurfsmustern auf die Zyklenbildung untersucht, inwiefern und welche Zyklengruppen zur Entstehung zyklischer Abhängigkeiten beitragen. Wir entwickeln ferner eine einfache, algorithmisch umsetzbare Richtlinie, die eine Einteilung der Zyklengruppen hinsichtlich ihrer »Güte« ermöglicht.

Inhaltsverzeichnis

1 Einführung

Softwaresysteme sind im Laufe ihres Lebens einer stetigen Anpassung an Erfordernisse der Wirklichkeit unterworfen. Sich ändernde Anforderungen erfordern die Entwicklung neuer Funktionalität, aufgetretene Fehler deren Reparatur.

Grundsätzlich sollte eine Änderung an einem Softwaresystem keine oder nur wohlbegründete kleine Einbußen an Qualitätsmerkmalen wie Erweiterbarkeit, Wartbarkeit und Testbarkeit ergeben. In der Praxis führt die stetige Anpassung zu einer zunehmenden Degeneration des Ursprungsentwurfs [23, 25] und zu einer damit einhergehenden Verschlechterung der Qualitätsmerkmale.

Als Indikator und Ursache dieser Degeneration zählt die Existenz zyklischer Abhängigkeiten [8, 19], die sich auf das Programmverständnis sowie auf Erweiterbarkeit, Wartbarkeit und Testbarkeit negativ auswirken [20]. Ein erstrebenswertes Ziel zur Verbesserung von Softwaresystemen liegt daher in der Eliminierung oder zumindest der Verringerung von zyklischen Abhängigkeiten (kurz: Zyklen).

Bevor wir jedoch die Auflösung von Zyklen in Angriff nehmen können, müssen wir erst die Anatomie zyklischer Abhängigkeiten kennen. Aus diesem Grunde wurde eine Untersuchung von Softwaresystemen verschiedener Größe durchgeführt, Kategorien von Zyklengruppen identifiziert und eine einfache Metrik zur Beurteilung dieser entwickelt. Dieses derart ermittelte Wissen soll später in Strategien für die Auflösung von Zyklen münden.

Zur schnellen Feststellung jener »guten« Zyklengruppen, die keiner Aufösung bedürfen, entwickeln wir aufgrund der in dieser Untersuchung gewonnenen Erkenntnisse eine einfache, auch als Algorithmus implementierbare Richtlinie.

1.1 Kapitelübersicht

In Kapitel 3 werden Vorarbeiten und ähnliche Untersuchungen aufgeführt und zu dieser Untersuchung in Beziehung gesetzt.

Kapitel 2 enthält detaillierte Definitionen der in diesem Artikel verwendeten Begriffe rund um die Analyse zyklischer Abhängigkeiten.

Kapitel 4 führt verschiedene Erkenntnisse und Verfahren auf, die zur Zyklenerkennung und auch zur Zyklenmessung verwendet wurden.

Kapitel 5 beschreibt den Modus der Untersuchung, die getroffenen Annahmen sowie die Durchführung.

Kapitel 6 interpretiert und analysiert die aus der Untersuchung gewonnenen Ergebnisse und nimmt die Einteilung der gefundenen zyklischen Abhängigkeiten sowie die Auswirkung der Entwurfsmuster auf die Zyklenbildung vor.

Zuletzt fasst Kapitel 7 das in diesem Beitrag Gesagte zusammen und liefert einen Ausblick auf weitergehende Forschungen.

2 Definitionen

Zuvor erfolgen noch einige Begriffsdefinitionen, auf die wir uns im gesamten Dokument beziehen (so nicht explizit als anders aufzufassen aufgeführt).

2.1 Begriffe

Artefakt Ein *Artefakt* ist ein physisches oder abstrahiertes Element eines Softwaresystems wie zum Beispiel ein Symbol, eine Funktion, eine Klasse, ein Paket, ein Subsystem oder eine Architekturschicht.

Primärartefakt Ein *Primärartefakt* ist ein Artefakt feinster Granularität, das in der Untersuchung gerade noch von primärem Interesse ist. Alle Artefakte feinerer Granularität werden implizit in Primärartefakte aggregiert betrachtet, ebenso die Abhängigkeiten zwischen diesen. Die feineren Artefakte konstituieren zwar die Einzigartigkeit des Primärartefakts, sind aber jeweils *für sich alleine* genommen für die Aussage und den Fortschritt der Untersuchung nicht bedeutend genug.

Für diese Untersuchung stellt die *Klasse* das Primärartefakt dar.

Die folgenden zwei Artefaktdefinitionen sind an [6] angelehnt.

Verhaltensartefakt Ein *Verhaltensartefakt* ist ein Artefakt, das unmittelbar ausführbaren Code enthält und somit das Verhalten des Programms bestimmt. Verhaltensartefakte werden je nach Programmiersprache Prozeduren, Funktionen oder Methoden genannt.

Strukturartefakt Ein *Strukturartefakt* ist ein Artefakt, das unmittelbar Speicherplatz belegt und somit zur Repräsentation des Programmzustands beiträgt. Strukturartefakte sind globale und lokale Variablen, Formalparameter, Klassenvariablen und Attribute (auch Instanzvariablen oder Felder genannt).

Zunächst definieren wir den Grundbegriff der Abhängigkeit zwischen zwei Artefakten.

Abhängigkeit Eine *Abhängigkeit* zwischen zwei Artefakten liegt vor, wenn Artefakt A Artefakt B zu seiner korrekten Funktionsweise benötigt.

Eine Abhängigkeit ist *mittelbar* oder *transitiv*, wenn ein Artefakt A keine direkte Abhängigkeit zu einem Artefakt B unterhält, sondern indirekt über ein drittes Artefakt. Beispielsweise hängt in Abbildung 1 M von L ab und C von M. C hängt damit transitiv auch von L ab.

Zyklische Abhängigkeit Eine Abhängigkeit ist *zyklisch*, wenn zwischen zwei Artefakten A und B sowohl A von B (transitiv) abhängt als auch B von A.

Ein Beispiel zyklischer Abhängigkeiten zeigt Abbildung 1 mit den Knoten I, J, K und L.

Zyklengruppe Eine *Zyklengruppe* ist eine in einem Graph größtmögliche Menge von Artefakten, in der zwischen jedem Paar von Artefakten eine zyklische Abhängigkeit besteht.

Der Graph in Abbildung 1 enthält drei Zyklengruppen, die eingerahmt visualisiert sind.

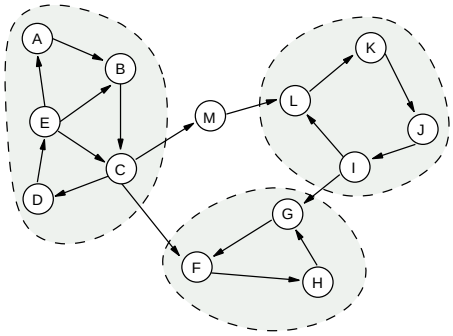


Abbildung 1: Knoten, Kanten und Zyklengruppen

```
1  class A extends B { // Vererbungsbez. zu B ✪
2    int a;          // Attribut
3    B b;            // Attribut
4    void m() {
5      a = b.a;      // Lesezugriff auf b
6                    // Lesezugriff auf B.a ✪
7                    // Schreibzugriff auf a
8      b.a = a;      // Lesezugriff auf a
9                    // Lesezugriff auf b
10                   // Schreibzugriff auf B.a ✪
11     b.f();        // Lesezugriff auf b
12                   // Aufrufabhängigkeit zu B.f ✪
13     m();          // Aufrufabhängigkeit zu A.m
14   }
15 }
```

(a) Klasse A

```
1  class B {
2
3    int a;
4
5    void f() {
6      m();          // Aufrufabhängigkeit zu A.m
7
8    }
9
10   void m() {
11     a = 0;        // Schreibzugriff auf a
12
13   }
14
15 }
```

(b) Klasse B

Abbildung 2: Beispiele für Abhängigkeitstypen

2.2 Abhängigkeitstypen

Hier erfolgt eine genaue Beschreibung aller Abhängigkeitstypen zwischen Artefakten, die in diesem Artikel Erwähnung finden.

Die nächsten drei Typen stellen grundlegende Abhängigkeitstypen dar, die sich direkt aus der Betrachtung des Quelltexts ergeben. Grundlegende Abhängigkeitstypen bestehen immer nur zwischen zwei Verhaltensartefakten oder zwischen Verhaltens- und Strukturartefakten oder zwischen Klassen. Abbildung 2 enthält Beispiele zu den Abhängigkeiten, auf die die einzelnen Definitionen verweisen. Unterlegte Zeilen weisen auf primärartefaktübergreifende Abhängigkeiten hin.

Aufrufabhängigkeit Ein Verhaltensartefakt A besitzt eine Aufrufabhängigkeit zu Verhaltensartefakt B, wenn der Quelltext von A einen Aufrufbefehl von B enthält. Nur direkt dem Quelltext entnehmbare Aufrufe zählen als Aufrufabhängigkeit. Aufrufe indirekter Art, wie zum Beispiel durch Polymorphismus verursacht, zählen nicht als Aufrufabhängigkeit.

In Abbildung 2(a) enthalten die Zeilen 11 und 13, in Abbildung 2(b) die Zeile 6 Aufrufabhängigkeiten. Dem Verhaltensartefakt A entspricht hier die Methode A.m().

Zugriffsabhängigkeit Ein Verhaltensartefakt A besitzt eine Zugriffsabhängigkeit zu Strukturartefakt

B, wenn der Quelltext von A B ausliest oder beschreibt. Die Zugriffsabhängigkeit ist eine Vereinigung von Lesezugriff und Schreibzugriff.

Abbildung 2(a) enthält Beispiele für Zugriffsabhängigkeiten in den Zeilen 5 bis 11. Streng genommen besitzt sogar Zeile 13 eine Zugriffsabhängigkeit auf die Klasse selbst (über den impliziten `this`-Zeiger). Diese implizit erzeugten Abhängigkeiten können als für den Betrieb notwendige Infrastruktur betrachtet werden, die der Entwickler nicht direkt beeinflussen kann, womit eine Berücksichtigung nicht angebracht ist.

Vererbungsabhängigkeit Eine Klasse A besitzt eine Vererbungsabhängigkeit zu einer Klasse B, wenn A von B abgeleitet ist (Beispiel siehe Abb. 2, jew. Zl. 1).

Im Artikel erwähnte aber in der Untersuchung nicht berücksichtigte Abhängigkeitstypen folgen hier.

Namensabhängigkeit Ein Artefakt A besitzt eine Namensabhängigkeit zu einem Artefakt B, wenn es sich im Quelltext auf den Namen des Artefakts B bezieht.

Normalerweise geht eine Namensabhängigkeit immer mit einer der obigen Abhängigkeitstypen einher, weswegen eine reine Namensabhängigkeit nicht berücksichtigt werden muss.

Ein Beispiel für eine reine Namensabhängigkeit ist der Java-Befehl `import java.io.InputStream`. Da auf `InputStream` typischerweise mittels Zugriffs-, Aufruf- oder Vererbungsabhängigkeit operiert wird, liefert die Namensabhängigkeit keine zusätzliche Information. Findet keinerlei Operation auf `InputStream` statt, so ist die Namensabhängigkeit überflüssig und kann keinen Einfluss mehr auf das Programmverständnis ausüben.

2.3 Einschränkungen

Für die Analyse zyklischer Abhängigkeiten interessieren uns ausschließlich die Abhängigkeiten *zwischen Primärartefakten*. Diese Abhängigkeiten (in Abbildung 2 mit ✪ hervorgehoben) stellen schließlich Verbindungen zwischen den verschiedenen Systemteilen her und verhelfen dem betrachteten Softwaresystem zu seiner Gesamtfunktion.

Daher fallen sämtliche Abhängigkeiten innerhalb eines Primärartefakts aus der Untersuchung heraus.

3 Verwandte Arbeiten

Die verwandten Arbeiten teilen sich in zwei Gebiete auf, nämlich in die direkt auf zyklische Abhängigkeiten und die auf Klassifizierung von Softwareartefakten bezogenen Publikationen.

3.1 Zyklenstudien

Melton und Tempero [20] leisteten mit ihrer umfangreichen Studie über Zyklen in einer repräsentativen Stichprobe von Java-Applikationen eine elementare Vorarbeit zu der im vorliegenden Artikel beschriebenen Untersuchung. In dieser Studie untersuchten sie 78 Java-Applikationen und fanden heraus, dass 45% aller Applikationen mindestens eine Zyklengruppe mit 100 Klassen, 10% sogar mindestens eine mit 1000 Klassen enthielten.

Weiters berechneten Melton und Tempero für jede untersuchte Applikation mittels minimaler Kantenrückkopplungsmenge [27] den theoretischen Umbauaufwand, der zur Auflösung der Zyklen aufzubringen wäre.

Unsere Untersuchungen bestätigen die quantitativen Messungen weitgehend, unterscheiden sich jedoch insofern, dass wir rein syntaktische Namensabhängigkeiten nicht als zyklenerzeugende Abhängigkeiten betrachteten. Dadurch fallen bei äquivalenten Anwendungen die Zyklengruppen kleiner aus als bei [20].

In [12] werden zyklische Abhängigkeiten auf Java-Paketebene mittels einer eigenen Metrik untersucht, welche gewünschte Abhängigkeiten zu den gesamten Abhängigkeiten in Proportion setzt. Zusätzlich präsentiert der Artikel eine »intelligente« Schichtung. Dabei werden in einer Paketzyklengruppe sämtliche »nicht wünschenswerten« Abhängigkeiten ermittelt und bei der Schichtzuordnung ignoriert. Details über die Funktionsweise der »intelligenten« Schichtung sind nicht verfügbar.

Im Gegensatz zu unserer Untersuchung wurden keine Analysen von Form und Häufigkeit unternommen.

Mit stärkerem Hinblick auf Testbarkeit beschäftigen sich [11, 3] mit der Entfernung von Kanten aus Zyklengruppen, sodass eine Reihenfolge zum Testen der Klassen aufgestellt wird. Die Artikel beschreiben Verfahren zur Aufstellung einer Testreihenfolge, in der die Anzahl der zu entfernenden Kanten im Vergleich zu den existierenden Verfahren signifikant verringert ist.

Obwohl unsere Untersuchung in Hinblick auf die letztliche Auflösung zyklischer Abhängigkeiten geführt wird, beschränkt sie sich zum gegenwärtigen Zeitpunkt auf die Analyse des Ist-Zustands.

Von der Seite der architekturellen Schichtung betrachten Sarkar et al. [26] die Einführung von zyklischen Abhängigkeiten in Softwaresystemen über Schichtgrenzen hinweg. Typische Systeme verletzen die Schichtung durch Einführung zyklischen Abhängigkeiten zwischen Schichten und erschweren damit die Wartbarkeit, Veränderbarkeit, Wiederverwendbarkeit und Portabilität. Der Artikel stellt Metriken vor, die verschiedene Arten von korrekten und fehlerhaften schichtübergreifenden Zugriffen repräsentieren.

Im Gegensatz zu unserer Zyklenuntersuchung stellen die in [26] untersuchten Artefakte Architekturebenen dar, die sich nicht automatisch aus dem Quelltext extrahieren lassen. Eine manuelle vorherige Bestimmung der Ebenen ist daher Voraussetzung.

Eine der unsrigen ähnliche Analyse führten Brinkley und Harman [2] durch, indem zyklische Abhängigkeiten von Befehlen in Programmschnitten zu sogenannten *Abhängigkeitshaufen* zusammengefasst wurden. Dabei deckte die Untersuchung in 80% der untersuchten Programme die Existenz großer Abhängigkeitshaufen auf, die mehr als 10% des jeweiligen Programms umfassten.

3.2 Klassifizierung von Softwareartefakten

Die Bewertung der Softwarekomplexität erfolgt bei Lanza [15] mittels Evaluierung der Revisionsgeschichte eines Softwaresystems und der Klassifizierung entdeckter Muster auf Klassenebene durch Metaphern aus der Astronomie. Die Klassen werden über polymetrische Sichten [16], das heißt als Kästchen deren Höhe, Breite und Farbe jeweils den Wert einer Metrik widerspiegeln, in einer Matrix dargestellt und aus den sich ergebenden Zeilen Klassifikationen abgeleitet.

So beschreibt beispielsweise eine »Supernova« eine kleine Klasse, deren Größe innerhalb weniger Revisionen exorbitant ansteigt, ein »roter Riese« eine immergroße Klasse über viele Revisionen hinweg und ein »weißer Zwerg« eine über die Revisionen hinweg immer weiter in der Größe abnehmende Klasse.

Lungu et al. [17] behandeln Artefakte auf Paketebene, indem Pakete weitgehend automatisch in architekturrelevante und implementierungstechnische unterteilt werden. Die architekturrelevanten Pakete verdienen nähere Betrachtung (stellen also Angelpunkte des Programmverständnisses dar), implementierungstechnische Pakete können ausgeblendet bleiben.

Ein Paket ist entweder »still« (keine Abhängigkeiten), »Konsument« (ausgehende Abhängigkeiten), »Anbieter« (eingehende Abhängigkeiten) oder »hybrid« (beides). Aufgrund dieser Einteilung wurden Muster in der Pakethierarchie identifiziert, die typische Verteilungen von architekturrelevanten zu implementierungstechnischen Paketen widerspiegeln. Als Beispiele seien die Muster »Eisberg«, in dem lediglich das oberste Paket als »Anbieter« fungiert und »Schornstein«, in dem ausschließlich die Blätter der Hierarchie architekturrelevant sind, genannt.

4 Beschreibung der Verfahren und Algorithmen

Zur Ermittlung und Untersuchung zyklischer Abhängigkeiten sowie deren Beurteilung wurden eine Reihe an Verfahren eingesetzt. Dieses Kapitel diskutiert einerseits die Feststellung zyklischer Abhängigkeiten, andererseits Metriken zur Bewertung eben dieser.

4.1 Zyklenfindung

Grundsätzlich betrachten wir ein Softwaresystem als einen gerichteten Graphen, dessen Knoten Artefakte wie Klassen, Methoden und globale Variablen darstellen und dessen Kanten diese Artefakte verbinden. In einem zyklenfreien System lässt sich jedem Knoten ein Wert zuweisen, der die maximale Länge eines Pfades zu einem Knoten unterster Ebene (das ist ein Knoten ohne ausgehende Verbindungen) darstellt [7, 12].

In einem Zyklus befindliche Knoten lassen sich nicht eindeutig einer Ebene zuteilen. Darüber hinaus können sich mehrere Zyklen so überlappen, sodass sich ein Untergraph ergibt, in dem jeder Knoten von jedem anderen Knoten erreichbar ist. Abbildung 1 auf Seite 4 zeigt beispielsweise den Zyklus $\overline{\text{ABCDE}}$, der von den Zyklen $\overline{\text{BCDE}}$ und $\overline{\text{CDE}}$ überlappt wird. Die Graphentheorie nennt einen größtmöglichen derartigen Untergraphen starke Komponente [4, S. 256], aus welchem wir den Begriff *Zyklengruppe* ableiten:

Eine *Zyklengruppe* ist eine starke Komponente mit mehr als einem Knoten.

Der Graph starker Komponenten (in dem jeder Knoten eine starke Komponente repräsentiert) ist azyklisch und erlaubt eine Zuweisung der Knoten zu Ebenen. Die in einer starken Komponente zusammengefassten Knoten gehören dabei alle derselben Ebene an.

Zur Auffindung von starken Komponenten in einem Graphen existieren eine Reihe Algorithmen, wobei derjenige von Tarjan lineares Laufzeitverhalten aufweist. Für die Untersuchungen setzen wir den auf Tarjan basierenden, leicht verbesserten Algorithmus NEWSCC1 von Nuutila-Soisalon-Soininen [24] ein.

4.2 Zyklenmessung

Obwohl für jede Zyklengruppe ein erfahrener Softwareentwickler ein subjektives Urteil über ihren Beitrag zum Verständnis, ihre Wartbarkeit und Testbarkeit fällen kann, führt eine alleinig subjektive Beurteilung zu folgenden Problemen.

- Ein subjektives Urteil ist personenabhängig.

- Ein subjektives Urteil ist nicht vergleichbar.
- Ein subjektives Urteil lässt sich nicht zuverlässig reproduzieren.

Wir wollen daher eine Richtlinie entwickeln, die eine Beurteilung der »Güte« einer Zyklengruppe ohne die obigen Probleme ermöglicht. Anspruch auf vollständige Objektivität wird nicht erhoben, da ein mathematischer Beweis eines Optimums aufgrund der letztendlichen Wahrnehmung und Beurteilung durch einen Menschen nicht zielführend ist.

Wir stellen neben der Vermeidung obiger Probleme zusätzliche Bedingungen an die für die Durchführung der Richtlinie zulässigen Messungen, nämlich dass

- die Messungen vollständig auf dem vorliegenden Modell des Softwaresystems durchführbar sind und
- sich die Messungen ohne menschliches Zutun vornehmen lassen.

Für die Richtlinie gibt es drei Kategorien von Zyklengruppen, zu deren Bestimmung die Anwendung Richtlinie betragen soll.

- *Harmlose* Zyklengruppen sind jene, deren Auflösung mutmaßlich mehr Aufwand verursacht, als die Zyklengruppe unangetastet zu belassen.
- *Beabsichtigte* Zyklengruppen wurden als vorsätzliche Entwurfs- oder Implementierungsentscheidungen eingebracht. Typischerweise können solche Zyklengruppen nicht einfach aufgelöst werden und sind daher im Rahmen der Richtlinie wie harmlose Zyklengruppen zu werten.
- *Verbesserungswürdige* Zyklengruppen versprechen eine Verringerung der Komplexität hinsichtlich Wartbarkeit und Verständnis, wenn deren Auflösung erfolgt.

5 Untersuchungsanordnung und -durchführung

Dieser Abschnitt beschreibt die Werkzeuge, Anordnung und Durchführung der Untersuchung der betrachteten Softwaresysteme.

5.1 Werkzeug zur Zyklenerkennung

Zum Zwecke unserer Untersuchung wurde ein prototypisches Java-Werkzeug zur Erkennung von Zyklen entwickelt. Es analysiert class-Dateien im Bytecode und verwendet dazu eine Erweiterung des in [6] beschriebenen FAMIX-Modells.

Das Werkzeug schreibt sämtliche Schritte und Analyseergebnisse in ein detailliertes Protokoll im HTML-Format, das sich mit einem gewöhnlichen Webbrowser betrachten lässt.

5.2 Anordnung

Die Untersuchung der Softwaresysteme fand unter der nachfolgend beschriebenen Anordnung statt.

Am Anfang erfolgte die Auswahl einer Reihe von Java-Applikationen. Die Applikationen wurden aus der verfügbaren Menge freier Java-Software aus den Bereichen Softwareentwicklung (find-bugs,

Eclipse, ArgoUML, CVSGrab), Serverdienste (Tomcat) und Datenbanken (Kowari) gewählt und sowohl große als auch kleine Applikationen in der Auswahl berücksichtigt. Zusätzlich wurden die sieben im Performanz-Benchmark SpecJVM98 enthaltenen Java-Applikationen und die Applikation des SPECjbb2005-Benchmarks zwecks Vergleichbarkeit einbezogen.

Als Testgerät stand der Entwicklerrechner zur Verfügung. Da keine zeitlichen Messungen vorgenommen werden mussten, war eine sich während des Testlaufs ändernde Systemlast ohne Belang. Weiters arbeitet der zum Einsatz gelangte Zyklenfindungsalgorithmus [24] deterministisch und idempotent, sodass bereits ein einziger Analyselauf zum Auffinden aller zyklischen Abhängigkeiten genügt.

5.3 Durchführung

Die Untersuchung erfolgte in folgenden Schritten:

1. Für jede untersuchte Java-Applikation wird zuerst der Quelltext, so vorhanden, in Bytecode inklusive voller Variablen- und Zeileninformationen übersetzt.
2. Anschließend werden die Zyklengruppen mit dem bereits erwähnten Werkzeug im Java-Bytecode identifiziert und in ein Protokoll geschrieben. Das Protokoll wird gesichert, sodass es für nachträgliche Überprüfungen und zukünftige Analysen zur Verfügung steht.
3. Zuletzt erfolgt die manuelle Auswertung der Zyklengruppen mit der Erkennung von Mustern, sowie eine skriptgesteuerte Berechnung der Statistiken.

5.4 Erläuterungen

Als zyklenerzeugend wurden in jedem Softwaresystem die elementaren Abhängigkeitstypen

- primärartefaktübergreifende Aufrufe, wobei Polymorphismus keine Berücksichtigung findet,
- primärartefaktübergreifende Lese- und Schreibzugriffe sowie
- Vererbungs- und Implementierungsabhängigkeiten zwischen Klassen beziehungsweise Schnittstellen und Klassen

in Betracht gezogen und die Verbindungen auf Ebene von Primärartefakten aggregiert.

Dies bedeutet, dass beispielsweise die elementaren Aufrufe von A.f()→B.c() und A.g()→B.d() bei der Zyklenfindung als eine einzige Kante zwischen den Klassen A und B repräsentiert wurden.

Das vom Werkzeug erzeugte Protokoll schlüsselt sämtliche aggregierten Kanten wieder in ihrem vollen Detailreichtum auf, sodass eine beliebig genaue manuelle Analyse erfolgen kann.

6 Auswertung

Zur Erforschung der Zyklenanatomie wurden 259 Zyklengruppen aus 14 Softwaresystemen evaluiert. Tabelle 1 zeigt die Eckdaten der ausgewerteten Systeme. Aus der Tabelle lässt sich ersehen, dass von jedem untersuchten Softwaresystem ein bis zwei Drittel der Artefakte in zyklischen Abhängigkeiten gebunden sind.

Applikation	Gr	ZG	Max	Min	Med	Sch	AZG (%)	ASZG (%/%)
find-bugs 1.1.3	783	34	138	2	2	4	260 (33)	188 (24/72)
Tomcat 5.5.20	773	50	95	2	3	4	295 (38)	166 (21/56)
Eclipse 3.2.1	1805	86	244	2	2	9	582 (32)	379 (20/65)
ArgoUML 0.23.5	1403	10	860	2	2	2	899 (64)	869 (61/96)
Kowari 1.2	1909	58	64	2	2	8	273 (14)	155 (8/56)
CVSGrab 2.2.2	73	4	26	2	2	2	38 (52)	35 (47/92)
compress	9	1	3	3	3	0	3 (33)	0 (0/0)
JESS 3.2	151	2	27	2	2	1	29 (19)	27 (17/93)
db	3	0	–	–	–	–	–	–
javac 1.0.2	176	9	43	2	2	2	64 (36)	48 (27/75)
mpegaudio	55	1	15	15	15	1	15 (27)	15 (27/100)
raytrace	25	2	9	2	2	1	11 (44)	9 (36/81)
Jack 0.1	56	0	–	–	–	–	–	–
SPECjbb2005	61	2	26	2	2	1	28 (45)	26 (42/92)
Σ	7282	259	860	2	2	35	2497 (34)	1917 (26/76)

Tabelle 1: Übersicht über die Auswertung.
Gr Anz. untersuchter Primärartefakte (Klassen), **ZG** Anz. Zyklengr., **Max** größte Zyklengr. in Klassen, **Min** kleinste Zyklengr. in Klassen, **Med** Median Zyklengr.größe, **Sch** Anz. als verbesserungswürdig identifiz. Zyklengr., **AZG (%)** Anz. Artefakte in Zyklengruppen (in %), **ASZG (%/%)** Anz. Artefakte in verbesserungswürdigen Zyklengruppen (in % rel. zu Gr/in % rel. zu AZG)

6.1 Struktur der Softwaresysteme

Die Inspektion der Untersuchungsprotokolle zeigte, dass Softwaresysteme mit geringerem Zyklenanteil (bis zu etwa gut einem Drittel) auch tendenziell weniger stark verzykelte Abhängigkeiten unter höherschichtigen Artefakten aufwiesen als solche mit höherem Anteil. Auch machten diese Systeme erkennbar von Prinzipien objektorientierten Entwurfs Gebrauch wie zum Beispiel von Entwurfsmustern [9], wie aus der Namensgebung von Artefakten eruierbar war.

Hierzu wurden die Softwaresysteme jeweils auf Paketebene betrachtet und als Indikator »strukturierterer« Softwaresysteme die Abwesenheit oder geringere Anzahl zyklischer Paketabhängigkeiten gewertet. Dies diente allerdings nur zur Grobeinschätzung des Systementwurfs, weshalb keine weiteren detaillierten Auswertung erfolgten.

Stark verzykelte Systeme (Beispiel: ArgoUML) zeichneten sich in der Regel durch wenige, aber dafür riesige Zyklengruppen aus. Visualisierungen auf höherer Ebene zeigten weiter, dass sich die Verzyklung über viele Pakete hinweg erstreckt und das System zu einem dichten Abhängigkeitsgeflecht zusammenbindet.

Nur zwei der untersuchten Softwaresysteme wiesen überhaupt keine Zyklengruppen auf. Hierbei handelte es sich um kleine bis kleinste Benchmarkprogramme, die in ihrer Art keine direkten Rückschlüsse auf Softwareentwicklung im Großen zulassen.

6.2 Anatomie der Zyklengruppen

Während der Untersuchung der Softwaresysteme wurde versucht, die angetroffenen Zyklengruppen in Schemata zu fassen, damit Rezepte für eine zukünftige automatische Bewertung entwickelt werden können. Die Einteilung von Zyklengruppen erfolgt einerseits nach deren Größe (das heißt nach der

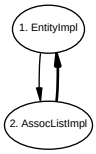
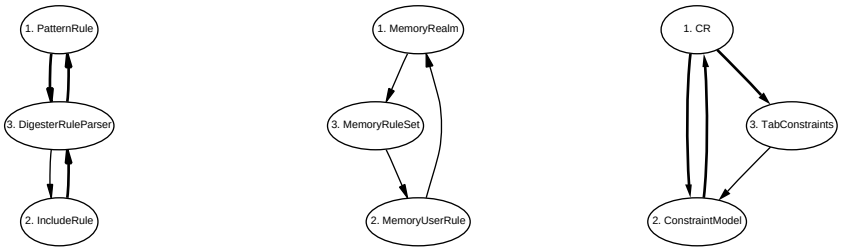


Abbildung 3: Zyklengruppe aus zwei Elementen



(a) Stern (b) Kreis (c) Andere

Abbildung 4: Zyklengruppen aus drei Elementen

Anzahl der darin enthaltenen Artefakte), andererseits nach deren Form und Eigenschaften.

Zweierzyklen Die einfachste Zyklengruppe stellt ein Zyklus aus zwei Elementen dar. Topologisch gesehen besitzt der Zweierzyklus lediglich eine einzige Form, wie Abbildung 3 zeigt. Von allen Zyklengruppen stellen die Zweierzyklen mit 56% den größten Anteil.

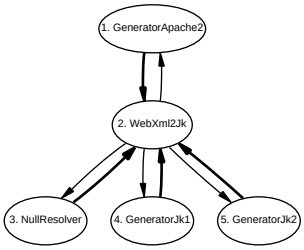
Softwaretechnisch birgt ein Zweierzyklus noch einiges mehr an Informationen. Die Untersuchung zeigte, dass von allen Zweierzyklen 92% durch eine Haupt-Neben-Beziehung verursacht werden. Hierbei lässt sich ein Artefakt eindeutig als Hauptartefakt identifizieren, welches sich des anderen Artefakts zur Erfüllung seiner Aufgabe bedient.

Ein Indikator für eine Haupt-Neben-Beziehung ist (bei Artefaktgranularität auf Klassenebene) die Instantiierung der Nebenklasse durch die Hauptklasse. Nebenklassen liegen darüber hinaus in den von uns untersuchten Java-Applikationen zu 41% als innere Klasse der Hauptklasse vor.

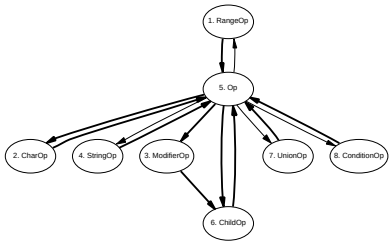
Die wenigen nicht den Haupt-Neben-Beziehungen entsprechenden Zweierzyklen stellen in der Regel eine *intrinsische gegenseitige Abhängigkeit* [14, Kap. 5.1.3] dar, in der zwei Artefakte Kenntnis voneinander besitzen müssen, ohne jedoch eine Unterordnung des einen unter das andere bestimmen zu können. Ein Beispiel hierfür ist das auch in den untersuchten Anwendungen aufgetretene Knoten-Kanten-Problem. Hier benötigt ein *Knoten* Kenntnis über die mit ihm verbundenen Kanten, sowie eine *Kante* die durch sie verbundenen Knoten kennen muss. Beide Artefakte *Knoten* und *Kante* sind jedoch gleichberechtigt – eine Einteilung in ein Haupt- und ein Nebenartefakt kann nicht zielsicher erfolgen.

Dreierzyklen Zyklengruppen aus drei Elementen stellen mit 18% die zweithäufigste Art aller Zyklengruppen dar. Die häufigsten Topologien zeigt Abbildung 4.

Der *Stern*, in dem zwei Artefakte jeweils auf das dritte zugreifen, aber nicht gegenseitig, das dritte jedoch auf beide anderen zugreift, macht 39% aller Dreierzyklen aus. Weiters besteht die Gruppe der Dreierzyklen zu 6% aus echten Kreisen im Sinne der Graphentheorie (wie aus Abb. 4(b) ersichtlich).



(a) Stern der Größe fünf



(b) Entarteter Stern

Abbildung 5: Sterne

Der Rest (54%) entfällt auf Dreierzyklen, die sich aus Kombinationen obiger Formen zusammensetzen.

Auch bei Dreierzyklen lassen sich in 17 von 48 Fällen (35%) Haupt-Neben-Beziehungen feststellen. Eine Haupt-Neben-Beziehung wurde allerdings nur dann als solche identifiziert, wenn die Nebenklassen innere Klassen der Hauptklasse darstellten.

Ab einer Zyklengröße von vier ist eine Einteilung nach der Größe nicht mehr zielführend. Daher werden alle weiteren Zyklengruppen nach gemeinsamen Merkmalen gruppiert und abgehandelt.

Sterne Wie bereits bei den Dreierzyklen angedeutet, weist eine gewisse Anzahl an Zyklengruppen eine sternförmige Anordnung der Abhängigkeiten auf. Ein Artefakt liegt hierbei in der Mitte und unterhält Abhängigkeiten zu jedem anderen Artefakt, wobei jedes andere Artefakt genau eine und *nur* eine Abhängigkeit zum Mittelartefakt besitzt. Abbildung 5 zeigt Sterne verschiedener Größe.

Sterne traten mit 51% am häufigsten bei Zyklengruppen der Größe drei auf, wurden allerdings sowohl in ihrer Reinform als auch mit leichten Entartungen (Abb. 5(b)) bei Zyklengruppen bis zur Größe 15 beobachtet.

Große Zyklengruppen Als große Zyklengruppen wurden all jene Zyklengruppen aufgefasst, die mindestens zehn Artefakte enthielten. Der Anteil an Zyklengruppen ab der Größe zehn betrug 21, derjenige ab der Größe fünfzig 7 Stück.

Obwohl sie lediglich 8% beziehungsweise 2% am Anteil aller Zyklengruppen stellen, zeichnen sich große Zyklengruppen anscheinend für einen Großteil der Programmsteuerung und -funktionalität verantwortlich.

Große Zyklengruppen spiegeln nicht zwangsläufig eine hochkomplexe Struktur wider. Bei Betrachtung großer Zyklengruppen in nach Entwurfsmustern entwickelten Softwaresystemen fällt bei Betrachtung der graphischen Darstellung eine gewisse Regelmäßigkeit auf.

Abbildung 6 zeigt beispielsweise eine 64 Elemente umfassende Zyklengruppe aus Eclipse. Ein Großteil aller Verbindungen verläuft von beziehungsweise zu zwei erkennbaren Zentren an der oberen beziehungsweise unteren Seite des Diagramms, die restlichen Elemente weisen untereinander tendenziell schwächere Verbindungen auf.

In den betrachteten Softwaresystemen wurden mit Ausnahme von ArgoUML keine Zyklengruppen gefunden, die bei einer Größe von ungefähr 50 eine unüberschaubare Struktur aufweisen und somit veranschaulichend als Gegenbeispiel zu Abbildung 6 dienen können. Diese Erkenntnis traf uns

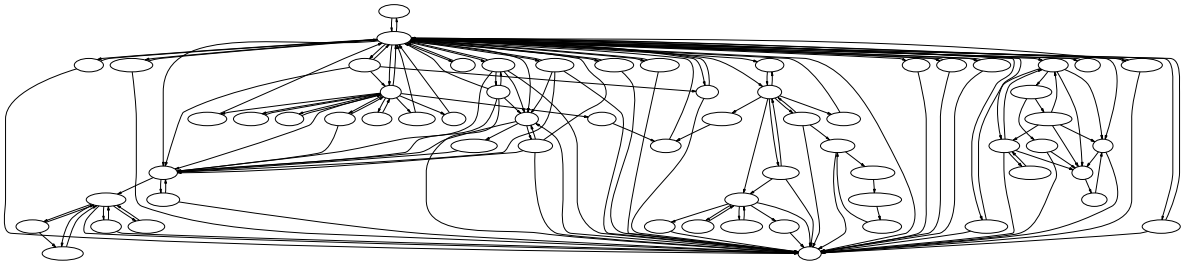


Abbildung 6: Große Zyklengruppe mit erkennbarer Regelmäßigkeit

überraschend, da wir bei großen Zyklengruppen eine überwiegend unüberschaubare Struktur vermuteten. Die einzige hochkomplexe Zyklengruppe ist die größte gefundene überhaupt, nämlich die 860 Klassen umfassende Gruppe von ArgoUML.

Gründe hierfür liegen möglicherweise in der Auswahl zu vieler »gut« entworfener Softwaresysteme und zu weniger Systeme mit bekannt schlechtem Entwurf. Ein weiterer Grund mag die Beschränkung auf ausschließlich freie Software sein, die in punkto Zuverlässigkeit proprietäre Software übertrifft [22], was sich vermutlich in entsprechenden Entwurfsentscheidungen ausdrückt.

6.3 Entwurfsmuster in Zyklen

Objektorientierte Entwurfsmuster gelten als bewährte Mittel für einen guten Entwurf [9]. Bei strikter Einhaltung der in [9] gegebenen Anweisungen induziert keines der Entwurfsmuster bis auf das Iteratormuster zyklische Abhängigkeiten. Bei der realen Implementierung treten Entwurfsmuster jedoch sehr wohl als Teile von Zyklengruppen auf.

Uns interessiert daher, welche Entwurfsmuster in welcher Häufigkeit auf welche Art und Weise zur Verzyklung von Artefakten beitragen.

Als Entwurfsmuster wurden im Rahmen der Analyse nur jene ausgewertet, die sich durch eine oberflächliche Betrachtung der Klassen und ihrer Abhängigkeiten eruieren ließen. Als Indikator für Entwurfsmuster dienten daher

- Klassennamen (*Iterator, *Factory, usw.),
- Methodennamen (create*, new*, usw.) sowie
- Abhängigkeiten gemäß Kapitel 5.4 zwischen den vorhergehenden Artefakten zur Auflösung von Mehrdeutigkeiten, wenn der Mustervergleich nicht genügte.

Tabelle 2 beschreibt die Namensmuster, die zur Erkennung von Entwurfsmustern Anwendung fanden. * steht dabei für eine beliebige Folge von Zeichen. Die Entwurfsmuster Erzeuger, Verbund, Schablonenmethode und Interpreter wurden nicht berücksichtigt, da sie nicht trivialerweise mit passenden Namensmustern zu identifizieren waren.

Alle analysierten Artefakte und Abhängigkeiten von Entwurfsmustern befanden sich innerhalb einer Zyklengruppe, nach außen führende oder von außen kommende Abhängigkeiten sowie außenliegende Artefakte wurden nicht betrachtet.

Die Untersuchung zeigte einen 18%igen Anteil an Entwurfsmustern in den untersuchten Zyklengruppen auf, insbesondere in den kleinen mit zwei oder drei Elementen. Im Nachfolgenden ist zu jedem Entwurfsmuster ausgeführt, inwiefern in den untersuchten Zyklengruppen Entwurfsmuster einen

Entwurfsmuster	Klassenmuster	Methodenmuster	Abhängigkeiten
Abstrakte Fabrik	`*Factory*`	–	Vererbungsabhängigkeit
Adapter	`*Adapter`	–	–
Befehl	`*Command`	–	–
Beobachter	`*Listener`, `*Observer`	–	–
Besucher	`*Visitor`	–	–
Brücke	`*Bridge`	–	–
Erzeuger		nicht berücksichtigt	
Dekorator	`*Decorator`	–	–
Fabrikmethode	`*Factory*`	`new*`, `create*`	–
Fassade	`*Facade*`	–	–
Fliegengewicht	`*Flyweight`	–	–
Interpreter		nicht berücksichtigt	
Iterator	`*Iterator`	–	–
Mediator	`*Mediator`	–	–
Memento	`*Memento`	–	–
Prototyp	`*Prototype`	`clone*`	–
Proxy	`*Proxy`	–	–
Schablonenmethode		nicht berücksichtigt	
Singleton	–	`getInstance`	–
Strategie	`*Strategy`	–	–
Verantwortungskette	`*Handler`	–	–
Verbund		nicht berücksichtigt	
Zustand	`*State`	–	–

Tabelle 2: Angewandte Namensmuster zur Erkennung von Entwurfsmustern

Beitrag zur Zyklenbildung leisten. Dabei werden nur jene Entwurfsmuster erwähnt, die tatsächlich innerhalb von Zyklen beobachtet wurden.

Adapter Das Entwurfsmuster Adapter [9] bildet die Schnittstelle einer Klasse auf eine andere ab.

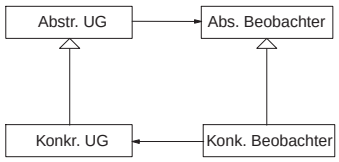
Als in einen Zyklus involvierten Adapter wurde eine Klasse des Musters `*Adapter` betrachtet.

Über alle untersuchten Softwaresysteme wurde lediglich ein Adapter identifiziert. Hierbei instantiiert die zu adaptierende Klasse den Adapter, wodurch eine zyklische Abhängigkeit erzeugt wird.

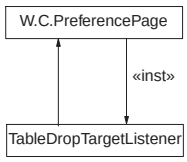
Beobachter Das Entwurfsmuster Beobachter [9] (s. Abb. 7(a)) sieht eine Abhängigkeit vom abstrakten Untersuchungsgegenstand (UG) zum abstrakten Beobachter zwecks Benachrichtigung über eine Änderung sowie Abhängigkeiten vom konkreten Beobachter zum konkreten Untersuchungsgegenstand zwecks Einholen von Zustandsinformationen vor.

Als Kennzeichen für ein in eine Zyklengruppe involviertes Beobachtermuster dienten Klassennamen des Musters `*Listener`.

In den untersuchten Softwaresystemen waren 4 Beobachter in Zyklengruppen involviert. In der Mehrzahl der Fälle wurde eine zyklische Abhängigkeit zwischen konkretem Beobachter und konkretem Untersuchungsgegenstand durch eine Instantiierungsabhängigkeit vom konkreten Untersuchungsgegenstand zum konkreten Beobachter verursacht. Beispielsweise bedient sich der UG `WelcomeCustomizationPreferencePage` des Beobachters `TableDropTargetListener`, der

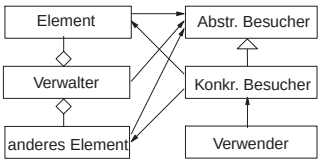


(a) Beobachtermuster

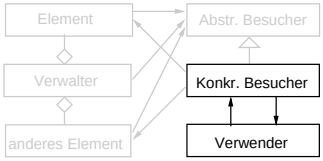


(b) Zyklus durch Instantiierung

Abbildung 7: Beobachtermuster im Zyklenkontext



(a) Besuchermuster



(b) Zyklus durch Kopplung

Abbildung 8: Besuchermuster im Zyklenkontext

als innere nichtstatische Klasse des UGs vorliegt und daher nur durch den UG instantiiert werden kann (s. Abb. 7(b)).

In den verbleibenden Fällen unterhielt die Beobachterklasse Abhängigkeiten zu weiteren Klassen, ohne jedoch den zugeordneten UG in zyklische Abhängigkeiten zu verstricken.

Besucher Das Entwurfsmuster Besucher [9] (s. Abb. 8(a)) ermöglicht die Iteration über die Elemente einer Objektmenge, wobei weder der Typ der Elemente gleich noch die Iterationsreihenfolge dem Verwender bekannt sein muss.

Die Identifizierung eines in eine Zyklengruppe involvierten Besuchers erfolgte über einen Klassennamen des Musters `*Visitor`.

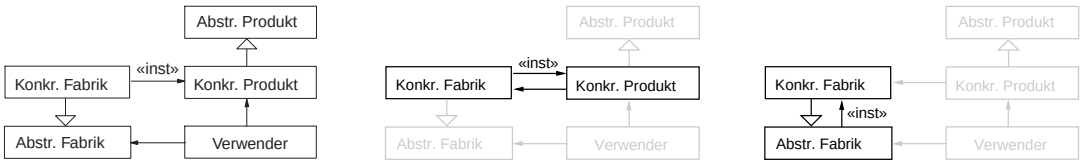
Von allen gefundenen Entwurfsmustern entsprachen 6 dem Besuchermuster. Besucherklassen wurden in der Regel (5 von 6) als innere Klasse des Verwenders realisiert und griffen auf Attribute und Methoden des Verwenders zu, was eine zyklische Abhängigkeit zur Folge hat (s. Abb. 8(b)). Hierbei scheint es sich um ein typisches Implementierungsmuster beim Einsatz von Besuchern zu handeln.

In einem Fall (`!org!apache!jasper!compiler:Validator`) stellten die konkreten Besucherklassen Teil einer großen Zyklengruppe dar, ohne dass eine ursächliche Beteiligung der Besucherklassen an der Entstehung oder Vergrößerung dieser Zyklengruppe abgeleitet werden konnte.

Abstrakte Fabrik Eine abstrakte Fabrik [9] genannte Klasse stellt (abstrakte) Methoden zur Instantiierung von Objekten zur Verfügung, die von »konkreten Fabriken« implementiert werden und tatsächliche Objekte erzeugen (s. Abb. 9(a)).

Die Identifizierung abstrakter Fabriken erfolgte durch Klassen des Musters `*Factory*`, die in einer Vererbungsabhängigkeit standen.

In den untersuchten Softwaresystemen wurden 4 abstrakte Fabriken entdeckt. Zwei induzierten eine in Abbildung 9(b) ersichtliche zyklische Abhängigkeit zwischen der konkreten Fabrik und dem



(a) Fabrikmuster

(b) Zyklus durch Fabriksverwendung

(c) Zyklus durch Standardfabrik

Abbildung 9: Fabrikmuster im Zyklenkontext

konkreten Produkt, indem das konkrete Produkt auf Attribute und Methoden der Fabrik zurückgreift. Es handelt sich hierbei um keine für das Fabrikmuster typische Implementierung.

Die anderen zwei Fabriken erzeugten eine zyklische Abhängigkeit durch bewusste Verletzung der Vererbungshierarchie über Instantiierung einer konkreten Fabrik aus der abstrakten Fabrik heraus (s. Abb. 9(c)). Dieses Muster findet Anwendung, um dem Verwender transparent über die abstrakte Fabrik eine konkrete Standardfabrik zur Verfügung zu stellen.

Fabrikmethode Von allen in Zyklen involvierten Entwurfsmustern trugen die Fabrikmethoden [9] mit 8 zur Gesamtanzahl bei. Einer Fabrikmethode kommt dabei die Aufgabe der Instantiierungsabstraktion zu.

Die Existenz einer Fabrikmethode wurde durch Klassenmuster `*Factory*` oder Methodenmuster `create*` oder `new*` festgestellt. Falls es sich um eine Fabrikklasse handelte, wurde sie nicht als Fabrikmethode gezählt.

Zwei der gefundenen Muster ähneln denjenigen der abstrakten Fabrik. In der Tat handelt es sich um eine analoge Ursache, nämlich dass das Produkt durch Rückbezüge auf die Fabrikklasse eine zyklische Abhängigkeit verursachte (s. Abb. 9(b)). Lediglich die Tatsache, dass die involvierten Fabrikklassen keine Implementierungen *abstrakter* Fabrikklassen darstellten, führten zu ihrer Abhandlung in diesem Abschnitt.

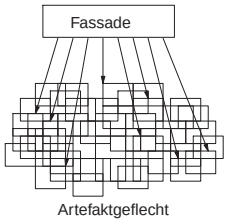
In den restlichen Fällen ließen sich in zweien keine im Zuge der Verwendung des Entwurfsmusters verursachten zyklischen Abhängigkeiten feststellen, in zweien instantiierte eine Fabrikklasse untergeordnete Fabrikklassen, die wiederum die Instantiierung von Objekten an die übergeordnete Fabrikklasse delegierten und somit zyklische Abhängigkeiten verursachten, sowie einer Klasse, die abstraktes Produkt und Fabrik in einem war und deren abgeleitete Klassen aufgrund der Vererbungshierarchie zwangsläufig in einen Zyklus gebunden wurden (veranschaulichend: `Fabrik` instantiiert über Fabrikmethode `AbgeleitetVonFabrik` (Hinbeziehung), welche wiederum eine Vererbungsabhängigkeit zurück zu `Fabrik` unterhält (Rückbeziehung)).

In einem Fall erfolgte die Zyklenbildung durch die Instantiierung der Fabrikklasse aus dem konkreten Produkt heraus.

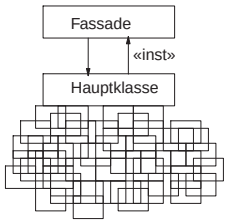
Fassade Eine Fassade [9] bietet eine einheitliche Schnittstelle nach außen zu einem funktional zusammengehörigen Artefaktgeflecht (s. Abb. 10(a)).

Die Identifizierung der Fassaden erfolgte durch Klassen des Musters `*Facade*`.

Insgesamt konnten 2 Fassaden ausgemacht werden. In beiden Fällen entstand ein Zyklus dadurch, dass die Hauptklasse eine Fassade instantiierte, die wiederum auf die Hauptklasse zugriff (s. Abb. 10(b)).



(a) Fassadenmuster



(b) Zyklus durch Instantiierung

Abbildung 10: Fassadenmuster im Zyklenkontext

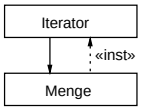


Abbildung 11: Iteratormuster im Zyklenkontext

Iterator Das Iteratormuster [9] ermöglicht einem Verwender das Durchlaufen aller Elemente einer Menge, soweit sie gleichen Typs sind. Der Iterator ist das einzige Entwurfsmuster, dessen Beschreibung die Errichtung einer zyklischen Abhängigkeit suggeriert, wenn auch nicht zwingend vorschreibt (s. Abb. 11).

Die Identifikation von Entwurfsmustern erfolgte anhand von Klassen des Musters `*Iterator`.

Iteratoren stellen mit 21 (42%) den größten Anteil aller in Zyklen involvierten Entwurfsmuster. Alle untersuchten Iteratoren weisen eine Rückbeziehung durch deren Instantiierung durch die Mengenklasse auf.

Proxy Ein Proxy [9] entspricht einer nahen Schnittstelle für eine »entfernte« Klasse.

Die Feststellung des Proxymusters erfolgte anhand von Klassen des Musters `*Proxy`.

Es wurden zwei Vorkommen des Proxymusters identifiziert, die beide eine zyklische Abhängigkeit durch Abbildung der Proxymethoden auf entsprechende entfernte Methoden in der Hinbeziehung sowie Instantiierung der Proxyklasse durch seine entfernten Klasse als Rückbeziehung erzeugten. Die beobachtete Verzyklung ähnelt der des Adapters.

Singleton Das Singletonmuster [9] begrenzt die Anzahl der Instanzen einer Klasse auf eine.

Als Singleton wurden Klassen mit Methoden des Musters `getInstance` in Erwägung gezogen. Entwurfsmuster, die oftmals ein Singletonmuster implizieren, wie Fabrikklassen, wurden nicht speziell als Singleton gezählt.

Nur ein einziges Vorkommen konnte entdeckt werden. Das Singleton existiert als innere Klasse und wird durch die äußere Klasse verwendet. Als innere greift das Singleton auch auf Methoden der äußeren Klasse zu und verursacht damit eine zyklische Abhängigkeit. Angesichts der Tatsache, dass nur ein in Zyklen involviertes Vorkommen beobachtet und die Rückbeziehung eher durch die Implementierung als innere Klasse denn durch das Entwurfsmuster verursacht wurde, kann beim Singletonmuster kein nennenswerter Beitrag zur Zyklenbildung festgestellt werden.

Von den 23 in [9] gegebenen Entwurfsmustern wurden 19 gesucht und davon 9 als in Zyklengruppen involviert identifiziert. Soweit die Implementierung eines Entwurfsmusters überhaupt unmittelbar

Applikation	**Gr**	**ZG**	**2 (%)**	**3 (%)**	☆ (%)	**GZG (%)**	**10+ (%)**	**50+ (%)**	**EMZ**
find-bugs 1.1.3	783	34	24 (6)	5 (1)	0 (0)	5 (25)	2 (22)	1 (17)	14
Tomcat 5.5.20	773	50	23 (5)	14 (5)	11 (6)	7 (20)	2 (20)	2 (20)	5
Eclipse 3.2.1	1805	86	53 (5)	16 (2)	10 (2)	14 (20)	4 (19)	2 (17)	22
ArgoUML 0.23.5	1403	10	6 (0)	1 (0)	1 (0)	2 (61)	1 (61)	1 (61)	2
Kowari 1.2	1909	58	30 (3)	11 (1)	12 (2)	10 (6)	4 (6)	1 (3)	6
CVSGrab 2.2.2	73	4	2 (5)	0 (0)	0 (0)	2 (47)	1 (35)	0 (0)	0
compress	9	1	0 (0)	1 (33)	1 (33)	0 (0)	0 (0)	0 (0)	0
JESS 3.2	151	2	1 (1)	0 (0)	0 (0)	1 (17)	1 (17)	0 (0)	0
db	3	0	–	–	–	–	–	–	–
javac 1.0.2	176	9	6 (6)	0 (0)	1 (2)	2 (27)	1 (24)	0 (0)	0
mpegaudio	55	1	0 (0)	0 (0)	1 (27)	0 (0)	0 (0)	0 (0)	0
raytrace	25	2	1 (8)	0 (0)	0 (0)	1 (36)	0 (0)	0 (0)	0
Jack 0.1	56	0	–	–	–	–	–	–	–
SPECjbb2005	61	2	1 (3)	0 (0)	0 (0)	1 (42)	1 (42)	0 (0)	0
Σ	7282	259	147 (4)	48 (1)	37 (2)	63 (23)	21 (22)	7 (20)	49

Tabelle 3: Aufschlüsselung der Analyseergebnisse nach Applikationen
Gr Anz. untersuchter Primärartefakte (Klassen), **ZG** Anz. Zyklengr., **2** Anz. Zweierzyklen, **3** Anz. Dreierzyklen, ☆ Anz. Sterne, **GZG** Anz. großer Zyklengruppen, **10+** Anz. Zyklengr. ab 10 Elementen, **50+** Anz. Zyklengr. ab 50 Elementen, **EMZ** Anz. Entwurfsmuster in Zyklengr., **(%)** in % von Gr.

zur Zyklenentstehung oder -vergrößerung beitrug, entstanden in keinem Fall Zyklengruppen mit mehr als zwei Artefakten.

Die Muster *Iterator* und *Besucher* wirkten immer zyklenerzeugend, da sie in der Regel als innere Klasse der zu iterierenden beziehungsweise besuchenden Klasse implementiert wurden und direkt auf Elemente der äußeren Klasse zugriff. Andere häufig anzutreffende Entwurfsmuster waren Fabrikmethoden, deren enthaltende Klasse entweder von einem Produkt oder von einer abstrakten Fabrikklasse referenziert wurde. Jene Entwurfsmuster, deren Verzyklung nur ein einziges Mal beobachtet werden konnten, erscheinen für eine Aussage nicht als statistisch relevant genug.

Die SpecJVM98- und SPECjbb2005-Benchmarks wiesen keine in Zyklen involvierte Entwurfsmuster auf.

Tabelle 3 schlüsselt die Analyseergebnisse noch einmal nach Art der Zyklengruppe sowie der an Zyklen beteiligten Entwurfsmustern je Applikation auf.

6.4 Einstufung von Zyklengruppen

Wie bereits eingangs erwähnt, erschweren zyklische Abhängigkeiten Programmverständnis und Qualitätsmerkmale wie Wartbarkeit und Testbarkeit. Im Rahmen dieser Untersuchung wurde daher versucht, die »Güte« einer Zyklengruppe festzustellen, also eine Einteilung einer Zyklengruppe in eine der Kategorien »harmlos« oder »verbesserungswürdig« vorzunehmen.

Die Bezeichnungen »harmlos« und »verbesserungswürdig« sind lediglich im technischen Sinne aufzufassen und erheben keinerlei Anspruch auf inhaltliche Wertung.

Eine große Menge an Artefakten bedarf eines großen Zeitaufwandes zu deren Analyse. Wenn darüber hinaus die Artefakte in einer Zyklengruppe enthalten sind und somit jedes Artefakt von jedem anderen Artefakt derselben Zyklengruppe erreicht werden kann, ist eine Erhöhung des benötigten

Zeitaufwandes durch die über die zyklischen Abhängigkeiten herbeigeführte Erreichbarkeit aller Artefakte zu erwarten. Deswegen erfolgte die Beurteilung großer Zyklengruppen durchwegs als »verbesserungswürdig«, ohne auf konkrete Einzelfälle einzugehen, während im Gegensatz dazu je weniger Artefakte enthaltende Zyklengruppen als umso »harmloser« einzustufen sind.

Wir benötigen daher einer einen Schwellwert für die Größe, unter dem eine Zyklengruppe als »harmlos« erachtet werden kann und keine weiteren Berücksichtigung in der Auflösung von Zyklengruppen finden soll. Dabei sind einerseits die Einflüsse falscher Negative zu minimieren, um nicht vorzeitig Gelegenheiten zur Verbesserung von Programmverständnis und Wartbarkeit auszusondern, andererseits soll der Schwellwert nicht so niedrig liegen, dass eine Vielzahl »harmloser« Zyklengruppen als »verbesserungswürdig« eingestuft werden und somit die tatsächlichen Problemfälle verdecken.

Die Eruierung geeigneter Schwellwerte stellt ein Problem für sich dar. Allgemein akzeptierte Schwellwerte für objektorientierte Metriken scheinen nicht zu existieren, und Untersuchungen in der Kognitionstheorie [1] legen sogar nahe, dass es keine kognitiven Schwellwerte gibt, die zuverlässig eine zu hohe Komplexität eines Artefakts oder einer Menge von Artefakten anzuzeigen vermögen.

Bereits Melton und Tempero [21] stellten die Frage nach der Güte einer Zyklengruppe hinsichtlich ihrer Auflösungsnotwendigkeit und gestand Zweierzyklen das Attribut »harmlos« zu (im Original »gut«), vermied aber die Angabe eines Schwellwerts.

Für *Zweierzyklen* schließen wir uns der Betrachtung von [21] an und ergänzen diese um die Begründung, dass zum Verständnis einer interessanten Abhängigkeitskette (...→A→B→C→...) ohnehin alle Artefakte zu prüfen sind. Beispielsweise müssen zum Verständnis der in der Abhängigkeitskette enthaltenen zwei Artefakte A und B mit der azyklischen Verbindung A→B sowohl A als auch B betrachtet werden.

Besteht nun zwischen A und B auch eine Rückbeziehung, sodass nun ein Zweierzyklus vorliegt, müssen auch in diesem Fall die Artefakte A und B betrachtet werden. Dies steht in völligem Gegensatz zu einer größeren Zyklengruppe, die A und B neben einer Anzahl weiterer Artefakte umfasste, die jedoch nicht Teil der interessanten Abhängigkeitskette sind. Hier sind neben A und B auch die restlichen in der Zyklengruppe enthaltenen Artefakte zu betrachten, da die zyklischen Abhängigkeiten Rückkoppelungen auf A und B verursachen könnten und daher nicht von der Betrachtung ausgeschlossen werden dürfen.

Dreierzyklen gelten aufgrund der geringen Anzahl von Artefakten ebenfalls als »harmlos«. Obwohl in kreisförmigen (d. h. nicht sternförmigen) Zyklengruppen zwei Artefakte nicht ohne Einbeziehung des dritten verstanden werden können, vertreten wir die Ansicht, dass bei durch als »verbesserungswürdig« eingestufte Dreierzyklen verursachte Umbauten der zukünftige Wartungsaufwand sogar steigt. Warum? Eine typische Auflösungsstrategie sieht zum Beispiel eine Abhängigkeitsumkehr [18] vor, was allerdings die Einführung einer neuen Schnittstelle bedingt. Die Zyklengröße verringert sich zwar von drei auf zwei, allerdings sind nun vier Klassen anstatt drei vorhanden, die gewartet werden müssen. Bei größeren Zyklen fällt das Hinzufügen einzelner Klassen weniger schwer ins Gewicht, wenn dadurch die Größe der Zyklengruppe stark reduziert werden kann.

Sterne sind im Grunde genommen eine Menge von Zweierzyklen, deren eines Artefakt immer dasselbe darstellt. Das Verständnis eines Blattartefakts bedingt lediglich die Einbeziehung des Hauptartefakts. Nur das Hauptartefakt selbst benötigt zum Verständnis die Einbeziehung aller Blattartefakte. Da das Zustandekommen eines Sterns nicht als Zufall zu werten, sondern als vorsätzliche Entwurfsentscheidung anzusehen ist, kann ein Stern beliebiger Größe ebenfalls als »harmlos« betrachtet werden.

Zyklengruppen egal welcher Form, die durch innere Klassen hervorgerufen wurden und sich auf innere Klassen derselben äußeren Klasse beschränken, sind ebenso als »harmlos« anzusehen. Der

Grund liegt in der Vermutung einer entwurfstechnisch gewollten Entscheidung für eine hochkohäsive Einheit. Insbesondere fördert die Implementierung innerer Klassen in Java [10] die Entstehung zyklischer Abhängigkeiten, da innere Klassen transparent auf Attribute und Methoden äußerer Klassen zugreifen können und zumindest einmal durch die äußere Klasse instantiiert werden mussten.

Zuletzt bleibt noch die Suche nach einem geeigneten Schwellwert, der die obere Grenze der Harmlosigkeit einer Zyklengruppe beschreibt, die nicht bereits durch eine der obigen Fälle festgelegt wurde. Wir definieren nun einen allgemeinen Schwellwert von 4, der nach als Trennlinie über die Güte einer Zyklengruppe entscheidet und durch folgende Überlegungen zustande kam: (1) Es existiert anscheinend kein objektiver Schwellwert für Komplexität [1]. (2) Die Zahl vier stellt anscheinend eine Grenze für die Anzahl von Objekten dar, ab der die gleichzeitige Erinnerung und Verarbeitung im Gedächtnis erschwert ist [5].

Damit lässt sich eine Richtlinie aufstellen, die zur Feststellung der »Güte« einer Zyklengruppe dient. Die Richtlinie besteht aus einer Folge von Vorschriften in aufsteigender Reihenfolge. Wenn eine Vorschrift zutrifft, werden keine weiteren Vorschriften größerer Ordnungszahl angewandt.

1. Besteht die Zyklengruppe aus inneren Klassen ausschließlich derselben äußeren Klasse und höchstens der äußeren Klasse selbst, so gilt die Zyklengruppe als »harmlos«.

2. Ist eine Zyklengruppe ein *Stern* (s. S. 12), so gilt sie als »harmlos«.

3. Beträgt oder überschreitet die Größe der Zyklengruppe den Schwellwert 4, so gilt die Zyklengruppe als »verbesserungswürdig«.

4. Die Zyklengruppe gilt als »harmlos«.

Die Anwendung der Richtlinie auf die Menge aller 259 untersuchten Zyklengruppe ergibt 35 als »verbesserungswürdig« festgestellte Zyklengruppen. Obwohl der Anteil an »verbesserungswürdigen« Gruppen lediglich 13% beträgt, enthält er 26% aller Primärartefakte und sogar76% aller Primärartefakte innerhalb von Zyklengruppen (s. auch Tabelle 1 für Aufschlüsselung nach Anwendungen). Wäre also ein Entwickler mit der Auflösung von Zyklengruppen betraut, so sparte ihm diese Richtlinie die sinnlose Untersuchung 224 »harmloser« Zyklengruppen.

7 Zusammenfassung und Ausblick

In dieser Untersuchung wurden 259 Zyklengruppen aus 14 Java-Anwendungen analysiert und eine Einteilung der Formen sowie eine Auswertung der Häufigkeit vorgenommen. Weiters wurde die Beteiligung von Entwurfsmustern [9] an der Zyklenbildung untersucht und festgestellt, dass Entwurfsmuster höchstens zur Bildung von Zweierzyklen führen.

Wir versuchten außerdem eine Einteilung von Zyklengruppen hinsichtlich ihrer »Güte« finden und entwickelten eine einfache größenbasierte, vollautomatisch berechenbare Richtlinie, die unter Einbeziehung von Größe und Anatomie einer Zyklengruppe eine Unterscheidung zwischen »Harmlosigkeit« und »Verbesserungswürdigkeit« ermöglicht.

In der Mehrheit der untersuchten Applikationen befand sich gut ein Drittel aller Artefakte in Zyklengruppen, der Anteil der Artefakte in Zyklengruppen der Güteklasse »verbesserungswürdig« belief sich auf ungefähr ein Fünftel.

Für zukünftige Forschungen wären weitere Untersuchungen an einer größeren, repräsentativeren Stichprobe angebracht, insbesondere die Untersuchung von Applikationen in C++, wodurch sich

möglicherweise neue Formen von Zyklengruppen ergeben. Zumindest erwarten wir von C++-Applikationen eine tendenziell stärkere Zyklenbildung, da die Sprache kein eigenes Schnittstellenkonstrukt anbietet und der Entwickler das Überschreiben von Methoden mittels `virtual` explizit ermöglichen muss. Beides mag den Entwickler zu verfrühter Mikrooptimierung [13] verleiten.

Darüber hinaus sollte ein zuverlässigeres Maß für die »Güte« einer Zyklengruppe gefunden werden, welches auch die Strukturiertheit einer Zyklengruppe miteinbezieht. Unter Umständen stellte sich sogar heraus, dass bestimmte Abhängigkeiten zwischen Artefakten als nicht verständniserschwerend anzusehen sind und daher für die Betrachtung zyklischer Abhängigkeiten weggelassen werden können. In diesem Falle vereinfachte sich die Betrachtung der Zyklengruppe, ohne dass die unterliegenden Quelltextartefakte zu ändern wären.

Diese Untersuchung diente der Erhebung zyklischer Abhängigkeiten zur Entwicklung von Strategien zu deren Auflösung. Wir werden die gewonnenen Erkenntnisse zu weiteren Forschungen für die automatisierte Auflösung zyklischer Abhängigkeiten einsetzen.

Dank

Mein Dank gilt Prof. Hanspeter Mössenböck für seine wertvollen Hinweise zur Verbesserung und Präzisierung dieses Artikels.

Literatur

[1] BENLARBI, SAIDA, KHALED EI-EMAM, NISHITH GOEL und SHESH N. RAI: *Thresholds for Object-Oriented Measures*. Technischer Bericht, Conseil national de recherches Canada, Institut de Technologie de l'information, 2000.

[2] BINKLEY, DAVID und MARK HARMAN: *Locating Dependence Clusters and Dependence Pollution*. In: *Proceedings of the International Conference on Software Maintenance*, 2005.

[3] BRIAND, LIONEL C., YVAN LABICHE und YIHONG WANG: *An Investigation of Graph-Based Class Integration Test Order Strategies*. IEEE Transactions on Software Engineering, 29(7):594–607, 2003.

[4] CLARK, JOHN und DEREK ALLAN HOLTON: *Graphentheorie*. Spektrum Akademischer Verlag, 1994.

[5] COWAN, NELSON: *The magical number 4 in short-term memory: A reconsideration of mental storage capacity*. Behavioral and Brain Sciences, 24:87–185, 2000.

[6] DEMEYER, SERGE, SANDER TICHELAAR und PATRICK STEYAERT: *FAMIX 2.0: The FAMOOS Information Exchange Model*. Technischer Bericht, Universität Bern, 1999.

[7] DIJKSTRA, EDSGER W.: *The structure of the THE-multiprogramming system*. Communications of the ACM, 11(5):341–346, 1968.

[8] FOWLER, MARTIN R.: *Reducing Coupling*. IEEE Software, 18(4):102–104, 2001.

[9] GAMMA, ERICH, RICHARD HELM, RALPH JOHNSON und JOHN VLISSIDES: *Design Patterns CD*. Addison Wesley Longman, 1996.

[10] GOSLING, JAMES, BILL JOY, GUY STEELE und GILAD BRACHA: *The Java? Language Specification*. Addison-Wesley, 3. Auflage, 2005.

[11] HASHIM, NOR LAILY, HEINZ W. SCHMIDT und SITA RAMAKRISHNAN: *Test order for class-based integration testing of java applications*. In: *Proceedings of International Conference on Quality Software 2005*, 2005.

[12] HAUTUS, EDWIN: *Improving Java Software Through Package Structure Analysis*. In: *Proceedings of the 6th IASTED International Conference Software Engineering and Applications*, 2002.

[13] HYDE, RANDALL: *The Fallacy of Premature Optimization*. Ubiquity, 7(24), 2006.

[14] LAKOS, JOHN: *Large-Scale C++ Software Design*. Addison Wesley Longman, 1996.

[15] LANZA, MICHELE: *The Evolution Matrix: Recovering Software Evolution using Software Visualization Techniques*. In: *Proceedings of the 4th International Workshop on Principles of Software Evolution*, Seiten 37–42, 2001.

[16] LANZA, MICHELE und STÉPHANE DUCASSE: *Polymetric Views – A lightweight Visual Approach to Reverse Engineering*. IEEE Transaction on Software Engineering, 29(9):782–794, 2003.

[17] LUNGU, MIRCEA, MICHELE LANZA und TUDOR GÎRBA: *Package Patterns for Visual Architecture Recovery*. In: *Proceedings of the Conference on Software Maintenance and Reengineering*, 2006.

[18] MARTIN, ROBERT C.: *The Dependency Inversion Principle*. C++ Report, 1996.

[19] MARTIN, ROBERT C.: *Granularity*. The C++ Report, 8(11), 1996.

[20] MELTON, HAYDEN und EWAN TEMPERO: *Empirical Study of Cycles among Classes in Java*. Technischer Bericht, Department of Computer Science, Universität Auckland, 2006.

[21] MELTON, HAYDEN und EWAN TEMPERO: *Identifying Refactoring Opportunities by Identifying Dependency Cycles*. In: *Proceedings of the 29th Australasian Computer Science Conference*, 2006.

[22] MILLER, BARTON P., DAVID KOSKI, CJIN PHEOW LEE, VIVEKANANDA MAGANTY, RAVI MURTHY, AJITKUMAR NATARAJAN und JEFF STEIDL: *Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services*. Technischer Bericht, Universität Wisconsin, Computer Sciences Department, 1995.

[23] NIERSTRASZ, OSCAR, STÉPHANE DUCASSE und SERGE DEMEYER: *Object-Oriented Reengineering Patterns – An Overview*. In: *Proceedings of Generative Programming and Component Engineering*, Seiten 1–9. Springer Verlag, 2005. LNCS 3676.

[24] NUUTILA, ESKO und ELJAS SOISALON-SOININEN: *On Finding the Strong Components in a Directed Graph*. Technischer Bericht, Laboratory of Information Processing Science, Technische Universität Helsinki, 1994.

[25] PARNAS, DAVID LORGE: *Software aging*. In: *Proceedings of the 16th International Conference of Software Engineering*, Seiten 279–287, 1994.

[26] SARKAR, SANTONU, GIRISH MASKERI RAMA und SHUBHA R: *A Method for Detecting and Measuring Architectural Layering Violations in Source Code*. In: *Proceedings of XIII Asia Pacific Software Engineering Conference*, 2006.

[27] SKIENA, STEVEN S.: *The Algorithm Design Manual*. Springer-Verlag, 1997.