1. Einleitung und Überblick

Als einer der Vorteile eines objektorientierten Entwicklungsprozesses wird die Durchgängigkeit der Methode von der Analyse über das Design bis zur objektorientierten Implementierung genannt [Jacobson 93, Graham 93]. Dieser Artikel stellt einen Rahmengenerator (*GENCXX)* für C++ Klassen- und Methodenheader vor und diskutiert ihn unter den Aspekten der Durchgängigkeit des objektorientierten Entwicklungsprozesses, der Codequalität - speziell beim "programming in the large" - und der Wartbarkeit der im Entwicklungsprozeß entstehenden Ergebnisse.

Der Artikel ist ein Erfahrungsbericht aus einem großen, durchgängig objektorientiert durchgeführten Projekt und möchte ein paar praktische Tips für die Umsetzung von objektorientierten Spezifikationen in die ebenfalls objektorientierte Realisierung geben. Unsere Diskussionen mit vielen anderen Entwicklern haben gezeigt, daß ähnliche Generatoren selten eingesetzt werden, obwohl man sich mit ihnen eine Menge lästiger Routinearbeit ersparen, und die Qualität erheblich steigern kann. Die positiven Erfahrungen mit einem solchen Werkzeug und ein paar Tips, was man sinnvollerweise durch Tools generieren läßt und was nicht, sollen hier weitergegeben werden.

Zunächst wird ein kurzes Ausschnitt aus einer Spezifikation vorgestellt. Im Kapitel 3. werden seine Eigenschaften und Leistungen des Programmrahmengenerators *GENCXX* diskutiert. Es wird beschrieben, welche Funktionalität man sinnvollerweise nicht in einen Generator für Coderahmen einbauen sollte. Behandelt werden auch Aspekte wie Qualitätsverbesserung, Wartung und schnelle und bequeme Erzeugung von Testdummies. Kapitel **Fehler! Verweisquelle konnte nicht gefunden werden.** beschreibt kurz die Implementierung des Generators. Man kann daraus den Aufwand für eine Implementierung in einer anderen Umgebung ableiten. Kapitel *Kann man GENCXX kaufen? - Portierungshemmnisse*

GENCXX wurde für eine spezielle Entwicklungsumgebung und einen speziellen Kunden entworfen. Daher wurde in *GENCXX* einiges Wissen über die Umgebung integriert, das sich nur mit erhöhtem Aufwand hätte herausabstrahieren lassen. Ein universeller Generator für jedes C++-Projekt ist entweder weniger leistungsfähig oder erheblich kostspieliger. *GENCXX* wurde auf die Projektsituationen an folgenden Punkten spezialisiert:

- *Typkonzept* - Der Generator macht Gebrauch von Namenskonventionen eines speziellenTypkonzeptes.
- *Klassenbrowser* - Der Generator erzeugt Header und Marken für einen speziellen Klassenbrowser.

? *Codestandards* - Der Generator muß an die für das Projekt gültigen C++-Richtlinien für Code, Header etc. angepaßt werden. Das kann auch die Trennung von Klassen in eine Header- und eine Datenhaltungsklasse bedeuten.

5. schildert die Erfahrungen, die mit dem Generator in einem großen Projekt gemacht wurden.

2. Dokumentenorientierte Spezifikation objektorientierter Systeme

Vor der Realisierung sollte ein System in einer Analyse- und einer Designphase eingehend untersucht und sein Verhalten vollständig spezifiziert sein. Objektorientierte Methoden halten sich zu gute, daß sich mit ihnen dieser Prozeß durchgängig von der Problemanalyse bis zur Implementierung ohne Brüche gestalten läßt [Graham 93, Jacobson 93].

In der Analyse- und Designphase arbeiten die Entwickler an Objektmodellen, die sie stetig verfeinern. Objektmodelle sind somit beim objektorientierten Vorgehen auch die Grundlage für die Realisierung. Dieses Vorgehen wird bei uns durch die dokumentenorientierte Softwareentwicklung [Denert 93a] unterstützt. Objektmodelle werden bei uns in Form von Texten beschrieben. Diese Texte sind jeweils mit soviel syntaktischer Struktur versehen, daß eine sinnvolle Auswertung der Dokumente mit Texttransformationswerkzeugen möglich ist. Man kann natürlich auch passende CASE-Tools für die Spezifikation der Objektmodelle verwenden. Die benötigte syntaktische Struktur wird dort nicht in Form von Schlüsselworten, sondern durch die Struktur der Datenbank des CASE-Tools repräsentiert. Dieser Ansatz ist jedoch aus unserer Sicht wesentlich teurer, unbequemer und weniger flexibel.

2.1. Beispiel für Spezifikationssprache

Hier soll als Beispiel ein kurzer Ausschnitt aus einem unserer Spezifikationsdokumente gezeigt werden. Die Spezifikationssprache TOLEDO [HöTe 93, BarDen 93] stellt Objekte in einer lesbaren und verständlichen Form dar. Dabei werden die meisten Aspekte eines Metamodells für objektorientierte Systementwicklung abgedeckt. Abbildung 1 zeigt einen Ausschnitt aus einer TOLEDO-Spezifikation.

In der Abbildung 1 wird sichtbar, daß die üblichen Komponenten eines Objektmodells wie Vererbung, Attribute und Funktionen eines Objektes durch entsprechende Schlüsselworte (IST_EIN, HAT, KANN) gekennzeichnet sind.

Bei der dokumentenorientierten Arbeitsweise wird die Softwareentwicklung als eine Folge von Texttransformationsschritten verstanden. Von einem formal wenig strukturierten Text (zum Beispiel Vorstudie, Problemanalyse) gelangt man schließlich zu stark formalisierten Texten (z.B. Programme).

```
JEDE          Flugreservierung
IST_EINE      Reservierung
BESCHREIBUNG  Eine FlugReservierung belegt ein Flugangebot fuer mehrere
              Personen, die zusammen eine Reise gebucht haben.

HAT           Reise               --->      Reise

HAT           FlugAngebot         --->      FlugAngebot

HAT           Klasse              :         FlugKlasse

HAT           AnzahlPersonen      :         PersonenZahl
              Die Zahl der Personen, die an dem Flug teilnehmen

KANN          PreisBerechnen
BRAUCHT       Steuersatz          :         DsProzent
ERGIBT        Preis               :         DMK
EFFEKT        Der Preis berechnet sich aus AnzahlPersonen die beteiligt sind
              mal ( ^FlugAngebot.Verkaufspreis + die VerkaufspreisDifferenz der
              jeweiligen ^Klasse ) + der jeweils gültige Steuersatz fuer den
              Vertrag.

              PSEUDOCODE
              Hier kann man noch Pseudocode eintragen, wenn man schon in der
              Spezifikation etwas festhalten möchte.
```

Bild 1: Ausschnitt aus einer TOLEDO Spezifikation[1]

2.2. Nutzen für die Rahmengenerierung

Transformationswerkzeuge erleichtern und unterstützen die Übergänge zwischen den einzelnen Phasen wie Analyse, Design und Realisierung. Der noch vorzustellende Generator *GENCXX* ist ein solches Texttransformationswerkzeug - und zwar ein relativ einfaches - das Objekttyp-Spezifikationen, wie die in Abbildung 1 in Klassenrahmen umsetzt. Dafür wird eine definierte syntaktische Struktur der Spezifikation benötigt.

Auch für objektorientierte CASE-Tools gibt es Rahmengeneratoren und sogar Code-Editoren, so daß schließlich "auf Knopfdruck" das Programm synthetisiert wird. Dort kann man jedoch meist keinen Einfluß auf das nehmen, was in den Rahmen erscheint. Auch ist das Meta-Modell selten ohne größeren Aufwand anpaßbar. Der im folgenden vorgestellte Weg

[1] Beispiel angelehnt an [Denert 93b]

erlaubt wesentlich mehr Freiheitsgrade für die Entwicklung. Die Vor- und Nachteile der Arbeit mit CASE-Tools sollen hier nicht weiter diskutiert werden - Eine solche Diskussion findet sich z.B. bei Denert [Denert 93] in ausführlicher Form..

Für die Zwecke eines Rahmengenerators sollte die Spezifikation mit genau *soviel* syntaktischer Struktur vorliegen, daß man einen Parser bauen kann, der daraus die relevanten Teile herausfiltert. Verwendet man ein CASE-Tool, das eine offengelegte Datenbankstruktur verwendet, kann man dasselbe Ergebnis mit etwas mehr Mühe auch in Form einer Datenbankauswertung erreichen. Unsere dokumentenorientierte Arbeitsweise ist also nicht zwingende Voraussetzung für ein Tool, wie den im folgenden vorgestellten Rahmengenerator.

3. Leistungen von *GENCXX*

3.1. Aufgabe des Generators

Der Generator *GENCXX* erstellt aus Objekttypspezifikationen, wie den in Abbildung 1 dargestellten Spezifikationen, C++ - Klassenrahmen und Header. Diese Aufgabe ist prinzipiell einfach, da ein Objekttyp meist in genau eine Klasse umgesetzt werden kann. Schon im Fachkonzept werden die Methoden und ihre Schnittstellen typgenau spezifiziert, so daß die wesentlichen Informationen für einen Rahmen vorliegen.

3.2. Wo der Generator stark ist

Ein Rahmengenerator soll den Entwickler von den langweiligen und damit oft fehlerbeladenen Routinearbeiten entlasten, ihm aber keinerlei unerwünschte Vorschriften machen, wie die Algorithmen aussehen, mit denen der spezifizierte Effekt einer Methode zu erreichen ist.

Solche langweiligen Arbeiten, die *GENCXX* übertragen wurden, sind:

? *Ausfüllen von Headern*

  Es wird die komplette Spezifikationsinformation in den Header übernommen, wie Methodenbeschreibung, Parameter, Pseudocode etc. Dadurch wird auch erreicht, daß die Header garantiert zur Spezifikation konsistent sind. Wie später noch zu zeigen ist, kann dies auch im Wartungsfall sichergestellt werden. Der Generator erspart also nicht nur die Arbeit des Ausfüllens sondern auch den Qualitätssicherungsaufwand für die Sicherstellung der Konsistenz.

? *Instrumentieren mit Trace-Anweisungen*

Trace-Anweisungen sind ein nützliches Mittel zum Debuggen von Code in der Testphase. Trace-Anweisungen erlauben es, Aufrufe und Werte von Variablen einfach zu verfolgen. Die entstehenden Trace-Protokolle sind meist wesentlich bequemer zu benutzen, als ein Debugger. Trace-Anweisungen werden für das Betreten einer Methode, für sämtliche Parameterwerte und für das Verlassen von Methoden generieren. Trace-Anweisungen sollten *bedingt compilierbar* generiert werden, da sie sonst das Laufzeitverhalten stark verschlechtern.

? *Instrumentieren mit Typchecks für Parameter*

Mindestens während der Testphase ist es praktisch, jeden Eingabeparameter einer Methode einem Typcheck zu unterziehen. Fehler werden auf diese Weise so schnell wie möglich erkannt. Das Debuggen wird erleichtert. Möglich wird dies dadurch, daß alle Klassen mit einer Methode *check()* versehen werden, die einen Selbsttest des Objektes vornimmt. Typchecks in Anwendungskernobjekten werden nur in der Testphase wirklich benutzt, da sie sonst das Laufzeitverhalten stark verschlechtern - sie sollten daher *bedingt compilierbar* generiert werden.

? *Rahmen für Exception-Handling*

Die Rahmen werden mit *try-catch* Statements [EllStr 91] für ein minimales Exception Handling ausgestattet.

? *Kommentare für Klassenbrowser*

Verfügt man über einen Klassenbrowser, der nicht selbst die nötige syntaktische Intelligenz zur Auswertung von Quellen hat, so kann man Informationen, die für die Aufbereitung durch den Browser benötigt werden, mit in die Rahmen generieren.

? *Ungarische Notation*

Ungarische Notation ist in der C- und C++-Programmierung geläufig. Sie kann problemlos in einen Rahmengenerator mit integriert werden.

Abbildung 2 zeigt Ausschnitte aus einem generierten Methodenrahmen. Dieser ist einem Projektbeispiel entnommen und enthält daher syntaktische Elemente zum Beispiel für einen spezifischen Klassenbrowser, die in anderen Programmierumgebungen überflüssig sein könnten.

Die obige Liste erhebt keinen Anspruch darauf, alles zu enthalten, was machbar ist. Die obige Grundausstattung von Methoden hat sich jedoch bei uns über viele vor allem große Projekte bewährt. Die meisten Elemente sind auch für andere Programmierumgebungen einsetzbar.

```
/******************************************************************************/
// METHODNAME:
//           Flugreservierung :: PreisBerechnen
//
// ABSTRACT: einzeilige (60 Zeichen) Beschreibung der Methode
//
// AUTOR:     sd&m, WK ( Sun May 29 17:54:58 1994 )
//
// DESCRIPTION:
//..generate.DESCRIPTION.Flugreservierung.PreisBerechnen
//.  Der Preis berechnet sich aus AnzahlPersonen die beteiligt sind
//.  mal ( ^FlugAngebot.Verkaufspreis + die VerkaufspreisDifferenz der
//.  jeweiligen ^Klasse ) + der jeweils gültige Steuersatz fuer den
//.  Vertrag.
//.
//..endgenerate
//
// SEE_ALSO:
//
// EXCEPTIONS:
//..generate.EXCEPTIONS.Flugreservierung.PreisBerechnen
//..endgenerate
//
// INTERNAL:
//..generate.INTERNAL.Flugreservierung.PreisBerechnen
//. PSEUDOCODE:
//. Hier kann man noch Pseudocode eintragen, wenn man schon in der
//. Spezifikation etwas festhalten möchte.
//..endgenerate
//
// PARAMETERS:
//..generate.PARAMETERS.Flugreservierung.PreisBerechnen
//.-    Steuersatz            IN    const
//..endgenerate
//
// RETURNVALUE:
//..generate.RETURNVALUE.Flugreservierung.PreisBerechnen
//.    DMK
//..endgenerate
//
// $1$
// SYNOPSIS:
// $-2$
/******************************************************************************/
```

```
//..generate.METHODHEADER.Flugreservierung.PreisBerechnen
DMK
 Flugreservierung :: PreisBerechnen (
                const DsProzent                 &    Steuersatz,
  )
//..endgenerate
 { /* $END$ */
   try {
//..generate.ENTER.Flugreservierung.PreisBerechnen
       #ifdef __DS_DOCUMENT_TRACE__
         DsTrace::Enter(1,"Flugreservierung","PreisBerechnen","",
                     __FILE__,__LINE__);
           DsTrace::Parameter(1,"Steuersatz",Steuersatz);
       #endif
//..endgenerate

//..generate.PARAMCHECKS.Flugreservierung.PreisBerechnen
       // Parameterchecks - nur fuer die IN - Parameter
       #ifdef __DS_INTEGRATION__

       if ( ! Steuersatz.Check() )
       {
        throw ExDs("Korrupter Parameter Steuersatz","flugres.cxx",
                   ERRDS_KORRUPTER_PARAMETER,
                   "PreisBerechnen", "Flugreservierung","<Dokument>");
       }
       #endif
//..endgenerate

       // Implementierung in C++

//..generate.LEAVE.Flugreservierung.PreisBerechnen
       #ifdef __DS_DOCUMENT_TRACE__
         DsTrace::Leave(2,"Flugreservierung","PreisBerechnen","",
                     __FILE__,__LINE__);
       #endif
//..endgenerate
   }
   catch (...) {
     // Abräumen aller mit new erzeugten Objekte

     // Exception werfen
     throw ExDs("catch(...)","flugres.cxx", 0,
            "PreisBerechnen", "Flugreservierung","<Dokument>");
   }
 }
```

Bild 2: Generierter Methodenrahmen

3.3. Was der Generator nicht tun soll

Man kann mit einem Generator vieles tun, was nicht unbedingt zu einem effizienteren Prozeß der Softwareerstellung führt. Dazu gehören aus unserer Sicht die Umsetzung von Pseudocode in Coderahmen und die Umsetzung von Kardinalitäten bei Attributen und Parametern.

Umsetzung von Pseudocode

Die Meinungen zu Pseudocode sind gespalten. Wir halten es nicht für sinnvoll, Pseudocode automatisch in Coderahmen umzusetzen. Verwendet man Pseudocode, so beginnt die Diskussion schon mit der Frage, welchen Pseudocode man denn verwenden möchte. Pseudocode führt auch dazu, daß die Spezifikationen nicht mehr von den Anwendern gelesen werden oder für Anwender unverständlich werden.

In den Spezifikation, die *GENCXX* als Eingabe erhält, wird daher strikt zwischen *Effektbeschreibungen* und *Pseudocode* unterschieden.

- *Effektbeschreibungen* sind fachliche Beschreibungen der Wirkung einer Methode auf das Objekt oder andere Objekte. Effektbeschreibungen sind ergebnisorientiert und sind das wesentliche Element der Spezifikation.

- Wenn ein Entwickler den Algorithmus kennt, mit dem er eine Methode implementieren möchte, kann er diese zusätzlich in syntaktisch freiem *Pseudocode* schon vorab spezifizieren. Er muß es aber nicht - er wird aber auch nicht durch Formalismus daran gehindert. Einzige Anforderung ist, daß Pseudocode (jedweder Form) strikt von Effektbeschreibungen getrennt wird.

Der Pseudocode, den der Entwickler schreibt, wird nicht analysiert oder umgesetzt.

Umsetzung von Kardinalitäten

Auf Spezifikationsniveau besteht die Möglichkeit, die Kardinalität von Beziehungen syntaktisch auszudrücken. So steht zum Beispiel der Operator "-->>" für eine 1:n Beziehung. Abbildung 3 zeigt weitere Beispiele.

Dabei hat der Programmierer, der einen solchen Objekttyp in eine Klasse umsetzt zahlreiche Wahlmöglichkeiten, wie Kardinalitäten realisiert werden können, zum Beispiel als einfach oder doppelt verkette Listen, mit Containerklassen ohne und mit Typbindung etc..

```
JEDE          Reise
BESCHREIBUNG  Unter einer Reise werden alle Leistungen, Buchungen und
              sonstigen Dinge zusammengefaßt, die eine Gruppe von Kunden,
              die eine Reise zusammen buchen, in Anspruch nehmen.


HAT           Teilnehmer          -->>       Reiseteilnehmer

HAT           Leistungen          -[3:n]->>  Leistung
              Nach den IATA-Regeln fuer Flugreisen, muß jeder Teilnehmer
              mindestens drei Leistungen in Anspruch nehmen, nämlich
              Flug, Hotel und eine Leistung am Ort
```

Bild 3: Attribute und Parameter mit Kardinalitäten

Diese Wahlentscheidungen kann ein einfacher Generator, wie der hier vorgestellte, nicht leisten. *GENCXX* macht daher in solchen Fällen lediglich Vorschläge, die aber deutlich mit Kommentaren gekennzeichnet werden. Das Design eines vollständigen C++-Klassenheaders und der Signaturen erfordert mehr Expertenwissen und Ingenieurskunst, als man in einen schnell entwickelten Generator packen kann.

3.4. Qualitätsaspekte

Gute Programmierer sind oft Individualisten die Routinearbeiten nicht lieben. Wer füllt schon gerne Formulare aus? Die Wertschöpfung liegt darin, daß die Programmierer ein gutes Programmdesign entwickeln und sich die Routinearbeiten durch Werkzeuge abnehmen lassen.

Ein Generator wie *GENCXX* hat in unseren Projekten durch mehrere Aspekte qualitätsfördernd gewirkt:

- Codierungsstandards für Header und Rahmen werden am besten im Generator implementiert. Das erspart Reviews und sorgt garantiert dafür, daß die Standards eingehalten werden, wenn der Generator fehlerfrei ist.

- Da viele Menschen ermüdende Routinetätigkeiten hassen, erledigen sie sie mit einer erhöhten Fehlerquote. Wenn die Arbeit interessant ist sinkt die Fehlerquote.

- Was man maschinell generiert, kann auch einfach maschinell kontrolliert werden. Wenn noch Felder zum Ausfüllen durch die Entwickler bleiben, kann man dem Entwickler einfache Analysewerkzeuge zur Selbstkontrolle zur Verfügung stellen, ob alles ausgefüllt wurde - natürlich nicht, ob es auch korrekt ausgefüllt wurde.

- Informationen für nachgelagerte Analysatoren wie Browser, Dictionaries und Wartungswerkzeuge sind oft aufwendig zu schreiben. Hier erreicht man durch Generierung leicht Null-Fehler-Qualität, wo man sonst aufwendig nachbessern muß, wenn man festgestellt hat, daß bestimmte Klassen nicht in Browsern erscheinen etc.

Zusammenfassend läßt sich sagen, daß wir den Generator alleine aus Qualitätsgründen schon als lohnend für unser Projekt empfunden hätten. Daß der Aufwand durch Zeitersparnis bei der Realisierung sogar überkompensiert wurde, ist ein angenehmer Nebeneffekt.

3.5. Wartung von Spezifikation und Code

Industrielles Software-Engineering erfordert die Wartung nicht nur des Codes sondern auch die Wartung aller Dokumente, aus denen er abgeleitet wurde. Sonst läßt sich die Qualität bezogen auf die Spezifikation nach einigen Wartungszyklen nicht mehr prüfen. Regressionstests werden unmöglich und die Wartung wird mit fortschreitendem Alter der Programme immer teurer, aufwendiger und zugleich unsicherer, weil die Bezugsbasis für Tests fehlt.

Will man verhindern, daß die Qualität über die Dauer der Wartung verwässert, so muß man für die Wartung aller Dokumente des Software-Lifecycle Vorkehrungen treffen.

Ein häufig geäußerter Kritikpunkt an Generatoren ist, daß sie die Wartung erschweren, wenn im generierten Code Änderungen vorgenommen werden. Es gibt 2 Verfahren, dem hier vorzubeugen:

- Wartung nur in der Quelle für den Generator - jedoch nie im generierten Code.
- Wartung von Quelle für den Generator und Generat.

Der erste Ansatz ist der auf den ersten Blich erstrebenswerte. Da mit *GENCXX* eine First-Cut Generierung vorgenommen werden soll (Rahmen und Header - jedoch keine Ablauflogik - siehe

```
/******************************************************************************/
// METHODNAME:
//           Flugreservierung :: PreisBerechnen
//
// ABSTRACT: einzeilige (60 Zeichen) Beschreibung der Methode
//
// AUTOR:     sd&m, WK ( Sun May 29 17:54:58 1994 )
//
// DESCRIPTION:
//..generate.DESCRIPTION.Flugreservierung.PreisBerechnen
//.  Der Preis berechnet sich aus AnzahlPersonen die beteiligt sind
//.  mal ( ^FlugAngebot.Verkaufspreis + die VerkaufspreisDifferenz der
//.  jeweiligen ^Klasse ) + der jeweils gültige Steuersatz fuer den
//.  Vertrag.
//.
//..endgenerate
//
// SEE_ALSO:
//
// EXCEPTIONS:
//..generate.EXCEPTIONS.Flugreservierung.PreisBerechnen
//..endgenerate
//
// INTERNAL:
//..generate.INTERNAL.Flugreservierung.PreisBerechnen
//. PSEUDOCODE:
//. Hier kann man noch Pseudocode eintragen, wenn man schon in der
//. Spezifikation etwas festhalten möchte.
//..endgenerate
//
// PARAMETERS:
//..generate.PARAMETERS.Flugreservierung.PreisBerechnen
//.-    Steuersatz            IN    const
//..endgenerate
//
// RETURNVALUE:
//..generate.RETURNVALUE.Flugreservierung.PreisBerechnen
//.    DMK
//..endgenerate
//
// $1$
// SYNOPSIS:
// $-2$
/******************************************************************************/
```

```
//..generate.METHODHEADER.Flugreservierung.PreisBerechnen
DMK
 Flugreservierung :: PreisBerechnen (
                const DsProzent                 &    Steuersatz,
  )
//..endgenerate
 { /* $END$ */
   try {
//..generate.ENTER.Flugreservierung.PreisBerechnen
       #ifdef __DS_DOCUMENT_TRACE__
         DsTrace::Enter(1,"Flugreservierung","PreisBerechnen","",
                     __FILE__,__LINE__);
           DsTrace::Parameter(1,"Steuersatz",Steuersatz);
       #endif
//..endgenerate

//..generate.PARAMCHECKS.Flugreservierung.PreisBerechnen
       // Parameterchecks - nur fuer die IN - Parameter
       #ifdef __DS_INTEGRATION__

       if ( ! Steuersatz.Check() )
       {
        throw ExDs("Korrupter Parameter Steuersatz","flugres.cxx",
                   ERRDS_KORRUPTER_PARAMETER,
                   "PreisBerechnen", "Flugreservierung","<Dokument>");
       }
       #endif
//..endgenerate

       // Implementierung in C++

//..generate.LEAVE.Flugreservierung.PreisBerechnen
       #ifdef __DS_DOCUMENT_TRACE__
         DsTrace::Leave(2,"Flugreservierung","PreisBerechnen","",
                     __FILE__,__LINE__);
       #endif
//..endgenerate
   }
   catch (...) {
     // Abräumen aller mit new erzeugten Objekte

     // Exception werfen
     throw ExDs("catch(...)","flugres.cxx", 0,
            "PreisBerechnen", "Flugreservierung","<Dokument>");
   }
 }
```

Bild 2: Generierter Methodenrahmen

3.3.), scheidet der erste Weg aus. Der zweite Weg ist extrem aufwendig, wenn man ihn nicht durch Werkzeuge unterstützt. Dafür muß man jedoch schon die Voraussetzungen in den generierten Rahmen schaffen.

3.5.1. Generierung und Nachgenerierung

Die Wartung eines Systems umfaßt somit nicht nur die Programmquellen, sondern auch die Quellen, aus denen C++- Rahmen generiert wurden - also die Systemspezifikation. Abbildung 4 zeigt das Vorgehen im Wartungsfall.

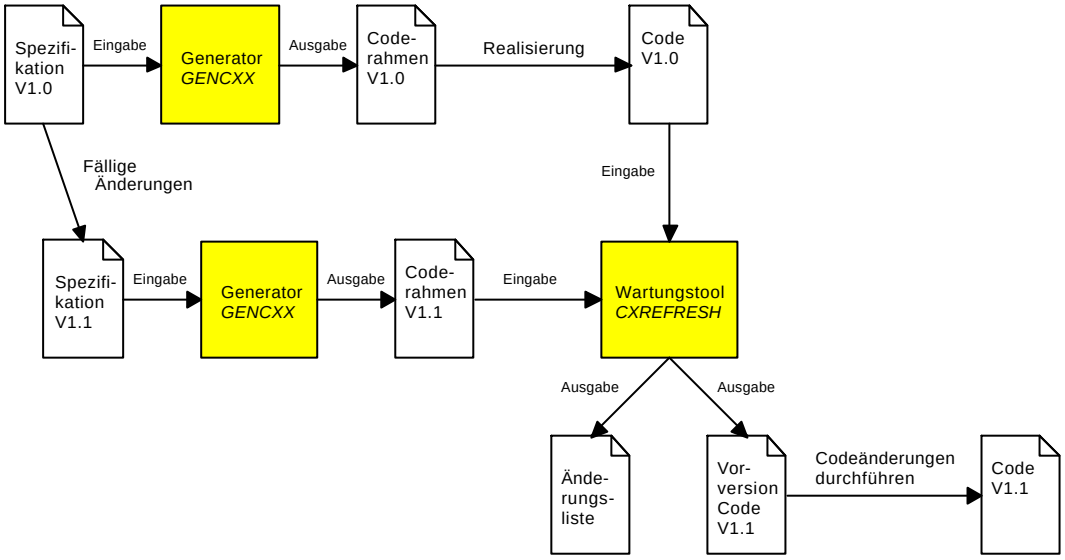

Bild 4: Nachgenerierung im Wartungsfall

Im Wartungsfall wird als erster Schritt die Spezifikation aktualisiert. Dann wird auf Basis der aktualisierten Spezifikation eine Generierung mit *GENCXX* vorgenommen. Es entsteht eine nicht ausgefüllte Quelle, die noch keinen Code enthält, aber die aktuellen Auszüge aus der Spezifikation.

In einem Abgleichsschritt (CXREFRESH) werden die Differenzen zwischen dem ersten Generat und dem neu generierten Rahmen ermittelt und die geänderten Abschnitte werden ausgetauscht. Dies geschieht mit einfachen Textverarbeitungswerkzeugen[2]. Daraus kann eine Wartungsliste erzeugt werden, die der Entwickler abarbeitet.

[2] im Beispiel mit ProMac [Scholz 91, Scholz 93], man kann aber auch zum Beispiel AWK [AWK 88, AWK 91] verwenden.

Aus diesem Vorgehen folgen einige Anforderungen für den Generator:

? Jeder Generierungspunkt, der gewartet werden kann, muß eindeutig identifiziert werden. Dies geschieht durch spezielle Kommentare. Man kann dann feststellen,

  — welche Generierungspunkte entfallen sind - zum Beispiel durch entfallene Methoden oder Attribute.

  — welche Generierungspunkte neu hinzugekommen sind - zum Beispiel durch Einfügen von Methoden

  — wo geändert wurde - durch einen Textvergleich der Texte in den eindeutig identifizierten Generierungspunkten.

? Es müssen folgende Kommentararten unterschieden werden:

  — Kommentare, die generiert sind, und nicht geändert werden sollten. *GENCXX* generiert diese in C++ als "//.".

  — Kommentare, die der Entwickler eingefügt hat. Diese sind mit normalen Kommentaren gekennzeichnet. Normale Kommentare werden auch dort verwendet, wo garantiert keine Nachgenerierung erforderlich ist.

  — Kommentare, die entfallen können und nicht gewartet werden. Dies sind Warnungen des Generators etc. Solche Kommentare sind mit "//.-" gekennzeichnet.

3.5.2. Diskussion

Zu dem oben geschilderten Verfahren der Wartung von generierten Rahmen lassen sich folgende Nachteile nennen:

? Das Vorgehen ist *nicht perfekt automatisierbar*. Die Quellen können nicht vollautomatisch geändert werden. Dies ist jedoch auch theoretisch nicht möglich, wenn man davon ausgeht, daß der Programmierer nach der First-Cut Generierung noch Entscheidungen treffen muß. Die Entscheidungen müssen natürlich dann auch im Wartungsfall getroffen werden, was eine Vollautomatik unmöglich macht.

? Die Generierungsmarken *verschlechtern die Lesbarkeit* des Codes. Sie können jedoch mit einem einfachen Textverarbeitungswerkzeug [AWK 88, AWK 91] jederzeit entfernt werden, wenn es zum Beispiel darum geht, Extrakte für eine Klassenbrowser oder ähnliches zu erzeugen.

? Der *Änderungsaufwand* des Verfahrens ist hoch. Die Erfahrung zeigt, daß man kleine Änderungen schneller von Hand nachziehen kann. Für große Änderungen ist zumindest die Liste, wo geändert werden muß, unverzichtbar.

Arbeitet man nicht mit Vollgenerierung - und damit mit einer Spezifikation, die nichts mehr offen läßt - lassen sich die Nachteile jedoch nicht vermeiden, wenn man nicht wesentlich höheren Toolaufwand treiben will.

3.6. Zusatznutzen: Testdummies, Integration

Im Projektalltag hat sich ein wesentlicher Zusatznutzen der hier vorgestellten Generierung ergeben. Für Integrationstests und den Austausch von Schnittstellen zwischen mehreren Teilteams lassen sich schnell sogenannte Trace-Versionen von Klassen erzeugen.

Dies sind Klassen, deren Schnittstellen korrekt sind, die Traces erzeugen können, die aber immer keine oder nur wenige Ergebnisse zurückliefern.

Trace-Versionen sind so compilierbar, daß später nur noch die "scharfe" Klasse eingebunden werden muß. Der Zeitaufwand für Integration läßt sich so drastisch verringern. Das Testen kann durch Trace-Versionen vereinfacht und verbilligt werden.

4. Implementierung des Generators

Der Generator *GENCXX* wurde mit der universellen Makroprogrammiersprache ProMac [Scholz 91, Scholz 93] entwickelt. ProMac erleichtert durch einige Konstrukte die Codegenerierung und stellt neben dem Funktionsumfang einer 3GL Sprache vor allem Konstrukte für die Text- und Stringverarbeitung zur Verfügung. Ähnliche Codegeneratoren werden oft auch mit awk [AWK 88, AWK 91, Baldwin 90] realisiert.

GENCXX arbeitet als Top-Down-Parser auf der in Abschnitt **Fehler! Verweisquelle konnte nicht gefunden werden.** vorgestellten Spezifikationssprache. Der Generator war dadurch einfach und schnell zu implementieren.

Kann man GENCXX kaufen? - Portierungshemmnisse

GENCXX wurde für eine spezielle Entwicklungsumgebung und einen speziellen Kunden entworfen. Daher wurde in *GENCXX* einiges Wissen über die Umgebung integriert, das sich nur mit erhöhtem Aufwand hätte herausabstrahieren lassen. Ein universeller Generator für jedes C++-Projekt ist entweder weniger leistungsfähig oder erheblich kostspieliger. *GENCXX* wurde auf die Projektsituationen an folgenden Punkten spezialisiert:

? *Typkonzept* - Der Generator macht Gebrauch von Namenskonventionen eines speziellenTypkonzeptes.

? *Klassenbrowser* - Der Generator erzeugt Header und Marken für einen speziellen Klassenbrowser.

? *Codestandards* - Der Generator muß an die für das Projekt gültigen C++-Richtlinien für Code, Header etc. angepaßt werden. Das kann auch die Trennung von Klassen in eine Header- und eine Datenhaltungsklasse bedeuten.

5. Projekterfahrung

GENCXX wurde in einem größeren Projekt mit zunächst 7 Bearbeitern seit einem Jahr eingesetzt und unterstützt derzeit ca. 10 Entwickler mit der gewarteten Urversion. Für andere Projekte mit anderen Kunden - und daher anderen Typkonzepten und Codierungsstandards - haben sich Nebenäste gebildet.

Die Erfahrungen mit der Qualität in den Projekten waren durchwegs positiv. Dies liegt aber vor allem an der Durchgängigkeit der objektorientierten Methodik und weniger am Generator. Der Generator hat lediglich geholfen, die Code-Reviews der Kunden gut zu überstehen und trotzdem schnell zu arbeiten. Der Generator hat auch geholfen, mehr Zeit für Spezifikation und Test zu haben, und weniger Zeit mit dem Ausfüllen von Formularen zu verbringen.

Der derzeitige Nachgenerierungsmechanismus kann noch verbessert werden, speziell beherrscht der Generator noch kein *name mengling* für die Identifizierung der Generierungspunkte in überladenen Methoden. Da die bisher angefallenen Änderungen klein waren, war es aber nicht notwendig, extremen Aufwand in die Nachgenerierung zu investieren.

Literatur

[AWK 88] Aho, Kernighan, Weinberger; *The awk Programming Language*; Addison-Wesley 1988

[AWK 91] Herold: *UNIX und seine Werkzeuge; awk und sed*; Addison-Wesley 1991.

[Baldwin 90] Baldwin, W.: *awk as a Code Generator*; Dr. Dobbs Journal; Vol 15(8), 1990.

[BarDen 93] Bartsch, W.; Denert, E.: *Objektorientierte Spezifikation: Konzepte und eine Notation*; in Mayr, H. C.; Wagner, R. (Hrsg.): *Objektorientierte Methoden für Informationssysteme*; Proceedings, Fachtagung der GI-Fachgruppe EMISA, Klagenfurt, 7. - 9. Juni 1993 (Springer Verlag).

[Denert 93a] Denert, E.: *Dokumentenorientierte Software-Entwicklung*; Informatik Spektrum (1993) 16: S. 159 - 164.

[EllStr 91] Ellis, M. A.; Stroustrup, B.: *The Annotated C++ Reference Manual*, Addison Wesley 1991.

[Graham 93] Graham, I.: *Object Oriented Methods*, Second Edition, Addison Wesley 1993.

[HöTe 93] Höbel, N.; Tensi, T.: *Manual für die TOLEDO-Entwicklungsmethodik*; Interne Unterlagen, Siemens AG - CC Kordoba, 1993

[Jacobson 93] Jacobson, I.: *Object-Oriented Software Engineering - A Use Case Driven Approach;* 4. Auflage, Addison-Wesley 1993

[Scholz 91] Scholz, G.: *Programmiersprachen und Makrotechnik* in: P.Schnupp (Hrsg.) *Moderne Programmiersprachen*; Oldenbourg 1991.

[Scholz 93] Scholz, G.: *Maßgeschneiderte Software-Generatoren*; Proceedings ONLINE 1993 Congress VI - C636.

Inhaltsverzeichnis

Generierung von C++ Rahmen aus objektorientierten Spezifikationen

Der Generator *GENCXX*

Wolfgang Keller

sd&m gmbh
München

August 1994

sd&m
software design & management
GmbH & Co. KG
Thomas-Dehler-Straße 18
81737 München
Telefon (089) 6 27 02 - 0
Telefax (089) 6 27 02 50

Abstract

Der Artikel beschreibt einen Programmrahmengenerator, der objektorientierte Spezifikationen mit einer First-Cut-Generierung in C++ Code-Rahmen umsetzt. Dabei wird Code für Traces, Parameterprüfung und Rahmen für Exception-Handling automatisch generiert. Der Inhalt der Spezifikationen wird in die Header übernommen. Der funktionale Teil des Codes wird von Hand ergänzt. Das Verfahren unterstützt auch ein Wartungskonzept für die Wartung von C++-Code und Spezifikationen. Es wird diskutiert, was man mit einem Generator besser nicht generieren sollte. Die in praktischen Projekten mit *GENCXX* gemachten Erfahrungen werden beschrieben.

The article discribes a program frame generator that takes object oriented specifications as an input for a first cut generation of class headers and member function frames for C++. Trace code, parameter checks and frames for exception handling are generated automatically. The contents of the underlying specification are inserted into the member function headers. The functional part of the code has to be added by hand. The generator also supports a maintenance concept for both, specification and derived code. The discussion contains do's and nodo's for first-cut generation and practical project experiences with the generator *GENCXX*.

Keywords: C++, code generation, software engineering, object oriented software production, software maintenance, awk, ProMac.

About the author: Wolfgang Keller is a software engineer at software design and management in Munich, Germany, with a special interest in software engineering environmets and the object oriented software development process. He can be reached via the Internet: wolfgang.keller@sdm.de or Compuserve 100043,3073.