Objektorientierte Systeme unter der Lupe

Markus Bauer und **Oliver Ciupke**
Forschungszentrum Informatik
Haid-und-Neu-Straße 10-14
76131 Karlsruhe
bauer@fzi.de, ciupke@fzi.de

1 Motivation

In modernen, großen objektorientierten Systemen ist eine sinnvolle Strukturierung nötig, die auch Abstraktionsebenen oberhalb von Klassen einschließt. Eine solche Strukturierung muss auch während der Evolution der Systeme gewahrt werden. Dies ist schwierig, weil neue Anforderungen immer wieder in die Systeme eingearbeitet werden müssen und dies in vielen Fällen zur schrittweisen Zerwartung des Systems führt. Systemkonstrukteure benötigen daher Methoden und Werkzeuge, um die Struktur und die Architektur der Systeme während des kompletten Entwicklungszyklus zu untersuchen, Problemstellen aufzudecken und so einer solchen Zerwartung vorbeugen zu können.

Das vorliegende Kurzpapier stellt das Werkzeug *Goose* vor, das für solche Untersuchungen eingesetzt werden kann, und berichtet über Ergebnisse einer Fallstudie, in der ein größeres Java-System mit Hilfe dieses Werkzeugs untersucht wurde.

2 Das Werkzeug *Goose*

Goose[1] wurde als Reengineeringwerkzeug im Rahmen des ESPRIT-Projektes FAMOOS [2] entwickelt. *Goose* ermöglicht es, mit Hilfe von Design-Anfragen die statische Struktur eines objektorientierten Systems zu untersuchen und kann so Softwareentwickler dabei unterstützen, Schwachstellen im Systemdesign zu entdecken und zu beheben [1]. Solche Untersuchungen müssen auf Basis des Quellcode des Systems erfolgen, weil nur dieser den tatsächlichen Zustand des Systems widerspiegelt. Extern erstellte Design- und Architekturdokumentation dagegen ist in aller Regel nicht geeignet, solche Untersuchungen durchzuführen, weil sie entweder gar nicht existiert oder – falls sie existiert – meist nicht konsistent zur Implementierung des Systems gehalten wird und so nicht (mehr) das tatsächliche Design des Systems reflektiert.

Goose arbeitet daher wie folgt: Zunächst erfolgt eine *Analyse* des Quellcode des Systems. Dabei wird eine Datenbank gefüllt, die das Design des Systems repräsentiert. Diese Design-Datenbank enthält alle Konstrukte des Systems (Subsysteme, Klassen, Methoden, . . .) und alle Beziehungen und Abhängigkeiten zwischen diesen (Vererbungsbeziehungen, Methodenaufrufe, Attributdeklarationen, . . .).

[1] http://www.fzi.de/prost/tools/goose/

In einem weiteren Schritt kann die Designdatenbank manipuliert werden:

- Mit Hilfe von *Selektionsoperationen* können die weiteren Untersuchungen auf bestimmte Teile der Datenbank eingeschränkt werden. Dadurch lassen sich beispielsweise Klassen, die lediglich Infrastrukturcode enthalten und wenig zur wirklichen Systemfunktionalität beitragen, von weiteren Betrachtungen ausnehmen.
- *Aggregationsoperationen* dienen dazu, zusammengehörige Elemente des Systems zu neuen, logischen Einheiten zusammenzufassen. Dadurch lässt sich eine abstrahierte Sicht auf das System herstellen, die insbesondere bei größeren Systemen von besonderem Nutzen ist.

Eine auf diese Weise gewonnene, *abstrahierte* Designdatenbank des Systems kann dann mit Hilfe von Designanfragen, die in einer geeigneten Abfragesprache formuliert worden sind, untersucht werden. Momentan setzen wir zu diesem Zweck *Prolog* ein, weil sich damit auf einfache und verständliche Art und Weise effiziente Anfragen erstellen lassen.

3 Fallstudie

Wir haben *Goose* und die zugrundeliegenden Techniken bereits in mehreren Projekten erfolgreich angewandt. In diesem Abschnitt berichten wir aus einem dieser Projekte. Bei der dem Projekt zugrundeliegenden Fallstudie handelt es sich um ein ca. 1 Million Zeilen großes System, welches zum Vertragsmanagement in einer Versicherungsgesellschaft eingesetzt werden soll. Das System ist in Java entwickelt worden und basiert auf *Enterprise JavaBeans (EJB)*.

Wie in modernen Systemen üblich, sieht die Architektur des Systems Komponenten im Sinne von strukturgebenden Elementen auf verschiedenen Abstraktionsstufen vor. Im vorliegenden System finden wir auf der unterster Abstraktionsebene *Klassen* bzw. *Enterprise JavaBeans* vor, die zu logischen *Geschäftsobjekten* zusammengefasst werden. Mehrere Geschäftsobjekte wiederum bilden ein *Subsystem*. Abbildung 1 illustriert diese Strukturierung und gibt zusätzlich an, aus wievielen Komponenten jeder Abstraktionstufe das System besteht. Aus diesen Zahlen wird sofort klar, dass eine Untersuchung des Systems der Übersicht halber stets auf

höheren Abstraktionsstufen beginnen und dann schrittweise auf niedrigere Stufen verfeinert werden sollte. Hierbei leisten die oben erwähnten Selektions- und Abstraktionsmechanismen wertvolle Hilfe.

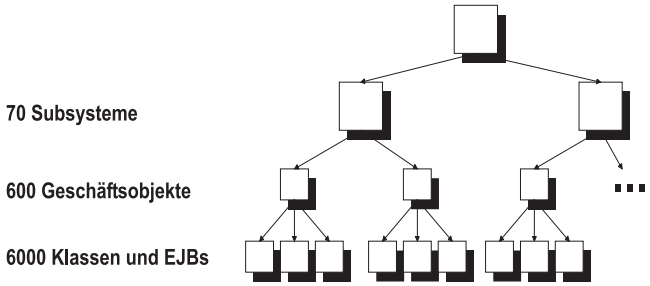

Abbildung 1: Komponententypen in der Fallstudie

Durch Selektionsmechanismen kann EJB-spezifischer Infrastrukturcode und Code zur Datenbank- und Großrechnerintegration herausgefiltert werden, so dass sich die Analyse auf die eigentliche Anwendungslogik beschränken kann. Mechanismen zur Aggregation werden hauptsächlich an zwei Stellen verwendet. Zum einen müssen Client-Interfaces und Server-Implementierungen der EJBs zusammengeführt werden, damit die Abhängigkeiten zwischen Client-Code und Servercode, die im Quellcode nur implizit über Mechanismen zum entfernten Methodenaufruf (via CORBA bzw. RMI) vorhanden sind, berücksichtigt werden können. Zum anderen muss die Geschäftsobjektstruktur, die sich nur implizit im Quellcode des Systems wiederfindet, wiederhergestellt werden. Diejenigen Java-Klassen, die ein Geschäftsobjekt implementieren, können durch Namenskonventionen identifiziert werden und zu einer neuen logischen Einheit *Geschäftsobjekt* aggregiert werden.

Das auf diese Weise bereinigte Modell des Systems haben wir dann mit Hilfe von Prolog-Anfragen auf eine Reihe von problematischen Eigenschaften hin überprüft, die sich einerseits aus allgemein bekannten *Heuristiken* [6] [4] und andererseits aus technologie- bzw. projektspezifischen und *Richtlinien* [3] [5] ergeben haben. Im folgenden skizzieren wir kurz einig dieser problematischen Eigenschaften:

- *Fragile Klassen, Geschäftsobjekte und Subsysteme; Flaschenhälse, starke Kopplung*. Unter *fragilen* Klassen versteht man Klassen, die von sehr vielen anderen Klassen abhängen. Eine fragile Klasse kann problematisch sein, weil mit der Anzahl der Abhängigkeiten einer Klasse auch ihre Empfindlichkeit gegenüber Änderungen im System zunimmt. *Flaschenhalsklassen* sind fragile Klassen, die zusätzlich von zahlreichen weiteren Klassen im System genutzt werden. Solche Klassen sind problematisch, weil sie ihre Änderungsempfindlichkeit an alle diese abhängigen Klassen weitergeben. Diese Konzepte lassen sich von Klassen auch auf Geschäftsobjekte und Subsysteme übertragen.

  Zahlreiche Abhängigkeiten zwischen verschiedenen Systemteilen führen zu hoher *Kopplung* zwischen diesen. Starke Kopplungen sind unerwünscht, weil sie einerseits die getrennte Entwicklung beteiligter Systembestandteile erschweren, und andererseits die separate Wiederverwendung dieser Systemteile verhindern.

  In der Fallstudie traten flaschenhalsartige Klassen bzw. Subsysteme in einigen Systemteilen auf, die das Rückgrat der Anwendungsarchitekur bilden. Von diesen Systemteilen hängen einerseits fast alle anderen Systemteile ab, andererseits implementieren sie die umfangreiche Persistenzschicht des Systems und nutzen damit zahlreiche Infrastrukturklassen zu deren Realisierung. In diesem Fall kann man den Infrastrukturcode als stabil einordnen, so dass der Flaschenhalscharakter dieser Systemteile kaum zu negativen Konsequenzen führen dürfte. Allerdings sind viele Systemteile stark mit den oben bereits erwähnten Basisklassen der Anwendungsarchitektur gekoppelt, so dass wesentliche Teile der Anwendungslogik nur schwer in andere Systeme mit einer anderen Anwendungsarchitektur übertragen werden können.

- *Mangelhaft gekapselte Subsysteme.* Subsysteme dienen zur logischen Gliederung des Systems. Diese Gliederung wird am besten dann erreicht, wenn Subsysteme nur über klar definierte Schnittstellen mit anderen Systemteilen zusammenhängen und die inneren Strukturen des Subsystems verborgen bleiben. Mangelhafte Kapselung kann man nur schwer genau identifizieren. Als Indikator für mangelhafte Kapselung kann ein schlechtes Zahlenverhältnis zwischen den privaten, internen Klassen eines Subsystems und den öffentlichen Klassen der Subsystemschnittstelle dienen.

  In der Fallstudie verfügten alle Subsysteme über schlanke Schnittstellen, die einen wesentlichen Teil der inneren Implementierung der Subsysteme verbergen.

- *Fragile Subsystemschnittstellen.* In Subsystemschnittstellen sollten keine fragilen Klassen enthalten sein, da sich diese Fragilität auf die Schnittstelle selbst übertragen kann. Zudem sollten Klassen in Subsystemschnittstellen nicht direkt auf andere Subsysteme zugreifen.

  In der Fallstudie waren keinerlei fragile Klassen in den Subsystemschnittstellen zu finden.

- *Schlechte Trennung von persistenten Entitäten und Aktivitäten bzw. Prozessen.* In EJB-basierten Systemen sollen Aktivitäts- bzw. Prozesskomponenten für einen kontrollierten, gekapselten Zugriff auf persistente Entitäten sorgen. Komponenten zur Implementierung höherwertiger Anwendungslogik sollten nie direkt auf Entitäten zugreifen, sondern nur über spezielle Aktivitätskomponenten. Diese Richtlinie soll die Anwendungslogik einerseits robuster gegenüber Änderungen im persistenten Datenschema machen, zum ande-

ren können die Aktivitätskomponenten für einen optimierten Zugriff auf die persistenten Daten sorgen. Diese Richtlinie ist ein Beispiel für eine typische technologie- bzw. projektspezifische Regel.

Entsprechende Prologanfragen konnten in der Fallstudie fast keine Verletzungen dieser Richtlinie aufdecken.

- *Abhängigkeiten zwischen Framework und eigentlicher Anwendung.* Bestimmte Teile des untersuchten Systems sind als Framework im Hinblick auf zukünftige Erweiterungen des Systems zu einer Produktfamilie entworfen worden. Andere Teile implementieren eine Instanzierung dieses Framework für eine bestimmte Anwendung. Mit Hilfe von Designanfragen haben wir überprüft, dass keine Abhängigkeiten von Klassen des Framework-Codes zu Teilen der Anwendungsimplementierung bestehen.

Mit Hilfe von *Goose* und den zugehörigen Designanfragen konnten wir die Fallstudie auf eine Reihe von potenziell problematischen Eigenschaften hin untersuchen. Wir haben dabei folgende Erfahrungen gemacht:

- Im Gegensatz zu vielen zuvor auf diese Weise untersuchten Systemen wies das vorliegende System relativ wenig problematische Eigenschaften auf. Wir führen dies darauf zurück, dass das System sehr systematisch und auf Basis eines Katalogs von Best-Practice-Richtlinien entworfen wurde.

- Während der Untersuchung des Systems konnten wir in relativ kurzer Zeit unser Verständnis über die Strukturierung und den Aufbau des Systems stark verbessern. Wir gehen daher davon aus, dass Designanfragen, wie sie *Goose* verwendet, über ihren eigentlichen Zweck hinaus auch wertvolle Dienste bei der Einarbeitung neuer Entwickler in ein existierendes System leisten können.

4 Zusammenfassung

Wir fassen im folgenden die wichtigsten Erkenntnisse dieses Kurzpapiers nochmals zusammen:

- Sauber definierte (und auch eingehaltene) Softwarestrukturen sind für große Systeme besonders wichtig. Um die Qualität sicherzustellen, empfiehlt es sich, diese Strukturen über den ganzen Lebenszyklus der Systeme hinweg werkzeuggestützt zu untersuchen.

- Durch Abstraktion kann man auch in Komponentensystemen automatisch überprüfen, ob bestimme Richtlinien und Regeln auch während der Implementierungeingehalten wurden, obwohl das Design im Quellcode so nicht mehr direkt zu sehen ist.

- Ergebnisse der automatischen Analyse machen durch die vereinfachende, abstrakte Sicht die Beurteilung durch den menschlichen Designer überhaupt erst möglich. Der Designer muss nicht mehr von Hand das ganze System analysieren, sondern kann sein Augenmerk auf die wenigen, wirklich kritischen Stellen lenken, die sich aus der Analyse mittels Heuristiken ergeben haben.

Unsere zukünftigen Arbeiten zum Thema automatisierbare Untersuchung von objektorientierten Systemen werden sich zum einen darauf konzentrieren, unseren Satz von Regeln und Heuristiken zu erweitern und zu konsolidieren. Wir hoffen, dass sich dabei eine Sammlung von aussagekräftigen Heuristiken herauskristallisiert, die man einfach als allgemeine Regeln zur Untersuchung beliebiger objekt- und komponentenorientierter Systeme heranziehen kann. Zum anderen wollen wir die Benutzerfreundlichkeit von *Goose* mit dem Ziel verbessern, dass es auch von anderen interessierten Softwareentwicklern leicht für eigene Untersuchungen herangezogen werden kann.

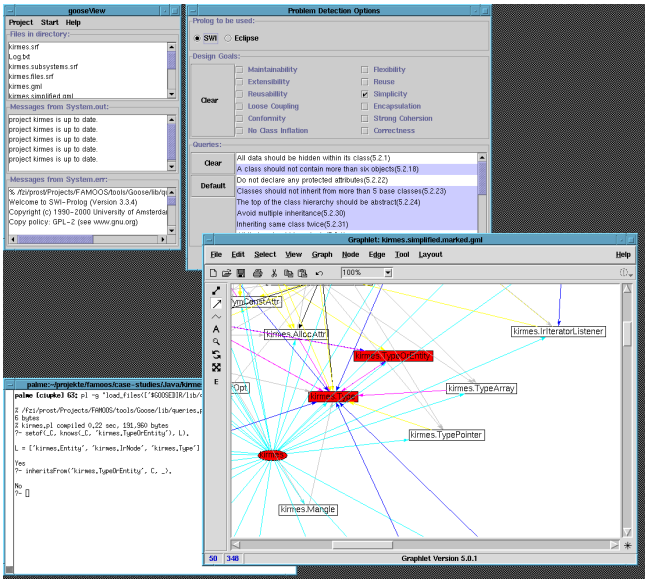

Abbildung 2: Eine Sitzung mit *Goose*

LITERATUR

[1] CIUPKE, OLIVER: *Automatic Detection of Design Problems in Object-Oriented Reengineering*. In: FIRESMITH, DONALD, RICHARD RIEHLE, GILDA POUR und BERTRAND MEYER (Herausgeber): *Technology of Object-Oriented Languages and Systems - TOOLS 30*, Seiten 18–32. IEEE Computer Society, August 1999.

[2] CIUPKE, OLIVER, SERGE DEMEYER, STEPHANE DUCASSE und JOACHIM WEISBROD: *The FAMOOS Object-Oriented Reengineering Handbook*. Technischer Bericht Forschungzentrum Informatik, Karlsruhe, Software Composition Group, University of Berne, 1999.

[3] KASSEM, NICHOLAS: *Designing Enterprise Applications with the Java2 Platform*. Addison-Wesley, 2000.

[4] MARTIN, ROBERT C.: *Stability*. C++ Report, Februar 1997. An article about the interrelationships between large scale modules.

[5] PICON, JOAQIN: *Design and Implement Servlets, JSPs, and EJBs for IBM WebSphere Application Server*. IBM Redbooks, 2000.

[6] RIEL, ARTHUR J.: *Object-Oriented Design Heuristics*. Addison-Wesley, 1996.