Komponenten und Generative Programmierung

Jan Hinzmann
Universität Hannover
jan.hinzmann@stud.uni-hannover.de

29. Mai 2005

Zusammenfassung

In dieser Seminararbeit wird der Artikel „Components and Generative Programming“ von Krzysztof Czarnecki und Ulrich W. Eisenecker, der 1999 in den Proceedings der *Joint European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering* beim ©Springer-Verlag Berlin Heidelberg erschienen ist, besprochen.

1 Überblick über den Artikel

Der Artikel „Components and Generative Programming“ von Krzysztof Czarnecki und Ulrich W. Eisenecker, ist 1999 in den Proceedings der *Joint European Software Engineering Conference and ACM SIGSOFT International Symposium on the Foundations of Software Engineering* beim ©Springer-Verlag Berlin Heidelberg erschienen.
Er stellt den Paradigmenwechsel von der vorherrschen Praxis, bei der Komponenten manuell gesucht, adaptiert und zu Systemen zusammengestellt werden, zur Generativen Programmierung, bei der dies auf Grundlage einer abstrakteren Problembeschreibung und einem generativen System automatisch erfolgt, vor.

Czarnecki und Eisenecker kritisieren dabei zunächst, dass in der momentanen objektorientierten Technologie die Wiederverwendbarkeit und die Konfigurierbarkeit nicht auf effektive Weise unterstützt wird.
Anschließend zeigen sie, wie ein sogenannter *System family approach* dabei helfen kann, wiederverwendbare Komponenten zu entwickeln.
Dann wird beschrieben, wie mit Hilfe des sogenannten *Configuration Knowledge* das Zusammensetzen der einzelnen Komponenten zu einem Lösungssystem automatisiert werden kann.
Ferner vergleichen sie die Generative Programmierung mit der Einführung von austauschbaren Teilen und dem automatischen Zusammenbau in der

Automobilindustrie und geben anschließend ein Beispiel, das mit Hilfe von templatebasierter Metaprogrammierung den Ansatz der Generativen Programmierung verdeutlicht. Auf das Beispiel wird un dieser Seminararbeit allerdings nicht weiter eingegangen.

Im Folgenden werden nun die einzelnen Abschnitte des Artikels vorgestellt und besprochen.

2 From Handcrafting to an Automatied Assembly Line

In diesem ersten Abschnitt ist die Rede von einem Paradigmenwechsel. Dabei wird davon ausgegangen, dass bei der momentanen Softwareentwicklung zwar Komponenten benutzt werden, diese aber gesucht und angepasst werden müssen, um schlieslich zu einem fertigen System zusammengesetzt zu werden. Der Wechsel soll nun hin zur Generativen Programmierung erfolgen, bei der Komponenten automatisch gesucht und zusammengesetzt werden.

Für diesen Paradigmenwechsel sind die folgenden zwei Schritte nötig:

1. Vom *Single-System-Engineering* zum *System-Family-Approach*
2. Automatisierung durch die *Automated Assembly Line*

Der Artikel verdeutlicht die momentane Vorgehensweise im komponentenbasierten Softwareengineering mit Hilfe einer Metapher: Man stelle sich vor, man bekommt beim Kauf eines Autos alle Einzelteile geliefert und muss sich das Auto, von dem es nur eine Version gibt, selbst zusammenbauen. Ausserdem passen einige Teile nicht richtig und es ist Handarbeit nötig, um diese Teile anzupassen.
Durch den Einsatz von standardisierten Bausteinen, könnte die manuelle Anpassung einiger Teile entfallen und der Zusammenbau würde sich derart vereinfachen, dass man ihn automatisieren kann.
Diese Entwicklung ist vergleichbar mit der Industriellen Revolution, in der durch den Einsatz von austauschbaren Teilen Musketen gebaut wurden, es aber noch Jahrzehnte dauerte, bis diese Idee auch in anderen Industriezweigen Fuß fasste und für die Massenfertigung benutz worden ist. Dies war beispielsweise bei Henry Ford der Fall, der bis 1913 eine Fließbandproduktion errichtete, um der Nachfrage nach dem bekannten T-Modell nachzukommen.

Diese Metapher lässt sich auch auf die Softwaretechnik übertragen, denn selbst wenn man eine Bibliothek von wiederverwendbaren Komponenten hat, ist manuelle Anpassungsarbeit zwischen den Schnittstellen nötig, um ein Gesamtsystem zu realisieren.

Will man die heutige Situation berücksichtigen und bleibt in der Metapher, so wollen sicherlich nicht alle Kunden einen Ford T-Modell kaufen. Heutzutage gibt es ganze Modellreihen, wie etwa die Klassen bei Mercedes (S-/E-/C-Klasse), die aus standardisierten Teilen automatisch zusammengebaut werden können. In den Klassen gibt es jeweils verschiedene Features, die ein Kunde an- oder abwählen kann. Da man dem Kunden nicht zumuten kann, das ganze Automobil mit seinen tausenden von Einzelteilen selbst zu konfigurieren sind abstraktere Terme nötig. Durch diese können bestimmte Konfigurationen zusammengefasst werden und es reicht für den Kunden aus, wenn er die Klasse und eventuelle Extras spezifiziert (S-Klasse, alle Extras). Das Auto kann dann gebaut und ausgeliefert werden.

Damit dieser Mechanismus funktionieren kann, müssen die einzelnen Komponenten für den Einsatz in einer Produktlinie entwickelt worden sein. Ferner muss das Wissen, was nötig ist, um von einer abstrakten Beschreibung zu einer konkreten Umsetzung zu gelangen entsprechend modelliert werden. Dieses sogenannte *configuration knowledge* muss schließlich als Generatoren implementiert werden.

Der beschriebene Weg ist das, was in der Automobilindustrie passiert ist:
Das Prinzip von austauschbaren Teilen war die Vorraussetzung für die Einführung der *assembly lane* durch Ransome Olds (1901). Dieses Konzept wurde von Henry Ford 1913 für die Produktion des T-Modells aufgegriffen und durch industrielle Roboter in den 80er Jahren automatisiert.

3 „One-of-a-Kind“ Development

Bei den meisten objektorientierten Analyse- und Designmethoden (OOA/D-Methoden) liegt der Schwerpunkt auf der Entwicklung von Systemen, die eine bestimmte Aufgabe übernehmen können. In der Regel wird nicht der Aufwand betrieben, ein System zu erstellen, mit dem eine ganze Problemfamilie bearbeitet werden kann. Dies ist der Grund dafür, dass die Wiederverwendung von Komponenten solcher Systeme nicht richtig unterstützt wird.

Czarnecki und Eisenecker identifizieren im Folgenden vier wesentliche Defizite:

- In den OOA/D-Methoden wird nicht zwischen der Entwicklung für Wiederverwendung (*engineering for reuse*) und der Entwicklung mit Wedervervendung (*engineering with reuse*) unterschieden.
  Der gesamte Entwicklungsprozess sollte in zwei Teilprozesse unterteilt werden:

  Entwicklung für Wiederverwendung

In diesem Teilprozess werden weiderverwendbare Komponenten entwickelt. Der Fokus liegt somit auf Systemfamilien, nicht auch Einzelsystemen.

Entwicklung durch Wiederverwendung
In diesem Teilprozess werden die entwickelten Komponenten wiederverwendet, um ein konkretes Problem zu lösen.

- Es gibt keine *Domain scoping phase*.
  Die Domaine wird also nicht richtig untersucht. So kommt es zu dem Resultat, dass die Klasse des zu entwickelnden System nicht eindeutig identifiziert wird. So kommt es, dass nur „der eine Kunde“ berücksichtigt wird, nicht aber alle Stakeholder der Domaine. Das System kann dann im Anschluss nicht richtig auf neue Probleme angepasst werden, wenn grundlegende Aspekte, die zwar in die Systemfamilie gehören, aber für den speziellen Kunden nicht berücksichtigt worden sind zu nun unumstößlichen Designentscheidungen geführt haben.

- Ferner gibt es keine Unterscheidung zwischen der Modellierung der Variablilitäten eines Einzelsystems und der bei Systemfamilien.

- Schließlich gibt es keine implementierungsunabhängigen Mittel, um die Variabilität zu modellieren, da man sich beispielsweise schon beim Zeichnen eines UML-Diagramms zwischen Vererbung oder Aggregation entscheiden muss.

4 System Family Approach

Als nächster Abschnitt wird in dem Artikel der sogenannte *System Family Approach* eingeführt, bei dem keine Einzelsysteme mehr entwickelt werden. Das erklärte Ziel ist die Entwicklung für Systemfamilien. Hierzu muss zunächst zwischen der Entwicklung *für* und der Entwicklung *mit* Wiederverwendung unterschieden werden. Das erste nennt man auch *Domain Engineering* und das zweite *Application Engineering*.

Im wesentlichen wird der gesamte Entwicklungsprozess in diese beiden Teilprozesse aufgeteilt. Der erste Teil löst das domainenspezifische Problem allgemein, in dem er beispielsweise ein Framework zur Lösung von Problemen dieser Systemklasse entwickelt.
Der zweite Teil verwendet dann dieses Framework, um ein konkretes und domainenspezifisches Problem zu lösen.

Die beiden Teilprozesse bedingen sich gegenseitigt und es entsteht ein iterativer Gesamtprozess, in dem der zweite neue Anforderungen für den ersten

generiert. Während das Entwicklerteam des Prozesses für Wiederverwendung in der Rolle des Entwicklers bleibt, wechselt das zweite Team die Rolle vom Entwickler zum Kunden, sobald Eigenschaften verlangt werden, die das Framework des ersten Teams noch nicht unterstützt. Mit der nächsten Version des Frameworks, kann dann das konkrete Problem besser abgebildet werden.
Zusätzlich entsteht aber auch die Möglichkeit, künftige Probleme aus der Problemfamilie schneller abzuarbeiten. Es kann dann durch die Wiederverwendung Entwicklungsarbeit eingespart und Kosten gesenkt werden. Ausserdem wird die Stabilität erhöht, da die wiederverwendeten Komponenten bereits getestet sind.
Czarnecki und Eisenecker gehen nun noch weiter auf das Feld des *Domain Engineering* ein. Die Entwicklung für eine Domaine gliedert sich so in die drei Teilaufgaben:

- Domain Analyse
- Domain Design
- Domain Implementation

Bei der Analyse, gilt es zunächst die Grenzen der Domaine festzulegen, also zu sagen, welche Systeme und Eigenschaften der Domaine angehören und welche nicht. In einem zweiten Schritt werden dann die Features festgelegt, die das zu entwickelnde System unterstützen muss. In dieser Phase des Entwicklungsprozesses spielen nicht nur technische Aspekte eine Rolle, sondern auch politische und marktwirtschaftliche Interessen nehmen Einfluss auf die Entscheidungen.

In der anschließenden Design-Phase gilt es eine allgemeingültige Architektur zu entwickeln, die die Systemfamilie abdeckt und die identifizierten Anforderungen umsetzt.

Schließlich wird das Design in eine Implementierung umgesetzt und die Komponenten und Generatoren implementiert. Ausserdem muss die Umgebung entwickelt werden, die die Wiederverwendung erlaubt und die vom Benutzer erstellte Spezifikation für das konkrete Problem versteht. Diese soll dann in eine Konfiguration der Komponenten umgesetzt werden und anschliessend muss das Endsystem erstellt und ausgeliefert werden.

5 Problem vs. Solution Space and Configuration Knowledge

Den beschriebenen Prozess kann man in die drei Teile *Problem Space*, *Solution Space* und *Mapping*, wie in Abbildung 1 gezeigt, aufteilen.

Abbildung 1: Vom Problemraum zum Lösungsraum

Die vom Nutzer erstellte Spezifikation, die zum Beispiel in einer domainenspezifischen Sprache geschrieben sein kann, wird dem *Problem Space* zugeordnet. Zusammen mit dem sogenannten *Configuration Knowledge* ist es dann möglich, aus den wiederverwendbaren und konfigurierbaren Komponenten ein System zu erstellen, welches eine Lösung zu dem beschriebenen Problem darstellt.

Das Konfigurationswissen muss einige Bedingungen erfüllen:

- Es müssen ungültige Konbinationen von Kompontenten erkannt werden.
- Werden Angaben ausgelassen, sollen Default-Werte angenommen werden.
- Es müssen Abhängigkeiten unter den Komponenten erkannt werden.
- Es ist nötig, bestimmte Konstruktionsregeln zu beachten (beispielsweise muss eine Komponente vor einer anderen übersetzt werden).
- Es sollten Optimierungen durchgeführt werden.

Durch dieses Konzept kann der Nutzer bei der Beschreibung von Lösungen im Problemraum die Menge an Informationen liefern, die für ihn ausreichend ist und das System ergänzt die restlichen Informationen.

Dies wird in der analogen Welt deutlich, wenn jemand beispielsweise ein Auto kauft. Es reicht aus, sich für eine Marke und einen Typ zu entscheiden (z.B. Opel, Ascona, Bj 1980), um ein bestimmtes Auto kaufen zu können. Man muss nicht zusätzlich die ganzen Einzelteile spezifizieren.
Zu beachten ist hier noch, dass die Features im Problemraum zum Teil sehr abstrakt sein können und es keine direkten Gegenstücke (Komponenten)

im Lösungsraum geben muss. Das Feature kann dennoch durch Komposition/Aggregation/... bereitgestellt werden.
Dies ist ein wesentlicher Unterschied zwischen der Generativen und der Generischen Programmierung.
In der Generativen Programierung müssen die übergebenen Parameter nicht konkreten Komponenten im Lösungsraum entsprechen, sondern können abstrakte Begriffe sein, die durch das Configuration Knowledge beispielsweise zu einem Kompositum von Komponenten übersetzt werden.

Ausserdem kann die Spezifikationssprache so gehalten werden, dass eine Spanne der Genauigkeit erlaubt werden kann. Der Nutzer kann sein Problem recht allgemein schildern, allerdings kann er auch speziell werden. Die Spanne kann also beispielsweise von „Sportwagen“ bis hin zu einer genauen Beschreibung des geforderten Automobils inklusive der Einzelteile reichen.

6 Applications

In dem Artikel werden drei Anwendungen genannt, die das Konzept der Generative Programmierung verwirklichen.

Generative Matrix Computation Library Eine generative C++ Bibliothek, mit deren Hilfe sich Matrizenrechnungen realisieren lassen.

Generative Matrix Factorization Library Liefert einen Konfigurationsgenerator, mit dem sich verschiedene Instanzen aus der *LU Faktorisationsalgorithmusfamilie* (Gauss, Cholsky, LDL^T) erstellen lassen.

Generative Library for Statistics in Postal Automation Eine von Siemens Electrocom benutzte Suite für die erzeugung von Konfigurationsgeneratoren, mit denen sich verschiedene Timer, Zähler und statistische Algorithmen erstellen lassen.

7 Conclusions

In der Zusammenfassung des Artikels machen Czarnecki und Eisenecker deutlich, dass der beschriebene Paradigmenwechsel wie in der industriellen Fertigung einige Jahrzehnte in Anspruch nehmen kann. Weiter sprechen Sie davon , dass ein kultureller Wandel auf Seiten der Kunden, Berater und Anbieter von Software nötig ist, damit der Ansatz der Generativen Programmierung sich gegenüber der 'artistischen' Einzellösung durchsetzen kann. Der Einsatz von austauschbaren Software-Komponenten benötigt eine „product-line“ Architektur, nur so kann man schnell und einfach sagen, ob eine Komponente den Anforderungen eines Systems genügt oder auch nicht.

Hierzu bedarf es einer besseren architektonischen Standardisierung für die einzelnen Belange der verschiedenen Industrien, bevor die Idee der Softwarekomponenten wirklich durchstarten kann.

Wenn das Gesamtsystem aus den Komponenten manuell zusammengestellt werden kann, ist es auch möglich, diesen Vorgang durch Generatoren zu automatisieren. Diese Automatisierung ist der logisch nächste Schritt, wenn man eine plug-and-play Architektur etabliert hat. Hierfür entstehen natürlicherweise Kosten, genauso wie für die Entwicklung von wiederverwendbaren Technologien. Da sich allerding schon einige Standards in Form von wiederverwendbaren Komponenten und Architekturen etabliert haben (jakarta-commons-net: ftp, mail, ..., tomcat. cocoon, httpd, ...), diese frei verfügbar sind und industrielle Qualität erreicht haben, kann der „Break-Even-Point“ somit schneller erreicht werden.

Schließlich ist zu beachten, dass Komponenten immer einen Teil eines wohldefinierten Produktionsprozesses darstellen. So ist beispielsweise ein Backstein eine Komponente für den Hausbau und nicht für den Automobilbau. Für die Softwaretechnik spielt dies im Rahmen der Kriterien wie

- binär Format,
- Interoperabilität,
- Programmiersprachenunabhängigkeit
- ...

eine Rolle. Diese Kriterien sind immer an den Prosuktionsprozess gekoppelt.

8 Diskussion

Czarnecki und Eisenecker haben in ihrem Artikel die verschiedenen Aspekte der Generativen Programmierung erläutert.

Momentan werden oftmals Systeme für ein bestimmtes Problem entwickelt. Für neue Probleme werden neue, andere Lösungssysteme erstellt. Wenn nun ein Problem ähnlich zu einem bereits gelösten Problem ist, wäre es wünschenswert, die bereits erstellten Komponenten wiederzuverwenden, um Entwicklungsarbeit zu sparen und somit Kosten zu senken.

Die Herangehensweise hierfür ist der sogenannte *System-Family Approach* oder auch *Domain Engineering*, der Entwicklung für Systemfamilien an stelle von Einzelsystemen für spezielle Problematiken.

Ist man in der Lage, auf eine Bibliothek von wiederverwendbaren, anpassbaren und konfigurierbaren Komponenten zuzugreifen, kann man den Prozess der Systementwicklung weitestgehend automatisieren.
Es ist dann nötig, das vorliegende Problem in geeigneter Weise zu spezifizieren und anschließend können die benötigten Komponenten aus der nun vorliegenden Konfiguration erstellt und zu einem System zusammengesetzt werden. Wenn dieser Prozess zusätzlich automatisiert wird, nennt man das *Generative Progrmmierung.*

Für den Paradigmenwechsel sind also zwei Schritte nötig:

1. Beim Software-Engineering muss von der Vorstellung *ein* System zu erschaffen abgekommen werden. Stattdessen soll der Fokus auf der Entwicklung für *Familien von Systemen* gelegt werden.

2. Mit Hilfe von Generatoren können dann die jeweiligen Komponenten zu einem fertigen System über- und zusammengesetzt werden.

Momentan sind Komponenten verfügbar, die durch Konfiguration oder Anpassung leicht eingesetzt werden können, das Zusammenfügen zu einem Gesamtsystem ist dabei aber immer noch Handarbeit.
Dies soll nun durch die Generative Programmierung automatisiert werden und den Entwickler befähigen, sein Problem auf einer abstrakteren Ebene zu beschreiben. Ein generatives System kann dann die erstellte Problembeschreibung heranziehen, geeignete Komponenten konfigurieren oder gegebenenfalls sogar anpassen und schließlich das Lösungssystem oder die Komponente generieren.
Hierzu sind Standards nötig, an die sich ein solches System halten muss. Die Implementierungkomponenten müssen in ein definiertes Schema passen.

Es muss eine Zuordnung getroffen werden, mit deren Hilfe das generative System die abstrakten Anforderungen des Entwicklers in konkrete Konfigurationen von Komponenten und deren Konstellation zueinander übersetzen kann. Dieses „Mapping “ nennt man *configuration knowledge.*

Seit 1999 wurde der Artikel nach citeseer 20 mal zitiert; unteranderem von Don Batory in weiteren Arbeiten.
Don Batory zeigt in einer Fallstudie für den Einsatz von *product-line architectures* (PLAs) und domainenspezifischen Sprachen (DSLs) in einem Kommandier- und Kontrollsimulator für die Armee, dass durch den Einsatz der hier beschriebenen Methodik die Komplexität der Implementierung stark vereinfacht werden konnte.

Literatur

[1] Czarnecki, K. und Eisenecker, U.: Components and Generative Programming, Springer Verlag, 1999

[2] Czarnecki, K.: Overview of Generative Software Development

[3] Don Batory, Product-line architectures, aspects, and reuse (tutorial session), Proceedings of the 22nd international conference on Software engineering, p.832, June 04-11, 2000, Limerick, Ireland (link)

[4] Don Batory, A tutorial on feature oriented programming and product-lines, Proceedings of the 25th international conference on Software engineering, p.753, May 03-10, 2003, Portland, Oregon (link)

[5] Achieving extensibility through product-lines and domain-specific languages: a case study, ACM Transactions on Software Engineering and Methodology (TOSEM), v.11 n.2, p.191-214, April 2002 (link)