

HANSER

Einfach generieren

Susanne Klar, Michael Klar

Generative Programmierung verständlich und praxisnah

ISBN 3-446-40448-1

Leseprobe

Weitere Informationen oder Bestellungen unter
<http://www.hanser.de/3-446-40448-1> sowie im Buchhandel

3

Gleichförmigkeiten

3 Gleichförmigkeiten

Die Frage nach dem Schema

In diesem Beispiel-Kapitel werden wir vor allem auf das Thema „Gleichförmigkeiten“ abzielen. Gleichförmigkeit drückt aus, wie gleichartig eine Applikation in sich aufgebaut ist. Bei der Suche nach Gleichförmigkeit stellt sich die Frage: „Welches Schema oder welche Schemen stecken in der Applikation?“

Formalisierung als Voraussetzung für Generieren

Diese Schemen zu erkennen und zu formalisieren, ist Voraussetzung für Generierung.

Wie Sie Schemen finden und darauf basierend dann generieren können, wollen wir anhand eines kleinen verteilten Systems verdeutlichen.

➔ *Das lernen Sie:*

- Woran Sie Gleichförmigkeit erkennen.
- Wie Sie eine DSL finden.
- Ein erstes Prozessmodell für Generative Programmierung.
- Warum Generieren gerade im Umfeld verteilter Systeme (embedded oder nicht) zur Qualitätssteigerung beitragen kann.
- Wie Queraspekte über mehrere Dateien hinweg durch einen Generator pflegbar bleiben.
- Dass Applikations-Entwicklung in zwei Rollen eingeteilt werden kann: Modellierer und Entwickler.
- Wie Sie mit HuGo dafür sorgen, dass generierte Quellcodedateien um eigene Codings des Entwicklers ergänzt werden können.

➔ *Die Listingdateien finden Sie unter:*

Projekt `simple_com`

3.1 Schema in Schnittstellenbeschreibung der DSL

Verteiltes System

Angenommen, Sie entwickeln ein verteiltes System. Die Schnittstelle zwischen den verteilten Softwarekomponenten geschieht über Telegrammverkehr, der über entfernte Funktionsaufrufe (remote procedure calls – RPCs) realisiert wird. Um das Beispiel im *Kapitel 3* minimal zu halten, gehen wir von einer Punkt-zu-Punkt-Verbindung bei der Kommunikation aus.

Bei der Entwicklung eines verteilten Systems wird einer der ersten Schritte die Definition der Schnittstelle zwischen den Komponenten sein. D.h.: Wer kommuniziert mit wem worüber?

Noch bevor Sie sich um die Realisierung der Kommunikationsschicht als solche kümmern, werden Sie die Schnittstelle(n) spezifizieren, über welche Telegramme/RPCs die Komponenten miteinander kommunizieren. Dies werden Sie umso mehr tun, wenn die Komponenten aus Zeitgründen parallel von verschiedenen Entwicklern realisiert werden müssen.

Gerade bei Schnittstellenspezifikationen bietet es sich an, über schematische Beschreibungen nachzudenken. Schematische Beschreibungen haben den Vorteil, dass sie schnell zu lesen sind, wenn das Schema vom Leser einmal verstanden wurde.

Schematische
Beschreibung

Wie könnte nun eine einfache, schematische Beschreibung einer RPC-Schnittstelle aussehen?

Listing 3.1 zeigt ein mögliches Schema im XML-Format. Sie finden die Datei im Projektverzeichnis `simple.com/model`.

Listing 3.1 `peer_interface.model`

```
<model>
  <com_if name="com_if1">
    <func name="foo1" >
      <param name="value1" type="int"/>
      <param name="value2" type="int" default="400"/>
    </func>
  </com_if>
  <!--
  <com_if name="com_if2">
    <func name="foo2_1" >
      <param name="data1" type="int" default="-1"/>
      <param name="data2" type="double" default="3.5"/>
    </func>
    <func name="foo2_2" >
      <param name="code" type="char" default="a"/>
      <param name="no" type="int" default="0"/>
    </func>
  </com_if>
  -->
</model>
```

Dieses in *Listing 3.1* beschriebene Kommunikations-Modell geht davon aus, dass es mehrere Schnittstellen (`<com_if>`) geben kann. Pro Schnittstelle gibt es einen Satz von Funktionen (`<func>`). Jede Funktion hat einen Satz an Parametern (`<param>`).

In `peer_interface.model` wurden konkret zwei Schnittstellen spezifiziert: `com_if1` und `com_if2`. `com_if1` besteht aus einer Funktion `foo1`, `com_if2` aus zwei Funktionen, `foo2_1` und `foo2_2`.

Bei der Einführung des Schemas sollten Sie – wie bereits mehrfach erwähnt – darauf achten, dass die verwendeten Tags selbsterklärende Namen besitzen. Ebenso sollte die Struktur des Schemas der Denkweise derjenigen entsprechen, die diese Beschreibung/dieses Modell verwenden.

Selbsterklärende
Tag-Namen

Basierend auf dieser Schnittstellenbeschreibung könnten nun die Entwickler der einzelnen Komponenten bereits beginnen, sich über die Realisierung der einzelnen Funktionen Gedanken zu machen.

3.2 Architektur für verteiltes System

Bei verteilten Systemen wird als Architektur für die Kommunikation üblicherweise ein Schichtenmodell mit Proxy und Stub verwendet.

Das Proxymodul verpackt und serialisiert das Telegramm auf der Senderseite. Auf der Empfängerseite wird das Telegramm in einem zugehörigen Stubmodul entsprechend deserialisiert und entpackt und anschließend eine zugehörige Handlerroutine aufgerufen.

Indiz-Wörter
für Schemen

Die Worte *entsprechend* und *zugehörig* machen bereits deutlich, dass hier gewisse *Analogien* oder „*Gleichförmigkeiten*“ vorliegen, die einem gewissen *Schema* oder *Muster* unterliegen.

Abbildung 3.1 visualisiert diesen Sachverhalt noch.

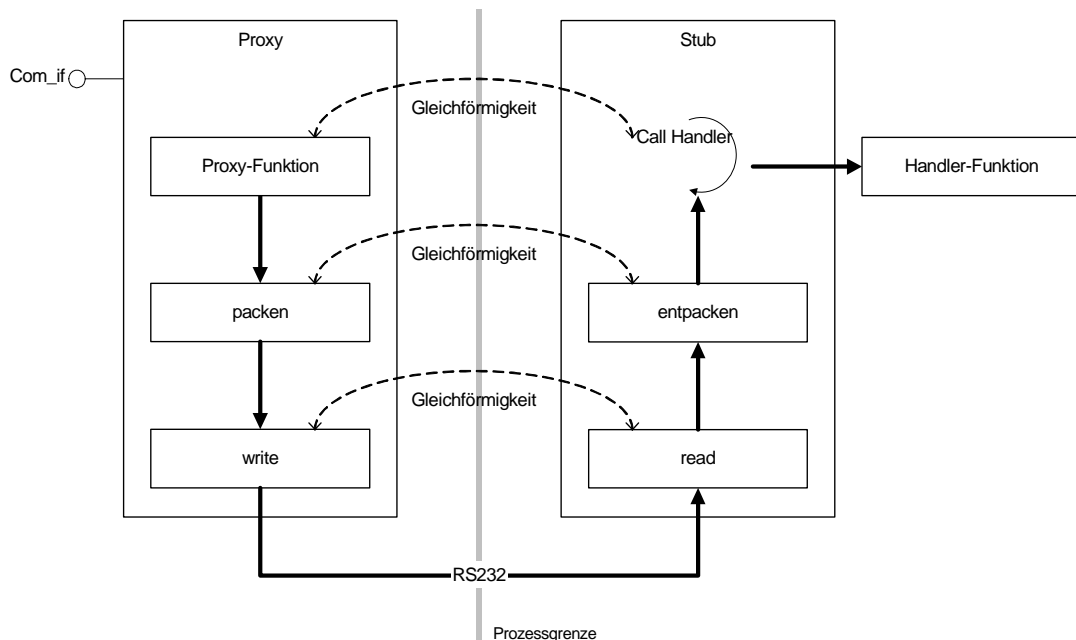


Abbildung 3.1 Analogien in den Schichten bei Proxy und Stub

Je länger Sie sich mit Automatisierungspotenzial beschäftigen, desto mehr werden Sie für gleichförmige/analoge Abbildungen und damit verbundene Begriffe („Schema“, „Muster“, ...) sensibilisiert. Oder suchen Sie einfach danach bzw. streben Sie danach, dies zu erreichen. Denn je gleichförmiger Ihre

Architektur und die eingesetzten Bibliotheken sind, desto höher ist das Generierungspotenzial.

3.3 Referenzimplementierung

Sehr hilfreich kann es sein, zum gefundenen Schema eine Referenzimplementierung durchzuführen, bevor die Code-Schablonen (bei Hugo die Frames) erstellt werden. Je komplexer die Muster, desto hilfreicher, denn oft lässt sich das Schema leichter formulieren, wenn man das Ziel konkret vor Augen hat. Code-Schablone

Um das Beispiel an dieser Stelle klein zu halten, wurden einige Vereinfachungen bei der skizzierten Realisierung getroffen:

- Die RS232-Übertragung wird nur simuliert. Sie geschieht im Beispiel durch Datenaustausch über Datei (erst auf Datei schreiben, dann von Datei lesen).
- Die Prozessgrenze wird hier nur simuliert skizziert. Es werden nacheinander erst Sender und dann Empfänger aufgerufen, die Prozessgrenze wird nur über einen Kommentar angedeutet.

Listing 3.2 zeigt die Verwendung der Kommunikationsschicht. Sie finden sie im Projektunterverzeichnis `simple_com/src`.

Listing 3.2 testmain.c

```
#include <stdio.h>
#include <io.h>
#include <com_if1.h>

/* Simulation: Verschicken des Telegramms */
void main_sender()
{
    int fd;
    fd=open("demo.txt",O_TRUNC|O_WRONLY);
    foo(fd,200,220);
    close(fd);
}
/* Simulation: Empfangen des Telegramms */
void main_receiver()
{
    int fd;

    while(1)
    {
        fd=open("demo.txt",O_RDONLY);
        process_stub_com_if1(fd);
        close(fd);
    }
}
/* Start der Simulation */
```

```

void main()
{
    main_sender();
    /* hier: Prozessgrenze */
    main_receiver();
}

```

Simulation der
Prozessgrenze

Die Funktion `main_sender()` simuliert den Funktionsaufruf von `foo1()`. Dies tut sie über eine entsprechende Proxyfunktion, die die Daten über die RS232 (hier simuliert über Datei) schickt. Die Funktion `main_receiver()` würde sich in der anderen Softwarekomponente befinden, d.h. hier läge eine Prozessgrenze. Die Funktion `main_receiver()` hört an ihrem Kommunikationskanal auf Funktionsaufrufe. Empfangene Aufrufe werden an entsprechende Handlerrountinen weitergeleitet. Diese Funktionalität übernimmt die Stubfunktion `process_sub_com_if1()`.

Für jede Schnittstelle gibt es ein zugehöriges Proxy- und ein Stubmodul. Die gemeinsame Schnittstelle liegt in einem Headerfile.

Listing 3.3 zeigt das Headerfile für die Schnittstelle `com_if1`.

Listing 3.3 Headerfile `com_if1.h`

```

#ifndef INC_com_if1
#define INC_com_if1
enum enumCOM_IF1
{
    enCOM_IF1_FOO1,
    enCOM_IF1_END
};
void process_stub_com_if1(int fd);
void foo1(int value1 , int value2);
#endif

```

Listing 3.4 zeigt die Realisierung des Proxymoduls zu `com_if1`.

Listing 3.4 Proxymodul `com_if1_proxy.c`

```

#include <stdio.h>
#include <io.h>
#include <com_if1.h>
void foo1(int fd,int value1 , int value2)
{
    write(fd,&enCOM_IF1_FOO1,sizeof(int));
    write(fd,&(value1),sizeof(int));
    write(fd,&(value2),sizeof(int));
}

```

Listing 3.5 zeigt die Realisierung des Submoduls zu `com_if1`.

Listing 3.5 Stubmodul `com_if1_stub.c`

```

#include <stdio.h>
#include <io.h>
#include <com_if1.h>

```

```

void foo1Decode(int fd)
{
    /*decl parameters*/
    int value1;
    int value2;
    /*read parameters*/
    read(fd,&(value1),sizeof(int));
    read(fd,&(value2),sizeof(int));
    /*call handler*/
    foo1Handler(value1 , value2)
}
void process_stub_com_if1(int fd)
{
    int func_id;
    read(fd,&(func_id),sizeof(int));
    switch(func_id)
    {
        case enCOM_IF1_FOO1:
            foo1Decode(fd);
            break;
        default:
            break;
    };
}
/*****/
/* user defined code
/*****/
void foo1Handler(int value1 , int value2)
{
    /*{{foo1Handler*/
    printf("foohandler: %d %d" ,value1 ,value2);
    /*}}foo1Handler*/
}

```

Wie Sie in *Listing 3.5* sehen, liest `process_stub_com_if1()` zunächst die Funktions-ID. Je nach Funktions-ID wird eine entsprechende Decodierfunktion aufgerufen (hier `foo1Decode()`), die ihrerseits eine entsprechende Handleroutine (hier `foo1Handler()`) aufruft.

Die Funktion `foo1Handler()` ist zwar von ihrem Funktionskopf her schematisiert bekannt, ihr Rumpf ist allerdings individuell. Dieser Code ist zwischen zwei speziellen Kommentarzeilen platziert. Diese werden später vom Generator erhalten bleiben, auch wenn sich z.B. ein Parameter der Funktion in der `peer_interface.model` (*Listing 3.1*) ändern sollte.

3.4 Generator zur Transformation

Wie bei der Referenzimplementierung noch einmal konkret deutlich wurde, läuft die Kommunikation – bis auf die individuelle Programmierung der Handleroutinen an sich – schematisiert ab. Dieses Schema kann nun in einem Generator hinterlegt werden.

Wie Sie in *Kapitel 2.1* gesehen haben, wird bei HuGo für jedes XML-Tag der `.model`-Datei eine Frame-Datei hinterlegt. Die Frame-Datei für ein Interface (`<com_if>`) zeigt *Listing 3.6*.

Listing 3.6 Frame-Datei `com_if.jxgp`

```
package c;
public class com_if extends base
{
    @slot decl_proxy;
    @slot impl_proxy;
    @slot decl_enum;
    @slot impl_stub;
    @slot impl_stub_handler;
    @slot case_stub;
    public @String name;
    public @conString con_proxy_name;
    public @conString con_stub_name;
    public @conString con_base_name;
    @constructor
    {
        con_base_name=name;
        con_proxy_name=name+"_proxy";
        con_stub_name=name+"_stub";
    }
    @gen MODUL_INC<!name+".h", "inc", "INC"!>
    <#cpp
        #ifndef INC_<!name!>
        #define INC_<!name!>
        enum enum<!enum_name(name)!>
        {
            @decl_enum
            en<!enum_name(name)!>_END
        };
        void process_stub_<!name!>(int fd);
        @decl_proxy

        #endif
    #>
    @gen MODUL_SRC_PROXY<!con_proxy_name+".c", "src", "SRC"!>
    <#cpp
        #include <stdio.h>
        #include <io.h>
        #include <<!name!>.h>
        @impl_proxy
    #>
    @gen MODUL_SRC_STUB<!con_stub_name+".c", "src", "SRC"!>
    <#cpp
        #include <stdio.h>
        #include <io.h>
        #include <<!name!>.h>
        @impl_stub
        void process_stub_<!name!>(int fd)
        {
            int func_id;
            read(fd, &(func_id), sizeof(int));
```

```

        switch(func_id)
        {
            @#case_stub
            default:
                break;
        };
    }
    /*****
    /* user defined code
    /*****
    @#impl_stub_handler

    #>
}

```

In `com_if.jxgp` gibt es drei `@#gen` Methoden, denn wie die Referenzimplementierung zeigte, müssen pro Schnittstelle drei Dateien angelegt werden:

- Das Headerfile `.h`
- Ein Proxymodul `.c`
- Ein Stubmodul `.c`

Pro Funktion in der Schnittstelle (`<func>` in `peer_interface.model`) müssen angelegt werden:

- Eine Funktions-ID in der Enum-Struktur der Schnittstelle
- Eine Deklaration der Funktion im Headerfile
- Eine Definition der Funktion im Proxymodul
- Eine Decodierfunktion im Stubmodul
- Eine `case`-Verzweigung im Stubmodul
- Ein Skelett einer Handleroutine im Stubmodul

Listing 3.7 zeigt die Realisierung durch die HuGo Frame-Klasse.

Listing 3.7 Frame func.jxgp

```

package c;
public class func extends base
{
    @slot_list<!"!"> param;
    @slot_list<!"!"> call_param;
    @slot    init_param;
    @slot    save_param;
    @slot    decl_param;
    public @String name;
    public @String type="void";
    public String enum_name;
    public String stub_name_decode;
    public String stub_name_handler;
    public @conString con_base_name;
    @constructor
    {
        enum_name="en"+enum_name(con_base_name+"_"+name);
        stub_name_decode=name+"Decode";
        stub_name_handler=name+"Handler";
    }
}

```

```

    @#gen decl_enum
    <#cpp
        <!enum_name!>,
    #>
    @#gen case_stub
    <#cpp
        case <!enum_name!>:
            <!stub_name_decode!>(fd);
            break;
    #>
    @#gen impl_stub
    <#cpp
        void <!stub_name_decode!>(int fd)
        {
            /*decl parameters*/
            @#decl_param
            /*read parameters*/
            @#init_param
            /*call handler*/
            <!stub_name_handler!>(@#call_param);
        }
    #>
    @#gen impl_stub_handler
    <#cpp
        void <!stub_name_handler!>(@#param)
        {
            @#coding<!stub_name_handler!>
        }
    #>
    @#gen impl_proxy
    <#cpp
        void <!name!>(int fd,@#param)
        {
            write(fd,&<!enum_name!>,sizeof(int));
            @#save_param
        }
    #>
    @#gen decl_proxy
    <#cpp
        void <!name!>(@#param);
    #>
}

```

@#coding

In der Methode @#gen impl_stub_handler wird die Hugo-Anweisung @#coding verwendet. Sie bewirkt das Einfügen eines Coding-Blocks. Wird an dieser Stelle im Nachfeld der Generierung Code eingefügt, so bleibt dieser bei erneutem Generatorlauf erhalten – geschützt durch einen speziellen Kommentar im Sourcefile (siehe *Listing 3.5*).

Die Generiermethoden für das <param>-Tag zeigt *Listing 3.8*.

Listing 3.8 Die Frame-Datei param.jxgp

```

package c;
public class param extends base
{
    public @#String name;
}

```

```
public @#String type;
@#constructor { }

@#gen param <#cpp <!type!> <!name!> #>

@#gen call_param <#cpp <!name!> #>

@#gen decl_param
<#cpp
    <!type!> <!name!>;
#>
@#gen init_param
<#cpp
    read(fd, &<!name!>, sizeof(<!type!>));
#>
@#gen save_param
<#cpp
    write(fd, &<!name!>, sizeof(<!type!>));
#>
}
```

3.5 Fazit und Ausblick

3.5.1 Ein erstes Prozessmodell

Wenn Sie das Vorgehen bei diesem Beispiel rekapitulieren, entdecken Sie bereits ein kleines Vorgehensmodell bei der Entwicklung von Generatoren:

1. DSL-Modell entwerfen
2. Basisarchitektur konzipieren
3. Referenzimplementierung erstellen
4. Generator(-Frames) ableiten

Dieses Modell wird sich in den folgenden Beispielen immer wieder durchziehen.

Wir möchten es an dieser Stelle bereits einmal in einen größeren Kontext rücken, damit Sie den Zusammenhang mit der Trennung von Applikations- und Domänen-Entwicklung nicht verlieren. *Abbildung 3.2* zeigt diesen Zusammenhang.

Die vorher genannten vier Schritte sind in *Abbildung 3.2* mit schwarzen Kreisen markiert. Genau genommen handelt es sich hier nicht um Schritte, sondern um die Ergebnisse („Artefakte“) von Schritten aus dem Domänen-Entwicklungsstrang.

Der linke Bereich in *Abbildung 3.2*, der Applikations-Entwicklungsstrang, ist grau dargestellt. Er dient lediglich dem Kontext.

Etwas distanziert betrachtet: Worum geht es im Kern der Generativen Programmierung? Analog aufgebaute Software-(Teile) werden aus einer Art Baukasten automatisch zusammengesetzt und produziert. Hierzu werden Basis-

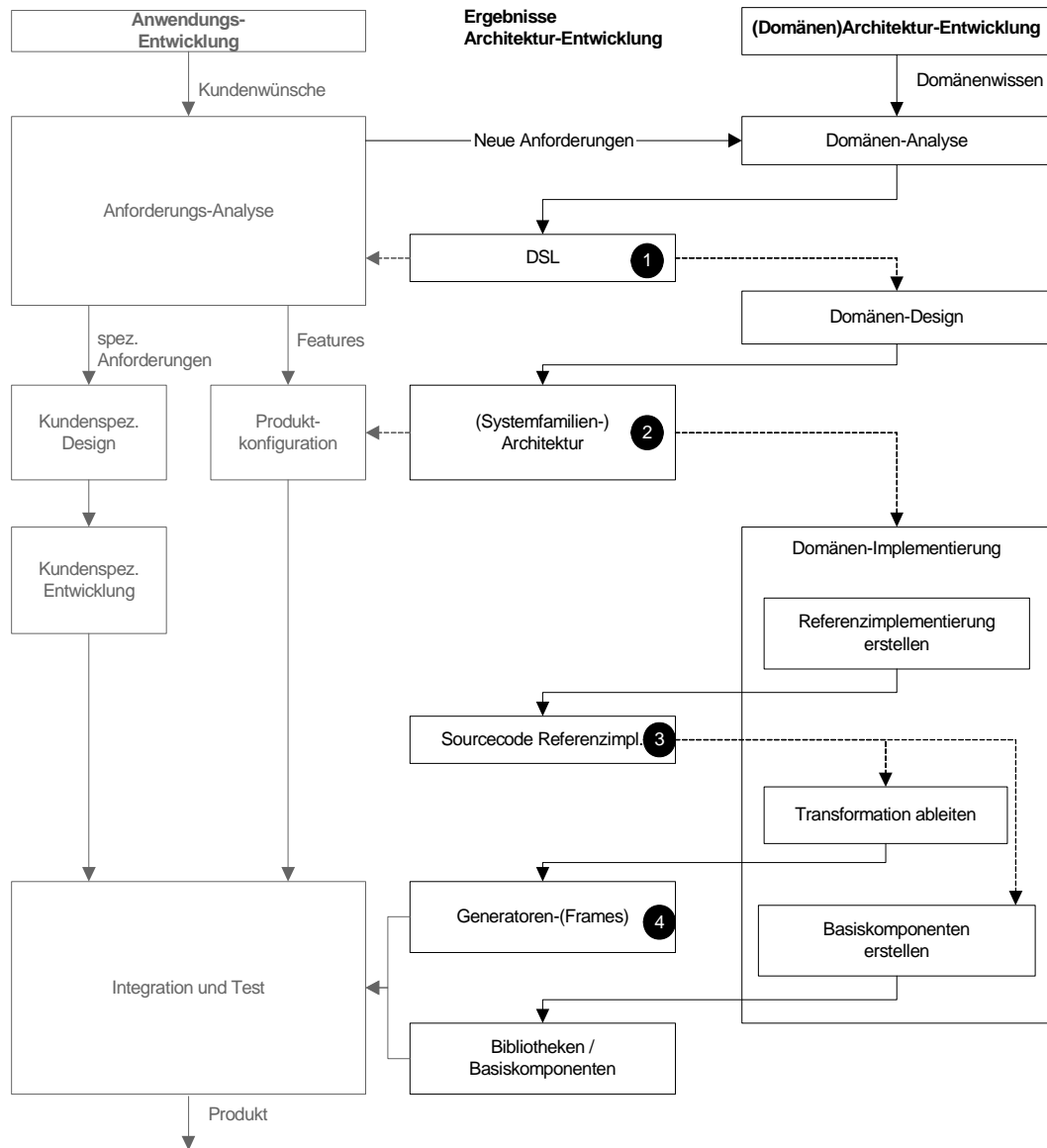


Abbildung 3.3 Ein erstes Prozessmodell

komponenten benötigt sowie eine Logik für eine Transformation, wie aus diesen Basiskomponenten eine Applikation erstellt werden kann. Diese Logik liegt in Form von Generatoren vor. Damit dies möglich ist, wird eine gute Software-Architektur benötigt, die einen automatischen Zusammenbau zulässt.

Die Architektur spielt eine entscheidende Rolle. Je gleichförmiger sie aufgebaut ist, desto besser lässt sie sich generieren. Gleichförmige Architekturen haben außerdem den Vorteil, dass sie transparenter sind, weswegen man z.B.

auch unabhängig von Generierung in der Software-Entwicklung über Entwurfsmuster nachgedacht hat.

Achten Sie in Architekturskizzen auf Gleichförmigkeiten und in Texten auf Begriffe wie „Schema“, „analog“, „entsprechend“, „zugehörig“. Sie sind stets ein Indiz für Generierungspotenzial und helfen Ihnen, die richtige Architektur und die richtigen Transformationen zu finden.

Erstellen Sie eine Referenzimplementierung und markieren Sie sich die Stellen, die schematisiert sind. So finden Sie sehr schnell zugehörige Generatoren(-Frames).

Neben den Generatoren stellt der Domänenarchitektur-Entwicklungsstrang auch fertig implementierte Komponenten für die Basisfunktionalität in der Domäne zur Verfügung. In unserem Beispiel könnten dies Module zur nativen Übertragung der Bytes über RS232 sein.

Bei dem Prozessmodell aus hätten Sie unabhängig von unserem Beispiel spontan vielleicht an eine große Softwarelösung gedacht (z.B. ein Warenwirtschaftssystem). Mit unseren kleinen Beispielen wollen wir jedoch zeigen, dass Generieren schon in Teilbereichen von Softwareapplikationen große Vorteile bringen kann, das heißt: das generative Prozessmodell lässt sich nicht nur im Großen, sondern auch im Kleinen anwenden.

Abgesehen davon, lassen sich kleine Beispiele in einem Buch leichter nachvollziehen. Und wenn Sie das Prinzip erst einmal verinnerlicht haben, werden Sie es sicherlich auch auf große Softwaresysteme übertragen können.

3.5.2 Finden der „richtigen“ DSL

Ein zentraler Aspekt bei Generativer Softwareentwicklung ist die Tatsache, dass die Modelldatei eine semantisch domänenspezifische Sprache, eben eine DSL, verwendet.

Nomenklatur und die Struktur der XML-Tags sollten dem Denkschema der Person(en) entsprechen, die das konkrete Modell spezifizieren.

Hören Sie hier den Fachleuten der Domäne genau zu.

- Welche Begriffe verwenden sie? Verwenden Sie diese Begriffe als XML Tag- und Attributnamen.
- Welche Strukturen verwenden die Fachleute bei der Beschreibung von Sachverhalten? Orientieren auch Sie sich mit Ihrer XML-Struktur daran.

Je besser Sie das Denkschema der Fachleute abbilden können, desto sicherer ist es, dass sie anfangs eventuell vergessene Aspekte später im Modelldateischema noch unterbringen.

In unserem Beispiel war dies sehr einfach, da es um die Spezifikation von RPCs ging. Hier liegt die DSL sehr nahe an einer Programmiersprache.

3.5.3 Die Rolle des Modellierers

In *Kapitel 1* haben wir zwei Rollen bei der Entwicklung herausgehoben:

- Der Applikations-Entwickler erstellt auf Basis der vorhandenen Infrastruktur die konkrete Applikation.
- Der Domänen-Entwickler erstellt eine Infrastruktur mit entsprechenden Basisfunktionalitäten und Generatoren, sodass der Applikations-Entwickler effizient zum Ziel kommt.

Zentraler Bestandteil beider Rollen ist die Tatsache, dass sich die Software-Entwicklung bei Generativer Programmierung prinzipiell in zwei Entwicklungsstränge aufteilt: den Applikations-Entwicklungs- und den Domänen-Entwicklungsstrang.

An dem vorgestellten Beispiel erkennt man nun sehr gut, dass der Applikations-Entwickler ebenfalls zwei Rollen übernimmt: die des Modellierers und die des Entwicklers, sodass man grob die folgenden drei Rollen unterscheiden kann:

- **Domänen-Entwickler/Architekten** entwickeln die prinzipielle, auf die Domäne zugeschnittene Softwarearchitektur (im Beispiel: die Frames für die Erzeugung von Proxy und Stubmodul).
- **Modellierer/Designer** modellieren mit der DSL die konkrete Applikation/das konkrete Applikationsgerüst (im Beispiel: Editieren der `peer_interface.model` (*Listing 3.1*), als Vorgabe für den Generator, um konkrete Proxies und Stubmodule zu erzeugen).
- **Entwickler** kodieren die Fachlogik und integrieren sie in den generierten Infrastrukturcode (im Beispiel: Coding-Blocks in den Handlerrouinen).

Abbildung 3.3 visualisiert die drei Rollen in Bezug auf unser Beispiel.

Die Rolle des Modellierers könnte hier z.B. auch der Teamleiter übernehmen, zur Spezifikation der Schnittstelle benötigt er nur wenig Programmierkenntnisse. Er muss dann lediglich mit einer prinzipiellen Signaturbeschreibung von Funktionen vertraut sein.

Je entfernter die DSL von der tatsächlichen Codeumsetzung ist, desto weniger Programmierkenntnisse benötigt der Modellierer. Die DSL könnte ja auch schrittweise (bei XML-Syntax über XSLT-Skripte¹) übergeführt werden in eine DSL, die dann erst als Input für die Frames dient.

Wir möchten an dieser Stelle darauf hinweisen, dass der Modellierer niemand sein muss, der ein UML-Tool bedienen können muss. Eine selbsterklärende

¹ XSLT-Skripte definieren Regeln, wie ein XML-Dokument in ein anderes XML-Dokument oder Text-dokument übergeführt wird. Ein XSLT-Prozessor bearbeitet die XSLT-Skripte. XSLT-Skripte verwenden als Syntax selbst XML.

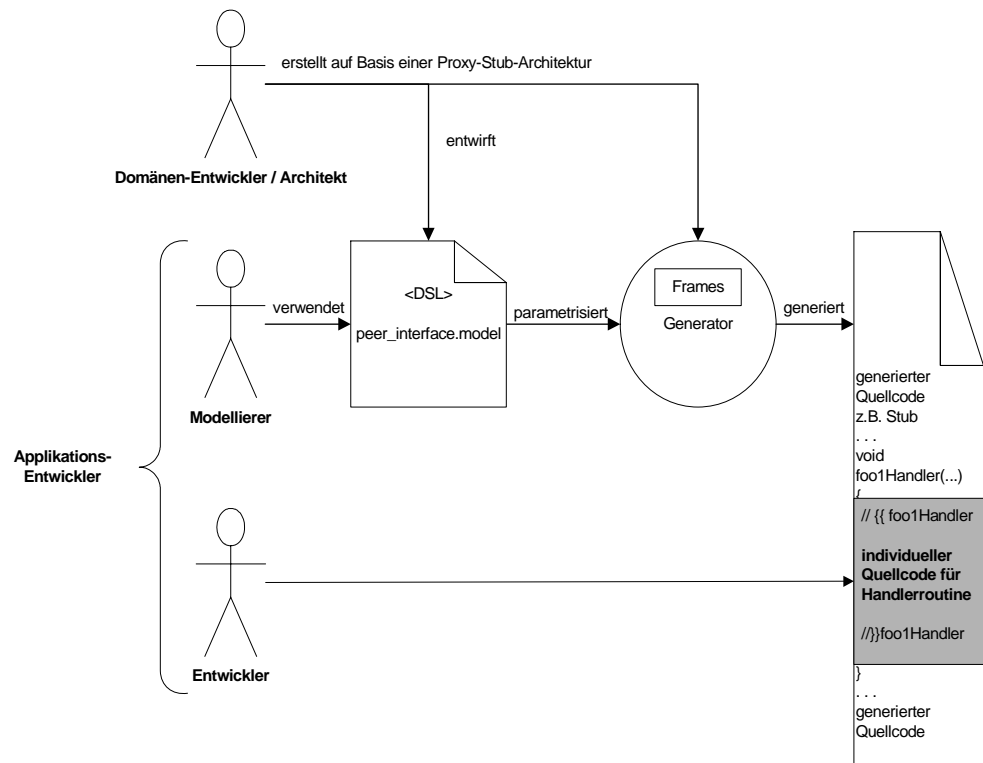


Abbildung 3.3 Die Rolle des Modellierers

XML-Datei oder ein spezifischer Editor ist oft viel hilfreicher als ein UML-Modell, um die Sachverhalte zu spezifizieren. Was nicht heißt, dass wir UML-Modelle und daraus generierten Code nicht gut fänden. Entscheidend ist jedoch: Wer ist der Modellierer? Welche Sprache spricht er?

3.5.4 Validierung des Modells

Bei der Editierung der XML-Datei `peer_interface.model` könnten natürlich auch Fehler passieren. Die Fehlermöglichkeiten können Sie jedoch z.B. durch folgende Maßnahmen herabsetzen:

- Sie erstellen eine *XML-Schema-Datei oder DTD*. Mit einem Schema können Sie z.B. überprüfen, ob ein Attribut oder ein Sub-XML-Tag an der Stelle erlaubt bzw. erforderlich ist. Sie können jedoch z.B. nicht Plausibilitäten zwischen einzelnen Attributen überprüfen, z.B. wenn `Attribut1="int"`, dann darf `Attribut2` nicht "3.5" sein. XML-Schema
- Sie erstellen einen *Editor* zu Ihrer XML-Datei. Der Editor kann von vornherein nur sinnvolle Eingaben zulassen. Außerdem kann durch zusätzliche Benutzerführung eine bessere Benutzbarkeit (neudeutsch: „Usabili- Editor

ty“) erreicht werden. Gerade im vorliegenden Beispiel könnte man sich sehr gut einen formularbasierten Editor vorstellen, um die Schnittstelle zu spezifizieren.

Prüfungen
in Frames

- Sie implementieren *in Ihren Generatorframes Plausibilitätsprüfungen*. Da diese jedoch bei jedem Generatorlauf durchgeführt werden, kann Ihr Generator dann stark an Performance verlieren.

Egal, welche Maßnahmen Sie hier ergreifen, Sie müssen– wie überall – abwägen, ob der Nutzen den Aufwand rechtfertigt. Hilfreiche relevante Fragen können sein:

- Wie viele Personen editieren die Modelldatei?
- Wird eine Dokumentation zum Schema der Modelldatei benötigt (auch abhängig von der Anzahl der Nutzer)
- Welche Fehler haben welche Konsequenzen? Rechtfertigen diese Konsequenzen den Mehraufwand für eine Validierung des Modells?
- Welche Fehler können durch welche Maßnahmen abgestellt werden? Z.B. kann ein XML-Schema keine semantische Validierung übernehmen!
- Welchen Aufwand bedeutet es im vorliegenden Fall konkret, das Schema der Modelldatei verbal zu formulieren oder ein XML-Schema zu erstellen oder einen Editor einzusetzen?
- Welchen Aufwand bedeutet es, die Maßnahme entsprechend anzupassen, wenn sich das Modell im Laufe der Entwicklungszeit weiterentwickelt/ändert? Kann eventuell der Generator so angepasst werden, dass er die entsprechenden Maßnahmen, z.B. entsprechende Änderungen im Editor, gleich mit generiert?

3.5.5 Vorteile von Generieren

Das Beispiel des kleinen, verteilten Systems veranschaulicht praxisnah einige zentrale Vorteile von Generieren gegenüber „manueller“ Realisierung:

Pflegbarkeit

- *DRY-Prinzip/Queraspekte*: Die Schnittstellenfunktionen werden an einer Stelle, in der `peer_interface.model`, spezifiziert. Daraus werden sowohl das zugehörige Proxy-Modul als auch das Stub-Modul generiert. Solche Queraspekte, die sich über mehrere Module erstrecken, lassen sich mit einfachen Template-Funktionen, wie sie z.B. die Sprache C++ enthält, nicht mehr abbilden. Hier wird ein Generator benötigt. Er hält Änderungen am System (neuer Parameter, neue Schnittstellenfunktion, ...) leicht pflegbar.

Zuverlässigkeit

- *Zuverlässige Vergabe von IDs*: Damit die Handlerrouninen den Telegrammen eindeutig zugeordnet werden können, werden IDs, am besten in Form eines `enums`, verwendet. Wird bei manueller Realisierung der Eintrag im

enum vergessen, so wird dieses Telegramm nie richtig bearbeitet. Dies resultiert vielleicht in einen latenten Fehler, der wahrscheinlich nur bei einem Telegramm vergessen wurde. Natürlich können auch im Generator Fehler liegen. Ein Fehler im Generator stellt aber einen systematischen Fehler dar; denn ein Generator macht entweder alles richtig oder alles konsequent falsch.

- *Performance*: Varianten zu generieren, ist schneller als ein interpretierender Ansatz, bei dem die Telegramme konfiguriert werden (z.B. über ein Array mit Funktionspointern für die Handlerrountinen). Performance
- *Typsicherheit*: Flexibilität, die auf Interpretation basiert, benutzt generische Datentypen wie z.B. void*, bei der Zuweisung können hier sehr leicht Fehler vorkommen, weil der Compiler die Typsicherheit nicht überprüfen kann, z.B. bei der Zuweisung des Funktionspointers für die Handlerroutine im Array. Ein Generator generiert einfach den entsprechenden Typ. Auf generische Datentypen kann weitestgehend verzichtet werden. Die typsichere Schreibweise ist zudem kürzer, weil auf Cast-Operatoren verzichtet werden kann. Typsicherheit
- *Plattformunabhängigkeit*: Auf Basis der Spezifikation in `peer_interface.model` könnte die Realisierung in C/C++ (wie gezeigt) erfolgen, aber auch in jeder anderen Programmiersprache. So könnten aus einem DSL-Modell diverse Komponenten auch für unterschiedliche Programmiersprachen generiert werden, die miteinander kommunizieren. (Dass dies möglich ist, hat das Beispiel der Klassengenerierung für Java und C/C++ aus *Kapitel 2* ja bereits skizziert.) Plattform-unabhängigkeit

Etliche dieser Punkte können gerade im Embedded-Umfeld schlagkräftige Argumente für den Einsatz generativer Programmierung sein.

3.5.6 Potenzielle Erweiterungen des Beispiels

Das Beispiel wurde bewusst sehr klein gewählt, in der Praxis könnten die Generator-Frames z.B. noch um folgende Aspekte erweitert werden:

- Einbau eines Fehlerhandlings. Was passiert z.B., wenn während der Übertragung der Stecker herausgezogen wird?
- Berücksichtigung von Übertragungen über Hardware-Architekturen hinweg, z.B. 80x86/680xx Hardware-Architekturen.
- Mitgenerierung einer Monitoring-Software, die den Telegrammverkehr visualisiert. Dies ist besonders bei BUS-Telegrammen interessant, bei denen nicht nur eine Punkt-zu-Punkt-Kommunikation vorliegt.

Bei Implementierung dieser weiteren Aspekte würden Sie ebenso wieder Stellen entdecken, an denen Querbeziehungen zwischen Modulen vorhanden sind.