Institut für Softwaretechnologie

Masterarbeit Nr. 84

Modelle und Programmierparadigmen für Geschäftslogik

Sven Schnaible

Studiengang:	Softwaretechnik M.Sc.
Prüfer/in:	Prof. Dr. Erhard Plödereder
Betreuer/in:	Dipl. Ing. Torsten Görg Dr. Armin Müller Jonathan Streit
Beginn am:	16. März 2016
Beendet am:	15. September 2016
CR-Nummer:	D.1

Kurzfassung

Viele Software-Systeme mit Geschäftslogik sind über Jahrzehnte gewachsen und sind durch stetige Änderungen und Erweiterungen von Codeduplikationen und schlechter Behandlung von Sonderfällen geprägt. In dieser wurden Arbeit gängige Programmierparadigmen anhand relevanter Kriterien bewertet und verglichen. Das Ergebnis ist ein Leitfaden, der Entwicklerteams bei der Wahl des geeigneten Paradigmas für ihr Projekt helfen soll.

Inhaltsverzeichnis

1. Einleitung

Noch immer sind viele Software-Systeme in Betrieb, die in Cobol oder anderen imperativen Programmiersprachen implementiert sind. Laut David Stephenson, dem Manager des Softwareunternehmens *Micro Focus*, werden in den Vereinigten Königreichen noch immer etwa 70–80% der Finanztransaktionen in Cobol-basierten Software-Systemen abgearbeitet[1]. Durch jahrelanges, teilweise unkontrolliertes Wachstum sind diese Software-Systeme heute meist geprägt von naiven Abarbeitungen aller möglichen Ausprägungen und Codeduplikation zur Abbildung von Varianten. Grund dafür sind oft fehlende Möglichkeiten Abstraktionen darzustellen.

Heutzutage setzen viele Softwareunternehmen auf objekt-orientierte Programmiersprachen wie Java, das seit der Erstveröffentlichung vor etwa zwei Jahrzehnten nahezu „boomt". Doch auch Objekt-orientierung löst nicht alle Probleme. Es sind jedoch weitere Alternativen, wie funktionale Programmierung, bekannt, die in gewissen Bereichen der Softwareentwicklung Abhilfe schaffen könnten. Diese werden in dieser Arbeit auf ihre Eignung zur Umsetzung von Geschäftslogik untersucht.

Dazu wurden zwei repräsentative Fallbeispiele gewählt und mit einer funktionalen Programmiersprache implementiert. Die beiden Fallbeispiele sind zu berechnender und entscheidender Logik und stammen aus realen Projekten des Industriepartners itestra GmbH. Zuallererst wurden durch Expertenbefragung und Literaturrecherche die relevanten Kriterien für Geschäftslogik herausgearbeitet. In einer Umfrage wurden dann die verschiedenen Codefragemente auf die identifizierten Kriterien hin verglichen. Das Ergebnis ist ein Leitfaden, das Entwicklerteams bei der Wahl des geeigneten Paradigmas für ihr Projekt unterstützen soll.

Diese Masterarbeit entstand in Kooperation mit der itestra GmbH[2]. Die itestra GmbH ist ein innovatives und international tätiges Unternehmen mit dem Ziel, durch Branchenexpertise und Know-How den Kunden bei der Umsetzung ihrer Geschäftslogik zu beraten. Zum Portfolio gehören die Konzeption und Realisierung von geschäftskritischen Systeme, sowie die Wiederherstellung bestehender Software-Systeme, um somit die Leistungsfähigkeit der IT sicherzustellen.

[1] https://www.theguardian.com/technology/2009/apr/09/cobol-internet-programming
[2] http://www.itestra.de/

Gliederung

In Kapitel 2 werden wichtige Kriterien für Geschäftslogik herausgearbeitet. In Kapitel 3 werden die gängigen Programmierparadigmen vorgestellt. Kapitel 4 stellt Ergebnisse aus der Forschung vor. Kapitel 5 stellt die umgesetzten Fallbeispiele vor. Kapitel 6 gibt einen Leitfaden für die Wahl des geeigneten Paradigmas. In Kapitel 7 werden die Ergebnisse der Arbeit zusammengefasst

2. Geschäftslogik

Geschäftslogik bezeichnet die Umsetzung der projekt-spezifischen Logik in Abgrenzung zu technischen Details. In der Schichtenarchitektur bezeichnet es den Teil oberhalb der Datanhaltungsschicht und unterhalb der Präsentationsschicht.

2.1. Kategorisierung

Nach Gesprächen mit Experten wurden drei Kategorien identifiziert, in die sich Geschäftslogik aufteilen lässt. Diese sind: Berechnungen, Entscheidungen und Validierung und Verifizierung von Daten, wobei die meiste Geschäftslogik in die ersten beiden Kategorien fällt.

Berechnungslogik beschreibt im Groben alle Systeme die Mengen von Zahlen verarbeiten. Typische Fälle sind die Berechnung von Steuern und Versicherungstarifen und generell das Finanzwesen. Als Beispiel ist in Tabelle 2.1 die Berechnungformel der Einkommenssteuer für 2016 dargestellt.

Zwei weitere Beispiele für Berechnungs- und Entscheidungslogik finden sich in Kapitel 5.

2.2. Situation

Durch die Entwicklungsgeschichte der Programmiersprachen sind viele alte, noch heute bestehende Software-Systeme (sog. *Legacy-Systeme*) mit imperativen Programmiersprachen

zu versteuerndes Einkommen (zvE)	Einkommensteuer
bis 8.652 Euro	0 (Grundfreibetrag)
8.653 Euro bis 13.669 Euro	$(993,62 * Y + 1.400) * Y; Y = (zvE - 8.652)/10.000$
13.670 Euro bis 53.665 Euro	$225,4 * Y + 2.397) * Y + 952,48; Y = (zvE - 13.669)/10.000$
53.666 Euro bis 254.446 Euro	$0,42 * zvE - 8.394,14$
ab 254.447 Euro	$0,45 * zvE - 16.027,52$

Tabelle 2.1.: Berechnungsformel für die Einkommenssteuer 2016

implementiert. Durch jahrelanges Wachtsum und stetige Änderungen und Erweiterungen treten dadurch häufig sogenannte *Code-Smells* auf. Darunter:

- Übermäßig viele und unverständliche Literale im Code (Beispiel: if BAUSPARVERTRAG = 2 OR = 3). Grund dafür ist entweder Nachlässigkeit der Entwickler oder Einschränkungen der Programmiersprachen.
- Schlechte Abarbeitung von Sonderfällen
- Codeduplikation: Diese entstehen meistens wenn eine neue Funktionalität (bspw. ein Versicherungstarif) hinzugefügt wird, die sich ähnlich zu einer bisherigen Funktionalität verhält. Alten imperativen Programiersprachen fehlt es meistens an Möglichkeiten die Gemeinsamkeiten zu abstrahieren, weswegen die Funktionalität stattdessen einfach kopiert und angepasst wird.

In den letzten Jahrzehnten setzen viele Firmen deshalb auf objekt-orientierte Programmiersprachen. Doch auch bei diesen können nach unkontrolliertem Wachtum ungewollte Effekte entstehen (siehe Abschnitt 3.2). Die umfassende Litaratur über Programmmuster und Richtlinien für gute Programmierung verdeutlichen, dass es prinzipiell möglich ist guten, objekt-orientierten Code zu schreiben, der Entwickler aber häufig zum Gegenteil verleitet wird [FFBS04][Ker04][Mar08].

2.3. Anforderungen

Wie die Forschung herausgefunden hat, gehen mehr als 90% der Softwarekosten in die Wartung [Erl00][Moa90]. Um also die Gesamtkosten gering zu halten, sollte zuallererst auf Wartbarkeit geachtet werden. Als wichtigste Aktivität in der Wartung wurde das Programmlesen und -verstehen identifiziert [BW08][TOAY13]. Ein weiterer Faktor spielt die Fehleranfälligkeit [JB12]. Umso weniger Fehler auftreten, umso weniger Ressources müssen aufgewandt werden um diese Fehler zu beheben, und umso weniger neue Fehler entstehen bei der Umsetzung von Erweiterungen. Dies ist die oberste Priorität und gilt nicht nur für Geschäftslogik sondern für alle Arten von Software.
Nach Gesprächen mit Experten wurden außerdem folgende Anforderungen identifiziert, die insbesondere für Geschäftslogik gelten:

Wenig Redundanz Wenig Redundanz ist ein zentrales Konzept einiger Paradigmen, wie der objekt-orientierten Programmierung. Sie benötigt Möglichkeiten zur Parametrisierung und Abstraktien, wie beispielsweise Methodenüberladung und Vererbung. Gerade viele Modellierungswerkzeuge des *Model Driven Development (MDD)* fehlt es an Abstraktioinsmechanismen, wodurch spätere Änderungen und Erweiterungen erschwert werden.

Geeignete Implementierung von Sonderfällen Sonderfälle sind gut implementiert, wenn sie nicht die allgemeine Abarbeitung stören. Jede Methode sollte nur auf einer Abstraktionsebene operieren [Mar08]. Dies sollte durch das Programmierparadigma unterstützt werden. Ein anderes Problem haben Ansätze mit domänen-spezifischen Sprachen. Sie müssen entweder umfangreich genug sein um alle Sonderfälle abdecken zu können oder einen Mechanismus besitzen, mit dem auf eine *general purpose* Programmiersprache zurückgefallen werden kann.

Benutzbarkeit Jeder Ansatz, egal wie verständlich die Logik präsentiert wird oder wie redundanzfrei sie implementiert werden kann, ist beschränkt durch die Benutzbarkeit. Sie ist ein Minuspunkt für viele Codegeneratoren und Modellierungswerkzeuge, die den Entwicklungsprozess durch weitere Schritte erweitern.

Insbesondere spielt die Ausführungsperformanz der Programmiersprache nur eine untergeordnete Rolle. Die Performanz wird haupsächlich durch die gewählten Algorithmen, Datenstrukturen und I/O-Operationen (wie Datenbankzugriffe), die sich außerhalb der Geschäftslogik befinden, beeinflusst.

3. Modelle und Paradigmen

In diesem Kapitel werden einige Programmierparadigmen vorgestellt. Die vier gängigsten sind die imperative, objekt-orientierte, funktionale und logische Programmierung [Seb12]. Programmierparadigmen sind aber keinesfalls disjunkte Mengen, sondern nur eine grobe Zuordnung von Programmiersprachen. Viele Sprachen haben zwar eine Hauptausrichtung, jedoch gibt es kaum eine, die sich nur genau einem Programmierparadigma zuordnen lässt. So sind beispielsweise viele objekt-orientierte Sprachen grundsätzlich imperativ. Auch gibt es viele funkionale Programmiersprachen die Objekt-orientierung unterstützen. Genauso gibt es objekt-orientierte Sprachen mit funkionalen Elementen (siehe Lambda-Ausdrücke und Streams in Java 8).

Van Roy [Van09] sieht Programmierparadigmen als eine wage Sammlung von Konzepten. Dies gibt Hilfestellung bei der Suche nach der richtigen Programmiersprache für ein Problem. Wenn man ständig ein bestimmtes Konstrukt verwendet um ein Problem zu lösen, dann ist das ein Indiz für ein fehlendes Konzept der Programmiersprache. Van Roy nennt das das *creative extension principle*. Als Beispiel nennt Van Roy ein Konzept bei dem Methoden Fehler erkennen und weiterleiten können. Jeder Rückgabewert muss um mögliche Fehlercodes erweitert werden, welche von übergeordneten Methoden überprüft und eventuell weitergeleitet werden müssen. All dies lässt sich vereinfacht durch ein einzelnes Konzept darstellen: *exceptions*.

3.1. Imperative Programmierung

Imperative Programmiersprachen zeichnen sich dadurch aus, dass Befehle in einer fest definierten Reihenfolge abgearbeitet werden. Sie basieren direkt auf der Von-Neumann-Architektur. Viele der frühen Programmiersprachen sind imperativ (Fortran, COBOL, C) und auch viele der heute verbreiteten Programmiersprachen haben ein imperative Basis (Java, C++).
Ein wichtiges Merkmal der imperativen Programmierung ist das Vorhandensein eines Zustands im Form von Variablen, die während der Programmausführung wiederholt gelesen und verändert werden. Als Erweiterung zur imperativen Programmierung versteht sich die strukturierte Programmierung (oder auch prozedurale Programmierung) mit dem Ziel, das Problem in überschaubare Teilprobleme aufzuteilen. Auf unterster Ebene sollen nur folgende Elemente verwendet werden: Sequenzen von Befehlen, Verzweigungen und Schleifen. Insbesondere soll kein *goto* verwendet werden, da dies den Quelltext unverständlich macht. Die strukturierte Programmierung ist heutzutage weit verbreitet.

3.2. Objekt-orientierte Programmierung

Die zentralen Elemente des objekt-orientierten Programmierparadigmas sind Klassen und Objekte. Ein Klasse ist eine abstrakte Beschreibung eines *Dings*. Ein Objekt ist eine Instanz eines solchen *Dings* mit konkreten Werten. Andere wichtige Konzpte in der objekt-orientierten Programmierung sind:

Datenkapselung Eine Klasse hat die Kontrolle über ihre Daten und bietet nach außen hin eine Menge von Methoden als Schnittstelle an.

Vererbung Klassen können voneinander erben. Eine Subklasse kann die Methoden der Superklasse überschreiben und so ein anderes Verhalten definieren.

Polymorphie Objekt-orientierte Programmiersprachen bieten in der Regel drei Arten von Polymorphie an.

- **Parametrische Polymorphie** Eine Funktion oder ein Datentyp kann generische Variablen und Werte verarbeiten ohne von dem kontreten Type abzuhängen. In Java ist die parametrische Polymorphie bekannt als *Generics*.
- **Inklusionspolymorphie** Diese Polymorphie ist eng verbunden mit dem Prinzip der Vererbung. Eine Funktion die einen Wert vom Typ A verarbeiten kann, kann auch einen Wert vom Typ B verarbeiten, wenn Typ B von Typ A erbt.
- **Ad-hoc-Polymorphie** Dies beschreibt polymorphe Funktionen bei denen die konkrete Implementierung vom Typ der Parameter abhängt. Dieses Prinzip wird auch als Methodenüberladung bezeichnet.

Viele objekt-orientierte Sprachen haben in den letzten Jahrzehnten eine weite Verbreitung gefunden. Aber auch viele ursprünglich rein imperative Sprachen wurden über die Jahre durch objekt-orientierte Elemente erweitert (bspw. COBOL) um sich dem Trend anzupassen. Objekt-orientierte Programmiersprachen haben viele positive Neuerungen gegenüber imperativen Programmiersprachen gebracht. Jedoch bringen diese neuen Konzepte auch Effekte wie den Beaujolais-Effekt mit sich, dessen Wirkung sich viele Entwickler nicht bewusst sind. Dieser Effekt beschreibt die Situation in der das Hinzufügen oder Wegnehmen einer einzelnen Methodendeklaration die Semantik eines anderen Moduls ohne Warnung verändert. Die Auswirkungen des Beaujolais-Effekts werden meistens stillschweigend hingenommen, jedoch gerade bei großen, unübersichtlichen Vererbungshierarchien und wenn von fremden Bibliotheken geerbt wird, kann es zu Situationen kommen, in denen nicht mehr nachvollzogen werden kann, warum sich das Programm nicht wie beabsichtigt verhält.

Listing 3.1 Beispielfunktion *filter* in Haskell

```
filter :: (a -> Bool) -> [a] -> [a]
filter pred [] = []
filter pred (x:xs)
  | pred x     = x : filter pred xs
  | otherwise  = filter pred xs

filter (<5) [1, 2, 3, 4, 5, 6, 7] -- ergibt [1, 2, 3, 4]
```

3.3. Funktionale Programmierung

Funktionale Programmierung beruht auf dem λ-Kalkül, das in den 1930er Jahren von Alonzo Church entwickelt wurde [Chu41]. Ein funktionales Programm besteht aus einer Menge von Funktionen die wiederum selbst durch Funktionen und gebundene Variablen definiert sind.

3.3.1. Geschichte

Die erste funktionale Programmiersprache war *Lisp* 1958 mit dem Hauptziel der Listenverarbeitung. Aus *Lisp* entstanden die Dialekte *Scheme* und *Common Lisp*. Beide fanden jedoch nur wenig Verbreitung. *Scheme* wird hauptsächlich verwendet um funktionale Progammierung zu lehren. Die Entwicklung der statischen Typisierung führte zu den Sprachen *ML*, *Haskell*, *OCaml* und *F#*. *F#* ist die funktionale Programmiersprache von Microsoft für die .NET-Umgebung. Auch für die JVM wurden funktionale Programmiersprachen entwickelt. Diese sind das statisch typisierte *Scala*, das dynamisch typisierte *Clojure* und der *Erlang*-Dialekt *Erjang*

3.3.2. Konzepte

Funktionen höherer Ordnung

Funktionen höherer Ordnung sind solche, die ihrerseits Funktionen als Parameter übergeben bekommen oder eine Funktion zurückgeben. Als Beispiel ist die Haskell-Funktion *filter* in Codeausschnitt 3.1 gegeben. *Filter* bekommt eine Prädikatsfunktion und eine Liste von Elementen übergeben und gibt eine Liste aller Elemente zurück, für die die Prädikatsfunkion *True* ergibt. Dadurch, dass Funktionen selbst Parameter oder Rückgabewert sein können, nennt man sie auch *first-class functions*. Eng damit verbunden ist das Konzept der anonymen Funktionen, also Funktionen ohne Namen. In Beispiel 3.1 ist die Prädikatsfunktion *(<5)* eine solche anonyme Funktion. Als Synonym für anonyme Funktion wird auch häufig der Begriff *Lambda-Funktion* verwendet.

Reine Funktionen

In funktionalesn Programmiersprachen gibt es selten einen expliziten Zustand und somit auch keinen Zuweisungsoperator. Funktionen, die bei gleichen Eingabeparametern immer das selbe Ergebnis liefern, bezeichnet man als *rein*. *Reinheit* ist ein wichtiges Konzept, da es Seiteneffekte ausschließt und somit die Testbarkeit und Modifizierbarkeit des Programms erhöht. Um Reinheit auch für Programmausgaben zu gewährleisten, wurde in Haskell das Konzept der Monaden entwickelt. Nur wenige funktionale Programmiersprachen erzwingen Reinheit. Viele erlauben in irgendeiner Form einen globalen Zustand oder Seiteneffekte.

Rekursion

In rein funktionalen Programmiersprachen gibt es nicht die aus dem imperativen Paradigma bekannten *for*- und *while*-Schleifen. Deswegen wird häufig Rekursion verwendet, um sich wiederholende Aktionen umzusetzen. Auf den ersten Blick mag das erschreckende Auswirkungen auf die Performanz haben. Für viele funktionale Programmiersprachen gibt es jedoch ausgeklügelte Compiler, die Endrekursionen optimieren können.

Algebraische Datentypen und Pattern Matching

Algebraische Datentypen sind ein Konzept, das es in einigen (aber nicht allen) funktionalen Programmiersprachen gibt. Man unterscheidet drei Arten von algebraischen Datentypen:

Aufzählungstypen Beispiel: Wochentage. Aufzählungstypen sind vergleichbar mit *Enums* in anderen Programmiersprachen.

Produkttypen Beispiel: Datum (Tag, Monat, Jahr). Diese Art wird Produkttyp genannt, da der Zustandsraum das Produkt der Zustandsräume der einzelnen Komponenten ist.

Summentypen Beispiel: Baum = Nichts | Blatt Integer | Knoten Baum Baum.

Algebraischen Datentypen (gerade Summentypen) zeigen ihre Stärke erst in Kombination mit Pattern Matching. Pattern Matching erlaubt eine Fallunterscheidung auf Struktur und Inhalt von Werten. Als Beispiel ist in Codeausschnitt 3.2 eine Funktion gegeben, die die Summe aller Elemente in einem Baum berechnet.

3.4. Logische Programmierung

Die Logische Programmierung basiert auf dem Konzept der Prädikatenlogik. Ein Programm wird dabei als Menge von Fakten und Relationen implementiert. An dieses System können dann Fragen (sog. *Queries*) gestellt werden, die mit einer ausgeklügelten Tiefensuche ausgewertet werden. Ein Programm in der logischen Programmiersprachen Prolog ist in Beispiel 3.3

Listing 3.2 Beispielfunktion mit Pattern Matching in Haskell

```
data Baum a = Nichts | Blatt a | Knoten a Baum Baum

summe :: Baum Int -> Int
summe Nichts = 0
summe (Blatt wert) = wert
summe (Knoten baumA baumB) = a + (summe baumA) + (summe baumB)

summe (Knoten 5 (Blatt 3) (Blatt 4)) -- ergibt 12
```

Listing 3.3 Beispielprogramm in Prolog

```
vater(klaus, uwe).
vater(hans, klaus).

enkel(Z, X) :-
      vater(X, Y),
      vater(Y, Z).

enkel(Z, hans). % ergibt: Z = uwe
```

dargestellt. Die ersten beiden Zeilen modellieren die Fakten, dass Klaus der Vater von Uwe und Hans der Vater von Klaus ist. Anschließend kommt die Relation *enkel*, die aussagt, dass eine Person X der Enkel einer Person Z ist, wenn es eine Person Y gibt, sodass X der Vater von Y und Y der Vater von Z ist. Anschließend kommt die *Query* nach allen Personen Z, die ein Enkel von *Hans* sind. Das Ergebnis in diesem Beispiel ist *Uwe*.

Die Eleganz der logischen Programmierung liegt darin, dass Relationen in der Regel in mehrere Richtungen ausgewertet werden können. In Beispiel 3.3 verdeutlicht heißt das, dass die Relation *enkel* verwendet werden kann, um alle Personen Z zu finden, von denen eine gegebene Person Z der Enkel ist. Aber sie kann auch verwendet werden, um alle Personen Z zu finden, die Enkel einer gegebenen Person X sind. Sie kann sogar dazu verwendet werden, zu testen ob eine gegebene Person Z der Enkel einer gegebenen Person X ist.

Logische Programmierung findet Anwendung in der Umsetzung von Expertensystemen und der maschinellen Sprachverarbeitung, in der sie auch ihren Ursprung hat. Bekannte Sprachen sind Prolog und Datalog.

3.5. Language-oriented Programming

Der Begriff *Language-oriented Programming* wurde 1994 von Martin P. Ward geprägt [War94]. Er bezeichnet damit einen neuen Ansatz der Softwareentwicklung. In einem ersten Schritt wird dabei eine hohe Programmiersprache konzipiert, die speziell auf die Problemstellung zugeschnitten ist. Darauf aufbauend werden parallel der Übersetzer für die neue Programmiersprache und das eigentliche System in der Programmiersprache entwickelt. Als Vorteil

dieses Ansatzes wird vorallem die mögliche neue Höhe der domänenspezifischen Sprache (englisch *domain-specific language*, kurz DSL) genannt, durch die sich das System mit weniger Code-Zeilen darstellen lässt und dadurch der Entwicklungs- und Wartungsaufwand reduziert wird. Weitere Vorteile sind die Kontrolle über das Sprachdesign, die Möglichkeit zur Wiederverwendung der DSL in ähnlichen Projekten und die Abgrenzung von Problemstellung und Implementierungsdetails.

Marting Fowler bechreibt in einem Artikel drei Formen von DLSs [Fow05]:

Interne DSL: Eine interne DSL ist in der Zielsprache implementiert. Dadurch hat der Entwickler Zugriff auf alle Vorteile von Werkzeugen und Entwicklungsumgebungen die er auch für die Zielsprache hat. Allerdings ist die DSL durch die Konzepte und Syntax der Zielsprache eingeschränkt.

Externe DSL: Eine externe DSL ist eine komplett neue Sprache die durch einen Übersetzer in eine Zielsprache übersetzt wird. Der Sprachentwickler hat dabei völlig frei Wahl bei der Konzipierung der Sprache und muss sich nicht an bestehenden Formen richten.

Language Workbenches: Language Workbenches können als DSL für DSLs verstanden werden. Die Language Workbench *MPS* von JetBrains [Dmi04] verwendet drei Sprachen um eine DSL zu definieren. Mit der *Structure Language* wird die grundlegende Struktur der Sprache festgelegt. Mit der *Editor Language* wird definiert, wie und in welcher Form die DSL dem Benutzer dargestellt wird. Als drittes wird in der *Transformation Language* definiert, wie die DSL in eine Zielsprache übersetzt wird.

3.6. Kalkulationstabellen

Kalkulationstabellen sind Software die Daten in Zeilen und Spalten verwaltet. Eine einzelne Zelle enthält entweder einen konstanten Wert (numerisch oder alphanumerisch), oder eine Formel, die einen Wert in Abhängigkeit von anderen Zellen berechnet. Bekannte Software für Kalkulationstabellen sind Microsoft Excel, Google Spreadsheets und OpenOffice Calc
Kalkulationstabellen können durch Skripte und Formeln durchaus komplexe Verhalte modellieren, was durch die automatische Auswertung von Formeln und Abhängigkeiten unterstützt wird. Durch einfache Bedienungskonzepte sind sie auch für Laien schnell zugänglich. Kalkulationstabellen sind als ausführendes Element in Großprojekten jedoch ungeeignet, da ihnen viele Eigenschaften fehlen, die wir bei Programmtext als selbstverständlich erachten. Das sind unter anderen die fehlende Unterstützen der Fehlerfindung sowie mangelhafte Möglichkeiten der Versionierung, was die Grundlage für paralleles Arbeiten von mehreren Entwicklern ist.

4. Stand der Forschung

In diesem Kapitel werden existierende Forschungsergebnisse zum Vergleich von Programmierparadigmen vorgestellt.

Nanz und Furia [NF15] untersuchten 7087 Programme in 8 verschiedenen Sprachen aus dem Rosetta Code Repository[1] auf Länge, Laufzeit, RAM-Verbrauch und weiteres. Das Rosetta Code Repository bietet für über 700 festgelegte Aufgabenstellungen Lösungen in vielen verschiedenen Sprachen. Die Aufgabenstellungen reichen von Mathematik über Sortieralgorithmen bis hin zu grundlegender I/O, sind jedoch zum Großteil berechnender Natur. Die Programmiersprachen, die verglichen wurden, sind: C und Go (imperativ); C# und Java (objekt-orientiert); F# und Haskell (funktional); Python und Ruby (Skriptsprachen). Nanz und Furia kamen zu dem Ergebnis, dass Programme in funktionalen Sprachen und Skriptsprachen etwa halb so lang sind als imperative und objekt-orientiert Sprachen (Faktor 2,2–2,9).

Pankratius et al. [PSG12] haben 13 Entwickler bei der Entwicklung von 3 parallelen Programmen, jeweils in Java und Scala, beobachtet und bestätigen, dass die Programme in Scala kürzer waren als die in Java. Allerdings fanden sie auch heraus, dass der Entwicklungsaufwand der Scala-Programme um ca. 30% größer war. Dies lässt sich ihren Ergebnissen zufolge auf den höheren Test- und Debugging-Aufwand zurückführen. Die Teilnehmer berichteten, dass Features wie die automatische Typinferenz, die eigentlich den Entwicklungsaufwand verringern sollen, im Gegenzug die Fehlersuche erschweren. Es ist wichtig anzumerken, dass alle Teilnehmer im Voraus ein vierwöchiges Training in Scala und Java erhalten haben. Die Ergebnisse lassen sich also nicht auf Unterschiede in den Programmierkenntnissen zurückführen.

Zu dem Ergebnis, dass funktionale Implementierungen kürzer sind als objekt-orientierte, kommen auch die Autoren [Cou14] und [McL12]. Cousins berichtet von einem System zur Berechnung von Ausgleichsleistungen im Stromnetz des Vereinigten Königreich, das sowohl und C# und in F# implementiert wurde. Seinem Bericht zufolge, hat die funktionale Implementierung eine Codereduzierung von ca. Faktor 11 in der Anzahl der Codezeilen gebracht (30.801 Zeilen in F# gegenüber 348.430 Zeilen in C#). McLoone hat, wie Nanz und Furia, Programme aus dem Rosetta Code Repository verglichen und kam zu dem Ergebnis, dass funktionale Implementierungen sowohl in der Anzahl an Zeilen, als auch in der Anzahl an Zeichen und Anzahl an Token sehr gut abschneiden.

[1] rosettacode.org

Sprachniveau	Sprache	Programmierbefehle pro *Function Point*
1,00	Assembler	320,00
2,50	C	128,00
3,00	COBOL	106,67
5,00	Lisp	64,00
5,00	Prolog	64,00
6,00	C++	53,33
6,00	Java	53,33
6,25	C#	51,20
8,50	Haskell	37,65
12,00	Objective C	26,67

Tabelle 4.1.: Sprachniveaus einiger Programmiersprachen

Capers Jones argumentiert hingegen, dass *Lines of Code* (LOC) eine ungeeignete Metrik ist, um Programmiersprachen zu vergleichen [Jon12]. Er beschäftigt sich daher intensiv mit *Function Points* [Jon13]. *Function Points* ist eine Metrik, die die Menge an Funktionalität einer Software misst. Sie kann verwendet werden um die Kosten für ein Projekt vorherzusagen. Dazu wurden Programmiersprachen in Sprachniveaus unterteilt. Je größer das Sprachniveau, desto weniger Programmierbefehle werden für einen *Function Point* benötigt. Ein Ausschnitt von bekannten Programmiersprachen ist in Tabelle 4.1 angegeben. Als Maßstab wird Assembler mit dem Sprachniveau 1.0 und 320 Programmierbefehle pro *Function Point* verwendet. Rein imperative Sprachen wie C und COBOL scheinen ein eher niedriges Sprachniveau zu haben. Zwischen funktionalen und objekt-orientierten Sprachen lässt sich jedoch keine eindeutige Abgrenzung ausmachen.

Ray et al. [RPFD14] verglichen 729 Projekte von GitHub[2] mit insgesamt etwa 80 Millionen Zeilen und untersuchten die Korrelation zwischen Sprachklassen und Anzahl der Fehlerbehebungen. Dazu wurden die Programmiersprachen in folgenden drei Klassen aufgeteilt: C, C++, C#, Objective–C, Java, Go (prozedural); CoffeeScript, JavaScript, Python, Perl, PHP, Ruby (Skriptsprachen); Clojure, Erlang, Haskell, Scala (funktional). Es wurde also keine Abgrenzung von objekt-orientierten und imperativen Sprachen gemacht. Ray et al. kamen zu dem Ergebnis, dass funktionale Programme weniger fehleranfällig sind als Programme in prozeduralen Sprachen und Skriptsprachen. Außerdem kamen sie zu dem Ergebnis, dass stark typisierte Sprachen gegenüber schwach typisierten Sprachen und statisch typisierte Sprachen gegenüber dynamisch typisierten Sprachen vorzuziehen sind, wenn es um die Fehleranfälligkeit geht. Dies wird von Nanz und Furia bestätigt [NF15].

[2] github.com

5. Fallbeispiele

Es wurden zwei Fallbeispiele ausgewählt, die in einer funktionalen Programmiersprache umgesetzt und mit bestehenden Lösungen in imperativen und objekt-orientierten Sprachen verglichen wurden. Dazu wurde eine Umfrage mit Mitarbeitern der itestra GmbH durgeführt.

5.1. Fallbeispiel 1: Berechnungslogik

Das erste Fallbeispiel ist aus dem Bereich der Kapitalgeschäfte und lässt sich der Berechnungslogik zuordnen.

Hintergrund

Erträge aus Kapitalvermögen unterliegen in Deutschland der Kapitalertragssteuer. Dies ist eine Quellensteuer und wird somit von der auszahlenden Stelle (z.B. eine Bank) an das Finanzamt abgeführt. Nun gibt es aber Möglichkeiten Teile dieser Erträge von der Besteuerung zu befreien. Dies kann zum Beispiel durch einen Frestellungsauftrag oder die Verrechnung mit Verlusten aus dem Vorjahr geschehen. Dabei können Verluste aus Aktiengeschäften allerdings nur mit Gewinnen aus Aktiengeschäften verrechnet werden.

Tabelle 5.1 zeigt an einem Beispiel wie die Verrechnung von Verlusten und Freistellung aussehen könnte. Die grau hinterlegten Zellen kennzeichen die Eingabewerte für die Berechnung. Die Bruttoerträge sind die Erträge vor der Verrechnung. Die Nettoerträge sind die Erträge, die am Ende zu versteuernd sind. Erträge werden immer von Links nach Rechts und von Oben nach Unten verrechnet. Eine Ausnahme bildet der Sonderfall *VVTA*, da, wie oben beschrieben, Verluste aus Aktiengeschäften nur mit Gewinnen aus Aktiengeschäften verrechnet werden können.

Codeausschnitte

Die itestra GmbH hat diese Berechnungslogik aus einem Legacy-System in Cobol extrahiert, analysiert und mit einem neuen Algorithmus (wie oben beschrieben) in Java umgesetzt. Im Rahmen dieser Arbeit wurde die Berechnungslogik funktional (in F#) implementiert. Die objekt-orientierte Lösung (siehe Anhang A.1) verwendet eine doppelte for-Schleife (Zeilen

Ertragsart	Bruttoertrag	VVTA	VVTS	FSA	Nettobetrag
Verfügbar	-	20.00	10.00	80.00	-
AD	50.00	0.00	10.00	40.00	0.00
ZI	30.00	0.00	0.00	30.00	0.00
ID	0.00	0.00	0.00	0.00	0.00
VGA	100.00	20.00	0.00	10.00	70.00
Summe	190.00	20.00	10.00	80.00	70.00

Abkürzungen:

AD – Ausländische Dividenden

ZI – Zinsen

ID – Inländische Dividenden

VGA – Veräußerungsgewinne Aktien

VVTA – Verlustverechnungstopf Aktien

VVTS – Verlustverechnungstopf Sonstiges

FSA – Freistellungsauftrag

Tabelle 5.1.: Beispieltabelle zur Berechnung der zu versteuernden Beträge

28–36) um über die Ertragsarten und Verrechnungsarten zu iterieren. Temporäre Daten werden dabei in einem Objekt von Typ *TaxOffsetBucketUtilizationBBE* gespeichert und während der Berechnung aktualisiert. Ob eine Ertragsart verrechnet werden kann (vgl. Sonderfall *VVTA*), wird in der Methode *taxOffsetBucketCanBeUtilized* (Zeilen 73–80) entschieden.

Die funktionale Lösung (siehe A.2) verwendet Rekursion in der Funktion *berechneTabelleSummenVorher* (Zeilen 36–48) um über die Verechnugsarten zu iterieren. Über die Ertragsarten wird mit der abstrakten Funktion *List.scan* iteriert (Zeile 16). Der Logik für den Sonderfall *VVTA* ist in der Funktion *berechneVVTA* (Zeilen 24–30) getrennt von der allgemeinen Berechnung in der Funktion *berechneSpalte* (Zeilen 18–22) modelliert. Beide Funktionen haben die Signatur *Spaltenberechnung* (Zeile 9) die von der Funktion *berechneTabelleSummenVorher* (Zeilen 36–48) verwendet wird.

5.2. Fallbeispiel 2: Entscheidungslogik

Das zweite Fallbeispiel ist ebenfalls aus dem Bereich Kapitalgeschäfte und lässt sich zu Entscheidungslogik zuordnen.

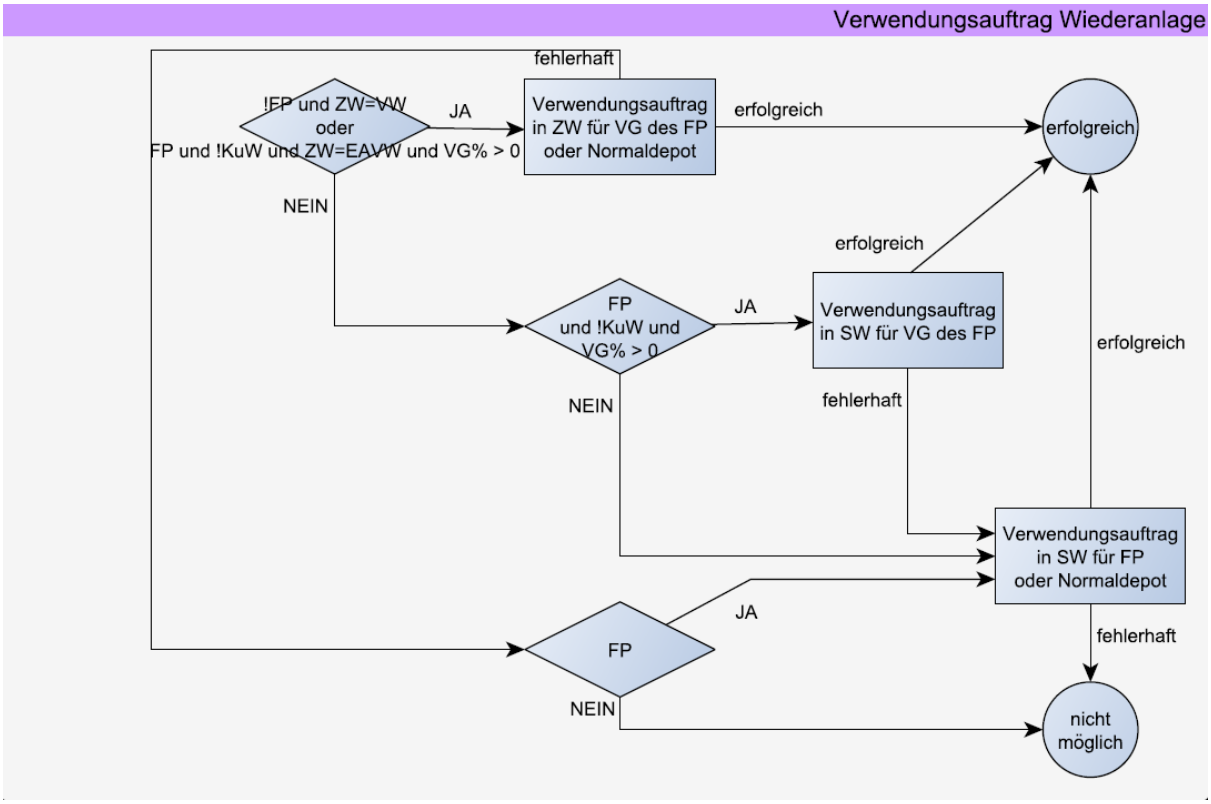

Abbildung 5.1.: Verwendungsauftrag Wiederanlage

Hintergrund

Wenn es einen Gewinn aus Kapitalgeschäften gibt, dann muss entschieden werden was mit diesem Gewinn passiert. Diese Entscheidung hängt von vielen Faktoren ab: Gibt es eine Kundenweisung? Soll der Gewinn ausgezahlt oder wiederangelegt werden? Was ist die Zahlungwährung und was ist die Vermögenswährung? Als konkreten Ausschnitt wurde der Fall der versuchten Wiederanlage gewählt. In Abbildung 5.1 ist die Logik dieser Entscheidungsfindung als Graph dargestellt.

Codeausschnitte

Die Entscheidungslogik wurde von der itestra GmbH in Java und im Rahmen dieser Arbeit in F# umgesetzt. Wie man in Anhang A.4 und Anhang A.5 sieht, implementieren beide Lösungen die Entscheidungslogik als Kette von if-Verzweigungen. In der funktionalen Implementierung wurde der Versuch unternommen, die Logik gemäß Abbildung 5.1 zu strukturieren.

5.3. Umfrage

Um die Codeausschnitte auf Verständlichkeit hin zu vergleichen, wurde eine Umfrage mit Experten der itestra GmbH und Studenten durchgeführt.

Durchführung

Die Umfrage wurde mit 6 Teilnehmern (4 Experten, 2 Studenten), unabhängig voneinander, durchgeführt. Dabei wurden alle Teilnehmer zuerst in die Thematik eingeführt und beantworteten die Fragen anschließend selbständig. Keiner der teilnehmenden Mitarbeiter von itestra hatte vorher näheren Kontakt mit dem Projekt aus dem die Fallbeispiele stammen. Die Umfrage hatte zwei Teile, einen zu jedem Fallbeispiel. Ziel der Umfrage war es, die verschiedenen Implementierungen der Fallbeispiele auf Verständlichkeit hin zu untersuchen. Dies wurde in Kapitel 2 als Hauptkriterium für gute Geschäftslogik identifiziert, da es die Wartungskosten gering hält.

Ergebnis

Zuallererst wurden die Teilnehmer gebeten ihre Kenntnisse der einzelnen Paradigmen einzuschätzen. Abbildung 5.2 zeigt, dass die Teilnehmer (wie erwartet) mehr Erfahrung mit objekt-orientierten Sprachen als mit funktionalen oder rein imperativen Sprachen haben. Dies hat natürlich Einfluss darauf, wie natürlich und verständlich die Teilnehmer die verschiedenen Codeausschnitte der Umfrage empfinden. Der Grad dieses Einflusses lässt sich jedoch durch die geringe Teilnehmerzahl nicht feststellen.

Im ersten Teil wurden den Teilnehmern nacheinander die Codeausschnitte zum Fallbeispiel Berechnungslogik präsentiert. Die Teilnehmern sollten dann in allen Codeausschnitten die Behandlung des Sonderfalls *VVTA* (vgl. Abschnitt 5.1) finden, die generelle Verständlichkeit des Codes bewerten und einzelne Stellen nennen, die sie als besonders unverständlich empfinden. Abbildung 5.4 zeigt, dass die Verständlichkeit des funktionalen Codeausschnitts ein wenig besser bewertet wurde als die der anderen. Aus den Kommentaren lässt sich jedoch herauslesen, dass dies hauptsächlich an der Wahl der Bezeichner liegt, was die Forschung von Tashtoush et al. [TOAY13] bestätigt. Ein Teilnehmer bemängelte die Zeilen 15 und 16 im funktionalen Codeausschnitt (Anhang A.2). Die Funktion *baueSpaltenberechnung* erhöht zwar die Abstraktion und verringert so die Duplikation, jedoch ist das Konstrukt durch versteckte Parameter und fehlende Signatur äußerst unverständlich.

Im zweiten Teil wurden die Teilnehmer nacheinander die Codeausschnitte zum Fallbeispiel Entscheidungslogik präsentiert. Die Teilnehmer sollten dann versuchen zu verstehen unter welcher Bedingung die Wiederanlage fehlschlägt, die generelle Verständlichkeit des Codes bewerten und wieder die Stellen nennen, die sie als besonders unverständlich empfinden.

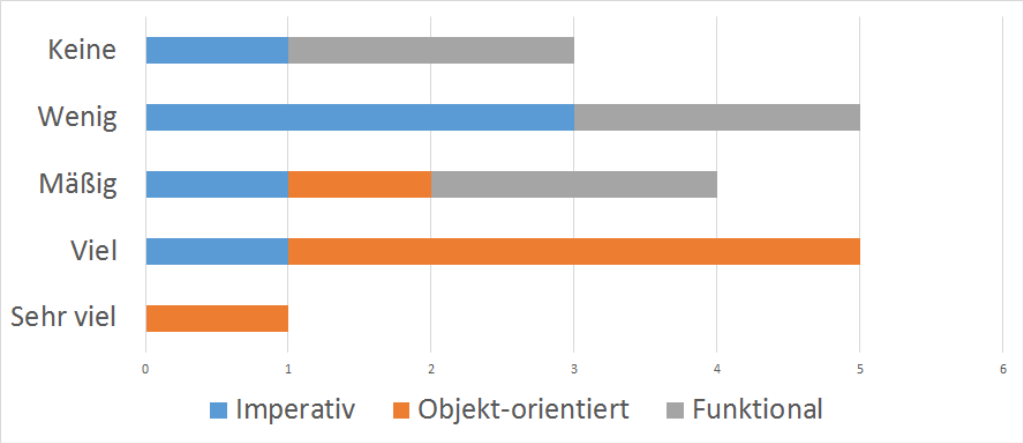

Abbildung 5.2.: Erfahrung mit den Paradigmen

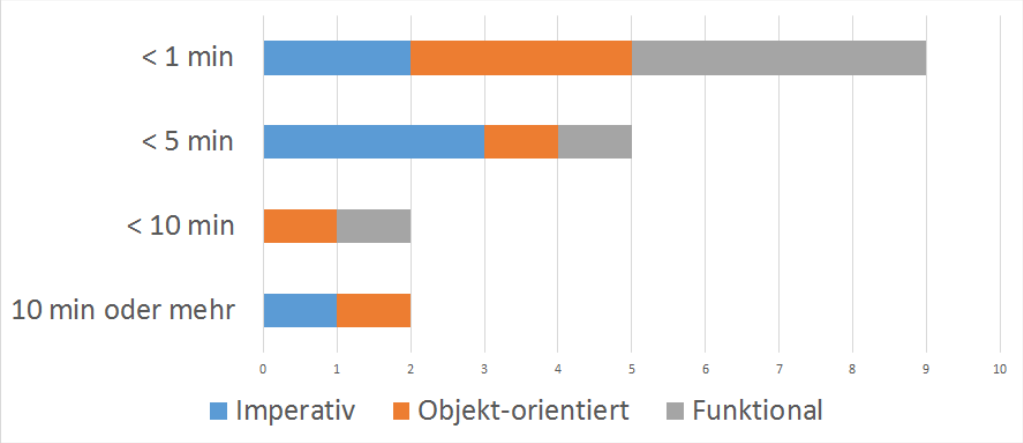

Abbildung 5.3.: Benötigte Zeit um den Sonderfall zu finden

Aus den Abbildungen 5.5 und 5.6 lässt sich herauslesen, dass der funktionale Codeausschnitt verständlicher ist als der objekt-orientierte. Die Kommentare zeigen aber auch wieder, dass dies hauptsächlich an der Wahl der Bezeichner und der generellen Darstellung (wie Zeilenlänge) liegt. Tatsächlich sind beide Codeausschnitte, wie ein Teilnehmer kommentierte, "weitgehend ein imperatives Programm". Die Logik dieser Entscheidung lässt sich also auf unterster Ebene nur durch eine Verkettung von if-then-else-Verzweigungen darstellen für die objekt-orientierte oder funktionale Konzepte keine Verbesserung bieten. Wie sich nach Gesprächen mit den Teilnehmern auch herausstellte, fand keiner alle Bedinungen die für ein Fehlschlagen der Wiederanlage nötig sind. Es lässt sich also festellen, dass eine zeilenweise Darstellung mit if-Verzweigungen für komplexe Entscheidungslogik ungeeignet für eine einfache Verständlichkeit ist.

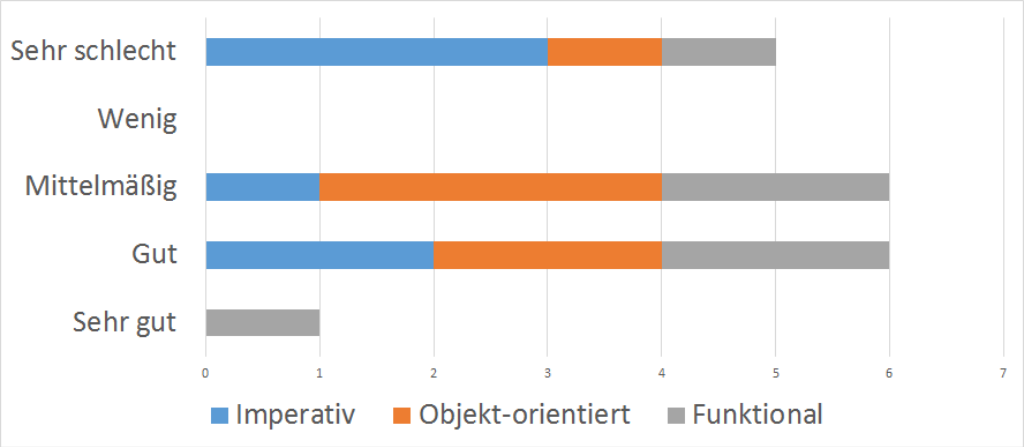

Abbildung 5.4.: Verständlichkeit der Codeausschnitte zu Berechnungslogik

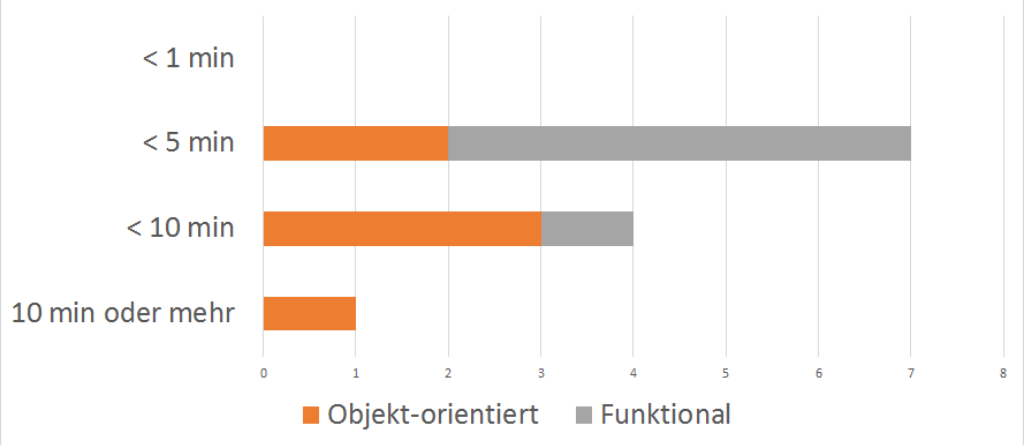

Abbildung 5.5.: Benötigte Zeit für das Verstehen der Entscheidungslogik

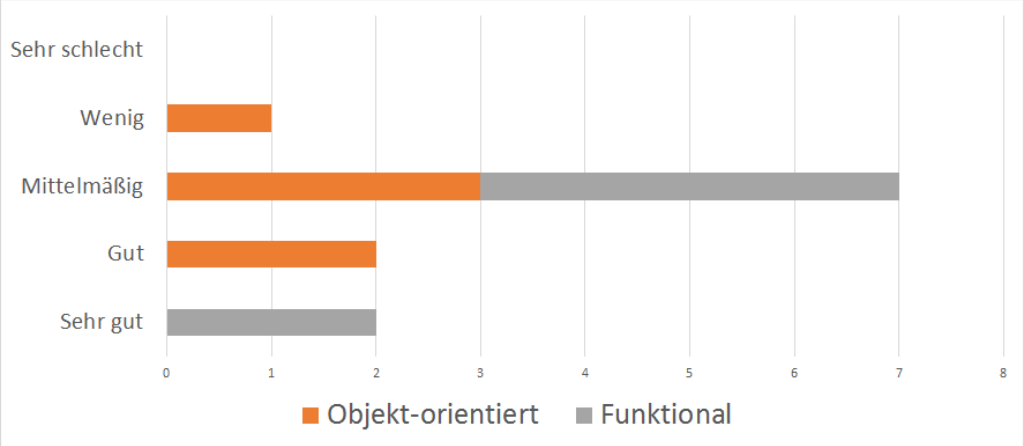

Abbildung 5.6.: Verständlichkeit der Codeausschnitte zu Entscheidungslogik

Threats to Validity

Es gibt einige Punkte zu nennen, die die Aussagekraft dieser Umfrage bedrohen. Der erste Punkt ist, dass die imperativen und objekt-orientierten Codeausschnitte aus Projekten der itestra GmbH stammen und somit mit allen nötigen Details behaftet sind. Die funktionalen Codeausschnitte jedoch sind nur Prototypen, die zwar ausführbar sind, aber nicht jedes Detail behandeln. Ein weiterer Faktor sind die Programmiersprachen, in denen die Codeausschnitte implementiert sind. Cobol und Java sind vermutlich die am weitesten verbreiteten Sprachen in ihrem Paradigma, doch sie sind nicht zwingend repräsentativ dafür. Java ist dafür bekannt, eine sehr ausführliche Syntax zu haben und Bezeichner in Cobol sind meist sehr kryptisch, was als einer der wichtigsten Punkte für schlechte Lesbarkeit identifiziert wurde [TOAY13].

6. Leitfaden

Das wichtigeste Kriterium für die Wahl einer Programmiersprache ist die Erfahrung des Entwicklerteams [JB12]. Ein Entwickler der seit Jahren nur objekt-orientierte Sprachen verwendet wird kaum effizient arbeiten, wenn er plötzlich eine funktionale Sprache verwenden soll. Das gleiche gilt umgekehrt. So sollte dieser Leitfaden unter der Annahme verstanden werden, dass das Entwicklerteam alle vorgeschlagenen Paradigmen gleich gut beherrscht.

Die Wahl des geeignesten Paradigma wird von vielen Faktoren eines Projekts bestimmt. Das primäre Ziel ist es, die Kosten gering zu halten. Wie in Kapitel 2 herausgearbeitet wurde, gelingt das in der Regel durch eine hohe Verständlichkeit und eine geringe Fehleranfälligkeit des Programms, um so die Wartungskosten gering zu halten.
Dies gilt aber nur für eine gewisse Langlebigkeit des Projekts. Yuri Khramov [Khr06] hat gezeigt, dass es auch kurzlebige Projekte gibt, für die die Entwicklungszeit priorisiert werden sollte. Wie Pankratius et al. [PSG12] herausgefunden haben, scheint die Entwicklungszeit von funktionalen Programmen länger zu sein als die von objekt-orientierten Programmen. Auch erscheint es als seien dynamische Programmiersprachen vorzuziehen [Han10], wenn es um die Entwicklungszeit geht. Für kurzlebige Projekte in denen die Entwicklungskosten die Wartungskosten überwiegen ist also eine dynamische, objekt-orientierte Programmiersprache wie Python oder Ruby zu empfehlen.

Die Forschung (Kapitel 4) und die Umfrage (Abschnitt 5.3) haben gezeigt, dass Verständlichkeit zum Großteil von paradigmen-unabhängigen Parametern (wie Benennung und Kommentare) abhängt [TOAY13]. Es wurde aber von Ray et al. [RPFD14] gezeigt, dass funktionale Programme weniger fehleranfällig sind. Nanz und Furia [NF15] haben gezeigt, dass statische, stark-typisierte Sprachen weniger fehleranfällig sind als dynamische, schwach-typisierte Sprachen. Für langlebige Projekte in denen die Wartungskosten dominieren sollte also eine funktionale, statische, stark-typisierte Programmiersprache wie Haskell, F# oder Scala verwendet werden. Da Ray et al. [RPFD14] keine genaue Domänenunterscheidung gemacht haben, scheinen die Ergebnisse für alle Arten von Geschäftslogik zu gelten.

Wie die Umfrage (Abschnitt 5.3) gezeigt hat, ist die zeilenweise Darstellung durch Quellcode nicht sehr übersichtlich für komplexe Entscheidungsgraphen. Ein anderer Ansatz wäre es, eine Darstellung wie in Abbildung 5.1 zu wählen und daraus Code zu generieren. Der Gewinn eines solchen Ansatzes ist die deutlich bessere Verständlichkeit. Die Struktur der Entscheidungslogik lässt sich in kurzer Zeit verstehen. Außerdem lässt sich der Fachbereich mit einer solchen Darstellung leichter in den Entwicklungsprozess einbeziehen. Ein solcher Ansatz kommt aber auch mit Kosten. Unter anderem muss ein geeigneter Editor mit Codegenerierung erst gefunden oder

selbst entwickelt werden. Kapteijns et al. [KJB+09] und Krogmann und Becker [KB07] haben in ihren Vergleichen zwischen modellgetriebenen und traditionellen Ansätzen herausgefunden, dass es nahezu unmöglich ist, alle Sonderfälle mit einem modellgetriebenen Ansatz abzudecken und so immer ein Weg offen gehalten werden muss, solche Sonderfälle traditionell umzusetzen. Daher ist nicht sichergestellt, dass sich ein modellgetriebener Ansatz lohnt. Die Wahrscheinlichkeit steigt aber, wenn der Editor in weiteren Projekten wiederverwendet werden kann. Dazu muss jedoch das Projekt- und Vertriebsmodell der Firma stimmen.

7. Zusammenfassung

In dieser Arbeit wurden die Kriterien für gute Geschäftslogik sowie gängige Paradigmen identifiziert. Anschließend wurden zwei Fallbeispiele aus einem Projekt des Industriepartners gewählt. Diese wurden in einer funktionalen Programmiersprache implementiert und in einer Umfrage mit bestehenden Lösungen in imperativen und objekt-orientierten Sprachen verglichen.

Es hat sich herausgestellt, dass die Verständlichkeit eines Programms hauptsächlich von Aspekten wie der Bennenung von Variablen und Zeilenlänge als von paradigmen-spezifischen Konstrukten abhängt. Dies bestätigt bestehende Forschung. Da die Programme in funktionalen Programmiersprachen jedoch weniger fehleranfällig sind, wird Entwicklern die Verwendung einer funktionalen Programmiersprache zur Umsetzung von Geschäftslogik empfohlen.

Ausblick

Es wurde festgestellt, dass die Verwendung von funktionalen Programmiersprachen zu einer geringeren Fehleranfälligkeit führt. Es ist allerdings keine Forschung bekannt, die untersucht was die Ursache für diese geringere Fehleranfälligkeit ist. Es wäre interessant die Ursachen zu kennen um feststellen zu können, ob ein funktionaler Programmierstil in einer eigentlich nicht funktionalen Programmiersprache (bspw. Streams in Java, Lambda-Ausdrücke in C#) dieselben Vorteile mit sich bringt.

A. Anhang

A.1. Fallbeispiel Berechnungslogik: Objekt-orientiert

```
1  public final class TaxOffsetBucketUtilizationBBE extends AbstractBaseBBE {
2
3        // ...
4
5        public static TaxOffsetBucketUtilizationBBE calculateUtilization(
6             final TaxBucketTypeDBE kindOfTaxBucket,
7             final AmountsPerIncomeTypeBBE yieldPerIncomeType,
8             final Map<TaxOffsetBucketTypeEnum, TaxAmount>
                  utilizableAmountByTaxOffsetBucket) {
9
10             // calculate virtual yield (absolut sum of all negative utilizable amounts)
11             TaxAmount virtualYield = TaxAmount.ZERO;
12             for (final TaxAmount utilizableAmount :
                   utilizableAmountByTaxOffsetBucket.values()) {
13               if (EqualUtils.isLessThanZero(utilizableAmount.getValue())) {
14                     virtualYield = virtualYield.subtract(utilizableAmount);
15               }
16             }
17
18             // initialize calculation helper class (contains currently utilized amounts
                   and remaining not yet exempted yields)
19             final TaxOffsetBucketUtilizationBBE currentlyUtilizedAmounts = new
                   TaxOffsetBucketUtilizationBBE(yieldPerIncomeType, virtualYield);
20
21             // calculate utilized amount for each pair of TaxOffsetBucketType and
                   incomeType
22             final List<Pair<TaxOffsetBucketTypeEnum, TaxAmount>>
                   sortedTaxBucketTypesWithUtilizableAmount = new ArrayList<>();
23             for (final TaxOffsetBucketTypeEnum taxOffsetBucketType :
                   TaxOffsetBucketTypeEnum.getEnumsSortedByPriority()) {
24               sortedTaxBucketTypesWithUtilizableAmount.add(Pair.with(taxOffsetBucketType,
                       TaxAmount.safeAmount(utilizableAmountByTaxOffsetBucket.get(taxOffsetBucketType))));
25             }
26             final List<InvestmentIncomeTypeDBE> sortedIncomeTypes =
                   yieldPerIncomeType.getSortedIncomeTypes();
27
28             for (final Pair<TaxOffsetBucketTypeEnum, TaxAmount>
                   taxBucketTypeWithUtilizableAmount :
                   sortedTaxBucketTypesWithUtilizableAmount) {
```

```
29                  for (final InvestmentIncomeTypeDBE incomeType : sortedIncomeTypes) {
30                        currentlyUtilizedAmounts.calculateUtilization(
31                              taxBucketTypeWithUtilizableAmount.getValue0(),
32                              incomeType,
33                              kindOfTaxBucket,
34                              taxBucketTypeWithUtilizableAmount.getValue1());
35                  }
36            }
37
38            return currentlyUtilizedAmounts;
39      }
40
41      private void calculateUtilization(
42           final TaxOffsetBucketTypeEnum taxOffsetBucketTypeToRecalculateFor,
43           final InvestmentIncomeTypeDBE incomeTypeToRecalculateFor,
44           final TaxBucketTypeDBE kindOfTaxBucket,
45           final TaxAmount utilizableAmountInTaxOffsetBucket) {
46
47            final TaxAmount recalculatedUtilizedAmount;
48
49            if (taxOffsetBucketCanBeUtilized(taxOffsetBucketTypeToRecalculateFor,
                  incomeTypeToRecalculateFor.getId(), kindOfTaxBucket)) {
50
51                  final TaxAmount notYetExemptedYield =
                        getNotYetExemptedYield(incomeTypeToRecalculateFor);
52                  final TaxAmount notYetUtilizedAmount =
                        utilizableAmountInTaxOffsetBucket.subtract(
53                        getSumUtilizedOf(taxOffsetBucketTypeToRecalculateFor));
54
55                  if (notYetUtilizedAmount.isLessThanZero()) {
56                        if
                              (incomeTypeToRecalculateFor.equals(incomeTypeToHandleVirtualYield))
                              {
57                              recalculatedUtilizedAmount = notYetUtilizedAmount;
58                        } else {
59                              recalculatedUtilizedAmount = TaxAmount.ZERO;
60                        }
61                  } else if (notYetExemptedYield.isGreaterThanZero()) {
62                        recalculatedUtilizedAmount = TaxAmount.min(notYetExemptedYield,
                              notYetUtilizedAmount);
63                  } else {
64                        recalculatedUtilizedAmount = TaxAmount.ZERO;
65                  }
66            } else {
67                  recalculatedUtilizedAmount = TaxAmount.ZERO;
68            }
69
70            setUtilizedAmount(incomeTypeToRecalculateFor,
                  taxOffsetBucketTypeToRecalculateFor, recalculatedUtilizedAmount);
71      }
72
```

```
73      private static boolean taxOffsetBucketCanBeUtilized(final TaxOffsetBucketTypeEnum
            taxExemptionBucketType, final Long incomeTypeId, final TaxBucketTypeDBE
            kindOfTaxBucket) {
74            Assert.isNotNull(kindOfTaxBucket, "kindOfTaxBucket is null");
75
76            final boolean doStockLossOnlyUseForStockGains =
                  !TaxOffsetBucketTypeEnum.VVTA.equalsEnum(taxExemptionBucketType) ||
77                InvestmentIncomeTypesEnum.STOCK_GAINS.equalsKey(incomeTypeId);
78
79            return
                  kindOfTaxBucket.taxOffsetBucketCanBeUtilizedForTaxBucketType(taxExemptionBucketType)
                  && doStockLossOnlyUseForStockGains;
80      }
81
82      // ...
83
84  }
```

Abbildung A.1.: Objekt-orientierter Codeausschnitt für das Fallbeispiel Berechnung

A.2. Fallbeispiel Berechnungslogik: Funktional

```
module Steuerberechnung =

   type Ertragsart = AD | ZI | VGS | ZIC | ID | IM | VGA

   type Spalte = list<decimal>
   type Tabelle = list<Spalte>

   // MaxInanspruchDurchSpalte -> Ertragsarten -> MaxInanspruchDurchErtragsart -> Spalte
   type Spaltenberechnung = decimal -> list<Ertragsart * decimal> -> Spalte

   ////////////////////////////
   // Spaltenberechnung
   ////////////////////////////

   let baueSpaltenberechnung zellenBerechnung maxDurchSpalte maxDurchErtragsarten =
      List.scan zellenBerechnung (maxDurchSpalte, 0m) maxDurchErtragsarten |> List.tail
          |> List.map snd

   let berechneSpalte : Spaltenberechnung =
      baueSpaltenberechnung (
         fun (durchSpalte, _) (_, durchErtragsart) ->
            let inanspruch = min durchSpalte durchErtragsart
            in (durchSpalte - inanspruch, inanspruch))

   let berechneVVTA : Spaltenberechnung =
      baueSpaltenberechnung (
         fun (durchSpalte, _) (ertragsart, durchErtragsart) ->
            let inanspruch = if ertragsart = Ertragsart.VGA
                        then min durchSpalte durchErtragsart
                        else 0.00m
            in (durchSpalte - inanspruch, inanspruch))

   ////////////////////////////
   // Tabellenberechnung
   ////////////////////////////

   let berechneTabelleSummenVorher ertragsarten bruttoertrag nettoertrag
       spaltenberechnungen =

      let rec berechneSpalten spaltenberechnungen inanspruchBisher =
         match spaltenberechnungen with
         | [] ->
            [nettoertrag]
         | (firstSpaltenberechnung::restSpaltenberechnungen) ->
            let spalte = firstSpaltenberechnung (List.zip ertragsarten inanspruchBisher)
            let newInanspruchBisher : Spalte = List.map2 (-) inanspruchBisher spalte
            in spalte :: berechneSpalten restSpaltenberechnungen newInanspruchBisher

      let inanspruchAnfang = (List.map2 (-) bruttoertrag nettoertrag)
```

```
48        in bruttoertrag :: berechneSpalten spaltenberechnungen inanspruchAnfang
49
50
51    let berechneTabelleSummenNachher ertragsarten (bruttoertrag : decimal list)
          spaltenberechnungen =
52
53        let rec berechneSpalten spaltenberechnungen inanspruchBisher =
54           match spaltenberechnungen with
55           | [] ->
56              [inanspruchBisher]
57           | (firstSpaltenberechnung::restSpaltenberechnungen) ->
58              let spalte = firstSpaltenberechnung (List.zip ertragsarten inanspruchBisher)
59              let newInanspruchBisher = List.map2 (-) inanspruchBisher spalte
60              in spalte :: berechneSpalten restSpaltenberechnungen newInanspruchBisher
61
62        in bruttoertrag :: berechneSpalten spaltenberechnungen bruttoertrag
63
64    let berechneTabelleBewegung berechneTabelleSummenNachher berechneTabelleSummenVorher =
65        List.map2 (List.map2 (-)) berechneTabelleSummenNachher berechneTabelleSummenVorher
66
67    let berechneTabellen ertragsarten nettoertragVorher bruttoertragVorher
          bruttoertragBewegung spaltenberechnungen =
68        let bruttoertragNachher = List.map2 (+) bruttoertragVorher bruttoertragBewegung
69        let tabelleSummenVorher = berechneTabelleSummenVorher ertragsarten
              bruttoertragVorher nettoertragVorher spaltenberechnungen
70        let tabelleSummenNachher = berechneTabelleSummenNachher ertragsarten
              bruttoertragNachher spaltenberechnungen
71        let tabelleBewegung = berechneTabelleBewegung tabelleSummenNachher
              tabelleSummenVorher
72        in (tabelleSummenVorher, tabelleBewegung, tabelleSummenNachher)
```

Abbildung A.2.: Funktionaler Codeausschnitt für das Fallbeispiel Berechnung

A.3. Fallbeispiel Berechnungslogik: Imperativ

```
1       *-----------------------------------------------------------------
2        BER-ZUG-ERTR-WIDAUFL SECTION.
3       *-----------------------------------------------------------------
4       *   WIEDERAUFLEBEN FSA/QUEST NUR IM VVT ...
5           EVALUATE TRUE
6             WHEN AGSBE-M-VVT (C)
7       *       ... BEI EVVR OHNE WID. NUR IM VVT-VORTRAG AKT. TOPF
8               IF (   NOT AGSBE-E-RUFER-EVZ
9                   AND NOT AGSBE-E-RUFER-EQZ
10                  AND NOT AGSBE-E-RUFER-EVA
11                  AND NOT AGSBE-E-RUFER-EQA)
12                OR HS-IM-VORTR-J
13                 PERFORM BER-RECHNE-WID
14              END-IF
15            WHEN AGSBE-M-VVT-SCHATTEN (C)
16            WHEN AGSBE-M-VVT-AUSL (C)
17            WHEN AGSBE-M-BETRIEBL (C)
18              NEXT SENTENCE
19            WHEN OTHER
20              PERFORM FATAL-NO-TOPF
21          END-EVALUATE.
22          EXIT.
23      *-----------------------------------------------------------------
24       BER-RECHNE-WID SECTION.
25      *-----------------------------------------------------------------
26      *   GIBT ES KEINE ZU VERSTEUERNDEN BETRAEGE MEHR
27          PERFORM ST-C-SUMMIEREN-ERTR-ARTEN.
28          SET HS-QST-WID-N         TO TRUE.
29          IF AGSBE-M-ESU-ERTRNDI-SW (C) = 0
30           AND AGSBE-M-ESU-ERTRNDM-SW (C) = 0
31      *     ES IST VVT DA
32            IF  AGSBE-M-ASSTA-VVT-SW (C) > 0
33      *       ES IST QUEST VERWENDET
34              IF AGSBE-M-ASSTA-AQSTI-SW(C) > 0
35                PERFORM BER-WID-QUEST
36              END-IF
37      *       ES IST FSA VERWENDET
38              IF AGSBE-M-ASSTA-FSAI-SW (C) > 0
39      *        BEI "EVVR VVT-ZUGG. WG. WID QUEST" KEIN WID FSA
40               AND NOT AGSBE-E-RUFER-EVW
41                 PERFORM BER-WID-FSA
42              END-IF
43            END-IF
44      *     ** QUELLENSTEUER IN ANSPRUCH GENOMMEN
45            IF AGSBE-M-ASSTA-AQSTI-SW(C) > 0
46              PERFORM BER-WID-QUEST-FSA
47            END-IF
48          END-IF.
49      *   TOEPFE NUR ERZEUGEN, WENN WIEDERAUFL.
```

```
50          IF AGSBE-A-FSA-WID-J (ABF)
51           OR AGSBE-A-FSA-WID-J (VBF)
52           OR HS-QST-WID-J
53      *      SVK RECHNEN
54             PERFORM BER-BUH-C-SVK
55      *      TOEPFE N. WID.
56             PERFORM BER-STORE-C-TOPF-W
57             PERFORM BER-ANTEIL-DW
58             PERFORM BER-LOAD-C-TOPF-W
59          END-IF.
60          EXIT.
61      *---------------------------------------------------------------
62       BER-WID-FSA SECTION.
63      *---------------------------------------------------------------
64      *   RECHNE VVT-GUTHABEN
65          PERFORM BER-VVT-GUTHABEN.
66      *   ES GIBT WAS ZU VERRECHNEN
67          IF AGSBE-A-VVT-GUTH <> 0
68      *      FSA INANSPR. VERRINGERN, VVT-INANSPR. ERHOEHEN
69             IF  AGSBE-M-ASSTA-FSAI-SW (C) >= AGSBE-A-VVT-GUTH
70      *        FSA-I.A. ABBAUEN U. VVT KOMPLETT BEANSPRUCHEN
71               SUBTRACT AGSBE-A-VVT-GUTH FROM AGSBE-M-ASSTA-FSAI-SW(C)
72               ADD    AGSBE-A-VVT-GUTH TO AGSBE-M-ASSTA-VVTI-SW(C)
73      *        NUN ALLES WIRD IM FSA BEHANDELT ...
74               IF HS-VORJAHR
75                 SUBTRACT AGSBE-A-VVT-GUTH
76                    FROM AGSBE-M-ASFBS-INANSPR-VJ (C)
77               ELSE
78                 SUBTRACT AGSBE-A-VVT-GUTH
79                    FROM AGSBE-M-ASFBS-INANSPR-LJ (C)
80               END-IF
81             ELSE
82      *        FSA INANSPR. KOMPLETT VERRECHNEN
83               IF HS-VORJAHR
84                 SUBTRACT AGSBE-M-ASSTA-FSAI-SW (C)
85                    FROM AGSBE-M-ASFBS-INANSPR-VJ (C)
86               ELSE
87                 SUBTRACT AGSBE-M-ASSTA-FSAI-SW (C)
88                    FROM AGSBE-M-ASFBS-INANSPR-LJ (C)
89               END-IF
90      *        FSA-I.A. KOMPLETT ABBAUEN U. VVT DAMIT BEANSPRUCHEN
91               ADD AGSBE-M-ASSTA-FSAI-SW (C)
92                                   TO AGSBE-M-ASSTA-VVTI-SW (C)
93               MOVE ZEROS          TO AGSBE-M-ASSTA-FSAI-SW (C)
94             END-IF
95      *      KENNZEICHNEN FSA IST WIEDERAUFGELEBT
96             SET AGSBE-M-FSA-WID-J (C) TO TRUE
97      *      GEAENDERTEN FSA SPEICHERN U. ANTEIL BERECHNEN
98             PERFORM BER-FSA-STORE-ANTEIL
99          END-IF.
100         EXIT.
```

```
101      *-----------------------------------------------------------------
102       BER-INANSPRN-VVT-AKTIEN SECTION.
103      *-----------------------------------------------------------------
104          EVALUATE TRUE
105            WHEN AGSBE-M-VVT (C)
106      * RECHNE INANSPR. VVT AKTIEN, FREISTELLG. ERFOLGT
107              PERFORM BER-FREIST-INAN-VVT-AKTIEN
108            WHEN AGSBE-M-VVT-SCHATTEN (C)
109            WHEN AGSBE-M-VVT-AUSL (C)
110            WHEN AGSBE-M-BETRIEBL (C)
111      * DERZEIT KEINE INANSPR. VVT AKTIEN
112              NEXT SENTENCE
113          END-EVALUATE.
114          EXIT.
115      *-----------------------------------------------------------------
116       BER-FREIST-INAN-VVT-AKTIEN SECTION.
117      *-----------------------------------------------------------------
118          IF AGSBE-M-ASSTE-AZXER-ART (C,MC) = 11
119
120      * VVTA und freizustellender Aktienertrag vorhanden?
121            COMPUTE HF-REFE-16-2 = FUNCTION MIN (
122              HF-FREIZUSTELL,
123              AGSBE-M-ASSTE-ERTRB-SW (C,MC),
124              AGSBE-A-VVT-AKTIEN-GUTH)
125            IF HF-REFE-16-2 > 0
126              SUBTRACT HF-REFE-16-2 FROM HF-FREIZUSTELL
127              SET MFF TO MC
128              ADD HF-REFE-16-2 TO AGSBE-M-FREIVVTAKTIEN (C,MFF)
129              ADD HF-REFE-16-2 TO AGSBE-M-ASSTA-VVTI-AKTIEN-SW (C)
130              ADD HF-REFE-16-2 TO AGSBE-M-ASSTE-FREI-SW (C,MC)
131            END-IF
132          END-IF.
133          EXIT.
134      *-----------------------------------------------------------------
135       BER-DO-VVR-RECHNEN SECTION.
136      *-----------------------------------------------------------------
137      * WIR SIND Z.ZT. IN DER VVR
138          SET HS-BER-WO-VVR TO TRUE.
139      * JEDE ERTRAGSART IM TOPF PRUEFEN ...
140          SET MEI TO 1.
141          PERFORM VARYING MEI FROM 1 BY 1
142           UNTIL MEI > ETA-MAX
143           OR ERTRARTEN-ENDE (MEI)
144      *
145         SET MC TO MEI
146         SET QST TO MEI
147      * ... WENN NOCH NETTOERTRAG VORH. U. FREISTELL-GUTHABEN ...
148         PERFORM BER-FREISTELL-GUTHABEN
149         IF ( AGSBE-M-ASSTE-ERTRNDI-SW (C,MC) > 0
150          OR AGSBE-M-ASSTE-ERTRNDM-SW (C,MC) > 0)
151          AND ( HF-QUEST-FSA-VVT-GUTH > 0
```

```
152          OR ( AGSBE-M-ASSTE-AZXER-ART (C,MC) = 11
153          AND AGSBE-M-ASSTA-VVTI-AKTIEN-SW (C) <
154          AGSBE-M-ASSTA-VVT-AKTIEN-SW (C)))
155            PERFORM BER-INIT-BETR-ERTRAGSART
156       * ... VVR FUER JEWEILIGE ERTRAGSART
157            PERFORM BER-TOPFABH-VVR
158          END-IF
159          END-PERFORM.
160       * SVK RECHNEN
161          PERFORM BER-BUH-C-SVK.
162          EXIT.
163       *---------------------------------------------------------------
164        BER-TOPFABH-VVR SECTION.
165       *---------------------------------------------------------------
166       * UNTERSCHEIDUNG NACH TOPF-ARTEN, DIE THEORET. AUFRUFBAR SIND
167          EVALUATE TRUE
168            WHEN AGSBE-M-VVT (C)
169       * ES GIBT NOCH QUEST/VVT/FSA
170              IF AGSBE-M-ASSTA-AQSTI-SW (C) < AGSBE-M-ASSTA-AQST-SW(C)
171               OR AGSBE-M-ASSTA-VVTI-SW (C) < AGSBE-M-ASSTA-VVT-SW (C)
172               OR ( HS-VORJAHR
173               AND AGSBE-M-ASFBS-INANSPR-VJ (C)
174               < AGSBE-M-ASFBS-FBETR-VJ(C))
175               OR (NOT HS-VORJAHR
176               AND AGSBE-M-ASFBS-INANSPR-LJ (C)
177               < AGSBE-M-ASFBS-FBETR (C))
178               OR ( AGSBE-M-ASSTE-AZXER-ART (C,MC) = 11
179               AND AGSBE-M-ASSTA-VVTI-AKTIEN-SW (C) <
180               AGSBE-M-ASSTA-VVT-AKTIEN-SW (C))
181       * DIESE ERTRAGSART STEUER-BERECHNEN
182                 PERFORM BER-STEUER-EINE-ERTRART
183              END-IF
184            WHEN OTHER
185              NEXT SENTENCE
186          END-EVALUATE.
187          EXIT.
```

Abbildung A.3.: Imperativer Codeausschnitt für das Fallbeispiel Berechnung

A.4. Fallbeispiel Entscheidungslogik: Objekt-orientiert

```
1  public class BookEarningDistributionDispositionOrderTBE extends
       AbstractBookDispositionOrderTBE {
2
3        // ...
4
5        private static boolean createDispositionOrderReinvest(
6            @Modified final BookEarningDistributionDispositionOrderTBE dispositionOrder,
7            final Date fundPriceDate,
8            final SubDepotDBE subDepotForReinvestment,
9            final SubDepotDBE subDepotOfEarningsDistribution,
10           final AssetDBE earningsDistributionAsset,
11           final MoneyValueScaled dispositionAmountInPaymentCurrency,
12           final MoneyValueScaled dispositionAmountInDefaultCurrency,
13           final DeterminedMoneyCircuitAccountBBE determinedMoneyCircuitAccount,
14           final PeriodicPaymentDBE periodicPayment) throws BookingOrderCreationException {
15
16            if (addDecisionInformationForZeroDisposition(dispositionOrder,
                  dispositionAmountInPaymentCurrency)) {
17                return true;
18            }
19
20            final String decisionInformationInfix = periodicPayment != null ? " nach
                  Kundenweisung" : "";
21
22            // either subDepot has an asset and asset currency is equal to payment
                  currency -> reinvest with same currency is possible
23            // or subDepot is a risk class and only if it is the same subDepot of the
                  earning distribution (periodicPayment == null) we know that the asset
24            // exists in the risk class subDepot of the reinvest (compare payment
                  currency with asset currency of earning distribution asset)
25            if ((subDepotForReinvestment.hasAsset() &&
                  EqualUtils.areEqual(dispositionAmountInPaymentCurrency.getCurrency(),
                  subDepotForReinvestment.getAsset().getShareIdentifier().getCurrency()))
26                || (subDepotForReinvestment.hasRiskClass() && periodicPayment == null &&
                      EqualUtils.areEqual(dispositionAmountInPaymentCurrency.getCurrency(),
                      earningsDistributionAsset.getShareIdentifier().getCurrency()))) {
27                if (createDispositionOrderReinvest(dispositionOrder,
                      decisionInformationInfix, fundPriceDate, subDepotForReinvestment,
                      subDepotOfEarningsDistribution, earningsDistributionAsset,
28                    dispositionAmountInPaymentCurrency, determinedMoneyCircuitAccount,
                          periodicPayment, false /* useWholeRiskClass */)) {
29                    return true;
30                }
31                // if creation of an order for a subDepot with asset failed every other
                      currency will also fail
32                if (subDepotForReinvestment.hasAsset()) {
33                    return false;
34                }
35            } else {
```

```
36                  // if it is a reinvest in the same subDepot of a risk class subDepot
                        try the reinvest for the same asset in the subDepot in default
                        currency
37                  if (subDepotForReinvestment.hasRiskClass() && periodicPayment == null) {
38                        if (createDispositionOrderReinvest(dispositionOrder,
                              decisionInformationInfix, fundPriceDate,
                              subDepotForReinvestment, subDepotOfEarningsDistribution,
                              earningsDistributionAsset,
39                            dispositionAmountInPaymentCurrency,
                                  determinedMoneyCircuitAccount, periodicPayment, false /*
                                  useWholeRiskClass */)) {
40                              return true;
41                        }
42                  }
43            }
44            // disposition order for reinvestment in default currency in asset or whole
                  risk class
45            if (createDispositionOrderReinvest(dispositionOrder,
                  decisionInformationInfix, fundPriceDate, subDepotForReinvestment,
                  subDepotOfEarningsDistribution, earningsDistributionAsset,
46                dispositionAmountInDefaultCurrency, determinedMoneyCircuitAccount,
                      periodicPayment, true /* useWholeRiskClass */)) {
47                return true;
48            }
49            return false;
50      }
51
52      // ...
53
54 }
```

Abbildung A.4.: Objekt-orientierter Codeausschnitt für das Fallbeispiel Entscheidung

A.5. Fallbeispiel Entscheidungslogik: Funktional

```
type Ergebnis = Erfolgreich | Fehlerhaft
type Waehrung = EUR | GBP | USD // |...

type Kundendaten = { FP : bool // Fondsportfolio
                   ; SW : Waehrung // Standardwaehrung
                   ; ZW : Waehrung // Zwischenwaehrung
                   ; VW : Waehrung // Vermoegenswaehrung
                   ; KuW : bool // Kundenweisung
                   ; EAVW : Waehrung // Vermoegenswaehrung EA Fonds
                   ; VG : decimal // Anteil Vermoegenswaehrung in Fondsportfolio
                   }


// Dummyfunktion zur Erstellung von Verwendungsauftraegen
let erstelleVerwendungsauftrag waehrung = Erfolgreich

// Entscheidungslogik
let versucheWiederanlage (kundendaten : Kundendaten) =

   let versucheVerwendungsauftragInSWfuerFP =
      erstelleVerwendungsauftrag kundendaten.ZW

   let versucheVerwendungsauftragInSWfuerVG =
      if erstelleVerwendungsauftrag kundendaten.SW = Erfolgreich
            then Erfolgreich
         else versucheVerwendungsauftragInSWfuerFP

   let pruefeZW =
      if kundendaten.FP && not kundendaten.KuW && kundendaten.VG > 0m
         then versucheVerwendungsauftragInSWfuerVG
         else versucheVerwendungsauftragInSWfuerFP

   let pruefeFP =
      if kundendaten.FP
         then versucheVerwendungsauftragInSWfuerFP
         else Fehlerhaft

   let versucheVerwendungsauftragInZW =
      if erstelleVerwendungsauftrag kundendaten.ZW = Erfolgreich
         then Erfolgreich
         else pruefeFP

   if (not kundendaten.FP && kundendaten.ZW = kundendaten.VW) || (kundendaten.FP && not
       kundendaten.KuW && kundendaten.ZW = kundendaten.EAVW && kundendaten.VG > 0m)
      then versucheVerwendungsauftragInZW
      else pruefeZW
```

Abbildung A.5.: Funktionaler Codeausschnitt für das Fallbeispiel Entscheidung

Literaturverzeichnis

[BW08] R. P. L. Buse, W. Weimer. „A metric for software readability“. In: *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis*. 2008, S. 121–130 (zitiert auf S. 10).

[Chu41] A. Church. *The Calculi of Lambda-Conversion*. Princeton University Press, 1941 (zitiert auf S. 15).

[Cou14] S. Cousins. *Does the Language You Use Make a Difference (revisited)?* 2014. URL: http://simontylercousins.net/does-the-language-you-use-make-a-difference-revisited/ (zitiert auf S. 19).

[Dmi04] S. Dmitriev. *Language Oriented Programming: The Next Programming Paradigm*. 2004. URL: http://www.onboard.jetbrains.com/is1/articles/04/10/lop/ (zitiert auf S. 18).

[Erl00] L. Erlikh. „Leveraging Legacy System Dollars for E-Business“. In: *IT Professional* 2.3 (2000), S. 17–23 (zitiert auf S. 10).

[FFBS04] E. Freeman, E. Freeman, B. Bates, K. Sierra. *Head First Design Patterns*. O' Reilly & Associates, Inc., 2004 (zitiert auf S. 10).

[Fow05] M. Fowler. *Language Workbenches: The Killer-App for Domain Specific Languages?* 2005. URL: http://www.martinfowler.com/articles/languageWorkbench.html (zitiert auf S. 18).

[Han10] S. Hanenberg. „An Experiment About Static and Dynamic Type Systems: Doubts About the Positive Impact of Static Type Systems on Development Time“. In: *SIGPLAN Not.* 45.10 (2010), S. 22–35 (zitiert auf S. 29).

[JB12] C. Jones, O. Bonsignour. *The Economics of Software Quality*. Addison-Wesley, 2012 (zitiert auf S. 10, 29).

[Jon12] C. Jones. *A Short History of the Lines of Code Metric*. 2012 (zitiert auf S. 20).

[Jon13] C. Jones. „Function points as a universal software metric“. In: *ACM SIGSOFT Software Engineering Notes* 38.4 (2013), S. 1–27 (zitiert auf S. 20).

[KB07] K. Krogmann, S. Becker. „A Case Study on Model-Driven and Conventional Software Development: The palladio editor“. In: *Software Engineering 2007 - Beiträge zu den Workshops, Fachtagung des GI-Fachbereichs Softwaretechnik, 27.-30.3.2007 in Hamburg*. 2007, S. 169–176 (zitiert auf S. 30).

[Ker04] J. Kerievsky. *Refactoring to Patterns*. Pearson Higher Education, 2004 (zitiert auf S. 10).

[Khr06] Y. Khramov. „The Cost of Code Quality“. In: *AGILE 2006 Conference*. 2006, S. 119–125 (zitiert auf S. 29).

[KJB+09] T. Kapteijns, S. Jansen, S. Brinkkemper, H. Houet, R. Barendse. „A Comparative Case Study of Model Driven Development vs Traditional Development: The Tortoise or the Hare“. In: *4th European Workshop on "From code centric to model centric software engineering: Practices, Implications and ROI"*. 2009, S. 22–33 (zitiert auf S. 30).

[Mar08] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1. Aufl. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008 (zitiert auf S. 10, 11).

[McL12] J. McLoone. *Code Length Measured in 14 Languages*. 2012. URL: http://blog.wolfram.com/2012/11/14/code-length-measured-in-14-languages/ (zitiert auf S. 19).

[Moa90] J. Moad. „Maintaining the competitive edge“. In: 36.4 (1990), S. 61–62, 64, 66 (zitiert auf S. 10).

[NF15] S. Nanz, C. A. Furia. „A Comparative Study of Programming Languages in Rosetta Code“. In: *37th IEEE/ACM International Conference on Software Engineering*. 2015, S. 778–788 (zitiert auf S. 19, 20, 29).

[PSG12] V. Pankratius, F. Schmidt, G. Garreton. „Combining functional and imperative programming for multicore software: An empirical study evaluating Scala and Java“. In: *34th International Conference on Software Engineering*. 2012, S. 123–133 (zitiert auf S. 19, 29).

[RPFD14] B. Ray, D. Posnett, V. Filkov, P. T. Devanbu. „A large scale study of programming languages and code quality in github“. In: *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2014, S. 155–165 (zitiert auf S. 20, 29).

[Seb12] R. W. Sebesta. *Concepts of Programming Languages*. 10th. Pearson, 2012 (zitiert auf S. 13).

[TOAY13] Y. Tashtoush, Z. Odat, I. Alsmadi, M. Yatim. „Impact of Programming Features on Code Readability“. In: Bd. 7. 6. 2013, S. 441–458 (zitiert auf S. 10, 24, 27, 29).

[Van09] P. Van Roy. „Programming Paradigms for Dummies: What Every Programmer Should Know“. In: 2009 (zitiert auf S. 13).

[War94] M. P. Ward. „Language-Oriented Programming“. In: *Software - Concepts and Tools* 15.4 (1994), S. 147–161 (zitiert auf S. 17).

Alle URLs wurden zuletzt am 02. 09. 2016 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift