Softwareentwicklung mit FlowProtocol 2

Wolfgang Maier

8. Januar 2025

Inhaltsverzeichnis

Vorwort

Die vorliegende Anleitung beschreibt Möglichkeiten, die Anwendung *FlowProtocol 2* in der professionellen Softwareentwicklung einzusetzen und damit die besonderen Herausforderungen zu meistern, die sich in diesem Bereich stellen. An vielen kleinen und auch auch größeren Beispielen wird gezeigt, wie man Skripte in *FlowProtocol 2* erstellt und welche Bandbreite an Hilfsmitteln sich damit bereitstellen lassen. Man lernt, Skripte als Ablageform für Wissen und Konventionen innerhalb eines Entwicklungsteams zu sehen und die eigenen Fähigkeiten in dieser Form weiterzugeben.

FlowProtocol 2 ist die komplett überarbeitete Nachfolgeversion von FlowProtocol, das im Winter 2021/22 entwickelt wurde. Es handelt sich um eine kleine Anwendung. die über einen Browser bedient wird, und auf der Skripte ausgeführt werden können. Die Skripte bestehen aus einfachen Textdateien, die in einem beliebigen Editor erstellt werden können. Die Verwaltung der Skripte erfolgt in einer lokalen Verzeichnisstruktur. mit der eine organisatorische oder aufgabenbezogene Gliederung abgebildet werden kann.

Bei der Ausführung eines Skriptes werden Informationen über Eingabefelder abgefragt und über die Anweisungen im Skript verarbeitet. Die Ausgabe erfolgt ausschließlich als Ergebnisdokument im Browser, aus dem dann z.B. Textpassagen über die Zwischenablage weiterverwendet werden können. Alle eingegebenen Daten werden ausschließlich als Parameter in der URL verwaltet, es gibt keine angebundene Datenbank, keine Benutzerverwaltung und es werden durch die Anwendung keine Dateien erstellt und geändert.

Der Anwendungsfall, für den FlowProtocol ursprünglich entwickelt wurde, ist die Erstellung von Checklisten, die durch Interaktion mit dem Benutzer auf einen individuellen Fall zugeschnitten werden, und so beliebig ins Detail gehen können, ohne unnötige Einträge aufzulisten. Daran werden sich auch die ersten Beispiele in dieser Anleitung orientieren. Auf dieser Basis entstanden zahlreiche Skripte für den Product Owner, die für die verschiedenen Standardentwicklungen alle benötigten Einstellungen und Informationen abfragten, die durch das Designsystem und die damit assoziierten Framework-Komponenten verfügbar waren. Der Vorteil für den Product Owner bestand darin, dass ihm aufgrund der in den Skripten hinterlegten Abhängigkeiten immer nur die Optionen angeboten wurden, die für den jeweiligen Fall sinnvoll waren, und er so auch an alle Entscheidungen herangeführt wurde, die an der jeweiligen Stelle getroffen werden mussten. Das Ergebnisdokument bestand in diesem Fall aus einem sehr umfangreichen Userstory-Entwurf, in dem schon alle abgefragten Informationen und Entscheidungen eingearbeitet waren, und der mehr oder weniger nur noch um einige Benennungen und Aufzählungen angereichert werden musste. Schon bald wurde FlowProtocol um die Möglichkeit erweitert, auch Texteingaben abzufragen und diese in das Ergebnisdokument einzuarbeiten.

Die Systematik, die auf Seiten der Story-Formulierungen möglich ist, ist hauptsächlich begründet durch die Komponenten des über viele Jahre aufgebauten eigenen Frameworks und deren Möglichkeiten und die Erfahrung des Teams, also das über zahlreiche Wiederholungen aufgebaute Wissen, wie man wiederkehrende Muster umsetzt, und was dabei zu beachten ist. Entsprechend lag es nahe, ähnliche Unterstützungswerkzeuge auch auf Entwicklungsseite zu schaffen, die den Programmierer dazu anleiten, die richtigen Klassen zu verwenden und am Ende idealerweise sogar fertigen Programmcode erstellen. Zu diesem Zweck wurde die Formatierungsform als Code geschaffen, bei der man einen Block per Schaltfläche in die Zwischenablage zu kopieren kann, und der CamelCase-Befehl, mit dem Namen für Felder, Funktionen oder Variablen erzeugen kann. Die Verwendung

interaktiver Anleitungen mit Code-Generierung steigert nicht nur die Effizienz bei der Umsetzung von Standardaufgaben, sie hilft auch bei der Sicherstellung von Einheitlichkeit und ist damit eine wichtige Säule in der Qualitätssicherung.

Auch wenn *FlowProtocol 2* selbst nicht direkt mit anderen Anwendungen interagiert, so kann in einem Skript praktisch jede URL mit beliebigen Parametern aufgebaut werden, so dass sowohl bei der Ausführung, als auch aus dem Ergebnisdokument der parametrisierte Aufruf anderer Web-, oder Intranet-Anwendungen möglich ist. Allen voran können Skripte auf diese Weise andere Skripte aufrufen, aber auch viele der im Entwicklungsumfeld eingesetzten Anwendungen für die Verwaltung von Vorgängen oder als Wiki bieten gute Steuerungsmöglichkeiten. Schon allein mit der Übergabe eines Suchbegriffs kommt man schon recht weit, und über einen MailTo-Link lassen sich sogar vollständig ausformulierte E-Mails vorbereiten. Die Codierung der Parameter für die URL lässt sich dabei sehr einfach mit dem UrlEncode-Befehl umsetzen.

Inzwischen umfasst der Befehlssatz von *FlowProtocol 2* alle notwendigen Befehle für den Aufbau von Programmen, wie If-Abfragen, For-Schleifen, sowie die Definition von Funktionen, die auch rekursiv aufgerufen werden können. Zusammen mit den verschiedenen Befehlen für das Rechnen mit Zahlen und Datum-Uhrzeit-Werten und zur Manipulation von Zeichenketten lassen sich mit sehr geringem Aufwand kleinere und größere nützliche Hilfsanwendungen schreiben, die unmittelbar auf jedem Arbeitsplatz verfügbar sind.

Der Nutzen von *FlowProtocol 2* innerhalb eines Teams oder einer Einrichtung erhöht sich in besonderem Maße dadurch, dass die Skripte von einer größeren Zahl an Kollegen gepflegt und erweitert werden, und dass der bestehenden Erfahrungsschatz auf diese Weise permanent durch neues Wissen erweitert wird. In einfacher Form lässt sich dies sogar hinbekommen, ohne dass dafür Programmierkenntnisse vorausgesetzt werden, indem ein Skript die für seine eigene Erweiterung notwendigen Informationen abfragt, und daraus den resultierenden Programmcode samt Einbauanleitung selbst erzeugt. Auch diese Methode wird an einem Beispiel beschrieben.

Bis dahin werden aber erst einmal die grundlegenden Befehle und Funktionsweisen beschrieben, um einen einfachen Einstieg in *FlowProtocol 2* zu ermöglichen. Ich freue mich über jeden, der mit Hilfe dieser kleinen Anleitung auf Entdeckungstour geht, und wünsche viel Spaß und einen hoffentlich nutzbringenden Einsatz.

Zuletzt möchte ich noch Danke sagen, an alle Kollegen bei easySoft, die *FlowProtocol 2* bei der täglichen Arbeit eingesetzt, und durch ihr Feedback zur permanenten Verbesserung beigetragen haben, insbesondere an Éric Louvard, der dort für eine permanent robuste und leicht aktualisierbare Installation der Anwendung gesorgt hat. Vielen Dank auch an alle Entwickler und Mitentwickler der zahlreichen Open-Source-Produkte, die ich bei der Entwicklung und der Erstellung dieser Dokumentation mit viel Freude genutzt habe.

Wolfgang Maier

1 Bezug und Konfiguration

FlowProtocol 2 steht unter der MIT-Lizenz und ist unter folgender Adresse auf GitHub verfügbar :

`https://github.com/maier-san/FlowProtocol2`

Die Programmcode kann direkt mit Git oder der frei verfügbaren Entwicklungsumgebung Visual Studio Code in ein lokales Verzeichnis, z.B. `D:\Anwendungen\FlowProtocol2`, übertragen, und dort mit dem dotnet-Befehl kompiliert werden:

```
dotnet.exe build
    D:\Anwendungen\FlowProtocol2/FlowProtocol2/FlowProtocol2.csproj
```

Die Konfiguration erfolgt über die Datei `appsettings.json`, in der hauptsächlich der Pfad auf das Skripte-Verzeichnis mit dem Parameter `ScriptPath` eingestellt werden muss. Dieses Verzeichnis enthält die Skripte und Unterverzeichnisse, die von *FlowProtocol 2* auf der Startseite angezeigt werden, und ist damit der Dreh- und Angelpunkt der Skriptverwaltung. Für die ersten Versuche kann man den Parameter auf den Scripts-Ordner innerhalb des Projektes setzen, etwa so:

```
"ScriptPath": "D:\\Anwendungen\\FlowProtocol2\\Scripts",
```

In einer Firma oder Einrichtung wird man das Verzeichnis so wählen, dass die Mitarbeiter, die aktiv an den Skripten arbeiten, dort direkt Dateien editieren und erstellen können, z.B. indem man dieses auf einem Netzlaufwerk verfügbar macht. Zusätzlich wird man die Skripte regelmäßig sichern, idealerweise mit Hilfe einer Versionsverwaltung. Wenn man Manipulation befürchtet, kann man das Editieren auch vollständig auf den Weg über die Versionsverwaltung beschränken, verbaut sich damit aber die Möglichkeit, die Wirkung von Änderungen an einem Skript unmittelbar nach dem Speichern durch die Aktualisierung des Browser-Tabs zu überprüfen. In diesem Fall wäre dann eine getrennte Skript-Entwicklungsumgebung sinnvoll, analog zu den sonstigen Entwicklungsumgebungen. *FlowProtocol 2* selbst benötigt auf dieses Verzeichnis nur Leserechte.

Wie schon im Vorwort beschrieben ist die Möglichkeit, Links zu generieren, ein mächtiges Mittel um die Interaktion mit anderen Anwendungen zu ermöglichen. Wie man jedoch aus jeder IT-Sicherheitsbelehrung weiß, kann das Anklicken von Links auch Gefahren mit sich bringen, und speziell in Phishing-E-Mails nutzen Angreifer vertrauenswürdig aussehende Links, um den Empfänger auf eine nachgebaute oder mit Schadcode gespickte Seite zu leiten. Auch mit den Links in einem Skript sind solche Angriffe möglich, wenn auch aufgrund der Beschränkung auf die eigenen Mitarbeiter eher unwahrscheinlich. Aus diesem Grund werden alle durch ein Skript ausgegebenen Links, deren Domäne nicht in der Auflistung des `LinkWhitelist`-Parameters steht, zusätzlich zu dem Anzeigetext mit der vollständigen URL ausgegeben. Damit kann bei der Konfiguration entschieden werden, welche Seitenaufrufe so vertrauenswürdig sind, dass sie auch nur mit dem Anzeigetext dargestellt werden können, was zum einen die Lesbarkeit erhöht und zum anderen die Aufmerksamkeit des Anwenders auf die Links konzentriert, die nicht diese Einstufung haben.

Für die regelmäßige lokale Bereitstellung von *FlowProtocol 2* ist es am einfachsten, vom Basisrepository auf GitHub mittels Fork zu verzweigen und die lokalen Anpassungen im eigen Zweig zu verwalten.

2 Grundlagen

2.1 Das erste Beispiel

In diesem Abschnitt bleibt der Bezug zur Softwareentwicklung erst einmal darauf beschränkt, dass wohl jeder, der in dieser Branche tätig ist, schon an dem einen oder anderen Hallo-Welt-Beispiel vorbeigekommen ist. In dieser Tradition starten auch wir und beginnen mit dem einfachsten aller Beispiele:

Beispiel 1 – Hallo Welt

```
>> Hallo Welt!
```

Das bedeutet, wir erstellen eine neue Textdatei mit dem Dateinamen *01 Hallo Welt.fp2* und speichern diese in einem Unterordner im Skripte-Verzeichnis der *FlowProtocol 2*-Installation (vgl. Abschnitt 1). Und ja, spätestens jetzt ist es Zeit, *FlowProtocol 2* in einer eigenen Umgebungen zum Laufen zu bringen, denn nur durch Ausführen der gezeigten Beispiele und viel eigenem Rumprobieren ist es möglich, den bestmöglichen Nutzen aus dieser Anleitung zu ziehen. Natürlich sind alle Beispiele aus dieser Anleitung auch als Textdatei im Projekt enthalten, so dass man diese nur auszuwählen braucht.

Nach erfolgreicher Konfiguration wird man von *FlowProtocol 2* nach dem Start mit dem Text *Willkommen bei FlowProtocol 2* begrüßt und man sieht das dem Logo, das auch das Titelblatt dieser Anleitung ziert. Über die Schaltfläche *Zur Skriptauswahl* oder dem Menüpunkt *Start* gelangt man von dort aus zur Auflistung der Skript-Hauptgruppen, also den Ordnern im Skripte-Verzeichnis. Für das Durcharbeiten dieser Anleitung ist es am besten, man kopiert den Ordner *Doku-Beispiele* in das Skripte-Verzeichnis, so dass man die einzelnen Beispiele direkt aufrufen kann.

Die Ausführung des oben genannten Beispiels führt zur Ausgabe des Textes *1. Hallo Welt!* unter der Überschrift *01 Hallo Welt*.

Als Überschrift wird standardmäßig der Dateinahme ohne Endung ausgegeben, und da wir keinen Befehl angegeben haben, der explizit eine Überschrift setzt, ist das auch hier der Fall. Die Syntax >> gibt den dahinter stehenden Text aus, standardmäßig als nummerierte Aufzählung, was die *1.* erklärt.

Die Schaltflächen *Neue Ausführung* und *Zurück zur Startseite* werden immer angezeigt, wenn die Ausführung abgeschlossen ist. Mit *Neue Ausführung* kann das aktuell ausgewählte Skript neu gestartet werden und mit *Zurück zur Startseite* kommt man zurück zur Auswahl der Skript-Hauptgruppen, was auch mit dem Menüpunkt *Start* möglich ist. Wir werden noch einige Anwendungsfälle sehen, die sich am besten mit einem Skript umsetzen lassen, das mehrmals hintereinander ausgeführt wird.

2.2 Eine erste Auswahlabfrage

Bis jetzt haben wir nur Text in Form gebracht und in einem Browser-Fenster ausgegeben, was in Anbetracht der recht einfachen Syntax und der schönen Formatierungsmöglichkeiten auch schon Mehrwerte bieten kann, aber der Hauptnutzen von *FlowProtocol 2* liegt ganz klar in der Interaktivität. Diese lässt sich umsetzen mit Hilfe von Auswahlabfragen und Texteingaben. Das nachfolgende Beispiel zeigt eine einfache Auswahlabfrage.

Beispiel 2 – Hallo Welt mit Auswahl

```
?Q: Wie soll die Welt begrüßt werden?
    #h: Mit "Hallo Welt!"
        >> Hallo Welt!
    #a: Mit "Aloah Welt!"
        >> Aloah Welt!
```

Beim Ausführen dieses Skriptes wird man als Anwender zuerst mit der Frage *Wie soll die Welt begrüßt werden?* konfrontiert, mit den beiden Antwortmöglichkeiten *Mit "Hallo Welt!"* und *Mit "Aloah Welt!"*. Es muss einer der beiden vorgegebenen Auswahlmöglichkeiten gewählt werden, denn wenn man die Skriptausführung mit *Weiter* fortsetzt, werden die nicht beantworten Fragen einfach erneut gestellt und man gelangt nicht zur Ausgabeseite. Nach getroffener Wahl wird auf der Ausgabeseite dann entweder *1. Hallo Welt!* oder *1. Aloah Welt!* ausgegeben. Dort hat man dann die Möglichkeit, das eben ausgeführte Skript mittels *Neue Ausführung* erneut auszuführen oder zur Startseite der Skriptauswahl zu wechseln.

Das Grundprinzip, dass auf einer oder mehreren Eingabeseiten die eventuell auch voneinander abhängigen Eingaben abgefragt werden, um dann am Ende die daraus resultierenden Ausgaben auf einer Ausgabeseite anzuzeigen, gilt für alle Skripte.

Ein Blick in der Adresszeile des Browsers zeigt, dass die Antwort auf die Frage bei Wahl von *Hallo Welt* als Parametersequenz `Q=h` in der URL gespeichert wurde. Das ist auch der einzige Ort, wo sich diese Information wiederfindet, denn *FlowProtocol 2* selbst speichert keine Eingabedaten auf Serverseite. Die beiden Buchstaben `Q` und `h`, bzw. `a` sind hierbei durch den Skriptcode festgelegt und heißen Schlüssel. Die Auswahlabfrage wird durch das `?`-Zeichen eingeleitet, unmittelbar gefolgt vom Schlüssel der Frage, gefolgt von einem Doppelpunkt an den sich die als Text ausformulierte Fragestellung oder Eingabeaufforderung anschließt. Eingerückt auf erster Ebene stehen unter der Frage die verschiedenen Antwortmöglichkeiten, die jeweils mit dem `#`-Zeichen beginnen, analog gefolgt vom Schlüssel der Antwort, einem Doppelpunkt und der Ausformulierung der jeweiligen Antwortmöglichkeit oder Option.

Der wiederum unterhalb einer Antwortmöglichkeit eingerückte Skriptcode kann wieder aus Befehlen bestehen, die auch für sich alleine stehen können, Dieser Code wird wird nur dann ausgeführt, wenn die dazugehörende Antwortmöglichkeit ausgewählt wurde.

Die Schlüssel für Fragen und Antworten und auch für die später noch kommenden Texteingaben können aus einem oder mehreren Buchstaben und Zahlen bestehen und dienen wie schon genannt dazu, die Eingaben des Anwenders in der URL zu speichern und auch innerhalb des Skriptes abrufbar zu machen. Der Schlüssel für eine Auswahlabfrage muss eindeutig im gesamte Skript sein, der einer Antwortmöglichkeit muss eindeutig innerhalb einer Auswahlabfrage sein.

Da die Gesamtlänge der URL für Seitenaufrufe auf knapp über 2000 Zeichen begrenzt ist, muss man sich über die Länge von Schlüsseln erst Gedanken machen, wenn man wirklich viele Auswahlabfragen in ein Skript einbaut.

2.3 Allgemeine Ergänzungen zur Syntax

In *FlowProtocol 2* wird jede nicht leere Zeile einem Befehl zugeordnet. Die Einrückung am Anfang wird getrennt erfasst und verarbeitet, Leerzeilen und Leerraum am Ende wird ignoriert. Die Einrückung kann sowohl mit dem Tabulatorzeichen, also auch mit Leerzeichnen erfolgen, wobei das Tabulatorzeichen intern in vier Leerzeichen umgerechnet wird. Es sollte also vermieden werden, innerhalb einer Skriptdatei sowohl mit Tabulatorzeichen, als auch mit Lerzeichen einzurücken.

Die Zeichenfolge `//` leitet einen Kommentar ein, jedoch nur zu Beginn einer Zeile.

Lange Zeilen können auf mehrere Zeilen umgebrochen werden, wobei der Umbruch durch die Zeichenfolge `__` zu Beginn jeder Folgezeile kenntlich gemacht werden muss.

Beispiel 3 – Syntaxergänzungen

```
// Das ist ein Kommentar

>> Das ist eine Ausgabe,
    __ die im Scriptcode
    __ auf drei Zeilen verteilt wird.
```

2.4 Die FlowProtocol-2-Befehlsreferenz

Um einen Überblick über alle in *FlowProtocol 2* vorhandenen Befehle zu bekommen und für jeden die genaue Syntax, die dabei möglichen Fehler und die Verwendung anhand eines kleinen Beispiels nachschlagen zu können, ist die *FlowProtocol 2*-Befehlsreferenz der richtige Ort. Gemeint ist der Ordner *FP2-Tutorial*, der im *Skripts*-Ordner zusammen mit der Anwendung bereitgestellt wird. Dort ist für jeden Befehl ein Skript vorhanden, das diesen soweit es möglich ist, von anderen Befehlen unabhängig beschreibt. In vielen Fällen erzeugt das Skript direkt eine Ausgabe, die auch das Beispiel mit einschließt. Bei Befehlen die sich auf den Eingabeseiten auswirken, muss zuerst eine Eingabe durchlaufen werden, sodass man die Wirkung des dokumentierten Befehls direkt in der Programmoberfläche sehen kann.

2.5 Fehlermeldungen und Debug-Techniken

Das Entwickeln von Skripten wird nicht ohne Fehler ablaufen, wie auch? Wichtig ist, dass man diese schnell erkennt und korrigieren kann, und zielsicher an seiner Ergebnis kommt.

FlowProtocol 2 kann zwar nicht mit einem syntaxunterstützenden Editor aufwarten, liefert dafür aber klare Hinweise auf Syntax- und Laufzeitfehler. Diese werden mit Dateiname und Zeilennummer benannt. Der am häufigsten auftretende Fehler dürfte die *Parsing Exception* sein, also das Problem, dass eine Zeile nicht interpretiert werden kann. Da in*FlowProtocol 2* jeder Befehl in einer eigenen Zeile angegeben wird, muss man dementsprechend nur noch schauen, welchen Befehl man in der entsprechenden Zeile verwenden wollte, und wo die Zeile syntaktisch abweicht.

Hier ein Beispiel für so einen Fehler:

Beispiel 4 – Fehler im Skript

```
?W: Wo ist der Fehler?
    #A: In Antwort A
        >> Antwort A
    #B: In Antwort B
        Antwort B
```

Laufzeitfehler treten dagegen meist im Zusammenhang mit Variablen auf, z.B wenn mit diesen Rechenoperationen durchgeführt werden sollen, diese aber keinen Zahlenwert enthalten. Auch hier kann man anhand der Angaben in der Fehlermeldung die Stelle lokalisieren und sich an die Klärung der Ursache machen.

In den meisten restlichen Fällen erscheint keine Fehlermeldung, aber das Skript produziert nicht die gewünschte Ausgabe. Und dann geht es an das Debuggen also das Entfernen der Fehler. Hier gibt es bei *FlowProtocol 2* den großen Vorteil, dass das Feedback nach einer Änderung sehr schnell zu bekommen ist. Man ändert einfach eine Stelle im Skriptcode, speichert und drückt die Aktualisieren-Schaltfläche im Browser und erhält umgehend das Resultat der aktualisierten Version. Man muss weder neu kompilieren, noch irgendwelche Eingaben wiederholen.

Der beste Ansatz, um inhaltlichen Fehlern auf die Spur zu kommen, besteht darin, an vielen Stellen Hilfsausgaben zu erzeugen, z.B mit den Werten von Variablen, der Information, ob eine bestimmten Stelle durchlaufen wurde, und so weiter. Diese können ganz normal als Ausgaben in das Skript eingebaut werden, idealerweise gesammelt in einem eigenen Debug-Abschnitt (siehe Abschnitt 4.1). Sobald die Entwicklung des Skriptes abgeschlossen ist, werden diese Hilfsausgaben auskommentiert oder entfernt.

3 Entscheidungsbäume

Aus mehreren ineinander verschachtelten Auswahlabfragen lassen sich große Entscheidungsbäume und Flussdiagramme aufbauen, deren Stärke darin liegt, an jeder Stelle im Skriptcode die vollständige bis dahin getroffene Auswahl zu kennen, und so die nächste Fragestellung ganz exakt auf diese Situation abzustimmen. Die Anwendungsfälle dafür sind enorm vielfältig, aber um im Bereich der Softwareentwicklung zu bleiben, sind hier einige typische Beispiele.

3.1 Strukturabbildung einer Software

Durch den Nachbau der Struktur, bzw. des Menübaums einer Software von der Programmoberfläche aus betrachtet, kann man es dem Anwender des Skriptes enorm erleichtern, eine Stelle innerhalb dieses Softwareproduktes eindeutig zu benennen. Der daraus resultierende Pfad oder eine diesem zugeordnete Kennung kann für die Suche nach Vorgängen und Informationen verwendet werden. Da diese Art von Information oft Teil abteilungsübergreifender Kommunikation ist, können mit einer solchen Auswahl sowohl Missverständnissen vermieden, und auch längerer Tastatureingaben eingespart werden.

3.2 Kategorisierungshilfen

Eine wohl in allen Softwarehäusern stattfindende Kategorisierung ist die des Schweregrades von neu erfassten Fehlern. Diese variieren in der Regel von *hoch kritisch* bis hin zu *niedrig* oder einer numerischen Abstufung mit dieser Entsprechung. Die subjektive und oft situationsbedingte Einschätzung der beteiligten Personen sollte hierbei nicht der Gradmesser sein, sondern vielmehr die objektive Kriterien und der Vergleich mit verständlichen Beispielen. Auf diese Weise können die verfügbaren Ressourcen zielgerichtet eingesetzt werden.

Die nachfolgende Schweregradeinstufung beginnt damit, explizit nach Kriterien für die höchste Einstufung zu fragen und geht weiter zu den Kriterien der zweiten Stufe, wenn diese nicht zutreffen. Nach der zweiten Stufe wird direkt nach Kriterien für die unterste Stufe gefragt, so dass automatisch für alles die Zuordnung zu Stufe 3 erfolgt, was auch dort nicht zugeordnet werden kann. Das macht die Aufstellung der Kriterien leichter, da sich einfacher Kriterien für die ganz niedrige Priorität finden lassen, als für die vorgelagerte Stufe.

Beispiel 5 – Schweregradeinstufung

```
?S1: Treffen die Kriterien für einen Stufe-1-Fehler zu?
    #k1: Ausgegebene Werte entsprechen nicht der Spezifikation.
        >> Stufe-1-Fehler: Spezifikationsverletzung
    #k2: Der Datenschutz ist an einer wesentlichen Stelle verletzt.
        >> Stufe-1-Fehler: Datenschutzverletzung
    #k3: Der Datensicherheit ist verletzt oder gefährdet.
        >> Stufe-1-Fehler: Datensicherheitssverletzung
    #no: Nein, die o.g. Kriterien sind nicht erfüllt.
        ?S2: Treffen die Kriterien für einen Stufe-2-Fehler zu?
            #k1: Der Fehler stört zentrale Programmfunktionen.
                >> Stufe-2-Fehler: Störung zentraler Funktion.
            #k2: Es gibt zwei oder mehr Supportanfragen zu
                __ diesem Fehler.
                >> Stufe-2-Fehler: Mindestens zwei Supportanfragen
            #no: Nein, die o.g. Kriterien sind nicht erfüllt.
                ?S4: Treffen die Kriterien für einen
                    __ Stufe-4-Fehler zu?
                    #k1: Es handelt sich um einen erkennbaren
                        __ Tippfehler.
                        >> Stufe-4-Fehler: Tippfehler
                    #k2: Der Fehler tritt nur bei unsinnigen
                        __ Eingaben auf.
                        >> Stufe-4-Fehler: unsinnige Eingaben
                    #no: Nein, die o.g. Kriterien sind nicht erfüllt.
                        >> Stufe-3-Fehler
```

Für die Abarbeitung der Fehler-Aufgaben der verschiedene Schweregradstufen kann man dann klare Regeln definieren, wie z.B. dass Stufe-1-Fehler umgehend behoben werden

und dass die Lösung aller Stufe-2-Fehler zumindest in das nächste Service Release einfließen muss.

Insgesamt ist die Sortierung nach dem Schweregrad keine gute Idee, wenn sich viele Vorgänge ansammeln, da es in diesem Fall erst dann zur Bearbeitung eines Stufe-4-Fehlers kommt, wenn alle Stufe-3-Fehler abgearbeitet sind, was unter Umständen nie der Fall ist. Ein System, das die Einstufung berücksichtigt, aber gleichzeitig die Abarbeitung jeder Aufgabe sicherstellt, ist angelehnt an die Spuren der Autobahn. Die Aufgaben jeder Schweregradstufe bilden jeweils eine Queue, in der neu erstellte Aufgaben unten angefügt und die ältesten oben abgearbeitet werden. Um den Schweregrad zu berücksichtigen, lässt man die linke Spur schneller laufen, indem man die Mengen für die Abarbeitung in ein vorgegebenes Verhältnis setzt, z.B.

$$M(2) : M(3) : M(4) = 6 : 3 : 1$$

d.h. für eine abgearbeitete Stufe-4-Aufgabe werden drei Stufe-3-Aufgaben und sechs Stufe-2-Aufgaben abgearbeitet.

3.3 Expertensysteme

Mit Expertensystem oder auch Troubleshooting-System oder Chatbot ist hier eine Anwendung gemeint, die wie ein lebendiger Experte durch immer weiter in die Tiefe gehende Fragestellungen eine in der Regel problematische Situation möglichst genau erfasst, um dann dem Anwender dazu passende Handlungsanweisungen und Lösungsansätze aufzuzeigen und die Situation darüber hinaus auch schon mal vollständig zu beschreiben. Probleme gibt es überall und die zur Lösung befähigten Experten erkennt man zuallererst daran, dass sie die richtigen Fragen stellen. Die Möglichkeit, einen großen Teil der Probleme auch in Abwesenheit menschlicher Experten lösen zu können, ist unheimlich wertvoll, insbesondere dann, wenn eine gut gemachte Anleitung Zeit spart und die Sicherheit gibt, nicht irgendetwas wildes auszuprobieren.

Wie auch schon die oben genannten Beispiele, die auch kombiniert werden können, zeigt sich hier der enormen Nutzen, der sich schon aus diesen wenigen Skriptbausteinen ergibt. Der Grund dafür ist nicht, dass diese Befehle irgendetwas raffiniert technisches machen, sondern dass sie es ermöglichen, strukturelles oder fachliches Wissen in eine Form zu bringen, so dass dieses Wissen vom Anwender unmittelbar genutzt werden kann, ohne dass er es sich aneignen muss. *FlowProtocol 2* übernimmt die Anwendung des Wissens, der Anwender selbst muss sich immer nur mit einer einzelnen und idealerweise für ihn verständlichen Fragestellung auseinandersetzen, bekommt Führung und Anleitung und muss nicht selbst mühsam zum Experten werden.

Der Umfang solcher Entscheidungsbäume kann riesig werden und trotzdem muss der Anwender am Ende nur eine Handvoll Fragen beantworten, um zum Ziel zu kommen.

Hier ist ein sehr vereinfachtes Beispiel für ein Expertensystem, das eine Problemsituation analysiert und eine Handlungsempfehlung gibt:

Beispiel 6 – Mini-Expertensystem

```
?: Startet die Anwendung?
    #: Ja
```

```
    >> Anwendung startet
    ?: Ist eine Benutzeranmeldung möglich?
        #: Ja
            >> Benutzeranmeldung möglich.
            >> Wo liegt das Problem?
        #: Nein
            >> Benutzeranmeldung nicht möglich
            >> Empfehlung: Passwort zurücksetzen
#: Nein
    >> Anwendung startet nicht
    ?: Wurde das System schon neu gestartet?
        #: Ja
            >> System wurde schon neu gestartet.
            >> Empfehlung: 2nd-Level-Support kontaktieren
        #: Nein
            >> System wurde noch nicht neu gestartet.
            >> Empfehlung: System neu starten
```

Zuerst fällt auf, dass hier keine Schlüssel angegeben sind, weder für die Auswahlabfragen, noch für die Antwortmöglichkeiten. Tatsächlich sind diese nicht zwingend erforderlich und werden von *FlowProtocol 2* selbständig anhand der Reihenfolge im Code vergeben, wenn sie im Skript weggelassen werden. Dies ist dann möglich, wenn man sich nicht an einer anderen Stelle im Skript auf eine Auswahl beziehen oder dieses wiederholen möchte, und es ist dann besonders vorteilhaft, wenn der Baum sehr groß werden kann und permanent erweitert wird und man den Überblick über die schon vergebenen Schlüssel zu verlieren droht. Der Nachteil der impliziten Schlüsselvergabe liegt darin, dass sich durch das Einfügen einer weiteren Auswahlabfrage am Anfang alle darunter liegenden Schlüssel verschieben, und so eine URL, die die Antworten in Bezug auf die Vorversion enthält, in der neuen Version falsch interpretiert wird.

Die Ausführung dieses Beispiels zeigt darüber hinaus, wie das System mit ineinander verschachtelten Fragestellungen umgeht. In jedem Schritt werden immer genau die Fragen gestellt, die noch offen sind, und die aufgrund der vorangegangenen Schritte durchlaufen werden müssen. Sobald alle Fragen beantwortet sind, werden die gesammelten Ausgaben ausgegeben und die Ausführung des Skriptes ist abgeschlossen.

4 Formatierung der Ausgabe

Dieser Abschnitt widmet sich der Formatierung der Ausgabe und beschreibt Möglichkeiten, Abschnitte zu bilden, Aufzählungen zu verschachteln, sowie Links und Code in die Ausgabe zu integrieren.

4.1 Ausgabe in Abschnitten

Bis jetzt habe wir nur den Befehl >> verwendet, um Aufzählungspunkte auszugeben, und dabei merkt man recht schnell, dass das Ergebnis zumeist nicht nur aus einer Liste besteht, oder bestehen könnte. Nehmen wir das Expertensystem aus Beispiel 3.3. Die Beschreibung der Situation und der schon durchgeführten Maßnahmen ist inhaltlich betrachtet eine

Liste für sich, die Handlungsempfehlung eine andere und läuft alles am Ende in eine Kontaktierung des 2nd-Level-Supports hinaus, könnte die Liste der benötigten Informationen und Materialien als dritte Liste ausgegeben werden.

Dieser Anwendungsfall zeigt auch schon, dass es durchaus üblich ist, dass verschiedenen Listen in einem Skript nicht nacheinander, sondern parallel zusammengestellt werden. Dies ist auch im folgenden Beispiel der Fall, das eine sehr einfache, nicht interaktive Implementierungsanleitung darstellt, die einen Entwickler bei der Implementierung eines neuen Moduls anleitet. Da jeder Entwicklungsschritt auf einen Integrationstest nach sich ziehen sollte, und der Entwickler in diesem Fall auch für die Erstellung der Testpunkte verantwortlich ist, wird zusammen mit der Anleitung auch gleich noch eine Liste von Testpunkten ausgegeben.

Beispiel 7 – Ausgabe in Abschnitten

```
@Anleitung >> Erstelle eine neue Modul-Klasse
@Testpunkte >> Das Modul wird korrekt gestartet
@Anleitung >> Implementiere die Start-Methode
@Testpunkte >> Das Modul wird korrekt beendet
@Anleitung >> Implementiere die Beenden-Methode
@Anleitung >> Nehme das Modul in den Modulkatalog auf
@Anleitung >> Übernehme die Testpunkte von unten
```

Die Ausführung erzeugt einen Abschnitt *Anleitung* mit entsprechender Überschrift und den fünf Punkten der Anleitung, gefolgt von einem Abschnitt *Testpunkte* mit den beiden Testpunkten. Im Gegensatz zur Zusammenstellung erfolgt die Ausgabe Abschnittsweise, wobei die Reihenfolge durch die jeweils erste Ausgabe des Abschnitts festgelegt wird.

Die Erzeugung einer Ausgabe in einem Abschnitt setzt gleichzeitig diesen Abschnitt für alle darauffolgenden Ausgaben, sofern diese nicht explizit einem Abschnitt zugeordnet sind. Um also nur die Anleitung auszugeben, reicht eine einzige Festlegung des Abschnitts am Anfang:

Beispiel 8 – Nur ein Abschnitt

```
@Anleitung >> Erstelle eine neue Modul-Klasse
>> Implementiere die Start-Methode
>> Implementiere die Beenden-Methode
>> Nehme das Modul in den Modulkatalog auf
```

4.2 Unterpunkte und Absatzformate

Eine gute Anleitung ist gleichermaßen geeignet für erfahrene und weniger erfahrene Benutzer und ergänzt die primär durchzuführenden Schritte mit einer detaillierten Beschreibung der dazugehörenden Teilschritte. *FlowProtocol 2* unterstützt generell zwei Aufzählungsebenen, die mit den Befehlen >> und > angesteuert werden, jeweils mit der optional möglichen Angabe eines Abschnitts mittels `@`. Ausgaben der zweiten Ebene gliedern sich der letzten Ausgabe der ersten Ebene des entsprechenden Abschnitts unter.

Zusätzlich kann man für beide Ebenen angeben, ob eine Ausgabe als nummerierter Aufzählungspunkt (>>#), nicht nummerierter Aufzählungspunkt (>>*), einfache Textzeile (>>_) oder Codezeile (»|) ausgegeben wird.

Beispiel 9 – Unterpunkte

```
@Anleitung >>_ Geschätzter Zeitaufwand ca. 15 min
>>* Erstelle eine neue Modul-Klasse.
    >* Erstelle C#-Datei mit dem Namen des Moduls.
    >* Füge folgenden Code ein:
    >| public class NeuesModul : BaseModul
    >| {
    >| }
>>* Implementiere die Start-Methode
    >#  Gibt die Zeichenfolge "override" innerhalb der Klasse ein
    >#  und wähle in der Auswahl des Editors die Methode Start.
    >#  Ergänze den Start-Code des Moduls wie im Wiki beschrieben.
```

Die komplette Ausgabe erfolgt im Abschnitt *Anleitung*, die Angabe des geschätzten Zeitaufwands erfolgt als normale Textzeile, die Aufzählungspunkte der ersten Ebene sind entgegen dem Standard nicht nummeriert. Unter dem ersten Aufzählungspunkt werden zwei ebenfalls nicht nummerierte Unterpunkte ausgegeben, gefolgt von einem Codeblock mit drei Zeilen. Unter dem zweiten Aufzählungspunkt sieht man drei Unterpunkte die mit Kleinbuchstaben durchnummeriert sind,

Die drei aufeinanderfolgenden Codezeilen werden zu einem Codeblock zusammengefasst, der mit der automatisch darunter angeordneten Schaltfläche *In Zwischenablage kopieren* in die Zwischenablage genommen werden kann.

Auf die vielfältigen Möglichkeiten der Codegenerierung kommen wir später zurück. Im nächsten Abschnitt bleiben wir nochmal bei der Formatierung und zeigen, wie sich Text innerhalb einer Zeile formatieren lässt.

4.3 Links und Inline-Code

Gerade in Anleitungen wird auf Stellen im Code in Form von Klassen-, Methoden- oder Variablennamen verwiesen, und da erleichtert es den Lesefluss deutlich, wenn man diese Textelemente in der Ausgabe entsprechend hervorhebt. Dies ist mit dem Befehl `~AddCode` möglich, der der Ausgabe der letzten Zeile Text hinzufügt, der als Code formatiert ist. Mit dem Befehl `~AddText` kann danach wieder weiterer Text hinzugefügt werden. Man beachte das zusätzliche Leerzeichen, das immer am Anfang des angehängten Textes angefügt werden muss, wenn zwischen diesem und dem vorangegangen Text ein Leerzeichen stehen soll, da Leerraum am Zeilenende generell ignoriert wird.

Als Anwendung im Browser kann *FlowProtocol 2* natürlich auch Links erzeugen, und damit den Anwender sowohl zu statischen Seiten, also auch zu anderen Anwendungen weiterleiten. Auch dieser Möglichkeit widmen wir später noch einen eigenen Abschnitt. Ein Link besteht dabei aus einer URL und dem anzuzeigenden Text, die in dieser Reihenfolge, getrennt von einem vertikalen Strich angegeben werden. Auch hier kann mit `~AddText` wieder weiterer Text im Anschluss angefügt werden.

Beispiel 10 – Links und Inline-Code

```
@Anleitung >> Implementiere die Start-Methode
    > Gibt die Zeichenfolge
    ~AddCode  override
    ~AddText  innerhalb der Klasse ein
    > und wähle in der Auswahl des Editors die Methode Start.
    > Überschreibe die Start-Methode wie im
    ~AddLink  https://learn.microsoft.com/de-de/dotnet/csharp
        __/language-reference/keywords/override | Internet
    ~AddText  beschrieben.
```

Das Beispiel formatiert den Text *override* als Code und den Text *Internet* als Link, also blau, unterstrichen und klickbar. Sofern die Domäne der angegebenen URL in der Konfiguration nicht als sicher angegeben ist (siehe Abschnitt 1), wird die URL aus Sicherheitsgründen hinter dem Anzeigen-als-Text ergänzt. Der Link wird standardmäßig immer in einem neuen Tab geöffnet.

5 Variablen und Texteingaben

Die erste Zielsetzung der Vorversion von *FlowProtocol 2* bestand tatsächlich nur darin, eine große Menge an möglichen Ausgaben mittels iterierten Fragestellungen auf eine spezifische Situation einzugrenzen und die damit verbundenen Anwendungsfälle abzudecken. Sehr schnell wurde jedoch klar, dass mit ein paar grundlegenden Erweiterungen noch sehr viel mehr möglich ist.

5.1 Variablen setzen und verwenden

Unter einer Variablen versteht man kurz gesagt einen Platzhalter, der verschiedene Werte annehmen kann. Innerhalb von Programmier- oder Skriptsprachen lassen sich die Werte von Variablen setzen und auch abrufen. In *FlowProtocol 2* wird einer Variablen mit dem `~Set`-Befehl ein Wert zugewiesen. Die Variable kann dann an nahezu allen Stellen im Skript verwendet werden, indem man Sie mit vorangestelltem `$`-Zeichen kennzeichnet.

Beispiel 11 – Hallo Welt mit Variable

```
?Q: Wie soll die Welt begrüßt werden?
    #h: Mit "Hallo Welt!"
        ~Set Gruss=Hallo
    #a: Mit "Aloah Welt!"
        ~Set Gruss=Aloah
>> $Gruss Welt!
```

Das Beispiel setzt die Variable `Gruss` je nach Auswahl auf den Wert *Hallo* oder *Aloah* und ruf den Wert in der Ausgabe am Ende auf, so dass entweder *Hallo Welt!* oder *Aloah Welt!* ausgegeben wird.

Eine Variable kann aus Buchstaben (ohne Umlaute), Zahlen, runden Klammern und anderen Variablen zusammengesetzt werden. Der Wert einer Variable kann jede beliebige Zeichenkette sein.

Die Ersetzung von Variablen durch die jeweils zugeordneten Werte erfolgt durch einfache Ersetzung, d.h. das Ende der Variablen muss beim Aufruf nicht gekennzeichnet werden. Variablen, die nicht gesetzt wurden, werden auch nicht ersetzt.

Beispiel 12 – Variablenersetzung

```
~Set X=abc
>> X = $X
>> Z = $Z (Z wurde nie zugeweisen)
>> XY = $XY (vor der Zuweisung von XY)
~Set XY=def
>> XY = $XY (nach der Zuweisung von XY)
~Set i=5
~Set F($i)=ghi
>> F(i) = F($i) = $F($i)
```

Dieses Beispiel zeigt den Umgang mit Variablen, deren Anfang identisch mit einer anderen Variable ist, in diesem Fall `XY`. Die Ersetzung erfolgt in diesem Fall absteigend sortiert, sodass `$XY` vor `$X` ersetzt wird. Die Verwendung der Variablen `$i` im Ausdruck `$F($i)` gibt schon einen Ausblick darauf, dass man auch ganze Felder von Variablen anlegen und durchlaufen kann. Bei der Auswertung wird hierbei zunächst `$i` durch `5` und dann `$F(5)` durch `ghi` ersetzt. Die runden Klammern sind hier übrigens nur schmückendes Beiwerk, um den Index-Teil der Variablen hervorzuheben und vom Feldname abzugrenzen.

Wir werden später noch viele andere Befehle kennenlernen, die Zeichenketten verarbeiten und die das Ergebnis in einer Variablen zurückgeben und lernen auch selbst Funktionen zu schreiben, die ihre Eingabewerte über Variablen übergeben bekommen.

5.2 Bewertungssysteme

Das Beispiel in diesem Abschnitt verwendet eine Variable, um einen Zahlenwert zu verwalten und in Abhängigkeit der Auswahlwerte Werte zu addieren. Dies ist z.B. dann erforderlich, wenn man mehrere gleichartige Dinge nach einem formalen Bewertungssystem bewerten möchte, um sie anschließend anhand der daran angeschlossenen Metrik in eine Rangfolge zu bringen.

Nehmen wir konkret die Bewertung von Aufgaben, die zur Stablilisierung bestimmter Programmfunktionen in einem Softwareprodukt erstellt wurden. Die Erstellung und Ausformulierung solcher Aufgaben werden meist den Entwicklern selbst überlassen, da nur diese über das Wissen und den technischen Einblick verfügen, um dort Verbesserungspotential auszumachen. Am Ende sollte es aber auch hier der Product Owner sein, der über Umfang und Priorität der Maßnahmen entscheidet, und um die dafür relevanten Aspekte der Entwicklung herauszuarbeiten, könnte er jeder dieser Aufgaben mit dem folgendem Bewertungssystem bewerten lassen:

Beispiel 13 – Bewertungssystem

```
~Set punkte=0
@Bewertung >>* Bewertungkrterien:
?A1: Wie groß ist der Verbesserungsbedarf an dieser Stelle?
    #w1: Gering. Es gibt wenig Meldungen zu Einschränkungen an
        __ dieser Stelle.
        >* Bedarf: gering (-)
        ~AddTo punkte+=10
    #w2: Mittel. Es gibt immer wieder Meldungen zu spürbaren
        __  Einschränkungen an dieser Stelle.
        >* Bedarf: mittel
        ~AddTo punkte+=25
    #w3: Groß. Es permanent Meldungen zu störenden Einschränkungen
        __ an dieser Stelle.
        >* Bedarf: groß (+)
        ~AddTo punkte+=50
?A2: Wie klar ist die durchzuführende Maßnahme beschrieben?
    #w1: Unkrokret. Es sind noch Analysen und Vorarbeiten notwendig.
        >* Klarheit: unkonkret (-)
        ~AddTo punkte+=10
    #w2: Hinreichend konkret. Der Ansatz ist klar beschrieben,
        __ aber noch nicht erprobt.
        >* Klarheit: konkret
        ~AddTo punkte+=25
    #w3: Übertragbar. Der Ansatz kann von einer anderen Stelle
        __ hierher übertragen werden.
        >* Klarheit: übertragbar (+)
        ~AddTo punkte+=50
?A3: Wie effektiv wird die Maßnahmen voraussichtlich sein?
    #w1: Unklar. Der tatsächliche Effekt ist erst nach der
        __ Umsetzung erkennbar.
        >* Effektivität: unklar (-)
        ~AddTo punkte+=10
    #w2: Gering bis mittel. Es bleiben Einschränkungen, aber
        __ weniger oft und weniger groß.
        >* Effektivität: gering bis mittel
        ~AddTo punkte+=25
    #w3: Mittel bis gut. Die vorhandenen Einschränkungen werden
        __ weitestgehend behoben.
        >* Effektivität: mittel bis gut (+)
        ~AddTo punkte+=50
?A4: Wie aufwändig ist die Umsetzung der Maßnahme?
    #w1: Sehr aufwändig. 20 Storypunkte oder mehr.
        >* Aufwand: hoch (-)
        ~AddTo punkte+=10
    #w2: Aufwändig. 13-20 Storypunkte.
        >* Aufwand: mäßig hoch
```

```
        ~AddTo punkte+=25
    #w3: Im Rahmen. Maximal 8 Storypunkte.
        >* Aufwand: im Rahmen (+)
        ~AddTo punkte+=50
?A5: Wie gut ist die Wiederverwendbarkeit?
    #w1: Gering. Die Lösung ist speziell auf diese eine Stelle
        __ zugeschnitten.
        >* Wiederverwendbarkeit: gering (-)
        ~AddTo punkte+=10
    #w2: Übertragbar. Die Lösung kann auf andere Stellen
        __ übertragen werden.
        >* Wiederverwendbarkeit: übertragbar
        ~AddTo punkte+=25
    #w3: Umfassend. Die Lösung wird zentral eingebaut und
        __ wirkt sich an mehren Stellen aus.
        >* Wiederverwendbarkeit: umfassend (+)
        ~AddTo punkte+=50
@Bewertung >>* Gesamtbewertung: $punkte Punkte.
```

Im Anschluss wird die Variable `punkte` auf den Wert 0 gesetzt, was man auch weglassen könnte, und danach folgen fünf Fragen, die jeweils ein Kriterium der in durch die Aufgabe beschriebenen Entwicklungsmaßnahme abfragen. Für jede Frage werden drei Auswahlmöglichkeiten angeboten, die jeweils aufsteigend eine geringe bis gute Erfüllung des genannten Kriteriums beschreiben. Der ausgewählte Wert wird dann sowohl in einer Zusammenfassung in Textform ausgegeben, und auch in der Gesamtpunktezahl berücksichtigt, indem zu dem in der Punkte-Variablen vorhandenen Wert mehr oder weniger hinzugezählt wird. Die Erhöhung einer Variablen um einen Wert erfolgt dabei mit dem Befehl `~AddTo`. Die Gesamtpunktezahl wird dann ebenfalls am Ende ausgegeben.

Die Verwendung eines solchen Bewertungsskriptes hat primär den Vorteil, dass die Bewertung sehr objektiv ausfällt, da die Einschätzung in Bezug auf einzelnen Kriterien schon von sich aus reflektiert und nicht mittels Bauchgefühl passieren muss, was zusätzlich dadurch verstärkt werden kann, dass die Formulierung der Auswahlwerte eine gewisse Belegbarkeit assoziiert, die insbesondere bei einer gemeinsamen Bewertung im Team auch ausdiskutiert werden kann. Durch das Festhalten der ausgewählten Kriterien zusammen mit der Bewertung ist deren Begründung zudem gut dokumentiert. Die wiederholte Auseinandersetzung mit den zum größten Teil wirtschaftlichen Kriterien schärft in diesem Beispiel auch gleich die Sichtweise der Entwickler auf diese Aspekte, so dass diese bei der Suche nach Verbesserungen immer mehr in den Fokus gelangen.

Ob man nun wie im Beispiel oben eine einfache additive Bewertung abbildet, oder komplexere Formeln, die auch berücksichtigt, dass sich Faktoren gegenseitig verstärken können, bleibt der eigenen Kreativität überlassen. Mit *FlowProtocol 2* lassen sich alle Formeln realisieren.

Viele zumeist abstrakte Fragestellungen lassen sich auf derartige Bewertungssysteme herunterbrechen: Wie stark wird durch eine Entwicklungsmaßnahme die Produktstrategie verfolgt? Welcher über die Jahre gerechnete Mehraufwand wird durch Support und Aufrechterhaltung einer Funktion notwendig sein? In der Regel sind es viele kleine Faktoren, wie die Abhängigkeit von Drittanbieter-Komponenten oder externen Diensten die Länge der Kommunikationswege, der Grad der Konfigurierbarkeit, das Hinzukommen neuer

Strukturen mit Semantik und Bedienkonzepten und die allgemeine Komplexität und Fehlertoleranz, die dauerhaft Aufwand verursachen. Die Identifikation dieser Faktoren kann zum einen schon bei der Planung der Entwicklung helfen, die Kosten-Nutzen-Verteilung besser einzuschätzen, zum anderen können durch ein geeignetes Skript aber auch direkt geeignete Maßnahmen abgeleitet werden, um diese Dinge bestmöglich und systematisch in den Griff zu bekommen.

5.3 Texteingaben

Schon bei der Erstellung der ersten Skripte gab es die Assoziation zu den zahlreichen Konfigurationsportalen, mit denen man im Internet sein Fahrrad oder Auto nach Wunsch zusammenstellen konnte, und daran angelehnt auch die ersten Versuche, die Konfiguration rund um die vielfach wiederkehrenden Standardentwicklungen in gleicher Weise abzufragen und es so dem Product Owner zu erleichtern, die passende Konfiguration zu wählen. Da die Auswahl schon technisch auf auf die Menge der möglichen Konfigurationen beschränkt blieb, gleichzeitig aber alle notwendigen Entscheidungen abgefragt wurden, konnte man Aufgabenbeschreibungen generieren, die fast alle wesentlichen Konfigurationsaspekte auflisteten. Fast alle deshalb, da die Benennung der zu erstellenden Instanz eben nicht abgefragt werden konnte, und deshalb durch einen Platzhalter *xxx* in der Ausgabe ersetzt werden musste, was den Komfort deutlich schmälerte.

Inzwischen gehören Texteingaben zum Grundumfang von *FlowProtocol 2* und zusammen damit wurden auch zahlreiche Befehle hinzugefügt, um Texteingaben zu verarbeiten.

Beispiel 14 – Textersetzung

```
~Input si: Suche in
~Input sn: Suche nach
~Input ed: Ersetze durch
~Replace erg=$si|$sn->$ed
@Aufgabe >> Ersetze "$sn" in "$si" durch "$ed".
@Ergebnis >> $erg
```

In diesem Beispiel werden die drei Texte für eine einfache Ersetzungsaufgabe mit dem `~Input`-Befehl abgefragt und über die zu den Schlüsseln gehörenden Variablen `si`, `sn` und `ed` an den `~Replace`-Befehl übergeben, der das Ergebnis über die Variablen `erg` bereitstellt.

Mit der Möglichkeit, Texte oder Zahlen einzugeben lassen sich jede Menge kleine und größere Hilfsfunktionen realisieren, die sich direkt im Browser aufrufen lassen und damit die eine oder andere Datei auf Basis von Tabellenkalkulation überflüssig machen, besonders, wenn sie eine gute Benutzerführung aufweisen. Hierfür schauen wir uns im nächsten Abschnitt an, was für weitere Formatierungsmöglichkeiten es gibt.

6 Weitere Formatierungsmöglichkeiten

In Abschnitt 4 haben wir ja schon gesehen, wie man das Format der Aufzählung variieren, und die Ausgabe in Abschnitte unterteilen kann. Hier gibt es nur noch wenig zu ergänzen. Mindestens in gleicher Weise wichtig sind jedoch die Formatierungsmöglichkeiten,

bei den Seiten, in denen die Eingabe abgefragt wird. Sie müssen den Bogen zum Aufgabenkontext schließen und genau erklären und dem Anwender dabei helfen, die richtigen Daten einzugeben und die korrekte Auswahl zu treffen.

6.1 Ausgabetitel und Abschnittsverschiebungen

Der Titel der auf der Ausgabeseite angezeigt wird, kann mit dem Befehl `~SetTitle` unabhängig vom Dateinamen gesetzt werden, so dass man als Dateinamen und damit auch für die Auflistung im Menü kürzere Titel wählen kann. Durch das Setzen des Titels im Scriptcode lassen sich auch Variablen verwenden, die die gemachten Eingaben berücksichtigen. Gerade wenn man die Ergebnisseite eines Skriptes an jemand anderen weitergeben möchte, z.B. über den Link mit den Parametern, kann ein möglichst passender Titel helfen, um das Dokument einzuordnen.

Beim Verschieben eines Abschnitts wird der Inhalt eines Abschnitts an einen anderen Abschnitt angehängt, wobei dieser ggf. zuvor erstellt wird. Dies ermöglicht es, verschiedene Informationen zunächst parallel in verschiedenen Abschnitten zu sammeln und diese dann in einem Abschnitt aneinanderzufügen. Das ist besonders nützlich, wenn die Reihenfolge der Informationen in der Ausgabe eine andere ist als in der Abfrage.

Beispiel 15 – Abschnittsverschiebung

```
~Input Bez: Wie soll das Feld heißen?
@AnlEig >> Setze die Bezeichnung "$Bez"
?: Welche Art von Werten soll eingegeben werden?
    #: Texte
        ~Set Klasse=TextFeld
    #: Zahlen
        ~Set Klasse=Zahlenfeld
    #: Datumsangaben
        ~Set Klasse=Datumsfeld
    #: Werte aus einer vorgegebenen Menge
        ~Set Klasse=Auswahlfeld
~SetTitle Implementerung der $Klasse-Instanz $Bez
@AnlInst >> Erstelle eine $Klasse-Instanz
?: Kann das Feld leer gelassen werden?
    #j: Ja
        ~Set KannLeerSein=true
    #n: Nein
        ~Set KannLeerSein=false
@AnlEig >> Setze KannLeerSein = $KannLeerSein
~MoveSection AnlInst -> Anleitung
~MoveSection AnlEig -> Anleitung
```

In diesem Beispiel wird eine kleine Anleitung für den Einbau eines Feldes in ein Programm erstellt. Hierbei werden Bezeichnung, Typ und die Eigenschaft, ob das Feld leer gelassen werden kann, abgefragt. Sobald Bezeichnung und Typ festgelegt sind, wird der Titel unter Verwendung dieser Informationen neu gesetzt. Die Anleitung wird zunächst

in zwei Abschnitten getrennt zusammengestellt, `@AnlEig` für die beiden Eigenschaften und `@AnlInst` für die Instanziierung. Am Ende werden die beiden Abschnitte mit dem `~MoveSection`-Befehl nacheinander in den neuen Abschnitt *Anleitung* überführt, so dass die Punkte dort in einer logischen Reihenfolge stehen.

Für dieses Minimalbeispiel hätte man die gewünschte Reihenfolge auch ohne die Verschiebung von Abschnitten hinbekommen, aber gerade bei größeren Anleitungsskripten kann eine Aufteilung helfen, die jeweils zu einem Anleitungsschritt dazugehörenden Aspekte wie Programmcode und Testpunkte im Skript eng beieinanderzuhalten.

6.2 Formatierung der Eingabe

So wie man den Titel für die Ausgabeseite setzen kann, lässt sich der Titel auch für die Eingabeseiten setzen und im Laufe der Skriptausführung ändern, ebenso lassen sich auch bei der Eingabe Abschnitte einfügen, um die Eingabeseiten weiter zu gliedern. Zuletzt kann auch noch unterhalb der Überschrift eine Beschreibung des Skriptes ausgegeben werden, um auch den Anwender abzuholen, der nur durch die Skriptauswahl stöbert, und der bislang nur den Dateinamen kennt.

Beispiel 16 – Bug-Planer

```
~SetTitle Bug-Planer
~SetInputTitle Bug-Planer 6-3-1
~SetInputDescription Der Bug-Planer berechnet
    __ für eine Anzahl von Bugs, wie viele Bugs von
    __ jedem Schweregrad eingeplant werden müssen,
    __ damit diese im Verhältnis 6:3:1 liegen.
~SetInputSection Vorhandene Bug-Vorgänge
~Input A2: Anzahl Bugs mit Schweregrad 2
~Input A3: Anzahl Bugs mit Schweregrad 3
~Input A4: Anzahl Bugs mit Schweregrad 4
~SetInputSection Planung
~Input AP: Wie viele Bugs möchtest du einplanen?
~Execute
~Set S2=0
~Set S3=0
~Set S4=0
~Set Sum=0
~Split Indexlist=4;3;2;2;3;2;2;3;2;2|;
~Set anybug = true
~DoWhile $Sum<$AP && $anybug
    ~ForEach idx in Indexlist
        ~If $Sum<$AP && $S$idx<$A$idx
            ~AddTo S$idx+=1
            ~AddTo Sum+=1
    ~Loop
    ~EvalExpression anybug = $S4<$A4 || $S3<$A3 || $S2<$A2
~Loop
```

```
@Ergebnis >>* Eingeplant werden sollen $AP Vorgänge.
>>* Folgende Aufteilung wird vorgeschlagen:
    > $S2 von $A2 Bugs mit Schweregrad 2
    > $S3 von $A3 Bugs mit Schweregrad 3
    > $S4 von $A4 Bugs mit Schweregrad 4
```

Der Bug-Planer aus dem Beispiel oben ist eine schönes Beispiel für ein Berechnungsskript, bei dem Werte abgefragt, und daraus andere Werte berechnet werden. In diesem Fall wird die Anzahl der Bug-Vorgänge für jeden Schweregrad von 2 bis 4 abgefragt, der z.B. wie in Abschnitt 3.2 beschrieben festgelegt werden kann. Zusätzlich wird noch die Anzahl der einzuplanenden Bug-Vorgänge abgefragt und damit dann ein Vorschlag für eine Aufteilung berechnet, die möglichst im Verhältnis $6:3:1$ steht. Eine solche Angleichung geht natürlich nur, wenn von allen Schweregraden ausreichend viele Vorgänge vorhanden sind. Wenn nicht, wird das verplant, was da ist und das Sollverhältnis auf die anderen Schweregrade angewendet.

Für die Eingabeseite wurde hier `~SetInputTitle` eine eigene Überschrift gewählt. Zusätzlich wurde mit `~SetInputDescription` eine Skriptbeschreibung formuliert, die unterhalb der Überschrift ausgegeben wird. Die vier Eingabefelder wurden mit dem Befehl `~SetInputSection` in die beiden Abschnitte *Vorhandene Bug-Vorgänge* und *Planung* unterteilt. Man beachte, dass gleich benannte Eingabeabschnitte an verschiedenen Stellen nicht zusammengeführt.

Nach der vierten Eingabe steht der Befehl `~Execute`, der ebenfalls zur Formatierung verwendet werden kann. Der Befehl bewirkt, dass der Aufbau der Eingabeseite an diesem Punkt abgebrochen, und die Eingabeseite im aktuellen Zustand angezeigt wird. Erst wenn alle Abfragen und Eingaben bis zu diesem Punkt getätigt wurden, läuft das Skript weiter bis zum Ende oder bis zum nächsten `~Execute`-Befehl. Bei Skripten mit sehr vielen Abfragen kann man diesen Befehl nutzen, um die Fragen so aufzuteilen, dass sie jeweils auf einen Bildschirm passen oder man kann erzwingen, dass die von einer Auswahl abhängigen Eingaben zuerst vollständig abgefragt werden, bevor der nachfolgenden Block ausgeführt wird. Im Beispiel oben sorgt der Befehl dafür, dass die Berechnungen in der zweiten Hälfte des Skriptes erst ausgeführt werden, wenn die an die Eingaben gebundenen Variablen mit Werten belegt sind. Ansonsten würden die würden die ersten mit diesen Variablen durchgeführten Vergleiche zu einem Fehler führen.

Die Berechnung selbst funktioniert wie folgt: Zunächst werden Variablen `S2`, `S3` und `S4` für die Zahl der zu verteilenden Aufgaben des jeweiligen Schweregrades auf 0 gesetzt, ebenso eine Variable `Sum` für die Zwischensumme der insgesamt schon verteilten Aufgaben.

Die äußere `~DoWhile`-Schleife wiederholt so lange, wie diese Zwischensumme noch kleiner als die Anzahl der zu verteilenden Aufgaben ist und noch ein Vorgang zum verteilen übrig ist, was über den Befehl `~EvalExpression` getrennt ausgewertet und in der Variablen `anybug` abgespeichert wird.

In der Inneren Schleife wird die Indexsequenz $4,3,2,2,3,2,2,3,2,2$ durchlaufen, was die Index-Variable `idx` nacheinander auf diese Werte setzt. Dies bewirkt, dass die Indices 2, 3 und 4 gleichmäßig im Verhältnis $6:3:1$ durchlaufen werden. Für den jeweils ausgewählten Index wird dann die Anzahl der zu verteilenden Aufgaben S_i um 1 erhöht, sofern noch Aufgaben des jeweiligen Schweregrades übrig sind ($S_i < A_i$) und die Zwischensumme S der schon verteilten Aufgaben noch kleiner ist, als die Anzahl A der Aufgaben, die ver-

teilt werden soll ($S < A$), also `$S$idx<$A$idx` und `$Sum<$AP`. Für jede verteilte Aufgabe wird auch die Zwischensumme um 1 erhöht.

Eine nette Erweiterung dieses Skriptes könnte so aussehen, dass man die Gesamtzahl der zu verplanenden Vorgänge nicht abfragt, sondern prozentual aus der Zahl der vorhandenen Bugs berechnet und dann nach oben oder unten durch einen absoluten Wert begrenzt. Die Eingaben könnten damit auf die des ersten Abschnitts beschränkt werden.

6.3 Hilfezeilen

Einer der großen Vorteile einer interaktiven Anleitung liegt darin, dass man eben nicht wissen muss, welche Schritte man ausführen muss, weil das die Anleitung ja zusammen mit dem Anwender herausarbeitet. Die Informationen, unter welcher Bedingung welcher Schritt erforderlich ist, und was dabei genau zu tun ist, kann alles vom Ersteller in die Struktur und die Ausgaben des Skriptes gepackt werden, so dass der Anwender nur noch richtig auf die gestellten Fragen antworten und die benötigen Eingaben tätigen muss.

Diese Fragen und Eingaben so einfach und verständlich zu halten, so dass auch Anwender mit wenig Erfahrung in der Lage sind, das Skript auszuführen, ist eine Kunst für sich. In manchen Fällen wird das nicht möglich sein, und man wird einfach ein gewisses Systemwissen beim Anwender voraussetzen müssen, in den meisten Fällen wird man jedoch ein oder zwei Zeilen Anleitung auskommen, um zu beschreiben, wo man einen Wert findet oder wie die gestellte Frage oder einzelne Auswahlwerte genau zu interpretieren sind.

Beispiel 17 – Hilfezeilen

```
~Input Bez: Trage die Lizenz für FlowProtocol 2 ein:
~AddHelpLine Du findest die zugeordnete Lizenz auf der
~AddHelpLink https://github.com/maier-san/FlowProtocol2
    __| FlowProtocol2-Seite
~AddHelpText  bei GitHub auf der rechten Seite als
    __ zweiten Punkt unter "About".
?K: Handelt es sich um eine freie Lizenz?
    ~AddHelpLine Suche im Internet nach Details der Lizenz.
    ~AddHelpLine "Freie Lizenz" bedeutet in diesem
        __ Zusammenhang, dass für die Nutzung der Software
        __ auch im kommerziellen Umfeld keine Kosten anfallen.
    #: Ja
    #: Nein
```

Einzelne Hilfezeilen lassen sich für Texteingaben und Auswahlabfragen mit dem Befehl `~AddHelpLine` hinzufügen. Mit `~AddHelpLink` lassen sich, analog wie bei den Ausgaben, auch Links einbauen, so dass man auf Wiki-Seiten oder andere *FlowProtocol 2*-Skripte verweisen kann. Danach kann man mit `~AddHelpText` wieder normalen Text an die letzte Zeile anhängen.

7 Programmierung

Auch wenn es ursprünglich nicht die Absicht war, aus *FlowProtocol 2* eine Programmierumgebung zu machen, gab es doch immer wieder Anforderungen, die mit den grundlegenden Befehlen einer üblichen Programmiersprache gut umsetzbar gewesen wären. Der Wunsch, die Lösung dieser Anforderungen auch zentral, und für alle Mitarbeiter einfach über den Browser bereitzustellen, führte schließlich dazu, die wichtigsten Programmierbefehle auch in *FlowProtocol 2* einzubauen.

7.1 Sprünge

Wir starten das Thema Programmierung mit einem Befehl, der in den höheren Programmiersprachen eher selten zum Einsatz kommt, dem `~Goto`-Befehl. Dieser ermöglicht es einen Sprungmarke anzuspringen, und die Skriptausführung dort fortzusetzen. Die Sprungmarke wird wiederum mit dem Befehl `~JumpMark` gesetzt. Eine Sprungmarke kann auch ganz normal als nächste Zeile in der Ausführung durchlaufen werden. Ihr Durchlaufen selbst hat keinen Effekt.

Der `~GoTo`-Befehl ist für *FlowProtocol 2* dahingehend interessant, da er den eng mit dem namensgebenden Ursprung der Anwendung zusammenhängt und sehr direkt die Abbildung von Flussdiagrammen ermöglicht. Im Gegensatz zur Verschachtelung von Abfragen kann man damit weitere Muster gut umsetzen, ohne dabei Skriptcode wiederholen zu müssen, wie etwa im folgenden Beispiel:

Beispiel 18 – Sprünge

```
?Q1: Um welche Art von Entwicklung handelt es sich?
    #F: Neues Framework
        ~GoTo Framework
    #I: Individuelle Einzelentwicklung
        ~GoTo Einzelentwicklungen
    #V: Standard-Entwicklung auf Framework-Basis
        ~GoTo Alle Entwicklungen
~JumpMark Framework
>> Fragen für Framework-Entwicklungen
~JumpMark Einzelentwicklungen
>> Fragen für Einzelentwicklungen
~JumpMark Alle Entwicklungen
>> Fragen für alle Entwicklungen
```

Zu Beginn wird nach der Kategorisierung einer Entwicklung gefragt, die entweder eine Framework-Entwicklung ist, eine individuelle Einzelentwicklung oder eine Standardentwicklung auf Basis des vorhandenen Frameworks ist. In Abhängigkeit der Antwort werden entweder alle drei Blöcke, nur die letzten neiden Blöcke oder nur der letzte Block durchlaufen.

Man kann Sprungmarken auch einfach dazu verwenden, um tief verschachtelte Abfragen in übersichtlicher Art und Weise als Folgen anzuordnen.

7.2 Berechnungen

Aus Abschnitt 5.2 kennen wir schon den `~AddTo`-Befehl, mit dem man einen Wert zu einer Variablen dazuzählen kann. Später wurde der `~Calculate`-Befehl ergänzt, der eine Berechnung mit zwei Argumenten und einem der Operatoren `+`, `-`, `*`, `/`, und `%` (modulo) durchführen konnte. Umfangreichere Ausdrücke mussten dementsprechend in mehreren Zeilen zerlegt und mit entsprechend vielen Hilfsvariablen berechnet werden.

Mit dem Befehl `~CalculateExpression` lassen sich inzwischen auch längere Ausdrücke auf Basis der Operatoren `+`, `-`, `*`, `/`, `%` und `^` und den Funktionen `sqrt`, `sin`, `cos`, `tan`, `exp` und `ln` und einer beliebigen Klammerung in einer Zeile berechnen. Die Berechnungen erfolgen nummerisch und die Ergebnisse werden in Dezimal- oder ggf. auch in Exponentialschreibweise dargestellt.

Beispiel 19 – Berechnungen

```
~CalculateExpression U = 72 % 7
~CalculateExpression V = (-2,5 + 1,57)*(1,3 - 0,4)/18
~CalculateExpression W = 2^3*16^(1/2)+7
~CalculateExpression X = exp(sin(1/4))
~Round Z=$X|3
@Ausgabe >> U = $U
    >> V = $V
    >> W = $W
    >> X = $X
    >> Z = $Z
```

In die letzten Zeile vor der Ausgabe wird der Wert der Variablen `X` auf drei Stellen hinter dem Komma gerundet und das Ergebnis der Variablen `Z` zugewiesen.

7.3 Schleifen

Die große Stärke von Computern liegt darin, die gleichen Dinge in leichter Variation oft zu wiederholen, ohne sich zu langweilen und ohne in der Ausführung nachlässig zu werden. Mit Hilfe von Schleifen lassen sich solche Mehrfachausführungen einfach implementieren.

Die bekannten höheren Programmiersprachen unterscheiden drei prinzipielle Arten von Schleifen: Die *For-Schleife* wird mit einer Zählervariablen eingeleitet, die mit einem Startwert beginnt, diesen bei jedem Durchlauf um eine Schrittlänge hoch zählt, und beendet wird, wenn ein Zielwert erreicht oder eine Abbruchbedingung erfüllt wird. Die *Do-While-Schleife* läuft dagegen so lange durch, bis eine Abbruchbedingung erfüllt ist. Schließlich gibt es noch die *For-Each-Schleife* bei der eine vorgegebene Menge von Elementen durchlaufen wird. Eine Indexvariable nimmt dabei nacheinander den Wert dieser Elemente an.

In *FlowProtocol 2* gibt es die Beiden Befehle `~DoWhile` und `~ForEach` für die Einleitung von Do-While-Schleifen und For-Each-Schleifen. Reine For-Schleifen lassen sich sehr direkt auch als Do-While-Schleife abbilden, deshalb gibt es für diese keinen eigenen Befehl. Das Schleifenende, also die Stelle, an der ggf. ein weiterer Durchlauf gestartet wird,

wird in beiden Fällen mit dem ~Loop-Befehl gekennzeichnet, der die gleiche Einrückung wie der Beginn der Schleife haben muss. Auf diese Weise lassen sich mehrere Schleifen ineinander verschachteln. Zusätzlich gibt es noch den ~ExitLoop-Befehl, mit dem die aktuell ausgeführte Schleife unmittelbar verlassen wird.

In Beispiel 6.2 hatten beide Schleifenvarianten schon einen kleinen Gastauftritt, aber um die Funktionsweise nochmal genauer anzuschauen ist die folgende Primzahlberechnung gut geeignet:

Beispiel 20 – Primzahlen

```
~Set n=2
~Set pidx=0
~DoWhile $n<20
    ~Set prim=Ja
    ~ForEach p in PZ
        ~CalculateExpression r = $n % $p
        ~If $r==0
            ~Set prim=nein
            ~ExitLoop
    ~Loop
    ~If $prim==Ja
        ~AddTo pidx+=1
        >> P($pidx) = $n
        ~Set PZ($pidx)=$n
    ~AddTo n+=1
~Loop
```

Die äußeren Do-While-Schleife wird durchlaufen, solange die Variable n kleiner als 20 ist. Der Wert von n wird dabei jedes Mal vor Schleifenende um eins erhöht, was vom Prinzip her einer For-Schleife entspricht. Gestartet wird mit n=2. Auf die Formulierung von Bedingungen gehen wir in Abschnitt 7.4 näher ein.

Die innere For-Each-Schleife durchläuft eine Menge PZ unter Verwendung der Variablen p. PZ wird hierbei als eindimensionales Feld interpretiert, das die fortlaufend nummerierten Variablen PZ(1), PZ(2), PZ(3), usw. enthält. Beim ersten Durchlauf der äußeren Schleife ist die Variable PZ(1) noch nicht definiert und die Menge PZ daher leer, d.h. die innere Schleife wird nicht durchlaufen.

Für jede Zahl *n*, für die nach Durchlauf der inneren Schleife die Variable prim noch auf Ja steht, wird die Menge der gefundenen Primzahlen PZ um *n* erweitert, weil *n* in diesem Fall durch keine der bisher gefundenen Primzahlen teilbar ist. Ist *n* dagegen durch eine dieser Primzahlen *p* teilbar, also wenn der Rest bei Division durch *p* null ergibt, dann wird die Variable prim auf Nein gesetzt und die innere Schleife verlassen, weil auch die folgenden Durchläufe das Ergebnis nicht mehr verändern würden.

Das Programm geht also recht effizient vor und versucht soweit es geht, unnötige Schleifendurchläufe zu vermeiden. Dies ist dahingehend kein Fehler, da die Ausführung von *FlowProtocol 2*-Skripten, verglichen mit einer kompilierbaren Programmiersprache wie C#, sehr langsam ist.

Die Programmierung von Schleifen bringt immer auch mit sich, dass man versehentlich eine Endlosschleife durchläuft, z.B. wenn man das Hochzählen der Variable vergisst und so die Abbruchbedingung nie erreicht wird. Um dies und auch Endlosrekursionen frühzeitig zu erkennen, werden alle Durchläufe von Schleifen im Hintergrund mitgezählt, und die Ausführung mit dem Fehler *Maximale Anzahl Schleifendurchläufe erreicht* beendet, wenn der intern gesetzte Stop-Zähler für Schleifen oder die Gesamtzahl an durchlaufenen Befehlen erreicht wird. Mit dem Befehl `~SetStopCounter` lassen sich diese beiden Werte hochsetzen, wenn man abschätzen kann, dass eine reguläre Skriptanwendung so viele Durchläufe und Befehlsausführungen mit sich bringt. Raffinierte Berechnungen vielen Iterationen und Rekursionen sind zwar möglich, liegen aber eher am Rand der Wohlfühlzone von *FlowProtocol 2*.

Ein häufigerer Anwendungsfall, insbesondere wenn man Skripte schreibt, die Programmcode generieren, ist die Abfrage einer Sequenz von Elementen, z.B. die Variablennamen für die Parameter einer Funktion. Im Beispiel unten wird über eine Schleife wiederholt abgefragt, ob es noch einen weiteren Parameter gibt, und wenn ja, nach dem dazugehörenden Variablenname gefragt.

Beispiel 21 – Abfrageschleife

```
?V1: Hat die Funktion Parameter?
    #j: Ja
        ~Set i=1
        ~DoWhile $V$i==j
            ~Input B$i: Variable für den $i. Paramter:
            ~Set Vars($i)=$B$i
            ~AddTo i+=1
            ?V$i: Hat die Funktion einen $i. Parameter?
                #j: Ja
                #n: Nein
            ~Execute
        ~Loop
        ~ForEach v in Vars
            @Variablen >> $v
        ~Loop
    #n: Nein
```

Die Antworten und Eingaben werden in fortlaufend nummerierten Schlüsseln verwaltet und die Ausführung nach jedem Schleifendurchlauf angehalten.

7.4 If-Abfragen und Bedingungen

Zu den wichtigsten Steuermechanismen innerhalb eines Programms gehört ganz klar die Möglichkeit, Bedienungen auszuwerten und Fallunterscheidungen abzubilden. Der dazugehörende Befand lautet `~If` und kann kombiniert werden mit den befehlen `~ElseIf` und `~Else`. Er ist so elementar, dass wir ihn schon in zahlreichen Beispielen verwendet haben, ohne näher darauf einzugehen.

`~If` führt den darunter eingerückten Skriptcode genau dann aus, wenn die hinter dem `~If`-Befehl stehende Bedingung erfüllt ist. Mit `~ElseIf` kann über eine weitere Bedingung ein weiterer Codeblock angeschlossen werden, die aber nur dann ausgewertet, bzw. ausgeführt werden, wenn keine der davorstehenden Bedingungen erfüllt ist. Auf diese Weise können beliebig Viele ElseIf-Bedingungen aneinander gereiht werden. Abschließend kann mit `~Else` noch eine Codeblock angehängt werden, der nur dann ausgeführt, wenn keiner der davorstehenden Bedingungen erfüllt wurde.

Beispiel 22 – Bewertungsschema

```
~Input P: Wie hoch ist die Punktezahl (0-20)
~Execute
~If $P>=18
    ~Set Note=sehr gut
~ElseIf $P>=14
    ~Set Note=gut
~ElseIf $P>=10
    ~Set Note=befriedigend
~ElseIf $P>=5
    ~Set Note=ausreichend
~Else
    ~Set Note=ungenügend
>> Ergebnis: $P Punkte (Note $Note)
```

Ein klassischer Anwendungsfall für mehrstufige If-Abfragen ist die Anwendung eines Bewertungsschemas bei der für einen vorgegebenen Wert das dazu passende Intervall gesucht werden muss, mit dem diesem Wert dann ein Zielwert zugeordnet wird. Im Beispiel oben wird so einem Punktewert von 0 bis 20 eine Note in Textform zugeordnet.

Die Bedingungen im Beispiel sind einfache Größer-gleich-Vergleiche, aber es können auch komplexere Ausdrücke formuliert werden. Die Bedingung muss dabei in der disjunktiven Normalform angegeben werden, also als Oder-Verknüpfung (`||`) von Und-Verknüpfungen (`&&`), wobei keine Klammerung notwendig ist. Als Literale sind die Konstanten `1` (wahr), `true` (wahr), `0` und `false` (falsch) verwendbar, sowie für Zeichenketten *s* und *t*, Zahlen *x* und *y* und Variablen *v* die folgenden Ausdrücke zulässig: `s==t` (*s* ist gleich *t*), `s!=t` (*s* ist ungleich *t*), `x<>y` (*x* ist ungleich *y*), `x<y` (*x* ist kleiner als *y*), `x<=y` (*x* ist kleiner oder gleich *y*), `x>y` (*x* ist größer als *y*), `x>=y` (*x* ist größer oder gleich *y*), `s~t` (*s* enthält *t*), `s!~t` (*s* enthält *t* nicht), `?$v` (*v* ist gesetzt), `!?$v` (*v* ist nicht gesetzt).

7.5 Funktionen

Viele spannende und mächtige Algorithmen gründen auf Funktionen, die sich rekursiv selbst aufrufen. Es gibt Programmiersprachen wie Scheme, die einen dazu verleiten, so gut wie jede Problemstellung mittels Rekursion zu lösen. Auch *FlowProtocol 2* erlaubt die Definition von Funktionen und rekursive Aufrufe, obgleich wie schon im früheren Abschnitten erwähnt, dort nicht der Schwerpunkt dieser Skriptsprache liegt.

Das nachfolgende Beispiel zeigt den Klassiker unter den rekursiv lösbaren Aufgaben, nämlich die Türme von Hanoi. Dabei geht es darum, einen Turm von mehreren absteigend großen Schreiben von einer Stange unter Zuhilfenahme einer zweiten Stange auf

eine dritte Stange umzustapeln, wobei jeweils nur eine Scheibe bewegt werden darf und niemals eine größere Scheibe auf einer kleineren Scheibe abgelegt werden darf.

Beispiel 23 – Türme von Hanoi

```
~Set MoveFromS=1
~Set MoveToS=3
~Set MoveCountS=4
~GoSub Move; BaseKey=S
~End

// Bewegt n Scheiben von A nach B
// MoveFrom$BaseKey:  Ausgangsturm A
// MoveTo$BaseKey:    Zielturm B
// MoveCount$BaseKey: Anzahl Scheiben n
~DefineSub Move
    ~If $MoveCount$BaseKey>0
        ~Set Q1$BaseKey=$MoveFrom$BaseKey
        ~Set Q2$BaseKey=$MoveTo$BaseKey
        ~Set Q3$BaseKey=123
        ~Replace Q3$BaseKey=$Q3$BaseKey|$Q1$BaseKey->
        ~Replace Q3$BaseKey=$Q3$BaseKey|$Q2$BaseKey->
        ~Set MC$BaseKey=$MoveCount$BaseKey
        ~AddTo MC$BaseKey+=-1

        ~Set MoveFrom$BaseKeyA=$Q1$BaseKey
        ~Set MoveTo$BaseKeyA=$Q3$BaseKey
        ~Set MoveCount$BaseKeyA=$MC$BaseKey
        ~GoSub Move; BaseKey=$BaseKeyA

        @Lösung >> Von $Q1$BaseKey nach $Q2$BaseKey
            __ (BaseKey=$BaseKey)

        ~Set MoveFrom$BaseKeyB=$Q3$BaseKey
        ~Set MoveTo$BaseKeyB=$Q2$BaseKey
        ~Set MoveCount$BaseKeyB=$MC$BaseKey
        ~GoSub Move; BaseKey=$BaseKeyB
~Return
```

Der Lösungsansatz besteht darin, das Umstapeln eines Turms mit n Scheiben von Stange a nach Stange b in drei Schritte zu zerlegen: Zunächst stapelt man den oberen Teil des Turms aus $n-1$ Schreiben um auf die freie Stange c, bewegt dann die $n$te Schreibe auf die Zielstange b und stapelt dann wieder denselben Turm von Stange c auf Stange b oben auf diese Scheibe drauf. Auf diese Weise hat man das Problem auf das Verschieben eines kleineren Turms zurückgeführt und kann das wiederholt anwenden, bis man schließlich bei einem Turm mit 0 Scheiben angekommen ist, bei dem nichts mehr zu tun ist.

Die Definition der Funktionen erfolgt am Ende des Skriptes, das zuvor mit dem Befehl `~End` beendet werden muss. Jede Funktion wird mit dem Befehl `~DefineSub` eingeleitet

und mit dem Befehl `~Return` beendet. Der dazwischen liegende, eingerückte Codeblock wird bei jedem Aufruf durchlaufen. Ein Aufruf wird mit mir im Befehl `~GoSub` in die Wege geleitet, bei dem auch der `BaseKey`-Wert übergeben werden kann. Nach dem Aufruf wird das Skript in der nächsten Zeile fortgesetzt.

Da es in *FlowProtocol 2* keine lokalen Variablenbereiche gibt, muss man die Variablen, die für einen bestimmten Funktionsaufruf erhalten bleiben sollen, explizit von den anderen Variablen abgrenzen. Dafür kann über die Sondervariable `BaseKey` eine Zahl oder Zeichenfolge übergeben werden, die für jeden Funktionsaufruf getrennt verwaltet wird und die in den innerhalb der Funktion genutzten Variablen als Teil der Bezeichnung eingebaut werden kann, um auch diese zu trennen. In gleicherweise können auch die Funktionsargumente vor dem Funktionsaufruf als ganz normale Variablen definiert werden.

Der Umgang mit rekursiven Funktionen ist in *FlowProtocol 2* zugegebenermaßen etwas gewöhnungsbedürftig, aber er lässt sich mit einigen Debug-Ausgaben und etwas Übung doch gut hinbekommen. Analog zu den Variablen lassen sich auch die Schlüssel für Eingaben mithilfe der `BaseKey`-Variablen definieren, so das in jeder Rekursion auch Benutzerinteraktionen stattfinden können. Auf diese Weise kann zum Beispiel ein Sortieralgorithmus wie Mergesort implementiert werden, der den einzelnen Vergleich zweier Elemente an den Anwender weitergibt. Damit kann etwa die Sortierung von Projekten oder Aufgaben nach Priorität auf Einzelvergleiche heruntergebrochen werden, die sich oft mit Bauchgefühl ganz gut festlegen lassen.

Natürlich kann man Funktionen auch ohne Rekursion nutzen. In diesem Fall benötigt man keine `BaseKey`-Variable und kann sowohl die Funktionsargumente, als auch die Rückgabewerte ganz normal in Variablen übergeben.

8 Verarbeitung von Texten

Der Zweck eines Skriptes ist es, eine Ausgabe zu erzeugen und der Zweck von Texteingaben liegt meist darin, diese Texte in irgendeiner Form in der Ausgabe einzuarbeiten. Sofern die Eingabe unverändert in der Ausgabe erscheinen soll, reicht es, einfach die entsprechende Variable in der Ausgabe zu verwenden. In manchen Fällen möchte man jedoch einen Text in einer bestimmten Form abwandeln oder entsprechend einer vorgegebenen Struktur zerlegen. Mit den in diesem Abschnitt beschriebenen Befehlen lassen sich die meisten Textveränderungen hinbekommen.

8.1 Texte aufteilen und neu kombinieren

Wie wir schon in Beispiel 6.2 gesehen haben, Kann man mit Schleifen sehr gut Listen durchlaufen und eine Menge von Elementen bearbeiten. In diesem Beispiel wurde auch schon der `~Split`-Befehl verwendet, um aus einer kommagetrennten Aufzählung eine solche indizierte Liste zu erstellen, die mit dem `~ForEach`-Befehl durchlaufen werden kann. In der gleichen Weise kann auch eine ganze Aufzählung durch eine einzelne `~Input`-Abfrage als Benutzereingabe entgegengenommen, und in eine Liste umgewandelt werden.

Im nachfolgenden Beispiel wird sogar eine verschachtelte Liste, also eine Liste von Listen als Eingabe abgefragt und verarbeitet. Der `~Split`-Befehl funktioniert so, dass er den

übergebenen Text anhand des angegebenen Trennzeichens aufteilt und in einer indizierten Liste speichert.

Beispiel 24 – Kombinator

```
~Input C: Kombinierbare Eigenschaften
    __ (z.B.: A,B,C : X,Y,Z)
~Split A=$C|:
~Set aidx=1
~Set QAnz(1)=1
~Set Q(1)(1)=#Start#
~ForEach a in A
    ~Set aprev=$aidx
    ~AddTo aidx+=1
    ~Split W=$a|,
    ~Set widx=0
    ~ForEach q in Q($aprev)
        ~ForEach w in W
            ~AddTo widx+=1
            ~Replace Q($aidx)($widx)=$q - $w|#Start# -->
        ~Loop
    ~Loop
~Loop
~ForEach q in Q($aidx)
    @Kombinationen >> $q
~Loop
```

Der *Kombinator* nimmt eine Liste von kombinierbaren Eigenschaften entgegen, die jeweils verschiedene Werte annehmen können, z.B *Farbe gleich blau, Farbe gleich grün* und *Material gleich Holz, Material gleich Metall*. Die Werte jede Eigenschaft werden dabei durch Kommata getrennt, die einzelnen Wertlisten wiederum durch einen Doppelpunkt. Bei Ausführung des Skriptes werden alle Kombinationen aus den Werten jeder Eigenschaft gebildet und aufgelistet. In unserem Beispiel ergibt das vier Kombinationen, angefangen mit *Farbe gleich blau und Material gleich Holz.*

Der Umgang mit solchen kombinatorischen Auflistungen ist in der Softwareentwicklung wichtiger, als man denkt und der größte Teil aller Fehler bei der Implementierung von Programmfunktionalität geht darauf zurück, dass Fälle übersehen oder vergessen werden. Dies passiert oft schon beim Entwurf der Userstory, in der hauptsächlich die Geradeaus-Fälle betrachtet werden und Fälle, die keinem normalen Anwendungsfall entsprechen, oft außer Acht gelassen werden. Wenn solche Fälle jedoch formal möglich sind, muss auch die Software damit in irgendeiner Form umgehen können, und damit sie das kann müssen auch diese Fälle betrachtet und behandelt werden.

Gerade bei der Implementierung von Formeln ist es ungeheuer wichtig, dass der komplett zulässige Definitionsbereich bei Implementierung und Test abgedeckt werden, und dass systematisch sichergestellt wird, dass alle nicht zulässigen Eingaben und Fälle sauber und für den Benutzer transparent durch die Programmoberfläche abgefangen werden. Eine solche Systematik bekommt man am besten dadurch hin dass man zunächst die vorhandenen Fälle in ihre Eigenschaften zerlegt und diese dann mit Hilfe so eines Werkzeugs

zu allen Fallunterscheidungen kombiniert, was allerdings recht umfangreich werden kann. Entsprechend empfiehlt es sich, den Kombinator so zu erweitern, dass sich zumindest die sich gegenseitig ausschließenden Kombinationen angeben und entfernen lassen.

8.2 Ersetzungen und Zufallsgenerierung

Den Ersetzungsbefehl `~Replace` haben wir schon in vielen Skripten gesehen, etwa in Beispiel 5.3. Die Ersetzung von Texten in anderen Texten ist für viele Zwecke einsetzbar. Im nachfolgende Beispiel wird die Textersetzung dazu verwendet um systematisch einen Text aufzubauen. Hierbei kommt auch der Befehl `~Random` zum Einsatz, mit dem eine Zufallszahl in einem Zahlenbereich generiert werden kann.

Beispiel 25 – Passwortgenerator

```
~Set L=12
~Random AnzZiffer=1..2
~Random AnzGross=1..2
~Random AnzSonder=1..2
~Split SZ=33,35,36,37,38,42,43,47,64|,
~Set M="
~Set i=0
~DoWhile $i<$L
    ~Random r=0..$i
    ~AddTo i+=1
    ~If $r==0
        ~Set M="+$i$M
    ~Else
        ~Replace M=$M|"+$r"->"+$r"+$i"
~Loop
~Set i=0
~DoWhile $i<$L
    ~AddTo i+=1
    ~If $AnzZiffer>0
        ~Random z=48..57
        ~AddTo AnzZiffer+=-1
    ~ElseIf $AnzGross>0
        ~Random z=65..90
        ~AddTo AnzGross+=-1
    ~ElseIf $AnzSonder>0
        ~Random s=1..9
        ~Set z=$SZ($s)
        ~AddTo AnzSonder+=-1
    ~Else
        ~Random z=97..122
    ~Replace M=$M|"+$i"->"$Chr($z)"
~Loop
~Replace PW=$M|"->
@Ausgabe >>* Passwortvorschlag
```

```
>|$PW
```

Das Beispiel zeigt einen Passwortgenerator, der für eine vorgegebene Länge ein Passwort aus zufälligen Zeichen erzeugt. Die von den meisten Passwortrichtlinien geforderten Zusatzbedingungen, dass das Passwort auch Ziffern, Großbuchstaben und Sonderzeichen enthalten muss, sind ebenfalls berücksichtigt. Da diese Zeichen zum Eintippen oftmals lästig sind, kann ihre Anzahl hier beschränkt werden. Ebenso kann die Menge der Sonderzeichen festgelegt werden, die zum Einsatz kommen soll.

Im ersten Teil des Skriptes wird eine Zeichenkette aus den einzelnen Zahlen 1 bis *L* für die gewünschte Länge *L* des Passworts und einigen Trennzeichen erstellt. Diese wird bereits bei der Erstellung permutiert, indem das Einfügen an einer beliebigen Stelle mit dem Ersetzungsbefehl durchgeführt wird. Im zweiten Teil wird für jede Zahl von 1 bis *L* ein Zeichen aus den verschiedenen Zeichengruppe per Zufallsgenerator bestimmt und die Zahl durch das entsprechende Zeichen ersetzt. Hierbei wird der `$Chr`-Befehl verwendet der das Zeichen mit dem dazu gehörenden Dezimalcode erzeugt. Nacheinander wird so die geforderte Anzahl an Ziffern, Großbuchstaben und Sonderzeichen eingefügt. Die restlichen Zeichen werden mit Kleinbuchstaben aufgefüllt. Am Ende werden die Trennzeichen entfernt, sodass nur noch die Zeichen übrig bleiben, die das Passwort bilden.

8.3 Reguläre Ausdrücke und Datenabfragen

Dieser Abschnitt demonstriert zwei der wohl mächtigsten Befehle im Besteckkasten von *FlowProtocol 2*: reguläre Ausdrücke und Datenabfragen. Reguläre Ausdrücke dienen dazu, Textmuster zu beschreiben, so dass man diese innerhalb von Texten auffinden kann, z.B *ein Großbuchstabe gefolgt von zwei Ziffern* was mit `[A-Z]\d{2}` beschrieben wird. Zur Formulierung von regulären Ausdrücken gibt es zahlreiche Hilfsseiten und Online-Tools und natürlich Auch die verschiedenen Sprachmodelle, denen man den gewünschten Ausdruck umgangssprachlich beschreiben kann. Ein von mir gerne genutztes Tool zur Überprüfung regulärer Ausdrücke ist *https://regex101.com*.

Datenabfragen sind Zugriffe auf Dateien, die im gleichen Grundordner liegen, wie die Skripte, die sich über das Menü von *FlowProtocol 2* auswählen lassen. Es ist zwar nicht möglich, schreibend auf diese Dateien zuzugreifen, aber lesend. Entsprechend gibt es mit `~ForEachLine` einen noch nicht genannten Schleifen-Befehl, mit dem man über die Zeilen einer beliebigen Datei iterieren kann. Kombiniert man nun das Einlesen einer Datei mit den Möglichkeiten der regulären Ausdrücke, so kann man mehr oder weniger jedes beliebige textbasierte Dateiformat verarbeiten.

Das nachfolgende Beispiel ist ein einfaches Werkzeug zur Anwendung einer Terminologie auf einen Eingabetext. Die Terminologie für einen bestimmten Kontext beschreibt die dort relevanten Begriffe und gibt an, welche der jeweils möglichen Benennungen innerhalb des Kontextes zulässig sind und welche nicht. In der Regel einigt man sich für jeden Begriff auf eine zulässige Benennung und schließt mehrere andere aus. So entsteht mit der Zeit eine Terminologiedatenbank aus Begriffen, Benennungen, Definitionen und Ausschlussbegründungen, die in einem sehr einfachen Fall so aussehen könnte:

Beispiel 26 – Terminologieliste.txt

```
1    Passwort    ja    Zeichenfolge, deren Kenntnis Zugang...
```

```
1    Kennwort    nein    Weniger üblich.
2    Schaltfläche    ja    Steuerelement, das bei Klick eine...
2    Button    nein    Anglizismus.
2    Knopf    nein    Wird eher mit Geräten assoziiert.
```

Jede Zeile hat vier Einträge, die durch Tabulatoren getrennt sind. Im ersten wird die ID des Begriffes angegeben und da ein Begriff mehrere Benennungen haben kann, können auch mehrere Zeilen zur selben ID vorhanden sein. Die zweite Spalte gibt eine Benennung an. Die dritte Spalte enthält entweder den Inhalt *ja* oder *nein* und gibt an, ob die jeweilige Benennung zulässig ist oder nicht. Im ersten Fall gibt die vierte Spalte die Definition des Begriffes an, im zweiten Fall beschreibt sie die Begründung, warum die entsprechende Benennung nicht verwendet werden soll. Für die Verarbeitung ist es wichtig, dass die zulässige Benennung immer in der ersten Zeile für jede ID steht und dass alle Zeilen zur selben ID aufeinander folgen.

Der Terminologieprüfer ist das dazu gehörende Terminologie-Tool und sieht so aus:

Beispiel 27 – Terminologieprüfer

```
~Input T:Text
@Eingabetext >>_ $T
~ForEachLine z in 24 Terminologieliste.txt
    ~RegExMatch ti=$z|([0-9]*)\t([^\t]*)\t(ja|nein)\t(.*)
    ~If $ti(0)
        ~If $ti(3)==ja
            ~Set EmpfBen=$ti(2)
            ~Set DefBegr=$ti(4)
        ~If $T~$ti(2) && $ti(3)==ja
            @Erlaubte Benennungen >> $ti(2)
                > ID: $ti(1)
                > Definition: $ti(4)
        ~If $T~$ti(2) && $ti(3)==nein
            @Nicht erlaubte Benennungen >> $ti(2)
                > ID: $ti(1)
                > Empfohlen: $EmpfBen
                > Definition: $DefBegr
                > Begründung: $ti(4)
~Loop
```

Es durchläuft alle Benennungen aus der Terminologiedatenbank und prüft, ob diese im eingegebenen Text vorhanden sind. Je nachdem ob es sich um eine zulässige Benennung handelt oder nicht, wird diese dann in einem entsprechenden Abschnitt aufgelistet. Bei einer nicht zulässigen Benennung wird die zum entsprechenden Begriff empfohlene Benennung samt Definition ausgegeben, ebenso die Begründung warum die Benennung nicht verwendet werden soll. Für die Eingabe *nach Eingabe des Passworts muss der Button gedrückt werden* wird *Passwort* als zulässige Benennung bestätigt und *Button* mit Hinweis auf *Schaltfläche* als nicht zulässige Benennung abgelehnt.

Der Befehl `~RegExMatch` liefert in der Ergebnisvariablen `ti` eine Liste zurück. Das nullte Element gibt an, ob für den Ausdruck ein Treffer gefunden wurde, die nachfolgenden Ele-

mente enthalten die Teiltexte des gefundenen Ausdrucks, die über die runden Klammern im Ausdruck als Gruppen angegeben wurden.

Das Skript bietet noch viel Spielraum für Erweiterungen, beispielsweise könnten alle im Text vorhandenen großgeschriebenen Wörter, die noch nicht gefunden wurden als Vorschläge für die Aufnahme in die Terminologie aufgelistet werden, direkt mit einem Link zu einem Terminologieerfassungskript, das die benötigten Informationen abfragt und daraus die entsprechenden Einträge für die Terminologiedatenbank erstellt.

8.4 Parametrisierte Links generieren

Der große Vorteil von Webanwendungen liegt darin, dass man innerhalb der Programmoberfläche überall Links auf andere Programmbereiche oder sogar andere Anwendungen einbauen kann. Diese lassen sich dann bequem in neuen Tabs öffnen, um einen zwischengelagerten Arbeitsschritt auszuführen und danach kann man zu seiner ursprünglichen Aufgabe zurückkehren. Umgekehrt kann eine Webanwendung wiederum über einen parametrisierten Link aufgerufen werden, dem man einzelne Begriffe oder auch kleinere Texte als Argumente übergeben kann.

Wie man Links in die Ausgabe einbauen kann, haben wir in Abschnitt 4.3 schon gesehen. Um jedoch beliebigen Text als Parameter in einem Link einzufügen, muss man diesen in einer speziellen Form kodieren. Hierfür gibt es in *FlowProtocol 2* den Befehl `~UrlEncode`. Das nachfolgende Beispiel zeigt die Verwendung von URL-Parametern am Beispiel eines Suchbegriffs, den man als Parameter an eine Suchmaschine übergeben kann, hier am Beispiel von Google und Bing.

Beispiel 28 – Suchlink-E-Mail

```
~Input S:Suchbegriff
~UrlEncode uS=$S
@Suchanfragen >>
    ~AddLink https://www.google.com/search?q=$uS |
    __Suche "$S" bei Google
>>
    ~AddLink https://www.bing.com/search?q=$uS |
    __ Suche "$S" bei Bing
~UrlEncode usub=Suchlinks zu "$S"
~UrlEncode ubody=Hallo,$CRLF
    __hier sind zwei Suchlinks zu "$S":$CRLF
    __https://www.google.com/search?q=$uS$CRLF
    __https://www.bing.com/search?q=$uS$CRLF
    __MfG
>> Suchanfragen
    ~AddLink mailto:a@bc.de?subject=$usub&body=$ubody |
    __ per E-Mail verschicken
```

Zuerst wird ein Suchbegriff abgefragt. Dieser wird mittels `~UrlEncode` in die Variable `uS` in codierter Form abgelegt. Für den mailto-Link werden nacheinander Betreff (*subject*)

und Inhalt (*body*) der E-Mail in gleicher Weise in die beiden Variablen `usub` und `ubody` codiert und am Ende in den Link eingebaut.

Das Beispiel zeigt zum einen den Einsatz von Zeilenumbrüche die man mit dem Befehl `$CRLF` erzeugen kann, was die Zeichenkombination *Carriage Return* + *Line Feed* ausgibt, und zum anderen, dass auch kodierte Texte wie die Suchlinks wieder Bestandteil von Codierungen sein können.

Analog zu *mailto* verfügen auch manche Ticketsysteme über Adressen, mit denen sich Vorgänge über parametrisierte Aufrufe vollständig vorbereiten lassen, so dass alle relevanten Felder ausgefüllt sind. In gleicher Weise können auch spezialisierte Suchen im selben Ticketsystem in Form von Links bereitgestellt werden, so dass direkt innerhalb der richtigen Projekt- oder Aufgabenkategorie gesucht wird. Ein klassisches Beispiel ist die Einrichtung einer Metasuche die für einen eingegebenen Suchbegriff ähnlich wie das Beispiel oben verschiedene Links in den verschiedenen Firmensystemen wie Wiki, Programmdokumentation und dem Ticketsystem bereitstellt, und in letzterem noch zusätzlich unter verschiedenen Suchparametern.

Oft müssen solche Tickets oder andere Vorgänge durch Personen außerhalb des Fachbereichs erstellt werden, die sich besonders schwer damit tun alle Eigenschaften in der für die Bearbeitung gewünschten Art und Weise zu setzen. Wenn hier mit einem Skript soweit unterstützt werden kann, dass am Ende die Erstellung des Vorgangs direkt über einen Link möglich ist, hat man viel gewonnen. Idealerweise kombiniert man so ein Erstellungsskript mit einem Expertensystem (siehe Abschnitt 3.3) und der angeleiteten Suche nach eventuell schon vorhandenen Tickets zum Thema.

Eine weitere Anwendungsmöglichkeit für parametrisierte Links sind natürlich die Aufrufe anderer *FlowProtocol 2*-Skripte. Wie schon beschrieben, müssen für komplexe Aufgaben oftmals Werte und Informationen abgefragt werden, die sich nicht so ohne weiteres herausfinden lassen, und für deren Bestimmung eigene Anleitungen hilfreich wären. Diese könnten dann ebenfalls als Skripte bereitgestellt und über die Hilfszeilen der entsprechenden Felder mit Parametern verlinkt werden.

9 Systementwicklung

In diesem Abschnitt kommen wir endlich an die Kerntätigkeit der professionellen Softwareentwicklung, also das Schreiben von Programmcode, und auch wenn man den Eindruck bekommt, dass hier die KI-Systeme, insbesondere die LLM-Chatbots zukünftig alles dominieren werden, gibt es trotzdem noch Fälle, in denen das nicht so bald der Fall sein wird.

9.1 Zielsetzung der Systementwicklung

Die Überschrift lautet bewusst Systementwicklung da viele Softwarehäuser über Jahre hinweg ein umfangreiches System aufbauen und weiterentwickeln, dass auf einem ebenso umfangreichen und durchdachten Framework aufbaut, das wiederum durchgängig auf einem Designsystem gründet. Jeder Teil der Anwendung sieht damit einheitlich aus, folgt den gleichen Bedienprinzipien sowie der für den Domänenkontext festgelegten Terminologie.

Die Erweiterung des Frameworks für ein solches System ist aufwendig, da all diese Punkte berücksichtigt werden müssen. Die Erweiterung der Anwendung auf Basis des vorhandenen Frameworks ist dagegen recht einfach, oder sollte es zumindest sein, da in diesem Fall keine technischen Probleme mehr gelöst werden müssen, sondern nur noch die richtigen Komponenten in der richtigen Art und Weise zusammengebaut, und nach den Vorgaben des Product Owners konfiguriert werden müssen. Und je besser das Framework ist, umso weniger Spielraum gibt es beim Anwenden und damit auch weniger Fragen, weniger Abweichungen und weniger Fehler.

Wer bis hierher durchgehalten hat, wird zustimmen, dass der Aufbau neuer Programmbereiche auf Basis eines vorhandenen Frameworks geradezu prädestiniert ist für den Einsatz von *FlowProtocol 2*-Skripten und dass diese Methode sowohl von der Effizienz als auch von der Qualität her punkten kann.

Zielsetzung ist ein System aus Anleitungen und Unterstützungssystemen, das alle Komponenten des Frameworks in jeweils allen Phasen der Softwareentwicklung repräsentiert, angefangen bei der Auswahl der Komponenten zur Umsetzung bestimmter Anforderungen, über die Verwendung im Programmcode, bis hin zur Testphase. Alle Abhängigkeiten, die sich zwischen den verschiedenen Entwicklungsphasen ergeben, sollten weitestgehend in den Skripten abgebildet sein und automatisch Berücksichtigung finden.

9.2 Entwicklungsanleitungen

Entwicklungsanleitungen sind, wie der Begriff schon sagt, Anleitungen für die Entwickler. Diese beschreiben, wie Anforderungen mit Hilfe der zur Verfügung stehenden Mittel umgesetzt werden können oder sollen.

Idealerweise gibt es für jede wiederkehrende Entwicklung eine dokumentierte Best Practice, die vorgibt, wie diese umzusetzen ist so dass unabhängig von einzelnen Entwicklern immer dieselbe Lösung herauskommen sollte, die sowohl Qualität, als auch Wartbarkeit garantiert. Die Anleitung für ein wiederkehrendes Entwicklungsmuster ist ein guter Ort um so eine Best Practice direkt zu hinterlegen und aktuell zu halten, denn die Tatsache, das einem eine solche Anleitung die Arbeit einfacher macht, stell praktisch sicher, dass diese auch verwendet wird, und damit auch die Best Practice Anwendung findet.

Auch wenn die eigentliche Arbeit an einer Entwicklung beim Product Owner beginnt, ist es sinnvoll beim Aufbau solcher Anleitungen bei der Entwicklung zu beginnen. Zum einen sind die Entwickler diejenigen, die am wenigsten Berührungsängste mit der Skriptentwicklung haben dürften, zum anderen ist durch das Vorhandensein von Framework-Komponenten schon eine sehr weit fortgeschrittene Systematisierung der wiederkehrenden Entwicklungen vorhanden, die nur noch in Skriptform gebracht werden muss.

Die Vorgehensweise hierfür ist relativ einfach und folgt dem Top-Down-Ansatz. Idealerweise nutzt man einen konkreten Anwendungsfall, um eine Anleitung zu erstellen, also die angeforderte Umsetzung einer entsprechenden Aufgabe. Ohne Anleitung würde man sich hierbei vermutlich an einem früheren Beispiel derselben Art orientieren und versuchen, daraus mit der Methode *kopieren, einfügen, anpassen* die gewünschte neue Instanz zu erzeugen. Die Erstellung der Anleitung geht im Prinzip denselben Weg.

Im ersten Schritt erzeugt man zunächst ein Skript, dass nur 1-zu-1 den Programmcode erzeugt, der das frühere Beispiel ausmacht, also ohne Verallgemeinerungen und ohne Platzhalter. Nun geht man diesen Stück für Stück durch und prüft was man für die geforderte

Instanz anpassen würde, z.B kontextbezogene Benennungen, bestimmte IDs, Änderungen aufgrund anderer Anforderungen oder Voraussetzungen.

Die Idee ist, diese Dinge nun so zu systematisieren dass sie durch das Skript auf jeden gegebenen Anwendungsfall hin angepasst werden können, und der Programmcode den Anwendungsfall am Ende so gut wie möglich vollständig abbildet. Bezeichnungen, sowie das Vorhandensein bestimmter Voraussetzungen oder Anforderungen können abgefragt werden und benötigte IDs können nach Anleitung ermittelt oder mittels Zufallsgenerator generiert werden. Stück für Stück wird so aus einem Skript, das konkreten Programmcode ausgibt, ein Skript, dass die Situation abfragt, und für diese den passenden Programmcode ausgibt, zumindest solchen, der für den aktuellen Anwendungsfall passt.

Die große Kunst dabei ist es, die Abfrage der Situation so niederschwellig wie möglich zu gestalten, so dass die gestellten Fragen und geforderten Eingaben mit so wenig technischem Wissen wie möglich getätigt werden können. Wenn also eine Eigenschaft x nur in bestimmten Fällen gesetzt werden kann, und je nach Situation einen von drei verschiedenen Werten annehmen kann, sollte zunächst verstanden werden, was diesen Fall und diese Situationen ausmachen und was die entsprechenden Werte bedeuten. Die Fragestellung des Skriptes sollte auf diesem Wissen aufbauen. Man sieht, dass der Entwickler des Skriptes ein ähnlich tiefgreifendes Wissen benötigt, wie der Entwickler des Frameworks, zu dessen Verwendung es anleitet.

Nähern wir uns dieser Vorgehensweise mit einem einfachen Beispiel. Entwickelt werden soll eine Archivierungsklasse für die Archivierung eines Dozenten und damit verbunden auch gleich die *FlowProtocol 2*-Anleitung für weitere Archivierungsklassen. Zurückgreifen können wir auf den Programmcode der Archivierungsklasse für Kurse, der schon geschrieben wurde.

Wir beginnen also damit, eine Anleitung für genau diese Klasse zu schreiben, die dann wie folgt aussieht:

Beispiel 29 – Archivklasse 1

```
@Anleitung >> Ordner Archivierer öffnen.
@Anleitung >> Neue cs-Datei erstellen mit Bezeichnung
    >|KursArchivierer.cs
@Anleitung >> Folgenden Code einfügen:
    >|// Klasse für die Archivierung von Kurs-Objekten
    >|public class KursArchivierer : BasisArchivierer<Kurs>
    >|{
    >|    // Kennung der Archivierer-Klasse
    >|    public override Guid ArchiviererKennung()
    >|    {
    >|        return Guid("14df603a-cb42-4560-bf6c-a740356532b6");
    >|    }
    >|
    >|    // Methode zum Archivieren des Kursbesuchs
    >|    public override void Archiviere(Kurs k)
    >|    {
    >|      var konflikt = PruefeAufKonflikte(k);
    >|        if (konflikt.vorhanden)
```

```
>|        {
>|            ALogger.Note($"Konflikt für Kurs '{k}',
>|                {konflikt.Details}");
>|            return;
>|        }
>|        base.StarteArchivierung(k);
>|    }
>|
>|    // Konfliktprüfung für die Kurs-Archivierung
>|    private Konflikt PruefeAufKonflikte(Kurs k)
>|    {
>|        if (k.Status == KStatus.Aktiv)
>|            return new Konflikt("Kurs ist noch aktiv");
>|
>|        // Ansonsten
>|        return Konflikt.KeinKonflikt();
>|    }
>|}
```

Um einen Codeblock auszugeben, der einen vorgegebenen Programmcode erzeugt, muss man lediglich vor jeder Zeile des Codes das Ausgabezeichen `>|` einfügen. Bessere Editoren wie z.B. Notepad++ ermöglichen dies sehr einfach dadurch dass man den Cursor mit der Tastenkombination *Shift + Alt +Cursor down* über mehrere Zeilen ausdehnen kann, so dass nachfolgend eingetippte Zeichen gleichzeitig in jeder einzelnen Zeile eingefügt werden.

Da wir natürlich nicht die Anleitung für die schon vorhandene Kurse-Archivierungsklasse benötigen, sondern eine allgemeine Anleitung haben wollen, geht es im nächsten Schritt darum, die variablen Anteile zu identifizieren und das Skript dahingehend anzupassen.

Die erste Abhängigkeit ist die Bezeichnung *Kurs*, die hier gleich in verschiedenen Bedeutungen auftritt, zum einen als lesbare Bezeichnung des Objekttyps zum anderen aber auch als Name einer Klasse, die an den Generik-Parameter übergeben wird, und zuletzt auch noch als Bestandteil des Klassennamens. Letzteres ist dahingehend zu beachten, da als Objekttypbezeichnung eventuell auch Bezeichnungen mit Umlauten oder Bindestrichen auftreten können, die in einem Klassennamen nichts zu suchen haben. Ein weiterer Unterschied besteht darin, dass die Bezeichnung der Archivierungsklasse hier festgelegt werden kann, wohin gegen die Kurs-Klasse schon im System vorhanden ist und hier nur referenziert wird.

Wir fragen also die Bezeichnung des Objekttyp, sowie den Namen der dazugehörenden Klasse ab und leiten die übrigen Verwendungen daraus ab.

Für die Umwandlung eines Textes in eine Zeichenkette, die weder Umlaute, noch Sonderzeichen enthält gibt es in *FlowProtocol 2* den Befehl `~CamelCase`, der die dazu notwendigen Ersetzungen durchführt. Die Zeichenkette bleibt insgesamt lesbar und kann z.B als Klassenname, Eigenschaft oder Variable im Programmcode verwendet werden.

Um es ganz schön zu haben, leiten wir auch die Variable `k` aus dem Objekttyp ab und ersetzen diese an den verschiedenen Stellen. Dazu suchen wir mit einem regulären Ausdruck den ersten Großbuchstaben in der Bezeichnung und wandeln diesen mit dem `~ToLower`-Befehl in einen kleinen Buchstaben um. Hier muss man natürlich darauf achten dass die

Variable nicht in Konflikt mit anderen Variablen kommen kann. Die Umwandlung einer Zeichenkette in Großbuchstaben wäre analog mit dem Befehl `~ToUpper` möglich.

Die in der Klasse hart codierte Guid-Kennung kann frei gewählt werden, und wird vom Skript selbst als Zufallswert vom Typ *Guid* erzeugt. Dafür wird der `$NewGuid`-Befehl verwendet.

Der Programmcode in der Routine für die Konfliktprüfung ist sehr spezifisch auf die Kursklasse bezogen und lässt sich nicht verallgemeinern. Wir ersetzen diesen durch eine ToDo-Aufforderung im Code und ergänzen die Anleitung um einen Punkt. Wir gehen sogar zusätzlich davon aus, dass es nicht in jedem Fall Konflikte geben kann und fragen diese Voraussetzung explizit ab. Für den Fall, das ist keine Konflikte geben kann, lassen wir die entsprechenden Programmzeilen und Anleitungsschritte weg.

Die beiden Stufen der Aufzählung lassen sich bei Anleitungen sehr gut so verwenden, dass man in der ersten Ebene die durchzuführenden Punkte benennt und die zweite Ebene dazu verwendet, diese genauer auszuführen und zusätzliche Informationen bereitzustellen.

Am Ende bekommen wir die folgende allgemeine Anleitung:

Beispiel 30 – Archivklasse 2

```
~Input Bez: Bezeichnung Objekttyp (Singular)
    ~AddHelpLine Die Bezeichnung des Objekttyps,
    __ für den Objekte archiviert werden sollen.
~Input Klasse: Datenklasse für diesen Objekttyp
    ~AddHelpLine Der Name der Datenklasse, mit der Objekte
    __ dieses Typs im Programm abgebildet werden.
?QKonf: Kann es Konflikte geben?
    ~AddHelpLine Gemeint sind Konflikte, die einer Archivierung
    __ im Wege stehen würden.
    #j: Ja
        ~Set KonflikteMoeglich=ja
    #n: Nein
~CamelCase uBez=$Bez
~Set AKennung=$NewGuid
~RegExMatch vmatch = $BezX|([A-Z])
~ToLower v=$vmatch(1)
@Anleitung >> Ordner Archivierer öffnen.
@Anleitung >> Neue cs-Datei erstellen mit Bezeichnung
    >|$uBezArchivierer.cs
@Anleitung >> Folgenden Code einfügen:
    >|// Klasse für die Archivierung von $Bez-Objekten
    >|public class $uBezArchivierer : BasisArchivierer<$Klasse>
    >|{
    >|    // Kennung der Archivierer-Klasse
    >|    public override Guid ArchiviererKennung()
    >|    {
    >|        return Guid("$AKennung");
    >|    }
```

```
    >|
    >|    // Methode zum Archivieren des Kursbesuchs
    >|    public override void Archiviere($Klasse $v)
    >|    {
    ~If $KonflikteMoeglich==ja
        >|      var konflikt = PruefeAufKonflikte($v);
        >|        if (konflikt.vorhanden)
        >|        {
        >|            ALogger.Note($"Konflikt für $Bez '{$v}',
        >|                {konflikt.Details}");
        >|            return;
        >|        }
    >|        base.StarteArchivierung($v);
    >|    }
    ~If $KonflikteMoeglich==ja
        >|
        >|    // Konfliktprüfung für die $Bez-Archivierung
        >|    private Konflikt PruefeAufKonflikte($Klasse $v)
        >|    {
        >|        //ToDo: Konfliktprüfung implementieren
        >|
        >|        // Ansonsten
        >|        return Konflikt.KeinKonflikt();
        >|    }
    >|}
~If $KonflikteMoeglich==ja
    @Anleitung >> Konfliktprüfung implementieren
        > in $uBezArchivierer.cs in
        ~AddCode  PruefeAufKonflikte(...)
        > bei
        ~AddCode  //ToDo: Konfliktprüfung implementieren
        > Auflistung der Konflikte nach folgendem Muster:
        >|        if (Konfliktbedingung)
        >|            return new Konflikt("Konfliktbeschreibung");
```

9.3 Testvorbereitung

Jede Entwicklung sollte getestet werden, im Idealfall sogar durch automatisierte Tests. Standardentwicklungen sind davon nicht ausgeschlossen auch wenn sie dadurch, dass sie auf schon mehrfach genutztem Framework aufbauen, ein geringeres Fehlerrisiko aufweisen, wie neue Individualentwicklungen.

Ähnlich wie die Implementierung, variieren auch die Testpunkte nur in Abhängigkeit der Bezeichnungen und Optionen innerhalb der Entwicklung und können so ebenfalls durch das Anleitungskript erstellt werden. Unabhängig davon, ob die Entwickler ihre Entwicklungen gegenseitig testen, oder ob dafür eine eigene Testabteilung bereitsteht ist letztlich der Ersteller des Programmcodes dafür verantwortlich, den Testumfang so zu formulieren, dass jede neu entwickelte oder geänderte Zeile zumindest einmal durchlaufen wird. Hierfür muss er die Perspektive des Anwenders annehmen, und die dazugehörenden Schritte

der Programmbedienung so formulieren, dass im Hintergrund die Zustände und Fälle auftreten, bei denen die implementierte Funktionalität durchlaufen wird.

In gleicher Weise muss er auch den Effekt der Funktionen so beschreiben, dass dieser an der Programmoberfläche überprüft werden kann. Das kann in manchen Fällen herausfordernd sein, insbesondere wenn es darum geht, bewusst Konflikte und Ausnahmezustände zu provozieren, die im normalen Ablauf eher selten auftreten.

Im Anleitungsskript für den Entwickler können wir auf jeden Fall schon einmal die Integrationstests ausformulieren, die sich unmittelbar aus dem Programmcode aus der Anleitung ergeben. Für den Fall, dass Konflikte möglich sind, ergänzen wir im ersten Testpunkt den Zusatz *sofern keine Konflikte auftreten*. Zusätzlich ergänzen wir die Anleitung dahingehend, eine Testaufgabe zu erstellen und mögliche Konfliktarten in den Testpunkten aufzuzählen. Die Testpunkte werden in einem eigenen Abschnitt aufgelistet und können so parallel zu den dazugehörenden Anleitungsschritten gesammelt werden.

Damit kommen folgende Zeilen in unserer Anleitung hinzu:

Beispiel 31 – Archivklasse 3

```
...
@Testpunkte >> Ein $Bez-Objekt kann archiviert werden
~If $KonflikteMoeglich==ja
    ~AddText , sofern keine Konflikte auftreten
~AddText .
@Testpunkte >> Die Archivierungskennung für $Bez-Objekte
    __ lautet $AKennung.
    > Kann im Protokoll nachgeschlagen werden.
...
@Anleitung >> Testaugabe erstellen
    > Titel: Test der $Bez-Archivierung
    > Testpunkte aus Abschnitt "Testpunke" übernehmen
~If $KonflikteMoeglich==ja
    @Anleitung >> Konflikte in Testpunkten auflisten
        > Bei "ToDo: Konflikte auflisten"
    @Testpunkte >> Folgende Konflikte werden erkannt:
        > ToDo: Konflikte auflisten
    @Testpunkte >> Wenn ein Konflikt erkannt wird,
        >#  wird die Archivierung abgebrochen,
        >#  protokolliert, wo der Konflikt aufgetreten ist,
        >#  Details zum Konflikt protokolliert.
```

Was mit Integrationstests in Textform möglich ist, ist je nach Art der Entwicklung auch noch in Form von Unittests möglich. Unitest sind im Programmcode hinterlegte Klassen und Funktionen, die nur dafür da sind, die im Produktivcode verwendeten elementaren Klassen auf Korrektheit zu testen. Die Tests selbst gehören nicht zum Produkt und haben keine Funktionen in der Produktivumgebung. Unittests werden in der Regel durch spezielle Attribute gekennzeichnet und in eigenen Projekten verwaltet.

Im Fall unserer Archivierungsklasse könnte z.B. die Archivierer-Kennung sehr einfach über einen Unittest geprüft werden. Das Skript könnte dazu wie folgt den entsprechenden Testcode erzeugen:

Beispiel 32 – Archivklasse 4

```
...
@Anleitung >> Ordner ArchiviererTest öffnen.
@Anleitung >> Neue cs-Datei erstellen mit Bezeichnung
    >|$uBezArchiviererTests.cs
@Anleitung >> Folgenden Code einfügen:
    >|public class $uBezArchiviererTests
    >|{
    >|    [Fact]
    >|    public void ArchiviererKennung_ShouldReturnExpectedGuid()
    >|    {
    >|        // Arrange
    >|        var archivierer = new $uBezArchivierer();
    >|
    >|        // Act
    >|        var result = archivierer.ArchiviererKennung();
    >|
    >|        // Assert
    >|        Assert.Equal(Guid.Parse("$AKennung"), result);
    >|    }
    >|}
```

Denkbar wäre auch ein Skript, dass direkt von den Testern verwendet wird, um sich die eigene Arbeit zu vereinfachen. Dieses müsste dann auf Basis der Informationen ausgefüllt werden, die von den Entwicklern bereitgestellt werden, und es müsste Mehrwert generieren, in dem es daraus hilfreiche Informationen für die Durchführung von Tests ableitet.

Es ist schwierig, in diesem Bereich Beispiele zu finden, die nicht besser in der oben genannten Form als Teil der Testvorbereitung in der Entwicklungsanleitung aufgehoben sind. Ein denkbares Anwendungsgebiet wären auf jeden Fall die automatisierten Oberflächentests. Hierbei wird mit speziellen Tools eine Programmbedienung über die Benutzeroberfläche simuliert, indem Schaltflächen gedrückt und Eingaben getätigt werden, wobei die Reaktion des Programms mit einem vorgegebenen Sollverhalten abgeglichen wird.

Solche Oberflächentests werden in der Regel ebenfalls als Programmcode verwaltet und folgen oft ähnlichen Strukturen, wie der Produktivcode, also mit wiederverwendbaren Anteilen und Mustern und einem eigenen Framework. Insbesondere das Vorbereiten einer Testsituation und das Aufräumen nach Durchführung des Tests sind meist aufwendige und oft benötigte Funktionen, die einen gut organisierten Baukasten voraussetzen.

Ein Unterstützungsskript könnte sehr gut die Voraussetzungen für einen geplanten automatisierten Oberflächentest abfragen und damit schon mal den Test soweit wie möglich vorbereiten.

9.4 Product-Owner-Unterstützung

Der Product Owner oder kurz PO ist die zentrale Figur im Entwicklungsprozess. Er kennt das Produkt wie seine Westentasche, weiß, wie es sich entwickelt hat und wohin es sich

entwickeln soll, und mit welchen Bausteinen und nach welchen Prinzipien diese Entwicklung abläuft. Er steht in direkt im Austausch mit den Kunden, kennt die Anwender und hat gelernt, dass auch die schlimmsten Entscheidungen aus guten Absichten getroffen wurden.

Die Menge an Dingen, die bei der professionellen Softwareentwicklung zu beachten sind, sind ungeheuer vielschichtig, und die permanente Weiterentwicklung eines großen bestehenden Systems bringt noch zahlreiche eigene Aspekte dazu. Auch wenn der PO bei dieser Aufgabe nicht allein steht, und sowohl auf das ganze Entwicklerteam, als auch auf die ihm zuarbeiten den Spezialisten für Usability und Oberflächendesign bauen kann, hängt es nachher doch an ihm, aus dem ganzen Wissen und der Erfahrung ein gemeinsames Ganzes zu machen, dass individuelle Vorlieben und Wünsche zurückstellt und auf für alle nachvollziehbare klaren Regeln aufbaut.

Unterstützungssysteme für die Product Owner waren mit die ersten Anwendungsbeispiele, die mit *FlowProtocol 2* umgesetzt wurden. Gerade dort, wo Standard Entwicklungen hauptsächlich in einem sehr engen Rahmen konfiguriert werden konnten und mussten, konnten entsprechende Skripte all die benötigten Punkte abfragen und daraus eine weitestgehend vollständige Beschreibung erstellen, die alle entscheidenden Aspekte der Entwicklung in hohem Detailgrad umfasste.

Index