Softwareentwicklung mit FlowProtocol 2

Wolfgang Maier

3. August 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 kopieren kann, und der `~CamelCase`-Befehl, mit dem Namen für Felder, Funktionen oder Variablen erzeugen können. 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`

Der Programmcode kann direkt mit Git oder der Entwicklungsumgebung Visual Studio Code in ein lokales Verzeichnis, z.B. `D:\Apps\FlowProtocol2`, übertragen, und dort mit dem dotnet-Befehl kompiliert werden, wobei als Zielframework .NET 8 vorausgesetzt wird:

```
dotnet.exe build
    D:\Apps\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 einem Unternehmen 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 im Umfeld der Softwareherstellung. *FlowProtocol 2* selbst benötigt auf dieses Verzeichnis nur Leserechte.

In den Programm- und Konfigurationsdateien *launchSettings.json*, *appsettings.json* und *Program.cs* sind die Ports 5000, bzw. 5001 eingetragen, die man je nach Installation auch nochmal anpassen möchte.

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 *Hallo Welt.fp2* und speichern diese direkt, oder 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 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 *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. Der Befehl >> 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 Auswahlabfragen und die später noch kommenden Texteingaben können primär aus einer Folge aus Buchstaben und Ziffern, sowie runden Klammern be-

stehen, die Schlüssel für Auswahlwerte nur aus Buchstaben und Ziffern. Sie dienen wie schon beschreiben 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 innerhalb des gesamten Skriptes 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 Leerzeichen 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 *Scipts*-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.

Ergänzend dazu gibt es in Abschnitt 12 ebenfalls eine Befehlsreferenz, die alle Befehle auflistet, und auch auf deren Verwendung in dieser Dokumentation hier verweist.

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 sein 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 jeweiligen 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.

Beispiel 33 zeigt, wie man einen solchen Auswahlbaum als wiederverwendbare Komponente anlegt.

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 objektiven Kriterien und der Vergleich mit Referenzbeispielen. 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: Die Datensicherheit ist verletzt oder gefährdet.
        >> Stufe-1-Fehler: Datensicherheitssverletzung
    #no: Nein, die o.g. Kriterien treffen nicht zu.
        ?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 treffen nicht zu.
                ?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 treffen nicht zu.
    >> Stufe-3-Fehler
```

Für die Abarbeitung der Fehler-Aufgaben der verschiedenen 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.

In Beispiel 16 wird ein Skript gegeben, dass die Menge der Vorgänge in jedem Schweregrad für eine vorgegebene Anzahl so berechnet, dass sie so gut wie möglich in diesem Verhältnis steht.

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. Dort wo diese nicht gelingt, lässt sich anhand der beantworteten Fragen zumindest eine gute und aufs Wesentliche beschränkte Beschreibung der Problemsituation erstellen. 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 riskantes auszuprobieren.

Wie schon bei dem zuletzt genannten Beispiel, das auch mit dem Expertensystem kombiniert werden kann, zeigt sich der enorme 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 sich dieses Wissen aneignen muss. *FlowProtocol 2* übernimmt die Anwendung des Wissens und 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 automatische Nummerierung von Schlüsseln für Auswahlabfragen und Texteingabe ist auch explizit möglich, in dem man einem selbstgewählten Basisschlüssel das '-

Zeichen anschließt. Mehrere nacheinander im Scriptcode vorkommenden Schlüsselangaben `Q'` werden dann intern zu `Q_1`, `Q_2`, `Q_3` usw. expandiert. Durch den Unterstrich werden Übereinstimmungen mit explizit vergebenen Schlüsseln vermieden.

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 6. 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 auch 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 drei 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 jeweiligen Abschnitts unter und Ausgaben der dritten Ebene dementsprechend der letzten Ausgabe der zweiten Ebene.

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 eine 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
        >.* Der Editor bietet Methoden zur Auswahl an.
    ># 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 wird ein ebenfalls nicht nummerierter Unterpunkt ausgegeben, unter dem zweiten ein Codeblock mit drei Zeilen. Unter dem dritten Aufzählungspunkt sieht man drei Unterpunkte, die mit

Kleinbuchstaben durchnummeriert sind, der erste davon hat wiederum einen nicht nummerierten Unterpunkt.

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 noch 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 Kernelement 12.3), 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 ruft 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 semantischer Zucker, um den Index-Teil der Variablen hervorzuheben und vom Feldnamen 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 gibt 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.
```

Am Anfang 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 der Aufgabe be-

schriebenen 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, als 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 passieren muss und nicht mittels Bauchgefühl, 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 verwendet, die auch berücksichtigen, 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, um 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 so Aufgabenbeschreibungen generieren, die fast alle wesentlichen Konfigurationsaspekte auflisteten. Manuell nachgearbeitet werden mussten individuelle Beschreibungstexte und eben Begriffe und Benennungen, die dafür meist mit einem gut erkennbaren Platzhalter *xxx* in der Ausgabe ersetzt wurden, was insbesondere bei den

wiederholt vorkommenden Begriffen den Komfort 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
~Execute
~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.

Der Befehl `~Execute` sorgt dafür, dass das Skript bis zu dieser Stelle ausgeführt wird, solange, bis alle darüber stehenden Eingabebefehle abgearbeitet wurden, und die Eingabewerte in den Variablen zur Verfügung stehen. Er ist an dieser Stelle notwendig, da Variablen ohne Wert beim `~Replace`-Befehl zu Fehlern führen würden. Der Befehl kann auch bewusst zu Formatierung der Eingabe genutzt werden wie in Abschnitt 6.2 beschrieben wird.

Die Möglichkeit, Texte und Zahlen über eine einfach per Link adressierbare Webanwendung abzufragen und zur Durchführung von Berechnungen zu verwenden, ist eine gute Alternative für so manche Kalkulationshilfe, die momentan noch in Form von Tabellenkalkulationsdateien verteilt werden.

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 Anwendungskontext 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. Beim 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, mehrere 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=2,3,2,4,2,3,2,2,3,2|,
~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 mit `~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 werden.

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 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 Bug-Vorgang zum verteilen übrig ist. Letzteres wird mit dem Befehl `~EvalExpression` getrennt ausgewertet und in der Variablen `anybug` abgespeichert.

In der Inneren Schleife wird die Indexsequenz $2,3,2,4,2,3,2,2,3,2$ 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 verteilt 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 dabei nach oben und unten durch einen absoluten Wert begrenzt. Zusätzlich könnte man das Planungsverhältnis abfragen und das Gesamtpaket aus Prozentwert, obere und untere Grenze, sowie Verhältnis als fertige Planungsstrategie auswählbar machen.

Für ein beliebiges Verhältnis $S_0 : S_1 : \cdots : S_{v-1}$ für die Einplanung von v verschiedenen Schweregraden und eine Folge $a = (a_i)_{i \in n}$ von Indices $a_i \in v$ kann man berechnen, für

welchen nächsten Index $a_n \in v$ die erweiterte Gesamtfolge $a' = (a_i)_{i \in n+1}$ die kleinste Abweichung $d(a_n)$ zum vorgegebenen Verhältnis hat, mit

$$d(a_n) = \min_{l \in v} \left(1 + \left| \{ i \in n \colon a_i = a_n \} \right| - \frac{(n+1) S_l}{\sum_{j \in v} S_j} \right)$$

Diesen wählt man dann, um die Folge zu erweitern. Die Berechnung so oft wiederholt werden, wie man weitere Folgenglieder benötigt.

6.3 Hilfezeilen

Einer der großen Vorteile einer interaktiven Anleitung liegt darin, dass man eben nicht wissen muss, welche Schritte man ausführen muss und welche man weglassen kann, 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 mit 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. Die Vorteile, den Aufbau von Anleitungen mit Programmlogik zu verknüpfen, Bedingungen abzuprüfen und Werte zu berechnen, waren insgesamt so groß, dass schließlich alle wichtigen Programmierbefehle in *FlowProtocol 2* Einzug fanden.

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 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 beiden 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` mit dem `~Round`-Befehl 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 16 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 mit dem Befehl `~ExitLoop` 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 mit 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 $i. Parameter:
            ~Set Vars($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, Bedingungen 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 eine weitere Bedingung mit einem weiteren 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 aneinandergereiht werden. Abschließend kann mit `~Else` noch ein Codeblock angehängt werden, der nur dann ausgeführt wird, 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` und `true` (wahr), sowie `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). Siehe auch Kernelement 12.1.

7.5 Funktionen

Viele spannende und mächtige Algorithmen gründen auf Funktionen, die sich rekursiv selbst aufrufen. Es gibt Programmiersprachen wie Scheme, die einen dazu zwingen, so gut wie jede Problemstellung mittels Rekursion zu lösen. Auch *FlowProtocol 2* erlaubt die Definition von Funktionen und rekursive Aufrufe, obgleich wie schon in früheren Abschnitten erwähnt, dort nicht unbedingt 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=A
~Set MoveToS=C
~Set MoveCountS=4
~GoSub Move; BaseKey=S
~End

// Bewegt n Scheiben von Stange a nach b
// MoveFrom$BaseKey:  Ausgangsstange a (A, B, C)
// MoveTo$BaseKey:    Zielstange b (A, B, C)
// MoveCount$BaseKey: Anzahl Scheiben n
~DefineSub Move
    ~If $MoveCount$BaseKey>0
        ~Set Q1$BaseKey=$MoveFrom$BaseKey
        ~Set Q2$BaseKey=$MoveTo$BaseKey
        ~Set Q3$BaseKey=ABC
        ~Replace Q3$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 Umstapeln 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 standardmäßig am Ende des Skriptes. Ihr Code wird nur durchlaufen, wenn sie aufgerufen werden, trotzdem wird hier zur besseren Übersichtlichkeit die Ausführung vorher mit dem Befehl `~End` beendet. 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 dem 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.

In jedem Schritt muss zunächst die Stange herausgefunden werden, die aktuell weder Ausgangspunkt, noch Ziel der Umschichtung darstellt und die als Zwischenablage verwendet werden kann. Das wird dadurch bewerkstelligt, dass man im Ausdruck `ABC` die Kennungen der beiden gegebenen Stangen mit dem `~Replace`-Befehl entfernt.

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 Variablen definiert werden, die sich auf den übergebenen `BaseKey`-Wert beziehen. Im Beispiel wird vor dem ersten Aufruf im Skript `MoveFromS=A` als einer von drei Funktionsparametern gesetzt und beim Funktionsaufruf wird `BaseKey=S` übergeben. Damit wird innerhalb der Funktion `$MoveFrom$BaseKey` zu `$MoveFromS` und schließlich zu `A` ersetzt. Bei den rekursiven Aufrufen wird der `BaseKey`-Wert dann jeweils um die Buchstaben A und B erweitert, so dass am Ende Funktionsparameter zu 15 verschiedenen `BaseKey`-Werten verwaltet 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 dass 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 16 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 jeder 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 ein großer 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 theoretisch möglich sind, muss auch die Software damit in irgendeiner Form umgehen können, und damit sie das kann, müssen auch diese Fälle von Anfang an betrachtet und behandelt werden.

Gerade bei der Umsetzung von Formeln ist es ungeheuer wichtig, dass der komplette zulässige Definitionsbereich bei Implementierung und Test abgedeckt sind, 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.

In vielen Fällen hängen weitere Teilkombinationen nur an einzelnen Werten und die Bestimmung aller Kombinationen entspricht dem ausmultiplizieren eines Terms mit den Operationen $+$ und $\cdot$ und einer beliebigen Klammerung, nur dass die Elemente Textfragmente sind und keine Zahlen. Das folgende Beispiel zeigt, wie die Summanden des Ergebnisausdruckes am Ende die einzelnen Kombinationen bilden:

$$
\begin{aligned}
K = &(\text{Material Holz} + \text{Material Metall}) \\
&\cdot (\text{Farbe blau} + \text{Farbe grün} \cdot (\text{einfarbig} + \text{mit gelben Streifen})) \\
= &\ \text{Material Holz} \cdot \text{Farbe blau} \\
&+ \text{Material Holz} \cdot \text{Farbe grün} \cdot \text{einfarbig} \\
&+ \text{Material Holz} \cdot \text{Farbe grün} \cdot \text{mit gelben Streifen} \\
&+ \text{Material Metall} \cdot \text{Farbe blau} \\
&+ \text{Material Metall} \cdot \text{Farbe grün} \cdot \text{einfarbig} \\
&+ \text{Material Metall} \cdot \text{Farbe grün} \cdot \text{mit gelben Streifen}
\end{aligned}
$$

Mit der in Abschnitt 8.3 beschriebenen Funktion zur Anwendung regulärer Ausdrücke lassen sich solche Klammerausdrücke wiederholt in einem Eingabetext suchen und ausmultiplizieren, bis am Ende der gesamte Ausdruck expandiert ist. Ebenso lässt sich nach definierten Textmarken suchen, die, wenn sie in einer Kombination aufeinandertreffen, diese aus dem Ergebnis entfernen. Zusätzlich lässt sich auch noch eine Textmarke für eine Ergebnisgruppierung definieren, so dass die Kombinationen als zweistufige Liste ausgegeben werden.

8.2 Ersetzungen und Zufallsgenerierung

Den Ersetzungsbefehl `~Replace` haben wir schon in vielen Skripten gesehen, etwa in Beispiel 14. 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 $AnzSonder>0
        ~Random s=1..$SZ(0)
        ~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 ANSI-Code 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.

Für die Erzeugung der Sonderzeichen wird zunächst eine Zufallszahl von 1 bis zur Länge der Liste `SZ` generiert, wobei diese mit `$SZ(0)` abgerufen wird. Mit dieser Zahl wird der entsprechende Listeneintrag ausgelesen, der dann in gleicher Weise wie die anderen Codes in ein Zeichen umgewandelt wird. Damit kann die Liste um weitere Codes ergänzt werden, ohne dass an anderer Stelle die Anzahl der Liste angepasst werden muss.

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 Chatbots der großen 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 irgendwo im Skripte-Verzeichnis 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 man kann sie auslesen. 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, Beschreibungen 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. In der ersten Spalte 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 oder Beschreibung des Begriffs an, im zweiten Fall enthält 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 >>_ $T
~ForEachLine z in Terminologieliste.txt; NoFormat
    ~RegExMatch ti=$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
```

Der Terminologieprüfer 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* und der Begründung, dass es sich um einen Anglizismus handelt, als nicht zulässige Benennung abgelehnt.

Der Befehl `~RegExMatch` liefert in der Ergebnisvariablen `ti` eine Liste zurück. Das nullte Element `ti(0)` gibt an, ob für den Ausdruck ein Treffer gefunden wurde, die nachfolgenden Elemente enthalten die Teiltexte des gefundenen Ausdrucks, die über die runden Klammern im Ausdruck als Gruppen angegeben wurden.

Das `NoFormat`-Argument gibt an, dass die Datei nicht das Dateilisten-Format hat und daher alle Zeilen unverändert zurückgegeben werden sollen. Beispiele für Dateilisten findet man in Beispiel 46 und 47.

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 Hilfezeilen 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. 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, desto 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 hier 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. Sie 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 vom ausführenden Entwickler 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, dass einem eine solche Anleitung die Arbeit leichter macht, stellt auch praktisch sicher, dass diese verwendet wird und damit 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, das die vorliegende Situation abfragt, und für diese den passenden Programmcode samt Einbauanleitung generiert.

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 eines Kurs-Objektes
    >|    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++, bzw. Visual Studio Code ermöglichen dies sehr einfach dadurch, dass man den Cursor mit der Tastenkombination *Shift* + *Alt* + *Cursor down*, bzw. *Strg* + *Alt* + *Cursor down* über mehrere Zeilen ausdehnen kann, so dass nachfolgend eingetippte Zeichen gleichzeitig in jeder einzelnen Zeile eingefügt werden.

Da wir ja 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 Objekttyps, sowie den Namen der dazugehörenden Klasse ab und können daraus die übrigen Verwendungsvarianten ableiten.

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 Befehl `~ToLower` in einen Kleinbuchstaben 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 cBez=$Bez
~Set AKennung=$NewGuid
~RegExMatch vmatch = $BezX|([A-Z])
~ToLower v=$vmatch(1)
@Anleitung >> Ordner Archivierer öffnen.
@Anleitung >> Neue cs-Datei erstellen mit Bezeichnung
    >|$cBezArchivierer.cs
@Anleitung >> Folgenden Code einfügen:
    >|// Klasse für die Archivierung von $Bez-Objekten
    >|public class $cBezArchivierer : BasisArchivierer<$Klasse>
    >|{
    >|    // Kennung der Archivierer-Klasse
    >|    public override Guid ArchiviererKennung()
    >|    {
    >|        return Guid("$AKennung");
    >|    }
    >|
    >|    // Methode zum Archivieren eines $Bez-Objektes
    >|    public override void Archiviere($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 $cBezArchivierer.cs in
        ~AddCode  PruefeAufKonflikte(...)
        > bei
        ~AddCode  //ToDo: Konfliktprüfung implementieren
        > Auflistung der Konflikte nach folgendem Muster:
        >|        if (Konfliktbedingung)
        >|            return new Konflikt("Konfliktbeschreibung");
```

Beim Aufbau eines Anleitungsskriptes für eine Standardentwicklung stechen die Schwächen der eigenen Framework-Klassen besonders hervor, etwa zwingend erforderliche Methoden, die jedoch nicht erzwungen werden, unschöne Abhängigkeiten von Aufrufreihenfolgen oder generell ungewollte Codewiederholungen. Eine ideal aufgebaute Klasse vermeidet Wiederholungen und führt den Entwickler allein mit Hilfe des Compilers an allen relevanten Fragestellungen vorbei und sorgt so selbständig für einen vollständigen und mustergültigen Einbau. Da dieser Anspruch schwierig aufrecht zu halten, und noch schwieriger im Nachhinein herzustellen ist, und da die Interaktion mit einem gut gemachten Anleitungsskript gefälliger sein dürfte, als die mit dem C#-Compiler, bleiben trotzdem genug Gründe, um solche Anleitungen zu erstellen.

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 Anleitungsskript 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
    >|$cBezArchiviererTests.cs
@Anleitung >> Folgenden Code einfügen:
    >|public class $cBezArchiviererTests
    >|{
    >|    [Fact]
    >|    public void ArchiviererKennung_ShouldReturnExpectedGuid()
    >|    {
    >|        // Arrange
    >|        var archivierer = new $cBezArchivierer();
    >|
    >|        // Act
    >|        var result = archivierer.ArchiviererKennung();
    >|
    >|        // Assert
    >|        Assert.Equal(Guid.Parse("$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, indem 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, d.h. es werden Schaltflächen gedrückt und Eingaben getätigt und dabei wird immer wieder die Reaktion des Programms mit einem vorgegebenen Sollverhalten abgeglichen.

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 im direkten Austausch mit den Kunden, kennt die Anwender und hat gelernt, dass auch die schlimmsten Entscheidungen aus guten Absichten getroffen werden.

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 Entwicklungsteam, als auch auf die ihm zuarbeitenden Spezialisten für Usability und Oberflächendesign bauen kann, hängt es nachher doch an ihm, aus dem ganzen Wissen und der Erfahrung ein großes Ganzes zu machen, das individuelle Vorlieben und Wünsche zurückstellt und auf für alle nachvollziehbaren Regeln aufbaut.

Unterstützungssysteme für die Product Owner waren mit die ersten Anwendungsbeispiele, die mit *FlowProtocol 2* umgesetzt wurden. Gerade dort, wo Standardentwicklungen hauptsächlich in einem sehr engen Rahmen konfiguriert werden 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.

Wir bauen nun ein Beispiel für ein solches System auf und beginnen mit einem Teilaspekt, der in vielen PO-Unterstützungssystemen benötigt wird, nämlich der Beschreibung der Stelle innerhalb der Anwendung, an der die Entwicklung greifen soll. Wie schon in Abschnitt 3.1 beschrieben, lässt sich die Struktur einer Anwendung gut als Entscheidungsbaum aufbauen, so dass sich ein beliebiger Programmbereich oder ein Registerblatt mit wenigen Klicks auswählen lässt.

Wir speichern dieses Skript für die Auswahl in einer eigenen Funktionsdatei mit der Dateiendung *.fps* und können diese dann in verschiedenen anderen Skripten mit Hilfe des `~Include`-Befehls aufrufen. Durch die andere Dateiendung wird das Skript nicht in der Auswahl von *FlowProtocol 2* angezeigt, so dass es nicht eigenständig aufgerufen werden kann. Der Befehl verfügt ähnlich wie der `~GoSub`-Befehl über einen `BaseKey`-Parameter, so dass man in der Datei die `$BaseKey`-Variable nutzen kann und mehrere Aufrufe innerhalb desselben Skriptes möglich sind, ohne dass es zu Konflikten bei den Schlüsseln für Auswahlabfragen und Eingaben kommt.

Da die Auswahl eines Programmbereichs für verschiedene Zwecke verwendet werden kann, schaffen wir über die Variable `PHinweis` die Möglichkeit, einen Text zu übergeben, der die Fragestellung um den Verwendungszweck erweitert. Die getroffene Auswahl wird dann nach Ausführung über die beiden Variablen `PAusw` und `UAusw` an das aufrufende Skript zurück übergeben.

Beispiel 33 – Programmauswahl.fps

```
?P: Programmbereich $PHinweis
    #D: Dozenten
        ~Set PAusw=Dozenten
        ?U: Unterbereich in Dozenten $PHinweis
            #A: Auswahliste Dozenten
                ~Set UAusw=Auswahliste Dozenten
            #S: Registerblatt Stammdaten
                ~Set UAusw=Registerblatt Stammdaten
            #F: Registerblatt Fächer
                ~Set UAusw=Registerblatt Fächer
    #K: Klassen
        ~Set PAusw=Klassen
        ?U: Unterbereich in Klassen $PHinweis
            #A: Auswahliste Klassen
                ~Set UAusw=Auswahliste Klassen
            #S: Registerblatt Schüler
                ~Set UAusw=Registerblatt Schüler
            #U: Registerblatt Unterricht
                ~Set UAusw=Registerblatt Unterricht
```

Unsere Anwendungsbeispiel selbst unterstützt bei der Beschreibung einer Listenfunktion, womit in diesem Zusammenhang eine Programmfunktion gemeint ist, die für ausgewählte Zeilen einer Liste aufgerufen werden kann.

Diese Aufgabenstellung klingt zunächst recht einfach, aber wie wir sehen werden, gibt es viel zu beachten, an das man nicht unmittelbar denkt. Genau hier liegt der Vorteil eines solchen Skiptes, das diese Punkte systematisch abfragt.

Beispiel 34 – Listenfunktion

```
~Input Bez: Bezeichnung der Funktion
~Set PHinweis= (Wo soll die Funktion eingebaut werden?)
~Include Programmauswahl.fps; BaseKey=E

@Story >>_ Einbau der Listenfunktion "$Bez"
    __ in die Liste des Bereichs "$PAusw - $UAusw".

@Story >> Die Funktion ist per Kontextmenü aufrufbar.
    > Bezeichnung Menüpunkt: $Bez
?Pos1: Gibt es schon einen Funktionen-Abschnitt im
    __ Kontextmenü?
    #j: Ja
        ?Pos2: An welcher Stelle soll der Menüpunkt in
        __ das Kontextmenü eingebaut werden?
            #O: Ganz oben
                ~Set wogenau=ganz oben
            #U: Ganz unten
```

```
                ~Set wogenau=ganz oben
            #D: Irgendo dazwischen
                ~Input Pos3: Einbau unterhalb von...
                ~Set wogenau=unterhalb von "$Pos3"
        > ...in vorhandenen Funktionen-Abschnitt $wogenau
    #n: Nein
        > Funktionen-Abschnitt anlegen nach Std.-funktionen

@Story >> Die Funktion
?QRechte: Welche Rechte setzt die Funktion voraus?
    ~AddHelpLine Schreibrechte bei schreibenden Funktionen
    #L: Leserechte auf die Liste
        ~AddText  setzt Leserechten auf die Liste voraus.
    #S: Schreibrechte auf die Liste
        ~AddText  setzt Schreibrechte auf die Liste voraus.
    #E: Die Funktion hat eigene Rechte
        ~AddText  hat eigene Rechte.
        > Die Berechtigungsstruktur muss erweitert werden.
        > Eingliederung unter Listenfunktionen - $Bez
        > Das neue Recht ist zunächst nur für Admins gesetzt.

@Story >> Die Funktion ist
?AKont: Die Funktion ist
    #E: ...beschränkt auf Einfachauswahl
        ~AddText  beschränkt auf Einfachauswahl.
        > Bei Mehrfachauswahl Menüpunkt deaktivieren
    #M: ...auch verfügbar für Mehrfachauswahl
        ~AddText  auch verfügbar für Mehrfachauswahl.

@Story >> Die Funktion
?Wirk: Die Funktion
    #A: ...öffnet einen Assistenten
        ~AddText  öffnet einen Assistenten.
        > Beschreibung und Skizze Assistent: xxx
        @ToDo >> Assisten skizzieren und beschreiben
    #R: ...wird nach Dialog-Rückfrage ausgeführt
        ~AddText  wird nach Dialog-Rückfrage ausgeführt.
        > Typ: Rückfrage-Dialog vor Ausführung
        > Text: xxx
        @ToDo >> Text Dialog-Rückfrage ausformulieren
    #O: ...wird ohne Dialog-Rückfrage ausgeführt
        ~AddText  wird ohne Dialog-Rückfrage ausgeführt.

@Story >> Die Funktion bewirkt folgendes:
        > xxx
@ToDo >> Wirkung der Funktion beschreiben.

@Story >> Das Benutzerfeedback
```

```
?Feed: Das Benutzerfeedback
    #E: ...ist unmittelbar erkennbar
        ~AddText  ist unmittelbar erkennbar.
        > Kein Feedback-Dialog notwendig.
    #D: ...wird explizit über einen Dialog gegeben
        ~AddText  wird explizit über einen Dialog gegeben.
        > Typ: Feedback-Dialog nach Ausführung
        > Text: xxx
        @ToDo >> Text Feedback-Dialog ausformulieren
```

Zunächst muss die Funktion benannt werden und dann muss die Liste beschrieben werden, für die die Funktion eingebaut werden soll. Dies ist mit einer Texteingabe und dem Aufruf der als Funktionsdatei ausgelagerten Auswahlfunktion schnell umgesetzt.

Die Festlegung, dass eine Listenfunktion über das Kontextmenü der Liste aufgerufen wird, ist fest im Skript hinterlegt. Der Product Owner hat hier keine Auswahlmöglichkeit und soll diese auch nicht haben. Er bestimmt, was er haben möchte und das Designsystem legt Form und Aussehen fest.

Dadurch, dass die grundlegende Design-Entscheidung fest im Skript hinterlegt ist, kann diese wiederum explizit in die Beschreibung einfließen, sodass diese durchgängig berücksichtigt wird und niemand das Designsystem auswendig im Kopf haben muss.

Das Designsystem legt in unserem Fall ebenfalls fest, dass innerhalb des Kontextmenüs ein Abschnitt für solche Funktionen angelegt werden muss, der beim Einbau einer neuen Funktion auch schon vorhanden sein kann. Wenn der Abschnitt schon vorhanden ist, wird der Product Owner aufgefordert, die Position für den neuen Menüpunkt explizit vorzugeben. Auch derartig kleine Details sollten nicht dem Zufall überlassen werden.

Ein Funktionsaufruf erfordert Berechtigungen, die sich entweder aus den Berechtigungen für die Liste ableiten lassen oder explizit angelegt werden müssen. Wenn die Funktion Daten verändert, sollte sie in jedem Fall Schreibrechte auf die Liste voraussetzen, wenn die Daten nur lesend verwendet werden, reichen auch Leserechte. Solche Hinweise auf allgemein festgelegte Regeln sollten im Skript als Hilfezeilen bei den damit verbundenen Fragen und Texteingaben ergänzt werden.

Im nächsten Schritt wird gefragt, ob die Funktion auf einzelne Zeilen beschränkt ist, oder auch für eine Mehrfachauswahl von Zeilen aufgerufen werden kann. Es ist wichtig, sich über solche Punkte im Vorfeld Gedanken zu machen, da oft beides möglich, der allgemeine Fall jedoch meist deutlich schwieriger umzusetzen ist und manchmal auch noch weitere Fragen nach sich zieht. Oft wird diese Entscheidung erst nach Rücksprache mit den Entwicklern getroffen, wenn der Mehraufwand abgeschätzt ist. Die Rücksprache mit den Entwicklern könnte bei solchen Punkten sogar als eigene Antwortoption zur Auswahl gestellt werden.

Nun wird festgelegt, wie die Funktion bei Auswahl reagiert. Soll ein Assistent geöffnet werden oder soll die Funktion direkt ausgeführt werden? Soll die Ausführung der Funktion über einen Rückfrage-Dialog abgesichert werden, in Sinne von *sind Sie sicher?* Auch hier bezeichnet Rückfrage-Dialog ein Element aus dem Designsystem, dessen Aussehen weitestgehend festgelegt ist. Symbol und die zur Auswahl stehenden Schaltflächen werden daher nicht abgefragt, nur der Text.

In gleicher Weise wird am Ende sichergestellt dass der Benutzer auch ein Feedback für die Ausführung der Funktion bekommt. Wenn dieses unmittelbar über die Programmo-

berfläche erkennbar ist, muss dafür nichts zusätzliches getan werden. Ansonsten sollte eine explizite Rückmeldung in Form eines Dialogs gegeben werden.

Die eigentliche Beschreibung, was die Funktion genau macht und wie der Assistent aussieht, der eventuell aufgerufen wird, ist ebenfalls Teil der Beschreibung. Wie alles, was nur einmalig in Textform beschrieben werden muss, werden diese Informationen nicht über Texteingaben abgefragt, sondern am Ende durch Platzhalter in der Beschreibung gekennzeichnet, und als noch ausstehende Punkte in einer ToDo-Liste notiert.

Nachdem alles ausgefüllt ist, kann der Product Owner die Beschreibung in einen Vorgang übertragen und die noch ausstehenden Punkte dort mit Leben füllen, wobei er sich an dieser ToDo-Liste orientieren kann. Durch das Skript wurde sichergestellt, dass alle von ihm zu treffenden Entscheidungen berücksichtigt wurden und explizit als Teil des Vorgangs aufgelistet sind.

Das explizite Auflisten solcher Details ist ein wichtiger Punkt zur Sicherstellung der Qualität. Die Verschriftlichung hilft zuallererst dem Produkt Owner selbst, der dadurch nochmal eine bessere Vorstellung von der von ihm beauftragten Entwicklung bekommt. In gleicher Weise hilft sie dem Entwickler der sich bei der gesamten Entwicklung klar an diesen Punkten orientieren kann, weniger Rückfragen stellen muss und die Liste auch nach Fertigstellung zur Gegenprüfung und zum Abhaken aller Punkte verwenden kann. Auch bei der Abnahme durch den Product Owner kann die explizite Auflistung der einzelnen Detailanforderungen unterstützend eingesetzt werden.

Diese Form von Entwicklungsunterstützung scheint alles andere als agil zu sein und passt nicht zu den blumig beschriebenen User Stories im Srum-Prozess und dem permanenten Austausch zwischen Entwicklung und Product Owner. Hier muss man wieder ganz klar unterscheiden zwischen der Entwicklung von Framework und der Verwendung des entwickelten Frameworks. Die Entwicklung von neuem Framework kann und sollte agil verlaufen, da Anforderungen und technische Möglichkeiten permanent aufeinander abgestimmt werden müssen. Hier ist viel Aufwand und viel Austausch notwendig, damit das neu Geschaffene am Ende zum Bestehenden passt und systematisch, ohne großen Abstimmungsbedarf verwendet werden kann. Die Erweiterung des Designsystems, die Aufstellung von Regeln für die Verwendung und idealerweise auch die Erstellung von Unterstützungskripten sollten fester Bestandteil bei der Entwicklung neuer Framework-Komponenten sein.

9.5 Qualitätprotokolle

Qualitätsprotokolle sind ähnlich wie die Unterstützungskripte Hilfsmittel, die den am Entwicklungsprozess beteiligten Mitarbeitern helfen, ihre Arbeit einfacher und nach einer definierten Vorgehensweise durchzuführen. Letzteres steht hierbei im Vordergrund, denn das Protokoll macht Vorgaben, die zwingend einzuhalten sind, und die auch Gegenstand von internen Richtlinien sein können.

Insbesondere dann, wenn man als Unternehmen eine Zertifizierung nach den Normen ISO 9001 oder ISO 25001 anstrebt und die damit verbundenen Audits bestehen möchte, reicht es nicht, Qualität, Datenschutz und Datensicherheit einfach nur zu wollen, man muss Maßnahmen im Management zur permanenten Aufrechterhaltung diese Aspekte definieren, dokumentieren, einhalten und auch nachweisen können. Dies gelingt langfristig nur dadurch, dass man diese Dinge tatsächlich in den Arbeitsalltag integriert und lebt, und

den Nutzen zu schätzen lernt, den diese für den Kunden und auch das Unternehmen selbst haben.

Je mehr man diese Integration auf den tatsächlich beabsichtigten Effekt ausrichtet, anstatt nur nach einer formalen Erfüllung der verlangten Punkte zu streben, desto einfacher fällt es am Ende, diese Dinge in der Routine zu berücksichtigen und einen wirklichen Mehrwert daraus zu generieren.

Qualitätsprotokolle sind ein einfaches Mittel dafür und lassen sich an vielen Stellen einrichten, am besten dort, wo regelmäßige Tätigkeiten auftreten, bei denen es wichtige Dinge zu beachten gibt.

Als Fortsetzung von Abschnitt 9.4 beginnen wir mit einem Protokoll für den Product Owner, das dabei hilft, bestimmte Wirkungen der von ihm beauftragten Entwicklungen zu beleuchten, und sich damit auseinanderzusetzen.

Mit einem Fehler im Kontext der Softwareentwicklung assoziieren wir gemeinhin eine fehlerhafte Abweichung vom Sollzustand, die bei der Entwicklung entstanden ist, aber es gibt auch Fehler die schon vorher entstehen und die als Teil der Anforderungen den gesamten Entwicklungsprozess bis hin zum Test durchlaufen können, ohne dass jemand merkt, dass es falsch ist. Oft geschieht so etwas durch einen unbedachten Umgang mit personenbezogenen Daten und führt am Ende zu einem meist kleinen, aber dennoch ungewollten Datenschutzverstoß der hätte vermieden werden können.

Ein klassisches Beispiel dafür ist der Wunsch, das Geburtsdatum als Spalte für einen Auswahldialog für Personen hinzuzunehmen. Der Grund dafür ist gut nachvollziehbar, und mit Sicherheit erleichtert diese zusätzliche Information die Auswahl an verschiedenen Stellen, und hilft besonders beim Vorhandensein von gleichnamigen Personen. Andererseits wird die personenbezogene Information des Geburtsdatums an Stellen im Programm abrufbar, wo es primär nicht um die Stammdaten der Personen geht, und damit erweitert sich der Zugriff auf diese Information eventuell auf Benutzer, die diesen zuvor nicht hatten.

Die Gefahr, solche Dinge zu übersehen, ist recht groß, da sie oft nebenläufig zu den eigentlich ein Anforderungen stehen. Insofern ist es gut, ein Protokoll zu haben, dass solche Aspekte einzeln und jedes mal abfragt.

Der nachfolgende PO-Assistent enthält beispielhaft drei Kontrollfragen aus den Bereichen Datenschutz und Datensicherheit, die der Product Owner nach Beendigung seiner Story-Formulierung durchgehen sollte.

Beispiel 35 – PO-Assistent

```
?Q1: Werden durch die Entwicklung Kundendaten
    __ außerhalb der Programmumgebung gespeichert?
    #j: Ja
        @Wirkungen >> Speicherung von Kundendaten
        __ außerhalb der Programmumgebung: Ja
        @Ergänzungsmaßnahmen >> Speicherung von Kundendaten
        __ außerhalb der Programmumgebung:
        > Sicherstellen, dass der Speicherort so gewählt
        __ werden kann, dass kein unautorisierter Zugriff
        __ möglich ist
```

```
        > Sicherstellen, dass Speichervorgang und Speicherort
        __ für den Anwender gut erkennbar sind
    #n: Nein
        @Wirkungen >> Speicherung von Kundendaten
        __ außerhalb der Programmumgebung: Nein
?Q2: Werden durch die Entwicklung personenbezogene Daten an
    __ Stellen im Programm sichtbar wo sie es bisher nicht waren?
    #j: Ja
        @Wirkungen >>  Erweiterte Sichtbarkeit
        __ personenbezogener Daten: Ja
        @Ergänzungsmaßnahmen >> Erweiterte Sichtbarkeit
        __ personenbezogener Daten:
        > Sicherstellen, dass der Zugriff auf diese Stellen durch
        __ die Vergabe von Berechtigungen abgesichert ist
        > Sicherstellen dass nicht mehr Daten angezeigt werden
        __ als in diesem Bearbeitungskontext notwendig ist
        > Ggf. Datenschutzbeauftragten mit einbeziehen
    #n: Nein
        @Wirkungen >>  Erweiterte Sichtbarkeit
        __ personenbezogener Daten: Nein
?Q3: Werden durch die Entwicklung neue benutzerbezogene
    __ Daten protokolliert oder verarbeitet?
    #j: Ja
        @Wirkungen >> Protokollierung von Nutzerdaten: Ja
        @Ergänzungsmaßnahmen >> Protokollierung von Nutzerdaten:
        > Sicherstellen dass die Protokollierung zweckgebunden
        __ und durch die Benutzervereinbarung abgedeckt ist
        > Sicherstellen, dass die Protokollierung in das
        __ Datenblatt für Protokollierung aufgenommen wird
    #n: Nein
        @Wirkungen >> Protokollierung von Nutzerdaten: Nein
```

Die Ausgabe besteht aus zwei Abschnitten. Der Abschnitt *Ergänzungsmaßnahmen* beschreibt Maßnahmen, die im Zusammenhang mit der Story sichergestellt werden sollten, z.B. indem entsprechende Anforderungen ergänzt werden. Der Abschnitt *Wirkungen* dient als Protokoll, dass in jedem Fall in der Story hinterlegt wird. Damit wird nachweisbar sichergestellt, dass der Assistent für alle eingeplanten Stories durchlaufen wurde, und man kann im Falle eines Fehlers besser nachvollziehen, wo dieser entstanden ist, und wie man diesem zukünftig vorbeugen kann.

Wie schon erwähnt, kann man solche Protokolle an vielen Stellen verankern, und natürlich gibt es auch auf technischer Ebene sehr viele Dinge, die vom Entwickler berücksichtigt werden müssen.

Viele Teams praktizieren Verfahren wie das Vier-Augen-Prinzip oder eine Form von Peer-Review, bei der abgeschlossene Entwicklungen einem Kollegen zur Prüfung vorgelegt werden. Die verschärfte Varianten davon nennt sich *Smash It*. Hier versucht der angefragte Kollege gezielt, und unter Nutzung seines Systemwissens, Fehler und Schwachstellen der Entwicklung aufzudecken, damit diese vor Herausgabe der Version behoben werden können. Dies sind gute Ergänzungen zu den Integrationstests, die ja primär auf

das Abprüfen der neuen Funktionalität ausgerichtet sind und nicht auf das Auffinden von Schwachstellen in deren Umgebung. Trotzdem hängt es am Ende hautsächlich von der Erfahrung und der investierten Zeit des Einzelnen ab, ob ein kritischer Fehler oder eine Verwundbarkeit entdeckt wird oder nicht.

Im Laufe der Zeit und vor allem durch die systematische Untersuchung der leider doch nicht rechtzeitig erkannten schwerwiegenden Fehler stellt man fest, dass jedes Entwicklungsmuster seine eigenen Risikofaktoren mit sich bringt, und es in den meisten Fällen systematische Vorgehensweisen gibt, um die jeweiligen Risiken zu verringern.

Das Risiko, dass Daten unbeabsichtigt gelöscht werden, wird dann auftreten, wenn überhaupt Daten gelöscht werden, Kompatibilitätsprobleme bekommt man dann, wenn man mit versionsabhängigen Formaten hantiert. Probleme mit Drittkomponenten zeigen sich oft erst Monate oder Jahre später, entstehen aber in dem Moment wo man sich nicht genügend Zeit nimmt, um den Einbau einer neuen Drittkomponente sorgsam auszuführen.

Das nachfolgende Quali-Assistent-Protokoll ist ähnlich aufgebaut wie Beispiel 35 und zeigt anhand der oben genannten Punkte, wie ein Entwickler seine Entwicklung durch zusätzliche Maßnahmen absichern kann, wenn sie bestimmte Aspekte erfüllt.

Beispiel 36 – Quali-Assistent

```
?Q1: Umfasst die Entwicklung Funktionen, die Datensätze löschen?
    #j: Ja
        @Wirkungen >> Löschen von Datensätzen: Ja
        @Ergänzungsmaßnahmen >> Löschen von Datensätzen:
        > Sicherstellen, dass keine Datensätze ungewollt
        __ gelöscht werden.
        > Abgrenzungstests formulieren
        > Löschweitergaben prüfen
    #n: Nein
        @Wirkungen >> Löschen von Datensätzen: Nein
?Q2: Umfasst die Entwicklung die Verwendung
    __ neuer Drittkomponenten?
    #j: Ja
        @Wirkungen >> Verwendung von Drittkomponenten: Ja
        @Ergänzungsmaßnahmen >> Verwendung von Drittkomponenten:
        > Checkliste für Drittkomponenten durchgehen
        > Drittkomponente in eigene Klasse kapseln
    #n: Nein
        @Wirkungen >> Verwendung von Drittkomponenten: Nein
?Q3: Umfasst die Entwicklung die Verarbeitung von
    __ versionsabhängigen Formaten?
    #j: Ja
        @Wirkungen >> Versionsabhängige Formate: Ja
        @Ergänzungsmaßnahmen >> Versionsabhängige Formate:
        > Sicherstellung der Versionsbestimmbarkeit
        > Sicherstellung von Abwärtskompatibilität
        > Sicherstellung von Aufwärtskompatibilität
        > Kennzeichnung der Versionsabhängigkeit
```

```
    __ im Datenblatt.
#n: Nein
    @Wirkungen >> Versionsabhängige Formate: Nein
```

Bei der Implementierung einer Funktion, die Datensätze löscht, wird meist nur geprüft, ob das zu Löschende auch wirklich gelöscht wird. Viel wichtiger ist die Überprüfung, ob das, was nicht gelöscht werden soll, auch wirklich nicht gelöscht wird. Hierfür kann man Abgrenzungstests formulieren und man kann man sich – auch mit *FlowProtocol 2* – einfache Werkzeuge bauen, die Abweichungen der Datensatz-Anzahl tabellenübergreifend erkennbar machen. Oft wird auch übersehen, dass eine Löschung durch festgelegte Löschweitergaben weitere Löschungen nach sich zieht. Diese sollten immer explizit geprüft werden.

Der Umgang mit Drittkomponenten umfasst viele Aspekte, über die man einen eigenen Abschnitt schreiben könnte. Gerade im kommerziellen Umfeld gibt es viel zu beachten, angefangen von der Verträglichkeit der Lizenz, bis hin zur Dokumentation und Planung notwendiger Überwachungsaufgaben für die Sicherstellung der Aktualität und regelmäßige Prüfung auf Verwundbarkeiten. Komponenten, die für sicherheitsrelevante Funktionalität wie Verschlüsselung eingesetzt werden, müssen diesbezüglich besondere Ansprüche erfüllen. Im Beispiel oben wird dementsprechend auch nur auf eine Checkliste für Drittkomponenten verwiesen, die man natürlich ebenfalls mit *FlowProtocol 2* abbilden kann.

Durch die Kapselung einer Drittkomponente in eine eigenen Klasse lässt sich gut sicherstellen, dass nur die benötigte Funktionalität verwendet wird und die Konfiguration auf die Eigenschaften beschränkt bleibt, die man zulassen möchte.

Beim Umgang mit versionsabhängigen Formaten sollte von Anfang an berücksichtigt werden, dass sich diese ändern können, selbst wenn sie Teil des Produktes sind. Wenn ein Format um neue Eigenschaften erweitert wird, sollten vorhandene Daten in einem älteren Format immer noch verwendbar bleiben. Insbesondere sollte ein Format immer explizit bestimmbar sein, und nicht durch das Vorhandensein oder Nicht-Vorhandensein von Eigenschaften hergeleitet werden müssen.

Auch hier ist das Skript wieder so aufgebaut, dass parallel zu den ergänzenden Maßnahmen ein Protokoll mit den Wirkungen der Entwicklung ausgegeben wird, das zum Nachweis der Auseinandersetzung in der Aufgabe hinterlegt werden könnte.

Die Liste solcher Aspekte ließe sich noch um viele Punkte verlängern, und man muss aufpassen, dass die Relevanzklärung noch mit überschaubarem Aufwand möglich ist. Besonders sollte man darauf achten, dass die Fragestellungen nach bestimmten Aspekten grundsätzlich unmittelbar, also ohne langes überlegen durch den Entwickler beantwortet werden kann, zumindest wenn er sich gerade ausführlich mit der Aufgabenstellung auseinandergesetzt hat.

10 Organisationswerkzeuge

Im Drumherum der täglichen Arbeit gibt es viele kleine Aufgaben, die sich oft mehrfach täglich wiederholen, und bei denen man froh ist, wenn man auf passende Hilfswerkzeuge zurückgreifen kann. In diesem Abschnitt beschreiben wir einige solche Werkzeuge, die sich mit wenig Aufwand als *FlowProtocol-2*-Skript anfertigen lassen.

10.1 Zeitmessung

Zeitmessung ist ein mächtiges Instrument, und meist unerlässlich, insbesondere wenn man direkt für einzelne Kunden arbeitet. Da Zeitmessung oftmals mit Leistungsmessung assoziiert wird, und damit auch als Argument für Kritik an der eigenen Leistungsfähigkeit herangezogen werden kann, tut man sich oft schwer, diese für verschiedene Tätigkeiten zu etablieren.

Umgekehrt finden wir durch Zeitmessung heraus, in welche Arten von Aufgaben unsere Arbeitszeit fließt, und können auf dieser Grundlage beurteilen, ob sie dort auch in ausreichendem Maße zur Wertschöpfung beiträgt. Stellen wir etwa fest, dass viele Stunden Zeit für Supportunterstützung aufgewendet werden müssen, weil die Benutzerführung missverständlich ist, können wir gezielt gegensteuern und mehr Zeit für dahingehende Maßnahmen investieren.

Darüber hinaus erlaubt uns die Zeitmessung das sogenannte Controlling, also die permanente Überprüfung ob jüngere Anpassungen der Arbeitsweise den beabsichtigten Effekt haben.

Aufgrund der großen Bedeutung ist Zeitmessung in vielen Systemen fest integriert und muss nicht durch ein zusätzliches Hilfsmittel ergänzt werden. Das nachfolgende Beispiel zeigt eine einfache Zeitmessung über ein *FlowProtocol-2*-Skript. Dieses ermöglicht es, die Zeitmessung zu pausieren und fortzusetzen und gibt am Ende ein minutengenaues Zeitprotokoll aus.

Dadurch, dass man das Skript parallel in mehreren Browser-Tabs öffnen kann, kann man so auch zwischen der Zeitmessung für verschiedenen Aufgaben hin und her wechseln.

Beispiel 37 – Zeitmesser

```
~Input A: Aufgabe
~SetInputTitle Zeitmessung für Aufgabe $A
~Set idx=0
~GoSub NotiereZeit
~Set Status=läuft
~Set Aktion=pausieren
~Set Summe=0
~Set fertig=Nein
~DoWhile $fertig!=Ja
    ~Set pri=$idx
    ?A($idx): Zeitmessung $Status (Summe: $Summe Minuten)
        #P: Zeitmessung $Aktion
        #A: Zeitmessung abschließen
            ~Set fertig=Ja
    ~Execute
    ~GoSub NotiereZeit
    ~If $Status==läuft
        ~Replace ZTerm=$ZP($idx) - ($ZP($pri))|:-> * 60 +
        ~CalculateExpression Diff = $ZTerm
        ~AddTo Summe += $Diff
        @Zeitprotokoll für Aufgabe $A >>|$ZP($pri)-$ZP($idx) Uhr
```

```
            __ - $Diff Minuten
        ~Set Status=pausiert
        ~Set Aktion=fortsetzten
    ~Else
        ~Set Status=läuft
        ~Set Aktion=pausieren
~Loop
@Zeitprotokoll für Aufgabe $A >>|Summe: $Summe Minuten

~DefineSub NotiereZeit
    ~AddTo idx+=1
    ~SetDateTime Zeit = HH:mm
    ~AddNewKey ZP($idx) = $Zeit
~Return
```

Das Skript funktioniert wie folgt: nach Abfrage der Aufgabe, damit diese als Titel dargestellt werden kann, wird wiederholt eine Schleife durchlaufen, bis die Zeitmessung abgeschlossen wird. In jedem Schleifendurchlauf hat man die Wahl zwischen *Zeitmessung pausieren* und *Zeitmessung abschließen* oder *Zeitmessung fortsetzen* und *Zeitmessung abschließen*, je nachdem ob die Zeitmessung gerade läuft oder pausiert. Die Implementierung entspricht also einem Kippschalter mit dem zwischen diesen beiden Zuständen hin und her geschaltet werden kann.

Bei jeder Aktion wird die Funktion `NotiereZeit` durchlaufen, die mit Hilfe des Befehls `~SetDateTime` die Uhrzeit notiert und diese mit `~AddNewKey` als Teil der URL speichert, wobei der fortlaufenden Schlüssel `ZP(...)` verwendet wird.

Der Befehl `~SetDateTime` speichert die Systemzeit in einem vorgegebenen Format in einer Variable und der Befehl `~AddNewKey` funktioniert so, dass er den Wert nur dann als Teil der URL speichert, wenn dort noch kein Wert für diesen Schlüssel vorhanden ist. Er verhält sich damit analog zu den Auswahlabfragen und Texteingaben, die auch nur dann eine Eingabe verlangen, wenn der Wert noch nicht in der URL vorhanden ist. Die Zeitpunkte aller Aktionen werden damit in der URL gespeichert.

Bei aktuell laufender Zeitmessung wird aus den Uhrzeiten zwischen letzter und aktueller Aktion die Zeitdifferenz in Minuten berechnet und zu einer Summe aufaddiert. Für die Differenzberechnung wird dabei etwas trickreich der Doppelpunkt der Uhrzeiten einfach durch die Zeichenfolge `* 60 +` ersetzt, sodass sich beispielsweise aus dem Ausdruck `12:36 - (11:51)` der Rechenterm $12 \cdot 60 + 36 - (11 \cdot 60 + 51)$ ergibt, der die Differenz der Uhrzeiten in Minuten, also 45 berechnet.

Bei Beendigung des Skriptes mittels *Zeitmessung abschließen* wird dann das vollständige Zeitprotokoll mit den Uhrzeiten, den Differenzminuten und der Gesamtzeitdauer ausgegeben.

10.2 Meeting-Vorbereitung

Auch das nächste Beispiel verwendet Befehle zur Abfrage der Systemzeit um eine Ausgabe zu erzeugen, in diesem Fall eine Überschrift für das Protokoll des nächsten Jour fixe, der an jedem dritten Dienstag stattfindet.

Es ist sinnvoll, Themen für regelmäßige Besprechungen im Vorfeld zu sammeln, was man am besten in einem Wiki-Bereich macht, in dem man für jedes Meeting eine eigene Protokollseite anlegt, die nach einer festen Konvention mit dem Datum als Teil der Überschrift benannt wird.

Wenn einem nun aktuell ein Thema einfällt, das man für das kommende Meeting dort notieren möchte, muss dort gegebenenfalls zuerst die entsprechende Seite mit Überschrift angelegt werden. Das dazugehörende Datum findet man sicher im Kalender, wo das nächste Meeting als Teil einer Terminserie eingetragen ist, und für die Konvention der Überschrift kann man sich an den letzten Einträgen im Wiki orientieren. Einfacher ist es jedoch, das nachfolgen des Skript aufzurufen, das Datum und Überschrift für das aus heutiger Sicht nächste Meeting anhand der Terminregeln automatisch berechnet und bei Bedarf auch noch direkt auf den entsprechenden Wiki-Bereich verlinken könnte.

Beispiel 38 – Jour-fixe-Planer

```
~SetCulture de-DE
~SetDateTime dheute=yyyy-MM-dd
~SetDateTime Jahr = yyyy
~DateSet tag = 06.01.$Jahr|dd.MM.yyyy
~Set z=0
~Set nr=0
~DoWhile true
    ~DateAdd tag = $tag|1|d
    ~DateFormat dvgl=$tag|yyyy-MM-dd
    ~If $dvgl == $dheute
        ~Set zukunft=ja
    ~DateFormat wotag=$tag|ddd
    ~If $wotag==Di
        ~AddTo z+=1
        ~If $z==3
            ~Set z=0
            ~AddTo nr+=1
            ~If $zukunft==ja
                ~ExitLoop
~Loop
~DateFormat erg=$tag|dd.MM.yyyy
@jour fixe WM-XX >>* Wiki-Bereich
    ~AddLink ... | jour fixe WM-XX
    ~AddText  öffnen.
>>* Falls nicht vorhanden, neue Seite anlegen:
    >|$erg - Jour fixe WM-XX (Nr. $nr/$Jahr)
>>* (findet an jedem 3ten Dienstag statt)
```

Das Beispiel des *Jour-fixe-Planers* zeigt auch noch weitere Befehle für den Umgang mit Datumsangaben, angefangen mit `~SetCulture`, mit dem man die Lokalisierungseinstellungen für die Ausführung setzen kann, die sich insbesondere auch auf die Datums- und Uhrzeitformate auswirken.

In unserem Beispiel werden mit `de-DE` deutschsprachige Einstellungen gesetzt. Danach wird aus der Systemzeit das heutige Datum und zusätzlich auch noch die Jahreszahl abgeleitet. Im Anschluss wird mit dem `~SetDateTime`-Befehl die Variable `tag` auf den 06.01. des aktuellen Jahres gesetzt, wobei das Format, in dem der zu setzende Wert vorliegt, ebenfalls angegeben werden muss.

Die eigentliche Berechnung erfolgt in einer Schleife, in der die `tag`-Variable mit dem Befehl `~DateAdd` sukzessive um einen Tag hochgezählt wird. Für den Vergleicht wird dieses Datum mit dem `~DateFormat`-Befehl auf das gleiche Format gebracht wie `dheute`. Sobald das aktuelle Datum erreicht wird, wird die Variable `zukunft` auf `ja` gesetzt, danach wird der Wochentag über das Format `ddd` bestimmt.

In der verschachtelten If-Abfrage wird zuerst geprüft, ob es sich um einen Dienstag handelt, und wenn ja, wird ein Zähler `z` bis auf maximal drei hochgezählt. Sobald ein dritter Dienstag gefunden ist, der in der Zukunft liegt, wird die Schleife mit `~ExitLoop` verlassen. Die Termine werden zusätzlich über das Jahr hinweg durchgezählt.

Das gefundene Datum wird dann zusammen mit der laufenden Nummer in die Überschrift für die Protokollseite eingebaut und als Codeblock ausgegeben, so dass alles direkt in die Zwischenablage kopiert werden kann.

Auch dieses Skript lässt sich nach Belieben anpassen z.B. durch Variation der Terminregeln, die zusätzliche Ausgabe der letzten vergangenen oder weiterer zukünftigen Termine, durch Ausgabe eines Such-Links, ähnlich wie in Beispiel 28, der direkt nach der entsprechenden Seite im Wiki sucht, oder durch die Ausgabe einer Standard-Agenda, die man auf die neu erstellte Seite übernehmen kann. Diese könnte sogar als Jahres-Agenda in einer Datei verwaltet werden und die wiederkehrenden Punkte zum richtigen Zeitpunkt auf den Plan bringen.

Umso einfacher Dinge sind, umso eher und öfter wird gebraucht davon gemacht. Meist sind es die kleinen Hürden, die nicht genommen werden.

10.3 Asset-Dokumentation

In einem Softwareunternehmen sollten verschiedene Arten von Assets dokumentiert werden, um die Ressourcen, das geistige Eigentum und die Infrastruktur des Unternehmens zu verwalten und zu schützen. Die Dokumentation sollte gleichermaßen einen Überblick über die vorhandenen Assets geben, als auch die damit verbundenen Aufgaben beschreiben. Je nach Art des Assets sind unterschiedliche Eigenschaften relevant, die jedoch insgesamt in einheitlicher Form dokumentiert werden sollten.

Sofern dafür nicht eine spezialisierte Asset-Datenbank verwendet wird, ist das Firmenwiki der richtige Ort, um diese Informationen zu verwalten. *FlowProtocol 2* kann dabei helfen, die je nach Art benötigten Detailinformationen interaktiv abzufragen und daraus das Datenblatt und die Liste der regelmäßigen oder aktuell noch ausstehenden Aufgaben zu erstellen.

Das dazugehörende Beispiel beginnt mit der Abfrage der allgemeinen Rahmendaten des Assets, zu denen Bezeichnung, Verwendungszweck und vor allem die Art gehören, von der die Menge der weiteren Eigenschaften abhängt. Das am Ende erstellte Dokument wird Teil des integrierten Managementsystems (IMS) und sollte demzufolge auch einen Dokumenteigentümer haben, der das Dokument regelmäßig sichtet und aktuell hält.

Beispiel 39 – Asset-Dokumentation (Teil 1)

```
~Input Bez: Bezeichnung des neuen Assets
?Asset: Welche Art von Asset soll dokumentiert werden?
    #: Drittanbieterkomponente
    #: Dienst (produktiv)
    #: Dienst (intern)
        ~Set ArtAsset=Dienst (intern)
        ~Set WirdProduktivEingesetzt=Nein
        ~Set HatLizenz=Ja
        ~Set HatAufrufURL=Ja
        ~Set HatLokaleAdministratoren=Ja
        ~Set HatLokaleInstallation=Nein
        ~Set IstEinCloudDienst=Ja
        ~Set IstInformationsquelle=Nein
        ~Set IstEinGeraet=Nein
        ~Set SetztFachwissenVoraus=Ja
    #: Server (produktiv)
    #: Server (intern)
    #: Datenblatt
    #: Software
    #: Anleitung/Beschreibung
    #: Wissensdokumentation
    #: Manueller Prozess
    #: Mobiles Endgerät

~Input DEig: Dokumenteigentümer (Mitarbeiterkürzel)
~Input VZweck: Wozu wird dass Asset verwendet?
~SetTitle Asset "$Bez"
@Datenblatt >>* Dokumenteigentümer: $DEig
@Offboarding >> Dokumenteigentümer prüfen
@Datenblatt >>* Art: $ArtAsset
@Datenblatt >>* Verwendungszweck
    >* $VZweck
@Jährl. Aufgaben >> Verwendung/Bedarf prüfen
@Außerbetriebenahme >> Asset-Dokumentation archivieren
~Execute
```

Je nach Art gibt es bestimmte Eigenschaften oder Aspekte, die für die Dokumentation relevant sind oder sein können, und die abgefragt werden müssen. Ob eine Eigenschaften für eine Art relevant ist oder nicht, wird über eine entsprechend benannte Variable gesetzt, die im späteren Verlauf abgefragt werden. Durch die Verwendung dieser Schalter muss jede Eigenschaft oder jeder Aspekt nur einmalig im Skript implementiert werden und die Reihenfolge bleibt einheitlich.

Im Beispiel oben ist nur der Fall des intern genutzten Dienstes ausimplementiert. Für alle anderen Typen würde man entsprechende Schalter setzen, z.B. für den Typ *Software* die Eigenschaft `HatLokaleInstallation=Ja`.

Die Implementierung der einzelnen Aspekte sieht prinzipiell so aus, dass man die dafür

benötigten Eigenschaften abfragt und im Datenblatt auflistet. Hier können wieder untergeordnete Aspekte eine Rolle spielen, wie z.B. verschiedene Arten von Lizenzen, die man ebenfalls mit Schaltern abbilden kann. Für Assets mit Lizenz sind im Beispiel schon mal zwei Arten hinterlegt:

Beispiel 40 – Asset-Dokumentation (Teil 2)

```
~If $HatLizenz==Ja
    ?Liz: Welche Art von Lizenz liegt vor?
        #: Frei, kommerziell nutzbar
            ~Set ArtLizenz=Frei, kommerziell nutzbar
            ~Set KommerziellNutzbar=Ja
            ~Set Benutzerbezogen=Nein
            ~Set Kostenpflichtig=Nein
        #: Abo-Lizenz, benutzerbezogen
            ~Set ArtLizenz=Abo-Lizenz, benutzerbezogen
            ~Set KommerziellNutzbar=Ja
            ~Set Benutzerbezogen=Ja
            ~Set Kostenpflichtig=Ja
    @Datenblatt >>* Lizenz
        >* Art: $ArtLizenz
        >* Kommerziell nutzbar: $KommerziellNutzbar
        >* Benutzerbezogen: $Benutzerbezogen
        >* Kostenpflichtig: $Kostenpflichtig
    ~If $Benutzerbezogen==Ja
        @Jährl. Aufgaben >> Lizenzumfang abgleichen
        @Offboarding >> Lizenzumfang anpassen
    ~If $Kostenpflichtig==Ja
        @Jährl. Aufgaben >> Preis-Leistung prüfen
        @Außerbetriebenahme >> Lizenz kündigen
        ~Set ALGegenstand=Lizenzverwaltung
        ~Set ALKey=LIV
        ~GoSub AssetLink
```

Die meisten Informationen landen im Datenblatt. Bei benutzerbezogenen und kostenpflichtigen Lizenzen leiten sich zusätzlich Aufgaben für die jährliche Überprüfung, das Offboarding und die Außerbetriebnahme ab. So kann z.B. eine Lizenz zurückgegeben oder stillgelegt werden, wenn der nutzende Mitarbeiter das Unternehmen verlässt. Der Funktionsaufruf `AssetLink` am Ende wird in diesem Skript noch häufiger verwendet. Er ruft die folgende Funktion auf:

Beispiel 41 – Asset-Dokumentation (Teil 3)

```
~DefineSub AssetLink
    ?QAL$ALKey: Ist ein Asset für "$ALGegenstand" vorhanden?
    #j: Ja
        ~Input AL$ALKey: Link auf Asset für "$ALGegenstand"
        @Datenblatt > $ALGegenstand:
```

```
        ~AddLink $AL$ALKey| als Asset erfasst
    #n: Nein
        @Datenblatt > $ALGegenstand:
        @ToDos >> Asset für $ALGegenstand erfassen
~Return
```

Diese fragt eine bestimmte variable Eigenschaft des Assets ab, die vorzugsweise wieder als eigenes Asset angelegt sein sollte, so dass mittels Link darauf verwiesen werden kann. In diesem Fall ist es die Lizenzverwaltung, also die Information über die Anzahl, Art und Zuteilung der vorhandenen, kostenpflichtigen Lizenzen. Um die Komplexität überschaubar zu halten, ist es sinnvoll, möglichst viele Teilsysteme als eigene Assets zu erfassen und jeweils darauf zu verweisen. Auf diese Weise spart man sich die verschachtelte Abfrage der damit verbundenen Eigenschaften und Aufgaben und bekommt eine sehr homogene Dokumentation.

Wenn man mit einer derartigen Dokumentation beginnt, muss man davon ausgehen, dass einige der abgefragten Dinge zunächst nicht vorhanden sind und der Ist-Zustand des Assets nicht alle Anforderungen erfüllt. Entsprechend sollte man diese Möglichkeit in den Auswahlabfragen grundsätzlich berücksichtigen. Wenn dann auf diese Weise festgestellt wird, dass wichtige Dinge fehlen, kann daraus eine Aufgabe in einem ToDo-Abschnitt abgeleitet werden. In gleicher Weise leiten sich aus bestimmten Eigenschaften oder Antworten Aufgaben ab, die regelmäßig, z.B. jährlich im Zusammenhang mit dem Asset durchgeführt werden sollten, und solche, die dann notwendig werden, wenn die mit dem Asset verbundenen Mitarbeiter das Unternehmen verlassen. Am Ende gibt es auch noch Aufgaben, die bei der Außerbetriebnahme des Assets durchzuführen sind, wie die Archivierung der Asset-Dokumentation selbst.

Für die restlichen oben gesetzten Schalter könnte die Erfassung in sehr vereinfachter Form so aussehen:

Beispiel 42 – Asset-Dokumentation (Teil 4)

```
~If $HatAufrufURL==Ja
    ~Input AufrufURL: Unter welcher URL wird der Dienst aufgerufen?
    @Datenblatt >>* Aufruf
    >* URL:
    ~AddLink $AufrufURL | $AufrufURL
    @Jährl. Aufgaben >> Verfügbarkeit prüfen
~If $HatLokaleAdministratoren==Ja
    ~Input Admins: Mitarbeiter mit Admin-Zugang und -kenntnissen:
    @Datenblatt >>* Administration
        > Mitarbeiter mit Zugang und Wissen: $Admins
    ~Set ALGegenstand=Admin-Wissen
    ~Set ALKey=ADW
    ~GoSub AssetLink
    @Offboarding >> Admin-Zugang übergeben
    ~Set ALGegenstand=Admin-Notfallzugang
    ~Set ALKey=ANZ
    ~GoSub AssetLink
~If $IstEinCloudDienst==Ja
```

```
    ~Input AnbCloudName: Anbieter des Cloud-Dienstes
    ~Input AnbCloudWeb: Internetseite Anbieter
    @Datenblatt >>* Anbieter
    >* Name: $AnbCloudName
    >* Webseite:
    ~AddLink $AnbCloudWeb| $AnbCloudWeb
    @Jährl. Aufgaben >> Weitere Verfügbarkeit prüfen
    @Datenblatt >>* Risiko: Wegfall des Dienstes
    ~Set ALGegenstand=Wegfall-Notfallplan
    ~Set ALKey=WNP
    ~GoSub AssetLink
    @Außerbetriebenahme >> Konfigurationsdaten sichern
~If $SetztFachwissenVoraus==Ja
    ~Set ALGegenstand=Nutzungswissen
    ~Set ALKey=WFN
    ~GoSub AssetLink

~MoveSection Jährl. Aufgaben->Jährliche Aufgaben
~MoveSection Offboarding->Offboarding-Aufgaben
~MoveSection Außerbetriebenahme->Aufgaben bei Außerbetriebenahme
~MoveSection ToDos->Aktuelle ToDos
```

Die Menge der abgefragten Informationen sollte sich primär daran orientieren, was in Ausnahmesituationen benötigt wird, um den Kernbetrieb am Laufen zu halten. Auch das Ausfallrisiko auf Mitarbeiterseite ist nicht zu unterschätzen. Mitarbeiter, die zentrales Wissen oder Zugänge zu essentiellen Diensten haben, können verunfallen und für längere Zeit vollständig ausfallen. Insofern sollte immer darauf geachtet werden, dass relevantes Wissen festgehalten wird und dass essentielle Zugänge notfallmäßig hinterlegt werden.

Die vier `~MoveSection`-Befehle am Ende stellen sicher, dass die einzelnen Abschnitte immer in der gleichen Reihenfolge aufgelistet werden. Zusätzlich wird dabei die Abschnittsüberschrift komplett ausgeschrieben.

Das Ergebnisdokument kann mit Überschrift und allen Abschnitten auf eine eigene Wiki-Seite innerhalb der Asset-Dokumentation übernommen werden. Der vorzugsweise nach Art gegliederte Unterbereich innerhalb der Dokumentation und die genaue Vorgehensweise könnten hierbei in einem eigenen Anleitungsabschnitt beschrieben werden. Ebenso sollte man die Abfragen und Texteingaben im Skript noch durch zusätzliche Hilfezeilen selbsterklärender gestalten, so dass man am Ende ein Skript bekommt, mit dem die jetzt schon für das entsprechende Asset verantwortlichen Mitarbeitern direkt in die Lage versetzt werden, diese in der gewünschten Form zu dokumentieren.

Diese Dokumentation ermöglicht es dann, jährlich die vorhandenen Assets zu sichten und auf Notwendigkeit zu prüfen. Nicht selten werden Vorgehensweisen oder Softwareprodukte eingeführt, die sich dann irgendwann von selbst ausschleichen, ohne dass eine Außerbetriebnahme stattgefunden hat. Das ist vor allem dann unschön, wenn weiterhin Geld dafür bezahlt wird. Die jährliche Sichtung der Assets, zusammen mit den notwendigen Wartungsaufgaben ist wichtig, um z.B. rechtzeitig darüber informiert zu sein wenn, etwa aufgrund einer Abkündigung, Nutzungseinschränkungen absehbar werden.

Gerade solche Rechercheaufgaben nach neuen oder abgekündigten Versionen lassen sich heute – sofern es die Unternehmensrichtlinien zulassen – mit einer einfachen Anfrage an

einen Chatbot erledigen, z.B. *Für welche Versionen des Windows Server Betriebssystems läuft der Support als nächstes aus und gibt es schon Ankündigungen für neue Versionen?* Die Erzeugung des passenden Eingabe-Promptes auf Basis von Komponentenname und Komponententyp lässt sich dabei sehr gut in ein *FlowProtocol-2*-Skript für diese Art von Wartungsaufgaben integrieren, dessen Aufruf wiederum als Link direkt in der Asset-Dokumentation hinterlegt ist. Das Formulieren effektiver Prompts für KI-Anfragen, auch *Prompt Engineering* genannt, ist eine zunehmend wichtige und auch anspruchsvolle Aufgabe, und wenn man einmal einen guten Prompt im Zusammenhand mit einer wiederholt auftretenden Aufgabenstellung gefunden hat, sollte man diesen unbedingt in die dazugehörende Anleitung integrieren.

Durch die hohe Flexibilität eines solchen Skriptes, lassen sich die abgefragten Aspekte so anpassen, dass sie punktgenau auf die für ein Unternehmen bedeutsamen Assets und die dafür angelegten Strukturen ausgerichtet sind. Die Dokumentation im Wiki kann dann bei Änderungen unabhängig von dem Skript, mit dem sie erstellt wurde, angepasst werden.

Der entscheidende Vorteil bei einer interaktiven Anleitung für die Dokumentation von Dingen ist jedoch die Möglichkeit, Fragestellungen und daraus abgeleitete Maßnahmen direkt damit zu verbinden, und deren Durchführung gut zu unterstützen. So kann ein Skript, das ursprünglich nur für die Dokumentation gedacht war, und nur Informationen abgefragt hat, schrittweise bis zu einem Protokoll erweitert werden, dass am Ende garantiert, dass das Dokumentierte definierte Standards erfüllt.

10.4 Terminplanung

Unser letztes Beispiel im Abschnitt Organisationswerkzeuge ist ein kleines Terminplanungswerkzeug, das schön zeigt, wie sich mit ein bisschen Programmierung Aufgaben so lösen lassen, dass es für alle Beteiligten von Vorteil ist.

Die Ausgangssituation dürfte in einem Unternehmen recht häufig, aber mindestens einmal pro Jahr und Abteilung vorkommen: für eine Menge von Mitarbeiter sollen innerhalb eines Zeitraums Einzeltermine geplant werden, so dass am Ende für jeden Mitarbeiter der Menge ein Termin mit Raum eingeplant ist.

Die normale Vorgehensweise wäre nun, mit Hilfe des eigenen Terminkalenders und denen der Mitarbeiter und Räume einen Termin nach dem anderen zu planen, was bedeutet, dass man mindestens immer drei Kalender offen haben muss. Unschön wird das ganze dann gegen Ende der Planung, wenn man für die verbleibenden Mitarbeiter keine möglichen Termine mehr findet, und man somit Anfang und Mittelteil der Planung wieder umstellen muss.

Die deutlich einfachere Vorgehensweise besteht darin, zunächst nur Termine für sich selbst und den Raum einzuplanen, dafür eventuell auch einen mehr als benötigt wird. Anschließend veröffentlicht man die Liste dieser Termine im Wiki und bittet die Mitarbeiter, sich dort Ihre Wunschtermine auszuwählen, also je nach Gesamtzahl drei bis fünf Wunschtermine pro Mitarbeiter. Hierzu trägt sich der Mitarbeiter einfach in die Tabelle mit seinem Kürzel hinter dem jeweiligen Termin ein.

Bei der Auswahl kann zusätzlich noch eine Präferenz angegeben werden, um auszudrücken, ob der jeweilige Termin die erste, zweite oder dritte Wahl für den Mitarbeiter ist. Dies wird einfach durch die entsprechende Ziffer hinter dem Kürzel angegeben.

Für eine Auswahl von fünf Terminen könnten die Wünsche von vier Mitarbeitern mit den

Kürzeln AB, CD, EF und GH wie folgt aussehen:

Termin 1	AB2
Termin 2	CD, EF1
Termin 3	EF2, AB1
Termin 4	GH, CD
Termin 5	CD, EF3

In diesem Fall hat der Mitarbeiter AB nur zwei Termine angegeben und GH nur einen, eventuell weil er von auswärts anreisen muss. Der Mitarbeiter CD hat keine Präferenzen, die drei von ihm gewählten Termine sind für ihn gleichwertig.

Nach Ablauf der Wahlfrist werden nun einfach die gemachten Angaben in die Eingabefelder des folgenden Skriptes übertragen:

Beispiel 43 – Terminplaner

```
~SetStopCounter 100000; 3000000
~Input Anz: Anzahl Termine
~Execute
@Terminpräferenzen >>_ $Anz Termine
~Set idx=0
~DoWhile $idx<$Anz
    ~AddTo idx+=1
    ~Input Prf$idx: Termin $idx - Präferenzen (kommagetrennt)
    >>* Termin $idx: $Prf$idx
~Loop
~Set BestPSum=10000
~Set BestAnzV=0
~Set TidxS=1
~Set MAVS=
~Set AnzVS=0
~Set PSumS=0
~GoSub SucheLoesung; BaseKey=S
@Ergebnis >>_ $BestAnzV Termine verplant, $BestPSum Punkte
~Set idx=0
~DoWhile $idx<$Anz
    ~AddTo idx+=1
    >>* Termin $idx: $FErg($idx)
~Loop

// Tidx$BaseKey = Nächster Terminindex
// MAV$BaseKey = Liste der schon verplanten Mitarbeiter
// PSum$BaseKey = Aktuelle Präferenz-Summe
// Rückgabe: FErg(1..Anz) = Belegung der Termine
~DefineSub SucheLoesung
    ~Split VerplanteMA=$MAV$BaseKey|/
    ~Calculate AnzV$BaseKey=$VerplanteMA(0)-1
    ~If $Tidx$BaseKey<=$Anz
```

```
        ~Split pr$BaseKey=$Prf$Tidx$BaseKey|,
        ~ForEach pi in pr$BaseKey
            ~RegExMatch pw = $pi|([A-Z][A-Z][a-z]*)([0-9]?)
            ~If $pw(0)
                ~Set ku=$pw(1)
                ~Set va=$pw(2)
                ~If $va==
                    ~Set va=1
                ~If $MAV$BaseKey/!~/$ku/
                    ~Set Erg($Tidx$BaseKey)=$ku
                    ~Calculate Tidx$BaseKeyX$ku=$Tidx$BaseKey+1
                    ~Calculate PSum$BaseKeyX$ku=$PSum$BaseKey+$va
                    ~Set MAV$BaseKeyX$ku=$MAV$BaseKey/$ku
                    ~GoSub SucheLoesung; BaseKey=$BaseKeyX$ku
        ~Loop
        ~CalculateExpression restpot=$Anz-$BestAnzV
            __-($Tidx$BaseKey-$AnzV$BaseKey)
        ~If $restpot>=0
            ~Set Erg($Tidx$BaseKey)=Nicht besetzt
            ~Calculate Tidx$BaseKey0=$Tidx$BaseKey+1
            ~Set PSum$BaseKey0=$PSum$BaseKey
            ~Set MAV$BaseKey0=$MAV$BaseKey
            ~GoSub SucheLoesung; BaseKey=$BaseKey0
    ~Else
        ~If $AnzV$BaseKey>$BestAnzV || $AnzV$BaseKey==$BestAnzV
            __ && $PSum$BaseKey<$BestPSum
            ~Set BestAnzV=$AnzV$BaseKey
            ~Set BestPSum=$PSum$BaseKey
            ~Set idx=0
            ~DoWhile $idx<$Anz
                ~AddTo idx+=1
                ~Set FErg($idx)=$Erg($idx)
            ~Loop
~Return
```

Da das Skript zur Lösungsfindung exponentiell viele Variationen durchprobiert, werden gleich zu Beginn die Stop-Zähler für die maximale Anzahl an Schleifendurchläufen und Befehlsausführungen mit dem Befehls `~SetStopCounter` auf sehr große Werte gesetzt.

Zuerst wird die Anzahl der Termine abgefragt, dann für jeden Termin die Präferenzen, wie üblich kommagetrennt.

Die Suche nach der passenden Lösung erfolgt mit Hilfe der Funktion `SucheLoesung`, die als Übergabeparameter den Index des nächsten zu planenden Termins, die Liste der schon verplanten Mitarbeiter und die Zwischensumme der Präferenzwerte übergeben bekommt. Die Anzahl der schon verplanten Termine wird mit Hilfe des `~Split`-Befehls aus der Aufzählung der schon verplanten Mitarbeiter berechnet.

Immer wenn der Index des letzten Termins erreicht ist, wird die aktuelle Lösung mit der bisher besten Lösung verglichen, wobei primär die Anzahl der geplanten Termine

zählt, und bei gleicher Anzahl eine möglichst geringe Präferenzsumme. Ist eine neue beste Lösung gefunden, wird diese in das Feld `FErg(...)` übertragen.

Ansonsten werden alle Mitarbeiter durchlaufen die sich für den als nächstes zur Planung anstehenden Termin eingetragen haben. Die Kürzel werden mit Hilfe eines regulären Ausdrucks vom Präferenzwert getrennt. Wenn keiner angegeben ist, wird der Wert 1 gesetzt. Mit Hilfe des Enthält-nicht-Operators `!~` wird dann geprüft, ob der entsprechende Mitarbeiter noch nicht in der Menge der bisher verplanten Mitarbeiter enthalten ist, wobei das Trennzeichen `/` verwendet wird. In diesem Fall werden die Parameter für einen weiteren Funktionsaufruf gesetzt, wobei die `BaseKey`-Variable um den Buchstaben X und das Mitarbeiterkürzel erweitert werden.

Nach Durchlauf aller Mitarbeiter wird auch noch die Variante geprüft, bei der der anstehende Termin nicht besetzt wird. Gerade wenn mehr Termine angeboten werden, als Mitarbeiter vorhanden sind, müssen Termine unbesetzt bleiben.

Da dadurch die Menge der zu untersuchende Lösungen noch mal deutlich erhöht wird, wird hier noch zusätzlich das Restpotenzial R abgeprüft. Dieses berechnet sich über die Formel $R = A - A_B - (i - A_T)$, wobei A die Anzahl der angebotenen Termine ist, A_B, die Anzahl der verplanten Termine der bislang besten Lösung, i der aktuelle Index und A_T die Anzahl der bis jetzt verplanten Termine. Ist $R \geq 0$, kann trotz nicht besetzen des anstehenden Termins potentiell noch eine bessere Lösung gefunden werden, ansonsten ist das ausgeschlossen und diese Variante muss nicht weiterverfolgt werden.

Am Ende wird die beste gefundene Lösung als Ergebnis ausgegeben. Das Skript kann auch mit den Daten aus unserem Beispiel oben umgehen und findet zielsicher die beste Lösung mit vier verplanten Terminen und einer Präferenzsumme von 4, d.h. jeder Mitarbeiter bekommt einen Termin seiner ersten Wahl:

Termin 1	nicht besetzt
Termin 2	EF
Termin 3	AB
Termin 4	GH
Termin 5	CD

Der Vorteil dieser Methode liegt darin, dass die lästige Suche nach freien Terminen auf die einzelnen Mitarbeiter aufgeteilt wird, die aber dafür nur jeweils ihren eigenen Kalender sichten müssen. Als Ausgleich dafür bekommen sie die Möglichkeit, mit der Präferenz Einfluss auf die Terminauswahl zu nehmen.

Dadurch, dass auch die Reihenfolge, in der die Wünsche eingetragen werden so gut wie keine Rolle spielt, ist dieser Ansatz auch besser als die Startschuss-Variante, bei dem sich jeder nur für genau einen Termin eintragen darf, der noch nicht gewählt wurde, so dass die Mitarbeiter benachteiligt werden, die nicht unmittelbar auf die Aufforderung reagieren können.

11 Mitgestaltung

Wie schon mehrfach beschrieben, sollte man Skripte nicht einfach nur als Programmcode sehen, der durch die Umgebung ausgeführt werden kann und mit dem sich bestimmte Aufgaben einfach lösen lassen, sondern man sollte Skripte vor allem auch als ein Medium betrachten, das es ermöglicht, fachliches Wissen so zu hinterlegen, dass es bei der

Anwendung des Skriptes automatisch berücksichtigt wird. Es ist nicht notwendig, dass der Anwender weiß, wie man die Aufgabe richtig umsetzt, so lange ihm das Skript keine Möglichkeit gibt, sie falsch umzusetzen.

Ausgehend davon sollte die Zielsetzung sein, möglichst viel in den Köpfen vorhandenes Wissen in so eine Form zu bringen. In diesem Abschnitt zeigen wir anhand von Beispielen, wie das möglich ist, auch ohne dass man jedem Mitarbeiter die *FlowProtocol-2*-Programmierung ans Herz legen muss.

11.1 Angeleitete Skript-Erweiterungen

Bei Auswahlabfragen hat man die Auswahl zwischen verschiedenen Werten. Was aber kann der Anwender machen, wenn der von ihm gewünschte Wert nicht in der Liste aufgeführt ist? Für diesem Fall könnte man einen entsprechenden zusätzlichen Auswahlwert *Nicht in Liste* ergänzen, und versuchen, diesen neuen Fall so gut wie möglich im Skript zu behandeln.

Meist werden in Abhängigkeit von Auswahlwerten nur bestimmte Variablen oder Schalter gesetzt, die dann in einem anderen, schon vorhandenen Teil des Skriptes, weiterverarbeitet werden. In diesem Fall kann man die entsprechenden Werte für die zu setzenden Variablen direkt abfragen und hat so den allgemeinen Fall behandelt. Schöner wäre es jedoch, wenn die verschiedenen Eingaben, die eventuell mit einigem Aufwand zusammengetragen werden müssen, für den nächsten Anwender direkt im Skript als auswählbarer Wert eingebaut wären.

FlowProtocol 2 kann selbst keine Änderung an den Skripten durchführen, aber es kann sehr gut dazu anleiten.

Betrachten wir die Fragestellung, unter welcher Lizenz ein Softwareprodukt bereitgestellt wird, um daraus bestimmte Eigenschaften abzuleiten, wie die Bezeichnung, eine Link auf eine Beschreibung der Lizenz, die Information ob diese Lizenz den Einsatz im kommerziellen Umfeld erlaubt und ob sie kostenpflichtig ist.

Hier bietet es sich an, die Auswahlabfrage zunächst mit einem Wert zu beginnen, und weitere Lizenzen dann hinzuzunehmen, wenn sie tatsächlich benötigt werden. Diese Abfrage könnte man wie folgt aufbauen:

Beispiel 44 – Lizenzabfrage

```
?Lizenz: Unter welcher Lizenz wird die Software bereitgestellt?
    #: MIT-Lizenz
        ~Set BezeichnungLizenz=MIT-Lizenz
        ~Set LinkAufLizenzbeschreibung=
            __https://de.wikipedia.org/wiki/MIT-Lizenz
        ~Set ErlaubtKomVerw=Ja
        ~Set IstKostenpflichtig=Nein
    // Oberhalb von hier neue Einträge einfügen
    #x: Nicht in Liste
        @Skript-Erweiterung >>_ Es wird empfohlen, das Skript
            __ um die nicht vorhandenen Einträge zu erweitern.
            __ Gehe dazu wie folgt vor:
```

```
        >> Öffne die Skriptdatei in einem Texteditor:
            >|$ScriptFilePath
        >> Füge vor Zeile $LineNumber-7 folgenden Codeblock ein:
            ~Input ILizenz: Unter welcher Lizenz wird die
                __ Software bereitgestellt?
            >|$Chr(9)#: $ILizenz
            ~Input IBez: Bezeichnung Lizenz
            >|$Chr(9)$Chr(9)~Set BezeichnungLizenz=$IBez
            ~Set BezeichnungLizenz=$IBez
            ~Input ILink: Link auf Lizenzbeschreibung
            >|$Chr(9)$Chr(9)~Set LinkAufLizenzbeschreibung=$ILink
            ~Set LinkAufLizenzbeschreibung=$ILink
            ?QKom: Erlaubt kommerzielle Verwendung?
                #j: Ja
                    >|$Chr(9)$Chr(9)~Set ErlaubtKomVerw=Ja
                    ~Set ErlaubtKomVerw=Ja
                #n: Nein
                    >|$Chr(9)$Chr(9)~Set ErlaubtKomVerw=Nein
                    ~Set ErlaubtKomVerw=Nein
            ?QKost: Ist kostenpflichtig?
                #j: Ja
                    >|$Chr(9)$Chr(9)~Set IstKostenpflichtig=Ja
                    ~Set IstKostenpflichtig=Ja
                #n: Nein
                    >|$Chr(9)$Chr(9)~Set IstKostenpflichtig=Nein
                    ~Set IstKostenpflichtig=Nein
        >> Speichere die Datei ab.
@Ausgabe >>* Die Software wird unter der Lizenz "$BezeichnungLizenz"
        __ bereitgestellt.
    >* Link:
    ~AddLink $LinkAufLizenzbeschreibung | $LinkAufLizenzbeschreibung
    >* Kommerzielle Verwendung möglich? - $ErlaubtKomVerw
    >* Kostenpflichtig? - $IstKostenpflichtig
```

Die Auswahl von *Nicht in Liste* fragt die benötigen Werte ab und setzt die Variablen in der gleichen Weise, wie es die anderen Einträge machen, so dass das Skript seine Aufgabe erfüllen kann, z.B. die Dokumentation der abgefragten Lizenz mit den ganzen Eigenschaften.

Es passiert aber noch mehr. Als Teil der Ausgabe wird in einem eigenen Abschnitt *Skript-Erweiterung* eine Schritt-für-Schritt-Anleitung gegeben, die beschreibt, wie man die neu erfasste Lizenz permanent in das Skript einbauen kann. Hierbei wird beschrieben, wo man die Skript-Datei findet, und welche Zeilen man wo einfügen muss. Der einzufügende Codeblock wird vollständig ausgegeben und kann direkt in die Zwischenablage übernommen werden. Die Erweiterung kann damit ohne jede Kenntnis der *FlowProtocol-2*-Programmiersprache umgesetzt werden.

Für die Zusammenstellung der benötigten Informationen wird zum einen die Variable `$ScriptFilePath` verwendet, die den Dateipfad des aktuellen Skriptes ausgibt. Dieser ist aus Sicht des Servers, auf dem *FlowProtocol 2* läuft, und muss gegebenenfalls gegen

einen ausgeschriebenen Pfad ausgetauscht werden, wenn der Zugriff auf die Datei für den Anwender über einen anderen Pfad erfolgen soll. Die Zeilennummer wird mit Hilfe der Variablen `$LineNumber` abgefragt, die stets die Nummer der entsprechenden Zeile zurückgibt. Möchte man sich auf eine Zeile n Zeilen darüber beziehen, kann für diese Variable direkt ein Korrekturwert von $-n$ angegeben werden.

Das Tabulatorzeichen für die Einrückung des auszugebenden Skriptcodes wird über die Sequenz `$Chr(9)` angegeben, da unmaskierte Tabulatorzeichen beim Einlesen von Skripten durch vier Leerzeichen ersetzt werden.

11.2 Metaskripting

Jetzt kann man zu Recht bemängeln, dass der Scriptcode, der für die *Nicht in Liste*-Auswahlmöglichkeit erstellt werden muss, selbst sehr komplex und relativ umfangreich ist. Die Hürde eine solche Möglichkeit zu schaffen scheint damit recht hoch zu sein, und selbst wenn man sich dafür bei einer Vorlage bedienen kann, sind immer noch einige Anpassungen notwendig, da sich die abzufragenden Variablen ja von Fall zu Fall unterscheiden.

Die Lösung für dieses Problem ist, man ahnt es vermutlich schon, ein *FlowProtocol-2*-Skript, das einem diese Arbeit abnimmt.

Das sogenannte Metaskripting, also das Erstellen von Skriptcode der neuen Skriptcode erzeugt, ist ein mächtiges Instrument und lässt sich an vielen Stellen einsetzen, sogar über verschiedene Programmiersprachen hinweg. Es gibt Beispiele für produktiv eingesetzte *FlowProtocol-2*-Skripte, die SQL-Anweisungen erzeugen, der wiederum C#-Code ausgegeben.

Für unseren Anwendungsfall der angeleiteten Skript-Erweiterung benötigen wir folgende Eingaben: die Fragestellung, einen Schlüssel, eine Liste von Werten, die entweder als Texteingabe oder Schalter abgefragt werden sollen, und die Information in welcher Tiefe der Code eingerückt sein soll.

Die Abfrage erfolgt entsprechend mit vier Texteingaben, wobei die Menge der abzufragenden Werte über eine kommagetrennte Aufzählung gemacht wird und die Unterscheidung zwischen Textangaben und Schaltern durch die Angabe eines Fragezeichens bei jedem Wert getroffen wird. Die Einrückung wird durch eine Sequenz von T-Zeichen angegeben, wobei später jedes T durch Tabulatorzeichen ersetzt wird.

Das Ergebnis sieht wie folgt aus:

Beispiel 45 – Meta-Erweiterung

```
~Input Frage: Welche Frage soll ausgegeben werden?
~Input Key: Schlüssel für die Frage
~Input WListe: Abzufragende Werte (kommagetrennt, @ wenn leer)
~AddHelpLine Werte mit "?" werden als Schalter abgebildet.
~AddHelpLine Der Wert mit "=" wird auf den Auswahlwert gesetzt.
~Input Tabs: Einrückung (T pro Tab, @ sonst)
~CamelCase CKey=$Key
~Set T=$Chr(9)
~Set ST=$Chr(36)Chr(36)Chr(9)
```

```
~Replace EE=$Tabs|@->
~Replace FF=$EE|T->$Chr(36)Chr(36)Chr(9)
~Replace EE=$EE|T->$Chr(9)
@Erweiterung >>* Auswahlabfrage
>|$EE?$CKey: $Frage
>|$EE$T// Oberhalb von hier neue Einträge einfügen
>|$EE$T#x: Nicht in Liste
>|$EE$T$T@Skript-Erweiterung >>_ Es wird empfohlen, das
    __ Skript um die nicht vorhandenen Einträge zu erweitern.
    __ Gehe dazu wie folgt vor:
>|$EE$T$T>> Öffne die Skriptdatei in einem Texteditor:
>|$EE$T$T$T>|$Chr(36)ScriptFilePath
>|$EE$T$T>> Füge vor Zeile $Chr(36)LineNumber-5 folgenden
    __ Codeblock ein:
>|$EE$T$T$T~Input I$CKey: $Frage
>|$EE$T$T$T>|$FF$ST#: $I$CKey
~If $WListe!=@
    ~Split Werte=$WListe|,
    ~ForEach widx in Werte
        ~CamelCase vidx=$widx
        ~If $widx~?
            @Erweiterung >|$EE$T$T$T?Q$vidx: $widx
            >|$EE$T$T$T$T#j: Ja
            >|$EE$T$T$T$T$T>|$FF$ST$ST~Set $vidx=Ja
            >|$EE$T$T$T$T$T~Set $vidx=Ja
            >|$EE$T$T$T$T#n: Nein
            >|$EE$T$T$T$T$T>|$FF$ST$ST~Set $vidx=Nein
            >|$EE$T$T$T$T$T~Set $vidx=Nein
            ~If $ErsterSchalter!=Ja
                @IfBlocks >>* If-Abfragen für die Schalter
                ~Set ErsterSchalter=Ja
            @IfBlocks >|$EE~If $$vidx==Ja
            >|$EE$T//$widx
        ~ElseIf $widx~=
            @Erweiterung >|$EE$T$T$T>|$FF$ST$ST~Set $vidx=$I$CKey
            >|$EE$T$T$T~Set $vidx=$I$CKey
        ~Else
            @Erweiterung >|$EE$T$T$T~Input I$vidx: $widx
            >|$EE$T$T$T>|$FF$ST$ST~Set $vidx=$I$vidx
            >|$EE$T$T$T~Set $vidx=$I$vidx
    ~Loop
@Erweiterung >|$EE$T$T>> Speichere die Datei ab.
~MoveSection IfBlocks->Erweiterung
```

Um den Code schmal zu halten, werden am Anfang die Hilfsvariablen `T` und `ST` mit den häufig benötigten Sequenzen belegt. Damit die Tabulatorzeichen (`$Chr(9)`) im innersten Code bis zum letztendlichen Einfügen in das Zielskript noch maskiert bleiben, wird das `$`-Zeichen der `$Chr(...)`-Sequenzen teilweise wiederholt durch `$Chr(36)` ersetzt. In gleicher Weise werden auch die `$`-Zeichen in den Aufrufen der Systemvariablen

`$ScriptFilePath` und `$LineNumber` durch `$Chr(36)` ersetzt, so dass diese erst bei der Ausführung im Zielskript aufgelöst werden und nicht bereits in der Ausgabe dieses Skriptes.

Für die Schalter, also die Werte, die mit einem Fragezeichen gekennzeichnet wurden, werden in einem zweiten Codeblock noch If-Abfragen erstellt. Damit kann man, ähnlich wie in den Beispielen 39 bis 42, für jeden Schalter individuellen Code ausführen, wenn dieser gesetzt ist.

Metaskripting ist ohne Frage anspruchsvoll, aber dafür ist der Nutzen aufgrund der Multiplikatorwirkung recht groß, und wenn ein solches Skript am Ende dazu führt, dass alle Mitarbeiter in der Lage sind, die produktiv eingesetzten Skripte permanent zu erweitern und ihr Wissen dort einzupflegen, dann ist das den Aufwand auf jeden Fall wert.

Auch hier verhält es sich ähnlich wie bei der Entwicklung von Anwendungscode. Wird etwas nur einmal benötigt, kann es der Spezialist machen. Wird etwas mehrfach benötigt, kann der Spezialist passendes Werkzeug dafür anfertigen, mit dem die Aufgabe einfacher umgesetzt werden kann.

11.3 Wissensdokumentation

Die Dokumentation von Wissen oder auch die Verwaltung von Wissen ist eine der wichtigsten Aufgaben in einem Unternehmen, das seine Wertschöpfung in erster Linie aus dem Wissen der Mitarbeiter generiert.

Fast alle Abläufe, die im Arbeitsalltag stattfinden, setzen Wissen voraus, ebenso die meisten Anwendungen und Werkzeuge, die dabei eingesetzt werden.

Dort, wo dieses Wissen jeden Tag, oder zumindest einmal in der Woche benötigt wird, wird es sich von selbst verfestigen, so dass man hier keinen Aufwand betreiben muss. Anders sieht es aus bei Wissen, dass nur gelegentlich benötigt wird, und bei dem zusätzlich gar nicht offensichtlich ist, ob es hier etwas zu wissen gibt oder nicht. Wenn dieses Wissen fehlt, laufen Dinge schief, was im besten Fall nur Mehraufwand bedeutet und im schlimmsten Fall zu problematischen Fehler führen kann.

Wer nicht weiß, wie eine bestimmte Art von Skript erstellt wird, wird keines erstellen können, wer nicht weiß, welches Risiko eine bestimmte Entwicklung mit sich bringt, wird nicht darauf achten können und die Eintrittswahrscheinlichkeit erhöhen, und wer nicht weiß, welche Richtlinien im Umgang mit Kundendaten zu beachten sind, wird möglicherweise einen Datenschutzverstoß verursachen.

Wie schon mehrfach beschrieben, lassen sich mit Hilfe von Skripten gute Anleitungen und Hilfsmittel erstellen, die das wesentliche Detailwissen über bestimmte Arbeitsschritte enthalten und so den einzelnen Mitarbeiter auch wissensmäßig entlasten.

Auch das Grundwissen zu allgemeinen Themengebieten, wie C#-Programmierung oder Kenntnis der SQL-Syntax muss nicht unbedingt firmenspezifisch verwaltet werden, da es ja eine generelle Vorstellung davon gibt, was es bedeutet, dieses Wissen zu haben, und man es ggf. mit zahlreichen Schulungen oder Tutorials abgleichen und auffrischen kann.

Generell sollte das vorausgesetzte Wissen auf die Punkte beschränkt werden, die unternehmensspezifisch sind, die einem helfen, die richtigen Entscheidungen zu treffen, Fehler zu vermeiden und das für bestimmte Arbeitsschritte benötigte Detailwissen, bzw. die Anleitungen dafür zielsicher aufzufinden. Das ist immer noch eine ganze Menge, und für

jede spezielle Aufgabe und für jeden Verantwortungsbereich gibt es eigenes Wissen, das benötigt wird.

Wie aber verwaltet man dieses Wissen und wie erfährt ein neuer Mitarbeiter, was er alles wissen muss? Wie lässt sich überprüfen ob ein Mitarbeiter alles weiß, was er wissen sollte? Die Verantwortung über diese Punkte liegt in der Regel beim direkten Vorgesetzten, der aber auch nicht in den einzelnen Menschen hineinschauen kann. Meist behilft man sich damit die zwingend einzuhaltenden Richtlinien schriftlich festzuhalten und für das restliche Wissen grobe Themengebiete aufzulisten, die zur Orientierung verwendet werden können.

Ein im Gegensatz dazu sehr expliziter Ansatz für die Wissensverwaltung ist die Auflistung von Wissen in Form von Reflektionsaussagen. Eine Reflektionsaussage ist dabei eine Aussage, die in der Form *Ich weiß, ...* formuliert wird, z.B. *Ich wie man in einem PowerShell-Skript auf Befehlszeilenargumente zugreift*.

Die Aussage dient, wie die Bezeichnung schon sagt, dazu, das eigene Wissen zu reflektieren, also darüber nachzudenken, ob man diese Aussage mit Ja oder mit Nein beantworten kann. Der große Vorteil bei dieser Form der Erfassung liegt darin, dass es sehr einfach ist, Wissen in dieser Form zu erfassen, da man ja nicht den Inhalt des Wissens erfassen muss, sondern nur einen Verweis darauf. Das ist sogar dann möglich wenn man die Aussage für sich selbst gar nicht mit Ja beantworten kann. Darüberhinaus erlaubt es die Formulierung, das abgefragte Wissen sehr genau abzugrenzen, so dass man in der Regel klar verstehen kann, was gemeint ist.

Genau diese Einfachheit ist wichtig, um mit möglichst wenig Aufwand möglichst alles an Wissen zu dokumentieren, das man für wichtig und relevant hält. Vor allem in dem Moment, wo man feststellt, dass aufgrund fehlenden Wissens etwas schief gelaufen ist, sollte man umgehend dafür sorgen, dass dieses fehlende Wissen in die Dokumentation aufgenommen wird.

Diese Praktik lässt sich recht gut mit *FlowProtocol 2* anwenden. Die Listen mit den Reflektionsaussagen zu einzelnen Wissensgebieten lassen sich einfach als Textdateien anlegen und in einem Ordner speichern. In unserem Beispiel gibt es einen Ordner *.Wissensgebiete*, der wiederum weitere Ordner für bestimmte Wissensgebiete enthält, hier *Wissen Flowprotocol2* und *Wissen Softwareentwicklung*. Durch die Markierung des Ordners *.Wissensgebiete* mit einem Punkt am Anfang, wird dieser nicht als Skript-Unterordner im Menübaum von *FlowProtocol 2* aufgelistet.

Im Ordner *Wissen für FlowProtocol2* sind die beiden Textdateien *Basisbefehle.txt* und *Programmierung.txt* hinterlegt, die wie die Skripte auch, mit einem einfachen Texteditor erstellt wurden. Diese haben folgenden Inhalt:

Beispiel 46 – Basisbefehle.txt

```
[Ausgabe]
Ich weiß, wie man eine Ausgabe erzeugt.
Ich weiß, wie man eine zweistufige Ausgabe erzeugt.
Ich weiß, wie man die Ausgabe in Abschnitte gliedert.
Ich weiß, wie man die Nummerierung der Aufzählung steuert.
Ich weiß, wie man einen Codeblock erzeugt.
Ich weiß, wie man einen Link auf eine Internetseite erzeugt.
```

```
[Eingabe]
Ich weiß, wie man eine Auswahlabfrage erstellt.
Ich weiß, wie man eine Texteingabe abfragt.
Ich weiß, wie man einen eingegebenen Text wieder ausgibt.
Ich weiß, wie man Auswahlabfragen ineinander verschachtelt.
```

Beispiel 47 – Programmierung.txt

```
[Strukturen]
Ich weiß, wie eine If-Abfrage aufgebaut ist.
Ich weiß, wie eine Bedingung aufgebaut ist.
Ich weiß, wie man eine Von-1-bis-10-Schleife implementiert.
Ich weiß, wie man eine Liste von Elementen durchläuft.
Ich weiß, wie man eine Funktion definiert und aufruft.
Ich weiß, wie man eine Sprungmarke definiert und anspringt.

[Verarbeitung]
Ich weiß, wie man Variablen setzt und verwendet.
Ich weiß, wie man Texte ersetzen kann.
Ich weiß, wie man reguläre Ausdrücke verwenden kann.
Ich weiß, wie man Berechnungen durchführen kann.
```

Beide Dateien haben das Dateilisten-Format, d.h. Leerzeilen und mit \\ beginnende Kommentarzeilen werden ignoriert und Zeilen, die mit eckigen Klammern beginnen und enden werden als Abschnittsüberschriften für die darauffolgenden Zeilen interpretiert.

Das Skript, mit dem man auf diese Daten zugreift sieht wie folgt aus:

Beispiel 48 – Wissensreflektion

```
~ListDirectories wdir = .Wissensgebiete; Pattern=Wissen *
~DynamicOptionGroup WGeb:wdir;Wähle das Wissensgebiet
~Execute
~ListFiles wdats = .Wissensgebiete|$wdir($WGeb); Pattern=*.txt
~Set idx=0
~ForEach wdat in wdats
    ~AddTo idx+=1
    ~Replace wpak($idx)=$wdat|.txt->
~Loop
~If $idx==0
    @Hinweis >>_ Im Ordner "Wissensgebiete\$wdir($WGeb)"
    __ sind keine Dateien vorhanden. Die Ausführung
    __ wird abgebrochen.
    ~End
~DynamicOptionGroup WPak:wpak;Wähle das Wissenspaket
~Execute
~Set WPfad=.Wissensgebiete|$wdir($WGeb)|$wdats($WPak)
?A: Was möchtest du tun?
    #sa: Alle Reflektionsaussagen durchgehen
```

```
    ~ForEachLine ref in $WPfad; IndexVar=idx; SectionVar=Ab
        ~SetInputSection $Ab
        ?R$idx: $ref
        #j: Ja
            @Was ich weiß >> $ref
        #n: Nein
            @NN >> $ref
        ~Calculate mod4 = $idx % 4
        ~If $mod4==0
            ~Execute
    ~Loop
    ~MoveSection  NN->Was ich nicht weiß
#s4: Drei Reflektionsaussagen als Stichprobe
    ~ForEachLine ref in $WPfad; Take=3; IndexVar=idx
        ?R$idx: $ref
        #j: Ja
        #n: Nein
            ~Replace refmod=$ref|Ich weiß, ->...
            @Heute sollte ich rausfinden... >> $refmod
    ~Loop
#db: Weitere Reflektionsaussagen erfassen
    @Anleitung >> Öffne die Wissensdatei
        >|$CurrentScriptPath\Wissensgebiete\$wdir($WGeb)
        __\$wdats($WPak)
    >> Füge neue Reflektionsaussagen ein
        > Diese sollten mit "Ich weiß, " beginnen
        > ... und nach Möglichkeit nicht die Antwort enthalten
```

Es beginnt mit einer Abfrage aller Verzeichnisse im Ordner *.Wissensgebiete* die mit *Wissen* beginnen. Dazu wird der Befehl `~ListDirectories` verwendet, der das Ergebnis in einem Feld abspeichert.

Durch den Befehl `~DynamicOptionGroup` wird diese Auflistung als Auswahl einer Auswahlabfrage angeboten, sodass der Anwender ein Wissensgebiet und damit gleichzeitig einen Ordner auswählen kann.

Im nächsten Schritt werden mit dem Befehl `~ListFiles` die Dateien innerhalb des gewählten Ordners abgefragt, die auf *.txt* enden. Ähnlich wie oben wird hier eine Auswahlabfrage generiert, zuvor wird jedoch in einer Schleife eine Auflistung ohne die txt-Endungen erzeugt. Wenn in dem gewählten Ordner keine Dateien vorhanden sind, wird die Ausführung mit einer Hinweismeldung abgebrochen.

Nach Auswahl einer Wissenspaket-Datei hat der Anwender nun drei Möglichkeiten, diese zu verwenden. Er kann *Alle Reflektionsaustragen durchgehen*, wobei diese mit den Antwortmöglichkeiten Ja und Nein abgefragt werden. Am Ende werden dann die beiden Listen mit den Überschriften *Was ich weiß* und *Was ich nicht weiß* ausgegeben. Diese Auflistung wird mit dem schon in Abschnitt 8.3 verwendeten Befehl `~ForEachLine` erzeugt, der jeder Zeile der ausgewählten Datei durchläuft. Die Zeile selbst wird dabei in der `ref`-Variablen zurückgegeben. Zusätzlich können Variablen für den Zeilenzähler über das `IndexVar`-, und für die Abschnittsüberschrift über das `SectionVar`-Argument angegeben werden. Bei jedem Schleifendurchlauf wird überprüft ob der Zähler durch vier

teilbar ist und wenn ja, der `~Execute`-Befehl ausgeführt, sodass die Reflektionsaussagen in Viererpaketen abgefragt werden.

Bei *Drei Reflektionsaussagen als Stichprobe* werden zufällig drei Zeilen aus der Datei ausgewählt. Die Zahl drei wird mit dem `Take`-Argument angegeben. Der Seed-Wert des Zufallsgenerators wird dabei als Parameter `_rseed` in der URL notiert, so dass beim erneuten Aufrufen oder einer Aktualisierung der Seite wieder die genau gleiche Zufallsfolge entsteht. Hier werden nur die mit Nein beantworteten Punkte unter der Überschrift *Heute sollte ich rausfinden...* in leicht angepasster Form aufgelistet.

Bei *Weitere Reflektionsaussagen erfassen* wird eine Anleitung ausgegeben, die dabei unterstützt, die richtige Datei zu finden und dort weitere Aussagen einzufügen. Hierbei wird für die Zusammensetzung des Pfades die Systemvariable `$CurrentScriptPath` verwendet, die den Pfad des aktuell ausgeführten Skriptes ausgibt.

Bei der Anwendung dieser Mittel gibt es verschiedene Möglichkeiten. Die Zielsetzung sollte in jedem Fall darin bestehen, dass am Ende alle Mitarbeiter die Dinge wissen, die Sie wissen müssen, um ihre Arbeit gut zu machen, und dabei sollte auch immer davon ausgegangen werden, dass die Mitarbeiter die Bereitschaft dazu mitbringen.

Zu sehen, wie viel Fachwissen für die eigene Arbeit notwendig ist, unterstreicht wie anspruchsvoll diese ist, und sollte daher ein gutes Gefühl vermitteln. Besonders dann, wenn ein Großteil dieses Wissens über Jahre hinweg aus dem Team heraus aufgebaut wurde, ist die Sichtweise darauf eine andere, als wenn die Inhalte ausschließlich aus externen Quellen stammen. Der Entwickler einer Framework-Komponente hat letztendlich das gleiche Interesse daran, dass sein Kollege diese richtig verwendet, wie der Datenschutzbeauftragte daran hat, dass die Datenschutz-Richtlinien allen bekannt sind und eingehalten werden.

Das Durchgehen der Reflektionsaussagen ermöglicht es jedem einzelnen, die weißen Flecken auf seiner Wissenslandkarte zu identifizieren und eventuell vorhandene Lücken zu schließen. Dafür muss er diese Lücken auch nicht gegenüber einem anderen offenlegen, denn *FlowProtocol 2* arbeitet diskret und gibt keine Informationen weiter. Zu wissen, was man nicht weiß, ist die Grundvoraussetzung dafür, um fehlendes Wissen zielgerichtet zu ergänzen. Man kann damit die benötigte Information recherchieren oder direkt bei einem Kollegen nachfragen. Durch die Reflektionsaussage hat man sogar eine kompakte Beschreibung, mit der man bei seiner Anfrage auf das Wissen Bezug nehmen kann..

Generell kann man davon ausgehen, dass es etliche Punkte gibt, die noch von mehreren Mitarbeitern nicht gewusst werden, oder bei denen es aus anderen Gründen sinnvoll ist, sie nochmal in gemeinsamer Runde zu besprechen, z.B. um auf den gleichen Stand zu kommen. Gleichzeitig unterliegt auch das Wissen selbst einem permanenten Wandel, es kommen neue Punkte dazu und bestehendes Wissen wird eventuell nicht mehr benötigt. Von daher ist auch eine regelmäßige redaktionelle Überarbeitung notwendig und es macht Sinn beides zusammenzulegen.

Unter dem Titel Wissensaktualisierung plant man mindestens einmal im Jahr einen Termin ein, bei dem alle Kollegen die Möglichkeit haben, sowohl ihr eigenes, als auch das gemeinsame Wissen aufzufrischen und zu aktualisieren. Idealerweise geht jeder dafür im Vorfeld die vorhandenen Listen durch und notiert die Punkte, die er im Meeting angesprochen haben möchte, sei es um sich selbst oder seine Kollegen auf den aktuellen Stand zu bringen, oder über die Herausnahme oder Überarbeitung des Reflektionsaussage zu diskutieren.

Für die Sammlung dieser Punkte kann eine eigene Variante im Wissensreflektion-Skript

eingebaut werden. Die von den verschiedenen Mitarbeitern gesammelten Punkte werden dann vom Moderator des Meetings zusammengetragen und je nach Häufigkeit zu einer Agenda umgesetzt. Je nach Menge der zu besprechenden Reflektionsaussagen ist es sinnvoll, diese in Vorbereitung des Meetings noch mit Antworten oder Verweisen auf eventuelle Informationsquellen anzureichern, sodass man im Meeting selbst keine Zeit mehr dafür aufwenden muss. Die Diskussion über die zu wissenden Dinge ist ebenso wertvoll wie der Wissensaustausch selbst, und so wird jeder am Ende des Tages viel gewonnen haben.

Die im Beispiel gezeigte Variante mit der Stichprobe aus drei Reflektionsaussagen lässt sich gut dafür einsetzen, sich jeden Morgen oder zweimal die Woche eine solche Auswahl zu generieren, und so permanent sein eigenes Wissen auf die Probe zu stellen. Die dabei gefundenen Lücken lassen sich dann ebenso beiläufig im Arbeitsalltag schließen.

Diese Form der Stichproben-Abfrage kann mit einer etwas größeren Anzahl auch dafür verwendet werden, eine Art von Prüfungsbogen zu erstellen, der dann für eine Kontrolle des Wissens durch eine dritte Person verwendet wird. Hier muss man sich jedoch genau fragen, ob und unter welcher Voraussetzungen man einen derartigen Kontrollmechanismus tatsächlich einsetzen möchte.

Umgekehrt gibt es auch gesetzliche Nachweispflichten, die man in Bezug auf seine Mitarbeiter wahrnehmen muss, etwa dass diese einen Führerschein haben, wenn sie mit Firmenfahrzeugen unterwegs sind. Das Zusenden der mit Ja beantworteten Reflektionsaussagen ist eine einfache Möglichkeit, das Vorhandensein von Wissen gegenüber seinen Vorgesetzten konkret zu bestätigen. Diese Vorgehensweise ist dahingehend stressfrei, da sich der Mitarbeiter ja die für sich notwendige Zeit nehmen kann, um eine eventuell vorgegebene Mindestquote zu erreichen. Ergänzend dazu gibt es ja auch noch den gemeinsamen Austausch und den Wissensaktualisierungstermin.

Sowohl die Berechnung der Quote, als auch das optionale Hinterlegen eventueller Antworten und Informations-Verweise direkt bei den Reflektionsaussagen lässt sich sehr einfach im Skript umsetzen. Ersteres ist eine einfache Berechnung und für den zweiten Punkt kann man für das Hinterlegen von Antwort-Informationen eine bestimmte Syntax vorgeben, die man mit Hilfe eines regulären Ausdrucks verarbeiten kann.

11.4 Personalisierung

Bei vielen Arten von interaktiven Anleitungen wird man zwangsläufig auf Elemente verweisen wollen, die lokal von der jeweiligen Umgebung des Mitarbeiters abhängig sind. Dazu gehören in erster Linie Datei- und Ordnerpfade, mit denen sich etwa parametrisierte Kommandozeilenaufrufe vorbereiten lassen, so dass der Anwender diese über eine Konsole oder die Tastenkombination *Win-R* ausführen kann. Dazu kommen Benutzerkürzel oder andere Kennzeichen, die sich z.B. in erzeugten Programmcode einarbeiten lassen.

Die Pfadstruktur könnte man vereinheitlichen, was allerdings mühsam und einschränkend ist, und die ansonsten benötigten Werte kann man abfragen, allerdings ist es gleichermaßen für Ersteller und Anwender mit viel Aufwand verbunden, wenn dies in jedem Skript geschieht.

Die Lösung besteht darin, eine einzelne Funktionsdatei zu erstellen, die bei Auswahl eines Benutzers eine definierte Menge von Variablen benutzerbezogen setzt, so dass diese im ganzen Skript verfügbar sind. Diese könnte so aussehen:

Beispiel 49 – Benutzervariablen.fps

```
?BK: Welche Benutzer-/Umgebungsvariablen sollen genutzt werden?
    #AB: Anton Barium
        ~Set BenVarKuerzel=AB
        ~Set BenVarPfadAufgabenmappen=H:\Aufgabenmappen
        ~Set BenVarPfadTools=C:\Work\Tools
    #CD: Christian Deuterium
    #EF: Ernst Flour
```

Im Beispiel sind exemplarisch die Variablen für den Benutzer AB ausformuliert, und in einer im Echtbetrieb eingesetzten Variante würde es vermutlich noch viel mehr benutzerbezogene Variablen geben.

Mit den Hilfsmitteln aus Abschnitt 11.2 könnte man auch dieses Skript ohne Mühe selbsterweiterbar machen, so dass ein Mitarbeiter, der erstmalig mit dieser Abfrage konfrontiert wird, bei den Eingaben angeleitet und unterstützt wird.

Zu beachten ist auch, dass alle Variablen mit einem einheitlichen Präfix benannt wurden, sodass versehentliche Namenskonflikte mit den im Skript selbst gesetzten Variablen ausgeschlossen werden können.

Ein Skript, das auf solchen Variablen aufbaut, könnte wie folgt aussehen:

Beispiel 50 – Patchskript

```
~Include Benutzervariablen.fps
~Input AKennung: Gib die Aufgabenkennung ein:
~Input PatchCode: Gib den Korrekturcode ein:
~SetDateTime Datum=yyyy-MM-dd
~Set DatBasis=$BenVarPfadAufgabenmappen\$Datum_$AKennung
~XmlEncode xPatchCode=$PatchCode
@Anleitung >> Erstelle eine neue Datei im Editor.
>> Gib den folgenden Code ein:
    >|<?xml version="1.0"?>
    >|<PATCHSCRIPT>
    >|  <NAME>$AKennung</NAME>
    >|  <CODE>$xPatchCode</CODE>
    >|  <CREATOR>$BenVarKuerzel</CREATOR>
    >|  <CREATIONDATE>$Datum</CREATIONDATE>
    >|</PATCHSCRIPT>
>> Speichere die Datei ab unter
    >|$DatBasis\$AKennung.xml
>> Rufe den Patch-Converter für die Datei auf:
    >|$BenVarPfadTools\PatchConverter\Bin\PatchConverter.exe
        __ "$DatBasis\$AKennung.xml"
>>  Das fertige Patchscript
    ~AddCode  $AKennung.psc
    ~AddText  liegt unter
    >|$DatBasis
```

In diesem Beispiel wird die Erstellung eines einfachen Patchskriptes angeleitet, das in einer nach Standardkonventionen benannten Aufgabenmappe als XML-Datei erstellt und mit Hilfe einer internen Hilfsanwendung in ein anderes Format konvertiert wird.

Die Anleitung kombiniert dafür verschiedene innerhalb der Anleitung abgefragte und berechnete Eigenschaften mit den benutzerabhängigen Variablen. Dazu wird die Funktionsdatei *Benutzervariablen.fps* gleich zu Beginn eingelesen, womit die Auswahlabfrage nach den Benutzervariablen als Teil des Skriptes erscheint.

Ein Anwender kann den Link auf das Skript in den Lesezeichen seines Browsers nun auch so abspeichern, dass diese erste Abfrage dort schon beantwortet ist. Die URL dafür bekommt er, indem er nur die Auswahlabfrage nach den Benutzervariablen beantwortet und auf *Weiter* klickt. In unserem Fall wird dabei der URL-Parameter `BK=AB` angefügt. Damit ist beim Aufruf gar nicht mehr erkennbar, dass das Skript benutzerabhängige Anteile hat.

Mit dem Befehl `~XmlEncode` wird im Beispiel der eingegebene Patchcode in die XML-Codierung umgewandelt, so dass die für die Verwendung innerhalb eines XML-Tags kritischen Zeichen wie `<`, `>` und `&`, sowie die Anführungszeichen maskiert werden. Bei den anderen Elementen lässt sich das Vorhandensein dieser Zeichen semantisch ausschließen.

12 Befehlsreferenz

12.1 Kernelemente

Kernelement 12.1 Eine *Bedingung* ist ein Ausdruck, der zu *wahr* oder *falsch* ausgewertet werden kann. Bedingungen werden bei verschiedenen Befehlen verwendet, z.B. in Do-While-Schleifen oder bei If-Bedingungen.

Eine Bedingung muss in *FlowProtocol 2* in der disjunktiven Normalform angegeben werden, also als Oder-Verknüpfung (`||`) von Und-Verknüpfungen (`&&`), wobei keine Klammerung notwendig ist.

Als Literale sind die Konstanten `1` und `true` (wahr), sowie `0` und `false` (falsch) verwendbar, sowie die folgenden Vergleichsoperatoren:

Ausdruck	Formel	Sprachform	Typenbeschränkung
`$s==$t`	$s = t$	s ist gleich t	für Zeichenketten s und t
`$s!=$t`	$s \neq t$	s ist ungleich t	für Zeichenketten s und t
`$x<>$y`	$x \neq y$	x ist ungleich y	für Zahlen x und y
`$x<$y`	$x < y$	x ist kleiner y	für Zahlen x und y
`$x<=$y`	$x \leq y$	x ist kleiner gleich y	für Zahlen x und y
`$x>$y`	$x > y$	x ist größer y	für Zahlen x und y
`$x>=$y`	$x \geq y$	x ist größer gleich y	für Zahlen x und y
`$s~$t`	$s \supseteq t$	s enthält t	für Zeichenketten s und t
`$s!~$t`	$s \not\supseteq t$	s enthält t nicht	für Zeichenketten s und t
`?$v`		v ist gesetzt	für eine Variable v
`!?$v`		v ist nicht gesetzt)	für eine Variable v

Kernelement 12.2 Ein *Dateipfad*, bzw. *Ordnerpfad* ist eine Zeichenkette, die einen Pfad auf eine Datei, bzw. einen Ordner angibt. Als Trennzeichen in den Pfadangaben kann das Trennzeichen des Betriebssystems verwendet werden, auf dem *FlowProtocol 2* läuft, also

\ für Windows und / für Linux, es kann aber auch | verwendet werden, das unabhängig vom Betriebssystems richtig aufgelöst wird.

Der Zugriff von *FlowProtocol 2* ist dabei beschränkt auf Dateien und Ordner, die im Skripte-Verzeichnis liegen. Die Angabe . verweist auf den Ordner, in dem das gerade ausgeführte Skript liegt. Angaben, die mit .| (bzw. .\ oder ./) beginnen, werden ausgehend vom Skripte-Verzeichnis interpretiert und alle anderen werden wieder ausgehend von dem Ordner interpretiert, in dem das gerade ausgeführte Skript liegt

Kernelement 12.3 Ein *Link* ist ein im Browser angezeigter Text, der optisch meist blau und unterstrichen hervorgehoben ist, und der bei Anklicken eine im Link hinterlegte Adresse (URL) in einem Tab oder einen neuen Fenster öffnet.
Links können in *FlowProtocol 2* sowohl in der Ausgabe, als auch in Hilfezeilen erzeugt werden, wobei stets eine URL und ein Anzeigetext angegeben werden können.

Um die Gefahr abzuwenden, dass über Links nicht erkennbar schädliche URL-Verweise in Skripten eingebaut werden, kann man festlegen, welche URLs ausschließlich durch den Anzeigetext angezeigt werden sollen. Ein Link wird im Browser ausschließlich mit dem Anzeigetext dargestellt, wenn...

- der Anfang der URL mit einem Eintrag der in der Konfiguration hinterlegten Whitelist (Datei appsettings.json, Eigenschaft `LinkWhitelist`) übereinstimmt oder
- die Whitelist leer ist oder
- auf ein anderes *FlowProtocol-2*-Skript verwiesen wird oder
- der Anzeigetext den Link als Text enthält (ggf. auch ohne "https://").

Anderenfalls wird die URL dem Anzeigetext in Klammern nachgestellt.

Die in *FlowProtocol 2* angezeigten Links werden standardmäßig in einem neuen Tab geöffnet.

Kernelement 12.4 Eine *Liste* ist in *FlowProtocol 2* eine Folge von fortlaufend indizierten Variablen, z.B. `V(1)`, `V(2)`, `V(3)`,..., die mit dem Index 1 beginnt und mit dem letzten Index endet, für den eine entsprechende Variable noch gesetzt ist. Listen werden durch verschiedene Befehle erzeugt und können selbst auch als Argumente an Befehle übergeben werden. Hierbei wird dann nur der Grundname der Variable (hier also `V`) angegeben.

12.2 Umgebungsvariablen

`$BaseKey` gibt innerhalb von Funktionen oder mit dem `~Include`-Befehl aufgerufene Funktionsdateien den Wert zurück, der mit dem gleichnamigen Argument beim Aufruf übergeben wurde. Diese Variable gibt auch bei rekursiv verschachtelten Selbstaufrufen immer den zum jeweiligen Aufruf passenden Wert zurückgibt. Mit ihr können Argumente und Variablen bei rekursiven Aufrufen derselben Funktion voneinander abgetrennt werden.

`$BaseURL` gibt die URL für das aktuell ausgeführte Skript ohne Parameter zurück.

`$Chr(...)` gibt das Zeichen mit dem angegebenen ANSI-Code zurück

`$CRLF` gibt die Zeichenkombination *Carriage Return* + *Line Feed* zurück.

`$LineNumber` gibt die Zeilennummer der aktuellen Zeile zurück. Mittels Zusatz $-n$ wird direkt der um n verringerte Wert zurückgegeben, z.B. `$LineNumber-3`.

`$NewGuid` erzeugt eine Guid, z.B. 8bceae6b-aa8b-497e-aa3a-17a6634b7215.

`$ResultURL` gibt die URL für das aktuell ausgeführte Skript mit den aktuell gesetzten Parametern zurück.

`$ScriptFilePath` gibt den Dateipfad für das aktuell ausgeführte Skript aus Sicht der Anwendung zurück.

`$ScriptPath` gibt den Dateipfad für das Skripte-Verzeichnis aus Sicht der Anwendung zurück.

`$CurrentScriptPath` gibt den Dateipfad des Ordners für das aktuell ausgeführte Skript aus Sicht der Anwendung zurück.

12.3 Befehle

`>>` [⟨Format⟩] ⟨Text⟩
: Gibt einen Text als Aufzählung der ersten Ebene aus. Als Format kann zwischen `#` für nummerierte Aufzählung, `*` für nicht nummerierte Aufzählung, `_` für Fließtext und `|` für Codezeile gewählt werden. Standardmäßig wird die nummerierte Aufzählung verwendet.

  Optional kann durch Voranstellen von `@`⟨Abschnitt⟩ ein Abschnitt ausgewählt werden.

`>` [⟨Format⟩] ⟨Text⟩
: Gibt einen Text als Aufzählung der zweiten Ebene aus. Als Format kann zwischen `#` für nummerierte Aufzählung, `*` für nicht nummerierte Aufzählung, `_` für Fließtext und `|` für Codezeile gewählt werden. Standardmäßig wird die nicht nummerierte Aufzählung verwendet.

  Optional kann durch Voranstellen von `@`⟨Abschnitt⟩ ein Abschnitt ausgewählt werden.

`>.` [⟨Format⟩] ⟨Text⟩
: Gibt einen Text als Aufzählung der dritten Ebene aus. Als Format kann zwischen `#` für nummerierte Aufzählung, `*` für nicht nummerierte Aufzählung, `_` für Fließtext und `|` für Codezeile gewählt werden. Standardmäßig wird die nicht nummerierte Aufzählung verwendet.

  Optional kann durch Voranstellen von `@`⟨Abschnitt⟩ ein Abschnitt ausgewählt werden.

`?` [⟨Schlüssel⟩] : ⟨Text⟩
: Erzeugt eine Auswahlabfrage mit mehreren Optionen, von denen genau eine ausgewählt werden kann. Der Anzeigetext wird oberhalb der Auswahlliste angezeigt. Der Schlüssel wird verwendet, um die Eingabe in der URL zu speichern, und innerhalb des Skriptes abzufragen. Er muss eindeutig sein, bzw. wird bei Wiederholung

als schon gesetzt betrachtet. Innerhalb einer Datei können die Schlüssel durch Anhängen des '-Zeichens automatisch durchnummeriert werden.

Die Optionen werden darunter mit dem #-Befehl angegeben. Über die beiden Variablen ⟨Schlüssel⟩`_OptionGroupPromt` und ⟨Schlüssel⟩`_SelectedOptionText` lassen sich der Text der Frage und der ausgewählten Option abrufen.

`#` [⟨Optionsschlüssel⟩] : ⟨Anzeigetext⟩
Fügt eine Option zur übergeordneten Auswahlabfrage hinzu. Wird der Optionsschlüssel weggelassen, wird dieser automatisch durchnummeriert. Für die jeweils gewählte Option werden die darunter eingerückten Befehle ausgeführt, für die anderen nicht.

`~AddCode` ⟨Text⟩
Hängt einen Text in der für Programmcode typischen Schriftart an die letzte Ausgabezeile an. Um dazwischen ein Leerzeichen zu erzeugen, müssen zwischen Befehlsname und Argument zwei Leerzeichen angegeben werden.
Siehe Abschnitt 4.3, verwendet in Beispiel 10, 30 und 50.

`~AddHelpLine` ⟨Text⟩
Fügt eine Hilfezeile für die übergeordnete Auswahlabfrage oder Texteingabe mit einem Text hinzu.
Siehe Abschnitt 6.3, verwendet in Beispiel 17, 30 und 45.

`~AddHelpLink` ⟨URL⟩ | ⟨Anzeigetext⟩
Fügt einen Link mit der entsprechenden URL und dem Anzeigetext in der aktuellen Hilfezeile ein (siehe auch Kernelement 12.3).
Siehe Abschnitt 6.3, verwendet in Beispiel 17.

`~AddHelpText` ⟨Text⟩
Fügt den Text ⟨Text⟩ in der aktuellen Hilfezeile ein. Um dazwischen ein Leerzeichen zu erzeugen, müssen zwischen Befehlsname und Argument zwei Leerzeichen angegeben werden.
Siehe Abschnitt 6.3, verwendet in Beispiel 17.

`~AddLink` ⟨URL⟩ | ⟨Anzeigetext⟩
Fügt einen Link mit der entsprechenden URL und dem Anzeigetext in der aktuellen Ausgabezeile ein (siehe auch Kernelement 12.3).
Siehe Abschnitt 12.3, verwendet in Beispiel 10, 28, 38 und 44.

`~AddNewKey` ⟨Schlüssel⟩ = ⟨Wert⟩
Speichert einen Wert unter einem Schlüssel in der URL, wenn der Schlüssel dort noch nicht enthalten ist.
Siehe Abschnitt 10.1, verwendet in Beispiel 37.

`~AddText` ⟨Text⟩
Fügt einen Text in der aktuellen Ausgabezeile ein. Um dazwischen ein Leerzeichen zu erzeugen, müssen zwischen Befehlsname und Argument zwei Leerzeichen angegeben werden.
Siehe Abschnitt 4.3, verwendet in Beispiel 10, 31, 34, 38 und 50.

`~AddTo` ⟨Variable⟩ += ⟨Zahlenwert⟩
Addiert eine Zahlenwert zu einer Variablen dazu. Hat die Variable noch keinen Wert, so wird der Ausgangswert 0 impliziert.
Siehe Abschnitte 5.2 und 7.2, verwendet in Beispiel 13, 16, 20, 21, 23, 24, 25, 37, 38, 43 und 48.

`~Calculate` ⟨Variable⟩ = ⟨Operand 1⟩ ⟨Operator⟩ ⟨Operand 2⟩
Führt eine einfache Rechnung aus und speichert das Ergebnis in einer Variablen. Als Operator stehen zur Auswahl `+`, `-`, `*`, `/` und `%` (Modulo).
Siehe Abschnitt 7.2, verwendet in Beispiel 19, 20, 37, 43 und 48.

`~CalculateExpression` ⟨Variable⟩ = ⟨Ausdruck⟩
Berechnet einen mathematischen Ausdruck, der aus den vier Grundrechenarten `+`, `-` , `*` und `/`, dem `^`-Operator für die Potentenzierung, dem `%`-Operator für Modulo-Rechnung, sowie den Operatoren `sqrt`, `sin`, `cos`, `tan`, `exp` und `ln` in einer beliebigen Klammerung besteht. Das Ergebnis wird einer Variablen zugewiesen. Es gilt die Punkt-vor-Strich-Rechnung. Die Argumente der trigonometrischen Funktionen werden im Bogenmaß interpretiert.
Siehe Abschnitt 7.2, verwendet in Beispiel 19, 20, 37 und 43.

`~CamelCase` ⟨Variable⟩ = ⟨Text⟩
Wandelt einen Text in CamelCase-Schreibweise um und speichert das Ergebnis in einer Variablen. Hierbei werden Whitespace, Umlaute und Sonderzeichen entfernt und durch Großschreibung des darauffolgenden Buchstaben gekennzeichnet.
Siehe Abschnitt 9.2, verwendet in Beispiel 30 und 45.

`~ClearVar` ⟨Variablenmuster⟩ Löscht alle Variablen aus den Speicher, die einem Variablenmuster entsprechen. Dies kann ein Variablenname sein oder der Anfang eines Variablennamens, gefolgt von `*`, wobei im zweiten Fall alle Variablen gelöscht werden, die auf diese Weise beginnen. Die Ausführungszeit für einzelne Befehle steigt deutlich an, wenn eine große Anzahl an Variablen, also 2000 oder mehr, verwendet werden. Durch den Befehl können ganze Listen von Variablen nach Gebrauch wieder entfernt werden, so dass die Ausführungszeit für die nachfolgenden Befehle wieder sinkt.

`~DateAdd` ⟨Variable⟩ = ⟨Ausgangsdatum⟩ | ⟨Zahlenwert⟩ | ⟨Intervall⟩
Addiert zu einem Ausgangsdatum eine Zeitspanne hinzu, die über einen ganzzahligen Zahlenwert und ein Intervall gegeben wird. Als Intervall sind folgende Zeichen zugelassen: `y` = Jahre, `M` = Monate, `w` = Wochen, `d` = Tage, `h` = Stunden, `m` = Minuten
Das Ausgangsdatum muss im internen Format für Datumswerte angegebe werden (yyyy-MM-dd HH:mm:ss).
Siehe Abschnitt 10.2, verwendet in Beispiel 38.

`~DateFormat` ⟨Variable⟩ = ⟨Datumswert⟩ | ⟨Format⟩
Wanelt einen Datumswert in ein bestimmten Format um und speichert das Ergebnis in einer Variablen. Als Format sind die in der Programmiersprache C# möglichen Formatzeichen zulässig. Der Datumswert muss im internen Format für Datumswerte angegebe werden (yyyy-MM-dd HH:mm:ss).
Siehe Abschnitt 10.2, verwendet in Beispiel 38.

`~DateSet` ⟨Variable⟩ = ⟨Wert⟩ | ⟨Format⟩
: Interpretiert einen Text als Datum in einem vorgegebenen Format und speichert das Ergebnis in einer Variablen. Als Format sind die in der Programmiersprache C# möglichen Formatzeichen zulässig. Der Ergebniswert wird im internen Format für Datumswerte gespeichert (yyyy-MM-dd HH:mm:ss).
Siehe Abschnitt 12.3, verwendet in Beispiel 38.

`~DecryptText` ⟨Variable⟩ = ⟨Text⟩ | ⟨Schlüssel⟩
: Entschlüsselt einen mit `~EncryptText` verschlüsselten Text mit dem angegebenen Schlüssel und speichert das Ergebnis in einer Variablen.

`~DefineSub` ⟨Funktionsname⟩
: Definiert eine Funktion mit dem angegebenen Funktionsnamen, die dann mit dem `~GoSub`-Befehl aufgerufen werden kann. Die Funktion muss mit `~Return` abgeschlossen werden. Siehe Abschnitt 7.5, verwendet in Beispiel 23, 37 und 43.

`~DoWhile` ⟨Bedingung⟩
: Führt die nachfolgenden eingerückten Befehle wiederholt aus, solange die Bedingung erfüllt ist. Die Schleife wird mit dem Befehl `~Loop` auf gleicher Ebene abgeschlossen. Siehe auch Kernelement 12.1.
Siehe Abschnitte 6.2 und 7.3, verwendet in Beispiel 16, 20, 21, 25, 37, 38 und 43.

`~DynamicOptionGroup` ⟨Schlüssel⟩ : ⟨Optionenliste⟩ ; ⟨Fragestellung⟩
: Erzeugt eine Auswahlabfrage mit einem Schlüssel, einer Fragestellung und einer Menge von Optionen, von denen genau eine ausgewählt werden kann. Im Unterschied zum `?`-Befehl werden die Optionen hier zur Laufzeit aus einer Liste ausgelesen, siehe auch Kernelement 12.4. Das mit dem Schlüssel in der URL gespeicherte und als Variable abrufbare Ergebnis der Auswahlabfrage ist der Index der ausgewählten Option. Der Schlüssel der Auswahlabfrage kann wie beim `?`-Befehl mit `'` per Durchnummerierung im Skript festgelegt werden. Ist die Optionenliste leer, wird der Befehl übersprungen.
Siehe Abschnitt 11.3, verwendet in Beispiel 48.

`~Else`
: Ist eine optionale Ergänzung zu einer If-Abfrage und evtl. vorangehenden ElseIf-Abfragen. Der Befehl führt die nachfolgend eingerückten Befehle aus, wenn alle vorherigen Bedingungen nicht erfüllt waren. Pro If-Abfrage darf maximal eine Else-Abfrage angegeben werden.
Siehe Abschnitt 7.4, verwendet in Beispiel 22, 25, 37, 43 und 45.

`~ElseIf` ⟨Bedingung⟩
: Prüft eine alternative Bedingung nach einer If-Abfrage und evtl. anderen vorangehenden ElseIf-Abfragen. Der Befehl führt die nachfolgend eingerückten Befehle aus, wenn die angegebene Bedingung erfüllt ist und alle Bedingungen der vorangegangenen If-, und ElseIf-Befehle nicht erfüllt waren. Es können beliebig viele ElseIf-Abfragen aneinandergereiht werden.
Siehe Abschnitt 7.4, verwendet in Beispiel 22 und 25.

`~EncryptText` ⟨Variable⟩ = ⟨Text⟩ | ⟨Schlüssel⟩
: Verschlüsselt einen Text mit dem angegebenen Schlüssel und speichert das Ergbnis in einer Variablen.

`~End`
: Beendet die Skriptausführung.
Siehe Abschnitt 7.5, verwendet in Beispiel 23 und 48.

`~EndParagraph`
: Beendet den aktuellen Absatz im aktuellen Textblock der Ausgabe. Das Hinzufügen der nächsten Zeile beginnt damit automatisch einen neuen Absatz mit einem entsprechenden Abstand.

`~EvalExpression` ⟨Variable⟩ [`!`] = ⟨Bedingung⟩
: Wertet eine Bedingung aus und speichert das Ergebnis in einer Variablen als `true` oder `false`. Mit = wird der Wert direkt zugewiesen, mit `!=` wird der Wert bei der Zuweisung negiert. Die Ergebniswerte solcher Auswertungen können in anderen Ausdrücken weiterverwendet werden. Siehe auch Kernelement 12.1.
Siehe Abschnitt 6.2, verwendet in Beispiel 16.

`~Execute`
: Führt alle bis zu dieser Stelle durchlaufenen Eingabebefehle aus, sofern für diese noch kein Wert gesetzt ist. Erst wenn alle diese Eingaben vorhanden sind, werden die nachfolgenden Befehle ausgeführt. Mit dem Befehl kann die Menge der gleichzeitig abgefragten Eingaben reduziert werden, und es kann sichergestellt werden, dass die nachfolgenden Befehle erst dann ausgeführt werden, wenn die von diesen benötigten Eingaben auch vorhanden sind.
Siehe Abschnitte 5.3, 6.2 und 11.3, verwendet in Beispiel 16, 14, 21, 22, 37, 43 und 48.

`~ExitLoop`
: Verlässt eine Schleife und setzt die Ausführung nach dem Loop-Befehl fort.
Siehe Abschnitte 7.3 und 10.2, verwendet in Beispiel 20 und 38.

`~FileExists` ⟨Variable⟩ = ⟨Dateipfad⟩
: Prüft, ob eine entsprechende Datei existiert und speichert das Ergebnis als Wahrheitswert in einer Variablen. Siehe auch Kernelement 12.2.

`~ForEach` ⟨Variable⟩ `in` ⟨Liste⟩
: Führt die nachfolgenden eingerückten Befehle bis zum dazugehörenden `~Loop`-Befehl für jedes Element der Liste aus, siehe auch Kernelement 12.4. Bei jedem Durchlauf wird das jeweilige Element der Variablen zugewiesen.
Siehe Abschnitte 7.3 und 8.1, verwendet in Beispiel 16, 20, 21, 24, 27, 43, 45 und 48.

`~ForEachLine` ⟨Variable⟩ `in` ⟨Dateipfad⟩ [`;` `Take=` ⟨Anzahl⟩]
[`;` `IndexVar=` ⟨Indexvariable⟩][`;` `SectionVar=` ⟨Abschnittsvariable⟩]
[`;` `NoFormat`]
: Liest Zeilen der über den Dateipfad angegebenen Datei ein (siehe auch Kernelement 12.2) und führt die nachfolgenden eingerückten Befehle bis zum dazugehörenden `~Loop`-Befehl wiederholt aus. Für jede durchlaufene Zeile wird die Variable mit dem Inhalt der Zeile belegt.

  Mit dem optionalen `Take`-Parameter kann die Anzahl der durchlaufenen Zeilen vorgegeben werden. Diese werden dann zufällig ausgewählt. Bei Angabe des optiona-

len `IndexVar`-Parameters wird die laufende Nummer in der Indexvariablen zurückgegeben.

Der Befehl ist standardmäßig auf das Einlesen von Datenlisten ausgerichtet, also Textdateien bei denen Leerzeilen und mit `\\` beginnende Kommentarzeilen übersprungen werden, und Zeilen, die mit einer öffnenden eckigen Klammer beginnen und mit einer schließenden eckigen Klammer enden als Abschnittskennung interpretiert werden. Für alle Zeilen werden Whitespace-Zeichen an Anfang und Ende entfernt. Bei Angabe des optionalen `SectionVar`-Parameters wird der jeweilige Abschnitt in der Abschnittsvariablen zurückgegeben.

Mit dem `NoFormat`-Argument werden alle Zeilen der Textdatei zurückgegeben, auch Leerzeilen. Es werden keine Formate wie Abschnittskennungen oder Kommentare interpretiert und kein Whitespace entfernt.

Siehe Abschnitte 8.3 und 11.3, verwendet in Beispiel 27 und 48.

`~GoSub` ⟨Funktionsname⟩ [`; BaseKey=` ⟨Basisschlüssel⟩]
Führt die angegebenen Funktion aus. Bei Angabe des optionalen `BaseKey`-Parameters wird die gleichnamige Systemvariable auf den angegebenen Basisschlüssel gesetzt. Nachdem die Funktion durchlaufen wurde, wird die Skriptausführung in der nächsten Zeile fortgesetzt.
Siehe Abschnitte 7.5 und 9.4, verwendet in Beispiel 23, 37 und 43.

`~GoTo` ⟨Sprungmarke⟩
Setzt die Skriptausführung an der angegebenen Sprungmarke fort. Diese wird mit dem `~JumpMark`-Befehl gesetzt.
Siehe Abschnitt 7.1, verwendet in Beispiel 18.

`~If` ⟨Bedingung⟩
Führt die nachfolgend eingerückten Befehle genau dann aus, wenn die Bedingung erfüllt ist (siehe auch Kernelement 12.1). Ist die Bedingung nicht erfüllt, wird der nächste Befehl auf gleicher Ebene ausgeführt.

Der If-Befehl kann mit beliebig vielen ElseIf-Befehlen und einem Else-Befehl kombiniert werden.

Siehe Abschnitt 7.4, verwendet in Beispiel 16, 20, 22, 23, 25, 27, 30, 31, 37, 38, 43, 45 und 48.

`~Implies` ⟨Schlüssel⟩ = ⟨Wert⟩
Setzt einen Wert für einen Schlüssel in der URL. Damit können z.B. Antworten für noch ausstehende Auswahlabfragen impliziert werden.

`~Include` ⟨Dateipfad⟩ [`; BaseKey=` ⟨Basisschlüssel⟩]
Führt die über den Dateipfad angegebene Funktionsdatei aus. Die standardmäßige Dateiendung für Funktionsdateien ist *.fps*. Die Syntax ist diesselbe wie in den Skriptdateien. Bei Angabe des optionalen `BaseKey`-Parameters wird die gleichnamige Systemvariable auf den angegebenen Basisschlüssel gesetzt. Nachdem die Datei durchlaufen wurde, wird die Skriptausführung in der nächsten Zeile fortgesetzt.
Siehe Abschnitt 9.4, verwendet in Beispiel 34 und 50.

`~Input` ⟨Schlüssel⟩ : ⟨Anzeigetext⟩
: Fragt eine Texteingabe ab und verwaltet diese unter dem angegebenen Schlüssel. Der Anzeigetext wird oberhalb des Eingabefeldes angezeigt. Der Schlüssel wird verwendet, um die Eingabe in der URL zu speichern, und innerhalb des Skriptes abzufragen. Er muss eindeutig sein, bzw. wird bei Wiederholung als schon gesetzt betrachtet. Innerhalb einer Datei können die Schlüssel durch Anhängen des `'`-Zeichens automatisch durchnummeriert werden.
Siehe Abschnitte 5.3 und 8.1, verwendet in Beispiel 14, 15, 16, 17, 21, 22, 24, 27, 28, 30, 34, 37, 43, 44, 45, 48 und 50.

`~JumpMark` ⟨Sprungmarke⟩
: Setzt eine Sprungmarke, die mit dem `~GoTo`-Befehl angesprungen werden kann.
Siehe Abschnitt 7.1, verwendet in Beispiel 18.

`~ListDirectories` ⟨Listenvariable⟩ = ⟨Verzeichnispfad⟩ [`; Pattern=` ⟨Suchmuster⟩]
: Listet alle Verzeichnisnamen im Verzeichnispfad auf, die dem Suchmuster entsprechen (siehe Kernelement 12.2). Das Ergebnis wird als Liste zu der Listenvariable gespeichert (siehe Kernelement 12.4). Der Eintrag mit dem Index 0 gibt die Anzahl der gefundenen Verzeichnisse zurück.
Siehe Abschnitt 11.3, verwendet in Beispiel 48.

`~ListFiles` ⟨Listenvariable⟩ = ⟨Verzeichnispfad⟩ [`; Pattern=` ⟨Suchmuster⟩]
: Listet alle Dateinamen im Verzeichnispfad auf, die dem Suchmuster entsprechen (siehe Kernelement 12.2). Das Ergebnis wird als Liste zu der Listenvariable gespeichert (siehe Kernelement 12.4). Der Eintrag mit dem Index 0 gibt die Anzahl der gefundenen Dateien zurück.
Siehe Abschnitt 11.3, verwendet in Beispiel 48.

`~Loop`
: Beendet einen Schleifendurchlauf und beginnt den nächsten Schleifendurchlauf. Eine Schleife kann durch verschiedene Befehle initiiert werden.
Siehe Abschnitt 7.3, verwendet in Beispiel 16, 20, 21, 24, 25, 27, 37, 38, 43, 45 und 48.

`~MoveSection` ⟨Ausgangsabschnitt⟩ `->` ⟨Zielabschnitt⟩
: Überträgt den Inhalt eines Ausgangsabschnitts in einen Zielabschnitt. Der Ausgangsabschnitt wird gelöscht. Ist der Zielabschnitt noch nicht vorhanden, so wird dieser am Ende angelegt. Ist der Ausgangsabschnitt nicht vorhanden, so bleibt der Aufruf ohne Wirkung. Sind Ausgangsabschnitt und Zielabschnitt identisch, so wird dieser Aschnitt entfernt. Die beien aufeinandertreffenden Textblöcke werden zusammengeführt, wenn sie die gleiche Formatierung haben.
Siehe Abschnitte 6.1 und 10.3, verwendet in Beispiel 15, 45 und 48.

`~Random` ⟨Variable⟩ = ⟨Untergrenze⟩ `..` ⟨Obergrenze⟩
: Erzeugt eine ganze Zufallszahl im Bereich von Untergrenze bis Obergrenze und speichert diese in einer Variablen. Bei Aktualisierung der Seite wird jeweils ein neuer Zufallswert generiert.
Siehe Abschnitt 8.2, verwendet in Beispiel 25.

`~RegExMatch` ⟨Listenvariable⟩ = ⟨Text⟩ `|` ⟨Regulärer Ausdruck⟩
: Wendet einen regurären Ausdruck auf einen Text an und speichert das Ergebnis

in einer Liste zur angegebenen Listenvariablen. In der Variablen mit Index 0 wird `true` zurückgegeben, wenn der Text dem regulären Ausdruck entspricht, ansonsten `false`. In den nachfolgenden Variablen mit den Indices 1, 2, ... werden die Werte der Gruppen abgelegt, die im Ausdruck enthalten sind.
Siehe Abschnitt 8.3, verwendet in Beispiel 27, 30 und 43.

`~RegExReplace` ⟨Variable⟩ = ⟨Text⟩ | ⟨Regulärer Ausdruck⟩ -> ⟨Ersetzungstext⟩
Ersetzt in einem Text alle Vorkommen, die dem regulären Ausdruck entsprechen, durch einen Ersetzungstext und speichert das Ergebnis in einer Variablen.

`~Replace` ⟨Variable⟩ = ⟨Text⟩ | ⟨Suchtext⟩ -> ⟨Ersetzungstext⟩
Ersetzt alle Vorkommen des Suchtextes in einem Text durch den Ersetzungstext und speichert das Ergebnis in einer Variablen.
Siehe Abschnitte 5.3, 7.5 und 8.2, verwendet in Beispiel 14, 23, 24, 25, 37, 45 und 48.

`~Return`
Beendet eine Funktion und setzt die Skriptausführung in der nächsten Zeile des `~GoSub`-Befehls fort, an dem die Funktion aufgerufen wurde.
Siehe Abschnitt 7.5, verwendet in Beispiel 23, 32, 37 und 43.

`~Round` ⟨Variable⟩ = ⟨Dezimalzahl⟩ | ⟨Anzahl an Stellen⟩
Rundet eine Dezimalzahl auf die angegebene Anzahl an Stellen und speichert das Ergebnis in einer Variablen.
Siehe Abschnitt 7.2, verwendet in Beispiel 19.

`~Set` ⟨Variable⟩ = ⟨Wert⟩
Speichert einen Wert in einer Variablen. Variablenbezeichnungen können aus Buchstaben, Ziffern und runden Klammern bestehen und auch wieder andere Variablen enthalten. Die in Variablennamen verwendeten Klammern haben keine syntaktische Bedeutung, können aber die Lesbarkeit von Variablen verbessern.
Siehe Abschnitt 5.1, verwendet in Beispiel 11, 12, 13, 15, 16, 20, 21, 22, 23, 24, 25, 27, 30, 34, 37, 38, 43, 44, 45, 48 und 50.

`~SetCulture` ⟨Variable⟩ = ⟨Kultur-Kennzeichen⟩
Setzt die Kultureinstellungen anhand des angegebenen Kultur-Kennzeichens, z.B. "de-DE". Diese werden z.B. bei formatierten Datumsausgaben angewendet wie der Ausgabe von Wochentagen.
Siehe Abschnitt 10.2, verwendet in Beispiel 38.

`~SetDateTime` ⟨Variable⟩ = ⟨Format⟩
Speichert das aktuelle Datum oder den aktuellen Zeitpunkt in einer vorgegebenen Formatierung in einer Variablen.
Siehe Abschnitte 10.1 und 10.2, verwendet in Beispiel 37, 38 und 50.

`~SetInputDescription` ⟨Beschreibung⟩
Setzt die Beschreibung für die Eingabeseiten. Diese wird dort unterhalb der Überschrift angezeigt.
Siehe Abschnitt 6.2, verwendet in Beispiel 16.

`~SetInputSection` ⟨Abschnittsüberschrift⟩
: Setzt die Abschnittsüberschrift für die nachfolgenden Eingabebefehle.
Siehe Abschnitt 6.2, verwendet in Beispiel 16 und 48.

`~SetInputTitle` ⟨Titel⟩
: Setzt den Titel für die Eingabeseiten.
Siehe Abschnitt 6.2, verwendet in Beispiel 16 und 37.

`~SetSection` ⟨Abschnittsüberschrift⟩
: Setzt die Abschnittsüberschrift für die nachfolgenden Ausgabebefehle. Der Befehl ist eine Alternative zu der `@`-Syntax der Ausgabebebefehle.

`~SetStopCounter` ⟨Schleifengrenze⟩ ; ⟨Befehlsgrenze⟩
: Setzt die Obergrenzen für die Anzahl der Schleifendurchläufe und der durchlaufenenen Befehle pro Seitenaufruf. Beim Überschreiten der Grenzen bricht das Programm mit einer Fehlermeldung ab. Die Grenzen dienen zur Verhinderung von Endlosschleifen und Endlosrekursionen und stehen Werksseitig auf 1000 und 20000, was für die meisten Skripte ausreichen sollte. Ist für ein komplexes Skript absehbar, dass diese Grenzen im regulären Betrieb überschritten werden, können die Grenzen mit dem o.g. Befehl hochgesetzt werden.
Siehe Abschnitte 7.3 und 10.4, verwendet in Beispiel 43.

`~SetTitle` ⟨Titel⟩
: Setzt den Titel für die Ausgabeseite.
Siehe Abschnitt 6.1, verwendet in Beispiel 15 und 16.

`~Sort` ⟨Ergebnisliste⟩ = ⟨Ausgangsliste⟩ [`To` ⟨Indexliste⟩]
: Sortiert eine Ausgangsliste alphanumerisch aufteigend und speichert das Ergebnis als Ergebnisliste (siehe Kernelement 12.4). Wird das `To`-Attribut mit einer Indexliste angegeben, so wird dort die Indexzuordung eingetragen, mit der die Umsortierung auf weitere Listen übertragen werden kann.

`~Split` ⟨Variable⟩ = ⟨Text⟩ | ⟨Trennzeichen⟩
: Teilt einen Text anhand eines Trennzeichens auf und speichert das Ergebnis als Liste (siehe Kernelement 12.4). Als Trennzeichen sind auch längere Zeichenkombinationen zulässig. Über den Eintrag mit dem Index 0 kann die Anzahl der Listenelemente abgerufen werden. Bei der Aufteilung wird Whitespace vor und nach den Trennzeichen standardmäßig entfernt. Wenn dieser erhalten bleiben soll, muss er mit `$Chr(...)`-Befehlen maskiert werden.
Siehe Abschnitte 8.1 und 10.4, verwendet in Beispiel 16, 24, 25, 43 und 45.

`~ToLower` ⟨Variable⟩ = ⟨Text⟩
: Wandelt einen Text in Kleinbuchstaben um und speichert das Ergebnis in einer Variablen.
Siehe Abschnitt 9.2, verwendet in Beispiel 30.

`~ToUpper` ⟨Variable⟩ = ⟨Text⟩
: Wandelt einen Text in Großbuchstaben um und speichert das Ergebnis in einer Variablen.
Siehe Abschnitt 9.2.

`~Trim` ⟨Variable⟩ = ⟨Text⟩
: Entfernt umschließenden Whitespace aus dem Text und speichert das Ergebnis in einer Variablen.

`~UrlEncode` ⟨Variable⟩ = ⟨Text⟩
: Wandelt einen Text in die URL-Codierung um und speichert das Ergebnis in einer Variablen.
Siehe Abschnitt 8.4, verwendet in Beispiel 28.

`~XmlEncode` ⟨Variable⟩ = ⟨Text⟩
: Wandelt einen Text in die XML-Codierung um und speichert das Ergebnis in einer Variablen. Die XML-Codierung maskiert die Zeichen `<`, `>`, `&`, sowie die Anführungszeichen, so dass der Text innerhalb von XML-Tags stehen kann.
Siehe Abschnitt 11.4, verwendet in Beispiel 50.

Index