Von der Grafik zum Quellcode

Michael Klar
Klar Automation GmbH
Marktplatz 7
91475 Lonnerstadt
michael.klar@klar-automation.de
www.klar-automation.de

Susanne Klar
Klar Automation GmbH
Marktplatz 7
91475 Lonnerstadt
susanne.klar@klar-automation.de
www.klar-automation.de

Abstract

Eine Herausforderung in der Software-Entwicklung von grafischen Benutzungsoberflächen ist ein vorgegebenes Screen-Layout mit möglichst wenig Aufwand optimal umzusetzen.

Gerade im Desktop und Embedded Bereich werden die Vorgaben meist manuell umgesetzt. Hier kann es zu Diskrepanzen kommen zwischen den Vorgaben eines Ergonomen oder Grafikers und dem, was technisch machbar ist und zugleich einen vertretbaren Aufwand hat. Dieser Beitrag beschreibt ein in der Praxis bewährtes Vorgehen, wie das Zusammenspiel zwischen grafischen Vorgaben und deren Umsetzung für das Zielsystem durch Automatisierung optimiert werden kann.

Dieses Vorgehen basiert auf der Methodik generativer / modellgetriebener Software-Entwicklung, bei der aus einem Modell – in dem Fall der schematisierten Beschreibung einer dynamisch veränderbaren Grafik – automatisch Quellcode für das Zielsystem erzeugt wird. Dieser ist dann im Zielsystem sofort einsatzfähig.

Keywords

Grafische Benutzungsoberflächen, Automatische Codegenerierung, Software Prozessoptimierung, MMI / HMI / HCI / MCI

1.0 Herausforderung

Um eine ansprechende Benutzungsoberfläche in der Software zu erhalten, werden in Projekten oftmals Ergonomen oder Grafiker eingebunden. Sie erstellen Screen-Layouts, welche die Entwickler dann in Software umsetzen.

1.1 Realisierbarkeit auf Zielsystem

Oftmals gibt das Zielsystem, auf dem die Software später läuft, technische Einschränkungen vor. Beispielsweise könnte ein Embedded System nur Grafikprimitive oder nur Bitmaps ohne Transparenz zulassen. Das bedeutet, dass hier – unter Umständen auch mehrere – Abstimmungen zwischen Grafiker und Entwickler erforderlich sind.

Der Schwerpunkt des Grafikers liegt auf Ästhetik und Usability, der Schwerpunkt des Entwicklers auf technischer Machbarkeit. Meist verwenden Grafiker Grafikprogramme, die oftmals wesentlich mehr Funktionen besitzen, als auf dem technischen Zielsystem umgesetzt werden können. Kennt der Grafiker die technischen Einschränkungen nicht von Beginn an, so kann es mehrere Feedback-Schleifen benötigen, bis beide, Entwickler und Grafiker, mit dem Ergebnis zufrieden sind. Auch Speicher oder Performance (z.B. für das Laden einer komplexen Grafik) können – gerade auch bei Embedded Systemen – ein K.O. Kriterium für eine bestimmte Art von Grafik sein.

Mögliche Diskrepanzen zwischen Grafikvorgabe und technisch machbarer Umsetzung liegen vor allem in den Einschränkungen der verwendeten Grafikbibliotheken begründet und betreffen nicht nur Embedded Systeme.

1.2 Feedbackschleifen verkürzen

Betrachtet man Abbildung 1, so ist es für eine Prozessoptimierung wichtig, die Feedback-Schleifen zwischen Grafiker und Entwickler zu verkürzen. D.h. der Grafiker muss möglichst frühzeitig die Einschränkungen des Zielsystems kennen, um diese in seinen Entwürfen entsprechend berücksichtigen zu können.

Oftmals stellt sich aber erst während der Realisierung heraus, dass sich das vorgegebene Screen-Layout so nicht umsetzen lässt. Gründe hierfür können sein:

- Der Entwickler hat vergessen, dem Grafiker mit zu teilen, welche Einschränkungen das Zielsystem hat.
- Der Entwickler hat Stellen übersehen, die grafische Einschränkungen bedeuten.
- Der Entwickler hat zu wenig Erfahrung in der Umsetzung von grafischen Benutzeroberflächen und hat die Umsetzung unterschätzt.

1.3 Umsetzungsaufwand reduzieren

Ein weiterer Punkt, der nicht selten unterschätzt wird, ist der Aufwand für die Umsetzung eines individuellen Screen-Layouts. Der Grafiker weiß oftmals nicht, was eine bestimmte grafische Vorgabe an Aufwand für den Entwickler bedeutet.

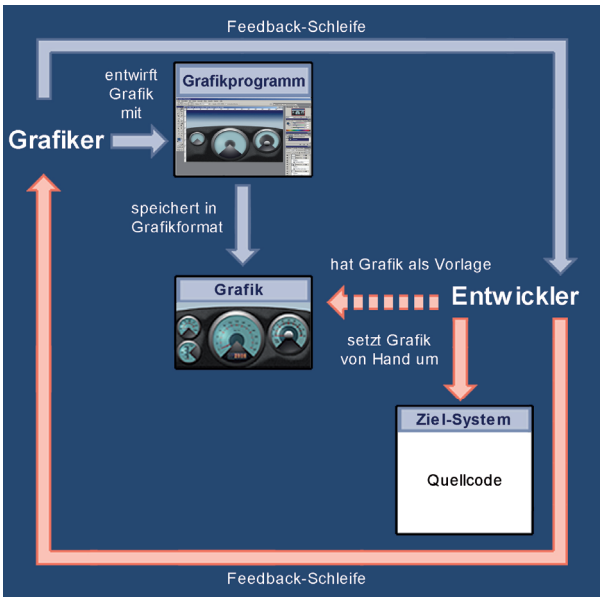

Abbildung 1: Manueller Umsetzungsprozess

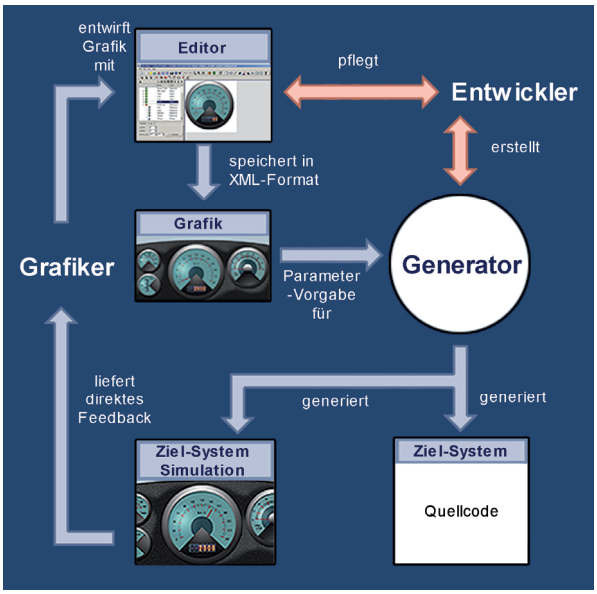

Abbildung 2: Automatischer Umsetzungsprozess

Unter Umständen wäre eine andere, auch mögliche Vorgabe mit weniger Aufwand in der Umsetzung verbunden.

2.0 Lösung: Generativer Ansatz

Ein Weg, um die technische Machbarkeit von Anfang an überprüfen zu können und den Entwicklungsaufwand für grafische Benutzeroberflächen zu reduzieren, liegt in einem generativen Ansatz.

Hier wird der Grafiker direkt mit in den Erstellungsprozess eingebunden. Er liefert die Grafik in einer Beschreibungssprache, aus der auf Knopfdruck direkt Quellcode für das Zielsystem generiert werden kann. Diese Beschreibungssprache liegt hinter einem Editor, so dass der Grafiker damit nicht konfrontiert wird und die Erstellung für ihn intuitiv erfolgen kann.

Abbildung 2 visualisiert dies. Die folgenden Unterkapitel gehen auf die einzelnen Komponenten der Abbildung näher ein.

2.1 Editor mit passenden Features

Der Editor ist nicht nur benutzungsfreundlich, sondern enthält auch nur genau die Features, die im Zielsystem umgesetzt werden können. Auf diese Weise ist dem Grafiker von Anfang an klar, was möglich ist. Sollte er eine Funktion vermissen, so wird er gezielt auf den Entwickler zugehen. Dieser kann dann sofort Feedback geben, ob sich dieses weitere grafische Feature auch in das Zielsystem überführen lässt. Damit entsteht automatisch ein gesteuerter Feedback-Prozess.

2.2 Generator für Ziel-Quellcode

Der Editor speichert die erstellte Grafik bzw. das Screen-Layout in einer Beschreibungssprache, die als Parametervorgabe für den Generator dient. (Als Parametersprache für den Generator wird als Syntax oftmals XML verwendet.)

Auf Knopfdruck kann nun die so gespeicherten Grafik direkt in Quellcode umgesetzt werden.

Die Logik für die Umsetzung im Generator erstellt der Entwickler. Dies ist ein einmaliger, initialer Aufwand, der lediglich pro Zielsystem entsteht.

Ist der Generator für ein Zielsystem fertig, kann er für beliebige Grafiken / Screens eingesetzt werden. Die Erstellung der Grafiken erfolgt sodann automatisch durch den Generator und beansprucht hier keine weitere Entwicklungsleistung.

Der initiale Aufwand für den Generator kann sich sehr schnell rentieren, je nachdem wie viele individuelle Screen-Layouts damit erstellt werden und wie aufwendig diese wären, wenn sie per Hand umgesetzt würden.

2.3 Generator für Simulation

Wird zusätzlich zum Quellcode eine Simulationsumgebung generiert, kann der Grafiker das Ergebnis auch überprüfen, was vor allem im Hinblick auf animierte Grafiken (z.B. Rotationen von Skalen-Zeigern) oder Eventreaktionen (z.B. Mausklicks) als sehr gutes Feedback dienen kann.

2.4 Resultierende Vorteile

Zusammenfassend lassen sich folgende Vorteile eines solchen Systems festhalten:

- Die Kommunikation zwischen Grafiker und Entwickler läuft gezielt. Wenn der Editor ein benötigtes Feature nicht bereitstellt, spricht der Grafiker zwangsläufig mit dem Entwickler.
- Die Feedbackschleife erfolgt direkt (siehe Abbildung 2).
- Das Screen-Layouts wird automatisch in Quellcode umgesetzt, d.h. nach Initialaufwand für den Zielsystem-Generator entsteht auf Entwicklerseite kein weiterer Aufwand für die optische Gestaltung eines Screens.

2.5 Resultierender Prozess

Der Entwicklungsprozess stellt sich damit wie folgt dar:

1. Entscheidung für das technische Zielsystem
2. Entwickler: Entwicklungsumgebung schaffen, d.h. Generator für Zielsystem und Simulationsumfeld erstellen, falls noch nicht vorhanden. Editor gegebenenfalls um zusätzliche Features erweitern.
3. Grafiker: Screen-Layouts unter Verwendung des Editors erstellen.

3.0 Umsetzung eines Screen-Generator

Vielleicht schreckt der eine oder andere vor der Erstellung eines Generators zurück oder fragt sich, wie dies mit wenig Aufwand geschehen kann.

Generatorensysteme arbeiten in der Regel so, dass es Codevorlagen gibt, die festlegen, wie der Quellcode für das Zielsystem aussieht. Diese Codevorlagen sind parametrisiert. Die Hauptaufgabe eines Generatorentwicklers besteht darin, ein Schema zu erkennen, um daraus Codevorlagen ableiten zu können.

Ist das Schema einmal herausgearbeitet, so lassen sich hierfür relativ schnell Codevorlagen erstellen.

3.1 Schema einer Grafik

Eine Grafik lässt sich dadurch gut schematisieren, indem man sie als Vektorgrafik auffasst. Ihre Zeichenelemente (Linien, Texte, Farben, Rotationen,....) lassen sich so als Baum hierarchisch anordnen. Hierüber kann die Grafik vollständig beschrieben werden.

Interessant aus Softwaresicht sind jedoch nicht statische Grafiken, sondern von der Software veränderbare Grafiken, z.B. Skalen, deren Wert durch die Software vorgegeben wird oder Mausreaktionen, die an die Software weitergeleitet werden. Damit benötigt der Baum auch weitere Elemente, die sich auf die Software-Komponente beziehen, d.h. die Angabe von Variablen, die damit verbunden sind, oder Klickregionen etc.

3.2 Die Rolle des Grafikers

Der Grafiker, den wir in den Prozess integriert haben, entpuppt sich damit ansatzweise als „Zwitter“. Auf der einen Seite entwirft er Komponenten aus ästhetisch und ergonomischer Sicht, auf der anderen Seite muss er die Schnittstelle zur Software kennen, d.h. welche Variablen sollen weitergegeben werden und welche Ereignisse sollen mit der Grafik verknüpft werden. Um dies definieren zu können, muss er aber keinerlei Programmierkenntnisse besitzen. Kenntnisse in Vektorgeometrie reichen hier völlig aus, sowie ein Gefühl für die Schnittstelle zur Software, die aber im Hinblick auf Ergonomie ohnehin wichtig ist, z.B. wenn es darum geht, wo sinnvollerweise Klickregionen liegen.

Die Akzeptanz des Tools (d.h. Editor mit Quellcode-Generator für Zielsystem) hängt dabei hauptsächlich vom Selbstverständnis des Grafikers ab. Versteht er sich als Person, deren Arbeit beendet ist, wenn die statischen Entwürfe vorliegen oder macht er sich auch Gedanken um den gesamten ergonomischen Ablauf der Benutzungsoberfläche, z.B. wie sich ein Button optisch ändert, wenn er gedrückt wird. Diese können mit herkömmlichen Grafikprogrammen nicht modelliert werden, sondern bedürfen eines Tools, das entsprechenden Quellcode für das Zielsystem generiert. Nur so entsteht kein Bruch zwischen Wunsch und technischer Umsetzbarkeit.

Da sich optisch anspruchsvolle, statische Bilder in herkömmlichen Grafikprogrammen am besten zeichnen lassen, sollte ein solches Generiersystem entsprechende Grafiken im pixel- oder vektororientierten Format integrieren können (vorausgesetzt das Zielsystem kann diese entsprechend darstellen). Auf diese Weise könnte z.B. im Hintergrund eines Tachometers ein optisch ansprechendes statisches Bild liegen, das mit einem herkömmlichen Grafikprogramm gezeichnet wurde. Die Skala hingegen und die Rotation des Tachometer-Zeigers, dessen Wert später durch die Software gesteuert wird, werden dann in diesem speziellen Editor entworfen. So kann die komplette Tachometerkomponente generiert werden. Das Generierprinzip lässt sich dabei nicht nur auf einzelne Oberflächenelemente übertragen, sondern auch auf bildschirmfüllende Screens, die sich wiederum aus einzelnen Elementen zusammensetzen.

Man kann die hier beschriebene Rolle des Grafikers auch auf zwei Personen aufteilen: auf eine, welche die statischen Bilder entwirft und eine andere Person, welche als grafiknaher Entwickler fungiert und mit dem Editor arbeitet. Diese Person agiert dann im Sin-

ne eines „Interaction Architects“ gemäß des Artikels „Mind the Gap!“ (Müller-Prove, 2003). Die in dem Artikel angesprochene Lücke wird dann jedoch durch das Generiersystem automatisch geschlossen; der Interaction Architect ist gleichzeitig Entwickler. Er sieht im WYSIWYG-Editor sofort wie sich die ingesamte Grafik darstellt, die 1:1 im Quellcode für das Zielsystem zur Verfügung steht.

3.3 Praxiserprobtes System

Ein solches System wird bereits innerhalb der Klar Automation GmbH bei der Entwicklung für individuelle grafische Benutzungsoberflächen sowie bei Kunden im militärischen Umfeld für Schiffs- und Flugsimulationen seit mehreren Jahren erfolgreich eingesetzt. Hier hat man Einsparungen im Bereich von Faktor 10 erreicht.

Das System ist seit Frühjahr 2006 auch als Produkt unter dem Namen GUI-Artist®[1] erhältlich. Der Editor speichert die Grafiken im XML-Format. Dieses Dokument dient als Parametrierung des dahinter liegenden Generators. Als Generatorsystem wird HuGo®[2] eingesetzt, das ebenfalls seit 2006 auch unter GPL-Lizenz erhältlich ist. Damit können auch andere individuelle Generatoren erstellt werden. Das Prinzip des GUI-Artists zeigt Abb. 3.

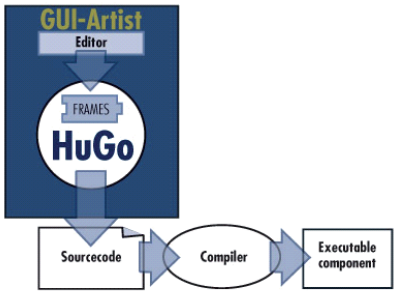

Abbildung 3: GUI-Artist Generator

[1] www.gui-artist.de
[2] www.hugosweb.net

Durch Austausch der sogenannten „Frames“ – das sind die Quellcode-Vorlagen für das Zielsystem – kann Quellcode für beliebige Zielsysteme generiert werden. Standardmäßig generiert das System C++ Quellcode für Qt, womit die Screens performant unter allen gängigen Plattformen zum Einsatz kommen können.

Das Generatorsystem ist als SOAP-Server realisiert, so dass damit auch einzelne Grafiken via Internet generiert werden können.

4.0 Hintergrund: Modellierung

Hintergrund dieses Vorgehens ist die Software-Methodik des generativen Programmierens oder die Art verwandte Methodik der modellgetriebene Software-Entwicklung. Die Grafikbeschreibung stellt hier das Modell dar. Der hier bezeichnete „Grafiker“ wäre der „Application Engineer“, der basierend auf einem Generator verschiedene Varianten einer Systemfamilie (Screen-Layouts) automatisch erstellt. Der in diesem Beitrag bezeichnete „Entwickler“ wäre der „Domain Engineer“, der spezifisch für die Domäne (Screen-Layout-Erstellung) einen Generator erstellt.

Nähere Infos zu generativer Programmierung oder modellgetriebener Software-Entwicklung finden Sie unter (Klar 2006, Stahl 2005, Krzysztof 2000).

Für größere Unternehmen ist es auch interessant, weitere Generatoren miteinander zu koppeln, um den Prozess der Software-Entwicklung durch Automatisierung firmenspezifisch noch weiter zu optimieren. Die Automatisierung mit all Ihren Qualitäts- und Produktivitätsvorteilen ist die Grundmotivation, die hinter der Entwicklung von Codegeneratoren steckt.

5.0 Fazit

Egal ob GUI-Artist oder ein anderes, eigenes System, wichtig ist die zentrale Idee, den Grafiker als „Modellierer“ mit in den Software-Entwicklungsprozess einzubinden und so im Sinne von modellgetriebener Software-Entwicklung oder generativer Programmierung Aufwand zu erniedrigen und bei höherer Qualität eine schnellere Time to Application zu erreichen.

Durch Generatoren wird die Realisierung von ansprechenden Benutzungsoberflächen bezahlbar, was vor allem bei kostensensitiven Systemen und Geräten sehr interessant ist.

6.0 Referenzen

Klar, M.; Klar S. (2006): Einfach generieren. München: Hanser Verlag

Krzysztof, C., Eisenecker, U. (2000): Generative Programming: München: Addison-Wesley Verlag.

Müller-Prove, M. (2003): Mind the Gap! Software Engineers and Interaction Architects: Workshop Paper, INTERACT 2003 www.mprove.de/script/03/interact/

Stahl, T.; Völter, M. (2005): Modellgetriebene Softwareentwicklung. Heidelberg: dpunkt.verlag.