Ausarbeitung zum Pipeline-Architekturstil

Alber Jonas, Schweitzer Tim

15. Mai 2025

Zusammenfassung

Diese Ausarbeitung befasst sich mit dem Pipeline-Architekturstil, auch bekannt als Pipes-and-Filters-Architektur. Es werden die grundlegenden Konzepte, Komponenten und Funktionsweisen dieses Stils erläutert. Darauf aufbauend werden typische Anwendungsbereiche und konkrete Beispiele vorgestellt. Abschließend erfolgt eine kritische Bewertung der architektonischen Eigenschaften, indem die wesentlichen Vor- und Nachteile des Pipeline-Stils diskutiert werden. Die Darstellung basiert auf etablierten Quellen der Softwarearchitektur-Literatur.

Inhaltsverzeichnis

1 Einführung und Kernkonzepte **2**
- 1.1 Filter: Die Verarbeitungseinheiten . . . 2
- 1.2 Pipes: Die Kommunikationskanäle . . . 3
- 1.3 Stärke durch Komposition: Das Unix-Beispiel . . . 3

2 Anwendungsbereiche und Beispiele **3**
- 2.1 Beispiel: Verarbeitung von Telemetriedaten mit Kafka . . . 4

3 Bewertung: Vor- und Nachteile **5**
- 3.1 Vorteile . . . 5
- 3.2 Nachteile . . . 6
- 3.3 Fazit zur Bewertung . . . 6

4 Praktische Umsetzung: Beispiel einer Datenverarbeitungs-Pipeline **7**
- 4.1 Gesamtarchitektur . . . 7
- 4.2 Implementierung der Pipeline-Steuerung: Die Wrapper-Klasse . . . 8
- 4.3 Factory Pattern zur dynamischen Komponentenauswahl . . . 9
  - 4.3.1 Anwendung im Datenlader . . . 9
  - 4.3.2 Vorteile des Factory Patterns in der Pipeline . . . 10

1 Einführung und Kernkonzepte

In der Welt der Softwarearchitektur existieren diverse etablierte Muster und Stile, die als Blaupausen für die Strukturierung von Softwaresystemen dienen. Einer dieser fundamentalen und weit verbreiteten Stile ist die Pipeline-Architektur, oft auch als Pipes-and-Filters-Architektur bezeichnet [1]. Dieser Stil entstand als natürliche Konsequenz der Modularisierung von Anwendungsfunktionalität in eigenständige Verarbeitungsschritte [1]. Seine Prinzipien sind vielen Entwicklern und Architekten intuitiv vertraut, insbesondere durch die Kommandozeilen-Interpreter von Unix-basierten Betriebssystemen wie Bash oder Zsh, wo Befehle über Pipes (‘|‘) miteinander verkettet werden [1]. Auch in der funktionalen Programmierung finden sich Parallelen, da viele Sprachelemente die Komposition von Funktionen fördern, was dem Konzept der Pipeline ähnelt [1]. Moderne Datenverarbeitungsparadigmen wie MapReduce bauen ebenfalls auf dieser Grundtopologie auf [1]. Es ist hervorzuheben, dass der Pipeline-Stil nicht auf Low-Level-Systemprogrammierung beschränkt ist, sondern ebenso effektiv für die Gestaltung komplexer Geschäftsanwendungen eingesetzt werden kann [1].

Die Architektur basiert auf zwei zentralen Komponenten: **Filtern** (Filters) und **Pipes** [1].

1.1 Filter: Die Verarbeitungseinheiten

Filter repräsentieren die aktiven Bausteine der Architektur, die für die eigentliche Datenverarbeitung zuständig sind [1]. Charakteristisch für Filter ist ihre Eigenständigkeit und Unabhängigkeit voneinander. Idealerweise sind sie zustandslos (‘stateless‘), was bedeutet, dass das Ergebnis einer Verarbeitung ausschließlich von den aktuellen Eingabedaten abhängt und nicht von vorherigen Interaktionen [1]. Ein Kernprinzip des Designs ist die *Single Responsibility*: Jeder Filter sollte genau eine klar definierte Aufgabe erfüllen [1]. Steht eine komplexere Verarbeitungslogik an, wird diese nicht in einem einzigen, monolithischen Filter implementiert, sondern auf eine Sequenz spezialisierter Filter verteilt, die nacheinander ausgeführt werden [1]. Diese Granularität fördert die Wiederverwendbarkeit und Testbarkeit der einzelnen Komponenten.

Innerhalb der Pipeline-Architektur unterscheidet man typischerweise vier Arten von Filtern [1]:

- **Producer (Erzeuger oder Quelle):** Dieser Filtertyp steht am Anfang einer Pipeline und ist für die Generierung der initialen Daten verantwortlich. Er empfängt keine Daten von einem vorherigen Filter, sondern agiert als Datenquelle, z.B. durch Lesen aus einer Datei, einer Datenbank oder einem Netzwerk-Socket. Er wird daher auch als »Source« bezeichnet [1].
- **Transformer (Umwandler):** Ein Transformer nimmt Daten von einer Pipe entgegen, führt eine Transformation darauf aus (z.B. Daten anreichern, Format ändern, Werte berechnen) und leitet das modifizierte Ergebnis an die nächste Pipe weiter. Dieser Filtertyp verkörpert das zentrale Verarbeitungselement in vielen Pipelines. Funktionale Programmierer erkennen hier eine Analogie zur ‘map‘-Operation [1].
- **Tester (Filter im engeren Sinne):** Dieser Filter empfängt ebenfalls Daten, prüft diese jedoch anhand bestimmter Kriterien. Abhängig vom Ergebnis dieser Prüfung leitet er die Daten entweder unverändert weiter, verwirft sie oder leitet sie möglicherweise auf unterschiedliche Ausgänge (obwohl dies die klassische lineare Pipeline erweitert). Er selektiert also Datenströme. Funktionale Programmierer mögen hier an ‘filter‘- oder ‘reduce‘-Operationen denken, je nach spezifischer Implementierung [1].
- **Consumer (Verbraucher oder Senke):** Der Consumer bildet das Ende einer Pipeline. Er nimmt die final verarbeiteten Daten entgegen und führt eine abschließende Aktion durch, wie das Speichern der Daten in einer Datenbank, das Anzeigen auf einer Benutzeroberfläche oder das Senden an ein externes System. Er wird auch als »Sink« bezeichnet [1].

1.2 Pipes: Die Kommunikationskanäle

Pipes stellen die Verbindungen zwischen den Filtern dar und definieren den Datenfluss durch die Architektur [1]. Sie fungieren als gerichtete Kommunikationskanäle, die typischerweise genau einen Filterausgang mit genau einem Filtereingang verbinden (Punkt-zu-Punkt-Verbindung). Dies unterscheidet sie von Broadcast-Mechanismen, bei denen Daten an mehrere Empfänger gleichzeitig gesendet werden, was in der Regel aus Performancegründen vermieden wird [1]. Eine Pipe nimmt die Ausgabe eines Filters entgegen und leitet sie als Eingabe an den nächsten Filter weiter [1].

Die Daten, die als Nutzlast (‘payload‘) durch die Pipes transportiert werden, können prinzipiell jedes beliebige Format haben – von einfachen Textströmen über strukturierte Daten wie XML oder JSON bis hin zu binären Objekten [1]. Für eine hohe Performance und geringe Kopplung zwischen den Filtern wird jedoch oft empfohlen, das Datenformat möglichst standardisiert und die Datenmenge pro Nachricht relativ klein zu halten [1]. Die Implementierung von Pipes kann vielfältig sein: von einfachen In-Memory-Warteschlangen in monolithischen Anwendungen bis hin zu komplexeren Message-Queues oder Netzwerkprotokollen in verteilten Systemen.

1.3 Stärke durch Komposition: Das Unix-Beispiel

Die wahre Stärke der Pipeline-Architektur liegt in der Komposition einfacher, wiederverwendbarer Bausteine zu komplexen Verarbeitungsketten. Die Unidirektionalität der Pipes und die Fokussierung der Filter auf einzelne Aufgaben machen sie hochgradig modular und wiederverwendbar [1]. Dieses Prinzip ist vielen Entwicklern aus der Unix-Shell-Nutzung bestens bekannt [1].

Eine oft zitierte Anekdote illustriert eindrucksvoll die Eleganz und Mächtigkeit dieses Ansatzes [1]: Donald Knuth, ein renommierter Informatiker, wurde einst gebeten, ein Programm zu schreiben, das eine Textdatei einliest, die n häufigsten Wörter identifiziert und diese zusammen mit ihrer Häufigkeit sortiert ausgibt. Sein Ergebnis war ein über zehn Seiten langes, dokumentiertes Pascal-Programm, das einen eigens entwickelten Algorithmus implementierte [1]. Doug McIlroy, einer der Väter von Unix und Pipes, präsentierte daraufhin eine Lösung als einfaches Shell-Skript, das dieselbe Aufgabe löste – wesentlich kürzer, verständlicher und oft sogar performanter. Dieses Skript, das problemlos in eine kurze Nachricht passt, lautete:

```
tr -cs A-Za-z ’
textbackslashn’ < input.txt | tr A-Z a-z | sort | uniq -c | sort -rn | head
-n ${1:-10}
```

Hierbei wird der Text zunächst in einzelne Wörter zerlegt (‘tr‘), in Kleinbuchstaben umgewandelt (‘tr‘), alphabetisch sortiert (‘sort‘), Duplikate gezählt (‘uniq -c‘), nach Häufigkeit numerisch und absteigend sortiert (‘sort -rn‘) und schließlich die ersten n (hier standardmäßig 10) Zeilen ausgegeben (‘head‘). Jeder Befehl ist ein Filter, das Pipe-Symbol ‘|‘ repräsentiert die Pipe. Dieses Beispiel zeigt, wie mächtig die Kombination einfacher Werkzeuge sein kann, eine Komplexität, die selbst die Unix-Erfinder manchmal überraschte [1].

2 Anwendungsbereiche und Beispiele

Der Pipeline-Architekturstil ist aufgrund seiner Struktur und Eigenschaften für eine Vielzahl von Anwendungsfällen prädestiniert. Insbesondere eignet er sich hervorragend für Systeme, die eine sequentielle, unidirektionale Verarbeitung von Datenströmen erfordern [1]. Die klare Trennung der Verarbeitungsschritte und der definierte Datenfluss machen ihn robust und nachvollziehbar für bestimmte Problemklassen.

Typische Anwendungsbereiche umfassen [1]:

- **Elektronischer Datenaustausch (EDI):** Im Bereich EDI müssen oft Daten zwischen verschiedenen Formaten (z.B. unternehmensspezifische Formate, Industriestandards wie EDIFACT oder XML-Derivate) konvertiert werden. Pipeline-basierte Werkzeuge zerlegen diesen Prozess in Schritte wie Validierung, Mapping, Anreicherung und Formatkonvertierung, wobei jeder Schritt von einem spezialisierten Filter durchgeführt wird.

- **ETL-Prozesse (Extract, Transform, Load):** ETL-Werkzeuge, die Daten aus Quellsystemen extrahieren, transformieren und in Zielsysteme (oft Data Warehouses) laden, nutzen inhärent das Pipeline-Muster. Die Phasen Extraktion (Producer), Transformation (mehrere Transformer und Tester) und Laden (Consumer) bilden eine natürliche Pipeline. Die Architektur unterstützt hierbei sowohl den Datenfluss als auch die schrittweise Modifikation der Daten.

- **Orchestrierung und Mediation in Integrationsszenarien:** Werkzeuge wie Apache Camel oder Mule ESB verwenden oft Pipeline-Konzepte, um komplexe Integrationsprozesse und Geschäftsworkflows zu modellieren. Nachrichten oder Datenobjekte werden durch eine Kette von Verarbeitungsschritten (z.B. Protokolltransformation, Routing, Inhaltsanreicherung) geleitet, die als Filter in einer Pipeline konfiguriert sind.

- **Compiler-Konstruktion:** Klassische Compiler arbeiten oft nach einem Pipeline-Prinzip: Lexikalische Analyse -> Syntaktische Analyse -> Semantische Analyse -> Zwischencode-Erzeugung -> Optimierung -> Zielcode-Erzeugung. Jede Phase nimmt die Ausgabe der vorherigen als Eingabe und führt eine spezifische Transformation durch.

- **Verarbeitung von Streaming-Daten:** Systeme, die kontinuierliche Datenströme verarbeiten (z.B. Sensordaten, Log-Events, Finanztransaktionen), setzen häufig Pipelines ein. Frameworks wie Apache Kafka Streams, Flink oder Spark Streaming ermöglichen die Definition von Verarbeitungstopologien, die dem Pipes-and-Filters-Modell folgen.

2.1 Beispiel: Verarbeitung von Telemetriedaten mit Kafka

Ein anschauliches Beispiel für die Anwendung der Pipeline-Architektur ist die Verarbeitung von Telemetriedaten, die von verteilten Diensten erzeugt und über Apache Kafka gestreamt werden [1]. Ziel könnte sein, aus dem rohen Datenstrom spezifische Metriken wie Antwortzeiten und Verfügbarkeit zu extrahieren und zu aggregieren.

Eine mögliche Pipeline könnte wie folgt aufgebaut sein [1]:

1. **Producer (‘ServiceInfoCapture‘):** Dieser Filter abonniert das relevante Kafka-Topic, in das die Rohdaten der Telemetrie (z.B. Logs oder Ereignisse über Serviceaufrufe) geschrieben werden. Er fungiert als Quelle der Pipeline und liest die Nachrichten aus Kafka. 2. **Tester (‘DurationFilter‘):** Die vom Producer gelesenen Daten werden an diesen Filter weitergeleitet. Seine Aufgabe ist es zu prüfen, ob die aktuelle Nachricht Informationen über die Dauer (Antwortzeit) eines Serviceaufrufs enthält (z.B. einen Messwert in Millisekunden). Dies illustriert die klare Trennung: Der ‘ServiceInfoCapture‘-Filter kümmert sich nur um das Lesen aus Kafka, der ‘DurationFilter‘ nur um die Identifikation relevanter Daten für diesen Zweig der Verarbeitung. 3. **Transformer (‘DurationCalculator‘):** Wenn der ‘DurationFilter‘ feststellt, dass die Daten die Antwortzeit betreffen, leitet er sie an diesen Filter weiter. Der ‘DurationCalculator‘ könnte beispielsweise die Zeit von Millisekunden in Sekunden umrechnen, statistische Berechnungen durchführen (z.B. gleitender Durchschnitt) oder die Daten für die spätere Aggregation vorbereiten. 4. **Tester (‘UptimeFilter‘):** Falls der ‘DurationFilter‘ die Nachricht nicht als relevant für die Antwortzeit einstuft, könnte sie alternativ an den ‘UptimeFilter‘ geleitet werden. Dieser prüft, ob die Nachricht stattdessen Informationen zur Verfügbarkeit oder zum Betriebsstatus des Dienstes enthält (z.B. Heartbeat-Signale, Fehlermeldungen). 5. **Transformer (‘UptimeCalculator‘):** Wenn der ‘UptimeFilter‘ Uptime-relevante Daten erkennt, werden diese an den

‘UptimeCalculator‘ übergeben. Dieser Filter könnte den Status interpretieren und daraus Verfügbarkeitsmetriken ableiten (z.B. Berechnung der Uptime-Prozentrate über ein Zeitfenster). 6. **Consumer (‘DatabaseOutput‘ oder ‘MetricsAggregator‘):** Die von ‘DurationCalculator‘ oder ‘UptimeCalculator‘ transformierten Metrikdaten werden schließlich an einen oder mehrere Consumer übergeben. Ein ‘DatabaseOutput‘-Filter könnte die berechneten Metriken in einer Zeitreihendatenbank (z.B. InfluxDB, MongoDB) speichern [1]. Ein anderer Consumer könnte die Metriken an ein Monitoring-System weiterleiten.

Daten, die weder vom ‘DurationFilter‘ noch vom ‘UptimeFilter‘ als relevant eingestuft werden, verlassen die Pipeline an dieser Stelle, da sie für den spezifischen Zweck dieser Verarbeitungskette nicht benötigt werden [1]. Dieses Beispiel demonstriert, wie durch die Kombination verschiedener Filtertypen ein komplexer Verarbeitungsworkflow strukturiert werden kann, wobei jeder Filter eine klar umrissene Aufgabe übernimmt und der Datenfluss explizit durch die Pipes definiert ist.

3 Bewertung: Vor- und Nachteile

Wie jeder Architekturstil weist auch die Pipeline-Architektur spezifische Stärken und Schwächen auf, die bei der Entscheidung für oder gegen ihre Anwendung in einem konkreten Projektkontext abgewogen werden müssen. Die Bewertung erfolgt anhand gängiger architektonischer Qualitätsmerkmale [1].

3.1 Vorteile

Die Pipeline-Architektur bietet eine Reihe signifikanter Vorteile, die sie für bestimmte Anwendungsfälle attraktiv machen:

- **Einfachheit und Verständlichkeit:** Der lineare Datenfluss und die klare Aufgabentrennung der Filter machen die Architektur relativ einfach zu verstehen und nachzuvollziehen. Dies reduziert die kognitive Last für Entwickler und erleichtert die Einarbeitung neuer Teammitglieder [1].
- **Modularität und Wiederverwendbarkeit:** Filter sind per Definition unabhängige Einheiten mit einer spezifischen Aufgabe. Sie können oft ohne Modifikation in verschiedenen Pipelines wiederverwendet werden. Die lose Kopplung über die Pipes erlaubt es, Filter auszutauschen, neu anzuordnen oder hinzuzufügen, ohne andere Filter direkt zu beeinflussen [1]. Dies fördert die Wartbarkeit und Flexibilität des Systems.
- **Technische Partitionierung:** Die Logik der Anwendung ist klar entlang der technischen Verarbeitungsschritte (Filtertypen wie Producer, Transformer, Tester, Consumer) aufgeteilt. Dies unterstützt eine strukturierte Entwicklung [1].
- **Flexibilität bei der Komposition:** Die Möglichkeit, komplexe Aufgaben durch die Verkettung einfacher Filter zu lösen, bietet eine hohe Flexibilität im Design von Verarbeitungsprozessen. Neue Anforderungen können oft durch Hinzufügen oder Austauschen von Filtern implementiert werden.
- **Natürliche Parallelisierbarkeit (potenziell):** Obwohl oft monolithisch implementiert, bietet das Modell prinzipiell Potenzial für Parallelisierung. Einzelne Filterinstanzen könnten parallel arbeiten (Task-Parallelität), oder die Pipeline selbst könnte auf mehrere Datenströme parallel angewendet werden (Daten-Parallelität), sofern die Filter zustandslos sind und die Pipes dies unterstützen (z.B. durch konkurrente Queues).

- **Geringe Gesamtkosten (in bestimmten Szenarien):** Durch die Einfachheit, Verständlichkeit und Wiederverwendbarkeit können die Entwicklungs- und Wartungskosten, insbesondere für klar definierte sequentielle Prozesse, vergleichsweise niedrig sein [1].

3.2 Nachteile

Trotz ihrer Vorteile bringt die Pipeline-Architektur auch einige Nachteile und Einschränkungen mit sich, die insbesondere bei komplexen oder hochskalierbaren Systemen relevant werden:

- **Monolithisches Deployment (häufig):** In vielen Fällen wird eine Pipeline-Architektur als eine einzige Einheit (Monolith) bereitgestellt [1]. Das bedeutet, das gesamte System bildet ein einzelnes architektonisches Quantum. Änderungen an einem einzigen Filter erfordern potenziell das Testen und Deployen der gesamten Anwendung [1].
- **Eingeschränkte Skalierbarkeit und Elastizität:** Aufgrund des typischerweise monolithischen Deployments ist die Skalierbarkeit oft auf die Skalierung der gesamten Anwendung beschränkt. Es ist schwierig, nur einzelne, ressourcenintensive Filter gezielt zu skalieren, ohne komplexe interne Mechanismen wie Threading oder asynchrone Verarbeitung einzuführen, für die der Stil nicht primär ausgelegt ist [1]. Die horizontale Skalierung des gesamten Monolithen kann ineffizient sein, wenn nur ein kleiner Teil der Pipeline der Flaschenhals ist. Dies führt zu einer niedrigen Bewertung bei Elastizität und Skalierbarkeit (oft nur ein Stern) [1].
- **Geringe Fehlertoleranz und Verfügbarkeit:** In einer monolithischen Implementierung kann der Ausfall eines kritischen Filters (z.B. durch einen Fehler oder Ressourcenmangel wie Speicher) die gesamte Pipeline oder sogar die gesamte Anwendung zum Stillstand bringen [1]. Da es nur ein Deployment-Quantum gibt, fehlt die Isolation, die verteilte Systeme bieten. Die durchschnittliche Wiederherstellungszeit (MTTR) kann bei Monolithen relativ hoch sein (Minuten bis Stunden), was die Verfügbarkeit beeinträchtigt [1].
- **Mittelmäßige Testbarkeit und Bereitstellbarkeit:** Obwohl einzelne Filter gut isoliert getestet werden können (Unit-Tests), erfordert jede Änderung am Code oft einen vollständigen Integrationstest und die erneute Bereitstellung des gesamten Monolithen. Dies verlangsamt den Entwicklungszyklus und erhöht das Risiko bei Deployments im Vergleich zu feingranulareren Architekturen wie Microservices [1]. Die Bewertung für Bereitstellbarkeit (‘Deployability‘) und Testbarkeit ist daher oft nur durchschnittlich [1].
- **Potenzielle Latenzprobleme:** In langen Pipelines kann sich die Latenz summieren, da Daten sequenziell durch viele Filter fließen müssen. Wenn die Pipes zudem I/O-intensiv implementiert sind (z.B. über Festplatten-basierte Queues statt In-Memory), kann dies die End-to-End-Verzögerung weiter erhöhen.
- **Gemeinsames Datenformat als Herausforderung:** Die Notwendigkeit eines gemeinsamen Datenformats für die Pipes kann einschränkend sein oder zu Overhead führen, wenn Filter sehr unterschiedliche Datenrepräsentationen bevorzugen würden. Änderungen am Datenformat können Anpassungen in vielen Filtern erfordern.
- **Schwierigkeiten bei komplexen Kontrollflüssen:** Der Stil eignet sich am besten für lineare Abläufe. Komplexe Verzweigungen, Zusammenführungen oder iterative Prozesse können schwierig und unübersichtlich in einer reinen Pipeline-Struktur abzubilden sein.

3.3 Fazit zur Bewertung

Zusammenfassend lässt sich sagen, dass die Pipeline-Architektur ein eleganter und effektiver Stil für Probleme ist, die sich gut als sequentielle Kette von Verarbeitungsschritten modellieren

lassen. Ihre Stärken liegen in der Einfachheit, Modularität und Wiederverwendbarkeit der Komponenten, was oft zu kosteneffizienten Lösungen führt [1]. Die größten Herausforderungen ergeben sich typischerweise aus der oft monolithischen Natur der Implementierung, was zu Einschränkungen bei Skalierbarkeit, Fehlertoleranz, Testbarkeit und Bereitstellbarkeit führt, insbesondere bei großen, komplexen oder hochverfügbaren Systemen [1]. Die Wahl dieses Stils sollte daher sorgfältig gegen die spezifischen Anforderungen des Systems abgewogen werden.

4 Praktische Umsetzung: Beispiel einer Datenverarbeitungs-Pipeline

4.1 Gesamtarchitektur

Um die Konzepte der Pipeline-Architektur zu verdeutlichen, betrachten wir im Folgenden eine beispielhafte Datenverarbeitungs-Pipeline. Diese Pipeline ist darauf ausgelegt, Sensordaten zu laden, zu verarbeiten und schließlich zu speichern. Der Startpunkt der Verarbeitung ist das Laden der Daten aus einer Datei.

Die grundlegende Architektur dieser Beispiel-Pipeline ist in Abbildung 1 schematisch dargestellt. Sie visualisiert den sequenziellen Fluss der Daten durch die einzelnen Verarbeitungsstufen.

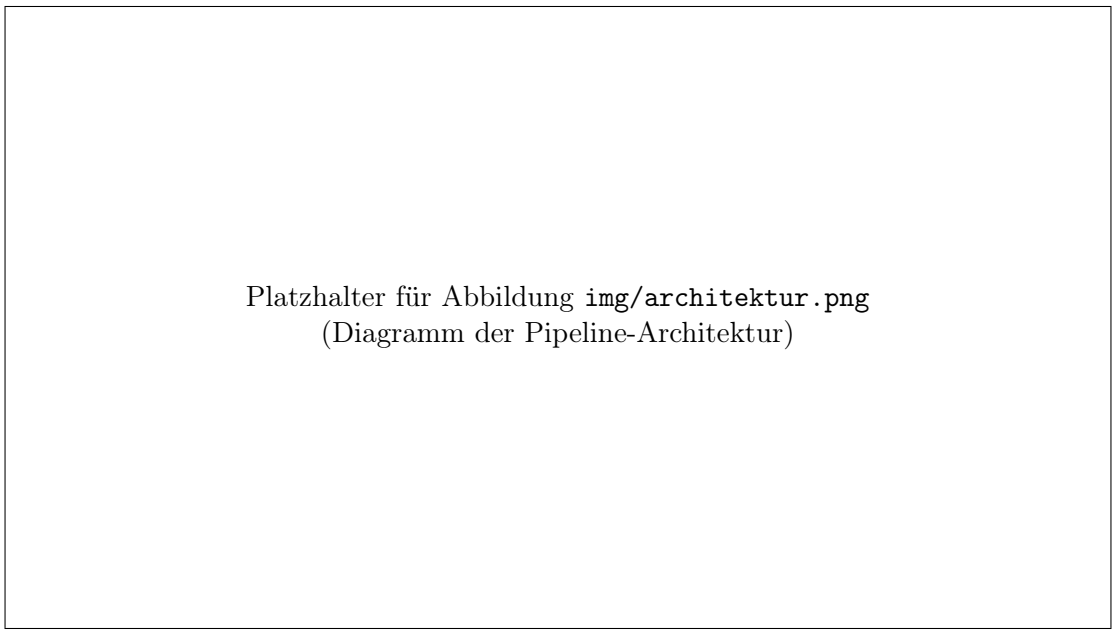

Abbildung 1: Schematische Darstellung der Datenverarbeitungs-Pipeline mit ihren Hauptstufen.

Die Pipeline lässt sich in die folgenden logischen Stufen unterteilen, die nacheinander durchlaufen werden:

1. **Datenerfassung (Laden):** In dieser ersten Stufe werden die Rohdaten von der Datenquelle (z.B. einer Sensordatei) eingelesen. Dies beinhaltet das Öffnen der Datei und das Parsen des Inhalts in eine interne Datenstruktur.

2. **Vorverarbeitung:** Die geladenen Rohdaten werden bereinigt und gefiltert. Dies kann Schritte wie das Entfernen von Ausreißern, das Behandeln fehlender Werte oder das Glätten von Signalen umfassen.

3. **Anreicherung:** Die vorverarbeiteten Daten werden mit zusätzlichen Kontextinformationen angereichert. Beispiele hierfür sind das Hinzufügen von Zeitstempeln, geografischen Koordinaten oder Metadaten zur Datenquelle.

4. **Analyse:** In dieser Stufe erfolgt die eigentliche fachliche Verarbeitung. Dies kann spezifische Berechnungen, die Anwendung von Algorithmen zur Mustererkennung oder die Extraktion relevanter Merkmale beinhalten.

5. **Speicherung/Ausgabe:** Die verarbeiteten Ergebnisse werden persistiert (z.B. in einer Datenbank oder Datei gespeichert) oder an nachfolgende Systeme zur weiteren Verwendung oder Visualisierung übergeben.

Jede dieser Stufen repräsentiert eine logische Einheit innerhalb der Pipeline. Wie diese Stufen technisch implementiert und zu einer funktionierenden Pipeline zusammengesetzt werden, wird in den folgenden Abschnitten detailliert erläutert, beginnend mit der Implementierung der Pipeline-Steuerung.

4.2 Implementierung der Pipeline-Steuerung: Die Wrapper-Klasse

Nachdem die grundlegende Architektur und die einzelnen Stufen definiert sind, bedarf es einer Komponente, welche die Pipeline als Ganzes verwaltet und deren Ausführung steuert. Hierfür gibt es verschiedene Ansätze:

- **Direkte Implementierung im Code:** Die Abfolge der Stufen wird direkt im ausführenden Code definiert. Dies ist einfach für kleine Pipelines, kann aber bei wachsender Komplexität unübersichtlich und schwer wartbar werden.
- **Konfigurationsbasierter Ansatz:** Die Pipeline-Struktur wird in einer externen Datei (z.B. YAML, JSON) definiert und zur Laufzeit geladen. Dies bietet hohe Flexibilität und ermöglicht Anpassungen ohne Code-Änderungen, erfordert jedoch zusätzlichen Aufwand für das Parsen und Validieren der Konfiguration.
- **Wrapper-Klasse:** Eine dedizierte Klasse kapselt die Logik zur Verwaltung der Stufen (Hinzufügen, Reihenfolge) und zur Ausführung der Pipeline. Dies fördert die Struktur, Kapselung und Wiederverwendbarkeit.

Für dieses Beispiel wird der Ansatz der Wrapper-Klasse gewählt, da er eine gute Balance zwischen Struktur, Flexibilität und Implementierungsaufwand bietet. Die Kernidee ist eine Klasse (z.B. ‘Pipeline‘), die eine Liste von Pipeline-Stufen verwaltet und eine Methode zur Ausführung der gesamten Sequenz bereitstellt. Abbildung 2 zeigt ein Klassendiagramm für eine solche Wrapper-Klasse.

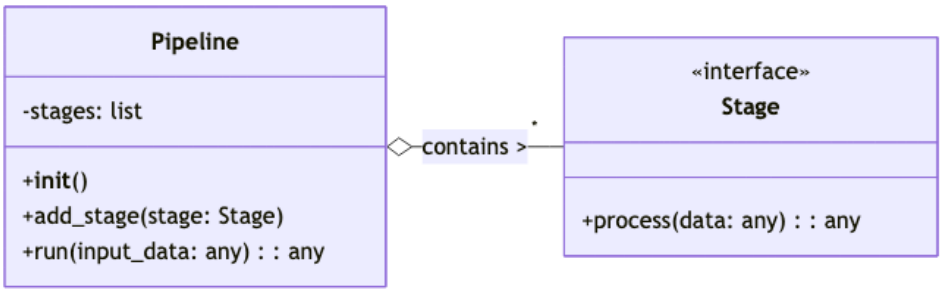

Abbildung 2: Klassendiagramm der Pipeline Wrapper-Klasse (‘Pipeline‘) und der abstrakten Stufenklasse (‘Stage‘).

Die ‘Pipeline‘-Klasse könnte typischerweise Methoden wie ‘__init__‘ (zur Initialisierung, z.B. einer leeren Liste für die Stufen), ‘add_stage‘ (zum Hinzufügen einer Stufe zur Pipeline) und ‘execute‘ oder ‘run‘ (zum Durchlaufen aller Stufen mit den Eingabedaten) besitzen.

Ein zentrales Element dieses Ansatzes ist die Verwendung einer abstrakten Basisklasse oder einer Schnittstelle für die einzelnen Pipeline-Stufen (im Diagramm und folgenden Codebeispiel als ‘Stage‘ bezeichnet). Jede konkrete Stufe (wie Datenerfassung, Vorverarbeitung etc.) muss von dieser Basisklasse erben und deren definierte Methoden (z.B. eine ‘process‘ oder ‘execute‘-Methode) implementieren. Dies stellt sicher, dass alle Stufen einheitlich behandelt und von der ‘Pipeline‘-Klasse aufgerufen werden können.

Die Wrapper-Klasse kann zudem sicherstellen, dass nur valide Stufen hinzugefügt werden. Dies wird oft durch Typ-Prüfungen realisiert, wie im folgenden Python-Codeausschnitt für die ‘add_stage‘-Methode gezeigt:

```
# Innerhalb der Pipeline-Wrapper-Klasse
def add_stage(self, stage: Stage):
    """Fügt eine neue Stufe zur Pipeline hinzu."""
    if not isinstance(stage, Stage): # Prüft, ob das Objekt eine Instanz von Stage oder einer
        raise TypeError("Übergebene Stufe muss eine Instanz von Stage sein.")
    self.stages.append(stage) # Annahme: self.stages ist die Liste der Stufen
```

Diese Prüfung mittels ‘isinstance(stage, Stage)‘ stellt sicher, dass jedes Objekt, das der Pipeline hinzugefügt wird, tatsächlich vom Typ ‘Stage‘ (oder einer davon abgeleiteten Klasse) ist und somit die erwartete Schnittstelle (z.B. eine ‘process‘-Methode) implementiert. Wird ein unpassendes Objekt übergeben, wird ein ‘TypeError‘ ausgelöst, was die Robustheit der Pipeline erhöht.

Die Kapselung der Pipeline-Logik in einer Wrapper-Klasse in Kombination mit einer abstrakten Basisklasse für die Stufen führt zu einem modularen, gut testbaren und erweiterbaren Design. Änderungen an einzelnen Stufen oder das Hinzufügen neuer Stufen beeinflussen die Wrapper-Klasse nur minimal, solange die definierte Schnittstelle eingehalten wird.

```
def add_stage(self, stage:Stage):
    if(not issubclass(stage, Stage)):
        raise TypeError("stage must be an instance of Stage")
```

Diese Sicherung wird in Python durch das Prüfen des Klassentyps (issubclass) sichergestellt. Wird eine Klasse übergeben welche keine Subklasse von Stage ist wird ein TypeError geworfen. Dies geschieht bei issubclass selbst bei einer erzeugten Instanz von Stage selbst.

4.3 Factory Pattern zur dynamischen Komponentenauswahl

Ein weiteres nützliches Entwurfsmuster, das häufig in Verbindung mit der Pipeline-Architektur eingesetzt wird, ist das Factory Pattern. Dieses Muster gehört zu den Erzeugungsmustern und dient dazu, Objekte zu erstellen, ohne die genaue Klasse des zu erstellenden Objekts im Voraus festlegen zu müssen. Stattdessen wird die Verantwortung für die Objekterzeugung an eine spezialisierte FactoryKlasse oder -Methode delegiert.

Das Factory Pattern ermöglicht eine signifikante Entkoppelung des aufrufenden Codes (in unserem Fall die Pipeline oder eine ihrer Stufen) von der konkreten Implementierung der zu erzeugenden Objekte. Die Factory fungiert als zentraler Punkt für die Erzeugung von Objekten eines bestimmten Typs oder einer bestimmten Schnittstelle. Abhängig von den übergebenen Parametern oder dem Kontext entscheidet die Factory, welche spezifische Unterklasse instanziiert und zurückgegeben werden soll.

4.3.1 Anwendung im Datenlader

In unserem Beispiel einer Datenverarbeitungs-Pipeline findet das Factory Pattern eine praktische Anwendung, insbesondere bei der Funktion zum Auslesen der Sensordateien. Unterschiedliche

Sensoren oder Datenquellen können ihre Daten in verschiedenen Formaten bereitstellen (z.B. CSV, JSON, XML, Binärformate). Anstatt die Logik zur Erkennung und Verarbeitung jedes Formats direkt in die Pipeline-Stufe einzubetten, die die Daten lädt, wird eine ‘LoaderFactory‘ eingesetzt.

Der Ablauf ist typischerweise wie folgt:

1. Eine vorherige Pipeline-Stufe oder der Initialaufruf übergibt den Dateipfad oder eine andere Kennung der zu ladenden Daten an die Stufe, die für das Laden zuständig ist.
2. Diese Ladestufe delegiert die Erzeugung des passenden Ladeobjekts an die ‘LoaderFactory‘. Sie übergibt dabei relevante Informationen, wie z.B. den Dateipfad oder explizit das Dateiformat.
3. Die ‘LoaderFactory‘ analysiert die übergebenen Informationen (z.B. die Dateiendung wie ‘.csv‘ oder ‘.json‘).
4. Basierend auf dieser Analyse instanziiert die Factory ein konkretes Ladeobjekt (z.B. eine Instanz von ‘CsvLoader‘ oder ‘JsonLoader‘), das eine gemeinsame Schnittstelle (z.B. ‘DataLoader‘) implementiert.
5. Die Factory gibt das erzeugte Ladeobjekt an die aufrufende Pipeline-Stufe zurück.
6. Die Pipeline-Stufe verwendet nun das erhaltene Ladeobjekt, um die Daten zu laden, ohne die spezifische Implementierung des Laders kennen zu müssen. Sie interagiert nur über die definierte Schnittstelle (‘DataLoader‘).

Abbildung 3 illustriert diesen Entscheidungsprozess innerhalb der Factory. Die Factory prüft das angeforderte Format und wählt den entsprechenden Loader aus. Sollte kein passender Loader für das angeforderte Format registriert sein oder die Datei nicht existieren, wird typischerweise ein Fehler ausgelöst oder ein Null-Objekt zurückgegeben, um das Problem zu signalisieren.

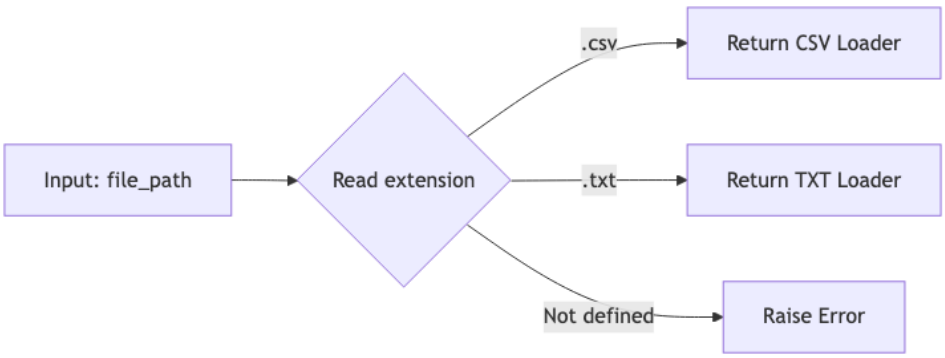

Abbildung 3: Entscheidungsdiagramm der LoaderFactory zur Auswahl des passenden Datenladers basierend auf dem Dateiformat.

4.3.2 Vorteile des Factory Patterns in der Pipeline

Der Einsatz des Factory Patterns in diesem Kontext bietet mehrere signifikante Vorteile:

- **Entkoppelung:** Der Code der Pipeline-Stufe, die Daten lädt, ist von den konkreten Ladeimplementierungen entkoppelt. Er interagiert nur mit der Factory und der abstrakten ‘DataLoader‘-Schnittstelle.

- **Flexibilität und Erweiterbarkeit (Open/Closed Principle):** Das Hinzufügen der Unterstützung für neue Dateiformate wird erheblich vereinfacht. Es muss lediglich eine neue 'DataLoader'-Unterklasse (z.B. 'XmlLoader') implementiert und die 'LoaderFactory' entsprechend erweitert werden, um diese neue Klasse bei Bedarf zu instanziieren. Der Code der Pipeline-Stufe selbst muss nicht modifiziert werden. Das System ist offen für Erweiterungen, aber geschlossen für Modifikationen.
- **Zentralisierung der Erzeugungslogik:** Die Logik zur Entscheidung, welcher Loader wann erstellt wird, ist an einem einzigen Ort – der Factory – konzentriert. Dies verbessert die Wartbarkeit und Übersichtlichkeit des Codes. Änderungen an der Erzeugungslogik müssen nur an dieser zentralen Stelle vorgenommen werden.
- **Verbesserte Testbarkeit:** Die einzelnen Loader-Klassen sowie die Factory selbst können isoliert getestet werden.
- **Wiederverwendbarkeit:** Die Factory und die Loader-Klassen können potenziell auch in anderen Teilen der Anwendung oder in anderen Projekten wiederverwendet werden. Wie bereits erwähnt, ist dies besonders nützlich, wenn die Ladefunktionalität als Teil eines wiederverwendbaren Moduls, Plugins oder Pakets (Pip Package) bereitgestellt wird. Entwickler, die das Paket nutzen, können eigene Loader hinzufügen und über die Factory registrieren (falls die Factory dies unterstützt), ohne den Kerncode des Pakets ändern zu müssen.

Zusammenfassend lässt sich sagen, dass das Factory Pattern die Struktur der Pipeline verbessert, indem es die Erzeugung von Komponenten wie Datenladern flexibler, wartbarer und erweiterbarer gestaltet. Es fördert ein sauberes Design, indem es Abhängigkeiten reduziert und Verantwortlichkeiten klar trennt.

Literatur

[1] Mark Richards and Neal Ford. *Handbuch Moderner Softwarearchitektur: Architekturstile, Patterns und Best Practices.* O'Reilly Verlag GmbH & Co. KG, 2020. Zugriff über ProQuest Ebook Central.