ChatGPT

Strukturierte, modulare und teamfreundliche Entwicklung in Power Query M (Clean-Code-Prinzipien)

1. Grundlagen: Warum Modularität in Power Query M wichtig ist

In Power Query M (für Power BI und Excel) bedeutet *Modularität*, komplexe Datenabfragen in logisch getrennte Teile aufzuteilen, anstatt einen monolithischen „Alles-in-einem"-Query zu bauen. Eine modulare Herangehensweise verbessert **Lesbarkeit, Wartbarkeit und Wiederverwendbarkeit** des Codes – zentrale Ziele der Clean-Code-Prinzipien [1] [2]. So können Entwickler*innen und Teammitglieder einfacher nachvollziehen, was im ETL-Prozess (Extrahieren, Transformieren, Laden) passiert, und Änderungen gezielt an einzelnen Modulen vornehmen, ohne den ganzen Query zu gefährden.

Power Query erlaubt es beispielsweise, **Abfragen aufeinander aufzubauen** („Referenz-Queries"): Eine Abfrage kann das Ergebnis einer vorherigen Abfrage als Quelle nutzen [2]. Dies ermöglicht es, Datenaufbereitungen in **Schichten** zu organisieren. Ein empfohlener Ansatz ist das **Staging-Prinzip**: Man trennt die Abfragen nach Phase – etwa in **Rohdaten-Extrakt, Bereinigung/Transformation und finale Ausgabe** [3] [4]. Ken Puls und Miguel Escobar schlagen beispielsweise drei Abfrage-Typen vor: **Raw Data (Extrakt)**, **Staging (Transformation)** und **Endgültige Tabelle (Load)** [5] [6]. Jede Schicht hat einen klar definierten Fokus (Datenquelle anbinden, Daten bereinigen, Ergebnis für das Datenmodell formen) und kann separat betrachtet und geprüft werden.

Vorteile der Modularität: Durch solche aufgeteilten Abfragen steigt die Übersichtlichkeit – anstatt einer undurchschaubaren Sequenz von 30 Schritten in einer Abfrage sieht man mehrere kleinere Queries mit jeweils definierter Aufgabe [7] [8]. Zudem fördert es die Wiederverwendung: Mehrere End-Abfragen können auf die gleiche vorbereitende Staging-Abfrage aufsetzen, ohne die Datenquelle mehrfach auszulesen [9]. Power BI nutzt hierbei *Query Caching*, sodass eine geteilte Quelle nur einmal geladen wird und Folgeabfragen das Zwischenergebnis aus dem Cache beziehen können [10]. Dadurch verbessert sich die Performance und die Datenquelle wird nicht unnötig mehrfach belastet [11]. Dieses Prinzip entspricht dem DRY-Prinzip (*Don't Repeat Yourself*), da gemeinsame Transformationen zentral definiert und von mehreren Stellen genutzt werden [12].

Zusammengefasst sorgt Modularität dafür, dass Power Query M-Code **verständlicher, wartungsfreundlicher und effizienter** wird. In einem Team können verschiedene Entwickler*innen an unterschiedlichen Modulen arbeiten oder diese wiederverwenden, ohne sich gegenseitig ins Gehege zu kommen [13]. *Modularität unterstützt auch die Separation of Concerns** (Trennung von Zuständigkeiten): Jede Abfrage oder Funktion erfüllt einen klar umrissenen Zweck, was dem Clean Code Credo „eine Sache gut machen" entspricht [14].

2. Code-Struktur nach logischen Verarbeitungsschritten (Quelle → Transformation → Logik → Output)

Eine bewährte Praxis ist, den M-Code – sei es innerhalb einer einzelnen Abfrage oder über mehrere Abfragen verteilt – entlang der **natürlichen Datenverarbeitungs-Pipeline** zu strukturieren [3]. Typischerweise gliedert sich der Ablauf in vier Phasen:

- **Quelle (Extract):** In diesem ersten Abschnitt wird die Datenquelle angebunden und die Rohdaten werden abgerufen. In Power Query beginnt eine Abfrage oft mit einem Schritt namens `"Source"` (automatisch generiert) [15], welcher die Verbindung zum Datenquelle herstellt (z.B. eine Datei laden, eine Datenbanktabelle abfragen). In einem strukturierten Query sollten ab diesem Punkt alle weiteren Schritte auf dieser Quelle aufbauen und sie schrittweise verfeinern [7]. *Best Practice:* Halten Sie in diesem Schritt nur das Nötigste (keine Transformationen, nur Laden der Rohdaten).

- **Transformation (Transform/Clean):** Anschließend folgt die Datentransformation und -bereinigung. Hier werden z.B. Spalten umbenannt, Datentypen gesetzt, irrelevante Zeilen/Spalten gefiltert oder entfernt, fehlerhafte Werte ersetzt etc. Ziel ist, die Rohdaten in einen sauberen, konsistenten Zustand zu bringen. Es empfiehlt sich, diese **Bereinigungsschritte in einer Staging-Abfrage** vorzunehmen [3] – d.h. eine Abfrage, die nichts weiter tut als die Quelle zu säubern und vorzubereiten, ohne schon finale Berechnungslogik zu enthalten. Innerhalb dieser Phase gilt: *einfache Schritte zuerst, teure Operationen später durchführen* [16]. Zum Beispiel sollte man früh filtern (um Datenvolumen zu reduzieren) [17] und aufwändige Aktionen wie Sortieren möglichst ans Ende stellen [16]. Dadurch bleibt auch das **Query Folding** (die automatische Übersetzung von Schritten in die Quellsprache, etwa SQL) länger intakt, was Performance gewinnt [18] [19].

- **Logik (Business Logic/Calculation):** In vielen ETL-Szenarien folgt nach dem generischen Säubern eine Phase, in der *geschäftsspezifische Logik* angewendet wird. Das können z.B. Berechnungen neuer Kennzahlen, Klassifizierungen (Kategorien zuordnen), Aggregationen oder komplexe bedingte Spalten sein. Diese Schritte kann man in der gleichen Abfrage (nach der Bereinigung) durchführen oder – bei umfangreicher Logik – in einer eigenen **Logik-Abfrage** kapseln, die auf der gereinigten Staging-Abfrage aufbaut. Dadurch bleibt die Trennung zwischen allgemeiner Bereinigung und spezifischer Berechnung gewahrt (Prinzip der **Single Responsibility** jeder Abfrage). Ein Beispiel: Eine Staging-Abfrage liefert bereinigte Verkaufsdaten, und darauf setzt eine Abfrage *UmsatzAnalyse* auf, die z.B. Rankings berechnet oder Umsatzklassen zuordnet. Diese Trennung erhöht die **Übersichtlichkeit**, da klar erkennbar ist, welche Schritte der generellen Datenaufbereitung dienen und welche der eigentlichen fachlichen Auswertung.

- **Output (Load/Endergebnis):** Schließlich mündet der Query in der Ausgabe, d.h. den Daten, die ins Datenmodell geladen oder ins Excel-Blatt geschrieben werden sollen. In vielen Fällen ist dies einfach der letzte Schritt der Transformationskette (*Endergebnis*). In einer modularen mehrstufigen Architektur entspricht dies der letzten Abfrage (im Sinne von Ken Puls' ETL-Standard die **Endgültige Tabelle**) [20]. Best Practice ist, **nur die finalen Queries zu laden**, während Zwischenschritte (Raw/Staging-Abfragen) auf *Disable Load* gestellt werden, sodass nur das Endergebnis im Datenmodell landet [21]. Die finalen Output-Abfragen sollten sprechende Namen tragen (z.B. "FertigProduktdaten"), die für Anwender klar machen, dass es sich um die endgültigen Daten handelt.

Diese Aufteilung der Schritte kann man auch **innerhalb einer einzelnen Abfrage** durch geschickte Schritt-Benennung und Kommentare sichtbar machen. Beispielsweise könnte man Abschnitte im `let`-Block durch Kommentare trennen: `// Quelle`, `// Transformation`, `// Berechnung`, `// Output` – so erkennt man sofort den logischen Fluss. Unten ein kurzes Beispiel für eine strukturierte Abfrage, die einen Excel-Dateipfad als Parameter nutzt, Daten einliest und transformiert:

```PowerQuery M // Parameter pDateiPfad (Text) enthält den Pfad zur Excel-Datei let QuelleDatei = Excel.Workbook(File.Contents(pDateiPfad), null, true), GeheZuTabelle = QuelleDatei{[Item="Umsatzdaten",Kind="Sheet"]}[Data], PromotedHeaders = Table.PromoteHeaders(GeheZuTabelle, [PromoteAllScalars=true]), GeaenderterDatentyp = Table.TransformColumnTypes(PromotedHeaders, {{"Datum", type date}, {"Umsatz", Int64.Type}, {"Land", type text}}), GefilterteZeilenNachJahr = Table.SelectRows(GeaenderterDatentyp, each Date.Year([Datum]) = 2023), Ergebnis = Table.Group(GefilterteZeilenNachJahr, {"Land"}, {{"SummeUmsatz", each List.Sum([Umsatz]), type number}}) in Ergebnis
```

In dieser Query sind die Schritte klar der Reihe nach aufgebaut: Zuerst `QuelleDatei` (Excel einlesen), dann Navigation zum relevanten Blatt, Kopfzeilen übernommen, Datentypen konvertiert, Daten gefiltert (`GefilterteZeilenNachJahr`) und schließlich aggregiert. Durch die **selbsterklärenden Schrittnamen** erkennt man den Ablauf sofort. Diese Struktur – von Quelle über Transformation/Logik zum Resultat – sollte konsequent eingehalten werden, um die **Leserlichkeit** und **Nachvollziehbarkeit** zu maximieren [22] [7].

3. Konsistente Namenskonventionen für Abfragen und Schritte

Aussagekräftige Benennungen sind ein Grundpfeiler von Clean Code – das gilt ebenso in Power Query M [23]. Jeder Schritt, jede Variable, Abfrage oder Funktion sollte einen Namen tragen, der ihren Zweck klar beschreibt. Leider generiert die Power Query-Oberfläche standardmäßig generische Namen wie *"Geänderter Typ1"* oder *"Gefilterte Zeilen"* für Schritte [24]. Solche Standardnamen bieten kaum Kontext – man erfährt nicht, **was genau geändert oder gefiltert** wurde und warum [24]. **Best Practice:** *Niemals Standardnamen unverändert lassen!* Investieren Sie die Zeit, jedem Schritt und jeder Abfrage einen klaren, spezifischen Namen zu geben [25] [26]. Dies erleichtert Ihnen selbst und anderen das Verständnis des ETL-Prozesses enorm [25]. Microsoft empfiehlt ausdrücklich, Queries durch **Umbenennen von Schritten oder Hinzufügen von Beschreibungen** zu dokumentieren [27] – die Liste der angewandten Schritte sollte wie eine Lesefassung der Transformationen wirken.

Als **zentraler Leitfaden** hat sich das Namensschema **"Aktion + Was + Nach Kriterium"** bewährt [28] [29]. Dabei wird der Name dreigeteilt: Er beginnt mit einem **Verb**, gefolgt vom **Objekt** (Worauf wird die Aktion angewendet?) und optional einer **Bedingung oder genaueren Beschreibung**. Dieses Schema fördert konsistente, selbstdokumentierende Schrittnamen in deutscher Sprache [28]. Im Deutschen ergibt sich daraus typischerweise *Verb+Objekt+Bedingung* [30]. **Beispiele für häufige Transformationen:**

- **Filtern:** `Filtern<Objekt>Nach<Kriterium>` – z.B.

FilternKundenNachLand für einen Schritt, der Kunden nach Land filtert [31]. Oder *FilternZeilenNachStatus* für das Filtern von Zeilen anhand eines Statuswerts [31].
- **Gruppieren:** `Gruppieren<Objekt>Nach<Kriterium>` – z.B. *GruppierenUmsaetzeNachJahr* gruppiert Umsätze nach Jahr [32].
- **Berechnen/Hinzufügen einer neuen Spalte:** `Berechnen<Was>Nach<Kriterium>` oder `Hinzufügen<Was>` – z.B. *BerechnenUmsatzProKategorie* berechnet den Umsatz pro Kategorie [33], oder *HinzufügenIndexSpalte* fügt eine Index-Spalte hinzu [34]. (Je nach Kontext kann man „Nach“ auch durch „Pro“/„Je“ ersetzen, z.B. *BerechneDurchschnittJeProdukt* [33]).
- **Entfernen:** `Entfernen<Was>` – z.B. *EntfernenLeereZeilen* zum Herausfiltern leerer Datensätze, *EntfernenHilfsspalten* zum Entfernen nicht benötigter Spalten [35].
- **Ersetzen:** `Ersetzen<Was>Durch<NeuenWert>` – z.B. *ErsetzenNullDurch0* (Null durch 0 ersetzen) [36].
- **Aufteilen (Split):** `Aufteilen<Was>Nach<Kriterium>` – z.B. *AufteilenNameNachLeerzeichen* (entspricht *Split Name by Space*) [37].
- **Zusammenführen (Join):** `Joinen<X>Mit<Y>Nach<Schluessel>` – z.B. *JoinenBestellungenMitKundenNachKundenID* für das Verknüpfen der Bestellungen mit Kundendaten anhand der KundenID [38]. (Man könnte hier auch Synonyme wie "Zusammenführen" oder "Verbinden" statt "Joinen" verwenden – wichtig ist nur, das Muster einheitlich zu halten [39]).
- **Pivotieren/Entpivotieren:** `Pivotieren<Objekt>Nach<Attribut>` – z.B. *PivotierenUmsaetzeNachRegion* (Pivot auf Region) [40], und analog ein Unpivot-Schritt wie *EntpivotierenMonateNachWert* [41].
- **Sortieren:** `Sortieren<Objekt>Nach<Kriterium>` – z.B. *SortierenProdukteNachPreis* für das Sortieren einer Produktliste nach Preis [42]. Bei mehrstufigen Sortierungen kann man mehrere Kriterien nennen, z.B. *SortierenProdukteNachKategorieUndPreis* [43].

Diese Muster decken die gängigsten Transformationen ab und können sinngemäß auf weitere Aktionen übertragen werden [44]. Entscheidend ist die **Konsistenz im gesamten Projekt**: Verwenden Sie für die gleiche Aktion stets das **gleiche Verb** und dieselbe Wortwahl [45]. Wenn Sie z.B. einmal das Verb "Filtern" gewählt haben, nutzen Sie nicht an anderer Stelle "Selektieren" für die gleiche Art von Schritt [46]. Ebenso sollten unterschiedliche Aktionen durch unterschiedliche Verben klar unterscheidbar sein – nicht einmal "HinzufügenSpalte" und anderswo "ErstellenSpalte" für das gleiche Konzept [47]. Einheitliche Benennungen steigern die Eindeutigkeit und verhindern Verwirrung [45].

Neben den Schrittnamen selbst gilt die Namenskonvention **auch für Abfragen, Variablen, Parameter und Funktionen** [48] [49]. Wir empfehlen, alle solchen Bezeichner in **PascalCase** (Innerhalb des Namens jedes Wort mit Großbuchstaben beginnen, keine Leer- oder Sonderzeichen) zu schreiben [50]. Da M **case-sensitiv** ist, vermeiden einheitliche PascalCase-Namen mit eindeutiger Schreibweise viele Fehler [51] [52]. Außerdem umgehen wir so die unschöne `#"Step Name"`-Syntax, die nötig wird, wenn Leerzeichen im Namen wären [53]. **Keine unerklärlichen Abkürzungen:** Verwenden Sie ganze Wörter in natürlicher Sprache [54]. Etablierte Kürzel wie *ID* oder *Nr.* sind okay, aber

vermeiden Sie teamintern unbekannte Akronyme [55]. **Klarheit vor Kürze:** Ein längerer, aber selbsterklärender Name ist besser als ein kurzer, rätselhafter [56] – schließlich wird Code häufiger *gelesen* als geschrieben (Stichwort: *“Readability counts”* [56]).

Beispiel für konsistente Benennung: Angenommen, in mehreren Schritten wird schrittweise eine Kundenklassifizierung vorgenommen. Statt kryptischer Namen könnte man die Schritte nennen: *FilternKundenNachAktivität* → *BerechnenUmsatzJeKunde* → *KlassifizierenKundenNachUmsatz*. Schon die Namen zeigen den **roten Faden**: Alle betreffen Kunden und Umsatz und bauen aufeinander auf [57] [58]. Auch Hilfsschritte sollte man nicht einfach "Temp" oder "Schritt1" nennen, nur weil sie temporär sind – auch sie können einen sprechenden Namen erhalten, der ihren Zweck klärt (z.B. *BerechnenZwischenwert* statt *TempCalc* [59]). Oft machen optimal gewählte Schrittnamen Kommentare überflüssig [60], da die Logik schon aus der **Schrittfolge** gelesen werden kann [61]. Das minimiert die „WTFs pro Minute“ beim Code-Lesen [61].

Teamweite Standards: Wichtig ist, dass einmal etablierte Konventionen von allen Teammitgliedern konsequent eingehalten werden [62]. Ähnliche Dinge sollten *überall gleich* benannt sein. So können Kollegen den Code schneller „parsen“ und verstehen, weil sie sich an die Muster gewöhnen [63]. Es lohnt sich, untereinander die besten Bezeichnungen abzustimmen und vielleicht einen kleinen Styleguide oder eine Namensliste zu pflegen. Auf diese Weise wird die Power Query-Entwicklung konsistent und **teamfreundlich**.

4. Verwendung von Parametern und „Steuerabfragen“

Ein häufiger Anti-Pattern in BI-Projekten ist die **Hardcodierung** von Werten – z.B. Dateipfade, URLs, Filterwerte oder Schwellen – direkt im M-Code [64]. Solche fest verdrahteten Konstanten erschweren Wartung und Wiederverwendung: Ändert sich etwa der Dateipfad, muss man ihn in jeder Abfrage manuell anpassen [65] [66]. Die Lösung sind **Parameter** und **steuernde Abfragen** (Konfigurationsabfragen). Power BI Desktop bietet die Möglichkeit, Parameter zu definieren (über *Parameter verwalten* im Power Query-Editor) – etwa einen Parameter `pDatenpfad` für einen Dateipfad oder `pStichtag` für ein bestimmtes Datum [67] [68]. Ein solcher Parameter kann dann im Code referenziert werden, z.B. `File.Contents(pDatenpfad)` anstelle eines fest kodierten String-Pfads [68]. Ändert sich der Pfad oder soll die Abfrage auf eine andere Datei zeigen, genügt es, den Parameterwert zu ändern, anstatt jede einzelne Abfrage bearbeiten zu müssen [65]. Das erhöht die Flexibilität und Wartbarkeit enorm [66]. Microsoft bezeichnet den Einsatz von Parametern als Best Practice, um **dynamische und flexible Abfragen** zu erstellen [69]. Parameter dienen als zentrales, einfach zu änderndes Speicher für Werte, die an vielen Stellen genutzt werden [69] [70] – sei es als Argument für Schritte (z.B. Filtergrenzen) oder als Input für benutzerdefinierte Funktionen [71].

Neben einfachen Parametern (die einzelnen Werte entsprechen) kann man auch **Konfigurations- oder Steuerabfragen** einsetzen – das sind Abfragen, die z.B. eine ganze Tabelle mit Konfigurationswerten oder Mapping-Informationen liefern. In vielen Projekten findet man etwa eine Query *Parameter* oder

Konfiguration, die Schlüssel-Wert-Paare enthält (z.B. eine Tabelle mit Einträgen wie `{"URL_API", "https://api.../"}`) [72] [73]. Über Funktionen wie `Record.Field` oder per Merge kann man diese Konfigurationswerte dann in allen Abfragen nutzen [73]. Dadurch sind z.B. URLs, Dateinamen, Schwellenwerte etc. **zentral an einer Stelle gepflegt**, statt über den Code verstreut [72]. Genauso kann man bestimmte **Mapping-Tabellen** (für z.B. Übersetzungen von Codes, Gruppierungen oder Kategorien) als eigene Query halten und bei Bedarf heranziehen [74]. Ein gängiges Muster: Statt überall im Code verschachtelte `if/else`- oder `switch`-Logik zu haben, lagert man die Zuordnung in eine Tabelle aus und führt einen **Merge** mit dieser Mapping-Tabelle durch [74]. So ersetzt man komplizierte Code-Logik durch Daten, was Übersicht und Änderbarkeit verbessert (neue Fälle fügt man einfach der Tabelle hinzu, ohne Code zu ändern).

Keine "Magic Numbers": In diesem Zusammenhang ein wichtiger Clean-Code-Aspekt: Verwenden Sie keine bedeutungsvollen Zahlen oder Strings unkommentiert im Code [75]. Wenn ein Wert eine semantische Bedeutung hat (z.B. `7` als Wochentagezahl, oder `"DE"` als Ländercode für Deutschland), sollte er entweder als **Name** (Konstante/Parameter) verwendet werden oder aus einer Konfiguration stammen, nicht als nackte Literalzahl oder -text im Code stehen [75]. Das erhöht die Verständlichkeit – jemand der den Code liest, sieht dann `MaxTage = 7` oder nutzt `LandCode = "DE"` aus einer Lookup-Tabelle, was selbsterklärend ist, statt dass überall einfach `7` oder `"DE"` auftaucht.

Parameter richtig strukturieren: Benennen Sie Parameter genauso sorgfältig wie Abfragen nach dem Namensschema [76] [49]. Also zum Beispiel `StichtagDatum` statt nur `Datum` oder gar `x` [77]. Verwenden Sie auch hier konsistente Sprache (nicht `StartDate` auf Englisch und woanders `EndeDatum` auf Deutsch mischen) [49]. Manche Entwickler nutzen Präfixe wie `p` für Parameter (z.B. `pPfad`, `pStichtag`) – das ist Geschmackssache [78]. Wichtiger ist, dass der Name den Inhalt klar beschreibt. In Power BI Desktop werden definierte Parameter in der Abfrageübersicht unter "Parameter" geführt; man kann sie auch in Ordnern organisieren. Ein einheitliches Präfix *ParameterXYZ* oder ein eigener Ordner "Parameter" kann helfen, solche Queries sofort zu erkennen [79] [80]. Markieren Sie Parameter-Queries als *"Laden deaktivieren"*, damit sie nicht als eigenständige Tabelle ins Datenmodell übernommen werden (sie dienen ja nur als Hilfsobjekte) [81].

Steuerabfragen organisieren: Ebenso lohnt es sich, **Hilfsabfragen** wie Konfigurations- und Mapping-Queries in einem eigenen Ordner oder mit Präfixen zu versehen (z.B. alle Abfragen, die nur für Nachschlagetabellen da sind, in einem Ordner "Lookup" gruppieren). In komplexeren BI-Modellen gruppiert man oft die Abfragen thematisch: z.B. einen Ordner "Parameter" für Parameter und Config-Queries, einen Ordner "Funktionen" (siehe nächster Abschnitt), vielleicht "Staging" für Zwischenergebnisse und "Dimensionen"/"Fakten" für die finalen Outputs [82]. Diese organisatorische Struktur in der Abfragenliste ist **Gold wert für Teamverständlichkeit** – ein neuer Kollege sieht sofort, wo er beispielsweise die Parameter findet und welche Queries nur Hilfszwecken dienen [79] [80].

Anwendungsbeispiel Parameter: Stellen wir uns vor, wir haben einen

```
Parameter `pStichtag` (vom Typ Datum). In einer Abfrage wollen wir alle
Datensätze ab diesem Stichtag filtern. Statt das Datum fest einzutragen,
nutzen wir:

```m
GefilterteDaten = Table.SelectRows(Quelle, each [Datum] >= pStichtag)
```

Wechselt der Stichtag, ändert man einfach den Parameterwert, und alle abhängigen Abfragen adaptieren sich beim nächsten Refresh. Ähnlich könnte man einen Parameter `pDateipfad` definieren und in `File.Contents(pDateipfad)` nutzen – beim Wechsel der Datei nur dort den Pfad anpassen. Diese **Zentralisierung von Steuergrößen** macht Lösungen robust und portabel [83]. Auch für **Verbindungen** empfiehlt Paul Turley, stets Parameter für Server- oder Dateipfade zu verwenden, um die Lösung portabler zu machen (z.B. einfacher von Entwicklung auf Produktion umstellen zu können) [84] [85].

Zusammengefasst: Nutzen Sie Parameter, wo immer Werte mehrfach vorkommen oder sich ändern könnten, und setzen Sie auf Steuerabfragen für Konfiguration und Mapping. Dies macht den Code **dynamischer, besser wartbar und teamfreundlich**, da Einstellungen an einem Ort vorgenommen werden können und kein Suchen & Ersetzen im Code nötig ist [65] [83].

5. Robuste benutzerdefinierte Funktionen mit Typdefinitionen

Power Query M ist eine **funktionale Sprache**, d.h. Benutzer können eigene Funktionen definieren, um Logik wiederzuverwenden [86]. Wann immer Sie bemerken, dass Sie ein bestimmtes Transformationsmuster mehrfach benötigen oder komplexe Berechnungen aus dem Hauptquery auslagern möchten, ist eine **benutzerdefinierte Funktion** (Custom Function) sinnvoll. Eine Funktion kapselt eine **Aufgabe oder Berechnung** und kann mit unterschiedlichen Eingabewerten immer wieder aufgerufen werden – das folgt dem DRY-Prinzip und der Separation of Concerns (jedes Stück Code hat eine definierte Zuständigkeit).

Funktionen richtig schreiben: Orientieren Sie sich an Clean-Code-Grundsätzen wie "eine Funktion – eine Aufgabe". In M heißt das: Eine Funktion sollte nicht unübersichtlich groß sein oder sehr heterogene Dinge tun [14] [87]. Lieber mehrere kleine, fokussierte Funktionen (die man bei Bedarf verschachteln kann) als ein monolithischer Klotz [88] [87]. Kleine, klar fokussierte Funktionen sind leichter zu testen und zu warten [88]. Zum Beispiel statt einer Mega-Funktion, die zugleich Filter, Berechnungen und Formatierungen vornimmt, lieber drei spezifische Funktionen: etwa `fnFilterGueltigeDaten()`, `fnBerechneKennzahl()` und `fnFormatiereErgebnis()` – diese könnten nacheinander angewendet werden.

Typdefinitionen nutzen: Power Query M ist schwach typisiert, aber man kann Funktionen und Parametern *Typen* zuweisen, um den Code selbst-dokumentierend und weniger fehleranfällig zu machen. Das erfolgt durch Angabe von `as <Type>` nach Parametern und nach dem Funktionskörper für den Rückgabewert. Beispiel einer einfachen Funktion mit Typen:

```
// Berechnet einen Rabattbetrag basierend auf Umsatz und Prozentsatz
(fnBerechneRabatt as function) =
    (Umsatz as number, RabattProzent as number) as number =>
    let
        Betrag = Umsatz * RabattProzent
```
```

```
    in
        Betrag
```

Hier haben wir definiert, dass `Umsatz` und `RabattProzent` Zahlen (Number) sein müssen und die Funktion ebenfalls eine Zahl zurückgibt. Typdefinitionen wirken wie eine **eingebaute Dokumentation** – ein Teammitglied sieht sofort, welche Datentypen erwartet werden, und Tippfehler oder falsche Datentypen können früher auffallen. Außerdem unterstützen Typen z.B. IntelliSense in einigen Editoren und können bei komplexeren Abfragen helfen, Fehler zu vermeiden.

Ein weiteres Beispiel: Eine Funktion, die das Alter eines Kunden in Jahren berechnet, könnte so definiert werden:

```
(fnBerechneAlter) = (Geburtsdatum as date) as number =>
    Date.Year(DateTime.FixedLocalNow()) - Date.Year(Geburtsdatum)
```

Jetzt ist klar, dass `Geburtsdatum` ein Datum sein muss und die Ausgabe eine Zahl (Alter) ist.

Funktionen in Queries organisieren: In Power BI Desktop erscheinen Funktionen als spezielle Abfragen (mit fx-Icon). Man kann – und sollte – sie in der Abfrageübersicht in einem eigenen Ordner (z.B. "Funktionen") sammeln [89]. Setzen Sie diese Funktions-Abfragen ebenfalls auf "Nicht laden" [81], damit sie nicht als Tabellen ins Modell geladen werden, sondern nur im Hintergrund als Funktionsbibliothek dienen. Das Schöne an M ist, dass Funktionen in separaten Queries liegen können und somit den **Datenfluss nicht stören** [89] [90]. Statt eine Funktion mitten im Code klobig zu definieren, kann man sie separat auslagern – das hält die Hauptabfragen schlanker. Laut Best-Practice-Empfehlungen sollten *wiederverwendbare Funktionen stets in eigenen Abfragen* ausgelagert werden [91]. Diese können bei Bedarf von allen Queries aufgerufen werden und tauchen in der GUI nicht als Schritte auf, was die Übersicht erhöht [92].

Namensgebung für Funktionen: Verwenden Sie auch hier klare Namen, idealerweise in PascalCase und beschreibend. Manche Entwickler nutzen das Prefix `fn` für Funktions-Abfragen (z.B. `fnBerechneAlter`) [93]. Alternativ kann man Funktionsabfragen auch mit Kategorien im Namen versehen, etwa `DatumBerechneAlter` oder `TextBereinigenSonderzeichen` – je nachdem, was die Team-Konvention ist. Wichtig ist Einheitlichkeit und Klarheit darüber, was die Funktion tut. Die Parameter innerhalb der Funktion sollten ebenfalls sprechende Namen haben (statt `Value1, Value2` lieber `KundenUmsatz, KundenSeitJahren` etc. [94]).

Wiederverwendung und Team-Bibliothek: Gerade in größeren Teams oder Projekten lohnt es sich, häufig benötigte Funktionen zentral vorzuhalten. Einige Teams erstellen eine Art **„M-Cookbook"** – eine PBIX- oder PQ-Datei, die eine Sammlung nützlicher Funktionen (z.B. für Datumstabellen, Feiertagsberechnung, Textbereinigung, API-Aufrufe etc.) enthält [95]. Diese Bibliothek kann versioniert und bei neuen Projekten importiert werden [96]. So müssen nicht alle das Rad neu erfinden, und man fördert **Standardisierung**. Wenn jede*r im Team dieselben getesteten Funktionen nutzt, erhöht das die Zuverlässigkeit und Einheitlichkeit der Lösungen [95].

Zusammengefasst: Benutzerdefinierte Funktionen machen den M-Code **modular und wiederverwendbar**. Nutzen Sie sie, um duplizierte Logik zu vermeiden und komplexe Berechnungen aus dem Hauptablauf auszulagern. Durch **Typangaben** und klare Benennung werden Funktionen zu robusten Bausteinen, die in einer teamweiten Bibliothek geteilt werden können. So wird aus dem

einmal investierten Aufwand ein Mehrwert für alle (Stichwort DRY). Und denken Sie daran – auch Funktionen sollten eine Sache gut machen und nicht zu Monster-Prozeduren wachsen [14] [87].

6. Best Practices für strukturiertes M-Coding (let/in, Kommentare, Schritt-Struktur)

Neben der großen Architektur gibt es viele **kleinere Best Practices**, die den Code in Power Query M lesbarer und teamtauglicher machen. Hier einige wichtige Punkte:

- **Verwenden Sie `let … in`-Blöcke konsistent:** Jeder Query sollte als einzelner `let`-Ausdruck strukturiert sein, der alle Schritte als **benannte Variablen** definiert, und mit einem `in` das Endergebnis referenziert. Dadurch ist klar, dass es eine sequentielle Folge von Transformationen gibt. Vermeiden Sie es, zu viele Verschachtelungen zu nutzen – normalerweise reicht ein `let` pro Abfrage. Falls Sie Hilfsberechnungen kapseln wollen, bieten sich eher separate Abfragen oder Funktionen an, anstatt innerhalb eines Schritts nochmal einen `let` zu öffnen (das könnte die Lesbarkeit erschweren, außer es ist wirklich nötig).

- **Schrittanzahl optimieren vs. Lesbarkeit:** Es gibt oft mehrere Wege, in M etwas zu erreichen. Aus **Clean-Code-Sicht** ist es besser, mehr kurze, klar benannte Schritte zu haben als einen einzigen Schritt mit einer Mammut-Formel. Z.B. statt eine extrem lange Bedingung innerhalb von `Table.SelectRows` zu schreiben, kann man erst in einem Schritt relevante Daten vormerken und im nächsten Schritt filtern – das erleichtert das Verständnis [97]. Ebenso kann es sinnvoll sein, komplexe Berechnungen in Zwischenschritte aufzuteilen (erst Teilberechnungen durchführen, dann im letzten Schritt kombinieren). Das Motto lautet: **„Sparse is better than dense“** – also lieber etwas verteilter Code, der leicht zu lesen ist, als dichter Code, der alles in einer Zeile macht [98]. Natürlich sollte man nicht übertreiben: unnötige Schritte (z.B. fünf separate „Changed Type“-Schritte hintereinander, die man in einem Schritt hätte machen können) kann man am Ende zusammenfassen, um keinen Ballast zu haben [99] [100]. Die Query sollte effizient sein, aber nicht zu Lasten der Verständlichkeit optimiert werden. Finden Sie hier einen **guten Kompromiss** zwischen Klarheit und Kompaktheit.

- **Einrückungen und Zeilenumbrüche:** Formatieren Sie den M-Code sauber. Lange Funktionsaufrufe oder verschachtelte Ausdrücke können über mehrere Zeilen gebrochen werden. Beispielsweise bietet es sich an, bei Aufrufen wie `Table.SelectRows` die `each`-Bedingung in eine neue Zeile zu setzen, oder bei `Table.TransformColumns` jede Spalten-Transformation ins nächste Line zu schreiben. Die **visuelle Struktur** hilft, den Code zu scannen [101]. Ein gängiges Muster: Nach dem `let` alle Schritte auf gleicher Einrückungsebene untereinander, das `in` auf gleicher Höhe wie `let`, und die finale Variable danach. Innerhalb langer Formeln (etwa einer verschachtelten If-Then-Else-Logik) kann man mit zusätzlichen Einrückungen arbeiten, um die Blöcke klar zu zeigen. Vermeiden Sie horizontales Scrollen im Code-Editor – wenn eine Zeile zu lang wird, teilen Sie sie. **Kommentar:** Die PDF-Empfehlung sprach davon, Zeilen nicht „horizontal explodieren“ zu lassen, sondern zu segmentieren [102] – das ist genau der Punkt: Gut gesetzte Zeilenumbrüche verhindern zu dichte Textblöcke und erleichtern das Lesen.

- **Kommentare einsetzen (aber sparsam):** Kommentare sind in M mit `//` für einzelne Zeilen oder `/* ... */` für Blöcke möglich. Nutzen Sie Kommentare, um ungewöhnliche Tricks, schwierige Logik oder wichtige Annahmen zu erläutern. Auch eine Header-Kommentarsektion am Anfang der Abfrage kann hilfreich sein (z.B. wer die Abfrage erstellt hat, Datum, kurzer

Verwendungszweck). Allerdings sollten Kommentare **nicht als Ausrede für schlechten Code** dienen. Ideal ist, wenn die Schrittnamen und Struktur so selbsterklärend sind, dass kaum Kommentare nötig sind [60]. Kommentare können aber nützlich sein, um größere Abschnitte zu gliedern (etwa `// --- Dimensionstabelle erstellen ---`) oder komplexe Berechnungsschritte einzuleiten. Setzen Sie sie bewusst und vermeiden Sie Redundanz ("// Filtere Tabelle nach Datum XY" als Kommentar, wenn der Schrittname schon *FilternDatenNachDatumXY* lautet, ist überflüssig). In Teamumgebungen können deutschsprachige Kommentare sinnvoll sein, sofern das Team deutsch kommuniziert – wichtiger ist Einheitlichkeit.

- **Schritte sinnvoll benennen und gruppieren:** Wie in Abschnitt 3 ausführlich beschrieben, ist Schrittbenennung entscheidend. Geben Sie *jedem* Schritt einen eindeutigen Namen, auch Hilfsschritten. Wenn bestimmte Schritte logisch zusammengehören (z.B. erst *FilternBestellungenNachDatum*, dann *GruppierenBestellungenNachKunde*), kann man sie durch ein gemeinsames Objekt im Namen (hier "Bestellungen") kenntlich machen. Das schafft visuelle **Gruppierung** und erleichtert das Verfolgen des Datenflusses [103] [57]. Für sehr komplexe Queries kann man auch mit **Abschnittskommentaren** arbeiten, um Blöcke zu markieren.

- **Keine doppelten Aktionen:** Vermeiden Sie es, denselben Schritt mehrmals zu machen. Z.B. nicht erst Datentyp ändern, dann wieder als neuen Schritt Datentyp ändern (es sei denn, es ist wirklich nötig). Oft entstehen solche Duplikate bei wiederholter GUI-Benutzung. Besser ist, solche **redundanten Schritte zu konsolidieren** – z.B. alle Spaltenumbenennungen in einem Schritt sammeln oder zumindest benachbarte Rename-Schritte zusammenführen [99]. So bleibt die Schritteliste klar und schlank.

- **Query Folding im Hinterkopf behalten:** Für performance-sensitive Queries sollte man wissen, welche Schritte Query Folding brechen (z.B. bestimmte benutzerdefinierte Spalten, Merges zwischen unterschiedlichen Quellen, etc.). Eine Best Practice ist: *Foldable Schritte zuerst, Non-Foldable so spät wie möglich*. D.h. Filter, Projektion (Spaltenauswahl) usw. kommen idealerweise vor z.B. einem Merge oder komplexen Berechnungen, um maximal von der Datenbank ausführen zu lassen [104] [105]. Wenn ein Schritt Folding verhindert, möglichst danach keine weiteren Heavy-Lifting-Transformationen mehr einbauen, die dann lokal ausgeführt würden – oder diesen Schritt ggf. in die Datenquelle verlagern. Tools wie der Abfrageplan oder die „Native Query“ Anzeige helfen, das zu prüfen. Aber all das ändert nichts daran, dass **Klarheit vor Optimierung** geht: Schreiben Sie keinen obskuren Code nur für Folding, dokumentieren Sie lieber, warum eine bestimmte Reihenfolge eingehalten wird, falls es wichtig ist.

- **Datenqualität und Fehlerbehandlung:** Im Sinne von robustem Code sollte man darauf achten, gängige Fehlerquellen abzufangen. Beispielsweise, wenn man mit `Merge` arbeitet, sicherstellen, dass man nach dem Expandieren die passenden Zeilen für Nicht-Matches behandelt (z.B. mit `Table.NestedJoin` inkl. JoinKind und anschließendem Null-Handling). Oder nach einer Typänderung mögliche Fehlerzellen (Error) behandeln, damit Refreshes nicht plötzlich fehlschlagen, wenn mal ein unparsebarer Wert auftaucht. M erlaubt z.B. mit `Try ... Otherwise` Fehler abzufangen. Diese Dinge gehören zur **strukturierten Entwicklung**, damit die Abfrage nicht nur bei Sonnenschein-Daten läuft. Microsoft rät dazu, Queries *zukunftssicher* zu gestalten – z.B. durch Entfernen unerwarteter Spalten oder Umgang mit dynamischen Schemas [106] [107]. Als Beispiel: Statt sich darauf zu verlassen, dass keine neuen Spalten auftauchen, lieber gezielt benötigte Spalten auswählen (*Choose Columns*), um gegen zusätzliche Spalten immun zu sein [108] [107]. Solche defensiven Techniken erhöhen die Zuverlässigkeit.

Insgesamt folgen diese Best Practices einem Motto: **Schreibe Code, den ein anderer (oder du selbst in 6 Monaten) sofort versteht.** Formatierung, Kommentare, sinnvolle Schrittfolgen und konsistente Namen sind keine Kosmetik, sondern wesentliche Faktoren, um Power Query-Lösungen teamfähig zu machen [109] [110]. Denken Sie daran, dass BI-Projekte oft von mehreren Personen gepflegt werden und Datenmodelle länger leben – ein sauber strukturierter M-Code zahlt sich hier in weniger Bugs und schnellerem Onboarding neuer Entwickler aus.

7. Wiederverwendbare Strukturen: Templates und Vorlagen

Neben allgemeinen Prinzipien ist es hilfreich, bestimmte **wiederkehrende Muster als Templates** parat zu haben. Das können Vorlagen für typische Aufgaben sein – z.B. **Datei-Import, dynamisches Laden mehrerer Dateien, Stammdaten-Abfragen, Dimensionstabellen** usw. Solche Templates stellen sicher, dass man nicht jedes Mal bei Null anfängt und dass bewährte Strukturen wiederverwendet werden.

Projektstruktur-Templates: Für umfangreichere Modelle hat es sich bewährt, die Abfragen in einer **einheitlichen Ordnerstruktur** zu organisieren. Ein mögliches Template für ein Power BI Datenmodell:

- *Ordner "Parameter":* Enthält alle Parameter-Queries und ggf. eine Konfigurations-Tabelle. Z.B. `ParameterStichtag`, `ParameterDateipfad`, `ParameterAPI_Key` etc. Wie zuvor erwähnt, kann man Parameter-Queries mit Präfix oder im Ordner sofort erkennbar machen [111].
- *Ordner "Funktionen":* Enthält alle benutzerdefinierten Funktions-Queries (fx). Z.B. `fnBerechneAlter`, `fnTextBereinigen` etc. Dies kapselt wiederverwendbare Logik.
- *Ordner "Staging" (oder "Rohdaten"):* Hier liegen Queries, die die **rohen Datenquellen** anzapfen und ggf. erste leichte Transformationen durchführen. Oft pro Datenquelle oder pro Tabelle eine Query, benannt nach der Quelle, z.B. `Quelle_ProdukteRoh` oder `Staging_Produkte` für die rohen Produktdaten.
- *Ordner "Transform" (optional):* In manchen Fällen fügt man eine weitere Schicht ein, etwa wenn man mehrere Rohabfragen kombiniert. Oft kann man Staging und Transform auch zusammenfassen. Wichtig ist: Hier passiert die Haupt-Logik der Bereinigung und Transformation, aber es wird noch nichts als endgültige Tabelle deklariert.
- *Ordner "Data Model" / "Endgültig":* Enthält die **finalen Abfragen**, die ins Modell geladen werden. Diese könnten z.B. den Fakten- und Dimensionsabfragen entsprechen: `Produkt` (bereit für Modell), `Kunde`, `UmsatzFaktentabelle` etc. Jede finale Query sollte idealerweise aus den vorbereitenden Staging-Abfragen referenzieren, nicht direkt aus der Quelle (so nutzt man das Cache und DRY-Prinzip) [112] [113].

Ein solches Template kann natürlich je nach Projekt angepasst werden, aber der Kern ist: **eine klare Trennung der Bereiche und Verantwortlichkeiten**. Wenn ein neues Projekt startet, kann man dieses Grundgerüst klonen und hat direkt eine Ordnung, an die sich alle halten. Einige Organisationen erstellen interne **Power BI Starter-Kits**, wo z.B. eine PBIX mit Beispielordnern, Parametern und DAX- sowie M-Templates bereitgestellt wird.

Template: Dateiimport mit Transformationslogik: Ein häufiger Anwendungsfall ist das Laden von Dateien (Excel, CSV) und deren Verarbeitung. Hier ein Muster, das man immer wieder ähnlich nutzen kann:

1. **Parameter für Pfad/Dateinamen:** Erstellen eines Parameters `pDateipfad` (Text) für den Speicherort der Datei. Damit ist der Query flexibel, falls der Pfad sich ändert.
2. **Quell-Query (Raw):** Abfrage `QuelleDatei` (Staging) nutzt `Excel.Workbook(File.Contents(pDateipfad))` bzw.

`Csv.Document(File.Contents(pDateipfad))`, um den Dateiinhalt zu lesen. Bei Excel: navigiert ggf. zum richtigen Sheet oder Tabelle. Ergebnis ist noch untransformiert.

3. **Transformations-Query:** Abfrage `TransformierteDaten` referenziert `QuelleDatei` und führt alle nötigen Bereinigungen durch: Header promoten, Datentypen setzen, irrelevante Zeilen rausfiltern, Spalten umbenennen etc. Diese Query gibt saubere, aber noch unverarbeitete Daten zurück.
4. **Logik-Query (optional):** Falls komplexe Berechnungen/Aggregationen nötig sind, könnte eine weitere Abfrage `LogikErgebnisse` die `TransformierteDaten` referenzieren und z.B. neue Spalten anfügen, Berechnungen durchführen, gruppieren usw.
5. **Finale Ausgabe:** Abfrage `FertigTabelle` (oder treffender Name, z.B. `Umsatzberichte`) referenziert entweder `TransformierteDaten` (wenn keine extra Logik-Query) oder `LogikErgebnisse` und bereitet das finale Layout: z.B. nur relevante Spalten auswählen, sortieren, ggf. zusammenführen mit anderen Tabellen, damit es ins Datenmodell passt.

In kleineren Fällen kann man Schritt 3–5 in einer Abfrage kombinieren, aber das Prinzip bleibt: von Parameter -> Quelle -> transformieren -> Ergebnis. Ob man das als ein Query oder mehrere macht, hängt von Komplexität und Wiederverwendbarkeit ab. Ein Template in Codeform für Excel-Import haben wir bereits im Abschnitt 2 demonstriert (dort in einer Abfrage alles, aber sauber in Schritte gegliedert).

Template: Ordner mit Dateien importieren: Ein komplexeres, aber gängiges Muster ist das Importieren aller Dateien aus einem Ordner (etwa monatliche CSVs konsolidieren). Hier etabliert sich folgender Pattern: - Eine **Funktion** `fnTransformDatei(file as binary) as table`, die den Inhalt *einer* Datei transformiert (im Prinzip die Logik, die man für eine einzelne Datei braucht, als Funktion). - Eine **Parameter-Query** oder einfache Query, die den Ordnerpfad enthält. - Eine **Hauptabfrage**, die den Ordnerpfad liest, alle Dateien im Ordner auflistet (`Folder.Files`), evtl. auf relevante Dateien filtert (z.B. nach Erweiterung oder Namensmuster), dann mittels `Table.AddColumn(..., each fnTransformDatei([Content]))` die Funktion auf jede Datei anwendet, und schließlich alle Ergebnis-Tabellen mit `Table.Combine` zusammenführt. - Dieses Muster kann man sehr gut als Template vorbereiten, da es oft vorkommt. Microsoft hat dafür auch *Power Query Templates* (im Kontext von Datenflows/Fabric) eingeführt, aber im Grunde kann man sich so etwas auch manuell bauen.

Vorlagen für Dimensionstabellen oder Berechnungen: Ein weiteres Beispiel: die **Datumsdimension**. Statt in jedem Projekt neu zu überlegen, wie man eine Kalender-Tabelle baut, kann man eine parametrisierte Funktion `fnErzeugeKalender(Start as date, Ende as date) as table` vorhalten. Diese erstellt eine Tabelle mit allen Daten zwischen Start und Ende und berechnet Feiertage, KW, etc. Dann in jedem neuen Bericht einfach diese Funktion nutzen. Solche Vorlagen beschleunigen die Entwicklung und stellen sicher, dass erprobte Logik konsistent bleibt.

Häufige Muster dokumentieren: Es ist sinnvoll, intern eine kleine Bibliothek oder Dokumentation von solchen Patterns zu pflegen – sei es in Confluence, OneNote oder als kommentierte PBIX. Wenn zum Beispiel ein Teammitglied ein elegantes Konstrukt für einen bestimmtes Problem gebaut hat (etwa ein flexibles Unpivot für dynamische Spaltenanzahl), kann das als Code-Snippet festgehalten werden. Power Query ist ein Low-Code-Tool, aber dennoch gibt es immer wieder raffinierte Lösungen, die man teilen kann. Das fördert die **Kollektiv-Effizienz** des Teams.

Kurz gesagt: **Wiederverwendbare Strukturen** helfen, Rad-Neuerfindungen zu vermeiden. Nutzen Sie Templates für gängige Aufgaben, und scheuen Sie nicht, in neuen Projekten auf bestehenden Lösungen aufzubauen. Auch Microsoft selbst bietet über die Community und Doku Beispiele, z.B. für Parameter,

für den Einsatz von Templates in Datenflows etc., die man adaptieren kann. Diese Vorlagen sparen Zeit und sorgen für **Standardisierung** quer durch Projekte.

8. Typische Anti-Patterns und wie man sie vermeidet

Selbst erfahrene Entwickler tappen manchmal in Fallen – hier einige **häufige Anti-Patterns** in Power Query M und Vorschläge, wie man sie korrigiert:

- **Hardcodierte Werte:** Wie oben beschrieben, fest einprogrammierte Pfade, URLs oder Parameter (z.B. `Source = Sql.Database("DEVSERVER", "DB_Test")` direkt im Code). *Problem:* Wechsel der Umgebung oder Updates werden mühsam. **Lösung:** Parameter verwenden oder Konfigurationsabfrage nutzen. Z.B. Servernamen in einen Parameter auslagern, Filterwerte als Parameter definieren. Dadurch wird der Code dynamisch und anpassbar [64] [65].

- **Magische Konstanten:** Verstreute "Magic Numbers" oder Strings mit besonderer Bedeutung (z.B. `IF [Status] = "A"` an vielen Stellen für "active"). *Problem:* Unklar für Leser, schwer zu ändern (was heißt "A"?). **Lösung:** Konstante definieren oder Mapping-Tabelle nutzen. Z.B. Parameter `StatusAktiv = "A"` definieren und dann `IF [Status] = StatusAktiv` schreiben, oder eine kleine Query `StatusMapping` die z.B. {"A","Aktiv"} enthält und per Join Klartext liefert. So ist die Intention klar und Änderungen zentral möglich [75].

- **Default-Schrittnamen unverändert lassen:** Schritte heißen noch "Geänderter Typ", "Entfernte Spalten1", etc. *Problem:* Null Kontext, späteres Verständnis schwierig [24]. **Lösung:** Alle Schritte aussagekräftig umbenennen [25]. Siehe Abschnitt 3: nach Namensschema benennen, Operation + Kontext beschreiben. Das kostet minimal mehr Zeit beim Erstellen, spart aber enorm viel Zeit beim Warten und Debuggen [26] [109].

- **Uneinheitliche Benennung / Sprache mixen:** Mal deutsche, mal englische Schritt- oder Abfragenamen; mal Singular, mal Plural, mal Abkürzungen. *Problem:* Wirkt chaotisch, Teammitglieder müssen mehr überlegen, ob z.B. "Cust" wohl "Customer" bedeutet. **Lösung:** Einheitliche Namenskonvention für das Projekt festlegen (möglichst in der Muttersprache des Teams, hier Deutsch, um keine Sprachbarriere zu haben) [49]. Keine nichtssagenden Kürzel, keine wechselnden Begriffe für dasselbe (z.B. immer "Hinzufügen" statt mal "Einfügen") [45]. **Styleguide** gemeinsam definieren und konsequent einhalten.

- **Monolithische Queries ohne Referenzierung:** Eine einzige Abfrage macht alles – liest zig Quellen, vereinigt sie, transformiert, berechnet, erstellt am Ende zig Zwischenergebnisse. *Problem:* Sehr schwer wartbar, kein Wiederverwenden von Teil-Ergebnissen, Performance leidet (weil evtl. gleiche Teilberechnungen mehrfach stattfinden). **Lösung:** Abfrage in sinnvolle Teilabfragen aufsplitten (Staging, Berechnung, Final) und mit Query-Referenzen arbeiten [2] [3]. Auch ruhig mal *Extra-Queries* als Hilfsqueries einführen – viele zögern, Hilfsabfragen anzulegen, aus Angst das Modell zu überfrachten, aber nicht jede Query muss geladen werden. "Keine Angst vor Extra-Queries" kann man hier sagen: Sie erhöhen die Übersicht und dank *Disable Load* muss das Modell nicht leiden [114].

- **Doppelter Code (DRY-Verstoß):** Ähnliche oder gleiche Transformationen werden in mehreren Abfragen separat gemacht (oft per Copy-Paste). *Problem:* Änderungsaufwand doppelt, Inkonsistenzen möglich. **Lösung:** Gemeinsame Logik in **eine Basis-Abfrage oder Funktion** auslagern und von den verschiedenen Stellen referenzieren [113] [12]. Z.B. statt in zwei Queries jeweils die Produktdaten zu bereinigen, eine *Produkt_Bereinigt* Query erstellen und beide nutzen

diese als Quelle [113]. Oder statt einen komplizierten Regex zwei Mal zu schreiben, eine Funktion `fnParseXYZ` definieren und zweimal aufrufen. Damit hält man die *Single Source of Truth* und reduziert Fehler.

- **Frühes Brechen von Query Folding:** Komplexe oder nicht-foldbare Schritte werden zu früh ausgeführt (z.B. benutzerdefinierte Funktion auf jede Zeile anwenden, bevor man filtert). *Problem:* Performanceeinbruch, da Rest nicht mehr auf Datenbank ausgeführt wird. **Lösung:** Wenn möglich, Filter, Aggregationen und Joins, die foldable sind, *vor* solche Schritte ziehen [115] [105]. Nicht nötige Spalten entfernen, bevor man in M viel verarbeitet [116]. Und: Nur wenn wirklich nötig, auf native SQL zurückgreifen – oft kann man mit M-Mitteln und Abfrage-Faltung mehr erreichen. Falls man einen nicht-foldbaren Schritt braucht, idealerweise so spät wie möglich (nachdem Datenmenge minimiert wurde) einbauen [104].

- **Zu viele Spalten oder Zeilen laden:** Abfragen, die "SELECT * machen (alle Spalten) oder ungefiltert riesige Tabellen ziehen.* Problem: *Unnötige Datenmasse, langsam, unübersichtlich. Lösung: Nur notwendige Spalten auswählen (Power Query: Start > Spalten auswählen) [116]. Ungebrauchte Spalten konsequent entfernen [116]. Früh filtern (z.B. nur relevante Zeiträume laden) [17]. Für große Tabellen evtl. mit Parametern (z.B. RangeStart/RangeEnd für Datum) arbeiten, um die Datenmenge zu steuern [117]. Kurz: Daten-Diät halten**, das verbessert Performance und Übersicht.

- **Kein Setzen von Datentypen:** Einige belassen Spalten im Typ "Any" (beliebig) oder löschen den automatischen `Changed Type`-Schritt. *Problem:* Kann zu Problemen führen, z.B. bestimmte Transformationsmenüs stehen nicht zur Verfügung, und Inkonsistenzen, wenn später Daten kommen, die anders interpretiert werden [118] [119]. **Lösung:** Datentypen **immer explizit setzen**, möglichst direkt nach dem Einlesen bzw. nach dem Promoten der Header. So weiß man, was man hat, und Power BI kann effizienter arbeiten (Komprimierung im Modell hängt z.B. davon ab, ob eine Spalte als Zahl oder Text geladen wird). Falls Typumwandlungen zu Fehlervalueten führen (z.B. "ABC" in einer Zahlenspalte), sollte man diese Fehler gezielt behandeln (z.B. filtern oder ersetzen), statt unbemerkt drin zu lassen [120] [121].

- **Keine Dokumentation/Meta-Info:** Abfragen ohne Beschreibungen, keine Kommentare, keine Struktur – der nächste im Team muss alles durchklicken, um zu verstehen. *Problem:* Hoher Einarbeitungsaufwand, Fehleranfällig bei Übergabe. **Lösung:** Siehe nächster Abschnitt – **Dokumentation & Visualisierung**. Also: Beschreibungen im Abfrageeditor pflegen, ggf. externe Doku.

Natürlich ist jedes Projekt anders, aber diese Anti-Patterns tauchen sehr häufig auf. Sie zu vermeiden, bedeutet in Summe: **Klarheit, Einfachheit, Konsistenz** anstreben. Lieber vorab einen Moment überlegen ("Gibt es das schon irgendwo? Sollte das ein Parameter sein?") als später stundenlang suchen, warum der Code so schwer zu ändern ist. Ein guter Test ist: Wenn man selbst den eigenen Query zwei Wochen nicht gesehen hat – versteht man auf Anhieb wieder, was gemacht wurde? Falls nein, ist irgendwo ein Anti-Pattern verborgen .

9. Verständlichkeit und Übergabefähigkeit im Team (Dokumentation & Visualisierung)

Eine Power-Query-Lösung entfaltet ihren Wert oft erst, wenn mehrere Personen daran arbeiten oder sie langfristig betrieben wird. Deshalb ist **Verständlichkeit und Übergabefähigkeit** ein zentrales

Qualitätsmerkmal. Folgende Maßnahmen helfen, dass Ihre M-Queries auch im Team leicht verstanden, gewartet und weiterentwickelt werden können:

- **Selbstdokumentierender Code:** Der erste Schritt zur Verständlichkeit ist all das, was wir oben ausgeführt haben – *sprechende Namen*, *klare Struktur*, *Modularität*. Wenn Sie diese Prinzipien beherzigen, ist Ihr Code bereits "selbst-dokumentierend". Ein neues Teammitglied kann oft allein aus den Abfrage- und Schrittnamen erkennen, was passiert [61]. Das reduziert den Bedarf, alles verbal zu erklären oder extra Doku zu wälzen.

- **Kommentare und Beschreibungen:** Ergänzen Sie den Code, wo nötig, mit Kommentaren (z.B. komplexe Berechnungen oder Workarounds kurz erläutern). Außerdem bietet Power Query die Möglichkeit, jeder Abfrage und sogar jedem Schritt eine **Beschreibung** zu hinterlegen (Rechtsklick auf Abfrage > Eigenschaften > Beschreibung, bzw. Rechtsklick auf Schritt > Beschreiben). Diese Beschreibungen sieht man, wenn man mit der Maus über den Schritt hovert, und sie sind für andere sehr hilfreich, um z.B. den Zweck einer Abfrage zu verstehen [122]. Sie könnten z.B. in die Beschreibung einer Abfrage schreiben: "Staging-Abfrage: Bereinigt Produktdaten und reichert Kategorieinfos an. Nicht laden." – so ist sofort klar, was Sache ist.

- **Query-Gruppen als visuelle Ordnung:** Nutzen Sie die Ordner/Gruppierungsfunktion im Abfragen-Fenster, um verwandte Queries zu bündeln [123]. Wie in Abschnitt 7 beschrieben, ordnen Sie z.B. alle Parameter zusammen, alle Dimensionstabellen zusammen etc. Diese Gliederung wirkt wie eine Inhaltsübersicht und erleichtert es, in einem PBIX mit vielen Abfragen den Überblick zu bewahren [124]. Sinnvolle Gruppen-Namen (z.B. "Parameter", "Staging", "Faktentabellen", "Dimensionstabellen") geben direkt Kontext [82]. Das fühlt sich banal an, ist aber in der Praxis enorm hilfreich, wenn jemand neu dazukommt.

- **Query-Abhängigkeitsansicht (Query Dependencies):** Power BI Desktop hat eine eingebaute Visualisierung der Abhängigkeitsbeziehungen zwischen Abfragen (Menü *Ansicht* > *Abfrageabhängigkeiten*). Diese zeigt in einem Diagramm, welche Abfrage von welcher abhängt (z.B. Parameter->Staging->Final) [125]. Für die Team-Dokumentation kann es nützlich sein, einen Screenshot dieser **Datenfluss-Visualisierung** ins Wiki oder Doku aufzunehmen, um neuen Entwicklern schnell zu zeigen, wie die Daten fließen. Gerade bei komplexen Modellen kann man so die Modularität sichtbar machen. Einige nutzen auch externe Tools oder Custom Scripts, um M-Code zu dokumentieren, aber oft reicht diese eingebaute Visualisierung und eine gut strukturierte Abfragenliste.

- **Team-Standards und Reviews:** Für Teamfähigkeit ist nicht nur der Code selbst, sondern auch der Prozess wichtig. Etablieren Sie gemeinsam **Standards** (wie Namenskonvention, Ordnerstruktur, Performance-Richtlinien). Machen Sie evtl. Code-Reviews im Team – ein frisches Paar Augen findet oft Unklarheiten, die man selbst übersieht. Diese Reviews stellen sicher, dass der Code für andere lesbar ist und nicht nur für den Autor. Fragen, die man sich stellen kann: "Würde mein Kollege verstehen, was ich hier gemacht habe? Habe ich irgendwo eine Annahme, die nur mir bekannt ist?" – falls ja, entweder Code verbessern oder Kommentar hinzufügen.

- **Übergabedokumentation:** Falls ein Projekt übergeben wird (z.B. an einen anderen Entwickler oder Kunde), lohnt es sich, eine kurze **Dokumentation** beizulegen. Darin kann z.B. stehen: welche Datenquellen angebunden sind, was die Hauptabfragen sind, welche Parameter es gibt und was angepasst werden muss beim Deployment (z.B. "Pfad XYZ muss in Parameter soundso gesetzt werden"). Auch Informationen über etwaige Workarounds (z.B. "API liefert max 1000 Zeilen, daher in Abfrage X Paginierung implementiert") sollten erwähnt werden. Im Idealfall

enthält der Code das schon als Kommentare, aber eine Zusammenfassung im Klartext schadet nicht – so wird die Hürde geringer, sich einzuarbeiten.

- **Visualisierung der Ergebnisse und Testbarkeit:** Während der Entwicklung und auch danach im Team ist es hilfreich, Teil-Ergebnisse überprüfen zu können. Durch die modulare Aufteilung kann man jede Stufe einzeln anschauen (z.B. Staging-Abfrage allein öffnen, um zu sehen ob Bereinigung passt). Encourage a practice of *"build and test in increments"* – also schrittweise Aufbauen und immer wieder prüfen. Das ist zwar kein Dokumentationspunkt per se, aber erhöht die Nachvollziehbarkeit, weil man Vertrauen in jede Stufe gewinnt. Man könnte auch spezielle **Prüf-Abfragen** bauen, die z.B. Row Counts oder Schema-Überprüfungen durchführen (z.B. eine Query, die die letzten Refresh-Datenstände ausgibt, um zu sehen ob alles aktualisiert wurde). Solche können für die Übergabe hilfreich sein, um dem Nachfolger zu zeigen: Schau, so kannst du prüfen, ob die Daten vollständig geladen wurden etc.

- **Bezug zu fachlicher Doku/Visuals:** Power Query bildet oft einen Teil der Lösung. Es ist sinnvoll, den Bogen zur fachlichen Seite zu spannen: z.B. kann man im Code referenzieren, auf welches Feld im Bericht er sich auswirkt ("// Dieser Schritt berechnet die Verkaufs-KPI, genutzt in Visual XY"). Oder umgekehrt in der Berichtsdokumentation vermerken, welche Query die Daten liefert. Diese Traceability macht es neuen Teammitgliedern leichter, Änderungen durchzuführen, weil sie wissen, wo sie ansetzen müssen.

Letztlich soll durch all diese Maßnahmen der Power Query-Code **transparent** werden. Das Ziel ist erreicht, wenn Teammitglieder ohne große Einarbeitung verstehen, was die Abfragen tun, und sich im Code sicher bewegen können. Ein gut strukturierter M-Code in einem BI-Projekt ist genauso wichtig wie sauberer Code in einer Softwareentwicklung – er spart Zeit, reduziert Fehler und erleichtert die **Zusammenarbeit im Team**.

Checkliste für gut strukturierte Power Query M-Lösungen

Zum Abschluss eine kompakte **Checkliste**, anhand derer man prüfen kann, ob eine Power Query-Lösung die beschriebenen Best Practices erfüllt:

- [ ] **Konsistente Namenskonventionen** werden eingehalten (PascalCase, Deutsch, Schema "AktionWasNachKriterium") [50] [29] . Keine Default-Namen oder kryptischen Abkürzungen vorhanden [24] [55] .
- [ ] **Modularität gegeben:** Abfragen sind nach ETL-Schichten aufgeteilt (Rohdaten/Staging, Transformation, Output) [3] [6] . Wiederkehrende Schritte sind in Referenz-Queries oder Funktionen ausgelagert (DRY-Prinzip) [12] [126] .
- [ ] **Parameter genutzt statt Hardcodierung:** Alle externen Pfade, URLs, sensiblen Filter etc. sind als Parameter oder in einer Konfigurationstabelle definiert [67] [68] . Keine Magic Numbers im Code – falls ja, sind sie als Konstanten benannt [75] .
- [ ] **Benutzerdefinierte Funktionen für Wiederverwendung:** Gemeinsame Logik ist in Funktionen gekapselt und nicht dupliziert in mehreren Abfragen [14] [126] . Funktionen haben Typdefinitionen für Parameter/Rückgabe, um Klarheit zu schaffen.
- [ ] **Saubere Schritt-Struktur:** Jeder Query hat einen logischen, chronologischen Ablauf: Quelle -> nötige Transformationen -> Ergebnis. Teure Operationen (Sort, Merges) kommen spät, Filter/Removes früh [17] [16] .
- [ ] **Schrittnamen und Kommentare:** Alle Schritte sind sinnvoll benannt (keine "Schritt1", "Geänderter Typ2" etc.) [24] . Wo notwendig, gibt es Kommentare oder Beschreibungen, die komplexe Logik erläutern. Schrittabfolgen eventuell durch Abschnittskommentare gegliedert.

- [ ] **Formatierung und Lesbarkeit:** M-Code ist ordentlich formatiert (Einrückungen konsistent, Zeilenumbrüche bei langen Ausdrücken) [101] . Keine überlangen Zeilen, die den Code schwer lesbar machen.
- [ ] **Query-Organisation:** Abfragen sind in sinnvollen Ordnern gruppiert (z.B. Parameter, Funktionen, Staging, Dimensionen, Fakten). Nicht benötigte Hilfsqueries sind auf *Nicht laden* gesetzt, so dass das Datenmodell sauber bleibt [81] .
- [ ] **Performance-Bewusstsein:** Unnötige Spalten werden entfernt [116] , unnötige Zeilen gefiltert, Query Folding wo möglich genutzt (prüfen: keine vermeidbaren Folding-Breaker an den Anfang setzen). Bei sehr großen Daten eventuell Parameter zum Laden eines Teilsets während Entwicklung (Stichwort: RangeStart/RangeEnd für Incremental Refresh) implementiert.
- [ ] **Fehlervermeidung:** Datentypen sind überall explizit gesetzt [118] . Mögliche Fehler (z.B. bei Typumwandlung oder Merge-Nulls) werden behandelt, so dass Refreshes nicht unerwartet scheitern.
- [ ] **Dokumentation & Team-Fit:** Abfragen/Schritte haben Beschreibungen wo nötig [27] . Die Query-Abhängigkeitsübersicht ergibt ein verständliches Bild (keine völlig unverbundenen, chaotischen Abhängigkeitsstränge). Kollegen können den Ablauf nachvollziehen, Begriffe sind im ganzen Projekt einheitlich verwendet.

Wenn die meisten Punkte mit "Ja" abgehakt werden können, hat man eine solide, *clean* Power Query M-Lösung vor sich, die sowohl technisch stabil als auch für Menschen gut verständlich ist.

Zum Abschluss sei betont: **Clean Code in Power Query** mag initial etwas mehr Disziplin erfordern (ein paar Minuten mehr beim Benennen und Strukturieren). Aber es zahlt sich zigfach aus – in Form von weniger Fehlern, glücklicheren Mitentwicklern und Lösungen, die auch morgen noch funktionieren und verständlich sind [56] [109] . In diesem Sinne: *Happy Querying!*

---

[1] [2] [3] [5] [9] [10] [11] [12] [13] [14] [23] [24] [25] [28] [29] [30] [31] [32] [33] [34] [35] [36] [37] [38] [39] [40] [41] [42] [43] [44] [45] [46] [47] [48] [49] [50] [51] [52] [53] [54] [55] [56] [57] [58] [59] [60] [61] [62] [63] [64] [65] [66] [67] [68] [72] [73] [74] [75] [76] [77] [78] [79] [80] [81] [82] [86] [87] [88] [89] [90] [91] [92] [93] [94] [95] [96] [97] [98] [101] [102] [103] [111] [112] [113] [114] Best Practice Guide_ Power Query M – Code-Struktur und Formatierung.pdf
file://file-9uvPkC535Lv5SLNCvtGqkp

[4] [6] [7] [8] [18] [19] [20] [21] [115] Power Query Optimization | Svitla Systems
https://svitla.com/blog/power-query-optimization/

[15] [26] [109] [110] Naming convention for Power Query steps | DAX Pro Services
https://daxproservices.com/naming-convention-for-power-query-steps/

[16] [17] [22] [27] [69] [70] [71] [83] [106] [107] [108] [118] [119] [120] [121] [122] [123] [124] [126] Best practices when working with Power Query - Power Query | Microsoft Learn
https://learn.microsoft.com/en-us/power-query/best-practices

[84] [85] [99] [100] [104] [105] [116] [117] Doing Power BI the Right Way: 4. Power Query design best practices – Paul Turley's SQL Server BI Blog
https://sqlserverbi.blog/2021/02/14/doing-power-bi-the-right-way-4-power-query-design-best-practices/

[125] Power Query Dependencies Viewer - Excelguru
https://excelguru.ca/power-query-dependencies-viewer/