Komponenten und Generative Programmierung

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

21. Juni 2005

Zusammenfassung

In dieser Seminararbeit wird der Artikel „Components and Generative Programming“ von Czarnecki und Eisenecker besprochen. Der Artikel beschreibt einen Paradigmenwechsel in der Softwaretechnik. Wurden früher Speziallösungen für ein konkretes Problem entwickelt, liegt heutzutage der Fokus auf dem Domain Engineering. Hierbei geht man von dem Gedanken aus, dass konkrete Probleme einer Problemfamilie angehören, für die allgemeingültige Lösungen entwickelt werden können. Werden Systemkomponenten nun vor diesem Hintergrund entwickelt, können sie im Anschluß zu einem System zusammengesetzt werden, welches ein konkretes Problem löst und sie können wesentlich besser für Probleme aus der gleichen Problemfamilie wiederverwendet werden. Bei der Generativen Programmierung geht man dann noch einen Schritt weiter und versucht den Zusammenbau der einzelnen Komponenten zu einem Gesamtsystem zu automatisieren. Dann kann ein Entwickler sein konkretes Problem auf einer höheren Abstraktionsebene beschreiben und ein generatives System findet die geeigneten Komponenten und setzt das Lösungsystem zusammen.

1 Einleitung

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.

Der Artikel stellt einen Paradigmenwechsel von der vorherrschen Praxis, bei der Komponenten manuell gesucht, adaptiert und zu Systemen zusammengestellt werden, zur Generativen Programmierung vor, bei der dies auf Grundlage einer abstrakteren Problembeschreibung und einem generativen

System automatisch erfolgt.

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 fertigen 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 in dieser Seminararbeit allerdings nicht weiter eingegangen.

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

2 Der Paradigmenwechsel

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 schließlich 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. Übergang 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 in Generatoren implementiert werden, welche dann die einzelnen Komponenten zu einem Gesamtsystem verbinden.

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 Entwicklung von Einzelsystemen

In diesem Abschnitt beschreiben Czarnecki und Eisenecker den Ist-Zustand beim Softwareengineering.

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 die folgenden Defizite:

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

  Entwicklung für Wiederverwendung
  : In diesem Teilprozess werden wiederverwendbare 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 Systems nicht eindeutig identifiziert wird. So kommt es, dass nur „der ein 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 Variabilitäten zu modellieren. Beim Zeichnen eines UML-Klassendiagramms entscheidet man sich schon zwischen Vererbung oder Aggregation und legt so Implementierungsdetails fest.

In manchen Unternehmen wird die sogenannte *(1,2,n)-Regel* angewandt. Das bedeutet, dass für ein Problem zunächst eine Lösung gefunden wird, die genau dieses Problem löst. Tritt dann ein ähnliches Problem auf, wird für dieses wieder eine Lösung gefunden. Kommt nun allerdings anschließend wieder ein ähnliches Problem vor, so wird eine allgemeingültige Lösung entwickelt. Dies entspricht der Forderung nach Entwicklung für Systemfamilien.

Der letzte Punkt von Czarnecki und Eisenecker erscheint etwas gekünstelt, da das Zeichnen von Klassendiagrammen erst zu einem späteren Zeitpunkt erfolgen kann und naturgemäß nahe an der Implementierung ist. Es ist also nicht verwunderlich, dass ein Klassendiagramm Implementierungsdetails festlegt.

4 Entwicklung für Systemfamilien

Als nächster Abschnitt wird in dem Artikel der sogenannte *System Family Approach* eingeführt. Bei diesem Ansatz werden keine Einzelsysteme mehr entwickelt, sondern es wird gleich nach der allgemeingültigen Lösung gesucht. Das erklärte Ziel ist die *Entwicklung für Systemfamilien* und das konkrete Problem stellt nur eine Instanz der Systemfamilie dar. Hierzu muss zunächst zwischen der Entwicklung *für Wiederverwendung* 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 Problem der Domaine 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 gegenseitig 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 des nun vorhandenen Frameworks 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 Vom Problemraum zum Lösungsraum

Hat man den System-Family-Approach gewählt und wiederverwendbare Komponenten entwickelt, so können Anforderungen aus dem Problemraum durch das Konfugurationwissen automatisch in den Lösungsraum übersetzt werden.

Den beschriebenen Prozess kann man in die drei Teile

- *Problem Space*
- *Solution Space*
- *Mapping*

aufteilen(s. Abb. 1). Im Mapping ist das sogenannte *Configuration Knowledge* festgeschrieben. Das Mapping erlaubt die Übersetzung von Anforderungen, die im Problemraum beschrieben werden, in konkrete Lösungssysteme.

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. Allerdings scheint es bei einem solchen Auto auch nicht möglich, Details frei wählen zu können. Man müsste schon zu einer Werkstatt gehen, wenn man Umbauwünsche hat.
Die Features im Problemraum können zum Teil sehr abstrakt sein und es muss keine direkten Gegenstücke (Komponenten) im Lösungsraum

geben. Das Feature kann dennoch durch Komposition/Aggregation/... von vorhandenen Komponenten bereitgestellt werden. Dies leistet das Konfigurationswissen.
So ist klar, welche Teile zusammengesetzt werden müssen, wenn ein S-Klasse Mercedes mit allen Extras bestellt wird, obwohl es hierfür keine direkte Entsprechung im Lösungsraum gibt. Dies wird erst mit dem Zusammenspiel von Problemraum und dem Mapping möglich.
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.

Die Spezifikationssprache des Problemraums kann so gehalten werden, dass eine Spanne der Genauigkeit erlaubt werden kann. Der Nutzer kann sein Problem also recht allgemein schildern und die fehlenden Angaben werden durch Defaultwerte aus dem Konfigurationswissen ergänzt; oder er liefert eine detailliete Beschreibung seines Problems. Die Spanne kann also beispielsweise von „Sportwagen“ bis hin zu einer genauen Beschreibung des geforderten Automobils inklusive der Einzelteile reichen.

6 Zusammenfassung der bisherigen Ergebnisse

In ihrer 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.

Momentan werden oftmals noch Systeme für ein bestimmtes Problem entwickelt. Für neue Probleme werden dann 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 erstellt und zusammen mit dem vorliegenden Konfigurationswissen zu einem System zusammengesetzt werden.
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*.

7 Diskussion

Die bereits erwähnte *(1,2,n)-Regel* stellt meines Erachtens eine gute pragmatische Herangehensweise an das Problem der Wiederverwendbarkeit dar. Ein System family approach verursacht hohe Initialkosten, die sich erst durch Einsparungen nach mehrfach wiederholtem Einsatz bezahlt machen. Allgemeingültige Lösungen zu erstellen kann also sinnvoll sein, wenn man eine Domaine klar abgrenzen kann und es auch abzusehen ist, dass man noch mehrere Folgeaufträge aus diesem Bereich bekommt.
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.

Allerdings scheint es momentan doch so, dass Komponenten von verschiedensten Anbietern kommen und deshalb deren Bedürfnissen entsprechen. Diese Anbieter stellen die Komponenten zur freien Verfügung (z.B. jakarta-commons[4]) und ein Nutzer kann diese Komponenten konfigurieren oder sogar an seine speziellen Bedürfnisse anpassen und benutzen. Nun kann man andereseits seine eigenen Komponenten entwickeln, um sozusagen die Fäden in der Hand zuhalten, allerdings sollte man sich hier genau überlgen, ob man bereits gemachte Entwicklungsarbeit wiederholen sollte.

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.

Bei Komponentenbasierten Architekturen hat sich heute schon eine Konfigurations- bzw. Beschreibungssprache im XML-Format etabliert. Die einzelnen Komponenten können mit Hilfe weniger Zeilen XML-Code kofiguriert und zu einem System zusammengestellt werden. Für diese XML-Beschreibung lassen sich wiederum Tools (z.B. mit Eclipse/GEF[3]) entwickeln, sodass man in der Lage ist ein konkretes Problem grafisch zu notieren und sich eine Komponentenkonstellation „zusammenzuklicken“.

Komponenten stellen immer einen Teil eines wohldefinierten Produktionsprozesses dar. 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,
- Domainenzugehörigkeit

eine Rolle. Diese Kriterien sind immer an den Produktionsprozess gekoppelt. Lässt sich dieser klar abgrenzen, ist es durchaus möglich, geeignete Komponenten zu entwickeln und damit eine Problemfamilie zu lösen.

Dann ist man sicherlich auch in der Lage Werkzeuge zu erstellen, die diese Komponenten in geeigneter Weise unter Berücksichtigung des Konfigurationswissens zusammenstellen.

Die bis hier diskutierten Argumente lassen allerdigns einen wesentlichen Aspekt der Generativen Programmierung ausser acht: Die Codegenerierung. Solange die Komponenten als Kompilate vorliegen und lediglich durch XML-Beschreibungen konfiguriert und zusammen gestellt werden müssen, lassen sich gut Werkzeuge bauen, mit deren Hilfe die Problembeschreibungen generiert werden können.

Soll hingegen Code generiert werden gestaltet sich dies meines Erachtens als schwieriger. Hier sind Aspekte zu klären, in wie weit sich testen lässt, ob sich jeder generierte Code wie gewünscht verhält. Deshalb sehe ich es als erstrebenswerter an, die Konfigurierbarkeit von Komponenten soweit zu treiben, dass alle wünschenswerten Einstellungsmöglichkeiten beispielsweise in Property-Dateien externalisiert werden. Diese können dann durch ein Generatives System erzeugt werden. Weiterhin kann jeder Komponente bei Erhalt einer Konfiguration Tests durchführen, ob sie richtig konfiguriert worden ist. Zusätzlich kann das generative System als abstrakte Maschine entworfen werden, die neben der Generierung von Komponentenkonfigurationen auch die -zusammensetzung zu einem Gesamtsystem übernimmt.

Literatur

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

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

[3] Eclipse, an open platform for tool integration built by an open community of tool providers. (link)

GEF - Graphical Editor Framework (link)

[4] Collection of open source reusable Java components from the Apache/Jakarta community (link).