OBJEKTORIENTIERTE RE-ENGINEERING-MUSTER: EIN ÜBERBLICK

Erfolgreiche Softwaresysteme müssen so konzipiert sein, dass sie sich weiterentwickeln können – ansonsten gehen sie ein. Obwohl objektorientierte Softwaresysteme auf eine lange Lebensdauer ausgerichtet sind, veralten sie über die Jahre genauso wie jedes andere Legacy-Softwaresystem. Um die weiteren Entwicklungskosten niedrig zu halten, müssen daher Investitionen im Bereich Re-Engineering getätigt werden. Softwaresysteme und der Geschäftskontext, in dem sie eingesetzt werden, unterscheiden sich in vielen Punkten – dennoch ähneln sich die Techniken, sie zu verstehen, zu analysieren und zu transformieren. Folglich lassen sich verschiedene Re-Engineering-Muster identifizieren, die bewährte Vorgehensweisen beim Reverse- und Re-Engineering objektorientierter Legacy-Systeme beinhalten. Dieser Artikel stellt einen kleinen Ausschnitt aus einer großen Sammlung dieser Muster vor, die auf der mehrjährigen Erfahrung mit objektorientierten Legacy-Systemen basieren. Darüber hinaus werden Hinweise gegeben, wie einige dieser Muster durch geeignete Werkzeuge unterstützt werden können[1].

Ein *Legacy-Softwaresystem* ist ein System, das *übernommen* wurde und das einen *Wert* besitzt. Erfolgreiche (also „wertvolle") Softwaresysteme entwickeln sich typischerweise im Laufe der Jahre weiter, da sich Anforderungen und Geschäftsbedürfnisse ändern. Das führt zu dem allgemein bekannten Phänomen, dass diese Systeme über die Zeit immer komplexer werden und dadurch immer schwerer zu warten sind – es sei denn, es werden spezielle Maßnahmen ergriffen, um ihre Architektur und ihr Design zu vereinfachen (vgl. [Leh85]).

Wenn das Legacy-System anfängt zu einer Belastung zu werden, treten zahlreiche Probleme zu Tage. Zuallererst geht das *Wissen* über das System verloren. Häufig fehlt Dokumentation oder sie ist überholt. Die ursprünglichen Entwickler oder Benutzer haben das Projekt unter Umständen verlassen. Folglich ist Insider-Wissen über das System nicht mehr vorhanden. Automatisierte Tests, die zeigen, wie das System funktioniert, gibt es nur selten.

Zweitens verliert der *Änderungsprozess* an Effektivität. Einfache Änderungen dauern zu lange; Fehlerbehebungen („Bug Fixes") sind an der Tagesordnung. Gegenseitige Abhängigkeiten erschweren Änderungen bzw. die Entkopplung einzelner Produkte.

Schließlich weist der *Code* selbst diverse hässliche Symptome auf: Große Mengen duplizierten Codes kommen genauso häufig vor wie andere „Verunreinigungen", z.B. Verletzung des Kapselungsprinzips, große prozedurale Klassen und explizite Typprüfungen.

Das heißt, der Code weist nicht nur *architektonische Probleme* (z. B. unsaubere Schichtenbildung, fehlende Modularisierung) auf, sondern auch *Probleme beim Design* (z. B. falsche oder fehlende Anwendung des Vererbungsprinzips oder falsch platzierte Methoden). Exzessive Build-Zeiten sind ebenfalls ein deutliches Zeichen dafür, dass mit der Architektur etwas faul ist.

Da die Wartung bekanntlich den Großteil des Lebenszyklus eines (erfolgreichen) Softwaresystems ausmacht und wiederum ein Großteil der „Wartung" daraus besteht, neue Funktionalität einzubauen (vgl. [Lie80]), wird das Identifizieren und Lösen dieser Probleme zum entscheidenden Kriterium für das Überleben von Legacy-Systemen.

An dieser Stelle ist es sinnvoll, zwischen Reverse-Engineering und Re-Engineering von Softwaresystemen zu unterscheiden (vgl. [Chi92]). Mit *Reverse-Engineering* bezeichnen wir den Prozess, ein Softwaresystem zu analysieren, um seine Struktur und das zu Grunde liegende Design auf einem höheren Abstraktionsniveau herauszuarbeiten; es handelt sich also um einen Prozess, bei dem aus dem konkreten Softwaresystem unterschiedliche *Modelle* gewonnen werden. Unter dem Begriff *Re-Engineering* verstehen wir dem Prozess, das ursprüngliche System in ein neues System zu transformieren, das im Wesentlichen dieselben funktionalen Anforderungen erfüllt, darüber hinaus aber eine Weiterentwicklung ermöglicht (vgl. [Dem02], [Kaz98]).

[1] *Bei dem Artikel handelt es sich um eine Übersetzung des Beitrags „Object-oriented Reengineering Patterns – an Overview", in: Proc. of GPCE 2005, M.L.R. Glück (Hrsg.), LNCS 3676, 2005, S. 1-9.*

die autoren

Serge Demeyer (E-Mail: serge.demeyer@uia.ua.ac.be) ist Professor am Institut für Mathematik und Informatik der Universität Antwerpen. Dort leitet er die Forschungsgruppe LORE (Lab On REengineering).

Stéphane Ducasse (E-Mail: ducasse@iam.unibe.ch) ist Professor an der Universität Savoie und gleichzeitig Professor bei der „Software Composition Group" der Universität Bern. Er war technischer Leiter des Esprit-Projekts FAMOOS.

Oscar Nierstraß (E-Mail: oscar@iam.unibe.ch) ist Professor für Informatik an der Universität Bern, wo er die „Software Composition Group" leitet. Er ist Autor zahlreicher Publikationen zum Thema objektorientierte und komponentenbasierte Technologien.

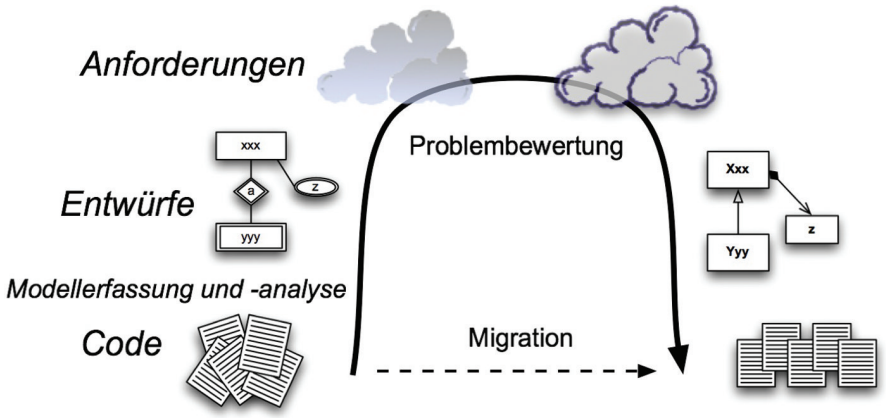

Abb. 1: Der Reverse- und Re-Engineering-Lebenszyklus

Reverse- und Re-Engineering umfassen zahlreiche Aktivitäten; dazu gehören unter anderem das Überarbeiten (*Recovery*) von Design und Architektur, Testgenerierung, das Erkennen von Problemen sowie unterschiedlichste Refaktorisierungsmaßnahmen (*Refactoring*) auf verschiedenen Ebenen. In Abbildung 1 ist der optimale Verlauf des Reverse- und Re-Engineering-Lebenszyklus dargestellt (vgl. [Dem02], [Kaz98]).

Auch wenn es zwischen den Beweggründen, ein Re-Engineering eines Softwaresystems vorzunehmen bezogen auf die Geschäftsanforderungen einer Organisation zum Teil beträchtliche Unterschiede geben mag, ähneln sich die eigentlichen technischen Schritte doch sehr. Folglich lassen sich verschiedene nützliche *Prozessmuster* identifizieren, die beim Reverse- und Re-Engineering von Legacy-Systemen eingesetzt werden können. Im folgenden Abschnitt geben wir einen kurzen Überblick über diese Muster. Darüber hinaus gibt es verschiedene *Werkzeuge*, die den Re-Engineering-Prozess unterstützen. Im Abschnitt „Re-Engineering-Werkzeuge und -Techniken" stellen wir einige der von uns entwickelten und auf verschiedene Legacy-Systeme angewandten Werkzeuge kurz vor.

Re-Engineering-Muster

Der Begriff „Muster" (*Pattern*) wird im Softwarekontext meistens mit Entwurfsmustern (*Design Patterns*), die sich wiederholende Lösungen für Designprobleme darstellen, in Verbindung gebracht. *Re-Engineering-Muster* sind keine Entwurfsmuster, sondern vielmehr *Prozessmuster*; sie stellen sich wiederholende Problemlösungen dar, die während des Reverse- und Re-Engineering-Prozesses auftauchen.

Wir unterscheiden Muster von „Regeln" oder „Leitlinien", weil jedes Muster in einem gegebenen Kontext interpretiert werden muss. Muster werden nicht blind angewendet, denn ihr Einsatz hat Folgen und muss daher sorgfältig bedacht werden. Genauso wie niemand absichtlich bei der Implementierung eines Softwaresystems alle GOF-Muster (vgl. [Gam95]) anwenden würde, sollte man auch Re-Engineering-Muster nicht wahllos einsetzen, ohne zunächst über die möglichen Konsequenzen nachzudenken.

Im Rahmen des Projekts *FAMOOS (Framework-based Approach for Mastering Object-Oriented Software Evolution)* konnten wir eine große Anzahl von Re-Engineering-Mustern identifizieren. FAMOOS ist ein europäisches Projekt (vgl. [ESP]), das das Ziel hatte, die Evolution objektorientierter Software der ersten Generation in Richtung objektorientierter Frameworks zu unterstützen. Der Schwerpunkt von FAMOOS lag zum einen auf Methoden und Werkzeugen zur Analyse und zum Aufdecken von Designproblemen bei objektorientierten Legacy-Systemen; zum anderen sollten diese Systeme flexiblere Architekturen erhalten. Im FAMOOS-Handbuch ([Duc99]) und in dem Buch „Object-Oriented Reengineering Patterns" ([Dem02]) sind die wesentlichen Projektergebnisse zusammengefasst.

In Abbildung 2 ist dargestellt, wie verschiedene Cluster von Re-Engineering-Mustern auf unseren optimalen Re-Engineering-Lebenszyklus abgebildet werden können. Jeder Name umfasst eine Menge von Prozessmustern, die zu einem bestimmten Zeitpunkt des Re-Engineerings eines Legacy-Systems angewendet werden können.

„Die Richtung vorgeben" (*Setting Direction*) enthält eine Reihe von Mustern, die dabei helfen, den Fokus der Re-Engineering-Arbeiten festzulegen und diesen nicht aus den Augen zu verlieren. „Erster Kontakt" (*First Contact*) umfasst Muster, die nützlich sein können, wenn Sie das erste Mal mit dem Legacy-System konfrontiert werden. Mit Hilfe von „Grundverständnis" (*Initial Understanding*) lässt sich ein erstes, einfaches Modell eines Legacy-Systems erstellen, meistens in Form von Klassendiagrammen. „Detaillierte Modellerfassung" (*Detailed Model Capture*) hilft dabei, ein detaillierteres Modell einer bestimmten Systemkomponente zu entwickeln. „Tests: Ihre Lebensversicherung" (*Tests: Your Life Insurance*) betont die Bedeutung von Tests – nicht nur, um das Legacy-System besser zu verstehen, sondern auch, um dieses aufs Re-Engineering vorzubereiten. „Migrationsstrategien" (*Migration Strategies)* helfen dabei, das System während des Re-Engineering-Prozesses am Laufen zu halten; darüber hinaus verbessern die Migrationsstrategien die Chancen, dass das neue System von den Benutzern akzeptiert wird. „Duplizierten Code aufspüren" (*Detecting Duplicated Code*) kann dabei

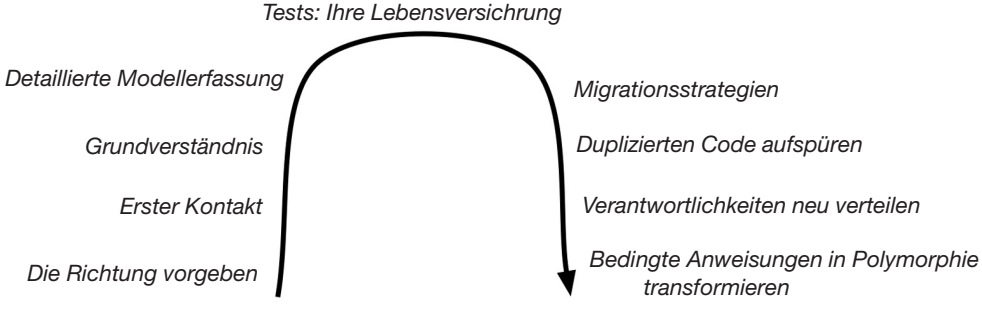

Abb. 2: Cluster von Re-Engineering-Mustern

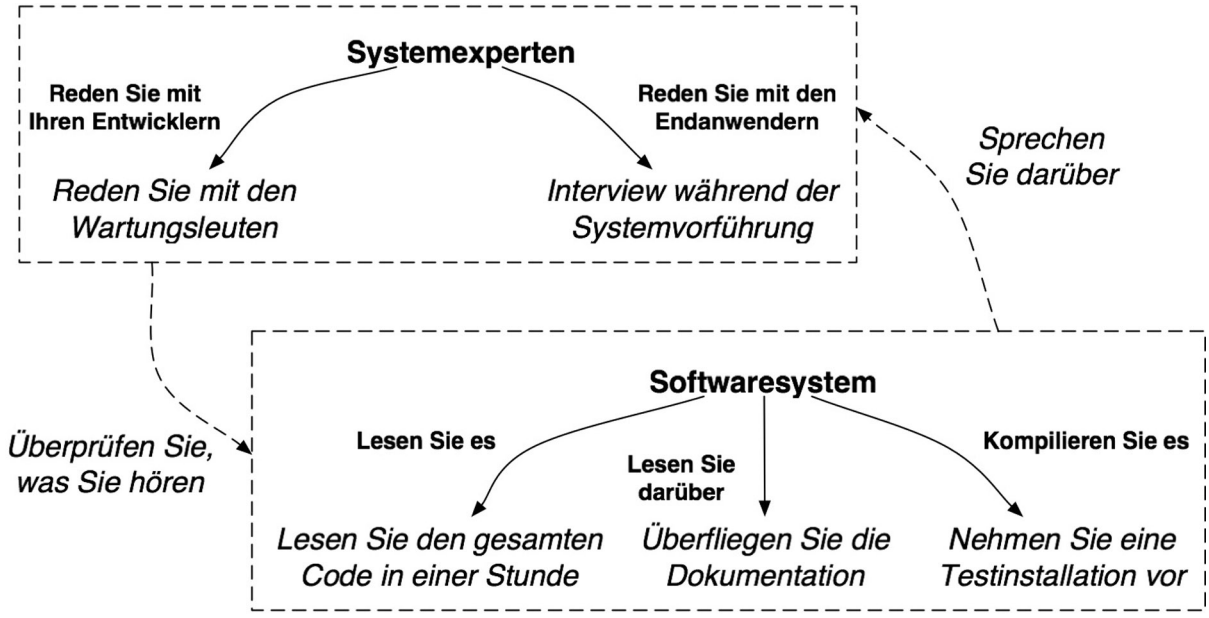

Abb. 3: Erster Kontakt

helfen, Stellen zu identifizieren, wo Code per „Copy&Paste" eingefügt oder wo Code aus verschiedenen Softwareversionen vermischt worden ist. Mit „Verantwortlichkeiten neu verteilen" (*Redistribute Responsibilities*) lassen sich Klassen mit zu vielen Aufgaben auffinden und einem Re-Engineering unterziehen. Und schließlich kann das Muster „Bedingte Anweisungen in Polymorphie transformieren" (*Transform Conditionals to Polymorphism*) dabei helfen, Verantwortlichkeiten neu zu verteilen, wenn ein objektorientiertes Design im Laufe der Zeit Schaden genommen hat.

Da eine detaillierte Beschreibung der Muster den Rahmen dieses Artikels sprengen würde, wollen wir uns im Folgenden mit einem einzelnen Muster-Cluster kurz näher befassen. „Erster Konkakt" umfasst – wie bereits gesagt – Muster, die nützlich sein können, wenn Sie sich das erste Mal mit einem Legacy-System befassen. Es gibt zahlreiche Einflussfaktoren, die man sich in diesem Zusammenhang bewusst machen sollte. Insbesondere sind Legacy-Systeme in der Regel *groß* und komplex, sodass es schwierig ist, einen Überblick über das System zu bekommen. Da die *Zeit knapp* ist, ist es wichtig, qualitativ hochwertige Informationen schnell zu bekommen. Darüber hinaus sind *erste Eindrücke gefährlich* – daher ist es wichtig, sich nicht auf eine einzelne Informationsquelle zu verlassen.

Als Informationsquellen stehen verschiedene Ressourcen zur Verfügung: der

Name	*Interview während der Systemvorführung*
Ziel/Absicht	Ein erstes Gefühl für die gewünschte Funktionalität eines Softwaresystems erhalten, indem man sich dieses demonstrieren lässt und die Person, die die Systemvorführung vornimmt, interviewt.
Problem	Wie können Sie eine Vorstellung von den typischen Anwendungsszenarien und den zentralen Features eines Softwaresystems erhalten?
Lösung	Sehen Sie sich das System im Betrieb an und interviewen Sie die Person, die die Vorführung macht. Beachten Sie dabei, dass das Interview selber mindestens so aufschlussreich ist wie die Demo.
Tipps	Die Anwender, die die Systemvorführung vornehmen, sind von entscheidender Bedeutung für die Ergebnisse dieses Musters – Sie sollten die entsprechenden Personen daher sorgfältig auswählen. Am besten lassen Sie sich das System mehrere Male von unterschiedlichen Personen demonstrieren.
Nebeneffekte	*Pro:* Legt den Fokus auf erprobte Features. *Kontra*: Zeigt nur Einzelberichte. *Schwierigkeiten:* Erfordert Interview-Erfahrung.
Beispiel	(*Beschreibung eines typischen Interviews...*)
Grundprinzip	Weil die Anwender von einem funktionierenden System ausgehen, nehmen Sie eine positive Grundhaltung ein, wenn sie erklären, was funktioniert. Der Interviewer kann präzise Fragen stellen, erhält präzise Antworten und kann dabei Expertenwissen über die Systemanwendung zu Tage fördern.
Allgemeine Anwendungsgebiete	Wird im Allgemeinen für die Evaluierung von Benutzungsschnittstellen verwendet.
Verwandte Muster	siehe auch *Customer-Interaction*-Muster (vgl. [RisOO])
Nächste Schritte	Führen Sie mehrere Runden von *Interviews während der Systemvorführung* mit unterschiedlichen Arten von Projektbeteiligten (*Stakeholder*) durch. Machen Sie das vor, während oder gleichzeitig mit den Mustern *Lesen Sie den gesamten Code in einer Stunde* und *Überfliegen Sie die Dokumentation*. Anschließend sollten Sie mit Ihren Wartungsfachleuten sprechen, um einige Ihrer Beobachtungen zu überprüfen.

Tabelle 1: Ein Muster im Überblick

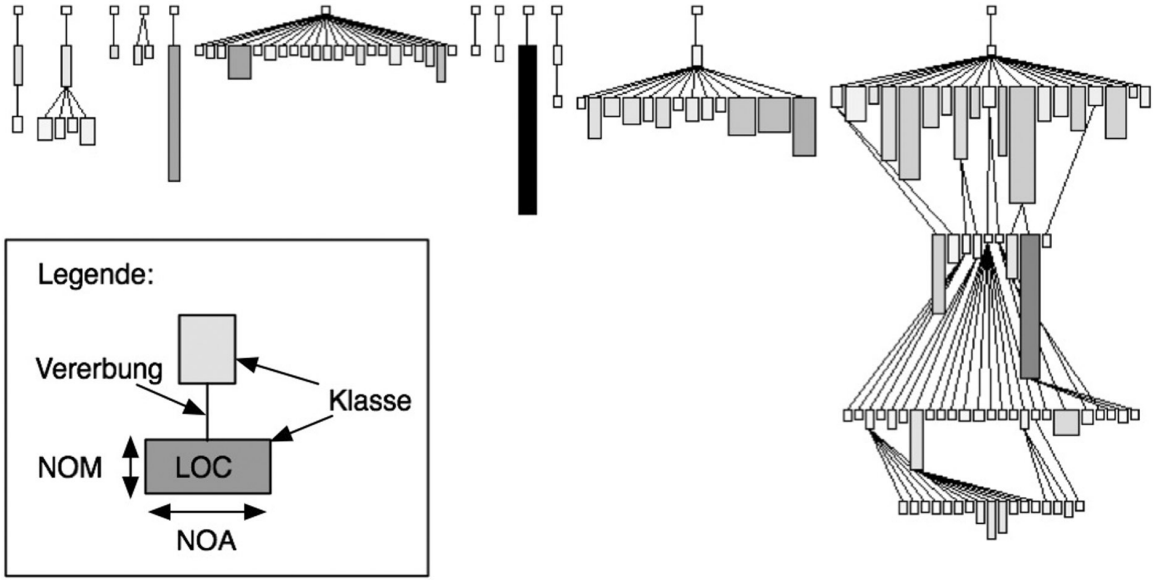

Abb. 4: Sicht auf die Systemkomplexität mit dem Werkzeug „CodeCrawler"

Quellcode, das laufende System, die Benutzer, die Wartungsmannschaft, Dokumentation, das Quellcode-Repository, Änderungsprotokolle, *Bug-Reports*, Testfälle usw. Auch wenn einige dieser Quellen fehlen oder unzuverlässig sind, muss man aufpassen, nichts frühzeitig zu verwerfen.

In Abbildung 3 sehen wir eine Übersicht der Muster aus diesem Cluster sowie ihre Abhängigkeiten untereindander. Wie bei jedem Muster-Cluster unterstützen sich die Muster gegenseitig, um die vorhandenen Einflussfaktoren zu zerlegen. Bei dem Cluster „Erster Kontakt" geschieht dies, indem das Wissen, das Sie von den Benutzern und Wartungsleuten erhalten, dem Wissen gegenüber gestellt wird, das Sie aus dem Quellcode ziehen können.

Tabelle 1 gibt einen Überblick über ein gut bekanntes Muster dieses Clusters. *Name* ist in der Regel eine Handlung, die ausgeführt werden soll; er drückt die Idee des Musters aus. Nicht jedes Muster ist in jedem Kontext relevant, sodass man sich über Folgendes im Klaren sein sollte: das *Ziel* des Musters, das *Problem*, das es löst, die zentrale Idee der *Lösung* sowie die mit ihm verbundenen *Nebeneffekte*. Bei dem hier betrachteten Muster dient eine Systemvorführung als Werkzeug, das dem Benutzer dabei hilft, sich auf die konkreten statt auf die abstrakten Eigenschaften der Anwendung zu konzentrieren, während er dem Software-Ingenieur typische Anwendungsfälle und Szenarien erklärt. Jedes Muster kann außerdem noch *Tipps*, *Varianten*, *Beispiele*, *Grundprinzipien*, *verwandte Muster* und Hinweise, was als Nächstes zu tun ist (*nächste Schritte*), enthalten. Die *allgemeinen Anwendungsgebiete* sind sehr wichtig, denn nur etablierte „Best Practices" lassen sich als echte „Muster" bezeichnen.

Re-Engineering-Werkzeuge und -Techniken

Es kommt schnell vor, dass man zu sehr auf den Einsatz von Werkzeugen setzt. Die Re-Engineering-Muster betonen daher stärker den Prozess als den Gebrauch von Werkzeugen („A fool with a tool is still a fool"). Nichtsdestotrotz können bestimmte Aktivitäten durch den Einsatz sorgfältig ausgewählter Werkzeuge rationalisiert werden. So kann insbesondere der Prozess des Reverse-Engineerings durch Werkzeuge, die aus dem Quellcode Modelle erzeugen, unterstützt werden.

Beachten Sie bitte, dass es dabei nicht darum geht, aus dem Quellcode UML-Diagramme zu generieren (10.000 Klassendiagramme tragen nicht unbedingt zu einem besseren Programmverständnis bei als 1.000.000 Zeilen Quellcode).

Auf der anderen Seite ist „Untersuchen ungewöhnlicher Entitäten" ein zentrales Muster des Clusters „Grundverständnis". Häufig sind es die sehr großen, die sehr kleinen, die am engsten gekoppelten, die am wenigsten oder am meisten erbenden Softwareeinheiten, die einem viel darüber verraten, wie ein Softwaresystem arbeitet. Solche „Ausreißer" können Hinweise auf Designprobleme sein, müssen es aber nicht. „CodeCrawler" ist ein Werkzeug, das – basierend auf direkten Metriken (vgl. [Lan05]) – einfache Visualisierungen von Softwareentitäten vornimmt. Eine polymetrische Sicht ist eine zweidimensionale Darstellung von Knoten (Entitäten) und Kanten (Beziehungen), die verschiedene metrische Werte auf Attribute der Knoten und Kanten abbildet. Beispielsweise können verschiedene Metriken auf die Größe, Position und Farbe eines Knotens oder auf die Stärke und Farbe einer Kante abgebildet werden.

Polymetrische Sichten können für unterschiedliche Zwecke erzeugt werden: grobkörnige Sichten, um globale Systemeigenschaften zu beurteilen, feinkörnige Sichten, um die Eigenschaften einzelner Softwareartefakte zu bewerten, und evolutionäre Sichten, um Eigenschaften im Zeitablauf zu beobachten.

Abbildung 4 zeigt eine grobkörnige Sicht auf die Systemkomplexität (vgl. [Lan03]). Die Grafik zeigt die Hierarchien in CodeCrowler selber. Jeder Knoten repräsentiert eine Klasse und jede Kante eine Vererbungsbeziehung. Die Länge eines Knotens repräsentiert die Anzahl der Methoden, seine Breite die Anzahl von Attributen und die (graustufige) Farbe die Anzahl der Codezeilen. Diese Sicht kann schnell dabei helfen, viele unterschiedliche Arten von „Ausreißern" ausfindig zu machen. Lange und schmale, isolierte, dunkle Knoten haben zum Beispiel viele

Methoden, viele Codezeilen und wenige Attribute und können darauf hindeuten, dass es sich hier um prozedurale Klassen mit langen algorithmischen Methoden handelt.

CodeCrowler setzt auf „Moose" auf, einer Re-Engineering-Umgebung, die eine gemeinsame Infrastruktur für verschiedene Reverse- und Re-Engineering-Werkzeuge zur Verfügung stellt (vgl. [Duc05-a], [Nie05]). Im Kern von Moose befindet sich ein gemeinsames Metamodell zur sprachunabhängigen Darstellung von Softwaresystemen. Um diesen Kern herum gibt es unterschiedliche Dienste für die verschiedenen Werkzeuge. Zu diesen Diensten gehören: das Evaluieren und Visualisieren von Metriken, ein Repository zum Speichern mehrfacher Modelle, ein Meta-Metamodell zum Anpassen des Moose-Metamodells sowie eine generische Benutzungsschnittstelle zum Browsen, zum Erstellen neuer Abfragen und zur Gruppierung von Elementen.

Im Rahmen von FAMOOS bzw. anschließend im Zusammenhang mit Moose wurde noch eine Reihe weiterer Werkzeuge entwickelt. Hierzu gehören:

- „Duploc": Mit diesem Werkzeug kann duplizierter Code in großen Softwaresystemen sprachenunabhängig aufgespürt werden (vgl. [Duc05-b], [Rie05]).
- „ConAn": Dieses Tool führt eine formale Begriffsanalyse durch, um implizite Verträge in objektorientierter Software aufzudecken (vgl. [Aré05]).
- „Van": Van analysiert die Versionsverläufe von Softwaresystemen, um die Entwicklungsrichtung aufzuzeigen (vgl. [Gir04].).
- „TraceScraper": Dieses Werkzeug analysiert die Laufzeitprotokolle von ausgeführter Software mit dem Ziel, Features mit Softwareartefakten zueinander in Beziehung zu setzen (zu korrelieren) (vgl. [Gre05]).

Literatur & Links

[Aré05] G. Arévalo, High Level Views in Object Oriented Systems using Formal Concept Analysis, PhD thesis, University of Berne, Januar 2005

[Chi92] E.J. Chikofsky, J.H. Cross, II. Reverse Engineering and Design Recovery: A Taxonomy, in R.S. Arnold (Hrsg.), Software Reengineering, S. 54-58, IEEE Computer Society Press, 1992

[Dem02] S. Demeyer, S. Ducasse, O. Nierstraß, Object-Oriented Reengineering Patterns, Morgan Kaufmann, 2002

[Duc99] S. Ducasse, S. Demeyer (Hrsg.), The FAMOOS Object-Oriented Reengineering Handbook, University of Bern, Oktober 1999

[Duc05-a] S. Ducasse, T. Girba, M. Lanza, S. Demeyer, Moose: a collaborative and extensible reengineering Environment, in: Tools for Software Maintenance and Reengineering, RCOST / Software Technology Series, S. 55-71, Franco Angeli, 2005

[Duc05-b] S. Ducasse, O. Nierstraß, M. Rieger, On the effectiveness of clone detection by string matching, in: International Journal on Software Maintenance: Research and Practice, 2005 (in Vorbereitung)

[ESP] ESPRIT Project 21975, Framework-based Approach for Mastering Object-Oriented Software Evolution, siehe: www.iam.unibe.ch/ ~scg/Archive/famous

[Gam95] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-Oriented Software, Addison Wesley, 1995

[Gir04] T. Girba, S. Ducasse, M. Lanza, Yesterday's weather: Guiding early reverse engineering efforts by summarizing the evolution of changes, in: Proc. of ICSM '04 (International Conference on Software Maintenance), S. 40-49, IEEE Computer Society Press, 2004

[Gre05] O. Greevy, S. Ducasse, Correlating features and code using a compact two-sided trace analysis approach, in: Proc. of CSMR 2005 (9th Eur. Conf. on Software Maintenance and Reengineering), IEEE Computer Society Press, 2005

[Kaz98] R. Kazman, S.G. Woods, S.J. Carriére, Requirements for integrating software architecture and reengineering models: Corum ii, in: Proc. of WCRE '98, S. 154-163, IEEE Computer Society, 1998

[Lan03] M. Lanza, S. Ducasse, Polymetric views – a lightweight visual approach to reverse engineering, in: IEEE Transactions on Software Engineering, 29(9), S. 782-795, September 2003

[Lan05] M. Lanza, S. Ducasse, Codecrawler – an extensible and language independent 2d and 3d software visualization tool, in: Tools for Software Maintenance and Reengineering, RCOST / Software Technology Series, S. 74-94, Franco Angeli, 2005

[Leh85] M.M. Lehman, L. Belady, Program Evolution – Processes of Software Change, London Academic Press, 1985

[Lie80] B. Lientz, B. Swanson, Software Maintenance Management, Addison Wesley, 1980

[Nie05] O. Nierstraß, S. Ducasse, T. Girba, The story of Moose: an agile reengineering environment, in: Proc. of ESEC/FSE 2005, LNCS, 2005 (in Vorbereitung)

[Rie05] M. Rieger, Effective Clone Detection Without Language Barriers, PhD thesis, University of Berne, Juni 2005

[Ris00] L. Rising, Customer interaction patterns, in: N. Harrison, B. Foote, H. Rohnert (Hrsg.), Pattern Languages of Program Design 4, S. 585-609, Addison Wesley, 2000

Fazit

Unter der Prämisse, dass „das einzig Beständige der Wandel ist", muss sich ein interessantes Softwaresystem weiterentwickeln, um interessant zu bleiben. Damit Architektur und Design mit den sich ändernden Anforderungen Schritt halten, müssen wir folglich in Re-Engineering-Maßnahmen investieren. Auch wenn kein System ist wie das andere, lassen sich verschiedene nützliche Re-Engineering-Muster identifizieren, die den Prozess vereinfachen, ein komplexes Legacy-System zu verstehen, seine Probleme zu identifizieren und es in ein flexibleres Design zu überführen.

Wir haben hier nur Muster beschrieben, von deren Erfolg wir uns persönlich überzeugen konnten. Die im Rahmen von FAMOOS entstandenen Re-Engineering-Muster stellen daher nur einen Ausgangspunkt dar und kein abgeschlossenes Werk. Wichtig ist, dass jedes Muster bewährte Vorgehensweisen dokumentiert, so wie sie Experten in bestimmten Feldern erfahren haben, und nicht reine Forschungsideen, die sich noch nicht im industriellen Kontext bewährt haben. Sicherlich gibt es hier reichlich Forschungsbedarf und -möglich-

keiten, z. B. bezüglich der Synergie von Werkzeugen und Re-Engineering-Mustern, aber man sollte beides nicht miteinander vermischen. Wir hoffen sehr, dass der Nutzen von Re-Engineering-Mustern – oder allgemeiner, Prozessmustern – im zunehmenden Maße anerkannt wird und Teil der *Best Practices* im Software-Engineering wird.

Danksagung

Wir möchten uns ganz herzlich bei der Swiss National Science Foundation für die finanzielle Unterstützung des Projekts „RECAST: Evolution of Object-Oriented Applications" (SNF Project No. 620-066077, 9/02 bis 8/06) bedanken. Unser Dank geht auch an **Markus Gälli** für zahlreiche Verbesserungsvorschläge an diesem Text.