http://www.metasafe-repository.com/Papers.html
http://www.software-modernisierung.ch/document/whitepapers/Tabu_Thema_Software-Modernisierung_2013-01-19.pdf

Modellbasiertes Software-Reengineering: Teilautomatisiert Migration eines GUI in eine Web-Umgebung

Die Migration alter Applikationen auf eine neue technische Basis ist eine echte Alternative zur radikalen Ablösung – sofern sie rasch und zumindest halbautomatisch erfolgt. Wir zeigen in diesem Artikel die Migration einer in C++ geschriebenen GUI-Applikation mit 130 Masken und mehr als 10.000 Feldern zu einer modernen Web-Umgebung auf der Basis von AngularJS. Dies gelang durch den Einsatz von Parsern, einem Metadaten-Repository und Generatoren in nur acht Monaten mit hoher Qualität und vergleichsweise bescheidenem Aufwand.

In der Praxis geht es häufig darum, bestehende Applikationen auf eine neue Plattform oder eine neue technische Basis zu migrieren. Eine rein manuelle Migration ist meist zu zeitaufwändig und zu teuer. Eine automatisierte oder teil-automatisierte Migration nutzt die vorhandenen Metadaten, um daraus die neuen Programmkomponenten zumindest teilweise zu generieren.

Unser konkretes Beispiel

Eine Client/Server-Applikation wurde vor 15 Jahren geschrieben, ging danach durch viele Entwicklerhände und erlebte entsprechend viele unterschiedliche Lösungsansätze. Die Benutzungsoberfläche umfasst über 130 in drei Sprachen ausgeführte Masken mit rund 10.000 Feldern. Die ursprüngliche Hostversion (IMS) wurde vor Jahren in ein mit Visual C++ geschriebenes Benutzerprogramm unter Verwendung des *MFC*-Frameworks (*Microsoft Foundation Classes*) migriert. Die einzige verlässliche Dokumentation waren die Quellen- und Masken-Ressourcen-Files der migrierten Version. Um einem verteilten Benutzerkreis (2.000 Benutzer) weiterhin den Zugang zu der Applikation zu ermöglichen, sollte zu einer Web-basierten Benutzungsoberfläche migriert werden. Die Masken sollten zwar geschönt, aber – um Fortbildungskosten zu vermeiden – nicht in der Struktur verändert werden.

Die Machbarkeitsstudie

Statt uns direkt in die Umsetzung zu stürzen, erstellten wir eine Machbarkeitsstudie und prüften die folgenden drei möglichen Vorgehensweisen.

Variante 1: **Manuelle technische Umsetzung**
Eine technische Dokumentation sowie die Programme und Metadaten waren praktisch die einzige Form von „Dokumentation“. Ein Neuschreiben bedingt daher die Analyse des Systems und die Erstellung von Spezifikationen. Experten schätzten den Aufwand für eine manuelle 1:1-Umstellung einer Maske auf ca. 40 bis 50 Arbeitsstunden, immer unter der Voraussetzung, dass Personen mit dem entsprechenden Know-how sowohl über das Alt-System als auch über die neue Lösung verfügbar sind. Weder war der Aufwand (6.000 Stunden) akzeptabel, noch waren solche Fachkräfte verfügbar. Daher fiel diese Lösung ohne weitere Untersuchung außer Betracht.

Variante 2: **Klassischer Parser/Generator**
Ein erstes Musterbeispiel zeigte, dass die anfängliche Aussage einer 1:1-Umstellung nicht ganz zutraf (siehe Abbildung 1). Der Kunde verlangte doch einige Korrekturen und Verbesserungen im Erscheinungsbild, wie zum Beispiel Pick-Boxen für die Datumseingabe und Auswahl-Boxen für bestimmte Attribute. Eine direkte Umsetzung der gesamten Quell-Metadaten in die Zielressourcen erwies sich als sehr komplexes Problem, weil kein einfaches 1:1-Mapping von Input- zu Output-Ressourcen möglich ist. Wir haben geschätzt, dass der XML-Zwischenspeicher für alle Daten einen Umfang von mehreren 10.000 Zeilen erreicht und entsprechend schlecht handhabbar ist. Auch diese Lösung wurde daher fallen gelassen.

Variante 3: **Modellbasierte Architektur mit einem Repository**
Wir erweiterten die ursprüngliche Vorstellung für die Architektur um ein Repository als Daten-Drehscheibe. Damit erreichten wir eine Entkopplung der Parser, der manuellen Korrekturen und der Generatoren und vereinfachten alle Komponenten stark. Bei dieser Lösung bildet das Repository mit dem Datenmodell und den Metadaten die Drehscheibe zwischen den aktiven Komponenten des Systems (siehe Abbildung 2). Im Repository sind nämlich sowohl das Datenmodell als auch die Metadaten gespeichert. Das Metamodell (des Repository) ist für beide – Modelle und Metadaten – ein *Entity-Relationship-Modell*. Daher gibt es keinen Strukturbruch zwischen dem Modell und den Metadaten.

Die Umsetzung

Wenn es in der Praxis darum geht, bestehende Applikationen auf eine neue Plattform oder auf eine neue technische Basis

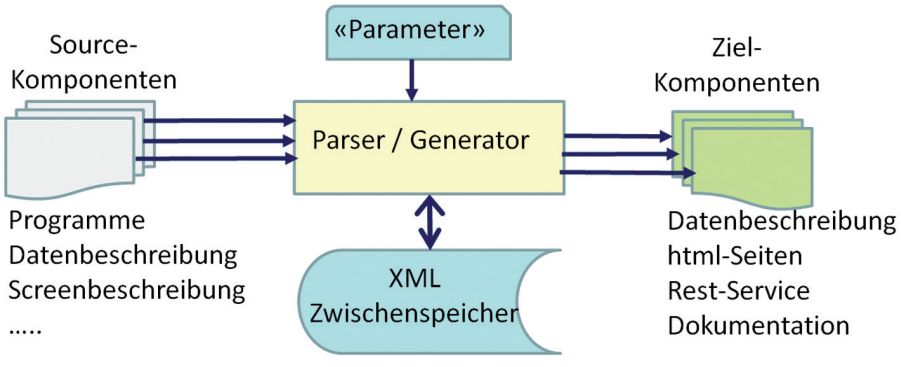

Abb. 1: Klassischer Parser/Generator.

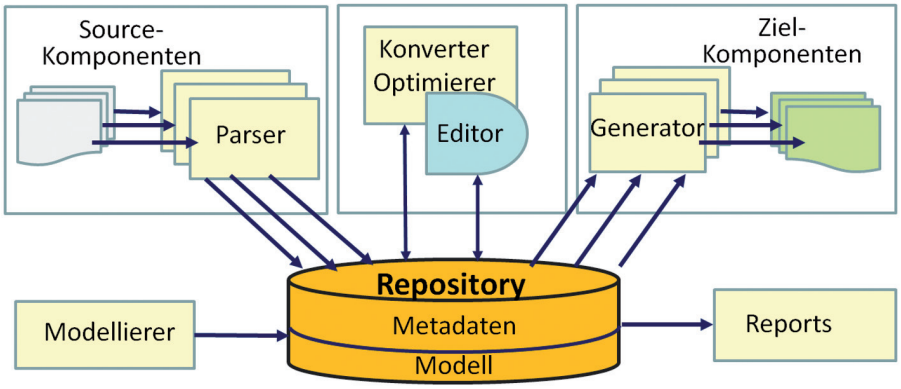

Abb. 2: Modellbasierte Architektur mit einem Repository.

zu migrieren, muss man sich bewusst sein, dass man bereits eine Menge an Metadaten zur Verfügung hat. Diese gilt es zu nutzen, damit daraus die neuen Programmkomponenten wenigstens teilweise generiert werden können. Speziell wenn die bestehenden Programme in Quellenform verfügbar sind, sollten diese mit einem Parser verarbeitet und als Metadaten gespeichert werden. Jede gewonnene Information spart später Aufwand und fördert die Qualität der Umsetzung.

Kanditaten zur Gewinnung von Metadaten sind:

- Programme
- Masken-Beschreibungsdaten
- Konfigurationsdaten

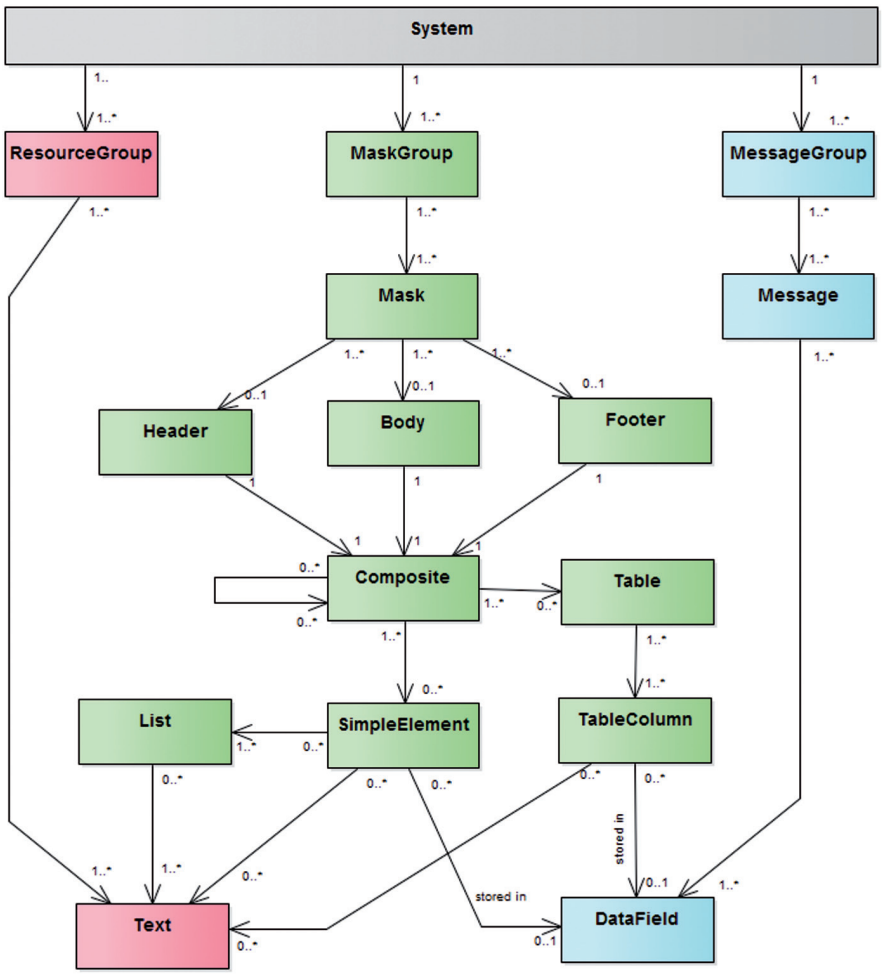

Abb. 3: Datenmodell Repository.

Bei Programmen besteht die Problematik, dass die Programmiersprache nicht immer in der gleichen konsequenten Art durch die Entwicklerinnen und Entwickler angewendet wurde. Das heißt, dass dieselbe Funktion oder derselbe Effekte auf viele verschiedene Arten durch Programmiercode erreicht werden kann. Dies gestaltet die Gewinnung von Metadaten etwas schwieriger, macht diese aber nicht unmöglich. Durch genauere Analyse bestehender Codesequenzen stellt man fest, welche Formen verwendet wurden und was davon nützlich ist. Bei Konfigurationsdaten, die bereits von einem Programm verwendet wurden, ist es einfach, Metadaten zu gewinnen, da dieser ja bereits maschinell lesbar sind.

In Abstimmung mit dem Kunden haben wir uns für die dritte Variante entschieden. Der Einsatz eines kommerziellen Meta-Repository-Systems (vgl. [Met]) hat sich aus Zeitgründen aufgedrängt, aber auch als Chance erwiesen, sich durch Flexibilität in der Modell-Anpassung empirisch an das zu ersetzende System heranzutasten.

Das Datenmodell

Mit Hilfe des Modell-Editors wurde ein erster Entwurf des Datenmodells erstellt und im Repository abgespeichert. Das Datenmodell wurde im Laufe des Projekts mehrmals geändert, weil sich während der Implementierung weitere Anforderungen an das Modell ergaben. Dies war recht einfach möglich, weil Modell und Metadaten-Bestand kongruent sind (ER-Metamodell) und das Modell einfach erweitert werden kann, selbst wenn bereits Metadaten gespeichert und Programme implementiert sind. Dieses Verfahren ist deutlich effizienter als das langwierige Entwickeln eines „vollständigen“ Modells, das am Ende den Anforderungen doch nicht genügt (**siehe Abbildung 3**).

Bei Betrachtung eines Ausschnitt aus dem Metamodell (**siehe Abbildung 4**) sieht man die zugehörigen Attribute und erhält einen Eindruck der konkreten Daten, die gespeichert sind. Trifft man während des Migrationsprozesses auf zusätzliche Informationen, die gewonnen oder generiert werden sollten, können die notwendigen Attribute hinzugefügt werden, ohne mit dem ganzen Prozess von vorne beginnen zu müssen.

Entwickeln der Parser

Einzelne dedizierte Parser (**siehe Kasten 1**) von Source-Komponenten (Programmteile, Datenbeschreibungen, Skripte, Daten in Repositories oder Bibliotheken) lesen je-

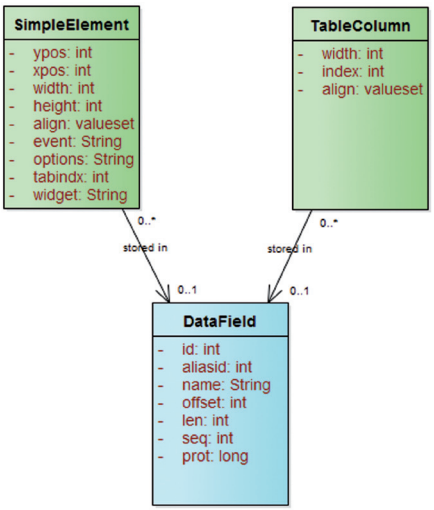

Abb. 4: Modellausschnitt mit Attributen.

weils einen Typ und hinterlegen die gefundenen Entitäten und Beziehungen mit ihren Attributen im Repository (auf Metadaten-Ebene). Welche Daten benötigt werden, lernen die Entwickler Schritt um Schritt von den generierten Daten verschiedener Komponenten (Generator, Editor, Optimierer, Reporter).

Für die diskutierte Lösung haben wir die Ressourcen-Dateien der MFC/VC++-Umgebung analysiert und dazu einfache Parser geschrieben. Dafür haben wir einfache Java-Programme verwendet, welche diese zeilenorientierten Spezifikationen eingelesen und als entsprechende Metadaten im Repository abgespeichert haben. Diesen Vorgang haben wir pro Sprache wiederholt und im Repository hinzugefügt. Während dieser Phase konnten wir das so oft wiederholen, bis wir die benötigte Information gewonnen hatten. Danach wurden diese Parser obsolet. Entsprechend darf der Aufwand für das Schreiben dieser Programmteile auch nicht als zu aufwändig angesehen werden. Parser sammeln oder ergänzen Informationen aus dem Modell, sodass der letzte Generator dann diese Daten verwenden kann, um ein Programm-Artefakt zu generieren (**siehe Abbildung 5**).

Ein Parser kann auch einfach ein Leseprogramm sein, das die aktuellen Daten aus dem Repository liest. Ein Parser kennt seinen Generator nicht und umgekehrt. Aber die gemeinsame Basis ist das im Speicher abgebildeten Modell. Der Parser schreibt in dieses Modell (als *Composite/Component* ausgeführt) und der Generator liest daraus und schreibt den Code (typisch als *Visitor*-Pattern ausgeführt). Damit sind Parser und Generatoren entkoppelt und man kann über die gleiche Modell-Instanz auch mehrere Parser oder Generatoren laufen lassen, wenn dies angebracht erscheint. Bei dieser Migration wurden vor allem auch mehrere Parsers verwendet, um Daten aus verschiedenen Quellen ins Repository zu schreiben und bestehende zu ergänzen (**siehe Tabelle 1**). Der Einlese-Vorgang kann beliebig oft wiederholt werden, solange die Daten im Repository nicht manuell verändert wurden. Neben der einen oder anderen Hilfsdatei wurden HTML-Templates für die Web-Benutzer-Schnittstelle generiert, die vom Java-Skript im Client-Browser interpretiert werden.

Das Prinzip ist immer gleich und es kann beim Aufruf des entsprechenden Paares bestimmt werden, was das Eingabe- und das Ausgabe-Format ist. Diese Definitionen wurden in einem *Ant*-Skript definiert, sodass sie einfach aufgerufen werden können.

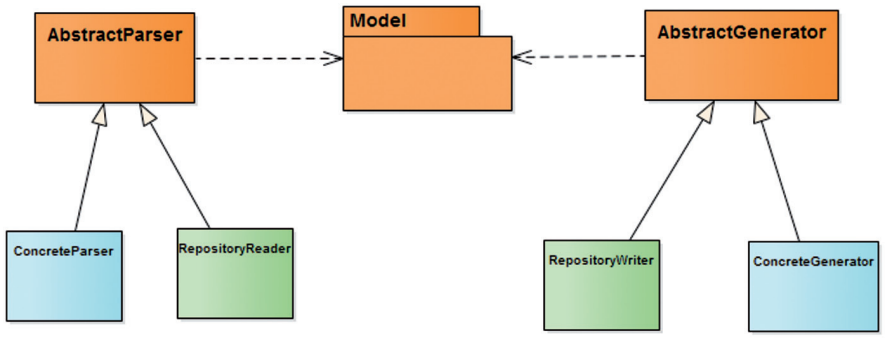

Abb. 5: Parser-Generator-Modell.

Optimierung und Transformation

Optimierungs- oder Transformationsfunktionen greifen direkt auf die Metadaten zu und hinterlegen die Ergebnisse im Repository. Das sind bestimmte Paare von Parser/Generatoren, die Daten aus dem Meta-Repository lesen, verändern und diese wieder dorthin zurück schreiben.

Durch eine Analyse der Informationen im Repository konnte vor allem die Zahl der Spezialitäten eruiert werden, die für die automatische, spätere Generierung in Betracht gezogen werden konnten. Man verlässt sich besser nicht darauf, dass 100 Prozent der verfügbaren Daten eingele-

Mancher Leser zuckt beim Wort Parser zusammen und stellt sich vielleicht einen Parser für eine umfangreichen Sprache vor. Mit den heute verfügbaren Werkzeugen wie ANTLR (vgl. [Par14]) und anderen ist es aber keine große Hexerei, einen Parser zu erstellen. Für einfache Sprachen wie C, Java, COBOL und sogar C++ gibt es oft Grammatik-Definitionen, die für das entsprechende Tool im Netz verfügbar sind (vgl. [Par14]). Bei komplexeren und älteren Sprachen wie PL/1 kann man sich hingegen die Zähne ausbeißen.

Oft muss man aber nicht vollständige Programme parsen, um die gewünschten Metadaten zu gewinnen. Häufig reicht es, nur Teile des Programmcodes zu analysieren. So wurden zum Beispiel nur für die Message-Strukturen, die in PL/1 definiert waren, ein Parser erstellt und daraus die Metainformationen gewonnen.

Kasten 1: Keine Angst vor Parsern!

Ressource (Name)	Ausgabeformat	Anzahl	Volumen	Parser (Name)	Größe (LOC)
⁎.rc, ⁎.rcs	Meta-Repository	874 Files	3.5 MB	RcsParser	622
Message-Repository	Meta-Repository	141 Strukturen 9.563 Felder	660 KB	MsgNavigation Parser	139
Meta-Repository	Modell	gesamthaft	-	RepositoryReader	463

Tabelle 1: Verwendete Parser.

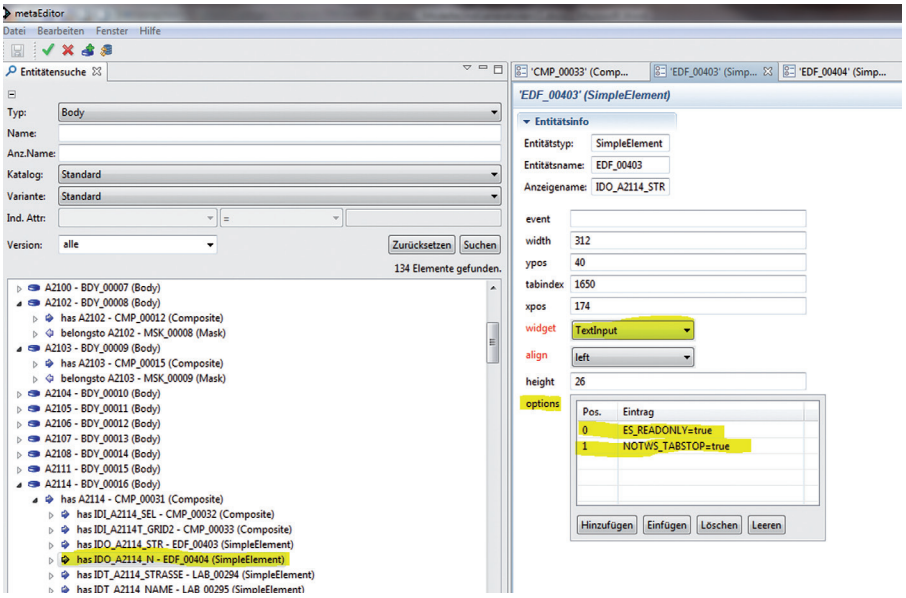

Abb. 6: Metadaten-Editor.

sen und 1:1 auf das neue System oder die neue Plattform konvertiert werden können. Gegen Überraschungen ist man am besten gefeit durch das iterative Verfahren, das heißt: Generieren, Testen, Modell optimieren etc. Im vorliegenden Fall konnten diese Daten entweder manuell oder halbautomatisch bereinigt und ergänzt werden. Da daraus eine Web-Applikation erstellt werden sollte, war dieser Schritt sehr wichtig.

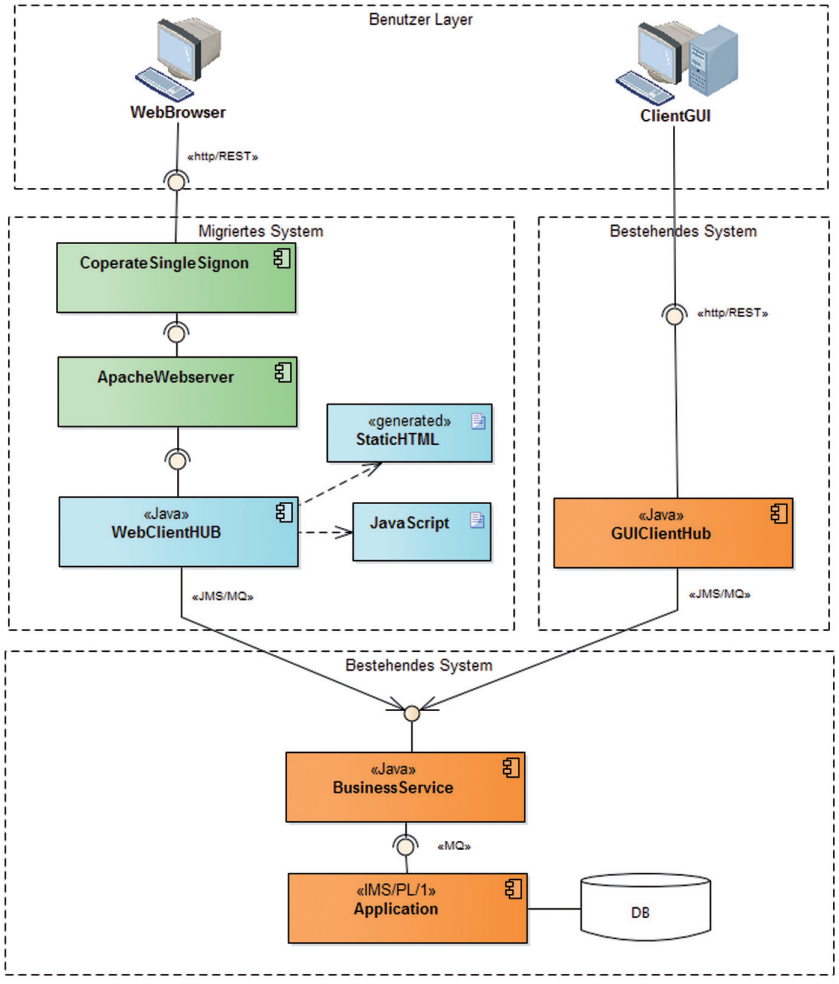

Abb. 7: Systemarchitektur mit alten und neuen Komponenten.

Erfassen von Zusatzinformationen

Nicht alle benötigten Daten stehen in maschinell lesbarer Form zur Verfügung. Sie werden daher mit dem Editor (für Metadaten) erfasst und im Repository gespeichert (siehe Abbildung 6).

Das in Tabelle 2 erwähnte Parser/Generator-Paar („Metasafe2FXML“) wurde zu Testzwecken erstellt, damit die importierten Masken mit dem „JavaFX Scene Builder“ dargestellt und möglicherweise von der Darstellung her verbessert werden konnten. Später wurde dies aber direkt mittels Meta-Editor im Repository vorgenommen und danach sofort mit dem HTML-Generator neu generiert, womit die Darstellung überprüft werden konnte. Diese Arbeitsweise hat sich als effizienter erwiesen. Auf die gleiche Art wurden spezielle Optionen für die Spezialbearbeitung von Feldern oder Tabellen hinzugefügt. Diese Optionen wurden dann bei der Generierung in Tags für das JS-Framework dazu verwendet, die nötigen Controller-Funktionen zu aktivieren.

Architektur der Web-Lösung

Da es sich im beschriebenen Projekt um ein bestehendes, vor über 30 Jahren entwickeltes System handelt, das bereits mehrere Migrationsschritte durchlebt hatte, mussten wir bei der Umsetzung darauf achten, dass für eine Zeitperiode sowohl die bestehende GUI-Lösung als auch die neue Web-Lösung parallel betrieben werden konnten. Der Fokus der Kunden liegt darauf, gewisse Kerngeschäfte (Analog-Telefonie) weiter effizient abwickeln zu können, bis dieser Bereich durch eine neue Technologie (IP-Telefonie) vollständig abgelöst wird. Der Termin, zu dem diese Applikation eliminiert wird, hängt von vielen Faktoren ab und kann nicht mit großer Genauigkeit vorhergesagt werden. Entsprechend versucht man, die Investitionen und das Ausfallsrisiko zu minimieren.

Die bestehende GUI-Lösung wurde vor 15 Jahren als Ablösung einer reinen Zeichen-Bildschirmlösung in C++ entwickelt und vor sechs Jahren bereits von einer CORBA-Middleware auf eine einfache http/REST-Kommunikation umgestellt. Dazu wurde eine Client-Hub-Komponente erstellt, die eine Umsetzung der eingehenden Anfragen von http/REST auf JMS konvertiert hatte.

Generierte Ressource	Generator	Größe (LOC)	Verwendung
HTML-Seiten	Metasafe2html	1.654	HTML-Generierung
FXML-Spezifikationen für Java FX	Metasafe2FXML	526	Darstellung in JavaFX
MetasafeWriter	RC2MetasafeWriter	1.005	Import in Metasafe
MessageOutputter	MessageOutputter	127	Export-Meldungen in JSON

Tabelle 2: Verwendete Generatoren.

Die bestehende GUI-Lösung hatte die Authentifizierung über das *Windows*-Login sichergestellt, was mit einem Web-Browser nicht direkt möglich ist. Damit dies mit der neuen Lösung möglich wurde, hat man ein „Single-Signon"-System verwendet, das bereits im Unternehmen etabliert war. Der nachgeschaltete Apache-Webserver auf den Server-Systemen wird lediglich dazu verwendet, die regelmäßige Lastverteilung auf vier Client-Hub-Services auf zwei Servern sicherzustellen (**siehe Abbildung 7**).
Die restlichen Services wie auch die Software auf dem Mainframe wurden weitgehend unverändert gelassen. Gewisse Funktionen, die früher in der Client-Software realisiert waren, wurden durch gewisse Regeln im Business-Service ergänzt. Die grundlegende Absicht war es, keine geschäftsrelevanten Regeln im Web-UI abzudecken, sondern nur im Business-Service sowie in der Kern-Applikation. Dies wurde aber immer so realisiert, dass sowohl die frühere GUI-Lösung als auch die neue Web-Lösung funktionierten.

Generierung der Zielkomponenten

Spezialisierte Generatoren erzeugen die Zielkomponenten aus den Metadaten, die im Repository gespeichert sind (**siehe Tabelle 2**).
Nach der Bereinigung der Daten und der Definition der in der Web-Applikation festgelegten Formate konnte der Generator erstellt werden. Für die Single-Page-Applikation wurden die HTML-Seiten mit den für das JavaScript-Framework wichtigen Attributen generiert. Da die Schriftarten und -größen der neuen Applikation etwas größer gewählt wurden, konnte durch eine nochmalige Überarbeitung der Metadaten ein Layout mit optisch besserem Erscheinungsbild erstellt werden. Auch für die weitere Pflege der Seiten (z. B. für zusätzliche Felder oder Masken) werden die Metadaten weiterverwendet.

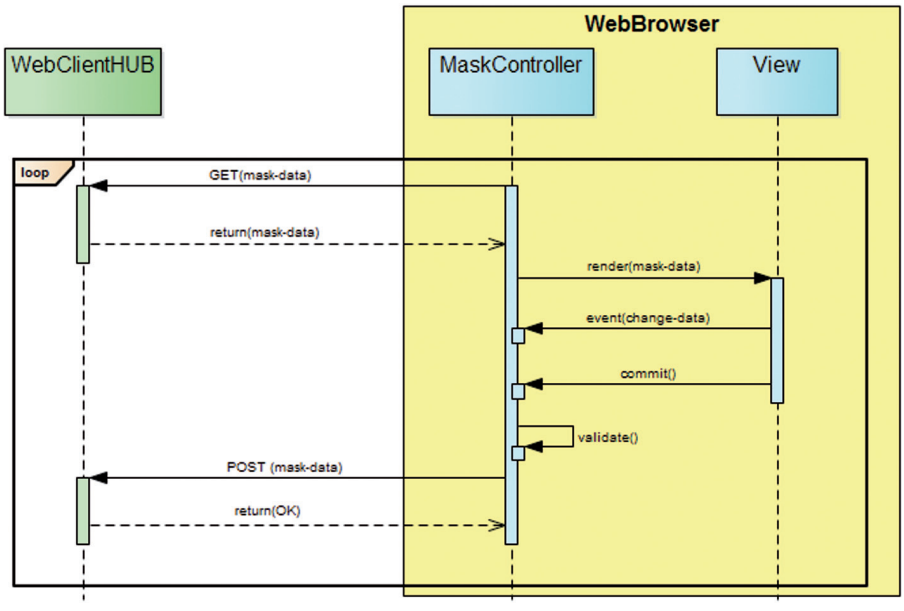

Abb. 8: Interaktion zwischen Browser und Web-Client-Hub.

Umsetzung als Web-Applikation

Im vorliegenden Fall lag der Fokus klar auf den Maskenfeldern und deren Metadaten, da die Applikationsführung vom Client-Hub gesteuert wird und diese Logik somit auf der Client-Seite minimal ist. Die Masken-Templates sollten aus den Metadaten generiert werden und dabei ihre statischen Eigenschaften (wie Position, Länge, Typ, Darstellung, Validierung usw.) bereits beim Generieren anpassen sowie eine dynamische Anbindung an die zur Laufzeit vom Serversystem gelieferten Daten haben.

Kommunikation zwischen Browser und Web-Client-Hub

Die Applikation wird – wie erwähnt – vom Web-Client-Hub gesteuert. Dieser Hub ist ein Web-Service, der den aktuellen Session-Zustand speichert und die eigentliche *Business*-Logik beinhaltet. Aus diesem Grund besteht die Aufgabe der Web-Applikation nur darin, die vom Client-Hub verlangte Maske mit den erhaltenen Daten darzustellen und Änderungen an diesen Daten zurück an den Server zu senden. Danach werden die neu aktuellen Daten wieder auf dieselbe Weise angefordert und verarbeitet (**siehe Abbildung 8**).
Diese Vorgehensweise hat verschiedene Vorteile. Zum einen ist die *Business*-Logik so vollständig im Server gekapselt, wodurch verhindert wird, dass ein übermütiger User einen Teil der Logik im Browser verändert (durch direkten Eingriff in den laufenden JavaScript-Code) und so inkonsistente Datenzustände erzeugt. Andererseits muss sich der Client auch nicht um applikationsspezifische Fehlerbehandlungen kümmern, da der Server einfach dieselbe Maske zurücksendet, solange die darin modifizierten Daten nicht valide sind. Wir haben im aktuellen Fall die Möglichkeit gegeben, den Gang zum Server mittels spezifischer Validierungsfunktionen zu verhindern und so die Arbeitsgeschwindigkeit zu erhöhen. Dieselben Funktionen sind jedoch auch im Server implementiert, falls die clientseitigen Validierungen manuell umgangen würden.

Das Framework: AngularJS

Die Auswahl an browserseitigen Java Script-MVC-Frameworks (so genannten „Single Page Application"-Frameworks) ist mitunter riesig. Diese Frameworks decken inzwischen so gut wie alle Use-Cases ab, die zur Entwicklung einer Web-Applikation notwendig sind. Um eine Entscheidung für das richtige Framework herbeizuführen, müssen die Anforderungen an die Applikation genau ana-

```
<tui-box tui-box-title="Kunde">
  <input type="text" ng-model="model.id9116.data"
     tui-mode="model.id9116.mode" tui-mode-default="0x6000" required>
  <!- Template abgekürzt... ->
</tui-box>
```

Listing 1: Deklarative Beschreibung einer Maske (HTML mit eigenen Elementen).

lysiert werden, sodass das Framework den Entwickler unterstützt und nicht der Entwickler für das Framework programmiert.

Durch die erwähnten Anforderungen lag unsere Entscheidung nahe, den Google-Sprössling „AngularJS“ (vgl. [Ang]) zu verwenden. Der Hauptgrund für diese Wahl war der deklarative Ansatz dieses Frameworks, der zum damaligen Zeitpunkt noch nicht sehr verbreitet war. Deklarativ bedeutet, dass die Funktionalität eines im Browser dargestellten Elements nicht an einem zentralen Punkt im JavaScript-Code angehängt wird (wie es zum Beispiel bei jQuery-Plug-Ins oft der Fall ist), sondern dass bereits das HTML-Template darauf hinweist, dass ein Element zur Laufzeit eine bestimmte Funktion erhält.

AngularJS bewerkstelligt dies, indem es HTML selbst um eigene Elemente und Attribute erweitert. Dies mag in der heutigen Zeit – nachdem WebComponents (vgl. [Web]) vom World Wide Web Consortium (W3C) standardisiert wurden und in den neuesten Versionen von Google Chrome und Mozilla Firefox inzwischen implementiert sind – alltäglich klingen, war jedoch zur Zeit des ersten Release ziemlich revolutionär.

Generierte Masken

Ein generiertes Masken-Template kann in etwa so aussehen, wie in dem Beispiel in **Listing 1** gezeigt. Das Listing zeigt die Instanz eines HTML-Elements (`tui-box`) mit einem dazugehörigen Titel-Attribut.

Dank der Funktionalität von AngularJS wird dieser Elementtyp nun Teil der vom Browser interpretierten HTML-Sprache. Dasselbe gilt für die weiteren Attribute, die zusätzliche Funktionalität für die Input-Elemente (`ng-model`, `tui-mode`, `tui-mode-default`) auf diesem *DOM*-Element (*Document Object Model*) deklarieren. Der Browser stellt mit diesem Code eine Box mit dem Titel „Kunde” dar, die ein Input-Feld beinhaltet (**siehe Abbildung 10**). Wird dieser Elementtyp an einem anderen Ort wiederverwendet, wird der Browser eine weitere Box darstellen.

Diese Element/Attribut-Definitionen werden *Directives* genannt. Da die Attribute das Element selbst beschreiben, ist es sehr einfach, die Übersicht über die Funktionalität einzelner Komponenten zu behalten und Fehler zu isolieren.

Im dem Codeausschnitt in **Listing 1** wurde durch den Einsatz der Directive `ng-model` bereits eine dynamische Verknüpfung zum Datenmodell aufgebaut, sodass alle Änderungen im Textfeld direkt im Datenmodell abgebildet werden und umgekehrt. Dies ermöglichte es uns, eine weitere Controller-

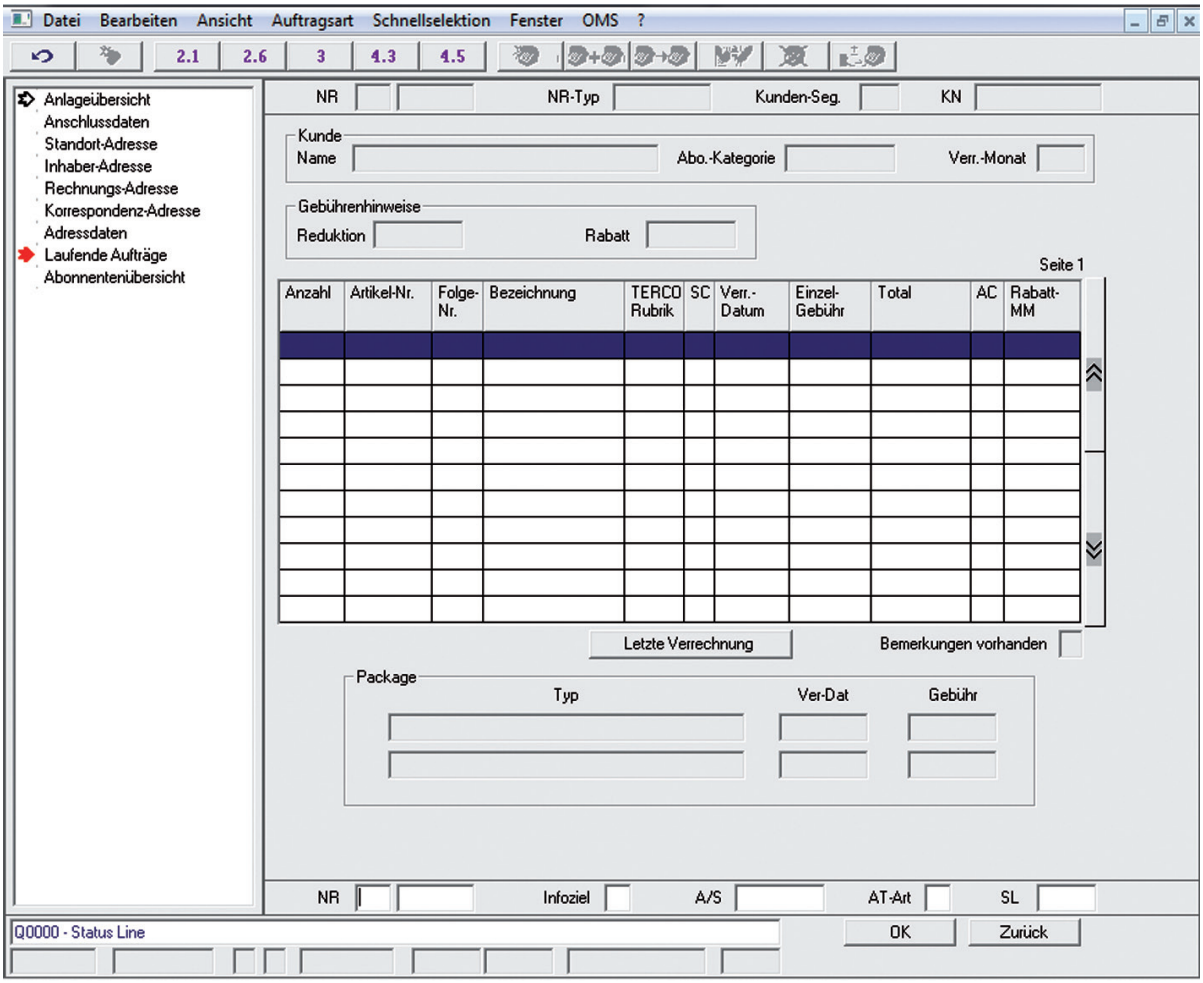

Abb. 9 Beispielsmaske bestehendes GUI.

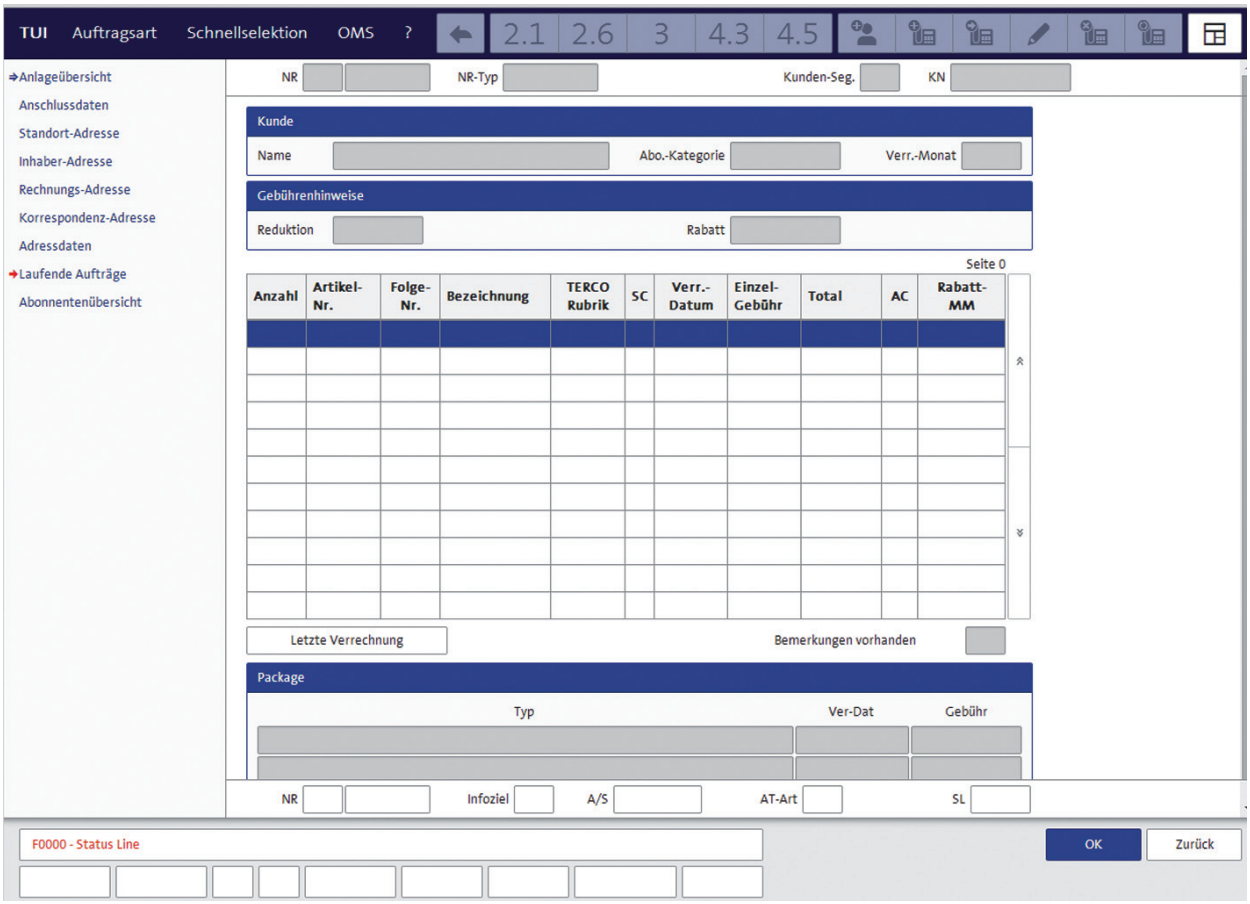

Abb. 10 Beispielsmaske Web UI.

Komponente einzusetzen, die spezifisch für jede Maske geschrieben wurde, falls Spezialfunktionen benötigt wurden (z. B. Leeren eines Datenfelds, wenn ein anderes gefüllt wird). Da Änderungen am Datenmodell direkt in den Views reflektiert wurden, mussten sich diese Controller nicht um darstellungstechnische Probleme oder DOM-Manipulationen kümmern und blieben deshalb sehr grundlegend.

Da die Funktionalität der Maskenfelder im Template definiert werden, das wiederum aus den Metadaten generiert wurde, musste ein Mapping dieser Metadaten-Parameter auf entsprechende *Directives* (Elemente und Attribute) erstellt werden. Dabei war die Herausforderung an den Generator, dass dieser entsprechend korrekte Elemente generiert und korrekt auf verschiedenste Metadaten-Konstellationen reagiert.

Realisierung des Masken-Designs

Die Positionierung und Größe der einzelnen Felder waren durch die Metadaten vorgegeben, die aus den alten Ressource-Dateien gewonnen wurden. Da jedoch einige Spezialfälle existierten, kamen wir nicht umhin, jede Maske einzeln auf ihre Funktion zu prüfen. Während dieses Vorgangs wurden gleichzeitig auch kosmetische Korrekturen angebracht, um das Gesamtbild angenehmer und moderner zu gestalten.

Abbildung 9 und **Abbildung 10** zeigen die Differenz zwischen der ursprünglichen Applikation und der Web-Applikation.

Auswertungen, Verwendungsnachweise und Dokumentationen

Durch den Einsatz des Meta-Repository-Systems haben wir noch weitere Vorteile erreicht. So können direkt aus dem Repository mit Hilfe der Query-Sprache und einem Auswertungstool, wie z. B. „BIRT“, Dokumentation und Verwendungsnachweise generiert werden. Zudem bildet das Meta-Repository auch für zukünftige Änderungen die Grundlage für die Neugenerierung von einzelnen Masken.

Fazit

Das Projekt wurde in acht Monaten (plus zwei Monate für die Machbarkeitsstudie) umgesetzt und produktiv eingesetzt. Der Gesamtaufwand betrug weniger als 50 Prozent der Schätzung des Aufwands, der bei einer rein manuellen Umsetzung benötigt worden wäre.

Mit einem Aufwand von 8,5 Stunden pro Maske (ohne GUI-Framework) gegenüber

Tätigkeit	Aufwand [h]	Aufwand/Maske [h]
Parser-/Generatorbau, Metamodell, Tool-Einführung	560	4,2h
GUI-Framework (JavaScript)	770	5,8h
Maskenvalidierung (Kosmetik, manuelle Ergänzungen)	570	4,3h
Client-Hub-Anpassung und Service-Funktionen, Installation, Integration, Test, Architektur, Projektleitung usw.	660	–

Tabelle 3: Aufwand in Bezug auf migrierte Artefakte.

einer manuellen Erstellung mit einem Aufwand von 15 bis 30 Stunden kann man feststellen, dass der eingeschlagene Weg bei der vorliegenden Anzahl an Masken richtig war. Der Schnittpunkt (Aufwand/Ertrag) läge bei 50 Masken, wenn man von einer durchschnittlichen Erstellungszeit von 15 Stunden pro Maske ausgeht (**siehe Tabelle 3**).

Insbesondere wäre bei der vorhandenen Menge und einer rein manuellen Umsetzung vermutlich nicht die gleiche Qualität erreicht worden, die sich bei der automatisierten Lösung ergab. Dank des Einsatzes von AngularJS konnte sowohl der Anforderung, die Lösung konform zu Internet-Explorer 8 zu machen, als auch die spätere Kompatibilität mit dem Browser des Internet-Explorers 11 zu erhalten, nachgekommen werden.

Der Aufwand für die Service-Funktionalität des WebClient-Hubs sowie für die grundlegende GUI-Funktionalität wäre sowohl bei einer manuellen als auch bei einer automatisierten Umsetzung angefallen.

Wenn auch vor und während der Umsetzung manchmal Mut und gute Ideen gefragt waren, haben wir doch einen Weg eingeschlagen, der sich für ähnliche Vorhaben erfolgreich wiederholen lässt. ■

Links

[Ang] AngularJS, siehe: `https://angularjs.org/`

[Met] Metasafe GmbH, Metasafe Repository, siehe: `www.metasafe-repository.com`

[Par14] T. Parr, ANTLR, 2014, siehe: `http://www.antlr.org/`

[Web] WebComponents.org, siehe: `http://webcomponents.org/`

Die Autoren

Dr. Reinhold Thurner
(Reinhold.Thurner@Metasafe-Repository.com)
befasst sich mit Modellierung, Repositories und Generatoren.

Andres Koch
(Andres.Koch@objeng.ch)
befasst sich seit mehr als 20 Jahren mit Software-Engineering, DSL, sowie mit Parserbau für Legacy-Migration und -Integration.

Remo Koch
(Remo.Koch@objeng.ch)
spezialisiert sich auf die Entwicklung von reinen, browserbasierten Web-Applikationen mit modernen JavaScript-Frameworks und Web-Components.