von marco emrich und max schlee

mehr zum thema:
www.polite-project.org

CODEGENERIERUNG MIT „XFRAMER" UND PROGRAMMIER-TECHNIKEN FÜR FRAMES

Die Generierung von Quellcode gewinnt neuerdings wieder stark an Bedeutung. Zunehmend werden Paradigmen wie MDA, Software-Produktlinien oder die generative Programmierung eingesetzt, um den wachsenden Bedarf an spezialisierterer Software zu decken. Dieser Beitrag stellt die Frame-Technologie vor, die für die genannten Entwicklungsansätze nutzbar ist.[1]) Nach einer kurzen Einführung steht das Werkzeug „XFramer" im Mittelpunkt. Anschließend werden ausgewählte nützliche Programmiertechniken für Frames vorgestellt, die ersten praktischen Erfahrungen entspringen.

Überblick über die Frame-Technologie

Der Ursprung der Frame-Technologie findet sich bereits in den siebziger Jahren und geht auf Marvin Minsky, einen bekannten KI-Forscher, zurück (vgl. [Min74]). Minsky verwendete Frames zur Darstellung natürlicher Sprache und schlug auch den Einsatz von Frames zur Bilderkennung vor. Seine Theorien wurden innerhalb der KI-Forschung aufgegriffen und weiterentwickelt. Frame-Technologie versteht sich jedoch keinesfalls als KI-Technik, sondern ist vollständig in der Softwareentwicklung angesiedelt. Unabhängig voneinander nutzten Paul Basset (vgl. [Bas97]) und die Firma Delta Software Technology (vgl. [DST]) Minskys Ideen als Grundlage, um eine Technik zur Generierung von Software zu entwickeln. Frame-Technologie ist eine Generatortechnik, bei der ein so genannter Frame-Prozessor aus einer Spezifikation Quellcode generiert. Verschiedene Hersteller bzw. Entwickler bieten bereits Frame-Prozessoren an (siehe Tabelle 1). Dabei gibt es zwei Grundkonzepte, denen die Prozessoren zugeordnet werden können (vgl. [Emr03]):

Dem *Adaptionskonzept* von Basset steht das von Delta Software Technology entwickelte *Abstraktionskonzept* gegenüber.

Letzteres wird vom Werkzeug „ANGIE" realisiert und soll im Folgenden kurz vorgestellt werden. XFramer ist eine von ANGIE angeregte Adaption des Abstraktionskonzeptes in Form einer Präprozessor-Technik.

Die Verwendung von Frames ähnelt der objektorientierten Programmierung, wobei Frames jedoch nicht zur Laufzeit, sondern zu Generierungszeit eingesetzt werden. Frames sind prinzipiell Codebausteine, die sich wie Objekte benutzen lassen (vgl. [DST]). Dabei wird zwischen Frames und Frame-Instanzen unterschieden, wobei Frames Klassen und Frame-Instanzen Objekten entsprechen. In den Frames sind so genannte *Slots* untergebracht, bei denen es sich um Variabilitätspunkte (ähnlich Makroparametern) handelt, die in verschiedenen Instanzen unterschiedlich ausgeprägt sein können. Ausgeprägte *Slots* enthalten textuelle Bausteine, wie zum Beispiel Codeblöcke, oder sie referenzieren wiederum Frame-Instanzen. Durch die auf *Slots* basierende Verbindung von Frame-Instanzen entsteht ein gerichteter azyklischer Graph, bei dem die Knotenpunkte Frame-Instanzen sind. Diese Graphen oder Bäume werden als Frame-Hierarchie bezeichnet.

XFramer

XFramer ist eine als Freeware verfügbare Spracherweiterung für C++, C# und Java, die den genannten Sprachen die Fähigkeit zur Verarbeitung von Frames verleiht. Zusammen mit dem Compiler der verwendeten Sprache lässt sich XFramer als Frame-Prozessor einsetzen (siehe Abb. 1). XFramer arbeitet dabei als Präprozessor, der Frame-Code in gültigen C++-, C#- oder Java-Code überführt, den der jeweilige Compiler dann weiterverarbeitet. Die Codeblöcke, aus denen sich das Generat zusammensetzt, können sowohl textuelle als auch binäre Komponenten enthalten. Textuelle Codeblöcke können Code einer beliebigen Zielsprache umfassen, also beispielsweise auch Basic, HTML, Perl, Prolog, XML, oder sogar natürlichsprachlichen Text, etwa zur Generierung von Dokumentation. Bei Bedarf kann für den Frame- und den Zielsprachen-Code auch dieselbe Sprache verwendet werden, sofern es sich um C++, C# oder Java handelt. Dies ist jedoch keineswegs notwendig.

Der Präprozessor erweitert die unterstützten Programmiersprachen um mehrere Elemente. Neu sind das Schlüsselwort `frame`, zwei XFramer-Makros und die Einbettungsmarkierungen (*Flags*[3]). Bei den Makros handelt es sich um `BEGIN_FRAME_TEXT()` und `END_FRAME_TEXT()`, die den Anfang und das Ende des Zielsprachencodes definieren. Dazwischen werden die Einbettungsmarkierungen für die *Slots* platziert. Diese Syntax-Elemente sind in allen unterstützten Programmiersprachen gleich. Abhängig von der verwendeten Programmiersprache für den Frame-Code unterscheiden sich jedoch die für diese Sprache zur Verfügung gestellten XFramer-Unterstützungsbibliotheken, „MFramer.X" und „MString.X". Die

[1]) *Teile der in diesem Beitrag vorgestellten Arbeiten und Ergebnisse wurden im Projekt PoLITe durchgeführt. PoLITe ist ein gemeinsames Projekt der Fraunhofer IESE, Kaiserslautern, und der Fachhochschule Kaiserslautern. Es wird von der Stiftung Rheinland-Pfalz für Innovation finanziert.*

die autoren

Marco Emrich (E-Mail: marcoemrich@web.de) studiert Digitale Medien an der Fachhochschule Kaiserslautern und ist Mitarbeiter des Projekts PoLITe. Er arbeitet zudem als freiberuflicher Dozent für XML und Java.

Max Schlee (E-Mail: schlee2@web.de) ist bei der Firma THOMSON Multimedia & Broadcast Solutions als Software Engineer beschäftigt. Seine Interessenschwerpunkte sind generative Programmierung, Frame-Technologie, Bildverarbeitung und GUI.

Name	ANGIE	FPL	FrameProcessor	Netron Fusion	XFramer	XVCL processor
Hersteller/Entwickler	Delta Software Technology	Frank Sauer	Thomas Patzke	Netron Inc.	Max Schlee	National University of Singapore
verfügbar als	kommerzielle und freie Version	frei & open source	frei & open source	kommerziell	frei	frei & open source
getestete Version	V 0.8 (frei)	V 0.9.1	V 0.7	V 3.2	V 1.43	V 1.0 beta 2
Erste Version von	1999	2002	2002	1986	2002	2001
Benötigtes OS	Windows	OS unabhängig	OS unabhängig	Windows	Linux oder Windows	OS unabhängig
Zielsprachencode	beliebig	beliebig	beliebig	beliebig/COBOL[2])	beliebig	beliebig
Zielplattform	beliebig	beliebig	beliebig	Windows NT, 9x, 3.x, OS/2, CICS, IMS, OS/400, Unix, Open VMS	beliebig	beliebig
Syntax	ähnlich Visual Basic	XML basiert	eigene	ähnlich COBOL	erweitertes C++, C# oder Java	XML-basiert
implementiert in	C	Java	Python	COBOL	C++, C# und Java	Java
Basis-Konzept	Abstraktion	Adaption	Adaption (Minimalansatz)	Adaption	Abstraktion	Adaption
Besondere Zusätze	IDE, viele zusätzliche Bibliotheken, XML-Unterstützung	Plugin für die IDE Eclipse	IDE, Test-werkzeug für Kompatibilität	IDE, diverse Zusatzwerkzeuge	Unterstützung für binäre Komponenten	–

Tabelle 1: Vergleich verfügbarer Frame-Prozessoren

erste der beiden Bibliotheken enthält die so genannten intrinsischen Methoden, die ein fester Bestandteil des Frame-Prozessors sind. „MString.X" ist eine Bibliothek, die Zeichenketten-Funktionalität anbietet. Die aktuellsten Versionen von XFramer für Linux und Windows sind unter www.geocities.com/mslerm/xFramer verfügbar.

Frame-Anatomie

In XFramer besteht ein Frame aus Attributen, Methoden und einem Frame-Text (siehe Abb. 2). Der Frame-Text enthält die wiederverwendbaren Daten, die bei der Generierung zusammengesetzt und wiederverwendet werden, wie zum Beispiel Codebausteine. Innerhalb des Frame-Textes können beliebig viele *Slots* angelegt werden. Diese enthalten bei XFramer Variablen oder Ausdrücke, die entweder elementar, vom Typ MString oder von einem Frame-Typ sein können. Somit kann die Typisierung der Frame-Code-Sprache (C++, C# oder Java) auch auf *Slots* angewendet werden.

Das erste Programm mit XFramer

Das klassische *Hello World* soll auch hier nicht fehlen (Listing 1). Es kann sicherlich nicht die ganze Breite der verfügbaren Möglichkeiten zeigen, die Grundlagen sind aber ersichtlich. Deklarationen und Definitionen eines Frames unterscheiden sich nur geringfügig von denen einer Klasse. Statt des Schlüsselworts class wird frame verwendet, gefolgt vom Framenamen (hier FrHelloWorld). Aus praktischen Gründen empfiehlt es sich, alle Framenamen mit dem Präfix Fr zu versehen, um sie gut erkennbar von verwendeten Klassennamen zu unterscheiden. Neben den klassenüblichen Attributen und Methoden enthält ein Frame zusätzlich den Frame-Text. Die Makros BEGIN_FRAME_TEXT() und END_FRAME_TEXT() definieren die Grenzen des Bereiches, der den Frame-Text enthält. Im Beispiel wird in der Funktion main die Frame-Instanz frHelloWorld mit dem Frame-Text „Hello World!" angelegt und anschließend mit Hilfe der intrinsischen Methode exportToFile in die Datei „output.txt" exportiert.

Um aus der Spezifikation (Listing 1) den generierten Quellcode in der Zielsprache – der in diesem Falle lediglich aus dem natürlichsprachlichen „Hello World" besteht – zu erzeugen, sind folgende Schritte notwendig: XFramer ist mit zwei Parametern zu starten, wobei der erste dem Namen der Quelldatei und der zweite dem der Zieldatei entspricht (hier xframer frHelloWorld.cpp frHelloWold Framed.cpp). XFramer erkennt automatisch die Sprache der Quelldatei und erzeugt anhand dieser eine Zieldatei, die mit dem entsprechenden Compiler zu übersetzen ist. Zur erfolgreichen Übersetzung werden die Unterstützungsbibliotheken „MString.X" und „MFramer.X" im Projektverzeichnis

[2]) *Eine beliebige Sprache ist möglich, Netron Fusion wurde aber speziell für COBOL konzipiert.*

[3]) *Der Term „Flag" wurde bewusst statt des verbreiteten Begriffs „Tag" gewählt. Das vermeidet Verwechslungen, da für Tags eine leicht andere Syntax üblich ist.*

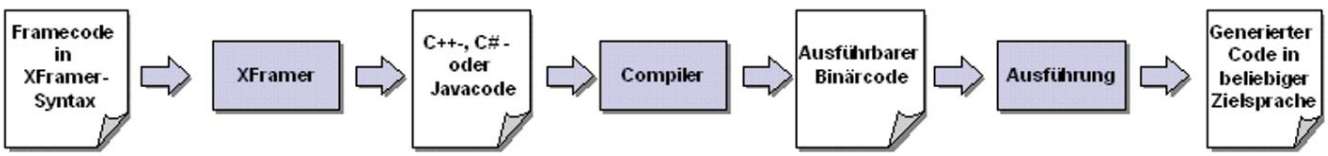

Abb. 1: Codegenerierung mit XFramer

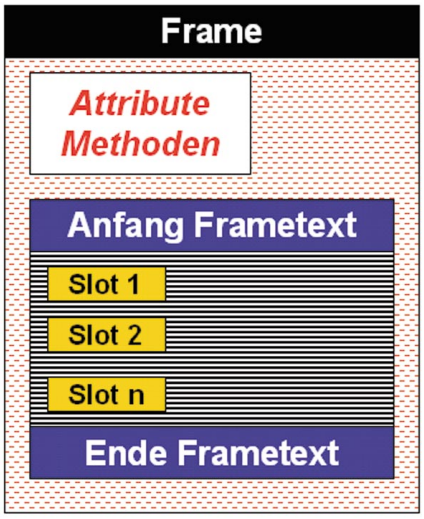

Abb. 2: Frame-Anatomie bei XFramer

benötigt. Die dabei entstandene ausführbare Binärdatei frHelloWorldFramed.exe erzeugt nach ihrem Aufruf die Datei output.txt, die den generierten Quellcode enthält (wird Linux als Betriebssystem verwendet, so heißt die Datei frHelloWorldFramed).

XFramer und Templates

Templates sind ein Sprachkonstrukt, das einige Sprachen verwenden, um parametrische Polymorphie zu realisieren. In C++ etwa gehören Templates zum Standard, während für Java (vgl. [Woy02]) und C# ein Template-Mechanismus bereits angekündigt ist. In anderen Sprachen wäre es jedoch ebenfalls wünschenswert, über eine Form des parametrischen Polymorphismus zu verfügen. Mit XFramer ist das für beliebige Sprachen möglich. Im Folgenden

```
frame FrHelloWorld {
public:
  FrHelloWorld()
  {
BEGIN_FRAME_TEXT()
Hello World!
END_FRAME_TEXT()
  }
};
void main()
{
  FrHelloWorld frHelloworld;
  frHelloworld.exportToFile("output.txt");
}
```

Listing 1: Hello World

```
template<class T>
void swap(T& a, T& b)
{
  T c = a;
  a = b;
  b = c;
}
```

Listing 2: C++ Templatefunktion Swap

```
frame FrSwap {
  FrSwap(char* T)
  {
BEGIN_FRAME_TEXT()
void swap(<!T!>& a, <!T!>& b)
{
  <!T!> c = a;
  a = b;
  b = c;
}
END_FRAME_TEXT()
  }
};
```

Listing 3: XFramer Swap

zeigen wir das Prinzip am Beispiel einer Tauschfunktion für Variableninhalte beliebigen Typs. Als Sprache wurde hier absichtlich C++ gewählt, damit der Einsatz von Templates und Frame-Technologie leicht zu vergleichen ist.

Der Aufruf des in **Listing 2** dargestellten Funktions-Templates veranlasst den Compiler, anhand der Parametertypen eine passende Funktion zu generieren und diese zu übersetzen. Dieser Prozess verläuft implizit und ist für den Entwickler nicht sichtbar.

Listing 3 zeigt die Realisierung der Tauschfunktion mit XFramer. Zunächst gilt es die Variabilitätspunkte der C++-Template-Funktion zu ermitteln. In diesem Fall handelt es sich um die Stellen im Code, die vom Template-Parameter T belegt sind. Die *Slots* – eingeschlossen von den *Flags* „<!" und „!>" – übernehmen dann die Rolle dieser variablen Elemente. Der Typparameter wird dem Frame-Konstruktor übergeben. Im Gegensatz zu den C++-Templates kann der Entwickler Einsicht in den erzeugten Zwischencode nehmen. Aus diesem Grund besitzen XFramer-Programme eine gute Wartbarkeit, denn dem Entwickler stehen alle bekannten Hilfsmittel vom Debugger bis hin zum Profiler zur Verfügung. Weiterhin ist es bei der Nachbildung der Template-Metaprogrammierung (vgl. [Cza00]) nicht mehr nötig die speziellen Funktionen zur Programmerzeugung mittels Templates zu realisieren, da jede normale Funktion eines XFramer-Programms zugleich als eine Metafunktion verwendet werden kann.

Generatorbau mit XFramer

Die Frame-Technologie eignet sich, wie anfangs schon erwähnt, auch sehr gut zur *Generativen Programmierung (GP)*. Merkmalmodelle (vgl. [Cza00]) aus der GP verfügen über alle nötigen Informationen, um eine Frame-Hierarchie aufzubauen. Im Folgenden wird der Generatorbau mit XFramer am Bespiel eines Pizza-Generators erklärt. **Abbildung 3** zeigt ein Merkmaldiagramm, das ein Pizza-Konzept beschreibt. *Kaese* und *Teig* sind Muss-Merkmale, denn sie müssen in jeder Pizza unbedingt vorhanden sein. Die *Sorte* ist ebenfalls ein Muss-Merkmal – deshalb muss aus der Gruppe der alternativen Merkmale *Fleisch* und *Fisch* genau eines ausgewählt werden. Das Merkmal *Zusatz* ist optional. *Zusatz* hat die Untermerkmale *Chili* und *Knoblauch*, die in einer Oder-Gruppe verbunden sind.

Listing 4 (FrTemplate) definiert eine Schablone für alle im Pizza-System vorhandenen Elementarbausteine. Der Variabilitätspunkt ist der Name, anhand dessen Typ, Konstruktor, Ausgabe und Objektname festgelegt werden. Die Realisierung des Merkmals *Sorte* findet sich in **Listing 5**. Der Frame FrSorte erhält als Parameter einen String und prägt die Komponentenschablone FrTemplate aus. Bei der Übergabe eines leeren Strings oder einer unzulässigen Bezeichnung, prägt FrSorte den Frame FrTemplate mit *Fisch* aus. Die Logik innerhalb der *Slots* lässt sich gut durch den

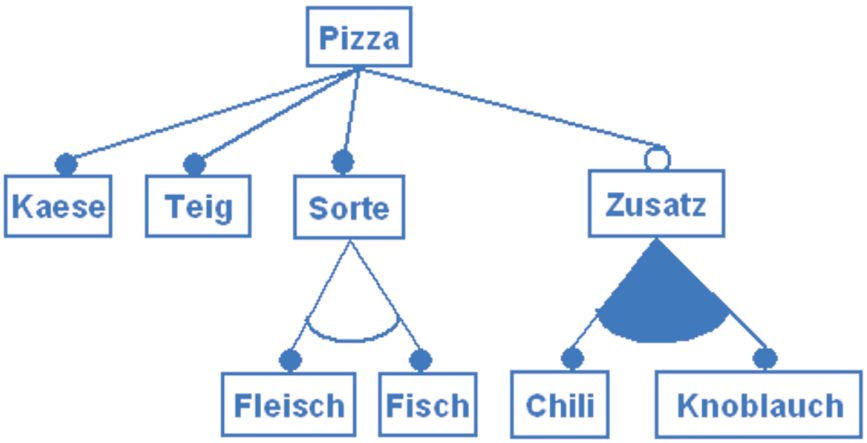

Abb. 3: Merkmaldiagramm des Pizza-Konzepts

```
frame FrTemplate {
public:
  FrTemplate(MString name)
  {
BEGIN_FRAME_TEXT()
struct <!name!> {
  <!name!>()
  {
    cout << "<!name!>" << endl;
  }
} obj<!name!>;
END_FRAME_TEXT()
  }
};
```

Listing 4: Schablone für die elementare Komponente des Pizza-Systems

ternären Operator „?" realisieren. Es ist jedoch nicht zwingend notwendig die Logik in den *Slots* zu verstecken. Sie lässt sich auch außerhalb des Frame-Textes realisieren (**Listing 6**). Die Funktionsweise des Frames FrZusatz ist der des Frames FrSorte ähnlich, mit dem Unterschied, dass die Komponenten *Chili* und *Knoblauch* ganz wegfallen oder auch beide vorkommen können. Bei doppelter Eingabe derselben Komponente greift eine Sonderbehandlung.

FrPizza (**Listing 7**) stellt die Wurzel der gesamten Pizza-Frame-Hierarchie dar. Der Konstruktor von FrPizza übernimmt Frame-Instanzen von FrSorte und optional FrZusatz. Durch die Ausprägung von FrTemplate erfolgt die Einbettung der Komponenten Teig und Kaese, die beide Muss-Bestandteile des Pizza-Systems sind. Danach erfolgen die Einbettungen der *Sorte-* und *Zusatz*-Frame-Instanzen. **Listing 8** zeigt schließlich, wie die unterschiedlichen Pizzen zusammengestellt werden können. Die Instanzen des Frames

```
frame FrZusatz {
public:
  FrZusatz(MString mstr1 = "", MString mstr2 = "")
  {
    MString mstrTemp1 = "";
    if (!mstr1.isEmpty()) {
      mstrTemp1 = (mstr1 != "Knoblauch")
        ? FrTemplate("Chili")
        : FrTemplate("Knoblauch");
    }
    MString mstrTemp2 = "";
    if (!mstr2.isEmpty() && mstr2 != mstr1) {
      mstrTemp2 = (mstr2 != "Knoblauch")
        ? FrTemplate("Chili")
        : FrTemplate("Knoblauch");
    }
BEGIN_FRAME_TEXT()
<!mstrTemp1!>
<!mstrTemp2!>
END_FRAME_TEXT()
  }
};
```

Listing 6: Realisierung des Merkmals „Zusatz"

```
frame FrSorte {
public:
  FrSorte(MString mstr = "")
  {
BEGIN_FRAME_TEXT()
<!(mstr != "Fleisch")
  ? FrTemplate("Fisch")
  : FrTemplate("Fleisch")
!>
END_FRAME_TEXT()
  }
};
```

Listing 5: Realisierung des Merkmals „Sorte"

FrPizza beinhalten ein C++-Programm, das mit der intrinsischen Funktion exportToFile in eine Datei exportiert und anschließend mit einem C++-Compiler übersetzt wird.

Dateieinbettung

Ein weiteres Merkmal von XFramer ist die Möglichkeit Dateien einzubetten. Wird bestehender Code in Frames gekapselt, so sind oft ganze Dateiinhalte manuell in Frames zu kopieren. Das kann einerseits sehr zeitaufwändig sein und andererseits den Frame-Code unübersichtlicher gestalten als nötig. Handelt es sich um binäre Dateien, sind sogar noch weitere Probleme zu bewältigen. Deswegen verfügt XFramer über die Dateieinbettung-*Flags* <# und #> (**Listing 9, Zeile 2**). Innerhalb dieser *Flags* lässt sich eine Datei mit Pfad angeben, die an dieser Stelle eingebunden wird, ähnlich der include-Anweisung in C++.

Dateien können aber auch innerhalb von *Slots* <! !> eingebunden werden. Dies geschieht mit Hilfe der Methode importFromFile der Klasse MString (**Listing 9, Zeile 3**). Die Dateiinhalte werden so erst zur Laufzeit des Generators geladen, während bei der Dateieinbettung die Dateiinhalte (ähnlich wie bei Ressourcen) bereits im Generator mit eingewoben sind.

```
frame FrPizza {
public:
  FrPizza(FrSorte frSorte, FrZusatz frZusatz =
FrZusatz())
  {
BEGIN_FRAME_TEXT()
#include <iostream.h>
<!FrTemplate("Teig")!>
<!FrTemplate("Kaese")!>
<!frSorte!>
<!frZusatz!>
void main(){}
END_FRAME_TEXT()
  }
};
```

Listing 7: Pizza-Konzept-Frame

Parametrisierung der Frame-Textblöcke

In einigen Fällen können die Markierungen, die einen *Slot* kennzeichnen (*Slot-Flag*), mit der Zielsprache in Konflikt geraten. Zum Beispiel kommt es bei der Generierung von HTML-Seiten zu einem Konflikt zwischen den HTML-Kommentarzeichen <!– und dem *Slot-Flag* <!. Das Problem ist allerdings leicht durch eine Umbenennung der *Slot-Flags* zu lösen (**Listing 10**). Mit Hilfe der XFramer-Direktiven, die mit dem Präfix FLAG_ beginnen, lässt sich eine neue Zeichenfolge zuweisen. Diese *Flag*-Substitutionen sind optional und werden innerhalb der Klammern des BEGIN_FRAME_TEXT-Makros angegeben. Die Reihenfolge der Zuweisungen spielt dabei keine Rolle. Die Gültigkeit der *Flag*-Substitutionen ist lokal und beschränkt sich nur auf den folgenden Frame-Textblock.

Bei der Implementierung von XFramer wurde im Übrigen bewusst die lokale Konfiguration gewählt statt der Lösung mit einer globalen Konfigurationsdatei. Das hat folgende Gründe:

- Aus dem Code geht immer deutlich hervor, welche *Flags* durch welche Zeichenketten ersetzt wurden.
- Andere Projekte können den Quellcode weiterverwenden, weil alle Frame-Parametrisierungen nicht in einer externen Datei, sondern im Code selbst enthalten sind.
- Die unterschiedlichen Frame-Text-Parametrisierungen lassen sich an jeder beliebigen Stelle und so oft wie gewünscht anwenden.

Techniken für Frame-Prozessoren

Um früh mit neuen Werkzeugen, Sprachen und Technologien arbeiten zu können, ist es hilfreich, sich nicht nur ein Verständnis der Neuerungen zu erarbeiten, sondern auch von den Erfahrungen bereits eingearbeiteter Entwickler zu profitieren, soweit diese vorhanden sind. Deswegen ist es üblich, solche Erfahrungen in der Form von Mustern zu sammeln. Im Rahmen des PoLITe-Projektes (siehe [IESE03]) wurden Techniken katalogisiert, die bereits heute bei der Arbeit mit Frame-Prozessoren Anwendung finden. Diese Techniken haben nicht den Reifegrad objektorientierter Muster (vergleichbar mit [Gam95]), lassen sich jedoch ähnlich strukturiert katalogisieren. Die vorliegende Strukturierung orientiert sich am generativen Domänenmodell (nach [Cza00]) mit seinen Bestandteilen

```
void main()
{
  FrPizza(FrSorte()).exportToFile("pizza1.cpp");
  FrPizza(FrSorte("Fisch"), FrZusatz("Chili")).exportToFile("pizza2.cpp");
  FrPizza(FrSorte("Fleisch"), FrZusatz("Knoblauch", "Chili")).exportToFile("pizza3.cpp");
}
```

Listing 8: Pizza-Bestellungen

```
BEGIN_FRAME_TEXT()
<#beispiel.cpp#>
<!MString().importFromFile("bsp.bin")!>
END_FRAME_TEXT()
```

Listing 9: Dateieinbettung

```
BEGIN_FRAME_TEXT(FLAG_<! = "<ANFANG_DER_EINBETTUNG>",
                 FLAG_!> = "<ENDE_DER_EINBETTUNG>")
<!–Hier ist ein HTML-Komentar –>
<ANFANG_DER_EINBETTUNG>
Slot
<ENDE_DER_EINBETTUNG>
END_FRAME_TEXT()
```

Listing 10: Substitution der Codeeinbettungflags

- Problemraum,
- Konfigurationswissen und
- Lösungsraum.

Die Technik „Constant Folding"

Die Technik *Constant Folding* für Frame-Prozessoren wir im Folgenden exemplarisch vorgestellt. Eine mögliche Realisierung zeigt ein Beispiel mit XFramer:

- **Name:** *Constant Folding*
- **Platzierung im generativen Domänenmodell:** Lösungsraum
- **Typ:** Optimierungstechnik
- **Anforderungen:** Ein Frame-Prozessor mit der Fähigkeit, Berechnungen ausführen zu können (ANGIE, XFramer oder XVCL)
- **Zielsprache:** beliebig
- **Herkunft der Beispiele:** eigenständig erstellt
- **Beispiele:** Java mit XFramer (C++ als Frame-Sprache)

Absicht: Ausdrücke mit bekannten Operanden werden während des Generierungsprozesses ausgewertet, um die Laufzeit-Performance zu verbessern.

Beschreibung: Diese Technik ist die Frame-Version der Generatoroptimierung *Constant Folding* aus [Cza00], die wiederum eine Variante von *Partial Evalution* darstellt. Die Technik bringt zwei wesentliche Vorteile mit sich: Sie ermöglicht Speicherersparnis bei gleichzeitiger Performanceverbesserung. Bis zu einem gewissen Grad kann *Constant Folding* auch mit einem Präprozessor oder einer Makrosprache verwirklicht werden. Beim Einsatz eines Frame-Prozessors dagegen ist es möglich, die Optimierungstechnik auch auf komplizierte Probleme anzuwenden. Außerdem kann sie damit auch auf Plattformen eingesetzt werden, die über keinerlei Präprozessoren, Makrosprachen oder ähnliches verfügen.

Motivation: Viele Softwaresysteme sind fragmentiert mit kleinen (und manchmal sogar mittleren oder größeren) Hilfsfunktionen und Ausdrücken, die Parameter besitzen, die bereits zur Laufzeit bekannt sind oder zumindest berechnet werden könnten. Oftmals handelt sich dabei um häufig gebrauchte Funktionen, deren Aufruf an vielen verschiedenen Stellen im System erfolgt. Stellen Sie sich beispielsweise eine Funktion für die Fakultätsberechnung vor, etwa `double fac(double x)`. Wird nun die Fakultät von 11 benötigt (11 sei hier ein bereits zur Übersetzungszeit bekannter Wert), so würde der Programmierer sicherlich `a = fac(11)` schreiben, statt einen Taschenrechner zu benutzen und `a = 39916800` einzutragen. Letzteres könnte jedoch eine große Verbesserung der Performance bewirken, insbesondere wenn der Aufruf der Funktion häufig erfolgt. In noch komplizierteren Fällen, wo ein Taschenrechner kaum mehr genügt, um den Rückgabewert des Ausdrucks zu berechnen, tendiert der Programmierer wohl noch eher dazu, die Laufzeit-Funktion zu verwenden, statt den Wert unbequem manuell zu bestimmen. Das ist allerdings nicht einmal eine schlechte Entscheidung. Die manuelle Bestimmung solcher Ausdrücke verschwendet wertvolle Entwicklungszeit. Durch das spätere Suchen und Ändern bekannter Werte in komplizierten Algorithmen wird sogar noch mehr Zeit verschwendet. Es ist außerordentlich schwierig, Code mit ausgeschriebenen Ausdruckswerten zu warten, wenn die Absicht (*Intention*) der Ausdrücke im Code nicht mehr erkennbar ist. Schlimmer noch ist die Benutzung eines zur Laufzeit bekannten Wertes in einer Funktion, deren Rückgabewerte von weiteren Funktionen verwendet wird.

Deswegen ist eine Technik erforderlich, die es erlaubt, die Auswertung von Ausdrücken mit zur Compile-Zeit bereits bekannten Operanden von der Laufzeit auf die Compile-Zeit zu verlegen, ohne jedoch Entwicklungszeit, Intentionalität oder Ausdrucksstärke einzubüßen.

Anwendbarkeit: Benutzen Sie die Technik *Constant-Folding*, wenn

- Ausdrücke oder auch Funktionen mit bekannten oder zur Compile-Zeit ermittelbaren Operanden häufig oder in zeitkritischer Umgebung verwendet werden,
- Ausdrücke oder auch Funktionen mit bekannten oder zur Compile-Zeit ermittelbaren Operanden benutzt werden, die derart komplex sind, dass ein merklicher Performance-Verlust auftritt.

Abstrakte Realisierung: Der Algorithmus, der den Ausdruck berechnet, muss in Frame-Code vorliegen. Die Ausdrücke mit den zur Compile-Zeit bekannten Operanden im Zielsprachen-Code können durch einen *Slot* ersetzt werden, der die Frame-Funktion aufruft. Dies hat zur Folge, dass im generierten Quellcode der Ausdruck durch seinen Wert beziehungsweise die Funktion durch ihren Rückgabewert ersetzt wird.

Realisierungsbeispiel in XFramer

Listing 11 zeigt ein Java-Programm, das ein typisches kombinatorisches Problem löst. Berechnet wird die Anzahl der Möglichkeiten von *k* Kombinationen aus *n* Elementen (sprich: *n* über *k*). In diesem Beispiel sind die Operanden bekannt. Es besteht keine Notwendigkeit die Berechnung zu Laufzeit durchzuführen und dabei Performance einzubüßen. Die Funktionen müssen im Generatorcode (**Listing 12, Zeilen 4 bis 14**) und bei Bedarf zusätz-▸

```
1  class CFExample //n über k
2  {
3    static long fac(int x)
4    {
5      long fac=1;
6      for (int i=2;i<=x;++i)
7        fac*=i;
8      return fac;
9    }
10
11   static long nOverK(int n,int k)
12   {
13     return fac(n) / (fac(k)*fac(n-k));
14   }
15
16   public static void main(String[] args)
17   {
18     System.out.println(nOverK(12,3));
19   }
20 }
```

Listing 11: Constant Folding(Beispiel „n über k")

lich im Zielsprachen-Code (**Listing 12, Zeile 25 bis 36**) implementiert werden. Bitte beachten Sie, dass der Generator in XFramer mit C++-Syntax implementiert ist. Sind beide Implementierungen vorhanden, ist es möglich, das *Constant Folding* jederzeit an- und wieder auszuschalten.

Listing 13 zeigt den generierten Java-Code. In Zeile 18 des generierten Codes wurde der Funktionsaufruf nOverK(12,3) durch den Rückgabewert 220 ersetzt. Um zur Laufzeitberechnung zurückzuschalten, ist lediglich der *Slot* in **Zeile 40 von Listing 12** <!MathAid::nOverK(12,3)!> wieder durch den ursprünglichen Ausdruck nOverK(12,3) zu ersetzen.

Verfügbare Frame-Prozessoren

Neben XFramer gibt es noch eine Reihe weiterer interessanter Frame-Prozessoren. Als Hauptvertreter des Abstraktionskonzeptes sei hier „ANGIE" erwähnt, bei dem es sich um einen sehr mächtigen Frame-Prozessor handelt, den die Firma Delta Software Technology in einer speziellen Version zum kostenlosen Herunterladen anbietet. Bei den Werkzeugen „FPL", „FrameProcessor", „Netron Fusion" und „XVCL-Frame-Prozessor" handelt es sich dagegen um Frame-Prozessoren, die das Adaptionskonzept verwenden. Weitere Information entnehmen Sie bitte der Tabelle.

Fazit

Der vorliegende Artikel kann natürlich nur einige wenige Möglichkeiten der Frame-Technologie aufzeigen. Frames sind

```
1. class MathAid
2. {
3. public:
4.   static long fac(int x)
5.   {
6.   long fac=1;
7.   for (int i=2;i<=x;++i)
8.     fac*=i;
9.   return fac;
10.   }
11.   static int nOverK(int n, int k)
12.   {
13.     return fac(n) / (fac(k)*fac(n-k));
14.   }
15. };
16.
17. frame Main
18. {
19. public:
20.   Main()
21.   {
22. BEGIN_FRAME_TEXT()
23. class CFExample //Lowest Common Multible
24. {
25.   static long fac(int x)
26.   {
27.     long fac=1;
28.     for (int i=2;i<=x;++i)
29.       fac*=i;
30.     return fac;
31.   }
32.
33.   static long nOverK(int n,int k)
34.   {
35.   return fac(n) / (fac(k)*fac(n-k));
36.   }
37.
38.   public static void main(String[] args)
39.   {
40.    System.out.println(<!MathAid::nOverK(12,3)!>);
41.   }
42. }
43. END_FRAME_TEXT()
44.       }
45. };
46.
47. int main()
48. {
49.    Main x;
50.    x.exportToFile("CFExample.java");
51. }
```

Listing 12 Constant Folding (Beispiel „n über k"), Spezifikationscode in XFramer (CFExample.cpp)

praktisch universell einsetzbar und die Anwendungsgebiete sind noch längst nicht abgesteckt. Bisherige Anwendungen der Frame-Technologie finden sich unter anderem in den Bereichen Euro-Umstellung, GUI-Generierung, Jahr2000-Problem, Produktlinien und musterbasierte Softwareentwicklung. Eine ausführliche Auflistung bestehender Projekte liefert das „PoLITe Survey" (vgl. [Emr03]). Frame-Technologie platziert sich damit als neue Alternative zu bestehenden Metaprogrammierungs- und Generierungstechnologien, wie z. B. statischer Template-Metaprogrammierung, kann aber auch in Kombi-

```
1. class CFExample //n über k
2. {
3.   static long fac(int x)
4.   {
5.     long fac=1;
6.     for (int i=2;i<=x;++i)
7.       fac*=i;
8.     return fac;
9.   }
10.
11.   static long nOverK(int n,int k)
12.   {
13.     return fac(n) / (fac(k)*fac(n-k));
14.   }
15.
16.   public static void main(String[] args)
17.   {
18.   System.out.println(220);
19.   }
20.   }
```

Listing 13 Constant Folding (Beispiel „n über k"), generierter Code (CFExample.java)

nation mit einigen dieser Techniken genutzt werden. Letzteres gilt beispielsweise für *Aspect Oriented Programming (AOP)* und *Model Driven Architecture (MDA)*. Frames lassen sich generell als neue aufstrebende Technologie verstehen, deren Bedeutung in Zukunft sicherlich noch wachsen kann. ■

Literatur & Links

[Bas97] P. Basset, Framing Software Reuse: Lessons from the Real World, Yourdon Press, Prentice Hall, 1997

[Cza00] K. Czarnecki, U. Eisenecker, Generative Programming – Methods, Tools, and Applications, Addison-Wesley, 2000, siehe: www.generative-programming.org)

[DST] Delta Software Technology, Delta Software Technology Web-Präsenz, siehe: www.d-s-t-g.com/

[Emr03] M. Emrich, U. Eisenecker, Ch. Endler, M. Schlee, Emerging Product Line Implementation Technologies: C++, Frames, and Generating Graphical User Interfaces (noch unveröffentlicht, siehe: www.polite-project.org)

[EC] Eclipse Consortium, The Eclipse Platform, siehe: www.eclipse.org

[Gam96] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software, Addison-Wesley, 1996

[IESE03] Fraunhofer IESE, University of Computer Sciences Kaiserslautern (Zweibrücken), Project PoLITe (Product Line Implementation Technologies), siehe: www.polite-project.de

[Min74] M. Minsky, A Framework for Representing Knowledge, Massachusetts Institute of Technology, A.I. Laboratory, 1974, siehe: ftp://publications.ai.mit.edu/ai-publications/0-499/AIM-306.ps

[Woy02] P. Woyzschke, G. Wilhelm, Java Generics, in: JavaSPEKTRUM Mai/Juni 2002