

Generative Softwareentwicklung in der Praxis

Alexander Knecht, sd&m

software design & management AG
Thomas-Dehler-Straße 27,
81737 München
Email: knecht@sdm.de

Abstract: Die klassische Softwareentwicklung in Projekten steht an der Schnittstelle von Anwendungsentwicklung und Framework immer wieder vor typischen Problemen. Führt man eine zusätzliche formale fachliche Schnittstelle ein, lassen sich diese Probleme lösen. Generatoren schlagen dabei die Brücke von der fachlicher Schnittstelle zum technischen Framework. Ein Fallbeispiel zeigt, wie den Kostentreibern des generativen Ansatzes erfolgreich begegnet werden kann.

1. Einleitung

Dieser Beitrag ging aus einem Projekt hervor, das bei sd&m auf Festpreisbasis durchgeführt wurde. Der umfassende Einsatz von Generatoren hat maßgeblich dazu beigetragen, den straffen Zeit- und Budgetrahmen zu halten, ohne dabei Qualitätseinbußen in Kauf zu nehmen. Die zugrunde liegenden Ideen und Ansätze werden zuerst projektneutral herausgearbeitet und anschließend in einem Fallbeispiel konkretisiert.

Abschnitt 2 beleuchtet prinzipielle Aspekte beim Generatoreinsatz. Abschnitt 3 stellt generative Entwicklung als Verfeinerung des manuellen Entwicklungsprozesses dar und vergleicht die Aufwände beider Ansätze. Abschnitt 4 skizziert ausgewählte Probleme der klassischen Softwareentwicklung in Projekten. Abschnitt 5 diskutiert generative Entwicklung im Projektkontext und löst die Probleme aus Abschnitt 4 auf. Abschnitt 6 beschreibt das Fallbeispiel und Highlights der konkreten Umsetzung. Abschnitt 7 fasst die Fallbeispiel-Erfahrungen zusammen und gibt einen kurzen Ausblick.

2. Single-Source Prinzip

Das Single-Source Prinzip beruht auf der Idee, Information nur einmal an einer Stelle zu definieren und immer dort wo sie benötigt wird zu referenzieren anstatt sie redundant zu halten. Die Vorteile liegen auf der Hand: Durch die Speicherung von Information an einer einzigen Stelle wird Redundanz vermieden und somit Änderbarkeit erhöht. Durch die Referenzierung der Information an anderen Stellen erhöht sich auch die Qualität, da Inkonsistenzen vermieden werden. Allerdings wird dies durch einen Verlust an Flexibi-

lität erkaufte, da man die Informationsmuster an den Verwendungsstellen nur soweit variieren kann, wie es bei der Musterdefinition vor(her)gesehen wurde.

Das Single-Source Prinzip entspricht der operationalisierten Form des „separation of concerns“ [Pa72], ist daher allgegenwärtig und kommt in den verschiedensten Bereichen zum Einsatz. Beispielsweise bei Begriffsdefinitionen oder Glossaren in Dokumenten, oder beim Software-Entwurf. Man schneidet die Funktionalität so zu Komponenten, dass eine Funktion nur in einer Komponente vorkommt, in der sie dann benutzt werden kann. In der Realisierung wird es am deutlichsten. Man definiert Klassen, Methoden, Attribute, Funktionen, Makros, um Muster zu hinterlegen, die an anderer Stelle wiederverwendet werden. Die Anwendung des Single-Source Prinzips lässt sich durch die folgende Regel charakterisieren:

$$a \bullet b + a \bullet c \rightarrow a \bullet (b + c) \quad (\text{SSP})$$

Dabei ist die $\bullet$ -Operation als Kopplung von Aspekten in einer Beschreibungseinheit und die $+$ -Operation als Aggregation von Beschreibungseinheiten zu betrachten. Hat man beispielsweise zwei HTML-Seiten, welche die gleichen Formatierungsstile verwenden, dann kann man diese gemeinsamen Stildefinitionen in ein Cascading-Stylesheet a auslagern und dieses Stylesheet in den einzelnen Seiten b und c einbinden. Der Browser sorgt dann für die Wiederverwendung der Stildefinition, so dass in der Darstellung später $a \bullet b$ bzw. $a \bullet c$ erscheint.

Generell bedingt die Anwendung des SSP gleichzeitig auch einen Wiederverwendungsmechanismus, der die kompakte Darstellung der Information auf der rechten Seite der Regel, die sich effizient pflegen lässt, wieder in die expandierte Form der linken Regelseite transformiert, die man eigentlich benötigt. Existierende Beschreibungssprachen besitzen in der Regel solche Mechanismen. Diese sind allerdings auf bestimmte Musterarten beschränkt. Für Muster, die nicht in der Beschreibungssprache definiert werden können, braucht man eine separate Musterbeschreibung und eine Vorverarbeitung durch einen Generator. Beispiele hierzu sind etwa der C-Präprozessor oder auch der Aspect-Weaver bei der Aspekt-orientierten Programmierung [Ki97].

3. Manuelle vs. generative Entwicklung

Der typische Entwicklungsprozess beim manuellen Ansatz ist in Abbildung 1 dargestellt. Der Entwickler liest ein Eingabedokument, trifft Entscheidungen wie er Richtlinien, Konzepte und Muster anwendet und schreibt ein Ausgabedokument, das eine Kombination aus der (aufbereiteten) Eingabe und den Entwicklungsmustern darstellt.

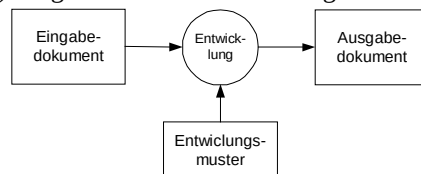


Abbildung 1 Manuelle Softwareentwicklung

Beim generativen Ansatz verschiebt sich der Fokus von der Erstellung des Ausgabedokumentes auf die Erstellung des Prozesses, durch den das Ausgabedokument er-

zeugt werden kann [CE00]. Bei dieser Automatisierung des manuellen Vorgehens ergeben sich, wie in Abbildung 2 dargestellt, zwei Aufgaben. Zum einen muss man die Eingabe formalisieren, damit sie maschinell gelesen werden kann. Man muss also eine Struktur bzw. Sprache definieren und die Eingabeinformation in dieser Sprache erfassen. Zum anderen müssen die Richtlinien, Konzepte und Muster formalisiert werden, um sie im Generator abzulegen und schließlich muss der Generator auch gebaut werden. Er enthält die Muster und deren Bezug zu Elementen der Eingabesprache.

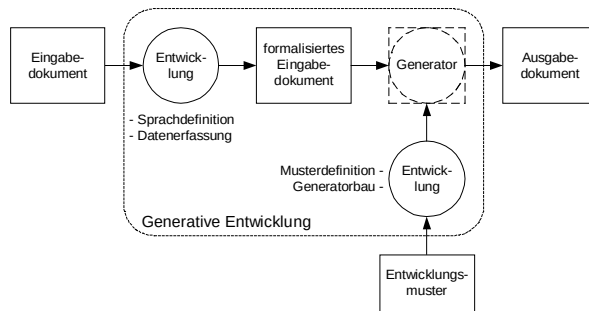


Abbildung 2 Generative Softwareentwicklung

Zum Vergleich der Aufwände der beiden Verfahren, wird angenommen, dass im Eingabedokument Elemente mit einer gemeinsamen Struktur vorkommen. Es lassen sich also eine Anzahl n von Instanzen dieser Struktur identifizieren, etwa alle Dialoge einer Anwendung oder alle Geschäftsprozesse. Die Kostenfunktion K_m für die manuelle Entwicklung lautet:

$$K_m(n) = n * E \quad (K_m)$$

Dabei steht E für die durchschnittlichen Entwicklungskosten einer Instanz. Die Kostenfunktion K_g für die generative Entwicklung ist dann:

$$K_g(n) = S + M + G + n * D \quad (K_g)$$

Dabei bezeichnen S , M und G die Kosten für Sprachdefinition, Musterdefinition und Generatorbau, und D die durchschnittlichen Kosten der Datenerfassung für eine Instanz. Wir unterstellen, dass der Generierungsaufwand für die Ausgabe vernachlässigbar klein gegenüber den Entwicklungsaufwänden ist und wir nehmen weiter an, dass die Kosten für die Datenerfassung kleiner sind als die Entwicklungskosten einer Instanz. Die Kostenfunktionen sind in Abbildung 3 dargestellt. Es zeigt sich, dass es einen break even point n^* gibt, ab dem der generative Ansatz günstigere Entwicklungskosten aufweist. Bei weniger Redundanz in der Eingabe erscheint der manuelle Ansatz günstiger.

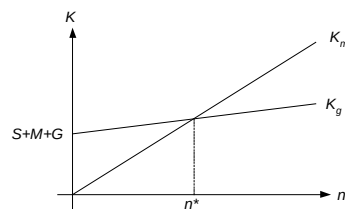


Abbildung 3 Kostenfunktionen für manuelle und generative Entwicklung

4. Softwareentwicklung im Projekt

Betrachten wir nun die Ergebnisse, die typischerweise in einem Software-Projekt entstehen. Wie in Abbildung 4 dargestellt ist da zum einen das Fachkonzept. Es beschreibt die fachliche Anwendungsstruktur und damit alle Anwendungselemente A_i , die den Umfang der Anwendung ausmachen. Dies umfasst beispielsweise Dialoge, Geschäftsprozesse, Front- und Backend-Systeme. Das DV-Konzept legt die technische Anwendungsstruktur fest. Es beschreibt die Architektur und Entwicklungsmuster für die Anwendungselemente im Fachkonzept. Im allgemeinen umfasst es ein Framework zusammen mit einem Nutzungskonzept NK , das die korrekte Verwendung des Frameworks erklärt. In der Realisierung wird das Fach- und DV-Konzept schließlich miteinander verknüpft und dabei die Anwendungselemente und das Framework unter Berücksichtigung des Nutzungskonzeptes in die Zielumgebung/-sprache transformiert.

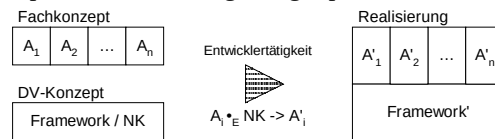


Abbildung 4 Manuelle Ergebniserstellung im Projekt

Vor besonderen Herausforderungen steht man immer wieder bei der Gestaltung der Schnittstelle zwischen Anwendungselementen und Framework. Hier müssen auf Basis unvollständigen Wissens Kompromisse getroffen und widersprüchliche Ziele gegeneinander abgewogen werden.

Stabilität: Ein kritischer Punkt ist immer der Übergang vom Entwurf in die Realisierungsphase. Ab jetzt wird in die Breite gegangen und die Kopplung von Anwendungselementen mit den technischen Schnittstellen des Frameworks vorgenommen. Damit werden Änderungen an der Framework-Schnittstelle sowie am Nutzungskonzept mit steigendem n immer teurer. Startet man zu früh mit der Realisierung, besteht die Gefahr, dass die Schnittstelle noch nicht stabil ist und Änderungen notwendig werden. Investiert man dagegen zuviel Zeit in die Konzeption der Schnittstelle, kann die Realisierung nicht starten und die Projektlaufzeit verlängert sich.

Granularität: Ein anderes Problem ergibt sich aus den zwei Sichten auf die Schnittstelle. Der Anwendungsentwickler hat es am einfachsten, wenn das Framework so spezielle Funktionen anbietet, dass er mit ein bis zwei Aufrufen die Entwicklungsaufgabe lösen kann. Dagegen soll ein Framework kleine, allgemeine Abstraktionen anbieten, die flexibel miteinander kombiniert werden können, dabei lediglich Spezialitäten einer technischen Schnittstelle kapseln aber ansonsten völlig unabhängig von Anwendungsaspekten sind.

Zwischen der Granularität der Abstraktionen in der jeweiligen Sicht besteht eine Lücke, die ich hier als *Wiederverwendungslücke* bezeichnen möchte. In der Praxis findet man oft Lösungen, welche die Lücke dadurch schließen, dass sie sowohl kleine, technische Basisabstraktionen und gleichzeitig mächtige, fachliche Spezialfunktionen in derselben Schnittstelle anbieten. Dies produziert ‚Eierlegendewollmilchsäue‘, die außerordentlich schwer zu warten sind. Eine andere Möglichkeit, die Lücke zu überbrücken, ist die Einführung von Schichten, so dass ein klarer, stufenweiser Übergang von Fachlichkeit zur Technik erreicht wird. Hier wiederholt sich zum einen die soeben beschriebene

Problematik beim Entwurf von Schnittstellen und zum anderen besteht die Gefahr, viele, eher schmale Schichten einzuziehen, um sich ja nichts für die Zukunft zu verbauen. Darunter leidet meistens die Performanz, aber auch die angestrebte Klarheit geht bei diesen ‚Lasagne-Architekturen‘ verloren. Bei einer zweckmäßig entworfenen Framework-Schnittstelle gibt es also eine Wiederverwendungslücke und es liegt in der Verantwortung des Anwendungsentwicklers unter Verwendung des Nutzungskonzeptes die Brücke zum Framework zu schlagen. Das Framework kann ihm dabei nur bis zu einem bestimmten Grad entgegenkommen.

Teamstruktur: Zum Schluss dieses Abschnitts noch zwei Bemerkungen zur Teamstruktur. Wenn die Entwicklung der Anwendungselemente auf mehrere Teammitglieder verteilt wird, fließen deren individuelle Erfahrung, Qualifikation, Motivation und Vorgehensweisen in die Realisierung mit ein. Die Anwendung der Richtlinien, Konzepte und Muster aus dem Nutzungskonzept variiert daher je nach Entwickler. Es ist deshalb davon auszugehen, dass das Nutzungskonzept immer wieder anders umgesetzt wird, wodurch spätere Änderung, Wartung und Pflege nicht gerade erleichtert werden.

Neben den schon erwähnten Kosten, tritt noch eine andere Schwierigkeit bei nachträglicher Änderung der Framework-Schnittstelle auf. Am Anfang der Realisierung steht für gewöhnlich eine Lernphase, in der sich die Anwendungsentwickler mit Fach- und DV-Konzept vertraut machen. Treten nun bei laufender Realisierung konzeptionelle Änderungen auf, müssen diese den betroffenen Entwicklern kommuniziert werden. Gerade in ‚heißen Phasen‘ der Realisierung fallen solche zusätzlichen Informationen, die außerhalb des normalen Arbeitsablaufes auftreten, gerne unter den Tisch, so dass später auftretende Inkonsistenzen programmiert sind.

5. Generative Entwicklung im Projekt

Der Einsatz von generativen Methoden in einem Softwareprojekt beruht auf der Anwendung des Single-Source Prinzips auf die Entwicklungsmuster, die im manuellen Ansatz direkt mit den Anwendungselementen verkoppelt werden. Der dazu benötigte Wiederverwendungsmechanismus wird über Generatoren realisiert. Abbildung 5 veranschaulicht den Einsatz von generativen Methoden als Verfeinerung von Abbildung 4.

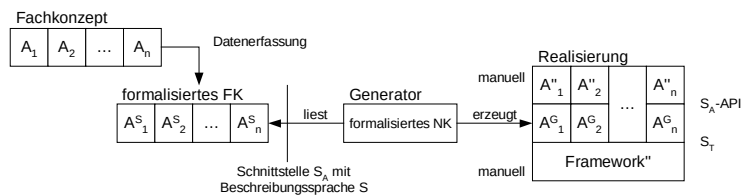


Abbildung 5 Generative Ergebniserstellung im Projekt

In der Praxis setzt man Generatoren dort ein, wo es sich lohnt. Die *Hebelwirkung* eines Generators

$$h_G = A^G / A^S$$

gibt Hinweise auf lohnende Bereiche, wobei A^S alle Instanzen eines Anwendungselements in der Beschreibungssprache S und A^G alle Ausgaben des Generators in der Zielsprache darstellt. Die Hebelwirkung sollte größer als Eins sein. Wenn es mehr Aufwand

ist, das Anwendungselement auf Basis von S zu formulieren, als es direkt in der Zielsprache zu realisieren, hat man nichts gewonnen. Der *Automatisierungsgrad* a_G eines Anwendungselements ergibt sich dann aus:

$$a_G = A^G / (A'' + A^G + \text{Framework}'')$$

Für jedes generativ entwickelte Anwendungselement braucht man eine Beschreibungssprache. Diese ist rein fachlich und unabhängig von technischen Aspekten. Sie sollte allgemein genug sein, um mögliche fachliche Änderungen erfassen zu können, aber speziell genug, um die Wiederverwendungslücke für Anwendungsentwickler möglichst klein zu halten.

Durch die Einführung der formalen Sprache S ergibt sich nun eine Trennung der klassischen Framework-Schnittstelle in eine fachliche Schnittstelle S_A und in eine technische Schnittstelle S_T . Bei vollständiger Automatisierung ($A'' = \emptyset$) wird die fachliche Schnittstelle aus Sicht der Anwendungsentwicklung nun zur Framework-Schnittstelle und die Aufgabe der Entwickler besteht ‚lediglich‘ in der Datenerfassung. Ist $A'' \neq \emptyset$ besteht die fachliche Schnittstelle aus der Beschreibungssprache S und einer Programmierschnittstelle S_{A-API} in der Zielsprache. Über diese fachliche API erfolgt die Kopplung von generiertem und manuell erstelltem Code. Die technische Schnittstelle ist nach wie vor auf Basis der Zielsprache realisiert. Die Trennung der klassischen Framework-Schnittstelle in S_A und S_T liefert einige nützliche Eigenschaften, um den Herausforderungen aus Abschnitt 4 zu begegnen.

Im Generator wird die Wiederverwendungslücke bezüglich S_T geschlossen. Er enthält die Entwicklungsmuster zur Abbildung von S_A auf S_T . Da die Entwicklungsmuster nicht mehr manuell vom Entwickler angewendet werden, können sie nun umfangreicher und aufwändiger gestaltet werden. Im Gegenzug kann S_T schlanker und flexibler sein. Das darunter liegende technische System wird direkter gekapselt, wodurch sich die Wiederverwendbarkeit der Schnittstelle erhöht. Die Trennung in S_A und S_T befreit zum einen von der Versuchung fachliche und technische Abstraktionen in einer Schnittstelle zu vermischen und zum anderen von der Notwendigkeit eine große Wiederverwendungslücke mit weiteren Schichten in der Zielsprache zu unterteilen. Eine solche Schichtung kann bei Bedarf auch durch Hintereinanderschalten mehrerer Generatoren geschehen.

Die Aufteilung in S_A und S_T ermöglicht auch eine zeitliche Trennung der Entwurfsentscheidungen, die für die Schnittstellen getroffen werden müssen. Für die Anwendungsentwicklung genügt es schon wenn die fachliche Schnittstelle stabil ist. Sobald die fachlichen Anforderungen klar sind und man sich von der prinzipiellen Machbarkeit einer technischen Lösung überzeugt hat, kann S_A festgelegt werden und die Entwicklung in voller Breite beginnen. Zur Erstellung der technischen Schnittstelle, Framework und Generaten bietet sich hier eine iterative Vorgehensweise an. Anfangs wird nur das Nötigste realisiert, um der Anwendungsentwicklung frühzeitig Feedback zu geben. Nach und nach kann man dann die technische Schnittstelle verfeinern und die Funktionalität auf Framework und Generate verteilen. Technische Entwurfsentscheidungen können gefahrlos frühzeitig getroffen werden, da sie später unabhängig von den bereits entwickelten Anwendungselementen änderbar sind.

Die Aufteilung in S_A und S_T ermöglicht ebenfalls eine räumliche Trennung der Entwurfsentscheidungen. Die Generate bilden eine Pufferzone, die Anwendungs- und Frameworkentwicklung vor Schnittstellenänderungen der jeweils anderen Seite schützt. Gerade beim Einsatz von neuen, zum Teil nicht ganz stabiler Technologie erlebt man immer wieder unvermutete Eigenheiten, die zu Workarounds und Änderungen der tech-

nischen Schnittstelle führen oder eine andere Frameworkverwendung bedingen. Die Anwendungsentwicklung bleibt davon unbehelligt. Dasselbe gilt auch umgekehrt: Falls die fachliche Schnittstelle geändert wird, kann die neue Semantik für gewöhnlich mit den Abstraktionen der technischen Schnittstelle formuliert werden. Dies bringt Ruhe in ein Projekt, in dem Anwendungs- und Frameworkentwicklung parallel betrieben wird, da Änderungen nicht mehr zwischen den Teams kommuniziert werden müssen, sondern nur noch mit der Generatorentwicklung.

Da die fachliche Schnittstelle weitgehend technikunabhängig ist, muss man kein Experte mehr für die eingesetzte Technologie sein, um Anwendungen zu entwickeln. Dies eröffnet Spielräume für Projektmanager in der Mitarbeiterdisposition.

6. Ein Fallbeispiel

Als Fallbeispiel dient ein Projekt, das bei sd&m mit der hier beschriebenen Verfahren erfolgreich durchgeführt wurde. Es wurde eine Extranet-Anwendung auf Basis von Java-Server-Pages und Servlets gebaut, die über 800 HTML-Seiten umfasst. Davon waren über 100 unabhängige, formularbasierte Dialoge mit ähnlichen Funktionen, sich in der Darstellung allerdings stark unterschieden und außerdem noch einige Seiten, die über ein Content-Management-System redaktionell gepflegt wurden. Die restlichen Seiten waren statisch. Alle Seiten folgen einem Styleguide, der von einer Medienagentur festgelegt wurde. Allerdings bestand die Anforderung, den Styleguide nachträglich noch ändern zu können. Die Anwendung ist mandantenfähig und unterstützt verschiedene Zielgruppen. Jede Zielgruppe hat ihre eigene Navigationsstruktur, über die eine spezielle Auswahl der Dialoge ermöglicht wird. Die fachliche Spezifikation lag in Form von PowerPoint Folien vor, mit denen alle Dialogseiten skizziert wurden. Bei Projektbeginn war schon abzusehen, dass sich die fachlichen Anforderungen noch stabilisieren mussten. Die Zielplattform war IBM RS/6000 unter AIX mit einem WebSphere Application-Server und einer DB2 Datenbank. Als Content-Management-System wurde Lotus-Notes verwendet. Die Entwicklungsumgebung lief unter Windows NT. Es wurde IBMs Visual Age for Java mit dem WebSphere Test Environment eingesetzt.

Die Projektlaufzeit betrug zehn Monate, wobei insgesamt etwa 15-20 Mitarbeiter beteiligt waren, die sich auf drei verschiedene Standorte verteilten. Das System ist seit November 2001 produktiv. Die folgende Tabelle gibt einen quantitativen Eindruck der Anwendung gemessen in Quellcode-Zeilen.

	n	A_S	A''	FW	A_G	Hebel	A-Grad
lines-of-code pro Instanz eines Anwendungselements							
Dialoge	202	47	159	34	1053	22,4	81,4%
Datentypen	98	10	0	22	167	16,7	83,9%
Menüpunkte	472	1	0	0,3	29	29,0	95,7%
lines-of-code über gesamte Anwendung							
Gesamt		12260	32216	51549	229087	18,6	70,4%

6.1 Sprachdefinition und Datenerfassung

Im Fallbeispiel werden drei Anwendungselemente auf Basis von XML formalisiert: Dialoge, fachliche Datentypen und die Menüs der Navigationsstruktur. Ein Dialog besteht aus ein oder mehreren Dialogseiten, die eine Menge von Dialogfelder enthalten. Die Dialogfelder unterteilen sich in reine Anzeigefelder für Texte und Bilder, Schaltfelder für Buttons, Datenfelder für Formulare sowie Gruppenfelder, die wiederum selbst Dialogfelder enthalten können. Des weiteren gibt es Dialogkommandos, welche die möglichen Anwendungsfälle im Dialog deklarieren und über Buttons aufgerufen werden können. Zur Erfassung und Bearbeitung der XML-Dialogdefinitionen wurde ein spezieller Editor auf Basis des Vektorgrafikprogramms ‚Visio‘ realisiert. Jedes XML-Element wurde als Visio-Shape dargestellt; seine Attribute konnten über einen ‚Eigenschaften‘-Dialog geändert werden. Es wurde ein pixelgenauer Maßstab zugrunde gelegt, in dem die Dialogfelder positioniert wurden. Über eine spezielle Symbolleiste wurden die Generatoren aufgerufen, die ihre Ausgabe direkt in die Testumgebung schrieben. Damit wurden sehr kurze ‚Edit-Compile-Test‘ Zyklen erreicht, gemäß den ‚Anforderungen an eine SEU‘ in [Li01]. Es war die Aufgabe der Anwendungsentwickler, die fachliche Spezifikation auf PowerPoint-Folien mittels des Dialogeditors zu erfassen.

Die fachlichen Datentypen wie Text, Aufzählung, Datum, Zeit und Geld sind im Framework hinterlegt. Spezielle Text- oder Aufzählungstypen, die sich nur durch bestimmte Parameter unterscheiden, wurden direkt als XML-Dokument gepflegt. Die Definition der vielen Aufzählungstypen wurde durch die Fachabteilung selbst durchgeführt, so dass der Erfassungsaufwand lediglich in der Nachbereitung der XML-Datei bestand.

Jede Zielgruppe hat seine eigene Menüstruktur. Die Menüpunkte legen Menütex, die Seitenüberschrift, einen Teil der URL sowie den aufzurufenden Dialog fest. Für die Navigation wurde auf Kundenwunsch ein Redaktionssystem auf Lotus-Notes Basis realisiert, mit dem die Menüs durch die Fachbereiche gepflegt werden.

Generell bietet der Editorbau bzw. die Verlagerung der Erfassung in die Fachbereiche gute Einsparmöglichkeiten für den Aufwand D in der Kostenfunktion K_g .

6.2 Musterdefinition und Generatorbau

Zur Entwicklung eines Generators wurde eine Generator-Definition in vier Schritten erarbeitet, aus der der eigentliche Generator wiederum erzeugt wurde. (1) *Prototyp-Erstellung*: Zuerst wird ein Prototyp der Generator-Ausgabe erstellt und in der Zielumgebung getestet. (2) *AT-Analyse*: Dabei werden im Ausgabe-Prototyp die anwendungsspezifischen Begriffe (A) identifiziert und somit von den rein technisch bedingten Textbereichen (T) getrennt. Die Notation A bzw. T ist an die „Software-Blutgruppen“ 0, A, T, AT aus [Si99] angelehnt. (3) *Muster-Definition*: Hierbei werden die A- und T-Teile in zwei Dateien ausgelagert und Redundanzen zusammengefasst. Die A-Stellen werden im T-Teil parametrisiert und in einem XML-Dokument zusammengefasst. Die Parameternamen aus dem T-Teil tauchen als Attributnamen im XML-Dokument wieder auf, um den Bezug zum A-Teil herzustellen. (4) *Generator-Definition*: Dazu werden die Muster-Definitionen über Pfadausdrücke mit den gewünschten XML-Knotenmengen verknüpft und die Reihenfolge der Musteranwendung bestimmt. Des weiteren wird ggf. benötigte

Transformationslogik von XML-Inhalten in Generator-Ausgaben hinzugefügt und die Ausgabedateien festgelegt.

Im Fallbeispiel wurden alle Generatoren mit dem Generator-Generator *Egg* erzeugt [Kn01] und dadurch der Aufwand G in Kostenfunktion K_g signifikant reduziert. Dieses Werkzeug erhält die Generator-Definition als Eingabe, die als ‚Quellcode-Stylesheet‘ wirkt. Im Prinzip funktionieren sie wie XSL-Stylesheets, jedoch können die Muster eins zu eins aus der Muster-Definition in Schritt (3) verwendet und die Transformationslogik in Java implementiert werden. Außerdem werden sie kompiliert und nicht interpretiert. Die Anbindung der XML-Elemente der Eingabe erfolgt wie in XSL über Pfadausdrücke. Von XSL wurde Abstand genommen, wegen unübersichtlicher Muster- und Transformationsdefinition, eingeschränkter Programmiermöglichkeiten, geringer Performanz und fehlender Whitespace Kontrolle in der Ausgabe.

6.3 Sandwich-Pattern

Das Sandwich-Pattern in Abbildung 6 ist ein Entwurfsmuster, das zur Kopplung von manuellem und generiertem Code bei Dialogen eingesetzt wurde. Die manuell erstellen Klassen bilden eine Klammer um die generierte Klasse in der Mitte. Eine *Framework-Klasse* stellt die Schnittstelle bereit, die für alle Dialoge gleich ist. Diese Klassen befinden sich im Dialog-Paket des Frameworks. Eine *generierte Klasse* ist dialogspezifisch. Sie enthält das Wissen aus der Dialogdefinition, deren Inhalte zur Laufzeit verfügbar gemacht werden sollen. Diese Information wird per Generator in die generierte Klasse transformiert. In der *konkreten Klasse* können nun individuelle Anpassungen für den Dialog vorgenommen werden, die nicht durch das der Dialogdefinition zugrundeliegende Datenmodell abgedeckt werden.

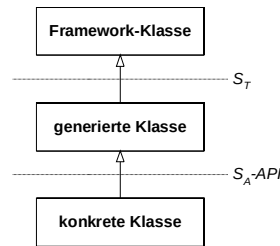


Abbildung 6 Kopplung von manuellem und generiertem Code

Das Zusammenspiel erfolgt über virtuelle Methoden. Benötigt man im Framework Informationen aus der konkreten Dialogdefinition, dann definiert man eine abstrakte Methode in der Framework-Klasse, die in der generierten Klasse mit dem gewünschten Wissen aus der Dialogdefinition implementiert wird. Die durch den Generator erzeugte Vorgabe kann individuell für einen Dialog in der konkreten Klasse wieder überschrieben werden. Die generierte Klasse weiß übrigens, dass die konkrete Klasse von ihr ableitet. In den Signaturen der generierten Klassen werden für Verweise direkt konkrete Klassen verwendet. Dies erspart unschöne Downcast-Operationen in der konkreten Klasse.

Der Vorteil dieser Aufteilung liegt darin, dass Änderungen in der Dialogdefinition oder im Generator durch wiederholtes Generieren der mittleren Klasse in die Anwen-

dung transportiert werden können, aber gleichzeitig durch virtuelle Methoden die generierte Funktionalität individuell angepasst werden kann, ohne generierten Code anzufassen.

7. Zusammenfassung und Ausblick

Zusammenfassend betrachtet hat sich der Generatoreinsatz im konkreten Projekt zum Fallbeispiel sehr bewährt. Die Vorteile der verbesserten Änderbarkeit, Qualität und verkürzten Entwicklungszeit ergaben sich aus der Anwendung des Single-Source Prinzips auf Entwicklungsmuster und der Einführung der formalen fachlichen Schnittstelle. Die Kommunikation wurde damit zuerst im Team und später auch mit dem Kunden präziser. Die Abhängigkeit zwischen der Anwendungs- und Frameworkentwicklung wurde durch die Einführung des Generatoren-Teams entkoppelt und gepuffert. So war es möglich, fachlich frühzeitig in die Breite zu gehen und gleichzeitig technische Entscheidungen spät zu treffen und diese auch noch nachträglich zu verändern. Die Qualifikation der Entwickler konnte dabei im Projektverlauf noch wachsen, ohne ein Projektrisiko einzugehen. Letztlich konnte das Team früher als geplant wieder abgebaut werden, da aufgrund der Hebelwirkung der Generatoren weniger Code manuell erstellt werden musste.

Generell bietet der generative Entwicklungsansatz hohes Wiederverwendungspotenzial im Bereich der Beschreibungssprachen. Gerade in betrieblichen Anwendungen ähneln sich die Strukturen sehr stark. Es gibt praktisch immer eine Oberfläche mit Dialogen, Plausibilitätsprüfungen für Benutzereingaben, sowie Geschäftslogik im Anwendungskern, der die Dialoge mit der Datenbank verknüpft. Aus einem vorhandenen Pool lässt sich sicherlich schnell eine Beschreibungssprache finden, die nur noch wenig auf das aktuelle Projekt angepasst werden muss. Des weiteren lassen sich Werkzeuge zum Generator- und Editorbau entwickeln, die mit der Sprache parametrisiert sind und so die Fixkosten in Kostenfunktion K_g drastisch reduzieren. Da die Entwicklungsprozesse in einer Firma im Allgemeinen sehr ähnlich sind, lassen sich diese Werkzeuge gut wiederverwenden. Tatsächlich konnten weite Teile des Fallbeispiels schon in einem anderen Projekt wiederverwendet werden. Die Adaption auf weitere Projekte ist geplant.

Literaturverzeichnis

- [CE00] Czarnecki, K; Eisenecker, U. W.: Generative Programming - Methods, Tools, and Applications. Addison-Wesley, June 2000
- [Ki97] Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J.-M.; Irwin, J.: Aspect-oriented programming. In ECOOP'97---Object-Oriented Programming, 11th European Conference, LNCS 1241, pages 220--242, 1997.
- [Kn01] Knecht, A.: Egg - Ein Generator Generator für Quellcode-Stylesheets. www.generia.de/egg.pdf, Juli 2001.
- [Li01] Lind, J.: Iterative Software Engineering for Multiagent Systems - The MASSIVE Method. Vol. 1994 of Lecture Notes in Computer Science, Springer Heidelberg, May 2001.
- [Pa72] Parnas, D. L.: On the criteria to be used in decomposing systems into modules. Communications of the ACM, 15(12):1053-1058, December 1972.
- [Si99] Siedersleben, J.; Albers, G.; Fuchs, P.; Weigend, J.: Schriftenreihe Informatik, Segregating the Layers of Business Information Systems. In (Donohoe, P Hrsg.) Software Architecture, 1999; S. 389-404.