ChatGPT

Best Practice Guide: Power Query M – Code-Struktur und Formatierung

Einführung

In Business-Intelligence-Projekten mit Power BI Desktop und Dataflows ist sauber strukturierter **Power Query M** Code essenziell. Dieser Guide stellt Best Practices für **Code-Struktur und Formatierung** vor – speziell ausgerichtet auf BI-Entwickler. Wir orientieren uns an bewährten Clean-Code-Konzepten aus Python, C#, SQL und JavaScript, um **lesbaren, wartbaren und performanten** M-Code zu erreichen. Herzstück ist ein benutzerdefiniertes **Namensschema** in deutscher Sprache: **„Aktion + *Was* + *Nach Kriterium*“**, das konsistente, selbstdokumentierende Schrittnamen fördert. Im Folgenden behandeln wir die Schwerpunkte:

1. **Namenskonventionen und Bezeichnungsstandards** – Konsistente PascalCase-Namen auf Deutsch nach dem Schema *„AktionWasNachKriterium“*.
2. **Code-Formatierung und visuelle Struktur** – Einrückungen, Zeilenumbrüche, Gruppierungen und Kommentare für optimale Lesbarkeit.
3. **Modularität und Wiederverwendbarkeit** – Nutzung von Custom Functions, Parametern und Mustern zur Vermeidung von Redundanz.
4. **Performance-Optimierung** – Einfluss der Struktur auf Query Folding und der Ausgleich von Namenslänge vs. Verständlichkeit.
5. **Tooling und Praxis** – Umsetzung im Team, Migrationsstrategien und Qualitätssicherung.

(Hinweis: Zitate von Quellen sind in eckigen Klammern mit 【Quelle†Zeilen】 angegeben und dienen der Verifizierung der Aussagen.)

1. Namenskonventionen und Bezeichnungsstandards

Aussagekräftige Bezeichnungen sind der Grundpfeiler von Clean Code und gelten ebenso in Power Query M. Entwickler sollten **aussagekräftige Namen für Variablen und Schritte verwenden**, damit beim Lesen des Codes sofort klar wird, was eine Funktion macht oder wofür eine Variable steht [1] [2]. In Power Query erstellt die GUI zwar automatisch generische Schrittbezeichnungen (wie *„Geänderter Typ1“* oder *„Gefilterte Zeilen“*), doch diese Standardnamen liefern kaum Kontext zu **Was** gefiltert wurde, **wo** und **warum** [3]. **Vermeiden Sie die Default-Namen** und investieren Sie die Zeit, jedem Schritt einen klaren, verständlichen Namen zu geben [4] – das erleichtert Ihnen und anderen das Verständnis des ETL-Prozesses deutlich [5] [6].

PascalCase ohne Leerzeichen: Wir empfehlen, alle Schritt- und Variablennamen in *deutscher PascalCase-Schreibweise* zu vergeben, d.h. **zusammengesetzte Worte mit jedem Wort beginnend in Großbuchstaben und ohne Leer- oder Sonderzeichen**. M ist case-sensitiv und „mag keine Leerzeichen“ in Bezeichnern [7] – Leerzeichen zwingen zu unschöner Syntax wie `#"Schritt Name"` und erschweren Referenzen. Mit zusammenhängenden PascalCase-Namen umgehen wir dieses Problem. (PascalCase bedeutet, ähnlich wie in C#, alle Anfangsbuchstaben groß zu schreiben, inkl. des ersten Wortes [8].) Beispielsweise wird aus `"Gefilterte Zeilen"` ein Schritt `GefilterteZeilen` oder besser noch ein beschreibenderer Name gemäß unserem Schema (siehe unten). Solche konsistenten Namenskonventionen lohnen sich, da **Power Query case-sensitiv ist** – ein einmal

festgelegter Name muss exakt getroffen werden – und ein einheitlicher Stil Fehler vermeidet [9]. Wichtig ist außerdem, *keine unverständlichen Abkürzungen* in Namen zu verwenden. Schreiben Sie lieber vollständige deutsche Wörter in natürlicher Sprache [10]. Eine Ausnahme können etablierte Abkürzungen wie *ID* oder *Nr.* sein – vermeiden Sie jedoch kryptische Kürzel, die nur Ihnen bekannt sind [11] [12]. **Bevorzugen Sie Klarheit vor Kürze** [13]: Ein längerer, aber selbsterklärender Name ist besser als ein kurzer, rätselhafter. („Der Code wird häufiger gelesen als geschrieben – *Readability counts* [14].“)

Schema „Aktion + Was + Nach Kriterium“: Für Schritte hat es sich bewährt, einem **dreiteiligen Schema** zu folgen, das **die durchgeführte Aktion**, das **Objekt** und ggf. das **Kriterium** der Aktion im Namen vereint. Dieses Schema liefert sofort Kontext, *was* der Schritt tut und *worauf* er sich bezieht, ganz ohne Kommentar [15]. Im Deutschen ergibt sich eine Konstruktion aus Verb + Objekt + Bedingung. Hier sind **Vorlagen für gängige M-Transformationen**:

- **Filtern**: `Filtern<Objekt>Nach<Kriterium>` – Beispiel: `FilternKundenNachLand` für einen Schritt, der Kunden nach Land filtert (entspricht *Filter Customers by Country*). Ein weiterer Beispielname wäre `FilternZeilenNachStatus`, um Zeilen anhand eines Statuswerts auszuwählen.
- **Gruppieren**: `Gruppieren<Objekt>Nach<Kriterium>` – Beispiel: `GruppierenUmsaetzeNachJahr` gruppiert Umsatzzahlen nach Jahr. (Alternative: `GruppierenBestellungenNachKunde` – Bestellungen nach Kunden gruppieren.)
- **Berechnen**: `Berechnen<Was>Nach<Kriterium>` – Beispiel: `BerechnenUmsatzProKategorie` berechnet einen Umsatzwert **pro** Kategorie. (Hier kann je nach Kontext auch *„Pro“* oder *„Je“* anstelle von *„Nach“* sinnvoll sein, z.B. `BerechneDurchschnittJeProdukt`.) Die Idee ist klarzumachen, welcher neue Wert oder KPI ermittelt wird.
- **Hinzufügen (neue Spalte)**: `Hinzufügen<Was>` oder `Berechnen<Was>` – Beispiel: `HinzufügenIndexSpalte` fügt eine Index-Spalte hinzu; `BerechnenGesamtpreisAusMenge` fügt z.B. einen Gesamtpreis basierend auf Menge und Einzelpreis hinzu.
- **Entfernen (Datensätze/Spalten)**: `Entfernen<Was>` – Beispiel: `EntfernenLeereZeilen` für das Herausfiltern leerer Datensätze oder `EntfernenHilfsspalten` für das Entfernen nicht benötigter Spalten.
- **Ersetzen (Werte)**: `Ersetzen<Was>Durch<NeuerWert>` – Beispiel: `ErsetzenNullDurch0` ersetzt Nullwerte durch 0, oder `ErsetzenTextDurchMarker` ersetzt etwa bestimmte Texte durch einen Marker.
- **Aufteilen (Split)**: `Aufteilen<Was>Nach<Kriterium>` – Beispiel: `AufteilenNameNachLeerzeichen` teilt einen Namen an Leerzeichen auf (entspricht *Split Name by Space*).
- **Joinen (Zusammenführen)**: `Joinen<X>Mit<Y>Nach<Schlüssel>` – Beispiel: `JoinenBestellungenMitKundenNachKundenID` für einen Merge (Join) der Bestellungen mit Kundendaten anhand der KundenID. Hier beschreibt *Aktion* = Joinen (Zusammenführen zweier Quellen), *Was* = Bestellungen mit Kunden, *Nach Kriterium* = KundenID als Join-Schlüssel. (Synonyme wie `Zusammenführen` oder `Verbinden` sind ebenfalls möglich; wichtig ist, das Muster beizubehalten und beide Seiten des Joins sowie den Key zu nennen.)
- **Pivotieren** (Pivot/Unpivot): `Pivotieren<Objekt>Nach<Attribut>` – Beispiel: `PivotierenUmsaetzeNachRegion` könnte Umsatzzahlen nach Region als Spalten pivotieren. Umgekehrt könnte man einen Unpivot-Schritt benennen als `EntpivotierenMonateNachWert` (auch wenn „Entpivotieren“ kein offizielles Wort ist, wäre es im Kontext verständlich). Alternativ spricht man von *„auflösen“*: `AuflösenSpaltenNachMonat`. Wichtig ist auch hier: das Schema betonen – was wird wie umgeformt.

- **Sortieren**: `Sortieren<Objekt>Nach<Kriterium>` – Beispiel: `SortierenProdukteNachPreis` sortiert eine Produktliste nach dem Preis. Wenn sortiert wird, gehört das Sortierkriterium in den Namen. Bei Mehrfachsortierung könnte man mehrere Kriterien nennen oder allgemein bleiben (`SortierenProdukteNachKategorieUndPreis`).

Diese Vorlagen decken die häufigsten Schritte ab (*Filtern, Gruppieren, Berechnen/Hinzufügen, Entfernen, Ersetzen, Aufteilen, Joinen, Pivotieren, Sortieren*). Sie können das Schema sinngemäß auf andere Aktionen übertragen. Entscheidend ist die **Konsistenz**: Verwenden Sie für gleiche Aktionen immer das gleiche Verb und Wortwahl [16] . Zum Beispiel, wenn Sie einmal *„Filtern“* verwenden, nutzen Sie nicht an anderer Stelle *„Selektieren“* für denselben Vorgang. Einheitliche Verben für gleiche Anwendungsfälle verhindern Verwirrung [16] . Analog sollten unterschiedliche Aktionen klar durch unterschiedliche Verben benannt werden (z.B. nicht einmal *„HinzufügenSpalte“* und anderswo *„ErstellenSpalte“* für dasselbe Konzept). Einheitliche Benennung steigert die **Eindeutigkeit** und Lesbarkeit im gesamten Projekt [2] .

Komplexe Mehrschritt-Prozesse: Bei umfangreicheren Transformationen, die sich über mehrere Schritte erstrecken, sollte das Namensschema dazu genutzt werden, den **roten Faden** erkennbar zu machen. Dabei können verwandte Schritte einen gemeinsamen Kontext im Namen tragen. Beispiel: Ein mehrstufiger Prozess zur Kundenklassifizierung könnte Schritte haben wie `FilternKundenNachAktivität` → `BerechnenUmsatzJeKunde` → `KlassifizierenKundenNachUmsatz`. Hier sieht man sofort, dass alle Schritte zum Thema *Kunden und Umsatz* gehören und aufeinander aufbauen. Ein anderes Beispiel: Angenommen, man ermittelt „Churn“-Kunden (abgewanderte Kunden) in mehreren Etappen – man könnte die Schritte benennen als `FilternKundenNachInaktivität`, `MarkierenKundenAlsChurn`, `AnreichernKundenMitChurnFlag`. Alle Schritte enthalten *Kunden* und ergeben zusammen einen Prozess. **Vermeiden Sie es**, Schritte lediglich durchnummeriert oder mit nichtssagenden Labels zu versehen (z.B. *„Schritt1“*, *„Schritt2“* oder *„Temp“*), nur weil es Hilfsschritte sind. Auch Hilfsschritte können einen beschreibenden Namen tragen, der ihren Zweck erklärt (etwa `BerechnenZwischenwert` statt `TempCalc`). Falls notwendig, nutzen Sie Kommentare, um komplexe Prozesse in Abschnitte zu gliedern – aber die Schrittnamen selbst sollten bereits möglichst selbsterklärend sein. Optimal gewählte *„sprechende“* Namen machen Kommentare oft überflüssig [1] .

Namenskonvention für Variablen und Funktionen: Neben den Abfrage-Schritten selbst sollten auch alle benannten Variablen, Parameter und benutzerdefinierten Funktionen dem Stil folgen. Verwenden Sie *PascalCase* für Parameternamen und vermeiden Sie Leerzeichen oder Sonderzeichen dort ebenso [17] . Einige Entwickler nutzen einen Präfix (`p` für Parameter, `fn` für Functions), um diese deutlich zu kennzeichnen [17] [18] – z.B. `pPfad` für einen Parameter, `fnBerechneAlter` für eine Custom Function. Dies ist Geschmackssache; wichtiger ist, dass der Name klar den Inhalt beschreibt. Beispielsweise wäre `ParameterDatum` oder `StichtagDatum` verständlicher als nur `Datum` oder `x`. Halten Sie auch hier die Sprache konsistent (nicht einen Parameter Englisch `StartDate` und den nächsten Deutsch `EndeDatum`). Insgesamt gilt: **einmal etablierte Namenskonventionen sollten von allen Teammitgliedern konsequent eingehalten werden** [19] . Benennen Sie ähnliche Dinge immer nach dem gleichen Muster in *allen* Abfragen und Dokumentationen – so können Teammitglieder eure M-Skripte schneller „parsen“ und verstehen, da sie sich an die Muster gewöhnen [19] .

Zusammengefasst sorgt eine klare Benennung dafür, dass jeder Schritt sofort erkennen lässt, *was* er tut und *worauf* er sich bezieht. Dies erhöht die *Lesbarkeit und Verständlichkeit* des M-Codes enorm [2] . Es fördert auch die Selbst-Dokumentation: Ein Entwickler kann oft allein aus der Schrittfolge lesen, welche Transformationen durchgeführt wurden, ohne jeden Schritt im Detail untersuchen zu müssen [6] . So werden **“WTF pro Minute”** beim Code-Lesen minimiert [20] .

2. Code-Formatierung und visuelle Struktur

Neben den richtigen Namen spielt die **visuelle Strukturierung** des Codes eine große Rolle für die Lesbarkeit. Ein einheitliches Format mit sinnvollen Einrückungen, Zeilenumbrüchen und Abschnitten erleichtert es, auch komplexe Abfragen zu erfassen. Clean Code Prinzipien betonen die Bedeutung einheitlicher Formatierung – passende **Einrückungen, Leerzeilen und Blockstruktur** machen Code lesbarer [21]. In Python wird dies etwa strikt erzwungen, und auch in M sollten wir darauf achten, den Code logisch zu strukturieren.

Einrückung und Zeilenumbrüche: Nutzen Sie konsistente Einrückungen, um die **Hierarchie** im M-Code deutlich zu machen. Innerhalb eines `let … in` Blocks ist es üblich, jede Zuweisung um ein Level einzurücken. Beispielsweise:

```
let
    QuelleUmsaetze = Excel.Workbook(File.Contents("Sales2024.xlsx"), null,
true),
    FilternUmsaetzeNach2020 = Table.SelectRows(QuelleUmsaetze, each [Jahr] >=
2020),
    JoinUmsaetzeMitKundenNachKundenID = Table.NestedJoin(
        FilternUmsaetzeNach2020, {"KundenID"},
        Kundenstamm, {"KundenID"},
        "KundenDaten", JoinKind.LeftOuter
    ),
    ExpandKundenDaten = Table.ExpandTableColumn(
        JoinUmsaetzeMitKundenNachKundenID, "KundenDaten",
        {"Name", "Ort"}, {"KundenName", "KundenOrt"}
    ),
    GruppierenUmsaetzeNachOrt = Table.Group(
        ExpandKundenDaten, {"KundenOrt"},
        {{"UmsatzSumme", each List.Sum([Umsatz]), type number}}
    ),
    SortierenOrteNachUmsatz = Table.Sort(
        GruppierenUmsaetzeNachOrt, {{"UmsatzSumme", Order.Descending}}
    )
in
    SortierenOrteNachUmsatz
```

Wie man sieht, steht jede Schritt-Zuweisung in einer eigenen Zeile und ist eingerückt. Lange Funktionsaufrufe werden übersichtlich auf mehrere Zeilen aufgeteilt (öffnende Klammer in gleicher Zeile, Parameter eingerückt, schließende Klammer in neuer Zeile), sodass sie nicht horizontal „explodieren“. Diese **Segmentierung durch Zeilenumbrüche** verhindert zu dichte Textblöcke und unterstützt das schnelle Verständnis (*„Sparse is better than dense. Readability counts.“* [14]). Eine Zeile sollte möglichst nicht übermäßig lang sein (eine grobe Orientierung sind 80–120 Zeichen, analog zu Konventionen in vielen Sprachen), damit man nicht horizontal scrollen muss oder im Editor die Übersicht verliert. Brechen Sie bei Bedarf komplexe Ausdrücke um – z.B. jeden Funktionsparameter in eine eigene Zeile, oder filtern Sie in zwei Schritten anstatt eine extrem lange Bedingung in `Table.SelectRows` zu packen.

Logische Blockbildung: Gruppieren Sie zusammengehörige Schritte visuell, um die Struktur der Transformation hervorzuheben. Das Einfügen **leerer Zeilen** oder Kommentare als Trenner kann helfen, Abschnitte zu bilden – z.B. Quelleneinbindung, dann Datenbereinigung, dann Aggregation. Im obigen Beispiel könnte man vor dem `Gruppieren...` einen Leerraum einfügen oder einen Kommentar `// Aggregation` einleiten, um zu signalisieren, dass ab hier die Aggregationslogik beginnt. Ein Beispiel mit Kommentaren:

```
let
    // *** Datenquelle laden ***
    QuelleUmsaetze = Excel.Workbook(File.Contents("Sales2024.xlsx"), null,
true),
    Kundenstamm   = Sql.Database("SQLServer01", "CRM", [Query="SELECT * FROM
dbo.Kunden"]),

    // *** Transformationen auf Umsätze ***
    GefilterteUmsaetzeNach2020 = Table.SelectRows(QuelleUmsaetze, each [Jahr]
>= 2020),
    UmsatzMitKunden = Table.NestedJoin(GefilterteUmsaetzeNach2020,
{"KundenID"}, Kundenstamm, {"KundenID"}, "KundenDaten", JoinKind.LeftOuter),
    ErweiterteUmsaetze = Table.ExpandTableColumn(UmsatzMitKunden,
"KundenDaten", {"Name","Ort"}, {"KundenName","KundenOrt"}),

    // *** Aggregation nach Ort ***
    Umsatzzusammenfassung = Table.Group(ErweiterteUmsaetze, {"KundenOrt"},
{{"UmsatzSumme", each List.Sum([Umsatz]), type number}}),
    SortierteZusammenfassung = Table.Sort(Umsatzzusammenfassung,
{{"UmsatzSumme", Order.Descending}})
in
    SortierteZusammenfassung
```

Durch die Kommentare (**** Abschnitte) und Leerzeilen wird deutlich, welcher Block welche Aufgabe hat. Diese *visuelle Gliederung*** unterstützt andere Leser dabei, den Code zu scannen und die Struktur zu begreifen, ohne jeden Schritt lesen zu müssen. Achten Sie darauf, Kommentare und Leerzeilen sparsam aber gezielt zu nutzen – sie sollten echte Abschnittsinformationen vermitteln und nicht willkürlich verstreut sein.

Kommentare und Dokumentation: Obwohl gut gewählte Schrittnamen viele Details bereits verraten, sind **Kommentare ein wichtiges Instrument**, um zusätzliche Infos oder rationale Hintergründe festzuhalten. In M können Sie Kommentare mit `//` für einzelne Zeilen oder `/* ... */` für ganze Blöcke einfügen. Nutzen Sie Kommentare, um *warum* ein bestimmter Schritt nötig ist oder *was* eine komplexe Ausdruckslogik bedeutet. Beispielsweise könnte man über einen komplizierten Filter schreiben: `// Filtert Kunden, die im letzten Jahr keine Transaktion hatten (Churn-Kandidaten)`. So ein Kommentar liefert den fachlichen Hintergrund, den der Code allein nicht zeigt. Wie jede Sprache ermöglicht M auch das Einfügen von **Schrittbeschreibungen** im Query-Editor (das kleine „i“-Symbol neben einem Schritt). Hier können Sie in der Power Query-Oberfläche eine Beschreibung hinterlegen, die bei Mouseover sichtbar wird [22] – praktisch, um ausführlichere Dokumentation zu verstauen, ohne den Code mit langen Kommentarblöcken zu überfrachten.

Beachten Sie jedoch die Clean-Code-Maxime: Kommentare *sollten gut überlegt und sparsam eingesetzt werden* [23] . Sie ersetzen nicht den Code selbst, sondern ergänzen ihn. Wenn Sie feststellen, dass Sie viel

kommentieren müssen, könnte das ein Zeichen sein, dass die Schrittnamen oder Struktur nicht selbsterklärend genug sind. Im Idealfall machen **„sprechende Namen Kommentare überflüssig“** [24] – d.h. der Code ist so klar, dass nur noch bei wirklich komplexen Sachverhalten ein erläuternder Kommentar nötig ist. Schreiben Sie keine Kommentare, die nur das Offensichtliche wiederholen (*„// Filtere Tabelle nach Land = 'DE'“* bei einem Schritt `FilternKundenNachLand` wäre redundant). Kommentieren Sie lieber Geschäftsregeln, Annahmen oder knifflige Workarounds, die man nicht sofort sieht. Ein Grundsatz lautet: *„Kommentiere den Code so, dass er Zweck und Funktionalität erklärt“*, aber kommentieren Sie nicht jeden Furz – der Kommentar sollte Mehrwert bieten [23].

Konsistente Formatierung im Team: Legen Sie teamweit fest, wie formatiert wird – z.B. **Einrückung mit 4 Leerzeichen** (oder 2, Hauptsache alle nutzen das Gleiche), **Schritt pro Zeile**, maximal **eine Zuweisung pro Zeile**, etc. Einheitlichkeit hilft, dass jeder sich in jedermanns Code zurechtfindet [19]. Wie in anderen Sprachen (etwa bei JavaScript mit Prettier oder Python mit PEP8) sollte es auch für M einen gemeinsamen Stil geben. Derzeit bietet der Power BI Editor keine eingebaute Autoformatierung, aber es gibt Tools: **Power Query Formatter** (von PowerPivot-Insights) formatiert M-Code automatisch gemäß festgelegter Konventionen [25]. Man kann den Code z.B. in Visual Studio Code mit dem *Power Query*-Plugin bearbeiten oder in externen Editoren wie Notepad++ manuell formatieren [26] [27]. Einige nutzen Tabular Editor mit einem C#-Script, um M-Abfragen zu formatieren [28]. Nutzen Sie solche Werkzeuge im Team, um einen einheitlichen Look & Feel sicherzustellen – **gut formatierter Code ist leichter zu lesen, zu verstehen und zu debuggen** [25].

Abschließend: Eine saubere Formatierung mit sinnvoller visueller Gliederung lässt Muster im Code sofort erkennen und reduziert die Zeit, die man zum „Parsen“ des Scripts braucht. Ein Entwickler sollte in der Lage sein, Ihren Query-Code **schnell zu scannen** und die Haupttransformationen sowie deren Abfolge zu erfassen – klare Einrückungen und Abschnittsbildung machen genau das möglich [21]. So, wie man in SQL-Abfragen Absätze und Zeilenumbrüche für SELECT, FROM, WHERE, GROUP BY etc. einsetzt, sollten in M die einzelnen Schritte und Blöcke klar strukturiert sein. Das Resultat ist **ein Clean-Code-Stil in M**, der professionelle Qualität widerspiegelt.

3. Modularität und Wiederverwendbarkeit

Ein weiterer Grundsatz von Clean Code und guter BI-Entwicklung lautet: **Dupliziere nichts unnötig**. In Power Query M bedeutet dies, Transformationen möglichst **modular** aufzubauen und **wiederverwendbare Bausteine** zu schaffen, anstatt gleiche Logik mehrfach zu implementieren. Gerade in größeren Power BI Modellen mit vielen Abfragen zahlt sich Modularisierung in besserer Wartbarkeit und geringerem Fehlerrisiko aus.

Custom Functions (Benutzerdefinierte Funktionen): Power Query M ist eine funktionale Sprache – Sie können eigene Funktionen definieren, um Logik wiederzuverwenden. Erkennen Sie also Muster oder Berechnungen, die mehrfach vorkommen, und kapseln Sie sie in eine Funktion. Beispiel: Wenn Sie eine komplizierte Berechnung zur Klassifizierung von Kunden in mehreren Abfragen brauchen, schreiben Sie eine Funktion `fnKlassifiziereKunde(Umsatz as number, Dauer as number) as text => ...` die einen Kundenklassifizierungs-Code zurückgibt. Diese Funktion kann dann in verschiedenen Abfragen oder in einer **benutzerdefinierten Spalte** wiederverwendet werden, anstatt denselben If-Else-Block zigmal zu schreiben. Achten Sie bei Funktions- und Parameter-Namen ebenfalls auf unser Namensschema bzw. klare Bezeichnungen. Parameter in Funktionen sollten nach ihrem Zweck benannt sein (z.B. `KundenUmsatz`, `KundenSeitJahren` statt generisch `value1`, `value2`). Funktionen selbst können – anders als Abfrage-Schritte – ruhig in Englisch sein, falls sie allgemeingültig sind (viele verwenden Prefix `fn` und einen englischen Namen, etwa `fnRemoveHtmlTags`), aber innerhalb eines

Unternehmens kann man sie auch deutsch benennen. Hauptsache, das Schema ist konsistent und der Name spiegelt *was* die Funktion tut.

Wie in anderen Sprachen (z.B. Python oder C#) gilt: **Funktionen sollten eine Aufgabe kapseln** – „eine Sache gut machen“. In M heißt das, eine Funktion sollte nicht unübersichtlich groß werden oder zu viele verschiedene Dinge tun. Lieber mehrere kleine Funktionen (die man dann verschachteln kann) als ein monolithischer Klotz. Kleine, fokussierte Funktionen sind leichter zu testen und zu warten [2]. Ein Vorteil von Funktionen in M ist auch, dass man sie in *separaten Queries* speichern kann (Stichwort: *Parameter- oder Funktions-Queries* in Power BI). So können Funktionen in der Abfrage-Übersicht als eigene Objekte organisiert werden (oft in einem Ordner "Funktionen") und stören nicht den Datenfluss direkt. Laut einem Best-Practice-Leitfaden sollten wiederverwendbare Funktionen in eigenen Abfragen ausgelagert werden, um die Organisation zu verbessern [29]. In Power BI Desktop markieren Sie solche Query-Parameter/Funktionen dann als *“Nicht laden”*, damit sie nicht als eigenständige Tabelle im Modell landen.

Vermeidung von Hardcodierung – Parameter und Konfiguration: Ein häufiger Stolperstein in BI-Projekten ist Hardcodierung, z.B. von Dateipfaden, URLs, Filtern oder Schwellenwerten, direkt im M-Code. Statt feste Werte im Code zu verstreuen, verwenden Sie **Parameter** oder **Konfigurationsabfragen**. Power Query erlaubt es, Parameter zu definieren (im PBID Editor über *“Neuer Parameter”*) – z.B. einen Parameter `pDatenpfad` für einen Dateipfad oder `pStichtag` für ein bestimmtes Datum. Diese Parameter kann man dann im Code referenzieren (z.B. `File.Contents(pDatenpfad)` anstelle des festen Strings). Der Vorteil: Wird der Pfad geändert, muss nur der Parameterwert geändert werden, nicht jede Abfrage [30]. Das erhöht Flexibilität und Wartbarkeit enorm [31]. In vielen Projekten gibt es ganze *Konfigurations-Tabellen* – etwa eine Query `Parameter` die Schlüssel-Wert-Paare hält (z.B. {"URL_API", "https://api..."}), welche per Funktionen wie `Record.Field` ausgelesen werden. Damit zentralisiert man alle Konfigurationen. Ähnlich können bestimmte Mapping-Tabellen (z.B. für Gruppierungen, Übersetzungen von Codes etc.) als eigene Queries vorgehalten werden und dann via Merge oder Funktionen angewandt werden, statt überall im Code mit verschachtelten if/else oder `switch` zu hantieren. **Keine Magic Numbers**: Wenn ein Wert semantische Bedeutung hat (z.B. 7 = Anzahl Wochentage, 0.2 = Wechselkurs), gehört er als Name oder in eine Konfiguration, nicht als nackte Zahl in den Code [32].

Verweis-Queries und Staging: Modularität bedeutet auch, komplexe Abfragen in sinnvoll getrennte Teile zu zerlegen. Power Query erlaubt es, eine Abfrage auf das Ergebnis einer anderen Abfrage aufzubauen (durch „Verweis“ im Editor oder einfach den Query-Namen als Source zu nutzen). Nutzen Sie das, um **Mehrschicht-Modelle** zu bauen: z.B. eine Abfrage `Quelle_Rohdaten` holt die Daten aus der Quelle (ohne große Transformation), eine zweite Abfrage `Transformierte_Daten` referenziert `Quelle_Rohdaten` und nimmt alle Bereinigungen vor, und eine dritte `Fertig_für_Model` macht die Aggregationen und lieferfertige Struktur. Diese Aufteilung in Extract, Transform, Load (ETL) Layer ist empfohlen von BI-Experten [33] [34] und hilft, Übersicht zu bewahren. Ken Puls und Miguel Escobar schlagen etwa drei Abfragetypen vor: *Raw Data (E)*, *Staging (T)* und *Endgültige Tabelle (L)* [33]. Die Idee dahinter: Jede Abfrage-Ebene hat einen klaren Fokus (Extraktion, Bereinigung, finales Modell) und kann getrennt betrachtet werden. Außerdem können mehrere End-Abfragen dieselbe Raw- oder Staging-Abfrage nutzen, ohne die Extraktion doppelt zu machen – dank *Query Caching* in Power BI muss die gemeinsame Quelle nur einmal geladen werden und wird dann von den Folgeabfragen aus dem Cache gelesen [35]. Das verbessert die Performance und vermeidet doppelte Last auf der Datenquelle.

Beispiel: Sie haben Produktdaten, die Sie in zwei verschiedenen Berichten etwas anders aufbereiten müssen. Statt zweimal die Quelle auszulesen und jeweils ähnlich zu bereinigen, machen Sie eine Abfrage `Produkt_Roh` (lädt alles aus DB), dann `Produkt_Bereinigt` (entfernt unnötige Spalten,

vereinheitlicht Formate, etc.). Für Bericht A nutzen Sie `Produkt_Bereinigt` weiter, für Bericht B ebenso, ggf. mit unterschiedlichen finalen Schritten. So ist sichergestellt, dass die Bereinigung zentral definiert ist. Wenn sich das Quellschema ändert, brauchen Sie nur `Produkt_Bereinigt` anzupassen, nicht beide Berichte. Das ist **DRY – Don't repeat yourself** im Power Query Kontext.

Schema-konforme Parameter und Schritte: Wenden Sie die Namenskonvention auch auf Ihre modularen Bausteine an. Benennen Sie also zum Beispiel eine Staging-Abfrage ebenfalls nach *AktionWasNachKriterium*, z.B. `FilternTransaktionenNachProdukt` als eigenständige Abfrage, die Transaktionen auf einen bestimmten Produktkreis einschränkt (diese könnte als Funktion mit Parameter Produktliste umgesetzt sein). Oder benennen Sie eine Parameter-Abfrage für einen Stichtag als `ParameterStichtag`. Einheitliche Präfixe wie `ParameterXYZ` oder Ordner für Parameter können helfen, diese sofort zu erkennen. In Foren nutzen einige für Parameter gerne den Präfix "p" [17] [36], z.B. `pDateiPfad`. Ob Präfix oder nicht – **Hauptsache, man erkennt den Zweck auf den ersten Blick**.

Durch Modularisierung erreichen Sie, dass Ihr M-Code **weniger Redundanz** enthält und Änderungen leichter implementiert werden können. Wenn morgen die Logik der Kundenklassifizierung angepasst werden muss, ändern Sie die Funktion an *einer* Stelle, statt zehn Abfragen zu durchforsten. Zudem erhöht es die **Testbarkeit**: Sie können z.B. eine Funktions-Query isoliert mit verschiedenen Eingaben testen. Modularität fördert auch Teamwork – wenn jeder gewisse Standardfunktionen oder -abfragen kennt, können Aufgaben besser aufgeteilt werden (z.B. einer baut die Rohdaten-Abfrage, ein anderer schreibt darauf aufbauend verschiedene Auswertungen).

Zum Abschluss dieses Abschnitts ein paar Best Practices aus Entwickler-Sicht:

- **Keine Copy-Paste-Doppelungen**: Wenn Sie Codeblöcke identisch in mehreren Queries sehen, ist es Zeit für eine Funktion oder zentrale Query. Die einzige Ausnahme könnten triviale 1-2 Zeilen sein, aber auch da lohnt oft eine kleine Hilfsabfrage.
- **Kleine Schritte, klare Zwischenergebnisse**: Scheuen Sie nicht, eine komplexe Transformation in mehrere Schritte zu zerlegen. „Wir verwenden Zwischen-Variablen (let-in Blocks), um komplexe Transformationen in kleinere Schritte aufzuteilen und den Code lesbarer zu machen“ [37]. Jeder Schritt kann einen Namen erhalten, der den Zwischenstand beschreibt. Das macht den Code narrativer und nachvollziehbarer, als eine Monsterformel, und Sie können leichter debuggen, indem Sie Schritt für Schritt die Ergebnisse prüfen.
- **Verwendung von Query-Gruppen**: Organisieren Sie Ihre Queries in Ordnern (Gruppen) nach Funktion – z.B. *Parameter*, *Staging*, *Dimensionen*, *Faktentabellen*. So behält man auch in der GUI den Überblick. Benennen Sie die Gruppen und Queries so, dass deren Rolle eindeutig ist (z.B. Gruppe "Parameter" enthält alle pXYZ, Gruppe "Fakten" alle Endtabellen etc.) [38].
- **Team-Abstimmung**: Entwickeln Sie gemeinsam mit dem Team Muster und Vorlagen. Wenn z.B. jeder häufig benötigte Funktionen (Datumstabellen-Generator, Feiertagsberechnung etc.) in einer *BI-Bibliothek* zur Verfügung hat, fördert das Wiederverwendung. Manche Teams halten eine „M Cookbooks“-Datei bereit, wo nützliche Snippets und Funktionen gesammelt sind [39] – das kann man versionieren und bei neuen Projekten importieren.
- **Keine Angst vor Extra-Queries**: Manchmal zögern Entwickler, Hilfsqueries anzulegen, um das Feld nicht zu füllen. Aber eine klare Aufteilung z.B. in 3 Queries statt einer riesigen Query ist oft besser. Power Query ist darauf ausgelegt und dank Abhängigkeitsgraph und Cache wird die Performance bei richtiger Anwendung nicht leiden [35]. Auch können Sie unnötige Endlast vermeiden, indem Sie nur finale Queries laden und reine Zwischenqueries auf *„Nur Verbindung“* belassen. Moderne Ansätze zeigen, dass Modularisierung die Verständlichkeit erhöht – *„man kann Abfragen als Schichten betrachten, um Verständnis zu erleichtern“* [40]. Natürlich sollte man es nicht übertreiben: Wenn zu viele kleine Queries entstehen, wird es unübersichtlich und

erschwert das Nachvollziehen des Datenflusses [41]. Finden Sie hier ein gutes Mittelmaß und dokumentieren Sie die Abfrage-Kaskaden.

Kurz gesagt: **Modularer M-Code** folgt dem Motto "divide and conquer". Er ist in überschaubare Teile zerlegt, vermeidet Wiederholungen und lässt sich anpassen, ohne dass man überall suchen und ersetzen muss. Das macht Ihr Power BI Projekt robuster und skalierbarer – ein Muss in professionellen BI-Entwicklungsprozessen.

4. Performance-Optimierung mit schema-basierten Namen

Sauberer Code sollte nicht nur schön aussehen, sondern auch **effizient** laufen. Interessanterweise hängen Lesbarkeit und Performance oft zusammen: Ein klar strukturierter, gut benannter Abfrageplan hilft dabei, ineffiziente Schritte zu erkennen und zu optimieren. In Power Query kommt ein zentrales Konzept hinzu – **Query Folding**. Damit ist gemeint, dass Power Query Ihre Schrittfolge in SQL oder eine andere Quellsystemsprache übersetzt, sodass Filter, Aggregationen usw. nach Möglichkeit vom Datenbankserver ausgeführt werden (und nicht lokal auf dem Gerät) [42]. Das führt zu enormen Beschleunigungen, wenn es gelingt, und sollte daher gefördert werden.

Zunächst wichtig: Die **Namen der Schritte beeinflussen die Performance direkt nicht** – sie sind vor allem für den Menschen da. Ob ein Schritt `GefilterteZeilen` oder `Schritt1` heißt, ändert an der Ausführungslogik nichts. **Aber**: Die **Anordnung und Art** der Schritte hat sehr wohl Einfluss. Unser Namensschema „Aktion+Was+Nach“ unterstützt hier indirekt, denn es zwingt uns, *jede Transformation explizit zu machen* und begünstigt eine sinnvolle Reihenfolge. Beispielsweise empfiehlt Microsoft: **Filter so früh wie möglich ausführen** [43]. Wenn Sie dem Schema folgen, haben Sie vermutlich einen Schritt namens `Filtern...Nach...` ziemlich weit oben in Ihrer Query. Das zeigt schon: es wurde gefiltert, und zwar nach einem bestimmten Kriterium. Tatsächlich ist *„so früh wie möglich filtern“* einer der bekanntesten Performance-Tipps [44]: Unerwünschte Daten sollen direkt am Anfang entfernt werden, damit nachfolgende, aufwändige Schritte mit weniger Daten arbeiten [44]. Idealerweise wird dieser Filter dank Query Folding sogar in die Quelle verlagert (z.B. als `WHERE`-Clause bei einer DB-Abfrage) [45], was enorme Effizienz bringt – Power Query zieht dann gar nicht erst alle Daten, sondern nur den gefilterten Satz. Unser Namensschema hilft hier, weil ein früh auftauchendes `Filtern...` in den Schritten (und zwar mit Benennung des Kriteriums) erkennen lässt, ob der Entwickler an dieser Stellschraube gedreht hat. Bleibt ein Filter-Schritt ganz am Ende übrig, wäre das ein *Code Smell* – man würde sehen „Warum filtert er **nach** dem Gruppieren?“, was unüblich ist. So dienen die Schrittnamen auch als Dokumentation der Reihenfolge.

Ähnliches gilt für andere teure Operationen: **“Expensive operations last”** lautet ein weiteres Prinzip [46]. Auf Deutsch: Rechenintensive Schritte wie Sortieren, Gruppieren, Joins etc. sollten nach Möglichkeit **ans Ende** der Query gestellt werden, wenn die Datenmenge bereits reduziert ist. Sortieren z.B. kann die Quelle oft falten, aber ist trotzdem heavy, weil es typischerweise alle Daten sichten muss [47]. Wenn zuerst gefiltert/gefiltert wird, sortiert man weniger Zeilen. Oder wenn man erst 10 Spalten entfernt und dann gruppiert, muss weniger durchs Aggregat geschleust werden. **Die Reihenfolge der Schritte ist daher kritisch** – eine falsche Reihenfolge kann die Möglichkeit zum Query Folding zerstören [46] oder unnötig viele Daten durch teure Schritte schicken. Gut benannte Schritte machen sofort deutlich, *was* wann passiert. Man erkennt zum Beispiel: `SortierenTabelleNachDatum` steht an vorletzter Stelle – okay. Wäre es ganz oben, würde man stutzen und überlegen, ob das notwendig ist oder ob es Folding verhindert (denn manchmal bricht eine Sortierung in PQ das Folding, falls z.B. danach noch ein Merge kommt, weil nicht alle Quellen sortiert ziehen können). Auch sieht man, welche Joins passieren (`Join…Nach…`) – wenn in einem Projekt mehrere Joins hintereinander passieren, sollte man genau hinschauen, ob man sie kombinieren kann oder ob beide noch gefaltet werden.

Query Folding überprüfen: Als Entwickler sollten Sie stets prüfen, ob Ihre Abfragen folden (im Power Query Editor: rechte Maustaste auf Schritt -> *"Abfragefaltung anzeigen"*). Die Schrittnamen helfen hierbei indirekt – sie sind **Checkpunkte**, an denen Sie sich fragen können: *Kann mein „FilternXY" gefaltet werden?*, *Konnte der „Gruppieren"-Schritt vom Server erledigt werden oder passiert er lokal?*. Wenn nicht, könnte man vielleicht die Gruppierung anders formulieren oder auf die Datenbank vorverlagern. In Summe fördert eine strukturierte Aufteilung, dass **ein Maximum an Schritten vom Quellsystem erledigt wird**. Ein Beispiel: Filter und Spaltenauswahl (Remove Columns) sind i.d.R. foldable. Wenn Sie diese beiden früh im Code ausführen, holen Sie nur relevante Daten und Felder aus der Quelle – "die Power Query Engine erhält nur den Teil des Datensatzes, der benötigt wird" [45]. Kommt dagegen ein nicht-faltbarer Schritt (z.B. ein benutzerdefiniertes M-Skript, das der Datenbank nicht bekannt ist) zu früh, bricht die Faltung ab und alles danach wird lokal ausgeführt. Daher: Zögern Sie möglichst alle nicht-faltbaren Aktionen (wie bestimmte Texttransformationen, komplexe benutzerdefinierte Funktionen, etc.) *bis zum Ende* heraus – zumindest bis nach den wesentlichen Filtern und Joins. Wie gesagt, gut benannte Schritte wie `BerechneRankingInTabelle` würden z.B. sofort signalisieren: Hier wird ein Ranking berechnet – vermutlich ein nicht foldbarer Schritt (da Ranking in SQL kompliziert). Diesen würde man idealerweise ans Ende stellen, nach allen Filtern, Sorts etc., die die Datenmenge begrenzen.

Namenslänge vs. Verständlichkeit: Oft kommt die Frage auf, ob die **ausführlichen Namen** (wie `JoinUmsaetzeMitKundenNachKundenID`) nicht zu lang und umständlich sind, evtl. sogar die Performance beeinflussen. Zur Performance: Die Schrittnamen sind rein textuelle Labels, sie beeinflussen die Ausführungszeit oder Ressourcen nicht spürbar. Selbst extrem lange Namen (50+ Zeichen) sind intern kein Problem – höchstens in der Benutzeroberfläche unhandlich. Performance-Probleme durch Namen sind höchstens indirekt (z.B. wenn man ein sehr langes benanntes Ausdruck-`Record` zusammenbaut). Viel wichtiger ist die **Usability**: Allzu lange Namen passen nicht mehr ins Query-Settings-Fenster, man sieht sie nur abgeschnitten. Daher rät ein Community-Guide: *„Vermeide lange Schritt-Namen, besonders wenn sie die Standardbreite des Interface überlaufen. Man sollte den Abfragefenster-Bereich nicht extra erweitern müssen, um die Schritte zu lesen."* [48]. Diese praktische Faustregel heißt: Mach die Namen **so lang wie nötig, aber so kurz wie möglich**. Lassen Sie unwichtige Wörter weg, nutzen Sie etablierte Abkürzungen, wo sie eindeutig sind. Z.B. reicht `KundenNachLandFiltern` statt `KundenNachLandFilternWoLandGleichXYZ` – das `Wo ...` kann man sich sparen, weil `NachLand` schon impliziert, dass gefiltert wird auf ein Landkriterium. Oder `GruppierenUmsaetzeNachProdukt` ist ausreichend statt `GruppierenUmsaetzeNachProduktIDZusammenfassen`. Die Kunst ist, **Klarheit zu bewahren, ohne in Romanlänge abzuschweifen** [13]. Wenn ein Name doch sehr lang wird, überlegen Sie, ob der Schritt zu viel auf einmal macht. Man kann ggf. lieber zwei Schritte draus machen, die jeweils einen kürzeren Namen tragen, was wieder der Lesbarkeit dient (und das Folding meistens nicht stört). Beispiel: Statt `FilternUndBerechnenUmsatzNachRegionUndJahr` als Ein-Schritt-Monstrum, lieber `FilternUmsaetzeNachRegion` und dann `BerechnenJahresumsatzNachRegion` getrennt.

Verständlichkeit geht vor Kürze [13], aber **Übertreiben Sie es nicht mit der Detailtiefe im Namen**, wenn es nicht nötig ist. Der Kontext der Abfrage kann implizit einiges klar machen. Befindet man sich z.B. in der Abfrage `Verkaufsdaten`, muss nicht jeder Schritt `Verkauf` oder `Umsatz` im Namen tragen – das ist offensichtlich. Da könnte `FilternNachAktuellesJahr` genügen, statt `FilternVerkaufsdatenNachAktuellemJahr`. Oder innerhalb einer Kunden-Abfrage muss nicht jeder Schritt `Kunde` enthalten; `SortierenNachName` ist klar auf Kunden bezogen, wenn die Abfrage ohnehin Kundendaten behandelt. Nutzen Sie also die **Kontext-Informationen**: Query-Name, vorherige Schritte etc. als Rahmen, damit die Schrittnamen nicht redundant werden.

Abkürzungen und Namensstil: Wie erwähnt, vermeiden Sie eigenwillige Abkürzungen. Wenn die Domäne jedoch gängige Akronyme hat, können diese benutzt werden (z.B. `SKU` für *stock keeping unit*, `EOM` für *End of Month*, im deutschen Umfeld evtl. `EK` für Einkaufspreis etc., falls teamweit bekannt). Ansonsten schreiben Sie Begriffe aus – `Menge` statt `Mg.`, `Durchschnitt` statt `avg` (im deutschen Kontext). Bei technischen Begriffen, die im Original Englisch sind (z.B. *Pivot*, *Merge/Join*, *Filter*), entscheiden Sie sich für eine Sprache und seien Sie konsistent. In unserem Schema haben wir uns für deutsche Verben entschieden (*Filtern, Gruppieren, Sortieren…*). Das passt, weil viele Power Query-Anwender im deutschen Umfeld so denken (auch die GUI im Deutschen verwendet *„Zeilen entfernen“*, *„Gruppieren nach“* etc.). Man kann aber genauso gut Englisch verwenden (*FilterCustomersByCountry*). **Wichtig ist Einheitlichkeit innerhalb eines Projekts.** Mischen Sie nicht Deutsch und Englisch wild – das wirkt unprofessionell. Gerade weil unsere Namen länger sind, macht Mischsprache es noch verwirrender. Also z.B. nicht `FilterCustomersNachLand` – hier besser komplett Englisch oder komplett Deutsch. Wenn Sie ein internationales Team haben, wäre Englisch konsistenter, ansonsten deutsch, damit die Fachabteilung im Code mitlesen kann.

Unnötige Schritte vermeiden: Im Kontext Performance sei noch betont: Jeder Schritt, der nichts zur Endtransformation beiträgt, sollte vermieden werden. Es gibt oft Fälle, wo der Power Query Editor „automatische“ Schritte einfügt (z.B. *Geänderter Typ* mehrfach, *Entfernte andere Spalten* obwohl man gleich danach ohnehin einen Filter auf die gleichen Spalten hat, etc.). Überprüfen Sie Ihre Abfolge auf solche Redundanzen. *„Unnötige Schritte und Transformationen minimieren“*, rät eine Anleitung [49], da jeder Schritt – falls nicht weggefoldet – natürlich Verarbeitung kostet. Das heißt aber nicht, dass man aus Angst vor zu vielen Schritten alles in einen packen soll. Lieber 10 gut durchdachte Schritte als 5 chaotische – aber vermeiden Sie z.B. zwei aufeinanderfolgende `Changed Type`-Schritte oder erst Spalten umbenennen und später wieder löschen etc. Jede Transformation sollte einen echten Zweck erfüllen. Wenn **Query Folding aktiv** ist, werden viele Schritte ohnehin zusammengefasst zum Source-Query, dann ist die Schrittanzahl fast egal. Dennoch: Weniger, klar definierte Schritte = weniger Fehlermöglichkeiten und in Summe schnellere Ausführung [49].

Abschließend: Unser Namensschema und Clean-Code-Ansatz *erzwingt* gewissermaßen einen guten Stil, der auch Performance zugutekommt. Indem Sie jeden Schritt klar benennen, denken Sie auch bewusster über dessen Notwendigkeit und Position nach. Sie erkennen eher, wenn z.B. ein `Filtern...` fehlt (der Name würde auffallen in der Sequenz) und fügen ihn hinzu, bevor Unmengen Daten lokal verarbeitet werden. Ebenso sehen Sie, wenn mehrere teure Aktionen nacheinander passieren – dann überlegen Sie ggf., ob diese die Reihenfolge des Faltens stören. All das führt zu effizienterem Code. Und wenn mal Optimierung nötig ist, erleichtert es die Arbeit immens, wenn man im Leistungsanalyse-Log von Power BI sofort Schritt-Namen sieht wie *„SortierenOrteNachUmsatz“* und weiß, was da passiert, anstatt „Schritt 7“ raten zu müssen. Gute Namen dokumentieren den „Query Plan“ für Menschen und helfen so indirekt, Performance-Probleme zu finden.

Query Folding bewusst steuern: Noch ein Tipp – man kann manchmal Folding erzwingen oder manuell gestalten (z.B. indem man einen SQL-Query als Source verwendet). Unser Best-Practice-Ansatz wäre: Wenn möglich, nutze die nativen PQ-Transformationen, da diese folden können. Sollte etwas nicht folden (z.B. komplexe Berechnung), überlegen, ob es im SQL-View oder in der Datenquelle vorgelagert erledigt werden kann. Wenn nicht, okay – dann zumindest restliche Schritte drumherum optimieren. Insofern: Halten Sie die Abfrage zunächst so einfach und *source-freundlich* wie möglich (Filter, Spaltenwahl, einfache Berechnungen), und führen Sie Speziallogik später aus. Das ist in Code nach unserem Schema gut sichtbar.

Balance finden: Abschließend zur Namenslänge vs Verständlichkeit – es ist ein Abwägen. Vermeiden Sie unleserliche Kürzelwüsten, aber auch extrem verbose Sätze. Ein guter Name ist prägnant und klar. Fragen Sie sich: *Würde jemand aus meinem Team oder ich selbst in 3 Monaten sofort kapieren, was dieser*

Schritt macht? Wenn ja, ist der Name gut. Wenn nein, verbessern. Und wenn der Name so lang ist, dass man ihn kaum mehr lesen mag, überlegen Sie, ob der Schritt zu komplex ist. Halten Sie sich an das Motto: **"Use meaningful and descriptive names… Prefer clarity over brevity."** [13] – dann sind Sie auf dem richtigen Weg.

5. Tooling und Praxisumsetzung

Die besten Richtlinien nützen wenig, wenn sie nicht in der Praxis gelebt werden. In diesem Abschnitt geht es um die **Umsetzung im Teamkontext**, die Einführung der Konventionen in bestehende Projekte (Migrationsstrategien) und die laufende Qualitätssicherung des M-Codes.

Teamweite Standards etablieren: Stellen Sie sicher, dass alle BI-Entwickler im Team die vereinbarten Namenskonventionen und Formatierungsrichtlinien kennen und anwenden. Dokumentieren Sie diese Guidelines (z.B. in einem internen Wiki oder Styleguide-Dokument) und verweisen Sie beim Code-Review darauf. Der Guide, den Sie gerade lesen, könnte z.B. als Grundlage für Ihren Team-Styleguide dienen. Wichtig ist die **Konsistenz im Team** – gleiche Begriffe für gleiche Dinge, gleicher Formatstil in allen PBIX oder Dataflow Definitionen [19] . Wenn jeder sein eigenes Süppchen kocht (einer snake_case englisch, der nächste PascalCase deutsch, etc.), verliert man die Vorteile der Einheitlichkeit. Gerade in Power BI, wo mehrere Entwickler an einem Modell arbeiten können (z.B. via *Deployment Pipelines* oder in Zukunft eventuell Git-Integration), muss klar sein, dass Code von Person A für Person B nahtlos lesbar und weiterpflegbar ist. *"Sorgt dafür, dass alle im Team die gleiche Idee verfolgen… fördert Abteilungs-Standards auch für Power Query Schritte und M-Code."* [19] .

Code Reviews & Pair Programming: Führen Sie für komplexere Abfragen Code Reviews ein, bei denen ein Kollege den M-Code gegenliest – analog zum Vier-Augen-Prinzip in der Softwareentwicklung. Dabei sollte nicht nur die Korrektheit, sondern explizit auch die **Einhaltung des Styleguides** geprüft werden (z.B. „Wurden die Schrittnamen sinnvoll vergeben?", „Gibt es zu lange oder kryptische Namen?", „Ist die Formatierung konsistent und gut lesbar?", „Gibt es unnötige Schritte?"). Solche Reviews fördern das gegenseitige Lernen und stellen sicher, dass Clean-Code-Praktiken eingehalten werden. Alternativ kann man auch mal *Pair Programming* im Power Query Editor betreiben: Zwei Entwickler schauen sich gemeinsam eine komplexe Transformation an und überlegen, wie man sie am besten strukturiert. Oft bringt das frische Ideen – z.B. jemand erkennt, dass man eine Funktion auslagern könnte oder dass ein bestimmter Schritt folding bricht.

Tool-Unterstützung: Nutzen Sie Werkzeuge, um die Codequalität sicherzustellen. Einige nützliche Tools und Tipps:

- **Visual Studio Code mit M Language Extension**: Microsoft stellt eine Power Query SDK für Visual Studio Code bereit, die Syntax-Highlighting und IntelliSense für M bietet. Man kann den M-Code aus Power BI Desktop exportieren (oder via Copy im Editor holen), in VSCode bearbeiten und zurückkopieren. So profitieren Sie von einer vollwertigen Editor-Umgebung (mit Suchen/Ersetzen, Code-Folding, eventuell Snippets). Auch gemeinsames Bearbeiten wird so einfacher, da VSCode git-verbunden sein kann.
- **Power Query Formatter**: Wie zuvor erwähnt, kann man mit Online-Formatter (z.B. powerqueryformatter.com) oder Tools wie dem Tabular Editor Script von Data Goblins den Code **automatisch formatieren lassen** [28] . Erwägen Sie, so einen Schritt in Ihren Workflow zu integrieren – z.B. immer bevor ein PBIX ins Repo eingecheckt wird, einmal den Code zu formatieren.
- **Versionsverwaltung**: Leider ist Power BI Desktop (.pbix) ein Binärformat, das nicht direkt git-diff-freundlich ist. Dennoch können Sie Ihren M-Code versionieren, indem Sie ihn **als Teil der**

Datasets-Dokumentation exportieren. Es gibt Community-Tools und Skripte, um M-Code aus einem PBIX oder einem PBI Dataflow zu extrahieren (z.B. der *Power BI Helper* oder mittels der Power BI REST API/PowerShell). In einem Dataflow (JSON basiertes Definition-File) ist der M-Code sogar als Text enthalten, der versioniert werden kann. Ziehen Sie in Betracht, zumindest für wichtige Modelle, den M-Code regelmäßig zu exportieren und in ein Git-Repo zu legen – so haben Sie Verlaufsinformationen und können Änderungen nachvollziehen.

- **Linting und Best Practice Checks**: Noch recht neu und selten sind automatisierte Linter für M. Aber man könnte mit etwas Aufwand eigene Prüfungen schreiben – zum Beispiel ein Skript, das alle Schrittnamen gegen eine Liste verbotener Wörter prüft (z.B. „Filtered Rows“ auftauchen? Dann wurde Default-Name nicht geändert – *Fehler!*). Oder man schreibt ein PowerShell-Skript, das warnt, wenn ein Schrittname länger als 50 Zeichen ist. Solche Tools gibt es nicht out-of-the-box, aber in einem ambitionierten Team könnte man sie entwickeln. Alternativ: Setzen Sie auf manuelle Checklisten im Review.

Migrationsstrategien: Wenn Sie bestehende Power BI Abfragen haben, die noch nicht den Clean-Code-Richtlinien entsprechen, gehen Sie schrittweise vor. Eine Migration bedeutet hier vor allem: Schrittnamen und ggf. Struktur nachziehen, ohne die inhaltliche Logik zu verändern. Zum Glück sind **Schrittumbenennungen relativ risikofrei**, solange Sie es *im Editor* machen, denn Power Query aktualisiert referenzierende Schritte automatisch. Beispiel: Sie haben `Schritt1 = Table.SelectRows(...); Schritt2 = Table.SelectColumns(Schritt1, {...})`. Wenn Sie `Schritt1` per Rename in `GefilterteZeilen` ändern, passt der Editor den Verweis in Schritt2 ebenfalls an. Daher können Sie beherzt durch Ihre Queries gehen und die Namen verbessern. Testen Sie danach, ob die Abfrage noch funktioniert (ggf. Vergleich mit dem alten Ergebnis, sofern möglich). Idealerweise machen Sie immer nur kleine Änderungen und prüfen sofort, um Fehler einzugrenzen.

Wenn ein bestehendes Projekt sehr viele Abfragen hat, priorisieren Sie: Fangen Sie bei den wichtigsten oder komplexesten an, denn dort zahlt sich Clean Code am meisten aus. Möglicherweise können triviale Abfragen so bleiben, wie sie sind (wenn nur 2 Schritte, Standardnamen, und nie jemand außer dem Ersteller schaut drauf – pragmatisch bleiben). Aber Kernabfragen, die Dataflows oder Modelle speisen, sollten aufgeräumt werden. Kommunizieren Sie dem Fachbereich ggf., dass Sie eine Refactoring-Runde einlegen (damit man sich nicht wundert, warum ein Deployment keine neuen Features bringt, sondern „nur“ Code-Verbesserung – das ist genauso wichtig für langfristige Qualität!).

Schulung und Bewusstsein: Machen Sie allen Beteiligten klar, *warum* diese Guidelines sinnvoll sind – nämlich um **Fehler zu vermeiden und Wartung zu erleichtern**. Neue Kollegen sollten im Onboarding mit dem Styleguide vertraut gemacht werden. Vielleicht kann man intern kleine „Clean M Code“-Sessions abhalten, wo man Beispiele zeigt: *Hier eine Abfrage mit schlechten Namen/unformatiert – unverständlich*. *Hier die gleiche Abfrage mit unseren Standards angewendet – deutlich klarer.* Oft überzeugt so ein Vorher/Nachher-Vergleich am meisten. (Nach dem Motto: „Stop using generic step names!“).

Praxis im Teamalltag: Im Entwicklungsalltag sollte die Schemanamen-Strategie genauso selbstverständlich werden wie z.B. das Benennen von Measures nach [Maßnahme] = Formel in DAX oder das Einrücken von SQL-Statements. Wenn jeder diese Kultur verinnerlicht, entstehen automatisch robustere Lösungen. Zudem fördert es die **Kommunikation mit Nicht-Entwicklern**: Wenn jemand aus dem Fachbereich mal in einen Dataflow schaut oder einen Fehler debuggt, kann er dank deutscher Klartext-Namen vielleicht erahnen, was passiert. Es ist keine Seltenheit, dass Fach-User im Power BI Service Dataflows betrachten. Mit Schritten wie `PivotierenUmsaetzeNachRegion` können sie grob folgen, was gemacht wird, während `Inserted Axis` o.ä. sie komplett ratlos ließe. Insofern trägt unser Clean-Code-Stil auch zur **Transparenz gegenüber Stakeholdern** bei.

Qualitätssicherung und laufende Wartung: Planen Sie regelmäßige Reviews Ihrer Power Query Schichten. Im Laufe der Zeit neigen Modelle dazu, komplexer zu werden. Mit jeder neuen Anforderung kommen Schritte hinzu. Es lohnt sich, hin und wieder innezuhalten und den M-Code auf Aufräumpotential zu prüfen. Das ist analog zum Refactoring in der Softwareentwicklung: Code von Zeit zu Zeit „entrümpeln". Prüfen Sie: Sind alle Schrittnamen noch konsistent? Gibt es neue Schritte, die nicht dem Schema folgen? Haben sich temporär eingefügte Hilfsqueries eingeschlichen, die man konsolidieren kann? Oft merkt man bei der Wartung, wie dankbar man ist, wenn saubere Namen und Struktur schon vorhanden sind. Das **Fehlerfinden** wird deutlich einfacher: *„Der letzte Refresh ist fehlgeschlagen – aha, im Schritt* `ErweitereKundendaten` *gab es einen Fehler laut Fehlermeldung."* Sofort weiß man, wo man suchen muss. Wäre es „Schritt 17" gewesen, müsste man erst zählen oder raten.

Ein weiterer Aspekt der Qualitätssicherung ist **Performance-Monitoring**. Power Query Editor bietet ja Vorschau, aber im Betrieb (Service) sollte man auf Refresh-Zeiten achten. Sollte ein Refresh plötzlich länger dauern, kann man sich mit den gut benannten Schritten besser auf Spurensuche begeben. Vielleicht sieht man, dass ein neuer Schritt `MergeXYZ` hinzugekommen ist, der nicht foldet und alles verlangsamt – dank der guten Beschriftung versteht man gleich, was dort passiert und kann gegensteuern (z.B. indem man die Zusammenführung anders gestaltet oder Indizes setzt).

Werkzeuge im Betrieb: In Power BI Dataflows kann man mittlerweile mehrere Entwickler an der gleichen Dataflow arbeiten lassen (Sequenziell, nicht gleichzeitig). Hier gilt noch mehr: Wenn ein Kollege eine von Ihnen erstellte Abfrage öffnet, sollte er sich schnell zurechtfinden. Der *Diagramm*-View von Dataflows zeigt die Query-Schritte als Blöcke – diese sind beschriftet mit den Schrittnamen. Man sieht dann ein schönes Ablaufdiagramm mit Kästchen „Quelle", „GefilterteZeilenNachX", „GruppierenNachY" etc., was super selbstdokumentierend ist. Bei Standardnamen würde dort lauter generisches Zeug stehen. Nutzen Sie also ruhig solche Visualisierungen (auch in Power BI Desktop gibt es ab Version 2023 einen Abhängigkeitsdiagramm-View für Queries), um anderen die Logik zu erklären. Wenn die Namen gut sind, *spricht das Diagramm für sich*. Ansonsten hilft ein konsequenter Stil natürlich auch demjenigen, der Fehler meldet: Ein Entwickler kann sagen „Die Abfrage schlägt im Schritt *SortierenKategorienNachUmsatz* fehl" – und jeder im Team weiß sofort, was gemeint ist und findet den Schritt.

Zusammenfassung: Die Einführung eines Clean-Code-Styleguides für Power Query M erfordert initial etwas Disziplin und Abstimmung, bringt aber erhebliche Vorteile. Code-Qualität ist kein Selbstzweck – sie spart mittel- und langfristig Zeit und Nerven. Indem wir auf **eindeutige Benennung, konsistente Formatierung, Modularität und Performance-Tipps** achten, verbessern wir Stabilität und Wartbarkeit. Wichtig ist, dass das **ganze Team an einem Strang zieht** und die Konventionen einhält [50], sonst entsteht Chaos. Fassen wir die Kernpunkte als *Best Practices* zusammen:

- **Sprechende, konsistente Schritt- und Variablennamen** nach dem Schema *„Aktion + Was + Nach Kriterium"*, bevorzugt in deutscher PascalCase-Schreibweise (z.B. `FilternKundenNachLand` statt `Gefilterte Zeilen1`). Keine kryptischen Kürzel; Klarheit geht vor Kürze [13].
- **Klare Code-Formatierung** mit Einrückungen, Zeilenumbrüchen und logischen Absätzen. Keine endlosen Zeilen; ein Schritt pro Zeile; zusammengehörige Transformationsblöcke visuell trennen. Kommentare gezielt einsetzen, um Kontext zu geben, aber primär den Code für sich sprechen lassen [23].
- **Modularisierung und Wiederverwendung**: Redundanz vermeiden durch Custom Functions und geteilte Abfragen. Parameter für alle hart verdrahteten Werte verwenden. Große Abfragen in verständliche Teilabfragen zerlegen (Extract/Staging/Load-Prinzip) [33]. So nach dem Motto: *"Wir nutzen bedeutungsvolle Variablennamen für Zwischenberechnungen und erstellen wiederverwendbare Funktionen für gängige Logik."* [31] [29].

- **Performance und Folding im Blick**: Schritte in sinnvoller Reihenfolge anordnen (Filter früh, aufwändige Schritte spät) [44] [46] . Folding prüfen und wo möglich sicherstellen (richtige Datenquelle/Connector wählen [42] , keine unnötigen Breaking Steps einbauen). Namensschema nutzen, um schnell zu erkennen, wo Optimierungspotenziale liegen. Keine unnötigen Schritte oder Dopplungen – jeder Schritt soll Mehrwert liefern [49] .
- **Team-Disziplin und Tools**: Den Styleguide teamweit verbindlich machen [19] . Code Reviews etablieren, um Qualität zu halten. Editor-Tools und Formatter einsetzen, um Konsistenz zu gewährleisten. Bestehende Abfragen schrittweise refaktorieren, um technischen Schulden vorzubeugen. Im Zweifel: Lieber jetzt etwas Zeit in Aufräumen investieren als später viel Zeit in der Fehlersuche verbringen, weil der Code unverständlich war.

Abschließend sei betont, dass all diese Best Practices darauf abzielen, Power Query Entwicklungen auf ein professionelles Niveau zu heben, vergleichbar mit klassischer Softwareentwicklung. Viele Prinzipien aus der Programmierung (**eindeutige Benennung, einheitliche Formatierung, DRY, KISS, etc.**) lassen sich hervorragend auf M übertragen – man muss es nur tun. Halten Sie sich an diese Richtlinien, und Sie werden feststellen, dass Ihre Power BI-Datenaufbereitung nicht nur stabiler und schneller wird, sondern dass auch die Einarbeitung neuer Kollegen erleichtert wird und die Zusammenarbeit insgesamt runder läuft. **Sauberer M-Code** mag zunächst mehr Aufwand erscheinen, aber er *„verbessert Verständlichkeit, Wartbarkeit und Erweiterbarkeit“* [51] [52] – genau das, was in dynamischen BI-Projekten letztlich den Erfolg ausmacht.

Quellen: Offizielle Power Query Dokumentation und Style-Empfehlungen, sowie bewährte Clean-Code-Prinzipien aus der Softwareentwicklung: - Microsoft Learn – *Best Practices when working with Power Query* [43] [53] (Tipps zu Filterung und Folding)
- DAXPro Blog – *Naming convention for Power Query steps* [10] [19] (Empfehlungen für sprechende Schritt-Namen)
- LinkedIn Pulse – *Keep your Power Query code clean* [15] [7] (Hinweise zu Schrittkommentaren und Leerzeichen-Problematik)
- The Modern Excel – *Ten Commandments for M* [54] [55] (u.a. Kommentare, sprechende Namen, keine Hardcodierung, Funktionen nutzen, Schritte aufteilen)
- Svitla – *Power Query Optimization Best Practices* [44] [46] (Reihenfolge der Transformationen, Connectoren und Folding)
- Robert C. Martin – *Clean Code* Prinzipien [2] [21] (u.a. aussagekräftige Namen, einheitliche Formatierung, sparsame Kommentare)
- Weitere Clean-Code Resourcen (Mittwald, Codigo Blogs) [1] [16] und Community-Beiträge (Reddit) zur Untermauerung praktischer Tipps.

---

[1] [12] [16] [20] [24] [32] Clean Coding: Einführung und aussagekräftige Namen » codigo.at
https://www.codigo.at/2020/02/29/clean-coding-einfuhrung-und-aussagekraftige-namen/

[2] [21] [23] [51] [52] Clean Code - Bedeutung, Herausforderungen und Prinzipien
https://teech.de/wiki/coding/was-ist-clean-code/

[3] [7] [15] [17] [22] Advanced editor and Power Query: keep your code clean!
https://www.linkedin.com/pulse/advanced-editor-power-query-keep-your-code-clean-micha%C5%82-zalewski-o9k5e

[4] [5] [6] [10] [19] [48] Naming convention for Power Query steps | DAX Pro Services
https://daxproservices.com/naming-convention-for-power-query-steps/

[8] Power Apps code readability - Power Apps | Microsoft Learn
https://learn.microsoft.com/en-us/power-apps/guidance/coding-guidelines/code-readability

9 29 31 37 49 54 55 The Modern Excel
https://www.themodernexcel.com/ExcelMacros/BestPracticesPowerQuery.html

11 13 Identifier names - rules and conventions - C# | Microsoft Learn
https://learn.microsoft.com/en-us/dotnet/csharp/fundamentals/coding-style/identifier-names

14 PEP 20 – The Zen of Python | peps.python.org
https://peps.python.org/pep-0020/

18 36 39 Power Query - Best Practices : r/excel
https://www.reddit.com/r/excel/comments/196lu8y/power_query_best_practices/

25 28 Format Power Query in Power BI (Updated) — DATA GOBLINS
https://data-goblins.com/power-bi/format-power-query-automatically

26 If Power Query M advanced editor would be actually...
https://community.fabric.microsoft.com/t5/Service/If-Power-Query-M-advanced-editor-would-be-actually-advanced/m-p/3441655

27 Format the M language in Power Bi Query - Stack Overflow
https://stackoverflow.com/questions/55963898/format-the-m-language-in-power-bi-query

30 38 43 53 Best practices when working with Power Query - Power Query | Microsoft Learn
https://learn.microsoft.com/en-us/power-query/best-practices

33 34 35 40 41 42 44 45 46 47 50 Power Query Optimization | Svitla Systems
https://svitla.com/blog/power-query-optimization/