

Ähnlichkeit gesucht

Generative Programmierung (GP) nutzt die »innere Gleichförmigkeit« von Software, um Varianten von Softwaresystemen zu erzeugen. Dass Entwickler mithilfe von GP immens Zeit und Kosten sparen können, verdeutlicht der folgende Erfahrungsbericht anhand eines konkreten Projektes, bei dem mittels der Standardtechnologien XML und XSLT C/C++-Code erzeugt wurde.

Trotz Modulen, Klassen und Komponenten gibt es häufig noch viele Teile in der Software, die sehr ähnliche Codepassagen aufweisen. Dies liegt daran, dass die »innere Gleichförmigkeit« der Software oft übersehen wird. Bei der Generativen Programmierung (GP) wird ein Softwaresystem von vornherein darauf ausgerichtet, dass es von einem Basis-Softwaresystem unterschiedliche Ausprägungen bzw. Varianten gibt. Diese Varianten lassen sich an Hand von Merkmalen beschreiben. GP

trennt daher zwischen der prinzipiellen Lösung des Problemkreises (Domain-Engineering) und der konkreten Realisierung einer Systemvariante (Application-Engineering).

Man spricht in diesem Zusammenhang auch von System- oder Produktfamilien, zu denen die einzelnen Varianten zählen. Die Varianten werden mittels Generatoren automatisch erzeugt. Als Input hierzu dient die Vorgabe der konkreten, mittels einer Domänen-spezifischen Sprache (domain specific language – DSL) beschriebenen Merkmale. GP ist eine Software-Methodik und kein Tool. Bereits mit einfachen standardisierten Mitteln wie XML und XSLT lässt sich die Theorie in die Tat umsetzen.

Dabei ist das Einsparungspotenzial an Zeit und Kosten immens.

GP in der Dentaltechnik

Ein Beispiel: Das Unternehmen Sirona Dentaltechnik entwickelt modernste Dentalgeräte. Bei der neuesten Generation kommunizieren die einzelnen Hardware-Komponenten der Anlage über ein eigens hierfür entwickeltes Protokoll. Das Protokoll basiert auf der CAN-Technologie, ist zudem aber so konzipiert, dass es eine möglichst hohe Zeitauflösung reali-

siert. Bei der Konzeption des Kommunikationsprotokolls wurde Klar Automation mit eingebunden.

Aus Sicht der Baugruppen im Dentalgerät werden Kommandos versendet und empfangen. Ein Kommando resultiert dabei – je nach Größe – in einem oder mehreren Telegrammen, die jeweils das spezifische Protokoll verwenden. Die Kommandos lassen sich als eine Systemfamilie betrachten, zu der es unterschiedliche Varianten gibt. Speicher ist auf den Baugruppen knapp, daher ist jedes Kommando bestimmten Baugruppen zugeordnet, für die es dann auch nur zur Verfügung stehen soll. Die Baugruppen kommunizieren nicht direkt untereinander, sondern stets über eine Masterbaugruppe. Dabei erfolgt die Kommunikation zwischen Baugruppe und Master in mehreren Schichten (Bild 1).

Michael Klar
ist Geschäftsführer
von Klar Automation



Bild 1 Schichtenmodell der Kommunikation im Dentalgerät

Der Schritt von einer Kommunikationsschicht zur nächsten erfolgt jeweils nach einheitlichen Regeln. Dennoch sind die Schritte für jedes Kommando zu implementieren, weil die Kommandostrukturen typorientiert sind. Hinzu kommt, dass als Programmiersprache bei den Baugruppen entweder C (bei Mikrocontrollerboards) oder C++ (PC als Kommunikationsteilnehmer) zur Anwendung kommt. Die einzelnen Schichten des Sendens und Empfangens korrelieren miteinander. Hier besteht, bezogen auf einen Typ, Gleichförmigkeit. Damit lässt sich die gesamte Kommunikation in Regeln beschreiben (Domain Engineering). Aus diesen Regeln heraus lassen sich die konkreten Kommandovarianten durch Vorgaben der Merkmale generieren (Application Engineering). Merkmale eines Kommandos sind in diesem Projekt die übertragenen Daten, der Empfänger sowie die Art und Weise der Übertragung (in einem oder mehreren Paketen). Pro Kommando variante wird jeweils die Struktur der übertragenen Daten generiert, sowie die API (Application Programming Interface), um das Kommando verschicken und empfangen zu können.

XML als Syntax

Als Syntax für die DSL wird XML verwendet. In der DSL werden die einzelnen zur Generierung des jeweiligen Kommandos mit seiner API benötigten Merkmale beschrieben. Um die Eingabe zu erleichtern, steht ein kleiner Editor bereit, der als Speicherformat XML verwendet. Für eine bessere Nachvollziehbarkeit soll im Folgenden stets das Kommando »MSG_MOTION« als Beispiel dienen. Durch Anwählen von Checkboxen im

Editor lässt sich festlegen, dass die Masterbaugruppe (DX_11) das Kommando »MSG_MOTION« an die Baugruppen DX_08 und DX_91 sendet. Dieses Kommando soll in einem Telegramm umgesetzt werden (STTP). Listing 1 zeigt den resultierenden XML-Code, der als Input für den Generator dient.

XSLT als Generatorsprache

Dieser Generator lässt sich aus dem kleinen Editor heraus anstoßen. Als Generatorsprache kommt XSLT zur Anwendung. Diverse XSLT-Skripten wandeln die Vorgaben in mehreren Schritten in Sourcecode (C bzw. C++) um. Listing 2 zeigt einen Ausschnitt des XSLT-Skriptes, das aus den XML-Vorgaben die Datenstrukturen für die entsprechenden Baugruppen generiert. Aufgrund dieses Skriptes entstehen in unserem Beispiel die beiden Headerdateien »p2_08_CANCommands.h« und »p2_91_CANCommands.h«.

Listing 3 zeigt den Ausschnitt der generierten Datei »p2_08_CanCommands.h«, welcher aufgrund der XML-Definition von »MSG_MOTION« in Listing 1 entstanden ist.

Im Original enthält die Headerdatei auf Grund des XSLT-Skriptes aus Listing 2 noch weitere Einträge für alle anderen Kommandos, welche die Baugruppe DX_08 zur Kommunikation verwendet. In den funktionalen Sourcecode fügt der Generator unter anderem auch die Kommentare ein, die bei der DSL mit angegeben wurden. Auf diese Weise entsteht dokumentierter Sourcecode. Neben dem XSLT-Skript aus Listing 2 gibt es noch weitere Skripten für z.B. folgende prinzipielle Aufgaben:

- Prototyp-Definition der Handler Routinen zum Empfang eines Kommandos,
 - Zuordnung zwischen Kommando und Handler routine,
 - Funktion zum prinzipiellen Versenden des Kommandos.
- Die Handler Routinen selbst werden nicht erzeugt. Nach welcher Logik der Empfänger auf das Kommando reagieren soll, bestimmt der Baugruppen-Implementierer. Selbst ein Codegerüst wird hier bewusst nicht generiert, da sonst Reverse-Engineering zu beachten wäre. Bei Kommando-Änderung würde die Inkonsistenz zwischen Funktionsprototyp (generiert) und Funktionskopf der Implementierung (herkömmlich erstellt) sofort auffallen.

Listing 1: XML-Code für Kommando »MSG_MOTION«

```
<type name="MSG_MOTION">
  <struct elements="1" low-type="new_type" name="MSG_MOTION">
    <long-doc/>
    <short-doc>Lauf-Command mit Frequenz</short-doc>
    <item elements="1" name="Frequency" type="SWORD">
      <long-doc/>
      <short-doc/>
    </item>
    <item elements="1" name="Position" type="SDWORD">
      <long-doc/>
      <short-doc/>
    </item>
    <component name="DX_08" nr="08"/>
    <component name="DX_91" nr="91"/>
    <protocol name="Master2Slave"/>
    <protocol name="STTP" nr="P"/>
  </struct>
</type>
```

Listing 2: XSLT-Skript zur Generierung der Datenstrukturen (Ausschnitt)

```
<xsl:template match="component" mode="commands">
  <xsl:variable name="nr" select="@nr"/>
  <impl>
    <generate generator="hppgen.xml" path="DX_{nr}"
      file="p2_{nr}_CanCommands.h"/>
    <xsl:if test="nr!=11">
      <xsl:for-each select="/typedb/struct|typedb/blob">
        <xsl:if test="@dest=$nr or @src=$nr">
          <xsl:call-template name="insert_doc"/>
          <line value="-----"/>
          <line value="typedef struct tag_{@name} {"/>
          <xsl:if test="sum(item)!=0">
            <xsl:for-each select="item">
              <xsl:variable name="array"><xsl:if test="
                @elements!='1'">[<xsl:value-of select="
                  @elements"/>]</xsl:if>
              </xsl:variable>
              <var name="{@name}{"$array}" type="
                @{@type}"><xsl:call-template name="
                  insert_doc"/>
              </var>
            </xsl:for-each>
          </xsl:if>
          <xsl:if test="sum(item)=0">
            <var name="pDummy" type="void*">
              </var>
          </xsl:if>
          <line value="}{"@name}"/>
        </xsl:if>
      </xsl:for-each>
    </xsl:if>
  </impl>
</template>
```

Listing 3: Erzeugte Datei »p2_8_CANCommand.h« (Ausschnitt)

```
//-----
typedef struct tag_MSG_MOTION
{
  /*!
   * SWORD Frequency;
   * /*!
   * SDWORD Position;
  }
  MSG_MOTION;
```

Mit dem Ansatz der Generativen Programmierung sind einige Einsparungen möglich. Durch die Kommando-Systemfamilie beschränkt sich die Einführung eines neuen Kommandos auf die Definition desselben im Editor. Den sonstigen Codierungs-Overhead deckt der Generator ab. Dies spart bereits an sich Zeit und vermeidet zudem typische Flüchtigkeitsfehler. Ein weiterer positiver Nebeneffekt ist die Realisierung des Testtools »DxMon«. Die Applikation »DxMon« protokolliert den Kommando-Verkehr und ermöglicht auch Kommando-Simulationen. Die grafische Benutzeroberfläche generiert sich zum größten Teil selbst aus den DSL-Angaben der Kommandos.

Einsparungen von 30%

Nach Schätzungen ließen sich bei der Realisierung der Kommandoschicht Einsparungen von 30% erzielen. Genau lässt sich dies natürlich nicht beziffern, da das Projekt dann doppelt – einmal herkömmlich, einmal generativ – hätte durchgeführt werden müssen. Neben dem Kostenfaktor spielte aber vor allem der Zeitfaktor eine entscheidende Rolle. Eine Baugruppe kann bis zu 128 Kommandos empfangen. Derzeit sind 10 Baugruppen im Gerät involviert.

Fazit: Der Einsatz einer neuen Technologie ruft während der Projektlaufzeit auch skeptische Stimmen hervor. Dies war auch in diesem Projekt der Fall. Mit der Zeit wurde aber für alle Beteiligten deutlich, welche Einsparungen an Zeit und Kosten mit GP verbunden waren. Dies war in der ohnehin knappen Projektlaufzeit ein großer Vorteil. Sirona hat durch den Einsatz von GP bei der Kommando-Realisierung neben dem

Kosten- auch einen Qualitätsvorteil erzielt. Bereits jetzt wird deutlich, wie die Generierung Flüchtigkeitsfehler vermeidet und sonst nötige aufwändige und disziplinfordernde Absprachen zum Thema Kommandos minimiert. Die neue Anlage wurde Ende März 2003 auf einer Messe ausgestellt und man darf gespannt sein, wie

sich GP auch in der Weiterentwicklung des Produkts bezahlt macht.

Klar Automation hat früher GP vor allem mit Hilfe von C++-Templates realisiert. Gegenüber diesem Ansatz bieten XML und XSLT aber noch weit mehr Möglichkeiten. Für den Anwender sind XSLT-Skripten zudem oft einfacher zu verstehen als

C++ Templates – auch wenn XSLT-Skripten auf den ersten Blick sehr gewöhnungsbedürftig erscheinen. Aus diesem Grund wird derzeit intern an der Entwicklung eines Tools gearbeitet. (cg)

Klar Automation

Telefon 0 91 93/63 97 60
Fax 0 91 93/63 97 66