Agile | Blazor | ASP.NET | Angular | C# | React

S&S

1.2021

windows .developer

www.windowsdeveloper.de

Agile:

Entwicklung in Bewegung

Sonderdruck für fecher

fecher

Agile Produktentwicklung von Theorie bis Praxis

Von der agilen Insel zur beweglichen Organisation

Von VB6 direkt zur Browser-App

Zwei Schritte vor und keinen zurück

Eine Visual-Basic-Anwendung mag ausgereift sein und ihren Dienst noch funktional erfüllen, sie ist technisch und in Bezug auf die Benutzeroberfläche dennoch eine Altlast. Der Autor zeigt einen neuen, werkzeugbasierten Ansatz zur Anwendungsmodernisierung, der mit einem hohen Automatisierungsanteil eine effiziente Alternative zum Neuschreiben bietet. Im dargestellten Kundenprojekt wurden so sechzehn Anwendungen mit zwei Millionen Lines of Code von Visual Basic (VB) direkt in eine moderne Browser-App transformiert.

von Andreas Glomm

Wer heute funktional ausgereifte Anwendungen am Markt anbietet, hat mit der Entwicklung meist zu einer Zeit begonnen, als Werkzeuge wie der Gupta Team Developer oder Visual Basic (VB5 oder VB6) als fortschrittliche Technologien galten. Heute bereiten diese Plattformen jedoch zunehmend Probleme: Die Herstellerunterstützung hat nachgelassen oder ist längst aufgekündigt, Softwareentwickler dafür lassen sich kaum noch finden und von einem Einsatz unter Windows 10, im Browser oder in der Cloud können die Anwender oft nur träumen.

Genau das waren auch die Problempunkte, aufgrund derer ein nordamerikanischer Anbieter von Software für den öffentlichen Dienst das Unternehmen des Autors kontaktiert hat: Insgesamt 16 VB6-Anwendungen waren bei über 400 Kunden erfolgreich im Einsatz. Darunter waren Programme für die Aufstellung des Haushaltsplans, die Organisation der Fahrzeugzulassung und die Abrechnung der Grundsteuer zu finden. Die Einschränkungen der alten Plattform hatten sich jedoch zu einem so massiven Hemmschuh für die Nutzerakzeptanz und den technischen Support entwickelt, dass der Vertrieb an Neukunden nahezu zum Erliegen gekommen war. Und bei einem Umfang von zwei Millionen Lines of Code war an Neuschreiben nicht zu denken.

Zukunftschance statt Altlast

Ein Befreiungsschlag aus dieser misslichen Situation konnte nur gelingen, indem zwei Schritte auf einmal unternommen wurden: Die Systemgrundlage musste vom altem VB6 auf aktuelles .NET umgestellt werden. Zugleich musste die Benutzeroberfläche von Windows in eine flexible Browser-App verlegt werden, die auch die Ausführung auf Nicht-Windows-Plattformen und mobilen Geräten ermöglichte. Zusätzlich galt es, in dieses ohnehin schon massive Migrationsprojekt eine externe Webagentur einzubinden, die ein grundlegendes Redesign der Anwendungsoberfläche erstellen sollte. Die vielfältigen Herausforderungen sollten durch den Einsatz spezialisierter Migrationswerkzeuge in einem klar strukturierten Phasenmodell angegangen werden.

Phase 1 – von VB6 nach .NET

Zunächst galt es, den in die Jahre gekommenen VB6-Code nach .NET zu portieren. Für diesen Projektschritt konnte auf den vbPORTER zurückgegriffen werden. Das ist ein Migrationswerkzeug, das sonst auch für isolierte Portierungen nach WinForms eingesetzt wird (**Abb. 1**). Vordergründig ging es dabei um eine Sprachtransformation, bei der die verwendeten Syntaxelemente von VB6 in ihre C#-Entsprechung umgesetzt werden. Das reichte vom Ersetzen der Operatoren *AND* und *OR* durch *&* und | bis zu komplexen Anpassungen wie beispielsweise

der Deklaration von Variablen durch Instanziierung statt durch *DIM* (Dimension). Eine Übersetzung nach VB.NET wird vom Werkzeug ebenfalls unterstützt, kam aber in diesem Projekt nicht zum Einsatz.

Konzepte, die unter .NET nicht mehr wie von VB6 gewohnt vorhanden sind, stellt eine Bibliothek namens fecherFoundation zur Verfügung. In ihr finden sich Komponenten mit entsprechender Signatur und Funktionsumfang, auf die der vbPORTER zurückgreifen kann. So implementiert die Library beispielsweise das beliebte Recordset aus RDO (Remote Data Objects):

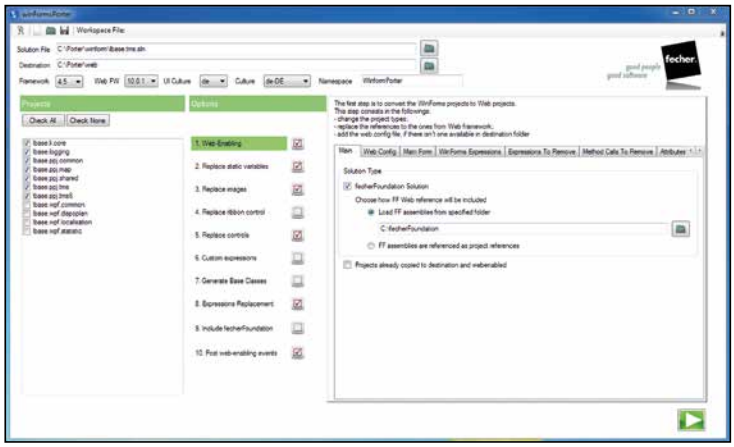

Abb. 1: Portierung von VB6 nach WinForms mit dem vbPORTER

```
public class FCRecordset : DataTable, IFCDataTable
```

Auf dieser Grundlage muss der vbPORTER nicht den gesamten Source Code umbauen, sondern kann einfach die Vorkommen von *Recordset* durch *FCRecordset* ersetzen. Ähnliches gilt auch für die visuellen Komponenten aus VB6, die ebenfalls auf die grundlegend anderen .NET-Konzepte umgestellt werden müssen. Auch hier helfen entsprechende Klassen aus der fecherFoundation, wie etwa *FCButton* (basierend auf *System.Windows.Forms.Button*), *FCForm* (basierend auf *System.Windows.Forms.Form*) und viele mehr. Die VB6-Controls wurden also nicht direkt in ihre .NET-Äquivalente überführt, sondern in fecherFoundation-Implementierungen. Diese setzen ihrerseits auf den Standard-.NET-Controls auf. Dabei macht es keinen Unterschied, ob es sich um simple Push-Buttons und Textboxen oder komplexe Third Party Controls handelt. Auf diese Weise bleibt der erzeugte Code übersichtlich und der volle Funktionalitätsumfang der VB6-Controls auch in der .NET-Welt erhalten. Die größte Herausforderung für den vbPORTER besteht darin, den Kontext jeder Operation festzustellen: Nur wenn im VB6-Quelltext korrekt ermittelt werden kann, aus welchem Modul eine Variable stammt, lässt sich in .NET die korrekte Qualifizierung vornehmen und die richtige Klasse ansteuern. Dazu arbeitet das Werkzeug in einem mehrstufigen Prozess, der zunächst den VB6 Source Code einliest, parst und dann einen semantischen Baum daraus baut. Anschließend führt er mit diesen Informationen Form für Form und Datei für Datei die eigentliche Transformation durch.

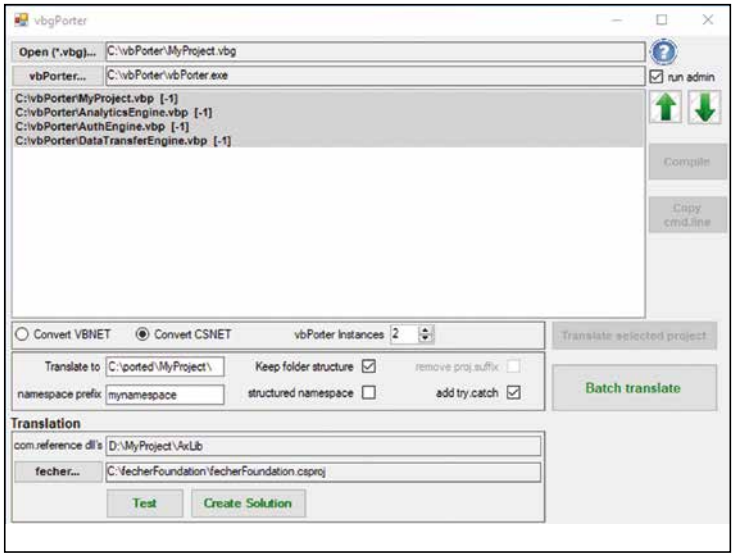

Abb. 2: Form „Account Maintenance“ in VB6

Als Nächstes erfordert die *ref*-Problematik in C# einiges an Aufmerksamkeit bei der Portierung. Während in VB6 jeder Methodenaufruf grundsätzlich ein Call by Reference ist, in den man dennoch Variablen, Properties oder sogar Werte direkt einsetzen kann, dürfen in C# lediglich Objekte als *ref*-Parameter eingesetzt werden. Eine direkte Umsetzung des vorhandenen VB6-Codes würde also zwingend zu Kompilierfehlern führen. Um das zu lösen, kann man mit Überladung arbeiten. Das bedeutet, man bietet für jede Methode mehrere Parameteraufrufe an, die letztendlich zur selben Implementierung führen. Diese werden vom vbPORTER bereits generiert, die eigentliche Auswahl des Aufrufs erfolgt aber im Nachgang mit Roslyn. Dieses Compiler-API von Microsoft stellt den kompilierten Code in einem Syntax-Tree dar. Das erlaubt es – auch in Abhängigkeit von Datentypen oder der Herkunft eines Methodenaufrufs –, variable Transformationsregeln darauf anzuwenden. Entsprechende Transformationsregeln in Roslyn dienen also als nachgelagerte, zweite Portierungsinstanz auf dem Weg zu fehlerfrei kompilierbarem .NET-Code. Ein standardisiertes Kontingent an Regeln aus früheren Projekten steht bereits zur Verfügung und wird im Lauf jedes Projekts immer weiter angepasst und verfeinert – theoretisch bis zur vollständigen Kompilierbarkeit. Verbleiben am Ende nur noch einige wenige, isolierte Fehlerstellen, für die sich das Schreiben neuer Regeln nicht lohnt, können diese natürlich auch ganz undogmatisch von Hand behoben werden. In diesem Projekt wurde allerdings bewusst darauf verzichtet, manuellen Aufwand zu investieren, um die eingesetzten

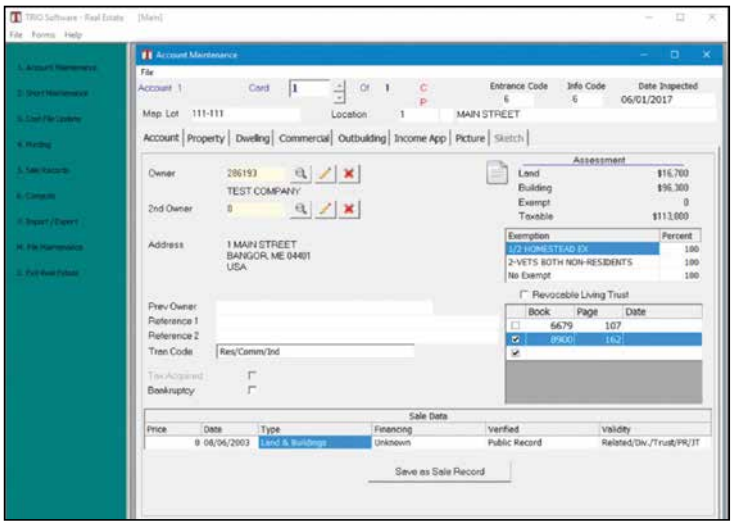

Abb. 3: Form „Account Maintenance“ nach dem Web-Enabling der portierten Anwendung

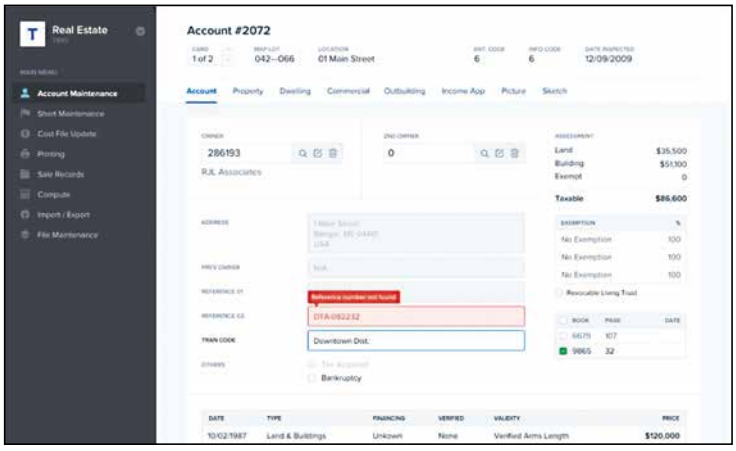

Abb. 4: Web-Enabling mit dem winformPORTER

Drittanbieterkomponenten vollständig nach WinForms zu portieren, da das nicht die endgültige Zielplattform darstellte. Der betreffende Code konnte deshalb einfach auskommentiert und die endgültige Implementierung auf später verschoben werden.

Phase 2 – Redesign

Der Kunde hatte sich entschieden, die Anwendung nicht einfach identisch mit der vorhandenen Benutzeroberfläche (**Abb. 2**) nach WinForms und dann ins Web zu übertragen. Vielmehr wollte er die Gelegenheit nutzen, seinen Anwendern mit dem doppelten Plattformwechsel auch gleich ein neues UI-Design zu präsentieren (**Abb. 3**). Mit dessen Entwicklung war ein externer Designer beauftragt, der noch während der laufenden VB-Portierungsphase begann, sich exemplarische Bildschirmmasken vorzunehmen, anhand derer er eine neue, modernere und browserkonforme Bediensprache entwickelte. Das Resultat war neben Photoshop-Designs und HTML-Mock-ups ein ausführlicher Styleguide, der alle Aspekte der Benutzeroberfläche definierte: „Welche Controls sollen eingesetzt und wie müssen sie gestylt werden?“, „Welche Abstände sind zwischen den einzelnen Elementen einzuhalten?“, „Welche Regeln gelten für die Farbgebung?“, „Wo sind welche Hintergrundbilder und Icons zu verwenden?“ waren nur einige der Designfragen.

Aus diesem detaillierten Styleguide entwickelte das Migrationsteam zunächst ein entsprechendes Theming für die Webplattform. Es musste die Regeln aber auch später jederzeit vor Augen haben: Das reichte von der Anordnung von Controls in den einzelnen Forms über das Verhalten von Docking und Anchoring bis hin zu Veränderungen an Masken, die erst zur Laufzeit vorgenommen werden, etwa beim Aufklappen von optionalen Eingabeelementen. Nach dem Einrichten des Themes hatte das Migrationsteam also noch während der gesamten folgenden Phase bis zum erfolgreichen Projektabschluss mit dem Redesign zu tun.

Phase 3 – Web-Enabling

Als Laufzeitumgebung für den finalen Schritt, das Web-Enabling der nun vorliegenden WinForms-Anwendung, kam vereinbarungsgemäß das Framework Wisej zum Einsatz. Es baut auf der Open-Source-JavaScript-Bibliothek Qooxdoo auf [1], die vom deutschen Internetprovider 1&1 ursprünglich für seine Webmailer entwickelt und später auch für eine Reihe weiterer kommerzieller Webanwendungen genutzt wurde [2]. Wisej bindet die HTML5-Widgets von Qooxdoo ein, die in Aussehen und Verhalten den jeweiligen WinForms Controls entsprechen. Darüber hinaus bietet das Framework jedoch auch hilfreiche eigene Widgets, etwa ein Grid Control mit .NET-konformer Anbindung an beliebige Datenprovider. Wisej stellt ein durchgängiges Distributed Object Management (DOM) zur Verfügung, das weitgehend dem Objektmodell der WinForms-Programmierung unter .NET entspricht. Dazu synchronisieren sich Browserclient und Serverkomponenten in Echtzeit über einfache, optimierte JSON-Pakete. Serverseitig sorgt Wisej für den Anschluss an das .NET DOM und garantiert so die lückenlose Anbindung an Geschäftslogik und Datenbankdienste. Diese interne Architektur ist jedoch vollständig innerhalb des Frameworks gekapselt und weder für Entwickler noch für Anwender sichtbar. Da das Framework den gleichen Funktionsumfang und eine identische Syntax wie WinForms bietet, würde es theoretisch genügen, im WinForms-Projekt für die dort enthaltenen Referenzen vom Typ *System.Windows.Forms.** jeweils die entsprechenden *Wisej.**-Referenzen einzusetzen. Zusätzlich müssen dann in den Forms die Usings ausgewechselt werden: Überall, wo die *System.Windows.Forms*-Bibliotheken referenziert werden, steht nun das *Wisej.Web*-Äquivalent.

Die einzelnen Anwendungen, die bisher in 16 separate Executables mit eigenen Ablaufumgebungen geteilt waren, sollten in eine einzige durchgehende Web-App zusammengefasst werden.

Um dieses lästige und fehlerträchtige Unterfangen an unzähligen Codestellen durchzuführen, kommt wiederum ein Werkzeug aus vorangegangenen Projekten zum Einsatz, der winformPORTER (**Abb. 4**). Er analysiert den Quellcode und übernimmt regelbasiert die Umsetzung von Referenzen, Usings sowie aller notwendigen Codeanpassungen, die damit im Zusammenhang stehen.

Entscheidend ist, dass das Werkzeug nicht rein zum Ersetzen verwendet wird, sondern den Source-Code-Aufbau analysiert und auf Basis des so extrahierten Code-DOMs arbeitet. So kann es auch ein Problem adressieren, das sich aus dem Paradigmenwechsel zum Web-Deployment ergibt: Statt wie bisher dezentral auf einzelnen Desktops, läuft die Anwendung nun zentral auf dem Server ab. Dadurch teilen sich alle Benutzer eine einzige Laufzeitumgebung mit dem gesamten Kontext. Man kann sich also nicht mehr darauf verlassen, Zustände in statischen Variablen abzuspeichern und später dort wiederzufinden. Deshalb verschiebt der winformPORTER statische Variablen gebündelt in eine neue Klasse *StaticVariables* und schaltet einen Session Swapper vor, der jeder einzelnen Benutzer-Session ihren individuellen Satz an statischen Variablen über die Property *public static StaticVariables Statics* zur Verfügung stellt. Spezielle Lösungskomponenten müssen einmalig individuell für das Einzelprojekt entwickelt werden. Der anschließende Einbau in den Code kann dann mit entsprechenden Umsetzungsregeln ebenfalls automatisch erfolgen.

In diesem Projekt kam zu dem grundlegenden Problem der statischen Variablen eine weitere Komplexitätsebene hinzu: Die einzelnen Anwendungen, die bislang in 16 separate Executables mit eigenen Ablaufumgebungen geteilt waren, sollten in eine einzige durchgehende Webanwendung zusammengefasst werden. Um Probleme durch identisch benannte statische Variablen zu vermeiden, musste also ein zusätzlicher Mechanismus entwickelt werden. Dieser stellt für jede Form fest, aus welcher ursprünglichen Applikation sie stammt und macht die Instanziierung der statischen Variablen im Session Swapper zusätzlich davon abhängig. Diese Sonderersetzungsregel konnte als projektspezifische Anpassung im winformPORTER konfiguriert werden und lief dann automatisiert über den gesamten Code. Ähnlich wurde die Bedienung per Tabbed MDI (Multiple Document Interface) eingeführt: Einmal entwickelt und im Regelwerk verankert, wurde das Bedienelement automatisch in alle Forms eingefügt. Es sorgt jetzt dafür, dass – im Gegensatz zu einzelnen Browserfenstern – alle Masken eines Benutzers innerhalb einer einzigen Web-Session laufen und dort mit gemeinsamen Daten arbeiten können.

Nicht alles geht im Web

Allerdings ergaben sich auch grundsätzliche Probleme durch den Wechsel ins Web, deren Lösungen sich nicht auf Knopfdruck automatisieren ließen. Zum einen sind hier Drittanbieterkomponenten zu nennen, die eine eigene Bildschirmausgabe oder Druckausgabe vorsahen – oder beides. So kam in diesem Projekt der Reportgenerator ActiveReports zum Einsatz. Die bislang genutzte Komponente ActiveX ließ sich unter .NET nicht nutzen, weshalb zunächst auf ActiveReports for .NET gewechselt wurde. In diesem Konzept lief die Komponente auf dem Server und erzeugte einen PDF-Report, der nach Fertigstellung an den Webclient übertragen und dort angezeigt wurde. So ließen sich die bislang genutzten Reportdefinitionen weiter einsetzen. Allerdings mussten die Anwender bei dieser Vorgehensweise immer auf die Erzeugung des gesamten Reports warten, bevor sie auch nur die erste Seite zu sehen bekamen. Bei Reports mit vielen hundert Seiten, wie sie in diesem Projekt regelmäßig vorkamen, erwies sich das als nicht praktikabel. Die Lösung brachte die Einbindung der Webkomponente von ActiveReports direkt in die Browser-App. Die bislang noch nicht vorhandene Integration in das Wisej-Framework war mit einem gewissen manuellen Entwicklungsaufwand verbunden. Dafür ließ sich die nachfolgende Ersetzung im Code schnell mit einer Transformationsregel für den winformPORTER erledigen. Da die Reports jetzt auf dem Webclient gerendert werden, ist im Ergebnis zumindest die erste Seite sofort sichtbar. Ebenso stehen zusätzliche Optionen etwa zum lokalen Export als Word- oder CSV-Datei zur Verfügung.

Nicht alle Druckausgaben der alten Anwendungen liefen jedoch über den Reportgenerator: In den meisten Installationen des Kunden kommen Quittungsdrucker zum Einsatz, die lokal angeschlossen sind und deren USB-Schnittstelle unter Umgehung von Windows-Treibern direkt angesteuert werden muss. Dieses Konzept ins Web zu übertragen, hat dem Team einiges Kopfzerbrechen bereitet: Da die Ansteuerung der Hardware zwingend lokal erfolgen muss, führte kein Weg daran vorbei, entgegen allen Webkonzepten doch Software auf dem Client zu installieren – nämlich einen Dienst, der die Kommunikation mit dem Drucker einerseits und dem Webserver andererseits ermöglichen sollte. Realisiert wurde dieser als

Windows-Tray-Anwendung, die eine WebSockets-Verbindung zum Server aufbaut und sich über ihre MAC-Adresse ausweist. Der Server ordnet diese Verbindung dann der Browser-Session zu, die die identische Adresse übermittelt. Die Kommunikation über WebSockets hat dabei den Vorteil, dass der Server den Dienst benachrichtigen kann sobald ein Druckauftrag vorliegt, ohne dass ständige Client-Requests nötig werden. Im Endeffekt löst also der Benutzer am Webclient den Druck aus, der Server übergibt die Daten per WebSockets an die zugehörige Windows-Tray-Anwendung und diese gibt den Auftrag dann lokal an den Drucker aus.

Jetzt wird's stabil

Nachdem die Anwendung sich kompilieren ließ, alle Architekturfragen gelöst waren und das erfolgte Redesign sämtlicher Forms vom externen Designer abgenommen war, konnte die Stabilisierungsphase beginnen. In dieser den eigentlichen Unit-Tests vorgeschalteten Projektphase starten die Entwickler die Anwendung und probieren zunächst noch unsystematisch die offensichtlichsten Funktionen aus. Entscheidend in dieser Phase ist es, für jeden gefundenen Fehler eine systematische Behebung zu finden, entweder durch einfaches Suchen und Ersetzen im Code oder – besser – wieder über Roslyn. Auch in der nächsten Phase der systematischen Klicktests mittels Screenvideos, die der Kunde zu Beginn des Projekts von seinen wichtigsten Use Cases zur Verfügung gestellt hat, wird dieser Ansatz beibehalten. Erst wenn alle so gefundenen Fehler systematisch behoben sind, geht die Software für die finalen Anwendungstests zum Kunden, der alle Anwendungsfälle idealerweise mit Echtdaten erneut durchspielt. Selbst dieser grundlegend manuelle Prozess wird noch von einem Werkzeug begleitet: Das Code-Coverage-Tool dotCover zeichnet bei jedem Testdurchlauf im Hintergrund auf, welcher Code ausgeführt wurde. Die Aufzeichnungen der einzelnen Test-Sessions lassen sich zusammenführen, um ein umfassendes Bild davon zu haben, welcher Anteil des Codes bereits getestet ist und wo vielleicht noch unentdeckte Fehler stecken können.

Dank des etablierten Vorgehensmodells und des systematischen Werkzeugeinsatzes konnte das Projekt nach einer Laufzeit von 18 Monaten planmäßig zum Abschluss gebracht werden. Verglichen mit einer Neuentwicklung hat sich der Kunde viel Zeit, Geld und Risiko erspart. Trotzdem ist das Ergebnis alles andere als ein Kompromiss: Der Roll-out an die ersten Installationen hat bereits begonnen und die Resonanz ist ausgesprochen positiv. Den Nutzern steht nun eine moderne Anwendung mit attraktivem UI-Design zur Verfügung, die sie gern und effizient verwenden. Aufgrund der neuen Webunterstützung ist die Applikation nun auch von überall nutzbar. Man benötigt lediglich einen Browser mit Internetzugang.

Als Head of Application Modernization ist **Andreas Glomm** verantwortlich für die Modernisierungsprojekte bei der fecher GmbH (www.fecher.eu). In seinen 20 Jahren im Team hat der Diplom-Informatiker über 200 VB6- und Gupta-Portierungen sowie Web-Enabling-Projekte für Kunden durchgeführt.

Links & Literatur

[1] https://wisej.com/docs/2.2/html/Welcome.htm

[2] https://qooxdoo.org/about.html