Softwareentwicklung mit FlowProtocol

Wolfgang Maier

25. November 2022

Inhaltsverzeichnis

Vorwort

FlowProtocol ist eine Webanwendung, mit der sich Protokolle, Analysen, Anleitungen und sonstige Dokumente anhand von beliebig verschachtelten Fragestellungen erstellen lassen. Der Name leitet sich ab von Flow wie Flussdiagramm und Protokoll und verdeutlicht die ursprüngliche Intention, dass Protokolle und Checklisten nicht mehr nur fest und unveränderlich gegeben sein sollen, sondern dynamisch durch gezielte Fragestellungen auf eine bestimmte Situation hin zugeschnitten werden, und somit deutlich mehr an Präzision und Nutzwert gewinnen. *FlowProtocol* selbst ist dabei nur die Anwendung, auf der Vorlagen ausgeführt werden können, die die Fragestellungen, Ausgaben und Protokollpunkte enthalten. Ein ganz wesentlicher Aspekt bei der Entwicklung des Systems bestand darin, dass die Erstellung von Vorlagen sehr einfach, und ohne besondere Werkzeuge oder entwicklungstechnische Kenntnisse möglich sein sollte, sodass damit innerhalb einer Einrichtung eine Plattform geschaffen werden kann, die von den Mitarbeitern nicht nur als Anwender genutzt, sondern auch gepflegt und angepasst und ständig durch neue Vorlagen erweitert wird. Dies wurde dadurch erreicht, dass Vorlagen mit einem beliebigen Texteditor in einer sehr einfachen und intuitiven Syntax erstellt werden können, und die reine Ablage in einem Verzeichnis ausreicht, um diese unmittelbar in die Anwendung einzubinden. Zusätzlich wurde ein sehr umfassendes Anwenderhandbuch mit vielen Demo-Beispielen erstellt.

Das Projekt wurde von mir im Winter 2021/22 begonnen und immer wieder um kleinere Funktionen erweitert, und wird wohl auch in Zukunft noch die eine oder andere Erweiterung erfahren. Es ist unter https://github.com/maier-san/FlowProtocol frei verfügbar. Die anfängliche Intention der Checklisten-Erstellung hat sich inzwischen in verschiedene Richtungen ausgedehnt, sodass die Anwendungsfälle nun um einiges breiter gefächert sind. Ausgangspunkt der verschiedenen Erweiterungen war dabei immer ein Bedarf, den ich bei meiner Arbeit als Teamleiter eines Softwareentwicklungsteams festgestellt hatte, und dementsprechend sind speziell für dieses Arbeitsumfeld nun so viele schöne und praktische Lösungen auf Basis von *FlowProtocol* entstanden, dass es sich lohnt, diese hier ausführlich zu beschreiben. Noch ein paar Worte zu meiner Person und dem Entstehungsumfeld von *FlowProtocol*. Nach meinem Studium der Mathematik und Informatik habe ich 2002 als Softwareentwickler bei einer damals noch kleinen Firma angefangen und konnte dort seit nunmehr über 20 Jahren meine Leidenschaft für das Programmieren, analysieren und ganz besonders den Aufbau von Prozess- und Unterstützungswerkzeuge tatkräftig einbringen und mich dabei auch permanent weiterentwickeln. Stetiges Wachstum und permanente Professionalisierung erfordern es, auch die Arbeitsmethoden immer wieder so auszurichten, dass das bestmögliche Zusammenspiel aller Beteiligten zugunsten der Anwender und Kunden möglich ist, und das sieht bei mehreren Scrum-Teams mit Product Owner, UX-Unterstützung und eigener Testabteilung anders aus, als bei drei individuellen Entwicklern, die sich Fragen und Antworten noch über den Schreibtisch hinweg zurufen können. Hinzu kommen definierte Abläufe und Prozesse, vielfach eingefordert durch die Normen ISO 9001 und ISO 27001 und nicht zuletzt jede Menge an Wissen, Erfahrung und Best-Practice, die es zu ordnen und gezielt zum Einsatz bringen gilt. Inzwischen angekommen in einer geteilten Rolle aus Teamleiter, Softwarearchitekt und technischer Product Owner habe ich den Vorteil, dass ich unmittelbar von den Teamkompetenzen profitieren kann, für deren Aufbau ich ebenfalls verantwortlich bin, und damit ein sehr direktes Feedback bekomme. Als wichtigste Erfahrungen kann ich hier schon vorwegnehmen, dass die Arbeitsatmosphäre wohl der wichtigste Faktor für ein gut

funktionierendes und effektives Team ist, aber gleich danach das Umfeld kommt, das eine möglichst reibungsfreie und direkte Umsetzung der eingebrachten Arbeitsenergie ermöglicht, und diese weder verschwendet, noch an unnötigen Barrieren aufreibt. Letzteres kann bei der oben genannten Menge an Vorgaben und Rahmenbedingungen auf jeden Fall zu einer Herausforderung werden.

1 Die Situation im Entwicklungsumfeld

In diesem Abschnitt möchte ich genauer beschreiben, warum *FlowProtocol* gerade im Umfeld der Softwareentwicklung gut eingesetzt werden kann. Die Tatsache, dass dort vermutlich die meisten Arbeitsplätze EDV-Arbeitsplätze sind, die von einer fähigen IT-Abteilung betreut werden, bietet schon mal gute Voraussetzungen. Damit das System jedoch allgemein als Plattform innerhalb der Firma genutzt und auch von allen Mitarbeitern aufgebaut und weiterentwickelt wird, braucht es Menschen, die Abläufe und Informationen zu einem gewissen Grad abstrahieren und in eine maschinell verwendbare Form bringen können, was ja sozusagen die Kernkompetenz der Softwareentwicklung ist. Umgekehrt hilft einem diese Form der Abstraktion aber auch wieder dabei, einen Gegenstand besser zu durchdringen, und dementsprechend habe ich mir schon zu Beginn meiner Berufstätigkeit folgendes Motto als Zitat für ein Profilbild ausgesucht: „Erst wenn man einen Sachverhalt von Grund auf verstanden hat, ist man in der Lage, ihn einer Maschine beizubringen.“

1.1 Arbeiten mit Informationen

Bei Softwareentwicklung denkt man zwangsläufig zunächst einmal an Softwareentwickler, die Programmcode schreiben, der dann irgendwie vertrieben und ausgeliefert wird. So essenziell dieser Teil der Arbeit ist, ist er trotzdem nur ein Teil in der langen Kette der Produktentwicklung, an der viele Leute aus unterschiedlichen Fachrichtungen mit jeweils spezialisierter Arbeitsweise und eigenem Expertenwissen beteiligt sind, die Hand in Hand zusammenarbeiten müssen. Angefangen von der Bedarfserfassung und der Priorisierung der Projekte, weiter über die Ausarbeitung neuer Lösungen durch die Product Owner mit Unterstützung von Kollegen aus den Bereichen Design und Usability, die Zusammenarbeit mit den Entwicklungsteams, die Kontrollinstanzen des Qualitätsmanagements, Dokumentation, Marketing und Vertrieb, Dienstleistungen und Projektmanagement, bis hin zu Support und Hotline für die Betreuung bei Problemen, arbeiten alle primär mit Informationen, die erfasst, verstanden, bewertet, ergänzt und vielfach in eine neue Form gebracht werden müssen. In vielen Bereichen wird zudem kreativ gearbeitet, d.h. Dinge werden auf dem sprichwörtlichen weißen Blatt Papier neu erstellt. Grundsätzlich gibt es in jedem Bereich zahlreiche Aufgaben, die sich gut mit *FlowProtocol* umsetzen lassen, so dass man dieses Werkzeug wie das in jenem Firmen-Netzwerk zu findende interne Wiki gleich von Anfang an für alle Mitarbeiter verfügbar machen kann.

1.2 Kommunikation und Teamzusammenarbeit

Die oben beschriebene Prozesskette zeigt schon sehr deutlich, dass die Zusammenarbeit zwischen Personen und insbesondere Teams ein sehr wichtiger Aspekt ist, der einen enormen Einfluss auf die Effizienz und Qualität der Arbeit hat. Aus eigener Erfahrung weiß

ich, dass die Zusammenarbeit innerhalb der Teams meist ohne viel Mühe sehr gut funktioniert, und durch die hierarchische Personalstruktur dort auch ausreichend Unterstützung erfährt. Bei der Zusammenarbeit zwischen den verschiedenen Bereichen gestaltet sich das oft schwieriger, was schon damit anfängt, dass die Kommunikationswege abstrakter und formaler sind. Wenn jeder Mitarbeiter aus Bereich A seine Anfrage willkürlich an einen selbst gewählten Mitarbeiter aus Bereich B adressieren würde, wäre das Chaos vorprogrammiert. Für eine geregelte und verlustfreie Abarbeitung von Anfragen und Aufgaben werden dementsprechend Systemen und Posteingänge eingerichtet, die unabhängig von Kenntnissen über die Personalsituation des Zielbereiches genutzt werden können, da Zuweisungen und Vertretungssituationen bereichsintern geregelt werden. Standardmäßig verwendet man hierfür ein Ticketsystem, das mit statusbehafteten Vorgängen arbeitet, und den beteiligten Personen zu jedem Zeitpunkt Zugriff auf den aktuellen Informationsstand bietet, was man mit dem Weiterleiten von persönlichen E-Mails nicht mal ansatzweise abdecken kann.

Ganz ohne Wissen des Zielbereiches kommt man jedoch trotzdem nicht aus. Die Übermittlung von Informationen zwischen Bereichen birgt immer die Gefahr von Verfälschungen und Verlusten. Schon das Weglassen einer kleinen Information kann die Verarbeitung eines Vorgangs um ein großes Maß erschweren. Wird bei einem Bug-Report, der vom Support an die Entwicklungsteilung übergeben wird, beispielsweise die Versionsnummer weggelassen, kann das dazu führen, dass unnötigerweise mehrere Versionen überprüft werden müssen, um den Fehler zu finden, was mit viel zeitlichem Aufwand verbunden ist. Die Abfrage und Bereitstellung der Versionsnummer wäre dagegen in wenigen Sekunden erledigt, und damit auch dann sinnvoll, wenn noch nicht klar ist, ob man sie wirklich benötigt. Entsprechend ist es auch meist nicht die Zeiteinsparung die zu einem solchen Versäumnis führt, sondern einfach das fehlende Wissen um die Bedeutung, die bestimmte Informationen innerhalb der Arbeitsabläufe in den anderen Bereichen haben. Manche Firmen wenden viel Geld und Mühe auf, um mit Hilfe von externen Beratern den Punkt herauszufinden, an dem die Mitarbeiter den größten Engpass sehen, und landen so ebenfalls bei dem Punkt Kommunikation, oder, wenn der Berater sein Geld wert ist, bei teamübergreifender Kommunikation und lateraler Führung.

Im oben genannten Beispiel ist es auch naheliegend, den Aufbau von Wissen über Abläufen und Informationen in anderen Bereichen einzufordern, da dies verständnisfördernd ist und es immer besser ist, wenn man weiß, aus welchem Grund man Dinge macht. In einem so komplexen Gebiet wie der Softentwicklung kommt man hierbei allerdings schnell an die Grenzen, denn zumeist gibt es viele potentielle Informationen, von denen je nach Situation nur wenig relevant sein können. Mit welchen Tests kann ein bestimmter Fehler schon auf dem Ausgangssystem gut eingegrenzt werden? Wann benötige ich sogar die Bereitstellung einer Datenbanksicherung? Mit der Zeit werden Mitarbeiter dafür ein Gespür entwickeln können, was aber einen aufwendigen Lernprozess und entsprechend viele Fehler voraussetzt. Wir werden später noch deutlich komplexere Beispiele zu diesem Thema sehen.

Die Alternative dazu sieht so aus, dass man die benötigten Informationen einfach generell abfragt, zum Beispiel in Form von Pflichtfeldern oder Checklisten oder der Zurückweisung unvollständig ausgefüllter Vorgänge. Wenn Informationen nur noch über den Zaun geworfen werden, verliert man zwangsläufig einen wichtigen Teil der Zusammenarbeitskultur und läuft auch Gefahr, dass die Sinnhaftigkeit der Arbeit infrage gestellt wird, was nicht zu unterschätzen ist. Bei immer gleich ablaufenden Tätigkeiten besteht zumindest die Chance, dass auch die dafür benötigten Informationen immer die gleichen sind, und

man am Ende mit einen Formular am besten bedient ist. Sobald die Aufgabenstellung jedoch variieren kann und man auf Fallunterscheidungen reagieren sollte, bekommt man hier das Problem, dass man entweder nicht benötigte Informationen abfragt, oder insgesamt auf die Abfrage der für die Spezialfälle benötigten Informationen aus dem gleichen Grund verzichtet.

Unabhängig davon ist das roboterartige Abfragen aller potentiell benötigten Informationen im Allgemeinen auch nicht unbedingt wirtschaftlicher, als bei Bedarf über eine Rückfrage in eine zweite Iteration zu gehen, insbesondere, wenn man den zeitlichen Aufwand und die damit einhergehende Zufriedenheit des Anwenders entsprechend hoch bewertet.

1.3 Kategorisierungshilfen und Unterstützungssysteme

Wie kann man also allgemein kommunizieren, welche Informationen man in welcher Situation am zwingend benötigt oder welche nächsten Schritte sich abhängig von bestimmten Voraussetzungen unmittelbar anschließen?

FlowProtocol bietet die Möglichkeit die auf den internen Abläufen eines Bereiches basierenden Entscheidungsketten in einer formalen Form abzubilden und als interaktives Hilfsmittel nach außen bereitzustellen, sodass sie auch von Mitarbeitern ohne dieses spezielle Fachwissen verwendet werden können. Die vorliegende Situation wird durch eine Reihe von Multiple-Choice-Fragen erfasst, wobei es die Möglichkeit zur Verschachtelung erlaubt, je nach Teilantwort beliebig in die Tiefe zu gehen, und bekommt am Ende der Ausführung dafür genau die Informationen, die auf diese Situation passen. Das entspricht auch einer der wesentlichen Grundideen der Anwendung, nämlich Wissen für andere nutzbar zu machen, ohne dass es dafür vermittelt und erworben werden muss. Der wesentliche Vorteil ist hierbei, dass der Verwender die Vorlage als Hilfestellung und nicht als fehlerbedingte Iteration wahrnimmt, und sich sicher sein kann, dass die abgefragten Informationen auch wirklich benötigt werden.

Begründet man die Abfrage der Informationen zusätzlich durch entsprechende Erläuterungstexte, die man ebenfalls über die Vorlage anzeigen lassen kann, fördert man zusätzlich den Transfer des zugrundeliegenden Wissens und befähigt die Mitarbeiter damit zusehmend, die Situationen auf Basis von Verständnis zu bewältigen, was man auf jeden Fall anstreben sollte.

Ein klassisches Beispiel für diese Form der Anwendung bilden sogenannte Unterstützungssysteme für die Problemanalyse, die sich sehr gut mit *FlowProtocol* aufbauen lassen. Man spricht hier auch von einem einfachen Expertensystem und meint eine interaktive Anwendung, die eine vorliegende Situation sukzessive mit Hilfe von Fragen erfasst und dazu passende Handungsoptionen anbietet. Bei der Problemanalyse in Verbindung mit einem Softwareprodukt besteht die Zielsetzung darin, die Einschränkung nach Möglichkeit durch Anwendung direkter Maßnahmen zu beheben, und falls dies nicht möglich ist, alles Notwendige an Informationen zu erfassen, um das Problem im Rahmen eines Vorgangs zielgerichtet und möglichst ohne weitere Iteration beheben zu können.

Dies gelingt, indem man zunächst die Art des Fehlers genau kategorisiert und dazu die Fehlerstelle so präzise wie möglich beschreibt. Ersteres kann z.B. so aussehen:

> *Um welche Art von Fehler handelt es sich?*
>
> ◯ *Ausnahmefehler mit Ausnahmecode. Es wird eine Fehlermeldung mit einem achstelligen Fehlercode angezeigt.*

- ◯ *Inhaltlicher Fehler. Ein angezeigter oder ausgegebener Wert entspricht nicht den Erwartungen.*
- ◯ *Fehler in der Zugriffssteuerung. Die Sichtbarkeit oder der Zugriff auf bestimmte Daten entspricht nicht den Erwartungen.*
- ◯ *Technischer Fehler. Die Anwendung startet nicht, stürzt ab oder zeigt ungewollte Auffälligkeiten an der Programmoberfläche.*
- ◯ *Performanceproblem. Das Öffnen von Programmbereichen oder Dialogen oder die Ausführung von Funktionen dauert unverhältnismäßig lange und behindert das flüssige Arbeiten mit der Anwendung.*
- ◯ *Sonstiger Fehler.*

Für die Erfassung der Fehlerstelle kann etwa die Menüstruktur der Anwendung als verschachtelte Auswahl nachgebildet werden. Der Vorteil dabei liegt weniger in der Zeitersparnis gegenüber einer Tastatureingabe, sondern vielmehr in der Einheitlichkeit der resultierenden Ausgabe. Bei der Softwareentwicklung strebt man die Verwendung einer allgegenwärtigen Sprache an und meint damit, dass der Sprachgebrauch rund um die Kerndomäne der Anwendung über alle Bereiche der Firma und Entwicklung hinweg durchgängig ist, und einer einheitlichen Terminologie folgt, was sich jedoch nur mit sehr viel Mühe bis zur letzten Konsequenz durchsetzen lässt. Gerade im Zusammenhang mit Supportfällen wird allzugern der Sprachgebebrauch des Kunden übernommen und selbst an der Programmoberfläche klar benannte Elemente und Menüpunkte werden teilweise sehr phantasievoll umbenannt. Für einen erfahrenen Mitarbeiter mag die Zuordnung zwar auch dann noch gut möglich sein, doch eine Volltextsuche im System kommt hier sehr schnell an ihre Grenzen. Die Wiederauffindbarkeit von Informationen in Vorgängen ist eine wichtige Qualität, die viele Stunden Zeit einsparen kann, und die leider oft vernachlässigt wird.

Das Beispiel oben zeigt schon recht gut, wie tief und systematisch das Wissen über potentielle Probleme in der Software sein muss, um eine zielführende und vollständige Kategorisierung hinzubekommen, und es lässt auch erahnen, mit welchen Fragen man für jeden der Punkte in die Tiefe gehen könnte.

Im nächsten Schritt kann man versuchen, Eigenschaften des Systems abzufragen und Abhängigkeiten zu finden, die das Auftreten des Problems beeinflussen oder auslösen, wie Programmbenutzer oder Arbeitsplatz. In Abhängigkeit davon kann man vielleicht auch schon bestimmte Sofortmaßnahmen ausmachen, die potentiell hilfreich sein könnten. Diese lassen sich ebenfalls als Frage formulieren und führen so auch bei schon versuchter erfolgloser Anwendung zumindest zum Festhalten dieser Information, was ansonsten nicht immer geschieht.

Wurde schon versucht, die Benutzereinstellungen zurückzusetzen?

- ◯ *Ja, jedoch ohne Erfolg*
- ◯ *Nein, diese Maßnahme steht noch aus.*
- ◯ *Nein, diese Maßnahme wird nicht als zielführend angesehen.*
- ◯ *Nein, diese Maßnahme ist aktuell nicht möglich.*

Je nach Antwort bekommt man damit am Ende der Ausführung entweder die Protokollierung, dass man den entsprechenden Schritt ohne Erfolg durchgeführt hat oder die Auflistung dieser Maßnahme in einer Liste unter der Überschrift „Mögliche nächste Schritte“.

Der Aufbau eines tief verschachtelten Expertensystems wird meist nicht in einem Anlauf gelingen und gerade am Anfang werden sich zusätzliche Iterationen dadurch nicht

vermeiden lassen, da man immer wieder Fälle haben wird, die durch die Vorlage nicht abgedeckt sind, und auch noch später werden immer wieder Sonderfälle auftreten, die nicht vollständig durch das allgemeine Schema der Vorlage abgebildet werden, was bei der Anwendung der eigenen Erfahrung allerdings auch nicht anders ist. Wichtig ist es, jetzt die Fälle zu erkennen, die man noch in die Vorlage einarbeiten sollte, und das unmittelbar auch zu tun. Auf diese Weise kann schon der nächste Mitarbeiter in der nächsten Minute von der Verbesserung profitieren, ohne dass er überhaupt am zugrundeliegenden Austausch beteiligt war.

1.4 Der Umgang mit Fachwissen

Erfahrung und Fachwissen sind zwei weitere wichtige Säulen, die den Prozess der Softwareentwicklung im Wesentlichen bestimmen. Natürlich sind auch viele operative Dinge entscheidend, aber um innovative Lösungen zu finden und diese mit modernen Mitteln umsetzen zu können, braucht es Fachwissen auf Höhe der Zeit und möglichst viel Erfahrung im Umgang damit, und in einer Branche, in der sich die eingesetzten Technologien und Methoden so schnell weiterentwickeln, wie bei der Entwicklung von Anwendungen, ist das eine besondere Herausforderung. Entsprechend ist es wichtig, dem Wissenserwerb und der Wissensvermittlung ausreichend Raum und auch Freiraum zu geben. Für Ersteres gibt es Fortbildungen und sehr gute Online-Portale mit tausenden von gut gemachten Tutorials. Die interne Vermittlung von Wissen und Erfahrung muss dagegen meist vollständig in Eigenregie organisiert werden, zum Beispiel in Form von kleinen hausinternen Fachkonferenzen, was auf jeden Fall einen besonderen Charme, aber auch seine Tücken hat. Der Umfang an benötigtem Wissen ist groß, und was man nicht regelmäßig benötigt, wird man wieder vergessen. Insofern wird man nicht umher kommen, dass Wissen in irgendeiner Form festzuhalten und bei Bedarf abrufbar zu machen.

Traditionell wird Wissen in Texten festgehalten, die dann später von denen gelesen werden, die es benötigen. Das ist schon bei der Erstellung mit viel Aufwand verbunden, da je nach Umfang und Komplexität des Gegenstandes viel Text mit zusätzlichen Beispielen und Materialien notwendig ist, um wirklich alles zu vermitteln. Entsprechend mutiert das Durchlesen dann auch eher zum Durcharbeiten und setzt so die Schwelle für die Auseinandersetzung mit dem Thema unnötig hoch. Es ist besser, hier andere Medien zu einzusetzen.

Erfahrungsgemäß orientierten sich Mitarbeiter bei der Anwendung von Wissen lieber an einem Beispiel, als an einer theoretischen Beschreibung. Was liegt also näher als die Theorie direkt an einem bereits umgesetzten Anwendungsfall zu erklären, und das nicht schriftlich, sondern in Form einer Screencast-Aufzeichnung, bei der man einfach auf die wichtigsten Stellen zeigt und diese kurz erläutert, was ohne große Vorbereitung und mit wenig Zeitaufwand möglich ist. Ergänzt man das Ganze noch durch eine kurze Wikiseite, auf der man die Grundidee und die wesentlichen Stichpunkte notiert, kann man damit im Laufe der Zeit eine beachtliche Bibliothek an Wissenseinheiten aufbauen, die sich auch noch gut überblicken und auffinden lassen. Auch das Anschauen einer solchen Aufzeichnung ist natürlich deutlich einfacher, als die Durcharbeitung einer langen schriftlichen Dokumentation und bietet darüber hinaus auch noch die Wahl, in welcher Tiefe man sich in das Thema einarbeiten möchte. Verwendet man die Aufzeichnung in Kombination mit den Artefakten des dort beschriebenen Anwendungsfalls, so hat man alles was man benötigt, um das dort vermittelte Wissen auf eine neue Situation anzuwenden.

Wichtig ist, die Dinge an dieser Stelle schlank zu halten. Die Länge der Aufnahmen sollte 10 Minuten nicht überschreiten, und die müssen nur inhaltlich richtig, aber in keinem Fall perfekt präsentiert sein. Entsprechend sollten weder Schnittprogramme oder Nachvertonungen zum Einsatz kommen. Eine solche Aufzeichnung sollte ohne großen Schmerz weggeworfenen und neu gemacht werden können, wenn sich etwas ändert. Auch die Zusatzdokumentation im Wiki sollte sich auf die wesentlichen Stichpunkte beschränken. Im Idealfall reicht diese sogar aus, wenn man sich später erneut mit dem Gegenstand auseinandersetzen muss.

Dieses Beispiel zeigt gut, wie wichtig das Medium bei der internen Wissensvermittlung ist. Eine funktionierende und aktive Wissensvermittlung steht und fällt mit der Bereitschaft der Mitarbeiter, ihr Wissen zu dokumentieren, und sich mit dem dokumentierten Wissen anderer Mitarbeiter auseinanderzusetzen. Das geschieht nur dann, wenn die Hürden dafür niedrig, und der Nutzen klar erkennbar ist. Unter den richtigen Bedingungen kann hier sogar eine richtige Kultur der Wissensvermittlung entstehen, die gleichermaßen effektiv, als auch ressourcenschonend ist.

Auch *FlowProtocol* ist unnwillkürlich ein Medium zum Umgang mit Wissen und ähnlich wie oben ist auch hier die Niederschwelligkeit ein entscheidender Faktor. Dieser besteht beim Aufbau einer umfangreichen Vorlage im Wesentlichen darin, dass diese eben nicht in einem Rutsch erstellt werden muss, sondern ausgehend von einer ersten Version bei Bedarf Stück für Stück erweitert werden kann. Oft ist der Anfang das schwierigste, deshalb ist es wichtig, diesen möglichst einfach zu gestalten. Im Gegensatz zu einer Screencast-Aufzeichnung ist eine *FlowProtocol*-Vorlage jedoch eher ungeeignet, um vollständig neues Wissen zu vermitteln, sondern vielmehr dafür gedacht, um bei der Anwendung von vorhandenem Wissen anzuleiten und zu unterstützen. Allein das Verstehen und die richtige Beantwortung der in der Vorlage gestellten Fragen setzen ja schon eine gewisse Fachlichkeit voraus, die erst erworben werden muss. Ob nun in Form eines Unterstützungssystems, eines Konfigurators oder einer interaktiven Anleitung, jede *FlowProtocol*-Vorlage enthält essentielles Systemwissen, die von dort wieder relativ bequem abgerufen oder angewendet werden kann. Idealerweise kombiniert man also die Vermittlung des dafür notwendigen Grundwissens in der herkömmlichen Form mit dem Aufbau passender Vorlagen, die dabei helfen, dass Wissen auf einem höheren Level einfach und präzise anzuwenden.

1.5 Wiederkehrende Muster und Konfigurierbarkeit

Konfiguratoren funktionieren ähnlich wie die schon beschriebenen Unterstützungssysteme, nur dass anstelle einer Situation Anforderungen abgefragt werden, in deren Abhängigkeit bestimmte Schritte durchzuführen sind oder nicht. Jeder kennt vermutlich die Konfiguratoren der Fahrzeughersteller, die die Zusammenstellung eines Neuwagens und die Auswahl der gewünschten Extras ermöglichen. Am Ende der Konfiguration erhält man ein Bild seines Traumautos samt Preiskalkulation, sowie eine Liste, mit der man direkt zum Händler gehen kann. Dies ist möglich, weil die Auswahl innerhalb klarer Vorgaben variiert und sich dort klare Abhängigkeiten abbilden lassen, wie z.B. die verfügbare Motorisierung für ein bestimmtes Fahrzeugmodell oder der Preisaufschlag für Nebelscheinwerfer.

Entwicklungen zur Erweiterung einer großen Software funktionieren meist analog ähnlich und orientieren sich idealerweise an bewährten Mustern, die schon an anderen Stellen eingesetzt wurden, und die relativ schnell mit dem vorhandenen Framework auf weitere

Stellen oder neue Anwendungsfälle übertragen werden können. Klassische Beispiele sind Listen-Steuerelemente für die Darstellung von Daten, mit der Möglichkeit diese zu suchen und zu sortieren, oder API-Endpunkte mit Parametern, Autorisierungsbedingungen und Rückgabe-Codes. Auch hier kann es Variationen, zwischen denen man auswählen kann, und auch hier lässt sich ein Preis in Form von Aufwand berechnen. Ansonsten geht es wie im Fall oben darum, die für eine spezielle fachliche Arbeit benötigten Informationen abzufragen und bereitzustellen.

In der am Scrum-Prozess ausgerichteten Softwareentwicklung übernimmt normalerweise der Product Owner die Aufgabe, die Anforderungen mit den verfügbaren Umsetzungsmustern abzugleichen und gegebenenfalls in die passende Zielform zu bringen, nicht zuletzt, um die Einheitlichkeit der Anwendung aufrechtzuerhalten, und der Aufbau eines Konfigurators scheint sich dafür nicht unbedingt zu rechnen. Umgekehrt hat man die wenigen Abhängigkeiten eines Musters relativ schnell in einer Vorlage abgebildet und profitiert so von einem System, dass sowohl bei der präzisen Formulierung, als auch bei einer standardisierten Form der Umsetzung unterstützt. Je nach Aufbau kann der Konfigurator auch direkt durch die Kollegen genutzt werden, die dem Product Owner die Anforderungen zutragen, so dass die darüber ausformulierten Entwicklungen nur noch geprüft, priorisiert und eingeplant werden müssen.

Einmal festgelegte Standards für Form und Umsetzungen werden gleichermaßen fest in die Konfiguration eingearbeitet und erlauben dahingehend keine Abweichungen. Es entfallen mühsame Diskussionen an Stellen, wo die Dinge schon entschieden sind und die Entwicklungsbausteine können in der Form verwendet werden, wie es vorgesehen ist – ein geradliniger Implementierungsprozess ohne unnötige Ecken und Kanten.

Im Abschnitt 4 wird die Idee des Konfigurators nochmals aufgegriffen und mit der Möglichkeit verknüpft, die aus einer Konfiguration resultierenden Entwicklungsschritte innerhalb einer Vorlage zu erstellen.

2 Bewertungen

Wie wichtig ist eine Anforderung, wie gravierend ein Fehler und wie gut passt eine Entwicklung in eine vorgegebene Strategie? Im Entwicklungsalltag treten permanent Fragen auf, bei denen die Antwort die Grundlage für wichtige Entscheidungen ist, und die das Handeln ganzer Teams bestimmen. Meistens handelt es sich dabei um Einschätzungen oder Bewertungen von bestimmten Situationen und allzu oft erfolgt die Beantwortung dann größtenteils nur auf der Grundlage von Intuition und Bauchgefühl und ist im Nachhinein nicht nachvollziehbar.

2.1 Entscheidungsprozesse

Der Umgang mit Bewertungen hat nicht nur einen großen Einfluss auf die tatsächliche Festlegung der Arbeit, sondern auch darauf, wie diese wahrgenommen wird. Entscheidungen, die transparent und im Detail nachvollziehbar sind, haben generell eine höhere Akzeptanz als solche, die einfach nur von oben festgelegt werden. Entsprechend wird auch die Arbeit an einem Projekt anders ablaufen, wenn die hohe Priorität des Projektes für den Mitarbeiter gut nachvollziehbar ist, und er nicht den Eindruck hat, eigentlich an etwas wichtigerem Arbeiten zu müssen. Unsicherheit hinter Entscheidungen bremst

unwillkürlich das Handeln aus, unabhängig davon wie klar die Entscheidung getroffen wurde. Dabei ist das Vorhandensein einer bewusst getroffenen, und bestenfalls dokumentierten und kommunizierten Entscheidung schon mal eine gute Ausgangssituation, selbst wenn diese weder begründet oder durchdacht ist. Nicht selten entstehen entscheidungswürdige Zustände auch vollständig durch Eigendynamik, und im Nachhinein fragt man sich, wer eigentlich das Ruder in der Hand hat. Der Ausgangspunkt kann ein einfacher Vorschlag sein, der am Ende nur mangels Alternativen, tagesaktueller Begeisterung oder aus Gründen der Konfliktvermeidung umgesetzt wird.

Obwohl die Entscheidungsprozesse innerhalb eines Bereiches zu den wichtigsten Prozessen gehören, werden sie selten formal als solche abgebildet. Dies gründet vermutlich hauptsächlich darauf, dass das Treffen von Entscheidungen mehr als Privileg oder Verantwortung der höheren Führungsebenen gesehen wird, und man sich hier eher nicht einem Prozess unterordnen möchte. Diese Einstellung spiegelt sich dann auch generell im Umgang mit Entscheidungen wider, und lässt sich oft schon daran erkennen, dass Entscheidungen nur dann getroffen werden, wenn jemand bewusst darum bittet, und das obwohl die Situationen regelmäßig mit klaren Abhängigkeiten im Geschäftsalltag auftreten. Vermutlich ist so die Nachbestellung neuer Kaffeebohnen vielerorts deutlich systematischer geregelt, als die strategisch relevanten Entscheidungsprozesse.

Hat man jedoch erst einmal erkannt, an welchen Stellen Entscheidungen benötigt werden, ist schon mal die erste Hürde überwunden und es muss noch geklärt werden, wer die Entscheidung unter welchen Optionen auf Basis welcher Grundlage treffen soll.

Der Anspruch, Entscheidungen objektiv und gemeinschaftlich zu treffen, ist weit verbreitet. Dies hat sicher auch damit zu tun, dass der Ärger für die Verantwortung einer schlecht getroffenen Entscheidung schwerer wiegt, als die Anerkennung für eine gut getroffene. Unabhängig davon gelten mehrheitlich getroffene Entscheidungen jedoch als besser, was sicher auch an dem größeren Maß an Auseinandersetzung liegt, das dafür notwendig ist. Wichtig ist die Unterscheidung zwischen Gemeinschaftlichkeit und Objektivität. Eine gemeinschaftlich getroffene Entscheidung muss nicht objektiv sein und eine objektiv getroffene Entscheidung kann auch alleine getroffen werden. Beides sind Faktoren, die das Treffen der Entscheidung weg vom subjektiven Bauchgefühl einer einzelnen Person rücken und damit sowohl Legitimation, als auch Qualität erhöhen sollen.

Dass das Treffen einer Entscheidung überhaupt eine Qualität im Sinne von gut oder schlecht haben kann, ist nicht offensichtlich, und dass diese zusätzlich objektiv bestimmbar oder eindeutig definiert ist, noch weniger. Nehmen wir folgendes Beispiel:

> *Wohin sollen wir heute zum Mittagessen gehen?*
>
> ◯ *Zum Italiener*
> ◯ *Zum Griechen*
> ◯ *Zur Pommes-Bude um die Ecke*

Je nachdem, worauf ich Lust habe, was ich die letzten drei Tage zum Mittagessen hatte, was ich bereit bin auszugeben, wie viel Zeit ich habe und wie weit ich dafür gehen möchte, kann meine Antwort unterschiedlich ausfallen. Es kann aber auch durchaus sein, dass mir alle drei Optionen gleichermaßen zusagen und ich keinen Favoriten benennen könnte. Eine objektive schlechte Wahl gäbe bei diesem Beispiel dann, wenn alle Beteiligten nur 45 Minuten Zeit hätten, der Besuch beim Griechen aber mindestens eine Stunde in Anspruch nehmen würde.

Es sind also die Folgen und Effekte, anhand derer wir die Qualität bemessen, und die können vielfätig sein. Generell ist eine schlechte Wahl leichter auszumachen, als eine gute, sowohl im Vorfeld, als auch im Nachhinein. Die Möglichkeit zu einem tatsächlichen Vergleich zweier Optionen ist in Normalfall nicht gegeben.

Im besten Fall trifft man Entscheidungen mehrheitlich und objektiv. Während Ersteres nur organisatorisch eine Herausforderung ist, die inzwischen mit vielen guten Abstimmungstools gelöst werden kann, ist Objektivität schwieriger zu erreichen. Nachweisliche Objektivität gelingt meist nur mit Kriterien, die man zuerst aufstellen und teilweise begründen muss, was manchmal recht schwierig ist. So bleibt es oft dabei, dass die Entscheidung für oder gegen eine Sache zwar mit einer augenscheinlich objektiven Begründung getroffen wird, man jedoch nicht näher auf die Details der dahinterliegenden Wirkungsweise eingeht. Ein typischer Vertreter einer solchen Begründung ist der Satz „Das ist genau das, was der Kunde braucht."

Oft sind subjektive Entscheidungen emotionsgetrieben und man muss sich anstrengen, um sich hier auf die Sachlichkeit zu beschränken.

2.2 Arbeiten mit Kriterien

Wie schon gesehen sind Kriterien ein wichtiges Werkzeug, wenn es darum geht, Dinge und Situationen in Bezug auf festgelegte Aspekte möglichst objektiv zu bewerten und einzustufen. Deren Aufstellung ist sowohl fachlich, als auch methodisch anspruchsvoll, und kann viel Zeit in Anspruch nehmen.

Der große Vorteil bei der Arbeit mit Kriterien ist der, dass sich diese immer wieder verwenden lassen, und jede darin investierte Zeit also vielfachen Nutzen bringt. Ein weiterer großer Vorteil ist der, dass die Arbeit mit Kriterien in großem Umfang dazu beiträgt, Sachverhalte, Zusammenhänge und Denkweisen tiefgreifend zu verstehen und zu vermitteln, insbesondere wenn man diese auch in der Kommunikation berücksichtigt. Wenn die Entscheidung für die Umsetzung von Projekt X damit begründet wird, das dieses im Gegensatz zu Projekt Y für die Hauptzielgruppe essentiell ist, hat man nicht nur eine Begründung, sondern auch den direkten Bezug zur Produktstrategie und – falls notwendig – einen guten Anlass, mal wieder über die Zielgruppe und deren Bedarf nachzudenken.

Kriterien sind die Einzelaspekte, die in ihrer Gesamtheit die Qualität oder Eignung einer Sache für eine bestimmte Zielverfolgung ausmachen, und die damit eine Begründung für Planungsentscheidungen liefern. In geeigneten Fällen lassen sich Kriterien sogar gewichten und am Ende sogar mit einer Metrik beaufschlagen, so dass man verschiedene Gegenstände direkt über einen Zahlenwert vergleichen kann.

Unabhängig davon, ob man nun bestimmte Aussagen mit einem Wert verknüpft, oder eine tatsächliche Größe abschätzt, steht und fällt die Aussagekraft des daraus abgeleiteten Resultats immer mit der Korrektheit, bzw. Genauigkeit der Abschätzung. Und anders als der tatsächliche Vergleich mit den anderen Optionen lassen sich diese Schätzungen tatsächlich im Nachhinein überprüfen. Wenn ich die vorhergesagte Fahrzeit von 58 Minuten für die vom Navigationsgerät favorisierte Strecke bestätigen kann, kann ich ohne weiteres davon ausgehen, dass auch die Abschätzung der anderen Routenvorschlägen zutreffend gewesen wäre, und die Entscheidung damit die richtige war. Wäre ich dagegen nach 40 Minuten am Ziel, hätte ich dagegen durchaus Zweifel an der Grundlage der Routenwahl. Die Überprüfung von Schätzwerten und Vorhersagen ist also wichtig und sollte ebenfalls einen festen Platz im Entscheidungsprozess haben.

Für die meisten Bereiche werden die Kriterien dauerhaft sein und sich im Laufe der Zeit nur im Detail verfeinern. Die zu bewertenden Dinge (z.B. Anforderungen und Wünsche) haben ihren Ursprung dagegen in den permanenten Eingangsströmen eines Ticketsystems, so dass sich die Anwendung der Kriterien recht einfach fest in einem Prozess verankern lässt.

Das schnelle Aufstellen und Verfügbarmachen von Kriterien ermöglicht es jedoch auch, Kriterien auch nur im Rahmen einer kurzfristigen Strategieverfolgungen oder eines Projektes aufstellen, und damit z.B. die Wunschdatenbank nach passenden Kandidaten zu durchsuchen.

Umgekehrt kann man auch die vorhandenen Voränge nach Themen gruppieren und die Vorgänge eines Themas anhand von darauf abgestimmten Kriterien nach Nutzwert bewerten, um so in dieser Menge die Vorgänge mit dem größten Effekt zu finden. Ein Beipiel wären hier etwa die Vorgänge, die die Verbesserung der Performance der Anwendung zum Gegenstand haben. Diese lassen sich in einem Ticketsystem leicht per Stichwort kennzeichnen und auflisten. Stellt man die Bewertung dann so auf, dass diese einen numerischen Ergebniswert zurückliefert, kann man diesen dann in ein selbstdefiniertes Feld eintragen und die Liste danach sortieren und entsprechend dieser Reihenfolge abarbeiten.

Natürlich kann man die Felder für die Bewertung von Vorgängen auch direkt im Ticketsystem anlegen, inklusive der Berechnung der numerischen Werte, und man könnte deren Befüllung sogar über Workflows erzwingen, doch dann wäre der Aufwand um ein vielfaches größer, so dass man das nur in einer sehr allgemeingültigen Form und nicht themenbezogen machen würde. Eine Bewertungsvorlage in *FlowProtocol* ist dagegen in sehr kurzer Zeit erstellt und angepasst und kann in gleicher Weise verwendet werden. Durch die Individualisierung auf einzelne Themen, Zielgruppen oder Strategien kann man die Bewertung jedoch sehr präzise ausformulieren und muss nicht Äpfel mit Birnen vergleichen. Im Performance-Beispiel oben könnte zum Beispiel folgender Aspekt abgefragt werden:

> *Wie weitreichend wäre die Wirkung der vorgesehenen Maßnahmen zur Performanceverbesserung innerhalb der Anwendung?*
>
> ◯ *Die Wirkung umfasst einen einzelnen Programmbereich oder eine spezielle Funktion.*
> ◯ *Die Wirkung umfasst zunächst einen einzelnen Programmbereich oder eine spezielle Funktion. Die gefundene Lösung wäre jedoch mit geringerem Aufwand auf weitere Programmbereiche anwendbar.*
> ◯ *Die Wirkung umfasst einen zentralen Programm- oder Funktionsbereich, der oft von vielen Anwendern genutzt wird.*
> ◯ *Die Wirkung umfasst eine Framework-Komponente, die an sehr vielen Stellen innerhalb der Anwendung verwendet wird.*

2.3 Fehlerpriorisierung nach Schweregrad

Um auf die Vorgehensweise bei der Aufstellung von Kriterien genauer einzugehen, schauen wir uns das folgene Beispiel an: Die Priorisierung der Fehlerbehebung soll sich an einem Schweregrad-System orientieren, um die vorhandenen Ressourcen bestmöglich auf die wichtigsten Vorgänge zu konzentrieren. In dieser einfachen Formulierung stecken bereits viele Details, die das Ergebnis erster Vorüberlegungen sind. Die Ausgangslage für

eine solche Optimierung ist am Anfang zunächst nur die Feststellung, dass die bestehende Vorgehensweise in Bezug auf bestimmte Aspekte Schwächen hat.

Im vorliegenden Fall ist es die Erkenntnis, dass es bei der vollständig am Erfassungsdatum orientierten Abarbeitung von Fehlervorgängen zu größeren Verzögerungen bei der Behebung von Engpässen kommen kann, die die Arbeitsfähigkeit des Kundensystems beeinträchtigen.

Mögliche Lösungsansätze für das Problem wären die massive Aufstockung der Ressourcen für die Fehlerbehebung oder die Verstärkung der Maßnahmen zur Reduzierung der Fehler, und beides sollte zumindest ebenfalls in Erwägung gezogen werden. Unabhängig davon würde jedoch wiederholt die Situation auftreten, dass ein als wenig störend empfundenes Problem vor einem als kritisch wahrgenommenen Fehler behoben werden müsste, was nicht im Sinne der Kundenzufriedenheit wäre. Eine Anpassung der Bearbeitungsreihenfolge ist also zwingend notwendig.

Um hierbei zukünftig den Schweregrad der Fehler zu berücksichtigen muss man diesen zunächst formalisieren, etwa in Form der vier Schweregrade „kritisch“(1), „hoch“(2), „moderat“(3) und „niedrig“(4), die man jedem Vorgang vom Typ Fehler (Bug) zuordnen kann.

Ein naheliegender Ansatz wäre nun der, die Vorgänge in erster Linie nach Schweregrad und danach nach Erfassungsdatum abzuarbeiten. Das kann jedoch je nach Fehlermenge problematisch sein, da der schon am längsten erfasste Fehler vom Schweregrad moderat erst dann bearbeitet wird, wenn zu diesem Zeitpunkt kein kritischer oder als hoch eingestufter Fehler mehr vorhanden ist. Insbesondere wird es fraglich sein, ob ein als niedrig eingestufter Fehler jemals bearbeitet wird.

Dieses Problem kann durch das Autobahnprinzip verhindert werden. Hierbei wird die Abarbeitung von Vorgängen der einzelnen Schweregrade so organisiert wie der Verkehrsfluss auf den Spuren einer Autobahn. Auf der linken Spur kommt man am schnellsten voran, auf der mittleren etwas langsamer und auf der rechten noch etwas langsamer, aber auf allen Spuren kommt man vorwärts. Übertragen auf die Schweregarde könnte das so aussehen, dass man kritische Aufgaben tatsächlich umgehend bearbeitet, und die Abarbeitung der restlichen Schweregrade mengenmäßig in einem dem Schweregrad entsprechenden Verhältnis einplant, etwa so:

$$T(\text{hoch}) : T(\text{moderat}) : T(\text{niedrig}) = 7 : 4 : 2$$

Innerhalb jedes Schweregrades kann man nun wieder Reihen bilden, die sich am Erfassungsdatum orientieren, bei denen sich neue Vorgänge also unten einreihen, innerhalb der Reihe mit der dort vorgegebenen Geschwindigkeit nach oben wandern und dann dort abgeschöpft werden.

Die Einplanung von Bug-Vorgängen lässt sich so gut in den Sprint-Rhythmus integrieren, wobei man lediglich die Anzahl der Bug-Vorgänge vorgeben werden muss, die man einplanen möchte. Diese kann sich zum Beispiel an der Gesamtzahl orientieren oder auch über eine Reduzierungsformel berechnet werden. Die Verteilung ergibt sich dann durch Multiplikation mit $\frac{7}{13}$, $\frac{4}{13}$ und $\frac{2}{13}$ und festgelegten Rundungs- und Übertragungsregeln. Bei einer festgelegten Minimalanzahl für die eingeplanten Bug-Aufgaben pro Sprint kann man auf diese Weise sogar Aussagen über den Zeitpunkt der Bearbeitung jeder Aufgabe mit Schweregrad treffen.

Da sich die Zuweisung eines Schweregrades nun unmittelbar auf die Reihenfolge der Umsetzung auswirkt, wird jeder Mitarbeiter, der direkt oder indirekt von einem Fehler be-

troffene ist, ein Interesse daran haben, den größtmöglichen Schweregrad zuzuordnen, um eine zeitnahe Bearbeitung sicherzustellen. Überlässt man diese Zuordnung ohne weitere Regulierung den Erstellern der Vorgänge, erlebt man schnell eine Schweregrad-Inflation, bei der am Ende selbst Tippfehler mit „hoch“eingestuft werden. Noch schlimmer ist es, wenn ein kritischer Fehler nicht als solcher erkannt und nicht sofort bearbeitet wird.

Es ist also zunächst wichtig, festzulegen, unter welchen Umständen ein Fehler objektiv gesehen kritisch ist oder einen entsprechend hohen Schweregrad hat. Am sinnvollsten ist es, die Bestimmung hier absteigend durch hinreichende Kriterien durchzuführen, d.h. man listet zuerst alle möglichen Aussagen auf, deren Zutreffen den Fehler allein schon als kritisch klassifizieren, und ergänzt diese durch die Option „Keine davon.“.

Welche der folgenden Aussagen trifft auf den Fehler zu?

- ◯ *Der Fehler blockiert die Anwendung vollständig. Kein Benutzer kann aktuell arbeiten.*
- ◯ *Der Fehler wirkt sich schädigend auf den Datenbestand aus. Daten gehen verloren oder werden unwiederbringlich in einer nicht vorgesehenen Form verändert.*
- ◯ *Der Fehler führt zu Verfälschungen von Daten, die finanzielle oder juristische Folgen haben können.*
- ◯ *Der Fehler führt zu einer Einschränkung einer explizit zum Datenschutz eingesetzten Funktion.*
- ◯ *Keine davon.*

Wählt man dies letzte Option, so werden im nächsten Schritt explizit die Aussagen aufgelistet, die dem nicht kritschen Fehler zumindest noch einen hohen Schweregrad bestätigen, usw., bis man dem Fehler am Ende nach dem Ausschlussprinzip den Schweregrad „niedrig“ zuordnet, ohne dafür noch explizit Aussagen zu formulieren. Das System könnte auch umgekehrt aufgebaut werden, allerdings dürfte es dann deutlich schwieriger sein, am Anfang alle Aussagen für einen niedrigen Schweregrad aufzulisten.

Mit Hilfe einer solche Bewertung kann prinzipiell jeder die Schweregrad-Einstufung eines Bug-Vorgangs vornehmen und sollte dabei auf dasselbe Ergebnis kommen. Um jedoch auzuschließen, dass eine nicht zutreffende Einstufung vorgenommen wird, weil sich jemand für seinen Vorgang Priorität erschleichen möchte, oder nicht über die notwendige Fachkenntnis verfügt, sollte die Einstufung regelmäßig zentralisiert vorgenommen werden, z.B. wöchentlich durch eine feste Gruppe von Entwicklern. Diese Form der Auseinandersetzung mit der Gesamtheit der neu eingegangenen Fehler hat durchaus auch einen verständisfördernden Effekt, der auch in die Teams hineingetragen wird.

Es ist sinnvoll, die Vorlage zur Bewertung des Schweregrades intern für alle verfügbar zu machen, da es immer wieder Fälle geben wird, wo der Ersteller eines Vorgangs nicht mit der getroffenen Einstufung übereinstimmt, und dann gebeten werden kann, seinerseits über die Vorlage eines Einstufung vorzunehmen. Weicht diese ab, kann im Doalog geklärt werden, welche Kriterien tatsächlich zutreffen und welche Einstufung sich daraus ergibt.

Auch hier gilt es natürlich wieder, die Kriterien bei Bedarf zu erweitern, was technisch gesehen sehr einfach ist, und gerade deshalb nach festgelegten Regeln ablaufen sollte, um Missbrauch und Wildwuchs zu verhindern.

2.4 Rangfolgenbestimmung

Ähnlich wie bei der Fehlerbehebung hat man auch bei der Umsetzung von Ideen und Projekten die Herausforderung, diese in der passenden Reihenfolge umzusetzen. Große Projekte lassen sich meist in kleinere aufteilen und selbst innerhalb einzelner Produktlinien findet man leicht Ausbaustufen, die sich nacheinander umsetzen lassen. Noch viel mehr Flexibilität hat man dann, wenn es um die Erweiterungen eines bestehenden Produktes geht, da hier Entwicklungen aller Größenordnungen und verschiedenen Richtungen aufeinandertreffen.

Da man mit dieser Arbeit Geld verdienen möchte, wird man sich natürlich in erster Linie an den wirtschaftlichen Aspekten orientieren und vorrangig die Projekte angehen, die das beste Verhältnis aus Aufwand und Gewinn versprechen. Bei der Umsetzung bezahlter Individualentwicklungen kann man hier sogar relativ direkt kalkulieren. Etwas schwieriger wird es, wenn es um die Erweiterung einer größeren Standardsoftware geht, mit der man Kundenbindung und Absatz erhöhen möchte. Der reine Entwicklungsaufwand ist zwar auch in diesem Fall noch relativ einfach kalkulierbar, dieser steht jedoch nicht allein auf der Aufwand-Seite. Jede weitere Funktion innerhalb der Standardsoftware und jede zusätzliche technische Abhängigkeit von einer Komponente oder einem Dienst bedeuten permanenten Aufwand in Form von Betreuung und Überwachung, sei es nun auf Seiten von Qualitätssicherung, Support, Projektmanagment oder innerhalb der Entwicklung. Damit verbunden sind Aufwände für die Schulung der Mitarbeiter und das Risiko von Störungen und Ausfällen. Es ist also ratsam, Aspekte wie diese bei der Betrachung von Projektkandidaten mit zu berücksichtigen:

> *Entsteht durch die Entwicklung eine neue Abhängigkeit von einer Komponente, Systemtechnologie oder einem externen Dienst?*
>
> ◯ *Ja*
> ◯ *Nein*

Im Ja-Fall sollte diese Abhängigkeit für jeden der drei Fälle präzisiert und bezüglich der permanenten Aufwände und Risiken bewertet werden, z.B. so:

> *Welche Situation besteht in Bezug auf die Versionsabhängigkeit bei der Anbindung des externen Dienstes?*
>
> ◯ *Die Schnittstelle für die Anbindung des Dienstes ist vollständig festgelegt und wird nur im Rahmen von Erweiterungsprojekten geändert.*
> ◯ *Die Schnittstelle für die Anbindung des Dienstes kann geändert werden, bleibt jedoch abwärtskompatibe, so dass die bestehende Programmfunktionalität nicht beeinträchtigt wird. Erweiterungen können geplant werden, müssen aber verschiedene Versionen unterscheiden.*
> ◯ *Die Schnittstelle für die Anbindung des Dienstes wird sich evtl. nicht abwärtskompatibel ändern, so dass es zur Laufzeit zu Einschränkungen von Programmfunktionalität kommen kann. Anpassungen müssen kurzfristig eingeplant werden und verschiedene Versionen unterscheiden.*

Die Gewinn-Seite wird ebenfalls schwerer greifbar. Der durch mehr Nutzwert und die höhere Produktattraktivität gesteigerte Umsatz und Auswirkungen auf die Kundenbindung

lassen sich meist nur mit den aufwändigen Mitteln der Marktanalyse grob vorhersagen und noch schwieriger verifizieren.

Auch wenn sich wie im Beispiel oben vieles davon in Form von Kriterien abbilden lässt, sind die Zusammenhänge dahinter oft so komplex, dass man nur sehr schwierig eine Bewertung auf Basis einfach und objektiv zu beantwortender Fragen hinbekommen wird. Und selbst wenn, ist immer noch fraglich, wie man die verschiedenen Kriterien gegeneinander abwägt. Für den einen überwiegt die Chance, der andere fürchtet mehr das Risiko, ein dritter scheut den Aufwand. Dass diese Kriterien meist auch noch verschiedene Abteilungen betreffen, macht eine einheitliche Bewertung zusätzlich schwierig. Während man Aufwände noch mit viel Mühe in einmalige und laufende Kosten abschätzen kann, muss man bei der Einbeziehung von Chancen und Risiken die Eintrittswahrscheinlichkeit abschätzen, was von der Vorhersagekraft her auch mit der Befragung einer Glaskugel gleichgesetzt werden kann, jedoch deutlich Aufwändiger ist.

Es ist also eher zu empfehlen, allgemeinerer Fragen der Form „Wie groß ist der Aufwand?“ zunächst soweit wie möglich in überschaubarere Teilaspekte herunterzubrechen, etwa so:

> *Wie groß ist der Schulungsaufwand für einen Mitarbeiter im technischen Support in Bezug auf die vorgesehene Erweiterung?*
>
> ◯ *Gering. Die Bedienung wird weitgehend intuitiv sein und sich an vorhandenen Mustern orientieren. Vorhandenen Kenntnisse werden für die Betreuung ausreichend sein.*
> ◯ *Mittel. Eine Einführung in die Bedienabläufe wird notwendig sein, es gibt jedoch keine neuen technischen Konzepte.*
> ◯ *Hoch. Die Entwicklung bringt eine Erweiterung der technischen Infrastruktur, neue Fehlerbilder oder Technologiekonzepte mit sich, die eine umfangreichere Schulung notwendig machen.*

Diese lassen sich anhand der Beschreibung der Zusammenhänge oder nach Bauchgefühl leichter abschätzen als bei groben Zusammenfassungen.

Das Ergebnis solcher Einschätzungen muss nicht notwendigerweise in eine Bewertungsmetrik einfließen, sondern kann auch dazu verwendet werden, die positiven und negativen Aspekte eines Projektes zum besseren Verständnis als Text aufzulisten, evtl. mit der eindrücklichen Kennzeichnung „$++$“, „$+$“, „$-$“, „$--$“, jedoch ohne Aufsummierung. Diese erweiterte Liste kann nun Ausgangspunkt für eine Rangfolgenbestimmung sein, die von einer Person oder Gruppe ohne weitere Einbeziehung von Kriterien vorgenommen wird.

Wenn es darum geht, Dinge in eine Reihenfolge zu bringen, gibt es eine Technik, die sich relativ einfach an einer Tafel durchführen lässt. Hierbei werden die zu priorisierenden Elemente zunächst als ungeordnete Liste untereinandergeschrieben, und am rechten Rand mit Buchstaben $A, B, C, \ldots$ durchnummeriert. Anschließend wird rechts noch ein schräglaufendes Gittermuster ergänzt, sodass für jedes Paar von Listeneinträgen ein Gitterkästchen vorhanden ist. Nun geht man alle Gitterkästchen und damit alle Paarungen aus der Liste durch und bestimmt für jede Paarung das bessere oder wichtigere Element, und trägt den entsprechenden Buchstaben dann in das Kästchen ein. Am Ende zählt man das Vorkommen jedes Buchstabens im Gittermuster und sortiert die Einträge absteigend nach der Häufigkeit.

Die Menge der Elemente wird hier also auf Zweierpaarungen heruntergebrochen, was es in der Tat einfacher macht, schnell und intuitiv eine Antwort zu finden, insbesondere wenn man zuvor die Vor- und Nachteile herausgearbeitet hat. Unterstützen kann man diesen Prozess zusätzlich noch dadurch, dass man die Fragestellung geeignet wählt:

> *Wenn nur die folgenden beiden Projekte zur Auswahl stünden und wir ab morgen genau eines davon umsetzen könnten, welches sollten wir wählen?*
>
> ◯ *Projekt X*
> ◯ *Projekt Y*

Die Anwendung dieser Methode ist dann sinnvoll, wenn man sich schwer tut, Kriterien aufzustellen oder gegeneinander abzuwägen oder bewusst eine schnelle intuitive Einschätzung von jemanden erhalten möchte. Bei der Durchführung der Methode an der Tafel wird man vermutlich die einzelnen Paarentscheidungen per Handzeichen mit einer ganzen Gruppe von Leuten machen und daraus die Rangfolge ableiten. Alternativ kann man das Schema auch von jedem einzeln auf Papier ausfüllen lassen und am Ende die Häufigkeiten der Buchstaben über alle Teilnehmer hinweg addieren.

Die oben beschriebene Methode ist auch fest in *FlowProtocol* eingebaut und kann so sehr einfach in digitaler Form angeboten werden. Durch die Möglichkeit, sich das Ergebnis mittels mailto-Link zuschicken zu lassen, ist insbesondere der Teilnahmeaufwand recht gering und kann ohne Weiteres investiert werden. Zu beachten ist allerdings, dass die Anzahl der Paare einer Liste der Länge n mit $\frac{1}{2}n(n-1)$ relativ schnell anwächst. Bei einer Liste mit 10 Elementen muss man demensprechend über 45 Paarungen entscheiden, was durchaus einige Minuten in Anspruch nehmen kann.

Alles in allem hat man viele Möglichkeiten und Methoden, um eine sinnvolle Rangfologe festzulegen, die sich noch dazu miteinander kombinieren lassen. Es lohnt sich auf jeden Fall, Zeit in den Aufbau der dazu nötigen Hilfsmittel zu investieren, nicht zuletzt deshalb, weil man durch diese Auseinandersetzung auch viel über die wirtschaftlichen Faktoren der verschiedenen Arbeitsabläufe lernen kann.

2.5 Risikobewertung

Ein weiteres Beispiel für eine Form der Bewertung kommt aus dem Qualitätsmanagement und dreht sich um das Risiko, das eine Veränderung der Software auf das Produkt und den davon abhängigen Produktivbetrieb hat. Grundsätzlich ist man bei der Softwareentwicklung angehalten, das Risiko für Fehler und Störungen einzuschätzen und durch geeignete Maßnahmen zu minimieren.

Die große Schwierigkeit liegt in diesem Fall darin, dass man Gefahren abschätzen soll, die ja schon im Entwicklungsprozess grundsätzlich vermieden werden sollten, und die man daher als sorgfältiger Entwickler ja zuallererst verneinen muss. Zumindest wird man nicht adhoc auf eine Stelle der gerade abgeschlossenen Entwicklung zeigen können und sagen „Genau hier ist zu befürchten, dass das System einen schlimmen Schaden anrichtet.“.

Dass Fehler trotzdem passieren, ist leider eine Tatsache, und schon schon in den neunziger Jahren des letzten Jahunderts wurden in Softwaretechnik-Vorlesungen Formeln dafür gegeben, wie viele Fehler im Durchschnitt in 100 Zeilen Code enthalten sind. Nach Abschluss einer Entwicklung hat man als Programmierer vermutlich ein bestimmtes Bauchgefühl, was die Korrektheit angeht, das jedoch meist sehr unspezifisch ist und keinen brauchbaren Ansatz für gezielte Vermeidungsmaßnahmen liefern kann.

Um wirklich zielgerichtet Risiken ausschließen oder bestmöglich minimieren zu können, muss man die Entwicklung möglichst spezifisch auf potentielle Schwachstellen prüfen. Idealerweise erfolgt so eine Prüfung durch oder mit Unterstützung eines unabhängigen Dritten, da man als Entwickler bei dieser Aufgabe selbst zumindest unbewusst beeinflusst wäre, und es darüberhinaus fraglich ist, ob man bei der Prüfung wirklich die Fehler findet, die man schon bei der Entwicklung übersehen hat. Viele kennen diesen Effekt von der Suche nach Tippfehlern in einem gerade geschriebenen Brief,

Eine gute Praktik ist es, Entwicklungen mit einem Peer-Review abzuschließen, bei dem ein Kollegen die Arbeit durchschaut, kommentiert und gegebenenfalls zusätzliche Maßnahmen zur Sicherstellung der Qualitätsstandards einfordert. Klar ist jedoch auch, dass ein Entwickler die Arbeit von sagen wir zwei Tagen nicht in einer halben Stunden durch Sichtung des Programmcodes auf Einhaltung der Cleancode-Prinzipien, Robustheit und mögliche Schwachstellen durchsehen kann, insofern ist er darauf angewiesen, sich die Entwicklung und das dahinter liegende Konzept durch den Ersteller präsentieren und erklären zu lassen. Nur so und im Dialog mit dem Ersteller lässt sich in kurzer Zeit ein Überblick über die Zusammenhänge gewinnen. Dieser kann dann ausreichen, um vergessene Fälle, kritische Abhängigkeiten und sonstige Schwächen zu erkennen und diesbezüglich nachzufragen.

Die Smash-it-Praktik geht sogar soweit, dass gezielt versucht wird, die Entwicklung mit Hilfe der Kenntnisse über die Implementierung zu einem nicht vorgesehenen Verhalten zu provozieren. Inwieweit der Entwickler dabei wirklich bereit ist, die gegen ihn verwendeten Implementierungsdetails offenzulegen, muss man ausprobieren.

In jedem Fall wird ein guter Prüfer viele Fragen stellen, um das Vorhandensein von Konstellationen festzustellen, die spezielle Gefahren mit sich bringen, und für die sich auch gezielte Maßnahmen ergreifen lassen. Diese lassen sich im Laufe der Zeit sammeln und am Ende einer Entwicklung routinemäßig über eine *FlowProtocol*-Vorlage abfragen. Die Auslöser sind meist sehr trivial und können vom Entwickler schnell als gegeben oder nicht gegeben bestätigt werden, z.B.:

> *Ist von der Entwicklung eine Funktion betroffen, die Datensätze aus dem Datenbestand löscht?*
>
> ◯ *Ja.*
> ◯ *Nein.*

In Abhänigkeit eines Auslösers können nun verschiedene Risiken benannt und die Durchführung geeigneter Abwehmaßnahmen abgefragt werden:

> *Wurde sichergestellt, dass die Löschung der Datensätze immer auf die zur Löschung vorgesehenen Datensätze beschränkt bleibt?*
>
> ◯ *Ja, es wurden spezielle Abgrenzungstests formuliert, mit denen die Beschränkung der Löschung sichergestellt wird.*
> ◯ *Ja, der Code der Löschroutine wurde von einem Kollegen speziell auf diese Gefahr hin gesichtet und abgenommen.*
> ◯ *Nein, noch nicht. Diese Maßnahme steht noch aus.*

Das Ergebnis einer solchen Ausführung erzeugt ein Protokoll und ggf. eine To-do-Liste der noch ausstehenden Maßnahmen, die der Entwickler dokumentieren, bzw. abarbeiten kann. Die Vorgabe, das Ergebnis einer so durchgeführten Risikoanalyse revisionssicher in dem mit der Entwicklung verknüpften Vorgang zu hinterlegen, gibt dem Vorgehen

schließlich die notwenige Verbindlichkeit, die auch durch die ISO-Normen eingefordert wird. Wenn es dann entgegen aller Maßnahmen doch zu einem Fall kommt, der durch die Maßnahmen der Risikovermeidung hätte verhindert werden sollen, kann man auf dieser Grundlage sehr einfach prüfen, ob die Vorlage erweitert, oder der Umgang damit besser geschult werden muss. Nach wie vor sind die eigenen Fehler die wichtigste Ressource zur stetigen Verbesserung, und aus jedem Fehler, den man nicht verhindern kann, sollte man zumindest so viel lernen, um ihn nicht ein weiteres Mal zu machen.

3 Unterstützung des Product Owners

Der Product Owner ist eine Rolle innerhalb des Scrum-Prozesses. Er vertritt gleichermaßen Kunden und Anwender innerhalb des Entwicklungsprozesses, formuliert die Userstories, gibt dort Anforderungen und Akzeptanzkriterien vor und nimmt die abgeschlossenen Entwicklungen ab. Er ist erster und wichtigster Ansprechpartner der Entwickler für Rückfragen im laufenden Sprint, wird aber meist abteilungsübergreifend für Rück- und Anfragen rund um das ganze Produkt in Beschlag genommen, und darf dieses darüberhinaus mancherorts auch noch technisch und strategisch weiterentwickeln. In gleicher Weise ist er auch für alles verantwortlich und natürlich Schuld an jedem Versäumnis und jeder Verzögerung. Kurz gesagt, der Product Owner ist jemand, der jedes Werkzeug dankbar annimmt, dass ihn bei seinen vielen Aufgaben unterstützt.

3.1 Die Liebe zu Mustern

Wie schon beschrieben, besteht die Hauptaufgabe des Product Owners darin, Userstories zu formulieren und diese bei der Umsetzung zu begleiten. Über Form und Detaillierungsgrad, die eine solche Story im Idealfall haben soll, gibt es zahreiche Abhandlungen und Aussagen, die vermutlich die komplette Bandbreite des Möglichen abdecken. Sehr bekannt ist die minimale Variante in Form eines Einzeilers, der sich am festen Satzmuster „Als *R* möchte ich *F*, um *Z*.“orientiert, um die Rolle *R*, die gewünschte Funktion *F* und das Ziel *Z* in Bezug zueinander zu bringen. Eine solche Story wird bewusst meist nur als platzhaltendes Artefakt dafür gesehen, dass man sich über die essentiellen Anforderungen und Rahmenbedingungen noch gründlich austauschen muss, was in so einem Fall dann doch sehr gerne auf Zuruf und mündlich geschieht, und nicht für jede Team-Konstellation ideal ist. Am anderen Ende steht die Ansicht, alle Aspekte der Entwicklung vollständig zu verschriftlichen und nichts implizit zu belassen. Traditionell bleibt zumindest die technische Form der Umsetzung weitestgehend dem Entwickler-Team überlassen, allerdings entstehen hieraus nicht selten Konsequenzen für Nutzwert und Handhabung, die dann auf jeden Fall wieder vom Product Owner berücksichtigt werden sollten. So oder so, wird sich ein Scrum-Team mit der Zeit auf eine Form, und den damit verbundene Grad an Detailliertheit und Explizitheit einigen, und damit gut zurechtkommen.

Softwareentwickler lieben Muster, die sich wiederverwenden lassen. Dies gilt selbst dann, wenn nur das Prinzip einer schon bestehenden Entwicklung wiederverwendet wird, und trotzdem jede Zeile neu geschrieben werden muss. In der Regel wird man seine Komponenten jedoch so aufbauen, dass sie spätestens bei Umsetzung der zweiten Instanz in einem maximalem Grad wiederverwendbar sind. Diese Vorgehensweise bringt nicht nur Vorteile hinsichtlich des geringen Entwicklungs- und Testaufwandes, sondern sichert gleichzeitig auch die Einheitlichkeit in der Benutzerführung. Eine große Anwendung

kann an mehr als 100 Stellen Daten in Form einer Liste anzeigen, und wenn diese Listen sich alle gleich in Bezug auf Sortierung und Gruppierung verhalten, kann der Anwender sein Wissen darüber an allen diesen Stellen nutzbringend verwenden, auch für zukünftige. Umgekehrt kann eine neue allgemeine Listenfunktion nachträglich ohne entwicklungstechinschen Mehraufwand für alle bestehenden Listen verfügbar gemacht werden. Die Anwendung wirkt optisch einheitlich und lässt sich nach kurzer Einarbeitung weitestgehend intuitiv bedienen.

Oberflächendesigner nutzen diesen Umstand bei ihrer Arbeit schon seit längerem und erstellen die ersten Entwürfe neuer Programmbereiche gerne unter Verwendung von Typ-Platzhaltern. Diese stellen meist skizzenhaft ein Element eines bestimmten Typs, z.B. eine Auswahlliste dar und zeigen, dass an einer bestimmten Stelle im Entwurf eine solche Liste eingebaut werden soll. Die Details des Elements, hier z.B. Umfang und Benennung der Spalten werden in dieser frühen Phase gerne weggelassen, da sich diese später weitestgehend aus dem Kontext ergeben und für eine Beurteilung des Entwurfs nicht benötigt werden. Die bei dieser Arbeit oft zum Einsatz komenden Mockup-Werkzeuge enthalten für häufigsten Oberflächen-Komponenten schon fest eingebaute Typ-Platzhalter. Je nach Spezialisierungsgrad der eigenen Komponentenauswahl und Umfang solcher Entwurfsphasen kann es sich aber auch lohnen, derartige Elemente auf Magnetfolie für die Arbeit am Whiteboard anzubringen.

3.2 Formulierungshilfen für Userstories

Wenn schon Entwickler und Anwender so viele Vorteile von Modularität und Komponentenbauweise haben, warum soll nicht auch der Product Owner davon profitieren? In Abschnitt 1.5 haben wir schon die Möglichkeit beschrieben, wiederkehrende Musterentwicklungen anhand von Vorlagen konfigurierbar zu machen, um so die bestehenden Anforderungen unmittelbar auf die Möglichkeiten der Komponenten abzustimmen. Der eigentlich maßgebende Vorteil bei dieser Vorgehensweise ist jedoch der, dass man dabei systematisch alle relevanten Fragen im Vorfeld der Entwicklung klären, und mit geringem Aufwand in der Story festhalten kann.

Nehmen wir als Beispiel wieder die Entwicklung einer Listenansicht. Dann ist folgende Frage eine von vielen, die für die Userstory geklärt werden muss:

> *Soll die Liste zur Auswahl der darin angezeigten Elemente verwendet werden können?*
>
> ◯ *Ja, es soll eine beliebige Auswahl möglich sein.*
> ◯ *Ja, die Auswahl soll allerdings auf eine Einfachauswahl beschränkt sein.*
> ◯ *Nein.*

Wie man leicht erkennt, lässt sich die Konsequenz aus jeder der drei Antworten vollständig als Text formulieren, ohne dass dieser nachbearbeitet werden müsste. Der Product Owner hat damit nicht nur einen Assistenten, der ihn an diesen Aspekt der Listenentwicklung erinnert, sondern auch gleichzeitig eine Schreibkraft, die aus dem einen Mausklick für die Antwort einen ausformulierten Aufzählungspunkt in der Userstory erstellt, und damit das hohe Maß an Explizitheit schafft, mit dem sich unnötige Rückfragen vermeiden lassen.

Tatsächlich gibt es bei jeder Art von Entwicklung sehr viele kleine Details, die im Idealfall alle explizit festgelegt werden sollten, wie die Einsortierung von Menüpunkten oder die Zuordnungen zu Lizenzen. Hinzukommen zahlreiche Abhängigkeiten, die bei bestimmten Erweiterungen berücksichtigt werden müssen, etwa die Anpassung verschiedener Datenquellen und Ansichten, wenn eine Entität um eine Eigenschaft erweitert wird. In der Praxis ist man schlecht beraten, diese Dinge in ihrer vollständig dem Entwicklungs-Team zu überantworten, denn selbst wenn man sie gut dokumentiert und dafür Regelwerke schafft, werden immer wieder Teile davon vergessen werden. Außerdem wird es auch Punkte geben, die bewusst durch den Product Owner entschieden werden sollten, und die sich nicht mit einem guten Ergebnis über Automatismen festlegen lassen. Die Arbeit mit einem Vorgang ist auch deutlich einfacher, wenn dort alle zum Gegenstand der Aufgabe gehörenden Punkte explizit aufgelistet sind, das beginnt schon bei der Schätzung für den Sprint und zieht sich durch bis zur Aufstellung der Testpunkte für die Integrationstests.

Die Methode, für jedes wiederkehrende Entwicklungsmuster eine *FlowProtocol*-Vorlage zu erstellen und dort alle Entscheidungsfragen und Abhängigkeiten abzubilden, zaht sich sehr bald aus. Für die Entwicklungen außerhalb dieser Muster kann man ergänzend dazu noch eine Vorlage anlegen, in der man bewusst nach den Auslösern wichtigter Abhängigkeiten fragt, und so sicherstellt, dass diese auch dort berücksichtigt werden.

> *Umfasst die Aufgabe den Einbau einer Referenzbeziehung auf die Personen-Entität?*
>
> ◯ *Ja.*
> ◯ *Nein.*

Die Menge an Abhängigkeiten und Einstellungen die man auf diese Weise allein in den ersten Wochen zusammenträgt, wird verblüffen und die daraus erzeugten expliziten Auflistungen in den Userstories werden entsprechend umfangreich ausfallen, und auch Einfluss auf den Schätzwert der Stories haben. Hier braucht man vertrauen, dass das Team im Laufe der Zeit lernt, damit umzugehen und die große Menge an Informationen nicht mit Aufwand oder Komplexität, sondern Klarheit und Explizitheit assoziiert. Dies gelingt besonders dann, wenn man das Team von Anfang an bei der Aufstellung dieser Mechanismen mitnimmt und evtl. auch Formulierungen mitgestalten lässt. Schließlich hat das Team als Verarbeiter der Information und Ersteller der Komponente die beste Einschätzung darüber, welche Informationen in welcher Form für die Arbeit benötigt werden, womit der Bogenzu Abschnitt **??** geschlossen wäre.

3.3 Der Inhalt von Userstories

Was gehört überhaupt alles in eine Userstory? In Abschnitt 3.1 haben wir dieses Thema schon gestreift und sind dabei hauptsächlich auf Detaillierungsgrad und Explizitheit eingegangen. Grundsätzlich sollte eine gute Userstory aber eine Vielzahl von Fragen beantworten, bei denen es insgesamt darum geht, wer der Anwender ist, und welche Aufgabe er in welcher Weise mit der vorgesehenen Entwicklung effektiv erledigen kann. Die Idee der Userstory ist tatsächlich, das ganze als möglichst plastische und gut vorstellbare Geschichte zu formulieren, damit sich alle Beteiligten eine nicht nur lebendige, sondern auch motivierende und gemeinsame Vorstellung davon machen können. Wollte man die Vollständigkeit einer Userstory auf den Prüfstand stellen, könnte man schauen, ob wirklich alle der folgenden Fragen geklärt sind:

Wähle falls zutreffend die erste der folgenden Fragen aus, die durch die Usertory nicht ausreichend geklärt ist?

- ◯ *Ist die Entwicklung Teil eines größeren Projektes und das Projekt Teil einer Strategie?*
- ◯ *Welche Entwicklung soll durchgeführt werden?*
- ◯ *Wer wird diese Entwicklung als Anwender verwenden und in welcher Rolle?*
- ◯ *Welches Ziel verfolgt der Anwender dabei? Was ist seine Aufgabe?*
- ◯ *Wie löst er diese Aufgabe jetzt?*
- ◯ *Wie löst er diese Aufgabe mit der Entwicklung?*
- ◯ *Welchen benennbaren Nutzwert, bzw. welche Zeitersparnis hat er durch die Entwicklung?*
- ◯ *In welchem Umfang (wie häufig und mit welchen Datenmengen) wird die Entwicklung dabei eingesetzt?*
- ◯ *Sind die alle klar messbaren Akzeptanzkriterien explizit genannt?*
- ◯ *Wie ist die Tätigkeit im Zusammenhang mit der Entwicklung in den Arbeitsablauf des Anwenders eingebettet? Wie werden Daten und Informationen dabei ausgetauscht?*
- ◯ *Alle Fragen sind hinreichend geklärt!*

So wichtig Plastizität und Anwendernähe sind, so liefert dieser Teil der Userstory nur eine hauptsächlich äußerliche Beschreibung der Entwicklung, die je nach Ausführlichkeit der Beschreibung auch recht grob sein kann. Technische Aspekte und Fragen rund um die Software- Architektur sind nicht vorgesehen. Selbst die Abgrenzung der dafür neu zu entwickelnden Dinge gegen die schon vorhandenen ist im Normalfall nicht Bestandteil der eigentlichen Story. Die Festlegung der zu tätigenden Entwicklungsmaßnahmen anhand dieser Informationen ist trotzdem möglich, und erfolgt in diesem Fall durch das Entwickler-Team, ggf. unterstützt durch die UX-Kollegen.

Der Scrum-Prozess dient im Kern dazu, die Komplexität von Entwicklungen auf beherrschbare Schritte herunterzubrechen. Die Formulierung der Userstories durch den Product Owner soll sich ausschließlich darum kümmern, das Was der Entwicklung von „unklar“ in Richtung „klar“ zu verschieben, ohne sich dabei um das Wie zu kümmern. Für die Userstories im Sprint-Backlog bricht das Entwicklungsteam dannn in der sogenannten Taskplanung die Stories auf einzelne Tasks herunter, und sorgt damit auf für das Wie für die gleiche Klarheit. Ein Task ist dabei eine Unteraufgabe innerhalb einer Story, die sich dadurch auszeichnet, dass sie klar beschreibt, was für den Entwickler zu tun ist.

Die Erstellung von Tasks für eine Userstory kann je nach Formulierung der Story ein aufwändiger Prozess sein, in dessen Verlauf es auch verschiedene Umsetzungsvorschläge und damit verbunden Diskussionen und Entscheidungsprozesse geben kann. Insbesondere wenn die Wahl auf technischer Ebene wieder Auswirkungen auf Aufwand und Benutzerführung hat, kommt wieder der Product Owner ins Spiel und muss entscheiden. Solche zusätzlichen Iterationen möchte man natürlich weitestgehen vermeiden, erst recht, wenn der Sprint schon läuft. In der ausführlichen Scrum-Variante mit Preview-Meeting und Planning-Meeting kannn das Team die Taskplanung zwischen diese Meetings legen und so zum einen für eine präzisere Schätzung nutzen, und zum anderen dafür, Rückfragen aus der Taskplanung noch vor dem Planning mit dem Product Owner zu klären. Unabhängig davon ist ein hohes Maß an Explizitheit in der Story die beste Möglichkeit, um die Taskplanung einfach zu halten. Die Voraussetzung dafür ist natürlich, dass der Product Owner

diese Explizitheit auch leisten kann, was entweder technisches Verständnis voraussetzt, oder die Verwendung von bewährten Mustern, für die explizite Entwicklungsvorgaben möglich sind.

Selbst wenn sich der Product Owner schwer damit tut, sollte er es probieren. Ist sein Umsetzungsvorsclag nicht umsetzbar oder bestehen dazu bessere Alternativen, wird ihn das Entwickler-Team darauf hinweisen. Die Anpassung des Vorschlags wird dann wahrscheinlich weniger aufwändiger sein, als die Erstellung eines eigenen Vorschlags durch das Team auf dem weißen Blatt Papier.

Hat man diese Voraussetzungen, bietet sich eine Dreiteilung innerhalb der Story. In einem Abschnitt „Hintergrund“ beschreibt man, warum die Story überhaupt existiert, zu welchem Projekt und welcher Strategie sie gehört und welches Problem damit gelöst werden soll. Der Abschnitt „Gegenstand der Aufgabe“ beschreibt, was zu tun ist, was am Ende vorhanden sein soll, damit die Story als fertig gekennzeichnet werden kann. Der dritte Teil ist die plastische Beschreibung der Nutzung der Entwicklung durch den Anwender und erfolgt ausschließlich mündlich im Preview-Meeting. Für die Formulierung von Hintergrund und Gegenstand sind Aufzählungslisten besser als Fließtext, da man auf diese Weise die einzelnen Informationen und zu entwicklenden Dinge klar voneinander abgrenzen und auch beim lesen Stück für Stück gedanklich verarbeiten kann. Bei Fließtext fällt diese Trennung deutlich schwieriger.

3.4 Abnahme von Userstories

Die Beziehung zwischen Product Owner und den Entwicklern ist wichtig und hat einen großen Einfluss auf das Ergebnis der Sprints. Mit der Zeit wird ein Entwickler den Product Owner blind verstehen und für vielen Aspekten sogar in der Lages sein, diese aus Sicht des Product Owners einzuschätzen oder zu entscheiden, selbst wenn diese Einschätzung nicht die eigene ist. Diese Fähigkeit sollte auf jeden Fall angestrebt werden, da sie die Eigenständigkeit des Teams erhöht und die Kommunikation vereinfacht. Voraussetzung dasfür ist, dass selbst kleine Entscheidungen gegenüber dem Team begründet werden, oder dass man in machen Fällen auch die verworfenene Alternativen erwähnt und warum man sich dagegen entschieden hat. Man sollte offen Kommunizieren, welche Aspekte für einen selbst welches Gewicht haben, und insbesondere schwierige Entscheidungen auch als solche darstellen.

Vieles davon kann bei der Vorstellung von Userstories passieren und ist dort gut aufgehoben, da man mehrere Zuhöhrer hat. Die Abnahme einer Story findet dagegen oft als Einzelgespräch statt, und bietet ebenfalls einen Rahmen, um soche Dinge auszutauschen. Hier hat man den Vorteil, dass man ungestört ist, offener reden kann, und mit der abgeschlossenen Story auch ein Arbeitsergebnis hat, für das man Lob oder Kritik äußern kann.

Rund um die Abnahme gibt es viele Standardfragen, die mal als Product Owner für jede Story stellen kann, etwa:

> *Wie hoch schätzt du den Grad an inhaltlicher Komplexität für diese Entwicklung ein?*
>
> ◯ *Unterdurchschnittlich.*
> ◯ *Etwa durchschnittlich.*
> ◯ *Überdurchschnittlich.*

Tatsächlich kann man viele Fragen der Form „Wie ging es dir bei dieser Entwicklung und welchen Eindruck hast du vom Ergebnis?“ sehr gut in eine Vorlage packen und im Vorfeld der Abnahme beantworten lassen. Man kann dabei zusätzlich für bestimmte Antworten Minutenwerte addieren und so automatisch eine Schätzung für den Zeitbedarf des Abnahmegespräches berechnen, ergänzt durch eine ebenfalls automatisch auf Basis der Antworten zusammengestellten Agenda. Das Auslagern solcher Fragen in eine Vorlage hat hier zusätzlich den Vorteil, dass der Entwickler beim Beantworten der Fragen keinen Zeitdruck hat und gegebenenfalls sogar zuerst durch geeignete Maßnahmen dafür sorgen kann, dass die gewünschte Antwort auch zutreffend ist. Die Fragen sollen ja schließlich keine Prüfungssituation erzeugen, sondern Gelegenheit zur Reflektion über die eigene Arbeit und die Einhaltung der damit verbundenen Standards geben. Gerade bei Aspekten aus den Bereichen Qualitätssicherung und Präsentationskompetenz kann man so seine Anforderungen gut vermitteln:

> *In welcher Form kannst du die Story im Review-Meeting vor dem Kollegium präsentieren?*
>
> ◯ *Vorführung der Funktionalität als Live-Demo mit Einbettung in eine Story samt Personas und einer anwendungsnahen Aufgabenstellung und entsprechenden Beispieldaten.*
> ◯ *Vorführung der Funktionalität als Live-Demo anhand von Dummy-Beispieldaten mit mündlicher Ergänzung der realen Anwendungssituation.*
> ◯ *Vorführung der Funktionalität als Live-Demo anhand von Dummy-Beispieldaten.*
> ◯ *Mündliche Beschreibung der Funktionalität ohne Live-Demo.*
> ◯ *Hier benötige ich noch Unterstützung.*

3.5 Sonstige Reflektionsfragen

Generell sollten Teamleiter und Product Owner immer im engen Austausch mit den Entwicklern stehen und sensibel auf Störungen oder Bedenken achten, die Einfluss auf die Umsetzung von Stories haben könnten. Diese werden nicht immer proaktiv geäußert, zum Teil, weil man davon ausgeht, dass diese Dinge sowieso bekannt sind, zum anderen aus Angst, dass darin eine persönliche Schwäche gesehen wird.

Die regelmäßige Durchführung einer Sprint-Retrospektive mit offenem Austausch ist auf jeden Fall schon mal eine Wichtige Maßnahme, um eine Möglichkeit zu geben, solche Dinge anzusprechen. Der Einbau dahingehender Rückfragen in bestimmte Prozessschritte wie z.B. oben beschrieben ist zusätzlich sinnvoll. Das Vorhandensein bestimmter Auswahlmöglichkeiten signalisiert klar, dass diese nicht tabu sind, und damit umgegangen werden kann. Zudem ist die Antwort eng auf einen bestimmten Kontext beschränkt und der Austausch darüber in ein sowieso stattfindendes Zwei-Personen-Gespräch eingebettet.

Geschickterweise formuliert man Fragen und Antworten so, dass nicht der Eindruck entsteht, mit bestimmten Antworten ein schon vorhandenes Konfiktpotential noch weiter anzuheizen, sondern mit dem Ziel, möglichst neutral und konstruktiv auf die Situation einzugehen:

> *Gab es bei der Tasktplanung für eine Story längere Diskussionen über die Form der Umsetzung?*

- ◯ *Nein, es wurde für alle Stories schnell eine von allen als gut empfundene Lösung gefunden.*
- ◯ *Ja, es wurden teilweise verschiedene Ansätze besprochen und am Ende gemeinschaftlich eine Lösung festgelegt.*
- ◯ *Ja, es wurden verschiedene Ansätze besprochen und am Ende zumindest mehrheitlich eine für alle akzptable Lösung festgelegt.*
- ◯ *Ja, es wurden Lösungen gefunden, die die Anforderungen erfüllen, diese werden jedoch teilweise als nicht ideal angesehen.*

Anhand der Antworten wird man schnell erkennen, wo weitere persönlich gestellte Fragen angebracht sind und kann auf etwas Bezug nehmen.

Es kann durchaus auch den Weg gehen, das Team selbst über die Fragen entscheiden zu lassen, zu denen es sich gerne nach Abschluss einer Story oder am Ende des Sprints äußern würde. Die entsprechenden Vorlagen können dann vollständig vom Team verwaltet und gepflegt werden.

4 Dynamische Anleitungen

In gleicher Weise, wie man eine nach einem Muster aufgebauet Entwicklung unterstützt beschreiben kann, kann man auch eine Anleitung erstellen, am der sich der Entwickler Schritt für Schritt bei der Implementierung orientieren kann. Die Erstellung von Anleitungen mit *FlowProtocol* hat gegenüber normalen schriftlichen Anleitungen zwei wesentliche Vorteile: Zum einen können Varianten und Sonderfälle im Vorfeld detailiert abgefragt, und bei der Zusammenstellung des Anleitungstextes berücksichtigt werden, zum anderen können über Eingabefelder auch Begriffe oder Benennungen abgefragt werden, die dann direkt in die erzeugten Codefragemente eingebaut werden, so dass diese im Idealfall 1-zu-1 verwendet werden können. Die Vorgehensweise, die Festlegung der Details einer Entwicklung über diese Mechanismen an den Anfang zu stellen, um anschließend eine passgenaue Anleitung abzuarbeiten, macht die Implementierung einfacher und verringert das Risiko von ungewollten Abweichungen.

4.1 Konfigurator mit Codegenerierung

Sowohl Product Owner, als auch Entwickler kennen die Möglichkeiten ihres Frameworks, allerdings von verschiedenen Seiten. Während der Product Owner die Komponenten fast ausschließlich aus Sicht des Anwenders betrachtet und sich um Funktionsumfang und Benutzerführung kümmert, befasst sich der Entwickler mit deren Entsprechung in der Welt der Klassen und Methoden. Die Implementierung einer Liste, die zur Auswahl von Elementen dient, wird andere Klassen benötigen, als eine vollständig gebundene Liste, die auch Eingaben ermöglicht, und diese über Geschäftslogik-Klassen validiert. Und für jede der beiden Entwicklungen gibt es wieder Varianten und diverse instanzbezogene Eigenschaften wie Überschriften und Objektabhängigkeiten.

Der generischer Aufbau der Frameworkklassen ist die wichtigste Voraussetzung, um eine bestmögliche Kombinierbarkeit der einzelnen Komponenten sicherzustellen. Ein generisch aufgebautes Listen-Steuerelement kann so in gleicher Weise Fahrzeuge oder Personen darstellen, nur dadurch, dass es mit einer entsprechenden Klasse parametrisiert

wird. In gleicher Weise lassen sich durch die Prinzipien der Funktionalen Programmierung auch spezifische Implementierungen wie Berechnungen oder Verhaltensweisen von außen in die allgemeinen Methoden anderer Klassen einbringen. Für das Framework bedeutet diese Trennung von Verantwortlichkeiten mehr Flexibilität und Robustheit und eine einfacherer und klare Vorgehensweise beim Zusammenbau der Komponenten, so wie bei einem Stecksystem.

Trotzdem darf man diese Aufgabe nicht unterschätzen, denn Verbindungen gibt es viele, und an jede davon passen viele Gegenstücke, die nach Funktion und Bedetung richtig ausgewählt werden müssen. Es ist also entscheidend, dass der Entwickler die Anforderungen klar verstanden hat und weiß, mit welcher Konfiguration er diese am besten umsetzen kann.

Was liegt also näher, als die Auswahl der Anforderungen direkt mit den Konfigurationen zu verbinden? Der Product Owner legt fest fest, welches Verhalten er möchte und gibt die Begrifflichkeiten vor, und erzeugt über eine entsprechend aufgebaute Vorlage direkt die passende Anleitung, die dann nur noch über einen Vorgang an das Entwicklungsteam weitergegeben werden muss. Dadurch wird es möglich, einmal erfasste Muster mit minimalem Aufwand beliebig zu reproduzieren und damit den größtmöglichen Nutzen daraus zu ziehen.

Eine solche Durchgängigkeit ist durchaus realisierbar, erfordert aber auch einiges an Abstimmung im Vorfeld. Die entscheidende Herausforderung bei so einem Konfigurator ist sicherlich die, dass technische Details von der Oberfläche und insbesondere der Auswahl fergehalten werden müssen, da sich der Product Owner damit nicht auseinandersetzen sollte. Entsprechend müssen technische Entscheidungen entweder auf die Ebene von Anforderungsentscheidungen gehoben, oder durch generelle Festlegung weggelassen werden.

Sich in dieser Form mit dem eigenen Framework auseinanderzusetzen und für alle Konfigurationsmöglichkeiten eine Entsprechung auf Anwendungsebene zu formulieren kann recht aufschlussreich sein und eventuell auch so manche im Laufe der Zeit entstandene Parallelimplementierung zu Tage fördern. Auf jeden Fall fördert es das Verständnis fü den eigene Code und dessen Nutzwert. Gleichzeitig zeigt sich dabei, ob die Trennung von Abhängigkeiten und der Grad an Wiederverwendbarkeit auch wirklich den Ansprüchen genügen, die für eine weitestgehend automatisisierbare Codegenerierung notwendig sind. So oder so wird man zumindest Erkenntnisse gewinnen, die sich positiv auf die Codequalität auswirken werden.

4.2 Anleitung zur Fehleranalyse

Neben der zuletzt beschriebene Erstellung von Programmerweiterungen, die ja das Kerngeschäft der Softwareentwicklung darstellt, gibt es noch viele andere Tätigkeiten, die ihre feste Position im Entwickleralltag haben, und für die ebenfalls Anleitungen geschaffen werden können. Die Durchführung von Fehler-, und Problemanalysen ist so eine Aufgabe, für deren Lösung meist Erfahrung als ausschlaggebend eingeschätzt wird. Diese zeigt sich hauptsächlich darin, dass man weiß, wohin man schauen muss und die richtigen Fragen stellt.

Ähnlich wie bei Entwicklungsmustern gibt es auch bei Fehler wiederkehrende und übertragbare Muster, die sich aufgrund ihres Effekts und des jeweiligen Kontextes identifizieren, und dadurch schließlich auffinden und beheben lassen. Aber auch bei sehr allge-

meinen Fehlern wie Null-Verweis-Ausnahmen kann hilfreich sein, die Suche nach einer strengen Systematik durchzuführen, insbesondere, wenn auch diese Schritte der Analyse dokumentiert werden. Hat man dann endlich das Problem gefunden, geht es an die Lösung, und im Gegensatz zur Analyse besteht hier nicht nur die Gefahr, unnötig viel Zeit aufzuwenden. Den Ausdruck „verschlimmbessert“ hört man im Bugfixing-Kontext leider viel zu häufig, und ebenfalls viel zu häufig gibt sich ein Entwickler damit zufrieden, dass ein problematischer Effekt durch seine Änderung verschwunden ist, manchmal selbst dann, wenn ihm der Zusammenhang zu seiner Änderung nicht wirklich klar ist. Bei einem über eine Vorlage identifizierten Problem kann es hier auf jeden Fall helfen, wenn eine Musterlösung vorgeschlagen wird, ergänzend durch Hinweise oder Kontrollfragen zur Vermeidung unerünschter Seiteneffekte. Letzteres kann auch unabhängig als Teil eines Qualitätssicherungsprotokolls geschehen:

> *Wurde sichergestellt, dass aufgrund der Änderung kein unerwünschtes abweichendes Verhalten bei der nachfolgenden Datenverarbeitung verursacht wird?*
>
> ◯ *Ja, für alle Fallunterscheidungen wurden relevante Anwendungsfälle zumindest theoretisch vollständig durchgespielt und das Sollverhalten sichergestellt. Zusätzlich wurden Integrationstests formuliert.*
> ◯ *Nein, dies wird noch nachgeholt.*

Insbesondere bei der Entfernung vermeintlich fehlerverursachenden Programmcodes sollte man sorgfältig nachforschen, zu welchem Zweck dieser ursprünglich eingefügt wurde, auch wenn man dazu die Versionsverwaltung behühen muss. Entsprechend sollte man Codezeilen, deren Effekt sich nicht unmittelbar erschließt, gut dokumentieren, am besten mit Verweis auf einen Vorgang. Dies gilt insbesondere für Korrekturmaßnahmen, denn sollte sich die Änderung entgegen aller Sorgfalt doch als Ausgangspunkt eines neuen Fehlers herausstellen, möchte man auf jeden Fall ein Zurück zum ursprünglichen Fehler vermeiden.

Bei der Verwendung von Analyse-Vorlagen sollte man nach jeder erfolgreichen Analyse auf jeden Fall prüfen, ob man die gewonnenen Erkenntnisse in die Vorlage einarbeiten kann. Gleiches gilt für erkannte Fehlerursachen bei der Verwendung von Qualitätssicherungsprotokollen. Dieser Punkt wurde ja bereits ausführlich im Abschnitt 2.5 besprochen.