Generative Software-Entwicklung mit UML

Christian Donker
Thomas Stahl

b+m Informatik AG, D-24109 Melsdorf, Germany

Wolfgang Goerigk

Institut für Informatik und Praktische Mathematik
Technische Fakultät der Christian-Albrechts-Universität zu Kiel

0 Einleitung

Dieses Dokument liefert einen kurzen Einblick in die generative Software-Entwicklung mit UML. Der Ansatz der architekturzentrierten Software-Entwicklung ist eine wichtige Voraussetzung für die generative Software-Entwicklung, um die generierbaren Anteile identifizieren und spezifizieren zu können.

Wir werden die Vorteile der architekturzentrierten Software-Entwicklung erörtern sowie Grundlagen der generativen Software-Entwicklung erklären.

1 Architekturzentrierte Software-Entwicklung

Architekturzentrierte Software-Entwicklung ist als Generalisierung der generativen Software-Entwicklung zu betrachten. Bei der architekturzentrierten Software-Entwicklung wird ein Vorgehensmodell definiert; die generative Software-Entwicklung ist dann eine Ausprägung dieser unter Verwendung eines Generator-FrameWorks.

Im Mittelpunkt der architekturzentrierten Software-Entwicklung stehen sogenannte frei konfektionierbare, musterbasierte *Anwendungsfamilien*. Sie beinhalten fundiertes Spezialwissen, welches einer breiten Schicht von Anwendungsentwicklern zur Verfügung gestellt wird und als Grundlage verschiedener ähnlicher Entwicklungen dienen sowie weiterentwickelt und verbessert werden kann. Als *ähnlich* bezeichnen wir Anwendungen einer Anwendungsfamilie, die also im wesentlichen von gleicher Architektur sind.

1.1 Anwendungsarchitektur

Mit Anwendungsarchitekturen sind an dieser Stelle die prinzipiellen *horizontalen* Strukturen (Muster) und Schichten, wie z. B. ein Komponentenbegriff Model-View-Controller Schichtung einer Anwendung gemeint und nicht *vertikale* Strukturen, wie z. B. der konkrete Komponentenschnitt einer Anwendung. Eine Anwendungsarchitektur kann somit eine ganze *Familie* von Anwendungen mit gleichen Konzepten tragen; wiederkehrende Problemstellungen werden auf eine einheitliche Art und Weise behandelt.

1.2 Architekturzentriertes Design

Die Verwendung von architekturspezifischen Design-Mustern (siehe auch [3]) wird als *architekturzentriertes Design* bezeichnet. Die Notation erfolgt mittels der UML^{TM} unter der Verwendung von Stereotypen, siehe auch [4]. Auf diese Weise entsteht eine anwendungsarchitekturspezifische, semantisch stark angereicherte UML-Design-Sprache. Diese Design-Sprache hat die Aufgabe, den Weg von der Analyse zum Design, sowie vom Design zur Implementierung zu erleichtern und zu beschleunigen.

Bei einem klassischen objektorientierten Entwicklungsprozeß entsteht aus dem Analyse-Modell iterativ und inkrementell ein Design-Modell, welches so weit verfeinert und detailliert ist, daß es durch einen Standard Code-Generator in einen Implementierungsrahmen überführt werden kann. Ein solches Design-Modell wird auch im folgenden mit *Implementierungsmodell* bezeichnet.

Abbildung 1 stellt eine Scheibe des Wasserfallmodells dar (Mikrowasserfall). Die Y-Achse trägt als Einheit den Detaillierungsgrad, die X-Achse die des Informationsgehaltes. Auf dem Weg zur Implementierung ist ein kontinuierlicher Informationsgewinn, sowie eine Erhöhung des Detaillierungsgrades zu erwarten. Die Schraffur ist als Aufwand zu interpretieren.

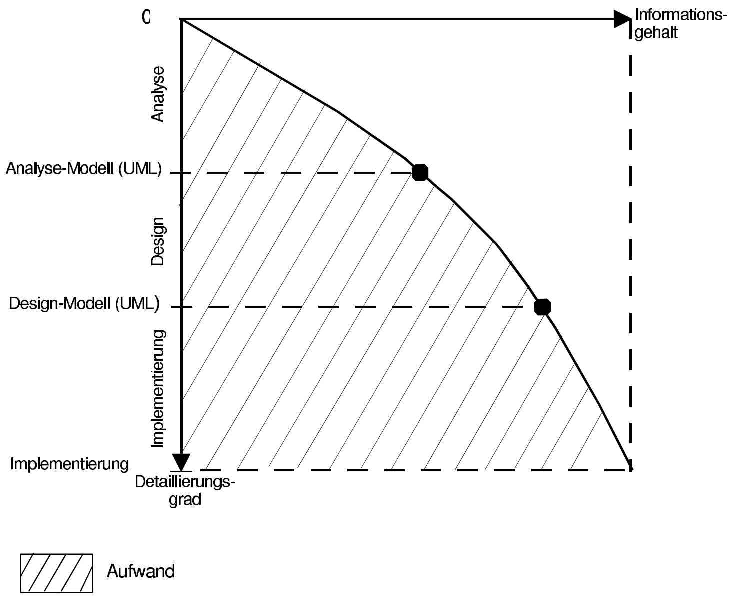

Abbildung 1. Informationsgewinn / Standard OO-Prozeß (aus [1])

Der Weg vom Analyse-Modell zum Design-Modell ist im allgemeinen sehr lang, und es gibt keine wohldefinierten Zwischenstopps. Des weiteren existieren Differenzen und Uneinheitlichkeiten in den Ergebnissen unterschiedlicher Designer.

Das Problem dabei ist, daß es zu keinem nutzbaren Gesamtmodell führt. Diesem Umstand sind nur intensives Coaching und Design-Reviews entgegenzusetzen.

Offensichtlich ist auch, daß ein so konzipiertes, konventionelles Design-Modell enorm detailliert ist. Die Kerninformationen gehen in dieser Fülle von Details unter.

Eine weitere unangenehme Eigenschaft eines solchen Implementierungsmodells ist, daß es sehr anfällig im Bezug auf Design-Änderungen ist. Das hat zur Folge, daß genau diese Änderungen vorzugsweise im Quelltext der Anwendung durchgeführt werden und dann das aus diesem Code resultierende, reverseengineerte Design-Modell als konsistente Dokumentation genommen wird, bzw. als Modell des nächsten Inkrements. Dieses Vorgehen wird deshalb auch als *Roundtrip-Engineering* bezeichnet, siehe dazu auch [7].

Packt man die Architekturmerkmale einer Anwendung in eine Design-Sprache und nutzt eine solche, um das Design-Modell zu entwerfen, so entsteht ein Design-Modell geringeren Detaillierungsgrades mit einem höheren Informationsgehalt, was deutlich in Abbildung 2 zu erkennen ist.

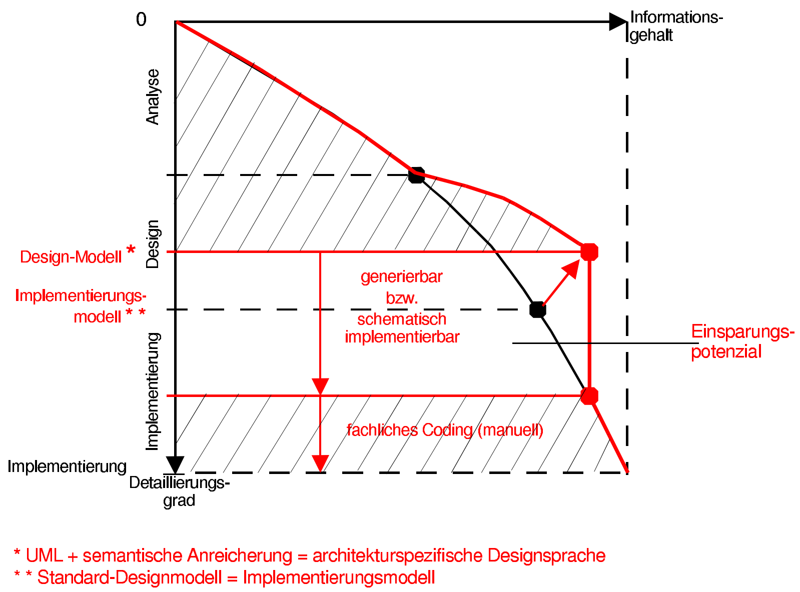

Abbildung 2. Informationsgewinn / Architekturzentrierter OO-Prozeß (aus [1])

Die Aufwärtsbewegung des Punktes des Design-Modells resultiert aus der Tatsache, daß mit der Design-Sprache komplexe Architekturmuster bzw. Design-Muster mit knapper UML-Notation ausdrückbar sind; z. B. impliziert die Verwendung eines Stereotyps `<<Entity>>` einer Enterprise JavaBean™ Design-Sprache eine Fülle von Klassen und Beziehungen, welche durch eine einzige knappe Benennung ausdrückbar ist.

Die Tatsache, daß die Semantik einer Design-Sprache auch eine Implementierung beinhaltet, hat die Rechtsbewegung des Punktes des Design-Modells zur Folge.

Ist nun der Ausgangspunkt ein Design-Modell, basierend auf einer Design-Sprache plus einer festen Architektur, so läßt sich die Implementierung des Design-Modells in zwei Teile zerlegen. Zuerst kann die Implementierung erfolgen, konzentriert auf die architektonischen Bestandteile der Anwendung. Dies erfolgt mittels der Bedeutung der Design-Sprache; die Umsetzung entspricht einem reinen „Funktionieren" der Implementierer. Der zweite Teil der Implementierung beschäftigt sich mit der Realisierung der sogenannten Fachlogik; dieser Abschnitt wird mit *fachlichem Coding* bezeichnet.

Betrachtet man in Abbildung 2 den Übergang vom Design-Modell zur Implementierung, so ist zu erkennen, daß die Implementierung der architektonischen Bestandteile der Anwendung, ausgehen vom Design-Modell kein Informationsgewinn bedeutet, da genau diese Informationen schon in der Semantik der Design-Sprache enthalten sind.

Ein weiterer Punkt, welcher nicht unerwähnt bleiben sollte, ist die optimale Trennung von Architektur, Design und fachlicher Implementierung. Dies impliziert ein klares und sauberes Rollenmodell mit entsprechenden Verantwortlichkeiten im Entwicklungsprozeß.

1.3 Syntax und Semantik der Design-Sprache

Die Syntax der Design-Sprache wird beschrieben durch eine Menge von erlaubten Stereotypen sowie eine Menge von Zusicherungen. Die Zusicherungen beschreiben, welche Stereotypen an welche UML-Modellelemente notiert werden dürfen und welcher Kontext vorhanden sein muß; z. B. daß ein Stereotyp <<Key>> nur an Attributen in einer Klasse mit dem Stereotyp <<Entity>> verwendet werden darf.

Die Semantik einer Design-Sprache wird unter Zunahme einer Architektur mittels eines Paares Referenz-Design und Referenz-Implementierung ausgedrückt. Das Referenz-Design umfaßt, soweit möglich, alle Elemente der Design-Sprache in einem sinnvollen Kontext, sowie alle möglichen Konstellationen. Die Referenz-Implementierung ist „die" Implementierung des Referenz-Designs zu der gegebenen Architektur. Durch diese Implementierung wird ein Programmiermodell vorgegeben, sowie die Interpretation der einzelnen Design-Elemente durch die entsprechende Bedeutung der Implementierung bzw. der Architektur.

Um Eindeutigkeit bei der Abbildung des Referenz-Modells auf die Referenz-Implementierung zu erhalten, sind sogenannte Projektionshilfen erforderlich, welche im allgemeinen als textuelle Informationen abgelegt werden; es handelt sich dabei um Quelltextfragmente aus der Referenz-Implementierung und eine Beschreibung, aus welchen Teilen des Referenz-Designs diese resultieren.

1.4 Anwendungsfamilie

Als *Anwendungsfamilie* bezeichnen wir ein Tupel, das aus einer architekturspezifischen Design-Sprache und einer konkreten Anwendungsarchitektur besteht. Mit Hilfe dieses Begriffs ist es möglich, auf eine elegante Art die architekturellen Gemeinsamkeiten ähnlicher Anwendungen zu extrahieren.

2 Generative Software-Entwicklung

Ausgehend von einem Design-Modell, spezifiziert mittels einer UML-Design-Sprache, kann aus diesem Modell die vollständige Implementierung der architektonischen Bestandteile erzeugt werden, wenn genau diese Informationen über die zu verwendende Architektur in Form von einer Anwendungsfamilie vorliegen.

In diesem Generat sind sogenannte *Protected Regions* vorgesehen, welche dann von einem Implementierer mit Fachlogik gefüllt werden. Beim Generieren ist es des weiteren möglich, ein Generat eines Vorgängerinkrements mit in den Generierungsprozeß einzubringen, um inkrementelle Entwicklung zu gewährleisten. Die Fachlogik aus einem Generat bleibt somit beim nächsten Generieren erhalten.

Dadurch, daß das Generat nur an gewissen, ausgezeichneten Markierungen das Hinzufügen von Fachlogik zuläßt, ist der Implementierer auf ein gegebenes Programmiermodell festgelegt. Des weiteren sind Design-Änderungen nicht mehr im Generat bzw. Quelltext durchführbar, diese <u>müssen</u> im Design vollzogen werden.

Außerdem ist durch diesen Ansatz eine sehr gute Wartbarkeit erreicht, da z. B. eine horizontale Änderung der Architektur nicht mehr in der vollen Breite der Anwendung nachvollzogen werden muß, sondern nur noch an genau einer Stelle, in der Anwendungsfamilie, und der Generierungsprozeß aus (vgl. Abschnitt 2.5) nur neu angestoßen werden muß.

2.1 Generator-FrameWork

Das Generator-FrameWork gehört zu der Klasse der sogenannten *White box frameworks*, siehe auch [9], d. h. eine gewisse Anzahl von Klassen müssen spezialisiert werden, um so die anwendungsfamilienspezifischen Anteile einzubringen. Dieser dynamische Anteil wird über den objektorientierten Polymorphismus zur Laufzeit an den generischen Teil des Generator-FrameWork gebunden; dies ist möglich, da als Entwicklungsumgebung Java™ gewählt wurde, was selbstverständlich <u>keinerlei</u> Auswirkungen auf die Zielsprache der Anwendung hat.

Um ein Design-Modell in Quelltext zu überführen, sind gewisse Informationen nötig, zum einen über die Design-Sprache und zum anderen, wie diese Elemente der Design-Sprache expandiert werden sollen. Diese Informationen bilden die Anwendungsfamilie. Die Definition der Design-Sprache wird mittels Spezialisierungen von FrameWork-Klassen realisiert. Die Quelltextfragmente sowie Expansionsregeln sind in sogenannte *Templates* verpackt (siehe 2.2).

2.2 Metamodell und Anwendungsfamilie

Eine Anwendungsfamilie ist definiert über eine Design-Sprache und eine konkreten Architektur. Technisch besteht die Anwendungsfamilie im Generator-FrameWork aus einer Menge von Spezialisierungen von Generatorklassen, welche aus der Definition der Design-Sprache resultieren und aus Templates, welche eine Implementierung der Architekturbestandteile enthalten. Somit sind die anwendungsfamilienspezifischen Anteile im Generator-FrameWork allgemein für <u>alle</u> Mitglieder dieser Familie von Anwendungen.

Spezielle Generatorklassen Die Spezialisierungen der Generatorklassen definieren die Design-Sprache. Über diese Klassen werden die erlaubten Konstrukte, Stereotypen spezifiziert, sowie die notwendigen Zusicherungen, welche von jedem Satz der Sprache erfüllt sein müssen.

Zusätzlich bieten diese Spezialisierungen dem Anwendungsfamilienentwickler die Möglichkeit, Funktionalitäten dem Template-Programmierer zur Verfügung zu stellen, falls die Expansion der Templates weitere Informationen aus dem Design-Modell benötigen.

Templates Die Templates einer Anwendungsfamilie bestehen aus einer Menge von Definitionen, gebunden an ein Modell-Element der UML plus Identifikator (Name des Templates). Diese Template-Definitionen sind angereichert mit Sprachelementen, welche für die Navigation auf Graphen und die Expansion von Zeichenreihen eigens entworfen worden sind.

Der Template-Programmierer ist somit in der Lage, ein Design-Modell zu analysieren und in Abhängigkeit dieses Modells Quelltext zu erzeugen. Zusätzlich hat der Programmierer innerhalb der Templates die Möglichkeit, Funktionalitäten aus den anwendungsfamilienspezifischen Generatorklassen zu nutzen. Somit ist eine mächtige und praxistaugliche Umgebung geschaffen, um genau die Implementierung der angewandten Muster im Design-Modell zu erzeugen.

2.3 Anwendungsdesign

Ein Anwendungsdesign als Satz einer Design-Sprache muß genau die Zusicherungen der Design-Sprache erfüllen, und es dürfen nur die definierten Elemente (Stereotypen) verwendet werden. Ein solches Anwendungsdesign enthält mehr Informationen bei geringerem Detaillierungsgrad, als ein konventionelles UML-Design-Modell, siehe dazu auch 1.2. Die Bedeutung eines Anwendungsdesigns, gebildet mittels einer architekturspezifischen Design-Sprache, ist somit über die Anwendungsfamilie (siehe 2.2) definiert.

Um dieses Design-Modell dem Generator-FrameWork zugänglich zu machen, wird es in XML/XMI (siehe [5] und [8]) überführt. Im Hause b+m wird zur Zeit mit dem UML-Werkzeug Rational Rose^{TM} gearbeitet, welches eine XMI-Exportfunktionalität zur Verfügung stellt.

2.4 Der generische Generator

Der generische Generator des Generator-FrameWorks ist allgemein für alle Anwendungen sowie für alle Anwendungsfamilien. Dieser umfaßt die Logik, die nötig ist, um Elemente der Design-Sprache mittels der Templates der Anwendungsfamilie zu expandieren und die Ausgabe zu produzieren. Die anwendungsfamilienspezifischen Anteile werden zur Laufzeit an den Generator gebunden; dies sind genau jene Spezialisierungen der Generatorklassen aus der Anwendungsfamilie (siehe 2.2).

Die Funktionalität des generischen Generators umfaßt weiterhin die Möglichkeit, Quelltext eines Vorgängerinkrementes mit einzubeziehen und diesen Quelltext genau in die „richtige " Position einzumischen, so daß sichergestellt ist, daß bei Neugenerierung keine Implementierungen verloren gehen.

2.5 Die Quelltextgenerierung

Das Anwendungsdesign wird in XMI überführt (siehe dazu [5]). XMI dient der standardisierten Darstellung eines UML-Satzes in XML (siehe dazu [8]). Diese XML/XMI-Repräsentation des Design-Modells erhält der generische Generator zusammen mit den Templates als Eingabe. Zusätzlich wird der Generator um spezialisierte Generatorklassen erweitert, in denen der anwendungsfamilienspezifische Anteil der Design-Sprache kodiert ist. Des weiteren enthalten diese Spezialisierungen Methoden, welche aus den Templates angesprochen werden können. Optional kann bei Generierung die Anwendung mit Quelltexten eines Vorgängerinkrements angereichert werden.

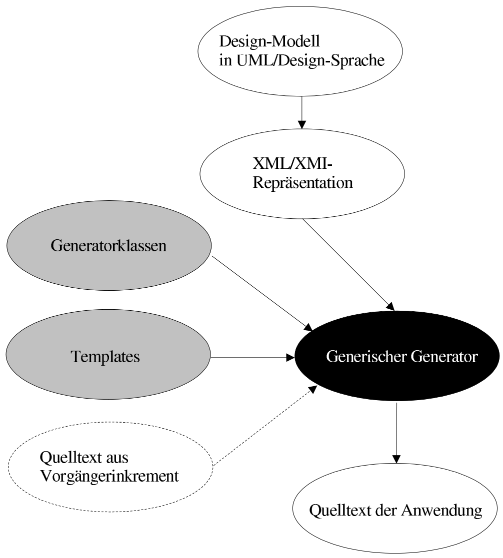

Abbildung 3. Ablauf

Der Ablauf der Generierung ist in Abbildung 3 dargestellt. Der generische Anteil ist schwarz dargestellt, der der Anwendungsfamilie grau.

3 Zusammenfassung und Erfahrungen

Der Ansatz der architekturzentrierten Software-Entwicklung ist im Hause b+m verfeinert und ausgearbeitet worden und wird in diversen Projekten mit großem Erfolg praktiziert. Durch den Einsatz einer Design-Sprache wird Expertenwissen für eine breite Schicht von Anwendungsentwicklern formal verfügbar gemacht, die dieses Wissen im Detail nicht mehr benötigen.

Durch den Einsatz eines Generators ist das Programmiermodell festgelegt, da in der Phase der Implementierung der Programmierer keine Design-Änderungen vornehmen kann, sondern diese im Design durchgeführt werden müssen. Es handelt sich somit um reines *Forward-Engineering*. Die nötige Flexibilität wird durch Einfachheit der Modifikation und Verbesserung des Designs und durch Schichtung und Modularisierung des gesamten Entwicklungsprozesses erreicht.

Das Generator-FrameWork wird zur Zeit bei zwei Projekten der Größe von 1000-2000 Personentagen eingesetzt. Es hat sich gezeigt, daß sich der Entwurf einer eigenen Designsprache und die Entwicklung spezieller Templates bezogen auf eine Anwendung dieser Größenordnung rentiert.

Die Erfahrungen zeigen, daß die architekturspezifischen Anteile, welche generierbar sind, bis zu 70% des gesamten Codes der Anwendung ausmachen können.

Literatur

1. b+m Informatik AG, `http://www.bmiag.de/de/_frame/b+m/kompetenz.htm`. *Software-Entwicklung bei b+m*, Juni 2001.
2. C. Donker. Generative Softwareentwicklung mit UML: Vom Design zum Codegenerator. Diplomarbeit, Christian-Albrechts-Universität zu Kiel, 2000.
3. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Entwurfsmuster*. Addison Wesley Longman Verlag GmbH, 1996.
4. OMG, `http://www.omg.org/technology/documents/formal/unified_modeling_language`. *Unified Modeling Language (UML)*, 1.3 edition, März 2000.
5. OMG, `http://www.omg.org/technology/documents/formal/xml_metadata_interchange`. *XML Metadata Interchange (XMI)*, 1.1 edition, November 2000.
6. I. Pavković. Generative Softwareentwicklung mit UML: Codegenerierung. Diplomarbeit, Christian-Albrechts-Universität zu Kiel, 2000.
7. B. Steppan. Prozessoptimierung, Roundtrip-Engineering mit Together 4.1 und JBuilder 4. *ix, Magazin für professionelle Informationstechnik*, 8:118, August 2001.
8. W3C, `http://www.w3.org/TR/2000/REC-xml-20001006`. *Extensible Markup Language (XML)*, 1.0 edition, Oktober 2000.
9. S. Zamir. *Handbook of Object Technology*. CRC Press LLC, 1999.